Table of contents

关于翻译 9
Kotlin 入门 11
欢迎参加我们的 Kotlin 观光之旅! 16
Hello world 18
基本类型 21
集合(Collection) 24
控制流 36
函数 47
类 60
Null 值安全性 69
Kotlin Multiplatform 74
使用 Kotlin 进行服务器端开发 78
使用 Kotlin 进行 Android 开发 81
使用 Kotlin 进行 Wasm 开发 83
使用 Kotlin/Native 进行原生(Native)程序开发 87
使用 Kotlin 进行 JavaScript 开发 90
在数据科学(Data Science)中使用 Kotlin 92
在编程竞赛(Competitive Programming)中使用 Kotlin 98
Kotlin 1.9.20 版中的新功能 105
Kotlin 1.9.0 版中的新功能 134
Kotlin 2.0.0-Beta5 版中的新功能 167
Kotlin 1.8.20 版中的新功能 168
Kotlin 1.8.0 版中的新功能 204
Kotlin 1.7.20 版中的新功能 227
Kotlin 1.7.0 版中的新功能 247
Kotlin 1.6.20 版中的新功能 274
Kotlin 1.6.0 版中的新功能 299
Kotlin 1.5.30 版中的新功能 320
Kotlin 1.5.20 版中的新功能 347
Kotlin 1.5.0 版中的新功能 355
Kotlin 1.4.30 版中的新功能 377
Kotlin 1.4.20 版中的新功能 388
Kotlin 1.4.0 版中的新功能 398
Kotlin 1.3 版中的新功能 439

Kotlin 1.2 版中的新功能 451
Kotlin 1.1 版中的新功能 463
Kotlin 的发布版本 482
Kotlin 发展路线图 495
基本语法 496
惯用法 511
编码规约 520
基本类型 546
数值类型 547
无符号整数(Unsigned Integer)类型 556
布尔(Boolean)类型 559
字符 561
字符串 563
数组 568
类型检查与类型转换 578
条件与循环 582
返回与跳转: break 与 continue 589
异常(Exception) 593
包(Package)与导入(Import) 596
类 599
继承 605
属性(Property) 611
接口(Interface) 617
函数式 (SAM) 接口 620
可见度修饰符 623
扩展 627
数据类(Data Class) 634
封闭类(Sealed Class)与封闭接口(Sealed Interface) 637
泛型(Generic): in, out, where 647
嵌套类与内部类 660
枚举类 662
内联的值类(Inline value class) 666
对象表达式,对象声明,以及同伴对象 672
委托 681
委托属性 684
类型别名 697
函数 699

高阶函数与 Lambda 表达式 709
内联函数(Inline Function) 719
操作符重载 725
类型安全的构建器 732
通过构建器类型推断(Builder Type Inference)使用构建器 741
Null 值安全性 748
相等判断 754
this 表达式 758
异步编程(Asynchronous Programming)技术 760
协程(Coroutine) 765
注解 767
解构声明 776
反射 779
Kotlin 跨平台程序开发入门 788
Kotlin Multiplatform 项目结构的基础知识 790
跨平台项目结构的高级概念 801
为 Kotlin Multiplatform 设置编译目标 811
在不同的平台之间共用代码 813
预期声明与实际声明 816
层级项目结构 829
添加跨平台库依赖项 838
添加 Android 依赖项 846
添加 iOS 依赖项 848
配置编译任务 855
[实验性 DSL] 构建最终的原生二进制文件 865
构建最终的原生二进制文件 874
发布跨平台的库 887
如何向你的开发团队介绍跨平台移动开发 893
跨平台程序的 Gradle DSL 参考文档 894
Android 源代码集布局 919
Kotlin Multiplatform 兼容性指南 923
Kotlin Multiplatform Mobile Plugin 的发布版本 924
Kotlin/JVM 入门 935
与 Java 比较 940
在 Kotlin 中调用 Java 代码 943
在 Java 中调用 Kotlin 代码 966
Spring Boot 和 Kotlin 入门 981

使用 Kotlin 创建 Spring Boot 项目 983
向 Spring Boot 项目添加数据类 995
为 Spring Boot 项目添加数据库支持 1000
使用 Spring Data CrudRepository 进行数据库访问 1013
教程 - 在 JVM 平台使用 JUnit 进行代码测试 1018
教程 - 在同一个项目中混合使用 Java 和 Kotlin 1024
在 Kotlin 中使用 Java 记录类(Record) 1027
Java 和 Kotlin 中的字符串 1030
Java 和 Kotlin 中的集合(Collection) 1039
Java 和 Kotlin 中的可空性(Nullability) 1061
入门 1069
可读性 1071
可预测性 1078
可调试性 1084
向后兼容性(Backward Compatibility) 1088
Kotlin/Native 开发入门 - 使用 IntelliJ IDEA 1098
Kotlin/Native 开发入门 - 使用 Gradle 1103
Kotlin/Native 开发入门 - 使用命令行编译器 1107
与 C 代码交互 1109
教程 - 映射 C 语言的基本数据类型 1122
教程 - 映射 C 语言的结构(Struct)和联合(Union)类型 1129
教程 - 映射 C 语言的函数指针(Function Pointer) 1139
教程 - 映射 C 语言的字符串 1146
教程 - 使用 C Interop 和 libcurl 创建应用程序 1154
与 Swift/Objective-C 代码交互 1161
教程 - 使用 Kotlin/Native 开发 Apple Framework 1176
CocoaPods 概述与设置 1189
添加 Pod 库依赖项 1197
将 Kotlin Gradle 项目用作 CocoaPods 依赖项 1205
CocoaPods Gradle plugin DSL 参考文档 1209
Kotlin/Native 库 1215
平台库 1220
教程 - 使用 Kotlin/Native 开发动态库 1221
Kotlin/Native 内存管理 1232
与 iOS 集成 1237
迁移到新的内存管理器 1244
调试 Kotlin/Native 代码 1249

符号化(Symbolicate) iOS 崩溃报告(Crash Report) 1256
Kotlin/Native 支持的目标平台 1259
改进 Kotlin/Native 编译速度 1265
Kotlin/Native 二进制文件的许可证 1268
Kotlin/Native FAQ 1271
使用 IntelliJ IDEA 开发 Kotlin/Wasm 入门 1275
向 Kotlin/Wasm 项目添加 Kotlin 库依赖项 1285
与 JavaScript 交互 1288
问题分析 1299
创建 Kotlin/JS 工程(Project) 1301
运行 Kotlin/JS 代码 1323
开发服务器(Development server)与持续编译(Continuous Compilation) 1326
调试 Kotlin/JS 代码 1329
在 Kotlin/JS 平台进行测试 1336
JavaScript 死代码剔除工具 1342
使用 IR 编译器 1344
将 Kotlin/JS 项目迁移到 IR 编译器 1350
浏览器与 DOM API 1356
在 Kotlin 中使用 JavaScript 代码 1357
动态类型 1363
使用 npm 中的依赖项 1365
在 JavaScript 中使用 Kotlin 代码 1367
JavaScript 模块 1372
Kotlin/JS 的反射(Reflection) 1378
类型安全的 HTML DSL 1380
教程 - 使用 React 和 Kotlin/JS 创建 Web 应用程序 1382
教程 - Kotlin 自定义脚本(Custom Scripting) 入门 1422
集合(Collection)概述 1435
创建集合 1444
迭代器(Iterator) 1449
值范围(Range)与数列(Progression) 1453
序列(Sequence) 1457
集合操作概述 1463
集合变换操作 1467
过滤(Filtering)集合 1475
加法(Plus)和减法(Minus)操作符 1479
分组(Grouping) 1480

获取集合的一部分 1482
获取集合的单个元素 1487
排序(Ordering) 1493
聚合(Aggregate)操作 1498
集合写入操作 1504
List 相关操作 1508
Set 相关操作 1516
Map 相关操作 1519
明确要求使用者同意的功能(Opt-in Requirement) 1526
作用域函数(Scope Function) 1534
时间测量 1550
协程指南 1561
协程的基本概念 1563
教程 - 协程与通道(Channel) 1570
取消与超时 1617
挂起函数(Suspending Function)的组合 1630
协程上下文与派发器(Dispatcher) 1641
异步的数据流(Asynchronous Flow) 1659
通道(Channel) 1705
协程的异常处理 1720
共享的可变状态与并发 1733
选择表达式(Select expression) (实验性功能) 1742
教程 - 使用 IntelliJ IDEA 调试协程 1754
教程 - 使用 IntelliJ IDEA 调试 Kotlin 数据流(Flow) 1759
序列化 1767
Lincheck 指南 1772
使用 Lincheck 编写你的第一个测试 1774
压力测试与模型检查 1782
操作参数 1791
数据结构约束 1795
进度保证 1798
顺序规格 1803
关键字与操作符 1805
Gradle 1813
Gradle 与 Kotlin/JVM 入门 1815
配置 Gradle 项目 1822
Kotlin Gradle plugin 中的编译器选项 1851

Kotlin Gradle plugin 中的编译与缓存 1864
对 Gradle plugin 变体的支持 1879
Maven 1884
Ant 1895
介绍 1901
Dokka 入门 1902
Gradle 1905
Maven 1943
CLI 1960
HTML 1985
Markdown 1995
Javadoc 2000
Dokka Plugin 2004
模块文档 2012
支持 Kotlin 开发的 IDE 2014
迁移到 Kotlin 编码风格 2017
运行代码片段 2021
使用 TeamCity 对 Kotlin 项目进行持续集成(Continuous Integration) 2030
为 Kotlin 代码编写文档: KDoc 2034
Kotlin 与 OSGi 2038
Kotlin 命令行编译器 2041
Kotlin 编译器选项 2045
All-open 编译器插件 2054
No-arg 编译器插件 2059
SAM-with-receiver 编译器插件 2062
kapt 编译器插件 2065
Lombok 编译器插件 2076
Kotlin 符号处理(Kotlin Symbol Processing) API 2081
KSP 快速入门 2088
为什么使用 KSP 2097
KSP 示例程序 2100
KSP 如何将 Kotlin 代码组织为模型 2102
针对 Java 注解处理器开发者的参考文档 2104
增量式处理(Incremental Processing) 2120
多轮(Multiple Round)处理 2125
在 Kotlin Multiplatform 中使用 KSP 2128
在命令行运行 KSP 2130

KSP FAQ 2132
学习资料概述 2135
Kotlin Koan 2137
Kotlin 动手课程(hands-on) 2138
Kotlin 小技巧 2140
Kotlin 书籍 2143
使用 Kotlin 惯用法的 Advent of Code 2147
使用 JetBrains Academy plugin 学习 Kotlin 2157
使用 JetBrains Academy plugin 教授 Kotlin 2158
参加 Kotlin EAP 项目 2159
安装 Kotlin EAP Plugin 2161
针对 EAP 进行构建配置 2164
FAQ 2168
Kotlin 的演化 2174
Kotlin 各部分组件的稳定性 2180
Kotlin 各部分组件的稳定性 (1.4 版以前) . . . .
Kotlin 1.9 兼容性指南 2189
Kotlin 1.8 兼容性指南 2190
Kotlin 1.7.20 兼容性指南 2191
Kotlin 1.7 兼容性指南 2193
Kotlin 1.6 兼容性指南 2194
Kotlin 1.5 兼容性指南 2195
Kotlin 1.4 兼容性指南 2196
Kotlin 1.3 兼容性指南 2215
兼容模式 2227
跨平台移动应用程序开发 2228
原生(Native)应用程序开发与跨平台(cross-platform)移动应用程序开发: 如何选择? 2234
跨平台应用程序开发最流行的 6 种框架 2240
使用 Kotlin 参加 Google 编程夏令营 2024 2246
使用 Kotlin 参加 Google 编程夏令营 2023 2247
安全性 2248
Kotlin 文档 (PDF 格式) 2249
为 Kotlin 项目贡献代码 2250
KUG 指南 2253
Kotlin Night 指南 2255
Kotlin 品牌资产 2257

关于翻译

最终更新: 2024/09/10

本文是 Kotlin 语言参考文档的中文翻译版.

原文

网址: https://kotlinlang.org/docs/reference/ (https://kotlinlang.org/docs/reference/)

代码库: https://github.com/JetBrains/kotlin-web-site (https://github.com/JetBrains/kotlin-web-site)

中文翻译版

网址: https://kotlin.liying-cn.net/docs/reference_zh/ (https://kotlin.liying-cn.net/docs/reference_zh/)

代码库: https://github.com/LiYing2010/kotlin-web-site (https://github.com/LiYing2010/kotlin-web-site)

翻译者: 李 颖 liying.cn.2010@gmail.com (mailto:liying.cn.2010@gmail.com)

关于本文档的任何问题, 欢迎与译者联系.

更新历史

- 2024 年 09 月: 第 15 次更新
- 2024 年 03 月: 第 14 次更新
- 2023 年 04 月: 第 13 次更新
- 2022 年 09 月: 第 12 次更新
- 2022 年 01 月: 第 11 次更新
- 2020 年 12 月: 第 10 次更新
- 2020 年 09 月: 第 9 次更新
- 2019 年 03 月: 第 8 次更新
- 2018 年 12 月: 第 7 次更新

- 2018 年 09 月: 第 6 次更新
- 2018 年 02 月: 第 5 次更新
- 2017 年 10 月: 第 4 次更新
- 2017 年 02 月: 第 3 次更新
- 2016 年 09 月: 第 2 次更新
- 2016 年 04 月: 初版翻译

Kotlin 入门

最终更新: 2024/09/10

Kotlin 是一门现代而成熟的编程语言, 设计目标是让开发者更加快乐. 它简洁, 安全, 能够与 Java 及其他语言交互, 并提供了很多方法在多个目标平台之间重用代码, 以提高开发效率.

作为入门学习, 请参加我们的 Kotlin 之旅. 这个教程包含 Kotlin 编程语言的基础知识.

Start Kotlin tour →

开始 Kotlin 之旅

(欢迎参加我们的 Kotlin 观光之旅!)

安装 Kotlin

Kotlin 包含在 IntelliJ IDEA (https://www.jetbrains.com/idea/download/) 和 Android Studio (https://developer.android.com/studio) 每个发行版之内. 下载并安装这些 IDE 中的一个, 就可以开始使用 Kotlin 了.

使用 Kotlin 来创建强大的应用程序

后端应用程序

下面是开发 Kotlin 服务端应用程序的开始步骤.

1. **创建你的第一个后端应用程序:**
   - 要从零开始创建, 可以 使用 IntelliJ IDEA 项目向导创建基本的 JVM 应用程序 (Kotlin/JVM 入门).
   - 如果想要阅读更加复杂的示例, 请选择下面的框架来创建项目:

Spring	Ktor
一个成熟的框架群, 包含复杂的生态系统, 全世界数百万开发者广泛使用. • 使用 Spring Boot 创建 RESTful Web 服务 (Spring Boot 和 Kotlin 入门) • 使用 Spring Boot 和 Kotlin 创建 Web 应用程序 (https://spring.io/guides/tutorials/spring-boot-kotlin/) • 使用 Spring Boot, Kotlin 协程(Coroutine) 和 RSocket (https://spring.io/guides/tutorials/spring-webflux-kotlin-rsocket/)	一个轻量的框架, 面向那些希望自由进行架构决策的开发者. • 使用 Ktor 创建 HTTP API (https://ktor.io/docs/creating-http-apis.html) • 使用 Ktor 创建 WebSocket 聊天程序 (https://ktor.io/docs/creating-web-socket-chat.html) • 使用 Ktor 创建交互式网站 (https://ktor.io/docs/creating-interactive-website.html) • 发布 Kotlin 服务端应用程序: 在 Heroku 上使用 Ktor (https://ktor.io/docs/heroku.html)

2. **在你的应用程序中使用 Kotlin 和第三方库.** 详情请参见 向你的项目添加库和工具依赖项 ("配置依赖项" in "配置 Gradle 项目").
   - Kotlin 标准库 (https://kotlinlang.org/api/latest/jvm/stdlib/) 提供了大量有用的功能, 比如 集合(Collection) (集合(Collection)概述) 和 协程(Coroutine) (协程指南).
   - 请参见 用于 Kotlin 的第三方框架, 库, 和工具 (https://blog.jetbrains.com/kotlin/2020/11/server-side-development-with-kotlin-frameworks-and-libraries/).
3. **关于服务器端开发的更多资料:**
   - 如何编写你的第一个 unit test (教程 - 在 JVM 平台使用 JUnit 进行代码测试).
   - 如何在你的应用程序中混合使用 Kotlin 和 Java 代码 (教程 - 在同一个项目中混合使用 Java 和 Kotlin).
4. **加入 Kotlin 服务器端开发社区:**

- Slack: 首先 得到邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up), 然后加入 #getting-started (https://kotlinlang.slack.com/archives/C0B8MA7FA), #server (https://kotlinlang.slack.com/archives/C0B8RC352), #spring (https://kotlinlang.slack.com/archives/C0B8ZTWE4), 或 #ktor (https://kotlinlang.slack.com/archives/C0A974TJ9) 频道.
- StackOverflow: 订阅 "kotlin" (https://stackoverflow.com/questions/tagged/kotlin), "spring-kotlin" (https://stackoverflow.com/questions/tagged/spring-kotlin), 和 "ktor" (https://stackoverflow.com/questions/tagged/ktor) 标签.

5. **订阅 Kotlin 官方帐号**: Twitter (https://twitter.com/kotlin), Reddit (https://www.reddit.com/r/Kotlin/), 和 Youtube (https://www.youtube.com/channel/UCP7uiEZIqci43m22KDl0sNw), 不要错过重要的生态系统更新信息.

如果遇到任何困难和问题, 请向我们的 YouTrack Bug 追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提交报告.

跨平台应用程序

在这里你将会学习如何使用 Kotlin Multiplatform (https://kotlinlang.org/lp/multiplatform/) 来开发并改进你的跨平台应用程序.

1. **为跨平台开发设置环境 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-setup.html).**

2. **创建你的第一个 iOS 和 Android 应用程序:**

   - 要从零开始创建, 可以 使用 IntelliJ IDEA 的项目向导创建一个基本的跨平台应用程序 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-create-first-app.html).
   - 如果你有已经存在的 Android 应用程序, 并且希望将它变为跨平台应用程序, 请阅读教程 让你的 Android 应用程序在 iOS 上运行 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-integrate-in-existing-app.html).
   - 如果你想看看比较真实的示例程序, 请 clone 并查看既有的项目, 比如 使用 Ktor 和 SQLDelight 创建跨平台应用程序 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ktor-sqldelight.html) 教程 或者 示例项目

(https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-samples.html) 中的网络和数据存储项目.

3. **使用大量的跨平台库** 在共用模块中实现需要的业务逻辑. 详情请参见 添加依赖项 (添加跨平台库依赖项).

库	详情
Ktor	文档 (https://ktor.io/docs/client.html)
序列化	文档 (序列化) 与 示例 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ktor-sqldelight.html#create-an-application-data-model)
协程(Coroutine)	文档 (协程指南) 与 示例 (教程 - 协程与通道(Channel))
日期与时间	文档 (https://github.com/Kotlin/kotlinx-datetime#readme)
SQLDelight	第三方库. 文档 (https://cashapp.github.io/sqldelight/)

> ⚠ 在 社区开发的库列表 (https://libs.kmp.icerock.dev/) 中还能找到其他跨平台库.

4. **关于 Kotlin Multiplatform 的更多资料:**

- 关于 Kotlin 跨平台开发 (Kotlin 跨平台程序开发入门).
- 阅读 示例项目 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-samples.html).
- 发布跨平台的库 (发布跨平台的库).
- 使用 Kotlin Multiplatform 的真实案例: Netflix (https://netflixtechblog.com/netflix-android-and-ios-studio-apps-kotlin-multiplatform-d6d4d8d25d23), VMware (https://kotlinlang.org/lp/multiplatform/case-studies/vmware/), Yandex

(https://kotlinlang.org/lp/multiplatform/case-studies/yandex/), 以及 其他很多公司 (https://kotlinlang.org/lp/multiplatform/case-studies/).

5. **加入 Kotlin 跨平台开发社区:**

   - Slack: 首先得到邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up), 然后加入 #getting-started (https://kotlinlang.slack.com/archives/C0B8MA7FA) 和 #multiplatform (https://kotlinlang.slack.com/archives/C3PQML5NU) 频道.
   - StackOverflow: 订阅 "kotlin-multiplatform" (https://stackoverflow.com/questions/tagged/kotlin-multiplatform) 标签.

6. **订阅 Kotlin 官方帐号**: Twitter (https://twitter.com/kotlin), Reddit (https://www.reddit.com/r/Kotlin/), 和 Youtube (https://www.youtube.com/channel/UCP7uiEZIqci43m22KDl0sNw), 不要错过重要的生态系统更新信息.

如果遇到任何困难和问题, 请向我们的 YouTrack Bug 追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提交报告.

Android 应用程序

- 如果希望使用 Kotlin 进行 Android 开发, 请阅读 Google 的 Kotlin Android 开发入门教程 (https://developer.android.com/kotlin/get-started).
- 如果你是 Android 新手, 希望学习如何使用 Kotlin 创建应用程序, 请阅读 这个 Udacity 课程 (https://www.udacity.com/course/developing-android-apps-with-kotlin--ud9012).

订阅 Kotlin 官方帐号: Twitter (https://twitter.com/kotlin), Reddit (https://www.reddit.com/r/Kotlin/), 和 Youtube (https://www.youtube.com/channel/UCP7uiEZIqci43m22KDl0sNw), 不要错过重要的生态系统更新信息.

没有找到需要的资料吗?

如果你没有找到需要的资料, 或对本页面内容感到疑惑, 请向我们 反馈你的意见 (https://surveys.hotjar.com/d82e82b0-00d9-44a7-b793-0611bf6189df).

欢迎参加我们的 Kotlin 观光之旅!

最终更新: 2024/09/10

> 这个教程包含 Kotlin 编程语言的基础知识, 而且全部可以在你的浏览器中完成. 不需要安装任何软件.

这个教程的各章包括以下内容:

- **理论**, 通过示例介绍 Kotlin 语言的关键概念.
- **实践**, 通过习题测试你对学习内容的理解程度.
- **解决方案**, 供你参考.

在这个教程中, 你将会学到:

- 变量 (Hello world)
- 基本类型 (基本类型)
- 集合(Collection) (集合(Collection))
- 控制流 (控制流)
- 函数 (函数)
- 类 (类)
- Null 值安全性 (Null 值安全性)

为了保证最好的体验, 我们建议你按照顺序阅读这些章节. 但如果你愿意, 也可以选择阅读你希望的章节.

准备好开始了吗?

开始 Kotlin 之旅

(Hello world)

Hello world

1 **Hello world** 2 基本类型 (基本类型) 3 集合(Collection) (集合(Collection)) 4 控制流 (控制流) 5 函数 (函数) 6 类 (类) 7 Null 值安全性 (Null 值安全性)

最终更新: 2024/09/10

下面是一个简单的程序, 输出 "Hello, world!":

```
fun main() {
    println("Hello, world!")
    // 输出结果为 Hello, world!
}
```

在 Kotlin 中:

- `fun` 用来声明一个函数
- `main()` 函数是你的程序开始的位置
- 函数体写在大括号 `{}` 之内
- `println()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/println.html) 和 `print()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/print.html) 函数将它们的参数打印到标准输出

> 函数会在后面的各章中详细介绍. 在此之前, 所有的示例程序都使用 `main()` 函数.

变量

所有的程序都需要存储数据, 变量可以帮助你实现这个目的. 在 Kotlin 中, 你可以:

- 使用 `val`, 声明只读的变量
- 使用 `var`, 声明可变的变量

要为变量赋值, 请使用赋值操作符 `=`.

例如:

```
fun main() {
//sampleStart
    val popcorn = 5    // 有 5 盒爆米花
    val hotdog = 7     // 有 7 个热狗
    var customers = 10 // 队列中有 10 个客户

    // 有些客户离开了队列
    customers = 8
    println(customers)
    // 输出结果为 8
//sampleEnd
}
```

> 变量可以声明在 `main()` 函数之外, 在你的程序开始的地方. 使用这种方式声明的变量, 我们称之为声明在 **顶级(top level)** 范围中.

由于 `customers` 是可变的变量, 可以在变量声明之后对它重新赋值.

> 我们建议你默认将所有变量都声明为只读(`val`)变量. 只有在需要的时候才声明可变的(`var`)变量.

字符串模板

确定的知道变量内容如何打印到标准输出将会很有用处. 你可以使用 **字符串模板** 做到这一点. 你可以使用模板表达式来访问存储在变量和其它对象中的数据, 并将它们转换为字符串. 字符串值是包含在双引号 `"` 中的一串字符. 模板表达式总是以美元符号 `$` 作为起始.

要在模板表达式中计算一段代码的值, 请在美元符号 `$` 之后放置一对大括号 `{}`, 然后将代码放在大括号之内.

例如:

```
fun main() {
//sampleStart
    val customers = 10
    println("There are $customers customers")
```

```
    // 输出结果为 There are 10 customers

    println("There are ${customers + 1} customers")
    // 输出结果为 There are 11 customers
//sampleEnd
}
```

更多详情请参见 字符串模板 (字符串).

你会注意到, 上面的示例中没有为变量声明类型. Kotlin 自己会推断它的类型: `Int`. 这个教程会在 下一章 (基本类型) 中解释 Kotlin 各种不同的基本类型, 以及如何声明这些类型.

实际练习

习题

完成以下代码, 让程序打印 `"Mary is 20 years old"` 到标准输出:

```
fun main() {
    val name = "Mary"
    val age = 20
    // 在这里编写你的代码
}
```

```
fun main() {
    val name = "Mary"
    val age = 20
    println("$name is $age years old")
}
```

下一步

基本类型 (基本类型)

基本类型

① Hello world (Hello world) ② **基本类型** ③ 集合(Collection) (集合(Collection)) ④ 控制流 (控制流) ⑤ 函数 (函数) ⑥ 类 (类) ⑦ Null 值安全性 (Null 值安全性)

最终更新: 2024/09/10

在 Kotlin 中, 每个变量和数据结构都有一个数据类型. 数据类型很重要, 因为它告诉编译器你可以对这个变量或数据结构做什么样的操作. 也就是说, 这个变量或数据结构有什么函数和属性.

在上一章中, Kotlin 能够知道上一个示例程序中的 `customers` 的类型是: `Int` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-int/). Kotlin **推断** 数据类型的能力称为 **类型推断**. `customers` 被赋值了一个整数值. 根据这一点, Kotlin 推断 `customers` 拥有数值型的数据类型: `Int`. 结果是, 编译器知道你可以对 `customers` 执行算数操作:

```
fun main() {
//sampleStart
    var customers = 10

    // 有些客户离开了队列
    customers = 8

    customers = customers + 3 // 加法示例, 结果为: 11
    customers += 7            // 加法示例, 结果为: 18
    customers -= 3            // 减法示例, 结果为: 15
    customers *= 2            // 乘法示例, 结果为: 30
    customers /= 3            // 除法示例, 结果为: 10

    println(customers) // 输出结果为 10
//sampleEnd
}
```

> ⚠ `+=`, `-=`, `*=`, `/=`, 和 `%=` 是计算并赋值操作符(Augmented Assignment Operator). 详情请参见 计算并赋值 ("计算并赋值" in "操作符重载").

总的来说, Kotlin 有以下数据类型:

类别	基本类型
整数	`Byte`, `Short`, `Int`, `Long`
无符号整数	`UByte`, `UShort`, `UInt`, `ULong`
浮点数	`Float`, `Double`
布尔值	`Boolean`
字符	`Char`
字符串	`String`

关于基本类型和它们的属性, 详情请参见 基本类型 (基本类型).

有了这些知识之后, 你可以声明变量, 并初始化这些变量. 只要变量在第一次读取之前初始化, Kotlin 就能够正确处理这些变量.

要声明一个变量但不初始化, 请使用 `:` 来指定它的类型.

例如:

```
fun main() {
//sampleStart
    // 声明变量, 但不初始化
    val d: Int
    // 变量被初始化
    d = 3

    // 明确指定了变量类型, 而且初始化
    val e: String = "hello"

    // 可以读取变量, 因为已经它们初始化了
    println(d) // 输出结果为 3
    println(e) // 输出结果为 hello
//sampleEnd
}
```

现在你已经知道了如何声明基本类型, 下面我们来学习 集合(Collection) (集合(Collection)).

实际练习

习题

为每个变量明确声明正确的类型:

```
fun main() {
    val a = 1000
    val b = "log message"
    val c = 3.14
    val d = 100_000_000_000_000
    val e = false
    val f = '\n'
}
```

```
fun main() {
    val a: Int = 1000
    val b: String = "log message"
    val c: Double = 3.14
    val d: Long = 100_000_000_000
    val e: Boolean = false
    val f: Char = '\n'
}
```

下一步

集合(Collection) (集合(Collection))

集合(Collection)

① Hello world (Hello world) ② 基本类型 (基本类型) ③ **集合(Collection)** ④ 控制流 (控制流) ⑤ 函数 (函数) ⑥ 类 (类) ⑦ Null 值安全性 (Null 值安全性)

最终更新: 2024/09/10

在程序开发中, 能够将数据组织到数据结构中以供后续的处理, 这样的能力非常有用. 为了这样的目的, Kotlin 提供了集合.

Kotlin 有以下集合来组织数据元素:

集合类型	描述
List	有顺序的元素组成的集合
Set	唯一的、无顺序的元素组成的集合
Map	一组键值对(key-value pair), 其中键是唯一, 并且每个键对应到唯一的值

每个集合类型都可以是可变的, 或只读的.

List

列表按照元素添加的顺序保存它们, 而且允许重复的元素.

要创建一个只读的 List (`List` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/)), 请使用 `listOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/list-of.html) 函数.

要创建一个可变的 List (`MutableList` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list.html)), 请使用 `mutableListOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-list-of.html) 函数.

创建 List 时, Kotlin 可以推断它存储的元素类型. 如果要明确声明元素类型, 请在 List 的声明之后的尖括号 `<>` 中添加类型:

```kotlin
fun main() {
//sampleStart
    // 只读 List
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println(readOnlyShapes)
    // 输出结果为 [triangle, square, circle]

    // 可变的 List, 带有明确的类型声明
    val shapes: MutableList<String> = mutableListOf("triangle",
"square", "circle")
    println(shapes)
    // 输出结果为 [triangle, square, circle]
//sampleEnd
}
```

> ⚠ 为了防止无意中修改 List 的内容, 你可以将可变的 List 赋值给一个 `List`, 来得到它的一个只读的视图:
>
> ```kotlin
> val shapes: MutableList<String> = mutableListOf("triangle",
> "square", "circle")
> val shapesLocked: List<String> = shapes
> ```
>
> 这种操作也叫做 **类型变换(casting)**.

List 是有顺序的, 因此要访问 List 内的元素, 请使用 下标访问操作符 ("下标访问操作符" in "操作符重载") `[]`:

```kotlin
fun main() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("The first item in the list is: ${readOnlyShapes[0]}")
    // 输出结果为 The first item in the list is: triangle
//sampleEnd
}
```

要获取 List 中的第一个或最后一个元素, 请分别使用 `.first()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first.html) 和 `.last()`

(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last.html) 函数:

```
fun main() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("The first item in the list is:
${readOnlyShapes.first()}")
    // 输出结果为 The first item in the list is: triangle
//sampleEnd
}
```

.first() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first.html) 和 .last() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last.html) 函数是 **扩展** 函数. 要对一个对象调用扩展函数, 请在对象之后加上点号 ., 然后把函数名写在后面.

关于扩展函数, 更多详情请参见 扩展函数 ("扩展函数(Extension Function)" in "扩展"). 对于这篇向导而言, 你只需要知道如何调用它们就行了.

要得到 List 中元素的数量, 请使用 .count() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/count.html) 函数:

```
fun main() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("This list has ${readOnlyShapes.count()} items")
    // 输出结果为 This list has 3 items
//sampleEnd
}
```

要检查一个元素是否存在于 List 中, 请使用 in 操作符 ("in 操作符" in "操作符重载"):

```
fun main() {
//sampleStart
    val readOnlyShapes = listOf("triangle", "square", "circle")
    println("circle" in readOnlyShapes)
    // 输出结果为 true
```

```
//sampleEnd
}
```

要对可变 List 添加或删除元素, 请分别使用 .add() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/add.html) 和 .remove() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/remove.html) 函数:

```
fun main() {
//sampleStart
    val shapes: MutableList<String> = mutableListOf("triangle",
"square", "circle")
    // 向 List 添加 "pentagon"
    shapes.add("pentagon")
    println(shapes)
    // 输出结果为 [triangle, square, circle, pentagon]

    // 从 List 中删除第一个 "pentagon"
    shapes.remove("pentagon")
    println(shapes)
    // 输出结果为 [triangle, square, circle]
//sampleEnd
}
```

Set

List 包含有顺序的元素, 并且允许元素重复, Set 则是 **无顺序的**, 并且只保存 **唯一的** 元素.

要创建一个只读的 Set (Set (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-set/)), 请使用 setOf() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/set-of.html) 函数.

要创建一个可变的 Set (MutableSet (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-set/)), 请使用 mutableSetOf() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-set-of.html) 函数.

创建 Set 时, Kotlin 可以推断它存储的元素类型. 如果要明确声明元素类型, 请在 Set 的声明之后的尖括号 <> 中添加类型:

```
fun main() {
//sampleStart
    // 只读的 Set
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    // 可变的 Set, 带有明确的类型声明
    val fruit: MutableSet<String> = mutableSetOf("apple", "banana",
"cherry", "cherry")

    println(readOnlyFruit)
    // 输出结果为 [apple, banana, cherry]
//sampleEnd
}
```

在上面的示例中你可以看到, 由于 Set 只包含唯一的元素, 重复的 `"cherry"` 元素被丢弃了.

> 为了防止无意中修改 Set 的内容, 你可以将可变的 Set 类型变换为 `Set`, 来得到它的一个只读的视图:
>
> ```
> val fruit: MutableSet<String> = mutableSetOf("apple",
> "banana", "cherry", "cherry")
> val fruitLocked: Set<String> = fruit
> ```

> 由于 Set 是 **无顺序的**, 你不能访问位于某个下标的元素.

要得到 Set 中元素的数量, 请使用 `.count()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/count.html) 函数:

```
fun main() {
//sampleStart
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    println("This set has ${readOnlyFruit.count()} items")
    // 输出结果为 This set has 3 items
//sampleEnd
}
```

要检查一个元素是否存在于 Set 中, 请使用 `in` 操作符 (["in 操作符" in "操作符重载"]()):

```
fun main() {
//sampleStart
    val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
    println("banana" in readOnlyFruit)
    // 输出结果为 true
//sampleEnd
}
```

要对可变 Set 添加或删除元素, 请分别使用 `.add()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-set/add.html) 和 `.remove()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/remove.html) 函数:

```
fun main() {
//sampleStart
    val fruit: MutableSet<String> = mutableSetOf("apple", "banana",
"cherry", "cherry")
    fruit.add("dragonfruit")    // 向 Set 添加 "dragonfruit"
    println(fruit)              // 输出结果为 [apple, banana, cherry,
dragonfruit]

    fruit.remove("dragonfruit") // 从 Set 中删除 "dragonfruit"
    println(fruit)              // 输出结果为 [apple, banana, cherry]
//sampleEnd
}
```

Map

Map 将元素保存为键值对(key-value pair). 你通过引用键(Key)来访问值(Value). 你可以将 Map 想象为好像一个食品菜单. 你可以通过寻找你想要吃的食物(键)来找到价格(值). 如果你想要查找一个值, 但不象 List 那样使用数字下标, 那么 Map 是很有用的.

- Map 中的每个键必须是唯一的, 这样 Kotlin 才能懂得你想要得到哪个值.
- 在 Map 中你可以有重复的值.

要创建一个只读的 Map (`Map` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/)), 请使用 `mapOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-of.html) 函数.

要创建一个可变的 Map (`MutableMap` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/)), 请使用 `mutableMapOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-map-of.html) 函数.

创建 Map 时, Kotlin 可以推断它存储的元素类型. 如果要明确声明元素类型, 请在 Map 的声明之后的尖括号 `<>` 中添加键和值的类型. 例如: `MutableMap<String, Int>`. 键的类型为 `String`, 值的类型为 `Int`.

创建 Map 的最简单的办法是在每个键和它对应的值之间使用 `to` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/to.html):

```
fun main() {
//sampleStart
    // 只读 Map
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190,
"orange" to 100)
    println(readOnlyJuiceMenu)
    // 输出结果为 {apple=100, kiwi=190, orange=100}

    // 可变的 Map, 带有明确的类型声明
    val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to
100, "kiwi" to 190, "orange" to 100)
    println(juiceMenu)
    // 输出结果为 {apple=100, kiwi=190, orange=100}
//sampleEnd
}
```

⚠ 为了防止无意中修改 Map 的内容, 你可以将可变的 Map 类型变换为 `Map`, 来得到它的一个只读的视图:

```
val juiceMenu: MutableMap<String, Int> =
mutableMapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
val juiceMenuLocked: Map<String, Int> = juiceMenu
```

要访问 Map 中的值, 请使用 下标操作符 ("下标访问操作符" in "操作符重载") [], 以它的键为下标:

```
fun main() {
//sampleStart
    // 只读 Map
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190,
"orange" to 100)
    println("The value of apple juice is:
${readOnlyJuiceMenu["apple"]}")
    // 输出结果为 The value of apple juice is: 100
//sampleEnd
}
```

要得到 Map 中元素的数量, 请使用 .count() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/count.html) 函数:

```
fun main() {
//sampleStart
    // 只读 Map
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190,
"orange" to 100)
    println("This map has ${readOnlyJuiceMenu.count()} key-value
pairs")
    // 输出结果为 This map has 3 key-value pairs
//sampleEnd
}
```

要对可变 Map 添加或删除元素, 请分别使用 .put() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/put.html) 和 .remove() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/remove.html) 函数:

```
fun main() {
//sampleStart
    val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to
100, "kiwi" to 190, "orange" to 100)
    juiceMenu.put("coconut", 150) // 向 Map 添加键 "coconut" 和值 150
    println(juiceMenu)
    // 输出结果为 {apple=100, kiwi=190, orange=100, coconut=150}
```

```
    juiceMenu.remove("orange")    // 从 Map 删除键 "orange"
    println(juiceMenu)
    // 输出结果为 {apple=100, kiwi=190, coconut=150}
//sampleEnd
}
```

要检查一个键是否存在于 Map 中, 请使用 `.containsKey()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/contains-key.html) 函数:

```
fun main() {
//sampleStart
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190,
"orange" to 100)
    println(readOnlyJuiceMenu.containsKey("kiwi"))
    // 输出结果为 true
//sampleEnd
}
```

要得到 Map 中所有键或所有值的集合, 请分别使用 `keys` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/keys.html) 和 `values` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/values.html) 属性:

```
fun main() {
//sampleStart
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190,
"orange" to 100)
    println(readOnlyJuiceMenu.keys)
    // 输出结果为 [apple, kiwi, orange]
    println(readOnlyJuiceMenu.values)
    // 输出结果为 [100, 190, 100]
//sampleEnd
}
```

> `keys` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/keys.html) 和 `values` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/values.html)

是对象的 **属性**. 要访问一个对象的属性, 请在对象之后加上点号 `.`, 然后把属性名写在后面.

属性会在 类 (类) 的章节中详细介绍. 目前你只需要知道如何访问它们就行了.

要检查一个键或值是否存在于 Map 中, 请使用 `in` 操作符 ("in 操作符" in "操作符重载"):

```
fun main() {
//sampleStart
    val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190,
"orange" to 100)
    println("orange" in readOnlyJuiceMenu.keys)
    // 输出结果为 true
    println(200 in readOnlyJuiceMenu.values)
    // 输出结果为 false
//sampleEnd
}
```

关于集合的其它更多功能, 请参见 集合 (集合(Collection)概述).

现在你已经知道了基本类型, 以及如何管理集合, 下面我们来看看在你的程序中能够使用的 控制流 (控制流).

实际练习

习题 1

你有一个 “绿色” 数字的 List, 和一个 “红色” 数字的 List. 完成下面的代码, 打印这两个 List 中总共有多少个数字.

```
fun main() {
    val greenNumbers = listOf(1, 4, 23)
    val redNumbers = listOf(17, 2)
    // 在这里编写你的代码
}
```

```
fun main() {
    val greenNumbers = listOf(1, 4, 23)
```

```
    val redNumbers = listOf(17, 2)
    val totalCount = greenNumbers.count() + redNumbers.count()
    println(totalCount)
}
```

习题 2

你有一个 Set, 其中包含你的服务器支持的协议. 一个用户要求使用某个协议. 完成下面的程序, 检查用户要求使用的协议是否支持 (`isSupported` 必须是 Boolean 值).

```
fun main() {
    val SUPPORTED = setOf("HTTP", "HTTPS", "FTP")
    val requested = "smtp"
    val isSupported = // 在这里编写你的代码
    println("Support for $requested: $isSupported")
}
```

提示

请确保使用字符串的大写格式来检查请求的协议. 你可以使用 .uppercase() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/uppercase.html) 函数来帮助你实现这一点.

```
fun main() {
    val SUPPORTED = setOf("HTTP", "HTTPS", "FTP")
    val requested = "smtp"
    val isSupported = requested.uppercase() in SUPPORTED
    println("Support for $requested: $isSupported")
}
```

习题 3

定义一个 Map, 将 1 到 3 的数字对应到它们的拼写. 使用这个 Map 来拼写指定的数字.

```
fun main() {
    val number2word = // 在这里编写你的代码
    val n = 2
```

```
    println("$n is spelt as '${< 在这里编写你的代码 >}'")
}
```

```
fun main() {
    val number2word = mapOf(1 to "one", 2 to "two", 3 to "three")
    val n = 2
    println("$n is spelt as '${number2word[n]}'")
}
```

下一步

控制流 (控制流)

控制流

1 Hello world (Hello world) 2 基本类型 (基本类型) 3 集合 (集合(Collection)) 4 **控制流** 5 函数 (函数) 6 类 (类) 7 Null 值安全性 (Null 值安全性)

最终更新: 2024/09/10

和其他的编程语言一样, Kotlin 能够根据一个代码片段的计算结果是否为 true 来做出决策. 这样的代码片段称为 **条件表达式**. Kotlin 还能够创建循环, 并在循环上迭代.

条件表达式

Kotlin 提供了 `if` 和 `when` 来检测条件表达式.

> 如果你必须在 `if` 和 `when` 之间做选择, 我们推荐使用 `when`, 因为它可以创建更加健壮和安全的程序.

If

要使用 `if`, 请将条件表达式放在小括号 `()` 之内, 当调节表达式的结果为 true 时要做的操作放在大括号 `{}` 之内:

```
fun main() {
//sampleStart
    val d: Int
    val check = true

    if (check) {
        d = 1
    } else {
        d = 2
    }

    println(d)
    // 输出结果为 1
//sampleEnd
}
```

在 Kotlin 中没有三元操作符 `condition ? then : else`. `if` 可以用作表达式, 替代三元操作符的功能. 把 `if` 用作表达式时, 不要使用大括号 `{}`:

```
fun main() {
//sampleStart
    val a = 1
    val b = 2

    println(if (a > b) a else b) // 返回值: 2
//sampleEnd
}
```

When

如果你的条件表达式存在多个分支, 请使用 `when`. `when` 可以用作语句, 也可以用作表达式.

下面是将 `when` 用作语句的例子:

- 将条件表达式放在小括号 `()` 之内, 将各个条件分支需要进行的操作放在大括号 `{}` 之内.
- 在每个分支中, 使用 `->` 分隔分支条件与对应的操作.

```
fun main() {
//sampleStart
    val obj = "Hello"

    when (obj) {
        // 检查 obj 是否等于 "1"
        "1" -> println("One")
        // 检查 obj 是否等于 "Hello"
        "Hello" -> println("Greeting")
        // 默认语句
        else -> println("Unknown")
    }
    // 输出结果为 Greeting
//sampleEnd
}
```

> 注意, 会按顺序检查所有的分支条件, 直到遇到一个条件被满足. 因此只有第一个满足条件的分支会被执行.

下面是将 `when` 用作表达式的例子. `when` 表达式的结果会被立即赋值给一个变量:

```
fun main() {
//sampleStart
    val obj = "Hello"

    val result = when (obj) {
        // 如果 obj 等于 "1", 将 result 设置为 "one"
        "1" -> "One"
        // 如果 obj 等于 "Hello", 将 result 设置为 "Greeting"
        "Hello" -> "Greeting"
        // 如果前面的条件都不满足, 将 result 设置为 "Unknown"
        else -> "Unknown"
    }
    println(result)
    // 输出结果为 Greeting
//sampleEnd
}
```

如果 `when` 被用作表达式, 必须存在 else 分支, 除非编译器能够检测出分支条件覆盖了所有的可能情况.

上面的例子演示了, `when` 可以用于对变量进行匹配. `when` 还可以用于对一组 Boolean 表达式进行检查:

```
fun main() {
//sampleStart
    val temp = 18

    val description = when {
        // 如果 temp < 0 为 true, 将 description 设置为 "very cold"
        temp < 0 -> "very cold"
        // 如果 temp < 10 为 true, 将 description 设置为 "a bit cold"
        temp < 10 -> "a bit cold"
        // 如果 temp < 20 为 true, 将 description 设置为 "warm"
```

```
        temp < 20 -> "warm"
        // 如果前面的条件都不满足, 将  description 设置为 "hot"
        else -> "hot"
    }
    println(description)
    // 输出结果为 warm
//sampleEnd
}
```

值范围

在讨论循环之前, 有必要了解如何构造一个作为循环迭代对象的值范围.

在 Kotlin 中, 创建值范围最常见的办法是使用 `..` 操作符. 例如, `1..4` 相当于 `1, 2, 3, 4`.

要声明一个值范围, 不包含它的终端值, 请使用 `..<` 操作符. 例如, `1..<4` 相当于 `1, 2, 3`.

要声明一个相反顺序的值范围, 请使用 `downTo`. 例如, `4 downTo 1` 相当于 `4, 3, 2, 1`.

要声明一个值范围, 递增步长不为 1, 请使用 `step` 指定你希望的递增步长值. 例如, `1..5 step 2` 相当于 `1, 3, 5`.

你也可以对 `Char` 的值范围进行相同的操作:

- `'a'..'d'` 相当于 `'a', 'b', 'c', 'd'`
- `'z' downTo 's' step 2` 相当于 `'z', 'x', 'v', 't'`

循环

在编程中两种最常见的循环结构是 `for` 和 `while`. 使用 `for` 可以对一个值范围进行遍历, 并执行某个操作. 使用 `while` 可以反复执行某个操作, 直到满足某个条件为止.

for

使用关于值范围的新知识, 你可以创建一个 `for` 循环, 对数字 1 到 5 进行遍历, 并打印每个数字.

请将迭代器(iterator)和值范围放在小括号 `()` 之内, 并使用关键字 `in`. 将你想要执行的操作放在大括号 `{}` 之内:

```
fun main() {
//sampleStart
    for (number in 1..5) {
```

```
        // number 是迭代器(iterator), 1..5 是值范围
        print(number)
    }
    // 输出结果为 12345
//sampleEnd
}
```

`for` 循环也可以对集合(Collection)进行遍历:

```
fun main() {
//sampleStart
    val cakes = listOf("carrot", "cheese", "chocolate")

    for (cake in cakes) {
        println("Yummy, it's a $cake cake!")
    }
    // 输出结果为 Yummy, it's a carrot cake!
    // 输出结果为 Yummy, it's a cheese cake!
    // 输出结果为 Yummy, it's a chocolate cake!
//sampleEnd
}
```

while

`while` 有两种使用方式:

- 当一个条件表达式为 true 时, 执行一个代码段. (`while`)
- 先执行一个代码段, 然后再检查条件表达式. (`do-while`)

在第一种使用场景 (`while`) 中:

- 在小括号 `()` 中声明条件表达式, 当满足这个条件表达式时, 循环会继续.
- 在大括号 `{}` 中, 添加你想要执行的操作.

> 下面的示例使用 递增操作符 ("递增与递减操作符" in "操作符重载") `++` 来增加 `cakesEaten` 变量的值.

```
fun main() {
//sampleStart
    var cakesEaten = 0
    while (cakesEaten < 3) {
        println("Eat a cake")
        cakesEaten++
    }
    // 输出结果为 Eat a cake
    // 输出结果为 Eat a cake
    // 输出结果为 Eat a cake
//sampleEnd
}
```

在第二种使用场景 (`do-while`) 中:

- 在小括号 `()` 中声明条件表达式, 当满足这个条件表达式时, 循环会继续.
- 在大括号 `{}` 中, 添加你想要执行的操作, 并添加关键字 `do`.

```
fun main() {
//sampleStart
    var cakesEaten = 0
    var cakesBaked = 0
    while (cakesEaten < 3) {
        println("Eat a cake")
        cakesEaten++
    }
    do {
        println("Bake a cake")
        cakesBaked++
    } while (cakesBaked < cakesEaten)
    // 输出结果为 Eat a cake
    // 输出结果为 Eat a cake
    // 输出结果为 Eat a cake
    // 输出结果为 Bake a cake
    // 输出结果为 Bake a cake
    // 输出结果为 Bake a cake
```

```
//sampleEnd
}
```

关于条件表达式与循环的更多示例, 请参见 条件与循环 (条件与循环).

现在你已经直到了 Kotlin 控制流的基本知识, 下面我们来学习如何编写你自己的 函数 (函数).

实际练习

习题 1

使用 `when` 表达式, 更新下面的程序, 当你输入 GameBoy 按钮的名称时, 打印对应的动作.

按钮	动作
A	Yes
B	No
X	Menu
Y	Nothing
其他	There is no such button

```
fun main() {
    val button = "A"

    println(
        // 在这里编写你的代码
    )
}
```

```
fun main() {
    val button = "A"
```

```
    println(
        when (button) {
            "A" -> "Yes"
            "B" -> "No"
            "X" -> "Menu"
            "Y" -> "Nothing"
            else -> "There is no such button"
        }
    )
}
```

习题 2

你有一个程序, 计算批萨的片数, 直到有了 8 片, 组成一整个批萨. 请用两种方式重构这个程序:

- 使用 `while` 循环.
- 使用 `do-while` 循环.

```
fun main() {
    var pizzaSlices = 0
    // 要重构的代码从这里开始
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    println("There's only $pizzaSlices slice/s of pizza :(")
    pizzaSlices++
    // 要重构的代码到这里结束
```

```
    println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D")
}
```

```
fun main() {
    var pizzaSlices = 0
    while ( pizzaSlices < 7 ) {
        pizzaSlices++
        println("There's only $pizzaSlices slice/s of pizza :(")
    }
    pizzaSlices++
    println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D")
}
```

```
fun main() {
    var pizzaSlices = 0
    pizzaSlices++
    do {
        println("There's only $pizzaSlices slice/s of pizza :(")
        pizzaSlices++
    } while ( pizzaSlices < 8 )
    println("There are $pizzaSlices slices of pizza. Hooray! We have a whole pizza! :D")
}
```

习题 3

编写一个程序, 模拟 Fizz buzz (https://en.wikipedia.org/wiki/Fizz_buzz) 游戏. 你的任务是打印从 1 到 100 的数字, 如果数字能被 3 整除, 则将它替换为 "fizz", 能被 5 整除, 则将它替换为 "buzz". 同时能被 3 和 5 整除, 则将它替换为 "fizzbuzz".

> **提示**
>
> 使用 for 循环来计数, 使用 when 表达式来决定每一步打印什么内容.

```
fun main() {
    // 在这里编写你的代码
}
```

```
fun main() {
    for (number in 1..100) {
        println(
            when {
                number % 15 == 0 -> "fizzbuzz"
                number % 3 == 0 -> "fizz"
                number % 5 == 0 -> "buzz"
                else -> number.toString()
            }
        )
    }
}
```

习题 4

你有一个单词列表. 使用 `for` 和 `if` 来打印以 `l` 字母开头的单词.

> **提示**
>
> 使用 String 类型的 .startsWith() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/starts-with.html) 函数.

```
fun main() {
    val words = listOf("dinosaur", "limousine", "magazine",
"language")
    // 在这里编写你的代码
}
```

```
fun main() {
    val words = listOf("dinosaur", "limousine", "magazine",
```

```
"language")
    for (w in words) {
        if (w.startsWith("l"))
            println(w)
    }
}
```

下一步

函数 (函数)

函数

① Hello world (Hello world) ② 基本类型 (基本类型) ③ 集合(Collection) (集合(Collection))
④ 控制流 (控制流) ⑤ **函数** ⑥ 类 (类) ⑦ Null 值安全性 (Null 值安全性)

最终更新: 2024/09/10

在 Kotlin 中, 你可以使用 `fun` 关键字声明你自己的函数.

```
fun hello() {
    return println("Hello, world!")
}

fun main() {
    hello()
    // 输出结果为 Hello, world!
}
```

在 Kotlin 中:

- 函数参数写在小括号 `()` 之内.
- 每个参数必须指定类型, 多个参数必须用逗号 `,` 隔开.
- 返回值类型写在函数的小括号 `()` 之后, 用冒号 `:` 隔开.
- 函数的 body 部写在大括号 `{}` 之内.
- `return` 关键字用来退出函数, 或从函数返回某个值.

> 如果函数不返回任何有用的值, 那么可以省略返回值类型和 `return` 关键字. 关于这个问题, 详情请参见 没有返回值的函数.

在下面的示例中:

- `x` 和 `y` 是函数参数.

- x 和 y 类型为 Int.

- 函数的返回值类型为 Int.

- 函数被调用时返回 x 和 y 的和.

```
fun sum(x: Int, y: Int): Int {
    return x + y
}

fun main() {
    println(sum(1, 2))
    // 输出结果为 3
}
```

在我们的 编码规约 ("函数名称" in "编码规约") 中, 我们建议函数名称以小写字母开头, 并使用驼峰式大小写(Camel case), 不使用下划线.

命名参数

为了让代码更简洁, 调用函数时, 你不必指定参数名称. 但是, 指定参数名称可以让你的代码更易于阅读. 这种方式称为 **命名参数(named argument)**. 如果你指定了参数名称, 那么可以用任意的顺序来写这些参数.

在下面的示例中, 使用了 字符串模板 ("字符串模板" in "字符串") ($) 来访问参数值, 并将它们转换为 String 类型, 然后拼接到一个字符串中, 用于打印输出.

```
fun printMessageWithPrefix(message: String, prefix: String) {
    println("[$prefix] $message")
}

fun main() {
    // 使用命名参数, 交换了参数的顺序
    printMessageWithPrefix(prefix = "Log", message = "Hello")
```

```
    // 输出结果为 [Log] Hello
}
```

默认的参数值

你可以为函数参数定义默认值. 调用你的函数时, 有默认值的参数可以省略. 要声明默认值, 请在参数类型之后使用赋值操作符 `=`:

```
fun printMessageWithPrefix(message: String, prefix: String = "Info")
{
    println("[$prefix] $message")
}

fun main() {
    // 使用两个参数调用函数
    printMessageWithPrefix("Hello", "Log")
    // 输出结果为 [Log] Hello

    // 只使用 message 参数调用函数
    printMessageWithPrefix("Hello")
    // 输出结果为 [Info] Hello

    printMessageWithPrefix(prefix = "Log", message = "Hello")
    // 输出结果为 [Log] Hello
}
```

> 你可以跳过某个有默认值的参数, 而不是省略所有参数. 但是, 在第一个跳过的参数之后, 你必须对后续的所有参数指定名称.

没有返回值的函数

如果你的函数不返回任何有用的值, 那么它的返回值类型为 `Unit`. `Unit` 类型只有唯一的一个值 – `Unit`. 你不必在你的函数 body 部明确的声明返回值为 `Unit`. 因此你不必使用 `return` 关键字, 也不必声明返回值类型:

```
fun printMessage(message: String) {
    println(message)
```

```
    // `return Unit` 或 `return` 都是可选的
}

fun main() {
    printMessage("Hello")
    // 输出结果为 Hello
}
```

单一表达式函数

为了让代码更加简洁, 你可以使用单一表达式函数. 例如, `sum()` 函数可以写得更短一些:

```
fun sum(x: Int, y: Int): Int {
    return x + y
}

fun main() {
    println(sum(1, 2))
    // 输出结果为 3
}
```

你可以删除大括号 `{}`, 使用赋值操作符 `=` 来声明函数的 body 部. 由于 Kotlin 的类型推断能力, 你还可以省略返回值类型. 这样, `sum()` 函数就变成只有 1 行:

```
fun sum(x: Int, y: Int) = x + y

fun main() {
    println(sum(1, 2))
    // 输出结果为 3
}
```

> 只有在你的函数没有 body 部(`{}`) 时, 才能够省略返回值类型. 否则你的函数的返回值类型将是 `Unit`.

函数的实际练习

习题 1

写一个名为 `circleArea` 的函数, 接受一个整数参数, 表示圆的半径, 输出圆的面积大小.

> 在这个习题中, 你会导入一个包, 以便通过 `PI` 来访问 pi 值. 关于包的导入, 更多详情请参见 包与导入 (包(Package)与导入(Import)).

```
import kotlin.math.PI

fun circleArea() {
    // 在这里编写你的代码
}
fun main() {
    println(circleArea(2))
}
```

```
import kotlin.math.PI

fun circleArea(radius: Int): Double {
    return PI * radius * radius
}

fun main() {
    println(circleArea(2)) // 输出结果为 12.566370614359172
}
```

习题 2

将前一个习题中的 `circleArea` 函数重写为单一表达式函数.

```
import kotlin.math.PI

// 在这里编写你的代码

fun main() {
```

```
    println(circleArea(2))
}
```

```
import kotlin.math.PI

fun circleArea(radius: Int): Double = PI * radius * radius

fun main() {
    println(circleArea(2)) // 输出结果为 12.566370614359172
}
```

习题 3

你有一个函数, 它接受一个时/分/秒单位给定的时间间隔, 然后翻译为秒单位. 大多数情况下, 你只需要传递 1 个或 2 个参数, 而其它参数为 0. 改进这个函数以及调用它的代码, 使用默认参数值和命名参数, 让代码更加易于阅读.

```
fun intervalInSeconds(hours: Int, minutes: Int, seconds: Int) =
    ((hours * 60) + minutes) * 60 + seconds

fun main() {
    println(intervalInSeconds(1, 20, 15))
    println(intervalInSeconds(0, 1, 25))
    println(intervalInSeconds(2, 0, 0))
    println(intervalInSeconds(0, 10, 0))
    println(intervalInSeconds(1, 0, 1))
}
```

```
fun intervalInSeconds(hours: Int = 0, minutes: Int = 0, seconds: Int
= 0) =
    ((hours * 60) + minutes) * 60 + seconds

fun main() {
    println(intervalInSeconds(1, 20, 15))
    println(intervalInSeconds(minutes = 1, seconds = 25))
    println(intervalInSeconds(hours = 2))
    println(intervalInSeconds(minutes = 10))
```

```
    println(intervalInSeconds(hours = 1, seconds = 1))
}
```

Lambda 表达式

Kotlin 允许你使用 Lambda 表达式, 为函数编写更加简洁的代码.

例如, 下面的 `uppercaseString()` 函数:

```
fun uppercaseString(string: String): String {
    return string.uppercase()
}
fun main() {
    println(uppercaseString("hello"))
    // 输出结果为 HELLO
}
```

可以写成一个 Lambda 表达式:

```
fun main() {
    println({ string: String -> string.uppercase() }("hello"))
    // 输出结果为 HELLO
}
```

Lambda 表达式初看起来可能难于理解, 所以我们将它分解成各个部分. Lambda 表达式写在大括号 `{}` 之内.

在 Lambda 表达式之内, 你会写以下内容:

- 参数, 在 `->` 之前.
- 函数 body 部, 在 `->` 之后.

在上面的示例中:

- `string` 是函数参数.
- `string` 类型为 `String`.
- 函数返回对 `string` 调用 `.uppercase()`

(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/uppercase.html) 函数的结果.

> 如果你声明没有参数的 Lambda 表达式, 那么不必使用 `->`. 例如:
>
> ```
> { println("Log message") }
> ```

可以用很多方式使用 Lambda 表达式. 你可以:

- 将 Lambda 表达式赋值给一个变量, 在后面的代码中调用它
- 将 Lambda 表达式用作另一个函数的参数
- 从一个函数返回 Lambda 表达式
- 单独调用一个 Lambda 表达式

赋值给变量

要将 Lambda 表达式赋值给一个变量, 请使用赋值操作符 `=`:

```
fun main() {
    val upperCaseString = { string: String -> string.uppercase() }
    println(upperCaseString("hello"))
    // 输出结果为 HELLO
}
```

传递给另一个函数

将 Lambda 表达式传递给另一个函数, 这个功能是很有用的, 一个很好的例子是对集合(Collection)使用 `.filter()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html) 函数:

```
fun main() {
    //sampleStart
    val numbers = listOf(1, -2, 3, -4, 5, -6)
    val positives = numbers.filter { x -> x > 0 }
    val negatives = numbers.filter { x -> x < 0 }
    println(positives)
    // 输出结果为 [1, 3, 5]
```

```
    println(negatives)
    // 输出结果为 [-2, -4, -6]
    //sampleEnd
}
```

.filter() 函数接受一个 Lambda 表达式, 作为判定条件:

- { x -> x > 0 } 接受 List 中的每个元素, 只返回正数.
- { x -> x < 0 } 接受 List 中的每个元素, 只返回负数.

> 如果一个 Lambda 表达式是函数的唯一参数, 你可以去掉函数的小括号 (). 这是 尾缀 Lambda 表达式(Trailing Lambda) 的一个例子, 我们会在本章末尾详细介绍.

另一个好的例子是, 使用 .map() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map.html) 函数, 对集合中的元素进行变换:

```
fun main() {
    //sampleStart
    val numbers = listOf(1, -2, 3, -4, 5, -6)
    val doubled = numbers.map { x -> x * 2 }
    val tripled = numbers.map { x -> x * 3 }
    println(doubled)
    // 输出结果为 [2, -4, 6, -8, 10, -12]
    println(tripled)
    // 输出结果为 [3, -6, 9, -12, 15, -18]
    //sampleEnd
}
```

.map() 函数接受一个 Lambda 表达式, 作为变换函数:

- { x -> x * 2 } 接受 List 中的每个元素, 返回这个元素乘以 2 的结果.
- { x -> x * 3 } 接受 List 中的每个元素, 返回这个元素乘以 3 的结果.

函数类型

在从一个函数返回一个 Lambda 表达式之前, 你首先需要理解 **函数类型**.

你已经学习了基本类型, 但函数本身也有它的类型. Kotlin 的类型推断功能能够通过参数类型推断一个函数的类型. 但有的时候你需要明确指定函数类型. 编译器需要函数类型, 然后才能知道对这个函数允许什么, 不允许什么.

函数类型的语法包括:

- 每个参数的类型, 写在小括号 `()` 之内, 以逗号 `,` 分隔.
- 返回值类型, 写在 `->` 之后.

例如: `(String) -> String`, 或 `(Int, Int) -> Int`.

如果为 `upperCaseString()` 定义一个函数类型, 那么 Lambda 表达式如下:

```
val upperCaseString: (String) -> String = { string ->
string.uppercase() }

fun main() {
    println(upperCaseString("hello"))
    // 输出结果为 HELLO
}
```

如果你的 Lambda 表达式没有参数, 那么小括号 `()` 保留为空. 例如: `() -> Unit`

> 你必须声明参数类型和返回值类型, 要么写在 Lambda 表达式内, 要么声明为函数类型. 否则, 编译器无法知道你的 Lambda 表达式的类型.
>
> 例如, 下面的代码无法工作:
>
> `val upperCaseString = { str -> str.uppercase() }`

从函数中返回

可以从函数中返回 Lambda 表达式. 为了让编译器知道返回的 Lambda 表达式 的类型, 你必须声明一个函数类型.

在下面的示例中, `toSeconds()` 函数返回的函数类型是 `(Int) -> Int`, 因为它总是返回一个 Lambda 表达式, 这个 Lambda 表达式接受一个 `Int` 类型的参数, 并返回一个 `Int` 值.

这个示例使用 `when` 表达式, 来确定在调用 `toSeconds()` 时返回哪个 Lambda 表达式:

```
fun toSeconds(time: String): (Int) -> Int = when (time) {
    "hour" -> { value -> value * 60 * 60 }
```

```
    "minute" -> { value -> value * 60 }
    "second" -> { value -> value }
    else -> { value -> value }
}

fun main() {
    val timesInMinutes = listOf(2, 10, 15, 1)
    val min2sec = toSeconds("minute")
    val totalTimeInSeconds = timesInMinutes.map(min2sec).sum()
    println("Total time is $totalTimeInSeconds secs")
    // 输出结果为 Total time is 1680 secs
}
```

单独调用

Lambda 表达式可以单独调用, 方法是在大括号 `{}` 之后添加小括号 `()`, 并在小括号中加上参数:

```
fun main() {
    //sampleStart
    println({ string: String -> string.uppercase() }("hello"))
    // 输出结果为 HELLO
    //sampleEnd
}
```

尾缀 Lambda 表达式(Trailing Lambda)

你已经看到, 如果一个 Lambda 表达式是函数的唯一参数, 你可以去掉函数的小括号 `()`. 如果一个 Lambda 表达式是函数的最后一个参数, 那么 Lambda 表达式可以写在函数的小括号 `()` 之外. 对这两种情况, 这样的语法称为 **尾缀 Lambda 表达式(Trailing Lambda)**.

例如, `.fold()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/fold.html) 函数接受一个初始值, 以及一个操作:

```
fun main() {
    //sampleStart
    // 初始值为 0.
    // 操作是对初始值累加 List 中的每个元素.
    println(listOf(1, 2, 3).fold(0, { x, item -> x + item })) // 输出
结果为 6
```

```
    // 或者, 也可以写成 尾缀 Lambda 表达式的形式
    println(listOf(1, 2, 3).fold(0) { x, item -> x + item })  // 输出
结果为 6
    //sampleEnd
}
```

关于 Lambda 表达式, 更多详情请参见 Lambda 表达式与匿名函数(Anonymous Function) ("Lambda 表达式与匿名函数(Anonymous Function)" in "高阶函数与 Lambda 表达式").

本教程的下一章是学习 Kotlin 中的 类 (类).

Lambda 表达式的实际练习

习题 1

你有一个 Web Service 支持的动作列表, 所有请求的一个共通前缀, 某个资源的一个 ID. 要对资源 ID 5 请求 `title` 动作, 你需要创建下面的 URL: `https://example.com/book-info/5/title`. 使用一个 Lambda 表达式, 从动作列表创建对应的 URL 列表.

```
fun main() {
    val actions = listOf("title", "year", "author")
    val prefix = "https://example.com/book-info"
    val id = 5
    val urls = // 在这里编写你的代码
    println(urls)
}
```

```
fun main() {
    val actions = listOf("title", "year", "author")
    val prefix = "https://example.com/book-info"
    val id = 5
    val urls = actions.map { action -> "$prefix/$id/$action" }
    println(urls)
}
```

习题 2

编写一个函数, 接受一个 `Int` 值和一个动作 (一个 `() -> Unit` 类型的函数), 然后重复执行这个动作指定的次数. 然后使用这个函数打印 “Hello” 5 次.

```
fun repeatN(n: Int, action: () -> Unit) {
    // 在这里编写你的代码
}

fun main() {
    // 在这里编写你的代码
}
```

```
fun repeatN(n: Int, action: () -> Unit) {
    for (i in 1..n) {
        action()
    }
}

fun main() {
    repeatN(5) {
        println("Hello")
    }
}
```

下一步

类 (类)

类

① Hello world (Hello world) ② 基本类型 (基本类型) ③ 集合(Collection) (集合(Collection)) ④ 控制流 (控制流) ⑤ 函数 (函数) ❻ **类** ❼ Null 值安全性 (Null 值安全性)

最终更新: 2024/09/10

Kotlin 通过类和对象支持面向对象的编程. 要在你的程序中存储数据, 对象是非常有用的. 类允许你为一个对象声明一组特性. 当你从一个类创建对象时, 你就可以节省时间和精力, 因为你不需要每次都声明这些特性.

要声明一个类, 请使用 `class` 关键字:

```
class Customer
```

属性

可以在属性中声明一个类的对象的特性. 你可以为一个类声明属性:

- 放在类的名称之后的小括号 `()` 之内.

```
class Contact(val id: Int, var email: String)
```

- 放在大括号 `{}` 定义的类的 body 部之内.

```
class Contact(val id: Int, var email: String) {
    val category: String = ""
}
```

除非在类的实例创建之后需要修改属性的值, 否则我们推荐将属性声明为只读的 (`val`).

在小括号内声明属性时, 你可以不使用 `val` 或 `var`, 但在实例创建之后, 这样的属性将不可访问.

- 包含在小括号 `()` 之内的内容称为 **类头部(Class Header)**.

* 声明类的属性时, 你可以使用 尾随逗号(Trailing Comma) ("尾随逗号(Trailing Comma)" in "编码规约").

和函数参数一样, 类的属性可以有默认值:

```
class Contact(val id: Int, var email: String = "example@gmail.com")
{
    val category: String = "work"
}
```

创建实例

要从一个类创建一个对象, 你需要使用 **构造器(Constructor)**, 声明一个类的 **实例**.

默认情况下, Kotlin 会使用类头部(Class Header)中声明的参数, 自动创建一个构造器.

例如:

```
class Contact(val id: Int, var email: String)

fun main() {
    val contact = Contact(1, "mary@gmail.com")
}
```

在上面的示例中:

* `Contact` 是一个类.
* `contact` 是 `Contact` 类的一个实例.
* `id` 和 `email` 是属性.
* `id` 和 `email` 和默认构造器一起, 用来创建 `contact`.

Kotlin 类可以有多个构造器, 包括你自己定义的构造器. 关于如何声明多个构造器, 详情请参见 构造器 ("构造器" in "类").

访问属性

要访问一个实例的属性, 请在实例名称之后加上点号 `.`, 然后写上属性名称:

```
class Contact(val id: Int, var email: String)

fun main() {
    val contact = Contact(1, "mary@gmail.com")

    // 打印属性的值: email
    println(contact.email)
    // mary@gmail.com

    // 更新属性的值: email
    contact.email = "jane@gmail.com"

    // 打印属性的新值: email
    println(contact.email)
    // 输出结果为 jane@gmail.com
}
```

⚠ 要把属性的值拼接为字符串的一部分, 你可以使用字符串模板 (`$`). 例如:

```
println("Their email address is: ${contact.email}")
```

成员函数

除了声明属性作为一个对象的特性之外, 你还可以通过成员函数来定义一个对象的行为.

在 Kotlin 中, 成员函数必须在类的 body 部之内声明. 要调用一个实例上的成员函数, 请在实例名称之后加上点号 `.`, 然后写上函数名称. 例如:

```
class Contact(val id: Int, var email: String) {
    fun printId() {
        println(id)
    }
}

fun main() {
    val contact = Contact(1, "mary@gmail.com")
    // 调用成员函数 printId()
```

```
    contact.printId()
    // 输出结果为 1
}
```

数据类

Kotlin 有 **数据类(Data Class)**, 非常适合于存储数据. 数据类有和普通类一样的功能, 但它们还自动带有一些额外的成员函数. 这些成员函数可以将实例打印为易于阅读的字符串输出, 比较类的实例, 复制实例, 等等等等. 由于这些函数是自动存在的, 因此你不必耗费时间为每个类编写相同的样板代码(Boilerplate Code).

要声明一个数据类, 请使用关键字 `data`:

```
data class User(val name: String, val id: Int)
```

数据类的预先定义的成员函数中, 最有用的是:

函数	描述
`.toString()`	将类实例和它的属性打印为一个易于阅读的字符串.
`.equals()` 或 `==`	比较一个类的实例.
`.copy()`	创建一个类的实例, 从另一个实例复制, 一部分属性可以不同.

关于这些函数的使用示例, 请参见以下小节:

- 打印为字符串
- 比较实例
- 复制实例

打印为字符串

要将一个类的实例打印为易于阅读的字符串, 你可以明确调用 `.toString()` 函数, 或使用打印函数 (`println()` 和 `print()`), 这些函数会自动为你调用 `.toString()`:

```
data class User(val name: String, val id: Int)

fun main() {
    val user = User("Alex", 1)

    //sampleStart
    // 自动使用 toString() 函数, 让输出结果易于阅读
    println(user)
    // 输出结果为 User(name=Alex, id=1)
    //sampleEnd
}
```

这个功能在调试程序或创建 log 时, 非常有用.

比较实例

要比较数据类的实例, 请使用相等比较操作符 `==`:

```
data class User(val name: String, val id: Int)

fun main() {
    //sampleStart
    val user = User("Alex", 1)
    val secondUser = User("Alex", 1)
    val thirdUser = User("Max", 2)

    // 比较 user 和 second user
    println("user == secondUser: ${user == secondUser}")
    // 输出结果为 user == secondUser: true

    // 比较 user 和 third user
    println("user == thirdUser: ${user == thirdUser}")
    // 输出结果为 user == thirdUser: false
    //sampleEnd
}
```

复制实例

要对一个数据类的实例创建一个完全相同的复制, 请对这个实例调用 `.copy()` 函数.

要对一个数据类的实例创建一个复制, **并且** 改变一部分属性, 请对这个实例调用 `.copy()` 函数, **并** 加上要替换的属性值, 作为函数的参数.

例如:

```
data class User(val name: String, val id: Int)

fun main() {
    //sampleStart
    val user = User("Alex", 1)
    val secondUser = User("Alex", 1)
    val thirdUser = User("Max", 2)

    // 创建 user 的完全相同的复制
    println(user.copy())
    // 输出结果为 User(name=Alex, id=1)

    // 创建 user 的复制, 但使用另一个 name: "Max"
    println(user.copy("Max"))
    // 输出结果为 User(name=Max, id=1)

    // 创建 user 的复制, 但使用另一个 id: 3
    println(user.copy(id = 3))
    // 输出结果为 User(name=Alex, id=3)
    //sampleEnd
}
```

创建一个实例的复制, 要比修改原来的实例更加安全, 因为你对复制品所做的任何操作, 不会影响到依赖于原来那个实例的其他代码.

关于数据类, 更多详情请参见 数据类 (数据类(Data Class)).

本教程的最后一章是介绍 Kotlin 的 Null 值安全性 (Null 值安全性).

实际练习

习题 1

定义一个数据类 `Employee`, 带有两个属性: 一个是姓名, 一个是工资. 请确保工资的属性是可变的, 否则你在年底就不可能涨工资了! 主函数演示你如何使用这个数据类.

```
// 在这里编写你的代码

fun main() {
    val emp = Employee("Mary", 20)
    println(emp)
    emp.salary += 10
    println(emp)
}
```

```
data class Employee(val name: String, var salary: Int)

fun main() {
    val emp = Employee("Mary", 20)
    println(emp)
    emp.salary += 10
    println(emp)
}
```

习题 2

为了测试你的代码, 你需要一个生成器, 它能够创建随机的员工数据. 定义一个类, 其中包括可用的姓名的固定列表 (包含在类的 body 部之内), 还可以指定工资的最小值和最大值 (包含在类头部之内). 这次也一样, 主函数演示你如何使用这个类.

> **提示**
>
> List 有一个名为 .random() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/random.html) 的扩展函数, 它返回 List 内的一个随机元素.

> **提示**
>
> Random.nextInt(from = ..., until = ...) 返回给你一个随机的 Int 值, 它在指定的上下限值之内.

```
import kotlin.random.Random

data class Employee(val name: String, var salary: Int)

// 在这里编写你的代码

fun main() {
    val empGen = RandomEmployeeGenerator(10, 30)
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    empGen.minSalary = 50
    empGen.maxSalary = 100
    println(empGen.generateEmployee())
}
```

```
import kotlin.random.Random

data class Employee(val name: String, var salary: Int)

class RandomEmployeeGenerator(var minSalary: Int, var maxSalary:
Int) {
    val names = listOf("John", "Mary", "Ann", "Paul", "Jack",
"Elizabeth")
    fun generateEmployee() =
        Employee(names.random(),
            Random.nextInt(from = minSalary, until = maxSalary))
}

fun main() {
    val empGen = RandomEmployeeGenerator(10, 30)
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())
    empGen.minSalary = 50
    empGen.maxSalary = 100
```

```
    println(empGen.generateEmployee())
}
```

下一步

Null 值安全性 (Null 值安全性)

Null 值安全性

1 Hello world (Hello world) 2 基本类型 (基本类型) 3 集合(Collection) (集合(Collection))
4 控制流 (控制流) 5 函数 (函数) 6 类 (类) 7 **Null 值安全性**

最终更新: 2024/09/10

在 Kotlin 中, 可以使用 `null` 值. 为了帮助在程序中防止 `null` 值相关的问题, Kotlin 提供了 null 值安全性功能. null 值安全性功能会在编译期检测 `null` 值潜在的问题, 而不是在运行期.

Null 安全性是多种功能的组合, 使得你能够:

- 如果你的程序允许 `null` 值, 可以明确声明.
- 检查 `null` 值.
- 对可能包含 `null` 值的属性或函数, 使用安全调用.
- 如果检测到 `null` 值时, 声明如何处理.

可为 null 的类型

Kotlin 支持可为 null 的类型, 这样的类型允许存在 `null` 值. 默认情况下, 一个类型 **不能** 接受 `null` 值. 声明可为 null 的类型的方法是, 在类型声明之后明确添加 `?`.

例如:

```
fun main() {
    // neverNull 的类型为: String
    var neverNull: String = "This can't be null"

    // 这里会出现编译器错误
    neverNull = null

    // nullable 的类型为: 可以为 null 的 String
    var nullable: String? = "You can keep a null here"

    // 这是可以的
    nullable = null
```

```
    // 默认情况下, 不能接受 null 值
    var inferredNonNull = "The compiler assumes non-nullable"

    // 这里会出现编译器错误
    inferredNonNull = null

    // notNull 不能接受 null 值
    fun strLength(notNull: String): Int {
        return notNull.length
    }

    println(strLength(neverNull)) // 输出结果为 18
    println(strLength(nullable))  // 这里会出现编译器错误
}
```

> ⚠ `length` 是 String (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/) 类的属性, 它表示字符串中字符的数量.

检查 null 值

你可以在条件表达式中检查 `null` 值. 在下面的示例中, `describeString()` 函数包含一个 `if` 语句, 它检查 `maybeString` 是不是 **非** `null` 值, 并且它的 `length` 是否大于 0:

```
fun describeString(maybeString: String?): String {
    if (maybeString != null && maybeString.length > 0) {
        return "String of length ${maybeString.length}"
    } else {
        return "Empty or null string"
    }
}

fun main() {
    var nullString: String? = null
    println(describeString(nullString))
    // 输出结果为 Empty or null string
}
```

使用安全调用

对于可能包含 `null` 值的对象, 要安全的访问它的属性, 请使用安全调用操作符 `?.`. 如果对象为 `null`, 安全调用操作符会返回 `null`. 如果你想要在你的代码中避免 `null` 值造成的错误, 这个功能会很有用.

在下面的示例中, `lengthString()` 函数使用安全调用, 返回字符串的长度, 或返回 `null` 值:

```
fun lengthString(maybeString: String?): Int? = maybeString?.length

fun main() {
    var nullString: String? = null
    println(lengthString(nullString))
    // null
}
```

> 可以对安全调用使用链式调用, 如果一个对象的任何属性保护 `null` 值, 则会返回 `null`, 而不会抛出错误. 例如:
>
> ```
> person.company?.address?.country
> ```

安全调用操作符也可以用来对扩展函数或成员函数进行安全调用. 这种情况下, 会在调用函数之前进行 null 值检查. 如果检测到 `null` 值, 那么会跳过函数调用, 返回 `null`.

在下面的示例中, `nullString` 是 `null` 值, 因此对 `.uppercase()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/uppercase.html) 的调用会被跳过, 并返回 `null`:

```
fun main() {
    var nullString: String? = null
    println(nullString?.uppercase())
    // 输出结果为 null
}
```

使用 Elvis 操作符

你可以使用 **Elvis 操作符** `?:`, 指定检测到 `null` 值时的默认返回值.

Elvis 操作符的左侧, 是需要检测 `null` 值的表达式. Elvis 操作符的右侧, 是检测到 `null` 值时应该返回的默认值.

在下面的示例中, `nullString` 是 `null` 值, 因此访问 `length` 属性的安全调用返回 `null` 值. 因此 Elvis 操作符的结果是, 返回 `0`:

```
fun main() {
    var nullString: String? = null
    println(nullString?.length ?: 0)
    // 输出结果为 0
}
```

关于 Kotlin 中的 Null 值安全性, 更多详情请参见 Null 值安全性 (Null 值安全性).

实际练习

习题

你有一个 `employeeById` 函数, 可以用来访问一个公司的员工数据库. 但是, 这个函数返回 `Employee?` 类型的值, 因此结果可能为 `null`. 你的目标是编写一个函数, 如果给定了员工 `id`, 则返回员工的工资, 如果在数据库中没有找到这个员工, 则返回 `0`.

```
data class Employee (val name: String, var salary: Int)

fun employeeById(id: Int) = when(id) {
    1 -> Employee("Mary", 20)
    2 -> null
    3 -> Employee("John", 21)
    4 -> Employee("Ann", 23)
    else -> null
}

fun salaryById(id: Int) = // 在这里编写你的代码

fun main() {
    println((1..5).sumOf { id -> salaryById(id) })
}
```

```kotlin
data class Employee (val name: String, var salary: Int)

fun employeeById(id: Int) = when(id) {
    1 -> Employee("Mary", 20)
    2 -> null
    3 -> Employee("John", 21)
    4 -> Employee("Ann", 23)
    else -> null
}

fun salaryById(id: Int) = employeeById(id)?.salary ?: 0

fun main() {
    println((1..5).sumOf { id -> salaryById(id) })
}
```

下一步做什么?

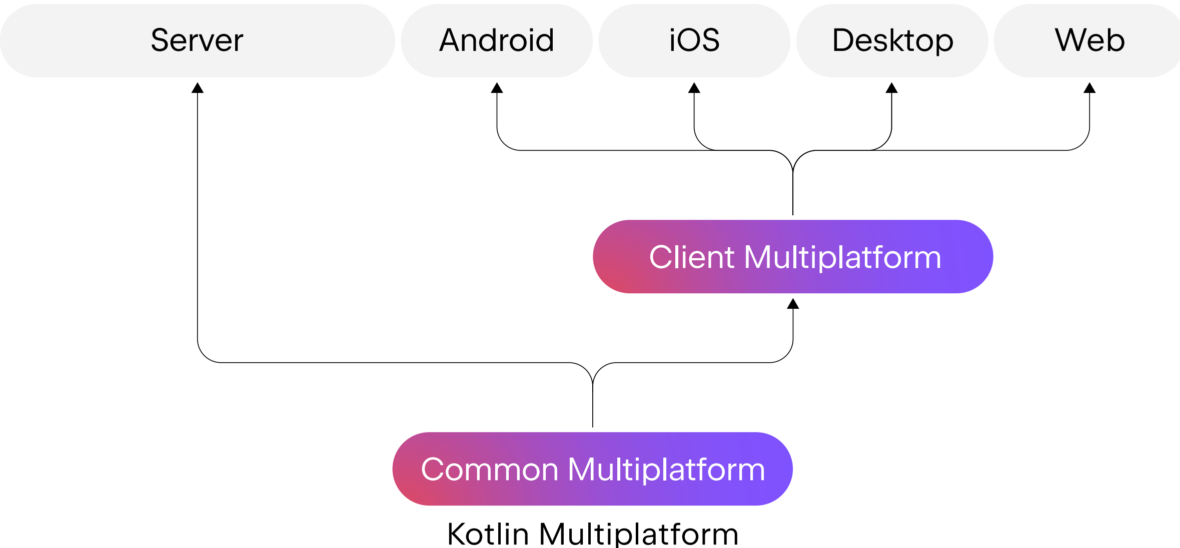
恭喜! 现在你已经完成了我们的 Kotlin 观光之旅, 下面请阅读我们的教程, 看看如何开发流行的 Kotlin 应用程序:

- 创建后端应用程序 (使用 Kotlin 创建 Spring Boot 项目)
- 为 Android 和 iOS 创建跨平台的应用程序 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html)

Kotlin Multiplatform

最终更新: 2024/09/10

Kotlin Multiplatform 技术的设计目的是为了简化跨平台项目的开发工作. 它可以减少对 不同的平台 编写和维护重复代码所耗费的时间, 同时又保持了原生程序开发的灵活性和其他益处.



Kotlin Multiplatform

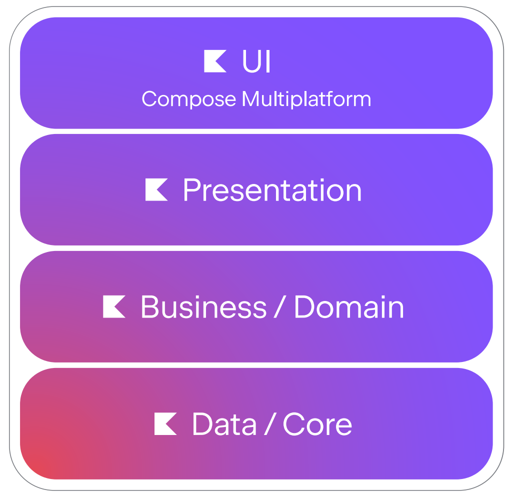
Kotlin Multiplatform 使用场景

Android 和 iOS 应用程序

在不同的移动平台上共用代码, 是 Kotlin 跨平台项目的一个主要使用场景, 通过 Kotlin Multiplatform, 你可以创建跨平台的移动应用程序, 在 Android 和 iOS 项目中共用代码, 实现网络连接, 数据存储, 数据验证, 分析, 计算, 以及其它应用程序逻辑.

请参见 Kotlin Multiplatform 入门 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html) 文档, 以及 使用 Ktor 和 SQLDelight 创建跨平台应用程序 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ktor-sqldelight.html) 教程, 在这个教程中你将创建运行于 Android 和 iOS 的应用程序, 其中包括对这两个平台共用代码的模块.

通过使用 Compose Multiplatform (https://www.jetbrains.com/lp/compose-multiplatform/), JetBrains 开发的、基于 Kotlin 的声明式 UI 框架, 你也可以在 Android 和 iOS 平台共用 UI, 创建完全跨平台的应用程序:



共用不同的模块层和 UI

请参见 Compose Multiplatform 入门 (https://github.com/JetBrains/compose-multiplatform-ios-android-template/#readme) 教程, 创建你自己的、在 Android 和 iOS 平台共用 UI 的移动应用程序.

跨平台库

Kotlin Multiplatform 也可以帮助库的开发者. 你可以为 JVM, Web, 和 Native 平台创建一个跨平台的库, 其中包含共通代码, 以及各个平台相关的实现. 发布之后, 跨平台的库可以作为依赖项, 在其他跨平台项目中使用.

详情请参见 发布跨平台库 (发布跨平台的库).

桌面应用程序

Compose Multiplatform 还可以在不同的桌面平台共用 UI, 例如 Windows, macOS, 以及 Linux. 很多应用程序, 包括 JetBrains Toolbox app (https://blog.jetbrains.com/kotlin/2021/12/compose-multiplatform-toolbox-case-study/), 已经采用了这样的方案.

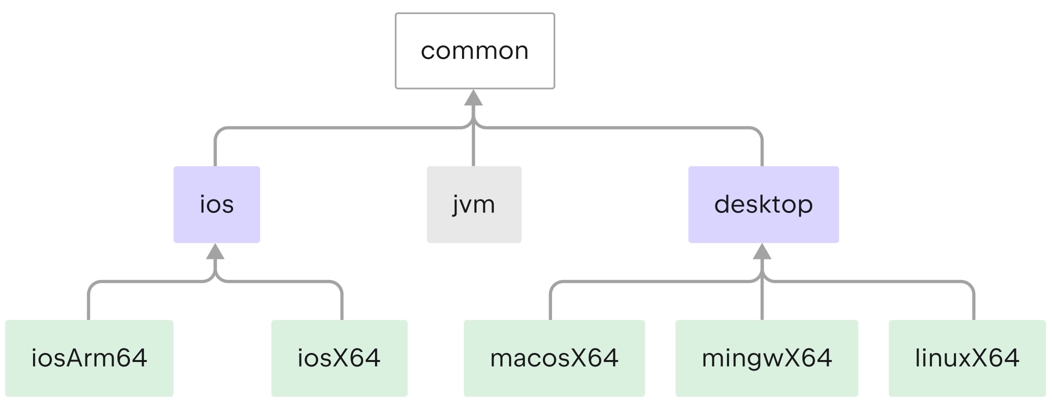
请试用这个 Compose Multiplatform 桌面应用程序 (https://github.com/JetBrains/compose-multiplatform-desktop-template#readme) 模板, 来创建你自己的、在不同的桌面平台共用 UI 的项目.

在不同平台间共用代码

通过 Kotlin Multiplatform, 你可以对 不同的平台 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 维护单一的应用程序逻辑代码库. 你还可以获得原生程序开发的利益, 包括优秀的性能, 以及对平台 SDK 的完全访问能力.

Kotlin 提供了以下代码共用机制:

- 在你的项目中使用的 所有平台 ("在所有平台上共用代码" in "在不同的平台之间共用代码") 共用代码.
- 在你的项目中使用的 一部分平台 ("在类似的平台上共用代码" in "在不同的平台之间共用代码") 共用代码, 这样可以在类似的平台上共用大量代码:



在不同的平台共用代码

- 如果需要在共用代码中访问平台相关的 API, 可以使用 Kotlin 预期声明与实际声明(expected and actual declaration) (预期声明与实际声明) 机制.

入门学习

- 如果你想要使用共通代码创建 iOS 和 Android 应用程序, 请参见 Kotlin Multiplatform 入门 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html)
- 如果你想要针对其他平台创建应用程序或库, 请参见 共用代码的原则与示例 (在不同的平台之间共用代码)

如果你是 Kotlin 新手, 请先阅读 Kotlin 入门 (Kotlin 入门)

示例项目

请参见 跨平台应用程序示例 (https://www.jetbrains.com/help/kotlin-multiplatform-

dev/multiplatform-samples.html), 来理解 Kotlin Multiplatform 的工作方式.

使用 Kotlin 进行服务器端开发

最终更新: 2024/09/10

Kotlin 非常适合于开发服务器端应用程序, 使用 Kotlin 可以编写出简洁高效的代码, 同时又可以完全兼容既有的 Java 技术栈(Java-based technology stacks), 而且其学习曲线比较平滑:

- **表达能力**: Kotlin 拥有许多创造性的语言特性, 比如它支持 类型安全的构建器(type-safe builder) (类型安全的构建器) 以及 委托属性(delegated property) (委托属性), 可以帮助你构造出强大而且易用的抽象层.
- **伸缩性**: Kotlin 对 协程(coroutine) (协程(Coroutine)) 的支持可以帮助你构建出性能强大的服务器端应用程序, 能够为巨量用户提供服务, 但只要求很低的硬件配置.
- **互操作性**: Kotlin 完全兼容于所有基于 Java 的框架(framework), 因此你既可以享受一个更加现代的语言带来的利益, 同时又可以继续使用你熟悉的技术栈.
- **可移植性**: 对于大规模的 Java 代码库, Kotlin 语言支持平滑地, 逐步的迁移. 你可以只使用 Kotlin 来编写新代码, 同时对系统中既有的部分继续沿用旧的 Java 代码.
- **开发工具**: 除了 IDE 的支持之外, 在 IntelliJ IDEA Ultimate 的插件中, Kotlin 还提供了针对特定框架(比如, Spring)的开发工具支持.
- **学习曲线**: 对于 Java 开发者, Kotlin 是非常易于学习的. Kotlin 插件中包含了 Java 代码到 Kotlin 代码的自动转换器, 可以帮助你完成最初的工作. Kotlin Koan (Kotlin Koan) 中有一系列的交互式练习题, 可以指导你学习 Kotlin 语言的关键特性.

Kotlin 服务器端开发的一些相关框架

下面是 Kotlin 服务器端框架的一些例子:

- Spring (https://spring.io) 从 5.0 版本开始, 使用 Kotlin 的语言特性实现了 更加简洁的 API (https://spring.io/blog/2017/01/04/introducing-kotlin-support-in-spring-framework-5-0). 在线工程生成器 (https://start.spring.io/#!language=kotlin) 可以帮助你使用 Kotlin 语言快速生成新的工程.
- Ktor (https://github.com/kotlin/ktor) 是 JetBrains 公司开发的框架, 用 Kotlin 来开发 Web 应用程序, 使用协程(coroutine)实现了高度伸缩性, 并提供了易用而且符合习惯的 API.

- Quarkus (https://quarkus.io/guides/kotlin) 对使用 Kotlin 提供了一级支持. 这个框架是开源的, 由 Red Hat 维护. Quarkus 是为 Kubernetes 全新构建的, 并利用数百种精选的库, 提供了一个整合的全栈框架.
- Vert.x (https://vertx.io), 一个创建基于 JVM 的交互式 Web 应用程序的框架, 对 Kotlin 提供了专门支持 (https://github.com/vert-x3/vertx-lang-kotlin), 包含 完整的文档 (https://vertx.io/docs/vertx-core/kotlin/).
- kotlinx.html (https://github.com/kotlin/kotlinx.html) 是一种 DSL, 可用于在 Web 应用程序中构建 HTML. 可用来替代传统的模板系统, 比如 JSP 和FreeMarker.
- Micronaut (https://micronaut.io/) 是一个现代化的, 基于 JVM 的全栈框架, 用于创建模块化的, 便于测试的微服务(microservice)或无服务(serverless)应用程序. 它带有很多内建的, 便利的功能.
- http4k (https://http4k.org/) 是一个尺寸很小的工具包, 用于 Kotlin HTTP 应用程序, 使用纯 Kotlin 编写. 这个库基于 Twitter 的 "通过函数实现你的服务器" 论文, 它将 HTTP 服务器端和客户端模型都表达为简单的 Kotlin 函数, 再将这些简单函数组合在一起.
- Javalin (https://javalin.io) 是一个用于 Kotlin 和 Java的, 非常轻量的 web 框架, 支持 WebSockets, HTTP2 以及异步请求.
- 关于数据的持久化存储, 可以选择直接的 JDBC 访问, 或者使用 JPA, 或者通过 Java 驱动程序使用 NoSQL 数据库. 对于 JPA, kotlin-jpa 编译器插件 ("JPA 支持" in "No-arg 编译器插件") 可以使 Kotlin 编译的 class文件符合 JPA 框架的要求.

> 你可以在 https://kotlin.link/ (https://kotlin.link/resources) 找到更多框架.

发布 Kotlin 服务器端应用程序

Kotlin 应用程序可以发布到任何支持 Java Web 应用程序的主机上, 包括 Amazon Web Services, Google Cloud Platform, 以及其他等等.

要在 Heroku (https://www.heroku.com) 上发布 Kotlin 应用程序, 你可以参照 Heroku 官方教程 (https://devcenter.heroku.com/articles/getting-started-with-kotlin).

AWS Labs 提供了一个 示例工程 (https://github.com/awslabs/serverless-photo-recognition), 演示如何使用 Kotlin 来编写 AWS Lambda (https://aws.amazon.com/lambda/) 函数.

Google 云平台也提供了一系列教程, 演示如何将 Kotlin 应用程序发布到 Google 云平台上, 包括在 Google App Engine 上运行 Kotlin Ktor 应用程序

(https://cloud.google.com/community/tutorials/kotlin-ktor-app-engine-java8) 和 在 Google App Engine 上运行 Kotlin Spring 应用程序 (https://cloud.google.com/community/tutorials/kotlin-springboot-app-engine-java8). 此外还有一篇 向导式代码文档 (https://codelabs.developers.google.com/codelabs/cloud-spring-cloud-gcp-kotlin) 介绍如何发布 Kotlin Spring 应用程序.

使用 Kotlin 进行服务端开发的产品

Corda (https://www.corda.net/) 是一个开源的分布式帐务平台, 受各大主要银行支持, 完全使用 Kotlin 语言开发.

JetBrains Account (https://account.jetbrains.com/), 这个系统负责 JetBrains 公司所有的许可证销售和验证过程, 系统 100% 使用 Kotlin 编写, 自 2015 年起运行在生产环境中, 未发生任何严重问题.

下一步

- 关于对 Kotlin 语言的更加深入介绍, 请本站阅读所有 Kotlin 文档, 以及 Kotlin Koan (Kotlin Koan).
- 观看网络研讨会 "使用 Kotlin 和 Micronaut 开发微服务(microservice)" (https://micronaut.io/2020/12/03/webinar-micronaut-for-microservices-with-kotlin/), 并阅读更详细的 向导 (https://guides.micronaut.io/latest/micronaut-kotlin-extension-fns.html), 这篇向导会介绍如何在 Micronaut framework 中使用 Kotlin 扩展函数 ("扩展函数 (Extension Function)" in "扩展").
- http4k 提供了 命令行工具(CLI) (https://toolbox.http4k.org) 来生成完整的项目, 还提供了一个 starter (https://start.http4k.org) 代码仓库, 可以通过一个简单的 bash 命令来生成整个持续部署流程(CD pipeline) (使用 GitHub, Travis, 以及 Heroku).
- 想要从 Java 迁移到 Kotlin 吗? 请阅读 在 Java 和 Kotlin 使用字符串的常见情况 (Java 和 Kotlin 中的字符串).

使用 Kotlin 进行 Android 开发

最终更新: 2024/09/10

2019 年的 Google I/O 大会宣布, Kotlin 成为 Android 移动应用开发的首选语言 (https://developer.android.com/kotlin/first).

超过 50% 的专业 Android 开发者使用 Kotlin 作为主要开发语言, 而使用 Java 作为主要语言的只有 30%. 使用 Kotlin 作为主要语言的开发者中, 70% 表示 Kotlin 提高了他们的生产性.

使用 Kotlin 进行 Android 开发, 你将获得以下益处:

- **代码更少, 可读性更好**. 编写代码耗费的时间, 以及理解他人代码耗费的时间, 都变得更少.
- **更少发生常见错误**. 根据 Google 的内部数据 (https://medium.com/androiddevelopers/fewer-crashes-and-more-stability-with-kotlin-b606c6a6ac04), 使用 Kotlin 开发的 App, 崩溃概率要低 20%.
- **Jetpack 库对 Kotlin 的支持**. Jetpack Compose (https://developer.android.com/jetpack/compose) 是 Android 推荐的现代化工具包, 可以使用 Kotlin 来构建原生 UI. KTX 扩展 (https://developer.android.com/kotlin/ktx) 为既有的 Android 库添加了 Kotlin 语言特性, 比如协程(Coroutine), 扩展函数(extension function), Lambda 表达式, 以及命名参数(named parameter).
- **支持跨平台(multiplatform)开发**. 使用 Kotlin Multiplatform, 不仅可以开发 Android 应用程序, 而且还可以开发 iOS (https://kotlinlang.org/lp/multiplatform/) 应用程序, 后端服务, 以及 Web 应用程序. 有些 Jetpack 库 (https://developer.android.com/kotlin/multiplatform) 已经可以支持跨平台(multiplatform). Compose Multiplatform (https://www.jetbrains.com/lp/compose-multiplatform/) 是 JetBrains 的声明式 UI 框架, 它基于 Kotlin 和 Jetpack Compose, 可以在多个平台共用 UI – iOS, Android, Desktop, 以及 Web.
- **成熟的语言和开发环境**. 自从 2011 诞生以来, Kotlin 作为一种语言, 以及它的整个生态系统, 包括各种强大的工具, 一切都在不断发展. 现在它已经无缝集成到 Android Studio (https://developer.android.com/studio) 中, 并且被许多公司在 Android 应用程序开发中积极使用.
- **与 Java 的互操作性**. 你可以在应用程序中同时使用 Kotlin 和 Java 语言, 而不必将你的代码全部迁移到 Kotlin.

- **易于学习**. Kotlin 非常易于学习, 尤其是对于 Java 开发者.
- **活跃的开发社区**. Kotlin 的开发社区提供了很多支持和贡献, 而且开发社区还在全世界范围内不断壮大. 领先的 1000 个 Android 应用程序中, 超过 95% 都是使用 Kotlin 开发的.

许多初创公司, 以及财富 500 强公司都使用 Kotlin 开发了 Android 应用程序, 详情请参见 面向 Android 开发者的 Google 网站 (https://developer.android.com/kotlin/stories).

要开始使用 Kotlin, 你可以阅读以下资料:

- 关于 Android 开发, 请阅读 Google 关于使用 Kotlin 开发 Android App 的文档 (https://developer.android.com/kotlin/get-started).
- 关于跨平台移动应用程序开发, 请阅读 针对 Android 和 iOS 的 Kotlin Multiplatform 入门 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html).

使用 Kotlin 进行 Wasm 开发

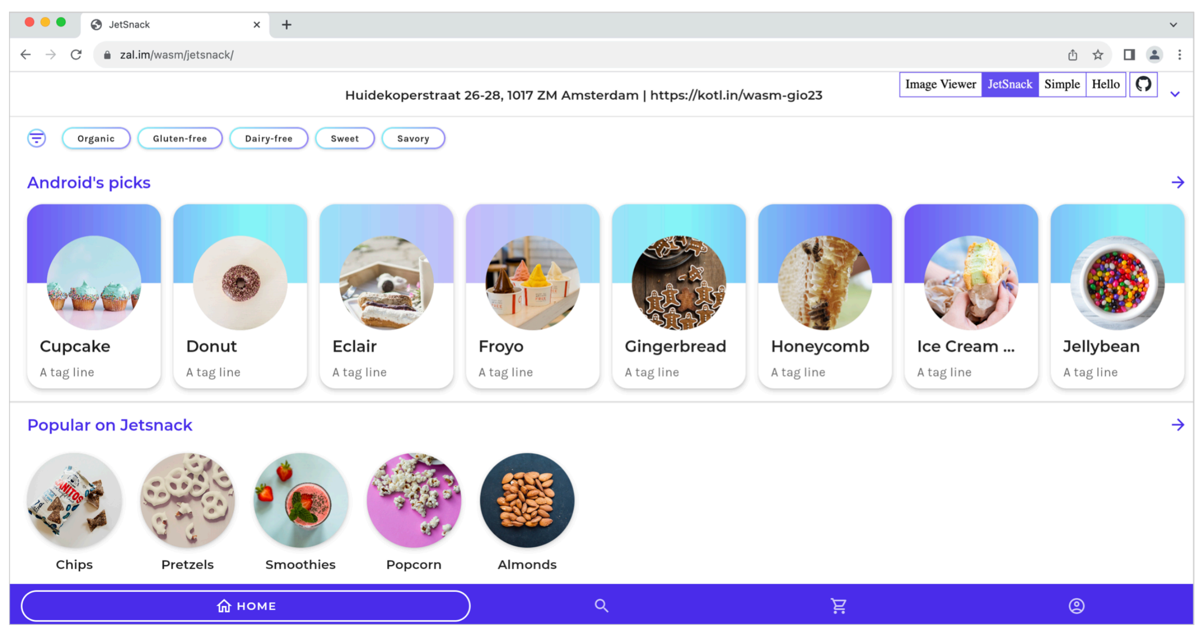
最终更新: 2024/09/10

> Kotlin Wasm 目前处于 Alpha 阶段 (Kotlin 各部分组件的稳定性). 它随时有可能变更. 你可以将它用于正式产品之前的各种场景. 希望你能通过我们的 问题追踪系统 (https://kotl.in/issue) 提供你的反馈意见.
>
> 加入 Kotlin/Wasm 社区 (https://slack-chats.kotlinlang.org/c/webassembly).

Kotlin 能够构建应用程序, 并通过 Compose Multiplatform 和 Kotlin/Wasm, 将移动应用和桌面应用中的用户界面(UI) 重用在你的 Web 项目中.

Compose Multiplatform (https://www.jetbrains.com/lp/compose-multiplatform/) 是一个声明式框架, 基于 Kotlin 和 Jetpack Compose (https://developer.android.com/jetpack/compose), 通过它, 你可以一次性实现你的 UI, 然后在你的所有目标平台上共用 UI. 针对 Web 平台, Compose Multiplatform 使用 Kotlin/Wasm 作为编译目标.

查看我们在线演示, 这是一个使用 Compose Multiplatform 和 Kotlin/Wasm 构建的应用程序 (https://zal.im/wasm/jetsnack/)



Kotlin/Wasm 演示

> ⚠ 要在浏览器中运行 Kotlin/Wasm 构建的应用程序, 你需要使用支持新的垃圾收集和异常处理协议的浏览器版本. 关于各浏览器目前的支持状态, 请参见 WebAssembly 路线图 (https://webassembly.org/roadmap/).

WebAssembly (Wasm) (https://webassembly.org/) 是一种二进制指令格式, 用于基于堆栈 (stack-based)的虚拟机. 这种格式是平台独立的, 因为它运行在自己的虚拟机上. Wasm 为 Kotlin 和其他编程语言提供了在 Web 上运行的编译目标.

Kotlin/Wasm 会将你的 Kotlin 代码编译为 Wasm 格式. 使用 Kotlin/Wasm, 你可以创建应用程序, 运行在不同的环境和设备上, 只要这些环境和设备支持 Wasm, 并满足 Kotlin 的要求.

你想自己尝试一下吗?

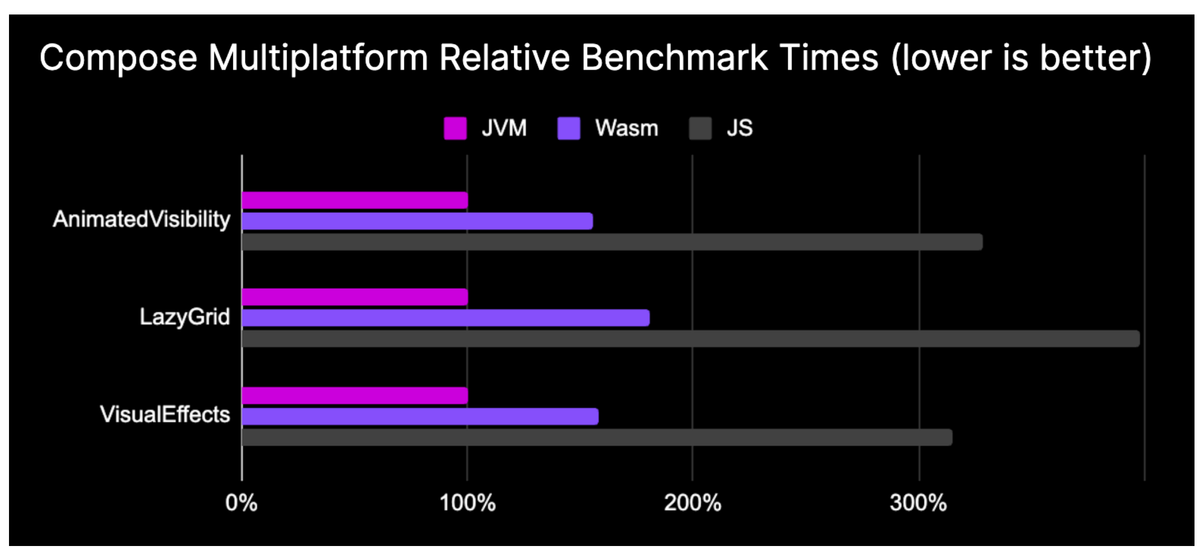
Get started with Kotlin/Wasm →

Kotlin/Wasm 入门

(使用 IntelliJ IDEA 开发 Kotlin/Wasm 入门)

Kotlin/Wasm 的性能

尽管 Kotlin/Wasm 还处于 Alpha 阶段，但在 Kotlin/Wasm 上运行的 Compose Multiplatform 已经表现出令人鼓舞的性能特性. 你可以看到, 它的执行速度超过了 JS, 接近于 JVM:



Kotlin/Wasm 的性能

我们会定期的在 Kotlin/Wasm 上运行基准测试(benchmark), 上面的结果来自我们在最新版本的 Google Chrome 上运行的测试结果.

对浏览器 API 的支持

Kotlin/Wasm 的标准库提供了浏览器 API 的声明 (包括 DOM API). 通过这些声明, 你可以直接使用 Kotlin API 来访问和使用浏览器的各种功能. 例如, 在你的 Kotlin/Wasm 应用程序中, 你可以操作 DOM 元素, 或使用 fetch API, 而不需要从头开始定义这些声明. 更多详情请参见我们的 Kotlin/Wasm 浏览器示例 (https://github.com/Kotlin/kotlin-wasm-examples/tree/main/browser-example).

用于支持浏览器 API 的声明是通过 JavaScript 互操作能力 (与 JavaScript 交互) 来定义的. 你可以使用同样的功能来定义你自己的声明. 此外, Kotlin/Wasm 与 JavaScript 互操作能力还允许你在 JavaScript 中使用 Kotlin 代码. 详情请参见 在 JavaScript 中使用 Kotlin 代码 ("在 JavaScript 中使用 Kotlin 代码" in "与 JavaScript 交互").

留下你的意见反馈

Kotlin/Wasm 意见反馈

- Slack: 在我们的 #webassembly (https://kotlinlang.slack.com/archives/CDFP59223) 频道, 直接向开发者提供你的意见反馈. 获得 Slack 邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up).
- 在 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-56492) 中报告问题.

Compose Multiplatform 意见反馈

- Slack: 在 #compose-web (https://slack-chats.kotlinlang.org/c/compose-web) 公开频道, 提供你的意见反馈.
- 在 GitHub 报告问题 (https://github.com/JetBrains/compose-multiplatform/issues).

更多信息

- 在这个 YouTube 播放列表 (https://kotl.in/wasm-pl) 中, 学习 Kotlin/Wasm 的更多信息.

* 在我们的 GitHub 仓库中查看 Kotlin/Wasm 示例 (https://github.com/Kotlin/kotlin-wasm-examples).

使用 Kotlin/Native 进行原生(Native)程序开发

最终更新: 2024/09/10

Kotlin/Native 是一种代码编译技术, 可以将 Kotlin 代码编译为原生二进制代码(native binary), 脱离 VM 运行. 它包含一个基于 LLVM (https://llvm.org/) 的后端, 用于编译 Kotlin 源代码, 以及一个原生代码实现的 Kotlin 运行库.

为什么要使用 Kotlin/Native?

Kotlin/Native 的主要设计目的是, 用来编译 Kotlin 代码, 使其能够运行在那些不应该使用 *虚拟机*, 或无法使用 *虚拟机* 的平台上, 比如嵌入式设备, 或 iOS. 它适合用于帮助开发者生成完整独立的, 不依赖于额外运行库和虚拟机的独立程序.

目标平台

Kotlin/Native 支持以下平台:

- macOS
- iOS, tvOS, watchOS
- Linux
- Windows (MinGW)
- Android NDK

> 要编译到 Apple 平台的编译目标, macOS, iOS, tvOS, 和 watchOS, 你需要安装 Xcode (https://apps.apple.com/us/app/xcode/id497799835) 以及它的命令行工具.

请参见所有支持的目标平台 (Kotlin/Native 支持的目标平台).

互操作性

Kotlin/Native 支持与各种操作系统的原生编程语言之间的双向互操作. 编译器会创建:

- 各种 平台 的可执行文件
- 静态库, 或 动态 (教程 - 使用 Kotlin/Native 开发动态库) 库, 以及供 C/C++ 项目使用的 C 头文件
- 供 Swift 和 Objective-C 项目使用的 Apple 框架 (教程 - 使用 Kotlin/Native 开发 Apple Framework)

Kotlin/Native 也支持在 Kotlin/Native 源代码中直接使用既有的库:

- 静态或动态的 C 库 (与 C 代码交互)
- C, Swift 和 Objective-C (与 Swift/Objective-C 代码交互) 框架

在既有的 C, C++, Swift, Objective-C 和其他语言的项目中, 可以很容易地包含编译后的 Kotlin 代码. 在 Kotlin/Native 代码中, 也可以很容易地直接使用既有的原生代码, 静态或动态的 C 库 (与 C 代码交互), Swift/Objective-C 框架 (与 Swift/Objective-C 代码交互), 图形引擎, 以及其他任何东西.

Kotlin/Native 的 库 (平台库) 可以帮助你在多个项目中共享 Kotlin 代码. POSIX, gzip, OpenGL, Metal, Foundation, 以及其他许多流行的库和 Apple 框架, 都已预先导入为 Kotlin/Native 库形式, 包含在编译器的包中了.

在不同的平台上共享代码

Kotlin Multiplatform (Kotlin Multiplatform) 可以帮助你在多个不同的平台上共用代码, 包括 Android, iOS, JVM, Web, 以及原生平台. 跨平台库为共通的 Kotlin 代码提供了必要的 API, 帮助我们用 Kotlin 代码编写项目中共通的部分, 这些代码只需要编写一次.

你可以通过 Kotlin Multiplatform 入门 (Kotlin 跨平台程序开发入门) 教程来创建应用程序, 并在 iOS 和 Android 平台共用业务逻辑. 要在 iOS, Android, Desktop, 以及 Web 平台上共用 UI, 请试用 Compose Multiplatform (https://www.jetbrains.com/lp/compose-multiplatform/), JetBrains 基于 Kotlin 和 Jetpack Compose (https://developer.android.com/jetpack/compose) 开发的声明式 UI 框架.

如何入门

如果你是 Kotlin 新手, 请先阅读 Kotlin 入门 (Kotlin 入门).

推荐文档:

- Kotlin Multiplatform 入门 (Kotlin 跨平台程序开发入门)

* 与 C 代码交互 (与 C 代码交互)
* 与 Swift/Objective-C 代码交互 (与 Swift/Objective-C 代码交互)

推荐教程:

* Kotlin/Native 入门 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA)
* Kotlin Multiplatform 入门教程 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html)
* 映射 C 语言的基本数据类型 (教程 - 映射 C 语言的基本数据类型)
* 使用 Kotlin/Native 开发动态链接库 (教程 - 使用 Kotlin/Native 开发动态库)
* 使用 Kotlin/Native 开发 Apple 框架 (教程 - 使用 Kotlin/Native 开发 Apple Framework)

使用 Kotlin 进行 JavaScript 开发

最终更新: 2024/09/10

Kotlin/JS 能够将你的 Kotlin 代码, Kotlin 标准库, 以及所有兼容的依赖项转换为 JavaScript. Kotlin/JS 目前的实现针对 ES5 (https://www.ecma-international.org/ecma-262/5.1/) 标准.

推荐的方式是通过 `kotlin.multiplatform` Gradle 插件来使用 Kotlin/JS. 这个插件可以帮助你便利的集中配置和控制针对 JavaScript 的 Kotlin 项目. 其中包含了很多必要的功能, 比如控制你的应用程序的打包(Bundling), 通过 npm 直接添加 JavaScript 依赖项, 等等. 关于具体的选项, 详情请参见设置 Kotlin/JS 项目 (创建 Kotlin/JS 工程(Project)).

Kotlin/JS IR 编译器

Kotlin/JS IR 编译器 (使用 IR 编译器) 与旧的默认编译器相比, 带来了许多改进. 比如, 它通过死代码消除, 改善了生成的可执行文件的尺寸, 而且与 JavaScript 生态系统的交互变得更加顺畅.

> 从 Kotlin 1.8.0 开始, 旧的编译器已被废弃.

通过从 Kotlin 代码生成 TypeScript 声明文件 (`d.ts`), IR 编译器使得 "混合(hybrid)" 应用程序的开发更加容易, 这种应用程序可以混合 TypeScript 和 Kotlin 代码, 也可以使用 Kotlin 跨平台项目来共用代码.

关于 Kotlin/JS IR 编译器的功能, 以及如何在项目中使用, 更多详情请参见 Kotlin/JS IR 编译器文档 (使用 IR 编译器) 以及 迁移向导 (将 Kotlin/JS 项目迁移到 IR 编译器).

Kotlin/JS 框架

现代的 Web 开发通过使用各种框架得到很大的益处, 框架可以简化 Web 应用程序的创建. 下面是可用于 Kotlin/JS 的一些流行的 Web 框架的例子:

KVision

KVision 是一个面向对象的 Web 框架, 可以使用 Kotlin/JS 编写应用程序, 可以通过各种现成的组件来组合成你的用户界面. 你可以使用响应式(reactive)或命令式(imperative)编程模式来创建你的前端, 然后使用 connector for Ktor, Spring Boot, 以及其他框架, 与你的服务端应用程序连接, 并使用 Kotlin 跨平台程序 (Kotlin Multiplatform) 来共用代码.

请 访问 KVision 网站 (https://kvision.io), 查看 KVision 的文档, 教程, 和示例.

关于框架的更新和讨论, 请加入 Kotlin Slack (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up) 的 #kvision (https://kotlinlang.slack.com/messages/kvision) 和 #javascript (https://kotlinlang.slack.com/archives/C0B8L3U69) 频道.

fritz2

fritz2 是一个独立的框架, 用于构建响应式(reactive) Web 用户界面. 它提供了自己的类型安全的 DSL 来构建和渲染 HTML 元素, 它使用 Kotlin 的协程(Coroutine)和数据流(Flow)来表达 UI 组件及其数据绑定. 它提供了状态管理, 校验, 路由, 以及很多其他功能, 并与 Kotlin 跨平台项目集成.

请 访问 fritz2 网站 (https://www.fritz2.dev), 查看 fritz2 的文档, 教程, 和示例.

关于框架的更新和讨论, 请加入 Kotlin Slack (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up) 的 #fritz2 (https://kotlinlang.slack.com/messages/fritz2) 和 #javascript (https://kotlinlang.slack.com/archives/C0B8L3U69) 频道.

Doodle

Doodle 是一个用于 Kotlin/JS 的基于矢量的(vector-based) UI 框架. Doodle 应用程序使用浏览器的图形功能来描绘用户界面, 而不是依赖于 DOM, CSS, 和 Javascript. 使用这种方法, Doodle 使你能够精确描绘任何 UI 元素, 矢量图形, 梯度(gradient), 以及定制的可视化图形.

请 访问 Doodle 网站 (https://nacular.github.io/doodle/), 查看 Doodle 的文档, 教程, 和示例.

关于框架的更新和讨论, 请加入 Kotlin Slack (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up) 的 #doodle (https://kotlinlang.slack.com/messages/doodle) 和 #javascript (https://kotlinlang.slack.com/archives/C0B8L3U69) 频道.

加入 Kotlin/JS 开发者社区

你可以加入官方 Kotlin Slack (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up) 的 #javascript (https://kotlinlang.slack.com/archives/C0B8L3U69) 频道, 与开发者社区和开发团队交谈.

在数据科学(Data Science)中使用 Kotlin

最终更新: 2024/09/10

无论是创建数据管道(data pipeline), 还是构建真实生产环境的机器学习模型(machine learning model), Kotlin 都可以是很好的数据处理工具:

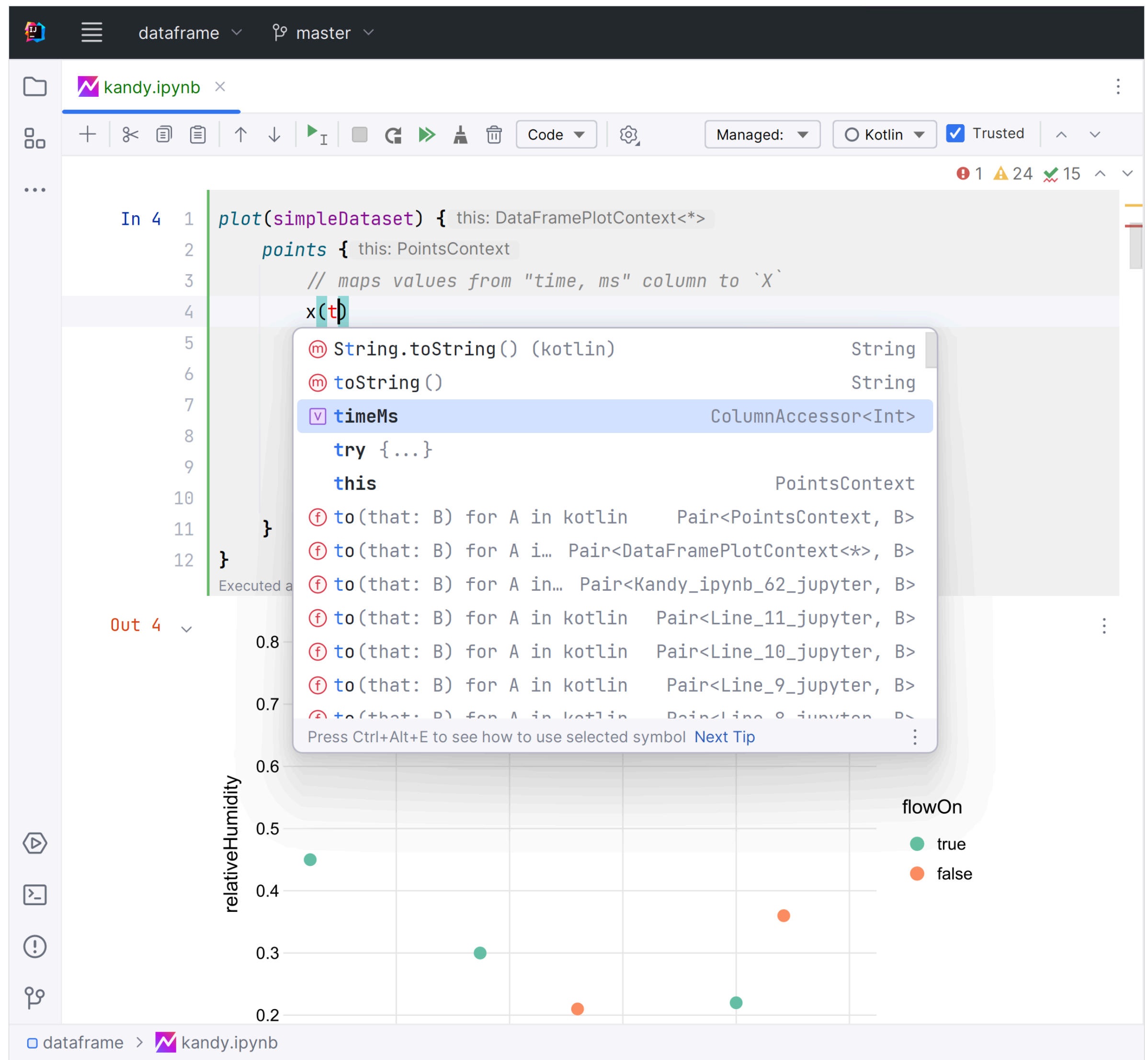
- Kotlin 代码简洁, 易读, 而且易于学习.
- 静态类型系统, 以及 null 值安全性, 有助于创建可靠, 易于维护的代码, 而且易于追中错误.
- Kotlin 是基于 JVM 平台的编程语言, 因此提供了非常好的运行性能, 并且可以灵活运用整个 Java 生态环境, 包括所有那些经过长期广泛使用的 Java 库.

交互式编辑器

Notebook, 比如 Kotlin Notebook (https://plugins.jetbrains.com/plugin/16340-kotlin-notebook), Jupyter Notebook (https://jupyter.org/), 以及 Datalore (http://jetbrains.com/datalore) 提供了许多便利的工具, 用来可视化数据, 以及探索研究. Kotlin 与这些工具集成, 可以帮助你研究数据, 将你的发现与同事共享, 逐渐提升你的数据科学和机器学习技能.

Kotlin Notebook

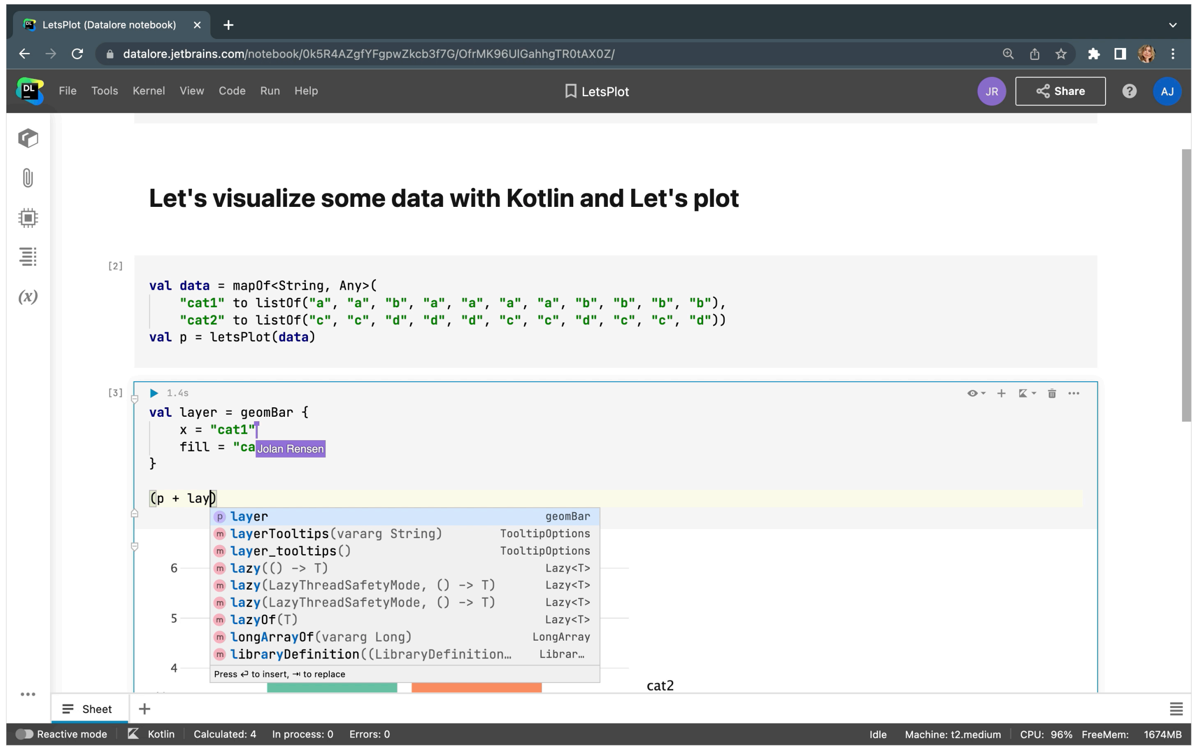
Kotlin Notebook (https://plugins.jetbrains.com/plugin/16340-kotlin-notebook) 是一个 IntelliJ IDEA plugin, 可以使用 Kotlin 来创建 notebook. 它利用了 Kotlin Kernel 来执行各个单元(cell), 并利用强大的 Kotlin IDE 支持, 实现实时的代码查看. 它现在是 Kotlin notebook 开发的推荐方式. 更多详情, 请阅读我们的 Blog (https://blog.jetbrains.com/kotlin/2023/07/introducing-kotlin-notebook/).



Kotlin Notebook

Datalore 的 Kotlin Notebooks

通过 Datalore, 你可以直接 在浏览器中使用 Kotlin, 不需要额外安装. 你还可以通过 Kotlin notebooks 实时的协作, 编写代码时得到智能的代码辅助, 以及通过交互式报告或静态报告共享结果. 请参见 示例报告 (https://datalore.jetbrains.com/view/report/9YLrg20eesVX2cQu1FKLiZ).

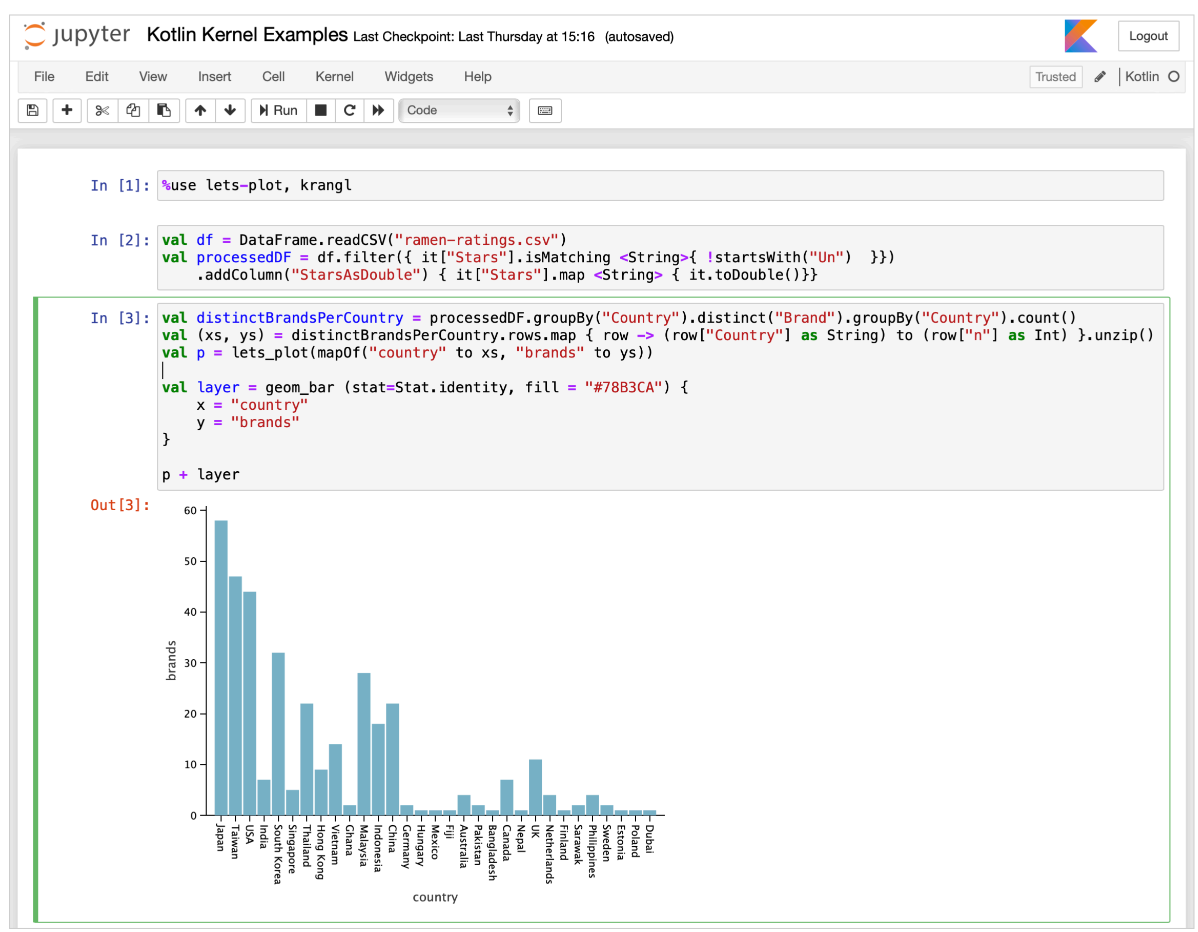


Kotlin in Datalore

注册免费的 Datalore 社区帐号来使用 Kotlin (https://datalore.jetbrains.com/).

Jupyter 的 Kotlin Kernel

Jupyter Notebook 是一个开源的 Web 应用程序, 你可以用来创建和分享文档(也称为 "notebook"), 其中包含代码, 可视化的数据, 以及 Markdown 格式的文本. Kotlin-jupyter (https://github.com/Kotlin/kotlin-jupyter) 是一个开源项目, 可以在 Jupyter Notebook 中添加对 Kotlin 的支持.



Kotlin in Jupyter notebook

关于 Kotlin Kernel 的安装指南, 文档, 以及示例, 请查看它的 GitHub 代码库 (https://github.com/Kotlin/kotlin-jupyter).

库

Kotlin 社区开发了针对数据处理任务的各种库, 并在迅速扩大. 下面这些库可能会对你很有用:

Kotlin 库

- Kotlin DataFrame (https://github.com/Kotlin/dataframe) 是一个结构化数据处理的库. 它既利用 Kotlin 语言的所有威力, 又利用 Jupyter notebook 和 REPL 中的间断性代码执行能力, 以图调和 Kotlin 的静态类型和数据的动态性质之间的冲突.
- Kandy (https://kotlin.github.io/kandy/welcome.html) 是一个使用 Kotlin 开发的开源的 JVM 绘图库. 它提供了用于创建 Chart 的强大而且灵活的 DSL, 以及与 Kotlin Notebook (https://plugins.jetbrains.com/plugin/16340-kotlin-notebook) 和 Kotlin DataFrame (https://kotlin.github.io/dataframe/gettingstarted.html) 的无缝集成.

- Multik (https://github.com/Kotlin/multik): Kotlin 编写的多维数组库. 这个库提供符合 Kotlin 语言习惯的, 类型安全并且维度安全的 API, 可对多维数组进行数学操作. Multik 提供了基于 JVM 和基于原生代码的计算引擎, 可相互替换, 以及两种引擎的组合, 用于性能优化.

- KotlinDL (https://github.com/jetbrains/kotlindl) 是一个高级的深度学习 API, 用 Kotlin 编写, 受 Keras 启发. 它提供了简单的 API, 可用于从头开始训练深度学习模型, 导入既有的 Keras 模型用于推断, 以及利用迁移学习(transfer learning)调节既有的预先训练的模型, 供你的任务使用.

- Kotlin for Apache Spark (https://github.com/JetBrains/kotlin-spark-api) 补足了 Kotlin 和 Apache Spark 之间缺少的兼容层. 它可以让 Kotlin 开发者使用熟悉的语言功能特性, 比如数据类, 以及在大括号或方法引用中将 Lambda 表达式用作简单表达式.

- kmath (https://github.com/mipt-npm/kmath) 是一个实验性的库, 最初受 NumPy (https://numpy.org/) 的启发, 但它演化为更加灵活的抽象层次. 它实现了 Kotlin 类型的代数结构组合的数学操作, 为线性结构(linear structure), 表达式, 直方图(histogram), 流运算 (streaming operation) 定义了 API, 对既有的 Java 和 Kotlin 库提供了可互换的包装, 包括 ND4J (https://github.com/eclipse/deeplearning4j/tree/master/nd4j), Commons Math (https://commons.apache.org/proper/commons-math/), Multik (https://github.com/Kotlin/multik), 以及其它库.

- lets-plot (https://github.com/JetBrains/lets-plot) 是一个 Kotlin 编写的的库, 用于统计数据绘图. Lets-Plot 是一个跨平台库, 不仅可用于 JVM 平台, 而且可用于 JS 和 Python 平台.

- kravis (https://github.com/holgerbrandl/kravis) 是受 R 语言的 ggplot (https://ggplot2.tidyverse.org/) 启发产生的库, 用于表格数据(tabular data)的可视化.

- londogard-nlp-toolkit (https://github.com/londogard/londogard-nlp-toolkit/) 是一个工具库, 用于自然语言处理(Natural Language Processing), 比如 字(word)/子字(subword)/语句 (sentence) 嵌入(embedding), 字频统计(word-frequency), 终止字(stopword), 词干 (stemming), 等等.

Java 库

由于 Kotlin 对与 Java 的交互功能提供了一级支持, 因此你也可以在你的 Kotlin 代码中使用 Java 库来进行数据处理. 以下是这些 Java 库的一些例子:

- DeepLearning4J (https://deeplearning4j.konduit.ai) - 针对 Java 的深度学习(deep learning) 库

- ND4J (https://github.com/eclipse/deeplearning4j/tree/master/nd4j) - 用于 JVM 平台的高效率矩阵数学库

- Dex (https://github.com/PatMartin/Dex) - 基于 Java 的数据可视化工具
- Smile (https://github.com/haifengl/smile) - 一个非常全面的系统, 包括机器学习, 自然语言处理, 线性代数, 图, 插值, 可视化. 除 Java API 外, Smile 还提供非常便利的 Kotlin API (https://haifengl.github.io/api/kotlin/index.html), 以及 Scala 和 Clojure API.
  - Smile-NLP-kt (https://github.com/londogard/smile-nlp-kt) - 针对 Smile 的自然语言处理部分的 Scala 实现, 提供 Kotlin 重写的扩展函数和接口.
- Apache Commons Math (https://commons.apache.org/proper/commons-math/) - 一个通用的 Java 库, 包括数学, 统计, 以及机器学习
- NM Dev (https://nm.dev/) - Java 数学库, 包含了所有的经典数学运算.
- OptaPlanner (https://www.optaplanner.org/) - 针对最优规划问题(optimization planning problem)的工具库
- Charts (https://github.com/HanSolo/charts) - 用于科学计算的 JavaFX 图表库, 正在开发中
- Apache OpenNLP (https://opennlp.apache.org/) - 一个基于机器学习的工具库 , 用于自然语言处理
- CoreNLP (https://stanfordnlp.github.io/CoreNLP/) - 一个自然语言处理工具库
- Apache Mahout (https://mahout.apache.org/) - 一个用于回归(regression), 聚类(clustering), 以及推荐(recommendation)的分布式框架
- Weka (https://www.cs.waikato.ac.nz/ml/index.html) - 用于数据挖掘任务的一组机器学习算法
- Tablesaw (https://github.com/jtablesaw/tablesaw) - 一个 Java 数据框架. 包含基于 Plot.ly 的可视化库.

在编程竞赛(Competitive Programming)中使用 Kotlin

最终更新: 2024/09/10

本教程针对的读者是以前未使用过 Kotlin 的编程竞赛参加者, 以及以前未参加过编程竞赛的 Kotlin 开发者. 对这两种情况, 我们假设读者已经具备相应的编程技能.

编程竞赛 (https://en.wikipedia.org/wiki/Competitive_programming) 是一种智力竞赛, 参赛者编写程序来解决指定的算法问题, 并要满足严格的限定条件. 这里的程序可能很简单, 任何开发者都能够解决, 只需要很少的代码就能得到答案, 也可能很复杂, 需要知道特定的算法, 数据结构, 以及大量的实践经验. 尽管 Kotlin 并不是针对编程竞赛特别设计的, 但它恰好适合这一领域, 能够大量减少程序员需要编写和阅读的样板代码(Boilerplate Code), 因此程序员既能够象使用动态类型(dynamically-typed)脚本语言那样高效率的读写代码, 同时又拥有静态类型(statically-typed)语言提供的工具支持和性能优势.

关于如何设置 Kotlin 开发环境, 请参见 Kotlin/JVM 入门 (Kotlin/JVM 入门). 在编程竞赛中, 通常会创建单个项目, 然后每个问题的解答会在单个源代码文件中编写.

简单的示例: 可达数(Reachable Number)问题

下面我们来看一个具体的例子.

Codeforces (https://codeforces.com/) 第 555 轮已于 4月26日 举办了第 3 组, 因此它有很多问题可供任何开发者尝试. 你可以通过 这个链接 (https://codeforces.com/contest/1157) 来阅读这些问题. 其中最简单的问题是 问题 A: 可达数(Reachable Number) (https://codeforces.com/contest/1157/problem/A). 它要求实现题干部分描述的一个简单算法.

我们来解决这个问题, 首先创建一个任意名称的 Kotlin 源代码文件. `A.kt` 也可以. 首先, 你需要实现题干部分指定的一个函数, 如下:

我们的函数 f(x) 如下: 我们对 x 加 1, 然后, 如果结果数字的末尾存在至少 1 个 0, 我们删除这个 0.

Kotlin 是一种注重实践、无固定成见的语言, 既支持命令式(imperative), 也支持函数式(function programming)编程风格, 并不强迫开发者使用哪一种. 你可以用函数式编程风格来实现函数 `f`, 使用 Kotlin 的 尾递归(tail recursion) ("尾递归函数(Tail recursive function)" in "函数") 功能:

```
tailrec fun removeZeroes(x: Int): Int =
    if (x % 10 == 0) removeZeroes(x / 10) else x
```

```
fun f(x: Int) = removeZeroes(x + 1)
```

或者, 你也可以为函数 f 编写命令式(imperative)的实现, 使用传统的 while 循环 (条件与循环), 和可变的变量, 在 Kotlin 中表达为 var ("变量" in "基本语法"):

```
fun f(x: Int): Int {
    var cur = x + 1
    while (cur % 10 == 0) cur /= 10
    return cur
}
```

在 Kotlin 中, 由于普遍使用类型推断(type-inference), 很多地方的类型声明是可选的, 但在编译时刻, 每一个声明仍然具有一个确定的静态类型.

现在, 只需要编写 main 函数, 读取输入, 以及实现题目要求的算法的其他部分 — 通过标准输入给定的初始整数 n, 对它反复执行函数 f, 计算产生的不同的整数的个数.

默认, Kotlin 运行在 JVM 平台, 能够直接访问大量而且高效的库, 包括通用的集合与数据结构, 比如动态大小的数组 (ArrayList), 基于 hash 的 map 和 set (HashMap/HashSet), 基于树(tree) 的有序 map 和 set (TreeMap/TreeSet). 我们使用整数的 hash set 来追踪执行函数 f 时已经到达过的数值, 这个问题的简单的命令式编程风格的版本可以编写如下:

Kotlin 1.6.0 及以后版本

```
fun main() {
    var n = readln().toInt() // 从标准输入读取整数
    val reached = HashSet<Int>() // 可变的 hash set
    while (reached.add(n)) n = f(n) // 反复执行函数 f
    println(reached.size) // 将答案打印到标准输出
}
```

在编程竞赛中不需要处理输入错误的情况. 编程竞赛中的输入格式永远是正确的, 实际的输入不会与题目中描述的不同. 因此你可以使用 Kotlin 的 readln() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/readln.html) 函数. 它假定输入字符串存在, 否则会抛出异常. 类似的, 如果输入字符串不是整数, String.toInt() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/to-int.html) 函数会抛出异常.

旧版本

```
fun main() {
    var n = readLine()!!.toInt() // 从标准输入读取整数
    val reached = HashSet<Int>() // 可变的 hash set
    while (reached.add(n)) n = f(n) // 反复执行函数 f
    println(reached.size) // 将答案打印到标准输出
}
```

注意, 在 readLine() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/read-line.html) 函数调用之后使用了 Kotlin 的 null 判断操作符 ("!! 操作符" in "Null 值安全性") `!!`. Kotlin 的 `readLine()` 函数定义是返回一个 可为 null 的类型 ("可为 null 的类型与不可为 null 的类型" in "Null 值安全性") `String?`, 并在输入结束时返回 `null`, 因此要求开发者处理没有输入的情况. 在编程竞赛中, 没有必要处理输入格式不正确的情况. 输入格式总是会明确指定, 而且实际输入不会违反题目描述中指定的输入格式. 这就是 null 判断操作符 `!!` 的含义 — 它假定输入字符串总是存在, 否则抛出一个异常. String.toInt() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/to-int.html) 也与此类似.

所有的在线编程竞赛都允许使用预先编写的代码, 因此你可以定义自己的工具函数库, 让你的解题代码更加简短, 容易阅读, 也容易编写. 然后可以使用这些代码作为解题答案的模板. 比如, 在编程竞赛中可以定义下面的辅助函数, 来读取输入:

Kotlin 1.6.0 及以后版本

```
private fun readStr() = readln() // 读取单个字符串行
private fun readInt() = readStr().toInt() // 读取单个整数
// 对你的解答中需要用到的其他类型, 编写类似函数
```

旧版本

```
private fun readStr() = readLine()!! // 读取单个字符串行
private fun readInt() = readStr().toInt() // 读取单个整数
// 对你的解答中需要用到的其他类型, 编写类似函数
```

注意这里使用了 `private` 可见度修饰符 (可见度修饰符). 虽然可见度修饰符的概念与编程竞赛无关, 但通过使用它, 你可以从相同的代码模板创建多个解答文件, 而不会由于相同的包内存在多个同名的

public 声明而出现编译错误.

函数式操作符示例: 长数(Long Number)问题

对于更复杂的问题, Kotlin 对集合的函数式操作的扩展库可以很便利的减少样板代码, 让代码变成自顶向下的线性结构, 以及从左向右的数据变换管道. 比如, 问题 B: 长数(Long Number) (https://codeforces.com/contest/1157/problem/B) 要求实现一个贪婪算法, 这个算法可以通过这种风格来实现, 完全不需要使用可变的变量:

Kotlin 1.6.0 及以后版本

```
fun main() {
    // 读取输入
    val n = readln().toInt()
    val s = readln()
    val fl = readln().split(" ").map { it.toInt() }
    // 定语局部函数 f
    fun f(c: Char) = '0' + fl[c - '1']
    // 贪婪查找第一个和最后一个下标
    val i = s.indexOfFirst { c -> f(c) > c }
        .takeIf { it >= 0 } ?: s.length
    val j = s.withIndex().indexOfFirst { (j, c) -> j > i && f(c)
< c }
        .takeIf { it >= 0 } ?: s.length
    // 组合答案, 并输出
    val ans =
        s.substring(0, i) +
        s.substring(i, j).map { c -> f(c) }.joinToString("") +
        s.substring(j)
    println(ans)
}
```

旧版本

```
fun main() {
    // 读取输入
    val n = readLine()!!.toInt()
```

```
    val s = readLine()!!
    val fl = readLine()!!.split(" ").map { it.toInt() }
    // 定语局部函数 f
    fun f(c: Char) = '0' + fl[c - '1']
    // 贪婪查找第一个和最后一个下标
    val i = s.indexOfFirst { c -> f(c) > c }
        .takeIf { it >= 0 } ?: s.length
    val j = s.withIndex().indexOfFirst { (j, c) -> j > i && f(c)
< c }
        .takeIf { it >= 0 } ?: s.length
    // 组合答案, 并输出
    val ans =
        s.substring(0, i) +
        s.substring(i, j).map { c -> f(c) }.joinToString("") +
        s.substring(j)
    println(ans)
}
```

在这段密集的代码中, 除了集合的变换之外, 你还看到很多 Kotlin 的便利功能, 比如局部函数, 以及 elvis 操作符 ("Elvis 操作符" in "Null 值安全性") `?:`, 它可以将 惯用法 (惯用法) "如果值为正, 则使用这个值, 否则使用字符串长度", 写成简洁可读的表达式 `.takeIf { it >= 0 } ?: s.length`, 但是 Kotlin 也完全可以定义额外的值可变的变量, 以命令式的风格来表达同样的代码.

在编程竞赛中, 为了让这种读取输入的任务更加简洁, 你可以定义下面这些辅助性的输入读取函数:

Kotlin 1.6.0 及以后版本

```
private fun readStr() = readln() // 读取单个字符串行
private fun readInt() = readStr().toInt() // 读取单个整数
private fun readStrings() = readStr().split(" ") // 读取多个字符串
private fun readInts() = readStrings().map { it.toInt() } // 读取
多个整数
```

旧版本

```
private fun readStr() = readLine()!! // 读取单个字符串行
private fun readInt() = readStr().toInt() // 读取单个整数
```

```
private fun readStrings() = readStr().split(" ") // 读取多个字符串
private fun readInts() = readStrings().map { it.toInt() } // 读取多个整数
```

通过这些辅助函数, 读取输入的那部分代码可以变得更简单, 可以与题目描述的输入规格逐行对应:

```
// 读取输入
val n = readInt()
val s = readStr()
val fl = readInts()
```

注意, 在编程竞赛中为变量取的名字, 经常会比实际工作中要短, 因为代码只编写一次, 之后就不再维护了. 但是, 这些变量名仍然遵守一些规则, 以便于记忆 — `a` 表示数组, `i`, `j`, 等等表示下标, `r`, 和 `c` 表述表中的行和列数, `x` 和 `y` 表示座标, 等等. 对输入数据使用与题目描述中相同的名称会比较简单. 但是, 更加复杂的问题要求更多的代码, 因此需要变量和函数的名称更长, 而且含义清晰.

更多提示和技巧

编程竞赛题目的输入通常类似如下:

输入的第一行包含两个整数 `n` 和 `k`

在 Kotlin 中, 可以对整数 List 使用 解构声明(Destructuring Declaration) (解构声明) 功能, 简洁的解析这样的输入行:

```
val (n, k) = readInts()
```

也可能会使用 JVM 的 `java.util.Scanner` 类来解析缺少结构的输入格式. Kotlin 被设计为能够与 JVM 的库良好交互, 因此在 Kotlin 中使用这些库会感到非常自然. 但是, 请注意 `java.util.Scanner` 非常的慢. 实际上, 它太慢了, 以至于解析 10^5 以上个整数时, 可能会超过题目通常要求的 2 秒运行时间限制, 这样的输入, 使用简单的 Kotlin 函数 `split(" ").map { it.toInt() }` 就可以解决.

在 Kotlin 中输出通常使用简单的 println(...) (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/println.html) 调用, 并使用 Kotlin 的 字符串模板 ("字符串模板" in "字符串"). 但是, 如果输出包含 10^5 行以上时, 一定要注意. 调用这样多次的 `println` 会非常的慢, 因为在 Kotlin 中标准输出会在每一行之后自动刷出. 要从数组或列表输出大量行内容, 更快的方式是使用 joinToString() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/join-to-string.html) 函数, 用 `"\n"` 作为分隔符, 比如:

```
println(a.joinToString("\n")) // 数组/列表的每个元素成为单独的行
```

学习 Kotlin

Kotlin 很容易学习, 尤其是对于那些已经熟悉 Java 的程序员. 针对软件开发者的 Kotlin 基本语法简短介绍, 请参见在本站参考文档: 基本语法 (基本语法).

IDEA 已经内置了 Java 到 Kotlin 转换器 (https://www.jetbrains.com/help/idea/converting-a-java-file-to-kotlin-file.html). 可供熟悉 Java 的人用来学习对应的 Kotlin 语法结构, 但它并不完美, 还需要你自己来熟悉 Kotlin, 并学习 Kotlin 惯用法 (惯用法).

要学习 Kotlin 语法和 Kotlin 标准库 API, 一个很好的资源是 Kotlin Koan (Kotlin Koan).

Kotlin 1.9.20 版中的新功能

最终更新: 2024/09/10

发布日期: 2023/11/01 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.9.20 已经发布了, K2 编译器对于所有编译目标已进入 Beta 版, Kotlin Multiplatform 已进入稳定版. 此外, 还有以下一些重要功能:

- 跨平台项目时默认使用新的层级结构模板
- Kotlin Multiplatform 中对 Gradle 配置缓存的完全支持
- 在 Kotlin/Native 中默认启用自定义内存分配器
- 在 Kotlin/Native 中垃圾收集器的性能改进
- Kotlin/Wasm 中的新的构建目标以及改名的构建目标
- 在 Kotlin/Wasm 标准库中支持 WASI API

关于本次更新的概要介绍, 你可以观看以下视频:

IDE 支持

在以下 IDE 中可以使用支持 1.9.20 版的 Kotlin plugin:

IDE	支持的版本
IntelliJ IDEA	2023.1.x, 2023.2.x, 2023.x
Android Studio	Hedgehog (2023.1.1), Iguana (2023.2.1)

> 从 IntelliJ IDEA 2023.3.x 和 Android Studio Iguana (2023.2.1) Canary 15 开始, 会自动包含并更新 Kotlin plugin. 你只需要在你的项目中更新 Kotlin 版本.

新 Kotlin K2 编译器的更新

JetBrains 的 Kotlin 开发组一直在努力稳定新的 K2 编译器, 这个编译器将会带来显著的性能改进, 加快新的语言功能的开发, 统一 Kotlin 支持的所有平台, 并为跨平台项目提供更好的架构.

K2 目前对所有的编译目标都处于 **Beta 版**. 详情请参见 release blog (https://blog.jetbrains.com/kotlin/2023/11/kotlin-1-9-20-released/)

对 Kotlin/Wasm 的支持

从这个发布版开始, Kotlin/Wasm 支持新的 K2 编译器. 参见 如何在你的项目中启用它.

针对 K2 的 kapt 编译器 plugin 预览版

在 kapt 编译器 plugin 中对 K2 的支持是 实验性功能 (Kotlin 各部分组件的稳定性). 需要使用者同意(Opt-in) (详情见下文), 请注意, 只为评估和试验目的来使用这个功能.

在 1.9.20 中, 你可以试用针对 K2 编译器的 kapt 编译器 plugin (kapt 编译器插件). 要在你的项目中使用 K2 编译器, 请向你的 `gradle.properties` 文件添加以下选项:

```
kotlin.experimental.tryK2=true
kapt.use.k2=true
```

或者, 你可以通过以下步骤启用针对 K2 的 kapt:

1. 在你的 `build.gradle.kts` 文件中, 设置语言版本 ("languageVersion 设置示例" in "Kotlin Gradle plugin 中的编译器选项") 为 `2.0`.
2. 在你的 `gradle.properties` 文件中, 添加 `kapt.use.k2=true`.

如果你在使用针对 K2 编译器的 kapt 时遇到任何问题, 请到我们的 问题追踪系统 (http://kotl.in/issue) 提交报告.

如何启用 Kotlin K2 编译器

在 Gradle 中启用 K2

要启用并检验 Kotlin K2 编译器, 请通过下面的编译器选项使用新的语言版本:

```
-language-version 2.0
```

你可以在你的 `build.gradle.kts` 文件中指定这个选项:

```kotlin
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = "2.0"
        }
    }
}
```

在 Maven 中启用 K2

要启用并检验 Kotlin K2 编译器, 请更新的你 `pom.xml` 文件的 `<project/>` 小节:

```xml
<properties>
    <kotlin.compiler.languageVersion>2.0</kotlin.compiler.languageVersion>
</properties>
```

在 IntelliJ IDEA 中启用 K2

要在 IntelliJ IDEA 中启用并检验 Kotlin K2 编译器, 请选择菜单 **Settings** | **Build, Execution, Deployment** | **Compiler** | **Kotlin Compiler**, 将 **Language Version** 选项更新为 `2.0 (experimental)`.

留下你对于新 K2 编译器的反馈意见

如果你能提供你的反馈意见, 我们将会非常感谢!

- 在 Kotlin Slack 频道中, 直接向 K2 开发者提供你的反馈意见 – 获得邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up?_gl=1*ju6cbn*_ga*MTA3MTk5NDkzMC4xNjQ2MDY3MDU4*_ga_9J976DJZ68*MTY1ODMzNzA3OS4xMDAuMS4xNjU4MzQwODEwLjYw), 并加入 #k2-early-adopters (https://kotlinlang.slack.com/archives/C03PK0PE257) 频道.
- 在 我们的问题追踪系统 (https://kotl.in/issue) 中, 报告你遇到的新 K2 编译器的问题.
- 启用 Send usage statistics 选项 (https://www.jetbrains.com/help/idea/settings-usage-statistics.html), 允许 JetBrains 收集关于 K2 使用状况的匿名数据.

Kotlin/JVM

从 1.9.20 版开始, 编译器能够生成包含 Java 21 字节码的类.

Kotlin/Native

Kotlin 1.9.20 包含稳定的内存管理器, 其中包括, 默认使用新的内存分配器, 对垃圾收集器的性能改进, 以及其他更新:

- 默认启用自定义内存分配器
- 垃圾收集器的性能改进
- `klib` artifact 的增量编译
- 解决库链接的问题
- 类构造器调用时的伴随对象初始化
- 对所有的 cinterop 声明要求使用者同意(Opt-in)
- 链接器错误的自定义信息
- 删除了旧的内存管理器
- 我们的编译目标层级策略的变更

默认启用自定义内存分配器

Kotlin 1.9.20 默认启用新的内存分配器. 它的设计目标是取代以前的默认分配器, `mimaloc`, 使得垃圾收集更加高效, 并提高 Kotlin/Native 内存管理器 (Kotlin/Native 内存管理) 的运行期性能.

新的自定义分配器将系统内存分为多个页面(Page), 允许按连续的顺序进行独立的清理. 每次分配的内存都会成为一个页面(Page)内的内存块(Memory Block), 并且页面会追踪各个块的大小. 各种不同的页面类型进行了不同的优化, 以适应于不同的内存分配大小. 内存块的连续排列保证了可以对所有的分配块进行高效的迭代.

当一个线程分配内存时, 它会根据分配的大小搜索适当的页面. 线程会根据不同的大小类别维护一组页面. 对于一个确定的大小, 当前页通常可以容纳这个内存分配. 如果不能, 那么线程会从共享的分配空间请求一个不同的页面. 这个页面的状态可能是可用, 需要清理, 或需要创建.

新的内存分配器允许同时使用多个多个独立的分配空间, 因此 Kotlin 开发组可以实验不同的页面布局, 进一步提高性能.

如何启用自定义内存分配器

从 Kotlin 1.9.20 开始, 新的内存分配器默认启用. 不需要额外的设置.

如果你遇到内存消耗过高的情况, 你可以在你的 Gradle 构建脚本中使用 `-Xallocator=mimalloc` 或 `-Xallocator=std` 选项, 切换回原来的 `mimaloc`, 或系统分配. 请将这样的问题报告到 YouTrack (https://kotl.in/issue), 帮助我们改进新的内存分配器.

关于新的分配器设计的技术细节, 请参见 README (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/runtime/src/alloc/custom/README.md).

垃圾收集器的性能改进

Kotlin 开发组一直在改进新的 Kotlin/Native 内存管理器的性能和稳定性. 这个发布版带来了对垃圾收集器 (GC)的很多重大变更, 包括以下重要功能:

- 使用完全并行标记(Full Parallel Mark), 减少 GC 的暂停时间
- 追踪大块内存, 提高分配性能

使用完全并行标记(Full Parallel Mark), 减少 GC 的暂停时间

以前, 默认的垃圾收集器执行的只是部分的并行标记. 当转换器线程(Mutator Thread) 被暂停时, 它会从它自己的根开始标记 GC, 例如线程局部变量(thread-local variable)和调用栈. 同时, 一个单独的 GC 线程负责从全局根(global root) 开始标记, 以及所有那些正在运行原生代码因此没有暂停的转换器的根.

如果只存在有限数量的全局对象, 而且转换器线程(Mutator Thread)花费很多时间在运行状态下执行 Kotlin 代码, 那么这种方案曾经工作得很好. 但是, 对于典型的 iOS 应用程序, 就不是这样的情况了.

现在 GC 使用完全并行标记(Full Parallel Mark), 它结合了暂停的转换器(Paused Mutator), GC 线程, 以及可选的标记线程(Marker Thread), 来处理标记队列(Mark Queue). 默认情况下, 标记过程由以下二者执行:

- 暂停的转换器(Paused Mutator). 不是处理它们自己的根, 然后进入空闲状态, 不执行代码, 相反, 它们会参与整个标记过程.
- GC 线程. 这样可以确保至少存在一个线程会执行标记过程.

这个新方案让标记过程更加高效, 减少了 GC 的暂停时间.

追踪大块内存, 提高分配性能

以前, GC 调度器分别追踪每个对象的内存分配. 但是, 新的默认的自定义分配器和 `mimalloc` 内存分配器都不会为每个对象分配单独的存储空间; 它们会一次性为多个对象分配大块的区域.

在 Kotlin 1.9.20 中, GC 追踪内存区域而不是追踪单独的对象. 这样会减少每次分配时执行的任务数量, 因此可以提高小对象的内存分配速度, 也有助于尽量减少垃圾收集器的内存使用量.

klib artifact 的增量编译

> ⚠ 这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://kotl.in/issue) 提供你的反馈意见.

Kotlin 1.9.20 引入了对 Kotlin/Native 的新的编译时间优化. 从 `klib` artifact 到原生代码的编译, 现在是部分增量的编译.

在 debug 模式下将 Kotlin 源代码编译为原生二进制代码时, 编译经过两个阶段:

1. 源代码编译为 `klib` artifact.
2. `klib` artifact, 以及依赖项, 编译为二进制代码.

为了优化第二阶段的编译时间, 开发组实现了对依赖项的编译器缓存. 依赖项到原生代码只会编译一次, 编译结果会在每次编译二进制代码时重新使用. 但是, 每次项目发生变更时, 从项目源代码到 `klib` artifact 的构建, 总是会完全重编译为原生代码.

通过新的增量编译, 如果项目模块的变更导致只有一部分源代码重编译为 `klib` artifact, 那么也只有一部分的 `klib` 会被重新编译为二进制代码.

要启用增量编译, 请向你的 `gradle.properties` 文件添加以下选项:

```
kotlin.incremental.native=true
```

如果你遇到问题, 请报告到 YouTrack (https://kotl.in/issue).

解决库链接的问题

这个发布版改进了 Kotlin/Native 编译器链接 Kotlin 库时发生问题的处理方式. 错误消息现在包含更加易于阅读的声明, 因为它们使用签名名称而不是 hash 值, 可以帮助你更加容易的查找并修复错误. 下面是一个例子:

```
No function found for symbol
'org.samples/MyClass.removedFunction|removedFunction(kotlin.Int;kotl
in.String){}[0]'
```

Kotlin/Native 编译器发现与第 3 方 Kotlin 库链接的错误, 并在运行期报告错误. 如果一个第 3 方 Kotlin 库的作者对实验性 API进行了不兼容的变更, 而这个 API 又被另一个第 3 方 Kotlin 库使用, 你就会遇到这样的问题.

从 Kotlin 1.9.20 开始, 编译器默认以静默模式检测链接错误. 你可以在你的项目中修改设定:

- 如果你希望在你的编译日志中记录这些错误, 请使用 `-Xpartial-linkage-loglevel=WARNING` 编译器选项, 启用警告.
- 也可以使用 `-Xpartial-linkage-loglevel=ERROR`, 将编译器报告的警告提升为编译错误. 这种情况下, 编译会失败, 你会在编译日志中得到所有的错误. 使用这个选项, 可以更加严格的检查链接错误.

```
// 在 Gradle 构建文件中传递编译器参数的示例:
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                // 将链接错误报告为警告:
                freeCompilerArgs.add("-Xpartial-linkage-
loglevel=WARNING")

                // 将链接错误的警告提升为错误:
                freeCompilerArgs.add("-Xpartial-linkage-
loglevel=ERROR")
            }
        }
    }
}
```

如果你在使用这个功能时遇到意外的问题, 你随时可以使用 `-Xpartial-linkage=disable` 编译器选项关闭这个功能. 请记得将这样的问题报告到 我们的问题追踪系统 (https://kotl.in/issue).

类构造器调用时的伴随对象初始化

从 Kotlin 1.9.20 开始, Kotlin/Native 后端会在类的构造器中调用伴随对象的静态初始化器:

```
class Greeting {
    companion object {
        init {
            print("Hello, Kotlin!")
        }
    }
}

fun main() {
    val start = Greeting() // 输出结果为 "Hello, Kotlin!"
}
```

这个行为现在与 Kotlin/JVM 平台统一了, 在 Kotlin/JVM 平台上, 在与 Java 静态初始化器语义匹配的对应的类被装载时(被解析时), 伴随对象就会被初始化.

现在这个功能的实现在各个平台之间更加一致了, 因此在 Kotlin Multiplatform 项目中更加容易共用代码.

对所有的 cinterop 声明要求使用者同意(Opt-in)

从 Kotlin 1.9.20 开始, 由 `cinterop` 工具从 C 和 Objective-C 库(例如 libcurl 和 libxml)生成的所有 Kotlin 声明, 都被标记为 `@ExperimentalForeignApi`. 如果缺少使用者同意(Opt-in)注解, 你的代码将无法编译.

这个要求反应了 C 和 Objective-C 库导入功能的 实验性 ("稳定性级别" in "Kotlin 各部分组件的稳定性") 状态. 我们建议你将这个功能限制在你的项目的特定区域中. 这样可以在我们稳定库导入功能之后, 让你的迁移更加容易.

> 对于随 Kotlin/Native 一起发布的原生平台库 (例如 Foundation, UIKit, 以及 POSIX), 它们的 API 只有一部分需要使用 `@ExperimentalForeignApi` 注解标注使用者同意. 对这样的情况, 你会得到警告信息, 要求你标注使用者同意.

链接器错误的自定义信息

如果你是库的作者, 现在你可以通过自定义消息来帮助你的用户解决链接器错误.

如果你的 Kotlin 库依赖于 C 或 Objective-C 库, 例如, 使用了 CocoaPods 集成 (CocoaPods 概述与设置), 那么你的库的使用者需要在本地机器上存在这些依赖库, 或者在项目的构建脚本中明确配置这些库. 否则, 使用者会遇到一个令人迷惑的 "Framework not found" 消息.

现在你可以在编译失败的信息中提供具体的指示或链接. 方法是, 向 `cinterop` 传递 `-Xuser-setup-hint` 编译器选项 或者在你的 `.def` 文件中添加 `userSetupHint=message` 属性.

删除了旧的内存管理器

在 Kotlin 1.6.20 中引入了 新的内存管理器 (Kotlin/Native 内存管理), 而且在 1.7.20 中默认启用. 之后, 我们对它进行了很多更新, 并改进了性能, 现在它已经称为稳定版.

现在已经到了完成废弃周期, 并删除旧的内存管理器的时刻. 如果你还在使用它, 请从你的 `gradle.properties` 文件删除 `kotlin.native.binary.memoryModel=strict` 选项, 并遵照我们的 迁移指南 (迁移到新的内存管理器), 进行必要的变更.

我们的编译目标层级策略的变更

我们决定升级对 第 1 层支持 ("第 1 层" in "Kotlin/Native 支持的目标平台") 的要求. Kotlin 开发组致力于对符合第 1 层条件的编译目标, 提供编译器发布版本之间的源代码和二进制兼容性. 这些编译目标还必须使用 CI 工具进行常规测试, 以确保能够编译和运行. 目前, 对于 macOS 主机, 第 1 层包括以下编译目标:

- `macosX64`
- `macosArm64`
- `iosSimulatorArm64`
- `iosX64`

在 Kotlin 1.9.20 中, 我们还删除了很多以前废弃的编译目标, 即:

- `iosArm32`
- `watchosX86`
- `wasm32`
- `mingwX86`
- `linuxMips32`
- `linuxMipsel32`

关于编译目标的完整列表, 请参见目前 支持的编译目标 (Kotlin/Native 支持的目标平台).

Kotlin Multiplatform

Kotlin 1.9.20 集中于 Kotlin Multiplatform 的稳定性, 并提供了新的项目向导和其它重要功能, 进一步改进开发者体验:

- Kotlin Multiplatform 已进入稳定版
- 用于配置跨平台项目的模板
- 新的项目向导
- 对 Gradle 配置缓存的完全支持
- 新的标准库版本在 Gradle 更容易配置
- 对第 3 方 cinterop 库的默认支持
- 在 Compose Multiplatform 项目中对 Kotlin/Native 编译缓存的支持
- 兼容性指南

Kotlin Multiplatform 已进入稳定版

1.9.20 版的发布标志了 Kotlin 演化历程中的一个重要的里程碑: Kotlin Multiplatform (Kotlin Multiplatform) 终于进入了稳定版. 这表示这个技术已经可以安全的用于你的项目, 并且 100% 可以用于真实生产环境. 还意味着 Kotlin Multiplatform 未来的开发会继续符合我们严格的 向后兼容性规则 (https://kotlinfoundation.org/language-committee-guidelines/).

请注意, Kotlin Multiplatform 的一些高级功能还在继续演化. 在使用这些功能时, 你会收到警告信息, 代表你使用的功能目前的稳定性状态. 在 IntelliJ IDEA 中使用任何实验性功能之前, 你需要通过菜单 **Settings** | **Advanced Settings** | **Kotlin** | **Experimental Multiplatform**, 明确的启用它.

- 查看 Kotlin blog (https://blog.jetbrains.com/kotlin/2023/11/kotlin-multiplatform-stable/), 阅读关于 Kotlin Multiplatform 稳定性和未来开发计划的更多信息.
- 查看 Multiplatform 兼容性指南 (Kotlin Multiplatform 兼容性指南), 看看在向稳定版演化时有哪些重要变更.
- 阅读 预期(Expected)声明与实际(Actual)声明机制 (预期声明与实际声明), 这是 Kotlin Multiplatform 的重要部分,在本次发布版中, 它也部分的稳定了.

用于配置跨平台项目的模板

从 Kotlin 1.9.20 开始, Kotlin Gradle plugin 会为常见的跨平台场景自动创建共享的源代码集. 如果你的项目设置属于这样的情况, 你就不需要手动配置源代码集层级结构. 只需要为你的项目明确指定必要的编译目标.

由于有了 Kotlin Gradle plugin 的新功能: 默认的层级结构模板, 设置变得更加容易了. 它是 plugin 内置的预定义的源代码集层级结构模板. 其中包含 Kotlin 为你声明的编译目标自动创建的中间源代码集. 参见完整的模板.

更容易的创建你的项目

假设有一个跨平台项目, 编译目标包括 Android 和 iPhone 设备, 开发环境为 Apple silicon MacBook. 我们来比较一下不同版本的 Kotlin 中的项目设置:

<table>
<tr><th>Kotlin 1.9.0 和以前的版本 (标准设置)</th><th>Kotlin 1.9.20</th></tr>
<tr><td>

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()

    sourceSets {
        val commonMain by getting

        val iosMain by creating {
            dependsOn(commonMain)
        }

        val iosArm64Main by getting
{
            dependsOn(iosMain)
        }

        val iosSimulatorArm64Main by
getting {
            dependsOn(iosMain)
        }
    }
}
```

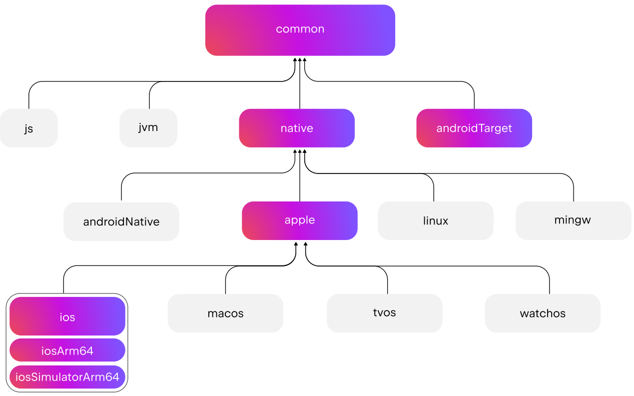
</td><td>

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()

    // iosMain 源代码集会
自动创建
}
```

</td></tr>
</table>

请注意, 使用默认的层级结构模板显著减少了设置你的项目时需要的样板代码的数量.

当你在你的代码中声明 `androidTarget`, `iosArm64`, 和 `iosSimulatorArm64` 编译目标时, Kotlin Gradle plugin 会从模板中找到合适的共享源代码集, 并为你创建这些源代码集. 最后产生的层级结构类似下图:



绿色的源代码集会自动创建并包含到项目中, 同时, 默认模板中的灰色的源代码集会被忽略.

对源代码集使用自动补完功能

为了更容易的使用创建的项目结构, IntelliJ IDEA 现在对使用默认层级结构模板创建的源代码集提供了自动补完功能:

```
plugins {
    kotlin("multiplatform")
}

group = "org.jetbrains.kotlin"
version = "1.0"

repositories {
    mavenCentral()
    mavenLocal()
    google()
}

kotlin {
    androidTarget()
    iosX64()
    iosSimulatorArm64()

    
}

```

IDE 对源代码集名称的自动补完

如果你没有声明某个编译目标, 因而不存在对应的源代码集, 那么在你试图访问这个不存在的源代码集时, Kotlin 会提示警告. 在下面的示例中, 不存在 JVM 编译目标 (只有 `androidTarget`, 这是不同的编译目标). 但我们来试试使用 `jvmMain` 源代码集, 看看会发生什么:

```
kotlin {
    androidTarget()
```

```
    iosArm64()
    iosSimulatorArm64()

    sourceSets {
        jvmMain {
        }
    }
}
```

在这种情况下, Kotlin 会在构建日志中报告警告:

```
w: Accessed 'source set jvmMain' without registering the jvm target:
  kotlin {
      jvm() /* <- register the 'jvm' target */

      sourceSets.jvmMain.dependencies {

      }
  }
```

设置编译目标层级结构

从 Kotlin 1.9.20 开始, 会自动启用默认的层级结构模板. 大多数情况下, 不需要更多设置.

但是, 如果你在迁移一个在 1.9.20 版之前创建的既有项目, 如果你曾经使用 `dependsOn()` 调用, 手动的引入了中间源代码(Intermediate Source), 你可能遇到警告. 为了解决这个问题, 请执行以下步骤:

- 如果现在默认的层级结构模板已经包含了你的中间源代码集, 请删除所有的手动 `dependsOn()` 调用, 以及使用 `by creating` 构造创建的源代码集.

  关于全部的默认源代码集, 请参见 完整的层级结构模板.

- 如果你想要默认的层级结构模板没有提供的其他源代码集, 例如, 在 macOS 和 JVM 编译目标之间共用代码的源代码集, 请调整层级结构, 方法是使用 `applyDefaultHierarchyTemplate()` 来明确的重新适用模板, 然后和通常的做法一样, 使用 `dependsOn()` 来手动配置其他源代码集:

  ```
  kotlin {
      jvm()
      macosArm64()
      iosArm64()
  ```

```
    iosSimulatorArm64()

    // 明确的适用默认的层级结构. 它会创建一系列源代码集, 例如, iosMain 源代
码集:
    applyDefaultHierarchyTemplate()

    sourceSets {
        // 额外创建一个 jvmAndMacos 源代码集
        val jvmAndMacos by creating {
            dependsOn(commonMain.get())
        }

        macosArm64Main.get().dependsOn(jvmAndMacos)
        jvmMain.get().dependsOn(jvmAndMacos)
    }
}
```

- 如果在你的项目中已经有了源代码集, 名称与模板生成的源代码集相同, 但在不同的编译目标之间共用, 目前没有办法可以修改模板的源代码集之间的默认的 `dependsOn` 关系.

  一种解决方法是, 为你的目标找到不同的源代码集, 要么使用默认的层级结构模板中的源代码集, 要么使用手动创建的源代码集. 另一种方法是完全禁用模板.

  要禁用模板, 请向你的 `gradle.properties` 文件添加 `kotlin.mpp.applyDefaultHierarchyTemplate=false`, 并手动配置其它所有的源代码集.

  我们目前正在开发一个 API, 用于创建你自己的层级结构模板, 以简化这类设置过程.

查看完整的层级结构模板

当你声明你的项目的编译目标时, plugin 会从模板中选取对应的共用源代码集, 并在你的项目中创建它们.

common

js
jvm
native
androidTarget

androidNativeX64
androidNativeX86
androidNativeArm64
androidNativeArm32
androidNative

linux
linuxX64
linuxArm64

mingw
mingwX64

apple

iosX64
iosArm64
iosSimulatorArm64
ios

macos
macosX64
macosArm64

tvosX64
tvosArm64
tvosSimulatorArm64
tvos

watchos
watchosX64
watchosArm32
watchosArm64
watchosSimulatorArm64
watchosDeviceArm64

默认的层级结构模板

这个示例只显示了项目的 production 部分, 省略了 `Main` 后缀 (例如, 使用 `common` 而不是 `commonMain`). 但是, 还有完全相同的一组 `*Test` 源代码集.

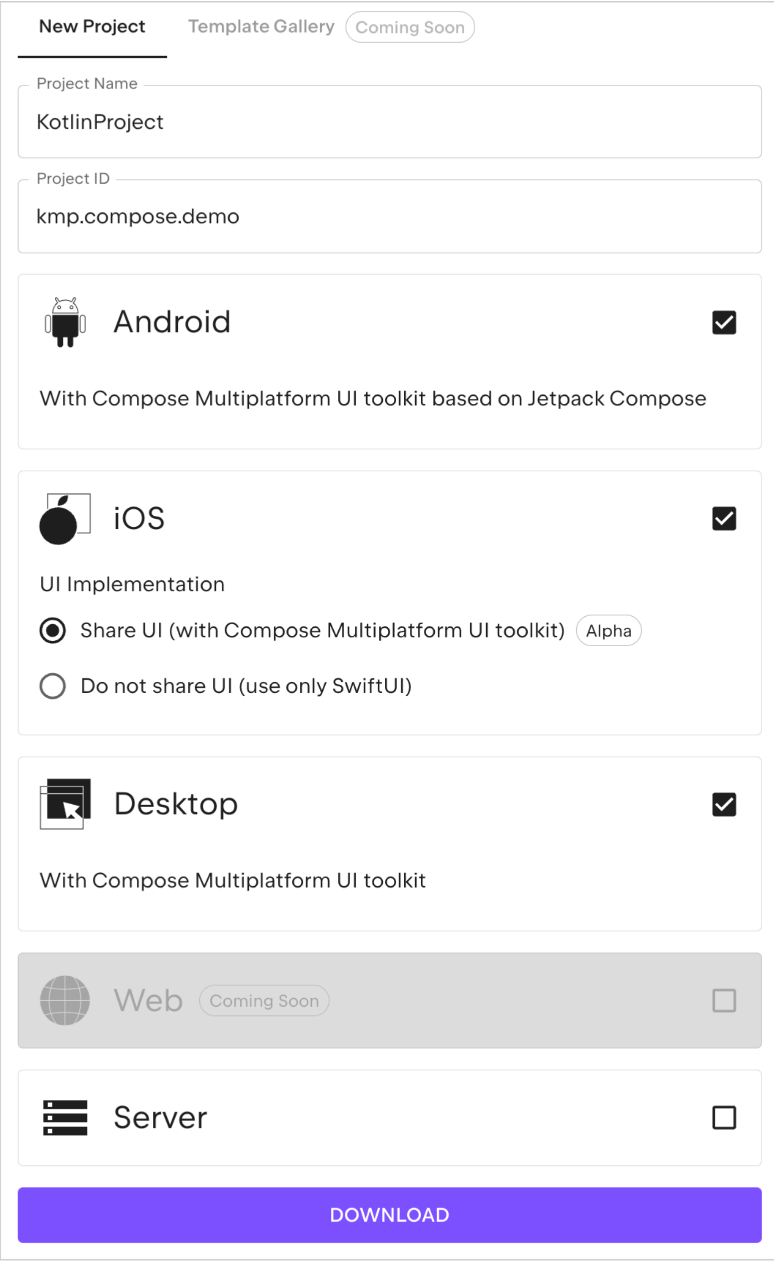
新的项目向导

JetBrains 开发组引入了一种新的方式来创建跨平台项目 – Kotlin Multiplatform web 向导 (https://kmp.jetbrains.com).

新的 Kotlin Multiplatform 向导的第一个实现包含了最常见的 Kotlin Multiplatform 使用场景. 它包含了对之前的项目模板的所有反馈意见, 使得架构尽可能的健壮和可靠.

新的向导使用分布式架构, 使得我们可以拥有统一的后端和不同的前端, 使用 web 版是其中的第一步. 我们正在考虑将来实现 IDE 版, 并创建命令行工具. 在 web 版中, 你永远能够使用向导的最新版本, 而在 IDE 中, 你则需要等待下一个版本的发布.

使用新的向导, 项目设置会变的比过去更加简单. 你可以针对移动环境, 服务器, 以及桌面开发来选择目标平台, 根据你的需求来调整你的项目. 我们还计划在未来的发布中增加 Web 开发目标平台.



Multiplatform Web 向导

新的项目向导现在是使用 Kotlin 创建跨平台项目的首选方式. 从 1.9.20 开始, Kotlin plugin 不再 在 IntelliJ IDEA 中提供 **Kotlin Multiplatform** 项目向导.

新的向导将会引导你轻松的完成初始设置, 让你的开始过程更加顺利. 如果你遇到任何问题, 请报告到 YouTrack (https://kotl.in/issue), 帮助我们改进向导的使用体验.

Create project →

创建项目

(https://kmp.jetbrains.com)

Kotlin Multiplatform 中对 Gradle 配置缓存的完全支持

在以前的版本中, 我们曾经引入了 Gradle 配置缓存功能的 预览版 ("Gradle 配置缓存功能的预览版" in "Kotlin 1.9.0 版中的新功能"), 可以用于 Kotlin 跨平台库. 从 1.9.20 开始, Kotlin Multiplatform plugin 更加前进了一步.

它现在对 Kotlin CocoaPods Gradle plugin (CocoaPods Gradle plugin DSL 参考文档) 支持 Gradle 配置缓存, 也对 Xcode 构建需要的集成任务, 例如 `embedAndSignAppleFrameworkForXcode`, 支持 Gradle 配置缓存.

现在所有的跨平台项目都可以由于构建时间的改进而获益. Gradle 配置缓存可以对后续的构建重用配置阶段的结果, 因而加快构建过程. 更多详情, 以及配置说明, 请参见 Gradle 文档 (https://docs.gradle.org/current/userguide/configuration_cache.html#config_cache:usage).

新的标准库版本在 Gradle 更容易配置

在你创建跨平台项目时, 会对每个源代码集自动添加对标准库(`stdlib`)的依赖项. 这是设置你的跨平台项目的最简单的方式.

在之前的版本中, 如果你想要手动配置对标准库的依赖项, 你需要对每个源代码集分别配置. 从 `kotlin-stdlib:1.9.20` 开始, 你只需要在 `commonMain` 根源代码集中, 配置这个依赖项 **一次**:

<table>
<tr><th>1.9.10 和之前版本的标准库</th><th>1.9.20 版本的标准库</th></tr>
<tr><td>

```kotlin
kotlin {
    sourceSets {
        // 设置 common 源代码集
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlin:kotlin-stdlib-common:1.9.10")
            }
        }

        // 设置 JVM 源代码集
        val jvmMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlin:kotlin-stdlib:1.9.10")
            }
        }

        // 设置 JS 源代码集
        val jsMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlin:kotlin-stdlib-js:1.9.10")
            }
        }
    }
}
```

</td><td>

```kotlin
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation("org.jetbrains.kotlin:kotlin-stdlib:1.9.20")
            }
        }
    }
}
```

</td></tr>
</table>

这个变更是通过在标准库的 Gradle metadata 中包含新信息而实现的. 这使得 Gradle 能够对其他源代码集自动解析正确的标准库 artifact.

对第 3 方 cinterop 库的默认支持

对使用了 Kotlin CocoaPods Gradle (CocoaPods 概述与设置) plugin 的项目, Kotlin 1.9.20 添加了对所有 cinterop 依赖项的默认支持 (而不是通过使用者同意(Opt-in)的支持).

因此, 你现在可以共用更多原生代码, 而不是限制于平台相关的依赖项. 例如, 你可以向 `iosMain` 共用源代码集添加 对 Pod 库的依赖项 (添加 Pod 库依赖项).

在以前的版本中, 这个功能只能用于随 Kotlin/Native 一起发布的 平台相关的库 (平台库) (例如 Foundation, UIKit, 和 POSIX). 现在, 所有的第 3 方 Pod 库默认都可以在共用源代码集中使用. 你不再需要指定单独的 Gradle 属性来支持它们.

在 Compose Multiplatform 项目中对 Kotlin/Native 编译缓存的支持

本次发布解决了与 Compose Multiplatform 编译器 plugin 的一个兼容性问题, 这个问题主要影响 iOS 的 Compose Multiplatform 项目.

过去, 为了绕过这个问题, 你需要使用 `kotlin.native.cacheKind=none` Gradle 属性禁用缓存. 但是, 这个变通方法会造成性能问题: 它使得编译速度变慢, 因为在 Kotlin/Native 编译器中缓存不再有效.

现在这个问题已经解决了, 你可以从你的 `gradle.properties` 文件中删除 `kotlin.native.cacheKind=none`, 并在你的 Compose Multiplatform 项目中享受编译速度的改进.

关于改进编译速度的更多技巧, 请参见 Kotlin/Native 文档 (改进 Kotlin/Native 编译速度).

兼容性指南

在配置你的项目时, 请检查 Kotlin Multiplatform Gradle plugin 与 Gradle, Xcode, 和 Android Gradle plugin (AGP) 版本之间的兼容性:

Kotlin Multiplatform Gradle plugin	Gradle	Android Gradle plugin	Xcode
1.9.20	7.5 及以后版本	7.4.2–8.2	15.0. 详情请参见下文

从这个发布版开始, Xcode 的推荐版本为 15.0. 完全支持随 Xcode 15.0 一起发布的库, 你可以在你的 Kotlin 代码的任何地方访问这些库.

但是, XCode 14.3 在大多数情况下仍然可以使用. 请记住, 如果你在你的本地机器上使用 14.3 版, 那么随 Xcode 15 一起发布的库将会可见, 但不能访问.

Kotlin/Wasm

在 1.9.20 中, Kotlin Wasm 的稳定性达到了 Alpha 级 (Kotlin 各部分组件的稳定性).

- 与 Wasm GC phase 4 和最新的 opcode 之间的兼容性
- 新的 `wasm-wasi` 编译目标, 以及 `wasm` 编译目标改名为 `wasm-js`
- 在标准库中支持 WASI API
- Kotlin/Wasm API 的改进

> Kotlin Wasm 现在处于 Alpha 版 (Kotlin 各部分组件的稳定性). 它随时可能发生变更. 请注意, 只为评估和试验目的来使用这个功能.
>
> 希望你能通过 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

与 Wasm GC phase 4 和最新的 opcode 之间的兼容性

Wasm GC 已经进入了最后阶段, 它要求更新 opcode – 在二进制表达中使用的常数值. Kotlin 1.9.20 支持最新的 opcode, 因此我们强烈推荐你将你的 Wasm 项目更新到最新版的 Kotlin. 我们还推荐使用支持 Wasm 环境的最新版本的浏览器:

- 对 Chrome 和基于 Chromium 的浏览器, 119 或更新版本.
- Firefox, 119 或更新版本. 注意, 在 Firefox 119 中, 你需要 手动启用 Wasm GC (问题分析).

新的 wasm-wasi 编译目标, 以及 wasm 编译目标改名为 wasm-js

在这个发布版中, 我们对 Kotlin/Wasm 引入了一个新的编译目标 – `wasm-wasi`. 我还将 `wasm` 编译目标改名为 `wasm-js`. 在 Gradle DSL 中, 这些编译目标可以分别通过 `wasmWasi {}` 和 `wasmJs {}` 访问.

要在你的项目中使用这些编译目标, 请更新 `build.gradle.kts` 文件:

```
kotlin {
    wasmWasi {
        // ...
    }
    wasmJs {
```

```
        // ...
    }
}
```

以前引入的 `wasm {}` 代码段已经被废弃, 请改为使用 `wasmJs {}`.

要迁移你的既有的 Kotlin/Wasm 项目, 请执行以下步骤:

- 在 `build.gradle.kts` 文件中, 将 `wasm {}` 代码段改名为 `wasmJs {}`.
- 在你的项目结构中, 将 `wasmMain` 目录改名为 `wasmJsMain`.

在标准库中支持 WASI API

在这个发布版中, 我们包含了对 WASI (https://github.com/WebAssembly/WASI) 的支持, 这是对 Wasm 平台的一个系统接口. 支持 WASI, 提供了一组标准化的 API 来访问系统资源, 使你能够更容易的在浏览器之外的环境中使用 Kotlin/Wasm, 例如, 在服务器端应用程序中. 此外, WASI 提供了基于能力的安全性(Capability-based Security) – 在访问外部资源时的另一个安全层.

要运行 Kotlin/Wasm 应用程序, 你需要一个支持 Wasm 垃圾收集 (GC) 的 VM, 例如, Node.js 或 Deno. Wasmtime, WasmEdge, 以及其它环境, 对 Wasm GC 的完整支持还在开发中.

要导入一个 WASI 函数, 请使用 `@WasmImport` 注解:

```
import kotlin.wasm.WasmImport

@WasmImport("wasi_snapshot_preview1", "clock_time_get")
private external fun wasiRawClockTimeGet(clockId: Int, precision:
Long, resultPtr: Int): Int
```

你可以在我们的 GitHub 仓库找到完整的示例 (https://github.com/Kotlin/kotlin-wasm-examples/tree/main/wasi-example).

> 对于 `wasmWasi` 编译目标, 不能使用 与 JavaScript 的交互功能 (与 JavaScript 交互).

Kotlin/Wasm API 的改进

在这个发布版中, 包含了对 Kotlin/Wasm API 使用体验的一些改进. 例如, 你不再需要从 DOM 事件监听器中返回一个值:

1.9.20 版之前	1.9.20 版

<pre>fun main() {
    window.onload = {
 	
document.body?.sayHello(
)
        null
    }
}</pre>
	<pre>fun main() {
	    window.onload = {
	document.body?.sayHello() }
	}</pre>

Gradle

Kotlin 1.9.20 完全兼容于 Gradle 6.8.3 到 8.1 的版本. 你也可以使用最新的 Gradle 版本, 但如果你这样做, 请注意, 你可能遇到废弃警告, 或一些新的 Gradle 功能无法工作.

这个发布版带来了以下变更:

- 支持 test fixture 访问内部声明
- 用于配置 Konan 目录路径的新属性
- 对 Kotlin/Native 任务的新的构建报告统计指标

支持 test fixture 访问内部声明

在 Kotlin 1.9.20 中, 如果你使用 Gradle 的 `java-test-fixtures` plugin, 那么你的 test fixture (https://docs.gradle.org/current/userguide/java_testing.html#sec:java_test_fixtures) 现在可以访问 `main` 源代码集的类中的 `internal` 声明. 而且, 任何 test 源代码都可以访问 test fixture 类中的任何 `internal` 声明.

用于配置 Konan 目录路径的新属性

在 Kotlin 1.9.20 中, `kotlin.data.dir` Gradle 属性可以用来定义你的指向 `~/.konan` 目录的路径, 因此你不需要通过环境变量 `KONAN_DATA_DIR` 来配置它.

或者, 你可以使用 `-Xkonan-data-dir` 编译器选项, 通过 `cinterop` 和 `konanc` 工具来配置你的指向 `~/.konan` 目录的自定义路径.

对 Kotlin/Native 任务的新的构建报告统计指标

在 Kotlin 1.9.20 中, Gradle 构建报告现在包含 Kotlin/Native 任务的统计指标. 下面是一个包含这些统计指标的构建报告示例:

```
Total time for Kotlin tasks: 20.81 s (93.1 % of all tasks time)
Time    |% of Kotlin time|Task
15.24 s|73.2 %          |:compileCommonMainKotlinMetadata
5.57 s |26.8 %          |:compileNativeMainKotlinMetadata

Task ':compileCommonMainKotlinMetadata' finished in 15.24 s
Task info:
  Kotlin language version: 2.0
Time metrics:
  Total Gradle task time: 15.24 s
  Spent time before task action: 0.16 s
  Task action before worker execution: 0.21 s
  Run native in process: 2.70 s
    Run entry point: 2.64 s
Size metrics:
  Start time of task action: 2023-07-27T11:04:17

Task ':compileNativeMainKotlinMetadata' finished in 5.57 s
Task info:
  Kotlin language version: 2.0
Time metrics:
  Total Gradle task time: 5.57 s
  Spent time before task action: 0.04 s
  Task action before worker execution: 0.02 s
  Run native in process: 1.48 s
    Run entry point: 1.47 s
Size metrics:
  Start time of task action: 2023-07-27T11:04:32
```

此外, `kotlin.experimental.tryK2` 构建报告现在包含所有被编译的 Kotlin/Native 任务, 并列出使用的语言版本:

```
##### 'kotlin.experimental.tryK2' results #####
:lib:compileCommonMainKotlinMetadata: 2.0 language version
```

```
:lib:compileKotlinJvm: 2.0 language version
:lib:compileKotlinIosArm64: 2.0 language version
:lib:compileKotlinIosSimulatorArm64: 2.0 language version
:lib:compileKotlinLinuxX64: 2.0 language version
:lib:compileTestKotlinJvm: 2.0 language version
:lib:compileTestKotlinIosSimulatorArm64: 2.0 language version
:lib:compileTestKotlinLinuxX64: 2.0 language version
##### 100% (8/8) tasks have been compiled with Kotlin 2.0 #####
```

> 如果你使用 Gradle 8.0, 你可能遇到构建报告的一些问题, 尤其是启用 Gradle 配置缓存时. 这是一个已知的问题, 在 Gradle 8.1 和之后的版本中已经修正.

标准库

在 Kotlin 1.9.20 中, the Kotlin/Native 标准库进入了稳定版, 还有一些新的功能:

- 替换了 Enum 类型值的泛型函数
- 改进了 Kotlin/JS 中的 HashMap 操作的性能

替换了 Enum 类型值的泛型函数

> 这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

在 Kotlin 1.9.0 中, 枚举类的 `entries` 属性进入了稳定版. `entries` 属性是 `values()` 合成(synthetic) 函数的现代而且高性能的替代者. 作为 Kotlin 1.9.20 的一部分, 还提供了 `enumValues<T>()` 泛型函数的替代者: `enumEntries<T>()`.

> `enumValues<T>()` 函数仍然继续支持, 但我们推荐你改为使用 `enumEntries<T>()` 函数, 因为它的性能影响较小. 每次你调用 `enumValues<T>()`, 都会创建一个新的数组, 而每次你调用 `enumEntries<T>()`, 都会返回相同的 List , 这样的方式效率要高很多.

例如:

```
enum class RGB { RED, GREEN, BLUE }

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T : Enum<T>> printAllValues() {
    print(enumEntries<T>().joinToString { it.name })
}

printAllValues<RGB>()
// 输出结果为 RED, GREEN, BLUE
```

如何启用 enumEntries 函数

要试用这个功能, 请使用 `@OptIn(ExperimentalStdlibApi)` 注解来标注使用者同意(Opt-in), 并使用 1.9 或更高的语言版本. 如果你使用 Kotlin Gradle plugin 的最新版, 那么你不需要指定语言版本来试用这个功能.

Kotlin/Native 标准库进入了稳定版

在 Kotlin 1.9.0 中, 我们 解释了 ("Kotlin/Native 标准库走向稳定" in "Kotlin 1.9.0 版中的新功能") 我们为了使标准库更加接近于我们的稳定性目标而采取的行动. 在 Kotlin 1.9.20 中, 我们终于完成了这项工作, 让 Kotlin/Native 标准库进入了稳定版. 以下是这个发布版的一些重要变更:

- `Vector128` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlinx.cinterop/-vector128/) 类从 `kotlin.native` 包移动到了 `kotlinx.cinterop` 包.
- 对 Kotlin 1.9.0 中引入的 `ExperimentalNativeApi` 和 `NativeRuntimeApi` 注解的使用者同意(Opt-in)要求级别, 从 `WARNING` 级提升到了 `ERROR` 级.
- Kotlin/Native 集合现在能够检测并发的修改, 例如, 在 `ArrayList` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-array-list/) 和 `HashMap` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-hash-map/) 集合中.
- `Throwable` 类的 `printStackTrace()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-throwable/print-stack-trace.html) 函数现在会打印到 `STDERR`, 而不是 `STDOUT`.

`printStackTrace()` 的输出格式还没有稳定, 可能会发生变更.

Atomics API 的改进

在 Kotlin 1.9.0 中, 我们提到, 在 Kotlin/Native 标准库进入稳定版时, Atomics API 也会进入稳定版. Kotlin 1.9.20 还包含了以下变更:

- 引入了实验性的 `AtomicIntArray`, `AtomicLongArray`, 和 `AtomicArray<T>` 类. 这些新的类专门设计用来与 Java 的 atomic 数组保持一致, 使得它们将来能够包含进入共通的标准库中.

> ⚠ `AtomicIntArray`, `AtomicLongArray`, 和 `AtomicArray<T>` 类是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它们随时有可能变更或被删除. 要试用这些功能, 请使用 `@OptIn(ExperimentalStdlibApi)` 标注使用者同意(Opt-in). 请注意, 只为评估和试验目的来使用这些功能. 希望你能通过我们的 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

- 在 `kotlin.native.concurrent` 包中, 在 Kotlin 1.9.0 中废弃的 Atomics API, 过去的废弃级别为 `WARNING`, 现在废弃级别提升到了 `ERROR`.
- 在 `kotlin.concurrent` 包中, `AtomicInt` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/-atomic-int/index.html) 和 `AtomicLong` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/-atomic-long/index.html) 类的成员函数, 过去的废弃级别为 `ERROR`, 现在已被删除.
- `AtomicReference` 类的所有 成员函数 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/-atomic-reference/#functions) 现在使用原子化的函数.

关于 Kotlin 1.9.20 中的所有变更, 请参见我们的 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-61028/Behavioural-changes-to-the-Native-stdlib-API).

改进了 Kotlin/JS 中的 HashMap 操作的性能

Kotlin 1.9.20 改进了 Kotlin/JS 中 `HashMap` 操作的性能, 并减少了它的内存占用量. 在内部, Kotlin/JS 将它的内部实现变更为开放寻址(Open Addressing). 因此, 对于以下情况你会看到性能的改进:

- 向 `HashMap` 插入新的元素.
- 在 `HashMap` 中查找存在的元素.
- 遍历 `HashMap` 的 key 或 value.

文档更新

Kotlin 文档有了一些重要变更:

- JVM 元数据 (https://kotlinlang.org/api/kotlinx-metadata-jvm/) API 参考文档 – 查阅这个文档, 看看如何使用 Kotlin/JVM 解析元数据.
- 时间测量指南 (时间测量) – 学习如何在 Kotlin 中计算和测量时间.
- 改进了 Kotlin 观光之旅 (欢迎参加我们的 Kotlin 观光之旅!) 中关于集合的章节 – 通过理论和实践章节, 学习 Kotlin 编程语言的基础知识.
- 确定不为 null 的类型 ("确定不为 null 的类型" in "泛型(Generic): in, out, where") – 学习确定不为 null 的泛型类型.
- 改进了关于 数组 (数组) 的章节 – 学习数组, 以及在什么情况下使用它们.
- Kotlin Multiplatform 中的预期声明与实际声明 (预期声明与实际声明) – 学习 Kotlin Multiplatform 中的预期声明与实际声明机制.

安装 Kotlin 1.9.20

检查 IDE 版本

IntelliJ IDEA (https://www.jetbrains.com/idea/download/) 2023.1.x 和 2023.2.x 会自动建议将 Kotlin plugin 更新到 1.9.20 版本. IntelliJ IDEA 2023.3 会包含 Kotlin 1.9.20 plugin.

Android Studio Hedgehog (231) 和 Iguana (232) 会在后续的发布版中支持 Kotlin 1.9.20.

新的命令行编译器可以通过 GitHub 发布页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.20) 下载.

配置 Gradle 的设置

要下载 Kotlin 的 artifact 和依赖项, 请更新你的 `settings.gradle(.kts)` 文件, 使用 Maven Central 仓库:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
```

```
    }
}
```

如果没有指定仓库, Gradle 会使用已废弃的 JCenter 仓库, 导致无法下载 Kotlin artifact 的错误.

Kotlin 1.9.0 版中的新功能

最终更新: 2024/09/10

发布日期: 2023/07/06 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.9.0 已经发布了, JVM 平台的 K2 编译器已经进入 **Beta** 版. 此外, 还有以下一些重要功能:

- 新的 Kotlin K2 编译器更新
- 枚举类值函数的替代进入稳定版
- 用于终端开放(open-ended)的值范围的 `..<` 操作符进入稳定版
- 新的共通函数, 根据名称获取正规表达式中捕获的组
- 新的路径工具函数, 用于创建父目录
- Kotlin Multiplatform 中, Gradle 配置缓存功能的预览版
- Kotlin Multiplatform 中, 对支持的 Android target 的变更
- Kotlin/Native 中, 自定义内存分配器的预览版
- Kotlin/Native 中, 库的链接
- Kotlin/Wasm 中, 与编译结果大小相关的优化

关于本次更新的概要介绍, 你可以观看以下视频:

IDE 支持

在以下 IDE 中可以使用支持 1.9.0 版的 Kotlin plugin:

IDE	支持的版本
IntelliJ IDEA	2022.3.x, 2023.1.x
Android Studio	Giraffe (223), Hedgehog (231)*

*Android Studio Giraffe (223) 和 Hedgehog (231) 的后续发布版中会包含 Kotlin 1.9.0 plugin.

IntelliJ IDEA 2023.2 的后续发布版中会包含 Kotlin 1.9.0 plugin.

> ⚠ 要下载 Kotlin 的 artifact 和依赖项, 请 配置你的 Gradle 设置, 使用 Maven Central 仓库.

新的 Kotlin K2 编译器更新

JetBrains 的 Kotlin 开发组一直在努力稳定 K2 编译器, 1.9.0 版引入了更多的新功能. JVM 平台的 K2 编译器现在已进入 **Beta** 版.

对于 Kotlin/Native 和跨平台项目, 也有了基本的支持.

kapt 编译器 plugin 与 K2 编译器之间的兼容性

你可以在你的项目中和 K2 编译器一起使用 kapt plugin (kapt 编译器插件), 但存在一些限制. 即使将 `languageVersion` 设置为 `2.0`, kapt 编译器 plugin 仍然会使用旧的编译器.

如果你对一个 `languageVersion` 设置为 `2.0` 的项目执行 kapt 编译器 plugin, kapt 会自动切换到 `1.9`, 并禁用特定版本的兼容性检查. 这个行为相当于包含了下面这些命令行参数:

- `-Xskip-metadata-version-check`
- `-Xskip-prerelease-check`
- `-Xallow-unstable-dependencies`

这些检查对 kapt 任务被禁用了. 所有其他的编译任务仍然会继续使用新的 K2 编译器.

如果你在和 K2 编译器一起使用 kapt 时遇到任何问题, 请报告到我们的 问题追踪系统 (http://kotl.in/issue).

在你的项目中试用 K2 编译器

从 1.9.0 开始, 到 Kotlin 2.0 发布之前, 你可以很容易的测试 K2 编译器, 只需要向你的 `gradle.properties` 文件添加 `kotlin.experimental.tryK2=true` Gradle 属性就可以了. 你也可以运行以下命令:

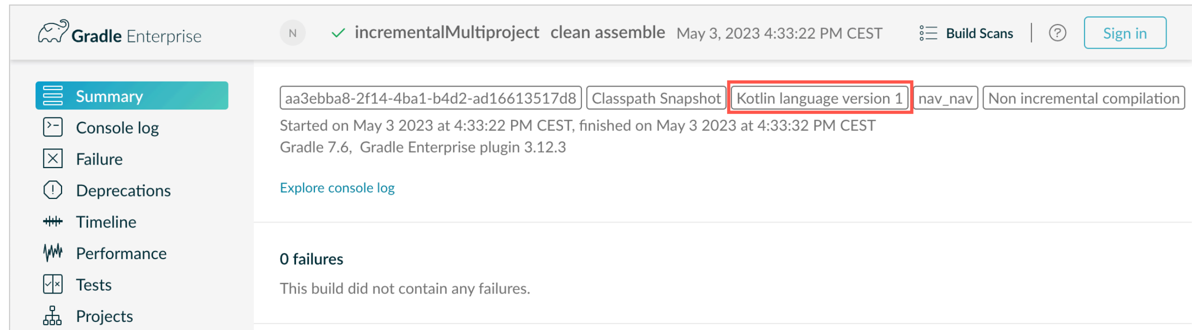
```
./gradlew assemble -Pkotlin.experimental.tryK2=true
```

这个 Gradle 属性会自动将语言版本设置为 2.0, 而且会更新构建报告, 包括 Kotlin 编译任务中, 使用 K2 编译器和使用当前编译器的任务数量:

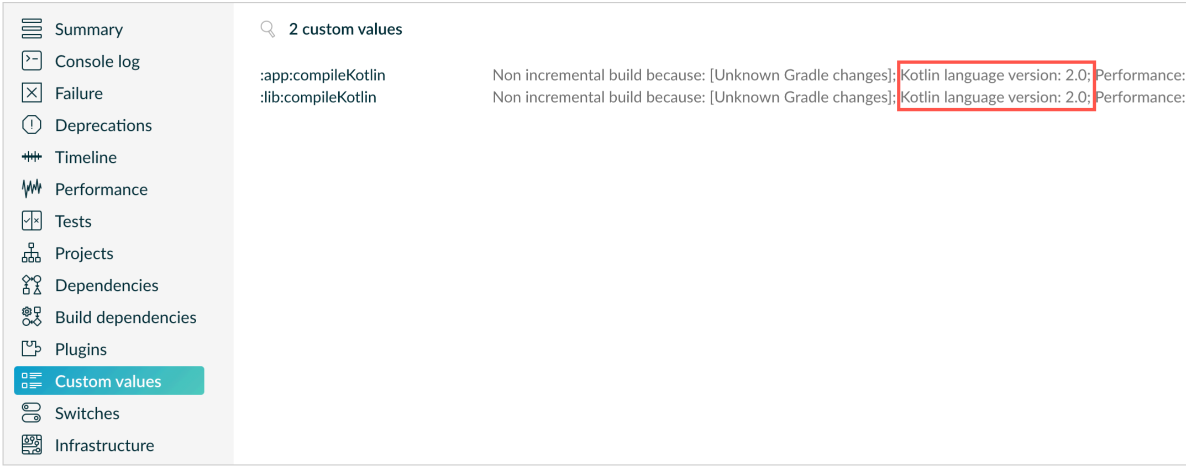
```
##### 'kotlin.experimental.tryK2' results (Kotlin/Native not checked) #####
:lib:compileKotlin: 2.0 language version
:app:compileKotlin: 2.0 language version
##### 100% (2/2) tasks have been compiled with Kotlin 2.0 #####
```

Gradle 构建报告

Gradle 构建报告 ("构建报告" in "Kotlin Gradle plugin 中的编译与缓存") 现在会显示编译代码时使用的是当前编译器还是 K2 编译器. 在 Kotlin 1.9.0 中, 你可以在你的 Gradle build scan (https://scans.gradle.com/) 中看到这些信息:



Gradle build scan - 使用 K1 编译器



Gradle build scan - 使用 K2 编译器

你还可以在构建报告中看到项目中使用的 Kotlin 版本:

```
Task info:
  Kotlin language version: 1.9
```

> 如果你使用 Gradle 8.0, 你可能遇到构建报告的一些问题, 尤其是启用 Gradle 配置缓存时. 这是一个已知的问题, 在 Gradle 8.1 和之后的版本中已经修正.

K2 编译器目前的限制

在你的 Gradle 项目中启用 K2 存在一些限制, 对使用 Gradle 8.3 以下版本的项目, 下面的情况可能会有影响:

- `buildSrc` 中源代码的编译.
- 在被包含的构建中的 Gradle plugin 的编译.
- 在 Gradle 8.3 以下版本的项目中使用的其他 Gradle plugin 的编译.
- Gradle plugin 依赖项的构建.

如果你遇到上面提到的问题, 你可以通过以下步骤来解决:

- 对 `buildSrc`, 任何 Gradle plugin, 以及它们的依赖项, 设置语言版本:

```
kotlin {
    compilerOptions {

languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KO
TLIN_1_9)

apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_
1_9)
    }
}
```

- 当 Gradle 8.3 可以使用时, 将你的项目的 Gradle 版本更新到 8.3.

留下你对于新 K2 编译器的反馈意见

如果你能提供你的反馈意见, 我们将会非常感谢!

* 在 Kotlin Slack 频道中, 直接向 K2 开发者提供你的反馈意见 – 获得邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up), 并加入 #k2-early-adopters (https://kotlinlang.slack.com/archives/C03PK0PE257) 频道.
* 在 我们的问题追踪系统 (https://kotl.in/issue) 中, 报告你遇到的新 K2 编译器的问题.
* 启用 **Send usage statistics** 选项 (https://www.jetbrains.com/help/idea/settings-usage-statistics.html), 允许 JetBrains 收集关于 K2 使用状况的匿名数据..

语言功能特性

在 Kotlin 1.9.0 中, 一些以前版本引入的新语言功能特性升级到了稳定版:

* 枚举类值函数的替代
* 数据对象与数据类的对称性
* 在内联的值类(inline value class)中支持有 body 的次级构造器(secondary constructor)

枚举类值函数的替代进入稳定版

在 1.8.20 中, 引入了实验性功能: 枚举类的 `entries` 属性. `entries` 属性是 `values()` 合成(synthetic) 函数的现代而且高性能的替代者. 在 1.9.0 中, `entries` 属性进入了稳定版.

> ⚠ `values()` 函数仍然继续支持, 但我们推荐你改为使用 `entries` 属性.

```
enum class Color(val colorName: String, val rgb: String) {
    RED("Red", "#FF0000"),
    ORANGE("Orange", "#FF7F00"),
    YELLOW("Yellow", "#FFFF00")
}

fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb ==
rgb }
```

关于枚举类的 `entries` 属性, 更多详情请参见 Kotlin 1.8.20 的新功能 ("枚举类值函数的现代而且高性能的替代者" in "Kotlin 1.8.20 版中的新功能").

数据对象与数据类的对称性进入稳定版

在 Kotlin 1.8.20 ("与数据类(Data Class)对称的数据对象(Data Object) (预览版)" in "Kotlin 1.8.20 版中的新功能") 中引入了数据对象的声明, 现在进入了稳定版. 包括为了与数据类保持对称而添加的函数: `toString()`, `equals()`, 和 `hashCode()`.

这个功能在 `sealed` 类型层级结构中非常有用 (例如一个 `sealed class` 或 `sealed interface` 层级结构), 因为 `data object` 声明可以与 `data class` 声明一起方便的使用. 在这个示例中, 将 `EndOfFile` 声明为 `data object`, 而不是普通的 `object`, 代表它自动拥有 `toString()` 函数, 不需要手动的覆盖这个函数. 这样就保持了与相应的数据类定义的对称性.

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // 输出结果为 Number(number=7)
    println(EndOfFile) // 输出结果为 EndOfFile
}
```

更多详情, 请参见 Kotlin 1.8.20 的新功能 ("与数据类(Data Class)对称的数据对象(Data Object) (预览版)" in "Kotlin 1.8.20 版中的新功能").

在内联的值类(inline value class)中支持有 body 的次级构造器(secondary constructor)

从 Kotlin 1.9.0 开始, 内联的值类(inline value class) (内联的值类(Inline value class)) 中有 body 的次级构造器(secondary constructor) 默认可以使用了:

```
@JvmInline
value class Person(private val fullName: String) {
    // 从 Kotlin 1.4.30 开始可以使用:
    init {
        check(fullName.isNotBlank()) {
            "Full name shouldn't be empty"
        }
    }
    // 从 Kotlin 1.9.0 开始默认可以使用:
    constructor(name: String, lastName: String) : this("$name
$lastName") {
        check(lastName.isNotBlank()) {
```

```
                "Last name shouldn't be empty"
            }
        }
}
```

以前, Kotlin 在内联类中只允许使用 public 的主构造器. 这就造成, 无法封装底层值, 或创建一个内联类来表达某些受限定的值.

随着 Kotlin 的发展, 解决了这个问题. Kotlin 1.4.30 取消了对 `init` 代码块的限制, 之后, Kotlin 1.8.20 提供了预览功能, 允许使用有 body 的次级构造器. 现在这个功能默认可以使用了. 关于 Kotlin 内联类的开发进程, 请参见 这个 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/inline-classes.md).

Kotlin/JVM

从 version 1.9.0 来时, 编译器能够生成字节码版本对应于 JVM 20的类. 此外, `JvmDefault` 注解和旧的 `-Xjvm-default` 模式的废弃周期继续向前推进.

JvmDefault 注解和旧的 -Xjvm-default 模式的废弃

从 Kotlin 1.5 开始, `JvmDefault` 注解的使用被废弃了, 取代它的是新的 `-Xjvm-default` 模式: `all` 和 `all-compatibility`. 随着 Kotlin 1.4 中引入的 `JvmDefaultWithoutCompatibility`, 以及 Kotlin 1.6 中引入的 `JvmDefaultWithCompatibility`, 这些模式提供了对 `DefaultImpls` 类的生成的全面的控制, 并确保与旧的 Kotlin 代码无缝的兼容性.

因此, 在 Kotlin 1.9.0 中, `JvmDefault` 注解不再具有任何意义, 并被标注为已废弃, 使用它会产生编译错误. 它最终将会从 Kotlin 中完全删除.

Kotlin/Native

除其他改进之外, 这个发布版还带来了 Kotlin/Native 内存管理器 (Kotlin/Native 内存管理) 的更多改进, 将会增强它的健壮性和性能:

- 自定义内存分配器的预览版
- 主线程上的 Objective-C 或 Swift 对象释放 hook
- 在 Kotlin/Native 中访问常数值时不会初始化对象
- 能够为 iOS 模拟器上的测试配置 standalone 模式
- Kotlin/Native 中库的链接

自定义内存分配器的预览版

Kotlin 1.9.0 引入了自定义内存分配器的预览版. 它的分配系统能够提高 Kotlin/Native 内存管理器 (Kotlin/Native 内存管理) 的运行期性能.

Kotlin/Native 中目前的对象分配系统使用一个一般性的分配器, 不能实现高效的垃圾收集. 作为补偿, 在垃圾收集器 (GC) 将所有已分配的对象合并入单个列表之前 它维护一个线程局部的(thread-local)链表, 其中包含已分配的对象, 这个列表可以在清理过程中遍历. 这种方案造成了几个性能缺陷:

- 清理顺序缺乏内存局部性(memory locality), 并且经常导致分散的内存访问模式, 造成潜在的性能问题.
- 链表对每个对象需要更多内存, 增加了内存使用量, 尤其是在处理大量的小对象的情况下.
- 包含所有已分配对象的单个列表使得难以进行并行清理, 当转换器线程(Mutator Thread)分配对象的速度超过 GC 线程回收它们的速度时, 可能造成内存使用量的问题.

为了解决这些问题, Kotlin 1.9.0 引入了自定义内存分配器的预览版. 它将系统内存分为多个页面(Page), 允许按连续的顺序进行独立的清理. 每次分配的内存都会成为一个页面(Page)内的内存块(Memory Block), 并且页面会追踪各个块的大小. 各种不同的页面类型进行了不同的优化, 以适应于不同的内存分配大小. 内存块的连续排列保证了可以对所有的分配块进行高效的迭代.

当一个线程分配内存时, 它会根据分配的大小搜索适当的页面. 线程会根据不同的大小类别维护一组页面. 对于一个确定的大小, 当前页通常可以容纳这个内存分配. 如果不能, 那么线程会从共享的分配空间请求一个不同的页面. 这个页面的状态可能是可用, 需要清理, 或需要创建.

新的内存分配器允许同时使用多个多个独立的分配空间, 因此 Kotlin 开发组可以实验不同的页面布局, 进一步提高性能.

关于新的内存分配器的设计, 更多详情请参见 README (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/runtime/src/alloc/custom/README.md).

如何启用

添加 `-Xallocator=custom` 编译器选项:

```
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
```

```
            compilerOptions.configure {
                freeCompilerArgs.add("-Xallocator=custom")
            }
        }
    }
}
```

留下你的反馈意见

希望你能通过 YouTrack (https://youtrack.jetbrains.com/issue/KT-55364/Implement-custom-allocator-for-Kotlin-Native) 提供你的反馈意见, 帮助改进自定义分配器.

主线程上的 Objective-C 或 Swift 对象释放 hook

从 Kotlin 1.9.0 开始, 对于 Objective-C 或 Swift 对象, 如果对象在主线程中被传递到 Kotlin, 那么对象的释放 hook 也会在主线程上被调用. Kotlin/Native 内存管理器 (Kotlin/Native 内存管理) 以前处理 Objective-C 对象引用的方式可能会导致内存泄露. 我们相信现在的新的行为可以改进内存管理器的健壮性.

考虑一个被 Kotlin 代码引用的 Objective-C 对象, 例如, 当对象作为参数传递时, 被函数返回时, 或者从一个集合获取时. 这种情况下, Kotlin 创建它自己的对象, 其中保持 Objective-C 对象的引用. 当 Kotlin 对象被释放时, Kotlin/Native 运行期库会调用 `objc_release` 函数, 释放 Objective-C 对象的引用.

在以前的版本中, Kotlin/Native 内存管理器在一个特殊的 GC 线程中运行 `objc_release`. 如果它是这个对象的最后引用, 那么对象会被释放. 问题发生在, 如果 Objective-C 对象有自定义的释放 hooks, 例如 Objective-C 中的 `dealloc` 方法, 或 Swift 中的 `deinit` 代码块, 这些 hook 期望在特定的线程上调用.

由于主线程中的对象的 hook 通常也期望在主线程中调用, Kotlin/Native 运行期库现在也在主线程上调用 `objc_release`. 它应该覆盖 Objective-C 对象在主线程上传递到 Kotlin, 并在主线程中创建一个 Kotlin 端的对等对象的情况. 这只对处理主调度队列的情况才有效， 对于通常的 UI 应用程序就是这种情况. 如果不是主调度队列, 或者对象在主线程以外的线程中传递到 Kotlin 的情况, 会和以前一样, 在特殊的 GC 线程中调用 `objc_release`.

如何关闭这个功能

如果你遇到问题, 你可以在你的 `gradle.properties` 文件中, 添加以下选项, 禁用这个行为:

```
kotlin.native.binary.objcDisposeOnMain=false
```

遇到这样的情况, 请报告到 我们的问题追踪系统 (https://kotl.in/issue).

在 Kotlin/Native 中访问常数值时不会初始化对象

从 Kotlin 1.9.0 开始, 在访问 `const val` 域变量时, Kotlin/Native 后端不会初始化对象:

```
object MyObject {
    init {
        println("side effect!")
    }

    const val y = 1
}

fun main() {
    println(MyObject.y) // 第 1 次不会初始化
    val x = MyObject    // 这里会发生初始化
    println(x.y)
}
```

这个行为现在与 Kotlin/JVM 平台统一了, Kotlin/JVM 平台的实现与 Java 一致, 对这种情况对象永远不会初始化. 由于这个变化, 你的 Kotlin/Native 项目还能够有一些性能改进.

能够为 iOS 模拟器上的测试配置 standalone 模式

默认情况下, 在对 Kotlin/Native 运行 iOS 模拟器上的测试时, 会使用 `--standalone` 选项, 以避免发生手动的模拟器启动和关闭. 在 1.9.0 中, 现在你可以在 Gradle task 中通过 `standalone` 属性配置是否使用这个选项. 默认会使用 `--standalone` 选项, 启用 standalone 模式.

下面的例子演示在你的 `build.gradle.kts` 文件中如何禁用 standalone 模式:

```
tasks.withType<org.jetbrains.kotlin.gradle.targets.native.tasks.Kotl
inNativeSimulatorTest>().configureEach {
    standalone.set(false)
}
```

> ⚠ 如果你禁用 standalone 模式, 那么必须手动启用模拟器. 要从 CLI 启动你的模拟器, 可以使用下面的命令:
>
> ```
> /usr/bin/xcrun simctl boot <DeviceId>
> ```

Kotlin/Native 中库的链接

从 Kotlin 1.9.0 开始, Kotlin/Native 编译器使用与 Kotlin/JVM 相同的方式来处理 Kotlin 库的链接问题. 如果一个第三方 Kotlin 库的作者对实验性 API 进行了不兼容的变更, 而这个 API 又被另一个第三方 Kotlin 库使用, 那么你就可能遇到这样的问题.

对于第三方 Kotlin 库之间发生链接错误的情况, 构建不会在编译过程中失败. 相反, 你只会在运行期间遇到这些错误, 这种行为与 JVM 完全相同.

每当 Kotlin/Native 编译器检测到库链接的问题就会报告警告. 你可以在你的编译日志中找到这样的警告, 例如:

```
No function found for symbol
'org.samples/MyRemovedClass.doSomething|3657632771909858561[0]'

Can not get instance of singleton 'MyEnumClass.REMOVED_ENTRY': No
enum entry found for symbol
'org.samples/MyEnumClass.REMOVED_ENTRY|null[0]'

Function 'getMyRemovedClass' can not be called: Function uses
unlinked class symbol 'org.samples/MyRemovedClass|null[0]'
```

在你的项目中, 你可以进一步配置, 甚至禁用这样的行为:

- 如果你不想在你的编译日志中看到这些警告, 可以使用 `-Xpartial-linkage-loglevel=INFO` 编译器选项来禁止警告.
- 也可以使用 `-Xpartial-linkage-loglevel=ERROR`, 将报告的警告级别提升为编译错误. 这种情况下, 编译会失败, 你会在编译日志中看到所有的错误. 使用这个选项可以更加严密的检测链接错误.
- 如果你在使用这个功能时遇到意想不到的问题, 你可以使用 `-Xpartial-linkage=disable` 编译器选项关闭这个功能. 遇到这样的情况, 请报告到 我们的问题追踪系统 (https://kotl.in/issue).

```
// 通过 Gradle 构建文件传递编译器选项的示例.
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
```

```
                // 禁止链接警告:
                freeCompilerArgs.add("-Xpartial-linkage-
loglevel=INFO")

                // 将链接警告提升为错误:
                freeCompilerArgs.add("-Xpartial-linkage-
loglevel=ERROR")

                // 完全禁用这个功能:
                freeCompilerArgs.add("-Xpartial-linkage=disable")
            }
        }
    }
}
```

用于与 C 代码交互时的隐式整数转换的编译器选项

我们引入了与 C 代码交互时的一个编译器选项, 允许你使用隐式整数转换. 经过仔细考虑之后, 我们引入了这个编译器选项, 以防止无意的使用, 因为这个功能还有待继续改进, 而我们的目标是拥有最高质量的 API.

下面的示例代码中, 一个隐式整数转换允许 `options = 0`, 尽管 `options` (https://developer.apple.com/documentation/foundation/nscalendar/options) 是无符号的 `UInt` 类型, 而 `0` 是有符号的整数.

```
val today = NSDate()
val tomorrow = NSCalendar.currentCalendar.dateByAddingUnit(
    unit = NSCalendarUnitDay,
    value = 1,
    toDate = today,
    options = 0
)
```

要对原生库使用隐式转换, 请使用 `-XXLanguage:+ImplicitSignedToUnsignedIntegerConversion` 编译器选项.

你可以在你的 Gradle `build.gradle.kts` 文件中进行配置:

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile
>().configureEach {
```

```
    compilerOptions.freeCompilerArgs.addAll(
        "-XXLanguage:+ImplicitSignedToUnsignedIntegerConversion"
    )
}
```

Kotlin Multiplatform

在 1.9.0 中, Kotlin Multiplatform 有了以下重要更新, 旨在改善你的开发者体验:

- 对支持的 Android target 的变更
- 默认启用新的 Android 源代码集布局
- 在跨平台项目中的 Gradle 配置缓存功能的预览版

对支持的 Android target 的变更

我们正在继续努力稳定 Kotlin Multiplatform. 其中必要的一步是为 Android target 提供一级支持. 我们很激动的宣布, 将来, Google 的 Android 开发组将会提供他们自己的 Gradle plugin, 来支持 Kotlin Multiplatform 中的 Android.

为了给这个来自 Google 的新解决方案开辟道路, 我们会重命名 1.9.0 的目前的 Kotlin DSL 中的 `android` 代码块. 请将你的构建脚本中的所有 `android` 代码块改为 `androidTarget`. 这是一个必要的临时变更, 目的是将 `android` 的名称留给未来由 Google 提供的 DSL 使用.

Google plugin 将成为在跨平台项目中使用 Android 的首选方式. 当它完成之后, 我们会提供必要的迁移说明, 让你能够象以前一样使用 `android` 的短名称.

默认启用新的 Android 源代码集布局

从 Kotlin 1.9.0 开始, 默认会使用新的 Android 源代码集布局. 它取代了以前的目录命名模式, 这个旧模式在很多方面令人难以理解. 新布局有很多优点:

- 简化的类型语义 – 新的 Android 源代码集布局提供了清晰而且一致的命名规约, 有助于区分不同类型的源代码集.
- 改进的源代码目录布局 – 使用新的布局, `SourceDirectories` 的排列变得更加连贯, 更易于组织代码和定位源代码文件.
- 清晰的 Gradle 配置命名模式 – 在 `KotlinSourceSets` 和 `AndroidSourceSets` 中, 命名模式现在更加一致, 更加易于预测.

新的布局需要使用 Android Gradle plugin 7.0 或更高版本, 以及 Android Studio 2022.3 或更高版本. 请参见我们的 迁移向导 (Android 源代码集布局), 在你的 `build.gradle(.kts)` 文件中进行必要的修改.

Gradle 配置缓存功能的预览版

Kotlin 1.9.0 增加了对跨平台库中的 Gradle 配置缓存 (https://docs.gradle.org/current/userguide/configuration_cache.html) 的支持. 如果你是库的作者, 你可以得益于构建性能的改善.

Gradle 配置缓存通过对后续的构建重用配置阶段的结果来加快构建过程. 这个功能从 Gradle 8.1 开始成为稳定版. 要启用它, 请遵照 Gradle 文档 (https://docs.gradle.org/current/userguide/configuration_cache.html#config_cache:usage) 中的说明.

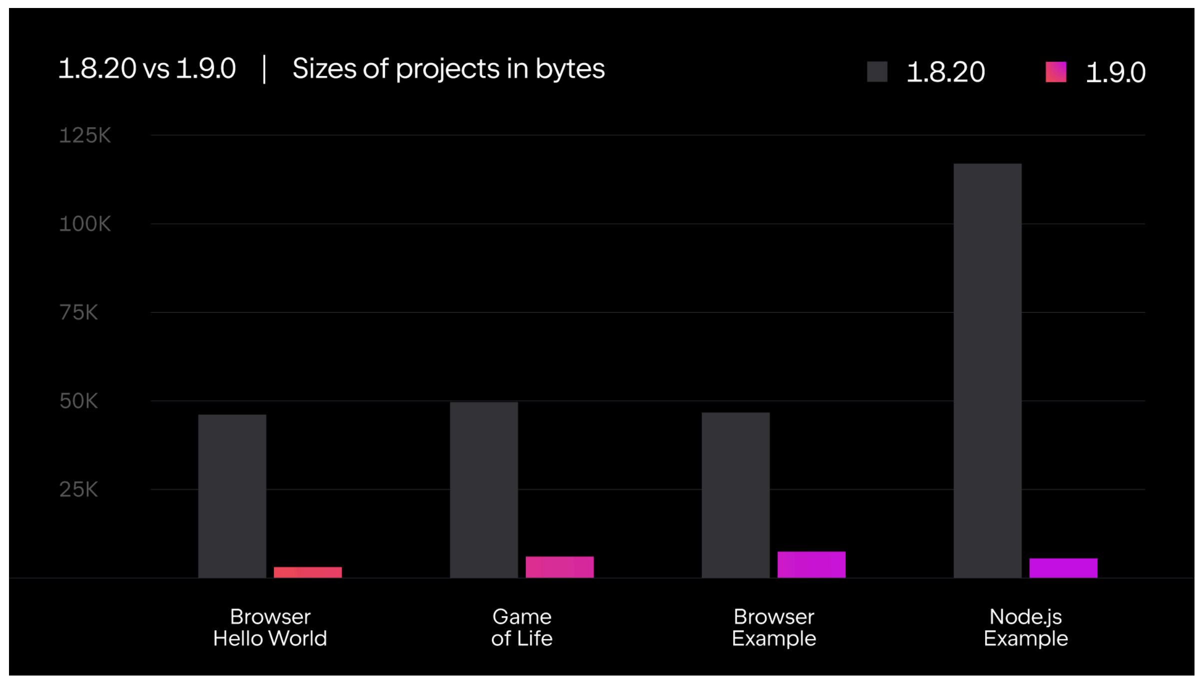
> 对于与 Xcode 集成的 task, 或 Kotlin CocoaPods Gradle plugin (CocoaPods Gradle plugin DSL 参考文档), Kotlin Multiplatform plugin 还不支持 Gradle 配置缓存. 我们期望在未来的 Kotlin 发布版中添加这个功能.

Kotlin/Wasm

Kotlin 开发组还在继续实验新的 Kotlin/Wasm 编译目标. 这个发布版引入了几个性能优化和 与编译结果大小相关的优化, 以及 与 JavaScript 交互功能的更新.

与编译结果大小相关的优化

对 WebAssembly (Wasm) 项目, Kotlin 1.9.0 引入了编译结果大小的显著改善. 比较两个 "Hello World" 项目, Kotlin 1.9.0 中的 Wasm 代码大小比 Kotlin 1.8.20 中要小超过 10 倍以上.



Kotlin/Wasm 与编译结果大小相关的优化

在使用 Kotlin 代码针对 Wasm 平台进行开发时, 这些代码大小优化可以更加高效的利用资源, 并改善性能.

与 JavaScript 交互功能的更新

这次 Kotlin 更新引入了 Kotlin/Wasm 的 Kotlin 与 JavaScript 之间交互能力的变更. 由于 Kotlin/Wasm 是一个 实验性 ("稳定性级别" in "Kotlin 各部分组件的稳定性") 功能, 它的互操作性存在一些限制.

动态类型的限制

从 1.9.0 版开始, Kotlin 在 Kotlin/Wasm 中不再支持使用 `Dynamic` 类型. 这个功能现在已被废弃, 由新的通用的 `JsAny` 类型取代, 这个类型游离于 JavaScript 互操作性.

更多详情, 请参见 Kotlin/Wasm 与 JavaScript 的互操作性 (与 JavaScript 交互) 文档.

非外部类型(non-external type)的限制

Kotlin/Wasm 在向 JavaScript 传递值时, 或从 JavaScript 传入值时, 支持对特定的 Kotlin 静态的转换. 支持的类型包括:

- 基本类型, 例如有符号的数值, `Boolean`, 以及 `Char`.
- `String`.
- 函数类型.

其他类型传递时不会转换, 而是作为不透明引用(Opaque Reference), 导致 JavaScript 与 Kotlin 子类型之间的不一致.

为了解决这个问题, Kotlin 在与 JavaScript 交互时, 限制为只允许使用一组良好支持的类型. 从 Kotlin 1.9.0 开始, 在 Kotlin/Wasm 的 JavaScript 交互中, 只支持外部(external) 类型, 基本类型, 字符串, 以及函数类型. 此外, 引入了一个单独的显式类型, 名为 `JsReference`, 用来表达可在 JavaScript 交互中使用的 Kotlin/Wasm 对象句柄.

更多详情, 请参见 Kotlin/Wasm 与 JavaScript 的互操作性 (与 JavaScript 交互) 文档.

Kotlin Playground 中的 Kotlin/Wasm

Kotlin Playground 支持 Kotlin/Wasm 编译目标. 你可以编写, 运行, 分享你的针对 Kotlin/Wasm 编译目标的 Kotlin 代码. 马上看看吧 (https://pl.kotl.in/HDFAvimga)

> 使用 Kotlin/Wasm 需要在你的浏览器中启用实验性的功能.
>
> 参见: 如何启用这些功能 (问题分析).

```
import kotlin.time.*
import kotlin.time.measureTime

fun main() {
    println("Hello from Kotlin/Wasm!")
    computeAck(3, 10)
}

tailrec fun ack(m: Int, n: Int): Int = when {
    m == 0 -> n + 1
    n == 0 -> ack(m - 1, 1)
    else -> ack(m - 1, ack(m, n - 1))
}

fun computeAck(m: Int, n: Int) {
    var res = 0
    val t = measureTime {
        res = ack(m, n)
    }
    println()
    println("ack($m, $n) = ${res}")
```

```
    println("duration: ${t.inWholeNanoseconds / 1e6} ms")
}
```

Kotlin/JS

这个发布版引入了 Kotlin/JS 的更新, 包括删除了旧的 Kotlin/JS 编译器, 废弃了 Kotlin/JS Gradle plugin, 以及实验性的支持 ES6:

- 删除了旧的 Kotlin/JS 编译器
- 废弃了 Kotlin/JS Gradle plugin
- 废弃了外部枚举类型(external enum)
- 实验性的支持 ES6 类和模块
- 更改了 JS 产品发布(production distribution)的默认目标
- 从 stdlib-js 中抽取了 org.w3c 声明

> 从 1.9.0 版开始, 对 Kotlin/JS 还启用了 部分的库链接.

删除了旧的 Kotlin/JS 编译器

在 Kotlin 1.8.0 中, 我们 宣布了 ("JS IR 编译器后端的稳定版" in "Kotlin 1.8.0 版中的新功能") 基于 IR 的后端已成为 稳定版 (Kotlin 各部分组件的稳定性). 从那之后, 不指定编译器成为一种错误, 使用旧的编译器会导致警告.

在 Kotlin 1.9.0 中, 使用旧的后端会导致错误. 请遵照我们的 迁移指南 (将 Kotlin/JS 项目迁移到 IR 编译器), 迁移到 IR 编译器.

废弃了 Kotlin/JS Gradle plugin

从 Kotlin 1.9.0 开始, `kotlin-js` Gradle plugin 已被废弃. 我们建议你改为使用 `kotlin-multiplatform` Gradle plugin 中的 `js()` 编译目标.

Kotlin/JS Gradle plugin 的功能本质上与 `kotlin-multiplatform` plugin 是重叠的, 并使用了相同的内部实现. 这种功能重叠导致了理解困难, 并增加了 Kotlin 开发组的维护负担.

关于迁移说明, 请参见我们的 Kotlin Multiplatform 兼容性指南 (Kotlin Multiplatform 兼容性指南). 如果你遇到迁移指南中没有提到的其它问题, 请报告到我们的 问题追踪系统 (http://kotl.in/issue).

废弃了外部枚举类型(external enum)

在 Kotlin 1.9.0 中, 外部枚举类型(external enum)的使用将被废弃, 原因是枚举类型的静态成员, 例如 `entries`, 不能存在于 Kotlin 之外. 我们建议改为使用外部的封闭类, 并以对象作为它的子类:

```
// 以前的代码
external enum class ExternalEnum { A, B }

// 现在的代码
external sealed class ExternalEnum {
    object A: ExternalEnum
    object B: ExternalEnum
}
```

通过切换为以对象为子类的外部封闭类, 你可以实现与外部枚举类型相似的功能, 同时又能避免与默认方法相关的问题.

从 Kotlin 1.9.0 开始, 外部枚举类型的使用将被标记为废弃. 我们建议你更新你的代码, 使用上面建议的外部封闭类来实现, 以保证兼容性, 并有利于未来的维护.

实验性的支持 ES6 类和模块

本次发布引入了对 ES6 模块和生成 ES6 类的 实验性 ("稳定性级别" in "Kotlin 各部分组件的稳定性") 支持:

- 模块提供了一种方式, 简化你的代码库, 并提高可维护性.
- 类允许你结合面向对象编程 (OOP) 原则, 产生更加清晰直观的代码.

要启用这些功能, 请更新你的 `build.gradle.kts` 文件:

```
// build.gradle.kts
kotlin {
    js(IR) {
        useEsModules() // 启用 ES6 模块
        browser()
    }
}

// 启用 ES6 类的生成
```

```
tasks.withType<KotlinJsCompile>().configureEach {
    kotlinOptions {
        useEsClasses = true
    }
}
```

关于ECMAScript 2015 (ES6), 更多详情请参见 官方文档 (https://262.ecma-international.org/6.0/).

更改了 JS 产品发布(production distribution)的默认目标

在 Kotlin 1.9.0 之前, 发布的目标目录是 `build/distributions`. 但是, 这是一个用于 Gradle archive 的共通目录. 为了解决这个问题, 在 Kotlin 1.9.0 中我们将默认的发布目标目录改为: `build/dist/<targetName>/<binaryName>`.

例如, `productionExecutable` 过去会发布到 `build/distributions`. 在 Kotlin 1.9.0 中, 它会发布到 `build/dist/js/productionExecutable`.

> ⚠ 如果你有一个使用这些构建结果的管道, 请确认更新目录的设置.

从 stdlib-js 中抽取了 org.w3c 声明

从 Kotlin 1.9.0 开始, `stdlib-js` 不再包含 `org.w3c` 声明. 这些声明改为移动到一个单独的 Gradle 依赖项中. 当你向你的 `build.gradle.kts` 文件添加 Kotlin Multiplatform Gradle plugin 时, 这些声明会自动包含到你的项目中, 和标准库类似.

不需要任何手动的迁移处理. 必要的调整工作会自动处理.

Gradle

Kotlin 1.9.0 带来了新的 Gradle 编译器选项, 以及很多其他功能:

- 删除了 classpath 属性
- 新的 Gradle 编译器选项
- Kotlin/JVM 的项目级编译器选项
- 用于 Kotlin/Native 模块名称的编译器选项
- 用于 Kotlin 官方库的单独的编译器 plugin

* 增加了最低支持版本
* kapt 不再过早创建 task
* JVM 编译目标校验模式的程序化配置

删除了 classpath 属性

在 Kotlin 1.7.0 中, 我们宣布了 `KotlinCompile` task 属性 `classpath` 废弃周期的开始. 在 Kotlin 1.8.0 中废弃级别提升到了 `ERROR`. 在本次发布版中, 我们最终删除了 `classpath` 属性. 所有的编译任务现在应该使用 `libraries` 输入, 得到编译所需要的库的列表.

新的编译器选项

Kotlin Gradle plugin 现在提供新的属性, 用于使用者同意(Opt-in), 以及编译器的渐进模式 (progressive mode).

* 要对新的 API 标注使用者同意(Opt-in), 现在你可以使用 `optIn` 属性, 传递一个字符串列表, 例如: `optIn.set(listOf(a, b, c))`.
* 要启用渐进模式, 请使用 `progressiveMode.set(true)`.

Kotlin/JVM 的项目级编译器选项

从 Kotlin 1.9.0 开始, 在 `kotlin` 配置代码块中, 可以使用一个新的 `compilerOptions` 代码块:

```
kotlin {
    compilerOptions {
        jvmTarget.set(JVM.Target_11)
    }
}
```

这个功能使得编译器选项的配置更加容易. 但是, 需要注意一些重要的细节:

* 这个配置只适用于项目级.
* 对于 Android plugin, 这个代码块与下面的代码配置相同的对象:

```
android {
    kotlinOptions {}
```

```
}
```

* `android.kotlinOptions` 和 `kotlin.compilerOptions` 配置块会相互覆盖. 只有构建文件中最后出现的 (最下方的) 代码块会起作用.
* 如果在项目级配置了 `moduleName`, 它的值在传递给编译器时可能会变更. 对 `main` 编译不会如此, 但对其它编译类型, 例如, test source, Kotlin Gradle plugin 会添加 `_test` 后缀.
* `tasks.withType<KotlinJvmCompile>().configureEach {}` (或 `tasks.named<KotlinJvmCompile>("compileKotlin") { }`) 之内的配置会覆盖 `kotlin.compilerOptions` 和 `android.kotlinOptions`.

用于 Kotlin/Native 模块名称的编译器选项

在 Kotlin Gradle plugin 中现在可以很容易的使用 Kotlin/Native 的 `module-name` ("-module-name name (Native)" in "Kotlin 编译器选项") 编译器选项.

这个选项对编译的模块指定一个名称, 也可以为导入到 Objective-C 的声明添加一个名称前缀.

你可以直接在你的 Gradle 构建文件的 `compilerOptions` 代码块中设置模块名称:

Kotlin

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompil
e>("compileKotlinLinuxX64") {
    compilerOptions {
        moduleName.set("my-module-name")
    }
}
```

Groovy

```
tasks.named("compileKotlinLinuxX64",
org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile.class) {
    compilerOptions {
        moduleName = "my-module-name"
    }
}
```

用于 Kotlin 官方库的单独的编译器 plugin

Kotlin 1.9.0 为它的官方库引入了单独的编译器 plugin. 以前, 编译器 plugin 内嵌在对应的 Gradle plugin 中. 如果编译器 plugin 编译时使用的 Kotlin 版本比 Gradle build 的 Kotlin 运行期版本更高, 就可能导致兼容性问题.

新的编译器 plugin 添加为单独的依赖项, 因此你不会再遇到与旧版本 Gradle 的兼容性问题. 新方案的另一个主要优点是, 新的编译器 plugin 可以在其他构建系统中使用, 例如 Bazel (https://bazel.build/).

以下是我们发布到 Maven Central 的新编译器 plugin 的列表:

- kotlin-atomicfu-compiler-plugin
- kotlin-allopen-compiler-plugin
- kotlin-lombok-compiler-plugin
- kotlin-noarg-compiler-plugin
- kotlin-sam-with-receiver-compiler-plugin
- kotlinx-serialization-compiler-plugin

每个 plugin 都有它对应的 `-embeddable`, 例如, `kotlin-allopen-compiler-plugin-embeddable` 用来与 `kotlin-compiler-embeddable` artifact 一起使用, 这是脚本化 artifact 的默认选项.

Gradle 将这些 plugin 添加为编译器参数. 你不需要对你既有的项目进行任何变更.

增加了最低支持版本

从 Kotlin 1.9.0 开始, 支持的 Android Gradle plugin 最低版本是 4.2.2.

参见 Kotlin Gradle plugin 与可用的 Gradle 版本之间的兼容性 ("应用(Apply) Kotlin Gradle Plugin" in "配置 Gradle 项目").

kapt 不再过早创建 Gradle 中的 task

在 1.9.0 之前, kapt 编译器 plugin (kapt 编译器插件) 会请求配置后的 Kotlin 编译 task 实例, 导致过早的创建 task. 在 Kotlin 1.9.0 中已经解决了这个问题. 如果你的 `build.gradle.kts` 文件使用默认的配置, 那么你的设置不会受到这个变更的影响.

> ⚠ 如果你使用自定义的配置, 你的设置会受到不利的影响. 例如, 如果你使用 Gradle 的 task API 修改了 `KotlinJvmCompile` task, 你必须在你的构建脚本中对 `KaptGenerateStubs`

task 进行类似的修改.

例如, 如果你的脚本对 `KotlinJvmCompile` task 的配置如下:

```
tasks.named<KotlinJvmCompile>("compileKotlin") { // 你的自定义配置 }
```

这种情况下, 你需要确定 `KaptGenerateStubs` task 中也包含相同的修改:

```
tasks.named<KaptGenerateStubs>("kaptGenerateStubs") { // 你的自定义配置 }
```

更多详情, 请参见我们的 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-54468/KAPT-Gradle-plugin-causes-eager-task-creation).

JVM 编译目标校验模式的程序化配置

在 Kotlin 1.9.0 之前, 只有一种方法来调整 Kotlin 与 Java 之间的 JVM 编译目标不兼容性的检测方式. 你必须在你的 `gradle.properties` 文件中对整个项目设置 `kotlin.jvm.target.validation.mode=ERROR`.

现在, 你也可以在你的 `build.gradle.kts` 文件中, 在 task 级进行配置:

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile>
("compileKotlin") {

jvmTargetValidationMode.set(org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidationMode.WARNING)
}
```

标准库

Kotlin 1.9.0 对标准库有了一些很大的改进:

- `..<` 操作符 和 时间 API 进入稳定版.
- Kotlin/Native 标准库经过了彻底的审查和更新
- `@Volatile` 注解可以在更多平台使用

- 有了一个 **共通的** 函数来通过名称获取正规表达式中捕获的组(capture group)
- 引入了 `HexFormat` 类, 用于 16 进制数的格式化和解析

用于终端开放(open-ended)的值范围的 ..< 操作符进入稳定版

新的 `..<` 操作符用于终端开放(open-ended)的值范围, 它在 Kotlin 1.7.20 (["..< 操作符的预览版, 用于创建终止端开放的值范围(open-ended range)" in "Kotlin 1.7.20 版中的新功能"]()) 中引入, 在 1.8.0 中进入稳定版. 在 1.9.0 中, 用于操作终端开放的值范围的标准库 API也进入了稳定版.

我们的研究显示, 在声明一个终端开放的值范围时, 新的 `..<` 操作符更加易于理解. 如果你使用 `until` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/until.html) 中缀函数, 很容易错误的理解为, 值范围包含它的上界(upper bound).

下面是使用 `until` 函数的示例:

```
fun main() {
    for (number in 2 until 10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 输出结果为 2 4 6 8
}
```

下面是使用新的 `..<` 操作符示例:

```
fun main() {
    for (number in 2..<10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 输出结果为 2 4 6 8
}
```

> 从 IntelliJ IDEA 2023.1.1 版开始, 有了一个新的代码审查, 对你可以使用 `..<` 操作符的地方, 会高亮显示.

关于如何使用这个操作符, 更多详情请参见 Kotlin 1.7.20 版中的新功能 ("..< 操作符的预览版, 用于创建终止端开放的值范围(open-ended range)" in "Kotlin 1.7.20 版中的新功能").

时间 API 进入稳定版

从 1.3.50 开始, 我们引入了一个新的时间测量 API 的预览版. API 中关于时间长度的部分在 1.6.0 中进入了稳定版. 在 1.9.0 中, 时间测量 API 的其他部分也进入了稳定版.

旧的时间 API 提供了 `measureTimeMillis` 和 `measureNanoTime` 函数, 使用起来不直观. 很明显, 这两个函数都测量时间, 使用不同的单位, 但很难清楚理解的是, `measureTimeMillis` 使用 wall clock (https://en.wikipedia.org/wiki/Elapsed_real_time) 来测量时间, 而 `measureNanoTime` 使用单调时间源(monotonic time source). 新的时间 API 解决了这个问题, 以及其他问题, 让 API 更加用户友好.

通过新的时间 API, 你可以很容易的实现以下功能:

- 使用单调时间源(monotonic time source), 测量执行某些代码消耗的时间, 使用你希望的时间单位.
- 标记一个时刻.
- 比较两个时刻, 并计算它们之间的差异.
- 检查从某个特定的时刻开始, 经过了多少时间.
- 检查当前时间是否已经经过了某个指定的时刻.

测量代码的执行时间

要测量执行一段代码消耗的时间, 请使用 `measureTime` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/measure-time.html) 内联函数.

要测量执行一段代码消耗的时间, **并且** 返回这段代码的执行结果, 请使用 `measureTimedValue` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/measure-timed-value.html) 内联函数.

默认情况下, 这两个函数使用一个单调时间源(monotonic time source). 但是, 如果你想要使用流逝的真实时间源(elapsed real-time source), 也是可以的. 例如, 在 Android 中, 默认的时间源 `System.nanoTime()` 在设备活动时才计算时间. 当设备进入深度睡眠时, 它会失去对时间的追踪. 想要在设备深度睡眠时继续追踪时间, 你可以改为创建一个使用 `SystemClock.elapsedRealtimeNanos()` (https://developer.android.com/reference/android/os/SystemClock#elapsedRealtimeNanos()) 的时间源:

```
object RealtimeMonotonicTimeSource :
AbstractLongTimeSource(DurationUnit.NANOSECONDS) {
    override fun read(): Long = SystemClock.elapsedRealtimeNanos()
}
```

标记时刻，并测量时刻之间的差异

要标记一个特定的时刻，请使用 `TimeSource` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-source/) 接口，和 `markNow()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-source/mark-now.html) 函数来创建一个 `TimeMark` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-mark/)。要测量来自同一个时间源的 `TimeMarks` 之间的差异，请使用减法操作符 (`-`):

```
import kotlin.time.*

fun main() {
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    Thread.sleep(500) // 睡眠 0.5 秒.
    val mark2 = timeSource.markNow()

    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        println("Measurement 1.${n + 1}: elapsed1=$elapsed1,
elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    // 也可以对时间标记进行比较.
    println(mark2 > mark1) // 比较结果为 true, 因为 mark2 是在 mark1 之
后捕获的.
}
```

要检查是否已经经过了某个截止时刻，或者是否已经到达超时时间，请使用 `hasPassedNow()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-mark/has-passed-now.html) 和 `hasNotPassedNow()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-mark/has-not-passed-now.html) 扩展函数:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    val fiveSeconds: Duration = 5.seconds
    val mark2 = mark1 + fiveSeconds

    // 还没有经过 5 秒
    println(mark2.hasPassedNow())
    // 输出结果为 false

    // 等待 6 秒
    Thread.sleep(6000)
    println(mark2.hasPassedNow())
    // 输出结果为 true
}
```

Kotlin/Native 标准库走向稳定

由于我们的 Kotlin/Native 标准库持续增长, 我们决定是时候对它进行一次全面的审查, 以确保它符合我们的高标准. 作为这次审查的一部分, 我们仔细的审查了 **每一个** 现有的 public 签名. 对每一个签名, 我们考虑它是否符合以下规则:

- 有一个单独的目的.
- 与其它 Kotlin API 一致.
- 与它在 JVM 版中的对应部分具有相似的行为.
- 面向未来.

基于这些考虑, 我们对每个签名进行了下面的某个决定:

- 让它进入稳定版.
- 让它进入实验版.
- 将它变为 `private`.

- 修改它的行为.
- 将它移动到其它地方.
- 废弃它.
- 将它标记为已过时.

> 如果一个现有的签名:
>
> - 移动到其它包, 那么这个签名会继续存在于原来的包中, 但它现在被废弃, 废弃级别为: `WARNING`. IntelliJ IDEA 会在代码审查后自动建议替换.
> - 被废弃, 那么它已被废弃, 废弃级别为: `WARNING`.
> - 被标记为已过时, 那么你可以继续使用它, 但将来它会被替换.

我们不会在这里列出这次审查的全部结果, 但下面是一些重要的部分:

- 我们让 Atomics API 进入了稳定版.
- 我们让 `kotlinx.cinterop` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlinx.cinterop/) 进入了实验版, 使用这个包, 现在会要求另一种使用者同意(Opt-in). 更多详情, 请参见 显式 C 互操作性的稳定性保证.
- 我们将 `Worker` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-worker/) 类和它的相关 API 标记为已过时.
- 我们将 `BitSet` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native/-bit-set/) 类标记为已过时.
- 我们将 `kotlin.native.internal` 包中的所有 `public` API 标记为 `private`, 或移动到了其它包.

显式 C 互操作性的稳定性保证

为了保护我们的 API 的高质量, 我们决定让 `kotlinx.cinterop` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlinx.cinterop/) 进入实验版. 尽管 `kotlinx.cinterop` 已经经过了彻底的试用和测试, 但在我们感到足够满意, 让它进入稳定版之前, 还存在改进的空间. 我们建议你使用这些 API 进行互操作, 但你应该将这些 API 的使用限制在你的项目中的特定部分. 当我们开始改进这个 API, 让它进入稳定版时, 这样可以让你的迁移工作更加容易.

如果你想要使用 C 风格的外部 API, 例如指针, 你必须使用 `@OptIn(ExperimentalForeignApi)` 标注使用者同意, 否则你的代码将不能编译.

要使用 `kotlinx.cinterop` 的其它部分, 包括 Objective-C/Swift 的互操作性, 你需要使用 `@OptIn(BetaInteropApi)` 标注使用者同意. 如果你使用这个 API 但没有标注使用者同意, 你的代码能够编译, 但编译器会提示警告, 对于你会遇到什么样的结果, 警告信息会提供一个清晰的解释.

关于这些注解, 更过详情请参见我们 `Annotations.kt` (https://github.com/JetBrains/kotlin/blob/56b729f1812733cb6a79673684c2fa5c4c6b3475/kotlin-native/Interop/Runtime/src/main/kotlin/kotlinx/cinterop/Annotations.kt) 的源代码.

关于这次审查带来的 **全部** 变更, 更多详情请参见我们的 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-55765).

我们欢迎你提供反馈意见! 你可以在这个 ticket (https://youtrack.jetbrains.com/issue/KT-57728) 中添加评论, 提供你的反馈意见.

@Volatile 注解进入稳定版

如果你使用 `@Volatile` 注解标注一个 `var` 属性, 那么它的后端域变量(Backing Field) 会被标注这个注解, 使得对这个域变量的所有读写操作都是原子化的, 而且写入操作永远对其它线程可见.

在 1.8.20 之前, `kotlin.jvm.Volatile` 注解 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-volatile/) 存在于在共通标准库中. 但是, 这个注解只对 JVM 有效. 如果你在其它平台上使用它, 它会被忽略, 因此导致错误.

在 1.8.20 中, 我们引入了一个实验性的共通注解, `kotlin.concurrent.Volatile`, 你可以在 JVM 和 Kotlin/Native 中试用.

在 1.9.0 中, `kotlin.concurrent.Volatile` 进入了稳定版. 如果你在你的跨平台项目中使用 `kotlin.jvm.Volatile`, 我们建议你迁移到 `kotlin.concurrent.Volatile`.

新的共通函数, 根据名称获取正规表达式中捕获的组

在 1.9.0 之前, 每个平台都有自己的扩展, 用于根据名称获取正规表达式中捕获的组. 但是, 没有共通的函数. 在 Kotlin 1.8.0 之前, 无法实现这样的共通函数, 因为标准库还支持 JVM 编译目标 1.6 和 1.7.

从 Kotlin 1.8.0 开始, 标准库使用 JVM 编译目标 1.8 来编译. 因此在 1.9.0 中, 现在有了 **共通的** `groups` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-match-result/groups.html) 函数, 你可以用来获取名称获取正规表达式中捕获的组的内容. 当你想要访问属于特定捕获组的正规表达式匹配结果时, 这会非常有用.

下面是一个示例, 使用正规表达式, 包含 3 个捕获组: `city`, `state`, 和 `areaCode`. 你可以使用这些组的名称来访问匹配的值:

```
fun main() {
    val regex = """\b(?<city>[A-Za-z\s]+),\s(?<state>[A-Z]{2}):\s(?
<areaCode>[0-9]{3})\b""".toRegex()
    val input = "Coordinates: Austin, TX: 123"

    val match = regex.find(input)!!
    println(match.groups["city"]?.value)
    // 输出结果为 Austin
    println(match.groups["state"]?.value)
    // 输出结果为 TX
    println(match.groups["areaCode"]?.value)
    // 输出结果为 123
}
```

新的路径工具函数, 用于创建父目录

在 1.9.0 中, 有一个新的 `createParentDirectories()` 扩展函数, 你可以用来创建一个新的文件, 如果需要, 还会创建所有的父目录. 如果你向 `createParentDirectories()` 指定一个文件路径, 它会检查父目录是否已经存在. 如果存在, 则不做处理. 但是, 如果父目录不存在, 它会为你创建这些父目录.

`createParentDirectories()` 在你复制文件时非常有用. 例如, 你可以结合 `copyToRecursively()` 函数来使用它:

```
sourcePath.copyToRecursively(
    destinationPath.createParentDirectories(),
    followLinks = false
 )
```

新的 HexFormat 类, 用于 16 进制数的格式化和解析

> ⚠ 新的 `HexFormat` 类以及相关的扩展函数是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"), 要使用它们, 你可以使用 `@OptIn(ExperimentalStdlibApi::class)` 注解标注使用者同意(Opt-in), 或者使用编译器参数 `-opt-in=kotlin.ExperimentalStdlibApi`.

在 1.9.0 中, `HexFormat` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-hex-format/) 类以及相关的扩展函数作为实验性的功能提供, 允许你对数字和 16 进制字符串进行转换. 具体来说, 你可以使用扩展函数对 16 进制字符串和 `ByteArrays` 或其他数字类型 (`Int`, `Short`, `Long`) 进行转换.

例如:

```
println(93.toHexString()) // 输出结果为 "0000005d"
```

`HexFormat` 类包含格式化选项, 你可以使用 `HexFormat{}` 构建器进行配置.

如果你在使用 `ByteArrays`, 你可以通过属性配置以下选项:

选项	描述
`upperCase`	16 进制数字是大写还是小写. 默认情况下, 使用小写. `upperCase = false`.
`bytes.bytesPerLine`	每行最大字节数.
`bytes.bytesPerGroup`	每组最大字节数.
`bytes.bytesSeparator`	字节之间的分隔符. 默认没有分隔符.
`bytes.bytesPrefix`	前缀字符串, 紧接在每个字节的 2 字符 16 进制表达之前, 默认没有前缀字符串.
`bytes.bytesSuffix`	后缀字符串, 紧接在每个字节的 2 字符 16 进制表达之后, 默认没有后缀字符串.

示例:

```
val macAddress = "001b638445e6".hexToByteArray()

// 使用 HexFormat{} 构建器, 在 16 进制字符串之间使用冒号分隔
println(macAddress.toHexString(HexFormat { bytes.byteSeparator = ":" }))
// 输出结果为 "00:1b:63:84:45:e6"

// 使用 HexFormat{} 构建器进行配置:
```

```
// * 对 16 进制字符串使用大写字符
// * 每 2 个字节分为 1 组
// * 使用点号分隔
val threeGroupFormat = HexFormat { upperCase = true;
bytes.bytesPerGroup = 2; bytes.groupSeparator = "." }

println(macAddress.toHexString(threeGroupFormat))
// 输出结果为 "001B.6384.45E6"
```

如果你在使用数字类型, 你可以通过属性配置以下选项:

选项	描述
`number.prefix`	16 进制字符串的前缀, 默认没有前缀.
`number.suffix`	16 进制字符串的后缀, 默认没有后缀.
`number.removeLeadingZeros`	是否删除 16 进制字符串中的前导 0. 默认不删除前导 0. `number.removeLeadingZeros = false`

示例:

```
// 使用 HexFormat{} 构建器, 解析 16 进制字符串, 前缀为: "0x".
println("0x3a".hexToInt(HexFormat { number.prefix = "0x" })) // 输出
结果为 "58"
```

文档更新

Kotlin 文档有了一些重要变更:

- Kotlin 观光之旅 (欢迎参加我们的 Kotlin 观光之旅!) – 通过理论和实践章节, 学习 Kotlin 编程语言的基础知识.
- Android 源代码集布局 (Android 源代码集布局) – 了解新的 Android 源代码集布局.
- Kotlin Multiplatform 兼容性指南 (Kotlin Multiplatform 兼容性指南) – 了解使用 Kotlin Multiplatform 开发项目时你可能遇到的不兼容的变更.

* Kotlin Wasm (使用 Kotlin 进行 Wasm 开发) – 了解 Kotlin/Wasm, 以及在你的 Kotlin Multiplatform 项目中如何使用它.
* 向 Kotlin/Wasm 项目添加 Kotlin 库的依赖项 (向 Kotlin/Wasm 项目添加 Kotlin 库依赖项) – 了解 Kotlin/Wasm 支持的 Kotlin 库.

安装 Kotlin 1.9.0

检查 IDE 版本

IntelliJ IDEA (https://www.jetbrains.com/idea/download/) 2022.3.3 和 2023.1.1 会自动建议将 Kotlin plugin 更新到 1.9.0 版本. IntelliJ IDEA 2023.2 会包含 Kotlin 1.9.0 plugin.

Android Studio Giraffe (223) 和 Hedgehog (231) 会在后续的发布版中支持 Kotlin 1.9.0.

新的命令行编译器可以通过 GitHub 发布页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.0) 下载.

配置 Gradle 的设置

要下载 Kotlin 的 artifact 和依赖项, 请更新你的 `settings.gradle(.kts)` 文件, 使用 Maven Central 仓库:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

如果没有指定仓库, Gradle 会使用已废弃的 JCenter 仓库, 导致无法下载 Kotlin artifact 的错误.

Kotlin 1.9.0 兼容性指南

Kotlin 1.9.0 是一个 功能发布版 ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化"), 因此其中的变更可能不兼容你之前针对旧版本 Kotlin 编写的代码. 关于这样的变更, 详情请参见 Kotlin 1.9.0 兼容性指南 (Kotlin 1.9 兼容性指南).

Kotlin 2.0.0-Beta5 版中的新功能

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/whatsnew-eap.html)

Kotlin 1.8.20 版中的新功能

最终更新: 2024/09/10

发布日期: 2023/04/25 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.8.20 已经发布了, 其中一些重要更新如下:

- 新的 Kotlin K2 编译器的更新
- 新的 Kotlin/Wasm 编译目标(实验性功能)
- Gradle 中默认启用新的 JVM 增量编译
- Kotlin/Native 编译目标的更新
- Kotlin Multiplatform 中的 Gradle 复合构建(composite build) (预览版)
- Xcode 中 Gradle 错误信息的改进
- 标准库支持 AutoCloseable 接口 (实验性功能)
- 标准库支持 Base64 编码(实验性功能)

关于本次更新的概要介绍, 你可以观看以下视频:

IDE 支持

在以下 IDE 中可以使用支持 1.8.20 版的 Kotlin plugin:

IDE	支持的版本
IntelliJ IDEA	2022.2.x, 2022.3.x, 2023.1.x
Android Studio	Flamingo (222)

> ⚠ 要正确下载 Kotlin 的 artifact 和依赖项, 请 配置你的 Gradle 设置 使用 Maven Central 仓库.

新的 Kotlin K2 编译器的更新

Kotlin 开发组一直在努力稳定 K2 编译器. 在 Kotlin 1.7.0 版发布公告 ("JVM 平台的新的 Kotlin K2 编译器 (Alpha 版)" in "Kotlin 1.7.0 版中的新功能") 中曾经提到, 它现在还处于 **Alpha 版**. 为了向 K2 Beta 版 (https://youtrack.jetbrains.com/issue/KT-52604) 推进, 本次发布引入了更多的改进.

从本次 1.8.20 发布版开始, Kotlin K2 编译器:

- 有了一个序列化 plugin (预览版).
- 对 JS IR 编译器 (使用 IR 编译器) 提供 Alpha 支持.
- 介绍未来版本: 新的语言版本, Kotlin 2.0 (https://blog.jetbrains.com/kotlin/2023/02/k2-kotlin-2-0/).

关于新编译器和它的益处, 更多详情请观看以下视频:

- 关于新 Kotlin K2 编译器, 每个人都应该了解的知识 (https://www.youtube.com/watch?v=iTdJJq_LyoY)
- 新 Kotlin K2 编译器: 专家评审 (https://www.youtube.com/watch?v=db19VFLZqJM)

如何启用 Kotlin K2 编译器

要启用并测试 Kotlin K2 编译器, 请通过下面的编译器选项, 使用新的语言版本:

```
-language-version 2.0
```

你可以在你的 `build.gradle(.kts)` 文件中指定这个选项:

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = "2.0"
        }
    }
}
```

以前的 `-Xuse-k2` 编译器选项已被废弃.

> 新 K2 编译器的 Alpha 版只能用于 JVM 和 JS IR 项目. 它还不支持 Kotlin/Native, 也不支持任何 跨平台项目.

留下你对于新 K2 编译器的反馈意见

如果你能提供你的反馈意见, 我们将会非常感谢!

- 在 Kotlin Slack 频道中, 直接向 K2 开发者提供你的反馈意见 – 获得邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up?_gl=1*ju6cbn*_ga*MTA3MTk5NDkzMC4xNjQ2MDY3MDU4*_ga_9J976DJZ68*MTY1ODMzNzA3OS4xMDAuMS4xNjU4MzQwODEwLjYw) 并加入 #k2-early-adopters (https://kotlinlang.slack.com/archives/C03PK0PE257) 频道.
- 在 我们的问题追踪系统 (https://kotl.in/issue) 中报告你遇到的新 K2 编译器的问题.
- 启用 **Send usage statistics** 选项 (https://www.jetbrains.com/help/idea/settings-usage-statistics.html), 允许 JetBrains 收集关于 K2 使用状况的匿名数据.

语言

随着 Kotlin 的不断演化, 我们在 1.8.20 中引入了新的语言功能的预览版:

- 枚举类值函数的现代而且高性能的替代者
- 与数据类(Data Class)对称的数据对象(Data Object)
- 解除对内联类(Inline class)中有 body 的次级构造器(secondary constructor)的限制

枚举类值函数的现代而且高性能的替代者

> 这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

枚举类有一个合成(synthetic)函数 `values()`, 它返回一个数组, 其中包含枚举类中定义的枚举常数. 但是, 使用数组可能导致 Kotlin 和 Java 中的 隐含的性能问题 (https://github.com/Kotlin/KEEP/blob/master/proposals/enum-entries.md#examples-of-performance-issues). 此外, 大多数 API 都使用集合, 因此最终还是需要转换. 为了解决这些问题,

我们为枚举类引入了 entries 属性, 用来替代 values() 函数. 调用时, entries 属性返回一个预先分配的可不变 List, 其中包含枚举类中定义的枚举常数.

> ⚠ values() 函数仍然继续支持, 但我们推荐你改为使用 entries 属性.

```
enum class Color(val colorName: String, val rgb: String) {
    RED("Red", "#FF0000"),
    ORANGE("Orange", "#FF7F00"),
    YELLOW("Yellow", "#FFFF00")
}

@OptIn(ExperimentalStdlibApi::class)
fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb ==
rgb }
```

如何启用 entries 属性

要试用这个功能, 请使用 @OptIn(ExperimentalStdlibApi) 注解标注使用者同意(Opt-in), 并启用 -language-version 1.9 编译器选项. 在 Gradle 项目中, 可以在你的 build.gradle(.kts) 文件中添加以下代码:

Kotlin

```
tasks

.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks

.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k.class)
    .configureEach {
        compilerOptions.languageVersion =

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

从 IntelliJ IDEA 2023.1 开始, 如果你对这个功能标注了使用者同意(Opt-in), IDE 的代码检查功能会通知你将 `values()` 转换为 `entries`, 并为你提供快速修正.

关于这个提案, 更多详情请参见 KEEP 条目 (https://github.com/Kotlin/KEEP/blob/master/proposals/enum-entries.md).

与数据类(Data Class)对称的数据对象(Data Object) (预览版)

数据对象(Data Object) 允许你声明 singleton 语义的对象, 并带有一个干净的 `toString()` 表达. 在下面的代码片段中, 你可以看到向一个对象声明添加 `data` 关键字, 如何改善它的 `toString()` 输出的可读性:

```
package org.example
object MyObject
data object MyDataObject

fun main() {
    println(MyObject) // 输出结果为 org.example.MyObject@1f32e575
    println(MyDataObject) // 输出结果为 MyDataObject
}
```

特别是对于 `sealed` 类型层级结构(例如 `sealed class` 或 `sealed interface` 类型层级结构), 非常适合使用 `data objects`, 因为可以与 `data class` 声明一起方便的使用. 在下面的代码片段中, 将 `EndOfFile` 声明为 `data object` 而不是普通的 `object`, 代表它自动拥有漂亮的 `toString`, 不需要手动的覆盖这个函数. 这样就保持了与相应的数据类定义的对称性.

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // 输出结果为 Number(number=7)
    println(EndOfFile) // 输出结果为 EndOfFile
}
```

数据对象的语义

从 Kotlin 1.7.20 ("对单子(Singleton)与带 data object 的封闭类层级结构(Sealed Class Hierarchy), 改善了它们的字符串表示" in "Kotlin 1.7.20 版中的新功能") 中的第一个预览版之后, 数据对象的语义有了一些改进. 编译器现在会自动为它们生成一些便利的函数:

toString

数据对象的 `toString()` 函数返回对象的简单名称:

```
data object MyDataObject {
    val x: Int = 3
}

fun main() {
    println(MyDataObject) // 输出结果为 MyDataObject
}
```

equals 和 hashCode

`data object` 的 `equals()` 函数会保证你的 `data object` 的所有对象都被看作相等. 大多数情况下, 你的数据对象在运行期只会存在单个实例(毕竟, `data object` 声明的就是一个单子(singleton)). 但是, 在某些特殊情况下, 也可以在运行期生成相同类型的其他对象 (例如, 通过 `java.lang.reflect` 使用平台的反射功能, 或通过底层使用了这个 API 的 JVM 序列化库), 这个功能可以确保这些对象被当作相等.

请确保只对 `data objects` 进行结构化的相等比较 (使用 `==` 操作符), 而不要进行引用相等比较 (使用 `===` 操作符). 如果数据对象在运行期有一个以上的实例存在, 这样可以帮助你避免错误. 下面的代

码片段演示这种特殊情况:

```
import java.lang.reflect.Constructor

data object MySingleton

fun main() {
    val evilTwin = createInstanceViaReflection()

    println(MySingleton) // 输出结果为 MySingleton
    println(evilTwin) // 输出结果为 MySingleton

    // 即使一个库强行创建了 MySingleton 的第二个实例, 它的 `equals` 方法也
会返回 true:
    println(MySingleton == evilTwin) // 输出结果为 true

    // 不要使用 === 比较数据对象.
    println(MySingleton === evilTwin) // 输出结果为 false
}

fun createInstanceViaReflection(): MySingleton {
    // Kotlin 的反射功能不允许创建数据对象的实例.
    // 这段代码 "强行" 创建新的 MySingleton 实例 (也就是通过 Java 平台的反
射功能)
    // 在你的代码中一定不要这样做!
    return (MySingleton.javaClass.declaredConstructors[0].apply {
isAccessible = true } as Constructor<MySingleton>).newInstance()
}
```

编译器生成的 `hashCode()` 函数的行为与 `equals()` 函数保持一致, 因此一个 `data object` 的所有运行期实例都拥有相同的 hash 值.

数据对象没有 copy 和 componentN 函数

尽管 `data object` 和 `data class` 声明经常一起使用, 而且很相似, 但对于 `data object` 有一些函数没有生成:

因为 `data object` 声明通常用作单子对象, 因此不会生成 `copy()` 函数.

这种单子模式将一个类限定为只有单个实例, 如果允许创建实例的拷贝, 就破坏了只存在单个实例的原则.

而且, 与 `data class` 不同, `data object` 没有任何数据属性. 对这种没有数据属性的对象进行解构是没有意义的, 因此不会生成 `componentN()` 函数.

关于这个功能, 希望你能通过 YouTrack (https://youtrack.jetbrains.com/issue/KT-4107) 提供你的反馈意见.

如何启用数据对象的预览版

要试用这个功能, 请启用 `-language-version 1.9` 编译器选项. 在 Gradle 项目中, 可以在你的 `build.gradle(.kts)` 文件中添加以下代码:

Kotlin

```
tasks

.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks

.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k.class)
    .configureEach {
        compilerOptions.languageVersion =
```

```
org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

解除对内联类(Inline class)中有 body 的次级构造器(secondary constructor)的限制 (预览版)

> ⚠ 这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

Kotlin 1.8.20 解除了在 内联类(Inline class) (内联的值类(Inline value class)) 中使用有 body 的次级构造器(secondary constructor)的限制.

内联类过去只允许 public 的主构造器, 不允许使用 `init` 代码块或次级构造器, 以便保证初始化代码的语义清晰. 这就造成, 无法封装底层值, 或创建一个内联类来表达某些受限定的值.

这些问题现在已经解决了. Kotlin 1.4.30 取消了对 `init` 代码块的限制. 现在我们更进一步, 允许有 body 的次级构造器 (预览版):

```
@JvmInline
value class Person(private val fullName: String) {
    // 从 Kotlin 1.4.30 开始可以使用:
    init {
        check(fullName.isNotBlank()) {
            "Full name shouldn't be empty"
        }
    }
    // 从 Kotlin 1.8.20 开始可以使用 (预览版):
    constructor(name: String, lastName: String) : this("$name
$lastName") {
        check(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }
}
```

如何启用有 body 的次级构造器

要试用这个功能, 请启用 `-language-version 1.9` 编译器选项. 在 Gradle 项目中, 可以在你的 `build.gradle(.kts)` 文件中添加以下代码:

Kotlin

```
tasks

.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks

.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k.class)
    .configureEach {
        compilerOptions.languageVersion =

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

我们鼓励你试用这个功能, 并在 YouTrack (https://kotl.in/issue) 中报告问题, 帮助我们让这个功能在 Kotlin 1.9.0 中默认启用.

关于 Kotlin 内联类的进展, 请参见 这个 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/inline-classes.md).

新的 Kotlin/Wasm 编译目标

Kotlin/Wasm (Kotlin WebAssembly) 在本次发布中进入了 实验阶段 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). Kotlin 开发组认为 WebAssembly (https://webassembly.org/) 是一项很有前途的技术, 并希望找到更好的方式, 让你使用它, 同时又得到 Kotlin 的一切益处.

Wasm 为 Kotlin 和其他编程语言提供了在 Web 上运行的编译目标. WebAssembly 二进制格式是平台独立的, 因为它运行在自己的虚拟机上. 几乎所有的现代浏览器都已经支持 WebAssembly 1.0. 要设置环境来运行 WebAssembly, 你只需要启用 Kotlin/Wasm 编译目标的一个实验性的垃圾收集模式. 具体做法请参见: 如何启用 Kotlin/Wasm.

我们想要重点介绍新的 Kotlin/Wasm 编译目标的以下优势:

- 与 `wasm32` Kotlin/Native 编译目标相比, 编译速度更快, 因为 Kotlin/Wasm 不必使用 LLVM.
- 与 `wasm32` 编译目标相比, 与 JS 的互操性以及与浏览器的集成都更加容易, 这是因为使用了 Wasm 垃圾收集器 (https://github.com/WebAssembly/gc).
- 与 Kotlin/JS 和 JavaScript 相比, 应用程序启动速度可能更快, 因为 Wasm 的字节码更小, 并且易于解析.
- 与 Kotlin/JS 和 JavaScript 相比, 应用程序的运行期性能更好, 因为 Wasm 是一种静态类型语言.

从 1.8.20 版开始, 你可以在你的实验性项目中使用 Kotlin/Wasm. 我们为 Kotlin/Wasm 提供了开箱即用的 Kotlin 标准库(`stdlib`) 和测试库(`kotlin.test`). IDE 支持会在未来的发布版中添加.

观看这个 YouTube 视频, 了解关于 Kotlin/Wasm 的更多信息 (https://www.youtube.com/watch?v=-pqz9sKXatw).

如何启用 Kotlin/Wasm

要启用并测试 Kotlin/Wasm, 请更新你的 `build.gradle.kts` 文件:

```
plugins {
    kotlin("multiplatform") version "1.8.20"
}

kotlin {
    wasm {
        binaries.executable()
        browser {
        }
```

```
    }
    sourceSets {
        val commonMain by getting
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
        val wasmMain by getting
        val wasmTest by getting
    }
}
```

> ⚠ 请查看 Kotlin/Wasm 示例程序的 GitHub 代码仓库 (https://github.com/Kotlin/kotlin-wasm-examples).

要运行 Kotlin/Wasm 项目, 你需要更新目标环境的设定:

Chrome

- 对 109 版本:

使用 `--js-flags=--experimental-wasm-gc` 命令行参数运行应用程序.

- 对 110 或以上版本:

  1. 在你的浏览器中进入 `chrome://flags/#enable-webassembly-garbage-collection`.
  2. 启用 **WebAssembly Garbage Collection**.
  3. 重新启动你的浏览器.

Firefox

对 109 或以上版本:

1. 在你的浏览器中进入 `about:config`.

2. 启用 `javascript.options.wasm_function_references` and `javascript.options.wasm_gc` 选项.

3. 重新启动你的浏览器.

Edge

对 109 或以上版本:
使用 `--js-flags=--experimental-wasm-gc` 命令行参数运行应用程序.

留下你对于 Kotlin/Wasm 的反馈意见

如果你能提供你的反馈意见, 我们将会非常感谢!

* 在 Kotlin Slack 频道中, 直接向开发者提供你的反馈意见 – 获得邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up?_gl=1*ju6cbn*_ga*MTA3MTk5NDkzMC4xNjQ2MDY3MDU4*_ga_9J976DJZ68*MTY1ODMzNzA3OS4xMDAuMS4xNjU4MzQwODEwLjYw), 并加入 #webassembly (https://kotlinlang.slack.com/archives/CDFP59223) 频道.
* 在 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-56492) 中, 报告你遇到的 Kotlin/Wasm 的问题.

Kotlin/JVM

Kotlin 1.8.20 引入了 Java 合成属性(synthetic property)的引用 (预览版) 和 在 kapt stub 生成任务中默认支持 JVM IR 后端.

Java 合成属性(synthetic property)的引用 (预览版)

> 这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

Kotlin 1.8.20 引入了新的功能, 可以创建 Java 合成属性(synthetic property) 引用, 例如, 对这段 Java 代码:

```
public class Person {
    private String name;
```

```
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Kotlin 允许你使用 `person.age`, 其中 `age` 是一个合成属性. 现在, 你还可以创建 `Person::age` 和 `person::age` 的引用. 对 `name` 也是一样.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12),
Person("Peter", 11))
    Persons
        // 调用 Java 合成属性的引用:
        .sortedBy(Person::age)
        // 通过 Kotlin 的属性语法, 调用 Java 取值方法:
        .forEach { person -> println(person.name) }
```

如何启用 Java 合成属性的引用

要试用这个功能, 请启用 `-language-version 1.9` 编译器选项. 在 Gradle 项目中, 你可以对你的 `build.gradle(.kts)` 文件添加以下内容:

Kotlin

```
tasks

.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k<*>>()
    .configureEach {
```

```
        compilerOptions
            .languageVersion
            .set(

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks

.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k.class)
    .configureEach {
        compilerOptions.languageVersion =

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
}
```

在 kapt stub 生成任务中默认支持 JVM IR 后端

在 Kotlin 1.7.20 中, 我们引入了 在 kapt stub 生成任务中支持 JVM IR 后端 ("在 kapt stub 生成 task 中支持 JVM IR 后端" in "Kotlin 1.7.20 版中的新功能") 功能. 从这个发布版开始, 默认启用这个支持. 你不再需要在你的 `gradle.properties` 中指定 `kapt.use.jvm.ir=true` 来启用这个功能. 关于这个功能, 希望你能通过 YouTrack (https://youtrack.jetbrains.com/issue/KT-49682) 提供你的反馈意见.

Kotlin/Native

Kotlin 1.8.20 包含的变更有: Kotlin/Native 支持的目标平台, 与 Objective-C 互操作性, CocoaPods Gradle plugin 的改进, 以及其他更新:

- 对 Kotlin/Native 目标平台的更新
- 废弃了旧的内存管理器
- 支持带 @import 指令的 Objective-C 头文件

- 支持 Cocoapods Gradle plugin 中的 link-only 模式
- 在 UIKit 中将 Objective-C 扩展导入为类的成员
- 在编译器中重新实现了编译器的缓存管理
- 在 Cocoapods Gradle plugin 中废弃了 `useLibraries()`

Kotlin/Native 目标平台的更新

Kotlin 开发组决定重新审查 Kotlin/Native 支持的目标平台, 将它们分为不同的支持层级, 并从 Kotlin 1.8.20 开始废弃其中的一部分. 关于支持的和废弃的目标平台的完整列表, 请参见 Kotlin/Native 支持的目标平台 (Kotlin/Native 支持的目标平台).

从 Kotlin 1.8.20 开始, 以下目标平台已被废弃, 将在 1.9.20 中删除:

- `iosArm32`
- `watchosX86`
- `wasm32`
- `mingwX86`
- `linuxArm32Hfp`
- `linuxMips32`
- `linuxMipsel32`

对于剩下的目标平台, 根据 Kotlin/Native 编译器中支持和测试程度的不同, 现在分为 3 个支持层级. 一个目标平台可能被移动到不同的层级. 例如, 将来我们会尽最大努力对 `iosArm64` 提供完全的支持, 因为它对 Kotlin Multiplatform (Kotlin 跨平台程序开发入门) 非常重要.

如果你是库的作者, 这 3 个支持层级能够帮助你决定在 CI 工具中测试哪些目标平台, 略过哪些目标平台. Kotlin 开发组在 Kotlin 官方库的开发中也使用这个方案, 例如 kotlinx.coroutines (协程指南).

关于这些变更的原因, 详情请阅读我们的 blog (https://blog.jetbrains.com/kotlin/2023/02/update-regarding-kotlin-native-targets/).

废弃了旧的内存管理器

从 1.8.20 开始, 旧的内存管理器已被废弃, 并将在 1.9.20 中删除. 新的内存管理器 (Kotlin/Native 内存管理) 已在 1.7.20 中默认启用, 之后还进行了一些稳定性更新和性能改进.

如果你还在使用旧的内存管理器, 请从你的 `gradle.properties` 文件删除 `kotlin.native.binary.memoryModel=strict` 选项, 并遵循我们的 迁移指南 (迁移到新的内存管理器) 进行必要的变更.

新的内存管理器不支持 `wasm32` 目标平台. 这个目标平台 从这个发布版开始已被废弃, 并将在 1.9.20 中删除.

支持带 @import 指令的 Objective-C 头文件

这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

Kotlin/Native 现在可以导入带 `@import` 指令的 Objective-C 头文件. 在使用具有自动生成的 Objective-C 头文件的 Swift 库, 或使用 Swift 编写的 CocoaPods 依赖项的类时, 这个功能非常有用.

在以前的版本中, cinterop 工具无法通过 `@import` 指令分析依赖于 Objective-C 模块的头文件. 因为它缺乏对 `-fmodules` 选项的支持.

从 Kotlin 1.8.20 开始, 你可以使用带 `@import` 的 Objective-C 头文件. 为了使用这个功能, 请在定义文件中通过 `compilerOpts` 向编译器传递 `-fmodules` 选项. 如果你使用 CocoaPods 集成 (CocoaPods 概述与设置), 请在 `pod()` 函数的在配置代码块中指定 cinterop 选项, 如下:

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("PodName") {
            extraOpts = listOf("-compiler-option", "-fmodules")
        }
    }
}
```

这是一个 期待已久的功能 (https://youtrack.jetbrains.com/issue/KT-39120), 我们欢迎你在 YouTrack (https://kotl.in/issue) 中提供你的反馈意见, 帮助我们在未来的发布版中将它变成默认功能.

支持 Cocoapods Gradle plugin 中的 link-only 模式

从 Kotlin 1.8.20 开始, 你可以将 Pod 依赖项和动态框架(dynamic framework)一起使用, 只用于链接, 而不生成 cinterop 绑定. 对于 cinterop 绑定已经生成的情况, 这个功能可能会有用.

考虑一个项目, 有 2 个模块, 1 个是库, 1 个是应用程序. 库依赖于一个 Pod, 但不产生框架, 只产生 1 个 `.klib`. 应用程序依赖于库, 并产生一个动态框架(dynamic framework). 对于这样的情况, 你需要使用使用库依赖的 Pod 来链接这个框架, 但你不需要 cinterop 绑定, 因为已经为库生成了绑定.

要启用这个功能, 请在添加 Pod 依赖项时使用 `linkOnly` 选项, 或构建器属性:

```
cocoapods {
    summary = "CocoaPods test library"
    homepage = "https://github.com/JetBrains/kotlin"

    pod("Alamofire", linkOnly = true) {
        version = "5.7.0"
    }
}
```

> 如果你对静态框架使用这个选项, 它会删除整个 Pod 依赖项, 因为对静态框架的链接不会使用 Pod.

在 UIKit 中将 Objective-C 扩展导入为类的成员

从 Xcode 14.1 开始, 来自 Objective-C 类的一些方法已经被移动为类别成员(category member). 这会导致生成不同的 Kotlin API, 而且这些方法会被导入为 Kotlin 扩展, 而不是方法.

在使用 UIKit 并覆盖方法时, 你可能已经遇到了这个变更造成的问题. 例如, 在 Kotlin 中继承 UIView 类时, 将会无法覆盖 `drawRect()` 或 `layoutSubviews()` 方法.

从 1.8.20 开始, 在与 NSView 和 UIView 类相同的头文件中声明的类别成员(category member), 会被导入为这些类的成员. 因此, 从 NSView 和 UIView 继承的子类, 可以很容易的覆盖这些方法, 就像其它方法一样.

如果一切顺利, 我们计划对所有的 Objective-C 类默认启用这个行为.

在编译器中重新实现了编译器的缓存管理

为了加快编译器缓存功能的演进速度, 我们将编译器缓存管理从 Kotlin Gradle plugin 移动到了 Kotlin/Native 编译器中. 这样做就使得我们可以进行几项重要的改进工作, 包括编译速度和编译器缓存灵活性相关的改进.

如果你遇到问题, 需要回到原来的行为, 请使用 Gradle 属性 `kotlin.native.cacheOrchestration=gradle`.

希望你能通过 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

在 Cocoapods Gradle plugin 中废弃了 useLibraries()

Kotlin 1.8.20 开始了 `useLibraries()` 函数的废弃周期, 这个函数用于静态库的 CocoaPods 集成 (CocoaPods 概述与设置).

我们过去引入 `useLibraries()` 函数, 是为了允许使用包含静态库的 Pod 依赖项. 随着时间的推移, 这样的情况变得非常罕见. 大多数 Pod 都使用源代码来发布, 而且二进制的发布通常会选择 Objective-C 框架或 XCFramework.

由于不再需要使用这个函数, 而且它会导致一些问题, 使得 Kotlin CocoaPods Gradle plugin 的开发变得复杂, 我们决定废弃它.

关于框架和 XCFramework, 更多详情请参见 构建最终的原生二进制文件 (构建最终的原生二进制文件).

Kotlin Multiplatform

Kotlin 1.8.20 致力于改善开发者体验, 对 Kotlin Multiplatform 进行了以下更新:

- 设置源代码集层级结构的新方案
- Kotlin Multiplatform 支持 Gradle 复合构建(composite build) (预览版)
- Xcode 中 Gradle 错误信息的改进

源代码集层级结构的新方案

源代码集层级结构的新方案是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 在未来的 Kotlin 发布版中, 它随时有可能变更, 不会预先通知. 需要使用者同意(Opt-in) (详情见下文). 希望你能通过 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

Kotlin 1.8.20 提供了一种新的方式, 在你的跨平台项目中设置源代码集层级结构 - 默认的编译目标层级结构. 新方案旨在替代编译目标的简写(shortcut), 例如 `ios`, 这些编译目标简写(shortcut)存在设计缺陷.

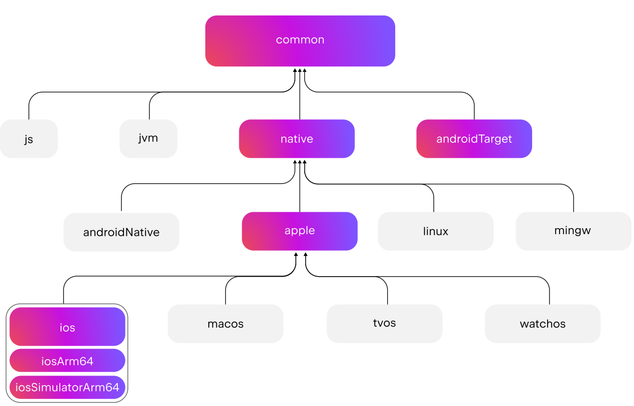
默认的编译目标层级结构背后的理念非常简单: 你要明确声明你的项目所有编译目标, Kotlin Gradle plugin 会根据指定的编译目标自动创建共用的源代码集.

设置你的项目

以下面这个简单的跨平台移动应用程序为例子:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
kotlin {
    // 启用默认的编译目标层级结构:
    targetHierarchy.default()

    android()
    iosArm64()
    iosSimulatorArm64()
}
```

你可以将默认的编译目标层级结构看作一个模板, 其中包含所有可能的编译目标以及它们的共用源代码集. 当你在你的代码中声明最终的编译目标 `android`, `iosArm64`, 和 `iosSimulatorArm64` 时, Kotlin Gradle plugin 会从模板中找到合适的共用源代码集, 并为你创建这些共用源代码集. 最终产生的层级结构如下:



绿色的源代码集会自动创建并包含到项目中, 同时, 默认模板中的灰色的源代码集会被忽略. 你可以看到, Kotlin Gradle plugin 没有创建一些源代码集, 例如 `watchos`, 因为项目中没有 watchOS 编译目标.

如果你添加一个 watchOS 编译目标, 例如 `watchosArm64`, `watchos` 源代码集就会被创建, 来自

apple, native, 和 common 源代码集的代码也会被编译到 watchosArm64.

关于默认的编译目标层级结构的完整构成, 请参见 文档 ("默认层级结构模板" in "层级项目结构").

> 在这个示例中, apple 和 native 源代码集只会对 iosArm64 和 iosSimulatorArm64 编译目标编译. 因此, 尽管它们的名字不是 ios, 它们可以访问完整的 iOS API. 对于 native 这样的源代码集, 这可能会违反直觉, 因为你可能会期望在这个源代码集中, 只能访问那些所有原生编译目标都能够使用的 API. 这个行为未来可能会变更.

为什么要替换简写(shortcut)

创建源代码集层级结构, 可能繁琐, 易出错, 而且对初学者不友好. 我们之前的解决方案是, 引入 ios 这样的简写(shortcut), 它会为你创建层级结构的一部分. 但是, 使用简写已被证明存在很大的设计缺陷: 它们很难变更.

以 ios 简写为例子. 它只创建 iosArm64 和 iosX64 编译目标, 这可能令人困惑, 而且如果使用基于 M1 的主机, 还需要 iosSimulatorArm64 编译目标, 就会导致错误. 但是, 添加 iosSimulatorArm64 编译目标, 对于用户项目来说可能是一个引起混乱的变更:

- 在 iosMain 源代码集中使用的所有依赖项必须支持 iosSimulatorArm64 编译目标; 否则, 依赖项解析会失败.
- 在添加新的编译目标时 (尽管对于 iosSimulatorArm64 的情况, 这不太可能), iosMain 中使用的一些原生 API 可能会消失.
- 某些情况下, 例如, 在你的基于 Intel 的 MacBook 上编写一个小的玩具项目的时候, 你可能根本不需要这个变更.

很明显, 简写并不能解决层级结构配置的问题, 所以我们在某个时候停止添加新的简写.

初看起来, 默认的编译目标层级结构可能与简写很类似, 但它们有一个关键的区别: **用户必须明确指定编译目标集**. 这个编译目标集定义你的项目如何编译, 如何发布, 如何参与依赖项解析. 由于这个编译目标集是固定的, Kotlin Gradle plugin 对默认配置的变更, 对于生态系统造成的影响应该会显著减少, 并且提供工具辅助的迁移将会更加容易.

如何启用默认的层级结构

这个新功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 对于 Kotlin Gradle 构建脚本, 你需要使用 @OptIn(ExperimentalKotlinGradlePluginApi::class) 标注使用者同意(Opt-in).

更多详情请参见 层级项目结构 ("默认层级结构模板" in "层级项目结构").

留下你的反馈意见

这是跨平台项目的重大变更. 希望你能提供你的 反馈意见 (https://kotl.in/issue), 帮助然它变得更好.

Kotlin Multiplatform 中支持 Gradle 复合构建(composite build) (预览版)

> 从 Kotlin Gradle Plugin 1.8.20 开始, 在 Gradle 构建中支持这个功能. 对于 IDE 支持, 请使用 IntelliJ IDEA 2023.1 Beta 2 (231.8109.2) 或更高版本, 以及 Kotlin Gradle plugin 1.8.20, 与任何版本的 Kotlin IDE plugin 一起使用.

从 1.8.20 开始, Kotlin Multiplatform 支持 Gradle 复合构建(composite build) (https://docs.gradle.org/current/userguide/composite_builds.html). 复合构建允许你将其他项目的构建, 或同一项目的其它部分的构建, 包含到单个构建中.

由于一些技术困难, 对 Kotlin Multiplatform 使用 Gradle 符合构建还只有部分的支持. Kotlin 1.8.20 包含了对复合构建支持的改进(预览版), 应该能够适用于更多种类的项目. 要试用这个功能, 请向你的 `gradle.properties` 添加以下选项:

```
kotlin.mpp.import.enableKgpDependencyResolution=true
```

这个选项会启用新的导入模式的预览版. 除了支持复合构建, 它还提供了跨平台项目中更流畅的导入体验, 因为我们包含了一些重大的 Bug 修复和改进, 使得导入功能更加稳定.

已知的问题

这个功能仍然是预览版, 需要继续改进稳定性, 在此过程中你可能遇到一些与导入相关的问题. 下面是一些已知的问题, 我们计划在 Kotlin 1.8.20 最终发布之前修复:

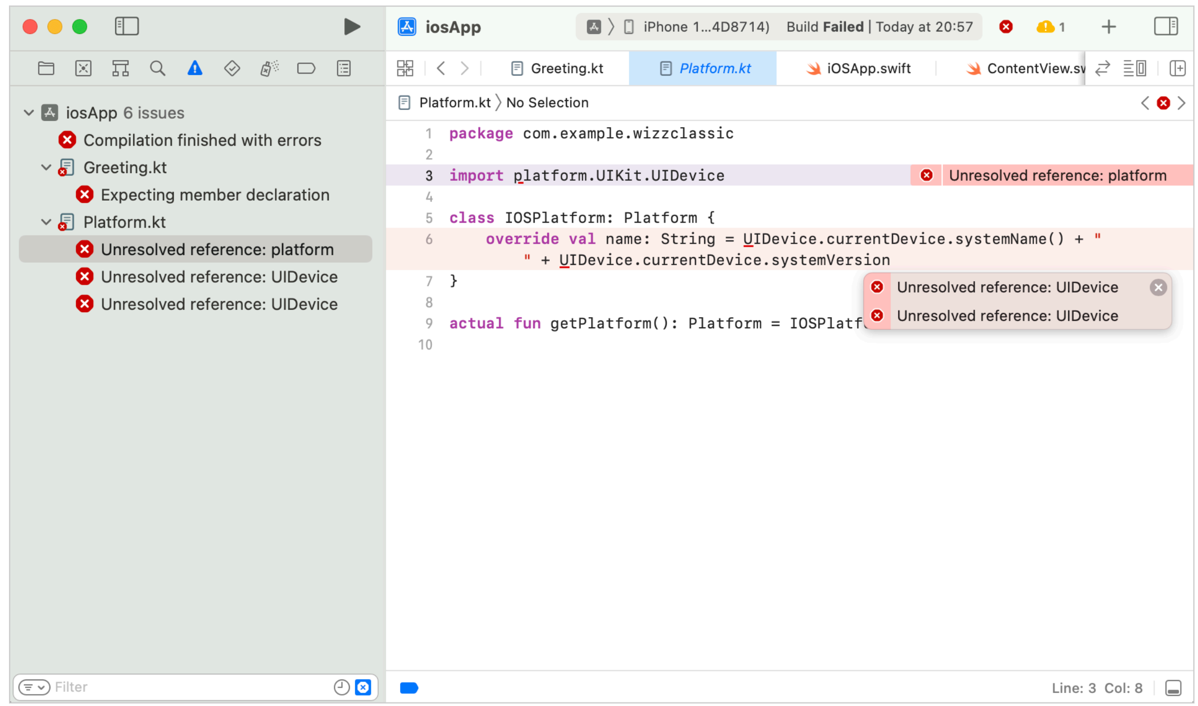
- 对于 IntelliJ IDEA 2023.1 EAP 目前还没有 Kotlin 1.8.20 plugin 可用. 尽管如此, 你还是可以将 Kotlin Gradle plugin 版本设置为 1.8.20-RC2, 在这个 IDE 中试用复合构建.
- 如果你的项目包含指定了 `rootProject.name` 的构建, 复合构建可能会无法解析 Kotlin metadata. 关于这个问题的详细情况, 以及变通方法, 请参见这个 Youtrack issue (https://youtrack.jetbrains.com/issue/KT-56536).

我们鼓励你试用这个功能, 并提交报告到 YouTrack (https://kotl.in/issue), 帮助我们, 让这个功能在 Kotlin 1.9.0 中默认启用.

Xcode 中 Gradle 错误信息的改进

如果在 Xcode 中构建你的跨平台项目时遇到问题, 你可能看到 "Command PhaseScriptExecution failed with a nonzero exit code" 错误信息. 这个错误信息表示 Gradle 调用失败了, 但要调查问题的原因, 这个错误信息就没什么帮助.

从 Kotlin 1.8.20 开始, Xcode 能够解析 Kotlin/Native 编译器的输出. 而且, 对于 Gradle 构建失败的情况, 你会在 Xcode 中看到来自根本原因异常的附加错误信息. 大多数情况下, 这些信息能够帮助你找到根本问题.



Xcode 中 Gradle 错误信息的改进

对用于 Xcode 集成的标准 Gradle task, 这个新行为默认启用, 例如 `embedAndSignAppleFrameworkForXcode`, 它能够将 iOS 框架从你的跨平台应用程序连接到 Xcode 中的 iOS 应用程序. 也可以使用 `kotlin.native.useXcodeMessageStyle` Gradle 属性来启用 (或关闭).

Kotlin/JavaScript

Kotlin 1.8.20 修改了 TypeScript 定义的生成方式. 还包含了一个变更, 改善你的调试体验:

- 从 Gradle plugin 中删除 Dukat 集成
- 代码映射(Source Map) 中的 Kotlin 变量和函数名称
- TypeScript 定义文件生成的使用者同意

从 Gradle plugin 中删除 Dukat 集成

在 Kotlin 1.8.20 中, 我们从 Kotlin/JavaScript Gradle plugin 中删除了 实验性的 ("稳定性级别" in "Kotlin 各部分组件的稳定性") Dukat 集成功能. Dukat 集成功能支持从 TypeScript 声明文件 (`.d.ts`) 到 Kotlin 外部声明的自动转换.

你仍然可以使用我们的 Dukat 工具 (https://github.com/Kotlin/dukat), 将 TypeScript 声明文件 (`.d.ts`) 转换为 Kotlin 外部声明.

Dukat 工具是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除.

代码映射(Source Map) 中的 Kotlin 变量和函数名称

为了帮助调试, 我们引入了一种功能, 能够向你的代码映射(Source Map)添加你在 Kotlin 代码中声明的变量和函数的名称. 在 1.8.20 之前, 这些名称在代码映射(Source Map)中是不可用的, 因此在调试器中, 你看到的是生成的 JavaScript 的变量和函数名称.

你可以在你的 Gradle 文件 `build.gradle.kts` 中使用 `sourceMapNamesPolicy` 来配置添加哪些名称, 也可以使用编译器选项 `-source-map-names-policy`. 下表是可用的设置:

设置	说明	输出示例
`simple-names`	添加变量名称和函数的简单名称. (默认值)	`main`
`fully-qualified-names`	添加变量名称和函数的完全限定名称.	`com.example.kjs.playground.main`
`no`	不添加变量名称和函数名称.	无

下面是在 `build.gradle.kts` 文件中配置的示例:

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.Kotlin2JsCompile>
().configureEach {

compilercompileOptions.sourceMapNamesPolicy.set(org.jetbrains.kotlin
.gradle.dsl.JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_FQ_NAMES)
// 或 SOURCE_MAP_NAMES_POLICY_NO, or
```

```
SOURCE_MAP_NAMES_POLICY_SIMPLE_NAMES
}
```

调试工具, 例如基于 Chromium 的浏览器中提供的调试工具, 能够从你的代码映射中获取原始的 Kotlin 名称, 改进你的调用栈的可读性. 祝你调试快乐!

> ⚠ 在代码映射中添加变量和函数名称是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除.

TypeScript 定义文件生成的使用者同意

以前, 如果你的项目生成可执行的文件 (`binaries.executable()`), Kotlin/JS IR 编译器会收集所有标注了 `@JsExport` 的顶级声明, 并自动在一个 `.d.ts` 文件中生成 TypeScript 定义.

由于这个功能并不是对每个项目都有用, 在 Kotlin 1.8.20 中我们修改了这个行为. 如果你想要生成 TypeScript 定义, 你需要在你的 Gradle 构建文件中明确的配置. 向你的 `build.gradle.kts.file` 文件的 `js` 小节 ("执行环境" in "创建 Kotlin/JS 工程(Project)") 添加 `generateTypeScriptDefinitions()`. 例如:

```
kotlin {
    js {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

> ⚠ TypeScript 定义 (`d.ts`) 的生成是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除.

Gradle

除 Multiplatform plugin 中的一些特殊情况 (https://youtrack.jetbrains.com/issue/KT-55751) 外, Kotlin 1.8.20 与 Gradle 6.8 到 7.6 完全兼容. 你也可以使用最新的 Gradle 版本, 但如果你这样做, 请注意, 你可能遇到废弃警告, 或一些新的 Gradle 功能无法工作.

这个发布版带来了以下变更:

- 新的 Gradle plugin 版本对齐方式
- Gradle 中默认启用新的 JVM 增量编译
- 对编译任务的输出的精确备份
- 对所有 Gradle 版本, 延迟创建 Kotlin/JVM 任务
- 处理编译任务的输出目录不是默认位置的情况
- 能够选择性禁用(opt out)向 HTTP 统计服务报告编译器参数的功能

新的 Gradle plugin 版本对齐方式

Gradle 提供了一种方式, 保证那些需要一起工作的依赖项能够 对齐它们的版本 (https://docs.gradle.org/current/userguide/dependency_version_alignment.html#aligning_versions_natively_with_gradle). Kotlin 1.8.20 也采用了这个方案. 这个功能默认启用, 因此你不需要修改或更新你的配置来启用它. 此外, 你不再需要 使用这个变通方法来解析 Kotlin Gradle plugin 的传递依赖项 ("Kotlin Gradle plugin 的传递依赖项的解析" in "Kotlin 1.8.0 版中的新功能").

希望你能通过 YouTrack (https://youtrack.jetbrains.com/issue/KT-54691) 提供你的反馈意见.

Gradle 中默认启用新的 JVM 增量编译

增量编译的新方案, 从 Kotlin 1.7.0 开始可以使用 ("增量编译的新方案" in "Kotlin 1.7.0 版中的新功能"), 现在变为默认使用. 你不再需要在你的 `gradle.properties` 中指定 `kotlin.incremental.useClasspathSnapshot=true` 来启用它.

希望你能提供你的反馈意见. 你可以在 YouTrack 中 提交一个 issue (https://kotl.in/issue).

对编译任务的输出的精确备份

> 对编译任务的输出的精确备份是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 要使用这个功能, 请向 `gradle.properties` 添加 `kotlin.compiler.preciseCompilationResultsBackup=true`. 希望你能通过 YouTrack (https://kotl.in/issue/experimental-ic-optimizations) 提供你的反馈意见.

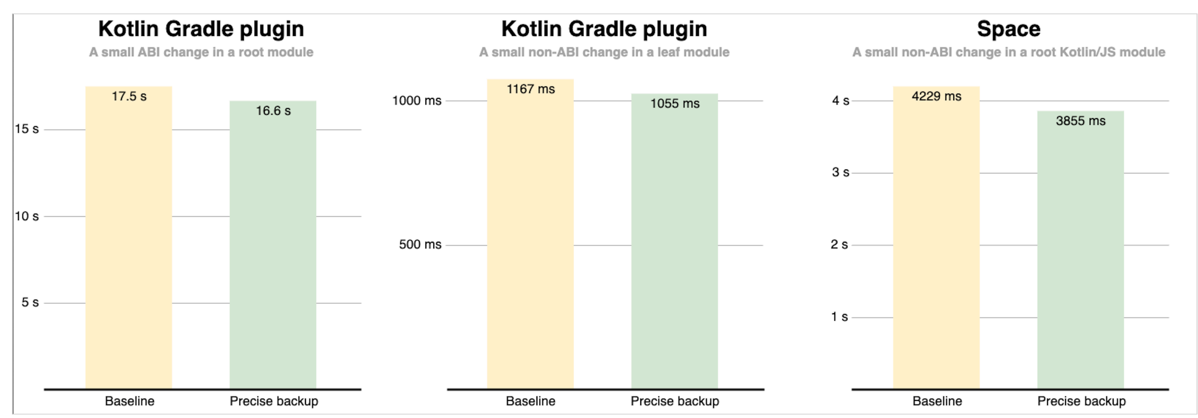
从 Kotlin 1.8.20 开始, 你可以启用精确备份, 这时只有 Kotlin 在 增量编译 ("增量编译(Incremental compilation)" in "Kotlin Gradle plugin 中的编译与缓存") 中重新编译的那些类会被备份. 完整备份和精确备份都可以帮助在发生编译错误后再次运行增量构建. 精确备份与完整备份相比, 会耗费较少的构建时间. 对于大型的项目, 或者很多任务都创建备份, 那么完整备份可能会花费 **明显** 更长的构建时间, 尤其是如果项目位于速度较慢的 HDD 上.

这个优化是实验性功能. 要启用这个功能, 请向 `gradle.properties` 文件添加 `kotlin.compiler.preciseCompilationResultsBackup` Gradle 属性:

```
kotlin.compiler.preciseCompilationResultsBackup=true
```

JetBrains 使用精确备份的例子

在下面的图表中, 你可以看到使用精确备份与完整备份相对比的示例:



完整备份与精确备份的对比

第一个和第二个对比图显示了在 Kotlin 项目中使用精确备份时对 Kotlin Gradle plugin 构建的影响:

1. 进行一个小的 ABI (https://en.wikipedia.org/wiki/Application_binary_interface) 变更之后: 向一个被大量模块依赖的模块添加一个新的 public 方法.
2. 进行一个小的非 ABI 变更之后: 向一个没有被其他模块依赖的模块添加一个 private 函数.

第三个对比图显示了在 Space (https://www.jetbrains.com/space/) 项目中使用精确备份时, 在小的非 ABI 更改后对 Web 前端构建的影响: 向一个被大量模块依赖的 Kotlin/JS 模块添加一个 private 函数.

我们在使用 Apple M1 Max CPU 的计算机上进行这些测量; 在不同的计算机上会出现稍微不同的结果. 影响性能的因素包括但不限于以下几点:

- Kotlin daemon ("Kotlin daemon 及其在 Gradle 中的使用" in "Kotlin Gradle plugin 中的编译与缓存") 和 Gradle daemon (https://docs.gradle.org/current/userguide/gradle_daemon.html) 热身状况(warm)如何..
- 硬盘速度如何.

- CPU 型号, 以及它的繁忙程度.

- 哪些模块受到变更的影响, 以及这些模块有多大.

- 是 ABI 变更还是非 ABI 变更.

使用构建报告来评估优化

要对你的项目和场景, 评估优化在你的计算机上的影响, 你可以使用 Kotlin 构建报告 ("构建报告" in "Kotlin Gradle plugin 中的编译与缓存"). 请向你的 `gradle.properties` 文件添加下面的属性, 启用文本文件格式的构建报告:

```
kotlin.build.report.output=file
```

下面是在启用精确备份之前, 构建报告的相关部分的示例:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.59 s
<...>
Time metrics:
 Total Gradle task time: 0.59 s
 Task action before worker execution: 0.24 s
  Backup output: 0.22 s // 注意这个数字
<...>
```

下面是在启用精确备份之后, 构建报告的相关部分的示例:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.46 s
<...>
Time metrics:
 Total Gradle task time: 0.46 s
 Task action before worker execution: 0.07 s
  Backup output: 0.05 s // 备份消耗的时间减少了
 Run compilation in Gradle worker: 0.32 s
  Clear jar cache: 0.00 s
  Precise backup output: 0.00 s // 与精确备份相关的输出
  Cleaning up the backup stash: 0.00 s // 与精确备份相关的输出
<...>
```

对所有 Gradle 版本, 延迟创建 Kotlin/JVM 任务

对于在 Gradle 7.3+ 中使用了 `org.jetbrains.kotlin.gradle.jvm` plugin 的项目, Kotlin Gradle plugin 不会过早的创建和配置 `compileKotlin` 任务. 在更低版本的 Gradle 中, 它只是简单的注册所有任务, 不会在空运行(dry run)阶段配置任务. 在使用 Gradle 7.3+ 时, 现在也会是相同的行为.

处理编译任务的输出目录不是默认位置的情况

如果你有下面的设置, 那么请更新你的构建脚本, 添加一些新的设置:

- 覆盖了 Kotlin/JVM `KotlinJvmCompile`/`KotlinCompile` 任务的 `destinationDirectory` 位置.
- 使用了废弃的 Kotlin/JS/非 IR 变体(variant) (对 Gradle plugin 变体的支持), 并覆盖了 `Kotlin2JsCompile` 任务的 `destinationDirectory`.

在你的 JAR 文件中, 除 `sourceSets.main.outputs` 之外, 你需要明确的添加 `sourceSets.main.kotlin.classesDirectories`:

```
tasks.jar(type: Jar) {
    from sourceSets.main.outputs
    from sourceSets.main.kotlin.classesDirectories
}
```

能够选择性禁用(opt out)向 HTTP 统计服务报告编译器参数的功能

现在你可以控制 Kotlin Gradle plugin 是否应该在 HTTP 构建报告 ("构建报告" in "Kotlin Gradle plugin 中的编译与缓存") 中包含编译器参数. 有些时候, 你可能不需要让 plugin 报告这些参数. 如果一个项目包含很多模块, 它在报告中的的编译器参数 可能非常多, 而且没什么用处. 现在有一种方法能够关闭这个信息, 并节省内存. 请在你的 `gradle.properties` 或 `local.properties` 文件中, 使用 `kotlin.build.report.include_compiler_arguments=(true|false)` 属性.

希望你能通过 YouTrack (https://youtrack.jetbrains.com/issue/KT-55323/) 提供你的反馈意见.

标准库

Kotlin 1.8.20 添加了很多新的功能, 包括一些对 Kotlin/Native 开发非常有用的功能:

- 支持 AutoCloseable 接口
- 支持 Base64 编码和解码
- 在 Kotlin/Native 中支持 @Volatile

- 在 Kotlin/Native 中使用正规表达式时堆栈溢出问题的重大修正

支持 AutoCloseable 接口

> ⚠ 新的 AutoCloseable 接口是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"), 要使用它, 你需要通过 @OptIn(ExperimentalStdlibApi::class) 标注使用者同意 (Opt-in), 或通过编译器参数 -opt-in=kotlin.ExperimentalStdlibApi.

AutoCloseable 接口已经添加到了共通的标准库, 因此你可以对所有的库使用共通的接口来关闭资源. 在 Kotlin/JVM 中, AutoCloseable 接口是 java.lang.AutoClosable (https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html) 的别名(alias).

此外, 还包含了扩展函数 use(), 它会对一个指定的资源执行一个给定的函数块, 然后正确的关闭这个资源, 无论函数块执行过程中是否抛出了异常.

在共通的标准库中没有实现 AutoCloseable 接口的 public 类. 在下面的示例中, 我们定义了一个 XMLWriter 接口, 假设有一个资源实现了这个接口. 例如, 这个资源可以是一个类, 它打开文件, 写入 XML 内容, 然后关闭文件.

```
interface XMLWriter : AutoCloseable {
    fun document(encoding: String, version: String, content:
XMLWriter.() -> Unit)
    fun element(name: String, content: XMLWriter.() -> Unit)
    fun attribute(name: String, value: String)
    fun text(value: String)
}

fun writeBooksTo(writer: XMLWriter) {
    writer.use { xml ->
        xml.document(encoding = "UTF-8", version = "1.0") {
            element("bookstore") {
                element("book") {
                    attribute("category", "fiction")
                    element("title") { text("Harry Potter and the
Prisoner of Azkaban") }
                    element("author") { text("J. K. Rowling") }
                    element("year") { text("1999") }
                    element("price") { text("29.99") }
                }
```

```
                element("book") {
                    attribute("category", "programming")
                    element("title") { text("Kotlin in Action") }
                    element("author") { text("Dmitry Jemerov") }
                    element("author") { text("Svetlana Isakova") }
                    element("year") { text("2017") }
                    element("price") { text("25.19") }
                }
            }
        }
    }
}
```

支持 Base64 编码

新的编码和解码功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"), 要使用它, 你需要通过 `@OptIn(ExperimentalEncodingApi::class)` 标注使用者同意(Opt-in), 或通过编译器参数 `-opt-in=kotlin.io.encoding.ExperimentalEncodingApi`.

我们添加了 Base64 编码和解码的支持. 我们提供了 3 个类实例, 每个使用不同的编码方案, 并表现出不同的行为. 对于标准的 Base64 编码方案 (https://www.rfc-editor.org/rfc/rfc4648#section-4), 请使用 `Base64.Default` 实例.

对于 "URL 和文件名安全的" (https://www.rfc-editor.org/rfc/rfc4648#section-5) 编码方案, 请使用 `Base64.UrlSafe` 实例.

对于 MIME (https://www.rfc-editor.org/rfc/rfc2045#section-6.8) 编码方案, 请使用 `Base64.Mime` 实例. 如果你使用 `Base64.Mime` 实例, 所有的编码函数会对每 76 个字符插入 1 个行分隔符. 对于解码的情况, 所有的非法字符会被跳过, 不抛出异常.

`Base64.Default` 实例 `Base64` 类的是伴随对象. 因此, 你可以通过 `Base64.encode()` 和 `Base64.decode()` 的方式调用它的函数, 而不必写为 `Base64.Default.encode()` 和 `Base64.Default.decode()`.

```
val foBytes = "fo".map { it.code.toByte() }.toByteArray()
Base64.Default.encode(foBytes) // 结果为 "Zm8="
// 也可以写为:
// Base64.encode(foBytes)
```

```
val foobarBytes = "foobar".map { it.code.toByte() }.toByteArray()
Base64.UrlSafe.encode(foobarBytes) // 结果为 "Zm9vYmFy"

Base64.Default.decode("Zm8=") // 结果等于 foBytes
// 也可以写为:
// Base64.decode("Zm8=")

Base64.UrlSafe.decode("Zm9vYmFy") // 结果等于 foobarBytes
```

你可以使用其它函数编码或解码字节, 结果输出到已经存在的缓冲区, 或者将结果添加到指定的 `Appendable` 类型对象.

在 Kotlin/JVM 中, 我们还添加了扩展函数 `encodingWith()` 和 `decodingWith()`, 可以对输入和输出流执行 Base64 编码和解码操作.

在 Kotlin/Native 中支持 @Volatile

> Kotlin/Native 中的 `@Volatile` 是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过 YouTrack (https://kotl.in/issue) 提供你的反馈意见.

如果你使用 `@Volatile` 注解标注一个 `var` 属性, 那么它的后端域变量(Backing Field) 会被标注这个注解, 使得对这个域变量的所有读写操作都是原子化的, 而且写入操作永远对其它线程可见.

在 1.8.20 之前, `kotlin.jvm.Volatile` 注解 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-volatile/) 存在于在共通标准库中. 但是, 这个注解只对 JVM 有效. 如果你在 Kotlin/Native 中使用它, 它会被忽略, 因此导致错误.

在 1.8.20 中, 我们引入了一个共通的注解, `kotlin.concurrent.Volatile`, 你可以在 JVM 和 Kotlin/Native 中使用.

如何启用

要试用这个功能, 请使用 `@OptIn(ExperimentalStdlibApi)` 标注使用者同意(Opt-in), 并启用 `-language-version 1.9` 编译器选项. 在 Gradle 项目中, 你可以在你的 `build.gradle(.kts)` 文件中添加以下内容:

Kotlin

```
tasks

.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

Groovy

```
tasks

.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k.class)
    .configureEach {
        compilerOptions.languageVersion =

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
}
```

在 Kotlin/Native 中使用正规表达式时堆栈溢出问题的重大修正

以前 Kotlin 的版本中, 如果你的正规表达式的输入包含了大量的字符, 可能会发生崩溃, 即使正规表达式模式本身非常简单. 在 1.8.20 中, 已经解决了这个问题. 更多详情, 请参见 KT-46211 (https://youtrack.jetbrains.com/issue/KT-46211).

序列化的更新

Kotlin 1.8.20 包含 对 Kotlin K2 编译器的 Alpha 支持, 以及 禁止通过伴随对象定制序列化器.

对 Kotlin K2 编译器的序列化编译器 plugin (Prototype)

> ⚠ 对 K2 的序列化编译器 plugin 支持处于 Alpha 阶段 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 要使用它, 请 启用 Kotlin K2 编译器.

从 1.8.20 开始, 序列化编译器 plugin 可以与 Kotlin K2 编译器一起使用. 请试用它, 并 向我们提供你的反馈意见!

禁止通过伴随对象隐含的定制序列化器

目前, 可以使用 `@Serializable` 注解将一个类声明为可序列化, 同时还可以在它的伴随对象上, 使用 `@Serializer` 注解声明一个自定义的序列化器.

例如:

```
import kotlinx.serialization.*

@Serializable
class Foo(val a: Int) {
    @Serializer(Foo::class)
    companion object {
        // KSerializer<Foo> 的自定义实现
    }
}
```

这种情况下, 从 `@Serializable` 注解无法看出使用了哪个序列化器. 实际上, `Foo` 类存在一个自定义的序列化器.

为了防止这种混乱, 在 Kotlin 1.8.20 中, 在检测到这种情况时, 我们引入了一个编译器警告. 警告信息中包含一个可能的迁移方案来解决这个问题.

如果你在你的代码中使用了这样的结构, 我们建议修改如下:

```
import kotlinx.serialization.*

@Serializable(Foo.Companion::class)
class Foo(val a: Int) {
    // 无论是是否标注 @Serializer(Foo::class), 都会起作用
    companion object: KSerializer<Foo> {
        // KSerializer<Foo> 的自定义实现
```

```
    }
}
```

如果这个方案, 可以很清楚的看到, Foo 类使用了伴随对象中声明的自定义的序列化器. 更多详情, 请参见我们的 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-54441).

在 Kotlin 2.0 中, 我们计划将编译警告升级为编译错误. 如果你看到这个警告, 我们建议你迁移你的代码.

文档更新

Kotlin 文档有了一些重要变更:

- Spring Boot 和 Kotlin 入门 (Spring Boot 和 Kotlin 入门) – 创建一个使用数据库的简单的应用程序, 详细了解 Spring Boot 和 Kotlin 的功能.
- 作用域函数(Scope Function) (作用域函数(Scope Function)) – 了解如何使用标准库中有用的作用域函数来简化代码.
- CocoaPods 集成 (CocoaPods 概述与设置) – 设置使用 CocoaPods 的环境.

安装 Kotlin 1.8.20

检查 IDE 版本

IntelliJ IDEA (https://www.jetbrains.com/idea/download/) 2022.2 和 2022.3 会自动建议将 Kotlin plugin 更新到 1.8.20. IntelliJ IDEA 2023.1 会包含 Kotlin plugin 1.8.20.

Android Studio Flamingo (222) 和 Giraffe (223) 会在后续的发布版中支持 Kotlin 1.8.20.

新的命令行编译器可以通过 GitHub 发布页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.8.20) 下载.

配置 Gradle 的设置

要正确下载 Kotlin 的 artifact 和依赖项, 请更新你的 settings.gradle(.kts) 文件, 使用 Maven Central 仓库:

```
pluginManagement {
    repositories {
```

```
        mavenCentral()
        gradlePluginPortal()
    }
}
```

如果没有指定仓库, Gradle 会使用已废弃的 JCenter 仓库, 导致无法下载 Kotlin artifact 的错误.

Kotlin 1.8.0 版中的新功能

最终更新: 2024/09/10

发布日期: 2022/12/28 (["各发布版详情" in "Kotlin 的发布版本"])

Kotlin 1.8.0 已经发布了, 以下是它的一些最重要的功能:

- JVM 平台的新的实验性功能: 目录内容的递归复制或递归删除
- kotlin-reflect 的性能改善
- 新的 -Xdebug 编译器选项, 改进调试体验
- `kotlin-stdlib-jdk7` 和 `kotlin-stdlib-jdk8` 合并到 `kotlin-stdlib`
- 与 Objective-C/Swift 交互能力的改进
- 兼容 Gradle 7.3

IDE 支持

以下 IDE 可以使用支持 1.8.0 的 Kotlin plugin:

IDE	支持的版本
IntelliJ IDEA	2021.3, 2022.1, 2022.2
Android Studio	Electric Eel (221), Flamingo (222)

> 在 IntelliJ IDEA 2022.3 中, 你可以将你的项目升级到 Kotlin 1.8.0, 而不必升级 IDE plugin.
>
> 在 IntelliJ IDEA 2022.3 中, 要将已有的项目迁移到 Kotlin 1.8.0, 请将 Kotlin 版本修改为 `1.8.0`, 然后重新导入你的 Gradle 项目或 Maven 项目.

Kotlin/JVM

从 1.8.0 版开始, 编译器可以生成字节码版本对应于 JVM 19 的类. 新的语言版本还包括以下功能:

- 一个编译器选项, 关闭 JVM 注解对象的生成
- 一个新的 `-Xdebug` 编译器选项, 禁用代码优化
- 删除了旧的编译器后端
- 支持 Lombok 的 @Builder 注解

不生成 TYPE_USE 和 TYPE_PARAMETER 注解目标(Target)的能力

如果一个 Kotlin 注解的 Kotlin 注解目标(Target)中包含 `TYPE`, 那么映射的 Java 注解目标会包含 `java.lang.annotation.ElementType.TYPE_USE`. 同样的, Kotlin 注解目标 `TYPE_PARAMETER` 会映射为 Java 注解目标 `java.lang.annotation.ElementType.TYPE_PARAMETER`. 对于 API 级别低于 26 的 Android 用户来说, 这会造成问题, 因为在 API 中不存在这些注解目标.

从 Kotlin 1.8.0 开始, 你可以使用新的编译器选项 `-Xno-new-java-annotation-targets`, 避免生成 `TYPE_USE` 和 `TYPE_PARAMETER` 注解目标.

新的编译器选项, 禁止代码优化

Kotlin 1.8.0 添加了一个新的 `-Xdebug` 编译器选项, 它会禁止代码优化, 改善调试体验. 目前, 这个选项会对 coroutine 禁用 "was optimized out" 功能. 将来, 我们添加了更多的代码优化之后, 这个选项也会禁用这些代码优化功能.

当你使用挂起函数时, "was optimized out" 功能会优化变量. 如果变量被优化, 调试代码会变得困难, 因为你看不到变量值.

绝对不要在产品环境中使用这个选项: 使用 `-Xdebug` 禁用这个功能, 会 导致内存泄露 (https://youtrack.jetbrains.com/issue/KT-48678/Coroutine-debugger-disable-was-optimised-out-compiler-feature#focus=Comments-27-6015585.0-0).

删除了旧的编译器后端

在 Kotlin 1.5.0 中, 我们 宣布了 ("JVM IR 后端的稳定版" in "Kotlin 1.5.0 版中的新功能") 基于 IR 的编译器后端进入 稳定版 (Kotlin 各部分组件的稳定性). 因此从 Kotlin 1.4.* 开始的旧的编译器后端已被废弃. 在 Kotlin 1.8.0 中, 我们完全删除了旧的编译器后端. 同时, 我们也删除了编译器选项 `-Xuse-old-backend` 和 Gradle 选项 `useOldBackend`.

支持 Lombok 的 @Builder 注解

由于开发社区大量投票支持 Kotlin Lombok: Support generated builders (@Builder) (https://youtrack.jetbrains.com/issue/KT-46959), 因此我们决定支持 @Builder 注解 (https://projectlombok.org/features/Builder).

我们还没有支持 `@SuperBuilder` 和 `@Tolerate` 注解的计划, 但如果有足够多的人投票支持 @SuperBuilder (https://youtrack.jetbrains.com/issue/KT-53563/Kotlin-Lombok-Support-SuperBuilder) 和 @Tolerate (https://youtrack.jetbrains.com/issue/KT-53564/Kotlin-Lombok-Support-Tolerate), 我们可以考虑增加这个功能.

参见 如何配置 Lombok 编译器插件 ("Gradle" in "Lombok 编译器插件").

Kotlin/Native

Kotlin 1.8.0 包含对 Objective-C 和 Swift 交互能力的改进, 支持 Xcode 14.1, 以及对 CocoaPods Gradle plugin 改进:

- 支持 Xcode 14.1
- Objective-C/Swift 交互能力的改进
- 在 CocoaPods Gradle plugin 中默认使用动态框架(Dynamic framework)

支持 Xcode 14.1

Kotlin/Native 编译器限制支持 Xcode 的最新稳定版, 14.1. 具体改善的内容包括:

- 对 watchOS 编译目标, 添加了新的 `watchosDeviceArm64` 预设置(preset), 支持 ARM64 平台上的 Apple watchOS.
- Kotlin CocoaPods Gradle plugin 对 Apple 框架不再默认包含 bitcode 嵌入(embedding).
- 更新了平台库, 以反应 Apple 编译目标 Objective-C 框架的变更.

Objective-C/Swift 交互能力的改进

为了增强 Kotlin 与 Objective-C 和 Swift 的交互能力, 添加了 3 个新的注解:

- `@ObjCName` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native/-obj-c-name/) 允许你指定一个在 Swift 或 Objective-C 中更加符合语言习惯的名称, 而不是使用 Kotlin 声明的名称.

这个注解指示 Kotlin 编译器, 对类, 属性, 参数, 或函数, 使用一个自定义的 Objective-C 和 Swift 名称:

```
@ObjCName(swiftName = "MySwiftArray")
class MyKotlinArray {
    @ObjCName("index")
    fun indexOf(@ObjCName("of") element: String): Int = TODO()
}

// 使用 ObjCName 注解后的用法
let array = MySwiftArray()
let index = array.index(of: "element")
```

- `@HiddenFromObjC` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native/-hidden-from-obj-c/) 允许你对 Objective-C 隐藏一个 Kotlin 声明.

  这个注解指示 Kotlin 编译器, 不要将一个函数或属性导出到 Objective-C 以及 Swift. 这样可以让你的 Kotlin 代码对 Objective-C/Swift 更加友好.

- `@ShouldRefineInSwift` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native/-should-refine-in-swift/) 可以将一个 Kotlin 声明在 Swift 中替换为一个 wrapper.

  这个注解指示 Kotlin 编译器, 将一个函数或属性标记为 `swift_private` 在生成的 Objective-C API 中. 这样的声明会带上 `__` 前缀, 因此对于 Swift 代码来说不可见.

  你仍然可以在 Swift 代码中使用这些声明, 来创建 Swift 友好的 API, 但在 Xcode 的代码自动完成功能中, 不会显示这些声明.

  关于如何在 Swift 中润色(Refine) Objective-C 声明, 详情请参见 Apple 官方文档 (https://developer.apple.com/documentation/swift/improving-objective-c-api-declarations-for-swift).

> 这个新注解需要 使用者同意(Opt-in) (明确要求使用者同意的功能(Opt-in Requirement)).

Kotlin 开发组非常感谢 Rick Clephas (https://github.com/rickclephas) 实现了这些注解.

CocoaPods Gradle plugin 中默认使用动态框架(Dynamic framework)

从 Kotlin 1.8.0 开始, 由 CocoaPods Gradle plugin 注册的 Kotlin 框架默认使用动态链接. 以前的静态实现与 Kotlin Gradle plugin 的行为不一致.

```
kotlin {
    cocoapods {
        framework {
            baseName = "MyFramework"
            isStatic = false // 现在默认是 dynamic
        }
    }
}
```

如果你已有的项目使用静态链接类型, 而且你想要升级到 Kotlin 1.8.0 (或者明确修改链接类型), 那么在项目执行时可能会发生错误. 要修正这个错误, 请关闭你的 Xcode 项目, 并在 Podfile 目录中运行 `pod install`.

详情请参见, CocoaPods Gradle plugin DSL 参考文档 (CocoaPods Gradle plugin DSL 参考文档).

Kotlin Multiplatform: 新的 Android 源代码集布局

Kotlin 1.8.0 引入了新的 Android 源代码集布局, 替换了以前的目录命名方式, 旧方式容易造成很多误解.

例如, 假设在当前的布局中创建了 2 个 `androidTest` 目录. 一个用于 `KotlinSourceSets`, 另一个用于 `AndroidSourceSets`:

- 这 2 个目录代表不同的含义: Kotlin 的 `androidTest` 属于 `unitTest` 类型, 而 Android 的属于 `integrationTest` 类型.
- 这 2 个目录造成了易于误解的 `SourceDirectories` 布局, 因为 `src/androidTest/kotlin` 包含 `UnitTest`, 而 `src/androidTest/java` 包含 `InstrumentedTest`.
- 对于 Gradle 配置来说, `KotlinSourceSets` 和 `AndroidSourceSets` 都使用类似的命名方式, 因此 Kotlin 和 Android 源代码集 `androidTest` 的配置结果是一样的: `androidTestImplementation`, `androidTestApi`, `androidTestRuntimeOnly`, 以及 `androidTestCompileOnly`.

为了解决这些问题, 以及其他一些问题, 我们引入了一种新的 Android 源代码集布局. 以下是两种布局的一些关键差别:

KotlinSourceSet 命名方式

当前的源代码集布局	新的源代码集布局
`targetName` + `AndroidSourceSet.name`	`targetName` + `AndroidVariantType`

`{AndroidSourceSet.name}` 与 `{KotlinSourceSet.name}` 的对应关系如下:

	当前源代码集布局	新的源代码集布局
main	androidMain	androidMain
test	androidTest	android**Unit**Test
androidTest	android**Android**Test	android**Instrumented**Test

SourceDirectories

当前源代码集布局	新的源代码集布局
布局会添加额外的 `/kotlin` 源代码目录	`src/{AndroidSourceSet.name}/kotlin`, `src/{KotlinSourceSet.name}/kotlin`

`{AndroidSourceSet.name}` 与 `{包含的 SourceDirectories}` 的对应关系如下:

	当前源代码集布局	新的源代码集布局
main	src/androidMain/kotlin, src/main/kotlin, src/main/java	src/androidMain/kotlin, src/main/kotlin, src/main/java
test	src/androidTest/kotlin, src/test/kotlin, src/test/java	src/android**Unit**Test/kotlin, src/test/kotlin, src/test/java
androidTest	src/android**Android**Test/kotlin, src/androidTest/java	src/android**Instrumented**Test/kotlin, src/androidTest/java, **src/androidTest/kotlin**

AndroidManifest.xml 文件位置

当前源代码集布局	新的源代码集布局
src/{**Android**SourceSet.name}/AndroidManifest.xml	src/{**Kotlin**SourceSet.name}/AndroidManifest.xml

`{AndroidSourceSet.name}` 与 `{AndroidManifest.xml 位置}` 的对应关系如下:

	当前源代码集布局	新的源代码集布局
main	src/main/AndroidManifest.xml	src/**android**Main/AndroidManifest.xml
debug	src/debug/AndroidManifest.xml	src/**android**Debug/AndroidManifest.xml

Android 测试与 common 测试之间的关系

新的 Android 源代码集布局改变了 Android-instrumented 测试 (在新的布局中名称变更为 `androidInstrumentedTest`) 与 common 测试之间的关系.

在以前的版本中, `androidAndroidTest` 和 `commonTest` 之间存在默认的 `dependsOn` 关系. 具体来说, 代表以下含义:

- 在 `androidAndroidTest` 中可以访问 `commonTest` 中的代码.
- `commonTest` 中的 `expect` 声明在 `androidAndroidTest` 中必须有对应的 `actual` 实现.
- 在 `commonTest` 中声明的测试, 也会作为 Android instrumented 测试执行.

在新的 Android 源代码集布局中, 不再默认添加这个 `dependsOn` 关系. 如果你期望切换到以前的行为, 请在你的 `build.gradle.kts` 文件中, 手动声明这个关系:

```
kotlin {
    // ...
    sourceSets {
        val commonTest by getting
        val androidInstrumentedTest by getting {
            dependsOn(commonTest)
        }
```

```
    }
}
```

对 Android flavor 的支持

在以前的版本中, Kotlin Gradle plugin 会在很早的阶段创建对应于 `debug` 和 `release` 构建类型的 Android 源代码集, 或对应于自定义 flavor 的 Android 源代码集, 例如 `demo` 和 `full`. 因此这些源代码集可以通过 `val androidDebug by getting { ... }` 这样的结构来访问.

在新的 Android 源代码集布局中, 这些源代码集会在 `afterEvaluate` 阶段创建. 因此上面的表达式不再有效, 会导致错误: `org.gradle.api.UnknownDomainObjectException: KotlinSourceSet with name 'androidDebug' not found`.

为了解决这样的错误, 请在你的 `build.gradle.kts` 文件中使用新的 `invokeWhenCreated()` API:

```
kotlin {
    // ...
    sourceSets.invokeWhenCreated("androidFreeDebug") {
        // ...
    }
}
```

配置与设置

在未来的发布版中, 将会默认使用新的布局. 你可以使用以下 Gradle 选项来启用它:

```
kotlin.mpp.androidSourceSetLayoutVersion=2
```

> 新的布局需要 Android Gradle plugin 7.0 或更高版本, 以及 Android Studio 2022.3 或更高版本.

现在不再鼓励使用以前的 Android 风格目录布局. Kotlin 1.8.0 开始启动了旧布局的废弃周期, 会对当前的布局提示警告信息. 你可以使用以下 Gradle 属性来禁止这个警告:

```
kotlin.mpp.androidSourceSetLayoutVersion1.nowarn=true
```

Kotlin/JS

Kotlin 1.8.0 发布了 JS IR 编译器后端的稳定版, 并对 JavaScript 相关的 Gradle 构建脚本带来了新的功能特性:

- JS IR 编译器后端的稳定版
- 新的设置, 用于报告 yarn.lock 文件已被更新
- 通过 Gradle 属性添加用于浏览器的测试目标
- 向你的项目添加 CSS 支持的新方式

JS IR 编译器后端的稳定版

从这个发布版开始, 基于中间代码 (Intermediate Representation, IR) 的 Kotlin/JS 编译器 (使用 IR 编译器) 后端进入稳定版. 我们花费了一些时间来对全部三种后端统一基础设施, 但现在这些后端对 Kotlin 代码可以使用相同的 IR.

由于 JS IR 编译器后端已经进入稳定版, 因此旧的后端现在已被废弃.

对于 JS IR 编译器的稳定版, 增量编译会默认启用.

如果你还在使用旧的编译器, 请将你的项目切换到新的后端, 具体方法请参见我们的 迁移指南 (将 Kotlin/JS 项目迁移到 IR 编译器).

新的设置, 用于报告 yarn.lock 文件已被更新

如果你使用 `yarn` 包管理器, 有 3 个新的专用 Gradle 设置, 可以通知你 `yarn.lock` 文件是否有更新. 如果你想要在 CI 构建过程中 `yarn.lock` 被更新时收到通知, 可以使用这些设定.

这 3 个新的 Gradle 属性是:

- `YarnLockMismatchReport`, 指示如何报告 `yarn.lock` 文件的变更. 可以使用以下设定值之一:
  - `FAIL` 让相应的 Gradle task 失败. 这是默认设定.
  - `WARNING` 将更新的相关信息写入 warning log.
  - `NONE` 禁用更新报告.
- `reportNewYarnLock`, 明确的报告最近创建的 `yarn.lock` 文件. 默认情况下, 这个选项会被禁用: 初次启动时生成新的 `yarn.lock` 文件是常见的做法. 你可以使用这个选项来确保这个文件被提交到你的代码仓库.
- `yarnLockAutoReplace`, 每次 Gradle task 运行时, 自动替换 `yarn.lock` 文件.

要使用这些选项, 请更新你的构建脚本文件 `build.gradle.kts`, 如下:

```
import
org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.
yarn.YarnPlugin::class.java) {
    rootProject.the<YarnRootExtension>().yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // 或 NONE | FAIL
    rootProject.the<YarnRootExtension>().reportNewYarnLock = false
// 或 true
    rootProject.the<YarnRootExtension>().yarnLockAutoReplace = false
// 或 true
}
```

通过 Gradle 属性添加用于浏览器的测试目标

从 Kotlin 1.8.0 开始, 你可以直接在 Gradle properties 文件中对不同的浏览器设置测试目标. 这样可以减少构建脚本文件的大小, 因为你不再需要在 `build.gradle.kts` 中编写所有的测试目标.

你可以使用这个属性对所有的模块定义浏览器列表, 然后在某些模块的构建脚本中添加特定的浏览器.

例如, 在你的 Gradle property 文件中, 以下代码将会对所有模块, 在 Firefox 和 Safari 中运行测试:

```
kotlin.js.browser.karma.browsers=firefox,safari
```

请参见 GitHub 代码中, 这个属性的所有可用值 (https://github.com/JetBrains/kotlin/blob/master/libraries/tools/kotlin-gradle-plugin/src/common/kotlin/org/jetbrains/kotlin/gradle/targets/js/testing/karma/KotlinKarma.kt#L106).

Kotlin 开发组非常感谢 Martynas Petuška (https://github.com/mpetuska) 实现了这个功能.

向你的项目添加 CSS 支持的新方式

这个发布版提供了一种新的方式来向你的项目添加 CSS 支持. 我们估计这个功能会影响到很多项目, 因此不要忘记更新你的 Gradle 构建脚本文件, 具体方法如下.

在 Kotlin 1.8.0 以前, 使用 `cssSupport.enabled` 属性来添加 CSS 支持:

```
browser {
    commonWebpackConfig {
        cssSupport.enabled = true
    }
}
```

现在, 你应该在 `cssSupport {}` 代码段中使用 `enabled.set()` 方法:

```
browser {
    commonWebpackConfig {
        cssSupport {
            enabled.set(true)
        }
    }
}
```

Gradle

Kotlin 1.8.0 **完全** 支持 Gradle 7.2 和 7.3. 你也可以使用 Gradle 的最新版本, 但如果你这样做, 请注意, 你可能会遇到 deprecation 警告, 或者 Gradle 的某些新功能可能不能工作.

这个版本带来了很多变更:

- 将 Kotlin 编译器选项导出为 Gradle 的 lazy 属性
- 提升了最低支持版本
- 可以禁用 Kotlin daemon 的 fallback 策略
- kotlin-stdlib 最新版本在传递依赖项中的使用
- 对相关的 Kotlin 和 Java 编译任务的 JVM 编译目标的兼容性是否相等的强制检查
- Kotlin Gradle plugin 的传递依赖项的解析
- 废弃与删除的功能

将 Kotlin 编译器选项导出为 Gradle 的 lazy 属性

为了将可用的 Kotlin 编译器选项导出为 Gradle 的 lazy 属性 (https://docs.gradle.org/current/userguide/lazy_configuration.html), 并更好的集成到 Kotlin

task 中, 我们进行了很多变更:

* 编译任务有了新的 compilerOptions 输入, 与已有的 kotlinOptions 类似, 但使用 Gradle 属性 API 的 Property (https://docs.gradle.org/current/javadoc/org/gradle/api/provider/Property.html) 作为返回类型:

```
tasks.named("compileKotlin",
org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile::class.java) {
    compilerOptions {
        useK2.set(true)
    }
}
```

* Kotlin 工具 task KotlinJsDce 和 KotlinNativeLink 有了新的 toolOptions 输入, 与已有的 kotlinOptions 输入类似.
* 新的输入带有 @Nested Gradle 注解 (https://docs.gradle.org/current/javadoc/org/gradle/api/tasks/Nested.html). 输入内的每个属性都有相关的 Gradle 注解, 例如 @Input 或 @Internal (https://docs.gradle.org/current/userguide/more_about_tasks.html#sec:up_to_date_checks).
* Kotlin Gradle plugin API artifact 有 2 个新的接口:
  * org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask, 它带有 compilerOptions 输入和 compileOptions() 方法. 所有的 Kotlin 编译 task 都实现这个接口.
  * org.jetbrains.kotlin.gradle.tasks.KotlinToolTask, 它带有 toolOptions 输入和 toolOptions() 方法. 所有的 Kotlin 工具 task – KotlinJsDce, KotlinNativeLink, 和 KotlinNativeLinkArtifactTask – 都实现这个接口.
* 有些 compilerOptions 使用新的类型, 而不是 String 类型:
  * JvmTarget (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JvmTarget.kt)
  * KotlinVersion (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-

types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/KotlinVersion.kt) (用于 apiVersion 和 languageVersion 输入)

- JsMainFunctionExecutionMode (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JsMainFunctionExecutionMode.kt)
- JsModuleKind (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JsModuleKind.kt)
- JsSourceMapEmbedMode (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JsSourceMapEmbedMode.kt)

例如, 你可以使用 compilerOptions.jvmTarget.set(JvmTarget.JVM_11), 而不是 kotlinOptions.jvmTarget = "11".

kotlinOptions 类型没有变更, 它会在内部转换为 compilerOptions 类型.

- Kotlin Gradle plugin API 与以前的发布版保持二进制兼容. 然而, 在 kotlin-gradle-plugin artifact 中存在一些源代码变更, 以及 ABI 不兼容的变更. 这些变更大多涉及到对某些内部类型的新增的泛型参数. 一个重要的变更是, KotlinNativeLink task 不再继承 AbstractKotlinNativeCompile task.
- KotlinJsCompilerOptions.outputFile 以及相关的 KotlinJsOptions.outputFile 选项已被废弃. 请改为使用 Kotlin2JsCompile.outputFileProperty task 输入.

Kotlin Gradle plugin 仍然会向 Android 扩展添加 KotlinJvmOptions DSL:

```
android {
    kotlinOptions {
        jvmTarget = "11"
    }
}
```

在 这个问题 (https://youtrack.jetbrains.com/issue/KT-15370/Gradle-DSL-add-module-level-kotlin-options) 中, 这个功能会被修改为, 将 compilerOptions DSL 添

加到模块级别.

限制

kotlinOptions task 输入和 kotlinOptions{...} task DSL 现在处于支持模式, 将会在未来的发布版中被废弃. 我们只会对 compilerOptions 和 toolOptions 进行功能改进.

对 kotlinOptions 调用任何 setter 或 getter, 会被代理到 compilerOptions 中的相关属性. 因此会造成以下限制:

- compilerOptions 和 kotlinOptions 在 task 的执行阶段不能修改 (有一种例外情况, 参见下面的章节).
- freeCompilerArgs 返回不可变的 List<String>, 也就是说, 比如, kotlinOptions.freeCompilerArgs.remove("something") 会失败.

有一些 plugin, 包括 kotlin-dsl, 以及启用了 Jetpack Compose (https://developer.android.com/jetpack/compose) 的 Android Gradle plugin (AGP), 会在 task 的执行阶段试图修改 freeCompilerArgs 属性. 在 Kotlin 1.8.0 中, 我们为它们添加了一个变通方法. 这个变通方法允许任何构建脚本或 plugin 在执行阶段修改 kotlinOptions.freeCompilerArgs, 但会在构建 log 中输出警告. 要禁用这个警告, 请使用新的 Gradle 属性 kotlin.options.suppressFreeCompilerArgsModificationWarning=true. Gradle 将会为 kotlin-dsl plugin (https://github.com/gradle/gradle/issues/22091) 和 启用了 Jetpack Compose 的 AGP (https://issuetracker.google.com/u/1/issues/247544167) 修正这个问题.

提升了最低支持版本

从 Kotlin 1.8.0 开始, 最低支持的 Gradle 版本是 6.8.3, 最低支持的 Android Gradle plugin 版本是 4.1.3.

详情请参见 Kotlin Gradle plugin 与可用的 Gradle 版本之间的兼容性 ("应用(Apply) Kotlin Gradle Plugin" in "配置 Gradle 项目")

可以禁用 Kotlin daemon 的 fallback 策略

有一个新的 Gradle 属性 kotlin.daemon.useFallbackStrategy, 默认值为 true. 当设定为 false 时, daemon 启动或通信时问题会导致构建失败. 在 Kotlin 编译 task 中还有一个新的 useDaemonFallbackStrategy 属性, 如果同时使用, 它的优先级会高于 Gradle 属性. 如果运行编译所需要的内存不足, 你会在 log 中看到相关信息.

Kotlin 编译器的 fallback 策略是, 如果 Kotlin daemon 因为某种原因失败, 那么会在 daemon 之外运行编译任务. 如果 Gradle daemon 已启动, 编译器会使用 "In process" 策略. 如果 Gradle daemon 没有启动, 编译器会使用 "Out of process" 策略. 详情请参见 执行策略的相关文档 ("定义 Kotlin 编译器执行策略" in "Kotlin Gradle plugin 中的编译与缓存"). 注意, 静默的 fallback 到其他策略, 会消耗大量的系统资源, 或导致不确定的构建结果; 关于这个问题, 详情请参见这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-48843/Add-ability-to-disable-Kotlin-daemon-fallback-strategy).

kotlin-stdlib 最新版本在传递依赖项中的使用

如果你在依赖项中将 Kotlin 版本明确指定为 1.8.0 或更高版本, 例如: `implementation("org.jetbrains.kotlin:kotlin-stdlib:1.8.0")`, 那么 Kotlin Gradle Plugin 对 `kotlin-stdlib-jdk7` 和 `kotlin-stdlib-jdk8` 传递依赖项会使用这个 Kotlin 版本. 这样做是为了避免不同的 stdlib 版本中出现重复的类 (详情请参见 `kotlin-stdlib-jdk7` 和 `kotlin-stdlib-jdk8` 合并到 `kotlin-stdlib`). 你可以使用 `kotlin.stdlib.jdk.variants.version.alignment` Gradle 属性禁用这个行为:

```
kotlin.stdlib.jdk.variants.version.alignment=false
```

如果你遇到与版本对齐相关的问题, 请使用 Kotlin BOM (https://docs.gradle.org/current/userguide/platforms.html#sub:bom_import) 对齐所有的版本, 方法是在你的构建脚本中声明一个 `kotlin-bom` 的平台依赖项:

```
implementation(platform("org.jetbrains.kotlin:kotlin-bom:1.8.0"))
```

关于其他情况, 以及我们建议的解决方案, 详情请参见 这篇文档 ("版本对齐的另一种方法" in "配置 Gradle 项目").

对相关的 Kotlin 和 Java 编译任务的 JVM 编译目标的强制检查

> 即使你的源代码文件全部都是 Kotlin, 并没有使用 Java, 本节仍然适用于你的 JVM 项目.

从这个发布版开始 (https://youtrack.jetbrains.com/issue/KT-54993/Raise-kotlin.jvm.target.validation.mode-check-default-level-to-error-when-build-is-running-on-Gradle-8), `kotlin.jvm.target.validation.mode` 属性 ("对相关联的编译任务检查 JVM 编译目标的兼容性" in "配置 Gradle 项目") 对 Gradle 8.0+ 项目 (Gradle 的这个版本还未正式发布) 的默认值是 `error`, 如果发现 JVM 编译目标不兼容, plugin 会让构建失败.

将默认值从 `warning` 提示到 `error` 是为了平滑的迁移到 Gradle 8.0 的预备步骤. **我们鼓励你将这个属性设置为 `error`**, 并 配置工具链 ("Gradle Java 工具链支持" in "配置 Gradle 项目"), 或者手动对

齐 JVM 版本.

详情请参见 如果你不检查编译目标的兼容性, 可能会导致什么样的错误 (["如果编译目标之间不兼容, 会发生什么问题" in "配置 Gradle 项目"]()).

Kotlin Gradle plugin 的传递依赖项的解析

在 Kotlin 1.7.0 中, 我们引入了 对 Gradle plugin 变体的支持 (["支持 Gradle plugin 变体(Variant)" in "Kotlin 1.7.0 版中的新功能"]()). 由于这些 plugin 变体的存在, 一个构建的 classpath 可以包含 Kotlin Gradle plugin (https://plugins.gradle.org/u/kotlin) 的不同版本, 并依赖到某些依赖项的不同版本, 通常是 `kotlin-gradle-plugin-api`. 这种情况可能导致依赖项解析的问题, 我们建议使用下面的变通方法, 下面以 `kotlin-dsl` plugin 为例.

Gradle 7.6 中的 `kotlin-dsl` plugin 依赖于 `org.jetbrains.kotlin.plugin.sam.with.receiver:1.7.10` plugin, 后一个 plugin 又依赖于 `kotlin-gradle-plugin-api:1.7.10`. 如果你添加 `org.jetbrains.kotlin.gradle.jvm:1.8.0` plugin, 这个 `kotlin-gradle-plugin-api:1.7.10` 的传递依赖项可能导致依赖项解析错误, 因为版本 (`1.8.0` 和 `1.7.10`) 与变体属性 `org.gradle.plugin.api-version` (https://docs.gradle.org/current/javadoc/org/gradle/api/attributes/plugin/GradlePluginApiVersion.html) 值不匹配. 变通方法是, 添加这个 constraint (https://docs.gradle.org/current/userguide/dependency_constraints.html#sec:adding-constraints-transitive-deps) 来对齐版本. 我们正在计划实现 Kotlin Gradle Plugin 库对齐平台 (https://youtrack.jetbrains.com/issue/KT-54691/Kotlin-Gradle-Plugin-libraries-alignment-platform), 在此之前, 可能一直需要使用这个变通方法:

```
dependencies {
    constraints {
        implementation("org.jetbrains.kotlin:kotlin-sam-with-
receiver:1.8.0")
    }
}
```

这个 constraint 对构建的 classpath 中的传递依赖项强制使用 `org.jetbrains.kotlin:kotlin-sam-with-receiver:1.8.0` 版本. 参见 Gradle issue tracker 中的一个类似情况 (https://github.com/gradle/gradle/issues/22510#issuecomment-1292259298).

废弃与删除的功能

在 Kotlin 1.8.0 中, 以下属性和方法的废弃周期继续向前推进:

- 在 Kotlin 1.7.0 的公告中 (["编译任务中的更新" in "Kotlin 1.7.0 版中的新功能"]()), `KotlinCompile` task 仍然有已废弃的 Kotlin 属性 `classpath`, 它将在未来的发布版中删除. 现在, 我们将

KotlinCompile task 的 classpath 属性的废弃级别修改为 error. 所有的编译任务使用 libraries 输入来指定编译所需要的库列表.

- 我们删除了 kapt.use.worker.api 属性, 它可以通过 Gradle Workers API 来运行 kapt (kapt 编译器插件). 从 Kotlin 1.3.70 开始, 默认情况下, kapt 使用 Gradle worker ("并行运行多个 KAPT 任务" in "kapt 编译器插件"), 我们建议使用这种方法.
- 在 Kotlin 1.7.0 中, 我们 宣布了 kotlin.compiler.execution.strategy 属性的废弃周期开始 ("废弃了系统属性 kotlin.compiler.execution.strategy" in "Kotlin 1.7.0 版中的新功能"). 在这个发布版中, 我们删除了这个属性. 详情请参见 如何使用其它方式定义 Kotlin 编译器执行策略 ("定义 Kotlin 编译器执行策略" in "Kotlin Gradle plugin 中的编译与缓存").

标准库

在 Kotlin 1.8.0 中:

- 更新了 JVM 编译目标.
- 一系列函数进入稳定版 – Java 与 Kotlin TimeUnit 之间的转换, cbrt(), Java Optionals 扩展函数.
- 提供了一个 可比较和可相减的 TimeMarks (预览版).
- 包含 java.nio.file.path 的扩展函数 (实验性功能).
- 提供了 kotlin-reflect 的性能改善.

更新了 JVM 编译目标

在 Kotlin 1.8.0 中, 标准库 (kotlin-stdlib, kotlin-reflect, 和 kotlin-script-*) 使用 JVM 1.8 编译. 以前的版本中, 标准库使用 JVM 1.6 编译.

Kotlin 1.8.0 不再支持 JVM 1.6 和 1.7 编译目标. 因此, 你不再需要在构建脚本中分布声明 kotlin-stdlib-jdk7 和 kotlin-stdlib-jdk8, 因为这些库文件的内容已经合并到了 kotlin-stdlib 之内.

> 如果在你的构建脚本中明确声明了 kotlin-stdlib-jdk7 和 kotlin-stdlib-jdk8 依赖项, 那么你应该将它们替换为 kotlin-stdlib.

注意, 混合使用 stdlib 库文件的不同版本可能导致类重复, 或类缺失. 为了避免这种问题, Kotlin Gradle plugin 可以帮助你 对齐 stdlib 版本.

cbrt()

`cbrt()` 函数, 可以计算一个 `double` 或 `float` 值的 real 三次方根, 现在进入稳定版.

```
import kotlin.math.*

fun main() {
    val num = 27
    val negNum = -num

    println("The cube root of ${num.toDouble()} is: " +
            cbrt(num.toDouble()))
    println("The cube root of ${negNum.toDouble()} is: " +
            cbrt(negNum.toDouble()))
}
```

Java 与 Kotlin TimeUnit 之间的转换

`kotlin.time` 中的 `toTimeUnit()` 和 `toDurationUnit()` 函数现在进入稳定版. 这些函数在 Kotlin 1.6.0 中作为实验性功能引入, 它们可以改进 Kotlin 与 Java 之间的交互能力. 现在你可以很容易在 Java `java.util.concurrent.TimeUnit` 和 Kotlin `kotlin.time.DurationUnit` 之间进行转换. 这些函数只能用于 JVM 平台.

```
import kotlin.time.*

// 供 Java 代码使用
fun wait(timeout: Long, unit: TimeUnit) {
    val duration: Duration =
timeout.toDuration(unit.toDurationUnit())
    ...
}
```

可比较和可相减的 TimeMarks

> ⚠ `TimeMarks` 的新功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"), 要使用这些功能, 你需要使用 `@OptIn(ExperimentalTime::class)` 或 `@ExperimentalTime` 进行使用者同意(Opt-in).

在 Kotlin 1.8.0 之前, 如果你想要计算多个 `TimeMarks` 和 **now** 之间的时间差, 你每次只能对一个 `TimeMark` 调用 `elapsedNow()`. 这个限制会造成比较结果时的困难, 因为两次 `elapsedNow()` 函数调用无法在完全相同的时刻执行.

为了解决这个问题, 在 Kotlin 1.8.0 中, 你可以对相同的时间源(Time Source) 相减和比较 `TimeMarks`. 现在你可以创建一个新的 `TimeMark` 实例来表达 **now**, 然后对它减去另一个 `TimeMarks`. 通过这种方式, 你从这些计算得到的结果可以保证是正确的.

```
import kotlin.time.*
fun main() {
//sampleStart
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    Thread.sleep(500) // 等待 0.5 秒
    val mark2 = timeSource.markNow()

    // 在 1.8.0 版以前
    repeat(4) { n ->
        val elapsed1 = mark1.elapsedNow()
        val elapsed2 = mark2.elapsedNow()

        // 根据两次 elapsedNow() 调用之间经过了多长时间不同
        // elapsed1 和 elapsed2 之间的差别可能发生变化
        println("Measurement 1.${n + 1}: elapsed1=$elapsed1, " +
                "elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    println()

    // 从 1.8.0 开始
    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        // 现在相对于 mark3 来计算经过的时间,
        // 而 mark3 是固定的值
        println("Measurement 2.${n + 1}: elapsed1=$elapsed1, " +
                "elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
```

```
    // 也可以对 time marks 进行比较
    // 这里的输出结果为 true, 因为 mark2 的捕获是在 mark1 之后
    println(mark2 > mark1)
//sampleEnd
}
```

这个新功能在动画计算中非常有用, 这种情况下, 你需要对代表不同的帧的多个 TimeMarks 计算它们之间的时间差, 或比较先后.

递归的复制或删除目录

java.nio.file.path 的这些新函数是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 要使用它们, 你需要使用 @OptIn(kotlin.io.path.ExperimentalPathApi::class) 或 @kotlin.io.path.ExperimentalPathApi 进行使用者同意(Opt-in). 或者, 你也可以使用编译器选项 -opt-in=kotlin.io.path.ExperimentalPathApi.

我们为 java.nio.file.Path 引入了 2 个新的扩展函数, copyToRecursively() 和 deleteRecursively(), 它们可以:

- 将一个目录以及其中的内容, 递归的复制到另一个目标目录.
- 递归的删除一个目录以及其中的内容.

要实现文件备份功能, 这些函数会非常有用.

错误处理

使用 copyToRecursively() 函数时, 你可以覆盖 onError lambda 函数, 来定义在复制过程中发生异常时, 应该如何处理:

```
sourceRoot.copyToRecursively(destinationRoot, followLinks = false,
    onError = { source, target, exception ->
        logger.logError(exception, "Failed to copy $source to
$target")
        OnErrorResult.TERMINATE
    })
```

当你使用 deleteRecursively() 时, 如果在删除一个文件或目录时发生异常, 那么这个文件或目录会被跳过. 删除过程结束后, deleteRecursively() 会抛出 IOException, 其中包含删除过程中发生的

所有异常.

文件覆盖

如果 `copyToRecursively()` 发现一个文件在目标目录中已经存在, 那么会发生异常. 如果你想要覆盖文件, 请使用这个函数带有 `overwrite` 参数的重载版本, 并将这个参数设置为 `true`:

```
fun setUpEnvironment(projectDirectory: Path, fixtureName: String) {
    fixturesRoot.resolve(COMMON_FIXTURE_NAME)
        .copyToRecursively(projectDirectory, followLinks = false)
    fixturesRoot.resolve(fixtureName)
        .copyToRecursively(projectDirectory, followLinks = false,
            overwrite = true) // 覆盖 common fixture 中相同的内容
}
```

自定义复制行为

要定义你自己的复制逻辑, 请使用这个函数的带有额外参数 `copyAction` 的覆盖版本. 使用 `copyAction`, 你可以提供一个 lambda 函数, 指定你想要的复制动作:

```
sourceRoot.copyToRecursively(destinationRoot, followLinks = false) {
source, target ->
    if (source.name.startsWith(".")) {
        CopyActionResult.SKIP_SUBTREE
    } else {
        source.copyToIgnoringExistingDirectory(target, followLinks =
false)
        CopyActionResult.CONTINUE
    }
}
```

关于这些扩展函数, 更多详情请参见 我们的 API 参考文档 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io.path/java.nio.file.-path/copy-to-recursively.html).

Java Optionals 扩展函数

在 Kotlin 1.7.0 ("对 Java Optionals 的新的扩展函数(实验性功能)" in "Kotlin 1.7.0 版中的新功能") 中引入的扩展函数现在进入稳定版. 这些函数简化了 Java 中的 Optional 类的使用. 可以用来在 JVM 中解包或变换 `Optional` 对象, 使得使用 Java API 时代码更加简洁. 更多详情请参见, Kotlin

1.7.0 版中的新功能 ("对 Java Optionals 的新的扩展函数(实验性功能)" in "Kotlin 1.7.0 版中的新功能").

kotlin-reflect 的性能改善

由于 `kotlin-reflect` 现在使用 JVM 1.8 进行编译, 因此我们将内部的缓存机制迁移到了 Java 的 `ClassValue`. 以前我们只缓存 `KClass`, 现在我们还可以缓存 `KType` 和 `KDeclarationContainer`. 这些变更带来了调用 `typeOf()` 时的显著的性能改善.

文档更新

Kotlin 文档有了很大的变更:

文档的改进和新增

- Gradle 概述 (Gradle) – 学习如何使用 Gradle 构建系统配置和构建一个 Kotlin 项目, 可用的编译器选项, 编译, 以及 Kotlin Gradle plugin 中的缓存.
- Java 和 Kotlin 中的可空性(Nullability) (Java 和 Kotlin 中的可空性(Nullability)) – 学习 Java 和 Kotlin 处理可空变量方式的的差异.
- Lincheck 指南 (Lincheck 指南) – 学习如何设置和使用 Lincheck 框架, 在 JVM 平台上测试并发算法.

教程的改进和新增

- Gradle 与 Kotlin/JVM 入门 (Gradle 与 Kotlin/JVM 入门) – 使用 IntelliJ IDEA 和 Gradle 创建一个控制台应用程序.
- 使用 Ktor 和 SQLDelight 创建跨平台应用程序 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ktor-sqldelight.html) – 使用 Kotlin Multiplatform Mobile, 创建一个运行于 iOS 和 Android 的移动应用程序.
- Kotlin Multiplatform 入门 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html) – 学习使用 Kotlin 进行跨平台移动应用程序开发, 并创建一个可以同时运行于 Android 和 iOS 平台的应用程序.

安装 Kotlin 1.8.0

IntelliJ IDEA (https://www.jetbrains.com/idea/download/) 2021.3, 2022.1, 和 2022.2 会自动建议将 Kotlin plugin 更新到版本 1.8.0. IntelliJ IDEA 2022.3 的后续小版本更新, 会带有 Kotlin plugin 的 1.8.0 版本.

> 要在 IntelliJ IDEA 2022.3 中将已有的项目迁移到 Kotlin 1.8.0, 请将 Kotlin 版本变更为 `1.8.0`, 然后重新导入你的 Gradle 或 Maven 项目.

对于 Android Studio Electric Eel (221) 和 Flamingo (222), Android Studios 的后续更新会带有 Kotlin plugin 的 1.8.0 版本. 新的命令行编译器可以通过 GitHub 发布页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.8.0) 下载.

Kotlin 1.8.0 的兼容性指南

Kotlin 1.8.0 是一个 功能性发布版(Feature Release) ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化"), 因此可能带来一些变更, 与你针对旧版本编写的代码不兼容. 关于这些不兼容的变更, 详情请参见 Kotlin 1.8.0 兼容性指南 (Kotlin 1.8 兼容性指南).

Kotlin 1.7.20 版中的新功能

> IDE 从 IntelliJ IDEA 2021.3, 2022.1, 和 2022.2 开始支持 Kotlin 1.7.20.

最终更新: 2024/09/10

发布日期: 2022/09/29 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.7.20 已经发布了! 以下是它的一些重要功能:

- 新的 Kotlin K2 编译器支持 `all-open`, SAM with receiver, Lombok, 以及其它编译器 plugin
- 我们引入了 `..<` 操作符的预览版, 用于创建终止端开放的值范围(open-ended range)
- 默认启用新的 Kotlin/Native 内存管理器
- 我们为 JVM 引入了一个新的实验性功能: 使用泛型类型的内联类

关于这个版本的变更概要, 请参见以下视频:

对 Kotlin K2 编译器 plugin 的支持

Kotlin 开发组还在继续稳定 K2 编译器. K2 仍然在 **Alpha** 阶段 (如同 Kotlin 1.7.0 发布版中宣布 ("JVM 平台的新的 Kotlin K2 编译器 (Alpha 版)" in "Kotlin 1.7.0 版中的新功能") 的那样), 但现在它支持几种编译器 plugin. 你可以关注 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-52604), 从 Kotlin 开发组得到新编译器的最新信息.

从 1.7.20 版开始, Kotlin K2 编译器支持以下 plugin:

- `all-open` (All-open 编译器插件)
- `no-arg` (No-arg 编译器插件)
- SAM with receiver (SAM-with-receiver 编译器插件)
- Lombok (Lombok 编译器插件)
- AtomicFU
- `jvm-abi-gen`

> ⚠ 新的 K2 编译器的 Alpha 版只能用于 JVM 项目. 不支持 Kotlin/JS, Kotlin/Native, 或其他跨平台项目.

关于新的编译器以及它的益处, 请观看以下视频:

- 通往新 Kotlin 编译器之路 (https://www.youtube.com/watch?v=iTdJJq_LyoY)
- K2 编译器: 概要介绍 (https://www.youtube.com/watch?v=db19VFLZqJM)

如何启用 Kotlin K2 编译器

要启用并测试 Kotlin K2 编译器, 请使用以下编译器选项:

```
-Xuse-k2
```

你可以在你的 `build.gradle(.kts)` 文件中指定这个选项:

Kotlin

```kotlin
tasks.withType<KotlinCompile> {
    kotlinOptions.useK2 = true
}
```

Groovy

```groovy
compileKotlin {
    kotlinOptions.useK2 = true
}
```

你可以在你的 JVM 项目中查看性能提升, 并与旧编译器的性能进行比较.

留下你对于新 K2 编译器的反馈意见

我们非常感谢你任何形式的反馈意见:

- 在 Kotlin Slack 中直接向 K2 开发者提供你的反馈意见: 得到邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up?_gl=1*ju6cbn*_ga*MTA3MTk5NDkzMC4xNjQ2MDY3MDU4*_ga_9J976DJZ68*MTY1ODMzNzA3OS4xMDAuMS4xNjU4MzQwODEwLjYw), 并加入 #k2-early-adopters (https://kotlinlang.slack.com/archives/C03PK0PE257) 频道.

- 如果在使用新 K2 编译器时遇到的任何问题, 请向 我们的问题追踪系统 (https://kotl.in/issue) 提交报告.

- 开启 **Send usage statistics** 选项 (https://www.jetbrains.com/help/idea/settings-usage-statistics.html), 允许 JetBrains 收集关于 K2 使用情况的匿名统计数据.

语言功能

Kotlin 1.7.20 引入了一些新的语言功能特性的预览版, 并对构建器类型推断增加了一些限制:

- ..< 操作符的预览版, 用于创建终止端开放的值范围(open-ended range)
- 新的 data object 声明
- 构建器类型推断的限制

..< 操作符的预览版, 用于创建终止端开放的值范围(open-ended range)

> ⚠ 这个新操作符是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"), 在 IDE 中只有非常有限的支持.

这个发布版引入了新的 `..<` 操作符. Kotlin 已经有了 `..` 操作符来表达一个值范围. 新的 `..<` 操作符与 `until` 函数类似, 帮助你定义终止端开放的值范围.

我们的研究显示, 这个新操作符更适合表示终止端开放的值范围, 更清楚的表示值范围的上界没有包含在内.

下面是在 `when` 表达式中使用 `..<` 操作符的示例:

```
when (value) {
    in 0.0..<0.25 -> // 第 1 个 1/4
    in 0.25..<0.5 -> // 第 2 个 1/4
    in 0.5..<0.75 -> // 第 3 个 1/4
```

```
    in 0.75..1.0 ->  // 最后 1 个 1/4  <- 注意这里是封闭的值范围
}
```

标准库 API 的变更

在共通的 Kotlin 标准库的 `kotlin.ranges` 包中, 将会引入以下新的类型和操作:

新的 OpenEndRange<T> 接口

用于表达终止端开放的值范围的新接口与已有的 `ClosedRange<T>` 接口非常类似:

```
interface OpenEndRange<T : Comparable<T>> {
    // 值范围的下界
    val start: T
    // 值范围的上界, 不包含在值范围内
    val endExclusive: T
    operator fun contains(value: T): Boolean = value >= start &&
value < endExclusive
    fun isEmpty(): Boolean = start >= endExclusive
}
```

在既有的可遍历的值范围中实现 OpenEndRange

现在, 如果开发者需要得到一个带开放的上界的值范围, 他们可以使用相同的值, 通过 `until` 函数等效的产生一个封闭的可遍历的值范围. 为了让这样的值范围能够用于接受 `OpenEndRange<T>` 的新 API, 我们希望在既有的可遍历的值范围中实现这个接口, 包括: `IntRange`, `LongRange`, `CharRange`, `UIntRange`, 和 `ULongRange`. 这样它们就能同时实现 `ClosedRange<T>` 和 `OpenEndRange<T>` 接口.

```
class IntRange : IntProgression(...), ClosedRange<Int>,
OpenEndRange<Int> {
    override val start: Int
    override val endInclusive: Int
    override val endExclusive: Int
}
```

用于标准类型的 rangeUntil 操作符

对于目前由 `rangeTo` 操作符定义的类型及其组合, 还会提供 `rangeUntil` 操作符. 作为原型, 我们以扩展函数的形式提供这些操作符, 但为了保持一致性, 在终止端开放的值范围 API 的稳定版发布之前, 我们计划让它们成为类的成员.

如何启用 ..< 操作符

要使用 `..<` 操作符, 或为你自己的类型实现这个操作符, 你需要启用 `-language-version 1.8` 编译器选项.

为支持标准类型的终止端开放值范围, 引入了新的 API 元素, 和通常的实验性标准库 API 一样, 这些元素需要使用者明确同意(opt-in): `@OptIn(ExperimentalStdlibApi::class)`. 或者, 你也可以使用编译器选项: `-opt-in=kotlin.ExperimentalStdlibApi`.

关于这个新操作符, 详情请参见这个 KEEP 文档 (https://github.com/kotlin/KEEP/blob/open-ended-ranges/proposals/open-ended-ranges.md).

对单子(Singleton)与带 data object 的封闭类层级结构(Sealed Class Hierarchy), 改善了它们的字符串表示

Data object 是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"), 目前在 IDE 中只有非常有限的支持.

这个发布版引入了新类型的 `object` 声明供你使用: `data object`. Data object (https://youtrack.jetbrains.com/issue/KT-4107) 的行为与通常的 `object` 声明相同, 但默认带有更加良好格式化的 `toString` 表示.

```
package org.example
object MyObject
data object MyDataObject

fun main() {
    println(MyObject) // 输出结果是 org.example.MyObject@1f32e575
    println(MyDataObject) // 输出结果是 MyDataObject
}
```

因此封闭类层级结构(Sealed Class Hierarchy)很适合使用 `data object` 声明, 你可以和其他 `data class` 声明一起使用. 在下面的代码中, 我们将 `EndOfFile` 声明为 `data object`, 而不是单纯的 `object`, 因此它将带有良好格式化的 `toString`, 而不必手动编写这个函数, 同时又保持这个对象与其他 `data class` 定义的一致性:

```
sealed class ReadResult {
    data class Number(val value: Int) : ReadResult()
    data class Text(val value: String) : ReadResult()
    data object EndOfFile : ReadResult()
}

fun main() {
    println(ReadResult.Number(1)) // 输出结果是 Number(value=1)
    println(ReadResult.Text("Foo")) // 输出结果是 Text(value=Foo)
    println(ReadResult.EndOfFile) // 输出结果是 EndOfFile
}
```

如何启用 data object

要在你的代码中使用 data object 声明, 请启用 `-language-version 1.9` 编译器选项. 在 Gradle 项目中, 你可以在你的 `build.gradle(.kts)` 文件中添加以下代码:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile>
().configureEach {
    // ...
    kotlinOptions.languageVersion = "1.9"
}
```

Groovy

```
compileKotlin {
    // ...
    kotlinOptions.languageVersion = '1.9'
}
```

关于 data object 的更多信息, 请参见 KEEP 文档 (https://github.com/Kotlin/KEEP/pull/316), 对于它们的实现, 也可以在这里提供你的反馈意见.

构建器类型推断的新限制

Kotlin 1.7.20 对 构建器类型推断 (通过构建器类型推断(Builder Type Inference)使用构建器) 功能添加了一些重要的限制, 可能会影响你的代码. 这些限制影响包含构建器 lambda 函数的代码, 在这些代码中, 不对 lambda 函数本身进行分析就无法判断参数类型. 而参数需要被用作类型参数. 现在, 编译器会对这样的代码一律报告错误, 要求你明确指定参数类型.

这是一个不兼容的变更, 但我们的研究演示, 这样的情况非常少见, 而且这样的限制通常不会影响你的代码. 如果有影响, 请参考下面的情况:

- 构建器推断包含扩展函数, 隐藏了同名的成员函数.

  如果你的代码包含扩展函数, 它的名称与在构建器推断中使用的名称相同, 那么编译器会提示错误:

```
class Data {
    fun doSmth() {} // 1
}

fun <T> T.doSmth() {} // 2

fun test() {
    buildList {
        this.add(Data())
        this.get(0).doSmth() // 这里解析的结果是函数 2 , 并导致错误
    }
}
```

  要修正这段代码, 你需要明确指定类型:

```
class Data {
    fun doSmth() {} // 1
}

fun <T> T.doSmth() {} // 2

fun test() {
    buildList<Data> { // 使用类型参数!
        this.add(Data())
        this.get(0).doSmth() // 这里解析的结果是函数 1
```

```
    }
}
```

- 构建器推断使用多个 lambda 函数, 而且没有明确指定类型参数.

  如果在构建器推断中存在 2 个或以上的 lambda 代码段, 它们会影响到类型. 为了避免错误, 编译器会要求你明确指定类型:

```
fun <T: Any> buildList(
    first: MutableList<T>.() -> Unit,
    second: MutableList<T>.() -> Unit
): List<T> {
    val list = mutableListOf<T>()
    list.first()
    list.second()
    return list
}

fun main() {
    buildList(
        first = { // this 的类型是: MutableList<String>
            add("")
        },
        second = { // this 的类型是: MutableList<Int>
            val i: Int = get(0)
            println(i)
        }
    )
}
```

  要修正这个错误, 你需要明确指定类型, 修正类型不匹配的问题:

```
fun main() {
    buildList<Int>(
        first = { // this 的类型是: MutableList<Int>
            add(0)
        },
        second = { // this 的类型是: MutableList<Int>
            val i: Int = get(0)
```

```
            println(i)
        }
    )
}
```

如果你遇到了以上情况以外的错误, 请向我们的团队 提交一个 issue (https://kotl.in/issue).

关于构建器推断的这次更新, 详情请参见这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-53797).

Kotlin/JVM

Kotlin 1.7.20 引入了泛型的内联类(Generic Inline Class), 对委托属性增加了更多的字节码优化, 还在 kapt stub 生成 task 中支持 IR, 因此可以在 kapt 中使用 Kotlin 的所有最新功能:

- 泛型的内联类(Generic Inline Class)
- 对委托属性的更多优化
- 在 kapt stub 生成 task 中支持 JVM IR 后端

泛型的内联类(Generic Inline Class)

> ⚠ 泛型的内联类是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文), 而且你应该只为评估的目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-52994) 提供你的反馈意见.

Kotlin 1.7.20 允许 JVM 内联类使用类型参数作为它的内部数据的类型. 编译器会将它映射为 `Any?`, 或者, 一般来说, 映射为类型参数的上界.

请参考下面的示例:

```
@JvmInline
value class UserId<T>(val value: T)

fun compute(s: UserId<String>) {} // 编译器生成 fun compute-<hashcode>
(s: Any?)
```

函数接受内联类作为参数. 参数会被映射为类型参数的上界, 而不是类型参数.

要启用这个功能, 请使用 `-language-version 1.8` 编译器选项.

欢迎你通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-52994) 提供你的反馈意见.

对委托属性的更多优化

在 Kotlin 1.6.0 中, 我们优化了委托到一个属性的情况, 具体做法是, 省略域变量 `$delegate`, 并 生成对被引用的属性的直接访问 ("代理属性优化, 不再在 KProperty 实例上调用 get/set 方法" in "Kotlin 1.6.0 版中的新功能"). 在 1.7.20 中, 我们对更多情况实现了这样的优化. 如果委托是以下情况, 现在也会省略域变量 `$delegate`:

- 命名对象:

  ```
  object NamedObject {
      operator fun getValue(thisRef: Any?, property: KProperty<*>):
  String = ...
  }

  val s: String by NamedObject
  ```

- 同一个模块内, 带有 后端域变量 ("属性的后端域变量(Backing Field)" in "属性(Property)") 和默认 getter 的 final `val` 属性:

  ```
  val impl: ReadOnlyProperty<Any?, String> = ...

  class A {
      val s: String by impl
  }
  ```

- 常数表达式, 枚举值(Enum Entry), `this`, 或 `null`. 下面是 `this` 的例子:

  ```
  class A {
      operator fun getValue(thisRef: Any?, property: KProperty<*>)
  ...

      val s by this
  }
  ```

详情请参见 委托属性 (委托属性).

欢迎你通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-23397) 提供你的反馈意见.

在 kapt stub 生成 task 中支持 JVM IR 后端

> 在 kapt stub 生成 task 中支持 JVM IR 后端是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能.

在 1.7.20 以前, kapt stub 生成 task 使用旧的编译器后端, 并且 kapt (kapt 编译器插件) 无法处理可重复的注解 ("可重复注解" in "注解"). 在 Kotlin 1.7.20 中, 我们添加了在 kapt stub 生成 task 中对 JVM IR 后端 ("JVM IR 后端的稳定版" in "Kotlin 1.5.0 版中的新功能") 的支持. 因此在 kapt 中可以使用 Kotlin 的所有新功能, 包括可重复的注解.

要在 kapt 中使用 IR 后端, 请在你的 `gradle.properties` 文件中添加以下选项:

```
kapt.use.jvm.ir=true
```

欢迎你通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-49682) 提供你的反馈意见.

Kotlin/Native

Kotlin 1.7.20 开始默认使用新的 Kotlin/Native 内存管理器, 并提供了选项来定制 `Info.plist` 文件:

- 默认使用新的内存管理器
- 定制 Info.plist 文件

默认启用新的 Kotlin/Native 内存管理器

这个发布版中, 改进了新内存管理器的稳定性, 并改善了性能, 因此我们将新内存管理器提升到 Beta 版 (Kotlin 各部分组件的稳定性).

以前的内存管理器使得编写并发和异步代码比较复杂, 包括实现 `kotlinx.coroutines` 库时的问题. 这些问题阻碍了 Kotlin Multiplatform Mobile 的应用, 因为同步的限制, 导致在 iOS 和 Android 平台共用 Kotlin 代码会发生问题. 新的内存管理器终于为 Kotlin Multiplatform Mobile 提升到 Beta 版

(https://blog.jetbrains.com/kotlin/2022/05/kotlin-multiplatform-mobile-beta-roadmap-update/) 铺好了道路.

新的内存管理器还支持编译器缓存, 使得编译时间能够与以前的版本媲美. 关于新内存管理器的更多益处, 请参见我们关于预览版的 Blog 文章 (https://blog.jetbrains.com/kotlin/2021/08/try-the-new-kotlin-native-memory-manager-development-preview/). 关于更多技术细节, 请参见这篇文档 (Kotlin/Native 内存管理).

配置与设置

从 Kotlin 1.7.20 开始, 默认使用新的内存管理器. 不需要额外的设置.

如果你已经手动启用了它, 你可以从你的 `gradle.properties` 文件删除 `kotlin.native.binary.memoryModel=experimental` 选项, 或从 `build.gradle(.kts)` 文件删除 `binaryOptions["memoryModel"] = "experimental"`.

如果需要, 你可以在你的 `gradle.properties` 文件中使用 `kotlin.native.binary.memoryModel=strict` 选项, 切换回原来的内存管理器. 但是, 对于原来的内存管理器, 编译器缓存支持就不再可用了, 因此编译时间会恶化.

冻结(Freezing)

在新的内存管理器中, 冻结(Freezing)已被废弃. 请不要使用它, 除非你需要你的代码在原来的内存管理器中工作 (旧内存管理器中继续需要冻结). 对于需要继续支持原来的内存管理器的库开发者, 或开发者在使用新的内存管理器时遇到问题, 想要退回到旧内存管理器的情况, 这个功能可能有帮助.

这种情况下, 你可以临时性的同时支持新的和原来的内存管理器. 要忽略废弃导致的编译警告, 请执行以下步骤中的某一个:

- 在使用废弃的 API 的地方, 标注 `@OptIn(FreezingIsDeprecated::class)` 注解.
- 在 Gradle 中对所有的 Kotlin 源代码集使用 `languageSettings.optIn("kotlin.native.FreezingIsDeprecated")`.
- 传递编译器选项 `-opt-in=kotlin.native.FreezingIsDeprecated`.

在 Swift/Objective-C 中调用 Kotlin suspending 函数

对于从 Swift 和 Objective-C 的主线程以外的线程调用 Kotlin `suspend` 函数的情况, 新的内存管理器还存在限制, 但你可以使用一个新的 Gradle 选项来解决.

这个限制最初是在原来的内存管理器中引入的, 针对代码将自己的后续代码派发在原来的线程中恢复执行的情况. 如果这个线程没有一个支持的事件循环, 这个任务就永远不会执行, 因此协程永远不会恢复执行.

在某些情况下, 可以不再需要这个限制, 但对所有必要条件的检查很难实现. 由于这个原因, 我们决定在新的内存管理器中继续保留这个限制, 同时引入一个选项, 允许你关闭这个限制. 要关闭它, 请向你的 `gradle.properties` 文件添加以下选项:

```
kotlin.native.binary.objcExportSuspendFunctionLaunchThreadRestriction=none
```

> ⚠ 如果你使用 `kotlinx.coroutines` 的 `native-mt` 版本, 或采用了相同的 "dispatch to the original thread" 方案的其他库, 请不要添加这个选项.

Kotlin 开发组非常感谢 Ahmed El-Helw (https://github.com/ahmedre) 实现了这个选项.

留下你的反馈意见

对我们的生态系统来说, 这是一个重大的变更. 如果你能够留下反馈意见, 帮助继续改善它, 我们将会非常感谢.

请在你的项目中试用新的内存管理器, 并 在我们的问题追踪系统 YouTrack 中留下你的反馈意见 (https://youtrack.jetbrains.com/issue/KT-48525).

定制 Info.plist 文件

生成框架时, Kotlin/Native 编译器会生成信息属性列表文件, `Info.plist`. 在以前的版本中, 定制这个文件的内容会很麻烦. 从 Kotlin 1.7.20开始, 你可以直接设置以下属性:

属性	二进制选项
`CFBundleIdentifier`	`bundleId`
`CFBundleShortVersionString`	`bundleShortVersionString`
`CFBundleVersion`	`bundleVersion`

要设置这些属性, 请使用对于的二进制选项. 可以指定编译器选项 `-Xbinary=$option=$value`, 或对需要的框架设置 `binaryOption(option, value)` Gradle DSL.

Kotlin 开发组非常感谢 Mads Ager 实现了这个功能.

Kotlin/JS

Kotlin/JS 有了一些功能增强, 改进了开发者体验, 并提升了性能:

- 由于依赖项装载的性能改进, 在增量构建和完全构建中, Klib 的生成都更加快速了.
- 重新实现了 对开发阶段二进制文件的增量编译 ("对开发阶段二进制文件进行增量编译" in "使用 IR 编译器") 功能, 实现了完全构建时的很大改进, 更快的增量构建, 以及稳定性提升.
- 我们对内嵌对象, 封闭类, 以及构造器中的可选参数, 改进了 `.d.ts` 文件的生成.

Gradle

Kotlin Gradle plugin 的更新主要是兼容新的 Gradle 功能和最新的 Gradle 版本.

Kotlin 1.7.20 包含的变更是支持 Gradle 7.1. 删除或替换了已废弃的方法和属性, 减少了由 Kotlin Gradle plugin 造成的废弃警告的数量, 而且有助于将来支持 Gradle 8.0.

但是, 存在一些潜在的不兼容的变更, 需要你注意:

编译目标的配置

- `org.jetbrains.kotlin.gradle.dsl.SingleTargetExtension` 现在有一个泛型参数, `SingleTargetExtension<T : KotlinTarget>`.
- `kotlin.targets.fromPreset()` convention 已被废弃. 作为代替, 你可以继续使用 `kotlin.targets { fromPreset() }` 方案, 但我们推荐使用更加 专门的方法来创建编译目标 (为 Kotlin Multiplatform 设置编译目标).
- 在 `kotlin.targets { }` 代码段内, 由 Gradle 自动生成的编译目标访问器不再可用. 请改为使用 `findByName("targetName")` 方法.

  注意, 对 `kotlin.targets` 的情况, 这些访问器仍然可以使用, 比如对 `kotlin.targets.linuxX64`.

源代码目录的配置

Kotlin Gradle plugin 对 Java `SourceSet` 组添加了 Kotlin `SourceDirectorySet`, 作为一个 `kotlin` 扩展. 因此可以在 `build.gradle.kts` 文件内, 以类似于对 Java, Groovy, 和 Scala (https://docs.gradle.org/7.1/release-notes.html#easier-source-set-configuration-in-kotlin-dsl) 的方式来配置源代码目录:

```
sourceSets {
    main {
        kotlin {
```

```
            java.setSrcDirs(listOf("src/java"))
            kotlin.setSrcDirs(listOf("src/kotlin"))
        }
    }
}
```

你不再需要使用已废弃的 Gradle 方式来为 Kotlin 指定源代码目录.

记住, 你还可以使用 `kotlin` 扩展来访问 `KotlinSourceSet`:

```
kotlin {
    sourceSets {
        main {
            // ...
        }
    }
}
```

JVM toolchain 配置的新方法

这个发布版提供了一个新的 `jvmToolchain()` 方法, 用来启用 JVM toolchain 功能 ("Gradle Java 工具链支持" in "配置 Gradle 项目"). 如果你不需要任何额外的 配置设定 (https://docs.gradle.org/current/javadoc/org/gradle/jvm/toolchain/JavaToolchainSpec.html), 比如 `implementation` 或 `vendor`, 你可以通过 Kotlin 扩展使用这个方法:

```
kotlin {
    jvmToolchain(17)
}
```

这样可以简化 Kotlin 项目的设置过程, 无需添加额外的配置. 在这次的发布版之前, 你只能通过以下方式指定 JDK 版本:

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(17))
    }
}
```

标准库

Kotlin 1.7.20 对 `java.nio.file.Path` 类提供了新的 扩展函数 ("扩展函数(Extension Function)" in "扩展"), 可以用来遍历文件树:

- `walk()` 惰性的(lazily)遍历以指定路径为根的文件树.
- `fileVisitor()` 可以单独创建一个 `FileVisitor`. `FileVisitor` 定义遍历目录和文件时的行为.
- `visitFileTree(fileVisitor: FileVisitor, ...)` 接收一个预先定义的 `FileVisitor`, 然后使用 `java.nio.file.Files.walkFileTree()` 来遍历文件树.
- `visitFileTree(..., builderAction: FileVisitorBuilder.() -> Unit)` 使用 `builderAction` 创建一个 `FileVisitor`, 然后调用 `visitFileTree(fileVisitor, ...)` 函数.
- `FileVisitResult` 是 `FileVisitor` 的返回类型, 默认值是 `CONTINUE`, 表示继续文件遍历过程.

> ⚠ `java.nio.file.Path` 的这些新扩展函数是 实验性功能 (Kotlin 各部分组件的稳定性). 随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能.

下面是使用这些新扩展函数能够实现的一些功能:

- 明确创建一个 `FileVisitor`, 然后使用它:

```
val cleanVisitor = fileVisitor {
    onPreVisitDirectory { directory, attributes ->
        // 这里可以实现访问目录时的某些逻辑
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // 这里可以实现访问文件时的某些逻辑
        FileVisitResult.CONTINUE
    }
}

// 这里可以实现某些逻辑
```

```
projectDirectory.visitFileTree(cleanVisitor)
```

- 使用 builderAction 创建一个 FileVisitor, 然后立即使用它:

```
projectDirectory.visitFileTree {
    // builderAction 的定义:
    onPreVisitDirectory { directory, attributes ->
        // 这里可以实现访问目录时的某些逻辑
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // 这里可以实现访问文件时的某些逻辑
        FileVisitResult.CONTINUE
    }
}
```

- 使用 walk() 函数, 遍历以指定的路径为根的文件树:

```
@OptIn(kotlin.io.path.ExperimentalPathApi::class)
fun traverseFileTree() {
    val cleanVisitor = fileVisitor {
        onPreVisitDirectory { directory, _ ->
            if (directory.name == "build") {
                directory.toFile().deleteRecursively()
                FileVisitResult.SKIP_SUBTREE
            } else {
                FileVisitResult.CONTINUE
            }
        }

        onVisitFile { file, _ ->
            if (file.extension == "class") {
                file.deleteExisting()
            }
            FileVisitResult.CONTINUE
        }
    }
```

```
    val rootDirectory = createTempDirectory("Project")

    rootDirectory.resolve("src").let { srcDirectory ->
        srcDirectory.createDirectory()
        srcDirectory.resolve("A.kt").createFile()
        srcDirectory.resolve("A.class").createFile()
    }

    rootDirectory.resolve("build").let { buildDirectory ->
        buildDirectory.createDirectory()
        buildDirectory.resolve("Project.jar").createFile()
    }


    // 使用 walk 函数:
    val directoryStructure =
rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    assertPrints(directoryStructure, "[, build, build/Project.jar,
src, src/A.class, src/A.kt]")

    rootDirectory.visitFileTree(cleanVisitor)

    val directoryStructureAfterClean =
rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    assertPrints(directoryStructureAfterClean, "[, src,
src/A.kt]")
}
```

和其他的实验性 API 一样, 这些新扩展函数需要使用者同意(Opt-in): `@OptIn(kotlin.io.path.ExperimentalPathApi::class)` 或 `@kotlin.io.path.ExperimentalPathApi`. 或者, 你可以使用编译器选项: `-opt-in=kotlin.io.path.ExperimentalPathApi`.

对于 `walk()` 函数 (https://youtrack.jetbrains.com/issue/KT-52909) 和 visit 扩展函数

(https://youtrack.jetbrains.com/issue/KT-52910), 我们期待你能通过 YouTrack 提供返回意见.

文档更新

从上一次发布之后, Kotlin 文档有了很大的变更:

文档的改进和新增

- 基本类型概述 (基本类型) – 学习 Kotlin 中使用的基本类型: 数值, Booleans, 字符, 字符串, 数组, 以及无符号整数.
- Kotlin 开发使用的 IDE (支持 Kotlin 开发的 IDE) – 查看带有官方 Kotlin 支持的 IDE, 以及带有社区支持的 plugin 的工具.

Kotlin Multiplatform 期刊中的新文章

- 原生(Native)应用程序开发与跨平台(cross-platform)移动应用程序开发: 如何选择? (原生(Native)应用程序开发与跨平台(cross-platform)移动应用程序开发: 如何选择?) – 阅读我们的概述, 以及跨平台(cross-platform)应用程序开发和原生(Native)方案各自的优势.
- 跨平台应用程序开发最流行的 6 种框架 (跨平台应用程序开发最流行的 6 种框架) – 查看各个框架的关键要素, 帮助你为跨平台项目选择正确的框架.

教程的改进和新增

- Kotlin Multiplatform 入门 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html) – 学习使用 Kotlin 进行跨平台移动应用程序开发, 并创建一个可以同时运行于 Android 和 iOS 平台的应用程序.
- 使用 React 和 Kotlin/JS 创建 Web 应用程序 (教程 - 使用 React 和 Kotlin/JS 创建 Web 应用程序) – 创建一个浏览器应用程序, 学习一个典型的 React 程序中用到的 Kotlin 的 DSL 和功能特性.

Kotlin 的发布版本文档的变更

我们不再对各个发布版提供推荐的 kotlinx 库列表. 这个列表只包含推荐的版本, 以及 Kotlin 本身测试过的版本. 其中不包括各个库直接的相互依赖, 以及它们需要 kotlinx 的哪个版本, 这些版本可能与推荐的 Kotlin 版本不同.

我们正在寻找方法来提供库之间相互关联相互依赖的信息, 以便于你来判断, 当你升级你的项目的 Kotlin 版本时, 应该使用 kotlinx 库的哪个版本.

安装 Kotlin 1.7.20

IntelliJ IDEA (https://www.jetbrains.com/idea/download/) 2021.3, 2022.1, 和 2022.2 会自动建议将 Kotlin plugin 更新到版本 1.7.20.

> 对于 Android Studio Dolphin (213), Electric Eel (221), 和 Flamingo (222), Android Studios 的后续更新会带有 Kotlin plugin 1.7.20.

新的命令行编译器可以通过 GitHub 发布页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.7.20) 下载.

Kotlin 1.7.20 的兼容性指南

尽管 Kotlin 1.7.20 是一个增量发布版, 但我们仍然不得不进行了一些不兼容的变更, 以解决 Kotlin 1.7.0 中的一些问题.

关于这些不兼容的变更, 详情请参见 Kotlin 1.7.20 兼容性指南 (Kotlin 1.7.20 兼容性指南).

Kotlin 1.7.0 版中的新功能

> IDE 从 IntelliJ IDEA 2021.2, 2021.3, 和 2022.1 开始支持 Kotlin 1.7.0.

最终更新: 2024/09/10

发布日期: 2022/06/09 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.7.0 已经发布了. 它公布了新的 Kotlin/JVM K2 编译器的 Alpha 版, 发布了语言功能的稳定版, 并为 JVM, JS, 和 Native 平台带来了性能改进.

下面是这个版本中的主要更新:

- 发布了新的 Kotlin K2 编译器 Alpha 版, 它带来很多性能改进. 这个编译器只能用于 JVM, 并且所有的编译器 plugin, 包括 kapt, 都不能使用.
- Gradle 中增量编译的新方案. 在依赖的非 Kotlin 模块中的变更, 现在也能支持增量编译, 并且与 Gradle 兼容.
- 我们发布了 明确要求使用者同意(Opt-in Requirement) 注解, 明确非 null 类型, 以及 构建器推断 的稳定版.
- 类型参数的新的下划线操作符. 当其它类型已指定时, 你可以使用这个操作符来自动推断一个参数类型.
- 允许接口的实现代理给内联类的内联值. 现在你可以创建轻量的封装(wrapper)类, 大多数情况下不会消耗内存.

关于这个版本的变更概要, 请参见以下视频:

JVM 平台的新的 Kotlin K2 编译器 (Alpha 版)

这个 Kotlin 发布版引入了新的 Kotlin K2 编译器的 **Alpha** 版. 新的编译器致力于提升新的语言功能的开发速度, 同一 Kotlin 支持的所有平台, 带来性能改进, 并为编译器扩展提供 API.

关于新编译器, 以及它的益处, 我们发布了一些详细解释:

- Kotlin 新编译器之路 (https://www.youtube.com/watch?v=iTdJJq_LyoY)
- K2 编译器: 概要介绍 (https://www.youtube.com/watch?v=db19VFLZqJM)

需要指出, 在新的 K2 编译器 Alpha 版中, 我们主要集中于性能改进, 并且它只能用于 JVM 项目. 它不支持 Kotlin/JS, Kotlin/Native, 以及其它跨平台项目, 并且所有的编译器 plugin, 包括 kapt (kapt 编译器插件), 都不能使用.

在我们的内部项目中进行的评测结果非常优异:

项目	现在的 Kotlin 编译器性能	新 K2 Kotlin 编译器性能	性能提升
Kotlin	2.2 KLOC/s	4.8 KLOC/s	~ 2.2倍
YouTrack	1.8 KLOC/s	4.2 KLOC/s	~ 2.3倍
IntelliJ IDEA	1.8 KLOC/s	3.9 KLOC/s	~ 2.2倍
Space	1.2 KLOC/s	2.8 KLOC/s	~ 2.3倍

这里的 KLOC/s 性能数字表示编译器每秒处理的千行代码数.

你可以在你的 JVM 项目中查看性能提升, 并与旧编译器的结果进行比较. 要启用 Kotlin K2 编译器, 请使用以下编译器选项:

```
-Xuse-k2
```

此外, K2 编译器还 包括很多 bug 修正 (https://youtrack.jetbrains.com/issues/KT?q=tag:%20FIR-preview-qa%20%23Resolved). 请注意, 就连这个列表中的状态为 **State: Open** 的问题, 在 K2 中事实上也被修正了.

Kotlin 的下一个发布版本将会改进 K2 编译器的稳定性, 并提供更多功能, 敬请期待!

如果你使用 Kotlin K2 编译器时遇到任何性能问题, 请 向我们的问题追踪系统提交报告 (https://kotl.in/issue).

语言功能

Kotlin 1.7.0 引入的新的语言功能, 支持通过代理实现接口, 以及新的类型参数的下划线操作符. 此外, 对于以前版本中引入的几个语言功能预览版, Kotlin 1.7.0 还发布了它们的稳定版:

- 接口的实现代理给内联类的内联值

- 类型参数的下划线操作符
- 构建器推断的稳定版
- 明确要求使用者同意(Opt-in Requirement)的稳定版
- 明确非 null 类型的稳定版

允许接口的实现代理给内联类的内联值

如果你想要对一个值或一个类实例创建一个轻量的封装(wrapper), 就需要手动实现所有的接口方法. 通过代理实现结构解决了这个问题, 但在 1.7.0 之前不能用于内联类. 这个限制现在已经解决了, 现在你可以创建轻量的封装, 大多数情况下不会消耗内存.

```
interface Bar {
    fun foo() = "foo"
}

@JvmInline
value class BarWrapper(val bar: Bar): Bar by bar

fun main() {
    val bw = BarWrapper(object: Bar {})
    println(bw.foo())
}
```

类型参数的下划线操作符

Kotlin 1.7.0 为类型参数引入了一个下划线操作符, `_`. 当其它类型已指定时, 你可以使用它来自动推断一个类型参数:

```
abstract class SomeClass<T> {
    abstract fun execute(): T
}

class SomeImplementation : SomeClass<String>() {
    override fun execute(): String = "Test"
}

class OtherImplementation : SomeClass<Int>() {
```

```
    override fun execute(): Int = 42
}

object Runner {
    inline fun <reified S: SomeClass<T>, T> run(): T {
        return
S::class.java.getDeclaredConstructor().newInstance().execute()
    }
}

fun main() {
    // T 被推断为 String, 因为 SomeImplementation 继承自
SomeClass<String>
    val s = Runner.run<SomeImplementation, _>()
    assert(s == "Test")

    // T 被推断为 Int, 因为 OtherImplementation 继承自 SomeClass<Int>
    val n = Runner.run<OtherImplementation, _>()
    assert(n == 42)
}
```

你可以在参数列表中的任何位置使用下划线操作符来推断一个类型参数.

构建器推断的稳定版

构建器推断是一种特殊的类型推断, 在调用泛型构建器函数时非常有用. 它可以帮助编译器, 利用一个调用的 Lambda 表达式参数之内的其它调用的类型信息, 推断这个调用本身的类型参数.

过去, 在 1.6.0 中引入 ("对构建器推断的变更" in "Kotlin 1.6.0 版中的新功能") 了编译器选项 `-Xenable-builder-inference`. 从 1.7.0 开始, 不需要指定这个编译器选项, 如果通常的类型推断对一个类型无法得到足够的信息, 构建器推断会自动启用.

参见 如何编写自定义的泛型构建器 (通过构建器类型推断(Builder Type Inference)使用构建器).

明确要求使用者同意(Opt-in Requirement)的稳定版

明确要求使用者同意(Opt-in Requirement) (明确要求使用者同意的功能(Opt-in Requirement)) 现在升级为 稳定版 (Kotlin 各部分组件的稳定性), 并且不再需要额外的编译器配置.

在 1.7.0 之前, opt-in 功能本身要求参数 `-opt-in=kotlin.RequiresOptIn` 来关闭警告信息. 现在不再需要了; 但是, 你仍然可以使用编译器参数 `-opt-in`, 在模块范围内 ("模块范围内同意使用(Module-

wide opt-in)" in "明确要求使用者同意的功能(Opt-in Requirement)") 同意使用其他注解.

明确非 null 类型的稳定版

在 Kotlin 1.7.0 中, 明确非 null 类型升级为 稳定版 (Kotlin 各部分组件的稳定性). 在扩展泛型的 Java 类和接口时, 这个功能提供了更好的互操作性.

你可以使用新的语法 `T & Any`, 在使用端将一个泛型类型参数标记为明确非 null. 这个语法来自 交叉类型(Intersection Types) (https://en.wikipedia.org/wiki/Intersection_type) 的标记形式, 并且现在 `&` 左侧必须是上界可为 null 的类型参数, 右侧必须是非 null 的 `Any`:

```
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y

fun main() {
    // OK
    elvisLike<String>("", "").length
    // 错误: 'null' 不能作为一个非 null 类型的值
    elvisLike<String>("", null).length

    // OK
    elvisLike<String?>(null, "").length
    // 错误: 'null' 不能作为一个非 null 类型的值
    elvisLike<String?>(null, null).length
}
```

关于明确非 null 类型, 详情请参见 这个 KEEP (https://github.com/Kotlin/KEEP/blob/c72601cf35c1e95a541bb4b230edb474a6d1d1a8/proposals/definitely-non-nullable-types.md).

Kotlin/JVM

这个发布版带来了对 Kotlin/JVM 编译器的性能改进, 以及一个新的编译器 选项. 此外, 对函数式接口构造器的可调用引用升级为稳定版. 注意, 从 1.7.0 开始, Kotlin/JVM 的默认编译目标版本是 `1.8`.

- 编译器性能优化
- 新的编译器选项 `-Xjdk-release`
- 对函数式接口构造器的可调用引用: 稳定版
- 删除了 JVM 编译目标版本 1.6

编译器性能优化

Kotlin 1.7.0 引入了对 Kotlin/JVM 编译器的性能改进. 根据我们的评测, 编译时间与 Kotlin 1.6.0 相比 平均缩减了 10% (https://youtrack.jetbrains.com/issue/KT-48233/Switching-to-JVM-IR-backend-increases-compilation-time-by-more-t#focus=Comments-27-6114542.0-0). 由于字节码后期处理的改进, 大量使用内联函数的项目, 比如 使用 `kotlinx.html` 的项目 (https://youtrack.jetbrains.com/issue/KT-51416/Compilation-of-kotlinx-html-DSL-should-still-be-faster), 编译速度会变得更快.

新的编译器选项: -Xjdk-release

Kotlin 1.7.0 添加了新的编译器选项, `-Xjdk-release`. 这个选项类似于 javac 的命令行选项 `--release` (http://openjdk.java.net/jeps/247). `-Xjdk-release` 选项控制编译目标的字节码版本, 并将 classpath 中的 JDK 的 API 限制为指定的 Java 版本. 比如, `kotlinc -Xjdk-release=1.8` 不会允许引用 `java.lang.Module`, 即使依赖项中的 JDK 是 9 或更高版本.

> 这个选项 不保证 (https://youtrack.jetbrains.com/issue/KT-29974) 对所有的 JDK 分发版都有效.

请在 这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-29974/Add-a-compiler-option-Xjdk-release-similar-to-javac-s-release-to) 中留下你的反馈.

对函数式接口构造器的可调用引用: 稳定版

对函数式接口构造器的 可调用的引用 ("可调用的引用" in "反射") 现在升级为 稳定版 (Kotlin 各部分组件的稳定性). 请参见, 如何从一个带构造器函数的接口 迁移 ("从带构造器函数的接口迁移到函数式接口" in "函数式 (SAM) 接口") 到一个使用可调用引用的函数式接口.

如果你遇到问题, 请在这个 YouTrack (https://youtrack.jetbrains.com/newissue?project=kt) 中提交报告.

删除了 JVM 编译目标版本 1.6

对 Kotlin/JVM 的默认编译目标版本现在是 `1.8`. 编译目标版本 `1.6` 已被删除.

请迁移到 JVM 编译目标 1.8 或更高版本. 关于如何更新 JVM 编译目标版本, 请参见:

- Gradle ("JVM 任务独有的属性" in "Kotlin Gradle plugin 中的编译器选项")
- Maven ("JVM 独有的属性" in "Maven")
- 命令行编译器 ("-jvm-target version" in "Kotlin 编译器选项")

Kotlin/Native

Kotlin 1.7.0 包括 与 Objective-C 和 Swift 交互性的变更, 并且将以前的发布版中引入的功能升级为稳定版. 还带来了对新的内存管理器的性能改进, 以及其他更新:

- 对新的内存管理器的性能改进
- 对 JVM 和 JS IR 后端统一的编译器 plugin ABI
- 支持独立的 Android 可执行文件
- 与 Swift async/await 交互: 返回 `Void` 而不是 `KotlinUnit`
- 禁止未声明的异常通过 Objective-C 桥
- 与 CocoaPods 集成的改进
- 修改 Kotlin/Native 编译器的下载 URL

对新的内存管理器的性能改进

> 新的 Kotlin/Native 内存管理器现在是 Alpha 版 (Kotlin 各部分组件的稳定性). 将来它可能发生不兼容的变化, 并需要手工迁移. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-48525) 提供你的反馈意见.

新的内存管理器还处于 Alpha 版, 但它在稳步的向 稳定版 (Kotlin 各部分组件的稳定性) 发展. 这个发布版带来了对新的内存管理器显著的性能改进, 尤其是垃圾收集(GC)功能. 具体来说, 在 1.6.20 中引入 (Kotlin 1.6.20 版中的新功能) 的 sweep phase 的并发实现, 现在默认启用了. 这个功能可以帮助减少 GC 执行时的应用程序暂停时间. 新的 GC 时间调度器能够更好的选择 GC 频率, 尤其是对更大的 heap 内存.

此外, 我们还特别优化了 debug 版二进制文件, 确保在内存管理器的实现代码中使用了适当的优化级别和链接时优化. 根据我们的测算, 对 debug 版二进制文件, 这些改进帮助我们改善了执行时间大约 30%.

请在你的项目中试用新的内存管理器, 看看它的效果如何, 并通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-48525) 提供你的反馈意见.

对 JVM 和 JS IR 后端统一的编译器 plugin ABI

从 Kotlin 1.7.0 开始, Kotlin Multiplatform Gradle plugin 对 Kotlin/Native 默认使用内嵌的编译器 jar. 这个功能作为实验性功能 在 1.6.0 中引入 ("JVM 和 JS IR 后端统一的编译器 plugin ABI" in "Kotlin 1.6.0 版中的新功能"), 现在它已经升级为稳定版, 可以使用了.

这个改进对于库的作者非常方便, 因为它改进了编译器 plugin 的开发体验. 在这个发布版之前, 你必须为 Kotlin/Native 提供单独的 artifact, 现在, 对 Native 和其他支持的平台, 你可以使用相同的编译器 plugin artifact.

> 这个功能可能需要 plugin 开发者对他们既有的 plugin 进行一些迁移步骤.
>
> 关于如何为这个更新调整你的 plugin, 请参见这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-48595).

支持独立的 Android 可执行文件

Kotlin 1.7.0 对 Android Native 编译目标生成标准的可执行文件提供了完全的支持. 这个功能 在 1.6.20 中引入 ("支持独立的 Android 可执行文件" in "Kotlin 1.6.20 版中的新功能"), 现在已默认启用.

如果你想要退回到以前的行为, 让 Kotlin/Native 生成共用的库, 请使用以下设置:

```
binaryOptions["androidProgramType"] = "nativeActivity"
```

与 Swift async/await 交互: 返回 Void 而不是 KotlinUnit

在 Swift 中, Kotlin `suspend` 函数现在返回 `Void` 类型而不是 `KotlinUnit`. 这是与 Swift 的 `async`/`await` 交互功能改进后的结果. 这个功能 在 1.6.20 中引入 ("与 Swift async/await 的交互: 返回 Swift 的 Void 类型, 而不是 KotlinUnit 类型" in "Kotlin 1.6.20 版中的新功能"), 这个发布版中会默认启用.

你不再需要使用 `kotlin.native.binary.unitSuspendFunctionObjCExport=proper` 属性来对这样的函数返回适当的类型.

禁止未声明的异常通过 Objective-C 桥

当你从 Swift/Objective-C 代码调用 Kotlin 代码时(或者反过来), 如果这个代码抛出一个异常, 它应该被异常发生处的代码来处理, 除非你明确的允许异常经过适当的转换后在语言之间传递(比如, 使用 `@Throws` 注解).

在以前的版本中, Kotlin 的行为不太正确, 某些情况下, 未声明的异常可以从一种语言"泄露"到另一种语言. Kotlin 1.7.0 修正了这个问题, 现在这样的情况会导致程序终止.

因此, 比如, 如果在 Kotlin 中你有一个 Lambda 表达式 `{ throw Exception() }`, 并从 Swift 调用它, 在 Kotlin 1.7.0 中, 程序会在异常到达 Swift 代码时立即终止. 在以前的 Kotlin 版本中, 这样的异常可以泄露到 Swift 代码中.

`@Throws` 注解会继续向以前一样工作.

与 CocoaPods 集成的改进

从 Kotlin 1.7.0 开始, 如果想要在你的项目中集成 CocoaPods, 不再需要安装 `cocoapods-generate` plugin.

在以前的版本中, 你需要安装 CocoaPods 依赖项管理器和 `cocoapods-generate` plugin 才能使用 CocoaPods, 比如, 用来在 Kotlin Multiplatform Mobile 项目中管理 iOS 依赖项 ("使用 CocoaPods" in "添加 iOS 依赖项").

现在设置与 CocoaPods 的集成变得更加简单, 而且我们解决了 `cocoapods-generate` 不能在 Ruby 3 和更高版本上安装的问题. 现在还支持最新的 Ruby 版本, 它在 Apple M1 上工作得更好.

关于如何设置环境, 请参见 设置与 CocoaPods 的集成 ("设置 CocoaPods 环境" in "CocoaPods 概述与设置").

修改 Kotlin/Native 编译器的下载 URL

从 Kotlin 1.7.0 开始, 你可以定制 Kotlin/Native 编译器的下载 URL. 当 CI 环境禁止使用外部链接时, 这个功能会很有用.

默认的起始 URL 是 `https://download.jetbrains.com/kotlin/native/builds`, 如果要修改, 请使用以下 Gradle 属性:

```
kotlin.native.distribution.baseDownloadUrl=https://example.com
```

> 下载器会向这个起始 URL 添加 native 版本和编译目标 OS, 确保下载到实际的编译器发布版.

Kotlin/JS

Kotlin/JS 包括对 JS IR 编译器后端 (使用 IR 编译器) 的更多改进, 以及改善你的开发体验的其他更新:

- 对新的 IR 后端的性能改进
- 使用 IR 时对成员名称极简化(Minification)

- 在 IR 后端中使用 polyfill 支持旧的浏览器
- 从 js 表达式动态装载 JavaScript 模块
- 为 JavaScript 测试运行器指定环境变量

对新的 IR 后端的性能改进

这个发布版包含一些大的更新, 可以改进你的开发体验:

- Kotlin/JS 增量编译的性能得到了显著改善. 它可以花费更少的时间来构建你的 JS 项目. 大多数情况下, 增量重构建现在应该大致和旧的后端差不多.
- Kotlin/JS 最终 bundle 占用更少的空间, 因为我们大大缩减了最终 artifact 的大小. 对一些大型项目, 我们的评测显示产品 bundle 大小与旧的后端相比缩减了 20%.
- 对接口的类型检查有了数量级程度的改进.
- Kotlin 生成更加高质量的 JS 代码

使用 IR 时对成员名称极简化(Minification)

Kotlin/JS IR 编译器现在会使用它的内部信息, 分析你的 Kotlin 类和函数的关系, 进行更加高效的极简化, 缩短函数, 属性, 以及类的名称. 这样可以缩减最终产生的捆绑的应用程序大小.

当你在 production 模式下构建 Kotlin/JS 应用程序时, 会自动进行这样的极简化, 并且这个功能默认启用. 如果要禁用成员名称极简化, 请使用 `-Xir-minimized-member-names` 编译器 flag:

```
kotlin {
    js(IR) {
        compilations.all {
            compileKotlinTask.kotlinOptions.freeCompilerArgs +=
listOf("-Xir-minimized-member-names=false")
        }
    }
}
```

在 IR 后端中使用 polyfill 支持旧的浏览器

Kotlin/JS 的 IR 编译器后端现在包含与旧后端相同的 polyfill. Kotlin 标准库使用的 ES2015 中的方法在旧浏览器上并不全部支持, 包含这些 polyfill, 可以让使用新编译器编译的代码能够在旧浏览器

上正确运行. 只有被项目实际使用到的 polyfill 才会包含到最终的 bundle 中, 这样可以尽量减少对 bundle 大小的影响.

在使用 IR 编译器时, 这个功能会默认启用, 你不需要对它进行配置.

从 js 表达式动态装载 JavaScript 模块

使用 JavaScript 模块时, 大多数应用程序使用静态导入, 具体的使用方法请参见 JavaScript 模块集成 (JavaScript 模块). 但是, Kotlin/JS 过去缺少一种机制, 在你的应用程序运行时动态的装载 JavaScript 模块.

从 Kotlin 1.7.0 开始, 在 `js` 代码段内, 支持使用 JavaScript 中的 `import` 语句, 因此你可以在运行时动态的将包引入到你的应用程序中:

```
val myPackage = js("import('my-package')")
```

为 JavaScript 测试运行器指定环境变量

为了对 Node.js 包的解析进行微调, 或者向 Node.js 测试代码传递外部信息, 现在你可以指定供 JavaScript 测试运行器使用的环境变量. 要定义一个环境变量, 请在你的构建脚本的 `testTask` 代码段之内, 使用 `environment()` 函数, 参数是一个 键-值对:

```
kotlin {
    js {
        nodejs {
            testTask {
                environment("key", "value")
            }
        }
    }
}
```

标准库

在 Kotlin 1.7.0 中, 标准库有了大量的变更和改进. 引入了新的功能, 将实验性功能升级到稳定版, 还对 Native, JS, 和 JVM 平台统一了对命名捕获组(Named Capturing Group)的支持:

- 集合函数 min() 和 max() 返回非 null 值
- 在明确指定的下标处查找正规表达式匹配

- 延长对旧的语言和 API 版本的支持
- 通过反射访问注解
- 深度递归(Deep Recursive) 函数升级为稳定版
- 对默认的时间源(Time Source)使用基于内联类的时间标记器(Time mark)
- 对 Java Optionals 的新的扩展函数(实验性功能)
- 在 JS 和 Native 中支持命名捕获组(Named Capturing Group)

集合函数 min() 和 max() 返回非 null 值

在 Kotlin 1.4.0 (Kotlin 1.4.0 版中的新功能) 中, 我们将集合函数 `min()` 和 `max()` 重命名为 `minOrNull()` 和 `maxOrNull()`. 这些新名称更好的反应函数的行为 – 如果接受者集合为空, 则返回 null. 还有助于让 Kotlin 集合 API 的函数行为与命名规约保持整体一致.

对函数 `minBy()`, `maxBy()`, `minWith()`, 和 `maxWith()` 也是如此, 在 Kotlin 1.4.0 中, 所有这些函数都有了对应的 *OrNull() 同义函数. 被这个变更影响的旧函数, 已被逐渐废弃.

Kotlin 1.7.0 重新引入了原来的函数名称, 但返回类型为非 null. 新的 `min()`, `max()`, `minBy()`, `maxBy()`, `minWith()`, 和 `maxWith()` 函数, 现在会严格的返回集合元素, 或抛出一个异常.

```
fun main() {
    val numbers = listOf<Int>()
    println(numbers.maxOrNull()) // 返回 "null"
    println(numbers.max()) // 抛出异常: "Exception in... Collection
is empty."
}
```

在明确指定的下标处查找正则表达式匹配

在 1.5.30 中引入 ("在一个指定的位置匹配正则表达式" in "Kotlin 1.5.30 版中的新功能") 的 `Regex.matchAt()` 和 `Regex.matchesAt()` 函数现在升级为稳定版. 这些函数提供了一种方法, 在一个 `String` 或 `CharSequence` 中的一个指定的位置, 检查正则表达式是否存在一个完整的匹配.

`matchesAt()` 检查一个匹配, 并返回一个 boolean 结果:

```
fun main() {
    val releaseText = "Kotlin 1.7.0 is on its way!"
    // 正则表达式: 一个数字, 点号, 一个数字, 点号, 一个或多个数字
```

```
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()

    println(versionRegex.matchesAt(releaseText, 0)) // 输出结果为
"false"
    println(versionRegex.matchesAt(releaseText, 7)) // 输出结果为
"true"
}
```

matchAt() 如果找到匹配结果, 则返回匹配结果 , 如果没有找到匹配结果, 则返回 null:

```
fun main() {
    val releaseText = "Kotlin 1.7.0 is on its way!"
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()

    println(versionRegex.matchAt(releaseText, 0)) // 输出结果为 "null"
    println(versionRegex.matchAt(releaseText, 7)?.value) // 输出结果为
"1.7.0"
}
```

希望你能通过这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-34021) 提供你的反馈意见.

延长对旧的语言和 API 版本的支持

为了支持库的作者开发库供更多旧版本的 Kotlin 使用, 也为了处理快速增长的 Kotlin 主发布版本, 我们延长了对旧的语言和 API 版本的支持.

在 Kotlin 1.7.0 中, 我们支持 3 个版本前的语言和 API 版本, 而不是 2 个. 因此使用 Kotlin 1.7.0 支持开发库, 最低供 Kotlin 1.4.0 版本使用. 关于向后兼容性, 详情请参见 兼容性模式 (兼容模式).

通过反射访问注解

在 1.6.0 中引入 ("对于 1.8 JVM 编译目标, 在运行期保留的可重复注解" in "Kotlin 1.6.0 版中的新功能") 的扩展函数 KAnnotatedElement.findAnnotations() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect.full/find-annotations.html), 现在升级为 稳定版 (Kotlin 各部分组件的稳定性). 这个 反射 (反射) 函数对一个元素返回一个指定类型的所有注解, 包括单独使用的注解和重复的注解.

```
@Repeatable
annotation class Tag(val name: String)
```

```
@Tag("First Tag")
@Tag("Second Tag")
fun taggedFunction() {
    println("I'm a tagged function!")
}

fun main() {
    val x = ::taggedFunction
    val foo = x as KAnnotatedElement
    println(foo.findAnnotations<Tag>()) // 输出结果为:
[@Tag(name=First Tag), @Tag(name=Second Tag)]
}
```

深度递归(Deep Recursive) 函数升级为稳定版

从 Kotlin 1.4.0 (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-rc-debugging-coroutines/#Defining_deep_recursive_functions_using_coroutines) 开始, 深度递归(Deep Recursive)函数作为实验性功能引入, 现在在 Kotlin 1.7.0 中升级为 稳定版 (Kotlin 各部分组件的稳定性). 使用 `DeepRecursiveFunction`, 你可以定义一个函数, 让它的调用栈保存在 heap 内存中, 而不是使用实际的调用栈. 因此你可以运行非常深的递归计算. 要调用一个深度递归函数, 只需要 `invoke` 它.

在这个示例中, 一个深度递归函数用来递归的计算一个二叉树的深度. 虽然这个示例函数递归的调用它自身 100,000 次, 也不会抛出 `StackOverflowError`:

```
class Tree(val left: Tree?, val right: Tree?)

val calculateDepth = DeepRecursiveFunction<Tree?, Int> { t ->
    if (t == null) 0 else maxOf(
        callRecursive(t.left),
        callRecursive(t.right)
    ) + 1
}

fun main() {
    // 生成一个深度为 100_000 的树
    val deepTree = generateSequence(Tree(null, null)) { prev ->
        Tree(prev, null)
    }.take(100_000).last()
```

```
    println(calculateDepth(deepTree)) // 输出结果为: 100000
}
```

如果你的递归深度超过 1000 次调用, 就可以考虑在你的代码中使用深度递归函数.

对默认的时间源(Time Source)使用基于内联类的时间标记器(Time mark)

Kotlin 1.7.0 改善了时间测量功能的性能, 方法是将 `TimeSource.Monotonic` 返回的时间标记器 (Time mark) 改为内联的值类. 因此, 调用 `markNow()`, `elapsedNow()`, `measureTime()`, 和 `measureTimedValue()` 之类的函数, 不会为它们的 `TimeMark` 实例包装类分配内存. 尤其是在测量一个热点部分中的一段代码时, 这个功能可以减少测量对性能的影响:

```
@OptIn(ExperimentalTime::class)
fun main() {
    val mark = TimeSource.Monotonic.markNow() // 返回的 `TimeMark` 是
内联类
    val elapsedDuration = mark.elapsedNow()
}
```

> 只有当获得 `TimeMark` 时所用的时间源(Time Source), 可以静态的确定为是 `TimeSource.Monotonic` 时, 这个优化才起作用.

对 Java Optionals 的新的扩展函数(实验性功能)

Kotlin 1.7.0 带来了新的便利函数, 可以简化 Java 中的 `Optional` 类的使用. 这些新函数可以用来在 JVM 上解封和转换 optional 对象, 让使用 Java API 更加简洁.

通过使用扩展函数 `getOrNull()`, `getOrDefault()`, 和 `getOrElse()`, 如果 `Optional` 的值存在, 可以让你得到这个值. 否则, 你会分别得到 `null`, 一个默认值, 或由一个函数返回的值:

```
val presentOptional = Optional.of("I'm here!")

println(presentOptional.getOrNull())
// 输出结果为: "I'm here!"

val absentOptional = Optional.empty<String>()

println(absentOptional.getOrNull())
// 输出结果为: null
```

```
println(absentOptional.getOrDefault("Nobody here!"))
// 输出结果为: "Nobody here!"
println(absentOptional.getOrElse {
    println("Optional was absent!")
    "Default value!"
})
// 输出结果为: "Optional was absent!"
// 输出结果为: "Default value!"
```

使用扩展函数 `toList()`, `toSet()`, 和 `asSequence()`, 如果 `Optional` 有值, 会将值转换为一个 List, Set, 或 Sequence, 否则返回一个空集合. 扩展函数 `toCollection()` 会将 `Optional` 的值添加到一个已经存在的目标集合中:

```
val presentOptional = Optional.of("I'm here!")
val absentOptional = Optional.empty<String>()
println(presentOptional.toList() + "," + absentOptional.toList())
// 输出结果为: ["I'm here!"], []
println(presentOptional.toSet() + "," + absentOptional.toSet())
// 输出结果为: ["I'm here!"], []
val myCollection = mutableListOf<String>()
absentOptional.toCollection(myCollection)
println(myCollection)
// 输出结果为: []
presentOptional.toCollection(myCollection)
println(myCollection)
// 输出结果为: ["I'm here!"]
val list = listOf(presentOptional, absentOptional).flatMap {
it.asSequence() }
println(list)
// 输出结果为: ["I'm here!"]
```

在 Kotlin 1.7.0 中, 这些扩展函数作为实验性功能引入. 关于 `Optional` 的扩展, 更多详情请参见 这个 KEEP (https://github.com/Kotlin/KEEP/pull/291). 和往常一样, 欢迎在 Kotlin 问题追踪系统 (https://kotl.in/issue) 中反馈你的意见.

在 JS 和 Native 中支持命名捕获组(Named Capturing Group)

从 Kotlin 1.7.0 开始, 命名捕获组(Named Capturing Group) 不仅在 JVM 上支持, 而且在 JS 和 Native 平台也支持了.

要为一个捕获组指定一个名称, 请在你的正规表达式中使用 (?<name>group) 语法. 要得到被这个组匹配的文本, 请调用新引入的函数 MatchGroupCollection.get() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/get.html), 参数是组的名称.

通过名称获取匹配的组的值

我们来看看这个示例, 它匹配城市的座标. 要得到正规表达式匹配的组的集合, 请使用 groups (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-match-result/groups.html). 比较一下得到组的两种方法, 一种是通过组的编号(下标)来得到, 另一种是通过组的名称来得到, 然后使用 value 得到组的内容:

```
fun main() {
    val regex = "\\b(?<city>[A-Za-z\\s]+),\\s(?<state>[A-Z]{2}):\\s(?<areaCode>[0-9]{3})\\b".toRegex()
    val input = "Coordinates: Austin, TX: 123"
    val match = regex.find(input)!!
    println(match.groups["city"]?.value) // 输出结果为: "Austin" — 通过名称得到组
    println(match.groups[2]?.value) // 输出结果为: "TX" — 通过下标得到组
}
```

命名的反向引用

你现在还可以在反向引用组时使用组的名称. 反向引用会匹配在前面曾经被一个捕获组匹配过的相同的文字. 要使用这个功能, 请在你的正规表达式中使用 \k<name> 语法:

```
fun backRef() {
    val regex = "(?<title>\\w+), yes \\k<title>".toRegex()
    val match = regex.find("Do you copy? Sir, yes Sir!")!!
    println(match.value) // 输出结果为: "Sir, yes Sir"
    println(match.groups["title"]?.value) // 输出结果为: "Sir"
}
```

在替换表达式中使用命名的组

命名的组引用可以与替换表达式一起使用. 比如 replace() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-regex/replace.html) 函数, 会使用一个替换表达式替换在输入文本中指定的正规表达式的所有匹配, 以及 replaceFirst() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-regex/replace-first.html) 函数, 只替换第一个匹配.

在替换字符串中出现的 `${name}` 会被替换为这个名称对应的捕获组的内容. 你可以比较在替换表达式通过名称和通过下标引用组的方法:

```
fun dateReplace() {
    val dateRegex = Regex("(?<dd>\\d{2})-(?<mm>\\d{2})-(?
<yyyy>\\d{4})")
    val input = "Date of birth: 27-04-2022"
    println(dateRegex.replace(input, "\${yyyy}-\${mm}-\${dd}")) //
输出结果为: "Date of birth: 2022-04-27" – 通过名称引用组
    println(dateRegex.replace(input, "\$3-\$2-\$1")) // 输出结果为:
"Date of birth: 2022-04-27" – 通过下标引用组
}
```

Gradle

这个发布版引入了新的构建报告功能, 支持 Gradle plugin 变体(Variant), 在 kapt 中的新的统计功能, 以及其他很多功能:

- 增量编译的新方案
- 新功能: 构建报告, 用于追踪编译器性能
- 对 Gradle 和 Android Gradle plugin 的最小支持版本的变更
- 支持 Gradle plugin 变体(Variant)
- Kotlin Gradle plugin API 中的更新
- 可以通过通过 plugin API 使用 sam-with-receiver plugin
- 编译任务中的更新
- 新功能: 在 kapt 中, 对每个注解处理器生成的文件的统计
- 废弃了系统属性 kotlin.compiler.execution.strategy
- 删除了废弃的选项, 方法, 和 plugin

增量编译的新方案

增量编译的新方案是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要明确同意使用(Opt-in)(详情请参见下文). 我们鼓励你只为评估目的来使用这个功能, 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

在 Kotlin 1.7.0 中, 我们重写了对跨模块变更的增量编译功能. 对发生在依赖的非 Kotlin 模块内的变更, 现在也支持增量编译了, 而且它兼容于 Gradle 构建缓存 (https://docs.gradle.org/current/userguide/build_cache.html). 对编译回避的支持也有了改进.

如果你使用构建缓存, 或在非 Kotlin Gradle 模块中频繁的进行修改, 那么我们期待你会看到新方案的显著改进. 我们对 Kotlin 项目的 `kotlin-gradle-plugin` 模块的测试显示, 对在缓存命中之后的变更, 性能改善超过 80%.

要试用这个新方案, 请在你的 `gradle.properties` 中设置以下选项:

```
kotlin.incremental.useClasspathSnapshot=true
```

增量编译的新方案目前只能用于 JVM 后端和 Gradle 构建系统.

关于增量编译新方案的实现方式, 详情请参见 这篇 blog (https://blog.jetbrains.com/kotlin/2022/07/a-new-approach-to-incremental-compilation-in-kotlin/).

我们的计划是继续稳定这个技术, 并添加对其他后端(比如 JS)和其他构建系统的支持. 如果你在这个编译体系中遇到任何问题, 或任何奇怪的行为, 欢迎通过 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 向我们反馈. 谢谢!

Kotlin 开发组非常感谢 Ivan Gavrilovic (https://github.com/gavra0), Hung Nguyen (https://github.com/hungvietnguyen), Cédric Champeau (https://github.com/melix), 以及其他外部贡献者提供的帮助.

对 Kotlin 编译器任务的构建报告

Kotlin 构建报告是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin 1.7.0 引入了构建报告功能, 帮助追踪编译器的性能. 报告包括不同编译阶段的执行时间, 以及编译不能增量执行的原因.

如果你想要调查编译器任务相关的问题, 构建报告会很方便, 比如:

- Gradle 构建耗费了太多时间, 你想要调查性能低下的根本原因.
- 相同项目的编译时间发生了变化, 有时花费几秒, 有时花费几分钟.

要启用构建报告, 请在 `gradle.properties` 中声明构建报告输出的保存位置:

```
kotlin.build.report.output=file
```

可以使用以下值 (以及它们的组合):

- `file` 将构建报告保存到本地文件.
- `build_scan` 将构建报告保存到 build scan (https://scans.gradle.com/) 的 `custom values` 小节.

  > Gradle Enterprise plugin 会限制 custom values 的数量和长度. 在很大的项目中, 有些值可能会丢失.

- `http` 通过 HTTP(S) 提交构建报告. 使用 POST 方法传送 JSON 格式的测量结果. 数据可能在各个版本中发生变化. 你可以在 Kotlin 代码仓库 (https://github.com/JetBrains/kotlin/blob/master/libraries/tools/kotlin-gradle-plugin/src/common/kotlin/org/jetbrains/kotlin/gradle/plugin/statistics/CompileStatisticsData.kt) 中看到传送的数据的当前版本.

有 2 种常见情况, 对长时间运行的编译, 分析构建报告可以帮助你解决问题:

- 构建没有增加运行. 分析原因并解决底层的问题.
- 构建是增加运行, 但耗费了太多时间. 可以试试重新组织源代码文件 — 把大的文件切分成小文件, 单独的类保存到不同的文件中, 对大的类进行重构, 在不同的文件中声明顶层函数, 等等.

关于新的构建报告功能, 详情请参见 这篇 blog (https://blog.jetbrains.com/kotlin/2022/06/introducing-kotlin-build-reports/).

欢迎在你的开发环境中试用构建报告. 如果你有任何反馈意见, 遇到任何问题, 或有改进意见, 请通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/newIssue) 报告. 谢谢!

提升最小支持版本

从 Kotlin 1.7.0 开始, Gradle 的最小支持版本是 6.7.1. 我们不得不 提升版本 (https://youtrack.jetbrains.com/issue/KT-49733/Bump-minimal-supported-Gradle-version-to-6-7-1) 以便支持 Gradle plugin 变体(Variant) 以及新的 Gradle API. 将来, 由于有了 Gradle plugin 变体功能, 我们应该不会再需要经常提升最小支持版本.

此外, Android Gradle plugin 的最小支持版本现在是 3.6.4.

支持 Gradle plugin 变体(Variant)

Gradle 7.0 为 Gradle plugin 作者引入了一个新功能 — 带变体的 plugin (https://docs.gradle.org/7.0/userguide/implementing_gradle_plugins.html#plugin-with-variants). 在为 Gradle 7.1 以下版本维护兼容性时, 这个功能使得更容易为新的 Gradle 功能添加支持. 详情请参见 Gradle 中的变体选择 (https://docs.gradle.org/current/userguide/variant_model.html).

使用 Gradle plugin 变体, 我们可以为不同的 Gradle 版本发布不同的 Kotlin Gradle plugin 变体. 目标是要在 `main` 变体中支持基本的 Kotlin 编译, 这个变体对应于最旧的 Gradle 支持版本. 每个变体将会拥有来自对应发布版的 Gradle 功能实现. 最新的变体将会支持最大的 Gradle 功能集. 通过这个方案, 我们可以继续支持旧的 Gradle 版本, 但包含功能限制.

目前, Kotlin Gradle plugin 只有 2 个变体:

- `main` 用于 Gradle 版本 6.7.1–6.9.3
- `gradle70` 用于 Gradle 版本 7.0 以及更高版本

在未来的 Kotlin 发布版中, 我们可能会添加更多变体.

要查看你的构建使用哪个变体, 请启用 `--info` log 级别 (https://docs.gradle.org/current/userguide/logging.html#sec:choosing_a_log_level), 然后在输出日志中查找以 `Using Kotlin Gradle plugin` 开头的字符串, 比如, `Using Kotlin Gradle plugin main variant`.

> 对于 Gradle 中变体选择的一些已知问题, 下面是变通方法:
>
> - pluginManagement 中的ResolutionStrategy 不支持有多个变体的 plugin (https://github.com/gradle/gradle/issues/20545)

- 当一个 plugin 被添加为 buildSrc 共通依赖项时, Plugin 变体被忽略 (https://github.com/gradle/gradle/issues/20847)

请到 这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-49227/Support-Gradle-plugins-variants) 提供你的反馈意见.

Kotlin Gradle plugin API 中的更新

Kotlin Gradle plugin API artifact 有了一些改进:

- 新的接口, 用于带有用户可配置输入的 Kotlin/JVM 和 Kotlin/kapt 任务.
- 一个新的 KotlinBasePlugin 接口, 所有的 Kotlin plugin 继承这个接口. 如果你想要在任何 Kotlin Gradle plugin (JVM, JS, Multiplatform, Native, 以及其他平台) 被应用时触发某些配置动作, 可以使用这个接口:

  ```
  project.plugins.withType<org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin>() {
      // 在这里配置你的动作
  }
  ```

  你可以在 这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-48008/Consider-offering-a-KotlinBasePlugin) 中留下关于 KotlinBasePlugin 的反馈.

- 我们完成了 Android Gradle plugin 的基础工作, 让它能够配置 Kotlin 编译, 因此你不需要在你的构建中添加 Kotlin Android Gradle plugin. 请参见 Android Gradle Plugin 发布公告 (https://developer.android.com/studio/releases/gradle-plugin), 查看添加了哪些功能, 并试用它!

可以通过 plugin API 使用 sam-with-receiver plugin

sam-with-receiver 编译器 plugin (SAM-with-receiver 编译器插件) 现在可以通过 Gradle plugins DSL (https://docs.gradle.org/current/userguide/plugins.html#sec:plugins_block) 使用:

```
plugins {
    id("org.jetbrains.kotlin.plugin.sam.with.receiver") version
```

```
"$kotlin_version"
}
```

编译任务中的更新

在这个发布版中, 编译任务也有了很多更新:

- Kotlin 编译任务不再集成 Gradle 的 `AbstractCompile` 任务. 改为只继承 `DefaultTask`.
- `AbstractCompile` 任务拥有 `sourceCompatibility` 和 `targetCompatibility` 输入. 由于不再继承 `AbstractCompile` 任务, 在 Kotlin 用户的脚本中不再能够使用这些输入.
- `SourceTask.stableSources` 输入不再可用, 你应该使用 `sources` 输入. `setSource(...)` 方法仍然可以使用.
- 对于编译所需要的库列表, 所有的编译任务现在使用 `libraries` 输入. `KotlinCompile` 任务仍然拥有已废弃的 Kotlin 属性 `classpath`, 将在未来的发布版本中删除.
- 编译任务仍然实现 `PatternFilterable` 接口, 可以过滤 Kotlin 源代码. `sourceFilesExtensions` 输入已被删除, 改为使用 `PatternFilterable` 方法.
- 废弃的 `Gradle destinationDir: File` 输出替换为 `destinationDirectory: DirectoryProperty` 输出.
- Kotlin/Native `AbstractNativeCompile` 任务现在继承 `AbstractKotlinCompileTool` 基类. 这个是迈向将 Kotlin/Native 构建工具集成到所有的其他工具中的第一步.

请在 这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-32805) 中留下你的反馈意见.

在 kapt 中, 对每个注解处理器生成的文件的统计

过去, `kotlin-kapt` Gradle plugin 可以 对每个处理器报告性能统计 (https://github.com/JetBrains/kotlin/pull/4280). 从 Kotlin 1.7.0 开始, 它还可以每个注解处理器报告生成的文件数量统计.

这个功能可以用来追踪构建中是否存在未使用的注解处理器. 你可以使用生成的报告, 查找哪些模块触发了不必要的注解处理器, 然后更新模块, 不再触发这些注解处理器.

要启用统计功能, 需要以下 2 步:

- 在你的 `build.gradle.kts` 中, 将 `showProcessorStats` flag 设置为 `true`:

```
kapt {
    showProcessorStats = true
}
```

- 在你的 `gradle.properties` 中, 将 `kapt.verbose` Gradle 属性设置为 `true`:

```
kapt.verbose=true
```

> 你还可以使用 命令行选项 `verbose` ("在命令行中使用" in "kapt 编译器插件"), 启用 verbose 输出.

统计结果会出现在 log 中, 级别为 `info`. 你会看到 `Annotation processor stats:` 行, 之后是每个注解处理器的执行时间统计. 再后面, 将是 `Generated files report:` 行, 之后是每个注解处理器生成的文件数量统计. 比如:

```
[INFO] Annotation processor stats:
[INFO] org.mapstruct.ap.MappingProcessor: total: 290 ms, init: 1 ms,
3 round(s): 289 ms, 0 ms, 0 ms
[INFO] Generated files report:
[INFO] org.mapstruct.ap.MappingProcessor: total sources: 2, sources
per round: 2, 0, 0
```

请在 这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-51132/KAPT-Support-reporting-the-number-of-generated-files-by-each-ann) 中留下你的反馈意见.

废弃了系统属性 kotlin.compiler.execution.strategy

Kotlin 1.6.20 中引入了 新的属性来定义 Kotlin 编译器的执行策略 ("用于定义 Kotlin 编译器执行策略的属性" in "Kotlin 1.6.20 版中的新功能"). 在 Kotlin 1.7.0 中, 开始了旧系统属性 `kotlin.compiler.execution.strategy` 的废弃周期, 改为使用新的属性.

使用系统属性 `kotlin.compiler.execution.strategy` 时, 你将收到一个警告信息. 这个属性将在将来的发布版中删除. 如果要保留旧的行为, 请将系统属性替换为相同名称的 Gradle 属性. 你在 `gradle.properties` 中可以这样做, 比如:

```
kotlin.compiler.execution.strategy=out-of-process
```

你也可以使用编译任务属性 `compilerExecutionStrategy`. 详情请参见 Gradle 章节 ("定义 Kotlin 编译器执行策略" in "Kotlin Gradle plugin 中的编译与缓存").

删除了废弃的选项, 方法, 和 plugin

删除了 useExperimentalAnnotation 方法

在 Kotlin 1.7.0 中, 我们完成了 Gradle 方法 `useExperimentalAnnotation` 的废弃周期. 如果使用一个模块中的一个 API 需要使用者的同意, 请改用 `optIn()`.

比如, 如果你的 Gradle 模块是跨平台模块:

```
sourceSets {
    all {
        languageSettings.optIn("org.mylibrary.OptInAnnotation")
    }
}
```

详情请参见 Kotlin 中的 明确要求使用者同意的功能(Opt-in Requirement) (明确要求使用者同意的功能(Opt-in Requirement)).

删除了废弃的编译器选项

我们完成了几个编译器选项废弃周期:

- 编译器选项 `kotlinOptions.jdkHome` 在 1.5.30 中被废弃, 在现在的发布版中已被删除. 如果包含这个选项, Gradle 构建现在会失败. 我们建议你使用 Java 工具链 ("支持 Java 工具链" in "Kotlin 1.5.30 版中的新功能"), 它从 Kotlin 1.5.30 开始支持.
- 废弃的编译器选项 `noStdlib` 也被删除了. Gradle plugin 使用属性 `kotlin.stdlib.default.dependency=true` 来控制是否存在 Kotlin 标准库.

> 编译器参数 `-jdkHome` 和 `-no-stdlib` 仍然可以使用.

删除了废弃的 plugin

在 Kotlin 1.4.0 中, `kotlin2js` 和 `kotlin-dce-plugin` plugin 已被废弃, 并在这个发布版中删除. 请使用新的 `org.jetbrains.kotlin.js` plugin 代替 `kotlin2js`. 如果 适当配置 (JavaScript 死代码剔除工具) Kotlin/JS Gradle plugin, 死代码剔除(Dead Code Elimination, DCE) 功能还会继续工作.

在 Kotlin 1.6.0 中, 我们将 `KotlinGradleSubplugin` 类的废弃级别修改为 `ERROR`. 开发者过去使用这个类来编写编译器 plugin. 在这个发布版中, 这个类已被删除

(https://youtrack.jetbrains.com/issue/KT-48831/). 请改为使用 KotlinCompilerPluginSupportPlugin 类.

> 最佳实践是在你的整个项目中使用 1.7.0 或更高版本的 Kotlin plugin.

删除了废弃的 coroutines DSL 选项和属性

我们删除了废弃的 Gradle DSL 选项 kotlin.experimental.coroutines 和 gradle.properties 中使用的属性 kotlin.coroutines. 现在你可以直接使用 *suspending 函数* (*"代码重构, 抽取函数" in "协程的基本概念"*) 或向你的构建脚本 添加 kotlinx.coroutines 依赖项 ("设置对 kotlinx 库的依赖项" in "配置 Gradle 项目").

关于协程, 详情请参见 协程指南 (协程指南).

删除了工具链扩展方法中的类型转换

在 Kotlin 1.7.0 之前, 在使用 Kotlin DSL 配置 Gradle 工具链时, 你必须将它类型转换为 JavaToolchainSpec 类:

```
kotlin {
    jvmToolchain {
        (this as
JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<MAJOR
_JDK_VERSION>)
    }
}
```

现在, 你可以省略 (this as JavaToolchainSpec) 部分:

```
kotlin {
    jvmToolchain {

languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
}
```

迁移到 Kotlin 1.7.0

安装 Kotlin 1.7.0

IntelliJ IDEA 2022.1 和 Android Studio Chipmunk (212) 会自动建议将 Kotlin plugin 更新到 1.7.0.

> 对于 IntelliJ IDEA 2022.2, 和 Android Studio Dolphin (213) 或 Android Studio Electric Eel (221), Kotlin plugin 1.7.0 会随之后的 IntelliJ IDEA 和 Android Studios 更新一起发布.

新的命令行编译器可以在 GitHub 发布页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.7.0) 下载.

将既有的项目迁移到 Kotlin 1.7.0, 或使用 Kotlin 1.7.0 创建新的项目

- 要将既有的项目迁移到 Kotlin 1.7.0, 请将 Kotlin 版本修改为 `1.7.0`, 然后重新导入你的 Gradle 或 Maven 项目. 详情请参见 如何更新到 Kotlin 1.7.0 ("更新到新的发布版" in "Kotlin 的发布版本").
- 要使用 Kotlin 1.7.0 创建一个新项目, 请更新 Kotlin plugin, 然后通过 **File** | **New** | **Project**, 运行项目向导.

Kotlin 1.7.0 兼容性指南

Kotlin 1.7.0 是一个 功能发布版 ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化"), 因此可能带来一些变更, 与你为更早的语言版本编写的代码不能兼容. 关于这样的变更, 详情请参见 Kotlin 1.7.0 兼容性指南 (Kotlin 1.7 兼容性指南).

Kotlin 1.6.20 版中的新功能

最终更新: 2024/09/10

发布日期: 2022/04/04 (["各发布版详情" in "Kotlin 的发布版本"])

Kotlin 1.6.20 带来了一些未来语言功能的预览版, 对跨平台项目默认使用层级结构, 还带来了对其它组件的改进.

你也可以观看这个概要介绍视频, 了解这个版本中的变更:

语言功能

在 Kotlin 1.6.20 中, 你可以试用 2 个新的语言功能:

- Kotlin/JVM 平台的上下文接受者(Context Receiver) 功能原型
- 明确非 null 类型

Kotlin/JVM 平台的上下文接受者(Context Receiver) 功能原型

> ⚠ 这是一个仅限 Kotlin/JVM 平台使用的功能原型. 启用 `-Xcontext-receivers` 选项后, 编译器将会产生预发布的二进制文件, 不能用于产品代码中. 请只在你的玩具项目中使用上下文接受者功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

在 Kotlin 1.6.20 中, 你的接受者可以不限于只有一个. 如果你需要更多接受者, 你可以让函数, 属性, 和类依赖于上下文 (或者叫做 *与上下文相关*) 方法是向它们的声明添加上下文接受者. 一个与上下文相关的声明会:

- 它要求所有声明的上下文接受者, 都作为隐含的接受者出现在调用者的作用范围内.
- 它将声明的上下文接受者代入函数体的作用范围内, 成为隐含的接受者.

```
interface LoggingContext {
    val log: Logger // 这个上下文提供一个 logger 的引用
}
```

```
context(LoggingContext)
fun startBusinessOperation() {
    // 你可以访问 log 属性, 因为 LoggingContext 是一个隐含的接受者
    log.info("Operation has started")
}

fun test(loggingContext: LoggingContext) {
    with(loggingContext) {
        // 你需要在这个作用范围内存在一个 LoggingContext, 作为隐含的接受者
        // 然后才能调用 startBusinessOperation()
        startBusinessOperation()
    }
}
```

要在你的项目中启用上下文接受者功能, 请使用 `-Xcontext-receivers` 编译器选项. 关于这个功能的详细描述, 以及它的语法, 请参见 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/context-receivers.md#detailed-design).

请注意, 目前的实现只是一个原型:

- 启用 `-Xcontext-receivers` 后, 编译器将会产生预发布的二进制文件, 不能用于产品代码中.
- 目前 IDE 对上下文接受者功能只有极少的支持

请在你的玩具项目中试用这个功能, 在 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-42435) 中并向我们反馈你的想法和体验. 如果你遇到任何问题, 请 提交新的 issue (https://kotl.in/issue).

明确非 null 类型

明确非 null 类型目前是 Beta 版 (Kotlin 各部分组件的稳定性). 已经接近稳定, 但未来可能会需要一些迁移步骤. 我们会尽力减少你需要进行的变更.

为了在扩展泛型的 Java 类和接口时提供更好的互操作性, Kotlin 1.6.20 允许你使用新的语法 `T & Any`, 将一个泛型类型参数标记为在使用端明确非 null. 这个语法来自 交叉类型(Intersection Types) (https://en.wikipedia.org/wiki/Intersection_type) 的标记形式, 并且现在 `&` 左侧必须是上界可为 null 的类型参数, 右侧必须是非 null 的 `Any`:

```kotlin
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y

fun main() {
    // OK
    elvisLike<String>("", "").length
    // 错误: 'null' 不能作为一个非 null 类型的值
    elvisLike<String>("", null).length

    // OK
    elvisLike<String?>(null, "").length
    // 错误: 'null' 不能作为一个非 null 类型的值
    elvisLike<String?>(null, null).length
}
```

请将语言版本设置为 `1.7`, 来启用这个功能:

Kotlin

```kotlin
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.7"
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.7'
        }
    }
}
```

关于明确非 null 类型, 详情请参见 KEEP (https://github.com/Kotlin/KEEP/blob/c72601cf35c1e95a541bb4b230edb474a6d1d1a8/proposals/definitely-non-nullable-types.md).

Kotlin/JVM

Kotlin 1.6.20 引入了以下变更:

- JVM 接口中默认方法的兼容性改进：用于接口的新的 `@JvmDefaultWithCompatibility` 注解 以及 `-Xjvm-default` 模式中的兼容性变更
- 在 JVM 后端中支持单个模块的并行编译
- 支持对函数式接口构造器的可调用引用

用于接口的新的 @JvmDefaultWithCompatibility 注解

Kotlin 1.6.20 引入了新的注解 `@JvmDefaultWithCompatibility` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-default-with-compatibility/): 这个注解和 `-Xjvm-default=all` 编译器选项一起使用, 可以为任何 Kotlin 接口中的任何非抽象成员, 在 JVM 接口中创建默认方法 ("接口中的默认方法(Default Method)" in "在 Java 中调用 Kotlin 代码").

如果已经存在客户代码使用你 Kotlin 接口, 但 Kotlin 接口没有使用 `-Xjvm-default=all` 选项编译, 那么这些客户代码可能与使用这个选项编译后的代码二进制不兼容. 在 Kotlin 1.6.20 之前, 要避免这个兼容性问题, 推荐的方案 (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-m3-generating-default-methods-in-interfaces/#JvmDefaultWithoutCompatibility) 是使用 `-Xjvm-default=all-compatibility` 模式, 并对不需要这种兼容性的接口使用 `@JvmDefaultWithoutCompatibility` 注解.

这个方案存在一些问题:

- 添加新接口时, 你很容易忘记添加注解.
- 在非公开部分中, 通常会存在比公开 API 更多的接口, 因此你不得不在你代码中的很多地方添加这个注解.

现在, 你可以使用 `-Xjvm-default=all` 模式, 并使用 `@JvmDefaultWithCompatibility` 注解标注接口. 这样你就可以向公开 API 中的所有接口一次性添加这个注解, 而且不需要对新的非公开代码使用任何注解.

关于这个新注解, 请在 这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-48217) 中留下你的反馈意见.

-Xjvm-default 模式中的兼容性变更

Kotlin 1.6.20 添加了选项, 对使用 `-Xjvm-default=all` 或 `-Xjvm-default=all-compatibility` 模式编译的模块, 可以使用默认模式(`-Xjvm-default=disable` 编译器选项)编译模块. 以前, 如果所有模块都使用 `-Xjvm-default=all` 或 `-Xjvm-default=all-compatibility` 模式, 编译也会成功. 你可以在这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-47000) 中留下你的反馈意见.

Kotlin 1.6.20 废弃了编译器选项 `-Xjvm-default` 的 `compatibility` 和 `enable` 模式. 在其它模式的描述中关于兼容性的部分也有变更, 但整体逻辑是没有变化. 详情请参见 更新后的描述 ("默认方法的兼容模式" in "在 Java 中调用 Kotlin 代码").

关于与 Java 互操作时的默认方法, 详情请参见 与 Java 互操作文档 ("接口中的默认方法(Default Method)" in "在 Java 中调用 Kotlin 代码"), 以及 这篇 blog (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-m3-generating-default-methods-in-interfaces/).

在 JVM 后端中支持单个模块的并行编译

> ⚠ 在 JVM 后端中支持单个模块的并行编译, 是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文), 而且你应该只为评估目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-46085) 提供你的反馈意见.

我们还在继续 改善新的 JVM IR 后端的编译时间 (https://youtrack.jetbrains.com/issue/KT-46768). 在 Kotlin 1.6.20 中, 我们添加了实验性的 JVM IR 后端模式, 并行的编译一个模块中的所有文件. 并行编译可以减少总的编译时间高达 15%.

要启用实验性的并行后端模式, 请使用 编译器选项 ("编译器选项" in "Kotlin 编译器选项") `-Xbackend-threads`. 对这个选项可以使用以下参数:

- `N` 是你想要使用的线程数量. 这个值不要大于你的 CPU 核数; 否则, 线程间的上下文切换会导致并行编译不会发生更多效果
- `0` 对每个 CPU 核, 使用单独的线程

Gradle (Gradle) 可以并行运行 task, 但如果从 Gradle 的观点来看, 一个项目(或一个项目的主要部分)只是一个很大的 task, 那么这种类型的并行带来的帮助不大. 如果你有非常大的单一模块, 请使用

并行编译来提高编译速度. 如果你的项目包含很多小模块, 并且由 Gradle 并行的构建, 添加另一层的并行, 可能由于上下文切换反而导致性能损失.

> 并行编译存在一些条件:
>
> - 它不能与 kapt (kapt 编译器插件) 一起工作, 因为 kapt 会禁用 IR 后端
> - 它的设计要求更多的 JVM heap 内存. heap 内存大小正比于线程数量

支持对函数式接口构造器的可调用引用

> 支持对函数式接口构造器的可调用引用, 是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文), 而且你应该只为评估目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-47939) 提供你的反馈意见.

支持对函数式接口构造器的 可调用引用 ("可调用的引用" in "反射"), 增加了一种源代码兼容的方式, 来将带构造器函数的接口迁移到 函数式接口 (函数式 (SAM) 接口).

我们来看看以下代码:

```
interface Printer {
    fun print()
}

fun Printer(block: () -> Unit): Printer = object : Printer {
override fun print() = block() }
```

有了对函数式接口构造器可调用引用, 这个代码可以替换为简单的函数式接口声明:

```
fun interface Printer {
    fun print()
}
```

它的构造器会隐含的创建, 任何使用 `::Printer` 函数引用的代码都可以正确编译. 比如:

```
documentsStorage.addPrinter(::Printer)
```

为了保持二进制兼容性, 可以对旧的函数 `Printer` 标注 `@Deprecated` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-deprecated/) 注解, 废弃级别设置为 `DeprecationLevel.HIDDEN`:

```
@Deprecated(message = "Your message about the deprecation", level =
DeprecationLevel.HIDDEN)
fun Printer(...) {...}
```

请使用编译器选项 `-XXLanguage:+KotlinFunInterfaceConstructorReference` 来启用这个功能.

Kotlin/Native

Kotlin/Native 1.6.20 继续更新了它的新组件. 我们进一步改善了 Kotlin 在各个平台的体验一致性:

- 新内存管理器的更新
- 新内存管理器中内存清理阶段的并发实现
- 注解类的实例化
- 与 Swift async/await 的交互: 返回 Swift 的 Void 类型, 而不是 KotlinUnit 类型
- 使用 libbacktrace 的更好的栈追踪信息(Stack Trace)
- 支持独立的 Android 可执行文件
- 性能改进
- cinterop 模块导入时的错误处理改进
- 支持 Xcode 13 库

新内存管理器的更新

> 新的 Kotlin/Native 内存管理器处于 Alpha (Kotlin 各部分组件的稳定性) 阶段. 未来它可能发生不兼容的变更, 并需要手动迁移. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-48525) 提供你的反馈意见.

在 Kotlin 1.6.20 中, 你可以试用新的 Kotlin/Native 内存管理器的 Alpha 版. 它消除 JVM 和 Native 平台之间的差别, 在跨平台项目中为开发者提供一致的体验. 例如, 你可以更加容易的创建新的跨平台移动应用程序, 同时工作在 Android 和 iOS 上.

新的 Kotlin/Native 内存管理器解除了在线程之间共享对象的限制. 还提供了并发编程用的, 无内存泄露的基本数据类型, 它安全, 而且不需要任何特殊的管理或注解.

新内存管理器在未来的版本中将会被默认使用, 因此我们推荐你现在就开始试用. 关于新的内存管理器, 请参见我们的 blog (https://blog.jetbrains.com/kotlin/2021/08/try-the-new-kotlin-native-memory-manager-development-preview/), 并查看示例项目, 或直接阅读 迁移指南 (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/NEW_MM.md), 自己来试用它.

请在你的项目中试用新的内存管理器, 看看它如何工作, 并在我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-48525) 提供你的反馈意见.

新内存管理器中内存清理阶段(Sweep Phase)的并发实现

如果你已经切换到了我们的新内存管理器, 它 在 Kotlin 1.6 中发布 ("新的内存管理器 (预览版)" in "Kotlin 1.6.0 版中的新功能"), 你可能会注意到显著的执行时间改善: 我们的评测显示平均改善了 35%. 从 1.6.20 开始, 对于新内存管理器的内存清理阶段(Sweep Phase)还可以使用一个并发实现. 这也能够改进性能, 减少垃圾收集器导致的程序暂停时间.

要为新的 Kotlin/Native 内存管理器启用这个功能, 请传递以下编译器选项:

```
-Xgc=cms
```

关于新内存管理器的性能, 欢迎在这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-48526) 中提供你的反馈意见.

注解类的实例化

在 Kotlin 1.6.0 中, 对 Kotlin/JVM 和 Kotlin/JS, 注解类的实例化进入 稳定版 (Kotlin 各部分组件的稳定性). 1.6.20 版本还提供对 Kotlin/Native 的支持.

详情请参见 注解类的实例化 ("创建注解类的实例" in "注解").

与 Swift async/await 的交互: 返回 Swift 的 Void 类型, 而不是 KotlinUnit 类型

> 与 Swift async/await 的并发交互能力是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-47610) 提供你的反馈意见.

我们继续改进了 与 Swift's async/await 的交互(实验性功能) ("与 Swift 5.5 async/await 的交互 (实验性功能)" in "Kotlin 1.5.30 版中的新功能") (从 Swift 5.5 开始可用). 在 Kotlin 1.6.20 中, 处理 `Unit` 返回类型的 `suspend` 函数的方式, 与以前的版本不同.

以前的版本中, 这样的函数在 Swift 中表达为 返回 `KotlinUnit` 的 `async` 函数. 但是, 正确的返回类型应该是 `Void`, 与非挂起的函数类似.

为了避免破坏已有的代码, 我们引入一个 Gradle 属性, 让编译器将返回 `Unit` 的挂起函数, 翻译为 Swift 中的 `Void` 返回类型的 `async` 函数:

```
# gradle.properties
kotlin.native.binary.unitSuspendFunctionObjCExport=proper
```

在未来的 Kotlin 发布版中, 我们计划让这个行为成为默认设置.

使用 libbacktrace 的更好的栈追踪信息(Stack Trace)

> ⚠ 使用 libbacktrace 来解析源代码位置是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-48424) 提供你的反馈意见.

Kotlin/Native 现在可以输出详细的栈追踪信息(Stack Trace), 其中包括文件位置和行号, 可以用于 `linux*` (`linuxMips32` 和 `linuxMipsel32` 除外) 和 `androidNative*` 编译目标上更好的进行错误调试.

这个功能的实现使用 libbacktrace (https://github.com/ianlancetaylor/libbacktrace) 库. 请参考以下代码, 看看具体的差别:

```
fun main() = bar()
fun bar() = baz()
inline fun baz() {
    error("")
}
```

- **在 1.6.20 以前:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0    example.kexe        0x227190       kfun:kotlin.Throwable#
<init>(kotlin.String?){} + 96
   at 1    example.kexe        0x221e4c       kfun:kotlin.Exception#
<init>(kotlin.String?){} + 92
```

```
   at 2   example.kexe        0x221f4c
kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 92
   at 3   example.kexe        0x22234c
kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} + 92
   at 4   example.kexe        0x25d708       kfun:#bar(){} + 104
   at 5   example.kexe        0x25d68c       kfun:#main(){} + 12
```

- 在 1.6.20 中，使用 libbacktrace:

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe        0x229550    kfun:kotlin.Throwable#
<init>(kotlin.String?){} + 96
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe        0x22420c    kfun:kotlin.Exception#
<init>(kotlin.String?){} + 92
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe        0x22430c
kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 92
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe        0x22470c
kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} + 92
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe        0x25fac8    kfun:#bar(){} + 104
[inlined]
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/libraries/stdlib/src/k
otlin/util/Preconditions.kt:143:56)
   at 5   example.kexe        0x25fac8    kfun:#bar(){} + 104
[inlined] (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:4:5)
   at 6   example.kexe        0x25fac8    kfun:#bar(){} + 104
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:13)
   at 7   example.kexe        0x25fa4c    kfun:#main(){} + 12
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14)
```

在 Apple 编译目标上，栈追踪信息中已经有了文件位置和行号，libbacktrace 对内联函数调用提供更多详细信息:

- **在 1.6.20 以前:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe    0x10a85a8f8    kfun:kotlin.Throwable#
<init>(kotlin.String?){} + 88
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe    0x10a855846    kfun:kotlin.Exception#
<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe    0x10a855936
kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe    0x10a855c86
kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe    0x10a8489a5    kfun:#bar(){} + 117
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:1)
   at 5   example.kexe    0x10a84891c    kfun:#main(){} + 12
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14)
...
```

- **在 1.6.20 中，使用 libbacktrace:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe    0x10669bc88    kfun:kotlin.Throwable#
<init>(kotlin.String?){} + 88
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe    0x106696bd6    kfun:kotlin.Exception#
<init>(kotlin.String?){} + 86
```

```
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe    0x106696cc6
kfun:kotlin.RuntimeException#<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe    0x106697016
kfun:kotlin.IllegalStateException#<init>(kotlin.String?){} + 86
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-
native/runtime/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe    0x106689d35    kfun:#bar(){} + 117
[inlined]
(/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/libraries/stdlib/src/k
otlin/util/Preconditions.kt:143:56)
>>  at 5   example.kexe    0x106689d35    kfun:#bar(){} + 117
[inlined] (/private/tmp/backtrace/src/commonMain/kotlin/app.kt:4:5)
   at 6   example.kexe    0x106689d35    kfun:#bar(){} + 117
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:2:13)
   at 7   example.kexe    0x106689cac    kfun:#main(){} + 12
(/private/tmp/backtrace/src/commonMain/kotlin/app.kt:1:14)
...
```

要使用 libbacktrace 输出更好的栈追踪信息, 请在 gradle.properties 中添加以下内容:

```
# gradle.properties
kotlin.native.binary.sourceInfoType=libbacktrace
```

请在 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-48424) 中, 告诉我们你使用 libbacktrace 调试 Kotlin/Native 程序的效果如何.

支持独立的 Android 可执行文件

以前, Kotlin/Native 中的 Android Native 可执行文件实际上并不是可执行文件, 而是共用的库, 你可以使用将它用作 NativeActivity. 现在有了一个选项, 可以为 Android Native 编译目标生成标准的可执行文件.

为了使用这个功能, 请在你的项目的 build.gradle(.kts) 中, 配置你的 androidNative 编译目标的 executable 代码段. 添加 the 以下 binary 选项:

```
kotlin {
    androidNativeX64("android") {
        binaries {
            executable {
                binaryOptions["androidProgramType"] = "standalone"
            }
        }
    }
}
```

注意, 在 Kotlin 1.7.0 中这个功能将成为默认设定. 如果你想要保留目前的行为, 请使用以下设置:

```
binaryOptions["androidProgramType"] = "nativeActivity"
```

感谢 Mattia Iavarone 提供的 实现 (https://github.com/jetbrains/kotlin/pull/4624)!

性能改进

我们在努力改进 Kotlin/Native 来 提升编译速度 (https://youtrack.jetbrains.com/issue/KT-42294), 改善你的开发体验.

Kotlin 1.6.20 带来了一些性能改进和 bug 修正, 影响到 Kotlin 生成的 LLVM IR. 根据我们内部项目的评测, 平均结果显示我们实现了下面的性能提升:

- 执行时间减少了 15%
- release 和 debug 二进制文件代码大小都减少了 20%
- release 二进制文件的编译时间减少了 26%

在一个大型的内部项目中, 这些变更也让 debug 二进制文件编译时间减少了 10%.

为了达到这个成果, 我们对一些编译器生成的合成对象实现了静态初始化, 改进了我们为每个函数组织 LLVM IR 的方式, 并优化了编译器缓存.

cinterop 模块导入时的错误处理改进

这个发布版改进了使用 `cinterop` 工具导入 Objective-C 模块时(通常用于 CocoaPods pod)的错误处理. 以前的版本中, 如果你在尝试使用 Objective-C 模块时发生错误(比如, 处理头文件中的编译错误), 你只能得到意义不明的错误消息, 比如 `fatal error: could not build module $name`. 我们对 `cinterop` 工具改进了这个部分, 因此你现在得到错误消息会包括更加详细的描述信息.

支持 Xcode 13 库

这个发布版对 Xcode 13 携带的库有了完全的支持. 你可以在你的 Kotlin 代码的任何地方使用这些库.

Kotlin Multiplatform

1.6.20 版中, Kotlin Multiplatform 有了以下重要更新:

- 对所有的新的跨平台项目, 现在默认支持层级结构
- Kotlin CocoaPods Gradle plugin 有了一些与 CocoaPods 集成的便利功能

对跨平台项目的层级结构支持

Kotlin 1.6.20 默认启用层级结构支持. 自从 在 Kotlin 1.4.0 中引入这个功能 ("使用层级项目结构在多个编译目标中共用代码" in "Kotlin 1.4.0 版中的新功能") 以来, 我们大大的改善了前端, 并稳定了 IDE 导入功能.

在以前的版本中, 有 2 种方法在跨平台项目中添加代码. 第 1 种是插入到平台相关的源代码集中, 这种方法只限于一个编译目标, 并且不能由其它平台重用. 第 2 种是使用一个共通源代码集, 在 Kotlin 目前支持的所有平台共用.

现在你可以在几个相似的原生编译目标中 共用源代码, 这些编译目标可以重用很多共通逻辑和第 3 方 API. 这个技术将会提供正确的默认依赖项, 并找到共用的代码中可用的 API. 以前的版本中需要使用复杂的构建设置, 而且必须使用变通办法来让 IDE 支持在多个原生编译目标共用源代码集, 这个功能消除了这些问题. 这个功能还有助于防止使用那些本来应该用于不同的编译目标的不安全的 API.

这个技术对于 库作者 也很方便, 因为层级项目结构允许他们对一部分编译目标发布和使用带有共通 API 的库.

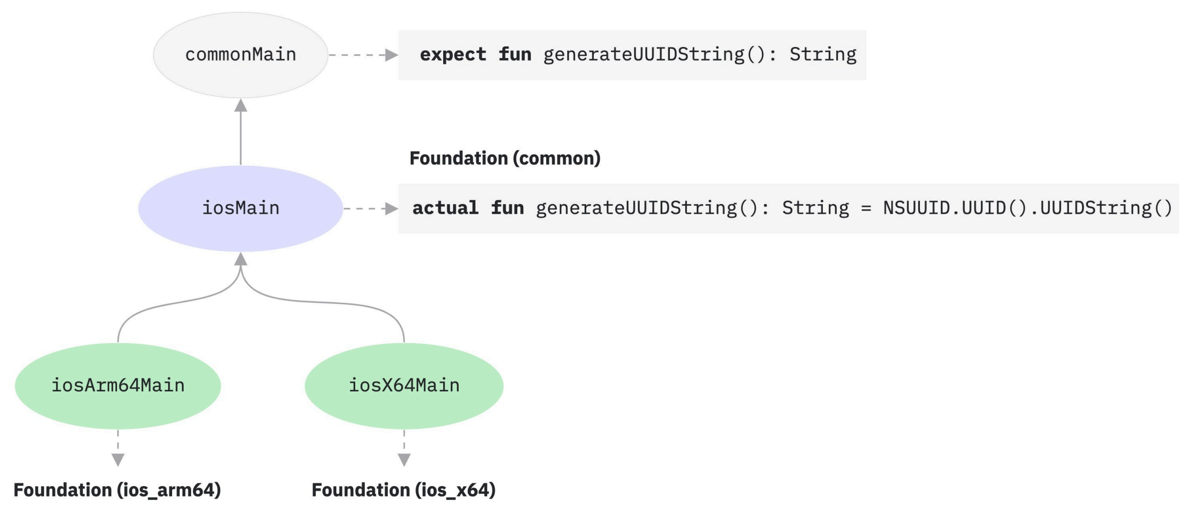
默认情况下, 使用层级项目结构发布的库只兼容于层级结构的项目.

在你的项目中更好的共用代码

没有层级结构支持, 就没有直接的方法在 *一部分* 而不是在 *所有* Kotlin 编译目标 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 中共用代码. 一个常见的例子是, 对所有的 iOS 编译目标共用代码, 并使用 iOS 专有的 依赖项 ("连接平台相关的库" in "在不同的平台之间共用代码"), 比如 Foundation.

感谢层级项目结构, 你现在可以直接达到这个目的. 在新的结构中, 源代码集组成一个层级结构. 你可以使用平台专有的语言功能, 以及一个源代码集所属的每个编译目标可用的依赖项.

例如, 假设有一个典型的跨平台项目, 带有 2 个编译目标 — iosArm64 和 iosX64, 分别用于 iOS 设备和模拟器. Kotlin 工具会理解, 2 个编译目标都拥有相同的函数, 并允许你从公共的源代码集, iosMain, 访问这些函数.



iOS 源代码层级结构示例

Kotlin 工具链会提供正确的默认依赖项, 比如 Kotlin/Native 标准库, 或原生库. 而且, Kotlin 工具会尽量查找共用的代码中可用的正确的 API 接口. 这样可以防止不正确的情况, 例如, 在针对 Windows 的共用代码中使用 macOS 专有的函数.

库作者的更多选择

在跨平台库发布之后, 它的共用源代码集的 API 现在也会和它一起正确的发布, 并可以供库的用户使用. 而且, Kotlin 工具链会自动判断出在库使用者的源代码集中能够使用哪些 API , 并密切注意不安全的使用, 比如在 JS 代码中使用针对 JVM 的 API. 详情请参见 在库中共用代码 ("在多个库之间共用代码" in "在不同的平台之间共用代码").

配置与设置

从 Kotlin 1.6.20 开始, 你所有的新的跨平台项目都将使用层级项目结构. 不需要额外的设置.

- 如果你已经进行了 手工转换 ("在类似的平台上共用代码" in "在不同的平台之间共用代码"), 你可以从 gradle.properties 中删除废弃的选项:

  ```
  # gradle.properties
  kotlin.mpp.enableGranularSourceSetsMetadata=true
  kotlin.native.enableDependencyPropagation=false // 或 'true', 取决于你以前的设置
  ```

* 对于 Kotlin 1.6.20, 我们建议使用 Android Studio 2021.1.1 (https://developer.android.com/studio) (Bumblebee) 或更高版本, 以获得最好的开发体验.
* 你可以也选择性禁用(opt out)这个功能. 要禁用层级结构支持, 请在 gradle.properties 中设置以下选项:

```
# gradle.properties
kotlin.mpp.hierarchicalStructureSupport=false
```

提供你的反馈意见

这是对整个生态系统的一个重大变更. 我们期望你能提供反馈意见, 帮助我们继续完善这个功能.

请开始试用这个功能, 并向 我们的问题追踪系统 (https://kotl.in/issue) 报告你遇到的任何问题.

Kotlin CocoaPods Gradle plugin

为了简化与 CocoaPods 的集成, Kotlin 1.6.20 发布了以下功能:

* CocoaPods plugin 现在有了 task, 可以对所有已注册的编译目标构建 XCFramework, 并生成 Podspec 文件. 当你不想直接与 Xcode 集成, 但想要构建 artifact 并部署到你的本地 CocoaPods 仓库, 这个功能可以很便利.

  详情请参见 构建 XCFramework ("构建 XCFramework" in "构建最终的原生二进制文件").
* 如果在你的项目中使用 CocoaPods 集成 (CocoaPods 概述与设置), 过去你需要对整个 Gradle 项目指定需要的 Pod 版本. 现在有了更多选择:
  * 在 cocoapods 代码块中直接指定 Pod 版本
  * 继续使用 Gradle 项目版本

  如果这些属性都没有配置, 会出现错误.
* 你现在可以在 cocoapods 代码块中配置 CocoaPod 名称, 而不需要修改整个 Gradle 项目的名称.
* CocoaPods plugin 引入了新的 extraSpecAttributes 属性, 你可以使用它来配置 Podspec 文件中的属性, 以前这些属性必须硬编码, 比如 libraries 或 vendored_frameworks.

```
kotlin {
    cocoapods {
        version = "1.0"
```

```
        name = "MyCocoaPod"
        extraSpecAttributes["social_media_url"] =
'https://twitter.com/kotlin'
        extraSpecAttributes["vendored_frameworks"] =
'CustomFramework.xcframework'
        extraSpecAttributes["libraries"] = 'xml'
    }
}
```

关于 Kotlin CocoaPods Gradle plugin 的完整信息, 请参见 DSL 参考文档 (CocoaPods Gradle plugin DSL 参考文档).

Kotlin/JS

在 1.6.20 中, Kotlin/JS 的改进主要涉及 IR 编译器:

- 对开发阶段二进制文件的增量编译 (IR)
- 默认对顶级属性(Top-Level Property)延迟初始化(Lazy initialization) (IR)
- 默认对项目模块输出单独的 JS 文件 (IR)
- Char 类优化 (IR)
- 导出功能的改进 (IR 后端和旧后端)
- 对异步的测试确保 @AfterTest

IR 编译器对开发阶段二进制文件的增量编译

为了提高使用 IR 编译器时的 Kotlin/JS 开发效率, 我们引入了新的 *增量编译* 模式.

在这个模式下, 使用 `compileDevelopmentExecutableKotlinJs` Gradle task 构建 **开发阶段二进制文件** 时, 编译器会在模块层级缓存前一次编译的结果. 它会在后续的编译中对未变更的源代码文件使用缓存的编译结果, 让编译更加快速, 尤其是对小的变更. 注意, 这个改进仅仅针对开发阶段(缩短 编辑-构建-调试 循环的时间), 而不会影响产品 artifact 的构建.

要对开发阶段二进制文件启用增量编译, 请向项目的 `gradle.properties` 文件添加以下内容:

```
# gradle.properties
kotlin.incremental.js.ir=true // 默认为 false
```

在我们的测试项目中, 新模式让增量编译的速度提高了 30%. 但是, 这个模式下的完整构建变得更慢, 因为需要创建和生成缓存.

请在你的 Kotlin/JS 项目中使用增量编译功能, 并在 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-50203) 中向我们提供你的反馈意见.

IR 编译器默认对顶级属性(Top-Level Property)延迟初始化(Lazy initialization)

在 Kotlin 1.4.30 中, 我们发布了 JS IR 编译器中 对顶级属性延迟初始化 ("顶级属性(top-level property)的延迟初始化(Lazy initialization)" in "Kotlin 1.4.30 版中的新功能") 功能的原型. 在应用程序启动时不再需要初始化所有属性, 因此延迟初始化可以缩短启动时间. 在一个真实的 Kotlin/JS 应用程序, 我们的评测结果是速度提升了大约 10%.

现在, 对这个机制进行改进和完善的测试之后, 我们在 IR 编译器中, 将顶级属性的延迟初始化作为默认模式.

```
// 延迟初始化
val a = run {
    val result = // 假设这里是一段计算密集的代码
        println(result)
    result
} // 直到变量初次使用时才会执行 run
```

如果由于某些原因你需要(在应用程序启动阶段)提早初始化一个属性, 可以对它标注 `@EagerInitialization` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native/-eager-initialization/) 注解.

IR 编译器默认对项目模块输出单独的 JS 文件

以前的版本中, JS IR 编译器可以为项目模块 生成单独的 `.js` 文件 (https://youtrack.jetbrains.com/issue/KT-44319). 默认选项是 – 对整个项目生成单个 `.js` 文件. 这个文件可能会非常巨大, 不便于使用, 因为如果你想要使用你的项目的一个函数, 你不得不将整个 JS 文件作为依赖项. 生成多个文件可以提高灵活性, 减少这些依赖项的大小. 这个功能可以通过 `-Xir-per-module` 编译器选项来使用.

从 1.6.20 开始, JS IR 编译器默认为项目模块生成单独的 `.js` 文件.

编译项目为单个的 `.js` 文件, 现在可以通过以下 Gradle 属性来使用:

```
# gradle.properties
kotlin.js.ir.output.granularity=whole-program // 默认值为 `per-
module`
```

在以前的版本中, 实验性的 per-module 模式 (可以通过 `-Xir-per-module=true` 选项启用)会在每个模块中调用 `main()` 函数. 这种行为与通常的单独 `.js` 模式不一致. 从 1.6.20 开始, 对这两种情况, `main()` 函数都只会在 main 模块中调用. 如果你确实需要在模块装载时运行某些代码, 你可以使用顶级属性(Top-Level Property), 并标注 `@EagerInitialization` 注解. 参见 默认对顶级属性(Top-Level Property)延迟初始化(Lazy initialization) (IR).

Char 类优化

`Char` 类现在由 Kotlin/JS 编译器处理, 不产生装箱(boxing)处理(类似于 内联类 (内联的值类(Inline value class))). 这样可以提高 Kotlin/JS 代码中对字符操作的速度.

除了性能改进之外, 这个功能还变更了 `Char` 输出到 JavaScript 的方式: 它现在被翻译为 `Number`.

导出功能的改进, 对 TypeScript 声明生成的改进

Kotlin 1.6.20 带来了很多修正, 并改进了导出机制(`@JsExport` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/-js-export/) 注解), 包括 TypeScript 声明 (`.d.ts`) 的生成 ("预览: 生成 TypeScript 声明文件 (d.ts)" in "使用 IR 编译器"). 我们添加了导出接口和枚举的功能, 我们还修正了以前报告给我们的, 某些边界情况下的不正确的导出行为. 详情请参见 YouTrack 中导出功能的改进 (https://youtrack.jetbrains.com/issues?q=Project:%20Kotlin%20issue%20id:%20KT-45434,%20KT-44494,%20KT-37916,%20KT-43191,%20KT-46961,%20KT-40236).

详情请参见 在 JavaScript 中使用 Kotlin 代码 (在 JavaScript 中使用 Kotlin 代码).

对异步的测试确保 @AfterTest

Kotlin 1.6.20 确保 `@AfterTest` (https://kotlinlang.org/api/latest/kotlin.test/kotlin.test/-after-test/) 函数 能够与 Kotlin/JS 异步的测试一同正确工作. 如果一个测试函数的返回类型静态的解析为 `Promise` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/-promise/), 编译器现在能够将 `@AfterTest` 函数的执行调度到对应的 `then()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/-promise/then.html) 回调.

安全性

Kotlin 1.6.20 包含了一些功能, 改进你的代码的安全性:

- 在 klibs 中使用相对路径
- 对 Kotlin/JS Gradle 项目保持 yarn.lock 文件
- 默认使用 `--ignore-scripts` 安装 npm 依赖项

在 klibs 中使用相对路径

一个 klib 格式的库 包含 ("库文件的格式" in "Kotlin/Native 库") 源代码文件的序列化后的 IR 表达, 其中包含文件路径, 用于生成正确的调试信息. 在 Kotlin 1.6.20 以前, 保存的文件路径是绝对路径. 由于库作者可能不希望公开他们的绝对路径, 1.6.20 版本引入了一个替代选项.

如果你正在发布一个 klib, 并且希望在 artifact 中只使用源代码文件的相对路径, 现在你可以传递 -Xklib-relative-path-base 编译器选项, 参数是 一个或多个源代码文件基准路径:

Kotlin

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinCompile::class).configureEach {
    // $base 是源代码文件的基准路径
    kotlinOptions.freeCompilerArgs += "-Xklib-relative-path-base=$base"
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinCompile).configureEach {
    kotlinOptions {
        // $base 是源代码文件的基准路径
        freeCompilerArgs += "-Xklib-relative-path-base=$base"
    }
}
```

对 Kotlin/JS Gradle 项目保持 yarn.lock 文件

> 这个功能也被反向导入到 Kotlin 1.6.10.

Kotlin/JS Gradle plugin 现在提供了保持 yarn.lock 文件的功能, 因此可以为你的项目锁定 npm 依赖项的版本, 而不需要额外的 Gradle 配置. 这个功能修改了默认的项目结构, 在项目的根目录下添加了自动生成的 kotlin-js-store 目录. 这个目录内保存 yarn.lock 文件.

我们强烈建议将 kotlin-js-store 目录及其内容提交到你的版本控制系统. 将这个锁文件提交到你的版本控制系统是一种 推荐的实践(Recommended Practice) (https://classic.yarnpkg.com/blog/2016/11/24/lockfiles-for-all/), 因为可以保证你的应用程序在所有机器上都使用完全相同的依赖项树进行构建, 无论是在其他机器上的开发环境中, 还是在 CI/CD 服务中. 当项目在一台新机器上 check out 时, 锁文件也可以防止你的 npm 依赖项被静悄悄的更新, 这样会导致安全性问题.

Dependabot (https://github.com/dependabot) 之类的工具可以也解析你的 Kotlin/JS 项目的 yarn.lock 文件, 如果你依赖的任何 npm 包存在安全问题, 它会向你提示警告.

如果需要, 你可以在构建脚本中变更目录和锁文件的名称:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin> {

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.Yarn
RootExtension>().lockFileDirectory =
        project.rootDir.resolve("my-kotlin-js-store")

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.Yarn
RootExtension>().lockFileName = "my-yarn.lock"
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).lockFileDirectory =
        file("my-kotlin-js-store")

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).lockFileName = 'my-yarn.lock'
}
```

> ⚠ 修改 lock 文件名称, 可能会导致依赖项检查工具不再正确读取这个文件.

默认使用 --ignore-scripts 安装 npm 依赖项

> ℹ 这个功能也被反向导入到 Kotlin 1.6.10.

Kotlin/JS Gradle plugin 在安装 npm 依赖项时, 现在默认会阻止执行 Life Cycle 脚本 (https://docs.npmjs.com/cli/v8/using-npm/scripts#life-cycle-scripts). 这个变更的目的是, 如果使用了存在安全问题的 npm 包, 可以减少执行恶意代码的可能性.

如果要回滚到旧的配置, 你可以明确的允许 Life Cycle 脚本执行, 方法是向 `build.gradle(.kts)` 文件添加以下设置:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin> {

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.Yarn
RootExtension>().ignoreScripts = false
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).ignoreScripts = false
}
```

详情请参见 Kotlin/JS Gradle 项目的 npm 依赖项 ("npm 依赖项目" in "创建 Kotlin/JS 工程 (Project)").

Gradle

Kotlin 1.6.20 包含对 Kotlin Gradle Plugin 的以下变更 :

- 新的 属性 `kotlin.compiler.execution.strategy` 和 `compilerExecutionStrategy` 用于定义 Kotlin 编译器执行策略
- 废弃选项 `kapt.use.worker.api`, `kotlin.experimental.coroutines`, 和 `kotlin.coroutines`
- 删除构建选项 `kotlin.parallel.tasks.in.project`

用于定义 Kotlin 编译器执行策略的属性

在 Kotlin 1.6.20 之前, 你可以使用系统属性 `-Dkotlin.compiler.execution.strategy` 来定义 Kotlin 编译器执行策略. 这个属性对于某些情况可以很便利. Kotlin 1.6.20 引入一个相同名称的 Gradle 属性, `kotlin.compiler.execution.strategy`, 以及编译 task 属性 `compilerExecutionStrategy`.

系统属性继续起作用, 但在未来的发布版本中会被删除.

目前的属性优先度如下:

- task 属性 `compilerExecutionStrategy` 优先度高于系统属性和 Gradle 属性 `kotlin.compiler.execution.strategy`.
- Gradle 属性优先度高于系统属性.

有 3 种编译器执行策略, 你可以赋值给这些属性:

策略	Kotlin 编译器在哪里执行	增量编译	其它特征
Daemon	在 Kotlin 自己的 daemon 进程之内	是	*默认策略*. 可以在不同的 Gradle daemon 之间共用
In process	在 Gradle daemon 进程之内	否	可以与 Gradle daemon 共用 heap
Out of process	对每个编译都在单独的进程内	否	—

相应的, 对于 (系统属性和 Gradle 属性) `kotlin.compiler.execution.strategy`, 可以设置的值是:

1. `daemon` (默认)
2. `in-process`
3. `out-of-process`

在 `gradle.properties` 中, 使用 Gradle 属性 `kotlin.compiler.execution.strategy`:

```
# gradle.properties
kotlin.compiler.execution.strategy=out-of-process
```

对于 task 属性 `compilerExecutionStrategy`, 可以设置的值是:

1. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.DAEMON` (默认)
2. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.IN_PROCESS`
3. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.OUT_OF_PROCESS`

在 `build.gradle.kts` 构建脚本中, 使用 task 属性 `compilerExecutionStrategy`:

```
import org.jetbrains.kotlin.gradle.dsl.KotlinCompile
import
org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType<KotlinCompile>().configureEach {

compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN_PRO
CESS)
}
```

请在 这个 YouTrack task (https://youtrack.jetbrains.com/issue/KT-49299)中提供你的反馈意见.

废弃用于 kapt 和 coroutines 的构建选项

在 Kotlin 1.6.20 中, 我们修改了这些属性的废弃级别 :

- 我们废弃了使用 kapt.use.worker.api 来通过 Kotlin daemon 运行 kapt (kapt 编译器插件) 的功能 – 现在这个选项会在 Gradle 的输出中产生一条警告信息. 默认情况下, 从 1.3.70 版开始 kapt 使用 Gradle worker ("并行运行多个 KAPT 任务" in "kapt 编译器插件"), 我们建议继续使用这种方法.

  我们将会在未来的发布版中删除选项 kapt.use.worker.api.

- 我们废弃了在 gradle.properties 中使用的 Gradle DSL 选项 kotlin.experimental.coroutines 和属性 kotlin.coroutines. 请直接使用 *挂起函数*, 或向你的 build.gradle(.kts) 文件 添加 kotlinx.coroutines 依赖项 ("设置对 kotlinx 库的依赖项" in "配置 Gradle 项目").

  关于协程, 详情请参见 协程指南 (协程指南).

删除构建选项 kotlin.parallel.tasks.in.project

在 Kotlin 1.5.20 中, 我们 废弃了构建选项 kotlin.parallel.tasks.in.project ("废弃 kotlin.parallel.tasks.in.project 属性" in "Kotlin 1.5.20 版中的新功能"). 在 Kotlin 1.6.20 中, 这个选项已被删除.

根据项目不同, 在 Kotlin daemon 中的并行编译可能需要更多的内存. 为了减少内存消耗, 请 对 Kotlin daemon 增加 heap 大小 ("设置 Kotlin daemon 的 JVM 参数" in "Kotlin Gradle plugin 中的编译与缓存").

详情请参见, 在 Kotlin Gradle plugin 中 目前支持的编译器选项 (Kotlin Gradle plugin 中的编译器选项).

Kotlin 1.6.0 版中的新功能

最终更新: 2024/09/10

发布日期: 2021/11/16 (["各发布版详情" in "Kotlin 的发布版本"])

Kotlin 1.6.0 引入了新的语言功能, 优化并改进了现有的功能, 并对 Kotlin 标准库进行了大量的改进.

关于这个版本的变更概要, 也可以查看 release blog (https://blog.jetbrains.com/kotlin/2021/11/kotlin-1-6-0-is-released/).

语言

在以前的 1.5.30 版中引入的一些语言功能的预览版, 在 Kotlin 1.6.0 中已变为稳定版:

- 对 enum, 封闭类 和 Boolean 值的穷尽式(exhaustive) when 语句 (稳定版)
- 挂起函数用作超类型 (稳定版)
- 挂起转换 (稳定版)
- 注解类的实例化 (稳定版)

此外还包括类型推断的各种改进, 并对类的类型参数支持注解:

- 改进了对递归泛型类型的类型推断
- 对构建器推断的变更
- 对类的类型参数支持注解

对 enum, 封闭类 和 Boolean 值的穷尽式(exhaustive) when 语句 (稳定版)

穷尽式(exhaustive) `when` (["when 表达式" in "条件与循环"]) 语句 对它的参数的所有可能的类型或值, 都包含对应的分支, 或者对于某些类型还存在一个 `else` 分支. 它可以覆盖所有可能的情况, 使你的代码更加安全.

我们很快会禁止使用非穷尽式的(non-exhaustive) `when` 语句, 使 `when` 语句的行为与 `when` 表达式保持一致. 为保证平滑的迁移, Kotlin 1.6.0 会对使用 enum, 封闭类, 或 Boolean 值的非穷尽式(non-exhaustive) `when` 语句报告警告. 这些警告在未来的发布版中将会变成错误.

```
sealed class Contact {
    data class PhoneCall(val number: String) : Contact()
    data class TextMessage(val number: String) : Contact()
}

fun Contact.messageCost(): Int =
    when(this) { // 错误: 'when' 表达式必须穷尽所有条件
        is Contact.PhoneCall -> 42
    }

fun sendMessage(contact: Contact, message: String) {
    // 从 1.6.0 开始

    // 警告: 对于 Boolean 值, 非穷尽式的 'when' 语句
    // 在 1.7 中将被禁止, 请添加 'false' 分支或 'else' 分支
    when(message.isEmpty()) {
        true -> return
    }
    // 警告: 对 封闭类/接口, 非穷尽式的 'when' 语句
    // 在 1.7 中将被禁止, 请添加 'is TextMessage' 分支或 'else' 分支
    when(contact) {
        is Contact.PhoneCall -> TODO()
    }
}
```

关于这个变更及其影响, 详细的解释请参见 这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-47709).

挂起函数用作超类型 (稳定版)

在 Kotlin 1.6.0 中, 挂起函数类型的实现已经成为 稳定版 (Kotlin 各部分组件的稳定性). 在 1.5.30 中 ("挂起函数用作超类型" in "Kotlin 1.5.30 版中的新功能") 曾发布过预览版.

在设计 API 时, 如果使用 Kotlin coroutine 并接受挂起函数类型, 功能会很有用. 将需要的行为封装到一个单独的类中, 并让这个类实现挂起函数类型, 这样你可以将你的代码变成流式风格.

```
class MyClickAction : suspend () -> Unit {
    override suspend fun invoke() { TODO() }
}
```

```
fun launchOnClick(action: suspend () -> Unit) {}
```

以前只能使用 lambda 表达式和挂起函数引用的地方, 现在你也可以使用这个类的实例了: `launchOnClick(MyClickAction())`.

这个功能的实现细节目前还存在两个限制:

- 在类的超类型列表中, 你不能混用通常的函数类型和挂起函数类型.
- 你不能使用多个挂起函数超类型.

挂起转换 (稳定版)

从通常函数到挂起函数类型的转换功能, Kotlin 1.6.0 引入了 稳定版 (Kotlin 各部分组件的稳定性). 从 1.4.0 开始, 这个功能支持只函数字面值和可调用的引用. 从 1.6.0 开始, 这个功能支持支持任意形式的表达式. 在调用参数需要挂起函数的地方, 现在你可以传递值为通常函数类型的任意表达式. 编译器将会自动进行一个隐含的转换.

```
fun getSuspending(suspending: suspend () -> Unit) {}

fun suspending() {}

fun test(regular: () -> Unit) {
    getSuspending { }           // OK
    getSuspending(::suspending) // OK
    getSuspending(regular)      // OK
}
```

注解类的实例化 (稳定版)

Kotlin 1.5.30 对 JVM 平台 引入了 ("创建注解类的实例" in "Kotlin 1.5.30 版中的新功能") 注解类实例化功能的实验性支持. 从 1.6.0 开始, 这个功能对 Kotlin/JVM 和 Kotlin/JS 平台都默认启用.

关于注解类的实例化, 更多详情请参见这个 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/annotation-instantiation.html).

改进了对递归泛型类型的类型推断

Kotlin 1.5.30 改进了对递归泛型类型的类型推断, 能够根据对应的类型参数的上界(Upper Bound)来推断类型参数. 这个功能可以通过编译器选项启用. 在 1.6.0 和之后的版本中, 这个功能默认启用.

```
// 在 1.5.30 版以前
val containerA = PostgreSQLContainer<Nothing>
(DockerImageName.parse("postgres:13-alpine")).apply {
  withDatabaseName("db")
  withUsername("user")
  withPassword("password")
  withInitScript("sql/schema.sql")
}

// 在 1.5.30 版中使用编译器选项, 或从 1.6.0 版开始默认启用
val containerB =
PostgreSQLContainer(DockerImageName.parse("postgres:13-alpine"))
  .withDatabaseName("db")
  .withUsername("user")
  .withPassword("password")
  .withInitScript("sql/schema.sql")
```

对构建器推断的变更

构建器推断是一种类型推断, 在调用泛型的构建器函数时很有用. 对于一个构建器函数调用, 根据它的 lambda 表达式参数内部的函数调用的类型信息, 可以推断出构建器函数的类型参数.

我们做了很多改变, 使得构建器推断接近完全稳定. 从 1.6.0 开始:

- 在构建器的 lambda 表达式内, 你可以调用一个返回值实例类型还未推断的函数, 而不需要指定 1.5.30 版中引入 ("去掉了构建器推断的限制" in "Kotlin 1.5.30 版中的新功能")的 `-Xunrestricted-builder-inference` 编译器选项.
- 使用 `-Xenable-builder-inference`, 你可以编写你自己的构建器, 而不需要添加 `@BuilderInference` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-builder-inference/) 注解.

  > 注意, 这些构建器的使用者也同样需要指定 `-Xenable-builder-inference` 编译器选项.

- 使用 `-Xenable-builder-inference`, 如果通常的类型推断无法得到足够的类型信息, 会自动激活构建器推断.

参见如何编写自定义的泛型构建器 (通过构建器类型推断(Builder Type Inference)使用构建器).

对类的类型参数支持注解

对类的类型参数支持注解, 如下:

```
@Target(AnnotationTarget.TYPE_PARAMETER)
annotation class BoxContent

class Box<@BoxContent T> {}
```

类型参数上的所有注解都会被编译输出到 JVM 字节码, 因此注解处理器能够使用这些注解.

关于这种使用场景的动机, 请参见这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-43714).

更多详情请参见 注解 (注解).

对旧版本的 API 提供更长时期的支持

从 Kotlin 1.6.0 开始, 我们将会支持 3 个版本之前的 API, 而不是 2 个版本, 再加上当前的稳定版 API. 目前我们支持 1.3, 1.4, 1.5 版, 以及 1.6 版.

Kotlin/JVM

对于 Kotlin/JVM, 从 1.6.0 开始, 编译器使用 JVM 17 字节码版本生成类. 新的语言版本还包括代理属性优化和可重复注解, 我们已将这些功能添加到路线图中:

- 对于 1.8 JVM 编译目标, 在运行期保留的可重复注解
- 代理属性优化, 不再在 KProperty 实例上调用 get/set 方法

对于 1.8 JVM 编译目标, 在运行期保留的可重复注解

Java 8 引入了 可重复的注解 (https://docs.oracle.com/javase/tutorial/java/annotations/repeating.html), 对单个代码元素, 这种注解可以使用多次. 这个功能要求在 Java 代码中出现两个声明: 可重复注解本身, 标注 `@java.lang.annotation.Repeatable` (https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Repeatable.html), 以及容器注解(containing annotation), 来表达可重复注解的值.

Kotlin 也有可重复注解, 但只需要在一个注解的声明中标注 `@kotlin.annotation.Repeatable` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-repeatable/) 来将它变成可重

复注解. 在 1.6.0 版之前, 这个功能只支持 `SOURCE` retention 设置, 而且不兼容 Java 的可重复注解. Kotlin 1.6.0 解除了这些限制. `@kotlin.annotation.Repeatable` 现在可以接受任意的 retention 设置, 而且这些注解在 Kotlin 和 Java 中都可重复. 在 Kotlin 代码中, 现在也支持 Java 的可重复注解.

尽管你可以声明一个容器注解(containing annotation), 但并不是必须的. 比如:

- 如果一个注解 `@Tag` 标注了 `@kotlin.annotation.Repeatable`, Kotlin 编译器会自动生成容器注解(containing annotation)类, 使用名称 `@Tag.Container`:

  ```
  @Repeatable
  annotation class Tag(val name: String)

  // 编译器会生成容器注解(containing annotation) @Tag.Container
  ```

- 要对容器注解(containing annotation)设置自定义的名称, 可以标注 `@kotlin.jvm.JvmRepeatable` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvmrepeatable/) 元注解, 并通过参数明确指定要使用的容器注解类:

  ```
  @JvmRepeatable(Tags::class)
  annotation class Tag(val name: String)

  annotation class Tags(val value: Array<Tag>)
  ```

Kotlin 反射通过新的函数 `KAnnotatedElement.findAnnotations()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect.full/find-annotations.html), 现在同时支持 Kotlin 和 Java 的可重复注解.

关于 Kotlin 可重复注解, 更多详情请参见 这个 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/repeatable-annotations.html).

代理属性优化, 不再在 KProperty 实例上调用 get/set 方法

我们优化了生成的 JVM 字节码, 省略了 `$delegate` field, 改为直接访问引用的属性.

比如, 在下面的代码中

```
class Box<T> {
    private var impl: T = ...
```

```
    var content: T by ::impl
}
```

Kotlin 不再生成 `content$delegate` field. 对 `content` 属性的访问会直接调用 `impl`, 略过代理属性的 `getValue`/`setValue` 操作, 因此不再需要使用`KProperty` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-property/index.html) 类型的属性引用对象.

感谢我们的 Google 同事实现了这个功能!

更多详情请参见 代理属性 (委托属性).

Kotlin/Native

Kotlin/Native 有了很多改进, 以及组件更新, 其中一部分还处于预览状态:

- 新的内存管理器 (预览版)
- 支持 Xcode 13
- 在任意的主机上编译到 Windows 编译目标
- LLVM 和链接器更新
- 性能改进
- JVM 和 JS IR 后端统一的编译器 plugin ABI
- 对 klib 链接错误提供详细的错误信息
- 重新设计了对未处理异常进行处理的 API

新的内存管理器 (预览版)

新的 Kotlin/Native 内存管理器还处于 实验阶段 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 使用这个功能需要明确要求使用者同意(详情请见下文), 而且你应该只用来进行功能评估, 不要用在你的正式产品中. 希望你能通过我们的 [问题追踪系统] YouTrack (https://youtrack.jetbrains.com/issue/KT-48525) 提供你的反馈意见.

在 Kotlin 1.6.0 中, 你可以试用新的 Kotlin/Native 内存管理器的开发预览版. 这个新的内存管理器可以帮助我们进一步消除 JVM 和 Native 平台之间的差异, 在跨平台项目中提供一致的开发体验.

一个值得注意的变化是, 顶级属性(top-level property)的延迟初始化(Lazy initialization), 和在 Kotlin/JVM 平台一样. 当同一个源代码文件中的顶级属性或函数第一次被访问时, 顶级属性才会被初始化. 这个模式还包括全局的过程间优化(interprocedural optimization)(只对正式发布版的二进制文件开启), 这个优化会删除多余的初始化检查.

关于新的内存管理器, 我们最近发布了一篇 blog (https://blog.jetbrains.com/kotlin/2021/08/try-the-new-kotlin-native-memory-manager-development-preview/). 请阅读这篇 blog, 可以了解新内存管理器的当前开发状态, 看到一些演示项目, 或者直接访问 迁移指南 (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/NEW_MM.md) 来试用这个功能. 请检查新内存管理器在你的项目中工作状况如何, 并向我们的问题追踪系统 YouTrack (https://youtrack.jetbrains.com/issue/KT-48525) 提交你的反馈意见.

支持 Xcode 13

Kotlin/Native 1.6.0 支持 Xcode 13 – Xcode 的最新版本. 你可以自由地更新你的 Xcode, 并继续针对 Apple 操作系统开发你的 Kotlin 项目.

> Xcode 13 中新添加的库在 Kotlin 1.6.0 中不可用, 但我们将在后面的版本中支持使用这些库.

在任意的主机上编译到 Windows 编译目标

从 1.6.0 开始, 你不需要 Windows 主机来编译到 Windows 编译目标 `mingwX64` 和 `mingwX86`. 这些编译目标可以在任何支持 Kotlin/Native 的主机上编译.

LLVM 和链接器更新

我们重新设计了 Kotlin/Native 底层使用的 LLVM 依赖项目. 这个变化带来了很多好处, 包括:

- LLVM 版本更新到 11.1.0.
- 减少依赖项目大小. 比如, 在 macOS 上现在是大约 300 MB, 而在以前的版本中是 1200 MB.
- 删除了对 `ncurses5` 库的依赖 (https://youtrack.jetbrains.com/issue/KT-42693), 在现代的 Linux 发布版中这个库已经不可用了.

除 LLVM 的更新之外, 对于 MingGW 编译目标, Kotlin/Native 现在使用 LLD (https://lld.llvm.org/) 链接器(来自 LLVM 项目的链接器). 与以前使用的 ld.bfd 链接器相比, 这个变化带来很多好处, 使得我们可以改善生成的二进制代码的运行期性能, 并对 MinGW 编译目标支持编译器缓存. 注意, LLD 需要引入用于 DLL 链接的库 ("对 MinGW 编译目标废弃无导入库的 DLL

链接" in "Kotlin 1.5.30 版中的新功能"). 更多详情请参见 这条 Stack Overflow 讨论主题 (https://stackoverflow.com/questions/3573475/how-does-the-import-library-work-details/3573527/#3573527).

性能改进

Kotlin/Native 1.6.0 带来了以下性能改进:

- 编译期: 对于 `linuxX64` 和 `iosArm64` 编译目标, 默认启用编译器缓存. 对于 debug 模式下的大多数编译, 可以提高速度(第一次编译除外). 在我们的测试项目中, 测量显示速度提升了大约 200%. 对于这些编译目标, 从 Kotlin 1.5.0 开始, 使用 额外的 Gradle 属性 ("性能改善" in "Kotlin 1.5.0 版中的新功能") 可以启用编译器缓存; 你现在可以删除这些 Gradle 属性.
- 运行期: 由于对生成的 LLVM 代码进行了优化, 使用 `for` 循环在数组上进行迭代现在速度最大可以提升 12%.

JVM 和 JS IR 后端统一的编译器 plugin ABI

对 Kotlin/Native 使用共通的 IR 编译器 plugin ABI 的选项还处于 实验阶段 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 使用这个功能需要明确要求使用者同意 (详情请见下文), 而且你应该只用来进行功能评估, 不要用在你的正式产品中. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-48595)提供你的反馈意见.

在之前的版本中, 由于 ABI 的不同, 编译器 plugin 的开发者必须为 Kotlin/Native 提供单独的 artifact.

从 1.6.0 开始, 对 Kotlin/Native, Kotlin 跨平台 Gradle plugin 可以使用内嵌的编译器 jar – 与 JVM 和 JS IR 后端使用的相同. 这是朝向统一编译器 plugin 开发体验的一步, 因为你现在可以对 Native 和其他平台使用相同的编译器 plugin artifact.

这个功能目前还是预览版, 需要使用者明确同意. 要对 Kotlin/Native 使用通用的编译器 plugin artifact, 请在 `gradle.properties` 添加以下内容: `kotlin.native.useEmbeddableCompilerJar=true`.

我们计划将来对 Kotlin/Native 默认使用内嵌的编译器 jar, 因此这个预览功能在你的环境工作状况如何, 你的反馈意见对于我们来说非常重要.

如果你是编译器 plugin 的开发者, 请试用这个模式, 检查它对你的是否正常工作. 注意, 根据你的 plugin 的结构不同, 可能会需要手工的迁移步骤. 关于迁移指南, 请参见 这个 YouTrack issue

(https://youtrack.jetbrains.com/issue/KT-48595), 并在评论中留下你的反馈意见.

对 klib 链接错误提供详细的错误信息

Kotlin/Native 编译器现在会对 klib linkage 错误提供详细的错误信息. 错误信息现在带有清楚的错误描述, 还包括可能的错误原因, 以及如何修复.

比如:

- 1.5.30 版的错误信息:

```
e: java.lang.IllegalStateException: IrTypeAliasSymbol expected:
Unbound public symbol for public
kotlinx.coroutines/CancellationException|null[0]
<stack trace>
```

- 1.6.0 版的错误信息:

```
e: The symbol of unexpected type encountered during IR
deserialization: IrClassPublicSymbolImpl,
kotlinx.coroutines/CancellationException|null[0].
IrTypeAliasSymbol is expected.

This could happen if there are two libraries, where one library
was compiled against the different version of the other library
than the one currently used in the project.
Please check that the project configuration is correct and has
consistent versions of dependencies.

The list of libraries that depend on
"org.jetbrains.kotlinx:kotlinx-coroutines-core
(org.jetbrains.kotlinx:kotlinx-coroutines-core-macosx64)" and may
lead to conflicts:
<list of libraries and potential version mismatches>

Project dependencies:
<dependencies tree>
```

重新设计了对未处理异常进行处理的 API

应用程序未处理的异常会抛到 Kotlin/Native 运行期, 我们统一了其处理过程, 并通过 `processUnhandledException(throwable: Throwable)` 函数公开了默认的处理, 可供自定义的执行环境使用, 比如 `kotlinx.coroutines`. 从 `Worker.executeAfter()` 中的操作逃逸的异常, 也适用这个处理, 但只针对新的 内存管理器.

API 的改进还影响通过 `setUnhandledExceptionHook()` 设置的 hook. Kotlin/Native 运行期会使用一个未被处理的异常调用这些 hook, 在之前的版本中, hook 调用结束之后, 这些 hook 会被重置, 而且此时程序总是会终止. 现在, 这些 hook 可以被使用多次, 而且, 如果你希望在发生未被处理的异常时程序总是终止, 那么, 你要么不要对未被处理的异常设置 hook (`setUnhandledExceptionHook()`), 要么在你的 hook 结束之后一定要调用 `terminateWithUnhandledException()`. 这个功能可以会帮助你将异常发送到第三方的崩溃报告服务(比如 Firebase Crashlytics), 并终止程序. 从 `main()` 中逃逸的异常, 以及跨越互操作边界的异常, 总是会终止程序, 即使 hook 没有调用 `terminateWithUnhandledException()`.

Kotlin/JS

我们正在继续改进 Kotlin/JS 编译器的 IR 后端的稳定性. Kotlin/JS 现在有一个 用于关闭 Node.js 和 Yarn 下载的选项.

使用预安装的 Node.js 和 Yarn 的选项

现在你可以在构建 Kotlin/JS 项目时关闭 Node.js 和 Yarn 的下载, 并使用主机上已安装的实例. 在没有 Internet 连接的服务器上构建时, 比如 CI 服务器, 这个功能会很有用.

要关闭外部组件的下载, 请在你的 `build.gradle(.kts)` 中添加以下代码:

- Yarn:

  Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().download = false // 或指定为 true, 使用默认行为
}
```

  Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targe
ts.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.t
argets.js.yarn.YarnRootExtension).download = false
}
```

- Node.js:

  Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targe
ts.js.nodejs.NodeJsRootPlugin> {

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.
NodeJsRootExtension>().download = false // 或指定为 true, 使用默
认行为
}
```

  Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targe
ts.js.nodejs.NodeJsRootPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.t
argets.js.nodejs.NodeJsRootExtension).download = false
}
```

Kotlin Gradle plugin

在 Kotlin 1.6.0 中, 我们将 `KotlinGradleSubplugin` 类的描述级别改为 'ERROR'. 这个类以前被用来编写编译器 plugin. 在以后的发布版中, 我们将会删除这个类. 请改为使用 `KotlinCompilerPluginSupportPlugin` 类.

我们删除了 `kotlin.useFallbackCompilerSearch` 构建选项, 以及 `noReflect` 和 `includeRuntime` 编译器选项. `useIR` 编译器选项已被隐藏, 而且将在未来的发布版中删除.

更多详情请参见 Kotlin Gradle plugin 目前支持的编译器选项 (Kotlin Gradle plugin 中的编译器选项).

标准库

新的标准库 1.6.0 版本稳定了实验性功能, 引入了新的功能, 并在各个平台统一了库的行为:

- 新的 readline 函数
- typeOf() 的稳定版
- 集合构建器的稳定版
- Duration API 的稳定版
- 使用正规表达式将字符串分割为序列
- 对整数的位轮转(Bit rotation)操作
- JS 中 replace() 和 replaceFirst() 函数的变更
- 既有 API 的改进
- 废弃的功能

新的 readline 函数

Kotlin 1.6.0 提供了新的函数用于处理标准输入: `readln()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/readln.html) 和 `readlnOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/readln-or-null.html).

> 新函数现在还只能用于 JVM 和 Native 编译目标平台.

以前的版本	1.6.0 中的替代函数	使用方法
readLine()!!	readln()	从标准输入读取一行, 并返回这一行的内容, 如果遇到 EOF 则抛出 RuntimeException 异常.
readLine()	readlnOrNull()	从标准输入读取一行, 并返回这一行的内容, 如果遇到 EOF 则返回 null.

我们认为, 在读取一行输入时不再需要使用 !!, 将会改进新使用者的编程体验, 并使得教授 Kotlin 语言更加简单. 为了让读取一行的操作名称与对应的 println() 操作保持一致, 我们决定将新函数的名称缩短为 'ln'.

```
println("What is your nickname?")
val nickname = readln()
println("Hello, $nickname!")
```

```
fun main() {
//sampleStart
    var sum = 0
    while (true) {
        val nextLine = readlnOrNull().takeUnless {
            it.isNullOrEmpty()
        } ?: break
        sum += nextLine.toInt()
    }
    println(sum)
//sampleEnd
}
```

在你的 IDE 代码自动补完时, 既存的 readLine() 函数优先度将会低于 readln() 和 readlnOrNull() 函数. IDE 检查还会推荐使用新函数代替旧的 readLine() 函数.

我们计划在未来的发布版中逐渐废弃 readLine() 函数.

typeOf() 的稳定版

Kotlin 1.6.0 版带来了 typeOf() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/type-of.html) 函数的 稳定版 (Kotlin 各部分组件的稳定性), 完成了一个 主要的 roadmap 项目 (https://youtrack.jetbrains.com/issue/KT-45396).

从 1.3.40 版开始 (https://blog.jetbrains.com/kotlin/2019/06/kotlin-1-3-40-released/), 在 JVM 平台上提供了 typeOf() 的实验性 API. 现在你可以在任何 Kotlin 平台上使用它, 并得到编译器可以推断的任何 Kotlin 类型的 KType (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-type/#kotlin.reflect.KType) 表达:

```
inline fun <reified T> renderType(): String {
    val type = typeOf<T>()
    return type.toString()
}

fun main() {
    val fromExplicitType = typeOf<Int>()
    val fromReifiedType = renderType<List<Int>>()
}
```

集合构建器的稳定版

在 Kotlin 1.6.0 中, 集合构建器函数提升为 稳定版 (Kotlin 各部分组件的稳定性). 集合构建器返回的集合现在序列化为只读状态.

现在你可以使用 buildMap() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-map.html), buildList() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-list.html), 以及 buildSet() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-set.html), 而不必添加 opt-in 注解:

```
fun main() {
//sampleStart
    val x = listOf('b', 'c')
    val y = buildList {
        add('a')
        addAll(x)
        add('d')
    }
    println(y)  // 输出结果为: [a, b, c, d]
```

```
//sampleEnd
}
```

Duration API 的稳定版

Duration (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/) 类 用于表示不同时间单位的持续时间, 它已被提升为 稳定版 (Kotlin 各部分组件的稳定性). 在 1.6.0 中, Duration API 发生了以下变化:

- `toComponents()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/to-components.html) 函数 将持续时间分解为日, 小时, 分, 秒, 和纳秒, 现在它输出的第一个元素类型为 `Long`, 而不是 `Int`. 以前, 如果值超过了 `Int` 类型范围, 会被截断到 `Int` 类型范围之内. 现在, 由于使用了 `Long` 类型, 你可以分解任意范围的持续时间值, 而不必截断超过 `Int` 类型范围的值.
- `DurationUnit` 枚举值现在是独立的类型, 而不是 JVM 平台 `java.util.concurrent.TimeUnit` 的类型别名. 我们没有发现任何有意义的场景还需要保留 `typealias DurationUnit = TimeUnit`. 而且, 通过类型别名来公布 `TimeUnit` API 可能导致 `DurationUnit` 使用者的理解混乱.
- 根据社区的返回意见, 我们恢复了 `Int.seconds` 之类的扩展属性. 但我们希望限制它们的适用范围, 因此我们将它们放在 `Duration` 类的同伴对象内. 尽管 IDE 在代码补完和自动添加 import 时仍然可以从同伴对象中提供这些扩展, 但在将来, 我们计划在需要 `Duration` 类型的地方限制这种行为.

  ```
  import kotlin.time.Duration.Companion.seconds

  fun main() {
  //sampleStart
      val duration = 10000
      println("There are ${duration.seconds.inWholeMinutes} minutes
  in $duration seconds")
      // 输出结果为: There are 166 minutes in 10000 seconds
  //sampleEnd
  }
  ```

  我们建议将之前引入的伴随函数, 比如 `Duration.seconds(Int)`, 以及已被废弃的顶级扩展, 比如 `Int.seconds`, 替换为 `Duration.Companion` 中的新的扩展函数.

  > 这样的替换可能导致旧的顶级扩展与新的伴随扩展之间的歧义. 在进行自动迁移之前,

请确认对 kotlin.time 包使用了通配符 import – `import kotlin.time.*`.

使用正规表达式将字符串分割为序列

`Regex.splitToSequence(CharSequence)` 和 `CharSequence.splitToSequence(Regex)` 函数已被提升为稳定版 (Kotlin 各部分组件的稳定性). 这些函数通过匹配正规表达式来分割字符串, 但将结果返回为 序列(Sequence) (序列(Sequence)), 使得对这个结果的所有操作都会以延迟计算(lazy)模式执行:

```
fun main() {
//sampleStart
    val colorsText = "green, red, brown&blue, orange, pink&green"
    val regex = "[,\\s]+".toRegex()
    val mixedColor = regex.splitToSequence(colorsText)
    // 或者也可以使用
    // val mixedColor = colorsText.splitToSequence(regex)
        .onEach { println(it) }
        .firstOrNull { it.contains('&') }
    println(mixedColor) // 输出结果为: "brown&blue"
//sampleEnd
}
```

对整数的位轮转(Bit rotation)操作

在 Kotlin 1.6.0 中, 用于位操作的 `rotateLeft()` 和 `rotateRight()` 函数已成为 稳定版 (Kotlin 各部分组件的稳定性). 这些函数会将数值的二进制表达向左或向右轮转指定的位数:

```
fun main() {
//sampleStart
    val number: Short = 0b10001
    println(number
        .rotateRight(2)
        .toString(radix = 2)) // 输出结果为: 100000000000100
    println(number
        .rotateLeft(2)
        .toString(radix = 2))  // 输出结果为: 1000100
//sampleEnd
}
```

JS 中 replace() 和 replaceFirst() 函数的变更

在 Kotlin 1.6.0 以前, 正规表达式函数 `replace()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-regex/replace.html) 和 `replaceFirst()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-regex/replace-first.html) 在替换包含 group 引用的字符串时, 在 Java 和 JS 平台的行为会有不同. 为了让这些函数在所有编译目标平台的行为保持一致, 我们改变了它们在 JS 平台的实现.

在替换字符串中出现的 `${name}` 或 `$index`, 会被替换为指定的 index 或名称所对应的被捕获的组(captured group):

- `$index` – '$' 之后的第一个数字会被认为是组引用(group reference)的一部分. 后续的数字只有在形成一个有效的组引用时, 才会被计入 `index`. 只有数字 '0'–'9' 会被当作组引用的组成部分. 注意, 被捕获的组的 index 值从 '1' 开始计算. index 值为 '0' 的组代表正规表达式的整个匹配.
- `${name}` – `name` 可以由字母 'a'–'z', 'A'–'Z', 或数字 '0'–'9' 组成. 第一个字符必须是字母.

  > 在替换模式中使用有名称的组, 目前只有 JVM 平台支持这个功能.

- 如果要把替换字符串中的后续字符当作文字使用, 请使用反斜杠字符 `\` 进行转义:

  ```
  fun main() {
  //sampleStart
      println(Regex("(.+)").replace("Kotlin", """\$ $1""")) // 输出结果为: $ Kotlin
      println(Regex("(.+)").replaceFirst("1.6.0", """\\ $1""")) // 输出结果为: \ 1.6.0
  //sampleEnd
  }
  ```

  如果要把替换字符串全部当作文字, 你可以使用 `Regex.escapeReplacement()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-regex/escape-replacement.html) 函数.

既有 API 的改进

- 1.6.0 版增加了 `Comparable.compareTo()` 的中缀扩展函数. 你现在可以使用中缀形式来比较两个对象的顺序:

```
class WrappedText(val text: String) : Comparable<WrappedText> {
    override fun compareTo(other: WrappedText): Int =
        this.text compareTo other.text
}
```

- JS 中的 `Regex.replace()` 函数现在也变为非 inline, 以便在所有平台上统一它的实现.
- String 的 `compareTo()` 和 `equals()` 函数, 以及 CharSequence 的 `isBlank()` 函数 在 JS 平台的行为现在与在 JVM 平台完全一致. 当出现非 ASCII 字符时, 之前版本的行为存在差异.

废弃的功能

在 Kotlin 1.6.0 中, 对某些仅限于 JS 的标准库 API, 我们开始了它的废弃周期, 会提示警告.

String 的 concat(), match(), 和 matches() 函数

- 要拼接一个字符串与另一个给定对象的字符串表达, 请使用 `plus()` 函数, 而不是 `concat()` 函数.
- 要在输入字符串中查找一个正规表达式的所有匹配, 请使用 Regex 类的 `findAll()` 函数, 而不是 `String.match(regex: String)` 函数.
- 要检查正规表达式是否匹配整个输入字符串, 请使用 Regex 类的 `matches()`函数, 而不是 `String.matches(regex: String)` 函数.

使用比较函数的数组 sort() 函数

我们废弃了 `Array<out T>.sort()` 函数, 以及 inline 函数 `ByteArray.sort()`, `ShortArray.sort()`, `IntArray.sort()`, `LongArray.sort()`, `FloatArray.sort()`, `DoubleArray.sort()`, 和 `CharArray.sort()`, 这些函数会按照比较函数决定的顺序来排序数组. 请使用其他标准库函数来对数组排序.

详情请参见 集合排序 (排序(Ordering)).

工具

Kover – 用于 Kotlin 的代码覆盖率工具

Kover Gradle plugin 还处于实验阶段. 欢迎通过 GitHub (https://github.com/Kotlin/kotlinx-kover/issues) 提供你的反馈意见.

在 Kotlin 1.6.0 中, 我们引入了 Kover – 一个 Gradle plugin, 支持 IntelliJ (https://github.com/JetBrains/intellij-coverage) 和 JaCoCo

(https://github.com/jacoco/jacoco) Kotlin 代码覆盖率统计工具. 它支持所有的语言结构, 包括 inline 函数.

更多详情请参见 Kover 的 GitHub 代码仓库 (https://github.com/Kotlin/kotlinx-kover), 或这个视频:

协程 1.6.0-RC 版

`kotlinx.coroutines` 1.6.0-RC (https://github.com/Kotlin/kotlinx.coroutines/releases/tag/1.6.0-RC) 已发布了, 包括很多新功能和改进:

- 支持 新的 Kotlin/Native 内存管理器
- 引入了派发器 *views* API, 可以限制并发, 而不需要创建额外的线程
- 从 Java 6 迁移到 Java 8 编译目标
- `kotlinx-coroutines-test`, 包括重新设计的 API 和跨平台支持
- 引入 `CopyableThreadContextElement` (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-copyable-thread-context-element/index.html), 允许协程以线程安全的方式写访问 `ThreadLocal` (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/ThreadLocal.html) 变量

更多详情请参见 变更列表 (https://github.com/Kotlin/kotlinx.coroutines/releases/tag/1.6.0-RC).

迁移到 Kotlin 1.6.0

IntelliJ IDEA 和 Android Studio 会建议你将 Kotlin plugin 更新到 1.6.0.

要将既有的项目迁移到 Kotlin 1.6.0, 请将 Kotlin 版本修改为 `1.6.0`, 然后重新导入你的 Gradle 或 Maven 项目. 更多详情请参见 如何更新到 Kotlin 1.6.0 ("更新到新的发布版" in "Kotlin 的发布版本").

要使用 Kotlin 1.6.0 创建新项目, 请更新 Kotlin plugin, 然后通过菜单 **File** | **New** | **Project** 运行项目向导.

新的命令行编译器可以通过 GitHub 发布页面
(https://github.com/JetBrains/kotlin/releases/tag/v1.6.0) 下载.

Kotlin 1.6.0 是一个 功能发布版 ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化") 因此可能带来一些变化, 造成与你使用以前版本编写的代码不兼容. 关于这些不兼容的变化, 详情请参见 Kotlin 1.6 兼容性指南 (Kotlin 1.6 兼容性指南).

Kotlin 1.5.30 版中的新功能

最终更新: 2024/09/10

发布日期: 2021/08/24 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.5.30 带来语言更新, 包括功能变更的预览, 平台支持与工具方面的大量改进, 以及新的标准库函数.

下面是主要的功能改进:

- 语言功能, 包括封闭式(sealed) `when` 语句(实验性功能), 明确要求使用者同意的功能(Opt-in Requirement)的使用方法变更, 以及其他更新
- 支持 Apple Silicon 平台的原生(Native)开发
- Kotlin/JS IR 后端升级为 Beta 版
- Gradle plugin 使用体验改进

关于功能变更的简短介绍, 也可以参见 发布公告 (https://blog.jetbrains.com/kotlin/2021/08/kotlin-1-5-30-released/), 以及下面的视频:

语言功能

Kotlin 1.5.30 提供了未来的语言功能变更的预览, 并带来了要求使用者同意的功能(Opt-in Requirement)和类型推断的改进:

- 针对封闭类或布尔值的穷尽式(exhaustive) when 语句
- 挂起函数用作超类型
- 隐含使用实验性 API 时要求使用者同意
- 要求使用者同意注解对不同目标的使用方式的变更
- 对递归泛型类型的类型推断的改进
- 去掉了构建器推断的限制

针对封闭类或布尔值的穷尽式(exhaustive) when 语句

> ⚠ 封闭 (穷尽式) when 语句是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文), 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-12380) 提供你的反馈意见.

一个 *穷尽式* `when` ("when 表达式" in "条件与循环") 语句包含对应于所有可能的类型或值的分支, 对于某些类型, 再加上 `else` 分支. 也就是说, 它覆盖所有可能的情况.

我们计划很快禁用非穷尽的 `when` 语句, 使得 `when` 语句的动作与 `when` 表达式一致. 为了保证平滑移植, 你可以配置编译器, 对封闭类或布尔值的非穷尽 `when` 语句报告警告. 在 Kotlin 1.6 中默认会出现这些警告, 将来会变为错误.

> ℹ 枚举类型已经有了这些警告.

```
sealed class Mode {
    object ON : Mode()
    object OFF : Mode()
}

fun main() {
    val x: Mode = Mode.ON
    when (x) {
        Mode.ON -> println("ON")
    }
// 编译器警告: Non exhaustive 'when' statements on sealed
classes/interfaces
// will be prohibited in 1.7, add an 'OFF' or 'else' branch instead

    val y: Boolean = true
    when (y) {
        true -> println("true")
    }
// 编译器警告: Non exhaustive 'when' statements on Booleans will be
prohibited
// in 1.7, add a 'false' or 'else' branch instead
}
```

要在 Kotlin 1.5.30 中启用这个功能, 请使用语言版本 1.6. 你也可以启用 渐进模式 ("渐进模式" in "Kotlin 1.3 版中的新功能"), 将警告变为错误.

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.6"
            //progressiveMode = true // 默认为 false
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.6'
            //progressiveMode = true // 默认为 false
        }
    }
}
```

挂起函数用作超类型

> ⚠ 挂起函数用作超类型是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-18707) 提供你的反馈意见.

Kotlin 1.5.30 提供了一个功能预览, 可以将一个 suspend 函数类型用作一个超类型, 但存在一些限制.

```
class MyClass: suspend () -> Unit {
    override suspend fun invoke() { TODO() }
}
```

使用 `-language-version 1.6` 编译器选项来启用这个功能:

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.6"
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.6'
        }
    }
}
```

这个功能存在以下限制:

- 在超类型中, 你不能混合使用一个通常的函数类型与一个的 `suspend` 函数类型. 这是由于 `suspend` 函数类型在 JVM 后端中的实现细节造成的. 它表达为一个通常的函数类型加上一个标记接口. 由于这个标记接口, 无法区分哪个父接口是挂起函数, 哪个是通常函数.
- 你不能使用多个 `suspend` 函数作为超类型. 如果存在类型检查, 你也不能使用多个通常的函数作为超类型.

隐含使用实验性 API 时要求使用者同意

> ⚠ 要求使用者同意机制是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 参见 如何明确要求使用者同意 (明确要求使用者同意的功能(Opt-in Requirement)). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

库的作者可以将一个试验性 API 标记为 要求使用者同意 ("创建表示要求使用者同意(Opt-in requirement)的注解" in "明确要求使用者同意的功能(Opt-in Requirement)"), 用来提醒使用者这个功能处于试验性状态. 当使用这个 API 时, 编译器会报告一个警告或错误, 并要求 明确同意使用 ("同意使用 API" in "明确要求使用者同意的功能(Opt-in Requirement)") 来消除这些警告或错误.

在 Kotlin 1.5.30 中, 对于签名中存在试验性类型的任何声明, 编译器都认为它们是试验性的. 也就是说, 即使对一个试验性 API 的隐含使用, 它也要求使用者同意. 比如, 如果函数的返回类型标记为试验性 API 元素, 那么使用这个函数会也要求你明确同意, 即使函数声明本身明确没有标记为需要使用者同意.

```
// 库代码

@RequiresOptIn(message = "This API is experimental.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS)
annotation class MyDateTime // 要求使用者同意的注解

@MyDateTime
class DateProvider // 一个要求使用者同意的类

// 客户端代码

// 编译器警告: experimental API usage
fun createDateSource(): DateProvider { /* ... */ }

fun getDate(): Date {
    val dateSource = createDateSource() // 这里也会出现编译器警告:
experimental API usage
    // ...
}
```

详情请参见 明确要求使用者同意 (明确要求使用者同意的功能(Opt-in Requirement)).

要求使用者同意注解对不同目标的使用方式的变更

> ⚠ 要求使用者同意机制是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 参见 如何明确要求使用者同意 (明确要求使用者同意的功能(Opt-in Requirement)). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

在不同的 注解目标 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-target/) 上的要求使用者同意的注解的使用和声明, Kotlin 1.5.30 现在使用新的规则. 对于编译期无法处理的使用场景, 编译器现在会报告错误. 在 Kotlin 1.5.30 中:

- 在使用端, 禁止对局部变量和值参数标注要求使用者同意的注解.
- 对 override, 只有当它的原始声明也进行了标注, 才允许进行标注.
- 禁止对后端域和 get 方法标注. 你可以改为标注属性.
- 在要求使用者同意的注解的声明端, 禁止将注解目标设置为 `TYPE` 和 `TYPE_PARAMETER`.

详情请参见 明确要求使用者同意的功能 (明确要求使用者同意的功能(Opt-in Requirement)).

对递归泛型类型的类型推断的改进

在 Kotlin 和 Java 中, 你可以定义一个递归泛型类型, 在它的类型参数中引用它自身. 在 Kotlin 1.5.30 中, 如果一个类型参数是递归泛型, 那么 Kotlin 编译器可以只根据对应的类型参数的上界(Upper Bound)推断出这个类型参数. 因此, 可以使用递归泛型类型, 创建出 Java 中经常用来创建构建器 API 的很多模式.

```
// Kotlin 1.5.20
val containerA = PostgreSQLContainer<Nothing>
(DockerImageName.parse("postgres:13-alpine")).apply {
    withDatabaseName("db")
    withUsername("user")
    withPassword("password")
    withInitScript("sql/schema.sql")
}

// Kotlin 1.5.30
val containerB =
```

```
PostgreSQLContainer(DockerImageName.parse("postgres:13-alpine"))
    .withDatabaseName("db")
    .withUsername("user")
    .withPassword("password")
    .withInitScript("sql/schema.sql")
```

你可以传递 `-Xself-upper-bound-inference` 或 `-language-version 1.6` 编译器选项来启用这个改进. 关于新支持的使用场景的其他示例, 请参见 这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-40804).

去掉了构建器推断的限制

构建器推断一种特殊的类型推断, 可以根据一个调用的 Lambda 参数之内的其他调用的类型信息, 来推断这个调用的类型参数. 当调用泛型构建器函数时, 这个功能可以很有用, 比如 `buildList()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-list.html) 或 `sequence()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/sequence.html): `buildList { add("string") }`.

在这样一个 Lambda 参数内部, 以前曾经存在一个限制, 不能使用构建器推断功能尝试推断的类型信息. 因此你只能指定这个类型信息, 而不能通过推断得到它. 比如, 在 `buildList()` 的 Lambda 参数之内, 除非明确指定类型参数, 否则你不能调用 `get()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/get.html).

Kotlin 1.5.30 使用 `-Xunrestricted-builder-inference` 编译器选项去掉了这个限制. 添加这个选项, 可以在泛型构建器函数的 Lambda 参数之内, 启用以前被禁止的调用:

```
@kotlin.ExperimentalStdlibApi
val list = buildList {
    add("a")
    add("b")
    set(1, null)
    val x = get(1)
    if (x != null) {
        removeAt(1)
    }
}

@kotlin.ExperimentalStdlibApi
val map = buildMap {
    put("a", 1)
```

```
    put("b", 1.1)
    put("c", 2f)
}
```

你还可以通过 `-language-version 1.6` 编译器选项启用这个功能.

Kotlin/JVM

在 Kotlin 1.5.30 版中, Kotlin/JVM 新增了以下功能:

- 创建注解类的实例
- 可否为 null(Nullability) 注解的支持配置的改进

关于 JVM 平台上的 Kotlin Gradle plugin 的更新, 请参见 Gradle 小节.

创建注解类的实例

> ⚠ 创建注解类的实例是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-45395) 提供你的反馈意见.

在 Kotlin 1.5.30 中, 现在你可以在任何代码中调用 注解类 (注解) 的构造器, 来获得一个实例. 这个功能能够用于 Java 中相同的使用场景, 可以实现一个注解接口.

```
annotation class InfoMarker(val info: String)

fun processInfo(marker: InfoMarker) = ...

fun main(args: Array<String>) {
    if (args.size != 0)
        processInfo(getAnnotationReflective(args))
    else
        processInfo(InfoMarker("default"))
}
```

使用 `-language-version 1.6` 编译器选项来启用这个功能. 注意, 注解类现有的所有限制都继续存在, 比如不能定义 非 `val` 参数, 或与次级构造器(secondary constructor)不同的成员.

关于创建注解类的实例, 更多详情请参见 这个 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/annotation-instantiation.md).

可否为 null(Nullability) 注解的支持配置的改进

Kotlin 编译器可以读取多种类型的 可否为 null(Nullability) 注解 ("可否为 null(Nullability) 注解" in "在 Kotlin 中调用 Java 代码"), 来从 Java 代码得到可否为 null 信息. 这个信息使得它能够报告 Kotlin 中调用 Java 代码时的可否为 null 不匹配的错误.

在 Kotlin 1.5.30 中, 你可以指定编译器是否根据指定的可否为 null 注解类型的信息来报告可否为 null 不匹配的错误. 只需要使用编译器选项 `-Xnullability-annotations=@<package-name>:<report-level>`. 在参数中, 指定可否为 null 注解的全限定包名称, 以及以下报告级别中的一个:

- `ignore`: 忽略可否为 null 不匹配
- `warn`: 报告为警告
- `strict`: 报告为错误.

详情请参见 支持的可否为 null 注解列表 ("可否为 null(Nullability) 注解" in "在 Kotlin 中调用 Java 代码") 以及它们的全限定包名称.

下面是一个示例, 演示如何对新支持的 RxJava (https://github.com/ReactiveX/RxJava) 3 可否为 null 注解启用错误报告: `-Xnullability-annotations=@io.reactivex.rxjava3.annotations:strict`. 注意, 所有这些可否为 null 不匹配, 默认设置为警告.

Kotlin/Native

Kotlin/Native 包含以下变更和改进:

- 支持 Apple Silicon
- CocoaPods Gradle plugin 的 Kotlin DSL 的改进
- 与 Swift 5.5 async/await 的交互(实验性功能)
- 对象和伴随对象到 Swift/Objective-C 的映射的改进
- 对 MinGW 编译目标废弃无导入库的 DLL 链接

支持 Apple Silicon

Kotlin 1.5.30 引入了对 Apple Silicon (https://support.apple.com/en-us/HT211814) 的原生支持.

在以前的版本中, Kotlin/Native 编译器和工具需要 Rosetta 翻译环境 (https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment) 才能在 Apple Silicon 主机上工作. 在 Kotlin 1.5.30 中, 不再需要翻译环境 – 编译器和工具可以在 Apple Silicon 硬件上运行, 不需要任何额外的操作.

我们还引入了新的编译目标, 可以使 Kotlin 代码在 Apple Silicon 上直接运行:

- `macosArm64`
- `iosSimulatorArm64`
- `watchosSimulatorArm64`
- `tvosSimulatorArm64`

这些编译目标可以用于 Intel 和 Apple Silicon 的主机. 所有既有的编译目标也可以在 Apple Silicon 主机上使用.

注意, 在 1.5.30 中, 我们只在 `kotlin-multiplatform` Gradle plugin 中提供对 Apple Silicon 编译目标的基本的支持. 具体来说, 在 `ios`, `tvos`, 和 `watchos` 编译目标简写(target shortcut) 中没有包含新的模拟器编译目标. 我们会继续改进这些新的编译目标的使用体验.

CocoaPods Gradle plugin 的 Kotlin DSL 的改进

Kotlin/Native Framework 的新参数

Kotlin 1.5.30 带来了 CocoaPods Gradle plugin DSL 关于 Kotlin/Native Framework 的改进. 除了 Framework 名称之外, 你还可以在 Pod 配置中指定其他参数:

- 指定 Framework 的动态或静态版本
- 明确启用导出依赖项
- 启用 Bitcode 内嵌

要使用新的 DSL, 请将你的项目更新到 Kotlin 1.5.30, 并在你的 `build.gradle(.kts)` 文件的 `cocoapods` 节中指定参数:

```
cocoapods {
    frameworkName = "MyFramework" // 这个属性已废弃, 并会在将来的版本中删除
    // Framework 配置的新的 DSL 如下:
    framework {
```

```
        // 支持所有的 Framework 属性
        // Framework 名称配置. 使用这个属性代替已废弃的 'frameworkName'
        baseName = "MyFramework"
        // 支持动态 Framework
        isStatic = false
        // 依赖项导出
        export(project(":anotherKMMModule"))
        transitiveExport = false // 这是默认设置.
        // Bitcode 内嵌
        embedBitcode(BITCODE)
    }
}
```

对 Xcode 配置支持自定义名称

Kotlin CocoaPods Gradle plugin 在 Xcode 构建配置中支持自定义名称. 如果你在 Xcode 中为构建配置使用了特殊的名称, 比如 `Staging`, 这个功能也可以帮助你.

要指定一个自定义名称, 请在你的 `build.gradle(.kts)` 文件的 `cocoapods` 节中使用 `xcodeConfigurationToNativeBuildType` 参数:

```
cocoapods {
    // 将自定义的 Xcode 配置映射到 NativeBuildType
    xcodeConfigurationToNativeBuildType["CUSTOM_DEBUG"] =
NativeBuildType.DEBUG
    xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] =
NativeBuildType.RELEASE
}
```

这个参数不会出现在 Podspec 文件中. 当 Xcode 运行 Gradle 构建过程时, Kotlin CocoaPods Gradle plugin 会选择必要的原生构建类型.

> 不需要声明 `Debug` 和 `Release` 配置, 因为默认支持它们.

与 Swift 5.5 async/await 的交互 (实验性功能)

> 与 Swift async/await 的并发交互是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的

问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-47610) 提供你的反馈意见.

过去我们曾经 在 1.4.0 中添加了从 Objective-C 和 Swift 中调用 Kotlin 挂起函数的功能 ("在 Swift 和 Objective-C 中支持 Kotlin 的挂起函数" in "Kotlin 1.4.0 版中的新功能"), 现在我们删除这个功能, 改为使用新的 Swift 5.5 功能 – 使用 `async` 和 `await` 修饰符的并发功能 (https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md).

对于返回类型可为 null 的挂起函数, Kotlin/Native 编译器现在会在生成的 Objective-C 头文件中, 输出 `_Nullable_result` 属性. 因此在 Swift 中可以将这些函数作为 `async` 函数来调用, 并且得到正确的可否为 null 结果.

注意, 这个功能是实验性功能, 未来可能由于 Kotlin 和 Swift 的变化而受到影响. 目前来说, 我们提供这个功能的一个预览版, 带有一些限制, 我们期待得到你的意见反馈. 请在 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-47610) 中查看这个功能目前的状态, 并留下你的反馈意见.

对象和伴随对象到 Swift/Objective-C 的映射的改进

对于原生 iOS 开发者来说, 现在可以通过更加符合直觉的方式得到对象和伴随对象. 比如, 如果在 Kotlin 中你有以下对象:

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

要在 Swift 中访问它们, 你可以使用 `shared` 和 `companion` 属性:

```
MyObject.shared
MyObject.shared.x
MyClass.companion
MyClass.Companion.shared
```

详情请参见 与 Swift/Objective-C 代码交互 (与 Swift/Objective-C 代码交互).

对 MinGW 编译目标废弃无导入库的 DLL 链接

LLD (https://lld.llvm.org/) 是 LLVM 项目的一个链接器, 我们计划在 Kotlin/Native 中对 MinGW 编译目标使用它, 因为它的好处比默认的 ld.bfd 更多 – 主要是它的性能更好.

但是, LLD 的最新稳定版本对 MinGW (Windows) 编译目标不支持直接链接到 DLL. 这样的链接需要使用 导入库 (https://stackoverflow.com/questions/3573475/how-does-the-import-library-work-details/3573527#3573527). 尽管对于 Kotlin/Native 1.5.30 来说不需要它们, 但我们添加了一个警告, 告知你这样的使用不兼容于 LLD, 将来 LLD 会成为 MinGW 编译目标的默认链接器.

关于转换到 LLD 浏览器, 请在 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-47605) 中提供你的反馈意见.

Kotlin Multiplatform

1.5.30 对于 Kotlin Multiplatform 带来了以下重要更新:

- 在共用的原生代码中可以使用自定义 `cinterop` 库
- 支持 XCFramework
- 对 Android artifact 的新的默认发布设置

在共用的原生代码中可以使用自定义 cinterop 库

Kotlin Multiplatform 提供了一个 选项 ("连接平台相关的库" in "在不同的平台之间共用代码"), 可以在共用的源代码集中使用平台相关的 interop 库. 在 1.5.30 之前, 这个功能只能用于随 Kotlin/Native 一同发布的 平台库 (平台库). 从 1.5.30 开始, 你可以使用你自定义的 `cinterop` 库. 要启用这个功能, 请在你的 `gradle.properties` 中添加 `kotlin.mpp.enableCInteropCommonization=true` 属性:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true
kotlin.native.enableDependencyPropagation=false
kotlin.mpp.enableCInteropCommonization=true
```

支持 XCFramework

所有的 Kotlin Multiplatform 项目现在可以使用 XCFramework 作为输出格式. Apple 引入了 XCFramework 来替代通用(Universal) (fat) Framework. 通过使用 XCFramework, 你:

- 可以将所有的编译目标平台和处理器架构的逻辑集中在一个单独的 bundle 中.
- 在将应用程序发布到 App Store 之前, 不必删除所有不需要的处理器架构.

如果你想要对 Apple M1 上的设备和模拟器使用你的 Kotlin Framework, XCFramework 会很有用.

要使用 XCFramework, 请更新你的 `build.gradle(.kts)` 脚本:

Kotlin

```
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFramework

plugins {
    kotlin("multiplatform")
}

kotlin {
    val xcf = XCFramework()

    ios {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
    watchos {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
    tvos {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
}
```

Groovy

```
import
org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFrameworkConfig

plugins {
    id 'org.jetbrains.kotlin.multiplatform'
}

kotlin {
    def xcf = new XCFrameworkConfig(project)

    ios {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
    watchos {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
    tvos {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
}
```

当你声明 XCFramework 时, 会注册这些新的 Gradle task:

- `assembleXCFramework`
- `assembleDebugXCFramework` (debug 用 artifact, 包含 dSYMs (符号化(Symbolicate) iOS

崩溃报告(Crash Report)))

- assembleReleaseXCFramework

关于 XCFramework, 详情请参见 这个 WWDC 视频 (https://developer.apple.com/videos/play/wwdc2019/416/).

对 Android artifact 的新的默认发布设置

使用 maven-publish Gradle plugin, 你可以在构建脚本中指定 Android 变体 (https://developer.android.com/studio/build/build-variants) 名称, 对 Android 编译目标发布你的跨平台库 ("发布 Android 库" in "发布跨平台的库"). Kotlin Gradle plugin 会自动生成发布.

在 1.5.30 之前, 生成的发布 metadata (https://docs.gradle.org/current/userguide/publishing_gradle_module_metadata.html) 包含每一个发布的 Android 变体的构建类型属性, 因此只能兼容于库使用者所使用的相同的构建类型. Kotlin 1.5.30 引入了一个新的默认发布设置:

- 如果项目发布的所有 Android 变体拥有相同的构建类型属性, 那么发布的变体不会拥有构建类型属性, 而且能够兼容于任何构建类型.
- 如果发布的变体拥有不同的构建类型属性, 那么只有带有 release 值的, 发布时会不带有构建类型属性. 因此 release 变体兼容于使者端的任何构建类型, 而 release 之外的其他变体只兼容于匹配的使者端构建类型.

如果要关闭这个功能, 并对所有变体保持构建类型属性, 你可以设置这个 Gradle 属性: kotlin.android.buildTypeAttribute.keep=true.

Kotlin/JS

在 1.5.30 中, Kotlin/JS 有 2 个主要改进:

- JS IR 编译器后端升级为 Beta 版
- 使用 Kotlin/JS IR 后端为应用程序带来更好的调试体验

JS IR 编译器后端升级为 Beta 版

1.4.0 版引入了 Kotlin/JS 的 基于 IR 的编译器后端 ("统一的后端和扩展性" in "Kotlin 1.4.0 版中的新功能"), 当时是 Alpha 版 (Kotlin 各部分组件的稳定性), 现在升级为 Beta 版.

以前, 我们发布了 移植到 JS IR 后端的向导 (将 Kotlin/JS 项目迁移到 IR 编译器), 来帮助你将你的项目移植到新的后端. 现在我们提供 Kotlin/JS Inspection Pack (https://plugins.jetbrains.com/plugin/17183-kotlin-js-inspection-pack/) IDE plugin, 它可以直接在 IntelliJ IDEA 中显示需要哪些修改.

使用 Kotlin/JS IR 后端为应用程序带来更好的调试体验

Kotlin 1.5.30 带来了对 Kotlin/JS IR 后端的 JavaScript 源代码映射生成功能. 这个功能可以改善启用 IR 后端时的 Kotlin/JS 调试体验, 支持所有的调试功能, 包括断点, 单步执行, 以及易读的调用栈信息, 带有正确的源代码引用.

详情请参见 如何在浏览器中或在 IntelliJ IDEA Ultimate 中调试 Kotlin/JS (调试 Kotlin/JS 代码).

Gradle

为了 改进 Kotlin Gradle plugin 使用者体验 (https://youtrack.jetbrains.com/issue/KT-45778), 我们实现了以下功能:

- 支持 Java 工具链, 包括 可以使用 `UsesKotlinJavaToolchain` 接口对 Gradle 旧版本指定 JDK Home
- 用更简单的方式明确指定 Kotlin Daemon 的 JVM 参数

支持 Java 工具链

Gradle 6.7 引入了 支持 Java 工具链 (https://docs.gradle.org/current/userguide/toolchains.html) 功能. 使用这个功能, 你可以:

- 使用与 Gradle 不同的 JDK 和 JRE 运行编译, 测试, 和可执行文件.
- 使用未发布的语言版本编译和测试代码.

通过工具链支持, Gradle 可以自动检测本地的 JDK, 并安装 Gradle 构建所需要但缺失的 JDK. 现在 Gradle 自身可以在任何 JDK 上运行, 而且还能够重用 构建缓存功能 ("对 Gradle 构建缓存的支持" in "Kotlin Gradle plugin 中的编译与缓存").

Kotlin Gradle plugin 对 Kotlin/JVM 编译任务支持 Java 工具链. Java 工具链会:

- 为 JVM 编译目标设置可用的 `jdkHome` 选项 ("JVM 任务独有的属性" in "Kotlin Gradle plugin 中的编译器选项").

> ⚠ 直接设置 jdkHome 选项的功能已废弃 (https://youtrack.jetbrains.com/issue/KT-46541).

- 如果使用者没有明确设置 jvmTarget 选项, 会将 kotlinOptions.jvmTarget ("JVM 任务独有的属性" in "Kotlin Gradle plugin 中的编译器选项") 设置为工具链的 JDK 版本. 如果工具链没有配置, jvmTarget 域会使用默认值. 详情请参见 JVM 编译目标兼容性 ("对相关联的编译任务检查 JVM 编译目标的兼容性" in "配置 Gradle 项目").
- 影响 kapt worker ("并行运行多个 KAPT 任务" in "kapt 编译器插件") 运行在哪个 JDK 上.

可以使用以下代码来设置一个工具链. 请将 <MAJOR_JDK_VERSION> 替换为你想要使用的 JDK 版本:

Kotlin

```
kotlin {
    jvmToolchain {
        (this as
JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<M
AJOR_JDK_VERSION>)) // "8"
    }
}
```

Groovy

```
kotlin {
    jvmToolchain {

languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
// "8"
    }
}
```

注意, 通过 kotlin 扩展设置工具链, 也会对 Java 编译任务更新工具链.

你可以通过 java 扩展设置工具链, Kotlin 编译任务会使用它:

```
java {
    toolchain {

languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) //
"8"
    }
}
```

关于为 `KotlinCompile` 任务设置 JDK 版本, 请参见 使用 Task DSL 设置 JDK 版本 ("使用 Task DSL 设置 JDK 版本" in "配置 Gradle 项目").

对于 Gradle 版本 6.1 到 6.6, 请 使用 `UsesKotlinJavaToolchain` 接口来设置 JDK Home.

使用 UsesKotlinJavaToolchain 接口指定 JDK Home

所有支持通过 `kotlinOptions` (Kotlin Gradle plugin 中的编译器选项) 设置 JDK 的 Kotlin 任务, 现在都实现 `UsesKotlinJavaToolchain` 接口. 要设置 JDK Home, 请设置你的 JDK 路径, 并替换 `<JDK_VERSION>` 部分:

Kotlin

```
project.tasks
    .withType<UsesKotlinJavaToolchain>()
    .configureEach {
        it.kotlinJavaToolchain.jdk.use(
            "/path/to/local/jdk",
            JavaVersion.<LOCAL_JDK_VERSION>
        )
    }
```

Groovy

```
project.tasks
    .withType(UsesKotlinJavaToolchain.class)
    .configureEach {
        it.kotlinJavaToolchain.jdk.use(
            '/path/to/local/jdk',
            JavaVersion.<LOCAL_JDK_VERSION>
```

```
        )
    }
```

对 Gradle 6.1 到 6.6 版本, 请使用 UsesKotlinJavaToolchain 接口. 从 Gradle 6.7 开始, 请改为使用 Java 工具链.

使用这个功能时, 请注意, kapt 任务 worker ("并行运行多个 KAPT 任务" in "kapt 编译器插件") 只使用 进程隔离模式 (https://docs.gradle.org/current/userguide/worker_api.html#changing_the_isolation_mode), kapt.workers.isolation 属性将被忽略.

用更简单的方式明确指定 Kotlin Daemon 的 JVM 参数

在 Kotlin 1.5.30 中, 对于 Kotlin Daemon 的 JVM 参数有了新的逻辑. 以下列表中的每个选项都会覆盖它之前的选项:

- 如果没有指定任何参数, Kotlin Daemon 会从 Gradle Daemon 继承参数(和以前一样). 比如, 在 gradle.properties 文件中:

  ```
  org.gradle.jvmargs=-Xmx1500m -Xms=500m
  ```

- 如果 Gradle Daemon 的 JVM 参数包含 kotlin.daemon.jvm.options 系统属性, 和以前一样使用它:

  ```
  org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m -Xms=500m
  ```

- 在 gradle.properties 文件中你可以添加 kotlin.daemon.jvmargs 属性:

  ```
  kotlin.daemon.jvmargs=-Xmx1500m -Xms=500m
  ```

- 你可以在 kotlin 扩展中指定参数:

  Kotlin

  ```
  kotlin {
    kotlinDaemonJvmArgs = listOf("-Xmx486m", "-Xms256m", "-
  XX:+UseParallelGC")
  }
  ```

Groovy

```
kotlin {
  kotlinDaemonJvmArgs = ["-Xmx486m", "-Xms256m", "-
XX:+UseParallelGC"]
}
```

- 你可以对一个特定的任务指定参数：

Kotlin

```
tasks
    .matching { it.name == "compileKotlin" && it is
CompileUsingKotlinDaemon }
    .configureEach {
        (this as
CompileUsingKotlinDaemon).kotlinDaemonJvmArguments.set(listOf(
"-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"))
    }
```

Groovy

```
tasks
    .matching {
        it.name == "compileKotlin" && it instanceof
CompileUsingKotlinDaemon
    }
    .configureEach {
        kotlinDaemonJvmArguments.set(["-Xmx1g", "-Xms512m"])
    }
```

> 在这种情况中, 在任务执行时可以启动一个新的 Kotlin Daemon 实例. 详情请参见 Kotlin Daemon 与 JVM 参数的交互 ("设置 Kotlin daemon 的 JVM 参数" in "Kotlin

Gradle plugin 中的编译与缓存").

关于 Kotlin Daemon, 详情请参见 Kotlin Daemon 以及它在 Gradle 中的使用 ("Kotlin daemon 及其在 Gradle 中的使用" in "Kotlin Gradle plugin 中的编译与缓存").

标准库

Kotlin 1.5.30 包括对标准库的 `Duration` 和 `Regex` API 的改进:

- 改变了 `Duration.toString()` 的输出
- 从字符串解析 Duration
- 在一个指定的位置匹配正则表达式
- 使用正则表达式将字符串切分为一个序列

改变了 Duration.toString() 的输出

Duration API 是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

在 Kotlin 1.5.30 以前, `Duration.toString()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/to-string.html) 函数会返回它的参数的字符串表达, 使用最紧凑的并且可以阅读的数值单位. 从现在开始, 它会返回一个字符串值, 表达为多个数值部分的组合, 每个数值部分使用自己的单位. 每个部分是一个数值, 加上一个单位缩写名称: `d`, `h`, `m`, `s`. 比如:

函数调用示例	以前的输出	现在的输出
Duration.days(45).toString()	45.0d	45d
Duration.days(1.5).toString()	36.0h	1d 12h
Duration.minutes(1230).toString()	20.5h	20h 30m
Duration.minutes(2415).toString()	40.3h	1d 16h 15m
Duration.minutes(920).toString()	920m	15h 20m
Duration.seconds(1.546).toString()	1.55s	1.546s
Duration.milliseconds(25.12).toString()	25.1ms	25.12ms

负值的时间长度表达也发生了同样的变化. 一个负值的时间长度使用负号 (`-`) 前缀, 如果它由多个部分组成, 会用括号括起: `-12m` 和 `-(1h 30m)`.

注意, 少于 1 秒的时间长度表达为单个数值, 使用秒以下单位的. 比如, `ms` (毫秒), `us` (微秒), 或 `ns` (纳秒): `140.884ms`, `500us`, `24ns`. 不再使用科学记数法来表达.

如果你想要使用单个单位来表达时间长度, 请使用超载函数 `Duration.toString(unit, decimals)`.

> 在某些情况下, 我们推荐使用 `Duration.toIsoString()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/to-iso-string.html), 包括序列化和数据交换. `Duration.toIsoString()` 使用更加严格的 ISO-8601 (https://www.iso.org/iso-8601-date-and-time-format.html) 格式, 而不是 `Duration.toString()`.

从字符串解析 Duration

> Duration API 是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 这个 issue (https://github.com/Kotlin/KEEP/issues/190) 提供你的反馈意见.

在 Kotlin 1.5.30 中, 有以下新的 Duration API 函数:

- `parse()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/parse.html), 支持解析以下函数的输出:
  - `toString()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/to-string.html).
  - `toString(unit, decimals)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/to-string.html).
  - `toIsoString()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/to-iso-string.html).
- `parseIsoString()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/parse-iso-string.html), 只解析 `toIsoString()` 输出的格式.
- `parseOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/parse-or-null.html) 和 `parseIsoStringOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/parse-iso-string-or-null.html), 与上面的函数类似, 但对于无效的 Duration 格式, 会返回 `null` 而不是抛出 `IllegalArgumentException` 异常.

下面是 `parse()` 和 `parseOrNull()` 的一些使用示例:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
//sampleStart
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    val singleUnitFormatString = "1.5h"
    val invalidFormatString = "1 hour 30 minutes"
    println(Duration.parse(isoFormatString)) // 输出为 "1h 30m"
    println(Duration.parse(defaultFormatString)) // 输出为 "1h 30m"
    println(Duration.parse(singleUnitFormatString)) // 输出为 "1h
30m"
    //println(Duration.parse(invalidFormatString)) // 抛出异常
```

```
    println(Duration.parseOrNull(invalidFormatString)) // 输出为
"null"
//sampleEnd
}
```

下面是 parseIsoString() 和 parseIsoStringOrNull() 的一些使用示例:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
//sampleStart
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    println(Duration.parseIsoString(isoFormatString)) // 输出为 "1h
30m"
    //println(Duration.parseIsoString(defaultFormatString)) // 抛出异
常
    println(Duration.parseIsoStringOrNull(defaultFormatString)) //
输出为 "null"
//sampleEnd
}
```

在一个指定的位置匹配正规表达式

> Regex.matchAt() 和 Regex.matchesAt() 函数是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-34021) 提供你的反馈意见.

新的 Regex.matchAt() 和 Regex.matchesAt() 函数提供一种方法, 可以检查一个正规表达式在一个 String 或 CharSequence 的指定位置是否存在完全匹配.

matchesAt() 返回一个 boolean 结果:

```
fun main(){
//sampleStart
    val releaseText = "Kotlin 1.5.30 is released!"
```

```
    // 正规表达式: 1个数字, 点, 1个数字, 点, 1个或多个数字
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()
    println(versionRegex.matchesAt(releaseText, 0)) // 输出为 "false"
    println(versionRegex.matchesAt(releaseText, 7)) // 输出为 "true"
//sampleEnd
}
```

如果找到匹配, 则 `matchAt()` 返回匹配结果, 否则返回 `null`:

```
fun main(){
//sampleStart
    val releaseText = "Kotlin 1.5.30 is released!"
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()
    println(versionRegex.matchAt(releaseText, 0)) // 输出为 "null"
    println(versionRegex.matchAt(releaseText, 7)?.value) // 输出为
"1.5.30"
//sampleEnd
}
```

使用正规表达式将字符串切分为一个序列

> ⚠ `Regex.splitToSequence()` 和 `CharSequence.splitToSequence(Regex)` 函数是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-23351) 提供你的反馈意见.

新的 `Regex.splitToSequence()` 函数是 `split()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-regex/split.html) 函数的 lazy 版本. 它根据正规表达式的匹配结果切分字符串, 但结果返回为一个 序列(Sequence) (序列(Sequence)), 因此对这个结果的所有操作都会以 lazy 模式执行.

```
fun main(){
//sampleStart
    val colorsText = "green, red , brown&blue, orange, pink&green"
    val regex = "[,\\s]+".toRegex()
    val mixedColor = regex.splitToSequence(colorsText)
        .onEach { println(it) }
        .firstOrNull { it.contains('&') }
```

```
    println(mixedColor) // 输出为 "brown&blue"
//sampleEnd
}
```

对 `CharSequence` 也添加了一个类似的函数：

```
val mixedColor = colorsText.splitToSequence(regex)
```

Serialization 的 1.3.0-RC 版

发布了 `kotlinx.serialization` 1.3.0-RC (https://github.com/Kotlin/kotlinx.serialization/releases/tag/v1.3.0-RC) 版, 包括新的 JSON 序列化功能:

- Java IO Stream 序列化
- 对默认值的属性级控制
- 一个选项, 可以在序列化中排除 null 值
- 在多态序列化中使用自定义类区分

详情请参见 变更列表 (https://github.com/Kotlin/kotlinx.serialization/releases/tag/v1.3.0-RC).

Kotlin 1.5.20 版中的新功能

最终更新: 2024/09/10

发布日期: 2021/06/24 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.5.20 修复了在 1.5.0 新功能中发现的问题, 还包含很多工具改进.

关于这个版本的变更概要, 可以查看 release blog (https://blog.jetbrains.com/kotlin/2021/06/kotlin-1-5-20-released/) 和以下视频:

Kotlin/JVM

Kotlin 1.5.20 包含 JVM 平台上的以下更新:

- 通过动态调用拼接字符串
- 支持 JSpecify 的可否为 null 注解
- 支持在包含 Kotlin 和 Java 代码的模块内调用 Lombok 生成的 Java 方法

通过动态调用拼接字符串

Kotlin 1.5.20 在 JVM 9+ 以上的目标平台, 将字符串拼接编译为 动态调用(dynamic invocation) (https://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html#invokedynamic) (`invokedynamic`), 与最新的 Java 版本保持一致. 确切的说, 它使用 `StringConcatFactory.makeConcatWithConstants()` (https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/StringConcatFactory.html#makeConcatWithConstants-java.lang.invoke.MethodHandles.Lookup-java.lang.String-java.lang.invoke.MethodType-java.lang.String-java.lang.Object...-) 实现字符串拼接.

要切换回以前版本中使用的 `StringBuilder.append()` (https://docs.oracle.com/javase/9/docs/api/java/lang/StringBuilder.html#append-java.lang.String-) 拼接模式, 请添加编译器选项 `-Xstring-concat=inline`.

关于如何添加编译器选项, 请参见 Gradle (Kotlin Gradle plugin 中的编译器选项), Maven ("指定编译器选项" in "Maven"), 和 命令行编译器 ("编译器选项" in "Kotlin 编译器选项") 文档.

支持 JSpecify 的可否为 null 注解

Kotlin 编译器可以读取多种类型的 可否为 null 注解 ("可否为 null(Nullability) 注解" in "在 Kotlin 中调用 Java 代码"), 以便将可否为 null 信息从 Java 传递到 Kotlin. 1.5.20 版增加了对 JSpecify 项

目 (https://jspecify.dev/) 的支持, 这个项目包括统一的 Java 可否为 null 注解.

通过 JSpecify, 你可以提供更加详细的可否为 null 信息, 帮助 Kotlin 与 Java 代码交互时保证 null 值安全性. 你可以对声明, 包, 或模块范围设置默认的可否为 null 设定, 也可以通过参数指定可否为 null, 以及其他功能. 详细的功能请参见 JSpecify 用户指南 (https://jspecify.dev/user-guide.html).

下面是一个示例, 演示 Kotlin 如何处理 JSpecify 注解:

```
// JavaClass.java
import org.jspecify.nullness.*;

@NullMarked
public class JavaClass {
  public String notNullableString() { return ""; }
  public @Nullable String nullableString() { return ""; }
}
```

```
// Test.kt
fun kotlinFun() = with(JavaClass()) {
  notNullableString().length // OK
  nullableString().length    // 警告: 接受者的可否为 null 设定不匹配
}
```

在 1.5.20 中, 根据 JSpecify 提供的可否为 null 信息, 所有的可否为 null 设定不匹配, 会被报告为编译警告. 可以添加 `-Xjspecify-annotations=strict` 和 `-Xtype-enhancement-improvements-strict-mode` 编译器选项, 在使用 JSpecify 时启动严格模式 (报告为编译错误). 请注意, JSpecify 项目还在活跃开发中. 它的 API 和实现随时有可能发生大的变化.

参见 null 值安全性与平台数据类型 ("Null 值安全性与平台数据类型" in "在 Kotlin 中调用 Java 代码").

支持在包含 Kotlin 和 Java 代码的模块内调用 Lombok 生成的 Java 方法

Lombok 编译器插件是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-7112) 提供你的反馈意见.

Kotlin 1.5.20 引入了 Lombok 编译器插件 (Lombok 编译器插件) (实验性功能). 这个插件能够在包含 Kotlin 和 Java 代码的模块内, 生成并使用 Java 的 Lombok (https://projectlombok.org/) 声明. Lombok 注解 只能用于 Java 源代码, 如果你在 Kotlin 代码中使用, 会被忽略.

插件支持以下注解:

- `@Getter`, `@Setter`
- `@NoArgsConstructor`, `@RequiredArgsConstructor`, 和 `@AllArgsConstructor`
- `@Data`
- `@With`
- `@Value`

我们还在继续完善这个插件. 关于目前的开发状态, 详情请参见 Lombok 编译器插件的 README (https://github.com/JetBrains/kotlin/blob/master/plugins/lombok/lombok-compiler-plugin/README.md).

目前, 我们不计划支持 `@Builder` 注解. 但如果你 在 YouTrack 投票支持 `@Builder` (https://youtrack.jetbrains.com/issue/KT-46959), 我们可以考虑增加这个功能.

参见 如何配置 Lombok 编译器插件 ("Gradle" in "Lombok 编译器插件").

Kotlin/Native

Kotlin/Native 1.5.20 提供了新功能的预览, 以及工具的改进:

- Opt-in: 导出 KDoc 注释到生成的 Objective-C 头文件
- 编译器 bug 修正
- 在同一个数组内的 Array.copyInto() 的性能改进

Opt-in: 导出 KDoc 注释到生成的 Objective-C 头文件

> ⚠ 导出 KDoc 注释到生成的 Objective-C 头文件是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者明确同意(Opt-in) (详情请参见下文), 而且你应该只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-38600) 提供你的反馈意见.

现在你可以设置 Kotlin/Native 编译器, 让它将 Kotlin 代码中的 文档注释 (KDoc) (为 Kotlin 代码编写文档: KDoc), 导出到生成的 Objective-C 框架, 使得框架的使用者可以访问这些文档注释.

比如, 下面是带有 KDoc 的 Kotlin 代码:

```
/**
 * 打印输出参数的和.
 * 妥善处理和超越 32 位整数的情况.
 */
fun printSum(a: Int, b: Int) = println(a.toLong() + b)
```

会生成下面的 Objective-C 头文件:

```
/**
 * 打印输出参数的和.
 * 妥善处理和超越 32 位整数的情况.
 */
+ (void)printSumA:(int32_t)a b:(int32_t)b
__attribute__((swift_name("printSum(a:b:)")));
```

这个功能同样适用于 Swift.

要试用这个功能, 导出 KDoc 注释到 Objective-C 头文件, 请使用 `-Xexport-kdoc` 编译器选项. 向你希望导出注释的 Gradle 项目的 `build.gradle(.kts)` 文件添加以下内容:

Kotlin

```
kotlin {

targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        compilations.get("main").kotlinOptions.freeCompilerArgs
+= "-Xexport-kdoc"
    }
}
```

Groovy

```
kotlin {

targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNa
tiveTarget) {
        compilations.get("main").kotlinOptions.freeCompilerArgs
+= "-Xexport-kdoc"
    }
}
```

如果你能通过这个 YouTrack 票 (https://youtrack.jetbrains.com/issue/KT-38600)反馈你的意见, 我们将会非常感谢.

编译器 bug 修正

在 1.5.20 中, Kotlin/Native 编译器修正了多个 bug. 完整列表请参见 变更列表 (https://github.com/JetBrains/kotlin/releases/tag/v1.5.20).

有一个重要的 bug 修正会影响到兼容性: 在以前的版本中, 包含不正确 UTF surrogate pair (https://en.wikipedia.org/wiki/Universal_Character_Set_characters#Surrogates) 的字符串常数, 在编译期间会丢失其内容. 现在这样的字符串值会保留. 应用程序开发者可以安全的更新到 1.5.20 – 不会出现任何问题. 但是, 使用 1.5.20 编译的库将不能兼容以前版本的编译器. 详情请参见 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-33175).

在同一个数组内的 Array.copyInto() 的性能改进

当 copy 的来源与目标是同一个数组时, 我们改进了 `Array.copyInto()` 的工作方式. 由于对这种场景的内存管理进行了优化, 现在这样的操作速度提高了 20 倍 (具体数字取决与复制的对象数量).

Kotlin/JS

1.5.20 版中, 我们发布了一个指南, 帮助你将项目迁移到 Kotlin/JS 的新的 基于 IR 的编译器后端 (使用 IR 编译器).

针对 JS IR 编译器后端的迁移指南

新的 针对 JS IR 编译器后端的迁移指南 (将 Kotlin/JS 项目迁移到 IR 编译器) 列举了你在迁移过程中可能遇到的问题, 并提供了解决方案. 如果你发现了迁移指南中未提到的其他问题, 请到我们的 问题追踪系统 (http://kotl.in/issue) 提交报告.

Gradle

Kotlin 1.5.20 增加了以下功能, 改进 Gradle 的使用体验:

- 在 kapt 中, 注解处理器的 classloader 缓存
- 废弃 `kotlin.parallel.tasks.in.project` 属性

在 kapt 中, 注解处理器的 classloader 缓存

> 在 kapt 中, 注解处理器的 classloader 缓存是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-28901) 提供你的反馈意见.

现在有一个新的实验性功能, 可以在 kapt (kapt 编译器插件) 中缓存注解处理器的 classloader. 对于连续执行很多 Gradle 任务, 这个功能可以提高 kapt 的速度.

要启用这个功能, 请在你的 `gradle.properties` 文件中添加以下属性:

```
# 正数值会启用缓存功能
# 请在这里指定与使用 kapt 的模块数相同的数字
kapt.classloaders.cache.size=5

# 为让缓存正确工作, 需要关闭这个设定
kapt.include.compile.classpath=false
```

详情请参见 kapt (kapt 编译器插件).

废弃 kotlin.parallel.tasks.in.project 属性

这个发布版中, Kotlin 的并行编译会通过 Gradle 并行执行标记 `--parallel` (https://docs.gradle.org/current/userguide/performance.html#parallel_execution) 来控制. 使用这个标记, Gradle 可以并行执行多个任务, 提高编译任务的速度, 更加有效的使用资源.

你不再需要使用 `kotlin.parallel.tasks.in.project` 属性. 这个属性已被废弃, 并将在下一个主发布版中删除.

标准库

Kotlin 1.5.20 修改了与字符相关的几个函数的平台相关实现, 因此统一了各个平台上的结果:

- 在 Kotlin/Native 和 Kotlin/JS 平台, Char.digitToInt() 函数支持所有的 Unicode 数字.
- 在所有平台统一了 Char.isLowerCase()/isUpperCase() 的实现.

在 Kotlin/Native 和 Kotlin/JS 平台, Char.digitToInt() 函数支持所有的 Unicode 数字

函数 `Char.digitToInt()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/digit-to-int.html) 返回字符所表达的十进制数字值. 在 1.5.20 之前, 这个函数只在 Kotlin/JVM 平台上支持所有的 Unicode 数字字符: Native 和 JS 平台的实现只支持 ASCII 数字.

从现在开始, 在 Kotlin/Native 和 Kotlin/JS 平台, 你可以对任何 Unicode 数字字符调用 `Char.digitToInt()`, 得到字符所表达的数值.

```
fun main() {
//sampleStart
    val ten = '\u0661'.digitToInt() + '\u0039'.digitToInt() // 阿拉伯
文数字1 + 数字9
    println(ten)
//sampleEnd
}
```

在所有平台统一了 Char.isLowerCase()/isUpperCase() 的实现

函数 `Char.isUpperCase()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-upper-case.html) 和 `Char.isLowerCase()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-lower-case.html) 根据字符的大小写返回布尔值. 对 Kotlin/JVM 平台, 函数的实现会检查 Unicode 属性 (https://en.wikipedia.org/wiki/Unicode_character_property) `General_Category` 和 `Other_Uppercase`/`Other_Lowercase`.

在 1.5.20 以前, 其他平台的函数实现工作方式不同, 只考虑了 general category. 在 1.5.20 中, 所有平台的实现全部统一, 全部使用这两个属性来确定字符的大小写:

```
fun main() {
//sampleStart
    val latinCapitalA = 'A' // general category 为 "Lu"
    val circledLatinCapitalA = 'Ⓐ' // 有 "Other_Uppercase" 属性
    println(latinCapitalA.isUpperCase() &&
circledLatinCapitalA.isUpperCase())
```

```
//sampleEnd
}
```

Kotlin 1.5.0 版中的新功能

最终更新: 2024/09/10

发布日期: 2021/05/05 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.5.0 引入了新的语言功能, 基于 IR 的 JVM 编译器后端的稳定版, 以及性能改善, 以及一些微小变更, 比如实验性功能的稳定版, 以及废弃了一些旧功能.

关于这个版本的变更概要, 也可以查看 release blog (https://blog.jetbrains.com/kotlin/2021/04/kotlin-1-5-0-released/).

语言功能

Kotlin 1.5.0 带来了 1.4.30 中提供预览 ("语言功能" in "Kotlin 1.4.30 版中的新功能") 的新语言功能的稳定版:

- 支持 JVM 记录类
- 封闭接口 和 封闭类的改进
- 内联类(Inline Class)

关于这些功能详情, 请参见 这篇 blog (https://blog.jetbrains.com/kotlin/2021/02/new-language-features-preview-in-kotlin-1-4-30/), 以及 Kotlin 文档的对应页面.

支持 JVM 记录类

Java 正在快速演化, 为了让 Kotlin 保持与 Java 的兼容, 我们现在支持它的最新功能之一 – 记录类 (https://openjdk.java.net/jeps/395).

Kotlin 对 JVM 记录类的支持包括双向的交互能力:

- 在 Kotlin 代码中, 你可以使用 Java 记录类, 就象使用通常的带属性的类一样.
- 要在 Java 代码中将 Kotlin 类当作记录类来使用, 可以将它声明为 `data` 类, 并标注 `@JvmRecord` 注解.

```
@JvmRecord
data class User(val name: String, val age: Int)
```

详情请参见 在 Kotlin 中使用 JVM 记录类 (在 Kotlin 中使用 Java 记录类(Record)).

封闭接口

Kotlin 接口现在可以标注 `sealed` 修饰符, 它对接口的功能与对类相同: 在编译时刻能够确定一个封闭接口的所有实现.

```
sealed interface Polygon
```

你可以利用这一点来实现很多功能, 比如, 编写穷尽式(exhaustive) `when` 表达式.

```
fun draw(polygon: Polygon) = when (polygon) {
    is Rectangle -> // ...
    is Triangle -> // ...
    // 这里不需要 else 分支 - 上面已经覆盖了所有可能的实现
}
```

此外, 封闭接口可以实现对类层级更加灵活的限制, 因为一个类可以直接继承多个封闭接口.

```
class FilledRectangle: Polygon, Fillable
```

详情请参见 封闭接口 (封闭类(Sealed Class)与封闭接口(Sealed Interface)).

包范围内的封闭类层级

封闭类的子类现在可以分布在同一个编译单元的同一个包下的所有文件中. 以前, 所有子类必须出现在同一个文件中.

直接子类可以是顶层类, 或嵌套在任意数量的其他有名称的类, 有名称的接口, 或有名称的对象之内.

一个封闭类的子类必须拥有正确限定的名称 – 不能是局部对象或匿名对象.

详情请参见, 封闭类的层级结构 ("继承" in "封闭类(Sealed Class)与封闭接口(Sealed Interface)").

内联类(Inline Class)

内联类(Inline Class) 是一种 基于值的类 (https://github.com/Kotlin/KEEP/blob/master/notes/value-classes.md), 这种类只包含值. 你可以将它用做某种类型的值的封装类, 而不会产生内存分配导致的额外的性能开销.

可以在类名称之前添加 `value` 修饰符来声明内联类:

```
value class Password(val s: String)
```

JVM 后端也需要专门的 `@JvmInline` 注解:

```
@JvmInline
value class Password(val s: String)
```

`inline` 修饰符现在已废弃, 会出现编译警告.

详情请参见 内联类 (内联的值类(Inline value class)).

Kotlin/JVM

Kotlin/JVM 也有了很多改进, 包括内部的, 和面向用户的变更. 重要的变更如下:

- JVM IR 后端的稳定版
- 新的默认 JVM 编译目标: 1.8
- 使用 invokedynamic 实现 SAM 转换
- 使用 invokedynamic 编译 Lambda 表达式
- 废弃 @JvmDefault 和旧的 Xjvm-default 模式
- 可否为 null(Nullability) 注解处理的改进

JVM IR 后端的稳定版

Kotlin/JVM 编译器的 基于 IR 的后端 ("新的 JVM IR 后端" in "Kotlin 1.4.0 版中的新功能") 现在进入稳定版 (Kotlin 各部分组件的稳定性), 并默认启用.

从 Kotlin 1.4.0 (Kotlin 1.4.0 版中的新功能) 开始, 可以预览使用基于 IR 的后端的早期版本, 现在对于语言版本 `1.5`, 它成为了默认后端. 对于更早的语言版本, 继续默认使用旧的后端.

关于 IR 后端优点以及它未来的开发, 更多详情请参见 这篇 blog (https://blog.jetbrains.com/kotlin/2021/02/the-jvm-backend-is-in-beta-let-s-make-it-stable-together/).

如果你需要在 Kotlin 1.5.0 中使用旧的后端, 可以向项目的配置文件添加以下内容:

- 在 Gradle 中:

  Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompil
e> {
    kotlinOptions.useOldBackend = true
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompil
e) {
    kotlinOptions.useOldBackend = true
}
```

- 在 Maven 中:

```
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

新的默认 JVM 编译目标: 1.8

Kotlin/JVM 编译的默认目标版本现在是 `1.8`. 目标版本 `1.6` 已被废弃.

如果你需要针对 JVM 1.6 进行构建, 你仍然可以切换到这个目标版本. 具体方法是:

- 在 Gradle 中切换版本 ("JVM 任务独有的属性" in "Kotlin Gradle plugin 中的编译器选项")
- 在 Maven 中切换版本 ("JVM 独有的属性" in "Maven")
- 在命令行编译器中切换版本 ("-jvm-target version" in "Kotlin 编译器选项")

使用 invokedynamic 实现 SAM 转换

Kotlin 1.5.0 现在使用动态调用 (`invokedynamic`) 来编译 SAM (Single Abstract Method) 转换:

- 如果 SAM 类型 是一个 Java 接口 ("SAM 转换" in "在 Kotlin 中调用 Java 代码"), 可以将任何表达式转换为 SAM

- 如果 SAM 类型是一个 Kotlin 函数接口 ("SAM 转换功能" in "函数式 (SAM) 接口"), 可以将 Lambda 表达式转换为 SAM

新的实现使用 `LambdaMetafactory.metafactory()` (https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html#metafactory-java.lang.invoke.MethodHandles.Lookup-java.lang.String-java.lang.invoke.MethodType-java.lang.invoke.MethodType-java.lang.invoke.MethodHandle-java.lang.invoke.MethodType-), 而且编译期间不再生成辅助的包装类. 这样可以减少应用程序JAR 文件的大小, 改善 JVM 启动时的性能.

要回退到旧的基于匿名类生成的实现方式, 可以添加编译器选项 `-Xsam-conversions=class`.

详情请参见, 如何在 Gradle (Kotlin Gradle plugin 中的编译器选项), Maven ("指定编译器选项" in "Maven"), 以及 命令行编译器 ("编译器选项" in "Kotlin 编译器选项") 中添加编译器选项.

使用 invokedynamic 编译 Lambda 表达式

将普通的 Kotlin Lambda 表达式编译为 invokedynamic 是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-45375) 提供你的反馈意见.

Kotlin 1.5.0 引入实验性的功能, 能够将普通的 Kotlin Lambda 表达式 (which are not converted to an instance of a functional 接口) 编译为动态调用(`invokedynamic`). 这个实现使用 `LambdaMetafactory.metafactory()` (https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html#metafactory-java.lang.invoke.MethodHandles.Lookup-java.lang.String-java.lang.invoke.MethodType-java.lang.invoke.MethodType-java.lang.invoke.MethodHandle-java.lang.invoke.MethodType-), 它能够在运行期高效的生成需要的类, 因此可以产生更轻量的二进制代码, 目前, 与通常的 Lambda 表达式编译相比, 它存在 3 个限制:

- 编译为 invokedynamic 之后的 Lambda 表达式不能序列化.

- 对这样的 Lambda 表达式调用 `toString()` 会产生比较难以阅读的字符串表达.

* 试验性的 `reflect` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect.jvm/reflect.html) API 不支持使用 `LambdaMetafactory` 创建的 Lambda 表达式.

要试用这个功能, 请添加 `-Xlambdas=indy` 编译器选项. 如果你能够在这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-45375) 中提供你的反馈意见, 我们十分感谢.

详情请参见, 如何在 Gradle (Kotlin Gradle plugin 中的编译器选项), Maven ("指定编译器选项" in "Maven"), 以及 命令行编译器 ("编译器选项" in "Kotlin 编译器选项") 中添加编译器选项.

废弃 @JvmDefault 和旧的 Xjvm-default 模式

在 Kotlin 1.4.0 之前, 我们支持 `@JvmDefault` 注解以及 `-Xjvm-default=enable` 和 `-Xjvm-default=compatibility` 模式. 它们用来对 Kotlin 接口中的特定的非抽象成员创建 JVM 默认方法.

在 Kotlin 1.4.0 中, 我们 引入了新的 `Xjvm-default` 模式 (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-m3-generating-default-methods-in-interfaces/), 它会对整个项目切换默认方法的生成.

在 Kotlin 1.5.0 中, 我们废弃了 `@JvmDefault` 和旧的 Xjvm-default 模式: `-Xjvm-default=enable` 和 `-Xjvm-default=compatibility`.

详情请参见 与 Java 交互时的默认方法 ("接口中的默认方法(Default Method)" in "在 Java 中调用 Kotlin 代码").

可否为 null(Nullability) 注解处理的改进

Kotlin 能够从 Java 代码的 可否为 null(Nullability) 注解 ("可否为 null(Nullability) 注解" in "在 Kotlin 中调用 Java 代码") 得到类型可否为 null(Nullability) 信息. Kotlin 1.5.0 对这个功能引入了很多改进:

* 对于编译后的 Java 库作为依赖项使用时, 可以读取其中的类型参数上的 nullability 注解.
* 对于以下类型, 支持 target 为 `TYPE_USE` 的 nullability 注解:
  * 数组
  * 可变参数
  * 域(Field)
  * 类型参数和它的类型边界(bound)
  * 基类和接口的类型参数

- 如果一个 nullability 注解拥有适用于一个类型的多个 target, 而且其中之一是 `TYPE_USE`, 那么会优先使用 `TYPE_USE`. 例如, 如果 `@Nullable` 同时支持 `TYPE_USE` 和 `METHOD` target, 那么 Java 中的方法签名 `@Nullable String[] f()` 会被识别为 Kotlin 的 `fun f(): Array<String?>!`.

对于这些新支持的情况, 在 Kotlin 中调用 Java 时如果使用错误的类型 nullability, 会导致编译警告. 对这样的情况, 可以使用 `-Xtype-enhancement-improvements-strict-mode` 编译器选项来启用严格模式 (产生编译错误).

详情请参见 null 值安全性与平台数据类型 ("Null 值安全性与平台数据类型" in "在 Kotlin 中调用 Java 代码").

Kotlin/Native

Kotlin/Native 有了性能提高, 并更加稳定. 重要的变更包括:

- 性能改善
- 禁用内存泄露检查器

性能改善

在 1.5.0 中, Kotlin/Native 有了很多性能改善, 提升了编译和执行速度.

对 `linuxX64` (只适用于 Linux 主机) 和 `iosArm64` 编译目标, 在 debug 模式中现在可以支持 编译器缓存 (https://blog.jetbrains.com/kotlin/2020/03/kotlin-1-3-70-released/#kotlin-native). 启用编译器缓存后, 除第 1 次编译之外, 大多数 debug 编译可以更快完成. 在我们的测试项目中, 测量结果显示速度提高了大约 200%.

要对新的编译目标使用编译器缓存, 需要明确同意, 方法是向项目的 `gradle.properties` 文件添加以下内容:

- 对于 `linuxX64` 编译目标: `kotlin.native.cacheKind.linuxX64=static`
- 对于 `iosArm64` 编译目标: `kotlin.native.cacheKind.iosArm64=static`

如果你在启用编译器缓存之后遇到任何问题, 请到我们的 问题追踪系统 (https://kotl.in/issue) 中报告.

还有其他改进, 提升了 Kotlin/Native 代码的执行速度:

- Trivial 属性访问器变成了内联模式.

- 字符串字面值的 `trimIndent()` 会在编译期间计算其结果.

禁用内存泄露检查器

内建的 Kotlin/Native 内存泄露检查器默认被禁用.

这个检查器原来计划供内部使用, 它只能发现少数情况下的内存泄露, 而不是所有情况. 而且后来发现存在问题, 可能导致应用程序崩溃. 因此我们决定关闭这个内存泄露检查器.

内存泄露检查器对某些情况仍然是有用的, 例如, 单元测试. 对这样的情况, 你可以添加以下代码来启用它:

```
Platform.isMemoryLeakCheckerActive = true
```

注意, 不推荐对运行期的应用程序启用这个检查器.

Kotlin/JS

Kotlin/JS 在 1.5.0 中有了一些演进变更. 我们正在继续开发 JS IR 编译器后端 (使用 IR 编译器) 的稳定版, 并发布了以下更新:

- 更新到 webpack 版本 5
- 针对 IR 编译器的框架和库

更新到 webpack 5

Kotlin/JS Gradle plugin 现在对浏览器编译目标使用 webpack 5 而不是以前的 webpack 4. 这是 webpack 的一个大版本更新, 因此带来了一些不兼容的变更. 如果你在使用自定义 webpack 配置, 请查看 webpack 5 发布公告 (https://webpack.js.org/blog/2020-10-10-webpack-5-release/).

详情请参见 使用 webpack 构建 Kotlin/JS 项目 ("webpack 打包(Bundling)" in "创建 Kotlin/JS 工程(Project)").

针对 IR 编译器的框架和库

Kotlin/JS IR 编译器现在是 Alpha (Kotlin 各部分组件的稳定性) 版. 它将来可能发生不兼容的变更, 并需要手动迁移. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

在开发 Kotlin/JS 编译器基于 IR 的后端的同时, 我们鼓励并帮助库作者以 `both` 模式构建他们的项

目. 这样可以产生适用于两种 Kotlin/JS 编译器的 artifact, 为新编译器扩大生态环境.

很多广为人知的框架和库已经可以在 IR 后端中使用了: KVision (https://kvision.io/), fritz2 (https://www.fritz2.dev/), doodle (https://github.com/nacular/doodle), 等等. 如果你在你的项目中使用这些框架和库, 你可以使用 IR 后端来构建你的项目, 看看它带来的益处.

如果你在编写自己的库, 使用 'both' 模式编译它 ("针对 IR 编译器开发向后兼容的库" in "使用 IR 编译器"), 这样你的客户也可以在新的编译器中使用你的库.

Kotlin Multiplatform

在 Kotlin 1.5.0 中, 对每个平台选择测试依赖项的工作得到了简化, 现在可以由 Gradle plugin 自动完成.

在跨平台项目中现在可以 使用新的 API 来得到字符种类.

标准库

标准库有了很多变更和改进, 有些实验性功能已经变为稳定版, 还添加了新的功能:

- 无符号整数类型已成为稳定版
- 用于文字大小写变换的 locale 无关 API 已成为稳定版
- 字符到整数编码的转换 API 已成为稳定版
- Path API 已成为稳定版
- 向下取整除法(floored division) 与 mod 操作符
- 时间长度 API 的变更
- 在跨平台代码中可以使用新的 API 来得到字符种类
- 新的集合函数 firstNotNullOf()
- String?.toBoolean() 的严格版本

关于标准库的变更, 详情请参见 这篇 blog (https://blog.jetbrains.com/kotlin/2021/04/kotlin-1-5-0-rc-released).

无符号整数类型已成为稳定版

无符号整数类型 `UInt`, `ULong`, `UByte`, `UShort` 现在已成为 稳定版 (Kotlin 各部分组件的稳定性). 对这些类型的操作, 以及这些类型的范围(range), 数列(progression) 也是如此. 无符号数组及其操

作还处于 Beta 版.

详情请参见 无符号整数类型 (无符号整数(Unsigned Integer)类型).

用于文字大小写变换的 locale 无关 API 已成为稳定版

这次的发布带来一个新的 locale 无关 API, 用于文字的大小写变换. 它可以替代 `toLowerCase()`, `toUpperCase()`, `capitalize()`, 和 `decapitalize()` API 函数, 这些既有的函数是与 locale 相关的. 新 API 可以帮助你避免由于不同的 locale 设定带来的错误.

Kotlin 1.5.0 提供了以下完全 稳定版 (Kotlin 各部分组件的稳定性) 的替代:

- 对于 `String` 函数:

以前的版本	1.5.0 版的替代
`String.toUpperCase()`	`String.uppercase()`
`String.toLowerCase()`	`String.lowercase()`
`String.capitalize()`	`String.replaceFirstChar { it.uppercase() }`
`String.decapitalize()`	`String.replaceFirstChar { it.lowercase() }`

- 对于 `Char` 函数:

以前的版本	1.5.0 版的替代
`Char.toUpperCase()`	`Char.uppercaseChar(): Char` `Char.uppercase(): String`
`Char.toLowerCase()`	`Char.lowercaseChar(): Char` `Char.lowercase(): String`
`Char.toTitleCase()`	`Char.titlecaseChar(): Char` `Char.titlecase(): String`

> 对于 Kotlin/JVM, 也有 `uppercase()`, `lowercase()`, 和 `titlecase()` 函数的覆盖版, 可以明确指定 `Locale` 参数.

旧的 API 函数已经标注为已废弃, 会在未来的发布版中删除.

关于文本处理函数的完整的变更列表, 请参见 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/stdlib/locale-agnostic-case-conversions.md).

字符到整数编码的转换 API 已成为稳定版

从 Kotlin 1.5.0 开始, 新的 "字符到编码" 和 "字符到数字" 转换函数已成为 稳定版 (Kotlin 各部分组件的稳定性). 这些函数替代目前的 API 函数, 旧函数经常会与类似的 "字符串到整数" 转换混淆.

新的 API 去掉了函数名中的混乱, 使得代码的行为更加清晰明确.

这个发布版引入了 `Char` 转换, 分为以下几组清晰命名的函数:

- 得到 `Char` 的整数代码, 以及将指定的代码转换到 `Char`:

```
fun Char(code: Int): Char
fun Char(code: UShort): Char
val Char.code: Int
```

- 将 `Char` 转换为它的数字对应的整数值:

```
fun Char.digitToInt(radix: Int): Int
fun Char.digitToIntOrNull(radix: Int): Int?
```

- `Int` 的扩展函数, 将它表达的非负的单个数字转换为对应的 `Char` 表达:

```
fun Int.digitToChar(radix: Int): Char
```

旧的转换 API, 包括 `Number.toChar()` 及其实现 (`Int.toChar()` 除外), 以及 `Char` 转换到数值类型的扩展函数, 比如 `Char.toInt()`, 现在都已废弃.

关于字符到整数的转换 API, 详情请参见 KEEP

(https://github.com/Kotlin/KEEP/blob/master/proposals/stdlib/char-int-conversions.md).

Path API 已成为稳定版

实验性的 Path API (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io.path/java.nio.file.-path/), 以及对 `java.nio.file.Path` 的扩展, 现在已成为 稳定版 (Kotlin 各部分组件的稳定性).

```
// 使用除 (/) 操作符构造路径
val baseDir = Path("/base")
val subDir = baseDir / "subdirectory"

// 列出一个目录中的文件
val kotlinFiles: List<Path> =
Path("/home/user").listDirectoryEntries("*.kt")
```

详情请参见 Path API ("用于 java.nio.file.Path 的扩展" in "Kotlin 1.4.20 版中的新功能").

向下取整除法(floored division) 与 mod 操作符

标准库添加了新的模运算操作:

- `floorDiv()` 返回 向下取整除法(floored division) (https://en.wikipedia.org/wiki/Floor_and_ceiling_functions) 结果. 这个函数可用于整数类型.
- `mod()` 返回向下取整除法(floored division) 的余数 (*模数(modulus)*). 这个函数可用于所有数值类型.

这些操作看起来与既有的 整数除法 ("数值类型的运算符(Operation)" in "数值类型") and rem() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-int/rem.html) 函数 (或 `%` 操作符) 非常类似, 但它们对于负数的处理不同:

- `a.floorDiv(b)` 与通常的 `/` 不同, `floorDiv` 将结果向下(向更小的整数方向)取整, 而 `/` 将结果截断, 得到更接近 0 的整数.
- `a.mod(b)` 是 `a` 和 `a.floorDiv(b) * b` 之间的差. 它要么是 0, 要么与 `b` 的正负号相同, 而 `a % b` 可能得到不同的正负号.

```
fun main() {
//sampleStart
    println("Floored division -5/3: ${(-5).floorDiv(3)}")
```

```
    println( "Modulus: ${(-5).mod(3)}")

    println("Truncated division -5/3: ${-5 / 3}")
    println( "Remainder: ${-5 % 3}")
//sampleEnd
}
```

时间长度 API 的变更

时间长度 API 是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin 中有一个实验性的 Duration (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/) 类, 表达不同单位的时间长度. 在 1.5.0 中, Duration API 有了以下变更:

- 内部的值表达现在使用 `Long` 而不是 `Double`, 以提供更好的精度.
- 有了新的 API 用于转换到指定的时间单位, 结果类型为 `Long`. 新 API 会替代旧 API, 旧 API 使用 `Double` 值, 现在已废弃. 例如, 新 API `Duration.inWholeMinutes` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/in-whole-minutes.html) 返回 `Long` 表达的时间长度值, 替代了旧的 API `Duration.inMinutes`.
- 有了新的伴随函数, 用于从一个数值构造 `Duration`. 例如, `Duration.seconds(Int)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/seconds.html) 创建一个 `Duration` 对象, 表示整数值的秒. 旧的扩展属性, 比如 `Int.seconds` 现在已废弃.

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
//sampleStart
    val duration = Duration.milliseconds(120000)
    println("There are ${duration.inWholeSeconds} seconds in
${duration.inWholeMinutes} minutes")
```

```
//sampleEnd
}
```

在跨平台代码中可以使用新的 API 来得到字符种类

Kotlin 1.5.0 引入了新的 API , 可以在跨平台项目中得到字符在 Unicode 中的种类(category). 这些函数现在可以在所有平台和共通代码中使用.

检查字符是字母还是数字的函数:

- `Char.isDigit()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-digit.html)
- `Char.isLetter()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-letter.html)
- `Char.isLetterOrDigit()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-letter-or-digit.html)

```
fun main() {
//sampleStart
    val chars = listOf('a', '1', '+')
    val (letterOrDigitList, notLetterOrDigitList) = chars.partition
{ it.isLetterOrDigit() }
    println(letterOrDigitList) // [a, 1]
    println(notLetterOrDigitList) // [+]
//sampleEnd
}
```

检查字符大小写的函数:

- `Char.isLowerCase()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-lower-case.html)
- `Char.isUpperCase()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-upper-case.html)
- `Char.isTitleCase()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-title-case.html)

```
fun main() {
//sampleStart
```

```
    val chars = listOf('ǅ', 'ǈ', 'ǋ', 'ǲ', '1', 'A', 'a', '+')
    val (titleCases, notTitleCases) = chars.partition {
it.isTitleCase() }
    println(titleCases) // [ǅ, ǈ, ǋ, ǲ]
    println(notTitleCases) // [1, A, a, +]
//sampleEnd
}
```

其他函数:

- `Char.isDefined()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-defined.html)
- `Char.isISOControl()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/is-i-s-o-control.html)

属性 `Char.category` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/category.html) 以及它的返回类型 enum 类 `CharCategory` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-char-category/), 现在也可以在跨平台项目中使用了, 其中 `CharCategory` 表示一个字符在 Unicode 中的一般种类.

详情请参见 字符 (字符).

新的集合函数 firstNotNullOf()

新的 `firstNotNullOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first-not-null-of.html) 和 `firstNotNullOfOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first-not-null-of-or-null.html) 函数, 组合了 `mapNotNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-not-null.html) 和 `first()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first.html) 或 `firstOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first-or-null.html). 它们使用自定义的选择函数来对原来的集合进行变换, 并返回第 1 个非 null 的值. 如果不存在非 null 的值, `firstNotNullOf()` 会抛出异常, `firstNotNullOfOrNull()` 会返回 null.

```
fun main() {
//sampleStart
    val data = listOf("Kotlin", "1.5")
    println(data.firstNotNullOf(String::toDoubleOrNull))
    println(data.firstNotNullOfOrNull(String::toIntOrNull))
```

```
//sampleEnd
}
```

String?.toBoolean() 的严格版本

相对于原有的 String?.toBoolean() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/to-boolean.html), 有 2 个新函数引入了大小写相关的严格版本:

- `String.toBooleanStrict()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/to-boolean-strict.html) 除字符串 `true` 和 `false` 之外, 对所有其他输入抛出异常.
- `String.toBooleanStrictOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/to-boolean-strict-or-null.html) 除字符串 `true` 和 `false` 之外, 对所有其他输入返回 null.

```
fun main() {
//sampleStart
    println("true".toBooleanStrict())
    println("1".toBooleanStrictOrNull())
    // println("1".toBooleanStrict()) // 这里会抛出 Exception
//sampleEnd
}
```

kotlin-test 库

kotlin-test (https://kotlinlang.org/api/latest/kotlin.test/) 库引入了一些新功能:

- 简化测试依赖项在跨平台项目中的使用
- 对 Kotlin/JVM 源代码集自动选择测试框架
- 断言函数的更新

简化测试依赖项在跨平台项目中的使用

现在你可以使用 `kotlin-test` 依赖项, 对 `commonTest` 源代码集中的测试代码添加依赖项, Gradle plugin 会对每个测试源代码集推断出对应的平台依赖项:

- 对 JVM 源代码集使用 `kotlin-test-junit`, 参见 对 Kotlin/JVM 源代码集自动选择测试框架

- 对 Kotlin/JS 源代码集使用 `kotlin-test-js`
- 对共通源代码集使用 `kotlin-test-common` 和 `kotlin-test-annotations-common`
- 对 Kotlin/Native 源代码集不会添加额外的依赖项

此外, 你还可以在任何共享的或平台相关的源代码集中, 使用 `kotlin-test` 依赖项.

既有的, 带有明确指定依赖项的 kotlin-test 设置, 在 Gradle 中和在 Maven 中都可以继续使用.

详情请参见 设置测试库的依赖项 ("设置对测试库的依赖项" in "配置 Gradle 项目").

对 Kotlin/JVM 源代码集自动选择测试框架

Gradle plugin 现在会自动选择并添加测试框架的依赖项. 你只需要在共通源代码集中添加依赖项 `kotlin-test`.

Gradle 默认使用 JUnit 4. 因此, `kotlin("test")` 依赖项会解析为 JUnit 4 变体, 名为 `kotlin-test-junit`:

Kotlin

```
kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // 这个设置会导致对
JUnit 4 的传递依赖
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // 这个设置会导致对
```

```
JUnit 4 的传递依赖
                }
            }
        }
    }
```

你可以在 test task 中调用 useJUnitPlatform() (https://docs.gradle.org/current/javadoc/org/gradle/api/tasks/testing/Test.html#useJUnitPlatform) 或 useTestNG() (https://docs.gradle.org/current/javadoc/org/gradle/api/tasks/testing/Test.html#useTestNG) 来选择 JUnit 5 或 TestNG:

```
tasks {
    test {
        // 使用 TestNG
        useTestNG()
        // 或
        // 使用 JUnit Platform (a.k.a. JUnit 5)
        useJUnitPlatform()
    }
}
```

你可以向项目的 gradle.properties 添加 kotlin.test.infer.jvm.variant=false, 来禁用测试框架的自动选择.

详情请参见 设置测试库的依赖项 ("设置对测试库的依赖项" in "配置 Gradle 项目").

断言函数的更新

这个发布版带来了新的断言函数, 并改进了既有的函数.

kotlin-test 库现在包含以下功能:

- **检查一个值的类型**

  你可以使用新的 assertIs<T> 和 assertIsNot<T> 来检查一个值的类型:

```
@Test
fun testFunction() {
    val s: Any = "test"
    assertIs<String>(s)  // 如果断言失败会抛出 AssertionError, 错误信
```

```
息包含 s 的实际类型
    // 可以现在打印 s.length, 因为在 assertIs 中已经判断了 s 的类型是字符串
    println("${s.length}")
}
```

由于类型擦除, 在以下示例中, 这个断言函数只能检查 value 是不是 List 类型, 但不能检查它是不是具体的 String 元素类型构成的 List: `assertIs<List<String>>(value)`.

- **对数组, 序列(Sequence), 以及任意的 iterable, 比较容器内容**

  对 `assertContentEquals()` 函数, 有了一组新的覆盖版本, 对没有实现 结构相等 ("结构相等" in "相等判断") 的各种集合, 可以比较其内容:

```
@Test
fun test() {
    val expectedArray = arrayOf(1, 2, 3)
    val actualArray = Array(3) { it + 1 }
    assertContentEquals(expectedArray, actualArray)
}
```

- **对于 `Double` 和 `Float` 数值, `assertEquals()` 和 `assertNotEquals()` 函数有了新的覆盖版本**

  对 `assertEquals()` 函数, 有了新的覆盖版本, 可以使用绝对精度比较 2 个 `Double` 或 `Float` 数值. 精度值通过比较函数的第 3 个参数指定:

```
@Test
fun test() {
    val x = sin(PI)

    // 精度参数
    val tolerance = 0.000001

    assertEquals(0.0, x, tolerance)
}
```

- **检查集合和元素内容的新函数**

  你现在可以使用 `assertContains()` 函数来检查集合或元素是否包含某个内容. 这个函数可以用于拥有 `contains()` 操作符的 Kotlin 集合和元素, 比如 `IntRange`, `String`, 等等:

```
@Test
fun test() {
    val sampleList = listOf<String>("sample", "sample2")
    val sampleString = "sample"
    assertContains(sampleList, sampleString)  // 元素在集合中存在
    assertContains(sampleString, "amp")       // 子字符串在字符串中存
在
}
```

- **assertTrue(), assertFalse(), expect() 函数现在成为内联函数**

  从现在开始, 你可以将这些函数作为内联函数来使用, 因此可以在 Lambda 表达式之内调用 挂起函数 (挂起函数(Suspending Function)的组合):

```
@Test
fun test() = runBlocking<Unit> {
    val deferred = async { "Kotlin is nice" }
    assertTrue("Kotlin substring should be present") {
        deferred.await().contains("Kotlin")
    }
}
```

kotlinx 库

和 Kotlin 1.5.0 一起, 我们还发布了 kotlinx 库的新版本:

- kotlinx.coroutines 1.5.0-RC
- kotlinx.serialization 1.2.1
- kotlinx-datetime 0.2.0

Coroutines 1.5.0-RC

kotlinx.coroutines 1.5.0-RC (https://github.com/Kotlin/kotlinx.coroutines/releases/tag/1.5.0-RC) 的新功能包括:

- 新的通道(Channel) API (通道(Channel))

- 与 reactive 集成 ("Reactive Extension" in "异步编程(Asynchronous Programming)技术")的稳定版
- 其他

从 Kotlin 1.5.0 开始, 禁用了 实验性协程 ("删除了已废弃的实验性协程" in "Kotlin 1.4.0 版中的新功能"), 并且不再支持 `-Xcoroutines=experimental` 标记.

详情请参见 changelog (https://github.com/Kotlin/kotlinx.coroutines/releases/tag/1.5.0-RC) 以及 `kotlinx.coroutines` 1.5.0 release blog (https://blog.jetbrains.com/kotlin/2021/05/kotlin-coroutines-1-5-0-released/).

serialization 1.2.1

`kotlinx.serialization` 1.2.1 (https://github.com/Kotlin/kotlinx.serialization/releases/tag/v1.2.1) 的新功能包括:

- JSON 序列化的性能改善
- 在 JSON 序列化中支持多名称
- 实验性功能: 从 `@Serializable` 类生成 .proto schema
- 其他

详情请参见 changelog (https://github.com/Kotlin/kotlinx.serialization/releases/tag/v1.2.1) 以及 `kotlinx.serialization` 1.2.1 release blog (https://blog.jetbrains.com/kotlin/2021/05/kotlinx-serialization-1-2-released/).

dateTime 0.2.0

`kotlinx-datetime` 0.2.0 (https://github.com/Kotlin/kotlinx-datetime/releases/tag/v0.2.0) 的新功能包括:

- `@Serializable` Datetime 对象
- `DateTimePeriod` 和 `DatePeriod` 的规范化 API
- 其他

详情请参见 changelog (https://github.com/Kotlin/kotlinx-datetime/releases/tag/v0.2.0) 以及 `kotlinx-datetime` 0.2.0 release blog (https://blog.jetbrains.com/kotlin/2021/05/kotlinx-datetime-0-2-0-is-out/).

迁移到 Kotlin 1.5.0

当 Kotlin plugin 1.5.0 可用之后, IntelliJ IDEA 和 Android Studio 会建议你更新这个版本.

要将既有的项目迁移到 Kotlin 1.5.0, 只需要修改 Kotlin 版本到 `1.5.0`, 然后重新导入你的 Gradle 或 Maven 项目. 详情请参见 如何更新到 Kotlin 1.5.0 ("更新到新的发布版" in "Kotlin 的发布版本").

要使用 Kotlin 1.5.0 创建新项目, 请更新 Kotlin plugin, 并通过菜单 **File** | **New** | **Project** 运行项目向导.

新的命令行编译器可以通过 GitHub release 页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.5.0) 下载.

Kotlin 1.5.0 是一个 功能发布版 ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化"), 因此可能在语言层带来不兼容的变更. 关于这些变更的完整列表, 请参见 Kotlin 1.5 兼容性指南 (Kotlin 1.5 兼容性指南).

Kotlin 1.4.30 版中的新功能

最终更新: 2024/09/10

发布日期: 2021/02/03 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.4.30 提供了新的语言功能的预览版, 将 Kotlin/JVM 编译器的新的 IR 后端升级到 Beta, 并带来了很多性能和功能的改进.

关于这个版本的变更概要, 可以查看 这篇 blog (https://blog.jetbrains.com/kotlin/2021/01/kotlin-1-4-30-released/).

语言功能

Kotlin 1.5.0 将会发布一些新的语言功能 – 支持 JVM 记录类(Record), 封闭接口(Sealed Interface), 以及内联类(Inline Class)的稳定版. 在 Kotlin 1.4.30 中, 你可以通过预览模式试用这些新功能和新改进. 如果你能够在相应的 YouTrack ticket 中提供你的反馈意见, 我们将会非常感谢, 你的反馈能够帮助我们在 1.5.0 正式发布之前解决这些问题.

- 支持 JVM 记录类(Record)
- 封闭接口(Sealed Interface) 和 封闭类(Sealed Class)的改进
- 内联类(Inline Class)的改进

要通过预览模式启用这些新功能和新改进, 你需要添加特定的编译器选项, 来表示你明确同意使用. 详情请阅读下面的章节.

关于新功能预览, 详情请参见 这篇 blog (https://blog.jetbrains.com/kotlin/2021/01/new-language-features-preview-in-kotlin-1-4-30).

支持 JVM 记录类(Record)

JVM 记录类(Record)功能是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要明确同意使用(Opt-in)(详情请参见下文), 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-42430) 提供你的反馈意见.

JDK 16 版 (https://openjdk.java.net/projects/jdk/16/) 中计划稳定新类型的 Java 类, 名为 Record (https://openjdk.java.net/jeps/395). 为了充分利用 Kotlin 的功能, 并保证与 Java 的交互

能力, Kotlin 会增加对记录类的支持(实验性功能).

你可以在 Kotlin 中使用 Java 中声明的记录类, 就和其他有属性的类一样. 不需要其他任何步骤.

从 1.4.30 开始, 你可以在 Kotlin 中对一个 数据类 (数据类(Data Class)) 使用 `@JvmRecord` 注解, 来声明记录类:

```
@JvmRecord
data class User(val name: String, val age: Int)
```

要试用 JVM 记录类功能的预览版, 请添加编译器选项 `-Xjvm-enable-preview` 和 `-language-version 1.5`.

我们还在继续完善这个功能, 如果你能通过这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-42430) 提供你的反馈意见, 我们会非常感谢.

关于这个功能的具体实现, 限制, 以及语法, 请参见 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/jvm-records.md).

封闭接口(Sealed Interface)

> ⚠ 封闭接口(Sealed Interface)是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要明确同意使用(Opt-in)(详情请参见下文), 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-42433) 提供你的反馈意见.

在 Kotlin 1.4.30 中, 我们发布了 *封闭接口(Sealed Interface)* 的原型. 这个功能是对封闭类的补充, 可以用来构建更加灵活的类层级关系约束.

封闭接口可以用作 "internal" 接口, 不能在同一个模块之外实现. 你可以利用这一点来实现很多功能, 比如, 编写穷尽式(exhaustive) `when` 表达式.

```
sealed interface Polygon

class Rectangle(): Polygon
class Triangle(): Polygon

// when() 语句是穷尽式(exhaustive)的: 这个模块编译完成后, 不可能再出现其它的
polygon 实现
fun draw(polygon: Polygon) = when (polygon) {
    is Rectangle -> // ...
```

```
    is Triangle -> // ...
}
```

另一种使用场景是: 使用封闭接口, 你可以从两个或多个封闭的超类继承一个类.

```
sealed interface Fillable {
   fun fill()
}
sealed interface Polygon {
   val vertices: List<Point>
}

class Rectangle(override val vertices: List<Point>): Fillable,
Polygon {
   override fun fill() { /*...*/ }
}
```

要试用封闭接口的预览版, 请添加编译器选项 `-language-version 1.5`. 切换到这个版本之后, 你就可以对接口使用 `sealed` 修饰符了. 如果你能通过这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-42433) 提供你的反馈意见, 我们会非常感谢.

更多详情请参见 封闭接口(Sealed Interface) (封闭类(Sealed Class)与封闭接口(Sealed Interface)).

包范围内的封闭类(Sealed Class)层级结构

> ⚠ 包范围内的封闭类(Sealed Class)层级结构是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要明确同意使用(Opt-in)(详情请参见下文), 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 YouTrack (https://youtrack.jetbrains.com/issue/KT-42433) 提供你的反馈意见.

封闭类(Sealed Class)现在可以组成更加灵活的层级结构. 子类可以存在于同一个编译单元同一个包的所有源代码文件中. 在以前的版本中, 所有子类必须存在于同一个源代码文件中.

直接子类可以是顶层类, 也可以内嵌在任意数量的其他命名类, 命名接口, 或命名对象之内. 封闭类的子类必须拥有适当的限定名称 – 不能是局部类, 也不能是匿名对象.

要试用包范围内的封闭类层级结构功能, 请添加编译器选项 `-language-version 1.5`. 如果你能通过这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-42433) 提供你的反馈意见, 我

们会非常感谢.

更多详情请参见 包范围内的封闭类(Sealed Class)层级结构 ("继承" in "封闭类(Sealed Class)与封闭接口(Sealed Interface)").

内联类(Inline Class)的改进

内联的数据类目前是 Beta 版 (Kotlin 各部分组件的稳定性). 已经基本稳定, 但未来可能需要执行一些迁移工作. 我们会尽量减少需要你进行的代码变更工作. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-42434) 提供你的反馈意见.

Kotlin 1.4.30 将 内联类(Inline Class) (内联的值类(Inline value class)) 升级到 Beta 版 (Kotlin 各部分组件的稳定性), 并带来以下功能和改进:

- 由于内联类是 基于值的 (https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/doc-files/ValueBased.html), 因此你可以使用 `value` 修饰符来定义内联类. `inline` 和 `value` 修饰符现在是相互等价的. 在未来的 Kotlin 版本中, 我们计划废弃 `inline` 修饰符.

  从现在开始, 对于 JVM 后端, Kotlin 要求在类的声明之前添加 `@JvmInline` 注解:

```
inline class Name(private val s: String)

value class Name(private val s: String)

// 对于 JVM 后端
@JvmInline
value class Name(private val s: String)
```

- 内联类可以拥有 `init` 代码段. 你可以添加需要在类实例创建之后立即执行的代码:

```
@JvmInline
value class Negative(val x: Int) {
  init {
      require(x < 0) { }
  }
}
```

- 在 Java 代码中使用内联类调用函数: 在 Kotlin 1.4.30 之前, 由于代码混淆(mangle), 你不能从 Java 代码中调用接受内联类参数的函数. 从现在开始, 你可以手动关闭代码混淆. 要从 Java 代码调用这样的函数, 你需要在函数声明前添加 `@JvmName` 注解:

  ```
  inline class UInt(val x: Int)

  fun compute(x: Int) { }

  @JvmName("computeUInt")
  fun compute(x: UInt) { }
  ```

- 在这个发布版中, 我们修改了对函数的代码混淆机制, 以便修正一些不正确的行为. 这些变更会导致 ABI 变化.

  从 1.4.30 开始, Kotlin 编译器默认使用新的代码混淆机制. 可以使用 `-Xuse-14-inline-classes-mangling-scheme` 编译器 flag 来强制编译器使用使用旧的 1.4.0 代码混淆机制, 以保证二进制兼容性.

Kotlin 1.4.30 将内联类升级为 Beta 版, 我们计划在未来的发布版中将它升级为稳定版. 如果你能通过这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-42434) 提供你的反馈意见, 我们会非常感谢.

要试用内联类的预览版, 请添加编译器选项 `-Xinline-classes` 或 `-language-version 1.5`.

关于代码混淆算法, 更多详情请参见 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/inline-classes.md).

更多详情请参见 内联类 (内联的值类(Inline value class)).

Kotlin/JVM

JVM IR 编译器后端升级为 Beta 版

Kotlin/JVM 的 基于 IR 的编译器后端 ("统一的后端和扩展性" in "Kotlin 1.4.0 版中的新功能"), 在 1.4.0 版引入时是 Alpha 版 (Kotlin 各部分组件的稳定性), 现在升级为 Beta 版. 这是稳定版之前的最后一个测试版, 稳定版发布后, Kotlin/JVM 编译器会默认使用 IR 后端.

我们现在会去掉使用 IR 编译器产生的二进制文件的限制. 以前的版本中, 你必须启用新的后端, 然后才能使用新的 JVM IR 后端编译的代码. 从 1.4.30 开始, 不再存在这样的限制, 因此你可以使用新的后端来构建供第三方使用的组件, 比如库. 请试用新后端的 Beta 版本, 并通过我们的 issue tracker (https://kotl.in/issue) 提供你的反馈意见.

要启用新的 JVM IR 后端, 请向项目的构建脚本添加以下设置:

- Gradle:

  Kotlin

  ```
  tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompil
  e::class) {
    kotlinOptions.useIR = true
  }
  ```

  Groovy

  ```
  tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompil
  e) {
    kotlinOptions.useIR = true
  }
  ```

- Maven:

  ```
  <configuration>
      <args>
          <arg>-Xuse-ir</arg>
      </args>
  </configuration>
  ```

关于 JVM IR 后端的变更, 更多详情请参见 这篇 blog (https://blog.jetbrains.com/kotlin/2021/01/the-jvm-backend-is-in-beta-let-s-make-it-stable-together).

Kotlin/Native

性能改善

在 1.4.30 中, Kotlin/Native 有了很多性能改善, 使得编译速度更加提升. 比如, 在 使用 Kotlin Multiplatform Mobile 开发的网络和数据存储 (https://github.com/kotlin-hands-on/kmm-

networking-and-data-storage/tree/final) 示例项目中, 重新构建框架所需要的时间从 9.5 秒 (1.4.10 版) 减少到了 4.5 秒 (1.4.30 版).

Apple watchOS 64-bit 模拟器编译目标

从 watchOS 版本 7.0 开始, x86 模拟器编译目标已被废弃. 为与 watchOS 的最新版本保持一致, Kotlin/Native 增加了新的编译目标 `watchosX64`, 用于在 64-bit 架构运行模拟器.

支持 Xcode 12.2 库

我们增加了对随 Xcode 12.2 发布的新的库的支持. 你现在可以在 Kotlin 代码中使用这些库了.

Kotlin/JS

顶级属性(top-level property)的延迟初始化(Lazy initialization)

顶级属性(top-level property)的延迟初始化(Lazy initialization)是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要明确同意使用(Opt-in)(详情请参见下文), 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-44320) 提供你的反馈意见.

Kotlin/JS 的 IR 后端 (使用 IR 编译器) 有了顶级属性(top-level property)的延迟初始化(Lazy initialization)功能的原型实现. 这个功能可以在应用程序启动时减少需要初始化的顶级属性, 可以显著改善应用程序的启动时间.

我们会继续改进延迟初始化功能, 我们希望你试用目前的原型实现, 并通过这个 YouTrack ticket (https://youtrack.jetbrains.com/issue/KT-44320) 或官方 Kotlin Slack (https://kotlinlang.slack.com) (请在 这里 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up) 得到邀请) 的 `#javascript` (https://kotlinlang.slack.com/archives/C0B8L3U69) 频道, 反馈你的想法和试用结果.

要使用延迟初始化功能, 请在使用 JS IR 编译器编译代码时添加 `-Xir-property-lazy-initialization` 编译器选项.

Gradle 项目的改进

支持 Gradle 配置缓存

从 1.4.30 开始, Kotlin Gradle plugin 支持 配置缓存 (https://docs.gradle.org/current/userguide/configuration_cache.html) 功能. 这个功能会提高

构建过程的速度: 一旦你执行命令, Gradle 会执行配置过程, 并计算任务图(task graph). Gradle 会缓存计算结果, 并在以后的构建中重用这些结果.

要启用这个功能, 你可以 使用 Gradle 命令 (https://docs.gradle.org/current/userguide/configuration_cache.html#config_cache:usage) 或 设置 IntelliJ based IDE (https://docs.gradle.org/current/userguide/configuration_cache.html#config_cache:ide:intellij).

标准库

针对大写/小写文字的 Locale 无关 API

Locale 无关 API 功能是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-42437) 提供你的反馈意见.

本次发布版增加了改变字符串和字符大小写的 Locale 无关 API (实验性功能). 现在的 `toLowerCase()`, `toUpperCase()`, `capitalize()`, `decapitalize()` API 函数是与 Locale 相关的. 也就是说, 不同的平台 Locale 设置可能影响代码的行为. 比如, 在 Turkish locale 中, 使用 `toUpperCase` 来转换字符串 "kotlin", 结果会是 "KOTLİN", 而不是 "KOTLIN".

```
// 使用现在的 API
println("Needs to be capitalized".toUpperCase()) // 结果是: NEEDS TO
BE CAPITALIZED

// 使用新 API
println("Needs to be capitalized".uppercase()) // 结果是: NEEDS TO BE
CAPITALIZED
```

Kotlin 1.4.30 提供了以下替代函数:

- 对 `String` 函数:

以前的版本	1.4.30 的替代函数
`String.toUpperCase()`	`String.uppercase()`
`String.toLowerCase()`	`String.lowercase()`
`String.capitalize()`	`String.replaceFirstChar { it.uppercase() }`
`String.decapitalize()`	`String.replaceFirstChar { it.lowercase() }`

* `Char` 函数:

以前的版本	1.4.30 的替代函数
`Char.toUpperCase()`	`Char.uppercaseChar(): Char` `Char.uppercase(): String`
`Char.toLowerCase()`	`Char.lowercaseChar(): Char` `Char.lowercase(): String`
`Char.toTitleCase()`	`Char.titlecaseChar(): Char` `Char.titlecase(): String`

> 对于 Kotlin/JVM 平台, 还有明确使用 `Locale` 参数的 overload 版本的 `uppercase()`, `lowercase()`, 和 `titlecase()` 函数

关于文字处理函数的所有变更, 请参见 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/stdlib/locale-agnostic-string-conversions.md).

明确的 "字符到代码" 和 "字符到数值" 转换

> 意义明确的 `Char` 转换 API 是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题

追踪系统 (https://youtrack.jetbrains.com/issue/KT-44333) 提供你的反馈意见.

目前的 Char 到数值的转换函数, 返回不同数值类型表达的 UTF-16 代码, 经常会与类似的字符串到整数值的转换混淆, 后一种转换会返回字符串表示的数值:

```
"4".toInt() // 返回 4
'4'.toInt() // 返回 52
// 而且没有共通函数可以对字符 '4' 返回数值 4
```

为了避免这样的混淆, 我们决定将 Char 转换分离为以下两组名称更加清晰的函数:

- 第一组的函数, 用于得到 Char 的整数代码, 以及通过指定的代码构建 Char:

  ```
  fun Char(code: Int): Char
  fun Char(code: UShort): Char
  val Char.code: Int
  ```

- 第二组的函数, 将 Char 转换为它所表达的数字的整数值:

  ```
  fun Char.digitToInt(radix: Int): Int
  fun Char.digitToIntOrNull(radix: Int): Int?
  ```

- Int 的一个扩展函数, 可以将整数表达的单个非负数字, 转换为对应的 Char 表达:

  ```
  fun Int.digitToChar(radix: Int): Char
  ```

更多详情请参见 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/stdlib/char-int-conversions.md).

序列化库的更新

随 Kotlin 1.4.30 一起, 我们还发布了 kotlinx.serialization 1.1.0-RC (https://github.com/Kotlin/kotlinx.serialization/releases/tag/v1.1.0-RC), 其中包含一些新功能:

- 支持内联类的序列化
- 支持无符号基本类型(Unsigned Primitive Type)的序列化

支持内联类的序列化

从 Kotlin 1.4.30 开始, 你可以让内联类 可序列化 (序列化):

```
@Serializable
inline class Color(val rgb: Int)
```

> 这个功能需要新的 1.4.30 IR 编译器.

当可序列化内联类被用在另一个可序列化类之内时, 序列化框架不会对可序列化内联类装箱.

更多详情请参见 `kotlinx.serialization` 的 文档 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/inline-classes.md#serializable-inline-classes).

支持无符号基本类型(Unsigned Primitive Type)的序列化

从 1.4.30 开始, 你可以对无符号基本类型(Unsigned Primitive Type): `UInt`, `ULong`, `UByte`, 和 `UShort`, 使用 kotlinx.serialization (https://github.com/Kotlin/kotlinx.serialization) 的标准的 JSON 序列化器:

```
@Serializable
class Counter(val counted: UByte, val description: String)
fun main() {
    val counted = 239.toUByte()
    println(Json.encodeToString(Counter(counted, "tries")))
}
```

更多详情请参见 `kotlinx.serialization` 的 文档 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/inline-classes.md#unsigned-types-support-json-only).

Kotlin 1.4.20 版中的新功能

最终更新: 2024/09/10

发布日期: 2020/11/23 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin 1.4.20 带来了很多新的实验性功能特性, 并对既有的提供了功能特性很多 bug 修正和改进, 包括 1.4.0 中添加的那些新功能特性.

关于新功能特性, 也可以阅读 这篇 Blog (https://blog.jetbrains.com/kotlin/2020/11/kotlin-1-4-20-released/), 其中包含很多示例.

Kotlin/JVM

Kotlin/JVM 改进的目标是为了让它跟上现代 Java 版本的功能特性:

- Java 15 编译目标
- invokedynamic 字符串 拼接

Java 15 编译目标

现在 Kotlin/JVM 编译目标可以使用 Java 15.

invokedynamic 字符串拼接

> `invokedynamic` 字符串拼接是 实验性功能 (Kotlin 各部分组件的稳定性). 这个功能随时可能抛弃或改变. 使用这个功能需要使用者同意(Opt-in)(详情请参见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

对于 JVM 9 以上版本的编译目标, Kotlin 1.4.20 能够将字符串拼接编译为 动态调用(dynamic invocation) (https://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html#invokedynamic), 进而改善性能.

现在, 这个功能特性还是实验性功能, 包含以下使用场景:

- `String.plus`: 操作符形式(`a + b`), 明确调用形式(`a.plus(b)`), 以及方法参照形式(`(a::plus)(b)`).
- `toString`: 在内联类和数据类中.

- 字符串模板, 带单个非常数参数的情况除外(参见 KT-42457 (https://youtrack.jetbrains.com/issue/KT-42457)).

要启用 `invokedynamic` 字符串拼接功能, 请添加 `-Xstring-concat` 编译器参数, 指定以下值:

- `indy-with-constants`, 对字符串执行 `invokedynamic` 拼接, 使用 StringConcatFactory.makeConcatWithConstants() (https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/StringConcatFactory.html#makeConcatWithConstants-java.lang.invoke.MethodHandles.Lookup-java.lang.String-java.lang.invoke.MethodType-java.lang.String-java.lang.Object...-).
- `indy`: 对字符串执行 `invokedynamic` 拼接, 使用 StringConcatFactory.makeConcat() (https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/StringConcatFactory.html#makeConcat-java.lang.invoke.MethodHandles.Lookup-java.lang.String-java.lang.invoke.MethodType-).
- `inline`: 切换回原来的拼接方法, 使用 `StringBuilder.append()`.

Kotlin/JS

Kotlin/JS 还在继续快速演进, 在 1.4.20 中你会看到很多实验性的功能特性和改进:

- Gradle DSL 的变更
- 新的向导模板
- IR 编译器忽略编译错误

Gradle DSL 的变更

Kotlin/JS 的 Gradle DSL 有了很多更新, 可以简化项目的配置和自定义. 这些更新包括 webpack 配置的调整, 自动生成的 `package.json` 文件的修正, 以及对传递依赖的控制的改进.

对 webpack 配置的单点控制

对浏览器编译目标可以使用新的配置代码段 `commonWebpackConfig`. 在这个代码段内, 可以集中在一处调整共通设置, 而不必对 `webpackTask`, `runTask`, 和 `testTask` 任务进行重复配置.

要对这 3 个任务默认启用 CSS 支持, 请在你的项目的 `build.gradle(.kts)` 文件中添加以下代码:

```
browser {
    commonWebpackConfig {
```

```
        cssSupport.enabled = true
    }
    binaries.executable()
}
```

更多详情请参见 webpack 打包(Bundling)配置 ("webpack 打包(Bundling)" in "创建 Kotlin/JS 工程(Project)").

通过 Gradle 自定义 package.json 文件

要对你的 Kotlin/JS 包的管理和发布进行更加精确的控制, 现在可以通过 Gradle DSL 向项目文件 `package.json` (https://nodejs.dev/learn/the-package-json-guide) 添加属性.

要向你的 `package.json` 文件添加自定义的项目, 请在编译任务的 `packageJson` 代码段中使用 `customField` 函数:

```
kotlin {
    js(BOTH) {
        compilations["main"].packageJson {
            customField("hello", mapOf("one" to 1, "two" to 2))
        }
    }
}
```

更多详情请参见 自定义 `package.json` 文件 ("自定义 package.json 文件" in "创建 Kotlin/JS 工程(Project)").

可选择的 yarn 依赖项解析

> 可选择的 yarn 依赖项解析是 实验性功能 (Kotlin 各部分组件的稳定性). 这个功能随时可能抛弃或改变. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin 1.4.20 提供一种方法来配置 Yarn 的 可选择的依赖项解析 (https://classic.yarnpkg.com/en/docs/selective-version-resolutions/) - 用来覆盖你依赖的包的依赖项的机制.

在 Gradle 中可以通过 `YarnPlugin` 中的 `YarnRootExtension` 使用这个功能. 要对你的项目影响一个包解析的版本, 请使用 `resolution` 函数, 传入参数是包名称选择器 (与 Yarn 指定的相同), 以及它应当解析的版本.

```
rootProject.plugins.withType<YarnPlugin> {
    rootProject.the<YarnRootExtension>().apply {
        resolution("react", "16.0.0")
        resolution("processor/decamelize", "3.0.0")
    }
}
```

这里, 你的 *所有* 需要 `react` 的 npm 依赖项都将得到版本 `16.0.0`, 而 `processor` 对它的依赖项 `decamelize` 将会得到版本 `3.0.0`.

禁用粗粒度 workspace

> 禁用粗粒度 workspace 是 实验性功能 (Kotlin 各部分组件的稳定性). 这个功能随时可能抛弃或改变. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

为了提高构建速度, Kotlin/JS Gradle plugin 只安装特定 Gradle 任务所需要的依赖项. 比如, `webpack-dev-server` 包, 只有在你执行某个 `*Run` 任务时才会安装, 而在执行 assemble 任务时不会安装. 当你并行运行多个 Gradle 任务时, 这种行为可能带来潜在的问题. 当需求的依赖项发生冲突时, npm 包的两种不同安装会导致错误.

为了解决这个问题, Kotlin 1.4.20 包含了一个参数来禁用这种 *粗粒度 workspace*. 在 Gradle 中可以通过 `YarnPlugin` 中的 `YarnRootExtension` 来使用这个功能特性. 要使用它, 请在你的 `build.gradle.kts` 文件中添加以下代码段:

```
rootProject.plugins.withType<YarnPlugin> {
    rootProject.the<YarnRootExtension>().disableGranularWorkspaces()
}
```

新的向导模板

在创建项目阶段, 为了提供更方便的方法来自定义你的项目, Kotlin 项目创建向导为 Kotlin/JS 应用程序提供了新的模板:

- **Browser 应用程序** - 一个最小的 Kotlin/JS Gradle 项目, 运行在浏览器内.
- **React 应用程序** - 一个 React 应用程序, 使用适当的 `kotlin-wrappers`. 它提供参数来集成样式表(style sheet), 导航组件, 或状态容器.

- **Node.js 应用程序** - 一个最小的项目, 运行在 Node.js 环境内. 它提供参数来包含实验性的 `kotlinx-nodejs` 包.

IR 编译器忽略编译错误

> *忽略编译错误* 模式是 实验性功能 (Kotlin 各部分组件的稳定性). 这个功能随时可能抛弃或改变. 使用这个功能需要使用者同意(Opt-in)(详情请参见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin/JS 的 IR 编译器 (使用 IR 编译器) 有一个新的实验性模式 - *带错误编译* 模式. 在这种模式下, 即使你的代码包含错误也可以运行, 比如, 你可能希望试验某部分功能, 虽然整个应用程序还未完成.

这个模式有两种错误宽容策略:

- `SEMANTIC`: 编译器会接受那些语法正确, 但语义不正确的代码, 比如 `val x: String = 3`.
- `SYNTAX`: 编译器会接受任何代码, 即使包含语法错误.

要允许带错误编译, 请添加 `-Xerror-tolerance-policy=` 编译器参数, 指定上述值.

更多详情请参见使用 Kotlin/JS IR 编译器 忽略编译错误 ("忽略编译错误" in "使用 IR 编译器").

Kotlin/Native

在 1.4.20 中, Kotlin/Native 的优先任务是改进性能, 以及改进既有的功能特性. 值得注意的改进包括:

- 逃逸分析(Escape analysis)
- 性能改进与 bug 修复
- 包装 Objective-C 异常(需要使用者同意)
- CocoaPods plugin 改进
- 支持 Xcode 12 库

逃逸分析(Escape analysis)

> 逃逸分析机制是 实验性功能 (Kotlin 各部分组件的稳定性). 这个功能随时可能抛弃或改变. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin/Native 有了一个新的 逃逸分析(escape analysis) (https://en.wikipedia.org/wiki/Escape_analysis) 机制的原型. 它会将特定的对象分配在栈(stack)中, 而不是分配在堆(heap)中, 因此改进了运行期性能. 在我们的基准测试中, 这个机制表现出 10% 的平均性能改进, 我们还会继续改进它, 让它能够更多的提高程序运行速度.

对发布构建(release build) (使用 `-opt` 编译器参数), 会在单独的编译阶段中运行逃逸分析.

如果你希望禁用逃逸分析, 请使用 `-Xdisable-phases=EscapeAnalysis` 编译器参数.

性能改进与 bug 修复

Kotlin/Native 的很多组件有了性能改进和 bug 修复, 包括 1.4.0 中添加的部分, 比如, 代码共用机制 ("在类似的平台上共用代码" in "在不同的平台之间共用代码").

包装 Objective-C 异常(需要使用者同意)

> Objective-C 异常包装机制是 实验性功能 (Kotlin 各部分组件的稳定性). 这个功能随时可能抛弃或改变. 使用这个功能需要使用者同意(Opt-in)(详情请参见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin/Native 现在可以处理 Objective-C 代码在运行期抛出的异常, 避免程序崩溃.

你可以同意使用(opt in) 将 `NSException` 包装为 Kotlin 的 `ForeignException` 类型的异常. 这些异常会保存原来的 `NSException` 的引用. 因此你可以得到原始错误的信息, 并进行正确的处理.

要启用 Objective-C 异常的包装, 请在 `cinterop` 调用中指定 `-Xforeign-exception-mode objc-wrap` 参数, 或对 `.def` 文件添加 `foreignExceptionMode = objc-wrap` 属性. 如果你使用 CocoaPods 集成 (CocoaPods 概述与设置), 请在一个依赖项的 `pod {}` 构建脚本代码段中指定参数, 比如:

```
pod("foo") {
    extraOpts = listOf("-Xforeign-exception-mode", "objc-wrap")
}
```

不启用这个功能时, 默认行为仍然保持不变: 当 Objective-C 代码抛出异常时, 程序会终止.

CocoaPods plugin 改进

Kotlin 1.4.20 继续改进与 CocoaPods 的集成. 你可以试用以下新功能特性:

- 任务执行的改进
- DSL 扩展
- 与 Xcode 集成的更新

任务执行的改进

CocoaPods plugin 的任务执行流程有改进. 比如, 如果你添加一个新的 CocoaPods 依赖项, 既有的依赖项不会重新构建. 添加一个额外的编译目标也不会影响既有编译目标依赖项的重新构建.

DSL 扩展

向你的 Kotlin 项目添加 CocoaPods (CocoaPods 概述与设置) 依赖项的 DSL, 有了新的功能.

除了本地的 Pod 以及来自 CocoaPods 仓库的 Pod 之外, 你也可以将以下类型的库添加为依赖项:

- 来自自定义规格仓库的库.
- 来自 Git 仓库的远程库.
- 来自 archive 的库 (也可以使用任意 HTTP 地址).
- 静态库.
- 使用自定义 cinterop 参数的库.

更多详情请参见 在 Kotlin 项目中 添加 CocoaPods 依赖项 (添加 Pod 库依赖项). 示例程序请参见 Kotlin 使用 CocoaPods 示例 (https://github.com/Kotlin/kmm-with-cocoapods-sample).

与 Xcode 集成的更新

为了更好的与 Xcode 协同工作, Kotlin 要求 Podfile 的一些变化:

- 如果你的 Kotlin Pod 有任何 Git, HTTP, 或 specRepo Pod 的依赖项, 那么你还需要在 Podfile 中指定这些依赖项.
- 当你添加一个来自自定义 spec 的库, 那么还需要在你的 Podfile 开头, 指定 spec 的 位置 (https://guides.cocoapods.org/syntax/podfile.html#source).

在 IDEA 中, 集成错误现在有了详细的描述信息. 因此当你遇到 Podfile 相关的问题, 可以立即知道如何修复.

更多详情请参见 创建 Kotlin pod (将 Kotlin Gradle 项目用作 CocoaPods 依赖项).

支持 Xcode 12 库

对随 Xcode 12 一起发布的新库, 我们添加了支持. 现在你可以在 Kotlin 代码中使用这些库.

Kotlin Multiplatform

跨平台库发布的结构更新

从 Kotlin 1.4.20 开始, 不再有单独的元数据发布. 元数据 artifact 现在包含在 *root* 发布之内, 它代表整个库, 并且在添加为共通源代码集的依赖项时, 会自动解析为适当的平台相关 artifact.

更多详情请参见 发布跨平台库 (发布跨平台的库).

与以前版本的兼容性

这样的结构变化, 破坏了使用 层级项目结构 ("在类似的平台上共用代码" in "在不同的平台之间共用代码") 的项目之间的兼容性. 如果一个跨平台项目和它依赖的一个库都使用了层级项目结构, 那么你需要将它们同步更新到 Kotlin 1.4.20 或更高版本. 使用 Kotlin 1.4.20 发布的库, 不能供以前版本发布的项目使用.

不使用层级项目结构的项目和库仍然保持兼容.

标准库

Kotlin 1.4.20 的标准库提供了一些用来处理文件的新的扩展, 并改进了性能.

- 对 java.nio.file.Path 的扩展
- String.replace 函数性能改进

用于 java.nio.file.Path 的扩展

> 用于 `java.nio.file.Path` 的扩展是 实验性功能 (Kotlin 各部分组件的稳定性). 这个功能随时可能抛弃或改变. 使用这个功能需要使用者同意(Opt-in)(详情请参见下文). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

标准库提供用于 java.nio.file.Path 的实验性的扩展. 现在可以通过符合 Kotlin 习惯的方式来使用现代的 JVM 文件 API, 与 kotlin.io 包中的 java.io.File 扩展类似.

```
// 使用除 (/) 操作符构造路径
val baseDir = Path("/base")
val subDir = baseDir / "subdirectory"

// 列出目录中的文件
val kotlinFiles: List<Path> =
Path("/home/user").listDirectoryEntries("*.kt")
```

这些扩展在 kotlin-stdlib-jdk7 模块的 kotlin.io.path 包内. 要使用这些扩展, 需要对实验性的注解 @ExperimentalPathApi 标注 使用者同意(opt-in) (明确要求使用者同意的功能(Opt-in Requirement)).

String.replace 函数性能改进

String.replace() 的新实现提高了这个函数的执行速度. 大小写相关的版本使用一种基于 indexOf 的手动替换循环, 大小写无关的版本使用正规表达式匹配.

Kotlin Android Extensions

在 1.4.20 中, Kotlin Android Extensions plugin 已废弃, Parcelable 实现代码生成器移动到了一个单独的 plugin 中.

- 废弃合成视图(synthetic view)
- Parcelable 实现代码生成器的新 plugin

废弃合成视图(synthetic view)

以前 Kotlin Android Extensions plugin 中曾有过 *合成视图(synthetic view)*, 用来简化与 UI 元素的交互, 并减少样板代码. 现在 Google 提供了一种原生机制来完成相同的工作 - Android Jetpack 的 视图绑定 (https://developer.android.com/topic/libraries/view-binding), 因此我们废弃了合成视图, 请改用这个新功能.

我们从 kotlin-android-extensions 中抽取了 Parcelable 实现代码生成器, 并对这个 plugin 的其他部分(也就是合成视图)开始了废弃周期. 这些功能现在还能继续工作, 但会有废弃警告信息. 将来, 你需要为你的项目使用其他解决方案. 这里有一篇 指南 (https://goo.gle/kotlin-android-

extensions-deprecation), 可以帮助你将 Android 项目从合成视图(synthetic view) 迁移到视图绑定(view binding).

Parcelable 实现代码生成器的新 plugin

`Parcelable` 实现代码生成器移动到了新的 `kotlin-parcelize` plugin 中. 请使用这个 plugin 而不是原来的 `kotlin-android-extensions`.

> 在同一个模块中, `kotlin-parcelize` 和 `kotlin-android-extensions` 不能同时使用.

`@Parcelize` 注解移动到了 `kotlinx.parcelize` 包中.

更多详情请参见 Android 文档 (https://developer.android.com/kotlin/parcelize) 中的 `Parcelable` 实现代码生成器.

Kotlin 1.4.0 版中的新功能

最终更新: 2024/09/10

发布日期: 2020/08/17 ("各发布版详情" in "Kotlin 的发布版本")

在 Kotlin 1.4.0 中, 我们对所有组件发布了许多改进, 专注于改善质量和性能 (https://blog.jetbrains.com/kotlin/2020/08/kotlin-1-4-released-with-a-focus-on-quality-and-performance/). 下文详细介绍 Kotlin 1.4.0 中最重要的变化.

语言方面的新功能和改进

Kotlin 1.4.0 包含很多语言方面的新功能和改进. 包括:

- 对 Kotlin 接口的 SAM 转换
- 供库作者使用的明确 API 模式
- 混合使用命名参数和按位置传递的参数
- 尾随逗号(trailing comma)
- 可调用引用的改进
- 在循环内部的 `when` 表达式中使用 `break` 和 `continue`

对 Kotlin 接口的 SAM 转换

在 Kotlin 1.4.0 之前, 只有 在 Kotlin 中使用 Java 方法和 Java 接口 ("SAM 转换" in "在 Kotlin 中调用 Java 代码") 时, 才可以进行 SAM (Single Abstract Method) 转换. 现在, 也可以对 Kotlin 接口进行 SAM 转换了. 要进行这种转换, 需要使用 `fun` 修饰符, 将 Kotlin 接口明确标记为函数接口.

如果函数接收的参数是一个仅有单个抽象方法的接口, 而你传递一个 Lambda 表达式作为实际参数, 这时就适用 SAM 转换. 这种情况下, 编译器将 Lambda 表达式自动转换为实现这个抽象成员函数的类的一个实例.

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }
```

```
fun main() {
    println("Is 7 even? - ${isEven.accept(7)}")
}
```

更多详情请参见 Kotlin 函数式接口与 SAM 转换 (函数式 (SAM) 接口).

供库作者使用的明确 API 模式

Kotlin 编译器对库作者提供了 *明确 API 模式(explicit API mode)*. 在这种模式下, 编译器会进行额外的检查, 让库的 API 更加清晰更加一致. 对导出到库的公开 API 的声明, 它会增加以下要求:

- 如果默认可见度会导出声明到公开 API, 那么对于声明必须指定可见度修饰符. 这个限制可以保证声明不会无意中导出到公开 API.
- 对于导出到公开 API 的属性和函数, 必须明确指定类型. 这个限制可以保证 API 使用者注意到他们所使用的API 成员的类型.

根据你的配置不同, 这些明确的 API 检查可以产生错误 (*strict* 模式) 或产生警告 (*warning* 模式). 考虑到代码可读性, 以及通常的习惯, 这些检查不包含以下类型的声明:

- 主构造器
- 数据类的属性
- 属性的 get 方法和 set 方法
- `override` 方法

明确 API 模式只会分析一个模块的产品源代码(production source).

要在明确 API 模式下编译你的模块, 请在你的 Gradle 构建脚本中添加以下代码:

Kotlin

```
kotlin {
    // 用于 strict 模式
    explicitApi()
    // 或
    explicitApi = ExplicitApiMode.Strict
```

```
    // 用于 warning 模式
    explicitApiWarning()
    // 或
    explicitApi = ExplicitApiMode.Warning
}
```

Groovy

```
kotlin {
    // 用于 strict 模式
    explicitApi()
    // 或
    explicitApi = 'strict'

    // 用于 warning 模式
    explicitApiWarning()
    // 或
    explicitApi = 'warning'
}
```

使用命令行编译器时, 要切换到明确 API 模式, 可以添加编译器选项 `-Xexplicit-api`, 指定值为 `strict` 或 `warning`.

```
-Xexplicit-api={strict|warning}
```

关于明确 API 模式的更多细节, 请参见 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/explicit-api-mode.md).

混合使用命名参数和按位置传递的参数

在 Kotlin 1.3 中, 如果使用 命名参数 ("命名参数" in "函数") 来调用一个函数, 那么必须将所有的无名称参数 (按位置传递的参数) 放在第一个命名参数之前. 比如, 可以调用 `f(1, y = 2)`, 但不能调用 `f(x = 1, 2)`.

如果所有的参数都在正确的位置, 但你希望对中间的一个参数指定名称, 这时这种限制就很麻烦. 如果能够标识清楚一个 boolean 值或 `null` 值到底属于哪个参数, 对于我们的代码是非常有帮助的.

在 Kotlin 1.4 中, 不再存在这样的限制可 – 现在, 在一组按位置传递的参数中, 你可以对一个位于中间的参数指定名称. 而且, 可以按任意方式混合使用按位置传递的参数和命名参数, 只要它们保持正

确的顺序就可以了.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Char = ' '
) {
    // ...
}

// 使用一个位于中间的命名参数来调用函数
reformat("This is a String!", uppercaseFirstLetter = false , '-')
```

尾随逗号(trailing comma)

在 Kotlin 1.4 中, 可以在各种列举中添加尾随逗号, 比如: 实际参数, 参数声明, `when` 语句的分支条件, 以及解构声明的元素. 通过使用尾随逗号, 可以添加新元素, 以及修改元素顺序, 而不必添加或删除逗号.

如果你对参数或值使用多行语法, 这个功能很有帮助. 添加尾随逗号之后, 对参数或值可以很容易的交换各行的位置.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Character = ' ', // 尾随逗号
) {
    // ...
}
```

```
val colors = listOf(
    "red",
    "green",
    "blue", // 尾随逗号
)
```

可调用引用的改进

Kotlin 1.4 对于可调用引用的使用, 支持更多情况:

- 对带默认参数值的函数的引用
- 在返回值为 Unit 的函数内使用函数引用
- 根据函数参数个数适用的引用
- 对可调用引用的挂起转换

对带默认参数值的函数的引用

现在, 你可以使用带默认参数值的函数的可调用引用. 如果对函数 foo 的可调用引用没有参数, 那么会使用默认值 0.

```
fun foo(i: Int = 0): String = "$i!"

fun apply(func: () -> String): String = func()

fun main() {
    println(apply(::foo))
}
```

在以前, 必须对函数 apply 编写额外的重载(overload)版, 才能使用默认参数值.

```
// 一些新的重载版本
fun applyInt(func: (Int) -> String): String = func(0)
```

在返回值为 Unit 的函数内使用函数引用

在 Kotlin 1.4 中, 在返回 Unit 的函数内, 可以使用返回任何类型的函数的可调用引用. 在 Kotlin 1.4 之前, 这种情况下只能使用 Lambda 参数. 现在既可以使用 Lambda 参数也可以使用可调用引用.

```
fun foo(f: () -> Unit) { }
fun returnsInt(): Int = 42

fun main() {
    foo { returnsInt() } // 在 1.4 版以前, 这是唯一的方法
    foo(::returnsInt) // 从 1.4 版开始, 也可以使用这种方法
}
```

根据函数参数个数适用的引用

当传递可变数量的参数(vararg)时, 现在可以适用函数的可调用引用. 在传递的参数列表的最后, 可以传递任意数量的相同类型参数.

```
fun foo(x: Int, vararg y: String) {}

fun use0(f: (Int) -> Unit) {}
fun use1(f: (Int, String) -> Unit) {}
fun use2(f: (Int, String, String) -> Unit) {}

fun test() {
    use0(::foo)
    use1(::foo)
    use2(::foo)
}
```

对可调用引用的挂起转换

除了对 Lambda 表达式的挂起转换之外, 从 1.4.0 版开始, Kotlin 现在还支持对可调用引用的挂起转换.

```
fun call() {}
fun takeSuspend(f: suspend () -> Unit) {}

fun test() {
    takeSuspend { call() } // 在 1.4 版以前可以这样
    takeSuspend(::call) // 在 Kotlin 1.4 版中, 也可以这样
}
```

在循环内部的 when 表达式中使用 break 和 continue

在 Kotlin 1.3 中, 在循环内部的 when 表达式之内, 不能使用没有位置标签的 break 和 continue. 原因是这些关键字被保留用于 when 表达式的可能的 跳过(fall-through)行为 (https://en.wikipedia.org/wiki/Switch_statement#Fallthrough).

因此, 如果想要在循环内部的 when 表达式中使用 break 和 continue, 你必须对它们 添加标签 ("Break 和 Continue 的位置标签" in "返回与跳转: break 与 continue"), 所以代码会变得比较笨重.

```
fun test(xs: List<Int>) {
    LOOP@for (x in xs) {
        when (x) {
```

```
            2 -> continue@LOOP
            17 -> break@LOOP
            else -> println(x)
        }
    }
}
```

在 Kotlin 1.4 中, 在循环内部的 `when` 表达式中可以使用不带标签的 `break` 和 `continue`. 这两条语句的会象我们预期的那样, 结束最内层的循环, 或者跳转到循环的下一步.

```
fun test(xs: List<Int>) {
    for (x in xs) {
        when (x) {
            2 -> continue
            17 -> break
            else -> println(x)
        }
    }
}
```

`when` 之内的跳过(fall-through)行为, 我们留待未来的设计解决.

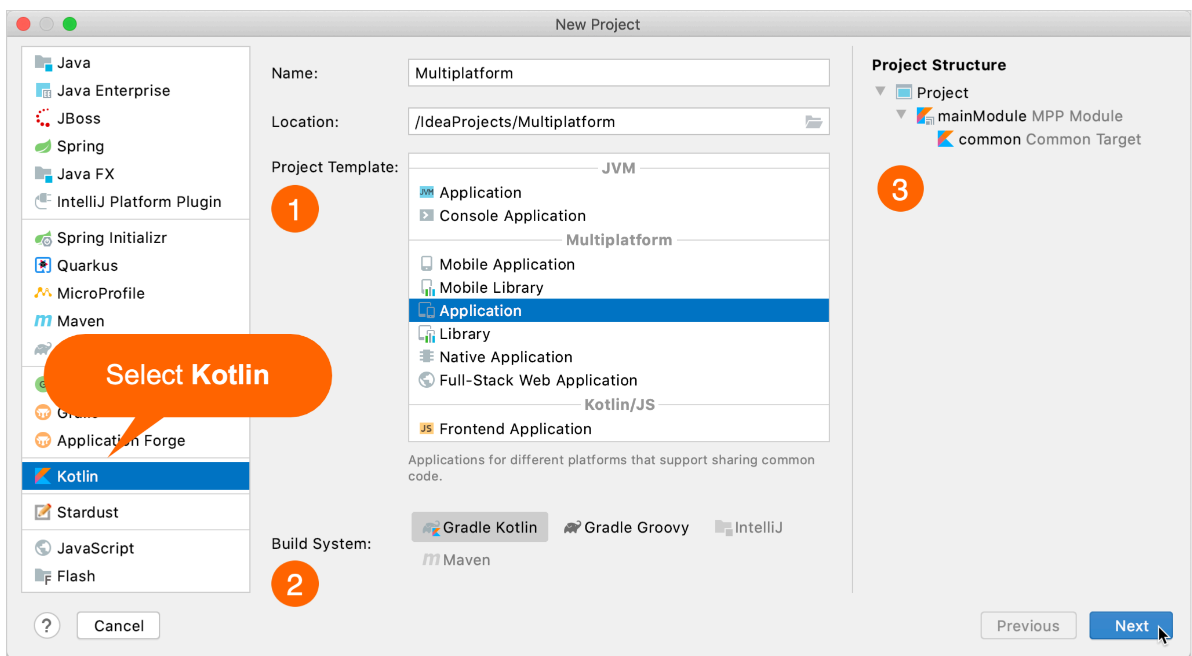
IDE 中的新工具

在 Kotlin 1.4 中, 可以在 IntelliJ IDEA 中使用新工具来简化 Kotlin 开发:

- 新的灵活的项目向导
- 协程调试器

新的灵活的项目向导

使用新的灵活的 Kotlin 项目向导, 可以非常简便的创建并配置不同类型的 Kotlin 项目, 包括 跨平台项目, 没有 UI 的帮助是很难配置的.



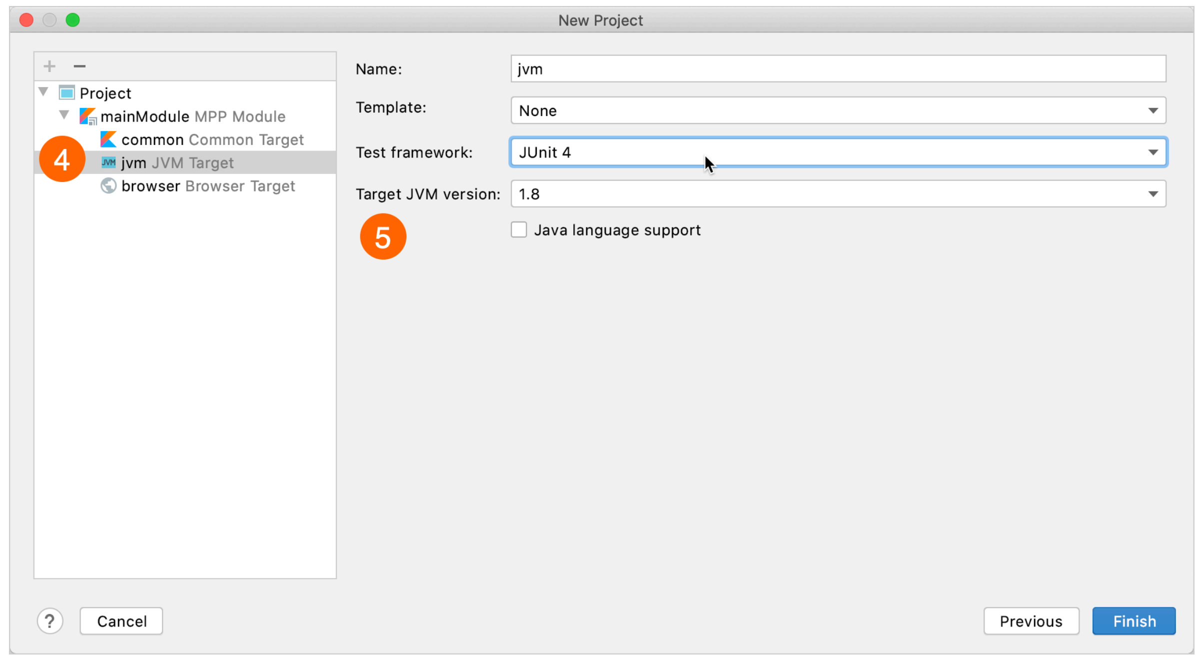
Kotlin 项目向导 – 跨平台项目

新的 Kotlin 项目向导既简单又灵活:

1. *选择项目模板*, 可以根据你的目的来不同的模板. 未来还会添加更多模板.
2. *选择构建系统* – Gradle (Kotlin 或 Groovy DSL), Maven, 或 IntelliJ IDEA. Kotlin 项目向导只会显示你选择的项目模板支持的构建系统.
3. *预览项目结构*, 可以直接在主画面上预览.

然后可以完成你的项目的创建, 或者也可以, 在下一个画面 *配置项目*:

4. *添加/删除这个项目模板支持的模块和编译目标.*
5. *配置模块和编译目标的设置*, 比如, 目标 JVM 版本, 目标模板, 以及测试框架.



Kotlin 项目向导 - 配置编译目标

将来, 我们还会添加更多配置选项和模板, 让 Kotlin 项目向导更加灵活.

你可以学习以下教程来试用新的 Kotlin 项目向导:

- 创建一个基于 Kotlin/JVM 的控制台应用程序 (Kotlin/JVM 入门)
- 创建一个针对 React 的 Kotlin/JS 应用程序 (教程 - 使用 React 和 Kotlin/JS 创建 Web 应用程序)
- 创建一个 Kotlin/Native 应用程序 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA)

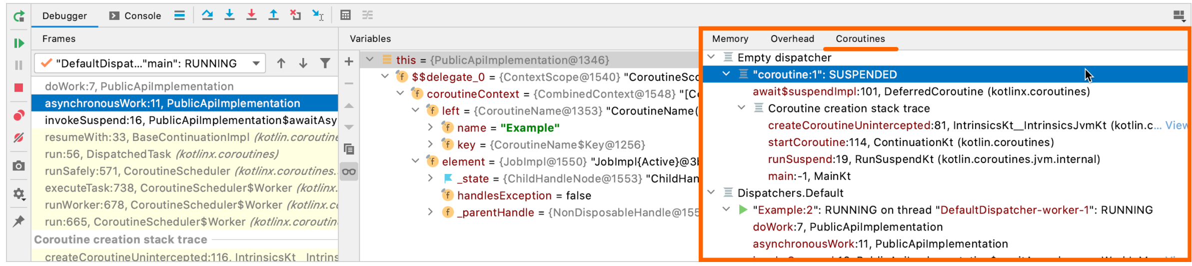
协程调试器

很多用户已经在异步程序开发中使用了 协程 (协程指南). 但在 Kotlin 1.4 以前, 协程的调试非常困难. 由于协程在不同的线程之间跳转, 因此很难理解某个协程正在做什么, 也很难查看它的上下文. 有些情况下, 在代码断点上单步运行根本无法工作. 因此, 你不得不依靠日志, 花费巨大的精力来调试使用协程的代码.

在 Kotlin 1.4 中, 有了 Kotlin plugin 的新功能, 调试协程现在变得更加方便了.

> 协程调试功能适用于 `kotlinx-coroutines-core` 的 1.3.8 或更高版本.

Debug Tool Window 现在包含新的 **Coroutines** 页. 在这个页中, 你可以查看当前正在运行的以及挂起的协程信息. 协程按照它运行时所属的派发器分组显示.

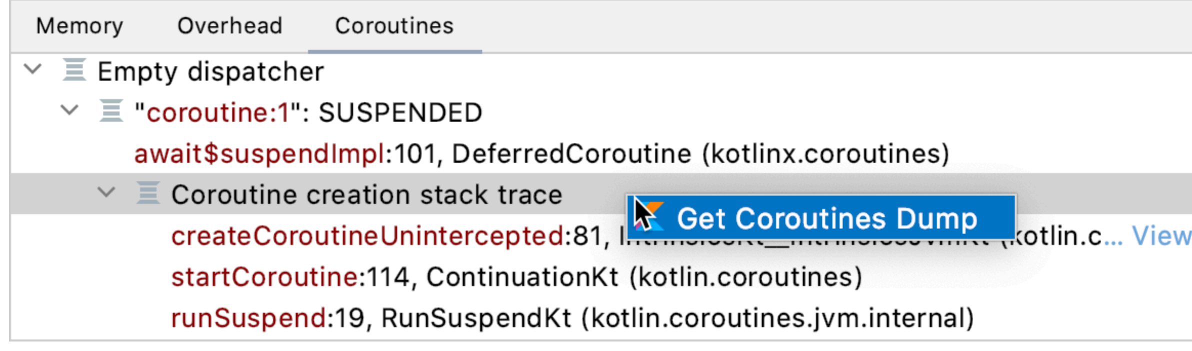


调试协程

现在你可以:

- 很容易的查看每个协程的状态.
- 对运行中的和挂起的协程, 查看局部变量和捕获变量的值.
- 查看完整的协程创建栈, 以及协程内的调用栈. 栈包括所有的栈层次(frame)及变量值, 甚至在标准调试时会丢失的那些变量值也可以查看.

如果你需要完整的报告, 包含每个协程的状态和它的栈, 可以在 **Coroutines** 页内点击鼠标右键, 然后点击 **Get Coroutines Dump**. 目前, 协程 dump 还很简单, 但在 Kotlin 的未来版本中, 我们会让它更加易读, 更能为你提供帮助.



协程 Dump

关于协程调试, 更多信息请参见 这篇 Blog (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-rc-debugging-coroutines/) 以及 IntelliJ IDEA 文档 (https://www.jetbrains.com/help/idea/debug-kotlin-coroutines.html).

新编译器

Kotlin 的新编译器将会非常快; 它还能统一所有支持的平台, 并为编译器扩展提供 API. 这是一个长期的项目, 在 Kotlin 1.4.0 中我们已经完成了一些部分:

- 新的更强大的类型推断算法, 默认情况下已启用.

- 新的 JVM 和 JS IR 后端. 开发达到稳定状态之后, 将会成为默认后端.

新的更强大的类型推断算法

Kotlin 1.4 使用一个新的, 更强大的类型推断算法. 在 Kotlin 1.3 中, 可以通过指定一个编译器选项来试用这个新算法, 现在它已经默认使用了. 你可以在 YouTrack (https://youtrack.jetbrains.com/issues/KT?q=Tag:%20fixed-in-new-inference%20) 找到新算法中修正的所有问题. 下面是一部分最值得注意的改进:

- 更多情况下类型能够自动推断

- 对 Lambda 内最后一条表达式的智能类型转换

- 对可调用引用的智能类型转换

- 对委托属性更好的类型推断

- 使用不同的参数对 Java 接口进行 SAM 转换

- 在 Kotlin 中使用 Java SAM 接口

更多情况下类型能够自动推断

对很多情况下旧算法无法推断类型, 因此要求你明确指定类型, 而新的推断算法能够正确推断类型. 比如, 下面示例程序中, Lambda 表达式的 `it` 参数类型能够正确的推断为 `String?`:

```
//sampleStart
val rulesMap: Map<String, (String?) -> Boolean> = mapOf(
    "weak" to { it != null },
    "medium" to { !it.isNullOrBlank() },
    "strong" to { it != null && "^[a-zA-Z0-
9]+$".toRegex().matches(it) }
)
//sampleEnd

fun main() {
    println(rulesMap.getValue("weak")("abc!"))
    println(rulesMap.getValue("strong")("abc"))
```

```
    println(rulesMap.getValue("strong")("abc!"))
}
```

在 Kotlin 1.3 中, 你需要引入一个明确的 Lambda 表达式参数, 或者把 `to` 替换为 `Pair` 构造函数, 并使用明确的泛型参数, 代码才能正确工作.

对 Lambda 内最后一条表达式的智能类型转换

在 Kotlin 1.3 中, Lambda 之内的最后一条表达式无法智能类型转换, 除非你明确指定类型. 因此, 在下面的示例程序中, Kotlin 1.3 将 `result` 变量的类型推断为 `String?`:

```
val result = run {
    var str = currentValue()
    if (str == null) {
        str = "test"
    }
    str // Kotlin 编译器知道这里的 str 不是 null
}
// 'result' 的类型在 Kotlin 1.3 中是 String?, 在 Kotlin 1.4 中则是
String
```

在 Kotlin 1.4 中, 由于新推断算法的帮助, Lambda 中最后一条表达式能够进行智能类型转换了, 并且, 这个新的更加准确的类型会被用来推断 Lambda 表达式的结果类型. 因此, `result` 变量的类型变成了 `String`.

在 Kotlin 1.3 中, 你经常需要添加明确的类型转换 (使用 `!!` 或 `as String` 之类的类型转换) 才能让这些代码正常工作, 现在, 这些类型转换代码不再需要了.

对可调用引用的智能类型转换

在 Kotlin 1.3 中, 你不能访问智能转换类型的成员引用. 在 Kotlin 1.4 中, 现在你可以了:

```
import kotlin.reflect.KFunction

sealed class Animal
class Cat : Animal() {
    fun meow() {
        println("meow")
    }
}
```

```
class Dog : Animal() {
    fun woof() {
        println("woof")
    }
}

//sampleStart
fun perform(animal: Animal) {
    val kFunction: KFunction<*> = when (animal) {
        is Cat -> animal::meow
        is Dog -> animal::woof
    }
    kFunction.call()
}
//sampleEnd

fun main() {
    perform(Cat())
}
```

animal 变量经过智能类型转换变为具体的类型 `Cat` 和 `Dog`, 在此之后, 你可以使用不同的成员引用 `animal::meow` 和 `animal::woof`. 在类型检查之后, 你可以访问对应的子类型的成员引用.

对委托属性更好的类型推断

在分析 `by` 关键字之后的委托表达式时, 不会考虑委托属性的类型. 比如, 下面的代码在以前的版本中无法编译, 但现在编译器对 `old` 和 `new` 参数的类型, 能够正确的推断为 `String?`:

```
import kotlin.properties.Delegates

fun main() {
    var prop: String? by Delegates.observable(null) { p, old, new ->
        println("$old → $new")
    }
    prop = "abc"
    prop = "xyz"
}
```

使用不同的参数对 Java 接口进行 SAM 转换

Kotlin 从一开始就支持对 Java 接口的 SAM 转换, 但有一种情况不支持, 因此在使用既有的 Java 库时造成一些麻烦. 如果你调用一个 Java 方法, 这个方法接受两个 SAM 接口作为参数, 两个参数必须都是 Lambda 表达式, 或者都是通常的对象. 这时你不能传递 Lambda 表达式给一个参数, 同时传递对象给另一个参数.

新算法解决了这个问题, 因此任何情况下你都可以象你很自然的预期的那样, 传递 Lambda 表达式, 而不需要传递 SAM 接口.

```
// FILE: A.java
public class A {
    public static void foo(Runnable r1, Runnable r2) {}
}
```

```
// FILE: test.kt
fun test(r1: Runnable) {
    A.foo(r1) {}  // 在 Kotlin 1.4 可以正常工作
}
```

在 Kotlin 中使用 Java SAM 接口

在 Kotlin 1.4 中, 你可以在 Kotlin 中使用 Java SAM 接口, 并对它们进行 SAM 转换.

```
import java.lang.Runnable

fun foo(r: Runnable) {}

fun test() {
    foo { } // OK
}
```

在 Kotlin 1.3 中, 你需要在 Java 代码中声明上面的函数 `foo`, 然后才能进行 SAM 转换.

统一的后端和扩展性

在 Kotlin 中, 我们有 3 种后端用来生成可执行代码: Kotlin/JVM, Kotlin/JS, 和 Kotlin/Native. Kotlin/JVM 和 Kotlin/JS 之间没有太多共用代码, 因为它们是分别独立开发的. Kotlin/Native 基于一种新的基础架构, 它对 Kotlin 代码使用一种中间表达形式(IR, intermediate representation).

我们现在将 Kotlin/JVM 和 Kotlin/JS 移植到同样的 IR. 因此, 所有这 3 种后端共用很多逻辑, 并且使用相同的输入输出管道. 所以对所有的平台, 我们对大多数功能特性, 优化, 以及 bug 修复, 只需要实

现一次. 这 2 种新的基于 IR 的后端都处于 Alpha (Kotlin 各部分组件的稳定性) 阶段.

这种共通的后端基础架构还使得我们可以开发跨平台的编译器扩展. 你可以在编译过程的输入输出管道中添加 plugin, 添加自定义的处理和转换, 你的扩展代码可以自动适用于所有的平台.

我们鼓励你使用我们的新 JVM IR 和 JS IR 后端, 它们还处于 Alpha 阶段, 希望你能向我们反馈意见.

Kotlin/JVM

Kotlin 1.4.0 包含很多针对 JVM 的改进, 比如:

- 新的 JVM IR 后端
- 在接口中生成默认方法的新模式
- 对 null 值检查统一异常类型
- 在 JVM 字节码中的类型注解

新的 JVM IR 后端

和 Kotlin/JS 一样, 我们正在将 Kotlin/JVM 移植到 统一的 IR 后端, 这个后端使得我们可以对大多数功能特性和 bug 修复在所有的平台只需要实现一次. 你也能够对这个后端创建跨平台的扩展, 可以在所有的平台上工作.

Kotlin 1.4.0 还没有提供公开的 API 对这些扩展, 但我们正在和我们的伙伴密切合作, 包括 Jetpack Compose (https://developer.android.com/jetpack/compose), 他们已经使用我们的新后端, 创建了他们的编译器 plugin.

我们鼓励你试用新的 Kotlin/JVM 后端, 它目前还处于 Alpha 阶段, 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提交问题和新特性请求. 你的帮助能够使我们更快的统一编译器的输入输出管道, 并将编译器扩展, 比如 Jetpack Compose, 带入 Kotlin 开发社区.

要启用新的 JVM IR 后端, 需要在你的 Gradle 构建脚本中指定一个额外的编译器选项:

```
kotlinOptions.useIR = true
```

> 如果 启用 Jetpack Compose (https://developer.android.com/jetpack/compose/setup?hl=en), 将会自动开始使用新的 JVM 后端, 而不必在 `kotlinOptions` 中指定编译器选项.

使用命令行编译器时, 需要添加编译器选项 `-Xuse-ir`.

> 只有在启用新后端时, 你才可以使用新的 JVM IR 后端编译的代码. 否则将会发生错误. 考虑到这一点, 我们不推荐库的作者在产品代码中切换到新的后端.

在接口中生成默认方法的新模式

编译 Kotlin 代码到 JVM 1.8 或以上版本时, 可以将 Kotlin 接口中的非抽象方法编译为 Java 的 `default` 方法. 实现这个目的的方式是, 使用 `@JvmDefault` 注解用来标记这些方法, 并使用 `-Xjvm-default` 编译器选项来启用对这个注解的处理.

在 1.4.0 中, 我们添加了一种新模式来生成默认方法: `-Xjvm-default=all` 将 *所有的* Kotlin 接口的非抽象方法 编译为 `default` 的 Java 方法. 对于那些编译为没有 `default` 方法的接口, 为了兼容使用它们的代码, 还添加了 `all-compatibility` 模式.

关于与 Java 交互时的默认方法, 更多详细信息请参见 Kotlin 与 Java 交互的相关文档 ("接口中的默认方法(Default Method)" in "在 Java 中调用 Kotlin 代码") 和 这篇 Blog (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-m3-generating-default-methods-in-interfaces/).

对 null 值检查统一异常类型

从 Kotlin 1.4.0 开始, 所有的运行时 null 值检查会抛出一个 `java.lang.NullPointerException` 异常, 而不是 `KotlinNullPointerException`, `IllegalStateException`, `IllegalArgumentException`, 和 `TypeCastException`. 这个规则适用于: `!!` 操作符, 在方法开始处的参数 null 值检查, 平台类型表达式的 null 值检查, 以及使用非 null 类型的 `as` 操作符. 但不适用于 `lateinit` null 值检查, 以及明确的库函数调用, 比如 `checkNotNull` 或 `requireNotNull`.

这个变化使得我们可以增加对 null 值检查的优化, 这些优化可以由 Kotlin 编译器执行, 或由各种字节码处理工具执行, 比如 Android R8 优化器 (https://developer.android.com/studio/build/shrink-code).

注意, 从开发者的角度来看, 并没有太大变化: Kotlin 代码还是会抛出异常, 错误消息和以前一样. 异常的类型变了, 但异常附带的信息是一样的.

在 JVM 字节码中的类型注解

Kotlin 现在可以在 JVM 字节码 (1.8+ 版本) 中生成类型注解, 因此在运行时可以通过 Java 反射得到这些信息. 要在字节码中产生类型注解, 需要以下步骤:

1. 确定你声明的注解指定了正确的注解目标 (Java 的 `ElementType.TYPE_USE`, 或 Kotlin 的 `AnnotationTarget.TYPE`), 以及正确的 retention (`AnnotationRetention.RUNTIME`).

2. 将注解类声明编译为版本 1.8+ 的 JVM 字节码. 你可以通过编译器选项 `-jvm-target=1.8` 来指定 JVM 字节码.

3. 将使用注解的代码编译为版本 1.8+ 的 JVM 字节码(`-jvm-target=1.8`), 并添加 `-Xemit-jvm-type-annotations` 编译器选项.

注意, 标准库中的类型注解现在不会生成到字节码中, 因为标准库编译使用的字节码版本是 1.6.

到目前位置, 只支持一些基本的场景:

- 方法参数, 方法返回类型, 以及属性类型上的类型注解;
- 类型参数的不可变投射(Invariant Projection), 比如 `Smth<@Ann Foo>`, `Array<@Ann Foo>`.

在下面的示例程序中, `String` 类型上的 `@Foo` 注解可以生成到字节码中, 然后可以供库代码使用:

```
@Target(AnnotationTarget.TYPE)
annotation class Foo

class A {
    fun foo(): @Foo String = "OK"
}
```

Kotlin/JS

对 JS 平台, Kotlin 1.4.0 提供了以下改进:

- 新的 Gradle DSL
- 新的 JS IR 后端

新的 Gradle DSL

`kotlin.js` Gradle plugin 带有调整过的 Gradle DSL, 它提供一些新的配置选项, 而且更加接近 `kotlin-multiplatform` plugin 使用的 DSL. 影响最大的变化包括:

- 通过 `binaries.executable()` 明确的创建可执行文件. 关于可执行的 Kotlin/JS 及其环境, 更多详情请阅读 Kotlin/JS 的运行及其环境 ("执行环境" in "创建 Kotlin/JS 工程(Project)").

- 在 Gradle 配置内, 通过 `cssSupport` 来配置 webpack 的 CSS 和样式装载器. 关于如何使用这些功能, 更多详情请阅读 使用 CSS 与样式装载器(Style Loader) ("CSS" in "创建 Kotlin/JS 工程

(Project)").

- 对 npm 依赖项管理的改进, 需要指定版本号, 或者 semver (https://docs.npmjs.com/misc/semver#versions) 版本范围, 以及使用 devNpm, optionalNpm 和 peerNpm 支持 *development*, *peer*, 和 *optional* npm 依赖项. 关于 npm 包的依赖项管理, 更多详情请阅读 直接使用 Gradle 管理 npm 的包依赖项目 ("npm 依赖项目" in "创建 Kotlin/JS 工程(Project)").

- 对 Kotlin 外部声明生成器 Dukat (https://github.com/Kotlin/dukat) 提供了更强的集成. 外部声明现在可以在构建时期生成, 也可以通过 Gradle 任务手动生成.

新的 JS IR 后端

用于 Kotlin/JS 的 IR 后端 (使用 IR 编译器), 稳定性现在处于 Alpha (Kotlin 各部分组件的稳定性) 阶段, 它针对 Kotlin/JS 编译目标提供了一些新的功能, 主要包括, 通过死代码消除来改善生成代码的大小, 以及改进与 JavaScript 和 TypeScript 的交互, 以及其他功能.

要启用 Kotlin/JS IR 后端, 请在你的 gradle.properties 文件中设置 kotlin.js.compiler=ir, 或者在你的 Gradle build 脚本中, 向 js 函数传递 IR 编译器类型:

```
kotlin {
    js(IR) { // 或者使用: LEGACY, BOTH
        // ...
    }
    binaries.executable()
}
```

关于如何配置新的编译器后端, 更多详情请阅读 Kotlin/JS IR 编译器文档 (使用 IR 编译器).

使用新的 @JsExport ("@JsExport 注解" in "在 JavaScript 中使用 Kotlin 代码") 注解, 以及 **从 Kotlin 代码 生成 TypeScript 定义 ("预览: 生成 TypeScript 声明文件 (d.ts)" in "使用 IR 编译器")** 的能力, Kotlin/JS IR 编译器后端改进了与 JavaScript & TypeScript 的交互能力. 也使得 Kotlin/JS 代码更容易与既有的工具集成, 来创建 **混合应用程序**, 并在跨平台项目利用代码共用功能.

关于 Kotlin/JS IR 编译器后端的详细功能特性, 请阅读这篇文档 (使用 IR 编译器).

Kotlin/Native

在 1.4.0 中, Kotlin/Native 有了大量的新功能和改进, 包括:

- 在 Swift 和 Objective-C 中支持挂起函数

- 默认支持 Objective-C 泛型
- 在与 Objective-C/Swift 交互中的异常处理
- 默认为 Apple 平台生成发行版的 .dSYM 文件
- 性能改进
- 简化 CocoaPods 依赖项管理

在 Swift 和 Objective-C 中支持 Kotlin 的挂起函数

在 1.4.0 中, 我们对 Swift 和 Objective-C 的挂起函数添加了基本的支持. 现在, 如果你将 Kotlin 模块编译为 Apple 框架, 挂起函数可以作为带有回调的函数来使用(用 Swift/Objective-C 术语来说就是 `completionHandler`). 如果在生成的框架头文件中中存在这样的函数, 可以从你的 Swift 或 Objective-C 代码中调用这些函数, 还可以覆盖它们.

比如, 如果你编写了这样的 Kotlin 函数:

```
suspend fun queryData(id: Int): String = ...
```

...那么可以从 Swift 代码中调用它, 如下:

```
queryData(id: 17) { result, error in
    if let e = error {
        print("ERROR: \(e)")
    } else {
        print(result!)
    }
}
```

关于在 Swift 和 Objective-C 中使用挂起函数, 请阅读这篇文档 (与 Swift/Objective-C 代码交互).

默认支持 Objective-C 泛型

以前的 Kotlin 版本 提供了实验性功能, 在与 Objective-C 交互时支持泛型. 从 1.4.0 开始, Kotlin/Native 默认情况下会从 Kotlin 代码生成带有泛型的 Apple 框架. 在一些情况下, 这个结果可能会导致既有的, 调用 Kotlin 框架的 Objective-C 或 Swift 代码无法工作. 如果要让生成的框架头文件不带泛型, 请添加编译器选项 `-Xno-objc-generics`.

```
kotlin {

targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNative
Target> {
        binaries.all {
            freeCompilerArgs += "-Xno-objc-generics"
        }
    }
}
```

请注意, 关于与 Objective-C 交互的文档 ("泛型" in "与 Swift/Objective-C 代码交互") 中列出的所有功能细节和限制, 仍然有效.

在与 Objective-C/Swift 交互中的异常处理

在 1.4.0 中, 我们稍微修改了从 Kotlin 生成的 Swift API, 使其符合异常翻译的方式. Kotlin 和 Swift 的错误处理存在根本的差别. 所有的 Kotlin 异常都是不受控的(unchecked), 而 Swift 只存在受控的(checked)错误. 因此, 要让 Swift 代码能够知道存在哪些预期的异常, Kotlin 函数应该标记 `@Throws` 注解, 指明这个函数可能发生的异常类型列表.

在编译为 Swift 或 Objective-C 框架时, 标记了或继承了 `@Throws` 注解的函数会被表达为 Objective-C 中的产生 `NSError*` 的方法, 以及 Swift 中的 `throws` 方法.

在以前的版本中, 除 `RuntimeException` 和 `Error` 之外的任何异常都会作为 `NSError` 来传递. 现在这个变更为: 只对`@Throws` 注解的参数中指定的异常类(或它们的子类)的实例才会抛出 `NSError`. 到达 Swift/Objective-C 的其他 Kotlin 异常将被认为是未处理的错误, 并导致程序终止.

默认为 Apple 平台生成发行版的 .dSYM 文件

从 1.4.0 开始, 在 Darwin 平台下, Kotlin/Native 编译器对发行版的二进制文件默认会产生 调试符号文件(debug symbol file) (https://developer.apple.com/documentation/xcode/building_your_app_to_include_debugging_information) (`.dSYM` 文件). 这个功能可以使用编译器选项 `-Xadd-light-debug=disable` 关闭. 在其他平台下, 这个功能默认是关闭的. 要在 Gradle 中设置这个选项, 请使用:

```
kotlin {

targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNative
Target> {
        binaries.all {
```

```
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

关于应用程序崩溃报告的符号化, 请阅读这篇文档 (符号化(Symbolicate) iOS 崩溃报告(Crash Report)).

性能改进

Kotlin/Native 完成了很多性能改进, 提高了开发速度, 也提高了执行速度. 下面是一些例子:

- 为了改善对象分配的速度, 我们现在提供 mimalloc (https://github.com/microsoft/mimalloc) 内存分配器, 作为系统默认分配器的一个替代选项. 在一些新能评测中, mimalloc 工作速度要快 2 倍. 目前, 在 Kotlin/Native 中使用 mimalloc 还是实验性功能; 你可以使用编译器选项 `-Xallocator=mimalloc` 来切换到 mimalloc.
- 我们重写了 C 交互库的构建工具. 使用新的工具, Kotlin/Native 产生交互库的速度达到以前的 4 倍, 而且库文件大小只有以前的 25% 到 30%.
- 通过 GC 中的优化改善了运行时的整体性能. 在在使用大量长生存齐对象的项目中, 这个性能改进尤其明显. 由于去掉了多余的对象装箱(boxing)处理, `HashMap` 和 `HashSet` 现在工作更加快速了.
- 在 1.3.70 中我们引入了 2 个改善 Kotlin/Native 编译性能的新功能: 缓存项目依赖项, 以及从 Gradle daemon 中运行编译器 (https://blog.jetbrains.com/kotlin/2020/03/kotlin-1-3-70-released/#kotlin-native). 在这之后, 我们修复了很多问题, 并改善了这些功能的整体稳定性.

简化 CocoaPods 依赖项管理

在以前的版本中, 只要你的项目集成了依赖项管理器 CocoaPods, 你的项目与 iOS, macOS, watchOS, 或 tvOS 相关的部分就只能在 Xcode 中构建, 这部分将与你的跨平台项目的其它部分分离. 其他部分可以在 IntelliJ IDEA 中构建.

而且, 每次你添加一个保存在 CocoaPods (Pod 库) 中的 Objective-C 库的依赖项, 你都必须从 IntelliJ IDEA 切换到 Xcode, 调用 `pod install`, 然后在 Xcode 中执行构建.

现在, 你可以直接在 IntelliJ IDEA 中管理 Pod 依赖项, 同时又能在编码时享受它提供的好处, 比如代码高亮度和自动完成. 你还可以使用 Gradle 构建整个 Kotlin 项目, 不再需要切换到 Xcode. 这就意味着, 只有在需要编写 Swift/Objective-C 代码时, 或需要在模拟器或设备上运行你的应用程序时, 才需要切换到 Xcode.

现在你还可以使用存储在本地的 Pod 库.

根据你的需求不同, 可以添加以下依赖项:

- Kotlin 项目 与 存储在远程 CocoaPods 仓库的或存储在本地机器的 Pod 库之间的依赖.
- Kotlin Pod (作为 CocoaPods 依赖项使用的 Kotlin 项目) 与 带有一个或多个编译目标的 Xcode 项目之间的依赖.

完成初始配置时, 以及添加新依赖项到 `cocoapods` 时, 只需要在 IntelliJ IDEA 中重新导入项目. 新依赖项会添加自动. 不需要额外的操作步骤.

关于如何添加依赖项, 请阅读这篇文档 (添加 Pod 库依赖项).

Kotlin Multiplatform

> 跨平台项目功能现在处于 Alpha (Kotlin 各部分组件的稳定性) 阶段. 在未来的 Kotlin 版本中, 这个功能的兼容性可能会改变, 需要手工迁移. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin Multiplatform (Kotlin Multiplatform) 可以减少对 不同的平台 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 编写和维护相同代码的时间, 又能同时保持原生程序开发的灵活性便利. 我们一直在努力开发各种跨平台的新功能特性和改进:

- 使用层级项目结构在多个编译目标中共用代码
- 在层级结构中使用原生库
- kotlinx 依赖项只需要指定一次

> 跨平台项目需要 Gradle 6.0 或更高版本.

使用层级项目结构在多个编译目标中共用代码

使用新的层级项目结构, 在一个 跨平台项目 (Kotlin Multiplatform 项目结构的基础知识) 中, 你可以在 多个平台 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 间共用代码.

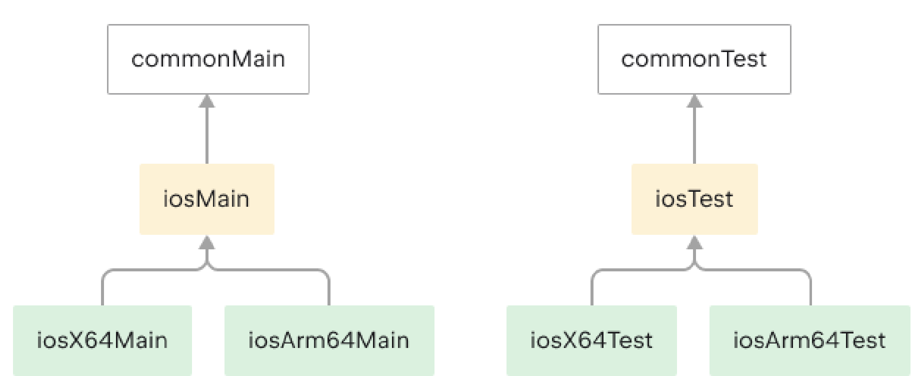
在以前的版本中, 添加到跨平台项目的代码, 可以放在平台相关的源代码集中, 只限于一个编译目标使用, 不能被其他任何平台重用, 也可以放在共通源代码集中, 比如 `commonMain` 或

`commonTest`, 被项目中的所有平台共用. 在共通源代码集中, 你只能通过使用 `expect` 声明(需要对应的平台相关的 `actual` 实现) (预期声明与实际声明) 来调用平台相关的 API.

通过这种机制很容易实现 在所有的平台上共用代码 ("在所有平台上共用代码" in "在不同的平台之间共用代码"), 但不容易 只在一部分编译目标中共用代码 ("在类似的平台上共用代码" in "在不同的平台之间共用代码"), 尤其是对于那些类似的编译目标, 本来可能重用很多共通逻辑和第三方 API.

比如, 在一个针对 iOS 平台的典型的跨平台项目中, 有 2 个 iOS 相关的编译目标: 一个针对 iOS ARM64 设备, 另一个针对 x64 模拟器. 它们的平台相关源代码集是分离的, 但实际上, 对真实设备和模拟器极少需要不同的代码, 并且它们的依赖项也是非常类似的. 因此对这 2 个编译目标, iOS 相关的代码是可以共用的.

显然, 在这样的设置中, 我们需要有 *对 2 个 iOS 编译目标的共用的源代码集*, 其中包含 Kotlin/Native 代码, 并且仍然能够直接调用那些对于 iOS 设备和模拟器共通的 API.



对于 iOS 编译目标的代码共用

现在你可以通过 层级项目结构 ("在类似的平台上共用代码" in "在不同的平台之间共用代码") 来实现这样的代码共用, 它能够通过使用源代码集合的编译目标, 来推断和适用各个源代码集中可用的 API 和语言功能特性.

对于共通的编译目标组合, 你可以使用编译目标的简写(shortcut)来创建层级结构. 比如, 可以通过 `ios()` 简写, 创建上面例子中的 2 个 iOS 编译目标以及共用的源代码集:

```
kotlin {
    ios() // 这里会创建 iOS 设备和模拟器的编译目标; iosMain 和 iosTest 源代码集
}
```

关于编译目标的其他组合, 请使用 `dependsOn` 关系连接源代码集, 来 手动创建层级结构 ("手动配置" in "层级项目结构").

Kotlin

```
kotlin{
    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain)
        }
        val linuxX64Main by getting {
            dependsOn(desktopMain)
        }
        val mingwX64Main by getting {
            dependsOn(desktopMain)
        }
        val macosX64Main by getting {
            dependsOn(desktopMain)
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        desktopMain {
            dependsOn(commonMain)
        }
        linuxX64Main {
            dependsOn(desktopMain)
        }
        mingwX64Main {
            dependsOn(desktopMain)
        }
        macosX64Main {
            dependsOn(desktopMain)
        }
```

```
    }
}
```

有了层级项目结构的帮助, 库也可以对一部分编译目标提供共通的 API. 更多详情请参见 在库中共用代码 ("在多个库之间共用代码" in "在不同的平台之间共用代码").

在层级结构中使用原生库

在几个原生编译目标间共用的源代码集中, 可以使用平台依赖的库, 比如 Foundation, UIKit, 和 POSIX. 这个功能可以帮助你共用更多的原生代码, 不受平台相关依赖项的限制.

不需要额外的步骤 – 所有事情都会自动完成. IntelliJ IDEA 会帮助你发现可以在共用代码中使用的共通声明.

更多详情请参见 使用平台依赖的库 ("连接平台相关的库" in "在不同的平台之间共用代码").

依赖项只需要指定一次

从现在开始, 在共用的和平台相关的源代码集中, 不再需要对同一个库的不同的变体指定依赖项, 只需要在共用的源代码集中指定依赖项一次.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
```

```
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3'
            }
        }
    }
}
```

不要使用 kotlinx 库的带有平台后缀的 artifact 名, 比如 `-common`, `-native`, 或其他类似名称, 因为这些名称已经不再支持了. 而应该使用库的基本 artifact 名, 在上面的示例程序中是 `kotlinx-coroutines-core`.

但是, 这个变化不适用于以下情况:

- `stdlib` 库 – 从 Kotlin 1.4.0 开始, stdlib 依赖项会添加自动.
- `kotlin.test` 库 – 你还是应该使用 `test-common` 和 `test-annotations-common`. 这些依赖项会在后文中讨论.

如果你只对一个特定的平台需要某个依赖项, 你仍然可以使用标准库和 kotlinx 库的平台相关的, 带后缀 `-jvm` 或 `-js` 的变体, 比如 `kotlinx-coroutines-core-jvm`.

关于配置依赖项, 请阅读这篇文档 ("配置依赖项" in "配置 Gradle 项目").

Gradle 项目的改进

除了 Kotlin Multiplatform, Kotlin/JVM, Kotlin/Native, 和 Kotlin/JS 的 Gradle 项目功能特性和改进之外, 对于所有的 Kotlin Gradle 项目, 还有以下变化:

- 标准库的依赖项现在会默认添加
- Kotlin 项目需要新版本的 Gradle
- IDE 中对 Kotlin Gradle DSL 支持的改进

标准库的依赖项现在会默认添加

在任何 Kotlin Gradle 项目中, 不再需要声明对 `stdlib` 库的依赖项, 包括跨平台项目. 这个依赖项会默认添加.

自动添加的标准库版本与 Kotlin Gradle plugin 相同, 因为它们使用相同的版本号发布.

对于平台相关的源代码集, 会使用标准库在对应的平台上的变体, 对其他源代码集会添加共通标准库. Kotlin Gradle plugin 会根据你的 Gradle build 脚本中的 `kotlinOptions.jvmTarget` 编译器选项 (Kotlin Gradle plugin 中的编译器选项), 来选择适当的 JVM 标准库.

关于如何改变默认行为, 请阅读这篇文档 ("对标准库的依赖项" in "配置 Gradle 项目").

Kotlin 项目需要的 Gradle 最低版本

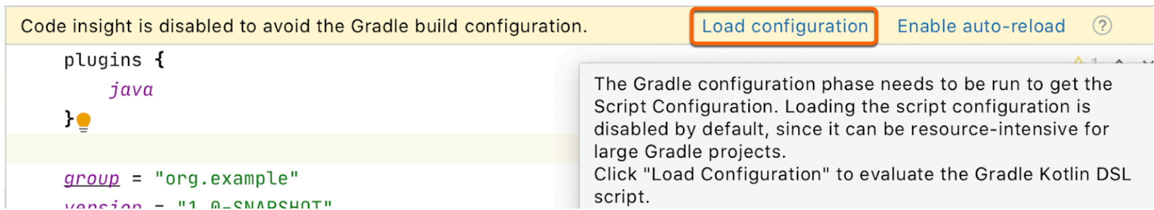
要在你的 Kotlin 项目中使用新的功能特性, 需要将 Gradle 更新到 最新版本 (https://gradle.org/releases/). 跨平台项目需要 Gradle 6.0 或更高版本, 其他 Kotlin 项目可以使用 Gradle 5.4 或更高版本.

IDE 中对 *.gradle.kts 支持的改进

在 1.4.0 中, 我们继续改进了 IDE 对 Gradle Kotlin DSL 脚本(`*.gradle.kts` 文件) 的支持. 新版本的带来的新功能如下:

- *脚本配置的明确加载*. 以前的版本中, 你对构建脚本的变更会在后台自动加载. 为了改善性能, 在 1.4.0 中我们关闭了构建脚本配置的自动加载. 现在只有在你明确适用时, IDE 才会加载这些变更.

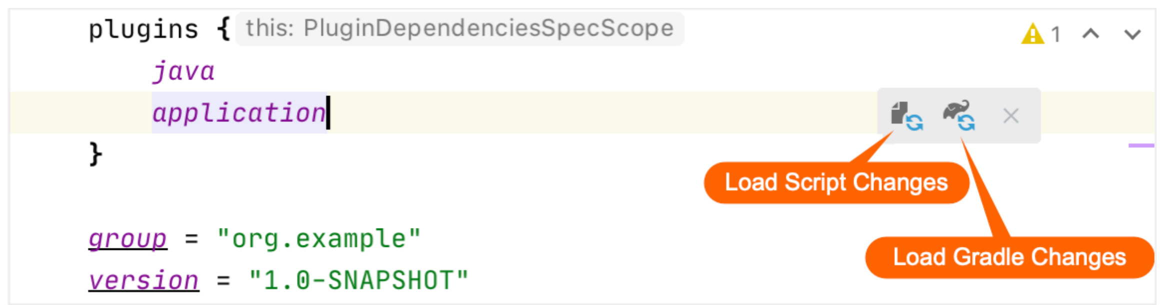
  Gradle 6.0 以前的版本中, 你需要在编辑器中点击 **Load Configuration** 来手动加载脚本配置.

  *.gradle.kts – 加载配置

  在 Gradle 6.0 及以上的版本, 你可以点击 **Load Gradle Changes** 来明确适用变更, 或者也可以重新导入 Gradle 项目.

  我们在 IntelliJ IDEA 2020.1 (使用 Gradle 6.0 及以上的版本) 中添加了一种新的动作 – **Load Script Configurations**, 它会加载脚本配置的变更, 而不会更新整个项目. 这样可以比重新导入整个项目花费更少的时间.

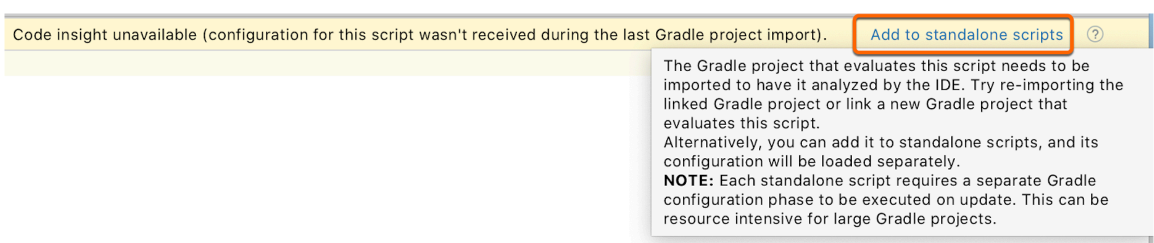


*.gradle.kts – 加载脚本变更和加载 Gradle 变更

对新创建的脚本, 或使用新版本的 Kotlin plugin 第一次打开项目时,也应该 **Load Script Configurations.**

使用 Gradle 6.0 及以上的版本, 你现在可以一次性加载所有的脚本, 以前的版本则不同, 这些脚本会分别独立加载. 由于每次加载都需要执行 Gradle 配置过程, 因此对大的 Gradle 项目可能消耗大量资源.

目前这些加载仅限于 `build.gradle.kts` 和 `settings.gradle.kts` 文件 (请投票支持相关的 issue (https://github.com/gradle/gradle/issues/12640)). 如果要对 `init.gradle.kts` 或应用的 脚本 plugin (https://docs.gradle.org/current/userguide/plugins.html#sec:script_plugins) 启用语法高亮, 请使用旧机制 – 将它们添加到独立脚本. 对这个脚本的配置会在你需要它的时候单独加载. 你也可以对这个脚本启用自动重加载 .



*.gradle.kts – 添加到独立脚本

- *更好的错误报告*. 在以前的版本中, Gradle Daemon 发生的错误只能在分离的日志文件中看到. 现在 Gradle Daemon 直接返回错误的所有信息, 并显示到 Build 工具窗口. 这个改进可以节省你很多时间和精力.

标准库

Kotlin 标准库 1.4.0 中最重要的变化如下:

- 共通的异常处理 API
- 用于数组和集合的新函数

- 字符串操作函数
- 位操作
- 委托属性的改进
- KType 转换为 Java 类型
- 用于 Kotlin 反射的 Proguard 配置
- 既有 API 的改进
- stdlib 库文件的 module-info 描述符
- 废弃的功能
- 删除了已废弃的实验性协程

共通的异常处理 API

以下 API 移动到了共通库中:

- `Throwable.stackTraceToString()` 扩展函数, 返回这个 throwable 的详细描述信息和 stack trace, 以及 `Throwable.printStackTrace()` 函数, 将这个描述信息打印到标准错误输出.
- `Throwable.addSuppressed()` 函数, 你可以指定被压制的异常, 用来传递这些异常, 以及 `Throwable.suppressedExceptions` 属性, 返回所有的被压制的异常的列表.
- `@Throws` 注解, 它列出(在 JVM 或 原生 平台)当函数编译为平台方法时, 需要检查的那些异常类型.

用于数组和集合的新函数

集合

在 1.4.0 中, 标准库包括一组有用的函数, 可用于处理 **collections**:

- `setOfNotNull()`, 根据参数指定的集合生成一个 set, 其中包含所有的非 null 元素.

```
fun main() {
//sampleStart
    val set = setOfNotNull(null, 1, 2, 0, null)
    println(set)
```

```
//sampleEnd
}
```

- shuffled(), 用于序列.

```
fun main() {
//sampleStart
    val numbers = (0 until 50).asSequence()
    val result = numbers.map { it * 2 }.shuffled().take(5)
    println(result.toList()) // 100 以内的 5 个随机偶数
//sampleEnd
}
```

- *Indexed() 版的 onEach() 和 flatMap()函数. 用于集合元素的操作, 参数是元素的下标.

```
fun main() {
//sampleStart
    listOf("a", "b", "c", "d").onEachIndexed {
        index, item -> println(index.toString() + ":" + item)
    }

   val list = listOf("hello", "kot", "lin", "world")
          val kotlin = list.flatMapIndexed { index, item ->
              if (index in 1..2) item.toList() else emptyList()
          }
//sampleEnd
          println(kotlin)
}
```

- *OrNull() 版的 randomOrNull(), reduceOrNull(), 和 reduceIndexedOrNull() 函数. 这些函数对空集合返回 null.

```
fun main() {
//sampleStart
     val empty = emptyList<Int>()
     empty.reduceOrNull { a, b -> a + b }
     //empty.reduce { a, b -> a + b } // 异常: 空集合不能执行 reduce
操作.
```

```
//sampleEnd
}
```

- `runningFold()`, 它是 `scan()` 的同义函数, 以及 `runningReduce()`, 对集合元素顺序的执行给定的操作, 这两个函数与 `fold()` 以及 `reduce()` 类似; 区别是这些新函数返回整个中间结果的序列.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(0, 1, 2, 3, 4, 5)
    val runningReduceSum = numbers.runningReduce { sum, item ->
sum + item }
    val runningFoldSum = numbers.runningFold(10) { sum, item ->
sum + item }
//sampleEnd
    println(runningReduceSum.toString())
    println(runningFoldSum.toString())
}
```

- `sumOf()`, 参数是一个选择器函数, 返回结果是集合所有元素的合计值. `sumOf()` 可以产生 `Int`, `Long`, `Double`, `UInt`, 和 `ULong` 类型的合计值. 在 JVM 平台, 还可以使用 `BigInteger` 和 `BigDecimal`.

```
data class OrderItem(val name: String, val price: Double, val
count: Int)

fun main() {
//sampleStart
    val order = listOf<OrderItem>(
        OrderItem("Cake", price = 10.0, count = 1),
        OrderItem("Coffee", price = 2.5, count = 3),
        OrderItem("Tea", price = 1.5, count = 2))

    val total = order.sumOf { it.price * it.count } // 结果为
Double 类型
    val count = order.sumOf { it.count } // 结果为 Int 类型
//sampleEnd
    println("You've ordered $count items that cost $total in
```

```
total")
}
```

- `min()` 和 `max()` 函数改名为 `minOrNull()` 和 `maxOrNull()`, 以便符合 Kotlin 集合 API 中使用的命名规约. 函数名称后缀 `*OrNull` 代表, 如果接受者集合为空, 它会返回 `null`. 同样的规则也适用于 `minBy()`, `maxBy()`, `minWith()`, `maxWith()` 函数 – 在 1.4 中, 这些函数都存在 `*OrNull()` 版本.

- 新的 `minOf()` 和 `maxOf()` 扩展函数, 指定的选择器函数, 返回集合元素的最小值和最大值.

```
data class OrderItem(val name: String, val price: Double, val
count: Int)

fun main() {
//sampleStart
    val order = listOf<OrderItem>(
        OrderItem("Cake", price = 10.0, count = 1),
        OrderItem("Coffee", price = 2.5, count = 3),
        OrderItem("Tea", price = 1.5, count = 2))
    val highestPrice = order.maxOf { it.price }
//sampleEnd
    println("The most expensive item in the order costs
$highestPrice")
}
```

还有 `minOfWith()` 和 `maxOfWith()` 函数, 参数是 `Comparator`, 以及所有这 4 个函数的 `*OrNull()` 版本, 对空集合会返回 `null`.

- `flatMap` 和 `flatMapTo` 的新的重载版本, 变换处理的返回类型允许与接受者类型不匹配, 也就是:

  - 将 `Iterable`, `Array`, 和 `Map` 变换为 `Sequence`

  - 将 `Sequence` 变换为 `Iterable`

```
fun main() {
//sampleStart
    val list = listOf("kot", "lin")
    val lettersList = list.flatMap { it.asSequence() }
    val lettersSeq = list.asSequence().flatMap { it.toList() }
//sampleEnd
```

```
    println(lettersList)
    println(lettersSeq.toList())
}
```

- `removeFirst()` 和 `removeLast()` 便捷函数, 从可变列表中删除元素, 以及这些函数的 `*orNull()` 版本.

数组

为了在使用不同的容器类型时提供一致的体验, 我们还为 **数组** 添加了新的函数:

- `shuffle()` 将数组元素按随机顺序排列.
- `onEach()` 对每个数组元素执行给定的操作, 返回数组自身.
- `associateWith()` 和 `associateWithTo()` 函数, 使用数组元素作为键(key), 构建 map.
- `reverse()` 对数组的子范围内的元素反转顺序.
- `sortDescending()` 对数组的子范围内的元素逆排序.
- `sort()` 和 `sortWith()` 共通库中现在有了用于数组的子范围的版本.

```
fun main() {
//sampleStart
    var language = ""
    val letters = arrayOf("k", "o", "t", "l", "i", "n")
    val fileExt = letters.onEach { language += it }
       .filterNot { it in "aeuio" }.take(2)
       .joinToString(prefix = ".", separator = "")
    println(language) // "kotlin"
    println(fileExt) // ".kt"

    letters.shuffle()
    letters.reverse(0, 3)
    letters.sortDescending(2, 5)
    println(letters.contentToString()) // [k, o, t, l, i, n]
//sampleEnd
}
```

此外, 还有用于 `CharArray`/`ByteArray` 和 `String` 之间转换的新函数:

- `ByteArray.decodeToString()` 和 `String.encodeToByteArray()`
- `CharArray.concatToString()` 和 `String.toCharArray()`

```
fun main() {
//sampleStart
    val str = "kotlin"
    val array = str.toCharArray()
    println(array.concatToString())
//sampleEnd
}
```

ArrayDeque

我们还添加了 `ArrayDeque` 类 – 双端队列(double-ended queue) 的一个实现. 对双端队列, 既可以从队列头部也可以从队列尾部添加或删除元素, 操作的平摊时间固定(amortized constant time). 如果你的代码需要队列或栈, 那么默认可以使用双端队列.

```
fun main() {
    val deque = ArrayDeque(listOf(1, 2, 3))

    deque.addFirst(0)
    deque.addLast(4)
    println(deque) // [0, 1, 2, 3, 4]

    println(deque.first()) // 0
    println(deque.last()) // 4

    deque.removeFirst()
    deque.removeLast()
    println(deque) // [1, 2, 3]
}
```

`ArrayDeque` 的实现在它的底层使用一个可变大小的数组: 它将队列内容保存在一个环形缓冲区, 也就是一个 `Array`, 并且只有当 `Array` 已满的时候才会调整这个 `Array` 的大小.

字符串操作函数

1.4.0 的标准库中, 字符串操作 API 包含一系列改进:

- `StringBuilder` 有很多有用的新扩展函数: `set()`, `setRange()`, `deleteAt()`, `deleteRange()`, `appendRange()`, 以及其他函数.

```
fun main() {
//sampleStart
    val sb = StringBuilder("Bye Kotlin 1.3.72")
    sb.deleteRange(0, 3)
    sb.insertRange(0, "Hello", 0 ,5)
    sb.set(15, '4')
    sb.setRange(17, 19, "0")
    print(sb.toString())
//sampleEnd
}
```

- 在共通库中可以使用 `StringBuilder` 的一些既有函数. 包括 `append()`, `insert()`, `substring()`, `setLength()`, 以及其他函数.

- 共通库添加了新函数 `Appendable.appendLine()` 和 `StringBuilder.appendLine()`. 这些函数替代了这些类的 `appendln()` 函数, 它只能用于 JVM 平台.

```
fun main() {
//sampleStart
    println(buildString {
        appendLine("Hello,")
        appendLine("world")
    })
//sampleEnd
}
```

位操作

用于位操作的新函数:

- `countOneBits()`
- `countLeadingZeroBits()`
- `countTrailingZeroBits()`
- `takeHighestOneBit()`

- `takeLowestOneBit()`
- `rotateLeft()` 和 `rotateRight()` (实验性功能)

```
fun main() {
//sampleStart
    val number = "1010000".toInt(radix = 2)
    println(number.countOneBits())
    println(number.countTrailingZeroBits())
    println(number.takeHighestOneBit().toString(2))
//sampleEnd
}
```

委托属性的改进

在 1.4.0 中, 我们添加了新的功能特性, 改进 Kotlin 委托属性的使用体验:

- 现在一个属性可以委托给另一个属性.
- 新接口 `PropertyDelegateProvider` 帮助你使用单条声明来创建委托 provider.
- `ReadWriteProperty` 现在继承 `ReadOnlyProperty`, 因此对只读属性这两个接口都可以使用.

除了这些新 API 之外, 我们还做了一些优化, 来减少编译产生的字节码大小. 关于这些优化, 请参见这篇 Blog (https://blog.jetbrains.com/kotlin/2019/12/what-to-expect-in-kotlin-1-4-and-beyond/#delegated-properties).

关于委托属性, 请阅读这篇文档 (委托属性).

KType 转换为 Java 类型

标准库中的一个新的扩展属性 `KType.javaType` (目前还在实验状态) 可以帮助你从 Kotlin 类型获取 `java.lang.reflect.Type`, 而不需要使用完整的 `kotlin-reflect` 依赖项.

```
import kotlin.reflect.javaType
import kotlin.reflect.typeOf

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T> accessReifiedTypeArg() {
   val kType = typeOf<T>()
   println("Kotlin type: $kType")
```

```
    println("Java type: ${kType.javaType}")
}

@OptIn(ExperimentalStdlibApi::class)
fun main() {
    accessReifiedTypeArg<String>()
    // Kotlin type: kotlin.String
    // Java type: class java.lang.String

    accessReifiedTypeArg<List<String>>()
    // Kotlin type: kotlin.collections.List<kotlin.String>
    // Java type: java.util.List<java.lang.String>
}
```

用于 Kotlin 反射的 Proguard 配置

从 1.4.0 开始, 我们在 `kotlin-reflect.jar` 中嵌入了对 Kotlin 反射的 Proguard/R8 配置. 使用这个内置配置, 大多数使用 R8 或 Proguard 的 Android 项目可以与 kotlin-reflect 协同工作, 不需要任何额外的配置. 你不再需要复制粘贴用于 kotlin-reflect 的 Proguard 规则. 但要注意, 你仍然需要明确列出你需要反射的所有 API.

既有 API 的改进

- 一些函数现在可以使用 null 接受者, 比如:
  - 用于字符串的 `toBoolean()`
  - 用于数组的 `contentEquals()`, `contentHashcode()`, `contentToString()`
- `Double` 和 `Float` 中的 `NaN`, `NEGATIVE_INFINITY`, 和 `POSITIVE_INFINITY` 现在定义为 `const`, 因此你可以将它们用作注解参数.
- `Double` 和 `Float` 中的新常数 `SIZE_BITS` 和 `SIZE_BYTES`, 常数值为这些类型的二进制表达形式使用的位数和字节数.
- 顶级函数 `maxOf()` 和 `minOf()` 可以接受可变数量的参数(`vararg`).

stdlib 库文件的 module-info 描述符

Kotlin 1.4.0 对标准库的默认 artifact 添加了 `module-info.java` 模块信息. 因此你可以使用在 jlink tool (https://docs.oracle.com/en/java/javase/11/tools/jlink.html) 中使用它们, 这个工具会生成

自定义的 Java 运行时可执行文件, 其中只包含 你的 App 所需要的平台模块. 以前, 你可能已经用 Kotlin 标准库 artifact 使用过 jlink, 但那时你需要使用分离的 artifact – 带有 "modular" 分类的那个 – 而且整个设置也不直观. 在 Android 中, 必须使用 Android Gradle plugin 版本 3.2 或更高版本, 这个版本可以正确处理带有模块信息的 jar 文件.

废弃的功能

Double 和 Float 的 toShort() 和 toByte() 函数

我们废弃了 `Double` 和 `Float` 的 `toShort()` 和 `toByte()` 函数, 原因是, 由于变量的取值范围变小, 这些函数可能产生预料之外的结果.

要将浮点数转换为 `Byte` 或 `Short`, 请使用二步转换: 首先, 转换为 `Int`, 然后再转换为目标类型.

浮点数数组的 contains(), indexOf(), 和 lastIndexOf() 函数

我们废弃了 `FloatArray` 和 `DoubleArray` 的 `contains()`, `indexOf()`, 和 `lastIndexOf()` 扩展函数, 因为它们使用 IEEE 754 (https://en.wikipedia.org/wiki/IEEE_754) 标准的相等比较, 在一些极端情况下会与全顺序相等性(total order equality)矛盾. 详情请参见 这个 issue (https://youtrack.jetbrains.com/issue/KT-28753).

集合的 min() 和 max() 函数

我们废弃了集合函数 `min()` 和 `max()`, 改用 `minOrNull()` 和 `maxOrNull()`, 新的函数名更能反映它们的行为 – 对空集合返回 `null`. 详情请参见 这个 issue (https://youtrack.jetbrains.com/issue/KT-38854).

删除了已废弃的实验性协程

在 1.3.0 中 `kotlin.coroutines.experimental` API 已被废弃, 改用 kotlin.coroutines. 在 1.4.0 中, 我们结束了 `kotlin.coroutines.experimental` 的废弃周期, 将它从标准库删除. 对于还在 JVM 上使用它的开发者, 我们提供了一个兼容的库 `kotlin-coroutines-experimental-compat.jar`, 其中包含所有的实验性协程 API. 我们已经将它发布到了 Maven, 而且在 Kotlin 发布版中, 除标准库之外也包括了这个库.

JSON 序列化的稳定版

在 Kotlin 1.4.0 中, 我们发布了 kotlinx.serialization (https://github.com/Kotlin/kotlinx.serialization) 的第一个稳定版本 - 1.0.0-RC. 现在我们高兴的宣布 `kotlinx-serialization-core` (以前称为 `kotlinx-serialization-runtime`) 中的 JSON 序列化 API 已经达到稳定状态. 用于其他序列化格式的库, 以及核心库的一些高级模块, 还继续处在实验状态.

我们还大量重写了 JSON 序列化 API, 使它更加统一, 而且更加易于使用. 今后我们还会继续开发 JSON 序列化 API, 并保持向后兼容. 但是, 如果你使用了这些 API 的以前的版本, 迁移到 1.0.0-RC

版时, 你将会需要重写你的一部分代码 . 为了帮助你进行版本迁移, 我们提供了一份 **Kotlin 序列化指南 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/serialization-guide.md)** – `kotlinx.serialization` 的完整文档. 这篇文档可以指导你如何使用那些最重要的功能特性 而且可以帮助你解决可能遇到的问题.

> **注意**: `kotlinx-serialization` 1.0.0-RC 只能在 Kotlin 编译器 1.4 下使用. 以前的编译器版本不能兼容.

脚本与 REPL

在 1.4.0 中, Kotlin 脚本有了很多功能和性能的改进, 以及其他更新. 下面是一些关键性的改变:

- 新的依赖项解析 API
- 新的 REPL API
- 脚本编译缓存
- Artifact 重命名

为了帮助你熟悉 Kotlin 脚本, 我们贮备了一个 示例项目 (https://github.com/Kotlin/kotlin-script-examples). 其中包含标准脚本 (`*.main.kts`) 示例, 以及 Kotlin 脚本 API 的使用和自定义脚本定义的示例. 欢迎试用, 并通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 反馈你的意见.

新的依赖项解析 API

在 1.4.0 中, 我们引入了一个新的 API 来解析外部依赖项 (比如 Maven artifact), 以及它的实现. 这个 API 已经发布到了新的 artifact `kotlin-scripting-dependencies` 和 `kotlin-scripting-dependencies-maven` 中. 以前在 `kotlin-script-util` 库中的依赖项解析功能现在已经废弃.

新的 REPL API

新的实验性的 REPL API 现在是 Kotlin 脚本 API 的一部分. 在发布的 artifact 中存在几个实现, 以及一些高级功能, 比如代码完成. 我们在 Kotlin Jupyter kernel (https://blog.jetbrains.com/kotlin/2020/05/kotlin-kernel-for-jupyter-notebook-v0-8/) 中使用了这个 API, 现在你可以在你自己的自定义 shell 和 REPL 中试用这个 API.

脚本编译缓存

Kotlin 脚本 API 现在能够实现脚本编译的缓存, 显著的提高了随后的未修改的脚本的执行速度. 我们的默认高级脚本实现 `kotlin-main-kts` 已经有了它自己的缓存.

Artifact 重命名

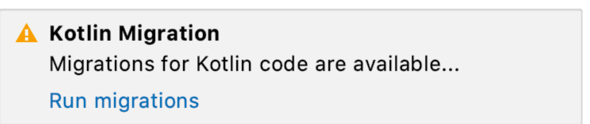
为了避免 artifact 的名称混淆, 我们将 `kotlin-scripting-jsr223-embeddable` 和 `kotlin-scripting-jvm-host-embeddable` 重命名为 `kotlin-scripting-jsr223` 和 `kotlin-scripting-jvm-host`. 这些 artifact 依赖于 `kotlin-compiler-embeddable` artifact, 它会对打包的第三方库进行 shade, 以避免库的使用冲突. 通过这次重命名, 我们让脚本 artifact 默认使用 `kotlin-compiler-embeddable` (它通常更加安全). 如果出于某些原因, 你希望使用的 artifact 依赖于没有 shade 的 `kotlin-compiler`, 那么请使用带 `-unshaded` 后缀的 artifact 版本, 比如 `kotlin-scripting-jsr223-unshaded`. 注意, 这次重命名只影响那些应该直接使用的脚本 artifact; 其他 artifact 的名称维持不变.

迁移到 Kotlin 1.4.0

Kotlin plugin 的迁移工具, 可以帮助你将项目从更早的 Kotlin 版本迁移到 1.4.0.

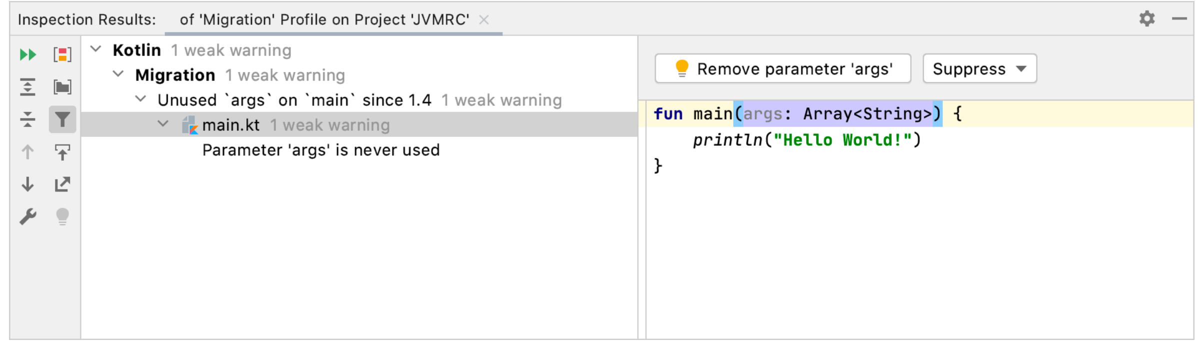
只需要修改 Kotlin 版本到 `1.4.0`, 然后重新 import 你的 Gradle 或 Maven 项目. 然后 IDE 会询问你是否迁移.

如果你同意, IDE 会执行迁移代码审查, 这个过程会检查你的代码, 找出在 1.4.0 中不能正确工作的代码, 或不推荐的做法, 并建议如何修正.



执行迁移

代码审查有不同的 严重等级 (https://www.jetbrains.com/help/idea/configuring-inspection-severities.html), 可以帮助你决定接受哪些修正建议, 忽略哪些修正建议.



迁移审查

Kotlin 1.4.0 是一个 功能性发布版(Feature Release) ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化"), 因此会对语言带来一些不兼容的变更. 关于这些变更的详情, 请参加 **Kotlin 1.4 兼容性指南 (Kotlin 1.4 兼容性指南)**.

Kotlin 1.3 版中的新功能

最终更新: 2024/09/10

发布日期: 2018/10/29

协程功能正式发布

经过长期广泛的实战测试之后, 协程功能终于正式发布了! 也就是说, 从 Kotlin 1.3 开始, 协程功能的语言级支持, 以及 API 都进入 完全稳定 (Kotlin 各部分组件的稳定性) 状态. 请参见新的 协程概述 (协程(Coroutine)) 文档.

Kotlin 1.3 引入了挂起函数的可调用的引用, 并在反射 API 中支持协程.

Kotlin/Native

Kotlin 1.3 继续改进对原生程序开发的. 详情请参见 Kotlin/Native 概述 (使用 Kotlin/Native 进行原生(Native)程序开发).

跨平台项目

在 1.3 中, 我们完成了对跨平台项目模式的重构工作, 改进了表达能力和灵活性, 使得共用代码变得更加容易. 而且, Kotlin/Native 现在也是我们支持的目标平台之一了!

与旧模式的主要不同在于:

- 在旧模式中, 共通代码和平台相关代码需要放在不同的模块中, 然后使用 `expectedBy` 依赖项导入. 现在, 共通代码和平台相关代码放在同一模块的不同源代码路径中, 项目配置变得更加容易.
- 对于支持的各种目标平台, 现在有了大量的 预定义平台配置 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档").
- 依赖项配置 (添加跨平台库依赖项)有了变化; 现在以各个源代码路径为单位分别指定依赖项.
- 源代码集现在可以在任意一部分平台之间共用(比如, 在编译目标平台为 JS, Android 和 iOS 的模块中, 你可以让某个源代码集只在 Android 和 iOS 平台中共用).
- 现在支持 发布跨平台的库 (发布跨平台的库).

更多详细信息, 请参见 跨平台程序开发文档 (Kotlin Multiplatform).

契约(Contract)

Kotlin 编译器会进行大量的静态分析, 产生警告信息, 并减少样板代码. 其中最值得注意的功能之一就是智能类型转换 — 根据已有的类型检查代码, 可以自动进行类型转换:

```
fun foo(s: String?) {
    if (s != null) s.length // 编译器自动将 's' 转换为 'String' 类型
}
```

但是, 一旦将这些类型检查抽取到一个独立的函数中, 这些智能类型转换就消失了:

```
fun String?.isNotNull(): Boolean = this != null

fun foo(s: String?) {
    if (s.isNotNull()) s.length // 这类没有智能类型转换 :(
}
```

为了改进这种情况下的编译器能力, Kotlin 1.3 引入了一个实验性的机制, 名为 *契约* (contract).

契约 允许一个函数以编译器能够理解的方式明确地描述它的行为. 目前, 支持两打大类使用场景:

- 声明一个函数调用的入口参数与输出结果之间的关系, 来改进编译器的智能类型转换分析能力:

```
fun require(condition: Boolean) {
    // 这是一个语法形式, 告诉编译器:
    // "如果这个函数成功地返回, 那么传入到这个函数内的 'condition' 为 true"
    contract { returns() implies condition }
    if (!condition) throw IllegalArgumentException(...)
}

fun foo(s: String?) {
    require(s is String)
    // 这里 's' 会被智能转换为 'String', 因为, 如果它不是 'String',
    // 'require' 应该抛出异常
}
```

- 出现高阶函数时, 改进编译器的变量初始化分析能力:

```
fun synchronize(lock: Any?, block: () -> Unit) {
    // 这段代码告诉编译器:
    // "这个函数会立即调用 'block', 而且只调用一次"
    contract { callsInPlace(block, EXACTLY_ONCE) }
}

fun foo() {
    val x: Int
    synchronize(lock) {
        x = 42 // 编译器知道传递给 'synchronize' 的 lambda 表达式会被刚好
调用一次
               // 因此不会报告 'x' 被多次赋值的错误
    }
    println(x) // 编译器知道 lambda 表达式一定会被调用一次, 并执行对 'x' 的
初始化
               // 因此在这里会认为 'x' 已被初始化
}
```

标准库中的契约

`stdlib` 已经使用了契约, 用来改进上文介绍的编译器分析能力. 这部分契约是 **稳定** 的, 也就是说你不必添加额外的编译选项, 也能得到编译器分析能力的提高:

```
//sampleStart
fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // 哇~~~, 可以智能转
换为非空类型!
    }
}
//sampleEnd
fun main() {
    bar(null)
    bar("42")
}
```

自定义的契约

也可以为你自己的函数声明契约, 但这个功能还是 **实验性** 的, 因为契约目前的语法还处于早期原型阶段, 将来很可能会改变. 而且请注意, 目前 Kotlin 编译器不会去验证契约的内容, 因此程序员需要自己负责编写正确而且完整的契约.

通过调用标注库的 `contract` 函数, 就可以声明自定义的契约, 这个函数会产生一个 DSL 作用域:

```
fun String?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }
    return this == null || isEmpty()
}
```

关于契约的语法, 以及兼容性问题, 详情请参见 KEEP (https://github.com/Kotlin/KEEP/blob/master/proposals/kotlin-contracts.md).

将 when 语句的判定对象保存到变量中

在 Kotlin 1.3 中, 可以将 `when` 语句的判定对象保存到变量中:

```
fun Request.getBody() =
        when (val response = executeRequest()) {
            is Success -> response.body
            is HttpError -> throw HttpException(response.status)
        }
```

虽然我们可以在 `when` 语句之前抽取这个变量, 但 `when` 语句中的 `val` 变量的作用范围会被限定在 `when` 的语句体之内, 因此可以防止它扩散到更广的范围. 关于 `when` 语句的完整文档, 请阅读这里 ("when 表达式" in "条件与循环").

对接口的同伴对象使用 @JvmStatic 和 @JvmField 注解

在 Kotlin 1.3 中, 可以对接口的 `companion` 对象的成员标记 `@JvmStatic` 和 `@JvmField` 注解. 在编译产生的类文件中, 这些成员会被提升到对应的接口内, 并变为 `static` 成员.

比如, 以下 Kotlin 代码:

```
interface Foo {
    companion object {
        @JvmField
```

```
        val answer: Int = 42

        @JvmStatic
        fun sayHello() {
            println("Hello, world!")
        }
    }
}
```

等价于以下 Java 代码:

```
interface Foo {
    public static int answer = 42;
    public static void sayHello() {
        // ...
    }
}
```

注解类中的嵌套声明

在 Kotlin 1.3 中, 注解可以拥有嵌套的类, 接口, 对象, 以及同伴对象:

```
annotation class Foo {
    enum class Direction { UP, DOWN, LEFT, RIGHT }

    annotation class Bar

    companion object {
        fun foo(): Int = 42
        val bar: Int = 42
    }
}
```

无参数的 main 函数

按照习惯, Kotlin 程序的入口是一个签名类似 `main(args: Array<String>)` 的函数, 其中 `args` 表示传递给这个程序的命令行参数. 但是, 并不是每个程序都支持命令行参数, 因此这个参数在程序中经常没有被使用.

Kotlin 1.3 引入了一个更简单的 `main` 函数形式, 它可以没有任何参数. "Hello, World" 程序在 Kotlin 代码中可以减少 19 个字符了!

```
fun main() {
    println("Hello, world!")
}
```

带巨量参数的函数

在 Kotlin 中, 函数类型被表达为一个泛型, 接受不同数量的参数: `Function0<R>`, `Function1<P0, R>`, `Function2<P0, P1, R>`, ... 这种方案存在的一个问题就是, 参数个数是有限的, 目前只支持到 `Function22`.

Kotlin 1.3 放宽了这个限制, 支持更多参数的函数:

```
fun trueEnterpriseComesToKotlin(block: (Any, Any, ... /* 另外还有 42 个 */, Any) -> Any) {
    block(Any(), Any(), ..., Any())
}
```

渐进模式

Kotlin 非常关注稳定性, 以及源代码的向后兼容: Kotlin 的兼容性政策是: 破坏性变更 (也就是, 某些变更会造成过去能够成功编译的代码无法编译) 只能出现在主版本中 (**1.2**, **1.3**, 等等.).

我们相信, 很多用户会使用更快速的升级, 对于严重的编译器 bug 可以立即得到修正, 使得代码更加安全, 更加正确. 因此, Kotlin 1.3 引入了 *渐进式* 编译模式, 可以向编译器添加 `-progressive` 参数来启用这个模式.

在渐进模式下, 会立即启用某些语法层面的修正. 这些修正包含两个重要的特性:

- 这些修正保证源代码在旧版本编译器上的向后兼容性, 也就是说, 凡是渐进模式下能够编译的代码, 在非渐进模式下也能正确编译.
- 这些修正只会让代码 *更正确* — 比如, 有些不适当的智能类型转换会被禁止, 编译产生的代码的行为可能会被修改, 变得更可预测, 更加稳定, 等等.

启用渐进模式可能会要求你重写某些代码, 但不会太多 — 渐进模式下启用的修正都经过仔细挑选, 检查, 并且提供了代码迁移的辅助工具. 对于那些活跃开发中, 快速更新语言版本的代码库, 我们期望

渐进模式能够成为一个好的选择.

内联类

> 内联类目前处于 Alpha 阶段 (Kotlin 各部分组件的稳定性). 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见. 详情请参见 参考文档 (内联的值类(Inline value class)).

Kotlin 1.3 引入了一种新的类型声明 — `inline class`. 内联类可以看作一种功能受到限制的类, 具体来说, 内联类只能有一个属性, 不能更多, 也不能更少:

```
inline class Name(val s: String)
```

Kotlin 编译器会使用这个限制, 尽力优化内联类的运行期表达, 用内联类底层属性的值来代替内联类的实例, 因此可以去除构造器调用, 减少 GC 压力, 而且可以进行进一步的代码优化:

```
inline class Name(val s: String)
//sampleStart
fun main() {
    // 下一行代码不会发生构造器调用, 而且在运行期 'name' 中只会包含字符串
"Kotlin"
    val name = Name("Kotlin")
    println(name.s)
}
//sampleEnd
```

关于内联类的详情, 请参见 参考文档 (内联的值类(Inline value class)).

无符号整数

> 无符号整数目前处于 Beta 阶段 (Kotlin 各部分组件的稳定性). 具体实现已经基本稳定, 但将来可能会需要手工迁移你的代码. 我们会尽力减少你需要修改的代码量.

Kotlin 1.3 引入了无符号整数类型:

- `kotlin.UByte`: 无符号的 8 位整数, 值范围是 0 到 255

- kotlin.UShort: 无符号的 16 位整数, 值范围是 0 到 65535
- kotlin.UInt: 无符号的 32 位整数, 值范围是 0 到 2^32 - 1
- kotlin.ULong: 无符号的 64 位整数, 值范围是 0 到 2^64 - 1

有符号整数所支持的大多数功能, 对无符号整数也适用:

```
fun main() {
//sampleStart
// 可以使用字面值后缀来定义无符号整数
val uint = 42u
val ulong = 42uL
val ubyte: UByte = 255u

// 使用标准库中的扩展函数, 可以将有符号类型转换为无符号类型, 或者反过来:
val int = uint.toInt()
val byte = ubyte.toByte()
val ulong2 = byte.toULong()

// 无符号整数支持类似的运算符:
val x = 20u + 22u
val y = 1u shl 8
val z = "128".toUByte()
val range = 1u..5u
//sampleEnd
println("ubyte: $ubyte, byte: $byte, ulong2: $ulong2")
println("x: $x, y: $y, z: $z, range: $range")
}
```

详情请参见 参考文档 (无符号整数(Unsigned Integer)类型).

@JvmDefault 注解

> @JvmDefault 目前还处于实验性阶段 (Kotlin 各部分组件的稳定性). 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin 支持许多 Java 版本, 包括 Java 6 和 Java 7, 在这些版本上还不支持接口的默认方法. 为了你编程的方便, Kotlin 编译器绕过了这个限制, 但是这个解决方法无法与 Java 8 中的 `default` 方法兼容.

这可能会造成与 Java 互操作时的问题, 因此 Kotlin 1.3 引入了 `@JvmDefault` 注解. 使用了这个注解的方法, 在 JVM 平台上会被编译为 `default` 方法:

```
interface Foo {
    // 会被编译为 'default' 方法
    @JvmDefault
    fun foo(): Int = 42
}
```

> 警告! 使用 `@JvmDefault` 注解来标注你的 API 会对二进制兼容性造成严重的影响. 在你的产品代码中使用 `@JvmDefault` 之前, 请一定要认真阅读 参考文档 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-default/index.html).

标准库

跨平台的 Random 类

在 Kotlin 1.3 之前, 没有统一的方法在所有的平台上生成随机数 — 我们必须使用各种平台独自的解决方案, 比如在 JVM 上使用 `java.util.Random`. Kotlin 1.3 版引入 `kotlin.random.Random` 类, 解决了这个问题, 这个类可以在所有的平台上使用:

```
import kotlin.random.Random

fun main() {
//sampleStart
    val number = Random.nextInt(42)  // 得到的随机数范围是 [0, limit)
    println(number)
//sampleEnd
}
```

isNullOrEmpty 和 orEmpty 扩展函数

标准库提供了对某些数据类型的 `isNullOrEmpty` 和 `orEmpty` 扩展函数. 如果接受者是 `null`, 或内容为空, 那么 `isNullOrEmpty` 函数返回 `true`, 如果接受者是 `null`, 那么 `orEmpty` 函数返回一个不

为 `null`, 但内容为空的实例. Kotlin 1.3 对集合(Collection), Map, 以及对象数组, 都提供了类似的扩展函数.

在两个既有的数组之间复制元素

对既有的数组类型, 包括无符号整数数组, 提供了 `array.copyInto(targetArray, targetOffset, startIndex, endIndex)` 扩展函数, 可以使用纯 Kotlin 代码, 更简单地实现基于数组的容器.

```
fun main() {
//sampleStart
    val sourceArr = arrayOf("k", "o", "t", "l", "i", "n")
    val targetArr = sourceArr.copyInto(arrayOfNulls<String>(6), 3,
startIndex = 3, endIndex = 6)
    println(targetArr.contentToString())

    sourceArr.copyInto(targetArr, startIndex = 0, endIndex = 3)
    println(targetArr.contentToString())
//sampleEnd
}
```

associateWith 函数

已有一组 key 值, 希望将每一个 Key 与某个值关联起来, 创建一个 Map, 这是很常见的情况. 以前, 使用 `associate { it to getValue(it) }` 函数, 也是可以做到的, 但是现在我们引入了一个更加高效, 而且更加易用的新函数: `keys.associateWith { getValue(it) }`.

```
fun main() {
//sampleStart
    val keys = 'a'..'f'
    val map = keys.associateWith {
it.toString().repeat(5).capitalize() }
    map.forEach { println(it) }
//sampleEnd
}
```

ifEmpty 和 ifBlank 函数

对于集合(Collection), Map, 对象数组, 字符序列, 以及值序列(equence), 现在有了 `ifEmpty` 函数, 对于接受者对象内容为空的情况, 可以指定一个替代值:

```
fun main() {
//sampleStart
    fun printAllUppercase(data: List<String>) {
        val result = data
        .filter { it.all { c -> c.isUpperCase() } }
            .ifEmpty { listOf("<no uppercase>") }
        result.forEach { println(it) }
    }

    printAllUppercase(listOf("foo", "Bar"))
    printAllUppercase(listOf("FOO", "BAR"))
//sampleEnd
}
```

除此之外, 字符序列和字符串还有一个 `ifBlank` 扩展函数, 它和 `ifEmpty` 函数一样, 也会使用指定的替代值, 但它检查的条件是字符串内容是否全部是空白字符.

```
fun main() {
//sampleStart
    val s = "    \n"
    println(s.ifBlank { "<blank>" })
    println(s.ifBlank { null })
//sampleEnd
}
```

在反射中使用封闭类

我们对 `kotlin-reflect` 添加了一个新的 API, 名为 `KClass.sealedSubclasses`, 可以用来得到 `sealed` 类的所有直接子类型.

细微变更

- `Boolean` 类型现在带有同伴对象.
- `Any?.hashCode()` 扩展函数, 对 `null` 值返回 0.
- `Char` 现在带有 `MIN_VALUE` 和 `MAX_VALUE` 常数.
- 基本类型的同伴对象中增加了 `SIZE_BYTES` 和 `SIZE_BITS` 常数.

工具

在 IDE 中支持代码风格

Kotlin 1.3 开始在 IntelliJ IDEA 中支持 推荐的代码风格 (编码规约). 关于代码迁移的方法, 请参见 参考文档 (迁移到 Kotlin 编码风格).

kotlinx.serialization

kotlinx.serialization (https://github.com/Kotlin/kotlinx.serialization) 是一个库, 在 Kotlin 中跨平台支持对象的序列化和反序列化. 以前它曾是一个独立的项目, 但从 Kotlin 1.3 起, 它和其他编译器 plugin 一样, 随 Kotlin 编译器一起发布. 主要的区别是, 你不需要手工维护 IDE 的序列化 Plugin 与你使用的 Kotlin IDE Plugin 之间的版本兼容问题: 因为现在 Kotlin IDE Plugin 已经包含了序列化功能!

详情请参见 参考文档 (https://github.com/Kotlin/kotlinx.serialization#current-project-status).

> 虽然现在 kotlinx.serialization 与 Kotlin 编译器一起发布, 但在 Kotlin 1.3 中它仍然是一个实验性功能.

脚本 API 升级

> 脚本是一个实验性功能 (Kotlin 各部分组件的稳定性), 这个功能随时可能会放弃或发生修改. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin 1.3 仍在持续改进脚本 API, 引入了一些实验性的功能, 支持脚本的定制, 包括添加外部属性, 提供静态或动态的依赖项, 等等.

详情请参见, KEEP-75 (https://github.com/Kotlin/KEEP/blob/master/proposals/scripting-support.md).

支持草稿文件(Scratch File)

Kotlin 1.3 开始支持可运行的 Kotlin *草稿文件(Scratch File)*. *草稿文件* 是一个扩展名为 .kts 的 Kotlin 脚本文件, 你可以直接在编辑器中运行这个文件, 并得到执行结果.

详情请参见 草稿文件参考文档 (https://www.jetbrains.com/help/idea/scratches.html).

Kotlin 1.2 版中的新功能

最终更新: 2024/09/10

发布日期: 2017/11/28

目录

- 跨平台项目(Multiplatform project)
- 语言层的其他特性
- 标准库
- JVM 环境(JVM Backend)
- JavaScript 环境(JavaScript Backend)

跨平台项目(Multiplatform Project) (实验性功能)

跨平台项目(Multiplatform Project)是 Kotlin 1.2 的一个 **实验性的** 新功能, 它允许你在 Kotlin 支持的多种编译目标平台之间共用代码 – JVM, JavaScript 以及 (将来的) 原生代码. 一个跨平台项目, 由以下 3 种模块构成:

- *common* 模块, 其中包含不依赖于任何平台的代码, 也可以包含与平台相关的 API 声明, 但不包括其实现.
- *platform* 模块, 其中包含 *common* 模块中声明的依赖于平台的 API 在具体平台上的实现代码, 以及其他依赖于平台的代码.
- 通常的模块, 这种模块针对特定的平台, 它可以被 *platform* 模块依赖, 也可以依赖于 *platform* 模块.

当针对某个特定的平台编译跨平台项目时, 共通部分的代码, 以及针对特定平台的代码, 都会被编译生成.

跨平台项目的一个关键性功能就是, 能够通过 **预期声明声明(expected declaration)** 与 **实际声明(actual declaration)** 来表达共通代码对依赖于平台的代码的依赖关系. **预期声明(expected**

declaration) 负责定义一个 API (类, 接口, 注解, 顶级声明, 等等). **实际声明(actual declaration)** 可以是这个 API 的依赖于平台的实现, 也可以是类型别名, 引用这个 API 在外部库中的实现. 示例如下:

在 common 代码中:

```
// 与平台相关的 API 的预期声明:
expect fun hello(world: String): String

fun greet() {
    // 通过预期声明定义的 API 可以这样使用:
    val greeting = hello("multiplatform world")
    println(greeting)
}

expect class URL(spec: String) {
    open fun getHost(): String
    open fun getPath(): String
}
```

在 JVM 平台的代码中:

```
actual fun hello(world: String): String =
    "Hello, $world, on the JVM platform!"

// 使用特定平台中已存在的实现:
actual typealias URL = java.net.URL
```

关于创建跨平台项目的详细步骤, 请参见 跨平台程序开发相关文档 (Kotlin Multiplatform).

语言层的其他特性

在注解中使用数组字面值

从 Kotlin 1.2 开始, 注解中的数组类型参数, 可以通过新的字面值语法来指定, 而不必使用 `arrayOf` 函数:

```
@CacheConfig(cacheNames = ["books", "default"])
public class BookRepositoryImpl {
```

```
    // ...
}
```

数组的字面值语法只能用于注解的参数.

顶级的属性和局部变量的延迟初始化(Lateinit)

`lateinit` 修饰符现在可以用于顶级属性和局部变量. 比如, 当某个对象的构造器参数是一个 lambda 表达式, 而这个 lambda 表达式又引用到了另一个之后才能定义的对象, 这时就可以使用延迟初始化的局部变量:

```
class Node<T>(val value: T, val next: () -> Node<T>)

fun main(args: Array<String>) {
    // 3 个节点组成的环:
    lateinit var third: Node<Int>

    val second = Node(2, next = { third })
    val first = Node(1, next = { second })

    third = Node(3, next = { first })

    val nodes = generateSequence(first) { it.next() }
    println("Values in the cycle: ${nodes.take(7).joinToString {
it.value.toString() }}, ...")
}
```

检查一个延迟初始化的变量是否已被初始化

现在你可以在属性引用上使用 `isInitialized`, 检查一个延迟初始化的变量是否已被初始化, :

```
class Foo {
    lateinit var lateinitVar: String

    fun initializationLogic() {
//sampleStart
        println("isInitialized before assignment: " +
this::lateinitVar.isInitialized)
        lateinitVar = "value"
        println("isInitialized after assignment: " +
```

```
this::lateinitVar.isInitialized)
//sampleEnd
    }
}

fun main(args: Array<String>) {
    Foo().initializationLogic()
}
```

内联函数(Inline function) 的函数性参数的默认值

内联函数的函数性参数, 现在允许使用默认值:

```
//sampleStart
inline fun <E> Iterable<E>.strings(transform: (E) -> String = {
it.toString() }) =
    map { transform(it) }

val defaultStrings = listOf(1, 2, 3).strings()
val customStrings = listOf(1, 2, 3).strings { "($it)" }
//sampleEnd

fun main(args: Array<String>) {
    println("defaultStrings = $defaultStrings")
    println("customStrings = $customStrings")
}
```

显式类型转换的相关信息可被用于类型推断

Kotlin 编译器现在可以将类型转换的相关信息用于类型推断. 如果你调用了一个泛型方法, 返回值为类型参数 `T`, 然后将其转换为确切的类型 `Foo`, 编译器能够正确地判定, 这个方法调用的类型参数 `T` 应该绑定为 `Foo` 类型.

这个功能对于 Android 开发者尤其重要, 因为编译器能够正确地分析 Android API level 26 的泛型方法 `findViewById` 调用:

```
val button = findViewById(R.id.button) as Button
```

智能类型转换的功能改进

如果将一个安全的方法调用(safe call)表达式赋值给一个变量, 然后对这个变量进行 null 值检查, 这时安全方法调用的接受者对象也会被智能类型转换:

```
fun countFirst(s: Any): Int {
//sampleStart
    val firstChar = (s as? CharSequence)?.firstOrNull()
    if (firstChar != null)
      return s.count { it == firstChar } // s: Any 类型被智能转换为
CharSequence 类型

    val firstItem = (s as? Iterable<*>)?.firstOrNull()
    if (firstItem != null)
      return s.count { it == firstItem } // s: Any 类型被智能转换为
Iterable<*> 类型
//sampleEnd
    return -1
}

fun main(args: Array<String>) {
  val string = "abacaba"
  val countInString = countFirst(string)
  println("called on \"$string\": $countInString")

  val list = listOf(1, 2, 3, 1, 2)
  val countInList = countFirst(list)
  println("called on $list: $countInList")
}
```

此外, 如果局部变量值的修改只发生在一个 lambda 表达式之前, 那么在这个 lambda 表达式之内, 这个局部变量也可以被智能类型转换:

```
fun main(args: Array<String>) {
//sampleStart
    val flag = args.size == 0
    var x: String? = null
    if (flag) x = "Yahoo!"

    run {
```

```
        if (x != null) {
            println(x.length) // x 被智能转换为 String 类型
        }
    }
//sampleEnd
}
```

允许将 this::foo 简写为 ::foo

绑定到 `this` 的成员上的可调用的引用, 可以不用明确地指定接受者, 也就是说, `this::foo` 可以简写为 `::foo`. 在 lambda 表达式中, 如果你引用外层接受者的成员, 这种简化语法也使得可调用的引用更加便于使用.

破坏性变更: 对 try 代码段之后的智能类型转换进行警告

在以前的版本, Kotlin 使用 `try` 代码段之内的赋值语句来控制 `try` 代码段之后的智能类型转换, 这样的规则可能会破坏类型安全性和 null 值安全性, 并导致运行期的错误. Kotlin 1.2 修正了这个问题, 对智能类型转换的限制变得更加严格, 但会导致依赖这种智能类型转换的代码无法运行.

如果想要继续使用旧版本的智能类型转换, 请对编译器指定 `-Xlegacy-smart-cast-after-try` 参数. 这个参数将在 Kotlin 1.3 版本中废弃.

已废弃的功能: 覆盖 copy 函数的数据类

当数据类的超类中已经存在相同签名的 `copy` 函数, 数据类中自动生成的 `copy` 函数实现将会使用超类中的默认实现, 这就会导致数据类的行为不符合我们通常的直觉, 而且, 如果超类中没有默认参数, 还会在运行期发生错误.

这种导致 `copy` 函数冲突的类继承关系, 在 Kotlin 1.2 中已被废弃, 会产生编译警告, 在 Kotlin 1.3 中将会变为编译错误.

已废弃的功能: 在枚举值内的嵌套类型

在枚举值内, 定义一个不是 `内部类(inner class)` 的嵌套类型, 这个功能已被废弃了. 因为会导致初始化逻辑中的错误. 在 Kotlin 1.2 中会产生编译警告, 在 Kotlin 1.3 中将会变为编译错误.

已废弃的功能: 以命名参数的方式对 vararg 参数传递单个值

我们已经支持了注解中的数组参数字面值, 为了保持统一, 以命名参数的方式对 vararg 参数传递单个值 (`foo(items = i)`) 的功能已被废弃了. 请使用展开(spread)操作符和创建数组的工厂函数:

```
foo(items = *arrayOf(1))
```

此时编译器会优化代码, 删除多余的数组创建过程, 因此不会发生性能损失. 单个值形式的参数传递方式, 在 Kotlin 1.2 中会产生编译警告, 在 Kotlin 1.3 中将会不再支持.

已废弃的功能: 泛型类的内部类继承 Throwable

泛型类的内部类如果继承自 `Throwable`, 在 throw-catch 语句中可能会破坏类型安全性, 因此这个功能已被废弃, 在 Kotlin 1.2 中会产生编译警告, 在 Kotlin 1.3 中将会变为编译错误.

已废弃的功能: 改变只读属性的后端域变量的值

在自定义的取值方法中使用 `field = ...` 赋值语句, 改变只读属性的后端域变量的值, 这个功能已被废弃, 在 Kotlin 1.2 中会产生编译警告, 在 Kotlin 1.3 中将会变为编译错误.

标准库

Kotlin 标准库的 artifact 变更, 以及包分割问题

Kotlin 标准库现在开始完全兼容 Java 9 的模块系统(module system), Java 9 的模块系统禁止分割包(多个 jar 文件将类声明在同一个包之下). 为了支持这个功能, 引入了新的 artifact `kotlin-stdlib-jdk7` 和 `kotlin-stdlib-jdk8`, 代替旧的 `kotlin-stdlib-jre7` 和 `kotlin-stdlib-jre8`.

新 artifact 中的声明, 从 Kotlin 的角度看位于相同的包之下, 但对于 Java 则在不同的包之下. 因此, 切换到新的 artifact 不需要对你的源代码做任何修改.

为兼容 Java 9 的模块系统还做了另一个变更, 就是删除了 `kotlin-reflect` 库中 `kotlin.reflect` 包下的废弃的声明. 如果你在使用这些声明, 你需要改为使用 `kotlin.reflect.full` 包下的声明, 这个包从 Kotlin 1.1 开始支持.

windowed, chunked, zipWithNext

对 `Iterable<T>`, `Sequence<T>`, 和 `CharSequence` 增加了新的扩展函数, 用来应对以下几种使用场景: 缓冲(buffering)或批处理(batch processing) (`chunked`), 滑动窗口(sliding window)和滑动平均值(sliding average)的计算 (`windowed`), 以及对子序列项目对的处理(`zipWithNext`):

```
fun main(args: Array<String>) {
//sampleStart
    val items = (1..9).map { it * it }

    val chunkedIntoLists = items.chunked(4)
    val points3d = items.chunked(3) { (x, y, z) -> Triple(x, y, z) }
    val windowed = items.windowed(4)
    val slidingAverage = items.windowed(4) { it.average() }
    val pairwiseDifferences = items.zipWithNext { a, b -> b - a }
```

```
//sampleEnd

    println("items: $items\n")

    println("chunked into lists: $chunkedIntoLists")
    println("3D points: $points3d")
    println("windowed by 4: $windowed")
    println("sliding average by 4: $slidingAverage")
    println("pairwise differences: $pairwiseDifferences")
}
```

fill, replaceAll, shuffle/shuffled

新增了一组扩展函数, 用于处理列表: 对 `MutableList` 增加了 `fill`, `replaceAll` 和 `shuffle`, 对只读的 `List` 增加了 `shuffled`:

```
fun main(args: Array<String>) {
//sampleStart
    val items = (1..5).toMutableList()

    items.shuffle()
    println("Shuffled items: $items")

    items.replaceAll { it * 2 }
    println("Items doubled: $items")

    items.fill(5)
    println("Items filled with 5: $items")
//sampleEnd
}
```

kotlin-stdlib 中的数学运算

为了满足开发者长期以来的需求, Kotlin 1.2 增加了 `kotlin.math` API 用于数学运算, 这组 API 对于 JVM 环境和 JS 环境是共通的, 包含以下内容:

- 常数: `PI` 和 `E`
- 三角函数: `cos`, `sin`, `tan` 以及它们的反函数: `acos`, `asin`, `atan`, `atan2`

- 双曲函数: `cosh`, `sinh`, `tanh` 以及它们的反函数: `acosh`, `asinh`, `atanh`
- 指数函数: `pow` (这是一个扩展函数), `sqrt`, `hypot`, `exp`, `expm1`
- 对数函数: `log`, `log2`, `log10`, `ln`, `ln1p`
- 舍入处理(Rounding):
  - `ceil`, `floor`, `truncate`, `round` (向最接近数字方向舍入模式(half to even)) 函数
  - `roundToInt`, `roundToLong` (half to integer 模式) 扩展函数
- 符号与绝对值:
  - `abs` 和 `sign` 函数
  - `absoluteValue` 和 `sign` 扩展属性
  - `withSign` 扩展函数
- 对两个数值的 `max` 和 `min` 操作
- 二进制表达:
  - `ulp` 扩展属性
  - `nextUp`, `nextDown`, `nextTowards` 扩展函数
  - `toBits`, `toRawBits`, `Double.fromBits` (这些函数在 `kotlin` 包之下)

对于 `Float` 类型参数, 也提供了完全相同的一组函数(但没有常数).

BigInteger 和 BigDecimal 类型的操作符和转换

Kotlin 1.2 增加了一组函数, 用于 `BigInteger` 和 `BigDecimal` 类型的操作, 以及通过其它数值类型来创建这两种类型. 这些函数是:

- `Int` 和 `Long` 类型的 `toBigInteger` 函数
- `Int`, `Long`, `Float`, `Double`, 和 `BigInteger` 类型的 `toBigDecimal`
- 算数运算函数和位运算函数:
  - 二元运算符 `+`, `-`, `*`, `/`, `%`, 以及中缀函数(infix function) `and`, `or`, `xor`, `shl`, `shr`
  - 一元运算符 `-`, `++`, `--`, 以及 `inv` 函数

浮点数到位(bit)的转换

增加了新的函数, 用于在 `Double` 和 `Float` 类型与它们的位表达(bit representation)之间进行相互转换:

- `toBits` 和 `toRawBits` 函数, 对 `Double` 返回 `Long` 类型, 对 `Float` 返回 `Int` 类型
- `Double.fromBits` 和 `Float.fromBits` 函数使用位表达来创建浮点数值

正规表达式(Regex)变成了可序列化的对象(serializable)

`kotlin.text.Regex` 类现在成为了 `Serializable`, 因此可以在对象序列化层级(serializable hierarchy)中使用正规表达式.

如果可能的话, Closeable.use 会调用 Throwable.addSuppressed

如果在某个异常发生之后, 在关闭资源时再次发生了异常, `Closeable.use` 函数会调用 `Throwable.addSuppressed`.

为了使这个功能有效, 你需要在依赖库中包含 `kotlin-stdlib-jdk7`.

JVM 环境(JVM Backend)

构造函数调用的正规化

从 1.0 版开始, Kotlin 就支持包含复杂控制流的表达式, 比如 try-catch 表达式, 以及内联函数调用. 按照 Java 虚拟机的规格定义, 这类代码是合法的. 不幸的是, 当调用构造函数时, 如果在参数中包含这类表达式, 某些字节码处理工具对这类代码的处理不够好.

对于这类字节码处理工具的使用者, 为了减轻这个问题, 我们新增了一个命令行编译器选项(`-Xnormalize-constructor-calls=MODE`), 用来告诉编译器, 使编译器对这类构造函数调用代码生成更加类似 Java 风格的字节码. 这里的 `MODE` 可以是以下几种选项之一:

- `disable` (默认值) – 使用与 Kotlin 1.0 和 1.1 相同的方式生成字节码.
- `enable` – 对构造函数调用代码, 生成 Java 风格的字节码. 这个选项可以改变类加载和初始化的顺序.
- `preserve-class-initialization` – 对构造函数调用代码, 生成 Java 风格的字节码, 保证类的初始化顺序是正确的. 这个选项可能会影响你的应用程序的整体性能; 如果你在多个类之间共享了复杂的状态信息, 并且在类的初始化过程中更新这些状态信息, 只有在这种情况下, 你才需要使用这个编译选项.

另一种"手工"的变通办法是, 把带有控制流的子表达式的值保存到变量中, 然后使用这些变量, 而不是在构造函数调用的参数中直接计算这些表达式. 这种做法的效果类似于 `-Xnormalize-constructor-calls=enable` 选项.

Java 默认方法(default method)调用

在 Kotlin 1.2 之前, 当编译目标为 JVM 1.6 时, 如果接口的成员函数覆盖 Java 默认方法(default method), 那么在调用超类方法时, 会产生一个警告: `Super calls to Java default methods are deprecated in JVM target 1.6. Recompile with '-jvm-target 1.8'`. 在 Kotlin 1.2 中, 这个警告变成了 **错误**, 因此这类代码必须在 JVM 1.8 上编译.

破坏性变更: 对于平台数据类型的 x.equals(null) 行为一致性

对一个平台数据类型(platform type), 如果它映射为 Java 基本类型(`Int!`, `Boolean!`, `Short!`, `Long!`, `Float!`, `Double!`, `Char!`), 对它调用 `x.equals(null)` 时, 如果 `x` 为 null, 会返回一个不正确的结果 `true`. 从 Kotlin 1.2 开始, 对一个值为 null 的平台数据类型调用 `x.equals(...)` 会 **抛出 NPE 异常** (但 `x == ...` 不会抛出异常).

如果想要回到 1.2 版以前的结果, 可以对编译器指定参数 `-Xno-exception-on-explicit-equals-for-boxed-null`.

破坏性变更: 对内联的扩展函数中值为 null 的平台数据类型变换的修正

对一个值为 null 的平台数据类型, 调用一个内联的扩展函数, 假如内联函数没有检查接受者是否为 null, 这时可能会导致 null 值被变换为其他代码. Kotlin 1.2 在调用端强制执行 null 值检查, 如果接受者为 null, 会抛出异常.

如果想要回到 1.2 版以前的结果, 可以对编译器指定一个回退参数 `-Xno-receiver-assertions`.

JavaScript 环境(JavaScript Backend)

默认支持 TypedArray

对 Java Script 有类型数组的支持, 可以将 Kotlin 基本类型数组, 比如 `IntArray`, `DoubleArray`, 翻译为 JavaScript 有类型数组 (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays), 这个功能以前需要通过选项打开, 现在已经默认开启了.

工具

把警告作为错误来处理

编译器现在提供了一个选项, 可以将所有的警告当作错误来处理. 方法是, 在命令行使用 `-Werror` 参数, 或者在 Gradle 编译脚本中添加以下代码:

```
compileKotlin {
    kotlinOptions.allWarningsAsErrors = true
}
```

Kotlin 1.1 版中的新功能

最终更新: 2024/09/10

发布日期: 2016/02/15

目录

- 协程(coroutine)
- 语言层的其他特性
- 标准库
- JVM 环境(JVM Backend)
- JavaScript 环境(JavaScript Backend)

JavaScript

从 Kotlin 1.1 开始, JavaScript 编译环境不再是实验性的功能了. 目前已支持 Kotlin 语言的所有功能, 而且有了很多新的工具, 可以实现与前端开发环境的集成. 关于这部分变化的详情, 请阅读下文.

协程(coroutine) (实验性功能)

Kotlin 1.1 中关键性的新特性就是 *协程(coroutine)*, 这个特性可以支持 `async`/`await`, `yield` 等等类似的编程模式. Kotlin 的设计特性是, 协程的运行由库来实现, 而不是语言的一部分, 因此你不会被局限到某个特定的编程模式, 或者某个特定的并发库.

一个协程实际上是一个轻量级的线程, 它可以被暂停, 然后在以后的某个时刻恢复运行. 协程的支持依赖于 *挂起函数(suspending function)* (*["代码重构, 抽取函数" in "协程的基本概念"]*): 对函数的调用有可能导致一个协程挂起(suspend), 要启动一个新的协程我们通常使用匿名的挂起函数 (也就是. 挂起 lambda 表达式).

我们来看一看 `async`/`await` 函数, 它们实现在一个外部库中, kotlinx.coroutines (https://github.com/kotlin/kotlinx.coroutines):

```
// 在后台线程池中执行代码
fun asyncOverlay() = async(CommonPool) {
    // 启动 2 个异步操作
```

```
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // 然后将这 2 个操作取得的图片进行叠加
    applyOverlay(original.await(), overlay.await())
}

// 在 UI 上下文(context)中启动新的协程
launch(UI) {
    // 等待异步的图片叠加操作完成
    val image = asyncOverlay().await()
    // 然后在 UI 中显示结果
    showImage(image)
}
```

在这个例子中, `async { ... }` 启动一个协程, 然后, 当我们调用 `await()` 时, 当协程等待的操作还在执行时, 协程的执行将被挂起, 然后, 当协程等待的操作执行完毕时, 协程将会恢复执行(可能会在一个不同的线程内).

`yield` 和 `yieldAll` 函数可以产生 *延迟生成的序列(lazily generated sequences)*, 标准库使用协程来支持这种功能. 在这类序列中, 当每个元素被取得之后， 产生序列元素的代码段会被暂停, 当请求下一个元素时, 代码的执行又会回复. 示例如下:

```
import kotlin.coroutines.experimental.*

fun main(args: Array<String>) {
  val seq = buildSequence {
    for (i in 1..5) {
        // 产生 i 的平方值
        yield(i * i)
    }
    // 产生一个整数值范围(Range)
    yieldAll(26..28)
  }

  // 打印值序列
  println(seq.toList())
}
```

你可以运行上面的代码, 并查看结果. 你可以修改代码, 然后再次运行, 看看结果如何!

关于这个功能的详情, 请参见 参考文档 (协程(Coroutine)) 以及 教程 (教程 - 协程与通道 (Channel)).

注意, 协程目前还是 **实验性功能**, 也就是说, 1.1 正式发布后, Kotlin 开发组不保证这个特性的向后兼容性(backwards compatibility).

语言层的其他特性

类型别名(Type alias)

类型别名(type alias)功能允许你为已经存在的数据类型定义一个不同的名称. 这个功能对于泛型类型非常有用, 比如集合, 对于函数类型也很有用. 下面是示例:

```
//sampleStart
typealias OscarWinners = Map<String, String>

fun countLaLaLand(oscarWinners: OscarWinners) =
        oscarWinners.count { it.value.contains("La La Land") }

// 注意类型名称 (初始名称 和 类型别名) 是可以互换的:
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
        oscarWinners["Best picture"] == "La La Land"
//sampleEnd

fun oscarWinners(): OscarWinners {
    return mapOf(
            "Best song" to "City of Stars (La La Land)",
            "Best actress" to "Emma Stone (La La Land)",
            "Best picture" to "Moonlight" /* ... */)
}

fun main(args: Array<String>) {
    val oscarWinners = oscarWinners()

    val laLaLandAwards = countLaLaLand(oscarWinners)
    println("LaLaLandAwards = $laLaLandAwards (in our small
example), but actually it's 6.")

    val laLaLandIsTheBestMovie =
checkLaLaLandIsTheBestMovie(oscarWinners)
```

```
    println("LaLaLandIsTheBestMovie = $laLaLandIsTheBestMovie")
}
```

关于这个功能的详情, 请参见 类型别名相关文档 (类型别名) 以及 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/type-aliases.md).

与对象实例绑定的可调用的引用

现在你可以使用 `::` 操作符来得到一个 成员的引用 ("函数引用(Function Reference)" in "反射"), 指向一个具体的对象实例的方法或属性. 从前这样的功能只能通过 lambda 表达式来实现. 下面是示例:

```
//sampleStart
val numberRegex = "\\d+".toRegex()
val numbers = listOf("abc", "123",
"456").filter(numberRegex::matches)
//sampleEnd

fun main(args: Array<String>) {
    println("Result is $numbers")
}
```

关于这个功能的详情, 请参见 参考文档 (反射) 以及 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/bound-callable-references.md).

封闭类(sealed class)与数据类(data class)

Kotlin 1.1 中删除了 Kotlin 1.0 中对封闭类(sealed class)与数据类(data class)的一些限制. 过去, 封闭类的子类只能声明为封闭类的内嵌类(nested class), 现在这一限制已经删除, 你可以在同一个源代码文件的顶级(top level)位置定义顶级封闭类(top-level sealed class)的子类. 数据类现在可以继承自其它类. 这些功能可以用来更好、更清晰地定义表达式类的层次结构:

```
//sampleStart
sealed class Expr

data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when (expr) {
```

```
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
}
val e = eval(Sum(Const(1.0), Const(2.0)))
//sampleEnd

fun main(args: Array<String>) {
    println("e is $e") // 3.0
}
```

关于这个功能的详情, 请参见 封闭类相关文档 (封闭类(Sealed Class)与封闭接口(Sealed Interface)), 或参见 封闭类(sealed class) (https://github.com/Kotlin/KEEP/blob/master/proposals/sealed-class-inheritance.md) 以及 数据类(data class) (https://github.com/Kotlin/KEEP/blob/master/proposals/data-class-inheritance.md) 的 KEEP 文档.

在 lambda 表达式中使用解构声明

现在你可以使用 解构声明 (解构声明) 语法, 将对象解构为多个值, 然后作为参数传递给 lambda 表达式. 示例代码如下:

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf(1 to "one", 2 to "two")
    // 以前的编码方式:
    println(map.mapValues { entry ->
      val (key, value) = entry
      "$key -> $value!"
    })
    // 现在的编码方式:
    println(map.mapValues { (key, value) -> "$key -> $value!" })
//sampleEnd
}
```

关于这个功能的详情, 请参见 解构声明相关文档 (解构声明) 以及 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/destructuring-in-parameters.md).

使用下划线代替未使用的参数

对于接受多个参数的 lambda 表达式，你可以使用 `_` 来代替你不使用的参数:

```
fun main(args: Array<String>) {
    val map = mapOf(1 to "one", 2 to "two")

//sampleStart
    map.forEach { _, value -> println("$value!") }
//sampleEnd
}
```

这个功能对于 解构声明 (解构声明) 同样有效:

```
data class Result(val value: Any, val status: String)

fun getResult() = Result(42, "ok").also { println("getResult()
returns $it") }

fun main(args: Array<String>) {
//sampleStart
    val (_, status) = getResult()
//sampleEnd
    println("status is '$status'")
}
```

关于这个功能的详情，请参见 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/underscore-for-unused-parameters.md).

在数字字面值中使用下划线

与 Java 8 一样，Kotlin 现在也允许在数字字面值中使用下划线，将数字分隔为多个部分，以便阅读:

```
//sampleStart
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
//sampleEnd

fun main(args: Array<String>) {
```

```
    println(oneMillion)
    println(hexBytes.toString(16))
    println(bytes.toString(2))
}
```

关于这个功能的详情, 请参见 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/underscores-in-numeric-literals.md).

更加简短的属性语法

如果一个属性的取值方法的函数体是一个表达式, 属性类型现在可以省略:

```
//sampleStart
    data class Person(val name: String, val age: Int) {
        val isAdult get() = age >= 20 // 属性类型自动推断为 'Boolean'
    }
//sampleEnd
fun main(args: Array<String>) {
    val akari = Person("Akari", 26)
    println("$akari.isAdult = ${akari.isAdult}")
}
```

内联的属性访问函数

如果属性不存在后端域变量(backing field), 那么你可以使用 `inline` 修饰符来标记属性的访问器方法. 这样的访问器方法将会以 内联函数 (内联函数(Inline Function)) 相同的方式来编译.

```
//sampleStart
public val <T> List<T>.lastIndex: Int
    inline get() = this.size - 1
//sampleEnd

fun main(args: Array<String>) {
    val list = listOf('a', 'b')
    // 取值方法将被内联
    println("Last index of $list is ${list.lastIndex}")
}
```

你也可以将整个属性标记为 `inline` - 这时 `inline` 修饰符将被同时应用于取值方法和设值方法.

关于这个功能的详情, 请参见 内联函数相关文档 ("内联属性(Inline property)" in "内联函数(Inline Function)") 以及 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/inline-properties.md).

局部的委托属性

你现在可以对局部变量使用 委托属性 (委托属性) 语法. 这个功能可以用来定义一个延迟计算的局部变量:

```
import java.util.Random

fun needAnswer() = Random().nextBoolean()

fun main(args: Array<String>) {
//sampleStart
    val answer by lazy {
        println("Calculating the answer...")
        42
    }
    if (needAnswer()) {                     // 返回随机的布尔值
        println("The answer is $answer.")   // 答案将在这里计算
    }
    else {
        println("Sometimes no answer is the answer...")
    }
//sampleEnd
}
```

关于这个功能的详情, 请参见 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/local-delegated-properties.md).

委托属性绑定的拦截

对于 委托属性 (委托属性), 现在可以使用 `provideDelegate` 操作符来拦截委托到属性的绑定. 比如, 如果我们希望在绑定之前检查属性名称, 我们可以编写以下代码:

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, prop: KProperty<*>):
ReadOnlyProperty<MyUI, T> {
```

```
        checkProperty(thisRef, prop.name)
        ... // 属性创建
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

在 `MyUI` 实例的创建过程中, 对每一个属性都会调用 `provideDelegate` 方法, 因此这个方法可以在此时进行必要的验证处理.

关于这个功能的详情, 请参见 参考文档 (委托属性).

枚举值访问的通用方式

现在可以使用泛型方式来列举一个枚举类(enum class)的所有值.

```
//sampleStart
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}
//sampleEnd

fun main(args: Array<String>) {
    printAllValues<RGB>() // 打印结果为 RED, GREEN, BLUE
}
```

对 DSL 中的隐含接受者, 控制其范围

`@DslMarker` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-dsl-marker/index.html) 注解 可以限制从 DSL 上下文的外部范围(outer scope)来访问接受者. 比如, 考虑一下我们经典的 HTML 构建器的例子 (类型安全的构建器):

```
table {
    tr {
        td { + "Text" }
    }
}
```

在 Kotlin 1.0 中, 传递给 td 的那个 lambda 表达式中的代码, 可以访问 3 个隐含的接受者: 分别是 table 的接受者, tr 的接受者, 以及 td 的接受者. 这就导致你可以访问在当前上下文中毫无意义的方法 - 比如可以在 td 之内调用 tr, 因此可以在 <td> 之内再放置一个 <tr> 标记.

在 Kotlin 1.1 中, 你可以限制对这些接收者的访问, 因此, 在传递给 td 的那个 lambda 表达式中, 只有定义在 td 的隐含接收者中的方法才可以被调用. 要实现这一点, 你可以定义一个注解, 并用元注解(meta-annotation) @DslMarker 标注这个注解, 然后将你的注解标记到 HTML tag 类的基类上.

关于这个功能的详情, 请参见 类型安全的构建器相关文档 (类型安全的构建器) 以及 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/scope-control-for-implicit-receivers.md).

rem 操作符

mod 操作符现在已被废弃, 改为使用 rem 操作符. 关于这个变更的原因, 请参见 这个问题 (https://youtrack.jetbrains.com/issue/KT-14650).

标准库

字符串到数值的转换

对于 String 类, 新增了许多扩展函数, 用来将字符串转换为数值, 并且对不正确的数值不会抛出异常: String.toIntOrNull(): Int?, String.toDoubleOrNull(): Double? 等等.

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

同样也增加了整数的转换函数, 比如 Int.toString(), String.toInt(), String.toIntOrNull(), 这些函数都有带 radix 参数的重载版本, 这个参数可用来指定转换时使用的底数(base)(允许使用的底数为 2 到 36 之间).

onEach()

对于集合和序列来说, onEach 是一个小的, 但非常有用的扩展函数, 这个函数可以对集合或序列中的所有元素来执行相同的操作, 这个操作可能会带有副作用(side effect). 这个函数能够以操作链(chain of operation)的形式来使用. 对于 iterable, 这个函数类似 forEach, 但它最后会返回这个

iterable 实例. 对于 sequence, 这个函数会返回一个包装过的 sequence, 这个包装过的 sequence 会延迟地对每个元素执行你给定的操作.

```
inputDir.walk()
        .filter { it.isFile && it.name.endsWith(".txt") }
        .onEach { println("Moving $it to $outputDir") }
        .forEach { moveFile(it, File(outputDir,
it.toRelativeString(inputDir))) }
```

also(), takeIf() 和 takeUnless()

新增了3个多用途的扩展函数, 可以用于任意类型的接受者.

`also` 函数类似于 `apply`: 它得到一个接受者, 对它执行某种操作, 然后返回这个接受者. 区别在于, 在 `apply` 的代码段内部, 接受者可以通过 `this` 得到, 而在 `also` 的代码段内部, 接受者是 `it` (而且如果你愿意, 也可以指定其他名称). 如果你不希望其他范围内的 `this` 被屏蔽掉, 那么这个功能就很方便了:

```
class Block {
    lateinit var content: String
}

//sampleStart
fun Block.copy() = Block().also {
    it.content = this.content
}
//sampleEnd

// 改为使用 'apply'
fun Block.copy1() = Block().apply {
    this.content = this@copy1.content
}

fun main(args: Array<String>) {
    val block = Block().apply { content = "content" }
    val copy = block.copy()
    println("Testing the content was copied:")
    println(block.content == copy.content)
}
```

`takeIf` 函数类似于 `filter`, 但适用于单个值. 这个函数首先检查接受者是否符合某些条件, 如果满足条件则返回接受者, 否则返回 `null`. 将这个函数与 Elvis 操作符, 以及快速返回(early return)组合起来, 可以编写下面这样的代码:

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?:
return false
// 对于已经存在的 outDirFile 进行某些处理
```

```
fun main(args: Array<String>) {
    val input = "Kotlin"
    val keyword = "in"

//sampleStart
    val index = input.indexOf(keyword).takeIf { it >= 0 } ?:
error("keyword not found")
    // 在 input 字符串中查找 keyword 子串, 如果找到, 对 keyword 在 input
内的 index 位置进行某些处理
//sampleEnd

    println("'$keyword' was found in '$input'")
    println(input)
    println(" ".repeat(index) + "^")
}
```

`takeUnless` 与 `takeIf` 类似, 但它使用相反的判断条件. 如果 不满足条件则返回接受者, 否则返回 `null`. 因此上面的示例可以使用 `takeUnless` 改写, 如下:

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?:
error("keyword not found")
```

对于可执行的方法引用而不是 lambda 表达式, 这个函数也是非常便利的:

```
private fun testTakeUnless(string: String) {
//sampleStart
    val result = string.takeUnless(String::isEmpty)
//sampleEnd

    println("string = \"$string\"; result = \"$result\"")
```

```
}

fun main(args: Array<String>) {
    testTakeUnless("")
    testTakeUnless("abc")
}
```

groupingBy()

这个 API 可以用来对一个集合按照某个 key 进行分组, 并同时合并所有的组. 比如, 可以用来计算一段文字中以各个字母开头的单词数量:

```
fun main(args: Array<String>) {
    val words = "one two three four five six seven eight nine
ten".split(' ')
//sampleStart
    val frequencies = words.groupingBy { it.first() }.eachCount()
//sampleEnd
    println("Counting first letters: $frequencies.")

    // 另一种方式是使用 'groupBy' 和 'mapValues' 来创建一个中间 map,
    // 而 'groupingBy' 方式则是直接进行计数.
    val groupBy = words.groupBy { it.first() }.mapValues { (_, list)
-> list.size }
    println("Comparing the result with using 'groupBy': ${groupBy ==
frequencies}.")
}
```

Map.toMap() 和 Map.toMutableMap()

这两个函数可以用来简化 Map 的复制处理:

```
class ImmutablePropertyBag(map: Map<String, Any>) {
    private val mapCopy = map.toMap()
}
```

Map.minus(key)

`plus` 操作符提供了一个方法, 可以将键-值对(key-value pair)添加到一个只读的 map, 构造出一个新的 map, 但是没有简单的办法进行相反的操作: 为了从 map 中删除一个 key, 你必须使用不那么

直观的办法, 比如使用 Map.filter() 或 Map.filterKeys(). 现在, minus 操作符解决了这个问题. 这个操作符有 4 个重载版本: 删除单个 key, 删除 key 的集合, 删除 key 的序列, 以及删除 key 的数组.

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf("key" to 42)
    val emptyMap = map - "key"
//sampleEnd

    println("map: $map")
    println("emptyMap: $emptyMap")
}
```

minOf() 和 maxOf()

这些函数可用于在2个或3个给定的值中查找最小值和最大值, 查找对象必须是原始类型的数值, 或者是 Comparable 对象. 这些函数还有一个重载版本, 可以接受一个额外的 Comparator 实例作为参数, 如果你希望比较的对象值不是 Comparable 对象, 可以使用这个参数来指定如何比较.

```
fun main(args: Array<String>) {
//sampleStart
    val list1 = listOf("a", "b")
    val list2 = listOf("x", "y", "z")
    val minSize = minOf(list1.size, list2.size)
    val longestList = maxOf(list1, list2, compareBy { it.size })
//sampleEnd

    println("minSize = $minSize")
    println("longestList = $longestList")
}
```

类似数组风格的 List 创建函数

与 Array 的参见函数类似, 现在新增了用来创建 List 和 MutableList 实例的函数, 并且会通过调用 lambda 表达式来初始化列表中的元素:

```
fun main(args: Array<String>) {
//sampleStart
    val squares = List(10) { index -> index * index }
```

```
    val mutable = MutableList(10) { 0 }
//sampleEnd

    println("squares: $squares")
    println("mutable: $mutable")
}
```

Map.getValue()

`Map` 的这个扩展函数会接受一个 key 作为参数, 如果这个 key 对应的值已经存在, 则返回这个值, 否则抛出一个异常, 表示没有找到这个 key. 如果 Map 在创建时使用了 `withDefault`, 那么对于未找到的 key, 这个函数将会返回默认值, 而不会抛出异常.

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf("key" to 42)
    // 返回不可为 null 的 Int 值 42
    val value: Int = map.getValue("key")

    val mapWithDefault = map.withDefault { k -> k.length }
    // 返回 4
    val value2 = mapWithDefault.getValue("key2")

    // map.getValue("anotherKey") // <- 这个调用将抛出
NoSuchElementException 异常
//sampleEnd

    println("value is $value")
    println("value2 is $value2")
}
```

抽象的集合类

实现 Kotlin 集合类时, 可以使用这些抽象类作为基类. 为了实现只读集合, 可以使用的基类有 `AbstractCollection`, `AbstractList`, `AbstractSet` 以及 `AbstractMap`, 对于可变的集合, 可以使用的基类有 `AbstractMutableCollection`, `AbstractMutableList`, `AbstractMutableSet` 以及 `AbstractMutableMap`. 在 JVM 环境中, 这些可变集合的抽象类的大多数功能, 通过继承 JDK 的集合抽象类得到.

数组处理函数

标准库现在提供了一系列函数, 用于逐个元素的数组操作: 比较函数 (`contentEquals` 和 `contentDeepEquals`), hash code 计算函数 (`contentHashCode` 和 `contentDeepHashCode`), 以及字符串转换函数 (`contentToString` 和 `contentDeepToString`). 这些函数都支持 JVM (这时这些函数对应于 `java.util.Arrays` 中的各个函数), 也支持 JavaScript (由 Kotlin 提供实现).

```
fun main(args: Array<String>) {
//sampleStart
    val array = arrayOf("a", "b", "c")
    println(array.toString())  // 这里会输出JVM 的实现: 数组类型名称, 加
hash code
    println(array.contentToString())  // 这里会输出格式化良好的数组内容列
表
//sampleEnd
}
```

JVM 环境(JVM Backend)

对 Java 8 字节码的支持

Kotlin 现在增加了编译选项, 可以编译产生 Java 8 字节码(使用命令行选项 `-jvm-target 1.8`, 或 Ant/Maven/Gradle 中的对应选项). 这个选项目前不会改变字节码的语义(具体来说, 接口内的默认方法以及 lambda 表达式的编译输出方式会与 Kotlin 1.0 中完全相同), 但我们将来计划对这个选项做更多的改进.

对 Java 8 标准库的支持

Kotlin 的标准库目前存在不同的版本, 分别支持 Java 7 和 8 中新增的 JDK API. 如果你需要使用新的 API, 请不要使用标准的 Maven artifact `kotlin-stdlib`, 改用 `kotlin-stdlib-jre7` 和 `kotlin-stdlib-jre8`. 这些 artifact 在 `kotlin-stdlib` 之上进行了微小的扩展, 而且会将 `kotlin-stdlib` 以传递依赖的方式引入到你的项目中.

字节码中的参数名称

Kotlin 现在支持在字节码中保存参数名称. 可以使用命令行参数 `-java-parameters` 打开这个功能.

常数内联(Constant inlining)

编译器现在可以将 `const val` 属性的值内联到这些属性被使用的地方.

可变的闭包变量(Mutable closure variable)

用于捕获 lambda 中的可变的闭包变量的封装类(box class) 不再拥有可变的域变量. 这个变化改进了性能, 但在某些罕见的使用场景下, 可能会导致新的竞争条件(race condition). 如果你受到这个问题的影响, 那么你在访问这些变量时, 需要自行实现同步控制.

对 javax.script 的支持

Kotlin 目前集成了 javax.script API (https://docs.oracle.com/javase/8/docs/api/javax/script/package-summary.html) (JSR-223). 这个 API 可以在运行期执行代码片段:

```
val engine = ScriptEngineManager().getEngineByExtension("kts")!!
engine.eval("val x = 3")
println(engine.eval("x + 2"))  // 输出结果为: 5
```

这里 (https://github.com/JetBrains/kotlin/tree/master/libraries/examples/kotlin-jsr223-local-example) 是使用这个 API 的一个更详细的示例工程.

kotlin.reflect.full

作为 支持 Java 9 的准备工作 (https://blog.jetbrains.com/kotlin/2017/01/kotlin-1-1-whats-coming-in-the-standard-library/), `kotlin-reflect.jar` 库中的扩展函数和扩属性已被移动到 `kotlin.reflect.full` 包内. 旧包 (`kotlin.reflect`) 内的名称已被标记为废弃, 并且将在 Kotlin 1.2 中删除. 注意, 反射功能的核心接口(比如 `KClass`) 是 Kotlin 标准库的一部分, 而不是 `kotlin-reflect` 的一部分, 因此不受此次包移动的影响.

JavaScript 环境(JavaScript Backend)

统一的标准库

编译为 JavaScript 的 Kotlin 代码, 现在可以访问 Kotlin 标准库中更多的部分了. 具体来说, 许多关键性的类, 比如集合(`ArrayList`, `HashMap` 等等.), 异常(`IllegalArgumentException` 等等.) 以及其他一些类(`StringBuilder`, `Comparator`) 现在被定义在 `kotlin` 包之下. 在 JVM 环境中, 这些名称是指向对应的 JDK 类的类型别名, 在 JS 环境中, 这些类在 Kotlin 标准库中实现.

更好的代码生成能力

JavaScript 环境生成的代码现在更容易进行静态检查了, 因此对于 JS 的代码处理工具更加友好, 比如代码压缩器(minifier), 优化器(optimiser), 校验检查器(linter), 等等.

external 修饰符

如果你需要在 Kotlin 中以类型安全的方式来访问一个 JavaScript 中实现的类, 你可以使用 `external` 修饰符编写一个 Kotlin 声明. (在 Kotlin 1.0 中, 使用的是 `@native` 注解.) 与 JVM 编译对象不同, JS 编译对象允许对类和属性使用 `external` 修饰符. 比如, 你可以这样声明 DOM 的 `Node` 类:

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // 等等
}
```

import 处理的改进

现在你可以更加精确地指定需要从 JavaScript 模块中导入哪些声明. 如果你将 `@JsModule("<module-name>")` 注解添加到一个外部声明上, 那么在编译过程中它就会被正确地导入模块系统中(无论是 CommonJS 还是 AMD). 比如, 在 CommonJS 中, 这个声明将会通过 `require(...)` 函数导入. 此外, 如果你希望导入一个声明, 无论是作为一个模块还是作为一个全局 JavaScript 对象, 你都可以使用 `@JsNonModule` 注解.

比如, 你可以这样将 JQuery 导入到 Kotlin 模块中:

```
external interface JQuery {
    fun toggle(duration: Int = definedExternally): JQuery
    fun click(handler: (Event) -> Unit): JQuery
}

@JsModule("jquery")
@JsNonModule
@JsName("$")
external fun jquery(selector: String): JQuery
```

在这段示例代码中, JQuery 将会导入为一个模块, 模块名称是 `jquery`. 或者, 也可以作为一个 $-对象来使用, 具体如何, 取决于 Kotlin 编译器被设置为使用哪种模块系统.

在你的应用程序中, 你可以这样使用这些声明:

```
fun main(args: Array<String>) {
    jquery(".toggle-button").click {
        jquery(".toggle-panel").toggle(300)
    }
}
```

Kotlin 的发布版本

最终更新: 2024/09/10

我们的发布版本包括几种不同的类型:

- *功能发布版(Feature release)* (1. *x*) 其中包括语言的重要变化.
- *增量发布版(Incremental release)* (1. *x*. *y*) 在功能发布版之间发布, 其中包括工具更新, 性能改善, 以及 bug 修复.
- *Bug 修复发布版(Bug fix release)* (1. *x*. *yz*) 其中包括针对增量发布版的 bug 修复.

比如, 对于功能发布版 1.3, 我们有几个增量发布版, 包括 1.3.10, 1.3.20, 以及 1.3.70. 对于 1.3.70, 我们有 2 个 bug 修复发布版 – 1.3.71 和 1.3.72.

对于每个增量发布版和功能发布版, 我们还会发布几个预览 (*EAP*) 版, 供开发者在正式发布之前试用新功能. 详情请参见 早期预览(Early Access Preview) (参加 Kotlin EAP 项目).

关于 Kotlin 的发布版本, 详情请参见 Kotlin 发布版的类型, 以及它们的兼容性 ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化").

更新到新的发布版

> 从 IntelliJ IDEA 2023.3 和 Android Studio Iguana (2023.2.1) Canary 15 开始, Kotlin plugin 会自动更新. 你只需要在你的项目中更新 Kotlin 版本.

新的发布版发布之后, IntelliJ IDEA 和 Android Studio 会建议你升级. 如果你接受建议, IDE 会自动将 Kotlin 插件更新到最新版本. 在 **Tools** | **Kotlin** | **Configure Kotlin Plugin Updates** 菜单中, 你可以选择 Kotlin 版本.

如果你的项目创建时使用了较早的 Kotlin 版本, 那么有可能需要在你的项目中改变 Kotlin 版本, 并更新 kotlinx 库.

如果你要迁移到新的功能发布版, Kotlin 插件的迁移工具可以帮助你进行迁移.

IDE 支持

以下版本的 IntelliJ IDEA 和 Android Studio 支持 Kotlin 语言最新版本:

* IntelliJ IDEA:
  * 最新的稳定版本
  * 前一个稳定版本
  * 早期预览 (https://www.jetbrains.com/resources/eap/) 版
* Android Studio:
  * 最新发布版 (https://developer.android.com/studio)
  * 早期预览版 (https://developer.android.com/studio/preview)

> 关于 IntelliJ IDEA 中 Kotlin 相关的最新更新, 请参见 IntelliJ IDEA 最新功能 (https://www.jetbrains.com/idea/whatsnew/) 的 **Kotlin** 小节.

各发布版详情

下表是 Kotlin 最新发布版的详情.

你也可以使用 Kotlin 的预览版 ("EAP 版本" in "参加 Kotlin EAP 项目").

构建信息	主要内容
1.9.23 发布日期: **2024/03/07** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.23)	针对 Kotlin 1.9.20, 1.9.21, 和 1.9.22 的 Bug 修复发布版. 关于 Kotlin 1.9.20, 请参见 Kotlin 1.9.20 版中的新功能 (Kotlin 1.9.20 版中的新功能).
1.9.22 发布日期: **2023/12/21** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.22)	针对 Kotlin 1.9.20 和 1.9.21 的 Bug 修复发布版. 关于 Kotlin 1.9.20, 请参见 Kotlin 1.9.20 版中的新功能 (Kotlin 1.9.20 版中的新功能).
1.9.21 发布日期: **2023/11/23** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.21)	针对 Kotlin 1.9.20 的 Bug 修复发布版. 关于 Kotlin 1.9.20, 请参见 Kotlin 1.9.20 版中的新功能 (Kotlin 1.9.20 版中的新功能).
1.9.20 发布日期: **2023/11/01**	一个新功能发布版, 包括 Kotlin K2 编译器的 Beta 版, 和 Kotlin Multiplatform 的稳定版. 详情请参见:

GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.20)	• Kotlin 1.9.20 版中的新功能 (Kotlin 1.9.20 版中的新功能)
1.9.10 发布日期: **2023/08/23** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.10)	针对 Kotlin 1.9.0 的 Bug 修复发布版. 关于 Kotlin 1.9.0, 请参见 Kotlin 1.9.0 版中的新功能 (Kotlin 1.9.0 版中的新功能). 对于 Android Studio 的 Giraffe 和 Hedgehog 版, Kotlin plugin 1.9.10 会在之后的 Android Studio 更新中发布.
1.9.0 发布日期: **2023/07/06** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.0)	一个新功能发布版, 包括 Kotlin K2 编译器的更新, 新的枚举类值函数, 用于终端开放(open-ended)的值范围的新的操作符, Kotlin Multiplatform 中的 Gradle 配置缓存功能的预览版, Kotlin Multiplatform 中支持的 Android target 的变更, Kotlin/Native 中自定义内存分配器功能的预览版. 详情请参见: • Kotlin 1.9.0 版中的新功能 (Kotlin 1.9.0 版中的新功能) • YouTube 视频: Kotlin 的新功能 (https://www.youtube.com/embed/fvwTZc-dxsM)
1.8.22 发布日期: **2023/06/08** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.8.22)	针对 Kotlin 1.8.20 的 Bug 修复发布版. 关于 Kotlin 1.8.20, 请参见 Kotlin 1.8.20 版中的新功能 (Kotlin 1.8.20 版中的新功能).

1.8.21 发布日期: **2023/04/25** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.8.21)	针对 Kotlin 1.8.20 的 Bug 修复发布版. 关于 Kotlin 1.8.20, 请参见 Kotlin 1.8.20 版中的新功能 (Kotlin 1.8.20 版中的新功能). 对于 Android Studio 的 Flamingo 和 Giraffe 版, Kotlin plugin 1.8.21 会在之后的 Android Studio 更新中发布.
1.8.20 发布日期: **2023/04/03** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.8.20)	一个新功能发布版, 包括 Kotlin K2 编译器的更新, 标准库中的 AutoCloseable 接口和 Base64 编码, 默认启用新的 JVM 增量编译, 新的 Kotlin/Wasm 编译器后端. 详情请参见: • Kotlin 1.8.20 版中的新功能 (Kotlin 1.8.20 版中的新功能) • YouTube 视频: Kotlin 的新功能 (https://youtu.be/R1JpkpPzyBU)
1.8.10 发布日期: **2023/02/02** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.8.10)	针对 Kotlin 1.8.0 的 Bug 修复发布版. 详情请参见 Kotlin 1.8.0 (https://github.com/JetBrains/kotlin/releases/tag/v1.8.0). 对于 Android Studio 的 Electric Eel 和 Flamingo 版, Kotlin plugin 1.8.10 会在之后的 Android Studio 更新中发布.
1.8.0 发布日期: **2022/12/28** GitHub 发布链接 (https://github.com/JetBrains/kotli	一个新功能发布版, 包括 kotlin-reflect 的性能改善, JVM 平台的新功能: 目录内容的递归复制或递归删除 (实验性功能), 与 Objective-C/Swift 交互功能的改进. 详情请参见: • Kotlin 1.8.0 版中的新功能 (Kotlin 1.8.0 版中的新功能)

n/releases/tag/v1.8.0)	• Kotlin 1.8 兼容性指南 (Kotlin 1.8 兼容性指南)
1.7.21 发布日期: **2022/11/09** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.7.21)	针对 Kotlin 1.7.20 的 Bug 修复发布版. 关于 Kotlin 1.7.20, 请参见 Kotlin 1.7.20 版中的新功能 (Kotlin 1.7.20 版中的新功能). 对于 Android Studio 的 Dolphin, Electric Eel, 以及 Flamingo 版, Kotlin plugin 1.7.21 会在之后的 Android Studio 更新中发布.
1.7.20 发布日期: **2022/09/29** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.7.20)	增量发布版, 包含新的语言语言特性, 在 Kotlin K2 编译器中支持几种编译器插件, 默认启用新的 Kotlin/Native 内存管理器, 以及支持 Gradle 7.1. 详情请参见: • Kotlin 1.7.20 版中的新功能 (Kotlin 1.7.20 版中的新功能) • YouTube 视频: Kotlin 的新功能 (https://youtu.be/OG9npowJgE8) • Kotlin 1.7.20 兼容性指南 (Kotlin 1.7.20 兼容性指南) 详情请参见 Kotlin 1.7.20 (https://github.com/JetBrains/kotlin/releases/tag/v1.7.20).
1.7.10 发布日期: **2022/07/07** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.7.10)	针对 Kotlin 1.7.0 的 Bug 修复发布版. 详情请参见 Kotlin 1.7.0 (https://github.com/JetBrains/kotlin/releases/tag/v1.7.0). 对于 Android Studio Dolphin (213) 和 Android Studio Electric Eel (221), Kotlin plugin 1.7.10 会在之后的 Android Studio 更新中发布.
1.7.0	一个新功能发布版, 包含 JVM 平台的 Kotlin K2 编译器(Alpha 版), 稳定

发布日期: **2022/06/09** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.7.0)	版的语言特性, 性能改善, 以及演进变化, 比如实验性 API 进入稳定状态. 详情请参见: • Kotlin 1.7.0 的新功能 (Kotlin 1.7.0 版中的新功能) • YouTube 视频: Kotlin 的新功能 (https://youtu.be/54WEfLKtCGk) • Kotlin 1.7.0 兼容性指南 (Kotlin 1.7 兼容性指南)
1.6.21 发布日期: **2022/04/20** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.6.21)	针对 Kotlin 1.6.20 的 Bug 修复发布版. 详情请参见 Kotlin 1.6.20 (Kotlin 1.6.20 版中的新功能).
1.6.20 发布日期: **2022/04/04** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.6.20)	增量发布版, 包含各种改进, 比如: • 上下文接受者(Context Receiver)的原型 • 对函数式接口构造器的可调用引用 • Kotlin/Native: 新的内存管理器的性能改善 • Multiplatform: 默认使用层级项目结构(Hierarchical Project Structure) • Kotlin/JS: IR 编译器改进 • Gradle: 编译器执行策略 详情请参见 Kotlin 1.6.20 (Kotlin 1.6.20 版中的新功能).

1.6.10 发布日期: **2021/12/14** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.6.10)	针对 Kotlin 1.6.0 的 Bug 修复发布版. 详情请参见 Kotlin 1.6.0 (https://github.com/JetBrains/kotlin/releases/tag/v1.6.0).
1.6.0 发布日期: **2021/11/16** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.6.0)	一个新功能发布版, 包含新的语言特性, 性能改善, 以及演进变化, 比如实验性 API 进入稳定状态. 详情请参见: • 关于新版本发布的 Blog (https://blog.jetbrains.com/kotlin/2021/11/kotlin-1-6-0-is-released/) • Kotlin 1.6.0 的新功能 (Kotlin 1.6.0 版中的新功能) • Kotlin 1.6.0 兼容性指南 (Kotlin 1.6 兼容性指南)
1.5.32 发布日期: **2021/11/29** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.5.32)	针对 Kotlin 1.5.31 的 Bug 修复发布版. 详情请参见 Kotlin 1.5.30 (Kotlin 1.5.30 版中的新功能).
1.5.31 发布日期: **2021/09/20** GitHub 发布链接 (https://github.co	针对 Kotlin 1.5.30 的 Bug 修复发布版. 详情请参见 Kotlin 1.5.30 (Kotlin 1.5.30 版中的新功能).

m/JetBrains/kotlin/releases/tag/v1.5.31)	
1.5.30 发布日期: **2021/08/23** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.5.30)	增量发布版, 包含各种改进, 比如: • JVM 平台上, 注解类的实例创建 • 改进 opt-in 要求机制和类型推断 • Kotlin/JS IR 后端进入 Beta 版 • 支持 Apple Silicon 编译目标 • 改进对 CocoaPods 的支持 • Gradle: Java 工具链的支持, 并改进 daemon 配置 详情请参见: • 关于新版本发布的 Blog (https://blog.jetbrains.com/kotlin/2021/08/kotlin-1-5-30-released/) • Kotlin 1.5.30 的新功能 (Kotlin 1.5.30 版中的新功能)
1.5.21 发布日期: **2021/07/13** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.5.21)	针对 Kotlin 1.5.20 的 Bug 修复发布版. 详情请参见 Kotlin 1.5.20 (Kotlin 1.5.20 版中的新功能).
1.5.20 发布日期: **2021/06/24**	增量发布版, 包含各种改进, 比如: • 在 JVM 平台 默认使用 `invokedynamic` 实现字符串拼接

GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.5.20)	• 改进对 Lombok 和 JSpecify 的支持 • Kotlin/Native: KDoc 导出 Objective-C 头文件, 改进在同一数组内 `Array.copyInto()` 的速度 • Gradle: 注解处理器的类装载器缓存, 支持 Gradle 的 `--parallel` 属性 • 标准库函数在各个平台的动作保持一致 详情请参见: • 关于新版本发布的 Blog (https://blog.jetbrains.com/kotlin/2021/06/kotlin-1-5-20-released/) • Kotlin 1.5.20 的新功能 (Kotlin 1.5.20 版中的新功能)
1.5.10 发布日期: **2021/05/24** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.5.10)	针对 Kotlin 1.5.0 的 Bug 修复发布版. 详情请参见 Kotlin 1.5.0 (https://blog.jetbrains.com/kotlin/2021/04/kotlin-1-5-0-released/).
1.5.0 发布日期: **2021/05/05** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.5.0)	一个新功能发布版, 包含新的语言特性, 性能改善, 以及演进变化, 比如实验性 API 进入稳定状态. 详情请参见: • 关于新版本发布的 Blog (https://blog.jetbrains.com/kotlin/2021/04/kotlin-1-5-0-released/) • Kotlin 1.5.0 的新功能 (Kotlin 1.5.0 版中的新功能) • Kotlin 1.5.0 兼容性指南 (Kotlin 1.5 兼容性指南)
1.4.32	针对 Kotlin 1.4.30 的 Bug 修复发布版.

发布日期: **2021/03/22** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.4.32)	详情请参见 Kotlin 1.4.30 (Kotlin 1.4.30 版中的新功能).
1.4.31 发布日期: **2021/02/25** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.4.31)	针对 Kotlin 1.4.30 的 Bug 修复发布版. 详情请参见 Kotlin 1.4.30 (Kotlin 1.4.30 版中的新功能).
1.4.30 发布日期: **2021/02/03** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.4.30)	增量发布版, 包含各种改进, 比如: • 新的 JVM 后端 进入 Beta 版 • 新语言特性的预览 • Kotlin/Native 的性能改进 • 标准库 API 改进 详情请参见: • 关于新版本发布的 Blog (https://blog.jetbrains.com/kotlin/2021/01/kotlin-1-4-30-released/) • Kotlin 1.4.30 的新功能 (Kotlin 1.4.30 版中的新功能)
1.4.21 发布日期: **2020/1**	针对 Kotlin 1.4.20 的 Bug 修复发布版 详情请参见 Kotlin 1.4.20 (Kotlin 1.4.20 版中的新功能).

2/07 GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.4.21)	
1.4.20 发布日期: **2020/11/23** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.4.20)	增量发布版, 包含各种改进, 比如: • 支持新的 JVM 功能特性, 比如通过 `invokedynamic` 拼接字符串 • 对 Kotlin Multiplatform Mobile 项目改进性能和异常处理 • 对 JDK 路径的扩展: `Path("dir") / "file.txt"` 详情请参见: • 关于新版本发布的 Blog (https://blog.jetbrains.com/kotlin/2020/11/kotlin-1-4-20-released/) • Kotlin 1.4.20 (Kotlin 1.4.20 版中的新功能)
1.4.10 发布日期: **2020/09/07** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.4.10)	针对 Kotlin 1.4.0 的 Bug 修复发布版. 详情请参见 Kotlin 1.4.0 (https://blog.jetbrains.com/kotlin/2020/08/kotlin-1-4-released-with-a-focus-on-quality-and-performance/).
1.4.0 发布日期: **2020/08/17** GitHub 发布链接 (https://github.co	一个新功能发布版, 包含很多新功能和改进, 主要集中于质量和性能. 详情请参见: • 关于新版本发布的 Blog (https://blog.jetbrains.com/kotlin/2020/08/kotlin-1-4-released-with-a-focus-on-quality-and-performanc

m/JetBrains/kotlin/releases/tag/v1.4.0)	e/) • Kotlin 1.4.0 的新功能 (Kotlin 1.4.0 版中的新功能) • Kotlin 1.4.0 兼容性指南 (Kotlin 1.4 兼容性指南) • 迁移到 Kotlin 1.4.0 ("迁移到 Kotlin 1.4.0" in "Kotlin 1.4.0 版中的新功能")
1.3.72 发布日期: **2020/04/15** GitHub 发布链接 (https://github.com/JetBrains/kotlin/releases/tag/v1.3.72)	针对 Kotlin 1.3.70 的 Bug 修复发布版. 详情请参见 Kotlin 1.3.70 (https://blog.jetbrains.com/kotlin/2020/03/kotlin-1-3-70-released/).

Kotlin 发展路线图

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/roadmap.html)

基本语法

最终更新: 2024/09/10

本章会通过示例程序向你介绍 Kotlin 的一系列基本语法元素. 在各节的末尾, 你可以找到各个专题详细信息的页面链接.

你也可以通过 JetBrains Academy 的免费 Kotlin 核心课程 (https://hyperskill.org/tracks?category=4&utm_source=jbkotlin_hs&utm_medium=referral&utm_campaign=kotlinlang-docs&utm_content=button_1&utm_term=22.03.23) 学习 Kotlin 的全部基本知识.

包的定义与导入

包的定义应该在源代码文件的最上方.

```
package my.demo

import kotlin.text.*

// ...
```

源代码所在的目录结构不必与包结构保持一致: 源代码文件可以放置在文件系统的任意位置.

参见 包 (包(Package)与导入(Import)).

程序入口点(entry point)

Kotlin 应用程序的入口点是 `main` 函数.

```
fun main() {
    println("Hello world!")
}
```

`main` 函数的另一种形式可以接受数量不定的 `String` 参数.

```
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

向标准输出(Standard Output)打印信息

`print` 函数会将传递给它的参数打印到标准输出(Standard Output).

```
fun main() {
//sampleStart
    print("Hello ")
    print("world!")
//sampleEnd
}
```

`println` 函数会打印它的参数, 并在末尾加上换行(Line Break), 因此之后的打印信息会出现在下一行.

```
fun main() {
//sampleStart
    println("Hello world!")
    println(42)
//sampleEnd
}
```

函数

以下函数接受两个 `Int` 类型参数, 并返回 `Int` 类型结果.

```
//sampleStart
fun sum(a: Int, b: Int): Int {
    return a + b
}
//sampleEnd

fun main() {
    print("sum of 3 and 5 is ")
    println(sum(3, 5))
}
```

以下函数使用表达式语句作为函数体, 返回类型由自动推断决定.

```
//sampleStart
fun sum(a: Int, b: Int) = a + b
//sampleEnd

fun main() {
    println("sum of 19 and 23 is ${sum(19, 23)}")
}
```

以下函数不返回有意义的结果.

```
//sampleStart
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

返回值为 `Unit` 类型时, 可以省略.

```
//sampleStart
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

参见 函数 (函数).

变量

只读的局部变量使用关键字 `val` 来定义, 它们只能赋值一次.

```
fun main() {
//sampleStart
    val a: Int = 1  // 立即赋值
    val b = 2   // 变量类型自动推断为 `Int` 类型
    val c: Int  // 没有初始化语句时，必须明确指定类型
    c = 3       // 延迟赋值
//sampleEnd
    println("a = $a, b = $b, c = $c")
}
```

可以多次赋值的变量使用关键字 `var` 来定义.

```
fun main() {
//sampleStart
    var x = 5 // 变量类型自动推断为 `Int` 类型
    x += 1
//sampleEnd
    println("x = $x")
}
```

也可以将变量声明在顶级(top level).

```
//sampleStart
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
//sampleEnd

fun main() {
    println("x = $x; PI = $PI")
    incrementX()
    println("incrementX()")
    println("x = $x; PI = $PI")
}
```

参见 属性(Property) (属性(Property)).

创建类与实例

要定义类, 请使用 `class` 关键字.

```
class Shape
```

类的属性(Property)可以在类声明部分或类主体部分中列出.

```
class Rectangle(val height: Double, val length: Double) {
    val perimeter = (height + length) * 2
}
```

会自动生成一个默认构造器, 参数是在类声明部分中定义的那些属性.

```
class Rectangle(val height: Double, val length: Double) {
    val perimeter = (height + length) * 2
}
fun main() {
//sampleStart
    val rectangle = Rectangle(5.0, 2.0)
    println("The perimeter is ${rectangle.perimeter}")
//sampleEnd
}
```

类之间的继承关系使用冒号(`:`)表示. 类默认为 final; 要允许一个类被后代继承, 请将它标记为 `open`.

```
open class Shape

class Rectangle(val height: Double, val length: Double): Shape() {
    val perimeter = (height + length) * 2
}
```

参见 类 (类) 和 对象与实例 (对象表达式,对象声明,以及同伴对象).

注释

与大多数现代编程语言一样, Kotlin 支持单行(或者叫做 *行尾*)注释, 也支持多行 (或者叫做 *块*) 注释.

```
// 这是一条行尾注释

/* 这是一条块注释
   可以包含多行内容. */
```

Kotlin 的块注释允许嵌套.

```
/* 注释从这里开始
/* 包含一个嵌套的注释 */
到这里结束. */
```

关于文档注释的语法, 详情请参见 Kotlin 代码中的文档 (为 Kotlin 代码编写文档: KDoc).

字符串模板

```
fun main() {
//sampleStart
    var a = 1
    // 在字符串模板内使用简单的变量名称
    val s1 = "a is $a"

    a = 2
    // 在字符串模板内使用任意的表达式:
    val s2 = "${s1.replace("is", "was")}, but now is $a"
//sampleEnd
    println(s2)
}
```

详情请参见 字符串模板 ("字符串模板" in "字符串").

条件表达式

```
//sampleStart
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
```

```
    } else {
        return b
    }
}
//sampleEnd

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

在 Kotlin 中, `if` 也可以用作表达式.

```
//sampleStart
fun maxOf(a: Int, b: Int) = if (a > b) a else b
//sampleEnd

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

参见 `if` 表达式 ("if 表达式" in "条件与循环").

for 循环

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

或者

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
```

```
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
//sampleEnd
}
```

参见 for 循环 ("for 循环" in "条件与循环").

while 循环

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    var index = 0
    while (index < items.size) {
        println("item at $index is ${items[index]}")
        index++
    }
//sampleEnd
}
```

参见 while 循环 ("while 循环" in "条件与循环").

when 表达式

```
//sampleStart
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
//sampleEnd

fun main() {
    println(describe(1))
```

```
    println(describe("Hello"))
    println(describe(1000L))
    println(describe(2))
    println(describe("other"))
}
```

参见 when 表达式 ("when 表达式" in "条件与循环").

值范围(Range)

使用 `in` 操作符检查一个数值是否在某个值范围(Range)之内:

```
fun main() {
//sampleStart
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
//sampleEnd
}
```

检查一个数值是否在某个值范围之外.

```
fun main() {
//sampleStart
    val list = listOf("a", "b", "c")

    if (-1 !in 0..list.lastIndex) {
        println("-1 is out of range")
    }
    if (list.size !in list.indices) {
        println("list size is out of valid list indices range, too")
    }
//sampleEnd
}
```

在一个值范围内进行遍历迭代.

```
fun main() {
//sampleStart
    for (x in 1..5) {
        print(x)
    }
//sampleEnd
}
```

在一个数列(progression)上进行遍历迭代.

```
fun main() {
//sampleStart
    for (x in 1..10 step 2) {
        print(x)
    }
    println()
    for (x in 9 downTo 0 step 3) {
        print(x)
    }
//sampleEnd
}
```

参见 值范围(Range)与数列(Progression) (值范围(Range)与数列(Progression)).

集合(Collection)

在一个集合上进行遍历迭代.

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
//sampleStart
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

使用 `in` 运算符检查一个集合是否包含某个对象.

```
fun main() {
    val items = setOf("apple", "banana", "kiwifruit")
//sampleStart
    when {
        "orange" in items -> println("juicy")
        "apple" in items -> println("apple is fine too")
    }
//sampleEnd
}
```

使用 Lambda 表达式 (高阶函数与 Lambda 表达式) 对集合元素进行过滤和变换:

```
fun main() {
//sampleStart
    val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
    fruits
      .filter { it.startsWith("a") }
      .sortedBy { it }
      .map { it.uppercase() }
      .forEach { println(it) }
//sampleEnd
}
```

参见 集合(Collection)概述 (集合(Collection)概述).

可为 null 的值 与 null 值检查

当一个引用可能为 `null` 值时, 对应的类型声明必须明确地标记为可为 null. 类型名称末尾带 `?` 符号表示可为 null 值.

当 `str` 中的字符串内容不是一个整数时, 返回 `null`:

```
fun parseInt(str: String): Int? {
    // ...
}
```

以下示例演示如何使用一个返回值可为 null 的函数:

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

//sampleStart
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // 直接使用 `x * y` 会导致错误，因为它们可能为 null.
    if (x != null && y != null) {
        // 在进行过 null 值检查之后, x 和 y 的类型会被自动转换为非 null 变量
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
//sampleEnd

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("a", "b")
}
```

或者

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

//sampleStart
    // ...
```

```
    if (x == null) {
        println("Wrong number format in arg1: '$arg1'")
        return
    }
    if (y == null) {
        println("Wrong number format in arg2: '$arg2'")
        return
    }

    // 在进行过 null 值检查之后, x 和 y 的类型会被自动转换为非 null 变量
    println(x * y)
//sampleEnd
}

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("99", "b")
}
```

参见 Null 值安全 (Null 值安全性).

类型检查与自动类型转换

`is` 运算符可以检查一个表达式的值是不是某个类型的实例. 如果对一个不可变的局部变量或属性进行过类型检查, 那么之后的代码就不必再对它进行显式地类型转换, 而可以直接将它当作需要的类型来使用:

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // 在这个分支中, `obj` 的类型会被自动转换为 `String`
        return obj.length
    }

    // 在类型检查所影响的分支之外, `obj` 的类型仍然是 `Any`
    return null
}
//sampleEnd
```

```
fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result:
${getStringLength(obj) ?: "Error: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

或者

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // 在这个分支中, `obj` 的类型会被自动转换为 `String`
    return obj.length
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result:
${getStringLength(obj) ?: "Error: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

甚至还可以

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    // 在 `&&` 运算符的右侧, `obj` 的类型会被自动转换为 `String`
    if (obj is String && obj.length > 0) {
        return obj.length
```

```
    }

    return null
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result:
${getStringLength(obj) ?: "Error: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength("")
    printLength(1000)
}
```

参见 类 (类) 和 类型转换 (类型检查与类型转换).

惯用法

最终更新: 2024/09/10

本章介绍 Kotlin 中的一些常见的习惯用法. 如果你有自己的好的经验, 可以将它贡献给我们. 你可以将你的修正提交到 git, 并创建一个 Pull Request.

创建 DTO 类(或者叫 POJO/POCO 类)

```
data class Customer(val name: String, val email: String)
```

以上代码将创建一个 `Customer` 类, 其中包含以下功能:

- 所有属性的 getter 函数(对于 `var` 型属性还有 setter 函数)
- `equals()` 函数
- `hashCode()` 函数
- `toString()` 函数
- `copy()` 函数
- 所有属性的 `component1()`, `component2()`, ... 函数(参见 数据类 (数据类(Data Class)))

对函数参数指定默认值

```
fun foo(a: Int = 0, b: String = "") { ... }
```

过滤 List 中的元素

```
val positives = list.filter { x -> x > 0 }
```

甚至还可以写得更短:

```
val positives = list.filter { it > 0 }
```

详情请参见 Java 与 Kotlin 过滤处理的区别 ("过滤元素" in "Java 和 Kotlin 中的集合(Collection)").

在集合中检查元素是否存在

```
if ("john@example.com" in emailsList) { ... }

if ("jane@example.com" !in emailsList) { ... }
```

在字符串内插入变量值

```
println("Name $name")
```

详情请参见 Java 与 Kotlin 字符串拼接处理的区别 ("拼接字符串" in "Java 和 Kotlin 中的字符串").

类型实例检查

```
when (x) {
    is Foo -> ...
    is Bar -> ...
    else   -> ...
}
```

只读 List

```
val list = listOf("a", "b", "c")
```

只读 Map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

访问 Map 中的条目

```
println(map["key"])
```

```
map["key"] = value
```

使用成对变量来遍历 Map, 或遍历 Pair 组成的 List

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

上例中的 `k`, `v` 可以使用任何方便的变量名, 比如 `name` 和 `age`.

在数值范围中遍历

```
for (i in 1..100) { ... }  // 终端封闭的(closed-ended)数值范围: 包括 100
for (i in 1..<100) { ... } // 终端开放的(open-ended)数值范围: 不包括 100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
(1..10).forEach { ... }
```

延迟计算(Lazy)属性

```
val p: String by lazy { // 只在第一次访问时计算属性值
    // 在这里计算字符串值
}
```

扩展函数

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

创建单例(Singleton)

```
object Resource {
    val name = "Name"
```

```
}
```

为抽象类(Abstract Class)创建实例

```
abstract class MyAbstractClass {
    abstract fun doSomething()
    abstract fun sleep()
}

fun main() {
    val myObject = object : MyAbstractClass() {
        override fun doSomething() {
            // ...
        }

        override fun sleep() { // ...
        }
    }
    myObject.doSomething()
}
```

If not null 的简写表达方式

```
val files = File("Test").listFiles()

println(files?.size) // 如果 files 不为 null，这里会打印 size 值
```

If-not-null-else 的简写表达方式

```
val files = File("Test").listFiles()

// 简单的 fallback 值:
println(files?.size ?: "empty") // 如果 files 为 null，这里会打印
"empty"

// 如果要通过一个代码段来计算更加复杂的 fallback 值，可以使用 `run`
```

```
val filesSize = files?.size ?: run {
    val someSize = getSomeSize()
    someSize * 2
}
println(filesSize)
```

当值为 null 时, 执行某个语句

```
val values = ...
val email = values["email"] ?: throw IllegalStateException("Email is
missing!")
```

从可能为空的集合中取得第一个元素

```
val emails = ... // 可能为空
val mainEmail = emails.firstOrNull() ?: ""
```

详情请参见 Java 与 Kotlin 集合第一个元素的获取方法的区别 ("从可能为空的集合得到第 1 个和最后 1 个元素" in "Java 和 Kotlin 中的集合(Collection)").

当值不为 null 时, 执行某个语句

```
val value = ...

value?.let {
    ... // 这个代码段将在 data 不为 null 时执行
}
```

当值不为 null 时, 进行映射变换

```
val value = ...

val mapped = value?.let { transformValue(it) } ?: defaultValue
// 如果 value 为 null, 会 transform 处理结果为 null, 则返回 defaultValue
```

在函数的 return 语句中使用 when 语句

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param
value")
    }
}
```

将 try-catch 用作一个表达式

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // 使用 result
}
```

将 if 用作一个表达式

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

返回值为 Unit 类型的多个方法, 可以通过 Builder 风格的方式来串联调用

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

使用单个表达式来定义一个函数

```
fun theAnswer() = 42
```

以上代码等价于:

```
fun theAnswer(): Int {
    return 42
}
```

这种用法与其他惯用法有效地结合起来, 可以编写出更简短的代码. 比如. 可以与 `when` 表达式结合起来:

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param
value")
}
```

在同一个对象实例上调用多个方法(with 函数)

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
```

```
}

val myTurtle = Turtle()
with(myTurtle) { // 描绘一个边长 100 像素的正方形
    penDown()
    for (i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

配置对象属性 (apply 函数)

```
val myRectangle = Rectangle().apply {
    length = 4
    breadth = 5
    color = 0xFAFAFA
}
```

这种方法可以非常方便地配置对象构造函数参数以外的那些属性.

类似 Java 7 中针对资源的 try 语句

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

需要泛型类型信息的泛型函数

```
//  public final class Gson {
//     ...
//     public <T> T fromJson(JsonElement json, Class<T> classOfT)
throws JsonSyntaxException {
//     ...
```

```
inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T =
this.fromJson(json, T::class.java)
```

交换两个变量的值

```
var a = 1
var b = 2
a = b.also { b = a }
```

将代码标记为未完成 (TODO)

Kotlin 标准库有一个 `TODO()` 函数, 它永远会抛出一个 `NotImplementedError`. 这个函数的返回值是 `Nothing`, 因此无论代码中需要的返回类型是什么, 都可以使用这个函数. 这个函数还有一个参数重载(overload)的版本, 接受一个参数, 用来解释具体的原因:

```
fun calcTaxes(): BigDecimal = TODO("Waiting for feedback from
accounting")
```

IntelliJ IDEA 的 Kotlin 插件能够理解 `TODO()` 函数的意义, 并会在 TODO 工具窗口中自动添加一条 TODO 项.

下一步做什么?

- 使用 Kotlin 的编程风格来解决 Advent of Code 谜题 (使用 Kotlin 惯用法的 Advent of Code).
- 学习如何执行 Java 与 Kotlin 中常见的字符串处理任务 (Java 和 Kotlin 中的字符串).
- 学习如何执行 Java 与 Kotlin 中常见的集合(Collection)处理任务 (Java 和 Kotlin 中的集合(Collection)).
- 学习如何 在 Java 与 Kotlin 中处理可空性(Nullability) (Java 和 Kotlin 中的可空性(Nullability)).

编码规约

最终更新: 2024/09/10

对任何编程语言来说, 都需要一种广为人知, 并且易于遵守的编码规约. 这里我们对使用 Kotlin 的项目, 给出一些编码规约和代码组织的指导原则.

在 IDE 中配置代码规则

最流行的2个 Kotlin IDE - IntelliJ IDEA (https://www.jetbrains.com/idea/) 和 Android Studio (https://developer.android.com/studio/) 对代码规则提供了强大的支持. 你可以配置代码规则来自动格式化你的代码, 是代码符合统一的规则.

应用代码规则

1. 进入设置界面 **Settings/Preferences | Editor | Code Style | Kotlin**.
2. 点击 **Set from...**.
3. 选择 **Kotlin style guide**.

验证你的代码是否符合代码规则

1. 进入设置界面 **Settings/Preferences | Editor | Inspections | General**.
2. 选中检查项 **Incorrect formatting**. 对于本编码规约中提到的其他问题 (比如命名规约), 相应的检查项目默认已经启用了.

源代码组织

目录结构

在纯 Kotlin 语言的项目中, 建议源代码文件的目录结构遵循包的结构, 但省略共通的源代码根目录. 比如, 如果项目内的所有源代码都在 `org.example.kotlin` 包及其子包之下, 那么 `org.example.kotlin` 包对应的文件应该直接保存到源代码的根目录下, 而 `org.example.kotlin.network.socket` 包下的文件应该保存在源代码根目录下的 `network/socket` 子目录下.

> 对于 JVM 平台: 在混合使用 Kotlin 和 Java 的项目中, Kotlin 源代码文件应该与 Java 源代码文件放在相同的源代码根目录下, 并且遵循相同的目录结构: 每个文件应该保存在它的 package 语句对应的目录之下.

源代码文件名

如果 Kotlin 源代码文件只包含单个类或接口 (以及相关的顶级声明), 那么源代码文件的名称应该与类名相同, 再加上 `.kt` 扩展名. 这个规则适用于所有类型的类和接口. 如果源代码文件包含多个类, 或者只包含顶级声明, 请选择一个能够描述文件所包含内容的名称, 用这个名称作为源代码文件名. 文件名如果包含多个单词, 请使用 驼峰式大小写 (https://en.wikipedia.org/wiki/Camel_case), 将首字母大写(又叫做 Pascal 风格大小写), 比如, `ProcessDeclarations.kt`.

文件的名称应该描述其中包含的代码的功能. 因此, 应该避免在文件名中使用无意义的单词, 比如 `Util`.

跨平台项目

在跨平台项目中, 在平台相关源代码集中, 带有顶级(top-level)声明的文件应该带有后缀, 后缀关联到源代码集名称. 例如:

- **jvm**Main/kotlin/Platform. **jvm** .kt
- **android**Main/kotlin/Platform. **android** .kt
- **ios**Main/kotlin/Platform. **ios** .kt

对于 common 源代码集, 带有顶级声明的文件不应该带有后缀. 例如, `commonMain/kotlin/Platform.kt`.

技术细节

我们推荐在跨平台项目中遵循这样的文件命名风格, 是因为 JVM 的限制: 它不允许存在顶层成员 (函数, 属性).

为了解决这个问题, Kotlin JVM 编译器会创建封装类(wrapper class), (也就是所谓的 "File Facade"), 通过这些封装类来包含顶层成员的声明. File Facade 拥有一个根据文件名称得到的内部名称.

而且, JVM 不允许多个类使用相同的完全限定名 (FQN). 这可能会导致 Kotlin 项目在 JVM 上无法编译:

```
root
|- commonMain/kotlin/myPackage/Platform.kt // 包含 'fun count() { }'
|- jvmMain/kotlin/myPackage/Platform.kt // 包含 'fun multiply() { }'
```

这时, 两个 `Platform.kt` 文件属于相同的包, 因此 Kotlin JVM 编译器生成两个 File Facade, 它们的 FQN 都是 `myPackage.PlatformKt`. 因此发生 "Duplicate JVM classes" 错误.

避免这个错误的最简单的方法是, 遵照上面所说的规约, 将某个文件改名. 这样的命名规约可以帮助避免名称冲突, 同时保持代码的可读性.

> 在两种情况下, 上面的命名规约可以省略, 但我们仍然建议遵循这种命名规约:
>
> - 非 JVM 平台 对重复的 File Facade 不会发生错误. 但是, 这种命名规约可以帮助你保持文件名称的一致性.
> - 在 JVM 平台上, 如果源代码文件不包含顶层声明, 就不会生成File Facade, 因此你不会遇到名称冲突的问题.
>
>   但是, 只要一次简单的代码重构, 或代码添加一个顶层函数, 就可以造成 "Duplicate JVM classes" 错误, 这种命名规约可以帮助你避免这样的情况.

源代码文件的组织

如果多个声明 (类, 顶级函数, 或顶级属性) 在语义上相互之间相关密切, 并且文件大小合理(不超过几百行的规模), 那么我们鼓励将这些放在同一个 Kotlin 源代码文件中.

尤其是, 当为类定义扩展函数时, 如果与这个类的所有使用者都有关系, 那么应该将它们与这个类放在一起. 如果定义的扩展函数, 只对特定的使用者有意义, 请将它们放在这个使用者的代码之后. 不要仅仅为了保存某个类的所有扩展函数而创建一个单独的源代码文件.

类的布局

类的内容按以下顺序排列:

1. 属性声明, 以及初始化代码端
2. 次构造器
3. 方法声明
4. 同伴对象

请不要将方法声明按照字母顺序排列, 也不要按照可见度顺序排列, 也不要将常规方法与扩展方法分开. 相反, 要将关系紧密的代码放在一起, 以便让他人从上到下阅读代码时, 能够理解代码的逻辑含义. 你应该选择一个排序原则 (将逻辑含义上比较顶层的代码在前, 或者反过来), 然后在所有的代码中都遵循相同的原则.

将嵌套类放在使用它的代码之后. 如果嵌套类是为了供外部使用, 没有被类内部的代码使用, 那么请将它放在最后, 放在同伴对象之后.

接口实现类的布局

实现一个接口时, 将实现类中的成员方法顺序, 保持与接口中的声明顺序一致 (如果需要的话, 中间可以插入被实现方法用到的其它私有方法).

重载方法的布局

将同一个类中的同名重载方法放在一起.

命名规约

Kotlin 中的包和类的命名规则非常简单:

- 包名称总是使用小写字母, 并且不使用下划线(`org.example.project`). 通常不鼓励使用多个单词的名称, 但如果的确需要, 你可以将多个单词直接连接在一起, 或者使用驼峰式大小写 (`org.example.myProject`).
- 类和对象的名称以大写字母开头, 并且使用驼峰式大小写:

```
open class DeclarationProcessor { /*...*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /*...*/
}
```

函数名称

函数, 属性, 以及局部变量的名称以小写字母开头, 并且使用驼峰式大小写, 而且不使用下划线:

```
fun processDeclarations() { /*...*/ }
var declarationCount = 1
```

例外情况: 用于创建类实例的工厂函数, 可以使用与它创建的抽象类型相同的名称:

```
interface Foo { /*...*/ }

class FooImpl : Foo { /*...*/ }

fun Foo(): Foo { return FooImpl() }
```

测试方法名称

在测试代码中 (而且**只有**在测试代码中), 可以使用由反引号括起的, 带空格的方法名. 注意, 对于 Android 运行环境, 这样的方法名只在 API level 30 才开始支持. 测试代码中的方法名, 也允许使用下划线.

```
class MyTestCase {
     @Test fun `ensure everything works`() { /*...*/ }

     @Test fun ensureEverythingWorks_onAndroid() { /*...*/ }
}
```

属性名称

对于常数 (标记了 `const` 的属性, 或者不存在自定义的 `get` 函数顶级的 `val` 属性, 或对象的 `val` 属性, 并且其值是深层不可变数据), 应该使用下划线分隔的大写 (吼叫式蛇形大小写 (https://en.wikipedia.org/wiki/Snake_case)) 名称:

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

顶级属性, 或对象属性, 如果它的值是对象, 或者包含可变的数据, 那么应该使用驼峰式大小写名称:

```
val mutableCollection: MutableSet<String> = HashSet()
```

如果属性指向单体对象, 那么可以使用与 `object` 声明相同的命名方式:

```
val PersonComparator: Comparator<Person> = /*...*/
```

对于枚举常数, 可以使用下划线分隔的大写名称 (吼叫式蛇形大小写 (https://en.wikipedia.org/wiki/Snake_case)) (`enum class Color { RED, GREEN }`), 也可以使用首字母大写的驼峰式大小写名称, 由你的具体用法来决定.

后端属性名称

如果类拥有两个属性, 它们在概念上是相同的, 但其中一个是公开 API 的一部分, 而另一个属于内部的实现细节, 此时请使用下划线作为私有属性名的前缀:

```
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
         get() = _elementList
}
```

选择好的名称

类的名称通常使用名词, 或名词短语, 要能够解释这个类 *是* 什么: `List`, `PersonReader`.

方法名称通常使用动词, 或动词短语, 说明这个方法 *做* 什么: `close`, `readPersons`. 方法名称还应该能够说明这个方法是变更这个对象, 或者还是返回一个新的实例. 比如 `sort` 是对集合(collection)本身的内容排序, 而 `sorted` 则是返回这个集合的一个副本, 其中包含排序后内容.

名称应该解释清楚这个类或方法的目的是什么, 因此最好在命名时避免使用含义不清的词语 (`Manager`, `Wrapper`).

在名称中使用缩写字母时, 如果缩写字母只包含两个字母, 请将它们全部大写 (比如 `IOStream`); 如果超过两个字母, 请将首字母大写, 其他字母小写 (比如 `XmlFormatter`, `HttpInputStream`).

代码格式化

缩进

缩进时使用 4 个空格. 不要使用 tab.

对于大括号, 请将开括号放在结构开始处的行末, 将闭括号放在单独的一行, 与它所属的结构缩进到同样的位置.

```
if (elements != null) {
    for (element in elements) {
        // ...
    }
}
```

> 在 Kotlin 中, 分号是可以省略的, 因此折行很重要. 语言设计时预想使用 Java 风格的大括号, 如果你使用不同的格式化风格, 你的代码执行时的行为可能会与你预想的不同.

水平空格

- 二元运算符前后应该加入空格 (`a + b`). 例外情况是: 不要在 "值范围" 运算符前后加入空格 (`0..i`).
- 一元运算符前后不要加入空格 (`a++`)
- 流程控制关键字(`if`, `when`, `for` 以及 `while`) 以及对应的开括号之间, 要加入空格.
- 对于主构造器声明, 方法声明, 以及方法调用, 不要在开括号之前加入空格.

```
class A(val x: Int)

fun foo(x: Int) { ... }

fun bar() {
    foo(1)
}
```

- 不要在 `(`, `[` 之后加入空格, 也不要在 `]`, `)` 之前加入空格.
- 不要在 `.` 或 `?.` 前后加入空格: `foo.bar().filter { it > 2 }.joinToString()`, `foo?.bar()`
- 在 `//` 之后加入空格: `// 这是一段注释`
- 对于用来表示类型参数的尖括号, 不要在它前后加入空格: `class Map<K, V> { ... }`
- 不要在 `::` 前后加入空格: `Foo::class`, `String::length`
- 对于用来表示可空类型的 `?`, 不要在它之前加入空格: `String?`

一般来说, 不要进行任何形式的水平对其. 如果将一个标识符改为不同长度的名称, 不应该影响到它的任何声明, 以及任何使用的格式.

冒号

以下情况, 要在 `:` 之前加入空格:

- 用作类型与父类型之间的分隔符时
- 委托给超类的构造器, 或者委托给同一个类的另一个构造器时
- 用在 `object` 关键字之后时

如果 `:` 用作某个声明与它的类型之间的分隔符时, 不要它前面加入空格.

在 `:` 之后, 一定要加入一个空格.

```
abstract class Foo<out T : Any> : IFoo {
    abstract fun foo(a: Int): T
}

class FooImpl : Foo() {
    constructor(x: String) : this(x) { /*...*/ }

    val x = object : IFoo { /*...*/ }
}
```

类头部

如果类的主构造器只有少量参数, 可以写成单独的一行:

```
class Person(id: Int, name: String)
```

如果类的头部很长, 应该调整代码格式, 将主构造器(primary constructor)的每一个参数放在单独的行中, 并对其缩进. 同时, 闭括号也应放在新的一行. 如果使用类的继承, 那么对超类构造器的调用, 以及实现的接口的列表, 应该与闭括号放在同一行内:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) { /*...*/ }
```

对于多个接口的情况, 对超类构造器的调用应该放在最前, 然后将每个接口放在单独的行中:

```
class Person(
    id: Int,
```

```
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker { /*...*/ }
```

如果类的父类型列表很长, 请在冒号之后换行, 并将所有的父类型名称缩进到同样的位置:

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne {

    fun foo() { /*...*/ }
}
```

当类头部很长时, 为了将类头部和类主体部分更清楚地分隔开, 可以在类头部之后加入一个空行(如上面的例子所示), 也可以将大括号放在单独的一行:

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne
{
    fun foo() { /*...*/ }
}
```

对构造器的参数, 使用通常的缩进(4 个空格). 这是为了让主构造器中声明的属性, 与类主体部分声明的属性的缩进保持一致.

修饰符顺序

如果一个声明带有多个修饰符, 修饰符一定要按照下面的顺序排列:

```
public / protected / private / internal
expect / actual
final / open / abstract / sealed / const
external
override
lateinit
```

```
tailrec
vararg
suspend
inner
enum / annotation / fun // 在 `fun interface` 中, `fun` 是修饰符
companion
inline / value
infix
operator
data
```

所有的注解要放在修饰符之前:

```
@Named("Foo")
private val foo: Foo
```

除非你在开发一个库, 否则应该省略多余的修饰符(比如 `public`).

注解(Annotation)

注解放在它修饰的声明之前, 放在单独的行中, 使用相同的缩进:

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

无参数的注解可以放在同一行中:

```
@JsonExclude @JvmField
var x: String
```

单个无参数的注解可以与它修饰的声明放在同一行中:

```
@Test fun foo() { /*...*/ }
```

文件注解

文件注解放在文件注释之后(如果存在的话), 在 `package` 语句之前, 与 `package` 语句之间用空行隔开 (为了强调注解的对象是文件, 而不是包).

```
/** License, copyright and whatever */
@file:JvmName("FooBar")

package foo.bar
```

函数

如果函数签名无法排列在一行之内, 请使用下面的语法:

```
fun longMethodName(
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType,
): ReturnType {
    // body
}
```

函数参数使用通常的缩进(4 个空格). 这是为了与构造器参数保持一致

如果函数体只包含单独的一个表达式, 应当使用表达式函数体.

```
fun foo(): Int {     // 这是不好的风格
    return 1
}

fun foo() = 1        // 这是好的风格
```

表达式体

如果函数体表达式太长, 它的第一行无法与函数声明放在同一行之内, 那么应该将 `=` 符号放在第一行, 然后表达式函数体放在下一行, 缩进 4 个空格.

```
fun f(x: String, y: String, z: String) =
    veryLongFunctionCallWithManyWords(andLongParametersToo(), x, y,
z)
```

属性

对于简单的只读属性, 应该使用单行格式:

```
val isEmpty: Boolean get() = size == 0
```

对更复杂一些的属性, 一定要将 `get` 和 `set` 关键字放在单独的行:

```
val foo: String
    get() { /*...*/ }
```

对于带有初始化器(initializer)的属性, 如果初始化器很长, 请在等号之后换行, 然后对初始化器缩进 4 个空格:

```
private val defaultCharset: Charset? =

EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(f
ile)
```

控制流语句

如果 `if` 或 `when` 语句的条件部分有多行代码, 一定要将主体部分用大括号括起. 将条件部分的每一个子句, 从语句开始的位置缩进 4 个空格. 将条件部分的闭括号, 与主体部分的开括号一起, 放在单独一行:

```
if (!component.isSyncing &&
    !hasAnyKotlinRuntimeInScope(module)
) {
    return createKotlinNotConfiguredPanel(module)
}
```

这样可以将条件部分与主体部分对齐.

将 `else`, `catch`, `finally` 关键字, 以及 `do-while` 循环语句的 `while` 关键字, 与它之后的开括号放在同一行中:

```
if (condition) {
    // 主体部分
} else {
    // 其它部分
}

try {
```

```
    // 主体部分
} finally {
    // 清除处理
}
```

在 `when` 语句中, 如果一个条件分支包含了多行语句, 应该将它与临近的条件分支用空行分隔开:

```
private fun parsePropertyValue(propName: String, token: Token) {
    when (token) {
        is Token.ValueToken ->
            callback.visitValue(propName, token.value)

        Token.LBRACE -> { // ...
        }
    }
}
```

对于比较短的分支, 与条件部分放在同一行中, 不用大括号.

```
when (foo) {
    true -> bar() // 这是比较好的风格
    false -> { baz() } // 这是不好的风格
}
```

方法调用

如果参数列表很长, 请在开括号之后换行. 参数缩进 4 个空格. 关系紧密的多个参数放在同一行中.

```
drawSquare(
    x = 10, y = 10,
    width = 100, height = 100,
    fill = true
)
```

在 `=` 前后加入空格, 将参数名与参数值分隔开.

链式调用(chained call)的换行

对链式调用(chained call)换行时, 将 `.` 字符或 `?.` 操作符放在下一行, 使用单倍缩进:

```
val anchor = owner
    ?.firstChild!!
    .siblings(forward = true)
    .dropWhile { it is PsiComment || it is PsiWhiteSpace }
```

链式调用中的第一个调用, 在它之前通常应该换行, 但如果能让代码更合理, 也可以省略换行.

Lambda 表达式

在 Lambda 表达式中, 在大括号前后应该加入空格, 分隔参数与表达式体的箭头前后也要加入空格. 如果一个函数调用可以接受单个 Lambda 表达式作为参数, 那么 Lambda 表达式应该尽可能写到函数调用的圆括号之外.

```
list.filter { it > 10 }
```

如果为 Lambda 表达式指定标签, 请不要在标签与表达式体的开括号之间加入空格:

```
fun foo() {
    ints.forEach lit@{
        // ...
    }
}
```

在多行的 Lambda 表达式中声明参数名称时, 请将参数名放在第一行, 后面放箭头, 然后换行:

```
appendCommaSeparated(properties) { prop ->
    val propertyValue = prop.get(obj)  // ...
}
```

如果参数列表太长, 无法放在一行之内, 请将箭头放在单独的一行:

```
foo {
   context: Context,
   environment: Env
   ->
   context.configureEnv(environment)
}
```

尾随逗号(Trailing Comma)

尾随逗号是指, 在一系列元素的最末尾之后出现的逗号:

```
class Person(
    val firstName: String,
    val lastName: String,
    val age: Int, // 尾随逗号
)
```

使用尾随逗号可以带来下面这些益处:

- 版本控制中的差分比较更加清晰 – 因为差分只会出现在真正修改过的代码行.
- 更加易于添加元素, 或改变元素顺序 – 修改元素时不再需要添加或删除逗号.
- 简化了代码生成工作, 比如, 对于对象的初始化代码. 最后一个元素也可以带有逗号.

尾随逗号完全是可选的 – 没有尾随逗号, 你的代码仍然可以工作. Kotlin 编码风格向导鼓励在声明处使用尾随逗号, 在调用处则由你自己决定.

要在 IntelliJ IDEA 的代码格式化工具中启用尾随逗号, 请进入设置界面 **Settings/Preferences | Editor | Code Style | Kotlin**, 打开 **Other** 页, 然后选中 **Use trailing comma** 选项.

枚举

```
enum class Direction {
    NORTH,
    SOUTH,
    WEST,
    EAST, // 尾随逗号
}
```

值参数

```
fun shift(x: Int, y: Int) { /*...*/ }
shift(
    25,
    20, // 尾随逗号
)
```

```
val colors = listOf(
    "red",
    "green",
    "blue", // 尾随逗号
)
```

类的属性和参数

```
class Customer(
    val name: String,
    val lastName: String, // 尾随逗号
)
class Customer(
    val name: String,
    lastName: String, // 尾随逗号
)
```

函数值参数

```
fun powerOf(
    number: Int,
    exponent: Int, // 尾随逗号
) { /*...*/ }
constructor(
    x: Comparable<Number>,
    y: Iterable<Number>, // 尾随逗号
) {}
fun print(
    vararg quantity: Int,
    description: String, // 尾随逗号
) {}
```

带有可选类型的参数 (包括属性的 set 函数)

```
val sum: (Int, Int, Int) -> Int = fun(
    x,
    y,
    z, // 尾随逗号
): Int {
```

```
    return x + y + x
}
println(sum(8, 8, 8))
```

下标后缀

```
class Surface {
    operator fun get(x: Int, y: Int) = 2 * x + 4 * y - 10
}
fun getZValue(mySurface: Surface, xValue: Int, yValue: Int) =
    mySurface[
        xValue,
        yValue, // 尾随逗号
    ]
```

Lambda 表达式的参数

```
fun main() {
    val x = {
            x: Comparable<Number>,
            y: Iterable<Number>, // 尾随逗号
        ->
        println("1")
    }
    println(x)
}
```

when 语句的分支条件

```
fun isReferenceApplicable(myReference: KClass<*>) = when
(myReference) {
    Comparable::class,
    Iterable::class,
    String::class, // 尾随逗号
        -> true
    else -> false
}
```

集合字面值 (在注解中)

```
annotation class ApplicableFor(val services: Array<String>)
@ApplicableFor([
    "serializer",
    "balancer",
    "database",
    "inMemoryCache", // 尾随逗号
])
fun run() {}
```

类型参数(Type argument)

```
fun <T1, T2> foo() {}

fun main() {
    foo<
            Comparable<Number>,
            Iterable<Number>, // 尾随逗号
            >()
}
```

类型参数(Type parameter)

```
class MyMap<
        MyKey,
        MyValue, // 尾随逗号
        > {}
```

解构声明

```
data class Car(val manufacturer: String, val model: String, val
year: Int)
val myCar = Car("Tesla", "Y", 2019)
val (
    manufacturer,
    model,
    year, // 尾随逗号
```

```
) = myCar
val cars = listOf<Car>()
fun printMeanValue() {
    var meanValue: Int = 0
    for ((
        _,
        _,
        year, // 尾随逗号
    ) in cars) {
        meanValue += year
    }
    println(meanValue/cars.size)
}
printMeanValue()
```

文档注释

对于比较长的文档注释, 请将开头的 `/**` 放在单独的行, 后面的每一行都用星号开始:

```
/**
 * 这是一段文档注释,
 * 其中包含多行.
 */
```

比较短的注释可以放在一行之内:

```
/** 这是一段比较短的文档注释. */
```

通常来说, 不要使用 `@param` 和 `@return` 标记. 相反, 对参数和返回值的描述应该直接合并到文档注释之内, 在提到参数的地方应该添加链接. 只有参数或返回值需要很长的解释, 无法写在文档注释中, 这时才应该使用 `@param` 和 `@return` 标记.

```
// 不要写这样的注释:

/**
 * 对于给定的数值, 返回其绝对值.
 * @param number 需要返回绝对值的对象数值.
 * @return 绝对值.
```

```
 */
fun abs(number: Int): Int { /*...*/ }

// 应该这样:

/**
 * 对于给定的 [number], 返回其绝对值.
 */
fun abs(number: Int): Int { /*...*/ }
```

避免冗余的结构

通常来说, 如果 Kotlin 代码中的某个语法结构是可省略的, 并且被 IDE 标记显示为可省略的, 那么你就应该在代码中省略这部分. 不要仅仅"为了解释清楚", 就在代码中留下不必须的语法元素.

Unit 返回类型

如果函数的返回值为 Unit 类型, 那么返回值的类型声明应当省略:

```
fun foo() { // 此处省略了 ": Unit"

}
```

分号

尽可能省略分号.

字符串模板

向字符串模板中插入简单变量时, 不要使用大括号. 只有对比较长的表达式, 才应该使用大括号.

```
println("$name has ${children.size} children")
```

各种语言特性的惯用法

数据的不可变性

尽量使用不可变的数据, 而不是可变的数据. 如果局部变量或属性的值在初始化之后不再变更, 尽量将它们声明为 `val`, 而不是 `var`.

对于内容不发生变化的集合，一定要使用不可变的集合接口(`Collection`, `List`, `Set`, `Map`) 来声明. 当使用工厂方法创建集合类型时，一定要尽可能使用返回不可变集合类型的函数:

```
// 这是不好的风格: 对于内容不再变化的值, 使用了可变的集合类型
fun validateValue(actualValue: String, allowedValues:
HashSet<String>) { ... }

// 这是比较好的风格: 改用了不可变的集合类型
fun validateValue(actualValue: String, allowedValues: Set<String>) {
... }

// 这是不好的风格: arrayListOf() 的返回类型为 ArrayList<T>, 这是一个可变的
集合类型
val allowedValues = arrayListOf("a", "b", "c")

// 这是比较好的风格: listOf() 的返回类系为 List<T>
val allowedValues = listOf("a", "b", "c")
```

参数默认值

尽可能使用带默认值的参数来声明函数，而不是声明多个不同参数的重载函数.

```
// 不好的风格
fun foo() = foo("a")
fun foo(a: String) { /*...*/ }

// 比较好的风格
fun foo(a: String = "a") { /*...*/ }
```

类型别名

如果你的某个函数类型，或者某个带类型参数的类型，在代码中多次用到，那么应该尽量为它定义一个类型别名:

```
typealias MouseClickHandler = (Any, MouseEvent) -> Unit
typealias PersonIndex = Map<String, Person>
```

如果你使用 private 或 internal 的类型别名来避免名称冲突，建议改为使用 包(Package)与导入(Import) (包(Package)与导入(Import)) 中介绍的 `import ... as ...` 功能.

Lambda 表达式参数

在比较短, 而且没有嵌套的 Lambda 表达式, 建议使用 `it` 规约, 而不要明确声明参数. 在有参数的嵌套 Lambda 表达式中, 参数一定要明确声明.

在 Lambda 表达式中返回

不要在 Lambda 表达式中使用多个带标签的返回. 应该考虑重构你的 Lambda 表达式, 使它只有一个退出点. 如果无法做到, 或者代码不够清晰, 那么可以考虑把 Lambda 改为一个匿名函数.

在 Lambda 表达式中, 不要使用带标签的返回语句作为最后一条语句.

命名参数

如果一个方法接受同一种基本类型的多个参数, 或者如果参数为 `Boolean` 类型, 除非通过代码的上下文, 可以非常清楚地确定所有参数的含义, 否则此时应该使用命名参数语法.

```
drawSquare(x = 10, y = 10, width = 100, height = 100, fill = true)
```

条件语句

尽量使用 `try`, `if` 以及 `when` 的表达式形式.

```
return if (x) foo() else bar()
```

```
return when(x) {
    0 -> "zero"
    else -> "nonzero"
}
```

上面的写法比下面的代码要好:

```
if (x)
    return foo()
else
    return bar()
```

```
when(x) {
    0 -> return "zero"
```

```
    else -> return "nonzero"
}
```

if 和 when

对于二元的条件分支, 尽量使用 `if` 而不是 `when`. 比如, 这里应该用 `if`:

```
if (x == null) ... else ...
```

而不是用 `when`:

```
when (x) {
    null -> // ...
    else -> // ...
}
```

如果存在三个或更多的条件分支, 尽量使用 `when`.

在条件中使用可为空的 Boolean 值

如果需要在条件语句中使用可为空的 `Boolean`, 请使用 `if (value == true)` 或者 `if (value == false)` 进行判断.

循环

尽量使用高阶函数(`filter`, `map` 等等.) 来进行循环处理. 例外情况: `forEach` (应该尽量使用通常的 `for` 循环, 除非 `forEach` 函数的接受者对象可能为空, 或者 `forEach` 是一个很长的链式调用的一部分).

应该使用多个高阶函数组成的复杂表达式, 还是应该使用一个循环语句, 选择之前应该理解这两种操作各自的代价, 并且注意考虑性能问题.

在数值范围上循环

对于终端开放(open-ended)的值范围(不包含其末尾元素), 那么应该使用 `..<` 操作符进行循环:

```
for (i in 0..n - 1) { /*...*/ }  // 不好的风格
for (i in 0..<n) { /*...*/ }  // 比较好的风格
```

字符串

尽量使用字符串模板来进行字符串拼接.

尽量使用多行字符串, 而不是在通常的字符串字面值中使用内嵌的 `\n` 转义符.

关于多行字符串中缩进的维护, 如果结果字符串内部不需要任何缩进, 应该使用 `trimIndent` 函数, 如果字符串内部需要缩进, 应该使用 `trimMargin` 函数:

```
fun main() {
//sampleStart
    println("""
        Not
        trimmed
        text
        """
    )

    println("""
        Trimmed
        text
        """.trimIndent()
    )

    println()

    val a = """Trimmed to margin text:
            |if(a > 1) {
            |    return a
            |}""".trimMargin()

    println(a)
//sampleEnd
}
```

详情请参见 Java 与 Kotlin 的多行字符串的区别 ("使用多行字符串" in "Java 和 Kotlin 中的字符串").

函数 vs 属性

有些情况下, 无参数的函数可以与只读属性相互替代. 虽然它们在语义上是相似的, 但从编程风格上的角度看, 存在一些规约来决定在什么时候应该使用函数, 什么时候应该使用属性.

当底层算法满足以下条件时, 应该选择使用只读属性, 而不是使用函数:

- 不会抛出异常
- 计算过程消费的资源不多(或者在初次运行时缓存了计算结果)
- 对象状态没有发生变化时, 多次调用会返回相同的结果

扩展函数

应该尽量多的使用扩展函数. 如果你的某个函数主要是为某个对象服务, 应该考虑将它转变为这个对象的一个扩展函数. 为了尽量减小 API 污染, 应该将扩展函数的可见度尽量限制在合理的程度. 如果需要, 尽量使用局部扩展函数, 成员扩展函数, 或者可见度为 private 的顶级扩展函数.

中缀函数

如果一个函数服务于两个参数, 而且这两个参数的角色很类似, 只有这种情况下才应该将函数声明为 `infix` 函数. 好的例子比如: `and`, `to`, `zip`. 坏的例子比如: `add`.

如果方法会变更它的接受者对象, 那么不应该将它声明为 `infix` 方法.

工厂函数

如果你为一个类声明一个工厂方法, 请不要使用与类相同的名称. 尽量使用一个不同的名称, 解释清楚工厂函数的行为有什么不同之处. 只有当工厂函数的确实不存在什么特殊意义的时候, 这时你才可以使用与类相同的名称作为函数名.

```
class Point(val x: Double, val y: Double) {
    companion object {
        fun fromPolar(angle: Double, radius: Double) = Point(...)
    }
}
```

如果某个对象拥有多个不同参数的重载构造器, 这些构造器不会调用超类中的不同的构造器, 而且无法缩减成带默认值参数的单个构造器, 这时应该将这些构造器改为工厂函数.

平台数据类型

对于 public 的函数或方法, 如果返回一个平台类型的表达式, 那么应该明确声明它在 Kotlin 中的类型:

```
fun apiCall(): String = MyJavaApi.getProperty("name")
```

(包级或者类级的)任何属性, 如果使用平台类型的表达式进行初始化, 那么应该明确声明它在 Kotlin 中的类型:

```
class Person {
    val name: String = MyJavaApi.getProperty("name")
}
```

局部变量值, 如果使用平台类型的表达式进行初始化, 那么可以为它声明类型, 也可以省略:

```
fun main() {
    val name = MyJavaApi.getProperty("name")
    println(name)
}
```

作用域函数(Scope Function): apply, with, run, also, let

Kotlin 提供了一组函数, 用来在某个指定的对象上下文中执行一段代码, 这些函数包括: `let`, `run`, `with`, `apply`, 以及 `also`. 对于具体的问题, 应该如何选择正确的作用域函数, 详情请参见 作用域函数(Scope Function) (作用域函数(Scope Function)).

针对库开发的编码规约

开发库时, 为了保证 API 的稳定性, 建议还要遵守以下规约:

- 始终明确指定成员的可见度 (以免不小心将某个声明暴露成 public API)
- 始终明确指定函数的返回类型, 以及属性类型 (以免修改实现代码时, 不小心改变了返回类型)
- 对所有的 public 成员编写 KDoc (为 Kotlin 代码编写文档: KDoc) 文档注释 (这是为了对库生成文档), 例外情况是, 方法或属性的覆盖不需要提供新的注释

关于为你的库编写 API 时的最佳实践, 以及需要考虑的问题, 请参见 库开发者指南 (入门).

基本类型

最终更新: 2024/09/10

在 Kotlin 中, 一切都是对象, 这就意味着, 你可以对任何变量访问它的成员函数和属性. 有些数据类型使用优化过的内部表现形式, 在运行时使用 Java 的基本类型(Primitive Value)来表达, (比如, 数值, 字符, 布尔值, 等等), 但对于使用者来说, 它们就和通常的类一样.

本章介绍 Kotlin 中使用的基本类型:

- 数值 (数值类型) 以及对应的 无符号数值 (无符号整数(Unsigned Integer)类型)
- 布尔值 (布尔(Boolean)类型)
- 字符 (字符)
- 字符串 (字符串)
- 数组 (数组)

> 参见, 在 Kotlin 中如何进行类型检查和类型转换 (类型检查与类型转换).

数值类型

最终更新: 2024/09/10

整数类型

Kotlin 提供了一组内建数据类型来表达数值. 对于整数数值, 有 4 种数据类型, 它们的大小不同, 因此表达的数值范围也不同:

类型	大小(bits)	最小值	最大值
`Byte`	8	-128	127
`Short`	16	-32768	32767
`Int`	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
`Long`	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

如果你初始化一个变量, 不明确指定类型, 编译器会自动推断类型, 使用从 `Int` 开始、足够表达这个值的最小的整数范围. 如果值没有超过 `Int` 类型的最大范围, 那么类型会推断为 `Int`. 如果超过, 那么类型将是 `Long`. 如果要明确指明一个数值是 `Long` 类型, 请在数值末尾添加 `L` 后缀. 如果明确指定类型, 编译器会检查值有没有超过指定类型的最大范围.

```
val one = 1 // Int 类型
val threeBillion = 3000000000 // Long 类型
val oneLong = 1L // Long 类型
val oneByte: Byte = 1
```

> 除整数类型外, Kotlin 还提供了无符号整数类型. 详情请参见 无符号整数类型 ([无符号整数(Unsigned Integer)类型](#)).

浮点类型

对于实数数值, Kotlin 提供了符合 IEEE 754 标准 (https://en.wikipedia.org/wiki/IEEE_754) 的浮点类型 `Float` 和 `Double`. `Float` 代表 IEEE 754 *单精度(single precision)浮点数*, 而 `Double` 代表 *双精度(double precision)浮点数*.

这两种类型的区别在于它们大小, 以及能够存储的浮点数值精度:

类型	大小(bits)	有效位数	指数位数	十进制位数
`Float`	32	24	8	6-7
`Double`	64	53	11	15-16

可以使用带小数部分的数值初始化 `Double` 和 `Float` 变量. 小数部分与整数部分用点号(`.`)分隔. 任何变量如果使用浮点数值初始化, 编译器推断的类型将是 `Double`:

```
val pi = 3.14 // Double 类型
// val one: Double = 1 // 编译错误: 类型不匹配
val oneDouble = 1.0 // Double 类型
```

如果要明确指明一个数值是 `Float` 类型, 请在数值末尾添加 `f` 或 `F` 后缀. 如果这个值包含 6-7 位以上的十进制数字, 这部分会被舍去:

```
val e = 2.7182818284 // Double 类型
val eFloat = 2.7182818284f // Float 类型, 实际的值将是 2.7182817
```

与其他一些语言不同, Kotlin 的数值类型没有隐式的拓宽变换. 比如, 如果函数使用 `Double` 参数, 那么只能使用 `Double` 值调用它, 而不能使用 `Float`, `Int`, 或其他数值类型的值:

```
fun main() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.0
    val f = 1.0f

    printDouble(d)
```

```
//    printDouble(i) // 编译错误: 类型不匹配
//    printDouble(f) // 编译错误: 类型不匹配
}
```

如果要将数值转换为不同的类型, 请使用 显式类型转换.

数值的字面值常数(Literal Constant)

对于整数值, 有以下几种类型的字面值常数:

- 10进制数: `123`
- Long 类型需要大写的 `L` 来标识: `123L`
- 16进制数: `0x0F`
- 2进制数: `0b00001011`

> Kotlin 不支持8进制数的字面值.

Kotlin 还支持传统的浮点数值表达方式:

- 无标识时默认为 Double 值: `123.5`, `123.5e10`
- Float 值需要用 `f` 或 `F` 标识: `123.5f`

你可以在数字字面值中使用下划线, 提高可读性:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

> 除此之外, 对无符号整数类型的字面值还有特殊的标识. 详情请参见 无符号整数类型的字面值 (无符号整数(Unsigned Integer)类型).

数值类型在 JVM 平台的内部表达

在 JVM 平台中, 数值的存储使用基本类型: `int`, `double`, 等等. 除非你创建一个可为 null 的数值引用, 比如 `Int?`, 或使用泛型. 这种情况下数值会被装箱(box)为 Java 类 `Integer`, `Double`, 等等.

可为 null 的数值引用即使指向相同的数值, 也可能指向不同的对象:

```
fun main() {
//sampleStart
    val a: Int = 100
    val boxedA: Int? = a
    val anotherBoxedA: Int? = a

    val b: Int = 10000
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b

    println(boxedA === anotherBoxedA) // 结果为 true
    println(boxedB === anotherBoxedB) // 结果为 false
//sampleEnd
}
```

所有指向 `a` 的可为 null 的引用实际上都是同一个对象, 因为JVM 针对 `-128` 与 `127` 之间的 `Integer` 类型会进行内存优化. 但这种优化对 `b` 的引用无效, 因此这些引用是不同的对象.

但是, 对象仍然是相等的:

```
fun main() {
//sampleStart
    val b: Int = 10000
    println(b == b) // 打印结果为 'true'
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b
    println(boxedB == anotherBoxedB) // 打印结果为 'true'
//sampleEnd
}
```

显式数值类型转换

由于数据类型内部表达方式的差异, 较小的数据类型 *不是较大数据类型的子类型(subtype)*. 如果小数据类型是大数据类型的子类型, 那么我们将会遇到以下问题:

```
// 以下为假想代码, 实际上是无法编译的:
val a: Int? = 1 // 装箱后的 Int (java.lang.Integer)
val b: Long? = a // 这里进行隐式类型转换, 产生一个装箱后的 Long
(java.lang.Long)
print(b == a) // 结果与你期望的相反! 这句代码打印的结果将是 "false", 因为
Long 的 equals() 方法会检查比较对象, 要求对方也是一个 Long 对象
```

这样, 不仅不能保持同一性(identity), 而且还静悄悄地失去了内容相等性(equality).

由于存在以上问题, Kotlin 中较小的数据类型 *不会隐式地转换为* 较大的数据类型. 也就是说, 要将一个 `Byte` 类型值赋给一个 `Int` 类型的变量需要进行显式类型转换:

```
val b: Byte = 1 // 这是 OK 的, 因为编译器会对字面值进行静态检查
// val i: Int = b // 编译错误
val i1: Int = b.toInt()
```

所有的数值类型都可以转换为其他类型:

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`

大多数情况下并不需要明确的的类型转换, 因为类型可以通过代码上下文自动推断得到, 而且数学运算符都进行了重载(overload), 可以适应各种数值类型的参数, 比如:

```
val l = 1L + 3 // Long 类型 + Int 类型, 结果为 Long 类型
```

数值类型的运算符(Operation)

Kotlin 对数值类型支持标准的数学运算符(operation): `+`, `-`, `*`, `/`, `%`. 这些运算符定义为相应的数值类上的成员函数:

```
fun main() {
//sampleStart
    println(1 + 2)
    println(2_500_000_000L - 1L)
    println(3.14 * 2.71)
    println(10.0 / 3)
//sampleEnd
}
```

你也可以对自己的类覆盖这些运算符. 详情请参见 操作符重载(Operator overloading) (操作符重载).

整数除法

整数值之间的除法返回的永远是整数值. 所有的小数部分都会被抛弃.

```
fun main() {
//sampleStart
    val x = 5 / 2
    //println(x == 2.5) // 错误: 不能在 'Int' 和 'Double' 类型值之间使用
'==' 操作符
    println(x == 2)
//sampleEnd
}
```

对任何两种整数类型之间的除法都是如此:

```
fun main() {
//sampleStart
    val x = 5L / 2
    println(x == 2L)
//sampleEnd
}
```

如果要返回浮点类型的结果, 需要将其中一个操作数显式转换为浮点类型:

```
fun main() {
//sampleStart
    val x = 5 / 2.toDouble()
    println(x == 2.5)
//sampleEnd
}
```

位运算符

Kotlin 对整数值提供了一组 *位运算符*. 这些运算符直接对数值的二进制表达的位(bit)进行操作. 位运算符表达为函数, 可以通过中缀表示法调用. 只能用于 `Int` 和 `Long`:

```
val x = (1 shl 2) and 0x000FF000
```

以下是位运算符的完整列表:

- `shl(bits)` – 带符号左移
- `shr(bits)` – 带符号右移
- `ushr(bits)` – 无符号右移
- `and(bits)` – 按位与(**AND**)
- `or(bits)` – 按位或(**OR**)
- `xor(bits)` – 按位异或(**XOR**)
- `inv()` – 按位取反

浮点值的比较

本节我们讨论的浮点值操作包括:

- 相等判断: `a == b` 以及 `a != b`
- 比较操作符: `a < b`, `a > b`, `a <= b`, `a >= b`
- 浮点值范围(Range) 的创建, 以及范围检查: `a..b`, `x in a..b`, `x !in a..b`

如果操作数 `a` 和 `b` 的类型能够静态地判定为 `Float` 或 `Double` (或者可为 null 值的 `Float?` 或 `Double?`), (比如, 类型明确声明为浮点值, 或者由编译器推断为浮点值, 或者通过智能类型转换 ("智能类型转换" in "类型检查与类型转换")变为浮点值), 那么此时对这些数值, 或由这些数值构成的范围的操作, 将遵循 IEEE 754 浮点数值运算标准 (https://en.wikipedia.org/wiki/IEEE_754).

但是, 为了支持使用泛型的情况, 并且支持完整的排序功能, 如果操作数 **不能** 静态地判定为浮点值类型, 那么判定结果会不同. 例如, `Any`, Comparable<...>, 或 Collection<T> 类型. 对于这样的情况, 对这些浮点值的操作将使用 `Float` 和 `Double` 类中实现的 `equals` 和 `compareTo` 方法. 因此判定结果是:

- `NaN` 会被判定为等于它自己
- `NaN` 会被判定为大于任何其他数值, 包括正无穷大(`POSITIVE_INFINITY`)
- `-0.0` 会被判定为小于 `0.0`

下面是一段的示例程序, 演示静态地判定为浮点值类型的操作数(`Double.NaN`) 与 **不能** 静态地判定为浮点值类型的操作数 (`listOf(T)`) 之间的动作差别.

```
fun main() {
    //sampleStart
    // 操作数静态地判定为浮点值类型
    println(Double.NaN == Double.NaN)                 // 输出结果为 false
    // 操作数 不能 静态地判定为浮点值类型
    // 因此 NaN 等于它自己
    println(listOf(Double.NaN) == listOf(Double.NaN)) // 输出结果为 true

    // 操作数静态地判定为浮点值类型
    println(0.0 == -0.0)                              // 输出结果为 true
    // 操作数 不能 静态地判定为浮点值类型
    // 因此 -0.0 小于 0.0
    println(listOf(0.0) == listOf(-0.0))              // 输出结果为 false

    println(listOf(Double.NaN, Double.POSITIVE_INFINITY, 0.0, -0.0).sorted())
    // 输出结果为 [-0.0, 0.0, Infinity, NaN]
```

```
    //sampleEnd
}
```

无符号整数(Unsigned Integer)类型

最终更新: 2024/09/10

除 整数类型 ("整数类型" in "数值类型") 外, Kotlin 还提供了以下无符号整数类型:

类型	大小 (位)	最小值	最大值
UByte	8 位	0	255
UShort	16 位	0	65,535
UInt	32 位	0	4,294,967,295 (2^{32} - 1)
ULong	64 位	0	18,446,744,073,709,551,615 (2^{64} - 1)

无符号整数支持有符号整数的大多数运算符.

> 无符号数值以 内联类 (内联的值类(Inline value class)) 的方式实现, 内部存储属性包含对应的同等宽度的有符号数值类型. 如果你想要在无符号和有符号的整数类型之间转换, 请确认更新了你的代码, 让所有的函数调用和操作都支持新的类型.

无符号整数的数组和值范围

> 无符号整数的数组以及对这些数组的操作目前处于 Beta (Kotlin 各部分组件的稳定性) 状态. 随时可能发生不兼容的变化. 使用时需要明确同意(Opt-in)(详情请参见下文).

与基本类型相同, 每一种无符号整数类型都有一个对应的类来表示由它构成的数组:

- UByteArray: 无符号 byte 构成的数组.
- UShortArray: 无符号 short 构成的数组.
- UIntArray: 无符号 int 构成的数组.

- ULongArray: 无符号 long 构成的数组.

与有符号的整数数组类类似, 这些无符号整数的数组类提供了与 Array 类相似的 API, 并且不会产生数值对象装箱带来的性能损耗.

使用无符号整数数组时, 会出现编译警告, 表示这个功能还未达到稳定状态. 要消除这个警告, 请使用 @ExperimentalUnsignedTypes 注解, 标注使用者同意(Opt-in). 你的代码的使用者是否也需要明确同意使用你的 API, 这一点由你来决定, 但请注意, 无符号整数数组还不是稳定的功能, 因此由于语言本身的变化, 使用它们的 API 可能会出现错误. 详情请参见 明确要求使用者同意的功能(Opt-in Requirement) (明确要求使用者同意的功能(Opt-in Requirement)).

为了支持 UInt 和 ULong 类型的 值范围与数列 (值范围(Range)与数列(Progression)) 功能, 还提供了 UIntRange, UIntProgression, ULongRange, ULongProgression 类.

无符号整数的字面值(literal)

为了无符号整数使用的便利, Kotlin 允许在整数字面值上添加后缀来表示特定的无符号类型 (与 Float 或 Long 的标记方式类似):

- u 和 U 用于标记无符号整数. 具体的无符号整数类型将根据程序此处期待的数据类型来决定. 如果未指定期待的数据类型, 编译器将根据整数值的大小来决定使用 UInt 或 ULong:

```
val b: UByte = 1u  // 字面值类型为 UByte, 因为程序指定了期待的数据类型
val s: UShort = 1u // 字面值类型为 UShort, 因为程序指定了期待的数据类型
val l: ULong = 1u  // 字面值类型为 ULong, 因为程序指定了期待的数据类型

val a1 = 42u // 字面值类型为 UInt: 因为程序未指定期待的数据类型, 而且整数值可以存入 UInt 内
val a2 = 0xFFFF_FFFF_FFFFu // 字面值类型为 ULong: 因为程序未指定期待的数据类型, 而且整数值无法存入 UInt 内
```

- uL 和 UL 将字面值明确标记为无符号的 long:

```
val a = 1UL // 字面值类型为 ULong, 即使这里未指定期待的数据类型, 而且整数值可以存入 UInt 内
```

使用场景

无符号数值的主要使用场景, 是利用整数的完整的二进制范围来表达正的数值. 比如, 要表达一个无法在有符号类型范围内表达的 16 进制常数, 例如 32 位 `AARRGGBB` 格式的颜色值:

```
data class Color(val representation: UInt)

val yellow = Color(0xFFCC00CCu)
```

你可以使用无符号数值来初始化字节数组, 而不需要明确的 `toByte()` 字面值转换:

```
val byteOrderMarkUtf8 = ubyteArrayOf(0xEFu, 0xBBu, 0xBFu)
```

另一种使用场景是与原生 API 交互. Kotlin 允许表达在方法签名中包含无符号类型的原生声明. 方法映射不会用有符号整数代替无符号整数, 保持语义无变化.

不适合的场景

尽管无符号整数只能表达正的数值或 0, 但在应用程序的业务逻辑中要求非负整数的情况下, 并不适合使用无符号整数. 例如, 用作集合大小或集合下标值的数据类型.

原因如下:

- 使用有符号的整数有助于发现数值溢出的异常情况, 以及标记错误条件, 比如 `List.lastIndex` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last-index.html) 对空的 List 返回结果是 -1.
- 无符号整数不能用作有符号整数的限定范围版本, 因为它们的值范围不是有符号整数值范围的子集. 有符号整数, 和无符号整数, 相互之间都不是子类型.

布尔(Boolean)类型

最终更新: 2024/09/10

`Boolean` 类型用来表示布尔型对象, 有两个可能的值: `true` 和 `false`. `Boolean` 还有对应的 可为 null ([Null 值安全性](#)) 的类型, 声明为 `Boolean?`.

> 在 JVM 平台, 布尔值保存为基本类型(Primitive Type) `boolean`, 通常使用 8 位.

布尔值的内建运算符有:

- `||` – 或运算 (逻辑 *或*)
- `&&` – 与运算 (逻辑 *与*)
- `!` – 非运算 (逻辑 *非*)

例如:

```
fun main() {
//sampleStart
    val myTrue: Boolean = true
    val myFalse: Boolean = false
    val boolNull: Boolean? = null

    println(myTrue || myFalse)
    // 输出结果为 true
    println(myTrue && myFalse)
    // 输出结果为 false
    println(!myTrue)
    // 输出结果为 false
    println(boolNull)
    // 输出结果为 null
//sampleEnd
}
```

`||` 和 `&&` 运算符会进行短路计算, 也就是说:

- 如果第一个操作数为 `true`, `||` 运算符不会计算第二个操作数.
- 如果第一个操作数为 `false`, `&&` 运算符不会计算第二个操作数.

> 在 JVM 平台, 可为 null 的布尔对象引用会被装箱(box)为 Java 类, 与 数值类型 ("数值类型在 JVM 平台的内部表达" in "数值类型") 一样.

字符

最终更新: 2024/09/10

字符使用 `Char` 类型表达. 字符的字面值(literal)使用单引号表达: `'1'`.

> 在 JVM 平台, 字符保存为基本类型(Primitive Type): `char`, 表示一个 16 位的 Unicode 字符.

特殊字符使用反斜线转义表达. Kotlin 支持的转义字符包括:

- `\t` – 制表符(TAB)
- `\b` – 退格
- `\n` – 换行 (LF)
- `\r` – 回车 (CR)
- `\'` – 单引号
- `\"` – 双引号
- `\\` – 反斜线
- `\$` – 美元符号

其他任何字符, 都可以使用 Unicode 转义表达方式: `'\uFF00'`.

```
fun main() {
//sampleStart
    val aChar: Char = 'a'

    println(aChar)
    println('\n') // 打印一个额外的换行符
    println('\uFF00')
//sampleEnd
}
```

如果字符的值是数字, 可以使用 `digitToInt()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/digit-to-int.html) 函数显式转换为 `Int` 值.

> 在 JVM 平台, 当需要一个可为 null 的字符引用时, 字符会被装箱(box)为 Java 类, 与 数值类型 ("数值类型在 JVM 平台的内部表达" in "数值类型") 一样. 装箱操作不保持对象的同一性(identity).

字符串

最终更新: 2024/09/10

Kotlin 中的字符串由 `String` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/) 类型表达.

> 在 JVM 平台, 使用 UTF-16 编码的 `String` 类型的对象, 大约使用每字符 2 个字节.

一般来说, 字符串值是一系列字符, 用双引号(`"`)括起:

```
val str = "abcd 123"
```

字符串中的元素是字符, 你可以通过下标操作符来访问: `s[i]`. 你可以使用 `for` 循环来遍历这些字符:

```
fun main() {
    val str = "abcd"
//sampleStart
    for (c in str) {
        println(c)
    }
//sampleEnd
}
```

字符串是不可变的. 一旦初始化之后, 将不能改变它的值, 也不能为它赋予一个新的值. 所有改变字符串内容的操作, 返回值都是新的 `String` 对象, 而操作对象的原字符串不会改变:

```
fun main() {
//sampleStart
    val str = "abcd"

    // 创建一个新的 String 对象, 并打印
    println(str.uppercase())
    // 输出结果为 ABCD

    // 原字符串保持原来的值不变
    println(str)
```

```
    // 输出结果为 abcd
//sampleEnd
}
```

要拼接字符串, 可以使用 `+` 操作符. 这个操作符也可以将字符串与其他数据类型的值拼接起来, 只要表达式中的第一个元素是字符串类型:

```
fun main() {
//sampleStart
    val s = "abc" + 1
    println(s + "def")
    // 输出结果为 abc1def
//sampleEnd
}
```

> 大多数情况下, 字符串拼接处理应该使用 字符串模板 或 多行字符串(Multiline String).

字符串的字面值(literal)

Kotlin 中存在两种字符串字面值:

- 转义(Escaped)字符串
- 多行(Multiline)字符串

转义(Escaped)字符串

转义(Escaped)字符串 可以包含转义字符. 转义字符串的示例如下:

```
val s = "Hello, world!\n"
```

转义字符使用通常的反斜线(`\`)方式表示. 关于 Kotlin 支持的转义字符, 请参见 字符 ([字符](#)).

多行(Multiline)字符串

多行(Multiline)字符串 可以包含换行符和任意文本. 由三重引号表示(`"""`), 其内容不转义, 可以包含换行符和任意字符:

```
val text = """
    for (c in "foo")
        print(c)
"""
```

要删除多行字符串的前导空白(leading whitespace), 可以使用 `trimMargin()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/trim-margin.html) 函数:

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

默认情况下, 会使用管道符号 `|` 作为前导空白的标记前缀, 但你可以通过参数指定使用其它字符, 比如 `trimMargin(">")`.

字符串模板

字符串字面值内可以包含 *模板表达式*, 它是一小段代码, 会被执行, 其计算结果将被拼接为字符串内容的一部分. 模板表达式以 `$` 符号开始, `$` 符号之后可以是一个变量名:

```
fun main() {
//sampleStart
    val i = 10
    println("i = $i")
    // 输出结果为 i = 10
//sampleEnd
}
```

`$` 符号之后也可以是表达式, 由大括号括起:

```
fun main() {
//sampleStart
    val s = "abc"
    println("$s.length is ${s.length}")
    // 输出结果为 abc.length is 3
```

```
//sampleEnd
}
```

在多行字符串(Multiline String)和转义字符串(Escaped String)中都可以使用模板. 由于多行字符串不能使用反斜线转义表达方式, 如果要在字符串中的任何符号之前插入美元符号 `$` 本身 (`$` 可以用作 标识符 (https://kotlinlang.org/docs/reference/grammar.html#identifiers) 的开始字符), 可以使用以下语法:

```
val price = """
${'$'}_9.99
"""
```

字符串格式化

> 使用 `String.format()` 函数进行字符串格式化, 只能用于 Kotlin/JVM 平台.

如果要按照你的需求来格式化一个字符串, 可以使用 `String.format()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/format.html) 函数.

`String.format()` 函数接受一个格式字符串, 以及一个或多个参数. 格式字符串对每个参数包含一个占位符(通过 `%` 表达), 之后是格式说明符. 格式说明符是针对对应参数的格式指令, 由符号, 宽度, 精度以及转换类型组成. 总的来说, 格式说明符决定了输出的格式. 通用的格式说明符包括: `%d` 用于整数, `%f` 用于浮点数, 以及 `%s` 用于字符串. 你还可以使用 `argument_index$` 语法, 在格式字符串中, 使用不同的格式多次引用同一个参数.

> 关于格式字符串的详细解释, 以及它的完整列表, 请参见 Java Formatter 类的文档 (https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#summary).

我们来看一个示例程序:

```
fun main() {
//sampleStart
    // 格式化 1 个整数, 添加前导的 0, 使结果长度为 7 个字符
    val integerNumber = String.format("%07d", 31416)
    println(integerNumber)
    // 输出结果为 0031416
```

```
    // 格式化 1 个浮点数，显示正负号，保留 4 位小数
    val floatNumber = String.format("%+.4f", 3.141592)
    println(floatNumber)
    // 输出结果为 +3.1416

    // 格式化 2 个字符串，显示为大写文字，每个字符串使用一个占位符
    val helloString = String.format("%S %S", "hello", "world")
    println(helloString)
    // 输出结果为 HELLO WORLD

    // 格式化 1 个负数，包含在括号中，然后使用 `argument_index$`，以不同的
格式输出同一个数字 (没有括号).
    val negativeNumberInParentheses = String.format("%(d means
%1\$d", -31416)
    println(negativeNumberInParentheses)
    //输出结果为 (31416) means -31416
//sampleEnd
}
```

`String.format()` 函数提供了与字符串模板类似的功能. 但是, `String.format()` 函数的功能要更多一些, 因为可以使用更多的格式选项.

此外, 可以通过变量来指定格式字符串. 当格式字符串本身可变时, 这是很有用的功能 例如, 在根据用户的语言设定进行本地化翻译时.

使用 `String.format()` 函数时要小心, 因为在参数与对应的占位符之间, 很容易写错它们的个数或位置.

数组

最终更新: 2024/09/10

数组是一种数据结构, 其中包含固定数量的值, 所有的值为同一个类型, 或这个类型的子类型. Kotlin 中最常见的数组类型是对象类型的数组, 使用 `Array` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-array/) 类表达.

> 如果你在对象类型的数组中使用基本类型(Primitive Type), 会造成性能损失, 因为你的基本类型会被 装箱 (https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html) 为对象. 要避免这种装箱造成的性能损失, 请使用 基本类型数组.

什么时候使用数组

当你需要满足某些特殊的低层级要求时, 可以在 Kotlin 中使用数组. 例如, 如果你的性能需求超过了通常的应用程序的需求, 或者需要构建自定义数据结构的情况. 如果你没有这种类型的限制, 请使用 集合(Collection) (集合(Collection)概述).

集合与数组相比, 有以下优点:

- 集合是只读的, 因此给了你更多的控制权, 使你能够编写意图清晰的, 更加健壮的代码.
- 更容易对集合添加或删除元素. 与此相反, 数组的大小是固定的. 要对数组添加或删除元素, 只能每次创建新的数组, 这是非常效率低下的:

```
fun main() {
//sampleStart
    var riversArray = arrayOf("Nile", "Amazon", "Yangtze")

    // 使用 += 赋值操作创建新的 riversArray,
    // 复制原来的元素, 并添加 "Mississippi"
    riversArray += "Mississippi"
    println(riversArray.joinToString())
    // 输出结果为 Nile, Amazon, Yangtze, Mississippi
//sampleEnd
}
```

- 你可以使用相等操作符(`==`) 来检查两个集合是否结构相等(Structurally Equal). 但不能对数组使用这个操作符. 相反, 你需要使用特殊的函数, 详情请参见 比较数组.

关于集合, 详情请参见 集合概述 ([集合(Collection)概述](#)).

创建数组

在 Kotlin 中要创建数组, 你可以使用:

- 函数, 例如 `arrayOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/array-of.html), `arrayOfNulls()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/array-of-nulls.html#kotlin$arrayOfNulls(kotlin.Int)) 或 `emptyArray()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/empty-array.html).
- `Array` 构造器.

下面的示例使用 `arrayOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/array-of.html) 函数, 并将数组元素的值传递给它:

```
fun main() {
//sampleStart
    // 使用元素值 [1, 2, 3] 创建数组
    val simpleArray = arrayOf(1, 2, 3)
    println(simpleArray.joinToString())
    // 输出结果为 1, 2, 3
//sampleEnd
}
```

下面的示例使用 `arrayOfNulls()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/array-of-nulls.html#kotlin$arrayOfNulls(kotlin.Int)) 函数创建指定大小的数组, 并使用 `null` 元素填充数组:

```
fun main() {
//sampleStart
    // 使用元素值 [null, null, null] 创建数组
    val nullArray: Array<Int?> = arrayOfNulls(3)
    println(nullArray.joinToString())
    // 输出结果为 null, null, null
```

```
//sampleEnd
}
```

下面的示例使用 `emptyArray()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/empty-array.html) 函数创建空数组:

```
var exampleArray = emptyArray<String>()
```

> 由于 Kotlin 的类型推断功能, 在赋值语句的左侧或右侧都可以指定空数组的类型.
>
> 例如:
>
> ```
> var exampleArray = emptyArray<String>()
>
> var exampleArray: Array<String> = emptyArray()
> ```

`Array` 构造器的参数是, 数组大小, 以及一个函数, 这个函数对指定的数组下标返回对应的元素值:

```
fun main() {
//sampleStart
    // 创建一个 Array<Int>, 初始化为 0 值: [0, 0, 0]
    val initArray = Array<Int>(3) { 0 }
    println(initArray.joinToString())
    // 输出结果为 0, 0, 0

    // 创建一个 Array<String>, 初始化为 ["0", "1", "4", "9", "16"]
    val asc = Array(5) { i -> (i * i).toString() }
    asc.forEach { print(it) }
    // 输出结果为 014916
//sampleEnd
}
```

> 与大多数编程语言一样, 在 Kotlin 中, 数组下标从 0 开始.

嵌套的数组

数组可以相互嵌套, 创建多维数组:

```
fun main() {
//sampleStart
    // 创建一个 2 维数组
    val twoDArray = Array(2) { Array<Int>(2) { 0 } }
    println(twoDArray.contentDeepToString())
    // 输出结果为 [[0, 0], [0, 0]]

    // 创建一个 3 维数组
    val threeDArray = Array(3) { Array(3) { Array<Int>(3) { 0 } } }
    println(threeDArray.contentDeepToString())
    // 输出结果为 [[[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [0,
0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 0]]]
//sampleEnd
}
```

> 嵌套的数组不需要类型相同, 也不需要大小相同.

访问和修改元素

数组永远是可以修改的. 要访问和修改数组中的元素, 请使用 下标访问操作符 ("下标访问操作符" in "操作符重载")`[]`:

```
fun main() {
//sampleStart
    val simpleArray = arrayOf(1, 2, 3)
    val twoDArray = Array(2) { Array<Int>(2) { 0 } }

    // 访问并修改元素
    simpleArray[0] = 10
    twoDArray[0][0] = 2

    // 输出修改后的元素
    println(simpleArray[0].toString()) // 输出结果为 10
    println(twoDArray[0][0].toString()) // 输出结果为 2
```

```
//sampleEnd
}
```

Kotlin 中的数组是 *不可变的(invariant)*. 这意味着 Kotlin 不允许你将一个 `Array<String>` 赋值给一个 `Array<Any>`, 以防止发生运行时错误. 相反, 你可以使用 `Array<out Any>`. 更多详情请参见, 类型投射 ("类型投射(Type projection)" in "泛型(Generic): in, out, where").

使用数组

在 Kotlin 中, 你可以使用数组, 向一个函数传递不定数量的参数, 或对数组元素本身执行操作. 例如, 比较数组, 变换数组内容, 或转换为集合.

向一个函数传递不定数量的参数

在 Kotlin 中, 你可以通过 `vararg` ("不定数量参数(varargs)" in "函数") 参数, 向一个函数传递不定数量的参数. 如果你不能预先知道参数的数量, 这个功能是很有用的, 例如格式化消息, 或者创建 SQL 查询的情况.

要向一个函数传递一个数组, 其中包含不定数量的参数, 请使用 *展开(spread)* 操作符 (`*`). 展开操作符会将数组的每个元素作为独立的参数传递给指定的函数:

```
fun main() {
    val lettersArray = arrayOf("c", "d")
    printAllStrings("a", "b", *lettersArray)
    // 输出结果为 abcd
}

fun printAllStrings(vararg strings: String) {
    for (string in strings) {
        print(string)
    }
}
```

更多详情请参见 不定数量参数(varargs) ("不定数量参数(varargs)" in "函数").

比较数组

要比较两个数组是否包含相同的元素, 并且顺序也相同, 请使用 `.contentEquals()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/content-equals.html) 和 `.contentDeepEquals()`

(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/content-deep-equals.html) 函数:

```
fun main() {
//sampleStart
    val simpleArray = arrayOf(1, 2, 3)
    val anotherArray = arrayOf(1, 2, 3)

    // 比较数组内容
    println(simpleArray.contentEquals(anotherArray))
    // 输出结果为 true

    // 使用中缀标记法(Infix notation), 在一个元素发生变化之后, 再次比较数组
内容
    simpleArray[0] = 10
    println(simpleArray contentEquals anotherArray)
    // 输出结果为 false
//sampleEnd
}
```

> ⚠ 不要使用相等 (`==`) 和不等 (`!=`) 操作符 ("结构相等" in "相等判断") 来比较数组内容. 这些操作符会检查赋值的变量是否指向相同的对象.
>
> 关于 Kotlin 中数组的行为为什么会如此, 详情请参见 这篇 blog (https://blog.jetbrains.com/kotlin/2015/09/feedback-request-limitations-on-data-classes/#Appendix.Comparingarrays).

变换数组

Kotlin 有很多有用的函数, 可以对数组进行变换. 这篇文档重点介绍少数几个函数, 但并不是完整的功能列表. 关于所有函数的完整列表, 请参见我们的 API 参考文档 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-array/).

求和

要得到一个数组中所有元素的和, 请使用 `.sum()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum.html) 函数:

```
fun main() {
//sampleStart
```

```
    val sumArray = arrayOf(1, 2, 3)

    // 对数组元素求和
    println(sumArray.sum())
    // 输出结果为 6
//sampleEnd
}
```

> .sum() 函数只能用于 数值类型 (数值类型) 的数组, 例如 Int.

随机打乱

要随机打乱数组中的元素, 请使用 .shuffle() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/shuffle.html) 函数:

```
fun main() {
//sampleStart
    val simpleArray = arrayOf(1, 2, 3)

    // 随机打乱元素 [3, 2, 1]
    simpleArray.shuffle()
    println(simpleArray.joinToString())

    // 再次随机打乱元素 [2, 3, 1]
    simpleArray.shuffle()
    println(simpleArray.joinToString())
//sampleEnd
}
```

将数组转换为集合

如果你同时使用不同的 API, 其中一些使用数组, 另一些使用集合, 那么你可以将数组转换为 集合 (集合(Collection)概述), 也可以反过来将集合转换为数组.

转换为 List 或 Set

要将数组转换为 List 或 Set, 请使用 .toList() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-list.html) 和 .toSet() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-set.html) 函数.

```
fun main() {
//sampleStart
    val simpleArray = arrayOf("a", "b", "c", "c")

    // 转换为 Set
    println(simpleArray.toSet())
    // 输出结果为 [a, b, c]

    // 转换为 List
    println(simpleArray.toList())
    // 输出结果为 [a, b, c, c]
//sampleEnd
}
```

转换为 Map

要将数组转换为 `Map`, 请使用 `.toMap()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-map.html) 函数.

只有元素类型为 `Pair<K,V>` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-pair/) 的数组能够转换为 `Map`. `Pair` 实例的第 1 个值成为键(key), 第 2 个值成为值(value). 下面的示例使用 中缀标记法(Infix notation) ("中缀标记法(Infix notation)" in "函数") 来调用 `to` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/to.html) 函数, 创建 `Pair` 的元祖:

```
fun main() {
//sampleStart
    val pairArray = arrayOf("apple" to 120, "banana" to 150,
"cherry" to 90, "apple" to 140)

    // 转换为 Map
    // 键(key)是水果, 值(value)是它们的卡路里数量
    // 注意, 键必须是唯一的, 因此最后一个 "apple" 的值会覆盖第一个的值
    println(pairArray.toMap())
    // 输出结果为 {apple=140, banana=150, cherry=90}

//sampleEnd
}
```

基本类型(Primitive Type)数组

如果你使用 Array 类来存储基本类型(Primitive Type), 这些元素值会被装箱为对象. 另一种选择是, 你可以使用基本类型数组, 它可以让你在数组中存储基本类型, 而不会发生装箱操作导致的性能损失副作用:

基本类型数组	相当于 Java 中的类型
BooleanArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-boolean-array/)	boolean[]
ByteArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-byte-array/)	byte[]
CharArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-char-array/)	char[]
DoubleArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-double-array/)	double[]
FloatArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-float-array/)	float[]
IntArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-int-array/)	int[]
LongArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-long-array/)	long[]
ShortArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-short-array/)	short[]

这些类与 Array 类没有继承关系, 但它们有相同的一组函数和属性.

下面的示例创建一个 IntArray 类的实例:

```kotlin
fun main() {
//sampleStart
    // 创建一个数组, 元素类型为 Int, 大小为 5, 元素值为 0
    val exampleArray = IntArray(5)
    println(exampleArray.joinToString())
    // 输出结果为 0, 0, 0, 0, 0
//sampleEnd
}
```

> 要将基本类型数组转换为对象类型数组, 请使用 .toTypedArray() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-typed-array.html) 函数.
>
> 要将对象类型数组转换为基本类型数组, 请使用 .toBooleanArray() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-boolean-array.html), .toByteArray() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-byte-array.html), .toCharArray() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-char-array.html), 等等函数.

下一步做什么?

- 为什么对大多数使用场景我们推荐使用集合, 请阅读我们的 集合概述 (集合(Collection)概述).
- 学习其他 基本类型 (基本类型).
- 如果你是 Java 开发者, 请阅读我们的 Java 到 Kotlin 迁移向导, 关于 集合 (Java 和 Kotlin 中的集合(Collection)) 的部分.

类型检查与类型转换

最终更新: 2024/09/10

在 Kotlin 中, 你可以进行类型检查, 在运行时检查一个对象的类型. 类型转换可以将对象转换为另一个类型.

> ⚠ 关于 **泛型** 的类型检查和转换, 例如 `List<T>`, `Map<K,V>`, 请参见 泛型的类型检查和转换 ("泛型的类型检查与类型转换" in "泛型(Generic): in, out, where").

is 与 !is 操作符

可以使用 `is` 操作符, 或者它相反的 `!is` 操作符, 在运行时检查一个对象与一个给定的类型是否一致:

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // 等价于 !(obj is String)
    print("Not a String")
} else {
    print(obj.length)
}
```

智能类型转换

大多数情况下, 在 Kotlin 中你不必使用显式的类型转换操作, 因为编译器会对不可变值的 `is` 检查和显式的类型转换 进行追踪, 然后在需要的时候自动插入(安全的)类型转换:

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x 被自动转换为 String 类型
    }
}
```

如果一个相反的类型检查导致了 return, 此时编译器足够智能, 可以判断出转换处理是安全的:

```
if (x !is String) return

print(x.length) // x 被自动转换为 String 类型
```

对于 `&&` 和 `||` 操作符, 如果在操作符左侧进行了适当的(通常的或相反的)类型检查, 那么操作符的右侧也是如此:

```
// 在 `||` 的右侧, x 被自动转换为 String 类型
if (x !is String || x.length == 0) return

// 在 `&&` 的右侧, x 被自动转换为 String 类型
if (x is String && x.length > 0) {
    print(x.length) // x 被自动转换为 String 类型
}
```

智能类型转换(Smart Cast) 对于 `when` 表达式 ("when 表达式" in "条件与循环") 和 `while` 循环 ("while 循环" in "条件与循环") 同样有效:

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

> ⚠ 注意, 在类型检查语句与变量使用语句之间, 只有在编译器能够确保变量不会改变的情况下, 智能类型转换才是有效的.

在以下条件下可以使用智能类型转换:

val 局部变量	永远有效, 但 局部的委托属性 (委托属性) 例外.
val 属性	如果属性是 private 的, 或 internal 的, 或者类型检查处理与属性定义出现在同一个 模块(module) ("模块(Module)" in "可见度修饰符") 内, 那么智能类型转换是有效的. 对于 open 属性, 或存在自定义 get 方法的属性, 智能类型转换是无效的.
var 局部变量	如果在类型检查语句与变量使用语句之间, 变量没有被改变, 而且它没有被 Lambda 表达式捕获并在 Lambda 表达式内修改它, 并且它不是一个局部的委托属性, 那么智能类型转换是有效的.
var 属性	永远无效, 因为其他代码随时可能改变变量值.

"不安全的" 类型转换操作符

如果类型转换不成功, 类型转换操作符通常会抛出一个异常. 因此, 称为 *不安全的(unsafe)* 类型转换. 在 Kotlin 中, 不安全的类型转换使用中缀操作符 as:

```
val x: String = y as String
```

注意, null 不能被转换为 String, 因为这个类型不是 可为 null 的(nullable) (Null 值安全性). 如果 y 为 null, 上例中的代码将抛出一个异常. 为了让这段代码能够处理 null 值, 需要在类型转换操作符的右侧使用可为 null 的类型:

```
val x: String? = y as String?
```

"安全的" (nullable) 类型转换操作

为了避免抛出异常, 请使用 *安全的* 类型转换操作符 as?, 当类型转换失败时, 它会返回 null.

```
val x: String? = y as? String
```

注意, 尽管 `as` 操作符的右侧是一个非 null 的 `String` 类型, 但这个转换操作的结果仍然是可为 null 的.

条件与循环

最终更新: 2024/09/10

if 表达式

在 Kotlin 中, `if` 是一个表达式: 它有返回值. 因此, Kotlin 中没有三元运算符(`条件 ? then 分支返回值 : else 分支返回值`), 因为简单的 `if` 表达式完全可以实现同样的任务.

```
fun main() {
    val a = 2
    val b = 3

    //sampleStart
    var max = a
    if (a < b) max = b

    // 使用 else 分支的方式
    if (a > b) {
        max = a
    } else {
        max = b
    }

    // if 作为表达式使用
    max = if (a > b) a else b

    // 在表达式中也可以使用 `else if`:
    val maxLimit = 1
    val maxOrLimit = if (maxLimit > a) maxLimit else if (a > b) a
else b

    //sampleEnd
    println("max is $max")
    println("maxOrLimit is $maxOrLimit")
}
```

if 表达式的分支可以是多条语句组成的代码段, 这种情况下, 代码段内最后一个表达式的值将成为整个代码段的返回值:

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

如果你将 if 作为表达式来使用, 比如, 将它的值作为函数的返回值, 或将它的值赋值给一个变量, 这种情况下必须存在 else 分支.

when 表达式

when 表示一个条件表达式, 带有多个分支. 类似于各种 C 风格语言中的 switch 语句. 它的最简单形式如下.

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

when 语句会将它的参数与各个分支逐个匹配, 直到找到某个分支的条件成立.

when 可以用作表达式, 也可以用作流程控制语句. 如果用作表达式, 第一个匹配成功的分支的返回值将成为整个表达式的值. 如果用作流程控制语句, 各个分支的返回值将被忽略. 与 if 类似, 各个分支可以是多条语句组成的代码段, 代码段内最后一个表达式的值将成为整个代码段的值.

如果其他所有分支的条件都不成立, 则会执行 else 分支.

如果 when 被用作 *表达式*, 则必须存在 else 分支, 除非编译器能够证明其他分支的条件已经覆盖了所有可能的情况, 比如, 使用 枚举(enum)类 (枚举类) 的常数 或 封闭(sealed)类 (封闭类(Sealed Class)与封闭接口(Sealed Interface)) 的子类型.

```
enum class Bit {
    ZERO, ONE
}

val numericValue = when (getRandomBit()) {
    Bit.ZERO -> 0
    Bit.ONE -> 1
    // 不需要 'else' 分支, 因为已经覆盖了所有可能的情况
}
```

在 when *语句* 中, 对于以下情况, else 分支是必须的:

- when 的判断对象是 Boolean, 枚举(enum)类 (枚举类), 或 封闭(sealed)类 (封闭类(Sealed Class)与封闭接口(Sealed Interface)) 类型, 或它们的可为 null 类型.
- when 的分支没有覆盖所有可能的情况.

```
enum class Color {
    RED, GREEN, BLUE
}

when (getColor()) {
    Color.RED -> println("red")
    Color.GREEN -> println("green")
    Color.BLUE -> println("blue")
    // 不需要 'else' 分支, 因为已经覆盖了所有可能的情况
}

when (getColor()) {
    Color.RED -> println("red") // 没有针对 GREEN 和 BLUE 的分支
    else -> println("not red") // 需要 'else' 分支
}
```

如果对多种条件需要进行相同的处理, 那么可以指定多个条件, 用逗号分隔:

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
```

```
    else -> print("otherwise")
}
```

在分支条件中,你可以使用任意的表达式(而不仅仅是常数值)

```
when (x) {
    s.toInt() -> print("s encodes x")
    else -> print("s does not encode x")
}
```

你还可以使用 `in` 或 ``!in` 来检查一个值是否属于一个 范围 (值范围(Range)与数列(Progression)), 或者检查是否属于一个集合:

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

还可以使用 `is` 或 `!is` 来检查一个值是不是某个类型. 注意, 由于 Kotlin 的 智能类型转换 ("智能类型转换" in "类型检查与类型转换") 功能, 进行过类型判断之后, 你就可以直接访问这个类型的方法和属性, 而不必再进行显式的类型检查.

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

`when` 也可以用来替代 `if`-`else` `if` 串. 如果没有指定参数, 那么所有的分支条件都应该是单纯的布尔表达式, 当条件的布尔表达式值为 true 时, 就会执行对应的分支:

```
when {
    x.isOdd() -> print("x is odd")
    y.isEven() -> print("y is even")
    else -> print("x+y is odd")
}
```

你可以使用下面这种语法, 将 `when` 语句的判断对象保存到一个变量中:

```
fun Request.getBody() =
    when (val response = executeRequest()) {
        is Success -> response.body
        is HttpError -> throw HttpException(response.status)
    }
```

由 `when` 引入的这个变量, 它的有效范围仅限于 `when` 语句之内.

for 循环

任何值, 只要能够产生一个迭代器(iterator), 就可以使用 `for` 循环进行遍历. 相当于 C# 等语言中的 `foreach` 循环. `for` 循环的语法如下:

```
for (item in collection) print(item)
```

`for` 循环体可以是多条语句组成的代码段.

```
for (item: Int in ints) {
    // ...
}
```

前面提到过, 凡是能够产生迭代器(iterator)的值, 都可以使用 `for` 进行遍历. 也就是说, 遍历对象需要满足以下条件:

- 存在一个成员函数或扩展函数 `iterator()`, 它的返回类型应该是 `Iterator<>` 类型, 并且这个 `Iterator<>` 类型应该:
  - 存在一个成员函数或扩展函数 `next()`
  - 存在一个成员函数或扩展函数 `hasNext()`, 它的返回类型为 `Boolean` 类型.

上述三个函数都需要标记为 `operator`.

要遍历一个数值范围, 可以使用 值范围表达式 (值范围(Range)与数列(Progression)):

```
fun main() {
//sampleStart
    for (i in 1..3) {
```

```
        println(i)
    }
    for (i in 6 downTo 0 step 2) {
        println(i)
    }
//sampleEnd
}
```

使用 for 循环来遍历数组或值范围(Range)时, 会被编译为基于数组下标的循环, 不会产生迭代器(iterator)对象.

如果你希望使用下标变量来遍历数组或 List, 可以这样做:

```
fun main() {
    val array = arrayOf("a", "b", "c")
//sampleStart
    for (i in array.indices) {
        println(array[i])
    }
//sampleEnd
}
```

或者, 你也可以使用 withIndex 库函数:

```
fun main() {
    val array = arrayOf("a", "b", "c")
//sampleStart
    for ((index, value) in array.withIndex()) {
        println("the element at $index is $value")
    }
//sampleEnd
}
```

while 循环

while 和 do-while 循环会在满足条件时反复执行它们的循环体. 但它们检查循环条件的时刻不同:

- while 先检查条件, 并在条件满足时, 执行循环体, 然后再次跳回到条件检查.

- `do-while` 先执行循环体, 然后再检查条件. 如果条件满足, 就会继续循环. 因此 `do-while` 的循环体至少会执行一次, 无论条件是否成立.

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

循环的中断(break)与继续(continue)

Kotlin 的循环支持传统的 `break` 和 `continue` 操作符. 详情请参见 返回与跳转 ([返回与跳转: break 与 continue]).

返回与跳转: break 与 continue

最终更新: 2024/09/10

Kotlin 中存在 3 种跳出程序流程的表达式:

- `return` 的默认行为是, 从最内层的函数或 匿名函数 ("匿名函数(Anonymous Function)" in "高阶函数与 Lambda 表达式") 中返回.
- `break` 结束最内层的循环.
- `continue` 在最内层的循环中, 跳转到下一次循环.

所有这些表达式都可以用作更大的表达式的一部分:

```
val s = person.name ?: return
```

这些表达式的类型都是 Nothing 类型 ("Nothing 类型" in "异常(Exception)").

Break 和 Continue 的位置标签

Kotlin 中的任何表达式都可以用 *label* 标签来标记. 标签由标识符后面加一个 `@` 符号构成, 比如 `abc@`, `fooBar@`. 要给一个表达式标记标签, 只需要将标签放在它之前.

```
loop@ for (i in 1..100) {
    // ...
}
```

然后, 我们就可以使用标签来限定 `break` 或 `continue` 的跳转对象:

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

通过标签限定后, `break` 语句, 将会跳转到这个标签标记的循环语句之后. `continue` 语句则会跳转到循环语句的下一次循环.

使用标签控制 return 的目标

在 Kotlin 中, 通过使用字面值函数(function literal), 局部函数(local function), 以及对象表达式(object expression), 可以实现函数的嵌套. 通过标签限定的 `return` 语句, 可以从一个外层函数中返回. 最重要的使用场景是从 Lambda 表达式中返回. 回忆一下我们曾经写过以下代码, `return` 表达式会从最内层的函数 `foo` 中返回:

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return // 非局部的返回(non-local return), 直接返回到 foo() 函数的调用者
        print(it)
    }
    println("this point is unreachable")
}
//sampleEnd

fun main() {
    foo()
}
```

注意, 这种非局部的返回(non-local return), 仅对传递给 内联函数(inline function) ([内联函数 (Inline Function)](#)) 的 Lambda 表达式有效. 如果需要从 Lambda 表达式返回, 可以对它标记一个标签, 然后使用这个标签来指明 `return` 的目标:

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit // 局部的返回(local return), 返回到 Lambda 表达式的调用者: 返回到 forEach 循环
        print(it)
    }
    print(" done with explicit label")
}
//sampleEnd

fun main() {
```

```
    foo()
}
```

这样, `return` 语句就只从 Lambda 表达式中返回. 通常, 使用 *隐含标签* 会更方便一些, 因为隐含标签的名称与 Lambda 表达式被传递去的函数名称相同.

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return@forEach // 局部的返回(local return), 返回到
Lambda 表达式的调用者: 返回到 forEach 循环
        print(it)
    }
    print(" done with implicit label")
}
//sampleEnd

fun main() {
    foo()
}
```

另一种方法是, 你也可以使用 匿名函数 ("匿名函数(Anonymous Function)" in "高阶函数与 Lambda 表达式") 来替代 Lambda 表达式. 匿名函数内的 `return` 语句会从匿名函数内返回.

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach(fun(value: Int) {
        if (value == 3) return  // 局部的返回(local return), 返回到匿名
函数的调用者: 返回到 forEach 循环
        print(value)
    })
    print(" done with anonymous function")
}
//sampleEnd

fun main() {
    foo()
}
```

注意, 上面三个例子中局部返回的使用, 都与通常的循环中的 `continue` 关键字的使用很类似.

不存在与 `break` 直接等价的语法, 但可以模拟出来, 方法是增加一个嵌套的 Lambda 表达式, 然后在它内部使用非局部的返回:

```
//sampleStart
fun foo() {
    run loop@{
        listOf(1, 2, 3, 4, 5).forEach {
            if (it == 3) return@loop // 非局部的返回(non-local
return), 从传递给 run 函数的 Lambda 表达式中返回
            print(it)
        }
    }
    print(" done with nested loop")
}
//sampleEnd

fun main() {
    foo()
}
```

当 return 语句指定了返回值时, 源代码解析器会将这样的语句优先识别为使用标签限定的 return 语句:

```
return@a 1
```

这里的含义是 "返回到标签 `@a` 处, 返回值为 `1` ", 而不是 "返回一个带标签的表达式 `(@a 1)` ".

异常(Exception)

最终更新: 2024/09/10

异常类

Kotlin 中所有的异常类都继承自 `Throwable` 类. 每个异常都带有错误消息, 调用堆栈, 以及可选的错误原因.

要抛出异常, 可以使用 `throw` 表达式:

```
fun main() {
//sampleStart
    throw Exception("Hi There!")
//sampleEnd
}
```

要捕获异常, 可以使用 `try`... `catch` 表达式:

```
try {
    // 某些代码
} catch (e: SomeException) {
    // 异常处理
} finally {
    // 可选的 finally 代码段
}
```

`try` 表达式中可以有 0 个或多个 `catch` 代码段, `finally` 代码段可以省略. 但是, `catch` 或 `finally` 代码段总计至少要出现一个.

Try 是一个表达式

`try` 是一个表达式, 也就是说, 它可以有返回值:

```
val a: Int? = try { input.toInt() } catch (e: NumberFormatException)
{ null }
```

`try` 表达式的返回值, 要么是 `try` 代码段内最后一个表达式的值, 要么是 `catch` 代码段内最后一个表

达式的值. `finally` 代码段的内容不会影响 `try` 表达式的结果值.

受控异常(Checked Exception)

Kotlin 中不存在受控异常(checked exception). 原因有很多, 我们举一个简单的例子.

下面的例子是 JDK 中 `StringBuilder` 类所实现的一个接口:

```
Appendable append(CharSequence csq) throws IOException;
```

这个方法签名代表, 每次我想要将一个字符串追加到某个对象(比如, 一个 `StringBuilder`, 某种 log, 控制台, 等等), 我都必须要捕获 `IOException` 异常. 为什么? 因为这个方法的具体实现有可能会执行 IO 操作 (比如 `Writer` 类也会实现 `Appendable` 接口). 因此就导致我们的程序中充满了这样的代码:

```
try {
    log.append(message)
} catch (IOException e) {
    // 实际上前面的代码必然是安全的
}
```

这样的结果就很不好, 请参见 Effective Java, 第 3 版 (https://www.oracle.com/technetwork/java/effectivejava-136174.html), 第 77 条: *不要忽略异常*.

Bruce Eckel 对受控异常评论说:

> 在小程序中的试验证明, 在方法定义中要求标明异常信息, 可以提高开发者的生产性, 同时提高代码质量, 但在大型软件中的经验则却指向一个不同的结论 – 生产性降低, 而代码质量改善不大, 或者根本没有改善.

关于这个问题还有其他讨论:

- Java 的受控异常是一个错误 (https://radio-weblogs.com/0122027/stories/2003/04/01/JavasCheckedExceptionsWereAMistake.html) (Rod Waldhoff)
- 受控异常带来的问题 (https://www.artima.com/intv/handcuffs.html) (Anders Hejlsberg) (译注, 他是 Borland Turbo Pascal 和 Delphi 的主要作者, 微软.Net概念的发起人之一, .Net首席架

构师)

从 Java, Swift, 或 Objective-C 中 调用 Kotlin 代码时, 如果你想对函数调用者提示可能发生异常, 可以使用 @Throws 注解. 关于这个注解 在 Java 中如何使用 ("受控异常(Checked Exception)" in "在 Java 中调用 Kotlin 代码") 以及 在 Swift 和 Objective-C 如何使用 ("错误与异常" in "与 Swift/Objective-C 代码交互"), 请阅读这些文档.

Nothing 类型

在 Kotlin 中, throw 是一个表达式, 比如说, 你可以将它用做 Elvis 表达式的一部分:

```
val s = person.name ?: throw IllegalArgumentException("Name
required")
```

throw 表达式的类型是 Nothing. 这个类型没有值, 它被用来标记那些永远无法执行到的代码位置. 在你自己的代码中, 你可以用 Nothing 来标记一个永远不会正常返回的函数:

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

如果你调用这个函数, 编译器就会知道, 执行到这个调用时, 程序就会停止:

```
val s = person.name ?: fail("Name required")
println(s)     // 在这里可以确定地知道 's' 已被正确地初始化
```

在使用类型推断时也可能遇到这个类型. 这个类型的可为 null 的变量, Nothing?, 只有唯一一个可能的值, 就是 null. 如果对一个自动推断类型的值, 使用 null 来初始化, 而且又没有更多的信息可以用来推断出更加具体的类型, 编译器会将类型推断为 Nothing?:

```
val x = null           // 'x' 的类型是 `Nothing?`
val l = listOf(null)   // 'l' 的类型是 `List<Nothing?>
```

与 Java 的互操作性

关于与 Java 的互操作性问题, 请参见 与 Java 的互操作性 (在 Kotlin 中调用 Java 代码) 中关于异常的小节.

包(Package)与导入(Import)

最终更新: 2024/09/10

源代码文件的开始部分可以是包声明:

```
package org.example

fun printMessage() { /*...*/ }
class Message { /*...*/ }

// ...
```

源代码内的所有内容, 比如类, 函数, 全部都包含在所声明的包之内. 因此, 上面的示例代码中, `printMessage()` 函数的完整名称将是 `org.example.printMessage`, `Message` 类的完整名称将是 `org.example.Message`.

如果没有指定包, 那么源代码文件中的内容将属于 *默认* 包, 这个包没有名称.

默认导入

以下各个包会被默认导入到每一个 Kotlin 源代码文件:

- kotlin.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/index.html)
- kotlin.annotation.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/index.html)
- kotlin.collections.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html)
- kotlin.comparisons.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.comparisons/index.html)
- kotlin.io.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/index.html)
- kotlin.ranges.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/index.html)
- kotlin.sequences.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/index.html)

- kotlin.text.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/index.html)

根据编译的目标平台不同, 还会导入以下包:

- JVM 平台:
  - java.lang.*
  - kotlin.jvm.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/index.html)
- JavaScript 平台:
  - kotlin.js.* (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/index.html)

导入(Import)

除默认导入(Import)的内容之外, 各源代码可以包含自己独自的 `import` 指令.

我们可以导入一个单独的名称:

```
import org.example.Message // 导入后 Message 就可以直接访问, 不必指定完整的限定符
```

也可以导入某个范围之内所有可访问的内容, 比如包, 类, 对象, 等等:

```
import org.example.* // 导入后 'org.example' 内的一切都可以访问了
```

如果发生了名称冲突, 你可以使用 `as` 关键字, 给重名实体指定新的名称(新名称仅在当前范围内有效):

```
import org.example.Message // 导入后 Message 可以访问了
import org.test.Message as TestMessage // 可以使用新名称 TestMessage 来访问 'org.test.Message'
```

`import` 关键字不仅可以用来导入类; 还可以用来导入其他声明:

- 顶级(top-level) 函数和属性
- 对象声明 ("对象声明(Object declaration)" in "对象表达式,对象声明,以及同伴对象") 中定义的函数和属性

- 枚举常数 (枚举类)

顶级(top-level) 声明的可见度

如果一个顶级(top-level) 声明被标注为 `private`, 它将成为私有的, 只有在它所属的文件内可以访问 (参见 可见度修饰符 (可见度修饰符)).

最终更新: 2024/09/10

Kotlin 中的类使用 `class` 关键字定义:

```
class Person { /*...*/ }
```

类的定义由以下几部分组成: 类名, 类头部(指定类的类型参数, 主构造器, 以及其他内容), 以及由大括号括起的类主体部分. 类的头部和主体部分都是可选的; 如果类没有主体部分, 那么大括号也可以省略.

```
class Empty
```

构造器

Kotlin 中的类有一个 *主构造器(primary constructor)*, 此外还可以有一个或多个 *次构造器(secondary constructor)*. 主构造器在类头部中声明, 位于类名称以及可选的类型参数之后.

```
class Person constructor(firstName: String) { /*...*/ }
```

如果主构造器没有任何注解(annotation), 也没有任何可见度修饰符, 那么 `constructor` 关键字可以省略:

```
class Person(firstName: String) { /*...*/ }
```

主构造器初始化类的实例, 以及它在类头部中的属性. 类头部不能包含任何可执行的代码. 如果你想要在对象创建时运行某些代码, 可以使用类 body 中的 *初始化代码段(initializer block)*. 初始化代码段使用 `init` 关键字来定义, 之后是大括号. 请将你想要运行的代码放在大括号之内.

在类的实例初始化过程中, 初始化代码段按照它们在类主体中出现的顺序执行, 初始化代码段之间还可以插入属性的初始化代码:

```
//sampleStart
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
```

```
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property:
${name.length}".also(::println)

    init {
        println("Second initializer block that prints
${name.length}")
    }
}
//sampleEnd

fun main() {
    InitOrderDemo("hello")
}
```

主构造器的参数可以在初始化代码段中使用. 也可以在类主体定义的属性初始化代码中使用:

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

Kotlin 有一种简洁语法, 可以通过主构造器来定义属性并初始化属性值:

```
class Person(val firstName: String, val lastName: String, var age:
Int)
```

这种声明还可以包含类属性的默认值:

```
class Person(val firstName: String, val lastName: String, var
isEmployed: Boolean = true)
```

声明类的属性时, 可以使用 尾随逗号(trailing comma) ("尾随逗号(Trailing Comma)" in "编码规约"):

```
class Person(
    val firstName: String,
```

```
    val lastName: String,
    var age: Int, // 尾随逗号(trailing comma)
) { /*...*/ }
```

与通常的属性一样, 主构造器中定义的属性可以是可变的(var), 也可以是只读的(val).

如果构造器有注解, 或者有可见度修饰符, 这时 constructor 关键字是必须的, 注解和修饰符要放在它之前:

```
class Customer public @Inject constructor(name: String) { /*...*/ }
```

详情请参见 可见度修饰符 ("构造器" in "可见度修饰符").

次级构造器(secondary constructor)

类还可以声明 *次级构造器(secondary constructor)*, 使用 constructor 关键字作为前缀:

```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // 将这个 pet 实例添加到它的主人的 pet 列表
    }
}
```

如果类有主构造器, 那么每个次级构造器都必须委托给主构造器, 要么直接委托, 要么通过其他次级构造器间接委托. 委托到同一个类的另一个构造器时, 使用 this 关键字实现:

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

初始化代码段中的代码实际上会成为主构造器的一部分. 在访问次级构造器的第一条语句时, 会执行对主构造器的委托调用, 因此所有初始化代码段中的代码, 以及属性初始化代码, 都会在次级构造器的函数体之前执行.

即使类没有定义主构造器, 也会隐含地委托调用主构造器, 因此初始化代码段仍然会被执行:

```
//sampleStart
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
//sampleEnd

fun main() {
    Constructors(1)
}
```

如果一个非抽象类没有声明任何主构造器和次级构造器, 它将带有一个自动生成的, 无参数的主构造器. 这个构造器的可见度为 public.

如果不希望你的类带有 public 的构造器, 可以声明一个空的构造器, 并明确设置其可见度:

```
class DontCreateMe private constructor() { /*...*/ }
```

> 在 JVM 中, 如果主构造器的所有参数都指定了默认值, 编译器将会产生一个额外的无参数构造器, 这个无参数构造器会使用默认参数值来调用既有的构造器. 有些库(比如 Jackson 或 JPA) 会使用无参数构造器来创建对象实例, 这个特性将使得 Kotlin 比较容易与这种库协同工作.
>
> ```
> class Customer(val customerName: String = "")
> ```

创建类的实例

要创建一个类的实例, 需要调用类的构造器, 调用方式与使用通常的函数一样:

```
val invoice = Invoice()
```

```
val customer = Customer("Joe Smith")
```

> Kotlin 没有 new 关键字.

关于嵌套类, 内部类, 以及匿名内部类的实例创建过程, 请参见嵌套类(Nested Class) (嵌套类与内部类).

类成员

类中可以包含以下内容:

- 构造器和初始化代码块
- 函数 (函数)
- 属性 (属性(Property))
- 嵌套类和内部类 (嵌套类与内部类)
- 对象声明 (对象表达式,对象声明,以及同伴对象)

继承

类可以相互继承, 构成类的继承层级结构. 详情请参加 Kotlin 中的继承 (继承).

抽象类

类本身, 或类中的部分成员, 都可以声明为 abstract 的. 抽象成员在类中不存在具体的实现. 你不必对抽象类或抽象成员标注 open 修饰符.

```
abstract class Polygon {
    abstract fun draw()
}

class Rectangle : Polygon() {
    override fun draw() {
        // 描绘长方形
```

```
    }
}
```

你可以使用抽象成员来覆盖一个非抽象的 `open` 成员:

```
open class Polygon {
    open fun draw() {
        // 某种默认的多边形描绘方法
    }
}

abstract class WildShape : Polygon() {
    // 从 WildShape 继承的类需要实现自己的 draw 方法,
    // 而不是使用 Polygon 中的默认方法
    abstract override fun draw()
}
```

同伴对象(Companion Object)

如果你需要写一个函数, 希望使用者不必通过类的实例来调用它, 但又需要访问类的内部信息(比如, 一个工厂方法), 你可以将这个函数写为这个类之内的一个 对象声明 (对象表达式,对象声明,以及同伴对象) 的成员, 而不是类本身的成员.

具体来说, 如果你在类中声明一个 同伴对象 ("同伴对象(Companion Object)" in "对象表达式,对象声明,以及同伴对象"), 那么只需要使用类名作为限定符就可以访问同伴对象的成员了.

继承

最终更新: 2024/09/10

Kotlin 中所有的类都有一个共同的超类 Any, 如果类声明时没有指定超类, 则默认为 Any:

```
class Example // 隐含地继承自 Any
```

Any 拥有三个函数: equals(), hashCode() 和 toString(). 因此, Kotlin 的所有类都拥有这些函数.

默认情况下, Kotlin 的类是 final 的 - 不能再被继承. 如果要允许一个类被继承, 需要使用 open 关键字标记这个类:

```
open class Base // 这个类现在是 open 的, 可以被继承
```

要明确声明类的超类, 要在类的头部添加一个冒号, 冒号之后指定超类:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

如果子类有主构造器, 那么可以(而且必须)在主构造器中使用主构造器的参数来初始化基类.

如果子类没有主构造器, 那么所有的次级构造器都必须使用 super 关键字来初始化基类, 或者委托到另一个构造器, 由被委托的构造器来初始化基类. 注意, 这种情况下, 不同的次级构造器可以调用基类中不同的构造器:

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx,
attrs)
}
```

方法的覆盖

Kotlin 要求使用明确的修饰符来标识允许被子类覆盖的成员, 也要求使用明确的修饰符来标识对超类成员的覆盖:

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

对于 `Circle.draw()` 必须添加 `override` 修饰符. 如果遗漏了这个修饰符, 编译器将会报告错误. 如果一个函数没有标注 `open` 修饰符, 比如上例中的 `Shape.fill()`, 那么不允许在子类中声明一个同名同参的方法, 无论是否添加 `override` 修饰符, 都不可以. 在一个 final 类(也就是, 没有添加 `open` 修饰符的类)的成员上添加 `open` 修饰符, 不会发生任何效果.

当一个子类成员标记了 `override` 修饰符来覆盖父类成员时, 覆盖后的子类成员本身也将是 open 的, 因此子类成员可以被自己的子类再次覆盖. 如果你希望禁止这种再次覆盖, 可以使用 `final` 关键字:

```
open class Rectangle() : Shape() {
    final override fun draw() { /*...*/ }
}
```

属性的覆盖

属性的覆盖方式与方法覆盖相同; 超类中声明的属性在后代类中再次声明时, 必须使用 `override` 关键字来标记, 而且覆盖后的属性数据类型必须与超类中的属性数据类型兼容. 超类中声明的属性, 在后代类中可以使用带初始化器的属性来覆盖, 也可以使用带 `get` 方法的属性来覆盖:

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
    override val vertexCount = 4
}
```

你也可以使用一个 `var` 属性覆盖一个 `val` 属性, 但不可以反过来使用一个 `val` 属性覆盖一个 `var` 属性. 允许这种覆盖的原因是, `val` 属性本质上只是定义了一个 `get` 方法, 使用 `var` 属性来覆盖它, 只是

向后代类中添加了一个 `set` 方法.

注意, 你可以在主构造器的属性声明中使用 `override` 关键字:

```
interface Shape {
    val vertexCount: Int
}

class Rectangle(override val vertexCount: Int = 4) : Shape // 长方形
总是拥有 4 个顶点

class Polygon : Shape {
    override var vertexCount: Int = 0  // 多边形的顶点数目不定, 可以变更
为任何数字
}
```

子类的初始化顺序

子类新实例构造的过程中, 首先完成的第一步是要初始化基类 (顺序上仅次于计算传递给基类构造器的参数值), 因此要在子类的初始化逻辑之前执行.

```
//sampleStart
open class Base(val name: String) {

    init { println("Initializing a base class") }

    open val size: Int =
        name.length.also { println("Initializing size in the base
class: $it") }
}

class Derived(
    name: String,
    val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also {
println("Argument for the base class: $it") }) {

    init { println("Initializing a derived class") }

```

```
    override val size: Int =
        (super.size + lastName.length).also { println("Initializing
size in the derived class: $it") }
}
//sampleEnd

fun main() {
    println("Constructing the derived class(\"hello\", \"world\")")
    Derived("hello", "world")
}
```

也就是说, 基类构造器执行时, 在子类中定义或覆盖的属性还没有被初始化. 如果在基类初始化逻辑中使用到这些属性 (无论是直接使用, 还是通过另一个被覆盖的 `open` 成员间接使用), 可能会导致不正确的行为, 甚至导致运行时错误. 因此, 设计基类时, 在构造器, 属性初始化器, 以及 `init` 代码段中, 你应该避免使用 `open` 成员.

调用超类中的实现

后代类中的代码, 可以使用 `super` 关键字来调用超类中的函数和属性访问器的实现:

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

在内部类(inner class)的代码中, 可以使用 `super` 关键字加上外部类名称限定符: `super@Outer` 来访问外部类(outer class)的超类:

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
```

```
    val borderColor: String get() = "black"
}

//sampleStart
class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // 调用 Rectangle 的 draw()
函数实现
            fill()
            println("Drawn a filled rectangle with color
${super@FilledRectangle.borderColor}") // 使用 Rectangle 的
borderColor 属性的 get() 函数
        }
    }
}
//sampleEnd

fun main() {
    val fr = FilledRectangle()
        fr.draw()
}
```

覆盖的规则

在 Kotlin 中, 类继承中的方法实现问题, 遵守以下规则: 如果一个类从它的直接超类中继承了同一个成员的多个实现, 那么这个子类必须覆盖这个成员, 并提供一个自己的实现(可以使用继承得到的多个实现中的某一个). 为了表示使用的方法是从哪个超类继承得到的, 可以使用 `super` 关键字, 将超类名称放在尖括号类, 比如, `super<Base>`:

```
open class Rectangle {
    open fun draw() { /* ... */ }
```

```
}

interface Polygon {
    fun draw() { /* ... */ } // 接口的成员默认是 'open' 的
}

class Square() : Rectangle(), Polygon {
    // 编译器要求 draw() 方法必须覆盖:
    override fun draw() {
        super<Rectangle>.draw() // 调用 Rectangle.draw()
        super<Polygon>.draw() // 调用 Polygon.draw()
    }
}
```

同时继承 `Rectangle` 和 `Polygon` 是合法的, 但他们都实现了函数 `draw()` 的继承就发生了问题, 因此你需要在 `Square` 类中覆盖函数 `draw()`, 并提供单独的实现, 这样才能消除歧义.

属性(Property)

最终更新: 2024/09/10

声明属性

Kotlin 类的属性可以使用 `var` 关键字声明为可变(mutable)属性, 也可以使用 `val` 关键字声明为只读属性.

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

使用属性时, 只需要简单地通过属性名来参照它:

```
fun copyAddress(address: Address): Address {
    val result = Address() // Kotlin 中没有 'new' 关键字
    result.name = address.name // 将会调用属性的访问器方法
    result.street = address.street
    // ...
    return result
}
```

取值方法(Getter)与设值方法(Setter)

声明属性的完整语法是:

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

其中的初始化器(initializer), 取值方法(getter), 以及设值方法(setter)都是可选的. 如果属性类型可以通过初始化器自动推断得到, 或者可以通过取值方法的返回值类型推断得到, 则属性类型的声明也

可以省略, 示例如下:

```
var initialized = 1 // 属性类型为 Int, 使用默认的取值方法和设值方法
// var allByDefault: Int? // 错误: 需要明确指定初始化器, 此处会隐含地使用默认的取值方法和设值方法
```

只读属性声明的完整语法与可变属性有两点不同: 由 `val` 开头, 而不是 `var`, 并且不允许指定设值方法:

```
val simple: Int? // 属性类型为 Int, 使用默认的取值方法, 属性值必须在构造器中初始化
val inferredType = 1 // 属性类型为 Int, 使用默认的取值方法
```

你可以为属性定义自定义的访问方法. 如果定义一个自定义取值方法(Getter), 那么每次读取属性值时都会调用这个方法 (因此你可以用这种方式实现一个计算得到的属性). 下面是一个自定义取值方法的示例:

```
//sampleStart
class Rectangle(val width: Int, val height: Int) {
    val area: Int // 属性类型是可选的, 因为可以从取值方法的返回类型推断得到
        get() = this.width * this.height
}
//sampleEnd
fun main() {
    val rectangle = Rectangle(3, 4)
    println("Width=${rectangle.width}, height=${rectangle.height}, area=${rectangle.area}")
}
```

如果能够从取值方法(Getter)推断得到属性类型, 那么可以省略:

```
val area get() = this.width * this.height
```

如果你定义一个自定义设值方法(Setter), 那么每次向属性赋值时都会调用这个方法, 属性初始化时除外. 自定义设值方法的示例如下:

```
var stringRepresentation: String
    get() = this.toString()
```

```
    set(value) {
        setDataFromString(value) // 解析字符串内容, 并将解析得到的值赋给对应的其他属性
    }
```

Kotlin 的编程惯例是, 设值方法的参数名称为 `value`, 但如果你喜欢, 也可以选择使用不同的名称.

如果你需要对属性访问方法添加注解, 或者需要改变其可见度, 但又不想修改它的默认实现, 你可以定义这个方法, 但不定义它的实现体:

```
var setterVisibility: String = "abc"
    private set // 设值方法的可见度为 private, 并使用默认实现

var setterWithAnnotation: Any? = null
    @Inject set // 对设值方法添加 Inject 注解
```

属性的后端域变量(Backing Field)

在 Kotlin 中, 只有需要将域变量(field)作为属性的一部分, 在内存中保存属性值的时候, 才会使用域变量(field). 域变量不能直接声明. 但是, 如果属性需要一个后端域变量(Backing Field), Kotlin 会自动提供. 在属性的取值方法或设值方法中, 使用 `field` 标识符可以引用这个后端域变量:

```
var counter = 0 // 这里的初始化代码直接赋值给后端域变量
    set(value) {
        if (value >= 0)
            field = value
            // counter = value // 此处会发生栈溢出错误: 使用属性名称
'counter' 会导致设值方法(setter)无限递归调用
    }
```

`field` 标识符只允许在属性的访问器函数内使用.

如果属性 get/set 方法中的任何一个使用了默认实现, 或者在 get/set 方法的自定义实现中通过 `field` 标识符访问属性, 那么编译器就会为属性自动生成后端域变量.

比如, 下面的示例代码中不会存在后端域变量:

```
val isEmpty: Boolean
    get() = this.size == 0
```

后端属性(Backing Property)

如果你希望实现的功能无法通过这种 *隐含的后端域变量* 方案来解决, 你可以使用 *后端属性(backing property)* 作为替代方案:

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // 类型参数可以自动推断得到, 不必指定
        }
        return _table ?: throw AssertionError("Set to null by 
another thread")
    }
```

> ⚠ 对于 JVM 平台: 如果私有属性的取值方法与设值方法都使用默认实现, 那么对这个属性的访问将被编译器优化, 变为直接读写后端域变量, 以避免不必要的函数调用造成性能损失.

编译期常数值

如果只读属性的值在编译期间就能确定, 请使用 `const` 修饰符, 将它标记为 *编译期常数值(compile time constant)*. 这类属性必须满足以下所有条件:

- 必须是顶级属性(Top-level Property), 或者是一个 `object` 声明 ("对象声明(Object declaration)" in "对象表达式,对象声明,以及同伴对象") 的成员, 或者是一个 _ 同伴对象(Companion object) ("同伴对象(Companion Object)" in "对象表达式,对象声明,以及同伴对象") 的成员.
- 值必须初始化为 `String` 类型, 或基本类型(primitive type)
- 不存在自定义的取值方法

编译器会对常数的使用进行内联(inline), 将对常数的引用替换为常数的实际值. 但是, 常数对应的域变量不会被删除, 因此可以通过使用 反射 (反射) 与它进行交互.

这类属性也可以用在注解内:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is 
deprecated"
```

```
@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

延迟初始化的(Late-Initialized)属性和变量

通常, 如果属性声明为非 null 数据类型, 那么属性值必须在构造器内初始化. 但是, 这种限制很多时候会带来一些不便. 比如, 属性值可以通过依赖注入来进行初始化, 或者在单元测试代码的 setup 方法中初始化. 这种情况下, 你就无法在构造器中为属性编写一段非 null 值的初始化代码, 但你仍然希望在类内参照这个属性时能够避免 null 值检查.

要解决这个问题, 你可以为属性添加一个 `lateinit` 修饰符:

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // 直接访问属性
    }
}
```

这个修饰符可以用于类主体部分之内声明的 `var` 属性, (不是主构造器中声明的属性, 而且属性没有自定义的取值方法和设值方法). 也可以用于顶级(top-level)属性和局部变量. 属性或变量的类型必须是非 null 的, 而且不能是基本类型.

在一个 `lateinit` 属性被初始化之前访问它, 会抛出一个特别的异常, 这个异常将会指明被访问的属性, 以及它没有被初始化这一错误.

检查 lateinit var 是否已完成初始化

为了检查一个 `lateinit var` 是否已经初始化完成, 可以对 属性的引用 ("属性引用(Property Reference)" in "反射") 调用 `.isInitialized`:

```
if (foo::bar.isInitialized) {
    println(foo.bar)
}
```

这种检查只能用于当前代码可以访问到的属性, 因此属性必须定义在当前代码的同一个类中, 或当前代码的外部类中, 或者是同一个源代码文件中的顶级属性.

属性的覆盖

参见 属性的覆盖 ("属性的覆盖" in "继承")

委托属性(Delegated Property)

最常见的属性只是简单地读取(也有可能会写入)一个后端域变量. 但是, 通过使用自定义的取值方法和设值方法也可以访问属性, 因此可以实现属性的任意复杂的行为. 在第一种极简单的情况与第二种极复杂的情况之间, 还存在一些常见的属性工作模式. 比如: 属性值的延迟加载, 通过指定的键值(key)从 map 中读取数据, 访问数据库, 属性被访问时通知监听器.

这些常见行为可以使用*委托属性(delegated property)* (委托属性), 以库的形式实现.

接口(Interface)

最终更新: 2024/09/10

Kotlin 中的接口可以包含抽象方法的声明, 也可以包含方法的实现. 接口与抽象类的区别在于, 接口不能存储状态数据. 接口可以有属性, 但这些属性必须是抽象的, 或者必须提供访问器的自定义实现.

接口使用 `interface` 关键字来定义:

```
interface MyInterface {
    fun bar()
    fun foo() {
      // 方法体是可选的
    }
}
```

实现接口

类或者对象可以实现一个或多个接口

```
class Child : MyInterface {
    override fun bar() {
        // 方法体
    }
}
```

接口中的属性

你可以在接口中定义属性. 接口中声明的属性要么是抽象的, 要么提供访问器的自定义实现. 接口中声明的属性不能拥有后端域变量(backing field), 因此, 在接口中定义的属性访问器也不能访问属性的后端域变量:

```
interface MyInterface {
    val prop: Int // 抽象属性

    val propertyWithImplementation: String
        get() = "foo"
```

```
    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

接口的继承

接口也可以继承其他接口, 因此它可以对父接口中的成员提供实现, 同时又声明新的函数和属性. 很自然的, 类在实现这样的接口时, 只需要实现缺少的函数和属性:

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // 不需要实现 'name' 属性
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

解决覆盖冲突(overriding conflict)

如果你为一个类指定了多个超类, 可能会导致对同一个方法继承得到了多个实现:

```
interface A {
    fun foo() { print("A") }
```

```
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

接口 *A* 和 *B* 都定义了函数 *foo()* 和 *bar()*. 它们也都实现了 *foo()*, 但只有 *B* 实现了 *bar()* (在 *A* 中 *bar()* 没有标记为 abstract, 因为在接口中, 如果没有定义函数体, 则函数默认为 abstract). 现在, 如果你从 *A* 派生一个实体类 *C*, 那么必须覆盖函数 *bar()*, 并提供一个实现.

然而, 如果你从 *A* 和 *B* 派生出 *D*, 对于从多个接口中继承得到的所有方法我们都需要实现, 并且指明 *D* 具体应该如何实现各个方法. 对于只继承得到了单个实现的方法(如上例中的 *bar()* 方法), 以及继承得到了多个实现的方法(如上例中的 *foo()* 方法), 都存在这个限制.

函数式 (SAM) 接口

最终更新: 2024/09/10

只有一个抽象方法的接口称为 *函数式接口 (Functional Interface)*, 或者叫做 *单抽象方法(SAM, Single Abstract Method) 接口*. 函数式接口可以拥有多个非抽象的成员, 但只能拥有一个抽象成员.

在 Kotlin 中声明函数式接口时, 请使用 `fun` 修饰符.

```
fun interface KRunnable {
    fun invoke()
}
```

SAM 转换功能

对于函数式接口, 可以通过 SAM 转换功能, 使用 Lambda 表达式 ("Lambda 表达式与匿名函数 (Anonymous Function)" in "高阶函数与 Lambda 表达式"), 让你的代码更加简洁易读.

你可以使用 Lambda 表达式, 而不必手动的创建一个类, 实现函数式接口. 只要 Lambda 表达式的签名与接口的唯一方法的签名相匹配, Kotlin 可以通过 SAM 转换功能, 将任意的 Lambda 表达式转换为一段代码, 创建一个实现接口的类的实例.

比如, 对于下面的 Kotlin 函数式接口:

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}
```

如果不使用 SAM 转换功能, 那么就需要编写这样的代码:

```
// 创建类的实例
val isEven = object : IntPredicate {
    override fun accept(i: Int): Boolean {
        return i % 2 == 0
    }
}
```

使用 Kotlin 的 SAM 转换功能, 就可以编写下面的代码, 效果相同:

```
// 使用 Lambda 表达式创建实例
val isEven = IntPredicate { it % 2 == 0 }
```

这样, 就通过更加简短的 Lambda 表达式代替了所有其他不必要的代码.

```
fun interface IntPredicate {
   fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
   println("Is 7 even? - ${isEven.accept(7)}")
}
```

也可以使用 对 Java 接口的 SAM 转换功能 ("SAM 转换" in "在 Kotlin 中调用 Java 代码").

从带构造器函数的接口迁移到函数式接口

从 1.6.20 开始, Kotlin 支持对函数式接口构造器的 可调用的引用 ("可调用的引用" in "反射"), 因此增加了一种源代码兼容的方式, 可以从带构造器函数的接口迁移到函数式接口. 我们来看看以下代码:

```
interface Printer {
    fun print()
}

fun Printer(block: () -> Unit): Printer = object : Printer {
override fun print() = block() }
```

由于可以使用对函数式接口构造器的可调用的引用, 这段代码可以替换为函数式接口声明:

```
fun interface Printer {
    fun print()
}
```

它的构造器会隐含的创建, 使用 `::Printer` 函数引用的任何代码都可以正确编译. 比如:

```
documentsStorage.addPrinter(::Printer)
```

如果要保留二进制兼容性, 可以对过去的函数 `Printer` 标记 `@Deprecated` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-deprecated/) 注解, 注解参数是 `DeprecationLevel.HIDDEN`:

```
@Deprecated(message = "Your message about the deprecation", level =
DeprecationLevel.HIDDEN)
fun Printer(...) {...}
```

函数式接口 与 类型别名(Type Alias)

你也可以对函数类型使用 类型别名(Type Alias) (类型别名), 简单的重写上面的代码:

```
typealias IntPredicate = (i: Int) -> Boolean

val isEven: IntPredicate = { it % 2 == 0 }

fun main() {
    println("Is 7 even? - ${isEven(7)}")
}
```

但是, 函数式接口 与 类型别名(Type Alias) (类型别名) 服务于不同的目的. 类型别名只是对已有的类型提供一个新的名称 – 它不会创建新的类型, 而函数式接口会. 对某个特定的函数式接口, 你可以提供扩展, 但对通常的函数或函数的类型别名则不可以.

类型别名只能拥有一个成员, 而函数式接口可以拥有多个非抽象的成员和一个抽象成员. 函数式接口也可以实现或继承其他接口.

函数式接口比类型别名更加灵活, 也提供了更多功能, 但语法上以及在运行时刻都存在更多代价, 因为需要转换为特定的接口. 当在你的代码中需要选择使用哪一种时, 应该考虑你的需求:

- 如果你的 API 需要接受一个函数 (任意的函数), 带有某些特定参数和返回类型 – 可以使用简单的函数类型, 或者为这个函数类型定义一个类型别名, 使得它的名称更简短.
- 如果你的 API 需要接受比函数更加复杂的实体 – 比如, 它带有比较重要的规约 和/或 操作, 无法表达为函数类型的签名 – 那么需要为它定义一个单独的函数式接口.

可见度修饰符

最终更新: 2024/09/10

类, 对象, 接口, 构造器, 函数, 属性, 以及属性的设值方法, 都可以使用 *可见度修饰符*. 属性的取值方法永远与属性本身的可见度一致, 因此不需要控制其可见度.

Kotlin 中存在 4 种可见度修饰符: `private`, `protected`, `internal`, 以及 `public`. 默认的可见度为 `public`.

本节中, 我们将介绍这些可见度标识符如何应用于不同范围的不同类型.

包

函数, 属性, 类, 对象, 接口, 都可以直接在包之内声明为"顶级的(top-level)":

```
// file name: example.kt
package foo

fun baz() { ... }
class Bar { ... }
```

- 如果你不使用可见度修饰符, 默认会使用 `public`, 其含义是, 你声明的东西在任何位置都可以访问.
- 如果你将声明的东西标记为 `private`, 那么它将只在同一个源代码文件内可以访问.
- 如果标记为 `internal`, 那么它将在同一个模块(module)内的任何位置都可以访问.
- 对于顶级(top-level)声明, `protected` 修饰符是无效的.

> 如果一个包中的顶级声明是可见的, 在另一个包中使用它时需要 导入(import) ("导入(Import)" in "包(Package)与导入(Import)").

示例:

```
// 文件名: example.kt
package foo
```

```
private fun foo() { ... } // 只在 example.kt 文件内可访问

public var bar: Int = 5 // 这个属性在任何地方都可以访问
    private set         // 但它的设值方法只在 example.kt 文件内可以访问

internal val baz = 6    // 在同一个模块(module)内可以访问
```

类成员

对于类内部声明的成员:

- `private` 表示这个成员只在这个类(以及它的所有成员)之内可以访问.
- `protected` 表示这个成员的可见度与 `private` 一样, 但它在子类中也可以访问.
- `internal` 表示在 *本模块之内*, 凡是能够访问到这个类的地方, 同时也能访问到这个类的 `internal` 成员.
- `public` 表示凡是能够访问到这个类的地方, 同时也能访问这个类的 `public` 成员.

> 在 Kotlin 中, 外部类(outer class)不能访问其内部类(inner class)的 `private` 成员.

如果你覆盖一个 `protected` 或 `internal` 成员, 并且没有明确指定可见度, 那么覆盖后成员的可见度将与覆盖前的成员一样.

示例:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal open val c = 3
    val d = 4  // 默认为 public

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
```

```
    // a 不可访问
    // b, c 和 d 可以访问
    // Nested 和 e 可以访问

    override val b = 5   // 'b' 可见度为 protected
    override val c = 7   // 'c' 可见度为 internal
}

class Unrelated(o: Outer) {
    // o.a, o.b 不可访问
    // o.c 和 o.d 可以访问(属于同一模块)
    // Outer.Nested 不可访问, Nested::e 也不可访问
}
```

构造器

要指定类的主构造器的可见度, 请使用以下语法:

> 你需要明确添加一个 constructor 关键字:

```
class C private constructor(a: Int) { ... }
```

这里构造器是 private 的. 所有构造器默认都是 public 的, 因此使得凡是可以访问到类的地方都可以访问到类的构造器 (因此 一个 internal 类的构造器只能在同一个模块内访问).

对于封闭类(Sealed Class), 构造器默认为 protected 的. 更多详情请参见 封闭类(Sealed Class) ("构造器" in "封闭类(Sealed Class)与封闭接口(Sealed Interface)").

局部声明

局部变量, 局部函数, 以及局部类, 都不能指定可见度修饰符.

模块(Module)

internal 修饰符表示这个成员只能在同一个模块内访问. 更确切地说, 一个模块(module)是指一起编译的一组 Kotlin 源代码文件, 例如:

- 一个 IntelliJ IDEA 模块.

- 一个 Maven 工程.
- 一个 Gradle 源代码集(source set) (`test` 源代码集例外, 它可以访问 `main` 中的 `internal` 声明).
- 通过 `<kotlinc>` Ant 任务的一次调用编译的一组文件.

扩展

最终更新: 2024/09/10

Kotlin 提供了向一个类或一个接口扩展新功能的能力, 而且不必从这个类继承, 也不必使用 *装饰器 (Decorator)* 之类的设计模式. 这种功能是通过一种特殊的声明来实现的, Kotlin 中称为 *扩展 (extension)*.

比如, 你无法修改第三方库中的一个类或一个接口, 但你可以为它编写新的函数. 这些函数可以象原来的类的方法那样调用. 这种机制称为 *扩展函数(extension function)*. 此外还有 *扩展属性 (extension property)*, 你可以用来向已有的类添加新的属性.

扩展函数(Extension Function)

要声明一个扩展函数, 需要在函数名之前添加前缀, 表示这个函数的 *接收者类型(receiver type)*, 也就是我们希望扩展的对象类型. 以下示例将为 `MutableList<Int>` 类型添加一个 `swap` 函数:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' 指代 list 实例
    this[index1] = this[index2]
    this[index2] = tmp
}
```

在扩展函数内, `this` 关键字指代接收者对象(receiver object)(也就是调用扩展函数时, 在点号之前指定的对象实例). 现在, 你可以对任意一个 `MutableList<Int>` 对象调用这个扩展函数:

```
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // 'swap()' 函数内的 'this' 将指向 'list' 的值
```

这个函数可以适用与任意元素类型的 `MutableList<T>`, 因此你可以使用泛型, 将它的元素类型泛化:

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' 指代 list 实例
    this[index1] = this[index2]
    this[index2] = tmp
}
```

你需要在函数名之前声明泛型的类型参数, 然后在接收者类型表达式中就可以使用泛型了. 关于泛型的更多详情, 请参见 泛型函数 (泛型(Generic): in, out, where).

扩展函数的解析是 静态 的

扩展函数并不会真正修改它所扩展的类. 定义扩展函数时, 并没有向类中插入新的成员方法, 而只是创建了一个新的函数, 并且可以通过点号标记法的形式, 对这个数据类型的变量调用这个新函数.

扩展函数的调用派发过程是 *静态* 的. 因此, 被调用的是哪个扩展函数, 是在编译期间通过接受者的类型决定的. 比如:

```
fun main() {
//sampleStart
    open class Shape
    class Rectangle: Shape()

    fun Shape.getName() = "Shape"
    fun Rectangle.getName() = "Rectangle"

    fun printClassName(s: Shape) {
        println(s.getName())
    }

    printClassName(Rectangle())
//sampleEnd
}
```

这段示例程序的打印结果将是 *Shape*, 因为调用哪个函数, 仅仅是由参数 `s` 声明的类型决定, 这里参数 `s` 的类型为 `Shape` 类.

如果类中存在成员函数, 同时又在同一个类上定义了同名的扩展函数, 并且与调用时指定的参数匹配, 这种情况下 *成员函数总是会优先使用*. 比如:

```
fun main() {
//sampleStart
    class Example {
        fun printFunctionType() { println("Class method") }
    }

    fun Example.printFunctionType(i: Int) { println("Extension
```

```
function #$i") }

    Example().printFunctionType(1)
//sampleEnd
}
```

这段代码的输出将是 *Class method*.

但是, 我们完全可以使用同名称但不同参数的扩展函数, 来重载(overload)成员函数:

```
fun main() {
//sampleStart
    class Example {
        fun printFunctionType() { println("Class method") }
    }

    fun Example.printFunctionType(i: Int) { println("Extension
function") }

    Example().printFunctionType(1)
//sampleEnd
}
```

可为空的接收者(Nullable Receiver)

注意, 对可以为空的接收者类型也可以定义扩展. 这样的扩展函数, 即使在对象变量值为 `null` 时也可以调用. 如果接受者为 `null`, 那么 `this` 也是 `null`. 因此, 在对可为 `null` 的接收者类型定义扩展时, 我们建议在函数体之内进行 `this == null` 检查, 以避免编译错误.

你可以在 Kotlin 中调用 `toString()` 函数, 而不必检查对象是否为 `null`, 因为在扩展函数内部已经进行了对象是否为 `null` 的检查.

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // 进行过 null 检查后, 'this' 会被自动转换为非 null 类型, 因此下面的
toString() 方法
    // 会被解析为 Any 类的成员函数
    return toString()
}
```

扩展属性(Extension Property)

Kotlin 也支持扩展属性, 与扩展函数类似:

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

> 由于扩展属性实际上不会向类添加新的成员, 因此无法让一个扩展属性拥有一个 后端域变量 ("属性的后端域变量(Backing Field)" in "属性(Property)"). 所以, *对于扩展属性不允许存在初始化器*. 扩展属性的行为只能通过明确给定的取值方法与设值方法来定义.

示例:

```
val House.number = 1 // 错误: 扩展属性不允许存在初始化器
```

对同伴对象(Companion Object)的扩展

如果一个类定义了同伴对象 ("同伴对象(Companion Object)" in "对象表达式,对象声明,以及同伴对象"), 你可以对这个同伴对象定义扩展函数和扩展属性. 与同伴对象的常规成员一样, 可以只使用类名限定符来调用这些扩展函数和扩展属性:

```
class MyClass {
    companion object { }  // 通过 "Companion" 来引用这个同伴对象
}

fun MyClass.Companion.printCompanion() { println("companion") }

fun main() {
    MyClass.printCompanion()
}
```

扩展的范围

大多数情况下, 你可以直接在包之下的顶级位置定义扩展:

```
package org.example.declarations

fun List<String>.getLongestString() { /*...*/}
```

要在这个包之外使用扩展, 需要在调用处 import 这个扩展:

```
package org.example.usage

import org.example.declarations.getLongestString

fun main() {
    val list = listOf("red", "green", "blue")
    list.getLongestString()
}
```

详情请参见 导入 ("导入(Import)" in "包(Package)与导入(Import)").

将扩展定义为成员

你可以在一个类的内部为另一个类定义扩展. 在这类扩展中, 存在多个 *隐含接受者(implicit receiver)* - 这些隐含接收者的成员可以不使用限定符直接访问. 扩展方法的定义所在的类的实例, 称为 *派发接受者(dispatch receiver)*, 扩展方法的目标类型的实例, 称为 *扩展接受者(extension receiver)*.

```
class Host(val hostname: String) {
    fun printHostname() { print(hostname) }
}

class Connection(val host: Host, val port: Int) {
    fun printPort() { print(port) }

    fun Host.printConnectionString() {
        printHostname()   // 这里会调用 Host.printHostname()
        print(":")
        printPort()   // 这里会调用 Connection.printPort()
    }

    fun connect() {
```

```
        /*...*/
        host.printConnectionString()   // 这里会调用扩展函数
    }
}

fun main() {
    Connection(Host("kotl.in"), 443).connect()
    //Host("kotl.in").printConnectionString()  // 错误, 在 Connection
之外无法访问扩展函数
}
```

当派发接受者与扩展接受者的成员名称发生冲突时, 扩展接受者的成员将会被优先使用. 如果想要使用派发接受者的成员, 请参见 带限定符的 `this` 语法 ("带限定符的 this" in "this 表达式").

```
class Connection {
    fun Host.getConnectionString() {
        toString()         // 这里会调用 Host.toString()
        this@Connection.toString()  // 这里会调用
Connection.toString()
    }
}
```

以成员的形式定义的扩展函数, 可以声明为 `open`, 而且可以在子类中覆盖. 也就是说, 在这类扩展函数的派发过程中, 针对派发接受者是虚拟的(virtual), 但针对扩展接受者仍然是静态的(static).

```
open class Base { }

class Derived : Base() { }

open class BaseCaller {
    open fun Base.printFunctionInfo() {
        println("Base extension function in BaseCaller")
    }

    open fun Derived.printFunctionInfo() {
        println("Derived extension function in BaseCaller")
    }
```

```
    fun call(b: Base) {
        b.printFunctionInfo()   // 这里会调用扩展函数
    }
}

class DerivedCaller: BaseCaller() {
    override fun Base.printFunctionInfo() {
        println("Base extension function in DerivedCaller")
    }

    override fun Derived.printFunctionInfo() {
        println("Derived extension function in DerivedCaller")
    }
}

fun main() {
    BaseCaller().call(Base())   // 输出结果为 "Base extension function
in BaseCaller"
    DerivedCaller().call(Base())  // 输出结果为 "Base extension
function in DerivedCaller" - 派发接受者的解析过程是虚拟的
    DerivedCaller().call(Derived())  // 输出结果为 "Base extension
function in DerivedCaller" - 扩展接受者的解析过程是静态的
}
```

关于可见度的注意事项

扩展函数或扩展属性 对其他元素的可见度 (可见度修饰符) 规则, 与定义在同一范围内的普通函数相同. 比如:

- 定义在源代码文件顶级(top-level)范围内的扩展, 可以访问同一源代码文件内的其他顶级 `private` 元素.
- 如果扩展定义在它的接受者类型的外部, 那么它不能访问接受者的 `private` 或 `protected` 成员.

数据类(Data Class)

最终更新: 2024/09/10

Kotlin 中数据类(Data Class)的主要用来保存数据. 对每个数据类, 编译器会自动生成一些额外的成员函数, 可以用来将对象输出为可读的格式, 比较对象实例, 复制对象实例, 等等. 数据类通过 data关键字标记:

```
data class User(val name: String, val age: Int)
```

编译器会根据主构造器中声明的全部属性, 自动推断产生以下成员函数:

- .equals()/.hashCode() 函数对.
- .toString() 函数, 输出格式为 "User(name=John, age=42)".
- .componentN() 函数群 (解构声明), 这些函数与类的属性对应, 函数名中的数字 1 到 N, 与属性的声明顺序一致.
- .copy() 函数 (详情见下文).

为了保证自动生成的代码的行为一致, 并且有意义, 数据类必须满足以下所有要求:

- 主构造器必须有一个以上参数.
- 主构造器的所有参数必须标记为 val 或 var.
- 数据类不能是抽象类, open 类, 封闭(sealed)类, 或内部(inner)类.

此外, 考虑到成员函数继承的问题, 成员函数的生成遵循以下规则:

- 对于 .equals(), .hashCode() 或 .toString() 函数, 如果在数据类的定义体中存在明确的实现, 或在超类中存在 final 的实现, 那么这些成员函数不会自动生成, 而会使用已存在的实现.
- 如果超类存在 open 的 .componentN() 函数, 并且返回一个兼容的数据类型, 那么子类中对应的函数会自动生成, 并覆盖超类中的函数. 如果超类中的函数签名不一致, 或者是 final 的, 导致子类无法覆盖, 则会报告编译错误.
- 不允许对 .componentN() 和 .copy() 函数提供明确的实现(译注, 这些函数必须由编译器自动生成).

数据类可以继承其他类 (示例请参见 封闭类(Sealed class) (封闭类(Sealed Class)与封闭接口 (Sealed Interface))).

> 在 JVM 平台, 如果自动生成的类需要拥有一个无参数的构造器, 那么需要为属性指定默认值 (参见 构造器 ("构造器" in "类")):
>
> ```
> data class User(val name: String = "", val age: Int = 0)
> ```

在类主体部声明的属性

编译器对自动生成的函数, 只使用主构造器中定义的属性. 如果想要在自动生成的函数实现中排除某个属性, 你可以将它声明在类的主体部:

```
data class Person(val name: String) {
    var age: Int = 0
}
```

在下面的示例中, 在 `.toString()`, `.equals()`, `.hashCode()`, 和 `.copy()` 函数的实现中, 默认只使用了 `name` 属性, 而且只存在 1 个组件函数 `.component1()`. `age` 属性定义在类的 body 部, 因此被排除了. 所以, 两个 `Person` 对象拥有相同的 `name`, 不同的 `age` 值, 它们会被认为值相等. 因为 `.equals()` 只计算主构造器中的属性:

```
data class Person(val name: String) {
    var age: Int = 0
}
fun main() {
//sampleStart
  val person1 = Person("John")
  val person2 = Person("John")
  person1.age = 10
  person2.age = 20

  println("person1 == person2: ${person1 == person2}")
  // 输出结果为 person1 == person2: true

  println("person1 with age ${person1.age}: ${person1}")
  // 输出结果为 person1 with age 10: Person(name=John)
```

```
    println("person2 with age ${person2.age}: ${person2}")
    // 输出结果为 person2 with age 20: Person(name=John)
//sampleEnd
}
```

对象复制

使用 `.copy()` 函数来复制对象, 可以修改 *一部分* 属性值, 但保持其他属性不变. 对于前面示例中的 `User` 类, 函数的实现将会是下面这样:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name,
age)
```

因此你可以编写下面这样的代码:

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

数据类中成员数据的解构

编译器会为数据类生成 *组件函数(Component function)*, 有了这些组件函数, 就可以在 解构声明 (destructuring declaration) (解构声明) 中使用数据类:

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age")
// 输出结果为 Jane, 35 years of age
```

标准库中的数据类

Kotlin 的标准库提供了 `Pair` 和 `Triple` 类可供使用. 但大多数情况下, 使用有具体名称的数据类是一种更好的设计方式, 因为, 数据类可以为属性指定有含义的名称, 因此可以让代码更加易读.

封闭类(Sealed Class)与封闭接口(Sealed Interface)

最终更新: 2024/09/10

封闭 类和接口提供了对类的继承关系进行控制的方式. 一个封闭类的所有的直接子类(Direct Subclass)在编译时刻就能够确定. 在定义封闭类的模块和包之外, 不可能再出现其他子类. 对于封闭接口和它们的实现类也是如此: 在含有封闭接口的模块编译完成之后, 就不可能再创建新的实现类.

> 直接子类(Direct Subclass) 是指直接从父类继承的那些类.
>
> 间接子类(Indirect Subclass) 是指从父类继承, 但继承关系超过一层的那些类.

当你将封闭类和接口与 `when` 表达式一起使用时, 你可以覆盖所有可能的子类的行为, 并保证不会有新的子类创建出来, 对你的代码产生不利的影响.

封闭类最适合使用的场景是:

- **期望限制类的继承:** 你有一组预定义的, 有限的子类, 来扩展一个类, 所有这些子类在编译期都是已知的.
- **需要实现类型安全的设计:** 安全性和模式匹配在你的项目中至关重要. 特别是对于状态管理, 或处理复杂的条件逻辑. 例如, 请参见 和 when 表达式一起使用封闭类.
- **使用封闭的 API:** 你希望为库提供健壮而且可维护的公开 API, 确保第三方客户端按预期的方式使用 API.

更详细的实际应用, 请参见 使用场景.

> Java 15 引入了 一个类似的概念 (https://docs.oracle.com/en/java/javase/15/language/sealed-classes-and-interfaces.html#GUID-0C709461-CC33-419A-82BF-61461336E65F), 它的封闭类使用 `sealed` 关键字和 `permits` 子句, 来定义受限制的层级结构.

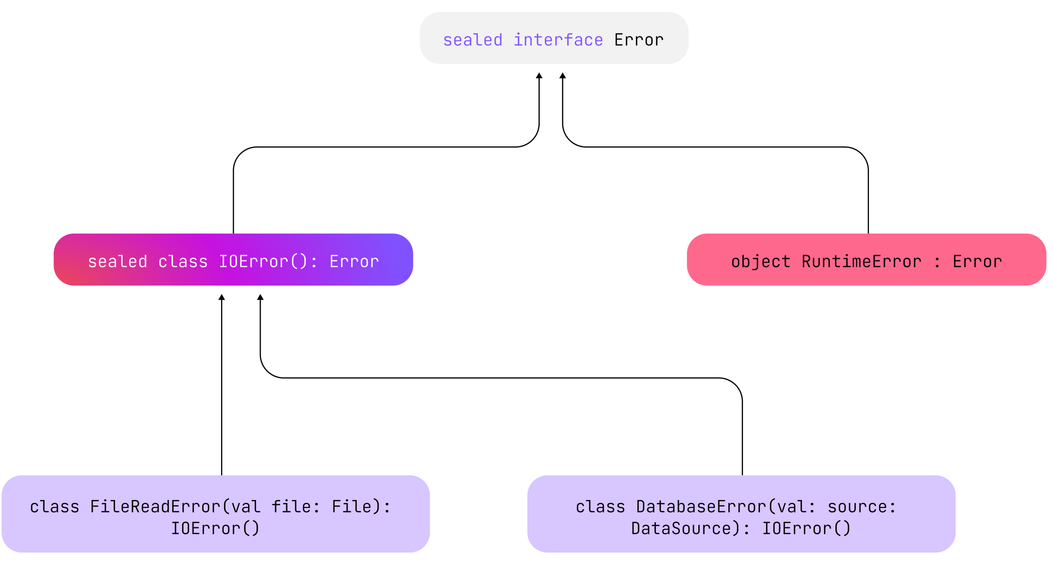
声明封闭类或接口

要声明一个封闭类或接口, 请使用 `sealed` 修饰符:

```
// 创建一个封闭接口
sealed interface Error

// 创建一个封闭类, 实现封闭接口 'Error'
sealed class IOError(): Error

// 定义子类, 继承封闭类 'IOError'
class FileReadError(val file: File): IOError()
class DatabaseError(val source: DataSource): IOError()

// 创建一个单子对象, 实现封闭接口 'Error'
object RuntimeError : Error
```

这个示例可以代表一个库的 API, 其中包含很多错误类, 以便类的使用者能够处理库可能抛出的错误. 如果这些错误类的继承层级包含在公开 API 可见的接口或抽象类, 那么就不能禁止其他开发者在他们的代码中实现这些接口或扩展这些抽象类. 由于库不知道在它外部定义的错误类, 因此库不能象它自己定义的类那样一致的处理这些外部定义的类. 如果将错误类的继承阶层封闭起来, 库的作者就能够确定的知道所有可能的错误类型, 并且能够确定以后不会出现其他错误类型. 但是, 使用 **封闭的** 错误类层级结构, 库的作者就能够确定他们知道了所有可能的错误类型, 而且之后也不会出现其他的错误类型.

示例代码中的层级关系如下:



封闭类和封闭接口层级结构示意图

构造器

封闭类本身永远是 抽象(abstract)类 ("抽象类" in "类"), 因此, 不能直接生成它的实例. 但是, 它可以包含或继承构造器. 这些构造器不是用来创建封闭类自身的实例, 而是用来创建它的子类. 我们来看看下面的例子, 有一个封闭类 `Error`, 以及它的几个子类, 我们创建这些子类的实例:

```
sealed class Error(val message: String) {
    class NetworkError : Error("Network failure")
    class DatabaseError : Error("Database cannot be reached")
    class UnknownError : Error("An unknown error has occurred")
}

fun main() {
    val errors = listOf(Error.NetworkError(), Error.DatabaseError(),
Error.UnknownError())
    errors.forEach { println(it.message) }
}
// 输出结果为
// Network failure
```

```
// Database cannot be reached
// An unknown error has occurred
```

你可以在你的封闭类中使用 `enum` (枚举类) 类, 用枚举常数来表示状态, 并提供更多细节信息. 每个枚举常数只存在 **单个** 实例, 而封闭类的子类可以有 **多个** 实例. 在下面的示例中, `sealed class Error` 和它的几个子类, 使用 `enum` 来表示错误的严重级别. 每个子类的构造器会初始化 `severity`, 并改变它的状态:

```
enum class ErrorSeverity { MINOR, MAJOR, CRITICAL }

sealed class Error(val severity: ErrorSeverity) {
    class FileReadError(val file: File): Error(ErrorSeverity.MAJOR)
    class DatabaseError(val source: DataSource):
Error(ErrorSeverity.CRITICAL)
    object RuntimeError : Error(ErrorSeverity.CRITICAL)
    // 这里可以添加更多错误类型
}
```

封闭类的构造器的 可见度 (可见度修饰符) 必须是: `protected` (默认值) 或 `private`:

```
sealed class IOError {
  // 封闭类的构造器默认可见度为 protected. 构造器在这个类和它的子类中可见.
  constructor() { /*...*/ }

  // private 构造器, 只在这个类中可见.
  // 在封闭类中使用 private 构造器, 可以更加严格的控制实例的创建, 实现类中特定
的初始化过程.
  private constructor(description: String): this() { /*...*/ }

  // 这里会发生错误, 因为在封闭类中不允许使用 public 和 internal 构造器
  // public constructor(code: Int): this() {}
}
```

继承

封闭类和接口的直接子类必须定义在同一个包之内. 可以是顶级位置, 也可以嵌套在任意多的其他有名称的类, 有名称的接口, 或有名称的对象之内. 子类可以设置为任意的 可见度 (可见度修饰符), 只要它们符合 Kotlin 中通常的类继承规则.

封闭类的子类必须拥有一个适当的限定名称. 不能是局部对象或匿名对象.

> `enum` 类不能扩展封闭类, 也不能扩展任何其他类. 但是, 它们可以实现封闭接口:
>
> ```
> sealed interface Error
>
> // 枚举类扩展封闭接口 'Error'
> enum class ErrorType : Error {
>     FILE_ERROR, DATABASE_ERROR
> }
> ```

这些限制不适用于非直接子类. 如果封闭的类一个直接子类没有标记为封闭, 那么它可以按照其修饰符允许的方式任意扩展:

```
// 封闭接口 'Error' 只在相同的模块和包中存在实现类
sealed interface Error

// 封闭类 'IOError' 扩展 'Error', 只能在相同的包中扩展 'IOError'
sealed class IOError(): Error

// 开放类 'CustomError' 扩展 'Error', 可以在 'CustomError' 可见的任何地方
扩展这个类
open class CustomError(): Error
```

跨平台项目中的继承

在 跨平台项目 (Kotlin 跨平台程序开发入门)中还存在一种继承限制: 封闭类的直接子类必须放在同一个 源代码集(Source Set) ("源代码集(Source Set)" in "Kotlin Multiplatform 项目结构的基础知识") 中. 这个限制适用于没有使用 `expect` 和 `actual` 修饰符 (预期声明与实际声明) 的封闭类.

如果封闭类声明为共通源代码集(common source set)中的 `expect`, 并且在平台相关的代码集内拥有 `actual` 实现类, 那么 `expect` 和 `actual` 的版本在各自的源代码集内都可以拥有子类. 此外, 如果你使用了层级结构(hierarchical structure), 你可以在 `expect` 和 `actual` 声明之间的任何源代码集内创建子类.

更多详情请参见 跨平台项目的层级结构(hierarchical structure) (层级项目结构).

和 when 表达式一起使用封闭类

when 表达式和封闭类一起使用时, Kotlin 编译器能够进行穷尽的检查, 是否覆盖了所有可能的情况. 这样的情况下, 你可以不必添加 else 分支:

```
// 封闭类和它的子类
sealed class Error {
    class FileReadError(val file: String): Error()
    class DatabaseError(val source: String): Error()
    object RuntimeError : Error()
}

//sampleStart
// 将错误输出到日志的函数
fun log(e: Error) = when(e) {
    is Error.FileReadError -> println("Error while reading file
${e.file}")
    is Error.DatabaseError -> println("Error while reading from
database ${e.source}")
    Error.RuntimeError -> println("Runtime error")
    // 不需要 `else` 分支, 因为已经覆盖了所有的可能情况
}
//sampleEnd

// 所有错误的列表
fun main() {
    val errors = listOf(
        Error.FileReadError("example.txt"),
        Error.DatabaseError("usersDatabase"),
        Error.RuntimeError
    )

    errors.forEach { log(it) }
}
```

> 在跨平台项目中, 如果与 when 表达式一起使用的封闭类, 是你的共通代码中的 预期声明 (预期声明与实际声明), 那么仍然需要 else 分支. 这是因为, actual 平台实现中的子类可以扩展封闭类, 但在共通代码中, 无法确定这些子类.

使用场景

我们来看看一些实际的使用场景, 封闭类和封闭接口可以非常有用.

管理 UI 应用程序中的状态

你可以使用封闭类来表示应用程序中的不同 UI 状态. 这种方法可以实现结构化并且安全的 UI 变更管理. 下面的例子演示如何管理不同的 UI 状态:

```
sealed class UIState {
    data object Loading : UIState()
    data class Success(val data: String) : UIState()
    data class Error(val exception: Exception) : UIState()
}

fun updateUI(state: UiState) {
    when (state) {
        is UIState.Loading -> showLoadingIndicator()
        is UIState.Success -> showData(state.data)
        is UIState.Error -> showError(state.exception)
    }
}
```

管理支付方式

在一些实际的商业应用程序中, 高效的处理各种支付方式是一种常见的需求. 你可以使用封闭类 和 `when` 表达式来实现这样的业务逻辑. 将不同的支付方式表达为封闭类的子类, 可以为交易过程的处理实现一个清晰而且易于管理的结构:

```
sealed class Payment {
    data class CreditCard(val number: String, val expiryDate:
String) : Payment()
    data class PayPal(val email: String) : Payment()
    data object Cash : Payment()
}

fun processPayment(payment: Payment) {
    when (payment) {
        is Payment.CreditCard ->
```

```
processCreditCardPayment(payment.number, payment.expiryDate)
        is Payment.PayPal -> processPayPalPayment(payment.email)
        is Payment.Cash -> processCashPayment()
    }
}
```

`Payment` 是一个封闭类, 表示电子商务系统中的各种支付方式: `CreditCard`, `PayPal`, 和 `Cash`. 每个子类可以拥有它独自的属性, 例如 `CreditCard` 有 `number` 和 `expiryDate`, `PayPal` 有 `email`.

`processPayment()` 函数演示如何处理不同的支付方式. 这种方案可以确保考虑到了所有可能的支付类型, 而且系统保持了灵活性, 可以在将来添加新的支付方式.

处理 API 请求/应答

你可以使用封闭类和封闭接口来实现一个用户认证系统, 它处理 API 的请求和应答. 用户认证系统有登入和登出功能. `ApiRequest` 封闭接口定义了特定的请求类型: `LoginRequest` 用于登入操作, `LogoutRequest` 用于登出操作. 封闭类, `ApiResponse`, 包括不同的应答场景: `UserSuccess`, 其中包含用户数据, `UserNotFound`, 表示用户不存在, `Error`, 表示失败. `handleRequest` 函数使用 `when` 表达式, 以一种类型安全的方式处理这些请求, `getUserById` 函数模拟用户检索:

```
// 引入必须的模块
import io.ktor.server.application.*
import io.ktor.server.resources.*

import kotlinx.serialization.*

// 定义封闭接口, 表示使用 Ktor 资源的 API 请求
@Resource("api")
sealed interface ApiRequest

@Serializable
@Resource("login")
data class LoginRequest(val username: String, val password: String)
: ApiRequest

@Serializable
@Resource("logout")
object LogoutRequest : ApiRequest

// 定义封闭类 ApiResponse, 包括具体的应答类型
```

```
sealed class ApiResponse {
    data class UserSuccess(val user: UserData) : ApiResponse()
    data object UserNotFound : ApiResponse()
    data class Error(val message: String) : ApiResponse()
}

// 用户数据类，在成功应答中使用
data class UserData(val userId: String, val name: String, val email:
String)

// 这个函数校验用户凭证（只为演示用）
fun isValidUser(username: String, password: String): Boolean {
    // 使用固定的校验逻辑（这只是一段演示代码）
    return username == "validUser" && password == "validPass"
}

// 这个函数使用具体的应答来处理 API 请求
fun handleRequest(request: ApiRequest): ApiResponse {
    return when (request) {
        is LoginRequest -> {
            if (isValidUser(request.username, request.password)) {
                ApiResponse.UserSuccess(UserData("userId",
"userName", "userEmail"))
            } else {
                ApiResponse.Error("Invalid username or password")
            }
        }
        is LogoutRequest -> {
            // 这个示例假设 logout 操作永远成功
            ApiResponse.UserSuccess(UserData("userId", "userName",
"userEmail")) // 演示用
        }
    }
}

// 这个函数模拟一个 getUserById 调用
fun getUserById(userId: String): ApiResponse {
    return if (userId == "validUserId") {
        ApiResponse.UserSuccess(UserData("validUserId", "John Doe",
```

```
"john@example.com"))
    } else {
        ApiResponse.UserNotFound
    }
    // 错误处理也会生成错误应答.
}

// 主函数，演示使用方法
fun main() {
    val loginResponse = handleRequest(LoginRequest("user", "pass"))
    println(loginResponse)

    val logoutResponse = handleRequest(LogoutRequest)
    println(logoutResponse)

    val userResponse = getUserById("validUserId")
    println(userResponse)

    val userNotFoundResponse = getUserById("invalidId")
    println(userNotFoundResponse)
}
```

泛型(Generic): in, out, where

最终更新: 2024/09/10

Kotlin 中的类也可以有类型参数, 与 Java 一样:

```
class Box<T>(t: T) {
    var value = t
}
```

要创建这样一个类的实例, 只需要指定类型参数:

```
val box: Box<Int> = Box<Int>(1)
```

但是, 如果类型参数可以通过推断得到, 比如, 通过构造器参数类型推断得到, 你可以省略类型参数:

```
val box = Box(1) // 1 的类型为 Int, 因此编译器知道类型为 Box<Int>
```

类型变异(Variance)

Java 的类型系统中, 最微妙最难于理解和使用的部分之一, 就是它的通配符类型(wildcard type) (参见 Java 泛型 FAQ (http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html)). Kotlin 中不存在这样的通配符类型. 而是使用声明处类型变异(declaration-site variance), 以及类型投射(type projection).

Java 中的类型变异(Variance)和通配符(Wildcard)

让我们思考一下为什么 Java 需要这些神秘的通配符类型. 首先, Java 中的泛型类型是 *不可变的 (invariant)*, 也就是说 `List<String>` *不是* `List<Object>` 的子类型. 因为, 如果 `List` 不是 *不可变的 (invariant)*, 那么下面的代码将可以通过编译, 但会在运行时导致一个异常, 那么 List 就并没有任何优于 Java 数组的地方了:

```
// Java
List<String> strs = new ArrayList<String>();

// 在编译期, Java 会在这里报告类型不匹配的错误.
List<Object> objs = strs;
```

```
// 如果不报告这个错误会怎么样 ?
// 那么我们就能够向 String 组成的 List 添加一个 Integer 类型的元素.
objs.add(1);

// 然后在运行期, Java 会抛出 ClassCastException 异常: Integer cannot be
cast to String
String s = strs.get(0);
```

Java 禁止上面示例中的做法, 以保证运行期的类型安全. 但这个原则背后存在一些隐含的影响. 比如, 我们来看看 `Collection` 接口的 `addAll()` 方法. 这个方法的签名应该是什么样的? 你可能会根据第一直觉, 将它定义为:

```
// Java
interface Collection<E> ... {
    void addAll(Collection<E> items);
}
```

但是这样的话, 你将无法进行下面这种操作(尽管它是绝对安全的):

```
// Java

// 如果 addAll 方法使用前面那种简单的定义, 下面的代码无法编译:
// Collection<String> is not a subtype of Collection<Object>
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from);
}
```

因此 `addAll()` 的签名定义实际上是这样的:

```
// Java
interface Collection<E> ... {
    void addAll(Collection<? extends E> items);
}
```

这里的 *通配符类型参数(wildcard type argument)* `? extends E` 表示, 该方法接受的参数是一个集合, 集合元素的类型是 `E` *或* `E` *的子类型*, 而不限于 `E` 本身. 这就意味着, 你可以安全地从集合元素中*读取* `E` (因为集合的元素是 `E` 的某个子类型的实例), 但 *不能写入* 到集合中去, 因为你不知道什么样的对象实例才能与这个 `E` 的未知子类型匹配. 尽管有这样的限制, 作为回报, 你得到了希望的功能:

`Collection<String>` *是* `Collection<? extends Object>` 的子类型. 也就是说, 指定了 *extends* 边界(*上* 边界)的通配符类型, 使得我们的类型成为一种 *协变(covariant)* 类型.

要理解这种模式的工作原理十分简单: 如果你只能从一个集合 *取得* 元素, 那么就可以使用一个 `String` 组成的集合, 并从中读取 `Object` 实例. 反过来, 如果你只能向集合 *放入* 元素, 那么就可以使用一个 `Object` 组成的集合, 并向其中放入 `String`: 在 Java 中有 `List<? super String>`, 它可以接受 `String`, 或 `String` 的任何父类型.

上面的后一种情况称为 *反向类型变异(contravariance)*, 对于 `List<? super String>`, 你只能调用那些接受 `String` 类型参数的方法 (比如, 可以调用 `add(String)`, 或 `set(int, String)`), 如果你对 `List<T>` 调用返回类型为 `T` 的方法时, 你得到的返回值将不会是 `String` 类型, 而是 `Object` 类型.

Joshua Bloch 在他的 Effective Java, 第 3 版 (http://www.oracle.com/technetwork/java/effectivejava-136174.html) 详细解释了这个问题, (第 31 条: "为增加 API 的灵活性, 应该使用限定范围的通配符类型(bounded wildcard)"). 他将那些只能 *读取* 的对象称为 *生产者(Producer)*, 将那些只能 *写入* 的对象称为 *消费者(Consumer)*. 他建议:

> "为尽量保证灵活性, 应该对代表生产者和消费者的输入参数使用通配符类型."

他还提出了下面的记忆口诀: *PECS*, 表示 *生产者(Producer)对应 Extends, 消费者(Consumer) 对应 Super*.

> 如果你使用一个生产者对象, 比如, `List<? extends Foo>`, 你将无法对这个对象调用 `add()` 或 `set()` 方法, 但这并不代表这个对象是 *值不变的(immutable)*: 比如, 你完全可以调用 `clear()` 方法来删除 List 内的所有元素, 因为 `clear()` 方法不需要任何参数.
>
> 通配符类型(或者其他任何的类型变异)唯一能够确保的仅仅是 *类型安全*. 对象值的不变性(Immutability)是与此完全不同的另一个问题.

声明处的类型变异(Declaration-site variance)

假设有一个泛型接口 `Source<T>`, 其中不存在任何接受 `T` 作为参数的方法, 仅有返回值为 `T` 的方法:

```
// Java
interface Source<T> {
    T nextT();
}
```

那么, 完全可以在 `Source<Object>` 类型的变量中保存一个 `Source<String>` 类型的实例 - 因为不存在对消费者方法的调用. 但 Java 不能理解这一点, 因此仍然禁止以下代码:

```
// Java
void demo(Source<String> strs) {
    Source<Object> objects = strs; // !!! 在 Java 中禁止这样的操作
    // ...
}
```

为了解决这个问题, 你需要将对象类型声明为 `Source<? extends Object>`, 这样其实是毫无意义的, 因为在这样修改之后, 你所能调用的方法与修改之前其实是完全一样的, 因此, 使用这样复杂的类型声明并未带来什么好处. 但编译器并不理解这一点.

在 Kotlin 中, 我们有办法将这种情况告诉编译器. 这种技术称为 *声明处的类型变异(declaration site variance)*: 你可以对 `Source` 的 *类型参数* `T` 添加注解, 来确保 `Source<T>` 的成员函数只会 *返回* (生产) `T` 类型, 而绝不会消费 `T` 类型. 为了实现这个目的, 可以对 `T` 使用 *out* 修饰符:

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // 这是 OK 的, 因为 T 是一个 out 类型参数
    // ...
}
```

一般规则是: 当 `C` 类的类型参数 `T` 声明为 `out` 时, 那么在 `C` 的成员函数中, `T` 类型只允许出现在 *输出* 位置, 这样的限制带来的回报就是, `C<Base>` 可以安全地用作 `C<Derived>` 的父类型.

也就是说, 你可以将 `C` 类称为, 在类型参数 `T` 上 *协变的(covariant)*, 或者说 `T` 是一个 *协变的(covariant)* 类型参数. 你可以将 `C` 类看作 `T` 类型对象的 *生产者*, 而不是 `T` 类型对象的 *消费者*.

out 修饰符称为 *协变注解(variance annotation)*, 而且, 由于这个注解出现在类型参数的声明处, 它提供了 *声明处的类型变异(declaration-site variance)*. 这种方案与 Java 中的 *使用处类型变异(use-site variance)* 刚好相反, 在 Java 中, 是类型使用处的通配符产生了类型的协变.

除了 `out` 之外, Kotlin 还提供了另一种类型变异注解: `in`. 这个注解导致类型参数 `反向类型变异(contravariant)`: 也就是说这个类型将只能被消费, 而不能被生产. 反向类型变异的一个很好的例子是 `Comparable`:

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 类型为 Double, 是 Number 的子类型
    // 因此, 你可以将 x 赋值给 Comparable<Double> 类型的变量
    val y: Comparable<Double> = x // OK!
}
```

in 和 *out* 关键字的意义看来是十分直观的(同样的关键字已经在 C# 中使用很长时间了), 因此, 前面提到的记忆口诀也没有必要了, 我们可以将它改写为更高的抽象层次:

存在主义 (https://en.wikipedia.org/wiki/Existentialism) 变形法则: 消费者进, 生产者出!:-)

译注: 上面两句翻译得不够好, 待校

类型投射(Type projection)

使用处的类型变异(Use-site variance): 类型投射(Type projection)

将声明类型参数 T 声明为 `out`, 就可以免去使用时子类化的麻烦, 这是十分方便的. 但是有些类 *不能* 限定为仅仅只返回 `T` 类型值! 关于这个问题, 一个很好的例子是 `Array` 类:

```
class Array<T>(val size: Int) {
    operator fun get(index: Int): T { ... }
    operator fun set(index: Int, value: T) { ... }
}
```

这个类对于类型参数 `T` 既不能协变, 也不能反向协变. 这就带来很大的不便. 我们来看看下面的函数:

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

这个函数应该将元素从一个 Array 复制到另一个 Array. 我们来试试使用一下这个函数:

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any)
//   ^ 这里发生编译错误, 期待的参数类型是 Array<Any>, 但实际类型是
Array<Int>
```

在这里, 你又遇到了熟悉的老问题: `Array<T>` 对于类型参数 `T` 是 *不可变的*, 因此 `Array<Int>` 和 `Array<Any>` 谁也不是谁的子类型. 为什么不是? 原因与以前一样, 因为 `copy` 函数内可能发生预想外的行为, 比如, 它可能会试图向 `from` 数组中写入一个 `String`, 这时如果你传入的实际参数是一个 `Int` 的数组, 就会导致一个 `ClassCastException`.

为了禁止 `copy()` 函数向 `from` 数组 *写入* 数据, 你可以这样:

```
fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

这种声明称为 *类型投射(type projection)*: 其含义是, `from` 不是一个单纯的数组, 而是一个被限制(*投射*)的数组. 你只能对这个数组调用那些返回值为类型参数 `T` 的方法, 在这个例子中, 只能调用 `get()` 方法. 这就是 *使用处的类型变异(use-site variance)* 的实现方案, 对应 Java 的 `Array<? extends Object>`, 但略为简单一些.

你也可以使用 `in` 关键字来投射一个类型:

```
fun fill(dest: Array<in String>, value: String) { ... }
```

`Array<in String>` 与 Java 的 `Array<? super String>` 相同. 也就是说, 你可以使用 `CharSequence` 数组, 或者 `Object` 数组作为 `fill()` 函数的参数.

星号投射(Star-projection)

有些时候, 你可能想表示你并不知道类型参数的任何信息, 但是仍然希望能够安全地使用它. 这里所谓"安全地使用"是指, 对泛型类型定义一个类型投射, 要求这个泛型类型的所有的实体实例, 都是这个投射的子类型.

对于这个问题, Kotlin 提供了一种语法, 称为 *星号投射(star-projection)*:

- 假如类型定义为 `Foo<out T : TUpper>`, 其中 `T` 是一个协变的类型参数, 上界(Upper Bound)为 `TUpper`, `Foo<*>` 等价于 `Foo<out TUpper>`. 它表示, 当 `T` 未知时, 你可以安全地从 `Foo<*>` 中 *读取* `TUpper` 类型的值.

- 假如类型定义为 Foo<in T>, 其中 T 是一个反向协变的类型参数, Foo<*> 等价于 Foo<in Nothing>. 它表示, 当 T 未知时, 你不能安全地向 Foo<*> *写入* 任何东西.

- 假如类型定义为 Foo<T : TUpper>, 其中 T 是一个协变的类型参数, 上界(Upper Bound)为 TUpper, 对于读取值的场合, Foo<*> 等价于 Foo<out TUpper>, 对于写入值的场合, 等价于 Foo<in Nothing>.

如果一个泛型类型中存在多个类型参数, 那么每个类型参数都可以单独的投射. 比如, 如果类型定义为 interface Function<in T, out U>, 你可以使用以下几种星号投射:

- Function<*, String>, 代表 Function<in Nothing, String>.
- Function<Int, *>, 代表 Function<Int, out Any?>.
- Function<*, *>, 代表 Function<in Nothing, out Any?>.

> 星号投射与 Java 的原生类型(raw type)非常类似, 但可以安全使用.

泛型函数

不仅类可以有类型参数. 函数一样可以有类型参数. 类型参数放在函数名称 *之前*:

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString(): String { // 扩展函数
    // ...
}
```

调用泛型函数时, 应该在函数名称 *之后* 指定调用端类型参数:

```
val l = singletonList<Int>(1)
```

如果可以通过程序上下文推断得到, 类型参数可以省略, 因此下面的例子也可以正确运行:

```
val l = singletonList(1)
```

泛型约束(Generic constraint)

对于一个给定的类型参数, 所允许使用的类型, 可以通过 *泛型约束(generic constraint)* 来限制.

上界(Upper Bound)

最常见的约束是 *上界(Upper Bound)*, 对应于 Java 中的 *extends* 关键字:

```
fun <T : Comparable<T>> sort(list: List<T>) {  ... }
```

冒号之后指定的类型就是类型参数的 *上界(Upper Bound)*: 表示对类型参数 `T`, 只允许使用 `Comparable<T>` 的子类型. 比如:

```
sort(listOf(1, 2, 3)) // 正确: Int 是 Comparable<Int> 的子类型
sort(listOf(HashMap<Int, String>())) // 错误: HashMap<Int, String> 不是 Comparable<HashMap<Int, String>> 的子类型
```

如果没有指定, 则默认使用的上界是 `Any?`. 在定义类型参数的尖括号内, 只允许定义唯一一个上界. 如果同一个类型参数需要指定多个上界, 这时需要使用单独的 *where* 子句:

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
    where T : CharSequence,
          T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```

传入的类型必须同时满足 `where` 子句中的所有条件. 在上面的示例中, `T` 类型必须 *同时* 实现 `CharSequence` 和 `Comparable` 接口.

确定不为 null 的类型

为了让与 Java 的泛型类和接口的互操作更加便利, Kotlin 允许将泛型类型参数为声明 **确定不为 null**.

要将泛型类型 `T` 声明为确定不为 null, 请使用 `& Any` 来声明这个类型. 例如: `T & Any`.

确定不为 null 的类型的 上界(Upper Bound) 必须是可以为 null 的类型.

确定不为 null 的类型的最常见的使用场景是, 你想要覆盖 override 一个包含 `@NotNull` 参数的 Java 方法. 例如, 考虑下面的 `load()` 方法:

```
import org.jetbrains.annotations.*;

public interface Game<T> {
    public T save(T x) {}
    @NotNull
    public T load(@NotNull T x) {}
}
```

要在 Kotlin 中成功的覆盖 `load()` 方法, 你需要将 `T1` 声明为确定不为 null:

```
interface ArcadeGame<T1> : Game<T1> {
    override fun save(x: T1): T1
    // T1 确定不为 null
    override fun load(x: T1 & Any): T1 & Any
}
```

如果只使用 Kotlin, 那么你不太可能需要明确的声明确定不为 null 的类型, 因为 Kotlin 的类型推断功能会帮你解决这个问题.

类型擦除

对使用泛型声明的代码, Kotlin 在编译期进行类型安全性检查. 在运行期, 泛型类型的实例不保存关于其类型参数的任何信息. 我们称之为, 类型信息 *被擦除* 了. 比如, `Foo<Bar>` 和 `Foo<Baz?>` 的实例, 其类型信息会被擦除, 只剩下 `Foo<*>`.

泛型的类型检查与类型转换

由于存在类型擦除的问题, 因此不存在一种通用的办法, 可以在运行期检查一个泛型类的实例是通过什么样的类型参数来创建的, 并且编译器禁止这样的 `is` 检查, 例如 `ints is List<Int>` 或 `list is T` (T 是类型参数). 但是, 你可以检查实例是否属于星号投射类型:

```
if (something is List<*>) {
    something.forEach { println(it) } // List 中元素的类型都被识别为
`Any?`
}
```

类似的, 如果(在编译期间)已经对一个实例的类型参数进行了静态检查, 你可以对泛型之外的部分进行 `is` 检查, 或类型转换. 注意, 下面的示例中省略了尖括号:

```
fun handleStrings(list: MutableList<String>) {
    if (list is ArrayList) {
        // `list` 会被智能转换为 `ArrayList<String>`
    }
}
```

对于不涉及类型参数的类型转换, 可以使用的相同语法, 但省略类型参数: `list as ArrayList`.

泛型函数调用的类型参数也只在编译期进行检查. 在函数内部, 类型参数不能用来进行类型检查, 而且向类型参数的类型转换 (`foo as T`) 也不做检查. 唯一的例外是使用 实体化的类型参数(Reified type parameter) (["实体化的类型参数(Reified type parameter)" in "内联函数(Inline Function)"]) 的内联函数, 会将它们的实际类型参数内联到每一个调用处. 因此可以对类型参数使用类型检查和转换. 但是, 在类型检查或转换内部使用的泛型类型实例, 仍然存在上述限制. 例如, 在类型检查 `arg is T` 中, 如果 `arg` 自身是一个泛型类型的实例, 它的类型参数仍然会被擦除.

```
//sampleStart
inline fun <reified A, reified B> Pair<*, *>.asPairOf(): Pair<A, B>?
{
    if (first !is A || second !is B) return null
    return first as A to second as B
}

val somePair: Pair<Any?, Any?> = "items" to listOf(1, 2, 3)

val stringToSomething = somePair.asPairOf<String, Any>()
val stringToInt = somePair.asPairOf<String, Int>()
val stringToList = somePair.asPairOf<String, List<*>>()
val stringToStringList = somePair.asPairOf<String, List<String>>()
// 这段代码能够编译, 但破坏了类型安全型!
// 请展开示例代码查看详情

//sampleEnd

fun main() {
    println("stringToSomething = " + stringToSomething)
    println("stringToInt = " + stringToInt)
    println("stringToList = " + stringToList)
    println("stringToStringList = " + stringToStringList)
```

```
    //println(stringToStringList?.second?.forEach() {it.length}) // 这里会抛出 ClassCastException 异常, 因为 list 中的元素不是字符串
}
```

未检查的类型转换

将类型转换为带有实际类型参数的泛型类型, 例如 `foo as List<String>`, 在运行期也无法进行检查. 如果不能由编译器直接推断得到类型安全, 但通过高层的程序逻辑能够保证, 那么可以使用这种未检查的类型转换. 请看下面的示例.

```
fun readDictionary(file: File): Map<String, *> = file.inputStream().use {
    TODO("Read a mapping of strings to arbitrary elements.")
}

// 我们把值为 `Int` 的 map 保存到了这个文件
val intsFile = File("ints.dictionary")

// 此处会出现编译警告: Unchecked cast: `Map<String, *>` to `Map<String, Int>`
val intsDictionary: Map<String, Int> = readDictionary(intsFile) as Map<String, Int>
```

最后一行中的类型转换会出现编译警告. 编译器无法对这个类型转换在运行期进行完整地检查, 因此不能保证 map 中的值是 `Int`.

为了避免这种未检查的类型转换, 你可以重新设计你的程序结构. 在上例中, 你可以声明 `DictionaryReader<T>` 和 `DictionaryWriter<T>` 接口, 然后对不同的数据类型提供类型安全的实现类. 你可以引入合理的抽象层次, 将未检查的类型转换, 从对接口的调用代码中, 移动到具体的实现类中. 正确使用 泛型类型变异(generic variance) 也可能有助于解决这类问题.

对于泛型函数, 使用 实体化的类型参数(Reified type parameter) ("实体化的类型参数(Reified type parameter)" in "内联函数(Inline Function)") 可以使得 `arg as T` 之类的类型转换变成可被检查的类型转换, 除非 `arg` 的类型带有 *它自己的* 类型参数, 并且在运行期间被擦除了.

对类型转换语句, 或这个语句所属的声明, 添加 `@Suppress("UNCHECKED_CAST")` 注解 (注解), 可以屏蔽未检查的类型转换导致的编译警告:

```
inline fun <reified T> List<*>.asListOfType(): List<T>? =
    if (all { it is T })
        @Suppress("UNCHECKED_CAST")
```

```
        this as List<T> else
        null
```

在 JVM 平台: 数组类型 (数组) (Array<Foo>) 保持了被擦除的数组元素类型信息, 将某个类型向数组类型进行的转换, 可以进行部分地检查: 数组元素可否为空, 以及数组元素本身的类型参数仍然会被擦除. 比如, 只要 foo 是一个数组, 并且元素类型是任意一种 List<*>, 无论元素可否为 null, 那么 foo as Array<List<String>?> 转换就会成功.

对类型参数的下划线操作符

可以对类型参数使用下划线操作符 _. 当其他类型已经明确指定时, 使用下划线操作符可以自动推断一个参数的类型:

```
abstract class SomeClass<T> {
    abstract fun execute() : T
}

class SomeImplementation : SomeClass<String>() {
    override fun execute(): String = "Test"
}

class OtherImplementation : SomeClass<Int>() {
    override fun execute(): Int = 42
}

object Runner {
    inline fun <reified S: SomeClass<T>, T> run() : T {
        return
S::class.java.getDeclaredConstructor().newInstance().execute()
    }
}

fun main() {
    // T 被推断为 String, 因为 SomeImplementation 继承自
SomeClass<String>
    val s = Runner.run<SomeImplementation, _>()
    assert(s == "Test")
```

```
    // T 被推断为 Int, 因为 OtherImplementation 继承自 SomeClass<Int>
    val n = Runner.run<OtherImplementation, _>()
    assert(n == 42)
}
```

嵌套类与内部类

最终更新: 2024/09/10

类可以嵌套在另一个类之内:

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

你也可以对接口进行嵌套. 类和接口的所有组合都是允许的: 可以在类中嵌套接口, 在接口中嵌套类, 以及在接口中嵌套接口.

```
interface OuterInterface {
    class InnerClass
    interface InnerInterface
}

class OuterClass {
    class InnerClass
    interface InnerInterface
}
```

内部类(Inner class)

嵌套类可以使用 `inner` 关键字来标记, 然后就可以访问它的外部类(outer class)的成员. 内部类会保存一个引用, 指向外部类的对象实例:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
```

```
    }
}

val demo = Outer().Inner().foo() // == 1
```

在内部类中使用 this 关键字会产生歧义, 关于如何消除这种歧义, 请参见 带限定符的 this 表达式 (this 表达式).

匿名内部类(Anonymous inner class)

匿名内部类的实例使用 对象表达式(object expression) ("对象表达式(Object expression)" in "对象表达式,对象声明,以及同伴对象") 来创建:

```
window.addMouseListener(object : MouseAdapter() {

    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
})
```

对于 JVM 平台, 如果这个对象是一个 Java 函数式接口的实例(也就是, 只包含唯一一个抽象方法的 Java 接口), 那么你可以使用带接口类型前缀的 Lambda 表达式来创建这个对象:

```
val listener = ActionListener { println("clicked") }
```

枚举类

最终更新: 2024/09/10

枚举类最基本的使用场景, 就是实现类型安全的枚举值:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

每个枚举常数都是一个对象. 枚举常数之间用逗号分隔.

由于每个枚举值都是枚举类的一个实例, 因此枚举值可以这样初始化:

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

匿名类

枚举常数可以定义它自己的匿名类, 这些匿名类可以拥有各自的方法, 也可以覆盖基类的方法:

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

如果枚举类中定义了任何成员, 需要用分号将枚举常数的定义与枚举类的成员定义分隔开.

在枚举类中实现接口

枚举类也可以实现接口 (但不能继承其他类), 对于接口的成员函数, 可以为所有的枚举常数提供一个共同的实现, 也可以在不同的枚举常数的匿名类中提供不同的实现. 枚举类实现接口时, 只需要在枚举类的声明中加入希望实现的接口名, 示例如下:

```
import java.util.function.BinaryOperator
import java.util.function.IntBinaryOperator

//sampleStart
enum class IntArithmetics : BinaryOperator<Int>, IntBinaryOperator {
    PLUS {
        override fun apply(t: Int, u: Int): Int = t + u
    },
    TIMES {
        override fun apply(t: Int, u: Int): Int = t * u
    };

    override fun applyAsInt(t: Int, u: Int) = apply(t, u)
}
//sampleEnd

fun main() {
    val a = 13
    val b = 31
    for (f in IntArithmetics.entries) {
        println("$f($a, $b) = ${f.apply(a, b)}")
    }
}
```

所有的枚举类都默认实现了 Comparable (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparable/index.html) 接口. 枚举常数值的大小顺序, 等于它在枚举类中的定义顺序. 详情请参见 排序(Ordering) ([排序(Ordering)](#)).

使用枚举常数

Kotlin 中的枚举类拥有编译器添加的合成的(synthetic)属性和方法, 可以列出枚举类中定义的所有枚举常数值, 可以通过枚举常数值的名称字符串得到对应的枚举常数值. 这些方法的签名如下(这里

假设枚举类名称为 `EnumClass`):

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.entries: EnumEntries<EnumClass> // 专门的 List<EnumClass>
```

下面是这些属性和方法的使用示例:

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    for (color in RGB.entries) println(color.toString()) // 输出结果为
RED, GREEN, BLUE
    println("The first color is: ${RGB.valueOf("RED")}") // 输出结果为
"The first color is: RED"
}
```

如果给定的名称不能匹配枚举类中定义的任何一个枚举常数值, `valueOf()` 方法会抛出 `IllegalArgumentException` 异常.

在 Kotlin 1.9.0 引入 `entries` 之前, 是使用 `values()` 函数来取得枚举常数的数组.

每个枚举常数值也拥有属性: `name` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-enum/name.html) 和 `ordinal` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-enum/ordinal.html), 可以取得它的名称, 以及在枚举类中声明的顺序(从 0 开始):

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    //sampleStart
    println(RGB.RED.name)    // 输出结果为 RED
    println(RGB.RED.ordinal) // 输出结果为 0
    //sampleEnd
}
```

你可以通过 `enumValues<T>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/enum-values.html) 和 `enumValueOf<T>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/enum-value-of.html) 函数, 以泛型方式取得枚举类中的常数:

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    println(enumValues<T>().joinToString { it.name })
}

printAllValues<RGB>() // 输出结果为 RED, GREEN, BLUE
```

> 关于内联函数(inline function)和实体化的类型参数(Reified type parameter), 详情请参见 内联函数 (内联函数(Inline Function)).

在 Kotlin 1.9.20 中, 引入了 `enumEntries<T>()` 函数, 作为 `enumValues<T>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/enum-values.html) 函数的未来的替代.

Kotlin 仍然支持 `enumValues<T>()` 函数, 但我们推荐你改为使用 `enumEntries<T>()` 函数, 因为它的性能损失较少. 每次调用 `enumValues<T>()` 都会创建一个新的数组, 而每次调用 `enumEntries<T>()` 都会返回相同的 List, 这样的性能要高效得多.

例如:

```
enum class RGB { RED, GREEN, BLUE }

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T : Enum<T>> printAllValues() {
    println(enumEntries<T>().joinToString { it.name })
}

printAllValues<RGB>()
// 输出结果为 RED, GREEN, BLUE
```

> `enumEntries<T>()` 函数是实验性功能. 要使用它, 需要标注 `@OptIn(ExperimentalStdlibApi)` 注解来表示使用者同意(Opt-in), 并 将语言版本设置为 1.9 以上 ("JVM 和 JS 任务支持的共通属性" in "Kotlin Gradle plugin 中的编译器选项").

内联的值类(Inline value class)

最终更新: 2024/09/10

如果将值封装到类中, 创建一些特定领域的类型, 有时候会非常有用. 但是, 这就会产生堆上的内存分配, 带来运行时的性能损失. 更坏的情况下, 如果被包装的类是基本类型, 那么性能损失会非常严重, 因为在运行时对基本类型本来可以进行极大地性能优化, 而它的包装类却不能享受这种好处.

为了解决这类问题, Kotlin 引入了一种特别的类, 称为 *内联类(inline class)*, 内联类是 基于值的类 (value-based class) (https://github.com/Kotlin/KEEP/blob/master/notes/value-classes.md) 的一个子集. 这种类没有标识符, 只用于包含值.

声明内联类时, 在类名称之前添加 `value` 修饰符:

```
value class Password(private val s: String)
```

要在 JVM 后端上声明内联类, 需要在类的定义之前使用 `value` 修饰符和 `@JvmInline` 注解:

```
// 针对 JVM 后端
@JvmInline
value class Password(private val s: String)
```

内联类必须拥有唯一的一个属性, 并在主构造器中初始化这个属性. 在运行期, 会使用这个唯一的属性来表达内联类的实例(关于运行期的内部表达, 请参见 下文):

```
// 'Password' 类的实例不会真实存在
// 在运行期, 'securePassword' 只包含 'String'
val securePassword = Password("Don't try this in production")
```

这就是内联类的主要功能, 受 *内联* 这个名称的启发而来: 类中的数据被 *内联* 到使用它的地方 (类似于 内联函数 ([内联函数(Inline Function)]()) 的内容被内联到调用它的地方).

成员

内联类支持与通常的类相同的功能. 具体来说, 内联类可以声明属性和函数, 也可以有 `init` 代码段和次级构造器(secondary constructor) (["次级构造器(secondary constructor)" in "类"]()):

```
@JvmInline
value class Person(private val fullName: String) {
```

```
    init {
        require(fullName.isNotEmpty()) {
            "Full name shouldn't be empty"
        }
    }

    constructor(firstName: String, lastName: String) :
this("$firstName $lastName") {
        require(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }

    val length: Int
        get() = fullName.length

    fun greet() {
        println("Hello, $fullName")
    }
}

fun main() {
    val name1 = Person("Kotlin", "Mascot")
    val name2 = Person("Kodee")
    name1.greet() // `greet()` 函数会作为静态方法来调用
    println(name2.length) // 属性的取值函数会作为静态方法来调用
}
```

内联类的属性不能拥有 后端域变量 ("属性的后端域变量(Backing Field)" in "属性(Property)"). 只能拥有简单的计算属性 (不能拥有 `lateinit` 属性或委托属性)

继承

内联类允许继承接口:

```
interface Printable {
    fun prettyPrint(): String
}
```

```
@JvmInline
value class Name(val s: String) : Printable {
    override fun prettyPrint(): String = "Let's $s!"
}

fun main() {
    val name = Name("Kotlin")
    println(name.prettyPrint()) // 仍然是调用静态方法
}
```

禁止内联类参与类继承. 也就是说, 内联类不能继承其他类, 而且它永远是 `final` 类, 不能被其他类继承.

内部表达

在通常的代码中, Kotlin 编译器会对每个内联类保留一个 *包装*. 内联类的实例在运行期可以表达为这个包装, 也可以表达为它的底层类型. 类似于 `Int` 可以 表达 ("数值类型在 JVM 平台的内部表达" in "数值类型") 为基本类型 `int`, 也可以表达为包装类 `Integer`.

Kotlin 编译器会优先使用底层类型而不是包装类, 这样可以产生最优化的代码, 运行时的性能也会最好. 但是, 有些时候会需要保留包装类. 一般来说, 当内联类被用作其他类型时, 它会被装箱(box).

```
interface I

@JvmInline
value class Foo(val i: Int) : I

fun asInline(f: Foo) {}
fun <T> asGeneric(x: T) {}
fun asInterface(i: I) {}
fun asNullable(i: Foo?) {}

fun <T> id(x: T): T = x

fun main() {
    val f = Foo(42)

    asInline(f)    // 拆箱: 用作 Foo 本身
    asGeneric(f)   // 被装箱: 被用作泛型类型 T
```

```
    asInterface(f) // 被装箱: 被用作类型 I
    asNullable(f)  // 被装箱: 被用作 Foo?, 这个类型与 Foo 不同

    // 下面的例子中, 'f' 首先被装箱(传递给 'id' 函数), 然后被拆箱 (从 'id'
函数返回)
    // 最终, 'c' 中包含拆箱后的表达(也就是 '42'), 与 'f' 一样
    val c = id(f)
}
```

由于内联类可以表达为底层类型和包装类两种方式, 引用相等性 ("引用相等" in "相等判断") 对于内联类是毫无意义的, 因此禁止对内联类进行引用相等性判断操作.

内联类也可以使用泛型类型参数作为底层类型. 这种情况下, 编译器将它映射为 `Any?`, 或者更一般的说, 映射为类型参数的上界(Upper Bound).

```
@JvmInline
value class UserId<T>(val value: T)

fun compute(s: UserId<String>) {} // 编译器生成的代码是 fun compute-<hashcode>(s: Any?)
```

函数名称混淆

由于内联类被编译为它的底层类型, 因此可能会导致一些令人难以理解的错误, 比如, 意料不到的平台签名冲突:

```
@JvmInline
value class UInt(val x: Int)

// 在 JVM 平台上表达为 'public final void compute(int x)'
fun compute(x: Int) { }

// 在 JVM 平台上也表达为 'public final void compute(int x)'!
fun compute(x: UInt) { }
```

为了解决这种问题, 使用内联类的函数会被进行名称 *混淆*, 方法是对函数名添加一些稳定的哈希值. 因此, `fun compute(x: UInt)` 会表达为 `public final void compute-<hashcode>(int x)`, 然后就解决了函数名称的冲突问题.

在 Java 代码中调用

你可以在 Java 代码中调用接受内联类为参数的函数. 为了实现这一点, 你需要手动禁止函数名称混淆: 在函数声明之前添加 `@JvmName` 注解:

```
@JvmInline
value class UInt(val x: Int)

fun compute(x: Int) { }

@JvmName("computeUInt")
fun compute(x: UInt) { }
```

内联类与类型别名

初看起来, 内联类似乎非常象 类型别名 (类型别名). 确实, 它们都声明了一个新的类型, 并且在运行期都表达为各自的底层类型.

但是, 主要的差别在于, 类型别名与它的底层类型是 *赋值兼容* 的 (与同一个底层类型的另一个类型别名, 也是兼容的), 而内联类不是如此.

也就是说, 内联类会生成一个真正的 *新* 类型, 相反, 类型别名只是给既有的类型定义了一个新的名字 (也就是别名):

```
typealias NameTypeAlias = String

@JvmInline
value class NameInlineClass(val s: String)

fun acceptString(s: String) {}
fun acceptNameTypeAlias(n: NameTypeAlias) {}
fun acceptNameInlineClass(p: NameInlineClass) {}

fun main() {
    val nameAlias: NameTypeAlias = ""
    val nameInlineClass: NameInlineClass = NameInlineClass("")
    val string: String = ""

    acceptString(nameAlias) // 正确: 需要底层类型的地方, 可以传入类型别名
    acceptString(nameInlineClass) // 错误: 需要底层类型的地方, 不能传入内
```

```
联类

    // 反过来:
    acceptNameTypeAlias(string) // 正确: 需要类型别名的地方, 可以传入底层
类型
    acceptNameInlineClass(string) // 错误: 需要内联类的地方, 不能传入底层
类型
}
```

内联类与代理

对于接口, 允许将它的实现代理给内联类的内联值:

```
interface MyInterface {
    fun bar()
    fun foo() = "foo"
}

@JvmInline
value class MyInterfaceWrapper(val myInterface: MyInterface) :
MyInterface by myInterface

fun main() {
    val my = MyInterfaceWrapper(object : MyInterface {
        override fun bar() {
            // 函数体
        }
    })
    println(my.foo()) // 输出为 "foo"
}
```

对象表达式,对象声明,以及同伴对象

最终更新: 2024/09/10

有时你需要创建一个对象, 这个对象在某个类的基础上略做修改, 但又不希望仅仅为了这一点点修改就明确地声明一个新类. Kotlin 对这种问题使用 *对象表达式(object expression)* 和 *对象声明 (object declaration)* 来解决.

对象表达式(Object expression)

对象表达式(object expression) 会为匿名类创建对象 , 匿名类就是指没有明确使用 `class` 声明的类. 这些类适合一次性使用. 你可以从头开始定义这种类, 也可以从既有的类继承, 或者实现接口. 匿名类的实例称为 *匿名对象*, 因为它们通过表达式来定义, 而不是通过名称.

从头创建匿名对象

对象表达式以 `object` 关键字起始.

如果你只是需要一个对象, 而不需要任何基类型, 可以将这个对象的成员写在 `object` 之后的大括号内:

```
fun main() {
//sampleStart
    val helloWorld = object {
        val hello = "Hello"
        val world = "World"
        // 对象表达式继承 Any 类型, 因此对 `toString()` 函数需要
`override`
        override fun toString() = "$hello $world"
    }

    print(helloWorld)
//sampleEnd
}
```

从基类继承匿名对象

要创建一个继承自某个类(或多个类)的匿名类的对象, 需要在 `object` 关键字和冒号(`:`)之后指定基类. 然后实现或覆盖基类的成员, 就和你在 继承 (继承) 这个基类时一样:

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { /*...*/ }

    override fun mouseEntered(e: MouseEvent) { /*...*/ }
})
```

如果某个基类有构造器, 那么必须向构造器传递适当的参数. 通过冒号之后的逗号分隔的类型列表, 可以指定多个基类:

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { /*...*/ }

val ab: A = object : A(1), B {
    override val y = 15
}
```

将匿名对象用作返回类型或值类型

如果匿名对象用作局部的, 或 private ("包" in "可见度修饰符") 但不 inline (内联函数(Inline Function)) 声明 (函数或属性) 的类型, 那么通过这个函数或属性的返回值, 可以访问匿名对象的成员:

```
class C {
    private fun getObject() = object {
        val x: String = "x"
    }

    fun printX() {
        println(getObject().x)
    }
}
```

如果这个函数或属性是 public 的, 或 private 并且 inline 的, 那么它的真实类型为:

- 如果匿名对象没有声明基类型, 则类型为 `Any`

- 如果匿名对象声明了唯一一个基类型, 则类型为这个基类型
- 如果匿名对象声明了多个基类型, 则需要为这个函数或属性明确声明类型

在这些情况中, 通过这个函数或属性的返回值, 对于匿名对象新添加的成员, 不可访问. 对于匿名对象覆盖的成员, 如果定义在这个函数或属性的真实类型中, 则可以访问:

```
interface A {
    fun funFromA() {}
}
interface B

class C {
    // 返回类型为 Any; x 不可访问
    fun getObject() = object {
        val x: String = "x"
    }

    // 返回类型为 A; x 不可访问
    fun getObjectA() = object: A {
        override fun funFromA() {}
        val x: String = "x"
    }

    // 返回类型为 B; funFromA() 和 x 都不可访问
    fun getObjectB(): B = object: A, B { // 这里需要明确声明返回类型
        override fun funFromA() {}
        val x: String = "x"
    }
}
```

通过匿名对象访问变量

对象表达式内的代码可以访问创建这个对象的代码范围内的变量:

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0
```

```
    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}
```

对象声明(Object declaration)

单例模式 (http://en.wikipedia.org/wiki/Singleton_pattern) 在有些情况下可能是很有用的, Kotlin 可以非常便利地声明一个单例:

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

这样的代码称为一个 *对象声明(object declaration)*, 在 `object` 关键字之后必须指定对象名称. 与变量声明类似, 对象声明不是一个表达式, 因此不能用在赋值语句的右侧.

对象声明中的初始化处理是线程安全的(thread-safe), 而且会在对象初次访问时完成初始化处理.

要引用这个对象, 直接使用它的名称:

```
DataProviderManager.registerDataProvider(...)
```

这样的对象也可以指定基类:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }
```

```
    override fun mouseEntered(e: MouseEvent) { ... }
}
```

> 对象声明不可以是局部的(也就是说, 不可以直接嵌套在函数之内), 但可以嵌套在另一个对象声明之内, 或者嵌套在另一个非内部类(non-inner class)之内.

数据对象

如果在 Kotlin 中打印一个普通的 `object` 声明, 它的字符串表达包含对象的名称和 hash 值:

```
object MyObject

fun main() {
    println(MyObject) // 输出结果为: MyObject@1f32e575
}
```

和 数据类 ([数据类(Data Class)](#)) 一样, 你可以使用 `data` 修饰符标记 `object` 声明. 这个修饰符会让编译器为你的对象生成一系列的函数:

- `toString()` 返回数据对象的名称
- `equals()`/`hashCode()` 函数对

> 你不可以为 `data object` 的 `equals` 或 `hashCode` 函数提供自定义实现.

数据对象的 `toString()` 函数会返回对象的名称:

```
data object MyDataObject {
    val x: Int = 3
}

fun main() {
    println(MyDataObject) // 输出结果为 MyDataObject
}
```

`data object` 的 `equals()` 函数会保证你的 `data object` 的所有对象都被看作相等. 大多数情况下, 你的数据对象在运行期只会存在单个实例 (毕竟, `data object` 声明的就是一个单子(singleton)). 但

是, 在某些特殊情况下, 也可以在运行期生成相同类型的其他对象 (例如, 通过 `java.lang.reflect` 使用平台的反射功能, 或通过底层使用了这个 API 的 JVM 序列化库), 这个功能可以确保这些对象被当作相等.

> ⚠ 请确保只对 `data objects` 进行结构化的相等比较 (使用 `==` 操作符), 而不要进行引用相等比较 (使用 `===` 操作符). 如果数据对象在运行期有一个以上的实例存在, 这样可以帮助你避免错误.

```
import java.lang.reflect.Constructor

data object MySingleton

fun main() {
    val evilTwin = createInstanceViaReflection()

    println(MySingleton) // 输出结果为 MySingleton
    println(evilTwin) // 输出结果为 MySingleton

    // 即使一个库强行创建了 MySingleton 的第二个实例, 它的 `equals` 方法也
会返回 true:
    println(MySingleton == evilTwin) // 输出结果为 true

    // 不要使用 === 比较数据对象.
    println(MySingleton === evilTwin) // 输出结果为 false
}

fun createInstanceViaReflection(): MySingleton {
    // Kotlin 的反射功能不允许创建数据对象的实例.
    // 这段代码 "强行" 创建新的 MySingleton 实例 (也就是通过 Java 平台的反
射功能)
    // 在你的代码中一定不要这样做!
    return (MySingleton.javaClass.declaredConstructors[0].apply {
isAccessible = true } as Constructor<MySingleton>).newInstance()
}
```

编译器生成的 `hashCode()` 函数的行为与 `equals()` 函数保持一致, 因此一个 `data object` 的所有运行期实例都拥有相同的 hash 值.

数据对象与数据类的不同

尽管 `data object` 和 `data class` 声明经常一起使用, 而且很相似, 但对于 `data object` 有一些函数没有生成:

- 没有 `copy()` 函数. 因为 `data object` 声明通常用作单子对象, 因此不会生成 `copy()` 函数. 这种单子模式将一个类限定为只有单个实例, 如果允许创建实例的拷贝, 就破坏了只存在单个实例的原则.
- 没有 `componentN()` 函数. 与 `data class` 不同, `data object` 没有任何数据属性. 对这种没有数据属性的对象进行解构是没有意义的, 因此不会生成 `componentN()` 函数.

在封闭层级结构(Sealed Hierarchy)中使用数据对象

数据对象声明非常适合在封闭层级结构(Sealed Hierarchy) 中使用, 例如 封闭类或封闭接口 (封闭类(Sealed Class)与封闭接口(Sealed Interface)), 这样的方式允许你声明数据类和数据对象, 并保持对称性. 在这个示例中, 将 `EndOfFile` 声明为 `data object`, 而不是普通的 `object`, 代表它自动拥有 `toString()` 函数, 不需要手动的覆盖这个函数:

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // 输出结果为 Number(number=7)
    println(EndOfFile) // 输出结果为 EndOfFile
}
```

同伴对象(Companion Object)

一个类内部的对象声明, 可以使用 `companion` 关键字标记为同伴对象:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

我们可以直接使用类名称作为限定符来访问同伴对象的成员:

```
val instance = MyClass.create()
```

同伴对象的名称可以省略, 如果省略, 则会使用默认名称 `Companion`:

```
class MyClass {
    companion object { }
}

val x = MyClass.Companion
```

类的成员可以访问对应的同伴对象的私有成员.

直接使用一个类的名称时 (而不是将它用作另一个名称前面的限定符) 会被看作是这个类的同伴对象的引用 (无论同伴对象有没有名称):

```
class MyClass1 {
    companion object Named { }
}

val x = MyClass1

class MyClass2 {
    companion object { }
}

val y = MyClass2
```

注意, 虽然同伴对象的成员看起来很像其他语言中的类的静态成员(static member), 但在运行时期, 这些成员仍然是真实对象的实例的成员, 它们与静态成员是不同的, 举例来说, 它还可以实现接口:

```
interface Factory<T> {
    fun create(): T
}

class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
```

```
}

val f: Factory<MyClass> = MyClass
```

但是, 如果使用 `@JvmStatic` 注解, 你可以让同伴对象的成员在 JVM 上被编译为真正的静态方法(static method)和静态域(static field). 详情请参见 与 Java 的互操作性 ("静态域(Static Fields)" in "在 Java 中调用 Kotlin 代码").

对象表达式与对象声明在语义上的区别

对象表达式与对象声明在语义上存在一个重要的区别:

- 对象表达式则会在使用处 *立即* 执行(并且初始化).
- 对象声明是 *延迟(lazily)* 初始化的, 只会在首次访问时才会初始化.
- 同伴对象会在对应的类被装载(解析)时初始化, 语义上等价于 Java 的静态初始化代码块(static initializer).

委托

最终更新: 2024/09/10

委托模式 (https://en.wikipedia.org/wiki/Delegation_pattern) 已被实践证明为类继承模式之外的另一种很好的替代方案, Kotlin 直接支持委托模式, 因此你不必再为了实现委托模式而手动编写那些无聊的样板代码(Boilerplate Code)了.

比如, `Derived` 类可以实现 `Base` 接口, 将接口所有的 public 成员委托给一个指定的对象:

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

`Derived` 类声明的基类列表中的 `by` 子句表示, `b` 将被保存在 `Derived` 的对象实例内部, 而且编译器将会生成继承自 `Base` 接口的所有方法, 并将调用转发给 `b`.

覆盖由委托实现的接口成员

函数和属性的 覆盖 ("方法的覆盖" in "继承") 会如你预期的那样工作: 编译器将会使用你的 `override` 实现, 而不会使用委托对象中的实现. 如果你想要在 `Derived` 中添加一段函数覆盖 `override fun printMessage() { print("abc") }`, 那么上面程序中调用 `printMessage` 时的打印结果将是 *abc*, 而不是 *10*:

```
interface Base {
    fun printMessage()
    fun printMessageLine()
```

```
}

class BaseImpl(val x: Int) : Base {
    override fun printMessage() { print(x) }
    override fun printMessageLine() { println(x) }
}

class Derived(b: Base) : Base by b {
    override fun printMessage() { print("abc") }
}

fun main() {
    val b = BaseImpl(10)
    Derived(b).printMessage()
    Derived(b).printMessageLine()
}
```

注意, 使用上述方式覆盖的接口成员, 在委托对象的成员函数内无法调用. 委托对象的成员函数内, 只能访问它自己的接口方法实现:

```
interface Base {
    val message: String
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override val message = "BaseImpl: x = $x"
    override fun print() { println(message) }
}

class Derived(b: Base) : Base by b {
    // 在 b 的 `print` 方法实现中无法访问这个属性
    override val message = "Message of Derived"
}

fun main() {
    val b = BaseImpl(10)
    val derived = Derived(b)
    derived.print()
```

```
    println(derived.message)
}
```

更多信息请参见 委托属性 (委托属性).

委托属性

最终更新: 2024/09/10

有许多非常具有共性的属性, 虽然你可以在每个需要这些属性的类中手工地实现它们, 但是, 如果能够只实现一次, 然后将它放在库中, 供所有需要者重复使用, 那将会很有帮助. 例如:

- *延迟加载(lazy)* 属性: 属性值只在初次访问时才会计算.
- *可观察(observable)* 属性: 属性发生变化时, 监听器会收到通知.
- 将多个属性保存在一个 *map* 内, 而不是将每个属性保存在一个独立的域内.

为了解决这些问题(以及其它问题), Kotlin 允许 *委托属性(delegated property)*:

```
class Example {
    var p: String by Delegate()
}
```

委托属性的语法是: `val/var <property name>: <Type> by <expression>`. 其中 `by` 关键字之后的表达式就是 *委托*, 属性的 `get()` 方法(以及 `set()` 方法) 将被委托给这个对象的 `getValue()` 和 `setValue()` 方法. 属性委托不必实现接口, 但必须提供 `getValue()` 函数(对于 `var` 属性, 还需要 `setValue()` 函数).

示例:

```
import kotlin.reflect.KProperty

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>):
String {
        return "$thisRef, thank you for delegating
'${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>,
value: String) {
        println("$value has been assigned to '${property.name}' in
$thisRef.")
```

```
    }
}
```

如果属性 p 委托给一个 Delegate 的实例, 那么当你读取属性值时, 就会调用到 Delegate 的 getValue() 函数. 此时函数收到的第一个参数将是你访问的属性 p 所属的对象实例, 第二个参数将是 p 属性本身的描述信息(比如, 你可以从这里得到属性名称).

```
val e = Example()
println(e.p)
```

这段代码的打印结果将是:

```
Example@33a17727, thank you for delegating 'p' to me!
```

类似的, 当你向属性 p 赋值时, 将会调用到 setValue() 函数. 这个函数收到的前两个参数与 getValue() 函数相同, 第三个参数将是即将赋给属性的新值:

```
e.p = "NEW"
```

这段代码的打印结果将是:

```
NEW has been assigned to 'p' in Example@33a17727.
```

对属性委托对象的要求, 详细的说明请参见下文.

你可以在函数内, 或者一个代码段内定义委托属性, 委托属性不一定需要是类的成员. 参见 示例.

标准委托

Kotlin 标准库中提供了一些工厂方法, 可以实现几种很有用的委托.

延迟加载(Lazy)属性

lazy() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/lazy.html) 是一个函数, 接受一个 Lambda 表达式作为参数, 返回一个 Lazy<T> 类型的实例, 这个实例可以作为一个委托, 实现延迟加载(lazy)属性. 第一次调用 get() 时, 将会执行 lazy() 函数受到的 Lambda 表达式, 然后会记住这次执行的结果. 以后所有对 get() 的调用都只会简单地返回以前记住的结果.

```
val lazyValue: String by lazy {
    println("computed!")
```

```
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

默认情况下, 延迟加载(lazy)属性的计算是 *同步的(synchronized)*: 属性值只会在唯一一个线程内计算, 但所有线程都将得到同样的属性值. 如果委托的初始化计算不需要同步, 多个线程可以同时执行初始化计算, 那么可以向`lazy()` 函数传入一个 `LazyThreadSafetyMode.PUBLICATION` 参数.

如果你确信初期化计算只可能发生在你访问属性的相同线程之内, 那么可以使用 `LazyThreadSafetyMode.NONE` 模式. 这种模式不会保持线程同步, 因此不会带来这方面的性能损失.

可观察(Observable)属性

`Delegates.observable()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.properties/-delegates/observable.html) 函数接受两个参数: 第一个是初始化值, 第二个是属性值变化事件的响应器(handler).

每次你向属性赋值时, 响应器(handler)都会被调用(在属性赋值处理完成 *之后*). 响应器收到三个参数: 被赋值的属性, 赋值前的旧属性值, 以及赋值后的新属性值:

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main() {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

如果你希望拦截属性的赋值操作, 并且还能够 *否决* 赋值操作, 那么不要使用 `observable()` 函数, 而应该改用 `vetoable()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.properties/-delegates/vetoable.html) 函数. 传递给 `vetoable` 函数的事件响应器, 会在属性赋值处理执行 *之前* 被调用.

委托给另一个属性

属性可以将它的 get 和 set 方法委托到另一个属性. 这种委托可以用于顶级属性和类属性 (包括成员属性和扩展属性). 委托属性可以是:

- 顶级属性
- 同一个类的成员属性, 或扩展属性
- 另一个类的成员属性, 或扩展属性

要将一个属性委托到另一个属性, 请在委托名称中使用 `::` 限定符, 比如, `this::delegate` 或 `MyClass::delegate`.

```
var topLevelInt: Int = 0
class ClassWithDelegate(val anotherClassInt: Int)

class MyClass(var memberInt: Int, val anotherClassInstance:
ClassWithDelegate) {
    var delegatedToMember: Int by this::memberInt
    var delegatedToTopLevel: Int by ::topLevelInt

    val delegatedToAnotherClass: Int by
anotherClassInstance::anotherClassInt
}
var MyClass.extDelegated: Int by ::topLevelInt
```

这种功能的用途是, 比如, 如果你希望修改属性名称, 同时又保持向后兼容: 这时可以引入一个新的属性, 将旧的属性标注 `@Deprecated` 注解, 然后将它的实现委托给新属性.

```
class MyClass {
   var newName: Int = 0
   @Deprecated("Use 'newName' instead", ReplaceWith("newName"))
   var oldName: Int by this::newName
```

```
}
fun main() {
   val myClass = MyClass()
   // 注意: 'oldName: Int' 已废弃.
   // 请改为使用 'newName'
   myClass.oldName = 42
   println(myClass.newName) // 42
}
```

将多个属性保存在一个 Map 内

有一种常见的使用场景是将多个属性的值保存在一个 map 之内. 在应用程序解析 JSON, 或者执行某些动态(dynamic)任务时, 经常会出现这样的需求. 这种情况下, 你可以使用 map 实例本身作为属性的委托.

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

上例中, 类的构造器接受一个 map 实例作为参数:

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

委托属性将从这个 map 中读取属性值, 使用属性名称字符串作为 key 值:

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}

fun main() {
    val user = User(mapOf(
        "name" to "John Doe",
        "age"  to 25
```

```
    ))
//sampleStart
    println(user.name) // 打印结果为: "John Doe"
    println(user.age)  // 打印结果为: 25
//sampleEnd
}
```

如果不用只读的 `Map`, 而改用值可变的 `MutableMap`, 那么也可以用作 `var` 属性的委托:

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

局部的委托属性(Local Delegated Property)

你可以将局部变量声明为委托属性. 比如, 你可以为局部变量添加延迟加载的能力:

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

`memoizedFoo` 变量直到初次访问时才会被计算. 如果 `someCondition` 的判定结果为 false, 那么 `memoizedFoo` 变量完全不会被计算.

属性委托的前提条件

对于一个 *只读* 属性 (`val` 属性), 它的委托应该提供 `getValue` 操作符函数, 参数如下:

- `thisRef` 参数, 类型必须与 *属性所属的类* 相同, 或者是它的基类 (对于扩展属性, 参数类型必须与被扩展的类型相同, 或者是它的基类).
- `property` 参数, 类型必须是 `KProperty<*>`, 或者是它的基类.

`getValue()` 函数的返回值类型必须与属性类型相同(或者是它的子类型).

```
class Resource

class Owner {
    val valResource: Resource by ResourceDelegate()
}

class ResourceDelegate {
    operator fun getValue(thisRef: Owner, property: KProperty<*>):
Resource {
        return Resource()
    }
}
```

对于一个 *值可变(mutable)* 属性(`var` 属性), 除 `getValue` 函数之外, 它的委托还必须另外再提供一个 `setValue` 操作符函数, 参数如下:

- `thisRef` 参数, 类型必须与 *属性所属的类* 相同, 或者是它的基类 (对于扩展属性, 参数类型必须与被扩展的类型相同, 或者是它的基类).
- `property` 参数, 类型必须是 `KProperty<*>`, 或者是它的基类.
- `value` 参数, 类型必须与属性类型相同(或者是它的基类).

```
class Resource

class Owner {
    var varResource: Resource by ResourceDelegate()
}

class ResourceDelegate(private var resource: Resource = Resource())
{
    operator fun getValue(thisRef: Owner, property: KProperty<*>):
Resource {
        return resource
    }
    operator fun setValue(thisRef: Owner, property: KProperty<*>,
value: Any?) {
        if (value is Resource) {
```

```
            resource = value
        }
    }
}
```

getValue() 和 setValue() 函数可以是委托类的成员函数, 也可以是它的扩展函数. 如果你需要将属性委托给一个对象, 而这个对象本来没有提供这些函数, 这时使用扩展函数会更便利一些. 这两个函数都需要标记为 operator.

通过使用 Kotlin 标准库中的 ReadOnlyProperty 和 ReadWriteProperty 接口, 可以用匿名对象的方式创建委托, 而不必创建新类. 这些接口提供了需要的方法: getValue() 声明在 ReadOnlyProperty 接口中; ReadWriteProperty 继承了这个接口, 然后增加了 setValue() 方法. 因此在需要 ReadOnlyProperty 的地方, 你也可以使用 ReadWriteProperty.

```
fun resourceDelegate(resource: Resource = Resource()):
ReadWriteProperty<Any?, Resource> =
    object : ReadWriteProperty<Any?, Resource> {
        var curValue = resource
        override fun getValue(thisRef: Any?, property:
KProperty<*>): Resource = curValue
        override fun setValue(thisRef: Any?, property: KProperty<*>,
value: Resource) {
            curValue = value
        }
    }

val readOnlyResource: Resource by resourceDelegate()  // 此处
ReadWriteProperty 被转换为 val
var readWriteResource: Resource by resourceDelegate()
```

编译器对委托属性的翻译规则

委托属性的底层实现是, 对某些类型的委托属性, Kotlin 编译器会生成辅助属性, 并将目标属性的存取操作委托给这些辅助属性.

> 为了优化的目的, 编译器 对有些情况 *不会* 生成辅助属性. 关于优化, 详情请参见 委托到另一个属性 中的示例.

比如, 对于属性 `prop`, 编译器会生成一个隐藏的 `prop$delegate` 属性, 然后属性 `prop` 的访问器代码会将存取操作委托给这个新增的属性:

```
class C {
    var prop: Type by MyDelegate()
}

// 编译器实际生成的代码如下:
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop,
value)
}
```

Kotlin 编译器通过参数来提供关于 `prop` 属性的所有必须信息: 第一个参数 `this` 指向外层类 `C` 的实例, 第二个参数 `this::prop` 是一个反射对象, 类型为 `KProperty`, 它将描述 `prop` 属性本身.

对委托属性优化的场景

如果委托属性是以下几种情况, 域成员 `$delegate` 会被省略:

- 属性的引用:

  ```
  class C<Type> {
      private var impl: Type = ...
      var prop: Type by ::impl
  }
  ```

- 命名对象

  ```
  object NamedObject {
      operator fun getValue(thisRef: Any?, property: KProperty<*>):
  String = ...
  }

  val s: String by NamedObject
  ```

- 同一模块内, 带有后端域和默认的 getter 的 final `val` 属性:

  ```
  val impl: ReadOnlyProperty<Any?, String> = ...

  class A {
      val s: String by impl
  }
  ```

- 常数表达式, 枚举值(Enum Entry), `this`, `null`. 以下是 `this` 的例子:

  ```
  class A {
      operator fun getValue(thisRef: Any?, property: KProperty<*>)
  ...

      val s by this
  }
  ```

委托到另一个属性时的翻译规则

委托到另一个属性时, Kotlin 编译器生成的代码会直接访问被参照的属性. 也就是说, 编译器不会生成域变量 `prop$delegate`. 这样的代码优化可以节约内存.

示例:

```
class C<Type> {
    private var impl: Type = ...
    var prop: Type by ::impl
}
```

`prop` 变量的属性访问器直接调用 `impl` 变量, 跳过被代理属性的 `getValue` 和 `setValue` 操作, 因此也不需要 `KProperty` 引用对象.

对于上面的代码, 编译器生成以下代码:

```
class C<Type> {
    private var impl: Type = ...

    var prop: Type
        get() = impl
        set(value) {
```

```
            impl = value
        }

    fun getProp$delegate(): Type = impl // 需要这个方法, 只是为了反射功能
}
```

控制属性委托的创建逻辑

通过定义一个 `provideDelegate` 操作符, 你可以控制属性委托对象的创建逻辑. 如果在 `by` 右侧的对象中定义了名为 `provideDelegate` 的成员函数或扩展函数, 那么这个函数将被调用, 用来创建属性委托对象的实例.

`provideDelegate` 的一种可能的使用场景, 是在属性初始化时检查属性的一致性.

比如, 如果要在(属性与其委托对象)绑定之前检查属性名称, 你可以编写这样的代码:

```
class ResourceDelegate<T> : ReadOnlyProperty<MyUI, T> {
    override fun getValue(thisRef: MyUI, property: KProperty<*>): T
{ ... }
}

class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // 创建委托
        return ResourceDelegate()
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

class MyUI {
    fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ...
}

    val image by bindResource(ResourceID.image_id)
```

```
    val text by bindResource(ResourceID.text_id)
}
```

`provideDelegate` 函数的参数与 `getValue` 相同:

- `thisRef` 参数, 类型必须与 *属性所属的类* 相同, 或者是它的基类 (对于扩展属性, 参数类型必须与被扩展的类型相同, 或者是它的基类);
- `property` 参数, 类型必须是 `KProperty<*>`, 或者是它的基类.

在 `MyUI` 的实例创建过程中, 将会对各个属性调用 `provideDelegate` 函数, 然后这个函数立即执行必要的验证.

如果不能对属性与其委托对象的绑定过程进行拦截, 要实现同样的功能, 你就必须在参数中明确地传递属性名称, 这就不太方便了:

```
// 如果没有 "provideDelegate" 功能, 我们需要这样来检查属性名称
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
    checkProperty(this, propertyName)
    // 创建委托
}
```

在编译器生成的代码中, 会调用 `provideDelegate` 方法, 用来初始化辅助属性 `prop$delegate`. 请看属性声明 `val prop: Type by MyDelegate()` 对应的生成代码, 并和上例 (没有 `provideDelegate` 方法的情况) 的代码对比以下:

```
class C {
    var prop: Type by MyDelegate()
}

// 当 'provideDelegate' 函数存在时
```

```
// 编译器生成以下代码:
class C {
    // 调用 "provideDelegate" 来创建 "delegate" 辅助属性
    private val prop$delegate = MyDelegate().provideDelegate(this,
this::prop)
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop,
value)
}
```

注意, `provideDelegate` 函数只影响辅助属性的创建, 而不会影响编译产生的属性取值方法和设值方法代码.

使用标准库中的 `PropertyDelegateProvider` 接口, 可以创建委托提供者(provider), 而不必创建新的类.

```
val provider = PropertyDelegateProvider { thisRef: Any?, property ->
    ReadOnlyProperty<Any?, Int> {_, property -> 42 }
}
val delegate: Int by provider
```

类型别名

最终更新: 2024/09/10

类型别名可以为已有的类型提供替代的名称. 如果类型名称太长, 你可以指定一个更短的名称, 然后使用新的名称.

这个功能有助于缩短那些很长的泛型类型名称. 比如, 缩短集合类型的名称通常是很吸引人的:

```
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

你也可以为函数类型指定不同的别名:

```
typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

你也可以为内部类和嵌套类指定新的名称:

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
typealias BInner = B.Inner
```

类型别名不会引入新的类型. 类型别名与它对应的真实类型完全等同. 如果你添加一个别名 `typealias Predicate<T>`, 然后在你的代码中使用 `Predicate<Int>`, Kotlin 编译器会把你的代码扩展为 `(Int) -> Boolean`. 因此, 在需要通常的函数类型的地方, 可以使用你定义的类型别名的变量, 反过来也是如此:

```
typealias Predicate<T> = (T) -> Boolean
```

```
fun foo(p: Predicate<Int>) = p(42)

fun main() {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // 打印结果为 "true"

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // 打印结果为 "[1]"
}
```

函数

最终更新: 2024/09/10

Kotlin 中的函数使用 `fun` 关键字定义:

```
fun double(x: Int): Int {
    return 2 * x
}
```

函数使用

函数的调用使用标准方式:

```
val result = double(2)
```

调用类的成员函数时, 使用点号标记法(dot notation):

```
Stream().read() // 创建一个 Stream 类的实例, 然后调用这个实例的 read() 函数
```

参数

函数参数的定义使用 Pascal 标记法 - *name*: *type* 的格式, 多个参数之间使用逗号分隔. 每个参数都必须明确指定类型:

```
fun powerOf(number: Int, exponent: Int): Int { /*...*/ }
```

声明函数参数时, 可以使用 尾随逗号(trailing comma) ("尾随逗号(Trailing Comma)" in "编码规约"):

```
fun powerOf(
    number: Int,
    exponent: Int, // 尾随逗号(trailing comma)
) { /*...*/ }
```

默认参数

函数参数可以指定默认值, 如果调用函数时省略了对应的参数, 就会使用默认值. 这种功能使得我们可以减少大量的重载(overload)函数定义:

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

参数默认值的定义方法是, 在参数类型之后, 添加 `=` 和默认值.

子类中覆盖的方法, 总是使用基类方法的默认参数值. 如果要覆盖一个有默认参数值的方法, 那么必须在方法签名中省略默认参数值:

```
open class A {
    open fun foo(i: Int = 10) { /*...*/ }
}

class B : A() {
    override fun foo(i: Int) { /*...*/ }  // 这里不允许指定默认参数值.
}
```

如果有默认值的参数 A 定义在无默认值的参数 B 之前, 那么调用函数时, 必须通过 命名参数 的方式为参数 B 指定值, 这时才能对参数 A 使用默认值:

```
fun foo(
    bar: Int = 0,
    baz: Int,
) { /*...*/ }

foo(baz = 1) // 这里将会使用默认参数 bar = 0
```

如果默认参数之后的最后一个参数是 lambda 表达式 ("Lambda 表达式的语法" in "高阶函数与 Lambda 表达式"), 那么你可以使用命名参数的方式传递这个 lambda 表达式, 也可以在括号之外传递 ("函数调用时使用尾缀 Lambda 表达式" in "高阶函数与 Lambda 表达式"):

```
fun foo(
    bar: Int = 0,
    baz: Int = 1,
```

```
    qux: () -> Unit,
) { /*...*/ }

foo(1) { println("hello") }     // 这里将会使用默认参数 baz = 1
foo(qux = { println("hello") }) // 这里将会使用默认参数 bar = 0 和 baz = 1
foo { println("hello") }        // 这里将会使用默认参数 bar = 0 和 baz = 1
```

命名参数

调用函数时, 你可以指定一个或多个参数名. 当函数参数很多时, 将实际参数值与函数参数一一对应起来会变得很困难, 尤其是如果参数值是布尔值, 或 `null` 值, 这种情况下, 指定参数名是一种非常便利的功能.

如果你在函数调用时使用命名参数, 那么可以任意改变参数的排列顺序, 如果想要使用参数的默认值, 那么只需要省略这部分参数即可.

比如, `reformat()` 函数有 4 个指定了默认值的参数.

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ',
) { /*...*/ }
```

调用这个函数时, 不需要对所有的参数进行命名:

```
reformat(
    "String!",
    false,
    upperCaseFirstLetter = false,
    divideByCamelHumps = true,
    '_'
)
```

可以省略所有那些带有默认值的参数:

```
reformat("This is a long String!")
```

也可以只省略带默认值的参数中指定的一部分, 而不是忽略全部. 但是, 在第一个省略的参数之后, 必须对后续的所有参数指定命名:

```
reformat("This is a short String!", upperCaseFirstLetter = false,
wordSeparator = '_')
```

还可以通过 `展开(spread)` 操作符, 以命名参数的方式传递 不定数量参数 (`vararg`):

```
fun foo(vararg strings: String) { /*...*/ }

foo(strings = *arrayOf("a", "b", "c"))
```

在 JVM 平台调用 Java 函数时, 不能使用这种命名参数语法, 因为 Java 字节码并不一定保留了函数参数的名称信息.

返回值为 Unit 的函数

如果一个函数不返回有意义的结果值, 那么它的返回类型为 `Unit`. `Unit` 类型只有唯一的一个值 - `Unit`. 在函数中, 不需要明确地返回这个值:

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // 这里可以写 `return Unit` 或者 `return`, 都是可选的
}
```

返回类型 `Unit` 的声明本身也是可选的. 上例中的代码等价于:

```
fun printHello(name: String?) { ... }
```

单表达式函数(Single-expression function)

如果函数体只包含单个表达式, 那么大括号可以省略, 函数体可以直接写在 `=` 之后:

```
fun double(x: Int): Int = x * 2
```

如果编译器可以推断出函数的返回值类型, 那么返回值的类型定义是 可选的:

```
fun double(x: Int) = x * 2
```

明确指定返回值类型

如果函数体为多行语句组成的代码段, 那么就必须明确指定返回值类型, 除非这个函数打算返回 `Unit`, 这时返回类型的声明可以省略.

对于多行语句组成的函数, Kotlin 不会推断其返回值类型, 因为这样的函数内部可能存在复杂的控制流, 而且返回值类型对于代码的阅读者来说并不是那么一目了然(有些时候, 甚至对于编译器来说也很难判定返回值类型).

不定数量参数(varargs)

你可以对一个函数的一个参数 (通常是参数中的最后一个) 标记 `vararg` 修饰符:

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts 是一个 Array
        result.add(t)
    return result
}
```

这种情况下, 调用时可以向这个函数传递不定数量的参数:

```
val list = asList(1, 2, 3)
```

在函数内部, 类型为 `T` 的 `vararg` 参数会被看作一个 `T` 类型的数组, 上例中的 `ts` 变量的类型为 `Array<out T>`.

只有一个参数可以标记为 `vararg`. 如果 `vararg` 参数不是函数的最后一个参数, 那么对于 `vararg` 参数之后的其他参数, 可以使用命名参数语法来传递参数值, 或者, 如果参数类型是函数, 可以在括号之外传递一个 Lambda 表达式.

调用一个存在 `vararg` 参数的函数时, 你可以传递单独的参数值, 比如, `asList(1, 2, 3)`. 如果你已经有了一个数组, 希望将它的内容传递给函数, 你可以使用 *展开(spread)* 操作符(在数组之前加一个 `*`):

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

如果要向 vararg 参数传递一个 基本类型的数组 ("基本类型(Primitive Type)数组" in "数组"), 你需要使用 toTypedArray() 函数, 将它转换为一个通常的(有类型的)数组:

```
val a = intArrayOf(1, 2, 3) // IntArray 是基本类型的数组
val list = asList(-1, 0, *a.toTypedArray(), 4)
```

中缀标记法(Infix notation)

使用 infix 关键字标记的函数, 也可以使用中缀标记法(infix notation)来调用(调用时省略点号和括号). 中缀函数需要满足以下条件:

- 必须是成员函数, 或者是扩展函数 (扩展).
- 必须只有单个参数.
- 参数不能是 不定数量参数, 而且不能有 默认值.

```
infix fun Int.shl(x: Int): Int { ... }

// 使用中缀标记法调用函数
1 shl 2

// 上面的语句等价于
1.shl(2)
```

中缀函数调用的优先级, 低于算数运算符, 类型转换, 以及 rangeTo 运算符. 以下表达式是等价的:

- 1 shl 2 + 3 等价于 1 shl (2 + 3)
- 0 until n * 2 等价于 0 until (n * 2)
- xs union ys as Set<*> 等价于 xs union (ys as Set<*>)

另一方面, 中缀函数调用的优先级, 高于布尔值运算符 `&&` 和 `||`, `is` 和 `in` 检查, 以及其他运算符. 以下表达式是等价的:

- `a && b xor c` 等价于 `a && (b xor c)`
- `a xor b in c` 等价于 `(a xor b) in c`

注意, 中缀函数的接受者和参数都需要明确指定. 如果使用中缀标记法调用当前接受者的一个方法, 需要明确指定 `this`. 这是为了保证语法解析不出现歧义.

```
class MyStringCollection {
    infix fun add(s: String) { /*...*/ }

    fun build() {
        this add "abc"   // 正确用法
        add("abc")       // 正确用法
        //add "abc"      // 错误用法: 方法的接受者必须明确指定
    }
}
```

函数的范围

Kotlin 的函数可以定义在源代码的顶级范围内(Top Level), 这就意味着, 你不必象在 Java, C# 或 Scala (从 Scala 3 开始可以使用顶级定义 (https://docs.scala-lang.org/scala3/book/taste-toplevel-definitions.html#inner-main)) 等等语言中那样, 创建一个类来容纳这个函数. 除顶级函数之外, Kotlin 的函数也可以在局部范围内, 定义为成员函数, 以及扩展函数.

局部函数

Kotlin 支持局部函数, 也就是嵌套在另一个函数内的函数:

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }
```

```
    dfs(graph.vertices[0], HashSet())
}
```

局部函数可以访问外部函数中的局部变量(闭包). 在上面的例子中, `visited` 可以定义为一个局部变量:

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

成员函数

成员函数是指定义在类或对象之内的函数:

```
class Sample {
    fun foo() { print("Foo") }
}
```

对成员函数的调用使用点号标记法:

```
Sample().foo() // 创建 Sample 类的实例, 并调用 foo 函数
```

关于类, 以及成员覆盖, 详情请参见 类 (类) 和 继承 ("继承" in "类").

泛型函数

函数可以带有泛型参数, 泛型参数通过函数名之前的尖括号来指定:

```
fun <T> singletonList(item: T): List<T> { /*...*/ }
```

关于泛型函数, 详情请参见 泛型 (泛型(Generic): in, out, where).

尾递归函数(Tail recursive function)

Kotlin 支持一种称为 尾递归(tail recursion) (https://en.wikipedia.org/wiki/Tail_call) 的函数式编程方式. 对于某些算法, 本来需要使用循环来实现, 你可以改用递归函数, 但同时不会存在栈溢出(stack overflow)的风险. 当一个函数标记为 `tailrec`, 并且满足某些形式上的要求, 编译器就会对代码进行优化, 消除函数的递归调用, 产生一段基于循环实现的, 快速而且高效的代码:

```
val eps = 1E-10 // 这个精度已经"足够"了, 也可以设置为更高精度: 10^-15

tailrec fun findFixPoint(x: Double = 1.0): Double =
    if (Math.abs(x - Math.cos(x)) < eps) x else
findFixPoint(Math.cos(x))
```

上面的代码计算余弦函数的 `不动点(fixpoint)`, 结果应该是一个数学上的常数. 这个函数只是简单地从 `1.0` 开始不断重复地调用 `Math.cos` 函数, 直到计算结果不再变化为止, 对于示例中给定的 `eps` 精度值, 计算结果将是 `0.7390851332151611`. 编译器优化产生的代码等价于下面这种传统方式编写的代码:

```
val eps = 1E-10 // 这个精度已经"足够"了, 也可以设置为更高精度: 10^-15

private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (Math.abs(x - y) < eps) return x
        x = Math.cos(x)
    }
}
```

要符合 `tailrec` 修饰符的要求, 函数必须在它执行的所有操作的最后一步, 递归调用它自身. 如果在这个递归调用之后还存在其他代码, 那么你不能使用尾递归, 也不能用在 `try`/`catch`/`finally` 结构内, 而且不能用于 open 的函数. 目前只有 Kotlin for JVM 和 Kotlin/Native 支持尾递归.

参见:

- 内联函数(Inline Function) (内联函数(Inline Function))
- 扩展函数 (扩展)

- 高阶函数(Higher-Order Function) 与 Lambda 表达式 (高阶函数与 Lambda 表达式)

高阶函数与 Lambda 表达式

最终更新: 2024/09/10

在 Kotlin 中函数是 一级公民 (https://en.wikipedia.org/wiki/First-class_function), 也就是说, 函数可以保存在变量和数据结构中, 也可以作为参数来传递给 高阶函数, 也可以作为 高阶函数 的返回值. 你可以就象对函数之外的其他数据类型值一样, 对函数执行任意的操作.

为了实现这些功能, Kotlin 作为一种静态类型语言, 使用了一组 函数类型 来表达函数, 并提供了一组专门的语言结构, 比如 lambda 表达式.

高阶函数(Higher-Order Function)

高阶函数(higher-order function)是一种特殊的函数, 它接受函数作为参数, 或者返回一个函数.

高阶函数的一个很好的例子就是 函数式编程(functional programming) 中对集合的 `折叠(fold)` (https://en.wikipedia.org/wiki/Fold_(higher-order_function)), 这个折叠函数的参数是一个初始的累计值, 以及一个结合函数, 然后将累计值与集合中的各个元素逐个结合, 最终得到结果值:

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

上面的示例代码中, `combine` 参数是 函数类型 `(R, T) -> R`, 所以这个参数接受一个函数, 函数又接受两个参数, 类型为 `R` 和 `T`, 返回值类型为 `R`. 这个函数在 `for` 循环内被 调用, 函数的返回值被赋值给 `accumulator`.

要调用上面的 `fold` 函数, 你需要向它传递一个 函数类型的实例 作为参数, 在调用高阶函数时, 我们经常使用 Lambda 表达式作为这种参数 (详细介绍请参见 后面的章节):

```
fun main() {
    //sampleStart
    val items = listOf(1, 2, 3, 4, 5)
```

```
    // Lambda 表达式是大括号括起的那部分代码.
    items.fold(0, {
        // 如果 Lambda 表达式有参数, 首先声明这些参数, 后面是 '->' 符
        acc: Int, i: Int ->
        print("acc = $acc, i = $i, ")
        val result = acc + i
        println("result = $result")
        // Lambda 表达式内的最后一个表达式会被看作返回值:
        result
    })

    // Lambda 表达式的参数类型如果可以推断得到, 那么参数类型的声明可以省略:
    val joinedToString = items.fold("Elements:", { acc, i -> acc + "
" + i })

    // 在高阶函数调用中也可以使用函数引用:
    val product = items.fold(1, Int::times)
    //sampleEnd
    println("joinedToString = $joinedToString")
    println("product = $product")
}
```

函数类型(Function Type)

为了在类型和参数声明中处理函数, 比如: `val onClick: () -> Unit = ...`, Kotlin 使用函数类型(Function Type), 比如 `(Int) -> String`.

这种函数类型使用一种特殊的表示方法, 用于表示函数的签名部分 - 也就是表示函数的参数和返回值:

- 所有的函数类型都带有参数类型列表, 用括号括起, 以及返回值类型: `(A, B) -> C` 表示一个函数类型, 它接受两个参数, 类型为 `A` 和 `B`, 返回值类型为 `C`. 参数类型列表可以为空, 比如 `() -> A`. `Unit` 类型的返回值 ("返回值为 Unit 的函数" in "函数") 不能省略.
- 函数类型也可以带一个额外的 *接受者* 类型, 以点号标记, 放在函数类型声明的前部: `A.(B) -> C` 表示一个可以对类型为 `A` 的接受者调用的函数, 参数类型为`B`, 返回值类型为 `C`. 对这种函数类型, 我们经常使用 带接受者的函数字面值.

- 挂起函数(Suspending function) ("代码重构, 抽取函数" in "协程的基本概念") 是一种特殊类型的函数, 它的声明带有一个特殊的 *suspend* 修饰符, 比如: `suspend () -> Unit`, 或者: `suspend A.(B) -> C`.

函数类型的声明也可以指定函数参数的名称: `(x: Int, y: Int) -> Point`. 参数名称可以用来更好地说明参数含义.

为了表示函数类型是 可以为 null 的 ("可为 null 的类型与不可为 null 的类型" in "Null 值安全性"), 可以使用括号: `((Int, Int) -> Int)?`.

函数类型也可以使用括号组合在一起: `(Int) -> ((Int) -> Unit)`

> 箭头符号的结合顺序是右侧优先, `(Int) -> (Int) -> Unit` 的含义与上面的例子一样, 而不同于: `((Int) -> (Int)) -> Unit`.

你也可以使用 类型别名 (类型别名) 来给函数类型指定一个名称:

```
typealias ClickHandler = (Button, ClickEvent) -> Unit
```

创建函数类型的实例

有几种不同的方法可以创建函数类型的实例:

- 使用函数字面值, 采用以下形式之一:
  - Lambda 表达式: `{ a, b -> a + b }`,
  - 匿名函数(Anonymous Function): `fun(s: String): Int { return s.toIntOrNull() ?: 0 }`

  带接受者的函数字面值 可以用作带接受者的函数类型的实例.

- 使用已声明的元素的可调用的引用:
  - 顶级函数 ("函数引用(Function Reference)" in "反射"), 局部函数 ("函数引用(Function Reference)" in "反射"), 成员函数 ("函数引用(Function Reference)" in "反射"), 或扩展函数 ("函数引用(Function Reference)" in "反射"), 比如: `::isOdd`, `String::toInt`,
  - 顶级属性 ("属性引用(Property Reference)" in "反射"), 成员属性 ("属性引用(Property Reference)" in "反射"), 或扩展属性 ("属性引用(Property Reference)" in "反射"), 比如: `List<Int>::size`,

- 构造器 ("构造器引用(Constructor Reference)" in "反射"), 比如: `::Regex`

  以上几种形式都包括 绑定到实例的可调用的引用 ("与对象实例绑定的函数和属性引用" in "反射"), 也就是指向具体实例的成员的引用: `foo::toString`.

- 使用自定义类, 以接口的方式实现函数类型:

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

如果有足够的信息, 编译器可以推断出变量的函数类型:

```
val a = { i: Int -> i + 1 } // 编译器自动推断得到的类型为 (Int) -> Int
```

带接受者和不带接受者的函数类型的 *非字面* 值是可以互换的, 也就是说, 接受者可以代替第一个参数, 反过来第一个参数也可以代替接受者. 比如, 如果参数类型或变量类型为 `A.(B) -> C`, 那么可以使用 `(A, B) -> C` 函数类型的值, 反过来也是如此:

```
fun main() {
    //sampleStart
    val repeatFun: String.(Int) -> String = { times ->
this.repeat(times) }
    val twoParameters: (String, Int) -> String = repeatFun // OK

    fun runTransformation(f: (String, Int) -> String): String {
        return f("hello", 3)
    }
    val result = runTransformation(repeatFun) // OK
    //sampleEnd
    println("result = $result")
}
```

注意, 自动推断的结果默认是不带接受者的函数类型, 即使给变量初始化赋值为一个扩展函数的引用, 也是如此. 要改变这种结果, 你需要明确指定变量类型.

调用一个函数类型的实例

要调用一个函数类型的值, 可以使用它的 `invoke(...)` 操作符 ("函数调用操作符" in "操作符重载"): `f.invoke(x)`, 或者直接写 `f(x)`.

如果函数类型值有接受者, 那么接受者对象实例应该作为第一个参数传递进去. 调用有接受者的函数类型值的另一种方式是, 将接受者写作函数调用的前缀, 就像调用 扩展函数 (扩展) 一样: `1.foo(2)`.

示例:

```
fun main() {
    //sampleStart
    val stringPlus: (String, String) -> String = String::plus
    val intPlus: Int.(Int) -> Int = Int::plus

    println(stringPlus.invoke("<-", "->"))
    println(stringPlus("Hello, ", "world!"))

    println(intPlus.invoke(1, 1))
    println(intPlus(1, 2))
    println(2.intPlus(3)) // 与扩展函数类似的调用方式
    //sampleEnd
}
```

内联函数(Inline Function)

有些时候, 使用 内联函数 (内联函数(Inline Function)) 可以为高阶函数实现更加灵活的控制流程.

Lambda 表达式与匿名函数(Anonymous Function)

Lambda 表达式和匿名函数, 都是 *函数字面值(function literal)*, 函数字面值没有象普通函数那样声明, 而是立即作为表达式传递出去. 看看下面的示例:

```
max(strings, { a, b -> a.length < b.length })
```

函数 `max` 是一个高阶函数, 因为它接受一个函数值作为第二个参数. 第二个参数是一个表达式, 本身又是另一个函数, 称为函数字面值. 这个函数字面值等价于下面这个有名称的函数:

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Lambda 表达式的语法

Lambda 表达式的完整语法形式如下:

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

- Lambda 表达式包含在大括号之内.
- 在完整语法形式中, 参数声明在大括号之内, 参数类型的声明是可选的.
- 函数体在 `->` 符号之后.
- 如果 Lambda 表达式自动推断的返回值类型不是 `Unit`, 那么 Lambda 表达式函数体中, 最后一条(或者就是唯一一条)表达式的值, 会被当作整个 Lambda 表达式的返回值.

如果把所有可选的内容都去掉, 那么剩余的部分如下:

```
val sum = { x: Int, y: Int -> x + y }
```

函数调用时使用尾缀 Lambda 表达式

根据 Kotlin 的编码规约, 如果函数的最后一个参数是一个函数, 那么如果使用 Lambda 表达式作为这个参数的值, 可以将 Lambda 表达式写在函数调用的括号之外:

```
val product = items.fold(1) { acc, e -> acc * e }
```

这种语法又称为 *尾缀 Lambda 表达式(Trailing Lambda)*.

如果 Lambda 表达式是函数调用时的唯一一个参数, 括号可以完全省略:

```
run { println("...") }
```

it: 单一参数的隐含名称

很多情况下 Lambda 表达式只有唯一一个参数.

如果编译器能够识别出 Lambda 表达式没有参数定义, 那么可以不必声明参数, 并省略 `->` 符号. 这个参数会隐含地声明, 参数名为 `it`:

```
ints.filter { it > 0 } // 这个函数字面值的类型是 '(it: Int) -> Boolean'
```

从 Lambda 表达式中返回结果值

如果使用 带标签限定的 return ("使用标签控制 return 的目标" in "返回与跳转: break 与 continue") 语法, 你可以在 Lambda 表达式内明确地返回一个结果值. 否则, 会隐含地返回 Lambda 表达式内最后一条表达式的值.

因此, 下面两段代码是等价的:

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

使用这个规约, 再加上 在括号之外传递 Lambda 表达式作为函数调用的参数, 我们可以编写 LINQ 风格 (https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/) 的程序:

```
strings.filter { it.length == 5 }.sortedBy { it }.map {
it.uppercase() }
```

使用下划线代替未使用的参数

如果 Lambda 表达式的某个参数未被使用, 你可以用下划线来代替参数名:

```
map.forEach { (_, value) -> println("$value!") }
```

在 Lambda 表达式中使用解构声明

关于在 Lambda 表达式中使用解构声明, 请参见 解构声明(destructuring declaration) ("在 Lambda 表达式中使用解构声明" in "解构声明").

匿名函数(Anonymous Function)

上面讲到的 Lambda 表达式语法, 还缺少了一种功能, 就是如何指定函数的返回值类型. 大多数情况下, 不需要指定返回值类型, 因为可以自动推断得到. 但是, 如果的确需要明确指定返回值类型, 你可

以可以选择另一种语法: *匿名函数(anonymous function)*.

```
fun(x: Int, y: Int): Int = x + y
```

匿名函数看起来与通常的函数声明很类似, 区别在于省略了函数名. 函数体可以是一个表达式(如上例), 也可以是多条语句组成的代码段:

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

参数和返回值类型的声明与通常的函数一样, 但如果参数类型可以通过上下文推断得到, 那么类型声明可以省略:

```
ints.filter(fun(item) = item > 0)
```

对于匿名函数, 返回值类型的自动推断方式与通常的函数一样: 如果函数体是一个表达式, 那么返回值类型可以自动推断得到, 但如果函数体是多条语句组成的代码段, 则返回值类型必须明确指定(否则被认为是 `Unit`).

> 匿名函数当作参数传递时, 一定要放在函数调用的圆括号内. 允许将函数类型参数写在圆括号之外的语法, 仅对 Lambda 表达式有效.

Lambda 表达式与匿名函数之间的另一个区别是, 它们的 非局部返回(non-local return) ("非局部返回(Non-local return)" in "内联函数(Inline Function)") 的行为不同. 不使用标签的 `return` 语句总是从 `fun` 关键字定义的函数中返回. 也就是说, Lambda 表达式内的 `return` 将会从包含这个 Lambda 表达式的函数中返回, 而匿名函数内的 `return` 只会从匿名函数本身返回.

闭包(Closure)

Lambda 表达式, 匿名函数 (此外还有 局部函数 ("局部函数" in "函数"), 对象表达式 ("对象表达式(Object expression)" in "对象表达式,对象声明,以及同伴对象")) 可以访问它的 *闭包*, 也就是, 定义在外层范围中的变量. 闭包中捕获的变量在 Lambda 表达式内是可以修改的:

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
```

```
}
print(sum)
```

带有接受者的函数字面值

带接受者的 函数类型, 比如 `A.(B) -> C`, 可以通过一种特殊形式的函数字面值来创建它的实例, 也就是带接受者的函数字面值.

上文讲到, Kotlin 提供了一种能力, 可以指定一个 *接收者对象(receiver object)*, 来 调用带接受者的函数类型的实例.

在这个函数字面值的函数体内部, 传递给这个函数调用的接受者对象会成为一个 *隐含的* `this`, 因此你可以访问接收者对象的成员, 而不必指定任何限定符, 也可以使用 `this` 表达式 (this 表达式) 来访问接受者对象.

这种行为很类似于 扩展函数 (扩展), 在扩展函数的函数体中, 你也可以访问接收者对象的成员.

下面的例子演示一个带接受者的函数字面值, 以及这个函数字面值的类型, 在函数体内部, 调用了接受者对象的 `plus` 方法:

```
val sum: Int.(Int) -> Int = { other -> plus(other) }
```

匿名函数语法允许你直接指定函数字面值的接受者类型. 如果你需要声明一个带接受者的函数类型变量, 然后在将来的某个地方使用它, 那么这种功能就很有用.

```
val sum = fun Int.(other: Int): Int = this + other
```

如果接受者类型可以通过上下文自动推断得到, 那么 Lambda 表达式也可以用做带接受者的函数字面值. 这种用法的一个重要例子就是 类型安全的构建器(Type-Safe Builder) (类型安全的构建器):

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()  // 创建接受者对象
    html.init()        // 将接受者对象传递给 Lambda 表达式
    return html
}

html {       // 带接受者的 Lambda 表达式从这里开始
```

```
    body()    // 调用接受者对象上的一个方法
}
```

内联函数(Inline Function)

最终更新: 2024/09/10

使用 高阶函数 (高阶函数与 Lambda 表达式) 在运行时会带来一些不利: 每个函数都是一个对象, 而且它还要捕获一个闭包, 闭包是指一个环境范围, 在这个范围内, 函数体内部可以访问外层变量. 内存占用(函数对象和类都会占用内存) 以及虚方法调用都会带来运行时的消耗.

但在很多情况下, 通过将 Lambda 表达式内联在使用处, 可以消除这些运行时消耗. 下文中的函数就是很好的例子. `lock()` 函数可以很容易地内联在调用处. 看看下面的例子:

```
lock(l) { foo() }
```

编译器可以直接产生下面的代码, 而不必为参数创建函数对象, 然后再调用这个参数指向的函数:

```
l.lock()
try {
    foo()
} finally {
    l.unlock()
}
```

为了让编译器做到这点, 需要对 `lock()` 函数标记 `inline` 修饰符:

```
inline fun <T> lock(lock: Lock, body: () -> T): T { ... }
```

`inline` 修饰符既会影响到函数本身, 也影响到传递给它的 Lambda 表达式: 这两者都会被内联到调用处.

函数内联也许会导致编译产生的代码尺寸变大. 但只要你合理的使用(不要内联太大的函数), 就可以换来性能的提高, 尤其是在循环内发生的 "megamorphic" 函数调用. (译注: 关于 megamorphic 请参见 Inline caching (https://en.wikipedia.org/wiki/Inline_caching#Megamorphic_inline_caching))

noinline

如果一个内联函数的参数中有多个 Lambda 表达式, 而你只希望内联其中的一部分, 可以对函数的一部分参数添加 `noinline` 修饰符:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit)
{ ... }
```

可内联的 Lambda 表达式只能在内联函数内部调用, 或者再作为可内联的参数传递给其他函数, 而 noinline 的 Lambda 表达式可以按照你喜欢的方式任意使用: 可以保存在域内, 也可以当作参数传递, 等等.

> 如果一个内联函数不存在可以内联的函数类型参数, 而且没有 实体化的类型参数, 编译器将会产生一个警告, 因为将这样的函数内联不太可能带来任何益处. (如果你确信需要内联, 可以使用 @Suppress("NOTHING_TO_INLINE") 注解关闭这个警告)

非局部返回(Non-local return)

在 Kotlin 中, 使用无限定符的通常的 return 语句, 只能用来退出一个有名称的函数, 或匿名函数. 要退出一个 Lambda 表达式, 可以使用一个 标签 ("使用标签控制 return 的目标" in "返回与跳转: break 与 continue"). 在 Lambda 表达式内禁止使用无标签的 return, 因为 Lambda 表达式不允许强制包含它的函数 return:

```
fun ordinaryFunction(block: () -> Unit) {
    println("hi!")
}
//sampleStart
fun foo() {
    ordinaryFunction {
        return // 错误: 这里不允许让 `foo` 函数返回
    }
}
//sampleEnd
fun main() {
    foo()
}
```

但是, 如果 Lambda 表达式被传递去的函数是内联函数, 那么 return 语句也可以内联, 因此 return 是允许的:

```
inline fun inlined(block: () -> Unit) {
    println("hi!")
```

```
}
//sampleStart
fun foo() {
    inlined {
        return // OK: 这里的 Lambda 表达式是内联的
    }
}
//sampleEnd
fun main() {
    foo()
}
```

这样的 `return` 语句(位于 Lambda 表达式内部, 但是退出包含 Lambda 表达式的函数) 称为 *非局部 (non-local)* 返回. 这样的结构经常出现在循环中, 而循环也常常就是包含内联函数的地方:

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // 从 hasZeros 函数返回
    }
    return false
}
```

注意, 有些内联函数可能并不在自己的函数体内直接调用传递给它的 Lambda 表达式参数, 而是通过另一个执行环境来调用, 比如通过一个局部对象, 或者一个嵌套函数. 这种情况下, 在 Lambda 表达式内, 非局部的控制流同样是禁止的. 为了标识内联函数的 Lambda 表达式参数不能使用非局部(non-local)返回, 需要对 Lambda 表达式参数添加 `crossinline` 修饰符:

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

> 在内联的 Lambda 表达式中目前还不能使用 `break` 和 `continue`, 但我们计划将来支持它们.

实体化的类型参数(Reified type parameter)

有些时候你需要访问作为参数传递来的类型:

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

这里, 你向上遍历一颗树, 然后使用反射来检查节点是不是某个特定的类型. 这些都没问题, 但这个函数的调用代码不太漂亮:

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

更好的解决方案是简单地将一个类型传递给这个函数, 可以象这样调用它:

```
treeNode.findParentOfType<MyTreeNode>()
```

为了达到这个目的, 内联函数支持 *实体化的类型参数(reified type parameter)*, 使用这个功能你可以将代码写成:

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

上面的代码给类型参数添加了 `reified` 修饰符, 使得它可以在函数内部访问, 就好象它是一个普通的类一样. 由于函数是内联的, 因此不必使用反射, 而且通常的操作符都可以使用, 比如 `!is` 和 `as`. 此外, 你可以通过上面提到那种方式来调用这个函数: `myTree.findParentOfType<MyTreeNodeType>()`.

虽然很多情况下并不需要, 但你仍然可以对一个实体化的类型参数使用反射:

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

通常的函数(没有使用 inline 标记的) 不能够使用实体化的类型参数. 一个没有运行时表现的类型(比如, 一个没有实体化的类型参数, 或者一个虚拟类型, 比如 `Nothing`) 不可以用作实体化的类型参数.

内联属性(Inline property)

对于不存在 后端域变量(Backing Field) ("属性的后端域变量(Backing Field)" in "属性(Property)") 的属性, 可以对它的取值和设值方法使用 `inline` 修饰符. 你可以标识单个的属性取值/设值方法:

```
val foo: Foo
    inline get() = Foo()

var bar: Bar
    get() = ...
    inline set(v) { ... }
```

也可以标注整个属性, 等于将它的取值和设值方法都标注为 `inline`:

```
inline var bar: Bar
    get() = ...
    set(v) { ... }
```

属性取值/设值方法被标注为 `inline` 后, 会被内联到调用处, 就像通常的内联函数一样.

对 Public API 内联函数的限制

当一个内联函数是 `public` 或 `protected` 的, 但不属于 `private` 或 `internal` 类型的一部分, 这个函数将被认为是一个 模块(module) ("模块(Module)" in "可见度修饰符") 的 Public API. 它可以在其它模块中调用, 并且被内联到调用处.

假如内联函数的定义模块发生了变化, 而调用它的模块没有重新编译, 这时就可能会造成二进制代码不兼容的风险.

为了解决由模块中的 *非*-public API 变更带来的不兼容性, Public API 内联函数的函数体部分, 不允许使用 非-Public-API, 也就是, 定义为 `private` 和 `internal` 的部分.

定义为 `internal` 的元素也可以使用 `@PublishedApi` 注解, 这就允许它被 Public API 内联函数使用. 当 `internal` 内联函数标注为 `@PublishedApi` 时, 也会象 Public API 内联函数一样检查它的函数体.

操作符重载

最终更新: 2024/09/10

Kotlin 允许你对数据类型的一组预定义的操作符提供自定义的实现函数. 这些操作符有预定义的表达符号(比如 `+` 或 `*`), 以及预定义的优先顺序. 要实现这些操作符, 需要对相应的数据类型实现一个特定名称的 成员函数 ("成员函数" in "函数") 或 扩展函数 (扩展), 这里的数据类型, 对于二元操作符, 是指左侧操作数的类型, 对于一元操作符, 是指唯一一个操作数的类型.

要重载操作符, 要对相应的函数使用 `operator` 修饰符.

```
interface IndexedContainer {
    operator fun get(index: Int)
}
```

如果在后代类中 重载 ("方法的覆盖" in "继承") 操作符, 可以省略 `operator`:

```
class OrdersList: IndexedContainer {
    override fun get(index: Int) { /*...*/ }
}
```

一元操作符

一元前缀操作符

表达式	翻译为
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

上表告诉我们说, 当编译器处理一元操作符时, 比如表达式 `+a`, 它将执行以下步骤:

- 确定 `a` 的类型, 假设为 `T`.

- 查找带有 `operator` 修饰符, 无参数的 `unaryPlus()` 函数, 而且函数的接受者类型为 `T`, 也就是说, `T` 类型的成员函数或扩展函数.
- 如果这个函数不存在, 或者找到多个, 则认为是编译错误.
- 如果这个函数存在, 并且返回值类型为 `R`, 则表达式 `+a` 的类型为 `R`.

> 这些操作符, 以其其它所有操作符, 都对 基本类型 (基本类型) 进行了优化 因此不会发生函数调用, 并由此产生性能损耗.

举例来说, 我们可以这样来重载负号操作符:

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
   println(-point)  // 打印结果为 "Point(x=-10, y=-20)"
}
```

递增与递减操作符

表达式	翻译为
`a++`	`a.inc()` (参见下文)
`a--`	`a.dec()` (参见下文)

`inc()` 和 `dec()` 函数必须返回一个值, 这个返回值将会赋值给使用 `++` 或 `--` 操作符的对象变量. 这两个函数不应该改变调用 `inc` 或 `dec` 函数的对象的内容.

对于 *后缀* 形式操作符, 比如 `a++`, 编译器解析时将执行以下步骤:

- 确定 `a` 的类型, 假设为 `T`.
- 查找带有 `operator` 修饰符, 无参数的 `inc()` 函数, 而且函数的接受者类型为 `T`.

- 检查函数的返回值类型是不是`T`的子类型.

计算这个表达式所造成的影响是:

- 将 `a` 的初始值保存到临时变量 `a0` 中.
- 将 `a0.inc()` 的结果赋值给 `a`.
- 返回 `a0`, 作为表达式的计算结果值.

对于 `a--`, 计算步骤完全类似.
对于 *前缀* 形式的操作符 `++a` 和 `--a`, 解析过程是一样的, 计算表达式所造成的影响是:

- 将 `a.inc()` 的结果赋值给 `a`.
- 返回 `a` 的新值, 作为表达式的计算结果值.

二元操作符

算数操作符

表达式	翻译为
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`
`a..b`	`a.rangeTo(b)`
`a..<b`	`a.rangeUntil(b)`

对于上表中的操作符, 编译器只是简单地解析 *翻译为* 列中的表达式.

下面是一个 `Counter` 类的例子, 它从一个给定的值开始计数, 可以通过 `+` 操作符递增:

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

in 操作符

表达式	翻译为
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

对于 `in` 和 `!in` 操作符, 解析过程也是一样的, 但参数顺序被反转了.

下标访问操作符

表达式	翻译为
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ..., i_n]`	`a.get(i_1, ..., i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ..., i_n] = b`	`a.set(i_1, ..., i_n, b)`

方括号被翻译为, 使用适当个数的参数, 对 `get` 和 `set` 函数的调用.

函数调用操作符

表达式	翻译为
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ..., i_n)`	`a.invoke(i_1, ..., i_n)`

圆括号被翻译为, 使用适当个数的参数, 调用 `invoke` 函数.

计算并赋值

表达式	翻译为
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)`

对于赋值操作符, 比如 `a += b`, 编译器执行以下步骤:

- 如果上表中右列的函数可用:
  - 如果对应的二元操作函数(也就是说, 对于 `plusAssign()` 来说, `plus()` 函数) 也可用, `a` 是一个值可变的变量, `plus` 的返回类型是 `a` 的类型的子类型, 报告错误(歧义).
  - 确认函数的返回值类型为 `Unit`, 否则报告错误.

    - 生成 `a.plusAssign(b)` 的代码.

- 否则, 尝试生成 `a = a + b` 的代码(这里包括类型检查: `a + b` 的类型必须是 `a` 的子类型).

> 在 Kotlin 中, 赋值操作 *不是* 表达式.

相等和不等比较操作符

表达式	翻译为
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

这些操作符是通过 `equals(other: Any?): Boolean` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/equals.html) 函数来实现的, 可以重载这个函数, 来实现自定义的相等判断. 同名但不同参数的任何其他函数 (比如 `equals(other: Foo)`) 都不会被调用.

> `===` 和 `!==` (同一性检查) 操作符不允许重载, 因此对这两个操作符不存在约定.

`==` 操作符是特殊的: 它被翻译为一个复杂的表达式, 其中包括对 `null` 值的处理. `null == null` 的判断结果永远为 `true`, 对于非 null 的 `x`, `x == null` 永远为 `false`, 并且不会调用 `x.equals()`.

比较操作符

表达式	翻译为
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

所有的比较操作符都被翻译为对 `compareTo` 函数的调用, 这个函数的返回值必须是 `Int` 类型.

属性委托操作符

关于 `provideDelegate`, `getValue` 和 `setValue` 操作符函数, 请参见 委托属性 (委托属性).

对命名函数的中缀式调用

使用 中缀式函数调用 ("中缀标记法(Infix notation)" in "函数"), 你可以模拟自定义的中缀操作符.

类型安全的构建器

最终更新: 2024/09/10

通过将恰当命名的函数用做构建器, 结合 带接受者的函数字面值 ("带有接受者的函数字面值" in "高阶函数与 Lambda 表达式"), 我们可以在 Kotlin 中创建出类型安全的, 静态类型的构建器.

类型安全的构建器(Type-safe builder) 可以用来创建基于 Kotlin 的, 特定领域专用语言(domain-specific language, DSL), 这些语言适合于使用半声明的方式创建复杂的层级式数据结构. 比如, 构建器的一些应用场景包括:

- 使用 Kotlin 代码来生成标记式语言, 比如 HTML (https://github.com/Kotlin/kotlinx.html) 或 XML
- 为 Web 服务器配置路由: Ktor (https://ktor.io/docs/routing.html)

我们来看看以下代码:

```
import com.example.html.* // 具体的声明参见下文

fun result() =
    html {
        head {
            title {+"XML encoding with Kotlin"}
        }
        body {
            h1 {+"XML encoding with Kotlin"}
            p  {+"this format can be used as an alternative markup
to XML"}

            // 一个元素, 指定了属性, 还指定了其中的文本内容
            a(href = "https://kotlinlang.org") {+"Kotlin"}

            // 混合内容
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "https://kotlinlang.org") {+"Kotlin"}
```

```
                +"project"
            }
            p {+"some text"}

            // 由程序生成的内容
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

上面是一段完全合法的 Kotlin 代码. 你可以 在这个页面中在线验证这段代码(可以在浏览器中修改并运行它) (https://play.kotlinlang.org/byExample/09_Kotlin_JS/06_HtmlBuilder).

工作原理

假设你需要用 Kotlin 来实现一个类型安全的构建器. 首先, 要对你想要构建的东西定义一组模型. 在这个示例中, 需要对 HTML 标签建模. 这个任务很简单, 只需要定义一组对象就可以了. 比如, `HTML` 是一个类, 负责描述 `<html>` 标签, 它可以定义子标签, 比如 `<head>` 和 `<body>`. (这个类的具体定义请参见下文.)

现在, 回忆一下为什么你可以写这样的代码:

```
html {
 // ...
}
```

`html` 实际上是一个函数调用, 它接受一个 Lambda 表达式 (高阶函数与 Lambda 表达式) 作为参数. 这个函数的定义如下:

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

这个函数只接受唯一一个参数, 名为 `init`, 这个参数本身又是一个函数. 其类型是 `HTML.() -> Unit`, 它是一个 *带接受者的函数类型*. 也就是说, 你应该向这个函数传递一个 `HTML` 的实例(一个 *接收者*)

作为参数, 而且在函数内, 你可以调用这个实例的成员.

接受者可以通过 `this` 关键字来访问:

```
html {
    this.head { ... }
    this.body { ... }
}
```

(`head` 和 `body` 是 `HTML` 类的成员函数.)

现在, `this` 关键字可以省略, 通常都是如此, 省略之后你的代码就已经非常接近一个构建器了:

```
html {
    head { ... }
    body { ... }
}
```

那么, 这个函数调用做了什么? 我们来看看上面定义的 `html` 函数体. 首先它创建了一个 `HTML` 类的新实例, 然后它调用通过参数得到的函数, 来初始化这个 `HTML` 实例 (在这个示例中, 这个初始化函数对 `HTML` 实例调用了 `head` 和 `body` 方法), 然后, 这个函数返回这个 `HTML` 实例. 这正是构建器应该做的.

`HTML` 类中 `head` 和 `body` 函数的定义与 `html` 函数类似. 唯一的区别是, 这些函数会将自己创建的对象实例添加到自己所属的 `HTML` 实例的 `children` 集合中:

```
fun head(init: Head.() -> Unit): Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit): Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
}
```

实际上这两个函数做的事情完全相同, 因此你可以编写一个泛型化的函数, 名为 initTag:

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

然后, 这你的函数就变得很简单了:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

现在你可以使用这两个函数来构建 <head> 和 <body> 标签了.

还需要讨论的一个问题是, 你要如何在标签内部添加文本. 在上面的示例程序中, 你写了这样的代码:

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}
```

你所作的, 仅仅只是将一个字符串放在一个标签之内, 但在字符串之前有一个小小的 +, 所以, 它是一个函数调用, 被调用的是前缀操作符函数 unaryPlus(). 这个操作符实际上是由扩展函数 unaryPlus() 定义的, 这个扩展函数是抽象类 TagWithText 的成员 (这个抽象类是 Title 类的祖先类):

```
operator fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

所以, 前缀操作符 + 所作的, 是将一个字符串封装到 TextElement 的一个实例中, 然后将这个实例添加到 children 集合中, 然后这个字符串就会成为标签树中一个适当的部分.

以上所有类和函数都定义在 com.example.html 包中, 上面的构建器示例程序的最上部引入了这个包. 在最后一节中, 你可以读到这个包的完整定义.

控制接受者的作用范围: @DslMarker

使用 DSL 时, 可能遇到的一个问题就是, 当前上下文中存在太多可供调用的函数. 在 Lambda 表达式内, 你可以调用所有隐含接受者的所有方法, 因此造成一种不正确的结果, 比如一个 `head` 之内可以嵌套另一个 `head` 标签:

```
html {
    head {
        head {} // 应该禁止这样的调用
    }
    // ...
}
```

在这个示例中, 应该只允许调用离当前代码最近的隐含接受者 `this@head` 的成员函数; `head()` 是更外层接受者 `this@html` 的成员函数,因此调用它应该是不允许的.

为了解决这个问题, 有一种特殊机制来控制接受者的作用范围.

要让编译器控制接受者的作用范围, 你只需要用一个相同的注解, 对 DSL 中用到的所有接受者的类型进行标注. 比如, 对 HTML 构建器你可以定义一个注解 `@HTMLTagMarker`:

```
@DslMarker
annotation class HtmlTagMarker
```

如果对一个注解类标注了 `@DslMarker` 注解, 我们将它称作一个 DSL 标记.

在我们的 DSL 中, 所有的标签类都继承自相同的超类 `Tag`. 只需要对超类标注 `@HtmlTagMarker` 注解就够了, 然后 Kotlin 编译器会将所有的派生类都看作已被标注了同样的注解:

```
@HtmlTagMarker
abstract class Tag(val name: String) { ... }
```

你不必对 `HTML` 或 `Head` 类再标注 `@HtmlTagMarker` 注解, 因为它们的超类已经标注过了这个注解:

```
class HTML() : Tag("html") { ... }

class Head() : Tag("head") { ... }
```

标注这个注解之后, Kotlin 编译器就可以知道哪些隐含的接受者属于相同的 DSL, 因此编译器只允许代码调用离当前位置最近的接受者的成员函数:

```
html {
    head {
        head { } // 编译错误: 这是外层接受者的成员函数, 因此不允许在这里调用
    }
    // ...
}
```

注意, 如果确实需要调用外层接受者的成员函数, 仍然是可以实现的, 但这时你必须明确指定具体的接受者:

```
html {
    head {
        this@html.head { } // 仍然可以调用外层接受者的成员函数
    }
    // ...
}
```

com.example.html 包的完整定义

下面是 `com.example.html` 包的完整定义(但只包含上文示例程序使用到的元素). 它可以构建一个 HTML 树. 这段代码大量使用了 扩展函数 (扩展) 和 带接受者的 Lambda 表达式 ("带有接受者的函数字面值" in "高阶函数与 Lambda 表达式").

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}
```

```
@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit):
T {
        tag.init()
        children.add(tag)
        return tag
    }

    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for ((attr, value) in attributes) {
            builder.append(" $attr=\"$value\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
```

```
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body : BodyTag("body")
class B : BodyTag("b")
class P : BodyTag("p")
class H1 : BodyTag("h1")

class A : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}
```

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

通过构建器类型推断(Builder Type Inference)使用构建器

最终更新: 2024/09/10

Kotlin 支持 *构建器类型推断(Builder Type Inference)* (或者叫构建器推断), 当你使用泛型构建器时, 这个功能可以很有用. 它能够帮助编译器, 通过构建器的 Lambda 表达式参数内的其它调用的类型信息, 推断出构建器调用的类型参数.

请参考下面的示例程序中对 `buildMap()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-map.html) 的使用:

```
fun addEntryToMap(baseMap: Map<String, Number>, additionalEntry:
Pair<String, Int>?) {
    val myMap = buildMap {
        putAll(baseMap)
        if (additionalEntry != null) {
            put(additionalEntry.first, additionalEntry.second)
        }
    }
}
```

这里没有足够的类型信息来通过通常的方式推断类型参数, 但构建器推断能够分析 Lambda 表达式参数内的函数调用. 根据 `putAll()` 和 `put()` 调用的类型信息, 编译器可以自动将 `buildMap()` 调用的类型参数推断为 `String` 和 `Number`. 使用泛型构建器时, 构建器推断功能允许我们省略类型参数.

编写你自己的构建器

启用构建器推断的要求条件

> 在 Kotlin 1.7.0 以前, 对一个构建器函数启用构建器推断, 需要添加编译器选项 `-Xenable-builder-inference`. 在 1.7.0 中, 这个选项会默认启用.

要对你自己的构建器使用构建器推断, 请确认它的声明有一个构建器 Lambda 表达式参数, 类型为带接受者的函数类型. 对接受者类型还有 2 个要求:

1. 它应该使用构建器推断需要推断的那个类型参数. 比如:

```
fun <V> buildList(builder: MutableList<V>.() -> Unit) { ... }
```

> 注意, 直接传递类型参数的类型, 比如 `fun <T> myBuilder(builder: T.() -> Unit)`, 目前还不支持.

2. 它应该提供 public 成员函数, 或扩展函数, 签名中包含对应的类型参数. 比如:

```
class ItemHolder<T> {
    private val items = mutableListOf<T>()

    fun addItem(x: T) {
        items.add(x)
    }

    fun getLastItem(): T? = items.lastOrNull()
}

fun <T> ItemHolder<T>.addAllItems(xs: List<T>) {
    xs.forEach { addItem(it) }
}

fun <T> itemHolderBuilder(builder: ItemHolder<T>.() -> Unit):
ItemHolder<T> =
    ItemHolder<T>().apply(builder)

fun test(s: String) {
    val itemHolder1 = itemHolderBuilder { // itemHolder1 的类型是
ItemHolder<String>
        addItem(s)
    }
    val itemHolder2 = itemHolderBuilder { // itemHolder2 的类型是
ItemHolder<String>
        addAllItems(listOf(s))
    }
    val itemHolder3 = itemHolderBuilder { // itemHolder3 的类型是
```

```
ItemHolder<String?>
        val lastItem: String? = getLastItem()
        // ...
    }
}
```

支持的功能

构建器推断支持以下功能:

- 推断多个类型参数

```
fun <K, V> myBuilder(builder: MutableMap<K, V>.() -> Unit): Map<K,
V> { ... }
```

- 推断一个调用之内, 相互依赖的多个构建器 Lambda 表达式的类型参数

```
fun <K, V> myBuilder(
    listBuilder: MutableList<V>.() -> Unit,
    mapBuilder: MutableMap<K, V>.() -> Unit
): Pair<List<V>, Map<K, V>> =
    mutableListOf<V>().apply(listBuilder) to mutableMapOf<K, V>
().apply(mapBuilder)

fun main() {
    val result = myBuilder(
        { add(1) },
        { put("key", 2) }
    )
    // result 的类型是 Pair<List<Int>, Map<String, Int>>
}
```

- 推断 Lambda 表达式的参数或返回类型中出现的类型参数

```
fun <K, V> myBuilder1(
    mapBuilder: MutableMap<K, V>.() -> K
): Map<K, V> = mutableMapOf<K, V>().apply { mapBuilder() }

fun <K, V> myBuilder2(
```

```
    mapBuilder: MutableMap<K, V>.(K) -> Unit
): Map<K, V> = mutableMapOf<K, V>().apply { mapBuilder(2 as K) }

fun main() {
    // result1 推断得到的类型是 Map<Long, String>
    val result1 = myBuilder1 {
        put(1L, "value")
        2
    }
    val result2 = myBuilder2 {
        put(1, "value 1")
        // 你可以将 `it` 用作 "推迟类型变量" 类型
        // 详情请参见以下章节
        put(it, "value 2")
    }
}
```

构建器推断的工作原理

推迟类型变量(Postponed Type Variable)

构建器推断使用 *推迟类型变量(Postponed Type Variable)*, 在构建器推断分析时, 它出现在构建器的 Lambda 表达式之内. 一个推迟类型变量的类型是类型参数中的一个, 具体类型还在推断过程中. 编译器使用它来收集类型参数的类型信息.

我们来看看下面示例中的 `buildList()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-list.html):

```
val result = buildList {
    val x = get(0)
}
```

这里 `x` 的类型是推迟类型变量: `get()` 调用返回一个类型 `E` 的值, 但 `E` 自身还未确定. 在这个时刻, 还不知道 `E` 的确定类型.

当一个推迟类型变量的值关联到一个确定的类型, 构建器推断会收集这个信息, 在构建器推断分析结束后, 推断对应的类型参数的结果类型. 比如:

```
val result = buildList {
    val x = get(0)
```

```
    val y: String = x
} // result 的类型推断为 List<String>
```

在推迟类型变量赋值给一个 `String` 类型变量之后, 构建器推断得到信息, `x` 是 `String` 的子类型. 这个赋值是构建器 Lambda 表达式内的最后一条语句, 因此构建器推断分析结束, 结果是将类型参数 `E` 推断为 `String`.

注意, 你总是可以将推迟类型变量作为接受者, 调用 `equals()`, `hashCode()`, 和 `toString()` 函数.

向构建器推断结果贡献信息

构建器推断可以收集不同种类的类型信息, 这些信息都会贡献到分析结果. 它会考虑以下信息:

- 对 Lambda 表达式的接受者, 使用类型参数的类型调用方法

```
val result = buildList {
    // 根据传递的 "value" 参数, 类型参数被推断为 String
    add("value")
} // result 的类型被推断为 List<String>
```

- 对返回类型参数类型的调用, 指定期望的类型

```
val result = buildList {
    // 根据期待的类型, 类型参数被推断为 Float
    val x: Float = get(0)
} // result 的类型被推断为 List<Float>
```

```
class Foo<T> {
    val items = mutableListOf<T>()
}

fun <K> myBuilder(builder: Foo<K>.() -> Unit): Foo<K> = Foo<K>
().apply(builder)

fun main() {
    val result = myBuilder {
        val x: List<CharSequence> = items
        // ...
```

```
    } // result 的类型被推断为 Foo<CharSequence>
}
```

- 向期待确定类型的方法传递推迟类型变量的类型

```
fun takeMyLong(x: Long) { ... }

fun String.isMoreThat3() = length > 3

fun takeListOfStrings(x: List<String>) { ... }

fun main() {
    val result1 = buildList {
        val x = get(0)
        takeMyLong(x)
    } // result1 的类型为 List<Long>

    val result2 = buildList {
        val x = get(0)
        val isLong = x.isMoreThat3()
    // ...
    } // result2 的类型为 List<String>

    val result3 = buildList {
        takeListOfStrings(this)
    } // result3 的类型为 List<String>
}
```

- 取得一个指向 Lambda 表达式接受者的成员的可调用的引用

```
fun main() {
    val result = buildList {
        val x: KFunction1<Int, Float> = ::get
    } // result 的类型为 List<Float>
}
```

```
fun takeFunction(x: KFunction1<Int, Float>) { ... }
```

```
fun main() {
    val result = buildList {
        takeFunction(::get)
    } // result 的类型为 List<Float>
}
```

在分析结束后, 构建器推断考虑收集的所有类型信息, 尝试合并这些信息得到结果类型. 请看下面的示例.

```
val result = buildList { // 开始推断推迟类型变量 E
    // 认为 E 是 Number 或 Number 的一个子类型
    val n: Number? = getOrNull(0)
    // 认为 E 是 Int 或 Int 的一个超类型
    add(1)
    // E 被推断为 Int
} // result 的类型为 List<Int>
```

结果类型是与分析过程中收集到的类型信息对应的最具体的类型. 如果给定的类型信息是发生矛盾, 无法合并, 编译器会报告错误.

注意, 只有在通常的类型推断无法推断类型参数时, Kotlin 编译器才会使用构建器推断. 也就是说, 你可以在构建器 Lambda 表达式之外贡献类型信息, 那么就不需要构建器推断分析了. 请看下面的示例:

```
fun someMap() = mutableMapOf<CharSequence, String>()

fun <E> MutableMap<E, String>.f(x: MutableMap<E, String>) { ... }

fun main() {
    val x: Map<in String, String> = buildMap {
        put("", "")
        f(someMap()) // 类型不匹配 (要求 String 类型, 但实际是
CharSequence 类型)
    }
}
```

这里会出现类型不匹配, 因为在构建器 Lambda 表达式之外指定了期待的 Map 类型. 编译器会使用固定的接受者类型 `Map<in String, String>` 来分析 Lambda 表达式内的所有的语句.

Null 值安全性

最终更新: 2024/09/10

可为 null 的类型与不可为 null 的类型

Kotlin 类型系统的设计目标就是希望消除 null 引用带来的危险, 也就是所谓的 造成十亿美元损失的大错误 (https://en.wikipedia.org/wiki/Tony_Hoare#Apologies_and_retractions).

在许多编程语言(包括 Java)中, 最常见的陷阱之一就是, 对一个指向 null 值的对象访问它的成员, 导致一个 null 引用异常. 在 Java 中就是 `NullPointerException`, 简称 *NPE*.

在 Kotlin 中只有以下情况可能导致 NPE:

- 明确调用 `throw NullPointerException()`.
- 使用 `!!` 操作符, 详情见后文.
- 初始化过程中存在数据不一致, 比如:
  - 在构造器中可以访问到未初始化的 `this`, 并且将它传递给了其他代码, 然后在其他代码中使用了这个未初始化的 `this` (也就是所谓的 "leaking `this` " 警告);
  - 基类的构造器调用了 open 的成员函数 ("子类的初始化顺序" in "继承"), 但这个成员函数在子类中的实现使用了未初始化的状态数据.
- Java 互操作:
  - 试图对一个 平台类型 ("Null 值安全性与平台数据类型" in "在 Kotlin 中调用 Java 代码")的 `null` 引用访问其成员函数.
  - 用于 Java 互操作的泛型类型的可空性存在问题. 比如, 一段 Java 代码可能向一个 Kotlin `MutableList<String>` 中添加一个 `null` 值, 因此, 对这种情况应该使用 `MutableList<String?>`.
  - 外部 Java 代码导致的其他问题.

在 Kotlin 中, 类型系统明确区分可以指向 `null` 的引用 (可为 null 引用) 与不可以指向 null 的引用 (非 null 引用). 比如, 一个通常的 `String` 类型变量不可以指向 `null`:

```
fun main() {
//sampleStart
    var a: String = "abc" // 通常的初始化语句代表默认类型为非 null
    a = null // 编译错误
//sampleEnd
}
```

要允许 null 值, 你可以将变量声明为可为 null 的字符串, 写作 `String?`:

```
fun main() {
//sampleStart
    var b: String? = "abc" // 可以设置为 null
    b = null // ok
    print(b)
//sampleEnd
}
```

现在, 假如你对 `a` 调用方法或访问属性, 可以确信不会产生 NPE, 因此你可以安全地编写以下代码:

```
val l = a.length
```

但如果你要对 `b` 访问同样的属性, 就不是安全的, 编译器会报告错误:

```
val l = b.length // 错误: 变量 'b' 可能为 null
```

但你仍然需要访问这个属性, 对不对? 有以下几种方法可以实现.

在条件语句中进行 null 检查

首先, 你可以明确地检查 `b` 是否为 `null`, 然后对两种情况分别处理:

```
val l = if (b != null) b.length else -1
```

编译器将会追踪你执行过的检查, 因此允许在 `if` 内访问 `length` 属性 . 更复杂的条件也是支持的:

```
fun main() {
//sampleStart
    val b: String? = "Kotlin"
```

```
    if (b != null && b.length > 0) {
        print("String of length ${b.length}")
    } else {
        print("Empty string")
    }
//sampleEnd
}
```

注意, 以上方案需要的前提是, `b` 的内容不可变 (也就是说, 对于局部变量的情况, 在 null 值检查与变量使用之间, 要求这个局部变量没有被修改, 对于类属性的情况, 要求是一个使用后端域变量的 `val` 属性, 并且不允许被后代类覆盖), 因为, 假如没有这样的限制的话, `b` 就有可能会在检查之后被修改为 `null` 值.

安全调用

要访问可能为 null 值的变量的属性, 第二个选择方案是使用安全调用操作符 `?.`:

```
fun main() {
//sampleStart
    val a = "Kotlin"
    val b: String? = null
    println(b?.length)
    println(a?.length) // 安全调用操作符在这里是不必要的
//sampleEnd
}
```

如果 `b` 不是 null, 这个表达式将会返回 `b.length`, 否则返回 `null`. 这个表达式本身的类型为 `Int?`.

安全调用在链式调用的情况下非常有用. 比如, 雇员 Bob 可能被派属某个部门 Department (也可能不属于任何部门), 这个部门可能存在另一个雇员担任部门主管. 为了取得 Bob 所属部门的主管的名字, (如果存在的话), 你可以编写下面的代码:

```
bob?.department?.head?.name
```

这样的链式调用, 只要属性链中任何一个属性为 `null`, 整个表达式就会返回 `null`.

如果需要只对非 null 的值执行某个操作, 你可以组合使用安全调用操作符和 `let` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/let.html):

```
fun main() {
//sampleStart
    val listWithNulls: List<String?> = listOf("Kotlin", null)
    for (item in listWithNulls) {
      item?.let { println(it) } // 打印 Kotlin, 并忽略 null 值
 }
//sampleEnd
}
```

在赋值运算的左侧也可以使用安全调用. 这时, 如果链式安全调用中的任何一个接受者为 `null`, 赋值运算就会被跳过, 完全不会对赋值运算右侧的表达式进行计算:

```
// 如果 `person` 或 `person.department` 为 null, 那么这个函数不会被调用:
person?.department?.head = managersPool.getManager()
```

可为 null 的接受者

可以对 可为 null 的接受者 ("可为空的接收者(Nullable Receiver)" in "扩展") 定义扩展函数. 通过这种方式, 你可以指定 null 值的行为, 而不必在每次调用时都进行 null 检查.

例如, `toString()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/to-string.html) 函数, 定义在可为 null 的接受者上. 它返回字符串 "null" (而不是一个 `null` 值). 在某些情况下会很有用, 比如, 在输出日志时:

```
val person: Person? = null
logger.debug(person.toString()) // 日志会输出 "null", 不会抛出异常
```

如果你希望你的 `toString()` 调用返回可为 null 的字符串, 请使用 安全调用操作符 `?.`:

```
var timestamp: Instant? = null
val isoTimestamp = timestamp?.toString() // 返回 String? 对象, 这里的结果是 `null`
if (isoTimestamp == null) {
   // 处理 timestamp 是 `null` 的情况
}
```

Elvis 操作符

假设你有一个可为 null 的引用 `b`, 你可以说, "如果 `b` 不为 `null`, 那么就使用它, 否则, 就使用某个非 null 的值":

```
val l: Int = if (b != null) b.length else -1
```

除了上例这种完整的 `if` 表达式之外, 你还可以使用 Elvis 操作符 `?:` 来表达:

```
val l = b?.length ?: -1
```

如果 `?:` 左侧的表达式值不是 `null`, Elvis 操作符就会返回它的值, 否则, 返回右侧表达式的值. 注意, 只有在左侧表达式值为 `null` 时, 才会计算右侧表达式.

由于在 Kotlin 中 `throw` 和 `return` 都是表达式, 因此它们也可以用在 Elvis 操作符的右侧. 这种用法很方便, 比如, 可以用来检查函数参数值是否合法:

```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw
IllegalArgumentException("name expected")
    // ...
}
```

!! 操作符

对于 NPE 的热爱者们来说, 还有第三个选择方案: 非 null 判定操作符(`!!`) 可以将任何值转换为 非 null 类型, 如果这个值是 `null` 则会抛出一个异常. 你可以写 `b!!`, 对于 `b` 不为 null 的情况, 这个表达式将会返回这个非 null 的值 (比如, 在我们的例子中就是一个 `String` 类型值), 如果 `b` 是 `null`, 这个表达式就会抛出一个 NPE:

```
val l = b!!.length
```

所以, 如果你确实想要 NPE, 你可以抛出它, 但你必须明确地提出这个要求, 否则 NPE 不会在你没有注意的地方忽然出现.

安全的类型转换

如果对象不是我们期望的目标类型, 那么通常的类型转换就会导致 `ClassCastException`. 另一种选择是使用安全的类型转换, 如果转换不成功, 它将会返回 `null`:

```
val aInt: Int? = a as? Int
```

可为 null 的类型构成的集合

如果你的有一个集合, 其中的元素是可为 null 的类型, 并且希望将其中非 null 值的元素过滤出来, 那么可以使用 `filterNotNull` 函数:

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

下一步做什么?

- 学习 在 Java 和 Kotlin 中如何处理可空性(nullability) (Java 和 Kotlin 中的可空性(Nullability)).
- 学习 确定不含 null 值的泛型 ("确定不为 null 的类型" in "泛型(Generic): in, out, where").

相等判断

最终更新: 2024/09/10

在 Kotlin 中, 存在两种相等判断:

- *结构相等* (`==`) - 使用 `equals()` 函数判断
- *引用相等* (`===`) - 判断两个引用指向同一个对象

结构相等

结构相等检查两个对象是否拥有相同的内容和结构. 结构相等使用 `==` 操作, 以及它的相反操作 `!=`, 来判断. 按照约定, `a == b` 这样的表达式将被转换为:

```
a?.equals(b) ?: (b === null)
```

如果 `a` 不为 `null`, 将会调用 `equals(Any?)` 函数. 否则(如果 `a` 为 `null`), 将会检查 `b` 是否指向 `null`:

```
fun main() {
    var a = "hello"
    var b = "hello"
    var c = null
    var d = null
    var e = d

    println(a == b)
    // 输出结果为 true
    println(a == c)
    // 输出结果为 false
    println(c == e)
    // 输出结果为 true
}
```

注意, 当明确地与 `null` 进行比较时, 没有必要优化代码: `a == null` 将会自动转换为 `a === null`.

在 Kotlin 中, 从 `Any` 开始的所有的类都会继承 `equals()` 函数. 默认情况下, `equals()` 函数实现 引用相等判断. 但是, Kotlin 中的类可以覆盖 `equals()` 函数, 实现一个自定义的相等判断逻辑, 并且通过

这种方式, 实现结构相等判断.

值类(Value Class)和数据类(Data Class) 是两种特定的 Kotlin 类型, 它们会自动覆盖 `equals()` 函数. 因此它们默认会实现结构相等判断.

但是, 对于数据类的情况, 如果 `equals()` 函数在父类中被标记为 `final`, 那么它的行为会保持不变.

很明显, 非数据类 (没有使用 `data` 修饰符声明的类) 默认不会覆盖 `equals()` 函数. 相反, 非数据类实现引用相等判断, 继承自 `Any` 类. 实现结构相等判断, 非数据类需要用一个自定义的相等判断逻辑来覆盖 `equals()` 函数.

如果需要实现自定义的相等判断, 请覆盖 `equals(other: Any?): Boolean` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/equals.html) 函数:

```
class Point(val x: Int, val y: Int) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Point) return false

        // 比较属性值, 实现结构相等判断
        return this.x == other.x && this.y == other.y
    }
}
```

> 在覆盖 equals() 函数时, 你还应该覆盖 hashCode() 函数 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/hash-code.html), 以保持相等判断和 hash 值的一致性, 确保这些函数的行为正确.

同名但参数不同的其他函数 (比如 `equals(other: Foo)`) 不会影响到使用操作符 `==` 和 `!=` 进行的相等判断.

结构相等与 `Comparable<...>` 接口定义的比较操作没有关系, 因此, 只有 `equals(Any?)` 函数的自定义实现才会影响相等操作符的结果.

引用相等

引用相等检查两个对象的内存地址, 判断它们是不是相同的实例.

引用相等使用 `===` 操作, 以及它的相反操作 `!==`, 来判断. 当, 且仅当, `a` 与 `b` 指向同一个对象时, `a === b` 结果为 `true`:

```
fun main() {
    var a = "Hello"
    var b = a
    var c = "world"
    var d = "world"

    println(a === b)
    // 输出结果为 true
    println(a === c)
    // 输出结果为 false
    println(c === d)
    // 输出结果为 true
}
```

对于运行时期表达为基本类型的那些值(比如, `Int`), `===` 判断等价于 `==` 判断.

> ⚠ 在 Kotlin/JS 中, 引用相等的实现方式是不同的. 关于相等判断, 更多详情请参见 Kotlin/JS ("相等判断" in "在 Kotlin 中使用 JavaScript 代码") 文档.

浮点数值的相等比较

如果相等比较的操作数类型可以静态地判定为 `Float` 或 `Double` (无论可否为 null), 那么相等判断将使用 IEEE 754 浮点数运算标准 (https://en.wikipedia.org/wiki/IEEE_754).

对于不是浮点值静态类型的操作数, 行为会不同. 对这样的情况, 将会使用结构相等判定. 因此, 对于不是浮点值静态类型的操作数, 判定不遵循 IEEE 标准. 在这种情况下:

- `NaN` 等于它自己
- `NaN` 认为大于任何其他元素 (包括 `POSITIVE_INFINITY`)
- `-0.0` 不等于 `0.0`

详情请参见: 浮点值的比较 ("浮点值的比较" in "数值类型").

数组的相等比较

要比较两个数组是否包含相同顺序的相同元素, 请使用 `contentEquals()`

(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/content-equals.html).

详情请参见, 数组的比较 ("比较数组" in "数组").

this 表达式

最终更新: 2024/09/10

为了表示当前函数的 *接收者(receiver)*, 你可以使用 `this` 表达式:

- 在 类 ("继承" in "类") 的成员函数中, `this` 指向这个类的当前对象实例.
- 在 扩展函数 (扩展) 中, 或 带接收者的函数字面值(function literal) ("带有接受者的函数字面值" in "高阶函数与 Lambda 表达式") 中, `this` 代表调用函数时, 在点号左侧传递的 *接收者* 参数.

如果 `this` 没有限定符, 那么它指向 *包含当前代码的最内层范围*. 如果想要指向其他范围内的 `this`, 需要使用 *标签限定符*:

带限定符的 this

为了访问更外层范围(比如 类 (类), 或 扩展函数 (扩展), 或有标签的 带接受者的函数字面值 ("带有接受者的函数字面值" in "高阶函数与 Lambda 表达式"))内的 `this`, 你可以使用 `this@label`, 其中的 `@label` 是一个 标签 (返回与跳转: break 与 continue), 代表你想要访问的 `this` 所属的范围:

```
class A { // 隐含的标签 @A
    inner class B { // 隐含的标签 @B
        fun Int.foo() { // 隐含的标签 @foo
            val a = this@A // 指向 A 的 this
            val b = this@B // 指向 B 的 this

            val c = this // 指向 foo() 函数的接受者, 一个 Int 值
            val c1 = this@foo // 指向 foo() 函数的接受者, 一个 Int 值

            val funLit = lambda@ fun String.() {
                val d = this // 指向 funLit 的接受者, 一个 String 值
            }

            val funLit2 = { s: String ->
                // 指向 foo() 函数的接受者, 因为包含当前代码的 Lambda 表达
式没有接受者
                val d1 = this
            }
```

```
        }
    }
}
```

隐含的 this

在 `this` 上调用成员函数时, 可以省略 `this.` 部分. 如果你有一个非成员函数使用了相同的名称, 那么使用时要小心, 因为某些情况下会调用到非成员函数:

```
fun main() {
//sampleStart
    fun printLine() { println("Top-level function") }

    class A {
        fun printLine() { println("Member function") }

        fun invokePrintLine(omitThis: Boolean = false)  {
            if (omitThis) printLine()
            else this.printLine()
        }
    }

    A().invokePrintLine() // 这里会调用到成员函数
    A().invokePrintLine(omitThis = true) // 这里会调用到顶级函数
//sampleEnd()
}
```

异步编程(Asynchronous Programming)技术

最终更新: 2024/09/10

过去几十年来, 作为开发者, 我们始终面对一个问题需要解决 - 怎么样才能让我们的应用程序不要发生阻塞. 无论我们在开发桌面应用程序, 移动应用程序, 甚至服务器端的应用程序, 我们都希望避免让用户等待甚至更糟的情况, 因为会导致瓶颈, 使得应用程序无法扩展到更大规模.

现在已经有了很多种方案来解决这个问题, 包括:

- 线程(Thread)
- 回调(Callback)
- Future, Promise, 以及其他
- Reactive Extension
- 协程(Coroutine)

在解释协程之前, 先让我们简单回顾一下其他解决方案.

线程(Thread)

要避免程序阻塞, 线程(Thread)可能是大家最熟悉的解决方案.

```
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}

fun preparePost(): Token {
    // 发起请求, 随后阻塞主线程
    return token
}
```

我们假设上面的代码中的 `preparePost` 是一个长时间执行的处理, 因此会阻塞 UI. 我们可以在一个独立的线程中启动它. 这样我们就可以避免 UI 阻塞. 这是非常常见的技术, 但有很多缺点:

- 线程代价高昂. 线程需要 context 切换, 这个代价很高.
- 线程不是无限的. 底层的操作系统限制了能够启动的线程数量. 在服务器端应用程序中, 这个限制可能导致显著的瓶颈.
- 线程并不总是可用的. 在某些平台, 比如 JavaScript, 甚至根本不支持线程.
- 线程使用困难. 调试线程, 以及避免竞争条件, 都是我们在多线程编程中遭遇的常见问题.

回调(Callback)

使用回调, 基本想法是将回调函数作为参数传给另一个函数, 然后在处理结束后调用这个回调函数.

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}

fun preparePostAsync(callback: (Token) -> Unit) {
    // 发起请求, 并立即返回
    // 调度回调函数, 在之后的时刻调用它
}
```

这个原则感觉好像是更加优雅的解决方案, 但仍然存在一系列的问题:

- 回调的嵌套很困难. 通常来说, 被用作回调的函数, 经常会需要它自己的回调. 因此导致一系列的嵌套回调, 代码极难理解. 这种模式经常被称为 "圣诞树" (代码中的括号相当于圣诞树的树枝).
- 错误处理很复杂. 嵌套模型导致错误的处理和传播变得更加复杂.

在事件循环架构中, 比如 JavaScript, 回调是非常常见的, 但即使在这种场景, 通常人们也会改为使用其他方案, 比如 Promise 或 Reactive Extension.

Future, Promise, 以及其他

Future 或 Promise (其他语言和平台上也可能会使用别的名称), 背后的理念是, 当我们发起一个调用, 它向我们承诺, 在某个时间点它会返回一个对象, 称为 Promise, 然后我们可以对它进行操作.

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token ->
            submitPostAsync(token, item)
        }
        .thenAccept { post ->
            processPost(post)
        }

}

fun preparePostAsync(): Promise<Token> {
    // 发起请求, 并返回一个 promise, 它会在之后的时刻完成
    return promise
}
```

这个方案要求我们的编程方式发生很多变化, 具体来说是:

- 不同的编程模型. 与回调类似, 编程模型不再是自顶向下的命令模式(top-down imperative approach), 而是变为一种由链式调用构成的组合模式(compositional model). 传统的编程结构比如循环, 异常处理, 等等, 在这种模式中通常不再可用了.
- 不同的 API. 通常需要学习完全不同的新 API, 比如 `thenCompose` 或 `thenAccept`, 而且这些函数在不同的平台上也可能存在差异.
- 特殊的返回类型. 返回类型不再是我们需要的实际数据, 而是一个新的类型 `Promise`, 我们需要从它得到数据.
- 错误处理很复杂. 错误的传播和链条通常很不直观.

Reactive Extension

Erik Meijer (https://en.wikipedia.org/wiki/Erik_Meijer_(computer_scientist)) 将 Reactive Extension (Rx) 引入到了 C# 中. 尽管它在 .NET 平台得到了大量应用, 但并没有被主流开发者采用,

直到 Netflix 将它移植到 Java, 命名为 RxJava. 在那之后, 对很多平台有了大量的移植, 包括 JavaScript (RxJS).

Rx 背后的理念是所谓 `可观察的流(observable stream)`, 我们将数据看作流(stream)(包含无限数量的数据), 而且可以观察这些流. 从实践层面来讲, Rx 只不过是 观察者模式(Observer Pattern) (https://en.wikipedia.org/wiki/Observer_pattern), 并带有一系列的扩展, 使得我们可以对数据进行操作.

这个方案与 Future 很类似, 但 Future 可以被看作是返回一个单独的元素, Rx 则返回一个流. 但是, 与前面的方案类似, Rx 也带来了编程模型的全新的理念, 如同下面这句名言所说:

```
"任何东西都是流, 而且可以观察"
```

这表示要用不同的方式来解决问题, 与我们以前编写同步代码相比, 编程方式发生显著的变化. 有一个优点是, 与 Future 不同, 由于 Rx 被移植到了很多平台, 因此不论编程语言是 C#, Java, JavaScript, 还是可以使用 Rx 的任何其他语言, 通常我们可以得到一致的 API 体验.

此外, Rx 还引入了比较好的错误处理方案.

协程(Coroutine)

Kotlin 的异步编程解决方案是使用协程(Coroutine), 它的理念是可被挂起的一段计算, 也就是说, 一个函数的执行在某个时刻可以被挂起, 并在之后的某个时刻恢复运行.

协程的优点之一是, 对于开发者来说, 非阻塞代码的编写方式与编写阻塞代码基本上是一样的. 编程模型本身并没有发生变化.

以下面的代码为例:

```
fun postItem(item: Item) {
    launch {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

suspend fun preparePost(): Token {
    // 发起请求, 并挂起协程
    return suspendCoroutine { /* ... */ }
}
```

这段代码会启动一个长时间运行的操作, 但不会阻塞主线程. `preparePost` 是一个 `挂起函数 (suspendable function)`, 因此它的前缀添加了关键字 `suspend`. 上面这段话的意思是说, 从时间序列上来看, 函数会在某个时刻开始运行, 暂停运行, 然后又恢复运行.

- 函数签名完全不变. 唯一的区别是添加 `suspend` 关键字. 但返回类型仍然是我们希望返回的数据类型.
- 代码的编写方式仍然与编写同步代码的方式一样, 自顶向下, 除了使用 `launch` 函数来启动协程之外(具体细节在其他教程中解释), 不需要任何特殊的语法.
- 编程模型和 API 仍然保持不变. 我们可以继续使用循环, 错误处理, 等等. 不需要学习完全不用的一组新 API.
- 平台独立. 无论我们的编译目标是 JVM, JavaScript 还是其他平台, 我们编写的代码是一样的. 编译器会负责协程代码与各个平台之间的调节工作.

协程不是由 Kotlin 独自发明的一个新概念. 它已经存在了几十年, 并在其他编程语言中大量使用, 比如 Go. 值得注意的是协程在 Kotlin 中的实现方式, 大多数功能交给库来实现. 实际上, 除了 `suspend` 关键字, Kotlin 语言没有添加其他关键字. 这一点与其他语言不同, 比如 C# 的语法添加了 `async` 和 `await`. 而在 Kotlin 中, 这些只是库函数.

更多详情, 请参见 协程参考文档 (协程(Coroutine)).

协程(Coroutine)

最终更新: 2024/09/10

异步(Asynchronous)程序开发, 或者叫非阻塞(non-blocking)程序开发, 是一种软件开发的一个重要部分. 当开发服务器端程序, 桌面程序, 或移动设备应用程序时, 不仅需要为用户提供流畅的使用体验, 而且还需要根据负载大小灵活伸缩, 这二者都是很重要的能力.

Kotlin 采用一种灵活的方式解决这类问题, 在语言层提供 协程 (https://en.wikipedia.org/wiki/Coroutine) 的支持, 然后将大部分具体功能交给运行库来实现.

协程不仅帮助我们实现了异步程序开发, 还提供了更丰富的可能, 比如, 可以实现并发型模式 (Concurrency), Actor模式.

如何开始

如果你是 Kotlin 新手, 请先阅读 Kotlin 入门 (Kotlin 入门).

文档

- 协程简介 (协程指南)
- 基本概念 (协程的基本概念)
- 频道(Channel) (通道(Channel))
- 协程上下文与派发器(Dispatcher) (协程上下文与派发器(Dispatcher))
- 共享的可变状态与并发 (共享的可变状态与并发)
- 异步的数据流(Asynchronous Flow) (异步的数据流(Asynchronous Flow))

教程

- 异步程序开发 (异步编程(Asynchronous Programming)技术)
- 协程(Coroutine)与通道(Channel)简介 (教程 - 协程与通道(Channel))
- 使用 IntelliJ IDEA 调试协程 (教程 - 使用 IntelliJ IDEA 调试协程)
- 使用 IntelliJ IDEA 调试 Kotlin 数据流(Flow) (教程 - 使用 IntelliJ IDEA 调试 Kotlin 数据流 (Flow))

* 在 Android 平台测试 Kotlin 协程 (https://developer.android.com/kotlin/coroutines/test)

示例项目

* kotlinx.coroutines 示例和源代码 (https://github.com/Kotlin/kotlin-coroutines/tree/master/examples)
* KotlinConf App (https://github.com/JetBrains/kotlinconf-app)

注解

最终更新: 2024/09/10

注解是用来为代码添加元数据(metadata)的一种手段. 要声明一个注解, 需要在类之前添加 `annotation` 修饰符:

```
annotation class Fancy
```

注解的其他属性, 可以通过向注解类添加元注解(meta-annotation)的方法来指定:

- `@Target` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-target/index.html) 指定这个注解可被用于哪些元素(比如类, 函数, 属性, 表达式);
- `@Retention` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-retention/index.html) 指定这个注解的信息是否被保存到编译后的 class 文件中, 以及在运行时是否可以通过反射访问到它 (默认情况下, 这两个设定都是 true);
- `@Repeatable` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-repeatable/index.html) 允许在单个元素上多次使用同一个注解;
- `@MustBeDocumented` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-must-be-documented/index.html) 表示这个注解是公开 API 的一部分, 在自动产生的 API 文档的类或者函数签名中, 应该包含这个注解的信息.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.TYPE_PARAMETER,
AnnotationTarget.VALUE_PARAMETER,
        AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

注解的使用

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
```

```
        return (@Fancy 1)
    }
}
```

如果你需要对一个类的主构造器添加注解, 那么必须在构造器声明中添加 `constructor` 关键字, 然后在这个关键字之前添加注解:

```
class Foo @Inject constructor(dependency: MyDependency) { ... }
```

也可以对属性的访问器函数添加注解:

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

构造器

注解可以拥有带参数的构造器.

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

允许使用的参数类型包括:

- 与 Java 基本类型对应的数据类型(Int, Long, 等等.)
- 字符串
- 类 (`Foo::class`)
- 枚举
- 其他注解
- 由以上数据类型构成的数组

注解的参数不能是可为 null 的类型, 因为 JVM 不支持在注解的属性中保存 `null` 值.

如果一个注解被用作另一个注解的参数, 那么在它的名字之前不使用 `@` 前缀:

```
annotation class ReplaceWith(val expression: String)

annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead",
ReplaceWith("this === other"))
```

如果你需要指定一个类作为注解的参数, 请使用 Kotlin 类 (参见 KClass (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-class/index.html)). Kotlin 编译器会将它自动转换为 Java 类, 因此 Java 代码可以正常访问这个注解和它的参数.

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any>)

@Ann(String::class, Int::class) class MyClass
```

创建注解类的实例

在 Java 中, 注解类型是一种形式的接口, 因此你不能实现一个注解类, 并使用它的实例. Kotlin 使用不同的机制, 允许你在任意代码中调用注解类的构造器, 然后使用得到的实例.

```
annotation class InfoMarker(val info: String)

fun processInfo(marker: InfoMarker): Unit = TODO()

fun main(args: Array<String>) {
    if (args.isNotEmpty())
        processInfo(getAnnotationReflective(args))
    else
        processInfo(InfoMarker("default"))
}
```

关于创建注解类的实例, 更多详情请参见 这篇 KEEP 文档 (https://github.com/Kotlin/KEEP/blob/master/proposals/annotation-instantiation.md).

Lambda 表达式

注解也可以用在 Lambda 上. 此时, Lambda 表达式的函数体内容将会生成一个invoke() 方法, 注解将被添加到这个方法上. 这个功能对于 Quasar (https://docs.paralleluniverse.co/quasar/) 这样的框架非常有用, 因为这个框架使用注解来进行并发控制.

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

注解的使用目标(Use-site Target)

当你对一个属性或一个主构造器的参数添加注解时, 从一个 Kotlin 元素会产生出多个 Java 元素, 因此在编译产生的 Java 字节码中, 你的注解存在多个可能的适用目标. 为了明确指定注解应该使用在哪个元素上, 可以使用以下语法:

```
class Example(@field:Ann val foo,    // 对 Java 域变量添加注解
              @get:Ann val bar,      // 对属性的 Java get 方法添加注解
              @param:Ann val quux)   // 对 Java 构造器参数添加注解
```

同样的语法也可以用来对整个源代码文件添加注解. 你可以添加一个目标为 file 的注解, 放在源代码文件的最顶端, package 指令之前, 如果这个源代码属于默认的包, 没有 package 指令, 则放在所有的 import 语句之前:

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

如果你有目标相同的多个注解, 那么可以目标之后添加方括号, 然后将所有的注解放在方括号之内, 这样就可以避免重复指定相同的目标:

```
class Example {
     @set:[Inject VisibleForTesting]
```

```
    var collaborator: Collaborator
}
```

Kotlin 支持的所有注解使用目标如下:

- `file`
- `property` (使用这个目标的注解, 在 Java 中无法访问)
- `field`
- `get` (属性的 get 方法)
- `set` (属性的 set 方法)
- `receiver` (扩展函数或扩展属性的接受者参数)
- `param` (构造器的参数)
- `setparam` (属性 set 方法的参数)
- `delegate` (保存代理属性的代理对象实例的域变量)

要对扩展函数的接受者参数添加注解, 请使用以下语法:

```
fun @receiver:Fancy String.myExtension() { ... }
```

如果不指定注解的使用目标, 那么将会根据这个注解的 `@Target` 注解来自动选定使用目标. 如果存在多个可用的目标, 将会使用以下列表中的第一个:

- `param`
- `property`
- `field`

Java 注解

Kotlin 100% 兼容 Java 注解:

```
import org.junit.Test
import org.junit.Assert.*
```

```
import org.junit.Rule
import org.junit.rules.*

class Tests {
    // 对属性的 get 方法使用 @Rule 注解
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

由于 Java 注解中没有定义参数的顺序, 因此不可以使用通常的函数调用语法来给注解传递参数. 相反, 你需要使用命名参数语法:

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

与 Java 一样, 有一个特殊情况就是 `value` 参数; 这个参数的值可以不使用明确的参数名来指定:

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

使用数组作为注解参数

如果 Java 注解的 `value` 参数是数组类型, 那么在 Kotlin 中会变为 `vararg` 类型:

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

对于其他数组类型的参数, 为其赋值时你需要使用数组字面值, 或使用 `arrayOf` 函数:

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```
@AnnWithArrayMethod(names = ["abc", "foo", "bar"])
class C
```

访问注解实例的属性值

Java 注解实例的值, 在 Kotlin 代码中可以通过属性的形式访问:

```
// Java
public @interface Ann {
    int value();
}
```

```
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

不生成 JVM 1.8+ 注解目标(Target)的能力

如果一个 Kotlin 注解的 Kotlin 注解目标(Target)中包含 `TYPE`, 那么映射的 Java 注解目标会包含 `java.lang.annotation.ElementType.TYPE_USE`. 同样的, Kotlin 注解目标 `TYPE_PARAMETER` 会映射为 Java 注解目标 `java.lang.annotation.ElementType.TYPE_PARAMETER`. 对于 API 级别低于 26 的 Android 用户来说, 这会造成问题, 因为在 API 中不存在这些注解目标.

要避免生成 `TYPE_USE` 和 `TYPE_PARAMETER` 注解目标, 请使用新的编译器参数 `-Xno-new-java-annotation-targets`.

可重复注解

就象 在 Java 中 (https://docs.oracle.com/javase/tutorial/java/annotations/repeating.html) 一样, Kotlin 也有可重复注解, 它可以对同个代码元素使用多次. 要让你的注解成为可重复注解, 请在它的声明中使用 `@kotlin.annotation.Repeatable` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-repeatable/) 元注解(meta-annotation). 这样会使得这个注解在 Kotlin 和 Java 中都成为可重复注解. 在 Kotlin 中, 也支持 Java 中定义的可重复注解.

与 Java 中使用的方法的主要区别在于, 不存在 *容器注解(containing annotation)*, Kotlin 编译器会使用预定义的名称自动生成容器注解. 对于下面示例中的注解, 会生成名为 `@Tag.Container` 的容器注解:

```
@Repeatable
annotation class Tag(val name: String)

// 编译器生成名为 @Tag.Container 的容器注解
```

你可以对容器注解设置自定义的名称, 方法是使用 `@kotlin.jvm.JvmRepeatable` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvmrepeatable/) 元注解(meta-annotation), 指定一个明确声明的容器注解类作为参数:

```
@JvmRepeatable(Tags::class)
annotation class Tag(val name: String)

annotation class Tags(val value: Array<Tag>)
```

要通过反射取得 Kotlin 或 Java 的可重复注解, 请使用 `KAnnotatedElement.findAnnotations()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect.full/find-annotations.html) 函数.

关于 Kotlin 的可重复注解, 更多详情请参见 这篇 KEEP
(https://github.com/Kotlin/KEEP/blob/master/proposals/repeatable-annotations.md).

解构声明

最终更新: 2024/09/10

有些时候, 能够将一个对象 *解构(destructure)* 为多个变量, 将会很方便, 比如:

```
val (name, age) = person
```

这种语法称为 *解构声明(destructuring declaration)*. 一个解构声明会一次性创建多个变量. 上例中你声明了两个变量: `name` 和 `age`, 并且可以独立地使用这两个变量:

```
println(name)
println(age)
```

解构声明在编译时将被分解为以下代码:

```
val name = person.component1()
val age = person.component2()
```

这里的 `component1()` 和 `component2()` 函数是 Kotlin 中广泛使用的 *约定原则(principle of convention)* 的又一个例子 (其它例子请参见 `+` 和 `*` 操作符, `for` 循环). 任何东西都可以作为解构声明右侧的被解构值, 只要可以对它调用足够数量的组件函数(component function). 当然, 还可以存在 `component3()` 和 `component4()` 等等.

> `componentN()` 函数需要标记为 `operator`, 才可以在解构声明中使用.

解构声明还可以使用在 `for` 循环中:

```
for ((a, b) in collection) { ... }
```

上面的代码将遍历集合中的所有元素, 然后对各个元素调用 `component1()` 和 `component2()` 函数, 变量 `a` 和 `b` 将得到 `component1()` 和 `component2()` 函数的返回值.

示例: 从一个函数返回两个值

假如你需要从一个函数返回两个值, 比如, 一个是结果对象, 另一个是某种状态值. 在 Kotlin 中有一种紧凑的方法实现这个功能, 我们可以声明一个 数据类 (数据类(Data Class)), 然后返回这个数据类

的一个实例:

```
data class Result(val result: Int, val status: Status)
fun function(...): Result {
    // 计算

    return Result(result, status)
}

// 然后, 可以这样使用这个函数:
val (result, status) = function(...)
```

由于数据类会自动声明 `componentN()` 函数, 因此可以在这里使用解构声明.

> 你也可以使用标准库中的 `Pair` 类, 让上例中的 `function()` 函数返回一个 `Pair<Int, Status>` 实例, 但是, 给你的数据恰当地命名, 通常是一种更好的设计.

示例: 解构声明与 Map

遍历一个 map 的最好的方式可能就是:

```
for ((key, value) in map) {
   // 使用 key 和 value 执行某种操作
}
```

为了让上面的代码正确运行, 你应该:

- 实现 `iterator()` 函数, 使得 map 成为多个值构成的序列.
- 实现 `component1()` 和 `component2()` 函数, 使得 map 内的每个元素成为一对值.

Kotlin 的标准库也的确实现了这些扩展函数:

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>>
= entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

因此, 你可以在对 map 的 `for` 循环中自由地使用解构声明 (也可以在对数据类集合的 `for` 循环中使用解构声明).

用下划线代替未使用的变量

如果在解构声明中, 你不需要其中的某个变量, 你可以用下划线来代替变量名:

```
val (_, status) = getResult()
```

以这种方式跳过的变量, 不会调用对应的 `componentN()` 操作符函数.

在 Lambda 表达式中使用解构声明

你可以在 lambda 表达式的参数中使用解构声明语法. 如果 lambda 表达式的一个参数是 `Pair` 类型 (或 `Map.Entry` 类型, 或者任何其他类型, 只要它拥有适当的 `componentN` 函数), 就可以使用几个新的参数来代替原来的参数, 只需要将新参数包含在括号内:

```
map.mapValues { entry -> "${entry.value}!" }
map.mapValues { (key, value) -> "$value!" }
```

请注意声明两个参数, 与将一个参数解构为多个参数的区别:

```
{ a -> ... } // 这里是一个参数
{ a, b -> ... } // 这里是两个参数
{ (a, b) -> ... } // 这里是将一个参数解构为两个参数
{ (a, b), c -> ... } // 这里是将一个参数解构为两个参数, 然后是另一个参数
```

如果解构后得到的某个参数未被使用到, 你可以用下划线代替它, 这样就不必为它编造一个变量名了:

```
map.mapValues { (_, value) -> "$value!" }
```

你可以为解构前的整个参数指定类型, 也可以为解构后的部分参数单独指定类型:

```
map.mapValues { (_, value): Map.Entry<Int, String> -> "$value!" }

map.mapValues { (_, value: String) -> "$value!" }
```

反射

最终更新: 2024/09/10

反射 是语言与库中的一组功能, 允许你在运行时刻获取程序本身的信息. 函数和属性在 Kotlin 是语言中的一等公民(first-class citizen), 而且, 通过反射获取它们的信息(比如, 在运行时刻得到一个函数或属性的名称和数据类型) 也是函数式或交互式的编程方式中的基本功能.

> Kotlin/JS 对反射只提供了有限的支持. 更多详情请参见 Kotlin/JS 中的反射功能 (Kotlin/JS 的反射(Reflection)).

JVM 依赖项

在 JVM 平台上, Kotlin 编译器包含了使用反射功能所需要的运行时组件, 它是一个单独的 JAR 文件 `kotlin-reflect.jar`. 这样做为了对那些不使用反射功能的应用程序, 减少其运行库的大小.

在 Gradle 或 Maven 项目中, 如果需要使用反射, 需要添加 `kotlin-reflect` 的依赖项:

- 在 Gradle 项目中:

  Kotlin

  ```
  dependencies {
      implementation(kotlin("reflect"))
  }
  ```

  Groovy

  ```
  dependencies {
      implementation "org.jetbrains.kotlin:kotlin-reflect:
  {{site.data.releases.latest.version}}"
  }
  ```

- 在 Maven 项目中:

```xml
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-reflect</artifactId>
    </dependency>
</dependencies>
```

如果你没有使用 Gradle 或 Maven, 请注意将 `kotlin-reflect.jar` 添加到你的项目的 classpath 中. 对于其他支持的场景(使用命令行编译器, 或 Ant 的 IntelliJ IDEA 项目), 这个 jar 文件默认会加入到 classpath 中. 在命令行编译器和 Ant 中, 你可以使用 `-no-reflect` 编译选项, 从 classpath 中删除 `kotlin-reflect.jar`.

类引用(Class Reference)

最基本的反射功能就是获取一个 Kotlin 类的运行时引用. 要得到一个静态的已知的 Kotlin 类的引用, 可以使用 *类字面值(class literal)* 语法:

```kotlin
val c = MyClass::class
```

类引用是一个 KClass (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-class/index.html) 类型的值.

> 在 JVM 平台: Kotlin 的类引用与 Java 的类引用不相同. 要得到 Java 的类引用, 请使用 `KClass` 对象实例的 `.java` 属性.

与对象实例绑定的类引用语法

`::class` 语法同样可以用于取得某个对象实例的类的引用:

```kotlin
val widget: Widget = ...
assert(widget is GoodWidget) { "Bad widget:
${widget::class.qualifiedName}" }
```

在这个例子中, 尽管 widget 的类型为 `Widget`, 但你会得到对象实例的确切的类的引用, 比如 `GoodWidget`, 或 `BadWidget`.

可调用的引用

指向函数, 属性, 构造器的引用, 可以被调用, 或用作 函数类型 ("函数类型(Function Type)" in "高阶函数与 Lambda 表达式") 的实例.

所有可调用的引用的共同的超类是 KCallable<out R> (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-callable/index.html), 这里的 R 是返回值的类型. 对于属性来说就是属性类型, 对构造器来说就是它创建出来的类的类型.

函数引用(Function Reference)

假设你有一个有名称的函数, 声明如下, 你可以直接调用它(isOdd(5)):

```
fun isOdd(x: Int) = x % 2 != 0
```

另一种情况是, 你可以将它用作一个函数类型的值, 比如, 传给另一个函数作为参数. 为了实现这个功能, 可以使用 :: 操作符:

```
fun isOdd(x: Int) = x % 2 != 0

fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd))
//sampleEnd
}
```

这里的 ::isOdd 是一个 (Int) -> Boolean 函数类型的值.

函数引用的类型属于 KFunction<out R> (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-function/index.html) 的子类之一, 具体是哪个由函数的参数个数决定. 比如, 可能是 KFunction3<T1, T2, T3, R>.

:: 也可以用在重载函数上, 前提是必须能够推断出对应的函数参数类型. 比如:

```
fun main() {
//sampleStart
    fun isOdd(x: Int) = x % 2 != 0
    fun isOdd(s: String) = s == "brillig" || s == "slithy" || s ==
"tove"

    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd)) // 指向 isOdd(x: Int) 函数
```

```
//sampleEnd
}
```

或者, 你也可以将方法引用保存到一个明确指定了类型的变量中, 通过这种方式来提供必要的函数参数类型信息:

```
val predicate: (String) -> Boolean = ::isOdd   // 指向 isOdd(x: String) 函数
```

如果你需要使用一个类的成员函数, 或者一个扩展函数, 就必须使用限定符: `String::toCharArray`.

即使你将一个变量初始化赋值为一个扩展函数的引用, 编译器自动推断得到的函数类型实际上是不带接受者的, 但它会带有一个额外的参数, 对应于接受者对象. 如果想要使用带接受者的函数类型, 需要明确指定函数类型:

```
val isEmptyStringList: List<String>.() -> Boolean = List<String>::isEmpty
```

示例: 函数组合

我们来看看下面的函数:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

这个函数返回一个新的函数, 由它的两个参数代表的函数组合在一起构成: `compose(f, g) = f(g(*))`. 你可以使用可以执行的函数引用来调用这个函数:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}

fun isOdd(x: Int) = x % 2 != 0

fun main() {
//sampleStart
    fun length(s: String) = s.length
```

```
    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")

    println(strings.filter(oddLength))
//sampleEnd
}
```

属性引用(Property Reference)

在 Kotlin 中, 可以将属性作为一等对象来访问, 方法是使用 `::` 操作符:

```
val x = 1

fun main() {
    println(::x.get())
    println(::x.name)
}
```

表达式 `::x` 的计算结果是一个属性对象, 类型为 `KProperty0<Int>`. 你可以通过它的 `get()` 方法得到属性值, 或者通过它的 `name` 属性得到属性名称. 详情请参见 `KProperty` 类的 API 文档 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-property/index.html).

对于值可变的属性, 比如, `var y = 1`, `::y` 返回的属性对象的类型为 `KMutableProperty0<Int>` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-mutable-property/index.html), 它有一个 `set()` 方法:

```
var y = 1

fun main() {
    ::y.set(2)
    println(y)
}
```

所有使用单参数函数的地方都可以使用属性引用:

```
fun main() {
//sampleStart
    val strs = listOf("a", "bc", "def")
    println(strs.map(String::length))
```

```
//sampleEnd
}
```

要访问类的成员属性, 需要使用限定符, 如下:

```
fun main() {
//sampleStart
    class A(val p: Int)
    val prop = A::p
    println(prop.get(A(1)))
//sampleEnd
}
```

对于扩展属性:

```
val String.lastChar: Char
    get() = this[length - 1]

fun main() {
    println(String::lastChar.get("abc"))
}
```

与 Java 反射功能的互操作性

在 Java 平台上, Kotlin 的标准库包含了针对反射类的扩展函数, 这些反射类提供了与 Java 反射对象的相互转换功能(参见包 `kotlin.reflect.jvm`). 比如, 要查找一个 Kotlin 属性的后端域变量, 或者查找充当这个属性取值函数的 Java 方法, 你可以编写下面这样的代码:

```
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main() {
    println(A::p.javaGetter) // 打印结果为: "public final int
A.getP()"
    println(A::p.javaField)  // 打印结果为: "private final int A.p"
}
```

要查找与一个 Java 类相对应的 Kotlin 类, 可以使用 `.kotlin` 扩展属性:

```
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

构造器引用(Constructor Reference)

与方法和属性一样, 也可以引用构造器. 凡是使用使用函数类型对象的地方, 你都可以使用构造器的引用, 但这个函数类型接受的参数应该与构造器相同, 返回值应该是构造器所属类的对象实例. 引用构造器使用 `::` 操作符, 再加上类名称. 我们来看看下面的函数, 它接受的参数是一个函数, 这个函数参数本身没有参数, 并返回 `Foo` 类型:

```
class Foo

fun function(factory: () -> Foo) {
    val x: Foo = factory()
}
```

使用 `::Foo`, 也就是 `Foo` 类的无参构造器的引用, 你可以这样调用上面的函数:

```
function(::Foo)
```

指向构造器的引用的类型是 [`KFunction<out R>`](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-function/index.html) (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-function/index.html) 的子类之一, 具体是哪个由函数的参数个数决定.

与对象实例绑定的函数和属性引用

你可以引用某个具体的对象实例的方法:

```
fun main() {
//sampleStart
    val numberRegex = "\\d+".toRegex()
    println(numberRegex.matches("29"))

    val isNumber = numberRegex::matches
    println(isNumber("29"))
//sampleEnd
}
```

上面的示例使用 `matches` 方法的引用, 而不是直接调用这个方法. 这样的引用会与方法的接受者绑定在一起. 这样的引用可以直接调用(就像上面的示例程序中那样), 也可以用在任何使用函数类型的

地方:

```
fun main() {
//sampleStart
    val numberRegex = "\\d+".toRegex()
    val strings = listOf("abc", "124", "a70")
    println(strings.filter(numberRegex::matches))
//sampleEnd
}
```

我们来比较一下绑定到对象实例的引用, 以及未绑定到实例的引用. 绑定到对象实例的引用与它的接受者对象实例结合在一起, 因此接受者的类型不再是它的一个参数:

```
val isNumber: (CharSequence) -> Boolean = numberRegex::matches

val matches: (Regex, CharSequence) -> Boolean = Regex::matches
```

同样, 属性的引用也可以与对象实例绑定:

```
fun main() {
//sampleStart
    val prop = "abc"::length
    println(prop.get())
//sampleEnd
}
```

你不需要指定 `this` 接收者: `this::foo` 可以简写为 `::foo`.

与实例绑定的构造器引用

(译注: 内部类与普通类不同, 在创建内部类实例时, 需要绑定到一个具体的外部类实例.) 通过指定一个外部类的实例, 可以得到与这个外部类实例绑定的 内部类 (inner class) ("内部类(Inner class)" in "嵌套类与内部类") 的构造器引用:

```
class Outer {
    inner class Inner
}
```

```
val o = Outer()
val boundInnerCtor = o::Inner
```

Kotlin 跨平台程序开发入门

最终更新: 2024/09/10

支持跨平台程序开发是 Kotlin 的关键益处之一. 它可以减少对 不同的平台 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 编写和维护相同的代码从头开始, 同时又保持原生程序开发的灵活性和益处.

详情请参见 Kotlin Multiplatform 的优点 (Kotlin Multiplatform).

从头开始

- Kotlin Multiplatform 入门教程 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html). 使用 Kotlin Multiplatform Mobile plugin for Android Studio (https://plugins.jetbrains.com/plugin/14936-kotlin-multiplatform-mobile) 创建你的第一个跨平台应用程序, 可以同时工作在 Android 和 iOS 平台. 学习如何创建, 运行跨平台移动应用程序, 以及添加依赖项.
- 在 iOS 和 Android 平台共用 UI (https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-multiplatform-getting-started.html). 创建一个 Kotlin Multiplatform 应用程序, 使用 Compose Multiplatform UI framework (https://www.jetbrains.com/lp/compose-multiplatform/) 在 iOS, Android, 和 Desktop 平台间共用业务逻辑和 UI.

深入 Kotlin Multiplatform

当你获得 Kotlin Multiplatform 的一些经验, 并且想要知道如何解决特定的跨平台开发任务:

- 在你的 Kotlin 跨平台项目中 在不同平台间共用代码 (在不同的平台之间共用代码).
- 在开发跨平台应用程序和库时 连接到平台相关的 API (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-connect-to-apis.html).
- 为你的 Kotlin 跨平台项目 手动设置编译目标 (为 Kotlin Multiplatform 设置编译目标).
- 对标准库, 测试库, 或其他 kotlinx 库 添加依赖项 (添加跨平台库依赖项).
- 在你的项目中为产品和测试目的 配置编译任务 (配置编译任务).

* 向 Maven 仓库 发布跨平台库 (发布跨平台的库).
* 构建原生二进制文件 (构建最终的原生二进制文件) 生成可执行文件或共用库, 比如通用框架 (Universal Framework)或 XCFramework.

如何获取帮助

* **Kotlin Slack**: 获得邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up) 并加入 #multiplatform (https://kotlinlang.slack.com/archives/C3PQML5NU) 频道
* **StackOverflow**: 订阅 "kotlin-multiplatform" 标签 (https://stackoverflow.com/questions/tagged/kotlin-multiplatform)
* **Kotlin issue tracker**: 报告新问题 (https://youtrack.jetbrains.com/newIssue?project=KT)

Kotlin Multiplatform 项目结构的基础知识

最终更新: 2024/09/10

使用 Kotlin Multiplatform, 你可以在不同的平台之间共用代码. 本文解释共用代码的限制, 如何区分代码的共用部分和平台相关部分, 以及如何指定这些共用代码运行的平台.

你还会了解 Kotlin Multiplatform 项目设置的核心概念, 例如共通代码, 编译目标, 平台相关的源代码集和中间源代码集, 以及测试集成. 这些知识将会帮助你将来设置你的跨平台项目.

与 Kotlin 真正使用的模型相比, 本文使用的是简化后的模型. 但是, 这个基本模型应该可以适用于大多数情况.

共通代码(Common Code)

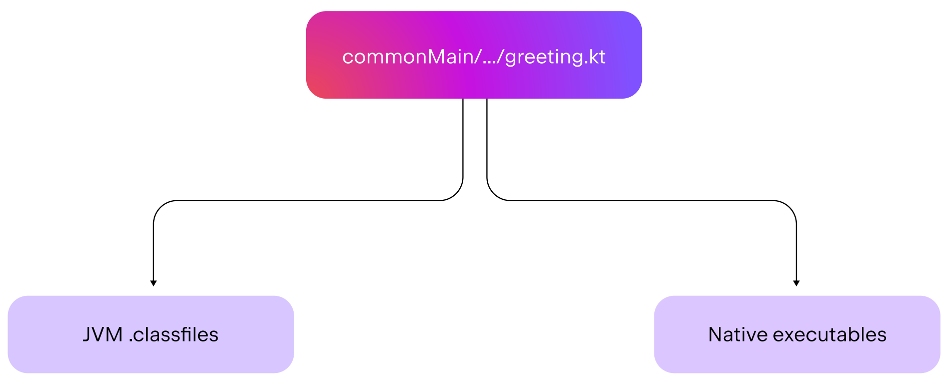
共通代码(Common Code) 是在不同平台之间共用的 Kotlin 代码.

考虑一个简单的 "Hello, World" 示例程序:

```
fun greeting() {
    println("Hello, Kotlin Multiplatform!")
}
```

在平台之间共用的 Kotlin 代码通常位于 `commonMain` 目录中. 代码文件的位置是很重要的, 它会影响到这些代码编译到哪些平台.

Kotlin 编译器以源代码作为输入, 生成一组平台相关的二进制文件作为结果. 在编译跨平台项目时, 它可以从相同的代码生成多个二进制文件. 例如, 编译器可以从相同的 Kotlin 文件生成 JVM 的 `.class` 文件, 以及原生的可执行文件:



共通代码

并不是每一段 Kotlin 代码都能够编译到所有的平台. Kotlin 编译器会阻止你在共用代码中使用平台相关的函数或类, 因为这样的代码不能编译到不同的平台.

例如, 你不能在共通代码中使用 `java.io.File` 依赖项. 它是 JDK 的一部分, 而共通代码还会被编译为原生代码, 这种情况下就不能使用 JDK 的类:

```
fun greeting() {
    java.io.File("greeting.kt").writeText("Hello, Multiplatform!")
}
```

Unresolved reference: java

Create local variable 'java' More actions...

未能解析的 Java 引用

在共通代码中, 你可以使用 Kotlin Multiplatform 库. 这些库提供了共通的 API, 在不同的平台上有不同的实现. 这种情况下, 还提供了额外的平台相关的 API, 在共通代码中使用这样的 API 会导致错误.

例如, `kotlinx.coroutines` 是一个 Kotlin Multiplatform 库, 支持所有的编译目标, 但它还有一个平台相关的部分, 将 `kotlinx.coroutines` 的并发原语转换为 JDK 的并发原语, 例如 `fun CoroutinesDispatcher.asExecutor(): Executor`. API 的这些附加部分不能在 `commonMain` 中使用.

编译目标(Target)

编译目标(Target) 定义 Kotlin 将共通代码编译到的目标平台. 这些平台可以是, 例如, JVM, JS, Android, iOS, 或 Linux. 上面的示例程序会将共通代码编译到 JVM 和原生平台.

一个 *Kotlin 编译目标* 是一个标识符, 描述一个编译的目标平台. 它定义生成的二进制文件格式, 可以使用的语言结构, 以及允许使用的依赖项.

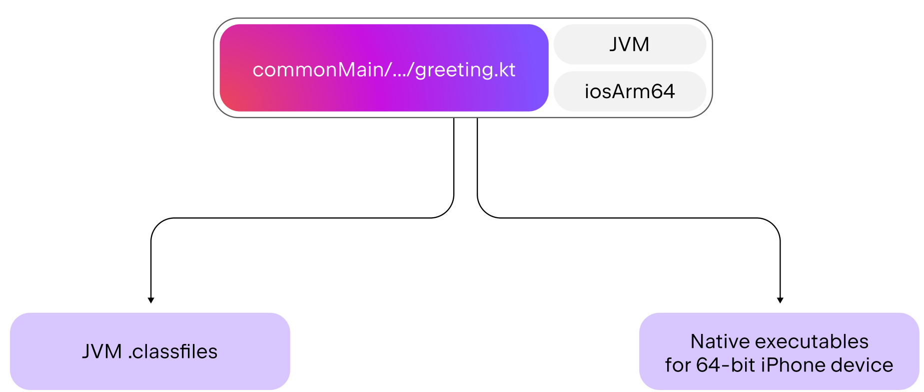
> 编译目标也可以叫做目标平台. 参见完整的 支持的编译目标列表 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档").

你应该首先 *声明* 一个编译目标, 指示 Kotlin 为这个特定的目标平台编译代码. 在 Gradle 中, 你可以在 `kotlin {}` 代码段内使用预定义的 DSL 调用来声明编译目标:

```
kotlin {
    jvm() // 声明 JVM 编译目标
    iosArm64() // 声明对应于 64-bit iPhones 的编译目标
}
```

通过这种方式, 每个跨平台项目定义一组支持的编译目标. 参见 层级项目结构 (层级项目结构) 章节, 进一步了解如何在你的构建脚本中声明编译目标.

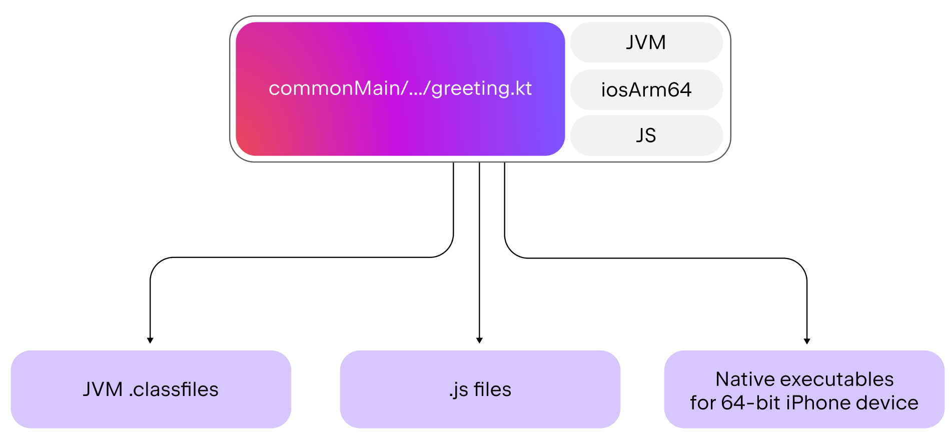
声明 `jvm` 和 `iosArm64` 编译目标之后, `commonMain` 中的共通代码将被编译到这些目标平台:



编译目标

要理解哪部分代码会被编译到特定的平台, 你可以将编译目标看作附加在 Kotlin 源代码文件上的标签. Kotlin 使用这些标签来决定如何编译你的代码, 生成哪个二进制文件, 以及代码中允许使用哪些语言结构和依赖项.

如果你还想将 `greeting.kt` 文件编译到 `.js`, 你只需要声明 JS 编译目标. `commonMain` 中的代码就会得到新的 `js` 标签, 对应于 JS 编译目标, 它会指示 Kotlin 生成 `.js` 文件:



编译目标标签

这就是 Kotlin 编译器处理共通代码的方式, 共通代码会编译到所有声明的编译目标. 参见 源代码集, 进一步了解如何编写平台相关的代码.

源代码集(Source Set)

一个 *Kotlin 源代码集(Source Set)* 是一组源代码文件, 有它独自的编译目标, 依赖项, 以及编译器选项. 它是在跨平台项目中共用代码的主要方式.

在一个跨平台项目中, 每个源代码集:

- 有一个项目中唯一的名称.
- 包含一组源代码文件和资源, 通常保存在与源代码集同名的目录中.
- 指定一组编译目标, 表示这个源代码集中的代码会编译到哪些目标平台. 这些编译目标会影响到, 这个源代码集中可以使用哪些语言结构和依赖项.
- 定义它自己的依赖项和编译器选项.

Kotlin 提供了一组预定义的源代码集. 其中一个是 `commonMain`, 它出现在所有的跨平台项目中, 并被编译到所有声明的编译目标.

在 Kotlin Multiplatform 项目中, 你可以将源代码集当作 `src` 中的目标. 例如, 一个项目有 `commonMain`, `iosMain`, 和 `jvmMain` 源代码集, 它的结构如下:

gradle
shared
build
src
commonMain
iosMain
jvmMain
build.gradle.kts 20.09.23, 13:30, 114 B

共用的代码

在 Gradle 脚本中, 你可以在 `kotlin.sourceSets {}` 代码段中通过名称访问源代码集:

```
kotlin {
    // 编译目标声明:
    // ...

    // 源代码集声明:
    sourceSets {
        commonMain {
            // 配置 commonMain 源代码集
        }
    }
}
```

除 `commonMain` 之外, 其他源代码集可以使平台相关的源代码集, 也可以是中间源代码集.

平台相关的源代码集

如果只有共通代码, 那将会非常便利, 但并不总是可行的. `commonMain` 中的代码会编译到所有声明的编译目标, 而 Kotlin 不允许你在共通代码中使用任何平台相关的 API.

在一个带有原生和 JS 编译目标的跨平台项目中, `commonMain` 中的以下代码将无法编译:

```
// commonMain/kotlin/common.kt
// 在共通代码中无法编译
fun common() {
```

```
    java.io.File("greeting.txt").writeText("Hello, Multiplatform!")
}
```

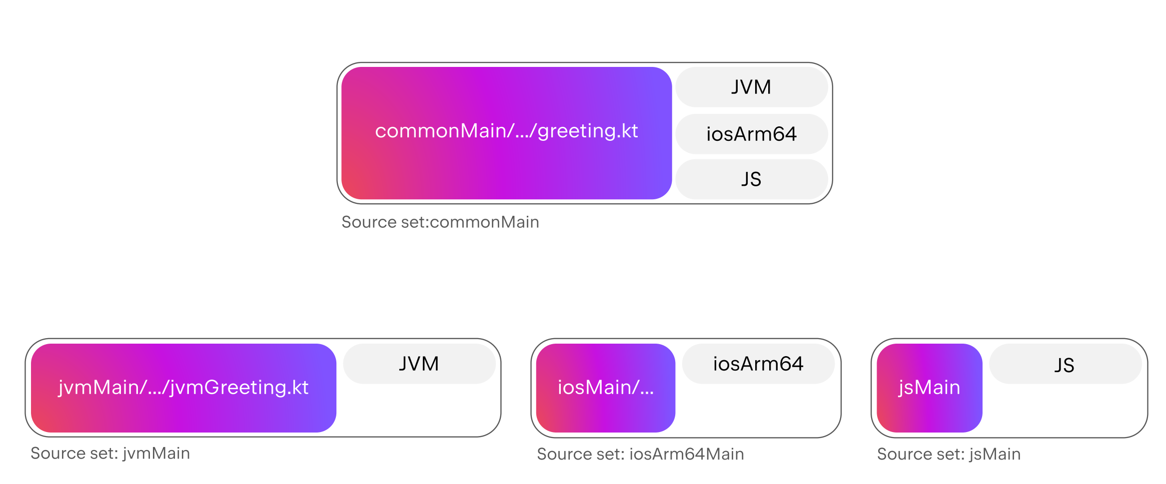
解决方案是, Kotlin 创建平台相关的源代码集, 也叫做平台源代码集. 每个编译目标有一个对应的平台源代码集, 这个源代码集只编译到这个编译目标. 例如, `jvm` 编译目标有对应的 `jvmMain` 源代码集, 这个源代码集只编译到 JVM. Kotlin 允许在这些源代码集中使用平台相关的依赖项, 例如, 在 `jvmMain` 中可以使用 JDK:

```
// jvmMain/kotlin/jvm.kt
// 你可以在 `jvmMain` 源代码集中使用 Java 依赖项
fun jvmGreeting() {
    java.io.File("greeting.txt").writeText("Hello, Multiplatform!")
}
```

编译到特定的编译目标

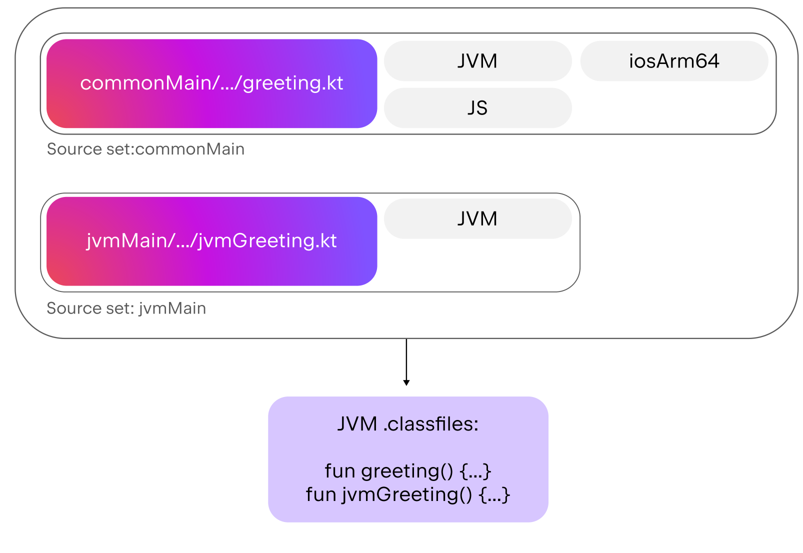
编译到特定的编译目标适用于多个源代码集. 当 Kotlin 将一个跨平台项目编译到一个特定的编译目标时, 它会收集带有这个编译目标标签的所有源代码集, 并从这些源代码集生成二进制文件.

例如, 假设有 `jvm`, `iosArm64`, 和 `js` 编译目标. Kotlin 为共通代码创建 `commonMain` 源代码集, 并为特点的编译目标创建对应的 `jvmMain`, `iosArm64Main`, 和 `jsMain` 源代码集:



编译到指定的编译目标

编译到 JVM 时, Kotlin 会选择带有 "JVM" 标签的所有源代码集, 也就是, `jvmMain` 和 `commonMain`. 然后它将这些源代码集一起编译为 JVM class 文件:



编译到 JVM

由于 Kotlin 将 `commonMain` 和 `jvmMain` 一起编译, 产生的结果二进制文件会包含来自 `commonMain` 和 `jvmMain` 的全部声明.

在开发跨平台项目时, 要记住:

- 如果你希望 Kotlin 将代码编译到特定的平台, 请声明相应的编译目标.
- 要选择一个目录或源代码文件来保存代码, 首先要决定你想在哪些编译目标之间共用你的代码:
  - 如果代码要在所有的编译目标之间共用, 那么它应该在 `commonMain` 中声明.
  - 如果代码只供一个编译目标使用, 那么它应该这个编译目标的平台源代码集在中定义(例如, 用于 JVM 平台的 `jvmMain` 源代码集).
- 在平台相关的源代码集中编写的代码, 可以访问共通源代码集中的声明. 例如, `jvmMain` 中的代码可以使用 `commonMain` 的代码. 但是, 反过来不行: `commonMain` 不能使用 `jvmMain` 中的代码.
- 在平台相关的源代码集中编写的代码, 可以使用对应的平台依赖项. 例如, `jvmMain` 中的代码可以使用 Java 专用的库, 例如 Guava (https://github.com/google/guava) 或 Spring (https://spring.io/).

中间源代码集

简单的跨平台项目通常只有共通代码和平台相关的代码. `commonMain` 源代码集代表共通代码, 在所有声明的编译目标之间共用. 平台相关的源代码集, 例如 `jvmMain`, 代表平台相关的代码, 只编译

到各自的编译目标.

在实践中, 你经常会需要更加细粒度的代码共用.

考虑一个例子, 你需要编译到所有现代的 Apple 设备和 Android 设备:

```
kotlin {
    android()
    iosArm64()   // 64 位 iPhone 设备
    macosArm64() // 现代的基于 Apple Silicon 的 Macs
    watchosX64() // 现代的 64 位 Apple Watch 设备
    tvosArm64()  // 现代的 Apple TV 设备
}
```

而且你需要一个源代码集, 用来添加一个函数, 为所有的 Apple 设备生成 UUID:

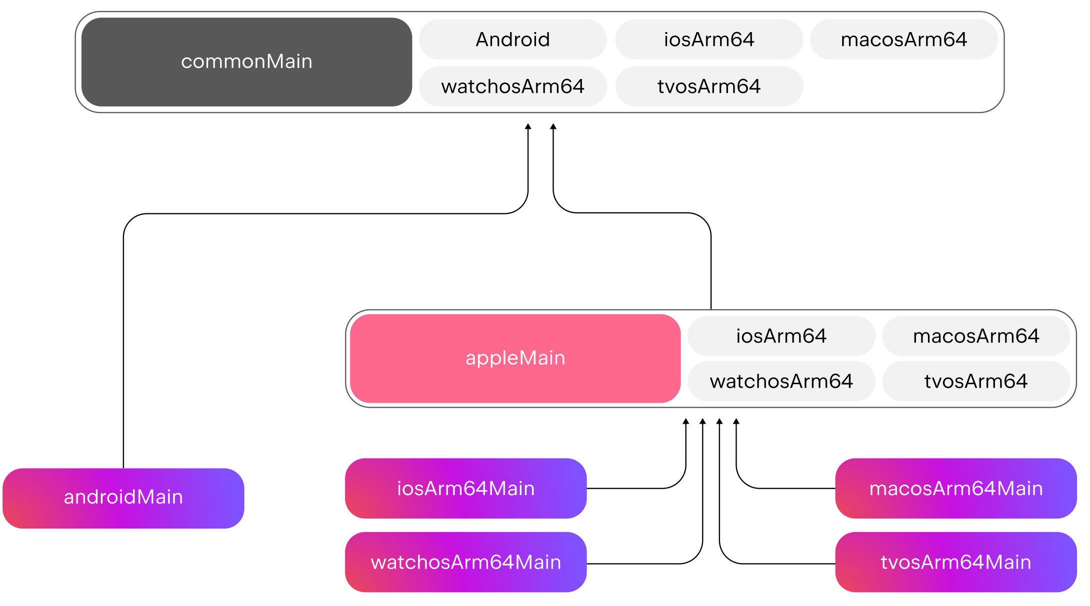
```
import platform.Foundation.NSUUID

fun randomUuidString(): String {
    // 你希望访问 Apple 专用的 API
    return NSUUID().UUIDString()
}
```

你不能将这个函数添加到 `commonMain`. `commonMain` 会编译到所有声明的编译目标, 包括 Android, 但 `platform.Foundation.NSUUID` 是一个 Apple 专用的 API, 在 Android 上无法使用. 如果你想要在 `commonMain` 中访问 `NSUUID`, Kotlin 会提示错误.

你可以将这段代码复制粘贴到每个 Apple 相关的源代码集: `iosArm64Main`, `macosArm64Main`, `watchosX64Main`, 和 `tvosArm64Main`. 但不推荐这样的方法, 因为这种重复的代码很容易导致错误.

为了解决这个问题, 你可以使用 *中间源代码集*. 中间源代码集是一个 Kotlin 源代码集, 它会编译到项目中的一部分编译目标, 但不是全部的编译目标. 你还会看到, 中间源代码集又叫做层级源代码集, 或者直接简称层级.

Kotlin 默认创建一些中间源代码集. 在这个具体案例中, 最终的项目结构类似这样:



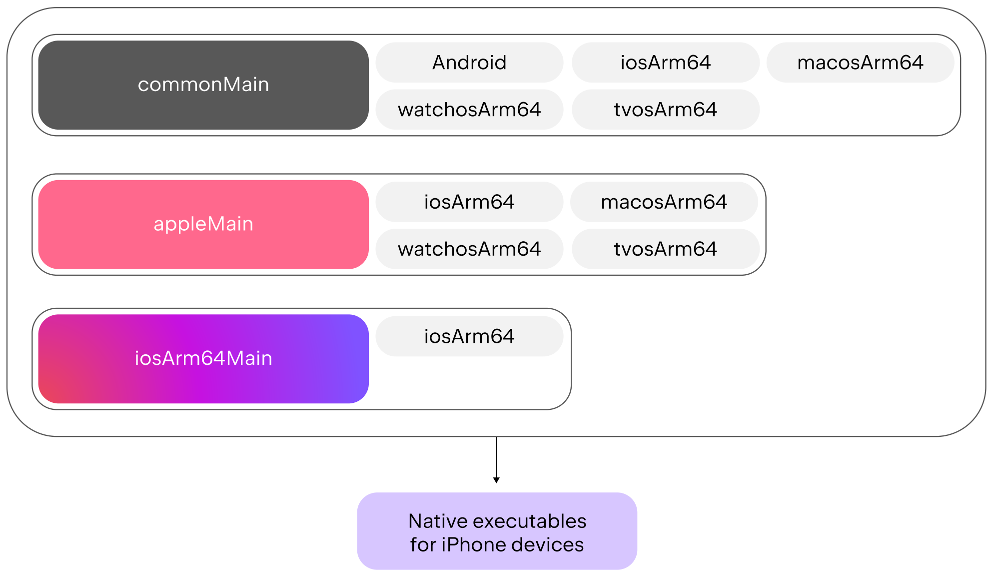
中间源代码集

其中, 下方的彩色方块是平台相关的源代码集. 为了清晰起见, 省略了编译目标的标签.

`appleMain` 方块是 Kotlin 创建的一个中间源代码集, 用于共用那些编译到 Apple 相关编译目标的代码. `appleMain` 源代码集只编译到 Apple 编译目标. 因此, Kotlin 允许在 `appleMain` 中使用 Apple 专用的 API, 你可以将 `randomUUID()` 函数添加在这里.

参见 层级项目结构 (层级项目结构), 在这里可以看到 Kotlin 默认创建和设置的所有中间源代码集, 并了解, 如果 Kotlin 没有默认提供你需要的中间源代码集, 应该如何处理.

在编译到特定的编译目标时, Kotlin 会得到所有的源代码集, 包括带有这个编译目标标签的中间源代码集. 因此, 在编译到 `iosArm64` 目标平台时, `commonMain`, `appleMain`, 和 `iosArm64Main` 源代码集中编写的所有代码会组合到一起:



原生可执行文件

> ⚠ 如果一部分源代码集中没有源代码也是可以的. 例如, 在 iOS 开发中, 通常不需要提供专用于 iOS 设备但不用于 iOS 模拟器的代码. 因此 `iosArm64Main` 很少需要用到.

Apple 设备与模拟器的编译目标

如果你使用 Kotlin Multiplatform 开发 iOS 移动应用程序, 你通常会使用 `iosMain` 源代码集. 你可能会认为它是一个平台相关的源代码集, 用于 `ios` 编译目标, 但其实并没有单独的 `ios` 编译目标. 大多数移动项目需要至少 2 个编译目标:

- **设备编译目标** 用于生成能够在 iOS 设备上执行的二进制文件. 对于 iOS, 目前只有 1 个 设备编译目标: `iosArm64`.
- **模拟器编译目标** 用于为你的机器上启动的 iOS 模拟器生成二进制文件. 如果你使用基于 Apple silicon 的 Mac 计算机, 请选择 `iosSimulatorArm64` 作为模拟器编译目标. 如果你使用基于 Intel 的 Mac 计算机, 请使用 `iosX64` 作为模拟器编译目标.

如果你只声明 `iosArm64` 设备编译目标, 那么你将无法在你的本地机器上运行和调试你的应用程序和测试程序.

平台相关的源代码集, 例如 `iosArm64Main`, `iosSimulatorArm64Main`, 和 `iosX64Main`, 通常是空的, 因为 Kotlin 用于 iOS 设备和模拟器的代码通常在同一处. 你可以只使用 `iosMain` 中间源代码集, 对所有这些平台共用代码.

对于其他非 Mac Apple 的编译目标也是如此. 例如, 如果你有 `tvosArm64` 设备编译目标, 用于 Apple TV, 以及 `tvosSimulatorArm64` 和 `tvosX64` 模拟器编译目标, 分别用于基于 Apple silicon 和基于 Intel 的设备上的 Apple TV 模拟器, 你可以对所有这些编译目标使用 `tvosMain` 中间源代码集.

与测试集成

除了 main 产品代码之外, 现实世界的项目还需要测试. 因此默认创建的所有源代码集都带有 `Main` 和 `Test` 后缀. `Main` 包含产品代码, `Test` 包含对产品代码的测试代码. 它们之间的连接会自动创建, 测试代码可以使用 `Main` 代码提供的 API, 不需要额外的配置.

对应的 `Test` 部分也是源代码集, 与 `Main` 类似. 例如, `commonTest` 对应于 `commonMain`, 并编译到所有声明的编译目标, 你可以用来编写共通的测试. 平台相关的测试源代码集, 例如 `jvmTest`, 用来编写平台相关的测试, 例如, JVM 相关的测试, 或需要 JVM API 的测试.

除了拥有源代码集来编写共通测试之外, 你还需要跨平台的测试框架. Kotlin 提供了一个默认的 `kotlin.test` (https://kotlinlang.org/api/latest/kotlin.test) 库, 其中有 `@kotlin.Test` 注解, 和各种断言方法, 例如 `assertEquals` 和 `assertTrue`.

对每个平台, 你可以在它们对应的源代码集中编写平台相关的测试, 和通常的测试一样. 和 main 代码一样, 你可以对每个源代码集设置平台相关的依赖项, 例如用于 JVM 的 `JUnit`, 和用于 iOS 的 `XCTest`. 要对特定的编译目标运行测试, 请使用 `<targetName>Test` task.

关于如何创建并运行跨平台的测试, 请参见 测试你的跨平台应用程序教程 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-run-tests.html).

下一步做什么?

- 学习如何在 Gradle 脚本中声明和使用预定义的源代码集 (层级项目结构)
- 探索跨平台项目结构中的高级概念 (跨平台项目结构的高级概念)
- 学习如何配置编译任务 (配置编译任务)

跨平台项目结构的高级概念

最终更新: 2024/09/10

本文解释 Kotlin Multiplatform 项目结构的高级概念, 以及如何对应到 Gradle 实现.

如果你正在进行下面的工作, 这些信息会对你很有用:

- 在一组特定的编译目标之间共用代码, 但 Kotlin 默认不会为这些编译目标创建源代码集. 在这种情况下, 你需要一个低层的 API, 它公开一些新的抽象.
- 需要使用 Gradle 构建的低层抽象， 例如配置, 任务, 发布, 等等.
- 正在为 Kotlin Multiplatform 构建创建 Gradle plugin.

在深入研究高级概念之前, 我们建议先学习 跨平台项目结构的基础知识 ([Kotlin Multiplatform 项目结构的基础知识](#)).

依赖关系与 dependsOn

本节介绍 2 种类型的依赖关系:

- `dependsOn` – 2 个 Kotlin 源代码集之间的, 特定的 Kotlin Multiplatform 关系.
- 通常的依赖项 – 对已发布的库的依赖, 例如 `kotlinx-coroutines`, 或对你的构建中的其他 Gradle 项目的依赖.

通常, 你会使用 *依赖项* 而不是 *`dependsOn`* 关系. 但是, 为了理解 Kotlin Multiplatform 项目的底层工作方式, 研究 `dependsOn` 是至关重要的.

dependsOn 与源代码集层级关系

`dependsOn` 是 2 个 Kotlin 源代码集之间的, Kotlin 特有的关系. 它可以是共通源代码集与平台相关源代码集之间的连接, 例如, 可以表示 `jvmMain` 源代码集依赖于 `commonMain`, `iosArm64Main` 依赖于 `iosMain`, 等等.

考虑一个一般的例子, 存在 Kotlin 源代码集 `A` 和 `B`. 表达式 `A.dependsOn(B)` 会指示 Kotlin:

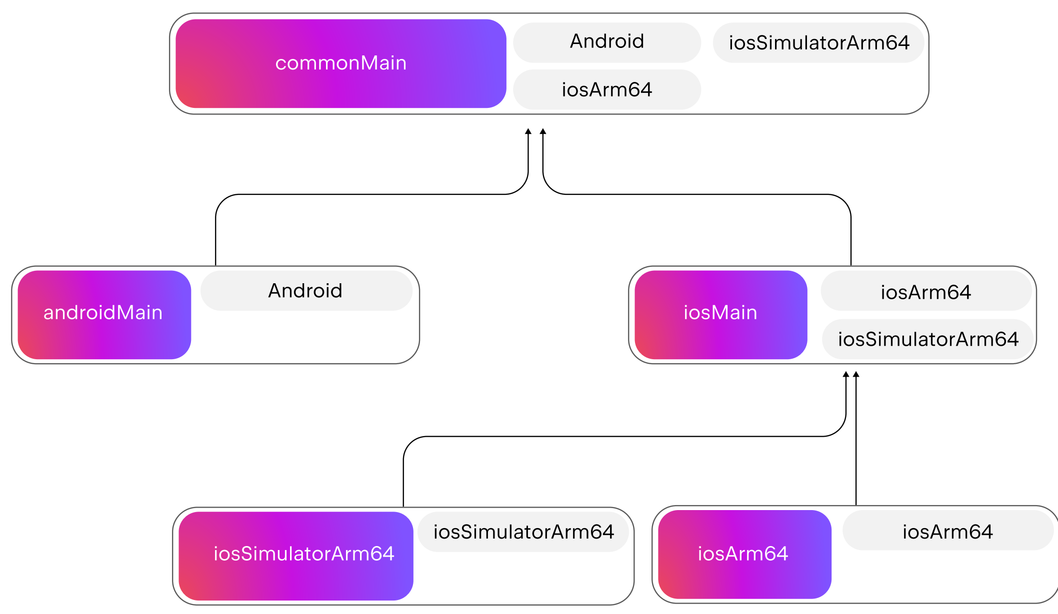
1. `A` 可以看到来自 `B` 的 API, 包括内部声明.

2. `A` 可以为来自 `B` 的预期声明提供实际实现. 这是一个必须而且充分的条件, 因为, 当而且仅当, 存在直接或间接的 `A.dependsOn(B)` 关系时, `A` 才可以为 `B` 提供 `actuals` 实现.

3. `B` 应该编译到 `A` 编译到的所有编译目标, 此外再加上它自己的编译目标.

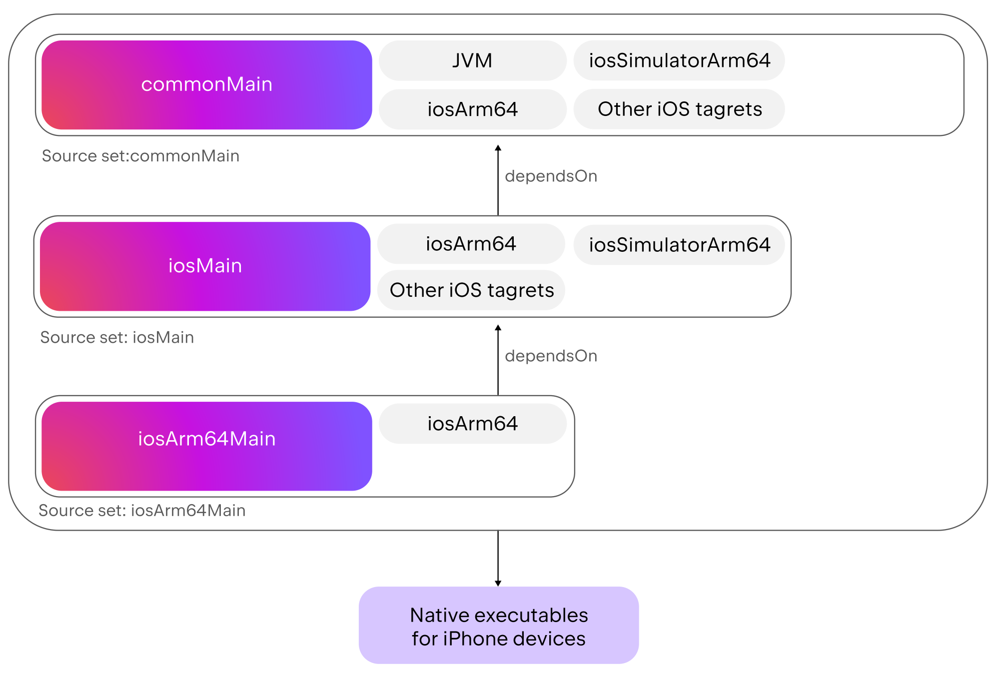
4. `A` 继承 `B` 的所有的常规依赖项.

`dependsOn` 关系会创建一个树形结构, 也叫做源代码集层级. 下面是一个通常的移动开发项目的例子, 针对的目标平台是 `androidTarget`, `iosArm64` (iPhone 设备), 和 `iosSimulatorArm64` (Apple Silicon Mac 上的 iPhone 模拟器):



DependsOn 关系树结构

图中的箭头表示 `dependsOn` 关系. 在编译平台二进制文件时, 也会保持这些关系. 所以 Kotlin 能够理解, `iosMain` 应该能够看到来自 `commonMain` 的 API, 但不能看到来自 `iosArm64Main` 的 API:



编译期间的 DependsOn 关系

`dependsOn` 关系使用 `KotlinSourceSet.dependsOn(KotlinSourceSet)` 调用来配置, 例如:

```
kotlin {
    // 编译目标的声明
    sourceSets {
        // 配置 dependsOn 关系的示例
        iosArm64Main.dependsOn(commonMain)
    }
}
```

- 这个示例演示如何在构建脚本中定义 `dependsOn` 关系. 但是, Kotlin Gradle plugin 会默认创建源代码集并设置它们的关系, 因此你不需要手动配置.
- 在构建脚本中, `dependsOn` 关系在 `dependencies {}` 代码块之外的地方声明. 这是因为 `dependsOn` 不是一种通常的依赖项; 相反, 它是 Kotlin 源代码集之间的一种特别关系, 在不同的编译目标之间共用代码时需要这些依赖关系.

你不能使用 `dependsOn` 来表达对已发布的库或对另一个 Gradle 项目的通常的依赖. 例如, 你不能设置 `commonMain` 依赖于 `kotlinx-coroutines-core` 库的 `commonMain`, 也不能调用 `commonTest.dependsOn(commonMain)`.

对其他库或项目的依赖

在跨平台项目中, 你可以设置通常的依赖项, 可以依赖到已发布的库, 或依赖到另一个 Gradle 项目.

Kotlin Multiplatform 一般会通过通常的 Gradle 方式来声明依赖项. 与 Gradle 类似, 你应该:

- 在你的构建脚本中使用 `dependencies {}` 代码块.
- 为依赖项选择适当的范围(scope), 例如, `implementation` 或 `api`.
- 引用依赖项, 如果它是已经发布到仓库中, 可以指定它的座标(coordinate), 例如 `"com.google.guava:guava:32.1.2-jre"`, 如果它是同一个构建内的一个 Gradle 项目, 可以指定它的路径, 例如 `project(":utils:concurrency")`.

跨平台项目中的依赖项配置有一些特别的功能. 每个 Kotlin 源代码集有它独自的 `dependencies {}` 代码块. 因此你可以在平台相关的源代码集中声明平台相关的依赖项:

```
kotlin {
    // 编译目标的声明
    sourceSets {
        jvmMain.dependencies {
            // 这是 jvmMain 的依赖项, 因此可以添加 JVM 相关的依赖项
            implementation("com.google.guava:guava:32.1.2-jre")
        }
    }
}
```

共通的依赖项要复杂一些. 考虑一个跨平台项目, 声明了对一个跨平台库的依赖项, 例如, `kotlinx.coroutines`:

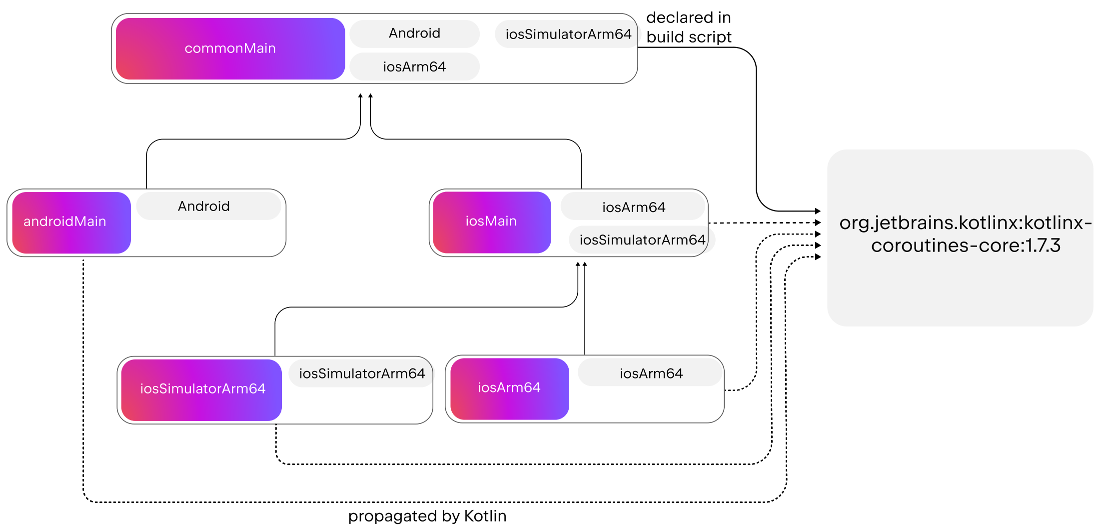
```
kotlin {
    androidTarget()     // Android
    iosArm64()          // iPhone 设备
    iosSimulatorArm64() // Apple Silicon Mac 上的 iPhone 模拟器

    sourceSets {
        commonMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3")
        }
```

```
    }
}
```

在依赖解析中, 有 3 个重要概念:

1. 跨平台依赖项会沿着 `dependsOn` 结构向下传播. 如果你对 `commonMain` 添加一个依赖项, 它会自动添加到声明了对 `commonMain` 直接或间接的 `dependsOn` 关系的所有源代码集.

   在这个例子中, 依赖项实际会被自动添加到所有的 `*Main` 源代码集: `iosMain`, `jvmMain`, `iosSimulatorArm64Main`, 和 `iosX64Main`. 所有这些源代码集会从 `commonMain` 源代码集继承 `kotlin-coroutines-core` 依赖项, 因此你不需要将依赖项手动的复制粘贴到这些源代码集中:



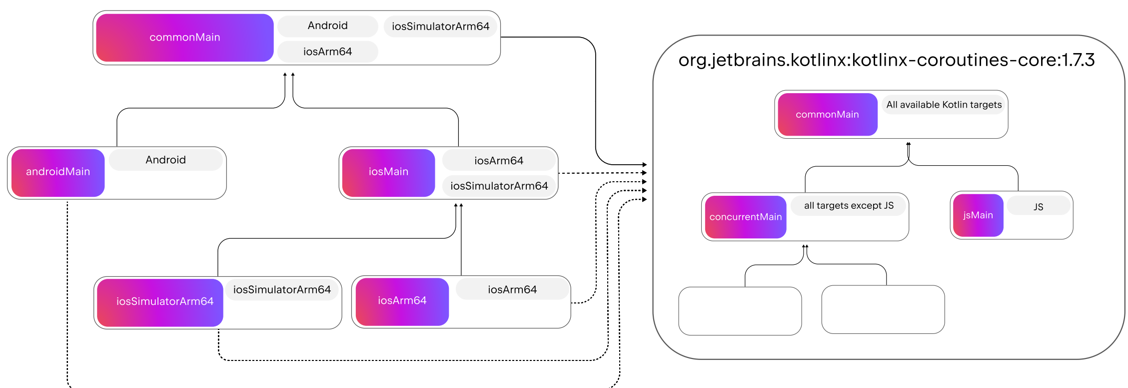
跨平台依赖项的传播

依赖项传播机制允许你通过选择特定的源代码集, 来指定接受到声明的依赖项的范围. 例如, 如果你希望在 iOS 上使用 `kotlinx.coroutines`, 但不在 Android 上使用, 你可以只对 `iosMain` 添加这个依赖项.

2. *源代码集 → 跨平台库* 依赖项, 例如上面的 `commonMain` 对 `org.jetbrians.kotlinx:kotlinx-coroutines-core:1.7.3` 的依赖, 表示依赖解析的中间状态. 解析的最终状态始终表示为 *源代码集 → 源代码集* 依赖项.

> 最终的 *源代码集 → 源代码集* 依赖项不是指 `dependsOn` 关系.

为了推断细粒度的 *源代码集* → *源代码集* 依赖项, Kotlin 会读取和每个跨平台库一起发布的源代码集结构. 完成这一步之后, 每个库的内部表达不是一个整体, 而是它的源代码集的集合. 请看 `kotlinx-coroutines-core` 的例子:



源代码集结构的序列化

3. Kotlin 对每个依赖关系解析为依赖项中源代码集的集合. 这个集合中的每个依赖项源代码集必须拥有 *兼容的编译目标*. 依赖项源代码集拥有兼容的编译目标是指, 它至少编译到 *与使用它的源代码集相同的编译目标*.

   例如, 示例项目中的 `commonMain` 编译到 `androidTarget`, `iosX64`, 和 `iosSimulatorArm64`:

   - 首先, 它解析到一个对 `kotlinx-coroutines-core.commonMain` 的依赖项. 因为 `kotlinx-coroutines-core` 编译到所有可能的 Kotlin 编译目标. 因此, 它的 `commonMain` 会编译到所有可能的编译目标, 包括这里要求的 `androidTarget`, `iosX64`, 和 `iosSimulatorArm64`.
   - 其次, `commonMain` 依赖 `kotlinx-coroutines-core.concurrentMain`. 因为 `kotlinx-coroutines-core` 中的 `concurrentMain` 编译到除 JS 之外的所有的编译目标, 它匹配使用它的项目中的 `commonMain` 的编译目标.

   但是, coroutines 中的 `iosX64Main` 之类的源代码集, 不兼容于使用它的 `commonMain` 源代码集. 即使 `iosX64Main` 编译到 `commonMain` 的编译目标之一, 也就是, `iosX64`, 但是它不编译到 `androidTarget` 或 `iosSimulatorArm64`.

   依赖解析的结果直接影响可以访问 `kotlinx-coroutines-core` 中的哪些代码:

在共通代码中使用 JVM 专用 API 的错误

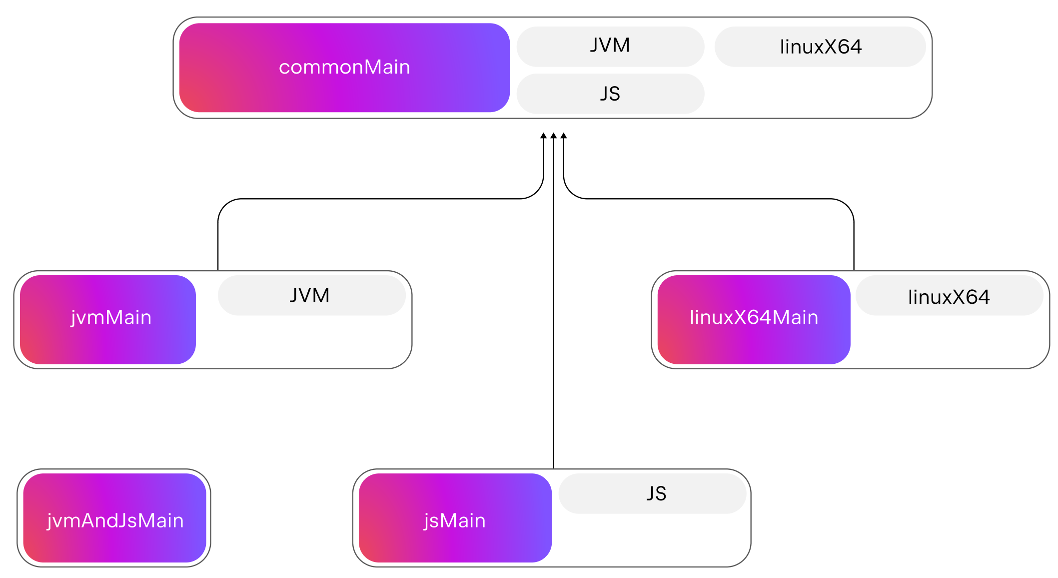
声明自定义的源代码集

有些情况下, 在你的项目中可能需要自定义的中间源代码集. 考虑一个项目, 编译到 JVM, JS, 和 Linux 平台, 你想要只在 JVM 和 JS 平台之间共用一些源代码. 这种情况下, 你应该为这组编译目标寻找一个特定的源代码集, 具体方法请参见 跨平台项目结构的基础知识 ([Kotlin Multiplatform 项目结构的基础知识](#)).

Kotlin 不会自动创建这样的源代码集. 因此你应该使用 `by creating` 构造来手动创建它:

```
kotlin {
    jvm()
    js()
    linuxX64()

    sourceSets {
        // 创建名为 "jvmAndJs" 的源代码集
        val jvmAndJsMain by creating {
            // ...
        }
    }
}
```

但是, Kotlin 仍然如何处理或者编译这个源代码集. 如果你画一个源代码集关系图, 这个源代码集将是孤立的, 没有添加任何编译目标的标签:



dependsOn 关系缺失

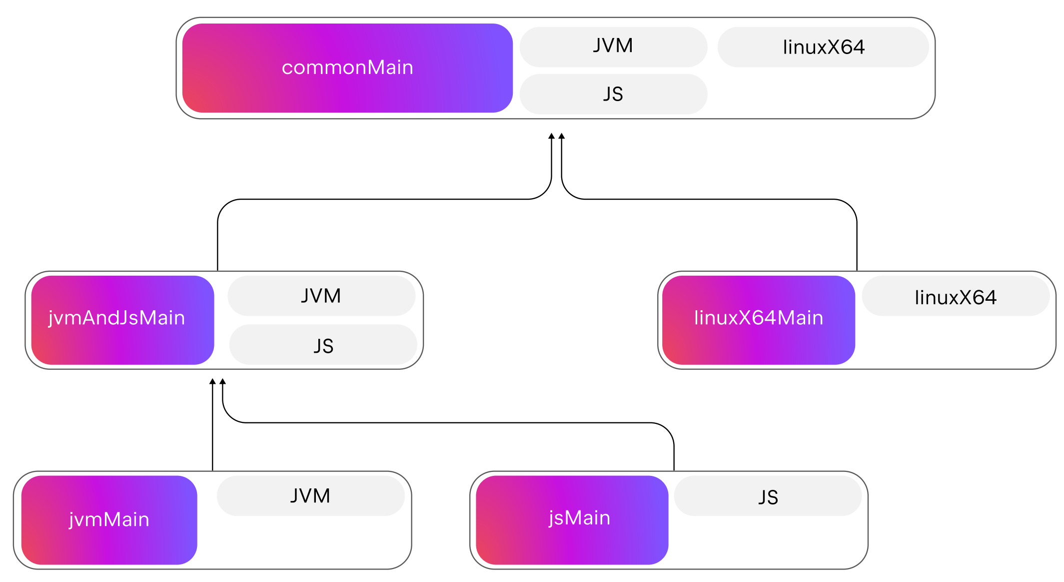
为了解决这个问题, 请添加几个 `dependsOn` 关系, 将 `jvmAndJsMain` 包含到层级结构中:

```
kotlin {
    jvm()
    js()
    linuxX64()

    sourceSets {
        val jvmAndJsMain by creating {
            // 不要忘记添加对 commonMain 的 dependsOn
            dependsOn(commonMain.get())
        }

        jvmMain {
            dependsOn(jvmAndJsMain)
        }

        jsMain {
            dependsOn(jvmAndJsMain)
        }
```

```
    }
}
```

这里, `jvmMain.dependsOn(jvmAndJsMain)` 会对 `jvmAndJsMain` 添加 JVM 编译目标, `jsMain.dependsOn(jvmAndJsMain)` 会对 `jvmAndJsMain` 添加 JS 编译目标.

最终的项目结构如下:



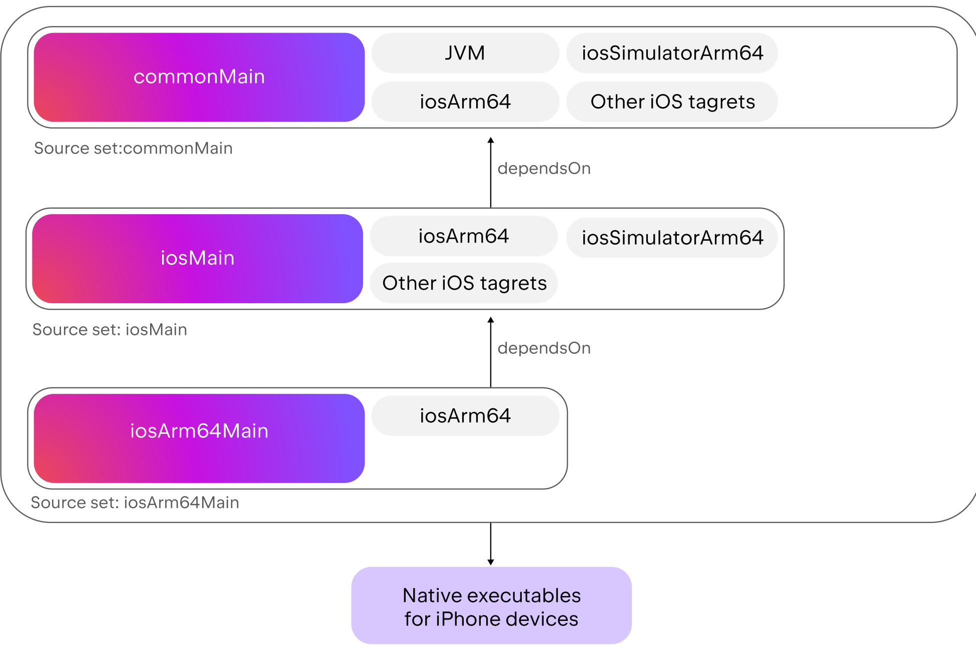
最终的项目结构

> 如果手动配置 `dependsOn` 关系, 会禁止自动使用默认的层级结构模板. 关于这样的情况, 以及处理方法, 详情请参见 附加配置 ("附加配置" in "层级项目结构").

编译

与单一平台的项目不同, Kotlin Multiplatform 项目需要多次编译器运行来构建所有的 artifact. 每次编译器运行都是一个 *Kotlin 编译*.

例如, 在前面提到的 Kotlin 编译过程中, 用于 iPhone 设备的二进制文件的生成方式如下:



针对 iOS 的 Kotlin 编译

Kotlin 编译会在编译目标之下分组. 默认情况下, Kotlin 为每个编译目标创建 2 个编译, `main` 编译用于产品源代码, `test` 编译用于测试源代码.

在构建脚本中, 编译通过类似的方式访问. 你首先选择一个 Kotlin 编译目标, 然后访问其中的 `compilations` 容器, 最后通过名称选择你需要的编译:

```
kotlin {
    // 声明并配置 JVM 编译目标
    jvm {
        val mainCompilation: KotlinJvmCompilation =
compilations.getByName("main")
    }
}
```

为 Kotlin Multiplatform 设置编译目标

最终更新: 2024/09/10

可以在使用 项目向导 (https://kmp.jetbrains.com/) 创建项目时添加编译目标. 如果之后再需要添加编译目标, 可以使用对 支持的平台 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 预设置的编译目标来手工添加.

更多详情请参见 编译目标的额外设置 ("所有编译目标的共通配置" in "跨平台程序的 Gradle DSL 参考文档").

```
kotlin {
    jvm() // 使用默认名称 'jvm' 创建 JVM 编译目标

    linuxX64() {
        /* 这里可以对 'linux' 编译目标指定额外设置 */
    }
}
```

每个编译目标可以有一个或多个 编译任务 (配置编译任务). 除了用于测试和产品的默认的编译任务之外, 你还可以 创建自定义的编译任务 ("创建自定义编译任务" in "配置编译任务").

对一个平台区分多个编译目标

在一个跨平台库中, 对一个平台可以有多个编译目标. 比如, 这些编译目标可以提供相同的 API, 但在运行时使用不同的库, 比如不同的测试框架, 不同的日志库. 对这样的跨平台库的依赖可能会解析失败, 因为不清楚应该选择哪个编译目标.

为解决这个问题, 需要在库的开发者和使用者端同时对编译目标标记一个自定义的属性, 供 Gradle 在解析依赖项目时使用.

比如, 假设有一个测试库, 在两个编译目标中支持 JUnit 和 TestNG. 库的作者需要在两个编译目标中添加一个属性, 如下:

Kotlin

```
val testFrameworkAttribute =
Attribute.of("com.example.testFramework", String::class.java)
```

```
kotlin {
    jvm("junit") {
        attributes.attribute(testFrameworkAttribute, "junit")
    }
    jvm("testng") {
        attributes.attribute(testFrameworkAttribute, "testng")
    }
}
```

Groovy

```
def testFrameworkAttribute =
Attribute.of('com.example.testFramework', String)

kotlin {
    jvm('junit') {
        attributes.attribute(testFrameworkAttribute, 'junit')
    }
    jvm('testng') {
        attributes.attribute(testFrameworkAttribute, 'testng')
    }
}
```

库的使用者编译目标如果出现依赖项目的歧义，就需要对他的编译目标添加这个属性来解决歧义.

在不同的平台之间共用代码

最终更新: 2024/09/10

通过 Kotlin Multiplatform, 你可以使用 Kotlin 提供的以下机制共用代码:

- 在你的项目中使用的所有平台上共用代码. 通过这种方式, 可以共用那些适用于所有平台的共通业务逻辑.
- 在你的项目中使用的一部分(但不是所有)平台上共用代码. 通过使用层级结构(Hierarchical Structure), 你可以在相似的平台上重用代码.

如果需要在共用代码中访问平台相关的 API, 可以使用 Kotlin 的 预期声明与实际声明(expected and actual declaration) (预期声明与实际声明) 机制.

在所有平台上共用代码

如果你的业务逻辑对所有的平台都是共通的, 那么没有必要对每个平台编写相同的代码 – 只需要在共通源代码集中共用这些代码就可以了.

源代码集之间一些依赖关系会默认设置. 对于以下源代码集, 你不需要手动指定任何 `dependsOn` 关系:

- 所有平台相关的源代码集会默认依赖于共通源代码集, 比如 `jvmMain`, `macosX64Main`, 等等.
- 某个特定编译目标的 `main` 与 `test` 源代码集之间会默认依赖, 比如 `androidMain` 与 `androidUnitTest`.

如果在共用的代码中需要访问平台相关的 API, 可以使用 Kotlin 的 预期声明与实际声明(expected and actual declaration) (预期声明与实际声明) 机制.

在类似的平台上共用代码

开发过程中经常会需要创建几种原生编译目标, 它们之间可能共用很多共通逻辑和第三方 API.

比如, 在一个包含 iOS 编译目标的典型的跨平台项目中, 有两种 iOS 相关的编译目标: 一个是 iOS ARM64 设备, 另一个是 x64 模拟器. 它们的平台相关源代码集是分开的, 但实际上真实设备和模拟器很少需要不同的代码, 而且它们的依赖项也基本相同. 因此对这些编译目标, iOS 相关的代码可以共用.

很明显, 在这种设置中, 对两个 iOS 编译目标我们需要一个共用的源代码集, 其中包含的 Kotlin/Native 代码, 仍然可以直接调用那些对 iOS 设备和模拟器共通的 API.

这种情况下, 可以通过以下任何一种方法, 使用层级结构(Hierarchical Structure) (层级项目结构), 在你的项目中的多个原生编译目标之间共用代码:

- 使用默认的层级模板 ("默认层级结构模板" in "层级项目结构")
- 手动配置层级结构 ("手动配置" in "层级项目结构")

更多详情请参见 在多个库之间共用代码 以及 连接平台相关的库.

在多个库之间共用代码

感谢项目层级结构的帮助, 多个库也可以对一组编译目标提供共通的 API. 当 库发布 (发布跨平台的库) 时, 它的中间源代码集的 API 会与项目结构信息一起嵌入到库的 artifact 中. 当你使用这个库时, 你的项目的中间源代码集只能访问各个源代码集的编译目标所能得到的库的 API.

比如, `kotlinx.coroutines` 代码仓库的源代码集层级结构如下:

`concurrent` 源代码集声明了函数 runBlocking, 然后针对 JVM 和原生编译目标进行编译. 当 `kotlinx.coroutines` 库更新并携带项目的层级结构信息一起发布之后, 你可以依赖到这个库, 并在 JVM 和原生编译目标之间共用的源代码集中调用 `runBlocking`, 因为它与库的 `concurrent` 源代码集的 "编译目标签名" 相符.

连接平台相关的库

为了共用更多的原生代码, 而不是仅仅局限于平台相关的依赖项, 你可以连接 平台相关的库 (平台库). Kotlin/Native 附带的库 (例如 Foundation, UIKit, 和 POSIX) 在共用的源代码集中默认可以使用.

此外, 如果在你的项目中使用 Kotlin CocoaPods Gradle (CocoaPods 概述与设置) plugin, 你也可以使用 `cinterop` 机制 (与 C 代码交互) 导入的第三方原生库.

下一步做什么?

- 关于使用 Kotlin 的 预期声明与实际声明 (预期声明与实际声明) 机制共用代码, 查看示例程序
- 学习 层级项目结构 (层级项目结构)

- 阅读我们推荐的 跨平台项目中的源代码文件命名规约 ("源代码文件名" in "编码规约")

预期声明与实际声明

最终更新: 2024/09/10

预期声明(Expected Declaration)与实际声明(Actual Declaration), 让你能够在 Kotlin Multiplatform 模块中访问平台相关的 API. 你可以在共通代码中提供平台无关的 API.

> ⚠ 本文描述预期声明与实际声明的语言机制. 关于使用平台相关 API 的各种方法的一般性建议, 请参见 使用平台相关的 API (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-connect-to-apis.html).

预期声明与实际声明的规则

要定义预期声明与实际声明, 请遵循这些规则:

1. 在共通源代码集中, 声明一个标准的 Kotlin 结构. 可以是一个函数, 属性, 类, 接口, 枚举, 或注解.
2. 使用 `expect` 关键字标注这个结构. 这就是你的 *预期声明(Expected Declaration)*. 这些声明可以在共通代码中使用, 但不能包含任何实现. 相反, 应该由平台相关的代码来提供实现.
3. 在每个平台相关的源代码集中, 在相同的包中声明相同的结构, 并用 `actual` 关键字标注它. 这就是你的 *实际声明(Actual Declaration)*, 它通常包含一个实现, 使用平台相关的库.

在对特定的编译目标进行编译时, 编译器尝试将它找到的每个 *实际* 声明与共通代码中对应的 *预期* 声明进行匹配. 编译器会保证以下几点:

- 共通源代码集中的每个预期声明, 在每个平台相关的源代码中都存在匹配的实际声明.
- 预期声明不包含任何实现.
- 每个实际声明与对应的预期声明使用相同的包, 例如 `org.mygroup.myapp.MyType`.

在为不同的平台生成结果代码时, Kotlin 编译器会合并相互对应的预期声明和实际声明. 它会为每个平台生成一个声明以及它的实际实现. 共通代码中对预期声明的每次使用, 在最终生成的平台代码中, 都会调用正确的实际声明.

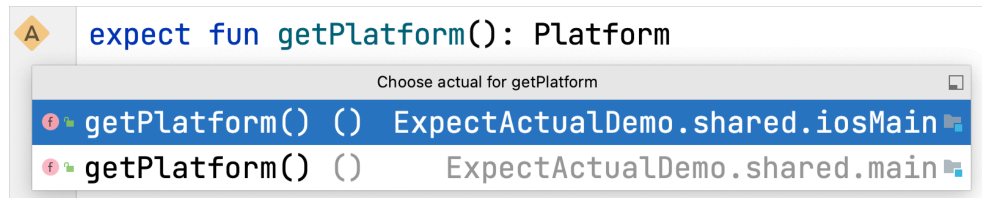
当你使用中间源代码集, 在不同的目标平台之间共用时, 你可以声明实际声明. 例如, `iosMain` 作为中间源代码集, 在平台源代码集 `iosX64Main`, `iosArm64Main`, 和 `iosSimulatorArm64Main` 之间

共用. 那么通常只有 `iosMain` 包含实际声明, 而不是那些平台源代码集. Kotlin 编译器会使用这些实际声明 来为对应的平台产生结果代码.

IDE 可以帮助解决常见的问题, 包括:

- 缺少声明
- 预期声明包含实现
- 声明的签名不匹配
- 声明处于不同的包内

你还可以使用 IDE, 在预期声明和实际声明之间导航. 请选择侧栏图标(gutter icon)来查看实际声明, 或者使用 快捷键 (https://www.jetbrains.com/help/idea/navigating-through-the-source-code.html#go_to_implementation).



IDE 中从预期声明到实际声明之间的导航

使用预期声明与实际声明的各种方案

下面我们来探索各种不同的方案, 使用 expect/actual 机制来解决访问平台 API 的问题, 同时仍然提供一种方法, 使得可以在公共代码中使用这些 API.

考虑一个 Kotlin Multiplatform 项目, 你需要实现 `Identity` 类型, 其中包含用户的登录名和当前进程 ID. 这个项目具有 `commonMain`, `jvmMain` 和 `nativeMain` 源代码集, 让应用程序可以在 JVM 运行, 也可以在 iOS 等原生环境中运行.

预期函数与实际函数

你可以定义一个 `Identity` 类型, 和一个工厂函数 `buildIdentity()`, 这个函数在共通源代码集中声明, 并在平台源代码集中以不同的方式实现:

1. 在 `commonMain` 中, 声明一个简单的类型, 以及预期的工厂函数:

```
package identity
```

```kotlin
class Identity(val userName: String, val processID: Long)

expect fun buildIdentity(): Identity
```

2. 在 `jvmMain` 源代码集中, 使用 Java 标准库实现:

```kotlin
package identity

import java.lang.System
import java.lang.ProcessHandle

actual fun buildIdentity() = Identity(
    System.getProperty("user.name") ?: "None",
    ProcessHandle.current().pid()
)
```

3. 在 `nativeMain` 源代码集中, 使用原生依赖项, 通过 POSIX (https://en.wikipedia.org/wiki/POSIX) 实现:

```kotlin
package identity

import kotlinx.cinterop.toKString
import platform.posix.getlogin
import platform.posix.getpid

actual fun buildIdentity() = Identity(
    getlogin()?.toKString() ?: "None",
    getpid().toLong()
)
```

这里, 平台函数返回平台相关的 `Identity` 实例.

> 从 Kotlin 1.9.0 开始, 使用 `getlogin()` 和 `getpid()` 函数需要标注 `@OptIn` 注解.

接口加上预期函数与实际函数

如果工厂函数变得太大, 可以考虑使用共通的 `Identity` 接口, 并在不同平台上以不同方式实现它.

`buildIdentity()` 工厂函数应该返回 `Identity`, 但这次它是一个实现共通接口的对象:

1. 在 `commonMain` 中, 定义 `Identity` 接口和 `buildIdentity()` 工厂函数:

```
// 在 commonMain 源代码集中:
expect fun buildIdentity(): Identity

interface Identity {
    val userName: String
    val processID: Long
}
```

2. 创建平台相关的接口实现, 这里不需要额外的使用预期声明和实际声明:

```
// 在 jvmMain 源代码集中:
actual fun buildIdentity(): Identity = JVMIdentity()

class JVMIdentity(
    override val userName: String =
System.getProperty("user.name") ?: "none",
    override val processID: Long = ProcessHandle.current().pid()
) : Identity
```

```
// 在 nativeMain 源代码集中:
actual fun buildIdentity(): Identity = NativeIdentity()

class NativeIdentity(
    override val userName: String = getlogin()?.toKString() ?:
"None",
    override val processID: Long = getpid().toLong()
) : Identity
```

这些平台函数返回平台相关的 `Identity` 实例, 这些实例通过 `JVMIdentity` 和 `NativeIdentity` 平台类型来实现.

预期属性与实际属性

你可以修改上面的示例, 使用一个预期的 `val` 属性来存储 `Identity`.

将这个属性标注为 `expect val`, 然后在平台源代码集中提供实际属性:

```
// 在 commonMain 源代码集中:
expect val identity: Identity

interface Identity {
    val userName: String
    val processID: Long
}
```

```
// 在 jvmMain 源代码集中:
actual val identity: Identity = JVMIdentity()

class JVMIdentity(
    override val userName: String = System.getProperty("user.name")
?: "none",
    override val processID: Long = ProcessHandle.current().pid()
) : Identity
```

```
// 在 nativeMain 源代码集中:
actual val identity: Identity = NativeIdentity()

class NativeIdentity(
    override val userName: String = getlogin()?.toKString() ?:
"None",
    override val processID: Long = getpid().toLong()
) : Identity
```

预期对象与实际对象

如果 `IdentityBuilder` 预期在每个平台上都是单子(singleton), 你可以将它定义为一个预期对象, 然后让每个平台实现它的实际对象:

```
// 在 commonMain 源代码集中:
expect object IdentityBuilder {
    fun build(): Identity
}

class Identity(
```

```
    val userName: String,
    val processID: Long
)
```

```
// 在 jvmMain 源代码集中:
actual object IdentityBuilder {
    actual fun build() = Identity(
        System.getProperty("user.name") ?: "none",
        ProcessHandle.current().pid()
    )
}
```

```
// 在 nativeMain 源代码集中:
actual object IdentityBuilder {
    actual fun build() = Identity(
        getlogin()?.toKString() ?: "None",
        getpid().toLong()
    )
}
```

关于依赖注入的建议

为了创建一种松散耦合的架构, 许多 Kotlin 项目都采用了依赖注入(DI, Dependency Injection)框架. DI 框架可以根据当前的环境将依赖注入到组件中.

例如, 你可能会在测试环境和生产环境中注入不同的依赖, 或者, 在部署到云端时与在本地托管时注入不同的依赖. 只要依赖通过接口来表达, 无论是在编译时期还是在运行时期, 都可以注入任意数量的不同实现.

当依赖与平台相关时, 也适用同样的原则. 在共通代码中, 一个组件可以使用通常的 Kotlin 接口 (接口(Interface)) 表达它的依赖. 然后可以配置 DI 框架, 来注入平台相关的实现, 例如, 来自 JVM 的实现, 或来自 iOS 模块的实现.

因此, 预期声明和实际声明只在 DI 框架的配置中需要用到. 具体的示例请参见 使用平台相关的 API (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-connect-to-apis.html#dependency-injection-framework).

使用这样的方案, 你只需要使用接口和工厂函数, 就可以便利的使用 Kotlin Multiplatform. 如果你已经使用了 DI 框架来管理你的项目中的依赖, 我们推荐使用同样的方案来管理平台相关的依赖.

预期类与实际类

预期类与实际类功能处于 Beta 版 ([Kotlin 各部分组件的稳定性](#)). 这个功能已经基本稳定, 但将来可能需要进行一些手动的源代码迁移工作. 我们会尽力减少你需要进行的代码变更.

你可以使用预期类与实际类来实现相同的解决方案:

```
// 在 commonMain 源代码集中:
expect class Identity() {
    val userName: String
    val processID: Int
}
```

```
// 在 jvmMain 源代码集中:
actual class Identity {
    actual val userName: String = System.getProperty("user.name") ?:
"None"
    actual val processID: Long = ProcessHandle.current().pid()
}
```

```
// 在 nativeMain 源代码集中:
actual class Identity {
    actual val userName: String = getlogin()?.toKString() ?: "None"
    actual val processID: Long = getpid().toLong()
}
```

你可能已经在演示材料中看到过这样方案. 但是, 对于简单的情况, 例如使用接口已经足够的情况, *不推荐* 使用类.

使用接口, 你的设计不会限制为每个目标平台一个实现. 而且, 在测试中替换一个假的实现要简单得多, 在单个平台上提供多个实现也很容易.

作为一般性原则, 应该尽可能依赖标准的语言结构, 而不要使用预期声明和实际声明.

如果你决定使用预期类和实际类, Kotlin 编译器会示警告你, 这个功能处于 Beta 状态. 要压制这个警告, 请在你的 Gradle 构建文件中添加以下编译器选项:

```
kotlin {
    compilerOptions {
        // 共通的编译器选项, 应用于所有的 Kotlin 源代码集
        freeCompilerArgs.add("-Xexpect-actual-classes")
    }
}
```

从平台类继承

有几种特殊情况, 对类使用 `expect` 关键字可能是最好的方案. 假设 `Identity` 类型在 JVM 中已经存在了:

```
open class Identity {
    val login: String = System.getProperty("user.name") ?: "none"
    val pid: Long = ProcessHandle.current().pid()
}
```

为了适合既有的代码库和框架, 你的 `Identity` 类型的实现可以从这个类型继承, 并重用它的功能:

1. 为了解决这个问题, 可以在 `commonMain` 中使用 `expect` 关键字声明一个类:

```
expect class CommonIdentity() {
    val userName: String
    val processID: Long
}
```

2. 在 `nativeMain` 中, 提供一个实际声明, 实现功能:

```
actual class CommonIdentity {
    actual val userName = getlogin()?.toKString() ?: "None"
    actual val processID = getpid().toLong()
}
```

3. 在 `jvmMain` 中, 提供一个实际声明, 从平台相关的基类继承:

```
actual class CommonIdentity : Identity() {
    actual val userName = login
```

```
    actual val processID = pid
}
```

这里, CommonIdentity 类型与你自己的设计相兼容, 同时又利用了 JVM 上既有类型的便利.

框架中的应用程序

作为框架的作者, 你也会发现预期声明和实际声明对你的框架非常有用.

假设上面的示例是一个框架的一部分, 使用者必须从 CommonIdentity 继承一个类型, 来提供一个显示名称.

这种情况下, 预期声明是抽象的, 并声明一个抽象方法:

```
// 在框架代码库的 commonMain 中:
expect abstract class CommonIdentity() {
    val userName: String
    val processID: Long
    abstract val displayName: String
}
```

类似的, 实际实现是抽象的, 声明 displayName 方法:

```
// 在框架代码库的 nativeMain 中:
actual abstract class CommonIdentity {
    actual val userName = getlogin()?.toKString() ?: "None"
    actual val processID = getpid().toLong()
    actual abstract val displayName: String
}
```

```
// 在框架代码库的 jvmMain 中:
actual abstract class CommonIdentity : Identity() {
    actual val userName = login
    actual val processID = pid
    actual abstract val displayName: String
}
```

框架的使用者需要编写共通代码, 从预期声明继承, 自行实现缺少的方法:

```
// 在使用者代码库的 commonMain 中:
class MyCommonIdentity : CommonIdentity() {
    override val displayName = "Admin"
}
```

高级使用场景

关于预期声明和实际声明, 存在一些特殊情况.

使用类型别名(type alias) 实现实际声明

实际声明的实现不一定需要从头编写. 它可以是一个既有的类型, 例如由第三方库提供的一个类.

你可以使用这个类型, 只要它满足与预期声明相关的所有要求. 例如, 考虑下面两个预期声明:

```
expect enum class Month {
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
    AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
}

expect class MyDate {
    fun getYear(): Int
    fun getMonth(): Month
    fun getDayOfMonth(): Int
}
```

在 JVM 模块中, `java.time.Month` 枚举可以用来实现第一个预期声明, `java.time.LocalDate` 类可以实现第二个. 但是, 无法直接向这些类型添加 `actual` 关键字.

相反, 你可以使用 类型别名 (类型别名) 来连接预期声明和平台相关的类型:

```
actual typealias Month = java.time.Month
actual typealias MyDate = java.time.LocalDate
```

这种情况下, 请在与预期声明相同的包中定义 `typealias` 声明, 而在其它地方创建被引用的类.

> 由于 `LocalDate` 类型使用了 `Month` 枚举, 你需要在共通代码中将它们都声明为预期类.

在实际声明中扩大可见度

你可以让实际实现的可见度超过对应的预期声明. 如果你不想将你的 API 公开给一般用户, 这个功能会非常有用.

目前, Kotlin 对于可见度改变的情况会提示错误. 你可以压制这个错误, 方法是向实际类型的别名声明标注 `@Suppress("ACTUAL_WITHOUT_EXPECT")` 注解. 从 Kotlin 2.0 开始, 不会再有这个限制.

例如, 如果你在共通源代码集中声明下面的预期声明:

```
internal expect class Messenger {
    fun sendMessage(message: String)
}
```

你也可以在平台相关的源代码集中使用下面的实际实现:

```
@Suppress("ACTUAL_WITHOUT_EXPECT")
public actual typealias Messenger = MyMessenger
```

这里, 预期类的可见度为 internal, 通过类型别名, 它的实际实现是既有的 `MyMessenger` 类, 可见度为 public.

在实际声明中增加枚举值

如果在共通源代码集中使用 `expect` 声明了一个枚举类, 每个平台模块都应该有一个对应的 `actual` 声明. 这些声明必须包含相同的枚举值常数, 但也可以包含额外的枚举值常数.

如果你使用既有的平台枚举类来实现预期的枚举类时, 这个功能会非常有用. 例如, 考虑共通源代码集中的以下枚举类:

```
// 在 commonMain 源代码集中:
expect enum class Department { IT, HR, Sales }
```

当你在平台源代码集中为 `Department` 提供实际声明时, 你可以添加额外的枚举值常数:

```
// 在 jvmMain 源代码集中:
actual enum class Department { IT, HR, Sales, Legal }
```

```
// 在 nativeMain 源代码集中:
actual enum class Department { IT, HR, Sales, Marketing }
```

但是, 对于这样的情况, 平台源代码集中的这些额外的枚举值常数与共通代码中的枚举值常数不能匹配. 因此, 编译器要求你处理所有的其他情况.

实现 `Department` 上的 `when` 构造的函数, 需要 `else` 分支:

```
// 需要 else 分支:
fun matchOnDepartment(dept: Department) {
    when (dept) {
        Department.IT -> println("The IT Department")
        Department.HR -> println("The HR Department")
        Department.Sales -> println("The Sales Department")
        else -> println("Some other department")
    }
}
```

预期注解类

预期声明和实际声明可以与注解一起使用. 例如, 你可以声明一个 `@XmlSerializable` 注解, 它在每个平台源代码集中需要存在对应的实际声明:

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
expect annotation class XmlSerializable()

@XmlSerializable
class Person(val name: String, val age: Int)
```

对于重用特定平台上的既有类型, 这个功能可能很有用. 例如, 在 JVM 上, 你可以使用 JAXB 标准 (https://javaee.github.io/jaxb-v2/) 中的既有类型来定义你的注解:

```
import javax.xml.bind.annotation.XmlRootElement

actual typealias XmlSerializable = XmlRootElement
```

将 `expect` 与注解类一起使用时, 还有一个额外的因素需要考虑. 注解用来向代码添加元数据, 并且它不会成为签名中的类型. 在没有使用这个注解的平台上, 预期注解不一定需要拥有实际类.

你只需要在使用注解的平台上提供 `actual` 声明. 这个行为默认不启用, 而且它要求对类型标注 `OptionalExpectation` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-optional-expectation/) 注解.

对上面声明的 `@XmlSerializable` 注解, 添加 `OptionalExpectation`:

```
@OptIn(ExperimentalMultiplatform::class)
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
@OptionalExpectation
expect annotation class XmlSerializable()
```

在没有使用这个注解的平台上, 如果缺少实际声明, 编译器不会产生错误.

下一步做什么?

关于使用平台相关 API 的各种方法的一般性建议, 请参见 使用平台相关的 API (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-connect-to-apis.html).

层级项目结构

最终更新: 2024/09/10

Kotlin Multiplatform 项目支持层级的源代码集结构. 也就是说, 你可以安排中间源代码集的层级结构, 用于在部分的, 但不是全部的 支持的编译目标 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档"), 之间共用共通的代码. 使用中间源代码集可以帮助你:

- 针对一部分编译目标, 提供特定的 API. 例如, 一个库可以在一个中间源代码集中添加原生代码相关的 API, 用于 Kotlin/Native 编译目标, 但不用于 Kotlin/JVM 编译目标.
- 针对一部分编译目标, 使用特定的 API. 例如, 你可以利用 Kotlin Multiplatform 库为某些编译目标提供的丰富的 API, 这些编译目标组成一个中间源代码集.
- 在你的项目中使用依赖于平台的库. 例如, 你可以通过 iOS 中间源代码集访问 iOS 相关的依赖项.

Kotlin 工具链会确保, 每个源代码集只能访问这个源代码集编译到的所有编译目标都可以使用的 API. 这样可以防止一些错误情况, 例如使用了 Windows 专用的 API, 然后将其编译到 macOS, 这样的情况会在运行期导致链接错误或未定义的行为.

要设置源代码集层级结构, 推荐的方法是使用 默认的层级结构模板. 模板会覆盖最常见的情况. 如果你有更加复杂的项目, 你可以 手动配置. 这是一种更加底层的方案: 它更加灵活, 但需要更多的努力和更多的知识.

默认层级结构模板

从 Kotlin 1.9.20 开始, Kotlin Gradle plugin 包含内建的默认 层级结构模板. 对于一些常见的情况, 模板包含了预定义的中间源代码集. Plugin 会根据你项目中指定的编译目标, 自动设置这些源代码集.

考虑下面的示例:

Kotlin

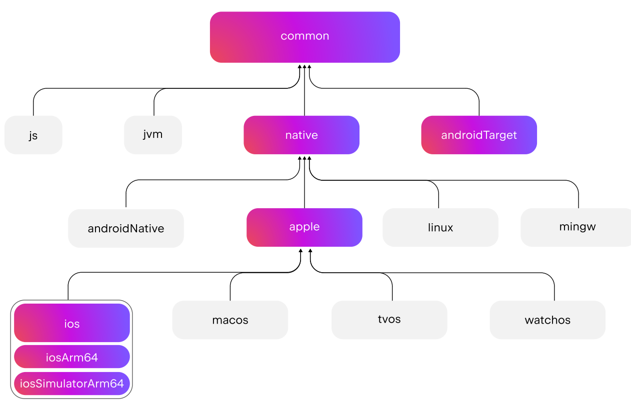
```
kotlin {
    androidTarget()
    iosArm64()
```

```
    iosSimulatorArm64()
}
```

Groovy

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()
}
```

当你在你的代码中声明编译目标 `androidTarget`, `iosArm64`, 和 `iosSimulatorArm64` 时, Kotlin Gradle plugin 会从模板中找到合适的共享源代码集, 并为你创建这些源代码集. 最后产生的层级结构类似下图:



绿色的源代码集会自动创建并包含到项目中, 同时, 默认模板中的灰色的源代码集会被忽略. Kotlin Gradle plugin 没有创建一些源代码集, 例如 `watchos`, 因为项目中没有 watchOS 编译目标.

如果你添加一个 watchOS 编译目标, 例如 `watchosArm64`, `watchos` 源代码集就会被创建, 来自 `apple`, `native`, 和 `common` 源代码集的代码也会被编译到 `watchosArm64`.

Kotlin Gradle plugin 会为来自默认层级结构模板的所有源代码集创建类型安全的访问器, 因此, 与 手动配置 相比, 你可以引用这些源代码集, 不需要使用 `by getting` 或 `by creating` 构建器:

Kotlin

```
kotlin {
    androidTarget()
    iosArm64()
```

```
    iosSimulatorArm64()

    sourceSets {
        iosMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3")
        }
    }
}
```

Groovy

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()

    sourceSets {
        iosMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3'
            }
        }
    }
}
```

> 在这个示例中, apple 和 native 源代码集只编译到 iosArm64 和 iosSimulatorArm64 编译目标. 因此, 尽管它们的名字不是 ios, 它们可以访问完整的 iOS API. 对于 native 这样的源代码集, 这可能会违反直觉, 因为你可能会期望在这个源代码集中, 只能访问那些所有原生编译目标都能够使用的 API. 这个行为未来可能会变更.

附加配置

你可能会需要对默认层级结构模板进行一些调整. 如果你曾经使用 dependsOn 调用, 手动 引入了

中间源代码集, 这样会取消默认层级结构模板的使用, 导致下面的警告:

```
The Default Kotlin Hierarchy Template was not applied to '<project-
name>':
Explicit .dependsOn() edges were configured for the following source
sets:
[<... names of the source sets with manually configured dependsOn-
edges...>]

Consider removing dependsOn-calls or disabling the default template
by adding
    'kotlin.mpp.applyDefaultHierarchyTemplate=false'
to your gradle.properties

Learn more about hierarchy templates: https://kotl.in/hierarchy-
template
```

要解决这个问题, 请通过以下任何一种方式来配置你的项目:

- 用默认层级结构模板替换你的手动配置
- 在默认层级结构模板中创建额外的源代码集
- 修改默认层级结构模板创建的源代码集

替换手动配置

问题场景. 你所有的中间源代码集都被默认层级结构模板覆盖.

解决方案. 删除所有的手动 `dependsOn()` 调用和使用 `by creating` 构建器的源代码集. 关于所有默认源代码集的列表, 请参见 完整的层级结构模板.

创建额外的源代码集

问题场景. 你想要添加默认层级结构模板没有提供的源代码集, 例如, macOS 和 JVM 编译目标之间的一个中间源代码集.

解决方案:

1. 明确调用 `applyDefaultHierarchyTemplate()`, 重新适用模板.
2. 使用 `dependsOn()`, 手动 配置额外的源代码集:

Kotlin

```kotlin
kotlin {
    jvm()
    macosArm64()
    iosArm64()
    iosSimulatorArm64()

    // 再次适用默认层级结构. 它会创建源代码集, 例如 iosMain:
    applyDefaultHierarchyTemplate()

    sourceSets {
        // 创建额外的 jvmAndMacos 源代码集:
        val jvmAndMacos by creating {
            dependsOn(commonMain.get())
        }

        macosArm64Main.get().dependsOn(jvmAndMacos)
        jvmMain.get().dependsOn(jvmAndMacos)
    }
}
```

Groovy

```groovy
kotlin {
    jvm()
    macosArm64()
    iosArm64()
    iosSimulatorArm64()

    // 再次适用默认层级结构. 它会创建源代码集, 例如 iosMain:
    applyDefaultHierarchyTemplate()

    sourceSets {
        // 创建额外的 jvmAndMacos 源代码集:
        jvmAndMacos {
```

```
                dependsOn(commonMain.get())
            }
            macosArm64Main {
                dependsOn(jvmAndMacos.get())
            }
            jvmMain {
                dependsOn(jvmAndMacos.get())
            }
        }
}
```

修改源代码集

问题场景. 你已经有了源代码集, 名字与模板生成的源代码集完全相同, 但在你的项目中的一些不同的编译目标之间共用. 例如, 一个 `nativeMain` 源代码集, 只在桌面专用的编译目标之间共用: `linuxX64`, `mingwX64`, 和 `macosX64`.

解决方案. 目前没有办法修改模板的源代码集之间的默认的 `dependsOn` 关系. 同样重要的是, 源代码集的实现和含义, 例如, `nativeMain`, 在所有的项目中应该保持一致.

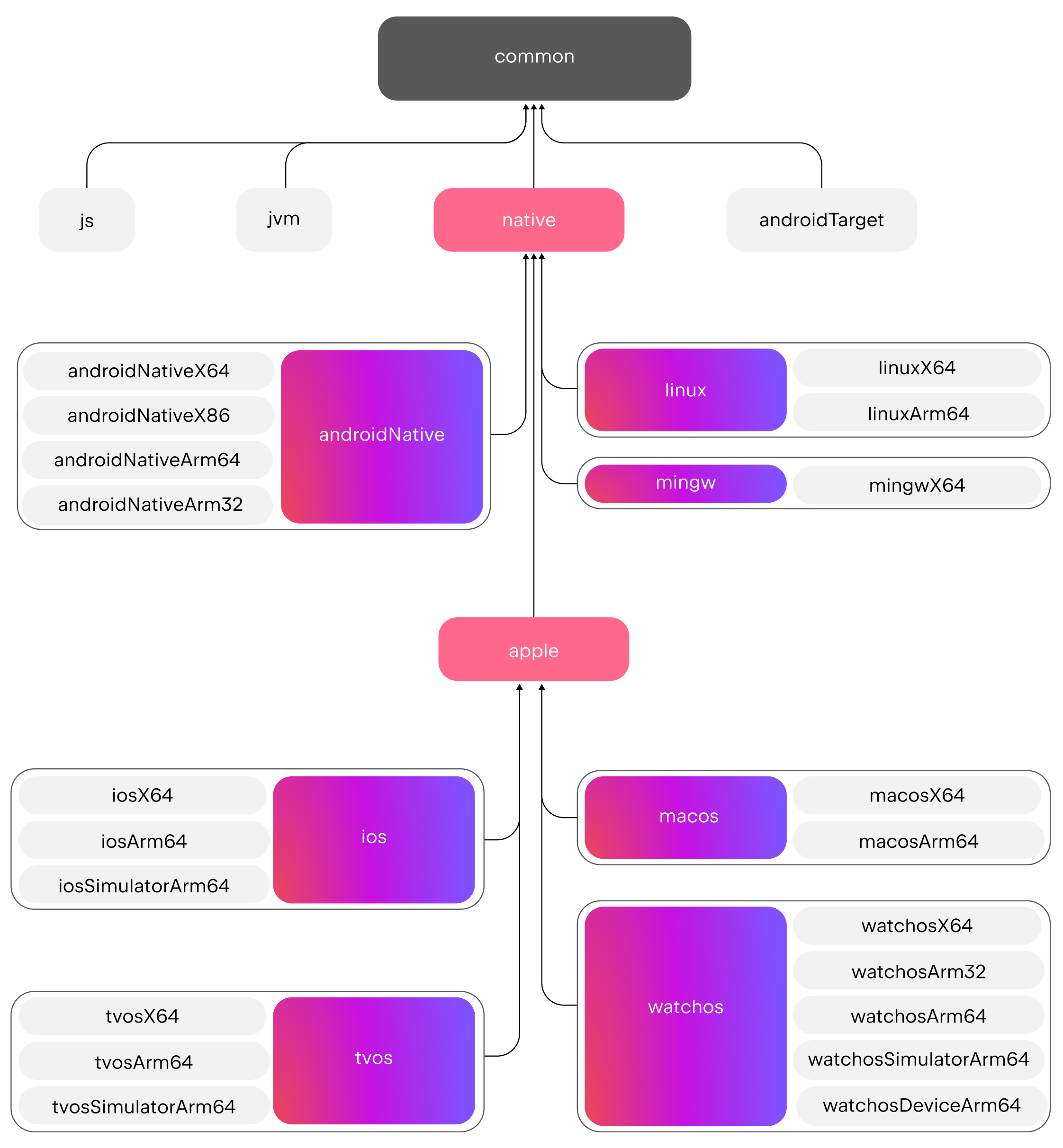
但是, 你还是可以执行下面的任何一种操作:

- 找到适合你的目的的另一个源代码集, 无论是默认层级结构模板中的, 还是手动创建的.
- 向你的 `gradle.properties` 文件添加 `kotlin.mpp.applyDefaultHierarchyTemplate=false`, 完全的选择性禁用(opt out)模板, 并手动配置所有的源代码集.

> ⚠ 我们正在开发一个 API, 用来创建你自己的层级结构模板. 对于层级结构配置与默认模板非常不同的项目, 这个功能会非常有用.
>
> 这个 API 还未完成, 但如果你迫切希望试用它, 可以查看 `applyHierarchyTemplate {}` 代码块, 以及 `KotlinHierarchyTemplate.default` 的声明作为示例. 请记住, 这个 API 还在开发中. 它没有经过足够的测试, 在未来的发布版中可能发生变更.

查看完整的层级结构模板

当你声明你的项目的编译目标时, plugin 会根据指定的编译目标, 从模板中选择共用的源代码集, 并在你的项目中创建这些源代码集.



默认的层级结构模板

> ⚠ 这个示例只显示了项目的 production 部分, 省略了 `Main` 后缀 (例如, 使用 `common` 而不是 `commonMain`). 但是, 还有完全相同的一组 `*Test` 源代码集.

手动配置

你可以在源代码集结构中手动的引入中间源代码集. 它包含多个编译目标之间的共用代码.

例如, 如果你想要在 Linux 原生环境, Windows, 和 macOS 编译目标 (`linuxX64`, `mingwX64`, 和 `macosX64`) 之间共用代码, 你可以这样做:

1. 添加中间源代码集 `desktopMain`, 包含用于这些编译目标的共用逻辑.
2. 使用 `dependsOn` 关系, 指定源代码集的层级结构.

Kotlin

```
kotlin {
    linuxX64()
    mingwX64()
    macosX64()

    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain.get())
        }

        linuxX64Main.get().dependsOn(desktopMain)
        mingwX64Main.get().dependsOn(desktopMain)
        macosX64Main.get().dependsOn(desktopMain)
    }
}
```

Groovy

```
kotlin {
    linuxX64()
    mingwX64()
    macosX64()

    sourceSets {
        desktopMain {
            dependsOn(commonMain.get())
        }
        linuxX64Main {
```

```
                dependsOn(desktopMain)
            }
            mingwX64Main {
                dependsOn(desktopMain)
            }
            macosX64Main {
                dependsOn(desktopMain)
            }
        }
}
```

最后产生的层级结构类似下图:

对以下编译目标组合, 可以共用源代码集:

- JVM 或 Android + JS + Native
- JVM 或 Android + Native
- JS + Native
- JVM 或 Android + JS
- Native

对以下编译目标组合, Kotlin 目前不支持共用源代码集:

- 多个 JVM 编译目标
- JVM + Android 编译目标
- 多个 JS 编译目标

如果你需要在共用的原生源代码集中访问平台相关的 API, IntelliJ IDEA 可以帮助你查找在共用的原生代码中可以使用的共通声明. 其他情况下, 请使用 Kotlin 的 预期声明与实际声明 (预期声明与实际声明) 机制.

添加跨平台库依赖项

最终更新: 2024/09/10

每个应用程序都需要一组库才能正常工作. 一个 Kotlin Multiplatform 项目可以依赖于可以在所有平台工作的跨平台库, 平台相关的库, 还可以依赖于其他跨平台项目.

要在一个库中添加依赖项, 需要更新你的项目包含共用代码的目录中的 build.gradle(.kts) 文件. 在 dependencies ("依赖项目" in "跨平台程序的 Gradle DSL 参考文档") 代码段内, 设置必要 类型 ("依赖项的类型" in "配置 Gradle 项目") 的依赖项 (比如, implementation):

Kotlin

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            implementation("com.example:my-library:1.0") // 对所
有源代码集共用的库
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'com.example:my-library:1.0'
            }
        }
    }
}
```

或者, 你也可以 在最顶层设置依赖项 ("在最顶层设置依赖项" in "配置 Gradle 项目").

对 Kotlin 库的依赖项

标准库

对每个源代码集(Source Set), 会自动添加对标准库 (`stdlib`) 的依赖项. 标准库的版本与 `kotlin-multiplatform` 版本相同.

对于与平台相关的源代码集, 会使用针对这个平台的标准库, 同时, 对其他源代码集会添加共通的标准库. Kotlin Gradle plugin 会根据你的 Gradle 构建脚本的 `compilerOptions.jvmTarget` 编译器选项 ([Kotlin Gradle plugin 中的编译器选项](#)) 设置, 选择适当的 JVM 标准库.

详情请参见 如何改变默认设置 (["对标准库的依赖项" in "配置 Gradle 项目"](#)).

测试库

对于跨平台的测试, 可以使用 `kotlin.test` (https://kotlinlang.org/api/latest/kotlin.test/) API. 当你创建跨平台项目时, 你可以在 `commonTest` 中使用一个依赖项, 对所有的源代码集添加测试依赖项:

Kotlin

```
kotlin {
    sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test")) // 会自动引入所有的平台依赖项
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // 会自动引入所有的平台依赖项
            }
```

```
        }
    }
}
```

kotlinx 库

如果使用跨平台的库, 并且需要 依赖共用代码, 只需要在共用源代码集中一次性设置依赖项. 请使用库的基本 artifact 名(base artifact name) – 比如 `kotlinx-coroutines-core`.

Kotlin

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3")
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3'
            }
        }
    }
}
```

如果使用 kotlinx 库, 并且需要 与平台相关的依赖项, 那么可以通过 `-jvm` 或 `-js` 之类的后缀, 来指定与平台相关的库版本, 比如, `kotlinx-coroutines-core-jvm`.

Kotlin

```kotlin
kotlin {
    sourceSets {
        jvmMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core-jvm:1.7.3")
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        jvmMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-
coroutines-core-jvm:1.7.3'
            }
        }
    }
}
```

对 Kotlin 跨平台库的依赖项

对于使用了 Kotlin Multiplatform 技术的库, 比如 SQLDelight (https://github.com/cashapp/sqldelight), 你可以将它添加为依赖项. 关于在你的项目中如何添加这些依赖项, 这些库的作者通常会提供指南.

参见 由社区维护的 Kotlin Multiplatform 库列表 (https://libs.kmp.icerock.dev/).

对所有源代码集共用的库

如果你想要在所有的源代码集中使用一个库, 你可以只在共通源代码集中添加它. Kotlin Multiplatform Mobile plugin 会对所有其他源代码集自动添加对应的依赖项.

> ⚠ 在共通源代码集中, 不可以设置对平台相关库的依赖项.

Kotlin

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            implementation("io.ktor:ktor-client-core:2.3.5")
        }
        androidMain.dependencies {
            // 对 ktor-client 库的平台相关部分的依赖项, 会自动添加
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'io.ktor:ktor-client-core:2.3.5'
            }
        }
        androidMain {
            dependencies {
                // 对 ktor-client 库的平台相关部分的依赖项, 会自动添加
            }
        }
    }
}
```

在特定源代码集中使用的库

如果你只想对特定的源代码集使用一个跨平台库, 你可以只在这些源代码集中添加它的依赖项. 特定库中的声明, 将只能在这些源代码集中使用.

> 这种情况下请使用库的通用名称, 而不要使用平台相关的名称, 比如对下面示例中的 SQLDelight, 请使用 `native-driver`, 而不要使用 `native-driver-iosx64`. 请到库的文档中查找确切的名称.

Kotlin

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            // kotlinx.coroutines 可以在所有源代码集中使用
            implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3")
        }
        androidMain.dependencies {

        }
        iosMain.dependencies {
            // SQLDelight 只在 iOS 源代码集中可以使用, 但在 Android
源代码集或共通源代码集不可使用
            implementation("com.squareup.sqldelight:native-
driver:2.0.0")
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                // kotlinx.coroutines 可以在所有源代码集中使用
                implementation 'org.jetbrains.kotlinx:kotlinx-
```

```
coroutines-core:1.7.3'
            }
        }
        androidMain {
            dependencies {}
        }
        iosMain {
            dependencies {
                // SQLDelight 只在 iOS 源代码集中可以使用, 但在
Android 源代码集或共通源代码集不可使用
                implementation 'com.squareup.sqldelight:native-
driver:2.0.0'
            }
        }
    }
}
```

对其他跨平台项目的依赖项

你可以将一个跨平台项目作为另一个项目的依赖项. 要实现这个目的, 只需要简单的向需要的源代码集添加一个项目依赖项. 如果你想要在所有源代码集中使用一个依赖项, 请将它添加到共通源代码集. 这种情况下, 其他源代码集将会自动得到对应的版本.

Kotlin

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            implementation(project(":some-other-multiplatform-
module"))
        }
        androidMain.dependencies {
            // :some-other-multiplatform-module 的平台相关部分, 会
自动添加
        }
```

```
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation project(':some-other-
multiplatform-module')
            }
        }
        androidMain {
            dependencies {
                // :some-other-multiplatform-module 的平台相关部分，
会自动添加
            }
        }
    }
}
```

下一步做什么?

查看跨平台项目中添加依赖项的其他资料，并学习以下内容:

- 添加 Android 依赖项 (添加 Android 依赖项)
- 添加 iOS 依赖项 (添加 iOS 依赖项)

添加 Android 依赖项

最终更新: 2024/09/10

向一个 Kotlin Multiplatform 模块添加 Android 专有依赖项的流程, 与纯 Android 项目是相同的: 在你的 Gradle 文件中声明依赖项, 然后导入项目. 然后在你的 Kotlin 代码中就可以使用这个依赖项了.

在 Kotlin Multiplatform 项目中声明 Android 依赖项时, 我们建议将它们添加到一个专门的 Android 源代码集. 为此, 请更新你的项目的 `shared` 目录中的 `build.gradle(.kts)` 文件:

Kotlin

```
sourceSets["androidMain"].dependencies {
    implementation("com.example.android:app-magic:12.3")
}
```

Groovy

```
sourceSets {
    androidMain {
        dependencies {
            implementation 'com.example.android:app-magic:12.3'
        }
    }
}
```

将 Android 项目中的一个顶层依赖项, 移动到 Multiplatform 项目中的一个专门的源代码集, 如果这个顶层依赖项使用了 non-trivial 的配置名称, 可能会很困难. 比如, 要从 Android 项目的顶层, 移动 `debugImplementation` 依赖项, 你需要向源代码集添加一个 implementation 依赖项, 名为 `androidDebug`. 在迁移过程中, 为了减少解决这类问题需要做的工作, 你可以在 `android` 代码块中添加一个 `dependencies` 代码块:

Kotlin

```
android {
    //...
    dependencies {
        implementation("com.example.android:app-magic:12.3")
    }
}
```

Groovy

```
android {
    //...
    dependencies {
        implementation 'com.example.android:app-magic:12.3'
    }
}
```

这里声明的依赖项将会与顶层代码块中的依赖项完全相同的处理, 但用这种方式声明可以在你的构建脚本中明确的分离 Android 依赖项, 使代码更容易理解.

将依赖项放在构建脚本末尾的一个单独的 `dependencies` 代码块之内, 这种方式是 Android 项目的习惯写法, 这样的写法也是支持的. 然而, 我们强烈 **不推荐** 这样的做法, 因为构建脚本在顶层代码块中配置 Android 依赖项, 又在各个源代码集中配置其他编译目标依赖项, 这样的写法很容易让人难以理解.

下一步做什么?

查看跨平台项目中添加依赖项的其他资料, 并学习以下内容:

- 关于添加依赖项的 Android 官方文档 (https://developer.android.com/studio/build/dependencies)
- 添加对跨平台库或其他跨平台项目的依赖项 (添加跨平台库依赖项)
- 添加 iOS 依赖项 (添加 iOS 依赖项)

添加 iOS 依赖项

最终更新: 2024/09/10

在 Kotlin Multiplatform 项目中, Apple SDK 依赖项(比如 Foundation 或 Core Bluetooth) 可以作为一组预构建的库来使用. 不需要额外的配置.

你也可以在你的 iOS 源代码集中重用 iOS 生态系统中的其它库和框架. Kotlin 支持与 Objective-C 依赖项交互, 也支持 Swift 依赖项, 但要求它们的 API 使用 `@objc` 属性导出到 Objective-C. 纯 Swift 的依赖项目前还不支持.

也支持与 CocoaPods 依赖项管理器的集成, 但有相同的限制 – 你不能使用纯 Swift 的 pod.

我们推荐在 Kotlin Multiplatform 项目中 使用 CocoaPods 来管理 iOS 依赖项. 如果你想要精密调节交互过程细节, 或者有某些很重要的原因, 只有这些情况才需要 手动管理依赖项.

使用 CocoaPods

1. 执行 CocoaPods 集成的初始设置 ("设置 CocoaPods 环境" in "CocoaPods 概述与设置").

2. 在你的项目的 `build.gradle(.kts)` 文件中加入 `pod()` 函数调用, 添加 CocoaPods 仓库中的你想要使用的 Pod 库的依赖项.

   Kotlin

   ```
   kotlin {
       cocoapods {
           //..
           pod("FirebaseAuth") {
               version = "10.16.0"
           }
       }
   }
   ```

   Groovy

   ```
   kotlin {
       cocoapods {
   ```

```
            //..
            pod('FirebaseAuth') {
                version = '10.16.0'
            }
        }
    }
```

你可以通过以下方式添加 Pod 库依赖项:

- 使用 CocoaPods 仓库 ("从 CocoaPods 仓库添加 Pod 库依赖项" in "添加 Pod 库依赖项")
- 使用本地存储的库 ("使用保存在本地的 Pod 库添加依赖项" in "添加 Pod 库依赖项")
- 使用自定义的 Git 仓库 ("从自定义的 Git 仓库添加 Pod 库依赖项" in "添加 Pod 库依赖项")
- 使用自定义的 Podspec 仓库 ("从自定义 Podspec 仓库添加 Pod 库依赖项" in "添加 Pod 库依赖项")
- 使用自定义的 cinterop 选项 ("使用自定义 cinterop 选项添加 Pod 库依赖项" in "添加 Pod 库依赖项")

3. 重新导入项目.

要在你的 Kotlin 代码中使用依赖项, 请导入包 `cocoapods.<library-name>`. 在上面的示例中中, 应该是:

```
import cocoapods.FirebaseAuth.*
```

不使用 CocoaPods

如果你不想使用 CocoaPods, 你可以使用 cinterop 工具来为 Objective-C 或 Swift 声明创建 Kotlin 绑定. 然后就可以从 Kotlin 代码调用它们.

对于 库 和 框架 的步骤略有不同, 但大致思想是一样的.

1. 下载你的依赖项.
2. 构建它, 得到它的二进制文件.
3. 创建一个专用的 `.def` 文件, 为 cinterop 描述这个依赖项.
4. 调节你的构建脚本, 在构建过程中生成绑定.

不使用 CocoaPods, 添加一个库

1. 下载库的源代码, 放在从你的项目可以引用的某个地方.
2. 构建库 (库作者通常会提供文档说明具体方法), 得到二进制文件路径.
3. 在你的项目中, 创建一个 `.def` 文件, 比如 `DateTools.def`.
4. 向这个文件添加第 1 行内容: `language = Objective-C`. 如果你想要使用一个纯 C 的依赖项, 请省略 language 属性.
5. 为这 2 个必须属性指定值:
   - `headers` 描述哪些头文件要由 cinterop 处理.
   - `package` 设置这些声明应该放置的包名称.

   比如:

   ```
   headers = DateTools.h
   package = DateTools
   ```

6. 向构建脚本添加与这个库交互的信息:
   - 传递 `.def` 文件的路径. 如果你的 `.def` 文件与 cinterop 名称相同, 并放置在 `src/nativeInterop/cinterop/` 目录中, 那么这个路径可以省略.
   - 使用 `includeDirs` 选项, 告诉 cinterop 到哪里寻找头文件.
   - 配置如何链接到库的二进制文件.

   Kotlin

   ```
   kotlin {
       iosX64() {
           compilations.getByName("main") {
               val DateTools by cinterops.creating {
                   // .def 文件路径

   defFile("src/nativeInterop/cinterop/DateTools.def")
   ```

```
                // 头文件查找目录 (类似于 -I<path> 编译器选项)
                includeDirs("include/this/directory",
"path/to/another/directory")
            }
            val anotherInterop by cinterops.creating { /* ...
*/ }
        }

        binaries.all {
            // 链接到库需要的链接器选项.
            linkerOpts("-L/path/to/library/binaries", "-
lbinaryname")
        }
    }
}
```

Groovy

```
kotlin {
    iosX64 {
        compilations.main {
            cinterops {
                DateTools {
                    // .def 文件路径

defFile("src/nativeInterop/cinterop/DateTools.def")

                    // 头文件查找目录 (类似于 -I<path> 编译器选项)
                    includeDirs("include/this/directory",
"path/to/another/directory")
                }
                anotherInterop { /* ... */ }
            }
        }

        binaries.all {
            // 链接到库需要的链接器选项.
```

```
            linkerOpts "-L/path/to/library/binaries", "-lbinaryname"
        }
    }
}
```

7. 构建项目.

现在你可以在你的 Kotlin 代码中使用这个依赖项了. 方法是, 导入你在 `.def` 文件的 `package` 属性中设置的那个包. 对于上面的示例, 应该是:

```
import DateTools.*
```

不使用 CocoaPods, 添加一个框架

1. 下载框架源代码, 放在从你的项目可以引用的某个地方.
2. 构建框架 (框架作者通常会提供文档说明具体方法), 得到二进制文件路径.
3. 在你的项目中, 创建一个 `.def` 文件, 比如 `MyFramework.def`.
4. 向这个文件添加第 1 行内容: `language = Objective-C`. 如果你想要使用一个纯 C 的依赖项, 请省略 language 属性.
5. 为这 2 个必须属性指定值:
   - `modules` – 需要由 cinterop 处理的框架名称.
   - `package` – 这些声明应该放置的包名称.

   比如:

   ```
   modules = MyFramework
   package = MyFramework
   ```

6. 向构建脚本添加与这个框架交互的信息:
   - 传递 .def 文件路径. 如果你的 `.def` 文件与 cinterop 名称相同, 并放置在 `src/nativeInterop/cinterop/` 目录中, 那么这个路径可以省略.

- 使用 `-framework` 选项, 向编译器和链接器传递框架名称. 使用 `-F` 选项, 向编译器和链接器传递框架源代码和二进制文件的路径.

Kotlin

```
kotlin {
    iosX64() {
        compilations.getByName("main") {
            val DateTools by cinterops.creating {
                // .def 文件路径

defFile("src/nativeInterop/cinterop/DateTools.def")

                compilerOpts("-framework", "MyFramework", "-
F/path/to/framework/")
            }
            val anotherInterop by cinterops.creating { /* ...
*/ }
        }

        binaries.all {
            // 告诉链接器框架的位置.
            linkerOpts("-framework", "MyFramework", "-
F/path/to/framework/")
        }
    }
}
```

Groovy

```
kotlin {
    iosX64 {
        compilations.main {
            cinterops {
                DateTools {
                    // .def 文件路径
```

```
defFile("src/nativeInterop/cinterop/MyFramework.def")

                    compilerOpts("-framework", "MyFramework",
"-F/path/to/framework/")
                }
                anotherInterop { /* ... */ }
            }
        }

        binaries.all {
            // 告诉链接器框架的位置.
            linkerOpts("-framework", "MyFramework", "-
F/path/to/framework/")
        }
    }
}
```

7. 构建项目.

现在你可以在你的 Kotlin 代码中使用这个依赖项了. 方法是, 导入你在 .def 文件的 package 属性中设置的那个包. 对于上面的示例, 应该是:

```
import MyFramework.*
```

详情请参见 与 Objective-C 和 Swift 交互 (与 Swift/Objective-C 代码交互) 以及 在 Gradle 中配置 cinterop ("CInterops" in "跨平台程序的 Gradle DSL 参考文档").

下一步做什么?

查看跨平台项目中添加依赖项的其他资料, 并学习以下内容:

- 连接到平台相关的库 ("连接平台相关的库" in "在不同的平台之间共用代码")
- 添加对跨平台库或其他跨平台项目的依赖项 (添加跨平台库依赖项)
- 添加 Android 依赖项 (添加 Android 依赖项)

配置编译任务

最终更新: 2024/09/10

Kotlin 跨平台项目使用编译任务来生成 artifact. 每个编译目标可以有一个或多个编译任务, 比如, 用于产品和测试的编译任务.

对每个编译目标, 默认的编译任务包括:

- 对于 JVM, JS, 和 Native 编译目标: `main` 和 `test` 编译任务.
- 对于 Android 编译目标: 每个 Android 构建变体(build variant) (https://developer.android.com/studio/build/build-variants) 一个 编译任务.

如果需要编译除产品代码与单元测试之外的其他代码, 比如, 集成测试, 或性能测试, 你可以 创建自定义编译任务.

你可以在以下范围配置如何产生 artifact:

- 对所有编译任务: 可以在你的项目中一次性设置.
- 对单个编译目标的编译任务: 因为一个编译目标可以有多个编译任务.
- 对一个指定的编译任务.

请参见对所有编译目标或特定编译目标可用的 编译任务参数列表 ("编译任务的参数" in "跨平台程序的 Gradle DSL 参考文档") 和 编译器选项 (Kotlin Gradle plugin 中的编译器选项).

配置所有编译任务

```
kotlin {
    targets.all {
        compilations.all {
            compilerOptions.configure {
                allWarningsAsErrors.set(true)
            }
        }
    }
}
```

配置单个编译目标的编译任务

Kotlin

```
kotlin {
    jvm().compilations.all {
        compilerOptions.configure {
            jvmTarget.set(JvmTarget.JVM_1_8)
        }
    }
}
```

Groovy

```
kotlin {
    jvm().compilations.all {
        compilerOptions.configure {
            jvmTarget.set(JvmTarget.JVM_1_8)
        }
    }
}
```

配置一个编译任务

Kotlin

```
kotlin {
    jvm {
        val main by compilations.getting {
            compilerOptions.configure {
                jvmTarget.set(JvmTarget.JVM_1_8)
            }
        }
```

```
    }
}
```

Groovy

```
kotlin {
    jvm {
        compilations.main {
            compilerOptions.configure {
                jvmTarget.set(JvmTarget.JVM_1_8)
            }
        }
    }
}
```

创建自定义编译任务

如果需要编译除产品代码与单元测试之外的其他代码, 比如, 集成测试, 或性能测试, 请创建自定义编译任务.

比如, 要对 `jvm()` 编译目标的集成测试创建自定义编译任务, 请向 `compilations` 集合内添加新的元素.

> 对于自定义编译任务, 需要手动设置所有的依赖项. 自定义编译任务的默认源代码集不会依赖 `commonMain` 和 `commonTest` 源代码集.

Kotlin

```
kotlin {
    jvm() {
        compilations {
            val main by getting

            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
```

```
                        // 使用 main 编译任务的编译期类路径和输出进行编
译:

implementation(main.compileDependencyFiles +
main.output.classesDirs)
                        implementation(kotlin("test-junit"))
                        /* ... */
                    }
                }

                // 创建 test 任务来运行这个编译任务产生的测试:
                tasks.create<Test>("integrationTest") {
                    // 运行测试使用的类路径包含:
                    // 编译期依赖项(包括 'main'), 运行时依赖项, 以及这
个编译任务的输出:
                    classpath = compileDependencyFiles +
runtimeDependencyFiles + output.allOutputs

                    // 只运行这个编译任务的输出中包含的测试:
                    testClassesDirs = output.classesDirs
                }
            }
        }
    }
}
```

Groovy

```
kotlin {
    jvm() {
        compilations.create('integrationTest') {
            defaultSourceSet {
                dependencies {
                    def main = compilations.main
                    // 使用 main 编译任务的编译期类路径和输出进行编译:
                    implementation(main.compileDependencyFiles +
main.output.classesDirs)
```

```
                    implementation kotlin('test-junit')
                    /* ... */
                }
            }

            // 创建 test 任务来运行这个编译任务产生的测试:
            tasks.create('jvmIntegrationTest', Test) {
                // 运行测试使用的类路径包含:
                // 编译期依赖项(包括 'main'), 运行时依赖项, 以及这个编
译任务的输出:
                classpath = compileDependencyFiles +
runtimeDependencyFiles + output.allOutputs

                // 只运行这个编译任务的输出中包含的测试:
                testClassesDirs = output.classesDirs
            }
        }
    }
}
```

对于其他情况也需要创建自定义编译任务, 比如, 如果希望在你的最终 artifact 中对不同的 JVM 版本组合编译任务, 或者已经在 Gradle 中设置过源代码集, 希望迁移到跨平台项目.

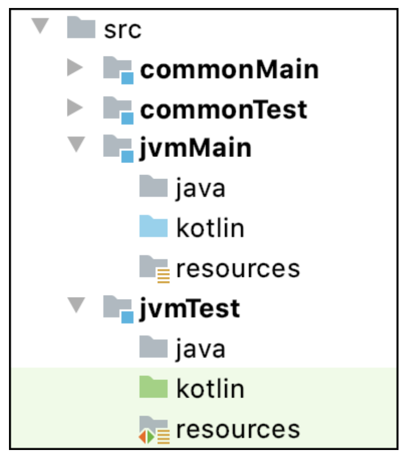
在 JVM 编译任务中使用 Java 源代码

使用 项目向导 (https://kmp.jetbrains.com/) 创建项目时, Java 源代码会包含在 JVM 编译任务内.

在构建脚本中, 以下代码会应用 Gradle `java` plugin, 并配置编译目标, 使其与 `java` plugin 协作:

```
kotlin {
    jvm {
        withJava()
    }
}
```

Java 源代码文件 放在 Kotlin 源代码根路径的子目录之内. 比如, 路径是:



Java 源代码文件

共通源代码集不可以包含 Java 源代码.

由于目前的限制, Kotlin plugin 会替换 Java plugin 配置的某些任务:

- 使用编译目标的 JAR 任务, 而不是 `jar` 任务 (比如, `jvmJar`).
- 使用编译目标的 test 任务, 而不是 `test` 任务 (比如, `jvmTest`).
- 资源由编译任务中的同等任务处理, 而不是 `*ProcessResources` 任务.

这个编译目标的发布由 Kotlin plugin 处理, 而且不需要 Java plugin 的具体步骤.

配置与原生语言的交互

Kotlin 提供 与原生语言的交互能力 (与 C 代码交互), 以及对编译任务进行相关配置的 DSL.

原生语言	支持的平台	备注
C	所有平台, WebAssembly 除外	
Objective-C	Apple 平台 (macOS, iOS, watchOS, tvOS)	
经由 Objective-C 的 Swift	Apple 平台 (macOS, iOS, watchOS, tvOS)	Kotlin 只能使用 `@objc` 属性标注过的 Swift 声明.

编译任务可以与几个原生库交互. 在编译任务的 cinterops 代码段中, 可以使用 可用的参数 ("CInterops" in "跨平台程序的 Gradle DSL 参考文档") 来配置交互能力.

Kotlin

```
kotlin {
    linuxX64 { // 请替换为你需要的编译目标.
        compilations.getByName("main") {
            val myInterop by cinterops.creating {
                // 描述原生 API 的 def 文件.
                // 默认路径是 src/nativeInterop/cinterop/<interop-
name>.def
                defFile(project.file("def-file.def"))

                // 用来放置生成的 Kotlin API 的包.
                packageName("org.sample")

                // 通过 cinterop 工具传递给编译器的参数.
                compilerOpts("-Ipath/to/headers")

                // 用来查找头文件的目录.
                includeDirs.apply {
                    // 用来查找头文件的目录 (等价于编译器的 -I<path>
参数).
                    allHeaders("path1", "path2")

                    // 用来查找 def 文件的 'headerFilter' 参数中指定
的头文件时使用的额外的目录.
                    // 等价于 -headerFilterAdditionalSearchPrefix
命令行参数.
                    headerFilterOnly("path1", "path2")
                }
                // includeDirs.allHeaders 的缩写方式.
                includeDirs("include/directory",
"another/directory")
            }

            val anotherInterop by cinterops.creating { /* ... */
```

```
}
        }
    }
}
```

Groovy

```
kotlin {
    linuxX64 { // 请替换为你需要的编译目标.
        compilations.main {
            cinterops {
                myInterop {
                    // 描述原生 API 的 def 文件.
                    // 默认路径是
src/nativeInterop/cinterop/<interop-name>.def
                    defFile project.file("def-file.def")

                    // 用来放置生成的 Kotlin API 的包.
                    packageName 'org.sample'

                    // 通过 cinterop 工具传递给编译器的参数.
                    compilerOpts '-Ipath/to/headers'

                    // 用来查找头文件的目录 (等价于编译器的 -I<path>
参数).
                    includeDirs.allHeaders("path1", "path2")

                    // 用来查找 def 文件的 'headerFilter' 参数中指定
的头文件时使用的额外的目录.
                    // 等价于 -headerFilterAdditionalSearchPrefix
命令行参数.
                    includeDirs.headerFilterOnly("path1",
"path2")

                    // includeDirs.allHeaders 的缩写方式.
                    includeDirs("include/directory",
"another/directory")
```

```
                }

                anotherInterop { /* ... */ }
            }
        }
    }
}
```

Android 编译任务

对 Android 编译目标默认创建的编译任务会与 Android 构建变体(build variant) (https://developer.android.com/studio/build/build-variants) 绑定: 对每个构建变体, 会创建一个相同名称的 Kotlin 编译任务.

然后, 对每个构建变体编译的每个 Android 源代码集 (https://developer.android.com/studio/build/build-variants#sourcesets), 会创建 Kotlin 源代码集, 名称是 Android 源代码集名前面加上编译目标名, 比如, 对于 Kotlin 编译目标 `android`, 以及 Android 源代码集 `debug`, 对应的 Kotlin 源代码集名为 `androidDebug`. 这些 Kotlin 源代码集会被添加到对应的构建变体的编译任务中.

默认的源代码集 `commonMain` 会被添加到所有的产品构建变体(无论是应用程序还是库) 的编译任务中. 类似的, 源代码集 `commonTest`, 会被添加到单元测试(Unit Test)和设备测试(Instrumented Test)的构建变体中.

也支持使用 kapt (kapt 编译器插件) 的注解处理, 但是由于目前的限制, 要求在配置 `kapt` 依赖项之前创建 Android 编译目标, 因此需要使用顶级的 `dependencies { ... }` 代码段, 而不是使用 Kotlin 源代码集的依赖项配置语法.

```
kotlin {
    android { /* ... */ }
}

dependencies {
    kapt("com.my.annotation:processor:1.0.0")
}
```

源代码集层级结构的编译任务

Kotlin 可以使用 `dependsOn` 关系来创建 源代码集层级结构 ("在类似的平台上共用代码" in "在不同的平台之间共用代码").

如果源代码集 `jvmMain` 依赖源代码集 `commonMain`, 那么:

- 对某个编译目标, 当 `jvmMain` 被编译时, `commonMain` 也会参与这个编译任务, 而且也会被编译称为同一个编译目标的二进制形式, 比如 JVM class 文件.
- `jvmMain` 的源代码可以 '看见' `commonMain` 的声明, 包括内部声明, 也能看见 `commonMain` 的 依赖项 (添加跨平台库依赖项), 即使是标记为 `implementation` 的依赖项.
- `jvmMain` 可以包含 `commonMain` 的 预期声明(expected declaration) (预期声明与实际声明) 的平台相关的实现.
- `commonMain` 的资源会与 `jvmMain` 的资源一起处理, 复制.
- `jvmMain` 和 `commonMain` 的 语言设置 ("语言设置" in "跨平台程序的 Gradle DSL 参考文档") 应该保持一致.

语言设置的一致性检查规则是:

- `jvmMain` 的 `languageVersion` 设置应该高于或等于 `commonMain` 的设置.
- `commonMain` 启用的所有非稳定的语言特性, `jvmMain` 都应该启用 (对于 bugfix 特性没有这个要求).
- `commonMain` 使用的所有实验性注解, `jvmMain` 都应该使用.
- `apiVersion`, bugfix 语言特性, 以及 `progressiveMode` 可以任意设定.

[实验性 DSL] 构建最终的原生二进制文件

最终更新: 2024/09/10

> 本章介绍的新 DSL 是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更. 我们建议你只为评估和试验目的来使用这个功能.
>
> 如果你无法使用新 DSL, 请参见构建原生二进制文件的 前一种方案 (构建最终的原生二进制文件).

Kotlin/Native 编译目标 ("原生(Native)编译目标" in "跨平台程序的 Gradle DSL 参考文档") 会被编译为 `*.klib` 库的 Artifact, 这些库 Artifact 可以被 Kotlin/Native 本身用作依赖项, 但不能用作原生库.

要声明最终的原生二进制文件, 请通过 `kotlinArtifacts` DSL, 使用新的二进制文件格式. 它代表为这个编译目标构建的原生二进制文件的集合, 以及默认的 `*.klib` Artifact, 它还提供了一组方法, 用来声明和配置这些文件.

> `kotlin-multiplatform` plugin 默认不会创建任何生产环境(production)版的二进制文件. 默认生成的二进制文件只有 debug 版的测试用可执行文件, 供你通过 `test` 编译任务来运行单元测试.

Kotlin Artifact DSL 可以帮助你解决一个常见问题: 你需要从你的 App 访问多个 Kotlin 模块. 由于不能使用多个 Kotlin/Native Artifact, 你可以使用新 DSL, 将多个 Kotlin 模块导出到单个 Artifact.

声明二进制文件

`kotlinArtifacts` 元素是 Gradle 构建脚本中用于 Artifact 配置的顶层代码块. 请使用以下二进制文件类型来声明 `kotlinArtifacts` DSL 的元素:

工厂方法	二进制文件类型	可用于
sharedLib	共享的原生库 ("怎样创建一个共享库?" in "Kotlin/Native FAQ")	所有的原生编译目标, WebAssembly 除外
staticLib	静态的原生库 ("怎样创建静态库, 或 object 文件?" in "Kotlin/Native FAQ")	所有的原生编译目标, WebAssembly 除外
framework	Objective-C 框架	只能用于 macOS, iOS, watchOS, 和 tvOS 编译目标
fatFramework	Universal fat 框架	只能用于 macOS, iOS, watchOS, 和 tvOS 编译目标
XCFramework	XCFramework 框架	只能用于 macOS, iOS, watchOS, 和 tvOS 编译目标

在 kotlinArtifacts 元素内, 你可以编写以下代码块:

- Native.Library
- Native.Framework
- Native.FatFramework
- Native.XCFramework

最简单的版本需要对选择的构建类型指定 target (或 targets) 参数. 目前, 可以使用 2 个构建类型:

- DEBUG – 生成一个无优化的二进制文件, 包含调试信息
- RELEASE – 生成优化的二进制文件, 不包含调试信息

在 modes 参数中, 你可以指定你想要为哪个构建类型创建二进制文件. 默认值包括 DEBUG 和 RELEASE 的可执行二进制文件:

Kotlin

```
kotlinArtifacts {
    Native.Library {
        target = iosX64 // 请改为定义你的编译目标
        modes(DEBUG, RELEASE)
        // 二进制文件配置
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.Library {
        target = iosX64 // 请改为定义你的编译目标
        modes(DEBUG, RELEASE)
        // 二进制文件配置
    }
}
```

你也可以使用自定义的名称来声明二进制文件：

Kotlin

```
kotlinArtifacts {
    Native.Library("mylib") {
        // 二进制文件配置
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.Library("mylib") {
        // 二进制文件配置
```

```
    }
}
```

这里的参数是名称前缀, 用作二进制文件的默认名称. 例如, 对于 Windows, 上面的代码会 `mylib.dll` 文件.

配置二进制文件

对于二进制文件配置, 可以使用以下共通参数:

名称	说明
`isStatic`	可选项, 定义库类型的链接类型. 默认值为 `false`, 库为动态库.
`modes`	可选项, 构建类型, 可指定的值是 `DEBUG` 和 `RELEASE`.
`kotlinOptions`	可选项, 编译时使用的编译器选项. 参见可用的 编译器选项 (Kotlin Gradle plugin 中的编译器选项).
`addModule`	除当前模块外, 你还可以向输出的 Artifact 添加其他模块.
`setModules`	你可以覆盖添加到输出的 Artifact 的模块列表.

库和框架

构建 Objective-C 框架或(共享的或静态的)原生库时, 你可能不仅当前项目中的类, 而且还需要将其他跨平台模块中的类也打包到单个库中, 并且将所有的模块都导出到这个库.

库

对于库的配置, 除共通参数外, 还可以使用 `target` 参数:

名称	说明
`target`	指定项目的一个特定的编译目标. 可用的编译目标名称请参见 编译目标 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 小节.

Kotlin

```
kotlinArtifacts {
    Native.Library("myslib") {
        target = linuxX64
        isStatic = false
        modes(DEBUG)
        addModule(project(":lib"))
        kotlinOptions {
            verbose = false
            freeCompilerArgs += "-Xmen=pool"
        }
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.Library("myslib") {
        target = linuxX64
        it.static = false
        modes(DEBUG)
        addModule(project(":lib"))
        kotlinOptions {
            verbose = false
            freeCompilerArgs += "-Xmen=pool"
        }
    }
}
```

以上代码注册的 Gradle 任务是 `assembleMyslibSharedLibrary`, 它会将所有已注册的 "myslib" 类型汇集到一个动态库.

框架

对于框架的配置, 除共通参数外, 还可以使用以下参数:

名称	说明
target	指定项目的一个特定的编译目标. 可用的编译目标名称请参见 编译目标 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 小节.
embed Bitcode	指定字节码的内嵌模式. 可以指定 MARKER 来内嵌字节码标记(用于 debug 构建), 或指定 DISABLE 来关闭字节码内嵌. 对于 Xcode 14 或更高版本, 不需要字节码内嵌.

Kotlin

```
kotlinArtifacts {
    Native.Framework("myframe") {
        modes(DEBUG, RELEASE)
        target = iosArm64
        isStatic = false
        embedBitcode = EmbedBitcodeMode.MARKER
        kotlinOptions {
            verbose = false
        }
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.Framework("myframe") {
        modes(DEBUG, RELEASE)
        target = iosArm64
        it.static = false
        embedBitcode = EmbedBitcodeMode.MARKER
        kotlinOptions {
            verbose = false
        }
```

```
    }
}
```

以上代码注册的 Gradle 任务是 `assembleMyframeFramework`, 它会汇集所有已注册的 "myframe" 框架类型.

如果你因为某些原因无法使用新 DSL, 请试用 前一种方案 (["将依赖项目导出到二进制文件" in "构建最终的原生二进制文件"](#)) 来将依赖项导出到二进制文件.

Fat 框架

默认情况下, 由 Kotlin/Native 生成的 Objective-C 框架只支持一个平台. 但是, 你可以将这样的框架合并为单个通用(fat)二进制文件. 这种做法对 32 位和 64 位 iOS 框架尤其有意义. 对于这样的情况, 生成的通用框架可以同时在 32 位和 64 位设备上使用.

对于 fat 框架的配置, 除共通参数外, 还可以使用以下参数:

名称	说明
`targets`	指定项目的所有编译目标.
`embedBitcode`	指定字节码的内嵌模式. 可以指定 `MARKER` 来内嵌字节码标记(用于 debug 构建), 或指定 `DISABLE` 来关闭字节码内嵌. 对于 Xcode 14 或更高版本, 不需要字节码内嵌.

Kotlin

```kotlin
kotlinArtifacts {
    Native.FatFramework("myfatframe") {
        targets(iosX32, iosX64)
        embedBitcode = EmbedBitcodeMode.DISABLE
        kotlinOptions {
            suppressWarnings = false
        }
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.FatFramework("myfatframe") {
        targets(iosX32, iosX64)
        embedBitcode = EmbedBitcodeMode.DISABLE
        kotlinOptions {
            suppressWarnings = false
        }
    }
}
```

以上代码注册的 Gradle 任务是 `assembleMyfatframeFatFramework`, 它会汇集所有已注册的 "myfatframe" fat 框架类型.

如果你因为某些原因无法使用新 DSL, 请试用 前一种方案 ("构建通用框架(Universal Framework)" in "构建最终的原生二进制文件") 来构建 fat 框架.

XCFramework

所有的 Kotlin Multiplatform 项目都可以使用 XCFramework 作为输出, 将所有目标平台和架构的逻辑集合到单个 bundle 中. 与 通用(fat)框架 不同, 在将应用程序发布到 App Store 之前, 你不需要删除所有不需要的架构.

对于 XCFramework 的配置, 除共通参数外, 还可以使用以下参数:

名称	说明
`targets`	指定项目的所有编译目标.
`embedBitcode`	指定字节码的内嵌模式. 可以指定 `MARKER` 来内嵌字节码标记 (用于 debug 构建), 或指定 `DISABLE` 来关闭字节码内嵌. 对于 Xcode 14 或更高版本, 不需要字节码内嵌.

Kotlin

```
kotlinArtifacts {
    Native.XCFramework("sdk") {
        targets(iosX64, iosArm64, iosSimulatorArm64)
        setModules(
            project(":shared"),
            project(":lib")
        )
    }
}
```

Groovy

```
kotlinArtifacts {
    it.native.XCFramework("sdk") {
        targets(iosX64, iosArm64, iosSimulatorArm64)
        setModules(
            project(":shared"),
            project(":lib")
        )
    }
}
```

以上代码注册的 Gradle 任务是 `assembleSdkXCFramework`, 它会汇集所有已注册的 "sdk" XCFramework 类型.

如果你因为某些原因无法使用新 DSL, 请试用 前一种方案 ("构建 XCFramework" in "构建最终的原生二进制文件") 来构建 XCFramework.

构建最终的原生二进制文件

最终更新: 2024/09/10

Kotlin/Native 编译目标默认会被编译输出为 `*.klib` 库文件, 这种库文件可以被 Kotlin/Native 用作依赖项, 但它不能执行, 也不能被用作一个原生的库.

如果要编译为最终的原生二进制文件, 比如可执行文件, 或共享库, 可以使用原生编译目标的 `binaries` 属性. 这个属性值是原生二进制文件的列表, 表示除默认的 `*.klib` 库文件之外, 这个编译目标还需要编译为哪些类型, 这个属性还提供了一组方法, 用来声明和配置这些原生二进制文件.

> `kotlin-multiplatform` plugin 默认不会创建任何产品版(production)的二进制文件. 默认情况下, 只会产生一个调试版(debug)的测试可执行文件, 你可以通过 `test` 编译任务来运行这个可执行文件内的单元测试.

Kotlin/Native 编译器生成的二进制文件可能包含第三方代码, 数据, 或衍生作品. 也就是说, 如果你发布 Kotlin/Native 编译的最终二进制文件, 那么你始终需要在你的二进制分发版中包含必要的 许可证文件 ([Kotlin/Native 二进制文件的许可证](#)).

声明二进制文件

请使用以下工厂方法来声明 `binaries` 列表中的元素.

工厂方法	二进制文件类型	可用于
`executable`	产品版的可执行文件	所有的原生编译目标
`test`	测试程序的可执行文件	所有的原生编译目标
`sharedLib`	Shared 原生库	所有的原生编译目标, `WebAssembly` 除外
`staticLib`	Static 原生库	所有的原生编译目标, `WebAssembly` 除外
`framework`	Objective-C 框架	仅限于 macOS, iOS, watchOS, 和 tvOS 编译目标

最简单的版本不需要任何额外参数, 并对每一个构建类型创建一个二进制文件. 现在有 2 种构建类

型:

- `DEBUG` – 产生一个未经优化的, 带调试信息的二进制文件
- `RELEASE` – 产生优化过的, 无调试信息的二进制文件

下面的代码会创建 2 个可执行的二进制文件, debug 和 release:

```
kotlin {
    linuxX64 { // 这里请改为你的编译目标.
        binaries {
            executable {
                // 这里指定二进制文件的配置信息.
            }
        }
    }
}
```

如果不需要额外的配置 ("原生(Native)编译目标" in "跨平台程序的 Gradle DSL 参考文档"), 那么可以省略这个 Lambda 表达式:

```
binaries {
    executable()
}
```

还可以指定对哪些构建类型创建二进制文件. 下面的示例只创建 `debug` 版的二进制文件:

Kotlin

```
binaries {
    executable(listOf(DEBUG)) {
        // 这里指定二进制文件的配置信息.
    }
}
```

Groovy

```
binaries {
    executable([DEBUG]) {
        // 这里指定二进制文件的配置信息.
    }
}
```

还可以使用自定义的名称来声明二进制文件:

Kotlin

```
binaries {
    executable("foo", listOf(DEBUG)) {
        // 这里指定二进制文件的配置信息.
    }

    // 可以省略构建类型
    // (这时会使用所有可用的构建类型).
    executable("bar") {
        // 这里指定二进制文件的配置信息.
    }
}
```

Groovy

```
binaries {
    executable('foo', [DEBUG]) {
        // 这里指定二进制文件的配置信息.
    }

    // 可以省略构建类型
    // (这时会使用所有可用的构建类型).
    executable('bar') {
        // 这里指定二进制文件的配置信息.
```

```
    }
}
```

这个示例中的第一个参数指定一个名称前缀, 它会是二进制文件的默认名称. 比如, 在 Windows 平台, 这个示例会输出 `foo.exe` 和 `bar.exe`. 还可以使用这个名称前缀 在构建脚本中访问二进制文件.

访问二进制文件

可以访问二进制文件来 对其进行配置 ("原生(Native)编译目标" in "跨平台程序的 Gradle DSL 参考文档"), 或者得到它们的属性 (比如, 得到输出文件的路径).

可以通过二进制文件的唯一名称来得到它. 这个名称由名称前缀(如果有指定), 构建类型, 以及二进制文件类型组成, 使用以下命名方式: `<optional-name-prefix><build-type><binary-kind>`, 比如, `releaseFramework` 或 `testDebugExecutable`.

> 静态库和共享库分别带有 static 和 shared 后缀, 比如, `fooDebugStatic` 或 `barReleaseShared`.

Kotlin

```
// 如果二进制文件不存在, 这个函数会失败.
binaries["fooDebugExecutable"]
binaries.getByName("fooDebugExecutable")

// 如果二进制文件不存在, 这个函数会返回 null.
binaries.findByName("fooDebugExecutable")
```

Groovy

```
// 如果二进制文件不存在, 这个函数会失败.
binaries['fooDebugExecutable']
binaries.fooDebugExecutable
binaries.getByName('fooDebugExecutable')
```

```
// 如果二进制文件不存在，这个函数会返回 null.
binaries.findByName('fooDebugExecutable')
```

另一种方法是, 可以使用名称前缀和构建类型, 通过有类型的 get 方法访问二进制文件.

Kotlin

```
// 如果二进制文件不存在，这个函数会失败.
binaries.getExecutable("foo", DEBUG)
binaries.getExecutable(DEBUG)          // 如果没有设置名称前缀，可以省略第一个参数.
binaries.getExecutable("bar", "DEBUG") // 对于构建类型，也可以使用字符串.

// 对其他二进制文件类型，可以使用类似的 get 方法:
// getFramework, getStaticLib 以及 getSharedLib.

// 如果二进制文件不存在，这个函数会返回 null.
binaries.findExecutable("foo", DEBUG)

// 对其他二进制文件类型，可以使用类似的 get 方法:
// findFramework, findStaticLib 以及 findSharedLib.
```

Groovy

```
// 如果二进制文件不存在，这个函数会失败.
binaries.getExecutable('foo', DEBUG)
binaries.getExecutable(DEBUG)          // 如果没有设置名称前缀，可以省略第一个参数.
binaries.getExecutable('bar', 'DEBUG') // 对于构建类型，也可以使用字符串.

// 对其他二进制文件类型，可以使用类似的 get 方法:
// getFramework, getStaticLib 以及 getSharedLib.

// 如果二进制文件不存在，这个函数会返回 null.
binaries.findExecutable('foo', DEBUG)
```

```
// 对其他二进制文件类型, 可以使用类似的 get 方法:
// findFramework, findStaticLib 以及 findSharedLib.
```

将依赖项目导出到二进制文件

编译 Objective-C 框架, 或原生库(共享库或静态库)时, 经常会出现一种需要, 不仅要打包当前项目的类文件, 同时还要打包它的依赖项的类. 我们可以用 `export` 方法, 指定需要导出哪些依赖项到二进制文件中.

Kotlin

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // 这些依赖项会被导出.
            api(project(":dependency"))
            api("org.example:exported-library:1.0")
            // 这个依赖项不会被导出.
            api("org.example:not-exported-library:1.0")
        }
    }
    macosX64("macos").binaries {
        framework {
            export(project(":dependency"))
            export("org.example:exported-library:1.0")
        }
        sharedLib {
            // 可以对不同的二进制文件导出不同的依赖项目.
            export(project(':dependency'))
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // 这些依赖项会被导出.
            api project(':dependency')
            api 'org.example:exported-library:1.0'
            // 这个依赖项不会被导出.
            api 'org.example:not-exported-library:1.0'
        }
    }
    macosX64("macos").binaries {
        framework {
            export project(':dependency')
            export 'org.example:exported-library:1.0'
        }
        sharedLib {
            // 可以对不同的二进制文件导出不同的依赖项目.
            export project(':dependency')
        }
    }
}
```

比如, 你用 Kotlin 实现了几个模块, 并且想要在 Swift 中访问这些模块. 在一个 Swift 应用程序中无法使用多个 Kotlin/Native 框架, 但你可以创建一个 umbrella 框架, 把所有这些模块都导出到这个框架.

> 只能导出对应的源代码集的 `api` 依赖项 (["依赖项的类型" in "配置 Gradle 项目"]()).

当你导出一个依赖项, 它的所有 API 到会包含框架 API 中. 编译器会向框架添加这个依赖项的代码, 即使你只使用了它的一小部分. 这就使得对导出的依赖项 (以及某种程度上对它的依赖项) 死代码消除功能不再有效.

默认情况下, 导出是非传递性的(non-transitively). 也就是说, 如果你导出的库 `foo` 依赖于库 `bar`, 只有 `foo` 中的方法会被添加到输出的框架中.

这种行为可以通过 `transitiveExport` 选项来修改. 如果设置为 `true`, 库 `bar` 中的声明也会被导出.

> 不推荐使用 transitiveExport: 它会将导出的依赖项的所有传递依赖项添加到框架. 这会增加编译时间, 并增大输出的二进制文件大小.
>
> 大多数情况下, 你不需要将所有这些依赖项添加到 framework API. 应该只对你需要在 Swift 或 Objective-C 代码中直接访问的依赖项, 明确使用 export.

Kotlin

```
binaries {
    framework {
        export(project(":dependency"))
        // 传递性导出.
        transitiveExport = true
    }
}
```

Groovy

```
binaries {
    framework {
        export project(':dependency')
        // 传递性导出.
        transitiveExport = true
    }
}
```

构建通用框架(Universal Framework)

默认情况下, Kotlin/Native 编译产生的 Objective-C 框架只支持单个平台. 但是, 使用 lipo 工具程序 (https://llvm.org/docs/CommandGuide/llvm-lipo.html), 可以将多个框架合并为单个通用的 (fat) 二进制文件. 对 32 位和 64 位 iOS 框架来说, 这种操作尤其合理. 这种情况下, 最终产生的通用框架可以同时运行在 32 位和 64 位设备上.

> fat 框架必须使用与原框架相同的基本名称(base name). 否则会发生错误.

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // 创建并配置编译目标.
    val ios32 = watchosArm32("watchos32")
    val ios64 = watchosArm64("watchos64")
    configure(listOf(watchos32, watchos64)) {
        binaries.framework {
            baseName = "my_framework"
        }
    }
    // 创建 fat 框架的构建任务.
    tasks.register<FatFrameworkTask>("debugFatFramework") {
        // fat 框架必须使用与原框架相同的基本名称(base name).
        baseName = "my_framework"
        // 默认的输出目录是 "<build directory>/fat-framework".
        destinationDir = buildDir.resolve("fat-framework/debug")
        // 指定需要合并的框架.
        from(
            ios32.binaries.getFramework("DEBUG"),
            ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // 创建并配置编译目标.
    targets {
        watchosArm32("watchos32")
        watchosArm64("watchos64")
```

```
        configure([watchos32, watchos64]) {
            binaries.framework {
                baseName = "my_framework"
            }
        }
    }
    // 创建 fat 框架的构建任务.
    tasks.register("debugFatFramework", FatFrameworkTask) {
        // fat 框架必须使用与原框架相同的基本名称(base name).
        baseName = "my_framework"
        // 默认的输出目录是 "<build directory>/fat-framework".
        destinationDir = file("$buildDir/fat-framework/debug")
        // 指定需要合并的框架.
        from(
            targets.ios32.binaries.getFramework("DEBUG"),
            targets.ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

构建 XCFramework

所有的 Kotlin 跨平台项目都可以使用 XCFramework 作为输出, 将用于所有目标平台和架构的逻辑收集在单个 bundle 之内. 与 单个通用的(fat)框架 不同, 在将应用程序发布到 App Store 之前, 你不需要删除所有不必要的架构.

Kotlin

```
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFramework

plugins {
    kotlin("multiplatform")
}

kotlin {
    val xcf = XCFramework()
    val iosTargets = listOf(iosX64(), iosArm64(),
iosSimulatorArm64())
```

```
    iosTargets.forEach {
        it.binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
}
```

Groovy

```
import
org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFrameworkConfig

plugins {
    id 'org.jetbrains.kotlin.multiplatform'
}

kotlin {
    def xcf = new XCFrameworkConfig(project)
    def iosTargets = [iosX64(), iosArm64(), iosSimulatorArm64()]

    iosTargets.forEach {
        it.binaries.framework {
            baseName = 'shared'
            xcf.add(it)
        }
    }
}
```

在你声明 XCFramework 时, Kotlin Gradle plugin 会注册 3 个 Gradle task:

- `assembleXCFramework`
- `assembleDebugXCFramework` (额外的 debug artifact, 其中包含 dSYMs ([符号化(Symbolicate) iOS 崩溃报告(Crash Report)]))

- `assembleReleaseXCFramework`

如果在你的项目中使用 CocoaPods 集成 ([CocoaPods 概述与设置](#)), 那么可以使用 Kotlin CocoaPods Gradle plugin 构建 XCFramework. 它包含以下 task, 使用所有已注册的编译目标构建 XCFramework, 并生成 podspec 文件:

- `podPublishReleaseXCFramework`, 生成 release 版 XCFramework 以及一个 podspec 文件.
- `podPublishDebugXCFramework`, 生成 debug 版 XCFramework 以及一个 podspec 文件.
- `podPublishXCFramework`, 生成 debug 版和 release 版 XCFramework 以及一个 podspec 文件.

通过这些 task, 可以帮助你将你的项目的共用部分从移动应用程序中分离出来, 单独通过 CocoaPod 发布. 你也可以使用 XCFramework 来发布到私有的或公共的 podspec 仓库.

如果 Kotlin 框架使用不同的 Kotlin 版本构建, 那么不推荐发布这些框架到公共仓库. 这样做可能导致在最终使用者的项目中发生冲突.

定制 Info.plist 文件

输出框架时, Kotlin/Native 编译器会生成信息属性列表文件, `Info.plist`. 你可以使用相应的二进制选项来定制其中的属性:

属性	二进制
`CFBundleIdentifier`	`bundleId`
`CFBundleShortVersionString`	`bundleShortVersionString`
`CFBundleVersion`	`bundleVersion`

要启用这个功能, 请对指定的框架使用 `-Xbinary=$option=$value` 编译器选项, 或通过 Gradle DSL 设置 `binaryOption("option", "value")`:

```
binaries {
    framework {
        binaryOption("bundleId", "com.example.app")
```

```
        binaryOption("bundleVersion", "2")
    }
}
```

发布跨平台的库

最终更新: 2024/09/10

你可以使用 `maven-publish` Gradle plugin (https://docs.gradle.org/current/userguide/publishing_maven.html), 将跨平台的库发布到本地的 Maven 仓库. 只需要在 `shared/build.gradle.kts` 文件中, 指定库的 group, version, 以及需要发布到的 仓库 (https://docs.gradle.org/current/userguide/publishing_maven.html#publishing_maven:repositories). plugin 会自动创建发布任务.

```
plugins {
    //...
    id("maven-publish")
}

group = "com.example"
version = "1.0"

publishing {
    repositories {
        maven {
            //...
        }
    }
}
```

> 你也可以将跨平台的库发布到 GitHub 仓库. 详情请参见 GitHub 文档 GitHub packages (https://docs.github.com/en/packages).

发布的结构

当与 `maven-publish` 一起使用时, Kotlin plugin 对在当前主机上能够构建的每个编译目标, 都会自动创建发布任务, Android 编译目标除外, 因为它需要 更多步骤来配置发布任务.

跨平台库的发布会包含一个额外的 *root* 发布 `kotlinMultiplatform`, 这是用作整个库的发布, 如果将它添加为共通源代码集的依赖项, 它会自动解析为适当的平台相关 artifact. 详情请参见 添加依赖项

(添加跨平台库依赖项).

这个 `kotlinMultiplatform` 发布包含元数据 artifact, 而且会引用其他发布作为它的变体(variant).

> 有些仓库, 比如 Maven Central, 要求 root 模块包含不带分类标识的 JAR artifact, 比如 `kotlinMultiplatform-1.0.jar`. Kotlin Multiplatform plugin 会自动产生需要的 artifact, 以及内嵌的元数据 artifact. 也就是说, 你不需要自定义你的构建脚本, 向你的库的 root 模块添加一个空的 artifact, 来满足仓库的要求.

如果仓库要求, `kotlinMultiplatform` 发布还可能会需要源代码和文档的 artifact. 这种情况下, 请在 publication 内使用 `artifact(...)` (https://docs.gradle.org/current/javadoc/org/gradle/api/publish/maven/MavenPublication.html#artifact-java.lang.Object-) 添加这些需要的 artifact.

对主机的要求

除 Apple 平台的编译目标之外, Kotlin/Native 支持交叉编译(cross-compilation), 可以在任何主机上生成需要的 artifact.

为了避免发布期间发生问题:

- 如果你的项目的编译目标包含 Apple 操作系统, 请只从 Apple 主机发布.
- 只从一个主机发布所有的 artifact, 以免在仓库中重复发布.

  例如, Maven Central, 明确禁止重复发布, 并会让发布过程失败.

如果你使用 Kotlin 1.7.0 或更早版本

在 1.7.20 之前, Kotlin/Native 编译器不支持全部的交叉编译(cross-compilation)选项. 如果你使用更早的版本, 你可能需要从多个主机发布跨平台项目: 使用 Windows 主机编译 Windows 编译目标, 使用 Linux 主机编译 Linux 编译目标, 等等. 这可能会导致那些交叉编译的模块被重复发布. 要避免这个问题, 最直接的方法是, 升级到比较新的 Kotlin 版本, 如上文描述的那样, 从单个主机进行发布.

如果无法升级, 请在 `shared/build.gradle(.kts)` 文件中为每个编译目标指定一个 main host, 并检查这个标记:

Kotlin

```
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()

    val publicationsFromMainHost =
        listOf(jvm(), js()).map { it.name } +
"kotlinMultiplatform"

    publishing {
        publications {
            matching { it.name in publicationsFromMainHost }.all
{
                val targetPublication = this@all
                tasks.withType<AbstractPublishToMaven>()
                    .matching { it.publication ==
targetPublication }
                    .configureEach { onlyIf {
findProperty("isMainHost") == "true" } }
            }
        }
    }
}
```

Groovy

```
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()

    def publicationsFromMainHost =
        [jvm(), js()].collect { it.name } +
"kotlinMultiplatform"
```

```
    publishing {
        publications {
            matching { it.name in publicationsFromMainHost }.all
{ targetPublication ->
                tasks.withType(AbstractPublishToMaven)
                    .matching { it.publication ==
targetPublication }
                    .configureEach { onlyIf {
findProperty("isMainHost") == "true" } }
            }
        }
    }
}
```

发布 Android 库

要发布一个 Android 库, 需要一些额外的配置.

默认情况下, 没有任何 Android 库的 artifact 会发布. 要发布一组 Android 编译变体(variant) (https://developer.android.com/studio/build/build-variants) 生成的 artifact, 需要在 `shared/build.gradle.kts` 文件的 Android 编译目标代码段内指定编译变体名称：

```
kotlin {
    android {
        publishLibraryVariants("release", "debug")
    }
}
```

上面的示例适用于没有 产品风格(Product Flavor) (https://developer.android.com/studio/build/build-variants#product-flavors) 的 Android 库. 对于存在产品风格(Product Flavor)的库, 编译变体名称还需要包含产品风格名称, 比如 `fooBarDebug` 或 `fooBarRelease`.

默认的发布设置如下:

- 如果发布的编译变体是相同的构建类型 (比如, 都是 `release` 或 `debug`), 那么它们将兼容任意的使用者构建类型.

- 如果发布的编译变体是不同的构建类型, 那么只有 release 变体兼容于与发布的编译变体不同的使用者构建类型. 所有的其他编译变体 (比如 `debug`) 只会与使用者的相同构建类型匹配, 除非使用者项目指定了 匹配回退(Matching Fallback) (https://developer.android.com/reference/tools/gradle-api/4.2/com/android/build/api/dsl/BuildType).

如果你希望让所有发布的 Android 变体都只兼容于库的使用者的相同构建类型, 请设置 Gradle 属性: `kotlin.android.buildTypeAttribute.keep=true`.

也可以将各个编译变体以产品风格为单位分组发布, 使得不同的编译类型的输出文件可以放在同一个模块内, 编译类型成为 artifact 中的一个分类符 (release 编译类型的结果发布时仍然不带分类符). 这种发布模式默认是关闭的, 如果要启用, 请在 `shared/build.gradle.kts` 文件中使用以下设置:

```
kotlin {
    android {
        publishLibraryVariantsGroupedByFlavor = true
    }
}
```

> 如果不同的编译变体存在不同的依赖项, 那么不推荐以产品风格为单位分组发布编译变体, 因为它们的依赖项会组合在一起, 成为一个庞大的依赖项列表.

禁用源代码的发布

Kotlin Multiplatform Gradle plugin 默认会对所有指定的编译目标发布源代码. 但是, 你可以在 `shared/build.gradle.kts` 文件中使用 `withSourcesJar()` API 配置并禁用源代码发布:

- 对所有的编译目标禁用源代码发布:

  ```
  kotlin {
      withSourcesJar(publish = false)

      jvm()
      linuxX64()
  }
  ```

- 只对指定的编译目标禁用源代码发布:

```
kotlin {
     // 只对 JVM 禁用源代码发布:
    jvm {
        withSourcesJar(publish = false)
    }
    linuxX64()
}
```

- 对指定的编译目标之外的所有编译目标禁用源代码发布:

```
kotlin {
    // 对 JVM 之外的所有编译目标禁用源代码发布:
    withSourcesJar(publish = false)

    jvm {
        withSourcesJar(publish = true)
    }
    linuxX64()
}
```

如何向你的开发团队介绍跨平台移动开发

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/multiplatform-introduce-your-team.html)

跨平台程序的 Gradle DSL 参考文档

最终更新: 2024/09/10

Kotlin Multiplatform Gradle plugin, 是一个用来创建 Kotlin Multiplatform (Kotlin Multiplatform) 项目的工具. 本章我们提供关于它的参考文档; 当你为 Kotlin Multiplatform 项目编写 Gradle 编译脚本时可以参考本文档. 详情请参见 关于 Kotlin Multiplatform 项目的基本概念, 如何创建和配置跨平台项目 (Kotlin 跨平台程序开发入门).

插件 Id 与版本

Kotlin 跨平台 Gradle 插件的完整限定名称是 `org.jetbrains.kotlin.multiplatform`. 如果使用 Kotlin Gradle DSL, 可以通过 `kotlin("multiplatform")` 语句应用这个插件. 插件的版本与 Kotlin 发布版本一致. 最新的版本是 1.9.23.

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}
```

顶级代码块

Gradle 编译脚本中跨平台项目配置的顶级代码块是 `kotlin`. 在 `kotlin` 之内, 你可以使用以下代码块:

代码块	解释
`<targetName>`	为项目声明一个特定的编译目标. 可以选择的编译目标名称请参见 编译目标 小节.
`targets`	项目的所有编译目标.
`presets`	所有预定义的编译目标. 使用这个代码块可以一次性 设置多个预定义的编译目标.
`sourceSets`	为项目设置预定义的源代码集, 并声明自定义 源代码集.

编译目标

编译目标(Target) 是指针对某个特定的支持的平台的一系列编译功能, 包括源代码编译, 测试, 打包. Kotlin 对每个平台提供了预设置的编译目标(Target Preset). 参见 如何使用预设置的编译目标 (Target Preset) (为 Kotlin Multiplatform 设置编译目标).

每个编译目标可以包含一个或多个 编译任务(compilation). 除了用于测试和产品的默认的编译任务之外, 你还可以 创建自定义的编译任务 ("创建自定义编译任务" in "配置编译任务").

跨平台项目的编译目标通过 `kotlin` 之内的相应代码块进行描述, 比如, `jvm`, `android`, `iosArm64`. 可选的编译目标如下:

目标平台	编译目标的预定义配置	注释
Kotlin/JVM	jvm	
Kotlin/JS	js	选择执行环境: • browser {} 用于运行在浏览器内的应用程序. • nodejs {} 运行在 Node.js 上的应用程序. 更多详情请参见 创建 Kotlin JavaScript 项目 ("执行环境" in "创建 Kotlin/JS 工程(Project)").
Kotlin/Native		对 macOS, Linux, 和 Windows 主机, 目前支持的编译目标请参见 Kotlin/Native 支持的目标平台 (Kotlin/Native 支持的目标平台).
Android 应用程序和库	android	手动应用 Android Gradle plugin: com.android.application 或 com.android.library. 对每个 Gradle 子项目, 只能创建一个 Android 编译目标.

构建中会忽略当前主机不支持的编译目标, 因此这些编译目标不会被发布.

```
kotlin {
    jvm()
    iosX64()
    macosX64()
    js().browser()
}
```

编译目标的配置包括以下两部分:

- 对所有编译目标有效的 共通配置.

- 特定编译目标独有的配置.

每个编译目标可以有一个或多个 编译任务(compilation).

所有编译目标的共通配置

在任何一种编译目标代码块之内, 都可以使用以下声明:

名称	解释
attributes	针对单个平台 对编译目标消除歧义 ("对一个平台区分多个编译目标" in "为 Kotlin Multiplatform 设置编译目标") 的属性设置.
preset	如果存在的话, 代表创建这个编译目标时使用的预定义设置.
platformType	指定这个编译目标的 Kotlin 平台. 允许的值是: jvm, androidJvm, js, native, common.
artifactsTaskName	负责编译这个编译目标的结果 artifact 的编译任务的名称.
components	用于设置 Gradle publication 的组件.

JVM 编译目标

除 所有编译目标的共通配置 之外, jvm 编译目标还支持以下专有函数:

名称	解释
withJava()	在 JVM 编译目标的编译任务中包含 Java 源代码.

对同时包含 Java 和 Kotlin 源代码文件的项目, 请使用这个函数. 注意 Java 源代码文件的默认目录与 Java 插件的默认设定不同. 相反, 这个默认设定继承自 Kotlin 源代码集. 比如, 如果 JVM 编译目标使用默认名称 jvm, 那么默认的 Java 源代码文件目录是 src/jvmMain/java (正式产品的 Java 源代码) 和 src/jvmTest/java (测试程序的 Java 源代码). 详情请参见 在 JVM 编译任务中使用 Java 源代码 ("在 JVM 编译任务中使用 Java 源代码" in "配置编译任务").

```
kotlin {
    jvm {
        withJava()
    }
}
```

JavaScript 编译目标

js 代码块描述 JavaScript 编译目标的配置. 根据编译目标的执行环境不同, 它可以包含以下两个代码块之一:

名称	解释
browser	浏览器编译目标的配置.
nodejs	Node.js 编译目标的配置.

详情请参见 配置 Kotlin/JS 项目 (创建 Kotlin/JS 工程(Project)).

浏览器

browser 代码块包含以下配置代码块:

名称	解释
testRuns	测试运行任务的配置.
runTask	项目运行的配置.
webpackTask	使用 Webpack (https://webpack.js.org/) 编译项目的配置.
dceTask	死代码剔除(Dead Code Elimination) (JavaScript 死代码剔除工具) 的配置.
distribution	输出文件的路径.

```
kotlin {
    js().browser {
```

```
        webpackTask { /* ... */ }
        testRuns { /* ... */ }
        dceTask {
            keep("myKotlinJsApplication.org.example.keepFromDce")
        }
        distribution {
            directory = File("$projectDir/customdir/")
        }
    }
}
```

Node.js

nodejs 代码块包含测试和运行任务的配置:

名称	解释
testRuns	测试任务的配置.
runTask	项目运行任务的配置.

```
kotlin {
    js().nodejs {
        runTask { /* ... */ }
        testRuns { /* ... */ }
    }
}
```

原生(Native)编译目标

对于原生(Native)编译目标, 可以使用以下代码块:

名称	解释
binaries	编译输出的 二进制文件(binary) 的配置.
cinterops	与 C 库文件交互 的配置.

二进制文件(Binary)

有以下几种二进制文件(Binary)任务:

名称	解释
`executable`	正式产品的可执行文件.
`test`	测试程序的可执行文件.
`sharedLib`	共享的库文件(Shared library).
`staticLib`	静态库文件(Static library).
`framework`	Objective-C 框架.

```
kotlin {
    linuxX64 { // 请在这里使用你的编译目标.
        binaries {
            executable {
                // 针对这个二进制文件的配置.
            }
        }
    }
}
```

对于二进制文件的配置, 可以设置的参数包括:

名称	解释
compilation	用于构建二进制文件的编译任务. 默认情况下, test 二进制文件 由 test 编译任务构建, 其他 二进制文件由 main 编译任务构建.
linkerOpts	构建二进制文件时, 传递给操作系统链接程序(linker)的选项.
baseName	对输出文件自定义它的基本名称(base name). 最终的完整文件名会在这个基本名称之上加上相应系统的前缀和后缀.
entryPoint	可执行二进制文件的入口点(entry point) 函数. 默认情况下, 是顶层包中的 main() 函数.
outputFile	用于访问输出文件.
linkTask	用于访问链接任务.
runTask	用于访问可执行二进制文件的运行任务. 对于 linuxX64, macosX64, 或 mingwX64 之外的编译目标, 这个属性的值为 null.
isStatic	用于 Objective-C 框架. 包含静态库(static library), 而不是动态库(dynamic library).

Kotlin

```
binaries {
    executable("my_executable", listOf(RELEASE)) {
        // 以测试编译任务为基础, 构建一个二进制文件.
        compilation = compilations["test"]

        // 对连接器自定义命令行选项.
        linkerOpts = mutableListOf("-L/lib/search/path", "-
L/another/search/path", "-lmylib")
```

```
        // 指定输出文件的基本名称.
        baseName = "foo"

        // 自定义入口点函数.
        entryPoint = "org.example.main"

        // 访问输出文件.
        println("Executable path: ${outputFile.absolutePath}")

        // 访问链接任务.
        linkTask.dependsOn(additionalPreprocessingTask)

        // 访问运行任务.
        // 注意, 对于当前编译环境不支持的(non-host) 平台, runTask 将会
是 null.
        runTask?.dependsOn(prepareForRun)
    }

    framework("my_framework" listOf(RELEASE)) {
        // 在框架中包含静态库而不是动态库.
        isStatic = true
    }
}
```

Groovy

```
binaries {
    executable('my_executable', [RELEASE]) {
        // 以测试编译任务为基础, 构建一个二进制文件.
        compilation = compilations.test

        // 对连接器自定义命令行选项.
        linkerOpts = ['-L/lib/search/path', '-
L/another/search/path', '-lmylib']

        // 指定输出文件的基本名称.
```

```
        baseName = 'foo'

        // 自定义入口点函数.
        entryPoint = 'org.example.main'

        // 访问输出文件.
        println("Executable path: ${outputFile.absolutePath}")

        // 访问链接任务.
        linkTask.dependsOn(additionalPreprocessingTask)

        // 访问运行任务.
        // 注意, 对于当前编译环境不支持的(non-host) 平台, runTask 将会
是 null.
        runTask?.dependsOn(prepareForRun)
    }

    framework('my_framework' [RELEASE]) {
        // 在框架中包含静态库而不是动态库.
        isStatic = true
    }
}
```

详情请参见 构建原生二进制文件 (构建最终的原生二进制文件).

CInterops

cinterops 用于描述与原生库的交互. 要与一个库交互, 请添加一个 cinterops 代码块, 并定义它的参数, 如下:

名称	解释
defFile	描述原生 API 的 def 文件.
packageName	生成 Kotlin API 时的包前缀.
compilerOpts	cinterop 工具传递给编译器的选项.
includeDirs	用于查找头文件的目录.

详情请参见 如何配置与原生语言的交互 ("配置与原生语言的交互" in "配置编译任务").

Kotlin

```
kotlin {
    linuxX64 { // 这里请替换为你需要的编译目标.
        compilations.getByName("main") {
            val myInterop by cinterops.creating {
                // 描述原生 API 的 def 文件.
                // 默认路径是 src/nativeInterop/cinterop/<interop-
name>.def
                defFile(project.file("def-file.def"))

                // 生成的 Kotlin API 所在的包.
                packageName("org.sample")

                // cinterop 工具传递给编译器的参数.
                compilerOpts("-Ipath/to/headers")

                // 用于查找头文件的目录 (等同于 -I<path> 编译选项).
                includeDirs.allHeaders("path1", "path2")

                // includeDirs.allHeaders 的缩写方式.
                includeDirs("include/directory",
"another/directory")
            }
```

```
            val anotherInterop by cinterops.creating { /* ... */
}
        }
    }
}
```

Groovy

```
kotlin {
    linuxX64 { // 这里请替换为你需要的编译目标.
        compilations.main {
            cinterops {
                myInterop {
                    // 描述原生 API 的 def 文件.
                    // 默认路径是
src/nativeInterop/cinterop/<interop-name>.def
                    defFile project.file("def-file.def")

                    // 生成的 Kotlin API 所在的包.
                    packageName 'org.sample'

                    // cinterop 工具传递给编译器的参数.
                    compilerOpts '-Ipath/to/headers'

                    // 用于查找头文件的目录 (等同于 -I<path> 编译选
项).
                    includeDirs.allHeaders("path1", "path2")

                    // includeDirs.allHeaders 的缩写方式.
                    includeDirs("include/directory",
"another/directory")
                }

                anotherInterop { /* ... */ }
            }
        }
```

```
    }
}
```

Android 编译目标

Kotlin Multiplatform plugin 针对 Android 编译目标提供了两个专有的函数. 这两个函数帮助你设置 构建变体(build variants) (https://developer.android.com/studio/build/build-variants):

名称	解释
publishLibraryVariants()	指定用于发布的构建变体. 详情请参见 发布 Android 库 ("发布 Android 库" in "发布跨平台的库").
publishAllLibraryVariants()	发布所有的构建变体.

```
kotlin {
    android {
        publishLibraryVariants("release", "debug")
    }
}
```

详情请参见 针对 Android 的编译 ("Android 编译任务" in "配置编译任务").

> kotlin 代码块之内的 android 配置, 不会替代任何 Android 项目的编译配置. 关于如何为 Android 项目编写编译脚本, 详情请参见 Android 开发文档 (https://developer.android.com/studio/build).

源代码集(Source set)

sourceSets 代码块描述项目的源代码集. 源代码集是指在某个编译任务中一起参与编译的 Kotlin 源代码文件, 相关的资源文件, 依赖项目, 以及语言设置.

跨平台项目的各个编译目标都包含 预定义的源代码集; 开发者也可以根据需要创建 自定义的源代码集.

预定义的源代码集

创建会在跨平台项目时会自动设置预定义的源代码集. 可用的预定义源代码集如下:

名称	解释
commonMain	所有平台共用的代码和资源. 对所有的跨平台项目都可用. 项目的所有 main 编译任务都会使用这个源代码集.
commonTest	所有平台共用的测试代码和资源. 对所有的跨平台项目都可用. 项目的所有 test 编译任务都会使用这个源代码集.
<targetName><compilationName>	各个编译目标专有的源代码集. 这里的 <targetName> 是预定义编译目标的名称, <compilationName> 是这个编译目标的编译任务名称. 比如: jsTest, jvmMain.

使用 Kotlin Gradle DSL 时, 预定义源代码集的代码块需要标记为 by getting.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting { /* ... */ }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain { /* ... */ }
    }
}
```

详情请参见 源代码集 ("源代码集(Source Set)" in "Kotlin Multiplatform 项目结构的基础知识").

自定义源代码集

自定义源代码集由项目开发者手动创建. 要创建一个自定义源代码集, 需要在 sourceSets 之内, 使用自定义源代码集的名称, 添加一个代码块. 如果使用 Kotlin Gradle DSL, 需要将自定义源代码集标记为 by creating.

Kotlin

```
kotlin {
    sourceSets {
        val myMain by creating { /* ... */ } // 创建一个新的, 名为
'MyMain' 的源代码集
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        myMain { /* ... */ } // 创建或设置一个名称为 'myMain' 的源代
码集
    }
}
```

注意, 新创建的源代码集不会关联到其他源代码集. 如果要在项目的编译任务中使用这个源代码集, 请将它关联到其他源代码集 ("手动配置" in "层级项目结构").

源代码集的参数

源代码集的配置保存在相应的 sourceSets 代码块之内. 一个源代码集包含以下参数:

名称	解释
kotlin.srcDir	源代码集目录之内的 Kotlin 源代码文件位置.
resources.srcDir	源代码集目录之内的资源文件位置.
dependsOn	关联到另一个源代码集 ("手动配置" in "层级项目结构")
dependencies	源代码集的 依赖项目.
languageSettings	用于这个源代码集的 语言设置.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            kotlin.srcDir("src")
            resources.srcDir("res")

            dependencies {
                /* ... */
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            kotlin.srcDir('src')
            resources.srcDir('res')
```

```
                dependencies {
                    /* ... */
                }
            }
        }
    }
```

编译任务

一个编译目标可以包含一个或多个编译任务, 比如, 用于正式产品的编译任务, 或用于测试代码的编译任务. 编译目标创建时会自动添加 预定义的编译任务. 你也可以另外创建 自定义编译任务.

如果需要访问一个编译目标的所有编译任务, 或某个特定的编译任务, 请使用 `compilations` 对象集合. 通过 `compilations`, 你可以使用名称来访问一个编译任务.

详情请参见 配置编译任务 (配置编译任务).

预定义的编译任务

对项目的每个编译目标, 会自动创建预定义的编译任务, Android 编译目标除外. 可用的预定义编译任务如下:

名称	解释
`main`	用于正式产品源代码的编译任务.
`test`	用于测试代码的编译任务.

Kotlin

```
kotlin {
    jvm {
        val main by compilations.getting {
            output // 得到 main 编译任务的输出
        }

        compilations["test"].runtimeDependencyFiles // 得到 test
编译任务的运行时 classpath
```

```
    }
}
```

Groovy

```
kotlin {
    jvm {
        compilations.main.output // 得到 main 编译任务的输出
        compilations.test.runtimeDependencyFiles // 得到 test 编译
任务的运行时 classpath
    }
}
```

自定义编译任务

除了预定义的编译任务之外, 你也可以创建自己的自定义编译任务. 要创建一个自定义编译任务, 请在 `compilations` 集合中添加一个新的项目. 如果使用 Kotlin Gradle DSL, 自定义编译任务需要标记为 `by creating`.

详情请参见 创建自定义编译任务 ("创建自定义编译任务" in "配置编译任务").

Kotlin

```
kotlin {
    jvm() {
        compilations {
            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
                        /* ... */
                    }
                }

                // 创建一个 test 任务, 用于运行这个编译任务产生的测试代
码:
                tasks.register<Test>("integrationTest") {
                    /* ... */
```

```
                }
            }
        }
    }
}
```

Groovy

```
kotlin {
    jvm() {
        compilations.create('integrationTest') {
            defaultSourceSet {
                dependencies {
                    /* ... */
                }
            }

            // 创建一个 test 任务, 用于运行这个编译任务产生的测试代码:
            tasks.register('jvmIntegrationTest', Test) {
                /* ... */
            }
        }
    }
}
```

编译任务的参数

一个编译任务可以包含以下参数:

名称	解释
`defaultSourceSet`	编译任务的默认源代码集.
`kotlinSourceSets`	源代码集, 参与这个编译任务.
`allKotlinSourceSets`	源代码集, 参与这个编译任务, 以及通过 `dependsOn()` 关联的所有其他编译任务.
`compilerOptions`	应用于这个编译任务的编译器选项. 关于所有可用的选项, 请参见 编译选项 (Kotlin Gradle plugin 中的编译器选项).
`compileKotlinTask`	编译 Kotlin 源代码的 Gradle 任务.
`compileKotlinTaskName`	`compileKotlinTask` 的名称.
`compileAllTaskName`	编译这个编译任务中所有源代码的Gradle 任务的名称.
`output`	编译任务的输出.
`compileDependencyFiles`	这个编译任务的编译时刻依赖项目文件(classpath).
`runtimeDependencyFiles`	这个编译任务的运行时刻依赖项目文件(classpath).

Kotlin

```
kotlin {
    jvm {
```

```
        val main by compilations.getting {
            compilerOptions.configure {
                // 为 'main' 编译任务设置 Kotlin 编译器选项:
                jvmTarget.set(JvmTarget.JVM_1_8)
            }

            compileKotlinTask // 得到编译 Kotlin 源代码的 Gradle 任务 'compileKotlinJvm'
            output // 得到 main 编译任务的输出
        }

        compilations["test"].runtimeDependencyFiles // 得到 test 编译任务的运行时刻 classpath
    }

    // 对所有编译目标的所有编译任务的设置:
    targets.all {
        compilations.all {
            compilerOptions.configure {
                allWarningsAsErrors.set(true)
            }
        }
    }
}
```

Groovy

```
kotlin {
    jvm {
        compilations.main.compilerOptions.configure {
            // 为 'main' 编译任务设置 Kotlin 编译器选项:
            jvmTarget.set(JvmTarget.JVM_1_8)
        }

        compilations.main.compileKotlinTask // 得到编译 Kotlin 源代码的 Gradle 任务 'compileKotlinJvm'
        compilations.main.output // 得到 main 编译任务的输出
```

```
            compilations.test.runtimeDependencyFiles // 得到 test 编译
任务的运行时刻 classpath
        }

        // 对所有编译目标的所有编译任务的设置:
        targets.all {
            compilations.all {
                compilerOptions.configure {
                    allWarningsAsError.set(true)
                }
            }
        }
}
```

依赖项目

源代码集的 dependencies 代码块包含这个源代码集的依赖项目.

详情请参见 配置依赖项 (配置 Gradle 项目).

有 4 种类型的依赖项目:

名称	解释
api	当前模块的 API 中使用的依赖项目.
implementation	当前模块中使用的依赖项目, 但不向外暴露.
compileOnly	只在当前模块的编译任务中使用的依赖项目.
runtimeOnly	运行时刻的依赖项目, 但在任何模块的编译任务中都不可见.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
```

```
            dependencies {
                api("com.example:foo-metadata:1.0")
            }
        }
        val jvmMain by getting {
            dependencies {
                implementation("com.example:foo-jvm:1.0")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                api 'com.example:foo-metadata:1.0'
            }
        }
        jvmMain {
            dependencies {
                implementation 'com.example:foo-jvm:1.0'
            }
        }
    }
}
```

除此之外, 源代码集之间还可以相互依赖, 形成一种层级结构. 这种情况下, 应该使用 `dependsOn()` 关系.

在编译脚本的最顶层 `dependencies` 代码块中, 也可以声明源代码集的依赖项目. 这种情况下, 依赖项目声明需要使用 `<sourceSetName><DependencyKind>` 格式, 比如, `commonMainApi`.

Kotlin

```
dependencies {
    "commonMainApi"("com.example:foo-common:1.0")
    "jvm6MainApi"("com.example:foo-jvm6:1.0")
}
```

Groovy

```
dependencies {
    commonMainApi 'com.example:foo-common:1.0'
    jvm6MainApi 'com.example:foo-jvm6:1.0'
}
```

语言设置

源代码集的 `languageSettings` 代码块用来定义项目分析和构建的某些方面. 可选的语言设置如下:

名称	解释
`languageVersion`	与某个 Kotlin 版本保持源代码级的兼容性.
`apiVersion`	允许使用从指定的 Kotlin 版本的库才开始提供的 API 声明.
`enableLanguageFeature`	启用指定的语言特性. 这个参数可选的值, 对应于那些目前还处于试验状态的语言特性, 或还没有正式公布的语言特性.
`optIn`	允许使用指定的 明确要求使用者同意(Opt-in) 注解 (明确要求使用者同意的功能(Opt-in Requirement)).
`progressiveMode`	启用 渐进模式(progressive mode) ("渐进模式" in "Kotlin 1.3 版中的新功能").

Kotlin

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.8" // 可选的值: "1.4", "1.5",
"1.6", "1.7", "1.8", "1.9"
            apiVersion = "1.8" // 可选的值: "1.3", "1.4", "1.5",
"1.6", "1.7", "1.8", "1.9"
            enableLanguageFeature("InlineClasses") // 这里请使用语
言特性的名称
            optIn("kotlin.ExperimentalUnsignedTypes") // 这里请使
用注解的完全限定名称
            progressiveMode = true // 默认值为 false
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.8' // 可选的值: '1.4', '1.5',
'1.6', '1.7', '1.8', '1.9'
            apiVersion = '1.8' // 可选的值: '1.3', '1.4', '1.5',
'1.6', '1.7', '1.8', '1.9'
            enableLanguageFeature('InlineClasses') // 这里请使用语
言特性的名称
            optIn('kotlin.ExperimentalUnsignedTypes') // 这里请使
用注解的完全限定名称
            progressiveMode = true // 默认值为 false
        }
    }
}
```

Android 源代码集布局

最终更新: 2024/09/10

Kotlin 1.8.0 引入了 Android 源代码集布局, 并在 1.9.0 中成为默认布局. 请阅读这篇向导, 理解的已废弃的旧布局与新布局之间的主要区别, 以及如何迁移你的项目.

> ⚠ 你不一定要实现这篇向导中的全部建议, 只需要实现那些适合于你的项目的部分.

检查兼容性

新的布局需要 Android Gradle plugin 7.0 或更高版本, 而且需要使用 Android Studio 2022.3 或更高版本. 请检查你的 Android Gradle plugin 版本, 如果需要的话, 请更新到新的版本.

重命名 Kotlin 源代码集

如果需要, 请重命名你的项目中的源代码集, 遵循下面的模式:

以前的源代码集布局	新的源代码集布局
`targetName` + `AndroidSourceSet.name`	`targetName` + `AndroidVariantType`

`{AndroidSourceSet.name}` 与 `{KotlinSourceSet.name}` 的对应关系如下:

	以前的源代码集布局	新的源代码集布局
main	androidMain	androidMain
test	androidTest	android**Unit**Test
androidTest	android**Android**Test	android**Instrumented**Test

移动源代码文件

如果需要, 请将你的源代码文件移动到新的目录, 遵循下面的模式:

以前的源代码集布局	新的源代码集布局
包含额外的 `/kotlin` 源代码目录的布局	`src/{KotlinSourceSet.name}/kotlin`

`{AndroidSourceSet.name}` 与 `{SourceDirectories included}` 的对应关系如下:

	以前的源代码集布局	新的源代码集布局
main	src/androidMain/kotlin src/main/kotlin src/main/java	src/androidMain/kotlin src/main/kotlin src/main/java
test	src/androidTest/kotlin src/test/kotlin src/test/java	src/android**Unit**Test/kotlin src/test/kotlin src/test/java
androidTest	src/android**Android**Test/kotlin src/androidTest/java	src/android**Instrumented**Test/kotlin src/androidTest/java, **src/androidTest/kotlin**

移动 AndroidManifest.xml 文件

如果你的项目中有 `AndroidManifest.xml` 文件, 请移动到新的目录, 遵循下面的模式:

以前的源代码集布局	新的源代码集布局
src/{**Android**SourceSet.name}/AndroidManifest.xml	src/{**Kotlin**SourceSet.name}/AndroidManifest.xml

`{AndroidSourceSet.name}` 与 `{AndroidManifest.xml location}` 的对应关系如下:

	以前的源代码集布局	新的源代码集布局
main	src/main/AndroidManifest.xml	src/**android**Main/AndroidManifest.xml
debug	src/debug/AndroidManifest.xml	src/**android**Debug/AndroidManifest.xml

检查 Android 测试和 common 测试之间的关系

新的 Android 源代码集布局改变了 Android 设备测试(Instrumented Test) (在新的布局中改名为 `androidInstrumentedTest`) 与 common 测试之间的关系.

以前, `androidAndroidTest` 与 `commonTest` 之间默认存在 `dependsOn` 关系. 这意味着:

- `commonTest` 中的代码可以在 `androidAndroidTest` 中访问.
- `commonTest` 中的 `expect` 声明, 在 `androidAndroidTest` 中必须有对应的 `actual` 实现.
- 在 `commonTest` 中声明的测试 也会作为 Android 设备测试(Instrumented Test)运行.

在新的 Android 源代码集布局中, 不会默认添加这个 `dependsOn` 关系. 如果你希望使用以前的行为, 请在你的 `build.gradle.kts` 文件中手动声明下面的关系:

```
kotlin {
    // ...
    sourceSets {
        val commonTest by getting
        val androidInstrumentedTest by getting {
            dependsOn(commonTest)
        }
    }
}
```

调整 Android flavor 的实现

在以前的版本中, Kotlin Gradle plugin 会在很早的阶段创建对应于 `debug` 和 `release` 构建类型的 Android 源代码集, 或对应于自定义 flavor 的 Android 源代码集, 例如 `demo` 和 `full`. 因此这些源代码集可以通过 `val androidDebug by getting { ... }` 这样的表达式来访问.

新的 Android 源代码集布局, 使用 Android 的 `onVariants` (https://developer.android.com/reference/tools/gradle-api/8.0/com/android/build/api/variant/AndroidComponentsExtension#onVariants(com.android.build.api.variant.VariantSelector,kotlin.Function1)) 来创建源代码集. 因此上面的表达式不再有效, 会导致错误: `org.gradle.api.UnknownDomainObjectException: KotlinSourceSet with name 'androidDebug' not found`.

为了解决这样的错误, 请在你的 `build.gradle.kts` 文件中使用新的 `invokeWhenCreated()` API:

```
kotlin {
    // ...
    @OptIn(ExperimentalKotlinGradlePluginApi::class)
    sourceSets.invokeWhenCreated("androidFreeDebug") {
        // ...
    }
}
```

Kotlin Multiplatform 兼容性指南

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/multiplatform-compatibility-guide.html)

Kotlin Multiplatform Mobile Plugin 的发布版本

最终更新: 2024/09/10

我们正在努力开发 Kotlin Multiplatform Mobile plugin for Android Studio (https://plugins.jetbrains.com/plugin/14936-kotlin-multiplatform-mobile) 的稳定版, 我们会不断发布新的版本, 包含新的功能, 改进, 和 bug 修正.

请确认你安装了最新版的 Kotlin Multiplatform Mobile plugin!

更新到新的发布版

如果出现了新的 Kotlin Multiplatform Mobile plugin 发布版, Android Studio 会建议你更新. 如果你接受建议, 它会自动更新 plugin 到最新版本. 你将会需要重新启动 Android Studio 来完成 plugin 的安装.

你可以通过菜单 **Settings/Preferences** | **Plugins**, 查看 plugin 的版本, 并手动更新它.

为了让 plugin 正确工作, 你需要一个兼容的 Kotlin 版本. 你可以在 发布版本详情 中找到对应的兼容版本. 你可以通过菜单 **Settings/Preferences** | **Plugins**, 或 **Tools** | **Kotlin** | **Configure Kotlin Plugin Updates**, 查看你的 Kotlin 版本, 并更新它.

> 如果你没有安装兼容的 Kotlin 版本, Kotlin Multiplatform Mobile plugin 将会被禁用. 你需要更新你的 Kotlin, 然后通过菜单 **Settings/Preferences** | **Plugins**, 启用 plugin.

发布版本详情

下表列出了 Kotlin Multiplatform Mobile plugin 最新发布版的详细信息:

发布版	主要功能	兼容的 Kotlin 版本
0.8.2 发布日期: 2024/01/25	• 支持新的 Canary 版 Android Studio Jellyfish. • 在共用模块中添加 `sourceCompatibility` 和 `targetCompatibility` 声明.	• Kotlin plugin 的任何版本 ("各发布版详情" in "Kotlin 的发布版本")
0.8.1 发布日期: 2023/11/09	• Kotlin 更新到 1.9.20. • Jetpack Compose 更新到 1.5.4. • 默认启用 Gradle 构建和配置的缓存. • 对新的 Kotlin 版本重构了构建配置. • iOS framework 默认为静态模式. • 修正了 iOS 设备上使用 Xcode 15 时的一个问题.	• Kotlin Plugin 的任何版本 ("各发布版详情" in "Kotlin 的发布版本")
0.8.0 发布日期: 2023/10/05	• KT-60169 (https://youtrack.jetbrains.com/issue/KT-60169) 迁移到 Gradle 版本目录. • KT-59269 (https://youtrack.jetbrains.com/issue/KT-59269) `android` 重命名为 `androidTarget`. • KT-59269 (https://youtrack.jetbrains.com/issue/KT-59269) 更新了 Kotlin 和其他依赖项的版本. • KTIJ-26773 (https://youtrack.jetbrains.com/issue/KTIJ-26773) 重构, 使用 `-destination` 参数, 代替 `-sdk` 和 `-arch`. • KTIJ-25839 (https://youtrack.jetbrains.com/issue/KTIJ-25839) 重构了生成的文件名.	• Kotlin Plugin 的任何版本 ("各发布版详情" in "Kotlin 的发布版本")

	• KTIJ-27058 (https://youtrack.jetbrains.com/issue/KTIJ-27058) 添加了 JVM 构建目标配置. • KTIJ-27160 (https://youtrack.jetbrains.com/issue/KTIJ-27160) 支持 Xcode 15.0. • KTIJ-27158 (https://youtrack.jetbrains.com/issue/KTIJ-27158) 将新的模块向导移动到实验状态.	
0.6.0 发布日期: 2023/05/24	• 支持新的 Canary Android Studio Hedgehog. • 更新 Kotlin, Gradle, 以及 Multiplatform 项目中库的版本. • 在 Multiplatform 项目中使用了新的 `targetHierarchy.default()` ("源代码集层级结构的新方案" in "Kotlin 1.8.20 版中的新功能"). • 在 Multiplatform 项目中, 对平台特定的文件使用源代码集名称后缀.	• Kotlin Plugin 的任何版本 ("各发布版详情" in "Kotlin 的发布版本")
0.5.3 发布日期: 2023/04/12	• 更新了 Kotlin 和 Compose 的版本. • 修正了 Xcode 项目 scheme 解析的一个问题. • 添加了 scheme 的 product 类型检查. • 如果 `iosApp` scheme 存在, 默认选中.	• Kotlin Plugin 的任何版本 ("各发布版详情" in "Kotlin 的发布版本")
0.5.2 发布日期: 2023/01/30	• 修正了 Kotlin/Native 调试器的一个问题 (Spotlight 索引缓慢) (https://youtrack.jetbrains.com/issue/KT-55988). • 修正了多模块项目中的 Kotlin/Native 调试器 (https://youtrack.jetbrains.com/issue/KT-24450). • 针对 Android Studio Giraffe 2022.3.1 Canary 的新构建版本 (https://youtrack.jetbrains.com/issue/KT-55274).	• Kotlin Plugin 的任何版本 ("各发布版详情" in "Kotlin 的发布版本")

	• 对 iOS App 构建添加了 provisioning 标记 (https://youtrack.jetbrains.com/issue/KT-55204). • 在生成的 iOS 项目中, 对 **Framework Search Paths** 选项添加了继承的路径 (https://youtrack.jetbrains.com/issue/KT-55402).	
0.5.1 发布日期: 2022/11/30	• 新项目生成时的修正: 删除不需要的 "app" 目录 (https://youtrack.jetbrains.com/issue/KTIJ-23790).	• Kotlin 1.7.0—* ("各发布版详情" in "Kotlin 的发布版本")
0.5.0 发布日期: 2022/11/22	• 修改 iOS framework distribution 的默认选项: 现在是 **Regular framework** (https://youtrack.jetbrains.com/issue/KT-54086). • 在生成的 Android 项目中, 将 `MyApplicationTheme` 移动到单独的文件 (https://youtrack.jetbrains.com/issue/KT-53991). • 修改了生成的 Android 项目 (https://youtrack.jetbrains.com/issue/KT-54658). • 修正了新建项目的目录被意外删除的问题 (https://youtrack.jetbrains.com/issue/KTIJ-23707).	• Kotlin 1.7.0—* ("各发布版详情" in "Kotlin 的发布版本")
0.3.4 发布日期: 2022/09/12	• 将 Android 应用程序迁移到 Jetpack Compose (https://youtrack.jetbrains.com/issue/KT-53162). • 删除旧的 HMPP flag (https://youtrack.jetbrains.com/issue/KT-52248). • 在 Android manifest 中删除包名称 (https://youtrack.jetbrains.com/issue/KTIJ-22633).	• Kotlin 1.7.0—1.7.* ("各发布版详情" in "Kotlin 的发布版本")

	• 对 Xcode 项目更新 .gitignore (https://youtrack.jetbrains.com/issue/KT-53703). • 更新向导项目, 更好的演示 expect/actual 功能 (https://youtrack.jetbrains.com/issue/KT-53928). • 更新与 Android Studio Canary 版的兼容性 (https://youtrack.jetbrains.com/issue/KTIJ-22063). • 对 Android 应用程序, 最小 Android SDK 版本更新为 21 (https://youtrack.jetbrains.com/issue/KTIJ-22505). • 修正安装之后初次启动时的问题 (https://youtrack.jetbrains.com/issue/KTIJ-22645). • 修正 M1 上的 Apple 运行配置的问题 (https://youtrack.jetbrains.com/issue/KTIJ-21781). • 修正 Windows OS 上 local.properties 的问题 (https://youtrack.jetbrains.com/issue/KTIJ-22037). • 修正 Android Studio Canary 版中 Kotlin/Native 调试器的问题 (https://youtrack.jetbrains.com/issue/KT-53976).	
0.3.3 发布日期: 2022/06/09	• 依赖项更新为 Kotlin IDE plugin 1.7.0.	• Kotlin 1.7.0—1.7.* ("各发布版详情" in "Kotlin 的发布版本")
0.3.2 发布日期: 2022/04/04	• 修正在 Android Studio 2021.2 和 2021.3 上的 iOS 应用程序调试性能问题.	• Kotlin 1.5.0—1.6.* ("各发布版详情" in "Kotlin 的发布版本")

0.3.1 发布日期: 2022/02/15	• 在 Kotlin Multiplatform Mobile 向导中启用 M1 iOS 模拟器 (https://youtrack.jetbrains.com/issue/KT-51105). • 对 XcProject 创建索引时的性能改善: KT-49777 (https://youtrack.jetbrains.com/issue/KT-49777), KT-50779 (https://youtrack.jetbrains.com/issue/KT-50779). • 清理构建脚本: 使用 `kotlin("test")`, 代替 `kotlin("test-common")` 和 `kotlin("test-annotations-common")`. • 增加与 Kotlin plugin 版本 (https://youtrack.jetbrains.com/issue/KTIJ-20167) 的兼容范围. • 修正在 Windows 主机上的 JVM 调试问题 (https://youtrack.jetbrains.com/issue/KT-50699). • 修正禁用 plugin 后的版本错误问题 (https://youtrack.jetbrains.com/issue/KT-50699).	• Kotlin 1.5.0—1.6.* ("各发布版详情" in "Kotlin 的发布版本")
0.3.0 发布日期: 2021/11/16	• 新的 Kotlin Multiplatform Library 向导 (https://youtrack.jetbrains.com/issue/KTIJ-19367). • 支持 Kotlin 跨平台库的新发布类型: XCFramework ("构建 XCFramework" in "构建最终的原生二进制文件"). • 对新跨平台移动项目启用 层级项目结构 ("手动配置" in "层级项目结构"). • 支持 iOS 编译目标的明确声明 (https://youtrack.jetbrains.com/issue/KT-46861). • 在非 Mac 机器上启用 Kotlin Multiplatform Mobile plugin 向导 (https://youtrack.jetbrains.com/issue/KT-48614). • 在 Kotlin Multiplatform 模块向导中支持子文件夹 (https://youtrack.jetbrains.com/issue/KT-47923). • 支持 Xcode `Assets.xcassets` 文件 (https://youtrack.jetbrains.com/issue/KT-49571).	• Kotlin 1.6.0 ("各发布版详情" in "Kotlin 的发布版本")

	• 修正了 plugin 的类装载器异常 (https://youtrack.jetbrains.com/issue/KT-48103). • 更新了 CocoaPods Gradle Plugin 模板. • 改进了 Kotlin/Native 调试器的类型计算. • 修正了使用 Xcode 13 的 iOS 设备启动功能.	
0.2.7 发布日期: 2021/08/02	• 为 AppleRunConfiguration 添加了 Xcode 配置选项 (https://youtrack.jetbrains.com/issue/KTIJ-19054). • 添加了 Apple M1 模拟器支持 (https://youtrack.jetbrains.com/issue/KT-47618). • 在项目向导中添加了关于 Xcode 集成选项的信息 (https://youtrack.jetbrains.com/issue/KT-47466). • 当一个使用 CocoaPods 的项目生成后, 但 CocoaPods gem 没有安装时, 添加了错误通知 (https://youtrack.jetbrains.com/issue/KT-47329). • 在使用 Kotlin 1.5.30 生成的共用模块中, 添加了 Apple M1 模拟器编译目标支持 (https://youtrack.jetbrains.com/issue/KT-47631). • 清除使用 Kotlin 1.5.20 生成的 Xcode 项目 (https://youtrack.jetbrains.com/issue/KT-47465). • 修正了真实 iOS 设备上启动 Xcode 的发布配置. • 修正了使用 Xcode 12.5 启动模拟器的功能.	• Kotlin 1.5.10 ("各发布版详情" in "Kotlin 的发布版本")
0.2.6 发布日期: 2021/06/10	• 兼容 Android Studio Bumblebee Canary 1. • 支持 Kotlin 1.5.20 (Kotlin 1.5.20 版中的新功能): 在项目向导中为 Kotlin/Native 使用新的框架打包任务.	• Kotlin 1.5.10 ("各发布版详情" in "Kotlin 的发布版本")

0.2.5 发布日期: 2021/05/25	• 修正了与 Android Studio Arctic Fox 2020.3.1 Beta 1 及更高版本的兼容问题 (https://youtrack.jetbrains.com/issue/KT-46834).	• Kotlin 1.5.10 ("各发布版详情" in "Kotlin 的发布版本")
0.2.4 发布日期: 2021/05/05	对 Android Studio 4.2 或 Android Studio 2020.3.1 Canary 8 或更高版本, 请使用这个 plugin 版本. • 兼容 Kotlin 1.5.0 (Kotlin 1.5.0 版中的新功能). • 在新的 Kotlin Multiplatform 模块中能够使用 CocoaPods 依赖项管理器, 用于 iOS 集成 (https://youtrack.jetbrains.com/issue/KT-45946).	• Kotlin 1.5.0 ("各发布版详情" in "Kotlin 的发布版本")
0.2.3 发布日期: 2021/04/05	• 项目向导: 命名模块的改进 (https://youtrack.jetbrains.com/issues?q=issue%20id:%20KT-43449,%20KT-44060,%20KT-41520,%20KT-45282). • 在项目向导中能够使用 CocoaPods 依赖项管理器, 用于 iOS 集成 (https://youtrack.jetbrains.com/issue/KT-45478). • 新项目中 `gradle.properties` 文件更好的可读性 (https://youtrack.jetbrains.com/issue/KT-42908). • 如果不选中 "Add sample tests for Shared Module", 则不再生成示例测试 (https://youtrack.jetbrains.com/issue/KT-43441). • Bug 修正和其它改进 (https://youtrack.jetbrains.com/issues?q=Subsystems:%20%7BKMM%20Plugin%7D%20Type:%20Feature,%20Bug%20State:%20-Obsolete,%20-%7BAs%20designed%7D,%20-Answered,%20-Incomplete%20resolved%20date:%202021-03-10%20..%202021-03-25).	• Kotlin 1.4.30 ("各发布版详情" in "Kotlin 的发布版本")

0.2.2 发布日期: 2021/03/03	• 能够在 Xcode 中打开 Xcode 相关文件 (https://youtrack.jetbrains.com/issue/KT-44970). • 能够在 iOS 运行配置中为 Xcode 项目文件设置位置 (https://youtrack.jetbrains.com/issue/KT-44968). • 支持 Android Studio 2020.3.1 Canary 8 (https://youtrack.jetbrains.com/issue/KT-45162). • Bug 修正和其它改进 (https://youtrack.jetbrains.com/issues?q=tag:%20KMM-0.2.2%20).	• Kotlin 1.4.30 ("各发布版详情" in "Kotlin 的发布版本")
0.2.1 发布日期: 2021/02/15	对 Android Studio 4.2, 请使用这个 plugin 版本. • 基础组件改进. • Bug 修正和其它改进 (https://youtrack.jetbrains.com/issues?q=tag:%20KMM-0.2.1%20).	• Kotlin 1.4.30 ("各发布版详情" in "Kotlin 的发布版本")
0.2.0 发布日期: 2020/11/23	• 支持 iPad 设备 (https://youtrack.jetbrains.com/issue/KT-41932). • 支持 Xcode 中配置的自定义 scheme 名称 (https://youtrack.jetbrains.com/issue/KT-41677). • 能够为 iOS 运行配置添加自定义构建步骤 (https://youtrack.jetbrains.com/issue/KT-41678). • 能够调试一个自定义 Kotlin/Native 二进制文件 (https://youtrack.jetbrains.com/issue/KT-40954). • 简化了 Kotlin Multiplatform Mobile 向导生成的代码 (https://youtrack.jetbrains.com/issue/KT-41712). • 删除了对 Kotlin Android Extensions plugin 的支持 (https://youtrack.jetbrains.com/issue/KT-42121), 这个功能在 Kotlin 1.4.20 中已废弃.	• Kotlin 1.4.20 ("各发布版详情" in "Kotlin 的发布版本")

	• 修正了从主机断开连接之后保存物理设备配置的功能 (https://youtrack.jetbrains.com/issue/KT-42390). • Bug 修正和其它改进.	
0.1.3 发布日期: 2020/10/02	• 添加了对 iOS 14 和 Xcode 12 的兼容性. • 修正了 Kotlin Multiplatform Mobile 向导创建的平台测试中的名称.	• Kotlin 1.4.10 ("各发布版详情" in "Kotlin 的发布版本") • Kotlin 1.4.20 ("各发布版详情" in "Kotlin 的发布版本")
0.1.2 发布日期: 2020/09/29	• 修正了对 Kotlin 1.4.20-M1 ("EAP 版本" in "参加 Kotlin EAP 项目") 的兼容性. • 默认启用向 JetBrains 发送错误报告.	• Kotlin 1.4.10 ("各发布版详情" in "Kotlin 的发布版本") • Kotlin 1.4.20 ("各发布版详情" in "Kotlin 的发布版本")
0.1.1 发布日期: 2020/09/10	• 修正了对 Android Studio Canary 8 和更高版本的兼容性.	• Kotlin 1.4.10 ("各发布版详情" in "Kotlin 的发布版本") • Kotlin 1.4.20 ("各发布版详

		情" in "Kotlin 的发布版本")
0.1.0 发布日期: 2020/08/31	• 这是 Kotlin Multiplatform Mobile plugin 的第 1 个版本. 详情请参见这篇 Blog (https://blog.jetbrains.com/kotlin/2020/08/kotlin-multiplatform-mobile-goes-alpha/).	• Kotlin 1.4.0 ("各发布版详情" in "Kotlin 的发布版本") • Kotlin 1.4.10 ("各发布版详情" in "Kotlin 的发布版本")

Kotlin/JVM 入门

最终更新: 2024/09/10

本教程演示如何使用 IntelliJ IDEA 创建一个控制台应用程序.

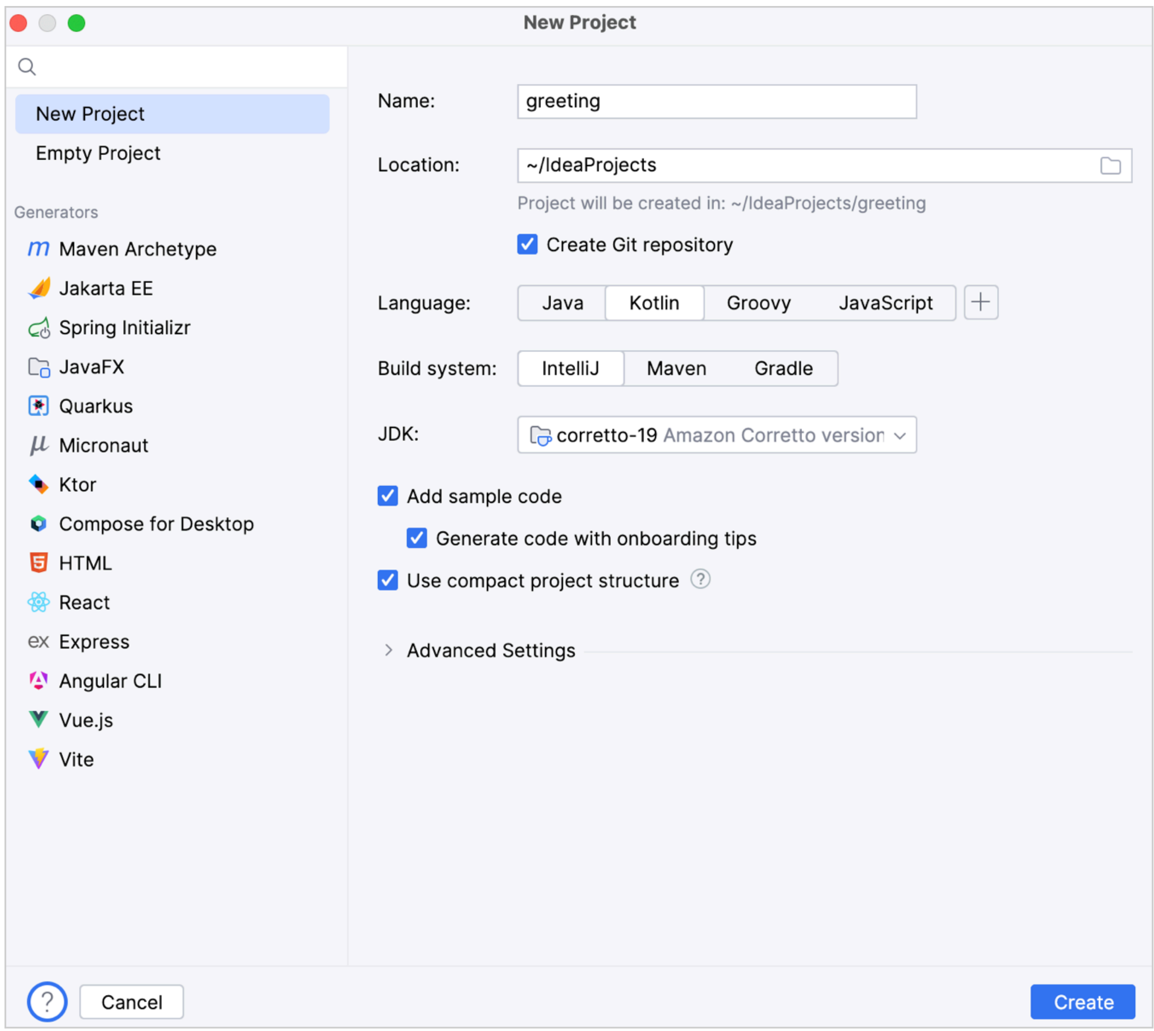
开始之前, 首先请下载并安装最新版的 IntelliJ IDEA (https://www.jetbrains.com/idea/download/index.html).

创建项目

1. 在 IntelliJ IDEA 中, 选择菜单 **File** | **New** | **Project**.
2. 在左侧面板中, 选择 **New Project**.
3. 输入项目名称, 如果需要的话, 修改它的保存位置.

   > 选择 **Create Git repository**, 可以将新项目存入版本控制系统. 这个操作也可以在之后的任何时候进行.

4. 在 **Language** 列表中, 选择 **Kotlin**.



创建一个控制台应用程序

5. 选择 **IntelliJ** 构建系统. 这是原生的构建器, 不需要下载任何额外的 artifact.

   如果你希望创建更加复杂的项目, 需要更多配置, 请选择 Maven 或 Gradle. 对于 Gradle,请选择构建脚本的语言: Kotlin 或 Groovy.

6. 在 **JDK list** 中, 选择希望在项目中使用的 JDK (https://www.oracle.com/java/technologies/downloads/).

   - 如果在你的计算机上已经安装了 JDK, 但在 IDE 中没有定义, 请选择 **Add JDK**, 并指定 JDK home 目录的路径.
   - 如果在你的计算机上还没有需要的 JDK, 请选择 **Download JDK**.

7. 启用 **Add sample code** 选项, 创建文件, 其中包含 `"Hello World!"` 示例程序.

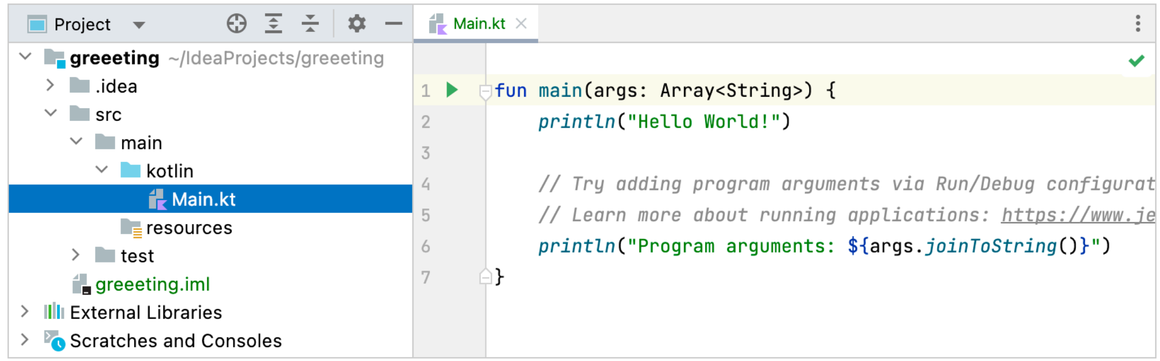
> ⚠ 你也可以启用 **Generate code with onboarding tips** 选项, 向你的示例代码添加一些有用的注释.

8. 点击 **Create**.

> ℹ 如果你选择了 Gradle 构建系统, 那么在你的项目中会有一个构建脚本文件: `build.gradle(.kts)`. 其中包含 `kotlin("jvm")` 插件, 以及你的控制台应用程序需要的依赖项目. 请确认使用了插件的最新版本:
>
> ```
> plugins {
>     kotlin("jvm") version "1.9.23"
>     application
> }
> ```

创建应用程序

1. 打开 `src/main/kotlin` 中的 `Main.kt` 文件. `src` 目录包含 Kotlin 源代码文件和资源文件. `Main.kt` 文件包含示例代码, 它会输出 `Hello World!`.



包括 main 函数的 Main.kt 文件

2. 修改代码, 让它询问你的名字, 然后只对你说 `Hello`, 而不是对整个世界:

- 使用关键字 `val` 引入一个局部变量 `name`. 它会得到你输入的名字 – `readln()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/readln.html).
- 使用字符串模板, 直接在输出的文本内, 在变量名之前添加一个 `$` 符号 – `$name`.

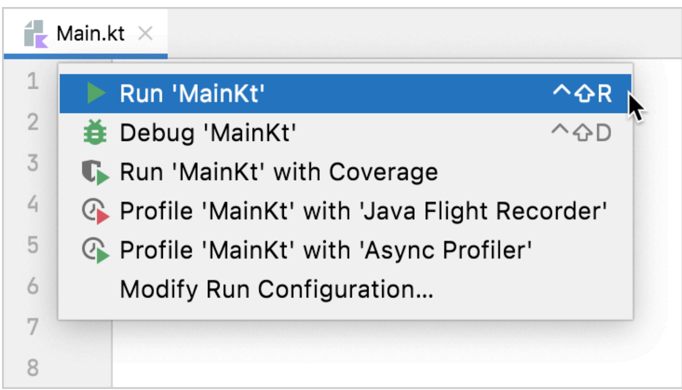
```
fun main() {
    println("What's your name?")
    val name = readln()
    println("Hello, $name!")
}
```

Main.kt
fun main() {
    println("What's your name?")
    val name = readln()
    println("Hello, $name!")
}

修改 main 函数

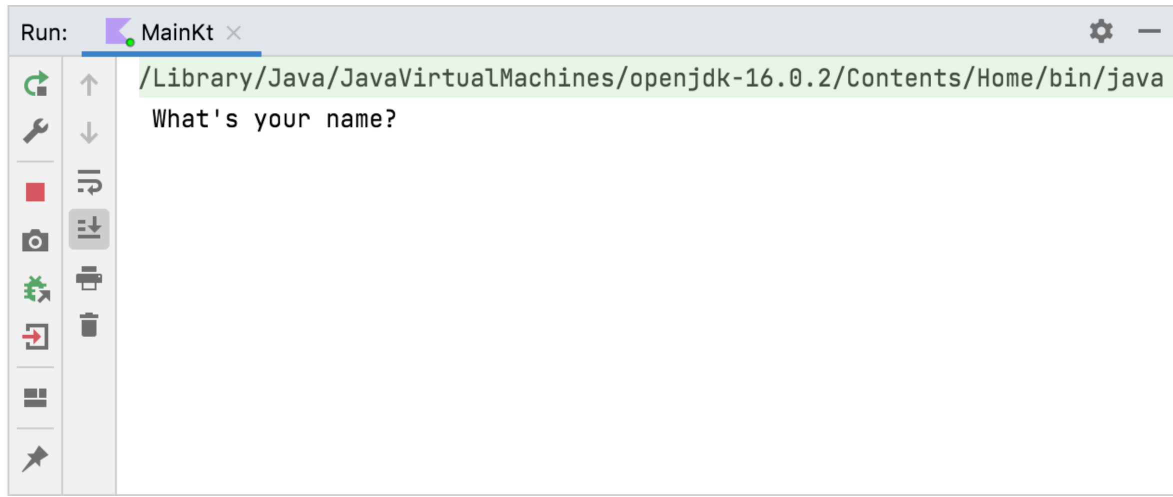
运行应用程序

现在应用程序已经可以运行了. 最简单的方法是, 在源代码编辑器侧栏中按绿色的 **Run** 图标, 然后选择 **Run 'MainKt'**.



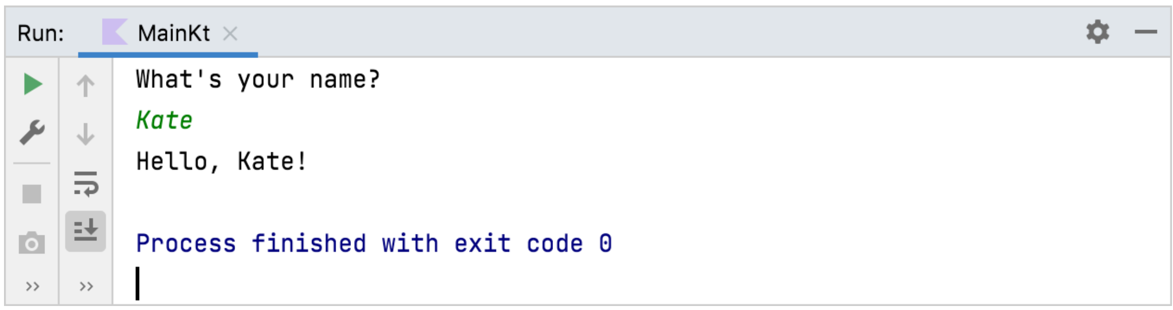
运行控制台应用程序

你可以在 **Run** 工具窗口中看到运行结果.



Kotlin 的运行输出

输入你的名字, 然后可以看到你的应用程序向你问候!



Kotlin 的运行输出

恭喜你! 你已经运行了你的第一个 Kotlin 应用程序.

下一步做什么?

创建了这个应用程序之后, 你可以开始更加深入的学习 Kotlin 语法:

- 从 Kotlin 示例程序 (https://play.kotlinlang.org/byExample/overview) 添加示例代码
- 在 IDEA 中安装 JetBrains Academy Plugin (https://plugins.jetbrains.com/plugin/10081-jetbrains-academy), 并完成 Kotlin Koan 课程 (https://plugins.jetbrains.com/plugin/10081-jetbrains-academy/docs/learner-start-guide.html?section=Kotlin%20Koans) 中的练习

与 Java 比较

最终更新: 2024/09/10

Kotlin 中得到解决的一些 Java 问题

Java 中长期困扰的一系列问题, 在 Kotlin 得到了解决:

- Null 引用 由类型系统管理 (Null 值安全性).
- 没有原生类型(raw type) ("在 Kotlin 中使用 Java 的泛型" in "在 Kotlin 中调用 Java 代码")
- Kotlin 中的数组是 类型不可变的 (数组)
- 与 Java 中的 SAM 变换方案相反, Kotlin 中存在专门的 函数类型(function type) ("函数类型(Function Type)" in "高阶函数与 Lambda 表达式")
- 不使用通配符的 使用处类型变异(Use-site variance) ("使用处的类型变异(Use-site variance): 类型投射(Type projection)" in "泛型(Generic): in, out, where")
- Kotlin 中不存在受控 异常 (异常(Exception))
- 集合的接口定义区分为只读集合与可变集合 (集合(Collection)概述)

Java 中有, 而 Kotlin 中没有的功能

- 受控异常 (异常(Exception))
- 不是类的 基本数据类型 (基本类型). Kotlin 编译产生的字节码会尽可能使用基本数据类型, 但在 Kotlin 源代码中并不能明确的使用基本数据类型.
- 静态成员 (类) 在 Kotlin 中由以下功能代替: 同伴对象(Companion Object) ("同伴对象(Companion Object)" in "对象表达式,对象声明,以及同伴对象"), 顶级(top-level) 函数 (函数), 扩展(extension) 函数 ("扩展函数(Extension Function)" in "扩展"), 以及 @JvmStatic 注解 ("静态方法(Static Method)" in "在 Java 中调用 Kotlin 代码").
- 通配符类型(Wildcard-type) (泛型(Generic): in, out, where) 在 Kotlin 中由以下功能代替: 声明处类型变异(declaration-site variance) ("声明处的类型变异(Declaration-site variance)" in "泛

型(Generic): in, out, where") 以及 类型投射(type projection) ("类型投射(Type projection)" in "泛型(Generic): in, out, where").

- 条件(三元)运算符 `a ? b : c` ("if 表达式" in "条件与循环") 在 Kotlin 中由以下功能代替: if 表达式 ("if 表达式" in "条件与循环").
- 记录类(Record) (https://openjdk.org/jeps/395)
- 记录模式(Record Pattern) (https://openjdk.org/jeps/440)
- **Java 22**: 未命名变量和模式 (https://openjdk.org/jeps/456)

Kotlin 中有, 而 Java 中没有的功能

- Lambda 表达式 (高阶函数与 Lambda 表达式) + 内联函数 (内联函数(Inline Function)) = 实现自定义的控制结构
- 扩展函数 (扩展)
- Null 值安全性 (Null 值安全性)
- 类型智能转换 (类型检查与类型转换) (**Java 16**: 对 instanceof 的模式匹配 (https://openjdk.org/jeps/394))
- 字符串模板 (字符串) (**Java 21**: 字符串模板 (预览版) (https://openjdk.org/jeps/430))
- 属性 (属性(Property))
- 主构造器 (类)
- 委托(First-class delegation) (委托)
- 变量和属性的类型推断 (基本类型) (**Java 10**: 局部变量的类型推断 (https://openjdk.org/jeps/286))
- 单例(Singleton) (对象表达式,对象声明,以及同伴对象)
- 声明处类型变异(Declaration-site variance) 和类型投射(Type projection) (泛型(Generic): in, out, where)
- 值范围表达式 (值范围(Range)与数列(Progression))
- 操作符重载 (操作符重载)

- 同伴对象(Companion object) ("同伴对象(Companion Object)" in "类")
- 数据类 (数据类(Data Class))
- 协程 (协程(Coroutine))
- 顶级(Top Level)函数 (函数)
- 默认参数 ("默认参数" in "函数")
- 命名参数 ("命名参数" in "函数")
- 中缀(Infix)函数 ("中缀标记法(Infix notation)" in "函数")
- 预期声明与实际声明 (预期声明与实际声明)

下一步做什么?

学习:

- 如何执行 Java 与 Kotlin 中常见的字符串处理任务 (Java 和 Kotlin 中的字符串).
- 如何执行 Java 与 Kotlin 中常见的集合(Collection)处理任务 (Java 和 Kotlin 中的集合(Collection)).
- 如何 在 Java 与 Kotlin 中处理可空性(Nullability) (Java 和 Kotlin 中的可空性(Nullability)).

在 Kotlin 中调用 Java 代码

最终更新: 2024/09/10

Kotlin 的设计过程中就考虑到了与 Java 的互操作性. 在 Kotlin 中可以通过很自然的方式调用既有的 Java 代码, 反过来在 Java 中也可以很流畅地使用 Kotlin 代码. 本章中我们介绍在 Kotlin 中调用 Java 代码的一些细节问题.

大多数 Java 代码都可以直接使用, 没有任何问题:

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 对 Java 集合使用 'for' 循环:
    for (item in source) {
        list.add(item)
    }
    // 也可以对 Java 类使用 Kotlin 的操作符:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // 这里会调用 get 和 set 方法
    }
}
```

Get 和 Set 方法

符合 Java 的 Get 和 Set 方法规约的方法(无参数, 名称以 `get` 开头, 或单个参数, 名称以 `set` 开头) 在 Kotlin 中会被识别为属性. 这些属性也被称为 *合成属性(Synthetic Property)*. `Boolean` 类型的属性访问方法(Get 方法名称以 `is` 开头, Set 方法名称以 `set` 开头), 会被识别为属性, 其名称与 Get 方法相同.

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // 这里会调用
getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY // 这里会调用
```

```
setFirstDayOfWeek()
    }
    if (!calendar.isLenient) { // 这里会调用 isLenient()
        calendar.isLenient = true // 这里会调用 setLenient()
    }
}
```

上面的 `calendar.firstDayOfWeek` 就是合成属性的一个例子.

注意, 如果 Java 类中只有 set 方法, 那么在 Kotlin 中不会被识别为属性, 因为 Kotlin 不支持只写 (set-only) 的属性.

Java 合成属性(Synthetic Property)的引用

> ⚠ 这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 我们建议你只为评估和试验目的来使用这个功能..

从 Kotlin 1.8.20 开始, 你可以创建 Java 合成属性的引用. 考虑下面的 Java 代码:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Kotlin 允许你使用 `person.age`, 其中 `age` 是一个合成属性. 现在, 你也可以创建 `Person::age` 和 `person::age` 的引用. 对 `name` 属性也是如此.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12),
Person("Peter", 11))
    Persons
         // 调用 Java 合成属性的引用:
        .sortedBy(Person::age)
         // 通过 Kotlin 属性语法调用 Java 的 get 方法:
        .forEach { person -> println(person.name) }
}
```

如何启动用 Java 合成属性的引用

要启用这个功能, 请设置 `-language-version 2.1` 编译器选项. 在 Gradle 项目中, 你可以在你的 `build.gradle(.kts)` 文件中添加以下设置:

Kotlin

```
tasks

.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_1
            )
    }
```

Groovy

```
tasks

.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
```

```
k.class)
    .configureEach {
        compilerOptions.languageVersion
            =
org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_1
}
```

> 在 Kotlin 1.9.0 以前的版本中, 要启用这个功能, 你必须设置 `-language-version 1.9` 编译器选项.

返回值为 void 的方法

如果一个 Java 方法返回值为 `void`, 那么在 Kotlin 中调用时将返回 `Unit`. 如果, 在 Kotlin 中使用了返回值, 那么会由 Kotlin 编译器在调用处赋值, 因为返回值已经预先知道了(等于 `Unit`).

当 Java 标识符与 Kotlin 关键字重名时的转义处理

某些 Kotlin 关键字在 Java 中是合法的标识符: `in`, `object`, `is`, 等等. 如果 Java 类库中使用 Kotlin 的关键字作为方法名, 你仍然可以调用这个方法, 只要使用反引号(`)对方法名转义即可:

```
foo.`is`(bar)
```

Null 值安全性与平台数据类型

Java 中的所有引用都可以为 `null` 值, 因此对于来自 Java 的对象, Kotlin 的严格的 null 值安全性要求就变得毫无意义了. Java 中定义的类型在 Kotlin 中会被特别处理, 被称为 *平台数据类型 (platform type)*. 对于这些类型, Null 值检查会被放松, 因此对它们来说, 只提供与 Java 中相同的 null 值安全保证(详情参见下文).

我们来看看下面的例子:

```
val list = ArrayList<String>() // 非 null 值 (因为是构造器方法的返回结果)
list.add("Item")
val size = list.size // 非 null 值 (因为是基本类型 int)
val item = list[0] // 类型自动推断结果为平台类型 (通常的 Java 对象)
```

对于平台数据类型的变量, 当你调用它的方法时, Kotlin 不会在编译时刻报告可能为 null 的错误, 但这个调用在运行时可能失败, 原因可能是发生 null 指针异常, 也可能是 Kotlin 编译时为防止 null 值错误而产生的断言, 在运行时导致失败:

```
item.substring(1) // 编译时允许这样的调用, 但在运行时如果 item == null 则会抛出异常
```

平台数据类型是 *无法指示的(non-denotable)*, 也就是说你不能在语言中明确指出这样的类型. 当平台数据类型的值赋值给 Kotlin 变量时, 你可以依靠类型推断 (这时变量的类型会被自动推断为平台数据类型, 比如上面示例程序中的变量 `item` 就是如此), 或者你也可以选择期望的数据类型(可为 null 的类型和非 null 类型都允许):

```
val nullable: String? = item // 允许, 永远不会发生错误
val notNull: String = item // 允许, 但在运行时刻可能失败
```

如果你选择使用非 null 类型, 那么编译器会在赋值处理之前输出一个断言(assertion). 它负责防止 Kotlin 中的非 null 变量指向一个 null 值. 比如, 当你将平台数据类型的值传递给 Kotlin 函数的非 null 值参数时, 也会输出断言, 其他情况也会类似处理. 总之, 编译器会尽可能地防止 null 值错误在程序中扩散, 然而, 有些时候由于泛型的存在, 不可能完全消除这种错误.

对平台数据类型的注解

上文中我们提到, 平台数据类型无法在程序中明确指出, 因此在 Kotlin 语言中没有专门的语法来表示这种类型. 然而, 有时编译器和 IDE 仍然需要表示这些类型(比如, 在错误消息中, 在参数信息中), 因此, 有一种助记用的注解:

- `T!` 代表 " `T` 或者 `T?` ",
- `(Mutable)Collection<T>!` 代表 "元素类型为 `T` 的Java 集合, 内容可能可变, 也可能不可变, 值可能允许为 null, 也可能不允许为 null",
- `Array<(out) T>!` 代表 "元素类型为 `T` (或 `T` 的子类型)的 Java 数组, 值可能允许为 null, 也可能不允许为 null"

可否为 null(Nullability) 注解

带有可否为 null(Nullability) 注解的 Java 类型在 Kotlin 中不会被当作平台数据类型, 而会被识别为可为 null 的, 或非 null 的 Kotlin 类型. 编译器支持几种不同风格的可否为 null 注解, 包括:

- JetBrain (https://www.jetbrains.com/idea/help/nullable-and-notnull-annotations.html) (`org.jetbrains.annotations` 包中定义的 `@Nullable` 和 `@NotNull` 注解)
- JSpecify (https://jspecify.dev/) (`org.jspecify.nullness`)
- Android (`com.android.annotations` 和 `android.support.annotations`)
- JSR-305 (`javax.annotation`, 详情请参见下文)
- FindBugs (`edu.umd.cs.findbugs.annotations`)
- Eclipse (`org.eclipse.jdt.annotation`)
- Lombok (`lombok.NonNull`)
- RxJava 3 (`io.reactivex.rxjava3.annotations`)

根据指定的可否为 null(Nullability) 注解信息, 编译器可以发现可否为 null(Nullability) 的设定不匹配错误, 你可以指定编译器是否报告这种错误. 请使用编译器选项 `-Xnullability-annotations=@<package-name>:<report-level>`. 在选项的参数中, 请指定可否为 null(Nullability) 注解的完整限定包名称, 以及以下错误报告等级:

- `ignore`, 忽略可否为 null(Nullability) 设定的不匹配错误
- `warn`, 报告为警告 to report warnings
- `strict`, 报告为错误.

Koltin 支持的可否为 null(Nullability) 注解的完整列表请参见 Kotlin 编译器源代码 (https://github.com/JetBrains/kotlin/blob/master/core/compiler.common.jvm/src/org/jetbrains/kotlin/load/java/JvmAnnotationNames.kt).

对类型参数添加注解

你也可以对泛型类型参数的实参(Type argument)和形参(Type parameter)添加注解, 标记它可否为 null.

> 本章所有示例程序都使用 JetBrain 的, `org.jetbrains.annotations` 包中的可否为 null 注解.

类型参数实参(Type argument)

比如, 在 Java 中添加这些注解:

```
@NotNull
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String>
elements) { ... }
```

这些注解使得 Kotlin 中的函数签名如下:

```
fun toSet(elements: (Mutable)Collection<String>) :
(Mutable)Set<String> { ... }
```

如果类型参数实参中缺少 `@NotNull` 注解, 你得到的将会是平台数据类型:

```
fun toSet(elements: (Mutable)Collection<String!>) :
(Mutable)Set<String!> { ... }
```

Kotlin 还会考虑基类和接口的类型参数上的 可否为 null 注解. 比如, 有 2 个 Java 类, 定义如下:

```
public class Base<T> {}
```

```
public class Derived extends Base<@Nullable String> {}
```

在 Kotlin 代码中, 在需要 `Base<String>` 的地方传递 `Derived` 的实例会导致警告.

```
fun takeBaseOfNotNullStrings(x: Base<String>) {}

fun main() {
    takeBaseOfNotNullStrings(Derived()) // 警告: 可否为 null 设置不匹配
}
```

`Derived` 的上界(Upper Bound) 被设置为 `Base<String?>`, 而不是 `Base<String>`.

更多详情请参考 在 Kotlin 中使用 Java 的泛型.

类型参数形参(Type parameter)

默认情况下, 通常的类型参数形参(Type parameter)可否为 null, 在 Kotlin 和 Java 中都没有定义. 在 Java 中, 你可以使用 可否为 null 注解来指定. 我们来为 `Base` 类的类型参数形参添加注解:

```
public class Base<@NotNull T> {}
```

继承 `Base` 时, Kotlin 会期待非 null 的类型参数实参(Type argument)或形参(Type parameter). 因此, 以下 Kotlin 代码会出现警告:

```
class Derived<K> : Base<K> {} // 警告: 未定义 K 可否为 null
```

你可以指定上界 `K : Any` 来修正这个问题.

Kotlin 还支持对 Java 类型参数上界添加可否为 null 注解. 我们来为 `Base` 类添加上界:

```
public class BaseWithBound<T extends @NotNull Number> {}
```

Kotlin 会翻译为:

```
class BaseWithBound<T : Number> {}
```

因此对类型参数实参(Type argument)或形参(Type parameter)传递可为 null 的类型会导致警告.

类型参数实参(Type argument)和形参(Type parameter)的注解需要 Java 8 或更高版本的编译环境. 还要求可否为 null 注解的 target 支持 `TYPE_USE` (从版本 15 开始, `org.jetbrains.annotations` 支持 `TYPE_USE`). 如果 Kotlin 代码使用的可否为 null 设置与 Java 中的注解不一致, 使用编译器选项 `-Xtype-enhancement-improvements-strict-mode` 可以报告这类错误.

> 注意: 如果可否为 null 注解除 `TYPE_USE` 之外还支持适用于类型的其他 target, 那么会优先使用 `TYPE_USE`. 比如, 如果 `@Nullable` 的 target 包括 `TYPE_USE` 和 `METHOD`, 那么 Java 方法签名 `@Nullable String[] f()` 在 Kotlin 中会成为 `fun f(): Array<String?>!`.

对 JSR-305 规范的支持

JSR-305 规范 (https://jcp.org/en/jsr/detail?id=305) 中定义了 `@Nonnull` (https://www.javadoc.io/doc/com.google.code.findbugs/jsr305/latest/javax/annotation/Nonnull.html) 注解. Kotlin 支持使用这个注解来标识 Java 类型可否为 null.

如果 `@Nonnull(when = ...)` 的值为 `When.ALWAYS`, 那么被注解的类型会被当作不可为 null 的; `When.MAYBE` 和 `When.NEVER` 对应于可为 null 的类型; `When.UNKNOWN` 则会被认为是 平台数据类型.

库编译时可以用到 JSR-305 规范的注解, 但对于库的使用者来说, 编译时不必依赖这些注解的 jar 文件(比如 `jsr305.jar`). Kotlin 编译器可以从库中读取 JSR-305 规范的注解, 而不需要这些注解存在

于类路径中.

此外还支持 自定义可空限定符 (KEEP-79) (https://github.com/Kotlin/KEEP/blob/master/proposals/jsr-305-custom-nullability-qualifiers.md) (详情请见下文).

类型限定符别名(Type qualifier nickname)

如果一个注解, 同时标注了 `@TypeQualifierNickname` (https://www.javadoc.io/doc/com.google.code.findbugs/jsr305/latest/javax/annotation/meta/TypeQualifierNickname.html) 注解 和 JSR-305 规范的 `@Nonnull` 注解(或者它的另一个别名, 比如 `@CheckForNull`), 那么这个注解可以用来标注类型是否可以为 null, 其含义与 JSR-305 规范的 `@Nonnull` 注解完全相同:

```
@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
@CheckForNull // 另一个 TypeQualifierNickname 的别名
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@MyNonnull String x);
    // Kotlin (strict 模式) 中会被看作: `fun foo(x: String): String?`

    String bar(List<@MyNonnull String> x);
    // Kotlin (strict 模式) 中会被看作: `fun bar(x: List<String>!):
String!`
}
```

类型限定符默认值(Type qualifier default)

`@TypeQualifierDefault` (https://www.javadoc.io/doc/com.google.code.findbugs/jsr305/latest/javax/annotation/me

ta/TypeQualifierDefault.html) 用来定义一个注解, 当使用这个注解时, 可以在被标注的元素的范围内, 定义默认的可否为 null 设定.

这种注解本身应该标注 `@Nonnull` 注解(或者使用它的别名), 并使用一个或多个 `ElementType` 值标注 `@TypeQualifierDefault(...)` 注解:

- `ElementType.METHOD` 表示注解对象为方法的返回值
- `ElementType.PARAMETER` 表示注解对象为参数值
- `ElementType.FIELD` 表示注解对象为类的成员域变量
- `ElementType.TYPE_USE` 表示注解对象为任何类型, 包含类型参数(type argument), 类型参数上界(Upper Bound), 以及通配符类型(wildcard type)

当一个类型没有标注可否为 null 注解时, 会使用默认的可否为 null 设定, Kotlin 会查找对象类型所属的最内层的元素, 要求这个元素使用了类型限定符默认值注解, 而且 `ElementType` 值与对象类型相匹配, 然后通过类型限定符默认值注解, 得到这个默认的可否为 null 设定.

```
@Nonnull
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NonNullApi {
}

@Nonnull(when = When.MAYBE)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER,
ElementType.TYPE_USE})
public @interface NullableApi {
}

@NullableApi
interface A {
    String foo(String x); // 在 Kotlin 中会被看作: fun foo(x:
String?): String?

    @NotNullApi // 这个注解将会覆盖接口上的可否为 null 默认设定
    String bar(String x, @Nullable String y); // 在 Kotlin 中会被看作:
fun bar(x: String, y: String?): String

    // List<String> 的类型参数会被看作可为 null,
```

```
    // 因为 `@NullableApi` 中包括了 `TYPE_USE` ElementType:
    String baz(List<String> x); // 在 Kotlin 中会被看作: fun
baz(List<String?>?): String?

    // 参数 `x` 的类型为平台类型, 因为它的可否为 null 注解明确标注为
UNKNOWN:
    String qux(@Nonnull(when = When.UNKNOWN) String x); // 在 Kotlin
中会被看作: fun baz(x: String!): String?
}
```

> 上面示例程序中的类型只在 strict 编译模式下才有效, 否则, Kotlin 会将它们识别为平台类型. 详情请参见本章的 `@UnderMigration` 注解小节 以及 编译器配置小节.

另外还支持包级别的可否为 null 默认设定:

```
// FILE: test/package-info.java
@NonNullApi // 'test' 包内的所有声明, 默认都是非 null
package test;
```

@UnderMigration 注解

库的维护者可以使用 `@UnderMigration` 注解 (由独立的库文件 `kotlin-annotations-jvm` 提供), 来定义可否为空(nullability)类型标识符的迁移状态.

如果不正确地使用了被注解的类型(比如, 把一个标注了 `@MyNullable` 的类型值当作非空类型来使用), `@UnderMigration(status = ...)` 注解中的 status 值指定编译器应当如何处理:

- `MigrationStatus.STRICT`: 让注解象任何通常的可否为空(nullability)注解那样工作, 也就是, 对不正确的使用报告错误, 并且影响 Kotlin 对被注解类型的识别
- `MigrationStatus.WARN`: 不正确的使用在编译时会被报告为警告, 而不是错误, 但被注解的声明中的类型, 在 Kotlin 中会被识别为平台类型
- `MigrationStatus.IGNORE`: 让编译器完全忽略可否为空(nullability)注解

库的维护者可以对类型限定符别名(Type qualifier nickname), 以及类型限定符默认值(Type qualifier default), 指定 `@UnderMigration` 的 status 值:

```
@Nonnull(when = When.ALWAYS)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
@UnderMigration(status = MigrationStatus.WARN)
public @interface NonNullApi {
}

// 这个类中的类型将是非空的, 但编译时只会报告警告
// 因为对 `@NonNullApi` 添加了 `@UnderMigration(status =
MigrationStatus.WARN)` 注解
@NonNullApi
public class Test {}
```

> 一个可否为空(nullability)注解的 MigrationStatus 值, 不会被它的类型限定符别名继承, 但在使用时类型限定符默认值会有效.

如果一个类型限定符默认值使用了一个类型限定符别名, 而且他们都添加了 `@UnderMigration` 注解, 这时会优先使用类型限定符默认值中的 MigrationStatus.

编译器配置

可以添加 `-Xjsr305` 编译器选项来配置 JSR-305 规范检查, 这个编译器选项可以使用以下设置之一(或者多个设置的组合):

- `-Xjsr305={strict|warn|ignore}` 用来设置非 `@UnderMigration` 注解的行为. 自定义的可否为空注解, 尤其是 `@TypeQualifierDefault`, 已经大量出现在很多知名的库中, 当使用者升级到支持 JSR-305 Kotlin 版本时, 可能会需要平滑地迁移这些库. 从 Kotlin 1.1.60 开始, 这个设置值影响非 `@UnderMigration` 的注解.
- `-Xjsr305=under-migration:{strict|warn|ignore}` 用来覆盖 `@UnderMigration` 注解的行为. 对于库的迁移状态, 库的使用者可能会存在不用的看法: 当库的作者发布的官方迁移状态为 `WARN` 时, 库的使用者却可能希望报告编译错误, 或者反过来, 他们也可能希望对于某些代码暂时不要报告编译错误, 直到他们完成迁移.
- `-Xjsr305=@<fq.name>:{strict|warn|ignore}` 用来覆盖单个注解的行为, 这里的 `<fq.name>` 是注解的完整限定类名(fully qualified class name). 对于不同的注解, 可以多次指定这个编译选项. 对于管理某个特定库的迁移状态, 这个编译选项非常有用.

这里的 `strict`, `warn` 以及 `ignore` 值, 与 `MigrationStatus` 中对应值的意义完全相同, 而且只有 `strict` 模式会影响 Kotlin 对被注解的声明中的类型的识别.

> 注意: 无论 `-Xjsr305` 编译器选项的设置如何, JSR-305 内置的注解 `@Nonnull` (https://www.javadoc.io/doc/com.google.code.findbugs/jsr305/latest/javax/annotation/Nonnull.html), `@Nullable` (https://www.javadoc.io/doc/com.google.code.findbugs/jsr305/3.0.1/javax/annotation/Nullable.html) 以及 `@CheckForNull` (https://www.javadoc.io/doc/com.google.code.findbugs/jsr305/latest/javax/annotation/CheckForNull.html) 始终是有效的, 并且会影响 Kotlin 对被注解声明中的类型的识别.

比如, 如果在编译器参数中添加 `-Xjsr305=ignore -Xjsr305=under-migration:ignore -Xjsr305=@org.library.MyNullable:warn`, 对于被 `@org.library.MyNullable` 注解的类型, 如果存在不正确的使用, 此时编译器会报告警告, 但对于 JSR-305 的所有注解, 则会忽略这种不正确的使用.

编译器的默认行为与 `-Xjsr305=warn` 一样. 目前 `strict` 设定还是实验性的 (未来可能会增加更多的检查).

数据类型映射

Kotlin 会对某些 Java 类型进行特殊处理. 这些类型会被从 Java 中原封不动地装载进来, 但被 *映射* 为对应的 Kotlin 类型. 映射过程只会在编译时发生, 运行时的数据表达不会发生变化. Java 的基本数据类型会被映射为对应的 Kotlin 类型(但请注意 平台数据类型 问题):

Java 类型	Kotlin 类型
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

有些内建类虽然不是基本类型, 也会被映射为对应的 Kotlin 类型:

Java 类型	Kotlin 类型
`java.lang.Object`	`kotlin.Any!`
`java.lang.Cloneable`	`kotlin.Cloneable!`
`java.lang.Comparable`	`kotlin.Comparable!`
`java.lang.Enum`	`kotlin.Enum!`
`java.lang.annotation.Annotation`	`kotlin.Annotation!`
`java.lang.CharSequence`	`kotlin.CharSequence!`
`java.lang.String`	`kotlin.String!`
`java.lang.Number`	`kotlin.Number!`
`java.lang.Throwable`	`kotlin.Throwable!`

Java 中的装箱的基本类型(boxed primitive type), 会被映射为 Kotlin 的可为 null 类型:

Java 类型	Kotlin 类型
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Character	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

注意, 装箱的基本类型用作类型参数时, 会被映射为平台类型: 比如, `List<java.lang.Integer>` 在 Kotlin 中会变为 `List<Int!>`.

集合类型在 Kotlin 中可能是只读的, 也可能是内容可变的, 因此 Java 的集合会被映射为以下类型(下表中所有的 Kotlin 类型都属于 `kotlin.collections` 包):

Java 类型	Kotlin 只读类型	Kotlin 内容可变类型	被装载的平台数据类型
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable)Iterator<T>!
Iterable<T>	Iterable<T>	MutableIterable<T>	(Mutable)Iterable<T>!
Collection<T>	Collection<T>	MutableCollection<T>	(Mutable)Collection<T>!
Set<T>	Set<T>	MutableSet<T>	(Mutable)Set<T>!
List<T>	List<T>	MutableList<T>	(Mutable)List<T>!
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)ListIterator<T>!
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map<K, V>!
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K,V>	(Mutable)Map.(Mutable)Entry<K, V>!

Java 数据的映射如下, 详情参见 下文:

Java 类型	Kotlin 类型
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

这些 Java 类型的静态成员, 无法通过 Kotlin 类型的同伴对象 ("同伴对象(Companion Object)" in "对象表达式,对象声明,以及同伴对象")直接访问. 要访问这些静态成员, 需要使用 Java 类型的完整限定名称, 比如 java.lang.Integer.toHexString(foo).

在 Kotlin 中使用 Java 的泛型

Kotlin 的泛型 与 Java 的泛型略有差异 (参见 泛型 (泛型(Generic): in, out, where)). 将 Java 类型导入 Kotlin 时, 会进行以下变换:

- Java 的通配符会被变换为 Kotlin 的类型投射:
  - `Foo<? extends Bar>` 变换为 `Foo<out Bar!>!`
  - `Foo<? super Bar>` 变换为 `Foo<in Bar!>!`
- Java 的原生类型(raw type) 转换为 Kotlin 的星号投射(star projection):
  - `List` 变换为 `List<*>!`, 也就是 `List<out Any?>!`

与 Java 一样, Kotlin 的泛型信息在运行时不会保留: 创建对象时传递给构造器的类型参数信息, 在对象中不会保留下来. 比如, `ArrayList<Integer>()` 与 `ArrayList<Character>()` 在运行时刻是无法区分的. 这就导致无法进行带有泛型信息的 `is` 判断. Kotlin 只允许对星号投射(star projection)的泛型类型进行 `is` 判断:

```
if (a is List<Int>) // 错误: 无法判断它是不是 Int 构成的 List
// 但是
if (a is List<*>) // OK: 这里的判断不保证 List 内容的数据类型
```

Java 数组

与 Java 不同, Kotlin 中的数组是不可变的(invariant). 这就意味着, Kotlin 不允许你将 `Array<String>` 赋值给 `Array<Any>` ,这样就可以避免发生运行时错误. 在调用 Kotlin 方法时, 如果参数声明为父类型的数组, 那么将子类型的数组传递给这个参数, 也是禁止的, 但对于 Java 的方法, 通过使用 `Array<(out) String>!` 形式的平台数据类型, 这是允许.

在 Java 平台上, 会使用基本类型构成的数组, 以避免装箱(boxing)/拆箱(unboxing)操作带来的性能损失. 由于 Kotlin 会隐藏这些实现细节, 因此与 Java 代码交互时需要使用一个替代办法. 对于每种基本类型, 都存在一个专门的类(`IntArray`, `DoubleArray`, `CharArray`, 等等) 来解决这种问题. 这些类与 `Array` 类没有关系, 而且会被编译为 Java 的基本类型数组, 以便达到最好的性能.

假设有一个 Java 方法, 接受一个名为 indices 的参数, 类型是 int 数组:

```
public class JavaArrayExample {
    public void removeIndices(int[] indices) {
```

```
        // 方法代码在这里...
    }
}
```

为了向这个方法传递一个基本类型值构成的数组, 在 Kotlin 中你可以编写下面的代码:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // 向方法传递 int[] 参数
```

编译输出 JVM 字节码时, 编译器会对数组的访问处理进行优化, 因此不会产生性能损失:

```
val array = arrayOf(1, 2, 3, 4)
array[1] = array[1] * 2 // 编译器不会产生对 get() 和 set() 方法的调用
for (x in array) { // 不会创建迭代器(iterator)
    print(x)
}
```

即使你使用下标来访问数组元素, 也不会产生任何性能损失:

```
for (i in array.indices) { // 不会创建迭代器(iterator)
    array[i] += 2
}
```

最后, `in` 判断也不会产生性能损失:

```
if (i in array.indices) { // 等价于 (i >= 0 && i < array.size)
    print(array[i])
}
```

Java 的可变长参数(Varargs)

Java 类的方法声明有时会对 indices 使用可变长的参数定义(varargs):

```
public class JavaArrayExample {

    public void removeIndicesVarArg(int... indices) {
        // 方法代码在这里...
```

```
    }
}
```

这种情况下, 为了将 IntArray 传递给这个参数, 需要使用展开(spread) * 操作符:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

操作符

由于 Java 中无法将方法标记为操作符重载方法, Kotlin 允许我们使用任何的 Java 方法, 只要方法名称和签名定义满足操作符重载的要求, 或者满足其他规约(invoke() 等等.) 使用中缀调用语法来调用 Java 方法是不允许的.

受控异常(Checked Exception)

在 Kotlin 中, 所有的异常都是不受控的(unchecked) (异常(Exception)), 也就是说编译器不会强制要求你捕获任何异常. 因此, 当调用 Java 方法时, 如果这个方法声明了受控异常, Kotlin 不会要求你做任何处理:

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // Java 会要求我们在这里捕获
IOException
    }
}
```

Object 类的方法

当 Java 类型导入 Kotlin 时, 所有 java.lang.Object 类型的引用都会被转换为 Any 类型. 由于 Any 类与具体的实现平台无关, 因此它声明的成员方法只有 toString(), hashCode() 和 equals(), 所以, 为了补足 java.lang.Object 中的其他方法, Kotlin 使用了 扩展函数 (扩展).

wait()/notify()

对 Any 类型的引用不能使用 wait() 和 notify() 方法, 通常也不建议使用这些方法, 而应该改用 java.util.concurrent 中的功能来替代. 如果你确实需要调用这些方法, 那么可以先将它变换为 java.lang.Object 类型:

```
(foo as java.lang.Object).wait()
```

getClass()

要得到一个对象的 Java Class 信息, 可以使用 类引用 ("类引用(Class Reference)" in "反射") 的 `java` 扩展属性:

```
val fooClass = foo::class.java
```

上面的示例程序中, 使用了一个 与对象实例绑定的类引用 ("与对象实例绑定的类引用语法" in "反射"). 你也可以使用 `javaClass` 扩展属性:

```
val fooClass = foo.javaClass
```

clone()

要覆盖 `clone()` 方法, 你的类需要实现 `kotlin.Cloneable` 接口:

```
class Example : Cloneable {
    override fun clone(): Any { ... }
}
```

别忘了 Effective Java, 第 3 版 (https://www.oracle.com/technetwork/java/effectivejava-136174.html), 第 13 条: *要正确地覆盖 clone 方法*.

finalize()

要覆盖 `finalize()` 方法, 你只需要声明它即可, 不必使用 `override` 关键字:

```
class C {
    protected fun finalize() {
        // finalization 处理逻辑
    }
}
```

按照 Java 的规则, `finalize()` 不能是 `private` 方法.

继承 Java 的类

Kotlin 类的超类中, 最多只能指定一个 Java 类(Java 接口的数量没有限制).

访问静态成员(static member)

Java 类的静态成员(static member)构成这些类的"同伴对象(companion object)". 你不能将这样的"同伴对象" 当作值来传递, 但可以明确地访问它的成员, 比如:

```
if (Character.isLetter(a)) { ... }
```

当一个 Java 类型映射为一个 Kotlin 类型时, 如果要访问其中的静态成员, 需要使用 Java 类型的完整限定名: `java.lang.Integer.bitCount(foo)`.

Java 的反射

Java 的反射在 Kotlin 类中也可以使用, 反过来也是如此. 我们在上文中讲到, 你可以使用 `instance::class.java`, `ClassName::class.java`, 或者 `instance.javaClass`, 得到 `java.lang.Class`, 然后通过它就可以使用 Java 的反射功能. 这里不要使用 `ClassName.javaClass`, 因为它引用的是 `ClassName` 的同伴对象的类, 也就是 `ClassName.Companion::class.java`, 而不是 `ClassName::class.java`.

对每种基本类型, 有两种不同的 Java 类, Kotlin 对这两种类都提供了取得方法, 比如 `Int::class.java` 会返回表示基本类型本身的类实例, 对应于 Java 中的 `Integer.TYPE`. 要取得对应的包装对象(wrapper type)的类, 请使用 `Int::class.javaObjectType`, 等于 Java 的 `Integer.class`.

此外还支持其他反射功能, 比如可以得到 Kotlin 属性对应的 Java get/set 方法或后端成员, 可以得到 Java 成员变量对应的 `KProperty`, 得到 `KFunction` 对应的 Java 方法或构造器, 或者反过来得到 Java 方法或构造器对应的 `KFunction`.

SAM 转换

Kotlin 支持 Java 和 Kotlin 接口 (函数式 (SAM) 接口) 的 SAM(Single Abstract Method) 转换. 支持 Java 的 SAM 转换就是说, 如果一个 Java 接口中仅有一个方法, 并且没有默认实现, 那么只要 Java 接口方法与 Kotlin 函数参数类型一致, Kotlin 的函数字面值就可以自动转换为这个接口的实现者.

你可以使用这个功能来创建 SAM 接口的实例:

```
val runnable = Runnable { println("This runs in a runnable") }
```

...也可以用在方法调用中:

```
val executor = ThreadPoolExecutor()
// Java 方法签名: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

如果 Java 类中有多个同名的方法, 而且方法参数都可以接受函数式接口, 那么你可以使用一个适配器函数(adapter function), 将 Lambda 表达式转换为某个具体的 SAM 类型, 然后就可以选择需要调用的方法. 编译器也会在需要的时候生成这些适配器函数:

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

SAM 转换只对接口有效, 不能用于抽象类, 即使抽象类中仅有唯一一个抽象方法.

在 Kotlin 中使用 JNI(Java Native Interface)

要声明一个由本地代码(C 或者 C++)实现的函数, 你需要使用 `external` 修饰符标记这个函数:

```
external fun foo(x: Int): Double
```

剩下的工作与 Java 中完全相同.

还可以将属性的取得方法和设值方法标记为 `external`:

```
var myProperty: String
    external get
    external set
```

这段代码会创建两个函数 `getMyProperty` 和 `setMyProperty`, 并且都标记为 `external`.

在 Kotlin 中使用 Lombok 生成的声明

在 Kotlin 代码中你可以使用 Java 代码由 Lombok 生成的声明. 如果你需要在 Java/Kotlin 代码混合的同一个模块中生成并使用这些声明, 具体方法请参见 Lombok 编译器插件 (Lombok 编译器插件). 如果你要在其他模块中使用这些声明, 那么不需要使用这个插件来编译这个模块.

在 Java 中调用 Kotlin 代码

最终更新: 2024/09/10

在 Java 中可以很容易地调用 Kotlin 代码. 比如, 在 Java 方法中可以非常自然的创建 Kotlin 类的实例, 并操作这些实例. 但是 Java 和 Kotlin 之间还是存在一些差异, 在 Java 中集成 Kotlin 代码时, 需要注意这些问题. 本章中, 我们将会介绍在 Java 代码中如何调用 Kotlin 代码.

属性

Kotlin 的属性会被编译为以下 Java 元素:

- 一个取值方法, 方法名由属性名加上 `get` 前缀得到
- 一个设值方法, 方法名由属性名加上 `set` 前缀得到 (只会为 `var` 属性生成设值方法)
- 一个私有的域变量, 名称与属性名相同 (只会为拥有后端域变量的属性生成域变量)

比如, `var firstName: String` 编译后的结果等于以下 Java 声明:

```
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

如果属性名以 `is` 开头, 会使用另一种映射规则: 取值方法名称会与属性名相同, 设值方法名称等于将属性名中的 `is` 替换为 `set`. 比如, 对于 `isOpen` 属性, 取值方法名称将会是 `isOpen()`, 设值方法名称将会是 `setOpen()`. 这个规则适用于任何数据类型的属性, 而不仅限于 `Boolean` 类型.

包级函数

在源代码文件 `app.kt` 的 `org.example` 包内声明的所有函数和属性, 包括扩展函数, 都被会编译成为 Java 类 `org.example.AppKt` 的静态方法.

```
// 源代码文件: app.kt
package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java 代码
new org.example.Util();
org.example.AppKt.getTime();
```

要改变编译生成的 Java 类的名称, 可以使用 `@JvmName` 注解:

```
@file:JvmName("DemoUtils")

package org.example

class Util

fun getTime() { /*...*/ }
```

```
// Java 代码
new org.example.Util();
org.example.DemoUtils.getTime();
```

如果多个源代码文件生成的 Java 类名相同(由于文件名和包名都相同, 或由于使用了相同的 `@JvmName` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-name/index.html) 注解) 这样的情况通常会被认为是错误. 但是, 编译器能够使用指定的名称生成单个 Java Facade 类, 其中包含所有源代码文件的所有内容. 要生成这样的 Facade 类, 可以在多个源代码文件中使用 `@JvmMultifileClass` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-multifile-class/index.html) 注解.

```
// 源代码文件: oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass
```

```
package org.example

fun getTime() { /*...*/ }
```

```
// 源代码文件: newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getDate() { /*...*/ }
```

```
// Java 代码
org.example.Utils.getTime();
org.example.Utils.getDate();
```

实例的域

如果希望将一个 Kotlin 属性公开为 Java 中的一个域, 需要对它添加 `@JvmField` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-field/index.html) 注解. 生成的域的可见度将与属性可见度一样. 要对属性使用 `@JvmField` 注解, 属性需要满足以下条件:

- 拥有后端域变量(backing field)
- 不是 private 属性
- 没有 `open`, `override` 或 `const` 修饰符
- 不是委托属性(delegated property)

```
class User(id: String) {
    @JvmField val ID = id
}
```

```
// Java 代码
class JavaClient {
    public String getID(User user) {
```

```
        return user.ID;
    }
}
```

延迟初始化属性 ("延迟初始化的(Late-Initialized)属性和变量" in "属性(Property)") 也会公开为 Java 中的域. 域的可见度将与属性的 `lateinit` 的设值方法可见度一样.

静态域(Static Fields)

声明在命名对象(named object)或同伴对象(companion object)之内的 Kotlin 属性, 将会存在静态的后端域变量(backing field), 对于命名对象, 静态后端域变量存在于命名对象内, 对于同伴对象, 静态后端域变量存在包含同伴对象的类之内.

通常这些静态的后端域变量是 private 的, 但可以使用以下方法来公开它:

- 使用 `@JvmField` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-field/index.html) 注解
- 使用 `lateinit` 修饰符
- 使用 `const` 修饰符

如果对这样的属性添加 `@JvmField` 注解, 那么它的静态后端域变量可见度将会与属性本身的可见度一样.

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value
}
    }
}
```

```
// Java 代码
Key.COMPARATOR.compare(key1, key2);
// 这里访问的是 Key 类中的 public static final 域
```

命名对象或同伴对象中的延迟初始化属性 ("延迟初始化的(Late-Initialized)属性和变量" in "属性(Property)") 对应的静态的后端域变量, 其可见度将与属性的设值方法可见度一样.

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java 代码
Singleton.provider = new Provider();
// 这里访问的是 Singleton 类中的 public static 非 final 域
```

声明为 `const` 的属性(无论定义在类中, 还是在顶级范围内(top level)) 会被转换为 Java 中的静态域:

```
// 源代码文件: example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

在 Java 中:

```
int constant = Obj.CONST;
int max = ExampleKt.MAX;
int version = C.VERSION;
```

静态方法(Static Method)

上文中提到, Kotlin 会将包级函数编译为静态方法. 此外, 如果你对函数添加 `@JvmStatic` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-static/index.html) 注解, Kotlin 也可以为命名对象或同伴对象中定义的函数生成静态方法. 如果使用这个注解, 编译器既会在对象所属的类中生成静态方法, 同时也会在对象中生成实例方法. 比如:

```
class C {
    companion object {
        @JvmStatic fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

现在, `callStatic()` 在 Java 中是一个静态方法, 而 `callNonStatic()` 不是:

```
C.callStatic(); // 正确
C.callNonStatic(); // 错误: 不是静态方法
C.Companion.callStatic(); // 实例上的方法仍然存在
C.Companion.callNonStatic(); // 这个方法只能通过实例来调用
```

对命名对象也一样:

```
object Obj {
    @JvmStatic fun callStatic() {}
    fun callNonStatic() {}
}
```

在 Java 中:

```
Obj.callStatic(); // 正确
Obj.callNonStatic(); // 错误
Obj.INSTANCE.callNonStatic(); // 正确, 这是对单体实例的一个方法调用
Obj.INSTANCE.callStatic(); // 也正确
```

从 Kotlin 1.3 开始, 接口的同伴对象中定义的函数也可以使用 `@JvmStatic` 注解. 这类函数会被编译为接口中的静态方法. 注意, 从 Java 1.8 才开始支持接口中的静态方法, 因此注意编译时需要选择正确的 JVM 目标平台.

```
interface ChatBot {
    companion object {
        @JvmStatic fun greet(username: String) {
            println("Hello, $username")
        }
```

```
    }
}
```

@JvmStatic 注解也可以用于命名对象或同伴对象的属性, 可以使得属性的取值方法和设值方法变成静态方法, 对于命名对象, 这些静态方法在命名对象之内, 对于同伴对象, 这些静态方法在包含同伴对象的类之内.

接口中的默认方法(Default Method)

> 只有 JVM 1.8 或更高版本的编译目标平台才支持默认方法.

从 JDK 1.8 开始, Java 中的接口可以包含 默认方法(Default Method) (https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html). 如果要把 Kotlin 接口的所有非抽象成员变为实现这些接口的 Java 类的默认方法, 请使用编译器选项 -Xjvm-default=all 来编译 Kotlin 代码.

下面是一个有默认方法的 Kotlin 接口的示例:

```
// 使用编译器选项 -Xjvm-default=all 编译

interface Robot {
    fun move() { println("~walking~") }  // 在 Java 接口中, 这个方法将
成为默认方法
    fun speak(): Unit
}
```

对于实现这个接口的 Java 类来说, 这个方法已经存在一个默认实现.

```
// Java 实现类
public class C3PO implements Robot {
    // Robot 接口中隐含存在 move() 方法的实现
    @Override
    public void speak() {
        System.out.println("I beg your pardon, sir");
    }
}
```

```
C3PO c3po = new C3PO();
c3po.move(); // 这里会调用 Robot 接口中的默认实现
c3po.speak();
```

接口的实现类也可以覆盖默认方法.

```
// Java 代码
public class BB8 implements Robot {
    // 通过自己的实现覆盖默认方法
    @Override
    public void move() {
        System.out.println("~rolling~");
    }

    @Override
    public void speak() {
        System.out.println("Beep-beep");
    }
}
```

在 Kotlin 1.4 之前, 要生成默认方法, 可以在方法上使用 `@JvmDefault` 注解. 在 1.4 以上版本中使用编译选项 `-Xjvm-default=all` 进行编译, 结果通常等于将接口的所有的非抽象方法添加 `@JvmDefault`注解, 然后再使用编译选项 `-Xjvm-default=enable` 进行编译. 但是, 有些情况下这两种办法的结果会有区别. 关于 Kotlin 1.4 中默认方法生成过程的具体变化, 请参见 Kotlin blog 中的 这篇文章 (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-m3-generating-default-methods-in-interfaces/).

默认方法的兼容模式

如果你的 Kotlin 接口在过去编译时没有使用编译选项 `-Xjvm-default=all`, 并且有客户代码正在使用这些接口, 那么, 在你的 Kotlin 接口代码使用这个编译选项再次编译之后, 可能导致客户代码与新代码二进制不兼容. 为了避免对这种客户代码的兼容性造成破坏, 请使用 `-Xjvm-default=all` 模式, 并使用 `@JvmDefaultWithCompatibility` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-default-with-compatibility/) 注

解标记接口. 这样可以让你一次性的向公开 API 中的所有接口添加这个注解, 而且你不需要对新的非公开代码使用任何注解.

> 从 Kotlin 1.6.20 开始, 对使用 `-Xjvm-default=all` 或 `-Xjvm-default=all-compatibility` 模式编译的模块, 你可以使用默认模式 (`-Xjvm-default=disable` 编译器选项) 编译模块.

兼容模式的详细解释:

disable 模式

这是默认的模式. 不会生成 JVM 默认方法, 并禁止使用 `@JvmDefault` 注解.

all 模式

对模块中所有带有函数体的接口声明生成 JVM 默认方法. 不会对带有函数体的接口声明生成 `DefaultImpls` (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-m3-generating-default-methods-in-interfaces/) 桩代码(stub), 在 `disable` 模式下则默认会生成.

如果接口从一个 `disable` 模式下编译的接口继承了带有函数体的方法, 并且没有覆盖这个(override)方法, 那么会对这个方法生成 `DefaultImpls` 桩代码(stub).

如果某些客户代码依赖于 `DefaultImpls` 类的存在, 那么 **会破坏二进制兼容性**.

> 如果使用了接口委托, 所有的接口方法都会被委托. 唯一的例外是使用被废弃的 `@JvmDefault` 注解标注的方法.

all-compatibility 模式

与 `all` 模式相比, 还会在 `DefaultImpls` (https://blog.jetbrains.com/kotlin/2020/07/kotlin-1-4-m3-generating-default-methods-in-interfaces/) 类中生成兼容性桩代码(stub). 对于库和运行时的作者来说, 兼容性桩代码可以很有用, 对于那些使用以前的库版本编译的既有的客户代码, 它可以帮助保持向后的二进制兼容性. `all` 和 `all-compatibility` 模式改变了库重新编译之后客户代码将要使用的 ABI 界面. 因此, 客户代码可能会与以前的库版本不能兼容. 这通常代表你需要适当的库版本号, 比如, 在语义化版本(SemVer)中增加主版本号.

编译器使用 `@Deprecated` 注解来生成 `DefaultImpls` 的所有成员: 你不应该在 Java 代码中使用这些成员, 因为编译器生成这些它们只是为了保持兼容性的目的.

对于从 `all` 或 `all-compatibility` 模式下编译的 Kotlin 接口继承的情况, `DefaultImpls` 兼容性桩代码会使用标准的 JVM 运行期解析语义来调用接口的默认方法.

对继承泛型接口的类, 有些情况下, 在 `disable` 模式中会生成带有特殊签名的额外的隐含方法, `all-compatibility` 模式对这些类会执行额外的兼容性检查: 与 `disable` 模式不同, 如果你没有明确的覆

盖这样的方法, 也没有使用 `@JvmDefaultWithoutCompatibility` 注解标注这个类, 那么编译器会报告错误(详情请参见 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-39603)).

可见度

Kotlin 中的可见度修饰符与 Java 的对应规则如下:

- `private` 成员会被编译为 Java 中的 `private` 成员
- `private` 顶级声明会被编译为 Java 中的 `private` 顶级声明. 如果从类的内部访问, 那么还会包含包的 `private` 访问器(Package-private accessor).
- `protected` 成员在 Java 中仍然是 `protected` 不变 (注意, Java 允许从同一个包内的其他类访问 protected 成员, 但 Kotlin 不允许, 因此 Java 类中将拥有更大的访问权限)
- `internal` 声明会被编译为 Java 中的 `public`. `internal` 类的成员名称会被混淆, 以降低在 Java 代码中意外访问到这些成员的可能性, 并以此来实现那些根据 Kotlin 的规则相互不可见, 但是其签名完全相同的函数重载
- `public` 在 Java 中仍然是 `public` 不变

KClass

调用 Kotlin 中的方法时, 有时你可能会需要使用 `KClass` 类型的参数. Java 的 `Class` 不会自动转换为 Kotlin 的 `KClass`, 因此你必须手动进行转换, 方法是使用 `Class<T>.kotlin` 扩展属性, 这个扩展属性对应的方法是:

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

使用 @JvmName 注解处理签名冲突

有时候我们在 Kotlin 中声明了一个函数, 但在 JVM 字节码中却需要一个不同的名称. 最显著的例子就是 *类型消除(type erasure)* 时的情况:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

这两个函数无法同时定义, 因为它们产生的 JVM 代码的签名是完全相同的: `filterValid(Ljava/util/List;)Ljava/util/List;`. 如果我们确实需要在 Kotlin 中给这两个函数定义相同

的名称, 那么可以对其中一个(或两个)使用 `@JvmName` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-name/index.html) 注解, 通过这个注解的参数来指定一个不同的名称:

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

在 Kotlin 中, 可以使用相同的名称 `filterValid` 来访问这两个函数, 但在 Java 中函数名将是 `filterValid` 和 `filterValidInt`.

如果我们需要定义一个属性 `x`, 同时又定义一个函数 `getX()`, 这时也可以使用同样的技巧:

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

如果想要改变编译生成的属性访问方法的名称, 又不希望明确地实现属性的取值和设值方法, 你可以使用 `@get:JvmName` 和 `@set:JvmName`:

```
@get:JvmName("x")
@set:JvmName("changeX")
var x: Int = 23
```

重载函数(Overload)的生成

通常, 如果在 Kotlin 中定义一个函数, 并指定了参数默认值, 这个方法在 Java 中只会存在带所有参数的版本. 如果你希望 Java 端的使用者看到不同参数的多个重载方法, 那么可以使用 `@JvmOverloads` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-jvm-overloads/index.html) 注解.

这个注解也可以用于构造器, 静态方法, 等等. 但不能用于抽象方法, 包括定义在接口内的方法.

```
class Circle @JvmOverloads constructor(centerX: Int, centerY: Int,
radius: Double = 1.0) {
    @JvmOverloads fun draw(label: String, lineWidth: Int = 1, color:
```

```
String = "red") { /*...*/ }
}
```

对于每个带有默认值的参数, 都会生成一个新的重载方法, 这个重载方法的签名将会删除这个参数, 以及右侧的所有参数. 上面的示例程序生成的结果如下:

```
// 构造器:
Circle(int centerX, int centerY, double radius)
Circle(int centerX, int centerY)

// 方法
void draw(String label, int lineWidth, String color) { }
void draw(String label, int lineWidth) { }
void draw(String label) { }
```

注意, 在 次级构造器 ("次级构造器(secondary constructor)" in "类") 中介绍过, 如果一个类的构造器方法参数全部都指定了默认值, 那么会对这个类生成一个 public 的无参数构造器. 这个特性即使在没有使用 `@JvmOverloads` 注解时也是有效的.

受控异常(Checked Exception)

Kotlin 中不存在受控异常. 因此, Kotlin 函数在 Java 中的签名通常不会声明它抛出的异常. 因此, 假如你有一个这样的 Kotlin 函数:

```
// 源代码文件: example.kt
package demo

fun writeToFile() {
    /*...*/
    throw IOException()
}
```

然后你希望在 Java 中调用它, 并捕获异常:

```
// Java 代码
try {
    demo.Example.writeToFile();
} catch (IOException e) {
    // 错误: writeToFile() 没有声明抛出 IOException 异常
```

```
    // ...
}
```

这时 Java 编译器会报告错误, 因为 writeToFile() 没有声明抛出 IOException 异常. 为了解决这个问题, 可以在 Kotlin 中使用 @Throws (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-throws/index.html) 注解:

```
@Throws(IOException::class)
fun writeToFile() {
    /*...*/
    throw IOException()
}
```

Null值安全性

在 Java 中调用 Kotlin 函数时, 没有任何机制阻止我们向一个非 null 参数传递一个 null 值. 所以, Kotlin 编译时, 会对所有接受非 null 值参数的 public 方法产生一些运行时刻检查代码. 由于这些检查代码的存在, Java 端代码会立刻得到一个 NullPointerException 异常.

泛型的类型变异(Variant)

如果 Kotlin 类使用了 声明处的类型变异(declaration-site variance) ("声明处的类型变异(Declaration-site variance)" in "泛型(Generic): in, out, where"), 那么这些类在 Java 代码中看到的形式存在两种可能. 比如, 你有下面这样的类, 以及两个使用这个类的函数:

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

如果用最简单的方式转换为 Java 代码, 结果将是:

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

问题在于, 在 Kotlin 中你可以这样: `unboxBase(boxDerived(Derived()))`, 但在 Java 中却不可以, 因为在 Java 中, `Box` 的类型参数 `T` 是 *不可变的(invariant)*, 因此 `Box<Derived>` 不是 `Box<Base>` 的子类型. 为了解决 Java 端的问题, 你需要将 `unboxBase` 函数定义成这样:

```
Base unboxBase(Box<? extends Base> box) { ... }
```

这个声明使用了 Java 的 *通配符类型(wildcards type)* (`? extends Base`), 通过使用处类型变异(use-site variance)来模仿声明处的类型变异(declaration-site variance), 因为 Java 中只有使用处类型变异.

为了让 Kotlin 的 API 可以在 Java 中正常使用, 如果一个类*被用作函数参数*, 那么对于定义了类型参数协变的 `Box` 类, 编译器会将 `Box<Super>` 生成为 Java 的 `Box<? extends Super>` (对于定义了类型参数反向协变的 `Foo` 类, 会生成为 Java 的 `Foo<? super Bar>`). 当类被用作返回值时, 编译器不会生成类型通配符, 否则 Java 端的使用者就不得不处理这些类型通配符(而且这是违反通常的 Java 编程风格的). 因此, 我们上面例子中的函数真正的输出结果是这样的:

```
// 返回值 - 没有类型通配符
Box<Derived> boxDerived(Derived value) { ... }

// 参数 - 有类型通配符
Base unboxBase(Box<? extends Base> box) { ... }
```

> 如果类型参数是 final 的, 那么生成类型通配符一般来说就没有意义了, 因此 `Box<String>` 永远是 `Box<String>`, 无论它出现在什么位置.

如果你需要类型通配符, 但默认没有生成, 可以使用 `@JvmWildcard` 注解:

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> =
Box(value)
// 将被翻译为
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

反过来, 如果默认生成了类型通配符, 但你不需要它, 可以使用 `@JvmSuppressWildcards` 注解:

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base =
box.value
```

```
// 将被翻译为
// Base unboxBase(Box<Base> box) { ... }
```

> @JvmSuppressWildcards 不仅可以用于单个的类型参数, 也可以用于整个函数声明或类声明, 这时它会使得这个函数或类之内的所有类型通配符都不产生.

Nothing 类型的翻译

Nothing ("Nothing 类型" in "异常(Exception)") 类型是很特殊的, 因为它在 Java 中没有对应的概念. 所有的 Java 引用类型, 包括java.lang.Void, 都可以接受 null 作为它的值, 而 Nothing 甚至连 null 值都不能接受. 因此, 在 Java 的世界里无法准确地表达这个类型. 因此, Kotlin 会在使用 Nothing 类型参数的地方生成一个原生类型(raw type):

```
fun emptyList(): List<Nothing> = listOf()
// 将被翻译为
// List emptyList() { ... }
```

Spring Boot 和 Kotlin 入门

最终更新: 2024/09/10

通过完成这个教程, 学习使用 Spring Boot 和 Kotlin: 本教程将会带领你使用 Spring Boot 创建一个简单的应用程序, 并添加数据库来存储信息.

完成以下 4 个步骤, 你将会学到 Kotlin 语言的很多基本功能:

1 创建 Spring Boot 项目 (使用 Kotlin 创建 Spring Boot 项目)

2 向 Spring Boot 项目添加数据类 (向 Spring Boot 项目添加数据类)

3 为 Spring Boot 项目添加数据库支持 (为 Spring Boot 项目添加数据库支持)

4 使用 Spring Data CrudRepository 进行数据库访问 (使用 Spring Data CrudRepository 进行数据库访问)

下一步

首先, 使用 IntelliJ IDEA 和 Kotlin 创建一个 Spring Boot 项目 (使用 Kotlin 创建 Spring Boot 项目).

参见

请阅读我们的 Java 到 Kotlin (J2K) 的互操作和迁移向导:

- 在 Kotlin 中调用 Java 代码 (在 Kotlin 中调用 Java 代码) 以及 在 Java 中调用 Kotlin 代码 (在 Java 中调用 Kotlin 代码)
- Java 和 Kotlin 中的集合(Collection) (Java 和 Kotlin 中的集合(Collection))
- Java 和 Kotlin 中的字符串 (Java 和 Kotlin 中的字符串)

加入开发社区

- **Kotlin slack**: 首先 得到邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up), 然后加入 #spring (https://kotlinlang.slack.com/archives/C0B8ZTWE4) 和 #server (https://kotlinlang.slack.com/archives/C0B8RC352) 频道.

- **Stack Overflow**: 订阅 "kotlin" (https://stackoverflow.com/questions/tagged/kotlin), "spring-kotlin" (https://stackoverflow.com/questions/tagged/spring-kotlin), 或 "ktor" (https://stackoverflow.com/questions/tagged/ktor) 标签
- **Kotlin YouTube channel**: 订阅并观看关于 使用 Spring 开发 Kotlin (https://www.youtube.com/playlist?list=PLlFc5cFwUnmxOJL0GSSZ1Vot4KL2Vwe7x) 的视频

使用 Kotlin 创建 Spring Boot 项目

> 这是 **Spring Boot 和 Kotlin 入门** 教程的第 1 部分:
> 1 **使用 Kotlin 创建 Spring Boot 项目** 2 向 Spring Boot 项目添加数据类 3 为 Spring Boot 项目添加数据库支持 4 使用 Spring Data CrudRepository 进行数据库访问

最终更新: 2024/09/10

本教程的第 1 部分向你演示如何在 IntelliJ IDEA 中使用 Project Wizard 创建一个 Spring Boot 项目.

开始之前的准备

下载并安装 IntelliJ IDEA Ultimate Edition (https://www.jetbrains.com/idea/download/index.html) 的最新版.

> 如果你使用的是 IntelliJ IDEA Community Edition 或其他 IDE, 你可以使用 基于 web 页面的项目生成器 (https://start.spring.io) 来生成 Spring Boot 项目.

创建 Spring Boot 项目

使用 IntelliJ IDEA Ultimate Edition 中的 Project Wizard, 创建新的使用 Kotlin 的 Spring Boot 项目:

> 你也可以使用 IntelliJ IDEA 和 Spring Boot plugin (https://www.jetbrains.com/help/idea/spring-boot.html) 来创建新项目.

1. 在 IntelliJ IDEA 中, 选择 **File** | **New** | **Project**.

2. 在左侧面板中, 选择 **New Project** | **Spring Initializr**.

3. 在 Project Wizard 窗口中, 指定以下项目和选项:

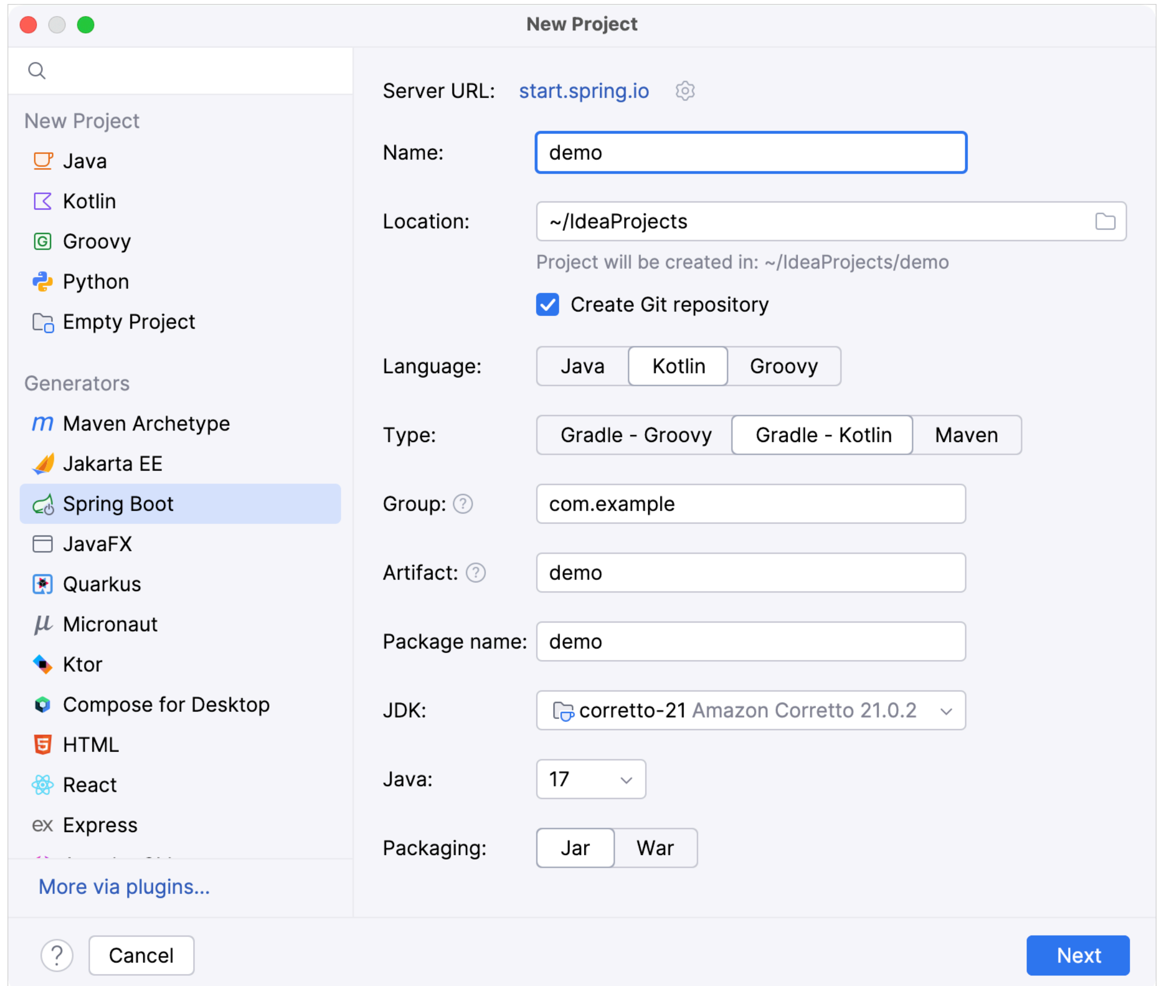
   - **Name**: demo

   - **Language**: Kotlin

- **Build system**: Gradle
- **JDK**: Java 17 JDK

> 本教程使用 **Amazon Corretto version 18.**

- **Java**: 17



创建 Spring Boot 项目

4. 确认填写了所有的项目, 然后点击 **Next.**

5. 选择以下依赖项, 本教程将会需要它们:

- **Web / Spring Web**
- **SQL / Spring Data JDBC**

- **SQL / H2 Database**

New Project

Spring Boot: 3.3.2

Dependencies:

Search

- Developer Tools
- Web
  - [x] Spring Web
  - [ ] Spring Reactive Web
  - [ ] Spring for GraphQL
  - [ ] Rest Repositories
  - [ ] Spring Session
  - [ ] Rest Repositories HAL Explorer
  - [ ] Spring HATEOAS
  - [ ] Spring Web Services
  - [ ] Jersey
  - [ ] Vaadin
  - [ ] Netflix DGS
- Template Engines
- Security
- SQL
  - [ ] JDBC API
  - [ ] Spring Data JPA
  - [x] Spring Data JDBC
  - [ ] Spring Data R2DBC
  - [ ] MyBatis Framework
  - [ ] Liquibase Migration
  - [ ] Flyway Migration
  - [ ] JOOQ Access Layer
  - [ ] IBM DB2 Driver
  - [ ] Apache Derby Database
  - [x] H2 Database
  - [ ] HyperSQL Database

H2 Database

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Added dependencies:

- Spring Web
- Spring Data JDBC
- H2 Database

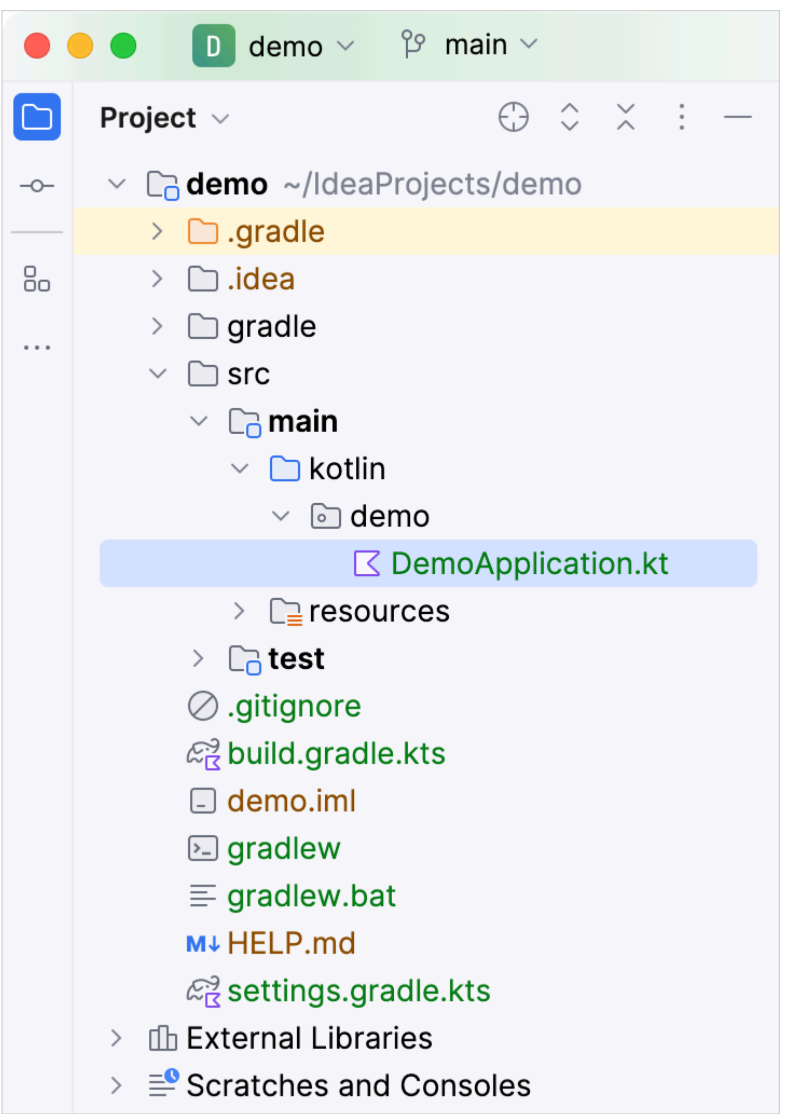
Cancel | Previous | Create

设置 Spring Boot 项目

6. 点击 **Create**, 生成并设置项目.

IDE 将会生成并打开新的项目. 可能需要一些时间来下载并导入项目的依赖项.

7. 之后, 你可以在 **Project view** 中看到下面的项目结构:



设置 Spring Boot 项目

生成的 Gradle 项目符合 Maven 的标注目录布局:

- 在 `main/kotlin` 文件夹下是属于应用程序的包和类.
- 应用程序的入口点是 `DemoApplication.kt` 文件的 `main()` 方法.

查看项目的 Gradle 构建文件

打开 `build.gradle.kts` 文件: 它是 Gradle Kotlin 构建脚本, 包含应用程序需要的依赖项目列表.

Gradle 文件是用于 Spring Boot 的标准内容, 但它也包含必须的 Kotlin 依赖项, 包括 kotlin-spring Gradle plugin – `kotlin("plugin.spring")`.

下面是完整的脚本, 包括各部分和依赖项的解释:

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile // 用于下面的
`KotlinCompile` task

plugins {
    id("org.springframework.boot") version "3.1.2"
    id("io.spring.dependency-management") version "1.1.2"
    kotlin("jvm") version "{{ site.data.releases.latest.version }}"
// 使用的 Kotlin 版本
    kotlin("plugin.spring") version "{{
site.data.releases.latest.version }}" // Kotlin Spring plugin
}

group = "com.example"
version = "0.0.1-SNAPSHOT"

java {
    sourceCompatibility = JavaVersion.VERSION_17
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-
data-jdbc")
    implementation("org.springframework.boot:spring-boot-starter-
web")
    implementation("com.fasterxml.jackson.module:jackson-module-
kotlin") // Jackson 扩展，用于在 Kotlin 中使用 JSON
    implementation("org.jetbrains.kotlin:kotlin-reflect") // Kotlin
反射库，使用 Spring 时需要
    runtimeOnly("com.h2database:h2")
    testImplementation("org.springframework.boot:spring-boot-
starter-test")
}

tasks.withType<KotlinCompile> { // `KotlinCompile` task 的设置
```

```
    kotlinOptions { // Kotlin 编译器选项
        freeCompilerArgs = listOf("-Xjsr305=strict") // `-
Xjsr305=strict` 对 JSR-305 注解启用 strict 模式
        jvmTarget = "17" // 这个选项指定生成的 JVM 字节码的目标版本
    }
}

tasks.withType<Test> {
    useJUnitPlatform()
}
```

你可以看到, Gradle 构建文件中添加了几个与 Kotlin 相关的库:

1. 在 `plugins` 代码段中, 有 2 个 Kotlin 库:
   - `kotlin("jvm")` – 这个 plugin 定义在项目中使用的 Kotlin 版本
   - `kotlin("plugin.spring")` – Kotlin Spring 编译器 plugin, 用于向 Kotlin 类添加 `open` 修饰符, 使它们能够与 Spring Framework 中的功能兼容
2. 在 `dependencies` 代码段中, 有几个 Kotlin 相关的模块:
   - `com.fasterxml.jackson.module:jackson-module-kotlin` – 这个模块支持Kotlin 类和数据类的序列化和反序列化
   - `org.jetbrains.kotlin:kotlin-reflect` – Kotlin 反射库
3. 在依赖项之后, 你可以看到 `KotlinCompile` task 配置模块. 在这里你可以向编译器添加额外的参数, 来启用或禁用某些语言特性.

查看生成的 Spring Boot 应用程序

打开 `DemoApplication.kt` 文件:

```
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication
```

```
fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

声明类 – DemoApplication 类

在包声明和 import 语句之后, 你可以看到第一个类声明, class DemoApplication.

在 Kotlin 中, 如果一个类不包含任何成员 (属性或函数), 你可以直接省略掉类的主体部分 ({}).

@SpringBootApplication 注解

@SpringBootApplication 注解 (https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.using-the-springbootapplication-annotation) 在 Spring Boot 应用程序中是一个很方便的注解. 它会启用 Spring Boot 的 自动配置 (https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.auto-configuration), 组件扫描 (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html), 而且可以对 "应用程序类" 定义额外的配置.".

程序入口点 – main()

main() ("程序入口点(entry point)" in "基本语法") 函数是应用程序的入口点.

它声明为在 DemoApplication 类之外的一个 顶层函数 ("函数的范围" in "函数"). main() 函数调用 Spring 的 runApplication(*args) 函数, 使用 Spring Framework 来启动应用程序.

可变参数 – args: Array<String>

查看 runApplication() 函数的声明, 你会看到函数的参数标记了 vararg 修饰符 ("不定数量参数(varargs)" in "函数"): vararg args: String. 这表示, 你可以向这个函数传递可变数量的字符串参数.

展开(spread)操作符 – (*args)

args 是 main() 函数的参数, 它声明为一个字符串数组. 由于存在的是字符串的数组, 而你想要将它的内容传递给函数, 请使用展开(spread)操作符 (在数组之前加上星号 *).

创建 Controller

应用程序已经可以运行了, 但我们先来更新它的逻辑.

在 Spring 应用程序中, Controller 用来处理 Web 请求. 在 `DemoApplication.kt` 文件中, 创建 `MessageController` 类, 如下:

```
@RestController
class MessageController {
    @GetMapping("/")
    fun index(@RequestParam("name") name: String) = "Hello, $name!"
}
```

@RestController 注解

你需要告诉 Spring, MessageController 是一个 REST Controller, 因此你应该对它标注 @RestController 注解.

这个注解表示这个类将会被组件扫描识别, 因为它和我们的 DemoApplication 类处在相同的包内.

@GetMapping 注解

@GetMapping 标注 REST Controller 的函数, 它实现了与 HTTP GET 调用对应的 endpoint:

@GetMapping("/") fun index(@RequestParam("name") name: String) = "Hello, $name!"

@RequestParam 注解

函数参数 name 标注了 @RequestParam 注解. 这个注解表示方法参数应该绑定到一个 Web 请求参数.

因此, 如果你访问应用程序的根路径, 并提供一个请求名为 "name" 的参数, 例如 /?name=<your-value>, 这个参数值将会被用做调用 index() 函数时的参数.

单表达式函数 – index()

由于 index() 函数只包含一条语句, 你可以将它声明为一个 单表达式函数 ("单表达式函数 (Single-expression function)" in "函数").

意思就是说, 大括号可以省略, 函数体直接放在等号 = 之后.

函数返回值的类型推断

index() 函数没有明确声明返回类型. 编译器会查看等号 = 右侧语句的结果, 以此推断返回类型.

Hello, $name! 表达式的类型是 String, 因此函数的返回类型也是 String.

字符串模板 – $name

Hello, $name! 表达式在 Kotlin 中称为 "字符串模板" in "字符串".

字符串模板是字符串的字面值, 其中包含内嵌的表达式.

对于字符串的拼接操作, 这是一个很方便的替代方法.

这些 Spring 注解需要额外的 import 语句:

```
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import
org.springframework.web.bind.annotation.RestController
```

下面是 `DemoApplication.kt` 的完整代码:

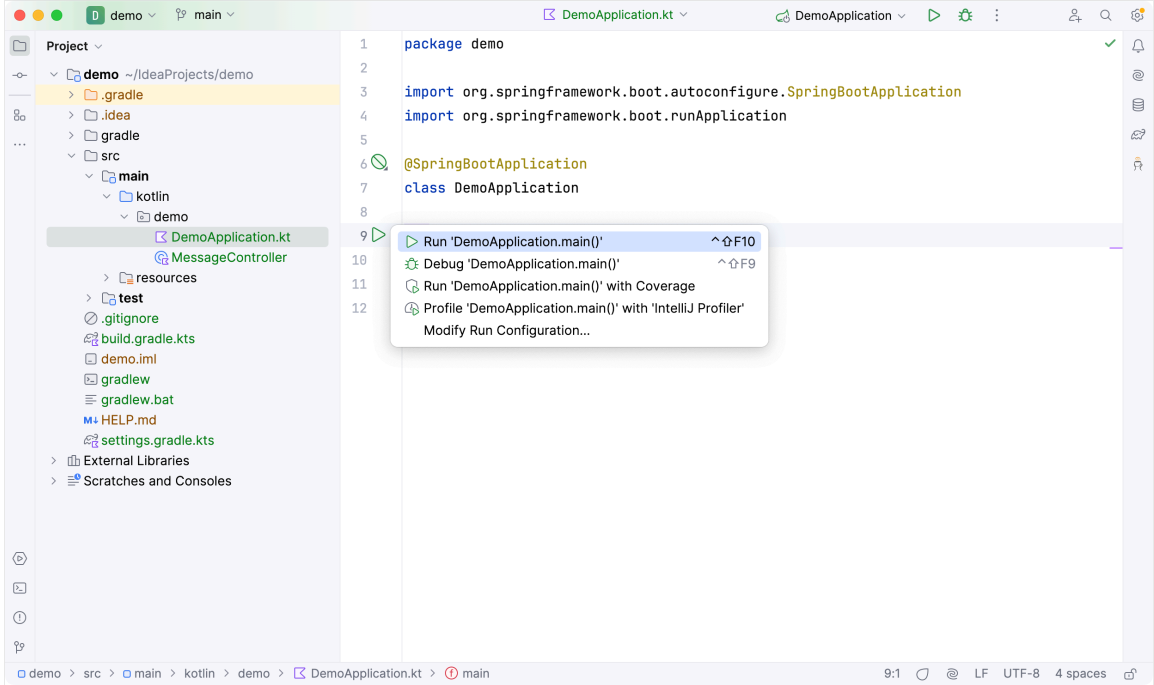
```
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
```

```
import org.springframework.boot.runApplication
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

@RestController
class MessageController {
    @GetMapping("/")
    fun index(@RequestParam("name") name: String) = "Hello, $name!"
}
```

运行应用程序

Spring 应用程序现在可以运行了:

1. 点击 `main()` 方法侧栏中的绿色 Run 图标:



运行 Spring Boot 应用程序

你也可以在终端窗口运行 `./gradlew bootRun` 命令.

这样会在你的计算机上启动本地服务器.

2. 应用程序启动后, 请打开以下 URL:

```
http://localhost:8080?name=John
```

你会看到输出的结果 "Hello, John!":

Spring 应用程序的应答

下一步

本教程的下一部分中, 你将学习 Kotlin 数据类, 以及如何在你的应用程序中使用.

阅读下一章 (向 Spring Boot 项目添加数据类)

向 Spring Boot 项目添加数据类

这是 **Spring Boot 和 Kotlin 入门** 教程的第 2 部分. 开始这一部分之前, 请确认你已经完成了前面的步骤:

① 使用 Kotlin 创建 Spring Boot 项目 (使用 Kotlin 创建 Spring Boot 项目) ② **向 Spring Boot 项目添加数据类** ③ 为 Spring Boot 项目添加数据库支持 ④ 使用 Spring Data CrudRepository 进行数据库访问

最终更新: 2024/09/10

在教程的这个部分, 你将会向应用程序添加更多功能, 并学会 Kotlin 语言的更多功能, 例如数据类.
我们需要修改 MessageController 类, 来返回 JSON 格式的应答, 其中包含一组序列化的对象.

更新你的应用程序

1. 在 DemoApplication.kt 文件中, 创建一个 Message 数据类, 包含 2 个属性: id 和 text:

```
data class Message(val id: String?, val text: String)
```

   Message 类将被用来传递数据: 一组序列化后的 Message 对象将会组成 JSON 文档, Controller 会对浏览器请求返回这个 JSON 文档.

   数据类 – Message

   Kotlin 中的 数据类 (数据类(Data Class)) 的主要目的是用来保存数据. 这样的类使用 data 关键字进行标记, 而且从类结构能够得到一些标准的功能和有用的函数.

   在上面的示例中, 你将 Message 声明为数据类, 因为它的主要目的是存储数据.

   val 和 var 属性

   Kotlin 类中的属性 (属性(Property)) 可以声明为:

   - , 使用 var 关键字
   - , 使用 val 关键字

Message 类使用 val 关键字声明了 2 个属性, id 和 text. 编译器会为这些属性自动生成 get 函数. 在 Message 类的实例创建之后, 将无法对这些属性重新赋值.

可为 Null 的类型 – String?

Kotlin 提供了 对可为 Null 的类型的内建支持 ("可为 null 的类型与不可为 null 的类型" in "Null 值安全性"). 在 Kotlin 中, 类型系统会区分可以为 null 值的引用 () 和不可以为 null 值的引用 (). 例如, 一个通常的 String 类型变量不能保存 null 值. 要允许使用 null 值, 你可以将一个变量声明为可为 null 的字符串, 写做 String?.

这里, Message 类的 id 属性声明为可为 null 的类型. 因此, 可以对 id 传递 `null` 来创建一个 Message 类的实例:

Message(null, "Hello!")

2. 在同一个文件内, 修改 `MessageController` 类的 `index()` 函数, 让它返回 `Message` 对象的 List:

```
@RestController
class MessageController {
    @GetMapping("/")
    fun index() = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}
```

集合 – listOf()

Kotlin 标准库提供了基本的集合类型的实现: Set, List, 和 Map. 对每个集合类型都存在一对接口:

- 一个 接口, 提供了访问集合元素的操作.
- 一个 接口, 扩展了对应的只读接口, 增加了写操作: 添加, 删除, 以及更新集合的元素.

Kotlin 标准库还提供了对应的工厂函数, 用来创建这些集合的实例.

本教程中, 你使用了 listOf() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/list-of.html) 函数来创

建 Message 对象的 List. 这是用来创建对象的 List 的工厂函数: 你不能向 List 添加或删除元素. 如果需要对 List 执行写操作, 请调用 mutableListOf() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-list-of.html) 函数来创建一个可变的 List 实例.

尾随逗号(Trailing Comma)

尾随逗号(Trailing Comma) ("尾随逗号(Trailing Comma)" in "编码规约") 是指在一系列元素的 **最终元素** 之后的逗号: Message("3", "Privet!"), 这是 Kotlin 语法中的一个便利的功能, 而且它是可选的 – 不使用尾随逗号, 你的代码也能正常运行.

在上面的示例中, 创建 Message 对象的 List 时, 在 listOf() 函数最后的参数之后包括了尾随逗号.

MessageController 的应答现在是一个 JSON 文档, 其中包含 Message 对象的集合.

> 如果 Jackson 库存在于类路径中, 那么 Spring 应用程序中的所有 Controller 都会默认输出 JSON 格式的应答. 由于你 在 build.gradle.kts 文件中指定了 spring-boot-starter-web 依赖项 ("查看项目的 Gradle 构建文件" in "使用 Kotlin 创建 Spring Boot 项目"), 你会通过 *传递(transitive)* 依赖项的方式得到 Jackson. 因此, 如果 endpoint 返回一个能够被序列化为 JSON 的数据结构, 应用程序就会应答一个 JSON 文档.

下面是 DemoApplication.kt 的完整代码:

```
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.data.annotation.Id
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
```

```
}

@RestController
class MessageController {
    @GetMapping("/")
    fun index() = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}

data class Message(val id: String?, val text: String)
```

运行应用程序

Spring 应用程序已经可以运行了:

1. 再次运行应用程序.
2. 应用程序启动后, 打开以下 URL:

   ```
   http://localhost:8080
   ```

   你将会看到一个页面, 包含 JSON 格式的 message 集合:

localhost:8080

Pretty print

```
[
  {
    "id": "1",
    "text": "Hello!"
  },
  {
    "id": "2",
    "text": "Bonjour!"
  },
  {
    "id": "3",
    "text": "Privet!"
  }
]
```

运行应用程序

下一步

本教程的下一部分中, 你将会向你的项目添加并配置一个数据库, 并发送 HTTP 请求.

阅读下一章 (为 Spring Boot 项目添加数据库支持)

为 Spring Boot 项目添加数据库支持

这是 **Spring Boot 和 Kotlin 入门** 教程的第 3 部分. 开始这一部分之前, 请确认你已经完成了前面的步骤:
① 使用 Kotlin 创建 Spring Boot 项目 (使用 Kotlin 创建 Spring Boot 项目) ② 向 Spring Boot 项目添加数据类 (向 Spring Boot 项目添加数据类) ③ **为 Spring Boot 项目添加数据库支持** ④ 使用 Spring Data CrudRepository 进行数据库访问

最终更新: 2024/09/10

在教程的这个部分, 你将会使用 JDBC 向你的项目添加并配置一个数据库. 在 JVM 应用程序中, 你要使用 JDBC 来操作数据库. 为了方便, Spring Framework 提供了 `JdbcTemplate` 类, 简化 JDBC 的使用, 并帮助避免常见的错误.

添加数据库支持

在使用 Spring Framework 的应用程序中, 通常的做法是在所谓的 *服务(Service)* 层实现数据库访问逻辑 – 这是实现业务逻辑的地方. 在 Spring 中, 你需要使用 `@Service` 注解来标注类, 表示类属于应用程序的服务层. 在这个应用程序中, 你将会创建 `MessageService` 类来实现这个目的.

在 `DemoApplication.kt` 文件中, 创建 `MessageService` 类, 如下:

```
import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate

@Service
class MessageService(val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from
messages") { response, _ ->
        Message(response.getString("id"),
response.getString("text"))
    }

    fun save(message: Message) {
        db.update(
            "insert into messages values ( ?, ? )",
            message.id, message.text
        )
```

```
    }
}
```

构造器参数与依赖注入 – (val db: JdbcTemplate)

Kotlin 中的类有一个主构造器. 还可以有一个或多个 次级构造器 ("次级构造器(secondary constructor)" in "类"). 是类头部的一部分, 位于类名称以及可选的类型参数之后. 在我们的例子中, 构造器是 (val db: JdbcTemplate).

val db: JdbcTemplate 是构造器的参数:

@Service class MessageService(val db: JdbcTemplate)

尾缀 Lambda 表达式(Trailing Lambda) 与 SAM 转换

findMessages() 函数调用 JdbcTemplate 类的 query() 函数. query() 函数接受 2 个参数: 一个 SQL 查询, 类型为字符串, 以及一个回调, 将每一行查询结果转换为对象:

db.query("...", RowMapper { ... })

RowMapper 接口只声明了一个方法, 因此可以使用 Lambda 表达式来实现它, 省略接口名称. Kotlin 编译器知道表达式需要转换成的接口, 因为你将它用作函数调用的一个参数. 这个功能称为 Kotlin 中的SAM 转换 ("SAM 转换" in "在 Kotlin 中调用 Java 代码"):

db.query("...", { ... })

在 SAM 转换之后, query 函数得到 2 个参数: 首先是一个 String, 后面是一个 Lambda 表达式. 根据 Kotlin 的习惯, 如果一个函数的最后一个参数是一个函数, 那么传递给这个参数的 Lambda 表达式可以放在括号之外. 这样的语法称为 尾缀 Lambda 表达式(Trailing Lambda) ("函数调用时使用尾缀 Lambda 表达式" in "高阶函数与 Lambda 表达式"):

db.query("...") { ... }

对未使用的 Lambda 表达式参数使用下划线

对于带有多个参数的 Lambda 表达式, 你可以使用下划线 _ 符号来代替你不需要使用的参数的名称.

因此, query 函数调用的最终语法如下:

db.query("select * from messages") { response, _ -> Message(response.getString("id"), response.getString("text")) }

更新 MessageController 类

更新 `MessageController` 来使用新的 `MessageService` 类:

```
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.PostMapping

@RestController
class MessageController(val service: MessageService) {
    @GetMapping("/")
    fun index(): List<Message> = service.findMessages()

    @PostMapping("/")
    fun post(@RequestBody message: Message) {
       service.save(message)
    }
}
```

@PostMapping 注解

负责处理 HTTP POST 请求的方法需要标注 @PostMapping 注解. 为了将 HTTP 请求 Body 部的 JSON 内容转换为对象, 你需要对方法参数使用 @RequestBody 注解. 由于 Jackson 库存在于应用程序的类路径中, 这个转换能够自动完成.

更新 MessageService 类

`Message` 类的 `id` 声明为可为 null 的字符串:

```
data class Message(val id: String?, val text: String)
```

但是, 将 `null` 作为 `id` 值保存到数据库是不正确的: 你需要恰当的处理这样的情况.

更新你的代码, 在将 message 保存到时数据库, 如果 `id` 为 `null`, 生成新的值:

```kotlin
import java.util.UUID

@Service
class MessageService(val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from
messages") { response, _ ->
        Message(response.getString("id"),
response.getString("text"))
    }

    fun save(message: Message) {
        val id = message.id ?: UUID.randomUUID().toString()
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
    }
}
```

Elvis 操作符 – ?:

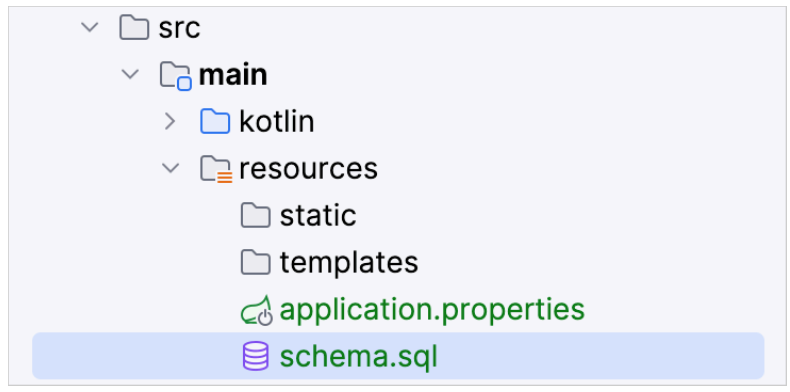
代码 message.id ?: UUID.randomUUID().toString() 使用了 Elvis 操作符 (if-not-null-else 的缩写) ?: ("Elvis 操作符" in "Null 值安全性"). 如果 ?: 左侧的表达式不是 null, Elvis 操作符会返回这个表达式的值; 否则, 它返回右侧表达式的值. 注意, 右侧表达式只有在左侧表达式为 null 的情况下才会计算.

应用程序代码已经可以访问数据库了. 现在需要配置数据源.

配置数据库

在应用程序中配置数据库:

1. 在 `src/main/resources` 目录中创建 `schema.sql` 文件. 它将会保存数据库对象的定义:



创建数据库 Schema

2. 更新 `src/main/resources/schema.sql` 文件, 内容如下:

```
CREATE TABLE IF NOT EXISTS messages (
id       VARCHAR(60)  PRIMARY KEY,
text     VARCHAR      NOT NULL
);
```

它创建 `messages` 表, 包含 2 个列: `id` 和 `text`. 表结构与 `Message` 类一致.

3. 打开 `src/main/resources` 文件夹内的 `application.properties` 文件, 添加以下应用程序属性:

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./data/testdb
spring.datasource.username=name
spring.datasource.password=password
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.mode=always
```

这些设置会为 Spring Boot 应用程序启用数据库. 关于完整的应用程序属性列表, 请参见 Spring 文档 (https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html).

通过 HTTP 请求, 向数据库添加 message

你应该使用一个 HTTP 客户端来访问前面创建的 Endpoint. 在 IntelliJ IDEA 中, 请使用内嵌的 HTTP Client:

1. 运行应用程序. 应用程序启动之后, 你可以执行 POST 请求来向数据库存储消息. 在 `src/main/resources` 文件夹中创建 `requests.http` 文件, 并添加以下 HTTP 请求:

   ```
   ### Post "Hello!"
   POST http://localhost:8080/
   Content-Type: application/json

   {
     "text": "Hello!"
   }

   ### Post "Bonjour!"

   POST http://localhost:8080/
   Content-Type: application/json

   {
     "text": "Bonjour!"
   }

   ### Post "Privet!"

   POST http://localhost:8080/
   Content-Type: application/json

   {
     "text": "Privet!"
   }

   ### 得到所有的 message
   GET http://localhost:8080/
   ```

2. 执行所有的 POST 请求. 使用请求声明侧栏中的绿色 **Run** 图标. 这些请求会将消息写入到数据库:

执行 POST 请求

3. 执行 GET 请求, 并在 **Run** 工具窗口查看结果:

执行 GET 请求

执行请求的其它方式

你也可以使用任何其它的 HTTP Client, 或 cURL 命令行工具. 比如, 在终端中运行以下命令, 得到同样的结果:

```
curl -X POST --location "http://localhost:8080" -H "Content-Type:
application/json" -d "{ \"text\": \"Hello!\" }"

curl -X POST --location "http://localhost:8080" -H "Content-Type:
application/json" -d "{ \"text\": \"Bonjour!\" }"

curl -X POST --location "http://localhost:8080" -H "Content-Type:
application/json" -d "{ \"text\": \"Privet!\" }"

curl -X GET --location "http://localhost:8080"
```

通过 id 获取 message

为应用程序增加新的功能, 通过 id 来获取单个的 message.

1. 在 `MessageService` 类中, 添加新的函数 `findMessageById(id: String)`, 通过 id 来获取单个的 message:

```
import org.springframework.jdbc.core.query

@Service
class MessageService(val db: JdbcTemplate) {

    fun findMessages(): List<Message> = db.query("select * from
messages") { response, _ ->
        Message(response.getString("id"),
response.getString("text"))
    }

    fun findMessageById(id: String): List<Message> =
db.query("select * from messages where id = ?", id) { response, _
->
        Message(response.getString("id"),
response.getString("text"))
    }
```

```kotlin
    fun save(message: Message) {
        val id = message.id ?: UUID.randomUUID().toString()
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
    }
}
```

> ⚠ 通过 id 来获取 message 的 `.query()` 函数是由 Spring Framework 提供的一个 Kotlin 扩展函数 ("扩展函数(Extension Function)" in "扩展"), 如上面的代码所示, 它需要一个额外的 `import org.springframework.jdbc.core.query` 语句.

2. 向 `MessageController` 类添加新的 `index(...)` 函数, 参数是 `id`:

```kotlin
import org.springframework.web.bind.annotation.*

@RestController
class MessageController(val service: MessageService) {
    @GetMapping("/")
    fun index(): List<Message> = service.findMessages()

    @GetMapping("/{id}")
    fun index(@PathVariable id: String): List<Message> =
        service.findMessageById(id)

    @PostMapping("/")
    fun post(@RequestBody message: Message) {
        service.save(message)
    }
}
```

从 context 路径得到值

Spring Framework 会从 context 路径得到 message 的 id 值, 因为你对新函数标注了 @GetMapping("/{id}") 注解. 通过对函数参数标注 @PathVariable 注解, 你告诉 Spring

Framework 使用得到的值作为函数参数. 新函数会调用 MessageService 来通过 id 取得单个 message.

vararg 参数在参数列表中的位置

query() 函数接受 3 个参数:

- SQL 查询字符串, 它执行时需要一个参数
- `id`, 类型为字符串的参数
- RowMapper 实例, 由 Lambda 表达式实现

query() 函数的第 2 个参数声明为 (vararg). 在 Kotlin 中, 不定数量参数的位置并不要求是在参数列表的最后.

下面是 `DemoApplication.kt` 的完整代码:

```
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate
import java.util.UUID
import org.springframework.jdbc.core.query
import org.springframework.web.bind.annotation.*

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

@RestController
class MessageController(val service: MessageService) {
    @GetMapping("/")
    fun index(): List<Message> = service.findMessages()
```

```
    @GetMapping("/{id}")
    fun index(@PathVariable id: String): List<Message> =
        service.findMessageById(id)

    @PostMapping("/")
    fun post(@RequestBody message: Message) {
        service.save(message)
    }
}

data class Message(val id: String?, val text: String)

@Service
class MessageService(val db: JdbcTemplate) {

    fun findMessages(): List<Message> = db.query("select * from
messages") { response, _ ->
        Message(response.getString("id"),
response.getString("text"))
    }

    fun findMessageById(id: String): List<Message> =
db.query("select * from messages where id = ?", id) { response, _ ->
        Message(response.getString("id"),
response.getString("text"))
    }

    fun save(message: Message) {
        val id = message.id ?: UUID.randomUUID().toString()
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
    }
}
```

运行应用程序

Spring 应用程序已经可以运行了:

1. 再次运行应用程序.
2. 打开 `requests.http` 文件, 添加新的 GET 请求:

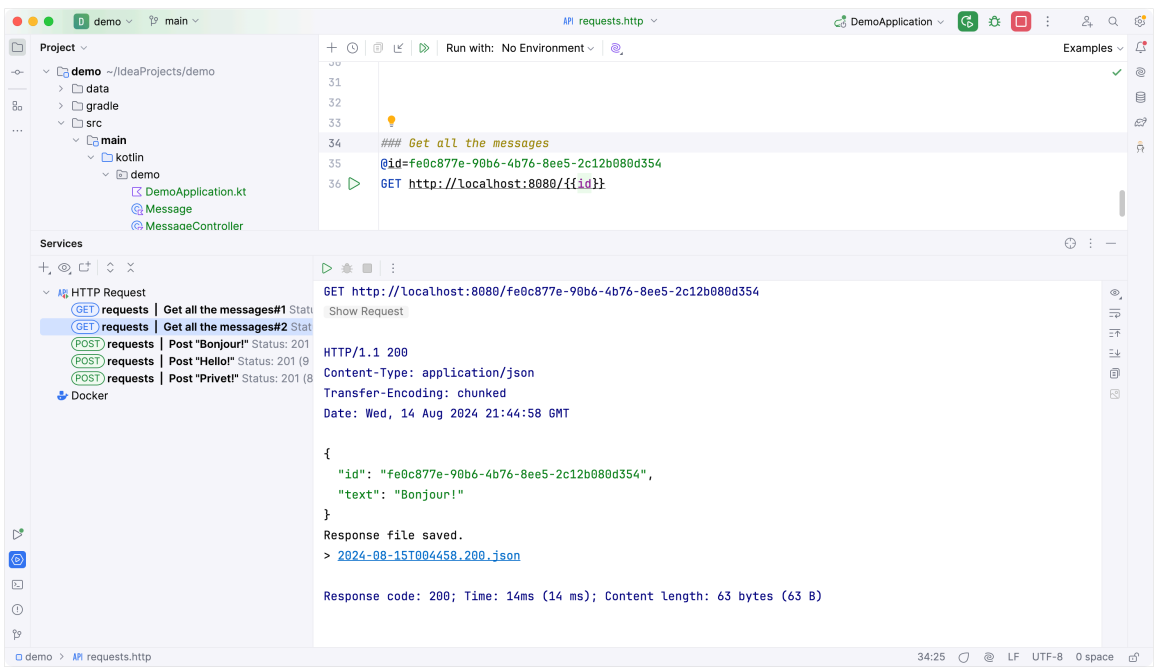
```
### 根据 id 得到 message
GET http://localhost:8080/id
```

3. 执行 GET 请求, 从数据库得到所有的 message.
4. 在 **Run** 工具窗口, 复制某个 message 的 id, 并添加到请求中, 例如:

```
### 根据 id 得到 message
GET http://localhost:8080/f16c1d2e-08dc-455c-abfe-68440229b84f
```

> 请使用你的 message 的真实 id, 不要使用上面例子中的值.

5. 执行 GET 请求, 并在 **Run** 工具窗口中查看结果:



根据 id 得到 message

下一步

本教程的最后部分会向你演示, 如何使用更加流行的数据库操作方式 Spring Data.

阅读下一章 (使用 Spring Data CrudRepository 进行数据库访问)

使用 Spring Data CrudRepository 进行数据库访问

这是 **Spring Boot 和 Kotlin 入门** 教程的最后部分. 开始这一部分之前, 请确认你已经完成了前面的步骤:
① 使用 Kotlin 创建 Spring Boot 项目 (使用 Kotlin 创建 Spring Boot 项目) ② 向 Spring Boot 项目添加数据类 (向 Spring Boot 项目添加数据类) ③ 为 Spring Boot 项目添加数据库支持 (为 Spring Boot 项目添加数据库支持) ④ **使用 Spring Data CrudRepository 进行数据库访问**

最终更新: 2024/09/10

在这一章中, 你将会迁移服务层, 使用 Spring Data (https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) `CrudRepository` 进行数据库访问, 而不是原来的 `JdbcTemplate`. *CrudRepository* 是一个 Spring Data 接口, 可以指定类型的仓库进行通常的 CRUD (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) 操作. 它提供了一些现成的方法来操作数据库.

更新你的应用程序

首先, 你需要调整 `Message` 类, 来配合 `CrudRepository` API:

1. 向 `Message` 类添加 `@Table` 注解, 声明它与数据库表的映射关系. 在 `id` 属性之前添加 `@Id` 注解.

   > 这些注解也需要额外的 import.

   ```
   import org.springframework.data.annotation.Id
   import org.springframework.data.relational.core.mapping.Table

   @Table("MESSAGES")
   data class Message(@Id var id: String?, val text: String)
   ```

除了添加这些注解之外, 你还需要让 `id` 成为可变属性 (`var`), 因为在将新对象插入到数据库时 `CrudRepository` 需要如此.

2. 为`CrudRepository` 声明一个接口, 它负责操作 `Message` 数据类:

```
import org.springframework.data.repository.CrudRepository

interface MessageRepository : CrudRepository<Message, String>
```

3. 更新 `MessageService` 类. 它现在调用 `MessageRepository`, 而不是执行 SQL 查询:

```
import java.util.*

@Service
class MessageService(val db: MessageRepository) {
    fun findMessages(): List<Message> = db.findAll().toList()

    fun findMessageById(id: String): List<Message> =
db.findById(id).toList()

    fun save(message: Message) {
        db.save(message)
    }

    fun <T : Any> Optional<out T>.toList(): List<T> =
        if (isPresent) listOf(get()) else emptyList()
}
```

扩展函数

CrudRepository 接口中 findById() 函数的返回类型是 Optional 类的实例. 但是, 从代码统一的角度来说, 返回一个包含单个 message 的 List 会更加方便. 要做到这一点, 如果 Optional 中有值, 你需要解包这个值, 并返回一个包含这个值的 List. 这部分功能可以实现为 Optional 类型的一个 扩展函数 ("扩展函数(Extension Function)" in "扩展").

在上面的代码中, Optional<out T>.toList(), .toList() 是 Optional 的扩展函数. 使用扩展函数, 你可以向任何类添加额外的函数, 当你想要扩展某些库中的类的功能时, 这样会非常有用.

CrudRepository save() 函数

这个函数的工作方式 (https://docs.spring.io/spring-data/jdbc/docs/current/reference/html/#jdbc.entity-persistence) 是假定新的对象在数据库中没有 id. 因此, 对 insertion 操作, id **需要为 null**.

如果 id 不是 , CrudRepository 假定对象在数据库中已经存在, 并且这是一个 操作, 而不是 操作. 在 insert 操作之后, id 会由数据库生成, 并反过来赋值给 Message 实例. 这就是 id 属性需要使用 var 关键字来声明的原因.

4. 更新 messages 表定义, 对 insert 的对象生成 id. 由于 `id` 是一个字符串, 你可以使用 `RANDOM_UUID()` 函数来生成默认的 id 值:

```
CREATE TABLE IF NOT EXISTS messages (
    id       VARCHAR(60)  DEFAULT RANDOM_UUID() PRIMARY KEY,
    text     VARCHAR      NOT NULL
);
```

5. 更新 `src/main/resources` 文件夹中的 `application.properties` 文件内的数据库名称:

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./data/testdb2
spring.datasource.username=name
spring.datasource.password=password
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.mode=always
```

下面是 `DemoApplication.kt` 的完整代码:

```
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.data.annotation.Id
import org.springframework.data.relational.core.mapping.Table
import org.springframework.data.repository.CrudRepository
import org.springframework.stereotype.Service
import org.springframework.web.bind.annotation.*
```

```
import java.util.*


@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

@RestController
class MessageController(val service: MessageService) {
    @GetMapping("/")
    fun index(): List<Message> = service.findMessages()

    @GetMapping("/{id}")
    fun index(@PathVariable id: String): List<Message> =
        service.findMessageById(id)

    @PostMapping("/")
    fun post(@RequestBody message: Message) {
        service.save(message)
    }
}

interface MessageRepository : CrudRepository<Message, String>

@Table("MESSAGES")
data class Message(@Id var id: String?, val text: String)

@Service
class MessageService(val db: MessageRepository) {
    fun findMessages(): List<Message> = db.findAll().toList()

    fun findMessageById(id: String): List<Message> =
db.findById(id).toList()

    fun save(message: Message) {
        db.save(message)
```

```
    }

    fun <T : Any> Optional<out T>.toList(): List<T> =
        if (isPresent) listOf(get()) else emptyList()
}
```

运行应用程序

应用程序可以在此运行了. 通过将 `JdbcTemplate` 替换为 `CrudRepository`, 功能并没有变更, 因此应用程序应该和以前相同的方式运行.

下一步做什么

得到你个人的语言导航地图, 它可以帮助你浏览 Kotlin 的功能特性, 并追踪你学习语言的进度:

Get the Kotlin Language Map →

得到 Kotlin 语言导航地图

(https://resources.jetbrains.com/storage/products/kotlin/docs/Kotlin_Language_Features_Map.pdf)

- 学习如何 在 Kotlin 中调用 Java 代码 (在 Kotlin 中调用 Java 代码) 和 在 Java 中调用 Kotlin 代码 (在 Java 中调用 Kotlin 代码).
- 学习如何使用 Java 到 Kotlin 转换器 ("使用 J2K 将既有的 Java 文件转换为 Kotlin" in "教程 - 在同一个项目中混合使用 Java 和 Kotlin") 将既有的 Java 代码转换为 Kotlin.
- 阅读我们的 Java 代码向 Kotlin 迁移指南:
  - Java 和 Kotlin 中的字符串 (Java 和 Kotlin 中的字符串).
  - Java 和 Kotlin 中的集合(Collection) (Java 和 Kotlin 中的集合(Collection)).
  - Java 和 Kotlin 中的可空性(Nullability) (Java 和 Kotlin 中的可空性(Nullability)).

教程 - 在 JVM 平台使用 JUnit 进行代码测试

最终更新: 2024/09/10

本教程将向你演示如何编写简单的单元测试, 并使用 Gradle 构建工具来运行测试.

教程中的示例程序使用了 kotlin.test (https://kotlinlang.org/api/latest/kotlin.test/index.html) 库, 并使用 JUnit 运行测试.

开始之前, 首先请下载并安装最新版的 IntelliJ IDEA (https://www.jetbrains.com/idea/download/index.html).

添加依赖项

1. 在 IntelliJ IDEA 中打开一个 Kotlin 项目. 如果你还没有项目, 请 创建一个新项目 ("创建应用程序" in "Kotlin/JVM 入门").

   > 创建项目时请选择 **JUnit 5** 作为测试框架.

2. 打开 `build.gradle(.kts)` 文件, 并向 Gradle 配置添加以下依赖项. 通过这个依赖项, 你将可以使用 `kotlin.test` 和 `JUnit`:

   Kotlin

   ```
   dependencies {
       // 其他依赖项.
       testImplementation(kotlin("test"))
   }
   ```

   Groovy

   ```
   dependencies {
       // 其他依赖项.
   ```

```
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}
```

3. 向 build.gradle(.kts) 文件添加 test 任务:

Kotlin

```
tasks.test {
    useJUnitPlatform()
}
```

Groovy

```
test {
    useJUnitPlatform()
}
```

如果你使用 **New Project** 向导创建项目, 会自动添加这个任务.

添加需要测试的代码

1. 打开 src/main/kotlin 中的 main.kt 文件.

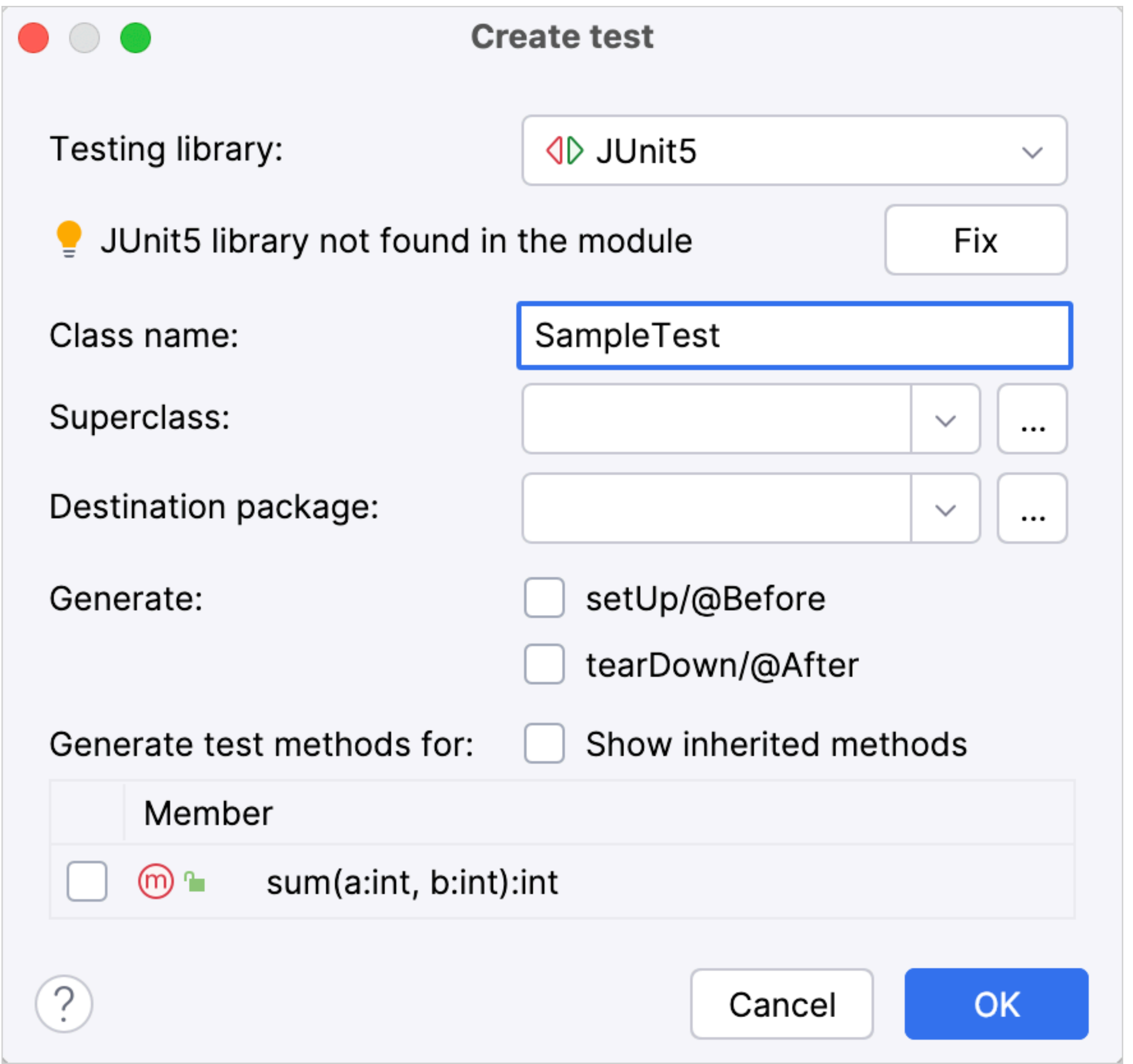
   src 目录包含 Kotlin 源代码文件和资源文件. main.kt 文件包含示例代码, 它将打印输出 Hello, World!.

2. 创建 Sample 类, 包含 sum() 函数, 它会将两个整数加在一起:

```
class Sample() {

    fun sum(a: Int, b: Int): Int {
        return a + b
    }
}
```

创建测试

1. 在 IntelliJ IDEA 中, 对 `Sample` 类选择 **Code** | **Generate** | **Test...**.



创建测试

2. 输入测试类的名称. 比如, `SampleTest`.

   IntelliJ IDEA 会在 `test` 目录中创建 `SampleTest.kt` 文件. 这个目录包含 Kotlin 测试源代码文件和资源文件.

   > 你也可以在 `src/test/kotlin` 目录中为测试代码手动创建一个 `*.kt` 文件.

3. 在 `SampleTest.kt` 中为 `sum()` 函数添加测试代码:

* 使用 @Test 注解 (https://kotlinlang.org/api/latest/kotlin.test/kotlin.test/-test/index.html), 定义测试函数 `testSum()`.
* 使用 assertEquals() (https://kotlinlang.org/api/latest/kotlin.test/kotlin.test/assert-equals.html) 函数, 检查 `sum()` 函数是否返回了期待的值.

```
import kotlin.test.Test
import kotlin.test.assertEquals

internal class SampleTest {

    private val testSample: Sample = Sample()

    @Test
    fun testSum() {
        val expected = 42
        assertEquals(expected, testSample.sum(40, 2))
    }
}
```

运行测试

1. 使用源代码编辑器侧栏中的图标运行测试.

运行测试

> 你也可以在命令行执行命令 `./gradlew check`, 运行整个项目的所有测试.

2. 在 **Run** 工具窗口检查测试结果:

   测试函数执行成功了.

3. 将 `expected` 变量值修改为 43, 确认测试正确工作:

```
@Test
fun testSum() {
    val expected = 43
    assertEquals(expected, classForTesting.sum(40, 2))
}
```

4. 再次运行测试, 并检查结果:

   测试执行失败.

下一步做什么?

完成你的第一个测试之后, 你可以:

* 尝试使用 kotlin.test (https://kotlinlang.org/api/latest/kotlin.test/kotlin.test/) 的其他函数, 编写其他测试. 比如, 你可以使用 `assertNotEquals()` (https://kotlinlang.org/api/latest/kotlin.test/kotlin.test/assert-not-equals.html) 函数.
* 使用 Kotlin 和 Spring Boot 创建你的第一个应用程序 (Spring Boot 和 Kotlin 入门).
* 在 YouTube 上观看 这些视频教程 (https://www.youtube.com/playlist?list=PL6gx4Cwl9DGDPsneZWaOFg0H2wsundyGr), 这些教程将演示如何与 Kotlin 和 JUnit 5 一起配合使用 Spring Boot.

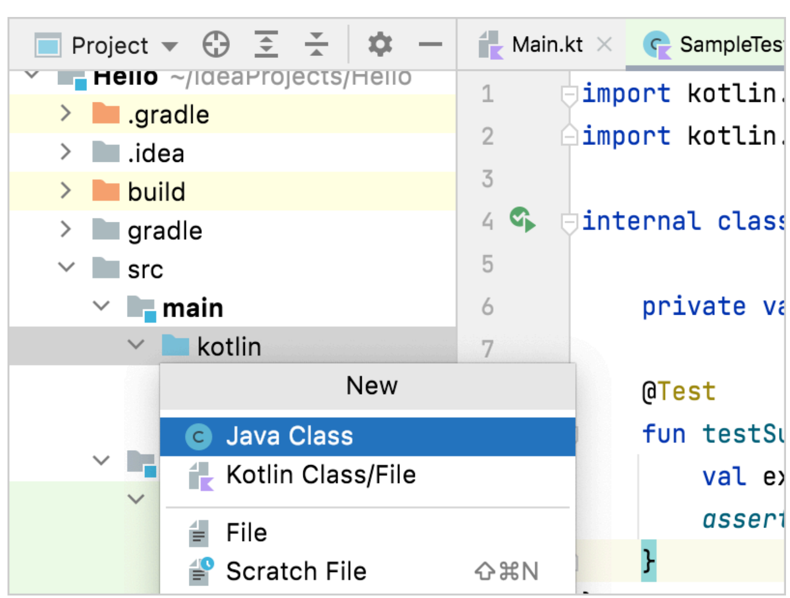
教程 - 在同一个项目中混合使用 Java 和 Kotlin

最终更新: 2024/09/10

Kotlin 对 Java 的交互功能提供了一级支持, 现代化的 IDE 还使得这个功能更加便利. 本教程中, 你将会学习在 IntelliJ IDEA 如何在同一个项目内同时使用 Kotlin 和 Java 代码. 关于在 IntelliJ IDEA 中如何启动一个新的 Kotlin 项目, 请参见 IntelliJ IDEA 中使用 Kotlin 入门 (Kotlin/JVM 入门).

向既有的 Kotlin 项目添加 Java 源代码

向 Kotlin 项目添加 Java 类非常简单. 你只需要创建一个新的 Java 文件. 在你的项目内选择一个目录或包, 然后选择菜单 **File** | **New** | **Java Class**, 或者使用快捷键 **Alt + Insert/Cmd + N**.



添加新的 Java 类

如果你已经有了 Java 类, 你可以直接将它们复制到项目目录内.

然后你就可以在 Kotlin 代码中使用 Java 类, 或者反过来, 不需要其它任何工作.

比如, 添加下面的 Java 类:

```
public class Customer {

    private String name;
```

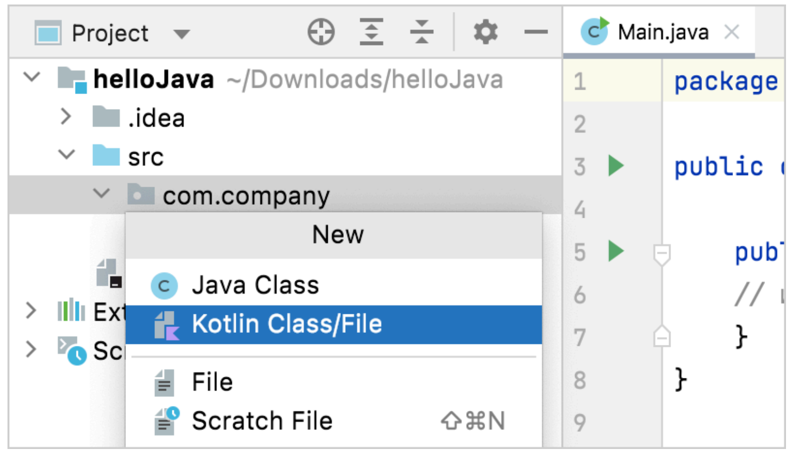
```
    public Customer(String s){
        name = s;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void placeOrder() {
        System.out.println("A new order is placed by " + name);
    }
}
```

下面在 Kotlin 代码中使用这个类, 就象使用 Kotlin 中的其他类型一样.

```
val customer = Customer("Phase")
println(customer.name)
println(customer.placeOrder())
```

向既有的 Java 项目添加 Kotlin 源代码

向既有的 Java 项目添加 Kotlin 源代码, 方法基本相同.



添加新的 Kotlin 类

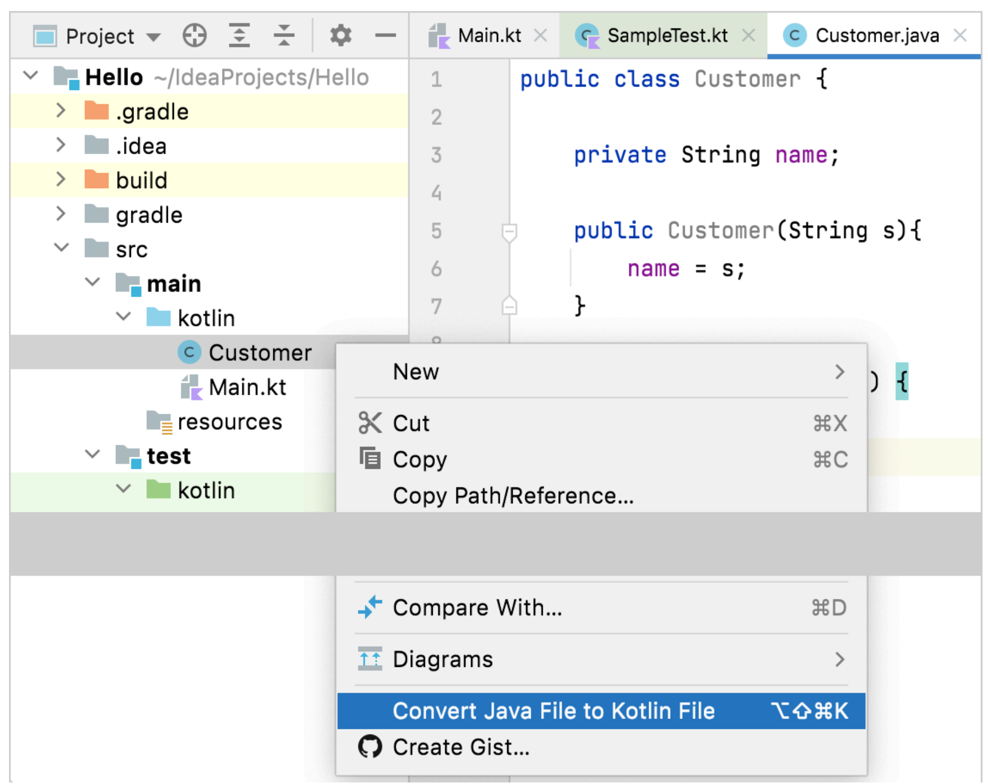
如果这是你第一次向这个项目添加 Kotlin 文件, IntelliJ IDEA 会自动添加需要的 Kotlin 运行时库.

绑定 Kotlin 运行时库

你也可以从菜单 **Tools** | **Kotlin** | **Configure Kotlin in Project** 手动打开 Kotlin 运行时库配置.

使用 J2K 将既有的 Java 文件转换为 Kotlin

Kotlin plugin 还带有一个 Java 到 Kotlin 的转换器 (*J2K*), 它可以将 Java 文件自动转换为 Kotlin. 要对一个源代码文件使用 J2K, 请在它的弹出菜单中, 或在 IntelliJ IDEA 的 **Code** 菜单中, 点击 **Convert Java File to Kotlin File**.



将 Java 文件转换为 Kotlin

转换器并不保证完全正确, 但它确实能将绝大部分样板代码从 Java 正确的转换为 Kotlin. 有时会需要进行一些手动的修正.

在 Kotlin 中使用 Java 记录类(Record)

最终更新: 2024/09/10

在 Java 中 *记录类(Record)* 是用于存储不可变数据的 类 (https://openjdk.java.net/jeps/395). 记录类携带一组固定的值 – *记录组件(Records Components)*. 在 Java 中记录类的语法很简洁, 可以为你节省编写样板代码的时间:

```
// Java
public record Person (String name, int age) {}
```

编译器会自动生成一个 final 类, 继承自 `java.lang.Record` (https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Record.html), 并包含以下成员:

- 对每个记录组件, 有一个 private final 域
- 一个 public 构造器, 参数是所有的域
- 一组方法, 实现结构化的相等比较: `equals()`, `hashCode()`, `toString()`
- 为读取每个记录组件, 有一个 public 方法

记录类非常类似于 Kotlin 数据类 (数据类(Data Class)).

在 Kotlin 代码中使用 Java 记录类

在 Kotlin 中, 你可以通过使用类和属性相同的方式来使用 Java 中声明的记录类及其组件. 要访问记录组件, 可以和使用 Kotlin 属性 (属性(Property)) 一样, 直接使用它的名称:

```
val newPerson = Person("Kotlin", 10)
val firstName = newPerson.name
```

在 Kotlin 中声明记录类

Kotlin 只支持对数据类声明记录类, 而且数据类必须符合 要求.

要在 Kotlin 中声明一个数据类, 请使用 `@JvmRecord` 注解:

向一个已存在的类添加 @JvmRecord, 这样的变化会导致二进制不兼容. 会改变类属性访问器的命名规约.

```
@JvmRecord
data class Person(val name: String, val age: Int)
```

这个 JVM 专用的注解会导致生成以下内容:

- 在 class 文件中, 生成与类属性对应的记录组件
- 符合 Java 记录类命名规约的类属性访问器方法名

数据类会提供 `equals()`, `hashCode()`, 和 `toString()` 方法的实现.

要求

要使用 `@JvmRecord` 注解来声明数据类, 它必须符合以下要求:

- 类所在模块的编译目标必须是 JVM 16 字节码 (或者 JVM 15, 并开启 `-Xjvm-enable-preview` 编译器选项).
- 类不能明确继承任何其他类 (包括 `Any`), 因为所有的 JVM 记录类都要隐含的继承 `java.lang.Record`. 但是, 类可以实现接口.
- 类不能声明任何带有后端域(Backing Field)的属性 – 通过主构造器参数初始化的对应属性除外.
- 类不能声明任何带有后端域(Backing Field)的可变属性.
- 不能是局部(local)类.
- 类的主构造器的可见度必须与类本身相同.

允许使用 JVM 记录类

对生成的 JVM 字节码, JVM 记录类要求的编译目标为 `16` 或更高版本.

要明确指定字节码版本, 请在 Gradle ("JVM 任务独有的属性" in "Kotlin Gradle plugin 中的编译器选项") 或 Maven ("JVM 独有的属性" in "Maven")中, 使用 `jvmTarget` 编译器选项.

更多讨论

关于更多技术细节和讨论, 请参见 对 JVM 记录类的语言规格建议
(https://github.com/Kotlin/KEEP/blob/master/proposals/jvm-records.md).

Java 和 Kotlin 中的字符串

最终更新: 2024/09/10

这篇向导通过示例程序演示如何在 Java 和 Kotlin 中进行通常的字符串处理. 将会帮助你从 Java 迁移到 Kotlin, 并以 Kotlin 的方式来编写代码.

拼接字符串

在 Java 中, 你可以通过以下方式实现:

```
// Java
String name = "Joe";
System.out.println("Hello, " + name);
System.out.println("Your name is " + name.length() + " characters
long");
```

在 Kotlin 中, 可以在变量名称之前使用 `$` 符号, 将这个变量的值添加到你的字符串之内:

```
fun main() {
//sampleStart
    // Kotlin
    val name = "Joe"
    println("Hello, $name")
    println("Your name is ${name.length} characters long")
//sampleEnd
}
```

你可以在字符串内添加复杂表达式的值, 方法是将表达式用括号括起, 比如 `${name.length}`. 详情请参见 字符串模板 (["字符串模板" in "字符串"]()).

构建一个字符串

在 Java 中, 你可以使用 StringBuilder
(https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StringBuilder.html
):

```
// Java
StringBuilder countDown = new StringBuilder();
for (int i = 5; i > 0; i--) {
    countDown.append(i);
    countDown.append("\n");
}
System.out.println(countDown);
```

在 Kotlin 中, 请使用 buildString() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/build-string.html) – 一个 内联函数 (内联函数(Inline Function)), 参数是一个 Lambda 表达式, 其中包含创建字符串的代码:

```
fun main() {
//sampleStart
       // Kotlin
       val countDown = buildString {
           for (i in 5 downTo 1) {
               append(i)
               appendLine()
           }
       }
       println(countDown)
//sampleEnd
}
```

在 buildString 的内部, 它使用 Java 中相同的 StringBuilder 类, 在 Lambda 表达式 ("带有接受者的函数字面值" in "高阶函数与 Lambda 表达式") 内, 你通过隐含的 this 访问它.

更多详情请参见 Lambda 表达式的编码规约 ("Lambda 表达式" in "编码规约").

通过集合中的元素创建一个字符串

在 Java 中, 你可以使用 Stream API (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html) 来过滤, 变换, 并收集元素:

```
// Java
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
String invertedOddNumbers = numbers
```

```
        .stream()
        .filter(it -> it % 2 != 0)
        .map(it -> -it)
        .map(Object::toString)
        .collect(Collectors.joining("; "));
System.out.println(invertedOddNumbers);
```

在 Kotlin 中, 请使用 joinToString() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/join-to-string.html) 函数, Kotlin 对所有的 List 都定义了这个函数:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf(1, 2, 3, 4, 5, 6)
    val invertedOddNumbers = numbers
        .filter { it % 2 != 0 }
        .joinToString(separator = ";") {"${-it}"}
    println(invertedOddNumbers)
//sampleEnd
}
```

在 Java 中, 如果想要在分隔符与后续元素之间插入空格, 你需要在分隔符中明确的加入空格.

详情请参见 joinToString() ("字符串表达(String representation)" in "集合变换操作") 的使用方法.

当字符串为空白时设置默认值

在 Java 中, 你可以使用 三元运算符(Ternary Operator) (https://en.wikipedia.org/wiki/%3F:):

```
// Java
public void defaultValueIfStringIsBlank() {
    String nameValue = getName();
    String name = nameValue.isBlank() ? "John Doe" : nameValue;
    System.out.println(name);
}
```

```
public String getName() {
    Random rand = new Random();
    return rand.nextBoolean() ? "" : "David";
}
```

Kotlin 提供了内联函数 ifBlank() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/if-blank.html), 它接受一个默认值作为参数:

```
// Kotlin
import kotlin.random.Random

//sampleStart
fun main() {
    val name = getName().ifBlank { "John Doe" }
    println(name)
}

fun getName(): String =
    if (Random.nextBoolean()) "" else "David"
//sampleEnd
```

替换一个字符串的最首字符和最末字符

在 Java 中, 你可以使用 replaceAll() (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html#replaceAll(java.lang.String,java.lang.String)) 函数. 在这里 `replaceAll()` 函数接受正则表达式 `^##` 与 `##$`, 分别定义以 `##` 开始和以 `##` 结束的字符串:

```
// Java
String input = "##place##holder##";
String result = input.replaceAll("^##|##$", "");
System.out.println(result);
```

在 Kotlin 中, 请使用 removeSurrounding() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/remove-surrounding.html) 函数, 使用 `##` 作为:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "##place##holder##"
    val result = input.removeSurrounding("##")
    println(result)
//sampleEnd
}
```

查找与替换

在 Java 中, 你可以使用 Pattern (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html) 和 Matcher (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Matcher.html) 类, 比如, 要混淆某些数据:

```
// Java
String input = "login: Pokemon5, password: 1q2w3e4r5t";
Pattern pattern = Pattern.compile("\\w*\\d+\\w*");
Matcher matcher = pattern.matcher(input);
String replacementResult = matcher.replaceAll(it -> "xxx");
System.out.println("Initial input: '" + input + "'");
System.out.println("Anonymized input: '" + replacementResult + "'");
```

在 Kotlin 中, 你可以使用 Regex (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-regex/) 类, 它可以简化正规表达式的相关操作. 此外, 可以使用 多行字符串(Multiline String) ("多行(Multiline)字符串" in "字符串") 来减少反斜杠的数量, 简化正规表达式的书写:

```
fun main() {
//sampleStart
    // Kotlin
    val regex = Regex("""\w*\d+\w*""") // 多行字符串
    val input = "login: Pokemon5, password: 1q2w3e4r5t"
    val replacementResult = regex.replace(input, replacement =
"xxx")
    println("Initial input: '$input'")
    println("Anonymized input: '$replacementResult'")
```

```
//sampleEnd
}
```

字符串切分

在 Java 中, 要使用句号字符 (.)切分一个字符串, 你需要使用转义 (\\). 因为 String 类的 split() (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html#split(java.lang.String)) 函数接受一个正规表达式作为参数:

```
// Java
System.out.println(Arrays.toString("Sometimes.text.should.be.split".
split("\\.")));
```

在 Kotlin 中, 请使用 Kotlin 函数 split() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/split.html), 它接受可变数量的分隔符作为参数:

```
fun main() {
//sampleStart
    // Kotlin
    println("Sometimes.text.should.be.split".split("."))
//sampleEnd
}
```

如果你需要使用正规表达式切分字符串, 请使用 split() 函数接受 Regex 作为参数的重载版本.

获取子字符串

在 Java 中, 你可以使用 substring() (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html#substring(int)) 函数, 它接受一个起始字符下标作为参数, 从这个位置开始获取子字符串. 要获取这个字符之后的子字符串, 你需要对下标加 1:

```
// Java
String input = "What is the answer to the Ultimate Question of Life,
the Universe, and Everything? 42";
String answer = input.substring(input.indexOf("?") + 1);
System.out.println(answer);
```

在 Kotlin 中, 请使用 substringAfter() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/substring-after.html) 函数, 想要获取某个字符之后的子字符串时, 不需要计算下标:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "What is the answer to the Ultimate Question of
Life, the Universe, and Everything? 42"
    val answer = input.substringAfter("?")
    println(answer)
//sampleEnd
}
```

此外, 你还可以获取某个字符最后一次出现位置之后的子字符串:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "To be, or not to be, that is the question."
    val question = input.substringAfterLast(",")
    println(question)
//sampleEnd
}
```

使用多行字符串

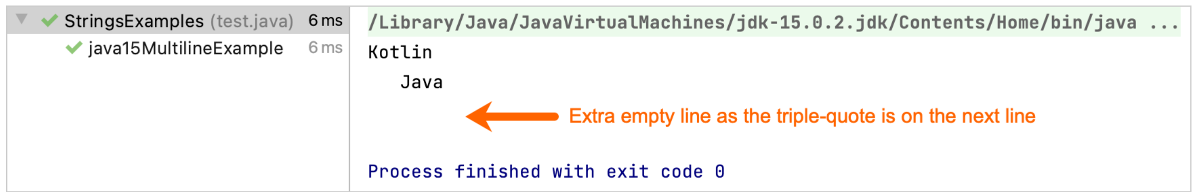
在 Java 15 之前, 有几种方法创建多行字符串. 比如, 使用 `String` 类的 join() (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html#join(java.lang.CharSequence,java.lang.CharSequence...)) 函数:

```
// Java
String lineSeparator = System.getProperty("line.separator");
String result = String.join(lineSeparator,
       "Kotlin",
       "Java");
System.out.println(result);
```

在 Java 15 中, 有了 文本块(Text Block) (https://docs.oracle.com/en/java/javase/15/text-blocks/index.html) 功能. 但需要记住: 如果你打印一个多行字符串, 而且三重引号出现在下一行, 那么会存在一个额外的空行:

```
// Java
String result = """
    Kotlin
       Java
    """;
System.out.println(result);
```

输出结果是:



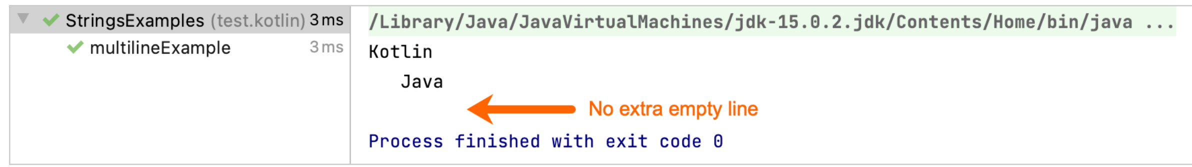
Java 15 的多行字符串输出

如果你将三重引号放在最后一个词的同一行, 那么这个差别会消失.

在 Kotlin 中, 你可以对引号放在新行的字符串进行格式化, 输出中不会存在额外的空行. 每行的最左侧字符标识这一行的起始. 与 Java 的区别是, Java 会自动删除缩进字符, 而在 Kotlin 中你需要明确的删除:

```
fun main() {
//sampleStart
    // Kotlin
    val result = """
        Kotlin
           Java
    """.trimIndent()
    println(result)
//sampleEnd
}
```

输出结果是:



Kotlin 的多行字符串输出

如果要输出额外的空行, 你需要在你的多行字符串中明确的添加这个空行.

在 Kotlin 中, 你也可以使用 trimMargin() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/trim-margin.html) 函数 修改缩进:

```
// Kotlin
fun main() {
    val result = """
       #  Kotlin
       #  Java
   """.trimMargin("#")
    println(result)
}
```

详情请参见 多行字符串 ("字符串" in "编码规约").

下一步做什么?

- 学习 Kotlin 惯用法 (惯用法).
- 学习如何使用 Java 到 Kotlin 的转换器 ("使用 J2K 将既有的 Java 文件转换为 Kotlin" in "教程 - 在同一个项目中混合使用 Java 和 Kotlin"), 将既有的 Java 代码转换为 Kotlin .

如果你有喜欢的惯用法, 欢迎提交一个 pull request, 共享给大家.

Java 和 Kotlin 中的集合(Collection)

最终更新: 2024/09/10

集合 是一组可变数量(可以为 0)的元素, 解决问题时起到重要作用, 而且经常被用到. 本文解释并比较 Java 和 Kotlin 中集合的概念以及操作方式. 本文将帮助你从 Java 迁移到 Kotlin, 并以真正 Kotlin 的方式编写你的代码.

本文第 1 部分包括在 Java 和 Kotlin 中对同一个集合进行操作的快速介绍. 分为 共同的操作 和 只存在于 Kotlin 中的操作. 本文第 2 部分, 从 可变性(Mutability) 开始, 通过例子来解释一些区别.

关于集合的介绍, 请参见 集合概述 (集合(Collection)概述), 或观看 Sebastian Aigner 讲解的这个视频 (https://www.youtube.com/watch?v=F8jj7e-_jFA), 他是 Kotlin 开发者 Advocate.

> 下文中的所有示例都只使用 Java 和 Kotlin 标准库 API.

在 Java 和 Kotlin 中相同的操作

在 Kotlin 中, 有很多集合操作与在 Java 中的对应操作完全相同.

对 List, Set, Queue, 和 Deque 的操作

描述	共通操作	Kotlin 中的更多选择
添加一个或多个元素	add(), addAll()	使用 加然后赋值(plusAssign) (+=) 操作符 (加法(Plus)和减法(Minus)操作符): collection += element, collection += anotherCollection.
检查集合是否包含一个或多个元素	contains(), containsAll()	使用 in 关键字 ("检测元素是否存在" in "获取集合的单个元素") 以操作符的形式调用 contains() 函数: element in collection.
检查集合是否为空	isEmpty()	使用 isNotEmpty() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/is-not-empty.html) 检查集合是否为非空.
指定条件删除	removeIf()	
只保留指定的元素	retainAll()	
从集合删除所有元素	clear()	
从集合得到一个 Stream	stream()	Kotlin 有自己的方式来处理 Stream: 序列(Sequence), 以及方法, 比如 map() (过滤(Filtering)集合) 和 filter().
从集合得到一个 Iterator	iterator()	

对 Map 的操作

描述	共通操作	Kotlin 中的更多选择
添加一个或多个元素	put(), putAll(), putIfAbsent()	在 Kotlin 中, 赋值操作 map[key] = value 的效果与 put(key, value) 相同. 你还可以使用 加然后赋值(plusAssign) (+=) 操作符 (加法(Plus)和减法(Minus)操作符): map += Pair(key, value) 或 map += anotherMap.
替换一个或多个元素	put(), replace(), replaceAll()	使用下标访问操作符 map[key] = value, 而不是 put() 和 replace().
得到元素	get()	使用下标访问操作符得到元素: map[index].
检查 Map 是否包含一个或多个元素	containsKey(), containsValue()	使用 in 关键字 ("检测元素是否存在" in "获取集合的单个元素") 以操作符形式调用 contains() 函数: element in map.
检查 Map 是否为空	isEmpty()	使用 isNotEmpty() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/is-not-empty.html) 检查 Map 是否为非空.
删除元素	remove(key), remove(key, value)	使用 减然后赋值(minusAssign) (-=) 操作符 (加法(Plus)和减法(Minus)操作符): map -= key.
从 Map 删除所有元素	clear()	
从 Map 得到一个 Stream	entries, keys, 或 values 的 stream() 函数	

只对 List 有效的操作

描述	共通操作	Kotlin 中的更多选择
得到元素下标	indexOf()	
得到元素的最后下标	lastIndexOf()	
得到元素	get()	使用下标访问操作符得到元素: list[index].
获取一个子 List	subList()	
替换一个或多个元素	set(), replaceAll()	使用下标访问操作符, 而不是 set(): list[index] = value.

略有不同的操作

对任何集合类型都有效的操作

描述	Java	Kotlin
得到集合的大小	size()	count(), size
平展访问(Flat Access) 嵌套的集合元素	collectionOfCollections.forEach(flatCollection::addAll) 或 collectionOfCollections.stream().flatMap().collect()	flatten() ("扁平化(Flattening)" in "集合变换操作") 或 flatMap() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/flat-map.html)
对每个元素使用指定的函数	stream().map().collect()	map() (过滤(Filtering)集合)
对集合元素顺序的使用指定的操作, 并返回累积的结果	stream().reduce()	reduce(), fold() ("折叠(fold) 与 简化(reduce)" in "聚合(Aggregate)操作")
通过一个分类器对元素分组, 并统计	stream().collect(Collectors.groupingBy(classifier, counting()))	eachCount() (分组(Grouping))
根据条件过滤	stream().filter().collect()	filter() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html)
检查集合元素是否满足条件	stream().noneMatch(), stream().anyMatch(), stream().allMatch()	none(), any(), all() (过滤(Filtering)集合)
对元素排序	stream().sorted().collect()	sorted() ("使用自然顺序排序" in "排序(Ordering)")
获取前 N 个元	stream().limit(N).collect()	take(N) ("提取(Take) 和 抛弃(Drop)" in

素		"获取集合的一部分")
指定条件获取元素	stream().takeWhile().collect()	takeWhile() ("提取(Take) 和 抛弃(Drop)" in "获取集合的一部分")
跳过前 N 个元素	stream().skip(N).collect()	drop(N) ("提取(Take) 和 抛弃(Drop)" in "获取集合的一部分")
指定条件跳过元素	stream().dropWhile().collect()	dropWhile() ("提取(Take) 和 抛弃(Drop)" in "获取集合的一部分")
构建从集合元素到关联值的 Map	stream().collect(toMap(keyMapper, valueMapper))	associate() ("关联(Association)" in "集合变换操作")

要对 Map 执行上述所有操作, 你首先需要得到 Map 的 entrySet.

对 List 的操作

描述	Java	Kotlin
按照自然顺序排序 List	sort(null)	sort()
按照逆序排序 List	sort(comparator)	sortDescending()
从 List 删除元素	remove(index), remove(element)	removeAt(index), remove(element) 或 collection -= element (加法(Plus)和减法(Minus)操作符)
将 List 的所有元素填充为指定的值	Collections.fill()	fill() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fill.html)
从 List 得到不重复的元素	stream().distinct().toList()	distinct() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/distinct.html)

在 Java 标准库中不存在的操作

- `zip()`, `unzip()` (集合变换操作) – 变换集合.
- `aggregate()` (分组(Grouping)) – 根据条件分组.
- `takeLast()`, `takeLastWhile()`, `dropLast()`, `dropLastWhile()` ("提取(Take) 和 抛弃(Drop)" in "获取集合的一部分") – 根据条件获取或删除元素.
- `slice()`, `chunked()`, `windowed()` (获取集合的一部分) – 获取集合的一部分.
- Plus (`+`) and minus (`-`) 操作符 (加法(Plus)和减法(Minus)操作符) – 添加或删除元素.

如果你想要深入了解 `zip()`, `chunked()`, `windowed()`, 以及其他一些操作, 请观看 Sebastian Aigner 讲解的这个视频, 关于Kotlin 中集合的高级操作:

可变性

在 Java 中, 有可变的集合:

```
// Java
// 这个 List 是可变的!
public List<Customer> getCustomers() { ... }
```

也有部分可变的集合:

```
// Java
List<String> numbers = Arrays.asList("one", "two", "three", "four");
numbers.add("five"); // 在运行时刻会发生
`UnsupportedOperationException` 错误
```

还有不可变的集合:

```
// Java
List<String> numbers = new LinkedList<>();
// 这个 List 是 不可变的!
List<String> immutableCollection =
Collections.unmodifiableList(numbers);
```

```
immutableCollection.add("five"); // 在运行时刻会发生
`UnsupportedOperationException` 错误
```

如果你在 IntelliJ IDEA 中编写后面两段代码, IDE 会提出警告, 告诉你正在修改不可变的对象. 这段代码能够编译, 并在运行时刻发生 `UnsupportedOperationException` 错误. 你不能通过集合的类型判断它是否可变.

与 Java 不同, 在 Kotlin 中, 你会根据需要明确声明可变的或只读的集合. 如果你试图修改只读集合, 代码将会无法编译:

```
// Kotlin
val numbers = mutableListOf("one", "two", "three", "four")
numbers.add("five")            // 这是正确的
val immutableNumbers = listOf("one", "two")
//immutableNumbers.add("five") // 编译错误 - 无法解析的引用: add
```

关于可变性, 详情请参见 Kotlin 编码规约 ("数据的不可变性" in "编码规约").

协变(Covariance)

在 Java 中, 如果函数的参数是祖先类型元素的集合, 那么你不能传递一个后代类型元素的集合. 比如, 如果 `Rectangle` 继承 `Shape`, 对于参数是 `Shape` 元素集合的函数, 你不能传递 `Rectangle` 元素类型的集合. 要让代码能够编译, 需要使用 `? extends Shape` 类型, 才能让函数接受从 `Shape` 继承的后代类型元素的集合:

```
// Java
class Shape {}

class Rectangle extends Shape {}

public void doSthWithShapes(List<? extends Shape> shapes) {
/* 如果只使用 List<Shape>, 那么如下面的例子那样, 使用 List<Rectangle> 作为
参数调用
   这个函数时, 代码将无法编译 */
}

public void main() {
    var rectangles = List.of(new Rectangle(), new Rectangle());
```

```
    doSthWithShapes(rectangles);
}
```

在 Kotlin 中, 只读集合类型是 协变的(Covariant) ("类型变异(Variance)" in "泛型(Generic): in, out, where"). 因此, 如果 Rectangle 类继承自 Shape 类, 那么在要求 List<Shape> 类型的地方, 你可以使用 List<Rectangle> 类型. 也就是说, 集合类型之间的子类型关系与元素类型之间相同. Map 根据 value 类型协变, 而不是根据 key 类型. 可变的集合不是协变的 – 否则会导致运行时错误.

```
// Kotlin
open class Shape(val name: String)

class Rectangle(private val rectangleName: String) :
Shape(rectangleName)

fun doSthWithShapes(shapes: List<Shape>) {
    println("The shapes are: ${shapes.joinToString { it.name }}")
}

fun main() {
    val rectangles = listOf(Rectangle("rhombus"),
Rectangle("parallelepiped"))
    doSthWithShapes(rectangles)
}
```

详情请参见 集合类型 ("集合类型" in "集合(Collection)概述").

值范围(Range)与数列(Progression)

在 Kotlin 中, 你可以使用 值范围(Range) (值范围(Range)与数列(Progression)) 创建数值范围. 比如, Version(1, 11)..Version(1, 30) 包括从 1.11 到 1.30 的所有版本. 你可以使用 in 操作符检查你的版本是否在范围中: Version(0, 9) in versionRange.

在 Java 中, 你需要手动检查一个 Version 是否在边界条件之内:

```
// Java
class Version implements Comparable<Version> {
    int major;
    int minor;
```

```
    Version(int major, int minor) {
        this.major = major;
        this.minor = minor;
    }

    @Override
    public int compareTo(Version o) {
        if (this.major != o.major) {
            return this.major - o.major;
        }
        return this.minor - o.minor;
    }
}

public void compareVersions() {
    var minVersion = new Version(1, 11);
    var maxVersion = new Version(1, 31);

   System.out.println(
           versionIsInRange(new Version(0, 9), minVersion,
maxVersion));
   System.out.println(
           versionIsInRange(new Version(1, 20), minVersion,
maxVersion));
}

public Boolean versionIsInRange(Version versionToCheck, Version
minVersion,
                                Version maxVersion) {
    return versionToCheck.compareTo(minVersion) >= 0
            && versionToCheck.compareTo(maxVersion) <= 0;
}
```

在 Kotlin 中, 你可以将值范围当作整个对象来操作. 你不需要创建两个变量, 并分别与 `Version` 进行比较:

```
// Kotlin
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
```

```
        if (this.major != other.major) {
            return this.major - other.major
        }
        return this.minor - other.minor
    }
}

fun main() {
    val versionRange = Version(1, 11)..Version(1, 30)

    println(Version(0, 9) in versionRange)
    println(Version(1, 20) in versionRange)
}
```

如果你需要排除边界值, 比如检查一个版本是否大于或等于 (`>=`) 最小版本, 并且小于 (`<`) 最大版本, 那么这种包含边界值的值范围无法适用.

根据多个条件比较

在 Java 中, 要根据多个条件比较对象, 你可以使用 `Comparator` (https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) 接口的 `comparing()` (https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparing-java.util.function.Function-) 和 `thenComparingX()` (https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparing-java.util.Comparator-) 函数 . 比如, 要按照姓名和年龄比较人:

```
class Person implements Comparable<Person> {
    String name;
    int age;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    Person(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return this.name + " " + age;
    }
}

public void comparePersons() {
    var persons = List.of(new Person("Jack", 35), new
Person("David", 30),
            new Person("Jack", 25));
    System.out.println(persons.stream().sorted(Comparator
            .comparing(Person::getName)
            .thenComparingInt(Person::getAge)).collect(toList()));
}
```

在 Kotlin 中, 你只需要列举你希望比较的属性:

```
data class Person(
    val name: String,
    val age: Int
)

fun main() {
    val persons = listOf(Person("Jack", 35), Person("David", 30),
        Person("Jack", 25))
    println(persons.sortedWith(compareBy(Person::name,
Person::age)))
}
```

序列(Sequence)

在 Java 中, 你可以这样生成一个数值序列(Sequence):

```
// Java
int sum = IntStream.iterate(1, e -> e + 3)
    .limit(10).sum();
System.out.println(sum); // 输出结果为 145
```

在 Kotlin 中, 请使用 *序列(Sequence)* (*序列(Sequence)*). 对序列的多个步骤处理会尽可能延迟执行 – 只有在需要整个处理串的结果时, 实际的计算才会发生.

```
fun main() {
//sampleStart
    // Kotlin
    val sum = generateSequence(1) {
        it + 3
    }.take(10).sum()
    println(sum) // 输出结果为 145
//sampleEnd
}
```

对于一些过滤操作, 序列可能会减少需要执行的步骤. 详情请参见 序列的处理示例 ("序列处理的示例" in "序列(Sequence)"), 这篇文档会演示 `Iterable` 和 `Sequence` 的区别.

从 List 删除元素

在 Java 中, `remove()` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html#remove(int)) 函数接受需要删除的元素下标.

要删除整数元素时, 使用 `Integer.valueOf()` 函数作为 `remove()` 函数的参数:

```
// Java
public void remove() {
    var numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    numbers.add(1);
    numbers.remove(1); // 这里根据下标删除元素
    System.out.println(numbers); // [1, 3, 1]
    numbers.remove(Integer.valueOf(1));
```

```
    System.out.println(numbers); // [3, 1]
}
```

在 Kotlin 中, 有 2 种类型的元素删除函数: 根据下标删除 `removeAt()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/remove-at.html), 以及根据值删除 `remove()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/remove.html).

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = mutableListOf(1, 2, 3, 1)
    numbers.removeAt(0)
    println(numbers) // [2, 3, 1]
    numbers.remove(1)
    println(numbers) // [2, 3]
//sampleEnd
}
```

遍历 Map

在 Java 中, 你可以通过 `forEach` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html#forEach(java.util.function.BiConsumer)) 来遍历 Map:

```
// Java
numbers.forEach((k,v) -> System.out.println("Key = " + k + ", Value
= " + v));
```

在 Kotlin 中, 请使用 `for` 或 `forEach` 循环, 类似于 Java 的 `forEach`, 来遍历 Map:

```
// Kotlin
for ((k, v) in numbers) {
    println("Key = $k, Value = $v")
}
// 或
numbers.forEach { (k, v) -> println("Key = $k, Value = $v") }
```

从可能为空的集合得到第 1 个和最后 1 个元素

在 Java 中, 你可以检查集合大小, 并使用下标安全的得到第 1 个和最后 1 个 元素:

```
// Java
var list = new ArrayList<>();
//...
if (list.size() > 0) {
    System.out.println(list.get(0));
    System.out.println(list.get(list.size() - 1));
}
```

对 `Deque` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Deque.html) 和它的后代类, 你还可以使用 `getFirst()` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Deque.html#getFirst()) 和 `getLast()` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Deque.html#getLast()) 函数:

```
// Java
var deque = new ArrayDeque<>();
//...
if (deque.size() > 0) {
    System.out.println(deque.getFirst());
    System.out.println(deque.getLast());
}
```

在 Kotlin 中, 有专门的函数 `firstOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first-or-null.html) 和 `lastOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last-or-null.html). 使用 `Elvis 操作符` ("Elvis 操作符" in "Null 值安全性"), 你可以根据函数结果执行更多操作. 比如, `firstOrNull()`:

```
// Kotlin
val emails = listOf<String>() // 可能为空
```

```
val theOldestEmail = emails.firstOrNull() ?: ""
val theFreshestEmail = emails.lastOrNull() ?: ""
```

从 List 创建 Set

在 Java 中, 要从 List (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html) 创建 Set (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Set.html), 你可以使用 Set.copyOf (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Set.html#copyOf(java.util.Collection)) 函数:

```
// Java
public void listToSet() {
    var sourceList = List.of(1, 2, 3, 1);
    var copySet = Set.copyOf(sourceList);
    System.out.println(copySet);
}
```

在 Kotlin 中, 使用函数 toSet():

```
fun main() {
//sampleStart
    // Kotlin
    val sourceList = listOf(1, 2, 3, 1)
    val copySet = sourceList.toSet()
    println(copySet)
//sampleEnd
}
```

对元素分组

在 Java 中, 你可以使用 Collectors (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Collectors.html) 函数 groupingBy(), 对元素分组:

```
// Java
public void analyzeLogs() {
```

```
    var requests = List.of(
        new Request("https://kotlinlang.org/docs/home.html", 200),
        new Request("https://kotlinlang.org/docs/home.html", 400),
        new Request("https://kotlinlang.org/docs/comparison-to-
java.html", 200)
    );
    var urlsAndRequests = requests.stream().collect(
            Collectors.groupingBy(Request::getUrl));
    System.out.println(urlsAndRequests);
}
```

在 Kotlin 中, 使用函数 groupBy()
(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/group-by.html):

```
class Request(
    val url: String,
    val responseCode: Int
)

fun main() {
//sampleStart
    // Kotlin
    val requests = listOf(
        Request("https://kotlinlang.org/docs/home.html", 200),
        Request("https://kotlinlang.org/docs/home.html", 400),
        Request("https://kotlinlang.org/docs/comparison-to-
java.html", 200)
    )
    println(requests.groupBy(Request::url))
//sampleEnd
}
```

过滤元素

在 Java 中, 要过滤集合的元素, 你需要使用 Stream API
(https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html). Stream API 包括 中间(Intermediate) 和 终止(Terminal) 操作. filter() 是一个中

间操作, 返回一个 Stream. 要得到输出的集合, 你需要使用终止操作, 比如 `collect()`. 比如, 要只保留 key 以 `1` 结尾并且 value 大于 `10` 的对:

```
// Java
public void filterEndsWith() {
    var numbers = Map.of("key1", 1, "key2", 2, "key3", 3, "key11",
11);
    var filteredNumbers = numbers.entrySet().stream()
        .filter(entry -> entry.getKey().endsWith("1") &&
entry.getValue() > 10)
        .collect(Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue));
    System.out.println(filteredNumbers);
}
```

在 Kotlin 中, 过滤是集合内建的操作, `filter()` 返回与过滤之前相同的集合类型. 因此, 你需要编写的代码只是 `filter()` 以及它的过滤条件:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key11" to 11)
    val filteredNumbers = numbers.filter { (key, value) ->
key.endsWith("1") && value > 10 }
    println(filteredNumbers)
//sampleEnd
}
```

详情请参见 过滤 Map ("过滤(Filtering)" in "Map 相关操作").

根据类型过滤元素

在 Java 中, 要根据类型过滤元素, 并对其执行操作, 你需要使用 `instanceof` (https://docs.oracle.com/en/java/javase/17/language/pattern-matching-instanceof-operator.html) 操作符检查元素类型, 然后进行类型转换:

```
// Java
public void objectIsInstance() {
```

```
    var numbers = new ArrayList<>();
    numbers.add(null);
    numbers.add(1);
    numbers.add("two");
    numbers.add(3.0);
    numbers.add("four");
    System.out.println("All String elements in upper case:");
    numbers.stream().filter(it -> it instanceof String)
        .forEach( it -> System.out.println(((String)
it).toUpperCase()));
}
```

在 Kotlin 中, 你可以直接对集合调用 `filterIsInstance<NEEDED_TYPE>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-is-instance.html), 类型转换会由 智能类型转换 ("智能类型转换" in "类型检查与类型转换") 完成:

```
// Kotlin
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf(null, 1, "two", 3.0, "four")
    println("All String elements in upper case:")
    numbers.filterIsInstance<String>().forEach {
        println(it.uppercase())
    }
//sampleEnd
}
```

验证判定条件

一些任务要求你检查是否所有元素, 不存在元素, 或存在某些元素符合某个条件. 在 Java 中, 你可以通过 Stream API (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html) 函数 `allMatch()` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#allMatch(java.util.function.Predicate)), `noneMatch()` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#noneMatch(java.util.function.Predicate)), 和 `anyMatch()`

(https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#anyMatch(java.util.function.Predicate)) 执行所有这些检查:

```
// Java
public void testPredicates() {
    var numbers = List.of("one", "two", "three", "four");
    System.out.println(numbers.stream().noneMatch(it ->
it.endsWith("e"))); // false
    System.out.println(numbers.stream().anyMatch(it ->
it.endsWith("e"))); // true
    System.out.println(numbers.stream().allMatch(it ->
it.endsWith("e"))); // false
}
```

在 Kotlin 中, 对所有的 Iterable (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterable/#kotlin.collections.Iterable) 对象, 可以使用 扩展函数 (扩展) `none()`, `any()`, and `all()`:

```
fun main() {
//sampleStart
// Kotlin
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.none { it.endsWith("e") })
    println(numbers.any { it.endsWith("e") })
    println(numbers.all { it.endsWith("e") })
//sampleEnd
}
```

详情请参见 验证判定条件 ("验证判定条件" in "过滤(Filtering)集合").

集合变换操作

合并(Zip)元素

在 Java 中, 你可以同时遍历两个集合, 将同一位置的两个元素变换为 pair :

```
// Java
public void zip() {
    var colors = List.of("red", "brown");
```

```
    var animals = List.of("fox", "bear", "wolf");

    for (int i = 0; i < Math.min(colors.size(), animals.size());
i++) {
        String animal = animals.get(i);
        System.out.println("The " + animal.substring(0,
1).toUpperCase()
                + animal.substring(1) + " is " + colors.get(i));
   }
}
```

如果你希望做某些更加复杂的操作, 而不仅仅是将元素 pair 打印输出, 你可以使用 Record (https://blogs.oracle.com/javamagazine/post/records-come-to-java). 在上面的示例中, Record 是 `record AnimalDescription(String animal, String color) {}`.

在 Kotlin 中, 使用 `zip()` ("合并(Zipping)" in "集合变换操作") 函数可以完成相同的功能:

```
fun main() {
//sampleStart
    // Kotlin
    val colors = listOf("red", "brown")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal ->
        "The ${animal.replaceFirstChar { it.uppercase() }} is
$color" })
//sampleEnd
}
```

`zip()` 返回 Pair (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-pair/) 对象组成的 List.

> 如果集合大小不同, `zip()` 的结果将是较小的那个集合大小. 结果中不包括较大那个集合的后面部分元素.

关联(Associate)元素

在 Java 中, 你可以使用 Stream API (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html) 将元素与某个特性关联在一起:

```
// Java
public void associate() {
    var numbers = List.of("one", "two", "three", "four");
    var wordAndLength = numbers.stream()
        .collect(toMap(number -> number, String::length));
    System.out.println(wordAndLength);
}
```

在 Kotlin 中, 使用 associate() ("关联(Association)" in "集合变换操作") 函数:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

下一步做什么?

- 访问 Kotlin Koans (Kotlin Koan) – 完成练习, 学习 Kotlin 语法. 每个练习从一个失败的 unit test 开始, 你的任务是让测试通过.
- 阅读其他的 Kotlin 惯用法 (惯用法).
- 学习如何使用 Java 到 Kotlin 转换器 ("使用 J2K 将既有的 Java 文件转换为 Kotlin" in "教程 - 在同一个项目中混合使用 Java 和 Kotlin"), 将既有的 Java 代码转换为 Kotlin .
- 学习 Kotlin 中的集合 (集合(Collection)概述).

如果你有喜欢的惯用法, 欢迎你发送一个 pull request, 分享给我们.

Java 和 Kotlin 中的可空性(Nullability)

最终更新: 2024/09/10

可空性(Nullability) 是指一个变量能否为 `null` 值的能力. 当变量值为 `null`, 使用这个变量指向的对象将会导致 `NullPointerException` 异常. 有很多种方法来编写代码, 来尽量减少发生指针异常的可能.

这篇向导会介绍在 Java 和在 Kotlin 中, 处理可为 null 值的变量的方案之间的区别. 这可以帮助你从 Java 迁移到 Kotlin, 并按照纯正的 Kotlin 风格来编写你的代码.

这篇向导的第一部分介绍最重要的差别 – Kotlin 对可为 null 类型的支持, 以及 Kotlin 如何处理 来自 Java 代码的类型. 第二部分, 从 检查函数调用的结果 开始, 通过几种具体的情况, 解释二者的差异.

参见 Kotlin 中的 null 值安全性 (Null 值安全性).

对可为 null 类型的支持

Kotlin 和 Java 的类型系统最重要的区别是, Kotlin 明确支持 可为 null 的类型 (Null 值安全性). 通过这种方式, 指定哪个变量可能包含 `null` 值. 如果一个变量可以为 `null`, 那么对这个变量调用方法是不安全的, 因为可能导致 `NullPointerException` 异常. Kotlin 在编译期禁止这样的调用, 因此防止很多潜在的异常. 在运行期, 可为 null 类型的对象与不可为 null 类型的对象的处理方式是一样的: 可为 null 的类型 并不是不可为 null 类型的一个包装. 所有的检查都在编译期进行. 这就意味着, 在 Kotlin 中使用可为 null 类型, 几乎不存在运行期负担.

> 我们说 "几乎", 是因为, 尽管 *确实* 生成了 内在的 (https://en.wikipedia.org/wiki/Intrinsic_function) 检查代码, 但它们在运行期的负担是非常非常小的.

在 Java 中, 如果你不编写 null 检查代码, 那么方法可能会抛出 `NullPointerException` 异常:

```
// Java
int stringLength(String a) {
    return a.length();
}

void main() {
```

```
    stringLength(null); // 这里会抛出 `NullPointerException` 异常
}
```

这个调用会产生下面的输出:

```
java.lang.NullPointerException: Cannot invoke "String.length()"
because "a" is null
    at test.java.Nullability.stringLength(Nullability.java:8)
    at test.java.Nullability.main(Nullability.java:12)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
```

在 Kotlin 中, 所有的通常类型默认都是不可为 null 的, 除非你将它们明确的标记为可为 null. 如果你不期望 `a` 可以为 `null`, 可以将 `stringLength()` 函数声明如下:

```
// Kotlin
fun stringLength(a: String) = a.length
```

参数 `a` 类型为 `String` 类型, 在 Kotlin 中代表它永远包含一个 `String` 实例, 而且不能为 `null`. Kotlin 中可为 null 的类型使用问号 `?` 来标记, 例如, `String?`. 如果 `a` 是 `String` 类型, 那么运行期出现 `NullPointerException` 是不可能的, 因为编译器会强制要求所有传递给 `stringLength()` 的参数不能为 `null`.

试图向 `stringLength(a: String)` 函数传递 `null` 值参数, 会导致编译期错误, "Null can not be a value of a non-null type String":

```
fun main() {
    stringLength( a: null)
}
}
```

Null can not be a value of a non-null type String
Change parameter 'a' type of function 'stringLength' to 'String?' ⌥⇧↩ More actions... ⌥↩

```
public final fun stringLength(
    a: String
): Int
```

java-to-kotlin-idioms-guide

向不可为 null 的函数传递 null 值参数时的错误

如果你想要向这个函数传递任意值的参数, 包括 `null` 值, 请在参数类型之后添加一个问号 `String?`, 并在函数体内部进行检查, 以确保参数值不是 `null`:

```
// Kotlin
fun stringLength(a: String?): Int = if (a != null) a.length else 0
```

在检查通过之后, 在编译器执行检查的范围内, 编译器会将这个变量当作是不可为 null 的类型 `String`.

你不进行这样的检查, 代码会编译失败, 错误信息如下: "Only safe (?.) ("安全调用" in "Null 值安全性") or non-nullable asserted (!!.) calls ("!! 操作符" in "Null 值安全性") are allowed on a nullable receiver ("可为空的接收者(Nullable Receiver)" in "扩展") of type String?".

这段代码还可以写得更简短一些 – 使用 安全调用操作符 ?. (If-not-null 的简写表达) ("If not null 的简写表达方式" in "惯用法"), 可以将 null 检查和方法调用结合为一个操作符:

```
// Kotlin
fun stringLength(a: String?): Int = a?.length ?: 0
```

平台类型(Platform types)

在 Java 中, 你可以使用注解来表示一个变量是否可以为 `null`. 这些注解不是标准库的一部分, 但你可以分别添加这些注解. 例如, 你可以使用 JetBrains 注解 `@Nullable` 和 `@NotNull` (来自 `org.jetbrains.annotations` 包), 或 Eclipse 的注解(`org.eclipse.jdt.annotation` 包). 当你 从 Kotlin 代码调用 Java 代码 ("可否为 null(Nullability) 注解" in "在 Kotlin 中调用 Java 代码") 时, Kotlin 能够识别这些注解, 并根据注解来处理这些类型.

如果你的 Java 代码没有这样的注解, Kotlin 会将 Java 类型当作 *平台类型(Platform types)*. 但由于 Kotlin 没有这些类型的可空性信息, 它的编译器会允许对这些类型进行操作. 需要由你来决定是否执行 null 检查, 因为:

- 和 Java 中一样, 如果你试图在 `null` 值上执行操作, 那么会发生 `NullPointerException` 异常.
- 编译器不会对多余的 null 检查进行高亮度显示, 如果你对一个不可为 null 类型的值,执行 null 值安全的操作, 通常会高亮度显示.

更多详情请参见 从 Kotlin 调用 Java 代码时, 如何处理 null 值安全性与平台类型 ("Null 值安全性与平台数据类型" in "在 Kotlin 中调用 Java 代码").

对确定不为 null (definitely non-nullable) 类型的支持

在 Kotlin 中, 如果要覆盖一个包含 `@NotNull` 参数的 Java 方法, 你需要 Kotlin 的确定不为 null

(definitely non-nullable) 类型.

例如, 对于 Java 中的这个 `load()` 方法:

```
import org.jetbrains.annotations.*;

public interface Game<T> {
  public T save(T x) {}
  @NotNull
  public T load(@NotNull T x) {}
}
```

要在 Kotlin 中成功的覆盖 `load()` 方法, 你需要将 `T1` 声明为确定不为 null (`T1 & Any`):

```
interface ArcadeGame<T1> : Game<T1> {
  override fun save(x: T1): T1
  // T1 声明为确定不为 null
  override fun load(x: T1 & Any): T1 & Any
}
```

关于泛型中的确定不为 null 类型, 详情请参见 确定不为 null 类型 ("确定不为 null 的类型" in "泛型 (Generic): in, out, where").

检查函数调用的结果

需要进行 `null` 值检查的一种常见的情况是, 通过函数调用得到结果.

在下面的示例中, 有 2 个 类, `Order` 和 `Customer`. `Order` 包含一个指向 `Customer` 实例的引用. `findOrder()` 函数返回 `Order` 类的一个实例, 如果它无法找到订单, 则返回 `null` 值. 目标要是处理得到的订单的客户实例.

下面是 Java 中的类:

```
//Java
record Order (Customer customer) {}

record Customer (String name) {}
```

在 Java 中, 调用函数, 并对结果进行 if-not-null 检查, 然后取得需要的属性值:

```
// Java
Order order = findOrder();

if (order != null) {
    processCustomer(order.getCustomer());
}
```

如果将上面的 Java 代码直接转换为 Kotlin 代码, 结果是:

```
// Kotlin
data class Order(val customer: Customer)

data class Customer(val name: String)

val order = findOrder()

// 直接转换
if (order != null){
    processCustomer(order.customer)
}
```

这里可以使用 安全调用操作符 `?.` (If-not-null 的简写表达) ("If not null 的简写表达方式" in "惯用法"), 结合标准库中的任何 作用域函数 (作用域函数(Scope Function)). 通常可以使用 `let` 函数:

```
// Kotlin
val order = findOrder()

order?.let {
    processCustomer(it.customer)
}
```

下面是更加简短的版本:

```
// Kotlin
findOrder()?.customer?.let(::processCustomer)
```

使用默认值代替 null

`null` 值检查通常用于对值为 `null` 的情况 设置默认值 ("默认参数" in "函数").

带有 null 检查的 Java 代码:

```
// Java
Order order = findOrder();
if (order == null) {
    order = new Order(new Customer("Antonio"))
}
```

要在 Kotlin 中表达同样的功能, 请使用 Elvis 操作符 (If-not-null-else 的简写表达) ("Elvis 操作符" in "Null 值安全性"):

```
// Kotlin
val order = findOrder() ?: Order(Customer("Antonio"))
```

返回一个值或返回 null 的函数

在 Java 中, 操作列表元素时你需要很小心. 你必须检查某个下标位置对应的元素是否存在, 然后才能使用这个元素:

```
// Java
var numbers = new ArrayList<Integer>();
numbers.add(1);
numbers.add(2);

System.out.println(numbers.get(0));
//numbers.get(5) // 这里会发生异常!
```

Kotlin 标准库通常会提供一些函数, 其名称表示它们可能会返回 `null` 值. 在集合 API 中, 这样的函数尤其普遍:

```
fun main() {
//sampleStart
    // Kotlin
    // 和 Java 中一样的代码:
    val numbers = listOf(1, 2)

    println(numbers[0])  // 如果集合为空, 会抛出
```

```
IndexOutOfBoundsException 异常
    //numbers.get(5)     // 这里会发生异常!

    // 更加智能的代码:
    println(numbers.firstOrNull())
    println(numbers.getOrNull(5)) // 这里会返回 null 值
//sampleEnd
}
```

聚合(Aggregate)操作

你需要得到最大元素, 或者如果不存在元素则得到 `null` 值, 在 Java 中你可以使用 Stream API (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html):

```
// Java
var numbers = new ArrayList<Integer>();
var max =
numbers.stream().max(Comparator.naturalOrder()).orElse(null);
System.out.println("Max: " + max);
```

在 Kotlin 中, 可以使用 聚合操作 (聚合(Aggregate)操作):

```
// Kotlin
val numbers = listOf<Int>()
println("Max: ${numbers.maxOrNull()}")
```

更多详情请参见 Java 和 Kotlin 中的集合 (Java 和 Kotlin 中的集合(Collection)).

安全的类型转换

如果你需要安全的转换一个类型, 在 Java 中你会使用 `instanceof` 操作符, 然后检查它是否成功:

```
// Java
int getStringLength(Object y) {
    return y instanceof String x ? x.length() : -1;
}

void main() {
```

```
    System.out.println(getStringLength(1)); // 输出结果为 `-1`
}
```

在 Kotlin 中为了避免异常, 可以使用 安全的转换操作符 (""安全的" (nullable) 类型转换操作" in "类型检查与类型转换") as?, 它会在转换失败时返回 null:

```
// Kotlin
fun main() {
    println(getStringLength(1)) // 输出结果为 `-1`
}

fun getStringLength(y: Any): Int {
    val x: String? = y as? String // 结果为 null
    return x?.length ?: -1 // 返回 -1, 因为 `x` 为 null
}
```

> 在上面的 Java 示例中, getStringLength() 函数返回的结果是基本类型 int. 如果要让它返回 null, 你可以使用 *装箱(boxed)* 类型 (https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html) Integer. 但是, 让这样的函数返回一个负值, 然后检查结果值, 在资源方面效率更高 – 如论如何你都需要进行检查, 但这种方式不会发生额外的装箱操作.

下一步做什么?

- 阅读其他的 Kotlin 惯用法 (惯用法).
- 学习如何使用 Java-to-Kotlin (J2K) 转换器 ("使用 J2K 将既有的 Java 文件转换为 Kotlin" in "教程 - 在同一个项目中混合使用 Java 和 Kotlin"), 将既有的 Java 代码转换为 Kotlin.
- 阅读其他的迁移向导:
  - Java 和 Kotlin 中的字符串 (Java 和 Kotlin 中的字符串)
  - Java 和 Kotlin 中的集合(Collection) (Java 和 Kotlin 中的集合(Collection))

如果你有喜欢的惯用法, 欢迎你发送一个 pull request, 分享给我们!

入门

最终更新: 2024/09/10

一个好的库应该具备以下特征:

- 向后兼容性(Backward Compatibility)
- 完整而且易于理解的文档
- 最低的 认知复杂度(cognitive complexity)
- API 保持一贯

这篇向导概要性的介绍为你的库编写 API 时的最佳实践, 以及需要考虑的问题. 包含以下几章:

- 可读性 (可读性)
- 可预测性 (可预测性)
- 可调试性 (可调试性)
- 向后兼容性(Backward Compatibility) (向后兼容性(Backward Compatibility))

下面的许多最佳实践提供了一些建议, 告诉你如何如何降低 API 的认知复杂度. 因此, 在进入到最佳实践之前, 我们先来解释一下认知复杂度.

认知复杂度(Cognitive complexity)

认知复杂度是指, 一个人为了理解一段代码所耗费的脑力劳动的多少. 认知复杂度高的代码库, 会更难理解更难维护, 因此导致发生 bug, 以及开发进度延迟.

一个没有遵循 单一功能原则 (https://en.wikipedia.org/wiki/Single-responsibility_principle) 的类或模块, 就是认知复杂度高的一个例子. 一个类或模块如果做了太多太多的事情, 就会难于理解和修改. 相反, 一个类或模块如果只包含一个清楚的, 明确定义的功能, 它就会更易于使用和维护.

函数也可能会有很高的认知复杂度. "写得不好" 的函数, 常见的特征是:

- 太多参数, 变量, 或循环.
- 太多嵌套的 if-else 语句构成的复杂的逻辑.

与逻辑清楚而且简单的函数(参数很少, 控制流易于理解) 相比, 这种认知复杂度高的函数更难于理解和维护. 认知复杂度高的函数的一个示例如下:

```
fun processData(
    data: List<String>,
    delimiter: String,
    ignoreCase: Boolean,
    sort: Boolean,
    maxLength: Int
) {
    // 一些复杂的处理逻辑
}
```

分解这些功能, 降低认知复杂度:

```
fun delimit(data: List<String>, delimiter: String) { ... }
fun ignoreCase(data: List<String>) { ... }
fun sortAscending(data: List<String>) { ... }
fun sortDescending(data: List<String>) { ... }
fun maxLength(data: List<String>, maxLength: Int) { ... }
```

通过 扩展函数 (扩展) 的帮助, 你可以更加简化上面的代码:

```
fun List<String>.delimit(delimiter: String): List<String> { ... }
fun List<String>.sortAscending(): List<String> { ... }
fun List<String>.sortDescending(): List<String> { ... }
fun List<String>.maxLength(maxLength: Int): List<String> { ... }
...
```

下一步做什么?

学习 API 的 可读性 (可读性).

可读性

最终更新: 2024/09/10

本章介绍关于 API 一致性 需要注意的问题, 并提供以下建议:

- 使用构建器 DSL
- 在适当的情况下, 使用类似构造器风格的函数
- 适当的使用成员函数和扩展函数
- 避免在函数中使用 Boolean 参数

API 一致性

API 保持一致, 并提供良好的文档, 对于良好的开发体验来说是非常重要的. 参数顺序, 整体的命名风格, 超载(overload) 也非常重要. 而且, 对于所有的惯例规约, 也应该编写文档.

例如, 如果你的一个方法接受 `offset` 和 `length` 参数, 那么其它方法也应该使用相同的参数, 而不是, 比如, 接受 `startIndex` 和 `endIndex` 参数. 这样的参数很可能是 `Int` 或 `Long` 类型, 因此很容易搞混它们.

对于参数顺序也是如此: 在各个方法之间, 以及超载的方法直接, 应该保持参数顺序一致. 否则, 库的使用者在传递参数时可能猜错参数的顺序.

下面是一个例子, 它保持了一致的参数顺序和名称:

```
fun String.chop(length: Int): String = substring(0, length)
fun String.chop(length: Int, startIndex: Int) =
    substring(startIndex, length + startIndex)
```

如果你有很多类似的方法, 应该对它们使用一致并且易于预见的名称. `stdlib` API 是这样做的: 有 `first()` 和 `firstOrNull()` 方法, `single()` 和 `singleOrNull()` 方法, 等等. 从它们的名称可以看出这些方法是成对的, 而且有些方法可能返回 `null`, 其它方法可能抛出异常.

使用构建器 DSL

在程序开发中, "构建器(Builder)"
(https://en.wikipedia.org/wiki/Builder_pattern#:~:text=The%20builder%20pattern%20is%20

a,Gang%20of%20Four%20design%20patterns) 是一个很著名的模式. 你可以用它来构建复杂的实体对象, 不是使用单个表达式一次性构建, 而是逐步的获得更多信息来构建. 当你需要使用构建器时, 最好使用构建器 DSL 语法, 它在二进制上是兼容的, 而且更符合语言习惯.

Kotlin 构建器 DSL 的典型例子是 `kotlinx.html`. 请看下面的示例:

```
header("modal-card-head") {
    p("modal-card-title") {
        +book.book.name
    }
    button(classes = "delete") {
        attributes["aria-label"] = "close"
        attributes["_"] = closeModalScript
    }
}
```

也可以通过传统的构建器的方式来实现, 但代码会明显的更冗长:

```
headerBuilder()
    .addClasses("modal-card-head")
    .addElement(
        pBuilder()
            .addClasses("modal-card-title")
            .addContent(book.book.name)
            .build()
    )
    .addElement(
        buttonBuilder()
            .addClasses("delete")
            .addAttribute("aria-label", "close")
            .addAttribute("_", closeModalScript)
            .build()
    )
    .build()
```

这样的实现存在太多你并不需要知道的细节, 而且它要求你构建每一个实体.

如果你需要在一个循环中动态的生成构建器的内容, 情况就变得更糟了. 在这样的情况下, 你必须创建变量实例, 并动态的覆盖它:

```
var buttonBuilder = buttonBuilder()
    .addClasses("delete")
for ((attributeName, attributeValue) in attributes) {
    buttonBuilder = buttonBuilder.addAttribute(attributeName,
attributeValue)
}
buttonBuilder.build()
```

在构建器 DSL 中, 你可以直接使用循环, 以及所有需要的 DSL 调用:

```
div("tags") {
    for (genre in book.genres) {
        span("tag is-rounded is-normal is-info is-light") {
            +genre
        }
    }
}
```

请记住, 在大括号内, 无法在编译期检查你是否设置了所有必须的属性. 为了避免这个问题, 请将必须的属性作为函数的参数, 而不是构建器的属性. 例如, 如果你希望 `href` 是一个必须的 HTML 属性, 你的函数应该是这样的:

```
fun a(href: String, block: A.() -> Unit): A
```

而不仅仅是:

```
fun a(block: A.() -> Unit): A
```

> 只要你不从构建器 DSL 中删除什么东西, 那么它就是 向后兼容的 (向后兼容性 (Backward Compatibility)). 通常情况下不会发生问题, 因为随着时间的推移, 大多数开发者只会向他们的构建器类添加更多的属性.

在适当的情况下, 使用类似构造器风格的函数

有时候, 你可以通过使用类似构造器风格的函数, 简化你的 API 的外观. 一个类似构造器风格的函数, 是指函数名称以大写字母开头, 因此看起来象一个类的构造器. 这种方式可以让你的库更易于理解.

假设你想要在你的库中引入一个 可选类型(Option Type)
(https://en.wikipedia.org/wiki/Option_type):

```
sealed interface Option<T>
class Some<T : Any>(val t: T) : Option<T>
object None : Option<Nothing>
```

你可以为所有的 `Option` 接口方法定义实现 – `map()`, `flatMap()`, 等等. 但是, 每次你的 API 使用者创建一个这样的 `Option` 时, 他们都必须写一些额外的逻辑, 来检查应该创建什么. 例如:

```
fun findById(id: Int): Option<Person> {
    val person = db.personById(id)
    return if (person == null) None else Some(person)
}
```

为了让你的用户不必每次都编写这些相同的检查代码, 你只需要在你的 API 中添加 1 行:

```
fun <T> Option(t: T?): Option<out T & Any> =
    if (t == null) None else Some(t)

// 上面代码的使用方式:
fun findById(id: Int): Option<Person> = Option(db.personById(id))
```

现在, 创建一个正确的 `Option` 变得非常简单: 只需要调用 `Option(x)`, 然后你就有了 null 值安全的, 功能正确的 Option 语法.

类似构造器风格的函数的另一种使用场景是, 当你需要返回某种 "隐藏的" 信息的时候, 例如 private 实例, 或 internal 对象. 作为例子, 我们来看看标准库中的一个方法:

```
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.isNotEmpty()) elements.asList() else emptyList()
```

在上面的例子中, `emptyList()` 返回下面的内容:

```
internal object EmptyList : List<Nothing>, Serializable,
RandomAccess
```

你可以编写一个类似构造器风格的函数, 降低你的代码的 认知复杂度 ("认知复杂度(Cognitive

complexity)" in "入门"), 并减少你的 API 的大小:

```
fun <T> List(): List<T> = EmptyList

// 上面代码的使用方式:
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.isNotEmpty()) elements.asList() else List()
```

适当的使用成员函数和扩展函数

只有 API 的非常核心的部分才应该写成 成员函数 ("成员函数" in "函数"), 其他所有功能应该写成 扩展函数 ("扩展函数(Extension Function)" in "扩展"). 这样可以帮助你告诉阅读代码的人, 什么是核心功能, 什么不是.

例如, 看看下面的 Graph 类:

```
class Graph {
    private val _vertices: MutableSet<Int> = mutableSetOf()
    private val _edges: MutableMap<Int, MutableSet<Int>> =
mutableMapOf()

    fun addVertex(vertex: Int) {
        _vertices.add(vertex)
    }

    fun addEdge(vertex1: Int, vertex2: Int) {
        _vertices.add(vertex1)
        _vertices.add(vertex2)
        _edges.getOrPut(vertex1) { mutableSetOf() }.add(vertex2)
        _edges.getOrPut(vertex2) { mutableSetOf() }.add(vertex1)
    }

    val vertices: Set<Int> get() = _vertices
    val edges: Map<Int, Set<Int>> get() = _edges
}
```

这个类只包含最少量的内容: vertices 和 edges 的 private 变量, 用于添加 vertices 和 edges 的函数, 以及访问函数, 返回当前状态的不可变的表达.

你可以在类之外添加所有其他功能:

```
fun Graph.getNumberOfVertices(): Int = vertices.size
fun Graph.getNumberOfEdges(): Int = edges.size
fun Graph.getDegree(vertex: Int): Int = edges[vertex]?.size ?: 0
```

只有属性, 覆盖, 以及访问器才应该作为类的成员.

避免在函数中使用 Boolean 参数

理想情况下, 读者应该只靠阅读读代码就能够判断函数参数的目的. 然而, 如果使用 `Boolean` 参数, 这就不太可能了, 尤其是如果你没有使用 IDE (例如, 如果你在某个版本管理系统中审查代码). 使用 命名的参数 ("命名参数" in "函数") 有助于说明参数的目的, 但目前不可能强迫开发者在 IDE 中使用命名的参数. 另一个方案是, 创建一个函数, 让它执行 `Boolean` 参数对应的功能, 并给这个函数一个非常有描述性的名称.

例如, 在标准库中, 有两个 `map()` 函数:

```
fun map(transform: (T) -> R): List<R>

fun mapNotNull(transform: (T) -> R?): List<R>
```

我们可以添加一个 `map(filterNulls: Boolean)` 函数, 然后编写这样的代码:

```
listOf(1, null, 2).map(false) { it.toString() }
```

只看这段代码, 很难推测出 `false` 到底代表什么意思. 但是, 如果你使用 `mapNotNull()` 函数, 读者就能立即理解它的逻辑:

```
listOf(1, null, 2).mapNotNull { it.toString() }
```

下一步做什么?

学习 API 的:

- 可预测性 (可预测性)
- 可调试性 (可调试性)

- 向后兼容性(Backward Compatibility) (向后兼容性(Backward Compatibility))

可预测性

最终更新: 2024/09/10

本章包含以下建议:

- 使用封闭接口(Sealed Interface)
- 通过封闭类(Sealed Class)隐藏具体实现
- 对你的输入和状态进行验证
  - 使用 require() 函数验证输入
  - 使用 check() 函数验证状态
- 在 public 签名中不要使用数组
- 不要使用 varargs

使用封闭接口(Sealed Interface)

当你需要对具体的实现进行功能抽象时, 你的 API 中通常会需要接口. 如果你需要使用接口, 请考虑使用 封闭接口(Sealed Interface) (封闭类(Sealed Class)与封闭接口(Sealed Interface)). 如果你不希望你的 API 使用者扩展你的类层次结构, 这一点是非常重要的.

请记住, 如果向一个封闭接口添加一个新的实现类, 会立即导致用户的现有代码变得不正确.

例如, JSON 类型可能是 6 种类型: 对象, 数组, 数值, 字符串, Boolean, 以及 null. 创建通常的 `interface JsonElement` 可能会导致错误, 因为使用者可能不小心定义一个新的 `JsonElement` 的实现类, 然后就会破坏你的代码. 相反, 你可以让 `interface JsonElement` *封闭*, 并为每个 JSON 类型添加实现类:

```
sealed interface JsonElement

class JsonNumber(val value: Number) : JsonElement
class JsonObject(val values: Map<String, JsonElement>) : JsonElement
class JsonArray(val values: List<JsonElement>) : JsonElement
```

```kotlin
class JsonBoolean(val value: Boolean) : JsonElement
class JsonString(val value: String) : JsonElement
object JsonNull : JsonElement
```

这种方案可以帮助你避免错误, 既包括库本身的错误, 也包括使用者的错误.

使用封闭类型最大的好处是在 `when` 表达式中. 如果能够确定条件分支语句覆盖了所有的情况, 你就不必添加 `else` 分支了:

```kotlin
fun processJson(json: JsonElement) = when (json) {
    is JsonNumber -> { /* 作为数值进行处理 */ }
    is JsonObject -> { /* 作为对象进行处理 */ }
    is JsonArray -> { /* 作为数组进行处理 */ }
    is JsonBoolean -> { /* 作为 Boolean 值进行处理 */ }
    is JsonString -> { /* 作为字符串进行处理 */ }
    is JsonNull -> { /* 作为 null 进行处理 */ }
    // 不需要 `else` 分支, 因为已经覆盖了所有的情况
}
```

通过封闭类(Sealed Class)隐藏具体实现

如果你的 API 中存在封闭接口, 并不代表你也应该在 API 中暴露所有的实现类. 公开最少的内容通常更好一些. 如果你需要避免抽象泄漏(Leaky Abstraction), 或者想要避免 API 的使用者扩展你的接口, 可以考虑对你的具体实现也使用封闭类(Sealed Class)或封闭接口(Sealed Interface).

例如, 一个库与多种不同的数据库共通工作, 它可能包含一个数据库应答的接口, 如下:

```kotlin
sealed interface DBResponse {
    operator fun <T> get(columnName: String): Sequence<T>
}
```

向 API 使用者暴露这个接口的实现类, 例如 `SQLiteResponse` 或 `MongoResponse`, 是一种 **抽象泄漏(Leaky Abstraction)**, 会使得支持这个 API 变得更加复杂. 在这样的库中, 你可能只处理了你的 `DBResponse` 实现类. 对于一个能够接受 `DBResponse` 的库方法, 如果使用者传入一个他们的 `DBResponse` 实现类, 就可能造成错误. 使用封闭接口和封闭类可以避免这种错误.

对你的输入和状态进行验证

使用 require() 函数验证输入

使用者有可能会误用一个 API. 为了帮助你的使用者正确使用你的 API, 你应该尽可能早的使用 require() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/require.html) 函数验证输入.

例如, 这是一个简单的库函数, 将保存用户到某个外部 API:

```
fun saveUser(username: String, password: String) {
    api.saveUser(User(username, password))
}
```

你应该对函数的参数进行验证, 确保输入符合要求. 例如, 要检查 `username` 是唯一的, 而且不为空, 即使你已经在你的数据库中定义了约束, 也要进行验证:

```
fun saveUser(username: String, password: String) {
    require(username.isNotBlank()) { "Username should not be blank"
}
    require(api.usernameAvailable(username)) { "Username $username
is already taken" }
    require(password.isNotBlank()) { "Password should not be blank"
}
    require(password.length > 6) { "Password should contain at least
7 letters" }
    require(
        /* 某些复杂的检查 */
    ) { "..." }

    api.saveUser(User(username, password))
}
```

通过这样的方法, 你可以确保你的使用者在遇到错误时不需要深入的分析复杂的、牵涉到数据库的 Stack Trace 信息. 如果出现异常, 将会是一个 `IllegalArgumentException`, 带有能够理解的错误消息, 而不是一个抽象的数据库异常.

如果你实现了输入验证, 那么应该对这些检查规则编写文档.

使用 check() 函数验证状态

同样的建议也适用于检查内部状态. 最明显的例子是 `InputStream`, 因为你不能从已经关闭的输入流读取数据.

看看下面的 InputStream 类, 它带有 readByte() 方法, 使用如下:

```
class InputStream : Closeable {
    private var open = true
    fun readByte(): Byte { /* 读取并返回一个 byte */ }
    override fun close() {
        // 销毁底层资源
        open = false
    }
}

fun readTwoBytes(inputStream: InputStream): Pair<Byte, Byte> {
    val first = inputStream.use { it.readByte() }
    val second = inputStream.readByte()
    return Pair(first, second)
}
```

readTwoBytes() 方法必须抛出 IllegalStateException, 因为 use{} (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/use.html) 会关闭 Closeable 的输入流, 使用者不应该能够从已关闭的流中读取数据. 要实现这一点, 需要修改 readByte() 函数的代码:

```
fun readByte(): Byte {
    check(open) { "Can't read from the already closed stream" }
    // 读取并返回一个 byte
}
```

在上面的示例中, 使用了 check() 函数, 而不是 require(). 这些函数会抛出不同的异常: require() 抛出 IllegalArgumentException, 而 check() 抛出 IllegalStateException. 在调试代码时, 这个区别可能会变得非常重要.

在 public 签名中不要使用数组

数组永远是可修改的, 而 Kotlin 的基础是安全的 – 只读的, 或者说值不可变的 – 对象. 如果必须在你的 API 中使用数组, 在将它们传递给其他代码之前, 请先复制数组, 这样你就能够数组不会被修改. 另一个选择方案是, 根据你的意图, 使用只读的或可变的集合(Collection). 一般来说, 最好避免使用数组, 如果你必须要用, 需要特别小心.

例如, Kotlin 中的枚举类有 values() 函数, 返回一个数组, 其中包含所有的枚举值. 如果数组没有复制, 使用者就可以重写数组中的元素:

```
enum class Test { A, B }

fun main() { Test.values()[0] = Test.B }
```

如果你在枚举类中缓存了这些值, 运行上面的代码之后, 缓存就会被损坏. 如果没有缓存这些值, 那么每次调用 values() 函数都会产生额外的运行时开销.

由于这个原因, Kotlin 从 1.9 开始废弃了 values() 函数, 并 引入了 (https://youtrack.jetbrains.com/issue/KT-48872/Provide-modern-and-performant-replacement-for-Enum.values) entries() 函数, 它返回一个不可变的 Set.

不要使用 varargs

vararg – 不定数量参数 ("不定数量参数(varargs)" in "函数") – 底层以数组的方式工作, 但数组元素会单独传递给函数, 而不是传递整个数组. 这个操作的成本很高, 因为它会不断复制同一个数组.

请看下面的代码:

```
fun printElements(delimiter: String, vararg elements: String) {
    for (i in elements.indices) {
        print(elements[i])
        if (i < elements.lastIndex) print(delimiter)
    }
}

fun printWithSpace(vararg elements: String) {
    printElements(" ", *elements)
}

fun main() {
    printWithSpace("x", "y", "z")
}
```

printElements() 函数打印 vararg 参数 elements 中的所有字符串, 中间加上分隔符, 而 printWithSpace() 函数调用 printElements(), 将分隔符定义为空格. 代码看起来似乎没问题: 你只是将 elements 从 printWithSpace() 传递给 printElements() 而已. 如果没有展开(spread)操作符 *, 这段代码将无法编译, 但加上这个操作符, 在传递给 printElements() 函数之前, **数组实际上会被复制**. 函数之间的调用链条越长, 就会创建越多的复制, 造成的意外的内存开销也就越大.

下一步做什么?

学习 API 的:

- 可调试性 (可调试性)
- 向后兼容性(Backward Compatibility) (向后兼容性(Backward Compatibility))

可调试性

最终更新: 2024/09/10

本章介绍关于可调试性需要注意的问题.

永远要提供 toString() 方法

为了便于调试, 要为你引入的每个类添加 `toString()` 方法的实现, 即使是对内部类也是如此. 如果 `toString()` 是契约(Contract) 的一部分, 那么要提供明确的文档说明.

下面的代码是图形建模代码简化后的例子:

```
class Vector2D(val x: Int, val y: Int)

fun main() {
    val result = (1..20).map { Vector2D(it, it) }
    println(result)
}
```

这段代码的输出没什么用处:

```
[Vector2D@27bc2616, Vector2D@3941a79c, Vector2D@506e1b77,...]
```

Debug Tool 窗口中提供的信息也没什么用处:

```
> 0 = {Vector2D@836} Vector2D@b4c966a
> 1 = {Vector2D@837} Vector2D@2f4d3709
> 2 = {Vector2D@838} Vector2D@4e50df2e
> 3 = {Vector2D@839} Vector2D@1d81eb93
> 4 = {Vector2D@840} Vector2D@7291c18f
> 5 = {Vector2D@841} Vector2D@34a245ab
```

在 Debug Tool 窗口中 Vector 对象的输出

为了让日志和调试信息更加易于阅读, 请添加一个简单的 `toString()` 实现, 如下:

```
override fun toString(): String =
    "Vector2D(x=$x, y=$y)"
```

改善后的输出如下:

```
[Vector2D(x=1, y=1), Vector2D(x=2, y=2), Vector2D(x=3, y=3), ...
```

0 = {Vector2D@836} Vector2D(x=1, y=1)
1 = {Vector2D@837} Vector2D(x=2, y=2)
2 = {Vector2D@838} Vector2D(x=3, y=3)
3 = {Vector2D@839} Vector2D(x=4, y=4)
4 = {Vector2D@840} Vector2D(x=5, y=5)
5 = {Vector2D@841} Vector2D(x=6, y=6)

在 Debug Tool 窗口中 Vector 对象的改善后的输出

> 使用 数据类 (数据类(Data Class)) 看起来好像很不错, 因为它们自动带有 toString() 方法. 在本向导的 向后兼容性(Backward Compatibility) (向后兼容性(Backward Compatibility)) 章节中, 你会学习 为什么不应该这样做 ("不要在 API 中使用数据类" in "向后兼容性(Backward Compatibility)").

即使你认为这个类不会在任何地方打印输出, 也应该考虑实现 toString(), 因为它可能会以意想不到的方式提供帮助. 例如, 在 构建器 (https://en.wikipedia.org/wiki/Builder_pattern#:~:text=The%20builder%20pattern%20is%20a,Gang%20of%20Four%20design%20patterns) 之内, 能够看到构建器目前的状态可能会非常重要.

```
class Person(
    val name: String?,
    val age: Int?,
    val children: List<Person>
) {
    override fun toString(): String =
        "Person(name=$name, age=$age, children=$children)"
```

```
}

class PersonBuilder {
    var name: String? = null
    var age: Int? = null
    val children = arrayListOf<Person>()
    fun child(personBuilder: PersonBuilder.() -> Unit = {}) {
        children.add(person(personBuilder))
    }
}

fun person(personBuilder: PersonBuilder.() -> Unit = {}): Person {
    val builder = PersonBuilder()
    builder.personBuilder()
    return Person(builder.name, builder.age, builder.children)
}
```

上面的代码预期的使用方式是:

```
2    person { this: PersonBuilder
3        name = "Pasha"
4        age = 36
5        child { this: PersonBuilder
6            name = "Mark"
7        }
8    }
```

Person DSL 和断点的使用方式

如果你在第一个 child 的右大括号之后的行设置断点 (如上图所示), 你会在 Debug Output 中看到一个无意义的字符串:

```
> p this = {PersonBuilder@830} PersonBuilder@eed1f14
```

PersonBuilder 调试时的结果

如果你添加一个简单的 toString() 实现, 如下:

```
override fun toString(): String =
    "PersonBuilder(name=$name, age=$age, children=$children)"
```

调试信息会变得更加清晰:

```
> this = {PersonBuilder@830} PersonBuilder(name=Pasha, age=36, children=[Person(name=Mark, age=null, children=[])])
```

你还能立即看到哪些域变量已被设置, 哪些还没有设置.

> 在 `toString()` 中暴露域变量时要小心, 因为很容易导致 `StackOverflowException`. 例如, 如果 `children` 引用到了 parent, 可能会造成循环引用. 而且, 暴露 List 和 Map 时也要小心, 因为 `toString()` 可能会展开一个非常深层的嵌套结构.

下一步做什么?

学习 API 的 向后兼容性(Backward Compatibility) (向后兼容性(Backward Compatibility)).

向后兼容性(Backward Compatibility)

最终更新: 2024/09/10

本章介绍关于 向后兼容性(Backward Compatibility) 需要注意的问题. 下面是 "不要做" 的建议:

- 不要向既有的 API 函数添加参数
- 不要在 API 中使用数据类
- 不要降低返回值类型的范围

要考虑使用:

- @PublishedApi 注解
- @RequiresOptIn 注解
- 明确 API 模式(Explicit API Mode)

详情请参见 用于增强向后兼容性的工具.

向后兼容性(Backward Compatibility)的定义

一个好的 API, 非常重要的一点就是向后兼容性. 向后兼容的代码, 使得新 API 版本的客户能够使用他们过去在旧 API 版本中曾经使用过的相同的 API 代码. 本节介绍为了让你的 API 保持向后兼容性所应该考虑的要点.

在我们讨论 API 时, 至少有三种类型的兼容性:

- 源代码兼容(Source)
- 行为兼容(Behavioral)
- 二进制兼容(Binary)

关于兼容性类型的详细讨论

如果你能够确信你的客户的应用程序能够使用你的库的新版本正确的重新编译, 那么你可以认为库的版本之间是 **源代码兼容的(Source-compatible)**. 通常来说, 除非变更非常微小, 否则源代码兼容

的实现和自动检查都是非常困难的. 在任何 API 中, 总是会存在一些特殊情况, 某些修改会导致破坏源代码兼容性.

行为兼容性(Behavioral Compatibility) 保证任何新的代码都不会改变原来代码行为的语义, Bug 修正除外.

库的 **二进制向后兼容(Binary Backward-compatible)** 版本可以替换这个库以前编译的版本. 使用这个库的以前版本编译的任何软件, 都应该能够继续正确工作.

在不破坏源代码兼容性的情况下, 也有可能会破坏二进制兼容性, 反过来也是如此.

保持二进制兼容性的某些原则是非常显而易见: 不要直接删除 Public API 的某些部分; 相反, 应该废弃(deprecate) (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-deprecated/) 它们. 后面的各节介绍一些比较少为人了解的原则.

"不要做" 的建议

不要向既有的 API 函数添加参数

向一个 Public API 添加无默认值的参数, 是一种破坏性变更(Breaking Change), 因为既有的代码将没有足够的信息来调用变更后的方法. 即使添加 默认参数 ("默认参数" in "函数") 也有可能会破坏你的使用者的代码.

下面的例子演示向后兼容性如何被破坏, 其中包含两个类: `lib.kt` 表示一个 "库", `client.kt` 表示这个 "库" 的一个 "客户端". 在真正的应用程序中, 这样的 "库/客户端" 结构是很常见的. 在这个示例中, "库" 有一个函数, 计算 Fibonacci 数列的第 5 个元素. `lib.kt` 文件内容如下:

```
fun fib() = ... // 返回第 5 个元素
```

我们从另一个文件 `client.kt` 中调用这个函数:

```
fun main() {
    println(fib()) // 返回 3
}
```

我们来编译这些类:

```
kotlinc lib.kt client.kt
```

编译结果是 2 个文件: `LibKt.class` 和 `ClientKt.class`.

我们来调用客户端, 确认它能正常工作:

```
$ kotlin ClientKt.class
3
```

这段代码的设计远远不够完美, 而且出于学习的目的, 使用了硬编码. 它预先定义了你想要从数列中获取哪个元素, 这样做是不正确的, 而且违反了代码清晰的原则. 我们来重写这段代码, 保持相同的默认行为: 默认情况下它会返回第 5 个元素, 但也可以指定你想要取得的元素序号.

`lib.kt`:

```
fun fib(numberOfElement: Int = 5) = ... // 返回指定的元素
```

我们只编译 "库": `kotlinc lib.kt`.

我们来运行 "客户端":

```
$ kotlin ClientKt.class
```

结果是:

```
Exception in thread "main" java.lang.NoSuchMethodError: 'int
LibKt.fib()'
        at LibKt.main(fib.kt:2)
        at LibKt.main(fib.kt)
        …
```

发生了 `NoSuchMethodError` 错误, 因为编译之后 `fib()` 函数的签名发生了变化.

如果你重新编译 `client.kt`, 它又可以正常工作了, 因为它会注意到新的函数签名. 在这个示例中, **在保持源代码兼容性的同时, 破坏了二进制兼容性.**

使用反编译(decompilation)来理解具体细节

> 这段解释只适用于 JVM 平台.

让我们对修改前的 `LibKt` 类调用 `javap` (https://docs.oracle.com/en/java/javase/20/docs/specs/man/javap.html):

```
> javap LibKt
Compiled from "lib.kt"
public final class LibKt {
```

```
 public static final int fib();
}
```

对修改后的类也做同样的调用:

```
> javap LibKt
Compiled from "lib.kt"
public final class LibKt {
 public static final int fib(int);
 public static int fib$default(int, int, java.lang.Object);
}
```

签名为 `public static final int fib()` 的方法被替换为一个新的方法, 签名为 `public static final int fib(int)`. 同时, 一个代理方法 `fib$default` 将调用委托给 `fib(int)`. 对于 JVM 平台, 可以绕过这个问题: 你需要添加 `@JvmOverloads` ("重载函数(Overload)的生成" in "在 Java 中调用 Kotlin 代码") 注解. 对于跨平台项目, 没有变通方法.

不要在 API 中使用数据类

我们通常会使用 数据类(Data Class) (数据类(Data Class)), 因为它们代码短, 简洁, 而且自动提供了很多好的功能. 但是, 由于数据类工作方式的某些细节, 在库的 API 中最好不要使用它们. 几乎任何变更都会导致 API 不能向后兼容.

一般来说, 很难预测随着时间的推移你将会需要如何修改一个类. 即使今天你认为这个类是独立的, 但没有办法确信你的需求在未来不会变化. 因此, 只有在你决定修改一个这样的类的时候, 数据类的这些问题才会发生.

首先, 上一节中介绍过的需要注意的问题, 不要向既有的 API 函数添加参数, 同样适用于构造器, 因为它也是一个方法. 第二, 即使你添加了次级构造器(Secondary Constructor), 也不能解决兼容性问题. 我们来看看下面的数据类:

```
data class User(
    val name: String,
    val email: String
)
```

例如, 随着时间的推移, 你发现用户需要办理一个激活过程, 因此你想要添加一个新的域变量, "active", 默认值为 "true". 这个新的域变量应该能够让既有的代码不需要修改就能正常工作.

在 上一节 中我们已经讨论过, 你不能仅仅只是添加新的域变量, 如下:

```
data class User(
    val name: String,
    val email: String,
    val active: Boolean = true
)
```

因为这个变更是 **二进制不兼容的**.

我们来添加一个新的构造器, 它只接受 2 个参数, 并对第 3 个参数使用默认值来调用主构造器:

```
data class User(
    val name: String,
    val email: String,
    val active: Boolean = true
) {
    constructor(name: String, email: String) :
            this(name, email, active = true)
}
```

现在有了 2 个构造器, 而且其中一个的签名与修改之前的类的构造器一致:

```
public User(java.lang.String, java.lang.String);
```

但问题不在于构造器 – 而出在 `copy` 函数. 它的签名发生了变更, 之前是:

```
public final User copy(java.lang.String, java.lang.String);
```

现在是:

```
public final User copy(java.lang.String, java.lang.String, boolean);
```

这个变更导致代码 **二进制不兼容**.

当然, 可以数据类的内部添加一个属性, 但这样就失去了数据类的所有优点. 因此, 在你的 API 中最好不要使用数据类, 因为对数据类的几乎所有变更都会破坏源代码兼容性, 二进制兼容性, 或行为兼容性.

如果你出于某种原因必须使用数据类, 那么你需要覆盖构造器和 `copy()` 方法. 此外, 如果你向类的 Body 部添加一个域变量, 你需要覆盖 `hashCode()` 和 `equals()` 方法.

> ⚠ 交换参数的顺序永远是一种不兼容的变更, 因为 `componentX()` 方法发生了变化. 这样的变更会破坏源代码兼容性, 可能也会破坏二进制兼容性.

不要降低返回值类型的范围

有些情况下, 尤其是如果你没有使用 明确 API 模式(Explicit API Mode) ("供库作者使用的明确 API 模式" in "Kotlin 1.4.0 版中的新功能"), 返回值类型声明可能发生隐含的变化. 但即使在这样的情况之外, 你也可能会降低返回值类型的范围. 例如, 你可能发现需要使用下标索引来访问你的集合中的元素, 并且希望将返回值类型从 `Collection` 修改为 `List`. 放宽返回值类型的范围通常会破坏源代码兼容性; 例如, 将 `List` 转换为 `Collection`, 会破坏所有使用下标索引访问元素的代码. 降低返回值类型的范围通常是源代码兼容的变更, 但会破坏二进制兼容性, 本节会进行解释.

我们来看看 `library.kt` 文件中的一个库函数:

```
public fun x(): Number = 3
```

在 `client.kt` 文件中对这个函数的使用示例:

```
fun main() {
    println(x()) // 输出结果为 3
}
```

我们使用 `kotlinc library.kt client.kt` 编译它, 并确认它能够正确工作:

```
$ kotlin ClientKt
3
```

下面我们将 "库" 函数 `x()` 返回值类型从 `Number` 改为 `Int`:

```
fun x(): Int = 3
```

并且只重编译客户端: `kotlinc client.kt`. 现在 `ClientKt` 不能再象期望的那样工作了. 它不会输出 `3`, 而是抛出一个异常:

```
Exception in thread "main" java.lang.NoSuchMethodError:
'java.lang.Number Library.x()'
        at ClientKt.main(call.kt:2)
```

```
        at ClientKt.main(call.kt)
        ...
```

发生这个问题是因为字节码中下面的这行:

```
0: invokestatic  #12 // 方法 Library.x:()Ljava/lang/Number;
```

这行的意思是, 你调用返回类型为 Number 的静态方法 x(). 但这个方法已经不存在了, 因此 **二进制兼容性被破坏了**.

@PublishedApi 注解

有些时候, 你可能需要使用你的一部分内部 API, 来实现 内联函数(Inline Function) (内联函数(Inline Function)). 你可以通过 @PublishedApi (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-published-api) 注解达到这个目的. 你应该将标注了 @PublishedApi 的代码看作是 Public API 的一部分, 因此, 你应该小心注意向后兼容性问题.

@RequiresOptIn 注解

有些时候, 你可能想要让用户试用你的 API. 在 Kotlin 中, 有很好的方法将某些 API 定义为不稳定状态 – 使用 @RequiresOptIn 注解 ("对 API 标记要求使用者同意" in "明确要求使用者同意的功能(Opt-in Requirement)"). 但是, 要注意以下问题:

1. 如果你很长时间没有修改你的 API 的某个部分, 而且它已经处于稳定状态, 你应该重新考虑是否使用 @RequiresOptIn 注解.
2. 你可以使用 @RequiresOptIn 注解来对 API 的不同部分定义不同的保证级别: 预览版, 实验版, 内部版, Delicate, 或 Alpha, Beta, RC.
3. 你应该明确定义各个 级别 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-requires-opt-in/-level/) 代表什么含义, 编写 KDoc (为 Kotlin 代码编写文档: KDoc) 注释, 并添加警告信息.

如果你依赖于一个 API 明确要求使用者同意, 不要使用 @OptIn 注解. 要使用 @RequiresOptIn 注解, 这样可以让你的用户能够有意识的选择他们想要哪个 API, 不想要哪个 API.

@RequiresOptIn 的另一个例子是, 如果你想要在使用者使用某个 API 时明确的提出警告. 例如, 如果你在维护一个库, 它利用了 Kotlin 的反射功能, 你可以对这个库中的类添加

`@RequiresFullKotlinReflection` 注解.

明确 API 模式(Explicit API Mode)

你应该让你的 API 保持尽可能的清楚明白. 为了强制让 API 清楚明白, 请使用 明确 API 模式 (Explicit API Mode) (["供库作者使用的明确 API 模式" in "Kotlin 1.4.0 版中的新功能"]()).

Kotlin 给了你很大的自由度来决定如何编写代码. 可以省略类型定义, 可见度声明, 或文档. 明确 API 模式强制要求你(作为开发者), 将这些隐含的信息明确的定义清楚. 在上面的链接中, 你可以看到如何启用这个功能. 我们来理解一下你为什么需要这个功能:

1. 不使用明确 API 模式, 会很容易破坏向后兼容性:

   ```
   // 版本 1
   fun getToken() = 1

   // 版本 1.1
   fun getToken() = "1"
   ```

   `getToken()` 的返回类型发生了变化, 而你甚至不需要修改它的签名, 就破坏了使用者的代码. 他们期望的返回值是 `Int`, 但实际得到的是 `String`.

2. 对可见度也是如此. 如果 `getToken()` 函数是 `private`, 那么向后兼容性不会被破坏. 但如果没有明确的可见度声明, 就不清楚 API 的使用者是否应该能够访问它. 如果使用者应该可以访问, 那么应该声明为 `public`, 并添加文档; 这种情况下, 上面的变更会破坏向后兼容性. 如果使用者不应该可以访问, 那么应该声明为 `private` 或 `internal`, 上面的变更就不会造成破坏.

用于增强向后兼容性的工具

在软件开发中, 向后兼容性是一个至关重要的方面, 因为它能够确保库或框架的新版本能够与既有的代码一起使用, 而不引起任何问题. 维护向后兼容性可能成为一项困难而且耗费时间的任务, 尤其是在处理大型代码库, 或复杂 API 的时候. 向后兼容性很难手动维护, 而且开发者经常需要依赖于测试和手动检查来确保新的变更不会破坏既有的代码. 为了解决这个问题, JetBrains 创建了 二进制兼容性验证器, 此外还有另一个解决方案: japicmp.

> 目前, 这两个工具都只能用于 JVM 平台.

这两个解决方案都有它们的优点和缺点. japicmp 可以用于任何 JVM 语言, 而且它既是一个 CLI 工具, 也是一个构建系统 plugin. 但是, 它要求应用程序的旧版本和新版本都以 JAR 文件形式提供. 如果你不能得到你的库的旧版本的构建, 它就不那么容易使用了. 而且, japicmp 会给出 Kotlin metadata 的变更信息, 你可能并不需要 (因为 metadata 格式并没有明确的规格, 而且它只供 Kotlin 内部使用).

二进制兼容性验证器只能作为 Gradle plugin 使用, 而且它还处于 Alpha 阶段 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它不需要访问 JAR 文件. 它只需要以前的 API 和当前 API 的特定的 dump. 它能够自己收集这些 dump. 关于这些工具, 详情请阅读下文.

二进制兼容性验证器

二进制兼容性验证器 (https://github.com/Kotlin/binary-compatibility-validator) 是一个工具, 它自动检测并报告 API 中的破坏性变更, 帮助确保你的库和框架的向后兼容性. 这个工具分析你进行修改之前和之后的库的字节码, 并比较两个版本, 找出可能破坏既有代码的变更. 这使得你可以在问题暴露给使用者之前, 更加容易的检测并修复问题.

这个工具可以节约你在手动测试和检查上耗费的大量的时间和精力. 它还能帮助防止 API 中的破坏性变更可能造成的问题. 最终可以带来更好的用户体验, 因为用户能够依赖于库和框架的稳定性和兼容性.

japicmp

如果你的开发平台只有 JVM, 你也可以使用 japicmp (https://siom79.github.io/japicmp/). japicmp 工作的层级与二进制兼容性验证器不同: 它比较两个 jar 文件 – 旧版本和新版本 – 并报告它们之间的不兼容性.

要注意, japicmp 不仅仅报告不兼容性, 还包括不会对使用者造成任何影响的变更. 例如, 对于下面的代码:

```
class Calculator {
    fun add(a: Int, b: Int): Int = a + b
    fun multiply(a: Int, b: Int): Int = a * b
}
```

如果你添加一个新的方法, 并不破坏兼容性, 如下:

```
class Calculator {
    fun add(a: Int, b: Int): Int = a + b
    fun multiply(a: Int, b: Int): Int = a * b
```

```
    fun divide(a: Int, b: Int): Int = a / b
}
```

japicmp 会报告以下变更:

```
***  MODIFIED CLASS: PUBLIC FINAL Calculator  (not serializable)
        ===  CLASS FILE FORMAT VERSION: 52.0 <- 52.0
        ===  UNCHANGED SUPERCLASS: java.lang.Object (<- java.lang.Object)
        ===  UNCHANGED CONSTRUCTOR: PUBLIC Calculator()
        ===  UNCHANGED METHOD: PUBLIC FINAL int add(int, int)
        +++  NEW METHOD: PUBLIC(+) FINAL(+) int divide(int, int)
        ===  UNCHANGED METHOD: PUBLIC FINAL int multiply(int, int)
        ***  MODIFIED ANNOTATION: kotlin.Metadata
                ===  UNCHANGED ELEMENT: xi=48
                ===  UNCHANGED ELEMENT: mv=1,8,0
                ===  UNCHANGED ELEMENT: k=1
                ***  MODIFIED ELEMENT: d1=��&#xA;���&#xA;���&#xA;���&#xA;���&#xA;����� � � � ���� ����� � ����� � ����� �
J����� � ����� � ����� � ����� � ����� � ����� �(<-  ��&#xA;���&#xA;���&#xA;���&#xA;���&#xA;����� � � � ���� ����� � ����
� � ����� � ����� � ����� � ����� �
                ***  MODIFIED ELEMENT: d2=LCalculator;,,()V,add,,a,b,divide,multiply (<- LCalculator;,,()V,add,,a,b,multi
ply)
```

japicmp 兼容性检查的输出

这仅仅是在 `@Metadata` 注解中的变更, 并没有什么重要意义, 但 japicmp 并不理解 JVM 语言, 必须报告它发现的一切变更.

Kotlin/Native 开发入门 - 使用 IntelliJ IDEA

最终更新: 2024/09/10

本教程演示如何使用 IntelliJ IDEA 创建一个 Kotlin/Native 应用程序.

开始之前, 首先请安装 IntelliJ IDEA (https://www.jetbrains.com/idea/download/index.html) 的最新版. 本教程适用于 IntelliJ IDEA Community 版和 Ultimate 版.

开始前的准备工作

1. 下载并安装最新版本的 IntelliJ IDEA (https://www.jetbrains.com/idea/) 和 Kotlin plugin (Kotlin 的发布版本).
2. 在 IntelliJ IDEA 中选择菜单 **File** | **New** | **Project from Version Control**, 克隆 项目模板 (https://github.com/Kotlin/kmp-native-wizard).
3. 打开 `build.gradle.kts` 文件, 这是构建脚本, 其中包含项目设置. 要创建 Kotlin/Native 应用程序, 你需要安装 Kotlin Multiplatform Gradle plugin. 请确认使用了 plugin 的最新版本:

   ```
   plugins {
       kotlin("multiplatform") version "1.9.23"
   }
   ```

- 关于这些设置, 详情请参见 跨平台程序的 Gradle DSL 参考文档 (跨平台程序的 Gradle DSL 参考文档).
- 关于 Gradle 构建系统, 详情请参见 Gradle 文档 (Gradle).

构建并运行应用程序

点击屏幕顶部运行配置旁边的 **Run**, 启动应用程序:

IntelliJ IDEA 会打开 **Run** tab, 并显示输出:

你可以 配置 IntelliJ IDEA (https://www.jetbrains.com/help/idea/compiling-applications.html#auto-build), 让它自动构建你的项目:

1. 打开菜单 **Settings/Preferences | Build, Execution, Deployment | Compiler**.
2. 在 **Compiler** 页, 选择 **Build project automatically**.
3. 保存变更.

现在, 当你在类文件中进行变更, 或者保存文件 (**Ctrl + S/Cmd + S**)时, IntelliJ IDEA 会自动对项目执行增量构建(Incremental Build).

更新应用程序

计算你的名字中的字母数量

1. 打开 `src/nativeMain/kotlin` 中的 `Main.kt` 文件.

   `src` 目录包含 Kotlin 源代码文件和资源. `Main.kt` 文件包含示例代码, 它使用 `println()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/println.html) 函数 打印 "Hello, Kotlin/Native!".

2. 添加代码读取输入. 使用 `readln()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/readln.html) 函数 读取输入值, 并赋值给 `name` 变量:

```
fun main() {
    // 读取输入值.
    println("Hello, enter your name:")
    val name = readln()
}
```

3. 在 `build.gradle.kts` 文件中, 指定 `System.in` 作为运行项目时的输入:

```
kotlin {
    // ...
    nativeTarget.apply {
        binaries {
            executable {
                entryPoint = "main"
```

```
                runTask?.standardInput = System.`in`
            }
        }
    }
    // ...
}
```

4. 删除空白字符, 计算字母数量:

- 使用 replace() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/replace.html) 函数 删除名字中的空白字符 .
- 使用作用域函数(Scope Function) let ("let 函数" in "作用域函数(Scope Function)") 在对象上下文之内运行函数.
- 使用一个 字符串模板 ("字符串模板" in "字符串") 来向一个字符串插入你的名字长度, 方法是添加一个 $ 符号, 并将表达式放在大括号内 – ${it.length}. it 是 lambda 表达式参数 ("Lambda 表达式参数" in "编码规约") 的默认名称.

```
fun main() {
    // 读取输入值.
    println("Hello, enter your name:")
    val name = readln()
    // 计算名字中的字母数量.
    name.replace(" ", "").let {
        println("Your name contains ${it.length} letters")
    }
}
```

5. 保存变更, 运行应用程序.

6. 输入你的名字, 查看结果:

计算你的名字中的不重复的字母数量

1. 打开 src/nativeMain/kotlin 中的 Main.kt 文件.

2. 为 String 声明新的 扩展函数 ("扩展函数(Extension Function)" in "扩展") countDistinctCharacters():

- 使用 lowercase() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/lowercase.html) 函数, 将名字转换为小写.
- 使用 toList() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/to-list.html) 函数, 将输入的字符转换为一个字符列表.
- 使用 distinct() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/distinct.html) 函数, 选择你的名字中的不重复的字符.
- 使用 count() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/count.html) 函数, 计算不重复的字符.

```
fun String.countDistinctCharacters() =
lowercase().toList().distinct().count()
```

3. 使用 countDistinctCharacters() 函数, 计算你的名字中不重复字符的数量:

```
fun String.countDistinctCharacters() =
lowercase().toList().distinct().count()

fun main() {
    // 读取输入值.
    println("Hello, enter your name:")
    val name = readln()
    // 计算名字中的字母数量.
    name.replace(" ", "").let {
        println("Your name contains ${it.length} letters")
        // 打印不重复字符的数量.
        println("Your name contains
${it.countDistinctCharacters()} unique letters")
    }
}
```

4. 保存变更, 运行应用程序.

5. 输入你的名字, 查看结果:

下一步做什么?

创建过你的第一个应用程序之后, 你可以完成我们的 Kotlin/Native 长教程, 使用 C Interop 和 libcurl 创建应用程序 (教程 - 使用 C Interop 和 libcurl 创建应用程序), 这个教程将会演示如何创建一个 native HTTP 客户端, 以及如何与 C 代码库交互.

Kotlin/Native 开发入门 - 使用 Gradle

最终更新: 2024/09/10

Gradle (https://gradle.org) 是在 Java, Android, 和其他开发环境中广泛使用的一个构建系统. Kotlin/Native 和 Multiplatform 的构建默认使用 Gradle.

虽然大多数 IDE, 包括 IntelliJ IDEA (https://www.jetbrains.com/idea), 都可以生成需要的 Gradle 文件, 但本教程还是介绍如何手工创建, 以便更好的理解工作原理.

开始之前, 请安装 Gradle (https://gradle.org/install/) 的最新版本.

> 如果你更希望使用 IDE, 请阅读教程 使用 IntelliJ IDEA (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA).

创建项目文件

1. 创建一个项目目录. 在这个目录内, 创建 Gradle 构建文件 `build.gradle(.kts)`, 内容如下:

   Kotlin

```
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "1.9.23"
}

repositories {
    mavenCentral()
}

kotlin {
    macosX64("native") { // 用于 macOS 环境
    // linuxX64("native") // 用于 Linux 环境
    // mingwX64("native") // 用于 Windows 环境
        binaries {
            executable()
        }
```

```
    }
}

tasks.withType<Wrapper> {
    gradleVersion = "8.1.1"
    distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}

repositories {
    mavenCentral()
}

kotlin {
    macosX64('native') { // 用于 macOS 环境
    // linuxX64('native') // 用于 Linux 环境
    // mingwX64('native') // 用于 Windows 环境
        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = '8.1.1'
    distributionType = 'BIN'
}
```

你可以使用各种 预定义编译目标 (Kotlin/Native 支持的目标平台), 比如 `macosX64`, `mingwX64`, `linuxX64`, `iosX64`, 来定义对应的编译目标平台. 预定义的名称描述了编译你的代

码所针对的目标平台. 这些预定义编译目标接受一个可选的参数, 表示编译目标名称, 在这个例子中是 native. 编译目标名称用来在项目中生成源代码路径和 Gradle task 名称.

2. 在项目目录内创建一个空的 settings.gradle 或 settings.gradle.kts 文件.
3. 创建一个目录 src/nativeMain/kotlin, 在其中创建文件 hello.kt, 内容如下:

```
fun main() {
    println("Hello, Kotlin/Native!")
}
```

根据一般约定, 所有的源代码放在 src/<target name>[Main|Test]/kotlin 目录下, 其中 main 放置产品代码, test 放置测试代码. <target name> 对应于构建文件中指定的编译目标平台(在我们的示例中是 native).

现在你可以构建你的项目并运行应用程序了.

构建并运行应用程序

1. 在项目的根目录, 通过以下命令运行构建:

```
gradle nativeBinaries
```

这个命令会创建 build/bin/native 目录, 其中包含 2 个子目录: debugExecutable 和 releaseExecutable. 分别包含对应的二进制文件.

默认情况下, 二进制文件的名称与项目目录相同.

2. 要运行项目, 请执行以下命令:

```
build/bin/native/debugExecutable/<project_name>.kexe
```

终端会输出 "Hello, Kotlin/Native!".

在 IDE 中打开项目

现在你可以在支持 Gradle 的任何 IDE 中打开你的项目. 如果你使用 IntelliJ IDEA:

1. 选择 **File** | **Open...**.

2. 选择项目目录, 并点击 **Open**. IntelliJ IDEA 会自动检测到这是一个 Kotlin/Native 项目.

> 如果你的项目发生任何问题, IntelliJ IDEA 会在 **Build** 页面显示错误信息.

下一步做什么?

学习如何 为真正的 Kotlin/Native 项目编写 Gradle 构建脚本 (跨平台程序的 Gradle DSL 参考文档).

Kotlin/Native 开发入门 - 使用命令行编译器

最终更新: 2024/09/10

获取编译器

Kotlin/Native 编译器可以运行于 macOS, Linux, 以及 Windows 环境. 它是一个命令行工具, 作为 Kotlin 的一部分发布, 可以在 GitHub 发布页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.23) 下载. 它支持不同的编译目标, 包括 Linux, macOS, iOS, 等等. 参见 所有支持的编译目标完整列表 (Kotlin/Native 支持的目标平台). 尽管跨平台编译是可能的, 也就是说可以在某个平台上针对另一个平台进行编译, 但在这个 Kotlin case 中, 我们只在相同的平台上编译.

尽管编译器的输出不包含任何依赖项, 也不要求任何虚拟机, 但编译器本身需要 Java 1.8 或更高版本的运行环境 (https://jdk.java.net/11/).

安装编译器的方法是, 在某个目录解开它的压缩包, 然后将它的 `/bin` 目录路径添加到 `PATH` 环境变量中.

编写 "Hello Kotlin/Native" 程序

这个应用程序将会向标准输出打印 "Hello Kotlin/Native". 在你选择的工作目录, 创建一个文件, 名为 `hello.kt`, 并输入以下内容:

```
fun main() {
    println("Hello Kotlin/Native!")
}
```

在控制台编译代码

要编译应用程序, 请使用从 这里 (https://github.com/JetBrains/kotlin/releases) 下载的编译器, 执行以下命令:

```
kotlinc-native hello.kt -o hello
```

`-o` 选项的值指定编译输出文件的名称, 因此这个命令会生成一个名为 `hello.kexe` (在 Linux 和 macOS 平台) 或 `hello.exe` (在 Windows 平台) 的二进制文件. 关于可用的编译器选项完整列表, 请参见 编译器选项参考 (Kotlin 编译器选项).

在控制台编译似乎很简单清楚, 但不适合于包含数百个文件和库的大项目. 对于真正的项目, 推荐使用 构建系统 (Kotlin/Native 开发入门 - 使用 Gradle) 和 IDE (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA).

与 C 代码交互

最终更新: 2024/09/10

> C 库的导入是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). cinterop 工具从 C 库生成的所有 Kotlin 声明都应该标注 @ExperimentalForeignApi 注解.
>
> Kotlin/Native 自带的原生平台库 (例如 Foundation, UIKit, 和 POSIX), 只对一部分 API 需要使用者明确同意(Opt-in). 对于这样的情况, 你会在 IDE 中看到警告信息.

Kotlin/Native 遵循 Kotlin 的传统, 提供与既有的平台软件的优秀的互操作性. 对于原生程序来说, 最重要的互操作性对象就是与 C 语言库. 因此 Kotlin/Native 附带了 `cinterop` 工具, 可以用来快速生成与既有的外部库交互时所需要的一切.

与原生库交互时的工作流程如下:

1. 创建一个 `.def` 文件, 描述需要绑定(binding)的内容.
2. 使用 `cinterop` 工具生成绑定.
3. 运行 Kotlin/Native 编译器, 编译应用程序, 产生最终的可执行文件.

互操作性工具会分析 C 语言头文件, 并产生一个 "自然的" 映射, 将数据类型, 函数, 常数, 引入到 Kotlin 语言的世界. 工具生成的桩代码(stub)可以导入 IDE, 用来帮助代码自动生成, 以及代码跳转.

此外还提供了与 Swift/Objective-C 语言的互操作功能, 详情请参见 与 Swift/Objective-C 的交互 (与 Swift/Objective-C 代码交互).

平台库

注意, 很多情况下不要用到自定义的互操作库创建机制(我们后文将会介绍), 因为对于平台上的标准绑定中的那些 API, 可以使用 平台库 (平台库). 比如, Linux/macOS 平台上的 POSIX, Windows 平台上的 Win32, macOS/iOS 平台上的以及 Apple 框架, 都可以通过这种方式来使用.

一个简单的示例

首先我们安装 libgit2, 并为 git 库准备桩代码:

```
cd samples/gitchurn
../../dist/bin/cinterop -def src/nativeInterop/cinterop/libgit2.def
```

```
\
 -compiler-option -I/usr/local/include -o libgit2
```

编译客户端代码:

```
../../dist/bin/kotlinc src/gitChurnMain/kotlin \
 -library libgit2 -o GitChurn
```

运行客户端代码:

```
./GitChurn.kexe ../..
```

为一个新库创建绑定

要对一个新的库创建绑定, 首先要创建一个 `.def` 文件. 它的结构只是一个简单的 property 文件, 大致是这个样子:

```
headers = png.h
headerFilter = png.h
package = png
```

然后运行 `cinterop` 文件, 参数大致如下 (注意, 对于主机上没有被包含到 sysroot 查找路径的那些库, 可能需要指定头文件):

```
cinterop -def png.def -compiler-option -I/usr/local/include -o png
```

这个命令将会生成一个编译后的库, 名为 `png.klib`, 以及 `png-build/kotlin` 目录, 其中包含这个库的 Kotlin 源代码.

如果需要修改针对某个平台的参数, 你可以使用 `compilerOpts.osx` 或 `compilerOpts.linux` 这样的格式, 来指定这个平台专用的命令行选项.

注意, 生成的绑定通常是平台专有的, 因此如果你需要针对多个平台进行开发, 那么需要重新生成这些绑定.

生成绑定后, IDE 可以使用其中的信息来查看原生库.

对于一个典型的带配置脚本的 Unix 库, 使用 `--cflags` 参数运行配置脚本的输出结果, 通常可以用做 `compilerOpts`, (但可能不使用完全相同的路径).

使用 --libs 参数运行配置脚本的输出结果, 编译时可以用作 kotlinc 的 -linkedArgs 参数值(带引号括起).

选择库的头文件

使用 #include 指令将库的头文件导入 C 程序时, 这些头文件包含的所有其他头文件也会一起被导入. 因此, 在生成的 stub 代码内, 也会带有所有依赖到的其他头文件.

这种方式通常是正确的, 但对于某些类来说可能非常不方便. 因此可以在 .def 文件内指定需要导入哪些头文件. 如果直接依赖某个头文件的话, 也可以对它单独声明.

使用 glob 过滤头文件

也可以使用 .def 文件内的过滤属性作为 glob 来过滤头文件. 这些属性值会被看作一个空格分隔的 glob 列表.

- 要包含头文件中的声明, 请使用 headerFilter 属性. 如果包含的头文件与任何一个 glob 匹配, 那么头文件的声明就会被包含在绑定内容中.

  glob 应用于相对于恰当的包含路径元素的头文件路径, 例如, time.h 或 curl/curl.h. 因此, 如果通常使用 #include <SomeLibrary/Header.h> 指令来包含某个库, 那么应该使用下面的过滤设置来过滤头文件:

  ```
  headerFilter = SomeLibrary/**
  ```

  如果没有指定 headerFilter, 那么会包含所有的头文件. 但是, 我们鼓励使用 headerFilter, 并尽量精确的指定 glob. 这种情况下, 生成的库只包含必须的声明. 在你的开发环境中升级 Kotlin 或工具时, 可以避免很多问题的发生.

- 要排除某个头文件, 请使用 excludeFilter 属性.

  这样可以删除多余的或有问题的头文件, 并优化编译过程, 因为指定的头文件中的声明不会被包含在绑定内容中.

  ```
  excludeFilter = SomeLibrary/time.h
  ```

> 如果同一个头文件由 headerFilter 指定为包含, 同时又由 excludeFilter 指定为排除, 那么后一个设定的优先级更高. 指定的头文件不会被包含在绑定内容中.

使用模块映射过滤头文件

有些库在它的头文件中带有 module.modulemap 或 module.map 文件. 比如, macOS 和 iOS 系统库和框架就是这样. 模块映射文件(module map file) (https://clang.llvm.org/docs/Modules.html#module-map-language) 描述头文件与模块之间的对应关系. 如果存在模块映射, 那么可以使用 .def 文件的实验性的 excludeDependentModules 选项, 将模块中没有直接使用的头文件过滤掉:

```
headers = OpenGL/gl.h OpenGL/glu.h GLUT/glut.h
compilerOpts = -framework OpenGL -framework GLUT
excludeDependentModules = true
```

如果同时使用 excludeDependentModules 和 headerFilter, 那么最终起作用的将是二者的交集.

C 编译器与链接器选项

可以在定义文件中使用 compilerOpts 和 linkerOpts 来分别指定 传递给 C 编译器 (用于分析头文件, 比如预处理定义信息) 和链接器 (用于链接最终的可执行代码) 的参数. 比如:

```
compilerOpts = -DFOO=bar
linkerOpts = -lpng
```

也可以指定某个目标平台独有的参数, 比如:

```
compilerOpts = -DBAR=bar
compilerOpts.linux_x64 = -DFOO=foo1
compilerOpts.macos_x64 = -DFOO=foo2
```

通过这样的配置, C 头文件在 Linux 上的会使用 -DBAR=bar -DFOO=foo1 参数进行分析, macOS 上则会使用 -DBAR=bar -DFOO=foo2 参数进行分析. 注意, 定义文件的任何参数, 都可以包含共用的, 以及平台独有的两部分.

链接器错误

当一个 Kotlin 库依赖于一个 C 或 Objective-C 库时, 可能会发生链接器错误, 例如, 使用 CocoaPods 集成 (CocoaPods 概述与设置) 时. 如果依赖的库在当前机器上没有安装, 在项目的构建脚本中也没有明确的配置, 那么就会发生 "Framework not found" 错误.

如果你是库的作者, 你可以通过自定义消息来帮助你的用户解决链接器错误. 方法是, 在你的 .def 文件中添加 userSetupHint=message 属性, 或者向 cinterop 传递 -Xuser-setup-hint 编译器选项.

添加自定义声明

在生成绑定之前, 有时会需要向库添加自定义的 C 声明(比如, 对 宏). 你可以将它们直接包含在 .def 文件尾部, 放在一个分隔行 --- 之后, 而不需要为他们创建一个额外的头文件:

```
headers = errno.h

---

static inline int getErrno() {
    return errno;
}
```

注意, .def 文件的这部分内容会被当做头文件的一部分, 因此, 带函数体的函数应该声明为 static 函数. 这些声明的内容, 会在 headers 列表中的文件被引入之后, 再被解析.

将静态库包含到你的 klib 库中

有些时候, 发布你的程序时附带上所需要的静态库, 而不是假定它在用户的环境中已经存在了, 这样会更便利一些. 如果需要在 .klib 中包含静态库, 可以使用 staticLibrary 和 libraryPaths 语句. 比如:

```
headers = foo.h
staticLibraries = libfoo.a
libraryPaths = /opt/local/lib /usr/local/opt/curl/lib
```

如果指定了以上内容, 那么 cinterop 工具将会在 /opt/local/lib 和 /usr/local/opt/curl/lib 目录中搜索 libfoo.a 文件, 如果找到这个文件, 就会把这个库包含到 klib 内.

使用这样的 klib, 库文件会就被自动链接到你的程序内.

绑定

基本的 interop 数据类型

C 中支持的所有数据类型, 都有对应的 Kotlin 类型:

- 有符号整数, 无符号整数, 以及浮点类型, 会被映射为 Kotlin 中的同样类型, 并且长度相同.
- 指针和数组映射为 CPointer<T>? 类型.
- 枚举型映射为 Kotlin 的枚举型, 或整数型, 由 heuristic 以及 定义文件中的提示 决定.

- 结构体(Struct)和联合体(Union)映射为通过点号访问的域的形式, 也就是 someStructInstance.field1 的形式.
- typedef 映射为 typealias.

此外, 任何 C 类型都有对应的 Kotlin 类型来表达这个类型的左值(lvalue), 也就是, 在内存中分配的那个值, 而不是简单的不可变的自包含值. 你可以想想 C++ 的引用, 与这个概念类似. 对于结构体(Struct) (以及指向结构体的 typedef) 左值类型就是它的主要表达形式, 而且使用与结构体本身相同的名字, 对于 Kotlin 枚举类型, 左值类型名称是 ${type}Var, 对于 CPointer<T>, 左值类型名称是 CPointerVar<T>, 对于大多数其他类型, 左值类型名称是 ${type}Var.

对于兼有这两种表达形式的类型, 包含 "左值(lvalue)" 的那个类型, 带有一个可变的 .value 属性, 可以用来访问这个左值.

指针类型

CPointer<T> 的类型参数 T 必须是上面介绍的 "左值(lvalue)" 类型之一, 比如, C 类型 struct S* 会被映射为 CPointer<S>, int8_t* 会被映射为 CPointer<int_8tVar>, char** 会被映射为 CPointer<CPointerVar<ByteVar>>.

C 的空指针(null) 在 Kotlin 中表达为 null, 指针类型 CPointer<T> 是不可为空的, 而 CPointer<T>? 类型则是可为空的. 这种类型的值支持 Kotlin 的所有涉及 null 值处理的操作, 比如 ?:, ?., !! 等等:

```
val path = getenv("PATH")?.toKString() ?: ""
```

由于数组也被映射为 CPointer<T>, 因此这个类型也支持 [] 操作, 可以使用下标来访问数组中的值:

```
fun shift(ptr: CPointer<BytePtr>, length: Int) {
    for (index in 0 .. length - 2) {
        ptr[index] = ptr[index + 1]
    }
}
```

CPointer<T> 的 .pointed 属性返回这个指针指向的那个位置的类型 T 的左值. 相反的操作是 .ptr: 它接受一个左值, 返回一个指向它的指针.

void* 映射为 COpaquePointer – 这是一个特殊的指针类型, 它是任何其他指针类型的超类. 因此, 如果 C 函数接受 void* 类型参数, 那么绑定的 Kotlin 函数就可以接受任何 CPointer 类型参数.

可以使用 .reinterpret<T> 来对一个指针进行类型变换(包括 COpaquePointer), 例如:

```
val intPtr = bytePtr.reinterpret<IntVar>()
```

或者

```
val intPtr: CPointer<IntVar> = bytePtr.reinterpret()
```

和 C 一样, 这样的类型变换是不安全的, 可能导致应用程序发生潜在的内存错误.

对于 `CPointer<T>?` 和 `Long` 也有不安全的类型变换方法, 由扩展函数 `.toLong()` 和 `.toCPointer<T>()` 实现:

```
val longValue = ptr.toLong()
val originalPtr = longValue.toCPointer<T>()
```

注意, 如果结果类型可以通过上下文确定, 那么类型参数可以省略, 就象 Kotlin 中通常的类型系统一样.

内存分配

可以使用 `NativePlacement` 接口来分配原生内存, 比如:

```
val byteVar = placement.alloc<ByteVar>()
```

或者

```
val bytePtr = placement.allocArray<ByteVar>(5)
```

内存最 "自然" 的位置就是在 `nativeHeap` 对象内. 这个操作就相当于使用 `malloc` 来分配原生内存, 另外还提供了 `.free()` 操作来释放已分配的内存:

```
val buffer = nativeHeap.allocArray<ByteVar>(size)
<使用 buffer>
nativeHeap.free(buffer)
```

然而, 分配的内存的生命周期通常会限定在一个指明的作用范围内. 可以使用 `memScoped { ... }` 来定义这样的作用范围. 在括号内部, 可以以隐含的接收者的形式访问到一个临时的内存分配位置, 因此可以使用 `alloc` 和 `allocArray` 来分配原生内存, 离开这个作用范围后, 已分配的这些内存会被自动释放.

比如, 如果一个 C 函数, 使用指针参数返回值, 可以用下面这种方式来使用这个函数:

```
val fileSize = memScoped {
    val statBuf = alloc<stat>()
    val error = stat("/", statBuf.ptr)
    statBuf.st_size
}
```

向绑定传递指针

尽管 C 指针被映射为 `CPointer<T>` 类型, 但 C 函数的指针型参数会被映射为 `CValuesRef<T>` 类型. 如果向这样的参数传递 `CPointer<T>` 类型的值, 那么会原样传递给 C 函数. 但是, 也可以直接传递值的序列, 而不是传递指针. 这种情况下, 序列会以值的形式传递(by value), 也就是说, C 函数收到一个指针, 指向这个值序列的一个临时拷贝, 这个临时拷贝只在函数返回之前存在.

`CValuesRef<T>` 形式表达的指针型参数是为了用来支持 C 数组字面值, 而不必明确地进行内存分配操作. 为了构造一个不可变的自包含的 C 的值的序列, 可以使用下面这些方法:

- `${type}Array.toCValues()`, 其中 `type` 是 Kotlin 的基本类型
- `Array<CPointer<T>?>.toCValues()`, `List<CPointer<T>?>.toCValues()`
- `cValuesOf(vararg elements: ${type})`, 其中 `type` 是基本类型, 或指针

比如:

C 代码:

```
void foo(int* elements, int count);
...
int elements[] = {1, 2, 3};
foo(elements, 3);
```

Kotlin 代码:

```
foo(cValuesOf(1, 2, 3), 3)
```

字符串

与其它指针不同, `const char*` 类型参数会被表达为 Kotlin 的 `String` 类型. 因此对于 C 中期望字符串的绑定, 可以传递 Kotlin 的任何字符串值.

还有一些工具, 可以用来在 Kotlin 和 C 的字符串之间进行手工转换:

- `fun CPointer<ByteVar>.toKString(): String`
- `val String.cstr: CValuesRef<ByteVar>`.

要得到指针, `.cstr` 应该在原生内存中分配, 比如:

```
val cString = kotlinString.cstr.getPointer(nativeHeap)
```

在所有这些场合, C 字符串的编码都是 UTF-8.

要跳过字符串的自动转换, 并确保在绑定中使用原生的指针, 可以在 `.def` 文件中使用 `noStringConversion` 语句, 也就是:

```
noStringConversion = LoadCursorA LoadCursorW
```

通过这种方式, 任何 `CPointer<ByteVar>` 类型的值都可以传递给 `const char*` 类型的参数. 如果需要传递 Kotlin 字符串, 可以使用这样的代码:

```
memScoped {
    LoadCursorA(null, "cursor.bmp".cstr.ptr)   // 对这个函数的 ASCII
版
    LoadCursorW(null, "cursor.bmp".wcstr.ptr)  // 对这个函数的 Unicode
版
}
```

作用范围内的局部指针

`memScoped { ... }` 内有一个 `CValues<T>.ptr` 扩展属性, 使用它可以创建一个指向 `CValues<T>` 的 C 指针, 这个指针被限定在一个作用范围内. 通过它可以使用需要 C 指针的 API, 指针的生命周期限定在特定的 `MemScope` 内. 比如:

```
memScoped {
    items = arrayOfNulls<CPointer<ITEM>?>(6)
    arrayOf("one", "two").forEachIndexed { index, value ->
items[index] = value.cstr.ptr }
    menu = new_menu("Menu".cstr.ptr, items.toCValues().ptr)
```

```
    ...
}
```

在这个示例程序中, 所有传递给 C API `new_menu()` 的值, 生命周期都被限定在它所属的最内层的 `memScope` 之内. 一旦程序的执行离开了 `memScoped` 作用范围, C 指针就不再存在了.

以值的方式传递和接收结构

如果一个 C 函数以传值的方式接受结构体(Struct)或联合体(Union) `T` 类型的参数, 或者以传值的方式返回结构体(Struct)或联合体(Union) `T` 类型的结果, 对应的参数类型或结果类型会被表达为 `CValue<T>`.

`CValue<T>` 是一个不透明(opaque)类型, 因此无法通过适当的 Kotlin 属性访问到 C 结构体的域. 如果 API 以句柄的形式使用结构体, 那么这样是可行的, 但是如果确实需要访问结构体中的域, 那么可以使用以下转换方法:

- `fun T.readValue(): CValue<T>`. 将(左值) `T` 转换为一个 `CValue<T>`. 因此, 如果要构造一个 `CValue<T>`, 可以先分配 `T`, 为其中的域赋值, 然后将它转换为 `CValue<T>`.
- `CValue<T>.useContents(block: T.() -> R): R`. 将 `CValue<T>` 临时放到内存中, 然后使用放置在内存中的这个 `T` 值作为接收者, 来运行参数中指定的 Lambda 表达式. 因此, 如果要读取结构体中一个单独的域, 可以使用下面的代码:

  ```
  val fieldValue = structValue.useContents { field }
  ```

回调

如果要将一个 Kotlin 函数转换为一个指向 C 函数的指针, 可以使用 `staticCFunction(::kotlinFunction)`. 也可以使用 Lambda 表达式来代替函数引用. 这里的函数或 Lambda 表达式不能捕获任何值.

向回调传递用户数据

C API 经常允许向回调传递一些用户数据. 这些数据通常由用户在设置回调时提供. 数据使用比如 `void*` 的形式, 传递给某些 C 函数 (或写入到结构体内). 但是, Kotlin 对象的引用无法直接传递给 C. 因此需要在设置回调之前包装这些数据, 然后在回调函数内部将它们解开, 这样才能通过 C 函数来再两段 Kotlin 代码之间传递数据. 这种数据包装可以使用 `StableRef` 类实现.

要封装一个 Kotlin 对象的引用, 可以使用以下代码:

```
val stableRef = StableRef.create(kotlinReference)
```

```
val voidPtr = stableRef.asCPointer()
```

这里的 voidPtr 是一个 COpaquePointer 类型, 因此可以传递给 C 函数.

要解开这个引用, 可以使用以下代码:

```
val stableRef = voidPtr.asStableRef<KotlinClass>()
val kotlinReference = stableRef.get()
```

这里的 kotlinReference 就是封装之前的 Kotlin 对象引用.

创建 StableRef 后, 最终需要使用 .dispose() 方法手动释放, 以避免内存泄漏:

```
stableRef.dispose()
```

释放后, 它就变得不可用了, 因此 voidPtr 也不能再次解开.

更多详情请参见 samples/libcurl.

宏

每个展开为常数的 C 语言宏, 都会表达为一个 Kotlin 属性. 其他的宏都不支持. 但是, 可以将它们封装在支持的声明中, 这样就可以手动映射这些宏. 比如, 类似于函数的宏 FOO 可以映射为函数 foo, 方法是向库 添加自定义的声明:

```
headers = library/base.h

---

static inline int foo(int arg) {
    return FOO(arg);
}
```

定义文件提示

.def 支持几种选项, 用来调整最终生成的绑定.

- excludedFunctions 属性值是一个空格分隔的列表, 表示哪些函数应该忽略. 有时会需要这个功能, 因为 C 头文件中的一个函数声明, 并不保证它一定可以调用, 而且常常很难, 甚至不可能自动判断. 这个选项也可以用来绕过 interop 工具本身的 bug.

- `strictEnums` 和 `nonStrictEnums` 属性值是空格分隔的列表, 分别表示哪些枚举类型需要生成为 Kotlin 枚举类型, 哪些需要生成为整数值. 如果一个枚举型在这两个属性中都没有包括, 那么就根据 heuristic 来生成.
- `noStringConversion` 属性值是一个空格分隔的列表, 表示哪些函数的 `const char*` 参数应该不被自动转换为 Kotlin 的字符串类型.

可移植性

有时, C 库中的函数参数, 或结构体的域使用了依赖于平台的数据类型, 比如 `long` 或 `size_t`. Kotlin 本身没有提供隐含的整数类型转换, 也没有提供 C 风格的整数类型转换 (比如, `(size_t) intValue`), 因此, 在这种情况下, 为了让编写可以移植的代码变得容易一点, 提供了 `convert` 方法:

```
fun ${type1}.convert<${type2}>(): ${type2}
```

这里, `type1` 和 `type2` 都必须是整数类型, 可以是有符号整数, 可以可以是无符号整数.

`.convert<${type}>` 的含义等同于 `.toByte`, `.toShort`, `.toInt`, `.toLong`, `.toUByte`, `.toUShort`, `.toUInt` 或 `.toULong` 方法, 具体等于哪个, 取决于 `type` 的具体类型.

使用 `convert` 的示例如下:

```
fun zeroMemory(buffer: COpaquePointer, size: Int) {
    memset(buffer, 0, size.convert<size_t>())
}
```

而且, 这个函数的类型参数可以自动推定得到, 因此很多情况下可以省略.

对象固定

Kotlin 对象可以固定(pin), 也就是, 确保它们在内存中的位置不会变化, 直到解除固定(unpin)为止, 而且, 指向这些对象的内部数据的指针, 可以传递给 C 函数. 比如:

```
fun readData(fd: Int): String {
    val buffer = ByteArray(1024)
    buffer.usePinned { pinned ->
        while (true) {
            val length = recv(fd, pinned.addressOf(0),
buffer.size.convert(), 0).toInt()

            if (length <= 0) {
               break
```

```
            }
            // 现在 `buffer` 中包含了从 `recv()` 函数调用得到的原生数据.
        }
    }
}
```

这里我们使用了服务函数 `usePinned`, 它会先固定一个对象, 然后执行一段代码, 最后无论是正常结束还是异常结束, 它都会将对象解除固定.

教程 - 映射 C 语言的基本数据类型

最终更新: 2024/09/10

> ⚠ C 库导入是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). `cinterop` 工具从 C 库生成的所有 Kotlin 声明都应该标注 `@ExperimentalForeignApi` 注解.
>
> Kotlin/Native 自带的原生平台库 (例如 Foundation, UIKit, 和 POSIX), 只对一部分 API 需要使用者明确同意(Opt-in). 对于这样的情况, 你会在 IDE 中看到警告信息.

通过本教程, 你将学习 C 数据类型在 Kotlin/Native 中会变成什么类型, 以及反过来. 你将会:

- 学习 C 语言中有什么数据类型.
- 创建一个 小小的 C 库, 在库的导出(export)中使用这些类型.
- 查看为 C 库生成的 Kotlin API.
- 学习 Kotlin 中的基本类型 如何映射到 C.

C 语言中的数据类型

在 C 语言中有些什么数据类型? 我们来看看 Wiki 词条 C 数据类型 (https://en.wikipedia.org/wiki/C_data_types). C 程序语言中有以下数据类型:

- 基本类型 `char, int, float, double`, 以及修饰符 `signed, unsigned, short, long`
- 结构(Structure), 联合(Union), 数组(Array)
- 指针(Pointer)
- 函数指针(Function Pointer)

还有更加具体的类型:

- boolean 类型 (由 C99 (https://en.wikipedia.org/wiki/C99) 定义)
- `size_t` 和 `ptrdiff_t` (以及 `ssize_t`)

- 定宽整数类型, 比如 int32_t 或 uint64_t (由 C99 (https://en.wikipedia.org/wiki/C99) 定义)

C 语言中还有以下类型修饰符: const, volatile, restruct, atomic.

要了解 C 数据类型在 Kotlin 中会变成什么, 最好的办法是实践一下.

C 库示例

创建一个 lib.h 文件, 看看 C 函数如何映射到 Kotlin:

```
#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void ints(char c, short d, int e, long f);
void uints(unsigned char c, unsigned short d, unsigned int e,
unsigned long f);
void doubles(float a, double b);

#endif
```

这个文件缺少了 extern "C" 代码段, 这个示例不需要这部分, 但如果你使用 C++ 和重载(overloaded)函数, 那么可能会需要. Stackoverflow 上的 C++ 兼容性(https://stackoverflow.com/questions/1041866/what-is-the-effect-of-extern-c-in-c) 一文介绍了关于这个问题的更多细节.

对每一组 .h 文件, 你将使用 Kotlin/Native 的 cinterop 工具 (与 C 代码交互) 来生成一个 Kotlin/Native 库, 也就是 .klib 文件. 生成的库会将来自 Kotlin/Native 的调用桥接给 C 代码. 它包含 .h 文件中的定义对应的 Kotlin 声明. 只需要一个 .h 文件来运行 cinterop 工具. 而且你不需要创建一个 lib.c 文件, 除非你想要编译并运行这个示例. 更多详情请参见 与 C 代码交互 (与 C 代码交互). 对本教程来说, 创建以下内容的 lib.def 文件就够了:

```
headers = lib.h
```

你可以在 .def 文件中直接包含所有的声明, 放在一个 --- 分隔之后. 要在 cinterop 工具生成的代码中包含宏或其他 C 定义, 这种方法可以很有用. 方法体会被编译并完全包含在二进制文件中. 使用这个功能可以产生一个可运行的示例, 而不需要使用 C 编译器. 要实现这一点, 你需要为来自 lib.h 文件的 C 函数添加实现, 并将这些函数放在一个 .def 文件内. 最后你的 interop.def 如下:

```
---
```

```
void ints(char c, short d, int e, long f) { }
void uints(unsigned char c, unsigned short d, unsigned int e,
unsigned long f) { }
void doubles(float a, double b) { }
```

这个 interop.def 文件已经足以编译并运行应用程序, 或在 IDE 中打开它. 现在我们来创建 项目 文件, 在 IntelliJ IDEA (https://jetbrains.com/idea) 中打开项目, 并运行它.

查看为 C 库生成的 Kotlin API

尽管可以直接使用命令行, 或者通过脚本文件(比如 .sh 或 .bat 文件)调用命令行, 但这种方法不适合于包含几百个文件和库的大项目. 更好的方法是使用带有构建系统的 Kotlin/Native 编译器, 因为它会帮助你下载并缓存 Kotlin/Native 编译器二进制文件, 传递依赖的库, 并运行编译器和测试. Kotlin/Native 能够通过 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 使用 Gradle (https://gradle.org) 构建系统.

关于如何使用 Gradle 设置 IDE 兼容的项目, 请参见教程 一个基本的 Kotlin/Native 应用程序 (Kotlin/Native 开发入门 - 使用 Gradle). 如果你想要寻找具体的步骤指南, 来开始一个新的 Kotlin/Native 项目并在 IntelliJ IDEA 中打开它, 请先阅读这篇教程. 在本教程中, 我们关注更高级的 C 交互功能, 包括使用 Kotlin/Native, 以及使用 Gradle 的 跨平台 ("编译到多个目标平台" in "配置 Gradle 项目") 构建.

首先, 创建一个项目文件夹. 本教程中的所有路径都是基于这个文件夹的相对路径. 有时在添加任何新文件之前, 会需要创建缺少的目录.

使用以下 build.gradle(.kts) Gradle 构建文件:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
```

```
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64("native") { // 用于 Windows 环境
        val main by compilations.getting
        val interop by main.cinterops.creating

        binaries {
            executable()
        }
    }
}

tasks.wrapper {
    gradleVersion = "8.1.1"
    distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64('native') { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64('native') { // 用于 Windows 环境
        compilations.main.cinterops {
            interop
        }

        binaries {
            executable()
```

```
        }
    }
}

wrapper {
    gradleVersion = '8.1.1'
    distributionType = 'BIN'
}
```

项目文件将 C interop 配置为构建的一个额外步骤. 下面将 `interop.def` 文件移动到 `src/nativeInterop/cinterop` 目录. Gradle 推荐使用符合约定习惯的文件布局, 而不是使用额外的配置, 比如, 源代码文件应该放在 `src/nativeMain/kotlin` 文件夹中. 默认情况下, 来自 C 的所有符号会被导入到 `interop` 包, 你可能想要在我们的 `.kt` 文件中导入整个包. 请查看 Multiplatform Gradle DSL 参考文档 (跨平台程序的 Gradle DSL 参考文档), 学习它的各种配置方法.

创建一个 `src/nativeMain/kotlin/hello.kt` 桩(stub)文件, 内容如下, 看看 C 基本数据类型声明在 Kotlin 中会变成什么:

```
import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    ints(/*fix me*/)
    uints(/*fix me*/)
    doubles(/*fix me*/)
}
```

现在你可以 在 IntelliJ IDEA 中打开项目 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA), 看看如何修正示例项目. 在这个过程中, 我们来看看 C 基本数据类型如何映射到 Kotlin/Native.

基本数据类型在 Kotlin 中的映射结果

通过 IntelliJ IDEA 的 **Go to | Declaration** 的帮助, 或查看编译器错误, 你可以看到为 C 函数生成的 API:

```
fun ints(c: Byte, d: Short, e: Int, f: Long)
fun uints(c: UByte, d: UShort, e: UInt, f: ULong)
```

```
fun doubles(a: Float, b: Double)
```

C 类型按照我们期望的方式映射成了 Kotlin 类型, 注意 char 类型映射为 kotlin.Byte, 因为它通常是 8 bit 有符号值.

C 类型	Kotlin 类型
char	kotlin.Byte
unsigned char	kotlin.UByte
short	kotlin.Short
unsigned short	kotlin.UShort
int	kotlin.Int
unsigned int	kotlin.UInt
long long	kotlin.Long
unsigned long long	kotlin.ULong
float	kotlin.Float
double	kotlin.Double

修正代码

你已经看到了所有的定义, 现在我们来修正代码. 在 IDE 中 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA) 运行 runDebugExecutableNative Gradle task, 或使用以下命令来运行代码:

```
./gradlew runDebugExecutableNative
```

最终的 hello.kt 文件中的代码大致如下:

```
import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    ints(1, 2, 3, 4)
    uints(5, 6, 7, 8)
    doubles(9.0f, 10.0)
}
```

下一步

阅读以下教程, 继续探索更多复杂的 C 语言数据类型, 以及它们在 Kotlin/Native 中的表达:

- 映射 C 语言的结构(Struct)和联合(Union)类型 (教程 - 映射 C 语言的结构(Struct)和联合(Union)类型)
- 映射 C 语言的函数指针(Function Pointer) (教程 - 映射 C 语言的函数指针(Function Pointer))
- 映射 C 语言的字符串 (教程 - 映射 C 语言的字符串)

与 C 代码交互 (与 C 代码交互) 文档还讲解了更多的高级使用场景.

教程 - 映射 C 语言的结构(Struct)和联合(Union)类型

最终更新: 2024/09/10

> C 库导入是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). cinterop 工具从 C 库生成的所有 Kotlin 声明都应该标注 @ExperimentalForeignApi 注解.
>
> Kotlin/Native 自带的原生平台库 (例如 Foundation, UIKit, 和 POSIX), 只对一部分 API 需要使用者明确同意(Opt-in). 对于这样的情况, 你会在 IDE 中看到警告信息.

这是本系列的第 2 篇教程. 第 1 篇教程是 映射 C 语言的基本数据类型 (教程 - 映射 C 语言的基本数据类型). 此外还有教程 映射 C 语言的函数指针(Function Pointer) (教程 - 映射 C 语言的函数指针(Function Pointer)) 和教程 映射 C 语言的字符串 (教程 - 映射 C 语言的字符串).

本教程中, 你将学习:

- 如何映射结构(Struct)和联合(Union)类型
- 如何在 Kotlin 中使用结构和联合类型

映射 C 的结构(Struct)和联合(Union)类型

要理解 Kotlin 和 C 之间的映射, 最好的方法是试验一段小示例程序. 我们在 C 语言中声明一个结构和一个联合, 看看它们如何映射到 Kotlin.

Kotlin/Native 带有 `cinterop` 工具; 这个工具会生成 C 语言和 Kotlin 之间的绑定. 它使用一个 `.def` 文件来指定一个要导入的 C 库. 详情请参见教程 与 C 库交互 (与 C 代码交互).

在 前面的教程 (教程 - 映射 C 语言的基本数据类型) 中, 你创建了一个 `lib.h` 文件. 现在, 直接在 `interop.def` 文件中包含这些声明, 在专门的 `---` 分割行之后:

```
---

typedef struct {
    int a;
    double b;
} MyStruct;
```

```
void struct_by_value(MyStruct s) {}
void struct_by_pointer(MyStruct* s) {}

typedef union {
    int a;
    MyStruct b;
    float c;
} MyUnion;

void union_by_value(MyUnion u) {}
void union_by_pointer(MyUnion* u) {}
```

这个 interop.def 文件已经足以编译和运行应用程序, 或在 IDE 中打开它. 现在来创建项目文件, 在 IntelliJ IDEA (https://jetbrains.com/idea)中打开项目, 并运行它.

查看为 C 库生成的 Kotlin API

尽管可以直接使用命令行, 或者通过脚本文件(比如 .sh 或 .bat 文件), 但这种方法不适合于包含几百个文件和库的大项目. 更好的方法是使用带有构建系统的 Kotlin/Native 编译器, 因为它会帮助你下载并缓存 Kotlin/Native 编译器二进制文件, 传递依赖的库, 并运行编译器和测试. Kotlin/Native 能够通过 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 使用 Gradle (https://gradle.org) 构建系统.

关于如何使用 Gradle 设置 IDE 兼容的项目, 请参见教程 一个基本的 Kotlin/Native 应用程序 (Kotlin/Native 开发入门 - 使用 Gradle). 如果你想要寻找具体的步骤指南, 来开始一个新的 Kotlin/Native 项目并在 IntelliJ IDEA 中打开它, 请先阅读这篇教程. 在本教程中, 我们关注更高级的 C 交互功能, 包括使用 Kotlin/Native, 以及使用 Gradle 的 跨平台 ("编译到多个目标平台" in "配置 Gradle 项目") 构建.

首先, 创建一个项目文件夹. 本教程中的所有路径都是基于这个文件夹的相对路径. 有时在添加任何新文件之前, 会需要创建缺少的目录.

使用以下 build.gradle(.kts) Gradle 构建文件:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}
```

```
repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64("native") { // 用于 Windows 环境
        val main by compilations.getting
        val interop by main.cinterops.creating

        binaries {
            executable()
        }
    }
}

tasks.wrapper {
    gradleVersion = "8.1.1"
    distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64('native') { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
```

```
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64('native') { // 用于 Windows 环境
        compilations.main.cinterops {
            interop
        }

        binaries {
            executable()
        }
    }
}

wrapper {
  gradleVersion = '8.1.1'
  distributionType = 'BIN'
}
```

项目文件将 C interop 配置为构建的一个额外步骤. 下面将 `interop.def` 文件移动到 `src/nativeInterop/cinterop` 目录. Gradle 推荐使用符合约定习惯的文件布局, 而不是使用额外的配置, 比如, 源代码文件应该放在 `src/nativeMain/kotlin` 文件夹中. 默认情况下, 来自 C 的所有符号会被导入到 `interop` 包, 你可能想要在我们的 `.kt` 文件中导入整个包. 请查看 Multiplatform Gradle DSL 参考文档 (跨平台程序的 Gradle DSL 参考文档), 学习它的各种配置方法.

创建一个 `src/nativeMain/kotlin/hello.kt` 桩(stub)文件, 内容如下, 看看 C 中的结构和联合类型声明在 Kotlin 会成为什么:

```
import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    struct_by_value(/*fix me*/)
    struct_by_pointer(/*fix me*/)
    union_by_value(/*fix me*/)
    union_by_pointer(/*fix me*/)
}
```

现在你可以 在 IntelliJ IDEA 中打开项目 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA), 看看如何修

正示例项目. 在这个过程中, 我们来看看 C 的结构和联合类型如何映射到 Kotlin/Native 声明.

结构和联合类型在 Kotlin 中的映射结果

通过 IntelliJ IDEA 的 **Go to | Declaration** 的帮助, 或查看编译器错误, 你可以看到为 C 函数, `struct`, 和 `union` 生成的 API:

```
fun struct_by_value(s: CValue<MyStruct>)
fun struct_by_pointer(s: CValuesRef<MyStruct>?)

fun union_by_value(u: CValue<MyUnion>)
fun union_by_pointer(u: CValuesRef<MyUnion>?)

class MyStruct constructor(rawPtr: NativePtr /* = NativePtr */) :
CStructVar {
    var a: Int
    var b: Double
    companion object : CStructVar.Type
}

class MyUnion constructor(rawPtr: NativePtr /* = NativePtr */) :
CStructVar {
    var a: Int
    val b: MyStruct
    var c: Float
    companion object : CStructVar.Type
}
```

你可以看到 `cinterop` 为我们的 `struct` 和 `union` 类型生成的包装类型. 对 C 中声明的 `MyStruct` 和 `MyUnion` 类型, 相应的生成了 Kotlin 类 `MyStruct` 和 `MyUnion`. 这些包装类型继承自基类 `CStructVar`, 并将所有的域声明为 Kotlin 属性. 它使用 `CValue<T>` 来表达一个传值(By-Value)的结构参数, 使用 `CValuesRef<T>?` 表达传递结构或联合的指针.

技术上, 在 Kotlin 中 `struct` 和 `union` 类型没有区别. 注意, Kotlin 中 `MyUnion` 类的 `a`, `b`, 和 `c` 属性使用相同的内存位置来读写它们的值, 和 C 语言中的 `union` 一样.

关于更多细节以及高级使用场景, 请参见 与 C 代码交互 (与 C 代码交互) 文档.

在 Kotlin 中使用结构和联合类型

在 Kotlin 中为 C 的 `struct` 和 `union` 类型生成的包装类很容易使用. 由于存在那些生成的属性, 在 Kotlin 代码中使用它们感觉很自然. 目前唯一的问题是, 如何为这些类创建一个新的实例. 如你所见, 在 `MyStruct` 的 `MyUnion` 声明中, 构造函数需要一个 `NativePtr` 参数. 当然, 你不希望手动处理指针. 相反, 你可以让 Kotlin API 来为我们初始化这些对象.

我们来看一下生成的那些接受 `MyStruct` 和 `MyUnion` 参数的函数. 你可以看到, 传值的参数表达为 `kotlinx.cinterop.CValue<T>`. 对于类型指针参数则是 `kotlinx.cinterop.CValuesRef<T>`. Kotlin 为我们提供了 API 方便的处理这两种类型, 我们来试一下.

创建一个 CValue<T>

`CValue<T>` 类型用来向 C 函数传递一个传值的参数. 使用 `cValue` 函数来创建 `CValue<T>` 对象实例. 函数要求一个 带接受者的 Lambda 函数 ("带有接受者的函数字面值" in "高阶函数与 Lambda 表达式") 来初始化底层的 C 类型. 函数声明如下:

```
fun <reified T : CStructVar> cValue(initialize: T.() -> Unit):
CValue<T>
```

现在来看看如何使用 `cValue`, 并传递传值的参数:

```
fun callValue() {
    val cStruct = cValue<MyStruct> {
        a = 42
        b = 3.14
    }
    struct_by_value(cStruct)

    val cUnion = cValue<MyUnion> {
        b.a = 5
        b.b = 2.7182
    }

    union_by_value(cUnion)
}
```

使用 CValuesRef<T> 创建结构和联合

在 Kotlin 中, `CValuesRef<T>` 类型用来传递 C 函数的有类型指针参数. 首先, 你需要 `MyStruct` 和 `MyUnion` 类的实例. 可以直接在 native 内存中创建它们. 方法是对 `kotlinx.cinterop.NativePlacement` 类型使用扩展函数:

```
fun <reified T : kotlinx.cinterop.CVariable> alloc(): T
```

`NativePlacement` 表示 native 内存, 它有类似于 `malloc` 和 `free` 的函数. `NativePlacement` 有几种实现. 全局实现通过 `kotlinx.cinterop.nativeHeap` 来调用, 在使用完毕后不要忘记调用 `nativeHeap.free(..)` 函数来释放内存.

另一个选择是使用函数:

```
fun <R> memScoped(block: kotlinx.cinterop.MemScope.() -> R): R
```

它创建一个短期存在(Short-Lived)的内存分配范围(Allocation Scope), 所有分配的内存将会在 `block` 的末尾自动清除.

使用指针调用函数的代码大致如下:

```
fun callRef() {
    memScoped {
        val cStruct = alloc<MyStruct>()
        cStruct.a = 42
        cStruct.b = 3.14

        struct_by_pointer(cStruct.ptr)

        val cUnion = alloc<MyUnion>()
        cUnion.b.a = 5
        cUnion.b.b = 2.7182

        union_by_pointer(cUnion.ptr)
    }
}
```

注意, 这段代码使用来自 `memScoped` Lambda 接受者类型的扩展属性 `ptr`, 来将 `MyStruct` 和 `MyUnion` 实例转换为 native 指针.

`MyStruct` 和 `MyUnion` 类内部保存指向 native 内存的指针. 内存会在 `memScoped` 函数结束时释放, 等于在是它的 `block` 的末尾. 请确保指针没有在 `memScoped` 调用范围之外被使用. 对于需要生存周期更长的指针, 或者缓存在 C 库内部的指针, 你可以使用 `Arena()` 或 `nativeHeap`.

CValue<T> 和 CValuesRef<T> 之间的转换

当然, 有些使用场景中, 对一个函数调用你需要以值的方式传递一个结构, 然后对另一个函数调用需要以引用的方式传递同一个结构. 在 Kotlin/Native 中也是可以实现的. 这里需要用到 `NativePlacement`.

首先我们来看看 `CValue<T>` 如何转换为指针:

```
fun callMix_ref() {
    val cStruct = cValue<MyStruct> {
        a = 42
        b = 3.14
    }

    memScoped {
        struct_by_pointer(cStruct.ptr)
    }
}
```

这段代码使用来自 `memScoped` Lambda 接受者类型的扩展属性 `ptr`, 来将 `MyStruct` 和 `MyUnion` 实例转换为 native 指针. 这些指针只在 `memScoped` block 之内有效.

对于反过来的转换, 要将一个指针转换为一个传值的变量, 我们调用 `readValue()` 扩展函数:

```
fun callMix_value() {
    memScoped {
        val cStruct = alloc<MyStruct>()
        cStruct.a = 42
        cStruct.b = 3.14

        struct_by_value(cStruct.readValue())
    }
}
```

运行代码

现在你已经学习了如何在你的代码中使用 C 的声明, 你可以在真实的例子中尝试了. 我们来修正代码, 看看它如何运行, 方法是 在 IDE 中 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA) 调用 `runDebugExecutableNative` Gradle task, 或使用以下命令:

```
./gradlew runDebugExecutableNative
```

最终的 hello.kt 文件中的代码大致如下:

```
import interop.*
import kotlinx.cinterop.alloc
import kotlinx.cinterop.cValue
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.ptr
import kotlinx.cinterop.readValue

fun main() {
    println("Hello Kotlin/Native!")

    val cUnion = cValue<MyUnion> {
        b.a = 5
        b.b = 2.7182
    }

    memScoped {
        union_by_value(cUnion)
        union_by_pointer(cUnion.ptr)
    }

    memScoped {
        val cStruct = alloc<MyStruct> {
            a = 42
            b = 3.14
        }

        struct_by_value(cStruct.readValue())
        struct_by_pointer(cStruct.ptr)
    }
}
```

下一步

阅读以下教程, 继续探索更多 C 语言数据类型, 以及它们在 Kotlin/Native 中的表达:

- 映射 C 语言的基本数据类型 (教程 - 映射 C 语言的基本数据类型)

- 映射 C 语言的函数指针(Function Pointer) (教程 - 映射 C 语言的函数指针(Function Pointer))
- 映射 C 语言的字符串 (教程 - 映射 C 语言的字符串)

与 C 代码交互 (与 C 代码交互) 文档还讲解了更多的高级使用场景.

教程 - 映射 C 语言的函数指针 (Function Pointer)

最终更新: 2024/09/10

> C 库导入是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). `cinterop` 工具从 C 库生成的所有 Kotlin 声明都应该标注 `@ExperimentalForeignApi` 注解.
>
> Kotlin/Native 自带的原生平台库 (例如 Foundation, UIKit, 和 POSIX), 只对一部分 API 需要使用者明确同意(Opt-in). 对于这样的情况, 你会在 IDE 中看到警告信息.

这是本系列的第 3 篇教程. 第 1 篇教程是 映射 C 语言的基本数据类型 (教程 - 映射 C 语言的基本数据类型). 此外还有教程 映射 C 语言的结构(Struct)和联合(Union)类型 (教程 - 映射 C 语言的结构(Struct)和联合(Union)类型) 和教程 映射 C 语言的字符串 (教程 - 映射 C 语言的字符串).

本教程中, 我们将学习如何:

- 将 Kotlin 函数作为 C 函数指针传递
- 在 Kotlin 中使用 C 函数指针

映射来自 C 的函数指针类型

要理解 Kotlin 和 C 之间的映射, 最好的方法是试验一段小示例程序. 声明一个函数, 它接受一个函数指针作为参数, 以及另一个函数, 它返回一个函数指针.

Kotlin/Native 带有 `cinterop` 工具; 这个工具会生成 C 语言和 Kotlin 之间的绑定. 它使用一个 `.def` 文件来指定一个要导入的 C 库. 详情请参见 与 C 库交互 (与 C 代码交互).

试验 C API 映射的最快方法是, 将所有 C 声明都写在 `interop.def` 文件中, 完全不需要创建任何 `.h` 或 `.c` 文件. 然后将 C 声明放在一个 `.def` 文件中, 在专门的 `---` 分割行之后:

```
---

int myFun(int i) {
    return i+1;
}
```

```
typedef int (*MyFun)(int);

void accept_fun(MyFun f) {
    f(42);
}

MyFun supply_fun() {
    return myFun;
}
```

这个 interop.def 文件已经足以编译和运行应用程序, 或在 IDE 中打开它. 现在来创建项目文件, 在 IntelliJ IDEA (https://jetbrains.com/idea)中打开项目, 并运行它.

查看为 C 库生成的 Kotlin API

尽管可以直接使用命令行, 或者通过脚本文件(比如 .sh 或 .bat 文件), 但这种方法不适合于包含几百个文件和库的大项目. 更好的方法是使用带有构建系统的 Kotlin/Native 编译器, 因为它会帮助你下载并缓存 Kotlin/Native 编译器二进制文件, 传递依赖的库, 并运行编译器和测试. Kotlin/Native 能够通过 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 使用 Gradle (https://gradle.org) 构建系统.

关于如何使用 Gradle 设置 IDE 兼容的项目, 请参见教程 一个基本的 Kotlin/Native 应用程序 (Kotlin/Native 开发入门 - 使用 Gradle). 如果你想要寻找具体的步骤指南, 来开始一个新的 Kotlin/Native 项目并在 IntelliJ IDEA 中打开它, 请先阅读这篇教程. 在本教程中, 我们关注更高级的 C 交互功能, 包括使用 Kotlin/Native, 以及使用 Gradle 的 跨平台 ("编译到多个目标平台" in "配置 Gradle 项目") 构建.

首先, 创建一个项目文件夹. 本教程中的所有路径都是基于这个文件夹的相对路径. 有时在添加任何新文件之前, 会需要创建缺少的目录.

使用以下 build.gradle(.kts) Gradle 构建文件:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}

repositories {
    mavenCentral()
```

```
}

kotlin {
    linuxX64("native") { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64("native") { // 用于 Windows 环境
        val main by compilations.getting
        val interop by main.cinterops.creating

        binaries {
            executable()
        }
    }
}

tasks.wrapper {
    gradleVersion = "8.1.1"
    distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64('native') { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64('native') { // 用于 Windows 环境
        compilations.main.cinterops {
```

```
            interop
        }

        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = '8.1.1'
    distributionType = 'BIN'
}
```

项目文件将 C interop 配置为构建的一个额外步骤. 下面将 `interop.def` 文件移动到 `src/nativeInterop/cinterop` 目录. Gradle 推荐使用符合约定习惯的文件布局, 而不是使用额外的配置, 比如, 源代码文件应该放在 `src/nativeMain/kotlin` 文件夹中. 默认情况下, 来自 C 的所有符号会被导入到 `interop` 包, 你可能想要在我们的 `.kt` 文件中导入整个包. 请查看 Multiplatform Gradle DSL 参考文档 (跨平台程序的 Gradle DSL 参考文档), 学习它的各种配置方法.

创建一个 `src/nativeMain/kotlin/hello.kt` 桩(stub)文件, 内容如下, 看看 C 函数指针声明在 Kotlin 中会变成什么:

```
import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    accept_fun(/*fix me*/)
    val useMe = supply_fun()
}
```

现在你可以 在 IntelliJ IDEA 中打开项目 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA), 看看如何修正示例项目. 在这个过程中, 我们来看看 C 函数如何映射为 Kotlin/Native 声明.

C 函数指针在 Kotlin 中的映射结果

通过 IntelliJ IDEA 的 **Go To** | **Declaration or Usages** 的帮助, 或查看编译器错误, 你可以看到为 C

函数生成的 API:

```
fun accept_fun(f: MyFun? /* = CPointer<CFunction<(Int) -> Int>>? */)
fun supply_fun(): MyFun? /* = CPointer<CFunction<(Int) -> Int>>? */

fun myFun(i: kotlin.Int): kotlin.Int

typealias MyFun =
kotlinx.cinterop.CPointer<kotlinx.cinterop.CFunction<(kotlin.Int) ->
kotlin.Int>>

typealias MyFunVar = kotlinx.cinterop.CPointerVarOf<lib.MyFun>
```

你可以看到, C 函数的 `typedef` 被转换为 Kotlin 的 `typealias`. 它使用 `CPointer<..>` 类型来表示指针参数, 使用 `CFunction<(Int)->Int>` 来表示函数签名. 对所有的 `CPointer<CFunction<..>` 类型, 有一个 `invoke` 操作符扩展函数, 因此可以象 Kotlin 中的任何其他函数一样调用它.

将 Kotlin 函数作为 C 函数指针传递

下面来试验在 Kotlin 程序中使用 C 函数. 调用 `accept_fun` 函数, 并传递一个指向 Kotlin Lambda 表达式的 C 函数指针:

```
fun myFun() {
    accept_fun(staticCFunction<Int, Int> { it + 1 })
}
```

这个调用使用 Kotlin/Native 的 `staticCFunction{..}` 帮助函数, 将一个 Kotlin Lambda 表达式函数封装为一个 C 函数指针. 它只允许使用无绑定(unbound), 并且无捕获(non-capturing)的 Lambda 表达式函数. 比如, 函数内不能使用局部变量. 你只能使用全局可见的声明. 从 `staticCFunction{..}` 之内抛出异常将会导致不确定的副作用. 因此必须保证你的代码内部不会抛出任何异常.

在 Kotlin 中使用 C 函数指针

下一步是通过你从 `supply_fun()` 调用得到的指针, 调用一个 C 函数指针:

```
fun myFun2() {
    val functionFromC = supply_fun() ?: error("No function is
returned")
```

```
    functionFromC(42)
}
```

Kotlin 将函数指针返回类型转换为一个可为 null 的 `CPointer<CFunction<..>` 对象. 调用函数指针之前不需要明确检查结果是否为 `null`. 上面的示例代码中使用 Elvis 操作符 (Null 值安全性) 进行这种检查. `cinterop` 工具帮助我们将一个 C 函数指针转换为一个 Kotlin 中方便调用的对象. 我们在最后一行做的就是这种调用.

修正代码

你已经看到了所有的定义, 现在我们来修正代码. 在 IDE 中 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA) 运行 `runDebugExecutableNative` Gradle task, 或使用以下命令来运行代码:

```
./gradlew runDebugExecutableNative
```

最终的 `hello.kt` 文件中的代码大致如下:

```
import interop.*
import kotlinx.cinterop.*

fun main() {
    println("Hello Kotlin/Native!")

    val cFunctionPointer = staticCFunction<Int, Int> { it + 1 }
    accept_fun(cFunctionPointer)

    val funFromC = supply_fun() ?: error("No function is returned")
    funFromC(42)
}
```

下一步

阅读以下教程, 继续探索更多 C 语言数据类型, 以及它们在 Kotlin/Native 中的表达:

- 映射 C 语言的基本数据类型 (教程 - 映射 C 语言的基本数据类型)
- 映射 C 语言的结构(Struct)和联合(Union)类型 (教程 - 映射 C 语言的结构(Struct)和联合(Union)类型)

- 映射 C 语言的字符串 (教程 - 映射 C 语言的字符串)

与 C 代码交互 (与 C 代码交互) 文档还讲解了更多的高级使用场景.

教程 - 映射 C 语言的字符串

最终更新: 2024/09/10

> C 库导入是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). cinterop 工具从 C 库生成的所有 Kotlin 声明都应该标注 @ExperimentalForeignApi 注解.
>
> Kotlin/Native 自带的原生平台库 (例如 Foundation, UIKit, 和 POSIX), 只对一部分 API 需要使用者明确同意(Opt-in). 对于这样的情况, 你会在 IDE 中看到警告信息.

这是本系列的最后 1 篇教程. 第 1 篇教程是 映射 C 语言的基本数据类型 (教程 - 映射 C 语言的基本数据类型). 此外还有教程 映射 C 语言的结构(Struct)和联合(Union)类型 (教程 - 映射 C 语言的结构(Struct)和联合(Union)类型) 和教程 映射 C 语言的函数指针(Function Pointer) (教程 - 映射 C 语言的函数指针(Function Pointer)).

本教程中, 你会看到在 Kotlin/Native 中如何处理 C 字符串. 你将学习如何:

- 将 Kotlin 字符串传递到 C
- 在 Kotlin 中读取 C 字符串
- 将 C 字符串的字节接收到 Kotlin 字符串中

使用 C 字符串

C 语言中没有专门的字符串类型. 开发者需要根据方法签名或者文档来判断一个 char * 是不是一个 C 字符串. C 语言中的字符串使用 null 作为终止符, 末尾 0 字符 \0 添加到字节序列之后, 表示字符串结束. 通常, 使用 UTF-8 编码的字符串 (https://en.wikipedia.org/wiki/UTF-8). UTF-8 编码使用变宽字符, 而且向后兼容 ASCII (https://en.wikipedia.org/wiki/ASCII) 编码. Kotlin/Native 默认使用 UTF-8 字符编码.

要理解 Kotlin 和 C 之间的映射, 最好的方法是试验一段小示例程序. 为此我们创建一个小的库头文件. 首先, 创建 一个 lib.h 文件, 包含以下使用 C 字符串的函数声明:

```
#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void pass_string(char* str);
char* return_string();
```

```
int copy_string(char* str, int size);

#endif
```

在这个示例中, 你可以看到 C 语言中传递或接收一个字符串的最常见方式. 注意 `return_string` 的返回值. 通常, 最好确保你使用了正确的 `free(..)` 函数调用来释放返回的 `char*`.

Kotlin/Native 带有 `cinterop` 工具; 这个工具会生成 C 语言和 Kotlin 之间的绑定. 它使用一个 `.def` 文件来指定一个要导入的 C 库. 详情请参见教程 与 C 库交互 (与 C 代码交互). 试验 C API 映射的最快方法是, 将所有 C 声明都写在 `interop.def` 文件中, 完全不需要创建任何 `.h` 或 `.c` 文件. 然后将 C 声明放在一个 `interop.def` 文件中, 在专门的 `---` 分割行之后:

```
headers = lib.h
---

void pass_string(char* str) {
}

char* return_string() {
    return "C string";
}

int copy_string(char* str, int size) {
    *str++ = 'C';
    *str++ = ' ';
    *str++ = 'K';
    *str++ = '/';
    *str++ = 'N';
    *str++ = 0;
    return 0;
}
```

这个 `interop.def` 文件已经足以编译和运行应用程序, 或在 IDE 中打开它. 现在来创建项目文件, 在 IntelliJ IDEA (https://jetbrains.com/idea)中打开项目, 并运行它.

查看为 C 库生成的 Kotlin API

尽管可以直接使用命令行, 或者通过脚本文件(比如 `.sh` 或 `.bat` 文件), 但这种方法不适合于包含几百个文件和库的大项目. 更好的方法是使用带有构建系统的 Kotlin/Native 编译器, 因为它会帮助你

下载并缓存 Kotlin/Native 编译器二进制文件, 传递依赖的库, 并运行编译器和测试. Kotlin/Native 能够通过 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 使用 Gradle (https://gradle.org) 构建系统.

关于如何使用 Gradle 设置 IDE 兼容的项目, 请参见教程 一个基本的 Kotlin/Native 应用程序 (Kotlin/Native 开发入门 - 使用 Gradle). 如果你想要寻找具体的步骤指南, 来开始一个新的 Kotlin/Native 项目并在 IntelliJ IDEA 中打开它, 请先阅读这篇教程. 在本教程中, 我们关注更高级的 C 交互功能, 包括使用 Kotlin/Native, 以及使用 Gradle 的 跨平台 ("编译到多个目标平台" in "配置 Gradle 项目") 构建.

首先, 创建一个项目文件夹. 本教程中的所有路径都是基于这个文件夹的相对路径. 有时在添加任何新文件之前, 会需要创建缺少的目录.

使用以下 `build.gradle(.kts)` Gradle 构建文件:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64("native") { // 用于 Windows 环境
        val main by compilations.getting
        val interop by main.cinterops.creating

        binaries {
            executable()
        }
    }
}

tasks.wrapper {
```

```
    gradleVersion = "8.1.1"
    distributionType = Wrapper.DistributionType.BIN
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64('native') { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64('native') { // 用于 Windows 环境
        compilations.main.cinterops {
            interop
        }

        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = '8.1.1'
    distributionType = 'BIN'
}
```

项目文件将 C interop 配置为构建的一个额外步骤. 下面将 `interop.def` 文件移动到 `src/nativeInterop/cinterop` 目录. Gradle 推荐使用符合约定习惯的文件布局, 而不是使用额外的配置, 比如, 源代码文件应该放在 `src/nativeMain/kotlin` 文件夹中. 默认情况下, 来自 C 的所有符号

会被导入到 interop 包, 你可能想要在我们的 .kt 文件中导入整个包. 请查看 Multiplatform Gradle DSL 参考文档 (跨平台程序的 Gradle DSL 参考文档), 学习它的各种配置方法.

创建一个 src/nativeMain/kotlin/hello.kt 桩(stub)文件, 内容如下, 看看 C 字符串声明在 Kotlin 中会成为什么:

```
import interop.*

fun main() {
    println("Hello Kotlin/Native!")

    pass_string(/*fix me*/)
    val useMe = return_string()
    val useMe2 = copy_string(/*fix me*/)
}
```

现在你可以 在 IntelliJ IDEA 中打开项目 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA), 看看如何修正示例项目. 在这个过程中, 我们来看看 C 字符串如何映射为 Kotlin/Native 声明.

字符串在 Kotlin 中的映射结果

通过 IntelliJ IDEA 的 **Go to | Declaration** 的帮助, 或查看编译器错误, 你可以看到为 C 函数生成的 API:

```
fun pass_string(str: CValuesRef<ByteVar /* = ByteVarOf<Byte> */>?)
fun return_string(): CPointer<ByteVar /* = ByteVarOf<Byte> */>?
fun copy_string(str: CValuesRef<ByteVar /* = ByteVarOf<Byte> */>?,
size: Int): Int
```

这些声明看起来很清楚. 所有的 char * 指针类型, 对于参数会转换为 str: CValuesRef<ByteVar>?, 对于返回类型会转换为 CPointer<ByteVar>?. Kotlin 将 char 类型转换为 kotlin.Byte 类型, 因为它通常是 8 bit 有符号值.

在生成的 Kotlin 声明中, 你可以看到 str 表达为 CValuesRef<ByteVar/>?. 这个类型是可为 null 的, 你可以直接传递 Kotlin 的 null 作为参数值.

将 Kotlin 字符串传递到 C

下面来试验在 Kotlin 程序中使用 API. 首先调用 pass_string:

```
fun passStringToC() {
    val str = "this is a Kotlin String"
    pass_string(str.cstr)
}
```

向 C 传递一个 Kotlin 字符串是很简单的, 感谢 Kotlin 的 String.cstr 扩展属性 ("扩展属性 (Extension Property)" in "扩展") 的帮助. 此外还有 String.wcstr, 需要 UTF-16 宽字符的情况可以使用.

在 Kotlin 中读取 C 字符串

下面来接收从 return_string 函数返回的一个 char *, 并将它转换为一个 Kotlin 字符串. 在 Kotlin 中需要编写以下代码:

```
fun passStringToC() {
    val stringFromC = return_string()?.toKString()

    println("Returned from C: $stringFromC")
}
```

上面这段代码使用 toKString() 扩展函数. 请不要与 toString() 函数混淆. Kotlin 中 toKString() 有 2 个重载版本扩展函数:

```
fun CPointer<ByteVar>.toKString(): String
fun CPointer<ShortVar>.toKString(): String
```

第 1 个扩展函数接收一个 char *, 将它作为 UTF-8 字符串, 转换为 Kotlin 字符串. 第 2 个扩展函数对 UTF-16 宽字符串执行同样的操作.

在 Kotlin 接收 C 字符串的字节

下面我们要求一个 C 函数向一个指定的缓冲区写入一个 C 字符串. 函数名为 copy_string. 它接受一个指针参数, 表示字符写入的位置, 以及允许的缓冲区大小参数. 函数返回某个值表示它成功还是失败. 我们假设 0 表示它成功, 并且假设提供的缓冲区足够大:

```
fun sendString() {
    val buf = ByteArray(255)
    buf.usePinned { pinned ->
```

```
        if (copy_string(pinned.addressOf(0), buf.size - 1) != 0) {
            throw Error("Failed to read string from C")
        }
    }

    val copiedStringFromC = buf.decodeToString()
    println("Message from C: $copiedStringFromC")
}
```

首先, 你需要有一个 native 指针传递给 C 函数. 使用 `usePinned` 扩展函数, 临时固定住字节数组的 native 内存地址. C 函数向这个字节数组填充数据. 使用另一个扩展函数 `ByteArray.decodeToString()`, 将字节数组转换为一个 Kotlin `String`, 假设使用 UTF-8 编码.

修正代码

你已经看到了所有的定义, 现在我们来修正代码. 在 IDE 中 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA) 运行 `runDebugExecutableNative` Gradle task, 或使用以下命令来运行代码:

```
./gradlew runDebugExecutableNative
```

最终的 `hello.kt` 文件中的代码大致如下:

```
import interop.*
import kotlinx.cinterop.*

fun main() {
    println("Hello Kotlin/Native!")

    val str = "this is a Kotlin String"
    pass_string(str.cstr)

    val useMe = return_string()?.toKString() ?: error("null pointer
returned")
    println(useMe)

    val copyFromC = ByteArray(255).usePinned { pinned ->
        val useMe2 = copy_string(pinned.addressOf(0),
pinned.get().size - 1)
        if (useMe2 != 0) throw Error("Failed to read string from C")
```

```
        pinned.get().decodeToString()
    }

    println(copyFromC)
}
```

下一步

阅读以下教程, 继续探索更多 C 语言数据类型, 以及它们在 Kotlin/Native 中的表达:

* 映射 C 语言的基本数据类型 (教程 - 映射 C 语言的基本数据类型)
* 映射 C 语言的结构(Struct)和联合(Union)类型 (教程 - 映射 C 语言的结构(Struct)和联合(Union)类型)
* 映射 C 语言的函数指针(Function Pointer) (教程 - 映射 C 语言的函数指针(Function Pointer))

与 C 代码交互 (与 C 代码交互) 文档还讲解了更多的高级使用场景.

教程 - 使用 C Interop 和 libcurl 创建应用程序

最终更新: 2024/09/10

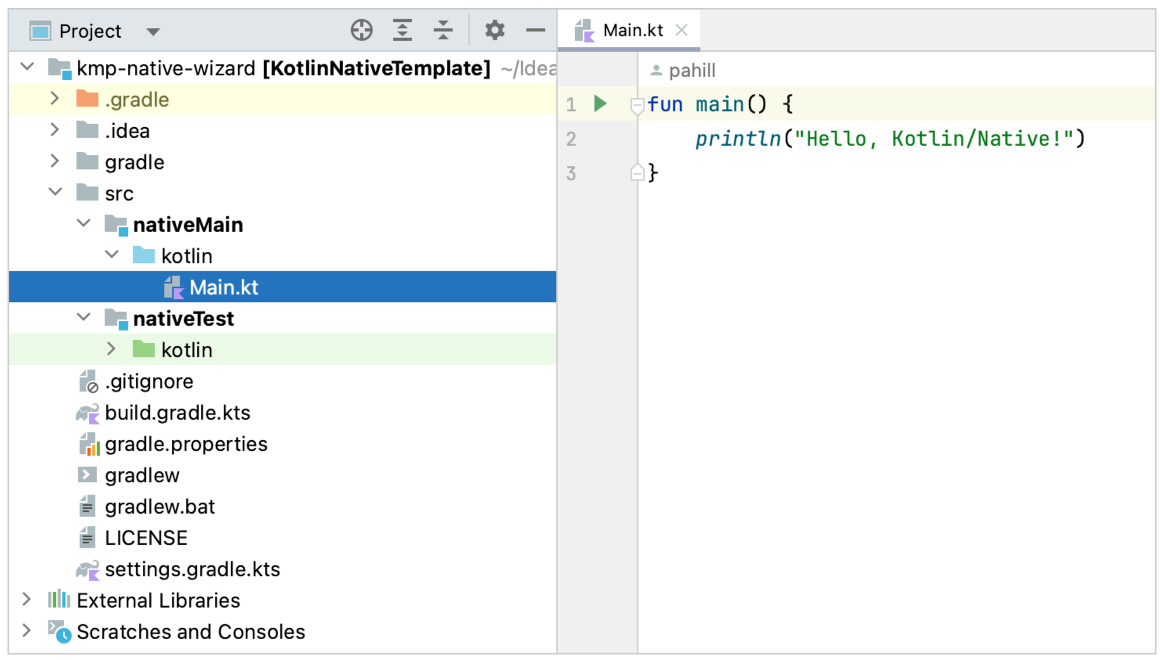
本教程演示如何使用 IntelliJ IDEA 创建一个命令行应用程序. 你将学习如何创建一个简单的 HTTP 客户端程序, 它使用 Kotlin/Native 和 `libcurl` 库, 可以作为原生程序运行在指定的平台上.

输出将是一个可执行的命令行应用程序, 你可以在 macOS 和 Linux 上运行, 发送简单的 HTTP GET 请求.

> 尽管可以直接使用命令行, 或者通过脚本文件(比如 `.sh` 或 `.bat` 文件), 但这种方法不适合于包含几百个文件和库的大项目. 这种情况下, 更好的方法是使用带有构建系统的 Kotlin/Native 编译器, 因为它会帮助你下载并缓存 Kotlin/Native 编译器二进制文件, 传递依赖的库, 并运行编译器和测试. Kotlin/Native 能够通过 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 使用 Gradle (Gradle) 构建系统.

开始前的准备工作

1. 下载并安装最新版本的 IntelliJ IDEA (https://www.jetbrains.com/idea/) 和 Kotlin plugin (Kotlin 的发布版本).
2. 在 IntelliJ IDEA 中选择菜单 **File** | **New** | **Project from Version Control**, 克隆 项目模板 (https://github.com/Kotlin/kmp-native-wizard).
3. 查看项目结构:



Native 应用程序项目结构

模板创建的项目带有你开始工作时所需要的文件和文件夹. 请注意, 如果代码中不包含与特定平台相关的需求, 那么使用 Kotlin/Native 编写的应用程序可以编译到不同的平台. 你的代码放在 `nativeMain` 目录中, 此外还有对应的 `nativeTest` 目录. 在这个教程中, 请不要修改这些文件夹结构.

4. 打开构建脚本文件 `build.gradle.kts`, 其中包含项目的设定. 请特别注意构建脚本文件中的以下内容:

```
kotlin {
    val hostOs = System.getProperty("os.name")
    val isArm64 = System.getProperty("os.arch") == "aarch64"
    val isMingwX64 = hostOs.startsWith("Windows")
    val nativeTarget = when {
        hostOs == "Mac OS X" && isArm64 -> macosArm64("native")
        hostOs == "Mac OS X" && !isArm64 -> macosX64("native")
        hostOs == "Linux" && isArm64 -> linuxArm64("native")
        hostOs == "Linux" && !isArm64 -> linuxX64("native")
        isMingwX64 -> mingwX64("native")
        else -> throw GradleException("Host OS is not supported in
Kotlin/Native.")
    }
```

```
    nativeTarget.apply {
        binaries {
            executable {
                entryPoint = "main"
            }
        }
    }
}
```

* 针对 macOS, Linux, 和 Windows 的编译目标分别通过 macosX64, macosArm64, linuxX64, linuxArm64, 和 mingwX64 定义. 关于所有支持的平台, 请参见 支持的平台 (Kotlin/Native 支持的目标平台).

* 构建脚本定义一系列属性, 指定二进制文件如何生成, 以及应用程序的入口点. 这些可以使用默认值.

* 与 C 的交互使用构建中的一个额外步骤来配置. 默认情况下, 来自 C 的所有符号会被导入到 interop 包. 你可能想要在 .kt 文件中导入整个包. 详情请参见 如何配置 ("编译到多个目标平台" in "配置 Gradle 项目").

创建一个定义文件

编写原生应用程序时, 你经常需要访问没有包含在 Kotlin 标准库 (https://kotlinlang.org/api/latest/jvm/stdlib/) 中的某些功能, 比如发起 HTTP 请求, 读写磁盘, 等等.

Kotlin/Native 帮助你使用 C 的标准库, 使你可以利用 C 的整个生态系统, 其中的功能几乎包含你需要的任何东西. Kotlin/Native 带有一组预构建的 平台库 (平台库), 提供了标准库之外的一些通用功能.

与 C 交互的理想场景是, 象调用 Kotlin 函数一样调用 C 函数, 使用相同的函数签名和规约. 这就是 cinterop 工具可以帮助你的地方. 它输入一个 C 库, 并生成对应的 Kotlin 绑定, 使得库可以象 Kotlin 代码那样使用.

要生成这些绑定, 要创建一个库定义 .def 文件, 包含一些关于需要的头的信息. 在这个应用程序中, 你需要 libcurl 库来发起 HTTP 调用. 创建一个定义文件的步骤如下:

1. 选择 src 文件夹, 使用 **File | New | Directory** 创建一个新目录.

2. 将新目录命名为 **nativeInterop/cinterop**. 这是头文件位置的默认约定, 如果你使用不同的位置, 也可以在 build.gradle.kts 文件中修改这个设置.

3. 选择新建的子文件夹, 使用 **File | New | File** 创建一个新的 `libcurl.def` 文件.

4. 将你的文件内容更新为以下代码:

```
headers = curl/curl.h
headerFilter = curl/*

compilerOpts.linux = -I/usr/include -I/usr/include/x86_64-linux-gnu
linkerOpts.osx = -L/opt/local/lib -L/usr/local/opt/curl/lib -lcurl
linkerOpts.linux = -L/usr/lib/x86_64-linux-gnu -lcurl
```

- `headers` 是需要生成 Kotlin 桩(stub)代码的头文件列表. 你可以在这里添加多个文件, 每个在新行加一个 `\` 来分隔. 在这个示例中, 只有 `curl.h`. 引用的文件路径需要存在于系统路径中(在这个示例中, 是 `/usr/include/curl`).

- `headerFilter` 指定具体包含什么. 在 C 中, 当一个文件使用 `#include` 指令引用另一个文件时, 所有的头文件都会被包含. 有时这些文件是不必要的, 你可以添加这个参数, 使用全局模式 (https://en.wikipedia.org/wiki/Glob_(programming)) 进行细微调节.

  `headerFilter` 是一个可选的参数, 当库作为系统库安装时, 经常使用这个参数. 你不希望获取外部的依赖项 (比如系统的 `stdint.h` 头文件) 到你使用的库中. 对于优化库的大小, 修正系统与 Kotlin/Native 编译环境之间的潜在的冲突, 这个参数可能是很重要的.

- 后面的行是提供链接器(linker) 和编译器的参数, 在不同的目标平台这些参数可能不同. 在这个示例中, 是针对 macOS (`.osx` 后缀) 和 Linux (`.linux` 后缀) 环境的参数. 也可以使用没有后缀的参数(比如, `linkerOpts=`), 会应用到所有的平台.

规约是, 每个库有自己的定义文件, 通常使用与库相同的名称库. 关于 `cinterop` 的所有选项, 详情请参见 与 C 代码交互 (与 C 代码交互).

> 在你的系统需要存在 `curl` 库二进制文件, 才能让示例程序正确工作. 在 macOS 和 Linux 上, 通常会包含这个库. 在 Windows 上, 你可以从 源代码 (https://curl.haxx.se/download.html) 构建它 (你需要 Visual Studio 或 Windows SDK 命令行工具). 详情请参见 相关的 blog (https://jonnyzzz.com/blog/2018/10/29/kn-libcurl-windows/). 或者, 你也可以考虑使用一个 MinGW/MSYS2 (https://www.msys2.org/) 的 `curl` 二进制文件.

向构建过程添加与 C 的交互

要使用头文件, 需要确保在构建过程中生成了它们. 要做到这一点, 请向 `build.gradle.kts` 文件添加以下内容:

```
nativeTarget.apply {
    compilations.getByName("main") {    // NL
        cinterops {                     // NL
            val libcurl by creating     // NL
        }                               // NL
    }                                   // NL
    binaries {
        executable {
            entryPoint = "main"
        }
    }
}
```

新加的行标注了 `// NL`. 首先, 添加 `cinterops`, 然后为每个 `def` 文件添加对应行. 默认情况下, 使用定义文件的名称. 你可以使用额外的参数来修改设定:

```
val libcurl by creating {
    defFile(project.file("src/nativeInterop/cinterop/libcurl.def"))
    packageName("com.jetbrains.handson.http")
    compilerOpts("-I/path")
    includeDirs.allHeaders("path")
}
```

关于可用的选项, 请参见 与 C 代码交互 (与 C 代码交互).

编写应用程序代码

现在你有了库, 以及对应的 Kotlin 桩代码(stub), 可以在你的应用程序中使用它们了. 本教程将 simple.c (https://curl.haxx.se/libcurl/c/simple.html) 示例代码改写为 Kotlin.

在 `src/nativeMain/kotlin/` 文件夹中, 将你的 `Main.kt` 文件更新为以下代码:

```
import kotlinx.cinterop.*
import libcurl.*
```

```
@OptIn(ExperimentalForeignApi::class)
fun main(args: Array<String>) {
    val curl = curl_easy_init()
    if (curl != null) {
        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com")
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L)
        val res = curl_easy_perform(curl)
        if (res != CURLE_OK) {
            println("curl_easy_perform() failed
${curl_easy_strerror(res)?.toKString()}")
        }
        curl_easy_cleanup(curl)
    }
}
```

你可以看到, 在 Kotlin 版本中删除了明确的变量声明, 但其他一切都和 C 版本一样. 在对应的 Kotlin 代码中, 可以使用 `libcurl` 库中所有的调用.

> 这段代码只是逐行的翻译. 你也可以以更加符合 Kotlin 风格的方式编写代码.

编译并运行应用程序

1. 编译应用程序. 方法是, Gradle 运行的 task 中调用 `runDebugExecutableNative`, 或者在终端运行以下命令:

   ```
   ./gradlew runDebugExecutableNative
   ```

   这里, `cinterop` 的生成部分隐含的包含在构建中.

2. 如果编译过程中没有错误, 点击`main()` 方法旁边侧栏中的绿色的 **Run** 图标, 或在 IntelliJ IDEA 中使用 **Alt+Enter** 快捷键启动 launch 菜单.

   IntelliJ IDEA 会打开 **Run** 页面, 并显示输出 — `https://example.com` 的内容:

```
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>
```

应用程序输出的 HTML 代码

你可以看到实际的输出, 因为调用 `curl_easy_perform` 会将结果打印到标准输出. 你可以使用 `curl_easy_setopt` 隐藏这些信息.

> 你可以在 这里 (https://github.com/Kotlin/kotlin-hands-on-intro-kotlin-native) 得到完整的代码.

与 Swift/Objective-C 代码交互

最终更新: 2024/09/10

> Objective-C 库的导入是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). `cinterop` 工具从 Objective-C 库生成的所有 Kotlin 声明都应该标注 `@ExperimentalForeignApi` 注解.
>
> Kotlin/Native 自带的原生平台库 (例如 Foundation, UIKit, 和 POSIX), 只对一部分 API 需要使用者明确同意(Opt-in). 对于这样的情况, 你会在 IDE 中看到警告信息.

本章介绍 Kotlin/Native 与 Swift/Objective-C 的交互能力的一些细节.

关于 iOS 和 Kotlin 之间的内存管理, 详情请参见 与 iOS 集成 (与 iOS 集成).

使用方法

Kotlin/Native 提供了与 Objective-C 的双向交互能力. Objective-C 框架和库可以在 Kotlin 代码中使用, 只需要正确地导入到编译环境中 (系统框架已经默认导入了). 参见 编译配置 ("配置与原生语言的交互" in "配置编译任务"). Swift 库也可以在 Kotlin 代码中使用, 只需要将它的 API 用 `@objc` 导出为 Objective-C. 纯 Swift 模块目前还不支持.

Kotlin 模块可以在 Swift/Objective-C 代码中使用, 只需要编译成一个框架 (参见 如何声明二进制文件 ("声明二进制文件" in "构建最终的原生二进制文件")). 我们提供了一个例子, 请参见 Kotlin Multiplatform Mobile 示例程序 (https://github.com/Kotlin/kmm-basic-sample).

隐藏 Kotlin 声明

如果你不希望将 Kotlin 声明导出到 Objective-C 和 Swift, 请使用专门的注解:

- `@HiddenFromObjC` 注解对 Objective-C 和 Swift 隐藏 Kotlin 声明. 这个注解会禁止一个函数或属性导出到 Objective-C, 让你的 Kotlin 代码对 Objective-C/Swift 更加友好.
- `@ShouldRefineInSwift` 可以将一个 Kotlin 声明替换为 Swift 编写的一个封装(Wrapper). 这个注解会在生成的 Objective-C API 中, 将一个函数或属性标记为 `swift_private`. 这样的声明会带有 `__` 前缀, 使得它们在 Swift 中不可见.

  你仍然可以在 Swift 代码中使用这些声明, 来创建 Swift 友好的 API, 但在 Xcode 的代码自动完成功能中, 不会显示这些声明.

关于如何在 Swift 中润色(Refine) Objective-C 声明, 详情请参见 Apple 官方文档 (https://developer.apple.com/documentation/swift/improving-objective-c-api-declarations-for-swift).

> 使用这些注解需要 使用者同意(Opt-in) (明确要求使用者同意的功能(Opt-in Requirement)).

映射

下表展示了 Kotlin 中的各种概念与 Swift/Objective-C 的对应关系.

"->" 和 "<-" 代表单方向的对应关系.

Kotlin	Swift	Objective-C	注意事项
class	class	@interface	名称翻译
interface	protocol	@protocol	
constructor/create	初始化器(Initializer)	初始化器(Initializer)	初始化器
属性	属性	属性	顶层函数和属性 设值方法(Setter)
方法	方法	方法	顶层函数和属性 方法名称翻译
enum class	class	@interface	枚举类
suspend->	completionHandler:/async	completionHandler:	错误与异常 挂起函数
@Throws fun	throws	error:(NSError**)error	错误与异常
Extension	Extension	Category 成员	扩展与 Category 成员
companion 成员 <-	Class 方法或属性	Class 方法或属性	
null	nil	nil	
Singleton	shared 或 companion 属性	shared 或 companion 属性	Kotlin 单子(singleton)
基本类型	基本类型 / NSNumb		NSNumber

	er		
Unit 类型返回值	Void	void	
String	String	NSString	
String	NSMutableString	NSMutableString	NSMutableString
List	Array	NSArray	
MutableList	NSMutableArray	NSMutableArray	
Set	Set	NSSet	
MutableSet	NSMutableSet	NSMutableSet	集合
Map	Dictionary	NSDictionary	
MutableMap	NSMutableDictionary	NSMutableDictionary	集合
Function 类型	Function 类型	Block pointer 类型	Function 类型
内联类(Inline class)	不支持	不支持	不支持的特性

名称翻译

Objective-C 类导入 Kotlin 时使用它们原来的名称. Protocol 导入 Kotlin 后会变成接口, 并使用 `Protocol` 作为名称后缀, 也就是说 `@protocol Foo` 会被导入为 `interface FooProtocol`. 这些类和接口会放在一个 在编译配置中指定 的包之内 (预定义的系统框架导入到 `platform.*` 包内).

Kotlin 类和接口导入 Objective-C 时会加上名称前缀. 前缀由框架名称决定.

Objective-C 不支持框架内的包. 如果 Kotlin 编译器发现同一个框架内的不同包下存在同名的 Kotlin 类, Kotlin 编译器会对类重命名. 这个算法还未稳定, 在不同的 Kotlin 发布版中可能发生变化.

要绕过这个问题, 你可以将框架内发生名称冲突的 Kotlin 类重命名.

如果要避免对 Kotlin 声明的重新命名, 请使用 `@ObjCName` 注解. 这个注解会指示 Kotlin 编译器对类, 接口, 以及其他 Kotlin 元素使用自定义的 Objective-C 和 Swift 名称:

```
@ObjCName(swiftName = "MySwiftArray")
class MyKotlinArray {
    @ObjCName("index")
    fun indexOf(@ObjCName("of") element: String): Int = TODO()
}

// ObjCName 注解的使用示例
let array = MySwiftArray()
let index = array.index(of: "element")
```

> 使用这个注解需要 使用者同意(Opt-in) (明确要求使用者同意的功能(Opt-in Requirement)).

初始化器(Initializer)

Swift/Objective-C 初始化器导入 Kotlin 时会成为构造器. 对于 Objective-C category 中声明的初始化器, 或声明为 Swift extension 的初始化器, 导入 Kotlin 时会成为名为 `create` 的工厂方法, 因为 Kotlin 没有扩展构造器的概念.

Kotlin 构造器导入 Swift/Objective-C 时会成为初始化器.

设值方法(Setter)

Objective-C 中可写的属性如果覆盖超类中的只读属性, 对于 `foo` 属性会表示为 `setFoo()` 方法. 对于一个协议(protocol)的只读属性, 如果实现为可变的属性, 那么也是同样的规则.

顶层函数和属性

Kotlin 的顶层函数和属性, 可以通过某个特殊类的成员来访问. 每个 Kotlin 源代码文件都会被翻译为一个这样的类. 比如:

```
// MyLibraryUtils.kt
package my.library

fun foo() {}
```

在 Swift 中可以这样调用:

```
MyLibraryUtilsKt.foo()
```

方法名称翻译

通常来说, Swift 的参数标签和 Objective-C 的 selector 会被映射为 Kotlin 的参数名称. 但这两种概念还是存在一些语义上的区别, 因此有时 Swift/Objective-C 方法导入时可能导致 Kotlin 中的签名冲突. 这时, 发生冲突的方法可以在 Kotlin 使用命名参数来调用, 比如:

```
[player moveTo:LEFT byMeters:17]
[player moveTo:UP byInches:42]
```

在 Kotlin 中, 应该这样调用:

```
player.moveTo(LEFT, byMeters = 17)
player.moveTo(UP, byInches = 42)
```

`kotlin.Any` 的方法 (`equals()`, `hashCode()` 和 `toString()`), 在 Objective-C 中被映射为方法 `isEquals:`, `hash` 和 `description`, 在 Swift 被映射为方法 `isEquals(_:)` 和属性 `hash`, `description`.

你可以在 Swift 或 Objective-C 中指定一个更加符合使用习惯的名称, 而不是对 Kotlin 声明自动重命名. 请使用 `@ObjCName` 注解, 指示 Kotlin 编译器对方法或参数使用自定义的 Objective-C 和 Swift 名称.

> 使用这个注解需要 使用者同意(Opt-in) (明确要求使用者同意的功能(Opt-in Requirement)).

错误与异常

Kotlin 中不存在受控异常(Checked Exception)的概念, 所有的 Kotlin 异常都是不受控的. Swift 则只有受控错误. 因此如果 Swift 或 Objective-C 的代码调用一个 Kotlin 方法, 这个方法抛出一个需要被处理的异常, 那么 Kotlin 方法应该使用 `@Throws` 注解, 指明一组 "期待的" 异常类.

编译为 Objective-C/Swift 框架时, 非-`suspend` 的函数如果拥有或继承了 `@Throws` 注解, 在 Objective-C 中会被表示为产生 `NSError*` 的方法, 在 Swift 中会被表示为 `throws` 方法. `suspend` 函数的表达中, 在它的 completion handler 中一定会有 `NSError*`/`Error` 参数.

如果从 Swift/Objective-C 代码调用的 Kotlin 函数中抛出异常, 而且这个异常是 `@Throws` 注解指定的异常类(或其子类)的实例, 那么这个异常会被转换为 `NSError`. 其他 Kotlin 异常到达

Swift/Objective-C 代码后, 会被认为是未处理的错误, 并导致程序终止.

没有 `@Throws` 注解的 `suspend` 函数, 只会把 `CancellationException` 异常变换为 `NSError`. 没有 `@Throws` 注解的非-`suspend` 函数, 则完全不会传播 Kotlin 的异常.

注意, 反过来的翻译目前还未实现: Swift/Objective-C 中抛出 error 的方法, 导入 Kotlin 时不会成为抛出异常的方法.

枚举类

Kotlin 枚举类会被导入为 Objective-C 中的 `@interface`, 以及 Swift 中的 `class`. 这些数据结构拥有与各个枚举值相对应的属性. 对于下面的 Kotlin 代码:

```
// Kotlin
enum class Colors {
    RED, GREEN, BLUE
}
```

在 Swift 中, 你可以这样访问这个枚举类的属性:

```
// Swift
Colors.red
Colors.green
Colors.blue
```

要在 Swift 的 `switch` 语句中使用 Kotlin 枚举类型的变量, 需要提供一个 default 语句, 以避免发生编译错误:

```
switch color {
    case .red: print("It's red")
    case .green: print("It's green")
    case .blue: print("It's blue")
    default: fatalError("No such color")
}
```

挂起函数

从 Swift 代码中 以 `async` 方式调用 `suspend`函数是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望

你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-47610) 提供你的反馈意见.

Kotlin 的 挂起函数 (协程的基本概念) (suspend) 在生成的 Objective-C 头文件中表达为带有回调的函数, 或用 Swift/Objective-C 术语称为 completion handlers (https://developer.apple.com/documentation/swift/calling_objective-c_apis_asynchronously).

从 Swift 5.5 开始, Kotlin 的 suspend 函数也可以从 Swift 代码中以 async 函数的方式调用, 而不需要使用 completion handler. 目前, 这个功能还处于非常初始的实验阶段, 存在很多限制. 详情请参见 这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-47610).

更多详情请参见 关于 async/await 机制的 Swift 文档 (https://docs.swift.org/swift-book/LanguageGuide/Concurrency.html).

扩展与 Category 成员

Objective-C Category 的成员, 以及 Swift extension 的成员, 导入 Kotlin 时通常会变成扩展函数. 因此这些声明在 Kotlin 中不能被覆盖. 另外, extension 初始化器 在 Kotlin 中不会成为类的构造器.

> 目前有两种例外情况. 从 Kotlin 1.8.20 开始, 在 NSView 类 (来自 AppKit 框架) 或 UIView 类 (来自 UIKit 框架) 的相同的头文件中声明的 Category 的成员, 会被导入为这些类的成员. 因此你可以覆盖从 NSView 或 UIView 继承的子类的方法.

对 "通常的" Kotlin 类的 Kotlin 扩展, 导入 Swift 和 Objective-C 后, 分别会成为扩展和 category 成员. 对其他类型的 Kotlin 扩展, 会被当作 顶层声明 处理, 带有额外的接受者参数. 这些类型包括:

- Kotlin String 类型
- Kotlin 集合类型, 及其子类型
- Kotlin interface 类型
- Kotlin 基本类型(primitive type)
- Kotlin inline 类
- Kotlin Any 类型
- Kotlin 函数类型, 及其子类型
- Objective-C 类和协议(protocol)

Kotlin 单子(singleton)

Kotlin 单子(singleton) (通过 `object` 声明产生, 包括 `companion object`) 导入 Swift/Objective-C 会成为一个类, 但它只有唯一一个实例.

这个实例可以通过 `shared` 和 `companion` 属性来访问.

对于下面的 Kotlin 代码:

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

可以通过以下方式访问这些对象:

```
MyObject.shared
MyObject.shared.x
MyClass.companion
MyClass.Companion.shared
```

> 通过 Objective-C 的 `[MySingleton mySingleton]` 和 Swift 的 `MySingleton()` 访问对象, 这个功能已被废弃.

NSNumber

Kotlin 基本类型的装箱类会被映射为 Swift/Objective-C 中的特殊类. 比如, `kotlin.Int` 装箱类在 Swift 中会被表达为 `KotlinInt` 类的实例 (或 Objective-C 中的 `${prefix}Int` 类的实例, 其中 `prefix` 是框架名称前缀). 这些类都继承自 `NSNumber`, 因此它们的实例都是 `NSNumber`, 也支持 `NSNumber` 上的所有的操作.

`NSNumber` 类型用做 Swift/Objective-C 的参数类型或返回值类型时, 不会自动翻译为 Kotlin 的基本类型. 原因是, `NSNumber` 类型没有提供足够的信息, 指明它内部包装的基本值类型是什么, 也

就是说, 通过 `NSNumber` 我们无法知道它究竟是 `Byte`, `Boolean`, 还是 `Double`. 因此 Kotlin 基本类型与 `NSNumber` 类型的相互转换必须手工进行 (详情请参见 下文).

NSMutableString

Objective-C 的 `NSMutableString` 类在 Kotlin 中无法使用. `NSMutableString` 所有实例在传递给 Konlin 之前都会被复制一次.

集合

Kotlin 集合会被转换为 Swift/Objective-C 的集合类型, 对应关系请参见上表. Swift/Objective-C 的集合也会以同样的方式映射为 Kotlin 的集合类型, 但 `NSMutableSet` 和 `NSMutableDictionary` 除外. `NSMutableSet` 不会转换为 Kotlin 的 `MutableSet`. 要创建一个 Kotlin `MutableSet` 类型的对象, 你可以明确地创建这个 Kotlin 集合类型的实例, 要么在 Kotlin 中创建, 比如使用 `mutableSetOf()` 方法, 或者在 Swift 中使用 `KotlinMutableSet` 类创建 (或者在 Objective-C 中使用 `${prefix}MutableSet` 类, 其中 `prefix` 是框架名称前缀). 对于 `MutableMap` 类型也是如此.

Function 类型

Kotlin 的函数类型对象 (比如 Lambda 表达式) 会被转换为 Swift 函数 或 Objective-C 代码段 (block). 但是, 在翻译函数和函数类型时, 对于参数类型和返回值类型的映射方法存在区别. 对于函数类型, 基本类型映射为它们的装箱类. Kotlin 的 `Unit` 返回值类型在 Swift/Objective-C 中会被表达为对应的 `Unit` 单子. 这个单子的值可以象其他任何 Kotlin `object` 一样, 通过相同的方式得到(参见上表中的单子). 综合起来的结果就是:

```
fun foo(block: (Int) -> Unit) { ... }
```

在 Swift 中会成为:

```
func foo(block: (KotlinInt) -> KotlinUnit)
```

调用方法是:

```
foo {
    bar($0 as! Int32)
    return KotlinUnit()
}
```

泛型

Objective-C 支持类上定义的 "轻量的泛型", 支持的功能相对有限. Swift 可以导入类上定义的泛型, 向编译器提供额外的类型信息.

Objective-C 和 Swift 对泛型功能的支持与 Kotlin 不同, 因此翻译过程不可避免的将会丢失部分信息, 但支持的那部分功能还能保留有意义的信息.

功能限制

Objective-C 泛型不支持 Kotlin 或 Swift 的全部特性, 因此在翻译过程中会有一些信息丢失.

泛型只能定义在类上, 而不能用于接口 (也就是 Objective-C 和 Swift 中的协议(protocol)), 也不能用于函数.

可空性(Nullability)

Kotlin 和 Swift 都把可空性(Nullability)的定义作为类型信息的一部分, 而 Objective-C 则在一个类型的方法或属性上定义可空性. 因此, 下面的代码:

```
class Sample<T>() {
    fun myVal(): T
}
```

(逻辑上)将会变成这样:

```
class Sample<T>() {
    fun myVal(): T?
}
```

为了支持可以为 null 的类型, Objective-C 头文件需要将 `myVal` 的返回值定义为可为 null.

为了减轻这个问题, 定义你的泛型类时, 如果泛型类型 *绝对不会* 为 null, 需要提供一个非-null 的类型约束(type constraint):

```
class Sample<T : Any>() {
    fun myVal(): T
}
```

这样将会强制要求 Objective-C 头文件将 `myVal` 标记为非-null.

类型变异(Variance)

Objective-C 允许泛型声明为协变(covariant), 或反向类型变异(contravariant). Swift 不支持类型变异(Variance). 如果需要, 对来自 Objective-C 的泛型类, 可以进行强制类型转换.

```
data class SomeData(val num: Int = 42) : BaseData()
class GenVarOut<out T : Any>(val arg: T)
```

```
let variOut = GenVarOut<SomeData>(arg: sd)
let variOutAny : GenVarOut<BaseData> = variOut as!
GenVarOut<BaseData>
```

类型约束

在 Kotlin 中, 你可以对泛型类型指定上界(Upper Bound). Objective-C 也支持这种功能, 但不能用于更复杂的情况, 而且在 Kotlin - Objective-C 交互中, 目前也不支持. 例外是, 上界(Upper Bound)指定为非-null, 会使得 Objective-C 方法/属性变为非-null.

关闭泛型功能

如要想要框架头文件不使用泛型, 需要在编译器配置中添加以下参数:

```
binaries.framework {
    freeCompilerArgs += "-Xno-objc-generics"
}
```

在映射的类型之间进行变换

编写 Kotlin 代码时, 对象可能需要从 Kotlin 类型转换为等价的 Swift/Objective-C 类型 (或者反过来). 这种情况下, 可以直接使用传统的 Kotlin 类型转换, 比如:

```
val nsArray = listOf(1, 2, 3) as NSArray
val string = nsString as String
val nsNumber = 42 as NSNumber
```

类继承

在 Swift/Objective-C 中继承 Kotlin 类和接口

Swift/Objective-C 类和 protocol 可以继承 Kotlin 类和接口.

在 Kotlin 中继承 Swift/Objective-C 类和接口

Kotlin 的 `final` class 可以继承 Swift/Objective-C 类和 protocol. 目前还不支持非 `final` 的 Kotlin 类继承 Swift/Objective-C 类型, 因此不可能声明一个复杂的类层级, 同时又继承

Swift/Objective-C 类型.

可以使用 Kotlin 的 `override` 关键字来覆盖通常的方法. 这种情况下, 子类方法的参数名称, 必须与被覆盖的方法相同.

有时我们会需要覆盖初始化器, 比如, 继承 `UIViewController` 时. 初始化器会被导入成为 Kotlin 中的构造器, 它可以被 Kotlin 中使用了 `@OverrideInit` 注解的构造器覆盖:

```
class ViewController : UIViewController {
    @OverrideInit constructor(coder: NSCoder) : super(coder)

    ...
}
```

子类构造器的参数名称和类型, 必须与被覆盖的构造器相同.

如果多个方法在 Kotlin 中发生了签名冲突, 要覆盖这些方法, 你可以在类上添加 `@Suppress("CONFLICTING_OVERLOADS")` 注解.

压制 Kotlin 签名冲突错误, 是一种临时的替代方法. 这样的情况下不能保证稳定性, 因此要小心使用. 我们将会在未来的 Kotlin 发布版本中解决这个问题.

Kotlin/Native 默认不会允许通过 `super(...)` 构造器来调用 Objective-C 的非指定(non-designated)初始化器. 如果在 Objective-C 库中没有正确地标注出指定的(designated)初始化器, 那么这种限制可能会造成我们的不便. 可以在这个库的 `.def` 文件中添加一个 `disableDesignatedInitializerChecks = true` 设定, 来关闭编译器的这个检查.

C 语言功能

请参见 与 C 代码交互 (与 C 代码交互), 其中有一些示例程序, 其中的库使用了某些 C 语言功能, 比如, 不安全的指针, 结构(struct), 等等.

将 KDoc 注释导出到生成的 Objective-C 头文件

KDoc 注释导出到生成的 Objective-C 头文件是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文), 而且你应该只为评估目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-38600) 提供你的反馈意见.

默认情况下, 在生成 Objective-C 头文件时, KDocs (为 Kotlin 代码编写文档: KDoc) 文档注释不会被翻译为头文件中对应的注释. 例如, 以下带 KDoc 文档的 Kotlin 代码:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit
integer.
 */
fun printSum(a: Int, b: Int) = println(a.toLong() + b)
```

会生成 Objective-C 声明, 没有任何注释:

```
+ (void)printSumA:(int32_t)a b:(int32_t)b
__attribute__((swift_name("printSum(a:b:)")));
```

要启用 KDoc 注释导出功能, 请在你的 `build.gradle(.kts)` 添加以下编译器选项:

Kotlin

```
kotlin {

targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNa
tiveTarget> {

compilations.get("main").compilerOptions.options.freeCompilerArg
s.add("-Xexport-kdoc")
    }
}
```

Groovy

```
kotlin {

targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNa
tiveTarget) {

compilations.get("main").compilerOptions.options.freeCompilerArg
```

```
s.add("-Xexport-kdoc")
    }
}
```

这样设置之后, Objective-C 头文件将包含对应的注释:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit
integer.
 */
+ (void)printSumA:(int32_t)a b:(int32_t)b
__attribute__((swift_name("printSum(a:b:)")));
```

已知的限制:

- 依赖项的文档不会导出, 除非它本身也使用 `-Xexport-kdoc` 选项来编译. 这个功能还是实验性功能, 因此使用这个选项编译的库可能与其他编译器版本不兼容.
- 绝大多数 KDoc 注释会 "保持原状" 导出. 很多 KDoc 功能(例如, `@property`)不支持.

不支持的特性

Kotlin 编程语言的一些特性目前还没有映射为 Objective-C 或 Swift 中对应的特性. 目前, 在生成的框架头文件中, 以下特性还不能正确地导出:

- 内联类(inline class) (参数会被映射为底层的基本类型, 或 `id`)
- 实现标准的 Kotlin 集合接口 (`List`, `Map`, `Set`) 的自定义类, 以及其他特殊的类
- Objective-C 类的 Kotlin 子类

教程 - 使用 Kotlin/Native 开发 Apple Framework

最终更新: 2024/09/10

> Objective-C 库导入 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). `cinterop` 工具从 Objective-C 库生成的所有 Kotlin 声明都应该标注 `@ExperimentalForeignApi` 注解.
>
> Kotlin/Native 自带的原生平台库 (例如 Foundation, UIKit, 和 POSIX), 只对一部分 API 需要使用者明确同意(Opt-in). 对于这样的情况, 你会在 IDE 中看到警告信息.

Kotlin/Native 提供了与 Objective-C/Swift 的双向交互能力. Objective-C Framework 和库可以在 Kotlin 代码中使用. Kotlin 模块也可以在 Swift/Objective-C 代码中使用. 此外, Kotlin/Native 还有 与 C 代码交互 (与 C 代码交互) 功能. 还可以参考教程 使用 Kotlin/Native 开发动态库 (教程 - 使用 Kotlin/Native 开发动态库).

在本教程中, 你将会看到在 macOS 和 iOS 的 Objective-C 和 Swift 应用程序中如何使用 Kotlin/Native 代码.

在本教程中, 你将会:

- 创建一个 Kotlin 库 并将它编译为一个 Framework
- 查看生成的 Objective-C 和 Swift API 代码
- 在 Objective-C 和 Swift 中使用 Framework
- 配置 Xcode 在 macOS 和 iOS 上使用 Framework

创建一个 Kotlin 库

Kotlin/Native 编译器可以从 Kotlin 代码生成 macOS 和 iOS 的 Framework. 创建的 Framework 包含在 Objective-C 和 Swift 中使用它所需要的所有声明和二进制文件. 理解这些技术的最好方法就是来试用一下它们. 首先我们创建一个小小的 Kotlin 库, 然后在一个 Objective-C 程序中使用它.

创建 `hello.kt` 文件, 包含库的内容:

```
package example

object Object {
    val field = "A"
}

interface Interface {
    fun iMember() {}
}

class Clazz : Interface {
    fun member(p: Int): ULong? = 42UL
}

fun forIntegers(b: Byte, s: UShort, i: Int, l: ULong?) { }
fun forFloats(f: Float, d: Double?) { }

fun strings(str: String?) : String {
    return "That is '$str' from C"
}

fun acceptFun(f: (String) -> String?) = f("Kotlin/Native rocks!")
fun supplyFun() : (String) -> String? = { "$it is cool!" }
```

尽管可以直接使用命令行, 或者通过脚本文件(比如 `.sh` 或 `.bat` 文件), 但这种方法不适合于包含几百个文件和库的大项目. 更好的方法是使用带有构建系统的 Kotlin/Native 编译器, 因为它会帮助你下载并缓存 Kotlin/Native 编译器二进制文件, 传递依赖的库, 并运行编译器和测试. Kotlin/Native 能够通过 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 使用 Gradle (https://gradle.org) 构建系统.

关于如何使用 Gradle 设置 IDE 兼容的项目, 请参见教程 一个基本的 Kotlin/Native 应用程序 (Kotlin/Native 开发入门 - 使用 Gradle). 如果你想要寻找具体的步骤指南, 来开始一个新的 Kotlin/Native 项目并在 IntelliJ IDEA 中打开它, 请先阅读这篇教程. 在本教程中, 我们关注更高级的 C 交互功能, 包括使用 Kotlin/Native, 以及使用 Gradle 的 跨平台 ("编译到多个目标平台" in "配置 Gradle 项目") 构建.

首先, 创建一个项目文件夹. 本教程中的所有路径都是基于这个文件夹的相对路径. 有时在添加任何新文件之前, 会需要创建缺少的目录.

使用以下 `build.gradle(.kts)` Gradle 构建文件:

Kotlin

```kotlin
plugins {
    kotlin("multiplatform") version "1.9.23"
}

repositories {
    mavenCentral()
}

kotlin {
    macosX64("native") {
        binaries {
            framework {
                baseName = "Demo"
            }
        }
    }
}

tasks.wrapper {
    gradleVersion = "8.1.1"
    distributionType = Wrapper.DistributionType.ALL
}
```

Groovy

```groovy
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}

repositories {
    mavenCentral()
}

```

```
kotlin {
    macosX64("native") {
        binaries {
            framework {
                baseName = "Demo"
            }
        }
    }
}

wrapper {
    gradleVersion = "8.1.1"
    distributionType = "ALL"
}
```

将源代码文件移动到项目的 `src/nativeMain/kotlin` 文件夹内. 这是使用 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 时的默认源代码路径. 使用以下代码块来配置项目, 生成一个动态库或共用库:

```
binaries {
    framework {
        baseName = "Demo"
    }
}
```

除 macOS `X64` 之外, Kotlin/Native 还支持 macos `arm64` 和 iOS `arm32`, `arm64` 以及 `X64` 编译目标. 你可以将 `macosX64` 替换为下表中对应的函数:

编译目标平台/设备	Gradle 函数
macOS x86_64	macosX64()
macOS ARM 64	macosArm64()
iOS ARM 64	iosArm64()
iOS Simulator (x86_64)	iosX64()
iOS Simulator (arm64)	iosSimulatorArm64

可以 在 IDE 中 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA) 运行 linkNative Gradle task 来构建库, 或执行以下控制台命令:

```
./gradlew linkNative
```

根据构建变体不同, 构建会 在 build/bin/native/debugFramework 和 build/bin/native/releaseFramework 文件夹生成 Framework. 我们来看到其中的内容.

生成的 Framework 头文件

创建的每个 Framework 在 <Framework>/Headers/Demo.h 中包含头文件. 头文件不依赖于编译目标平台 (至少 Kotlin/Native v.0.9.2 如此). 它包含我们的 Kotlin 代码的定义, 以及少量 Kotlin 全局声明.

Kotlin/Native 导出符号的方式可能会发生变化, 不另行通知.

Kotlin/Native 运行期声明

我们来看看 Kotlin 运行期声明:

```
NS_ASSUME_NONNULL_BEGIN

@interface KotlinBase : NSObject
- (instancetype)init __attribute__((unavailable));
+ (instancetype)new __attribute__((unavailable));
```

```
+ (void)initialize __attribute__((objc_requires_super));
@end;

@interface KotlinBase (KotlinBaseCopying) <NSCopying>
@end;

__attribute__((objc_runtime_name("KotlinMutableSet")))
__attribute__((swift_name("KotlinMutableSet")))
@interface DemoMutableSet<ObjectType> : NSMutableSet<ObjectType>
@end;

__attribute__((objc_runtime_name("KotlinMutableDictionary")))
__attribute__((swift_name("KotlinMutableDictionary")))
@interface DemoMutableDictionary<KeyType, ObjectType> :
NSMutableDictionary<KeyType, ObjectType>
@end;

@interface NSError (NSErrorKotlinException)
@property (readonly) id _Nullable kotlinException;
@end;
```

Kotlin 类在 Objective-C 中的基类是 `KotlinBase`, 这个类继承 `NSObject` 类. 还有一些对集合和异常的封装. 大多数集合类型映射为 Objective-C/Swift 中类似的集合类型:

Kotlin	Swift	Objective-C
List	Array	NSArray
MutableList	NSMutableArray	NSMutableArray
Set	Set	NSSet
Map	Dictionary	NSDictionary
MutableMap	NSMutableDictionary	NSMutableDictionary

Kotlin 数值类型与 NSNumber

<Framework>/Headers/Demo.h 的下一部分包含 Kotlin/Native 数值类型与 NSNumber 之间的映射. 在 Objective-C 中有一个名为 DemoNumber 的基类, 在 Swift 中是 KotlinNumber. 它继承 NSNumber. 对每个 Kotlin 数值类型也有各自的子类:

Kotlin	Swift	Objective-C	简单类型
-	KotlinNumber	<Package>Number	-
Byte	KotlinByte	<Package>Byte	char
UByte	KotlinUByte	<Package>UByte	unsigned char
Short	KotlinShort	<Package>Short	short
UShort	KotlinUShort	<Package>UShort	unsigned short
Int	KotlinInt	<Package>Int	int
UInt	KotlinUInt	<Package>UInt	unsigned int
Long	KotlinLong	<Package>Long	long long
ULong	KotlinULong	<Package>ULong	unsigned long long
Float	KotlinFloat	<Package>Float	float
Double	KotlinDouble	<Package>Double	double
Boolean	KotlinBoolean	<Package>Boolean	BOOL/Bool

每个数值类型有一个类方法, 可以从相关的简单类型创建一个新实例. 还有一个实例方法, 反过来抽取一个简单类型的值. 声明大致如下:

```
__attribute__((objc_runtime_name("Kotlin__TYPE__")))
__attribute__((swift_name("Kotlin__TYPE__")))
```

```
@interface Demo__TYPE__ : DemoNumber
- (instancetype)initWith__TYPE__:(__CTYPE__)value;
+ (instancetype)numberWith__TYPE__:(__CTYPE__)value;
@end;
```

其中 `__TYPE__` 是简单类型名称之一, `__CTYPE__` 是相关的 Objective-C 类型, 比如, `initWithChar(char)`.

这些类型用来将装箱的(boxed) Kotlin 数值类型映射到 Objective-C 和 Swift. 在 Swift 中, 你可以简单的调用构造器来创建一个实例, 比如, `KotlinLong(value: 42)`.

Kotlin 的类和对象

我们来看到 `class` 和 `object` 如何映射到 Objective-C 和 Swift. 生成的 `<Framework>/Headers/Demo.h` 文件包含 `Class`, `Interface`, 和 `Object` 的明确定义:

```
NS_ASSUME_NONNULL_BEGIN

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Object")))
@interface DemoObject : KotlinBase
+ (instancetype)alloc __attribute__((unavailable));
+ (instancetype)allocWithZone:(struct _NSZone *)zone
__attribute__((unavailable));
+ (instancetype)object __attribute__((swift_name("init()")));
@property (readonly) NSString *field;
@end;

__attribute__((swift_name("Interface")))
@protocol DemoInterface
@required
- (void)iMember __attribute__((swift_name("iMember()")));
@end;

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Clazz")))
@interface DemoClazz : KotlinBase <DemoInterface>
- (instancetype)init __attribute__((swift_name("init()")))
__attribute__((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable,
```

```
message="use object initializers instead")));
- (DemoLong * _Nullable)memberP:(int32_t)p
__attribute__((swift_name("member(p:)")));
@end;
```

代码充满了 Objective-C 的属性(attribute), 目的是帮助你在 Objective-C 和 Swift 两种语言中使用 Framework. `DemoClazz`, `DemoInterface`, 和 `DemoObject` 分别对应于 `Clazz`, `Interface`, 和 `Object`. `Interface` 被转换为 `@protocol`, 一个 `class` 和 一个 `object` 都表示为 `@interface`. `Demo` 前缀来自 `kotlinc-native` 编译器的 `-output` 参数, 以及 Framework 名称. 你可以看到可为 null 的返回 类型 `ULong?` 在 Objective-C 中转换为 `DemoLong*`.

Kotlin 的全局声明

Kotlin 的所有全局函数, 在 Objective-C 中转换为 `DemoLibKt`, 在 Swift 中转换为 `LibKt`, 这里 `Demo` 是 Framework 名称, 由 `kotlinc-native` 的 `-output` 参数指定.

```
NS_ASSUME_NONNULL_BEGIN

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("LibKt")))
@interface DemoLibKt : KotlinBase
+ (void)forIntegersB:(int8_t)b s:(int16_t)s i:(int32_t)i l:(DemoLong
* _Nullable)l __attribute__((swift_name("forIntegers(b:s:i:l:)")));
+ (void)forFloatsF:(float)f d:(DemoDouble * _Nullable)d
__attribute__((swift_name("forFloats(f:d:)")));
+ (NSString *)stringsStr:(NSString * _Nullable)str
__attribute__((swift_name("strings(str:)")));
+ (NSString * _Nullable)acceptFunF:(NSString * _Nullable (^)
(NSString *))f __attribute__((swift_name("acceptFun(f:)")));
+ (NSString * _Nullable (^)(NSString *))supplyFun
__attribute__((swift_name("supplyFun()")));
@end;
```

你可以看到 Kotlin `String` 直接映射为 Objective-C `NSString*`. 类似的, Kotlin 的`Unit` 类型映射为 `void`. 我们看到基本类型会直接映射. 不可为 null 的基本类型直接相互映射. 可为 null 的基本类型映射为 `Kotlin<TYPE>*` 类型, 如 上表 所述. 高阶函数 `acceptFunF` 和 `supplyFun` 都被包含了, 并接受 Objective-C 代码块.

关于所有其它类型的映射, 详情请参见文档 与 Swift/Objective-C 代码交互 (与 Swift/Objective-

C 代码交互).

垃圾收集与引用计数

Objective-C 和 Swift 使用引用计数. Kotlin/Native 也有自己的垃圾收集功能. Kotlin/Native 垃圾收集 会与 Objective-C/Swift 的引用计数集成. 在 Swift 或 Objective-C 中, 你不需要执行任何特殊操作来控制 Kotlin/Native 实例的生命周期.

在 Objective-C 中使用代码

我们来在 Objective-C 中调用 Framework. 要实现这个目的, 创建 `main.m` 文件, 内容如下:

```
#import <Foundation/Foundation.h>
#import <Demo/Demo.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        [[DemoObject object] field];

        DemoClazz* clazz = [[ DemoClazz alloc] init];
        [clazz memberP:42];

        [DemoLibKt forIntegersB:1 s:1 i:3 l:[DemoULong
numberWithUnsignedLongLong:4]];
        [DemoLibKt forIntegersB:1 s:1 i:3 l:nil];

        [DemoLibKt forFloatsF:2.71 d:[DemoDouble
numberWithDouble:2.71]];
        [DemoLibKt forFloatsF:2.71 d:nil];

        NSString* ret = [DemoLibKt acceptFunF:^NSString *
_Nullable(NSString * it) {
            return [it stringByAppendingString:@" Kotlin is fun"];
        }];

        NSLog(@"%@", ret);
        return 0;
```

```
    }
}
```

这里你在 Objective-C 代码中直接调用 Kotlin 类. Kotlin object 有类方法函数 object, 我们可以用它来得到唯一对象的实例, 并对它调用 Object 方法. widespread 模式用来创建 Clazz 类的一个实例. 在 Objective-C 中你调用 [[DemoClazz alloc] init]. 对于没有参数的构造器, 你也可以使用 [DemoClazz new]. 在 Objective-C 中, Kotlin 源代码的全局声明封装在 DemoLibKt 类内. 所有方法转换为这个类中的类方法. strings 函数转换为 Objective-C 中的 DemoLibKt.stringsStr 函数, 你可以直接传递 NSString 参数. 返回值类型也是 NSString.

在 Swift 中使用代码

你使用 Kotlin/Native 编译的 Framework 有一些帮助属性(attribute), 以方便在 Swift 中使用. 我们将前面的 Objective-C 示例转换为 Swift. 结果你将在 main.swift 中得到以下代码:

```
import Foundation
import Demo

let kotlinObject = Object()
assert(kotlinObject === Object(), "Kotlin object has only one
instance")

let field = Object().field

let clazz = Clazz()
clazz.member(p: 42)

LibKt.forIntegers(b: 1, s: 2, i: 3, l: 4)
LibKt.forFloats(f: 2.71, d: nil)

let ret = LibKt.acceptFun { "\($0) Kotlin is fun" }
if (ret != nil) {
    print(ret!)
}
```

Kotlin 代码在 Swift 中转换为非常类似的代码. 但存在很少的区别. 在 Kotlin 中, 所有的 object 都只有一个 实例. Kotlin 的 object Object 在 Swift 中则有了一个构造器, 而且我们使用 Object() 语法来访问它的唯一实例. 在 Swift 中这个实例永远是同一个, 因此 Object() === Object() 为 true. 方

法和属性转换为相同的名称. Kotlin 的 `String` 也转换为 Swift 的 `String`. Swift 也对我们隐藏 `NSNumber*` 的装箱(boxing). 我们可以向 Kotlin 传递一个 Swift 的闭包(closure), 也可以在 Swift 中调用一个 Kotlin 的 Lambda 函数.

关于类型映射, 更多详情请参见文档 与 Swift/Objective-C 代码交互 (与 Swift/Objective-C 代码交互).

Xcode 与 Framework 依赖项

你需要配置 Xcode 项目来使用我们的 Framework. 具体的配置依赖于编译目标平台.

针对 macOS 编译目标的 Xcode 配置

首先, 在 **target** 配置的 **General** 页中, 在 **Linked Frameworks and Libraries** 节中, 你需要包含我们的 Framework. 这个设置将会让 Xcode 查找我们的 Framework, 并对 Objective-C 和 Swift 解析 import.

第 2 步是配置输出的二进制文件的 Framework 查找路径. 也称为 `rpath` 或 运行时查找路径 (https://en.wikipedia.org/wiki/Rpath). 二进制文件使用这个路径来查找需要的 Framework. 如果没有必要, 我们不推荐在 OS 上安装额外的 Framework. 你应该知道你未来的应用程序的文件构成, 比如, 你可能在应用程序 bundle 中有 `Frameworks` 文件夹, 包含你使用的所有 Framework. 在 Xcode 中可以配置 `@rpath` 参数. 你需要打开 **project** 配置, 找到 **Runpath Search Paths** 节. 在这里你可以指定编译后的 Framework 的相对路径.

针对 iOS 编译目标的 Xcode 配置

首先, 你需要在 Xcode 项目中包含编译后的 Framework. 方法是, 在 **target** 配置页的 **General** 页的 **Frameworks, Libraries, and Embedded Content** 节, 添加 Framework.

第 2 步是, 在 **target** 配置页的 **Build Settings** 页 的 **Framework Search Paths** 节, 包含 Framework 路径. 可以使用宏 `$(PROJECT_DIR)` 来简化设置.

iOS 模拟器要求 Framework 编译到 `ios_x64` 编译目标, 在我们的例子中是 `iOS_sim` 文件夹.

这个 Stackoverflow 讨论串 (https://stackoverflow.com/questions/30963294/creating-ios-osx-frameworks-is-it-necessary-to-codesign-them-before-distributin) 包含其他一些建议. 此外 CocoaPods (https://cocoapods.org/) 包管理器也可以帮助你自动化这个依赖配置过程.

下一步做什么?

Kotlin/Native 支持与 Objective-C 和 Swift 语言的双向交互. Kotlin 对象与 Objective-C/Swift 的引用计数集成. 未使用的 Kotlin 对象会被自动删除. 与 Swift/Objective-C 代码交互 (与

Swift/Objective-C 代码交互) 文档介绍了交互的更多实现细节. 当然, 可以导入一个既有的 Framework 并在 Kotlin 中使用它. Kotlin/Native 带有很多预导入的系统 Framework.

Kotlin/Native 还支持与 C 语言交互. 详情请参见教程 使用 Kotlin/Native 开发动态库 (教程 - 使用 Kotlin/Native 开发动态库).

CocoaPods 概述与设置

最终更新: 2024/09/10

Kotlin/Native 提供了与 CocoaPods 依赖管理器 (https://cocoapods.org/) 的集成功能. 你可以添加对 Pod 库的依赖项, 也可以使用跨平台项目的原生编译目标作为 CocoaPods 依赖项.

可以直接在 IntelliJ IDEA 中管理 Pod 依赖项, 并使用所有额外的功能特性, 比如代码高亮度和代码自动完成. 可以使用 Gradle 来构建整个 Kotlin 项目, 而不必切换到 Xcode.

只有在需要编写 Swift/Objective-C 代码, 或在模拟器或设备上运行应用程序时, 才需要使用 Xcode. 要与 Xcode 正确的协同工作, 你需要 更新你的 Podfile.

根据你的项目和目的不同, 可以添加 Kotlin 项目对 Pod 库的依赖项 (添加 Pod 库依赖项), 以及 Kotlin Gradle 项目对 Xcode 项目的依赖项 (将 Kotlin Gradle 项目用作 CocoaPods 依赖项).

设置 CocoaPods 环境

使用你选择的安装工具, 安装 CocoaPods 依赖项管理器 (https://cocoapods.org/):

RVM

1. 如果你还没有, 请先安装 RVM (Ruby 版本管理器) (https://rvm.io/rvm/install).
2. 安装 Ruby. 你可以选择特定的版本:

   ```
   rvm install ruby 3.0.0
   ```

3. 安装 CocoaPods:

   ```
   sudo gem install -n /usr/local/bin cocoapods
   ```

Rbenv

1. 如果你还没有, 请先从 GitHub 安装 rbenv (https://github.com/rbenv/rbenv#installation).
2. 安装 Ruby. 你可以选择特定的版本:

```
rbenv install 3.0.0
```

3. 对某个目录设置局部的 Ruby 版本, 或对整个机器设置全局的 Ruby 版本:

```
rbenv global 3.0.0
```

4. 安装 CocoaPods:

```
sudo gem install -n /usr/local/bin cocoapods
```

默认的 Ruby

这种安装方法不能用于使用 Apple M 芯片的设备. 请使用其他工具来设置 CocoaPods 工作环境.

你可以使用 macOS 上默认的 Ruby 来安装 CocoaPods 依赖管理器:

```
sudo gem install cocoapods
```

Homebrew

使用 Homebrew 安装 CocoaPods 可能出现兼容性问题.

在安装 CocoaPods 时, Homebrew 也会安装与 Xcode 联合工作时所需要的 Xcodeproj (https://github.com/CocoaPods/Xcodeproj) gem. 但是, 它不能通过 Homebrew 来更新, 而且, 如果安装的 Xcodeproj 还不支持最新的 Xcode 版本, 那么你会在安装 Pod 时出现错误. 如果发生这样的情况, 请试用其他工具来安装 CocoaPods.

1. 如果你还没有, 请先安装 Homebrew (https://brew.sh/).

2. 安装 CocoaPods:

```
brew install cocoapods
```

如果你使用 Kotlin 1.7.0 以前的版本

如果你目前的 Kotlin 版本低于 1.7.0, 那么还需要安装 `cocoapods-generate` (https://github.com/square/cocoapods-generate) plugin:

`sudo gem install -n /usr/local/bin cocoapods-generate`

请注意, `cocoapods-generate` 不能安装在 Ruby 3.0.0 或更高版本上. 如果你使用的是 Ruby 3.0.0 或更高版本, 请降级 Ruby, 或将 Kotlin 升级到 1.7.0 或更高版本.

如果你在安装过程中遇到问题, 请参见 可能发生的问题与解决方案 小节.

添加并配置 Kotlin CocoaPods Gradle plugin

如果你的环境已经正确设置, 你可以 创建一个新的 Kotlin Multiplatform 项目 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-create-first-app.html), 并在 iOS framework distribution 选项中, 选择 **CocoaPods Dependency Manager**. 插件会为你自动生成项目.

如果想要手动配置你的项目:

1. 在你的项目的 `build.gradle(.kts)` 文件中, 应用 CocoaPods 插件和 Kotlin Multiplatform 插件:

```
plugins {
    kotlin("multiplatform") version "1.9.23"
    kotlin("native.cocoapods") version "1.9.23"
}
```

2. 在 `cocoapods` 代码段中, 配置 Podspec 文件的 `version`, `summary`, `homepage`, 和 `baseName`.

```
plugins {
    kotlin("multiplatform") version "1.9.23"
    kotlin("native.cocoapods") version "1.9.23"
}

kotlin {
    cocoapods {
        // 必须属性
```

```
        // 在这里指定需要的 Pod 版本. 否则, 会使用 Gradle 项目的版本.
        version = "1.0"
        summary = "Some description for a Kotlin/Native module"
        homepage = "Link to a Kotlin/Native module homepage"

        // 可选属性
        // 在这里配置 Pod 名称, 而不是修改 Gradle 项目名称
        name = "MyCocoaPod"

        framework {
            // 必须属性
            // 配置框架名称. 'frameworkName' 属性已废弃, 请改为使用这个
属性
            baseName = "MyFramework"

            // 可选属性
            // 指定框架的链接类型. 默认为 dynamic.
            isStatic = false
            // 导出依赖项
            export(project(":anotherKMMModule"))
            transitiveExport = false // 这是默认值.
            // 嵌入 bitcode
            embedBitcode(BITCODE)
        }

        // 将自定义的 Xcode 配置对应到 NativeBuildType
        xcodeConfigurationToNativeBuildType["CUSTOM_DEBUG"] =
NativeBuildType.DEBUG
        xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] =
NativeBuildType.RELEASE
    }
}
```

Kotlin DSL 的完整语法请参见 Kotlin Gradle plugin 代码仓库 (https://github.com/JetBrains/kotlin/blob/master/libraries/tools/kotlin-gradle-plugin/src/common/kotlin/org/jetbrains/kotlin/gradle/targets/native/cocoapods/CocoapodsExtension.kt).

3. 重新导入项目.

4. 生成 Gradle wrapper (https://docs.gradle.org/current/userguide/gradle_wrapper.html), 以免在 Xcode 构建时发生兼容性问题.

应用 CocoaPods 插件后, 它会完成如下工作:

- 对所有的 macOS, iOS, tvOS, 和 watchOS 编译目标, 将 `debug` 和 `release` 框架添加为输出的二进制文件.
- 创建 `podspec` 任务, 它会为项目生成一个 Podspec (https://guides.cocoapods.org/syntax/podspec.html) 文件.

`Podspec` 文件包含输出框架的路径, 以及一段脚本, 负责在 Xcode 项目的构建过程中, 自动构建这个框架.

为 Xcode 更新 Podfile 文件

如果要在一个 Xcode 项目中导入你的 Kotlin 项目, 需要修改你的 Podfile 文件:

- 如果你的项目存在任何 Git, HTTP, 或 自定义 Podspec 仓库的依赖项, 那么还需要在 Podfile 中指定 Podspec 的路径.

  例如, 如果添加了 `podspecWithFilesExample` 的依赖, 需要在 Podfile 文件中声明 Podspec 路径:

  ```
  target 'ios-app' do
      # ... 其他依赖项 ...
      pod 'podspecWithFilesExample', :path =>
  'cocoapods/externalSources/url/podspecWithFilesExample'
  end
  ```

  `:path` 应该包含 Pod 的文件路径.

- 如果添加一个来自自定义 Podspec 仓库的库, 那么还需要在 Podfile 文件的最开始指定 spec 的位置 (https://guides.cocoapods.org/syntax/podfile.html#source):

  ```
  source 'https://github.com/Kotlin/kotlin-cocoapods-spec.git'

  target 'kotlin-cocoapods-xcproj' do
  ```

```
    # ... 其他依赖项 ...
    pod 'example'
end
```

> 对 Podfile 文件进行这些修改后, 需要重新导入项目.

如果不对 Podfile 文件进行这些修改, podInstall 任务将会失败, CocoaPods plugin 会在 log 中显示错误消息.

可能发生的问题与解决方案

CocoaPods 安装

Ruby 安装

CocoaPods 是基于 Ruby 开发的, 你可以使用 macOS 上默认可用的 Ruby 环境来安装它. Ruby 1.9 或更高版本带有一个内建的 RubyGems 包管理框架, 可以帮助你安装 CocoaPods 依赖管理器 (https://guides.cocoapods.org/using/getting-started.html#installation).

如果你在安装或使用 CocoaPods 时遇到问题, 请参照 这篇向导文档 (https://www.ruby-lang.org/en/documentation/installation/) 来安装 Ruby, 或参照 RubyGems 网站 (https://rubygems.org/pages/download/) 来安装 RubyGems 框架.

版本兼容性

我们推荐使用最新的 Kotlin 版本. 如果你目前的版本低于 1.7.0, 你还需要安装 cocoapods-generate (https://github.com/square/cocoapods-generate#installation") 插件.

但是, cocoapods-generate 不兼容 Ruby 3.0.0 或更高版本. 这种情况下, 请降级 Ruby, 或升级 Kotlin 到 1.7.0 或更高版本.

找不到模块

你可能遇到 module 'SomeSDK' not found 错误, 这是 与 C 代码交互 (与 C 代码交互) 相关的问题. 请使用以下变通方法解决这个错误:

指定框架名称

1. 在下载的 Pod 目录 [shared_module_name]/build/cocoapods/synthetic/IOS/Pods/... 中找到 module.modulemap 文件:

2. 检查模块内的框架名称, 比如 `AppsFlyerLib {}`. 如果框架名称与 Pod 名称不匹配, 请明确指定它:

```
pod("FirebaseAuth") {
    moduleName = "AppsFlyerLib"
}
```

指定头文件

如果在生成的 `.def` 文件中 Pod 没有包含 `.modulemap` 文件, 比如 `pod("NearbyMessages")`, 请明确的指定 main 头文件:

```
pod("NearbyMessages") {
    version = "1.1.1"
    headers = "GNSMessages.h"
}
```

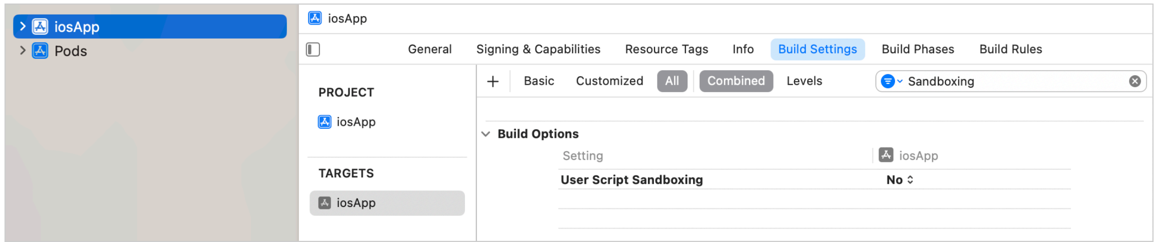
详情请参见 CocoaPods 文档 (https://guides.cocoapods.org/). 如果尝试过以上方法后, 仍然发生这个错误, 请到 YouTrack (https://youtrack.jetbrains.com/newissue?project=kt) 报告问题.

同步错误

你可能会遇到 `rsync error: some files could not be transferred` 错误. 这是一个 已知的问题 (https://github.com/CocoaPods/CocoaPods/issues/11946), 如果 Xcode 中的应用程序编译目标启用了用户脚本的沙箱功能(sandboxing), 就会发生这个错误.

要解决这个问题:

1. 在应用程序目标设定中禁用用户脚本的沙箱功能:



禁用 CocoaPods 沙箱功能

2. 停止可能已经启用了沙箱功能的 Gradle daemon 进程:

```
./gradlew --stop
```

添加 Pod 库依赖项

最终更新: 2024/09/10

要添加 Kotlin 项目对 Pod 库的依赖项, 需要 完成初始配置 ("设置 CocoaPods 环境" in "CocoaPods 概述与设置"). 然后你就可以添加各种类型的 Pod 库依赖项.

添加新的依赖项, 并在 IntelliJ IDEA 并重新导入项目之后, 新的依赖项会被自动添加进来. 不需要其他步骤.

要让你的 Kotlin 项目与 Xcode 协同工作, 应该 修改项目的 Podfile 文件 ("为 Xcode 更新 Podfile 文件" in "CocoaPods 概述与设置").

Kotlin 项目需要在 `build.gradle(.kts)` 中调用 `pod()` 函数来添加 Pod 依赖项. 每个依赖项都需要单独调用这个函数. 可以在函数的配置代码中对依赖项指定参数.

> 如果你不指定部署目标(deployment target)最小版本, 而且依赖项 Pod 需要更高的部署目标版本, 那么会发生错误.

示例项目参见 这里 (https://github.com/Kotlin/kmm-with-cocoapods-sample).

从 CocoaPods 仓库添加 Pod 库依赖项

1. 在 `pod()` 函数内指定 Pod 库名称.

   在配置代码段中, 可以使用 `version` 参数指定库的版本. 要使用库的最新版本, 可以完全省略这个参数.

   > 可以添加对 subspecs 的依赖项.

2. 指定 Pod 库的部署目标(deployment target)最小版本.

```
kotlin {
    ios()

    cocoapods {
        ios.deploymentTarget = "13.5"

        summary = "CocoaPods test library"
```

```
        homepage = "https://github.com/JetBrains/kotlin"

        pod("FirebaseAuth") {
            version = "10.16.0"
        }
    }
}
```

3. 重新导入项目.

要在 Kotlin 代码中使用这些依赖项, 需要导入 `cocoapods.<library-name>` 包:

```
import cocoapods.FirebaseAuth.*
```

使用保存在本地的 Pod 库添加依赖项

1. 在 `pod()` 函数内指定 Pod 库名称.

   在配置代码段中, 指定本地 Pod 库的路径: 在 `source` 参数值中使用 `path()` 函数.

   > 也可以添加本地 subspecs 的依赖项. `cocoapods` 代码段可以同时包含保存在本地的 Pod 库的依赖项, 以及 CocoaPods 仓库的 Pod 库的依赖项.

2. 指定 Pod 库的部署目标(deployment target)最小版本.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("pod_dependency") {
            version = "1.0"
            source = path(project.file("../pod_dependency"))
        }
```

```
        pod("subspec_dependency/Core") {
            version = "1.0"
            source = path(project.file("../subspec_dependency"))
        }
        pod("FirebaseAuth") {
            version = "10.16.0"
        }
    }
}
```

> 也可以在配置代码段中使用 version 参数指定库的版本. 要使用库的最新版本, 可以省略这个参数.

3. 重新导入项目.

要在 Kotlin 代码中使用这些依赖项, 需要导入 cocoapods.<library-name> 包:

```
import cocoapods.pod_dependency.*
import cocoapods.subspec_dependency.*
import cocoapods.FirebaseAuth.*
```

从自定义的 Git 仓库添加 Pod 库依赖项

1. 在 pod() 函数内指定 Pod 库名称.

   在配置代码段中, 指定 git 仓库路径: 在 source 参数中使用 git() 函数.

   此外, 还可以在 git() 之后的代码段中指定以下参数:

   - commit – 使用仓库中特定的 commit
   - tag – 使用仓库中特定的 tag
   - branch – 使用仓库中特定的 branch

   git() 函数的参数优先级顺序如下: commit, tag, branch. 如果不指定参数, Kotlin plugin 使用 master branch 中的 HEAD.

> 可以组合 `branch`, `commit`, 和 `tag` 参数来得到 Pod 库的特定版本.

2. 指定 Pod 库的部署目标(deployment target)最小版本.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("FirebaseAuth") {
            source = git("https://github.com/firebase/firebase-
ios-sdk") {
                tag = "10.16.0"
            }
        }

        pod("JSONModel") {
            source =
git("https://github.com/jsonmodel/jsonmodel.git") {
                branch = "key-mapper-class"
            }
        }

        pod("CocoaLumberjack") {
            source =
git("https://github.com/CocoaLumberjack/CocoaLumberjack.git") {
                commit =
"3e7f595e3a459c39b917aacf9856cd2a48c4dbf3"
            }
        }
    }
}
```

3. 重新导入项目.

要在 Kotlin 代码中使用这些依赖项, 需要导入 `cocoapods.<library-name>` 包:

```
import cocoapods.Alamofire.*
import cocoapods.JSONModel.*
import cocoapods.CocoaLumberjack.*
```

从自定义 Podspec 仓库添加 Pod 库依赖项

1. 在 `specRepos` 代码段之内, 使用 `url()` 函数, 指定自定义 Podspec 仓库的 HTTP 地址.

2. 在 `pod()` 函数内指定 Pod 库名称.

3. 指定 Pod 库的部署目标(deployment target)最小版本.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        specRepos {
            url("https://github.com/Kotlin/kotlin-cocoapods-
spec.git")
        }
        pod("example")
    }
}
```

4. 重新导入项目.

> 要与 Xcode 正确的协同工作, 需要在你的 Podfile 文件的最开始指定 spec 的位置. 例如:

```
source 'https://github.com/Kotlin/kotlin-cocoapods-spec.git'
```

要在 Kotlin 代码中使用这些依赖项, 需要导入 `cocoapods.<library-name>` 包:

```
import cocoapods.example.*
```

使用自定义 cinterop 选项添加 Pod 库依赖项

1. 在 `pod()` 函数内指定 Pod 库名称.

   在配置代码段中, 指定 cinterop 选项:

   - `extraOpts` – 指定对 Pod 库的选项列表. 例如, 指定 flag: `extraOpts = listOf("-compiler-option")`.
   - `packageName` – 指定包名称. 如果有指定, 可以使用这个包名称导入这个库: `import <packageName>`.

2. 指定 Pod 库的部署目标(deployment target)最小版本.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("YandexMapKit") {
            packageName = "YandexMK"
        }
    }
}
```

3. 重新导入项目.

要在 Kotlin 代码中使用这些依赖项, 需要导入 `cocoapods.<library-name>` 包:

```
import cocoapods.YandexMapKit.*
```

如果使用了 `packageName` 参数, 那么可以使用包这个名称导入这个库 `import <packageName>`:

```
import YandexMK.YMKPoint
import YandexMK.YMKDistance
```

对带 @import 命令的 Objective-C 头文件的支持

> 这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://kotl.in/issue) 提供你的反馈意见.

某些 Objective-C 库, 尤其是 Swift 库的封装库, 在它们的头文件中存在 `@import` 命令. 默认情况下, cinterop 不支持这些命令.

要启用对 `@import` 命令的支持, 请在 `pod()` 函数的配置代码段中指定 `-fmodules` 选项:

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("PodName") {
            extraOpts = listOf("-compiler-option", "-fmodules")
        }
    }
}
```

在依赖的 Pod 之间共用 Kotlin cinterop

如果你使用 `pod()` 函数添加了多个 Pod 库依赖项, 当你的 Pod 库的 API 之间存在依赖关系时, 你可能会遇到问题.

这种情况下, 为了让代码成功编译, 请使用 `useInteropBindingFrom()` 函数. 在为新的 Pod 构建绑定时, 这个函数会利用为另一个 Pod 生成的 cinterop 绑定.

你应该在设置依赖项之前声明依赖的 Pod 库:

```
// pod("WebImage") 的 cinterop:
fun loadImage(): WebImage

// pod("Info") 的 cinterop:
fun printImageInfo(image: WebImage)

// 你的代码:
printImageInfo(loadImage())
```

这样的情况下, 如果你没有正确配置 cinterop 之间的依赖关系, 这段代码会无效, 因为 `WebImage` 类型在不同的 cinterop 文件内, 因此, 它也属于不同的包.

将 Kotlin Gradle 项目用作 CocoaPods 依赖项

最终更新: 2024/09/10

要将带有 native 编译目标的 Kotlin Multiplatform 项目用作 CocoaPods 的依赖项, 需要 完成初始配置 ("设置 CocoaPods 环境" in "CocoaPods 概述与设置"). 你可以在 Xcode 项目的 Podfile 中, 通过它的名称和生成的 Podspec 文件的目录路径来包含这样的依赖项.

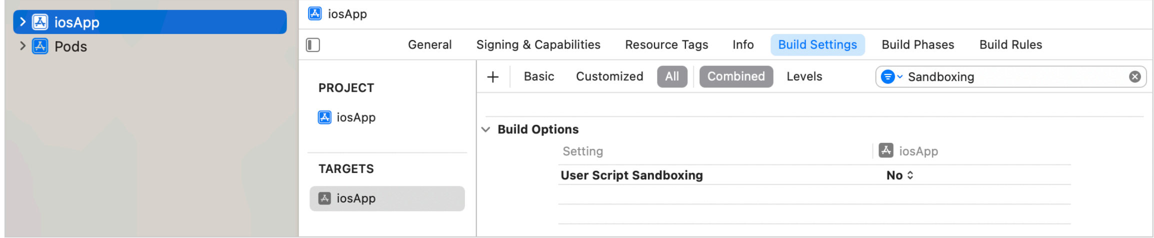
依赖项将会与项目一起自动构建(以及重构建). 这样的方案可以简化 Kotlin Multiplatform 项目到 Xcode 的导入工作, 因为不再需要编写对应的 Gradle task 和 Xcode 构建步骤.

你可以在 Kotlin Gradle 项目和带有一个或多个编译目标的 Xcode 项目之间添加依赖. 也可以在 Gradle 项目和多个 Xcode 项目之间添加依赖. 但是, 这种情况下, 你需要通过对每个 Xcode 项目调用手动 `pod install` 来添加依赖. 其他情况下, 这个调用可以自动完成.

- 要将依赖项正确导入到 Kotlin/Native 模块, `Podfile` 必须包含: `use_modular_headers!` (https://guides.cocoapods.org/syntax/podfile.html#use_modular_headers_bang) 或 `use_frameworks!` (https://guides.cocoapods.org/syntax/podfile.html#use_frameworks_bang) 指令.
- 如果你不指定部署目标(deployment target)最小版本, 而且依赖项 Pod 需要更高的部署目标版本, 那么会发生错误.

单个编译目标的 Xcode 项目

1. 如果你还没有 Xcode 项目, 请使用 `Podfile` 创建一个.
2. 在应用程序 Target 中, 请确认禁用了 **Build Options** 之下的 **User Script Sandboxing**:



禁用 sandboxing CocoaPods

3. 使用 `podfile = project.file(..)` 向你的 Xcode 项目 `Podfile` 添加路径, 其中的文件路径是你的 Kotlin 项目的 `build.gradle.kts` (`build.gradle`) 文件路径. 这个步骤可以通过对你的 `Podfile` 调用 `pod install`, 帮助你的 Xcode 项目与 Gradle 项目依赖项保持同步.

4. 指定 Pod 库的部署目标(deployment target)最小版本.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"
        ios.deploymentTarget = "13.5"
        pod("FirebaseAuth") {
            version = "10.16.0"
        }
        podfile = project.file("../ios-app/Podfile")
    }
}
```

5. 对你在 Xcode 项目中想要包含的 Gradle 项目, 将它的名称和路径添加到 `Podfile`.

```
use_frameworks!

platform :ios, '13.5'

target 'ios-app' do
    pod 'kotlin_library', :path => '../kotlin-library'
end
```

6. 重新导入项目.

多个编译目标的 Xcode 项目

1. 如果你还没有 Xcode 项目, 请使用 `Podfile` 创建一个.
2. 使用 `podfile = project.file(..)` 向你的 Xcode 项目 `Podfile` 添加路径, 其中的文件路径是你的 Kotlin 项目的 `build.gradle(.kts)`. 这个步骤可以通过对你的 `Podfile` 调用 `pod install`, 帮助你的 Xcode 项目与 Gradle 项目依赖项保持同步.
3. 对于你的项目中希望使用的 Pod 库, 使用`pod()` 添加依赖项.
4. 对每个目标, 指定 Pod 库的部署目标(deployment target)最小版本.

```
kotlin {
    ios()
    tvos()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"
        ios.deploymentTarget = "13.5"
        tvos.deploymentTarget = "13.4"

        pod("FirebaseAuth") {
            version = "10.16.0"
        }
        podfile =
project.file("../severalTargetsXcodeProject/Podfile") // 指定
Podfile 路径
    }
}
```

5. 对你在 Xcode 项目中想要包含的 Gradle 项目, 将它的名称和路径添加到 `Podfile`.

```
target 'iosApp' do
    use_frameworks!
    platform :ios, '13.5'
    # Pods for iosApp
    pod 'kotlin_library', :path => '../kotlin-library'
end
```

```
target 'TVosApp' do
    use_frameworks!
    platform :tvos, '13.4'

    # Pods for TVosApp
    pod 'kotlin_library', :path => '../kotlin-library'
end
```

6. 重新导入项目.

参见 示例项目 (https://github.com/Kotlin/kmm-with-cocoapods-multitarget-xcode-sample).

CocoaPods Gradle plugin DSL 参考文档

最终更新: 2024/09/10

Kotlin CocoaPods Gradle plugin 是一个用来创建 Podspec 文件的工具. 将你的 Kotlin 项目与 CocoaPods 依赖项管理器 (https://cocoapods.org/) 集成时, 会需要这些文件.

本文档将会介绍, 使用 CocoaPods 集成 (CocoaPods 概述与设置) 时, 你可以使用的 Kotlin CocoaPods Gradle plugin 的所有代码段, 函数, 属性.

- 学习如何 设置开发环境 ("设置 CocoaPods 环境" in "CocoaPods 概述与设置") 并 配置 Kotlin CocoaPods Gradle plugin ("添加并配置 Kotlin CocoaPods Gradle plugin" in "CocoaPods 概述与设置").
- 根据你的项目和目的不同, 你可以添加 Kotlin 项目与 Pod 库 (添加 Pod 库依赖项) 之间的依赖项, 也可以添加 Kotlin Gradle 项目与 Xcode 项目 (将 Kotlin Gradle 项目用作 CocoaPods 依赖项) 之间的依赖项.

启用 plugin

要启用 CocoaPods plugin, 请在 `build.gradle(.kts)` 文件添加以下代码:

```
plugins {
    kotlin("multiplatform") version "1.9.23"
    kotlin("native.cocoapods") version "1.9.23"
}
```

plugin 版本与 Kotlin 发布版本 (Kotlin 的发布版本) 相同. 最新的稳定版本是 1.9.23.

cocoapods 代码段

`cocoapods` 代码段是用于 CocoaPods 配置的的最顶层代码段. 它包含 Pod 的一般信息, 包括必须信息, 例如 Pod 版本, 概述, homepage, 以及可选的功能特性.

在这个代码段内部, 你可以使用以下代码段, 函数, 和属性:

名称	描述
`version`	Pod 版本. 如果不指定, 会使用 Gradle 项目的版本. 如果 Gradle 项目的版本也没有设置, 会发生错误.
`summary`	通过这个项目构建得到的 Pod 的描述信息, 必须指定.
`homepage`	通过这个项目构建得到的 Pod 的 homepage 链接, 必须指定.
`authors`	通过这个项目构建得到的 Pod 的作者.
`podfile`	配置已有的 `Podfile` 文件.
`noPodspec()`	设置 plugin 不要为 `cocoapods` 小节生成 Podspec 文件.
`name`	通过这个项目构建得到的 Pod 的名称. 如果不指定, 会使用项目的名称.
`license`	通过这个项目构建得到的 Pod 的许可协议类型, 以及文字.
`framework`	framework 代码段对 plugin 生成的框架进行配置.
`source`	通过这个项目构建得到的 Pod 的位置.
`extraSpecAttributes`	配置其他 Podspec 属性, 例如 `libraries` 或 `vendored_frameworks`.
`xcodeConfigurationToNativeBuildType`	将自定义的 Xcode 配置映射到 NativeBuildType: "Debug" 映射为 `NativeBuildType.DEBUG`, "Release" 映射为 `NativeBuildType.RELEASE`.
`publishDir`	配置 Pod 发布时的输出目录.
`pods`	返回 Pod 依赖项列表.

pod()	向通过这个项目构建得到的 Pod 添加一个 CocoaPods 依赖项.
specRepos	添加一个使用 url() 的特定的仓库. 当使用私有 Pod 作为依赖项时, 需要这样的设置. 详情请参见 CocoaPods 文档 (https://guides.cocoapods.org/making/private-cocoapods.html).

编译目标

- ios
- osx
- tvos
- watchos

对每个编译目标, 可以使用 deploymentTarget 属性来对 Pod 库指定编译目标最低版本.

指定这个属性后, CocoaPods 会对所有的编译目标添加 debug 和 release 框架, 作为输出的二进制文件.

```
kotlin {
    ios()

    cocoapods {
        version = "2.0"
        name = "MyCocoaPod"
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        extraSpecAttributes["vendored_frameworks"] =
'CustomFramework.xcframework'
        license = "{ :type => 'MIT', :text => 'License text'}"
        source = "{ :git =>
'git@github.com:vkormushkin/kmmpodlibrary.git', :tag => '$version'
}"
        authors = "Kotlin Dev"

        specRepos {
```

```
            url("https://github.com/Kotlin/kotlin-cocoapods-
spec.git")
        }
        pod("example")

        xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] =
NativeBuildType.RELEASE
    }
}
```

framework 代码段

`framework` 代码段嵌套在 `cocoapods` 代码段内, 用来配置通过项目构建的 Pod 的框架属性.

> 注意, `baseName` 是必须指定的项目.

名称	描述
`baseName`	框架名称, 必须指定. `frameworkName` 已被废弃, 请改为使用这个属性.
`isStatic`	定义框架的链接类型. 默认为 dynamic.
`transitiveExport`	启用依赖项导出.

```
kotlin {
    cocoapods {
        framework {
            baseName = "MyFramework"
            isStatic = false
            export(project(":anotherKMMModule"))
            transitiveExport = true
        }
    }
}
```

pod() 函数

pod() 函数调用会对通过这个项目构建得到的 Pod 添加一个 CocoaPods 依赖项. 每个依赖项都需要一个单独的 pod() 函数调用.

在函数参数中, 你可以指定一个 Pod 库的名称. 在这个函数的配置代码段中, 还可以指定其他参数, 例如库的 version 和 source:

名称	描述
version	库的版本. 要使用库的最新版本, 请省略这个参数.
source	可以使用以下几种方式来配置 Pod: 通过 git() 函数, 使用 Git 仓库中的 Pod. 在 git() 之后的代码段中, 你可以指定 commit 来使用特定的 commit, 可以指定 tag 来使用特定的 tag, 可以指定 branch 来使用 特定的 branch 通过 path() 函数, 使用本地仓库中的 Pod 通过 url() 函数, 使用打包的(tar, jar, zip) Pod 文件夹
packageName	指定包名称.
extraOpts	为 Pod 库指定选项列表. 例如, 特定的参数: extraOpts = listOf("-compiler-option")
linkOnly	让 CocoaPods plugin 使用动态框架(Dynamic Framework)的 Pod 依赖项, 不生成 cinterop 绑定. 如果对静态框架(Static Framework)使用这个选项, 会删除整个 Pod 依赖项.
interopBindingDependencies	包含对其他 Pod 的依赖项列表. 在对新的 Pod 构建 Kotlin 绑定时会使用这个列表.
useInteropBindingFrom()	指定用做依赖项的已存在的 Pod 的名称. 这个 Pod 需要在函数执行之前声明. 这个函数让 CocoaPods plugin 在对新的 Pod 构建 Kotlin 绑定时, 使用已存在的 Pod 的绑定.

```
kotlin {
    ios()

    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("pod_dependency") {
            version = "1.0"
            linkOnly = true
            source = path(project.file("../pod_dependency"))
        }
    }
}
```

Kotlin/Native 库

最终更新: 2024/09/10

Kotlin 编译器使用方法

要通过 Kotlin/Native 编译器编译产生库文件, 请使用 `-produce library` 或 `-p library` 参数. 例如:

```
$ kotlinc-native foo.kt -p library -o bar
```

这个命令会编译源代码文件 `foo.kt`, 输出为库文件 `bar.klib`.

要链接一个库, 请使用 `-library <name>` 或 `-l <name>` 参数. 例如:

```
$ kotlinc-native qux.kt -l bar
```

这个命令会编译源代码文件 `qux.kt`, 与库文件 `bar.klib` 链接, 输出为 `program.kexe`.

cinterop 工具使用方法

cinterop 工具会对原生的库文件生成 `.klib` 格式的包装. 比如, 可以使用 Kotlin/Native 发布中附带的简单的 `libgit2.def` 原生库定义文件

```
$ cinterop -def
samples/gitchurn/src/nativeInterop/cinterop/libgit2.def -compiler-
option -I/usr/local/include -o libgit2
```

我们可以得到 `libgit2.klib` 文件.

详情请参见 与 C 代码交互 (与 C 代码交互).

klib 工具

klib 库管理工具可以用来查看和安装库.

可用的命令如下:

- `content` – 列出库的内容:

```
$ klib contents <name>
```

- `info` – 查看库的内容细节:

```
$ klib info <name>
```

- `install` – 要把库安装到默认的位置, 可以使用:

```
$ klib install <name>
```

- `remove` – 从默认的仓库中删除一个库, 可以使用:

```
$ klib remove <name>
```

以上所有命令都可以接受一个 `-repository <directory>` 参数, 用来指定默认值以外的仓库位置.

```
$ klib <command> <name> -repository <directory>
```

几个例子

首先我们来创建一个库. 把我们这个小小的库的源代码放在 `kotlinizer.kt` 文件内:

```
package kotlinizer
val String.kotlinized
    get() = "Kotlin $this"
```

```
$ kotlinc-native kotlinizer.kt -p library -o kotlinizer
```

库会被创建到当前目录下:

```
$ ls kotlinizer.klib
kotlinizer.klib
```

现在我们来看看库的内容:

```
$ klib contents kotlinizer
```

你可以将 `kotlinizer` 库安装到默认的仓库中:

```
$ klib install kotlinizer
```

然后在当前目录中删除它的一切痕迹:

```
$ rm kotlinizer.klib
```

编写一个很短的程序, 放在 `use.kt` 文件中:

```
import kotlinizer.*

fun main(args: Array<String>) {
    println("Hello, ${"world".kotlinized}!")
}
```

然后编译这个程序, 并链接你刚才创建的库:

```
$ kotlinc-native use.kt -l kotlinizer -o kohello
```

然后运行这个程序:

```
$ ./kohello.kexe
Hello, Kotlin world!
```

祝你玩得开心!

高级问题

库的查找顺序

当我们指定 `-library foo` 参数时, 编译器会按照以下顺序查找 `foo` 库:

- 当前编译目录, 或一个绝对路径.
- 通过 `-repo` 参数指定的所有仓库.
- 默认仓库中安装的所有库.

> 默认仓库是 `~/.konan`. You can change it 你可以设置 Gradle 属性 `kotlin.data.dir` 来修改这个值.
>
> 或者, 也可以使用 `-Xkonan-data-dir` 编译器选项, 通过 `cinterop` 和 `konanc` 工具来配置你的的自定义目录路径.

* `$installation/klib` 目录中安装的所有库.

库文件的格式

Kotlin/Native 库是 zip 文件, 包含预定义的目录结构, 如下:

`foo.klib` 解压缩到 `foo/` 目录后会得到以下内容:

```
- foo/
  - $component_name/
    - ir/
      - 序列化后的 Kotlin IR.
    - targets/
      - $platform/
        - kotlin/
          - Kotlin 编译产生的 LLVM bitcode 文件.
        - native/
          - 其他原生对象的 bitcode 文件.
      - $another_platform/
        - 可能存在几组平台相关的目录, 其中都包含 kotlin 和 native 目录.
    - linkdata/
      - 一组 ProtoBuf 文件, 包含序列化链接元数据(serialized linkage
metadata).
    - resources/
      - 一般资源文件, 比如图像文件. (暂时没有使用).
    - manifest - 库的描述文件, 使用 java property 格式.
```

在你的 Kotlin/Native 环境的 `klib/stdlib` 目录下可以找到这些库文件结构的例子.

在 klib 中使用相对路径

> klib 中的相对路径功能从 Kotlin 1.6.20 开始可用.

源代码文件的序列化后的 IR 表达是一个 `klib` 库的一部分. 其中包含文件路径, 用于生成正确的调试信息. 默认情况下, 存储的路径是绝对路径. 使用 `-Xklib-relative-path-base`, 你可以修改库的格式, 在 artifact 中只使用相对路径. 要让这个功能有效, 需要向编译器选项的参数传递一个或多个源代码文件基准路径:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure
{
    // $base 是源代码文件的基准路径
    compilerOptions.freeCompilerArgs.add("-Xklib-relative-path-
base=$base")
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        // $base 是源代码文件的基准路径
        freeCompilerArgs.add("-Xklib-relative-path-base=$base")
    }
}
```

平台库

最终更新: 2024/09/10

为了实现对使用者的原生操作系统服务的访问能力, Kotlin/Native 发布版包含了一组针对各个平台预先编译好的库. 我们称之为 **平台库**.

POSIX 绑定

对于所有基于 Unix 或 Windows 的平台 (包括 Android 和 iOS 编译平台) 我们提供了 POSIX 平台库. 其中包含对 `POSIX` 标准 (https://en.wikipedia.org/wiki/POSIX) 在各平台实现的绑定.

要使用这个库, 只需要导入它:

```
import platform.posix.*
```

唯一不能使用这个库的平台是 WebAssembly (https://en.wikipedia.org/wiki/WebAssembly).

注意, `platform.posix` 的内容在各个平台是不同的, 因为各个平台的 POSIX 具体实现也是略微不同的.

流行的原生库

对于本机编译平台或交叉编译(cross-compilation)平台, 有很多可用的平台库. Kotlin/Native 发布版, 在可用的平台上提供了对 OpenGL, zlib 以及其他流行的原生库的访问能力.

在 Apple 平台, 提供了 `objc` 库, 用于与 Objective-C (https://en.wikipedia.org/wiki/Objective-C) 交互.

详情请参见 `Kotlin/Native` 发布版的 `dist/klib/platform/$target` 目录内容.

默认可用性

平台库中的包默认是可用的. 使用它们时, 不需要指定特别的链接参数. Kotlin/Native 编译器会自动检测访问了哪些平台库, 并自动链接需要的库文件.

另一方面, `Kotlin/Native` 发布版中的平台库仅仅只是包装并绑定到原生的库文件. 因此原生库文件本身 (`.so`, `.a`, `.dylib`, `.dll` 等等)必须安装在机器上.

教程 - 使用 Kotlin/Native 开发动态库

最终更新: 2024/09/10

通过本教程, 你将学习如何在既有的原生应用程序或库中使用 Kotlin/Native 代码. 为了这个目的, 你需要将 Kotlin 代码编译为一个动态库, `.so`, `.dylib`, 和 `.dll`.

Kotlin/Native 还与 Apple 技术高度集成. 使用 Kotlin/Native 开发 Apple Framework (教程 - 使用 Kotlin/Native 开发 Apple Framework) 教程介绍如何将 Kotlin 代码编译为一个框架, 供 Swift 和 Objective-C 使用.

在本教程中, 你将会:

- 将 Kotlin 代码编译为一个动态库
- 检查生成的 C 头文件
- 在 C 中使用 Kotlin 动态库
- 在 Linux 和 Mac 以及 Windows 上编译并运行示例程序

创建一个 Kotlin 库

Kotlin/Native 编译器能够从 Kotlin 代码生成一个动态库. 一个动态库通常带有一个头文件, 也就是一个 `.h` 文件, 你在 C 语言中使用它来调用编译后的代码.

理解这些技术的最好方法就是来试用一下它们. 首先我们创建一个小小的 Kotlin 库, 然后在一个 C 程序中使用它.

首先在 Kotlin 中创建一个库文件, 保存为 `hello.kt`:

```
package example

object Object {
    val field = "A"
}

class Clazz {
    fun memberFunction(p: Int): ULong = 42UL
}
```

```
fun forIntegers(b: Byte, s: Short, i: UInt, l: Long) { }
fun forFloats(f: Float, d: Double) { }

fun strings(str: String) : String? {
    return "That is '$str' from C"
}

val globalString = "A global String"
```

尽管可以直接使用命令行, 或者通过脚本文件(比如 .sh 或 .bat 文件), 但这种方法不适合于包含几百个文件和库的大项目. 更好的方法是使用带有构建系统的 Kotlin/Native 编译器, 因为它会帮助你下载并缓存 Kotlin/Native 编译器二进制文件, 传递依赖的库, 并运行编译器和测试. Kotlin/Native 能够通过 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 使用 Gradle (https://gradle.org) 构建系统.

关于如何使用 Gradle 设置 IDE 兼容的项目, 请参见教程 一个基本的 Kotlin/Native 应用程序 (Kotlin/Native 开发入门 - 使用 Gradle). 如果你想要寻找具体的步骤指南, 来开始一个新的 Kotlin/Native 项目并在 IntelliJ IDEA 中打开它, 请先阅读这篇教程. 在本教程中, 我们关注更高级的 C 交互功能, 包括使用 Kotlin/Native, 以及使用 Gradle 的 跨平台 ("编译到多个目标平台" in "配置 Gradle 项目") 构建.

首先, 创建一个项目文件夹. 本教程中的所有路径都是基于这个文件夹的相对路径. 有时在添加任何新文件之前, 会需要创建缺少的目录.

使用以下 build.gradle(.kts) Gradle 构建文件:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
```

```
    // mingwX64("native") { // 用于 Windows 环境
        binaries {
            sharedLib {
                baseName = "native" // 用于 Linux 和 macOS 环境
                // baseName = "libnative" // 用于 Windows 环境
            }
        }
    }
}

tasks.wrapper {
    gradleVersion = "8.1.1"
    distributionType = Wrapper.DistributionType.ALL
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}

repositories {
    mavenCentral()
}

kotlin {
    linuxX64("native") { // 用于 Linux 环境
    // macosX64("native") { // 用于 x86_64 macOS 环境
    // macosArm64("native") { // 用于 Apple Silicon macOS 环境
    // mingwX64("native") { // 用于 Windows 环境
        binaries {
            sharedLib {
                baseName = "native" // 用于 Linux 和 macOS 环境
                // baseName = "libnative" // 用于 Windows 环境
            }
        }
    }
```

```
    }

    wrapper {
        gradleVersion = "8.1.1"
        distributionType = "ALL"
    }
```

将源代码文件移动到项目的 src/nativeMain/kotlin 文件夹中. 这是使用 kotlin-multiplatform ("编译到多个目标平台" in "配置 Gradle 项目") plugin 时的默认源代码路径. 使用以下代码块来配置项目, 生成一个动态库或共用库:

```
binaries {
    sharedLib {
        baseName = "native" // 用于 Linux 和 macOS 环境
        // baseName = "libnative" // 用于 Windows 环境
    }
}
```

libnative 用作库名称, 以及生成的头文件名称前缀. 它还是头文件中所有声明的前缀.

现在你可以 在 IntelliJ IDEA 中打开项目 (Kotlin/Native 开发入门 - 使用 IntelliJ IDEA), 并查看如何修改示例项目. 在这个过程中, 我们会看看 C 函数如何映射为 Kotlin/Native 声明.

可以在 IDE 中运行 linkNative Gradle task 来构建库, 或执行以下控制台命令:

```
./gradlew linkNative
```

根据主机的 OS 不同, 构建会在 build/bin/native/debugShared 文件夹下生成以下文件:

- macOS: libnative_api.h 和 libnative.dylib
- Linux: libnative_api.h 和 libnative.so
- Windows: libnative_api.h, libnative_symbols.def 和 libnative.dll

Kotlin/Native 编译器对所有平台生成 .h 文件时, 使用相同的规则. 我们来看看我们的 Kotlin 库的 C API.

生成的头文件

在 libnative_api.h 中, 你将看到以下代码. 我们把代码分成各个部分来讨论, 这样比较容易理解.

> Kotlin/Native 导出符号的方式可能会发生变化, 不另行通知.

第一部分包含标准的 C/C++ 代码头部和尾部:

```
#ifndef KONAN_DEMO_H
#define KONAN_DEMO_H
#ifdef __cplusplus
extern "C" {
#endif

/// 这里是生成的代码的其它部分

#ifdef __cplusplus
}  /* extern "C" */
#endif
#endif  /* KONAN_DEMO_H */
```

在 libnative_api.h 中, 在上述惯例部分之外, 有一个代码块, 包含共通的类型定义:

```
#ifdef __cplusplus
typedef bool            libnative_KBoolean;
#else
typedef _Bool           libnative_KBoolean;
#endif
typedef unsigned short     libnative_KChar;
typedef signed char        libnative_KByte;
typedef short              libnative_KShort;
typedef int                libnative_KInt;
typedef long long          libnative_KLong;
typedef unsigned char      libnative_KUByte;
typedef unsigned short     libnative_KUShort;
typedef unsigned int       libnative_KUInt;
typedef unsigned long long libnative_KULong;
typedef float              libnative_KFloat;
```

```
typedef double                  libnative_KDouble;
typedef void*                   libnative_KNativePtr;
```

在创建的 `libnative_api.h` 文件中, Kotlin 对所有的声明使用 `libnative_` 前缀. 我们把类型的对应关系整理为下面的对应表, 这样更容易阅读:

Kotlin 定义	**C 类型**
`libnative_KBoolean`	`bool` 或 `_Bool`
`libnative_KChar`	`unsigned short`
`libnative_KByte`	`signed char`
`libnative_KShort`	`short`
`libnative_KInt`	`int`
`libnative_KLong`	`long long`
`libnative_KUByte`	`unsigned char`
`libnative_KUShort`	`unsigned short`
`libnative_KUInt`	`unsigned int`
`libnative_KULong`	`unsigned long long`
`libnative_KFloat`	`float`
`libnative_KDouble`	`double`
`libnative_KNativePtr`	`void*`

定义部分显示 Kotlin 基本类型如何映射为 C 基本类型. 反过来的对应关系请参见 映射 C 语言的基本数据类型 (教程 - 映射 C 语言的基本数据类型) 教程.

libnative_api.h 文件的下一部分包含库中使用的类型的定义:

```
struct libnative_KType;
typedef struct libnative_KType libnative_KType;

typedef struct {
    libnative_KNativePtr pinned;
} libnative_kref_example_Object;

typedef struct {
    libnative_KNativePtr pinned;
} libnative_kref_example_Clazz;
```

C 语言中使用 typedef struct { .. } TYPE_NAME 语法来声明一个结构(structure). Stackoverflow 上的 这个讨论串 (https://stackoverflow.com/questions/1675351/typedef-struct-vs-struct-definitions) 对这种模式有更详细的解释.

从这些定义你可以看到, Kotlin 对象 Object 映射为 libnative_kref_example_Object, Clazz 映射为 libnative_kref_example_Clazz. 两个结构都仅仅包含一个指针类型的 pinned 域变量, 域变量类型 libnative_KNativePtr 在上面定义为 void*.

C 中不支持命名空间(namespace), 因此 Kotlin/Native 编译器生成很长的名称, 以免与既有的原生项目中的其它符号发生名称冲突.

定义一个重要的部分也在 libnative_api.h 文件中. 它包含我们的 Kotlin/Native 库的定义:

```
typedef struct {
    /* 服务函数. */
    void (*DisposeStablePointer)(libnative_KNativePtr ptr);
    void (*DisposeString)(const char* string);
    libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const
libnative_KType* type);

    /* 使用者函数. */
    struct {
        struct {
            struct {
                void (*forIntegers)(libnative_KByte b,
```

```
libnative_KShort s, libnative_KUInt i, libnative_KLong l);
                void (*forFloats)(libnative_KFloat f,
libnative_KDouble d);
                const char* (*strings)(const char* str);
                const char* (*get_globalString)();
                struct {
                    libnative_KType* (*_type)(void);
                    libnative_kref_example_Object (*_instance)();
                    const char* (*get_field)
(libnative_kref_example_Object thiz);
                } Object;
                struct {
                    libnative_KType* (*_type)(void);
                    libnative_kref_example_Clazz (*Clazz)();
                    libnative_KULong (*memberFunction)
(libnative_kref_example_Clazz thiz, libnative_KInt p);
                } Clazz;
            } example;
        } root;
    } kotlin;
} libnative_ExportedSymbols;
```

这段代码使用匿名的结构声明. 代码 `struct { .. } foo` 在这个匿名结构类型(这个类型没有名称)的外层结构中声明一个域变量.

C 也不支持对象. 人们使用函数指针来模仿对象语义. 函数指针声明为 `RETURN_TYPE (* FIELD_NAME)(PARAMETERS)`. 这样的代码很难阅读, 但在上面的结构中我们可以看到函数指针类型的域变量.

运行时函数

上面的代码含义如下. 你有一个 `libnative_ExportedSymbols` 结构, 它定义 Kotlin/Native 和我们的库提供的所有函数. 它大量使用嵌套的匿名结构, 来模拟包. `libnative_` 前缀来自库的名称.

`libnative_ExportedSymbols` 结构包含一些帮助函数:

```
void (*DisposeStablePointer)(libnative_KNativePtr ptr);
void (*DisposeString)(const char* string);
libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const
libnative_KType* type);
```

这些函数处理 Kotlin/Native 对象. 调用 DisposeStablePointer 可以释放一个 Kotlin 对象, 调用 DisposeString 可以释放一个 Kotlin 字符串, 字符串在 C 中对应为 char* 类型. 可以使用 IsInstance 函数来检查一个 Kotlin 类型或一个 libnative_KNativePtr 是不是另一个类型的实例. 实际上生成哪些操作, 依赖于具体的使用场景.

Kotlin/Native 有垃圾收集功能, 但它不能帮助我们在 C 语言中处理 Kotlin 对象. Kotlin/Native 拥有与 Objective-C 和 Swift 交互的能力, 并与它们的引用计数集成. 与 Objective-C 代码交互 (与 Swift/Objective-C 代码交互) 文档中包含这些问题的更多详细信息. 此外还可以参考教程 使用 Kotlin/Native 开发 Apple Framework (教程 - 使用 Kotlin/Native 开发 Apple Framework).

你的库函数

我们来看一下 kotlin.root.example 域, 它通过一个 kotlin.root. 前缀来模拟我们 Kotlin 代码的包结构.

有一个 kotlin.root.example.Clazz 域表达 Kotlin 中的 Clazz. 通过 memberFunction 域可以访问 Clazz#memberFunction. 唯一的区别是 memberFunction 的第一个参数接受一个 this 引用. C 语言不支持对象, 所以需要明确的传递一个 this 指针.

在 Clazz 域中存在一个构造器 (也就是 kotlin.root.example.Clazz.Clazz), 它是构造器函数, 用来创建 Clazz 的实例.

Kotlin object Object 可以通过 kotlin.root.example.Object 访问. 有一个 _instance 函数可以得到对象的唯一实例.

属性被翻译为函数. get_ 和 set_ 前缀分别用来命名 getter 和 setter 函数. 比如, Kotlin 中的只读属性 globalString 在 C 中被转换为一个 get_globalString 函数.

全局函数 forInts, forFloats, 或 strings 被转换为 kotlin.root.example 匿名结构中的函数指针.

入口点

你可以看到 API 是如何创建的. 首先, 你需要初始化 libnative_ExportedSymbols 结构. 关于这一点, 我们来看看 libnative_api.h 的最后部分:

```
extern libnative_ExportedSymbols* libnative_symbols(void);
```

通过函数 libnative_symbols 你可以打开从原生代码访问 Kotlin/Native 库的道路. 这就是你将要使用的入口点. 库名称被用作函数名称的前缀.

> Kotlin/Native 对象引用不支持多线程访问. 可能需要对每个线程分别保存返回的 libnative_ExportedSymbols* 指针.

在 C 中使用生成的头文件

在 C 中的使用非常直接, 并没有任何复杂之处. 创建一个 `main.c` 文件, 包含以下代码:

```
#include "libnative_api.h"
#include "stdio.h"

int main(int argc, char** argv) {
    // 获得引用, 用来调用 Kotlin/Native 函数
    libnative_ExportedSymbols* lib = libnative_symbols();

    lib->kotlin.root.example.forIntegers(1, 2, 3, 4);
    lib->kotlin.root.example.forFloats(1.0f, 2.0);

    // 使用 C 和 Kotlin/Native 字符串
    const char* str = "Hello from Native!";
    const char* response = lib->kotlin.root.example.strings(str);
    printf("in: %s\nout:%s\n", str, response);
    lib->DisposeString(response);

    // 创建 Kotlin 对象实例
    libnative_kref_example_Clazz newInstance = lib-
>kotlin.root.example.Clazz.Clazz();
    long x = lib-
>kotlin.root.example.Clazz.memberFunction(newInstance, 42);
    lib->DisposeStablePointer(newInstance.pinned);

    printf("DemoClazz returned %ld\n", x);

    return 0;
}
```

在 Linux 和 macOS 上编译并运行示例程序

在 macOS 10.13 的 Xcode 中, 使用以下命令, 编译 C 代码, 并链接到动态库:

```
clang main.c libnative.dylib
```

在 Linux 上可以使用类似的命令:

```
gcc main.c libnative.so
```

编译器会生成一个可执行文件, 名为 `a.out`. 运行它, 看看从 C 库执行 Kotlin 代码实际效果. 在 Linux 上, 你将需要将 `.` 包含到 `LD_LIBRARY_PATH`, 使应用程序能够从当前文件夹加载 `libnative.so` 库.

在 Windows 上编译并运行示例程序

首先, 你需要安装 Microsoft Visual C++ 编译器, 要支持 x64_64 编译目标. 最简单的方法是, 在 Windows 机器上安装一份 Microsoft Visual Studio.

在这个示例中, 你将使用 `x64 Native Tools Command Prompt <VERSION>` 控制台. 在开始菜单中你会看到打开控制台的快捷方式. 它是随 Microsoft Visual Studio 一起安装的.

在 Windows 上, 要装载动态库, 可以通过生成的静态的库包装器, 或者通过手动代码来装载, 后一种方法使用 LoadLibrary (https://docs.microsoft.com/en-gb/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya) 或类似的 Win32API 函数. 我们使用第一种方法, 为 `libnative.dll` 生成静态的库包装器, 具体方法如下.

调用工具链中的 `lib.exe` 来生成静态的库包装器 `libnative.lib`, 它负责在代码中自动装载 DLL:

```
lib /def:libnative_symbols.def /out:libnative.lib
```

现在你可以将我们的 `main.c` 编译为可执行文件. 在构建命令中包含生成的 `libnative.lib`, 然后开始编译:

```
cl.exe main.c libnative.lib
```

这个命令会输出 `main.exe` 文件, 这就是你最终可以运行的文件.

下一步做什么?

动态库是从既有程序中使用 Kotlin 代码的主要方式. 使用动态库, 你可以在很多平台和语言上共用你的代码, 包括 JVM, Python, iOS, Android, 等等.

Kotlin/Native 还与 Objective-C 和 Swift 紧密集成. 详情请参见 使用 Kotlin/Native 开发 Apple Framework (教程 - 使用 Kotlin/Native 开发 Apple Framework) 教程.

Kotlin/Native 内存管理

最终更新: 2024/09/10

Kotlin/Native 使用一个现代化的内存管理器, 类似于 JVM, Go, 以及其它主流技术, 包括以下功能:

- 对象存储在共享的堆(heap)中, 可以在任何线程中访问.
- 定期执行追踪垃圾收集器(Garbage Collector, GC), 回收那些从 "根(roots)" 无法到达的对象, 比如局部变量, 全局变量.

垃圾收集器

Kotlin/Native 的 GC 算法一直在持续演进. 目前使用的算法是 Stop-the-World Mark 和 Concurrent Sweep 收集器, 它不会把堆(heap)分为不同的代(generation).

GC 使用完全并行标记(Full Parallel Mark), 它结合了暂停的转换器(Paused Mutator), GC 线程, 以及可选的标记线程(Marker Thread), 来处理标记队列(Mark Queue). 默认情况下, 暂停的转换器(Paused Mutator)和至少一个 GC 线程共通参与标记过程. 您可以使用 `-Xbinary=gcMarkSingleThreaded=true` 编译选项, 禁用完全并行标记(Full Parallel Mark). 但是, 这样做可能会增加垃圾收集器的暂停时间.

当标记阶段完成时, GC 会处理弱引用(Weak Reference), 并将指向未标记对象的引用(Unmarked Object)设置为 null. 为了减少 GC 的暂停时间, 你可以使用 `-Xbinary=concurrentWeakSweep=true` 编译选项, 禁用对弱引用(Weak Reference)的并发处理.

GC 在单独的线程中执行, 根据定时器和内存压力来启动. 此外, 也可以 手动调用.

手动启动垃圾收集

要强制启动垃圾收集器, 可以调用 `kotlin.native.internal.GC.collect()`. 这个方法会触发一次新的垃圾收集, 并等待它结束.

监测 GC 性能

目前没有专门的指标来监测 GC 性能. 但是, 可以查看 GC log 来进行问题诊断. 要启用 log, 请在 Gradle 构建脚本中设置以下编译选项:

```
-Xruntime-logs=gc=info
```

目前, log 只会被输出到 `stderr`.

禁用垃圾收集

我们推荐启用 GC. 但是, 某些情况下你也可以禁用它, 例如, 为了测试目的, 或者你遇到问题, 而且程序的生存周期很短. 要禁用 GC, 请在 Gradle 构建脚本中设置以下编译选项:

```
-Xgc=noop
```

> 使用这个选项后, GC 不会收集 Kotlin 对象, 因此只要程序继续运行, 内存消耗就会持续上升. 请注意, 不要耗尽系统内存.

内存消耗

Kotlin/Native 使用它自己的 内存分配器 (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/runtime/src/alloc/custom/README.md). 它将系统内存分为多个页面(Page), 允许按连续的顺序进行独立的清理. 每次分配的内存都会成为一个页面(Page)内的内存块(Memory Block), 并且页面会追踪各个块的大小. 各种不同的页面类型进行了不同的优化, 以适应于不同的内存分配大小. 内存块的连续排列保证了可以对所有的分配块进行高效的迭代.

当一个线程分配内存时, 它会根据分配的大小搜索适当的页面. 线程会根据不同的大小类别维护一组页面. 对于一个确定的大小, 当前页通常可以容纳这个内存分配. 如果不能, 那么线程会从共享的分配空间请求一个不同的页面. 这个页面的状态可能是可用, 需要清理, 或需要创建.

Kotlin/Native 内存分配器有一种保护功能, 可以防止突然激增的内存分配请求. 它可以防止转换器(Mutator)迅速的分配大量垃圾, 以至于 GC 线程无法处理, 导致内存使用量无限的增长. 在这种情况下, GC 会强制进入 Stop-the-World 阶段, 直到完成迭代.

你可以自己监控内存消耗, 检查内存泄漏, 并调整内存消耗.

检查内存泄露

要访问内存管理器的统计信息, 可以调用 `kotlin.native.internal.GC.lastGCInfo()`. 这个方法返回垃圾收集器最后一次运行的统计信息. 统计信息可以用于:

- 调试使用全局变量时的内存泄漏
- 在运行测试时检查是否存在内存泄漏

```
import kotlin.native.internal.*
import kotlin.test.*
```

```kotlin
class Resource

val global = mutableListOf<Resource>()

@OptIn(ExperimentalStdlibApi::class)
fun getUsage(): Long {
    GC.collect()
    return
GC.lastGCInfo!!.memoryUsageAfter["heap"]!!.totalObjectsSizeBytes
}

fun run() {
    global.add(Resource())
    // 如果删除下面这行，测试将会失败
    global.clear()
}

@Test
fun test() {
    val before = getUsage()
    // 这里使用一个单独的函数，确保所有的临时对象都被清除
    run()
    val after = getUsage()
    assertEquals(before, after)
}
```

调整内存消耗

如果程序中不存在内存泄露, 但你仍然观察到异常高的内存消耗, 请尝试将 Kotlin 更新到最新版本. 我们一直在持续改进内存管理器, 因此即使只是一次简单的编译器更新, 也可能改善你的程序的内存消耗情况.

更新 Kotlin 版本后, 如果您还是遇到内存消耗过高的情况, 可以选择以下几种解决方法:

- 在你的 Gradle 构建脚本中使用以下编译选项之一, 切换到不同的内存分配器:
  - `-Xallocator=mimalloc`, 使用 mimalloc (https://github.com/microsoft/mimalloc) 内存分配器.

    - `-Xallocator=std`, 使用系统的内存分配器.

- 如果你使用 mimalloc 内存分配器, 你可以命令它及时将内存释放回系统. 具体做法是, 在你的 `gradle.properties` 文件中启用以下二进制文件选项:

  ```
  kotlin.native.binary.mimallocUseCompaction=true
  ```

  这样的性能损失比较小, 但与系统的内存分配器相比, 它的结果比较不确定.

如果以上方法都不能改善内存消耗问题, 请到 YouTrack (https://youtrack.jetbrains.com/newissue?project=kt) 报告问题.

在后台进行单元测试

在单元测试中, 不会处理主线程队列, 因此, 除非 mock 过 `Dispatchers.Main`, 否则不要使用它. mock 它的方法是, 调用 `kotlinx-coroutines-test` 中的 `Dispatchers.setMain`.

如果你没有依赖于 `kotlinx.coroutines`, 或者因为某些原因 `Dispatchers.setMain` 不适合你的需求, 请使用以下变通方法, 实现测试启动器(test launcher):

```
package testlauncher

import platform.CoreFoundation.*
import kotlin.native.concurrent.*
import kotlin.native.internal.test.*
import kotlin.system.*

fun mainBackground(args: Array<String>) {
    val worker = Worker.start(name = "main-background")
    worker.execute(TransferMode.SAFE, { args.freeze() }) {
        val result = testLauncherEntryPoint(it)
        exitProcess(result)
    }
    CFRunLoopRun()
    error("CFRunLoopRun should never return")
}
```

然后, 使用 `-e testlauncher.mainBackground` 编译器选项来编译测试程序的二进制文件.

下一步

- 从旧的内存管理器迁移 (迁移到新的内存管理器)
- 与 iOS 集成的配置 (与 iOS 集成)

与 iOS 集成

最终更新: 2024/09/10

Kotlin/Native 垃圾收集器能够与 Swift/Objective-C ARC 无缝集成, 通常不需要额外的工作. 详情请参见 与 Swift/Objective-C 代码交互 (与 Swift/Objective-C 代码交互).

但是, 仍然有一些问题需要注意:

线程

销毁器(Deinitializer)

如果 Swift/Objective-C 对象和它们引用的对象, 在主线程中传递给 Kotlin/Native 代码, 那么对这些对象的销毁处理, 会在主线程中调用, 例如:

```
// Kotlin
class KotlinExample {
    fun action(arg: Any) {
        println(arg)
    }
}
```

```
// Swift
class SwiftExample {
    init() {
        print("init on \(Thread.current)")
    }

    deinit {
        print("deinit on \(Thread.current)")
    }
}

func test() {
    KotlinExample().action(arg: SwiftExample())
}
```

输出结果如下:

```
init on <_NSMainThread: 0x600003bc0000>{number = 1, name = main}
shared.SwiftExample
deinit on <_NSMainThread: 0x600003bc0000>{number = 1, name = main}
```

下面的情况下, Swift/Objective-C 对象的销毁处理会在一个特殊的 GC 线程中调用, 而不是在主线程中:

- Swift/Objective-C 对象在主线程之外的线程中传递给 Kotlin 代码.
- 主派发队列(main dispatch queue) 没有被处理.

如果你想要明确的在特殊的 GC 线程中调用销毁处理, 请在你的 `gradle.properties` 文件中设置 `kotlin.native.binary.objcDisposeOnMain=false`. 这个选项会允许在特殊的 GC 线程中调用销毁处理, 即使 Swift/Objective-C 对象在主线程中传递给 Kotlin 也是如此.

特殊的 GC 线程会与 Objective-C 运行库一起编译, 也就是说它拥有一个运行循环(run loop), 以及空的自动释放池(drain autorelease pool).

事件完成处理器(Completion handler)

从 Swift 中调用 Kotlin 挂起函数时, 事件完成处理器可能会在主线程之外的其它线程中调用, 例如:

```
// Kotlin
// coroutineScope, launch, 和 delay 都是 kotlinx.coroutines 中的函数
suspend fun asyncFunctionExample() = coroutineScope {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

```
// Swift
func test() {
    print("Running test on \(Thread.current)")
    PlatformKt.asyncFunctionExample(completionHandler: { _ in
        print("Running completion handler on \(Thread.current)")
```

```
    })
}
```

输出结果如下:

```
Running test on <_NSMainThread: 0x600001b100c0>{number = 1, name =
main}
Hello
World!
Running completion handler on <NSThread: 0x600001b45bc0>{number = 7,
name = (null)}
```

调用 Kotlin 挂起函数

如果从 Swift 和 Objective-C 的主线程以外的其它线程来调用 Kotlin 挂起函数, Kotlin/Native 内存管理器对于这样的情况存在限制.

这个限制最初是在旧的内存管理器出现的, 针对的是, 代码将挂起后的协程派发到原来的线程上恢复运行. 如果这个线程并没有支持的事件循环, 那么任务将无法运行, 因此协程永远无法恢复.

某些情况下, 不再需要这样的限制. 你可以在你的 `gradle.properties` 文件添加以下选项, 删除这个限制:

```
kotlin.native.binary.objcExportSuspendFunctionLaunchThreadRestrictio
n=none
```

垃圾收集与生存周期

对象回收

只有在垃圾收集期间对象才会被回收. 这个规则适用于跨越代码交互边界, 进入 Kotlin/Native 的 Swift/Objective-C 对象, 例如:

```
// Kotlin
class KotlinExample {
    fun action(arg: Any) {
        println(arg)
    }
}
```

```
// Swift
class SwiftExample {
    deinit {
        print("SwiftExample deinit")
    }
}

func test() {
    swiftTest()
    kotlinTest()
}

func swiftTest() {
    print(SwiftExample())
    print("swiftTestFinished")
}

func kotlinTest() {
    KotlinExample().action(arg: SwiftExample())
    print("kotlinTest finished")
}
```

输出结果如下:

```
shared.SwiftExample
SwiftExample deinit
swiftTestFinished
shared.SwiftExample
kotlinTest finished
SwiftExample deinit
```

Objective-C 对象生存周期

Objective-C 对象实际存在的时间可能比它应该存在的时间更长, 有时可能导致性能问题. 例如, 一个长时间运行的循环, 在每次循环时创建几个临时对象, 这些对象跨越 Swift/Objective-C 代码交互边界.

在 GC 日志 ("监测 GC 性能" in "Kotlin/Native 内存管理") 中, 有根对象集中稳定引用的数量. 如果这个数量持续增加, 可能代表 Swift/Objective-C 对象在需要释放的时候, 实际上没有被释放. 这种

情况下, 请在执行代码交互调用的循环体外部, 使用 `autoreleasepool` 代码段:

```
// Kotlin
fun growingMemoryUsage() {
    repeat(Int.MAX_VALUE) {
        NSLog("$it\n")
    }
}

fun steadyMemoryUsage() {
    repeat(Int.MAX_VALUE) {
        autoreleasepool {
            NSLog("$it\n")
        }
    }
}
```

Swift 与 Kotlin 对象链的垃圾收集

我们来看看下面的例子:

```
// Kotlin
interface Storage {
    fun store(arg: Any)
}

class KotlinStorage(var field: Any? = null) : Storage {
    override fun store(arg: Any) {
        field = arg
    }
}

class KotlinExample {
    fun action(firstSwiftStorage: Storage, secondSwiftStorage:
Storage) {
        // 这里, 我们创建下面的对象链:
        // firstKotlinStorage -> firstSwiftStorage ->
secondKotlinStorage -> secondSwiftStorage.
        val firstKotlinStorage = KotlinStorage()
```

```
        firstKotlinStorage.store(firstSwiftStorage)
        val secondKotlinStorage = KotlinStorage()
        firstSwiftStorage.store(secondKotlinStorage)
        secondKotlinStorage.store(secondSwiftStorage)
    }
}
```

```swift
// Swift
class SwiftStorage : Storage {

    let name: String

    var field: Any? = nil

    init(_ name: String) {
        self.name = name
    }

    func store(arg: Any) {
        field = arg
    }

    deinit {
        print("deinit SwiftStorage \(name)")
    }
}

func test() {
    KotlinExample().action(
        firstSwiftStorage: SwiftStorage("first"),
        secondSwiftStorage: SwiftStorage("second")
    )
}
```

在 log 中输出 "deinit SwiftStorage first" 和 "deinit SwiftStorage second" 消息之间, 会存在一些间隔时间. 原因是, `firstKotlinStorage` 和 `secondKotlinStorage` 会被不同的 GC 周期回收. 事件序列如下:

1. `KotlinExample.action` 结束. `firstKotlinStorage` 被认为处于 "dead" 状态, 因为没有任何对象引用它, 而 `secondKotlinStorage` 还不是 "dead" 状态, 因为它被 `firstSwiftStorage` 引用.
2. 第 1 次 GC 周期开始, `firstKotlinStorage` 被回收.
3. 没有对象引用 `firstSwiftStorage`, 因此它也处于 "dead" 状态, 并且调用它的 `deinit`.
4. 第 2 次 GC 周期开始. `secondKotlinStorage` 被回收, 因为 `firstSwiftStorage` 不再引用它.
5. 最后, `secondSwiftStorage` 被回收.

需要 2 次 GC 周期才能回收这 4 个对象, 因为 Swift 和 Objective-C 对象的销毁过程发生在 GC 周期之后. 这个限制是由于 `deinit` 造成的, 它可以调用任意的代码, 包括在 GC 造成的应用程序暂停时期无法运行的 Kotlin 代码.

支持后台状态和 App 扩展

目前的内存管理器默认不追踪应用程序状态, 而且没有集成 App 扩展 (https://developer.apple.com/app-extensions/).

因此, 内存管理器不会相应的调整 GC 行为, 有些情况下可能造成问题. 要改变这个行为, 请向你的 `gradle.properties` 添加下面的 实验性 (Kotlin 各部分组件的稳定性) 二进制选项:

```
kotlin.native.binary.appStateTracking=enabled
```

这个选项会在应用程序处于后台状态时关闭对垃圾收集器的定时调用, 因此只有当内存消耗量过高时才会调用 GC.

迁移到新的内存管理器

最终更新: 2024/09/10

本向导会对新的 Kotlin/Native 内存管理器 (Kotlin/Native 内存管理) 与旧的内存管理器进行比较, 并介绍如何迁移你的项目.

新内存管理器最重要的变化是解除了对象共享的限制. 在线程之间共享对象时, 你不需要冻结对象, 具体来说:

- 顶层属性不需要标注 `@SharedImmutable`, 可以被任意线程访问和修改.
- 通过代码交互传递的对象不需要冻结, 可以被任意线程访问和修改.
- `Worker.executeAfter` 不再要求其中的操作被冻结.
- `Worker.execute` 不再需要生成者返回一个孤立的对象子图(subgraph).
- 包含 `AtomicReference` 和 `FreezableAtomicReference` 的环形引用不会导致内存泄露.

除了对象共享更加容易之外, 新的内存管理器还带来了其他主要变化:

- 全局属性会在定义它们的文件被初次访问时延迟执行它们的初始化. 以前全局属性会在程序启动时初始化. 作为一个变通方法, 你可以使用 `@EagerInitialization` 注解, 将属性标注为必须在程序启动时初始化. 使用这个注解之前, 请先阅读它的 文档 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native/-eager-initialization/).
- `by lazy {}` 属性支持线程安全模式, 而且不处理无限递归.
- 从 `Worker.executeAfter` 的 `operation` 中抛出的异常, 会和运行时的其它部分一样进行处理, 尝试执行一个用户自定义的、未被处理的异常处理程序, 如果没有找到异常处理程序, 或异常处理程序自身也抛出异常而失败, 则终止程序.
- 冻结功能已被废弃, 默认禁用, 会在将来的发布版中删除. 如果你不需要你的代码在 旧的内存管理器 下工作, 请不要使用冻结.

要从旧的内存管理器迁移你的项目, 请遵循下面的步骤:

更新 Kotlin

从 Kotlin 1.7.20 开始会默认启用新的 Kotlin/Native 内存管理器. 请检查 Kotlin 版本, 如果需要的话, 请 更新到最新版 ("更新到新的发布版" in "Kotlin 的发布版本").

更新依赖项

kotlinx.coroutines	更新到 1.6.0 或更高版本. 不要使用带 native-mt 后缀的版本. 关于新内存管理器, 还有一些需要注意的问题: • 所有的基本元素(通道(Channel), 数据流(Flow), 协程(Coroutine)) 都可以跨越 Worker 边界工作, 因为不再需要冻结. • Dispatchers.Default 在 Linux 和 Windows 上通过 Worker 池来实现, 在 Apple 目标平台上则通过全局队列来实现. • 可以使用 newSingleThreadContext 来创建依靠单个 Worker 实现的协程派发器. • 可以使用 newFixedThreadPoolContext 来创建依靠 N 个 Worker 的池实现的协程派发器. • Dispatchers.Main 在 Darwin 上依靠主队列实现, 在其它平台依靠独立的 Worker 来实现.
Ktor	更新到 2.0 或更高版本.
其他依赖项	大多数库应该能够平滑升级, 但可能存在少量例外. 请确认你的依赖项更新到了最新版本, 针对旧的和新的内存管理器的库版本没有差别.

更新你的代码

要支持新的内存管理器, 请删除对受影响的 API 的调用:

旧 API	应该如何更新
@SharedImmutable (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-shared-immutable/)	你可以删除所有使用它的代码, 尽管在新的内存管理器中使用这个 API 也没有警告.
FreezableAtomicReference 类 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-freezable-atomic-reference/)	请改为使用 AtomicReference (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-atomic-reference/).
FreezingException 类 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-freezing-exception/)	删除所有使用它的代码.
InvalidMutabilityException 类 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-invalid-mutability-exception/)	删除所有使用它的代码.
IncorrectDereferenceException 类 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native/-incorrect-dereference-exception/)	删除所有使用它的代码.
freeze() 函数 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/freeze.html)	删除所有使用它的代码.
isFrozen 属性 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/is-frozen.html)	你可以删除所有使用它的代码. 由于冻结功能已被废弃, 这个属性永远返回 false.
ensureNeverFrozen() 函数 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.	删除所有使用它的代码.

concurrent/ensure-never-frozen.html)	
atomicLazy() 函数 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/atomic-lazy.html)	请改为使用 lazy() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/lazy.html).
MutableData 类 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-mutable-data/)	请改为使用通常的集合.
WorkerBoundReference<out T : Any> 类 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-worker-bound-reference/)	请直接使用 T.
DetachedObjectGraph<T> 类 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-detached-object-graph/)	请直接使用 T. 要通过 C 代码交互传递值, 请使用 StableRef 类 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlinx.cinterop/-stable-ref/).

同时支持新的和旧的内存管理器

如果你是库的作者, 需要维护你的代码支持旧的内存管理器, 或者在新的内存管理器出现问题时希望能够回退到旧的内存管理器, 你可以临时的支持新的和旧的内存管理器.

要忽略编译器的废弃警告, 请进行以下任何一种操作:

- 使用已废弃的 API 时添加 @OptIn(FreezingIsDeprecated::class) 注解.
- 在 Gradle 中, 对所有的 Kotlin 源代码集添加 languageSettings.optIn("kotlin.native.FreezingIsDeprecated").
- 传递编译器选项 -opt-in=kotlin.native.FreezingIsDeprecated.

详情请参见 明确要求使用者同意的功能 (明确要求使用者同意的功能(Opt-in Requirement)).

下一步做什么

* 关于新的内存管理器 (Kotlin/Native 内存管理)
* 配置与 iOS 的集成 (与 iOS 集成)

调试 Kotlin/Native 代码

最终更新: 2024/09/10

Kotlin/Native 编译器目前输出的调试信息兼容于 DWARF 2 规范, 因此现代的调试工具可以执行以下操作:

- 设置断点
- 单步执行
- 查看类型信息
- 查看变量

> 支持 DWARF 2 规范就意味着调试器会把 Kotlin 程序识别为 C89, 因为在 DWARF 5 规范之前, 还没有标识符可以标识语言类型是 Kotlin.

使用 Kotlin/Native 编译器输出带调试信息的二进制文件

要让 Kotlin/Native 编译器输出带调试信息的二进制文件, 请在命令行添加 `-g` 选项.

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat - > hello.kt
fun main(args: Array<String>) {
  println("Hello world")
  println("I need your clothes, your boots and your motocycle")
}
0:b-debugger-fixes:minamoto@unit-703(0)# dist/bin/konanc -g hello.kt
-o terminator
KtFile: hello.kt
0:b-debugger-fixes:minamoto@unit-703(0)# lldb terminator.kexe
(lldb) target create "terminator.kexe"
Current executable set to 'terminator.kexe' (x86_64).
(lldb) b kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 1: where =
terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at
hello.kt:2, address = 0x00000001000012e4
```

```
(lldb) r
Process 28473 launched: '/Users/minamoto/ws/.git-trees/debugger-
fixes/terminator.kexe' (x86_64)
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason =
breakpoint 1.1
    frame #0: 0x00000001000012e4
terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at hello.kt:2
   1    fun main(args: Array<String>) {
-> 2      println("Hello world")
   3      println("I need your clothes, your boots and your
motocycle")
   4    }
(lldb) n
Hello world
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step
over
    frame #0: 0x00000001000012f0
terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at hello.kt:3
   1    fun main(args: Array<String>) {
   2      println("Hello world")
-> 3      println("I need your clothes, your boots and your
motocycle")
   4    }
(lldb)
```

断点

现代调试器提供了多种方法可以设置断点, 各种调试工具的具体方法请看下文:

lldb

- 通过名称设置断点

  ```
  (lldb) b -n kfun:main(kotlin.Array<kotlin.String>)
  Breakpoint 4: where =
  ```

```
terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at
hello.kt:2, address = 0x00000001000012e4
```

`-n` 参数是可选的, 这个参数默认会启用

- 通过位置 (文件名, 行号) 设置断点

```
(lldb) b -f hello.kt -l 1
Breakpoint 1: where =
terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at
hello.kt:2, address = 0x00000001000012e4
```

- 通过地址设置断点

```
(lldb) b -a 0x00000001000012e4
Breakpoint 2: address = 0x00000001000012e4
```

- 通过正规表达式设置断点, 调试编译器生成的代码时, 你可能会发现这个功能很有用, 比如 Lambda 表达式, 等等. (因为它的名称中使用了 `#` 符号).

```
3: regex = 'main\(', locations = 1
  3.1: where =
terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at
hello.kt:2, address = terminator.kexe[0x00000001000012e4],
unresolved, hit count = 0
```

gdb

- 通过正规表达式设置断点

```
(gdb) rbreak main(
Breakpoint 1 at 0x1000109b4
struct ktype:kotlin.Unit &kfun:main(kotlin.Array<kotlin.String>);
```

- **不能** 通过名称设置断点, 因为名称中的 `:` 字符, 会被看作是通过位置设置断点目录命令的一个分隔符

```
(gdb) b kfun:main(kotlin.Array<kotlin.String>)
No source file named kfun.
Make breakpoint pending on future shared library load? (y or [n])
y
Breakpoint 1 (kfun:main(kotlin.Array<kotlin.String>)) pending
```

- 通过位置设置断点

```
(gdb) b hello.kt:1
Breakpoint 2 at 0x100001704: file /Users/minamoto/ws/.git-
trees/hello.kt, line 1.
```

- 通过地址设置断点

```
(gdb) b *0x100001704
Note: breakpoint 2 also set at pc 0x100001704.
Breakpoint 3 at 0x100001704: file /Users/minamoto/ws/.git-
trees/hello.kt, line 2.
```

单步调试

单步调试功能的使用方法与大多数 C/C++ 程序一样.

查看变量

对于 `var` 变量的查看功能, 对于基本类型是直接可用的. 对于非基本类型, 可以使用 `konan_lldb.py` 中针对 lldb 的自定义格式化工具:

```
λ cat main.kt | nl
     1  fun main(args: Array<String>) {
     2      var x = 1
     3      var y = 2
     4      var p = Point(x, y)
     5      println("p = $p")
     6  }

     7  data class Point(val x: Int, val y: Int)
```

```
λ lldb ./program.kexe -o 'b main.kt:5' -o
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b main.kt:5
Breakpoint 1: where =
program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 289 at
main.kt:5, address = 0x000000000040af11
(lldb) r
Process 4985 stopped
* thread #1, name = 'program.kexe', stop reason = breakpoint 1.1
    frame #0: program.kexe`kfun:main(kotlin.Array<kotlin.String>) at
main.kt:5
   2        var x = 1
   3        var y = 2
   4        var p = Point(x, y)
-> 5        println("p = $p")
   6    }
   7
   8    data class Point(val x: Int, val y: Int)

Process 4985 launched: './program.kexe' (x86_64)
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = 0x00000000007643d8
(lldb) command script import dist/tools/konan_lldb.py
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = [x: ..., y: ...]
(lldb) p p
(ObjHeader *) $2 = [x: ..., y: ...]
(lldb) script
lldb.frame.FindVariable("p").GetChildMemberWithName("x").Dereference
().GetValue()
'1'
(lldb)
```

把对象变量转换为易于阅读的字符串表达形式, 也可以使用内建的运行期函数 `Konan_DebugPrint` 来实现 (这个方法也适用于 gdb, 使用 command 语法中的一个模块):

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat ../debugger-plugin/1.kt
| nl -p
     1  fun foo(a:String, b:Int) = a + b
     2  fun one() = 1
     3  fun main(arg:Array<String>) {
     4    var a_variable = foo("(a_variable) one is ", 1)
     5    var b_variable = foo("(b_variable) two is ", 2)
     6    var c_variable = foo("(c_variable) two is ", 3)
     7    var d_variable = foo("(d_variable) two is ", 4)
     8    println(a_variable)
     9    println(b_variable)
    10    println(c_variable)
    11    println(d_variable)
    12  }
0:b-debugger-fixes:minamoto@unit-703(0)# lldb ./program.kexe -o 'b -
f 1.kt -l 9' -o r
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b -f 1.kt -l 9
Breakpoint 1: where =
program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 463 at 1.kt:9,
address = 0x0000000100000dbf
(lldb) r
(a_variable) one is 1
Process 80496 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason =
breakpoint 1.1
    frame #0: 0x0000000100000dbf
program.kexe`kfun:main(kotlin.Array<kotlin.String>) at 1.kt:9
   6      var c_variable = foo("(c_variable) two is ", 3)
   7      var d_variable = foo("(d_variable) two is ", 4)
   8      println(a_variable)
-> 9      println(b_variable)
   10     println(c_variable)
   11     println(d_variable)
   12   }
```

```
Process 80496 launched: './program.kexe' (x86_64)
(lldb) expression -- (int32_t)Konan_DebugPrint(a_variable)
(a_variable) one is 1(int32_t) $0 = 0
(lldb)
```

已知的问题

- Python 绑定的性能问题.

符号化(Symbolicate) iOS 崩溃报告(Crash Report)

最终更新: 2024/09/10

要对 iOS 应用程序的崩溃进行调试, 有时需要分析崩溃报告. 关于崩溃报告的详情, 请参见 Apple 文档 (https://developer.apple.com/library/archive/technotes/tn2151/_index.html).

崩溃报告通常需要经过符号化(symbolication), 才能供人类正确阅读: 符号化会将机器码地址转换为适合人阅读的源代码位置. 本文将介绍使用 Kotlin 开发 iOS 应用程序时, 符号化崩溃报告的一些具体细节.

对 release 版的 Kotlin 二进制文件产生 .dSYM 文件

要符号化 Kotlin 代码中的地址(比如, 得到调用栈各层分别对应的 Kotlin 源代码), 需要使用 Kotlin 代码的 `.dSYM` bundle.

默认情况下, 在 Darwin 平台上, 对 release版的(也就是, 优化过的) 二进制文件, Kotlin/Native 编译器会产生 `.dSYM` 文件. 这个选项可以使用编译器参数 `-Xadd-light-debug=disable` 来关闭. 同时, 对于其他平台, 这个选项默认是关闭的. 如果需要打开, 请使用编译器选项 `-Xadd-light-debug=enable`.

Kotlin

```
kotlin {

targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

Groovy

```
kotlin {

targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNa
tiveTarget) {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug=
{enable|disable}"
        }
    }
}
```

在 IntelliJ IDEA 或 AppCode 模板创建的项目中, 这些 .dSYM bundle 之后会被 Xcode 自动发现.

从 bitcode 重新构建时, 将框架(Framework)设置为静态

将 Kotlin 产生的框架从 bitcode 重新构建时, 会使得原来的 .dSYM 失效. 如果是在本地重新构建, 请注意, 符号化崩溃报告时一定要使用更新后的 .dSYM.

如果重新构建在 App Store 端进行, 那么重新构建的 *动态(dynamic)* 框架的 .dSYM 似乎会被抛弃, 而且无法从 App Store 下载. 这种情况下, 可能需要将框架设置为静态.

Kotlin

```
kotlin {

targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNa
tiveTarget> {

binaries.withType<org.jetbrains.kotlin.gradle.plugin.mpp.Framewo
rk> {
            isStatic = true
        }
    }
}
```

Groovy

```
kotlin {

targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNa
tiveTarget) {

binaries.withType(org.jetbrains.kotlin.gradle.plugin.mpp.Framewo
rk) {
            isStatic = true
        }
    }
}
```

Kotlin/Native 支持的目标平台

最终更新: 2024/09/10

Kotlin/Native 编译器支持大量的编译目标, 但是, 很难对所有编译目标提供同等程度的支持. 本文档描述 Kotlin/Native 支持哪些编译目标, 并根据编译器支持程度的不同, 将它们分为几个层级.

> ⚠ 未来我们可能会调整层级的数量, 支持的编译目标, 以及它们的功能特性.

请注意层级列表中使用到的以下名词:

- **Gradle 编译目标名称** 是一个 编译目标预设定(preset) (为 Kotlin Multiplatform 设置编译目标), 在 Kotlin Multiplatform Gradle plugin 中使用它来启用编译目标.
- **Target triple** 是一个符合 `<architecture>-<vendor>-<system>-<abi>` 格式的编译目标名称, 通常由 编译器 (https://clang.llvm.org/docs/CrossCompilation.html#target-triple) 使用.
- **运行测试** 表示是否默认支持在 Gradle 和 IDE 中运行测试.

  对于特定的编译目标, 运行测试只在原生主机上有效. 例如, 你只能在 macOS x86-64 主机上运行 `macosX64` 和 `iosX64` 测试.

第 1 层

- 编译目标在 CI 环境进行过常规测试, 保证能够编译和运行.
- 我们在编译器发布版之间提供源代码和 二进制兼容性 (https://youtrack.jetbrains.com/issue/KT-42293).

Gradle 编译目标名称	Target triple	运行测试	备注
`linuxX64`	`x86_64-unknown-linux-gnu`	✅	x86_64 平台上的 Linux
以下仅限于 Apple macOS 主机:			
`macosX64`	`x86_64-apple-macos`	✅	x86_64 平台上的 Apple macOS
`macosArm64`	`aarch64-apple-macos`	✅	Apple Silicon 平台上的 Apple macOS
`iosSimulatorArm64`	`aarch64-apple-ios-simulator`	✅	Apple Silicon 平台上的 Apple iOS 模拟器
`iosX64`	`x86_64-apple-ios-simulator`	✅	x86-64 平台上的 Apple iOS 模拟器

第 2 层

- 编译目标在 CI 环境进行过常规测试, 保证能够编译, 但可能没有进行过自动测试, 保证能够运行.
- 我们尽最大努力来保证在编译器发布版之间提供源代码和 二进制兼容性 (https://youtrack.jetbrains.com/issue/KT-42293).

Gradle 编译目标名称	Target triple	运行测试	Description
linuxX64	x86_64-unknown-linux-gnu	✅	x86_64 平台上的 Linux
linuxArm64	aarch64-unknown-linux-gnu		ARM64 平台上的 Linux
以下仅限于 Apple macOS 主机:			
watchosSimulatorArm64	aarch64-apple-watchos-simulator	✅	Apple Silicon 平台上的 Apple watchOS 模拟器
watchosX64	x86_64-apple-watchos-simulator	✅	x86_64 平台上的 Apple watchOS 64-bit 模拟器
watchosArm32	armv7k-apple-watchos		ARM32 平台上的 Apple watchOS
watchosArm64	arm64_32-apple-watchos		ARM64 平台上的 with ILP32 Apple watchOS
tvosSimulatorArm64	aarch64-apple-tvos-simulator	✅	Apple Silicon 平台上的 Apple tvOS 模拟器
tvosX64	x86_64-apple-tvos-simulator	✅	x86_64 平台上的 Apple tvOS 模拟器
tvosArm64	aarch64-apple-tvos		ARM64 平台上的 Apple tvOS
iosArm64	aarch64-apple-ios		ARM64 平台上的 Apple iOS 和 iPadOS

> ⚠ 我们正在尽最大努力将 `iosArm64` 移动到第 1 层, 因为它对于 Kotlin Multiplatform (Kotlin 跨平台程序开发入门) 是一个至关重要的编译目标. 为了达到这个目的, 我们首先需要创建一个专用的测试平台, 因为平台的限制使得在 Apple 设备上很难运行编译器测试.
>
> 目前, 我们有时会在 iOS 设备上手动运行测试, 而且依赖于类似编译目标的测试, 例如 `iosSimulatorArm64`, 对大多数情况来说已经足够了.

第 3 层

- 编译目标不保证能够在 CI 环境中测试.
- 我们不能在不同的编译器发布版之间保证源代码和二进制兼容性, 但是, 对这些编译目标的不兼容变更极少发生.

Gradle 编译目标名称	Target triple	运行测试	Description
androidNativeArm32	arm-unknown-linux-androideabi		ARM32 平台上的 Android NDK (https://developer.android.com/ndk)
androidNativeArm64	aarch64-unknown-linux-android		ARM64 平台上的 Android NDK (https://developer.android.com/ndk)
androidNativeX86	i686-unknown-linux-android		x86 平台上的 Android NDK (https://developer.android.com/ndk)
androidNativeX64	x86_64-unknown-linux-android		x86_64 平台上的 Android NDK (https://developer.android.com/ndk)
mingwX64	x86_64-pc-windows-gnu	✅	Windows 7 和之后版本上的 64 位 MinGW (https://www.mingw-w64.org)
以下仅限于 Apple macOS 主机:			
watchosDeviceArm64	aarch64-apple-watchos		ARM64 平台上的 Apple watchOS

linuxArm32Hfp 编译目标已被废弃, 将在未来的发布版中删除.

针对库的开发者

对于库的开发者, 我们不推荐测试比 Kotlin/Native 编译器更多的编译目标, 也不推荐支持比 Kotlin/Native 编译器更严格的兼容性保证. 在考虑支持原生编译目标时, 你可以使用以下方案:

- 支持第 1 层, 第 2 层, 第 3 层的全部编译目标.
- 第 1 层和第 2 层中的, 默认支持运行测试的常规测试编译目标.

Kotlin 开发组在 Kotlin 官方库的开发中也使用这个方案, 例如, kotlinx.coroutines (协程指南) 和 kotlinx.serialization (序列化).

改进 Kotlin/Native 编译速度

最终更新: 2024/09/10

Kotlin/Native 编译器正在不断更新, 并改进它的性能. 使用最新的 Kotlin/Native 编译器, 以及正确配置的构建环境, 对于使用 Kotlin/Native 编译目标的项目, 你可以显著的改进编译速度.

关于如何提高 Kotlin/Native 编译过程速度, 请阅读我们的建议.

一般性建议

- **使用最新版本的 Kotlin**. 这样你可以得到最新的性能改进.
- **不要创建巨大的类**. 这样的类会需要很长时间来编译, 执行期的装载时间也很长.
- **在多次构建之间, 保留下载的和缓存的组件**. 编译项目时, Kotlin/Native 会下载需要的组件, 并将它的一些工作结果缓存到 `$USER_HOME/.konan` 目录. 编译器会在后续的编译中使用这个目录, 使编译工作更快完成.

  在容器(比如 Docker)内或者使用持续集成系统(Continuous Integration)构建时, 编译器可能在每次构建时都需要重新创建 `~/.konan` 目录. 为了避免这样的步骤, 请配置你的环境, 使其在多次构建之间保留 `~/.konan`. 比如, 使用 Gradle 属性 `kotlin.data.dir` 来重新定义它的位置.

  或者, 也可以使用 `-Xkonan-data-dir` 编译器选项, 通过 `cinterop` 和 `konanc` 工具来配置你的的自定义目录路径.

Gradle 配置

使用 Gradle 的第 1 次编译通常会比之后的构建耗费更多时间, 因为需要下载依赖项目, 构建缓存, 并执行一些额外步骤. 你应该构建你的项目至少 2 次, 才能得到准确的编译时间.

下面是如何配置 Gradle 改善编译性能的一些建议:

- **增大 Gradle heap 尺寸 (https://docs.gradle.org/current/userguide/performance.html#adjust_the_daemons_heap_size)**. 向 `gradle.properties` 添加 `org.gradle.jvmargs=-Xmx3g` 设置. 如果你使用 并行构建 (https://docs.gradle.org/current/userguide/performance.html#parallel_execution) 功能, 你可能需要使用 `org.gradle.workers.max` 属性或 `--max-workers` 命令行选项, 来选择正确的 worker 数量. 默认值是 CPU 的处理器个数.

- **只构建你需要的二进制文件**. 除非你真的需要这样, 否则不要运行构建整个项目的 Gradle 任务, 比如 `build` 或 `assemble`. 这样的任务会多次构建相同的代码, 增加编译时间. 典型情况下, 比如在 IntelliJ IDEA 中运行测试, 或从 Xcode 启动应用程序, Kotlin 工具会避免执行不必要的 Gradle 任务.

  如果你遇到比较特殊的场景, 或使用了特殊的构建配置, 你可能需要自行选择 Gradle 任务.

  - `linkDebug*`: 为了在开发期间运行你的代码, 你通常只需要一个二进制文件, 因此只运行对应的 `linkDebug*` 任务通常就够了. 请记住, 编译一个证实发布版的二进制文件 (`linkRelease*`) 会比编译一个调试版本耗费更多时间.

  - `packForXcode`: 由于 iOS 各种模拟器和各种真实设备使用不同的处理器架构, 因此通常会将 Kotlin/Native 二进制文件以 universal (fat) 框架的形式发布. 在本地开发时, 只为你正在使用的平台构建 `.framework` 会比较快.

    要构建一个平台专用的框架, 请调用 Kotlin Multiplatform 项目向导 (https://kmp.jetbrains.com/) 创建的 `packForXcode` 任务.

    > 请记住, 这种情况下, 在设备和模拟器之间切换之后, 你将会需要使用 `./gradlew clean` 清除构建. 详情请参见 这个问题 (https://youtrack.jetbrains.com/issue/KT-40907).

- **不要禁用 Gradle daemon (https://docs.gradle.org/current/userguide/gradle_daemon.html)**, 如果没有重要的原因, 请不要这样做. 默认情况下 Kotlin/Native 从 Gradle daemon 启动 (https://blog.jetbrains.com/kotlin/2020/03/kotlin-1-3-70-released/#kotlin-native). Gradle daemon 启用时, 会使用相同的 JVM 进程, 因此不必为每次编译重新最准备.

- **不要使用 transitiveExport = true ("将依赖项目导出到二进制文件" in "构建最终的原生二进制文件")**. 使用传递导出(Transitive Export)很多情况下会导致死代码剔除(Dead Code Elimination) 功能被关闭: 编译器必须处理很多未使用的代码. 这样会增加编译时间. 要明确使用 `export`, 来导出需要的项目和依赖项.

- **使用 Gradle 的 构建缓存 (https://docs.gradle.org/current/userguide/build_cache.html)**:

  - **本地构建缓存**: 向你的 `gradle.properties` 文件添加设置 `org.gradle.caching=true`, 或者在命令行运行时添加 `--build-cache` 参数.

  - **远程构建缓存** 用于持续集成环境. 详情请参见 配置远程构建缓存 (https://docs.gradle.org/current/userguide/build_cache.html#sec:build_cache_config

ure_remote).

- **启用以前禁用的 Kotlin/Native 功能**. 有些属性会禁用 Gradle daemon 和编译器缓存 – `kotlin.native.disableCompilerDaemon=true` 和 `kotlin.native.cacheKind=none`. 如果你过去曾遇到与这些功能相关的问题, 并向你的 `gradle.properties` 文件或 Gradle 命令行添加过这些参数, 请删除这些参数, 再次检查构建是否成功. 有可能以前添加过这些属性来绕过某些问题, 但这些问题现在已经解决了.

- **尝试使用 klib artifact 的增量编译功能**. 使用增量编译时, 如果项目模块产生的 `klib` artifact 只发生了部分变更, 那么只有 `klib` 的一部分会被重新编译为二进制文件.

  这个功能是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 要启用这个功能, 请向你的 `gradle.properties` 文件添加 `kotlin.incremental.native=true` 选项. 如果你遇到问题, 请 在 YouTrack 中创建 issue (https://kotl.in/issue).

Windows OS 配置

- **配置 Windows Security**. Windows Security 可能会让 Kotlin/Native 编译器变慢. 为了避免这样的情况, 你可以将 `.konan` 目录添加到 Windows Security 的排除项目, 这个目录默认在 `%USERPROFILE%` 下. 详情请参见 添加 Windows Security 的排除项目 (https://support.microsoft.com/en-us/windows/add-an-exclusion-to-windows-security-811816c0-4dfd-af4a-47e4-c301afe13b26).

Kotlin/Native 二进制文件的许可证

最终更新: 2024/09/10

和其它很多开源项目一样, Kotlin 依赖于第三方代码, 也就是说, Kotlin 项目包含一部分并不是由 JetBrains 或 Kotlin 编程语言贡献者们开发的代码. 有时这些代码是派生作品, 例如将 C++ 代码重写为 Kotlin.

> 你可以在我们的 GitHub 仓库中找到 Kotlin 中使用的第三方作品的许可证:
>
> - Kotlin 编译器 (https://github.com/JetBrains/kotlin/tree/master/license/third_party)
> - Kotlin/Native (https://github.com/JetBrains/kotlin/tree/master/kotlin-native/licenses/third_party)

具体来说, Kotlin/Native 编译器生成的二进制文件, 其中可能包含第三方代码, 数据或派生作品. 这意味着, Kotlin/Native 编译的二进制文件, 受第三方许可证的条款和条件的约束.

具体来说, 如果你分发一个 Kotlin/Native 编译的 最终二进制文件 (构建最终的原生二进制文件), 你应该始终在你的二进制发行版中包含必要的许可证文件. 这些文件应该以你的发行版的使用者可以读取形式访问.

对于相应项目, 请你始终包含以下许可证文件:

项目	需要包含的文件
Kotlin (https://kotlinlang.org/)	• Apache license 2.0 (https://github.com/JetBrains/kotlin/blob/master/license/LICENSE.txt) • Apache Harmony 版权声明 (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/licenses/third_party/harmony_NOTICE.txt)
Apache Harmony (https://harmony.apache.org/)	
GWT (https://www.gwtproject.org/)	
Guava (https://guava.dev)	
libbacktrace (https://github.com/ianlancetaylor/libbacktrace)	带有版权声明的 3-clause BSD license (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/licenses/third_party/libbacktrace_LICENSE.txt)
mimalloc (https://github.com/microsoft/mimalloc)	MIT license (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/licenses/third_party/mimalloc_LICENSE.txt) 如果你使用 mimaloc 内存分配器而不是默认分配器(设置了 `-Xallocator=mimalloc` 编译器选项), 请包含这个许可证文件. 关于内存分配器, 更多详情请参见 Kotlin/Native 内存管理 (Kotlin/Native 内存管理)
Unicode character database (https://www.unicode.org/)	Unicode license (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/licenses/third_party/unicode_LICENSE.txt)
Multi-producer/multi-consumer bounded queue	版权声明 (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/licenses/third_party/mpmc_queue_LICENSE.txt)

`mingwX64` 编译目标还要求额外的许可证文件:

项目	需要包含的文件
MinGW-w64 头文件和运行时库 (https://www.mingw-w64.org/)	• MinGW-w64 运行时许可证 (https://sourceforge.net/p/mingw-w64/mingw-w64/ci/master/tree/COPYING.MinGW-w64-runtime/COPYING.MinGW-w64-runtime.txt) • Winpthreads license (https://sourceforge.net/p/mingw-w64/mingw-w64/ci/master/tree/mingw-w64-libraries/winpthreads/COPYING)

注意, 这些库要求你分发的 Kotlin/Native 二进制文件开源.

Kotlin/Native FAQ

最终更新: 2024/09/10

怎样运行程序?

你需要定义一个顶层的函数 `fun main(args: Array<String>)`, 如果你不需要接受命令行参数, 也可以写成 `fun main()`, 请注意不要把这个函数放在包内. 另外, 也可以使用编译器的 `-entry` 选项, 把任何一个函数指定为程序的入口点, 但这个函数应该接受 `Array<String>` 参数, 或者没有参数, 并且函数返回值类型应该是 `Unit`.

Kotlin/Native 的内存管理机制是怎样的?

Kotlin/Native 使用一种自动化的内存管理机制, 与 Java 和 Swift 类似.

详情请参见 Kotlin/Native 内存管理器 (Kotlin/Native 内存管理)

怎样创建一个共享库?

可以使用编译器的 `-produce dynamic` 选项, 或在 Gradle 中使用 `binaries.sharedLib()`.

```
kotlin {
    iosArm64("mylib") {
        binaries.sharedLib()
    }
}
```

编译器会产生各平台专有的共享库文件 (对 Linux 环境 `.so` 文件, 对 macOS 环境是 `.dylib` 文件, 对 Windows 环境是 `.dll` 文件), 还会生成一个 C 语言头文件, 用来在 C/C++ 代码中访问你的 Kotlin/Native 程序中的所有 public API.

怎样创建静态库, 或 object 文件?

可以使用编译器的 `-produce static` 选项, 或在 Gradle 中使用 `binaries.staticLib()`.

```
kotlin {
    iosArm64("mylib") {
        binaries.staticLib()
```

```
    }
}
```

编译器会产生各平台专有的 object 文件(.a 库格式), 以及一个 C 语言头文件, 用来在 C/C++ 代码中访问你的 Kotlin/Native 程序中的所有 public API.

怎样在企业的网络代理服务器之后运行 Kotlin/Native?

由于 Kotlin/Native 需要下载各平台相关的工具链, 因此你需要对编译器或 `gradlew` 设置 `-Dhttp.proxyHost=xxx -Dhttp.proxyPort=xxx` 选项, 或者通过 `JAVA_OPTS` 环境变量来设置这个选项.

怎样为 Kotlin 框架指定自定义的 Objective-C 前缀或名称?

可以使用编译器的 `-module-name` 选项, 或对应的 Gradle DSL 语句.

Kotlin

```
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += listOf("-module-name",
"TheName")
        }
    }
}
```

Groovy

```
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += ["-module-name", "TheName"]
        }
    }
}
```

怎样改变 iOS 框架的名称?

iOS 框架的默认名称是 `<project name>.framework`. 要使用自定义的名称, 请使用 `baseName` 选项. 这个选项也会设置模块的名称.

```
kotlin {
    iosArm64("myapp") {
       binaries {
          framework {
              baseName = "TheName"
          }
       }
    }
}
```

怎样对 Kotlin 框架启用 bitcode?

gradle plugin 默认会将 bitcode 添加到 iOS 编译目标中.

- 对于 debug 版, gradle plugin 会将 LLVM IR 数据占位器(placeholder)作为标记(marker)嵌入.
- 对于 release 版, gradle plugin 会将 bitcode 作为数据嵌入.

或者使用编译器参数: `-Xembed-bitcode` (用于 release 版) 和 `-Xembed-bitcode-marker` (用于 debug 版)

使用 Gradle DSL 的设置如下:

```
kotlin {
    iosArm64("myapp") {
        binaries {
            framework {
                // 使用 "marker" 嵌入 bitcode 标记 (用于 debug 版构建).
                // 使用 "disable" 关闭嵌入.
                embedBitcode("bitcode") // 用于 release 版构建.
            }
        }
```

```
    }
}
```

这个选项的效果几乎等于 clang 的 -fembed-bitcode/-fembed-bitcode-marker 和 swiftc 的 -embed-bitcode/-embed-bitcode-marker.

为什么会遇到 InvalidMutabilityException 异常?

> 这个问题只会在旧的内存管理器中发生. 从 Kotlin 1.7.20 开始会默认启用新的内存管理器, 详情请参见 Kotlin/Native 内存管理 (Kotlin/Native 内存管理).

这个异常发生很可能是因为, 你试图修改一个已冻结的对象值. 对象可以明确地转变为冻结状态, 对某个对象调用 kotlin.native.concurrent.freeze 函数, 那么只被这个对象访问的其他所有对象子图都会被冻结, 对象也可以隐含的冻结 (也就是, 它只被 enum 或全局单子对象访问 - 详情请参见下一个问题).

怎样让一个单子对象(Singleton Object)可以被修改?

> 这个问题只会在旧的内存管理器中发生. 从 Kotlin 1.7.20 开始会默认启用新的内存管理器, 详情请参见 Kotlin/Native 内存管理 (Kotlin/Native 内存管理).

目前, 单子对象都是不可修改的(也就是, 创建后就被冻结), 而且我们认为让全局状态值不可变更, 通常是比较好的编程方式. 如果处于某些理由, 你需要在这样的对象内包含可变更的状态值, 请在对象上使用 @konan.ThreadLocal 注解. 另外, kotlin.native.concurrent.AtomicReference 类可以用来在被冻结的对象内, 保存指向不同的冻结对象的指针, 而且可以自动更新这些指针.

怎样使用还未发布的 Kotlin/Native 版本来编译项目?

首先, 请考虑使用 预览版 (参加 Kotlin EAP 项目).

如果你需要更新的开发版, 你可以通过源代码来编译 Kotlin/Native: clone Kotlin 代码仓库 (https://github.com/JetBrains/kotlin), 然后按照 这些步骤 (https://github.com/JetBrains/kotlin/blob/master/kotlin-native/README.md#building-from-source) 进行编译.

使用 IntelliJ IDEA 开发 Kotlin/Wasm 入门

最终更新: 2024/09/10

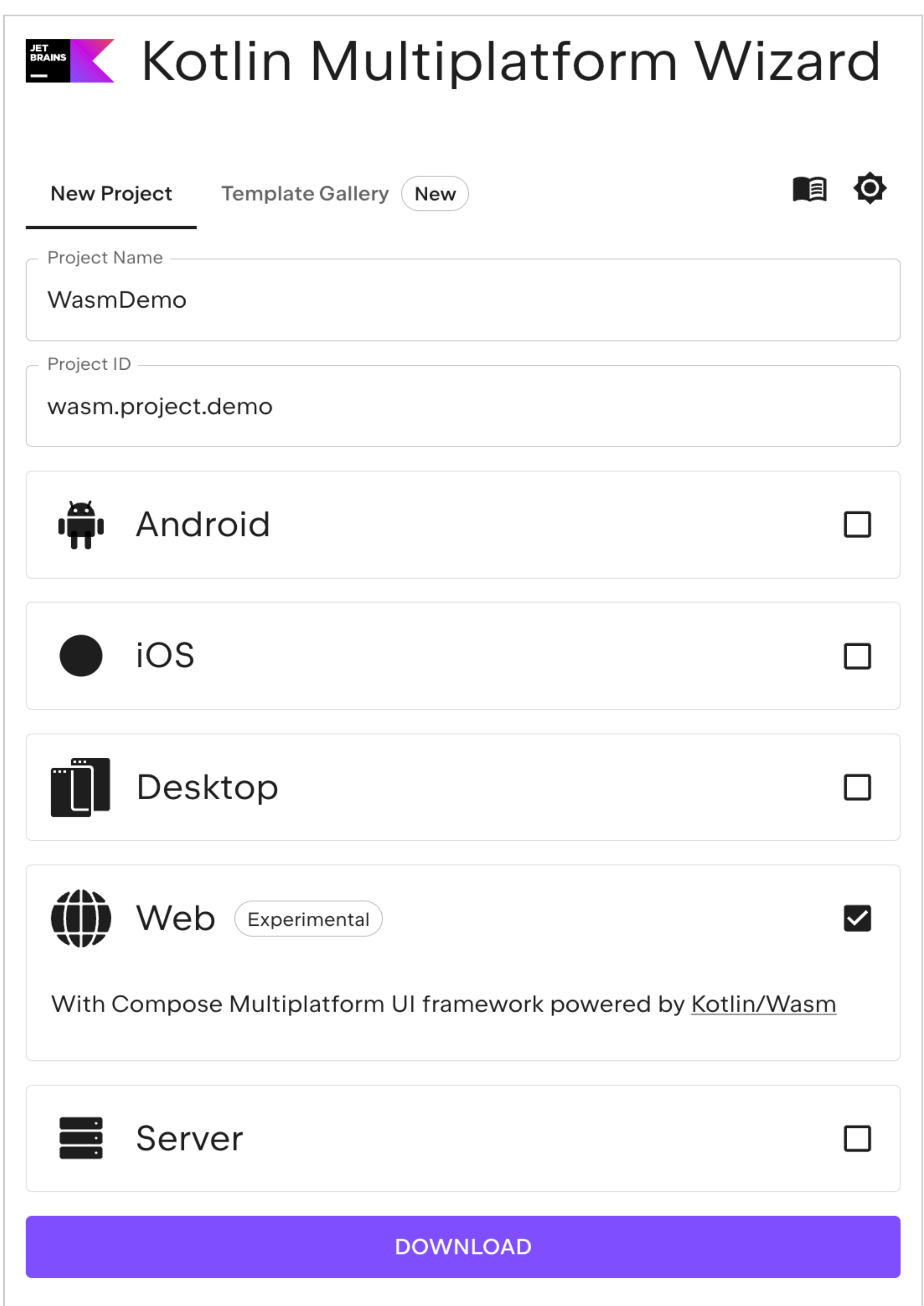
> Kotlin/Wasm 功能处于 Alpha 阶段 (Kotlin 各部分组件的稳定性). 它随时有可能变更.

本教程演示在 IntelliJ IDEA 中如何使用 Kotlin/Wasm 运行 Compose Multiplatform (https://www.jetbrains.com/lp/compose-multiplatform/) 应用程序, 以及如何生成 artifact, 并发布为 GitHub pages (https://pages.github.com/) 上的网站.

开始之前的准备步骤

使用 Kotlin Multiplatform 向导创建项目:

1. 打开 Kotlin Multiplatform 向导 (https://kmp.jetbrains.com/#newProject).
2. 在 **New Project** 页, 将项目名称修改为 "WasmDemo", 项目 ID 修改为 "wasm.project.demo".
3. 选择 **Web** 选项.
4. 点击 **Download** 按钮, 将生成的压缩包文件解包.



Kotlin Multiplatform 向导

在 IntelliJ IDEA 中打开项目

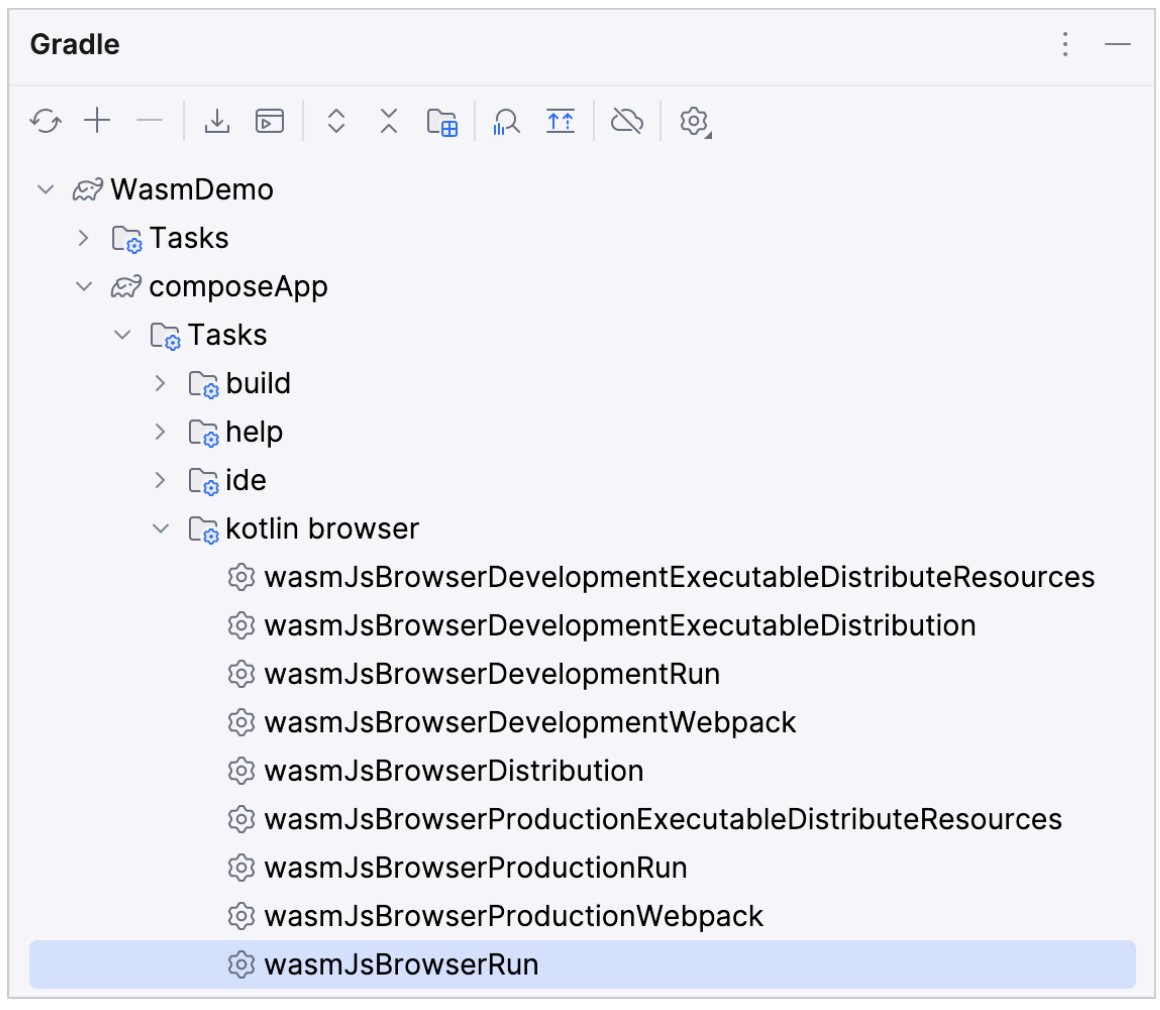
1. 下载并安装最新版本的 IntelliJ IDEA (https://www.jetbrains.com/idea/).
2. 在 IntelliJ IDEA 的欢迎界面, 点击 **Open**, 或在菜单栏选择 **File | Open**.
3. 导航到解包后的 "WasmDemo" 文件夹, 点击 **Open**.

运行应用程序

1. 在 IntelliJ IDEA 中, 选择菜单 **View** | **Tool Windows** | **Gradle**, 打开 **Gradle** 工具窗口.

    > 你需要至少 Java 11 以上版本, 用作你的 Gradle JVM, 才能成功装载 task.

2. 在 **composeApp** | **Tasks** | **kotlin browser** 中, 选中并运行 **wasmJsBrowserRun** 任务.



运行 Gradle 任务

或者, 你可以在终端窗口, 在 `composeApp` 目录下运行以下命令:

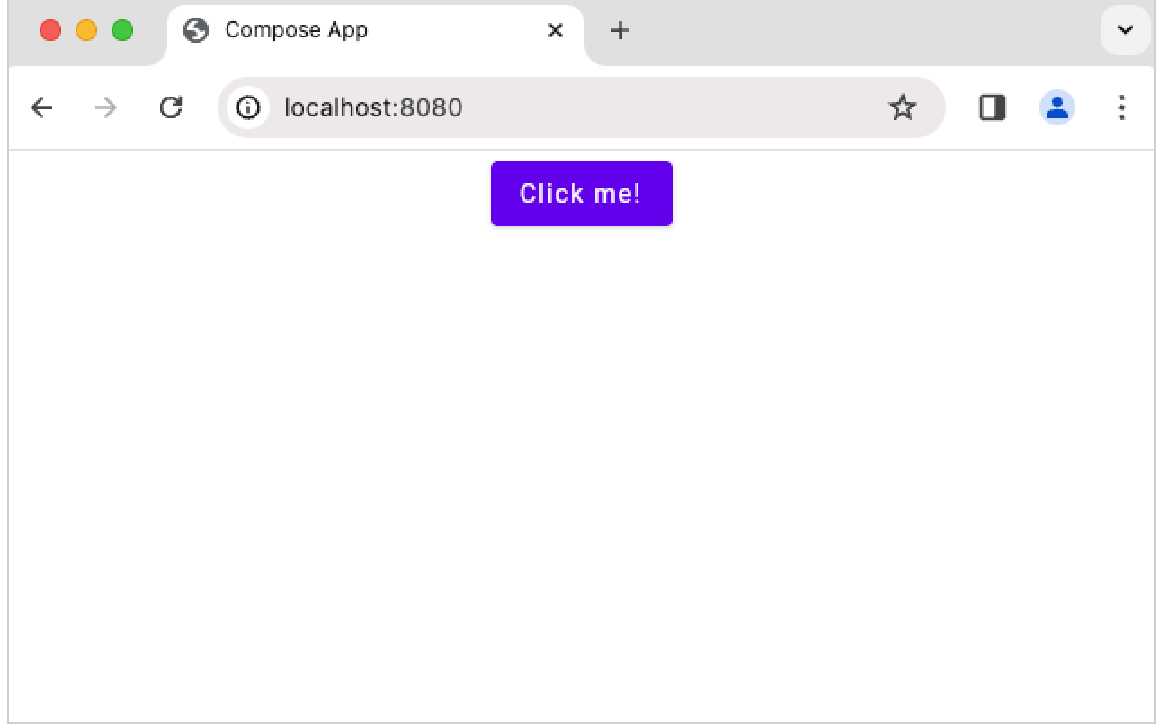
```
../gradlew wasmJsBrowserRun -t
```

3. 应用程序启动之后, 在你的浏览器中打开下面的 URL:

```
http://localhost:8080/
```

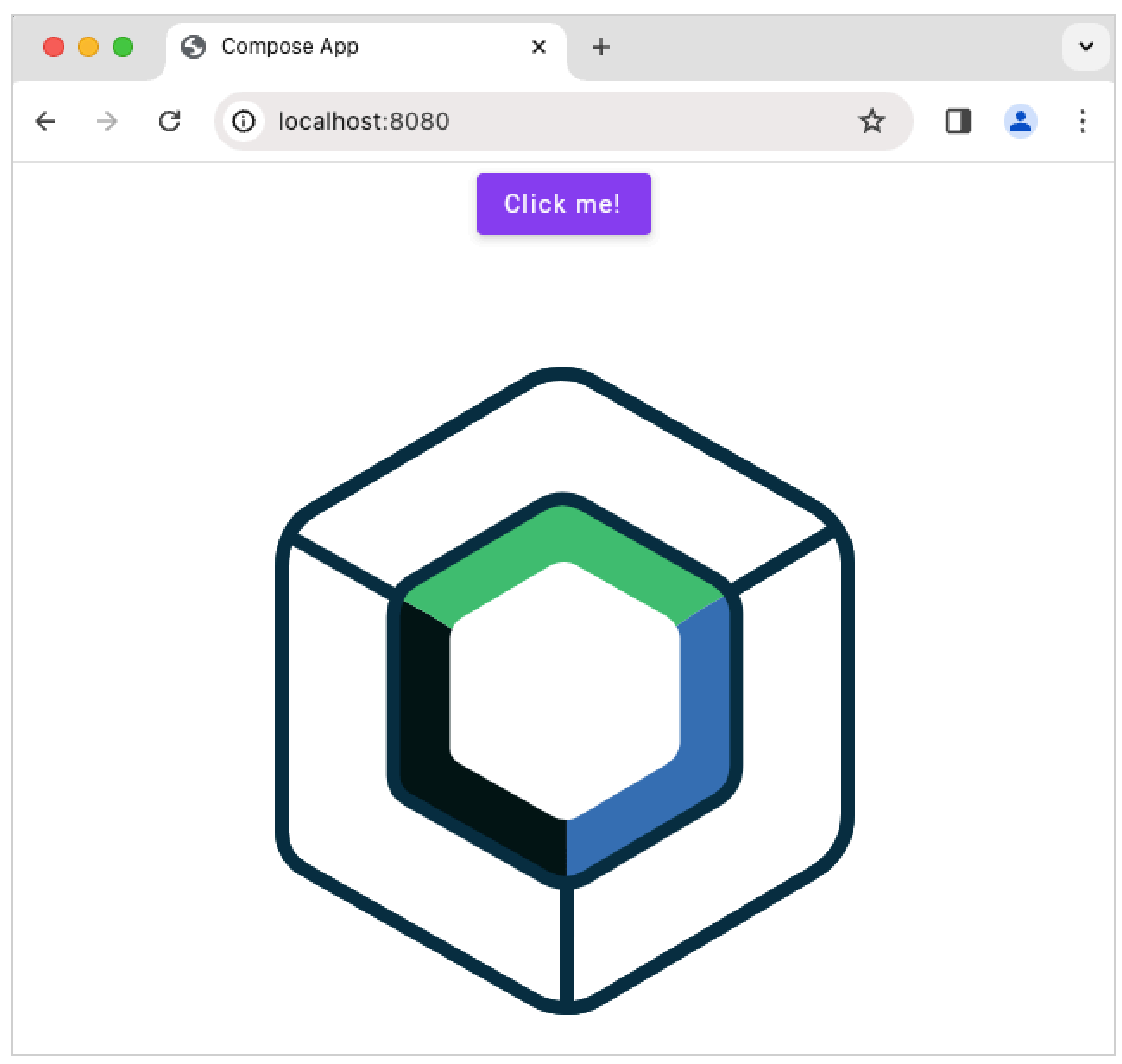
> ⚠ 端口号可能会变化, 因为 8080 端口可能不能使用. 你可以在 Gradle 构建任务的控制台看到打印输出的实际端口号.

你会看到一个 "Click me!" 按钮. 请点击它:



Click me

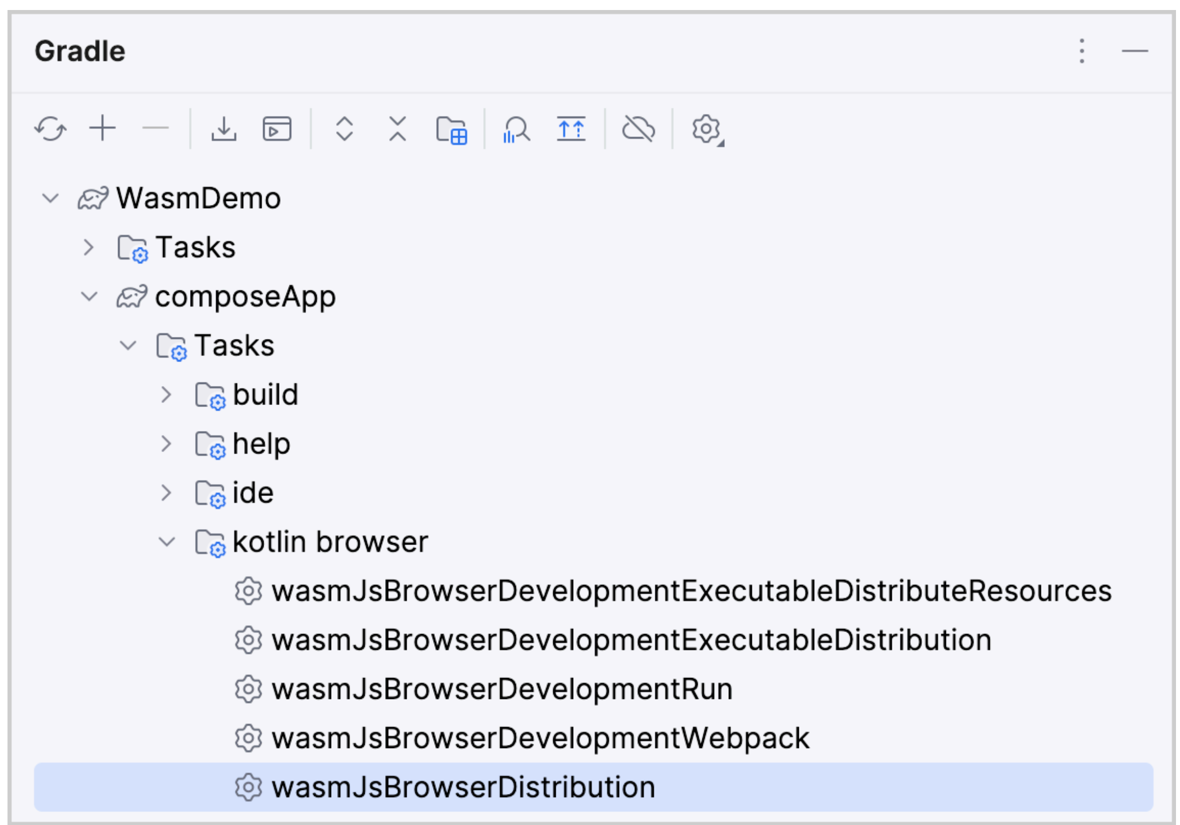
现在你会看到 Compose Multiplatform 的 Logo:



浏览器中的 Compose 应用程序

生成 artifact

在 composeApp | Tasks | kotlin browser 中, 选中并运行 wasmJsBrowserDistribution 任务.

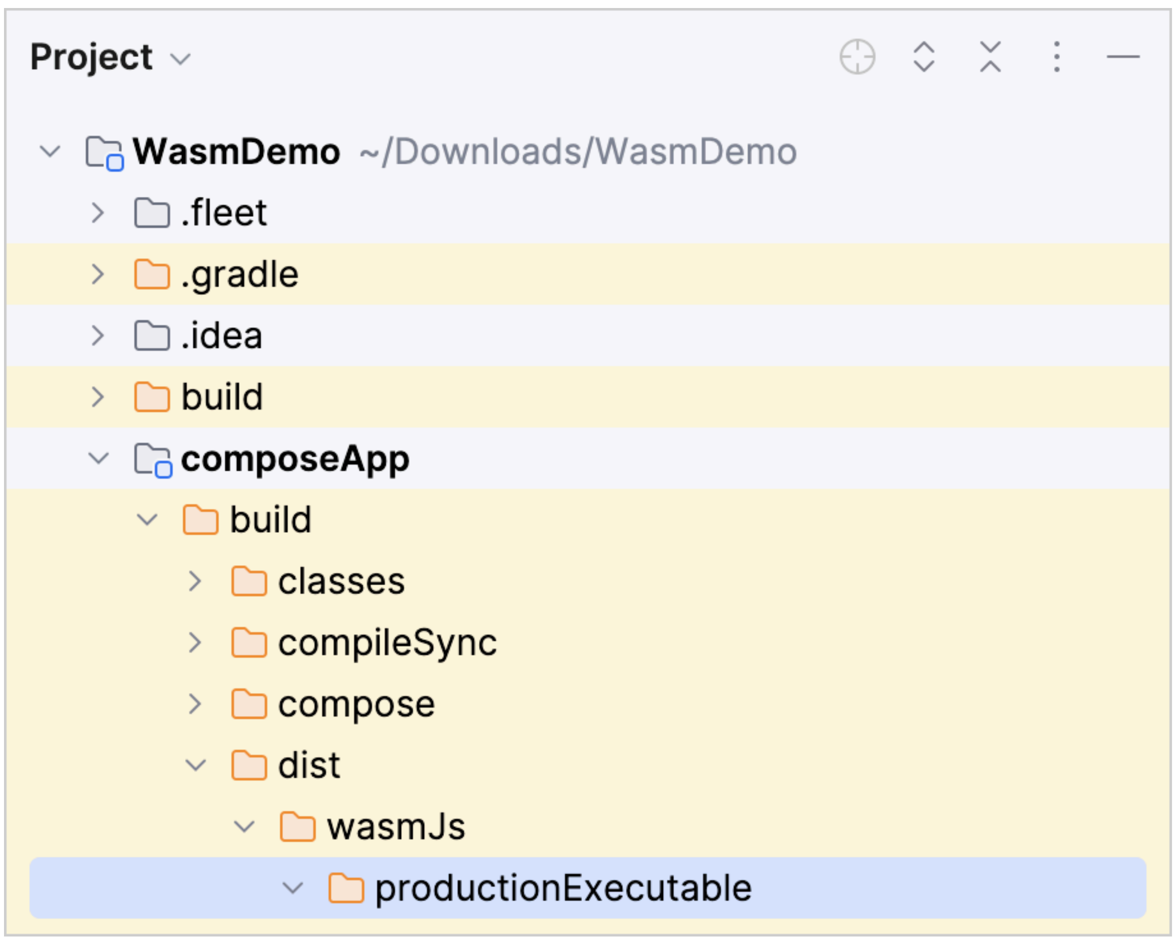


运行 Gradle 任务

或者, 你可以在终端窗口, 在 `composeApp` 目录下运行以下命令:

```
../gradlew wasmJsBrowserDistribution
```

应用程序任务结束之后, 你可以在 `composeApp/build/dist/wasmJs/productionExecutable` 文件夹中找到生成的 artifact 文件:



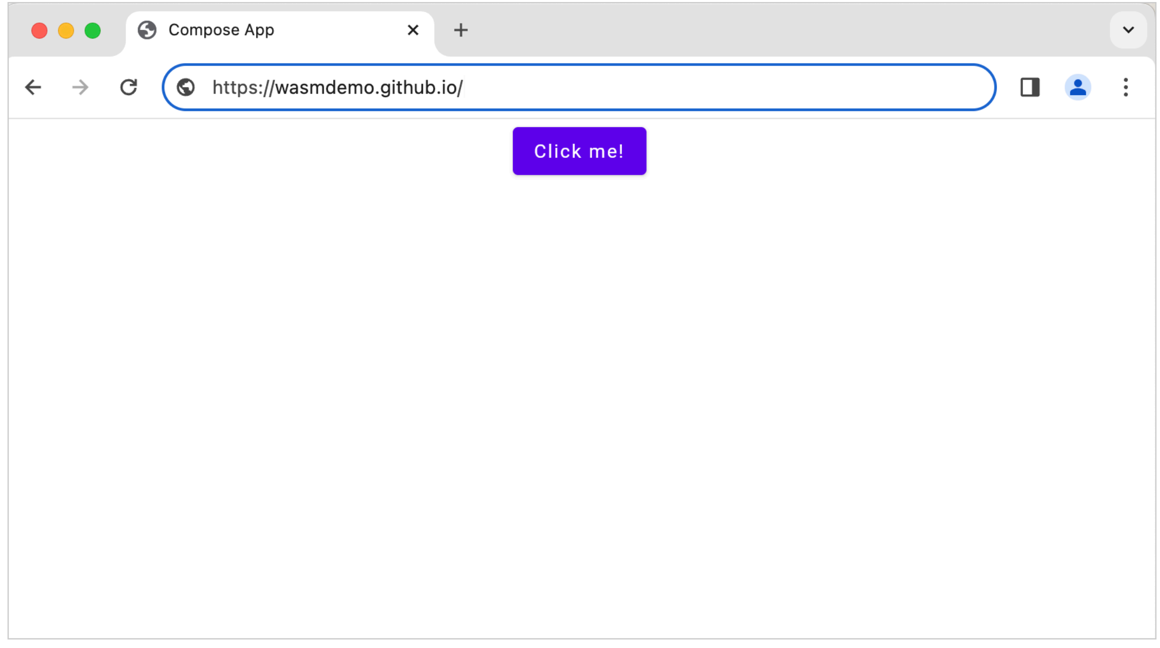
Artifact 文件目录

发布到 GitHub pages

1. 将你的 `productionExecutable` 目录中的所有内容复制到你想要创建网站的代码仓库.
2. 执行 GitHub 的 创建你的网站 (https://docs.github.com/en/pages/getting-started-with-github-pages/creating-a-github-pages-site#creating-your-site) 说明文档中的指令.

   > 在你将变更 push 到 GitHub 之后, 可能需要花费 10 分钟才能将这些变更发布到你的网站.

3. 在浏览器中, 访问你的 GitHub pages 的域名.



访问 GitHub pages

恭喜! 你已经将你的 artifact 发布到了 GitHub pages.

下一步做什么?

加入 Kotlin Slack 中的 Kotlin/Wasm 开发社区:

Join the Kotlin Slack channel →

加入 Kotlin/Wasm 开发社区

(https://slack-chats.kotlinlang.org/c/webassembly)

试试 `kotlin-wasm-examples` 代码仓库中的 Kotlin/Wasm 示例:

- Compose 图像浏览器 (https://github.com/Kotlin/kotlin-wasm-examples/tree/main/compose-imageviewer)
- Jetsnack 应用程序 (https://github.com/Kotlin/kotlin-wasm-examples/tree/main/compose-jetsnack)
- Node.js 示例 (https://github.com/Kotlin/kotlin-wasm-examples/tree/main/nodejs-example)
- WASI 示例 (https://github.com/Kotlin/kotlin-wasm-examples/tree/main/wasi-example)

* Compose 示例 (https://github.com/Kotlin/kotlin-wasm-examples/tree/main/compose-example)

向 Kotlin/Wasm 项目添加 Kotlin 库依赖项

最终更新: 2024/09/10

你可以在 Kotlin/Wasm 中使用最流行的 Kotlin 库, 这个功能开箱即用, 不需要额外的设置.

Kotlin/Wasm 支持的 Kotlin 库

使用 Maven central 仓库, 向你的项目添加 Kotlin 库:

```
// build.gradle.kts
repositories {
    mavenCentral()
}
```

支持的库
stdlib
kotlin-test
kotlinx-coroutines
Compose Multiplatform
Compose Compiler
kotlinx-serialization
kotlinx-atomicfu
kotlinx-collections-immutable
kotlinx-datetime
skiko

在你的项目中启用库

要设置对一个库的依赖项, 例如 `kotlinx.serilization` (序列化) 和 `kotlinx.coroutines` (协程指南), 请更新你的 `build.gradle.kts` 文件:

```
// `build.gradle.kts`

repositories {
    mavenCentral()
}

kotlin {
    sourceSets {
```

```
        val wasmJsMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-
serialization-core:1.6.0")
                implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3")
            }
        }
    }
}
```

下一步做什么?

查看 Kotlin/Wasm 与 JavaScript 的交互能力 (与 JavaScript 交互)

与 JavaScript 交互

最终更新: 2024/09/10

Kotlin/Wasm 允许你在 Kotlin 中使用 JavaScript 代码, 也允许你在 JavaScript 中使用 Kotlin 代码.

和 Kotlin/JS (使用 Kotlin 进行 JavaScript 开发) 一样, Kotlin/Wasm 编译器也具有与 JavaScript 互操作的能力. 如果你熟悉 Kotlin/JS 的互操作能力, 你会注意到 Kotlin/Wasm 的互操作能力与此类似. 然而, 还是存在一些关键的差异需要注意.

> Kotlin/Wasm 功能处于 Alpha 阶段 (Kotlin 各部分组件的稳定性). 它随时有可能变更. 请不要将这个功能用于正式产品. 希望你能通过我们的 问题追踪系统 (https://kotl.in/issue) 提供你的反馈意见.

在 Kotlin 中使用 JavaScript 代码

本节介绍如何在 Kotlin 中使用 JavaScript 代码, 方法是使用 `external` 声明, 带 JavaScript 代码块的函数, 以及 `@JsModule` 注解.

外部声明

在 Kotlin 中默认无法使用外部的 JavaScript 代码. 要在 Kotlin 中使用 JavaScript 代码, 你可以使用 `external` 声明来描述它的 API.

JavaScript 函数

对于这个 JavaScript 函数:

```
function greet (name) {
    console.log("Hello, " + name + "!");
}
```

你可以在 Kotlin 中将它声明为一个 `external` 函数:

```
external fun greet(name: String)
```

外部函数没有函数体, 你可以象通常的 Kotlin 函数那样调用它:

```
fun main() {
    greet("Alice")
}
```

JavaScript 属性

对于这个 JavaScript 全局变量:

```
let globalCounter = 0;
```

你可以在 Kotlin 中使用外部的 `var` 或 `val` 属性声明它:

```
external var globalCounter: Int
```

这些属性会在外部初始化. 在 Kotlin 代码中, 属性不能带有 `= value` 这样的初始化代码.

JavaScript 类

对于这个 JavaScript 类:

```
class Rectangle {
    constructor (height, width) {
        this.height = height;
        this.width = width;
    }

    area () {
        return this.height * this.width;
    }
}
```

你可以在 Kotlin 中将它作为外部类来使用:

```
external class Rectangle(height: Double, width: Double) : JsAny {
    val height: Double
    val width: Double
    fun area(): Double
}
```

在 `external` 类之内的所有声明都会隐含的被认为是外部声明.

外部接口

你可以在 Kotlin 中描述 JavaScript 对象的行为. 对于这个 JavaScript 函数和它的返回值:

```
function createUser (name, age) {
    return { name: name, age: age };
}
```

请看在 Kotlin 中如何使用一个 `external interface User` 类型来描述它的行为:

```
external interface User : JsAny {
    val name: String
    val age: Int
}

external fun createUser(name: String, age: Int): User
```

外部接口没有运行期的类型信息, 它是只在编译期存在的概念. 因此, 外部接口与通常的接口相比, 存在一些限制:

- 你不能在 `is` 检查的右侧使用外部接口.
- 你不能在类字面值表达式(例如 `User::class`)中使用外部接口.
- 你不能将外部接口用做实体化的类型参数(Reified type parameter).
- 使用 `as` 操作符, 将一个对象类型转换为外部接口时, 永远会成功.

外部对象

对于这些保存在对象中的 JavaScript 变量:

```
let Counter = {
    value: 0,
    step: 1,
    increment () {
        this.value += this.step;
    }
};
```

你可以在 Kotlin 中将它们作为外部对象来使用:

```
external object Counter : JsAny {
    fun increment()
    val value: Int
    var step: Int
}
```

外部的类型层次结构

与通常的类和接口相似, 你可以让外部声明扩展其他的外部类, 实现外部接口. 但是, 你不能在同一个类型层次结构中混合使用外部声明和非外部声明.

带有 JavaScript 代码的 Kotlin 函数

你可以定义一个函数, 函数体为 `= js("code")` 形式, 这样就可以将 JavaScript 代码段添加到 Kotlin/Wasm 代码中:

```
fun getCurrentURL(): String =
    js("window.location.href")
```

如果你想要运行多条 JavaScript 语句组成的代码段, 请将你的代码包含在大括号 `{}` 内, 再放在字符串中:

```
fun setLocalSettings(value: String): Unit = js(
    """{
        localStorage.setItem('settings', value);
}"""
)
```

如果你想要返回一个对象, 请在大括号 `{}` 之外加上小括号 `()`:

```
fun createJsUser(name: String, age: Int): JsAny =
    js("({ name: name, age: age })")
```

Kotlin/Wasm 会对 `js()` 函数调用特殊处理, 而且它的实现存在一些限制:

- `js()` 函数调用的参数必须是字符串字面值.
- `js()` 函数调用必须是函数体中唯一的表达式.

- 只允许在包级(package-level)函数中调用 `js()` 函数.
- 函数的返回值必须明确指定.
- 类型 存在限制, 与 `external fun` 类似.

Kotlin 编译器会生成 JavaScript 文件, 将代码字符串放入函数中, 并将 JavaScript 文件导入为 WebAssembly 格式. Kotlin 编译器不会验证这些 JavaScript 代码段. 如果存在 JavaScript 语法错误, 这些错误会在你运行 JavaScript 代码时报告.

> `@JsFun` 注解的功能与此类似, 可能会被弃用.

JavaScript 模块

默认情况下, 外部声明对应于 JavaScript 的全局范围(global scope). 如果你对一个 Kotlin 文件标注 `@JsModule` 注解 ("@JsModule 注解" in "JavaScript 模块"), 那么这个文件内的所有外部声明都会从指定的模块导入.

请看这个 JavaScript 代码示例:

```
// users.mjs
export let maxUsers = 10;

export class User {
    constructor (username) {
        this.username = username;
    }
}
```

在 Kotlin 中, 使用 `@JsModule` 注解来使用这段 JavaScript 代码:

```
// Kotlin
@file:JsModule("./users.mjs")

external val maxUsers: Int

external class User : JsAny {
    constructor(username: String)
```

```
    val username: String
}
```

在 JavaScript 中使用 Kotlin 代码

本节介绍在 JavaScript 中如何使用你的 Kotlin 代码, 方法是使用 `@JsExport` 注解.

带 @JsExport 注解的函数

要让一个 Kotlin/Wasm 函数可以被 JavaScript 代码使用, 请添加 `@JsExport` 注解:

```
// Kotlin/Wasm

@JsExport
fun addOne(x: Int): Int = x + 1
```

在生成的 `.mjs` 模块中, 标注了 `@JsExport` 注解的 Kotlin/Wasm 函数会成为一个 `default` export 上的属性. 然后你就可以在 JavaScript 中使用这个函数:

```
// JavaScript

import exports from "./module.mjs"

exports.addOne(10)
```

类型的对应关系

在与 JavaScript 交互的声明中, Kotlin/Wasm 只允许使用某些类型. 对于使用 `external`, `= js("code")` 或 `@JsExport` 的声明, 这些限制是统一的.

下面是 Kotlin 类型对应的 Javascript 类型:

Kotlin	JavaScript
Byte, Short, Int, Char	Number
Float, Double,	Number
Long,	BigInt
Boolean,	Boolean
String,	String
在 return 语句中的 Unit	undefined
函数类型, 例如 (String) → Int	Function
JsAny 和子类型	任何 JavaScript 值
JsReference	指向 Kotlin 对象的不透明引用(Opaque Reference)
其他类型	不支持

你也可以使用这些类型的可为 null 的版本.

JsAny 类型

JavaScript 值在 Kotlin 中使用 JsAny 类型和它的子类型来表示.

标准库提供了其中一些类型的表示:

- kotlin.js 包:
  - JsAny
  - JsBoolean, JsNumber, JsString
  - JsArray

- Promise

- org.khronos.webgl 包:
  - 类型数组, 例如 Int8Array
  - WebGL 类型
- org.w3c.dom.* 包:
  - DOM API 类型

你也可以声明一个 external 接口或类, 创建自定义的 JsAny 子类型.

JsReference 类型

使用 JsReference 类型, Kotlin 值可以作为不透明引用(Opaque Reference)传递给 JavaScript.

例如, 如果你想要将这个 Kotlin 类 User 公开给 JavaScript:

```
class User(var name: String)
```

你可以使用 toJsReference() 函数来创建 JsReference<User>, 并将它返回给 JavaScript:

```
@JsExport
fun createUser(name: String): JsReference<User> {
    return User(name).toJsReference()
}
```

这些引用在 JavaScript 中不能直接使用, 行为就像空的冻结 JavaScript 对象. 要操作这些对象, 你需要将更多函数导出到 JavaScript, 方法是, 使用 get() 方法解包这些引用值:

```
@JsExport
fun setUserName(user: JsReference<User>, name: String) {
    user.get().name = name
}
```

你可以在 JavaScript 中创建一个类, 并改变它的名称:

```
import UserLib from "./userlib.mjs"
```

```
let user = UserLib.createUser("Bob");
UserLib.setUserName(user, "Alice");
```

类型参数

与 JavaScript 互操作的声明, 可以拥有类型参数, 类型参数的上界(Upper Bound)是 JsAny 或它的子类型. 例如:

```
external fun <T : JsAny> processData(data: JsArray<T>): T
```

异常处理

Kotlin 的 try-catch 表达式不能捕获 JavaScript 的异常.

如果你使用 JavaScript 的 try-catch 表达式来捕获 Kotlin/Wasm 的异常, 那么捕获的异常会象是一个普通的 WebAssembly.Exception, 没有可以直接访问的错误消息和数据.

Kotlin/Wasm 互操作功能与 Kotlin/JS 互操作功能的区别

尽管 Kotlin/Wasm 互操作功能与 Kotlin/JS 互操作功能很类似, 但它们还是存在一些关键的差异需要注意:

	Kotlin/Wasm	Kotlin/JS
外部枚举类型	不支持外部枚举类.	支持外部枚举类.
类型扩展	不支持非外部类型扩展外部类型.	支持非外部类型.
JsName 注解	这个注解时只有对外部声明标注时才有效.	可以用来修改通常的非外部声明的名称.
js() 函数	只允许包级(package-level)函数调用 js("code") 函数, 而且js() 函数调用必须是函数体中唯一的表达式.	js("code") 函数可以在任何地方调用, 并返回一个 dynamic 值.
模块系统	只支持 ES 模块. 没有类似的 @JsNonModule 注解. 会导出为 default 对象上的属性. 只允许导出包级函数.	支持 ES 模块, 以及其它旧的模块系统. 提供有名称的 ESM 导出. 允许导出类和对象.
类型	对所有的互操作性声明 external, = js("code"), 和 @JsExport, 统一适用更严格的类型限制. 只允许使用一部分的 内建的 Kotlin 类型和 JsAny 子类型.	在 external 声明中允许使用所有的类型. 在 @JsExport 中可以使用限定的类型 ("JavaScript 中的 Kotlin 类型" in "在 JavaScript 中使用 Kotlin 代码").
Long	对应于 JavaScript 的 BigInt 类型.	在 JavaScript 中会成为一个自定义的类.
Arrays	在互操作功能中目前还不直接支持. 你可以改为使用新的 JsArray 类型.	实现为 JavaScript 数组.
其他类型	需要使用 JsReference<> 来将 Kotlin 对象传递给 JavaScript.	允许在外部声明中使用非外部的 Kotlin 类类型.
异常处理	不能捕获 JavaScript 的异常.	能够通过 Throwable 类型捕获 JavaScript 的 Error. 可以使用 dynamic 类型捕

		获 JavaScript 的任何异常.
动态类型(Dynamic Type)	不支持 dynamic 类型. 请改为使用 JsAny (参见下面的示例代码).	支持 dynamic 类型.

Kotlin/JS 用于互操作的 动态类型(Dynamic Type) (动态类型), 使用无类型的, 或松散类型的对象, 在 Kotlin/Wasm 中不支持这个功能. 你可以使用 JsAny 类型来替代 dynamic 类型:

```
// Kotlin/JS
fun processUser(user: dynamic, age: Int) {
    // ...
    user.profile.updateAge(age)
    // ...
}

// Kotlin/Wasm
private fun updateUserAge(user: JsAny, age: Int): Unit =
    js("{ user.profile.updateAge(age); }")

fun processUser(user: JsAny, age: Int) {
    // ...
    updateUserAge(user, age)
    // ...
}
```

问题分析

最终更新: 2024/09/10

Kotlin/Wasm 依赖于新的 WebAssembly 提案 (https://webassembly.org/roadmap/), 例如垃圾收集和异常处理, 这些提案会为 WebAssembly 带来改进和新功能.

但是, 要确保这些功能能够正常工作, 你需要一个支持这些新提案的环境. 有些情况下, 你可能需要设置环境, 使它与这些新提案兼容.

浏览器版本

要在浏览器中运行使用 Kotlin/Wasm 构建的应用程序, 你需要支持新的 WebAssembly 垃圾收集 (WasmGC) 功能 (https://github.com/WebAssembly/gc) 的浏览器版本. 请检查浏览器版本是否默认支持新的 WasmGC, 或者需要你对环境进行更改.

Chrome

- **对于 119 或以上版本:**

  默认能够工作.

- **对于旧版本:**

  > 要在旧版本的浏览器中运行应用程序, 需要 Kotlin 1.9.20 以前的版本.

  1. 在你的浏览器中, 进入 `chrome://flags/#enable-webassembly-garbage-collection`.
  2. 启用 **WebAssembly Garbage Collection**.
  3. 重新启动你的浏览器.

基于 Chromium 的浏览器

包括基于 Chromium 的浏览器, 例如 Edge, Brave, Opera, 或 Samsung Internet.

- **对于 119 或以上版本:**

  默认能够工作.

- **对于旧版本:**

> 要在旧版本的浏览器中运行应用程序, 需要 Kotlin 1.9.20 以前的版本.

使用 `--js-flags=--experimental-wasm-gc` 命令行参数运行应用程序.

Firefox

- **对于 120 或以上版本:**

  默认能够工作.

- **对于 119 版本:**

  1. 在你的浏览器中, 进入 `about:config`.
  2. 启用 `javascript.options.wasm_gc` 选项.
  3. 刷新页面.

Safari/WebKit

WebAssembly 垃圾收集功能的支持目前正在 积极开发中
(https://bugs.webkit.org/show_bug.cgi?id=247394).

> 要学习如何设置项目, 使用依赖项, 以及其他任务, 请参见我们的 Kotlin/Wasm 示例
> (https://github.com/Kotlin/kotlin-wasm-examples#readme).

创建 Kotlin/JS 工程(Project)

最终更新: 2024/09/10

Kotlin JavaScript 工程(Project) 使用 Gradle 进行编译. 为了方便开发者管理 Kotlin JavaScript 工程, 我们提供了 `kotlin.multiplatform` Gradle 插件, 其中包括工程配置工具, 以及对 JavaScript 开发中常见业务进行自动化处理的帮助性任务. 比如, 这个插件会在后台下载 Yarn (https://yarnpkg.com/) 包管理器, 用来管理 npm (https://www.npmjs.com/) 依赖项, 还可以使用 webpack (https://webpack.js.org/) 将 Kotlin 工程编译为 JavaScript bundle. 依赖项管理和配置调整大部分可以直接在 Gradle 构建脚本文件中完成, 还可以通过选项覆盖自动生成的配置, 获得完全的控制能力.

你可以在 Gradle 工程的 `build.gradle(.kts)` 文件中手动的应用 `org.jetbrains.kotlin.multiplatform` 插件:

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}
```

通过 Kotlin Multiplatform Gradle 插件, 你可以在编译脚本的 `kotlin` 节中管理工程的各方面设置.

```
kotlin {
    //...
}
```

在 `kotlin {}` 代码段中, 你可以管理以下方面:

- 目标执行环境: 浏览器, 或 Node.js
- 工程的依赖项目管理: Maven 或 npm
- 运行配置(configuration)
- 测试配置(configuration)
- 对于浏览器工程的 打包(Bundling) 和 CSS 支持
- 目标目录 和 模块名称
- 工程的 `package.json` 文件

执行环境

Kotlin/JS 工程可以运行于两种不同的执行环境:

- 浏览器环境, 用于浏览器内运行的客户端脚本
- Node.js (https://nodejs.org/), 在浏览器之外运行 JavaScript 代码, 比如, 运行服务器端脚本.

要为 Kotlin/JS 工程定义目标运行环境, 需要添加 `js {}` 代码段, 其中包含 `browser {}` 或 `nodejs {}`:

```
kotlin {
    js {
        browser {
        }
        binaries.executable()
    }
}
```

`binaries.executable()` 指令明确的指示 Kotlin 编译器输出可执行的 `.js` 文件. 使用当前的 Kotlin/JS 编译器时, 这是默认的行为, 但如果你使用 Kotlin/JS IR 编译器 (使用 IR 编译器), 或者在 `gradle.properties` 文件中设置过 `kotlin.js.generate.executable.default=false`, 则需要明确的指定这条指令. 这些情况下, 省略 `binaries.executable()` 会导致编译器只生成 Kotlin-internal 库文件, 这些库文件可以被其他项目使用, 但不能独立运行.

> ⚠ 与创建可执行文件相比, 这样通常会更快, 而且在处理你项目中的非叶(non-leaf)模块时,

这是一种可能的优化.

Kotlin Multiplatform 插件会针对选定的运行环境, 自动配置它的编译任务. 包括下载并安装应用程序运行和测试所需要的环境和依赖项目, 因此开发者可以编译, 运行, 以及测试简单的工程, 而无需再添加更多配置. 对于编译目标为 Node.js 的项目, 还有一个选项可以使用本地已安装的 Node.js. 详情请参见 使用已安装的 Node.js.

依赖项目

与其他 Gradle 工程一样, Kotlin/JS 工程编译脚本的 `dependencies {}` 代码段内, 支持添加传统的 Gradle 依赖项目声明 (https://docs.gradle.org/current/userguide/declaring_dependencies.html):

Kotlin

```
dependencies {
    implementation("org.example.myproject", "1.1.0")
}
```

Groovy

```
dependencies {
    implementation 'org.example.myproject:1.1.0'
}
```

Kotlin Multiplatform Gradle 插件也支持在编译脚本的 `kotlin {}` 代码段中添加特定源代码集合 (source set)的依赖项目声明:

Kotlin

```
kotlin {
    sourceSets {
        val jsMain by getting {
            dependencies {
                implementation("org.example.myproject:1.1.0")
            }
```

```
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        jsMain {
            dependencies {
                implementation 'org.example.myproject:1.1.0'
            }
        }
    }
}
```

> 并不是 Kotlin 编程语言中所有可用的库在 JavaScript 平台都可用: 只有那些包含针对 Kotlin/JS 的 artifact 的库才能使用.

如果你添加的库依赖于 来自 npm 的包, Gradle 也会自动解析这些传递性依赖项.

Kotlin 标准库

对 标准库 (https://kotlinlang.org/api/latest/jvm/stdlib/index.html) 的依赖项会自动添加. 标准库的版本与 Kotlin Multiplatform 插件的版本相同.

对于跨平台的测试, 可以使用 `kotlin.test` (https://kotlinlang.org/api/latest/kotlin.test/) API. 当你创建跨平台项目时, 你可以在 `commonTest` 中使用一个依赖项, 对所有的源代码集添加测试依赖项:

Kotlin

```
kotlin {
    sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test")) // 会自动引入所有的平台依
赖项
```

```
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // 会自动引入所有的平
台依赖项
            }
        }
    }
}
```

npm 依赖项目

在 JavaScript 的世界中, 管理依赖项目的最常见方式是 npm (https://www.npmjs.com/). 它提供了各种 JavaScript 模块(module) 的最大的公共仓库(repository).

通过 Kotlin Multiplatform Gradle 插件, 可以在 Gradle 编译脚本中声明 npm 依赖项目, 方法和声明其他依赖项目类似.

要声明一个 npm 依赖项目, 可以在一个依赖项目声明中使用 `npm()` 函数指定依赖项目的名称和版本. 也可以使用 npm semver 语法 (https://docs.npmjs.com/misc/semver#versions), 指定一个或多个版本范围.

Kotlin

```
dependencies {
    implementation(npm("react", "> 14.0.0 <=16.9.0"))
}
```

Groovy

```
dependencies {
    implementation npm('react', '> 14.0.0 <=16.9.0')
}
```

插件会使用 Yarn (https://yarnpkg.com/lang/en/) 包管理器来下载和安装 npm 依赖项. 不需要额外配置, 默认即可工作, 但你也可以根据需要对其进行调整. 详情请参见 在 Kotlin Multiplatform Gradle plugin 中配置 Yarn.

除了标准依赖项之外, 在 Gradle DSL 中使用还可以使用 3 种其他类型的依赖项. 关于什么情况下应该选择什么类型的依赖项, 请阅读 npm 提供的官方文档:

- devDependencies (https://docs.npmjs.com/files/package.json#devdependencies), 通过 `devNpm(...)` 使用,
- optionalDependencies (https://docs.npmjs.com/files/package.json#optionaldependencies) 通过 `optionalNpm(...)` 使用, 以及
- peerDependencies (https://docs.npmjs.com/files/package.json#peerdependencies) 如果 `peerNpm(...)` 使用.

一个 npm 依赖项目安装完成之后, 你就可以如 在 Kotlin 中调用 JavaScript (在 Kotlin 中使用 JavaScript 代码) 中介绍过的那样, 在你的代码中使用它的 API.

run 任务

Kotlin Multiplatform Gradle 插件提供了一个 `jsRun` 任务, 它可以运行你的纯 Kotlin/JS 工程, 无需额外的配置.

对于在浏览器内运行 Kotlin/JS 工程的情况, 这个是 `browserDevelopmentRun` 任务的一个别名 (在 Kotlin 跨平台项目也可以使用). 它使用 webpack DevServer (https://webpack.js.org/configuration/dev-server/) 来提供你的 JavaScript artifact. 如果你想要自定义 DevServer 的配置, 例如, 改变端口号, 请使用 webpack 配置文件.

对于在 Node.js 平台运行 Kotlin/JS 项目的情况, 请使用 `jsRun` 任务, 它是 `nodeRun` 任务的别名.

要运行一个工程, 请执行 Gradle 编译周期(lifecycle)中标准的 `jsRun` 任务, 或者运行它作为别名对应的真实的任务:

```
./gradlew jsRun
```

如果要在修改过源代码文件后自动对你的应用程序进行重新构建, 可以使用 Gradle 的 连续构建 (continuous build) (https://docs.gradle.org/current/userguide/command_line_interface.html#sec:continuous_build) 功能:

```
./gradlew jsRun --continuous
```

或者

```
./gradlew jsRun -t
```

工程构建成功后, `webpack-dev-server` 会自动刷新浏览器页面.

test 任务

Kotlin Multiplatform Gradle 插件会为工程自动设置测试环境. 对于浏览器工程, 它会下载并安装测试运行器 Karma (https://karma-runner.github.io/), 以及相关的依赖项目; 对于 Node.js 项目, 会使用 Mocha (https://mochajs.org/) 测试框架.

插件还提供了很多有用的测试功能, 比如:

- 生成源代码文件映射(Source map)
- 生成测试报告(Test report)
- 在控制台输出测试运行结果

为了运行浏览器中的测试, 插件会默认使用 Headless Chrome (https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md). 你也可以选择其他浏览器来运行测试, 方法是在编译脚本的 `useKarma {}` 代码段中添加相应的设置:

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useIe()
```

```
                    useSafari()
                    useFirefox()
                    useChrome()
                    useChromeCanary()
                    useChromeHeadless()
                    usePhantomJS()
                    useOpera()
                }
            }
        }
        binaries.executable()
        // ...
    }
}
```

或者你也可以在 `gradle.properties` 文件中添加测试的目标浏览器:

```
kotlin.js.browser.karma.browsers=firefox,safari
```

通过这种方法, 你可以为所有的模块定义一组浏览器, 然后在某些模块的构建脚本中添加特定的浏览器.

请注意, Kotlin Multiplatform Gradle 插件不会为你自动安装这些浏览器, 而只是使用那些在它的运行环境中可用的浏览器. 比如说, 如果在一个持续集成服务器上运行 Kotlin/JS 测试, 请注意确保安装了你需要测试的浏览器.

如果想要跳过测试, 可以在 `testTask {}` 代码段中添加 `enabled = false` 设置.

```
kotlin {
    js {
        browser {
            testTask {
                enabled = false
            }
        }
        binaries.executable()
        // ...
    }
}
```

要运行测试, 请执行 Gradle 编译周期(lifecycle)中标准的 `check` 任务:

```
./gradlew check
```

如果要指定你的 Node.js 测试运行器使用的环境变量 (比如, 向你的测试代码传递外部信息, 或对包的解析进行微调), 可以在你的构建脚本的 `testTask {}` 代码段中使用 `environment()` 函数, 参数是键-值对:

```
kotlin {
    js {
        nodejs {
            testTask {
                environment("key", "value")
            }
        }
    }
}
```

配置 Karma

Kotlin Multiplatform Gradle 插件会在构建时自动生成 Karma 配置文件, 其中包括你的 `build.gradle(.kts)` 文件中的 `kotlin.js.browser.testTask.useKarma {}` 代码段 中的设置. 你可以在 `build/js/packages/projectName-test/karma.conf.js` 找到这个文件. 要调整 Karma 所使用的配置, 请将你的额外配置文件 放在你的项目根目录的 `karma.config.d` 目录之下. 在构建时, 这个目录下的所有 `.js` 配置文件都会被读取, 并自动合并到生成的 `karma.conf.js` 文件中.

Karma 配置的详细功能请参见 Karma 的 文档 (https://karma-runner.github.io/5.0/config/configuration-file.html).

webpack 打包(Bundling)

如果编译目标为浏览器环境, Kotlin/JS 插件使用大家都熟悉的 webpack (https://webpack.js.org/) 来打包模块.

webpack 版本

Kotlin Multiplatform 插件使用 webpack 5.

如果你的项目通过 plugin 1.5.0 以前版本创建, 那么可以在你的项目的 `gradle.properties` 文件中添加以下设置, 临时切换回这些版本使用的 webpack 4:

```
kotlin.js.webpack.major.version=4
```

webpack 任务

在 Gradle 编译脚本的 `kotlin.js.browser.webpackTask {}` 配置代码段中, 可以直接调整最常见的 webpack 配置:

* `outputFileName` - webpack 的输出文件名称. 执行webpack 任务之后, 这个文件将生成在 `<projectDir>/build/dist/<targetName>` 文件夹内. 默认值是工程名称.
* `output.libraryTarget` - 用于 webpack 输出文件的模块系统. 详情请参见 Kotlin/JS 工程可用的模块系统 (JavaScript 模块). 默认值是 `umd`.

```
webpackTask {
    outputFileName = "mycustomfilename.js"
    output.libraryTarget = "commonjs2"
}
```

还可以在 `commonWebpackConfig {}` 代码段中配置 webpack 的共通设置, 用于打包 (bundling), 运行, 以及测试任务.

webpack 配置文件

Kotlin Multiplatform Gradle plugin 在构建时会自动生成一个标准的 webpack 配置文件. 位置是 `build/js/packages/projectName/webpack.config.js`.

如果还想对 webpack 配置进行进一步的调整, 请将你的额外的配置文件放在你的工程的 `webpack.config.d` 目录内. 编译你的工程时, 所有的 `.js` 配置文件都会被自动合并到 `build/js/packages/projectName/webpack.config.js` 文件内. 比如, 如果要添加一个新的 webpack loader (https://webpack.js.org/loaders/), 请要把以下内容添加到 `webpack.config.d` 目录内的一个 `.js` 中:

> 这种情况下, 配置对象是全局对象 `config`. 你需要在你的脚本中修改这个对象.

```
config.module.rules.push({
    test: /\.extension$/,
    loader: 'loader-name'
});
```

关于 webpack 的所有配置项目, 请参见它的 文档 (https://webpack.js.org/concepts/configuration/).

构建可执行文件

要通过 webpack 编译可执行的 JavaScript artifact, Kotlin Multiplatform Gradle 插件包含 Gradle 任务 `browserDevelopmentWebpack` 和 `browserProductionWebpack`.

- `browserDevelopmentWebpack` 创建开发模式的 artifact, 文件尺寸会比较大, 但构建时间比较短. 因此, 在活跃开发阶段请使用 `browserDevelopmentWebpack` 任务.
- `browserProductionWebpack` 会执行 死代码消除 (JavaScript 死代码剔除工具), 生成 artifact 文件, 并对输出结果的 JavaScript 文件最小化, 构建时间更长, 但生成的可执行文件尺寸更小. 因此, 在构建你的项目用于生成目的时, 请使用 `browserProductionWebpack` 任务.

执行这两个任务可以分别得到开发模式和生产模式的 artifact 文件. 生成的文件会在 `build/dist` 目录下, 除非 另有设置.

```
./gradlew browserProductionWebpack
```

注意, 只有在你的编译目标设置为生成可执行文件 (通过 `binaries.executable()`) 时, 这些任务才可用.

CSS

Kotlin Multiplatform Gradle 创建还支持 webpack 的 CSS (https://webpack.js.org/loaders/css-loader/) 和 style (https://webpack.js.org/loaders/style-loader/) 装载器. 虽然所有的选项都可以直接修改构建你的项目的 webpack 配置文件, 但最常用的方法是使用 `build.gradle(.kts)` 中直接可用的设定.

要在你的项目中打开 CSS 支持, 请在 Gradle 构建文件的 `commonWebpackConfig {}` 代码段中设置 `cssSupport.enabled` 选项. 通过 IDE 向导创建新工程时, 这个配置也会默认启用.

Kotlin

```
browser {
    commonWebpackConfig {
        cssSupport {
            enabled.set(true)
        }
```

```
    }
}
```

Groovy

```
browser {
    commonWebpackConfig {
        cssSupport {
            it.enabled.set(true)
        }
    }
}
```

或者, 也可以单独对 `webpackTask {}`, `runTask {}`, 和 `testTask {}` 添加 CSS 支持:

Kotlin

```
browser {
    webpackTask {
        cssSupport {
            enabled.set(true)
        }
    }
    runTask {
        cssSupport {
            enabled.set(true)
        }
    }
    testTask {
        useKarma {
            // ...
            webpackConfig.cssSupport {
                enabled.set(true)
            }
        }
```

```
    }
}
```

Groovy

```
browser {
    webpackTask {
        cssSupport {
            it.enabled.set(true)
        }
    }
    runTask {
        cssSupport {
            it.enabled.set(true)
        }
    }
    testTask {
        useKarma {
            // ...
            webpackConfig.cssSupport {
                it.enabled.set(true)
            }
        }
    }
}
```

对你的项目打开 CSS 支持, 有助于防止在未配置的项目中使用样式表时发生的常见错误, 比如 `Module parse failed: Unexpected character '@' (14:0)`.

可以使用 `cssSupport.mode` 来指定 CSS 应该如何处理. 可选的设定值如下:

- `"inline"` (默认值): 样式添加到全局的 `<style>` tag.
- `"extract"`: 样式抽取为单独的文件. 然后通过 HTML 页面来添加.
- `"import"`: 样式作为字符串处理. 如果你需要在你的代码中访问 CSS, 这会很有用 (比如, `val styles = require("main.css")`).

如果要对同一个项目使用不同的模式, 请使用 cssSupport.rules. 这里, 你可以指定一组 KotlinWebpackCssRules, 其中每一项定义一个 mode, 以及 include (https://webpack.js.org/configuration/module/#ruleinclude) 和 exclude (https://webpack.js.org/configuration/module/#ruleexclude) pattern.

Node.js

对于编译到 Node.js 的 Kotlin/JS 项目, plugin 会在主机上自动下载并安装 Node.js 环境. 如果已经安装过 Node.js, 你也可以使用已经存在的 Node.js.

使用已安装的 Node.js

如果在构建 Kotlin/JS 项目的主机上已经安装了 Node.js, 你可以配置 Kotlin Multiplatform Gradle 插件, 让它使用已安装的 Node.js, 而不要安装另外的 Node.js 实例.

要使用已安装的 Node.js, 请向 build.gradle(.kts) 文件添加以下内容:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets
.js.nodejs.NodeJsRootPlugin> {

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.No
deJsRootExtension>().download = false
    // 默认设置为 "true"
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets
.js.nodejs.NodeJsRootPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.nodejs.NodeJsRootExtension).download = false
}
```

Yarn

为了在构建时下载并安装你声明的依赖项, plugin 会管理它自己的 Yarn (https://yarnpkg.com/lang/en/) 包管理器实例. 不需要额外配置, 默认即可工作, 但你也可以对其进行调整, 或者使用你的主机上已经安装的 Yarn.

Yarn 的更多功能: .yarnrc

如果要配置 Yarn 的更多功能, 请将一个 `.yarnrc` 文件放在你的工程根目录. 编译时, 会自动使用它.

比如, 如果要对 npm 包使用一个自定义的仓库, 请将以下内容添加到工程根目录下的一个 `.yarnrc` 文件中:

```
registry "http://my.registry/api/npm/"
```

关于 `.yarnrc` 文件的更多信息, 请参见 Yarn 官方文档 (https://classic.yarnpkg.com/en/docs/yarnrc/).

使用已安装的 Yarn

如果在构建 Kotlin/JS 项目的主机上已经安装了 Yarn, 你可以配置 Kotlin Multiplatform Gradle 插件, 让它使用已安装的 Yarn, 而不要安装另外的 Yarn 实例.

要使用已安装的 Yarn, 请向 `build.gradle(.kts)` 文件添加以下内容:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin> {

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.Yarn
RootExtension>().download = false
    // 默认设置为 "true"
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
```

```
gets.js.yarn.YarnRootExtension).download = false
}
```

通过 kotlin-js-store 锁定版本

> 通过 kotlin-js-store 锁定版本, 这个功能从 Kotlin 1.6.10 开始可用.

项目根目录下的 kotlin-js-store 目录 由 Kotlin Multiplatform Gradle 插件自动生成, 存储 yarn.lock 文件, 这个文件用来锁定版本. lock 文件完全由 Yarn plugin 管理, 并在 Gradle 任务 kotlinNpmInstall 执行时被更新.

为了遵循 Yarn 推荐的最佳实践 (https://classic.yarnpkg.com/blog/2016/11/24/lockfiles-for-all/), 请将 kotlin-js-store 和其中的内容提交到你的版本管理系统. 这样可以保证你的应用程序在所有的机器上都使用完全相同的依赖项目树来进行构建.

如果需要, 你可以在 build.gradle(.kts) 文件修改目录和 lock 文件名称:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin> {

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.Yarn
RootExtension>().lockFileDirectory =
        project.rootDir.resolve("my-kotlin-js-store")

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.Yarn
RootExtension>().lockFileName = "my-yarn.lock"
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).lockFileDirectory =
```

```
        file("my-kotlin-js-store")

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).lockFileName = 'my-yarn.lock'
}
```

⚠ 修改 lock 文件名称, 可能会导致依赖项检查工具不再正确读取这个文件.

关于 .yarnrc 文件的更多信息, 请参见 Yarn 官方文档 (https://classic.yarnpkg.com/lang/en/docs/yarn-lock/).

报告 yarn.lock 的变更

Kotlin/JS 提供了 Gradle 设置, 可以通知你 yarn.lock 文件是否发生变更. 如果你想要在 CI 构建过程中 yarn.lock 发生变更时收到通知, 你可以使用这些设置:

- YarnLockMismatchReport, 指定对 yarn.lock 文件的变更如何进行报告. 你可以使用以下值之一:
  - FAIL 让对应的 Gradle task 失败. 这是默认设置.
  - WARNING 将变更信息写入警告日志.
  - NONE 禁用报告.
- reportNewYarnLock, 会明确的对最近创建的 yarn.lock 文件进行报告. 默认情况下, 这个选项是禁用的: 常见的做法是在最开始生成一个新的 yarn.lock 文件. 你可以使用这个选项, 来确保 yarn.lock 文件已经提交到了你的代码仓库.
- yarnLockAutoReplace, 会在 Gradle task 每次运行时自动替换 yarn.lock.

要使用这些选项, 请更新 build.gradle(.kts) 文件:

Kotlin

```
import
org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchRepo
rt
import
```

```
org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin::class.java) {
    rootProject.the<YarnRootExtension>().yarnLockMismatchReport
=
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.the<YarnRootExtension>().reportNewYarnLock =
false // true
    rootProject.the<YarnRootExtension>().yarnLockAutoReplace =
false // true
}
```

Groovy

```
import
org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchRepo
rt
import
org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).reportNewYarnLock = false //
true

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).yarnLockAutoReplace = false //
true
}
```

默认使用 --ignore-scripts 安装 npm 依赖项

> 默认使用 `--ignore-scripts` 安装 npm 依赖项, 这个功能从 Kotlin 1.6.10 开始可用.

如果 npm 包被攻击, 其中可能包含恶意代码, 为了减少执行这种恶意代码的可能性, Kotlin Multiplatform Gradle 插件在安装 npm 依赖项时默认会禁止执行 Life Cycle 脚本 (https://docs.npmjs.com/cli/v8/using-npm/scripts#life-cycle-scripts).

你可以明确的允许 Life Cycle 脚本执行, 方法是在 `build.gradle(.kts)` 中添加以下设定:

Kotlin

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin> {

rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.Yarn
RootExtension>().ignoreScripts = false
}
```

Groovy

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets
.js.yarn.YarnPlugin) {

rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.tar
gets.js.yarn.YarnRootExtension).ignoreScripts = false
}
```

设置发布目录

默认设置下, Kotlin/JS 工程编译的输出在工程根目录下的 `/build/dist/<targetName>/<binaryName>` 目录内.

> 在 Kotlin 1.9.0 以前, 默认的发布目录为 `/build/distributions`.

如果要为工程设置另外的发布位置, 请在编译脚本的 `browser {}` 代码段内, 添加 `distribution {}` 代码段, 在这里使用 `set()` 方法为 `outputDirectory` 属性设置一个值. 当你运行工程的构建任务时, Gradle 会将输出的 bundle 文件和工程的资源文件一起, 保存到这个位置.

Kotlin

```
kotlin {
    js {
        browser {
            distribution {

outputDirectory.set(projectDir.resolve("output"))
            }
        }
        binaries.executable()
        // ...
    }
}
```

Groovy

```
kotlin {
    js {
        browser {
            distribution {
                outputDirectory.set(file("$projectDir/output"))
            }
        }
        binaries.executable()
        // ...
    }
}
```

模块名称

如果要调整 JavaScript *模块(module)* 名称 (模块将被生成在 `build/js/packages/myModuleName` 路径), 包括对应的 `.js` 和 `.d.ts` 文件名称, 请使用 `moduleName` 选项:

```
js {
    moduleName = "myModuleName"
}
```

注意, 这个设置不会影响位于 `build/dist` 的 webpacked 输出.

自定义 package.json 文件

`package.json` 文件包含 JavaScript 包的元数据(metadata). 常用的包登记系统, 比如 npm, 要求所有发布的包带有这个文件. 包登记系统会使用这个文件来追踪和管理包的发布.

对 Kotlin/JS 工程, Kotlin Multiplatform Gradle 插件在构建时会自动生成 `package.json`. 这个文件默认会包含最基本的数据: 名称, 版本, 许可证, 依赖项目, 以及包的一些其他属性.

除了基本的包属性之外, `package.json` 还可以定义 JavaScript 包应该如何动作, 比如, 标识可以运行的脚本.

你可以通过 Gradle DSL 向工程的 `package.json` 文件添加自定义的内容. 要添加自定义的项目到你的 `package.json` 文件, 可以在编译任务的 `packageJson` 代码段中使用 `customField()` 函数:

```
kotlin {
    js {
        compilations["main"].packageJson {
            customField("hello", mapOf("one" to 1, "two" to 2))
        }
    }
}
```

构建工程时, 这段代码会向 `package.json` 文件添加以下内容:

```
"hello": {
    "one": 1,
    "two": 2
}
```

关于如何为 npm 登记项目编写 `package.json` 文件, 详情请参见 npm 文档 (https://docs.npmjs.com/cli/v6/configuring-npm/package-json).

错误排查

使用 Kotlin 1.3.xx 构建 Kotlin/JS 项目时, 如果你的某个依赖项 (或任何一个传递依赖项) 使用 Kotlin 1.4 或更高版本构建, 你可能会遇到 Gradle 错误: `Could not determine the dependencies of task ':client:jsTestPackageJson'.`/`Cannot choose between the following variants`. 这是一个已知的问题, 解决方法请参见 这个页面 (https://youtrack.jetbrains.com/issue/KT-40226).

运行 Kotlin/JS 代码

最终更新: 2024/09/10

Kotlin/JS 项目是通过 Kotlin Multiplatform Gradle plugin 管理的, 因此你可以使用适当的 Gradle 任务来运行你的项目. 如果你从一个空的项目开始, 请确认你已有一些示例代码可以运行. 创建文件 `src/jsMain/kotlin/App.kt`, 并在其内容中输入一段 "Hello, World" 式的代码:

```
fun main() {
    console.log("Hello, Kotlin/JS!")
}
```

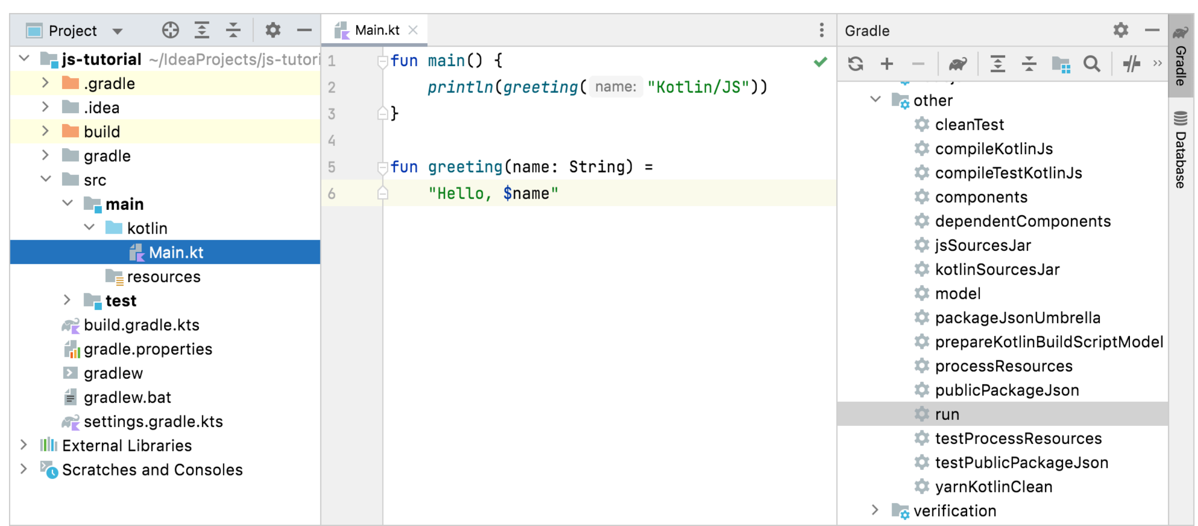
根据目标平台不同, 初次运行你的代码时, 可能需要一些额外的平台相关的设置.

在 Node.js 平台运行

当 Kotlin/JS 项目的编译目标为 Node.js 时, 你可以直接运行 Gradle 的 `jsRun` 任务. 比如, 在命令行环境, 使用 Gradle wrapper:

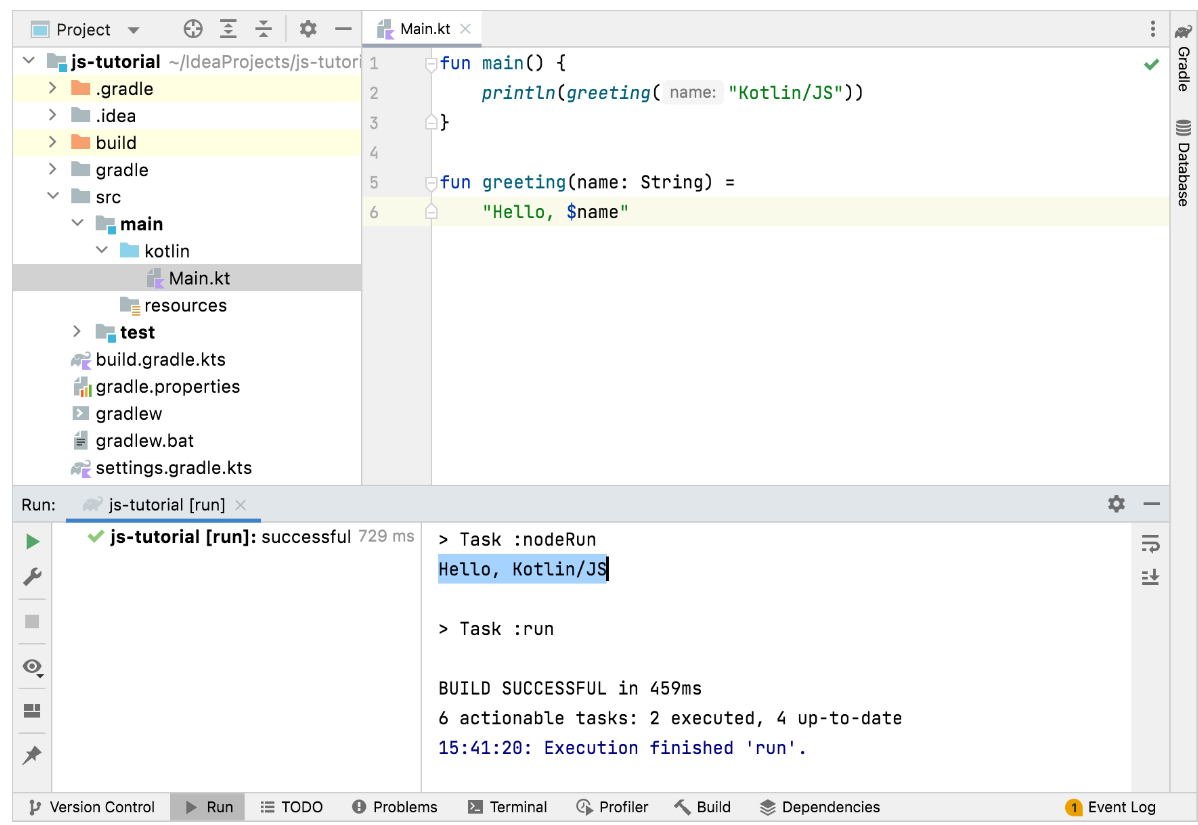
```
./gradlew jsRun
```

如果使用 IntelliJ IDEA, 你可以在 Gradle 工具窗口找到 `jsRun` 任务:



IntelliJ IDEA 中的 Gradle Run 任务

初次运行时, `kotlin.multiplatform` Gradle plugin 会下载所有需要的依赖项, 准备运行环境. 构建完成后, 程序会被执行, 你可以在终端看到输出:



在 IntelliJ IDEA 的 Kotlin Multiplatform 项目中执行 JS 编译目标

在浏览器平台运行

当 Kotlin/JS 项目的编译目标为浏览器时, 你的项目需要有一个 HTML 页面. 在开发你的应用程序时, 这个页面由开发服务器提供, 它应该嵌入你编译后的 Kotlin/JS 文件. 创建一个 HTML 文件 `/src/jsMain/resources/index.html`, 内容如下:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JS Client</title>
</head>
<body>
<script src="js-tutorial.js"></script>
</body>
</html>
```

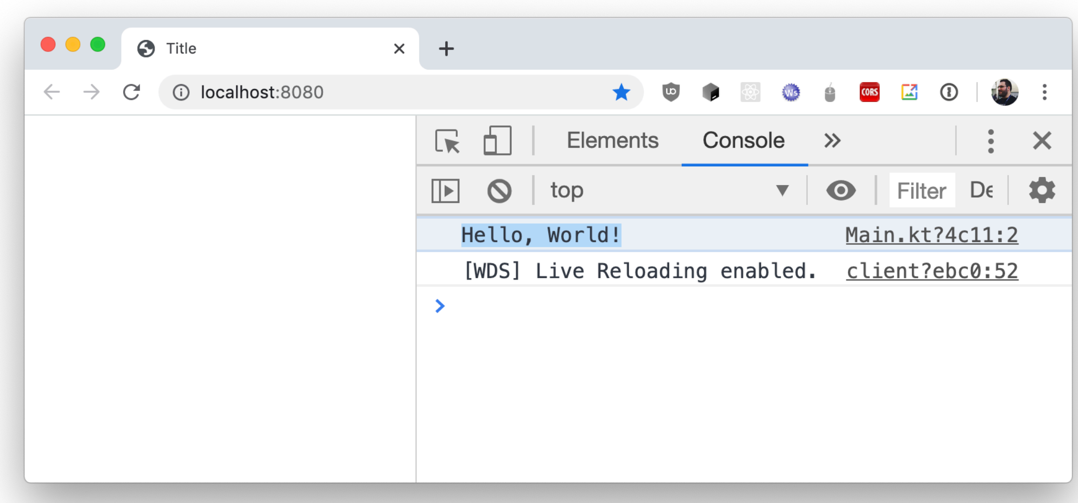
我们需要引用你的项目(通过 webpack 创建)生成的 js 文件. 默认情况下, js 文件的名称就是你的项目的名称 (这个示例中是, `js-tutorial`). 如果你的项目命名为 `followAlong`, 请注意要嵌入 `followAlong.js` 而不是 `js-tutorial.js`

完成这些调整之后, 启动集成的开发服务器. 你可以在命令行, 使用 Gradle wrapper:

```
./gradlew jsRun
```

使用 IntelliJ IDEA 时, 你可以在 Gradle 工具窗口找到 `jsRun` 任务.

项目构建完成后, 内嵌的 `webpack-dev-server` 会开始运行, 并会打开一个 (似乎是空白的) 浏览器窗口, 指向你之前指定的 HTML 文件. 要验证你的程序是否正确运行, 打开你的浏览器的开发者工具 (比如, 点击鼠标右键, 选择 *Inspect* 菜单项). 在开发者工具中, 打开控制台, 你可以看到 JavaScript 代码的执行结果:



浏览器的开发者工具中的控制台输出

通过这样的设置, 你可以在每次代码之后修改, 重新编译你的项目来查看你的修改结果. Kotlin/JS 还支持更方便的方式, 在开发过程中自动构建应用程序. 关于这种 *持续编译模式* 的设置方式, 请参见 开发服务器(Development server)与持续编译(Continuous Compilation) (开发服务器(Development server)与持续编译(Continuous Compilation)).

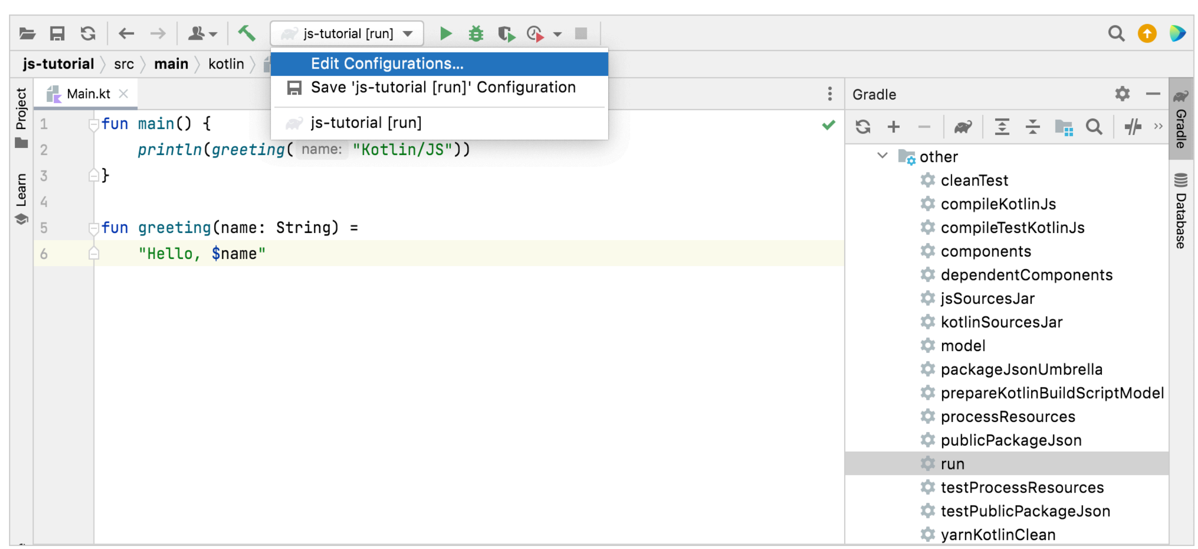
开发服务器(Development server)与持续编译(Continuous Compilation)

最终更新: 2024/09/10

你可以使用 *持续编译(Continuous Compilation)* 模式, 这样就不必每次想要查看修改结果时手动编译和执行 Kotlin/JS 项目. 使用 *持续(Continuous)* 模式调用 Gradle wrapper, 而不是不是使用通常的 `run` 命令:

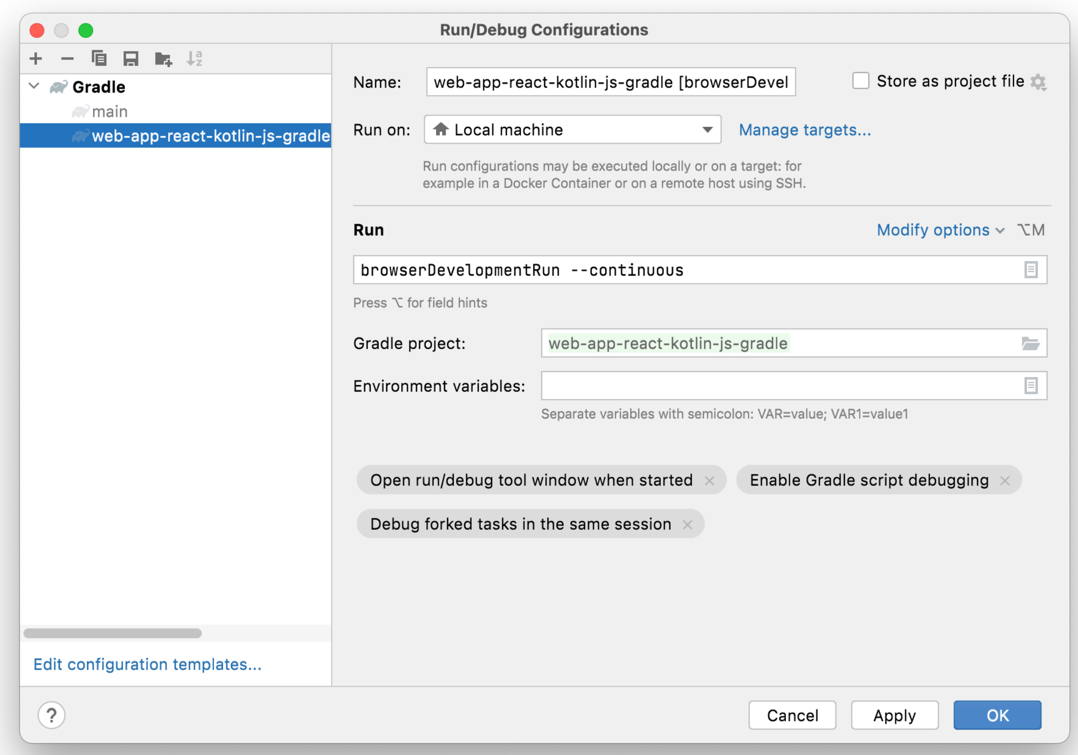
```
./gradlew run --continuous
```

如果你使用 IntelliJ IDEA, 那么可以通过 *运行配置(Run Configuration)* 传递相同的选项. 从 IDE 中初次运行 Gradle `run` task 后, IntelliJ IDEA 会自动生成运行配置, 然后你可以修改这个配置:



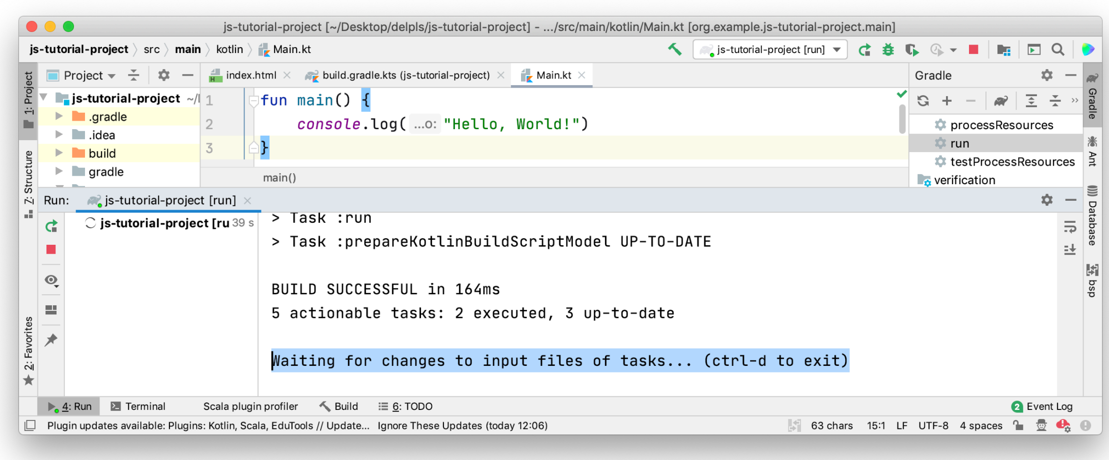
在 IntelliJ IDEA 中修改运行配置

在 **Run/Debug Configurations** 对话框中开启 持续(Continuous)模式, 只需要在运行配置的参数中添加 `--continuous` 选项:



在 IntelliJ IDEA 中向运行配置添加 continuous 选项

执行这个运行配置时, 你可以注意到 Gradle 进程会持续监视项目文件的变更:



Gradle 等待文件变更

一旦检测到文件变更, 你的程序会被自动重新编译. 如果你在浏览器中打开了页面, 开发服务器会触发页面自动更新, 然后你的变更会反应到页面中. 这是由 Kotlin Multiplatform Gradle plugin 管理

的 `webpack-dev-server` 提供的功能.

调试 Kotlin/JS 代码

最终更新: 2024/09/10

JavaScript 代码映射(Source Map), 提供了由打包器(bundler)或极简化器(minifier)产生的极简化代码与开发者编写的真实代码之间的对应关系. 通过这种方式, 代码映射可以支持代码执行时的调试.

Kotlin Multiplatform Gradle plugin 会为它构建的项目自动生成代码映射, 不需要任何额外的配置.

在浏览器中调试

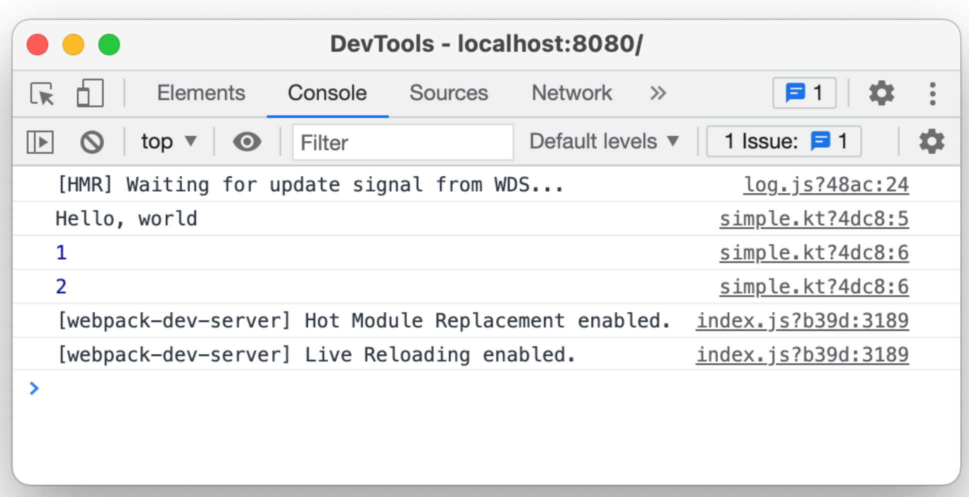
大多数现代浏览器提供了工具, 可以查看页面内容, 调试页面中执行代码. 详情请参见你的浏览器的文档.

要在浏览器中调试 Kotlin/JS, 请执行以下步骤:

1. 执行 Gradle 的某个 *run* 任务来运行项目, 比如, 跨平台项目内的 `browserDevelopmentRun` 或 `jsBrowserDevelopmentRun`. 详情请参见 运行 Kotlin/JS ("在浏览器平台运行" in "运行 Kotlin/JS 代码").

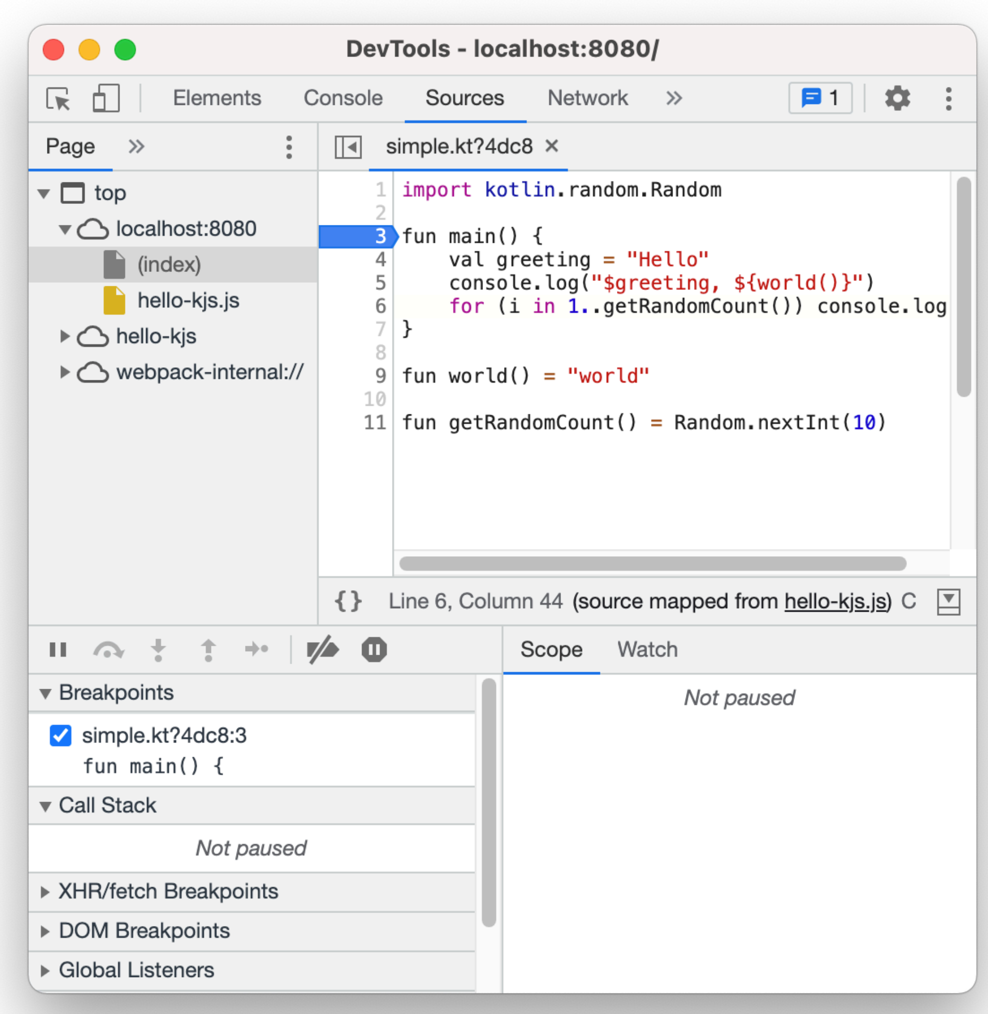
2. 在浏览器中访问页面, 并启动浏览器的开发者工具(比如, 点击鼠标右键, 选择 **Inspect** 菜单项). 详情请参见在流行的浏览器中 如何找到开发者工具 (https://balsamiq.com/support/faqs/browserconsole/).

3. 如果你的程序向控制台输出 log, 请打开 **Console** 页, 查看其中的输出. 根据你使用的浏览器不同, 这些 log 可能会参照到输出 log 的 Kotlin 源代码文件和行号:



Chrome 的 DevTools 控制台

4. 点击右侧的文件参照, 即可浏览源代码中对应的行. 或者, 你也可以手动切换到 **Sources** 页, 在文件树中找到你需要的文件. 浏览 Kotlin 文件, 会显示通常的 Kotlin 代码 (而不是极简化之后的 JavaScript 代码):



在 Chrome 的 DevTools 中调试

现在你可以开始调试程序了. 点击代码的行号可以设置一个断点. 开发者工具甚至还支持在一条语句内设置断点. 和通常的 JavaScript 代码一样, 在页面重新加载之后, 设置的任何断点都会继续存在. 因此可以调试 Kotlin 的 `main()` 方法, 这个方法会在脚本初次装载时执行.

在 IDE 中调试

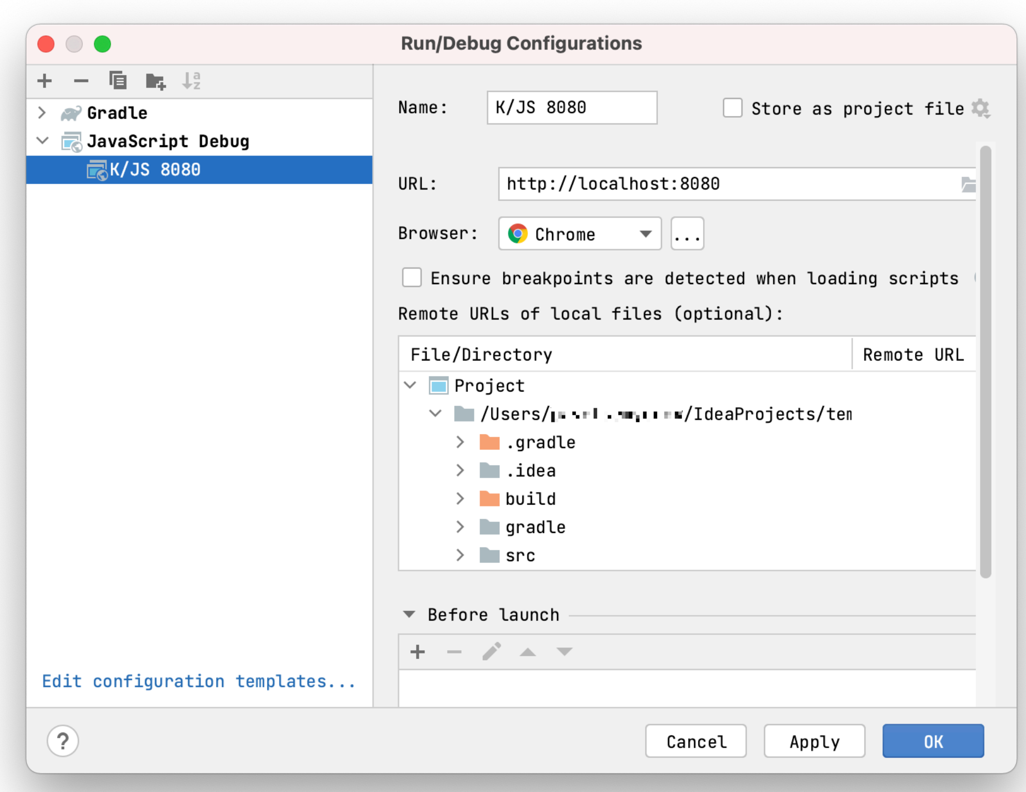
IntelliJ IDEA Ultimate (https://www.jetbrains.com/idea/) 提供了强大的工具用于开发时调试代码.

要在 IntelliJ IDEA 中调试 Kotlin/JS, 你需要一个 **JavaScript Debug** 配置. 要添加一个这样的调试配置, 请执行以下步骤:

1. 选择菜单 **Run | Edit Configurations.**

2. 点击 **+**, 选择 **JavaScript Debug.**

3. 指定配置的 **Name**, 并提供项目运行时的 **URL** (默认是 `http://localhost:8080`).



JavaScript 调试配置

4. 保存配置.

详情请参见 设置 JavaScript 调试配置 (https://www.jetbrains.com/help/idea/configuring-javascript-debugger.html).

现在你可以调试你的项目了!

1. 执行 Gradle 的某个 *run* 任务来运行项目, 比如, 跨平台项目的 `browserDevelopmentRun` 或 `jsBrowserDevelopmentRun`. 详情请参见 运行 Kotlin/JS (["在浏览器平台运行" in "运行 Kotlin/JS 代码"]()).

2. 运行你前面创建的 JavaScript 调试配置, 启动调试会话:

hello-kjs – simple.kt [hello-kjs.main]

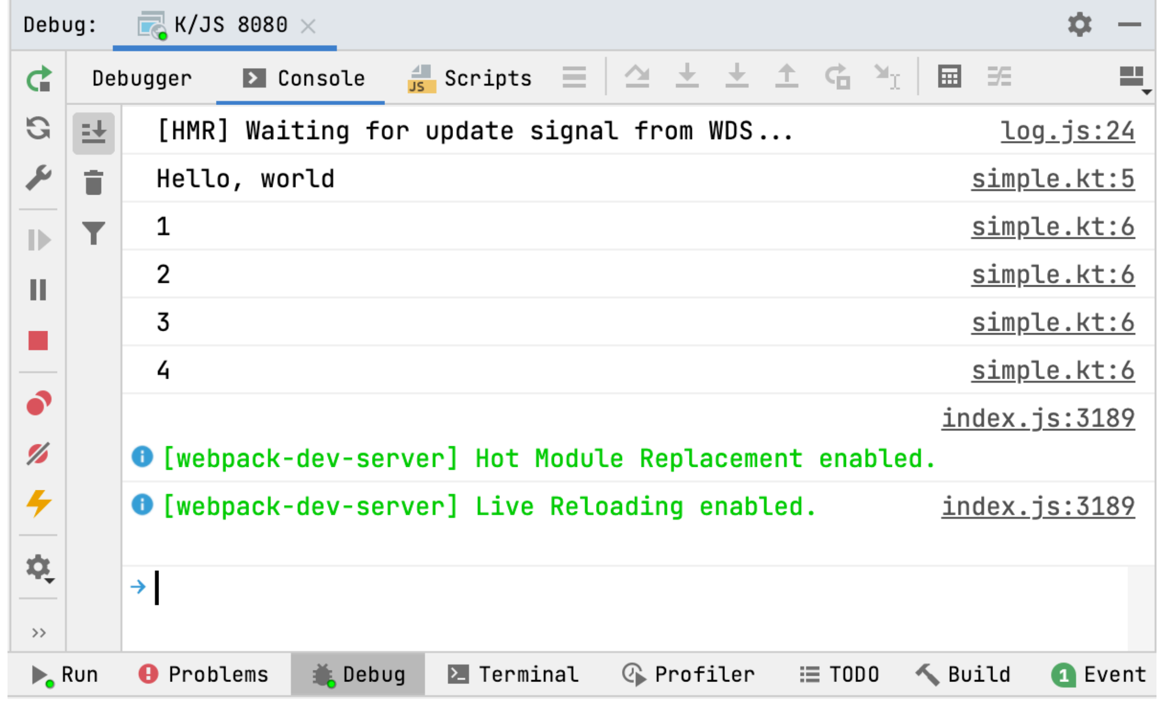
getRandomCount()　　K/JS 8080

simple.kt　index.html　favicon.ico　　Debug 'K/JS 8080'

```kotlin
import kotlin.random.Random

fun main() {
    val greeting = "Hello"
    console.log( ...o: "$greeting, ${world()}")
    for (i in 1..getRandomCount()) console.log(i)
}

fun world() = "world"

fun getRandomCount() = Random.nextInt( until: 10)
```

JavaScript 调试配置

3. 在 IntelliJ IDEA 的**Debug** 窗口中, 你可以看到你的程序的控制台输出. 输出项目会参照到输出 log 的 Kotlin 源代码文件和行号:



在 IDE 中的 JavaScript 调试输出

4. 点击右侧的文件参照, 即可浏览源代码中对应的行.

现在你可以使用 IDE 提供的全套工具调试程序了: 断点, 单步运行, 表达式计算, 等等. 详情请参见 在 IntelliJ IDEA 中进行调试 (https://www.jetbrains.com/help/idea/debugging-javascript-in-chrome.html).

> 由于 IntelliJ IDEA 中的 JavaScript 调试器目前的限制, 你可能会需要重新运行 JavaScript 调试, 才能让程序在断点处暂停.

在 Node.js 中调试

如果你的项目编译目标是 Node.js, 你可以在运行时调试它.

要调试一个编译目标为 Node.js 的 Kotlin/JS 应用程序, 请执行以下步骤:

1. 运行 Gradle 的 `build` 任务, 构建项目.
2. 在你的项目目录下的 `build/js/packages/your-module/kotlin/` 目录中, 找到针对 Node.js 输出的 `.js` 文件.
3. 参照 Node.js 调试指南 (https://nodejs.org/en/docs/guides/debugging-getting-started/#jetbrains-webstorm-2017-1-and-other-jetbrains-ides) 中的方法, 在 Node.js 中调试这个文件.

下一步做什么?

现在你了解了如何调试你的 Kotlin/JS 项目, 请阅读以下资料, 学习如何高效使用调试工具:

- 学习如何 在 Google Chrome 中调试 JavaScript (https://developer.chrome.com/docs/devtools/javascript/)
- 熟悉 IntelliJ IDEA JavaScript 调试器 (https://www.jetbrains.com/help/idea/debugging-javascript-in-chrome.html)
- 学习如何 在 Node.js 中调试 (https://nodejs.org/en/docs/guides/debugging-getting-started/).

如果你遇到问题

如果你遇到任何与调试 Kotlin/JS 相关的问题, 请到我们的问题追踪系统 YouTrack (https://kotl.in/issue) 提交报告.

在 Kotlin/JS 平台进行测试

最终更新: 2024/09/10

Kotlin Multiplatform Gradle plugin 允许你使用多种不同的测试运行器来运行测试, 测试运行器可以通过 Gradle 配置来指定.

当你创建跨平台项目时, 你可以在 `commonTest` 中使用一个依赖项, 对所有的源代码集添加测试依赖项, 包括 JavaScript 编译目标:

Kotlin

```
// build.gradle.kts

kotlin {
    sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test")) // 添加这个设置, 可以在
JS 中使用测试相关的注解和功能
        }
    }
}
```

Groovy

```
// build.gradle

kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // 添加这个设置, 可以
在 JS 中使用测试相关的注解和功能
            }
        }
```

```
    }
}
```

你可以在 Gradle 构建脚本中修改 `testTask` 代码段内的设定, 对 Kotlin/JS 中如何执行测试进行调节. 比如, 要使用 Karma 测试运行器, 和 headless 的 Chrome 浏览器, 和一个 Firefox 浏览器, 设置如下:

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useChromeHeadless()
                    useFirefox()
                }
            }
        }
    }
}
```

关于具体功能的详细说明, 请参见 Kotlin/JS 文档的 配置测试任务 ("test 任务" in "创建 Kotlin/JS 工程(Project)") 小节.

请注意, plugin 默认没有绑定浏览器. 也就是说, 你需要确保在目标系统中存在需要的浏览器.

要检查测试是否正确执行, 请添加一个 `src/jsTest/kotlin/AppTest.kt` 文件, 内容如下:

```
import kotlin.test.Test
import kotlin.test.assertEquals

class AppTest {
    @Test
    fun thingsShouldWork() {
        assertEquals(listOf(1,2,3).reversed(), listOf(3,2,1))
    }

    @Test
    fun thingsShouldBreak() {
        assertEquals(listOf(1,2,3).reversed(), listOf(1,2,3))
```

```
        }
    }
```

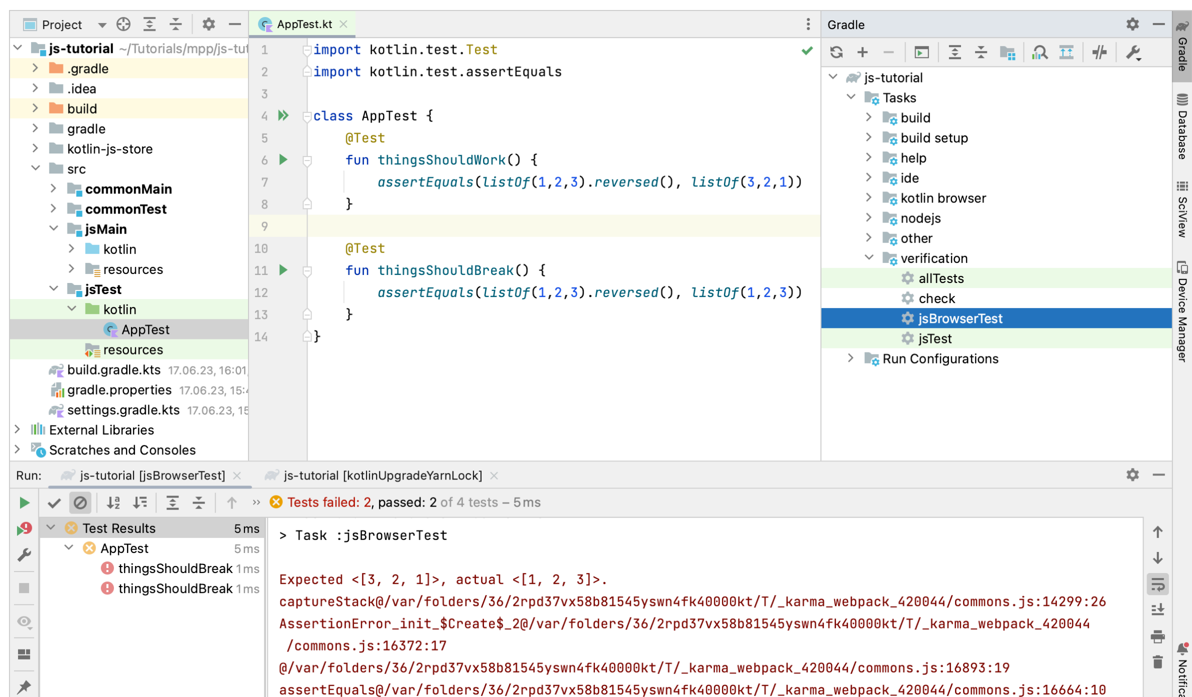
要在浏览器中运行测试, 请通过 IntelliJ IDEA 执行 `jsBrowserTest` 任务, 或使用源代码编辑器侧栏中的图标来执行全部测试, 或单独的某个测试:

Gradle 的 browserTest 任务

或者, 如果你想要通过命令行运行测试, 请使用 Gradle wrapper:

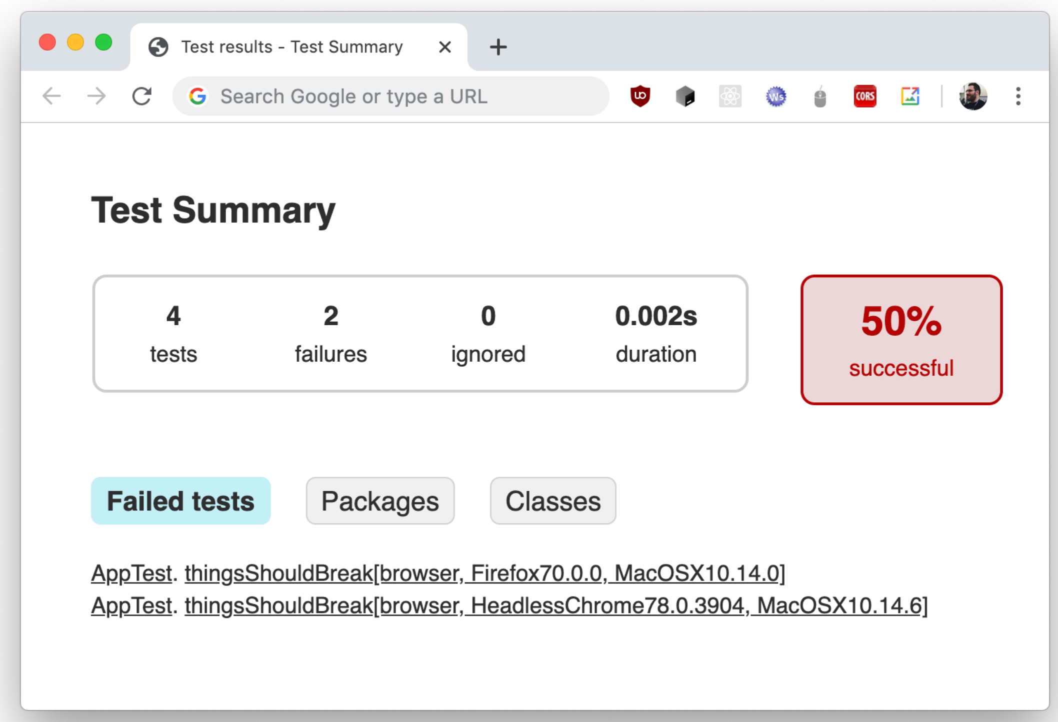
```
./gradlew jsBrowserTest
```

从 IntelliJ IDEA 运行测试之后, **Run** 工具窗口会显示测试结果. 你可以点击失败的测试, 查看它们的栈追踪(Stacktrace), 也可以双击测试结果, 查看对应的测试实现代码.



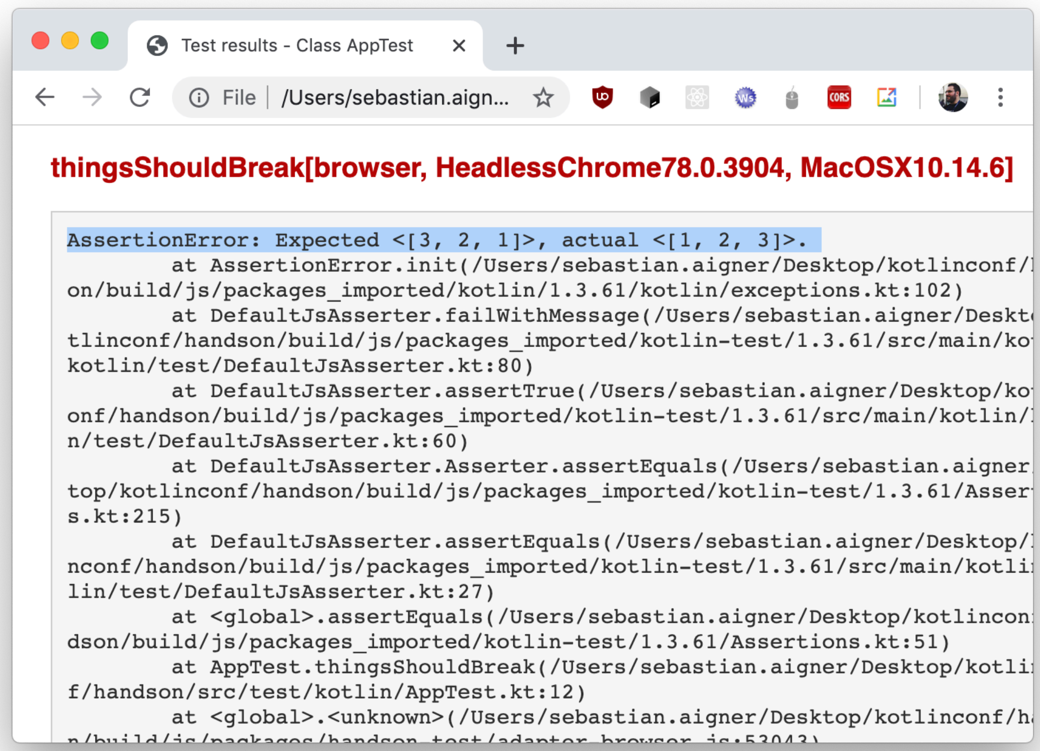
IntelliJ IDEA 中的测试结果

每次测试运行之后，无论你通过什么方式运行测试，你都可以找到 Gradle 生成的适当格式化的测试报告，位置是`build/reports/tests/jsBrowserTest/index.html`. 可以在浏览器打开这个文件，查看测试结果报告：



Gradle 测试报告

如果你使用上述示例代码中的那组测试, 一个测试会通过, 另一个测试会失败, 因此测试全体 50% 成功. 要查看各个测试用例的详细信息, 你可以通过页面内的链接进行浏览:



测试报告中失败的测试的栈追踪(Stacktrace)

JavaScript 死代码剔除工具

最终更新: 2024/09/10

Kotlin Multiplatform Gradle 插件包含一个 *死代码剔除(Dead Code Elimination)* (*https://wikipedia.org/wiki/Dead_code_elimination*) (*DCE*) 工具. 死代码剔除通常又被称为 *摇树(Tree Shaking)*. 它可以删除未被使用的属性, 函数, 以及类, 减少最终编译输出结果的 JavaScript 代码大小.

有几种情况可以导致代码中存在未被使用的声明:

- 函数可能会被内联, 因此不会被直接调用 (除极少数情况外, 总是会如此).
- 模块使用了一个共享库. 如果没有 DCE, 库中没有被用到的部分仍然会包含在编译输出的 bundle 之内. 比如, Kotlin 标准库包含了许多函数, 用于操作列表, 数组, 字符序列, 用于 DOM 的适配器, 等等. 所有这些功能输出为 JavaScript 文件总计需要 1.3 MB. 而一个简单的 "Hello, world" 应用程序只需要控制台相关函数, 整个文件只有几 KB.

构建 **产品版(production) bundle** 时, Kotlin Multiplatform Gradle 插件会自动处理 DCE, 比如, 使用 `browserProductionWebpack` 任务. **开发版(development) bundle** 的构建任务(比如 `browserDevelopmentWebpack`) 不会包含 DCE.

在 DCE 中排除一部分函数或类声明

有些时候, 即使在你的模块中并没有使用某个函数或类, 但你可能需要在最终输出的 JavaScript 代码中保留它, 比如, 你可能要在客户端 JavaScript 代码中使用它.

要在死代码剔除时保留这些函数或类的声明, 请在 Gradle 构建脚本中添加 `dceTask` 代码段, 并在 `keep` 函数的参数中列出需要保留的函数或类声明. 参数必须是这个声明的完整限定名称, 包含模块名作为前缀, 比如: `moduleName.dot.separated.package.name.declarationName`

> 除非另有设置, 否则在生成的 JavaScript 代码中, 函数和模块的名称会被 混淆(mangle) ("@JsName 注解" in "在 JavaScript 中使用 Kotlin 代码"). 如果要在 DCE 中保持这些函数不被剔除, 需要在 `keep` 的参数中使用它们混淆之后出现在生成的 JavaScript 代码中的名称.

```
kotlin {
    js {
```

```
        browser {
            dceTask {
                keep("myKotlinJSModule.org.example.getName",
"myKotlinJSModule.org.example.User" )
            }
            binaries.executable()
        }
    }
}
```

如果希望保持整个包或模块不被剔除, 可以使用它出现在生成的 JavaScript 代码中的完整限定名称.

> 保持整个包或模块不被剔除, 会阻碍 DCE 删除许多未被使用的声明. 因此, 最好逐个选择需要从 DCE 中排除的声明.

禁用 DCE

如果需要完全关闭 DCE 功能, 可以在 `dceTask` 中使用 `devMode` 选项:

```
kotlin {
    js {
        browser {
            dceTask {
                dceOptions.devMode = true
            }
        }
        binaries.executable()
    }
}
```

使用 IR 编译器

最终更新: 2024/09/10

Kotlin/JS IR 编译器后端是 Kotlin/JS 的主要创新方向, 并为以后的技术发展探索道路.

Kotlin/JS IR 编译器后端不是从 Kotlin 源代码直接生成 JavaScript 代码, 而是使用一种新方案. Kotlin 源代码首先转换为 Kotlin 中间代码(intermediate representation, IR) ("统一的后端和扩展性" in "Kotlin 1.4.0 版中的新功能"), 然后再编译为 JavaScript. 对于 Kotlin/JS, 这种方案可以实现更加积极的优化, 并能够改进以前的编译器中出现的许多重要问题, 比如, 生成的代码大小(通过死代码清除), 以及 JavaScript 和 TypeScript 生态环境的交互能力, 等等.

从 Kotlin 1.4.0 开始, 可以通过 Kotlin Multiplatform Gradle 插件使用 IR 编译器后端. 要在你的项目中启用它, 需要在你的 Gradle 构建脚本中, 向 `js` 函数传递一个编译器类型参数:

```
kotlin {
    js(IR) { // 或者: LEGACY, BOTH
        // ...
        binaries.executable() // 不兼容 BOTH, 详情请见下文
    }
}
```

- `IR` 对 Kotlin/JS 使用新的 IR 编译器后端.
- `LEGACY` 使用旧的编译器后端.
- `BOTH` 编译项目时使用新的 IR 编译器以及默认的编译器后端. 主要用于 编写同时兼容于两种后端的库.

> 从 Kotlin 1.8.0 开始, 旧的编译器后端已被废弃. 从 Kotlin 1.9.0 开始, 使用 `LEGACY` 或 `BOTH` 编译器类型会发生错误.

编译器类型也可以在 `gradle.properties` 文件中通过 `kotlin.js.compiler=ir` 来设置. 但是这个设置会被 `build.gradle(.kts)` 中的任何设置覆盖.

顶级属性(top-level property)的延迟初始化(Lazy initialization)

为了改善应用程序的启动速度, Kotlin/JS IR 编译器会对顶级属性(top-level property)进行延迟初始化(Lazy initialization). 通过这种方式, 应用程序启动时不会初始化它的代码中的全部顶级属性. 而只会初始化在启动阶段需要的那些顶级属性; 其他属性的初始化会被延迟, 直到使用它们的代码真正被执行时才会生成属性值.

```
val a = run {
    val result = // 假设这里是一段计算密集的代码
    println(result)
    result
} // 属性值直到初次使用时才会计算
```

如果由于某些原因你需要(在应用程序启动阶段)提早初始化一个属性, 可以对它标注 `@EagerInitialization` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/-eager-initialization/) 注解.

对开发阶段二进制文件进行增量编译

JS IR 编译器提供了 *对开发阶段二进制文件的增量编译模式*, 可以对开发过程提高速度. 在这种模式下, 编译器会在模型层级缓存 Gradle task `compileDevelopmentExecutableKotlinJs` 的结果. 在后续的编译中, 对未修改的源代码文件使用缓存的编译结果, 可以使得编译更快完成, 尤其是在对代码进行少量修改的情况.

增量编译是默认启用的. 如果要对开发阶段二进制文件禁用增量编译, 请向项目的 `gradle.properties` 或 `local.properties` 文件添加以下设置:

```
kotlin.incremental.js.ir=false // 默认为 true
```

> 在增量编译模式中, 完整编译通常会变得更慢, 因为需要创建和生成缓存.

输出 .js 文件: 对每模块输出一个文件, 或对整个项目输出一个文件

作为编译结果, JS IR 编译器对项目的每个模块输出单独的 `.js` 文件. 你也可以选择将整个项目编译为单个 `.js` 文件, 方法是向 `gradle.properties` 添加以下设置:

```
kotlin.js.ir.output.granularity=whole-program // 默认为 'per-module'
```

忽略编译错误

> ⚠ *忽略编译错误* 模式还处于 实验阶段 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 使用这个功能需要明确要求使用者同意(详情请见下文), 而且你应该只用来进行功能评估, 不要用在你的正式产品中. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

Kotlin/JS IR 编译器提供了一个在默认的编译器后端中没有的新编译模式 – *忽略编译错误* 模式. 在这个模式下, 即使代码包含错误, 你也可以试用你的应用程序. 比如, 你可能在进行一个非常复杂的代码重构时, 或者在编写系统的某一部分, 与发生编译错误的另一部分完全无关.

在这个新的编译器模式下, 编译器会忽略所有的错误代码. 因此, 你可以运行应用程序, 并试用与错误代码无关的那部分功能. 如果你试图运行那些存在编译错误的代码, 那么将会发生运行期异常.

要忽略你代码中的编译错误, 可以选择两种错误宽容策略:

- `SEMANTIC`. 编译器会接受语法正确但语义上无意义的代码. 比如, 将数值赋值给字符串变量 (类型不匹配).
- `SYNTAX`. 编译器会接受任何代码, 即使包含语法错误. 无论你编写什么样的代码, 编译器都会尝试生成可执行的代码.

作为一个试验性的功能, 忽略编译错误需要使用者同意(Opt-in). 要开始这个模式, 需要添加 `-Xerror-tolerance-policy={SEMANTIC|SYNTAX}` 编译器选项:

```
kotlin {
    js(IR) {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions.freeCompilerArgs.add("-Xerror-
tolerance-policy=SYNTAX")
            }
        }
    }
}
```

在产品(Production)模式中对成员名称的极简化(Minification)

Kotlin/JS IR 编译器会使用它的内部信息 关于 你的 Kotlin 类和函数之间的关系, 来实现更加有效的极简化(Minification), 缩短函数, 属性, 和类的名称. 这样可以缩减打包完成的应用程序的大小.

当你使用 产品(Production) ("构建可执行文件" in "创建 Kotlin/JS 工程(Project)") 模式构建你的 Kotlin/JS 应用程序时, 会自动应用这样的极简化处理, 并默认启用. 要关闭对成员名称的极简化处理, 请使用 `-Xir-minimized-member-names` 编译器选项:

```
kotlin {
    js(IR) {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions.freeCompilerArgs.add("-Xir-
minimized-member-names=false")
            }
        }
    }
}
```

预览: 生成 TypeScript 声明文件 (d.ts)

> 生成 TypeScript 声明文件 (`d.ts`)功能还处于 实验阶段 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 使用这个功能需要明确要求使用者同意(详情请见下文), 而且你应该只用来进行功能评估, 不要用在你的正式产品中. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues?q=%23%7BKJS:%20d.ts%20generation%7D) 提供你的反馈意见.

Kotlin/JS IR 编译器能够从你的 Kotlin 代码生成 TypeScript 定义. 在开发混合 App(hybrid app)时, JavaScript 工具和 IDE 可以使用这些定义, 来提供代码自动完成, 支持静态分析, 使得在 JavaScript 和 TypeScript 项目中包含 Kotlin 代码变得更加便利.

如果你的项目输出可执行文件 (`binaries.executable()`), Kotlin/JS IR 编译器会收集所有标注了 `@JsExport` ("@JsExport 注解" in "在 JavaScript 中使用 Kotlin 代码") 注解的顶级声明, 并自动在一个 `.d.ts` 文件中生成 TypeScript 定义.

如果你想要生成 TypeScript 定义, 你需要在 Gradle 构建文件中明确进行配置. 请在你的 `build.gradle.kts` 文件的 `js` 小节 ("执行环境" in "创建 Kotlin/JS 工程(Project)") 中添加 `generateTypeScriptDefinitions()`. 例如:

```
kotlin {
    js {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

这些声明位于 `build/js/packages/<package_name>/kotlin` 目录中, 与相应的未经 webpack 处理的 JavaScript 代码在一起.

IR 编译器目前的限制

新的 IR 编译器后端的一个重要变化是与默认后端之间 **没有二进制兼容性**. 使用新的 IR 编译器后端创建的库, 会使用 `klib` 格式 ("库文件的格式" in "Kotlin/Native 库"), 在默认后端中将无法使用. 同时, 使用旧编译器创建的库, 就是一个包含 `js` 文件的 `jar`, 也不能在 IR 后端中使用.

如果你希望在你的项目中使用 IR 编译器后端, 那么需要 **将所有的 Kotlin 依赖项升级到支持这个新后端的版本**. JetBrains 针对 Kotlin/JS 平台, 对 Kotlin 1.4+ 版本发布的库, 已经包含了使用新的 IR 编译器后端所需要的全部 artifact.

如果你是库的开发者, 希望为现在的编译器后端和新的 IR 编译器后端同时提供兼容性, 请阅读 针对 IR 编译器编写库 小节.

与默认后端相比, IR 编译器后端还存在一些差异. 试用新的后端时, 需要知道存在这些可能的问题.

- 有些 **库依赖于默认后端的独有的特性**, 比如 `kotlin-wrappers`, 可能会出现一些问题. 你可以通过 YouTrack (https://youtrack.jetbrains.com/issue/KT-40525) 跟踪这个问题的调查结果和进展.
- 默认情况下, IR 后端 **完全不会让 Kotlin 声明在 JavaScript 中可见**. 要让 JavaScript 可以访问 Kotlin 声明, 这些声明 **必须** 添加 `@JsExport` ("@JsExport 注解" in "在 JavaScript 中使用 Kotlin 代码") 注解.

将既有的项目迁移到 IR 编译器

由于两种 Kotlin/JS 编译器的接口签名的不同, 要让你的 Kotlin/JS 代码能由 IR 编译器来编译, 可能需要你修改部分代码. 关于如何将既有的 Kotlin/JS 工程迁移到 IR 编译器, 请参见 Kotlin/JS IR 编译

器 迁移向导 (将 Kotlin/JS 项目迁移到 IR 编译器).

针对 IR 编译器开发向后兼容的库

如果你是库的维护者，希望同时兼容默认后端和新的 IR 编译器后端，有一种编译器选择设置可以让你对两种后端都创建 artifact，因此可以支持下一代 Kotlin 编译器，同时又对你的既有用户保持兼容性. 这就是所谓的 `both` 模式，可以在 `gradle.properties` 文件中通过 `kotlin.js.compiler=both` 来启用，或者，也可以在 `build.gradle(.kts)` 文件的 `js` 代码块之内，设置为项目独有的选项:

```
kotlin {
    js(BOTH) {
        // ...
    }
}
```

使用 `both` 模式时，从你的源代码构建库时，IR 编译器后端和默认编译器后端都会被使用(如同它的名称所指的那样). 也就是说，Kotlin IR 的 `klib` 文件，以及默认编译器的 `jar` 文件都会生成. 当发布到相同的 Maven 路径时，Gradle 会根据使用场景自动选择正确的 artifact – 对旧的编译器是 `js`，对新的编译器是 `klib`. 因此对于使用两种编译器后端中的任何一个的项目，你都可以编译并发布你的库.

将 Kotlin/JS 项目迁移到 IR 编译器

最终更新: 2024/09/10

我们已经用 基于 IR 的编译器 (使用 IR 编译器) 替代了旧的 Kotlin/JS 编译器, 原因是为了在所有的平台上统一 Kotlin 的行为, 以及能够实现新的 JS 专有的优化, 还有其他一些原因. 关于两种编译器的内部区别, 请参见 Sebastian Aigner 的 Blog 将我们的 Kotlin/JS 应用程序迁移到新的 IR 编译器 (https://dev.to/kotlin/migrating-our-kotlin-js-app-to-the-new-ir-compiler-3o6i).

由于编译器之间的显著区别, 将你的 Kotlin/JS 项目从旧的编译器后端切换到新的, 可能会需要调整你的代码. 本章中, 我们会列举已知的迁移问题, 以及建议的解决方案.

> 关于如何修正迁移期间发生的某些问题, 请安装 Kotlin/JS Inspection pack (https://plugins.jetbrains.com/plugin/17183-kotlin-js-inspection-pack/) plugin, 可以得到有价值的提示.

注意, 由于我们修正了问题, 或者发现了新的问题, 本向导将来可能会发生变更. 请帮助我们完善这些信息 – 请报告你切换到 IR 编译器时遇到的问题, 提交到我们的问题追踪系统 YouTrack (https://kotl.in/issue), 或填写这个 表格 (https://surveys.jetbrains.com/s3/ir-be-migration-issue).

将 JS 和 React 相关的类和接口转换为外部接口(External Interface)

问题: 使用 Kotlin 接口和类 (包括数据类), 如果继承自纯 JS 类, 比如 React 的 `State` 和 `Props`, 可能导致 `ClassCastException` 异常. 出现这样的异常是因为, 编译器试图将这些类的实例象 Kotlin 对象一样使用, 然而它们实际上来自 JS.

解决方案: 将所有继承自纯 JS 类的类和接口转换为 外部接口(External Interface) ("external 接口" in "在 Kotlin 中使用 JavaScript 代码"):

```
// 替换以下代码
interface AppState : State { }
interface AppProps : Props { }
data class CustomComponentState(var name: String) : State
```

```
// 替换为
external interface AppState : State { }
```

```
external interface AppProps : Props { }
external interface CustomComponentState : State {
    var name: String
}
```

在 IntelliJ IDEA 中, 你可以使用这些 结构化查找与替换 (https://www.jetbrains.com/help/idea/structural-search-and-replace.html) 模板, 将接口自动标记为 `external`:

* 用于 `State` 的模板 (https://gist.github.com/SebastianAigner/62119536f24597e630acfdbd14001b98)
* 用于 `Props` 的模板 (https://gist.github.com/SebastianAigner/a47a77f5e519fc74185c077ba12624f9)

将外部接口的属性转换为 var

问题: 在 Kotlin/JS 代码中, 外部接口的属性不能为只读(`val`)属性, 因为这些属性的赋值, 只能在使用 `js()` 或 `jso()` (来自 `kotlin-wrappers` (https://github.com/JetBrains/kotlin-wrappers) 的帮助函数) 创建对象之后:

```
import kotlinx.js.jso

val myState = jso<CustomComponentState>()
myState.name = "name"
```

解决方案: 将外部接口的所有属性转换为 `var`:

```
// 替换以下代码
external interface CustomComponentState : State {
    val name: String
}
```

```
// 替换为
external interface CustomComponentState : State {
    var name: String
}
```

将外部接口中带接受者的函数转换为普通函数

问题: 外部声明不能包含带接受者的函数, 比如扩展函数, 或这类函数类型的属性.

解决方案: 将这样的函数和属性转换为通常的函数, 将接受者对象添加为一个参数:

```
// 替换以下代码
external interface ButtonProps : Props {
    var inside: StyledDOMBuilder<BUTTON>.() -> Unit
}
```

```
external interface ButtonProps : Props {
    var inside: (StyledDOMBuilder<BUTTON>) -> Unit
}
```

为与 JS 交互, 创建单纯 JS 对象

问题: 实现外部接口的 Kotlin 对象的属性不可 *列举*. 因此不能通过对象属性的遍历得到这些属性, 比如:

- `for (var name in obj)`
- `console.log(obj)`
- `JSON.stringify(obj)`

但属性仍然可以通过名称访问: `obj.myProperty`

```
external interface AppProps { var name: String }
data class AppPropsImpl(override var name: String) : AppProps
fun main() {
    val jsApp = js("{name: 'App1'}") as AppProps // 单纯 JS 对象
    println("Kotlin sees: ${jsApp.name}") // 结果为: "App1"
    println("JSON.stringify sees:" + JSON.stringify(jsApp)) // 结果
为: {"name":"App1"} - OK

    val ktApp = AppPropsImpl("App2") // Kotlin 对象
    println("Kotlin sees: ${ktApp.name}") // "App2"
```

```
    // JSON 只能得到后端域, 不能得到属性
    println("JSON.stringify sees:" + JSON.stringify(ktApp)) //
{"_name_3":"App2"}
}
```

解决方案 1: 使用 js() 或 jso() (来自 kotlin-wrappers (https://github.com/JetBrains/kotlin-wrappers) 的帮助函数) 创建单纯 JavaScript 对象:

```
external interface AppProps { var name: String }
data class AppPropsImpl(override var name: String) : AppProps
```

```
// 替换以下代码
val ktApp = AppPropsImpl("App1") // Kotlin 对象
```

```
// 替换为
val jsApp = js("{name: 'App1'}") as AppProps // 或使用 jso {} 函数
```

解决方案 2: 使用 kotlin.js.json() 创建对象:

```
// 或者替换为
val jsonApp = kotlin.js.json(Pair("name", "App1")) as AppProps
```

将函数引用上的 toString() 调用替换为 .name

问题: 在 IR 后端中, 对函数引用调用 toString()不会输出唯一的值.

解决方案: 使用 name 属性代替 toString() 调用.

在构建脚本中明确指定 binaries.executable()

问题: 编译器不会产生可执行的 .js 文件.

可能会发生这样的问题, 因为默认的编译器会默认产生 JavaScript 可执行文件, 但 IR 编译器则需要明确指定. 详情请参见 Kotlin/JS 项目设置指南 ("执行环境" in "创建 Kotlin/JS 工程(Project)").

解决方案: 在项目的 build.gradle(.kts) 文件中添加 binaries.executable().

```
kotlin {
    js(IR) {
```

```
        browser {
        }
        binaries.executable()
    }
}
```

关于使用 Kotlin/JS IR 编译器时其他问题的提示

这些提示也许能够帮助你解决在使用 Kotlin/JS IR 编译器的项目中遇到的问题.

将外部接口中的 boolean 属性标记为 nullable

问题: 当你对外部接口中的 `Boolean` 属性调用 `toString` 时, 你会得到 `Uncaught TypeError: Cannot read properties of undefined (reading 'toString')` 之类的错误. JavaScript 将 boolean 变量的 `null` 值或 `undefined` 值当作 `false` 处理. 如果你需要对可能为 `null` 或 `undefined` 的 `Boolean` 值调用 `toString` (比如, 如果你的代码被 JavaScript 代码调用, 而你无法控制这些 JavaScript 代码), 需要注意这个问题:

```
external interface SomeExternal {
    var visible: Boolean
}

fun main() {
    val empty: SomeExternal = js("{}")
    println(empty.visible.toString()) // Uncaught TypeError: Cannot
read properties of undefined (reading 'toString')
}
```

如果你想要在 Kotlin 中覆盖的函数内(比如, 一个 React `button`), 使用这样的属性, 将会发生 `ClassCastException` 异常:

```
button {
    attrs {
        autoFocus = props.visible // 这里会发生 ClassCastException 异
常
    }
}
```

解决方案: 你可以将你的外部接口的 `Boolean` 属性标记为 nullable (`Boolean?`):

```
// 替换以下代码
external interface SomeExternal {
    var visible: Boolean
}
```

```
// 替换为
external interface SomeExternal {
    var visible: Boolean?
}
```

浏览器与 DOM API

最终更新: 2024/09/10

Kotlin/JS 标准库允许你使用 `kotlinx.browser` 包访问浏览器专有的功能, 包括典型的顶级对象, 比如 `document` 和 `window`. 标准库对这些对象的功能尽可能提供了类型安全的封装. 对于无法支持的情况, 为了与不能正确映射到 Kotlin 类型系统的函数交互, 会使用 `dynamic` 类型.

与 DOM 交互

为了与文档对象模型(DOM) 交互, 你可以使用变量 `document`. 比如, 你可以通过这个对象设置网站的背景颜色:

```
document.bgColor = "FFAA12"
```

`document` 对象还为你提供了一种途径, 可以通过 ID, 名称, class 名, tag 名等等, 来取得特定的元素. 所有返回的元素都是 `Element?` 类型. 要访问它们的属性, 你需要将它们转换为正确的类型. 比如, 假定你有一个 HTML 页面, 其中有一个 email `<input>` 项:

```
<body>
    <input type="text" name="email" id="email"/>

    <script type="text/javascript" src="tutorial.js"></script>
</body>
```

注意, 你的脚本包含在 `body` tag 的底部. 这样可以保证在脚本加载之前 DOM 已经全部完全加载完成.

通过这样的设置, 你可以访问 DOM 元素. 要访问 `input` 项目的属性, 请调用 `getElementById`, 并将它转换为 `HTMLInputElement`. 然后就可以安全的访问属性了, 比如 `value`:

```
val email = document.getElementById("email") as HTMLInputElement
email.value = "hadi@jetbrains.com"
```

与访问这个 `input` 元素类似, 你也可以访问页面中的其他元素, 并转换为正确的类型.

关于如何使用简洁的方式创建和组织 DOM 中的元素, 请参见 类型安全的 HTML DSL (类型安全的 HTML DSL).

在 Kotlin 中使用 JavaScript 代码

最终更新: 2024/09/10

Kotlin 设计时最优先重视与 Java 平台交互的问题: Kotlin 代码可以将 Java 类当作 Kotlin 类来使用, Java 代码也可以将 Kotlin 类当作 Java 类来使用.

然而, JavaScript 是一种动态类型的语言, 也就是说它在编译时刻不做类型检查. 通过 动态类型 (动态类型), 在 Kotlin 中你可以自由地与 JavaScript 交互, 如果你希望完全发挥 Kotlin 类型系统的能力, 你可以为 JavaScript 库创建外部声明, Kotlin 编译器及相关工具能够正确处理这些外部声明.

内联 JavaScript

使用 js() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/js.html) 函数, 你可以将 JavaScript 代码内联到你的 Kotlin 代码中. 比如:

```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
```

由于 js 函数的参数会在编译时解析, 然后原样的("as-is")翻译为 JavaScript 代码, 因此参数必须是字符串常量. 所以, 以下代码是不正确的:

```
fun jsTypeOf(o: Any): String {
    return js(getTypeof() + " o") // 这里会出错
}
fun getTypeof() = "typeof"
```

注意, 调用 js() 返回的值是 dynamic (动态类型) 类型, 这个类型在编译时不保证任何类型安全性.

external 修饰符

你可以对某个声明使用 external 修饰符, 来告诉 Kotlin 它是由纯 JavaScript 编写的. 编译器看到这样的声明后, 它会假定对应的类, 函数, 或属性的实现, 会由外部提供 (由开发者提供, 或由 npm 依赖项 ("npm 依赖项目" in "创建 Kotlin/JS 工程(Project)")提供), 因此它不会为这个声明生成 JavaScript 代码. 由于同样的原因, external 声明不能带有 body 部. 比如:

```
external fun alert(message: Any?): Unit
```

```
external class Node {
    val firstChild: Node

    fun append(child: Node): Node

    fun removeChild(child: Node): Node

    // 等等
}

external val window: Window
```

注意, `external` 修饰符会被内嵌的声明继承下来, 因此, 在示例程序的 `Node` 类的内部, 在成员函数和属性之前没有添加 `external` 标记.

`external` 修饰符只允许用于包级声明. 对于非 `external` 的类, 不允许声明 `external` 的成员.

声明类的(静态)成员

在 JavaScript 中, 成员函数可以定义在 prototype 上, 也可以定义在类上:

```
function MyClass() { ... }
MyClass.sharedMember = function() { /* 实现代码 */ };
MyClass.prototype.ownMember = function() { /* 实现代码 */ };
```

在 Kotlin 中没有这样的语法. 但是, 在 Kotlin 中有 `同伴` (companion) ("同伴对象(Companion Object)" in "对象表达式,对象声明,以及同伴对象") 对象. Kotlin 以特殊的方式处理 `external` 类的同伴对象: 它不是期待一个对象, 而是假设同伴对象的成员在 JavaScript 中是定义在类上的成员函数. 上例中的 `MyClass`, 在 Kotlin 中可以写为:

```
external class MyClass {
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

声明可选的参数

如果一个 JavaScript 函数带有可选的参数, 那么编写外部声明时请使用 `definedExternally`. 这个设置会将参数默认值的生成委托给 JavaScript 函数自身:

```
external fun myFunWithOptionalArgs(
    x: Int,
    y: String = definedExternally,
    z: String = definedExternally
)
```

通过这样的外部声明, 你就可以使用一个必须参数和两个可选参数来调用 `myFunWithOptionalArgs` 函数, 其中, 可选参数的默认值将由 `myFunWithOptionalArgs` 函数的 JavaScript 实现负责计算.

扩展 JavaScript 类

你可以很容易地扩展 JavaScript 类, 就好像它们是 Kotlin 类一样. 你只需要定义一个 `external open` 类, 然后通过一个非 `external` 类来扩展它. 比如:

```
open external class Foo {
    open fun run()
    fun stop()
}

class Bar : Foo() {
    override fun run() {
        window.alert("Running!")
    }

    fun restart() {
        window.alert("Restarting")
    }
}
```

但存在以下限制:

- 如果 `external` 基类的函数已存在不同参数签名的重载版本, 那么你就不能在后代类中覆盖这个函数.
- 带默认参数的函数不能覆盖.

- `external` 类不能扩展非 `external` 类.

external 接口

JavaScript 没有接口的概念. 如果一个函数要求它的参数支持 `foo` 和 `bar` 两个方法, 你只需要传递一个确实带有这些方法的对象.

在严格检查类型的 Kotlin 语言中, 你可以使用接口来表达这种概念:

```
external interface HasFooAndBar {
    fun foo()

    fun bar()
}

external fun myFunction(p: HasFooAndBar)
```

`external` 接口的另一种典型的使用场景, 是用来描述配置信息对象. 比如:

```
external interface JQueryAjaxSettings {
    var async: Boolean

    var cache: Boolean

    var complete: (JQueryXHR, String) -> Unit

    // 等等
}

fun JQueryAjaxSettings(): JQueryAjaxSettings = js("{}")

external class JQuery {
    companion object {
        fun get(settings: JQueryAjaxSettings): JQueryXHR
    }
}

fun sendQuery() {
    JQuery.get(JQueryAjaxSettings().apply {
        complete = { (xhr, data) ->
```

```
                window.alert("Request complete")
            }
        })
}
```

`external` 接口存在一些限制:

- 它们不可以用在 `is` 检查语句的右侧.
- 它们不可以用作实体化的类型参数(reified type argument).
- 它们不可以用在类的字面值表达式中(比如 `I::class`).
- 使用 `as` 将对象转换为 `external` 接口, 永远会成功. 转换为 `external` 接口会产生编译期警告 "Unchecked cast to external interface". 要消除这个警告, 可以使用 `@Suppress("UNCHECKED_CAST_TO_EXTERNAL_INTERFACE")` 注解.

  IntelliJ IDEA 也能够自动生成 `@Suppress` 注解. 方法是在编辑器内点击灯泡图标, 或按下 Alt-Enter 快捷键, 打开 intentions 菜单, 然后点击代码检查信息 "Unchecked cast to external interface" 旁的小箭头. 在这里, 你可以选择对这个警告进行屏蔽的适用范围, 然后 IDE 会在你的源代码文件中添加对应的注解.

类型转换

"不安全的" 类型转换操作符 (""不安全的" 类型转换操作符" in "类型检查与类型转换") `as` 在转换失败时会抛出 `ClassCastException` 异常, 除此之外, Kotlin/JS 还提供了 `unsafeCast<T>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/unsafe-cast.html). 如果使用 `unsafeCast`, 在运行时 *完全不进行任何类型检查*. 比如, 对于下面两个方法:

```
fun usingUnsafeCast(s: Any) = s.unsafeCast<String>()
fun usingAsOperator(s: Any) = s as String
```

对应的编译结果分别是:

```
function usingUnsafeCast(s) {
    return s;
}

function usingAsOperator(s) {
    var tmp$;
```

```
    return typeof (tmp$ = s) === 'string' ? tmp$ : throwCCE();
}
```

相等判断

与其他平台相比, Kotlin/JS 的相等判断语义有所不同.

在 Kotlin/JS 中, Kotlin 引用相等 ("引用相等" in "相等判断") 操作符 (`===`) 永远会翻译为 JavaScript 的 严格相等 (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_equality) 操作符 (`===`).

JavaScript `===` 操作符不仅检查两个值相等, 而且检查这两个值的类型也相等:

```
fun main() {
    val name = "kotlin"
    val value1 = name.substring(0, 1)
    val value2 = name.substring(0, 1)

    println(if (value1 === value2) "yes" else "no")
    // 在 Kotlin/JS 平台, 输出结果为 'yes'
    // 在其他平台, 输出结果为 'no'
}
```

而且, 在 Kotlin/JS 中, 数值类型 `Byte`, `Short`, `Int`, `Float`, 和 `Double` ("JavaScript 中的 Kotlin 类型" in "在 JavaScript 中使用 Kotlin 代码") 在运行期都使用 JavaScript 类型 `Number` 表达. 因此, 这 5 种类型的值是无法区分的:

```
fun main() {
    println(1.0 as Any === 1 as Any)
    // 在 Kotlin/JS 平台, 输出结果为 'true'
    // 在其他平台, 输出结果为 'false'
}
```

> ⚠ 关于 Kotlin 中的相等判断, 更多详情请参见 相等判断 (相等判断) 文档.

动态类型

最终更新: 2024/09/10

> 当编译目标平台为 JVM 时, 不支持动态类型.

Kotlin 是一种静态类型的语言, 但它必须与无类型的或类型系统较为松散的语言交互, 比如 JavaScript 的环境. 为了方便这样的使用场景, Kotlin 语言 提供了 `dynamic` 类型:

```
val dyn: dynamic = ...
```

简单来说, `dynamic` 类型关闭了 Kotlin 的类型检查:

- 一个 `dynamic` 类型的值可以赋值给任何变量, 也可以作为参数传递给任何函数.
- 任何值都可以赋值给一个 `dynamic` 类型的变量, 或传递给函数的 `dynamic` 类型参数.
- 对 `dynamic` 类型的值不做 `null` 检查.

`dynamic` 最受欢迎的功能是, 对 `dynamic` 类型变量, 可以访问它的 **任何** 属性, 还可以使用任意参数访问它的任何函数:

```
dyn.whatever(1, "foo", dyn) // 没有在任何地方定义过 'whatever'
dyn.whatever(*arrayOf(1, 2, 3))
```

在 JavaScript 平台, 这些代码会被"原封不动"地编译: Kotlin 代码中的 `dyn.whatever(1)`, 编译产生的 JavaScript 代码就是同样的 `dyn.whatever(1)`.

对 `dynamic` 类型的值调用 Kotlin 编写的函数时, 要注意, Kotlin 到 JavaScript 编译器会进行名称混淆. 你可能需要使用 @JsName 注解 ("@JsName 注解" in "在 JavaScript 中使用 Kotlin 代码") 来为你需要调用的函数指定一个明确的名称.

一个动态调用永远会返回一个 `dynamic` 的结果, 因此你可以将这些调用自由地串联起来:

```
dyn.foo().bar.baz()
```

当你向一个动态调用传递一个 Lambda 表达式作为参数时, Lambda 表达式的所有参数类型默认都是 `dynamic`:

```
dyn.foo {
    x -> x.bar() // x 是 dynamic 类型
}
```

使用 dynamic 类型值的表达式, 会被"原封不动"地翻译为 JavaScript, 请注意不要使用 Kotlin 的运算符规约. 以下运算符是支持的:

- 二元运算符: +, -, *, /, %, >, < >=, <=, ==, !=, ===, !==, &&, ||
- 一元运算符
  - 前缀运算符: -, +, !
  - 可以用作前缀运算符, 也可以用作后缀运算符: ++, --
- 计算并赋值运算符: +=, -=, *=, /=, %=
- 下标访问运算符:
  - 读操作: d[a], 参数多于一个会报错
  - 写操作: d[a1] = a2, [] 内的参数多于一个会报错

对 dynamic 类型的值使用 in, !in 和 .. 操作是禁止的.

关于更加深入的技术性介绍, 请参见 规格文档 (https://github.com/JetBrains/kotlin/blob/master/spec-docs/dynamic-types.md).

使用 npm 中的依赖项

最终更新: 2024/09/10

在 Kotlin/JS 项目中, 所有的依赖项都可以通过 Gradle plugin 来管理. 包括 Kotlin/Multiplatform 库, 比如kotlinx.coroutines, kotlinx.serialization, 或 ktor-client.

对于 来自 npm (https://www.npmjs.com/) 的 JavaScript 包依赖项, Gradle DSL 公开了 npm 函数, 你可以用来指定希望从 npm 导入的包. 我们来看看导入 NPM 包 is-sorted (https://www.npmjs.com/package/is-sorted) 的情况.

Gradle 构建脚本文件中对应的部分如下:

```
dependencies {
    // ...
    implementation(npm("is-sorted", "1.0.5"))
}
```

由于 JavaScript 模块通常使用动态类型, 而 Kotlin 是静态类型语言, 因此你需要提供某种类型的转换. 在 Kotlin 中, 这种转换称为 *外部声明(External Declaration)*. 对于 is-sorted 包, 它只提供了一个函数, 它的外部声明很小, 很容易编写. 请在源代码文件夹中, 创建一个新文件, 名为 is-sorted.kt, 内容如下:

```
@JsModule("is-sorted")
@JsNonModule
external fun <T> sorted(a: Array<T>): Boolean
```

请注意, 如果你使用 CommonJS 作为编译对象, 那么 @JsModule 和 @JsNonModule 注解也需要做相应的调整.

这个 JavaScript 函数现在可以象通常的 Kotlin 函数一样使用了. 由于我们在头文件中提供了类型信息 (而不是简单的将参数和返回值类型定义为 dynamic), 因此也可以进行正确的编译器支持和类型检查.

```
console.log("Hello, Kotlin/JS!")
console.log(sorted(arrayOf(1,2,3)))
console.log(sorted(arrayOf(3,1,2)))
```

在浏览器内或在 Node.js 中运行这 3 行代码, 输出显示, 对 `sorted` 的调用被正确的映射为 `is-sorted` 包导出的函数:

```
Hello, Kotlin/JS!
true
false
```

由于 JavaScript 生态系统有很多种方式来导出包中的函数 (比如通过命名的导出, 或默认导出), 因此对于其他 npm 包, 它的外部声明可能需要稍微不同的结构.

关于如何编写外部声明, 请参见 在 Kotlin 中使用 JavaScript 代码 ([在 Kotlin 中使用 JavaScript 代码](#)).

在 JavaScript 中使用 Kotlin 代码

最终更新: 2024/09/10

根据选择的 JavaScript 模块 (JavaScript 模块) 系统不同, Kotlin/JS 编译期会产生不同的输出. 但通常 Kotlin 编译器会生成通常的 JavaScript 类, 函数, 和属性, 你可以在 JavaScript 代码中自由地使用它们. 但是, 有一些细节问题, 你需要记住.

将声明隔离在 plain 模式下的独立 JavaScript 对象内

如果你将模块类型明确设置为 `plain`, Kotlin 会创建一个对象, 其中包含来自当前模块的所有 Kotlin 声明, 以免破坏全局对象. 因此, 对于模块 `myModule`, 在 JavaScript 中可以通过 `myModule` 对象访问到所有的声明. 比如:

```
fun foo() = "Hello"
```

在 JavaScript 中可以这样调用:

```
alert(myModule.foo());
```

如果你将你的 Kotlin 模块编译为 JavaScript 模块, 比如 UMD, CommonJS 或 AMD (对编译目标 `browser` 和 `nodejs`, 默认设定是 UMD), 就会出现兼容问题. 这种情况下, 你的声明对外公开时使用的格式将由你选择的 JavaScript 模块系统决定. 比如, 如果使用 UMD 或 CommonJS, 那么需要这样来使用你的代码:

```
alert(require('myModule').foo());
```

关于 JavaScript 模块系统, 更多详情请参见 JavaScript 模块(Module) (JavaScript 模块).

包结构

Kotlin 会将它的包结构公开到 JavaScript 中, 因此, 除非你将你的声明定义在最顶层包中, 否则在 JavaScript 中就必须使用完整限定名来访问你的声明. 比如:

```
package my.qualified.packagename

fun foo() = "Hello"
```

比如, 如果使用 UMD 或 CommonJS, 那么调用端应该如下:

```
alert(require('myModule').my.qualified.packagename.foo())
```

如果模块系统使用 `plain` 模式, 那么应该是:

```
alert(myModule.my.qualified.packagename.foo());
```

@JsName 注解

某些情况下 (比如, 为了支持重载(overload)), Kotlin 编译器会对 JavaScript 代码中生成的函数和属性的名称进行混淆. 为了控制编译器生成的函数和属性名称, 你可以使用 `@JsName` 注解:

```
// 'kjs' 模块
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
```

然后, 你可以在 JavaScript 中通过以下方式来使用这个类:

```
// 如果需要, 请根据你选择的模块系统, import 对应的 'kjs' 模块
var person = new kjs.Person("Dmitry");   // 参照到 'kjs' 模块
person.hello();                          // 打印结果为 "Hello Dmitry!"
person.helloWithGreeting("Servus");      // 打印结果为 "Servus
Dmitry!"
```

如果我们不指定 `@JsName` 注解, 那么编译器将会根据函数签名计算得到一个后缀字符串, 添加到生成的函数名末尾, 比如 `hello_61zpoe$`.

注意, 有些情况下 Kotlin 编译器不会进行这样的名称混淆:

- 对 `external` 声明, 不会进行名称混淆.

- 从 `external` 类继承的非 `external` 类之内, 被覆盖的函数, 不会进行名称混淆.

`@JsName` 注解的参数要求是字面值的字符串常量, 而且必须是一个有效的标识符. 如果将非标识符字符串用于 `@JsName` 注解, 编译器会报告错误. 下面的示例会常数一个编译期错误:

```
@JsName("new C()")   // 此处发生错误
external fun newC()
```

@JsExport 注解

> `@JsExport` 注解目前还在实验性阶段. 在未来的发布版本中, 它的设计可能会发生变化.

对一个顶级声明 (比如一个类或函数)使用 `@JsExport` 注解, 就可以在 JavaScript 中访问 Kotlin 声明. 这个注解会使用 Kotlin 中给定的名称, 导出所有的下层声明. 使用 `@file:JsExport` 的方式, 还可以将这个注解应用于整个源代码文件.

为了在导出声明时解决名称的歧义(比如同名函数的重载(overload)), 可以将 `@JsExport` 注解和 `@JsName` 注解一起使用, 用来指定生成和导出的函数名称.

在当前默认的编译器后端, 以及新的 IR 编译器后端 (使用 IR 编译器)中, 都可以使用 `@JsExport` 注解. 如果你使用 IR 编译器后端, 那么 **必须** 从一开始就使用 `@JsExport` 注解, 来确保你的函数在 Kotlin 中可以使用.

对于跨平台项目, 在共通代码中也可以使用 `@JsExport`. 它只在针对 JavaScript 目标进行编译时才起作用, 并且允许你导出那些平台无关的 Kotlin 声明.

JavaScript 中的 Kotlin 类型

Kotlin 类型在 JavaScript 中映射为以下类型:

Kotlin 类型	JavaScript 类型	注释
Byte, Short, Int, Float, Double	Number	
Char	Number	Number 表示字符的编码.
Long	不支持	JavaScript 中没有 64 位整数类型, 因此它使用一个 Kotlin 类来模拟.
Boolean	Boolean	
String	String	
Array	Array	
ByteArray	Int8Array	
ShortArray	Int16Array	
IntArray	Int32Array	
CharArray	UInt16Array	包含属性 $type$ == "CharArray".
FloatArray	Float32Array	
DoubleArray	Float64Array	
LongArray	Array<kotlin.Long>	包含属性 $type$ == "LongArray". 另外请参见 Kotlin 的 Long 类型的注释.
BooleanArray	Int8Array	包含属性 $type$ == "BooleanArray".

Unit	Undefined	
Any	Object	
Throwable	Error	
可为 Null 的 Type?	Type | null | undefined	
Kotlin 的所有其他类型 (使用 Js Export 注解标注的类型除外)	不支持	包含 Kotlin 的集合(List, Set, Map, 等等.), 以及无符号类型(unsigned variant).

此外, 还要注意:

- Kotlin 为 kotlin.Int, kotlin.Byte, kotlin.Short, kotlin.Char 和 kotlin.Long 保留了溢出语义.
- Kotlin 在运行时无法区分数值类型(除 kotlin.Long 外), 因此以下代码能够正常工作:

```
fun f() {
    val x: Int = 23
    val y: Any = x
    println(y as Float)
}
```

- Kotlin 在 JavaScript 中保留了延迟加载对象的初始化处理.
- Kotlin 在 JavaScript 中没有实现顶级属性的延迟加载初始化处理.

JavaScript 模块

最终更新: 2024/09/10

你可以将你的 Kotlin 工程编译为 JavaScript 模块(module), 支持几种常见的 JavaScript 模块系统. 目前我们支持以下几种 JavaScript 模块设置:

- 统一模块定义(Unified Module Definitions (UMD)) (https://github.com/umdjs/umd), 这种方式同时兼容于 *AMD* 和 *CommonJS*, 没有导入, 或者不存在模块系统时, UMD 模块也可以执行. 对于 `browser` 和 `nodejs` 编译目标, UMD 是默认选项 .
- 异步模块定义 (Asynchronous Module Definition (AMD)) (https://github.com/amdjs/amdjs-api/wiki/AMD), RequireJS (https://requirejs.org/) 库使用的就是这个模块系统.
- CommonJS (http://wiki.commonjs.org/wiki/Modules/1.1), 广泛使用于 Node.js/npm (`require` 函数和 `module.exports` 对象).
- Plain 方式. 不针对任何模块系统进行编译. 你仍然可以在全局命名空间内通过模块的名称来访问这个模块.

针对浏览器平台

如果你期望在 Web 浏览器环境中运行你的代码, 并且希望使用 UMD 之外的模块系统, 那么可以在 `webpackTask` 配置代码段内指定希望的模块类型:

```
kotlin {
    js {
        browser {
            webpackTask {
                output.libraryTarget = "commonjs2"
            }
        }
        binaries.executable()
    }
}
```

Webpack 提供了 CommonJS 的两种不同的风格: `commonjs` 和 `commonjs2`, 这个设置会影响你的声明导出的方式. 大多数情况下, 你可能希望使用 `commonjs2`, 它会在生成的 JavaScript 库中添加 `module.exports` 语法. 或者你也可以选择 `commonjs` 选项, 它严格遵照 CommonJS 标准. 关于 `commonjs` 和 `commonjs2` 的区别, 更多详情请参见 Webpack 代码仓库 (https://github.com/webpack/webpack/issues/1114).

JavaScript 库文件和 Node.js 文件

如果你在创建供 JavaScript 或 Node.js 环境使用的库, 并且希望使用不同的模块系统, 那么编译指令略有不同.

选择编译目标的模块系统

要选择目标模块系统, 请在 Gradle 构建脚本中设置编译器选项 `moduleKind`:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.targets.js.ir.KotlinJ
sIrLink> {

compilerOptions.moduleKind.set(org.jetbrains.kotlin.gradle.dsl.J
sModuleKind.MODULE_COMMONJS)
}
```

Groovy

```
compileKotlinJs.compilerOptions.moduleKind =
org.jetbrains.kotlin.gradle.dsl.JsModuleKind.MODULE_COMMONJS
```

这里可以设置的值是: `umd` (默认设定), `commonjs`, `amd`, `plain`.

> 这种方法与修改 `webpackTask.output.libraryTarget` 不同. 库的输出目标设定, 改变的是 *webpack 库文件生成* 的输出(在你的代码经过 Kotlin 编译之后). `compilerOptions.moduleKind` 改变的则是 *Kotlin 编译器* 的输出.

在 Kotlin Gradle DSL 中, 设置 CommonJS 模块类型可以简写为:

```
kotlin {
    js {
        useCommonJs()
        // ...
    }
}
```

@JsModule 注解

你可以使用 `@JsModule` 注解, 告诉 Kotlin 一个 `external` 类, 包, 函数, 或属性, 是一个 JavaScript 模块. 假设你有以下 CommonJS 模块, 名为 "hello":

```
module.exports.sayHello = function (name) { alert("Hello, " + name); }
```

在 Kotlin 中你应该这样声明:

```
@JsModule("hello")
external fun sayHello(name: String)
```

对包使用 @JsModule 注解

某些 JavaScript 库会向外导出包 (名称空间), 而不是导出函数和类. 用 JavaScript 的术语来讲, 它是一个 *对象*, 这个对象的 *成员* 是类, 函数, 以及属性. 将这些包作为 Kotlin 对象导入, 通常很不自然. 编译器可以将导入的 JavaScript 包映射为 Kotlin 包, 语法如下:

```
@file:JsModule("extModule")

package ext.jspackage.name

external fun foo()

external class C
```

对应的 JavaScript 模块声明如下:

```
module.exports = {
    foo: { /* 某些实现代码 */ },
```

```
    C: { /* 某些实现代码 */ }
}
```

使用 `@file:JsModule` 注解标注的源代码文件中, 不能声明非 external 的成员. 下面的示例会发生编译期错误:

```
@file:JsModule("extModule")

package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // 此处发生错误
```

导入更深的包层次结构

在前面的示例中, JavaScript 模块导出了一个单独的包. 但是, 某些 JavaScript 库会从一个模块中导出多个包. Kotlin 也支持这样的情况, 但是你必须为导入的每一个包声明一个新的 `.kt` 文件.

比如, 把示例修改得稍微复杂一点:

```
module.exports = {
    mylib: {
        pkg1: {
            foo: function () { /* 某些实现代码 */ },
            bar: function () { /* 某些实现代码 */ }
        },
        pkg2: {
            baz: function () { /* 某些实现代码 */ }
        }
    }
}
```

要在 Kotlin 中导入这个模块, 你必须编写两个 Kotlin 源代码文件:

```
@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg1")

package extlib.pkg1
```

```
external fun foo()

external fun bar()
```

以及

```
@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg2")

package extlib.pkg2

external fun baz()
```

@JsNonModule 注解

假如一个声明标注了 `@JsModule` 注解, 如果你的代码不编译为 JavaScript 模块, 你就不能在 Kotlin 代码中使用它. 通常, 开发者发布他们的库时, 会同时使用 JavaScript 模块形式, 以及可下载的 `.js` 文件形式 (使用者可以复制到项目的静态资源中, 然后通过 `<script>` tag 来引用). 为了告诉 Kotlin, 一个标注了 `@JsModule` 注解的声明可以在非 JavaScript 模块的环境中使用, 你应该加上 `@JsNonModule` 声明. 比如, 对于下面的 JavaScript 代码:

```
function topLevelSayHello (name) { alert("Hello, " + name); }

if (module && module.exports) {
    module.exports = topLevelSayHello;
}
```

在 Kotlin 中可以这样声明:

```
@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)
```

Kotlin 标准库使用的模块系统

Kotlin 将 Kotlin/JS 标准库作为一个单个的文件发布, 这个库本身编译为一个 UMD 模块, 因此你可以在上面讲到的任何一种模块系统中使用这个库. 对于 Kotlin/JS 的大多数使用场景, 推荐通过

Gradle 依赖项 `kotlin-stdlib-js` 来使用它, 在 NPM 中也可以通过 `kotlin` (https://www.npmjs.com/package/kotlin) 包来使用.

Kotlin/JS 的反射(Reflection)

最终更新: 2024/09/10

Kotlin/JS 只对 Kotlin 反射 API (反射) 提供有限的支持. 目前仅支持以下 API:

- 类引用(class reference) ("类引用(Class Reference)" in "反射") (`::class`).
- `KType` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-type/) 和 `typeof()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/type-of.html) 函数.

类引用(class reference)

`::class` 语法返回一个对象实例的类引用, 一个指定的类型的类引用. 在 Kotlin/JS 中, `::class` 表达式的值是 KClass (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-class/) 的一个简化版实现, 它只支持:

- 成员函数 simpleName (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-class/simple-name.html) 和 isInstance() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-class/is-instance.html).
- 扩展函数 cast() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/cast.html) 和 safeCast() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/safe-cast.html).

除此之外, 你还可以使用 KClass.js (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/js.html) 来获取某个类的 JsClass (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/-js-class/index.html) 实例. `JsClass` 的实例本身是一个指向构造函数的引用. 因此可以用来与那些需要用到构造函数引用的 JS 函数互操作.

KType 与 typeOf()

`typeof()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/type-of.html) 函数 对一个指定的类型 创建 `KType` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-type/) 实例. Kotlin/JS 完全支持 `KType` API, 但 Java 专有的部分除外.

示例

下面是 Kotlin/JS 中反射的使用示例.

```
open class Shape
class Rectangle : Shape()

inline fun <reified T> accessReifiedTypeArg() =
    println(typeOf<T>().toString())

fun main() {
    val s = Shape()
    val r = Rectangle()

    println(r::class.simpleName) // 输出结果为 "Rectangle"
    println(Shape::class.simpleName) // 输出结果为 "Shape"
    println(Shape::class.js.name) // 输出结果为 "Shape"

    println(Shape::class.isInstance(r)) // 输出结果为 "true"
    println(Rectangle::class.isInstance(s)) // 输出结果为 "false"
    val rShape = Shape::class.cast(r) // 将 Rectangle "r" 转换为
Shape

    accessReifiedTypeArg<Rectangle>() // 通过 typeOf() 访问类型. 输出结
果为 "Rectangle"
}
```

类型安全的 HTML DSL

最终更新: 2024/09/10

kotlinx.html 库 (https://www.github.com/kotlin/kotlinx.html) 提供了使用静态类型的 HTML 构建器生成 DOM 元素的能力 (而且除 JavaScript 之外, 它甚至能在 JVM 平台使用!) 要使用这个库, 请在我们的 `build.gradle.kts` 文件中包含对应的仓库和依赖项:

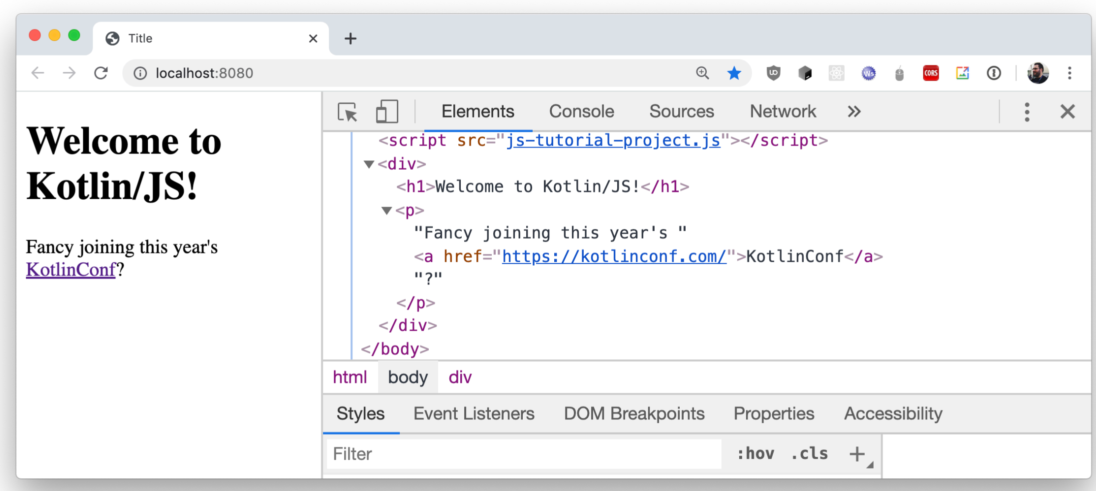
```
repositories {
    // ...
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-js"))
    implementation("org.jetbrains.kotlinx:kotlinx-html-js:0.8.0")
    // ...
}
```

依赖项添加完成后, 你可以访问其中提供的各自接口来生成 DOM. 要输出一个标题, 一些文字, 以及一个链接, 示例代码如下:

```
import kotlinx.browser.*
import kotlinx.html.*
import kotlinx.html.dom.*

fun main() {
    document.body!!.append.div {
        h1 {
            +"Welcome to Kotlin/JS!"
        }
        p {
            +"Fancy joining this year's "
            a("https://kotlinconf.com/") {
                +"KotlinConf"
            }
            +"?"
        }
```

```
    }
}
```

在浏览器中运行这个示例程序时, DOM 会被直接组装起来. 使用浏览器的开发工具查看网站的元素, 我们很容易确认结果:



使用 kotlinx.html 输出网页

关于 `kotlinx.html` 库, 更多详情请参见 GitHub Wiki 页面 (https://github.com/Kotlin/kotlinx.html/wiki/Getting-started), 在这里你可以找到更多信息, 关于如何 创建元素 (https://github.com/Kotlin/kotlinx.html/wiki/DOM-trees) 而不将其添加到 DOM, 如何 绑定事件 (https://github.com/Kotlin/kotlinx.html/wiki/Events), 比如 `onClick`, 以及 示例程序, 演示如何 添加 CSS 类 (https://github.com/Kotlin/kotlinx.html/wiki/Elements-CSS-classes) 到你的 HTML 元素, 以及其他更多信息.

教程 - 使用 React 和 Kotlin/JS 创建 Web 应用程序

最终更新: 2024/09/10

This tutorial will teach you how to build a browser application with Kotlin/JS and the React (https://reactjs.org/) framework. You will:

- Complete common tasks associated with building a typical React application.
- Explore how Kotlin's DSLs (类型安全的构建器) can be used to help express concepts concisely and uniformly without sacrificing readability, allowing you to write a full-fledged application completely in Kotlin.
- Learn how to use ready-made npm components, use external libraries, and publish the final application.

The output will be a *KotlinConf Explorer* web app dedicated to the KotlinConf (https://kotlinconf.com/) event, with links to conference talks. Users will be able to watch all the talks on one page and mark them as seen or unseen.

The tutorial assumes you have prior knowledge of Kotlin and basic knowledge of HTML and CSS. Understanding the basic concepts behind React may help you understand some sample code, but it is not strictly required.

You can get the final application as well as the intermediate steps here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle). Each step is available from its own branch and is linked at the bottom of its corresponding section.

Before you start

1. Download and install the latest version of IntelliJ IDEA (https://www.jetbrains.com/idea/download/index.html).

2. Clone the project template (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle) and open it in IntelliJ IDEA. The template includes a basic Kotlin/JS Gradle project with all required configurations and dependencies

- Dependencies and tasks in the `build.gradle.kts` file:

```
dependencies {
    // React, React DOM + Wrappers
    implementation(enforcedPlatform("org.jetbrains.kotlin-
wrappers:kotlin-wrappers-bom:1.0.0-pre.354"))
    implementation("org.jetbrains.kotlin-wrappers:kotlin-react")
    implementation("org.jetbrains.kotlin-wrappers:kotlin-react-
dom")

    // Kotlin React Emotion (CSS)
    implementation("org.jetbrains.kotlin-wrappers:kotlin-emotion")

    // Video Player
    implementation(npm("react-player", "2.10.1"))

    // Share Buttons
    implementation(npm("react-share", "4.4.0"))

    // Coroutines & serialization
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.6.3")
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-
json:1.3.3")
}
```

- An HTML template page in `src/main/resources/index.html` for inserting JavaScript code that you'll be using in this tutorial:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello, Kotlin/JS!</title>
```

```
</head>
<body>
    <div id="root"></div>
    <script src="confexplorer.js"></script>
</body>
</html>
```

Kotlin/JS projects are automatically bundled with all of your code and its dependencies into a single JavaScript file with the same name as the project, `confexplorer.js`, when you build them. As a typical JavaScript convention (https://faqs.skillcrush.com/article/176-where-should-js-script-tags-be-linked-in-html-documents), the content of the body (including the `root` div) is loaded first to ensure that the browser loads all page elements before the scripts.

- A code snippet in `src/main/kotlin/Main.kt`:

```
import kotlinx.browser.document

fun main() {
    document.bgColor = "red"
}
```

Run the development server

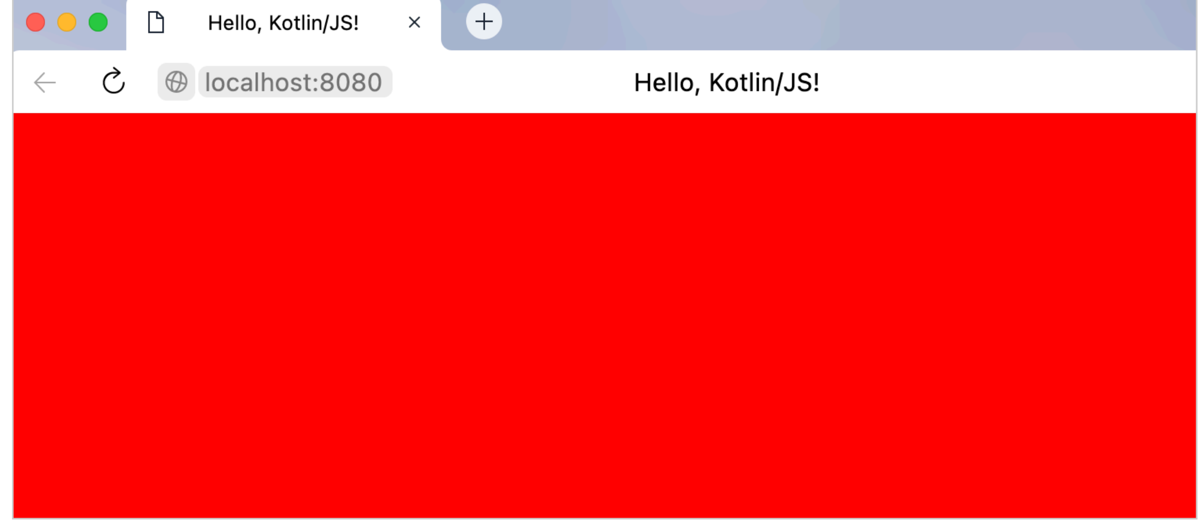
By default, the Kotlin/JS Gradle plugin comes with support for an embedded `webpack-dev-server`, allowing you to run the application from the IDE without manually setting up any servers.

To test that the program successfully runs in the browser, start the development server by invoking the `run` or `browserDevelopmentRun` task (available in the `other` or `kotlin browser` directory) from the Gradle tool window inside IntelliJ IDEA:

Gradle tasks list

To run the program from the Terminal, use `./gradlew run` instead.

When the project is compiled and bundled, a blank red page will appear in a browser window:

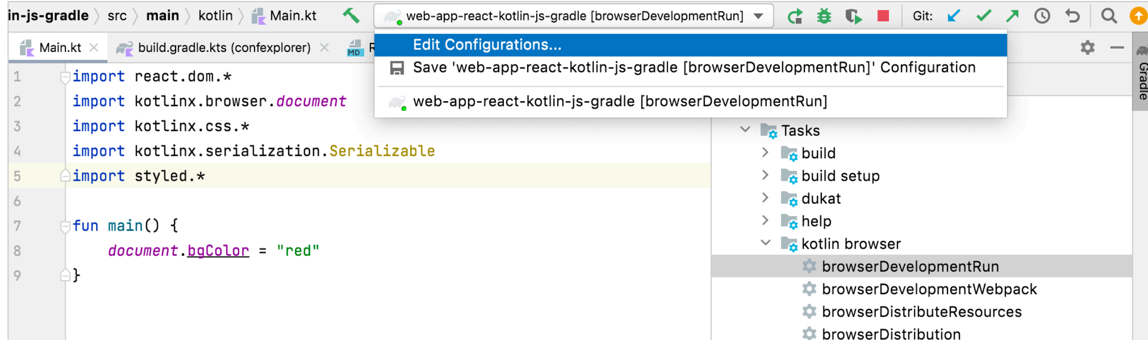


Blank red page

Enable hot reload / continuous mode

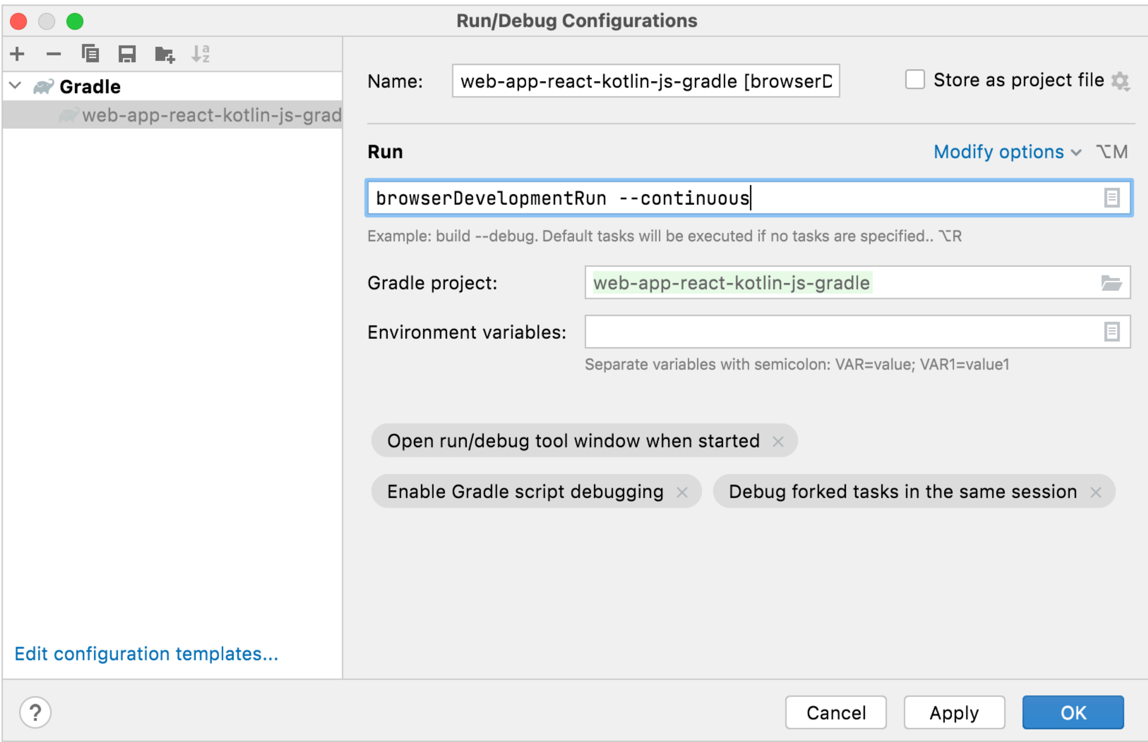
Configure *continuous compilation* ([*开发服务器(Development server)与持续编译(Continuous Compilation)*](#)) mode so you don't have to manually compile and execute your project every time you make changes. Make sure to stop all running development server instances before proceeding.

1. Edit the run configuration that IntelliJ IDEA automatically generates after running the Gradle `run` task for the first time:



Edit a run configuration

2. In the **Run/Debug Configurations** dialog, add the `--continuous` option to the arguments for the run configuration:



Enable continuous mode

After applying the changes, you can use the **Run** button inside IntelliJ IDEA to start the development server back up. To run the continuous Gradle builds from the Terminal, use `./gradlew run --continuous` instead.

3. To test this feature, change the color of the page to blue in the `Main.kt` file while the Gradle task is running:

```
document.bgColor = "blue"
```

The project then recompiles, and after a reload the browser page will be the new color.

You can keep the development server running in continuous mode during the development process. It will automatically rebuild and reload the page when you make changes.

You can find this state of the project on the `master` branch here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle/tree/master).

Create a web app draft

Add the first static page with React

To make your app display a simple message, replace the code in the `Main.kt` file with the following:

```
import kotlinx.browser.document
import react.*
import emotion.react.css
import csstype.Position
import csstype.px
import react.dom.html.ReactHTML.h1
import react.dom.html.ReactHTML.h3
import react.dom.html.ReactHTML.div
import react.dom.html.ReactHTML.p
import react.dom.html.ReactHTML.img
import react.dom.client.createRoot
import kotlinx.serialization.Serializable

fun main() {
    val container = document.getElementById("root") ?:
error("Couldn't find root container!")
    createRoot(container).render(Fragment.create {
        h1 {
            +"Hello, React+Kotlin/JS!"
        }
    })
}
```

- The render() function instructs kotlin-react-dom (https://github.com/JetBrains/kotlin-wrappers/tree/master/kotlin-react-dom) to render the first HTML element inside a fragment (https://reactjs.org/docs/fragments.html) to the root element. This element is a container defined in src/main/resources/index.html, which was included in the template.
- The content is an <h1> header and uses a typesafe DSL to render HTML.
- h1 is a function that takes a lambda parameter. When you add the + sign in front of a string literal, the unaryPlus() function is actually invoked using operator overloading (操作符重载). It appends the string to the enclosed HTML element.

When the project recompiles, the browser displays this HTML page:

Hello, React+Kotlin/JS!

An HTML page example

Convert HTML to Kotlin's typesafe HTML DSL

The Kotlin wrappers (https://github.com/JetBrains/kotlin-wrappers/blob/master/kotlin-react/README.md) for React come with a domain-specific language (DSL) (类型安全的构建器) that makes it possible to write HTML in pure Kotlin code. In this way, it's similar to JSX (https://reactjs.org/docs/introducing-jsx.html) from JavaScript. However, with this markup being Kotlin, you get all the benefits of a statically typed language, such as autocomplete or type checking.

Compare the classic HTML code for your future web app and its typesafe variant in Kotlin:

HTML

```
<h1>KotlinConf Explorer</h1>
<div>
    <h3>Videos to watch</h3>
    <p>John Doe: Building and breaking things</p>
    <p>Jane Smith: The development process</p>
    <p>Matt Miller: The Web 7.0</p>
    <h3>Videos watched</h3>
    <p>Tom Jerry: Mouseless development</p>
```

```
</div>
<div>
    <h3>John Doe: Building and breaking things</h3>
    <img src="https://via.placeholder.com/640x360.png?
text=Video+Player+Placeholder">
</div>
```

Kotlin

```
h1 {
    +"KotlinConf Explorer"
}
div {
    h3 {
        +"Videos to watch"
    }
    p {
        + "John Doe: Building and breaking things"
    }
    p {
        +"Jane Smith: The development process"
    }
    p {
        +"Matt Miller: The Web 7.0"
    }
    h3 {
        +"Videos watched"
    }
    p {
        +"Tom Jerry: Mouseless development"
    }
}
div {
    h3 {
        +"John Doe: Building and breaking things"
    }
    img {
```

```
        src = "https://via.placeholder.com/640x360.png?
text=Video+Player+Placeholder"
    }
}
```

Copy the Kotlin code and update the `Fragment.create()` function call inside the `main()` function, replacing the previous `h1` tag.

Wait for the browser to reload. The page should now look like this:

KotlinConf Explorer

Videos to watch

John Doe: Building and breaking things

Jane Smith: The development process

Matt Miller: The Web 7.0

Videos watched

Tom Jerry: Mouseless development

John Doe: Building and breaking things

Video Player Placeholder

The web app draft

Add videos using Kotlin constructs in markup

There are some advantages to writing HTML in Kotlin using this DSL. You can manipulate your app using regular Kotlin constructs, like loops, conditions, collections, and string interpolation.

You can now replace the hardcoded list of videos with a list of Kotlin objects:

1. In `Main.kt`, create a `Video` data class (数据类(Data Class)) to keep all video attributes in one place:

   ```
   data class Video(
       val id: Int,
       val title: String,
       val speaker: String,
       val videoUrl: String
   )
   ```

2. Fill up the two lists, for unwatched videos and watched videos, respectively. Add these declarations at file-level in `Main.kt`:

   ```
   val unwatchedVideos = listOf(
       Video(1, "Opening Keynote", "Andrey Breslav",
   "https://youtu.be/PsaFVLr8t4E"),
       Video(2, "Dissecting the stdlib", "Huyen Tue Dao",
   "https://youtu.be/Fzt_9I733Yg"),
       Video(3, "Kotlin and Spring Boot", "Nicolas Frankel",
   "https://youtu.be/pSiZVAeReeg")
   )

   val watchedVideos = listOf(
       Video(4, "Creating Internal DSLs in Kotlin", "Venkat
   Subramaniam", "https://youtu.be/JzTeAM8N1-o")
   )
   ```

3. To use these videos on the page, write a Kotlin `for` loop to iterate over the collection of unwatched `Video` objects. Replace the three `p` tags under "Videos to watch" with the following snippet:

```
for (video in unwatchedVideos) {
    p {
        +"${video.speaker}: ${video.title}"
    }
}
```

4. Apply the same process to modify the code for the single tag following "Videos watched" as well:

```
for (video in watchedVideos) {
    p {
        +"${video.speaker}: ${video.title}"
    }
}
```

Wait for the browser to reload. The layout should stay the same as before. You can add some more videos to the list to make sure that the loop is working.

Add styles with typesafe CSS

The kotlin-emotion (https://github.com/JetBrains/kotlin-wrappers/blob/master/kotlin-emotion/) wrapper for the Emotion (https://emotion.sh/docs/introduction) library makes it possible to specify CSS attributes – even dynamic ones – right alongside HTML with JavaScript. Conceptually, that makes it similar to CSS-in-JS (https://reactjs.org/docs/faq-styling.html#what-is-css-in-js) – but for Kotlin. The benefit of using a DSL is that you can use Kotlin code constructs to express formatting rules.

The template project for this tutorial already includes the dependency needed to use `kotlin-emotion`:

```
dependencies {
    // ...
    // Kotlin React Emotion (CSS) (chapter 3)
    implementation("org.jetbrains.kotlin-wrappers:kotlin-emotion")
    // ...
}
```

With `kotlin-emotion`, you can specify a `css` block inside HTML elements `div` and `h3`, where you can define the styles.

To move the video player to the top right-hand corner of the page, use CSS and adjust the code for the video player (the last `div` in the snippet):

```
div {
    css {
        position = Position.absolute
        top = 10.px
        right = 10.px
    }
    h3 {
        +"John Doe: Building and breaking things"
    }
    img {
        src = "https://via.placeholder.com/640x360.png?
text=Video+Player+Placeholder"
    }
}
```

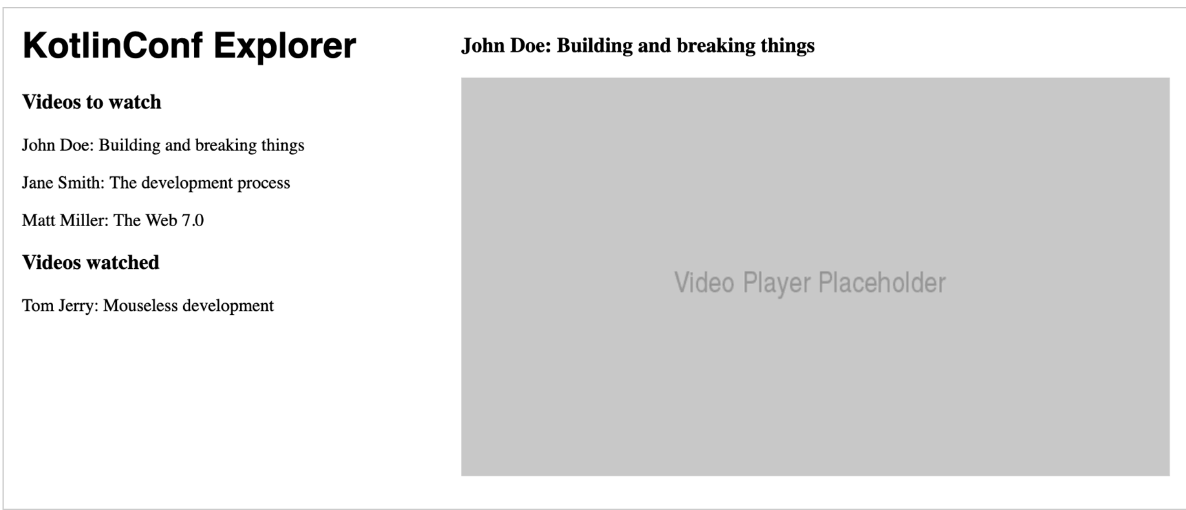
Feel free to experiment with some other styles. For example, you could change the `fontFamily` or add some `color` to your UI.

You can find this state of the project in the `02-first-static-page` branch here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle/tree/02-first-static-page).

Design app components

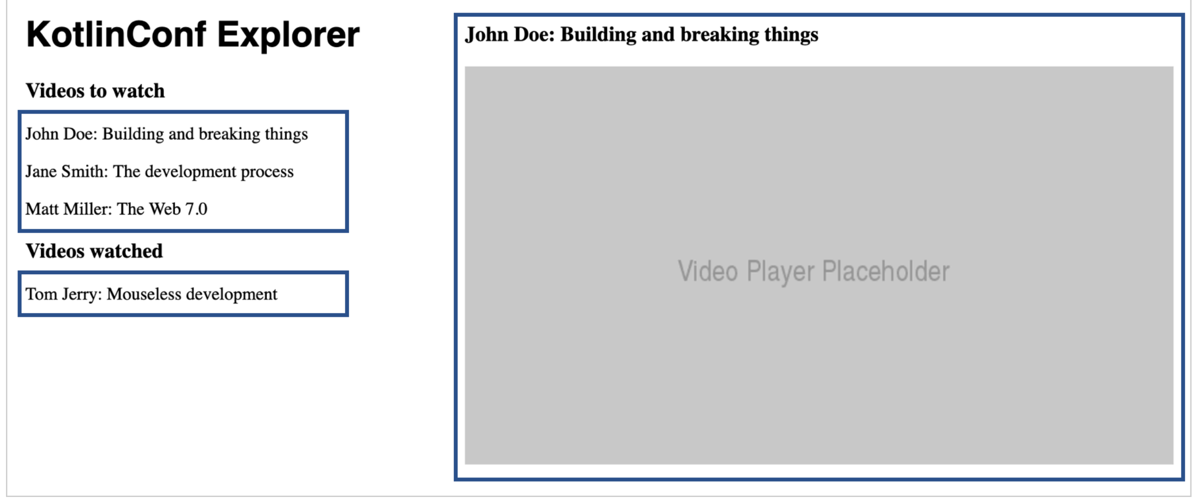
The basic building blocks in React are called *components (https://reactjs.org/docs/components-and-props.html)*. Components themselves can also be composed of other, smaller components. By combining components, you build your application. If you structure components to be generic and reusable, you'll be able to use them in multiple parts of the app without duplicating code or logic.

The content of the `render()` function generally describes a basic component. The current layout of your application looks like this:



Current layout

If you decompose your application into individual components, you'll end up with a more structured layout in which each component handles its responsibilities:



Structured layout with components

Components encapsulate a particular functionality. Using components shortens source code and makes it easier to read and understand.

Add the main component

To start creating the application's structure, first explicitly specify `App`, the main component for rendering to the `root`element:

1. Create a new `App.kt` file in the `src/main/kotlin` folder.

2. Inside this file, add the following snippet and move the typesafe HTML from `Main.kt` into it:

   ```
   import kotlinx.coroutines.async
   import react.*
   import react.dom.*
   import kotlinx.browser.window
   import kotlinx.coroutines.*
   import kotlinx.serialization.decodeFromString
   import kotlinx.serialization.json.Json
   import emotion.react.css
   import csstype.Position
   import csstype.px
   import react.dom.html.ReactHTML.h1
   import react.dom.html.ReactHTML.h3
   import react.dom.html.ReactHTML.div
   import react.dom.html.ReactHTML.p
   import react.dom.html.ReactHTML.img

   val App = FC<Props> {
       // typesafe HTML goes here, starting with the first h1 tag!
   }
   ```

   The `FC` function creates a function component (https://reactjs.org/docs/components-and-props.html#function-and-class-components).

3. In the `Main.kt` file, update the `main()` function as follows:

   ```
   fun main() {
       val container = document.getElementById("root") ?:
   error("Couldn't find root container!")
       createRoot(container).render(App.create())
   }
   ```

   Now the program creates an instance of the `App` component and renders it to the specified container.

For more information about React concepts, see the documentation and guides

(https://reactjs.org/docs/hello-world.html#how-to-read-this-guide).

Extract a list component

Since the `watchedVideos` and `unwatchedVideos` lists each contain a list of videos, it makes sense to create a single reusable component, and only adjust the content displayed in the lists.

The `VideoList` component follows the same pattern as the `App` component. It uses the `FC` builder function, and contains the code from the `unwatchedVideos` list.

1. Create a new `VideoList.kt` file in the `src/main/kotlin` folder and add the following code:

```
import kotlinx.browser.window
import react.*
import react.dom.*
import react.dom.html.ReactHTML.p

val VideoList = FC<Props> {
    for (video in unwatchedVideos) {
        p {
            +"${video.speaker}: ${video.title}"
        }
    }
}
```

2. In `App.kt`, use the `VideoList` component by invoking it without parameters:

```
// . . .
div {
    h3 {
        +"Videos to watch"
    }
    VideoList()

    h3 {
        +"Videos watched"
    }
    VideoList()
```

```
}
// . . .
```

For now, the App component has no control over the content that is shown by the VideoList component. It's hard-coded, so you see the same list twice.

Add props to pass data between components

Since you're going to reuse the VideoList component, you'll need to be able to fill it with different content. You can add the ability to pass the list of items as an attribute to the component. In React, these attributes are called *props*. When the props of a component are changed in React, the framework automatically re-renders the component.

For VideoList, you'll need a prop containing the list of videos to be shown. Define an interface that holds all the props which can be passed to a VideoList component:

1. Add the following definition to the VideoList.kt file:

```
external interface VideoListProps : Props {
    var videos: List<Video>
}
```

   The external ("external 修饰符" in "在 Kotlin 中使用 JavaScript 代码") modifier tells the compiler that the interface's implementation is provided externally, so it doesn't try to generate JavaScript code from the declaration.

2. Adjust the class definition of VideoList to make use of the props that are passed into the FC block as a parameter:

```
val VideoList = FC<VideoListProps> { props ->
    for (video in props.videos) {
        p {
            key = video.id.toString()
            +"${video.speaker}: ${video.title}"
        }
    }
}
```

   The key attribute helps the React renderer figure out what to do when the value of props.videos changes. It uses the key to determine which parts of a list need to be

refreshed and which ones stay the same. You can find more information about lists and keys in the React guide (https://reactjs.org/docs/lists-and-keys.html).

3. In the App component, make sure that the child components are instantiated with the proper attributes. In App.kt, replace the two loops underneath the h3 elements with an invocation of VideoList together with the attributes for unwatchedVideos and watchedVideos. In the Kotlin DSL, you assign them inside a block belonging to the VideoList component:

```
h3 {
    +"Videos to watch"
}
VideoList {
    videos = unwatchedVideos
}
h3 {
    +"Videos watched"
}
VideoList {
    videos = watchedVideos
}
```

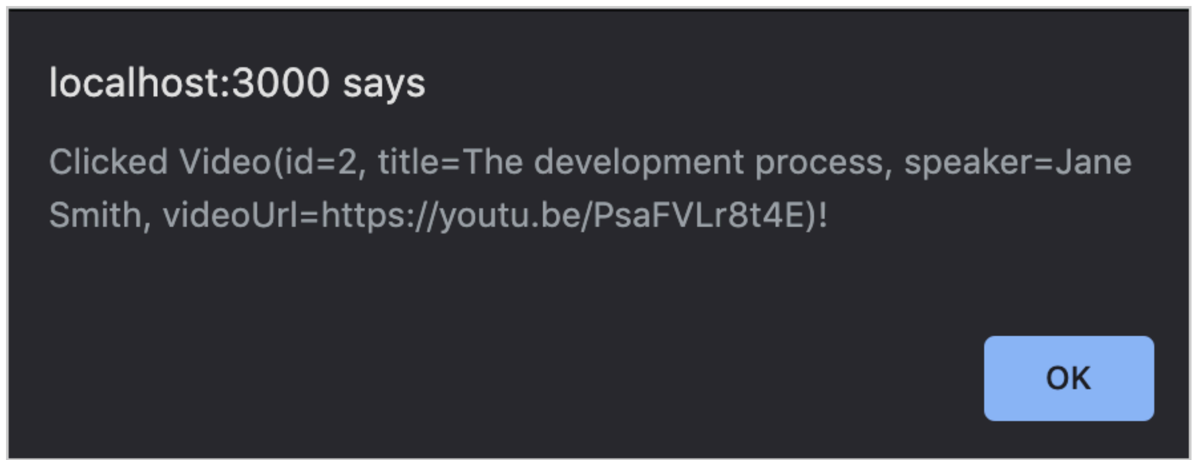
After a reload, the browser will show that the lists now render correctly.

Make the list interactive

First, add an alert message that pops up when users click on a list entry. In VideoList.kt, add an onClick handler function that triggers an alert with the current video:

```
// . . .
p {
    key = video.id.toString()
    onClick = {
        window.alert("Clicked $video!")
    }
    +"${video.speaker}: ${video.title}"
}
// . . .
```

If you click on one of the list items in the browser window, you'll get information about the video in an alert window like this:



Browser alert window

> ⚠ Defining an onClick function directly as lambda is concise and very useful for prototyping. However, due to the way equality currently works (https://youtrack.jetbrains.com/issue/KT-15101) in Kotlin/JS, performance-wise it's not the most optimized way to pass click handlers. If you want to optimize rendering performance, consider storing your functions in a variable and passing them.

Add state to keep values

Instead of just alerting the user, you can add some functionality for highlighting the selected video with a ▶ triangle. To do that, introduce some *state* specific to this component.

State is one of core concepts in React. In modern React (which uses the so-called *Hooks API*), state is expressed using the `useState` hook (https://reactjs.org/docs/hooks-state.html).

1. Add the following code to the top of the `VideoList` declaration:

```
val VideoList = FC<VideoListProps> { props ->
    var selectedVideo: Video? by useState(null)
// . . .
```

- The `VideoList` functional component keeps state (a value that is independent of the current function invocation). State is nullable, and has the `Video?` type. Its default value is `null`.
- The `useState()` function from React instructs the framework to keep track of state across multiple invocations of the function. For example, even though you specify a default value, React makes sure that the default value is only assigned in the beginning. When state changes, the component will re-render based on the new state.
- The `by` keyword indicates that `useState()` acts as a delegated property (委托属性). Like with any other variable, you read and write values. The implementation behind `useState()` takes care of the machinery required to make state work.

To learn more about the State Hook, check out the React documentation (https://reactjs.org/docs/hooks-state.html).

2. Change your implementation of the `VideoList` component to look as follows:

```
val VideoList = FC<VideoListProps> { props ->
    var selectedVideo: Video? by useState(null)
    for (video in props.videos) {
        p {
            key = video.id.toString()
            onClick = {
                selectedVideo = video
            }
            if (video == selectedVideo) {
                +"▶ "
            }
            +"${video.speaker}: ${video.title}"
        }
    }
}
```

- When the user clicks a video, its value is assigned to the `selectedVideo` variable.
- When the selected list entry is rendered, the triangle is prepended.

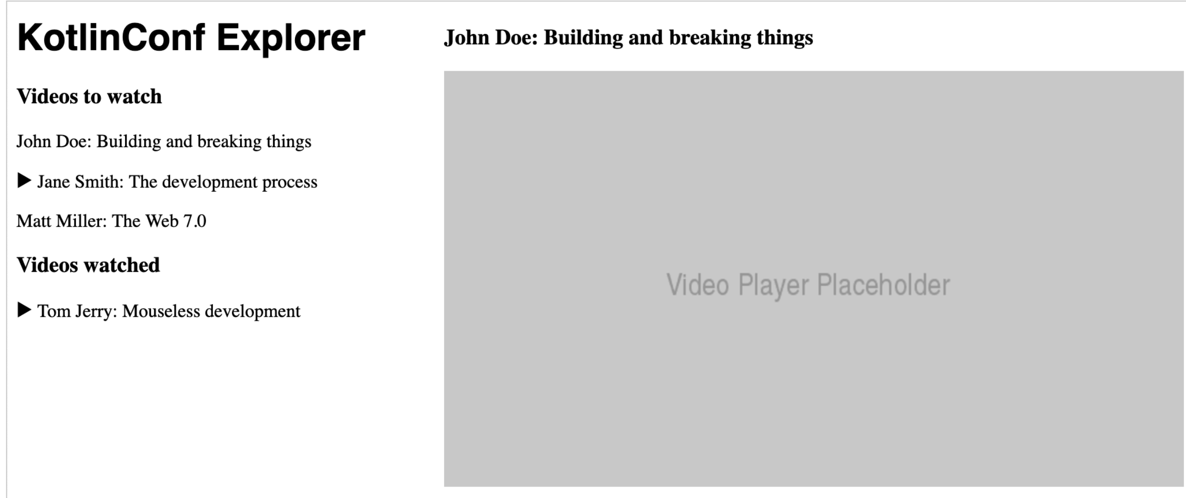
You can find more details about state management in the React FAQ (https://reactjs.org/docs/faq-state.html).

Check the browser and click an item in the list to make sure that everything is working correctly.

You can find this state of the project in the `03-first-component` branch here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle/tree/03-first-component).

Compose components

Currently, the two video lists work on their own, meaning that each list keeps track of a selected video. Users can select two videos, one in the unwatched list and one in watched, even though there's only one player:



Two videos are selected in both lists simultaneously

A list can't keep track of which video is selected both inside itself, and inside a sibling list. The reason is that the selected video is part not of the *list* state, but of the *application* state. This means you need to *lift* state out of the individual components.

Lift state

React makes sure that props can only be passed from a parent component to its children. This prevents components from being hard-wired together.

If a component wants to change state of a sibling component, it needs to do so via its parent. At that point, state also no longer belongs to any of the child components but to the overarching parent component.

The process of migrating state from components to their parents is called *lifting state*. For

your app, add `currentVideo` as state to the `App` component:

1. In `App.kt`, add the following to the top of the definition of the `App` component:

   ```
   val App = FC<Props> {
       var currentVideo: Video? by useState(null)
       // . . .
   }
   ```

   The `VideoList` component no longer needs to keep track of state. It will receive the current video as a prop instead.

2. Remove the `useState()` call in `VideoList.kt`.

3. Prepare the `VideoList` component to receive the selected video as a prop. To do so, expand the `VideoListProps` interface to contain the `selectedVideo`:

   ```
   external interface VideoListProps : Props {
       var videos: List<Video>
       var selectedVideo: Video?
   }
   ```

4. Change the condition of the triangle so that it uses `props` instead of `state`:

   ```
   if (video == props.selectedVideo) {
       +"▶ "
   }
   ```

Pass handlers

At the moment, there's no way to assign a value to a prop, so the `onClick` function won't work the way it is currently set up. To change state of a parent component, you need to lift state again.

In React, state always flows from parent to child. So, to change the *application* state from one of the child components, you need to move the logic for handling user interaction to the parent component and then pass the logic in as a prop. Remember that in Kotlin, variables can have the type of a function ("函数类型(Function Type)" in "高阶函数与 Lambda 表达式").

1. Expand the VideoListProps interface again so that it contains a variable onSelectVideo, which is a function that takes a Video and returns Unit:

```
external interface VideoListProps : Props {
    // ...
    var onSelectVideo: (Video) -> Unit
}
```

2. In the VideoList component, use the new prop in the onClick handler:

```
onClick = {
    props.onSelectVideo(video)
}
```

3. You can now go back to the App component and pass selectedVideo and a handler for onSelectVideo for each of the two video lists:

```
VideoList {
    videos = unwatchedVideos // and watchedVideos respectively
    selectedVideo = currentVideo
    onSelectVideo = { video ->
        currentVideo = video
    }
}
```

4. Repeat the previous step for the watched videos list.

Switch back to your browser and make sure that when selecting a video the selection jumps between the two lists without duplication.

You can find this state of the project on the 04-composing-components branch here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle/tree/04-composing-components).

Add more components

Extract the video player component

You can now create another self-contained component, a video player, which is currently a placeholder image. Your video player needs to know the talk title, the author of the talk, and the link to the video. This information is already contained in each `Video` object, so you can pass it as a prop and access its attributes.

1. Create a new `VideoPlayer.kt` file and add the following implementation for the `VideoPlayer` component:

```
import csstype.*
import react.*
import emotion.react.css
import react.dom.html.ReactHTML.button
import react.dom.html.ReactHTML.div
import react.dom.html.ReactHTML.h3
import react.dom.html.ReactHTML.img

external interface VideoPlayerProps : Props {
    var video: Video
}

val VideoPlayer = FC<VideoPlayerProps> { props ->
    div {
        css {
            position = Position.absolute
            top = 10.px
            right = 10.px
        }
        h3 {
            +"${props.video.speaker}: ${props.video.title}"
        }
        img {
            src = "https://via.placeholder.com/640x360.png?
text=Video+Player+Placeholder"
        }
    }
}
```

2. Because the VideoPlayerProps interface specifies that the VideoPlayer component takes a non-null Video, make sure to handle this in the App component accordingly.

   In App.kt, replace the previous div snippet for the video player with the following:

   ```
   currentVideo?.let { curr ->
       VideoPlayer {
           video = curr
       }
   }
   ```

   The let scope function ("let 函数" in "作用域函数(Scope Function)") ensures that the VideoPlayer component is only added when state.currentVideo is not null.

Now clicking an entry in the list will bring up the video player and populate it with the information from the clicked entry.

Add a button and wire it

To make it possible for users to mark a video as watched or unwatched and to move it between the two lists, add a button to the VideoPlayer component.

Since this button will move videos between two different lists, the logic handling state change needs to be *lifted* out of the VideoPlayer and passed in from the parent as a prop. The button should look different based on whether the video has been watched or not. This is also information you need to pass as a prop.

1. Expand the VideoPlayerProps interface in VideoPlayer.kt to include properties for those two cases:

   ```
   external interface VideoPlayerProps : Props {
       var video: Video
       var onWatchedButtonPressed: (Video) -> Unit
       var unwatchedVideo: Boolean
   }
   ```

2. You can now add the button to the actual component. Copy the following snippet into the body of the VideoPlayer component, between the h3 and img tags:

```
button {
    css {
        display = Display.block
        backgroundColor = if (props.unwatchedVideo)
NamedColor.lightgreen else NamedColor.red
    }
    onClick = {
        props.onWatchedButtonPressed(props.video)
    }
    if (props.unwatchedVideo) {
        +"Mark as watched"
    } else {
        +"Mark as unwatched"
    }
}
```

With the help of Kotlin CSS DSL that make it possible to change styles dynamically, you can change the color of the button using a basic Kotlin `if` expression.

Move video lists to the application state

Now it's time to adjust the `VideoPlayer` usage site in the `App` component. When the button is clicked, a video should be moved from the unwatched list to the watched list or vice versa. Since these lists can now actually change, move them into the application state:

1. In `App.kt`, add the following `useState()` calls to the top of the `App` component:

```
val App = FC<Props> {
    var currentVideo: Video? by useState(null)
    var unwatchedVideos: List<Video> by useState(listOf(
        Video(1, "Opening Keynote", "Andrey Breslav",
"https://youtu.be/PsaFVLr8t4E"),
        Video(2, "Dissecting the stdlib", "Huyen Tue Dao",
"https://youtu.be/Fzt_9I733Yg"),
        Video(3, "Kotlin and Spring Boot", "Nicolas Frankel",
"https://youtu.be/pSiZVAeReeg")
    ))
    var watchedVideos: List<Video> by useState(listOf(
        Video(4, "Creating Internal DSLs in Kotlin", "Venkat
```

```
Subramaniam", "https://youtu.be/JzTeAM8N1-o")
    ))
    // . . .
}
```

2. Since all the demo data is included in the default values for `watchedVideos` and `unwatchedVideos` directly, you no longer need the file-level declarations. In `Main.kt`, delete the declarations for `watchedVideos` and `unwatchedVideos`.

3. Change the call-site for `VideoPlayer` in the `App` component that belongs to the video player to look like this:

```
VideoPlayer {
    video = curr
    unwatchedVideo = curr in unwatchedVideos
    onWatchedButtonPressed = {
        if (video in unwatchedVideos) {
            unwatchedVideos = unwatchedVideos - video
            watchedVideos = watchedVideos + video
        } else {
            watchedVideos = watchedVideos - video
            unwatchedVideos = unwatchedVideos + video
        }
    }
}
```

Go back to the browser, select a video, and press the button a few times. The video will jump between the two lists.

You can find this state of the project in the `05-more-components` branch here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle/tree/05-more-components).

Use packages from npm

To make the app usable, you still need a video player that actually plays videos and some buttons to help people share the content.

React has a rich ecosystem with a lot of pre-made components you can use instead of building this functionality yourself.

Add the video player component

To replace the placeholder video component with an actual YouTube player, use the `react-player` package from npm. It can play videos and allows you to control the appearance of the player.

For the component documentation and the API description, see its README (https://www.npmjs.com/package/react-player) in GitHub.

1. Check the `build.gradle.kts` file. The `react-player` package should be already included:

   ```
   dependencies {
       // ...
       // Video Player
       implementation(npm("react-player", "2.10.1"))
       // ...
   }
   ```

   As you can see, npm dependencies can be added to a Kotlin/JS project by using the `npm()` function in the `dependencies` block of the build file. The Gradle plugin then takes care of downloading and installing these dependencies for you. To do so, it uses its own bundled installation of the `yarn` (https://yarnpkg.com/) package manager.

2. To use the JavaScript package from inside the React application, it's necessary to tell the Kotlin compiler what to expect by providing it with external declarations (在 Kotlin 中使用 JavaScript 代码).

   Create a new `ReactYouTube.kt` file and add the following content:

   ```
   @file:JsModule("react-player")
   @file:JsNonModule

   import react.*

   @JsName("default")
   external val ReactPlayer: ComponentClass<dynamic>
   ```

When the compiler sees an external declaration like `ReactPlayer`, it assumes that the implementation for the corresponding class is provided by the dependency and doesn't generate code for it.

The last two lines are equivalent to a JavaScript import like `require("react-player").default;`. They tell the compiler that it's certain that a component will conform to `ComponentClass<dynamic>` at runtime.

However, in this configuration, the generic type for the props accepted by `ReactPlayer` is set to `dynamic`. That means the compiler will accept any code, at the risk of breaking things at runtime.

A better alternative would be to create an `external interface` that specifies what kind of properties belong to the props for this external component. You can learn about the props' interface in the README (https://www.npmjs.com/package/react-player) for the component. In this case, use the `url` and `controls` props:

1. Adjust the content of `ReactPlayer.kt` accordingly:

```
@file:JsModule("react-player")
@file:JsNonModule

import react.*

@JsName("default")
external val ReactPlayer: ComponentClass<ReactPlayerProps>

external interface ReactPlayerProps : Props {
    var url: String
    var controls: Boolean
}
```

2. You can now use the new `ReactPlayer` to replace the gray placeholder rectangle in the `VideoPlayer` component. In `VideoPlayer.kt`, replace the `img` tag with the following snippet:

```
ReactPlayer {
    url = props.video.videoUrl
```

```
    controls = true
}
```

Add social share buttons

An easy way to share the application's content is to have social share buttons for messengers and email. You can use an off-the-shelf React component for this as well, for example, react-share (https://github.com/nygardk/react-share/blob/master/README.md):

1. Check the `build.gradle.kts` file. This npm library should already be included:

```
dependencies {
    // ...
    // Share Buttons
    implementation(npm("react-share", "4.4.0"))
    // ...
}
```

2. To use `react-share` from Kotlin, you'll need to write more basic external declarations. The examples on GitHub (https://github.com/nygardk/react-share/blob/master/demo/Demo.tsx#L61) show that a share button consists of two React components: `EmailShareButton` and `EmailIcon`, for example. Different types of share buttons and icons all have the same kind of interface. You'll create the external declarations for each component the same way you already did for the video player.

   Add the following code to a new `ReactShare.kt` file:

```
@file:JsModule("react-share")
@file:JsNonModule

import react.ComponentClass
import react.Props

@JsName("EmailIcon")
external val EmailIcon: ComponentClass<IconProps>

@JsName("EmailShareButton")
external val EmailShareButton: ComponentClass<ShareButtonProps>

@JsName("TelegramIcon")
```

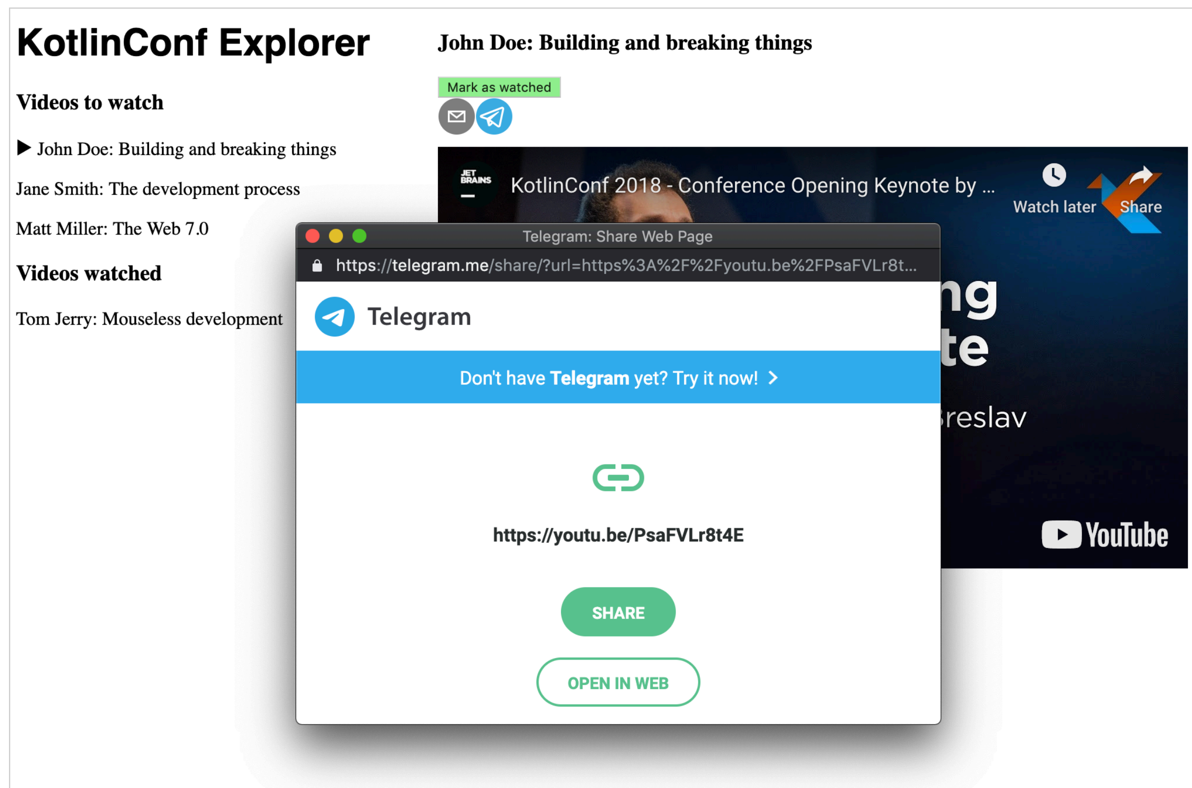
```
external val TelegramIcon: ComponentClass<IconProps>

@JsName("TelegramShareButton")
external val TelegramShareButton: ComponentClass<ShareButtonProps>

external interface ShareButtonProps : Props {
    var url: String
}

external interface IconProps : Props {
    var size: Int
    var round: Boolean
}
```

3. Add new components into the user interface of the application. In `VideoPlayer.kt`, add two share buttons in a `div` right above the usage of `ReactPlayer`:

```
// . . .
div {
    css {
         position = Position.absolute
         top = 10.px
         right = 10.px
     }
    EmailShareButton {
        url = props.video.videoUrl
        EmailIcon {
            size = 32
            round = true
        }
    }
    TelegramShareButton {
        url = props.video.videoUrl
        TelegramIcon {
            size = 32
            round = true
        }
    }
```

```
}
// . . .
```

You can now check your browser and see whether the buttons actually work. When clicking on the button, a *share window* should appear with the URL of the video. If the buttons don't show up or work, you may need to disable your ad and social media blocker.



Share window

Feel free to repeat this step with share buttons for other social networks available in react-share (https://github.com/nygardk/react-share/blob/master/README.md#features).

You can find this state of the project in the `06-packages-from-npm` branch here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle/tree/06-packages-from-npm).

Use an external REST API

You can now replace the hard-coded demo data with some real data from a REST API in the app.

For this tutorial, there's a small API (https://my-json-server.typicode.com/kotlin-hands-on/kotlinconf-json/videos/1). It offers only a single endpoint, `videos`, and takes a numeric parameter to access an element from the list. If you visit the API with your browser, you will see that the objects returned from the API have the same structure as `Video` objects.

Use JS functionality from Kotlin

Browsers already come with a large variety of Web APIs (https://developer.mozilla.org/en-US/docs/Web/API). You can also use them from Kotlin/JS, since it includes wrappers for these APIs out of the box. One example is the fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API), which is used for making HTTP requests.

The first potential issue is that browser APIs like `fetch()` use callbacks (https://developer.mozilla.org/en-US/docs/Glossary/Callback_function) to perform non-blocking operations. When multiple callbacks are supposed to run one after the other, they need to be nested. Naturally, the code gets heavily indented, with more and more pieces of functionality stacked inside each other, which makes it harder to read.

To overcome this, you can use Kotlin's coroutines, a better approach for such functionality.

The second issue arises from the dynamically typed nature of JavaScript. There are no guarantees about the type of data returned from the external API. To solve this, you can use the `kotlinx.serialization` library.

Check the `build.gradle.kts` file. The relevant snippet should already exist:

```
dependencies {
    // . . .
    // Coroutines & serialization
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.6.3")
}
```

Add serialization

When you call an external API, you get back JSON-formatted text that still needs to be turned into a Kotlin object that can be worked with.

`kotlinx.serialization` (https://github.com/Kotlin/kotlinx.serialization) is a library that makes it possible to write these types of conversions from JSON strings to Kotlin objects.

1. Check the `build.gradle.kts` file. The corresponding snippet should already exist:

```
plugins {
    // . . .
    kotlin("plugin.serialization") version "1.9.23"
}

dependencies {
    // . . .
    // Serialization
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-
json:1.3.3")
}
```

2. As preparation for fetching the first video, it's necessary to tell the serialization library about the `Video` class. In `Main.kt`, add the `@Serializable` annotation to its definition:

```
@Serializable
data class Video(
    val id: Int,
    val title: String,
    val speaker: String,
    val videoUrl: String
)
```

Fetch videos

To fetch a video from the API, add the following function in `App.kt` (or a new file):

```
suspend fun fetchVideo(id: Int): Video {
    val response = window
        .fetch("https://my-json-server.typicode.com/kotlin-hands-
on/kotlinconf-json/videos/$id")
        .await()
        .text()
        .await()
    return Json.decodeFromString(response)
}
```

* *Suspending function* `fetch()`es a video with a given `id` from the API. This response may take a while, so you `await()` the result. Next, `text()`, which uses a callback, reads the body from the response. Then you `await()` its completion.
* Before returning the value of the function, you pass it to `Json.decodeFromString`, a function from `kotlinx.coroutines`. It converts the JSON text you received from the request into a Kotlin object with the appropriate fields.
* The `window.fetch` function call returns a `Promise` object. You normally would have to define a callback handler that gets invoked once the `Promise` is resolved and a result is available. However, with coroutines, you can `await()` those promises. Whenever a function like `await()` is called, the method stops (suspends) its execution. Its execution continues once the `Promise` can be resolved.

To give users a selection of videos, define the `fetchVideos()` function, which will fetch 25 videos from the same API as above. To run all the requests concurrently, use the `async` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) functionality provided by Kotlin's coroutines:

1. Add the following implementation to your `App.kt`:

```
suspend fun fetchVideos(): List<Video> = coroutineScope {
    (1..25).map { id ->
        async {
            fetchVideo(id)
        }
    }.awaitAll()
}
```

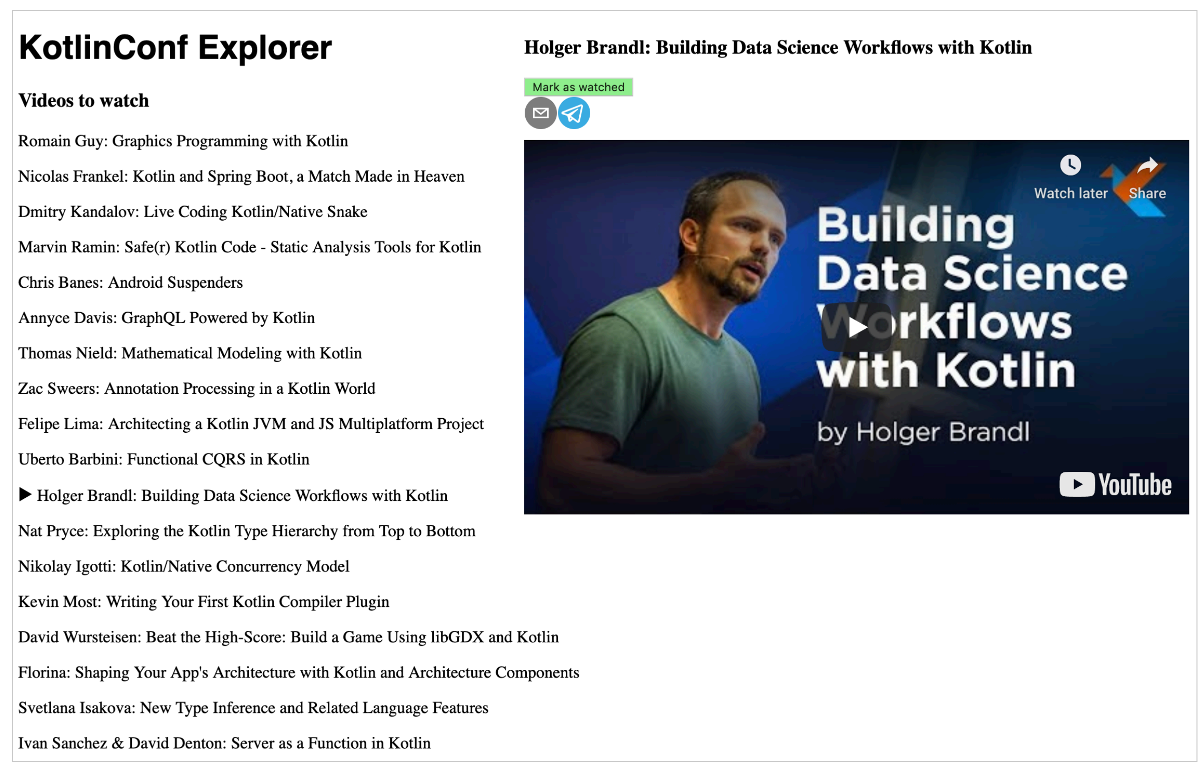
   Following the principle of structured concurrency (https://kotlinlang.org/docs/coroutines-basics.html#structured-concurrency), the implementation is wrapped in a `coroutineScope`. You can then start 25 asynchronous tasks (one per request) and wait for all of them to complete.

2. You can now add data to your application. Add the definition for a `mainScope`, and change your `App` component so it starts with the following snippet. Don't forget to replace demo values with `emptyLists` instances as well:

```
val mainScope = MainScope()

val App = FC<Props> {
    var currentVideo: Video? by useState(null)
    var unwatchedVideos: List<Video> by useState(emptyList())
    var watchedVideos: List<Video> by useState(emptyList())

    useEffectOnce {
        mainScope.launch {
            unwatchedVideos = fetchVideos()
        }
    }
// . . .
```

- The `MainScope()` (https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-main-scope.html) is a part of Kotlin's structured concurrency model and creates the scope for asynchronous tasks to run in.
- `useEffectOnce` is another React *hook* (specifically, a simplified version of the useEffect (https://reactjs.org/docs/hooks-effect.html) hook). It indicates that the component performs a *side effect*. It doesn't just render itself but also communicates over the network.

Check your browser. The application should show actual data:



Fetched data from API

When you load the page:

- The code of the `App` component will be invoked. This starts the code in the `useEffectOnce` block.
- The `App` component is rendered with empty lists for the watched and unwatched videos.
- When the API requests finish, the `useEffectOnce` block assigns it to the `App` component's state. This triggers a re-render.
- The code of the `App` component will be invoked again, but the `useEffectOnce` block *will not* run for a second time.

If you want to get an in-depth understanding of how coroutines work, check out this tutorial on coroutines ([教程 - 协程与通道(Channel)]).

You can find this state of the project in the `07-using-external-rest-api` branch here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle/tree/07-using-external-rest-api).

Deploy to production and the cloud

It's time to get the application published to the cloud and make it accessible to other people.

Package a production build

To package all assets in production mode, run the `build` task in Gradle via the tool window in IntelliJ IDEA or by running `./gradlew build`. This generates an optimized project build, applying various improvements such as DCE (dead code elimination).

Once the build has finished, you can find all the files needed for deployment in `/build/distributions`. They include the JavaScript files, HTML files, and other resources required to run the application. You can put them on a static HTTP server, serve them using GitHub Pages, or host them on a cloud provider of your choice.

Deploy to Heroku

Heroku makes it quite simple to spin up an application that is reachable under its own domain. Their free tier should be sufficient for development purposes.

1. Create an account (https://signup.heroku.com/).
2. Install and authenticate the CLI client (https://devcenter.heroku.com/articles/heroku-cli).
3. Create a Git repository and attach a Heroku app by running the following commands in the Terminal while in the project root:

   ```
   git init
   heroku create
   git add .
   git commit -m "initial commit"
   ```

4. Unlike a regular JVM application that would run on Heroku (one written with Ktor or Spring Boot, for example), your app generates static HTML pages and JavaScript files that need to be served accordingly. You can adjust the required buildpacks to serve the program properly:

   ```
   heroku buildpacks:set heroku/gradle
   heroku buildpacks:add https://github.com/heroku/heroku-buildpack-
   static.git
   ```

5. To allow the `heroku/gradle` buildpack to run properly, a `stage` task needs to be in the `build.gradle.kts` file. This task is equivalent to the `build` task, and the corresponding alias is already included at the bottom of the file:

```
// Heroku Deployment
tasks.register("stage") {
    dependsOn("build")
}
```

6. Add a new `static.json` file to the project root to configure the `buildpack-static`.

7. Add the `root` property inside the file:

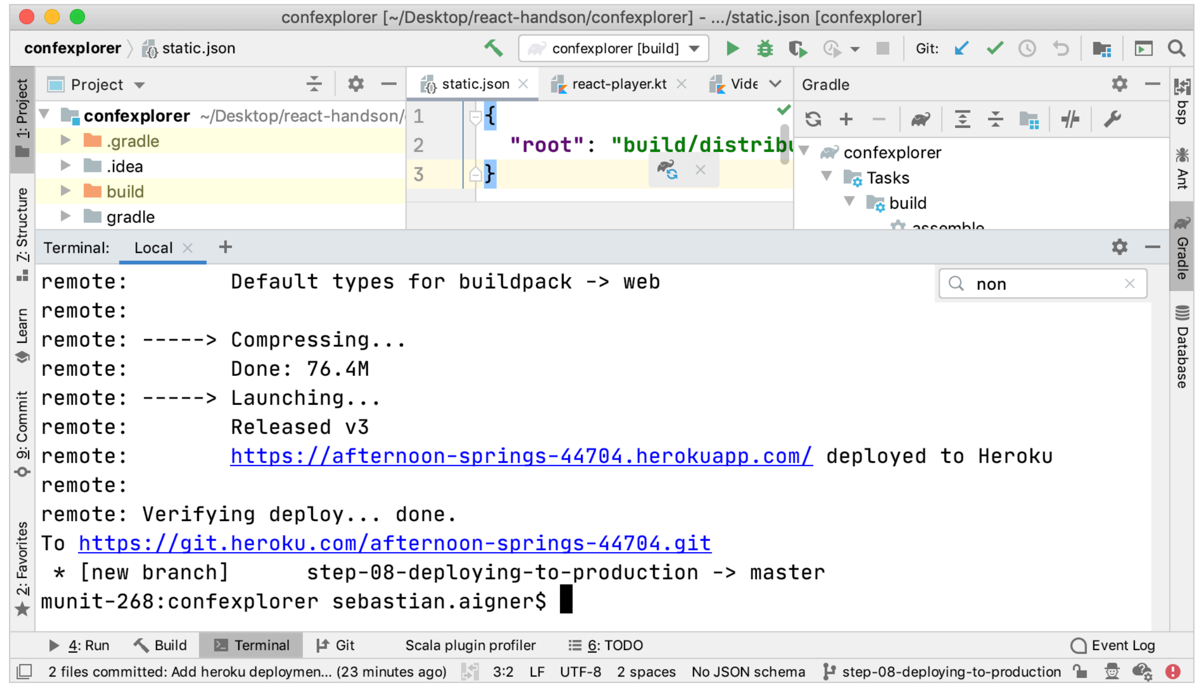
```
{
    "root": "build/distributions"
}
```

8. You can now trigger a deployment, for example, by running the following command:

```
git add -A
git commit -m "add stage task and static content root
configuration"
git push heroku master
```

If you're pushing from a non-main branch (like a step branch from the example repository), you need to adjust the command to push to the `main` remote (such as `git push heroku 08-deploying-to-production:main`).

If the deployment is successful, you will see the URL people can use to reach the application on the internet.



Web app deployment to production

You can find this state of the project in the `finished` branch here (https://github.com/kotlin-hands-on/web-app-react-kotlin-js-gradle/tree/finished).

What's next

Add more features

You can use the resulting app as a jumping-off point to explore more advanced topics in the realm of React, Kotlin/JS, and more.

- **Search**. You can add a search field to filter the list of talks – by title or by author, for example. Learn about how HTML form elements work in React (https://reactjs.org/docs/forms.html).
- **Persistence**. Currently, the application loses track of the viewer's watch list every time the page gets reloaded. Consider building your own backend, using one of the web frameworks available for Kotlin (such as Ktor (https://ktor.io/)). Alternatively, look into ways to store information on the client (https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage).

- **Complex APIs.** Lots of datasets and APIs are available. You can pull all sorts of data into your application. For example, you can build a visualizer for cat photos (https://thecatapi.com/) or a royalty-free stock photo API (https://unsplash.com/developers).

Improve the style: responsiveness and grids

The application design is still very simple and won't look great on mobile devices or in narrow windows. Explore more of the CSS DSL to make the app more accessible.

Join the community and get help

The best way to report problems and get help is the kotlin-wrappers issue tracker (https://github.com/JetBrains/kotlin-wrappers/issues). If you can't find a ticket for your issue, feel free to file a new one. You can also join the official Kotlin Slack (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up). There are channels for `#javascript` and `#react`.

Learn more about coroutines

If you're interested in finding out more about how you can write concurrent code, check out the tutorial on coroutines (教程 - 协程与通道(Channel)).

Learn more about React

Now that you know the basic React concepts and how they translate to Kotlin, you can convert some other concepts outlined in the official guides on React (https://reactjs.org/docs/) into Kotlin.

教程 - Kotlin 自定义脚本(Custom Scripting) 入门

最终更新: 2024/09/10

> Kotlin 脚本是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://kotl.in/issue) 提供你的反馈意见.

Kotlin 脚本 是这样一种技术, 它能够将 Kotlin 代码作为脚本来运行, 不需要编译并打包为可执行文件.

请观看 KotlinConf'19 大会上 Rodrigo Oliveira 的演讲 实现 Gradle Kotlin DSL (https://kotlinconf.com/2019/talks/video/2019/126701/), 这个演讲通过示例程序概要介绍 Kotlin 脚本.

在本教程中, 你将会创建一个 Kotlin 脚本项目, 它可以执行任意带 Maven 依赖项的 Kotlin 代码. 你将能够执行下面这样的脚本:

```
@file:Repository("https://maven.pkg.jetbrains.space/public/p/kotlinx
-html/maven")
@file:DependsOn("org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3")

import kotlinx.html.*
import kotlinx.html.stream.*
import kotlinx.html.attributes.*

val addressee = "World"

print(
    createHTML().html {
        body {
            h1 { +"Hello, $addressee!" }
        }
    }
)
```

在执行期间, 将会从指定的 Maven 仓库或本地缓存, 解析指定的 Maven 依赖项(这个例子中是 `kotlinx-html-jvm`), 并在脚本的其它部分中使用.

项目结构

一个最小的 Kotlin 自定义脚本项目包含两个部分:

* *脚本定义(Script Definition)* – 一组参数和配置, 定义这个脚本类型应该如何识别, 处理, 编译, 以及运行.
* *脚本主机(Scripting Host)* – 一个应用程序或组件, 负责处理脚本的编译和执行 – 实际运行这个类型的脚本.

考虑到这点, 最好将项目分为 2 个模块.

开始前的准备工作

下载并安装 IntelliJ IDEA (https://www.jetbrains.com/idea/download/index.html) 的最新版.

创建一个项目

1. 在 IntelliJ IDEA 中, 选择 **File** | **New** | **Project**.
2. 在左侧面板中, 选择 **New Project**.
3. 输入新项目名称, 如果需要, 修改它的位置.

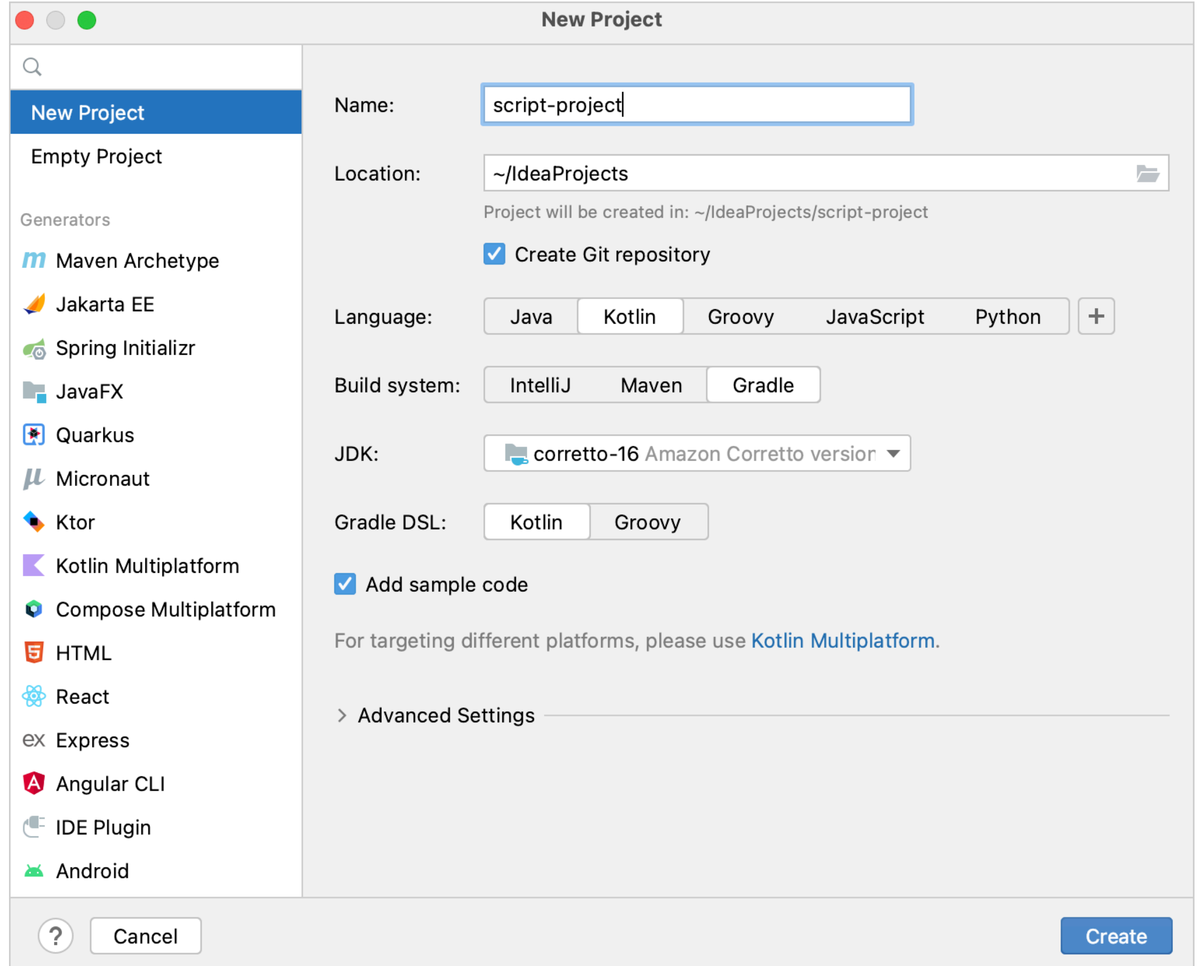
   > 可以选择 **Create Git repository** 选择框, 将新项目添加到源代码版本管理系统. 你也可以在之后的任何时候进行这个工作.

4. 在 **Language** 列表中, 选择 **Kotlin**.
5. 选择 **Gradle** 构建系统.
6. 在 **JDK** 列表中, 选择你的项目中希望使用的 JDK (https://www.oracle.com/java/technologies/downloads/).
   * 如果在你的计算机上已经安装了 JDK, 但没有在 IDE 中定义, 请选择 **Add JDK**, 并指定 JDK Home 目录的路径.

- 如果你的计算机上没有安装需要的 JDK, 请选择 **Download JDK**.

7. 请为 **Gradle DSL** 选择 Kotlin 或 Gradle 语言 .

8. 点击 **Create**.



为自定义 Kotlin 脚本创建一个根项目

添加脚本模块

现在你已经有了一个空的 Kotlin/JVM Gradle 项目. 下面我们添加需要的模块, 脚本定义模块和脚本主机模块:

1. 在 IntelliJ IDEA 中, 选择 **File | New | Module.**

2. 在左侧面板中, 选择 **New Module**. 这个模块将是我们的脚本定义模块.

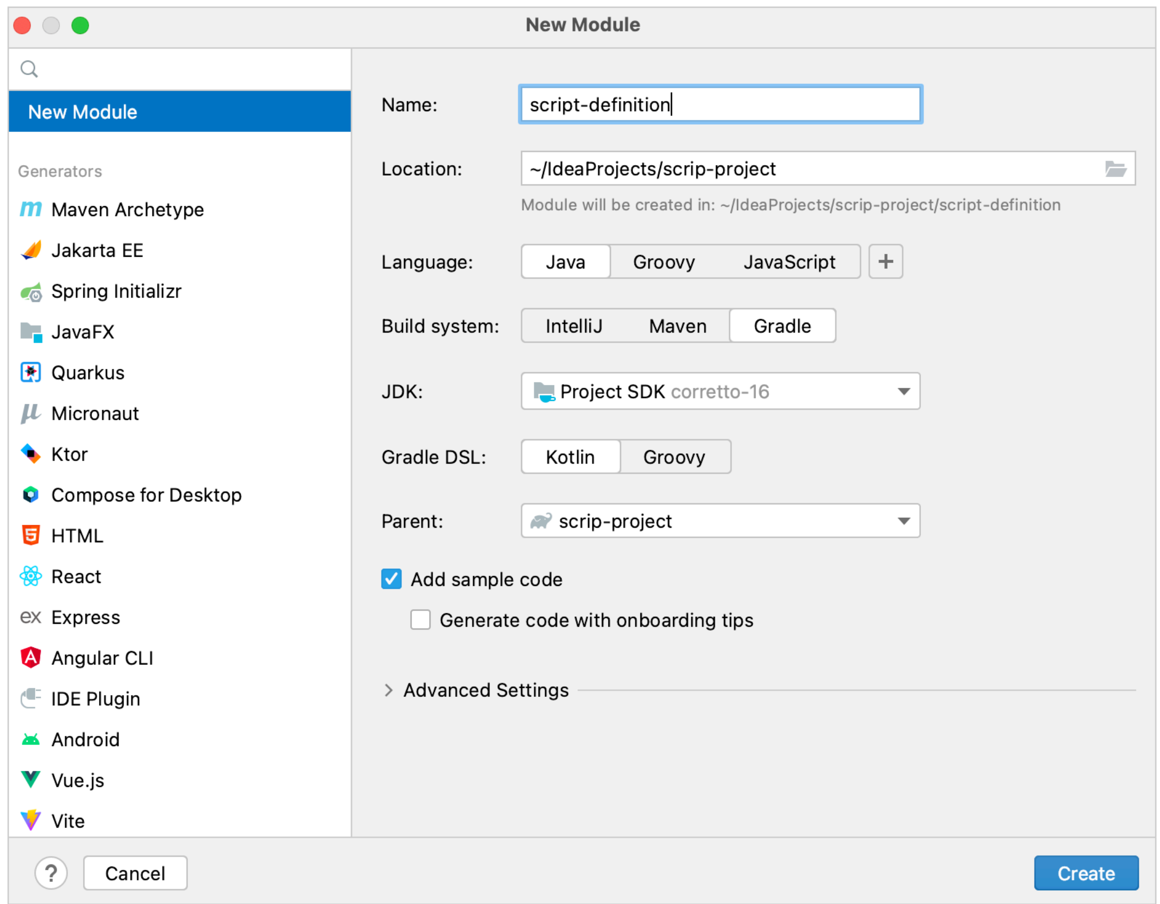
3. 输入新模块名称, 如果需要, 修改它的位置.

4. 在 **Language** 列表中, 选择 **Java**.

5. 选择 **Gradle** 构建系统, 如果你想要用 Kotlin 语言来编写构建脚本, 请为 **Gradle DSL** 选择 Kotlin.

6. 对模块的父模块, 选择根模块.

7. 点击 **Create**.

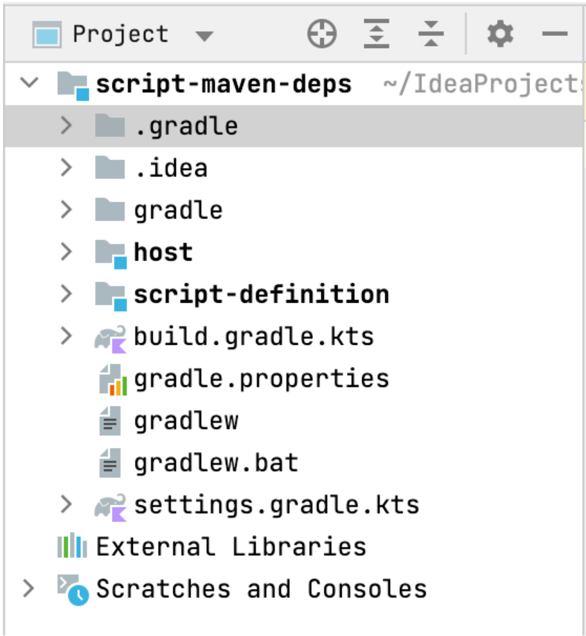


创建脚本定义模块

8. 在模块的 `build.gradle(.kts)` 文件中, 删除 Kotlin Gradle plugin 的 `version`. 因为它在根项目的构建脚本中已经定义过了.

9. 再重复一次上面的步骤, 创建一个脚本主机模块.

现在项目的结构应该如下:



自定义脚本项目的结构

请参见 kotlin-script-examples GitHub 仓库 (https://github.com/Kotlin/kotlin-script-examples/tree/master/jvm/basic/jvm-maven-deps), 你可以找到一个这样的项目的示例, 以及更多 Kotlin 脚本示例.

创建脚本定义

首先, 定义脚本类型: 开发者能够在这个类型的脚本中编写什么代码, 以及这些代码应该如何处理. 在本教程中, 这部分包括在脚本中支持 `@Repository` 和 `@DependsOn` 注解.

1. 在脚本定义模块中, 在 `build.gradle(.kts)` 文件的 `dependencies` 代码块中添加 Kotlin 脚本组件的依赖项. 这些依赖项会提供你在脚本定义中需要的 API:

   Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-scripting-common")
    implementation("org.jetbrains.kotlin:kotlin-scripting-jvm")
    implementation("org.jetbrains.kotlin:kotlin-scripting-dependencies")
    implementation("org.jetbrains.kotlin:kotlin-scripting-
```

```
dependencies-maven")
    // 对我们的脚本定义, 需要 coroutines 依赖项
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.7.3")
}
```

Groovy

```
dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-scripting-
common'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-
dependencies'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-
dependencies-maven'
    // 对我们的脚本定义, 需要 coroutines 依赖项
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-
core-jvm:1.7.3'
}
```

2. 在模块中创建 `src/main/kotlin/` 目录, 并添加一个 Kotlin 源代码文件, 例如, `scriptDef.kt`.

3. 在 `scriptDef.kt` 中, 创建一个类. 它将是这个类型的脚本的基类, 因此要将它声明为 `abstract` 或 `open`.

```
// 这个类型的脚本的 abstract (或 open) 基类
abstract class ScriptWithMavenDeps
```

这个类后面还会作为脚本定义的引用.

4. 要让这个类成为一个脚本定义, 需要为它标注 `@KotlinScript` 注解. 要向注解传递 2 个 参数:

- `fileExtension` – 一个以 `.kts` 结尾的字符串, 为这个类型的脚本定义文件扩展名.
- `compilationConfiguration` – 一个 Kotlin 类, 集成自 `ScriptCompilationConfiguration`, 为这个脚本定义指定编译规格. 你将在下一步中创建这个类.

```
// @KotlinScript 注解用来标注脚本定义类
@KotlinScript(
    // 脚本类型的文件扩展名
    fileExtension = "scriptwithdeps.kts",
    // 脚本类型的编译配置
    compilationConfiguration =
ScriptWithMavenDepsConfiguration::class
)
abstract class ScriptWithMavenDeps

object ScriptWithMavenDepsConfiguration:
ScriptCompilationConfiguration()
```

> 本教程中, 我们只提供能够工作的代码, 没有详细解释 Kotlin 脚本 API. 你可以在 GitHub 代码仓库 (https://github.com/Kotlin/kotlin-script-examples/blob/master/jvm/basic/jvm-maven-deps/script/src/main/kotlin/org/jetbrains/kotlin/script/examples/jvm/resolve/maven/scriptDef.kt) 找到相同的代码, 包含详细的解释.

5. 定义脚本的编译配置, 如下.

```
object ScriptWithMavenDepsConfiguration :
ScriptCompilationConfiguration(
    {
        // 用于这个类型的所有脚本的隐含 import
        defaultImports(DependsOn::class, Repository::class)
        jvm {
            // 从上下文 classloader 抽取完整的 classpath, 并用作依赖项
            dependenciesFromCurrentContext(wholeClasspath = true)
        }
        // 回调
        refineConfiguration {
            // 使用指定的处理器来处理特定的注解
            onAnnotations(DependsOn::class, Repository::class,
handler = ::configureMavenDepsOnAnnotations)
        }
```

```
    }
)
```

configureMavenDepsOnAnnotations 函数如下:

```
// 即时处理编译的重新配置
fun configureMavenDepsOnAnnotations(context:
ScriptConfigurationRefinementContext):
ResultWithDiagnostics<ScriptCompilationConfiguration> {
    val annotations =
context.collectedData?.get(ScriptCollectedData.collectedAnnotation
s)?.takeIf { it.isNotEmpty() }
        ?: return context.compilationConfiguration.asSuccess()
    return runBlocking {
        resolver.resolveFromScriptSourceAnnotations(annotations)
    }.onSuccess {
        context.compilationConfiguration.with {
            dependencies.append(JvmDependency(it))
        }.asSuccess()
    }
}

private val resolver =
CompoundDependenciesResolver(FileSystemDependenciesResolver(),
MavenDependenciesResolver())
```

完整的代码请参见 这里 (https://github.com/Kotlin/kotlin-script-examples/blob/master/jvm/basic/jvm-maven-deps/script/src/main/kotlin/org/jetbrains/kotlin/script/examples/jvm/resolve/maven/scriptDef.kt).

创建脚本主机

下一步是创建脚本主机 – 负责处理脚本执行的组件.

1. 在脚本主机模块中, 在 build.gradle(.kts) 文件的 dependencies 代码块中添加依赖项:

    - Kotlin 脚本组件, 它会提供你的脚本主机中需要的 API

- 你前面创建的脚本定义模块

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-scripting-
common")
    implementation("org.jetbrains.kotlin:kotlin-scripting-
jvm")
    implementation("org.jetbrains.kotlin:kotlin-scripting-jvm-
host")
    implementation(project(":script-definition")) // 脚本定义模
块
}
```

Groovy

```
dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-scripting-
common'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm-
host'
    implementation project(':script-definition') // 脚本定义模块
}
```

2. 在模块中创建 `src/main/kotlin/` 目录, 并添加一个 Kotlin 源代码 文件, 例如, `host.kt`.

3. 为应用程序定义 `main` 函数. 在它的函数体中, 检查它得到了 1 个参数 – 脚本文件的路径 – 然后执行脚本. 在下一步中, 你将会在单独的函数 `evalFile` 中定义脚本的执行过程. 现在我们只声明一个空函数.

   `main` 大致如下:

```
fun main(vararg args: String) {
    if (args.size != 1) {
        println("usage: <app> <script file>")
```

```
    } else {
        val scriptFile = File(args[0])
        println("Executing script $scriptFile")
        evalFile(scriptFile)
    }
}
```

4. 定义脚本的执行函数. 这里是你使用脚本定义的地方. 使用脚本定义类作为类型参数, 调用 `createJvmCompilationConfigurationFromTemplate`, 可以得到脚本定义. 然后调用 `BasicJvmScriptingHost().eval`, 将脚本代码和编译配置传递给它. `eval` 会返回 `ResultWithDiagnostics` 的实例, 因此请将它设置为你的函数的返回类型.

```
fun evalFile(scriptFile: File):
ResultWithDiagnostics<EvaluationResult> {
    val compilationConfiguration =
createJvmCompilationConfigurationFromTemplate<ScriptWithMavenDeps>
()
    return
BasicJvmScriptingHost().eval(scriptFile.toScriptSource(),
compilationConfiguration, null)
}
```

5. 调整 `main` 函数, 将脚本执行结果信息打印输出:

```
fun main(vararg args: String) {
    if (args.size != 1) {
        println("usage: <app> <script file>")
    } else {
        val scriptFile = File(args[0])
        println("Executing script $scriptFile")
        val res = evalFile(scriptFile)
        res.reports.forEach {
            if (it.severity > ScriptDiagnostic.Severity.DEBUG) {
                println(" : ${it.message}" + if (it.exception ==
null) "" else ": ${it.exception}")
            }
        }
```

```
    }
}
```

完整的代码请参见 这里 (https://github.com/Kotlin/kotlin-script-examples/blob/master/jvm/basic/jvm-maven-deps/host/src/main/kotlin/org/jetbrains/kotlin/script/examples/jvm/resolve/maven/host/host.kt)

运行脚本

要检查你的脚本主机如何工作, 需要准备一个要执行的脚本, 要需要一个运行配置.

1. 在项目的根目录, 创建文件 `html.scriptwithdeps.kts`, 内容如下:

```
@file:Repository("https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven")
@file:DependsOn("org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3")

import kotlinx.html.*; import kotlinx.html.stream.*; import kotlinx.html.attributes.*

val addressee = "World"

print(
    createHTML().html {
        body {
            h1 { +"Hello, $addressee!" }
        }
    }
)
```

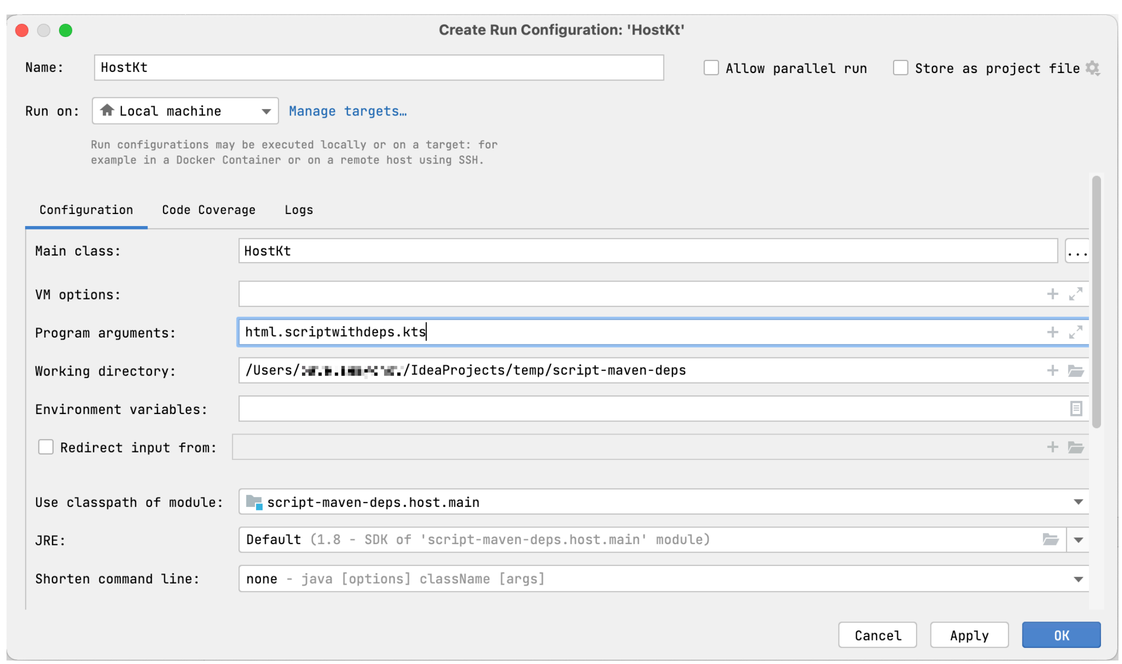
   它使用 `kotlinx-html-jvm` 库中的函数, 这个库在 `@DependsOn` 注解的参数中指定.

2. 创建一个运行配置, 它会启动脚本主机, 并执行这个文件:

   1. 点击 `host.kt`, 找到 `main` 函数. 编辑器侧栏中的有一个 **Run** 图标.

   2. 右击侧栏图标, 选择 **Modify Run Configuration**.

3. 在 **Create Run Configuration** 对话框中, 在 **Program arguments** 栏添加脚本文件名, 并点击 **OK**.



脚本主机运行配置

3. 运行创建后的配置.

你将会看到脚本是如何执行的, 它会从指定的仓库解析 `kotlinx-html-jvm` 的依赖项, 并打印出它调用的函数的结果:

```
<html>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

在第一次运行时, 解析依赖项会耗费一些时间. 之后的运行会更快结束, 因为会从本地 Maven 仓库使用已经下载好的依赖项.

下一步做什么?

你已经创建了一个简单的 Kotlin 脚本项目, 下面可以学习关于这个问题的更多内容:

- 阅读 Kotlin 脚本的 KEEP

(https://github.com/Kotlin/KEEP/blob/master/proposals/scripting-support.md)

- 浏览更多的 Kotlin 脚本示例 (https://github.com/Kotlin/kotlin-script-examples)
- 观看 Rodrigo Oliveira 的 实现 Gradle Kotlin DSL (https://kotlinconf.com/2019/talks/video/2019/126701/) 演讲

集合(Collection)概述

最终更新: 2024/09/10

Kotlin 标准库提供了丰富的工具用来管理 *集合(Collection)* – 数量可变的一组项目 (数量允许为 0), 这些集合对于解决我们的问题都非常重要, 而且使用类似的方式进行操作.

对大多数编程语言来说, 集合都是共通的概念, 所以如果你已经熟悉其他编程语言(比如 Java 或 Python)的集合, 那么你可以跳过这部分关于集合的介绍, 直接阅读后面的关于集合细节的章节.

集合通常包含一定数量的某个相同类型(及其子类型)的对象. 集合中的对象称为 *元素(element)* 或 *项目(item)*. 比如, 一个系的所有学生组成一个集合, 这个集合可以用来计算他们的平均年龄.

以下是 Kotlin 中的集合类型:

- *List* 是一个有顺序的集合, 通过下标来访问 – 下标是指反映元素位置的整数. 在一个 list 中相同的元素可以出现多次. list 的例子是电话号码: 它由许多数字组成, 数字的顺序很重要, 而且数字允许重复.
- *Set* 是由不重复的元素构成的集合. 它表示数学上的一个集(set): 一组不重复的对象. set 元素的顺序通常不重要. 例如, 彩票号码就组成一个 set: 数字不重复, 而且顺序不重要.
- *Map* (或者叫 *dictionary*) 是由成对的 键(key)-值(value) 构成的 set. 键(key)是不重复的, 并且每个键(key)对应一个值(value). 值(value)可以重复. Map 适合于存储对象之间的逻辑关联, 比如, 员工 ID 和他们职位之间的对应关系.

Kotlin 提供的集合操作功能, 与集合中元素的具体数据类型无关. 也就是说, 你可以将一个 `String` 元素添加到 `String` 组成的 list 中, 操作方法与 `Int` 组成的 list 完全相同, 与自定义类型组成的 list 也完全相同. 为此目的, Kotlin 标准库提供了泛型的接口, 类, 和函数, 可用于创建, 填充, 和管理任何数据类型组成的集合.

集合接口和相关的函数存放在 `kotlin.collections` 包之下. 下面我们大致介绍其中的内容.

> 数组不是集合(Collection)类型. 详情请参见 数组 (数组).

集合类型

Kotlin 标准库实现了基本的集合类型: set, list, 以及 map. 下面的每一对接口代表一种集合类型:

- 一个 *只读(read-only)* 接口, 提供对集合元素的访问操作.
- 一个 *可变(mutable)* 接口, 继承对应的只读接口, 另外增加了写操作: 添加, 删除, 以及更新集合元素.

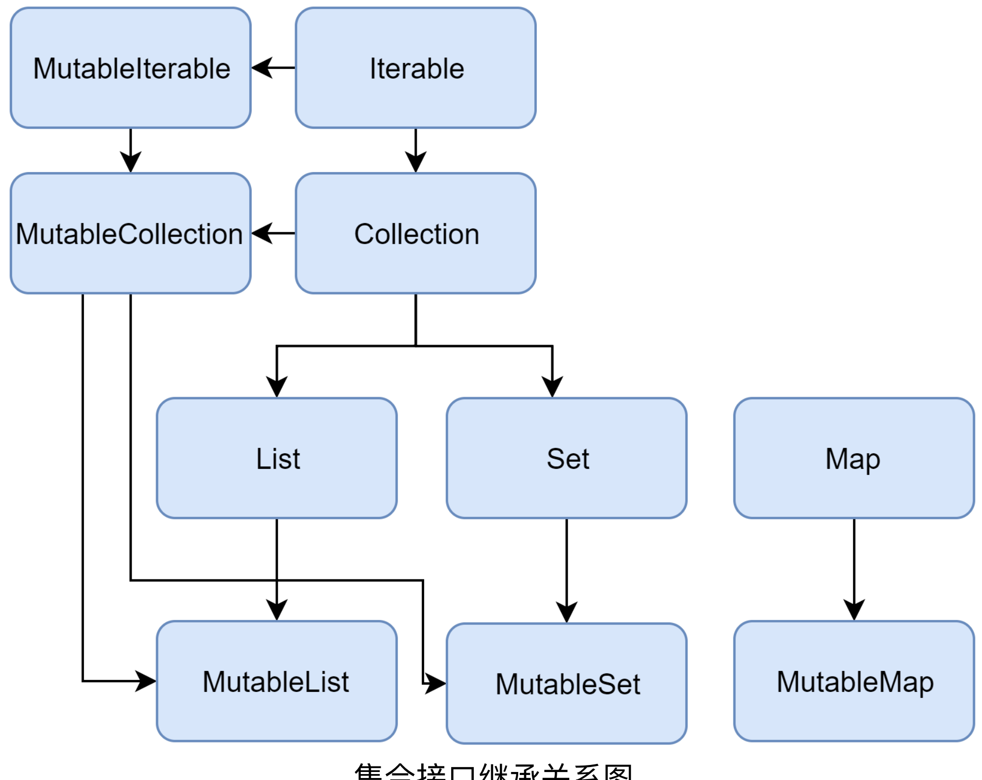
注意, 内容可变的集合, 并不要求集合变量声明为 `var` (["变量" in "基本语法"]()). 即使可变集合赋值给 `val`, 仍然可以对它进行写操作. 将可变集合复制给 `val` 的好处是, 你可以保证指向这个可变集合的引用不会被修改. 随着时间的流逝, 你的代码规模会逐渐增长, 并变得更加复杂, 防止无意的修改引用会变得更加重要. 尽可能的使用 `val`, 有助于编写更加安全和健壮的代码. 如果你想要对 `val` 类型的集合重新赋值, 会发生编译错误:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.add("five")   // 这是可以的
    println(numbers)
    //numbers = mutableListOf("six", "seven")      // 编译错误
//sampleEnd
}
```

只读的集合类型是 协变的(covariant) (["类型变异(Variance)" in "泛型(Generic): in, out, where"]()). 也就是说, 假设 `Rectangle` 类继承自 `Shape` 类, 那么在任何需要 `List<Shape>` 的地方你都可以使用 `List<Rectangle>`. 换句话说, 集合类型之间的父类型-子类型关系, 与集合中的元素类型之间的父类型-子类型关系相同. Map 类型对于它的值(value)的数据类型是协变的(covariant), 但对它的键(key)的数据类型不是.

与此相反, 可变集合不是协变的(covariant); 否则, 可能会导致运行时错误. 如果我们允许 `MutableList<Rectangle>` 成为 `MutableList<Shape>` 的子类型, 那么你可以向其中添加 `Shape` 的其他子类(比如, `Circle`), 这样就违反了元素类型必须为 `Rectangle` 的类型约束.

下面是 Kotlin 集合接口之间的继承关系图:



集合接口继承关系图

下面我们来介绍这些接口, 以及他们的实现. 关于 Collection, 请阅读本节以下部分. 关于 List, Set, 和 Map, 你可以阅读对应的章节, 也可以观看 Sebastian Aigner 讲解的视频, 他是 Kotlin 开发者 Advocate:

Collection

Collection<T> (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-collection/index.html) 是集合类型的最高层根接口. 这个接口表达只读集合的共通行为: 得到集合大小, 检查元素是否属于集合, 等等. Collection 继承自 Iterable<T> 接口, 这个接口定义了在元素上遍历的操作. 如果你的函数适用于各种不同的集合类型, 你可以适用 Collection 作为参数类型. 如果你的函数只能处理更具体的情况, 请使用 Collection 的子接口: List (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/index.html) 和 Set (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-set/index.html).

```
fun printAll(strings: Collection<String>) {
    for(s in strings) print("$s ")
    println()
}

fun main() {
    val stringList = listOf("one", "two", "one")
```

```
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet)
}
```

MutableCollection<T> (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-collection/index.html) 继承了 Collection, 并添加了元素的写操作, 比如 add 和 remove.

```
fun List<String>.getShortWordsTo(shortWords: MutableList<String>,
maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // 删除冠词(article)
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split("
")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords)
}
```

List

List<T> (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/index.html) 按指定的顺序存储元素, 并使用下标来访问元素. 下标从 0 开始 – 0 是第一个元素的下标 – 直到 lastIndex 为止, lastIndex 的值等于 (list.size - 1).

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println("Number of elements: ${numbers.size}")
    println("Third element: ${numbers.get(2)}")
    println("Fourth element: ${numbers[3]}")
    println("Index of element \"two\" ${numbers.indexOf("two")}")
```

```
//sampleEnd
}
```

List 中的元素 (包括 null) 允许重复: list 可以包含任意数量的相等对象, 也允许同一个对象多次出现. 如果两个 list 的元素数量相同, 并且相同位置的元素全都 结构相等(structurally equal) ("结构相等" in "相等判断"), 那么这两个 list 被认为是相等的.

```
data class Person(var name: String, var age: Int)

fun main() {
//sampleStart
    val bob = Person("Bob", 31)
    val people = listOf(Person("Adam", 20), bob, bob)
    val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
    println(people == people2)
    bob.age = 32
    println(people == people2)
//sampleEnd
}
```

`MutableList<T>` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/index.html) 继承了 `List`, 并添加了 list 专有的写操作, 比如, 在指定的位置添加或删除元素.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    numbers.removeAt(1)
    numbers[0] = 0
    numbers.shuffle()
    println(numbers)
//sampleEnd
}
```

你可以看到, 从某些角度看 list 与数组(array)非常类似. 但是, 它们之间存在一个重要的区别: 数组的大小是在初始化时固定的, 而且永远不能改变; 而 list 没有预定的大小; list 的大小可以通过写操作来改变: 添加, 更新, 或删除元素.

在 Kotlin 中, `MutableList` 的默认实现是 `ArrayList` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-array-list/index.html), 你可以把它看作是一个可以改变大小的数组.

Set

`Set<T>` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-set/index.html) 存储不重复的元素; 元素的顺序通常是无定义的. `null` 也算是不重复的元素: `Set` 可以只包含一个 `null`. 如果两个 set 的元素数量相同, 并且一个 set 中的任何一个元素都在另一个 set 中存在一个相等的元素, 那么这两个 set 被看作是相等的.

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3, 4)
    println("Number of elements: ${numbers.size}")
    if (numbers.contains(1)) println("1 is in the set")

    val numbersBackwards = setOf(4, 3, 2, 1)
    println("The sets are equal: ${numbers == numbersBackwards}")
//sampleEnd
}
```

`MutableSet` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-set/index.html) 继承自 `Set`, 并添加了继承自 `MutableCollection` 的写操作.

`MutableSet` 的默认实现是 `LinkedHashSet` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-linked-hash-set/index.html) – 它会保留元素插入的顺序. 因此, 依赖于元素顺序的那些函数, 比如 `first()` 或 `last()`, 在这些 set 上会返回可预测的结果.

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3, 4)  // 默认实现是 LinkedHashSet
    val numbersBackwards = setOf(4, 3, 2, 1)

    println(numbers.first() == numbersBackwards.first())
    println(numbers.first() == numbersBackwards.last())
//sampleEnd
}
```

另一个替代实现 – `HashSet` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-hash-set/index.html) – 对元素顺序不做任何保证, 因此对它调用这些函数会返回不可预知的结果. 但是, 存储相同数量的元素时, `HashSet` 消耗的内存更少.

Map

`Map<K, V>` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/index.html) 不继承自 `Collection` 接口; 但它仍然是 Kotlin 的集合类型. `Map` 存储成对的 *键(key)-值(value)* (或者叫 *条目(entry)*); 键(key)是不可重复的, 但不同的键(key)可以对应到相等的值(value). `Map` 接口提供了专用的函数, 比如根据指定的键(key)来得到对应的值(value), 查找键(key)和值(value), 等等.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key4" to 1)

    println("All keys: ${numbersMap.keys}")
    println("All values: ${numbersMap.values}")
    if ("key2" in numbersMap) println("Value by key \"key2\":
${numbersMap["key2"]}")
    if (1 in numbersMap.values) println("The value 1 is in the map")
    if (numbersMap.containsValue(1)) println("The value 1 is in the
map") // 结果与上面相同
//sampleEnd
}
```

如果两个 map 包含相等的 键(key)-值(value) 对, 那么这两个 map 被看作是相等的, 无论键(key)-值(value) 对的顺序如何.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key4" to 1)
    val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1,
"key3" to 3)

    println("The maps are equal: ${numbersMap == anotherMap}")
//sampleEnd
}
```

`MutableMap` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/index.html) 继承自 `Map`, 添加了 map 专有的写操作, 比如, 你可以添加新的键(key)-值(value) 对, 或者对指定的键(key)更新它对应的值(value).

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    numbersMap["one"] = 11

    println(numbersMap)
//sampleEnd
}
```

`MutableMap` 的默认实现是 `LinkedHashMap` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-linked-hash-map/index.html) – 它会在遍历 map 元素时使用元素插入时的顺序. 与此相反, 另一个替代实现 – `HashMap` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-hash-map/index.html) – 对元素顺序不做任何保证.

ArrayDeque

`ArrayDeque<T>` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-array-deque/) 是双向队列(double-ended queue)的一个实现, 对这种双向队列, 从前端或尾端都可以添加或删除元素. 因此, 在 Kotlin 中 `ArrayDeque` 可以同时充当 Stack 和 Queue 数据结构的角色. 在它内部的实现中, `ArrayDeque` 使用了一个可以变更大小的数组, 在需要的时候, 会自动调整数组大小:

```
fun main() {
    val deque = ArrayDeque(listOf(1, 2, 3))

    deque.addFirst(0)
    deque.addLast(4)
    println(deque) // 输出结果为 [0, 1, 2, 3, 4]

    println(deque.first()) // 输出结果为 0
    println(deque.last()) // 输出结果为 4

    deque.removeFirst()
```

```
    deque.removeLast()
    println(deque) // 输出结果为 [1, 2, 3]
}
```

创建集合

最终更新: 2024/09/10

通过指定的元素创建

创建集合最常用的方法是使用标准库中的函数 `listOf<T>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/list-of.html), `setOf<T>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/set-of.html), `mutableListOf<T>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-list-of.html), `mutableSetOf<T>()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-set-of.html). 如果使用逗号分隔的一系列集合元素作为这些函数的参数, 编译器会自动判定元素类型. 如果要创建空的集合, 就无法从参数推断元素类型, 因此需要明确指定元素类型.

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

创建 map 的方法类似, 使用的函数是 `mapOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-of.html) 和 `mutableMapOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-map-of.html). map 的键(key) 和值(value) 通过 `Pair` 对象传递给函数 (通常使用中缀函数 `to` 来创建 键(key) 和值(value) 的 `Pair` 对象).

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4"
to 1)
```

注意, 使用 `to` 这样的写法创建的是短期存在的 `Pair` 对象, 因此只有在性能问题不严重的情况下才推荐这种写法. 为避免消耗过多的内存, 可以使用其他方法. 比如, 可以创建可变的 map, 然后通过写操作向其中填充数据. 这种情况下使用 `apply()` ("apply 函数" in "作用域函数(Scope Function)") 函数可以让 map 的初始化过程更加流畅.

```
val numbersMap = mutableMapOf<String, String>().apply { this["one"]
= "1"; this["two"] = "2" }
```

使用集合构建函数来创建

创建集合的另一种方法是调用构建函数 – buildList() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-list.html), buildSet() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-set.html), 和 buildMap() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/build-map.html). 这些函数创建新的, 可变的集合, 然后使用 写操作 (集合写入操作)填充集合内容, 再返回一个只读的集合, 包含相同的元素:

```
val map = buildMap { // 在这里是 MutableMap<String, Int>, key 和 value 的类型通过下面的 `put()` 调用推断得到
    put("a", 1)
    put("b", 0)
    put("c", 4)
}

println(map) // {a=1, b=0, c=4}
```

创建空集合

还有一些函数可以用来创建不包含元素的空集合: emptyList() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/empty-list.html), emptySet() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/empty-set.html), 以及 emptyMap() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/empty-map.html). 创建空集合时, 你需要明确指定集合中存储的元素的数据类型.

```
val empty = emptyList<String>()
```

使用 list 的初始化函数

对于 list, 有一个类似构造器的函数, 它接受的参数是 list 大小, 以及一个初始化函数, 这个初始化函数负责根据元素的下标计算各个元素的值.

```
fun main() {
//sampleStart
    val doubled = List(3, { it * 2 })  // 如果希望将来改变其中的值, 可以使用 MutableList
```

```
    println(doubled)
//sampleEnd
}
```

使用具体类型的(Concrete type) 集合构造器

如果希望创建一个具体类型(concrete type)的集合, 比如 `ArrayList` 或 `LinkedList`, 可以使用这些集合类型的构造器. `Set` 和 `Map` 的实现类也有类似的构造器.

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))
val presizedSet = HashSet<Int>(32)
```

从既有的集合复制

如果想要从已有的集合创建一个相同内容的新集合, 可以使用复制(copy)函数. 标准库中提供的集合复制函数创建的是 *浅(shallow)* 复制的集合, 其中的元素与既有的集合指向相同的对象引用. 因此, 对集合中某个元素进行修改, 会反映到这个元素的所有副本.

集合复制函数, 比如 `toList()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-list.html), `toMutableList()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-mutable-list.html), `toSet()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-set.html) 等等, 会创建一个集合在函数调用那一刻的副本(snapshot). 这些函数的结果是一个新的集合, 但包含完全相同的元素. 如果对原来的集合添加或删除元素, 不会影响到副本集合中的内容. 同样的, 对副本的修改也不会影响到原来的集合.

```
class Person(var name: String)
fun main() {
//sampleStart
    val alice = Person("Alice")
    val sourceList = mutableListOf(alice, Person("Bob"))
    val copyList = sourceList.toList()
    sourceList.add(Person("Charles"))
    alice.name = "Alicia"
    println("First item's name is: ${sourceList[0].name} in source
and ${copyList[0].name} in copy")
    println("List size is: ${sourceList.size} in source and
${copyList.size} in copy")
```

```
//sampleEnd
}
```

这些函数也可以用来将一个集合转换为其他类型的集合, 比如从一个 list 创建一个 set, 或者相反.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val copySet = sourceList.toMutableSet()
    copySet.add(3)
    copySet.add(4)
    println(copySet)
//sampleEnd
}
```

另一种方法是, 你可以创建一个新的引用, 指向相同的集合实例. 如果用一个既有的集合初始化赋值给一个集合变量, 这时就创建了一个新的引用. 这种情况下, 如果通过一个引用修改了集合实例的内容, 这些变化会影响到所有其他的引用.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList = sourceList
    referenceList.add(4)
    println("Source size: ${sourceList.size}")
//sampleEnd
}
```

集合的初始化可以用来限制它的可变性. 比如, 如果创建一个 `List` 引用, 指向一个 `MutableList`, 那么如果你当你想要使用这个只读类型的 `List` 引用来修改集合内容, 编译器会报告编译错误.

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList: List<Int> = sourceList
    //referenceList.add(4)            // 编译错误
    sourceList.add(4)
    println(referenceList) // 显示 sourceList 的当前状态
```

```
//sampleEnd
}
```

调用其他集合的函数

在既有的集合上执行各种操作的结果也可以创建新的集合. 比如, 过滤(filtering) (过滤(Filtering)集合) 一个 list 会创建一个新的 list, 其中只包含满足过滤条件的元素:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)
//sampleEnd
}
```

映射(Mapping) ("映射(Mapping)" in "集合变换操作") 函数会创建一个新的 list, 其中包含各个元素变换后的结果:

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
//sampleEnd
}
```

关联(Association) ("关联(Association)" in "集合变换操作") 函数会创建 map:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

关于 Kotlin 集合的各种操作, 详情请参见 集合的各种操作(Operation)概述 (集合操作概述).

迭代器(Iterator)

最终更新: 2024/09/10

为了遍历集合中的元素, Kotlin 标准库支持常用的 *迭代器(Iterator)* 机制 – 迭代器可以用来按顺序访问集合元素, 而不必暴露集合的底层细节. 迭代器适用于逐个处理集合中的所有元素, 比如, 打印所有元素的值, 或者对所有元素进行类似的修改.

对于 `Iterable<T>` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterable/index.html) 接口的后代接口, 包括 `Set` 和 `List`, 调用 `iterator()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterable/iterator.html) 函数即可得到迭代器.

得到迭代器之后, 它指向集合中的第一个元素; 调用 `next()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterator/next.html) 函数会返回当前元素, 然后, 如果下一个元素存在的话, 会将迭代器的位置指向下一个元素.

迭代器经过最后一个元素之后, 它就不能再用来获取元素了; 也不能再移动到以前的位置. 如果要再次遍历集合, 必须创建新的迭代器.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val numbersIterator = numbers.iterator()
    while (numbersIterator.hasNext()) {
        println(numbersIterator.next())
        // 输出结果为
        // one
        // two
        // three
        // four
    }
//sampleEnd
}
```

遍历一个 `Iterable` 集合的另一种方法是使用大家都熟悉的 `for` 循环语句. 当在集合上使用 `for` 循环时, 会隐含地得到迭代器. 因此, 下面的示例代码与前面的例子是等价的:

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    for (item in numbers) {
        println(item)
        // 输出结果为
        // one
        // two
        // three
        // four
    }
//sampleEnd
}
```

最后, 还有一个方便的 `forEach()` 函数, 可以用来遍历集合, 并对每个元素执行一段指定的代码. 因此, 前面的示例可以写成这样:

```kotlin
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    numbers.forEach {
        println(it)
        // 输出结果为
        // one
        // two
        // three
        // four
    }
//sampleEnd
}
```

List 迭代器

对于 list, 有一个特别的迭代器实现类: `ListIterator` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/index.html). 这个迭代器支持在 list 上双方向的遍历: 向前, 以及向后.

反向遍历通过 `hasPrevious()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/has-previous.html) 和 `previous()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/previous.html) 函数实现. 此外, `ListIterator` 还可以通过 `nextIndex()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/next-index.html) 和 `previousIndex()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/previous-index.html) 函数得到遍历中元素的下标位置.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val listIterator = numbers.listIterator()
    while (listIterator.hasNext()) listIterator.next()
    println("Iterating backwards:")
    // 反向遍历:
    while (listIterator.hasPrevious()) {
        print("Index: ${listIterator.previousIndex()}")
        println(", value: ${listIterator.previous()}")
        // 输出结果为
        // Index: 3, value: four
        // Index: 2, value: three
        // Index: 1, value: two
        // Index: 0, value: one
    }
//sampleEnd
}
```

由于拥有双方向的遍历能力, 因此 `ListIterator` 在到达最后元素之后, 仍然可以继续使用.

可变的迭代器

要在可变的集合上进行遍历, 可以使用 `MutableIterator` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-iterator/index.html), 这个接口继承自 `Iterator`, 添加了元素删除函数 `remove()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-iterator/remove.html). 因此, 你可以在遍历集合的过程中删除元素.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val mutableIterator = numbers.iterator()

    mutableIterator.next()
    mutableIterator.remove()
    println("After removal: $numbers")
    // 输出结果为
    // After removal: [two, three, four]
//sampleEnd
}
```

对于可变的 list, 可以使用 MutableListIterator (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list-iterator/index.html). 除了删除元素之外, 这个迭代器还可以在遍历 list 的过程中, 使用 add() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list-iterator/add.html) 和 set() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list-iterator/set.html) 函数, 添加或替换元素.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "four", "four")
    val mutableListIterator = numbers.listIterator()

    mutableListIterator.next()
    mutableListIterator.add("two")
    println(numbers)
    // 输出结果为 [one, two, four, four]
    mutableListIterator.next()
    mutableListIterator.set("three")
    println(numbers)
    // 输出结果为 [one, two, three, four]
//sampleEnd
}
```

值范围(Range)与数列(Progression)

最终更新: 2024/09/10

Kotlin 允许你非常便利的创建值范围, 方法是使用 `kotlin.ranges` 包中的 `.rangeTo()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/range-to.html) 和 `.rangeUntil()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/range-until.html) 函数.

如果要创建:

- 终端封闭(closed-ended)的值范围, 请使用 `..` 操作符, 调用 `.rangeTo()` 函数.
- 终端开放(open-ended)的值范围, 请使用 `..<` 操作符, 调用 `.rangeUntil()` 函数.

例如:

```
fun main() {
//sampleStart
    // 终端封闭的值范围
    println(4 in 1..4)
    // true

    // 终端开放的值范围
    println(4 in 1..<4)
    // false
//sampleEnd
}
```

值范围非常适合用在 `for` 循环中遍历:

```
fun main() {
//sampleStart
    for (i in 1..4) print(i)
    // 输出结果为 1234
//sampleEnd
}
```

如果需要按反序遍历整数, 请使用标准库中的 `downTo`

(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/down-to.html) 函数代替 `..`.

```
fun main() {
//sampleStart
    for (i in 4 downTo 1) print(i)
    // 输出结果为 4321
//sampleEnd
}
```

还可以使用任意步长(不一定是 1)来遍历整数. 可以通过 `step` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/step.html) 函数实现.

```
fun main() {
//sampleStart
    for (i in 0..8 step 2) print(i)
    println()
    // 输出结果为 02468
    for (i in 0..<8 step 2) print(i)
    println()
    // 输出结果为 0246
    for (i in 8 downTo 0 step 2) print(i)
    // 输出结果为 86420
//sampleEnd
}
```

数列(Progression)

整数类型, 比如 `Int`, `Long`, 和 `Char`, 的值范围, 可以被当作 算数数列(Arithmetic Progression) (https://en.wikipedia.org/wiki/Arithmetic_progression). 在 Kotlin 中, 这些数列由相应的类型来定义: `IntProgression` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-int-progression/index.html), `LongProgression` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-long-progression/index.html), 以及 `CharProgression` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-char-progression/index.html).

数列有 3 个基本属性: `first` 元素, `last` 元素, 以及一个非 0 的 `step`. 数列的第一个元素就是 `first`, 后续的所有元素等于前一个元素加上 `step`. 在 step 为正数的数列上的遍历, 等价于 Java/JavaScript 中基于下标的 `for` 循环:

```
for (int i = first; i <= last; i += step) {
    // ...
}
```

当你在值范围上遍历时会隐含地创建一个数列, 这个数列的 `first` 和 `last` 元素就是值范围的边界值, `step` 为 1.

```
fun main() {
//sampleStart
    for (i in 1..10) print(i)
    // 输出结果为 12345678910
//sampleEnd
}
```

如果要自定义数列的步长, 可以在值范围上使用 `step` 函数.

```
fun main() {
//sampleStart
    for (i in 1..8 step 2) print(i)
    // 输出结果为 1357
//sampleEnd
}
```

数列的 `last` 元素计算方法如下:

- 如果步长为正: 小于或等于值范围结束值的最大值, 并且满足 `(last - first) % step == 0`.
- 如果步长为负: 大于或等于值范围结束值的最小值, 并且满足 `(last - first) % step == 0`.

因此, `last` 元素并不一定等同于值范围中指定的结束值.

```
fun main() {
//sampleStart
    for (i in 1..9 step 3) print(i) // last 元素是 7
    // 输出结果为 147
//sampleEnd
}
```

数列实现了 `Iterable<N>` 接口, 这里的 `N` 分别是 `Int`, `Long`, 或 `Char`, 因此数列可以用于很多 集合函数 (集合操作概述), 比如 `map`, `filter`, 等等.

```
fun main() {
//sampleStart
    println((1..10).filter { it % 2 == 0 })
    // 输出结果为 [2, 4, 6, 8, 10]
//sampleEnd
}
```

序列(Sequence)

最终更新: 2024/09/10

除集合之外, Kotlin 还提供了另一种类型 – *序列(sequence)* (`Sequence<T>` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/-sequence/index.html)). 与集合(Collection)不同, 序列并不包含元素, 而是在迭代时生成元素. 序列提供的函数和 `Iterable` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterable/index.html) 一样, 但对多步骤的集合处理提供另一种实现方式.

当对 `Iterable` 的处理包含多个步骤时, 会以及早计算(eager)模式执行: 每个处理步骤都会执行完毕, 并返回它的结果 – 也就是一个中间集合. 然后再对这个中间集合执行下一个步骤. 与此不同, 对序列的多步骤处理会尽量以延迟计算(lazy)模式执行: 只有在整个处理链的结果真正被使用到时, 才会执行相应的计算处理.

操作的执行顺序也不同: `Sequence` 对每个元素执行所有的处理步骤. 而 `Iterable` 会对整个集合执行单个处理步骤, 然后再对结果集合执行下一个处理步骤.

通过这种方式, 序列可以避免生成各个处理步骤的中间结果, 因此能够提高集合多步骤处理的整体性能. 然而, 序列的延迟计算(lazy)模式会增加一些开销, 在处理小集合, 或进行简单计算时, 这些开销可能会比较显著. 因此, 应该同时考虑使用 `Sequence` 和 `Iterable`, 根据你的具体情况决定哪一个比较好.

创建序列

通过指定的元素创建

要创建序列, 可以使用 `sequenceOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/sequence-of.html) 函数, 通过函数参数指定序列中的元素.

```
val numbersSequence = sequenceOf("four", "three", "two", "one")
```

通过 Iterable 创建

如果你已经有了一个 `Iterable` 对象 (比如 `List` 或 `Set`), 你可以调用它的 `asSequence()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/as-sequence.html). 函数来创建序列.

```
val numbers = listOf("one", "two", "three", "four")
val numbersSequence = numbers.asSequence()
```

通过函数创建

创建序列的另一种方式是, 通过一个函数来计算序列中的元素. 调用 `generateSequence()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/generate-sequence.html) 函数, 把序列元素的计算函数作为它的参数, 这样就可以通过函数来创建序列. 这个函数有一个可选的参数, 你可以明确指定第一个元素的值, 也可以指定一个函数来计算第一个元素的值. 当序列元素的计算函数返回 `null` 时, 序列的生成过程会停止. 所以, 下面示例中的序列是无限的.

```
fun main() {
//sampleStart
    val oddNumbers = generateSequence(1) { it + 2 } // `it` 是前一个元
素
    println(oddNumbers.take(5).toList())
    //println(oddNumbers.count())     // 这里会发生运行时错误: 序列是无限
的
//sampleEnd
}
```

如果想要使用 `generateSequence()` 函数创建有限的序列, 那么你的序列元素的计算函数应该在生成最后一个元素之后返回 `null`.

```
fun main() {
//sampleStart
    val oddNumbersLessThan10 = generateSequence(1) { if (it < 8) it
+ 2 else null }
    println(oddNumbersLessThan10.count())
//sampleEnd
}
```

通过数据块(chunk)创建

最后, 还有 `sequence()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/sequence.html) 函数, 可以逐个生成序列元素, 或者通过任意大小的数据块来生成元素. 这个函数的参数是一个 lambda 表达式, 其中包括对 `yield()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/-

sequence-scope/yield.html) 函数和 yieldAll() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/-sequence-scope/yield-all.html) 函数的调用. 这些函数会将元素返回给序列的使用者, 然后暂停 sequence() 函数的执行, 直到序列使用者请求下一个元素. yield() 的参数是单个元素; yieldAll() 的参数可以是一个 Iterable 对象, 或一个 Iterator, 或另一个 Sequence. yieldAll() 函数的 Sequence 参数可以是无限的. 但是, 这样的调用必须出现在序列的最末尾: 否则, 在这之后的所有序列元素都不会被执行到.

```
fun main() {
//sampleStart
    val oddNumbers = sequence {
        yield(1)
        yieldAll(listOf(3, 5))
        yieldAll(generateSequence(7) { it + 2 })
    }
    println(oddNumbers.take(5).toList())
//sampleEnd
}
```

序列的操作

根据对数据状态的要求不同, 序列的操作可以分为以下两类:

- *无状态(Stateless)* 操作, 不需要保存状态信息, 对每个元素的处理都是独立的, 比如, map() ("映射(Mapping)" in "集合变换操作") 或 filter() (过滤(Filtering)集合). 无状态操作本身可以使用固定数量的少量状态数据来处理一个元素, 比如, take() 或 drop() (获取集合的一部分).
- *有状态(Stateful)* 操作, 需要大量的状态信息, 通常正比于序列内的元素数量.

如果序列的一个操作返回另一个序列, 结果序列的内容是延迟计算的, 我们称这个操作为 *中间(intermediate)* 操作. 相反, 如果一个操作不返回新的序列, 那么称为 *终止(terminal)* 操作. 终止操作的例子, 比如 toList() ("从既有的集合复制" in "创建集合") 或 sum() (聚合(Aggregate)操作). 只有执行终止操作后, 才能取得序列中的元素.

序列元素可以多次遍历; 序列的某些实现类可能造成限制, 使得它只能遍历一次. 这样的限制会在这些序列的文档中明确说明.

序列处理的示例

下面我们通过一个示例来看看 Iterable 和 Sequence 区别.

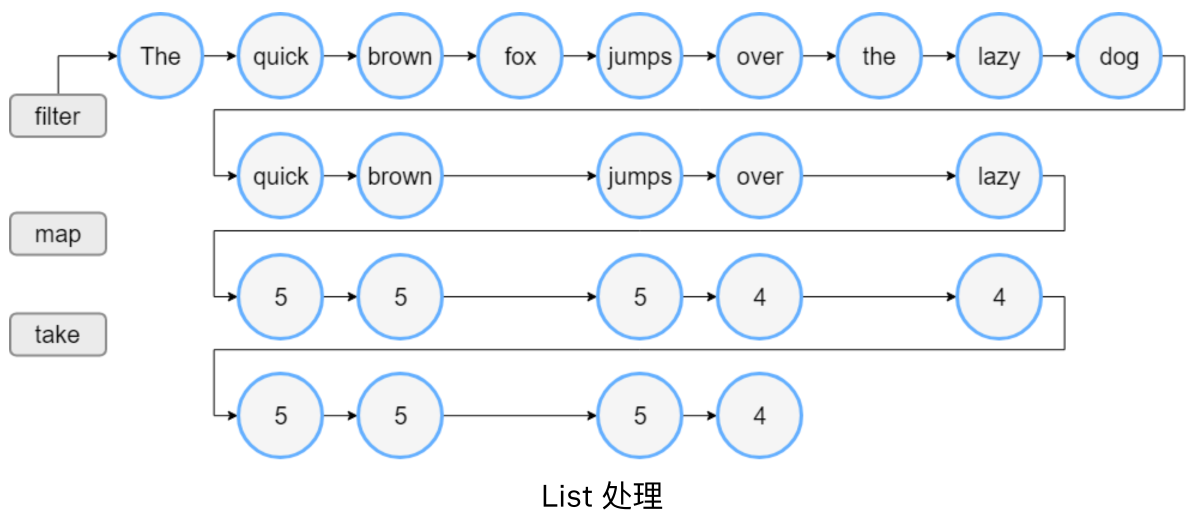
使用 Iterable

假设你有很多单词. 下面的代码会过滤长度超过 3 个字母的单词, 然后打印前 4 个这种单词的长度.

```
fun main() {
//sampleStart
    val words = "The quick brown fox jumps over the lazy
dog".split(" ")
    val lengthsList = words.filter { println("filter: $it");
it.length > 3 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars:")
    println(lengthsList)
//sampleEnd
}
```

运行这段代码时, 你可以看到 `filter()` 和 `map()` 函数的执行顺序与它们在代码中出现的顺序相同. 首先, 你会看到对所有元素输出 `filter:`, 然后对过滤之后剩余的元素输出 `length:`, 然后是最后两行代码的输出.

下图是 list 各处理步骤的具体执行过程:



List 处理

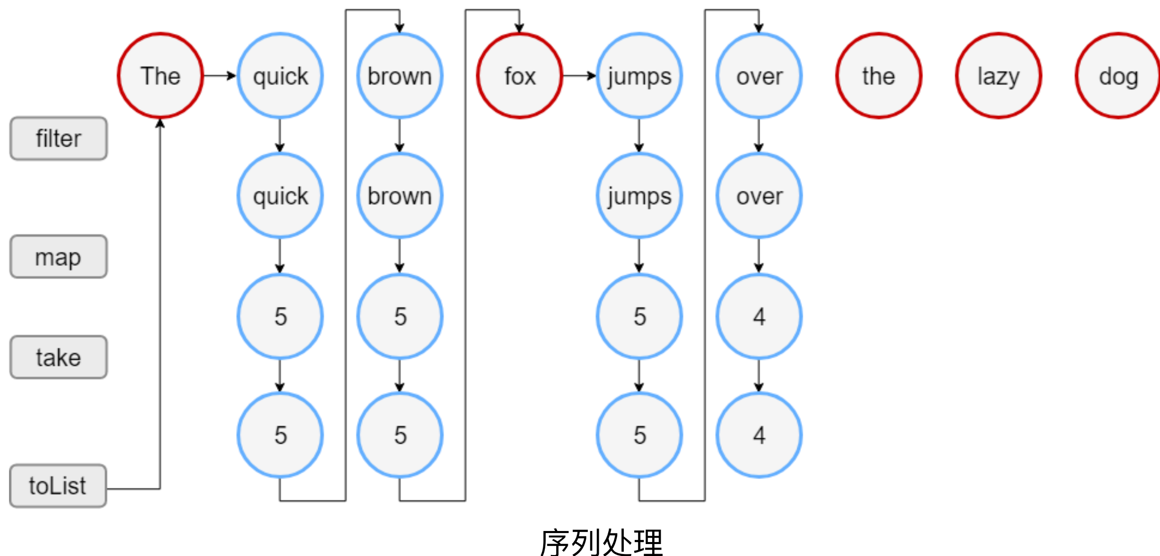
使用序列

下面我们用序列来实现同样的处理:

```
fun main() {
//sampleStart
    val words = "The quick brown fox jumps over the lazy
dog".split(" ")
    // 将 List 转换为序列
    val wordsSequence = words.asSequence()

    val lengthsSequence = wordsSequence.filter { println("filter:
$it"); it.length > 3 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars")
    // 终止操作: 以 List 形式获取结果
    println(lengthsSequence.toList())
//sampleEnd
}
```

这段代码的输出显示, `filter()` 和 `map()` 函数在创建最终的结果 list 时才被执行. 因此, 你首先看到的输出是 `"Lengths of.."`, 然后序列的处理才会开始. 注意, 对于过滤之后剩余的元素, 会在过滤下一个元素之前执行 map 操作. 当结果的元素数量到达 4 时, 处理将会停止, 因为这是 `take(4)` 能够返回的最大元素数量.

序列的处理过程如下:



序列处理

在这个示例中, 序列处理执行了 18 步, 而使用 list 时则需要 23 步.

集合操作概述

最终更新: 2024/09/10

Kotlin 标准库提供了大量的函数用来在集合上进行各种操作. 包括简单的操作, 比如获取元素, 添加元素, 以及更复杂的操作, 比如查找, 排序, 过滤(Filtering), 变换(Transformation), 等等.

扩展函数与成员函数

标准库中定义的操作有两类: 集合接口的 成员函数 ("类成员" in "类"), 以及 扩展函数 ("扩展函数(Extension Function)" in "扩展").

成员函数定义了的集合类型的基本操作. 比如, `Collection` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-collection/index.html) 包含函数 `isEmpty()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-collection/is-empty.html) 用来检查集合是否为空; `List` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/index.html) 包含 `get()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/get.html) 函数, 用于按下标访问元素, 等等.

当你自己实现集合接口时, 你必须实现集合的成员函数. 为了更简单地实现集合, 你可以使用标准库中集合接口的框架实现类: `AbstractCollection` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-abstract-collection/index.html), `AbstractList` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-abstract-list/index.html), `AbstractSet` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-abstract-set/index.html), `AbstractMap` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-abstract-map/index.html), 以及与它们对应的可变集合类.

其他集合操作声明为扩展函数. 包括过滤, 变换, 排序, 以及其他的集合处理函数.

共通操作

共通操作对 只读和可变集合 ("集合类型" in "集合(Collection)概述") 都有效. 共通操作包括以下几组:

- 变换 (集合变换操作)
- 过滤 (过滤(Filtering)集合)

* 加法和减法操作符 (加法(Plus)和减法(Minus)操作符)
* 分组 (分组(Grouping))
* 截取集合中的一部分 (获取集合的一部分)
* 取得集合中的单个元素 (获取集合的单个元素)
* 排序 (排序(Ordering))
* 聚合操作 (聚合(Aggregate)操作)

这些章节中介绍的集合操作返回的结果不会影响原来的集合. 比如, 过滤操作会产生一个 *新集合*, 其中包含符合过滤条件的所有元素. 因此, 这些操作的结果需要保存在变量中, 或者通过其他方式使用, 比如作为参数传递给其他函数.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    numbers.filter { it.length > 3 }  // `numbers` 不会受影响, 过滤结果会丢失
    println("numbers are still $numbers")
    val longerThan3 = numbers.filter { it.length > 3 } // 过滤结果保存在变量 `longerThan3` 中
    println("numbers longer than 3 chars are $longerThan3")
//sampleEnd
}
```

对某些集合操作, 可以指定 *目标* 对象. 这里的目标集合是一个可变的集合, 集合操作函数会将结果添加到目标集合中, 而不是创建新的集合作为返回值. 要使用目标集合来进行操作, 需要使用其他函数, 函数名带有 `To` 后缀, 比如, 要使用 `filterTo()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-to.html) 而不是 `filter()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html), 或者使用 `associateTo()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/associate-to.html) 而不是 `associate()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/associate.html). 这些函数接受一个额外的参数来指定目标集合.

```
fun main() {
//sampleStart
```

```
    val numbers = listOf("one", "two", "three", "four")
    val filterResults = mutableListOf<String>()  // 目标集合
    numbers.filterTo(filterResults) { it.length > 3 }
    numbers.filterIndexedTo(filterResults) { index, _ -> index == 0
}
    println(filterResults) // 目标集合中包含两次操作的全部结果
//sampleEnd
}
```

为了方便使用, 这些函数会将目标集合作为返回值, 因此你可以在调用这些函数的参数中直接创建目标函数的实例:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    // 过滤 numbers, 结果直接保存到一个新创建的 hash set,
    // 因此, 结果中重复的数字会被删除
    val result = numbers.mapTo(HashSet()) { it.length }
    println("distinct item lengths are $result")
//sampleEnd
}
```

带目标集合的函数可以用来执行过滤, 关联(associate), 分组, 压扁(flatten), 以及其他操作. 关于带目标集合的所有操作, 请参见 Kotlin 集合 API 文档 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html).

写入操作

对于可变的集合, 还有 *写操作* 可以改变集合的状态. 这类写操作包括添加元素, 删除元素, 以及变更元素. 关于写入操作的详情, 请参见 集合写入操作 (集合写入操作), 以及 List 相关操作 ("List 的写入操作" in "List 相关操作") 和 Set 相关操作 ("Map 的写入操作" in "Map 相关操作") 中对应的章节.

对于特定的写入操作, 存在一对函数用来执行相同的操作: 一个函数直接在原来的集合上进行变更, 另一个则将变更后的结果作为新的集合返回, 而不影响原来的集合. 比如, `sort()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sort.html) 函数直接对一个可变集合进行排序, 因此集合状态会发生变化; 而 `sorted()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted.html) 函数则会创建一个新的集合, 其中包含相同的元素排序后的结果, 而原来的集合不会变化.

```kotlin
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val sortedNumbers = numbers.sorted()
    println(numbers == sortedNumbers)  // 结果为 false
    numbers.sort()
    println(numbers == sortedNumbers)  // 结果为 true
//sampleEnd
}
```

集合变换操作

最终更新: 2024/09/10

Kotlin 标准库提供了一组扩展函数用于集合的 *变换(Transformation)*. 这些函数会使用指定的变换规则从原集合创建新的集合. 本节中, 我们概要介绍集合的这些变换函数.

映射(Mapping)

映射(Mapping) 变换, 会将集合的每个元素传递给一个函数, 然后用函数结果创建一个新的集合. 最基本的映射函数是 `map()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map.html). 它将每个元素传递给指定的 lambda 函数, 然后用 lambda 函数返回的结果创建一个 list. 结果的顺序与原集合中的元素顺序相同. 如果变换时还需要元素下标参数, 请使用 `mapIndexed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-indexed.html) 函数.

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
//sampleEnd
}
```

如果对某些元素变换的结果是 `null`, 你可以将这些 `null` 值从结果集合中过滤掉, 方法是使用 `mapNotNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-not-null.html) 函数代替 `map()` 函数, 或者相应的使用 `mapIndexedNotNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-indexed-not-null.html) 函数代替 `mapIndexed()` 函数.

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.mapNotNull { if ( it == 2) null else it * 3 })
    println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0)
null else value * idx })
```

```
//sampleEnd
}
```

对 map 进行变换时, 有两种选择: 只变换键(key), 不改变值(value), 或者相反. 如果要对键(key)进行指定的变换, 请使用 mapKeys() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-keys.html) 函数; 相应的, 如果要变换值(value), 请使用 mapValues() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-values.html) 函数. 这两个函数使用的变换函数参数都是 map 条目(entry), 因此在变换函数中你可以同时操作键(key)和值(value).

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key11" to 11)
    println(numbersMap.mapKeys { it.key.uppercase() })
    println(numbersMap.mapValues { it.value + it.key.length })
//sampleEnd
}
```

合并(Zipping)

合并(Zipping) 变换, 将两个集合中相同位置的元素合并为 pair. Kotlin 标准库中, 这个操作使用 zip() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/zip.html) 扩展函数实现.

可以对一个集合或数组调用 zip() 函数, 参数是另一个集合(或数组), 返回值是 Pair 对象构成的 List. 接受者对象集合中的元素, 将成为这些 pair 中的第一个元素.

如果两个集合的大小不同, zip() 返回的结果只包含较小的那个集合大小; 较大的集合中的末尾元素不会出现在结果中.

zip() 也可以使用中缀形式调用, 也就是 a zip b.

```
fun main() {
//sampleStart
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")
    println(colors zip animals)

    val twoAnimals = listOf("fox", "bear")
```

```
    println(colors.zip(twoAnimals))
//sampleEnd
}
```

调用 zip() 时也可以使用变换函数, 变换函数接受两个参数: 一个是接受者集合中的元素, 另一个是参数集合中的元素. 这时, 结果 List 中包含的将是, 使用接受者集合和参数集合相同位置的一对元素, 调用变换函数后返回的结果值.

```
fun main() {
//sampleStart
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal -> "The
${animal.replaceFirstChar { it.uppercase() }} is $color"})
//sampleEnd
}
```

如果已有 Pair 构成的 List, 你可以做相反的变换 – *分离(unzipping)* – 它会通过这些 pair 创建两个 list:

- 第一个 list 包含原 List 的每个 Pair 中的第一个元素.
- 第二个 list 包含 Pair 中的第二个元素.

要分离 pair 构成的 list, 请使用 unzip() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/unzip.html) 函数.

```
fun main() {
//sampleStart
    val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3,
"four" to 4)
    println(numberPairs.unzip())
//sampleEnd
}
```

关联(Association)

关联(Association) 变换, 可以使用指定集合的元素以及与各元素对应的值创建 map. 在不同的关联类型中, 原集合的元素可以是结果 map 中的键(key), 也可以是值(value).

基本的关联函数 `associateWith()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/associate-with.html) 会创建一个 `Map`, 原集合的元素成为它的键(key), 值(value)由一个变换函数通过这些元素计算得到. 如果两个元素相等, 那么只有后一个会保留在 map 中.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

如果要把集合元素变换为 map 中的值(value), 请使用 `associateBy()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/associate-by.html) 函数. 它的参数是一个函数, 这个函数根据元素值返回一个键(key). 如果两个元素的 Key 相等, 那么只有后一个会保留在 map 中.

调用 `associateBy()` 时, 也可以指定一个值变换函数.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.associateBy { it.first().uppercaseChar() })
    println(numbers.associateBy(keySelector = {
it.first().uppercaseChar() }, valueTransform = { it.length }))
//sampleEnd
}
```

构建 map 的另一种方法是 `associate()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/associate.html) 函数, 它根据集合元素通过某种方法同时产生键(key)和值(value). 这个函数的参数是一个 lambda 函数, lambda 函数返回一个 `Pair`: 其中包含对应的 map 条目(entry) 的键(key)和值(value).

注意, `associate()` 产生的是临时存在(short-living)的 `Pair` 对象, 可能会影响性能. 因此, 只有性能问题不是很关键, 或者它比其他方式更合理的情况下, 才应该使用 `associate()` 函数.

后一种情况的例子是, 如果需要从集合元素同时产生键(key)和对应的值(value), 那么就应该使用 associate() 函数了.

```
fun main() {
data class FullName (val firstName: String, val lastName: String)

fun parseFullName(fullName: String): FullName {
    val nameParts = fullName.split(" ")
    if (nameParts.size == 2) {
        return FullName(nameParts[0], nameParts[1])
    } else throw Exception("Wrong name format")
}

//sampleStart
    val names = listOf("Alice Adams", "Brian Brown", "Clara
Campbell")
    println(names.associate { name -> parseFullName(name).let {
it.lastName to it.firstName } })
//sampleEnd
}
```

这个示例中, 我们首先对元素调用一个变换函数, 然后根据变换函数结果的属性创建一个 pair.

扁平化(Flattening)

标准库提供了对嵌套集合(nested collection)的元素进行扁平化访问(flat access)的函数, 对于嵌套集合(nested collection)的操作非常便利.

第一个函数是 flatten() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/flatten.html). 可以对一个集合的集合调用这个函数, 比如, Set 构成的 List. 这个函数返回单个 List, 其中包含嵌套集合中的所有元素.

```
fun main() {
//sampleStart
    val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1,
2))
    println(numberSets.flatten())
```

```
//sampleEnd
}
```

另一个函数 – flatMap() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/flat-map.html) 提供了一种灵活的方式来处理嵌套集合. 它的参数是一个函数, 负责将集合中的一个元素变换为另一个集合. flatMap() 的结果返回单个 list, 其中包括对原集合各个元素调用变换函数后返回的集合中的所有元素. 因此, flatMap() 的行为等于调用 map() (映射(Mapping)的结果是一个集合) 之后再调用 flatten().

```
data class StringContainer(val values: List<String>)

fun main() {
//sampleStart
    val containers = listOf(
        StringContainer(listOf("one", "two", "three")),
        StringContainer(listOf("four", "five", "six")),
        StringContainer(listOf("seven", "eight"))
    )
    println(containers.flatMap { it.values })
//sampleEnd
}
```

字符串表达(String representation)

如果你需要将集合内容表达为人类可读的格式, 请使用将集合转换为字符串的函数: joinToString() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/join-to-string.html) 和 joinTo() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/join-to.html).

joinToString() 根据指定的参数, 从集合元素创建单个 String. joinTo() 执行同样的功能, 但把结果添加到指定的 Appendable (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-appendable/index.html) 对象中.

如果使用默认参数调用这些函数, 返回的结果与对集合调用 toString() 函数类似: 由各个元素的字符串表达组成的 String, 元素之间以逗号加空格分隔.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
```

```
    println(numbers)
    println(numbers.joinToString())

    val listString = StringBuffer("The list of numbers: ")
    numbers.joinTo(listString)
    println(listString)
//sampleEnd
}
```

如果要创建自定义的字符串表达, 可以指定函数参数 `separator`, `prefix`, 以及 `postfix`. 结果字符串以 `prefix` 开始, 以 `postfix` 结尾. `separator` 会出现在每个元素之后, 最后一个元素除外.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString(separator = " | ", prefix = "start: 
", postfix = ": end"))
//sampleEnd
}
```

对比较大的集合, 你可能需要指定 `limit` – 结果中包含的最大元素个数. 如果集合大小超过 `limit` 值, 所有超过的元素会被替换为 `truncated` 参数指定的值.

```
fun main() {
//sampleStart
    val numbers = (1..100).toList()
    println(numbers.joinToString(limit = 10, truncated = "<...>"))
//sampleEnd
}
```

最后, 如果要控制集合元素本身的字符串表达, 可以指定一个 `transform` 函数.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString { "Element: ${it.uppercase()}"})
//sampleEnd
}
```

过滤(Filtering)集合

最终更新: 2024/09/10

在对集合的处理中, 过滤是最常见的任务之一. Kotlin 中, 过滤条件使用 *判定条件(predicate)* 来表示 – 它是一个 lambda 函数, 接受的参数是集合元素, 返回结果是布尔值: `true` 代表这个元素满足判定条件, `false` 表示不满足判定条件.

标准库提供了一组扩展函数, 你可以只通过一次函数调用就能过滤集合. 这些函数不修改原来的集合, 因此对 可变集合和只读集合 ("集合类型" in "集合(Collection)概述") 都可以使用. 要操作过滤后的结果集合, 你应该将它赋值给一个变量, 或者在过滤之后链式调用其他函数.

使用判定条件进行过滤

最基本的过滤函数是 `filter()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html). 调用这个函数时使用判定条件作为参数, `filter()` 函数会返回集合中满足这个判定条件的元素. 对于 `List` 和 `Set`, 结果集合都是 `List`, 对于 `Map`, 结果集合也是 `Map`.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)

    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) ->
key.endsWith("1") && value > 10}
    println(filteredMap)
//sampleEnd
}
```

在 `filter()` 函数的判定条件中, 只能检查元素的值. 如果在过滤时还想使用元素的位置, 请使用 `filterIndexed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-indexed.html) 函数. 这个函数的判定条件接受两个参数: 第一个是元素下标, 第二个是元素值.

如果要按照相反的条件来过滤集合, 请使用 `filterNot()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-not.html) 函数. 它返回判

定条件结果为 `false` 的元素.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5)  }
    val filteredNot = numbers.filterNot { it.length <= 3 }

    println(filteredIdx)
    println(filteredNot)
//sampleEnd
}
```

还有一些函数, 可以根据元素的类型进行过滤, 得到元素类型缩窄后的集合:

- `filterIsInstance()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-is-instance.html) 函数返回指定类型的集合元素. 对 `List<Any>` 调用这个函数时, `filterIsInstance<T>()` 返回的结果集合类型为 `List<T>`, 因此你可以对结果集合的元素调用类型 `T` 的函数.

  ```
  fun main() {
  //sampleStart
      val numbers = listOf(null, 1, "two", 3.0, "four")
      println("All String elements in upper case:")
      numbers.filterIsInstance<String>().forEach {
          println(it.uppercase())
      }
  //sampleEnd
  }
  ```

- `filterNotNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-not-null.html) 函数返回不为 null 的元素. 对 `List<T?>` 调用这个函数时, `filterNotNull()` 返回的结果集合类型为 `List<T: Any>`, 因此你可以将结果集合的元素作为不为 null 的对象进行处理.

  ```
  fun main() {
  //sampleStart
  ```

```
    val numbers = listOf(null, "one", "two", null)
    numbers.filterNotNull().forEach {
        println(it.length)   // 对于可为 null 的 String, length 属性
是不可访问的
    }
//sampleEnd
}
```

划分(Partition)

另一个过滤函数 – partition() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/partition.html) – 根据一个判定条件过滤集合, 并把不满足判定条件的元素保存到另一个 list 中. 因此从返回值可以得到两个 List 构成的 Pair: 第一个 list 包含满足判定条件的元素, 第二个包含原集合中的所有其他元素.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val (match, rest) = numbers.partition { it.length > 3 }

    println(match)
    println(rest)
//sampleEnd
}
```

验证判定条件

最后, 还有一些函数用来对集合验证某个判定条件:

- any() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/any.html) 函数, 如果至少存在一个元素满足指定的判定条件, 则返回 true.
- none() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/none.html) 函数, 如果不存在任何元素满足指定的判定条件, 则返回 true.
- all() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/all.html) 函数, 如果所有的元素全部满足指定的判定条件, 则返回 true. 注意, 如果对空集合使用任何合法的判定条件调用

`all()`, 会返回 `true`. 这个结果在逻辑学上叫做 *虚空真(vacuous truth)* (*https://en.wikipedia.org/wiki/Vacuous_truth*).

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.any { it.endsWith("e") })
    println(numbers.none { it.endsWith("a") })
    println(numbers.all { it.endsWith("e") })

    println(emptyList<Int>().all { it > 5 })   // vacuous truth
//sampleEnd
}
```

`any()` 和 `none()` 函数也可以不指定判定条件: 这种情况下它们只检查集合是否为空. 如果集合中存在元素, 则 `any()` 返回 `true`, 集合中没有元素, 则 `false`; `none()` 的返回值刚好与此相反.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val empty = emptyList<String>()

    println(numbers.any())
    println(empty.any())

    println(numbers.none())
    println(empty.none())
//sampleEnd
}
```

加法(Plus)和减法(Minus)操作符

最终更新: 2024/09/10

在 Kotlin 中, 也为集合定义了 `加法(Plus)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus.html) (`+`) 和 `减法(Minus)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus.html) (`-`) 操作符. 这些操作符使用一个集合作为第一个操作数; 第二个操作数可以是单个元素, 也可以是另一个集合. 返回值是一个新的只读集合:

- `加法(Plus)` 的返回值包含原来集合中的元素 *和* 第二个操作数的元素.
- `减法(Minus)` 的返回值包含原来集合中的元素, 但要 *除去* 第二个操作数的元素. 如果第二个操作数是单个元素, `减法(Minus)` 只删除原来的集合中 *第一次* 出现的这个元素; 如果第二个操作数是一个集合, 那么原来的集合中 *所有* 出现的这些元素都会被删除.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val plusList = numbers + "five"
    val minusList = numbers - listOf("three", "four")
    println(plusList)
    println(minusList)
//sampleEnd
}
```

关于 map 的 `加法(Plus)` 和 `减法(Minus)` 操作的详情, 请参见 Map 相关操作 (Map 相关操作). 对于集合, 也定义了 计算并赋值操作符 ("计算并赋值" in "操作符重载") `加然后赋值(plusAssign)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus-assign.html) (`+=`) 和 `减然后赋值(minusAssign)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus-assign.html) (`-=`). 但是, 对于只读集合, 这些操作符实际上会使用 `加法(Plus)` 或 `减法(Minus)` 操作符, 然后将结果重新赋值给同一个变量. 因此, 如果集合是只读的, 那么这些操作符只能用于 `var` 类型的变量. 对于可变的集合, 如果是 `val` 类型的变量, 那么这些操作符会修改集合内容. 详情请参见 集合写入操作 (集合写入操作).

分组(Grouping)

最终更新: 2024/09/10

Kotlin 标准库提供了扩展函数, 用于对集合中的元素进行分组操作. 最基本的函数是 groupBy() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/group-by.html), 它接受一个 lambda 函数为参数, 返回结果是一个 Map. 在这个 map 中, 每个键(key)是 lambda 函数的一个返回结果, 与键(key)对应的值(value) 是一个 List, 其中包含返回这个结果的所有元素. 这个函数的用途, 举例来说, 我们可以对一个 String 组成的 list, 按字符串的首字母进行分组.

调用 groupBy() 函数时, 也可以使用另一个 lambda 函数作为第二个参数 – 这个函数负责对值进行变换. 象这样使用两个 lambda 函数调用 groupBy()时, 结果 map 中, 第一个参数(keySelector lambda 函数)负责生成键(key), 它对应的值(value) 则是由第二个参数(值转换 lambda 函数)产生的结果组成的 list, 而不是集合中原来元素组成的 list.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")

    println(numbers.groupBy { it.first().uppercase() })
    println(numbers.groupBy(keySelector = { it.first() },
valueTransform = { it.uppercase() }))
//sampleEnd
}
```

如果你希望对元素分组, 同时对所有的分组结果执行某个操作, 可以使用 groupingBy() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/grouping-by.html) 函数. 这个函数返回一个 Grouping (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-grouping/index.html) 类型的实例. 这个 Grouping 实例可以用来对所有分组结果以 lazy 模式执行操作: 只有在操作执行之前才会真正创建分组结果.

Grouping 支持以下操作:

- eachCount() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/each-count.html) 函数, 计算每个分组结果中的元素个数.
- fold() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold.html) 和 reduce() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce.html) 函

数, 将每个分组结果作为独立的集合, 执行 折叠(fold) 与 简化(reduce) ("折叠(fold) 与 简化(reduce)" in "聚合(Aggregate)操作") 操作, 并返回结果.

- `aggregate()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/aggregate.html) 函数, 对每个分组结果中的所有元素反复执行指定的操作, 并返回最后结果. 这是对 `Grouping` 执行任意操作的通用方式. 如果 折叠(fold) 与 简化(reduce) 不能满足你的需求, 可以用这种方式实现自定义的操作.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five",
"six")
    println(numbers.groupingBy { it.first() }.eachCount())
//sampleEnd
}
```

获取集合的一部分

最终更新: 2024/09/10

Kotlin 标准库包含扩展函数可以用来截取集合中的一部分. 这些函数提供了几种不同的方式来选择结果集合中的元素: 明确指定元素的下标, 或指定结果集合大小, 以及其他方法.

切片(Slice)

slice() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/slice.html) 函数返回一个 list, 其中包含集合中某些下标的元素. 下标可以通过一个 range (值范围(Range)与数列(Progression)) 或一个整数值的集合来指定.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five",
"six")
    println(numbers.slice(1..3))
    println(numbers.slice(0..4 step 2))
    println(numbers.slice(setOf(3, 5, 0)))
//sampleEnd
}
```

提取(Take) 和 抛弃(Drop)

要从集合头部开始取得指定数量的元素, 可以使用 take() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/take.html) 函数. 要从集合尾部开始取得指定数量的元素, 可以使用 takeLast() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/take-last.html) 函数. 如果参数中指定的期望取得元素数量超过集合大小, 那么这两个函数都会返回整个集合.

如果要从集合头部或尾部开始, 抛弃指定数量的元素, 并取得剩余的元素, 可以使用 drop() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/drop.html) 和 dropLast() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/drop-last.html) 函数.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five",
```

```
"six")
    println(numbers.take(3))
    println(numbers.takeLast(3))
    println(numbers.drop(1))
    println(numbers.dropLast(5))
//sampleEnd
}
```

也可以使用判定条件(predicate)来决定取得或抛弃的元素数量. 这类函数有以下 4 个, 与上面介绍的函数类似:

- `takeWhile()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/take-while.html) 等于使用判定条件的 `take()`: 这个函数从集合头部开始查找, 找到第一个不满足判定条件的元素, 并获取在它之前的所有元素(但不含这个元素). 如果集合的第一个元素不满足判定条件, 那么结果为空.
- `takeLastWhile()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/take-last-while.html) 与 `takeLast()` 类似: 这个函数从集合尾部开始获取一个的满足判定条件的连续区间内的元素, 这个元素区间的第一个元素, 是集合中不满足判定条件的最后一个元素之后的那个元素. 如果集合的最后一个元素不满足判定条件, 那么结果为空;
- `dropWhile()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/drop-while.html) 与 `takeWhile()` 相反: 这个函数从集合头部开始查找, 找到第一个不满足判定条件的元素, 并获取从这个元素到集合末尾的所有元素.
- `dropLastWhile()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/drop-last-while.html) 与 `takeLastWhile()` 相反: 它查找到最后一个不满足判定条件的元素, 并获取从集合头部到这个元素的所有元素.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five",
"six")
    println(numbers.takeWhile { !it.startsWith('f') })
    println(numbers.takeLastWhile { it != "three" })
    println(numbers.dropWhile { it.length == 3 })
    println(numbers.dropLastWhile { it.contains('i') })
```

```
//sampleEnd
}
```

分块(Chunk)

要把一个集合按照指定的大小分解为许多部分, 可以使用 chunked() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/chunked.html) 函数. chunked() 只有一个参数 – 块(chunk)的大小 – 它返回一个 List, 其中的元素又是 List, 这些 List 中包含指定数量的元素. 第一个块从原来集合的第一个元素开始, 包含 size 参数指定个数的元素, 第二个块包含之后的 size 参数指定个数的元素, 依此类推. 最后一个块有可能包含较少的元素.

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.chunked(3))
//sampleEnd
}
```

也可以立即对返回的块执行变换(transformation)操作. 这时需要通过 lambda 函数的方式指定变换操作, 并将这个 lambda 函数作为参数传递给 chunked() 函数. lambda 函数本身收到的参数是集合中的一个块. 当使用变换操作参数来调用 chunked() 函数时, 各个块都会是临时存在(short-living) 的 List, 应该在 lambda 函数内读取它的内容.

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.chunked(3) { it.sum() })  // `it` 是原始集合中的一
个块
//sampleEnd
}
```

滑动窗口(Window)

你可以指定一个大小, 然后得到集合中所有可能存在的元素区间(element range). 取得这些元素区间的函数是 windowed() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/windowed.html): 它返回的结果是一个 list, 其中包含的内容是, 如果通过一个指定大小的滑动窗口(sliding window)来观察某个集合时, 所能看到的所有可能的元素区间(element range). 与 chunked() 不同, windowed() 函数

返回以集合中 *每个* 元素为起点的元素范围(element range) (也叫做 *窗口(window)*). `windowed()` 函数的返回值, 就是所有这些元素范围组成的 `List`.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.windowed(3))
//sampleEnd
}
```

`windowed()` 通过可选的参数, 可以实现更加灵活的控制:

- `step` 参数指定两个相邻窗口的起始元素之间的距离. 这个参数默认值是 1, 因此返回结果会包含每个元素开始的窗口. 如果将窗口的移动步长增加到 2, 那么结果中的窗口都会从奇数元素开始: 第 1 个元素, 第 3 个元素, 依此类推.
- `partialWindows` 参数, 允许结果中包含那些从集合尾部元素开始, 并且比指定大小更小的窗口. 比如, 如果你取得大小为 3 的窗口, 那么不会创建从最末尾的 2 个元素开始的窗口. 如果打开 `partialWindows` 选项, 那么结果中将会包含 2 个新的 list, 大小分别为 2 和 1.

最后, 可以对返回的元素范围立即执行变换(transformation)操作. 这时需要在调用 `windowed()` 函数时, 通过 lambda 函数指定变换操作.

```
fun main() {
//sampleStart
    val numbers = (1..10).toList()
    println(numbers.windowed(3, step = 2, partialWindows = true))
    println(numbers.windowed(3) { it.sum() })
//sampleEnd
}
```

如果需要创建 2 个元素的窗口, 有一个单独的函数 - `zipWithNext()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/zip-with-next.html). 这个函数会将集合中邻接的元素创建为许多个 pair. 注意, `zipWithNext()` 不是将集合切断为 pair; 它会创建为 *每个* 元素创建 `Pair`, 最末尾元素除外, 因此, 对于 `[1, 2, 3, 4]` 的返回结果是 `[[1, 2], [2, 3], [3, 4]]`, 而不是 `[[1, 2], [3, 4]]`. 调用 `zipWithNext()` 时也可以指定变换操作函数; 这个函数应该接受原集合中的两个元素作为自己的参数.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.zipWithNext())
    println(numbers.zipWithNext() { s1, s2 -> s1.length >
s2.length})
//sampleEnd
}
```

获取集合的单个元素

最终更新: 2024/09/10

Kotlin 集合提供了一组函数, 用来获取集合中的单个元素. 本节介绍的函数同时适用于 list 和 set.

如 List 的定义 (集合(Collection)概述) 所说, list 是元素按顺序存储的集合. 因此, list 的每个元素都有自己的位置下标, 可以通过这个下标来访问元素. 除本节中介绍的函数之外, list 还提供了更多方法, 可以通过下标访问和查找元素. 更多细节, 请参照 List 相关操作 (List 相关操作).

相应的, 根据 Set 的定义 (集合(Collection)概述), set 是没有元素顺序的集合. 但是, Kotlin 的 `Set` 会按照某种顺序保存元素. 元素的保存顺序可以是它们的插入顺序 (`LinkedHashSet` 的情况), 元素的自然排列顺序(`SortedSet` 的情况), 或者其他顺序. set 元素的顺序也可能是不可知的. 这种情况下, 元素仍然会是按照某种顺序排列的, 因此依赖于元素位置的那些函数仍然会返回某种结果. 然而, 除非函数调用者知道所使用的 `Set` 的具体实现, 否则结果是不可预知的, .

根据位置获取元素

要获取某个指定位置的元素, 可以使用 `elementAt()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/element-at.html) 函数. 使用一个整数值参数调用这个函数, 将会获取集合在这个位置上的元素. 集合起始元素的位置是 `0`, 最末尾元素的位置是 `(size - 1)`.

对于没有提供按下标访问能力的集合, 或者我们不知道集合具体类型, 因此无法判断集合是否提供按下标访问能力的情况, `elementAt()` 函数是很便利的. 对于 `List` 的情况, 更符合习惯的方法是使用按下标访问操作符 ("使用下标获取元素" in "List 相关操作") (`get()` 或 `[]`).

```
fun main() {
//sampleStart
    val numbers = linkedSetOf("one", "two", "three", "four", "five")
    println(numbers.elementAt(3))

    val numbersSortedSet = sortedSetOf("one", "two", "three",
"four")
    println(numbersSortedSet.elementAt(0)) // 元素按照字母顺序存储
//sampleEnd
}
```

还有一些便利的别名函数, 用于获取集合的起始元素和末尾元素: first() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first.html) 和 last() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last.html).

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.first())
    println(numbers.last())
//sampleEnd
}
```

为了避免获取不存在的位置上的元素导致的异常, 可以使用 elementAt() 的更安全的变体:

- elementAtOrNull() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/element-at-or-null.html) 函数, 如果指定的位置越出了集合边界之外, 则返回 null.
- elementAtOrElse() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/element-at-or-else.html) 函数, 额外接受一个 lambda 函数作为参数, 这个 lambda 函数负责将 Int 参数变换为集合元素类型的一个实例. 如果指定的位置越出了集合边界之外, elementAtOrElse() 函数会返回使用这个位置值调用 lambda 函数的计算结果.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.elementAtOrNull(5))
    println(numbers.elementAtOrElse(5) { index -> "The value for
index $index is undefined"})
//sampleEnd
}
```

根据条件获取元素

first() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first.html) 和 last() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last.html) 函数还允许你使用一个指定的判定条件(predicate)来搜索集合中满足条件的元素. 如果调用 first() 时指定一个判定条件

来检查集合元素, 你会得到使得判定条件计算结果为 `true` 的第一个元素. 相应的, 如果调用 `last()` 时指定一个判定条件, 会返回满足这个判定条件的最后一个元素.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five",
"six")
    println(numbers.first { it.length > 3 })
    println(numbers.last { it.startsWith("f") })
//sampleEnd
}
```

如果没有元素满足判定条件, 这两个函数会抛出异常. 要避免异常, 请使用 `firstOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first-or-null.html) 和 `lastOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last-or-null.html) 函数: 如果没有找到满足条件的元素, 它们会返回 `null`.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five",
"six")
    println(numbers.firstOrNull { it.length > 6 })
//sampleEnd
}
```

这些函数还有一些别名函数, 如果函数名称更符合你的情况, 你可以使用这些别名函数:

- `find()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/find.html) 等价于 `firstOrNull()`
- `findLast()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/find-last.html) 等价于 `lastOrNull()`

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.find { it % 2 == 0 })
    println(numbers.findLast { it % 2 == 0 })
```

```
//sampleEnd
}
```

使用选择器(selector)获取元素

如果你需要在获取元素之前对集合进行映射, 可以使用 firstNotNullOf() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first-not-null-of.html) 函数. 这个函数组合了两个操作:

- 使用选择器(selector)函数对集合进行映射
- 返回映射结果中的第一个非 null 值

如果映射后的结果集合不包含非 null 元素, firstNotNullOf() 会抛出 NoSuchElementException 异常. 这种情况下如果希望返回 null 结果, 请使用 firstNotNullOfOrNull() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first-not-null-of-or-null.html) 函数.

```
fun main() {
//sampleStart
    val list = listOf<Any>(0, "true", false)
    // 将所有元素转换为字符串, 并返回长度满足条件的第一个元素
    val longEnough = list.firstNotNullOf { item ->
item.toString().takeIf { it.length >= 4 } }
    println(longEnough)
//sampleEnd
}
```

随机获取元素

如果你需要获取集合中的任何一个元素, 可以使用 random() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/random.html) 函数. 调用这个函数时, 可以不带参数, 或者使用 Random (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.random/-random/index.html) 对象作为随机数的产生器.

```
fun main() {
//sampleStart
```

```
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.random())
//sampleEnd
}
```

对于空的集合, random() 函数会抛出异常. 如果想要得到 null 值, 请使用 randomOrNull() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/random-or-null.html) 函数.

检测元素是否存在

要检查集合中是否存在某个元素, 请使用 contains() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/contains.html) 函数. 如果集合中存在一个元素与函数参数相等(equals()), 那么它会返回 true. 也可以用操作符的形式调用 contains() 函数, 方法是使用 in 关键字.

如果要一次性检查多个元素是否存在, 请使用 containsAll() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/contains-all.html) 函数, 它的参数是需要检查的元素的集合.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five",
"six")
    println(numbers.contains("four"))
    println("zero" in numbers)

    println(numbers.containsAll(listOf("four", "two")))
    println(numbers.containsAll(listOf("one", "zero")))
//sampleEnd
}
```

此外, 你还可以使用 isEmpty() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/is-empty.html) 或 isNotEmpty() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/is-not-empty.html) 函数, 检查集合中是否存在任何元素.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five",
```

```
"six")
    println(numbers.isEmpty())
    println(numbers.isNotEmpty())

    val empty = emptyList<String>()
    println(empty.isEmpty())
    println(empty.isNotEmpty())
//sampleEnd
}
```

排序(Ordering)

最终更新: 2024/09/10

对于一些集合类型来说, 元素的排序是非常重要的问题. 比如, 包含相同元素的两个 list, 如果元素顺序不同, 会被认为不相等.

在 Kotlin 中, 对象之间的顺序可以通过几种不同的方式来定义.

首先, 有 *自然(natural)* 排序. 这个概念是指 `Comparable` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparable/index.html) 接口的实现类. 对于这些类来说, 如果不指定其他排序方式, 则默认使用自然顺序.

Kotlin 的大多数内建数据类型都是可比较大小的:

- 数值(Numeric)类型使用数学上的大小顺序: `1` 大于 `0`; `-3.4f` 大于 `-5f`, 等等.
- `Char` 和 `String` 使用 字典顺序(lexicographical order) (https://en.wikipedia.org/wiki/Lexicographical_order): `b` 大于 `a`; `world` 大于 `hello`.

对于用户自定义的类型, 想要定义自然顺序, 需要让这个类型实现 `Comparable` 接口. 因此需要实现 `compareTo()` 函数. `compareTo()` 函数的参数是相同类型的另一个对象, 返回结果是一个整数, 表示两个对象哪个更大:

- 正的整数值表示 `compareTo()` 函数的接受者对象比参数对象大.
- 负的整数值表示 `compareTo()` 函数的接受者对象比参数对象小.
- 0 表示两个对象相等.

下面是一个有排序能力的版本号类, 由 major 和 minor 两部分组成.

```
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int = when {
        this.major != other.major -> this.major compareTo
other.major // 这里是 compareTo() 函数的中缀调用形式
        this.minor != other.minor -> this.minor compareTo
other.minor
        else -> 0
    }
```

```
}

fun main() {
    println(Version(1, 2) > Version(1, 3))
    println(Version(2, 0) > Version(1, 5))
}
```

另一种排序方式称为 *自定义(Custom)* 排序, 你可以对任何类型的实例以任意的方式进行排序. 具体来说, 你可以对不可比较的对象定义顺序, 也可以对可比较的对象定义与自然顺序不同的另一种顺序. 要对一个类型定义自定义顺序, 需要为它创建一个 `Comparator` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparator/index.html). `Comparator` 包含 `compare()` 函数: 它的参数是同一个类的两个实例, 返回一个整数, 代表它们的比较结果. 返回值代表的含义与上面介绍的 `compareTo()` 函数相同.

```
fun main() {
//sampleStart
    val lengthComparator = Comparator { str1: String, str2: String -
> str1.length - str2.length }
    println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))
//sampleEnd
}
```

有了 `lengthComparator`, 我们可以对字符串按照长度来排序, 而不是按照默认的字典顺序排序.

定义 `Comparator` 的一种简便方式是标准库提供的 `compareBy()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.comparisons/compare-by.html) 函数. `compareBy()` 函数的参数是一个 lambda 函数, 这个 lambda 函数负责将一个对象实例变换为一个 `Comparable` 值, 自定义排序的结果就是这个 `Comparable` 值的自然顺序.

使用 `compareBy()` 函数, 前面例子中的字符串长度比较器可以写成下面这样:

```
fun main() {
//sampleStart
    println(listOf("aaa", "bb", "c").sortedWith(compareBy {
it.length }))
//sampleEnd
}
```

Kotlin 集合包提供了用于集合排序的各种函数, 可以使用自然顺序, 自定义顺序, 甚至随机顺序. 本节中, 我们会介绍适用于 只读 ("集合类型" in "集合(Collection)概述") 集合的排序函数. 这些函数的返回结果是一个新集合, 其中包含原集合的元素按照指定顺序排序后的结果. 对 可变 ("集合类型" in "集合(Collection)概述") 集合进行原地(in place)排序的函数, 请参见 List 相关操作 ("排序(Sorting)" in "List 相关操作").

使用自然顺序排序

最基本的 `sorted()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted.html) 和 `sortedDescending()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-descending.html) 函数, 返回新的集合, 其中的元素分别使用自然顺序的正序和逆序排序. 这些函数适用于 `Comparable` 元素组成的集合.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println("Sorted ascending: ${numbers.sorted()}")
    println("Sorted descending: ${numbers.sortedDescending()}")
//sampleEnd
}
```

使用自定义顺序排序

如果要使用自定义顺序排序, 或者对不可比较的对象排序, 可以使用 `sortedBy()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-by.html) 和 `sortedByDescending()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-by-descending.html) 函数. 这些函数的参数是一个选择器函数, 负责将集合元素变换为 `Comparable` 值, 然后再按照这些 `Comparable` 值的自然顺序对集合进行排序.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val sortedNumbers = numbers.sortedBy { it.length }
    println("Sorted by length ascending: $sortedNumbers")
```

```
    val sortedByLast = numbers.sortedByDescending { it.last() }
    println("Sorted by the last letter descending: $sortedByLast")
//sampleEnd
}
```

要对集合排序指定一个自定义顺序, 你可以提供一个自己的 `Comparator`. 为了实现这个目的, 可以调用 `sortedWith()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-with.html) 函数, 并使用你的 `Comparator` 作为参数. 使用这个函数对字符串按照长度排序的示例如下:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println("Sorted by length ascending:
${numbers.sortedWith(compareBy { it.length })}")
//sampleEnd
}
```

逆序集合

可以按照相反的顺序访问集合, 方法是使用 `reversed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reversed.html) 函数.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.reversed())
//sampleEnd
}
```

`reversed()` 函数的返回值是一个新的集合, 其中复制了原集合中的所有元素. 因此, 如果之后改变了原元集合的内容, 不会影响到之前通过 `reversed()` 函数得到的结果.

另一个逆序函数 - `asReversed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/as-reversed.html) - 返回原 List 的一个逆序的视图(view), 因此, 如果原 List 不会改变, 那么这个函数可能比 `reversed()` 函数更轻量, 更适用.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
//sampleEnd
}
```

如果原 List 是可变的, 那么它的所有修改都会影响它的逆序视图, 反过来也是如此.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
    numbers.add("five")
    println(reversedNumbers)
//sampleEnd
}
```

但是, 如果不知道 List 是否可变, 或者原集合根本不是 List, 那么更适用使用 `reversed()` 函数, 因为它的结果是原集合的一个复制, 内容不会随原集合一起改变.

随机排序

最后, 还有一个函数, 它返回一个新的 `List`, 其中的元素按照随机顺序排列 - `shuffled()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/shuffled.html) 函数. 调用时可以不带参数, 或指定一个 `Random` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.random/-random/index.html) 对象作为参数.

```
fun main() {
//sampleStart
     val numbers = listOf("one", "two", "three", "four")
     println(numbers.shuffled())
//sampleEnd
}
```

聚合(Aggregate)操作

最终更新: 2024/09/10

Kotlin 的集合包含一些函数, 用于实现常见的 *聚合(Aggregate)操作* – 也就是根据集合内容返回单个结果的操作. 大多数聚合操作都是大家已经熟悉的, 并与其他语言中的类似操作的工作方式相同:

- `minOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-or-null.html) 和 `maxOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-or-null.html) 函数, 分别返回最小和最大的元素. 对空集合, 这些函数返回 `null`.
- `average()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/average.html) 函数, 返回数值集合中元素的平均值.
- `sum()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum.html) 函数, 返回数值集合中元素的合计值.
- `count()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/count.html) 函数, 返回集合的元素个数.

```
fun main() {
    val numbers = listOf(6, 42, 10, 4)

    println("Count: ${numbers.count()}")
    println("Max: ${numbers.maxOrNull()}")
    println("Min: ${numbers.minOrNull()}")
    println("Average: ${numbers.average()}")
    println("Sum: ${numbers.sum()}")
}
```

还有其他函数, 可以取得最小和最大元素, 但使用指定的选择器(selector)函数, 或自定义的 `Comparator` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparator/index.html):

- `maxByOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-by-or-null.html) 和 `minByOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-by-or-null.html) 函数,

参数是一个选择器(selector)函数, 返回的结果是, 经过选择器(selector)函数计算后的结果值最大或最小的那个元素.

- `maxWithOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-with-or-null.html) 和 `minWithOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-with-or-null.html) 函数, 参数是一个 `Comparator` 对象, 返回的结果是, 根据 `Comparator` 的比较结果判定为最大或最小的那个元素.
- `maxOfOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-of-or-null.html) 和 `minOfOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-of-or-null.html) 函数, 参数是一个选择器(selector)函数, 返回结果是, 选择器函数的结果值中的最大或最小值.
- `maxOfWithOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-of-with-or-null.html) 和 `minOfWithOrNull()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-of-with-or-null.html) 函数, 参数是一个 `Comparator` 对象, 返回的结果是, 选择器函数的结果值中, 根据 `Comparator` 判定的最大或最小值.

这些函数都对空集合返回 return `null`. 还有其他替代函数 – `maxOf` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-of.html), `minOf` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-of.html), `maxOfWith` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-of-with.html), 以及 `minOfWith` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-of-with.html) – 这些函数与上面的各个函数功能相同, 但对空集合会抛出 `NoSuchElementException` 异常.

```
fun main() {
//sampleStart
    val numbers = listOf(5, 42, 10, 4)
    val min3Remainder = numbers.minByOrNull { it % 3 }
    println(min3Remainder)

    val strings = listOf("one", "two", "three", "four")
    val longestString = strings.maxWithOrNull(compareBy { it.length
})
    println(longestString)
```

```
//sampleEnd
}
```

除通常的 sum() 函数外, 还有更高级的求和函数 sumOf() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum-of.html), 它接受一个选择器函数作为参数, 返回结果是对集合所有元素执行这个选择器函数之后的合计结果. 选择器函数可以返回不同的数值类型: Int, Long, Double, UInt, 以及 ULong (对 JVM 平台还支持 BigInteger 和 BigDecimal).

```
fun main() {
//sampleStart
    val numbers = listOf(5, 42, 10, 4)
    println(numbers.sumOf { it * 2 })
    println(numbers.sumOf { it.toDouble() / 2 })
//sampleEnd
}
```

折叠(fold) 与 简化(reduce)

对于更加专门的情况, 可以使用 reduce() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce.html) 和 fold() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold.html) 函数, 它们可以对集合中的元素顺序地执行指定的操作, 然后返回累计结果. 这些操作需要两个参数: 前一次计算的累计值, 以及当前处理中的集合元素.

这两个函数的区别是, fold() 通过参数指定初始值, 并把它用作第一步处理时的累计值, 而 reduce() 的第一步处理, 使用第一个和第二个元素作为操作参数.

```
fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)

    val simpleSum = numbers.reduce { sum, element -> sum + element }
    println(simpleSum)
    val sumDoubled = numbers.fold(0) { sum, element -> sum + element
* 2 }
    println(sumDoubled)
```

```
    // 错误：计算结果中，第一个元素没有被加倍
    //val sumDoubledReduce = numbers.reduce { sum, element -> sum +
element * 2 }
    //println(sumDoubledReduce)
//sampleEnd
}
```

上面的示例演示了它们的区别: 计算元素值加倍之后的合计值时, 我们使用了 `fold()` 函数. 如果将同样的计算函数传递给 `reduce()`, 会得到不同的结果, 因为它在第一步计算时会使用 list 的第一个和第二个元素, 因此第一个元素不会被加倍.

如果要对集合元素以相反的顺序调用处理函数, 可以使用 `reduceRight()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-right.html) 和 `foldRight()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold-right.html) 函数. 它们的工作方式与 `fold()` 和 `reduce()` 函数类似, 但从最末尾的元素开始, 然后继续处理前面的元素. 注意, 如果从右端开始进行折叠或简化操作, 那么计算函数得到的操作参数顺序也会改变: 第一个参数是元素值, 第二个参数是累计值.

```
fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)
    val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum
+ element * 2 }
    println(sumDoubledRight)
//sampleEnd
}
```

执行操作时还可以使用元素下标作为参数. 这时请使用 `reduceIndexed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-indexed.html) 和 `foldIndexed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold-indexed.html) 函数, 操作的第一个参数会是元素下标.

最后, 还有对应的函数, 可以对集合元素从右向左执行这样的操作 - `reduceRightIndexed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-right-indexed.html) 和 `foldRightIndexed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold-right-indexed.html).

```
fun main() {
//sampleStart
```

```
    val numbers = listOf(5, 2, 10, 4)
    val sumEven = numbers.foldIndexed(0) { idx, sum, element -> if
(idx % 2 == 0) sum + element else sum }
    println(sumEven)

    val sumEvenRight = numbers.foldRightIndexed(0) { idx, element,
sum -> if (idx % 2 == 0) sum + element else sum }
    println(sumEvenRight)
//sampleEnd
}
```

对于空的集合, 所有的简化(reduce) 操作都会抛出异常. 如果要得到 null 值, 请使用对应的 *OrNull() 函数:

- reduceOrNull() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-or-null.html)
- reduceRightOrNull() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-right-or-null.html)
- reduceIndexedOrNull() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-indexed-or-null.html)
- reduceRightIndexedOrNull() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-right-indexed-or-null.html)

如果你需要保存累加计算的中间结果值, 可以使用 runningFold() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/running-fold.html) (或者它的别名函数 scan() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/scan.html)) 和 runningReduce() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/running-reduce.html) 函数.

```
fun main() {
//sampleStart
    val numbers = listOf(0, 1, 2, 3, 4, 5)
    val runningReduceSum = numbers.runningReduce { sum, item -> sum
+ item }
```

```
    val runningFoldSum = numbers.runningFold(10) { sum, item -> sum
+ item }
//sampleEnd
    val transform = { index: Int, element: Int -> "N = ${index + 1}:
$element" }

println(runningReduceSum.mapIndexed(transform).joinToString("\n",
"Sum of first N elements with runningReduce:\n"))
    println(runningFoldSum.mapIndexed(transform).joinToString("\n",
"Sum of first N elements with runningFold:\n"))
}
```

如果执行操作时需要使用元素下标作为参数, 请使用 `runningFoldIndexed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/running-fold-indexed.html) 或 `runningReduceIndexed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/running-reduce-indexed.html) 函数.

集合写入操作

最终更新: 2024/09/10

可变集合 ("集合类型" in "集合(Collection)概述") 允许执行改变集合内容的操作, 比如, 可以添加或删除元素. 本节中, 我们会介绍所有 MutableCollection 都支持的共通的写入操作. List 和 Map 的专有的写入操作, 请分别参见 List 相关操作 (List 相关操作) 和 Map 相关操作 (Map 相关操作).

添加元素

要向 list 或 set 添加单个元素, 可以使用 add() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/add.html) 函数. 指定的对象会被添加到集合末尾.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    println(numbers)
//sampleEnd
}
```

addAll() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/add-all.html) 会将参数对象中的所有元素全部添加到 list 或 set. 参数可以是 Iterable, Sequence, 或 Array. 这个函数的接受者类型以及参数类型可以不同, 比如, 你可以将 Set 中的所有元素添加到 List.

如果在 list 上调用 addAll() 函数, 那么它将新元素添加到 list 的顺序, 会与元素在参数对象中的顺序相同. 调用 addAll() 函数时也可以增加一个额外的参数, 用来指定新元素添加的位置. 参数对象中的第一个元素会被添加到这个位置上, 所有其他元素会被添加在它之后, 将函数接受者对象集合中原有的元素移动到后面的位置.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 5, 6)
    numbers.addAll(arrayOf(7, 8))
    println(numbers)
    numbers.addAll(2, setOf(3, 4))
    println(numbers)
```

```
//sampleEnd
}
```

也可以使用 in-place 版本的 加(plus) 操作符 ([加法(Plus)和减法(Minus)操作符](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus-assign.html)) - 加然后赋值 (plusAssign) (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus-assign.html) (`+=`) 来添加元素. 当对可变集合使用 `+=` 操作符时, 会将第二个操作数 (可以是单个元素, 或另一个集合) 添加到集合的末尾.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two")
    numbers += "three"
    println(numbers)
    numbers += listOf("four", "five")
    println(numbers)
//sampleEnd
}
```

删除元素

要从可变集合中删除元素, 可以使用 remove() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/remove.html) 函数. remove() 的参数是元素值, 它会从集合中删除与这个值相等的一个元素.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.remove(3)                    // 删除第一个 `3`
    println(numbers)
    numbers.remove(5)                    // 不会删除任何元素
    println(numbers)
//sampleEnd
}
```

如果要一次性删除多个元素, 可以使用以下函数:

- removeAll() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/remove-all.html) 函数, 删除参数集合中出现的所有元素. 或者, 你也用一个判定条件(predicate)为参数来

调用这个函数; 这种情况下, 会删除所有使得判定条件计算结果为 `true` 的元素.

- `retainAll()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/retain-all.html) 函数, 与 `removeAll()` 相反: 它删除参数集合中出现的元素以外的所有元素. 如果使用判定条件(predicate)为参数, 会保留满足判定条件的元素, 删除其他元素.

- `clear()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/clear.html) 函数, 从 list 中删除所有元素, 使它变成一个空 list.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers)
    numbers.retainAll { it >= 3 }
    println(numbers)
    numbers.clear()
    println(numbers)

    val numbersSet = mutableSetOf("one", "two", "three", "four")
    numbersSet.removeAll(setOf("one", "two"))
    println(numbersSet)
//sampleEnd
}
```

从集合中删除元素的另一种方法是使用 `减然后赋值(minusAssign)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus-assign.html) (`-=`) 操作符 – 也就是 in-place 版的 `减(minus)` ([加法(Plus)和减法(Minus)操作符]()) 操作符. 第二个操作数可以是单个元素, 也可以是另一个集合. 如果第二个操作数是单个元素, `-=` 只从集合中删除 *第一个* 与它相等的元素. 相应的, 如果第二个操作数是集合, 那么这个集合中的 *所有* 元素都会被删除. 比如, 如果 list 包含重复的元素, 那么这些重复元素会被全部删除. 第二个操作数可以包含集合中不存在的元素. 这样的元素不会影响操作结果.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "three", 
"four")
    numbers -= "three"
    println(numbers)
```

```
    numbers -= listOf("four", "five")
    //numbers -= listOf("four")    // 与上面的结果是相同的
    println(numbers)
//sampleEnd
}
```

更新元素

list 和 map 还提供了更新元素的操作. 具体的介绍请参见 List 相关操作 (List 相关操作) 和 Map 相关操作 (Map 相关操作). 对于 set, 更新操作是无意义的, 因为它实际上等于删除一个旧元素, 然后添加一个新元素.

List 相关操作

最终更新: 2024/09/10

List ("List" in "集合(Collection)概述") 是 Kotlin 内建集合中最常用的类型. 基于下标的元素访问, 为 list 提供了很多功能强大的操作.

使用下标获取元素

List 支持所有集合共通的元素获取操作: elementAt(), first(), last(), 以及在 获取集合的单个元素 (获取集合的单个元素) 中介绍的其他操作. List 独有的功能是使用下标访问元素, 因此读取一个元素的最简单方法是使用下标来访问它. 这个功能通过 get() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/get.html) 函数实现, 参数是元素下标, 或者也可以使用更简短的 [index] 语法.

如果 list 大小小于指定的下标, 会抛出一个异常. 另外两个其他函数, 可以避免这类异常:

- getOrElse() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-or-else.html) 允许指定一个函数, 如果下标在集合中不存在, 可以通过这个函数来计算一个默认值.
- getOrNull() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-or-null.html) 返回 null 作为下标不存在时的默认值.

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.get(0))
    println(numbers[0])
    //numbers.get(5)                         // 发生异常!
    println(numbers.getOrNull(5))             // 返回 null
    println(numbers.getOrElse(5, {it}))        // 返回 5
//sampleEnd
}
```

获取 list 的一部分

除了 获取集合的一部分 (获取集合的一部分) 中介绍过的共通操作之外, list 还提供了一个 subList() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/sub-list.html)

函数, 它返回 list 中某个指定的下标范围中的元素构成的视图(view). 因此, 如果原集合中的元素发生变化, 那么在之前创建的子列表中它也会变化, 反过来也是如此.

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.subList(3, 6))
//sampleEnd
}
```

查找元素位置

线性查找(Linear search)

对任何 list, 你可以使用 `indexOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index-of.html) 和 `lastIndexOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last-index-of.html) 函数查找一个元素的位置. 这些函数返回 list 中第一个和最后一个与参数相等的元素的位置. 如果不存在匹配的元素, 这两个函数都返回 `-1`.

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4, 2, 5)
    println(numbers.indexOf(2))
    println(numbers.lastIndexOf(2))
//sampleEnd
}
```

还有另一组函数, 接收的参数是一个判定条件, 并查找满足判定条件的元素:

- `indexOfFirst()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index-of-first.html) 返回满足判定条件的 *第一个元素的下标*, 如果不存在匹配的元素, 则返回 `-1`.
- `indexOfLast()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index-of-last.html) 返回满足判定条件的 *最后一个元素的下标*, 如果不存在匹配的元素, 则返回 `-1`.

```
fun main() {
//sampleStart
```

```
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers.indexOfFirst { it > 2})
    println(numbers.indexOfLast { it % 2 == 1})
//sampleEnd
}
```

在排序的 list 中折半查找(Binary search)

在 list 中查找元素还有另一种方式 – 折半查找(binary search)
(https://en.wikipedia.org/wiki/Binary_search_algorithm). 这种方法的速度要比其他内建函数快很多, 但它 *要求 list 按照升序 排序 (排序(Ordering))*, 排序方法可以是: 自然顺序, 或通过函数参数指定的其它顺序. 否则, 这个函数的查找结果是不确定的.

要在排序的 list 中查找一个元素, 请使用 `binarySearch()`
(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/binary-search.html) 函数, 要查找的元素作为参数. 如果这个元素存在, 这个函数返回它的下标; 否则, 它返回 `(-insertionPoint - 1)`, 其中的 `insertionPoint` 是为了保持 list 正确排序, 这个元素应该插入的下标. 如果存在多个元素等于指定的值, 查找结果可能返回其中任何一个的下标.

也可以指定查找的下标范围: 这种情况下, 这个函数只在指定的两个下标之间进行查找.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.sort()
    println(numbers)
    println(numbers.binarySearch("two"))  // 结果是 3
    println(numbers.binarySearch("z")) // 结果是 -5
    println(numbers.binarySearch("two", 0, 2))  // 结果是 -3
//sampleEnd
}
```

使用比较器(Comparator)进行折半查找(Binary search)

如果 list 元素不是 `Comparable` 对象, 那么在进行折半查找(Binary search)时, 需要提供一个 `Comparator` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparator.html). list 中的元素必须按这个 `Comparator` 比较的结果升序排列. 下面我们来看看示例程序:

```
data class Product(val name: String, val price: Double)

fun main() {
```

```
//sampleStart
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch(Product("AppCode", 99.0),
compareBy<Product> { it.price }.thenBy { it.name }))
//sampleEnd
}
```

这里我们有一个 `Product` 的 list, 其中的 `Product` 对象不是 `Comparable`, 然后我们通过一个 `Comparator` 定义了它们的排序方式: 如果 `p1` 的价格低于 `p2`, 则产品 `p1` 排在 `p2` 之前. 因此, 首先让 list 按照这个规则升序排列, 然后我们使用 `binarySearch()` 来查找指定的 `Product` 的下标.

如果 list 中的元素是 `Comparable` 对象, 但不使用其自然顺序, 比如, 对 `String` 不区分大小写排序的情况, 这时自定义的比较器也是很方便的.

```
fun main() {
//sampleStart
    val colors = listOf("Blue", "green", "ORANGE", "Red", "yellow")
    println(colors.binarySearch("RED",
String.CASE_INSENSITIVE_ORDER)) // 结果是 3
//sampleEnd
}
```

使用比较(Comparison)函数进行折半查找(Binary search)

进行折半查找(Binary search)时, 使用 *比较(Comparison)* 函数, 不必指定确切的查找值即可查找元素. 这种查找方法不需要具体的元素值, 而是接受一个比较函数, 比较函数负责将元素变换为 `Int` 值, 然后查找变换结果为 0 的元素. list 必须按照比较函数规定的升序排序; 也就是说, list 中各个元素传递给比较函数之后的返回值必须是递增的.

```
import kotlin.math.sign
//sampleStart
data class Product(val name: String, val price: Double)

fun priceComparison(product: Product, price: Double) =
sign(product.price - price).toInt()
```

```
fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch { priceComparison(it, 99.0) })
}
//sampleEnd
```

使用比较器(Comparator)和比较(Comparison)函数的折半查找, 也同样可以针对 list 的下标范围进行查找.

List 的写入操作

除了 集合写入操作 (集合写入操作) 中介绍的集合共通的写操作之外, 可变(mutable) ("集合类型" in "集合(Collection)概述") list 还支持 list 独有的写操作. 这类操作使用下标访问元素的方式进行, 增加了 list 的修改能力.

添加元素

要将元素添加到 list 的指定位置, 可以使用 add() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/add.html) 和 addAll() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/add-all.html) 函数, 通过参数指定元素插入的位置. 这个位置之后的所有既有元素, 都会向右移动.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "five", "six")
    numbers.add(1, "two")
    numbers.addAll(2, listOf("three", "four"))
    println(numbers)
//sampleEnd
}
```

更新元素

List 还提供了函数, 可以替换指定位置的元素 - set()
(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/set.html) 函数,
以及相应的操作符 []. set() 函数不会改变其他任何元素的下标.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "five", "three")
    numbers[1] =  "two"
    println(numbers)
//sampleEnd
}
```

fill() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fill.html) 函数会将集合的所有元素简单地替换为指定的值.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.fill(3)
    println(numbers)
//sampleEnd
}
```

删除元素

要从 list 的指定位置删除元素, 可以使用 removeAt()
(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/remove-at.html) 函数, 参数是元素位置. 在这个被删除元素之后的所有其他既有元素, 下标会减少 1.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.removeAt(1)
    println(numbers)
//sampleEnd
}
```

排序(Sorting)

在 集合排序(Ordering) ([排序(Ordering)](#)) 中, 我们介绍了按照指定顺序获取集合元素的操作. 对于可变的 list, 标准库提供了类似的扩展函数, 对 list 原地(in place)执行相同的操作. 如果对一个 list 执行这类操作, 它会改变这个 list 实例中的元素顺序.

原地(in place)排序函数的名称与只读 list 的排序函数类似, 但没有 `ed/d` 后缀:

- 所有排序函数中的 `sorted*` 变为 `sort*`: `sort()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sort.html), `sortDescending()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sort-descending.html), `sortBy()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sort-by.html), 等等.
- `shuffled()` 变为 `shuffle()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/shuffle.html).
- `reversed()` 变为 `reverse()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reverse.html).

对可变 list 调用 `asReversed()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/as-reversed.html) 会返回另一个可变 list, 它是原 list 的一个反序视图(reversed view). 在这个视图中的变更会反映到原 list 中.
下面是可变 list 排序函数的示例:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")

    numbers.sort()
    println("Sort into ascending: $numbers")
    numbers.sortDescending()
    println("Sort into descending: $numbers")

    numbers.sortBy { it.length }
    println("Sort into ascending by length: $numbers")
    numbers.sortByDescending { it.last() }
    println("Sort into descending by the last letter: $numbers")

    numbers.sortWith(compareBy<String> { it.length }.thenBy { it })
    println("Sort by Comparator: $numbers")
```

```
    numbers.shuffle()
    println("Shuffle: $numbers")

    numbers.reverse()
    println("Reverse: $numbers")
//sampleEnd
}
```

Set 相关操作

最终更新: 2024/09/10

Kotlin 的集合包中包含着很多扩展函数, 用来实现常见的 set 操作: 计算两个 set 的交集, 合并两个 set, 或者从一个 set 中减去另一个集合.

要将两个集合合并为一个, 可以使用 `union()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/union.html) 函数. 这个函数可以通过中缀的形式来调用: `a union b`. 注意, 对于有序的集合来说, 参与操作的两个集合的顺序是很重要的. 在结果集合中, 第一个集合的元素出现在第二个集合的元素之前:

```
fun main() {
//sampleStart
    val numbers = setOf("one", "two", "three")

    // 输出结果对应于 set 中的元素顺序
    println(numbers union setOf("four", "five"))
    // 输出结果为 [one, two, three, four, five]
    println(setOf("four", "five") union numbers)
    // 输出结果为 [four, five, one, two, three]
//sampleEnd
}
```

要计算两个集合的交集 (也就是在两个集合中同时出现的元素), 可以使用 `intersect()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/intersect.html) 函数. 要查找出现在一个集合但没有出现在另一个集合的元素, 可以使用 `subtract()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/subtract.html) 函数. 这两个函数也都可以通过中缀的形式来调用, 比如, `a intersect b`:

```
fun main() {
//sampleStart
    val numbers = setOf("one", "two", "three")

    // 输出结果相同
    println(numbers intersect setOf("two", "one"))
    // 输出结果为 [one, two]
    println(numbers subtract setOf("three", "four"))
```

```
    // 输出结果为 [one, two]
    println(numbers subtract setOf("four", "three"))
    // 输出结果为 [one, two]
//sampleEnd
}
```

要计算出现在两个集合中的某一个之内, 但并不同时出现在两个集合之内的元素, 可以使用 `union()` 函数. 要实现这个操作(也叫做对称差异(symmetric difference)), 可以计算两个集合的差异, 然后将差异的计算结果合并起来:

```
fun main() {
//sampleStart
    val numbers = setOf("one", "two", "three")
    val numbers2 = setOf("three", "four")

    // 将两个差异结果合并
    println((numbers - numbers2) union (numbers2 - numbers))
    // 输出结果为 [one, two, four]
//sampleEnd
}
```

`union()`, `intersect()`, 和 `subtract()` 函数 也可以用于 List. 但是, 操作结果 *永远* 是一个 `Set`. 在操作结果中, 所有重复的元素都会被合并为同一个, 并且不能根据下标访问元素:

```
fun main() {
//sampleStart
    val list1 = listOf(1, 1, 2, 3, 5, 8, -1)
    val list2 = listOf(1, 1, 2, 2, 3, 5)

    // 两个 List 的 intersect 结果是一个 Set
    println(list1 intersect list2)
    // 输出结果为 [1, 2, 3, 5]

    // 相等的元素会被合并为一个
    println(list1 union list2)
    // 输出结果为 [1, 2, 3, 5, 8, -1]
```

```
//sampleEnd
}
```

Map 相关操作

最终更新: 2024/09/10

在 map ("Map" in "集合(Collection)概述") 中, 键(key)和值(value)的类型都是用户指定的. 通过键(key)对 map 条目(entry) 的访问, 可以实现各种 Map 相关操作, 比如通过键(key)得到值(value), 以及分别过滤键(key)和值(value). 本节中, 我们介绍标准库提供的 map 操作函数.

取得键(key)和值(value)

要从 map 中取得值(value), 你需要使用键(key)作为参数调用 `get()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/get.html) 函数. 更简短的写法是 `[key]`. 如果未找到指定的键(key), 会返回 `null`. 还有一个函数 `getValue()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-value.html), 它的功能略有不同: 在 map 中未找到键(key)时它会抛出异常. 此外, 还有另外两个选择, 可以对键(key)不存在的情况进行处理:

- `getOrElse()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-or-else.html) 与 list 中的同名函数一样: 对于不存在的键(key), 值(value)由指定的 lambda 函数返回.
- `getOrDefault()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-or-default.html): 如果键(key)不存在, 则返回指定的默认值(value).

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.get("one"))
    println(numbersMap["one"])
    println(numbersMap.getOrDefault("four", 10))
    println(numbersMap["five"])               // 得到 null
    //numbersMap.getValue("six")      // 抛出异常!
//sampleEnd
}
```

如果需要对 map 的所有键(key)或所有值(value)进行操作, 可以分别通过 `keys` 属性和 `values` 属性得到它们. `keys` 是 map 的所有键(key)构成的 set, `values` 是 map 所有值(value)构成的集合.

```kotlin
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.keys)
    println(numbersMap.values)
//sampleEnd
}
```

过滤(Filtering)

可以使用 `filter()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html) 函数和其他函数对 map 进行 过滤(filter) (过滤(Filtering)集合). 对 map 调用 `filter()` 时, 使用的参数是一个判定条件(predicate), 判定条件的参数是一个 `Pair`. 因此可以在过滤的判定条件中同时使用键(key)和值(value).

```kotlin
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) ->
key.endsWith("1") && value > 10}
    println(filteredMap)
//sampleEnd
}
```

还有两种特定的方式来过滤 map: 根据键(key)过滤, 以及根据值(value)过滤. 对每一种方式, 都有一个函数: `filterKeys()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-keys.html) 和 `filterValues()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-values.html). 这两个函数都会返回新的 map, 其中包含满足判定条件的条目(entry). `filterKeys()` 的判定条件只检查元素的键(key), `filterValues()` 的判定条件只检查元素的值(value).

```kotlin
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key11" to 11)
    val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
```

```
    val filteredValuesMap = numbersMap.filterValues { it < 10 }

    println(filteredKeysMap)
    println(filteredValuesMap)
//sampleEnd
}
```

加法(plus) 和 减法(minus) 运算符

由于 map 是通过键(key)访问的, 因此 `加法(plus)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus.html) (`+`) 和 `减法(minus)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus.html) (`-`) 运算符对 map 的工作方式与对其他集合不同. `加法(plus)` 返回一个 `Map`, 其中包含运算符两侧的所有元素: 运算符左侧是一个 `Map`, 右侧是一个 `Pair` 或者另一个 `Map`. 如果运算符右侧的键(key)在左侧的 `Map` 中已经存在, 那么结果 map 包含的是来自右侧的条目(entry).

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap + Pair("four", 4))
    println(numbersMap + Pair("one", 10))
    println(numbersMap + mapOf("five" to 5, "one" to 11))
//sampleEnd
}
```

`减法(minus)` 创建一个 `Map`, 其中包含左侧 `Map` 的条目(entry), 但键(key)出现在右侧的条目(entry)会被排除. 因此, 减法操作符的右侧可以是单个键(key), 也可以是键(key)的集合: list, set, 等等.

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap - "one")
    println(numbersMap - listOf("two", "four"))
//sampleEnd
}
```

对于可变 map 如何使用 `加然后赋值(plusAssign)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus-assign.html) (`+=`) 和 `减然后赋值(minusAssign)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus-assign.html) (`-=`) 操作符, 详情请参见下文的 Map 的写入操作.

Map 的写入操作

可变的 ("集合类型" in "集合(Collection)概述") map 允许执行 map 相关的写入操作. 执行操作允许你使用通过键(key)访问值(value)的方式修改 map 内容.

关于 map 的写入操作, 有一些特定的规则:

- 值(value)可以更新. 相反, 键(key)不能变化: 一旦添加了一个条目(entry), 它的键(key)将会是固定的.
- 对于每个键(key), 永远只有单个的值(value)与它关联. 你可以添加或删除整个条目(entry).

下面是关于可变 map 写入操作的标准库函数的介绍.

增加和更新条目(entry)

要向 map 添加新的 键(key)-值(value) 对, 可以使用 `put()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/put.html) 函数. 向 `LinkedHashMap` (map 的默认实现类) 添加新的条目(entry)时, 它添加的位置会使它在遍历 map 时出现在最后. 对于排序的 map, 新添加元素的位置由它的键(key)的顺序决定.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    println(numbersMap)
//sampleEnd
}
```

如果要一次性添加多个条目(entry), 可以使用 `putAll()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/put-all.html) 函数. 它的参数可以是一个 `Map`, 或一组 `Pair` 对象: `Iterable`, `Sequence`, 或 `Array`.

```
fun main() {
//sampleStart
```

```
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to
3)
    numbersMap.putAll(setOf("four" to 4, "five" to 5))
    println(numbersMap)
//sampleEnd
}
```

如果指定的键(key)已经存在于 map 中, 那么 `put()` 和 `putAll()` 都会覆盖原有的值(value). 因此, 可以使用这些函数来更新 map 条目(entry)中的值(value).

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    val previousValue = numbersMap.put("one", 11)
    println("value associated with 'one', before: $previousValue,
after: ${numbersMap["one"]}")
    println(numbersMap)
//sampleEnd
}
```

也可以使用更简短的操作符形式, 向 map 添加新的条目(entry). 由两种方式:

- `加然后赋值(plusAssign)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus-assign.html) (`+=`) 操作符.
- `[]` 操作符, 它是 `set()` 函数的别名(alias).

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap["three"] = 3     // 会调用 numbersMap.put("three", 3)
    numbersMap += mapOf("four" to 4, "five" to 5)
    println(numbersMap)
//sampleEnd
}
```

如果调用时使用 map 中已存在的键(key), 这些操作符会覆盖对应条目(entry)中的值(value).

删除条目(entry)

要从可变 map 中删除条目(entry), 请使用 `remove()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/remove.html) 函数. 调用 `remove()` 时, 传递的参数可以是键(key), 也可以是整个 键(key)-值(value)-对(pair). 如果同时指定键(key)和值(value), 那么只有在键(key)和值(value)都与参数匹配时, 才会删除对应的元素.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to
3)
    numbersMap.remove("one")
    println(numbersMap)
    numbersMap.remove("three", 4)            // 不会删除任何条目
    println(numbersMap)
//sampleEnd
}
```

也可以使用可变 map 的所有键(key)或所有值(value)来删除条目(entry). 方法是对 map 的 `keys` 或 `values` 属性调用 `remove()` 函数, 参数是想要删除的条目(entry)的键(key)或值(value). 如果是对 `values` 调用 `remove()`, 那么只会删除与指定值(value)匹配的第一个条目(entry).

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to
3, "threeAgain" to 3)
    numbersMap.keys.remove("one")
    println(numbersMap)
    numbersMap.values.remove(3)
    println(numbersMap)
//sampleEnd
}
```

对于可变 map, 还可以使用 `减然后赋值(minusAssign)` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus-assign.html) (`-=`) 操作符.

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to
3)
    numbersMap -= "two"
    println(numbersMap)
    numbersMap -= "five"             // 不会删除任何条目
    println(numbersMap)
//sampleEnd
}
```

明确要求使用者同意的功能(Opt-in Requirement)

最终更新: 2024/09/10

Kotlin 标准库提供了一种机制, 可以要求用户明确同意使用 API 中的某些部分. 对于某些需要使用者明确同意的情况, 库的开发者可以通过这种机制告知他们 API 的使用者, 比如, 如果一个 API 还出在实验性阶段, 未来可能发生变化.

为了防止潜在的问题, 编译器会向这些 API 的使用者提示这些条件的警告信息, 并要求他们同意(Opt-in), 然后才能够使用 API.

同意使用 API

如果库的作者将库中的一个 API 标记为 *要求使用者同意(requiring opt-in)*, 那么你在自己的代码中使用它时, 需要明确表示同意使用. 有几种方法来同意使用这样的 API, 可以使用任何一种方法, 它们都不存在技术上的限制. 你可以自由选择最适合你情况的方法.

传递式同意(Propagating opt-in)

如果你在代码中使用了某个 API, 而你的代码本身又打算给第三方使用(是一个库), 那么你也可以将这个的 API 的要求使用者同意设定传递给你的 API. 为了做到这一点, 需要将你的函数体中使用到的 API 的 *明确要求使用者同意(Opt-in Requirement) 注解* 添加到你的函数的声明部分. 这样你就可以使用要求使用者同意的 API 元素了.

```
// 库代码
@RequiresOptIn(message = "This API is experimental. It may be
changed in the future without notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // 明确要求使用者同意的注解

@MyDateTime
class DateProvider // 一个明确要求使用者同意的类
```

```
// 库的使用者代码
fun getYear(): Int {
    val dateProvider: DateProvider // 编译错误: DateProvider 要求使用者
```

```
同意
    // ...
}

@MyDateTime
fun getDate(): Date {
    val dateProvider: DateProvider // OK: 使用这个类的函数本身也要求使用
者同意
    // ...
}

fun displayDate() {
    println(getDate()) // 编译错误: getDate() 要求使用者同意
}
```

在上面的示例中我们可以看到, 被注解的函数变得象是 `@MyDateTime` API 的一部分. 因此, 这种对使用者同意的强制要求传递到了使用 API 的代码中; 使用这段代码的其他代码也会看到同样的编译警告, 并要求须明确同意使用.

隐含使用要求使用者同意的 API, 本身也需要使用者同意. 如果一个 API 元素没有要求使用者同意注解, 但它的签名包含了要求使用者同意的类型, 那么使用这个 API 仍然会产生警告. 请看下面的例子.

```
// 库的使用者代码
fun getDate(dateProvider: DateProvider): Date { // 编译错误:
DateProvider 要求使用者同意
    // ...
}

fun displayDate() {
    println(getDate()) // 编译警告: getDate() 的签名包含了
DateProvider, 这个类型要求使用者同意
}
```

如果要使用多个要求使用者同意的 API, 请在你的函数声明中分别添加它们的注解.

非传递式同意(Non-propagating opt-in)

在并不对外提供 API 的模块内部, 比如应用程序模块, 你可以同意使用 API, 而不必将这种对使用者同意的强制要求传递到你的代码中. 这种情况下, 请对你的函数声明标注 `@OptIn`

(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-opt-in/) 注解, 并在这个注解的参数中指明要求使用者同意的注解:

```
// 库代码
@RequiresOptIn(message = "This API is experimental. It may be
changed in the future without notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // 明确要求使用者同意的注解

@MyDateTime
class DateProvider // 一个明确要求使用者同意的类
```

```
// 库的使用者代码
@OptIn(MyDateTime::class)
fun getDate(): Date { // 使用 DateProvider 类; 但不会传递对使用者同意的强
制要求
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    println(getDate()) // OK: 不需要使用者同意
}
```

当使用者调用 `getDate()` 函数时, 他们不会由于这个函数内部使用的 API 而被被警告说需要明确同意.

注意, 如果 `@OptIn` 用在函数声明上, 而函数的签名包含了要求使用者同意的类型, 那个对使用者同意的要求仍然会向外传递:

```
// Client code
@OptIn(MyDateTime::class)
fun getDate(dateProvider: DateProvider): Date { // 函数签名包含
DateProvider 类型; 对使用者同意的要求会向外传递
    // ...
}
```

```kotlin
fun displayDate() {
    println(getDate()) // 警告: getDate() 要求使用者同意
}
```

如果想要在一个源代码文件的所有类和所有函数内使用某个要求使用者同意的 API, 可以在文件最前部, 在包声明和包导入语句之前, 添加源代码文件级别的 `@file:OptIn` 注解.

```kotlin
// 库的使用者代码
@file:OptIn(MyDateTime::class)
```

模块范围内同意使用(Module-wide opt-in)

> 从 Kotlin 1.6.0 开始支持编译器选项 `-opt-in`. 对更早的 Kotlin 版本, 请使用 `-Xopt-in`.

如果你不希望在你的代码中每次使用需要同意的 API 的地方都添加标注, 那么你可以在你的整个模块级别上同意使用这些 API. 要在一个模块内同意使用某个 API, 可以使用参数 `-opt-in` 来编译模块, 并指定你所使用的 API 的要求用户同意标注的完全限定名称: `-opt-in=org.mylibrary.OptInAnnotation`. 使用这个参数来编译代码, 效果等于让模块内的每一个声明都添加 `@OptIn(OptInAnnotation::class)` 注解.

如果使用 Gradle 编译模块, 你可以象下面的例子这样来添加参数:

Kotlin

```kotlin
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure
{
    compilerOptions.freeCompilerArgs.add("-opt-
in=org.mylibrary.OptInAnnotation")
}
```

Groovy

```groovy
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...
```

```
tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        freeCompilerArgs.add("-opt-
in=org.mylibrary.OptInAnnotation")
    }
}
```

如果你的 Gradle 模块是一个跨平台模块, 请使用 `optIn` 方法:

Kotlin

```
sourceSets {
    all {
        languageSettings.optIn("org.mylibrary.OptInAnnotation")
    }
}
```

Groovy

```
sourceSets {
    all {
        languageSettings {
            optIn('org.mylibrary.OptInAnnotation')
        }
    }
}
```

对于 Maven, 可以这样添加编译参数:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
```

```xml
                <executions>...</executions>
                <configuration>
                    <args>
                        <arg>-opt-in=org.mylibrary.OptInAnnotation</arg>
                    </args>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

如果要在整个模块级别同意使用多个 API, 请对你的模块中使用到的每个要求用户同意的元素逐个添加上述参数.

对 API 标记要求使用者同意

创建表示要求使用者同意(Opt-in requirement)的注解

如果你希望要求使用者明确同意使用你的模块中的 API, 需要创建一个注解, 用来作为 *要求使用者同意(Opt-in requirement)注解*. 这个注解类本身必须标注 @RequiresOptIn (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-requires-opt-in/) 注解:

```kotlin
@RequiresOptIn
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime
```

要求使用者同意(Opt-in requirement)注解, 必须满足几个要求:

- retention (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-retention/) 设置为 `BINARY` 或 `RUNTIME`
- targets (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/-target/) 中不含 `EXPRESSION`, `FILE`, `TYPE`, 或 `TYPE_PARAMETER`
- 没有参数.

对用户同意要求的严重 级别 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-requires-opt-in/-level/) 可以是以下两种之一:

* `RequiresOptIn.Level.ERROR`. 用户的明确同意是必须的. 否则, t被这个注解标注过的 API, 使用它的代码会编译失败. 默认使用这个严重级别.

* `RequiresOptIn.Level.WARNING`. 用户的明确同意不是必须的, 但建议你明确表示同意. 否则, 编译器会给出警告.

请使用 `@RequiresOptIn` 注解的`level` 参数来设置你希望的严重级别.

此外, 你还可以指定一个 `message` 来提示 API 使用者关于这个 API 的特定条件. 对于使用这个 API 但没有明确同意的用户, 编译器会显示提示这个警告信息.

```
@RequiresOptIn(level = RequiresOptIn.Level.WARNING, message = "This
API is experimental. It can be incompatibly changed in the future.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime
```

如果你需要对外公布多个独立的功能, 都要求用户同意, 那么请为其中的每一个分别定义不同的标注注解. 不同的注解可以使你的代码的使用者在使用这些功能时更加安全: 他们可以只使用他们明确接受的那部分功能. 同时也使你自己能够控制各个功能, 对各个功能分别取消用户同意的强制要求.

标记要求使用者同意的 API 元素

想要将 API 标记为要求使用者同意, 请在它的声明部分添加你定义的那个 opt-in requirement 要求使用者同意注解:

```
@MyDateTime
class DateProvider

@MyDateTime
fun getTime(): Time {}
```

注意, 对于一些语言元素, 不能使用要求使用者同意的注解:

* 对覆盖方法(overriding method)标注的注解只能是这个函数在基类声明中已经出现过的注解.

* 不能用这种注解标注属性的后端域变量(backing field)或取值函数(getter), 只能标注属性本身.

* 不能用这种注解标注局部变量或函数的值参数.

未稳定发布的 API 对使用者同意的要求

如果你对未稳定发布的功能要求使用者同意, 请仔细维护你的 API, 确保不要破坏使用者的代码.

一旦你的未稳定发布的 API 开发完成, 并以稳定模式发布之后, 请在它的声明中删除要求使用者同意的注解. 之后, 使用者的代码就能够不受限制地使用这些 API 了. 但是, 你还需要将这些注解类继续保留在模块中, 以与现有的使用者代码保持兼容.

如果想要让 API 的使用者相应地更新他们的模块(从他们的代码中删除这些注解, 并重新编译), 请将注解标记为 `@Deprecated` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-deprecated/), 并在描述信息中解释原因.

```
@Deprecated("This opt-in requirement is not used anymore. Remove its
usages from your code.")
@RequiresOptIn
annotation class ExperimentalDateTime
```

作用域函数(Scope Function)

最终更新: 2024/09/10

Kotlin 标准库提供了一系列函数, 用来在某个指定的对象上下文中执行一段代码. 你可以对一个对象调用这些函数, 并提供一个 Lambda 表达式 (高阶函数与 Lambda 表达式), 函数会创建一个临时的作用域(scope). 在这个作用域内, 你可以访问这个对象, 而不需要指定名称. 这样的函数称为 *作用域函数(Scope Function)*. 有 5 个这类函数: `let` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/let.html), `run` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/run.html), `with` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/with.html), `apply` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/apply.html), 以及 `also` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/also.html).

基本上, 这些函数都执行同样的操作: 在一个对象上执行一段代码. 它们之间的区别在于, 在代码段内如何访问这个对象, 以及整个表达式的最终结果值是什么.

下面是使用作用域函数的典型例子:

```
data class Person(var name: String, var age: Int, var city: String)
{
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
//sampleStart
    Person("Alice", 20, "Amsterdam").let {
        println(it)
        it.moveTo("London")
        it.incrementAge()
        println(it)
    }
//sampleEnd
}
```

如果不使用 `let` 函数, 为了实现同样的功能, 你就不得不引入一个新的变量, 并在每次用到它的时候使用变量名来访问它.

```kotlin
data class Person(var name: String, var age: Int, var city: String)
{
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
//sampleStart
    val alice = Person("Alice", 20, "Amsterdam")
    println(alice)
    alice.moveTo("London")
    alice.incrementAge()
    println(alice)
//sampleEnd
}
```

作用域函数并没有引入技术上的新功能, 但它能让你的代码变得更简洁易读.

由于作用域函数都很类似, 因此选择一个适合你使用场景的函数会稍微有点难度. 具体的选择取决于你的意图, 以及在你的项目内作用域函数的使用的一致性. 下面我们详细解释各个作用域函数之间的区别, 以及他们的使用惯例.

选择作用域函数

为了帮助你选择适合需要的作用域函数, 我们整理了这张表, 总结这些函数之间的关键区别.

函数	上下文对象的引用方式	返回值	是否扩展函数
`let` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/let.html)	`it`	Lambda 表达式的结果值	是
`run` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/run.html)	`this`	Lambda 表达式的结果值	是
`run` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/run.html)	-	Lambda 表达式的结果值	不是: 不使用上下文对象来调用.
`with` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/with.html)	`this`	Lambda 表达式的结果值	不是: 上下文对象作为参数传递.
`apply` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/apply.html)	`this`	上下文对象本身	是
`also` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/also.html)	`it`	上下文对象本身	是

这些函数的详情会在本章的后续小节中专门介绍.

下面是根据你的需求来选择作用域函数的简短指南:

- 在非 null 对象上执行 Lambda 表达式: `let`
- 在一个局部作用域内引入变量: `let`
- 对一个对象的属性进行设置: `apply`
- 对一个对象的属性进行设置, 并计算结果值: `run`
- 在需要表达式的地方执行多条语句: 非扩展函数形式的 `run`
- 对一个对象进行一些附加处理: `also`
- 对一个对象进行一组函数调用: `with`

不同的作用域函数的使用场景存在重叠, 因此你可以根据你的项目或你的开发组所使用的编码规约来进行选择.

尽管作用域函数可以使得你的代码变得更简洁, 但也要注意不要过度使用: 可能会是你的代码难以阅读, 造成错误. 我们也建议不要嵌套使用作用域函数, 对作用域函数的链式调用要特别小心, 因为很容易导致开发者错误理解当前的上下文对象, 以及 `this` 或 `it` 的值.

作用域函数之间的区别

由于作用域函数很类似, 因此理解它们之间的差别是很重要的. 它们之间主要存在两大差别:

- 它们访问上下文对象的方式.
- 它们的返回值.

访问上下文对象: 使用 this 或 使用 it

在传递给作用域函数的 Lambda 表达式内部, 可以通过一个简短的引用来访问上下文对象, 而不需要使用它的变量名. 每个作用域函数都会使用两种方法之一来引用上下文对象: 作为 Lambda 表达式的 接受者 ("带有接受者的函数字面值" in "高阶函数与 Lambda 表达式") (`this`)来访问, 或者作为 Lambda 表达式的参数(`it`)来访问. 两种方法的功能都是一样的, 因此我们分别介绍这两种方法在不同使用场景下的优点和缺点, 并提供一些使用建议.

```
fun main() {
    val str = "Hello"
    // 使用 this
    str.run {
        println("The string's length: $length")
        //println("The string's length: ${this.length}") // 这种写法的
功能与上面一样
    }

    // 使用 it
    str.let {
        println("The string's length is ${it.length}")
    }
}
```

使用 this

`run`, `with`, 和 `apply` 函数将上下文对象作为 Lambda 表达式的 接受者 ("带有接受者的函数字面值" in "高阶函数与 Lambda 表达式") - 通过 `this` 关键字来访问. 因此, 在这些函数的 Lambda 表达式内, 可以象通常的类函数一样访问到上下文对象.

大多数情况下, 访问接受者对象的成员时, 可以省略 `this` 关键字, 代码可以更简短. 另一方面, 如果省略了 `this`, 阅读代码时会很难区分哪些是接受者的成员, 哪些是外部对象和函数. 因此, 把上下文对象作为接受者(`this`)的方式, 建议用于那些主要对上下文对象成员进行操作的 Lambda 表达式: 调用上下文对象的函数, 或对其属性赋值.

```
data class Person(var name: String, var age: Int = 0, var city:
String = "")

fun main() {
//sampleStart
    val adam = Person("Adam").apply {
        age = 20                       // 等价于 this.age = 20
        city = "London"
    }
    println(adam)
//sampleEnd
}
```

使用 it

`let` 和 `also` 函数使用另一种方式, 它们将上下文对象作为 Lambda 表达式的 参数 ("Lambda 表达式的语法" in "高阶函数与 Lambda 表达式") 来访问. 如果参数名称不指定, 那么上下文对象使用隐含的默认参数名称 `it`. `it` 比 `this` 更短, 而且带 `it` 的表达式通常也更容易阅读.

但是, 你就不能象省略 `this` 那样, 隐含地访问访问对象的函数和属性. 因此, 通过 `it` 访问上下文对象的方式, 比较适合于对象主要被用作函数参数的情况. 如果你的代码段中存在多个变量, `it` 也是更好的选择.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
```

```
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
            writeToLog("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
    println(i)
//sampleEnd
}
```

下面的示例通过有名称的 Lambda 参数 `value` 来访问上下文对象.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also { value ->
            writeToLog("getRandomInt() generated value $value")
        }
    }

    val i = getRandomInt()
    println(i)
//sampleEnd
}
```

返回值

作用域函数的区别还包括它们的返回值:

- `apply` 和 `also` 函数返回作用域对象.
- `let`, `run`, 和 `with` 函数返回 Lambda 表达式的结果值.

你需要根据你的代码之后需要做什么, 来仔细考虑需要什么样的返回值. 这可以帮助你选择最适当的作用域函数.

返回上下文对象

`apply` 和 `also` 的返回值是作用域对象本身. 因此它们可以作为 *旁路(side step)* 成为链式调用的一部分: 你可以在这些函数之后对同一个对象继续调用其他函数.

```
fun main() {
//sampleStart
    val numberList = mutableListOf<Double>()
    numberList.also { println("Populating the list") }
        .apply {
            add(2.71)
            add(3.14)
            add(1.0)
        }
        .also { println("Sorting the list") }
        .sort()
//sampleEnd
    println(numberList)
}
```

还可以用在函数的 return 语句中, 将上下文对象作为函数的返回值.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
            writeToLog("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
```

```
//sampleEnd
}
```

返回 Lambda 表达式的结果值

`let`, `run`, 和 `with` 函数返回 Lambda 表达式的结果值. 因此, 如果需要将 Lambda 表达式结果赋值给一个变量, 或者对 Lambda 表达式结果进行链式操作, 等等, 你可以使用这些函数.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    val countEndsWithE = numbers.run {
        add("four")
        add("five")
        count { it.endsWith("e") }
    }
    println("There are $countEndsWithE elements that end with e.")
//sampleEnd
}
```

此外, 你也可以忽略返回值, 只使用作用域函数来为局部变量创建一个临时的作用域.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        val firstItem = first()
        val lastItem = last()
        println("First item: $firstItem, last item: $lastItem")
    }
//sampleEnd
}
```

函数

为了帮助你选择适当的作用域函数, 我们对各个函数进行详细介绍, 并提供一些使用建议. 技术上来讲, 很多情况下各个作用域函数是可以互换的, 因此这里的示例只演示常见的使用惯例.

let 函数

- **上下文对象** 通过参数 (`it`) 访问.
- **返回值** 是 Lambda 表达式的结果值.

`let` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/let.html) 函数可以用来在链式调用的结果值上调用一个或多个函数. 比如, 下面的代码对一个集合执行两次操作, 然后打印结果:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", 
"five")
    val resultList = numbers.map { it.length }.filter { it > 3 }
    println(resultList)
//sampleEnd
}
```

使用 `let` 函数, 可以改写上面的示例, 使得不必将 List 操作的结果赋值给一个变量:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", 
"five")
    numbers.map { it.length }.filter { it > 3 }.let { 
        println(it)
        // 如果需要, 还可以调用更多函数
    } 
//sampleEnd
}
```

如果传递给 `let` 的 Lambda 表达式的代码段只包含唯一的一个函数调用, 而且使用 `it` 作为这个函数的参数, 那么可以使用方法引用 (`::`) 来代替 Lambda 表达式:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", 
"five")
    numbers.map { it.length }.filter { it > 3 }.let(::println)
```

```
//sampleEnd
}
```

let 经常用来对非 null 值执行一段代码. 如果要对可为 null 的对象进行操作, 请使用 null 值安全的调用操作符 ?. ("安全调用" in "Null 值安全性"), 然后再通过 let 函数, 在 Lambda 表达式内执行这段操作.

```
fun processNonNullString(str: String) {}

fun main() {
//sampleStart
    val str: String? = "Hello"
    //processNonNullString(str)       // 编译错误: str 可能为 null
    val length = str?.let {
        println("let() called on $it")
        processNonNullString(it)      // OK: 在 '?.let { }' 之内可以保
证 'it' 不为 null
        it.length
    }
//sampleEnd
}
```

你也可以使用 let 函数, 在一个比较小的作用域内引入局部变量, 让你的代码更加易读. 为了对上下文对象定义一个新的变量, 请将变量名作为 Lambda 表达式的参数, 然后就可以在 Lambda 表达式使用这个参数名, 而不是默认名称 it.

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val modifiedFirstItem = numbers.first().let { firstItem ->
        println("The first item of the list is '$firstItem'")
        if (firstItem.length >= 5) firstItem else "!" + firstItem +
"!"
    }.uppercase()
    println("First item after modifications: '$modifiedFirstItem'")
//sampleEnd
}
```

with 函数

- **上下文对象** 通过接受者 (`this`) 访问.
- **返回值** 是 Lambda 表达式的结果值.

由于 `with` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/with.html) 不是一个扩展函数: 上下文对象通过参数传递, 但在 Lambda 表达式内部, 可以作为接受者 (`this`) 访问.

我们推荐使用 `with` 函数的情况是, 你可以用它在上下文对象上调用函数, 但不需要使用返回值. 在代码中, `with` 可以被理解为 " *使用这个对象, 进行以下操作.* "

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        println("'with' is called with argument $this")
        println("It contains $size elements")
    }
//sampleEnd
}
```

你也可以使用 `with` 函数, 引入一个辅助对象, 使用它的属性或函数来计算得到一个结果值.

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    val firstAndLast = with(numbers) {
        "The first element is ${first()}," +
        " the last element is ${last()}"
    }
    println(firstAndLast)
//sampleEnd
}
```

run 函数

- **上下文对象** 是接受者 (`this`).

* **返回值** 是 Lambda 表达式的结果值.

`run` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/run.html) 的功能与 `with` 一样, 但它作为扩展函数来实现. 因此和 `let` 一样, 你可以对上下文对象使用点号来调用它.

如果你的 Lambda 表达式既初始化对象, 也计算结果值, 那么就很适合使用 `run` 函数.

```
class MultiportService(var url: String, var port: Int) {
    fun prepareRequest(): String = "Default request"
    fun query(request: String): String = "Result for query
'$request'"
}

fun main() {
//sampleStart
    val service = MultiportService("https://example.kotlinlang.org",
80)

    val result = service.run {
        port = 8080
        query(prepareRequest() + " to port $port")
    }

    // 使用 let() 函数的实现方法是:
    val letResult = service.let {
        it.port = 8080
        it.query(it.prepareRequest() + " to port ${it.port}")
    }
//sampleEnd
    println(result)
    println(letResult)
}
```

你也可以把 `run` 作为非扩展函数来使用. 非扩展函数版本的 `run` 函数没有上下文对象, 但它仍然返回 Lambda 表达式的结果. 通过使用非扩展函数方式的 `run` 函数, 你可以在需要表达式的地方执行多条语句的代码段. 在代码中, 非扩展函数方式的 `run` 函数可以看作是 " *执行这个代码段, 并计算结果* ".

```
fun main() {
//sampleStart
    val hexNumberRegex = run {
        val digits = "0-9"
        val hexDigits = "A-Fa-f"
        val sign = "+-"

        Regex("[$sign]?[$digits$hexDigits]+")
    }

    for (match in hexNumberRegex.findAll("+123 -FFFF !%*& 88 XYZ"))
{
        println(match.value)
    }
//sampleEnd
}
```

apply 函数

- **上下文对象** 是接受者(`this`).
- **返回值** 是对象本身.

由于 `apply` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/apply.html) 会返回上下文对象本身, 因此我们推荐的使用场景是, 代码段没有返回值, 并且主要对接受者对象的成员进行操作. `apply` 函数最常见的使用场景是对象配置. 这样的代码调用可以理解为 " *将以下赋值操作应用于这个对象.* "

```
data class Person(var name: String, var age: Int = 0, var city:
String = "")

fun main() {
//sampleStart
    val adam = Person("Adam").apply {
        age = 32
        city = "London"
    }
    println(adam)
```

```
//sampleEnd
}
```

apply 的另一种使用场景是, 将 apply 函数用作链式调用的一部分, 用来实现复杂的处理.

also 函数

- **上下文对象** 是 Lambda 表达式的参数 (it).
- **返回值** 是对象本身.

also (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/also.html) 函数适合于执行一些将上下文对象作为参数的操作. 如果需要执行一些操作, 其中需要引用对象本身, 而不是它的属性或函数, 或者如果你不希望覆盖更外层作用域(scope)中的 this 引用, 那么就可以使用 also 函数.

如果在代码中看到 also 函数, 可以理解为 " *对这个对象还执行以下操作* ".

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    numbers
        .also { println("The list elements before adding new one:
$it") }
        .add("four")
//sampleEnd
}
```

takeIf 函数和 takeUnless 函数

除作用域函数外, 标准库还提供了 takeIf (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/take-if.html) 函数和 takeUnless (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/take-unless.html) 函数. 这些函数允许你在链式调用中加入对象的状态检查.

如果对一个对象使用一个检查条件调用 takeIf 函数, 在对象满足检查条件时 takeIf 会返回这个对象, 否则返回 null. 因此, takeIf 函数可以作为单个对象的过滤函数.

takeUnless 的逻辑与 takeIf 相反. 如果对一个对象使用一个检查条件调用 takeUnless 函数, 在对象满足检查条件时 takeUnless 会返回 null, 否则返回这个对象.

使用 takeIf 或 takeUnless 时, 在 Lambda 表达式内部, 可以通过参数 (it) 访问到对象.

```
import kotlin.random.*

fun main() {
//sampleStart
    val number = Random.nextInt(100)

    val evenOrNull = number.takeIf { it % 2 == 0 }
    val oddOrNull = number.takeUnless { it % 2 == 0 }
    println("even: $evenOrNull, odd: $oddOrNull")
//sampleEnd
}
```

> ⚠ 如果在 `takeIf` 函数和 `takeUnless` 函数之后链式调用其他函数, 别忘了进行 null 值检查, 或者使用 null 值安全的成员调用(`?.`), 因为它们的返回值是可以为 null 的.

```
fun main() {
//sampleStart
    val str = "Hello"
    val caps = str.takeIf { it.isNotEmpty() }?.uppercase()
   //val caps = str.takeIf { it.isNotEmpty() }.uppercase() // 这里会
出现编译错误
    println(caps)
//sampleEnd
}
```

`takeIf` 函数和 `takeUnless` 函数在与作用域函数组合使用时特别有用. 例如, 你可以将 `takeIf` 和 `takeUnless` 函数与 `let` 函数组合起来, 可以对满足某个条件的对象运行一段代码. 为了实现这个目的, 可以先对这个对象调用 `takeIf` 函数, 然后使用 null 值安全方式(`?.`)来调用 `let` 函数. 对于不满足检查条件的对象, `takeIf` 函数会返回 `null`, 然后 `let` 函数不会被调用.

```
fun main() {
//sampleStart
    fun displaySubstringPosition(input: String, sub: String) {
        input.indexOf(sub).takeIf { it >= 0 }?.let {
            println("The substring $sub is found in $input.")
            println("Its start position is $it.")
```

```
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
//sampleEnd
}
```

如果不使用 `takeIf` 和作用域函数, 同样的功能会写成下面这样, 你可以比较一下两种方式的差别:

```
fun main() {
//sampleStart
    fun displaySubstringPosition(input: String, sub: String) {
        val index = input.indexOf(sub)
        if (index >= 0) {
            println("The substring $sub is found in $input.")
            println("Its start position is $index.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
//sampleEnd
}
```

时间测量

最终更新: 2024/09/10

Kotlin 标准库为你提供了一些工具, 使用不同的单位计算和测量时间. 精确的时间测量对下面这些活动是非常重要的:

- 管理线程或进程
- 收集统计数据
- 检测超时
- 调试

默认情况下, 会使用一个单调时间源(monotonic time source)测量时间, 但也可以配置使用其他时间源. 详情请参见, 创建时间源.

计算持续时间

为了代表一段时间, 标注库提供了 `Duration` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/) 类. 一个 `Duration` 可以使用 `DurationUnit` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration-unit/) 枚举类中的下面这些单位来表达:

- `NANOSECONDS`
- `MICROSECONDS`
- `MILLISECONDS`
- `SECONDS`
- `MINUTES`
- `HOURS`
- `DAYS`

一个 `Duration` 可以是正值, 负值, 0, 正无穷, 或负无穷.

创建持续时间

要创建一个 `Duration`, 请使用 `Int`, `Long`, 和 `Double` 类型的 扩展属性 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/#companion-object-properties): `nanoseconds`, `microseconds`, `milliseconds`, `seconds`, `minutes`, `hours`, 和 `days`.

> ⚠ 天表示24小时的时间长度. 不是日历上的天.

示例:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.nanoseconds
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.Duration.Companion.seconds
import kotlin.time.Duration.Companion.minutes
import kotlin.time.Duration.Companion.days

fun main() {
//sampleStart
    val fiveHundredMilliseconds: Duration = 500.milliseconds
    val zeroSeconds: Duration = 0.seconds
    val tenMinutes: Duration = 10.minutes
    val negativeNanosecond: Duration = (-1).nanoseconds
    val infiniteDays: Duration = Double.POSITIVE_INFINITY.days
    val negativeInfiniteDays: Duration =
Double.NEGATIVE_INFINITY.days

    println(fiveHundredMilliseconds) // 输出结果为 500ms
    println(zeroSeconds)             // 输出结果为 0s
    println(tenMinutes)              // 输出结果为 10m
    println(negativeNanosecond)      // 输出结果为 -1ns
    println(infiniteDays)            // 输出结果为 Infinity
    println(negativeInfiniteDays)    // 输出结果为 -Infinity
//sampleEnd
}
```

你也可以对 `Duration` 对象进行基本的算数运算:

```kotlin
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
    val fiveSeconds: Duration = 5.seconds
    val thirtySeconds: Duration = 30.seconds

    println(fiveSeconds + thirtySeconds)
    // 输出结果为 35s
    println(thirtySeconds - fiveSeconds)
    // 输出结果为 25s
    println(fiveSeconds * 2)
    // 输出结果为 10s
    println(thirtySeconds / 2)
    // 输出结果为 15s
    println(thirtySeconds / fiveSeconds)
    // 输出结果为 6.0
    println(-thirtySeconds)
    // 输出结果为 -30s
    println((-thirtySeconds).absoluteValue)
    // 输出结果为 30s
//sampleEnd
}
```

获取字符串表达

得到 `Duration` 的字符串表达形式会非常有用, 你可以用来打印, 序列化, 传输, 或保存.

要得到字符串表达, 请使用 `.toString()` 函数. 默认情况下, 会使用存在的每个单位来报告时间. 例如: `1h 0m 45.677s` 或 `-(6d 5h 5m 28.284s)`

要配置输出, 请使用 `.toString()` 函数, 以你希望的 `DurationUnit` 和小数位数, 作为函数参数:

```kotlin
import kotlin.time.Duration
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.DurationUnit

fun main() {
//sampleStart
```

```
    // 使用秒单位，2 位小数
    println(5887.milliseconds.toString(DurationUnit.SECONDS, 2))
    // 输出结果为 5.89s
//sampleEnd
}
```

要得到 ISO-8601 兼容 (https://en.wikipedia.org/wiki/ISO_8601) 的字符串, 请使用 `toIsoString()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/to-iso-string.html) 函数:

```
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
    println(86420.seconds.toIsoString()) // 输出结果为 PT24H0M20S
//sampleEnd
}
```

转换持续时间

要把你的 `Duration` 转换为不同的 `DurationUnit`, 请使用以下属性:

- `inWholeNanoseconds`
- `inWholeMicroseconds`
- `inWholeSeconds`
- `inWholeMinutes`
- `inWholeHours`
- `inWholeDays`

示例:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.minutes

fun main() {
//sampleStart
```

```
    val thirtyMinutes: Duration = 30.minutes
    println(thirtyMinutes.inWholeSeconds)
    // 输出结果为 1800
//sampleEnd
}
```

或者，你也可以使用下面的扩展函数，以你希望的 `DurationUnit` 作为函数参数:

* `.toInt()`
* `.toDouble()`
* `.toLong()`

示例:

```
import kotlin.time.Duration.Companion.seconds
import kotlin.time.DurationUnit

fun main() {
//sampleStart
    println(270.seconds.toDouble(DurationUnit.MINUTES))
    // 输出结果为 4.5
//sampleEnd
}
```

比较持续时间

要检查 `Duration` 对象是否相等，请使用相等操作符 (`==`):

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.hours
import kotlin.time.Duration.Companion.minutes

fun main() {
//sampleStart
    val thirtyMinutes: Duration = 30.minutes
    val halfHour: Duration = 0.5.hours
    println(thirtyMinutes == halfHour)
    // 输出结果为 true
```

```
//sampleEnd
}
```

要比较 Duration 对象, 请使用比较操作符 (<, >):

```
import kotlin.time.Duration.Companion.microseconds
import kotlin.time.Duration.Companion.nanoseconds

fun main() {
//sampleStart
    println(3000.microseconds < 25000.nanoseconds)
    // 输出结果为 false
//sampleEnd
}
```

将持续时间分解为不同的部分

要将一个 Duration 分解为不同的时间组成部分, 并进行后续的操作, 请使用 toComponents() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-duration/to-components.html) 函数的重载版本. 将你希望执行的后续操作, 以函数或 Lambda 表达式的形式, 作为 toComponents() 函数的参数.

示例:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.minutes

fun main() {
//sampleStart
    val thirtyMinutes: Duration = 30.minutes
    println(thirtyMinutes.toComponents { hours, minutes, _, _ ->
"${hours}h:${minutes}m" })
    // 输出结果为 0h:30m
//sampleEnd
}
```

在上面的示例中, Lambda 表达式使用 hours 和 minutes 作为参数, 另外还有下划线 (_) 对用于未使用的参数 seconds 和 nanoseconds. Lambda 表达式使用 字符串模板 ("字符串模板" in "字符串"), 得到所需要的 hours 和 minutes 的输出格式, 最后返回拼接的字符串.

测量时间

为了跟踪时间的流逝, 标准库提供了工具, 以便你可以轻松的完成以下任务:

- 使用你希望的时间单位, 测量执行某些代码所需的时间.
- 标记一个时刻.
- 比较两个时刻, 并计算它们之间的差异.
- 检查从某个特定的时刻开始, 经过了多少时间.
- 检查当前时间是否已经经过了某个指定的时刻.

测量代码的执行时间

要测量执行一段代码消耗的时间, 请使用内联函数 `measureTime` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/measure-time.html):

```
import kotlin.time.measureTime

fun main() {
//sampleStart
    val timeTaken = measureTime {
        Thread.sleep(100)
    }
    println(timeTaken) // 例如 103 ms
//sampleEnd
}
```

要测量执行一段代码消耗的时间, **并且** 返回这段代码的执行结果, 请使用内联函数 `measureTimedValue` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/measure-time.html).

示例:

```
import kotlin.time.measureTimedValue

fun main() {
//sampleStart
```

```
    val (value, timeTaken) = measureTimedValue {
        Thread.sleep(100)
        42
    }
    println(value)     // 输出结果为 42
    println(timeTaken) // 例如 103 ms
//sampleEnd
}
```

默认情况下, 这两个函数使用一个单调时间源(monotonic time source).

标记一个时刻

要标记一个特定的时刻, 请使用 `TimeSource` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-source/) 接口, 和 `markNow()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-source/mark-now.html) 函数来创建一个 `TimeMark` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-mark/):

```
import kotlin.time.*

fun main() {
    val timeSource = TimeSource.Monotonic
    val mark = timeSource.markNow()
}
```

测量时刻之间的差异

要测量来自同一个时间源的 `TimeMarks` 对象之间的差异, 请使用减法操作符 (`-`).

要比较来自同一个时间源的 `TimeMark` 对象, 请使用比较操作符 (`<`, `>`).

示例:

```
import kotlin.time.*

fun main() {
//sampleStart
   val timeSource = TimeSource.Monotonic
   val mark1 = timeSource.markNow()
   Thread.sleep(500) // 睡眠 0.5 秒.
   val mark2 = timeSource.markNow()
```

```
    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        println("Measurement 1.${n + 1}: elapsed1=$elapsed1,
elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }

    println(mark2 > mark1) // 比较结果为 true, 因为 mark2 是在 mark1 之后
捕获的.
    // 输出结果为 true
//sampleEnd
}
```

要检查是否已经经过了某个截止时刻, 或者是否已经到达超时时间, 请使用 `hasPassedNow()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-mark/has-passed-now.html) 和 `hasNotPassedNow()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-mark/has-not-passed-now.html) 扩展函数:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    val fiveSeconds: Duration = 5.seconds
    val mark2 = mark1 + fiveSeconds

    // 还没有经过 5 秒
    println(mark2.hasPassedNow())
    // 输出结果为 false

    // 等待 6 秒
    Thread.sleep(6000)
    println(mark2.hasPassedNow())
```

```
    // 输出结果为 true

//sampleEnd
}
```

时间源

默认情况下, 会使用一个单调时间源(monotonic time source)测量时间. 单调时间源只会向前移动, 不会受系统变化的影响, 比如时区变化. 单调时间的替代方案是流逝的真实时间(elapsed real time), 也叫做挂钟时间(wall-clock time). 流逝的真实时间是相对于另一个时间点来测量的.

各个平台的默认时间源

下表是各个平台的默认单调时间源:

平台	时间源
Kotlin/JVM	`System.nanoTime()`
Kotlin/JS (Node.js)	`process.hrtime()`
Kotlin/JS (browser)	`window.performance.now()` 或 `Date.now()`
Kotlin/Native	`std::chrono::high_resolution_clock` or `std::chrono::steady_clock`

创建时间源

有些情况下, 你可能想要使用不同的时间源. 例如, 在 Android 中, `System.nanoTime()` 在设备活动时才计算时间. 当设备进入深度睡眠时, 它会失去对时间的追踪. 想要在设备深度睡眠时继续追踪时间, 你可以创建一个使用 `SystemClock.elapsedRealtimeNanos()` (https://developer.android.com/reference/android/os/SystemClock#elapsedRealtimeNanos()) 的时间源:

```
object RealtimeMonotonicTimeSource :
AbstractLongTimeSource(DurationUnit.NANOSECONDS) {
    override fun read(): Long = SystemClock.elapsedRealtimeNanos()
}
```

然后你就可以使用你的时间源来进行时间测量:

```
fun main() {
    val elapsed: Duration = RealtimeMonotonicTimeSource.measureTime
{
        Thread.sleep(100)
    }
    println(elapsed) // 例如 103 ms
}
```

关于 `kotlin.time` 包, 更多详情请参见我们的 标准库 API 参考文档 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/).

协程指南

最终更新: 2024/09/10

Kotlin 只在它的标准库中提供了最少量的低层 API, 让其它库来使用协程. 与拥有类似功能的其他语言不同, `async` 和 `await` 在 Kotlin 中不是关键字, 甚至不是标准库的一部分. 而且, Kotlin 的 *挂起函数* 的概念, 为异步操作提供了一种比 future 和 promise 更安全, 更不容易出错的的抽象模型.

`kotlinx.coroutines` 是 JetBrain 公司开发的一个功能强大的协程功能库. 本文档将会详细介绍这个库中包含的很多高层的协程基本操作, 包括 `launch`, `async`, 等等.

本文档将会针对各种不同的主题, 通过一系列示例程序来介绍 `kotlinx.coroutines` 库的各种核心功能.

为了使用协程功能, 以及本文档中的各种示例程序, 你需要添加 `kotlinx-coroutines-core` 依赖项, 详细方法请参见 项目的 README 文件 (https://github.com/Kotlin/kotlinx.coroutines/blob/master/README.md#using-in-your-projects).

章节目录

- 协程的基本概念 (协程的基本概念)
- 实际动手(hands-on)教程: 协程与通道(Channel)简介 (https://play.kotlinlang.org/hands-on/Introduction%20to%20Coroutines%20and%20Channels)
- 取消与超时 (取消与超时)
- 挂起函数(Suspending Function)的组合 (挂起函数(Suspending Function)的组合)
- 协程上下文与派发器(Dispatcher) (协程上下文与派发器(Dispatcher))
- 异步的执行流(Asynchronous Flow) (异步的数据流(Asynchronous Flow))
- 通道(Channel) (通道(Channel))
- 协程的异常处理 (协程的异常处理)
- 共享的可变状态与并发 (共享的可变状态与并发)
- 选择表达式 (实验性功能) (选择表达式(Select expression) (实验性功能))

- 教程: 使用 IntelliJ IDEA 调试协程 (教程 - 使用 IntelliJ IDEA 调试协程)
- 教程: 使用 IntelliJ IDEA 调试 Kotlin 数据流(Flow) (教程 - 使用 IntelliJ IDEA 调试 Kotlin 数据流(Flow))

其他参考文档

- 使用协程进行 UI 编程向导 (https://github.com/Kotlin/kotlinx.coroutines/blob/master/ui/coroutines-guide-ui.md)
- 协程功能设计文档 (KEEP) (https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md)
- kotlinx.coroutines API 完整参考文档 (https://kotlinlang.org/api/kotlinx.coroutines/)
- Android 中协程的最佳实践 (https://developer.android.com/kotlin/coroutines/coroutines-best-practices)
- 关于 Android 中使用 Kotlin 协程和数据流(Flow) 的更多资源 (https://developer.android.com/kotlin/coroutines/additional-resources)

协程的基本概念

最终更新: 2024/09/10

本章我们介绍协程的基本概念.

你的第一个协程

协程 是一段可挂起的计算代码的一个实例. 概念上类似于线程, 它包含一段需要运行的代码, 与其他代码并行工作. 但是, 协程并没有绑定到任何特定的线程上. 它的运行可以在一个线程上挂起, 然后在另一个线程中恢复运行.

协程可以看作是轻量的线程, 但有很多重要的区别, 使得协程的使用与线程非常不同.

下面请运行以下代码, 看看你的第一个协程:

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking { // this: CoroutineScope
    launch { // 启动一个新的协程, 然后继续执行当前程序
        delay(1000L) // 非阻塞, 等待 1 秒 (默认的时间单位是毫秒)
        println("World!") // 等待完成后输出信息
    }
    println("Hello") // 当前一个协程在后台等待时, 主协程继续执行
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-basic-01.kt).

你将看到以下运行结果:

```
Hello
World!
```

我们来分析一下这段代码做了什么.

launch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) 是一个 *协程构建器*. 它启动一个新的协程, 协程会与其他代码并行执行, 其他代码则会继续自己的工作. 所以 `Hello` 会先输出.

delay (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/delay.html) 是一个特殊的 *挂起函数*. 它 *挂起* 协程一段指定的时间. 挂起一个协程不会 *阻塞* 底层的线程, 而是允许其他协程运行, 并使用底层的线程执行它们的代码.

runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 也是一个协程构建器, 它负责联通通常的 `fun main()` 内的非协程的世界 与 `runBlocking { ... }` 括号之内使用协程的代码. IDE 会在 `runBlocking` 的开括号之后会提示 `this: CoroutineScope`.

如果你在这段代码中删除或者忘记了 `runBlocking`, 那么会在 launch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) 调用处发生错误, 因为 `launch` 声明在 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 上:

```
Unresolved reference: launch
```

`runBlocking` 的名称表示, 运行它的线程 (在这个示例中 — 是主线程) 在调用期间之内会被 *阻塞*, 直到 `runBlocking { ... }` 之内的所有协程执行完毕. 你会经常在应用程序的最顶层看到这样使用 `runBlocking`, 而在真正的代码之内则很少如此, 因为线程是代价高昂的资源, 阻塞线程是比较低效的, 我们通常并不希望阻塞线程.

结构化的并发

协程遵循 **结构化的并发** 原则, 意思就是说新的协程只能在一个指定的 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 之内启动, CoroutineScope 界定了协程的生命周期. 上面的示例代码中 runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 建立了相应的作用范围(Scope), 所以前面的示例程序会等待, 直到延迟 1 秒后 `World!` 打印完毕, 然后才会退出.

在真实的应用程序中, 你会启动很多协程. 结构化的并发保证协程不会丢失或泄露. 直到所有子协程结束之前, 外层的作用范围不会结束. 结构化的并发还保证代码中的任何错误都会正确的向外报告, 不会丢失.

代码重构, 抽取函数

下面我们把 `launch { ... }` 之内的代码抽取成一个独立的函数. 如果在 IDE 中对这段代码进行一个 "Extract function" 重构操作, 你会得到一个带 `suspend` 修饰符的新函数. 这就是你的第一个 *挂起函数*. 在协程内部可以象使用普通函数那样使用挂起函数, 但挂起函数与普通函数的不同在于, 它们又可以使用其他挂起函数(比如下面的例子中使用的 `delay` 函数)来 *挂起* 当前协程的运行.

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking { // this: CoroutineScope
    launch { doWorld() }
    println("Hello")
}

// 这是你的第一个挂起函数
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-basic-02.kt).

作用范围(Scope)构建器

除了各种构建器提供的协程作用范围之外, 还可以使用 coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 构建器来自行声明作用范围. 这个构建器可以创建一个新的协程作用范围, 并等待在这个范围内启动的所有子协程运行结束.

runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 和 coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 构建器看起来很类似, 因为它们都会等待自己的代码段以及所有的子任务执行完毕. 主要区别是, runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-

core/kotlinx.coroutines/run-blocking.html) 方法为了等待任务结束, 会 *阻塞* 当前线程, 而 coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 只会挂起协程, 而低层的线程可以被用作其他用途. 由于这种区别, runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 是一个通常的函数, 而 coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 是一个挂起函数.

你可以在任何挂起函数中使用 `coroutineScope`. 比如, 你可以将打印 `Hello` 和 `World` 并发代码移动到一个 `suspend fun doWorld()` 函数之内:

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking {
    doWorld()
}

suspend fun doWorld() = coroutineScope {  // this: CoroutineScope
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-basic-03.kt).

这段代码的输出同样是:

```
Hello
World!
```

作用范围构建器与并发

在任何挂起函数之内, 可以使用 coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 构建器来执行多个并发的操作. 我们在 `doWorld` 挂起函数内启动 2 个并发的协程:

```
import kotlinx.coroutines.*

//sampleStart
// 顺序执行 doWorld, 然后输出 "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}

// 并发执行 2 段代码
suspend fun doWorld() = coroutineScope { // this: CoroutineScope
    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
//sampleEnd
```

> 完整的代码请参见 这里 (https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-basic-04.kt).

`launch { ... }` 之内的 2 段代码会 *并发* 执行, 启动之后 1 秒会先输出 `World 1`, 启动之后 2 秒会输出 `World 2`. 直到 2 段代码都结束之后, `doWorld` 之内的 coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 才会结束, 然后 `doWorld` 函数会返回, 直到这时才会输出 `Done`:

```
Hello
World 1
World 2
Done
```

明确控制的 job

launch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) 协程构建器会返回一个 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) 对象, 它是被启动的协程的管理器, 可以用来明确的等待协程结束. 比如, 你可以等待子协程结束, 然后再输出 "Done":

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch { // 启动一个新的协程, 并保存它的 Job 实例
        delay(1000L)
        println("World!")
    }
    println("Hello")
    job.join() // 等待子协程结束
    println("Done")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-basic-05.kt).

这段代码的输出是:

```
Hello
World!
Done
```

协程是非常轻量的

与 JVM 线程相比, 协程消耗更少的资源. 有些代码使用线程时会耗尽 JVM 的可用内存, 如果用协程来表达, 则不会达到资源上限. 比如, 以下代码启动 50,000 个不同的协程, 每个协程等待 5 秒, 然后打印一个点号('.'), 但只消耗非常少的内存:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(50_000) { // 启动非常多的协程
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

> 完整的代码请参见 这里 (https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-basic-06.kt).

如果你使用线程来实现同样的功能 (删除 `runBlocking`, 将 `launch` 替换为 `thread`, 将 `delay` 替换为 `Thread.sleep`). 那么会消耗非常多的内存. 根据你的操作系统, JDK 版本, 以及程序的设定, 这段程序要么会抛出内存不足(out-of-memory)的错误, 要么会缓慢的启动线程, 以免出现太多并发运行的线程.

教程 - 协程与通道(Channel)

最终更新: 2024/09/10

In this tutorial, you'll learn how to use coroutines in IntelliJ IDEA to perform network requests without blocking the underlying thread or callbacks.

No prior knowledge of coroutines is required, but you're expected to be familiar with basic Kotlin syntax.

You'll learn:

- Why and how to use suspending functions to perform network requests.
- How to send requests concurrently using coroutines.
- How to share information between different coroutines using channels.

For network requests, you'll need the Retrofit (https://square.github.io/retrofit/) library, but the approach shown in this tutorial works similarly for any other libraries that support coroutines.

You can find solutions for all of the tasks on the `solutions` branch of the project's repository (http://github.com/kotlin-hands-on/intro-coroutines).

Before you start

1. Download and install the latest version of IntelliJ IDEA (https://www.jetbrains.com/idea/download/index.html).
2. Clone the project template (http://github.com/kotlin-hands-on/intro-coroutines) by choosing **Get from VCS** on the Welcome screen or selecting **File | New | Project from Version Control**.

   You can also clone it from the command line:

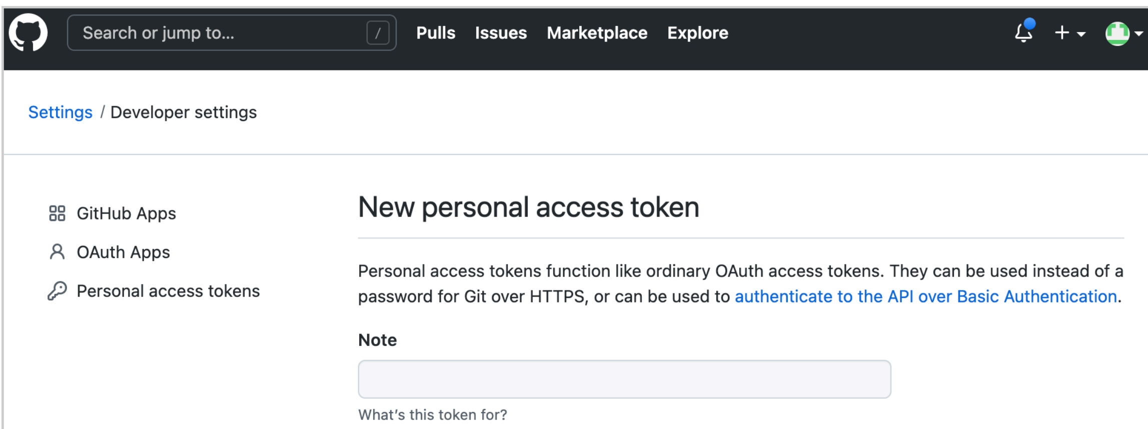
   ```
   git clone https://github.com/kotlin-hands-on/intro-coroutines
   ```

Generate a GitHub developer token

You'll be using the GitHub API in your project. To get access, provide your GitHub account name and either a password or a token. If you have two-factor authentication enabled, a token will be enough.

Generate a new GitHub token to use the GitHub API with your account (https://github.com/settings/tokens/new):

1. Specify the name of your token, for example, `coroutines-tutorial`:



Generate a new GitHub token

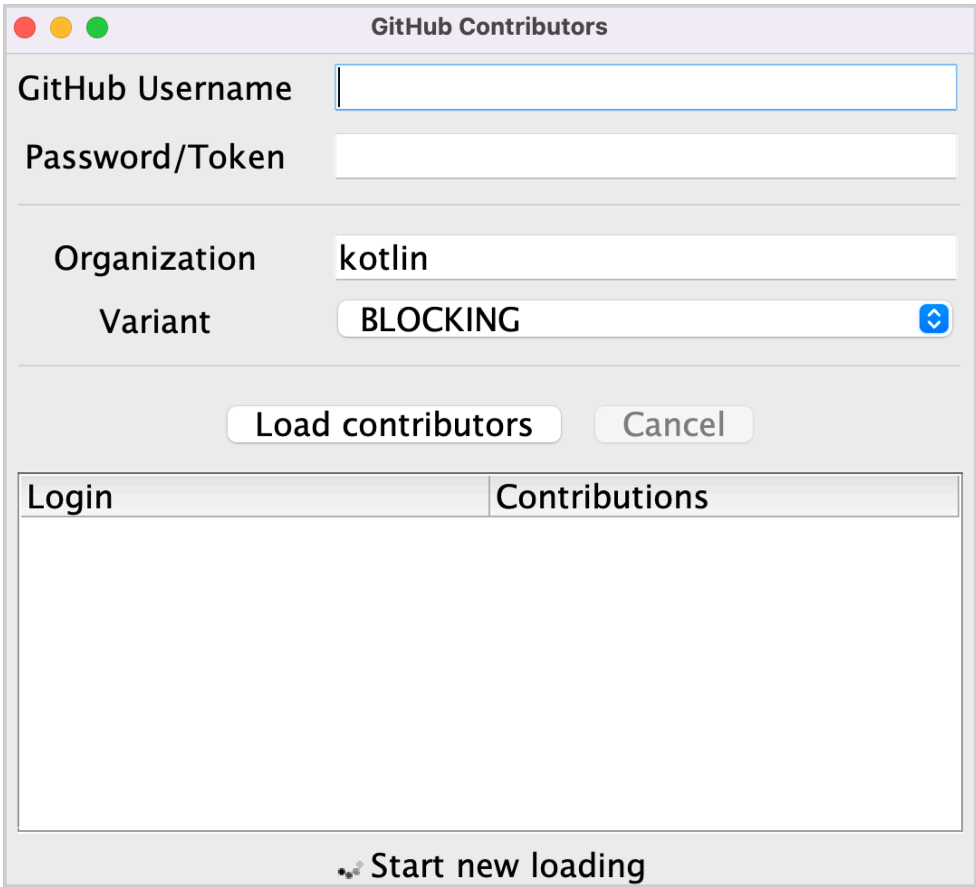
2. Do not select any scopes. Click **Generate token** at the bottom of the page.

3. Copy the generated token.

Run the code

The program loads the contributors for all of the repositories under the given organization (named "kotlin" by default). Later you'll add logic to sort the users by the number of their contributions.

1. Open the `src/contributors/main.kt` file and run the `main()` function. You'll see the following window:



First window

If the font is too small, adjust it by changing the value of `setDefaultFontSize(18f)` in the `main()` function.

2. Provide your GitHub username and token (or password) in the corresponding fields.
3. Make sure that the *BLOCKING* option is selected in the *Variant* dropdown menu.
4. Click *Load contributors*. The UI should freeze for some time and then show the list of contributors.
5. Open the program output to ensure the data has been loaded. The list of contributors is logged after each successful request.

There are different ways of implementing this logic: by using blocking requests or callbacks. You'll compare these solutions with one that uses coroutines and see how channels can be used to share information between different coroutines.

Blocking requests

You will use the Retrofit (https://square.github.io/retrofit/) library to perform HTTP requests to GitHub. It allows requesting the list of repositories under the given organization and the list of contributors for each repository:

```
interface GitHubService {
    @GET("orgs/{org}/repos?per_page=100")
    fun getOrgReposCall(
        @Path("org") org: String
    ): Call<List<Repo>>

    @GET("repos/{owner}/{repo}/contributors?per_page=100")
    fun getRepoContributorsCall(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Call<List<User>>
}
```

This API is used by the `loadContributorsBlocking()` function to fetch the list of contributors for the given organization.

1. Open `src/tasks/Request1Blocking.kt` to see its implementation:

```
fun loadContributorsBlocking(service: GitHubService, req:
RequestData): List<User> {
    val repos = service
        .getOrgReposCall(req.org)   // #1
        .execute()                  // #2
        .also { logRepos(req, it) } // #3
        .body() ?: emptyList()      // #4

    return repos.flatMap { repo ->
        service
            .getRepoContributorsCall(req.org, repo.name) // #1
            .execute()                                   // #2
            .also { logUsers(repo, it) }                 // #3
            .bodyList()                                  // #4
    }.aggregate()
}
```

* At first, you get a list of the repositories under the given organization and store it in the `repos` list. Then for each repository, the list of contributors is requested, and all of the lists are merged into one final list of contributors.
* `getOrgReposCall()` and `getRepoContributorsCall()` both return an instance of the `*Call` class (`#1`). At this point, no request is sent.
* `*Call.execute()` is then invoked to perform the request (`#2`). `execute()` is a synchronous call that blocks the underlying thread.
* When you get the response, the result is logged by calling the specific `logRepos()` and `logUsers()` functions (`#3`). If the HTTP response contains an error, this error will be logged here.
* Finally, get the response's body, which contains the data you need. For this tutorial, you'll use an empty list as a result in case there is an error, and you'll log the corresponding error (`#4`).

2. To avoid repeating `.body() ?: emptyList()`, an extension function `bodyList()` is declared:

```
fun <T> Response<List<T>>.bodyList(): List<T> {
    return body() ?: emptyList()
}
```

3. Run the program again and take a look at the system output in IntelliJ IDEA. It should have something like this:

```
1770 [AWT-EventQueue-0] INFO  Contributors - kotlin: loaded 40
repos
2025 [AWT-EventQueue-0] INFO  Contributors - kotlin-examples:
loaded 23 contributors
2229 [AWT-EventQueue-0] INFO  Contributors - kotlin-koans: loaded
45 contributors
...
```

* The first item on each line is the number of milliseconds that have passed since the program started, then the thread name in square brackets. You can see from which thread the loading request is called.

- The final item on each line is the actual message: how many repositories or contributors were loaded.

  This log output demonstrates that all of the results were logged from the main thread. When you run the code with a *BLOCKING* option, the window freezes and doesn't react to input until the loading is finished. All of the requests are executed from the same thread as the one called `loadContributorsBlocking()` is from, which is the main UI thread (in Swing, it's an AWT event dispatching thread). This main thread becomes blocked, and that's why the UI is frozen:

The blocked main thread

After the list of contributors has loaded, the result is updated.

4. In `src/contributors/Contributors.kt`, find the `loadContributors()` function responsible for choosing how the contributors are loaded and look at how `loadContributorsBlocking()` is called:

```
when (getSelectedVariant()) {
    BLOCKING -> { // Blocking UI thread
        val users = loadContributorsBlocking(service, req)
        updateResults(users, startTime)
    }
}
```

- The `updateResults()` call goes right after the `loadContributorsBlocking()` call.
- `updateResults()` updates the UI, so it must always be called from the UI thread.
- Since `loadContributorsBlocking()` is also called from the UI thread, the UI thread becomes blocked and the UI is frozen.

Task 1

The first task helps you familiarize yourself with the task domain. Currently, each contributor's name is repeated several times, once for every project they have taken part in. Implement the `aggregate()` function combining the users so that each contributor is added only once. The `User.contributions` property should contain the total number of

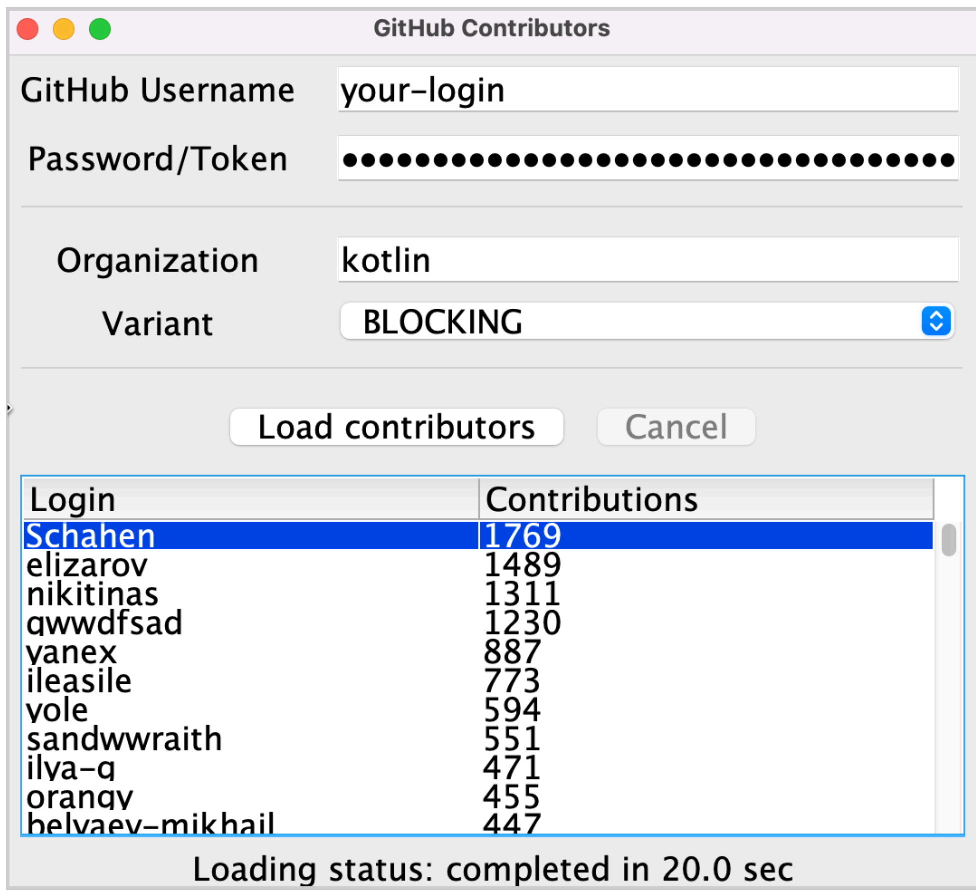
contributions of the given user to *all* the projects. The resulting list should be sorted in descending order according to the number of contributions.

Open `src/tasks/Aggregation.kt` and implement the `List<User>.aggregate()` function. Users should be sorted by the total number of their contributions.

The corresponding test file `test/tasks/AggregationKtTest.kt` shows an example of the expected result.

> You can jump between the source code and the test class automatically by using the IntelliJ IDEA shortcut (https://www.jetbrains.com/help/idea/create-tests.html#test-code-navigation) `Ctrl+Shift+T` / `⇧ ⌘ T`.

After implementing this task, the resulting list for the "kotlin" organization should be similar to the following:



The list for the "kotlin" organization

Solution for task 1

1. To group users by login, use groupBy() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/group-by.html), which returns a map from a login to all occurrences of the user with this login in different repositories.
2. For each map entry, count the total number of contributions for each user and create a new instance of the User class by the given name and total of contributions.
3. Sort the resulting list in descending order:

```
fun List<User>.aggregate(): List<User> =
    groupBy { it.login }
        .map { (login, group) -> User(login, group.sumOf {
it.contributions }) }
        .sortedByDescending { it.contributions }
```

An alternative solution is to use the groupingBy() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/grouping-by.html) function instead of groupBy().

Callbacks

The previous solution works, but it blocks the thread and therefore freezes the UI. A traditional approach that avoids this is to use *callbacks*.

Instead of calling the code that should be invoked right after the operation is completed, you can extract it into a separate callback, often a lambda, and pass that lambda to the caller in order for it to be called later.

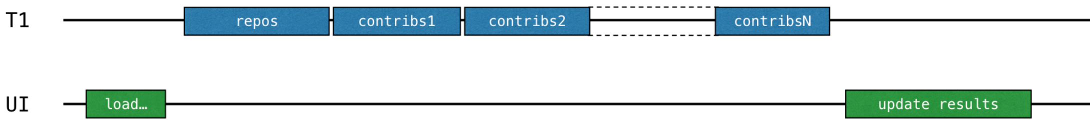
To make the UI responsive, you can either move the whole computation to a separate thread or switch to the Retrofit API which uses callbacks instead of blocking calls.

Use a background thread

1. Open src/tasks/Request2Background.kt and see its implementation. First, the whole computation is moved to a different thread. The thread() function starts a new thread:

```
thread {
    loadContributorsBlocking(service, req)
}
```

Now that all of the loading has been moved to a separate thread, the main thread is free and can be occupied by other tasks:



The freed main thread

2. The signature of the `loadContributorsBackground()` function changes. It takes an `updateResults()` callback as the last argument to call it after all the loading completes:

```
fun loadContributorsBackground(
    service: GitHubService, req: RequestData,
    updateResults: (List<User>) -> Unit
)
```

3. Now when the `loadContributorsBackground()` is called, the `updateResults()` call goes in the callback, not immediately afterward as it did before:

```
loadContributorsBackground(service, req) { users ->
    SwingUtilities.invokeLater {
        updateResults(users, startTime)
    }
}
```

By calling `SwingUtilities.invokeLater`, you ensure that the `updateResults()` call, which updates the results, happens on the main UI thread (AWT event dispatching thread).

However, if you try to load the contributors via the `BACKGROUND` option, you can see that the list is updated but nothing changes.

Task 2

Fix the `loadContributorsBackground()` function in `src/tasks/Request2Background.kt` so that the resulting list is shown in the UI.

Solution for task 2

If you try to load the contributors, you can see in the log that the contributors are loaded but the result isn't displayed. To fix this, call `updateResults()` on the resulting list of users:

```
thread {
    updateResults(loadContributorsBlocking(service, req))
}
```

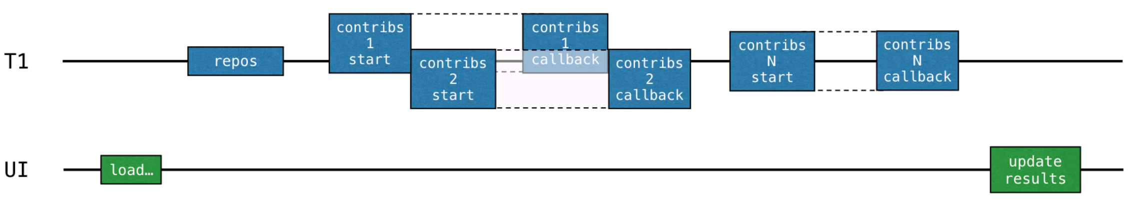
Make sure to call the logic passed in the callback explicitly. Otherwise, nothing will happen.

Use the Retrofit callback API

In the previous solution, the whole loading logic is moved to the background thread, but that still isn't the best use of resources. All of the loading requests go sequentially and the thread is blocked while waiting for the loading result, while it could have been occupied by other tasks. Specifically, the thread could start loading another request to receive the entire result earlier.

Handling the data for each repository should then be divided into two parts: loading and processing the resulting response. The second *processing* part should be extracted into a callback.

The loading for each repository can then be started before the result for the previous repository is received (and the corresponding callback is called):



Using callback API

The Retrofit callback API can help achieve this. The `Call.enqueue()` function starts an HTTP request and takes a callback as an argument. In this callback, you need to specify what needs to be done after each request.

Open `src/tasks/Request3Callbacks.kt` and see the implementation of `loadContributorsCallbacks()` that uses this API:

```
fun loadContributorsCallbacks(
    service: GitHubService, req: RequestData,
    updateResults: (List<User>) -> Unit
) {
    service.getOrgReposCall(req.org).onResponse { responseRepos ->
// #1
```

```
        logRepos(req, responseRepos)
        val repos = responseRepos.bodyList()

        val allUsers = mutableListOf<User>()
        for (repo in repos) {
            service.getRepoContributorsCall(req.org, repo.name)
                .onResponse { responseUsers ->  // #2
                    logUsers(repo, responseUsers)
                    val users = responseUsers.bodyList()
                    allUsers += users
                }
            }
        }
        // TODO: Why doesn't this code work? How to fix that?
        updateResults(allUsers.aggregate())
    }
```

- For convenience, this code fragment uses the `onResponse()` extension function declared in the same file. It takes a lambda as an argument rather than an object expression.
- The logic for handling the responses is extracted into callbacks: the corresponding lambdas start at lines `#1` and `#2`.

However, the provided solution doesn't work. If you run the program and load contributors by choosing the *CALLBACKS* option, you'll see that nothing is shown. However, the tests that immediately return the result pass.

Think about why the given code doesn't work as expected and try to fix it, or see the solutions below.

Task 3 (optional)

Rewrite the code in the `src/tasks/Request3Callbacks.kt` file so that the loaded list of contributors is shown.

The first attempted solution for task 3

In the current solution, many requests are started concurrently, which decreases the total loading time. However, the result isn't loaded. This is because the `updateResults()` callback

is called right after all of the loading requests are started, before the allUsers list has been filled with the data.

You could try to fix this with a change like the following:

```
val allUsers = mutableListOf<User>()
for ((index, repo) in repos.withIndex()) {   // #1
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            logUsers(repo, responseUsers)
            val users = responseUsers.bodyList()
            allUsers += users
            if (index == repos.lastIndex) {    // #2
                updateResults(allUsers.aggregate())
            }
        }
}
```

- First, you iterate over the list of repos with an index (#1).
- Then, from each callback, you check whether it's the last iteration (#2).
- And if that's the case, the result is updated.

However, this code also fails to achieve our objective. Try to find the answer yourself, or see the solution below.

The second attempted solution for task 3

Since the loading requests are started concurrently, there's no guarantee that the result for the last one comes last. The results can come in any order.

Thus, if you compare the current index with the lastIndex as a condition for completion, you risk losing the results for some repos.

If the request that processes the last repo returns faster than some prior requests (which is likely to happen), all of the results for requests that take more time will be lost.

One way to fix this is to introduce an index and check whether all of the repositories have already been processed:

```
val allUsers = Collections.synchronizedList(mutableListOf<User>())
val numberOfProcessed = AtomicInteger()
for (repo in repos) {
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            logUsers(repo, responseUsers)
            val users = responseUsers.bodyList()
            allUsers += users
            if (numberOfProcessed.incrementAndGet() == repos.size) {
                updateResults(allUsers.aggregate())
            }
        }
}
```

This code uses a synchronized version of the list and `AtomicInteger()` because, in general, there's no guarantee that different callbacks that process `getRepoContributors()` requests will always be called from the same thread.

The third attempted solution for task 3

An even better solution is to use the `CountDownLatch` class. It stores a counter initialized with the number of repositories. This counter is decremented after processing each repository. It then waits until the latch is counted down to zero before updating the results:

```
val countDownLatch = CountDownLatch(repos.size)
for (repo in repos) {
    service.getRepoContributorsCall(req.org, repo.name)
        .onResponse { responseUsers ->
            // processing repository
            countDownLatch.countDown()
        }
}
countDownLatch.await()
updateResults(allUsers.aggregate())
```

The result is then updated from the main thread. This is more direct than delegating the logic to the child threads.

After reviewing these three attempts at a solution, you can see that writing correct code with callbacks is non-trivial and error-prone, especially when several underlying threads

and synchronization occur.

As an additional exercise, you can implement the same logic using a reactive approach with the RxJava library. All of the necessary dependencies and solutions for using RxJava can be found in a separate rx branch. It is also possible to complete this tutorial and implement or check the proposed Rx versions for a proper comparison.

Suspending functions

You can implement the same logic using suspending functions. Instead of returning Call<List<Repo>>, define the API call as a suspending function (挂起函数(Suspending Function)的组合) as follows:

```
interface GitHubService {
    @GET("orgs/{org}/repos?per_page=100")
    suspend fun getOrgRepos(
        @Path("org") org: String
    ): List<Repo>
}
```

- getOrgRepos() is defined as a suspend function. When you use a suspending function to perform a request, the underlying thread isn't blocked. More details about how this works will come in later sections.
- getOrgRepos() returns the result directly instead of returning a Call. If the result is unsuccessful, an exception is thrown.

Alternatively, Retrofit allows returning the result wrapped in Response. In this case, the result body is provided, and it is possible to check for errors manually. This tutorial uses the versions that return Response.

In src/contributors/GitHubService.kt, add the following declarations to the GitHubService interface:

```
interface GitHubService {
    // getOrgReposCall & getRepoContributorsCall declarations
```

```
    @GET("orgs/{org}/repos?per_page=100")
    suspend fun getOrgRepos(
        @Path("org") org: String
    ): Response<List<Repo>>

    @GET("repos/{owner}/{repo}/contributors?per_page=100")
    suspend fun getRepoContributors(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Response<List<User>>
}
```

Task 4

Your task is to change the code of the function that loads contributors to make use of two new suspending functions, `getOrgRepos()` and `getRepoContributors()`. The new `loadContributorsSuspend()` function is marked as `suspend` to use the new API.

Suspending functions can't be called everywhere. Calling a suspending function from `loadContributorsBlocking()` will result in an error with the message "Suspend function 'getOrgRepos' should be called only from a coroutine or another suspend function".

1. Copy the implementation of `loadContributorsBlocking()` that is defined in `src/tasks/Request1Blocking.kt` into the `loadContributorsSuspend()` that is defined in `src/tasks/Request4Suspend.kt`.
2. Modify the code so that the new suspending functions are used instead of the ones that return `Call`s.
3. Run the program by choosing the *SUSPEND* option and ensure that the UI is still responsive while the GitHub requests are performed.

Solution for task 4

Replace `.getOrgReposCall(req.org).execute()` with `.getOrgRepos(req.org)` and repeat the same replacement for the second "contributors" request:

```kotlin
suspend fun loadContributorsSuspend(service: GitHubService, req: RequestData): List<User> {
    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    return repos.flatMap { repo ->
        service.getRepoContributors(req.org, repo.name)
            .also { logUsers(repo, it) }
            .bodyList()
    }.aggregate()
}
```

- `loadContributorsSuspend()` should be defined as a `suspend` function.
- You no longer need to call `execute`, which returned the `Response` before, because now the API functions return the `Response` directly. Note that this detail is specific to the Retrofit library. With other libraries, the API will be different, but the concept is the same.

Coroutines

The code with suspending functions looks similar to the "blocking" version. The major difference from the blocking version is that instead of blocking the thread, the coroutine is suspended:

```
block -> suspend
thread -> coroutine
```

Coroutines are often called lightweight threads because you can run code on coroutines, similar to how you run code on threads. The operations that were blocking before (and had to be avoided) can now suspend the coroutine instead.

Starting a new coroutine

If you look at how `loadContributorsSuspend()` is used in `src/contributors/Contributors.kt`, you can see that it's called inside `launch`. `launch` is a library function that takes a lambda as

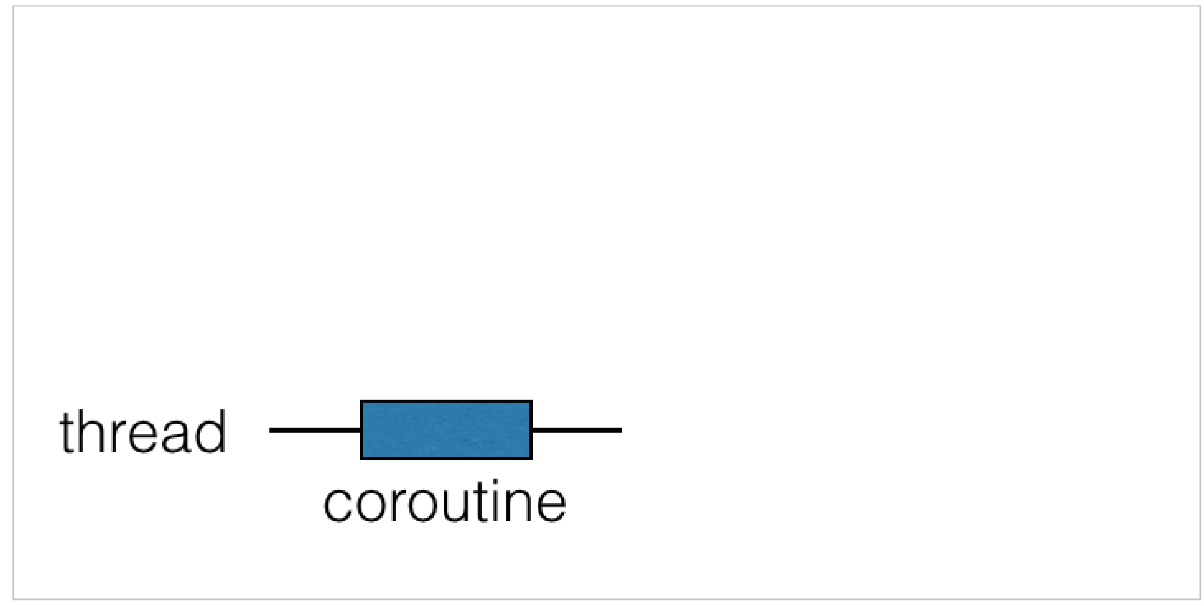
an argument:

```
launch {
    val users = loadContributorsSuspend(req)
    updateResults(users, startTime)
}
```

Here `launch` starts a new computation that is responsible for loading the data and showing the results. The computation is suspendable – when performing network requests, it is suspended and releases the underlying thread. When the network request returns the result, the computation is resumed.

Such a suspendable computation is called a *coroutine*. So, in this case, `launch` *starts a new coroutine* responsible for loading data and showing the results.

Coroutines run on top of threads and can be suspended. When a coroutine is suspended, the corresponding computation is paused, removed from the thread, and stored in memory. Meanwhile, the thread is free to be occupied by other tasks:



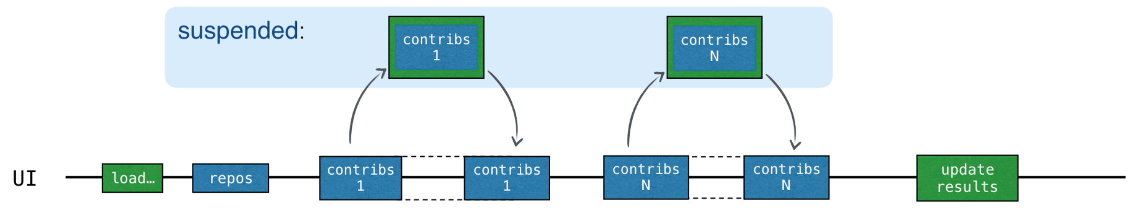
Suspending coroutines

When the computation is ready to be continued, it is returned to a thread (not necessarily the same one).

In the `loadContributorsSuspend()` example, each "contributors" request now waits for the result using the suspension mechanism. First, the new request is sent. Then, while waiting

for the response, the whole "load contributors" coroutine that was started by the `launch` function is suspended.

The coroutine resumes only after the corresponding response is received:



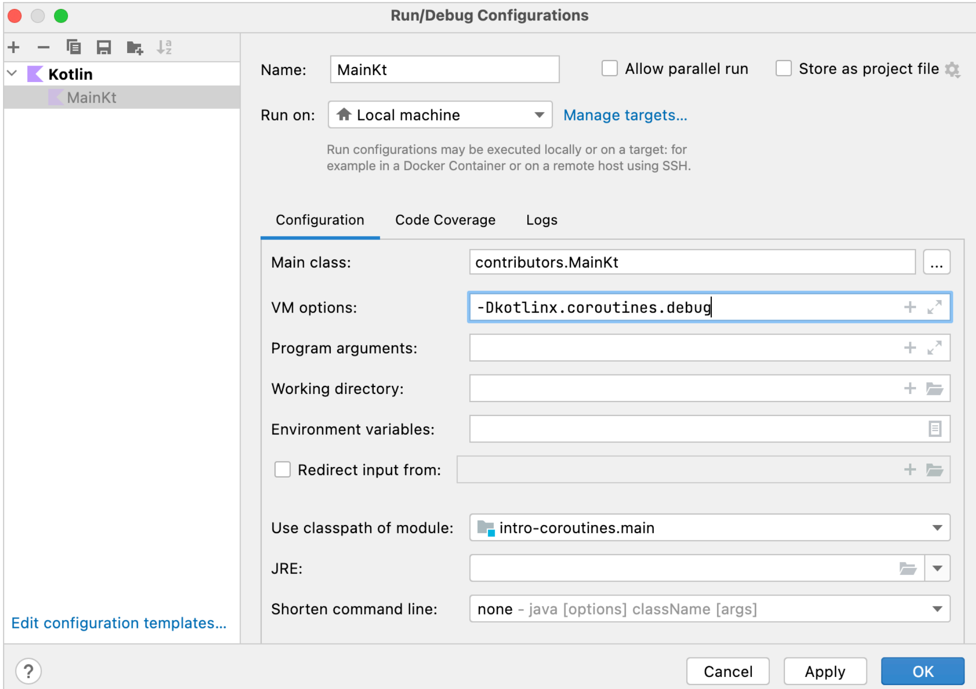
Suspending request

While the response is waiting to be received, the thread is free to be occupied by other tasks. The UI stays responsive, despite all the requests taking place on the main UI thread:

1. Run the program using the *SUSPEND* option. The log confirms that all of the requests are sent to the main UI thread:

   ```
   2538 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin: loaded 30 repos
   2729 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - ts2kt: loaded 11 contributors
   3029 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin-koans: loaded 45 contributors
   ...
   11252 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin-coroutines-workshop: loaded 1 contributors
   ```

2. The log can show you which coroutine the corresponding code is running on. To enable it, open **Run | Edit configurations** and add the `-Dkotlinx.coroutines.debug` VM option:



Edit run configuration

> The coroutine name will be attached to the thread name while `main()` is run with this option. You can also modify the template for running all of the Kotlin files and enable this option by default.

Now all of the code runs on one coroutine, the "load contributors" coroutine mentioned above, denoted as `@coroutine#1`. While waiting for the result, you shouldn't reuse the thread for sending other requests because the code is written sequentially. The new request is sent only when the previous result is received.

Suspending functions treat the thread fairly and don't block it for "waiting". However, this doesn't yet bring any concurrency into the picture.

Concurrency

Kotlin coroutines are much less resource-intensive than threads. Each time you want to start a new computation asynchronously, you can create a new coroutine instead.

To start a new coroutine, use one of the main *coroutine builders*: `launch`, `async`, or `runBlocking`. Different libraries can define additional coroutine builders.

`async` starts a new coroutine and returns a `Deferred` object. `Deferred` represents a concept known by other names such as `Future` or `Promise`. It stores a computation, but it *defers* the moment you get the final result; it *promises* the result sometime in the *future*.

The main difference between `async` and `launch` is that `launch` is used to start a computation that isn't expected to return a specific result. `launch` returns a `Job` that represents the coroutine. It is possible to wait until it completes by calling `Job.join()`.

`Deferred` is a generic type that extends `Job`. An `async` call can return a `Deferred<Int>` or a `Deferred<CustomType>`, depending on what the lambda returns (the last expression inside the lambda is the result).

To get the result of a coroutine, you can call `await()` on the `Deferred` instance. While waiting for the result, the coroutine that this `await()` is called from is suspended:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred: Deferred<Int> = async {
        loadData()
    }
    println("waiting...")
    println(deferred.await())
}

suspend fun loadData(): Int {
    println("loading...")
    delay(1000L)
    println("loaded!")
    return 42
}
```

`runBlocking` is used as a bridge between regular and suspending functions, or between the blocking and non-blocking worlds. It works as an adaptor for starting the top-level main coroutine. It is intended primarily to be used in `main()` functions and tests.

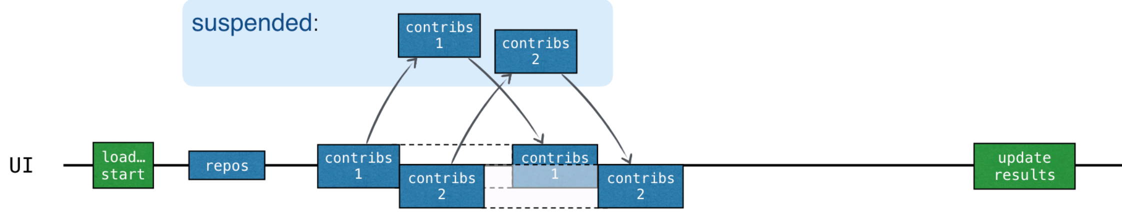
Watch this video (https://www.youtube.com/watch?v=zEZc5AmHQhk) for a better understanding of coroutines.

If there is a list of deferred objects, you can call `awaitAll()` to await the results of all of them:

```
import kotlinx.coroutines.*
```

```
fun main() = runBlocking {
    val deferreds: List<Deferred<Int>> = (1..3).map {
        async {
            delay(1000L * it)
            println("Loading $it")
            it
        }
    }
    val sum = deferreds.awaitAll().sum()
    println("$sum")
}
```

When each "contributors" request is started in a new coroutine, all of the requests are started asynchronously. A new request can be sent before the result for the previous one is received:



Concurrent coroutines

The total loading time is approximately the same as in the *CALLBACKS* version, but it doesn't need any callbacks. What's more, `async` explicitly emphasizes which parts run concurrently in the code.

Task 5

In the `Request5Concurrent.kt` file, implement a `loadContributorsConcurrent()` function by using the previous `loadContributorsSuspend()` function.

Tip for task 5

You can only start a new coroutine inside a coroutine scope. Copy the content from `loadContributorsSuspend()` to the `coroutineScope` call so that you can call `async` functions there:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService,
```

```
    req: RequestData
): List<User> = coroutineScope {
    // ...
}
```

Base your solution on the following scheme:

```
val deferreds: List<Deferred<List<User>>> = repos.map { repo ->
    async {
        // load contributors for each repo
    }
}
deferreds.awaitAll() // List<List<User>>
```

Solution for task 5

Wrap each "contributors" request with `async` to create as many coroutines as there are repositories. `async` returns `Deferred<List<User>>`. This is not an issue because creating new coroutines is not very resource-intensive, so you can create as many as you need.

1. You can no longer use `flatMap` because the `map` result is now a list of `Deferred` objects, not a list of lists. `awaitAll()` returns `List<List<User>>`, so call `flatten().aggregate()` to get the result:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService,
    req: RequestData
): List<User> = coroutineScope {
    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    val deferreds: List<Deferred<List<User>>> = repos.map { repo -
>
        async {
            service.getRepoContributors(req.org, repo.name)
                .also { logUsers(repo, it) }
                .bodyList()
```

```
        }
    }
    deferreds.awaitAll().flatten().aggregate()
}
```

2. Run the code and check the log. All of the coroutines still run on the main UI thread because multithreading hasn't been employed yet, but you can already see the benefits of running coroutines concurrently.

3. To change this code to run "contributors" coroutines on different threads from the common thread pool, specify `Dispatchers.Default` as the context argument for the `async` function:

```
async(Dispatchers.Default) { }
```

- `CoroutineDispatcher` determines what thread or threads the corresponding coroutine should be run on. If you don't specify one as an argument, `async` will use the dispatcher from the outer scope.

- `Dispatchers.Default` represents a shared pool of threads on the JVM. This pool provides a means for parallel execution. It consists of as many threads as there are CPU cores available, but it will still have two threads if there's only one core.

4. Modify the code in the `loadContributorsConcurrent()` function to start new coroutines on different threads from the common thread pool. Also, add additional logging before sending the request:

```
async(Dispatchers.Default) {
    log("starting loading for ${repo.name}")
    service.getRepoContributors(req.org, repo.name)
        .also { logUsers(repo, it) }
        .bodyList()
}
```

5. Run the program once again. In the log, you can see that each coroutine can be started on one thread from the thread pool and resumed on another:

```
1946 [DefaultDispatcher-worker-2 @coroutine#4] INFO  Contributors
- starting loading for kotlin-koans
```

```
1946 [DefaultDispatcher-worker-3 @coroutine#5] INFO  Contributors
- starting loading for dokka
1946 [DefaultDispatcher-worker-1 @coroutine#3] INFO  Contributors
- starting loading for ts2kt
...
2178 [DefaultDispatcher-worker-1 @coroutine#4] INFO  Contributors
- kotlin-koans: loaded 45 contributors
2569 [DefaultDispatcher-worker-1 @coroutine#5] INFO  Contributors
- dokka: loaded 36 contributors
2821 [DefaultDispatcher-worker-2 @coroutine#3] INFO  Contributors
- ts2kt: loaded 11 contributors
```

For instance, in this log excerpt, `coroutine#4` is started on the `worker-2` thread and continued on the `worker-1` thread.

In `src/contributors/Contributors.kt`, check the implementation of the *CONCURRENT* option:

1. To run the coroutine only on the main UI thread, specify `Dispatchers.Main` as an argument:

```
launch(Dispatchers.Main) {
    updateResults()
}
```

- If the main thread is busy when you start a new coroutine on it, the coroutine becomes suspended and scheduled for execution on this thread. The coroutine will only resume when the thread becomes free.
- It's considered good practice to use the dispatcher from the outer scope rather than explicitly specifying it on each end-point. If you define `loadContributorsConcurrent()` without passing `Dispatchers.Default` as an argument, you can call this function in any context: with a `Default` dispatcher, with the main UI thread, or with a custom dispatcher.
- As you'll see later, when calling `loadContributorsConcurrent()` from tests, you can call it in the context with `TestDispatcher`, which simplifies testing. That makes this solution much more flexible.

2. To specify the dispatcher on the caller side, apply the following change to the project while letting `loadContributorsConcurrent` start coroutines in the inherited context:

   ```
   launch(Dispatchers.Default) {
       val users = loadContributorsConcurrent(service, req)
       withContext(Dispatchers.Main) {
           updateResults(users, startTime)
       }
   }
   ```

   - `updateResults()` should be called on the main UI thread, so you call it with the context of `Dispatchers.Main`.
   - `withContext()` calls the given code with the specified coroutine context, is suspended until it completes, and returns the result. An alternative but more verbose way to express this would be to start a new coroutine and explicitly wait (by suspending) until it completes: `launch(context) { ... }.join()`.

3. Run the code and ensure that the coroutines are executed on the threads from the thread pool.

Structured concurrency

- The *coroutine scope* is responsible for the structure and parent-child relationships between different coroutines. New coroutines usually need to be started inside a scope.
- The *coroutine context* stores additional technical information used to run a given coroutine, like the coroutine custom name, or the dispatcher specifying the threads the coroutine should be scheduled on.

When `launch`, `async`, or `runBlocking` are used to start a new coroutine, they automatically create the corresponding scope. All of these functions take a lambda with a receiver as an argument, and `CoroutineScope` is the implicit receiver type:

```
launch { /* this: CoroutineScope */ }
```

- New coroutines can only be started inside a scope.

- `launch` and `async` are declared as extensions to `CoroutineScope`, so an implicit or explicit receiver must always be passed when you call them.
- The coroutine started by `runBlocking` is the only exception because `runBlocking` is defined as a top-level function. But because it blocks the current thread, it's intended primarily to be used in `main()` functions and tests as a bridge function.

A new coroutine inside `runBlocking`, `launch`, or `async` is started automatically inside the scope:

```
import kotlinx.coroutines.*

fun main() = runBlocking { /* this: CoroutineScope */
    launch { /* ... */ }
    // the same as:
    this.launch { /* ... */ }
}
```

When you call `launch` inside `runBlocking`, it's called as an extension to the implicit receiver of the `CoroutineScope` type. Alternatively, you could explicitly write `this.launch`.

The nested coroutine (started by `launch` in this example) can be considered as a child of the outer coroutine (started by `runBlocking`). This "parent-child" relationship works through scopes; the child coroutine is started from the scope corresponding to the parent coroutine.

It's possible to create a new scope without starting a new coroutine, by using the `coroutineScope` function. To start new coroutines in a structured way inside a `suspend` function without access to the outer scope, you can create a new coroutine scope that automatically becomes a child of the outer scope that this `suspend` function is called from. `loadContributorsConcurrent()`is a good example.

You can also start a new coroutine from the global scope using `GlobalScope.async` or `GlobalScope.launch`. This will create a top-level "independent" coroutine.

The mechanism behind the structure of the coroutines is called *structured concurrency*. It provides the following benefits over global scopes:

- The scope is generally responsible for child coroutines, whose lifetime is attached to the lifetime of the scope.

- The scope can automatically cancel child coroutines if something goes wrong or a user changes their mind and decides to revoke the operation.
- The scope automatically waits for the completion of all child coroutines. Therefore, if the scope corresponds to a coroutine, the parent coroutine does not complete until all the coroutines launched in its scope have completed.

When using `GlobalScope.async`, there is no structure that binds several coroutines to a smaller scope. Coroutines started from the global scope are all independent – their lifetime is limited only by the lifetime of the whole application. It's possible to store a reference to the coroutine started from the global scope and wait for its completion or cancel it explicitly, but that won't happen automatically as it would with structured concurrency.

Canceling the loading of contributors

Consider two versions of the `loadContributorsConcurrent()` function. The first uses `coroutineScope` to start all of the child coroutines, whereas the second uses `GlobalScope`. Compare how both versions behave when you try to cancel the parent coroutine.

1. Copy the implementation of `loadContributorsConcurrent()` from `Request5Concurrent.kt` to `loadContributorsNotCancellable()` in `Request5NotCancellable.kt`, and then remove the creation of a new `coroutineScope`.
2. The `async` calls now fail to resolve, so start them by using `GlobalScope.async`:

```
suspend fun loadContributorsNotCancellable(
    service: GitHubService,
    req: RequestData
): List<User> {   // #1
    // ...
    GlobalScope.async {   // #2
        log("starting loading for ${repo.name}")
        // load repo contributors
    }
    // ...
    return deferreds.awaitAll().flatten().aggregate()  // #3
}
```

- The function now returns the result directly, not as the last expression inside the

lambda (lines `#1` and `#3`).

- All of the "contributors" coroutines are started inside the `GlobalScope`, not as children of the coroutine scope (line `#2`).

3. Add a 3-second delay to all of the coroutines that send requests, so that there's enough time to cancel the loading after the coroutines are started but before the requests are sent:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService,
    req: RequestData
): List<User> = coroutineScope {
    // ...
    async {
        log("starting loading for ${repo.name}")
        delay(3000)
        // load repo contributors
    }
    // ...
}
```

4. Run the program and choose the *CONCURRENT* option to load the contributors.

5. Wait until all of the "contributors" coroutines are started, and then click *Cancel*. The log shows no new results, which means that all of the requests were indeed canceled:

```
2896 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin:
loaded 40 repos
2901 [DefaultDispatcher-worker-2 @coroutine#4] INFO  Contributors
- starting loading for kotlin-koans
...
2909 [DefaultDispatcher-worker-5 @coroutine#36] INFO  Contributors
- starting loading for mpp-example
/* click on 'cancel' */
/* no requests are sent */
```

6. Repeat step 5, but this time choose the `NOT_CANCELLABLE` option:

```
2570 [AWT-EventQueue-0 @coroutine#1] INFO  Contributors - kotlin:
loaded 30 repos
2579 [DefaultDispatcher-worker-1 @coroutine#4] INFO  Contributors
- starting loading for kotlin-koans
...
2586 [DefaultDispatcher-worker-6 @coroutine#36] INFO  Contributors
- starting loading for mpp-example
/* click on 'cancel' */
/* but all the requests are still sent: */
6402 [DefaultDispatcher-worker-5 @coroutine#4] INFO  Contributors
- kotlin-koans: loaded 45 contributors
...
9555 [DefaultDispatcher-worker-8 @coroutine#36] INFO  Contributors
- mpp-example: loaded 8 contributors
```

In this case, no coroutines are canceled, and all the requests are still sent.

7. Check how the cancellation is triggered in the "contributors" program. When the *Cancel* button is clicked, the main "loading" coroutine is explicitly canceled and the child coroutines are canceled automatically:

```
interface Contributors {

    fun loadContributors() {
        // ...
        when (getSelectedVariant()) {
            CONCURRENT -> {
                launch {
                    val users =
loadContributorsConcurrent(service, req)
                    updateResults(users, startTime)
                }.setUpCancellation()      // #1
            }
        }
    }

    private fun Job.setUpCancellation() {
        val loadingJob = this              // #2
```

```
        // cancel the loading job if the 'cancel' button was
clicked:
        val listener = ActionListener {
            loadingJob.cancel()            // #3
            updateLoadingStatus(CANCELED)
        }
        // add a listener to the 'cancel' button:
        addCancelListener(listener)

        // update the status and remove the listener
        // after the loading job is completed
    }
}
```

The `launch` function returns an instance of `Job`. `Job` stores a reference to the "loading coroutine", which loads all of the data and updates the results. You can call the `setUpCancellation()` extension function on it (line `#1`), passing an instance of `Job` as a receiver.

Another way you could express this would be to explicitly write:

```
val job = launch { }
job.setUpCancellation()
```

- For readability, you could refer to the `setUpCancellation()` function receiver inside the function with the new `loadingJob` variable (line `#2`).
- Then you could add a listener to the *Cancel* button so that when it's clicked, the `loadingJob` is canceled (line `#3`).

With structured concurrency, you only need to cancel the parent coroutine and this automatically propagates cancellation to all of the child coroutines.

Using the outer scope's context

When you start new coroutines inside the given scope, it's much easier to ensure that all of them run with the same context. It is also much easier to replace the context if needed.

Now it's time to learn how using the dispatcher from the outer scope works. The new scope created by the `coroutineScope` or by the coroutine builders always inherits the context from the outer scope. In this case, the outer scope is the scope the `suspend loadContributorsConcurrent()` function was called from:

```
launch(Dispatchers.Default) {  // outer scope
    val users = loadContributorsConcurrent(service, req)
    // ...
}
```

All of the nested coroutines are automatically started with the inherited context. The dispatcher is a part of this context. That's why all of the coroutines started by `async` are started with the context of the default dispatcher:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService, req: RequestData
): List<User> = coroutineScope {
    // this scope inherits the context from the outer scope
    // ...
    async {   // nested coroutine started with the inherited context
        // ...
    }
    // ...
}
```

With structured concurrency, you can specify the major context elements (like dispatcher) once, when creating the top-level coroutine. All the nested coroutines then inherit the context and modify it only if needed.

When you write code with coroutines for UI applications, for example Android ones, it's a common practice to use `CoroutineDispatchers.Main` by default for the top coroutine and then to explicitly put a different dispatcher when you need to run the code on a different thread.

Showing progress

Despite the information for some repositories being loaded rather quickly, the user only sees the resulting list after all of the data has been loaded. Until then, the loader icon runs

showing the progress, but there's no information about the current state or what contributors are already loaded.

You can show the intermediate results earlier and display all of the contributors after loading the data for each of the repositories:

Loading data

To implement this functionality, in the `src/tasks/Request6Progress.kt`, you'll need to pass the logic updating the UI as a callback, so that it's called on each intermediate state:

```
suspend fun loadContributorsProgress(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) {
    // loading the data
    // calling `updateResults()` on intermediate states
}
```

On the call site in Contributors.kt, the callback is passed to update the results from the Main thread for the *PROGRESS* option:

```
launch(Dispatchers.Default) {
    loadContributorsProgress(service, req) { users, completed ->
        withContext(Dispatchers.Main) {
            updateResults(users, startTime, completed)
        }
    }
}
```

- The updateResults() parameter is declared as suspend in loadContributorsProgress(). It's necessary to call withContext, which is a suspend function inside the corresponding lambda argument.
- updateResults() callback takes an additional Boolean parameter as an argument specifying whether the loading has completed and the results are final.

Task 6

In the Request6Progress.kt file, implement the loadContributorsProgress() function that shows the intermediate progress. Base it on the loadContributorsSuspend() function from Request4Suspend.kt.

- Use a simple version without concurrency; you'll add it later in the next section.
- The intermediate list of contributors should be shown in an "aggregated" state, not just the list of users loaded for each repository.
- The total number of contributions for each user should be increased when the data for each new repository is loaded.

Solution for task 6

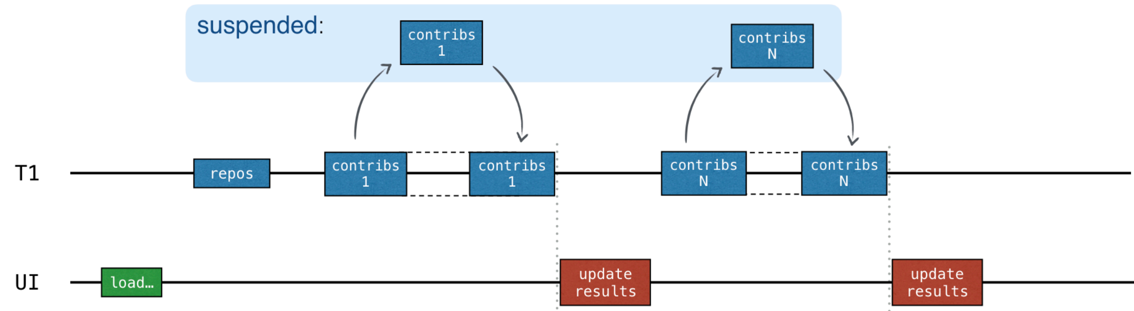
To store the intermediate list of loaded contributors in the "aggregated" state, define an allUsers variable which stores the list of users, and then update it after contributors for each new repository are loaded:

```
suspend fun loadContributorsProgress(
    service: GitHubService,
```

```
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) {
    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    var allUsers = emptyList<User>()
    for ((index, repo) in repos.withIndex()) {
        val users = service.getRepoContributors(req.org, repo.name)
            .also { logUsers(repo, it) }
            .bodyList()

        allUsers = (allUsers + users).aggregate()
        updateResults(allUsers, index == repos.lastIndex)
    }
}
```

Consecutive vs concurrent

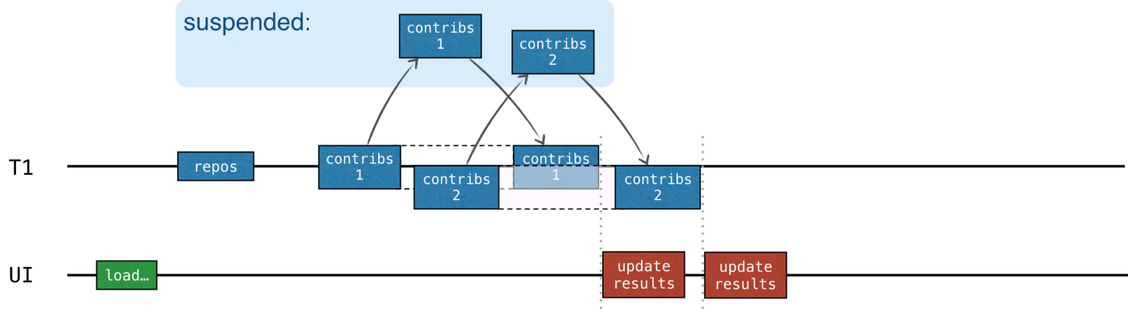
An `updateResults()` callback is called after each request is completed:



Progress on requests

This code doesn't include concurrency. It's sequential, so you don't need synchronization.

The best option would be to send requests concurrently and update the intermediate results after getting the response for each repository:



Concurrent requests

To add concurrency, use *channels*.

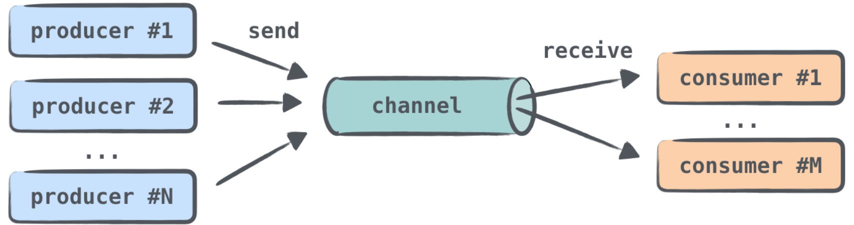
Channels

Writing code with a shared mutable state is quite difficult and error-prone (like in the solution using callbacks). A simpler way is to share information by communication rather than by using a common mutable state. Coroutines can communicate with each other through *channels*.

Channels are communication primitives that allow data to be passed between coroutines. One coroutine can *send* some information to a channel, while another can *receive* that information from it:

Using channels

A coroutine that sends (produces) information is often called a producer, and a coroutine that receives (consumes) information is called a consumer. One or multiple coroutines can send information to the same channel, and one or multiple coroutines can receive data from it:



Using channels with many coroutines

When many coroutines receive information from the same channel, each element is handled only once by one of the consumers. Once an element is handled, it is immediately removed from the channel.

You can think of a channel as similar to a collection of elements, or more precisely, a queue, in which elements are added to one end and received from the other. However, there's an important difference: unlike collections, even in their synchronized versions, a channel can *suspend* `send()`and `receive()` operations. This happens when the channel is empty or full. The channel can be full if the channel size has an upper bound.

`Channel` is represented by three different interfaces: `SendChannel`, `ReceiveChannel`, and `Channel`, with the latter extending the first two. You usually create a channel and give it to producers as a `SendChannel` instance so that only they can send information to the channel. You give a channel to consumers as a `ReceiveChannel` instance so that only they can receive from it. Both `send` and `receive` methods are declared as `suspend`:

```
interface SendChannel<in E> {
    suspend fun send(element: E)
    fun close(): Boolean
}

interface ReceiveChannel<out E> {
    suspend fun receive(): E
}

interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

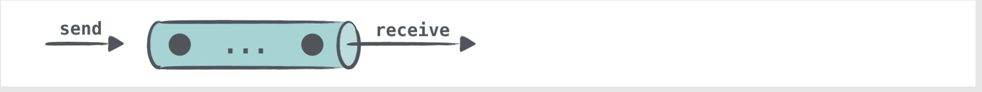
The producer can close a channel to indicate that no more elements are coming.

Several types of channels are defined in the library. They differ in how many elements they can internally store and whether the `send()` call can be suspended or not. For all of the

channel types, the `receive()` call behaves similarly: it receives an element if the channel is not empty; otherwise, it is suspended.

Unlimited channel

An unlimited channel is the closest analog to a queue: producers can send elements to this channel and it will keep growing indefinitely. The send() call will never be suspended. If the program runs out of memory, you'll get an OutOfMemoryException. The difference between an unlimited channel and a queue is that when a consumer tries to receive from an empty channel, it becomes suspended until some new elements are sent.



Unlimited channel

Buffered channel

The size of a buffered channel is constrained by the specified number. Producers can send elements to this channel until the size limit is reached. All of the elements are internally stored. When the channel is full, the next `send` call on it is suspended until more free space becomes available.

Buffered channel

Rendezvous channel

The "Rendezvous" channel is a channel without a buffer, the same as a buffered channel with zero size. One of the functions (send() or receive()) is always suspended until the other is called.

If the send() function is called and there's no suspended receive call ready to process the element, then send() is suspended. Similarly, if the receive function is called and

the channel is empty or, in other words, there's no suspended send() call ready to send the element, the receive() call is suspended.

The "rendezvous" name ("a meeting at an agreed time and place") refers to the fact that send() and receive() should "meet on time".

Rendezvous channel

Conflated channel

A new element sent to the conflated channel will overwrite the previously sent element, so the receiver will always get only the latest element. The send() call is never suspended.

Conflated channel

When you create a channel, specify its type or the buffer size (if you need a buffered one):

```
val rendezvousChannel = Channel<String>()
val bufferedChannel = Channel<String>(10)
val conflatedChannel = Channel<String>(CONFLATED)
val unlimitedChannel = Channel<String>(UNLIMITED)
```

By default, a "Rendezvous" channel is created.

In the following task, you'll create a "Rendezvous" channel, two producer coroutines, and a consumer coroutine:

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
```

```
    val channel = Channel<String>()
    launch {
        channel.send("A1")
        channel.send("A2")
        log("A done")
    }
    launch {
        channel.send("B1")
        log("B done")
    }
    launch {
        repeat(3) {
            val x = channel.receive()
            log(x)
        }
    }
}

fun log(message: Any?) {
    println("[${Thread.currentThread().name}] $message")
}
```

> Watch this video (https://www.youtube.com/watch?v=HpWQUoVURWQ) for a better understanding of channels.

Task 7

In `src/tasks/Request7Channels.kt`, implement the function `loadContributorsChannels()` that requests all of the GitHub contributors concurrently and shows intermediate progress at the same time.

Use the previous functions, `loadContributorsConcurrent()` from `Request5Concurrent.kt` and `loadContributorsProgress()` from `Request6Progress.kt`.

Tip for task 7

Different coroutines that concurrently receive contributor lists for different repositories can send all of the received results to the same channel:

```
val channel = Channel<List<User>>()
for (repo in repos) {
    launch {
        val users = TODO()
        // ...
        channel.send(users)
    }
}
```

Then the elements from this channel can be received one by one and processed:

```
repeat(repos.size) {
    val users = channel.receive()
    // ...
}
```

Since the `receive()` calls are sequential, no additional synchronization is needed.

Solution for task 7

As with the `loadContributorsProgress()` function, you can create an `allUsers` variable to store the intermediate states of the "all contributors" list. Each new list received from the channel is added to the list of all users. You aggregate the result and update the state using the `updateResults` callback:

```
suspend fun loadContributorsChannels(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) = coroutineScope {

    val repos = service
        .getOrgRepos(req.org)
        .also { logRepos(req, it) }
        .bodyList()

    val channel = Channel<List<User>>()
    for (repo in repos) {
        launch {
```

```
            val users = service.getRepoContributors(req.org,
repo.name)
                .also { logUsers(repo, it) }
                .bodyList()
            channel.send(users)
        }
    }
    var allUsers = emptyList<User>()
    repeat(repos.size) {
        val users = channel.receive()
        allUsers = (allUsers + users).aggregate()
        updateResults(allUsers, it == repos.lastIndex)
    }
}
```

- Results for different repositories are added to the channel as soon as they are ready. At first, when all of the requests are sent, and no data is received, the `receive()` call is suspended. In this case, the whole "load contributors" coroutine is suspended.
- Then, when the list of users is sent to the channel, the "load contributors" coroutine resumes, the `receive()` call returns this list, and the results are immediately updated.

You can now run the program and choose the *CHANNELS* option to load the contributors and see the result.

Although neither coroutines nor channels completely remove the complexity that comes with concurrency, they make life easier when you need to understand what's going on.

Testing coroutines

Let's now test all solutions to check that the solution with concurrent coroutines is faster than the solution with the `suspend` functions, and check that the solution with channels is faster than the simple "progress" one.

In the following task, you'll compare the total running time of the solutions. You'll mock a GitHub service and make this service return results after the given timeouts:

```
repos request - returns an answer within 1000 ms delay
repo-1 - 1000 ms delay
```

```
repo-2 - 1200 ms delay
repo-3 - 800 ms delay
```

The sequential solution with the `suspend` functions should take around 4000 ms (4000 = 1000 + (1000 + 1200 + 800)). The concurrent solution should take around 2200 ms (2200 = 1000 + max(1000, 1200, 800)).

For the solutions that show progress, you can also check the intermediate results with timestamps.

The corresponding test data is defined in `test/contributors/testData.kt`, and the files `Request4SuspendKtTest`, `Request7ChannelsKtTest`, and so on contain the straightforward tests that use mock service calls.

However, there are two problems here:

- These tests take too long to run. Each test takes around 2 to 4 seconds, and you need to wait for the results each time. It's not very efficient.
- You can't rely on the exact time the solution runs because it still takes additional time to prepare and run the code. You could add a constant, but then the time would differ from machine to machine. The mock service delays should be higher than this constant so you can see a difference. If the constant is 0.5 sec, making the delays 0.1 sec won't be enough.

A better way would be to use special frameworks to test the timing while running the same code several times (which increases the total time even more), but that is complicated to learn and set up.

To solve these problems and make sure that solutions with provided test delays behave as expected, one faster than the other, use *virtual* time with a special test dispatcher. This dispatcher keeps track of the virtual time passed from the start and runs everything immediately in real time. When you run coroutines on this dispatcher, the `delay` will return immediately and advance the virtual time.

Tests that use this mechanism run fast, but you can still check what happens at different moments in virtual time. The total running time drastically decreases:

Real time:		Virtual time:	
tasks.Request4SuspendKtTest	4 s 16 ms	tasks.Request4SuspendKtTest	3 ms
tasks.Request5ConcurrentKtTest	2 s 214 ms	tasks.Request5ConcurrentKtTest	6 ms
tasks.Request6ProgressKtTest	4 s 16 ms	tasks.Request6ProgressKtTest	3 ms
tasks.Request7ChannelsKtTest	2 s 276 ms	tasks.Request7ChannelsKtTest	3 ms

Comparison for total running time

To use virtual time, replace the `runBlocking` invocation with a `runTest`. `runTest` takes an extension lambda to `TestScope` as an argument. When you call `delay` in a `suspend` function inside this special scope, `delay` will increase the virtual time instead of delaying in real time:

```
@Test
fun testDelayInSuspend() = runTest {
    val realStartTime = System.currentTimeMillis()
    val virtualStartTime = currentTime

    foo()
    println("${System.currentTimeMillis() - realStartTime} ms") // ~
6 ms
    println("${currentTime - virtualStartTime} ms")             //
1000 ms
}

suspend fun foo() {
    delay(1000)    // auto-advances without delay
    println("foo") // executes eagerly when foo() is called
}
```

You can check the current virtual time using the `currentTime` property of `TestScope`.

The actual running time in this example is several milliseconds, whereas virtual time equals the delay argument, which is 1000 milliseconds.

To get the full effect of "virtual" `delay` in child coroutines, start all of the child coroutines with `TestDispatcher`. Otherwise, it won't work. This dispatcher is automatically inherited from the other `TestScope`, unless you provide a different dispatcher:

```
@Test
fun testDelayInLaunch() = runTest {
    val realStartTime = System.currentTimeMillis()
    val virtualStartTime = currentTime
```

```
    bar()

    println("${System.currentTimeMillis() - realStartTime} ms") // ~
11 ms
    println("${currentTime - virtualStartTime} ms")             //
1000 ms
}

suspend fun bar() = coroutineScope {
    launch {
        delay(1000)    // auto-advances without delay
        println("bar") // executes eagerly when bar() is called
    }
}
```

If `launch` is called with the context of `Dispatchers.Default` in the example above, the test will fail. You'll get an exception saying that the job has not been completed yet.

You can test the `loadContributorsConcurrent()` function this way only if it starts the child coroutines with the inherited context, without modifying it using the `Dispatchers.Default` dispatcher.

You can specify the context elements like the dispatcher when *calling* a function rather than when *defining* it, which allows for more flexibility and easier testing.

The testing API that supports virtual time is Experimental (Kotlin 各部分组件的稳定性) and may change in the future.

By default, the compiler shows warnings if you use the experimental testing API. To suppress these warnings, annotate the test function or the whole class containing the tests with `@OptIn(ExperimentalCoroutinesApi::class)`. Add the compiler argument instructing the compiler that you're using the experimental API:

```
compileTestKotlin {
    kotlinOptions {
        freeCompilerArgs += "-Xuse-experimental=kotlin.Experimental"
```

```
    }
}
```

In the project corresponding to this tutorial, the compiler argument has already been added to the Gradle script.

Task 8

Refactor the following tests in `tests/tasks/` to use virtual time instead of real time:

- Request4SuspendKtTest.kt
- Request5ConcurrentKtTest.kt
- Request6ProgressKtTest.kt
- Request7ChannelsKtTest.kt

Compare the total running times before and after applying your refactoring.

Tip for task 8

1. Replace the `runBlocking` invocation with `runTest`, and replace `System.currentTimeMillis()` with `currentTime`:

   ```
   @Test
   fun test() = runTest {
       val startTime = currentTime
       // action
       val totalTime = currentTime - startTime
       // testing result
   }
   ```

2. Uncomment the assertions that check the exact virtual time.
3. Don't forget to add `@UseExperimental(ExperimentalCoroutinesApi::class)`.

Solution for task 8

Here are the solutions for the concurrent and channels cases:

```
fun testConcurrent() = runTest {
    val startTime = currentTime
    val result = loadContributorsConcurrent(MockGithubService,
testRequestData)
    Assert.assertEquals("Wrong result for
'loadContributorsConcurrent'", expectedConcurrentResults.users,
result)
    val totalTime = currentTime - startTime

    Assert.assertEquals(
        "The calls run concurrently, so the total virtual time
should be 2200 ms: " +
                "1000 for repos request plus max(1000, 1200, 800) =
1200 for concurrent contributors requests)",
        expectedConcurrentResults.timeFromStart, totalTime
    )
}
```

First, check that the results are available exactly at the expected virtual time, and then check the results themselves:

```
fun testChannels() = runTest {
    val startTime = currentTime
    var index = 0
    loadContributorsChannels(MockGithubService, testRequestData) {
users, _ ->
        val expected = concurrentProgressResults[index++]
        val time = currentTime - startTime
        Assert.assertEquals(
            "Expected intermediate results after
${expected.timeFromStart} ms:",
            expected.timeFromStart, time
        )
        Assert.assertEquals("Wrong intermediate results after
$time:", expected.users, users)
    }
}
```

The first intermediate result for the last version with channels becomes available sooner than the progress version, and you can see the difference in tests that use virtual time.

The tests for the remaining "suspend" and "progress" tasks are very similar – you can find them in the project's `solutions` branch.

What's next

- Check out the Asynchronous Programming with Kotlin (https://kotlinconf.com/workshops/) workshop at KotlinConf.
- Find out more about using virtual time and the experimental testing package (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-test/).

取消与超时

最终更新: 2024/09/10

本章介绍协程的取消与超时.

取消协程的运行

在一个长期运行的应用程序中, 你可能会需要在你的后台协程中进行一些更加精细的控制. 比如, 使用者可能已经关闭了某个启动协程的页面, 现在它的计算结果已经不需要了, 因此协程的执行可以取消. launch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) 函数会返回一个 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html), 可以通过它来取消正在运行的协程:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(1300L) // 等待一段时间
    println("main: I'm tired of waiting!")
    job.cancel() // 取消 job
    job.join() // 等待 job 结束
    println("main: Now I can quit.")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-01.kt).

这个示例的运行结果如下:

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
main: Now I can quit.
```

一旦 main 函数调用 `job.cancel`, 我们就再也看不到协程的输出了, 因为协程已经被取消了. 还有一个 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) 上的扩展函数 cancelAndJoin (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/cancel-and-join.html), 它组合了 cancel (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/cancel.html) 和 join (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/join.html) 两个操作.

取消是协作式的

协程的取消是 *协作式的*. 协程的代码必须与外接配合, 才能够被取消. `kotlinx.coroutines` 库中的所有挂起函数都是 *可取消的*. 这些函数会检查协程是否被取消, 并在被取消时抛出 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html) 异常. 但是, 如果一个协程正在进行计算, 并且没有检查取消状态, 那么它是不可被取消的, 比如下面的例子:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (i < 5) { // 一个浪费 CPU 的计算任务循环
            // 每秒输出信息 2 次
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
```

```
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L) // 等待一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消 job, 并等待它结束
    println("main: Now I can quit.")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-02.kt).

运行一下这个示例, 我们会看到, 即使在取消之后, 协程还是继续输出 "I'm sleeping" 信息, 直到循环 5 次之后, 协程才自己结束.

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
job: I'm sleeping 3 ...
job: I'm sleeping 4 ...
main: Now I can quit.
```

如果捕获一个 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html) 然后不再抛出它, 也可以观察到同样的问题:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch(Dispatchers.Default) {
        repeat(5) { i ->
            try {
```

```
                // 每秒输出信息 2 次
                println("job: I'm sleeping $i ...")
                delay(500)
            } catch (e: Exception) {
                // 将异常输出到 log
                println(e)
            }
        }
    }
    delay(1300L) // 等待一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消 job, 并等待它结束
    println("main: Now I can quit.")
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-03.kt).

尽管示例中的捕获 `Exception` 是一种反模式, 但在更加微妙的情况下还是会出现这个问题, 比如在使用 `runCatching` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/run-catching.html) 函数时, 它不会重新抛出 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html).

使计算代码能够被取消

有两种方法可以让我们的计算代码变得能够被取消. 第一种办法是定期调用一个挂起函数, 检查协程是否被取消. 有一个 yield (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/yield.html) 函数可以用来实现这个目的. 另一种方法是显式地检查协程的取消状态. 我们来试试后一种方法.

我们来把前面的示例程序中的 `while (i < 5)` 改为 `while (isActive)`, 然后再运行, 看看结果如何.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
```

```
//sampleStart
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (isActive) { // 可被取消的计算循环
            // 每秒输出信息 2 次
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L) // 等待一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消 job, 并等待它结束
    println("main: Now I can quit.")
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-04.kt).

你会看到, 现在循环变得能够被取消了. isActive (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/is-active.html) 是一个扩展属性, 在协程内部的代码中可以通过 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 对象访问到.

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
main: Now I can quit.
```

使用 finally 语句来关闭资源

可被取消的挂起函数, 在被取消时会抛出 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html) 异常, 这个异常可以通过通常的方式来处理. 比如, 可以使用 `try {...} finally {...}` 表达式, 或者 Kotlin 的 `use` 函数, 以便在一个协程被取消时执行结束处理:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        try {
            repeat(1000) { i ->
                println("job: I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("job: I'm running finally")
        }
    }
    delay(1300L) // 等待一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消 job, 并等待它结束
    println("main: Now I can quit.")
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-05.kt).

join (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/join.html) 和 cancelAndJoin (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-

core/kotlinx.coroutines/cancel-and-join.html) 都会等待所有的结束处理执行完毕, 因此上面的示例程序会产生这样的输出:

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
job: I'm running finally
main: Now I can quit.
```

运行无法取消的代码段

如果试图在上面示例程序的 `finally` 代码段中使用挂起函数, 会导致 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html) 异常, 因为执行这段代码的协程已被取消了. 通常, 这不是问题, 因为所有正常的资源关闭操作(关闭文件, 取消任务, 或者关闭任何类型的通信通道)通常都是非阻塞的, 而且不需要用到任何挂起函数. 但是, 在极少数情况下, 如果你需要在已被取消的协程中执行挂起操作, 你可以使用 withContext (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-context.html) 函数和 NonCancellable (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-non-cancellable/index.html) 上下文, 把相应的代码包装在 `withContext(NonCancellable) {...}` 内, 如下例所示:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        try {
            repeat(1000) { i ->
                println("job: I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            withContext(NonCancellable) {
                println("job: I'm running finally")
                delay(1000L)
                println("job: And I've just delayed for 1 sec
```

```
because I'm non-cancellable")
            }
        }
    }
    delay(1300L) // 等待一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消 job, 并等待它结束
    println("main: Now I can quit.")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-06.kt).

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
job: I'm running finally
job: And I've just delayed for 1 sec because I'm non-cancellable
main: Now I can quit.
```

超时

取消一个协程最明显的实际理由就是, 它的运行时间超过了某个时间限制. 当然, 你可以手动追踪协程对应的 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html), 然后启动另一个协程, 在等待一段时间之后取消你追踪的那个协程, 但 Kotlin 已经提供了一个 withTimeout (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-timeout.html) 函数来完成这个任务. 请看下面的例子:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    withTimeout(1300L) {
```

```
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-07.kt).

这个例子的运行结果是:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Exception in thread "main"
kotlinx.coroutines.TimeoutCancellationException: Timed out waiting
for 1300 ms
```

withTimeout (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-timeout.html) 函数抛出的 `TimeoutCancellationException` 异常是 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html) 的子类. 我们在前面的例子中, 都没有看到过 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html) 异常的调用栈被输出到控制台. 这是因为, 在被取消的协程中 `CancellationException` 被认为是协程结束的一个正常原因. 但是, 在这个例子中我们直接在 `main` 函数内使用了 `withTimeout`.

由于协程的取消只是一个异常, 因此所有的资源都可以通过通常的方式来关闭. 如果你需要在超时发生时执行一些额外的操作, 可以将带有超时控制的代码封装在一个 `try {...} catch (e: TimeoutCancellationException) {...}` 代码块中, 也可以使用 withTimeoutOrNull (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-timeout-or-null.html) 函数, 它与 withTimeout (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-

core/kotlinx.coroutines/with-timeout.html) 函数类似, 但在超时发生时, 它会返回 null, 而不是抛出异常:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val result = withTimeoutOrNull(1300L) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
        "Done" // 协程会在输出这个消息之前被取消
    }
    println("Result is $result")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-08.kt).

这段代码的运行结果不会有异常发生了:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Result is null
```

异步的超时与资源管理

withTimeout (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-timeout.html) 中的超时事件异步于它的代码段中运行的代码, 超时事件可以在任何时刻发生, 甚至刚好在从超时的代码段中返回之前. 如果你在代码段之内打开或获取某种资源, 而且需要在代码段之外关闭或释放这些资源, 那么请牢记这一点.

比如, 我们使用 Resource 类模拟一个可关闭的资源, 它只是记录自己被创建了多少次, 在创建时增加 acquired 计数器, 并在 close 函数中减少计数器. 现在我们来创建很多个协程, 每个协程在

withTimeout 代码段的末尾创建一个 Resource, 然后在代码段之外释放资源. 我们添加一个小的延迟, 因此更可能在 withTimeout 代码段结束之后发生超时, 导致资源泄露.

```
import kotlinx.coroutines.*

//sampleStart
var acquired = 0

class Resource {
    init { acquired++ } // 获取资源
    fun close() { acquired-- } // 释放资源
}

fun main() {
    runBlocking {
        repeat(10_000) { // 启动 10K 个协程
            launch {
                val resource = withTimeout(60) { // 超时设定为 60 ms
                    delay(50) // 延迟 50 ms
                    Resource() // 获取资源, 然后从 withTimeout 代码段返
回这个资源
                }
                resource.close() // 释放资源
            }
        }
    }
    // 在 runBlocking 之外, 所有的协程都已运行结束
    println(acquired) // 输出未被释放的资源数量
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-09.kt).

运行上面的代码, 你会看到输出结果并不总是 0, 具体情况依赖于你的机器的时间. 你可能需要调整示例代码中的超时设置, 才能看到非 0 的结果.

> 请注意, 这个例子中, 从 10K 个协程中增加和减少 `acquired` 计数器, 完全是线程安全的, 因为这个处理永远发生在 `runBlocking` 所使用的同一个线程内. 更多细节将在下一章, 关于协程上下文的部分中解释.

这个问题的解决方法是, 可以将资源的引用保存到一个变量中, 而不是从 `withTimeout` 代码段直接返回资源.

```
import kotlinx.coroutines.*

var acquired = 0

class Resource {
    init { acquired++ } // 获取资源
    fun close() { acquired-- } // 释放资源
}

fun main() {
//sampleStart
    runBlocking {
        repeat(10_000) { // 启动 10K 个协程
            launch {
                var resource: Resource? = null // 这时资源还没有获取
                try {
                    withTimeout(60) { // 超时设定为 60 ms
                        delay(50) // 延迟 50 ms
                        resource = Resource() // 如果获取成功, 将资源保
存到变量
                    }
                    // 我们可以在这里对资源进行一些其他操作
                } finally {
                    resource?.close() // 如果获取成功, 释放资源
                }
            }
        }
    }
    // 在 runBlocking 之外, 所有的协程都已运行结束
    println(acquired) // 输出未被释放的资源数量
```

```
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-cancel-10.kt).

这段示例程序永远会输出 0. 也就是说, 没有发生资源泄露:

```
0
```

挂起函数(Suspending Function)的组合

最终更新: 2024/09/10

本章介绍将挂起函数组合起来的几种不同方式.

默认的连续执行

假设我们有两个挂起函数, 代表在其他地方进行一些有用的工作, 比如调用某种远程服务或运算. 我们先假定这两个函数都是有真实用途的, 但在示例程序中, 我们的挂起函数只是延迟 1 秒钟:

```
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了某些有用的工作
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了某些有用的工作
    return 29
}
```

如果我们需要 *连续地* 调用这两个函数 — 首先需要调用 `doSomethingUsefulOne` *然后再调用* `doSomethingUsefulTwo`, 并且计算这两个函数结果的总和, 那么我们应该怎么做呢? 实际应用中, 我们可能需要使用第一个函数的结果来做一些判断, 决定是否需要调用第二个函数, 或者决定应该如何调用第二个函数.

我们使用一个通常的连续调用, 因为在协程内的代码, 就好象通常的代码一样, 默认就是 *连续* 的. 下面的示例程序会测量执行两个挂起函数时的总执行时间, 演示两个挂起函数执行时的连续行:

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        val one = doSomethingUsefulOne()
```

```
        val two = doSomethingUsefulTwo()
        println("The answer is ${one + two}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了某些有用的工作
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了某些有用的工作
    return 29
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-compose-01.kt).

这个示例程序的输出大致会是:

```
The answer is 42
Completed in 2017 ms
```

使用 async 并发执行

如果在 `doSomethingUsefulOne` 和 `doSomethingUsefulTwo` 的调用之间不存在依赖关系, 我们想要 *并发地* 执行这两个函数, 以便更快得到结果, 那么应该怎么做? 这时 async (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) 可以帮助我们.

概念上来说, async (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) 就好象 launch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) 一样. 它启动一个独立的协程, 也就是一个轻量的线程, 与其他所有协程一起并发执行. 区别在于, `launch` 返回一个 Job

(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html), 其中不带有结果值, 而 async 返回一个 Deferred (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/index.html) — 一个轻量的, 非阻塞的 future, 代表一个未来某个时刻可以得到的结果值. 你可以对一个延期值(deferred value)使用 .await() 来得到它最终的计算结果, 但 Deferred 同时也是一个 Job, 因此如果需要的话, 你可以取消它.

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        val one = async { doSomethingUsefulOne() }
        val two = async { doSomethingUsefulTwo() }
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了某些有用的工作
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了某些有用的工作
    return 29
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-compose-02.kt).

这个示例程序的输出大致会是:

```
The answer is 42
Completed in 1017 ms
```

执行速度快了 2 倍, 因为两个协程的执行是并发的. 注意, 协程的并发总是需要明确指定的.

延迟启动的(Lazily started) async

将可选的 `start` 参数设置为 CoroutineStart.LAZY (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-start/-l-a-z-y/index.html), 可以让 async (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) 延迟启动. 这种模式下, 只有在通过 await (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/await.html) 访问协程的计算结果时, 或者调用协程的 `Job` 的 start (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/start.html) 函数时, 才会真正启动协程. 试着运行一下下面的示例程序:

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        val one = async(start = CoroutineStart.LAZY) {
doSomethingUsefulOne() }
        val two = async(start = CoroutineStart.LAZY) {
doSomethingUsefulTwo() }
        // 执行某些计算
        one.start() // 启动第一个协程
        two.start() // 启动第二个协程
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了某些有用的工作
```

```
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了某些有用的工作
    return 29
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-compose-03.kt).

这个示例程序的输出大致会是:

```
The answer is 42
Completed in 1017 ms
```

在上面的示例程序中, 我们定义了两个协程, 但并没有开始执行, 程序员负责决定什么时候调用 start (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/start.html) 函数来明确地启动协程的执行. 我们先启动了 `one`, 然后启动了 `two`, 然后等待两个协程分别结束.

注意, 如果我们在 `println` 内调用 await (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/await.html), 而在此之前没有对各个协程调用 start (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/start.html), 那么会导致两个协程的执行成为连续的, 而不是并行的, 因为 await (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/await.html) 会启动协程并一直等待执行结束, 这并不是我们使用延迟加载功能时期望的效果. 如果计算中使用到的值来自挂起函数的话, 可以使用 `async(start = CoroutineStart.LAZY)` 来代替标准的 `lazy` 函数.

async 风格的函数

> 在这个例子中展示的这种使用异步函数的编程风格只是为了演示目的, 但在其他编程语言中是一种很流行的风格. 我们 **强烈不鼓励** 在 Kotlin 协程中使用这种编程风格, 具体原因将在下文中解释.

我们可以定义一个 async 风格的函数, 它使用一个 GlobalScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/index.html) 引用, 通过 async (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) 协程构建器来 *异步地* 调用 `doSomethingUsefulOne` 和 `doSomethingUsefulTwo`, 以这种方式来关闭结构化的同步. 我们将这类函数的名称加上 "...Async" 后缀, 明确表示这些函数只负责启动异步的计算工作, 函数的使用者需要通过函数返回的延期值(deferred value)来得到计算结果.

> GlobalScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/index.html) 是一个非常精密的 API, 可能会造成严重的影响, 详情会在下文中解释 因此你需要通过 `@OptIn(DelicateCoroutinesApi::class)` 注解来明确的同意使用 `GlobalScope`.

```
// somethingUsefulOneAsync 函数的返回值类型是 Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

// somethingUsefulTwoAsync 函数的返回值类型是 Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}
```

注意, 这些 `xxxAsync` 函数 **不是** *挂起* 函数. 这些函数可以在任何地方使用. 但是, 使用这些函数总是会隐含着异步执行(这里的意思是 *并发*)它内部的动作.

下面的示例程序演示在协程之外使用这类函数:

```
import kotlinx.coroutines.*
import kotlin.system.*

//sampleStart
// 注意, 这个示例中我们没有在 `main` 的右侧使用 `runBlocking`
fun main() {
    val time = measureTimeMillis {
```

```kotlin
        // 我们可以在协程之外初始化异步操作
        val one = somethingUsefulOneAsync()
        val two = somethingUsefulTwoAsync()
        // 但是等待它的执行结果必然使用挂起或阻塞.
        // 这里我们使用 `runBlocking { ... }`, 在等待结果时阻塞主线程
        runBlocking {
            println("The answer is ${one.await() + two.await()}")
        }
    }
    println("Completed in $time ms")
}
//sampleEnd

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了某些有用的工作
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了某些有用的工作
    return 29
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-compose-04.kt).

```
The answer is 42
Completed in 1085 ms
```

考虑一下, 如果在 `val one = somethingUsefulOneAsync()` 和 `one.await()` 表达式之间, 代码存在某种逻辑错误, 程序抛出了一个异常, 程序的操作中止了, 那么会怎么样. 通常来说, 一个全局的错误处理器可以捕获这个异常, 将这个错误输出到 log, 报告给开发者, 但程序仍然可以继续运行, 执行其他的操作. 但在这里, 尽管负责启动 `somethingUsefulOneAsync` 的那部分程序其实已经中止了, 但它仍然会在后台继续运行. 如果使用结构化并发(structured concurrency)方式话, 就不会发生这种问题, 下面我们来介绍这种方式.

使用 async 的结构化并发

我们沿用 使用 async 并发执行 中的示例程序, 从中抽取一个函数, 并发地执行 `doSomethingUsefulOne` 和 `doSomethingUsefulTwo`, 并返回这两个函数结果的和. 由于 async (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) 协程构建器被定义为 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 上的扩展函数, 因此我们使用这个函数时就需要在作用范围内存在 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html), coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 函数可以为我们提供 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html):

```
suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}
```

通过这种方式, 如果 `concurrentSum` 函数内的某个地方发生错误, 抛出一个异常, 那么在这个函数的作用范围内启动的所有协程都会被取消.

```
import kotlinx.coroutines.*
import kotlin.system.*
```

```kotlin
fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        println("The answer is ${concurrentSum()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了某些有用的工作
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了某些有用的工作
    return 29
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-compose-05.kt).

上面的 `main` 函数的输出结果如下, 显然可以看出, 两个函数的执行仍然是并发的:

```
The answer is 42
Completed in 1017 ms
```

通过协程的父子层级关系, 取消总是会层层传递到所有的子协程, 以及子协程的子协程:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    try {
        failedConcurrentSum()
    } catch(e: ArithmeticException) {
        println("Computation failed with ArithmeticException")
    }
}

suspend fun failedConcurrentSum(): Int = coroutineScope {
    val one = async<Int> {
        try {
            delay(Long.MAX_VALUE) // 模拟一个长时间的计算过程
            42
        } finally {
            println("First child was cancelled")
        }
    }
    val two = async<Int> {
        println("Second child throws an exception")
        throw ArithmeticException()
    }
    one.await() + two.await()
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-compose-06.kt).

注意, 由于子协程中的某一个(也就是, `two`)失败, 第一个 `async`, 以及等待子协程的父协程都会被取消:

```
Second child throws an exception
First child was cancelled
Computation failed with ArithmeticException
```

协程上下文与派发器(Dispatcher)

最终更新: 2024/09/10

协程总是在某个上下文环境执行, 上下文环境通过 Kotlin 标准库中定义的 CoroutineContext (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.coroutines/-coroutine-context/) 类型的值来表示.

协程的上下文是一组不同的元素. 最主要的元素是协程的 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html), 这个概念我们前面已经介绍过了, 此外还有任务的派发器(Dispatcher), 本章我们来介绍派发器.

派发器与线程

协程上下文包含了一个 *协程派发器* (参见 CoroutineDispatcher (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-dispatcher/index.html)), 它负责确定对应的协程使用哪个或哪些线程来执行. 协程派发器可以将协程的执行限定在某个特定的线程上, 也可以将协程的执行派发给一个线程池, 或者不加限定, 允许协程运行在任意的线程上.

所有的协程构建器, 比如 launch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) 和 async (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html), 都接受一个可选的 CoroutineContext (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.coroutines/-coroutine-context/) 参数, 这个参数可以用来为新创建的协程显式地指定派发器, 以及其他上下文元素.

我们来看看下面的示例程序:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch { // 使用父协程的上下文, 也就是 main 函数中的 runBlocking 协程
        println("main runBlocking      : I'm working in thread 
${Thread.currentThread().name}")
    }
    launch(Dispatchers.Unconfined) { // 非受限 -- 将会在主线程中执行
```

```
        println("Unconfined            : I'm working in thread
${Thread.currentThread().name}")
    }
    launch(Dispatchers.Default) { // 会被派发到 DefaultDispatcher
        println("Default               : I'm working in thread
${Thread.currentThread().name}")
    }
    launch(newSingleThreadContext("MyOwnThread")) { // 将会在独自的新线
程内执行
        println("newSingleThreadContext: I'm working in thread
${Thread.currentThread().name}")
    }
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-01.kt).

这个示例程序的输出如下 (顺序可能略有不同):

```
Unconfined            : I'm working in thread main
Default               : I'm working in thread DefaultDispatcher-
worker-1
newSingleThreadContext: I'm working in thread MyOwnThread
main runBlocking      : I'm working in thread main
```

当 `launch { ... }` 没有参数时, 它将会从调用它的代码的 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 继承相同的上下文(因此也继承了相应的派发器). 在上面的示例程序中, 它继承了运行在 `main` 线程中主 `runBlocking` 协程的上下文.

Dispatchers.Unconfined (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-unconfined.html) 是一个特殊的派发器, 在我们的示例程序中, 它似乎也是在 `main` 线程中执行协程, 但实际上, 它是一种不同的机制, 我们在后文中详细解释.

如果作用范围(Scope) 中没有明确指定其他派发器, 则会使用默认派发器, 默认派发器用 Dispatchers.Default (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html) 表示, 它会使用后台共享的线程池.

newSingleThreadContext (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/new-single-thread-context.html) 会创建一个新的线程来运行协程. 一个专用的线程是一种非常昂贵的资源. 在真实的应用程序中, 这样的线程, 必须在不再需要的时候使用 close (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-executor-coroutine-dispatcher/close.html) 函数释放它, 或者保存在一个顶层变量中, 并在应用程序内继续重用.

非受限派发器(Unconfined dispatcher)与受限派发器(Confined dispatcher)

Dispatchers.Unconfined (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-unconfined.html) 协程派发器会在调用者线程内启动协程, 但只会持续运行到第一次挂起点为止. 在挂起之后, 它会在哪个线程内恢复协程的执行, 这完全由被调用的挂起函数来决定. 非受限派发器(Unconfined dispatcher) 适用的场景是, 协程不占用 CPU 时间, 也不更新那些限定于某个特定线程的共享数据(比如 UI).

另一方面, 默认情况下, 派发器会继承外层 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 的派发器. 具体来说, 对于 runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 协程, 默认的派发器会限定为调用它的那个线程, 因此继承这个派发器的效果就是, 将协程的执行限定在这个线程上, 并且执行顺序为可预测的先进先出 (FIFO)调度顺序.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch(Dispatchers.Unconfined) { // 非受限 -- 将会在主线程中执行
        println("Unconfined      : I'm working in thread
${Thread.currentThread().name}")
        delay(500)
        println("Unconfined      : After delay in thread
${Thread.currentThread().name}")
    }
```

```
    launch { // 使用父协程的上下文, 也就是 main 函数中的 runBlocking 协程
        println("main runBlocking: I'm working in thread
${Thread.currentThread().name}")
        delay(1000)
        println("main runBlocking: After delay in thread
${Thread.currentThread().name}")
    }
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-02.kt).

上面的示例程序的输出如下:

```
Unconfined      : I'm working in thread main
main runBlocking: I'm working in thread main
Unconfined      : After delay in thread
kotlinx.coroutines.DefaultExecutor
main runBlocking: After delay in thread main
```

因此, 继承了 `runBlocking {...}` 协程的上下文的协程会在 `main` 线程内恢复运行, 而非受限的协程会在默认的执行器线程内恢复运行, 因为它是挂起函数 delay (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/delay.html) 所使用的线程.

> 非受限派发器是一种高级机制, 对于某些极端情况, 如果我们不需要控制协程在哪个线程上执行, 或者由于协程中的某些操作必须立即执行, 因此对其进行控制会导致一些不希望的副作用, 这时使用非受限派发器就非常有用. 在通常的代码中不应该使用非受限派发器.

协程与线程的调试

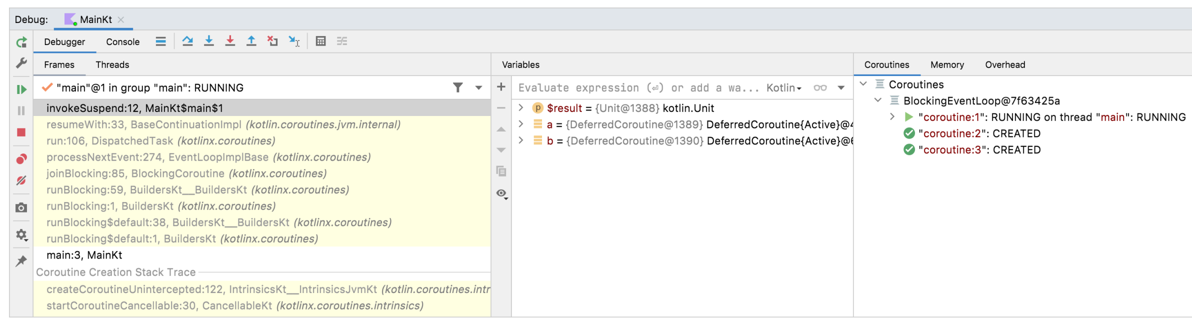
协程可以在一个线程内挂起, 然后在另一个线程中恢复运行. 如果不使用特殊的工具, 那么即使协程的派发器只使用一个线程, 也很难弄清楚协程在哪里, 在什么时间, 具体做了什么操作.

使用 IDEA 进行调试

Kotlin 插件的 Coroutine 调试器帮助我们在 IntelliJ IDEA 中调试协程.

> 调试功能适用于 `kotlinx-coroutines-core` 的 1.3.8 或以后版本.

Debug Tool Window 包含 **Coroutines** 页面. 在这个页面中, 你可以看到运行中的和挂起的协程的信息. 协程按照它们运行时所属的派发器分组.



调试协程

通过协程调试器, 你可以:

- 检查每个协程的状态.
- 对于运行中的协程和挂起的协程, 查看局部变量的值, 以及被捕获的变量的值.
- 查看协程的完整的创建栈, 以及协程之内的调用栈. 这些栈中的每一片都包括变量值, 即使是在通常的调试方式下会丢失的那些变量.
- 得到完整的报告, 包含每个协程的状态以及它的调用栈. 要得到这样的报告, 请在 **Coroutines** 页面内点击鼠标右键, 然后点击 **Get Coroutines Dump**.

要开始协程的调试, 你只需要设置断点, 然后以 debug 模式启动应用程序.

关于协程调试, 更多详情请参见这篇 教程 (https://kotlinlang.org/docs/tutorials/coroutines/debug-coroutines-with-idea.html).

使用日志进行调试

如果没有协程调试器, 那么对于多线程应用程序的另一种调试方法是, 在日志文件的每一条日志信息中输出线程名称. 在各种日志输出框架中都广泛的支持这个功能. 在使用协程时, 仅有线程名称还不足以确定协程的上下文, 因此 `kotlinx.coroutines` 包含了一些调试工具来方便我们的调试工作.

请使用 JVM 选项 `-Dkotlinx.coroutines.debug` 来运行下面的示例程序:

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}]
$msg")

fun main() = runBlocking<Unit> {
//sampleStart
    val a = async {
        log("I'm computing a piece of the answer")
        6
    }
    val b = async {
        log("I'm computing another piece of the answer")
        7
    }
    log("The answer is ${a.await() * b.await()}")
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-03.kt).

上面的例子中会出现 3 个协程. `runBlocking` 之内的主协程 (#1), 以及另外 2 个计算延迟值的协程 `a` (#2) 和 `b` (#3). 这些协程都在 `runBlocking` 的上下文内运行, 并且都被限定在主线程中. 这个示例程序的输出是:

```
[main @coroutine#2] I'm computing a piece of the answer
[main @coroutine#3] I'm computing another piece of the answer
[main @coroutine#1] The answer is 42
```

`log` 函数会在方括号内输出线程名称, 你可以看到, 是 `main` 线程, 而且线程名称之后还加上了目前正在执行的协程 id. 当打开调试模式时, 会将所有创建的协程 id 设置为连续的数字顺序.

> 当使用 `-ea` 参数运行 JVM 时, 也会打开调试模式. 关于调试工具的详情, 请参见 DEBUG_PROPERTY_NAME (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-

coroutines-core/kotlinx.coroutines/-d-e-b-u-g_-p-r-o-p-e-r-t-y_-n-a-m-e.html)
属性的文档.

在线程间跳转

请使用 JVM 参数 -Dkotlinx.coroutines.debug 运行下面的示例程序 (参见 debug):

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}]
$msg")

fun main() {
//sampleStart
    newSingleThreadContext("Ctx1").use { ctx1 ->
        newSingleThreadContext("Ctx2").use { ctx2 ->
            runBlocking(ctx1) {
                log("Started in ctx1")
                withContext(ctx2) {
                    log("Working in ctx2")
                }
                log("Back to ctx1")
            }
        }
    }
//sampleEnd
}
```

完整的代码请参见 这里
(https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-04.kt).

上面的示例程序演示了几种技巧. 一是使用明确指定的上下文来调用 runBlocking
(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html), 另一个技巧是使用 withContext
(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-

core/kotlinx.coroutines/with-context.html) 函数, 在同一个协程内切换协程的上下文, 运行结果如下, 你可以看到切换上下文的效果:

```
[Ctx1 @coroutine#1] Started in ctx1
[Ctx2 @coroutine#1] Working in ctx2
[Ctx1 @coroutine#1] Back to ctx1
```

注意, 这个示例程序还使用了 Kotlin 标准库的 `use` 函数, 以便在 newSingleThreadContext (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/new-single-thread-context.html) 创建的线程不再需要的时候释放它.

在上下文中的任务

协程的 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) 是协程上下文的一部分, 而且可以通过自己的上下文来访问到 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html), 方法是使用 `coroutineContext[Job]` 表达式:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    println("My job is ${coroutineContext[Job]}")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-05.kt).

在 调试模式 下运行时, 这个示例程序的输出类似于:

```
My job is "coroutine#1":BlockingCoroutine{Active}@6d311334
```

注意, CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 中的 isActive

(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/is-active.html) 只是 `coroutineContext[Job]?.isActive == true` 的一个简写.

协程的子协程

当一个协程在另一个协程的 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 内启动时, 它会通过 CoroutineScope.coroutineContext (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/coroutine-context.html) 继承这个协程的上下文, 并且新协程的 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) 会成为父协程的任务的一个 *子任务*. 当父协程被取消时, 它所有的子协程也会被取消, 并且会逐级递归, 取消子协程的子协程.

但是, 可以通过以下两种方法明确改变这种父-子关系:

1. 如果启动协程时明确指定了当不同的作用范围(比如, `GlobalScope.launch`), 那么协程不会从父协程继承 `Job`.
2. 如果传递了不同的 `Job` 对象作为新协程的 context 参数(参见下面的示例程序), 那么这个参数会覆盖父 scope 的 `Job`.

以上两种情况, 启动的协程都不会被绑定到启动它的那段代码的作用范围, 并会独自运行.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    // 启动一个协程, 处理某种请求
    val request = launch {
        // 它启动 2 个其他的任务
        launch(Job()) {
            println("job1: I run in my own Job and execute
independently!")
            delay(1000)
            println("job1: I am not affected by cancellation of the
request")
        }
        // 另一个继承父协程的上下文
```

```
        launch {
            delay(100)
            println("job2: I am a child of the request coroutine")
            delay(1000)
            println("job2: I will not execute this line if my parent
request is cancelled")
        }
    }
    delay(500)
    request.cancel() // 取消对请求的处理
    println("main: Who has survived request cancellation?")
    delay(1000) // 将主线程延迟 1 秒, 看看结果如何
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-06.kt).

这个示例程序的运行结果是:

```
job1: I run in my own Job and execute independently!
job2: I am a child of the request coroutine
main: Who has survived request cancellation?
job1: I am not affected by cancellation of the request
```

父协程的职责

父协程总是会等待它的所有子协程运行完毕. 父协程不必明确地追踪它启动的子协程, 也不必使用 Job.join (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/join.html) 来等待子协程运行完毕:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    // 启动一个协程, 处理某种请求
```

```
    val request = launch {
        repeat(3) { i -> // 启动几个子协程
            launch  {
                delay((i + 1) * 200L) // 各个子协程分别等待 200ms,
400ms, 600ms
                println("Coroutine $i is done")
            }
        }
        println("request: I'm done and I don't explicitly join my
children that are still active")
    }
    request.join() // 等待 request 协程执行完毕, 包括它的所有子协程
    println("Now processing of the request is complete")
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-07.kt).

这个示例程序的运行结果如下:

```
request: I'm done and I don't explicitly join my children that are
still active
Coroutine 0 is done
Coroutine 1 is done
Coroutine 2 is done
Now processing of the request is complete
```

为协程命名以便于调试

如果协程频繁输出日志, 而且你只需要追踪来自同一个协程的日志, 那么使用系统自动赋予的协程 id 就足够了. 然而, 如果协程与某个特定的输入处理绑定在一起, 或者负责执行某个后台任务, 那么最好明确地为协程命名, 以便于调试. 对协程来说, 上下文元素 CoroutineName (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-name/index.html) 起到与线程名类似的作用. 当 调试模式 开启时, 协程名称会包含在正在运行这个协程的线程的名称内.

下面的示例程序演示这个概念:

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}]
$msg")

fun main() = runBlocking(CoroutineName("main")) {
//sampleStart
    log("Started main coroutine")
    // 启动 2 个背景任务
    val v1 = async(CoroutineName("v1coroutine")) {
        delay(500)
        log("Computing v1")
        252
    }
    val v2 = async(CoroutineName("v2coroutine")) {
        delay(1000)
        log("Computing v2")
        6
    }
    log("The answer for v1 / v2 = ${v1.await() / v2.await()}")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-08.kt).

使用 JVM 参数 `-Dkotlinx.coroutines.debug` 运行这个示例程序时, 输出类似于以下内容:

```
[main @main#1] Started main coroutine
[main @v1coroutine#2] Computing v1
[main @v2coroutine#3] Computing v2
[main @main#1] The answer for v1 / v2 = 42
```

组合上下文中的元素

有些时候我们会需要对协程的上下文定义多个元素. 这时我们可以使用 `+` 操作符. 比如, 我们可以同时使用明确指定的派发器, 以及明确指定的名称, 来启动一个协程:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch(Dispatchers.Default + CoroutineName("test")) {
        println("I'm working in thread
${Thread.currentThread().name}")
    }
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-09.kt).

使用 JVM 参数 `-Dkotlinx.coroutines.debug` 运行这个示例程序时, 输出结果是:

```
I'm working in thread DefaultDispatcher-worker-1 @test#2
```

协程的作用范围(Scope)

下面我们把上下文, 子协程, 任务的相关知识综合起来. 假设我们的应用程序中有一个对象, 它存在一定的生命周期, 但这个对象不是一个协程. 比如, 我们在编写一个 Android 应用程序, 在一个 Android activity 的上下文内启动了一些协程, 执行一些异步操作, 来取得并更新数据, 显示动画, 等等等等. 当 activity 销毁时, 所有这些协程都必须取消, 以防内存泄漏. 我们当然可以手动操纵上下文和任务, 来将 activity 和它的协程的生命周期关联在一起, 但是 `kotlinx.coroutines` 提供了一种抽象机制来封装这种任务: CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html). 你应该已经熟悉了协程的作用范围概念, 所有的协程构建器都定义为作用范围的扩展函数.

我们创建 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 的实例, 并将它与 activity 的生命周期相

关联, 以此来管理协程的生命周期. `CoroutineScope` 的实例可以通过 CoroutineScope() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope.html) 或 MainScope() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-main-scope.html) 工厂函数来创建. 前一个函数会创建一个通常目的的作用范围, 后一个函数会创建一个用于 UI 应用程序的作用范围, 并且使用 Dispatchers.Main (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-main.html) 作为默认的派发器:

```
class Activity {
    private val mainScope = MainScope()

    fun destroy() {
        mainScope.cancel()
    }
    // 以下代码省略 ...
```

下面, 我们在这个 `Activity` 之内使用指定的 `scope` 来启动协程. 在这个示例程序中, 我们启动 10 个协程, 分别延迟一段不同长度的时间:

```
// Activity 类的内容继续
    fun doSomething() {
        // 启动 10 个协程, 每个工作一段不同长度的时间
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // 分别延迟 200ms, 400ms, ... 等等
                println("Coroutine $i is done")
            }
        }
    }
} // Activity 类结束
```

在我们的 main 函数中, 我们创建 activity, 调用我们的 `doSomething` 测试函数, 然后在 500ms 后销毁 activity. 销毁 activity 会取消 `doSomething` 之内启动的所有协程. activity 销毁之后, 即使再等待一段时间, 协程也不再向屏幕输出信息, 因此我们能够看出协程已经被取消了.

```
import kotlinx.coroutines.*

class Activity {
    private val mainScope = CoroutineScope(Dispatchers.Default) //
为测试目的，这里使用默认派发器

    fun destroy() {
        mainScope.cancel()
    }

    fun doSomething() {
        // 启动 10 个协程，每个工作一段不同长度的时间
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // 分别延迟 200ms, 400ms, ... 等
等
                println("Coroutine $i is done")
            }
        }
    }
} // Activity 类结束

fun main() = runBlocking<Unit> {
//sampleStart
    val activity = Activity()
    activity.doSomething() // 运行测试函数
    println("Launched coroutines")
    delay(500L) // 等待半秒
    println("Destroying activity!")
    activity.destroy() // 取消所有协程
    delay(1000) // 确认协程不再继续工作
//sampleEnd
}
```

完整的代码请参见 这里
(https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-

core/jvm/test/guide/example-context-10.kt).

这个示例程序的输出如下:

```
Launched coroutines
Coroutine 0 is done
Coroutine 1 is done
Destroying activity!
```

你会看到, 只有前面的 2 个协程输出了信息, 由于 `Activity.destroy()` 中调用了 `job.cancel()`, 其他所有协程都被取消了.

> 注意, Android 对于协程作用范围的整个生命周期提供了一类支持(first-party support). 详情请参见 相应的文档 (https://developer.android.com/topic/libraries/architecture/coroutines#lifecyclescope).

线程的局部数据

有些时候, 如果能够向协程传递, 或者在协程直接传递一些线程局部的数据(thread-local data), 将是一种很方便的功能, 但是, 协程并没有关联到某个具体的线程, 因此, 如果自己写代码来实现这种功能, 可能会导致大量的样板代码.

对于 `ThreadLocal` (https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html), 有一个扩展函数 asContextElement (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/as-context-element.html) 可以帮助我们. 它会创建一个额外的上下文元素, 用来保持某个给定的 `ThreadLocal` 的值, 并且每次当协程切换上下文时就恢复它的值.

我们通过一个例子来演示如何使用这个函数:

```
import kotlinx.coroutines.*

val threadLocal = ThreadLocal<String?>() // 声明线程局部变量

fun main() = runBlocking<Unit> {
//sampleStart
    threadLocal.set("main")
    println("Pre-main, current thread: ${Thread.currentThread()},
```

```
thread local value: '${threadLocal.get()}'")
    val job = launch(Dispatchers.Default +
threadLocal.asContextElement(value = "launch")) {
        println("Launch start, current thread:
${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
        yield()
        println("After yield, current thread:
${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
    }
    job.join()
    println("Post-main, current thread: ${Thread.currentThread()},
thread local value: '${threadLocal.get()}'")
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-context-11.kt).

在这个示例程序中, 我们使用 Dispatchers.Default (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html), 在后台线程池中启动了一个新的协程, 因此协程会在线程池的另一个线程中运行, 但它还是会得到我们通过 `threadLocal.asContextElement(value = "launch")` 指定的线程局部变量的值, 无论协程运行在哪个线程内. 因此, (使用 调试模式时)的输出结果是:

```
Pre-main, current thread: Thread[main @coroutine#1,5,main], thread
local value: 'main'
Launch start, current thread: Thread[DefaultDispatcher-worker-1
@coroutine#2,5,main], thread local value: 'launch'
After yield, current thread: Thread[DefaultDispatcher-worker-2
@coroutine#2,5,main], thread local value: 'launch'
Post-main, current thread: Thread[main @coroutine#1,5,main], thread
local value: 'main'
```

很容易会忘记设置对应的上下文元素. 如果这个协程由另一个线程执行, 那么从协程中访问线程局部变量可能会得到一个意想不到的值. 为了避免这样的情况, 推荐使用 ensurePresent (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/ensure-present.html) 方法, 并且在不当使用时尽快失败(fail-fast).

`ThreadLocal` 在协程中得到了一级支持, 可以在 `kotlinx.coroutines` 提供的所有基本操作一起使用. 但它还是有一个关键的限制: 当线程局部变量的值发生变化时, 新值不会传递到调用协程的线程中去 (因为上下文元素不能追踪对 `ThreadLocal` 对象的所有访问) 而且更新后的值会在下次挂起时丢失. 请在协程内使用 withContext (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-context.html) 来更新线程局部变量的值, 详情请参见 asContextElement (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/as-context-element.html).

另一种方法是, 值可以保存在可变的装箱类(mutable box)中, 比如 `class Counter(var i: Int)`, 再把这个装箱类保存在线程局部变量中. 然而, 这种情况下, 对这个装箱类中的变量可能发生并发修改, 你必须完全负责对此进行同步控制.

对于高级的使用场景, 比如与日志 MDC(Mapped Diagnostic Context) 的集成, 与事务上下文 (transactional context)的集成, 或者与其他内部使用线程局部变量来传递数据的库的集成, 应该实现 ThreadContextElement (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-thread-context-element/index.html) 接口, 详情请参见这个接口的文档.

异步的数据流(Asynchronous Flow)

最终更新: 2024/09/10

一个挂起函数可以异步地返回单个结果值, 但我们要如何才能返回多个异步计算的结果值? 这就是 Kotlin 的异步数据流要解决的问题.

多个值的表达

在 Kotlin 中, 多个值可以使用 集合 (集合(Collection)概述) 表达. 比如, 我们可以通过 `simple` 函数返回一个 List (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/), 其中包含 3 个数值, 然后使用 forEach (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/for-each.html) 输出这些数值:

```
fun simple(): List<Int> = listOf(1, 2, 3)

fun main() {
    simple().forEach { value -> println(value) }
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-01.kt).

这段代码的输出是:

```
1
2
3
```

序列(Sequence)

如果我们需要通过某些非常消耗 CPU 的阻塞性代码来计算这些数值(每个数值的计算消耗 100ms), 那么我们可以使用 Sequence (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/) 来表达这些数值:

```kotlin
fun simple(): Sequence<Int> = sequence { // 序列的构建器
    for (i in 1..3) {
        Thread.sleep(100) // 假设这里是数值的计算
        yield(i) // 产生下一个值
    }
}

fun main() {
    simple().forEach { value -> println(value) }
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-02.kt).

这段代码输出的数值与前面相同, 但它要在输出每个数值之前等待 100ms.

挂起函数(Suspending function)

但是, 数值的计算过程会阻塞运行这段代码的主线程. 如果这些数值由异步代码计算, 我们可以对 `simple` 函数添加 `suspend` 标记, 这样这个函数就可以执行它的工作, 而不会发生阻塞, 而且还能将结果返回为 list:

```kotlin
import kotlinx.coroutines.*

//sampleStart
suspend fun simple(): List<Int> {
    delay(1000) // 假设这里在执行某些异步操作
    return listOf(1, 2, 3)
}

fun main() = runBlocking<Unit> {
    simple().forEach { value -> println(value) }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-03.kt).

这段代码会等待 1 秒, 然后输出数值.

数据流(Flow)

使用 `List<Int>` 作为结果类型, 代码我们只能一次性返回所有的结果值. 为了表达异步计算的多个结果值构成的流(stream), 我们可以使用 `Flow<Int>` (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) 类型, 就象对同步计算的结果值使用 `Sequence<Int>` 类型一样:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow { // 数据流构建器
    for (i in 1..3) {
        delay(100) // 假设我们在这里进行某些计算工作
        emit(i) // 发射(emit)下一个值
    }
}

fun main() = runBlocking<Unit> {
    // 启动一个并发的协程, 检查主线程是否被阻塞
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // 收取(collect)流中的内容
    simple().collect { value -> println(value) }
}
//sampleEnd
```

> 完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-04.kt).

这段代码会在输出每个数值之前等待 100ms, 而不会阻塞主线程. 在主线程中运行的另一个独立的协程中, 每隔 100ms 会输出 "I'm not blocked" 消息, 因此可以确定主线程没有被阻塞:

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

请注意, 使用 Flow (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) 的代码与前面的示例代码之间的区别如下:

- Flow (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) 类型的构建器函数叫做 flow (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow.html).
- `flow { ... }` 构建器代码段之内的代码可以挂起.
- `simple` 函数不再带有 `suspend` 标识符.
- 使用 emit (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow-collector/emit.html) 函数, 从流中 *发射(emit)* 值.
- 使用 collect (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect.html) 函数, 从流中 *收取(collect)* 值.

> 在 `simple` 函数的 `flow { ... }` 代码段之内, 我们可以将 delay (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/delay.html) 替换为 `Thread.sleep`, 这时可以看到主线程会被阻塞.

数据流(Flow)是 "冷的"(cold)

数据流类似于 sequence, 但它是 *"冷的"(cold)* 流 — 直到流中的数据被收集时, 才会执行flow (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow.html) 构建器之内的代码. 下面的示例程序可以演示这个特性:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    println("Flow started")
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    println("Calling simple function...")
    val flow = simple()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-05.kt).

这段代码的输出是:

```
Calling simple function...
Calling collect...
Flow started
```

```
1
2
3
Calling collect again...
Flow started
1
2
3
```

simple 函数 (负责返回一个数据流) 不使用 suspend 标记符, 关键原因在这里. 对 simple() 的调用本身会立即返回, 不会等待任何任务. 数据流会在每次被收集的时候启动, 所以, 每次调用 collect 时我们都会再次看到 "Flow started" 消息的输出.

简要介绍数据流的取消

数据流的取消使用协程通常的协作取消机制. 和通常的机制一样, 如果数据流在一个可取消的挂起函数(比如 delay (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/delay.html))之内被挂起, 那么数据流的收集可以取消. 下面的示例程序会演示, 在 withTimeoutOrNull (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-timeout-or-null.html) 代码段之内运行时, 如果发生超时, 数据流会被取消, 并停止执行它的代码:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    withTimeoutOrNull(250) { // 250ms 后超时
        simple().collect { value -> println(value) }
    }
    println("Done")
```

```
}
//sampleEnd
```

> 完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-06.kt).

注意, `simple` 函数内的数据流只发射(emit)了 2 个数值, 最终的输出结果如下:

```
Emitting 1
1
Emitting 2
2
Done
```

更多详情, 请参见 检查数据流的取消 小节.

数据流构建器

前面的示例代码中使用的 `flow { ... }` 构建器是最基本的数据流构建器. 还有其他一些构建器可以声明数据流:

- flowOf (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow-of.html) 构建器, 定义一个数据流, 发射一组固定的值.
- 使用 `.asFlow()` 扩展函数, 可以将各种集合(collection)和序列(sequence)转换为数据流.

例如, 从数据流输出数值 1 到 3 的那段代码, 可以重写为以下代码:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    // 将整数范围(range)转换为数据流
    (1..3).asFlow().collect { value -> println(value) }
```

```
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-07.kt).

数据流的中间操作符(Intermediate flow operator)

数据流可以使用操作符进行变换, 与对集合(collection)和序列(sequence)进行变换的方式一样. 中间操作符(Intermediate operator) 应用于上游的数据流(upstream flow), 然后返回一个下游数据流(downstream flow). 与数据流一样, 这些操作符也是"冷的"(cold). 这样的操作符调用本身不是挂起函数. 它的工作会快速结束, 返回结果是, 变换后的数据流的定义.

基本的操作符的名称与 map (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/map.html) 和 filter (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/filter.html) 类似. 与序列的操作符的一个重要区别在于, 数据流的这些操作符之内的代码段可以调用挂起函数.

比如, 一个包含请求的数据流, 可以使用 map (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/map.html) 操作符映射为结果值, 即使一个请求的执行是由挂起函数实现的长时间的操作:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
suspend fun performRequest(request: Int): String {
    delay(1000) // 假设这里是一个长时间的异步工作
    return "response $request"
}

fun main() = runBlocking<Unit> {
    (1..3).asFlow() // 由请求构成的数据流
        .map { request -> performRequest(request) }
        .collect { response -> println(response) }
```

```
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-08.kt).

这段代码会输出以下 3 行, 各行之间等待 1 秒:

```
response 1
response 2
response 3
```

变换操作符(Transform operator)

数据流的变换操作符中, 最常用的就是 transform (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/transform.html). 它可以用来实现简单的变换, 比如 map (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/map.html) 和 filter (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/filter.html), 也可以实现更复杂的变换. 使用 `transform` 操作符, 我们可以 发射(emit) (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow-collector/emit.html) 任意次数的任意值.

比如, 使用 `transform` 我们可以在执行一个长时间运行的异步请求之前发射一个字符串, 之后再发射一个应答结果:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun performRequest(request: Int): String {
    delay(1000) // 假设这里是一个长时间运行的异步任务
    return "response $request"
}

fun main() = runBlocking<Unit> {
//sampleStart
```

```
    (1..3).asFlow() // 由请求构成的数据流
        .transform { request ->
            emit("Making request $request")
            emit(performRequest(request))
        }
        .collect { response -> println(response) }
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-09.kt).

这段代码的输出是:

```
Making request 1
response 1
Making request 2
response 2
Making request 3
response 3
```

限制大小操作符(Size-limiting operator)

限制大小(Size-limiting) 的中间操作符, 比如 take (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/take.html), 在达到相应的大小限制之后, 会取消数据流的执行. 协程的取消总是通过抛出异常来实现的, 因此, 在协程取消时, 所有的资源管理函数 (比如 `try { ... } finally { ... }` 代码段) 都能够正常工作:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun numbers(): Flow<Int> = flow {
    try {
        emit(1)
        emit(2)
```

```
        println("This line will not execute")
        emit(3)
    } finally {
        println("Finally in numbers")
    }
}

fun main() = runBlocking<Unit> {
    numbers()
        .take(2) // 只获取最前面的 2 个值
        .collect { value -> println(value) }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-10.kt).

这段代码的输出清楚的表面, 在 `numbers()` 函数中, `flow { ... }` 代码体, 会在发射第 2 个数值之后停止执行:

```
1
2
Finally in numbers
```

数据流的结束操作符(Terminal flow operator)

数据流上的结束操作符(Terminal operator)是 *挂起函数*, 它会开始收集数据流中的值. 最基本的结束操作符是 collect (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect.html), 但还有其他结束操作符, 可以方便地实现以下功能:

- 转换为各种集合, 比如 toList (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/to-list.html) 和 toSet (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/to-set.html).

- 取得 first (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/first.html) 值的操作符, 而且会确保数据流发射 single (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/single.html) 值.
- 使用 reduce (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/reduce.html) 和 fold (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/fold.html), 将数据流压缩为单个值.

比如:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val sum = (1..5).asFlow()
        .map { it * it } // 从 1 到 5 的平方
        .reduce { a, b -> a + b } // 求和 (结束操作符)
    println(sum)
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-11.kt).

最终结果是单个数值:

```
55
```

数据流的执行是顺序的(sequential)

数据流的每次单独的收集操作会顺序的执行, 除非使用了特殊的操作符, 比如对多个数据流进行操作. 收集操作直接在调用结束操作符的协程内工作. 默认不会启动新的协程. 每个发射的值, 会由从上游数据流到下游数据流的, 所有的中间操作符处理, 之后, 发送给结束操作符.

请看下面的示例程序，它会过滤偶数，然后映射为字符串：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    (1..5).asFlow()
        .filter {
            println("Filter $it")
            it % 2 == 0
        }
        .map {
            println("Map $it")
            "string $it"
        }.collect {
            println("Collect $it")
        }
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-12.kt).

输出结果为：

```
Filter 1
Filter 2
Map 2
Collect string 2
Filter 3
Filter 4
Map 4
Collect string 4
Filter 5
```

数据流的上下文(context)

数据流的收集工作总是会在调用收集函数的协程的上下文中执行. 比如, 如果存在一个数据流 `simple`, 那么不管数据流 `simple` 的具体实现细节如何, 以下代码总是会在这段代码中指定的上下文中执行:

```
withContext(context) {
    simple().collect { value ->
        println(value) // 在指定的上下文中执行
    }
}
```

数据流的这种特性称为 *上下文保留(context preservation)*.

因此, 默认情况下 `flow { ... }` 构建器中的代码, 会在由对应的数据流的收集器所提供的上下文中运行. 比如, 假设 `simple` 函数的实现会输出调用它的线程名称, 然后发射 3 个数值:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun log(msg: String) = println("[${Thread.currentThread().name}]
$msg")

//sampleStart
fun simple(): Flow<Int> = flow {
    log("Started simple flow")
    for (i in 1..3) {
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> log("Collected $value") }
}
//sampleEnd
```

> 完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-

core/jvm/test/guide/example-flow-13.kt).

这段代码的输出是:

```
[main @coroutine#1] Started simple flow
[main @coroutine#1] Collected 1
[main @coroutine#1] Collected 2
[main @coroutine#1] Collected 3
```

由于调用 simple().collect 的是主线程, simple 的数据流的代码体也由主线程调用. 对于快速执行的代码, 或异步执行的代码, 如果不关心执行时的上下文, 并且不阻塞调用者, 这是非常完美的默认动作.

使用 withContext 时的一个常见陷阱

但是, 对于长时间运行, 非常消耗 CPU 的代码, 可能需要在 Dispatchers.Default (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html) 上下文内执行, 而 UI 更新代码需要在 Dispatchers.Main (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-main.html) 上下文内执行. 通常, 使用 Kotlin 协程的代码可以通过 withContext (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-context.html) 来切换上下文, 但在 flow { ... } 构建器内的代码必须服从数据流的上下文保留(context preservation)特性, 因此不允许在不同的上下文内执行 emit (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow-collector/emit.html).

试试运行以下代码:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    // 在数据流构建器中, 对非常消耗 CPU 的代码切换上下文的错误方式
    kotlinx.coroutines.withContext(Dispatchers.Default) {
        for (i in 1..3) {
            Thread.sleep(100) // 假设我们在这里执行非常消耗 CPU 的计算过程
            emit(i) // 发射下一个值
        }
```

```
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> println(value) }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-14.kt).

这段代码会产生以下异常:

```
Exception in thread "main" java.lang.IllegalStateException: Flow
invariant is violated:
        Flow was collected in [CoroutineId(1),
"coroutine#1":BlockingCoroutine{Active}@5511c7f8,
BlockingEventLoop@2eac3323],
        but emission happened in [CoroutineId(1),
"coroutine#1":DispatchedCoroutine{Active}@2dae0000,
Dispatchers.Default].
        Please refer to 'flow' documentation or use 'flowOn' instead
    at ...
```

flowOn 操作符

这个异常告诉我们, 应该使用 flowOn (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow-on.html) 函数来切换发射数据时的上下文. 我们在下面的示例程序中演示切换数据流上下文的正确方式, 它会输出对应的线程名称, 演示它的工作方式:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun log(msg: String) = println("[${Thread.currentThread().name}]
$msg")
```

```
//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        Thread.sleep(100) // 假设我们在这里执行非常消耗 CPU 的计算过程
        log("Emitting $i")
        emit(i) // 发射下一个值
    }
}.flowOn(Dispatchers.Default) // 在数据流构建器中, 对非常消耗 CPU 的代码切换上下文的正确方式

fun main() = runBlocking<Unit> {
    simple().collect { value ->
        log("Collected $value")
    }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-15.kt).

注意 `flow { ... }` 工作在后台线程中, 而数据收集发生在主线程中:

另外还值得注意的是, flowOn (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow-on.html) 操作符改变了数据流默认的顺序性(sequential)特性. 现在, 数据的收集发生在一个线程内("coroutine#1"), 而数据的发射发生在另一个协程内("coroutine#2"), 而且发射协程在另一个线程内, 与收集协程并行执行. 当上游数据流在它的上下文内需要切换 CoroutineDispatcher (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-dispatcher/index.html) 时, flowOn (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow-on.html) 操作符为它创建了另一个协程.

缓冲(Buffering)

让数据流的不同部分在不同的协程中执行, 收集数据流所耗费的总时间可能会有所改进, 尤其是涉及长时间运行的异步操作的情况. 比如, 假设 数据流 `simple` 的数据发射操作很慢, 每产生一个元素需

要 100 ms; 而数据收集操作也很慢, 处理每个元素需要 300 ms. 那么我们来看看, 从这样的数据流收集 3 个数值需要多长时间:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 假设这里的异步操作需要等待 100 ms
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        simple().collect { value ->
            delay(300) // 假设处理值需要 300 ms
            println(value)
        }
    }
    println("Collected in $time ms")
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-16.kt).

输出的结果类似以下内容, 整个数据流的收集过程需要大约 1200 ms (3 个数值, 每个需要 400 ms):

```
1
2
3
Collected in 1220 ms
```

我们可以对数据流使用 buffer (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/buffer.html) 操作符, 让数据流 `simple` 的数据发射代码, 与数据收集代码并行执行, 而不是让它们顺序执行:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 假设这里的异步操作需要等待 100 ms
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .buffer() // 对数据发射进行缓冲, 不要等待
            .collect { value ->
                delay(300) // 假设处理值需要 300 ms
                println(value)
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-17.kt).

这段代码会输出同样的数值, 但运行速度更快, 因为我们实际上创建了数据的处理管道(processing pipeline), 只对第一个数值需要等待 100 ms, 然后对每个数值的处理花费 300 ms. 这种方式下, 整个运行过程花费大约 1000 ms:

```
1
2
3
Collected in 1071 ms
```

> 注意, flowOn (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow-on.html) 操作符在需要切换 CoroutineDispatcher (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-dispatcher/index.html), 会使用相同的缓冲机制, 但在这里, 我们明确的要求使用缓冲, 而不要切换协程执行的上下文.

合并(Conflation)

如果一个数据流只代表操作结果(或操作状态变更)的一部分, 可能没有必要处理每一个结果值, 而可以只处理最近的一部分结果. 这种情况下, 如果收集器速度太慢无法快速处理数据流中的所有值, 可以使用 conflate (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/conflate.html) 操作符跳过中间值. 在前面的示例程序的基础上, 我们可以编写这样的代码:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 假设这里的异步操作需要等待 100 ms
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .conflate() // 对发射操作进行合并, 并不处理每一个值
            .collect { value ->
                delay(300) // 假设处理值需要 300 ms
```

```
                println(value)
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-18.kt).

我们可以看到, 当第 1 个数值还在处理时, 第 2 个和第 3 个数值已经产生了, 因此第 2 个数值 *被合并 (conflated)*, 于是只有最近的(第 3 个数值) 被发送给了收集器:

```
1
3
Collected in 758 ms
```

处理最后的值

当数据的发射端和收集端都非常慢的时候, 合并(Conflation) 是提高处理速度的方法之一. 它的实现方法是丢弃发射的值. 另一种方法是, 每当数据流发射新值时, 将运行缓慢(未能即使处理完成)的收集器取消, 然后重新启动收集器. 有一组 `xxxLatest` 操作符, 它们执行与 `xxx` 操作符相同的逻辑, 区别在于, 如果出现新值则会取消它代码体中的代码. 我们试试将前面的示例程序中的 conflate (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/conflate.html) 替换为 collectLatest (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect-latest.html):

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 假设这里的异步操作需要等待 100 ms
        emit(i) // 发射下一个值
    }
```

```
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .collectLatest { value -> // 对前面的值取消收集代码, 并最后一个值重新执行
                println("Collecting $value")
                delay(300) // 假设处理值需要 300 ms
                println("Done $value")
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

> 完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-19.kt).

由于 collectLatest (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect-latest.html) 的代码体执行需要 300 ms, 而每隔 100 ms 就会发射新的值, 所以我们会看到代码体会对每个值执行, 但只对最后一个值执行完毕:

```
Collecting 1
Collecting 2
Collecting 3
Done 3
Collected in 741 ms
```

多个数据流的组合

有很多种方法可以组合多个数据流.

Zip

就象 Kotlin 标准库中的 Sequence.zip
(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/zip.html) 扩展函数一样, 数据流也有一个 zip (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/zip.html) 操作符, 可以将两个数据流中相应的值组合在一起:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow() // 数值 1..3
    val strs = flowOf("one", "two", "three") // 字符串
    nums.zip(strs) { a, b -> "$a -> $b" } // 组合为一个字符串
        .collect { println(it) } // 收集最后结果, 并输出
//sampleEnd
}
```

完整的示例代码请参见 这里
(https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-20.kt).

这段示例程序的输出结果是:

```
1 -> one
2 -> two
3 -> three
```

结合(Combine)

如果数据流代表一个变量的最近的值, 或者一个操作的最近的结果(参见相关小节 合并(Conflation)), 那么有可能需要根据相应的数据流中最近的值进行某种计算, 而且当某个上游数据流发射新值时, 又需要重新计算. 这组对应的操作符称为 combine
(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/combine.html).

比如, 如果前面示例程序中的数值每 300ms 更新一次, 而字符串每 400ms 更新一次, 那么使用 zip
(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-

core/kotlinx.coroutines.flow/zip.html) 操作符组合它们, 还是会产生相同的结果, 然而结果需要每 400ms 输出一次:

> 在这个示例程序中, 我们使用 onEach (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/on-each.html) 中间操作符实现每个元素的延迟, 让示例数据流的代码更加接近声明式风格, 而且更加简短.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow().onEach { delay(300) } // 数值 1..3, 每
隔 300 ms 发射一个值
    val strs = flowOf("one", "two", "three").onEach { delay(400) }
// 字符串, 每隔 400 ms 发射一个值
    val startTime = System.currentTimeMillis() // 记录开始时刻
    nums.zip(strs) { a, b -> "$a -> $b" } // 使用 "zip", 组合为一个字符
串
        .collect { value -> // 收集最后结果, 并输出
            println("$value at ${System.currentTimeMillis() -
startTime} ms from start")
        }
//sampleEnd
}
```

> 完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-21.kt).

但是, 如果使用 combine (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/combine.html) 操作符, 而不是 zip (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/zip.html):

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    val nums = (1..3).asFlow().onEach { delay(300) } // 数值 1..3, 每隔 300 ms 发射一个值
    val strs = flowOf("one", "two", "three").onEach { delay(400) } // 字符串, 每隔 400 ms 发射一个值
    val startTime = System.currentTimeMillis() // 记录开始时刻
    nums.combine(strs) { a, b -> "$a -> $b" } // 使用 "combine", 组合为一个字符串
        .collect { value -> // 收集最后结果, 并输出
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-22.kt).

我们会得到非常不同的输出结果, 每当 `nums` 或 `strs` 数据流发射一个值, 就会输出一行结果:

```
1 -> one at 452 ms from start
2 -> one at 651 ms from start
2 -> two at 854 ms from start
3 -> two at 952 ms from start
3 -> three at 1256 ms from start
```

压平(Flatten)数据流

数据流代表异步接收的值序列, 因此很容易遇到这种的情况, 每个值触发一个请求, 得到另外一组值. 比如, 假设我们有下面这样的函数, 它返回一个数据流, 发送 2 个字符串, 中间间隔 500ms:

```
fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 ms
    emit("$i: Second")
}
```

然后, 如果我们有一个数据流, 包含 3 个整数, 并对每个整数调用 `requestFlow`, 如下:

```
(1..3).asFlow().map { requestFlow(it) }
```

然后我们会得到一个数据流的数据流 (`Flow<Flow<String>>`), 这样的情况下, 就需要将它 *压平 (Flatten)*, 变为单个数据流, 然后才能进行进一步处理. 集合和序列都有 flatten (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/flatten.html) 和 flatMap (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/flat-map.html) 操作符来实现这样的功能. 但是, 由于数据流的异步特性, 需要使用不同的 *模式(mode)* 来进行压平(Flatten)处理, 因此, 对于数据流, 存在一组压平(Flatten)操作符.

flatMapConcat

将数据流的数据流串联(Concatenate)起来的功能, 由 flatMapConcat (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flat-map-concat.html) 和 flattenConcat (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flatten-concat.html) 操作符提供. 这两个操作符与序列的对应的操作符最类似. 在收集下一个值之前, 它们会等待内层的数据流完成, 如下例所示:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // 记录开始时刻
    (1..3).asFlow().onEach { delay(100) } // 每隔 100 ms 发射一个数值
```

```
        .flatMapConcat { requestFlow(it) }
        .collect { value -> // 收集最后结果, 并输出
            println("$value at ${System.currentTimeMillis() -
startTime} ms from start")
        }
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-23.kt).

输出结果如下, 清楚的显示出 flatMapConcat (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flat-map-concat.html) 顺序执行的特性:

```
1: First at 121 ms from start
1: Second at 622 ms from start
2: First at 727 ms from start
2: Second at 1227 ms from start
3: First at 1328 ms from start
3: Second at 1829 ms from start
```

flatMapMerge

另一种压平操作是, 同时收集所有的输入数据流, 然后将它们的值合并为单个数据流, 因此能够尽可能快的发射最终结果值. 这个模式由 flatMapMerge (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flat-map-merge.html) 和 flattenMerge (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flatten-merge.html) 操作符实现. 这两个操作符都接受一个可选的 `concurrency` 参数, 用来限制允许同时收集的数据流个数 (默认值等于 DEFAULT_CONCURRENCY (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-d-e-f-a-u-l-t_-c-o-n-c-u-r-r-e-n-c-y.html)).

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```
fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // 记录开始时刻
    (1..3).asFlow().onEach { delay(100) } // 每隔 100 ms 发射一个数值
        .flatMapMerge { requestFlow(it) }
        .collect { value -> // 收集最后结果, 并输出
            println("$value at ${System.currentTimeMillis() - startTime} ms from start")
        }
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-24.kt).

输出结果如下, 很清楚的显示出, flatMapMerge (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flat-map-merge.html) 是并发的:

```
1: First at 136 ms from start
2: First at 231 ms from start
3: First at 333 ms from start
1: Second at 639 ms from start
2: Second at 732 ms from start
3: Second at 833 ms from start
```

注意, flatMapMerge (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flat-map-merge.html) 对它的代码体 (上面示例程序中是 `{ requestFlow(it) }`) 的调用是顺序的, 但对结果数据流的收集是并发的, 最后结果等于首先顺序的执行 `map { requestFlow(it) }`, 然后对结果调用 flattenMerge

(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flatten-merge.html).

flatMapLatest

在 "处理最后的值" 小节中我们介绍过 collectLatest (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect-latest.html) 操作符, 与它类似, 有一个对应的 "Latest" 压平模式, 每次发射新的数据流, 对之前的数据流的收集(如果未完成)就会被取消. 这种模式由 flatMapLatest (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flat-map-latest.html) 操作符实现.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 ms
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // 记录开始时刻
    (1..3).asFlow().onEach { delay(100) } // 每隔 100 ms 发射一个数值
        .flatMapLatest { requestFlow(it) }
        .collect { value -> // 收集最后结果, 并输出
            println("$value at ${System.currentTimeMillis() - 
startTime} ms from start")
        }
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-25.kt).

输出结果如下, 清楚的演示了 flatMapLatest (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flat-map-latest.html) 的工作方式:

```
1: First at 142 ms from start
2: First at 322 ms from start
3: First at 425 ms from start
3: Second at 931 ms from start
```

> 注意, 在收到新值时, flatMapLatest (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flat-map-latest.html) 会取消它代码体中的所有代码 (在上面的例子中是 `{ requestFlow(it) }`). 在这个示例中不会产生差别, 因为 `requestFlow` 的调用是很快的, 不会发生挂起, 而且无法取消. 但是, 如果我们在 `requestFlow` 代码体之内使用挂起函数, 比如 `delay`, 那么在输出中就能够看到差别.

数据流的异常

如果发射器或操作符之内的代码抛出异常, 数据流的收集就会异常结束. 有几种方法来处理这些异常.

在收集器中使用 try/catch

收集器可以使用 Kotlin 的 `try/catch` (异常(Exception)) 代码段来处理异常:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
    try {
```

```
        simple().collect { value ->
            println(value)
            check(value <= 1) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-26.kt).

这段代码成功地捕获在 collect (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect.html) 结束操作符中发生的异常, 而且我们看到, 在这个异常之后, 没有发射其他值:

```
Emitting 1
1
Emitting 2
2
Caught java.lang.IllegalStateException: Collected 2
```

一切异常都会被捕获

前面的示例程序实际上会捕获任何异常, 包括发射器之内, 任何中间操作符之内, 以及结束操作符之内发生的一切异常. 比如, 我们来修改一下代码, 将发射的值 映射(map) (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/map.html) 为字符串, 但这段代码会产生一个异常:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
```

```
            println("Emitting $i")
            emit(i) // 发射下一个值
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-27.kt).

这个异常仍然会被捕获, 然后收集处理会停止:

```
Emitting 1
string 1
Emitting 2
Caught java.lang.IllegalStateException: Crashed on 2
```

异常的透明性(transparency)

但是数据流发射器的代码要怎么样才能封装它自己的异常处理逻辑呢?

数据流必须 *对异常透明(transparent to exception)*, 因此, 从 `try/catch` 代码块内部的 `flow { ... }` 构建器中 发射 (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow-collector/emit.html) 值, 是违反异常透明性的. 这个规则保

证了, 如果收集器会抛出异常, 那么总是能够使用 `try/catch` 捕获这些异常, 就象前面的示例程序那样.

发射器可以使用 catch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/catch.html) 操作符, 既能够符合这种异常透明性, 又能够封装它自己的异常处理代码. `catch` 操作符的代码体能够分析异常, 并根据捕获的异常类型作出不同的反应:

- 可以使用 `throw` 再次抛出异常.
- 在 catch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/catch.html) 代码体中使用 emit (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow-collector/emit.html), 可以将异常转换为值的发射.
- 异常可以忽略, 输出到日志, 或被其它代码处理.

比如, 我们可以对捕获的异常, 发射它的文字:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // 发射下一个值
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .catch { e -> emit("Caught $e") } // 根据异常, 发射值
        .collect { value -> println(value) }
```

```
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-28.kt).

这个示例程序的输出是相同的, 尽管我们没有在代码中使用 `try/catch`.

透明捕获(Transparent catch)

catch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/catch.html) 中间操作符遵守异常透明性规则, 只捕获上游数据流中的异常 (也就是在 `catch` 之前的所有操作符中发生的异常, 但不包含 `catch` 之后的). 如果 `collect { ... }` 之内的代码 (位置在 `catch` 之后) 抛出了异常, 那么这个异常不会被捕获:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple()
        .catch { e -> println("Caught $e") } // 不会捕捉下游数据流中的异
常
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-29.kt).

尽管存在 `catch` 操作符, 但这段示例代码不会输出 "Caught ..." 消息:

```
Emitting 1
1
Emitting 2
Exception in thread "main" java.lang.IllegalStateException:
Collected 2
    at ...
```

声明式异常捕捉

我们能够将 catch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/catch.html) 操作符的声明式特性, 与处理所有的异常的需求结合在一起, 方法是将 collect (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect.html) 操作符的代码体移动到 onEach (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/on-each.html) 之内, 并放在 `catch` 操作符之前. 然后, 需要通过不带参数调用 `collect()`, 来触发对这个数据流的收集:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .onEach { value ->
            check(value <= 1) { "Collected $value" }
```

```
            println(value)
        }
        .catch { e -> println("Caught $e") }
        .collect()
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-30.kt).

现在我们可以看到, 会输出 "Caught ..." 消息, 因此我们可以捕获所有的异常, 而不需要明确使用 `try/catch` 代码块:

```
Emitting 1
1
Emitting 2
Caught java.lang.IllegalStateException: Collected 2
```

数据流的完成

当数据流的收集完成时 (无论是正常完成, 还是异常完成), 它可能会需要执行某种操作. 你可能以及注意到了, 可以通过两种方式实现: 命令式, 或声明式.

命令式的 finally 代码块

除了 `try`/`catch` 之外, 收集器还可以使用 `finally` 代码块, 在 `collect` 完成时执行某种操作.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } finally {
```

```
            println("Done")
        }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-31.kt).

这段代码会输出 `simple` 数据流产生的 3 个数值, 之后输出一个 "Done" 字符串:

```
1
2
3
Done
```

声明式的完成处理

对于声明风格的方式, 数据流有 onCompletion (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/on-completion.html) 中间操作符, 当数据流收集完成时会调用它.

前面的示例程序可以使用 onCompletion (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/on-completion.html) 操作符改写如下, 输出结果相同:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .onCompletion { println("Done") }
        .collect { value -> println(value) }
//sampleEnd
}
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-32.kt).

onCompletion (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/on-completion.html) 的重要优点是, Lambda 表达式可以接受一个可为 null 的 `Throwable` 参数, 通过这个参数来确定数据流的收集是正常完成还是异常完成. 在下面的示例程序中, `simple` 数据流在发射数值 1 之后会抛出一个异常:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    emit(1)
    throw RuntimeException()
}

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> if (cause != null) println("Flow completed exceptionally") }
        .catch { cause -> println("Caught exception") }
        .collect { value -> println(value) }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-33.kt).

如你希望的那样, 这段代码的输出是:

```
1
Flow completed exceptionally
```

```
Caught exception
```

onCompletion (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/on-completion.html) 操作符, 与 catch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/catch.html) 不同, 不会处理异常. 从上面的示例我们可以看到, 异常仍然会流向下游. 它会被发送到更远的 `onCompletion` 操作符, 也可以由使用 `catch` 操作符来处理.

数据流的成功完成

与 catch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/catch.html) 操作符的另一个不同在于, onCompletion (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/on-completion.html) 可以收到所有的异常, 而且只有在上游数据流成功完成(没有取消, 也没有失败)的情况下, 才会收到一个 `null` 的异常 .

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> println("Flow completed with
$cause") }
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
//sampleEnd
```

> 完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-34.kt).

我们可以看到数据流完成的 cause 不是 null, 因为数据流被下游的异常终止了:

```
1
Flow completed with java.lang.IllegalStateException: Collected 2
Exception in thread "main" java.lang.IllegalStateException:
Collected 2
```

命令式 vs 声明式

现在我们知道了如何收集数据流, 以及如何通过命令式和声明式方式, 处理它的完成事件和异常. 下面自然要问, 通常应该使用哪种方式, 为什么? 作为一个库, 我们并不具体的主张使用哪一种方式, 而是相信这两种选择都有价值, 应该按照你自己的偏好和代码风格来进行选择.

启动数据流

很容易使用数据流来表达从某个来源得到的异步的事件. 这种情况下, 我们需要某种类似 `addEventListener` 函数的机制, 用来注册一段代码, 表示对收到的事件的响应, 然后继续后面的工作. onEach (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/on-each.html) 操作符可以实现这个目的. 但是, `onEach` 是一个中间操作符. 我们还需要一个结束操作符来收集数据流. 否则, 仅仅调用 `onEach` 是没有效果的.

如果我们在 `onEach` 之后使用 collect (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect.html) 结束操作符, 那么它之后的代码将会等待, 直到数据流开始收集 :

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
// 模拟一个事件的数据流
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }

fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .collect() // <--- 数据流的收集处理会等待
    println("Done")
}
//sampleEnd
```

> 完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-35.kt).

你可以看到, 这段代码的输出是:

```
Event: 1
Event: 2
Event: 3
Done
```

在这里, 使用 launchIn (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/launch-in.html) 结束操作符就很方便. 将 `collect` 替换为 `launchIn`, 我们可以在一个单独的协程内启动数据流的收集处理, 因此后面的代码可以立即执行:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// 模拟一个事件的数据流
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }

//sampleStart
fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .launchIn(this) // <--- 在一个单独的协程内启动数据流
    println("Done")
}
//sampleEnd
```

> 完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-36.kt).

这段代码的输出是:

```
Done
Event: 1
Event: 2
Event: 3
```

`launchIn` 需要通过参数指定一个 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html), 收集数据流的协程会在这个作用范围(scope)中启动. 上面的示例程序中, 这个作用范围(scope)来自 runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 协程构建器, 因此当数据流运行时, 这个 runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 作用范围(scope)会等待它的子协程完成, 因此能保证这个示例程序的 main 函数不会返回并终止运行.

在真正的应用程序中, 协程作用范围应该来自一个生存期有限的实体. 一旦这个实体的生存期结束, 对应的协程作用范围也会被取消, 并且会取消对应的数据流的收集处理. 通过这种方式, `onEach { ... }.launchIn(scope)` 的组合, 可以象 `addEventListener` 一样工作. 但是, 我们不需要相应的 `removeEventListener` 函数, 因为协程的取消以及结构化的并发功能已经实现了这个功能.

注意, launchIn (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/launch-in.html) 也会返回一个 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html), 这个任务(Job)可以用来 取消(cancel) (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/cancel.html) 相应的数据流收集协程, 但不会取消整个协程作用范围, 还可以用来 join (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/join.html), 等待这个任务(Job)完成.

检查数据流的取消

为了使用方便, flow (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow.html) 构建器会对每个发射的值额外执行一个 ensureActive (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/ensure-active.html) 检查, 用来检查数据流是否被取消. 也就是说, 如果在 `flow { ... }` 之内通过繁忙的循环代码来发射值, 这样的数据流是可以取消的:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```
//sampleStart
fun foo(): Flow<Int> = flow {
    for (i in 1..5) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    foo().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-37.kt).

我们得到的数值只到 3, 然后在试图发射 4 的时候发生 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html) 异常:

```
Emitting 1
1
Emitting 2
2
Emitting 3
3
Emitting 4
Exception in thread "main"
kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@6d7b4f4c
```

但是, 考虑到性能问题, 数据流的其他大多数操作符不会自己做这样的额外的取消检查. 比如, 如果使用 IntRange.asFlow (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/as-flow.html) 扩展, 编写同样的繁忙的循环代码, 并且不在任何地方挂起协程, 那么不会检查数据流是否取消:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun main() = runBlocking<Unit> {
    (1..5).asFlow().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-38.kt).

从 1 到 5 的所有数值都会被收集, 而协程的取消只有在从 `runBlocking` 返回之前才会被检测到:

```
1
2
3
4
5
Exception in thread "main"
kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@3327bd23
```

让繁忙的循环代码变得可以取消

如果你的协程中存在繁忙的循环, 那么就必须明确的检查协程是否被取消. 你可以添加代码 `.onEach { currentCoroutineContext().ensureActive() }`, 但已经有了现成可用的 cancellable

(https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/cancellable.html) 操作符来实现这样的功能:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun main() = runBlocking<Unit> {
    (1..5).asFlow().cancellable().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

完整的示例代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-flow-39.kt).

使用 `cancellable` 操作符之后, 收集的数值只有从 1 到 3:

```
1
2
3
Exception in thread "main"
kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@5ec0a365
```

数据流与响应式(Reactive) Stream

很多开发者已经熟悉了 响应式(Reactive) Stream (https://www.reactive-streams.org/), 或者其他响应式(Reactive)框架, 比如 RxJava 和 Project Reactor, 对这些开发者来说, 数据流的设计看起来应该非常熟悉.

确实如此, 数据流的设计参考了响应式(Reactive) Stream 和它的各种实现. 但是数据流的主要目标是, 要采用尽可能简单的设计, 要与 Kotlin 和协程挂起协调, 并且要遵守结构化并发的各种原则. 没有响应式(Reactive)项目的先驱者和他们的大量工作, 要达到这些目标是不可能的. 完整的故事请阅

读 响应式(Reactive) Stream 与 Kotlin 数据流 (https://medium.com/@elizarov/reactive-streams-and-kotlin-flows-bfd12772cda4).

尽管存在不同, 但在概念上, 数据流 *是* 一个响应式(Reactive) Stream, 数据流可以将转换为响应式发布者(Reactive Publisher) (规格兼容, 而且 TCK 兼容), 也能反过来转换. `kotlinx.coroutines` 包已经提供了这类直接可用的转换器, 可以在相应的响应式(Reactive)模块内找到 (对响应式(Reactive) Stream 是 `kotlinx-coroutines-reactive`, 对 Project Reactor 是 `kotlinx-coroutines-reactor`, 对 RxJava2/RxJava3 是 `kotlinx-coroutines-rx2`/`kotlinx-coroutines-rx3`). 集成模块包含从 `Flow` 的转换, 向 `Flow` 的转换, 与 Reactor 的 `Context` 的集成, 以及, 与协程挂起协调的, 与各种响应式(Reactive)实体共通工作.

通道(Channel)

最终更新: 2024/09/10

延迟产生的数据提供了一种方便的方式可以在协程之间传递单个值. 而通道则提供了另一种方式, 可以在协程之间传递数值的流.

通道的基本概念

Channel (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-channel/index.html) 在概念上非常类似于 `BlockingQueue`. 关键的不同是, 它没有阻塞的 `put` 操作, 而是提供挂起的 send (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-send-channel/send.html) 操作, 没有阻塞的 `take` 操作, 而是提供挂起的 receive (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-receive-channel/receive.html) 操作.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<Int>()
    launch {
        // 这里可能是非常消耗 CPU 的计算工作, 或者是一段异步逻辑, 但在这个例子中我们只是简单地发送 5 个平方数
        for (x in 1..5) channel.send(x * x)
    }
    // 我们在这里输出收到的整数:
    repeat(5) { println(channel.receive()) }
    println("Done!")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-

core/jvm/test/guide/example-channel-01.kt).

这段示例程序的输出是:

```
1
4
9
16
25
Done!
```

通道的关闭与迭代

与序列不同, 通道可以关闭, 表示不会再有更多数据从通道传来了. 在通道的接收端可以使用 `for` 循环很方便地从通道中接收数据.

概念上来说, close (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-send-channel/close.html) 操作类似于向通道发送一个特殊的关闭标记. 收到这个关闭标记之后, 对通道的迭代操作将会立即停止, 因此可以保证在关闭操作以前发送的所有数据都会被正确接收:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<Int>()
    launch {
        for (x in 1..5) channel.send(x * x)
        channel.close() // 我们已经发送完了所有的数据
    }
    // 我们在这里使用 `for` 循环来输出接收到的数据 (通道被关闭后循环就会结束)
    for (y in channel) println(y)
    println("Done!")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-channel-02.kt).

构建通道的生产者(Producer)

在协程中产生一个数值序列, 这是很常见的模式. 这是并发代码中经常出现的 *生产者(producer)/消费者(consumer)* 模式的一部分. 你可以将生产者抽象为一个函数, 并将通道作为函数的参数, 然后向通道发送你生产出来的值, 但这就违反了通常的函数设计原则, 也就是函数的结果应该以返回值的形式对外提供.

有一个便利的协程构建器, 名为 produce (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/produce.html), 它可以很简单地编写出生产者端的正确代码, 还有一个扩展函数 consumeEach (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/consume-each.html), 可以在消费者端代码中替代 `for` 循环:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun CoroutineScope.produceSquares(): ReceiveChannel<Int> = produce {
    for (x in 1..5) send(x * x)
}

fun main() = runBlocking {
//sampleStart
    val squares = produceSquares()
    squares.consumeEach { println(it) }
    println("Done!")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-channel-03.kt).

管道(Pipeline)

管道也是一种设计模式, 比如某个协程可能会产生出无限多个值:

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // 从 1 开始递增的无限整数流
}
```

其他的协程(或者多个协程)可以消费这个整数流, 进行一些处理, 然后产生出其他结果值. 下面的例子中, 我们只对收到的数字做平方运算:

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>):
ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

主代码会启动这些协程, 并将整个管道连接在一起:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val numbers = produceNumbers() // 从 1 开始产生无限的整数
    val squares = square(numbers) // 对整数进行平方
    repeat(5) {
        println(squares.receive()) // 输出前 5 个数字
    }
    println("Done!") // 运行结束
    coroutineContext.cancelChildren() // 取消所有的子协程
//sampleEnd
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // 从 1 开始递增的无限整数流
}
```

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>):
ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-channel-04.kt).

所有创建协程的函数都被定义为 CoroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 上的扩展函数, 因此我们可以依靠 结构化的并发 ("使用 async 的结构化并发" in "挂起函数(Suspending Function)的组合") 来保证应用程序中没有留下长期持续的全局协程.

使用管道寻找质数

下面我们来编写一个示例程序, 使用协程的管道来生成质数, 来演示一下管道的极端用法. 首先我们产生无限的整数序列.

```
fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // 从 start 开始递增的无限整数流
}
```

管道的下一部分会对输入的整数流进行过滤, 删除可以被某个质数整除的数字:

```
fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int)
= produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

下面我们来构建整个管道, 首先从 2 开始产生无限的整数流, 然后从当前通道中取得质数, 并对找到的每个质数执行管道的下一步:

```
numbersFrom(2) -> filter(2) -> filter(3) -> filter(5) -> filter(7)
...
```

下面的示例程序会输出前 10 个质数, 整个管道运行在主线程的上下文之内. 由于所有的协程都是在主 runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 协程的作用范围内启动的, 因此我们不必维护一个已启动的所有协程的列表. 我们可以在输出完前 10 个质数之后, 使用 cancelChildren (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/cancel-children.html) 扩展函数来取消所有的子协程.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    var cur = numbersFrom(2)
    repeat(10) {
        val prime = cur.receive()
        println(prime)
        cur = filter(cur, prime)
    }
    coroutineContext.cancelChildren() // 取消所有的子协程, 让 main 函数
结束
//sampleEnd
}

fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // 从 start 开始递增的无限整数流
}

fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int)
= produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-channel-05.kt).

这段示例程序的输出是:

```
2
3
5
7
11
13
17
19
23
29
```

注意, 你可以使用标准库的协程构建器 `iterator` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/iterator.html) 来创建相同的管道. 把 `produce` 函数替换为 `iterator`, 把 `send` 函数替换为 `yield`, 把 `receive` 函数替换为 `next`, 把 `ReceiveChannel` 替换为 `Iterator`, 就可以不用关心删除协程的作用范围了. 而且你也可以不再需要 `runBlocking`. 但是, 上面的示例中演示的, 使用通道的管道的好处在于, 如果你在 Dispatchers.Default (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html) 上下文中运行的话, 它可以使用 CPU 的多个核心.

总之, 这是一个极不实用的寻找质数的方法. 在实际应用中, 管道一般会牵涉到一些其他的挂起函数调用(比如异步调用远程服务), 而且这些管道不能使用 `sequence`/`iterator` 来构建, 因为这些函数不能允许任意的挂起, 而不象 `produce` 函数, 是完全异步的.

扇出(Fan-out)

多个协程可能会从同一个通道接收数据, 并将计算工作分配给这多个协程. 我们首先来创建一个生产者协程, 它定时产生整数(每秒 10 个整数):

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // 从 1 开始
    while (true) {
```

```
        send(x++) // 产生下一个整数
        delay(100) // 等待 0.1 秒
    }
}
```

然后我们创建多个数据处理协程. 这个示例程序中, 这些协程只是简单地输出自己的 id 以及接收到的整数:

```
fun CoroutineScope.launchProcessor(id: Int, channel:
ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

现在我们启动 5 个数据处理协程, 让它们运行大约 1 秒. 看看结果如何:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
//sampleStart
    val producer = produceNumbers()
    repeat(5) { launchProcessor(it, producer) }
    delay(950)
    producer.cancel() // 取消生产者协程, 因此也杀死了所有其他数据处理协程
//sampleEnd
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // 从 1 开始
    while (true) {
        send(x++) // 产生下一个整数
        delay(100) // 等待 0.1 秒
    }
}

fun CoroutineScope.launchProcessor(id: Int, channel:
```

```
ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-channel-06.kt).

这个示例程序的输出可能类似如下结果, 但处理协程的 id 和实际收到的具体的整数值可能会略微不同:

```
Processor #2 received 1
Processor #4 received 2
Processor #0 received 3
Processor #1 received 4
Processor #3 received 5
Processor #2 received 6
Processor #4 received 7
Processor #0 received 8
Processor #1 received 9
Processor #3 received 10
```

注意, 取消生产者协程会关闭它的通道, 因此最终会结束各个数据处理协程中对这个通道的迭代循环.

而且请注意, 在 `launchProcessor` 中, 我们是如何使用 `for` 循环明确地在通道上进行迭代, 来实现扇出(fan-out). 与 `consumeEach` 不同, 这个 `for` 循环模式完全可以安全地用在多个协程中. 如果某个数据处理协程失败, 其他数据处理协程还会继续处理通道中的数据, 而使用 `consumeEach` 编写的数据处理协程, 无论正常结束还是异常结束, 总是会消费(取消) 它的通道.

扇入(Fan-in)

多个协程也可以向同一个通道发送数据. 比如, 我们有一个字符串的通道, 还有一个挂起函数, 不断向通道发送特定的字符串, 然后暂停一段时间:

```
suspend fun sendString(channel: SendChannel<String>, s: String,
time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

现在, 我们启动多个发送字符串的协程, 来看看结果如何 (在这个示例程序中我们在主线程的上下文中启动这些协程, 作为主协程的子协程):

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<String>()
    launch { sendString(channel, "foo", 200L) }
    launch { sendString(channel, "BAR!", 500L) }
    repeat(6) { // 接收前 6 个字符串
        println(channel.receive())
    }
    coroutineContext.cancelChildren() // 取消所有的子协程, 让 main 函数
结束
//sampleEnd
}

suspend fun sendString(channel: SendChannel<String>, s: String,
time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

完整的代码请参见 这里
(https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-

core/jvm/test/guide/example-channel-07.kt).

输出结果是:

```
foo
foo
BAR!
foo
foo
BAR!
```

带缓冲区的通道

到目前为止我们演示的通道都没有缓冲区. 无缓冲区的通道只会在发送者与接收者相遇时(也叫做会合(rendezvous))传输数据. 如果先调用了发送操作, 那么它会挂起, 直到调用接收操作, 如果先调用接收操作, 那么它会被挂起, 直到调用发送操作.

Channel() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-channel.html) 工厂函数和 produce (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/produce.html) 构建器都可以接受一个可选的 `capacity` 参数, 用来指定 *缓冲区大小*. 缓冲区可以允许发送者在挂起之前发送多个数据, 类似于指定了容量的 `BlockingQueue`, 它会在缓冲区已满的时候发生阻塞.

我们来看看以下示例程序的运行结果:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
//sampleStart
    val channel = Channel<Int>(4) // 创建带缓冲区的通道
    val sender = launch { // 启动发送者协程
        repeat(10) {
            println("Sending $it") // 发送数据之前, 先输出它
            channel.send(it) // 当缓冲区满时, 会挂起
        }
    }
    // 不接收任何数据, 只是等待
```

```
    delay(1000)
    sender.cancel() // 取消发送者协程
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-channel-08.kt).

使用缓冲区大小为 *4* 的通道时, 这个示例程序会输出 "sending" *5* 次:

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

前 4 个数据会被添加到缓冲区中, 然后在试图发送第 5 个数据时, 发送者协程会挂起.

通道是平等的

如果从多个协程中调用通道的发送和接收操作, 从调用发生的顺序来看, 这些操作是 *平等的*. 通道对这些方法以先进先出(first-in first-out)的顺序进行服务, 也就是说, 第一个调用 `receive` 的协程会得到通道中的数据. 在下面的示例程序中, 有两个 "ping" 和 "pong" 协程, 从公用的一个 "table" 通道接收 "ball" 对象.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

//sampleStart
data class Ball(var hits: Int)

fun main() = runBlocking {
    val table = Channel<Ball>() // 一个公用的通道
    launch { player("ping", table) }
    launch { player("pong", table) }
    table.send(Ball(0)) // 把 ball 丢进通道
    delay(1000) // 延迟 1 秒
```

```
    coroutineContext.cancelChildren() // 游戏结束, 取消所有的协程
}

suspend fun player(name: String, table: Channel<Ball>) {
    for (ball in table) { // 使用 for 循环不断地接收 ball
        ball.hits++
        println("$name $ball")
        delay(300) // 延迟一段时间
        table.send(ball) // 把 ball 送回通道内
    }
}
//sampleEnd
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-channel-09.kt).

"ping" 协程首先启动, 因此它会先接收到 ball. 虽然 "ping" 协程将 ball 送回到 table 之后, 立即再次开始接收 ball, 但 ball 会被 "pong" 协程接收到, 因为它一直在等待:

```
ping Ball(hits=1)
pong Ball(hits=2)
ping Ball(hits=3)
pong Ball(hits=4)
```

注意, 由于使用的执行器(executor)的性质, 有时通道的运行结果可能看起来不是那么平等. 详情请参见 这个 issue (https://github.com/Kotlin/kotlinx.coroutines/issues/111).

定时器(Ticker)通道

定时器(Ticker)通道是一种特别的会合通道(rendezvous channel), 每次通道中的数据耗尽之后, 它会延迟一个固定的时间, 并产生一个 `Unit`. 虽然它单独看起来好像毫无用处, 但它是一种很有用的零件, 可以创建复杂的基于时间的 produce (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/produce.html) 管道, 以及操作器, 执行窗口操作和其他依赖于时间的处理. 定时器通道可以用在 select (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.selects/select.html) 中, 执行 "on tick" 动作.

可以使用 ticker (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/ticker.html) 工厂函数来创建这种通道. 使用通道的 ReceiveChannel.cancel (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-receive-channel/cancel.html) 方法来指出不再需要它继续产生数据了.

下面我们看看它的实际应用:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

//sampleStart
fun main() = runBlocking<Unit> {
    val tickerChannel = ticker(delayMillis = 100, initialDelayMillis
= 0) // 创建定时器通道
    var nextElement = withTimeoutOrNull(1) { tickerChannel.receive()
}
    println("Initial element is available immediately:
$nextElement") // 没有初始延迟

    nextElement = withTimeoutOrNull(50) { tickerChannel.receive() }
// 之后产生的所有数据的延迟时间都是 100ms
    println("Next element is not ready in 50 ms: $nextElement")

    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 100 ms: $nextElement")

    // 模拟消费者端的长时间延迟
    println("Consumer pauses for 150ms")
    delay(150)
    // 下一个元素已经产生了
    nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Next element is available immediately after large
consumer delay: $nextElement")
    // 注意, `receive` 调用之间的暂停也会被计算在内,因此下一个元素产生得更快
    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 50ms after consumer pause in
150ms: $nextElement")
```

```
    tickerChannel.cancel() // 告诉通道, 不需要再产生更多元素了
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-channel-10.kt).

这个示例程序的输出结果是:

```
Initial element is available immediately: kotlin.Unit
Next element is not ready in 50 ms: null
Next element is ready in 100 ms: kotlin.Unit
Consumer pauses for 150ms
Next element is available immediately after large consumer delay:
kotlin.Unit
Next element is ready in 50ms after consumer pause in 150ms:
kotlin.Unit
```

注意, ticker (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/ticker.html) 会感知到消费端的暂停, 默认的, 如果消费端发生了暂停, 它会调整下一个元素产生的延迟时间, 尽量保证产生元素时维持一个固定的间隔速度.

另外一种做法是, 将 `mode` 参数设置为 TickerMode.FIXED_DELAY (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-ticker-mode/-f-i-x-e-d_-d-e-l-a-y/index.html), 可以指定产生元素时维持一个固定的间隔速度.

协程的异常处理

最终更新: 2024/09/10

本章介绍异常处理, 以及发生异常时的取消. 我们已经知道, 协程被取消时会在挂起点(suspension point)抛出 CancellationException (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-cancellation-exception/index.html), 而协程机制忽略会这个异常. 下面我们来看看, 如果在取消过程中发生了异常, 或者同一个协程的多个子协程抛出了异常, 那么会出现什么情况

异常的传播(propagation)

协程构建器对于异常的处理有两种风格: 自动传播异常(launch (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) 构建器), 或者将异常交给使用者处理(async (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) 和 produce (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/produce.html) 构建器). 如果使用这些构建器创建一个 *根(root)* 协程, 也就是并不属于其他任何协程的 *子* 协程, 前一种构建器将异常当作 **未捕获的(uncaught)** 异常, 类似于 Java 的 `Thread.uncaughtExceptionHandler`, 后一种则要求使用者处理最终的异常, 比如使用 await (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/await.html) 或 receive (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-receive-channel/receive.html) 来处理异常. (关于 produce (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/produce.html) 和 receive (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-receive-channel/receive.html) 请参见 通道(Channel) (通道(Channel))).

我们通过一个简单的示例程序来演示一下, 我们使用 GlobalScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/index.html) 创建根协程:

> GlobalScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/index.html) 是一个非常精密的 API, 可能会造成严重的影响. 需要使用到 `GlobalScope` 的情况非常少, 其中包括为整个应用程序创建

一个根协程. 因此你需要通过 `@OptIn(DelicateCoroutinesApi::class)` 注解来明确的同意使用 `GlobalScope`.

```
import kotlinx.coroutines.*

//sampleStart
@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val job = GlobalScope.launch { // 通过 launch 创建根协程
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException() // 这个异常会被
Thread.defaultUncaughtExceptionHandler 输出到控制台
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async { // 通过 async 创建根协程
        println("Throwing exception from async")
        throw ArithmeticException() // 这个异常不会被输出, 由使用者调用
await 来得到并处理这个异常
    }
    try {
        deferred.await()
        println("Unreached")
    } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
    }
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-exceptions-01.kt).

(使用 调试模式 ("协程与线程的调试" in "协程上下文与派发器(Dispatcher)")时), 这段代码的输出结果是:

```
Throwing exception from launch
Exception in thread "DefaultDispatcher-worker-2 @coroutine#2" java.lang.IndexOutOfBoundsException
Joined failed job
Throwing exception from async
Caught ArithmeticException
```

CoroutineExceptionHandler

对于 **未捕获的(uncaught)** 异常, 默认的处理方式是输出到控制台, 但也可以自定义如何处理. *根* 协程的上下文元素 CoroutineExceptionHandler (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-exception-handler/index.html) 可以用作这个根协程以及所有子协程的通用的 `catch` 块, 我们可以在这里实现自定义的异常处理逻辑. 它的使用方法类似于 `Thread.uncaughtExceptionHandler` (https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#setUncaughtExceptionHandler(java.lang.Thread.UncaughtExceptionHandler)). 在 `CoroutineExceptionHandler` 内, 你无法从异常中恢复. 当异常处理器被调用时, 协程已经结束运行, 并返回了相应的异常. 通常, 异常处理器会用来将异常输出到日志, 显示某些错误信息, 结束程序运行, 或重启应用程序.

只有 **未捕获的** 异常 — 没有被任何其他方式处理的异常, 才会调用 `CoroutineExceptionHandler`. 具体来说, 所有的 *子* 协程 (在另一个 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) 的上下文内创建的协程) 会把它们的异常交给它们的父协程处理, 父协程又会交给自己的父协程, 如此传递, 直到根协程, 因此安装在子协程的上下文中的 `CoroutineExceptionHandler` 不会被使用. 此外, async (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) 构建器总是会捕获所有异常, 然后将异常作为函数结果 Deferred (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/index.html) 对象的内容, 因此它的 `CoroutineExceptionHandler` 同样不会产生任何效果.

> 在监控(supervision)作用范围内运行的协程, 不会将异常传播到它的父协程, 因此属于上述规则的例外情况. 详情请参见本章的 监控(Supervision) 小节.

```
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
```

```
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) { // 根协程, 运行在
GlobalScope 内
        throw AssertionError()
    }
    val deferred = GlobalScope.async(handler) { // 也是根协程, 但通过
async 创建, 而不是 launch
        throw ArithmeticException() // 这个异常不会被输出, 由使用者调用
deferred.await() 来得到并处理这个异常
    }
    joinAll(job, deferred)
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-exceptions-02.kt).

这个示例程序的输出结果是:

```
CoroutineExceptionHandler got java.lang.AssertionError
```

取消与异常

协程的取消与异常有着非常紧密的关系. 协程内部使用 `CancellationException` 来实现取消, 这些异常会被所有的异常处理器忽略, 因此它们只能用来在 `catch` 块中输出额外的调试信息. 如果使用 Job.cancel (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/cancel.html) 来取消一个协程, 那么协程会终止运行, 但不会取消它的父协程.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
```

```
//sampleStart
    val job = launch {
        val child = launch {
            try {
                delay(Long.MAX_VALUE)
            } finally {
                println("Child is cancelled")
            }
        }
        yield()
        println("Cancelling child")
        child.cancel()
        child.join()
        yield()
        println("Parent is not cancelled")
    }
    job.join()
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-exceptions-03.kt).

这个示例程序的输出结果是:

```
Cancelling child
Child is cancelled
Parent is not cancelled
```

如果一个协程遇到了 `CancellationException` 以外的异常, 那么它会使用这个异常来取消自己的父协程. 这种行为不能覆盖, 而且 Kotlin 使用这个机制来实现 结构化并发 ("使用 async 的结构化并发" in "挂起函数(Suspending Function)的组合") 中的稳定的协程层级关系. CoroutineExceptionHandler (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-exception-handler/index.html) 的实现对子协程不会使用.

> 在这些示例程序中, 我们总是在 GlobalScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/index.html) 内创建的协程上安装 CoroutineExceptionHandler (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-exception-handler/index.html). 如果在 main runBlocking (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 的作用范围内启动的协程上安装异常处理器, 是毫无意义的, 因为子协程由于异常而终止之后, 主协程一定会被取消, 而忽略它上面安装的异常处理器.

只有当所有的子协程全部终止之后, 最初的异常才会由父协程处理, 请看下面示例程序的演示.

```
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        launch { // 第 1 个子协程
            try {
                delay(Long.MAX_VALUE)
            } finally {
                withContext(NonCancellable) {
                    println("Children are cancelled, but exception
is not handled until all children terminate")
                    delay(100)
                    println("The first child finished its non
cancellable block")
                }
            }
        }
        launch { // 第 2 个子协程
            delay(10)
            println("Second child throws an exception")
```

```
            throw ArithmeticException()
        }
    }
    job.join()
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-exceptions-04.kt).

这个示例程序的输出结果是:

```
Second child throws an exception
Children are cancelled, but exception is not handled until all
children terminate
The first child finished its non cancellable block
CoroutineExceptionHandler got java.lang.ArithmeticException
```

异常的聚合(aggregation)

如果一个协程的多个子协程都由于发生异常而失败, 通常的规则是 "最先发生的异常优先", 因此第 1 个发生的异常会被处理. 在此之后发生的所有其他异常会被添加到最先发生的异常上, 作为被压制(suppressed)的异常.

```
import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception with
suppressed ${exception.suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
```

```
                delay(Long.MAX_VALUE) // 如果其他兄弟协程由于
IOException 异常而失败, 那么这个协程会被取消
            } finally {
                throw ArithmeticException() // 第二个异常
            }
        }
        launch {
            delay(100)
            throw IOException() // 第一个异常
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-exceptions-05.kt).

这个示例程序的输出结果是:

```
CoroutineExceptionHandler got java.io.IOException with suppressed
[java.lang.ArithmeticException]
```

注意, 异常聚合机制目前只能在 Java version 1.7+ 以上版本才能正常工作. JS 和 原生平台目前暂时不支持异常聚合, 将来会解决这个问题.

协程取消异常是透明的, 默认不会被聚合到其他异常中:

```
import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
```

```
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        val innerJob = launch { // 从这里开始的所有协程都会被取消
            launch {
                launch {
                    throw IOException() // 最初的异常
                }
            }
        }
        try {
            innerJob.join()
        } catch (e: CancellationException) {
            println("Rethrowing CancellationException with original
cause")
            throw e // 再次抛出协程被取消的异常, 但仍然是最初的
IOException 被处理
        }
    }
    job.join()
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-exceptions-06.kt).

这个示例程序的输出结果是:

```
Rethrowing CancellationException with original cause
CoroutineExceptionHandler got java.io.IOException
```

监控

正如我们前面学到的, 取消是一种双向关系, 它会在整个协程层级关系内传播. 下面我们来看看, 如果需要单向的取消, 会发生什么情况.

这种需求的一个很好的例子就是一个 UI 组件, 在它的作用范围内定义了一个任务. 如果 UI 的任何一个子任务失败, 并不一定有必要取消(最终效果就是杀死) 整个 UI 组件, 但是如果 UI 组件本身被销毁(而且它的任务也被取消了), 那么就有必要终止所有的子任务, 因为子任务的结果已经不再需要了.

另一个例子是, 一个服务器进程启动了多个子任务, 需要 *监控* 这些子任务的执行, 追踪它们是否失败, 只对那些失败的子任务进行重启.

监控任务

SupervisorJob (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-supervisor-job.html) 可以用作这类目的. 它与通常的 Job (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job.html) 类似, 唯一的区别在于取消只向下方传播. 我们用下面的示例程序来演示一下:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        // 启动第 1 个子协程 -- 在这个示例程序中, 我们会忽略它的异常 (实际应
用中不要这样做!)
        val firstChild = launch(CoroutineExceptionHandler { _, _ ->
}) {
            println("The first child is failing")
            throw AssertionError("The first child is cancelled")
        }
        // 启动第 2 个子协程
        val secondChild = launch {
            firstChild.join()
            // 第 1 个子协程的取消不会传播到第 2 个子协程
            println("The first child is cancelled:
${firstChild.isCancelled}, but the second one is still active")
            try {
                delay(Long.MAX_VALUE)
            } finally {
                // 但监控任务的取消会传播到第 2 个子协程
                println("The second child is cancelled because the
supervisor was cancelled")
            }
```

```
        }
        // 等待第 1 个子协程失败, 并结束运行
        firstChild.join()
        println("Cancelling the supervisor")
        supervisor.cancel()
        secondChild.join()
    }
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-supervision-01.kt).

这个示例程序的输出结果是:

```
The first child is failing
The first child is cancelled: true, but the second one is still
active
Cancelling the supervisor
The second child is cancelled because the supervisor was cancelled
```

监控作用范围

对于 *带作用范围* 的并发, 可以使用 supervisorScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/supervisor-scope.html) 代替 coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 来实现同一目的. 它也只向一个方向传播取消, 并且只在它自身失败的情况下取消所有的子协程. 它在运行结束之前也会等待所有的子协程结束, 和 coroutineScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 一样.

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
```

```
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("The child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("The child is cancelled")
                }
            }
            // 使用 yield, 给子协程一个机会运行, 并输出信息
            yield()
            println("Throwing an exception from the scope")
            throw AssertionError()
        }
    } catch(e: AssertionError) {
        println("Caught an assertion error")
    }
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-supervision-02.kt).

这个示例程序的输出结果是:

```
The child is sleeping
Throwing an exception from the scope
The child is cancelled
Caught an assertion error
```

被监控的协程中的异常

常规任务与监控任务的另一个重要区别就是对异常的处理方式. 每个子协程都应该通过异常处理机制自行处理它的异常. 区别在于, 子协程的失败不会传播到父协程中. 也就是说, 直接在 supervisorScope (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/supervisor-scope.html) 之内启动的协程, 就象根协程一样, 会 使用安装

在其作用范围上的 CoroutineExceptionHandler (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-exception-handler/index.html), (详情请参见 CoroutineExceptionHandler 小节).

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("The child throws an exception")
            throw AssertionError()
        }
        println("The scope is completing")
    }
    println("The scope is completed")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-supervision-03.kt).

这个示例程序的输出结果是:

```
The scope is completing
The child throws an exception
CoroutineExceptionHandler got java.lang.AssertionError
The scope is completed
```

共享的可变状态与并发

最终更新: 2024/09/10

使用多线程的派发器, 比如 Dispatchers.Default (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html), 协程可以并发执行. 因此协程也面对并发带来的所有问题. 主要问题是访问 **共享的可变状态值** 时的同步问题. 在协程的世界里, 这类问题的有些解决方案与在线程世界中很类似, 但另外一些方案就非常不同.

问题的产生

下面我们启动 100 个协程, 每个协程都将同样的操作执行 1000 次. 我们测量一下它们的结束时间, 并做进一步的比较:

```
suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程执行操作的重复次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用范围
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

我们先来执行一个非常简单的操作, 使用多线程的 Dispatchers.Default (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html), 把一个共享的可变变量加 1.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
```

```
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程执行操作的重复次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用范围
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-sync-01.kt)

最终的输出结果会是什么? 非常不太可能会输出 "Counter = 100000", 因为有 100 个协程, 从多个线程中同时增加 `counter` 的值, 却没有任何并发控制.

volatile 不能解决这个问题

有一种常见的错误观念, 认为把变量变为 `volatile` 就可以解决并发访问问题. 我们来试一下:

```kotlin
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程执行操作的重复次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用范围
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
@Volatile // 在 Kotlin 中, `volatile` 是注解
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-sync-02.kt).

代码运行变慢了, 但我们还是不能总是得到 "Counter = 100000" 的最后结果, 因为 volatile 变量保证线性的(linearizable) (意思就是 "原子性(atomic)") 读和写操作, 但不能保证更大的操作(在我们的例子中, 就是加 1 操作)的原子性.

线程安全的数据结构

一种对于线程和协程都能够适用的解决方案是, 使用线程安全的 (也叫 同步的(synchronized), 线性的(linearizable), 或者 原子化的(atomic)) 数据结构, 这些数据结构会对需要在共享的状态数据上进行的操作提供必要的同步保障. 在我们的简单的计数器示例中, 可以使用 `AtomicInteger` 类, 它有一个原子化的 `incrementAndGet` 操作:

```
import kotlinx.coroutines.*
import java.util.concurrent.atomic.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程执行操作的重复次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用范围
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counter = AtomicInteger()

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter.incrementAndGet()
        }
    }
```

```
    println("Counter = $counter")
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-sync-03.kt).

对于这个具体的问题, 这是最快的解决方案. 这种方案适用于计数器, 集合, 队列, 以及其他标准数据结构, 以及这些数据结构的基本操作. 但是, 这种方案并不能简单地应用于复杂的状态变量, 或者那些没有现成的线程安全实现的复杂操作.

细粒度的线程限定

线程限定(Thread confinement) 是共享的可变状态值问题的一种解决方案, 它把所有对某个共享值的访问操作都限定在唯一的一个线程内. 最典型的应用场景是 UI 应用程序, 所有的 UI 状态都被限定在唯一一个 事件派发(event-dispatch) 线程 或者叫 application 线程内. 通过使用单线程的上下文, 可以很容易地对协程使用这种方案.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程执行操作的重复次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用范围
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

```
//sampleStart
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // 把所有的加 1 操作限定在单一线程的上下文中
            withContext(counterContext) {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-sync-04.kt).

这段代码的运行速度会非常地慢, 因为它进行了 *细粒度(fine-grained)* 的线程限定. 每一次加 1 操作都必须使用 withContext(counterContext) (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-context.html), 从多线程的 Dispatchers.Default (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-dispatchers/-default.html) 上下文切换到单一线程上下文.

粗粒度的线程限定

在实际应用中, 通常在更大的尺度上进行线程限定, 比如, 将大块的状态更新业务逻辑限定在单个线程中. 下面的示例程序就是这样做的, 它在单一线程的上下文中运行每个协程.

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
```

```
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程执行操作的重复次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用范围
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    // 将所有操作限定在单一线程的上下文中
    withContext(counterContext) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-sync-05.kt).

现在我们的代码运行的很快, 而且能够得到正确的结果.

互斥

对于这个问题的另一个解决方案是互斥(Mutual exclusion), 它使用一个 *临界区(critical section)* 来保护所有针对共享状态值的修改动作, 临界区内的代码永远不会并发执行. 在阻塞式编程的世界, 你通常会使用 `synchronized` 或 `ReentrantLock` 来实现这个目的. 在线程中的方案叫做 Mutex (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.sync/-mutex/index.html). 它的 lock (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.sync/-mutex/lock.html) 和 unlock (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.sync/-mutex/unlock.html) 函数可以用来界定临界区. 主要的区别在于 `Mutex.lock()` 是一个挂起函数. 它不会阻塞线程.

还有一个扩展函数 withLock (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.sync/with-lock.html), 它用非常便利的方式实现 `mutex.lock(); try { ... } finally { mutex.unlock() }` 模式:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.sync.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程执行操作的重复次数
    val time = measureTimeMillis {
        coroutineScope { // scope for coroutines
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val mutex = Mutex()
var counter = 0

fun main() = runBlocking {
```

```
    withContext(Dispatchers.Default) {
        massiveRun {
            // 使用锁来保护每次加 1 操作
            mutex.withLock {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-sync-06.kt).

上面的示例程序中的锁是细粒度的, 因此会产生一些代价. 但是, 对于某些情况下, 你确实需要不时修改某些共享的状态值, 但是这个状态值又没有限定在某个线程之内, 那么使用锁是一种好的选择.

选择表达式(Select expression) (实验性功能)

最终更新: 2024/09/10

使用选择表达式, 我们可以同时等待多个挂起函数, 并且 *选择* 其中第一个执行完毕的结果.

> 选择表达式是 `kotlinx.coroutines` 中的一个实验性功能. 在以后的 `kotlinx.coroutines` 新版本库中, 与选择表达式相关的 API 将会发生变化, 可能带来一些不兼容的变更.

从通道中选择

假设我们有两个 string 值的生产者: `fizz` 和 `buzz`. 其中 `fizz` 每 500ms 产生一个 "Fizz" 字符串:

```
fun CoroutineScope.fizz() = produce<String> {
    while (true) { // 每 500ms 发送一个 "Fizz"
        delay(500)
        send("Fizz")
    }
}
```

`buzz` 每 1000ms 产生一个 "Buzz!" 字符串:

```
fun CoroutineScope.buzz() = produce<String> {
    while (true) { // 每 1000ms 发生一个 "Buzz!"
        delay(1000)
        send("Buzz!")
    }
}
```

使用 receive (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-receive-channel/receive.html) 挂起函数, 我们可以接收这两个通道中的 *任何一个*. 但使用 select (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.selects/select.html) 表达式的 onReceive (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-

core/kotlinx.coroutines.channels/-receive-channel/on-receive.html) 子句, 我们可以 *同时接收两个通道的数据*:

```
suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz:
ReceiveChannel<String>) {
    select<Unit> { // <Unit> 表示这个 select 表达式不产生任何结果值
        fizz.onReceive { value ->  // 这是第 1 个 select 子句
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // 这是第 2 个 select 子句
            println("buzz -> '$value'")
        }
    }
}
```

下面我们把这段代码运行 7 次:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.fizz() = produce<String> {
    while (true) { // 每 500ms 发送一个 "Fizz"
        delay(500)
        send("Fizz")
    }
}

fun CoroutineScope.buzz() = produce<String> {
    while (true) { // 每 1000ms 发生一个 "Buzz!"
        delay(1000)
        send("Buzz!")
    }
}

suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz:
ReceiveChannel<String>) {
    select<Unit> { // <Unit> 表示这个 select 表达式不产生任何结果值
```

```
        fizz.onReceive { value ->  // 这是第 1 个 select 子句
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // 这是第 2 个 select 子句
            println("buzz -> '$value'")
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val fizz = fizz()
    val buzz = buzz()
    repeat(7) {
        selectFizzBuzz(fizz, buzz)
    }
    coroutineContext.cancelChildren() // 取消 fizz 和 buzz 协程
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-select-01.kt).

这个示例程序的输出结果是:

```
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
buzz -> 'Fizz!'
```

在通道关闭时选择

如果通道已关闭, 那么 `select` 表达式的 onReceive (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-receive-channel/on-receive.html) 子句会失败, 并导致 `select` 表达式抛出一个异常. 我们可以使用 [onReceiveOrNull][onReceiveOrNull] 子句, 来对通道关闭的情况执行某个操作. 下面的示例程序还演示了 `select` 是一个表达式, 它会返回它的子句的结果:

```
suspend fun selectAorB(a: ReceiveChannel<String>, b:
ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "a -> '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "b -> '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }
```

假设 `a` 通道产生 4 次 "Hello" 字符串, `b` 通道产生 4 次 "World" 字符串, 我们来使用一下这个函数:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

suspend fun selectAorB(a: ReceiveChannel<String>, b:
ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it ->
```

```
            val value = it.getOrNull()
            if (value != null) {
                "a -> '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "b -> '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }

fun main() = runBlocking<Unit> {
//sampleStart
    val a = produce<String> {
        repeat(4) { send("Hello $it") }
    }
    val b = produce<String> {
        repeat(4) { send("World $it") }
    }
    repeat(8) { // 输出前 8 个结果
        println(selectAorB(a, b))
    }
    coroutineContext.cancelChildren()
//sampleEnd
}
```

> 完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-select-02.kt).

这个示例程序的输出结果比较有趣，所以我们来分析一下其中的细节：

```
a -> 'Hello 0'
a -> 'Hello 1'
b -> 'World 0'
a -> 'Hello 2'
a -> 'Hello 3'
b -> 'World 1'
Channel 'a' is closed
Channel 'a' is closed
```

从这个结果我们可以观察到几件事.

首先, `select` 会 *偏向* 第 1 个子句. 如果同时存在多个通道可供选择, 那么会优先选择其中的第 1 个. 在上面的示例中, 两个通道都在不断产生字符串, 因此第 1 个通道, 也就是 `a`, 会被优先使用. 然而, 由于我们使用了无缓冲区的通道, 因此 `a` 在调用 send (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-send-channel/send.html) 时有时会挂起, 因此 `b` 通道也有机会可以发送数据.

第 2 个现象是, 当通道被关闭时, 会立即选择 onReceiveCatching (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-receive-channel/on-receive-catching.html) 子句.

发送时选择

选择表达式也可以使用 onSend (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-send-channel/on-send.html) 子句, 它可以与选择表达式的偏向性结合起来, 起到很好的作用.

下面我们来编写一个示例程序, 有一个整数值的生产者, 当主通道的消费者的消费速度跟不上生产者的发送速度时, 会把它的值改为发送到 `side` 通道:

```
fun CoroutineScope.produceNumbers(side: SendChannel<Int>) =
produce<Int> {
    for (num in 1..10) { // 产生 10 个数值, 从 1 到 10
        delay(100) // 每隔 100 ms
        select<Unit> {
            onSend(num) {} // 发送到主通道
            side.onSend(num) {} // 或者发送到 side 通道
        }
```

```
    }
}
```

我们让消费者的运行速度变得比较慢一些, 每隔 250 ms 处理一个数值:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.produceNumbers(side: SendChannel<Int>) =
produce<Int> {
    for (num in 1..10) { // 产生 10 个数值, 从 1 到 10
        delay(100) // 每隔 100 ms
        select<Unit> {
            onSend(num) {} // 发送到主通道
            side.onSend(num) {} // 或者发送到 side 通道
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val side = Channel<Int>() // 创建 side 通道
    launch { // 这是 side 通道上的一个非常快速的消费者
        side.consumeEach { println("Side channel has $it") }
    }
    produceNumbers(side).consumeEach {
        println("Consuming $it")
        delay(250) // 我们多花点时间慢慢分析这个数值, 不要着急
    }
    println("Done consuming")
    coroutineContext.cancelChildren()
//sampleEnd
}
```

完整的代码请参见 这里
(https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-

core/jvm/test/guide/example-select-03.kt).

下面我们来看看运行结果会怎么样:

```
Consuming 1
Side channel has 2
Side channel has 3
Consuming 4
Side channel has 5
Side channel has 6
Consuming 7
Side channel has 8
Side channel has 9
Consuming 10
Done consuming
```

选择延迟的值

可以使用 onAwait (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/on-await.html) 子句来选择延迟的值(Deferred value). 我们先从一个异步函数开始, 它会延迟一段随机长度的时间, 然后返回一个延迟的字符串值:

```
fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}
```

然后我们用随机长度的延迟时间, 来启动这个函数 12 次.

```
fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}
```

下面, 我们让 main 函数等待这些异步函数的第 1 个运行完毕, 然后统计仍处于激活状态的延迟值的数量. 注意, 这里我们利用了 `select` 表达式是 Kotlin DSL 的这种特性, 因此我们可以使用任意的代

码来作为它的子句. 在这个示例程序中, 我们在一个延迟值的 List 上循环, 为每个延迟值产生一个 `onAwait` 子句.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.selects.*
import java.util.*

fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}

fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val list = asyncStringsList()
    val result = select<String> {
        list.withIndex().forEach { (index, deferred) ->
            deferred.onAwait { answer ->
                "Deferred $index produced answer '$answer'"
            }
        }
    }
    println(result)
    val countActive = list.count { it.isActive }
    println("$countActive coroutines are still active")
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-select-04.kt).

运行结果是:

```
Deferred 4 produced answer 'Waited for 128 ms'
11 coroutines are still active
```

在延迟值的通道上切换

下面我们来编写一个通道生产者函数, 它从一个通道得到延迟的字符串值, 等待每一个接收到的值, 但如果下一个延迟值到达, 或者通道被关闭, 就不再等待了. 这个示例程序在同一个 `select` 中结合使用了 onReceiveCatching (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-receive-channel/on-receive-catching.html) 子句 和 onAwait (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/on-await.html) 子句:

```
fun CoroutineScope.switchMapDeferreds(input:
ReceiveChannel<Deferred<String>>) = produce<String> {
    var current = input.receive() // 从第 1 个接收到的延迟值开始
    while (isActive) { // 无限循环, 直到通道被取消/关闭
        val next = select<Deferred<String>?> { // 这个 select 表达式返
回下一个延迟值, 或者 null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // 如果当前正在等待的延迟值已经产生, 将它发送出
去
                input.receiveCatching().getOrNull() // 再继续使用从输入
通道得到的下一个延迟值
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // 循环结束
        } else {
            current = next
        }
    }
}
```

要测试这段程序，我们使用一个简单的异步函数，它会等待一段指定的时间，然后返回一个指定的字符串：

```
fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}
```

main 函数启动一个协程来输出 `switchMapDeferreds` 的结果，并向它发送一些测试数据：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.switchMapDeferreds(input:
ReceiveChannel<Deferred<String>>) = produce<String> {
    var current = input.receive() // 从第 1 个接收到的延迟值开始
    while (isActive) { // 无限循环，直到通道被取消/关闭
        val next = select<Deferred<String>?> { // 这个 select 表达式返
回下一个延迟值，或者 null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // 如果当前正在等待的延迟值已经产生，将它发送出
去
                input.receiveCatching().getOrNull() // 再继续使用从输入
通道得到的下一个延迟值
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // 循环结束
        } else {
            current = next
        }
    }
}
```

```
fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}

fun main() = runBlocking<Unit> {
//sampleStart
    val chan = Channel<Deferred<String>>() // 用来测试的通道
    launch { // 启动输出结果的协程
        for (s in switchMapDeferreds(chan))
            println(s) // 输出每个收到的字符串
    }
    chan.send(asyncString("BEGIN", 100))
    delay(200) // 延迟足够长的时间, 让 "BEGIN" 输出到通道
    chan.send(asyncString("Slow", 500))
    delay(100) // 延迟的时间不够长, "Slow" 没有输出到通道
    chan.send(asyncString("Replace", 100))
    delay(500) // 在发送最后一条测试数据之前等待一段时间
    chan.send(asyncString("END", 500))
    delay(1000) // 给它一点时间运行
    chan.close() // 关闭通道 ...
    delay(500) // 等待一段时间, 让它结束运行
//sampleEnd
}
```

完整的代码请参见 这里 (https://github.com/kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/test/guide/example-select-05.kt).

这个示例程序的运行结果是:

```
BEGIN
Replace
END
Channel was closed
```

教程 - 使用 IntelliJ IDEA 调试协程

最终更新: 2024/09/10

本教程演示如何创建 Kotlin 协程, 并使用 IntelliJ IDEA 调试这些协程.

本教程假定你已经了解了 协程 (协程指南) 的基本概念.

创建协程

1. 在 IntelliJ IDEA 中打开一个 Kotlin 项目. 如果你没有项目, 请 创建一个项目 ("创建应用程序" in "Kotlin/JVM 入门").
2. 要在 Gradle 项目中使用 `kotlinx.coroutines` 库, 请向 `build.gradle(.kts)` 添加以下依赖项:

   Kotlin

   ```
   dependencies {
       implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")
   }
   ```

   Groovy

   ```
   dependencies {
       implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3'
   }
   ```

   对于其他构建系统, 请参见 `kotlinx.coroutines` README (https://github.com/Kotlin/kotlinx.coroutines#using-in-your-projects) 中的说明.
3. 打开 `src/main/kotlin` 中的 `Main.kt` 文件.

   `src` 目录包含 Kotlin 源代码文件和资源文件. `Main.kt` 文件包含示例代码, 它会输出 `Hello World!`.

4. 修改中 main() 函数的代码:

- 使用 runBlocking() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 代码块来封装一一个协程.
- 使用 async() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) 函数创建协程, 分别计算 a 和 b 的值.
- 使用 await() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/await.html) 函数等待计算结果.
- 使用 println() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/println.html) 函数输出计算状态, 以及乘法运算的结果.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val a = async {
        println("I'm computing part of the answer")
        6
    }
    val b = async {
        println("I'm computing another part of the answer")
        7
    }
    println("The answer is ${a.await() * b.await()}")
}
```

5. 点击 **Build Project**, 构建代码.

构建一个应用程序

调试协程

1. 在 `println()` 函数调用的行设置断点:

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val a = async {
        println("I'm computing part of the answer")
        6
    }
    val b = async {
        println("I'm computing another part of the answer")
        7
    }
    println("The answer is ${a.await() * b.await()}")
}
```

构建一个控制台应用程序

2. 点击画面顶部运行配置旁边的 **Debug**, 在调试模式下运行代码.

test 〉 src 〉 main 〉 kotlin 〉 Main.kt

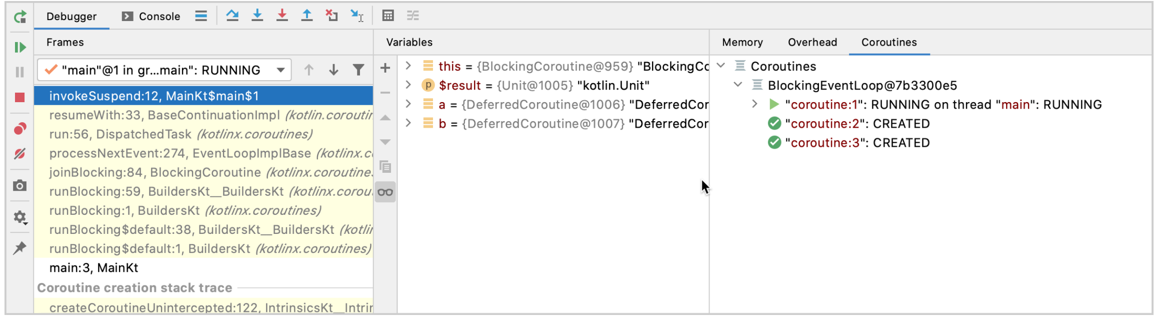
Debug 'MainKt' ^D

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val a = async {
```

构建一个控制台应用程序

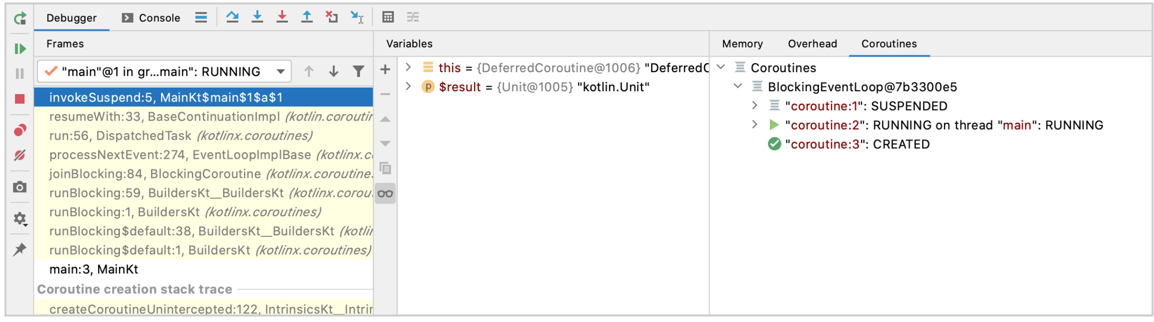
然后会出现 **Debug** 工具窗口:

- **Frames** 页包含调用栈.
- **Variables** 页包含当前上下文中的变量.
- **Coroutines** 页包含正在运行的或挂起的协程信息. 它显示存在 3 个协程. 第一个协程状态为 **RUNNING**, 其它两个状态为 **CREATED**.



调试协程

3. 点击 **Debug** 工具窗口中的 **Resume Program**, 恢复调试器 session:

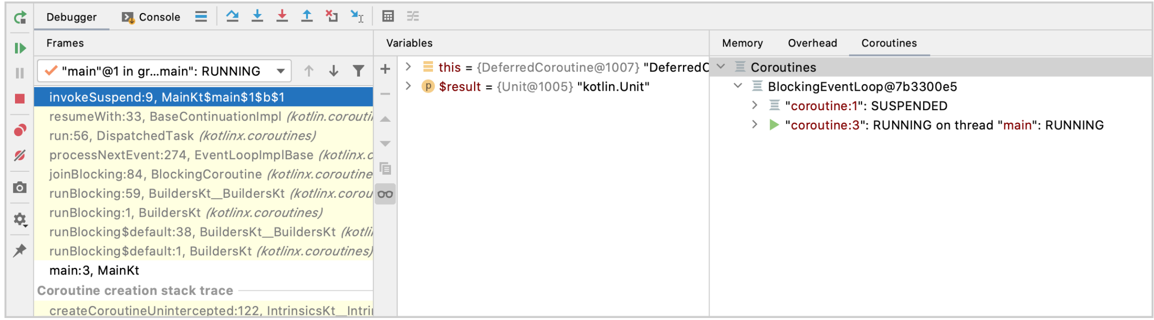


调试协程

现在 **Coroutines** 页显示如下:

- 第 1 个协程状态为 **SUSPENDED** – 它在等待值, 以便执行乘法运算.
- 第 2 个协程正在计算 `a` 的值 – 状态为 **RUNNING**.
- 第 3 个协程状态为 **CREATED**, 还没有计算 `b` 的值.

4. 点击 **Debug** 工具窗口中的 **Resume Program**, 恢复调试器 session:



构建一个控制台应用程序

现在 **Coroutines** 页显示如下:

* 第 1 个协程状态为 **SUSPENDED** – 它在等待值, 以便执行乘法运算.
* 第 2 个协程已经计算完毕, 并且消失了.
* 第 3 个协程正在计算 `b` 的值 – 它的状态为 **RUNNING**.

使用 IntelliJ IDEA 调试器, 你可以深入挖掘各个协程的信息, 调试你的代码.

被优化的变量

如果你使用 `suspend` 函数, 那么在调试器中, 你可能会在变量名称旁边看到 "was optimized out" 文字:

```
5  suspend fun main() {
6      val a = 1L
7      delay(a)
8      println("Slept for 1 ms")
9  }
```

```
> $result = {Unit@1140} kotlin.Unit
  'a' was optimised out
```

变量 "a" 被优化了

这段文字的意思是说, 变量的生存时间变短了, 而且变量已经不再存在了. 如果变量被优化, 调试代码会变得困难, 因为你看不到变量值. 你可以使用 `-Xdebug` 编译器选项禁止这种优化.

> ⚠ **绝对不要在产品(Production)模式中使用这个选项**: `-Xdebug` 可能 导致内存泄露 (https://youtrack.jetbrains.com/issue/KT-48678/Coroutine-debugger-disable-was-optimised-out-compiler-feature#focus=Comments-27-6015585.0-0).

教程 - 使用 IntelliJ IDEA 调试 Kotlin 数据流(Flow)

最终更新: 2024/09/10

本教程演示如何创建 Kotlin 数据流(Flow), 并使用 IntelliJ IDEA 调试它.

本教程假定你已经具备了 协程 (协程指南) 和 Kotlin 数据流(Flow) ("数据流(Flow)" in "异步的数据流(Asynchronous Flow)") 的相关知识.

创建 Kotlin 数据流

创建一个 Kotlin 数据流(Flow) (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow.html), 其中包括一个慢速的发射者(emitter)和一个慢速的收取者(collector):

1. 在 IntelliJ IDEA 中打开一个 Kotlin 项目. 如果你没有项目, 请 创建项目 ("创建应用程序" in "Kotlin/JVM 入门").
2. 要在 Gradle 项目中使用 `kotlinx.coroutines` 库, 请向 `build.gradle(.kts)` 添加以下依赖项:

   Kotlin

   ```
   dependencies {
       implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")
   }
   ```

   Groovy

   ```
   dependencies {
       implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3'
   }
   ```

对于其他构建系统, 请参见 kotlinx.coroutines README (https://github.com/Kotlin/kotlinx.coroutines#using-in-your-projects) 中的说明.

3. 打开 src/main/kotlin 中的 Main.kt 文件 .

   src 目录包含 Kotlin 源代码文件和资源. Main.kt 文件包含一段输出 Hello World! 的示例代码.

4. 创建 simple() 函数, 它返回一个数据流, 其中包含 3 个数字:

   - 使用 delay() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/delay.html) 函数来模拟一段大量消耗 CPU 的阻塞代码. 这个函数会挂起协程 100 ms, 不会阻塞线程.
   - 在 for 循环内使用 emit() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow-collector/emit.html) 函数产生值.

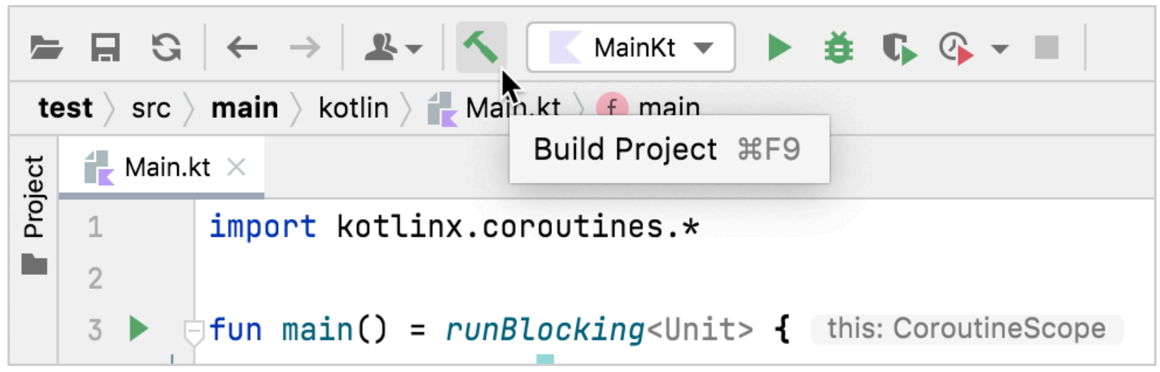
```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}
```

5. 修改 main() 函数中的代码:

   - 使用 runBlocking() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 代码块封装一个协程.
   - 使用 collect() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect.html) 函数收取发射的值.
   - 使用 delay() (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/delay.html) 函数来模拟一段大量消耗 CPU 的阻塞代码. 这个函数会挂起协程 300 ms, 不会阻塞线程.
   - 使用 println() (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/println.html) 函数打印从数据流收取的值.

```
fun main() = runBlocking {
    simple()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

6. 点击 **Build Project** 构建代码.



构建应用程序

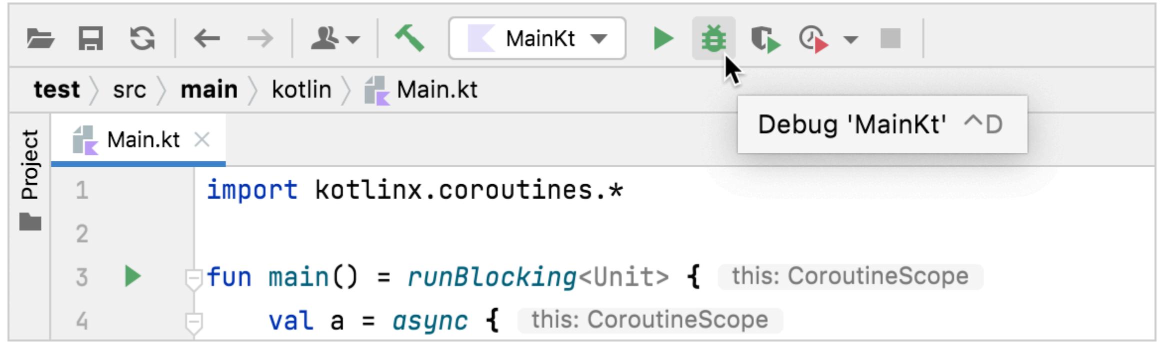
调试协程

1. 在调用 `emit()` 函数的代码行设置一个断点:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking {
    simple()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

构建一个控制台应用程序

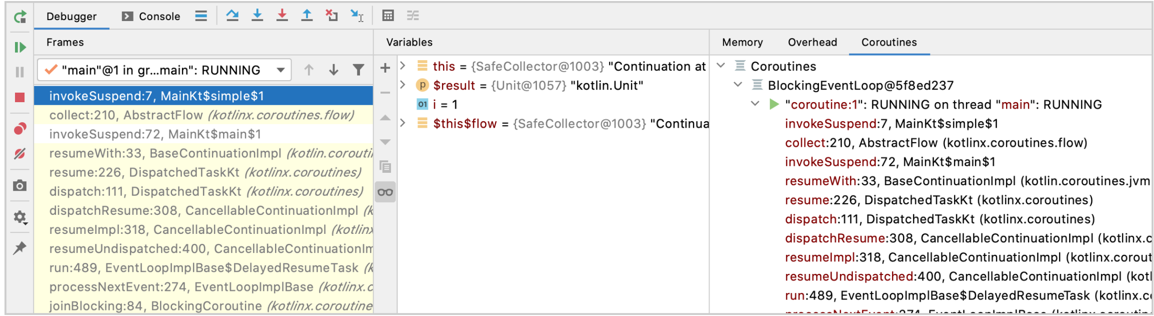
2. 点击屏幕顶部运行配置旁边的 **Debug**, 使用调试模式运行代码.



构建一个控制台应用程序

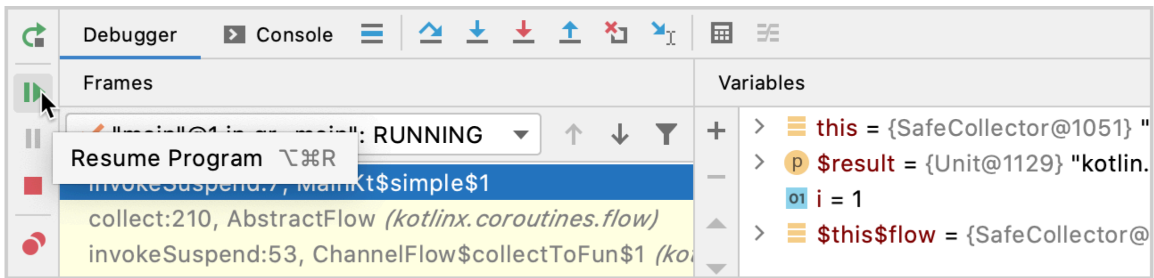
这时会出现 **Debug** 工具窗口:

- **Frames** 页包含调用栈信息.
- **Variables** 页包含当前上下文环境中的变量. 它告诉我们数据流正在发射第 1 个值.
- **Coroutines** 页包含正在运行中的或者被挂起的协程信息.



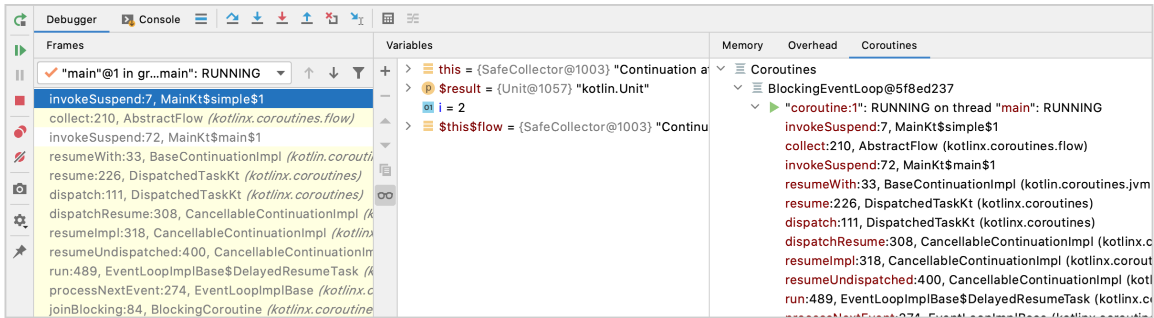
调试协程

3. 点击 **Debug** 工具窗口中的 **Resume Program**, 回复调试器运行. 程序会在同一个断点再次暂停.



调试协程

现在数据流发射第 2 个值.



调试协程

被优化的变量

如果你使用 `suspend` 函数, 那么在调试器中, 你可能会在变量名称旁边看到 "was optimized out" 文字:

```
5  suspend fun main() {
6      val a = 1L
7      delay(a)
8      println("Slept for 1 ms")
9  }
```

```
> $result = {Unit@1140} kotlin.Unit
  'a' was optimised out
```

变量 "a" 被优化了

这段文字的意思是说, 变量的生存时间变短了, 而且变量已经不再存在了. 如果变量被优化, 调试代码会变得困难, 因为你看不到变量值. 你可以使用 `-Xdebug` 编译器选项禁止这种优化.

绝对不要在产品(Production)模式中使用这个选项: `-Xdebug` 可能 导致内存泄露 (https://youtrack.jetbrains.com/issue/KT-48678/Coroutine-debugger-disable-was-optimised-out-compiler-feature#focus=Comments-27-6015585.0-0).

添加一个并发运行的协程

1. 打开 `src/main/kotlin` 中的 `main.kt` 文件.

2. 修改代码, 让发射者和收取者并发运行:

   - 推荐一个 `buffer()` (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/buffer.html) 函数调用, 并发运行发射者和收取者. `buffer()` 存储已发射的值, 并在一个独立的协程中运行数据流收取者.

```
fun main() = runBlocking<Unit> {
    simple()
        .buffer()
```

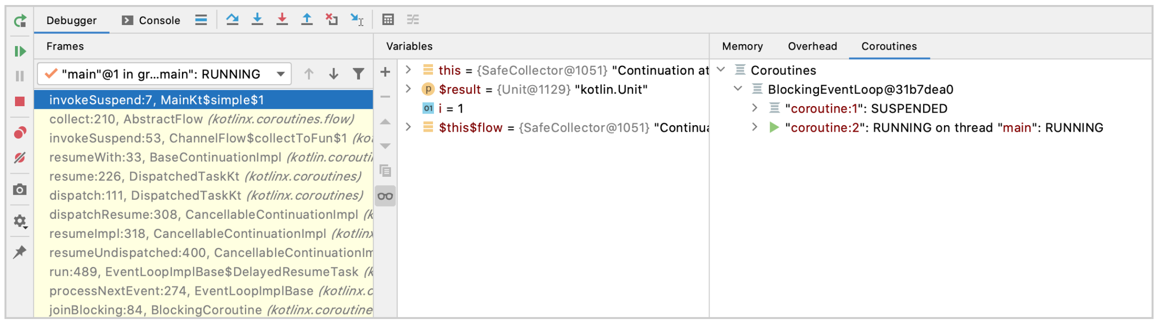
```
            .collect { value ->
                delay(300)
                println(value)
            }
}
```

3. 点击 **Build Project**, 构建代码.

调试有 2 个协程的 Kotlin 数据流

1. 在 `println(value)` 处设置一个新断点.

2. 点击屏幕顶部运行配置旁边的 **Debug**, 使用调试模式运行代码.

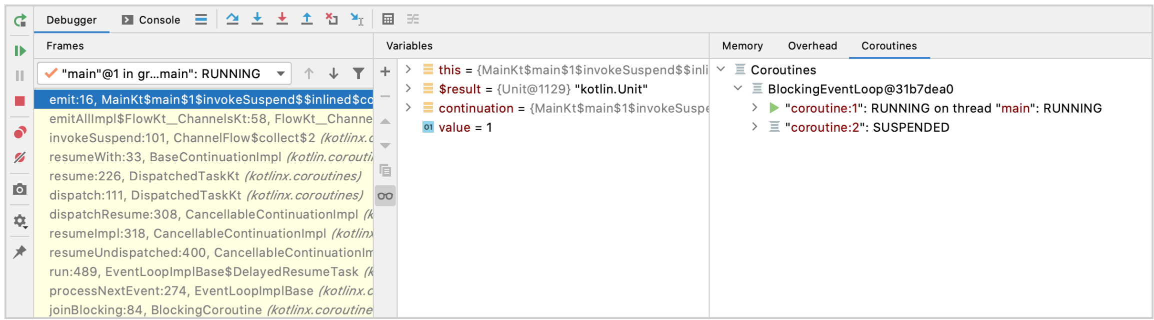


构建一个控制台应用程序

这时会出现 **Debug** 工具窗口.

在 **Coroutines** 页中, 你可以看到存在 2 个协程并发运行. 由于调用了 `buffer()` 函数, 数据流收取者和发射者运行在独立的协程中. `buffer()` 函数会缓存数据流中发射的值. 发射者协程状态为 **RUNNING**, 而收取者协程状态为 **SUSPENDED**.

3. 点击 **Debug** 工具窗口中的 **Resume Program**, 回复调试器运行.



调试协程

现在收取者协程状态为 RUNNING, 而发射者协程状态为 SUSPENDED.

你可以深入挖掘各个协程的信息, 来调试你的代码.

序列化

最终更新: 2024/09/10

序列化(serialization) 是指将应用程序使用的数据转换为一种格式, 使它可以通过网络传输, 或保存到数据库或文件. 相应的, *反序列化(deserialization)* 是指相反的过程, 从外部读取数据, 将它转换为运行时对象. 这两种功能结合到一起, 就是大多数应用程序与第三方交换数据时必不可少的部分.

有些序列化格式, 比如 JSON (https://www.json.org/json-en.html) 和 protocol buffers (https://developers.google.com/protocol-buffers) 是非常通用的. 它们独立于编程语言和运行平台, 可以用来在各种现代语言编写的系统之间交换数据.

在 Kotlin 中, 数据序列化工具是一个单独的组件, kotlinx.serialization (https://github.com/Kotlin/kotlinx.serialization). 其中包含几个主要部分: `org.jetbrains.kotlin.plugin.serialization` Gradle plugin, 运行库, 以及编译器插件.

编译器插件, `kotlinx-serialization-compiler-plugin` 和 `kotlinx-serialization-compiler-plugin-embeddable`, 直接发布到 Maven Central. 第 2 个插件用来与 `kotlin-compiler-embeddable` artifact 配合使用, 对于脚本 artifact, 它是默认选项. Gradle 会把编译器插件添加为你的项目的编译器参数.

库

`kotlinx.serialization` 提供了一组库, 用于所有支持的平台 – JVM, JavaScript, Native – 而且支持很多序列化格式 – JSON, CBOR, protocol buffers, 以及其他格式. 关于所支持的序列化格式的完整列表, 请参见下文.

所有的 Kotlin 序列化库都属于 `org.jetbrains.kotlinx:` 组. 库名称以 `kotlinx-serialization-` 开头, 后缀对应于序列化格式. 比如:

- `org.jetbrains.kotlinx:kotlinx-serialization-json`, 为 Kotlin 项目提供 JSON 序列化功能.
- `org.jetbrains.kotlinx:kotlinx-serialization-cbor`, 提供 CBOR 序列化功能.

针对各编译平台的 artifact 会自动选择; 你不需要手动添加. 在 JVM, JS, Native, 和跨平台项目中, 可以使用相同的依赖项.

注意, `kotlinx.serialization` 库使用单独的版本结构, 与 Kotlin 本身的版本不同. 最新的发布版本请参见 GitHub (https://github.com/Kotlin/kotlinx.serialization/releases) 的版本列表.

格式

`kotlinx.serialization` 包括针对多种序列化格式的库:

- JSON (https://www.json.org/): `kotlinx-serialization-json` (https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#json)
- Protocol buffers (https://developers.google.com/protocol-buffers): `kotlinx-serialization-protobuf` (https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#protobuf)
- CBOR (https://cbor.io/): `kotlinx-serialization-cbor` (https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#cbor)
- Properties (https://en.wikipedia.org/wiki/.properties): `kotlinx-serialization-properties` (https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#properties)
- HOCON (https://github.com/lightbend/config/blob/master/HOCON.md): `kotlinx-serialization-hocon` (https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md#hocon) (只限于 JVM 平台)

注意, 除 JSON 序列化(`kotlinx-serialization-json`)之外, 其他所有库都处于实验状态 (Kotlin 各部分组件的稳定性), 也就是说, 它们的 API 可能会发生变化, 不另行通知.

此外还有社区维护的库, 支持更多序列化格式, 比如 YAML (https://yaml.org/) 或 Apache Avro (https://avro.apache.org/). 关于可用的序列化格式, 详情请参见 `kotlinx.serialization` 文档 (https://github.com/Kotlin/kotlinx.serialization/blob/master/formats/README.md).

示例: JSON 序列化

下面我们来看一看如何将 Kotlin 对象序列化为 JSON.

添加插件和依赖项

开始之前, 你需要配置你的构建脚本, 使你的项目能够使用 Kotlin 序列化工具:

1. 应用 Kotlin 序列化 Gradle 插件 `org.jetbrains.kotlin.plugin.serialization` (在 Kotlin Gradle

DSL 中是 `kotlin("plugin.serialization")`).

Kotlin

```
plugins {
    kotlin("jvm") version "1.9.23"
    kotlin("plugin.serialization") version "1.9.23"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.9.23'
    id 'org.jetbrains.kotlin.plugin.serialization' version
'1.9.23'
}
```

2. 添加 JSON 序列化库的依赖项: `org.jetbrains.kotlinx:kotlinx-serialization-json:1.6.0`

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-
serialization-json:1.6.0")
}
```

Groovy

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-
serialization-json:1.6.0'
}
```

现在, 可以在你的代码中使用序列化 API 了. API 所在的包是 `kotlinx.serialization`, 以及各个格式专用的子包, 比如 `kotlinx.serialization.json`.

序列化和反序列化 JSON

1. 对一个类添加 `@Serializable` 注解, 使它可以被序列化.

```
import kotlinx.serialization.Serializable

@Serializable
data class Data(val a: Int, val b: String)
```

2. 调用函数 `Json.encodeToString()`, 序列化这个类的实例.

```
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import kotlinx.serialization.encodeToString

@Serializable
data class Data(val a: Int, val b: String)

fun main() {
  val json = Json.encodeToString(Data(42, "str"))
}
```

结果, 你会得到一个 JSON 格式的字符串, 其中包含这个对象的状态: `{"a": 42, "b": "str"}`

> 也可以通过单次函数调用, 序列化对象的集合, 比如 List:
>
> ```
> val dataList = listOf(Data(42, "str"), Data(12, "test"))
> val jsonList = Json.encodeToString(dataList)
> ```

3. 使用 `decodeFromString()` 函数, 从 JSON 字符串中反序列化对象:

```
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import kotlinx.serialization.decodeFromString
```

```kotlin
@Serializable
data class Data(val a: Int, val b: String)

fun main() {
  val obj = Json.decodeFromString<Data>("""{"a":42, "b":
"str"}""")
}
```

就是这样! 你已经成功的将对象序列化为 JSON 字符串, 然后将 JSON 字符串反序列化为对象.

下一步

关于 Kotlin 中的序列化, 更多详情请阅读 Kotlin 序列化指南 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/serialization-guide.md).

你可以阅读以下资料, 学习 Kotlin 序列化的不同方面:

- Kotlin 序列化及其核心概念 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/basic-serialization.md)
- Kotlin 的内建可序列化类 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/builtin-classes.md)
- 序列化器的更多详情, 并学习如何创建自定义的序列化器 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/serializers.md)
- Kotlin 如何处理多态序列化 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/polymorphism.md#open-polymorphism)
- Kotlin 序列化如何处理 JSON 的各种功能 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/json.md#json-elements)
- Kotlin 支持的实验性序列化格式 (https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/formats.md)

Lincheck 指南

最终更新: 2024/09/10

Lincheck 是一个实用的而且用户友好的框架, 用于在 JVM 平台上测试并发算法. 它提供了一个简单的, 声明式的方式, 来编写并发测试.

使用 Lincheck 框架, 不需要描述如何执行测试, 你可以通过声明所有需要验证的操作, 以及要求的正确性属性, 来指定 *测试什么*. 这样做的结果是, 一个通常的并发 Lincheck 测试只包含大约 15 行代码.

给定一个操作列表, Lincheck 会自动完成以下工作:

- 生成一组随机并发场景.
- 使用压力测试, 或绑定模型检查, 来验证这些场景.
- 验证每个调用的结果满足要求的正确性属性(默认设置是线性一致).

将 Lincheck 添加到你的项目

要使用 Lincheck, 需要在 Gradle 配置中包含对应的仓库和依赖项. 请在你的 `build.gradle(.kts)` 文件中, 添加以下代码:

Kotlin

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation("org.jetbrains.kotlinx:lincheck:2.28")
}
```

Groovy

```
repositories {
    mavenCentral()
```

```
}

dependencies {
    testImplementation "org.jetbrains.kotlinx:lincheck:2.28"
}
```

探索 Lincheck 的功能

本向导将会帮助你熟悉 Lincheck 框架, 并通过示例程序学习使用最有用的功能特性. 请按照以下步骤学习 Lincheck 的功能特性:

1. 使用 Lincheck 编写你的第一个测试 (使用 Lincheck 编写你的第一个测试)
2. 选择你的测试策略 (压力测试与模型检查)
3. 配置操作参数 (操作参数)
4. 考虑常见的算法约束 (数据结构约束)
5. 检查算法的非阻塞进度保证(non-blocking progress guarantee) (进度保证)
6. 定义算法的顺序规格(sequential specification) (顺序规格)

其他参考资料

- "我们如何测试 Kotlin Coroutine 中的并发算法", Nikita Koval: 视频 (https://youtu.be/jZqkWfa11Js). KotlinConf 2023
- "Lincheck: 在 JVM 上测试并发程序" 由 Maria Sokolova 主持的研讨会: 视频第 1 部分 (https://www.youtube.com/watch?v=YNtUK9GK4pA), 视频第 2 部分 (https://www.youtube.com/watch?v=EW7mkAOErWw). Hydra 2021

使用 Lincheck 编写你的第一个测试

最终更新: 2024/09/10

本教程演示如何编写你的第一个 Lincheck 测试, 设置 Lincheck 框架, 并使用它的基本 API. 你将会创建一个新的 IntelliJ IDEA 项目, 其中包含不正确的并发计数器实现, 为它编写一个测试, 然后查找并分析 bug.

创建一个项目

在 IntelliJ IDEA 中, 打开一个既有的 Kotlin 项目, 或 创建一个新项目 (Kotlin/JVM 入门). 创建项目时, 使用 Gradle 构建系统.

添加需要的依赖项

1. 打开 `build.gradle(.kts)` 文件, 确认参考列表中添加了 `mavenCentral()`.

2. 在 Gradle 配置中添加以下依赖项:

   Kotlin

```
repositories {
    mavenCentral()
}

dependencies {
    // Lincheck 依赖项
    testImplementation("org.jetbrains.kotlinx:lincheck:2.28")
    // 这个依赖项允许你使用 kotlin.test 和 JUnit:
    testImplementation("junit:junit:4.13")
}
```

   Groovy

```
repositories {
    mavenCentral()
```

```
}

dependencies {
    // Lincheck 依赖项
    testImplementation "org.jetbrains.kotlinx:lincheck:2.28"
    // 这个依赖项允许你使用 kotlin.test 和 JUnit:
    testImplementation "junit:junit:4.13"
}
```

编写一个并发的计数器, 并运行测试

1. 在 `src/test/kotlin` 目录中, 创建 `BasicCounterTest.kt` 文件, 并添加以下代码, 这是一个有 bug 的并发计数器, 然后为它编写一个 Lincheck 测试:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}

class BasicCounterTest {
    private val c = Counter() // 初始状态

    // 对计数器的操作
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()
```

```
    @Test // JUnit
    fun stressTest() = StressOptions().check(this::class) // 奇迹发
生在这里
}
```

Lincheck 测试会自动完成以下工作:

- 使用指定的 inc() 和 get() 操作生成一些随机的并发场景.
- 对生成的每个场景执行一系列调用.
- 验证每个调用的结果是否正确.

2. 运行上面的测试, 你将会看到以下错误:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |
```

这里, Lincheck 发现了测试运行结果违反了计数器的原子性 – 两个并发的增加操作返回了相同的结果 1. 这代表其中一个增加操作丢失了, 计数器的行为不正确.

追踪错误的运行结果

除了显示错误的运行结果之外, Lincheck 还提供了一种追踪错误原因的方法. 可以通过 模型检查 ("模型检查" in "压力测试与模型检查") 测试策略来使用这个功能, 这个测试策略使用有限次数的上下文切换来对多次执行进行检验.

1. 要切换测试策略, 请 options 类型从 StressOptions() 替换为 ModelCheckingOptions(). 修改后的 BasicCounterTest 类大致如下:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import
org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*
```

```
class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}

class BasicCounterTest {
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test
    fun modelCheckingTest() =
ModelCheckingOptions().check(this::class)
}
```

2. 再次运行测试. 你将会得到测试运行中导致错误结果的追踪信息:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |

The following interleaving leads to the error:
| ----------------------------------------------------------------
----- |
| Thread 1 |                         Thread  2
|
| ----------------------------------------------------------------
----- |
|          | inc()
```

```
|
|           |   inc(): 1 at
BasicCounterTest.inc(BasicCounterTest.kt:18) |
|           |     value.READ: 0 at
Counter.inc(BasicCounterTest.kt:10)    |
|           |     switch
|
| inc(): 1 |
|
|           |     value.WRITE(1) at
Counter.inc(BasicCounterTest.kt:10)   |
|           |     value.READ: 1 at
Counter.inc(BasicCounterTest.kt:10)    |
|           |   result: 1
|
| -----------------------------------------------------------------
----- |
```

根据这个追踪信息信息, 发生了以下事件:

- **T2**: 第 2 个线程开始了 `inc()` 操作, 读取当前的计数器值 (`value.READ: 0`), 然后暂停.
- **T1**: 第 1 个线程执行 `inc()`, 返回 `1`, 然后结束.
- **T2**: 第 2 个线程恢复运行, 对前面得到的计数器值加 1, 错误的将计数器更新为 `1`.

> 请在这里查看完整代码 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/BasicCounterTest.kt).

测试 Java 标准库

下面我们来发现一个 Java 标准库的 `ConcurrentLinkedDeque` 中的 bug. 下面的 Lincheck 测试会发现向双向队列头部删除和添加一个元素时发生的竞争情况:

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import
```

```
org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*
import java.util.concurrent.*

class ConcurrentDequeTest {
    private val deque = ConcurrentLinkedDeque<Int>()

    @Operation
    fun addFirst(e: Int) = deque.addFirst(e)

    @Operation
    fun addLast(e: Int) = deque.addLast(e)

    @Operation
    fun pollFirst() = deque.pollFirst()

    @Operation
    fun pollLast() = deque.pollLast()

    @Operation
    fun peekFirst() = deque.peekFirst()

    @Operation
    fun peekLast() = deque.peekLast()

    @Test
    fun modelCheckingTest() =
ModelCheckingOptions().check(this::class)
}
```

运行 `modelCheckingTest()`. 测试将会失败, 输出信息如下:

```
= Invalid execution results =
| ---------------------------------------- |
|      Thread 1     |       Thread 2       |
| ---------------------------------------- |
| addLast(22): void |                      |
| ---------------------------------------- |
| pollFirst(): 22   | addFirst(8): void    |
```

```
|                   | peekLast(): 22 [-,1] |
| -------------------------------------- |

---
All operations above the horizontal line | ----- | happen before
those below the line
---
Values in "[..]" brackets indicate the number of completed
operations
in each of the parallel threads seen at the beginning of the current
operation
---

The following interleaving leads to the error:
| ------------------------------------------------------------------
--------------------------------------------------------------- |
|                                                 Thread 1
|       Thread 2       |
| ------------------------------------------------------------------
--------------------------------------------------------------- |
| pollFirst()
|                       |
|   pollFirst(): 22 at
ConcurrentDequeTest.pollFirst(ConcurrentDequeTest.kt:17)
|                       |
|     first(): Node@1 at
ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:915)
|                       |
|     item.READ: null at
ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:917)
|                       |
|     next.READ: Node@2 at
ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:925)
|                       |
|     item.READ: 22 at
ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:917)
|                       |
|     prev.READ: null at
ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:919)
```

```
|                       |
|     switch
|                       |
|
| addFirst(8): void     |
|
| peekLast(): 22        |
|     compareAndSet(Node@2,22,null): true at
ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:920) |
|
|     unlink(Node@2) at
ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:921)
|                       |
|   result: 22
|                       |
| ------------------------------------------------------------------
--------------------------------------------------------------- |
```

请在这里查看完整代码 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/ConcurrentLinkedDequeTest.kt).

下一步

选择 你的测试策略, 并对测试的运行进行配置 (压力测试与模型检查).

参见

- 如何生成操作参数 (操作参数)
- 常见的算法约束 (数据结构约束)
- 检查非阻塞进度保证(non-blocking progress guarantee) (进度保证)
- 定义算法的顺序规格(sequential specification) (顺序规格)

压力测试与模型检查

最终更新: 2024/09/10

Lincheck 提供了 2 种测试策略: 压力测试与模型检查. 下面我们使用 前一章 (使用 Lincheck 编写你的第一个测试) 中在 `BasicCounterTest.kt` 文件中编写的 `Counter`, 来学习这 2 种策略的内部机制:

```
class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}
```

压力测试

编写一个压力测试

我们为 `Counter` 创建一个并发压力测试, 步骤如下:

1. 创建 `CounterTest` 类.
2. 在这个类中, 添加域 `c`, 类型为 `Counter`, 这样会在构建器中创建一个 `Counter` 的实例.
3. 列出计数器的操作, 使用 `@Operation` 注解标记这些操作, 将它们的实现代理到 `c`.
4. 使用 `StressOptions()`, 指定压力测试策略.
5. 调用 `StressOptions.check()` 函数, 运行测试.

最后的代码大致如下:

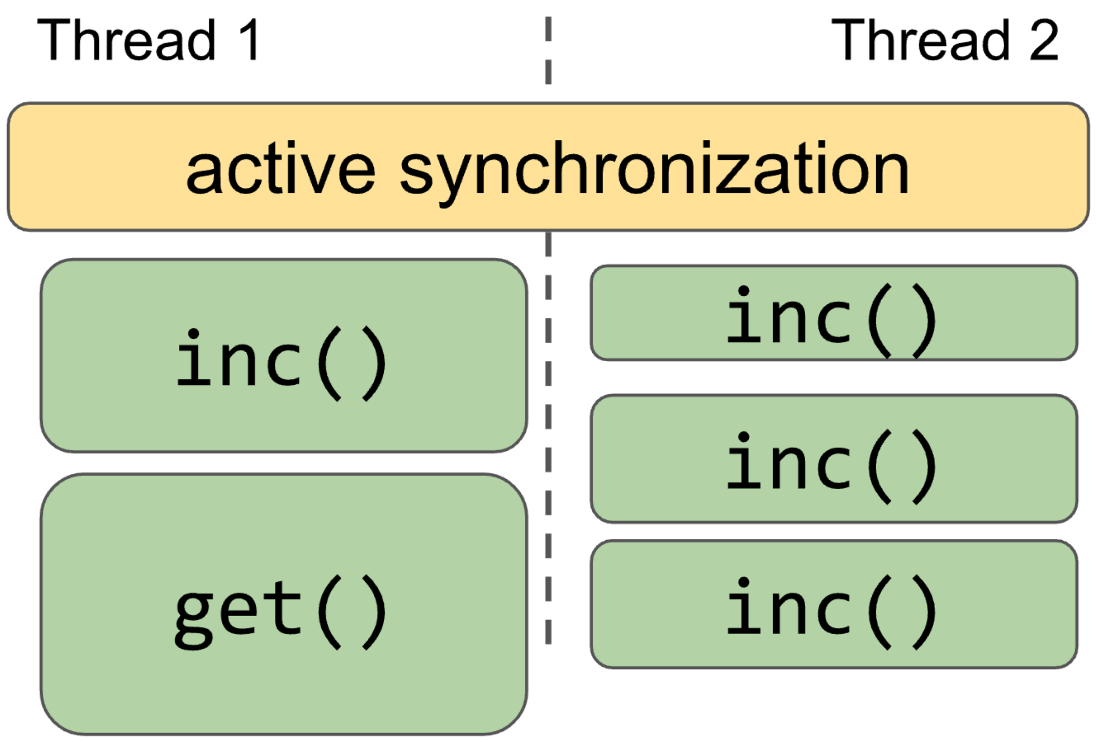
```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*
```

```kotlin
class CounterTest {
    private val c = Counter() // 初始化状态

    // 计数器上的操作
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test // 运行测试
    fun stressTest() = StressOptions().check(this::class)
}
```

压力测试的工作原理

首先, Lincheck 使用标注了 `@Operation` 注解的操作生成一组并发场景. 然后, 它启动原生的线程, 开始时同步这些线程, 以保证操作同时发生. 最后, Lincheck 在这些原生的线程上多次执行每个场景, 期待发现导致不正确结果的数据冲突.

下图说明 Lincheck 如何执行生成的并发场景:



计数器压力测试的执行情况

模型检查

压力测试的主要问题是, 你可能需要耗费几个小时才能理解如何重现你发现的 bug. 为了帮助你调查 bug, Lincheck 支持有限模型检查, 它可以自动提供数据冲突, 来重现 bug.

模型检查测试的构建方式与压力测试一样. 只需要将指定测试策略的 `StressOptions()` 替换为 `ModelCheckingOptions()`.

编写一个模型检查测试

要将压力测试策略修改为模型检查策略, 请将你的测试中的 `StressOptions()` 替换为 `ModelCheckingOptions()`:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import
org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*
```

```
class CounterTest {
    private val c = Counter() // 初始化状态

    // 计数器上的操作
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test // 运行测试
    fun modelCheckingTest() =
ModelCheckingOptions().check(this::class)
}
```

⚠ 在 Java 9 或更高版本上要使用模型检查策略, 请添加以下 JVM 属性:

```
--add-opens java.base/jdk.internal.misc=ALL-UNNAMED
--add-exports java.base/jdk.internal.util=ALL-UNNAMED
```

如果测试代码使用了 `java.util` 包中的类, 会需要这些属性, 因为有些类的内部实现使用了 `jdk.internal.misc.Unsafe`, 或其他类似的内部类. 如果你是 Gradle, 请在 `build.gradle.kts` 文件添加下面的内容:

```
tasks.withType<Test> {
    jvmArgs(
        "--add-opens", "java.base/java.lang=ALL-UNNAMED",
        "--add-opens", "java.base/jdk.internal.misc=ALL-
UNNAMED",
        "--add-exports", "java.base/jdk.internal.util=ALL-
UNNAMED",
        "--add-exports", "java.base/sun.security.action=ALL-
UNNAMED"
    )
}
```

模型检查的工作原理

要重现复杂并发算法中的大多数 bug, 可以使用典型的数据冲突, 同时将代码的执行切换从一个线程切换到另一个线程. 此外, 对于弱内存模型的模型检查器非常复杂, 因此 Lincheck 使用_循序一致性(sequential consistency)内存模型_ 下的有限模型检查.

简单的说, Lincheck 会分析所有的数据冲突, 从一个上下文切换开始, 然后是两个, 持续这个过程, 直到检测到指定数量的数据冲突. 这个策略可以使用最少的上下文切换次数找到不正确的数据冲突, 使得后面的 bug 调查更加容易.

为了控制运行, Lincheck 会在测试代码中插入特殊的切换点. 这些切换点标识可以执行上下文切换的位置. 本质上, 这些切换点是对共享内存的访问, 例如在 JVM 中读取或更新域和数组元素, 以及 `wait/notify` 和 `park/unpark` 调用. 为了插入切换点, Lincheck 会使用 ASM 框架, 在运行过程中转换测试代码, 对已有的代码添加内部函数调用.

由于模型检查策略会控制执行过程, Lincheck 能够对导致错误数据冲突的情况提供追踪信息, 实际运用中这些信息会非常有用. 在 使用 Lincheck 编写你的第一个测试 ("追踪错误的运行结果" in "使用 Lincheck 编写你的第一个测试") 教程中, 你可以看到对 `Counter` 的不正确执行的追踪信息的示例.

哪个测试策略更好?

模型检查策略 更适合在循序一致性内存模型下查找 bug, 因为它能够确保更好的覆盖率, 并在找到错误时提供执行失败的追踪信息.

尽管 *压力测试* 不保证覆盖率, 对算法中由于底层效应造成的 bug, 例如缺少 `volatile` 修饰符, 这种测试策略对这类 bug 的检查仍然很有帮助. 对于那些需要大量上下文切换才能重现的少见 bug, 压力测试也非常有用, 而模型检查策略, 由于目前的限制, 还无法分析这类 bug.

配置测试策略

要配置测试策略, 请在 `<TestingMode>Options` 类中设置选项.

1. 为 `CounterTest` 的场景生成和运行设置选项:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class CounterTest {
```

```
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test
    fun stressTest() = StressOptions() // 压力测试选项:
        .actorsBefore(2) // 并行运行部分之前的操作数量
        .threads(2) // 并行运行部分中的线程数量
        .actorsPerThread(2) // 并行运行部分的每个线程中的操作数量
        .actorsAfter(1) // 并行运行部分之后的操作数量
        .iterations(100) // 生成 100 个随机的并发场景
        .invocationsPerIteration(1000) // 对生成的每个场景运行 1000 次
        .check(this::class) // 运行测试
}
```

2. 在此运行 `stressTest()`, Lincheck 会生成类似于下面的场景:

```
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc()    |          |
| inc()    |          |
| ------------------- |
| get()    | inc()    |
| inc()    | get()    |
| ------------------- |
| inc()    |          |
| ------------------- |
```

   这里, 在并行运行部分之前有 2 个操作, 在并行运行部分中, 对每个操作都有 2 个线程, 最后是 1 个操作.

你也可以通过同样的方式来配置模型检查测试.

场景最小化

你可能已经注意到了, 检测到的错误通常代表的场景比在测试配置中指定的场景要小. Lincheck 会尝试对错误进行最小化, 努力删除操作, 同时又确保测试失败.

对上面的计数器测试最小化后的场景如下:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc()    | inc()    |
| ------------------- |
```

由于对更小的场景更容易分析, 因此默认会启用场景最小化. 要关闭这个功能, 请对 `[Stress, ModelChecking]Options` 配置添加 `minimizeFailedScenario(false)` 选项.

对数据结构状态输出日志

对于调试 bug 另一个非常有用的功能是 *状态日志*. 在分析导致错误的数据冲突时, 你通常会在纸上画出数据结构变化图, 在每个事件后修改它的状态. 为了自动完成这个过程, 你可以提供一个特别的方法, 返回数据结构的一个 `String` 表达, 然后 Lincheck 可以在每个修改数据结构的事件之后打印状态数据状态.

为了做到这一点, 请定义一个没有参数的方法, 并标注 `@StateRepresentation` 注解. 和这个方法应该线程安全, 无阻塞, 而且绝不修改数据结构.

1. 在 `Counter` 示例中, `String` 表达仅仅是计数器的值. 因此, 要在追踪信息中打印计数器状态, 请对 `CounterTest` 添加 `stateRepresentation()` 函数:

   ```
   import org.jetbrains.kotlinx.lincheck.annotations.*
   import org.jetbrains.kotlinx.lincheck.check
   import
   org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
   import org.junit.Test

   class CounterTest {
       private val c = Counter()
   ```

```
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @StateRepresentation
    fun stateRepresentation() = c.get().toString()

    @Test
    fun modelCheckingTest() =
ModelCheckingOptions().check(this::class)
}
```

2. 运行 `modelCheckingTest()`, 确认 `Counter` 的状态会在修改计数器状态的切换点被打印输出 (输出的文字以 `STATE:` 开始):

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| STATE: 0            |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |
| STATE: 1            |
| ------------------- |

The following interleaving leads to the error:
| -----------------------------------------------------------------
---- |
| Thread 1 |                         Thread 2
|
| -----------------------------------------------------------------
---- |
|          | inc()
|
|          |   inc(): 1 at CounterTest.inc(CounterTest.kt:10)
|
```

```
|           |     value.READ: 0 at
Counter.inc(BasicCounterTest.kt:10)  |
|           |     switch
|
| inc(): 1 |
|
| STATE: 1 |
|
|           |     value.WRITE(1) at
Counter.inc(BasicCounterTest.kt:10) |
|           |     STATE: 1
|
|           |     value.READ: 1 at
Counter.inc(BasicCounterTest.kt:10)  |
|           |   result: 1
|
| ----------------------------------------------------------------
---- |
```

对于压力测试的情况, Lincheck 会在场景的并行运行部分之前和之后打印状态信息, 还会在结束时打印.

- 查看 这些示例的完整代码 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/CounterTest.kt)
- 查看更多 测试示例 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/StackTest.kt)

下一步

学习如何 配置传递给操作的参数 (操作参数), 以及在什么情况下需要如此.

操作参数

最终更新: 2024/09/10

本教程中, 你将会学习如何配置操作参数.

以下面这个简单的 `MultiMap` 实现为例子. 它内部使用 `ConcurrentHashMap`, 其中保存的是一个值列表:

```
import java.util.concurrent.*

class MultiMap<K, V> {
    private val map = ConcurrentHashMap<K, List<V>>()

    // 维护与指定的 key 关联的值列表.
    fun add(key: K, value: V) {
        val list = map[key]
        if (list == null) {
            map[key] = listOf(value)
        } else {
            map[key] = list + value
        }
    }

    fun get(key: K): List<V> = map[key] ?: emptyList()
}
```

这个 `MultiMap` 实现是线性一致的吗? 如果不是, 在访问小范围的 key 时, 并发操作同一个 key 的可能性会变高, 因此这种情况下更可能检测到错误的数据.

对这样的情况, 我们为 `key: Int` 参数配置生成器:

1. 声明 `@Param` 注解.
2. 指定整数生成器类: `@Param(gen = IntGen::class)`. Lincheck 默认对几乎所有的基本类型和字符串支持随机参数的生成器.
3. 使用字符串配置 `@Param(conf = "1:2")`, 定义要生成的值的范围.
4. 指定参数配置的名称 (`@Param(name = "key")`), 以便在多个操作中共用这个配置.

下面是对 `MultiMap` 的压力测试, 它会为 `add(key, value)` 和 `get(key)` 操作生成 key 值, 范围是 `[1..2]`:

```
import java.util.concurrent.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.paramgen.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import
org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*

class MultiMap<K, V> {
    private val map = ConcurrentHashMap<K, List<V>>()

    // 维护与指定的 key 关联的值列表.
    fun add(key: K, value: V) {
        val list = map[key]
        if (list == null) {
            map[key] = listOf(value)
        } else {
            map[key] = list + value
        }
    }

    fun get(key: K): List<V> = map[key] ?: emptyList()
}

@Param(name = "key", gen = IntGen::class, conf = "1:2")
class MultiMapTest {
    private val map = MultiMap<Int, Int>()

    @Operation
    fun add(@Param(name = "key") key: Int, value: Int) =
map.add(key, value)

    @Operation
    fun get(@Param(name = "key") key: Int) = map.get(key)
```

```
    @Test
    fun stressTest() = StressOptions().check(this::class)

    @Test
    fun modelCheckingTest() =
ModelCheckingOptions().check(this::class)
}
```

5. 运行 `stressTest()`, 会看到以下输出:

```
= Invalid execution results =
| ---------------------------------- |
|    Thread 1     |     Thread 2     |
| ---------------------------------- |
| add(2, 0): void | add(2, -1): void |
| ---------------------------------- |
| get(2): [0]     |                  |
| ---------------------------------- |
```

6. 最后, 运行 `modelCheckingTest()`. 它会失败, 输出如下:

```
= Invalid execution results =
| ---------------------------------- |
|    Thread 1     |     Thread 2     |
| ---------------------------------- |
| add(2, 0): void | add(2, -1): void |
| ---------------------------------- |
| get(2): [-1]    |                  |
| ---------------------------------- |

---
All operations above the horizontal line | ----- | happen before
those below the line
---

The following interleaving leads to the error:
| -------------------------------------------------------------------
------ |
|    Thread 1     |                       Thread 2
```

```
|
| ----------------------------------------------------------------
------ |
|                  | add(2, -1)
|
|                  |   add(2,-1) at
MultiMapTest.add(MultiMap.kt:31)      |
|                  |     get(2): null at
MultiMap.add(MultiMap.kt:15)     |
|                  |     switch
|
| add(2, 0): void |
|
|                  |     put(2,[-1]): [0] at
MultiMap.add(MultiMap.kt:17) |
|                  |   result: void
|
| ----------------------------------------------------------------
------ |
```

由于 key 值范围很小, Lincheck 很快发现了竞争情况: 当 2 个值并发的添加到同一个 key 值的时候, 其中 1 个值可能会被覆盖, 并丢失.

> 请在这里查看完整代码 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/MultiMapTest.kt).

下一步

学习如何测试设置了 执行中访问约束 (数据结构约束) 的数据结构, 例如单生成者(single-producer)单消费者(single-consumer) 队列.

数据结构约束

最终更新: 2024/09/10

有些数据结构可能要求一部分操作不能并发执行, 例如单生产者(single-producer)/单消费者(single-consumer) 队列. Lincheck 对这样的约束提供了现成的支持, 会根据约束条件生成并发场景.

以 JCTools 库 (https://github.com/JCTools/JCTools) 中的 单消费者队列 (https://github.com/JCTools/JCTools/blob/66e6cbc9b88e1440a597c803b7df9bd1d60219f6/jctools-core/src/main/java/org/jctools/queues/atomic/MpscLinkedAtomicQueue.java) 为例. 我们来编写一个测试, 检查它的 `poll()`, `peek()`, 以及 `offer(x)` 操作的正确性.

在你的 `build.gradle(.kts)` 文件中, 添加 JCTools 依赖项:

Kotlin

```
dependencies {
    // jctools 依赖项
    testImplementation("org.jctools:jctools-core:3.3.0")
}
```

Groovy

```
dependencies {
    // jctools 依赖项
    testImplementation "org.jctools:jctools-core:3.3.0"
}
```

为了满足单消费者约束条件, 需要确保所有的 `poll()` 和 `peek()` 消费操作都只会从单个线程中调用. 为了做到这一点, 我们可以将对应的 `@Operation` 注解的 `nonParallelGroup` 参数设置为相同的值, 例如, `"consumers"`.

下面是测试代码:

```
import org.jctools.queues.atomic.*
import org.jetbrains.kotlinx.lincheck.annotations.*
```

```kotlin
import org.jetbrains.kotlinx.lincheck.check
import
org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class MPSCQueueTest {
    private val queue = MpscLinkedAtomicQueue<Int>()

    @Operation
    fun offer(x: Int) = queue.offer(x)

    @Operation(nonParallelGroup = "consumers")
    fun poll(): Int? = queue.poll()

    @Operation(nonParallelGroup = "consumers")
    fun peek(): Int? = queue.peek()

    @Test
    fun stressTest() = StressOptions().check(this::class)

    @Test
    fun modelCheckingTest() =
ModelCheckingOptions().check(this::class)
}
```

下面是为这个测试生成的并发场景的例子:

```
= Iteration 15 / 100 =
| --------------------- |
| Thread 1  | Thread 2  |
| --------------------- |
| poll()    |           |
| poll()    |           |
| peek()    |           |
| peek()    |           |
| peek()    |           |
| --------------------- |
| offer(-1) | offer(0)  |
```

```
| offer(0)  | offer(-1) |
| peek()    | offer(-1) |
| offer(1)  | offer(1)  |
| peek()    | offer(1)  |
| --------------------- |
| peek()    |           |
| offer(-2) |           |
| offer(-2) |           |
| offer(2)  |           |
| offer(-2) |           |
| --------------------- |
```

注意, 所有的消费操作 `poll()` 和 `peek()` 的调用, 都通过单个线程执行, 因此满足了 "单消费者" 约束.

> 请在这里查看完整代码 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/MPSCQueueTest.kt).

下一步

学次如何使用模型检查策略来 检查你的算法的进度保证 (进度保证).

进度保证

最终更新: 2024/09/10

很多并发算法会提供非阻塞的进度保证, 例如无锁(lock-freedom)和无等待(wait-freedom). 由于算法通常比较复杂, 因此很容易加入 bug, 导致算法阻塞. 使用模型检查策略, Lincheck 可以帮助你找到这种存活 bug.

要检查算法的进度保证性, 请在 `ModelCheckingOptions()` 中启用 `checkObstructionFreedom` 选项:

```
ModelCheckingOptions().checkObstructionFreedom()
```

创建一个 `ConcurrentMapTest.kt` 文件. 然后添加下面的测试代码, 它能够检测出 Java 标准库中的 `ConcurrentHashMap::put(key: K, value: V)` 是一个阻塞操作:

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import
org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*
import java.util.concurrent.*

class ConcurrentHashMapTest {
    private val map = ConcurrentHashMap<Int, Int>()

    @Operation
    fun put(key: Int, value: Int) = map.put(key, value)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .actorsBefore(1) // 初始化 HashMap
        .actorsPerThread(1)
        .actorsAfter(0)
        .minimizeFailedScenario(false)
        .checkObstructionFreedom()
        .check(this::class)
}
```

运行 modelCheckingTest(). 你会得到以下结果:

```
= Obstruction-freedom is required but a lock has been found =
| ---------------------- |
|  Thread 1  | Thread 2  |
| ---------------------- |
| put(1, -1) |           |
| ---------------------- |
| put(2, -2) | put(3, 2) |
| ---------------------- |

---
All operations above the horizontal line | ----- | happen before
those below the line
---

The following interleaving leads to the error:
| --------------------------------------------------------------------
--------------------------------------------------------------------
---------------------------------------------- |
|                                          Thread 1
|                                          Thread 2
|
| --------------------------------------------------------------------
--------------------------------------------------------------------
---------------------------------------------- |
|
| put(3, 2)
|
|
|   put(3,2) at ConcurrentHashMapTest.put(ConcurrentMapTest.kt:11)
|
|
|     putVal(3,2,false) at
ConcurrentHashMap.put(ConcurrentHashMap.java:1006)              |
|
|       table.READ: Node[]@1 at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1014)      |
|
```

```
|       tabAt(Node[]@1,0): Node@1 at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1018) |
|
|       MONITORENTER at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1031)              |
|
|       tabAt(Node[]@1,0): Node@1 at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1032) |
|
|       next.READ: null at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1046)           |
|
|       switch
|
| put(2, -2)
|
|
|   put(2,-2) at ConcurrentHashMapTest.put(ConcurrentMapTest.kt:11)
|
|
|     putVal(2,-2,false) at
ConcurrentHashMap.put(ConcurrentHashMap.java:1006)             |
|
|       table.READ: Node[]@1 at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1014)      |
|
|       tabAt(Node[]@1,0): Node@1 at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1018) |
|
|       MONITORENTER at
ConcurrentHashMap.putVal(ConcurrentHashMap.java:1031)              |
|
| ------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------- |
```

下面我们来为非阻塞的 `ConcurrentSkipListMap<K, V>` 添加一个测试，这个测试应该成功通过:

```kotlin
class ConcurrentSkipListMapTest {
    private val map = ConcurrentSkipListMap<Int, Int>()

    @Operation
    fun put(key: Int, value: Int) = map.put(key, value)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .checkObstructionFreedom()
        .check(this::class)
}
```

⚠ 共通的非阻塞进度保证包括 (从最强到最弱):

- **无等待(wait-freedom)**, 无论其他线程正在做什么, 每个操作都能够在有限次数的步骤内完成.
- **无锁(lock-freedom)**, 保证系统级别的进度, 至少一个操作能够在有限次数的步骤内完成, 其他某个操作可能会阻塞.
- **无阻塞(obstruction-freedom)**, 如果其他所有线程都暂停, 那么操作能够在有限次数的步骤内完成.

目前, Lincheck 只支持无阻塞(obstruction-freedom)的进度保证. 但是, 大多数现实生活中的存活 bug 都是由于添加了不正确的阻塞代码, 因此无阻塞(obstruction-freedom)检查 也有助于测试无锁(lock-freedom)和无等待(wait-freedom)算法.

- 请在这里查看示例的完整代码 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/ConcurrentMapTest.kt).
- 查看 另一个示例 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/ObstructionFreedomViolationTest.kt), 这里会对 Michael-Scott 队列实现测试它的进度保证.

下一步

学习如何对被测试的算法明确的 指定顺序规格 (顺序规格), 增加 Lincheck 测试的健壮性.

顺序规格

最终更新: 2024/09/10

为了确认算法实现了正确的顺序行为, 你可以为测试数据结构编写一个简单的顺序化实现, 用来定义算法的 *顺序规格(sequential specification)*.

> 使用这个功能, 你只需要编写单个测试, 而不必分别编写顺序测试和并发测试.

要指定需要验证的算法的顺序规格, 你需要:

1. 实现一个所有测试方法的顺序化版本.
2. 将带有顺序化实现的类传递给 `sequentialSpecification()` 选项:

   ```
   StressOptions().sequentialSpecification(SequentialQueue::class)
   ```

例如, 这里是一个测试, 它检查 Java 标准库的 `j.u.c.ConcurrentLinkedQueue` 类的正确性.

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*
import java.util.*
import java.util.concurrent.*

class ConcurrentLinkedQueueTest {
    private val s = ConcurrentLinkedQueue<Int>()

    @Operation
    fun add(value: Int) = s.add(value)

    @Operation
    fun poll(): Int? = s.poll()

    @Test
    fun stressTest() = StressOptions()
```

```
        .sequentialSpecification(SequentialQueue::class.java)
        .check(this::class)
}

class SequentialQueue {
    private val s = LinkedList<Int>()

    fun add(x: Int) = s.add(x)
    fun poll(): Int? = s.poll()
}
```

> 请在这里查看示例的完整代码 (https://github.com/Kotlin/kotlinx-lincheck/blob/guide/src/jvm/test/org/jetbrains/kotlinx/lincheck/test/guide/ConcurrentLinkedQueueTest.kt).

关键字与操作符

最终更新: 2024/09/10

硬关键字(Hard Keyword)

以下符号始终会被解释为关键字, 不能用作标识符(identifiers):

- `as`
  - 用于 类型转换 (""不安全的" 类型转换操作符" in "类型检查与类型转换").
  - 为 import 指定一个别名 (""导入(Import)" in "包(Package)与导入(Import)").
- `as?` 用于 安全的类型转换 (""安全的" (nullable) 类型转换操作" in "类型检查与类型转换").
- `break` 结束一个循环 (返回与跳转: break 与 continue).
- `class` 声明一个 类 (类).
- `continue` 跳转到最内层循环的下一次执行 (返回与跳转: break 与 continue).
- `do` 开始一个 do/while 循环 ("while 循环" in "条件与循环") (条件判定在后的循环).
- `else` 定义 if 表达式 ("if 表达式" in "条件与循环") 的一个分支, 这个分支在条件为 false 时执行.
- `false` 指定 布尔类型 (布尔(Boolean)类型) 的 'false' 值.
- `for` 开始一个 for 循环 ("for 循环" in "条件与循环").
- `fun` 声明一个 函数 (函数).
- `if` 开始一个 if 表达式 ("if 表达式" in "条件与循环").
- `in`
  - 指定 for 循环 ("for 循环" in "条件与循环") 的迭代对象.
  - 用作中缀操作符, 判断一个值是否在 一个值范围 (值范围(Range)与数列(Progression)) 之内, 或者是否属于一个集合, 或者是否属于其他 定义了 'contains' 方法 ("in 操作符" in "操作符重载") 的实体.
  - 在 when 表达式 ("when 表达式" in "条件与循环") 中做同样的判断.

  - 将一个类型参数标记为 反向类型变异 ("声明处的类型变异(Declaration-site variance)" in "泛型(Generic): in, out, where").

- `!in`

  - 用作操作符, 判断一个值是否 **不属于** 一个值范围 (值范围(Range)与数列(Progression)), 或者是否 **不属于** 一个集合, 或者是否 **不属于** 其他 定义了 'contains' 方法 ("in 操作符" in "操作符重载") 的实体.

  - 在 when 表达式 ("when 表达式" in "条件与循环") 中做同样的判断.

- `interface` 声明一个 接口 (接口(Interface)).

- `is`

  - 判断 一个值是不是某个类型 ("is 与 !is 操作符" in "类型检查与类型转换").

  - 在 when 表达式 ("when 表达式" in "条件与循环") 中做同样的判断.

- `!is`

  - 判断 一个值是否**不是**某个类型 ("is 与 !is 操作符" in "类型检查与类型转换").

  - 在 when 表达式 ("when 表达式" in "条件与循环") 中做同样的判断.

- `null` 是一个常数, 表示一个不指向任何对象的引用.

- `object` 同时声明一个类和它的对象实例 (对象表达式,对象声明,以及同伴对象).

- `package` 指定 当前源代码文件的包 (包(Package)与导入(Import)).

- `return` 从最内层的函数或匿名函数中返回 (返回与跳转: break 与 continue).

- `super`

  - 引用一个方法或属性在超类中的实现 ("调用超类中的实现" in "继承").

  - 在次级构造器中调用超类构造器 ("继承" in "类").

- `this`

  - 引用 当前接受者 (this 表达式).

  - 在次级构造器中调用同一个类的另一个构造器 ("构造器" in "类").

- `throw` 抛出一个异常 (异常(Exception)).

- `true` 指定 布尔类型 (布尔(Boolean)类型) 的 'true' 值.
- `try` 开始一个异常处理代码段 (异常(Exception)).
- `typealias` 声明一个 类型别名 (类型别名).
- `typeof` 保留, 将来使用.
- `val` 声明一个只读的 属性 (属性(Property)), 或者一个只读的 局部变量 ("变量" in "基本语法").
- `var` 声明一个可变的 属性 (属性(Property)), 或者一个可变的 局部变量 ("变量" in "基本语法").
- `when` 开始一个 when 表达式 ("when 表达式" in "条件与循环") (执行其中一个分支).
- `while` 开始一个 while 循环 ("while 循环" in "条件与循环") (条件判定在前的循环).

软关键字(Soft Keyword)

以下符号在适当的场合下可以是关键字, 在其他场合可以用作标识符:

- `by`
  - 将一个接口的实现委托给另一个对象 (委托).
  - 将一个属性的访问器函数实现委托给另一个对象 (委托属性).
- `catch` 开始一个 处理特定的异常类型 (异常(Exception)) 的代码段.
- `constructor` 声明一个 主构造器, 或次级构造器 ("构造器" in "类").
- `delegate` 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").
- `dynamic` 在 Kotlin/JS 代码中引用一个 动态类型 (动态类型).
- `field` 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").
- `file` 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").
- `finally` 开始一个 try 代码段结束时始终会被执行 (异常(Exception)) 的代码段.
- `get`
  - 声明 属性的取值方法 ("取值方法(Getter)与设值方法(Setter)" in "属性(Property)").
  - 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").

- `import` 从另一个包中将一个声明导入到当前源代码文件 (包(Package)与导入(Import)).
- `init` 开始一个 初始化代码段 ("构造器" in "类").
- `param` 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").
- `property` 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").
- `receiver` 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").
- `set`
  - 声明 属性的设值方法 ("取值方法(Getter)与设值方法(Setter)" in "属性(Property)").
  - 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").
- `setparam` 用作一种 注解的使用目标(target) ("注解的使用目标(Use-site Target)" in "注解").
- `value` 与 `class` 关键字一起使用, 声明一个 内联类(inline class) (内联的值类(Inline value class)).
- `where` 指定 泛型类型参数的约束 ("上界(Upper Bound)" in "泛型(Generic): in, out, where").

标识符关键字(Modifier Keyword)

以下符号在声明的标识符列表中用做关键字, 在其他场合可以用作标识符:

- `abstract` 将一个类或一个成员标注为 抽象元素 ("抽象类" in "类").
- `actual` 在 跨平台项目 (Kotlin Multiplatform) 中, 表示某个特定平台上的具体实现.
- `annotation` 声明一个 注解类 (注解).
- `companion` 声明一个 同伴对象 ("同伴对象(Companion Object)" in "对象表达式,对象声明,以及同伴对象").
- `const` 将一个属性标注为 编译期常数值 ("编译期常数值" in "属性(Property)").
- `crossinline` 禁止 传递给内联函数的 lambda 表达式中的非局部的返回 ("非局部返回(Non-local return)" in "内联函数(Inline Function)").
- `data` 指示编译器, 为类生成常用的成员函数 (数据类(Data Class)).
- `enum` 声明一个 枚举类 (枚举类).

- `expect` 标注一个 与平台相关的声明 (Kotlin Multiplatform), 在各个平台模块中, 需要存在对应的具体实现.
- `external` 标注一个声明在 Kotlin 代码之外 (可以通过 JNI ("在 Kotlin 中使用 JNI(Java Native Interface)" in "在 Kotlin 中调用 Java 代码") 实现, 或者用 JavaScript ("external 修饰符" in "在 Kotlin 中使用 JavaScript 代码") 实现).
- `final` 禁止 覆盖成员 ("方法的覆盖" in "继承").
- `infix` 允许使用 中缀标记法 ("中缀标记法(Infix notation)" in "函数") 来调用函数.
- `inline` 告诉编译器 将函数以及传递给函数的 lambda 表达式内联到函数的调用处 (内联函数(Inline Function)).
- `inner` 允许在 嵌套内 (嵌套类与内部类) 中引用外部类的实例.
- `internal` 将一个声明标注为 只在当前模块中可以访问 (可见度修饰符).
- `lateinit` 允许 在构造器之外初始化非 null 的属性 ("延迟初始化的(Late-Initialized)属性和变量" in "属性(Property)").
- `noinline` 关闭 对传递给内联函数的 lambda 表达式的内联 ("noinline" in "内联函数(Inline Function)").
- `open` 允许 继承类, 或者覆盖成员 ("继承" in "类").
- `operator` 将函数标记为 操作符重载, 或实现一个规约 (操作符重载).
- `out` 将类型参数标记为 协变的 ("声明处的类型变异(Declaration-site variance)" in "泛型(Generic): in, out, where").
- `override` 将成员标记为 对超类成员的覆盖 ("方法的覆盖" in "继承").
- `private` 将声明标记为 只在当前类中, 或当前源代码文件中可以访问 (可见度修饰符).
- `protected` 将声明标记为 只在当前类, 以及它的子类中可以访问 (可见度修饰符).
- `public` 将声明标记为 在任何位置都可以访问 (可见度修饰符).
- `reified` 将内联函数的类型参数标记为 在运行时刻可以访问 ("实体化的类型参数(Reified type parameter)" in "内联函数(Inline Function)").
- `sealed` 声明一个 封闭类 (封闭类(Sealed Class)与封闭接口(Sealed Interface)) (子类受到限制的类).

- `suspend` 将函数, 或 lambda 表达式, 标注为挂起函数, 或挂起lambda 表达式 (可在 协程 (协程(Coroutine)) 中使用).
- `tailrec` 将一个函数标注为 尾递归 ("尾递归函数(Tail recursive function)" in "函数") (允许编译器用迭代来代替递归).
- `vararg` 允许 对某个参数传递可变数量的参数值 ("不定数量参数(varargs)" in "函数").

特殊标识符

以下表述符在特定情况下由编译器定义, 在其他场合可以用作通常的标识符:

- `field` 在属性访问函数的内部, 用来引用 属性的后端域变量 ("属性的后端域变量(Backing Field)" in "属性(Property)").
- `it` 在 lambda 表达式内部, 用来 引用 lambda 表达式的隐含参数 ("it: 单一参数的隐含名称" in "高阶函数与 Lambda 表达式").

操作符与特殊符号

Kotlin 支持以下操作符与特殊符号:

- `+`, `-`, `*`, `/`, `%` - 算数运算符
  - `*` 也被用来 向一个不定数量参数传递数组 ("不定数量参数(varargs)" in "函数").
- `=`
  - 赋值操作符.
  - 用来指定 参数的默认值 ("默认参数" in "函数").
- `+=`, `-=`, `*=`, `/=`, `%=` - 计算并赋值 ("计算并赋值" in "操作符重载").
- `++`, `--` - 递增与递减操作符 ("递增与递减操作符" in "操作符重载").
- `&&`, `||`, `!` - '与', '或', '非' 逻辑运算符 (用于位运算, 使用对应的 中缀函数 ("数值类型的运算符(Operation)" in "数值类型")).
- `==`, `!=` - 相等和不等比较操作符 ("相等和不等比较操作符" in "操作符重载") (对非基本类型, 会翻译为对 `equals()` 函数的调用).

- `===`, `!==` - 引用相等比较操作符 ("引用相等" in "相等判断").
- `<`, `>`, `<=`, `>=` - 比较操作符 ("比较操作符" in "操作符重载") (对非基本类型, 会翻译为对 `compareTo()` 函数的调用).
- `[`, `]` - 下标访问操作符 ("下标访问操作符" in "操作符重载") (会翻译为对 `get` 和 `set` 函数的调用).
- `!!` 断言一个表达式的值不为 null ("!! 操作符" in "Null 值安全性").
- `?.` 执行一个 安全调用 ("安全调用" in "Null 值安全性") (如果接受者不为 null, 则调用一个方法, 或调用一个属性的访问函数).
- `?:` 如果这个运算符左侧的表达式值为 null, 则返回右侧的表达式值(也就是 elvis 操作符 ("Elvis 操作符" in "Null 值安全性")).
- `::` 创建一个 成员的引用 ("函数引用(Function Reference)" in "反射"), 或者一个 类引用 ("类引用(Class Reference)" in "反射").
- `..`, `..<` 创建 值范围 (值范围(Range)与数列(Progression)).
- `:` 在声明中, 用作名称与类型之间的分隔符.
- `?` 将一个类型标记为 可为 null ("可为 null 的类型与不可为 null 的类型" in "Null 值安全性").
- `->`
  - 在 lambda 表达式 ("Lambda 表达式的语法" in "高阶函数与 Lambda 表达式") 中, 用作参数与函数体之间的分隔符.
  - 在 函数类型 ("函数类型(Function Type)" in "高阶函数与 Lambda 表达式") 中, 用作参数与返回类型之间的分隔符.
  - 在 when 表达式 ("when 表达式" in "条件与循环") 的分支中, 用作分支条件与分支体之间的分隔符.
- `@`
  - 引入一个 注解 ("注解的使用" in "注解").
  - 定义, 或者引用一个 循环标签 ("Break 和 Continue 的位置标签" in "返回与跳转: break 与 continue").
  - 定义, 或者引用一个 lambda 表达式标签 ("使用标签控制 return 的目标" in "返回与跳转: break 与 continue").

    - 引用一个 外层范围的 'this' 表达式 ("带限定符的 this" in "this 表达式").
    - 引用一个 外部类的超类 ("调用超类中的实现" in "继承").
- `;` 用于在同一行中分隔多条语句.
- `$` 在 字符串模板 ("字符串模板" in "字符串") 中引用变量或表达式.
- `_`
    - 在 lambda 表达式 ("使用下划线代替未使用的参数" in "高阶函数与 Lambda 表达式") 中代替未使用的参数.
    - 在 解构声明 ("用下划线代替未使用的变量" in "解构声明") 中代替未使用的参数.

关于操作符优先顺序, 请参见 Kotlin 语法中的 这一章节 (https://kotlinlang.org/docs/reference/grammar.html#expressions).

Gradle

最终更新: 2024/09/10

Gradle 是一个构建系统, 帮助你自动化并管理你的构建过程. 它会下载需要的依赖项, 打包你的代码, 并做好编译前的准备工作. 关于 Gradle 的基本概念与详细信息, 请参见 Gradle 网站 (https://docs.gradle.org/current/userguide/userguide.html).

通过 这些使用说明 (配置 Gradle 项目), 你可以对不同的平台设置你自己的项目, 或者可以学习一个小的 step-by-step 教程 (Gradle 与 Kotlin/JVM 入门), 它会演示如何使用 Kotlin 创建一个简单的 "Hello World" 后台应用程序.

关于 Kotlin, Gradle, 和 Android Gradle plugin 各版本的兼容性, 详情请参见 这里 ("应用(Apply) Kotlin Gradle Plugin" in "配置 Gradle 项目").

在本章中, 你将会学习:

- 编译器选项, 以及如何传递编译器选项 (Kotlin Gradle plugin 中的编译器选项).
- 增量编译, 缓存, 构建报告, 以及 Kotlin Daemon (Kotlin Gradle plugin 中的编译与缓存).
- 对 Gradle plugin 变体(variant)的支持 (对 Gradle plugin 变体的支持).

下一步做什么?

学习:

- **Gradle Kotlin DSL**. Gradle Kotlin DSL (https://docs.gradle.org/current/userguide/kotlin_dsl.html) 是一种领域特定语言(Domain Specific Language), 你可以用它快速高效的编写构建脚本.
- **注解处理**. Kotlin 通过 Kotlin 符号处理 API (针对 Java 注解处理器开发者的参考文档) 支持注解处理.
- **生成文档**. 要为 Kotlin 项目生成文档, 请使用 Dokka (https://github.com/Kotlin/dokka); 关于它的配置方法, 请参照 Dokka README (https://github.com/Kotlin/dokka/blob/master/README.md#using-the-gradle-plugin). Dokka 支持混合语言项目, 可以生成多种格式的输出文档, 包括标准的 Javadoc.

- **OSGi**. 关于 OSGi 支持, 请参见 Kotlin OSGi 文档 (Kotlin 与 OSGi).

Gradle 与 Kotlin/JVM 入门

最终更新: 2024/09/10

本教程演示如何使用 IntelliJ IDEA 和 Gradle 来创建一个控制台应用程序.

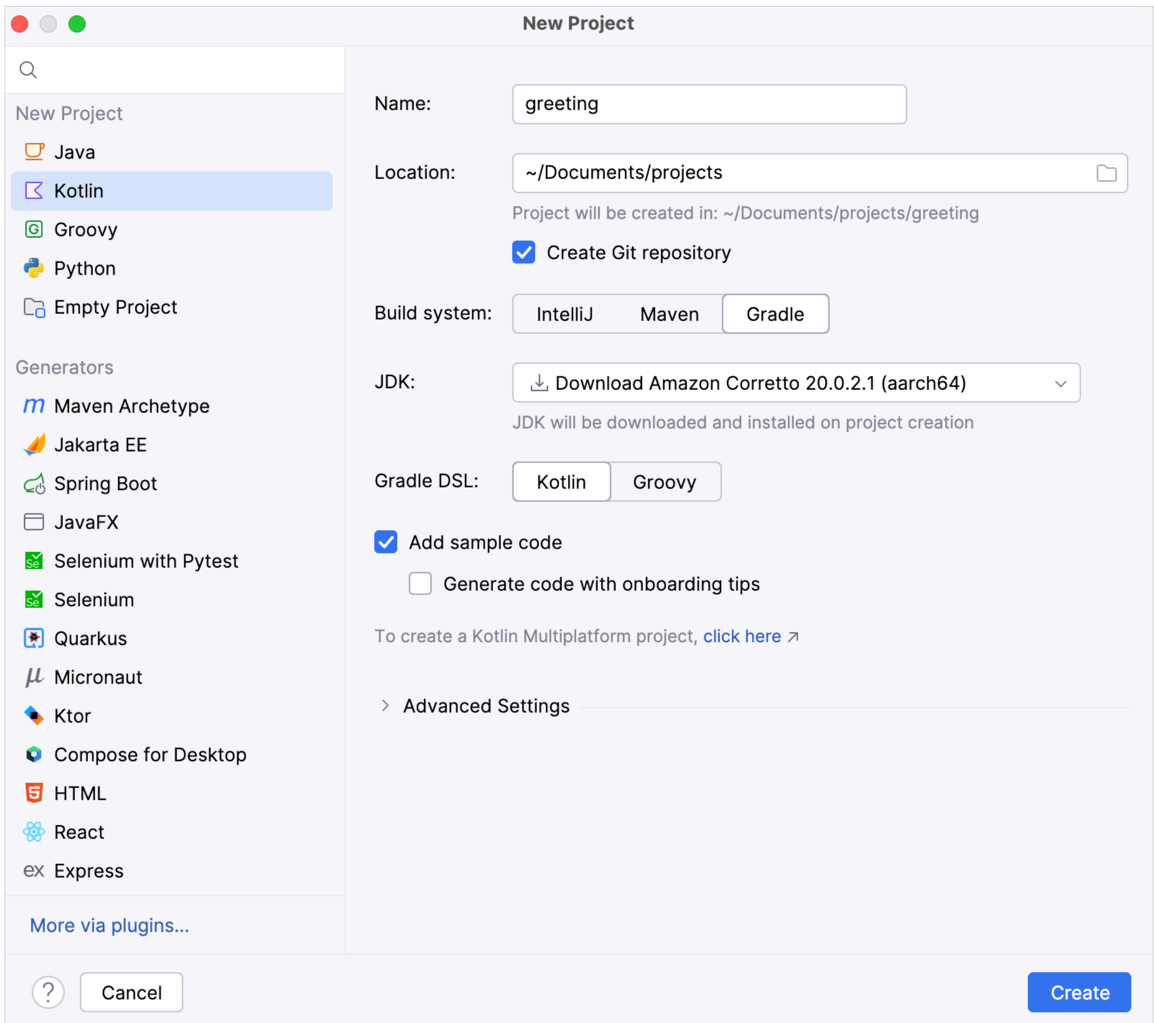
开始之前, 首先请下载病安装最新版本的 IntelliJ IDEA
(https://www.jetbrains.com/idea/download/index.html).

创建项目

1. 在 IntelliJ IDEA 中, 选择 **File** | **New** | **Project**.
2. 在左侧面板中, 选择 **New Project**.
3. 输入新项目的名称, 如果需要的话, 修改它的位置.

> 选择 **Create Git repository** 选项, 可以将新项目添加到版本管理系统. 你也可以在创建项目之后再做.

4. 在 **Language** 选项中, 选择 **Kotlin**.



创建一个控制台应用程序

5. 选择 **Gradle** 构建系统.

6. 在 **JDK list** 选项中, 选择你的项目希望使用的 JDK (https://www.oracle.com/java/technologies/downloads/).

   - 如果在你的计算机上已经安装了 JDK, 但在 IDE 中没有定义它, 请选择 **Add JDK**, 并指定 JDK home 目录的路径.
   - 如果在你的计算机上还没有安装需要的 JDK, 请选择 **Download JDK**.

7. 在 **Gradle DSL** 选项中, 选择 **Kotlin**.

8. 选中 **Add sample code** 选项, 创建一个文件, 其中包含一个 `"Hello World!"` 示例程序.

> ⚠ 你也可以启用 **Generate code with onboarding tips** 选项, 向你的示例代码添加一些

有用的注释.

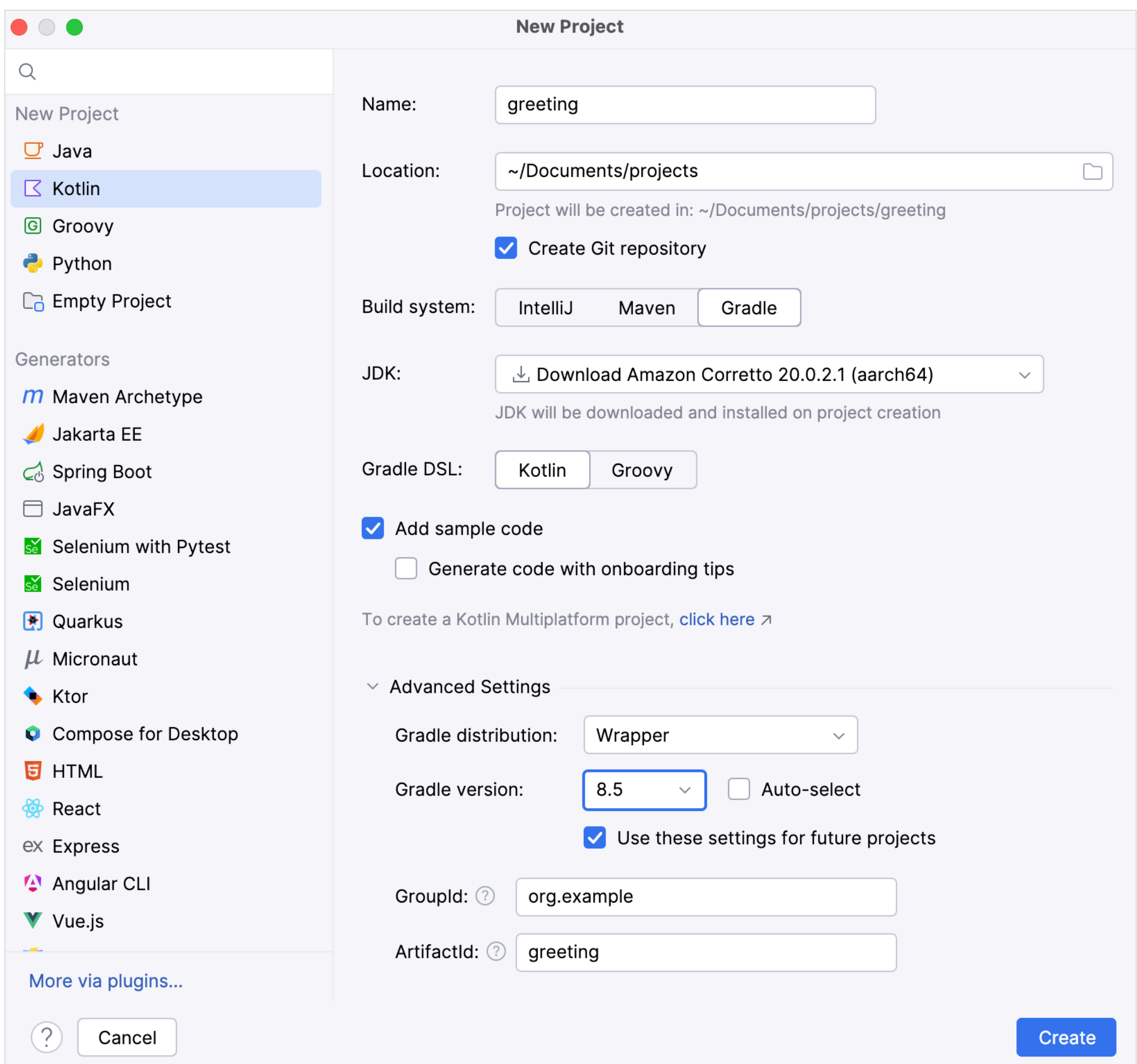
9. 点击 **Create.**

这样你就成功的创建了 Gradle 项目.

为你的项目指定 Gradle 版本

你可以在 **Advanced Settings** 中为你的项目明确指定 Gradle 版本, 可以使用 Gradle Wrapper, 也可以使用本地安装的 Gradle:

- **使用 Gradle Wrapper:** 在 **Gradle distribution** 选项列表中, 选择 **Wrapper**. 取消 **Auto-select** 选择框, 并在 **Gradle version** 选项列表中, 选择你的 Gradle 版本.
- **使用本地安装的 Gradle:** 在 **Gradle distribution** 选项列表中, 选择 **Local installation**. 对 **Gradle location**, 请指定你的本地 local Gradle 版本的路径.



高级设置

查看构建脚本

打开 build.gradle.kts 文件. 这是 Gradle 的 Kotlin 构建脚本, 包含 Kotlin 相关的 artifact 以及应用程序需要的其他部分:

```
// 下文中的 `KotlinCompile` task 需要这个 import
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    kotlin("jvm") version "1.9.23" // 使用的 Kotlin 版本
    application // Application plugin. 参见下文的 1️⃣
}
```

```
group = "org.example" // 公司名, 比如, `org.jetbrains`
version = "1.0-SNAPSHOT" // 构建后的 artifact 的版本

repositories { // 依赖项的下载源仓库. 参见 2
    mavenCentral() // Maven Central Repository. 参见 3
}

dependencies { // 你想要使用的所有库. 参见 4
    // 复制你在仓库中找到的依赖项名称
    testImplementation(kotlin("test")) // Kotlin test 库
}

tasks.test { // 参见 5
    useJUnitPlatform() // 用于测试的 JUnitPlatform. 参见 6
}

kotlin { // 扩展, 用于简化设置
    jvmToolchain(17) // 生成的 JVM bytecode 的目标版本. 参见 7
}

application {
    mainClass.set("MainKt") // 应用程序的 main class
}
```

- 1 Application plugin (https://docs.gradle.org/current/userguide/application_plugin.html) 支持构建 Java CLI 应用程序.
- 2 参见 依赖项的下载源仓库 (https://docs.gradle.org/current/userguide/declaring_repositories.html).
- 3 Maven Central Repository (https://central.sonatype.com/). 也可以使用 Google 的 Maven repository (https://maven.google.com/), 或你的公司的私有仓库.
- 4 参见 声明依赖项 (https://docs.gradle.org/current/userguide/declaring_dependencies.html).
- 5 参见 构建任务(Task) (https://docs.gradle.org/current/dsl/org.gradle.api.Task.html).

- 6️⃣ 用于测试的 JUnitPlatform (https://docs.gradle.org/current/javadoc/org/gradle/api/tasks/testing/Test.html#useJUnitPlatform).
- 7️⃣ 参见 设置 Java 工具链 ("Gradle Java 工具链支持" in "配置 Gradle 项目").

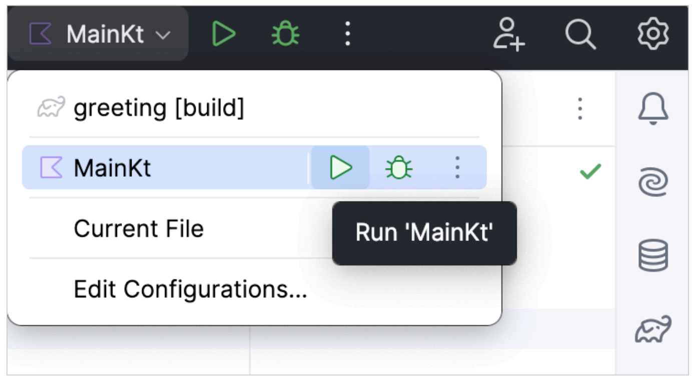
你可以看到, Gradle 构建文件中还添加了几个 Kotlin 相关的 artifact:

1. 在 `plugins{}` 代码段中, 有 `kotlin("jvm")` artifact – 这个 plugin 定义项目中使用的 Kotlin 版本.
2. 在 `dependencies` 内, 有 `testImplementation(kotlin("test"))`. 详情请参见 设置测试库的依赖项 ("设置对测试库的依赖项" in "配置 Gradle 项目").
3. 在依赖项之后, 有 `KotlinCompile` 任务的配置代码段. 在这里你可以向编译器添加额外的参数, 来开启或禁用 Kotlin 的各种功能特性.

运行应用程序

打开 `src/main/kotlin` 目录中的 `Main.kt` 文件 . `src` 目录包含 Kotlin 源代码文件和资源. `Main.kt` 文件包含示例代码, 打印输出 `Hello World!`.

运行应用程序的最简单的方法是, 点击编辑器侧栏中的绿色 **Run** 图标, 然后选择 **Run 'MainKt'**.



运行一个控制台应用程序

你可以在 **Run** 工具窗口看到结果.

程序运行的输出结果

恭喜! 你成功的运行了你的第一个 Kotlin 应用程序.

下一步做什么?

学习:

- Gradle 构建文件属性 (https://docs.gradle.org/current/dsl/org.gradle.api.Project.html#N14E9A).
- 针对不同的平台, 设置库的依赖项 (配置 Gradle 项目).
- 编译器选项, 以及如何传递编译器选项 (Kotlin Gradle plugin 中的编译器选项).
- 增量编译, 缓存, 构建报告, 以及 Kotlin Daemon (Kotlin Gradle plugin 中的编译与缓存).

配置 Gradle 项目

最终更新: 2024/09/10

要 Gradle (https://docs.gradle.org/current/userguide/userguide.html) 使用来构建 Kotlin 项目, 你需要向你的构建脚本文件 `build.gradle(.kts)` 添加 Kotlin Gradle plugin, 并在构建脚本文件中 配置项目的依赖项.

> 关于构建脚本, 更多内容请参见 查看构建脚本 ("查看构建脚本" in "Gradle 与 Kotlin/JVM 入门") 小节.

应用(Apply) Kotlin Gradle Plugin

要应用(Apply) Kotlin Gradle plugin, 请使用 Gradle plugin DSL 的 `plugins{}` 代码段 (https://docs.gradle.org/current/userguide/plugins.html#sec:plugins_block):

Kotlin

```
// 请将 `<...>` 替换为 plugin 名称
plugins {
    kotlin("<...>") version "1.9.23"
}
```

Groovy

```
// 请将 `<...>` 替换为 plugin 名称
plugins {
    id 'org.jetbrains.kotlin.<...>' version '1.9.23'
}
```

> Kotlin Gradle plugin (KGP) 和 Kotlin 的版本号一致.

配置你的项目时, 请检查 Kotlin Gradle plugin (KGP) 是否兼容于你的 Gradle 版本. 下表是, Kotlin **完全支持** 的 Gradle 和 Android Gradle plugin (AGP) 最低和最高版本:

KGP 版本	Gradle 最低和最高版本	AGP 最低和最高版本
1.9.20–1.9.23	6.8.3–8.1.1	4.2.2–7.4.0
1.9.0–1.9.10	6.8.3–7.6.0	4.2.2–7.4.0
1.8.20–1.8.22	6.8.3–7.6.0	4.1.3–7.4.0
1.8.0–1.8.11	6.8.3–7.3.3	4.1.3–7.2.1
1.7.20–1.7.22	6.7.1–7.1.1	3.6.4–7.0.4
1.7.0–1.7.10	6.7.1–7.0.2	3.4.3–7.0.2
1.6.20–1.6.21	6.1.1–7.0.2	3.4.3–7.0.2

你也可以使用最新版本之前的 Gradle 和 AGP 版本, 但如果你这样做, 请注意, 你可能会遇到弃用警告, 或者某些新功能可能无法正常工作.

例如, Kotlin Gradle plugin 和 `kotlin-multiplatform` plugin 1.9.23 最低需要 Gradle 版本 6.8.3 才能编译你的项目.

类似的, 完全支持的最高版本是 8.1.1. 这个版本不包含已废弃的 Gradle 方法和属性, 并且支持目前所有的 Gradle 功能特性.

编译到 JVM 平台

要编译到 JVM 平台, 需要应用 Kotlin JVM plugin.

Kotlin

```
plugins {
    kotlin("jvm") version "1.9.23"
```

```
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.9.23"
}
```

在这段代码中, `version` 必须是写明的字面值, 不能通过其他编译脚本得到.

Kotlin 源代码与 Java 源代码

Kotlin 源代码与 Java 源代码可以保存在相同的目录下, 也可以放在不同的目录下.

默认的约定是使用不同的目录:

```
project
    - src
        - main (root)
            - kotlin
            - java
```

> ⚠ 不要将 Java 的 `.java` 文件放在 `src/*/kotlin` 目录中, 因为这样的 `.java` 文件不会被编译.
>
> 你应该改为放在 `src/main/java` 目录中.

如果不使用默认约定的文件夹结构, 那么需要修改相应的 `sourceSets` 属性:

Kotlin

```
sourceSets.main {
    java.srcDirs("src/main/myJava", "src/main/myKotlin")
}
```

Groovy

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

对相关联的编译任务检查 JVM 编译目标的兼容性

在构建模块中, 你可能会有多个相互关联的编译任务, 比如:

- `compileKotlin` 与 `compileJava`
- `compileTestKotlin` 与 `compileTestJava`

> `main` 与 `test` 源代码集的编译任务之间没有关联.

对于这种相互关联的编译任务, Kotlin Gradle plugin 会检查 JVM 编译目标的兼容性. `kotlin` 扩展或任务中的 `jvmTarget` 属性 ("JVM 任务独有的属性" in "Kotlin Gradle plugin 中的编译器选项") 和 `java` 扩展或任务中的 `targetCompatibility` (https://docs.gradle.org/current/userguide/java_plugin.html#sec:java-extension) 如果设置为不同的值, 会导致 JVM 编译目标不兼容. 例如: `compileKotlin` 任务设置为 `jvmTarget=1.8`, 而 `compileJava` 任务设置为 (或 继承得到 (https://docs.gradle.org/current/userguide/java_plugin.html#sec:java-extension)) `targetCompatibility=15`.

要对整个项目的这个兼容性检查进行配置, 可以在 `build.gradle(.kts)` 文件中, 将 `kotlin.jvm.target.validation.mode` 属性设置为以下几个值:

- `error` – plugin 会让构建失败; 对于 Gradle 8.0 以上版本, 这是项目的默认值.
- `warning` – plugin 会输出警告信息; 对于低于 Gradle 8.0 的版本, 这是项目的默认值.
- `ignore` – plugin 会跳过检查, 不输出任何警告信息.

你也可以在你的 `build.gradle(.kts)` 文件中对各个编译任务单独进行配置:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompil
e>().configureEach {

jvmTargetValidationMode.set(org.jetbrains.kotlin.gradle.dsl.jvm.
JvmTargetValidationMode.WARNING)
}
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompil
e.class).configureEach {
    jvmTargetValidationMode =
org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidationMode.WARN
ING
}
```

要避免 JVM 编译目标不兼容, 需要 配置工具链, 或手动对齐(Align) JVM 版本.

如果编译目标之间不兼容, 会发生什么问题

有两种方式对 Kotlin 和 Java 源代码集手动设置 JVM 编译目标:

- 隐含设定, 通过 设置 Java 工具链 来设置.
- 明确设定, 通过设置 `kotlin` 扩展或任务中的 `jvmTarget` 属性, 以及`java` 扩展或任务中的 `targetCompatibility`.

如果你做以下设置, 就会发生 JVM 编译目标不兼容:

- 对 `jvmTarget` 和 `targetCompatibility` 明确设置不同的版本.
- 使用默认配置, 但你的 JDK 不等于 `1.8`.

如果在你的构建脚本中只有 Kotlin JVM plugin, 并且没有额外设置 JVM 编译目标, 我们来看看这时的默认 JVM 编译目标设置:

Kotlin

```
plugins {
    kotlin("jvm") version "1.9.23"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.9.23"
}
```

构建脚本中没有 jvmTarget 值的明确信息, 因此它的默认值为 null, 编译器将这个设置翻译为默认值 1.8. targetCompatibility 等于当前的 Gradle JDK 版本, 也就是你的 JDK 版本 (除非你使用 Java 工具链策略 ("Gradle Java 工具链支持" in "配置 Gradle 项目")). 假设你的 JDK 版本是 17, 你发布的库文件会 声明它兼容 (https://docs.gradle.org/current/userguide/publishing_gradle_module_metadata.html) 于 JDK 17 以上版本: org.gradle.jvm.version=17, 实际上是错误的. 这种情况下, 在你的主项目中, 会需要使用 Java 17 才能添加这个库, 尽管它的字节码版本其实是 1.8. 请 配置工具链 ("Gradle Java 工具链支持" in "配置 Gradle 项目") 来解决这个问题.

Gradle Java 工具链支持

> ⚠ 给 Android 使用者的警告. 要使用 Gradle 工具链支持, 需要使用 Android Gradle plugin (AGP) 的 8.1.0-alpha09 或更高版本.
>
> Gradle Java 工具链支持只在 AGP 7.4.0 以上版本 可用 (https://issuetracker.google.com/issues/194113162). 但是, 由于 这个问题 (https://issuetracker.google.com/issues/260059413), AGP 8.1.0-alpha09 以前的版本没有将 targetCompatibility 设置为等于工具链的 JDK. 如果你在使用低于 8.1.0-alpha09 的版本, 你需要通过 compileOptions 来手动配置 targetCompatibility. 请将占位符 <MAJOR_JDK_VERSION> 替换为你想要使用的 JDK 版本:
>
> ```
> android {
>     compileOptions {
>         sourceCompatibility = <MAJOR_JDK_VERSION>
>         targetCompatibility = <MAJOR_JDK_VERSION>
> ```

```
    }
}
```

Gradle 6.7 引入了 Java 工具链支持
(https://docs.gradle.org/current/userguide/toolchains.html). 通过这个功能, 你可以:

- 使用与 Gradle 不同的 JDK 和 JRE 来运行编译, 测试, 以及可执行程序.
- 使用还未发布的语言版本编译和测试代码.

通过工具链支持, Gradle 能够自动查找本地的 JDK, 还能安装 Gradle 运行构建时需要的 JDK. 目前 Gradle 自身能够在任何 JDK 上运行, 而且还对依赖于主要 JDK 版本的任务重用 远程构建缓存功能 ("对 Gradle 构建缓存的支持" in "Kotlin Gradle plugin 中的编译与缓存").

Kotlin Gradle plugin 对 Kotlin/JVM 编译任务支持 Java 工具链. JS 和 Native 任务则不会使用工具链. Kotlin 编译器永远会在运行 Gradle daemon 的 JDK 上运行. Java 工具链会:

- 为 JVM 编译目标设置 `-jdk-home` 选项 ("-jdk-home path" in "Kotlin 编译器选项").
- 如果用户没有明确设置 `jvmTarget` 选项, 则将 `compilerOptions.jvmTarget` ("JVM 任务独有的属性" in "Kotlin Gradle plugin 中的编译器选项") 设置为工具链的 JDK 版本. 如果用户没有配置工具链, 那么 `jvmTarget` 会使用默认值. 详情请参见 JVM 编译目标兼容性.
- 设置由任何 Java compile, test 和 javadoc 任务使用的工具链.
- 影响 `kapt` 任务执行器 ("并行运行多个 KAPT 任务" in "kapt 编译器插件") 使用哪个 JDK.

可以使用以下代码来设置工具链. 请将占位符 `<MAJOR_JDK_VERSION>` 替换为你想要使用的 JDK 版本:

Kotlin

```
kotlin {
    jvmToolchain {

languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
    // 或者使用更简短的写法:
    jvmToolchain(<MAJOR_JDK_VERSION>)
```

```
    // 例如:
    jvmToolchain(17)
}
```

Groovy

```
kotlin {
    jvmToolchain {

languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
    // 或者使用更简短的写法:
    jvmToolchain(<MAJOR_JDK_VERSION>)
    // 例如:
    jvmToolchain(17)
}
```

注意, 如果使用 `kotlin` 扩展设置工具链, 也会改变 Java 编译任务的工具链.

你可以通过 `java` 扩展设置工具链, Kotlin 编译任务会使用这个设置:

Kotlin

```
java {
    toolchain {

languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
}
```

Groovy

```
java {
    toolchain {

languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
```

```
    }
}
```

如果你使用 Gradle 8.0.2 或更高版本, 你还需要添加一个 工具链解析器 plugin (https://docs.gradle.org/current/userguide/toolchains.html#sub:download_repositories). 这种 plugin 会管理从哪个仓库下载工具链. 例如, 向你的 `settings.gradle(.kts)` 文件添加以下 plugin:

Kotlin

```
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention")
version("0.5.0")
}
```

Groovy

```
plugins {
    id 'org.gradle.toolchains.foojay-resolver-convention'
version '0.5.0'
}
```

关于与你的 Gradle 版本对应的 `foojay-resolver-convention` 版本, 请参见 Gradle 网站 (https://docs.gradle.org/current/userguide/toolchains.html#sub:download_repositories).

> 要确认 Gradle 使用哪个工具链, 请使用 log 级别 `--info` (https://docs.gradle.org/current/userguide/logging.html#sec:choosing_a_log_level) 来运行你的 Gradle 构建, 并在输出中查找 `[KOTLIN] Kotlin compilation 'jdkHome' argument:` 开头的字符串. 冒号之后的部分就是工具链使用的 JDK 版本.

要为特定的 Task 设置任意的 JDK (甚至本地 JDK), 请使用 Task DSL.

详情请参见 Kotlin plugin 中对 Gradle JVM 工具链的支持 (https://blog.jetbrains.com/kotlin/2021/11/gradle-jvm-toolchain-support-in-the-kotlin-plugin/).

使用 Task DSL 设置 JDK 版本

Task DSL 可以对任何实现了 `UsesKotlinJavaToolchain` 接口的任务, 设置任意的 JDK 版本. 目前, 这些任务只有 `KotlinCompile` 和 `KaptTask`. 如果希望 Gradle 搜索主要的 JDK 版本, 请在你的构建脚本中替换 `<MAJOR_JDK_VERSION>` 占位符:

Kotlin

```
val service = project.extensions.getByType<JavaToolchainService>
()
val customLauncher = service.launcherFor {

languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
}
project.tasks.withType<UsesKotlinJavaToolchain>().configureEach
{
    kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

Groovy

```
JavaToolchainService service =
project.getExtensions().getByType(JavaToolchainService.class)
Provider<JavaLauncher> customLauncher = service.launcherFor {

it.languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION
>))
}
tasks.withType(UsesKotlinJavaToolchain::class).configureEach {
task ->
    task.kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

或者你特也可以指定你的本地 JDK 路径, 然后使用这个 JDK 版本替换 `<LOCAL_JDK_VERSION>` 占位符:

```
tasks.withType<UsesKotlinJavaToolchain>().configureEach {
    kotlinJavaToolchain.jdk.use(
        "/path/to/local/jdk", // 这里设置你的 JDK 路径
        JavaVersion.<LOCAL_JDK_VERSION> // 例如, JavaVersion.17
    )
}
```

关联编译器任务

你可以将编译任务 *关联(Associate)* 在一起, 方法是在编译任务之间设置关联关系, 一个编译需要使用另一个编译的输出. 关联编译器任务会在编译任务之间建立 `internal` 的可见度.

Kotlin 编译器会默认的关联某些编译任务, 比如每个编译目标的 `test` 和 `main` 编译任务. 如果你需要表达你的某个自定义编译任务与其它编译任务相关联, 请创建你自己的编译任务关联.

要让 IDE 支持关联编译任务, 在源代码集之间推断可见度, 请向你的 `build.gradle(.kts)` 添加以下代码:

Kotlin

```
val integrationTestCompilation =
kotlin.target.compilations.create("integrationTest") {
    associateWith(kotlin.target.compilations.getByName("main"))
}
```

Groovy

```
integrationTestCompilation {
    kotlin.target.compilations.create("integrationTest") {

associateWith(kotlin.target.compilations.getByName("main"))
    }
}
```

在这个例子中, `integrationTest` 编译任务关联到 `main` 编译任务, 可以在功能测试(集成测试)代码中访问 `internal` 对象.

Java Modules (JPMS) 启用时的配置

要让 Kotlin Gradle plugin 与 Java 模块(Module) (https://www.oracle.com/corporate/features/understanding-java-9-modules.html) 共通工作, 请向你的构建脚本添加以下内容, 并将其中的 `YOUR_MODULE_NAME` 替换为你的 JPMS 模块的引用, 例如, `org.company.module`:

Kotlin

```
// 如果你使用的 Gradle 版本低于 7.0, 请添加以下 3 行
java {
    modularity.inferModulePath.set(true)
}

tasks.named("compileJava", JavaCompile::class.java) {

options.compilerArgumentProviders.add(CommandLineArgumentProvide
r {
        // 将编译后的 Kotlin 类提供给 to javac – 需要这样做才能让
Java/Kotlin 混合源代码正常工作
        listOf("--patch-module",
"YOUR_MODULE_NAME=${sourceSets["main"].output.asPath}")
    })
}
```

Groovy

```
// 如果你使用的 Gradle 版本低于 7.0, 请添加以下 3 行
java {
    modularity.inferModulePath = true
}

tasks.named("compileJava", JavaCompile.class) {
    options.compilerArgumentProviders.add(new
CommandLineArgumentProvider() {
        @Override
        Iterable<String> asArguments() {
```

```
            // Provide compiled Kotlin classes to javac – 需要这样做才能让 Java/Kotlin 混合源代码正常工作
            return ["--patch-module", "YOUR_MODULE_NAME=${sourceSets["main"].output.asPath}"]
        }
    })
}
```

> 和通常一样, 请将 module-info.java 文件放在 src/main/java 目录内.
>
> 对于模块, Kotlin 文件中的包名称应该等于 module-info.java 中的包名称, 否则会出现构建错误 "package is empty or does not exist".

更多详情请参见:

- 为 Java 模块系统构建模块 (https://docs.gradle.org/current/userguide/java_library_plugin.html#sec:java_library_modular)
- 使用 Java 模块系统构建应用程序 (https://docs.gradle.org/current/userguide/application_plugin.html#sec:application_modular)
- "module" 在 Kotlin 中的意义 ("模块(Module)" in "可见度修饰符")

其他细节

详情请参见 Kotlin/JVM (Kotlin/JVM 入门).

Kotlin/JVM 编译任务的延迟创建

从 Kotlin 1.8.20 开始, Kotlin Gradle plugin 在试运行(dry run)时会注册所有的编译任务, 但不对它们进行配置.

如果编译任务的输出目录不是默认位置

如果你覆盖了 Kotlin/JVM KotlinJvmCompile/KotlinCompile 编译任务的 destinationDirectory 位置, 请更新你的构建脚本. 在你的 JAR 文件中, 除 sourceSets.main.outputs 之外, 你需要明确添加 sourceSets.main.kotlin.classesDirectories:

```
tasks.jar(type: Jar) {
    from sourceSets.main.outputs
    from sourceSets.main.kotlin.classesDirectories
}
```

编译到多个目标平台

编译到 多个目标平台 ("编译目标" in "跨平台程序的 Gradle DSL 参考文档") 的项目, 称为 跨平台项目 (Kotlin 跨平台程序开发入门), 需要使用 `kotlin-multiplatform` 插件.

> `kotlin-multiplatform` 插件要求 Gradle 6.8.3 或更高版本.

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}
```

详情请参见 在不同的平台使用 Kotlin Multiplatform (Kotlin 跨平台程序开发入门) 和 在 iOS 和 Android 平台使用 Kotlin Multiplatform (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html).

编译到 Android 平台

建议使用 Android Studio 来创建 Android 应用程序. 详情请参见 如何使用 Android Gradle plugin (https://developer.android.com/studio/releases/gradle-plugin).

编译到 JavaScript

如果编译目标平台为 JavaScript, 也可以使用 kotlin-multiplatform 插件. 详情请阅读 如何设置 Kotlin/JS 项目 (创建 Kotlin/JS 工程(Project)):

Kotlin

```
plugins {
    kotlin("multiplatform") version "1.9.23"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.9.23'
}
```

JavaScript 项目的 Kotlin 源代码与 Java 源代码

这个 plugin 只能编译 Kotlin 源代码文件, 因此推荐将 Kotlin 和 Java 源代码文件放在不同的文件夹内(如果工程内包含 Java 文件的话). 如果不将源代码分开存放, 请在 sourceSets{} 代码段中指定源代码文件夹:

Kotlin

```
kotlin {
    sourceSets["main"].apply {
        kotlin.srcDir("src/main/myKotlin")
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        main.kotlin.srcDirs += 'src/main/myKotlin'
```

```
    }
}
```

使用 KotlinBasePlugin 接口触发配置动作

当任何 Kotlin Gradle plugin (JVM, JS, Multiplatform, Native, 等等) 被适用时, 要触发某些配置动作, 可以使用 `KotlinBasePlugin` 接口, 所有的 Kotlin plugin 都继承了这个接口:

Kotlin

```
import org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin

// ...

project.plugins.withType<KotlinBasePlugin>() {
    // 在这里配置你的动作
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin

// ...

project.plugins.withType(KotlinBasePlugin.class) {
    // 在这里配置你的动作
}
```

配置依赖项

如果要添加一个库的依赖, 需要在 source set DSL 中的 `dependencies{}` 代码段内, 设置必要 类型 的依赖项 (比如, `implementation`).

Kotlin

```kotlin
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("com.example:my-library:1.0")
            }
        }
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'com.example:my-library:1.0'
            }
        }
    }
}
```

或者, 你也可以 在最顶层设置依赖项.

依赖项的类型

请根据你的需要选择依赖项的类型.

类型	解释	使用场景
api	编译期和运行期都会使用, 并导出给库的使用者.	如果在当前模块的公开 API 中使用了一个依赖项中的任何类型, 请使用 api 依赖项.
implementation	对当前模块的编译期和运行期都会使用, 如果其他模块使用 `implementation` 依赖本模块, 那么对于其他模块的编译, 这个依赖项不会导出	对于模块的内部逻辑所需要的依赖项, 请使用这种类型. 如果一个模块是一个终端应用程序(endpoint application), 而且不对外公布(publish), 那么请使用 implementation 依赖项而不是 api 依赖项.
compileOnly	只用来编译当前模块, 在运行期不可用, 在编译其他模块时也不可用.	如果 API 在运行时存在第三方的实现, 那么可以使用这种依赖项.
runtimeOnly	运行时可用, 但在任何模块的编译期都不可用.	

对标准库的依赖项

对每个源代码集(Source Set), 会自动添加对标准库 (stdlib) 的依赖项. 使用的标准库版本与 Kotlin Gradle plugin 版本相同.

对于与平台相关的源代码集, 会使用针对这个平台的标准库, 同时, 对其他源代码集会添加共通的标准库. Kotlin Gradle plugin 会根据你的 Gradle 构建脚本的 compilerOptions.jvmTarget 编译器选项 (Kotlin Gradle plugin 中的编译器选项) 设置, 选择适当的 JVM 标准库.

如果明确的声明一个标准库依赖项(比如, 如果你需要使用不同的版本), Kotlin Gradle plugin 不会覆盖你的设置, 也不会添加第二个标准库.

如果你完全不需要标准库, 可以在你的 gradle.properties 文件中添加以下 Gradle 属性:

```
kotlin.stdlib.default.dependency=false
```

传递依赖项的版本对齐

从 Kotlin 标准库 1.9.20 版开始, Gradle 使用包含在标准库中的元数据(metadata), 来自动对齐传递依赖项 kotlin-stdlib-jdk7 和 kotlin-stdlib-jdk8 的版本.

如果你添加了 Kotlin 标准库版本 1.8.0 到 1.9.10 之间的依赖项, 例如: implementation("org.jetbrains.kotlin:kotlin-stdlib:1.8.0"), 那么 Kotlin Gradle Plugin 会对传递依赖项 kotlin-stdlib-jdk7 和 kotlin-stdlib-jdk8 使用这个 Kotlin 版本. 这样会避免标准库的不同版本出现重复的类. 详情请参见 kotlin-stdlib-jdk7 与 kotlin-stdlib-jdk8 合并到 kotlin-stdlib ("更新了 JVM 编译目标" in "Kotlin 1.8.0 版中的新功能"). 你可以在你的 gradle.properties 文件中使用 Gradle 属性 kotlin.stdlib.jdk.variants.version.alignment 来禁用这个动作:

```
kotlin.stdlib.jdk.variants.version.alignment=false
```

版本对齐的另一种方法

- 如果版本对齐出现了问题, 你可以使用 Kotlin BOM (https://docs.gradle.org/current/userguide/platforms.html#sub:bom_import) 来对齐所有依赖项的版本. 在你的构建脚本中声明对 kotlin-bom 的平台依赖项:

  Kotlin

  ```
  implementation(platform("org.jetbrains.kotlin:kotlin-
  bom:1.9.23"))
  ```

  Groovy

  ```
  implementation platform('org.jetbrains.kotlin:kotlin-
  bom:1.9.23')
  ```

- 如果你没有添加某个版本的标准库的依赖项, 但你有两个不同的依赖项, 分别带来 Kotlin 标准库不同旧版本的传递依赖, 那么你可以对这些传递依赖的库明确指定 1.9.23 版本:

  Kotlin

  ```
  dependencies {
      constraints {
          add("implementation", "org.jetbrains.kotlin:kotlin-
  ```

```
stdlib-jdk7") {
            version {
                require("1.9.23")
            }
        }
        add("implementation", "org.jetbrains.kotlin:kotlin-
stdlib-jdk8") {
            version {
                require("1.9.23")
            }
        }
    }
}
```

Groovy

```
dependencies {
    constraints {
        add("implementation", "org.jetbrains.kotlin:kotlin-
stdlib-jdk7") {
            version {
                require("1.9.23")
            }
        }
        add("implementation", "org.jetbrains.kotlin:kotlin-
stdlib-jdk8") {
            version {
                require("1.9.23")
            }
        }
    }
}
```

- 如果你添加了 Kotlin 标准库 `1.9.23` 的依赖项: `implementation("org.jetbrains.kotlin:kotlin-stdlib:1.9.23")`, 并且使用了旧版本的 (低于 `1.8.0`) Kotlin Gradle plugin, 请更新 Kotlin Gradle plugin, 保持与标准库的版本一致:

Kotlin

```kotlin
plugins {
    // 请将 `<...>` 替换为 plugin 名称
    kotlin("<...>") version "1.9.23"
}
```

Groovy

```groovy
plugins {
    // 请将 `<...>` 替换为 plugin 名称
    id "org.jetbrains.kotlin.<...>" version "1.9.23"
}
```

- 如果你使用旧版本 (低于 `1.8.0`) 的 `kotlin-stdlib-jdk7`/`kotlin-stdlib-jdk8`, 例如, `implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk7:SOME_OLD_KOTLIN_VERSION")`, 并且某个依赖项传递依赖到 `kotlin-stdlib:1.8+`, 请将你的 `kotlin-stdlib-jdk<7/8>:SOME_OLD_KOTLIN_VERSION` 替换为 `kotlin-stdlib-jdk*:1.9.23` ("更新了 JVM 编译目标" in "Kotlin 1.8.0 版中的新功能"), 或者在传递依赖它的库中 排除(exclude) (https://docs.gradle.org/current/userguide/dependency_downgrade_and_exclude.html#sec:excluding-transitive-deps) `kotlin-stdlib:1.8+`:

Kotlin

```kotlin
dependencies {
    implementation("com.example:lib:1.0") {
        exclude(group = "org.jetbrains.kotlin", module =
"kotlin-stdlib")
    }
}
```

Groovy

```
dependencies {
    implementation("com.example:lib:1.0") {
        exclude group: "org.jetbrains.kotlin", module:
"kotlin-stdlib"
    }
}
```

设置对测试库的依赖项

对于支持的所有平台, Kotlin 项目的测试可以使用 `kotlin.test` (https://kotlinlang.org/api/latest/kotlin.test/) API. 对 `commonTest` 源代码集添加 `kotlin-test` 依赖项, 然后 Gradle plugin 会为每个测试源代码集推断出对应的测试库依赖项:

- 对共通源代码集, 会添加 `kotlin-test-common` 和 `kotlin-test-annotations-common` 依赖项
- 对 JVM 源代码集, 会添加 `kotlin-test-junit` 依赖项
- 对 Kotlin/JS 源代码集, 会添加 `kotlin-test-js` 依赖项

Kotlin/Native 编译目标已经内建了 `kotlin.test` API 的实现, 不需要额外的测试依赖项.

Kotlin

```
kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // 这个设置会自动引
入对应平台的所有依赖项
            }
        }
    }
}
```

Groovy

```kotlin
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // 这个设置会自动引入对应平台的所有依赖项
            }
        }
    }
}
```

对 Kotlin 模块的依赖项, 可以使用简写, 比如, 对 "org.jetbrains.kotlin:kotlin-test" 的依赖项可以简写为 kotlin("test").

你也可以在任何共通源代码集或平台相关的源代码集中使用 `kotlin-test` 依赖项.

对于 Kotlin/JVM, Gradle 默认使用 JUnit 4. 因此, `kotlin("test")` 依赖项会解析为 JUnit 4 的变体, 名为 `kotlin-test-junit`.

也可以选择使用 JUnit 5 或 TestNG, 方法是在构建脚本的测试任务中调用 `useJUnitPlatform()` (https://docs.gradle.org/current/javadoc/org/gradle/api/tasks/testing/Test.html#useJUnitPlatform) 或 `useTestNG()` (https://docs.gradle.org/current/javadoc/org/gradle/api/tasks/testing/Test.html#useTestNG). 下面是一个 Kotlin Multiplatform 项目的示例:

Kotlin

```kotlin
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
```

```
            }
        }
    }
}
```

Groovy

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test")
            }
        }
    }
}
```

下面是一个 JVM 项目的示例:

Kotlin

```
dependencies {
    testImplementation(kotlin("test"))
}

tasks {
    test {
        useTestNG()
    }
}
```

Groovy

```
dependencies {
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}

test {
    useTestNG()
}
```

参见 在 JVM 平台上如何使用 JUnit 测试代码 (教程 - 在 JVM 平台使用 JUnit 进行代码测试).

如果需要使用不同的 JVM 测试框架, 可以在项目的 `gradle.properties` 文件添加 `kotlin.test.infer.jvm.variant=false`, 关闭测试框架的自动选择. 然后, 再将需要的测试框架添加为 Gradle 依赖项.

如果你在构建脚本中明确使用了 `kotlin("test")` 的变体, 而且项目的构建脚本出现兼容性冲突问题, 不再正常工作, 请参见 兼容性指南中的这个问题 (Kotlin 1.5 兼容性指南).

设置对 kotlinx 库的依赖项

如果使用 `kotlinx` 库 (https://github.com/Kotlin/kotlinx.coroutines), 并且需要与平台相关的依赖项, 那么可以通过 `-jvm` 或 `-js` 之类的后缀, 来指定与平台相关的库版本, 例如, `kotlinx-coroutines-core-jvm`. 也可以使用库的基本 artifact 名(base artifact name) — `kotlinx-coroutines-core`.

Kotlin

```
kotlin {
    sourceSets {
        val jvmMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core-jvm:1.7.3")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        jvmMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-
coroutines-core-jvm:1.7.3'
            }
        }
    }
}
```

如果使用跨平台的库, 并且需要依赖共用代码, 那么只需要在共用源代码集中一次性设置依赖项. 请使用库的基本 artifact 名(base artifact name), 例如 `kotlinx-coroutines-core` 或 `ktor-client-core`.

Kotlin

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3")
            }
        }
    }
}
```

Groovy

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
```

```
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3'
            }
        }
    }
}
```

在最顶层设置依赖项

另一种做法是, 可以在最顶层指定依赖项, 方法是使用 `<sourceSetName><DependencyType>` 格式的配置名称. 对于某些 Gradle 内建的依赖项, 比如 `gradleApi()`, `localGroovy()`, 或 `gradleTestKit()`, 这种方法会很有用, 这些依赖项在 Source Set 依赖项 DSL 中是不能使用的.

Kotlin

```
dependencies {
    "commonMainImplementation"("com.example:my-library:1.0")
}
```

Groovy

```
dependencies {
    commonMainImplementation 'com.example:my-library:1.0'
}
```

声明仓库

你可以声明一个可公开访问的仓库, 使用它的 open source 依赖项. 请在 `repositories{}` 代码段中, 设置仓库的名称:

Kotlin

```
repositories {
    mavenCentral()
```

```
}
```

Groovy

```
repositories {
    mavenCentral()
}
```

常用的仓库是 Maven Central (https://central.sonatype.com/) 和 Google's Maven repository (https://maven.google.com/web/index.html).

> ⚠ 如果你同时也在使用 Maven 项目, 我们建议不要将 `mavenLocal()` 添加为仓库, 因为在 Gradle 和 Maven 项目间切换时, 你可能遇到问题. 如果你一定需要添加 `mavenLocal()` 仓库, 请在你的 `repositories{}` 代码段中, 将它添加为最后一个仓库. 更多详情请参见 使用 mavenLocal() 的情况 (https://docs.gradle.org/current/userguide/declaring_repositories.html#sec:case-for-maven-local).

如果你需要在多个子项目中声明相同的仓库, 请在你的 `settings.gradle(.kts)` 文件中, 在 `dependencyResolutionManagement{}` 代码段中集中声明仓库:

Kotlin

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

Groovy

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
```

```
    }
}
```

在子项目中声明的任何仓库, 都会覆盖集中声明的仓库. 关于如何控制这种行为, 有什么解决办法, 详情请参见 Gradle 的文档 (https://docs.gradle.org/current/userguide/declaring_repositories.html#sub:centralized-repository-declaration).

下一步做什么?

学习:

- 编译器选项, 以及如何传递编译器选项 (Kotlin Gradle plugin 中的编译器选项).
- 增量编译, 缓存, 构建报告, 以及 Kotlin Daemon (Kotlin Gradle plugin 中的编译与缓存).
- Gradle 基本概念与详细信息 (https://docs.gradle.org/current/userguide/userguide.html).
- 对 Gradle plugin 变体的支持 (对 Gradle plugin 变体的支持).

Kotlin Gradle plugin 中的编译器选项

最终更新: 2024/09/10

Kotlin 的每一个发布版本都包含它所支持的各个编译目标的编译器: JVM, JavaScript, 以及 支持的平台的 ("目标平台" in "使用 Kotlin/Native 进行原生(Native)程序开发") 原生二进制文件.

这些编译器会在以下情况下使用:

- 当你对你的 Kotlin 工程按下 **Compile** 或 **Run** 按钮时, 由 IDE 使用.
- 当你在控制台或在 IDE 内调用 `gradle build` 命令时, 由 Gradle 使用.
- 当你在控制台或在 IDE 内调用 `mvn compile` 或 `mvn test-compile`, 由 Maven 使用.

你也可以从命令行手动运行 Kotlin 编译器, 详情请参见教程 使用命令行编译器 (Kotlin 命令行编译器).

如何定义编译器选项

Kotlin 编译器带有很多选项, 用来定制编译过程.

使用构建脚本, 你可以指定额外的编译选项. 可以通过 Kotlin 编译任务的 `compilerOptions` 属性来添加编译选项. 例如:

Kotlin

```
tasks.named("compileKotlin",
org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask::class.java) {
    compilerOptions {
        freeCompilerArgs.add("-Xexport-kdoc")
    }
}
```

Groovy

```
tasks.named('compileKotlin',
org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class) {
    compilerOptions {
        freeCompilerArgs.add("-Xexport-kdoc")
    }
}
```

JVM 目标平台

对于 JVM 目标平台, 编译产品代码的编译任务名为 `compileKotlin`, 编译测试代码的编译任务名为 `compileTestKotlin`. 针对自定义源代码集的编译任务名, 是与源代码集名称对应的 `compile<Name>Kotlin`.

Android 项目的编译任务名称, 包含 构建变体(build variant) (https://developer.android.com/studio/build/build-variants.html) 的名称, 完整名称是 `compile<BuildVariant>Kotlin`, 比如, `compileDebugKotlin`, `compileReleaseUnitTestKotlin`.

对于 JVM 和 Android 项目, 可以使用项目的 Kotlin 扩展 DSL 来定义选项:

Kotlin

```
kotlin {
    compilerOptions {

apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOT
LIN_2_0)
    }
}
```

Groovy

```
kotlin {
    compilerOptions {
        apiVersion =
org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_0
```

```
    }
}
```

有一些重要的细节需要注意:

- `android.kotlinOptions` 和 `kotlin.compilerOptions` 配置代码块会相互覆盖. 只有最后出现的 (最下方的) 代码块会起作用.
- `kotlin.compilerOptions` 会配置项目中所有的 Kotlin 编译任务.
- 你可以使用 `tasks.named<KotlinJvmCompile>("compileKotlin") { }` (或 `tasks.withType<KotlinJvmCompile>().configureEach { }`) 来由覆盖 `kotlin.compilerOptions` DSL 提供的配置.

JavaScript 目标平台

对于 JavaScript 目标平台, 产品代码的编译任务名是 `compileKotlinJs`, 测试代码的编译任务名是 `compileTestKotlinJs`, 针对自定义源代码集的编译任务名, 是 `compile<Name>KotlinJs`.

要对单个编译任务进行配置, 请使用它的名称. 示例如下:

Kotlin

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

val compileKotlin: KotlinCompilationTask<*> by tasks

compileKotlin.compilerOptions.suppressWarnings.set(true)
```

Groovy

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        suppressWarnings.set(true)
```

```
    }
}
```

注意, 使用 Gradle Kotlin DSL 时, 你应该先从编译工程的 `tasks` 属性得到编译任务.

编译 JavaScript 和 Common 时, 请使用相应的 `Kotlin2JsCompile` 和 `KotlinCompileCommon` 类型.

配置所有的 Kotlin 编译任务

也可以对项目中的所有 Kotlin 编译任务进行配置:

Kotlin

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure
{
    compilerOptions { /*...*/ }
}
```

Groovy

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions { /*...*/ }
}
```

所有的编译器选项

Gradle 任务所支持的选项完整列表如下:

共通属性

属性名称	描述	可以选择的值	默认值
optIn	配置 opt-in 编译器参数 (明确要求使用者同意的功能(Opt-in Requirement)) 列表	listOf(/* opt-ins */)	emptyList()
progressiveMode	启用 渐进编译模式 ("渐进模式" in "Kotlin 1.3 版中的新功能")	true, false	false

JVM 任务独有的属性

属性名称	描述	可以选择的值	默认值
javaParameters	为 Java 1.8 的方法参数反射功能生成 metadata		false
jvmTarget	指定编译输出的 JVM 字节码的版本	"1.8", "9", "10", ..., "20", "21". 参见 编译器选项的数据类型	"1.8"
noJdk	不要自动将 Java 运行库包含到 classpath 内		false
jvmTargetValidationMode	验证 Kotlin 和 Java 编译任务的 JVM 编译目标兼容性 ("对相关联的编译任务检查 JVM 编译目标的兼容性" in "配置 Gradle 项目"). 适用于 KotlinCompile 类型的任务.	WARNING, ERROR, INFO	ERROR

JVM, JS, 和 JS DCE 任务支持的共通属性

属性名称	描述	可以选择的值	默认值
`allWarningsAsErrors`	把警告作为错误来处理		false
`suppressWarnings`	不产生警告信息		false
`verbose`	输出详细的 log 信息. 只在 Gradle debug log 级别启用 (https://docs.gradle.org/current/userguide/logging.html) 时有效		false
`freeCompilerArgs`	指定额外的编译参数, 可以是多个. 这里也可以使用实验性的 `-X` 参数. 参见 示例		[]

> ⚠ 在未来的发布版中, 我们将会废弃 `freeCompilerArgs` 属性. 如果你希望恢复 Kotlin Gradle DSL 中的某些选项, 请在 Youtrack 中 提出问题 (https://youtrack.jetbrains.com/newissue?project=kt).

通过 freeCompilerArgs 使用额外参数的示例

可以使用 `freeCompilerArgs` 属性来指定额外的 (包括实验性的) 编译器参数. 你可以对这个属性添加单个参数, 也可以是多个参数的列表:

Kotlin

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

val compileKotlin: KotlinCompilationTask<*> by tasks

// 单个实验性参数
compileKotlin.compilerOptions.freeCompilerArgs.add("-Xexport-
kdoc")
```

```
// 单个额外参数，可以是 key-value 对
compileKotlin.compilerOptions.freeCompilerArgs.add("-Xno-param-assertions")
// 多个参数的列表
compileKotlin.compilerOptions.freeCompilerArgs.addAll(listOf("-Xno-receiver-assertions", "-Xno-call-assertions"))
```

Groovy

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        // 单个实验性参数
        freeCompilerArgs.add("-Xexport-kdoc")
        // 单个额外参数，可以是 key-value 对
        freeCompilerArgs.add("-Xno-param-assertions")
        // 多个参数的列表
        freeCompilerArgs.addAll(["-Xno-receiver-assertions", "-Xno-call-assertions"])
    }
}
```

JVM 和 JS 任务支持的共通属性

属性名称	描述	可以选择的值	默认值
`apiVersion`	只允许使用指定的版本的运行库中的 API	"1.4" (已废弃 DEPRECATED), "1.5" (已废弃 DEPRECATED), "1.6", "1.7", "1.8", "1.9", "2.0" (实验性功能), "2.1" (实验性功能)	
`languageVersion`	指定源代码所兼容的 Kotlin 版本	"1.4" (已废弃 DEPRECATED), "1.5" (已废弃 DEPRECATED), "1.6", "1.7", "1.8", "1.9", "2.0" (实验性功能), "2.1" (实验性功能)	

languageVersion 设置示例

要设置语言版本, 请使用下面的语法:

Kotlin

```
tasks

.withType<org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_0
            )
    }
```

Groovy

```
tasks

.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTas
k.class)
```

```
    .configureEach {
        compilerOptions.languageVersion =

org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_0
    }
```

参见 编译器选项的数据类型.

JS 任务独有的属性

属性名称	描述	可以选择的值	默认值
`friendModulesDisabled`	指定是否关闭内部声明的输出		false
`main`	指定执行时是否调用 main 函数	"call", "noCall". 参见 编译器选项的数据类型	"call"
`metaInfo`	指定是否生成带有 metadata 的 .meta.js 和 .kjsm 文件. 用于创建库		true
`moduleKind`	指定编译器生成的 JS 模块类型	"umd", "commonjs", "amd", "plain", "es". 参见 编译器选项的数据类型	"umd"
`outputFile`	指定编译结果输出的 *.js 文件		"\<buildDir>/js/packages/\<project.name>/kotlin/\<project.name>.js"
`sourceMap`	指定是否生成源代码映射文件(source map)		true
`sourceMapEmbedSources`	指定是否将源代码文件嵌入到源代码映射文件中	"never", "always", "inlining". 参见 编译器选项的数据类型	

sourceMapNamesPolicy	将你在 Kotlin 代码中声明的变量和函数名称添加到源代码映射文件中. 详情请参见 编译器参考文档 ("-source-map-names-policy {simple-names|fully-qualified-names|no}" in "Kotlin 编译器选项").	"simple-names", "fully-qualified-names", "no". 参见 编译器选项的数据类型	"simple-names"
sourceMapPrefix	对源代码映射文件中的路径添加一个指定的前缀		
target	指定生成的 JS 文件 的 ECMA 版本	"v5"	"v5"
typedArrays	将基本类型数组转换为 JS 的有类型数组		true

编译器选项的数据类型

有些 `compilerOptions` 使用新的数据类型, 而不是旧的 `String` 类型:

选项	数据类型	示例
jvmTarget	JvmTarget (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JvmTarget.kt)	compilerOptions.jvmTarget.set(JvmTarget.JVM_11)
apiVersion and languageVersion	KotlinVersion (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/KotlinVersion.kt)	compilerOptions.languageVersion.set(KotlinVersion.KOTLIN_2_0)
main	JsMainFunctionExecutionMode (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JsMainFunctionExecutionMode.kt)	compilerOptions.main.set(JsMainFunctionExecutionMode.NO_CALL)
moduleKind	JsModuleKind (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JsModuleKind.kt)	compilerOptions.moduleKind.set(JsModuleKind.MODULE_ES)
sourceMapEmbedSources	JsSourceMapEmbedMode (https://github.com/JetBrains/kotlin/blob/1.8.0/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JsSourceMapEmbedMode.kt)	compilerOptions.sourceMapEmbedSources.set(JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CONTENT_INLINING)
sourceMapNamesPolicy	JsSourceMapNamesPolicy (https://github.com/JetBrains/kotlin/blob/1.8.20/libraries/tools/kotlin-gradle-compiler-types/src/generated/kotlin/org/jetbrains/kotlin/gradle/dsl/JsSourceMapNamesPolicy.kt)	compilerOptions.sourceMapNamesPolicy.set(JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_FQ_NAMES)

下一步做什么?

学习:

- 增量编译, 缓存, 构建报告, 以及 Kotlin Daemon (Kotlin Gradle plugin 中的编译与缓存).
- Gradle 的基本概念与详细信息 (https://docs.gradle.org/current/userguide/userguide.html).
- 对 Gradle plugin 变体的支持 (对 Gradle plugin 变体的支持).

Kotlin Gradle plugin 中的编译与缓存

最终更新: 2024/09/10

在本章中, 你将学习以下内容:

- 增量编译(Incremental compilation)
- 对 Gradle 构建缓存的支持
- 对 Gradle 配置缓存的支持
- Kotlin daemon 及其在 Gradle 中的使用
- 定义 Kotlin 编译器执行策略
- Kotlin 编译器的 fallback 策略
- 构建报告

增量编译(Incremental compilation)

Kotlin Gradle plugin 支持增量编译模式. 增量编译模式会监视源代码文件在两次编译之间的变更, 因此只有变更过的文件会被编译.

增量编译模式支持 Kotlin/JVM 和 Kotlin/JS 工程, 并且默认开启.

有以下几种方式可以禁用增量编译设定:

- 对 Kotlin/JVM 项目: 设置 `kotlin.incremental=false`.
- 对 Kotlin/JS 项目: 设置 `kotlin.incremental.js=false`.
- 在命令行参数中, 添加 `-Pkotlin.incremental=false` 或 `-Pkotlin.incremental.js=false`.

  需要向所有后续的编译命令都添加这个参数.

注意: 任何一次编译如果关闭了增量编译模式, 都会导致增量编译的缓存失效. 初次编译不会是增量编译.

有时增量编译的问题会在错误发生之后再经过多轮才报告给使用者. 请使用 构建报告 来

追踪变更历史和编译历史. 这样可以帮助你提供可重现的 bug 报告.

增量编译的新方案

从 Kotlin 1.7.0 开始, 针对 JVM 后端, 而且只在 Gradle 构建系统中, 可以使用增量编译的新方案. 从 Kotlin 1.8.20 开始, 默认启用这个新方案. 这种方案支持发生在依赖的非 Kotlin 模块内的变更, 包括编译回避功能的改进, 并且与 Gradle 构建缓存 兼容.

这些功能改进可以减少非增量式构建的次数, 让整体的编译时间更加快速. 如果你使用构建缓存, 或者在非 Kotlin Gradle 模块中频繁进行修改, 那么可以得到显著的性能改进.

要关闭这个新方案, 请在你的 `gradle.properties` 中设置以下选项:

```
kotlin.incremental.useClasspathSnapshot=false
```

希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-49682) 提供你对这个功能的反馈意见.

关于增量编译的新方案的底层实现细节, 请参见 这篇 Blog (https://blog.jetbrains.com/kotlin/2022/07/a-new-approach-to-incremental-compilation-in-kotlin/).

对编译任务的输出的精确备份

对编译任务的输出的精确备份是 实验性功能 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 希望你能通过我们的 问题追踪系统 (https://kotl.in/issue/experimental-ic-optimizations) 提供你的反馈意见.

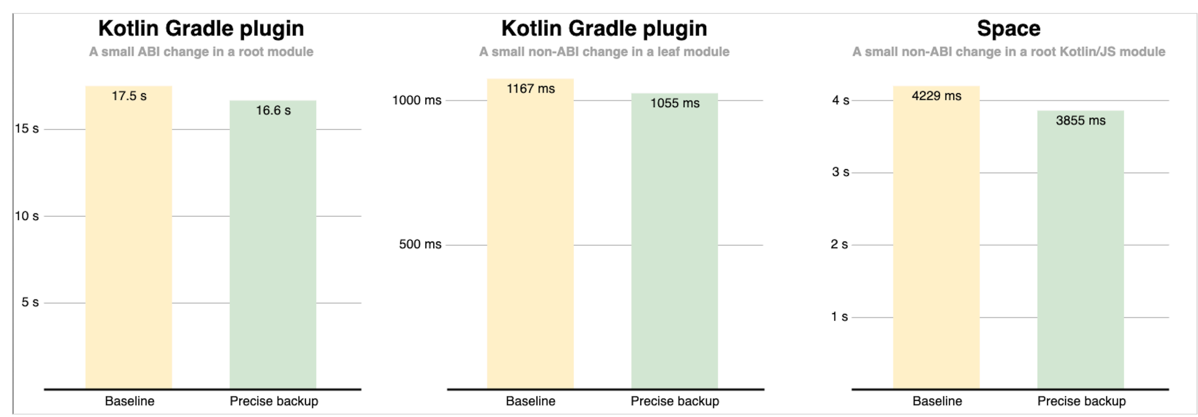
从 Kotlin 1.8.20 开始, 你可以启用精确备份功能, 这时只有 Kotlin 在增量编译中重新编译的那些类会被备份. 完整备份和精确备份都可以帮助在发生编译错误后再次运行增量构建. 精确备份与完整备份相比, 会耗费较少的构建时间. 对于大型的项目, 或者很多任务都创建备份, 那么完整备份可能会花费 **明显** 更长的构建时间, 尤其是如果项目位于速度较慢的 HDD 上.

要启用这个优化功能, 请向 `gradle.properties` 文件添加 `kotlin.compiler.preciseCompilationResultsBackup` Gradle 属性:

```
kotlin.compiler.preciseCompilationResultsBackup=true
```

JetBrains 使用精确备份的例子

在下面的图表中, 你可以看到使用精确备份与完整备份相对比的示例:



完整备份与精确备份的对比

第一个和第二个对比图显示了在 Kotlin 项目中使用精确备份时对 Kotlin Gradle plugin 构建的影响:

1. 进行一个小的 ABI (https://en.wikipedia.org/wiki/Application_binary_interface) 变更之后: 向一个被大量模块依赖的模块添加一个新的 public 方法.
2. 进行一个小的非 ABI 变更之后: 向一个没有被其他模块依赖的模块添加一个 private 函数.

第三个对比图显示了在 Space (https://www.jetbrains.com/space/) 项目中使用精确备份时, 在小的非 ABI 更改后对 Web 前端构建的影响: 向一个被大量模块依赖的 Kotlin/JS 模块添加一个 private 函数.

我们在使用 Apple M1 Max CPU 的计算机上进行这些测量; 在不同的计算机上会出现稍微不同的结果. 影响性能的因素包括但不限于以下几点:

- Kotlin daemon 和 Gradle daemon (https://docs.gradle.org/current/userguide/gradle_daemon.html) 热身状况(warm)如何.
- 硬盘速度如何.
- CPU 型号, 以及它的繁忙程度.
- 哪些模块受到变更的影响, 以及这些模块有多大.
- 是 ABI 变更还是非 ABI 变更.

使用构建报告来评估优化

要对你的项目和场景, 评估优化在你的计算机上的影响, 你可以使用 Kotlin 构建报告. 请向你的 `gradle.properties` 文件添加下面的属性, 启用文本文件格式的构建报告:

```
kotlin.build.report.output=file
```

下面是在启用精确备份*之前*, 构建报告的相关部分的示例:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.59 s
<...>
Time metrics:
 Total Gradle task time: 0.59 s
 Task action before worker execution: 0.24 s
  Backup output: 0.22 s // 注意这个数字
<...>
```

下面是在启用精确备份*之后*, 构建报告的相关部分的示例:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.46 s
<...>
Time metrics:
 Total Gradle task time: 0.46 s
 Task action before worker execution: 0.07 s
  Backup output: 0.05 s // 备份消耗的时间减少了
 Run compilation in Gradle worker: 0.32 s
  Clear jar cache: 0.00 s
  Precise backup output: 0.00 s // 与精确备份相关的输出
  Cleaning up the backup stash: 0.00 s // 与精确备份相关的输出
<...>
```

对 Gradle 构建缓存的支持

Kotlin 插件支持 Gradle 构建缓存 (https://docs.gradle.org/current/userguide/build_cache.html), 构建缓存会保存构建的输出, 并在未来的构建中重复使用.

如果想要对所有的 Kotlin 任务禁用缓存, 请将系统属性 `kotlin.caching.enabled` 设置为 `false` (也就是使用参数 `-Dkotlin.caching.enabled=false` 来执行编译).

对 Gradle 配置缓存的支持

Kotlin plugin 使用 Gradle 配置缓存 (https://docs.gradle.org/current/userguide/configuration_cache.html), 通过对之后的构建重用配置阶段的结果, 来增加构建处理的速度.

关于如何启用配置缓存, 请参见 Gradle 文档 (https://docs.gradle.org/current/userguide/configuration_cache.html#config_cache:usage). 启用这个功能之后, Kotlin Gradle plugin 会自动开始使用它.

Kotlin daemon 及其在 Gradle 中的使用

Kotlin daemon 会:

- 与 Gradle daemon 共同运行来编译项目.
- 当你使用 IntelliJ IDEA 内建的构建系统来编译项目时, 它会在 Gradle daemon 之外单独运行.

在 Gradle 的 执行阶段 (https://docs.gradle.org/current/userguide/build_lifecycle.html#sec:build_phases), 当一个 Kotlin 编译任务开始编译源代码时, Kotlin daemon 就会启动. Kotlin daemon 会和 Gradle daemon 一起停止, 或在没有 Kotlin 编译任务执行, 空闲 2 个小时之后停止.

Kotlin daemon 使用与 Gradle daemon 相同的 JDK.

设置 Kotlin daemon 的 JVM 参数

以下列表是设置参数的几种不同方式, 列表中设置方式按照优先级排列, 每种方式都会覆盖在它之前的其他方式:

- 继承 Gradle daemon 参数
- 设置系统属性 `kotlin.daemon.jvm.options`
- 设置属性 `kotlin.daemon.jvmargs`
- 使用 `kotlin` 扩展
- 使用特定的任务定义

继承 Gradle daemon 参数

如果不做任何设定, Kotlin daemon 会从 Gradle daemon 继承 JVM 参数. 比如, 在 `gradle.properties` 文件中:

```
org.gradle.jvmargs=-Xmx1500m -Xms=500m
```

设置系统属性 kotlin.daemon.jvm.options

如果 Gradle daemon 的 JVM 参数包含 `kotlin.daemon.jvm.options` 系统属性 – 请在 `gradle.properties` 文件中指定:

```
org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m,Xms=500m
```

传递参数时, 要遵守以下规则:

- **只有** 在参数 `Xmx`, `XX:MaxMetaspaceSize`, 和 `XX:ReservedCodeCacheSize` 之前要使用减号 `-`, 在其它参数之前不要使用.
- 参数之间的分隔使用逗号 (`,`), *不带空格*. 空格之后的参数会被 Gradle daemon 使用, 而不是被 Kotlin daemon 使用.

> 如果满足以下所有条件, Gradle 会忽略这些属性:
>
> - Gradle 使用 JDK 1.9 或更高版本.
> - Gradle 版本在 7.0(含) 和 7.1.1(含) 之间.
> - Gradle 正在编译 Kotlin DSL 脚本.
> - Kotlin daemon 没有运行.
>
> 要解决这个问题, 请升级 Gradle 到 7.2 (或更高版本), 或者使用 `kotlin.daemon.jvmargs` 属性 – 参见以下章节.

设置属性 kotlin.daemon.jvmargs

你可以在 `gradle.properties` 文件中添加 `kotlin.daemon.jvmargs` 属性:

```
kotlin.daemon.jvmargs=-Xmx1500m -Xms=500m
```

使用 kotlin 扩展

你可以在 `kotlin` 扩展中指定参数:

Kotlin

```
kotlin {
    kotlinDaemonJvmArgs = listOf("-Xmx486m", "-Xms256m", "-
XX:+UseParallelGC")
}
```

Groovy

```
kotlin {
    kotlinDaemonJvmArgs = ["-Xmx486m", "-Xms256m", "-
XX:+UseParallelGC"]
}
```

使用特定的任务定义

你可以对特定的任务指定参数:

Kotlin

```
tasks.withType<CompileUsingKotlinDaemon>().configureEach {
    kotlinDaemonJvmArguments.set(listOf("-Xmx486m", "-Xms256m",
"-XX:+UseParallelGC"))
}
```

Groovy

```
tasks.withType(CompileUsingKotlinDaemon::class).configureEach {
task ->
    task.kotlinDaemonJvmArguments.set(["-Xmx1g", "-Xms512m"])
}
```

> 这种情况下, 会在任务执行时启动一个新的 Kotlin daemon 实例. 更多详情请参见 指定 JVM 参数时 Kotlin daemon 的行为.

指定 JVM 参数时 Kotlin daemon 的行为

配置 Kotlin daemon 的 JVM 参数时, 请注意:

- 如果不同的子项目或任务设置了不同的 JVM 参数, 那么会存在多个 Kotlin daemon 实例同时运行.
- 只有当 Gradle 运行相关的编译任务, 而且现存的 Kotlin daemon 实例没有使用相同的 JVM 参数时, 才会启动新的 Kotlin daemon 实例. 假设你的项目包含很多子项目. 大部分子项目对 Kotlin daemon 需要某种 heap memory 设定, 但有一个模块需要很大的 heap memory 设定 (然而这个模块很少被编译). 这种情况下, 你应该对这个模块设定不同的 JVM 参数, 这样, 就可以只有在开发者编译这个特定模块时, 才会使用很大的 heap memory 启动一个 Kotlin daemon.

  > 如果已有某个 Kotlin daemon 在运行中, 并且它的 heap memory 尺寸足够满足编译的需求, 那么即使另一个任务要求的 JVM 参数不同, 也仍会重用这个 daemon, 而不是启动一个新的实例.

- 如果 `Xmx` 参数未指定, Kotlin daemon 会从 Gradle daemon 继承.

Kotlin 的新编译器

Kotlin 的新 K2 编译器处于 Beta 阶段 ("稳定性级别" in "Kotlin 各部分组件的稳定性"). 它对 Kotlin JVM, Native, Wasm 和 JS 项目提供基本的支持.

新编译器的目标是加速新的语言功能的开发, 统一 Kotlin 支持的所有平台, 带来性能改进, 并为编译器扩展提供 API.

从 Kotlin 2.0 开始, 将会默认使用 K2 编译器. 要在你的项目中试用它, 并检查它的性能, 请使用 `kotlin.experimental.tryK2=true` Gradle 属性, 或执行下面的命令:

```
./gradlew assemble -Pkotlin.experimental.tryK2=true
```

这个 Gradle 属性会自动将默认语言版本设置为 2.0, 并更新 构建报告 其中包含, 与当前的编译器相比, 使用 K2 编译器编译的 Kotlin 任务的数量.

关于 K2 编译器的稳定性, 更多详情请参见我们的 Kotlin blog

(https://blog.jetbrains.com/kotlin/2023/02/k2-kotlin-2-0/)

定义 Kotlin 编译器执行策略

Kotlin 编译器执行策略 定义 Kotlin 编译器在哪里执行, 以及各种情况下是否支持增量编译.

有 3 种编译器执行策略:

策略	Kotlin 编译器在哪里执行	增量编译	其它特征, 以及注意事项
Daemon	在 Kotlin 自己的 daemon 进程之内	是	*默认的, 而且最快的策略*. 可以在不同的 Gradle daemon, 以及多个并行编译之间共用.
In process	在 Gradle daemon 进程之内	否	可以与 Gradle daemon 共用 heap. "In process" 执行策略比 "Daemon" 执行策略 *更慢*. 每个 worker (https://docs.gradle.org/current/userguide/worker_api.html) 会为每个编译创建单独的 Kotlin 编译器 classloader.
Out of process	对每个编译都在单独的进程内	否	这是最慢的执行策略. 与 "In process" 类似, 但还会为每个编译在 Gradle worker 内创建单独的 Java 进程.

要定义一个 Kotlin 编译器执行策略, 你可以使用以下属性之一:

- Gradle 属性 `kotlin.compiler.execution.strategy`.
- Compile Task 属性 `compilerExecutionStrategy`.

Task 属性 `compilerExecutionStrategy` 的优先级高于 Gradle 属性 `kotlin.compiler.execution.strategy`.

`kotlin.compiler.execution.strategy` 属性可以使用的值是:

1. `daemon` (默认值)

2. `in-process`

3. `out-of-process`

在 `gradle.properties` 中使用 Gradle 属性 `kotlin.compiler.execution.strategy`:

```
kotlin.compiler.execution.strategy=out-of-process
```

Task 属性 `compilerExecutionStrategy` 可以使用的值是:

1. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.DAEMON` (默认值)

2. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.IN_PROCESS`

3. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.OUT_OF_PROCESS`

在你的构建脚本中使用 Task 属性 `compilerExecutionStrategy`:

Kotlin

```
import
org.jetbrains.kotlin.gradle.tasks.CompileUsingKotlinDaemon
import
org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrateg
y

// ...

tasks.withType<CompileUsingKotlinDaemon>().configureEach {

compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN
_PROCESS)
}
```

Groovy

```
import
org.jetbrains.kotlin.gradle.tasks.CompileUsingKotlinDaemon
```

```
import
org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrateg
y

// ...

tasks.withType(CompileUsingKotlinDaemon)
    .configureEach {

compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN
_PROCESS)
    }
```

Kotlin 编译器的 fallback 策略

Kotlin 编译器的 fallback 策略是指, 如果 daemon 因为某种原因失败, 则在 Kotlin daemon 之外运行编译任务. 如果 Gradle daemon 启动, 编译器会使用 "In process" 策略. 如果 Gradle daemon 没有启动, 编译器会使用 "Out of process" 策略.

当 fallback 发生时, 在你的 Gradle 构建输出中会收到以下警告信息:

```
Failed to compile with Kotlin daemon: java.lang.RuntimeException:
Could not connect to Kotlin compile daemon
[exception stacktrace]
Using fallback strategy: Compile without Kotlin daemon
Try ./gradlew --stop if this issue persists.
```

但是, 静默的 fallback 到其他策略, 会消耗大量的系统资源, 或导致不确定的构建结果. 关于这个问题, 请参见这个 YouTrack issue (https://youtrack.jetbrains.com/issue/KT-48843/Add-ability-to-disable-Kotlin-daemon-fallback-strategy). 要避免这个问题, 有一个 Gradle 属性 `kotlin.daemon.useFallbackStrategy`, 默认值为 `true`. 当它的值设置为 `false` 时, daemon 启动或通信时问题会导致构建失败. 请在 `gradle.properties` 文件中声明这个属性:

```
kotlin.daemon.useFallbackStrategy=false
```

在 Kotlin 编译任务中也有一个 `useDaemonFallbackStrategy` 属性, 如果同时使用, 它的优先级会高于 Gradle 属性.

Kotlin

```
tasks {
    compileKotlin {
        useDaemonFallbackStrategy.set(false)
    }
}
```

Groovy

```
tasks.named("compileKotlin").configure {
    useDaemonFallbackStrategy = false
}
```

如果运行编译所需要的内存不足, 你会在 log 中看到相关信息.

构建报告

构建报告是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 需要使用者同意(Opt-in) (详情见下文). 请注意, 只为评估目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issues/KT) 提供你的反馈意见.

构建报告包括不同编译阶段的持续时间, 以及为什么不能进行增量编译的原因. 如果编译时间太长, 或对于相同的项目出现了不同的编译时间, 可以使用构建报告来调查性能问题.

Kotlin 构建报告可以帮助你调查构建性能相关的问题, 它比 Gradle build scans (https://scans.gradle.com/) 更加有效, Gradle Build Scan 中的粒度只是单个 Gradle Task.

通过对长时间运行的编译分析构建报告, 可以帮助你解决两种常见问题:

- 构建不能增量模式运行. 分析原因, 并解决底层问题.
- 构建是增量模式运行, 但耗费太多时间. 可以尝试重新组织源代码文件 — 切分大的文件, 将不同的类保存到不同的文件, 重构大的类, 在不同的文件中声明顶层函数, 等等.

构建报告还会显示项目中使用的 Kotlin 版本. 此外, 从 Kotlin 1.9.0 开始, 你可以在你的 Gradle Build Scan (https://scans.gradle.com/) 中看到, 是使用当前编译器还是 K2 编译器来编译代码.

请参见 如何阅读构建报告 (https://blog.jetbrains.com/kotlin/2022/06/introducing-kotlin-build-reports/#how_to_read_build_reports) 以及 JetBrains 如何使用构建报告 (https://blog.jetbrains.com/kotlin/2022/06/introducing-kotlin-build-reports/#how_we_use_build_reports_in_jetbrains).

启用构建报告

要启用构建报告, 请在 `gradle.properties` 中指定构建报告输出的保存位置:

```
kotlin.build.report.output=file
```

以下各个值的组合可以用于输出:

选项	含义
`file`	将构建报告保存到本地文件, 使用可供人类阅读的格式. 默认设置是 `${project_folder}/build/reports/kotlin-build/${project_name}-timestamp.txt`
`single_file`	将构建报告保存到本地文件, 使用二进制对象格式
`build_scan`	将构建报告保存到 build scan (https://scans.gradle.com/) 的 `custom values` 小节. 注意, Gradle Enterprise plugin 会限制 custom values 的数量和长度. 在很大的项目中, 有些值可能会丢失.
`http`	通过 HTTP(S) 提交构建报告. 使用 POST 方法传送 JSON 格式的测量结果. 你可以在 Kotlin 代码仓库 (https://github.com/JetBrains/kotlin/blob/master/libraries/tools/kotlin-gradle-plugin/src/common/kotlin/org/jetbrains/kotlin/gradle/plugin/statistics/CompileStatisticsData.kt) 中看到传送的数据的当前版本. 你可以在 这篇 Blog (https://blog.jetbrains.com/kotlin/2022/06/introducing-kotlin-build-reports/#enable_build_reports) 看到 HTTP Endpoint 的示例

下面是 `kotlin.build.report` 的选项列表:

```
# 需要的报告输出格式. 可以任意组合
kotlin.build.report.output=file,single_file,http,build_scan

# 如果使用 single_file 输出, 则必须设置. 表示报告的输出位置
# 请使用这个设定, 代替已废弃的
`kotlin.internal.single.build.metrics.file` 属性
kotlin.build.report.single_file=some_filename

# 可选项. 文件格式的报告的输出目录. 默认值是: build/reports/kotlin-build/
kotlin.build.report.file.output_dir=kotlin-reports

# 可选项. 用来标记你的构建报告的标签 (例如, debug parameters)
kotlin.build.report.label=some_label
```

只适用于 HTTP 输出的选项:

```
# 必须设置. 基于 HTTP(S) 的报告的 POST 地址
kotlin.build.report.http.url=http://127.0.0.1:8080

# 可选项. 如果 HTTP endpoint 要求身份验证, 通过这个设置指定用户名和密码
kotlin.build.report.http.user=someUser
kotlin.build.report.http.password=somePassword

# 可选项. 将构建的 Git branch 名称添加到构建报告
kotlin.build.report.http.include_git_branch.name=true|false

# 可选项. 将编译器参数添加到构建报告
# 如果一个项目包含很多模块, 构建报告中的编译器参数可能非常重, 而且并没有多大帮助
kotlin.build.report.include_compiler_arguments=true|false
```

custom values 的限制

为了收集 build scan 的统计信息, Kotlin 构建报告会使用 Gradle 的 custom values (https://docs.gradle.com/enterprise/tutorials/extending-build-scans/). 你和各种 Gradle plugin 都可以向 custom value 写入数据. custom value 的数量存在限制. 关于 custom value 目前的最大件数, 请参见 Build scan plugin 文档 (https://docs.gradle.com/enterprise/gradle-plugin/#adding_custom_values).

如果你的项目很大, 这些 custom value 的数量可能非常大. 如果超过的限制, 你会在 log 中看到以下信息:

```
Maximum number of custom values (1,000) exceeded
```

要减少 Kotlin plugin 产生的 custom value 数量, 你可以在 `gradle.properties` 文件中使用以下属性:

```
kotlin.build.report.build_scan.custom_values_limit=500
```

关闭对项目和系统属性的收集

HTTP 构建统计 log 可能包含某些项目和系统属性. 这些属性可以改变构建的行为, 因此将它们输出到构建统计信息中会很有用处. 这是属性也可能存储了敏感信息, 例如, 密码, 或项目的完整路径.

你可以向你的 `gradle.properties` 文件添加 `kotlin.build.report.http.verbose_environment` 属性, 来禁止收集这些统计信息.

> JetBrains 不会收集这些统计信息. 你需要选择一个地方来 存储你的统计报告.

下一步做什么?

学习:

- Gradle 的基本概念与详细信息 (https://docs.gradle.org/current/userguide/userguide.html).
- 对 Gradle plugin 变体的支持 (对 Gradle plugin 变体的支持).

对 Gradle plugin 变体的支持

最终更新: 2024/09/10

Gradle 7.0 为 Gradle plugin 开发者引入了一个新功能 — 带变体(variant)的 plugin (https://docs.gradle.org/7.0/userguide/implementing_gradle_plugins.html#plugin-with-variants). 使用这个功能, 可以在 plugin 中支持最新的 Gradle 功能, 同时保持与旧版本 Gradle 的兼容性. 详情请参见 Gradle 中的变体选择 (https://docs.gradle.org/current/userguide/variant_model.html).

使用 Gradle plugin 变体, Kotlin 开发组能够针对不同的 Gradle 版本发布不同的 Kotlin Gradle plugin (KGP) 变体. 目标是在 `main` 变体中支持基本的 Kotlin 编译, 这个变体对应于支持的 Gradle 最低版本. 每个变体将会支持对应的 Gradle 版本的功能. 最新的变体将会支持最新的 Gradle 功能. 通过这种方式, 可以对旧的 Gradle 版本支持较少的功能, 对新的 Gradle 版本支持更多的功能.

目前, Kotlin Gradle plugin 有以下变体:

变体名	对应的 Gradle 版本
`main`	6.8.3–6.9.3
`gradle70`	7.0
`gradle71`	7.1-7.4
`gradle75`	7.5
`gradle76`	7.6
`gradle80`	8.0
`gradle81`	8.1.1 或更高版本

在未来的 Kotlin 发布版中, 还会增加更多的变体.

要检查你的构建使用的是哪个变体, 请启用 `--info` log 级别 (https://docs.gradle.org/current/userguide/logging.html#sec:choosing_a_log_level), 然后在

日志输出中查找以 `Using Kotlin Gradle plugin` 开头的字符串, 例如, `Using Kotlin Gradle plugin main variant`.

问题与解决方案

关于 Gradle 中的变体选择功能, 下面是一些已知问题的变通方法:

- pluginManagement 中的 ResolutionStrategy, 对于存在多个变体的 plugin, 不能正常工作 (https://github.com/gradle/gradle/issues/20545)
- 当 Plugin 被添加为 `buildSrc` 的共通依赖项时, Plugin 变体会被忽略 (https://github.com/gradle/gradle/issues/20847)

在自定义配置中, Gradle 无法选择 KGP 变体

这是一种预料中的状况, 在自定义配置中, Gradle 无法选择 KGP 变体. 如果你使用了自定义的 Gradle 配置:

Kotlin

```
configurations.register("customConfiguration") {
    // ...
}
```

Groovy

```
configurations.register("customConfiguration") {
    // ...
}
```

并且你想要添加一个 Kotlin Gradle plugin 的依赖项, 例如:

Kotlin

```
dependencies {
    customConfiguration("org.jetbrains.kotlin:kotlin-gradle-
plugin:1.9.23")
}
```

Groovy

```
dependencies {
    customConfiguration 'org.jetbrains.kotlin:kotlin-gradle-
plugin:1.9.23'
}
```

你需要向你的 `customConfiguration` 添加以下属性:

Kotlin

```
configurations {
    customConfiguration {
        attributes {
            attribute(
                Usage.USAGE_ATTRIBUTE,
                project.objects.named(Usage.class,
Usage.JAVA_RUNTIME)
            )
            attribute(
                Category.CATEGORY_ATTRIBUTE,
                project.objects.named(Category.class,
Category.LIBRARY)
            )
            // 如果你想要使用某个特定的 KGP 变体的依赖:
            attribute(

GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                project.objects.named("7.0")
            )
```

```
        }
    }
}
```

Groovy

```
configurations {
    customConfiguration {
        attributes {
            attribute(
                Usage.USAGE_ATTRIBUTE,
                project.objects.named(Usage, Usage.JAVA_RUNTIME)
            )
            attribute(
                Category.CATEGORY_ATTRIBUTE,
                project.objects.named(Category,
Category.LIBRARY)
            )
            // 如果你想要使用某个特定的 KGP 变体的依赖:
            attribute(

GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                project.objects.named('7.0')
            )
        }
    }
}
```

否则, 你会看到这样的错误:

```
> Could not resolve all files for configuration
':customConfiguration'.
     > Could not resolve org.jetbrains.kotlin:kotlin-gradle-
plugin:1.7.0.
       Required by:
           project :
        > Cannot choose between the following variants of
```

```
org.jetbrains.kotlin:kotlin-gradle-plugin:1.7.0:
            - gradle70RuntimeElements
            - runtimeElements
          All of them match the consumer attributes:
            - Variant 'gradle70RuntimeElements' capability
org.jetbrains.kotlin:kotlin-gradle-plugin:1.7.0:
                - Unmatched attributes:
```

下一步做什么?

学习 Gradle 基本概念与详细信息 (https://docs.gradle.org/current/userguide/userguide.html).

Maven

最终更新: 2024/09/10

Maven 是一个构建系统, 你可以使用它来构建和管理基于 Java 的项目.

配置并启用插件

`kotlin-maven-plugin` 插件用来在 maven 环境中编译 Kotlin 源代码和模块. 目前只支持 Maven v3.

在你的 `pom.xml` 文件中, 可以通过 `kotlin.version` 属性来指定你希望使用的 Kotlin 版本:

```
<properties>
    <kotlin.version>1.9.23</kotlin.version>
</properties>
```

要启用 `kotlin-maven-plugin`, 请更新你的 `pom.xml` 文件:

```
<plugins>
    <plugin>
        <artifactId>kotlin-maven-plugin</artifactId>
        <groupId>org.jetbrains.kotlin</groupId>
        <version>1.9.23</version>
    </plugin>
</plugins>
```

使用 JDK 17

要使用 JDK 17, 请在你的 `.mvn/jvm.config` 文件中添加:

```
--add-opens=java.base/java.lang=ALL-UNNAMED
--add-opens=java.base/java.io=ALL-UNNAMED
```

声明仓库

默认情况下, 所有的 Maven 项目都可以使用 `mavenCentral` 仓库. 要访问其他仓库中的 artifact, 请在 `<repositories>` 元素中为各个仓库指定 ID 和 URL:

```xml
<repositories>
    <repository>
        <id>spring-repo</id>
        <url>https://repo.spring.io/release</url>
    </repository>
</repositories>
```

如果你在 Gradle 项目中将 mavenLocal() 声明为仓库, 你可能会在 Gradle 和 Maven 项目之间切换时遇到问题. 详情请参见 声明仓库 ("声明仓库" in "配置 Gradle 项目").

设置依赖项

Kotlin 有一个内容广泛的标准库, 可以在你的应用程序中使用. 要在你的项目中使用 Kotlin 标准库, 请在你的 pom.xml 文件中添加以下依赖项设置:

```xml
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

如果你的编译目标平台是 JDK 7 或 8

- 对于 Kotlin 1.8 以前版本, 请使用 kotlin-stdlib-jdk7 或 kotlin-stdlib-jdk8.
- 对于 Kotlin 1.2 以前版本, 请使用 kotlin-stdlib-jre7 或 kotlin-stdlib-jre8.

如果你的项目使用了 Kotlin 反射功能 (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect.full/index.html), 或测试功能, 那么还需要添加相应的依赖. 反射功能库的 artifact ID 是 kotlin-reflect, 测试功能库的 artifact ID 是 kotlin-test 和 kotlin-test-junit.

编译纯 Kotlin 源代码

要编译 Kotlin 源代码, 请在 `<build>` 标签内指定源代码目录:

```
<build>

<sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory
>

<testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceD
irectory>
</build>
```

编译源代码时, 需要引用 Kotlin Maven 插件:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>

            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>

                <execution>
                    <id>test-compile</id>
                    <goals>
                        <goal>test-compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
```

```
    </plugins>
</build>
```

从 Kotlin 1.8.20 开始, 你可以将上面的整个 `<executions>` 元素替换为 `<extensions>true</extensions>`. 启用 `extensions` 会向你的构建自动添加 `compile`, `test-compile`, `kapt`, 和 `test-kapt` 的 `execution`, 绑定到适当的 生命周期阶段 (https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html). 如果你需要配置某个 `execution`, 那么需要指定它的 ID. 在下一节中, 您可以找到这样的示例.

> 如果有多个构建插件覆盖了默认的生命周期, 而且你启用了 `extensions` 选项, 那么 `<build>` 节中的最后一个插件拥有生命周期设定优先权. 在它之前对生命周期设定的所有修改都会被忽略.

编译 Kotlin 与 Java 的混合源代码

要编译同时包含 Kotlin 与 Java 源代码的项目, 需要在 Java 编译器之前调用 Kotlin 编译器. 用 Maven 的术语来说就是, `kotlin-maven-plugin` 应该在 `maven-compiler-plugin` 之前运行. 也就是说, 在你的 `pom.xml` 文件中, `kotlin` plugin 要放在 `maven-compiler-plugin` 之前, 如下例:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <extensions>true</extensions> <!-- 你可以设置这个选项, 自动
获取有关生命周期的信息 -->
            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal> <!-- 如果你对 plugin 启用
了 extensions, 那么可以省略 <goals> 元素 -->
                    </goals>
                    <configuration>
                        <sourceDirs>

<sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
```

```
<sourceDir>${project.basedir}/src/main/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <goals>
                        <goal>test-compile</goal> <!-- 如果你对
plugin 启用了 extensions，那么可以省略 <goals> 元素 -->
                    </goals>
                    <configuration>
                        <sourceDirs>

<sourceDir>${project.basedir}/src/test/kotlin</sourceDir>

<sourceDir>${project.basedir}/src/test/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.5.1</version>
            <executions>
                <!-- 替换 default-compile，因为它会被 Maven 特别处理 -->
                <execution>
                    <id>default-compile</id>
                    <phase>none</phase>
                </execution>
                <!-- 替换 default-testCompile，因为它会被 Maven 特别处理
-->
                <execution>
                    <id>default-testCompile</id>
                    <phase>none</phase>
                </execution>
                <execution>
```

```
                    <id>java-compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>java-test-compile</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

启用增量编译(Incremental compilation)

为了提高编译速度, 你可以启用增量编译模式, 方法是添加 `kotlin.compiler.incremental` 属性:

```
<properties>
    <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>
```

或者, 使用命令行选项 `-Dkotlin.compiler.incremental=true` 来执行你的编译任务.

配置注解处理

详情请参见 `kapt` – 在 Maven 中使用 ("在 Maven 中使用" in "kapt 编译器插件").

创建 JAR 文件

假如要创建一个小的 JAR 文件, 其中只包含你的模块中的代码, 那么请将以下代码添加到你的 Maven `pom.xml` 文件的 `build->plugins` 之下, 其中的 `main.class` 是一个属性, 指向 Kotlin 或 Java 的 main class:

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>${main.class}</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

创建自包含的(Self-contained) JAR 文件

要创建一个自包含的 JAR 文件, 其中包含你的模块中的代码, 以及它依赖的库文件, 那么请将以下代码添加到你的 Maven `pom.xml` 文件的 `build->plugins` 之下, 其中的 `main.class` 是一个属性, 指向 Kotlin 或 Java 的 main class:

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.6</version>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals> <goal>single</goal> </goals>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>${main.class}</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-
dependencies</descriptorRef>
```

```xml
                </descriptorRefs>
            </configuration>
        </execution>
    </executions>
</plugin>
```

编译产生的自包含的 JAR 文件, 可以直接传递给一个 JRE, 然后就可以运行你的应用程序了:

```
java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

指定编译器选项

额外的编译器选项和参数, 可以通过 Maven plugin 节点的 `<configuration>` 元素下的标签来设置:

```xml
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <extensions>true</extensions> <!-- 如果你想要为你的构建自动添加
executions, 可以添加这个选项 -->
    <executions>...</executions>
    <configuration>
        <nowarn>true</nowarn>  <!-- 关闭警告信息 -->
        <args>
            <arg>-Xjsr305=strict</arg> <!-- 对 JSR-305 注解使用 strict
模式 -->
            ...
        </args>
    </configuration>
</plugin>
```

很多编译器选项也可以通过属性来设置:

```xml
<project ...>
    <properties>

<kotlin.compiler.languageVersion>1.9</kotlin.compiler.languageVersio
n>
```

```
    </properties>
</project>
```

支持的编译选项列表如下:

JVM 独有的属性

名称	Maven 属性名	描述	可以选择的值	默认值
nowarn		不产生警告信息	true, false	false
languageVersion	kotlin.compiler.languageVersion	指定源代码所兼容的 Kotlin 语言版本	"1.3" (已废弃 DEPRECATED), "1.4" (已废弃 DEPRECATED), "1.5", "1.6", "1.7", "1.8", "1.9", "2.0" (实验性功能), "2.1" (实验性功能)	
apiVersion	kotlin.compiler.apiVersion	只允许使用指定的版本的运行库中的 API	"1.3" (已废弃 DEPRECATED), "1.4" (已废弃 DEPRECATED), "1.5", "1.6", "1.7", "1.8", "1.9", "2.0" (实验性功能), "2.1" (实验性功能)	
sourceDirs		指定编译对象源代码文件所在的目录		工程的源代码根路径
compilerPlugins		允许使用编译器插件		[]
pluginOptions		供编译器插件使用的选项		[]
args		额外的编译器参数		[]
jvmTarget	kotlin.compiler.jvmTarget	指定编译输出的 JVM 字节码的版本	"1.8", "9", "10", ..., "21"	"1.8"

jdkHome	kotlin.compiler.jdkHome	指定一个自定义的 JDK 路径, 添加到 classpath 内, 替代默认的 JAVA_HOME 值		

使用 BOM

要使用 Kotlin Bill of Materials (BOM) (https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#bill-of-materials-bom-poms), 请添加 kotlin-bom (https://mvnrepository.com/artifact/org.jetbrains.kotlin/kotlin-bom) 的依赖项:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-bom</artifactId>
            <version>1.9.23</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

生成文档

标准的 Javadoc 生成 plugin (maven-javadoc-plugin) 不支持 Kotlin 源代码. 要对 Kotlin 项目生成文档, 请使用 Dokka (https://github.com/Kotlin/dokka). Dokka 支持混合语言的项目, 可以将文档输出为多种格式, 包括标准的 Javadoc 格式. 关于如何在你的 Maven 项目中配置 Dokka, 请参见 Maven (Maven).

启用 OSGi 支持

参见 如何在你的 Maven 项目中启用 OSGi 支持 ("Maven" in "Kotlin 与 OSGi").

Ant

最终更新: 2024/09/10

安装 Ant Task

Kotlin 提供了 3 个 Ant Task:

- kotlinc: 面向 JVM 的 Kotlin 编译器
- kotlin2js: 面向 JavaScript 的 Kotlin 编译器
- withKotlin: 使用标准的 *javac* Ant Task 来编译 Kotlin 代码

这些 Task 定义在 *kotlin-ant.jar* 库文件内, 这个库文件位于 Kotlin 编译器 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.23) 的 lib 文件夹内. 需要的 Ant 版本是 1.8.2 以上.

面向 JVM, 编译纯 Kotlin 代码

如果工程内只包含 Kotlin 源代码, 这种情况下最简单的编译方法是使用 kotlinc Task:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc src="hello.kt" output="hello.jar"/>
    </target>
</project>
```

这里的 ${kotlin.lib} 指向 Kotlin standalone 编译器解压缩后的文件夹.

面向 JVM, 编译包含多个根目录的纯 Kotlin 代码

如果工程中包含多个源代码根目录, 可以使用 src 元素来定义源代码路径:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
```

```
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="root1"/>
            <src path="root2"/>
        </kotlinc>
    </target>
</project>
```

面向 JVM, 编译 Kotlin 和 Java 的混合代码

如果工程包含 Kotlin 和 Java 的混合代码, 这时尽管也能够使用 `kotlinc`, 但为了避免重复指定 Task 参数, 推荐使用 `withKotlin` Task:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <delete dir="classes" failonerror="false"/>
        <mkdir dir="classes"/>
        <javac destdir="classes" includeAntRuntime="false"
srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

还可以通过 `moduleName` 属性来指定被编译的模块名称:

```
<withKotlin moduleName="myModule"/>
```

面向 JavaScript, 编译单个源代码文件夹

```xml
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js"/>
    </target>
</project>
```

面向 JavaScript, 使用 Prefix, PostFix 和 sourcemap 选项

```xml
<project name="Ant Task Test" default="build">
    <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml"
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js" outputPrefix="prefix"
outputPostfix="postfix" sourcemap="true"/>
    </target>
</project>
```

面向 JavaScript, 编译单个源代码文件夹, 使用 metaInfo 选项

如果你希望将编译结果当作一个 Kotlin/JavaScript 库发布, 可以使用 `metaInfo` 选项. 如果 `metaInfo` 设值为 `true`, 那么编译时会额外创建带二进制元数据(binary metadata)的 JS 文件. 这个文件需要与编译结果一起发布:

```xml
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <!-- 会创建 out.meta.js 文件, 其中包含二进制元数据(binary
metadata) -->
        <kotlin2js src="root1" output="out.js" metaInfo="true"/>
```

```
    </target>
</project>
```

参照

完整的 Ant Task 元素和属性一览表如下:

kotlinc 和 kotlin2js 的共通属性

名称	说明	是否必须	默认值
`src`	需要编译的 Kotlin 源代码文件或源代码目录	是	
`nowarn`	屏蔽编译时的警告信息	否	false
`noStdlib`	不要将 Kotlin 标准库包含在 classpath 内	否	false
`failOnError`	如果编译过程中检测到错误, 是否让整个构建过程失败	否	true

kotlinc 独有的属性

名称	说明	是否必须	默认值
`output`	编译输出的目标目录, 或目标 .jar 文件名	是	
`classpath`	编译时的 class path 值	否	
`classpathref`	编译时的 class path 参照	否	
`includeRuntime`	当 `output` 是 .jar 文件时, 是否将 Kotlin 运行库包含在这个 jar 内	否	true
`moduleName`	被编译的模块名称	否	编译目标的名称(如果有指定), 或工程名称

kotlin2js 独有的属性

名称	说明	是否必须
output	编译输出的目标文件	是
libraries	Kotlin 库文件路径	No
outputPrefix	生成 JavaScript 文件时使用的前缀	否
outputSuffix	生成 JavaScript 文件时使用的后缀	否
sourcemap	是否生成 sourcemap 文件	否
metaInfo	是否生成带二进制描述符(binary descriptor)的元数据(metadata)文件	否
main	编译器是否生成对 main 函数的调用代码	否

指定编译参数

如果需要指定自定义的编译参数, 可以使用 `<compilerarg>` 元素的 `value` 或 `line` 属性. 这个元素可以放在 `<kotlinc>`, `<kotlin2js>`, 以及 `<withKotlin>` 任务元素之内, 示例如下:

```
<kotlinc src="${test.data}/hello.kt" output="${temp}/hello.jar">
    <compilerarg value="-Xno-inline"/>
    <compilerarg line="-Xno-call-assertions -Xno-param-assertions"/>
    <compilerarg value="-Xno-optimize"/>
</kotlinc>
```

运行 `kotlinc -help` 命令, 可以看到参数的完整列表.

介绍

最终更新: 2024/09/10

Dokka 是一个用于 Kotlin 的 API 文档引擎.

和 Kotlin 本身一样, Dokka 支持混合语言的项目. 它能够理解 Kotlin 的 KDoc 注释 ("KDoc 语法" in "为 Kotlin 代码编写文档: KDoc") 和 Java 的 Javadoc 注释 (https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html).

Dokka 能够使用很多种格式生成文档, 包括它自己的现代化 HTML 格式 (HTML), 多种风格的 Markdown 格式 (Markdown), 以及 Java 的 Javadoc HTML 格式 (Javadoc).

下面是一些库, 它们使用 Dokka 来生成 API 参考文档:

- kotlinx.coroutines (https://kotlinlang.org/api/kotlinx.coroutines/)
- Bitmovin (https://cdn.bitmovin.com/player/android/3/docs/index.html)
- Hexagon (https://hexagonkt.com/api/index.html)
- Ktor (https://api.ktor.io/)
- OkHttp (https://square.github.io/okhttp/5.x/okhttp/okhttp3/)

要运行 Dokka, 你可以使用 Gradle (Gradle), Maven (Maven) 或者 命令行 (CLI). 它也是 高度插件化的 (Dokka Plugin).

要开始使用 Dokka, 首先请参见 Dokka 入门 (Dokka 入门) 文档.

社区

在 Kotlin Community Slack (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up) 中有专门的 `#dokka` 频道, 你可以在这里讨论 Dokka 相关的问题, 包括它的 plugin, 如何开发, 也可以与维护者保持接触.

Dokka 入门

最终更新: 2024/09/10

下面你可以看到一段简单的指南, 帮助你开始学习使用 Dokka.

Gradle Kotlin DSL

在你的项目的根构建脚本中应用 Gradle plugin for Dokka:

```
plugins {
    id("org.jetbrains.dokka") version "1.9.20"
}
```

如果要对 多项目(multi-project) (https://docs.gradle.org/current/userguide/multi_project_builds.html) 构建生成文档, 你还需要对各个子项目应用 Gradle plugin:

```
subprojects {
    apply(plugin = "org.jetbrains.dokka")
}
```

要生成文档, 需要运行以下 Gradle task:

- `dokkaHtml`: 用于单项目构建
- `dokkaHtmlMultiModule`: 用于多项目构建

输出目录默认设置为 `/build/dokka/html` 和 `/build/dokka/htmlMultiModule`.
关于如何在 Gradle 中使用 Dokka, 更多详情请参见 Gradle (Gradle).

Gradle Groovy DSL

在你的项目的根构建脚本中应用 Gradle plugin for Dokka:

```
plugins {
    id 'org.jetbrains.dokka' version '1.9.20'
}
```

如果要对 多项目(multi-project) (https://docs.gradle.org/current/userguide/multi_project_builds.html) 构建生成文档, 你

还需要对各个子项目应用 Gradle plugin:

```
subprojects {
    apply plugin: 'org.jetbrains.dokka'
}
```

要生成文档, 需要运行以下 Gradle task:

- dokkaHtml: 用于单项目构建
- dokkaHtmlMultiModule: 用于多项目构建

输出目录默认设置为 /build/dokka/html 和 /build/dokka/htmlMultiModule.
关于如何在 Gradle 中使用 Dokka, 更多详情请参见 Gradle (Gradle).

Maven

在你的 POM 文件的 plugins 小节添加 Maven plugin for Dokka:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.dokka</groupId>
            <artifactId>dokka-maven-plugin</artifactId>
            <version>1.9.20</version>
            <executions>
                <execution>
                    <phase>pre-site</phase>
                    <goals>
                        <goal>dokka</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

要生成文档, 需要运行 dokka:dokka goal.
输出目录默认设置为 target/dokka.
关于如何在 Maven 中使用 Dokka, 更多详情请参见 Maven (Maven).

Gradle

最终更新: 2024/09/10

要为基于 Gradle 的项目要生成文档, 你可以使用 Gradle plugin for Dokka (https://plugins.gradle.org/plugin/org.jetbrains.dokka).

它对你的项目进行了基本的自动配置, 带有方便的 Gradle task 用于生成文档, 还提供了大量的 配置选项 用来定制输出.

你可以访问我们的 Gradle 示例项目 (https://github.com/Kotlin/dokka/tree/{{ site.data.releases.dokkaVersion }}/examples/gradle). 实际接触一下 Dokka, 看看它如何对各种项目进行配置.

应用 Dokka

应用 Gradle plugin for Dokka 时, 推荐的方式是使用 plugin DSL (https://docs.gradle.org/current/userguide/plugins.html#sec:plugins_block):

Kotlin

```
plugins {
    id("org.jetbrains.dokka") version "1.9.20"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.dokka' version '1.9.20'
}
```

要对 多项目 构建生成文档, 你还需要对子项目应用 Gradle plugin for Dokka. 你可以使用 Gradle 配置 `allprojects {}` 或 `subprojects {}` 来做到这一点:

Gradle Kotlin DSL

```
subprojects {
    apply(plugin = "org.jetbrains.dokka")
}
```

Gradle Groovy DSL

```
subprojects {
    apply plugin: 'org.jetbrains.dokka'
}
```

如果你不确定在哪里应用 Dokka, 请参见 配置示例.

> Dokka 内部会使用 Kotlin Gradle plugin ("应用(Apply) Kotlin Gradle Plugin" in "配置 Gradle 项目") 来对需要生成文档的 源代码集 ("源代码集(Source Set)" in "Kotlin Multiplatform 项目结构的基础知识") 进行自动配置. 请确认应用了 Kotlin Gradle Plugin, 或手动的 配置了源代码集.

> 如果你在 预编译的脚本 plugin (https://docs.gradle.org/current/userguide/custom_plugins.html#sec:precompiled_plugins) 中使用 Dokka, 你需要将 Kotlin Gradle plugin ("应用(Apply) Kotlin Gradle Plugin" in "配置 Gradle 项目") 添加为它的依赖项, 才能让它正常工作.

如果你因为某种原因无法使用 plugin DSL, 你可以使用 旧的方式 (https://docs.gradle.org/current/userguide/plugins.html#sec:old_plugin_application) 来应用 plugin.

生成文档

Gradle plugin for Dokka 默认带有 HTML (HTML), Markdown (Markdown) 和 Javadoc (Javadoc) 输出格式. 对 单项目构建和 多项目 构建, 它都添加了很多 task 用于生成文档.

单项目构建

对简单的单项目应用程序和库, 请使用以下 task 来构建文档:

Task	描述
dokkaHtml	使用 HTML (HTML) 格式生成文档.

实验性的格式

Task	描述
dokkaGfm	使用 GitHub 风格的 Markdown ("GFM" in "Markdown") 格式生成文档.
dokkaJavadoc	使用 Javadoc (Javadoc) 格式生成文档.
dokkaJekyll	使用 Jekyll 兼容的 Markdown ("Jekyll" in "Markdown") 格式生成文档.

默认情况下, 生成的文档会输出到你的项目的 build/dokka/{format} 目录中. 输出位置, 以及其他很多设置, 都可以进行 配置.

多项目构建

要对 多项目构建 (https://docs.gradle.org/current/userguide/multi_project_builds.html) 生成文档, 请确认, 不仅对父项目, 也对你想要生成文档的子项目 应用了 Gradle plugin for Dokka.

MultiModule task

MultiModule task 通过 Partial task 对每个子项目分别生成文档, 收集并处理所有的输出, 然后使用共同的目录和解析后的跨项目引用, 处理完整的文档.

Dokka 对 父 项目 自动创建以下 task:

Task	描述
dokkaHtmlMultiModule	使用 HTML (HTML) 输出格式生成多模块文档.

实验性的格式 (MultiModule)

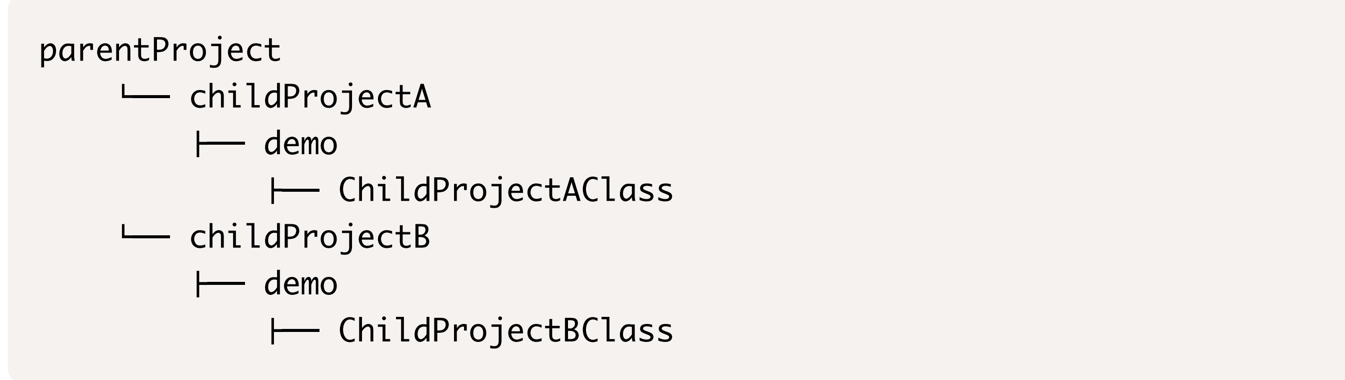
Task	描述
dokkaGfmMultiModule	使用 GitHub 风格的 Markdown ("GFM" in "Markdown") 输出格式生成多模块文档.
dokkaJekyllMultiModule	使用 Jekyll 兼容的 Markdown ("Jekyll" in "Markdown") 输出格式生成多模块文档.

> Javadoc (Javadoc) 输出格式没有 MultiModule task, 但可以改为使用 Collector task.

默认情况下, 你可以在 {parentProject}/build/dokka/{format}MultiModule 目录中找到直接可用的文档.

MultiModule task 的输出结果

假设一个项目的结构如下:



```
parentProject
    └── childProjectA
        ├── demo
            ├── ChildProjectAClass
    └── childProjectB
        ├── demo
            ├── ChildProjectBClass
```

运行 dokkaHtmlMultiModule 后生成的文档如下:

parentProject

- childProjectA
  - demo
    - ChildProjectAClass
- childProjectB
  - demo
    - ChildProjectBClass

All modules:

childProjectA

This is the child module a

childProjectB

This is the child module b

dokkaHtmlMultiModule task 的输出画面截图

更多详情, 请参见我们的 多模块项目示例 (https://github.com/Kotlin/dokka/tree/{{ site.data.releases.dokkaVersion }}/examples/gradle/dokka-multimodule-example).

Collector task

与 `MultiModule` task 类似, 对各个父项目创建了 `Collector` task: `dokkaHtmlCollector`, `dokkaGfmCollector`, `dokkaJavadocCollector` 以及 `dokkaJekyllCollector`.

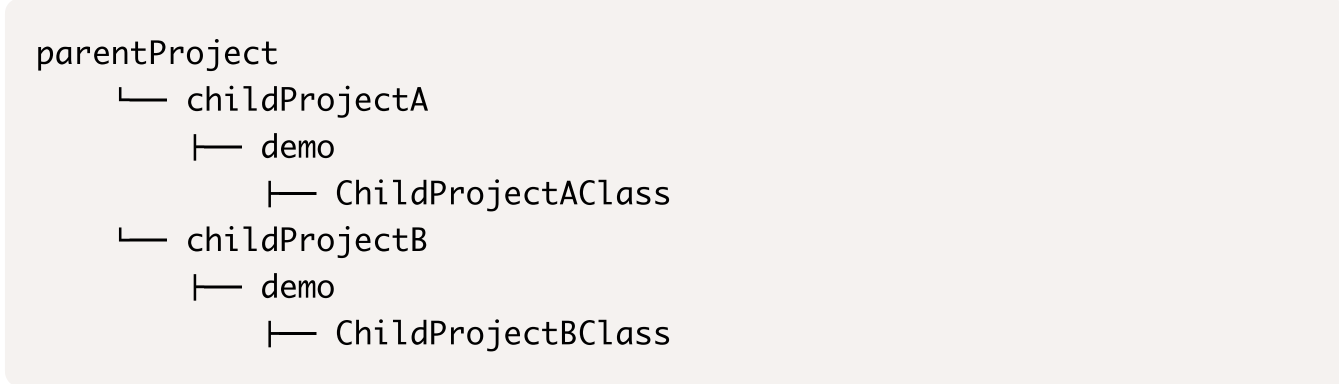
`Collector` task 会对每个子项目执行对应的 单项目 task (例如, `dokkaHtml`), 并将所有的输出合并到一个单独的虚拟项目.

最终生成的结果文档, 看起来就好象一个单项目构建, 其中包含来自子项目的所有声明.

> 如果你需要为你的多项目构建创建 Javadoc 文档, 请使用 `dokkaJavadocCollector` task.

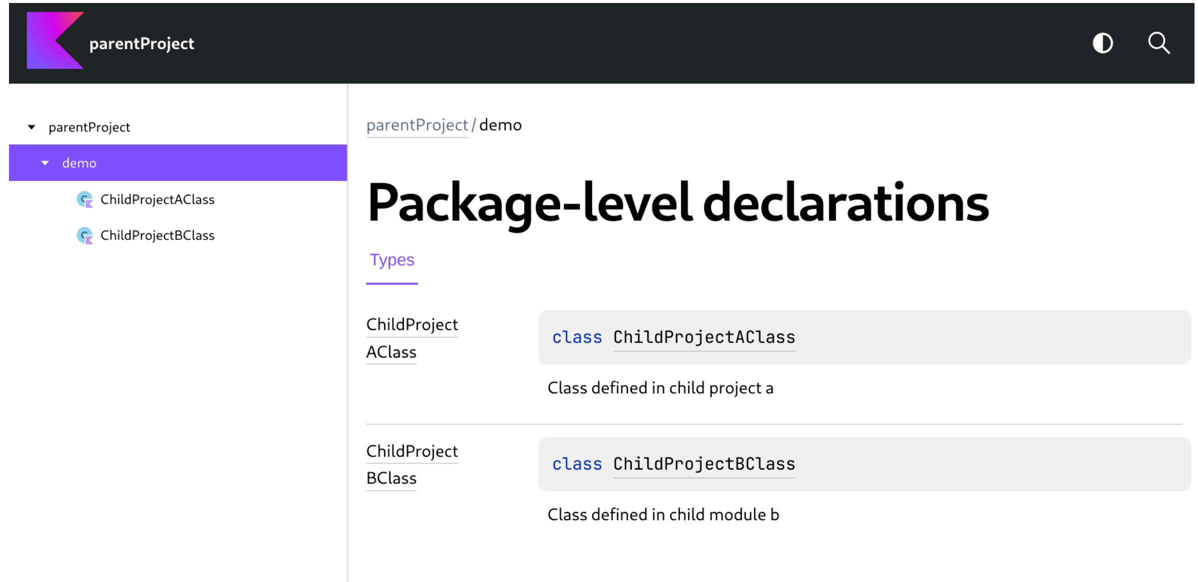
Collector task 的输出结果

假设一个项目的结构如下:



```
parentProject
    └── childProjectA
        ├── demo
            ├── ChildProjectAClass
    └── childProjectB
        ├── demo
            ├── ChildProjectBClass
```

运行 `dokkaHtmlCollector` 之后会生成这样的页面:



dokkaHtmlCollector task 的输出画面截图

更多详情, 请参见我们的 多模块项目示例 (https://github.com/Kotlin/dokka/tree/{{ site.data.releases.dokkaVersion }}/examples/gradle/dokka-multimodule-example).

Partial task

对每个子项目都创建了 `Partial` task: `dokkaHtmlPartial`, `dokkaGfmPartial`, 和 `dokkaJekyllPartial`.

这些 task 并不用来单独运行, 它们会被父项目的 MultiModule task 调用.

但是, 你可以 配置 `Partial` task, 为你的子项目定制 Dokka.

> `Partial` task 生成的输出包含未解析的 HTML 模板和引用, 因此在父项目的 `MultiModule` task 进行后续处理之前, 这些文档还不能直接使用.

> 如果你只想对单个子项目生成文档, 请使用 单项目 task. 例如, :subprojectName:dokkaHtml.

构建 javadoc.jar

如果你想要将你的库发布到仓库, 你可能需要提供一个 javadoc.jar 文件, 其中包含你的库的 API 参考文档.

例如, 如果你想要发布到 Maven Central (https://central.sonatype.org/), 你 必须 (https://central.sonatype.org/publish/requirements/) 和你的项目一起提供一个 javadoc.jar. 但是, 并不是所有的仓库都有这样的规则.

Gradle plugin for Dokka 没有提供任何方式来直接完成这个任务, 但可以通过自定义的 Gradle task 实现. 下面的示例中, 一个 task 使用 HTML (HTML) 格式生成文档, 另一个使用 Javadoc (Javadoc) 格式:

Kotlin

```kotlin
tasks.register<Jar>("dokkaHtmlJar") {
    dependsOn(tasks.dokkaHtml)
    from(tasks.dokkaHtml.flatMap { it.outputDirectory })
    archiveClassifier.set("html-docs")
}

tasks.register<Jar>("dokkaJavadocJar") {
    dependsOn(tasks.dokkaJavadoc)
    from(tasks.dokkaJavadoc.flatMap { it.outputDirectory })
    archiveClassifier.set("javadoc")
}
```

Groovy

```groovy
tasks.register('dokkaHtmlJar', Jar.class) {
    dependsOn(dokkaHtml)
    from(dokkaHtml)
    archiveClassifier.set("html-docs")
}
```

```
tasks.register('dokkaJavadocJar', Jar.class) {
    dependsOn(dokkaJavadoc)
    from(dokkaJavadoc)
    archiveClassifier.set("javadoc")
}
```

> ⚠ 如果你将你的库发布到 Maven Central, 你可以使用 javadoc.io (https://javadoc.io/) 之类的服务, 免费托管你的库的 API 文档, 而且不需要任何设置. 它直接从 `javadoc.jar` 得到文档页面. 它可以很好的显示 HTML 格式文档, 参见 这个示例 (https://javadoc.io/doc/com.trib3/server/latest/index.html).

配置示例

根据你的项目类型不同, 你应用和配置 Dokka 的方式也略有不同. 但是, 配置选项 本身是相同的, 无论你的项目类型如何.

对于简单的项目, 在项目的根目录下包含单个 `build.gradle.kts` 或 `build.gradle` 文件, 请参见 单项目配置.

对更加复杂的构建, 包含子项目, 以及多个下级的 `build.gradle.kts` 或 `build.gradle` 文件, 请参见 多项目 配置.

单项目配置

单项目构建, 通常只在项目的根目录下存在单个 `build.gradle.kts` 或 `build.gradle` 文件, 其结构通常如下:

Kotlin

单平台项目:

```
.
├── build.gradle.kts
└── src
    └── main
        └── kotlin
            └── HelloWorld.kt
```

跨平台项目:

```
.
├── build.gradle.kts
└── src
    └── commonMain
        └── kotlin
            └── Common.kt
    └── jvmMain
        └── kotlin
            └── JvmUtils.kt
    └── nativeMain
        └── kotlin
            └── NativeUtils.kt
```

Groovy

单平台项目:

```
.
├── build.gradle
└── src
    └── main
        └── kotlin
            └── HelloWorld.kt
```

跨平台项目:

```
.
├── build.gradle
└── src
    └── commonMain
        └── kotlin
            └── Common.kt
    └── jvmMain
        └── kotlin
            └── JvmUtils.kt
    └── nativeMain
```

```
        └── kotlin
            └── NativeUtils.kt
```

在这样的项目中, 你需要在根目录的 `build.gradle.kts` 或 `build.gradle` 文件中应用 Dokka 及其配置.

你可以单独配置 task 和输出格式:

Kotlin

在 `./build.gradle.kts` 中:

```
plugins {
    id("org.jetbrains.dokka") version "1.9.20"
}

tasks.dokkaHtml {

outputDirectory.set(layout.buildDirectory.dir("documentation/htm
l"))
}

tasks.dokkaGfm {

outputDirectory.set(layout.buildDirectory.dir("documentation/mar
kdown"))
}
```

Groovy

在 `./build.gradle` 中:

```
plugins {
    id 'org.jetbrains.dokka' version '1.9.20'
}

dokkaHtml {
    outputDirectory.set(file("build/documentation/html"))
}
```

```
dokkaGfm {
    outputDirectory.set(file("build/documentation/markdown"))
}
```

或者你也可以同时配置所有的 task 和输出格式:

Kotlin

在 ./build.gradle.kts 中:

```
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.gradle.DokkaTaskPartial
import org.jetbrains.dokka.DokkaConfiguration.Visibility

plugins {
    id("org.jetbrains.dokka") version "1.9.20"
}

// 同时配置所有的单项目 Dokka task,
// 例如 dokkaHtml, dokkaJavadoc 和 dokkaGfm.
tasks.withType<DokkaTask>().configureEach {
    dokkaSourceSets.configureEach {
        documentedVisibilities.set(
            setOf(
                Visibility.PUBLIC,
                Visibility.PROTECTED,
            )
        )

        perPackageOption {
            matchingRegex.set(".*internal.*")
            suppress.set(true)
        }
    }
}
```

Groovy

在 ./build.gradle 中:

```
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.gradle.DokkaTaskPartial
import org.jetbrains.dokka.DokkaConfiguration.Visibility

plugins {
    id 'org.jetbrains.dokka' version '1.9.20'
}

// 同时配置所有的单项目 Dokka task,
// 例如 dokkaHtml, dokkaJavadoc 和 dokkaGfm.
tasks.withType(DokkaTask.class) {
    dokkaSourceSets.configureEach {
        documentedVisibilities.set([
                Visibility.PUBLIC,
                Visibility.PROTECTED
        ])

        perPackageOption {
            matchingRegex.set(".*internal.*")
            suppress.set(true)
        }
    }
}
```

多项目配置

Gradle 的 多项目构建 (https://docs.gradle.org/current/userguide/multi_project_builds.html) 的结构和配置都更加复杂. 这样的项目通常包含多个下级的 build.gradle.kts 或 build.gradle 文件, 其结构通常如下:

Kotlin

```
.
├── build.gradle.kts
├── settings.gradle.kts
├── subproject-A
```

```
    └── build.gradle.kts
    └── src
        └── main
            └── kotlin
                └── HelloFromA.kt
├── subproject-B
    └── build.gradle.kts
    └── src
        └── main
            └── kotlin
                └── HelloFromB.kt
```

Groovy

```
.
├── build.gradle
├── settings.gradle
├── subproject-A
    └── build.gradle
    └── src
        └── main
            └── kotlin
                └── HelloFromA.kt
├── subproject-B
    └── build.gradle
    └── src
        └── main
            └── kotlin
                └── HelloFromB.kt
```

对于这样的项目, 有很多种方式来应用和配置 Dokka.

子项目配置

要在多项目构建中配置子项目, 你需要配置 `Partial` task.

你可以在根目录的 `build.gradle.kts` 或 `build.gradle` 文件中, 使用 Gradle 的 `allprojects {}` 或 `subprojects {}` 配置代码块, 同时配置所有的子项目:

Kotlin

在根目录的 `./build.gradle.kts` 中:

```
import org.jetbrains.dokka.gradle.DokkaTaskPartial

plugins {
    id("org.jetbrains.dokka") version "1.9.20"
}

subprojects {
    apply(plugin = "org.jetbrains.dokka")

    // 只配置 HTML task
    tasks.dokkaHtmlPartial {

outputDirectory.set(layout.buildDirectory.dir("docs/partial"))
    }

    // 配置所有格式
    tasks.withType<DokkaTaskPartial>().configureEach {
        dokkaSourceSets.configureEach {
            includes.from("README.md")
        }
    }
}
```

Groovy

在根目录的 `./build.gradle` 中:

```
import org.jetbrains.dokka.gradle.DokkaTaskPartial

plugins {
    id 'org.jetbrains.dokka' version '1.9.20'
}

subprojects {
```

```
    apply plugin: 'org.jetbrains.dokka'

    // 只配置 HTML task
    dokkaHtmlPartial {
        outputDirectory.set(file("build/docs/partial"))
    }

    // 配置所有格式
    tasks.withType(DokkaTaskPartial.class) {
        dokkaSourceSets.configureEach {
            includes.from("README.md")
        }
    }
}
```

另一种方法是, 你可以对各个子项目分别应用和配置 Dokka.

例如, 为了只对 `subproject-A` 子项目进行特定的设置, 你需要在 `./subproject-A/build.gradle.kts` 中使用以下代码:

Kotlin

在 `./subproject-A/build.gradle.kts` 中:

```
apply(plugin = "org.jetbrains.dokka")

// 只配置 subproject-A.
tasks.dokkaHtmlPartial {

outputDirectory.set(layout.buildDirectory.dir("docs/partial"))
}
```

Groovy

在 `./subproject-A/build.gradle` 中:

```
apply plugin: 'org.jetbrains.dokka'

// 只配置 subproject-A.
```

```
dokkaHtmlPartial {
    outputDirectory.set(file("build/docs/partial"))
}
```

父项目配置

如果你想要配置跨越所有文档的某个设定, 而且它不属于子项目 - 也就是说, 它是父项目的某个属性 - 那么你需要配置 `MultiModule` task.

例如, 如果你想要修改 HTML 文档标题中的项目名称, 你需要在根目录的 `build.gradle.kts` 或 `build.gradle` 文件中使用以下代码:

Kotlin

在根目录的 `./build.gradle.kts` 文件中:

```
plugins {
    id("org.jetbrains.dokka") version "1.9.20"
}

tasks.dokkaHtmlMultiModule {
    moduleName.set("在标题中使用的项目名称")
}
```

Groovy

在根目录的 `./build.gradle` 文件中:

```
plugins {
    id 'org.jetbrains.dokka' version '1.9.20'
}

dokkaHtmlMultiModule {
    moduleName.set("在标题中使用的项目名称")
}
```

配置选项

Dokka 有很多配置选项, 可以用来定制你和你的读者的体验.

下面是对每个配置小节的一些示例和详细解释. 在本章的最后, 你还可以看到一个示例, 它使用了 所有的配置选项.

关于应该在哪里使用配置代码块, 以及如何使用, 请参见 配置示例.

一般配置

下面是所有 Dokka task 的一般配置的示例, 无论是对源代码集还是包:

Kotlin

```
import org.jetbrains.dokka.gradle.DokkaTask

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//       参见本文档的 "配置示例" 小节.
tasks.withType<DokkaTask>().configureEach {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())

outputDirectory.set(layout.buildDirectory.dir("dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    // ...
    // 参见本文档的 "源代码集配置" 小节
    // ...
}
```

Groovy

```
import org.jetbrains.dokka.gradle.DokkaTask

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//       参见本文档的 "配置示例" 小节.
tasks.withType(DokkaTask.class) {
```

```
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(file("build/dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    // ...
    // 参见本文档的 "源代码集配置" 小节
    // ...
}
```

moduleName

用来引用模块的显示名称. 这个名称会用于目录, 导航, 日志, 等等.

如果对单项目构建或 MultiModule task 设置这个选项, 它会被用作项目名称.

默认值: Gradle 项目名称

moduleVersion

模块版本. 如果对单项目构建或 MultiModule task 设置这个选项, 它会被用作项目版本.

默认值: Gradle 项目版本

outputDirectory

文档生成的目录, 无论哪种格式. 这个选项可以对每个 task 进行设置.

默认值是 {project}/{buildDir}/{format}, 其中 {format} 是 task 名称, 删去了 "dokka" 前缀. 例如, 对于 dokkaHtmlMultiModule task, 输出目录是 project/buildDir/htmlMultiModule.

failOnWarning

如果 Dokka 输出警告或错误, 是否让文档生成任务失败. 进程首先会等待所有的错误和警告输出完毕.

这个设置可以与 reportUndocumented 选项配合工作.

默认值: false

suppressObviousFunctions

是否禁止输出那些显而易见的函数.

满足以下条件的函数, 会被认为是显而易见的函数:

- 继承自 kotlin.Any, Kotlin.Enum, java.lang.Object 或 java.lang.Enum, 例如 equals, hashCode, toString.
- 合成(由编译器生成的)函数, 而且没有任何文档, 例如 dataClass.componentN 或 dataClass.copy.

默认值: true

suppressInheritedMembers

是否禁止输出在指定的类中继承得到的而且没有显式覆盖的成员.

注意: 这个选项可以禁止输出 equals/hashCode/toString 之类的函数, 但不能禁止输出 dataClass.componentN 和 dataClass.copy 之类的合成函数. 对合成函数, 请使用 suppressObviousFunctions 选项.

默认值: false

offlineMode

是否通过你的网络来解析远程的文件/链接.

包括用来生成外部文档链接的包列表. 例如, 可以让来自标准库的类成为文档中可以点击的链接.

将这个选项设置为 true, 某些情况下可以显著提高构建速度, 但也会降低文档质量和用户体验. 例如, 可以不解析来自你的依赖项的类/成员的链接, 包括标准库.

注意: 你可以将已取得的文件缓存到本地, 并通过本地路径提供给 Dokka. 参见 externalDocumentationLinks 小节.

默认值: false

源代码集配置

Dokka 对 Kotlin 源代码集 ("源代码集(Source Set)" in "Kotlin Multiplatform 项目结构的基础知识") 允许配置一些选项:

Kotlin

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//       参见本文档的 "配置示例" 小节.
tasks.withType<DokkaTask>().configureEach {
    // ...
    // 参见本文档的 "一般配置" 小节
    // ...

    dokkaSourceSets {
        // 专属于 'linux' 源代码集的配置
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
```

```
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md",
"extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(),
file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt",
"samples/Advanced.kt")

            sourceLink {
                // 参见本文档的 "源代码链接配置" 小节
            }
            externalDocumentationLink {
                // 参见本文档的 "外部文档链接配置" 小节
            }
            perPackageOption {
                // 参见本文档的 "包选项" 小节
            }
        }
    }
}
```

Groovy

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//      参见本文档的 "配置示例" 小节.
tasks.withType(DokkaTask.class) {
    // ...
    // 参见本文档的 "一般配置" 小节
    // ...
```

```
    dokkaSourceSets {
        // 专属于 'linux' 源代码集的配置
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set([Visibility.PUBLIC])
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md",
"extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(),
file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt",
"samples/Advanced.kt")

            sourceLink {
                // 参见本文档的 "源代码链接配置" 小节
            }
            externalDocumentationLink {
                // 参见本文档的 "外部文档链接配置" 小节
            }
            perPackageOption {
                // 参见本文档的 "包选项" 小节
            }
```

```
            }
        }
    }
```

suppress

在生成文档时, 是否应该跳过这个源代码集.

默认值: false

displayName

用来引用这个源代码集的显示名称.

这个名称在外部用途使用(例如, 源代码集名称会显示给文档读者), 也在内部使用(例如, 用于 reportUndocumented 的日志信息).

默认情况下, 这个值会从 Kotlin Gradle plugin 提供的信息推断得到.

documentedVisibilities

这个选项设置需要生成文档的可见度修饰符

如果你想要对 protected/internal/private 声明生成文档, 以及如果你想要排除 public 声明, 只为 internal API 生成文档, 这个选项会很有用.

可以对每个包为单位进行配置.

默认值: DokkaConfiguration.Visibility.PUBLIC

reportUndocumented

是否对可见的、无文档的声明输出警告, 这是指经过 documentedVisibilities 和其他过滤器过滤之后, 需要输出文档, 但没有 KDocs 的声明.

这个设置可以与 failOnWarning 选项配合工作.

可以对每个包为单位进行配置.

默认值: false

skipEmptyPackages

是否跳过经各种过滤器过滤之后不包含可见声明的包.

例如, 如果 skipDeprecated 设置为 true, 而且你的包中只包含已废弃的声明, 那么这个包会被认为是空的.

默认值: true

skipDeprecated

是否对标注了 @Deprecated 注解的声明生成文档.

可以对每个包为单位进行配置.

默认值: false

suppressGeneratedFiles

是否对生成的文件生成文档/进行分析.

生成的文件(generated file), 是指出现在 {project}/{buildDir}/generated 目录下的那些文件.

如果设置为 true, 它的效果等于将这个目录下的所有文件添加到 suppressedFiles 选项, 因此你可以手动配置它.

默认值: true

jdkVersion

在为 Java 类型生成外部文档链接时使用的 JDK 版本.

例如, 如果你在某些 public 声明的签名中使用了 java.util.UUID, 而且这个选项设置为 8, Dokka 会为它生成一个指向 JDK 8 Javadocs (https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html) 的外部文档链接.

默认值: JDK 8

languageVersion

设置代码分析和 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 环境时使用的 Kotlin 语言版本 (兼容模式).

默认情况下, 会使用 Dokka 的内嵌编译器所能够使用的最新的语言版本.

apiVersion

设置代码分析和 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 环境时使用的 Kotlin API 版本 (兼容模式).

默认情况下, 从 languageVersion 推断得到.

noStdlibLink

是否生成指向 Kotlin 标准库的 API 参考文档的外部文档链接.

注意: 当 noStdLibLink 设置为 false 时, **会** 生成链接.

默认值: false

noJdkLink

是否生成指向 JDK 的 Javadoc 的外部文档链接.

JDK Javadoc 版本会通过 jdkVersion 选项决定.

注意: 当 noJdkLink 设置为 false 时, **会** 生成链接.

默认值: false

noAndroidSdkLink

是否生成指向 Android SDK API 参考文档的外部文档链接.

这个选项只用于 Android 项目, 其它情况会被忽略.

注意: 当 noAndroidSdkLink 设置为 false 时, **会** 生成链接.

默认值: false

includes

包含 模块和包文档 (模块文档) 的 Markdown 文件列表.

指定的文件的内容会被解析, 并嵌入到文档内, 作为模块和包的描述文档.

关于这个选项的使用方法, 请参见 Dokka Gradle 示例 (https://github.com/Kotlin/dokka/tree/master/examples/gradle/dokka-gradle-example)

platform

设置代码分析和 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 环境时使用的平台.

默认值通从 Kotlin Gradle plugin 提供的信息推断得到.

sourceRoots

需要分析并生成文档的源代码根目录. 允许的输入是目录和单独的 .kt/.java 文件.

默认情况下, 源代码根目录从 Kotlin Gradle plugin 提供的信息推断得到.

classpath

用于代码分析和交互式示例的类路径.

如果来自依赖项的某些类型无法自动的解析/查找, 这个选项会很有用.

这个选项可以接受 .jar 和 .klib 文件.

默认情况下, 类路径从 Kotlin Gradle plugin 提供的信息推断得到.

samples

目录或文件的列表, 其中包含通过 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") KDoc tag 引用的示例函数.

源代码链接配置

`sourceLinks` 配置块可以用来为每个签名添加 `source` 链接, 链接地址是带有特定代码行的 `remoteUrl`. (代码行可以通过 `remoteLineSuffix` 进行配置).

这样可以帮助读者找到每个声明的源代码.

例如, 请参见 `kotlinx.coroutines` 中 `count()` (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/count.html) 函数的文档.

Kotlin

```
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//       参见本文档的 "配置示例" 小节.
tasks.withType<DokkaTask>().configureEach {
    // ...
    // 参见本文档的 "一般配置" 小节
    // ...

    dokkaSourceSets.configureEach {
        // ...
        // 参见本文档的 "源代码集配置" 小节
        // ...

        sourceLink {
            localDirectory.set(projectDir.resolve("src"))

remoteUrl.set(URL("https://github.com/kotlin/dokka/tree/master/s
rc"))
            remoteLineSuffix.set("#L")
        }
    }
}
```

Groovy

```
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL
```

```
// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//      参见本文档的 "配置示例" 小节.
tasks.withType(DokkaTask.class) {
    // ...
    // 参见本文档的 "一般配置" 小节
    // ...

    dokkaSourceSets.configureEach {
        // ...
        // 参见本文档的 "源代码集配置" 小节
        // ...

        sourceLink {
            localDirectory.set(file("src"))
            remoteUrl.set(new
URL("https://github.com/kotlin/dokka/tree/master/src"))
            remoteLineSuffix.set("#L")
        }
    }
}
```

localDirectory

本地源代码目录的路径. 必须是从当前项目根目录开始的相对路径.

remoteUrl

可以由文档读者访问的源代码托管服务 URL, 例如 GitHub, GitLab, Bitbucket, 等等. 这个 URL 用来生成声明的源代码链接.

remoteLineSuffix

向 URL 添加的源代码行数后缀. 这样可以帮助读者, 不仅能够导航到文件, 而且是声明所在的确定的行数.

行数本身会添加到后缀之后. 例如, 如果这个选项设置为 #L, 行数是 10, 那么最后的 URL 后缀会是 #L10.

各种常用的源代码托管服务的行数后缀是:

- GitHub: #L
- GitLab: #L
- Bitbucket: #lines-

默认值: #L

包选项

`perPackageOption` 配置块可以对指定的包设置一些选项, 包通过 `matchingRegex` 来匹配.

Kotlin

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//      参见本文档的 "配置示例" 小节.
tasks.withType<DokkaTask>().configureEach {
    // ...
    // 参见本文档的 "一般配置" 小节
    // ...

    dokkaSourceSets.configureEach {
        // ...
        // 参见本文档的 "源代码集配置" 小节
        // ...

        perPackageOption {
            matchingRegex.set(".*api.*")
            suppress.set(false)
            skipDeprecated.set(false)
            reportUndocumented.set(false)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
```

```
        }
    }
}
```

Groovy

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//       参见本文档的 "配置示例" 小节.
tasks.withType(DokkaTask.class) {
    // ...
    // 参见本文档的 "一般配置" 小节
    // ...

    dokkaSourceSets.configureEach {
        // ...
        // 参见本文档的 "源代码集配置" 小节
        // ...

        perPackageOption {
            matchingRegex.set(".*api.*")
            suppress.set(false)
            skipDeprecated.set(false)
            reportUndocumented.set(false)
            documentedVisibilities.set([Visibility.PUBLIC])
        }
    }
}
```

matchingRegex

用来匹配包的正规表达式.

默认值: .*

suppress

在生成文档时, 是否应该跳过这个包.

默认值: false

skipDeprecated

是否对标注了 @Deprecated 注解的声明生成文档.

这个选项可以在源代码集级配置.

默认值: false

reportUndocumented

是否对可见的、无文档的声明输出警告, 这是指经过 documentedVisibilities 和其他过滤器过滤之后, 需要输出文档, 但没有 KDocs 的声明.

这个设置可以与 failOnWarning 选项配合工作.

这个选项可以在源代码集级配置.

默认值: false

documentedVisibilities

应该生成文档的可见度标识符集合.

如果你想要对这个包内的 protected/internal/private 声明生成文档, 以及如果你想要排除 public 声明, 只为 internal API 生成文档, 这个选项可以很有用.

这个选项可以在源代码集级配置.

默认值: DokkaConfiguration.Visibility.PUBLIC

外部文档链接配置

`externalDocumentationLink` 配置块可以创建链接, 指向你的依赖项的外部文档.

例如, 如果你使用来自 `kotlinx.serialization` 的类型, 默认情况下, 在你的文档中这些类型不是可点击的链接, 因为无法解析这些类型. 但是, 由于 `kotlinx.serialization` 的 API 参考文档是使用 Dokka 构建的, 而且 发布到了 kotlinlang.org (https://kotlinlang.org/api/kotlinx.serialization/), 因此你可以对它配置外部文档链接. 然后就可以让 Dokka 对来自这个库的类型生成链接, 让它们成功的解析, 并在文档中成为可点击的链接.

默认情况下, 已经配置了对 Kotlin 标准库, JDK, Android SDK 和 AndroidX 的外部文档链接.

Kotlin

```
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//       参见本文档的 "配置示例" 小节.
tasks.withType<DokkaTask>().configureEach {
    // ...
    // 参见本文档的 "一般配置" 小节
    // ...

    dokkaSourceSets.configureEach {
        // ...
        // 参见本文档的 "源代码集配置" 小节
        // ...

        externalDocumentationLink {

url.set(URL("https://kotlinlang.org/api/kotlinx.serialization/"))
)
            packageListUrl.set(

rootProject.projectDir.resolve("serialization.package.list").toU
RL()
            )
        }
    }
}
```

Groovy

```
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//      参见本文档的 "配置示例" 小节.
tasks.withType(DokkaTask.class) {
    // ...
    // 参见本文档的 "一般配置" 小节
    // ...

    dokkaSourceSets.configureEach {
        // ...
        // 参见本文档的 "源代码集配置" 小节
        // ...

        externalDocumentationLink {
            url.set(new
URL("https://kotlinlang.org/api/kotlinx.serialization/"))
            packageListUrl.set(
                file("serialization.package.list").toURL()
            )
        }
    }
}
```

url

链接到的文档的根 URL. 最末尾 **必须** 包含斜线.

Dokka 会尽量对给定的 URL 自动寻找 package-list, 并将声明链接到一起.

如果自动解析失败, 或者如果你想要使用本地缓存的文件, 请考虑设置 packageListUrl 选项.

packageListUrl

package-list 的确切位置. 这是对 Dokka 自动解析的一个替代手段.

包列表包含关于文档和项目自身的信息, 例如模块和包的名称.

也可以使用本地缓存的文件, 以避免发生网络访问.

完整的配置

下面的例子中, 你可以看到同时使用了所有的配置选项.

Kotlin

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//      参见本文档的 "配置示例" 小节.
tasks.withType<DokkaTask>().configureEach {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())

outputDirectory.set(layout.buildDirectory.dir("dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    dokkaSourceSets {
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
```

```
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md",
"extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(),
file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt",
"samples/Advanced.kt")

            sourceLink {
                localDirectory.set(projectDir.resolve("src"))

remoteUrl.set(URL("https://github.com/kotlin/dokka/tree/master/s
rc"))
                remoteLineSuffix.set("#L")
            }

            externalDocumentationLink {

url.set(URL("https://kotlinlang.org/api/latest/jvm/stdlib/"))
                packageListUrl.set(

rootProject.projectDir.resolve("stdlib.package.list").toURL()
                )
            }

            perPackageOption {
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
```

```
                    documentedVisibilities.set(
                        setOf(
                            Visibility.PUBLIC,
                            Visibility.PRIVATE,
                            Visibility.PROTECTED,
                            Visibility.INTERNAL,
                            Visibility.PACKAGE
                        )
                    )
                }
            }
        }
}
```

Groovy

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// 注意: 要配置多项目构建, 你需要配置子项目的 Partial task.
//       参见本文档的 "配置示例" 小节.
tasks.withType(DokkaTask.class) {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(file("build/dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    dokkaSourceSets {
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
```

```
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set([Visibility.PUBLIC])
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md",
"extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(),
file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt",
"samples/Advanced.kt")

            sourceLink {
                localDirectory.set(file("src"))
                remoteUrl.set(new
URL("https://github.com/kotlin/dokka/tree/master/src"))
                remoteLineSuffix.set("#L")
            }

            externalDocumentationLink {
                url.set(new
URL("https://kotlinlang.org/api/latest/jvm/stdlib/"))
                packageListUrl.set(
                        file("stdlib.package.list").toURL()
                )
            }

            perPackageOption {
```

```
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
                documentedVisibilities.set([Visibility.PUBLIC])
            }
        }
    }
}
```

Maven

最终更新: 2024/09/10

要为基于 Maven 的项目生成文档, 你可以使用 Maven plugin for Dokka.

> 与 Gradle plugin for Dokka (Gradle) 相比, Maven plugin 只包括基本的功能, 不支持多模块构建.

你可以访问我们的 Maven 示例 (https://github.com/Kotlin/dokka/tree/{{ site.data.releases.dokkaVersion }}/examples/maven) 项目, 实际接触一下 Dokka, 看看它如何对 Maven 项目进行配置.

应用 Dokka

要应用 Dokka, 你需要在你的 POM 文件的 `plugins` 部分添加 `dokka-maven-plugin`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.dokka</groupId>
            <artifactId>dokka-maven-plugin</artifactId>
            <version>1.9.20</version>
            <executions>
                <execution>
                    <phase>pre-site</phase>
                    <goals>
                        <goal>dokka</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

生成文档

Maven plugin 提供了以下 goal:

Goal	描述
dokka:dokka	通过应用的 Dokka plugin 生成文档. 默认使用 HTML (HTML) 格式.

试验性功能

Goal	描述
dokka:javadoc	生成文档, 使用 Javadoc (Javadoc) 格式.
dokka:javadocJar	生成包含文档的 javadoc.jar 文件, 使用 Javadoc (Javadoc) 格式.

其他输出格式

Maven plugin for Dokka 默认使用 HTML (HTML) 输出格式构建文档.

其他所有输出格式都以 Dokka plugin (Dokka Plugin) 的形式实现. 要使用你需要的格式来生成文档, 你需要在配置中以 Dokka plugin 的形式添加这种格式.

例如, 要使用试验性的 GFM ("GFM" in "Markdown") 格式, 你需要添加 gfm-plugin artifact:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>gfm-plugin</artifactId>
                <version>1.9.20</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

使用这个配置, 运行 dokka:dokka goal 会生成 GFM 格式的文档.

关于 Dokka plugin, 更多详情请参见 Dokka plugin (Dokka Plugin).

构建 javadoc.jar

如果你想要将你的库发布到仓库, 你可能需要提供一个 `javadoc.jar` 文件, 其中包含你的库的 API 参考文档.

例如, 如果你想要发布到 Maven Central (https://central.sonatype.org/), 你 必须 (https://central.sonatype.org/publish/requirements/) 和你的项目一起提供一个 `javadoc.jar`. 但是, 并不是所有的仓库都有这样的规则.

与 Gradle plugin for Dokka ("构建 javadoc.jar" in "Gradle") 不同, Maven plugin 包括直接可以使用的 `dokka:javadocJar` goal. 默认情况下, 它使用 Javadoc (Javadoc) 输出格式, 在`target` 文件夹生成文档.

如果你对内建的 goal 不满意, 或者想要自定义输出 (例如, 你想要使用 HTML (HTML) 格式而不是 Javadoc 来生成文档), 可以使用以下配置添加 Maven JAR plugin:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.3.0</version>
    <executions>
        <execution>
            <goals>
                <goal>test-jar</goal>
            </goals>
        </execution>
        <execution>
            <id>dokka-jar</id>
            <phase>package</phase>
            <goals>
                <goal>jar</goal>
            </goals>
            <configuration>
                <classifier>dokka</classifier>

<classesDirectory>${project.build.directory}/dokka</classesDirectory
>
                <skipIfEmpty>true</skipIfEmpty>
```

```xml
                </configuration>
            </execution>
        </executions>
    </plugin>
```

文档, 以及包含文档的 `.jar` 包, 现在通过运行 `dokka:dokka` 和 `jar:jar@dokka-jar` goal 来生成:

```
mvn dokka:dokka jar:jar@dokka-jar
```

如果你将你的库发布到 Maven Central, 你可以使用 javadoc.io (https://javadoc.io/) 之类的服务, 免费托管你的库的 API 文档, 而且不需要任何设置. 它直接从 `javadoc.jar` 得到文档页面. 它可以很好的显示 HTML 格式文档, 参见 这个示例 (https://javadoc.io/doc/com.trib3/server/latest/index.html).

配置示例

Maven 的 plugin 配置代码库可以用来配置 Dokka.

下面是一个基本配置的示例, 它只修改你的文档的输出位置:

```xml
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>

<outputDir>${project.basedir}/target/documentation/dokka</outputDir>
    </configuration>
</plugin>
```

配置选项

Dokka 有很多配置选项, 可以用来定制你和你的读者的体验.

下面是对每个配置小节的一些示例和详细解释. 在本章的最后, 你还可以看到一个示例, 它使用了 所有的配置选项.

一般配置

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!-- ... -->
    <configuration>
        <skip>false</skip>
        <moduleName>${project.artifactId}</moduleName>

<outputDir>${project.basedir}/target/documentation</outputDir>
        <failOnWarning>false</failOnWarning>
        <suppressObviousFunctions>true</suppressObviousFunctions>
        <suppressInheritedMembers>false</suppressInheritedMembers>
        <offlineMode>false</offlineMode>
        <sourceDirectories>
            <dir>${project.basedir}/src</dir>
        </sourceDirectories>
        <documentedVisibilities>
            <visibility>PUBLIC</visibility>
            <visibility>PROTECTED</visibility>
        </documentedVisibilities>
        <reportUndocumented>false</reportUndocumented>
        <skipDeprecated>false</skipDeprecated>
        <skipEmptyPackages>true</skipEmptyPackages>
        <suppressedFiles>
            <file>/path/to/dir</file>
            <file>/path/to/file</file>
        </suppressedFiles>
        <jdkVersion>8</jdkVersion>
        <languageVersion>1.7</languageVersion>
        <apiVersion>1.7</apiVersion>
        <noStdlibLink>false</noStdlibLink>
        <noJdkLink>false</noJdkLink>
        <includes>
            <include>packages.md</include>
            <include>extra.md</include>
        </includes>
        <classpath>${project.compileClasspathElements}</classpath>
        <samples>
```

```
            <dir>${project.basedir}/samples</dir>
        </samples>
        <sourceLinks>
            <!-- 参见其它章节 -->
        </sourceLinks>
        <externalDocumentationLinks>
            <!-- 参见其它章节 -->
        </externalDocumentationLinks>
        <perPackageOptions>
            <!-- 参见其它章节 -->
        </perPackageOptions>
    </configuration>
</plugin>
```

skip

是否跳过文档生成.

默认值: false

moduleName

用来引用项目/模块的显示名称. 这个名称会用于目录, 导航, 日志, 等等.

默认值: {project.artifactId}

outputDir

文档生成的目录, 无论哪种格式.

默认值: {project.basedir}/target/dokka

failOnWarning

如果 Dokka 输出警告或错误, 是否让文档生成任务失败. 进程首先会等待所有的错误和警告输出完毕.

这个设置可以与 reportUndocumented 选项配合工作.

默认值: false

suppressObviousFunctions

是否禁止输出那些显而易见的函数.

满足以下条件的函数, 会被认为是显而易见的函数:

- 继承自 kotlin.Any, Kotlin.Enum, java.lang.Object 或 java.lang.Enum, 例如 equals, hashCode, toString.
- 合成(由编译器生成的)函数, 而且没有任何文档, 例如 dataClass.componentN 或 dataClass.copy.

默认值: true

suppressInheritedMembers

是否禁止输出在指定的类中继承得到的而且没有显式覆盖的成员.

注意: 这个选项可以禁止输出 equals/hashCode/toString 之类的函数, 但不能禁止输出 dataClass.componentN 和 dataClass.copy 之类的合成函数. 对合成函数, 请使用 suppressObviousFunctions 选项.

默认值: false

offlineMode

是否通过你的网络来解析远程的文件/链接.

包括用来生成外部文档链接的包列表. 例如, 可以让来自标准库的类成为文档中可以点击的链接.

将这个选项设置为 true, 某些情况下可以显著提高构建速度, 但也会降低文档质量和用户体验. 例如, 可以不解析来自你的依赖项的类/成员的链接, 包括标准库.

注意: 你可以将已取得的文件缓存到本地, 并通过本地路径提供给 Dokka. 参见 externalDocumentationLinks 小节.

默认值: false

sourceDirectories

需要分析并生成文档的源代码根目录. 允许的输入是目录和单独的 .kt/.java 文件.

默认值: {project.compileSourceRoots}

documentedVisibilities

这个选项设置需要生成文档的可见度修饰符.

如果你想要对 protected/internal/private 声明生成文档, 以及如果你想要排除 public 声明, 只为 internal API 生成文档, 这个选项会很有用.

可以对每个包为单位进行配置.

默认值: PUBLIC

reportUndocumented

是否对可见的、无文档的声明输出警告, 这是指经过 documentedVisibilities 和其他过滤器过滤之后, 需要输出文档, 但没有 KDocs 的声明.

这个设置可以与 failOnWarning 选项配合工作.

这个设置可以在包级覆盖.

默认值: false

skipDeprecated

是否对标注了 @Deprecated 注解的声明生成文档.

这个设置可以在包级覆盖.

默认值: false

skipEmptyPackages

是否跳过经各种过滤器过滤之后不包含可见声明的包.

例如, 如果 skipDeprecated 设置为 true, 而且你的包中只包含已废弃的声明, 那么这个包会被认为是空的.

默认值: true

suppressedFiles

需要禁止输出的目录或单独的文件, 意思是说, 对于来自这些目录和文件的声明, 不会生成文档.

jdkVersion

在为 Java 类型生成外部文档链接时使用的 JDK 版本.

例如, 如果你在某些 public 声明的签名中使用了 java.util.UUID, 而且这个选项设置为 8, Dokka 会为它生成一个指向 JDK 8 Javadocs (https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html) 的外部文档链接.

默认值: JDK 8

languageVersion

设置代码分析和 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 环境时使用的 Kotlin 语言版本 (兼容模式).

默认情况下, 会使用 Dokka 的内嵌编译器所能够使用的最新的语言版本.

apiVersion

设置代码分析和 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 环境时使用的 Kotlin API 版本 (兼容模式).

默认情况下, 从 languageVersion 推断得到.

noStdlibLink

是否生成指向 Kotlin 标准库的 API 参考文档的外部文档链接.

注意: 当 noStdLibLink 设置为 false 时, **会** 生成链接.

默认值: false

noJdkLink

是否生成指向 JDK 的 Javadoc 的外部文档链接.

JDK Javadoc 版本会通过 jdkVersion 选项决定.

注意: 当 noJdkLink 设置为 false 时, **会** 生成链接.

默认值: false

includes

包含 模块和包文档 (模块文档) 的 Markdown 文件列表.

指定的文件的内容会被解析, 并嵌入到文档内, 作为模块和包的描述文档.

classpath

用于代码分析和交互式示例的类路径.

如果来自依赖项的某些类型无法自动的解析/查找, 这个选项会很有用. 这个选项可以接受 .jar 和 .klib 文件.

默认值: {project.compileClasspathElements}

samples

目录或文件的列表, 其中包含通过 @sample KDoc tag. ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 引用的示例函数.

源代码链接配置

`sourceLinks` 配置块可以用来为每个签名添加 `source` 链接, 链接地址是带有特定代码行的 `url`. (代码行可以通过 `lineSuffix` 进行配置).

这样可以帮助读者找到每个声明的源代码.

例如, 请参见 `kotlinx.coroutines` 中 `count()` (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/count.html) 函数的文档.

```xml
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <sourceLinks>
            <link>
                <path>src</path>

<url>https://github.com/kotlin/dokka/tree/master/src</url>
                <lineSuffix>#L</lineSuffix>
            </link>
        </sourceLinks>
    </configuration>
</plugin>
```

path

本地源代码目录的路径. 必须是从当前模块根目录开始的相对路径.

注意: 只允许使用 Unix 风格路径, Windows 风格路径会产生错误.

url

可以由文档读者访问的源代码托管服务 URL, 例如 GitHub, GitLab, Bitbucket, 等等. 这个 URL 用来生成声明的源代码链接.

lineSuffix

向 URL 添加的源代码行数后缀. 这样可以帮助读者, 不仅能够导航到文件, 而且是声明所在的确定的行数.

行数本身会添加到后缀之后. 例如, 如果这个选项设置为 #L, 行数是 10, 那么最后的 URL 后缀会是 #L10.

各种常用的源代码托管服务的行数后缀是:

- GitHub: #L
- GitLab: #L
- Bitbucket: #lines-

外部文档链接配置

`externalDocumentationLinks` 配置块可以创建链接, 指向你的依赖项的外部文档.

例如, 如果你使用来自 `kotlinx.serialization` 的类型, 默认情况下, 在你的文档中这些类型不是可点击的链接, 因为无法解析这些类型. 但是, 由于 `kotlinx.serialization` 的 API 参考文档是使用 Dokka 构建的, 而且 发布到了 kotlinlang.org (https://kotlinlang.org/api/kotlinx.serialization/), 因此你可以对它配置外部文档链接. 然后就可以让 Dokka 对来自这个库的类型生成链接, 让它们成功的解析, 并在文档中成为可点击的链接.

默认情况下, 已经配置了对 Kotlin 标准库和 JDK 的外部文档链接.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!-- ... -->
    <configuration>
        <externalDocumentationLinks>
            <link>

<url>https://kotlinlang.org/api/kotlinx.serialization/</url>

<packageListUrl>file:/${project.basedir}/serialization.package.list</packageListUrl>
            </link>
        </externalDocumentationLinks>
    </configuration>
</plugin>
```

url

链接到的文档的根 URL. 最末尾 **必须** 包含斜线.

Dokka 会尽量对给定的 URL 自动寻找 package-list, 并将声明链接到一起.

如果自动解析失败, 或者如果你想要使用本地缓存的文件, 请考虑设置 packageListUrl 选项.

packageListUrl

package-list 的确切位置. 这是对 Dokka 自动解析的一个替代手段.

包列表包含关于文档和项目自身的信息, 例如模块和包的名称.

也可以使用本地缓存的文件, 以避免发生网络访问.

包选项

`perPackageOptions` 配置块可以对指定的包设置一些选项, 包通过 `matchingRegex` 来匹配.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!-- ... -->
    <configuration>
        <perPackageOptions>
            <packageOptions>
                <matchingRegex>.*api.*</matchingRegex>
                <suppress>false</suppress>
                <reportUndocumented>false</reportUndocumented>
                <skipDeprecated>false</skipDeprecated>
                <documentedVisibilities>
                    <visibility>PUBLIC</visibility>
                    <visibility>PRIVATE</visibility>
                    <visibility>PROTECTED</visibility>
                    <visibility>INTERNAL</visibility>
                    <visibility>PACKAGE</visibility>
                </documentedVisibilities>
            </packageOptions>
        </perPackageOptions>
    </configuration>
</plugin>
```

matchingRegex

用来匹配包的正规表达式.

默认值: .*

suppress

在生成文档时, 是否应该跳过这个包.

默认值: false

documentedVisibilities

应该生成文档的可见度标识符集合.

如果你想要对这个包内的 protected/internal/private 声明生成文档, 以及如果你想要排除 public 声明, 只为 internal API 生成文档, 这个选项可以很有用.

默认值: PUBLIC

skipDeprecated

是否对标注了 @Deprecated 注解的声明生成文档.

这个选项可以在项目/模块级设置.

默认值: false

reportUndocumented

是否对可见的、无文档的声明输出警告, 这是指经过 documentedVisibilities 和其他过滤器过滤之后, 需要输出文档, 但没有 KDocs 的声明.

这个设置可以与 failOnWarning 选项配合工作.

默认值: false

完整的配置

下面的例子中, 你可以看到同时使用了所有的配置选项.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <skip>false</skip>
        <moduleName>${project.artifactId}</moduleName>

<outputDir>${project.basedir}/target/documentation</outputDir>
        <failOnWarning>false</failOnWarning>
        <suppressObviousFunctions>true</suppressObviousFunctions>
        <suppressInheritedMembers>false</suppressInheritedMembers>
        <offlineMode>false</offlineMode>
        <sourceDirectories>
            <dir>src</dir>
        </sourceDirectories>
        <documentedVisibilities>
            <visibility>PUBLIC</visibility>
            <visibility>PRIVATE</visibility>
            <visibility>PROTECTED</visibility>
            <visibility>INTERNAL</visibility>
            <visibility>PACKAGE</visibility>
        </documentedVisibilities>
        <reportUndocumented>false</reportUndocumented>
        <skipDeprecated>false</skipDeprecated>
        <skipEmptyPackages>true</skipEmptyPackages>
        <suppressedFiles>
            <file>/path/to/dir</file>
            <file>/path/to/file</file>
        </suppressedFiles>
        <jdkVersion>8</jdkVersion>
        <languageVersion>1.7</languageVersion>
        <apiVersion>1.7</apiVersion>
        <noStdlibLink>false</noStdlibLink>
        <noJdkLink>false</noJdkLink>
        <includes>
            <include>packages.md</include>
            <include>extra.md</include>
```

```
        </includes>
        <classpath>${project.compileClasspathElements}</classpath>
        <samples>
            <dir>${project.basedir}/samples</dir>
        </samples>
        <sourceLinks>
            <link>
                <path>${project.basedir}/src</path>

<url>https://github.com/kotlin/dokka/tree/master/src</url>
                <lineSuffix>#L</lineSuffix>
            </link>
        </sourceLinks>
        <externalDocumentationLinks>
            <link>

<url>https://kotlinlang.org/api/latest/jvm/stdlib/</url>

<packageListUrl>file:/${project.basedir}/stdlib.package.list</packag
eListUrl>
            </link>
        </externalDocumentationLinks>
        <perPackageOptions>
            <packageOptions>
                <matchingRegex>.*api.*</matchingRegex>
                <suppress>false</suppress>
                <reportUndocumented>false</reportUndocumented>
                <skipDeprecated>false</skipDeprecated>
                <documentedVisibilities>
                    <visibility>PUBLIC</visibility>
                    <visibility>PRIVATE</visibility>
                    <visibility>PROTECTED</visibility>
                    <visibility>INTERNAL</visibility>
                    <visibility>PACKAGE</visibility>
                </documentedVisibilities>
            </packageOptions>
        </perPackageOptions>
```

```
    </configuration>
</plugin>
```

CLI

最终更新: 2024/09/10

如果你由于某些原因无法使用 Gradle (Gradle) 或 Maven (Maven) 构建工具, Dokka 有一个命令行(CLI)运行器用来生成文档.

与 Gradle plugin for Dokka 相比, 它拥有相同甚至更多的能力. 但它更加难于设置, 因为没有自动配置, 尤其是对于跨平台和多模块环境.

入门

CLI 运行器作为单独的可执行 artifact 发布到 Maven Central.

你可以在 Maven Central (https://central.sonatype.com/artifact/org.jetbrains.dokka/dokka-cli) 找到它, 或者 直接下载它 (https://repo1.maven.org/maven2/org/jetbrains/dokka/dokka-cli/1.9.20/dokka-cli-1.9.20.jar).

将 `dokka-cli-1.9.20.jar` 文件保存到你的计算机, 使用 `-help` 选项运行它, 可以看到所有的配置选项, 以及这些选项的描述:

```
java -jar dokka-cli-1.9.20.jar -help
```

也可以查看一些嵌套的选项, 例如 `-sourceSet`:

```
java -jar dokka-cli-1.9.20.jar -sourceSet -help
```

生成文档

前提条件

由于没有构建工具来管理依赖项, 你必须自己提供依赖项 `.jar` 文件.

对于所有的输出格式, 你所需要的依赖项如下:

Group	Artifact	版本	链接
org.jetbrains.dokka	dokka-base	1.9.20	下载 (https://repo1.maven.org/maven2/org/jetbrains/dokka/dokka-base/1.9.20/dokka-base-1.9.20.jar)
org.jetbrains.dokka	analysis-kotlin-descriptors	1.9.20	下载 (https://repo1.maven.org/maven2/org/jetbrains/dokka/analysis-kotlin-descriptors/1.9.20/analysis-kotlin-descriptors-1.9.20.jar)

对于 HTML (HTML) 输出格式, 你需要的额外的依赖项如下:

Group	Artifact	版本	链接
org.jetbrains.kotlinx	kotlinx-html-jvm	0.8.0	下载 (https://repo1.maven.org/maven2/org/jetbrains/kotlinx/kotlinx-html-jvm/0.8.0/kotlinx-html-jvm-0.8.0.jar)
org.freemarker	freemarker	2.3.31	下载 (https://repo1.maven.org/maven2/org/freemarker/freemarker/2.3.31/freemarker-2.3.31.jar)

使用命令行选项运行

你可以传递命令行选项来配置 CLI 运行器.

你至少需要提供以下选项:

- `-pluginsClasspath` - 指向已下载的依赖项的绝对/相对路径列表, 使用分号 `;` 分隔
- `-sourceSet` - 要生成文档的源代码的绝对路径
- `-outputDir` - 文档输出目录的绝对/相对路径

```
java -jar dokka-cli-1.9.20.jar \
     -pluginsClasspath "./dokka-base-1.9.20.jar;./analysis-kotlin-descriptors-1.9.20.jar;./kotlinx-html-jvm-0.8.0.jar;./freemarker-2.3.31.jar" \
```

```
    -sourceSet "-src /home/myCoolProject/src/main/kotlin" \
    -outputDir "./dokka/html"
```

执行上面示例中的命令, 会使用 HTML (HTML) 输出格式生成文档.

关于配置的更多详细信息, 请参见 命令行选项.

使用 JSON 配置运行

可以使用 JSON 来配置 CLI 运行器. 这种情况下, 你需要提供指向 JSON 配置文件的绝对/相对路径, 作为第一个也是唯一一个参数. 所有其它的配置选项都从 JSON 配置文件解析得到.

```
java -jar dokka-cli-1.9.20.jar dokka-configuration.json
```

你至少需要以下 JSON 配置文件:

```
{
  "outputDir": "./dokka/html",
  "sourceSets": [
    {
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "sourceRoots": [
        "/home/myCoolProject/src/main/kotlin"
      ]
    }
  ],
  "pluginsClasspath": [
    "./dokka-base-1.9.20.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./analysis-kotlin-descriptors-1.9.20.jar",
    "./freemarker-2.3.31.jar"
  ]
}
```

更多详情请参见 JSON 配置选项.

其它输出格式

默认情况下, `dokka-base` artifact 只包含 HTML (HTML) 输出格式.

其他所有输出格式都以 Dokka plugin (Dokka Plugin) 的形式实现. 要使用这些格式, 你需要将它们添加到 plugin classpath.

例如, 如果你想要使用试验性的 GFM ("GFM" in "Markdown") 输出格式生成文档, 你需要下载 gfm-plugin 的 JAR 文件 (下载 (https://repo1.maven.org/maven2/org/jetbrains/dokka/gfm-plugin/1.9.20/gfm-plugin-1.9.20.jar)), 并将它传递给 `pluginsClasspath` 配置选项.

通过命令行选项传递:

```
java -jar dokka-cli-1.9.20.jar \
     -pluginsClasspath "./dokka-base-1.9.20.jar;...;./gfm-plugin-1.9.20.jar" \
     ...
```

通过 JSON 配置传递:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.9.20.jar",
    "...",
    "./gfm-plugin-1.9.20.jar"
  ],
  ...
}
```

通过传递给 `pluginsClasspath` 的 GFM plugin, CLI 运行器会使用 GFM 输出格式生成文档.

更多详情, 请参见 Markdown (Markdown) 和 Javadoc ("生成 Javadoc 文档" in "Javadoc") 章节.

命令行选项

要查看所有可用的命令行选项列表, 以及它们的详细描述, 请运行:

```
java -jar dokka-cli-1.9.20.jar -help
```

简单的总结如下:

选项	描述
moduleName	项目/模块名称.
moduleVersion	需要生成文档的版本.
outputDir	输出目录路径, 默认值为 ./dokka.
sourceSet	对 Dokka 源代码集的配置. 包含嵌套的配置选项.
pluginsConfiguration	对 Dokka plugin 的配置.
pluginsClasspath	Dokka plugin 以及它们的依赖项的 jar 文件列表. 可以接受多个路径, 以分号分隔.
offlineMode	是否通过网络来解析远程的文件/链接.
failOnWarning	如果 Dokka 输出警告或错误, 是否让文档生成任务失败.
delayTemplateSubstitution	是否延迟替换某些元素. 用于多模块项目的增量构建.
noSuppressObviousFunctions	是否禁止输出那些显而易见的函数, 例如继承自 kotlin.Any 和 java.lang.Object 的函数.
includes	包含模块和包的文档的 Markdown 文件. 可以接受多个值, 以分号分隔.
suppressInheritedMembers	是否禁止输出在指定的类中继承得到的而且没有显式覆盖的成员.
globalPackage	全局的包配置选项列表, 格式为 "matchingRegex,-deprecated,-privateApi,+warnUndocumented,+suppress;+visibility:PUBLIC;...". 可以接受多个

Options	值, 以分号分隔.
globalLinks	全局的外部文档链接, 格式为 {url}^{packageListUrl}. 可以接受多个值, 以 ^^ 分隔.
globalSrcLink	源代码目录与用于浏览源代码的 Web Service 之间的全局的对应. 可以接受多个路径, 以分号分隔.
helpSourceSet	对嵌套的 -sourceSet 配置输出帮助信息.
loggingLevel	日志级别, 可以设置的值: DEBUG, PROGRESS, INFO, WARN, ERROR.
help, h	关于使用方法的帮助信息.

源代码集选项

要查看嵌套的 -sourceSet 配置的命令行选项列表, 请运行:

```
java -jar dokka-cli-1.9.20.jar -sourceSet -help
```

简单的总结如下:

选项	描述
sourceSetName	源代码集名称.
displayName	源代码集的显示名称, 这个名称会在内部和外部使用.
classpath	对示例进行分析和交互时的类路径. 可以接受多个路径, 以分号分隔.
src	需要分析并生成文档的源代码根目录. 可以接受多个路径, 以分号分隔.
dependentSourceSets	依赖的源代码集名称, 格式为 moduleName/sourceSetName. 可以接受多个值, 以分号分隔.
samples	包含示例函数的目录或文件的列表. 可以接受多个路径, 以分号分隔.
includes	包含 模块和包文档 (模块文档) 的 Markdown 文件. 可以接受多个路径, 以分号分隔.
documentedVisibilities	需要生成文档的成员可见度. 可以接受多个值, 以分号分隔. 可以设置的值: PUBLIC, PRIVATE, PROTECTED, INTERNAL, PACKAGE.
reportUndocumented	是否对无文档的声明输出警告.
noSkipEmptyPackages	是否对空的包创建页面.
skipDeprecated	是否跳过废弃的声明.
jdkVersion	生成 JDK Javadoc 链接时使用的 JDK 版本.
languageVersio	设置代码分析和示例环境时使用的 Kotlin 语言版本.

n	
apiVersion	设置代码分析和示例环境时使用的 Kotlin API 版本.
noStdlibLink	是否生成指向 Kotlin 标准库的链接.
noJdkLink	是否生成指向 JDK Javadoc 的链接.
suppressedFiles	需要禁止输出的文件路径. 可以接受多个路径, 以分号分隔.
analysisPlatform	设置代码分析环境时使用的平台.
perPackageOptions	包源代码集配置列表, 格式为 `matchingRegexp,-deprecated,-privateApi,+warnUndocumented,+suppress;...`. 可以接受多个值, 以分号分隔.
externalDocumentationLinks	外部文档链接, 格式为 `{url}^{packageListUrl}`. 可以接受多个值, 以 `^^` 分隔.
srcLink	源代码目录与用于浏览源代码的 Web Service 之间的对应. 可以接受多个路径, 以分号分隔.

JSON 配置

下面是对每个配置小节的一些示例和详细解释. 在本章的最后, 你还可以看到一个示例, 它使用了 所有的配置选项.

一般配置

```
{
  "moduleName": "Dokka Example",
  "moduleVersion": null,
  "outputDir": "./build/dokka/html",
  "failOnWarning": false,
  "suppressObviousFunctions": true,
```

```
  "suppressInheritedMembers": false,
  "offlineMode": false,
  "includes": [
    "module.md"
  ],
  "sourceLinks":  [
    { "_comment": "参见其它章节" }
  ],
  "perPackageOptions": [
    { "_comment": "参见其它章节" }
  ],
  "externalDocumentationLinks":  [
    { "_comment": "参见其它章节" }
  ],
  "sourceSets": [
    { "_comment": "参见其它章节" }
  ],
  "pluginsClasspath": [
    "./dokka-base-1.9.20.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./analysis-kotlin-descriptors-1.9.20.jar",
    "./freemarker-2.3.31.jar"
  ]
}
```

moduleName

用来引用模块的显示名称. 这个名称会用于目录, 导航, 日志, 等等.

默认值: root

moduleVersion

模块版本.

默认值: 空

outputDirectory

文档生成的目录, 无论哪种格式.

默认值: ./dokka

failOnWarning

如果 Dokka 输出警告或错误, 是否让文档生成任务失败. 进程首先会等待所有的错误和警告输出完毕.

这个设置可以与 reportUndocumented 选项配合工作.

默认值: false

suppressObviousFunctions

是否禁止输出那些显而易见的函数.

满足以下条件的函数, 会被认为是显而易见的函数:

- 继承自 kotlin.Any, Kotlin.Enum, java.lang.Object 或 java.lang.Enum, 例如 equals, hashCode, toString.
- 合成(由编译器生成的)函数, 而且没有任何文档, 例如 dataClass.componentN 或 dataClass.copy.

默认值: true

suppressInheritedMembers

是否禁止输出在指定的类中继承得到的而且没有显式覆盖的成员.

注意: 这个选项可以禁止输出 equals/hashCode/toString 之类的函数, 但不能禁止输出 dataClass.componentN 和 dataClass.copy 之类的合成函数. 对合成函数, 请使用 suppressObviousFunctions.

默认值: false

offlineMode

是否通过你的网络来解析远程的文件/链接.

包括用来生成外部文档链接的包列表. 例如, 可以让来自标准库的类成为文档中可以点击的链接.

将这个选项设置为 true, 某些情况下可以显著提高构建速度, 但也会降低文档质量和用户体验. 例如, 可以不解析来自你的依赖项的类/成员的链接, 包括标准库.

注意: 你可以将已取得的文件缓存到本地, 并通过本地路径提供给 Dokka. 参见 externalDocumentationLinks 小节.

默认值: false

includes

包含 模块和包文档 (模块文档) 的 Markdown 文件列表.

指定的文件的内容会被解析, 并嵌入到文档内, 作为模块和包的描述文档.

这个选项可以对每个包为单位进行配置.

sourceSets

对 Kotlin 源代码集 ("源代码集(Source Set)" in "Kotlin Multiplatform 项目结构的基础知识") 的额外配置.

关于它的所有选项的列表, 请参见 源代码集配置.

sourceLinks

源代码链接的全局配置, 应用于所有的源代码集.

关于它的所有选项的列表, 请参见 源代码链接配置.

perPackageOptions

对匹配的包的全局配置, 不论它们属于哪个源代码集.

关于它的所有选项的列表, 请参见 包配置.

externalDocumentationLinks

外部文档链接的全局配置, 不论它们属于哪个源代码集.

关于它的所有选项的列表, 请参见 外部文档链接配置.

pluginsClasspath

Dokka plugin 以及它们的依赖项的 JAR 文件列表.

源代码集配置

如何配置 Kotlin 源代码集 ("源代码集(Source Set)" in "Kotlin Multiplatform 项目结构的基础知识"):

```
{
  "sourceSets": [
    {
      "displayName": "jvm",
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "dependentSourceSets": [
        {
          "scopeId": "dependentSourceSetScopeId",
          "sourceSetName": "dependentSourceSetName"
        }
      ],
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED",
"INTERNAL", "PACKAGE"],
      "reportUndocumented": false,
      "skipEmptyPackages": true,
      "skipDeprecated": false,
      "jdkVersion": 8,
      "languageVersion": "1.7",
      "apiVersion": "1.7",
      "noStdlibLink": false,
```

```
      "noJdkLink": false,
      "includes": [
        "module.md"
      ],
      "analysisPlatform": "jvm",
      "sourceRoots": [
        "/home/ignat/IdeaProjects/dokka-debug-mvn/src/main/kotlin"
      ],
      "classpath": [
        "libs/kotlin-stdlib-1.9.23.jar",
        "libs/kotlin-stdlib-common-1.9.23.jar"
      ],
      "samples": [
        "samples/basic.kt"
      ],
      "suppressedFiles": [
        "src/main/kotlin/org/jetbrains/dokka/Suppressed.kt"
      ],
      "sourceLinks":  [
        { "_comment": "参见其它章节" }
      ],
      "perPackageOptions": [
        { "_comment": "参见其它章节" }
      ],
      "externalDocumentationLinks":  [
        { "_comment": "参见其它章节" }
      ]
    }
  ]
}
```

displayName

用来引用这个源代码集的显示名称.

这个名称在外部用途使用(例如, 源代码集名称会显示给文档读者), 也在内部使用(例如, 用于 reportUndocumented 的日志信息).

如果你没有更好的选择, 可以使用平台名称.

sourceSetID

源代码集的技术性 ID

documentedVisibilities

需要生成文档的可见度修饰符集合.

如果你想要对 protected/internal/private 声明声明生成文档, 以及如果你想要排除 public 声明, 只为 internal API 生成文档, 这个选项会很有用.

这个选项可以对每个包为单位进行配置.

可以设置的值:

- PUBLIC
- PRIVATE
- PROTECTED
- INTERNAL
- PACKAGE

默认值: PUBLIC

reportUndocumented

是否对可见的、无文档的声明输出警告, 这是指经过 documentedVisibilities 和其他过滤器过滤之后, 需要输出文档, 但没有 KDocs 的声明.

这个设置可以与 failOnWarning 选项配合工作.

这个选项可以对每个包为单位进行配置.

默认值: false

skipEmptyPackages

是否跳过经各种过滤器过滤之后不包含可见声明的包.

例如, 如果 skipDeprecated 设置为 true, 而且你的包中只包含已废弃的声明, 那么这个包会被认为是空的.

对 CLI 运行器的默认值为 false.

skipDeprecated

是否对标注了 @Deprecated 注解的声明生成文档.

这个选项可以对每个包为单位进行配置.

默认值: false

jdkVersion

在为 Java 类型生成外部文档链接时使用的 JDK 版本.

例如, 如果你在某些 public 声明的签名中使用了 java.util.UUID, 而且这个选项设置为 8, Dokka 会为它生成一个指向 JDK 8 Javadocs (https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html) 的外部文档链接.

languageVersion

设置代码分析和 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 环境时使用的 Kotlin 语言版本 (兼容模式).

apiVersion

设置代码分析和 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 环境时使用的 Kotlin API 版本 (兼容模式).

noStdlibLink

是否生成指向 Kotlin 标准库的 API 参考文档的外部文档链接.

注意: 当 noStdLibLink 设置为 false 时, **会** 生成链接.

默认值: false

noJdkLink

是否生成指向 JDK 的 Javadoc 的外部文档链接.

JDK Javadoc 版本会通过 jdkVersion 选项决定.

注意: 当 noJdkLink 设置为 false 时, **会** 生成链接.

默认值: false

includes

包含 模块和包文档 (模块文档) 的 Markdown 文件列表.

指定的文件的内容会被解析, 并嵌入到文档内, 作为模块和包的描述文档.

analysisPlatform

设置代码分析和 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") 环境时使用的平台.

可以设置的值:

- jvm
- common
- js
- native

sourceRoots

需要分析并生成文档的源代码根目录. 允许的输入是目录和单独的 .kt/.java 文件.

classpath

用于代码分析和交互式示例的类路径.

如果来自依赖项的某些类型无法自动的解析/查找, 这个选项会很有用.

这个选项可以接受 .jar 和 .klib 文件.

samples

目录或文件的列表, 其中包含通过 @sample ("@sample identifier" in "为 Kotlin 代码编写文档: KDoc") KDoc tag 引用的示例函数.

suppressedFiles

需要禁止生成文档的文件.

sourceLinks

源代码链接的一组参数, 只应用于这个源代码集.

关于它的所有选项的列表, 请参见 源代码链接配置.

perPackageOptions

一组参数, 应用于这个源代码集之内匹配的包.

关于它的所有选项的列表, 请参见 各包配置.

externalDocumentationLinks

外部文档链接的一组参数, 只应用于这个源代码集.

关于它的所有选项的列表, 请参见 外部文档链接配置.

源代码链接配置

`sourceLinks` 配置块可以用来为每个签名添加 `source` 链接, 链接地址是带有特定代码行的 `remoteUrl`. (代码行可以通过 `remoteLineSuffix` 进行配置).

这样可以帮助读者找到每个声明的源代码.

例如, 请参见 `kotlinx.coroutines` 中 `count()` (https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/count.html) 函数的文档 .

你可以对所有的源代码集一起配置源代码链接, 也可以 分别配置:

```
{
  "sourceLinks": [
    {
      "localDirectory": "src/main/kotlin",
      "remoteUrl":
"https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
      "remoteLineSuffix": "#L"
    }
  ]
}
```

localDirectory

本地源代码目录的路径.

remoteUrl

可以由文档读者访问的源代码托管服务 URL, 例如 GitHub, GitLab, Bitbucket, 等等. 这个 URL 用来生成声明的源代码链接.

remoteLineSuffix

向 URL 添加的源代码行数后缀. 这样可以帮助读者, 不仅能够导航到文件, 而且是声明所在的确定的行数.

行数本身会添加到后缀之后. 例如, 如果这个选项设置为 `#L`, 行数是 10, 那么最后的 URL 后缀会是 `#L10`.

各种常用的源代码托管服务的行数后缀是:

- GitHub: #L
- GitLab: #L
- Bitbucket: #lines-

默认值: 空 (没有后缀)

各包配置

`perPackageOptions` 配置块可以对指定的包设置一些选项, 包通过 `matchingRegex` 来匹配.

你可以对所有的源代码集一起添加包配置, 也可以 分别配置:

```
{
  "perPackageOptions": [
    {
      "matchingRegex": ".*internal.*",
      "suppress": false,
      "skipDeprecated": false,
      "reportUndocumented": false,
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED",
"INTERNAL", "PACKAGE"]
    }
  ]
}
```

matchingRegex

用来匹配包的正规表达式.

suppress

在生成文档时, 是否应该跳过这个包.

默认值: false

skipDeprecated

是否对标注了 @Deprecated 注解的声明生成文档.

这个选项可以在项目/模块级设置.

默认值: false

reportUndocumented

是否对可见的、无文档的声明输出警告, 这是指经过 documentedVisibilities 和其他过滤器过滤之后, 需要输出文档, 但没有 KDocs 的声明.

这个设置可以与 failOnWarning 选项配合工作.

这个选项可以在源代码集级设置.

默认值: false

documentedVisibilities

这个选项设置需要生成文档的可见度修饰符.

如果你想要对这个包内的 protected/internal/private 声明生成文档, 以及如果你想要排除 public 声明, 只为 internal API 生成文档, 这个选项会很有用.

这个选项可以在源代码集级设置.

默认值: PUBLIC

外部文档链接配置

`externalDocumentationLinks` 配置块可以创建链接, 指向你的依赖项的外部文档.

例如, 如果你使用来自 `kotlinx.serialization` 的类型, 默认情况下, 在你的文档中这些类型不是可点击的链接, 因为无法解析这些类型. 但是, 由于 `kotlinx.serialization` 的 API 参考文档是使用 Dokka 构建的, 而且 发布到了 kotlinlang.org (https://kotlinlang.org/api/kotlinx.serialization/), 因此你可以对它配置外部文档链接. 然后就可以让 Dokka 对来自这个库的类型生成链接, 让它们成功的解析, 并在文档中成为可点击的链接.

你可以对所有的源代码集一起配置外部文档链接, 也可以 分别配置:

```
{
  "externalDocumentationLinks": [
    {
      "url": "https://kotlinlang.org/api/kotlinx.serialization/",
      "packageListUrl":
"https://kotlinlang.org/api/kotlinx.serialization/package-list"
    }
  ]
}
```

url

链接到的文档的根 URL. 最末尾 **必须** 包含斜线.

Dokka 会尽量对给定的 URL 自动寻找 package-list, 并将声明链接到一起.

如果自动解析失败, 或者如果你想要使用本地缓存的文件, 请考虑设置 packageListUrl 选项.

packageListUrl

package-list 的确切位置. 这是对 Dokka 自动解析的一个替代手段.

包列表包含关于文档和项目自身的信息, 例如模块和包的名称.

也可以使用本地缓存的文件, 以避免发生网络访问.

完整的配置

下面的例子中, 你可以看到同时使用了所有的配置选项.

```
{
  "moduleName": "Dokka Example",
  "moduleVersion": null,
  "outputDir": "./build/dokka/html",
  "failOnWarning": false,
  "suppressObviousFunctions": true,
  "suppressInheritedMembers": false,
  "offlineMode": false,
  "sourceLinks": [
```

```
    {
      "localDirectory": "src/main/kotlin",
      "remoteUrl":
"https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
      "remoteLineSuffix": "#L"
    }
  ],
  "externalDocumentationLinks": [
    {
      "url": "https://docs.oracle.com/javase/8/docs/api/",
      "packageListUrl":
"https://docs.oracle.com/javase/8/docs/api/package-list"
    },
    {
      "url": "https://kotlinlang.org/api/latest/jvm/stdlib/",
      "packageListUrl":
"https://kotlinlang.org/api/latest/jvm/stdlib/package-list"
    }
  ],
  "perPackageOptions": [
    {
      "matchingRegex": ".*internal.*",
      "suppress": false,
      "reportUndocumented": false,
      "skipDeprecated": false,
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED",
"INTERNAL", "PACKAGE"]
    }
  ],
  "sourceSets": [
    {
      "displayName": "jvm",
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "dependentSourceSets": [
        {
          "scopeId": "dependentSourceSetScopeId",
```

```
          "sourceSetName": "dependentSourceSetName"
        }
      ],
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED",
"INTERNAL", "PACKAGE"],
      "reportUndocumented": false,
      "skipEmptyPackages": true,
      "skipDeprecated": false,
      "jdkVersion": 8,
      "languageVersion": "1.7",
      "apiVersion": "1.7",
      "noStdlibLink": false,
      "noJdkLink": false,
      "includes": [
        "module.md"
      ],
      "analysisPlatform": "jvm",
      "sourceRoots": [
        "/home/ignat/IdeaProjects/dokka-debug-mvn/src/main/kotlin"
      ],
      "classpath": [
        "libs/kotlin-stdlib-1.9.23.jar",
        "libs/kotlin-stdlib-common-1.9.23.jar"
      ],
      "samples": [
        "samples/basic.kt"
      ],
      "suppressedFiles": [
        "src/main/kotlin/org/jetbrains/dokka/Suppressed.kt"
      ],
      "sourceLinks": [
        {
          "localDirectory": "src/main/kotlin",
          "remoteUrl":
"https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
          "remoteLineSuffix": "#L"
        }
      ],
      "externalDocumentationLinks": [
```

```
        {
          "url": "https://docs.oracle.com/javase/8/docs/api/",
          "packageListUrl":
"https://docs.oracle.com/javase/8/docs/api/package-list"
        },
        {
          "url": "https://kotlinlang.org/api/latest/jvm/stdlib/",
          "packageListUrl":
"https://kotlinlang.org/api/latest/jvm/stdlib/package-list"
        }
      ],
      "perPackageOptions": [
        {
          "matchingRegex": ".*internal.*",
          "suppress": false,
          "reportUndocumented": false,
          "skipDeprecated": false,
          "documentedVisibilities": ["PUBLIC", "PRIVATE",
"PROTECTED", "INTERNAL", "PACKAGE"]
        }
      ]
    }
  ],
  "pluginsClasspath": [
    "./dokka-base-1.9.20.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./analysis-kotlin-descriptors-1.9.20.jar",
    "./freemarker-2.3.31.jar"
  ],
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "
{\"separateInheritedMembers\":false,\"footerMessage\":\"© 2021
pretty good Copyright\"}"
    }
  ],
  "includes": [
```

```
    "module.md"
  ]
}
```

HTML

最终更新: 2024/09/10

HTML 是 Dokka 的默认并且推荐的输出格式. 目前处于 Beta 版, 正在接近稳定发布版.

你可以浏览 kotlinx.coroutines (https://kotlinlang.org/api/kotlinx.coroutines/) 的文档, 看看 HTML 输出的示例.

生成 HTML 文档

所有的运行器都支持 HTML 输出格式. 要生成 HTML 文档, 请根据你的构建工具和运行器, 执行以下步骤:

- 对于 Gradle ("生成文档" in "Gradle"), 运行 `dokkaHtml` 或 `dokkaHtmlMultiModule` task.
- 对于 Maven ("生成文档" in "Maven"), 运行 `dokka:dokka` goal.
- 对于 CLI 运行器 ("生成文档" in "CLI"), 运行 HTML 依赖项集合.

> 这种格式生成的 HTML 页面, 需要托管在 Web 服务器上, 才能正常显示它的全部内容.
>
> 你可以使用任何免费的静态网站托管服务, 例如 GitHub Pages (https://docs.github.com/en/pages/getting-started-with-github-pages/about-github-pages).
>
> 在本地, 你可以使用 内建的 IntelliJ Web 服务器 (https://www.jetbrains.com/help/idea/php-built-in-web-server.html).

配置

HTML 格式是 Dokka 的基本格式, 因此它可以通过 `DokkaBase` 和 `DokkaBaseConfiguration` 类配置:

Kotlin

通过类型安全的 Kotlin DSL:

```
import org.jetbrains.dokka.base.DokkaBase
import org.jetbrains.dokka.gradle.DokkaTask
```

```kotlin
import org.jetbrains.dokka.base.DokkaBaseConfiguration

buildscript {
    dependencies {
        classpath("org.jetbrains.dokka:dokka-base:1.9.20")
    }
}

tasks.withType<DokkaTask>().configureEach {
    pluginConfiguration<DokkaBase, DokkaBaseConfiguration> {
        customAssets = listOf(file("my-image.png"))
        customStyleSheets = listOf(file("my-styles.css"))
        footerMessage = "(c) 2022 MyOrg"
        separateInheritedMembers = false
        templatesDir = file("dokka/templates")
        mergeImplicitExpectActualDeclarations = false
    }
}
```

通过 JSON:

```kotlin
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType<DokkaTask>().configureEach {
    val dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg",
      "separateInheritedMembers": false,
      "templatesDir": "${file("dokka/templates")}",
      "mergeImplicitExpectActualDeclarations": false
    }
    """
    pluginsMapConfiguration.set(
        mapOf(
            // plugin 的完整限定名称, to, json 配置
            "org.jetbrains.dokka.base.DokkaBase" to
dokkaBaseConfiguration
```

```
        )
    )
}
```

Groovy

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType(DokkaTask.class) {
    String dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
      "separateInheritedMembers": false,
      "templatesDir": "${file("dokka/templates")}",
      "mergeImplicitExpectActualDeclarations": false
    }
    """
    pluginsMapConfiguration.set(
        // plugin 的完整限定名称, :, json 配置
        ["org.jetbrains.dokka.base.DokkaBase":
dokkaBaseConfiguration]
    )
}
```

Maven

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <pluginsConfiguration>
            <!-- plugin 的完整限定名称 -->
            <org.jetbrains.dokka.base.DokkaBase>
```

```xml
                    <!-- 选项名称 -->
                    <customAssets>
                        <asset>${project.basedir}/my-
image.png</asset>
                    </customAssets>
                    <customStyleSheets>
                        <stylesheet>${project.basedir}/my-
styles.css</stylesheet>
                    </customStyleSheets>
                    <footerMessage>(c) MyOrg 2022
Maven</footerMessage>

<separateInheritedMembers>false</separateInheritedMembers>

<templatesDir>${project.basedir}/dokka/templates</templatesDir>

<mergeImplicitExpectActualDeclarations>false</mergeImplicitExpec
tActualDeclarations>
                </org.jetbrains.dokka.base.DokkaBase>
            </pluginsConfiguration>
        </configuration>
</plugin>
```

CLI

通过 命令行选项 ("使用命令行选项运行" in "CLI"):

```
java -jar dokka-cli-1.9.20.jar \
     ...
     -pluginsConfiguration "org.jetbrains.dokka.base.DokkaBase=
{\"customAssets\": [\"my-image.png\"], \"customStyleSheets\":
[\"my-styles.css\"], \"footerMessage\": \"(c) 2022 MyOrg\",
\"separateInheritedMembers\": false, \"templatesDir\":
\"dokka/templates\", \"mergeImplicitExpectActualDeclarations\":
false}
"
```

通过 JSON 配置 ("使用 JSON 配置运行" in "CLI"):

```json
{
  "moduleName": "Dokka Example",
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"customAssets\": [\"my-image.png\"],
\"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\":
\"(c) 2022 MyOrg\", \"separateInheritedMembers\": false,
\"templatesDir\": \"dokka/templates\",
\"mergeImplicitExpectActualDeclarations\": false}"
    }
  ]
}
```

配置选项

下表包括所有可以使用的配置选项, 以及它们的用途.

选项	描述
customAssets	要与文档绑定到一起的图片资源的路径列表. 图片资源可以使用任意的文件扩展名. 更多详情请参见 自定义资源.
customStyleSheets	要与文档绑定到一起并在显示时使用的 .css 样式表的路径列表. 更多详情请参见 自定义样式表.
templatesDir	包含自定义 HTML 模板的目录路径. 更多详情请参见 模板.
footerMessage	在页脚显示的文字.
separateInheritedMembers	这是一个 boolean 选项. 如果设置为 true, Dokka 会将属性/函数与继承的属性/继承的函数分开显示. 这个设置默认关闭.
mergeImplicitExpectActualDeclarations	这是一个 boolean 选项. 如果设置为 true, Dokka 合并那些没有声明为 expect/actual (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-connect-to-apis.html) 的声明, 但使用相同的完整限定名称. 这个设置对旧的代码库可能很有用. 这个设置默认关闭.

关于 Dokka plugin 配置的更多详情, 请参见 配置 Dokka plugins ("配置 Dokka plugin" in "Dokka Plugin").

自定义

为了帮助你为你的文档添加自己的外观和风格, HTML 格式支持很多的自定义选项.

自定义样式表

你可以使用 customStyleSheets 配置选项, 使用你自己的样式表. 这些配置会被应用于所有的页面.

可以通过提供相同名称的文件来覆盖 Dokka 的默认样式表:

样式表名称	描述
style.css	主样式表, 包含在所有页面中使用的大部分样式
logo-styles.css	页头 logo 样式
prism.css	用于 PrismJS (https://prismjs.com/) 语法高亮度显示器的样式

Dokka 所有样式表的源代码, 请参见 GitHub (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-base/src/main/resources/dokka/styles).

自定义资源

你可以使用 customAssets 配置选项, 提供你自己的绑定到文档的图片.

这些文件会被复制到 <output>/images 目录.

可以通过提供相同名称的文件来覆盖 Dokka 的图片和图标. 最重要的是 logo-icon.svg, 它是用于页头的图片. 其他主要是图标.

Dokka 使用的所有图片, 请参见 GitHub (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-base/src/main/resources/dokka/images).

修改 logo

要自定义 logo, 你可以从 提供你自己的 logo-icon.svg 资源 开始.

如果你不喜欢它的外观, 或者你想要使用 .png 文件, 而不是默认的 .svg 文件, 你可以 覆盖 logo-styles.css 样式表 来定制它.

具体的例子, 请参见我们的 自定义格式的示例项目 (https://github.com/Kotlin/dokka/tree/1.9.20/examples/gradle/dokka-customFormat-example).

修改页脚

你可以使用 footerMessage 配置选项, 修改页脚中的文字.

模板

Dokka 提供了修改用于生成文档页面的 FreeMarker (https://freemarker.apache.org/) 模板的能力.

你可以完全彻底的修改页头, 添加你的自己的横幅/菜单/搜索, 负载析, 修改页面 body 样式, 等等等等.

Dokka 使用以下模板:

模板	描述
base.ftl	定义显示的所有页面的通常设计.
includes/header.ftl	页头, 默认包含 logo, 版本, 源代码集选择器, 浅色/深色主题切换, 以及搜索.
includes/footer.ftl	页脚, 包含 footerMessage 配置选项, 以及版权信息.
includes/page_metadata.ftl	在 <head> 容器内使用的元数据.
includes/source_set_selector.ftl	页头中的 源代码集 ("源代码集(Source Set)" in "Kotlin Multiplatform 项目结构的基础知识") 选择器.

基础模板是 base.ftl, 它引入(include)其它所有模板. Dokka 所有模板的源代码请参见 GitHub (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-base/src/main/resources/dokka/templates).

你可以使用 templatesDir 配置选项 覆盖任何一个模板. Dokka 会在指定的目录搜索指定的模板名称. 如果无法找到用户定义的模板, 它会使用默认模板.

变量

下表是在所有模板内可以使用的变量:

变量	描述
${pageName}	页面名称
${footerMessage}	footerMessage 配置选项 设置的文字
${sourceSets}	用于对跨平台页面的 源代码集 ("源代码集(Source Set)" in "Kotlin Multiplatform 项目结构的基础知识") List, 可为 null. List 中的每个元素包含 name, platform, 和 filter 属性.
${projectName}	项目名称. 只能在 template_cmd 命令内使用.
${pathToRoot}	从当前页面到根的路径. 可以用于定位资源, 只能在 template_cmd 命令内使用.

变量 projectName 和 pathToRoot 只能在 template_cmd 命令内使用, 因为它们需要更多的上下文信息, 因此需要在更晚的阶段由 MultiModule ("多项目构建" in "Gradle") task 解析:

```
<@template_cmd name="projectName">
    <span>${projectName}</span>
</@template_cmd>
```

命令

你也可以使用下面这些由 Dokka 定义的 命令 (https://freemarker.apache.org/docs/ref_directive_userDefined.html):

变量	描述
`<@content/>`	主页面内容.
`<@resources/>`	资源, 例如脚本和样式表.
`<@version/>`	从配置得到的模块版本. 如果应用了 versioning plugin (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-versioning), 它会被替换为一个版本导航器.

Markdown

最终更新: 2024/09/10

> Markdown 输出格式还处于 Alpha 状态, 因此你在使用时可能遇到 bug, 或迁移问题. **使用这个功能时, 请自行承担风险.**

Dokka 能够生成 GitHub 风格 和 Jekyll 兼容的 Markdown 格式文档.

使用这些格式, 你可以更加自由的将文档发布到网站, 因为输出可以嵌入到你的文档页面之内. 例如, 象 OkHttp's API 参考文档 (https://square.github.io/okhttp/4.x/okhttp/okhttp3/) 页面.

Markdown 输出格式作为 Dokka plugins (Dokka Plugin) 来实现, 由 Dokka 开发组维护, 并且是开源的.

GFM

GFM 输出格式会生成 GitHub 风格的 Markdown (https://github.github.com/gfm/) 格式的文档.

Gradle

Gradle plugin for Dokka (Gradle) 包含了 GFM 输出格式. 你可以通过以下 task 使用它:

Task	描述
`dokkaGfm`	为单个项目生成生成 GFM 文档.
`dokkaGfmMultiModule`	只为多项目构建中的父项目创建的 `MultiModule` ("多项目构建" in "Gradle") task. 它会为子项目生成文档, 并将所有的输出合并到单个位置, 使用共同的内容目录.
`dokkaGfmCollector`	只为多项目构建中的父项目创建的 `Collector` ("Collector task" in "Gradle") task. 它会为每个子项目调用 `dokkaGfm`, 并将所有的输出合并到一个单独的虚拟项目.

Maven

由于 GFM 格式是以 Dokka plugin ("应用 Dokka plugin" in "Dokka Plugin") 方式实现的, 因此你需要以 plugin 依赖项的方式使用它:

```xml
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>gfm-plugin</artifactId>
                <version>1.9.20</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

完成上面的配置之后, 请运行 `dokka:dokka` goal 来生成 GFM 格式文档.
更多详情, 请参见 Maven plugin 文档中的 其它输出格式 ("其他输出格式" in "Maven").

CLI

由于 GFM 格式是以 Dokka plugin ("应用 Dokka plugin" in "Dokka Plugin") 方式实现的, 因此你需要 下载 JAR 文件 (https://repo1.maven.org/maven2/org/jetbrains/dokka/gfm-plugin/1.9.20/gfm-plugin-1.9.20.jar) 并将它传递给 `pluginsClasspath`.
通过 命令行选项 ("使用命令行选项运行" in "CLI") 方式:

```
java -jar dokka-cli-1.9.20.jar \
     -pluginsClasspath "./dokka-base-1.9.20.jar;...;./gfm-plugin-1.9.20.jar" \
     ...
```

通过 JSON 配置 ("使用 JSON 配置运行" in "CLI") 方式:

```json
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.9.20.jar",
```

```
    "...",
    "./gfm-plugin-1.9.20.jar"
  ],
  ...
}
```

更多详情, 请参见 CLI 运行器文档中的 其它输出格式 ("其它输出格式" in "CLI").

源代码请参见 GitHub (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-gfm).

Jekyll

Jekyll 输出格式生成 Jekyll (https://jekyllrb.com/) 兼容的 Markdown 格式文档.

Gradle

Gradle plugin for Dokka (Gradle) 包含了 Jekyll 输出格式. 你可以通过以下 task 使用它:

Task	描述
dokkaJekyll	为单个项目生成生成 Jekyll 文档.
dokkaJekyllMultiModule	只为多项目构建中的父项目创建的 MultiModule ("多项目构建" in "Gradle") task. 它会为子项目生成文档, 并将所有的输出合并到单个位置, 使用共同的内容目录.
dokkaJekyllCollector	只为多项目构建中的父项目创建的 Collector ("Collector task" in "Gradle") task. 它会为每个子项目调用 dokkaJekyll, 并将所有的输出合并到一个单独的虚拟项目.

Maven

由于 Jekyll 格式是以 Dokka plugin ("应用 Dokka plugin" in "Dokka Plugin") 方式实现的, 因此你需要以 plugin 依赖项的方式使用它:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
```

```xml
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>jekyll-plugin</artifactId>
                <version>1.9.20</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

完成上面的配置之后, 请运行 `dokka:dokka` goal 来生成 GFM 格式文档.

更多详情, 请参见 Maven plugin 文档中的 其它输出格式 ("其他输出格式" in "Maven").

CLI

由于 Jekyll 格式是以 Dokka plugin ("应用 Dokka plugin" in "Dokka Plugin") 方式实现的, 因此你需要 下载 JAR 文件 (https://repo1.maven.org/maven2/org/jetbrains/dokka/jekyll-plugin/1.9.20/jekyll-plugin-1.9.20.jar). 这个格式也是基于 GFM 格式, 因此你还需要提供 GFM 格式的依赖项. 两个 JAR 文件都需要传递给 `pluginsClasspath`:

通过 命令行选项 ("使用命令行选项运行" in "CLI") 方式:

```
java -jar dokka-cli-1.9.20.jar \
     -pluginsClasspath "./dokka-base-1.9.20.jar;...;./gfm-plugin-1.9.20.jar;./jekyll-plugin-1.9.20.jar" \
     ...
```

通过 JSON 配置 ("使用 JSON 配置运行" in "CLI") 方式:

```json
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.9.20.jar",
    "...",
    "./gfm-plugin-1.9.20.jar",
    "./jekyll-plugin-1.9.20.jar"
  ],
```

```
    ...
}
```

更多详情, 请参见 CLI 运行器文档中的 其它输出格式 ("其它输出格式" in "CLI").

源代码请参见 GitHub (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-jekyll).

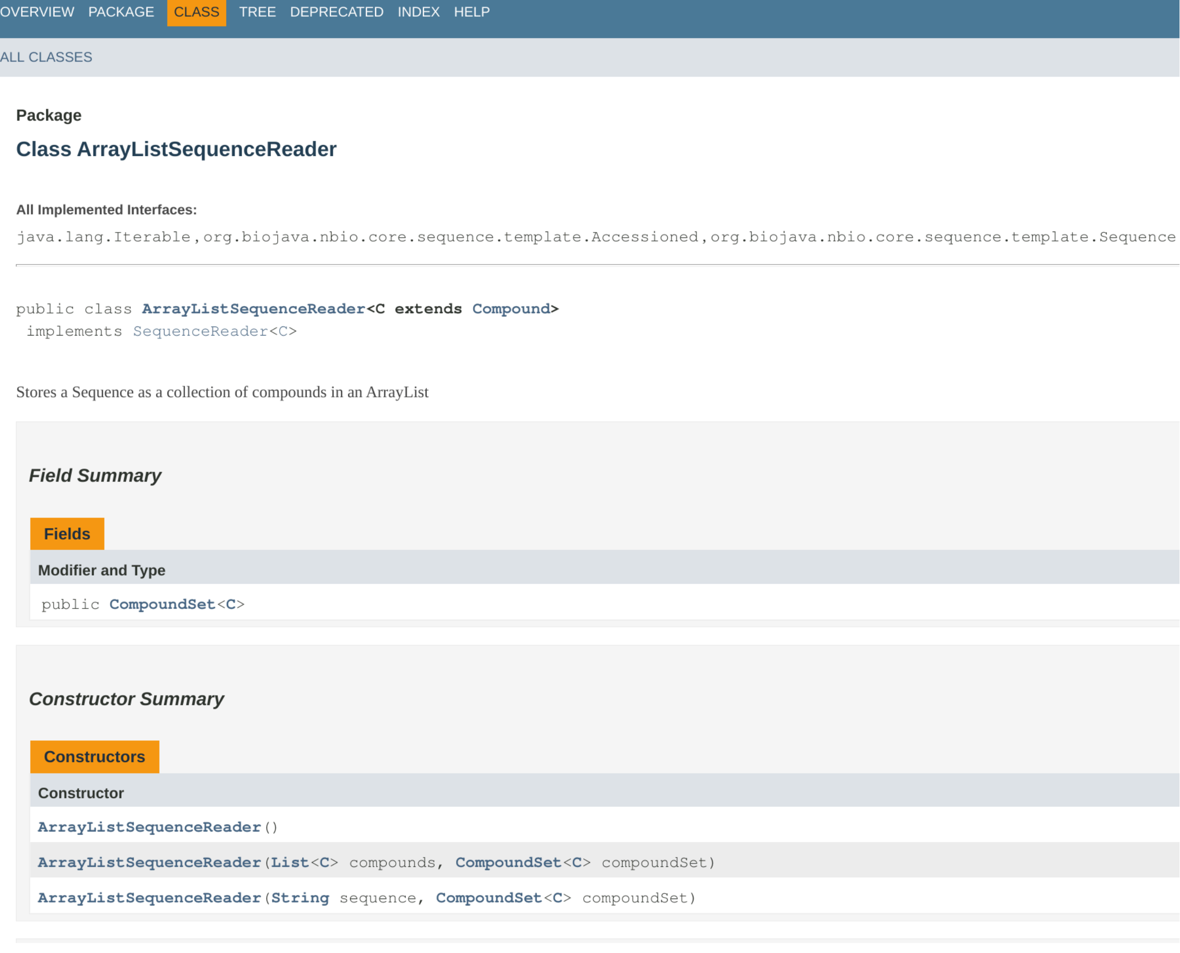
Javadoc

最终更新: 2024/09/10

> Javadoc 输出格式还处于 Alpha 状态, 因此你在使用时可能遇到 bug, 或迁移问题. 我们不保证能够与那些接受 Java 的 Javadoc HTML 格式作为输入的工具成功的集成. **使用这个功能时, 请自行承担风险.**

Dokka 的 Javadoc 输出格式与 Java 的 Javadoc HTML 格式 (https://docs.oracle.com/en/java/javase/19/docs/api/index.html) 类似.

它试图在视觉效果上模仿由 Javadoc 工具生成的 HTML 页面, 但它不是 Javadoc 的直接实现, 也不是完全一样的复制.



javadoc 输出格式

所有的 Kotlin 代码和签名都会以 Java 的视角来显示. 这是通过我们的 Kotlin as Java Dokka plugin (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-kotlin-as-java) 来实现的, 这个 plugin 是 Dokka 默认附带的, 而且对这个格式会默认使用.

Javadoc 输出格式作为一个 Dokka plugin (Dokka Plugin) 来实现, 由 Dokka 开发组维护. 它是开源的, 源代码请参见 GitHub (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-javadoc).

生成 Javadoc 文档

Javadoc 格式不支持跨平台项目.

Gradle

Gradle plugin for Dokka (Gradle) 包含了 Javadoc 输出格式. 你可以使用以下 task:

Task	描述
dokkaJavadoc	为单个项目生成 Javadoc 文档.
dokkaJavadocCollector	只为多项目构建中的父项目创建的 Collector ("Collector task" in "Gradle") task. 它会为每个子项目调用 dokkaJavadoc, 并将所有的输出合并到一个单独的虚拟项目.

javadoc.jar 文件可以单独生成. 详情请参见, 构建 javadoc.jar ("构建 javadoc.jar" in "Gradle").

Maven

Maven plugin for Dokka (Maven) 包含了 Javadoc 输出格式. 你可以使用以下 goal 生成文档:

Goal	描述
dokka:javadoc	生成 Javadoc 格式文档
dokka:javadocJar	生成 javadoc.jar 文件, 其中包含 Javadoc 格式文档

CLI

由于 Javadoc 输出格式是一个 Dokka plugin ("应用 Dokka plugin" in "Dokka Plugin"), 因此你需要 下载 plugin 的 JAR 文件 (https://repo1.maven.org/maven2/org/jetbrains/dokka/javadoc-plugin/1.9.20/javadoc-plugin-1.9.20.jar).
Javadoc 输出格式有 2 个依赖项, 你需要通过额外的 JAR 文件的方式提供:

- kotlin-as-java plugin (https://repo1.maven.org/maven2/org/jetbrains/dokka/kotlin-as-java-plugin/1.9.20/kotlin-as-java-plugin-1.9.20.jar)
- korte-jvm (https://repo1.maven.org/maven2/com/soywiz/korlibs/korte/korte-jvm/3.3.0/korte-jvm-3.3.0.jar)

通过 命令行选项 ("使用命令行选项运行" in "CLI") 方式:

```
java -jar dokka-cli-1.9.20.jar \
     -pluginsClasspath "./dokka-base-1.9.20.jar;...;./javadoc-plugin-1.9.20.jar" \
     ...
```

通过 JSON 配置 ("使用 JSON 配置运行" in "CLI") 方式:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.9.20.jar",
    "...",
    "./kotlin-as-java-plugin-1.9.20.jar",
    "./korte-jvm-3.3.0.jar",
    "./javadoc-plugin-1.9.20.jar"
  ],
```

```
    ...
}
```

更多详情, 请参见 CLI 运行器文档中的 其他输出格式 ("其它输出格式" in "CLI").

Dokka Plugin

最终更新: 2024/09/10

Dokka 的设计思想是易于扩展, 而且高度可定制化, 因此对于 Dokka 缺少的, 或者没有默认提供的细节功能, 社区开发者可以实现 plugin.

Dokka plugin 的范围很广, 包括支持其他编程语言的源代码, 到支持各种输出格式. 你可以对你自己的 KDoc tag 或注解添加支持, 教会 Dokka 如何输出 KDoc 描述中出现的各种的 DSL, 对 Dokka 页面的外观重新设计, 使其无缝的集成到你的公司的网站, 将 Dokka 与其他工具集成, 等等等等.

如果你想要学习如何 创建 Dokka plugin, 请参见 开发者指南 (https://kotlin.github.io/dokka/1.9.20/developer_guide/introduction/).

应用 Dokka plugin

Dokka plugin 作为单独的 artifact 发布, 因此要应用一个 Dokka plugin, 你只需要将它添加为依赖项. 之后, plugin 会自行扩展 Dokka - 不需要你再进行更多工作.

> 使用相同扩展点的 plugin, 或者以类似方式工作的 plugin, 可能会互相影响. 因此可能会导致文档外观上的 bug, 不确定的行为, 甚至构建失败. 但是, 应该不会导致一致性问题, 因为 Dokka 没有公开任何可变的数据结构和对象.
>
> 如果你发现这类问题, 建议检查应用了哪些 plugin, 以及这些 plugin 的行为.

我们来看看在你的项目中如何应用 mathjax plugin (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-mathjax):

Kotlin

Gradle plugin for Dokka 会创建便利的依赖项配置, 你可以对全局应用 plugin, 或只对特定的输出格式应用 plugin.

```
dependencies {
    // 对全局应用
    dokkaPlugin("org.jetbrains.dokka:mathjax-plugin:1.9.20")

    // 只对单模块的 dokkaHtml task 应用
    dokkaHtmlPlugin("org.jetbrains.dokka:kotlin-as-java-
plugin:1.9.20")
```

```
    // 对多项目构建中的 HTML 格式应用
    dokkaHtmlPartialPlugin("org.jetbrains.dokka:kotlin-as-java-
plugin:1.9.20")
}
```

> 在对 多项目 ("多项目构建" in "Gradle") 构建生成文档时, 你需要同时对子项目和它们的父项目应用 Dokka plugin.

Groovy

Gradle plugin for Dokka 会创建便利的依赖项配置, 你可以对全局应用 plugin, 或只对特定的输出格式应用 plugin.

```
dependencies {
    // 对全局应用
    dokkaPlugin 'org.jetbrains.dokka:mathjax-plugin:1.9.20'

    // 只对单模块的 dokkaHtml task 应用
    dokkaHtmlPlugin 'org.jetbrains.dokka:kotlin-as-java-
plugin:1.9.20'

    // 对多项目构建中的 HTML 格式应用
    dokkaHtmlPartialPlugin 'org.jetbrains.dokka:kotlin-as-java-
plugin:1.9.20'
}
```


> 在对 多项目 ("多项目构建" in "Gradle") 构建生成文档时, 你需要同时对子项目和它们的父项目应用 Dokka plugin.

Maven

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
```

```xml
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>mathjax-plugin</artifactId>
                <version>1.9.20</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

CLI

如果你使用 CLI (CLI) 运行器的 命令行选项 ("使用命令行选项运行" in "CLI") 模式, Dokka plugin 应该以 .jar 文件的方式传递给 -pluginsClasspath:

```
java -jar dokka-cli-1.9.20.jar \
     -pluginsClasspath "./dokka-base-1.9.20.jar;...;./mathjax-plugin-1.9.20.jar" \
     ...
```

如果你使用 JSON 配置 ("使用 JSON 配置运行" in "CLI") 模式, Dokka plugin 应该在 pluginsClasspath 之下指定.

```json
{
  ...
  "pluginsClasspath": [
    "./dokka-base-1.9.20.jar",
    "...",
    "./mathjax-plugin-1.9.20.jar"
  ],
  ...
}
```

配置 Dokka plugin

Dokka plugin 也可以带有它们自己的配置选项. 要查看有哪些选项可以使用, 请参考你使用的 plugin 的文档.

我们来看看如何配置 `DokkaBase` plugin, 它负责生成 HTML (HTML) 文档. 我们向 assets 添加自定义的图片(使用 `customAssets` 选项), 添加自定义的样式表 (使用 `customStyleSheets` 选项), 修改页脚文字 (使用 `footerMessage` 选项):

Kotlin

Gradle 的 Kotlin DSL 可以使用类型安全的 plugin 配置. 做法是, 在 `buildscript` 代码段, 向 classpath 依赖项添加 plugin 的 artifact, 然后导入 plugin 和配置的类:

```
import org.jetbrains.dokka.base.DokkaBase
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.base.DokkaBaseConfiguration

buildscript {
    dependencies {
        classpath("org.jetbrains.dokka:dokka-base:1.9.20")
    }
}

tasks.withType<DokkaTask>().configureEach {
    pluginConfiguration<DokkaBase, DokkaBaseConfiguration> {
        customAssets = listOf(file("my-image.png"))
        customStyleSheets = listOf(file("my-styles.css"))
        footerMessage = "(c) 2022 MyOrg"
    }
}
```

另一种方法是, plugin 可以通过 JSON 进行配置. 通过这种方法, 不需要添加额外的依赖项.

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType<DokkaTask>().configureEach {
    val dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
```

```
    }
    """
    pluginsMapConfiguration.set(
        mapOf(
            // plugin 的完整限定名称, to, json 配置
            "org.jetbrains.dokka.base.DokkaBase" to
dokkaBaseConfiguration
        )
    )
}
```

Groovy

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType(DokkaTask.class) {
    String dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
    }
    """
    pluginsMapConfiguration.set(
        // plugin 的完整限定名称, :, json 配置
        ["org.jetbrains.dokka.base.DokkaBase":
dokkaBaseConfiguration]
    )
}
```

Maven

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
```

```xml
    <configuration>
        <pluginsConfiguration>
            <!-- plugin 的完整限定名称 -->
            <org.jetbrains.dokka.base.DokkaBase>
                <!-- 选项名称 -->
                <customAssets>
                    <asset>${project.basedir}/my-
image.png</asset>
                </customAssets>
                <customStyleSheets>
                    <stylesheet>${project.basedir}/my-
styles.css</stylesheet>
                </customStyleSheets>
                <footerMessage>(c) MyOrg 2022
Maven</footerMessage>
            </org.jetbrains.dokka.base.DokkaBase>
        </pluginsConfiguration>
    </configuration>
</plugin>
```

CLI

如果你使用 CLI (CLI) 运行器的 命令行选项 ("使用命令行选项运行" in "CLI") 模式, 请使用 `-pluginsConfiguration` 选项来接受 JSON 配置, 选项值的格式是 `fullyQualifiedPluginName=json`.
如果你需要配置多个 plugin, 你可以传递多个值, 以 `^^` 分隔.

```
java -jar dokka-cli-1.9.20.jar \
     ...
     -pluginsConfiguration "org.jetbrains.dokka.base.DokkaBase=
{\"customAssets\": [\"my-image.png\"], \"customStyleSheets\":
[\"my-styles.css\"], \"footerMessage\": \"(c) 2022 MyOrg CLI\"}"
```

如果你使用 JSON 配置 ("使用 JSON 配置运行" in "CLI"), 也有类似的 `pluginsConfiguration` 数组, 在它的 `values` 中接受 JSON 配置.

```
{
  "moduleName": "Dokka Example",
  "pluginsConfiguration": [
```

```
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"customAssets\": [\"my-image.png\"],
\"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\":
\"(c) 2022 MyOrg\"}"
    }
  ]
}
```

重要的 plugin

下面是一些重要的 Dokka plugin, 可能对你有用:

名称	描述
Android documentation plugin (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-android-documentation)	改善 Android 上的文档体验
Versioning plugin (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-versioning)	添加版本选择器, 帮助组织你的应用程序/库的多个不同版本的文档
MermaidJS HTML plugin (https://github.com/glureau/dokka-mermaid)	输出 KDocs 中出现的 MermaidJS (https://mermaid-js.github.io/mermaid/#/) 图和视觉效果
Mathjax HTML plugin (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-mathjax)	美化输出 KDocs 中出现的数学公式
Kotlin as Java plugin (https://github.com/Kotlin/dokka/tree/1.9.20/dokka-subprojects/plugin-kotlin-as-java)	以 Java 视角输出 Kotlin 签名

如果你是 Dokka plugin 的开发者, 希望将你的 plugin 添加到这个列表, 请通过 Slack ("社区" in "介绍") 或 GitHub (https://github.com/Kotlin/dokka/) 联系维护者.

模块文档

最终更新: 2024/09/10

针对模块整体的文档, 以及针对模块内包的文档, 可以通过单独的 Markdown 文件的形式提供.

文件格式

在 Markdown 文件内, 针对模块整体的文档, 以及针对各个包的文档, 分别使用各自的顶级标题来指定. 对于模块, 标题文字 **必须** 是 **Module `<module name>`**, 对于包, 标题文字 **必须** 是 **Package `<package qualified name>`**.

文件不一定需要同时包含模块和包的文档. 你可以让文件只包含包或模块的文档. 也可以为每个模块或包分别生成 Markdown 文件.

使用 Markdown 语法 (https://www.markdownguide.org/basic-syntax/), 你可以添加:

- 最多 6 层的标题
- 粗体或斜体格式的强调内容
- 链接
- 内嵌代码
- 代码块
- 引用块

下面是一个示例文件, 同时包含模块和包的文档:

```
# Module kotlin-demo

这段内容出现在你的模块名称之下.

# Package org.jetbrains.kotlin.demo

这段内容出现在包列表中的你的包名称之下.
也出现在你的包的页面的顶级标题之下.

## 包 org.jetbrains.kotlin.demo 的二级标题
```

```
这个标题之后的内容也是 `org.jetbrains.kotlin.demo` 的文档的一部分

# Package org.jetbrains.kotlin.demo2

这段内容出现在包列表中的你的包名称之下.
也出现在你的包的页面的顶级标题之下.

## 包 org.jetbrains.kotlin.demo2 的二级标题

这个标题之后的内容也是 `org.jetbrains.kotlin.demo2` 的文档的一部分
```

使用 Gradle 的示例项目, 可以参见 Dokka Gradle 示例 (https://github.com/Kotlin/dokka/tree/1.9.20/examples/gradle/dokka-gradle-example).

向 Dokka 传递文件

要将文件传递给 Dokka, 你需要对 Gradle, Maven, 或 CLI, 使用相关的 **includes** 选项:

Gradle

在 源代码集配置 ("源代码集配置" in "Gradle") 中 使用 includes ("includes" in "Gradle") 选项.

Maven

在 一般配置 ("一般配置" in "Maven") 中 使用 includes ("includes" in "Maven") 选项.

CLI

如果你使用命令行配置, 请在 源代码集选项 ("源代码集选项" in "CLI") 中 使用 includes ("源代码集选项" in "CLI") 选项.
如果你使用 JSON 配置, 请在 一般配置 ("一般配置" in "CLI") 中 使用 includes ("includes" in "CLI") 选项.

支持 Kotlin 开发的 IDE

最终更新: 2024/09/10

JetBrains 为 IntelliJ IDEA, JetBrains Fleet, 和 Android Studio 提供了官方的 Kotlin plugin.

其他 IDE 和代码编辑器, 比如 Eclipse, Visual Studio Code, 和 Atom, 也有 Kotlin 社区支持的 plugin.

IntelliJ IDEA

IntelliJ IDEA (https://www.jetbrains.com/idea/download/) 是一个用于 JVM 语言的 IDE, 它致力于最大限度的提高开发者的生产效率. 它替你你完成日常重复的任务, 提供智能代码完成, 静态代码分析, 以及重构等功能, 让你的精力集中到软件开发的美好的一面, 不仅生产率高, 而且使用体验很愉快.

Kotlin plugin 包含在 IntelliJ IDEA 的每个发布版内.

关于 IntelliJ IDEA, 详情请参见 官方文档 (https://www.jetbrains.com/help/idea/discover-intellij-idea.html).

Fleet

> JetBrains Fleet 目前是 Public Preview 状态, 可以自由使用.

JetBrains Fleet (https://www.jetbrains.com/fleet/) 是一个支持多种语言的 IDE 和代码编辑器, 为 Kotlin 提供高级支持, 对 Kotlin 开发者提供流畅的开发体验. 你可以使用 Fleet 作为代码编辑器, 以便快速修改代码, 也可以打开 Smart Mode, 将它变成一个功能完备的 IDE, 包含智能代码提示功能.

各个 Fleet 发布版都包含了 Kotlin plugin.

Fleet 也支持 Kotlin Multiplatform 项目, 目标平台包括 Android, iOS, 以及 Desktop 平台, 支持代码的测试和调试. Fleet 的 Smart Mode 会选择适当的代码处理引擎, 并可以在 Kotlin Multiplatform 代码 和其他能与 Kotlin 互操作的语言编写的代码之间, 进行代码导航.

Fleet 的使用入门, 请参见教程 使用 Fleet 进行跨平台开发 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/fleet.html).

Android Studio

Android Studio (https://developer.android.com/studio) 是用于 Android App 开发的官方 IDE, 它基于 IntelliJ IDEA (https://www.jetbrains.com/idea/). 在 IntelliJ 强大的代码编辑器和开发工具的基础之上, Android Studio 还提供了更多功能, 在 Android App 的开发过程中提高你的生产效率.

Kotlin plugin 包含在 Android Studio 的每个发布版内.

关于 Android Studio, 详情请参见 官方文档 (https://developer.android.com/studio/intro).

Eclipse

Eclipse (https://eclipseide.org/release/) 是一个用于各种编程语言开发应用程序的 IDE, 包括 Kotlin. Eclipse 也有 Kotlin plugin: 最初由 JetBrains 开发, 现在 Kotlin plugin 由 Kotlin 社区贡献者维护.

你可以 从 Eclipse Marketplace 手动安装 Kotlin plugin (https://marketplace.eclipse.org/content/kotlin-plugin-eclipse).

Kotlin 开发组管理 Kotlin plugin for Eclipse 的开发和贡献过程. 如果你想要贡献内容给这个 plugin, 请向 Kotlin for Eclipse 的 GitHub 代码仓库 (https://github.com/Kotlin/kotlin-eclipse) 提交 pull request.

与 Kotlin 语言各版本的兼容性

对于 IntelliJ IDEA, Fleet 和 Android Studio, Kotlin plugin 包含在 IDE 的各个发布版之内. 新的 Kotlin 版本发布之后, 这些 IDE 会自动建议更新 Kotlin 到最新的版本. 关于各个 IDE 支持的最新的语言版本, 请参见 Kotlin 的发布版本 ("IDE 支持" in "Kotlin 的发布版本").

对其他 IDE 的支持

JetBrains 没有为其他 IDE 提供 Kotlin plugin. 但是, 其他 IDE 和源代码编辑器, 比如 Eclipse, Visual Studio Code, 和 Atom, 也有它们自己的, 由 Kotlin 社区维护的 Kotlin plugin.

你可以使用任何文本编辑器来编写 Kotlin 代码, 但没有 IDE 相关的功能: 代码格式化, 调试工具, 等等. 要在文本编辑器中使用 Kotlin, 你可以从 Kotlin GitHub 发布页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.23) 下载最新的 Kotlin 命令行编译器 (`kotlin-compiler-1.9.23.zip`), 然后 手动安装它 ("手动安装" in "Kotlin 命令行编译器"). 你也可以使用包管理器, 比如 Homebrew ("使用 Homebrew 安装" in "Kotlin 命令行编译器"), SDKMAN! ("使用 SDKMAN! 安装" in "Kotlin 命令行编译器"), 以及 Snap 包 ("使用 Snap 包安装" in "Kotlin 命令行编译器").

下一步做什么?

- 使用 IntelliJ IDEA IDE 创建你的第一个项目 (Kotlin/JVM 入门)
- 使用 Fleet 创建 Multiplatform 项目 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/fleet.html)
- 使用 Android Studio 创建你的第一个跨平台移动应用程序 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-create-first-app.html)
- 学习如何 安装 Kotlin plugin 的 EAP 版本 (安装 Kotlin EAP Plugin)

迁移到 Kotlin 编码风格

最终更新: 2024/09/10

Kotlin 编码规约与 IntelliJ IDEA 源代码格式化

关于如何编写符合 Kotlin 习惯的代码, Kotlin 编码规约 (编码规约) 讲到了很多方面的内容, 其中还包括一些代码格式化方面的建议, 以便提高 Kotlin 代码的可读性.

不幸的是, 在本文档发布之前很长时间, IntelliJ IDEA 内建的源代码格式化工具就已经开始工作了, 因此它目前默认的设置与现在推荐的代码格式化规则存在一些不同.

符合逻辑的做法似乎是切换 IntelliJ IDEA 默认设置, 消除这些不一致, 让格式化规则与 Kotlin 编码规约保持一致. 但是这就意味着, Kotlin plugin 安装的那一刻, 所有现存的 Kotlin 项目都会使用新的代码风格. 这并不是我们更新 Kotlin plugin 时期待的结果, 对不对?

所以我们制定了下面的迁移计划:

- 从 Kotlin 1.3 开始, 默认启用官方的代码风格格式化设置, 而且只用于新项目 (旧的格式化设置可以手工启用)
- 既有项目的作者可以选择迁移到 Kotlin 编码规约
- 既有项目的作者可以在某个项目内明确指定使用旧的代码风格格式化设置 (这样, 将来切换到默认设置时项目不会受影响)
- 在 Kotlin 1.4 中切换到默认的格式化设置, 并使它与 Kotlin 编码规约一致

"Kotlin 编码规约" 与 "IntelliJ IDEA 默认代码风格" 之间的不同

最大的变化就是连续缩进规则. 使用双倍缩进来表示一个多行的表达式在前一行还未结束, 这是很好的. 这是一个非常简单而且非常通行的规则, 但是这样格式化之后, 有些 Kotlin 构造器看起来会有点奇怪. 在 Kotlin 编码规约中推荐使用单倍缩进, 而以前会强制使用很长的连续缩进.

```kotlin
class User(
        val name: String,
        val address: String) {
    val key: String = getUserId(name, address)
}

fun findUser(
        users: List<User>,
        key: String? = null,
        name: String? = null): User? {
    val filteredUsers = users
            .asSequence()
            .filter { user -> key == null || user.key == key }
            .filter { user -> name == null || user.name.contains(name) }

    return filteredUsers.single()
}
```

```kotlin
class User(
    val name: String,
    val address: String
) {
    val key: String = getUserId(name, address)
}

fun findUser(
    users: List<User>,
    key: String? = null,
    name: String? = null
): User? {
    val filteredUsers = users
        .asSequence()
        .filter { user -> key == null || user.key == key }
        .filter { user -> name == null || user.name.contains(name) }

    return filteredUsers.single()
}
```

代码格式化的不同

在实际使用中, 会有很多代码受到影响, 因此这个变化被认为是一个大的代码风格变更.

关于迁移到新的代码风格的讨论

采用一种新的代码风格, 对于一个新的项目来说也许是非常自然的步骤, 因为并没有源代码是使用旧的规则格式化的. 因此从 1.3 版开始, Kotlin IntelliJ Plugin 创建项目时, 默认使用与 Kotlin 编码规约 (编码规约)一致的代码格式化规则.

对一个已有的项目改变它的代码格式化规则就是一件费力得多的工作了, 而且应该先在整个开发团队中就此进行讨论.

在已有的项目中修改代码格式带来的主要坏处是, 源代码版本管理系统(VCS)的 blame/annotate 功能会更多地指向无关的提交(commit). 虽然每种源代码版本管理系统都有某种办法可以解决这个问题 (在 IntelliJ IDEA 中可以使用 "注解前一个版本" (https://www.jetbrains.com/help/idea/investigate-changes.html)), 但是事先考虑一下新的代码风格是不是值得我们耗费这些努力, 还是很重要的. 在源代码版本管理系统中, 将源代码格式化导致的提交与真正有意义的修改区分开来, 对于将来查看代码变更历史有很大帮助.

而且, 对于比较大的开发组来说迁移会更困难, 因为在多个子系统中提交大量文件可能会在个人的开发分支中导致文件合并时的冲突. 虽然每个冲突的解决通常都很简单, 但还是应该事先搞清楚, 是不是存在某个分支正在进行大的功能开发工作.

总的来说, 对于小的项目, 我们建议一次性转换所有文件.

对于中型和大型项目, 决策可能比较困难. 如果你还没有准备好马上更新大量文件, 你可以决定逐个模块进行迁移, 或者只对你开发中修改的文件逐渐迁移.

迁移到新的代码风格

可以通过 **Settings/Preferences** | **Editor** | **Code Style** | **Kotlin** 对话框切换到 Kotlin 编码规约的代码风格. 将 scheme 切换为 **Project**, 然后激活 **Set from...** | **Kotlin style guide**.

如果要将这些变更共享给项目的所有开发者, 必须把 `.idea/codeStyle` 文件夹提交到源代码版本管理系统.

如果使用外部的编译系统来配置项目, 而且决定不共享 `.idea/codeStyle` 文件夹, 可以通过额外的属性来强制使用 Kotlin 编码规约:

对于 Gradle

在项目根目录下的 `gradle.properties` 文件中, 添加 `kotlin.code.style=official` 属性, 并把这个文件提交到源代码版本管理系统.

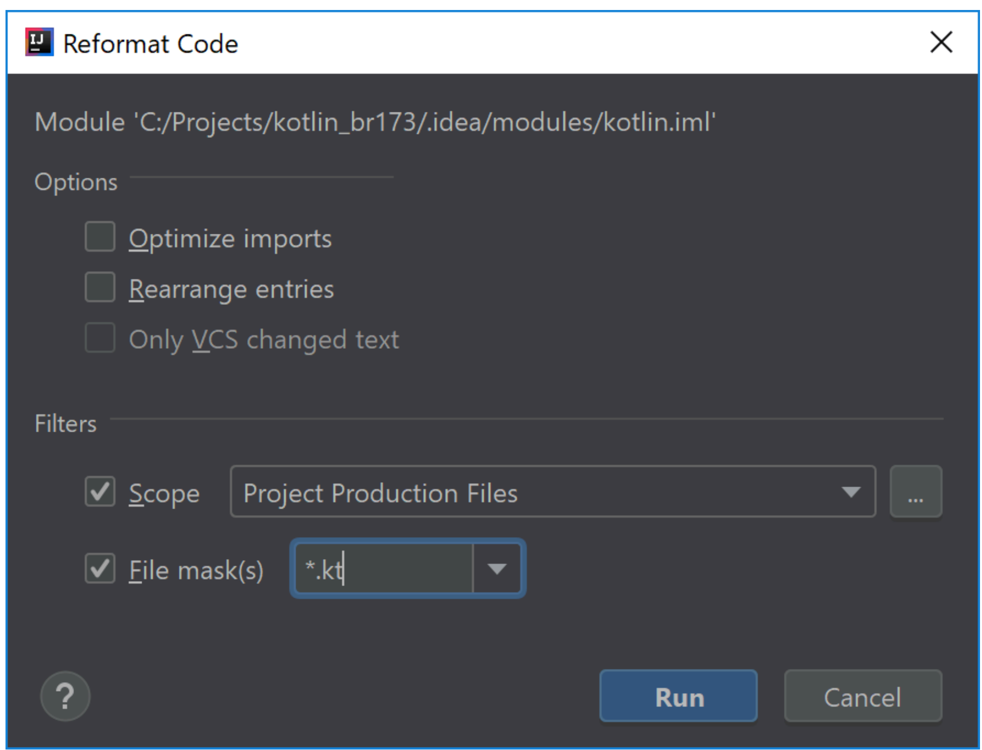
对于 Maven

对项目的根 `pom.xml` 文件, 添加 `kotlin.code.style official` 属性.

```
<properties>
    <kotlin.code.style>official</kotlin.code.style>
</properties>
```

> ⚠ 设置 **kotlin.code.style** 属性后, 导入项目时可能会修改 IDE 的代码风格 scheme, 并且可能修改 IDE 的代码风格设置.

升级你的代码风格设置之后, 可以在 project 中选择你希望的范围, 然后启动 **Reformat Code** 对话框.



源代码重格式化对话框

对于各个文件逐渐迁移的情况, 可以激活 **File is not formatted according to project settings** 检查器. 这个检查器会高亮标识需要重新格式化的代码. 打开 **Apply only to modified files** 选项时, 检查器会只显示修改过的文件中的代码格式化问题. 这些文件很可能就是你将要提交的文件.

将旧的代码风格保存到项目中

如果你需要, 可以将 IntelliJ IDEA 的代码风格明确设置为当前项目的代码风格:

1. 在 **Settings/Preferences** | **Editor** | **Code Style** | **Kotlin** 对话框中, 将 scheme 切换为 **Project**.
2. 打开 **Load/Save** 标签页, 在 **Use defaults from** 项目中选择 **Kotlin obsolete IntelliJ IDEA codestyle**.

如果要将每个开发者的 `.idea/codeStyle` 文件夹的变更共享给整个开发组, 必须将这个文件夹提交到源代码版本管理系统. 或者, 对于通过 Gradle 或 Maven 配置的项目, 可以使用 **kotlin.code.style=obsolete**.

运行代码片段

最终更新: 2024/09/10

Kotlin 代码通常使用项目来管理, 你可以通过 IDE, 文本编辑器, 或其他工具来开发这些项目. 但是, 如果你想要快速查看一个函数如何工作, 或想要计算一个表达式的值, 就没有必要创建新的项目并构建它. 我们来看看在各种环境中直接运行 Kotlin 代码的 3 种便利方法:

- IDE 环境: 草稿(Scratch)与工作簿(Worksheet).
- 浏览器环境: Kotlin Playground.
- 命令行环境: ki shell.

IDE: 草稿(Scratch)与工作簿(Worksheet)

IntelliJ IDEA 和 Android Studio 支持 Kotlin 草稿(Scratch)文件与工作簿(Worksheet) (https://www.jetbrains.com/help/idea/kotlin-repl.html#efb8fb32).

- *草稿(Scratch)文件* (或者直接简称 *草稿*) 可以在你的项目的同一个 IDE 窗口内创建代码草稿, 并立即运行. 草稿不会关联到项目; 你可以从你的 OS 上的任何 IntelliJ IDEA 窗口, 访问并运行你的所有草稿.

  要创建一个 Kotlin 草稿, 请选择菜单 **File** | **New** | **Scratch File**, 然后选择 **Kotlin** 类型.

- *工作簿* 是项目内的文件: 它们存储在项目目录内, 并属于项目的模块. 工作簿可以用来编写那些还不能真正构成软件单元, 但仍然需要与项目一起保存的代码片段. 比如, 教学或演示素材.

  要在项目目录内创建一个 Kotlin 工作簿, 请在项目树的目录上点击鼠标右键, 并选择菜单 **New** | **Kotlin Worksheet**.

在草稿和工作簿中, 支持语法高亮, 自动完成, 以及 IntelliJ IDEA 代码编辑器的所有其他功能. 不需要声明 `main()` 函数 - 你编写的所有代码都会被执行, 就好像它们在 `main()` 函数内一样.

你的代码在草稿或工作簿之内编写完成后, 点击 **Run**. 执行结果将会出现在你的代码行的对面.

运行草稿

交互模式

IDE 可以从草稿和工作簿自动运行代码. 要在你停止输入代码时立即得到执行结果, 请切换到 **Interactive mode**.

草稿交互模式

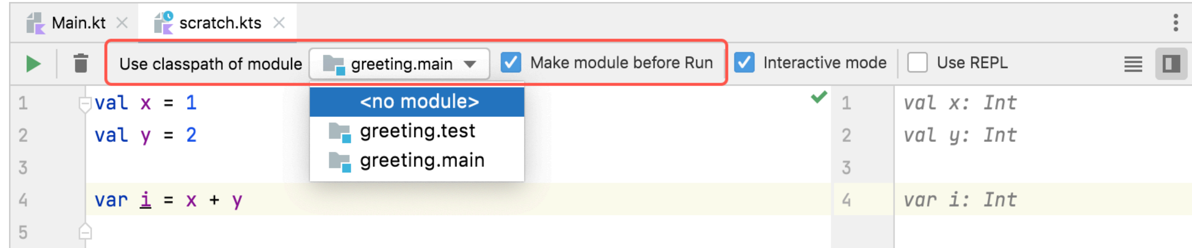
使用模块

在你的草稿和工作簿中, 可以使用 Kotlin 项目中的类和函数.

工作簿自动得到它所属模块中类和函数的访问权.

如果要在草稿中使用项目中的类或函数, 在草稿文件中需要和通常一样使用 `import` 语句导入它们. 然后编写你的代码, 然后在 **Use classpath of module** 中选择适当的模块来运行.

草稿和工作簿都使用相关模块编译后的版本. 因此, 如果你修改了模块的源代码文件, 要到重新构建模块之后, 变更才会反应到草稿和工作簿. 要在草稿或工作簿的每次运行之前自动重新构建模块, 请选择 **Make module before Run**.



草稿选择模块

以 REPL 模式运行

要计算草稿或工作簿中的每一个表达式, 请使用 **Use REPL** 模式来运行. 代码行会按顺序运行, 对每一个调用输出结果. 以后你可以在同一个文件内使用结果, 方法是引用它们自动生成的 `res*` 名称 (名称会显示在对应的行中).

草稿 REPL

浏览器: Kotlin Playground

Kotlin Playground (https://play.kotlinlang.org/) 是一个在线应用程序, 可以在你的浏览器内编写, 运行, 以及共享 Kotlin 代码.

编写和编辑代码

在 Playground 的编辑器区域, 你可以象在一个源代码文件一样的编写代码:

- 以任意顺序添加你自己的类, 函数, 以及顶层声明.
- 在 `main()` 的函数体之内编写可执行的部分.

与通常的 Kotlin 项目一样, Playground 中的 `main()` 函数可以有 `args` 参数, 也可以完全没有参数. 要在执行时传递程序参数, 请在 **Program arguments** 栏写入这些参数.

```
class Person(val name: String)

fun greet(person: Person) = println("Hello ${person.name}!")

fun main(args: Array<String>) {

    greet(Person(args[0]))
    listOf(1, 2, 3).filt
}
```

Playground: 代码自动完成

Playground 会对代码高亮显示, 并为你输入的代码显示自动完成选项. 它会从自动标准库和 `kotlinx.coroutines` (协程(Coroutine)) 导入声明.

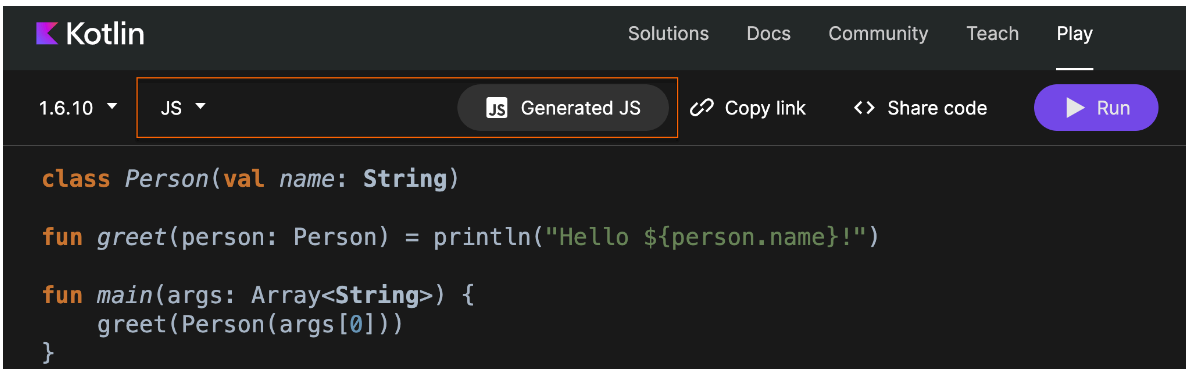
选择执行环境

Playground 提供几种方法来定制执行环境:

- 多个 Kotlin 版本, 包括可用的 未来版本的预览版 (参加 Kotlin EAP 项目).
- 用来运行代码的多个后端: JVM, JS (旧编译器或 IR 编译器 (使用 IR 编译器), 或 Canvas), 或 JUnit.

Playground: 环境设置

对于 JS 后端, 你也可以查看生成的 JS 代码.

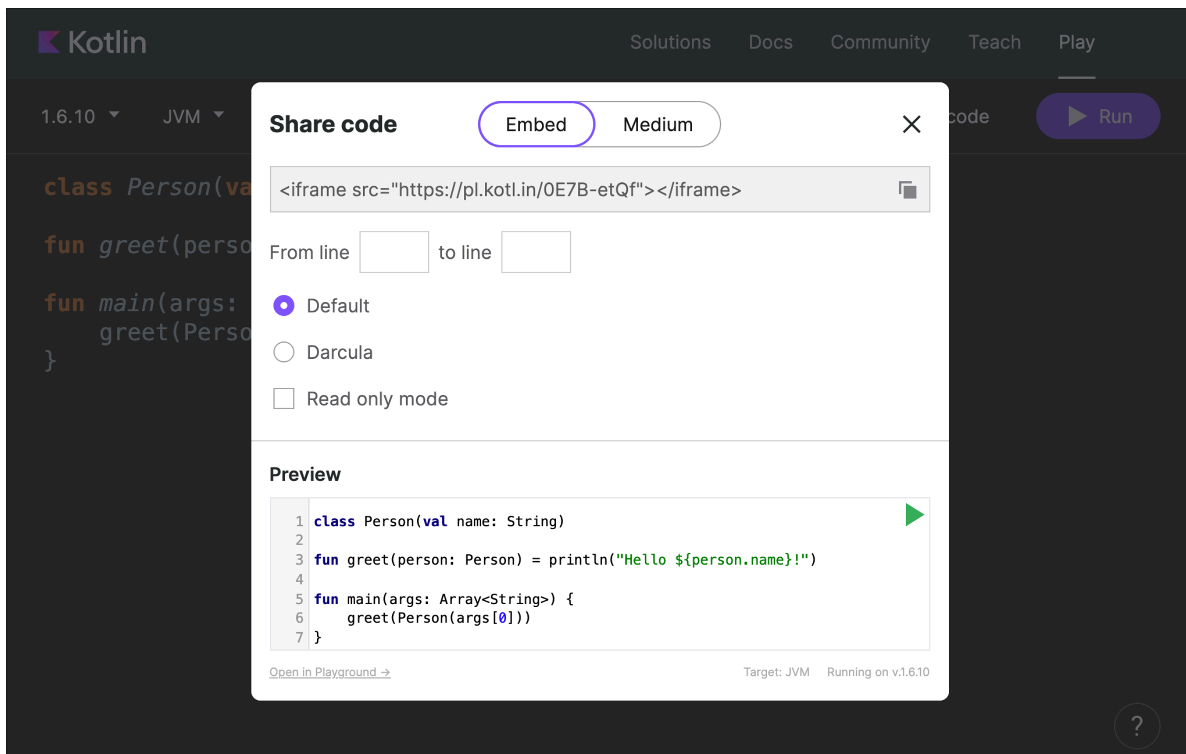


Playground: 生成 JS

在线共享代码

使用 Playground 与他人共享你的代码 – 请点击 **Copy link**, 然后发送给你想要展示代码的任何人.

你可以也将 Playground 中的代码片段内嵌到其他网站, 甚至还能让它们运行起来. 请点击 **Share code**, 将你的示例内嵌到任何网页, 或内嵌到 Medium (https://medium.com/) 的一篇文章.



Playground: 共享代码

命令行: ki shell

ki shell (https://github.com/Kotlin/kotlin-interactive-shell) (*Kotlin Interactive Shell*) 是一个命令行工具, 用来在终端运行 Kotlin 代码. 它可以在 Linux, macOS, 和 Windows 环境运行.

ki shell 提供基本的代码计算能力, 以及一些高级功能, 比如:

- 代码自动完成
- 类型检查
- 外部依赖项
- 代码片段的粘贴模式
- 脚本支持

详情请参见 ki shell GitHub 代码仓库 (https://github.com/Kotlin/kotlin-interactive-shell).

安装并运行 ki shell

要安装 ki shell, 请从 GitHub (https://github.com/Kotlin/kotlin-interactive-shell) 下载最新版本, 并解压缩到你指定的目录.

在 macOS 上, 你也可以运行以下命令, 使用 Homebrew 来安装 ki shell:

```
brew install ki
```

要启动 ki shell, 在 Linux 和 macOS 上请运行 `bin/ki.sh` (如果是通过 Homebrew 安装 ki shell, 可以直接输入 `ki`), 在 Windows, 请运行 `bin\ki.bat`.

shell 开始运行后, 你就可以开始在终端编写 Kotlin 代码了. 可以输入 `:help` (或 `:h`) 查看在 ki shell 中能够使用的命令.

代码自动完成与高亮显示

当你按下 **Tab** 键时, ki shell 会显示代码自动完成选项. 它还为你输入的代码提供语法高亮显示. 你可以输入 `:syntax off` 来关闭这个功能.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] val x = 1 + 2 + 3
[1] var greeting = "Hello"
[2] listOf(1, 2, 3).fil
file                filterIndexedTo(      filterIsInstanceTo(     filterNotNullTo(
filter {            filterIsInstance(     filterNot {             filterNotTo(
filterIndexed {     filterIsInstance()    filterNotNull()         filterTo(
```

ki shell 语法高亮与自动完成

当你 按下 **Enter** 键时, ki shell 会计算输入的行, 并打印结果. 表达式值会作为变量打印输出, 变量使用自动生成的名称 `res*`. 以后你可以在你运行的代码中使用这些变量. 如果输入的结构不完整 (比如, 一个 `if` 有条件部分但没有代码体部分), shell 会打印 3 个点号, 等待输入剩余的部分.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] val greeting = "Hello"
[1] greeting + " world!"
res1: String = Hello world!
[2] println(res1)
Hello world!
[3] if (res1.length > 10)
...
```

ki shell 运行结果

检查表达式类型

对于你不熟悉的复杂的表达式或 API, ki shell 提供 `:type` (或 `:t`) 命令, 它会显示表达式的类型:

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :t sequenceOf("one", "two", "three").associateWith { it.length }
Map<String, Int>
[1]
```

ki shell 类型

装载代码

如果你需要的代码保存在别处, 有 2 种方法可以装载代码并在 ki shell 中使用:

- 使用 `:load` (或 `:l`) 命令, 装载一个源代码文件.
- 使用 `:paste` (或 `:p`) 命令, 在粘贴模式中复制粘贴代码片段.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :l Desktop/sourceFile.kt
[1] :ls
val variableFromFile: String
fun functionFromFile(): Unit
[2] println(variableFromFile)
Hello from file!
[3] functionFromFile()
Calling function from file
[4]
```

ki shell 装载文件

`ls` 命令显示可以使用的符号 (变量和函数).

添加外部依赖项

除标准库之外, ki shell 也支持外部依赖项. 因此你可以试用第三方的库, 而不必创建完整的项目.

要在 ki shell 中添加第三方库, 请使用 `:dependsOn` 命令. ki shell 默认使用 Maven Central, 但如果你使用 `:repository` 命令连接到其他仓库, 也可以使用其他仓库:

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :repository https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven
[1] :dependsOn org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3
[2] import kotlinx.html.*
[3] import kotlinx.html.stream.*
[4] val html = createHTML().html {
...body {
...h1 { "Hello" }
...}
...}
[9] html
res5: String = <html>
  <body>
    <h1></h1>
  </body>
</html>

[10]
```

ki shell 外部依赖项

使用 TeamCity 对 Kotlin 项目进行持续集成(Continuous Integration)

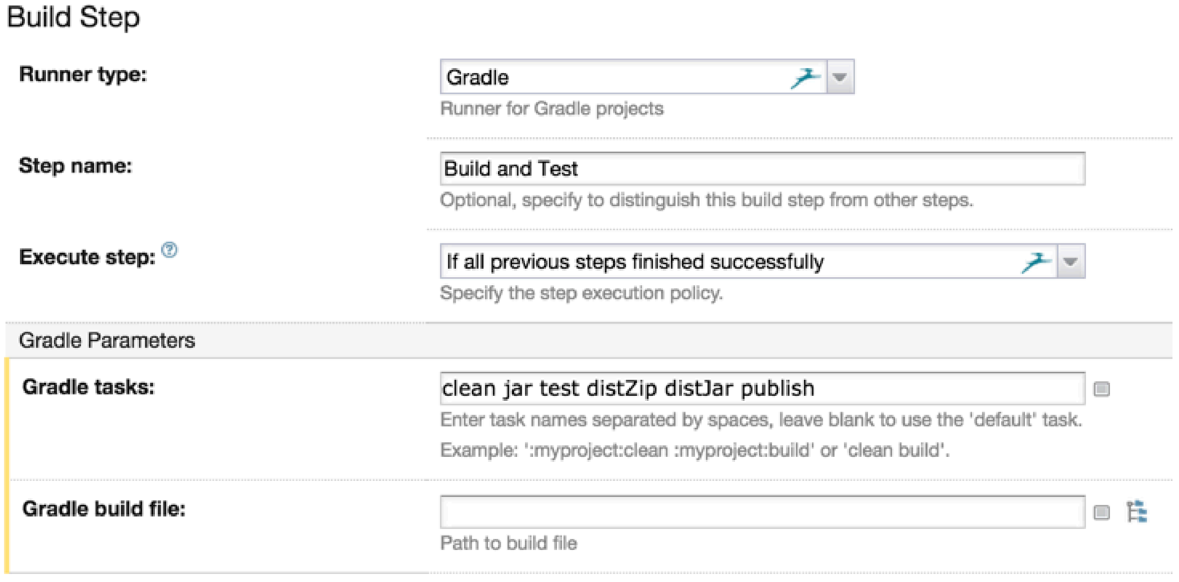
最终更新: 2024/09/10

在本章中, 你将学习如何设置 TeamCity (https://www.jetbrains.com/teamcity/) 来构建你的 Kotlin 项目. 关于 TeamCity 的基本知识和更多信息, 请参见 官方文档 (https://www.jetbrains.com/teamcity/documentation/), 其中包括如何安装, 基本配置, 等等.

Kotlin 可以使用不同的构建工具, 因此如果你在使用标准的构建工具, 比如 Ant, Maven 或 Gradle, 那么设置 Kotlin 项目的过程, 与这些工具集成的其他语言或库是相同的. TeamCity 也支持 IntelliJ IDEA 的内置构建系统, 使用时存在少量的要求和配置不同.

Gradle, Maven, 和 Ant

如果使用 Ant, Maven 或 Gradle, 设置过程很简单. 只需要定义构建步骤(Build Step). 比如, 如果使用 Gradle, 只需要直接定义需要的参数, 比如 Step Name, 以及对这个 Runner Type 需要执行的 Gradle tasks.



Gradle Build Step

由于 Kotlin 所有需要的依赖项都定义在 Gradle 文件中, 因此 Kotlin 不需要其他配置, 即可正确运行.

如果使用 Ant 或 Maven, 可以使用相同的配置. 唯一的区别是, Runner Type 应该是 Ant 或 Maven.

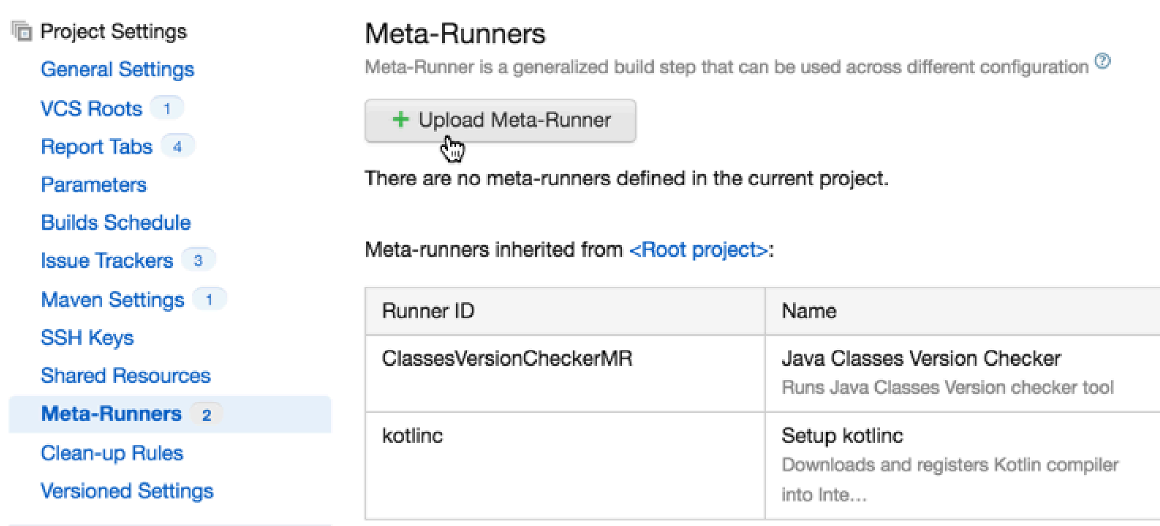
IntelliJ IDEA 构建 System

如果在 TeamCity 中使用 IntelliJ IDEA 构建系统, 请确认 IntelliJ IDEA 使用的 Kotlin 的版本与 TeamCity 运行的版本相同. 你可能需要下载 Kotlin plugin 的特定版本, 并安装到 TeamCity.

幸运的是, 可以使用一个元运行器(meta-runner), 它能够完成大部分的手工工作. 如果不熟悉 TeamCity 元运行器(meta-runner) 的概念, 请参见 文档 (https://www.jetbrains.com/help/teamcity/working-with-meta-runner.html). 元运行器是非常简单而且强大方法, 可以引入自定义的运行器, 而不需要编写插件.

下载并安装元运行器(meta-runner)

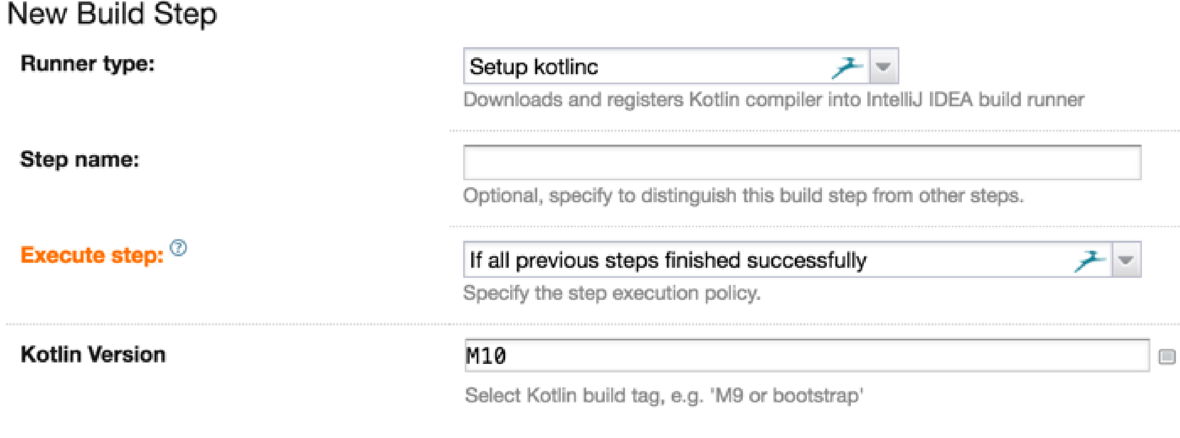
针对 Kotlin 的元运行器可从 GitHub (https://github.com/jonnyzzz/Kotlin.TeamCity) 找到. 请下载这个元运行器, 并从 TeamCity 用户界面导入它



元运行器

设置 Kotlin 编译器的获取步骤

这个步骤基本上只包括定义 Step Name, 以及你需要的 Kotlin 版本. 可以使用 Tag.

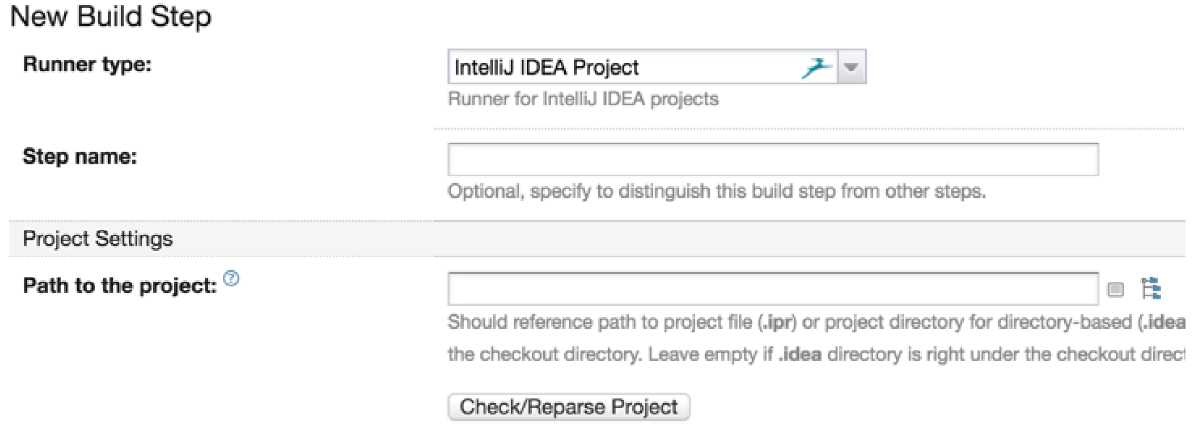


设置 Kotlin 编译器

运行器会根据 IntelliJ IDEA 项目的路径设置, 为属性 `system.path.macro.KOTLIN.BUNDLED` 设置正确的值. 但是, 这个值需要在 TeamCity 中定义 (并且可以设置为任意值). 因此, 你需要将它定义为一个系统变量.

设置 Kotlin 编译步骤

最终的步骤是定义项目的实际编译任务, 它使用标准的 IntelliJ IDEA Runner Type.



IntelliJ IDEA Runner

这样, 我们的项目现在可以构建, 并能够产生对应的 artifact 了.

其他 CI 服务器

如果使用 TeamCity 以外的其他持续集成(Continuous Integration)工具, 只要它支持构建工具的任何一种, 或者调用命令行工具, 编译 Kotlin 并对一些工作进行自动化, 使其成为 CI 过程的一部分, 都应该是可以实现的.

为 Kotlin 代码编写文档: KDoc

最终更新: 2024/09/10

为 Kotlin 代码编写文档使用的语言 (相当于 Java 中的 Javadoc) 称为 **KDoc**. 本质上, KDoc 结合了 Javadoc 和 Markdown, 它在块标签(block tag)使用 Javadoc 语法(但做了扩展, 以便支持 Kotlin 特有的概念), Markdown 则用来表示内联标记(inline markup).

> Kotlin 的文档引擎: Dokka, 它能够理解 KDoc, 而且可以用来生成各种不同格式的文档. 详情请参见我们的 Dokka 文档 (介绍).

KDoc 语法

与 Javadoc 一样, KDoc 以 `/**` 开始, 以 `*/` 结束. 文档中的每一行以星号开始, 星号本身不会被当作文档内容.

按照通常的习惯, 文档的第一段(直到第一个空行之前的所有文字)是对象元素的概要说明, 之后的内容则是详细说明.

每个块标签(block tag) 都应该放在新的一行内, 使用 `@` 字符起始.

下面的例子是使用 KDoc 对一个类标注的文档:

```
/**
 * 由多个 *成员* 构成的一个组.
 *
 * 这个类没有任何有用的逻辑; 只是一个文档的示例.
 *
 * @param T 组内成员的类型.
 * @property name 组的名称.
 * @constructor 创建一个空的组.
 */
class Group<T>(val name: String) {
    /**
     * 向组添加一个 [成员].
     * @return 添加之后的组大小.
     */
```

```
    fun add(member: T): Int { ... }
}
```

块标签(Block Tag)

KDoc 目前支持以下块标签:

@param name

对一个函数的参数, 或一个类, 属性, 或函数的类型参数标注文档. 如果你希望的话, 为了更好地区分参数名与描述文本, 可以将参数名放在方括号内. 所以下面两种语法是等价的:

```
@param name 描述.
@param[name] 描述.
```

@return

对函数的返回值标注文档.

@constructor

对类的主构造器(primary constructor)标注文档.

@receiver

对扩展函数的接受者(receiver)标注文档.

@property name

对类中指定名称的属性标注文档. 这个标签可以用来标注主构造器中定义的属性, 如果将文档放在主构造器的属性声明之前会很笨拙, 因此可以使用标签来对指定的属性标注文档.

@throws class, @exception class

对一个方法可能抛出的异常标注文档. 由于 Kotlin 中不存在受控异常(checked exception), 因此也并不要求对所有的异常标注文档, 但如果异常信息对类的使用者很有帮助的话, 你可以使用这个标签来标注异常信息.

@sample identifier

为了演示对象元素的使用方法, 可以使用这个标签将指定名称的函数体嵌入到文档内.

@see identifier

这个标签会在文档的 **See also** 部分, 添加一个指向某个类或方法的链接.

@author

标识对象元素的作者.

@since

标识对象元素最初引入这个软件时的版本号.

@suppress

将对象元素排除在文档之外. 有些元素, 不属于模块的正式 API 的一部分, 但站在代码的角度又需要被外界访问, 对这样的元素可以使用这个标签.

> KDoc 不支持 `@deprecated` 标签. 请使用 `@Deprecated` 注解来代替.

内联标记(Inline Markup)

对于内联标记(inline markup), KDoc 使用通常的 Markdown (https://daringfireball.net/projects/markdown/syntax) 语法, 但添加了一种缩写语法来生成指向代码内其他元素的链接.

指向元素的链接

要生成指向其他元素(类, 方法, 属性, 或参数)的链接, 只需要简单地将它的名称放在方括号内:

```
请使用 [foo] 方法来实现这个目的.
```

如果你希望对链接指定一个标签, 请使用 Markdown 参照风格(reference-style)语法:

```
请使用 [这个方法](foo) 来实现这个目的.
```

在链接中也可以使用带限定符的元素名称. 注意, 与 Javadoc 不同, 限定符的元素名称永远使用点号来分隔各个部分, 包括方法名称之前的分隔符, 也是点号:

```
请使用 [kotlin.reflect.KClass.properties] 来列举一个类的属性.
```

链接中的元素名称使用的解析规则, 与这个名称出现在对象元素之内时的解析规则一样. 具体来说, 如果你在当前源代码文件中导入(import)了一个名称, 那么在 KDoc 注释内使用它时, 就不必再指定完整的限定符了.

注意, KDoc 没有任何语法可以解析链接内出现的重载函数. 由于 Kotlin 的文档生成工具会将所有重载函数的文档放在同一个页面之内, 因此不必明确指定某一个具体的重载函数, 链接也可以正常工作.

外部链接

要添加外部链接, 请使用通常的 Markdown 语法:

```
关于 KDoc 语法的更多详情, 请参见 [KDoc](<example-URL>).
```

下一步做什么?

学习如何使用 Kotlin 的文档生成工具: Dokka (介绍).

Kotlin 与 OSGi

最终更新: 2024/09/10

要在你的项目中使用 Kotlin 的 OSGi (https://www.osgi.org/) 支持功能, 需要使用 `kotlin-osgi-bundle`, 而不是通常的 Kotlin 库文件. 此外还建议你删除 `kotlin-runtime`, `kotlin-stdlib` 和 `kotlin-reflect` 依赖, 因为 `kotlin-osgi-bundle` 已经包含了这些库的内容. 此外还需要注意不要引用外部的 Kotlin 库文件. 大多数通常的 Kotlin 库依赖都不能用于 OSGi 环境, 因此你不应该使用它们, 要将它们从你的工程中删除.

Maven

在 Maven 工程中引入 Kotlin OSGi bundle:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-osgi-bundle</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

从外部库中删除 Kotlin 的标准库(注意, exclusion 设置中星号只在 Maven 3 中有效):

```
<dependency>
    <groupId>some.group.id</groupId>
    <artifactId>some.library</artifactId>
    <version>some.library.version</version>

    <exclusions>
        <exclusion>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Gradle

在 Gradle 工程中引入 `kotlin-osgi-bundle`:

Kotlin

```
dependencies {
    implementation(kotlin("osgi-bundle"))
}
```

Groovy

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-osgi-
bundle:1.9.23"
}
```

通过传递依赖, 你可能会间接依赖到一些默认的 Kotlin 库, 你可以使用以下方法删除这些库:

Kotlin

```
dependencies {
    implementation("some.group.id:some.library:someversion") {
        exclude(group = "org.jetbrains.kotlin")
    }
}
```

Groovy

```
dependencies {
    implementation('some.group.id:some.library:someversion') {
        exclude group: 'org.jetbrains.kotlin'
```

```
    }
}
```

FAQ

为什么不直接向所有的 Kotlin 库添加需要的 manifest 设值呢?

虽然这是提供 OSGi 支持时最优先的方法, 但很不幸, 目前我们无法做到这一点, 原因是所谓的 "包分裂(package split)" 问题 (http://wiki.osgi.org/wiki/Split_Packages), 这个问题很难解决, 所以目前我们不打算进行这样巨大的变更. 另外还有一种 `Require-Bundle` 功能, 但也不是最好的选择, 而且并不推荐采用这种方案. 因此我们决定为 OSGi 创建一个独立的库文件.

Kotlin 命令行编译器

最终更新: 2024/09/10

每个 Kotlin 发布版都带有独立版本的编译器. 你可以手动下载最新版本, 也可以使用包管理器下载.

> 要使用 Kotlin, 安装命令行编译器并不是必须的. 编写 Kotlin 应用程序的通常方式是使用 IDE - IntelliJ IDEA (https://www.jetbrains.com/idea/) 或 Android Studio (https://developer.android.com/studio). IDE 已经提供了完全的 Kotlin 支持, 不需要添加其它组件. 更多详情请参见 在 IDE 中使用 Kotlin (Kotlin 入门).

安装编译器

手动安装

手动安装 Kotlin 编译器的步骤:

1. 从 GitHub Release 页面 (https://github.com/JetBrains/kotlin/releases/tag/v1.9.23) 下载最新版 (`kotlin-compiler-1.9.23.zip`).
2. 将独立版本的编译器解包到一个目录, 并将 `bin` 目录添加到系统的 path 设定中(可选). `bin` 目录包含在 Windows, macOS, 和 Linux 上编译和运行 Kotlin 所需要的脚本.

> 对于想要使用命令行编译器的 Windows 用户, 我们推荐使用手动安装的方法.

使用 SDKMAN! 安装

在基于 UNIX 的系统中, 比如 macOS, Linux, Cygwin, FreeBSD, 以及 Solaris, 安装 Kotlin 的更简单的方法是 SDKMAN! (https://sdkman.io). 它也能用于 Bash 和 ZSH shell. 参见 如何安装 SDKMAN! (https://sdkman.io/install).

要通过 SDKMAN! 安装 Kotlin 编译器, 请在终端中运行以下命令:

```
sdk install kotlin
```

使用 Homebrew 安装

另一种方法是, 在 macOS 上你可以通过 Homebrew (https://brew.sh/) 来安装编译器:

```
brew update
brew install kotlin
```

使用 Snap 包安装

如果你在 Ubuntu 16.04 或更高版本上使用 Snap (https://snapcraft.io/), 你可以通过命令行安装编译器:

```
sudo snap install --classic kotlin
```

创建并运行应用程序

1. 创建一个简单的 Kotlin 应用程序, 它输出 `"Hello, World!"`. 使用你喜欢的比较器, 创建一个名为 `hello.kt` 的新文件, 内容如下:

   ```
   fun main() {
       println("Hello, World!")
   }
   ```

2. 使用 Kotlin 编译器编译应用程序:

   ```
   kotlinc hello.kt -include-runtime -d hello.jar
   ```

   `-d` 选项指定生成的类文件的输出路径, 可以是一个目录或一个 *.jar* 文件. `-include-runtime` 选项让生成的 *.jar* 文件称为自包含文件, 其中包含了 Kotlin 运行期库, 因此可以独立运行.

   要查看所有可用的选项, 请运行

   ```
   kotlinc -help
   ```

3. 运行应用程序.

   ```
   java -jar hello.jar
   ```

编译一个库

如果你在开发库, 供使用其他 Kotlin 应用程序使用, 那么构建 *.jar* 文件时可以不包含 Kotlin 运行库:

```
kotlinc hello.kt -d hello.jar
```

由于这种方式编译的二进制文件依赖于 Kotlin 运行库, 当你编译的库被使用时, 你需要确保 Kotlin 运行库存在于类路径中.

你也可以使用 `kotlin` 脚本来运行 Kotlin 编译器生成的二进制文件:

```
kotlin -classpath hello.jar HelloKt
```

`HelloKt` 是 Kotlin 编译器为 `hello.kt` 文件生成的 main 类名.

运行 REPL

你可以不带任何参数来运行编译器, 启动一个交互环境. 在这个环境中, 你可以输入任何有效的 Kotlin 代码, 并看到结果.

```
[Ocean] ~/tutorials/kotlin/command_line/kotlinc$ bin/kotlinc-jvm
Kotlin interactive shell
Type :help for help, :quit for quit
>>> 2+2
4
>>> println("Welcome to the Kotlin Shell")
Welcome to the Kotlin Shell
>>>
```

Shell

运行脚本

Kotlin 也可以用作脚本语言. 一个脚本就是一个 Kotlin 源代码文件 (`.kts`), 其中包含顶级(Top-Level)可执行代码.

```
import java.io.File

// 得到参数传入的路径, 也就是 "-d some/path", 或使用当前路径.
val path = if (args.contains("-d")) args[1 + args.indexOf("-d")]
           else "."
```

```
val folders = File(path).listFiles { file -> file.isDirectory() }
folders?.forEach { folder -> println(folder) }
```

要运行一个脚本, 请向编译器传递 `-script` 选项, 加上对应的脚本文件:

```
kotlinc -script list_folders.kts -- -d <path_to_folder_to_inspect>
```

Kotlin 支持脚本的自定义功能(实验性功能), 例如添加外部属性, 提供静态或动态依赖项, 等等. 自定义通过所谓的 *脚本定义(Script Definition)* 来定义 - 它是带注解的 Kotlin 类, 带有适当的支持代码. 通过脚本文件名称扩展来选择适当的定义. 详情请参见 Kotlin 自定义脚本 (教程 - Kotlin 自定义脚本(Custom Scripting) 入门).

在编译类路径中包含了正确的 jar 文件时, 会自动检测并使用适当准备的脚本定义. 或者, 你也可以向编译器传递 `-script-templates` 选项, 手动指定脚本定义:

```
kotlinc -script-templates org.example.CustomScriptDefinition -script
custom.script1.kts
```

更多详情请参见 KEEP-75
(https://github.com/Kotlin/KEEP/blob/master/proposals/scripting-support.md).

Kotlin 编译器选项

最终更新: 2024/09/10

Kotlin 的各个发布版都带有针对各种编译目标的编译器: JVM, JavaScript, 以及 所支持的各种平台 ("目标平台" in "使用 Kotlin/Native 进行原生(Native)程序开发") 的原生二进制可执行文件(native binary).

这些编译器会在以下情况下使用:

- 当你对你的 Kotlin 工程按下 **Compile** 或 **Run** 按钮时, 由 IDE 使用.
- 当你在控制台或在 IDE 内调用 `gradle build` 命令时, 由 Gradle 使用.
- 当你在控制台或在 IDE 内调用 `mvn compile` 或 `mvn test-compile`, 由 Maven 使用.

你也可以从命令行手动运行 Kotlin 编译器, 详情请参见教程 使用命令行编译器 (Kotlin 命令行编译器).

编译器选项

Kotlin 编译器带有很多选项, 用于控制编译过程. 本章会列出针对各种编译目标的编译器选项, 并分别进行介绍.

有几种方式来设置各个编译器选项, 以及相应的值(即 *编译参数(compiler argument)*):

- 在 IntelliJ IDEA 中, 可以在 **Settings/Preferences** | **Build, Execution, Deployment** | **Compiler** | **Kotlin Compiler** 设定窗口的 **Additional command line parameters** 文本框中输入编译器参数
- 如果使用 Gradle, 可以在 Kotlin 编译任务的 `compilerOptions` 属性中指定编译参数. 详情请参见 Gradle 编译器选项 ("如何定义编译器选项" in "Kotlin Gradle plugin 中的编译器选项").
- 如果使用 Maven, 可以在 Maven 插件的 `<configuration>` 元素中指定编译参数 . 详情请参见 Maven ("指定编译器选项" in "Maven").
- 如果在命令行运行编译器, 可以在调用编译器时直接添加编译参数, 或者将编译参数写在 参数文件 内.

  例如:

```
$ kotlinc hello.kt -include-runtime -d hello.jar
```

在 Windows 上, 如果传递的编译器参数中包含分隔字符(空格, `=`, `;`, `,`), 请将这些参数值使用双引号(`"`)括起.

```
$ kotlinc.bat hello.kt -include-runtime -d "My
Folder\hello.jar"
```

各平台共通选项

下面是所有 Kotlin 编译器的共通选项.

-version

显示编译器版本.

-nowarn

进制编译器在编译过程中显示警告信息.

-Werror

将警告信息变为编译错误.

-verbose

允许输出最详细的 log, 其中包括编译过程的各种细节信息.

-script

运行 Kotlin 脚本文件. 使用这个选项调用编译器时, 编译器会运行参数中指定的第一个 Kotlin 脚本文件(`*.kts`).

-help (-h)

显示编译器使用方法的帮助信息, 然后退出. 帮助信息中只会显示标准的编译选项. 如果需要显示更多的高级编译选项, 请使用 `-X` 参数.

-X

显示编译器高级选项的帮助信息, 然后退出. 这些选项目前还不稳定: 选项的名称和行为都有可能变更, 并且不会有相关公告.

-kotlin-home path

对 Kotlin 编译器指定一个自定义的路径, 用来查找运行时期的库文件.

-P plugin:pluginId:optionName=value

向 Kotlin 编译器插件传递一个选项. 相关的编译器插件, 以及它们的选项, 请参见本文档的 **工具** > **编译器插件** 章节.

-language-version version

与指定的 Kotlin 版本保持源代码级兼容.

-api-version version

允许使用从指定的 Kotlin 版本的库才开始提供的 API 声明.

-progressive

允许编译器使用 渐进模式(progressive mode) ("渐进模式" in "Kotlin 1.3 版中的新功能").

在渐进模式下, 对不稳定代码中功能废弃和 bug 修正, 会立即生效, 而不会等待完整的版本迁移周期完成. 渐进模式下编写的代码可以向后兼容(backwards compatible); 但是, 非渐进模式下编写的代码, 在渐进模式下编译时, 可能导致编译错误.

@argfile

从指定的文件中读取编译器选项. 这样的文件可以包含编译器选项, 相应的值, 以及源代码文件的路径. 选项和文件路径使用空格分隔. 比如:

```
-include-runtime -d hello.jar
hello.kt
```

如果想要传递的参数值本身包含空格, 请使用单引号 (') 或双引号 (") 将参数值括起. 如果参数值本身包含引号, 请使用反斜线 (\) 转义符表示.

```
-include-runtime -d 'My folder'
```

也可以传递多个参数文件, 比如, 如果想要将编译器选项和源代码文件分开的情况.

```
$ kotlinc @compiler.options @classes
```

如果文件位置不在当前目录下, 请使用相对路径.

```
$ kotlinc @options/compiler.options hello.kt
```

-opt-in annotation

指定注解的全限定名称, 通过这个注解启用 明确要求使用者同意(opt-in) ([明确要求使用者同意的功能(Opt-in Requirement)](...)) API.

Kotlin/JVM 编译器选项

针对 JVM 平台的 Kotlin 编译器将 Kotlin 源代码文件编译为 Java class 文件. 将 Kotlin 文件编译到 JVM 平台的命令行工具是 `kotlinc` 和 `kotlinc-jvm`. 也可以使用它们来运行 Kotlin 脚本文件.

除 共通选项 之外, Kotlin/JVM 编译器还支持以下选项.

-classpath path (-cp path)

在指定的路径中查找 class 文件. 如果 classpath 中存在多个路径, 请使用操作系统的路径分隔符来分隔(对 Windows 系统是 ;, 对 macOS/Linux 系统是 :). classpath 可以包含文件路径, 目录路径, ZIP 文件, 或 JAR 文件.

-d path

将生成的 class 文件输出到指定的位置. 输出位置可以是一个目录, 一个 ZIP 文件, 或一个 JAR 文件.

-include-runtime

将 Kotlin 运行时库文件包含在最终输出的结果 JAR 文件中. 这样将使得最终输出的包可以在任何安装了 Java 环境中运行.

-jdk-home path

如果自定义的 JDK home 目录与默认的 `JAVA_HOME` 不用, 这个选项会将它添加到 classpath 中.

-Xjdk-release=version

指定生成的 JVM 字节码的目标版本. 将类路径中的 JDK API 限制为指定的 Java 版本. 自动设置 `-jvm-target version`. 可以指定的值是 `1.8`, `9`, `10`, ..., `21`. 默认值是 `{{ site.data.releases.defaultJvmTargetVersion }}`.

> 这个选项 不保证 (https://youtrack.jetbrains.com/issue/KT-29974) 对所有的 JDK 发布版都有效.

-jvm-target version

指定编译产生的 JVM 字节码(bytecode)版本. 可以指定的值是 `1.8`, `9`, `10`, ..., `21`. 默认值是 `{{ site.data.releases.defaultJvmTargetVersion }}`.

-java-parameters

针对 Java 1.8 的方法参数反射(reflection on method parameter)生成元信息(metadata). 译者注: 等于 Java 1.8 编译参数 `-parameters`, 参见 javac 命令行编译器 (https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html)

-module-name name (JVM)

对编译产生的 `.kotlin_module` 指定一个自定义的名称.

-no-jdk (JVM)

不要自动将 Java 运行时期库文件添加到 classpath 中.

-no-reflect

不要自动将 Kotlin 反射库文件(`kotlin-reflect.jar`) 添加到 classpath 中.

-no-stdlib

不要自动将 Kotlin/JVM 标准库文件(`kotlin-stdlib.jar`) 和 Kotlin 反射库文件(`kotlin-reflect.jar`) 添加到 classpath 中.

-script-templates classnames[,]

脚本定义的模板类. 请使用类的完全限定名称, 如果有多个, 请使用逗号(,) 分隔.

Kotlin/JS 编译器选项

针对 JS 平台的 Kotlin 编译器将 Kotlin 源代码文件编译为 JavaScript 代码. 将 Kotlin 文件编译到 JS 平台的命令行工具是 `kotlinc-js`.

除 共通选项 之外, Kotlin/JS 编译器还支持以下选项.

-libraries path

包含 `.meta.js` 和 `.kjsm` 文件的 Kotlin 库路径, 如果有多个路径, 请使用操作系统的路径分隔符分隔.

-main {call|noCall}

指定执行时是否要调用 `main` 函数.

-meta-info

生成 `.meta.js` 和 `.kjsm` 文件时附带元信息(metadata). 开发 JS 库时, 请使用这个选项 .

-module-kind {umd|commonjs|amd|plain}

指定编译器生成的 JS 模块类型:

- `umd` - 统一模块定义(Universal Module Definition) (https://github.com/umdjs/umd) 模块
- `commonjs` - CommonJS (http://www.commonjs.org/) 模块
- `amd` - 异步模块定义(Asynchronous Module Definition) (https://en.wikipedia.org/wiki/Asynchronous_module_definition) 模块
- `plain` - 普通 JS 模块

关于各种 JS 模块类型, 以及它们之间的差别, 请参见 这篇文章 (https://www.davidbcalhoun.com/2014/what-is-amd-commonjs-and-umd/).

-no-stdlib (JS)

不要自动将默认的 Kotlin/JS 标准库添加到编译依赖中.

-output filepath

指定编译结果的输出目标文件. 参数值必须是一个 `.js` 文件路径, 包含文件名.

-output-postfix filepath

将指定文件的内容添加到编译输出文件的末尾部分.

-output-prefix filepath

将指定文件的内容添加到编译输出文件的先头部分.

-source-map

生成源代码映射文件(source map).

-source-map-base-dirs path

使用指定的路径作为起始目录(base directory). 起始目录用来计算源代码映射文件(source map)中的相对路径.

-source-map-embed-sources {always|never|inlining}

是否将源代码文件嵌入到源代码映射文件(source map)中.

-source-map-names-policy {simple-names|fully-qualified-names|no}

将你在 Kotlin 代码中声明的变量和函数名称添加到源代码映射文件(source map)中.

设置	说明	输出示例
simple-names	添加变量名称和函数的简单名称. (默认值)	main
fully-qualified-names	添加变量名称和函数完全限定名称.	com.example.kjs.playground.main
no	不添加变量名称和函数名称.	无

-source-map-prefix

向源代码映射文件(source map)中的路径添加指定的前缀.

Kotlin/Native 编译器选项

Kotlin/Native 编译器将 Kotlin 源代码文件编译为 所支持的各种平台 ("目标平台" in "使用 Kotlin/Native 进行原生(Native)程序开发") 的二进制可执行文件(native binary). Kotlin/Native 编译的命令行工具是 kotlinc-native.

除 共通选项 之外, Kotlin/Native 编译器还支持以下选项.

-enable-assertions (-ea)

在生成的代码中允许运行时断言(runtime assertion).

-g

允许编译产生 debug 信息.

-generate-test-runner (-tr)

生成一个应用程序, 用于在工程中运行单元测试.

-generate-no-exit-test-runner (-trn)

生成一个应用程序, 用于运行单元测试, 但不会有明确的进程结束信息(explicit process exit).

-include-binary path (-ib path)

将外部的二进制文件打包到编译产生的 klib 文件内.

-library path (-l path)

链接指定的库文件. 关于在 Kotlin/native 工程中如何使用库, 请参见 Kotlin/Native 库 (Kotlin/Native 库).

-library-version version (-lv version)

指定库的版本.

-list-targets

列出可用的硬件目标平台(hardware target).

-manifest path

指定一个 manifest 补充文件.

-module-name name (Native)

为编译产生的模块指定名称. 这个选项也可以用来对导出给 Objective-C 的声明指定名称前缀: 怎样为 Kotlin 框架指定自定义的 Objective-C 前缀? ("怎样为 Kotlin 框架指定自定义的 Objective-C 前缀或名称?" in "Kotlin/Native FAQ")

-native-library path (-nl path)

包含原生的 bitcode 库文件.

-no-default-libs

不要将用户代码与编译器附带的 默认的平台库文件 (平台库) 链接.

-nomain

假定外部的库文件会提供应用程序启动时的 `main` 入口点(entry point).

-nopack

不要将库文件打包进入 klib 文件.

-linker-option

在二进制文件构建过程中, 向链接程序传递一个参数. 这个选项可以用来链接到某些原生库文件.

-linker-options args

在二进制文件构建过程中, 向链接程序传递多个参数. 参数之间用空格分隔.

-nostdlib

不要链接到标准库.

-opt

允许编译优化(compilation optimization).

-output name (-o name)

指定编译输出文件的名称.

-entry name (-e name)

指定入口点的限定名称(qualified entry point name).

-produce output (-p output)

指定编译输出文件的类型:

- `program`
- `static`
- `dynamic`
- `framework`
- `library`
- `bitcode`

-repo path (-r path)

库文件的搜索路径. 详情请参见, 库的查找顺序 ("库的查找顺序" in "Kotlin/Native 库").

-target target

指定编译的硬件目标平台(hardware target). 要查看可选择的硬件目标平台, 请使用 `-list-targets` 选项.

All-open 编译器插件

最终更新: 2024/09/10

Kotlin 的类和成员默认都是 final 的, 但有些框架和库, 比如 Spring AOP, 需要类是 open 的, 因此造成一些不便. all-open 编译器插件会调整 Kotlin 类, 以这些框架的需求, 它会将标记了特定注解的类及其成员变为 open , 而不需要在代码中明确标记 open 关键字.

比如, 当你使用 Spring 时, 你不需要所有的类都变为 open, 只需要标注了特定注解的类, 比如 @Configuration 或 @Service. all-open 插件允许你指定这样的注解.

Kotlin 为 all-open 插件提供了 Gradle 和 Maven 支持, 并带有完整的 IDE 集成.

> 对于 Spring, 你可以使用 kotlin-spring 编译器插件.

Gradle

在你的 build.gradle(.kts) 文件中添加插件:

Kotlin

```
plugins {
    kotlin("plugin.allopen") version "1.9.23"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.allopen" version "1.9.23"
}
```

然后指定需要将类变为 open 的注解:

Kotlin

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation",
"com.third.Annotation")
}
```

Groovy

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation",
"com.third.Annotation")
}
```

如果类 (或它的任何超类) 标注了 `com.my.Annotation` 注解, 那么类本身和它的成员都会变为 open.

对元注解(meta-annotation)同样有效:

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // all-open 插件也会将这个类变为 open
```

`MyFrameworkAnnotation` 标注了 all-open 元注解 `com.my.Annotation`, 因此它也成为一个 all-open 注解.

Maven

在你的 `pom.xml` 文件中添加插件:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>
```

```xml
    <configuration>
        <compilerPlugins>
            <!-- 或者使用 "spring", 支持 Spring -->
            <plugin>all-open</plugin>
        </compilerPlugins>

        <pluginOptions>
            <!-- 每个注解放在单独的行 -->
            <option>all-open:annotation=com.my.Annotation</option>
            <option>all-
open:annotation=com.their.AnotherAnnotation</option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-allopen</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

关于 all-open 注解的工作方式, 详情请参见 Gradle 小节.

Spring 支持

如果你使用 Spring, 你可以启用 `kotlin-spring` 编译器插件, 而不必手动指定 Spring 注解. `kotlin-spring` 是对 `all-open` 的一个上层封装, 它的工作方式完全相同.

在你的 `build.gradle(.kts)` 文件中添加 `spring` 插件:

Kotlin

```kotlin
plugins {
    id("org.jetbrains.kotlin.plugin.spring") version "1.9.23"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.spring" version "1.9.23"
}
```

在 Maven 中, `spring` 插件由 `kotlin-maven-allopen` 插件依赖项提供, 因此在你的`pom.xml` 文件中要这样启用它:

```xml
<compilerPlugins>
    <plugin>spring</plugin>
</compilerPlugins>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-allopen</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

这个插件指定了以下注解:

- `@Component` (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html)
- `@Async` (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/annotation/Async.html)
- `@Transactional` (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html)
- `@Cacheable` (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html)
- `@SpringBootTest` (https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/context/SpringBootTest.html)

得益于元注解功能, 由注解 `@Configuration` (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html), `@Controller` (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Controller.html), `@RestController` (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html), `@Service` (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Service.html) 或 `@Repository` (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html) 标注的类, 都会自动变为 open, 因为这些注解都标注了元注解 `@Component` (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html).

当然, 你也可以在同一个项目内同时使用 `kotlin-allopen` 和 `kotlin-spring`.

> 如果你使用 start.spring.io (https://start.spring.io/#!language=kotlin) 服务生成项目模板, `kotlin-spring` 插件默认会被启用.

命令行编译器

All-open 编译器插件的 JAR 文件 存在于 Kotlin 编译器的二进制发布包中. 你可以使用 `-Xplugin` kotlinc 选项指定它的 JAR 文件路径, 来添加这个插件:

```
-Xplugin=$KOTLIN_HOME/lib/allopen-compiler-plugin.jar
```

可以使用 `annotation` 插件选项直接指定 all-open 注解, 或者启用 *预设置(preset)*:

```
# 插件选项格式是: "-P plugin:<plugin id>:<key>=<value>".
# 选项可以重复.

-P plugin:org.jetbrains.kotlin.allopen:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.allopen:preset=spring
```

`all-open` 插件可以使用的预设置是: `spring`, `micronaut`, 和 `quarkus`.

No-arg 编译器插件

最终更新: 2024/09/10

对带有指定注解的类, *no-arg* 编译器插件会为它生成一个额外的无参数构造器.

生成的构造器是合成的(Synthetic), 因此不能在 Java 或 Kotlin 代码中直接调用, 但可以使用反射调用.

这个功能使得 Java Persistence API (JPA) 可以创建类的实例, 即使从 Kotlin 或 Java 的观点看, 它并没有无参数的构造器 (参见 下文 关于 `kotlin-jpa` 插件的介绍).

Gradle

添加插件, 并指定需要为类生成无参数构造器的注解.

Kotlin

```
plugins {
    kotlin("plugin.noarg") version "1.9.23"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.noarg" version "1.9.23"
}
```

然后指定 no-arg 注解:

```
noArg {
    annotation("com.my.Annotation")
}
```

如果你希望插件在合成的(Synthetic)构造器中执行初始化逻辑, 请打开 `invokeInitializers` 选项. 这个选项默认关闭.

```
noArg {
    invokeInitializers = true
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- 对 JPA 的情况请使用 "jpa" 插件 -->
            <plugin>no-arg</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>no-arg:annotation=com.my.Annotation</option>
            <!-- 在合成的(Synthetic)构造器中调用实例的初始化代码 -->
            <!-- <option>no-arg:invokeInitializers=true</option> -->
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-noarg</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

JPA 支持

和 `kotlin-spring` 插件封装了 `all-open` 一样, `kotlin-jpa` 也是 `no-arg` 的上层封装. 这个插件自动指定 *no-arg* 注解为 `@Entity`

(https://docs.oracle.com/javaee/7/api/javax/persistence/Entity.html), @Embeddable (https://docs.oracle.com/javaee/7/api/javax/persistence/Embeddable.html), 和 @MappedSuperclass (https://docs.oracle.com/javaee/7/api/javax/persistence/MappedSuperclass.html).

在 Gradle 中添加这个插件的方法如下:

Kotlin

```
plugins {
    kotlin("plugin.jpa") version "1.9.23"
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.jpa" version "1.9.23"
}
```

在 Maven 中, 启用 jpa 插件:

```
<compilerPlugins>
    <plugin>jpa</plugin>
</compilerPlugins>
```

命令行编译器

将插件的 JAR 文件添加到编译器的插件 classpath, 并指定需要处理的注解, 或使用预设定(preset):

```
-Xplugin=$KOTLIN_HOME/lib/noarg-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.noarg:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.noarg:preset=jpa
```

SAM-with-receiver 编译器插件

最终更新: 2024/09/10

sam-with-receiver 编译器插件可以将带注解的 Java "single abstract method" (SAM) 接口方法的第一个参数变成 Kotlin 中的接受者. 只有在使用 SAM 适配器(Adapter) 和 SAM 构造器(Constructor), 将 Kotlin Lambda 表达式作为 SAM 接口传递时, 这个变换才有效. (详情请参见 SAM 变换文档 ("SAM 转换" in "在 Kotlin 中调用 Java 代码")).

下面是一段示例:

```
public @interface SamWithReceiver {}

@SamWithReceiver
public interface TaskRunner {
    void run(Task task);
}
```

```
fun test(context: TaskContext) {
    val runner = TaskRunner {
        // 这里的 'this' 是一个 'Task' 实例

        println("$name is started")
        context.executeTask(this)
        println("$name is finished")
    }
}
```

Gradle

使用方法与 all-open (All-open 编译器插件) 插件和 no-arg (No-arg 编译器插件) 插件相同, 区别是 sam-with-receiver 没有任何预定义, 因此你需要自己指定需要特别处理的注解.

Kotlin

```
plugins {
    kotlin("plugin.sam.with.receiver") version "1.9.23"
```

```
}
```

Groovy

```
plugins {
    id "org.jetbrains.kotlin.plugin.sam.with.receiver" version
"1.9.23"
}
```

然后指定需要特别处理的 SAM-with-receiver 注解:

```
samWithReceiver {
    annotation("com.my.SamWithReceiver")
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <plugin>sam-with-receiver</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>
                sam-with-receiver:annotation=com.my.SamWithReceiver
            </option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
```

```xml
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-sam-with-receiver</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

命令行编译器

将插件的 JAR 文件添加到编译器的插件 classpath, 并指定需要处理的 sam-with-receiver 注解:

```
-Xplugin=$KOTLIN_HOME/lib/sam-with-receiver-compiler-plugin.jar
-P
plugin:org.jetbrains.kotlin.samWithReceiver:annotation=com.my.SamWit
hReceiver
```

kapt 编译器插件

最终更新: 2024/09/10

> kapt 已进入维护模式. 我们会继续保证它兼容最新版的 Kotlin 和 Java, 但不会再实现新的功能特性. 请改用 Kotlin 符号处理(Symbol Processing) API (KSP) (Kotlin 符号处理(Kotlin Symbol Processing) API) 来处理注解. 详情请参见 KSP 支持的注解库列表 ("支持的库" in "Kotlin 符号处理(Kotlin Symbol Processing) API").

Kotlin 使用 *kapt* 编译器插件来支持注解处理器(参见 JSR 269 (https://jcp.org/en/jsr/detail?id=269)). 译注: kapt 是 "Kotlin annotation processing tool" 的缩写

简单地说, 你可以在 Kotlin 项目中使用 Dagger (https://google.github.io/dagger/) 或 Data Binding (https://developer.android.com/topic/libraries/data-binding/index.html) 之类的库.

关于如何在你的 Gradle/Maven 编译脚本中使用 *kapt* 插件, 请阅读下文.

在 Gradle 中使用

执行以下步骤:

1. 应用 `kotlin-kapt` Gradle plugin:

   Kotlin

   ```
   plugins {
       kotlin("kapt") version "1.9.23"
   }
   ```

   Groovy

   ```
   plugins {
       id "org.jetbrains.kotlin.kapt" version "1.9.23"
   }
   ```

2. 在你的 `dependencies` 块中使用 `kapt` 配置来添加对应的依赖:

Kotlin

```kotlin
dependencies {
    kapt("groupId:artifactId:version")
}
```

Groovy

```groovy
dependencies {
    kapt 'groupId:artifactId:version'
}
```

3. 如果你以前对注解处理器使用过 Android support (https://developer.android.com/studio/build/gradle-plugin-3-0-0-migration.html#annotationProcessor_config), 请将使用 `annotationProcessor` 配置的地方替换为 `kapt`. 如果你的工程中包含 Java 类, `kapt` 也会正确地处理这些 Java 类.

   如果你需要对 `androidTest` 或 `test` 源代码使用注解处理器, 那么与 `kapt` 配置相对应的名称应该是 `kaptAndroidTest` 和 `kaptTest`. 注意, `kaptAndroidTest` 和 `kaptTest` 从 `kapt` 继承而来, 因此你只需要提供 `kapt` 的依赖项, 它可以同时用于产品代码和测试代码.

试用 Kotlin K2 编译器

> ⚠ kapt 编译器插件对 K2 编译器的支持是 实验性功能 (Kotlin 各部分组件的稳定性). 需要使用者同意(Opt-in) (详情见下文), 你应该只为评估和试验目的来使用这个功能.

从 Kotlin 1.9.20 开始, 你可以对 K2 编译器 (https://blog.jetbrains.com/kotlin/2021/10/the-road-to-the-k2-compiler/) 试用 kapt 编译器插件, K2 编译器带来了性能改进和很多其它优点. 要在你的项目中使用 K2 编译器, 请在你的 `gradle.properties` 文件中添加以下选项:

```
kotlin.experimental.tryK2=true
kapt.use.k2=true
```

或者, 你也可以通过以下步骤, 对 kapt 启用 K2 编译器:

1. 在你的 `build.gradle.kts` 文件中, 将 语言版本 ("languageVersion 设置示例" in "Kotlin Gradle plugin 中的编译器选项") 设置为 `2.0`.
2. 在你的 `gradle.properties` 文件中, 添加 `kapt.use.k2=true`.

如果你在对 K2 编译器使用 kapt 插件时遇到任何问题, 请报告到我们的 问题追踪系统 (http://kotl.in/issue).

注解处理器的参数

可以使用 `arguments {}` 代码段来传递参数给注解处理器:

```
kapt {
    arguments {
        arg("key", "value")
    }
}
```

支持 Gradle 编译缓存

kapt 注解处理任务默认情况下不会 被 Gradle 缓存 (https://guides.gradle.org/using-build-cache/). 因为注解处理器可以运行任意代码, 并不一定只是将编译任务的输入文件转换为输出文件, 它还可能访问并修改未被 Gradle 追踪的其他文件. 如果确实需要为 kapt 启用 Gradle 编译缓存, 请将以下代码加入到你的编译脚本中:

```
kapt {
    useBuildCache = false
}
```

改进使用 kapt 时的构建速度

并行运行多个 KAPT 任务

为了改进使用 kapt 时的构建速度, 你可以对 kapt 任务启用 Gradle Worker API (https://guides.gradle.org/using-the-worker-api/). 使用 Worker API 可以让 Gradle 并行运行单个项目中的多个独立的注解处理任务, 某些情况下能够显著缩短运行时间.

如果在 Kotlin Gradle 插件中使用了 自定义 JDK home ("Gradle Java 工具链支持" in "配置 Gradle 项目") 功能, kapt 任务执行器只会使用 进程隔离模式 (https://docs.gradle.org/current/userguide/worker_api.html#changing_the_isolation_mode). 注意, `kapt.workers.isolation` 属性会被忽略.

如果你想要对 kapt worker 进程指定额外的 JVM 参数, 请使用 `KaptWithoutKotlincTask` 的输入参数 `kaptProcessJvmArgs`:

Kotlin

```
tasks.withType<org.jetbrains.kotlin.gradle.internal.KaptWithoutK
otlincTask>()
    .configureEach {
        kaptProcessJvmArgs.add("-Xmx512m")
    }
```

Groovy

```
tasks.withType(org.jetbrains.kotlin.gradle.internal.KaptWithoutK
otlincTask.class)
    .configureEach {
        kaptProcessJvmArgs.add('-Xmx512m')
    }
```

注解处理器的 classloader 缓存

> ⚠ 在 kapt 中, 注解处理器的 classloader 缓存是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-28901) 提供你的反馈意见.

如果连续执行很多 Gradle 任务, 注解处理器的 classloader 缓存功能可以帮助 kapt 提高运行速度.

要启用这个功能, 可以在你的 `gradle.properties` 文件中使用以下属性:

```
# 正数值会启用缓存功能
# 请在这里指定与使用 kapt 的模块数相同的数字
```

```
kapt.classloaders.cache.size=5

# 为让缓存正确工作, 需要关闭这个设定
kapt.include.compile.classpath=false
```

如果你遇到与注解处理器缓存相关的问题, 可以对这些处理器关闭缓存:

```
# 在这里指定注解处理器的完整名称, 可以对这些处理器关闭缓存
kapt.classloaders.cache.disableForProcessors=[注解处理器的完整名称]
```

测量注解处理器的性能

可以使用 `-Kapt-show-processor-timings` plugin 选项得到注解处理器执行时的性能统计. 输出示例:

```
Kapt Annotation Processing performance report:
com.example.processor.TestingProcessor: total: 133 ms, init: 36 ms, 2 round(s): 97 ms, 0 ms
com.example.processor.AnotherProcessor: total: 100 ms, init: 6 ms, 1 round(s): 93 ms
```

你可以使用 plugin 选项 `-Kapt-dump-processor-timings` (`org.jetbrains.kotlin.kapt3:dumpProcessorTimings`) (https://github.com/JetBrains/kotlin/pull/4280), 将这个报告输出到一个文件. 以下命令将会运行 kapt, 并将统计报告输出到 `ap-perf-report.file` 文件:

```
kotlinc -cp $MY_CLASSPATH \
-Xplugin=kotlin-annotation-processing-SNAPSHOT.jar -P \
plugin:org.jetbrains.kotlin.kapt3:aptMode=stubsAndApt,\
plugin:org.jetbrains.kotlin.kapt3:apclasspath=processor/build/libs/processor.jar,\
plugin:org.jetbrains.kotlin.kapt3:dumpProcessorTimings=ap-perf-report.file \
-Xplugin=$JAVA_HOME/lib/tools.jar \
-d cli-tests/out \
-no-jdk -no-reflect -no-stdlib -verbose \
sample/src/main/
```

测量注解处理器生成的文件数量

`kotlin-kapt` Gradle plugin 可以对每个注解处理器统计生成的文件数量.

这个功能可以用于追踪构建过程中是否存在未使用的注解处理器. 你可以使用生成的报告来寻找哪些模块触发了不必要的注解处理器, 然后更新这些模块, 不再触发这些注解处理器.

使用以下步骤启用这个统计功能:

- 在你的 `build.gradle(.kts)` 文件中, 将 `showProcessorStats` flag 设置为 `true`:

```
kapt {
    showProcessorStats = true
}
```

- 在你的 `gradle.properties` 文件中, 将 `kapt.verbose` Gradle 属性设置为 `true`:

```
kapt.verbose=true
```

> 也可以使用 命令行选项 `verbose` 启用 verbose 输出.

统计结果将出现在日志中, 级别为 `info`. 你将会看到 `Annotation processor stats:` 行, 之后是每个注解处理器的执行时间统计. 再后面, 将是 `Generated files report:` 行, 之后是每个注解处理器生成的文件数量统计. 比如:

```
[INFO] Annotation processor stats:
[INFO] org.mapstruct.ap.MappingProcessor: total: 290 ms, init: 1 ms,
3 round(s): 289 ms, 0 ms, 0 ms
[INFO] Generated files report:
[INFO] org.mapstruct.ap.MappingProcessor: total sources: 2, sources
per round: 2, 0, 0
```

对 KAPT 使用编译回避功能

为了改进使用 kapt 时的增量构建次数, 可以使用 Gradle 的 编译回避(compile avoidance) 功能 (https://docs.gradle.org/current/userguide/java_plugin.html#sec:java_compile_avoidance). 启用编译回避时, Gradle 可以在重新构建项目时跳过注解处理任务. 具体来说, 在以下情况下会跳过注解处理任务:

- 项目的源代码文件没有变化.
- 依赖项目中的变更满足 ABI (https://en.wikipedia.org/wiki/Application_binary_interface) 兼容. 比如说, 变更只发生在方法体之内, 而方法接口没有变更.

但是, 编译回避不能用于编译类路径中发现的注解处理器, 因为它们的 *任何变更* 都需要运行注解处理任务.

要使用编译回避模式运行 kapt, 你需要:

- 对 `kapt*` 配置手工添加注解处理器依赖项目, 具体方法参见 上文.
- 不要在编译类路径中查找注解处理器, 方法是在你的 `gradle.properties` 文件中添加以下代码:

```
kapt.include.compile.classpath=false
```

增量式(Incremental)注解处理

kapt 支持增量式(Incremental)注解处理, 这个功能默认启用. 目前, 只有当所有注解处理器都以增量模式使用时, 注解处理才可以增量式运行.

要关闭增量式注解处理, 请在你的 `gradle.properties` 文件添加以下代码:

```
kapt.incremental.apt=false
```

注意, 增量式注解处理同时还需要启用 增量式编译(Incremental Compilation) ("增量编译(Incremental compilation)" in "Kotlin Gradle plugin 中的编译与缓存").

Java 编译器选项

kapt 使用 Java 编译器来运行注解处理器. 下面的例子是, 如何向 javac 传递任意的参数:

```
kapt {
    javacOptions {
        // 增加注解处理器允许的最大错误数.
        // 默认值为 100.
        option("-Xmaxerrs", 500)
    }
}
```

对不存在的类型进行纠正

有些注解处理库(比如 AutoFactory), 依赖于类型声明签名中的明确的数据类型. 默认情况下, kapt 会将所有的未知类型替换为 NonExistentClass, 包括编译产生的类的类型信息, 但是你可以修改这种行为. 在 build.gradle(.kts) 文件中添加一个选项, 就可以对桩代码中推断错误的数据类型进行修正:

```
kapt {
    correctErrorTypes = true
}
```

在 Maven 中使用

在 compile 之前, 执行 kotlin-maven-plugin 中的 kapt 目标:

```
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal>
        <!-- 如果你对 kapt plugin 启用了扩展(extension), 那么可以省略
<goals> 元素 -->
    </goals>
    <configuration>
        <sourceDirs>
            <sourceDir>src/main/kotlin</sourceDir>
            <sourceDir>src/main/java</sourceDir>
        </sourceDirs>
        <annotationProcessorPaths>
            <!-- 请在此处指定你的注解处理器 -->
            <annotationProcessorPath>
                <groupId>com.google.dagger</groupId>
                <artifactId>dagger-compiler</artifactId>
                <version>2.9</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</execution>
```

要配置注解处理的级别(level), 请在 `<configuration>` 代码段中将 `aptMode` 设置为下面的值之一:

- `stubs` – 只生成注解处理所需要的桩代码.
- `apt` – 只允许注解处理.
- `stubsAndApt` – (默认值) 生成桩代码, 并运行注解处理.

例如:

```
<configuration>
    ...
    <aptMode>stubs</aptMode>
</configuration>
```

在 IntelliJ 构建系统中使用

IntelliJ IDEA 自有的构建系统不支持 kapt. 如果你想要重新运行注解处理过程, 请通过 "Maven Projects" 工具栏启动编译过程.

在命令行中使用

kapt 编译器插件随 Kotlin 编译器的二进制发布版一同发布.

编译时, 你可以添加这个插件, 方法是使用 kotlinc 的 `Xplugin` 编译选项, 指定它的 JAR 文件路径:

```
-Xplugin=$KOTLIN_HOME/lib/kotlin-annotation-processing.jar
```

以下是这个插件的命令行选项列表:

- `sources` (*必须*): 指定生成的源代码文件的输出路径.
- `classes` (*必须*): 指定生成的 class 文件和资源文件的输出路径.
- `stubs` (*必须*): 指定生成的桩(stub)源代码文件的输出路径. 也可以理解为, 某种临时目录.
- `incrementalData`: 指定生成的桩二进制文件的输出路径.
- `apclasspath` (*可多次指定*): 指定注解处理器的 JAR 文件路径. 你需要多少个 JAR 文件, 就要指定多少个 `apclasspath` 选项.

- `apoptions`: 传递给注解处理器的选项列表, 使用 base64 编码. 详情请参见 AP/javac 选项编码.
- `javacArguments`: 传递给 javac 编译器的选项列表, 使用 base64 编码. 详情请参见 AP/javac 选项编码.
- `processors`: 注解处理器的全限定类名列表, 多个类名之间以逗号分隔. 如果指定了这个选项, kapt 不会在 `apclasspath` 中查找注解处理器.
- `verbose`: 启用详细输出.
- `aptMode` (*必须*)
  - `stubs` – 只生成注解处理所需要的桩代码.
  - `apt` – 只进行注解处理.
  - `stubsAndApt` – 生成桩代码, 并且进行注解处理.
- `correctErrorTypes`: 详情请参见 下文. 默认关闭.
- `dumpFileReadHistory`: 输出路径, 用于输出每个文件的注解处理过程中使用的类的列表.

plugin 的命令行选项格式是: `-P plugin:<plugin id>:<key>=<value>`. 命令行选项可以重复.

示例:

```
-P plugin:org.jetbrains.kotlin.kapt3:sources=build/kapt/sources
-P plugin:org.jetbrains.kotlin.kapt3:classes=build/kapt/classes
-P plugin:org.jetbrains.kotlin.kapt3:stubs=build/kapt/stubs

-P plugin:org.jetbrains.kotlin.kapt3:apclasspath=lib/ap.jar
-P plugin:org.jetbrains.kotlin.kapt3:apclasspath=lib/anotherAp.jar

-P plugin:org.jetbrains.kotlin.kapt3:correctErrorTypes=true
```

生成 Kotlin 源代码

kapt 可以生成 Kotlin 源代码. 它会将生成的 Kotlin 源代码文件写入到 `processingEnv.options["kapt.kotlin.generated"]` 指定的目录, 这些文件会和主源代码文件一起编译.

注意, 对于生成的 Kotlin 文件, kapt 不支持多轮处理.

AP/Javac 选项编码

`apoptions` 和 `javacArguments` 命令行选项可以接受一个编码的参数 map. 对参数 map 编码的方法如下:

```
fun encodeList(options: Map<String, String>): String {
    val os = ByteArrayOutputStream()
    val oos = ObjectOutputStream(os)

    oos.writeInt(options.size)
    for ((key, value) in options.entries) {
        oos.writeUTF(key)
        oos.writeUTF(value)
    }

    oos.flush()
    return Base64.getEncoder().encodeToString(os.toByteArray())
}
```

保留 Java 编译器的注解处理器

kapt 默认会运行所有的注解处理器, 并关闭 javac 编译器的注解处理. 但是, 你有可能会需要 javac 的某些注解处理器继续运行 (比如, Lombok (https://projectlombok.org/)).

在 Gradle 构建脚本文件中, 可以使用 `keepJavacAnnotationProcessors` 选项:

```
kapt {
    keepJavacAnnotationProcessors = true
}
```

如果使用 Maven, 需要指定具体的 plugin 设置. 详情请参见 Lombok 编译器插件的设置示例 ("和 kapt 一起使用" in "Lombok 编译器插件").

Lombok 编译器插件

最终更新: 2024/09/10

> Lombok 编译器插件是 实验性功能 (Kotlin 各部分组件的稳定性). 它随时有可能变更或被删除. 请注意, 只为评估和试验目的来使用这个功能. 希望你能通过我们的 问题追踪系统 (https://youtrack.jetbrains.com/issue/KT-7112) 提供你的反馈意见.

使用 Kotlin Lombok 编译器插件, 可以在 Java/Kotlin 混合代码的同一模块内, 在 Kotlin 代码中生成并使用 Java 代码中的 Lombok 声明. 如果你从另一个模块调用这样的声明, 那么你不需要使用这个插件来编译这个模块.

Lombok 编译器插件不能代替 Lombok (https://projectlombok.org/), 但它能够在 Java/Kotlin 混合代码的模块中帮助 Lombok 正确工作. 因此, 使用这个插件时, 你还是需要象通常那样配置 Lombok. 详情请参见 如何配置 Lombok 编译器插件.

支持的注解

插件支持以下注解:

- `@Getter`, `@Setter`
- `@Builder`
- `@NoArgsConstructor`, `@RequiredArgsConstructor`, 和 `@AllArgsConstructor`
- `@Data`
- `@With`
- `@Value`

我们还在继续改进这个插件. 关于当前的开发状态, 请参见 Lombok 编译器插件的 README 文件 (https://github.com/JetBrains/kotlin/tree/master/plugins/lombok).

目前, 我们没有支持 `@SuperBuilder` 和 `@Tolerate` 注解的计划. 但如果你在 YouTrack 投票支持 @SuperBuilder (https://youtrack.jetbrains.com/issue/KT-53563/Kotlin-Lombok-Support-SuperBuilder) 和 @Tolerate (https://youtrack.jetbrains.com/issue/KT-53564/Kotlin-Lombok-Support-Tolerate), 我们可以考虑增加这个功能.

> 如果在 Kotlin 代码中使用 Lombok 注解, Kotlin 编译器会忽略这些注解.

Gradle

在 `build.gradle(.kts)` 文件中添加 `kotlin-plugin-lombok` Gradle 插件:

Kotlin

```
plugins {
    kotlin("plugin.lombok") version "1.9.23"
    id("io.freefair.lombok") version "8.1.0"
}
```

Groovy

```
plugins {
    id 'org.jetbrains.kotlin.plugin.lombok' version '1.9.23'
    id 'io.freefair.lombok' version '8.1.0'
}
```

详情请参见 关于 Lombok 编译器插件使用方法的测试用示例项目 (https://github.com/kotlin-hands-on/kotlin-lombok-examples/tree/master/kotlin_lombok_gradle/nokapt).

使用 Lombok 配置文件

如果要使用 Lombok 配置文件 (https://projectlombok.org/features/configuration) `lombok.config`, 你需要设置文件路径, 让插件能够找到它. 路径必须是从模块目录开始的相对路径. 例如, 向你的 `build.gradle(.kts)` 文件添加以下代码:

Kotlin

```
kotlinLombok {
    lombokConfigurationFile(file("lombok.config"))
}
```

Groovy

```groovy
kotlinLombok {
    lombokConfigurationFile file("lombok.config")
}
```

详情请参见 关于 Lombok 编译器插件和 `lombok.config` 使用方法的测试用示例项目 (https://github.com/kotlin-hands-on/kotlin-lombok-examples/tree/master/kotlin_lombok_gradle/withconfig).

Maven

要使用 Lombok 编译器插件, 请在 `compilerPlugins` 部分添加插件 `lombok`, 并在 `dependencies` 部分添加依赖项 `kotlin-maven-lombok`. 如果需要使用 Lombok 配置文件 (https://projectlombok.org/features/configuration) `lombok.config`, 请在 `pluginOptions` 部分将它的路径提供给插件. 在 `pom.xml` 文件添加以下内容:

```xml
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <configuration>
        <compilerPlugins>
            <plugin>lombok</plugin>
        </compilerPlugins>
        <pluginOptions>

<option>lombok:config=${project.basedir}/lombok.config</option>
        </pluginOptions>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-lombok</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
```

```xml
                <artifactId>lombok</artifactId>
                <version>1.18.20</version>
                <scope>provided</scope>
            </dependency>
        </dependencies>
    </plugin>
```

详情请参见 关于 Lombok 编译器插件和 lombok.config 使用方法的测试用示例项目 (https://github.com/kotlin-hands-on/kotlin-lombok-examples/tree/master/kotlin_lombok_maven/nokapt).

和 kapt 一起使用

默认情况下, kapt ([kapt 编译器插件](#)) 编译器插件 运行所有的注解处理器, 并禁止 javac 的注解处理. 要和 kapt 一起运行 Lombok (https://projectlombok.org/), 请设置 kapt, 允许 javac 的注解处理器继续工作.

如果你使用 Gradle, 请向 `build.gradle(.kts)` 文件添加以下选项:

```
kapt {
    keepJavacAnnotationProcessors = true
}
```

如果使用 Maven, 请使用以下设置, 通过 Java 的编译器启动 Lombok:

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>${lombok.version}</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
```

```
    </configuration>
</plugin>
```

如果注解处理器不依赖于 Lombok 生成的代码, Lombok 编译器插件可以和 kapt (kapt 编译器插件) 一起正确工作.

请参见同时使用 kapt 和 Lombok 编译器插件的测试用示例项目:

- 使用 Gradle (https://github.com/JetBrains/kotlin/tree/master/libraries/tools/kotlin-gradle-plugin-integration-tests/src/test/resources/testProject/lombokProject/yeskapt).
- 使用 Maven (https://github.com/kotlin-hands-on/kotlin-lombok-examples/tree/master/kotlin_lombok_maven/yeskapt)

命令行编译器

在 Kotlin 编译器的二进制文件发布版中可以找到 Lombok 编译器插件的 JAR 文件. 你可以使用 kotlinc 的 `Xplugin` 选项, 指定插件的 JAR 文件路径来加载这个插件:

```
-Xplugin=$KOTLIN_HOME/lib/lombok-compiler-plugin.jar
```

如果你想要使用 `lombok.config` 文件, 请将以下代码中的 `<PATH_TO_CONFIG_FILE>` 替换为你的 `lombok.config` 文件的路径:

```
# 插件选项的格式是: "-P plugin:<plugin id>:<key>=<value>".
# 选项可以重复.

-P plugin:org.jetbrains.kotlin.lombok:config=<PATH_TO_CONFIG_FILE>
```

Kotlin 符号处理(Kotlin Symbol Processing) API

最终更新: 2024/09/10

Kotlin 符号处理(Kotlin Symbol Processing, *KSP*) 是一组 API, 你可以使用它开发轻量的编译器插件. KSP 提供一组简化的编译器插件 API, 利用 Kotlin 的能力, 同时保持最小的学习曲线. 与 kapt (kapt 编译器插件) 相比, 使用 KSP 的注解处理器运行速度可以快 2 倍.

- 关于 KSP 与 kapt 与比较, 详情请参见 为什么需要 KSP (为什么使用 KSP).
- 要开始编写 KSP 处理器, 请参见 KSP 快速入门 (KSP 快速入门).

概述

KSP API 按照语言习惯来处理 Kotlin 程序. KSP 理解 Kotlin 专有的功能特性, 比如扩展函数, 声明处类型变异(declaration-site variance), 以及局部函数. 它还明确的对类型建立模型, 并提供基本类型检查, 比如相等性, 以及赋值兼容性.

API 根据 Kotlin 语法 (https://kotlinlang.org/docs/reference/grammar.html), 在符号层面对 Kotlin 程序结构建立模型. 当基于 KSP 的插件处理源程序时, 处理器可以访问各种结构, 比如类, 类成员, 函数, 以及关联的参数, 而 `if` 代码段和 `for` 循环之类则不可以访问.

概念上来讲, KSP 类似于 Kotlin 反射中的 KType (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect/-k-type/). API 允许处理器从类的声明查找到相应的带特定类型参数的类型, 或者反过来. 你也可以替换类型参数, 特定的变体, 应用星号投射(Star Projection), 以及标注类型可否为空.

看待 KSP 的另一种方式是当作 Kotlin 程序的一个预处理器框架. 将基于 KSP 的插件看作 *符号处理器*, 或者简称 *处理器*, 编译过程中的数据流可以描述为以下步骤:

1. 处理器读取并分析源程序和资源.
2. 处理器生成代码或其他形式的输出.
3. Kotlin 编译器将源程序和生成的代码一起编译.

与功能完全的编译器插件不同, 处理器不能修改代码. 修改程序语义的编译器插件有时可能会非常令人困惑. KSP 将源程序当作只读, 避免这种情况.

你也可以观看这个视频, 大致了解 KSP:

KSP 如何看待源代码文件

大多数处理器会浏览输入的源代码的各种程序结构. 在介绍 API 的使用方法之前, 我们来看一下从 KSP 的观点如何看待文件:

```
KSFile
  packageName: KSName
  fileName: String
  annotations: List<KSAnnotation>  // 源代码文件注解
  declarations: List<KSDeclaration>
    KSClassDeclaration // 类, 接口, 对象
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      classKind: ClassKind
      primaryConstructor: KSFunctionDeclaration
      superTypes: List<KSTypeReference>
      // 包含内部类, 成员函数, 属性, 等等.
      declarations: List<KSDeclaration>
    KSFunctionDeclaration // 顶层函数
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      functionKind: FunctionKind
      extensionReceiver: KSTypeReference?
      returnType: KSTypeReference
      parameters: List<KSValueParameter>
      // 包含局部类, 局部函数, 局部变量, 等等.
      declarations: List<KSDeclaration>
    KSPropertyDeclaration // 全局变量
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
```

```
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      extensionReceiver: KSTypeReference?
      type: KSTypeReference
      getter: KSPropertyGetter
        returnType: KSTypeReference
      setter: KSPropertySetter
        parameter: KSValueParameter
```

这个图列出了在源代码文件中声明的大多数东西: 类, 函数, 属性, 等等.

SymbolProcessorProvider: 入口点

KSP 要求实现 `SymbolProcessorProvider` 接口, 使用它来创建 `SymbolProcessor` 实例:

```
interface SymbolProcessorProvider {
    fun create(environment: SymbolProcessorEnvironment):
SymbolProcessor
}
```

其中 `SymbolProcessor` 定义如下:

```
interface SymbolProcessor {
    fun process(resolver: Resolver): List<KSAnnotated> // 我们集中看这
里
    fun finish() {}
    fun onError() {}
}
```

`SymbolProcessor` 使用 `Resolver` 来访问编译器细节, 比如符号. 如果一个处理器要查找所有的顶层函数和顶层类中的非局部函数, 大概实现如下:

```
class HelloFunctionFinderProcessor : SymbolProcessor() {
    // ...
    val functions = mutableListOf<KSClassDeclaration>()
    val visitor = FindFunctionsVisitor()

    override fun process(resolver: Resolver) {
        resolver.getAllFiles().forEach { it.accept(visitor, Unit) }
```

```
    }

    inner class FindFunctionsVisitor : KSVisitorVoid() {
        override fun visitClassDeclaration(classDeclaration:
KSClassDeclaration, data: Unit) {
            classDeclaration.getDeclaredFunctions().forEach {
it.accept(this, Unit) }
        }

        override fun visitFunctionDeclaration(function:
KSFunctionDeclaration, data: Unit) {
            functions.add(function)
        }

        override fun visitFile(file: KSFile, data: Unit) {
            file.declarations.forEach { it.accept(this, Unit) }
        }
    }
    // ...

    class Provider : SymbolProcessorProvider {
        override fun create(environment:
SymbolProcessorEnvironment): SymbolProcessor = TODO()
    }
}
```

资源

- KSP 快速入门 (KSP 快速入门)
- 为什么使用 KSP? (为什么使用 KSP)
- 示例 (KSP 示例程序)
- KSP 如何将 Kotlin 代码组织为模型 (KSP 如何将 Kotlin 代码组织为模型)
- 针对 Java 注解处理器开发者的参考文档 (针对 Java 注解处理器开发者的参考文档)
- 增量式处理 (增量式处理(Incremental Processing))

- 多轮处理 (多轮(Multiple Round)处理)
- 在跨平台项目中使用 KSP (在 Kotlin Multiplatform 中使用 KSP)
- 在命令行运行 KSP (在命令行运行 KSP)
- FAQ (KSP FAQ)

支持的库

下面是 Android 上的流行的库, 以及它们对 KSP 的支持情况:

库	状态
Room	官方支持 (https://developer.android.com/jetpack/androidx/releases/room#2.3.0-beta02)
Moshi	官方支持 (https://github.com/square/moshi/)
RxHttp	官方支持 (https://github.com/liujingxing/rxhttp)
Kotshi	官方支持 (https://github.com/ansman/kotshi)
Lyricist	官方支持 (https://github.com/adrielcafe/lyricist)
Lich SavedState	官方支持 (https://github.com/line/lich/tree/master/savedstate)
gRPC Dekorator	官方支持 (https://github.com/mottljan/grpc-dekorator)
EasyAdapter	官方支持 (https://github.com/AmrDeveloper/EasyAdapter)
Koin Annotations	官方支持 (https://github.com/InsertKoinIO/koin-annotations)
Glide	官方支持 (https://github.com/bumptech/glide)
Micronaut	官方支持 (https://micronaut.io/2023/07/14/micronaut-framework-4-0-0-released/)
Epoxy	官方支持 (https://github.com/airbnb/epoxy)
Paris	官方支持 (https://github.com/airbnb/paris)
Auto Dagger	官方支持 (https://github.com/ansman/auto-dagger)

SealedX	官方支持 (https://github.com/skydoves/sealedx)
DeeplinkDispatch	通过 airbnb/DeepLinkDispatch#323 支持 (https://github.com/airbnb/DeepLinkDispatch/pull/323)
Dagger	Alpha (https://dagger.dev/dev-guide/ksp)
Motif	Alpha (https://github.com/uber/motif)
Hilt	开发中 (https://dagger.dev/dev-guide/ksp)
Auto Factory	目前不支持 (https://github.com/google/auto/issues/982)

KSP 快速入门

最终更新: 2024/09/10

要快速入门 KSP, 你可以创建自己的处理器, 或者参考 示例代码 (https://github.com/google/ksp/tree/main/examples/playground).

创建一个你自己的处理器

1. 创建一个空的 gradle 项目.
2. 在根项目中指定 Kotlin plugin 版本 `1.9.21`, 供其他项目模块使用:

   Kotlin

```kotlin
plugins {
    kotlin("jvm") version "1.9.21" apply false
}

buildscript {
    dependencies {
        classpath(kotlin("gradle-plugin", version = "1.9.21"))
    }
}
```

   Groovy

```groovy
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.9.21' apply false
}

buildscript {
    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:1.9.21'
```

```
    }
}
```

3. 添加一个模块, 容纳处理器.

4. 在模块的构建脚本中, 使用 Kotlin plugin, 并在 `dependencies` 代码段添加 KSP API.

Kotlin

```kotlin
plugins {
    kotlin("jvm")
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("com.google.devtools.ksp:symbol-processing-api:1.9.21-1.0.15")
}
```

Groovy

```groovy
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.9.23'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.google.devtools.ksp:symbol-processing-
```

```
api:1.9.21-1.0.15'
}
```

5. 你需要实现 `com.google.devtools.ksp.processing.SymbolProcessor` (https://github.com/google/ksp/tree/main/api/src/main/kotlin/com/google/devtools/ksp/processing/SymbolProcessor.kt) 和 `com.google.devtools.ksp.processing.SymbolProcessorProvider` (https://github.com/google/ksp/tree/main/api/src/main/kotlin/com/google/devtools/ksp/processing/SymbolProcessorProvider.kt). 你实现的 `SymbolProcessorProvider` 将被作为一个服务装载, 负责创建你实现的 `SymbolProcessor` 实例. 注意以下几点:

   - 实现 `SymbolProcessorProvider.create()` (https://github.com/google/ksp/blob/master/api/src/main/kotlin/com/google/devtools/ksp/processing/SymbolProcessorProvider.kt), 负责创建一个 `SymbolProcessor`. 通过 `SymbolProcessorProvider.create()` 的参数传递你的处理器需要的依赖项 (比如 `CodeGenerator`, 处理器选项).

   - 你的主逻辑应该在 `SymbolProcessor.process()` (https://github.com/google/ksp/blob/master/api/src/main/kotlin/com/google/devtools/ksp/processing/SymbolProcessor.kt) 方法中.

   - 使用 `resolver.getSymbolsWithAnnotation()`, 给定一个注解的完全限定名称, 得到你希望处理的符号.

   - KSP 的一个常见使用场景是实现一个自定义的访问器 (`com.google.devtools.ksp.symbol.KSVisitor` 接口) 来操作符号. 一个简单的访问器模板是 `com.google.devtools.ksp.symbol.KSDefaultVisitor`.

   - 关于 `SymbolProcessorProvider` 和 `SymbolProcessor` 接口实现的例子, 请参见示例项目中的以下文件.

     - `src/main/kotlin/BuilderProcessor.kt`

     - `src/main/kotlin/TestProcessor.kt`

   - 编写完你自己的处理器之后, 需要向包注册你的处理器 provider, 方法是在 `resources/META-INF/services/com.google.devtools.ksp.processing.SymbolProcessorProvider` 中包含它的完全限定名称.

在一个项目中使用你自己的处理器

1. 创建另一个模块, 包含一段工作程序, 用来试验你的处理器.

   Kotlin

```
pluginManagement {
    repositories {
        gradlePluginPortal()
    }
}
```

   Groovy

```
pluginManagement {
    repositories {
        gradlePluginPortal()
    }
}
```

2. 在模块的构建脚本中, 使用指定版本的 `com.google.devtools.ksp` plugin, 并在依赖项中添加你的处理器.

   Kotlin

```
plugins {
    id("com.google.devtools.ksp") version "1.9.21-1.0.15"
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))
    implementation(project(":test-processor"))
    ksp(project(":test-processor"))
}
```

Groovy

```groovy
plugins {
    id 'com.google.devtools.ksp' version '1.9.21-1.0.15'
}

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib:1.9.23'
    implementation project(':test-processor')
    ksp project(':test-processor')
}
```

3. 运行 `./gradlew build`. 你可以在 `build/generated/source/ksp` 目录下看到生成的代码.

下面是一个构建脚本示例, 它对工作程序使用 KSP plugin:

Kotlin

```kotlin
plugins {
    id("com.google.devtools.ksp") version "1.9.21-1.0.15"
    kotlin("jvm")
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))
    implementation(project(":test-processor"))
    ksp(project(":test-processor"))
}
```

Groovy

```
plugins {
    id 'com.google.devtools.ksp' version '1.9.21-1.0.15'
    id 'org.jetbrains.kotlin.jvm' version '1.9.23'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib:1.9.23'
    implementation project(':test-processor')
    ksp project(':test-processor')
}
```

向处理器传递选项

在 gradle 构建脚本中指定 `SymbolProcessorEnvironment.options` 中的处理器选项:

```
ksp {
    arg("option1", "value1")
    arg("option2", "value2")
    ...
}
```

让 IDE 感知生成的代码

> 从 KSP 1.8.0-1.0.9 开始, 生成的源代码文件会自动进行注册. 如果你在使用 KSP 1.0.9 或更高版本, 但不需要让 IDE 感知生成的资源, 那么可以跳过这一章节.

默认情况下, IntelliJ IDEA 或其他 IDE 不知道生成的代码. 因此 IDE 会将生成的符号标记为无法解析. 要让 IDE 能够理解生成的符号, 请将以下路径标记为生成的源代码根目录:

```
build/generated/ksp/main/kotlin/
```

```
build/generated/ksp/main/java/
```

如果你的 IDE 支持资源目录, 那么还需要标记下面的路径:

```
build/generated/ksp/main/resources/
```

在你的 KSP 使用者模块的构建脚本中, 可能还需要配置这些目录:

Kotlin

```kotlin
kotlin {
    sourceSets.main {
        kotlin.srcDir("build/generated/ksp/main/kotlin")
    }
    sourceSets.test {
        kotlin.srcDir("build/generated/ksp/test/kotlin")
    }
}
```

Groovy

```groovy
kotlin {
    sourceSets {
        main.kotlin.srcDirs += 'build/generated/ksp/main/kotlin'
        test.kotlin.srcDirs += 'build/generated/ksp/test/kotlin'
    }
}
```

如果你使用 IntelliJ IDEA, 并在 Gradle plugin 中使用 KSP, 那么上面的代码段会出现以下警告:

```
Execution optimizations have been disabled for task
':publishPluginJar' to ensure correctness due to the following
reasons:
Gradle detected a problem with the following location:
'../build/generated/ksp/main/kotlin'.
```

```
Reason: Task ':publishPluginJar' uses this output of task
':kspKotlin' without declaring an explicit or implicit dependency.
```

这种情况下，请改为使用下面的构建脚本:

Kotlin

```
plugins {
    // ...
    idea
}

idea {
    module {
        // 由于 https://github.com/gradle/gradle/issues/8749, 不要
使用 +=
        sourceDirs = sourceDirs +
file("build/generated/ksp/main/kotlin") // 或者
tasks["kspKotlin"].destination
        testSourceDirs = testSourceDirs +
file("build/generated/ksp/test/kotlin")
        generatedSourceDirs = generatedSourceDirs +
file("build/generated/ksp/main/kotlin") +
file("build/generated/ksp/test/kotlin")
    }
}
```

Groovy

```
plugins {
    // ...
    id 'idea'
}

idea {
    module {
        // 由于 https://github.com/gradle/gradle/issues/8749, 不要
```

```
使用 +=
        sourceDirs = sourceDirs +
file('build/generated/ksp/main/kotlin') // 或者
tasks["kspKotlin"].destination
        testSourceDirs = testSourceDirs +
file('build/generated/ksp/test/kotlin')
        generatedSourceDirs = generatedSourceDirs +
file('build/generated/ksp/main/kotlin') +
file('build/generated/ksp/test/kotlin')
    }
}
```

为什么使用 KSP

最终更新: 2024/09/10

编译器插件是强大的元编程(Metaprogramming)工具, 能够大大增强你编写代码的方式. 编译器插件直接将编译器作为库来调用, 分析并修改输入的程序. 这些插件还能够为各种用途生成输出. 比如, 它们能够生成样板代码(Boilerplate Code), 它们甚至还能对使用了特殊标记的程序元素生成完整的实现, 比如 `Parcelable`. 插件还有很多很多其他用途, 甚至可以用于实现并精密调节(fine-tune)语言没有直接提供的那些功能特性.

尽管编译器插件很强大, 这种能力也带来了代价. 要编写即使是最简单的插件, 你也需要编译器相关的背景知识, 还需要对你的编译器的实现细节有一定程度的了解. 另一个实际问题是, 插件经常与特定的编译器版本紧密结合, 因此每次你想要支持编译器的新版本时, 你可能都需要更新你的插件.

KSP 使得创建轻量的编译器插件更加容易

KSP 的设计意图是隐藏编译器的变更, 对使用它的处理器尽量减少维护工作量. KSP 的设计目的是不要与 JVM 紧密结合, 因此将来它能够更加容易的应用到其他平台. KSP 的设计目的还包括减少构建时间. 对于某些处理器, 比如 Glide (https://github.com/bumptech/glide), 与 kapt 相比, KSP 减少了完全编译时间最高达到 25%.

KSP 本身作为一个编译器插件实现. Google 的 Maven 仓库中有一些预构建的包, 你可以下载使用, 不需要自己构建项目.

与 kotlinc 编译器插件比较

`kotlinc` 编译器插件能访问编译器中的几乎所有功能, 因此有极强的功能和灵活性. 但是, 由于这些插件可能潜在的依赖于编译器中的任何功能, 因此它们对编译器的变更很敏感, 需要频繁的维护. 这些插件 还需要 深入的理解 `kotlinc` 的实现, 因此学习曲线会很陡.

KSP 的目标是通过一组完善定义的 API 来隐藏编译器的大多数变更, 尽管编译器甚至 Kotlin 语言的大的变更仍然需要公开给 API 使用者.

KSP 希望提供 API, 牺牲性能来提高简易性, 以此实现通常的使用场景. 它的能力只是通常的 `kotlinc` 插件的一小部分. 比如, `kotlinc` 能够计算表达式和语句, 甚至还能够修改代码, 而 KSP 则不可以.

尽管编写 `kotlinc` 插件可能很有趣, 但也会耗费很多时间. 如果你不打算学习 `kotlinc` 的实现, 也不需要修改源代码, 或者读取表达式, KSP 可能更适合你的需要.

与反射(Reflection)比较

KSP 的 API 与 `kotlin.reflect` 类似. 主要差别是, KSP 中的类型引用需要明确的解析. 这是二者不使用共同的接口的原因之一.

与 kapt 比较

kapt (kapt 编译器插件) 是一个出色的解决方案, 它使得大量的 Java 注解处理器能够直接用于 Kotlin 程序. KSP 相比 kapt 的主要优势是, 提高了构建性能, 没有与 JVM 紧密结合, 更符合 Kotlin 惯用法的 API, 以及能够理解 Kotlin 专有的符号.

为了不加修改的直接运行 Java 注解处理器, kapt 将 Kotlin 代码编译为 Java 桩代码(stub), 其中保留了 Java 注解处理器关注的信息. 为了创建这些桩代码, kapt 需要解析 Kotlin 程序中的所有符号. 桩代码生成占据了 `kotlinc` 完整分析过程的大约 1/3, `kotlinc` 的代码生成过程也是如此. 对于很多注解处理器, 这个过程比处理器本身耗费的时间要长很多. 比如, Glide 只会分析使用了预定义注解的, 非常少量的类, 它的代码生成非常快速. 几乎所有的构建开销都发生在桩代码生成阶段. 切换到 KSP 可以立即减少编译器消耗时间的 25%.

为了性能评估, 我们用 KSP 实现了一个 简化版本 (https://github.com/google/ksp/releases/download/1.4.10-dev-experimental-20200924/miniGlide.zip) 的 Glide (https://github.com/bumptech/glide), 让它为 Tachiyomi (https://github.com/inorichi/tachiyomi) 项目生成代码. 在我们的测试设备上, 尽管项目的 Kotlin 全体编译时间是 21.55 秒, 但 kapt 生成代码耗费了 8.67 秒, 而我们的 KSP 实现生成代码只耗费 1.15 秒.

与 kapt 不同, KSP 中的处理器不会以 Java 的方式看待输入程序. API 对 Kotlin 来说更加自然, 尤其是对于 Kotlin 专有的功能, 比如顶层函数. 由于 KSP 不会象 kapt 那样将处理代理给 `javac`, 因此它不会依赖于 JVM 专有的行为, 并且将来有可能用于其它平台.

限制

尽管 KSP 想要成为一个适用于大多数使用场景的简单解决方案, 但它与其它插件解决方案相比, 还是进行了一些折中. KSP 的目标不包括以下功能:

- 计算源代码中表达式层的信息.
- 修改源代码.
- 100% 兼容 Java 注解处理 API.

我们还在探索一些额外的功能特性. 这些功能特性目前还不可用:

- IDE 集成: 目前 IDE 不能感知生成的代码.

KSP 示例程序

最终更新: 2024/09/10

得到所有成员函数

```
fun KSClassDeclaration.getDeclaredFunctions():
Sequence<KSFunctionDeclaration> =
    declarations.filterIsInstance<KSFunctionDeclaration>()
```

检查一个类或函数是否为 local

```
fun KSDeclaration.isLocal(): Boolean =
    parentDeclaration != null && parentDeclaration !is
KSClassDeclaration
```

查找类型别名指向的实际的类或接口声明

```
fun KSTypeAlias.findActualType(): KSClassDeclaration {
    val resolvedType = this.type.resolve().declaration
    return if (resolvedType is KSTypeAlias) {
        resolvedType.findActualType()
    } else {
        resolvedType as KSClassDeclaration
    }
}
```

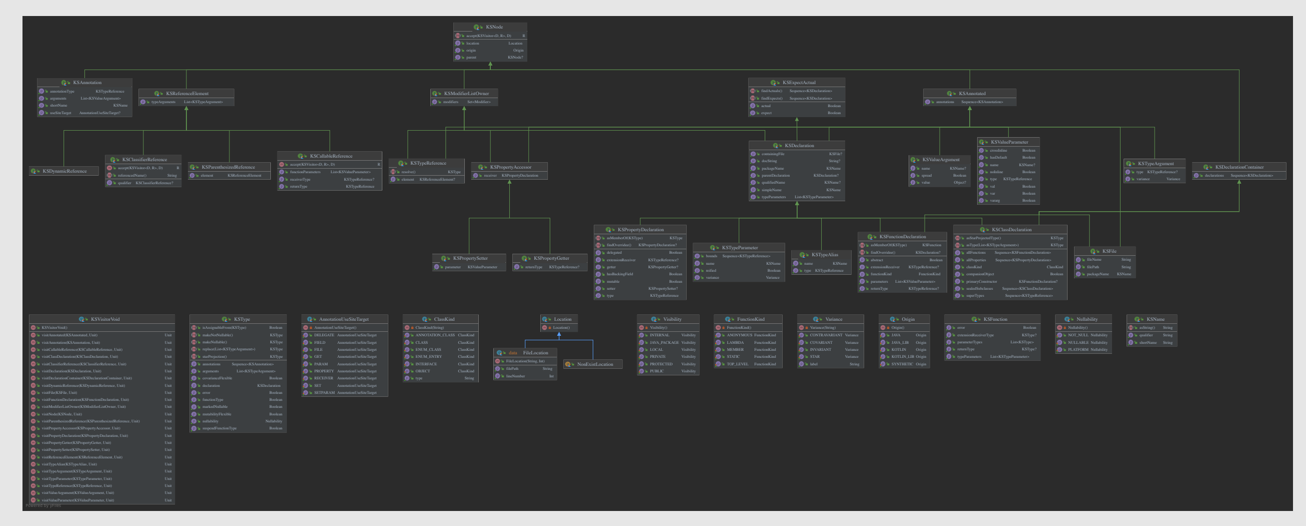
在源代码文件的注解中查找被压制(Suppressed)的名称

```
// @file:kotlin.Suppress("Example1", "Example2")
fun KSFile.suppressedNames(): Sequence<String> = annotations
    .filter {
        it.shortName.asString() == "Suppress" &&
```

```
it.annotationType.resolve().declaration.qualifiedName?.asString() ==
"kotlin.Suppress"
    }.flatMap {
        it.arguments.flatMap {
            (it.value as Array<String>).toList()
        }
    }
```

KSP 如何将 Kotlin 代码组织为模型

最终更新: 2024/09/10

> **可以在 KSP GitHub 代码仓库 找到 API 定义 . 下面的类图大致说明了 KSP 中 Kotlin 的模型结构**
>
> 

> 查看完整大小的图片 (https://kotlinlang.org/docs/images/ksp-class-diagram.svg).

类型解析

在 API 的底层实现中, 主要的资源消耗是类型解析. 因此类型引用设计为由处理器明确解析(也有少数例外情况). 当引用一个 *类型(Type)* (比如 `KSFunctionDeclaration.returnType` 或 `KSAnnotation.annotationType`), 永远是 `KSTypeReference`, 这是一个带有注解和修饰符的 `KSReferenceElement`.

```
interface KSFunctionDeclaration : ... {
    val returnType: KSTypeReference?
    // ...
}

interface KSTypeReference : KSAnnotated, KSModifierListOwner {
```

```
    val type: KSReferenceElement
}
```

一个 KSTypeReference 可以解析为一个 KSType, 它引用到 Kotlin 类型系统中的一个类型.

一个KSTypeReference 拥有一个 KSReferenceElement, 它是 Kotlin 程序结构的数据模型: 也就是, 类型引用是如何编写的. 它对应于 Kotlin 语法中的 type (https://kotlinlang.org/docs/reference/grammar.html#type) 元素.

一个 KSReferenceElement 可以是一个 KSClassifierReference 或 KSCallableReference, 其中包含很多不需要解析的有用信息. 比如, KSClassifierReference 拥有 referencedName, KSCallableReference 拥有 receiverType, functionArguments, 和 returnType.

如果需要一个 KSTypeReference 引用的原始声明, 通常可以解析到 KSType, 并通过访问 KSType.declaration 得到. 要从一个类型被使用的地方, 得到它的类声明的地方, 代码如下:

```
val ksType: KSType = ksTypeReference.resolve()
val ksDeclaration: KSDeclaration = ksType.declaration
```

类型解析的代价很高, 因此需要明确调用. 通过解析得到的有些信息在 KSReferenceElement 中已经存在了. 比如, 通过 KSClassifierReference.referencedName 可以过滤掉很多不感兴趣的元素. 你应该只有在需要从 KSDeclaration 或 KSType 得到具体信息的时候才进行类型解析.

指向一个函数类型的 KSTypeReference 在它的元素中已经有了大部分信息. 尽管可以解析到 Function0, Function1, 等等的函数群, 但这些解析不会带来比 KSCallableReference 更多的任何信息. 有一种情况需要解析函数类型引用, 就是处理函数原型(Function Prototype)的 identity.

针对 Java 注解处理器开发者的参考文档

最终更新: 2024/09/10

程序元素

Java	KSP 中的类似功能	注意事项
`AnnotationMirror`	`KSAnnotation`	
`AnnotationValue`	`KSValueArguments`	
`Element`	`KSDeclaration`/`KSDeclarationContainer`	
`ExecutableElement`	`KSFunctionDeclaration`	
`PackageElement`	`KSFile`	KSP 不将包建模为程序元素
`Parameterizable`	`KSDeclaration`	
`QualifiedNameable`	`KSDeclaration`	
`TypeElement`	`KSClassDeclaration`	
`TypeParameterElement`	`KSTypeParameter`	
`VariableElement`	`KSValueParameter`/`KSPropertyDeclaration`	

类型

KSP 要求明确解析类型, 因此在解析之前, Java 中的有些功能只能通过 `KSType` 和对应的元素得到.

Java	KSP 中的类似功能	注意事项
ArrayType	KSBuiltIns.arrayType	
DeclaredType	KSType / KSClassifierReference	
ErrorType	KSType.isError	
ExecutableType	KSType / KSCallableReference	
IntersectionType	KSType / KSTypeParameter	
NoType	KSType.isError	KSP 中没有这样的功能
NullType		KSP 中没有这样的功能
PrimitiveType	KSBuiltIns	与 Java 中的基本类型不完全相同
ReferenceType	KSTypeReference	
TypeMirror	KSType	
TypeVariable	KSTypeParameter	
UnionType	没有这样的功能	Kotlin 的 每个 catch 代码段只有 1 个类型. 即使对 Java 注解处理器来说, UnionType 也是不可访问的

`WildcardType`	`KSType`/`KSTypeArgument`	

杂项

Java	KSP 中的类似功能	注意事项
Name	KSName	
ElementKind	ClassKind/FunctionKind	
Modifier	Modifier	
NestingKind	ClassKind/FunctionKind	
AnnotationValueVisitor		
ElementVisitor	KSVisitor	
AnnotatedConstruct	KSAnnotated	
TypeVisitor		
TypeKind	KSBuiltIns	有些可以在 builtin 中得到, 其他通过 KSClassDeclaration 得到 DeclaredType
ElementFilter	Collection.filterIsInstance	
ElementKindVisitor	KSVisitor	
ElementScanner	KSTopDownVisitor	

`SimpleAnnotationValueVisitor`		KSP 中不需要
`SimpleElementVisitor`	`KSVisitor`	
`SimpleTypeVisitor`		
`TypeKindVisitor`		
`Types`	`Resolver`/`utils`	有些 `utils` 也被集成在符号接口中
`Elements`	`Resolver`/`utils`	

细节

这部分介绍 KSP 怎样提供 Java 注解处理 API 的功能.

AnnotationMirror

Java	KSP 中的同等功能
`getAnnotationType`	`ksAnnotation.annotationType`
`getElementValues`	`ksAnnotation.arguments`

AnnotationValue

Java	KSP 中的同等功能
`getValue`	`ksValueArgument.value`

Element

Java	KSP 中的同等功能
asType	ksClassDeclaration.asType(...) 只对 KSClassDeclaration 有效. 需要提供类型参数.
getAnnotation	未实现
getAnnotationMirrors	ksDeclaration.annotations
getEnclosedElements	ksDeclarationContainer.declarations
getEnclosingElements	ksDeclaration.parentDeclaration
getKind	通过 ClassKind 或 FunctionKind 进行类型检查和转换
getModifiers	ksDeclaration.modifiers
getSimpleName	ksDeclaration.simpleName

ExecutableElement

Java	KSP 中的同等功能
`getDefaultValue`	未实现
`getParameters`	`ksFunctionDeclaration.parameters`
`getReceiverType`	`ksFunctionDeclaration.parentDeclaration`
`getReturnType`	`ksFunctionDeclaration.returnType`
`getSimpleName`	`ksFunctionDeclaration.simpleName`
`getThrownTypes`	Kotlin 中不需要
`getTypeParameters`	`ksFunctionDeclaration.typeParameters`
`isDefault`	检查父类型是不是接口
`isVarArgs`	`ksFunctionDeclaration.parameters.any { it.isVarArg }`

Parameterizable

Java	KSP 中的同等功能
`getTypeParameters`	`ksFunctionDeclaration.typeParameters`

QualifiedNameable

Java	KSP 中的同等功能
`getQualifiedName`	`ksDeclaration.qualifiedName`

TypeElement

Java	KSP 中的同等功能
getEnclosedElements	ksClassDeclaration.declarations
getEnclosingElement	ksClassDeclaration.parentDeclaration

getInterfaces	// 不需要类型解析也应该能够实现
	ksClassDeclaration.superTypes
	    .map { it.resolve() }
	    .filter { (it?.declaration as?
	KSClassDeclaration)?.classKind == ClassKind.INTERFACE
	}

getNestingKind	Check KSClassDeclaration.parentDeclaration 和 inner 修饰符
getQualifiedName	ksClassDeclaration.qualifiedName
getSimpleName	ksClassDeclaration.simpleName

getSuperclass	// 不需要类型解析也应该能够实现
	ksClassDeclaration.superTypes
	    .map { it.resolve() }
	    .filter { (it?.declaration as?
	KSClassDeclaration)?.classKind == ClassKind.CLASS }

getTypeParameters	ksClassDeclaration.typeParameters

TypeParameterElement

Java	KSP 中的同等功能
`getBounds`	`ksTypeParameter.bounds`
`getEnclosingElement`	`ksTypeParameter.parentDeclaration`
`getGenericElement`	`ksTypeParameter.parentDeclaration`

VariableElement

Java	KSP 中的同等功能
`getConstantValue`	未实现
`getEnclosingElement`	`ksValueParameter.parentDeclaration`
`getSimpleName`	`ksValueParameter.simpleName`

ArrayType

Java	KSP 中的同等功能
`getComponentType`	`ksType.arguments.first()`

DeclaredType

Java	KSP 中的同等功能
`asElement`	`ksType.declaration`
`getEnclosingType`	`ksType.declaration.parentDeclaration`
`getTypeArguments`	`ksType.arguments`

ExecutableType

> 函数的 KSType 只是一个签名, 由 FunctionN<R, T1, T2, ..., TN> 群表达.

Java	KSP 中的同等功能
getParameterTypes	ksType.declaration.typeParameters, ksFunctionDeclaration.parameters.map { it.type }
getReceiverType	ksFunctionDeclaration.parentDeclaration.asType(...)
getReturnType	ksType.declaration.typeParameters.last()
getThrownTypes	Kotlin 中不需要
getTypeVariables	ksFunctionDeclaration.typeParameters

IntersectionType

Java	KSP 中的同等功能
getBounds	ksTypeParameter.bounds

TypeMirror

Java	KSP 中的同等功能
getKind	对于基本类型, Unit, 与 KSBuiltIns 中的类型比较, 其他情况使用 DeclaredType

TypeVariable

Java	KSP 中的同等功能
`asElement`	`ksType.declaration`
`getLowerBound`	未决定. 只存在 capture, 并且需要明确的边界检查时, 才需要这个功能.
`getUpperBound`	`ksTypeParameter.bounds`

WildcardType

Java	KSP 中的同等功能
`getExtendsBound`	`if (ksTypeArgument.variance == Variance.COVARIANT) ksTypeArgument.type else null`
`getSuperBound`	`if (ksTypeArgument.variance == Variance.CONTRAVARIANT) ksTypeArgument.type else null`

Elements

Java	KSP 中的同等功能
`getAllAnnotationMirrors`	`KSDeclarations.annotations`
`getAllMembers`	`getAllFunctions`, `getAllProperties` 未实现
`getBinaryName`	未决定, 参见 Java Specification (https://docs.oracle.com/javase/specs/jls/se13/html/jls-13.html#jls-13.1)
`getConstantExpression`	常数值, 而不是表达式
`getDocComment`	未实现
`getElementValuesWithDefaults`	未实现
`getName`	`resolver.getKSNameFromString`
`getPackageElement`	不支持包, 但可以取得包信息. KSP 中不能对包进行操作.
`getPackageOf`	不支持包
`getTypeElement`	`Resolver.getClassDeclarationByName`
`hides`	未实现
`isDeprecated`	`KsDeclaration.annotations.any {` `it.annotationType.resolve()!!.declaration.qualified`

	``
	Name!!.asString() ==
	Deprecated::class.qualifiedName
	}
	``

`overrides`	`KSFunctionDeclaration.overrides`/`KSPropertyDeclaration.overrides` (各个类的成员函数)
`printElements`	KSP 对大多数类有基本的 `toString()` 实现

Types

Java	KSP 中的同等功能
asElement	ksType.declaration
asMemberOf	resolver.asMemberOf
boxedClass	不需要
capture	未决定
contains	KSType.isAssignableFrom
directSuperTypes	(ksType.declaration as KSClassDeclaration).superTypes
erasure	ksType.starProjection()
getArrayType	ksBuiltIns.arrayType.replace(...)
getDeclaredType	ksClassDeclaration.asType
getNoType	ksBuiltIns.nothingType/null
getNullType	根据上下文确定，可能可以使用 KSType.markNullable
getPrimitiveType	不需要，检查 KSBuiltins
getWildcardType	在需要 KSTypeArgument 的地方使用 Variance
isAssignable	ksType.isAssignableFrom

`isSameType`	`ksType.equals`
`isSubsignature`	`functionTypeA == functionTypeB`/`functionTypeA == functionTypeB.starProjection()`
`isSubtype`	`ksType.isAssignableFrom`
`unboxedType`	不需要

增量式处理(Incremental Processing)

最终更新: 2024/09/10

增量式处理是一种处理技术, 尽可能的避免重新处理源代码. 增量式处理的主要目的是减少典型的修改-编译-测试循环的处理时间. 请参见 Wikipedia 词条 增量计算 (https://en.wikipedia.org/wiki/Incremental_computing).

为了检测哪个源代码是 *脏的(dirty)* (也就是需要重新处理), KSP 需要处理器的帮助, 确定哪个输入源代码对应到哪个生成的输出. 为了改善这种经常很累赘, 而且容易出错的处理, KSP 设计目标是 只需要处理器使用的最少量的 *根源代码*, 作为起点来浏览代码结构. 也就是说, 如果 `KSNode` 从以下方式得到, 那么处理器需要将一个输出关联到对应的 `KSNode` 的源代码:

- `Resolver.getAllFiles`
- `Resolver.getSymbolsWithAnnotation`
- `Resolver.getClassDeclarationByName`
- `Resolver.getDeclarationsFromPackage`

目前增量式处理会默认启用. 要关闭它, 请设置 Gradle 属性 `ksp.incremental=false`. 要为依赖项和输出对应的脏文件集启用 log, 请使用 `ksp.incremental.log=true`. 你可以在 `build` 输出目录中找到这些 log 文件, 扩展名为 `.log`.

在 JVM 平台, 默认会追踪 classpath 中的变更, 以及 Kotlin 和 Java 源代码的变更. 如果要只追踪 Kotlin 和 Java 源代码的变更, 请设置 Gradle 属性 `ksp.incremental.intermodule=false`, 关闭对 classpath 中变更的追踪.

聚集(Aggregating) vs 隔离(Isolating)

与 Gradle 注解处理 (https://docs.gradle.org/current/userguide/java_plugin.html#sec:incremental_annotation_processing) 中的概念类似, KSP 支持 *聚集(Aggregating)* 和 *隔离(Isolating)* 模式. 注意, 与 Gradle 注解处理不同, KSP 将每个输出分类为聚集或隔离, 而不是对整个处理器分类.

聚集输出潜在的可以被任何输入变更影响, 不影响其他文件的删除文件除外. 意思就是说, 任何输入变更都会导致一次所有聚集输出的重构建, 因此, 会重新处理所有对应的注册过的, 新增的和修改的源代码文件.

例如, 收集带有一个特定注解的所有符号的输出, 会被认为是一个聚集输出.

隔离输出只依赖于特定的源代码. 对其他源代码的变更不会影响隔离输出. 注意, 与 Gradle 注解处理不同, 你可以对一个指定的输出定义多个源代码文件.

例如, 针对一个接口生成的实现类, 会被认为是隔离输出.

总结来说, 如果一个输出 可能依赖于新的或任何变更过的源代码, 那么它被认为是聚集输出. 否则, 是隔离输出.

下面是针对熟悉 Java 注解处理的读者的总结:

- 在一个隔离的 Java 注解处理器中, KSP 中所有的输出都是隔离的.
- 在一个聚集的 Java 注解处理器中, KSP 中有些输出可以是隔离的, 有些可以是聚集的.

实现细节

依赖关系通过输入和输出文件的关联来计算, 而不是通过注解. 这是一个多对多的关系.

由输入-输出关联导致的脏文件传递规则是:

1. 如果一个输入文件有变更, 它一定会被重新处理.
2. 如果一个输入文件有变更, 而且它关联到一个输出, 那么关联到同一个输出的所有其他输入文件也会被重新处理. 这个规则是传递性的, 也就是说, 文件无效会重复发生, 直到没有新的脏文件.
3. 关联到一个或多个聚集输出的所有输入文件都会被重新处理. 也就是说, 如果一个输入文件没有关联到任何聚集输出, 它不会被重新处理(除非它符合上面讲的规则 1 或 规则2).

原因如下:

1. 如果一个输入有变更, 可能会引入新的信息, 因此处理器需要对这个输入再次运行.
2. 一个输出由一组输入产生. 处理器可能需要所有的输入才能重新产生输出.
3. `aggregating=true` 代表, 一个输出可能潜在的依赖于新的信息, 可能来自新的文件, 或者既有的但被变更的文件. `aggregating=false` 代表, 处理器确定它的信息只来自特定的输入文件, 不会来自其它文件或新的文件.

示例 1

一个处理器读取 `A.kt` 中的类 `A` 和 `B.kt` 中的类 `B`, 其中 `A` 继承 `B`, 然后生成 `outputForA`. 处理器通过 `Resolver.getSymbolsWithAnnotation` 得到 `A`, 然后从 `A` 使用

`KSClassDeclaration.superTypes` 得到 `B`. 因为 包含 `B` 是由于 `A` 造成的, 在 `outputForA` 的 `dependencies` 中不需要指定 `B.kt`. 这种情况下你仍然可以指定 `B.kt`, 但不是必须的.

```
// A.kt
@Interesting
class A : B()

// B.kt
open class B

// Example1Processor.kt
class Example1Processor : SymbolProcessor {
    override fun process(resolver: Resolver) {
        val declA =
resolver.getSymbolsWithAnnotation("Interesting").first() as
KSClassDeclaration
        val declB = declA.superTypes.first().resolve().declaration
        // B.kt 不是必须的, 因为它可以被 KSP 推断为一个依赖项
        val dependencies = Dependencies(aggregating = true,
declA.containingFile!!)
        // outputForA.kt
        val outputName = "outputFor${declA.simpleName.asString()}"
        // outputForA 依赖于 A.kt 和 B.kt
        val output = codeGenerator.createNewFile(dependencies,
"com.example", outputName, "kt")
        output.write("// $declA : $declB\n".toByteArray())
        output.close()
    }
    // ...
}
```

示例 2

一个处理器读取 `sourceB`, 然后读取 `sourceA` 和 `outputB`, 然后生成 `outputA`.

如果修改了 `sourceA`:

- 如果 `outputB` 是聚集的, 那么 `sourceA` 和 `sourceB` 都会被重新处理.

- 如果 `outputB` 是隔离的, 那么只有 `sourceA` 会被重新处理.

如果添加了 `sourceC`:

- 如果 `outputB` 是聚集的, 那么 `sourceC` 和 `sourceB` 都会被重新处理.
- 如果 `outputB` 是隔离的, 那么只有 `sourceC` 会被重新处理.

如果删除了 `sourceA`, 那么没有任何代码需要重新处理.

如果删除了 `sourceB`, 那么没有任何代码需要重新处理.

如何判断文件是否为脏

一个脏文件 要么直接被用户 *修改*, 或者间接被其他脏文件 *影响*. KSP 通过 2 个步骤传播文件的变更:

- 通过 *解析追踪(Resolution Tracing)* 传播: (隐含的或明确的)解析一个类型引用, 是从一个文件浏览到另一个文件的唯一方式. 处理器解析一个类型引用时, 如果在一个被修改或被影响的文件中包含变更, 可能影响到解析结果, 那么这个变更将会影响包含这个引用的文件.
- 通过 *输入-输出对应关系(Input-Output Correspondence)* 传播: 一个源代码文件被修改或被影响时, 如果其他源代码文件与这个文件存在共同的输出, 那么其他源代码文件也都会被影响.

注意, 这 2 个步骤都是传递性的, 第 2 种会形成等价类(Equivalence Class).

报告 bug

要报告一个 bug, 请设置 Gradle 属性 `ksp.incremental=true` 和 `ksp.incremental.log=true`, 然后执行一次 clean 构建. 这个构建会产生 2 个 log 文件:

- `build/kspCaches/<source set>/logs/kspDirtySet.log`
- `build/kspCaches/<source set>/logs/kspSourceToOutputs.log`

然后你可以运行有 bug 的增量构建, 会生成 2 个新的 log 文件:

- `build/kspCaches/<source set>/logs/kspDirtySetByDeps.log`
- `build/kspCaches/<source set>/logs/kspDirtySetByOutputs.log`

这些 log 包含源代码和输出的文件名, 以及这些构建的时间戳.

多轮(Multiple Round)处理

最终更新: 2024/09/10

KSP 支持 *多轮(Multiple Round)处理*, 也就是通过多次步骤处理文件. 因此前一轮处理的输出可以供后一轮处理作为额外的输入.

对你的处理器的变更

为了使用多轮处理, `SymbolProcessor.process()` 函数需要对无效的符号返回延迟(deferred)符号列表 (`List<KSAnnotated>`). 请使用 `KSAnnotated.validate()` 来过滤无效的符号, 让它们延迟到下一轮.

以下示例代码演示如何使用有效性检查, 来延迟无效的符号:

```
override fun process(resolver: Resolver): List<KSAnnotated> {
    val symbols =
resolver.getSymbolsWithAnnotation("com.example.annotation.Builder")
    val result = symbols.filter { !it.validate() }
    symbols
        .filter { it is KSClassDeclaration && it.validate() }
        .map { it.accept(BuilderVisitor(), Unit) }
    return result
}
```

多轮处理的行为

将符号延迟到下一轮处理

处理器可以将特定符号的处理延迟到下一轮. 如果符号被延迟, 代表处理器在等待其他的处理器来提供更多的信息. 它可以根据需要继续延迟这个符号. 一旦另一个处理器提供了需要的信息, 处理器就可以处理被延迟的符号了. 处理器应该只延迟那些缺乏必要信息的无效符号. 因此, 处理器 **不应该** 延迟来自 classpath 的符号, KSP 也会过滤掉来自源代码以外的任何被延迟的符号.

比如, 一个处理器创建一个构建器 为一个被注解的类, 可能需要它的构造函数的所有参数类型都是有效的 (也就是说解析到一个具体的类型). 在第 1 轮中, 其中 1 个类型无法解析. 然后在第 2 轮中, 由于有了第 1 轮生成的文件, 这个类型可以解析了.

校验符号

决定符号是否应该延迟的一个便利方法是进行校验. 一个处理器应该知道为了正确的处理符号需要哪些信息. 注意, 校验通常需要类型解析, 类型解析的代价可能很高, 因此我们推荐只检查必须的信息. 继续上面的例子, 对于构建器处理器来说, 一个理想的校验只检查被注解的符号的构造函数的所有已解析的参数类型是否包含 `isError == false`.

KSP 提供一个默认校验工具. 详情请参见 高级内容 小节.

终止条件

当一整轮处理不再生成新的文件, 此时多轮处理会终止. 当终止条件达到时, 如果还存在未处理的延迟符号, KSP 会对每个带有未处理的延迟符号的处理器, 向 log 输出一个错误信息.

在每一轮中可以访问的文件

新生成的文件和已经存在的文件都可以通过一个 `Resolver` 访问. KSP 提供 2 个 API来访问文件: `Resolver.getAllFiles()` 和 `Resolver.getNewFiles()`. `getAllFiles()` 返回一个组合的 List, 包含已经存在的文件和新生成的文件, 而 `getNewFiles()` 只返回新生成的文件.

改为使用 getSymbolsAnnotatedWith()

为了避免对符号不必要的重新处理, `getSymbolsAnnotatedWith()` 只返回在新生成的文件中发现的符号, 以及在最后一轮处理中被延迟的符号.

创建处理器实例

一个处理器实例只创建一次, 因此你可以在处理器对象中保存信息, 供下一轮使用.

不同轮之间的信息一致性

所有的 KSP 符号都不能在不同轮之间重复使用, 因为前一轮生成的结果有可能导致解析结果发生改变. 但是, 由于 KSP 不允许修改已经存在的代码, 有些信息应该还是可以重复使用的, 比如一个符号的名称字符串值. 总结, 处理器可以保存前一轮的信息, 但需要记住, 在后续的轮中这些信息可能会无效.

错误和异常处理

如果发生了错误 (由处理器调用 `KSPLogger.error()` 来定义) 或异常, 处理在当前轮完毕之后会停止. 所有的处理器 会调用 `onError()` 方法, 而且 **不会** 调用 `finish()` 方法.

注意, 即使发生了错误, 其他处理器还会对这一轮继续正常的处理. 因此错误处理会发生在处理对这一轮完毕之后.

对于异常, KSP 会尝试区分来自 KSP 的异常和来自处理器的异常. 异常会导致处理立即终止, 并且会在 KSPLogger 中作为错误输出到 log. 来自 KSP 的异常应该报告给 KSP 开发者, 进行进一步调查. 在发生异常或错误的轮结束后, 所有的处理器将会调用 `onError()` 函数, 执行它们自己的错误处理.

在 `SymbolProcessor` 接口中, KSP 为 `onError()` 提供一个默认的无操作(no-op) 实现. 你可以覆盖这个方法, 提供你自己的错误处理逻辑.

高级内容

校验的默认行为

KSP 提供的默认校验逻辑, 会对被校验的符号所属的封闭范围(Enclosing Scope)之内的所有直接可到达(directly reachable)符号进行验证. 默认校验会检查 封闭范围中的引用是否是否可解析到一个具体的类型, 但不会递归深入被引用的类型, 对其进行校验.

编写你自己的校验逻辑

默认的校验行为可能不适用于情况. 你可以参考 `KSValidateVisitor` 编写你自己的校验逻辑, 方法是提供自定义的 `predicate` Lambda 表达式, 它会被 `KSValidateVisitor` 用来过滤需要被检查的符号.

在 Kotlin Multiplatform 中使用 KSP

最终更新: 2024/09/10

作为一个快速入门的示例, 可以参见 Kotlin Multiplatform 示例项目 (https://github.com/google/ksp/tree/main/examples/multiplatform), 其中定义了 KSP 处理器.

从 KSP 1.0.1 开始, 在跨平台项目中使用 KSP, 与在单一平台的 JVM 项目中类似. 主要区别是, 在依赖项中不是编写 `ksp(...)` 配置, 而是使用 `add(ksp<Target>)` 或 `add(ksp<SourceSet>)`, 指定哪个编译目标在编译之前需要符号处理.

```
plugins {
    kotlin("multiplatform")
    id("com.google.devtools.ksp")
}

kotlin {
    jvm {
        withJava()
    }
    linuxX64() {
        binaries {
            executable()
        }
    }
    sourceSets {
        val commonMain by getting
        val linuxX64Main by getting
        val linuxX64Test by getting
    }
}

dependencies {
    add("kspCommonMainMetadata", project(":test-processor"))
    add("kspJvm", project(":test-processor"))
    add("kspJvmTest", project(":test-processor")) // 不会进行任何处理,
因为对 JVM 平台没有测试代码
```

```
    // 对于 Linux x64 的 main 源代码集没有任何处理, 因为没有指定
kspLinuxX64
    add("kspLinuxX64Test", project(":test-processor"))
}
```

编译与处理

在跨平台项目中, 对每个平台 Kotlin 编译可能发生多次 (`main`, `test`, 或其他构建配置). 符号处理也是如此. 每存在一个 Kotlin 编译 task, 并且指定了对应的 `ksp<Target>` 或 `ksp<SourceSet>` 配置, 就会创建一个符号处理 task.

比如, 在上面的 `build.gradle.kts` 中, 有 4 个编译 task: common/metadata, JVM main, Linux x64 main, Linux x64 test, 以及 3 个符号处理 task: common/metadata, JVM main, Linux x64 test.

在 KSP 1.0.1+ 中不再使用 ksp(...) 配置

在 KSP 1.0.1 之前, 只有唯一一个, 统一的 `ksp(...)` 配置可以使用. 因此, 处理器要么对所有的编译目标适用, 要么不对任何编译目标适用. 注意, 即使是在传统的非跨平台项目中, `ksp(...)` 配置不仅适用于 main 源代码集, 如果存在 test 源代码集的话, 也会适用. 这就对构建时间带来了不必要的负担.

从 KSP 1.0.1 开始, 提供了对各个编译目标分别进行配置的功能, 如上面的示例所示. 将来:

1. 对于跨平台项目, `ksp(...)` 配置将被废弃, 并删除.
2. 对于单一平台项目, `ksp(...)` 配置将只适用于 main, 默认编译 task. 其他编译目标, 比如 `test`, 将需要指定 `kspTest(...)` 来适用处理器.

从 KSP 1.0.1 开始, 有一个早期预览版的 flag `-DallowAllTargetConfiguration=false`, 可以切换到更加高效率的模式. 如果目前的模式造成了性能问题, 请试用这个 flag. 在 KSP 2.0 中, 这个 flag 的默认值将会从 `true` 切换到 `false`.

在命令行运行 KSP

最终更新: 2024/09/10

KSP 是一个 Kotlin 编译器 plugin, 需要与 Kotlin 编译器一起运行. 请下载并解压缩它们.

```
#!/bin/bash

# Kotlin 编译器
wget
https://github.com/JetBrains/kotlin/releases/download/v1.9.21/kotlin
-compiler-1.9.21.zip
unzip kotlin-compiler-1.9.21.zip

# KSP
wget https://github.com/google/ksp/releases/download/1.9.21-
1.0.15/artifacts.zip
unzip artifacts.zip
```

要和 `kotlinc` 一起运行 KSP, 请向 `kotlinc` 传递 `-Xplugin` 选项.

```
-Xplugin=/path/to/symbol-processing-cmdline-1.9.21-1.0.15.jar
```

这与 `symbol-processing-1.9.21-1.0.15.jar` 不同, 它专门用于在 Gradle 中运行时和 `kotlin-compiler-embeddable` 一起使用. 而命令行的 `kotlinc` 则需要 `symbol-processing-cmdline-1.9.21-1.0.15.jar`.

你还需要 API jar.

```
-Xplugin=/path/to/symbol-processing-api-1.9.21-1.0.15.jar
```

完整的示例如下:

```
#!/bin/bash

KSP_PLUGIN_ID=com.google.devtools.ksp.symbol-processing
KSP_PLUGIN_OPT=plugin:$KSP_PLUGIN_ID
```

```
KSP_PLUGIN_JAR=./com/google/devtools/ksp/symbol-processing-
cmdline/1.9.21-1.0.15/symbol-processing-cmdline-1.9.21-1.0.15.jar
KSP_API_JAR=./com/google/devtools/ksp/symbol-processing-api/1.9.21-
1.0.15/symbol-processing-api-1.9.21-1.0.15.jar
KOTLINC=./kotlinc/bin/kotlinc

AP=/path/to/your-processor.jar

mkdir out
$KOTLINC \
        -Xplugin=$KSP_PLUGIN_JAR \
        -Xplugin=$KSP_API_JAR \
        -Xallow-no-source-files \
        -P $KSP_PLUGIN_OPT:apclasspath=$AP \
        -P $KSP_PLUGIN_OPT:projectBaseDir=. \
        -P $KSP_PLUGIN_OPT:classOutputDir=./out \
        -P $KSP_PLUGIN_OPT:javaOutputDir=./out \
        -P $KSP_PLUGIN_OPT:kotlinOutputDir=./out \
        -P $KSP_PLUGIN_OPT:resourceOutputDir=./out \
        -P $KSP_PLUGIN_OPT:kspOutputDir=./out \
        -P $KSP_PLUGIN_OPT:cachesDir=./out \
        -P $KSP_PLUGIN_OPT:incremental=false \
        -P $KSP_PLUGIN_OPT:apoption=key1=value1 \
        -P $KSP_PLUGIN_OPT:apoption=key2=value2 \
        $*
```

KSP FAQ

最终更新: 2024/09/10

为什么需要使用 KSP?

KSP 相比 kapt (kapt 编译器插件) 具有很多优点:

- 更快.
- 对于 Kotlin 使用者来说 API 更加流畅.
- 对生成的 Kotlin 源代码支持 多轮(Multiple Round)处理 (多轮(Multiple Round)处理).
- 设计时考虑了跨平台兼容性.

为什么 KSP 比 kapt 更快?

kapt 必须分析并解析所有的类型引用, 才能生成 Java 桩代码(stub), 而 KSP 只在需要的时候才解析类型引用. 将一些处理代理到 javac 也需要消耗时间.

此外, 与仅仅隔离和聚集相比, KSP 的 增量处理(Incremental Processing)模型 (增量式处理(Incremental Processing)) 拥有更细的粒度. 可以有更多的机会避免重新处理所有信息. 而且, 由于 KSP 动态追踪符号的解析, 一个源代码文件的变更不太容易影响到其他文件, 因此需要处理的源代码文件比较少. 这是 kapt 无法做到的, 因为它将处理代理给 javac.

KSP 是 Kotlin 专有的吗?

KSP 也能处理 Java 源代码. API 是统一的, 也就是说, 当你解析 Java 类和 Kotlin 类时, 在 KSP 中会得到统一的数据结构.

如何更新 KSP?

KSP 由 API 和实现构成. API 很少改变, 并且会保持向后兼容: 会出现新的接口, 但旧的接口不会改变. 实现则绑定到特定的编译器版本. 在每次新版本发布时, 支持的编译器版本都会改变.

处理器只依赖于 API, 因此不会绑定到编译器版本. 但是, 处理器的使用者在他们的项目中升级编译器版本时, 也需要升级 KSP 版本. 否则, 会出现以下错误:

```
ksp-a.b.c is too old for kotlin-x.y.z. Please upgrade ksp or
downgrade kotlin-gradle-plugin
```

> 处理器的使用者不需要更新处理器的版本, 因为处理器只依赖于 API.

比如, 某个处理器在 KSP 1.0.1 中发布并测试, 这个 KSP 版本严格绑定到 Kotlin 1.6.0. 要让这个处理器在在 Kotlin 1.6.20 中工作, 你只需要将 KSP 升级到为 Kotlin 1.6.20 构建的某个版本(比如, KSP 1.1.0).

我可以在旧版本的 Kotlin 编译器中使用新版本的 KSP 实现吗?

如果语言版本相同, Kotlin 编译器应该支持向后兼容. 大多数情况下 Kotlin 编译器只是小版本的升级. 如果你需要新版本的 KSP 实现, 请对应的升级 Kotlin 编译器.

KSP 的更新频度如何?

KSP 尽可能的遵循 语义化版本(Semantic Versioning) (https://semver.org/). 对于 KSP 版本 `major.minor.patch`,

- `major` 表示不兼容的 API 变更. 这部分的版本更新, 没有预定的时间表.
- `minor` 表示新的功能特性. 这部分的版本更新, 大约每季度一次.
- `patch` 表示bug 修正, 以及支持新的 Kotlin 发布版. 这部分的版本更新, 大约每月一次.

通常在 Kotlin 新版本发布之后, 几天之内会发布对应的 KSP 版本, 包括 预发布版 (Beta 或 RC) (参加 Kotlin EAP 项目).

除 Kotlin 之外, 是否还有其它的库版本要求?

以下是库/基础设施的版本要求:

- Android Gradle Plugin 4.1.0+
- Gradle 6.5+

KSP 未来的发展路线图如何?

我们有以下计划:

- 支持 新的 Kotlin 编译器 (Kotlin 发展路线图)
- 改进对跨平台项目的支持. 比如, 一部分编译目标上运行 KSP / 在多个编译目标之间共用计算结果.
- 性能改进. 需要进行很多优化工作!
- 不断修正 bug.

如果你有什么想法想要讨论, 欢迎通过 #Kotlin Slack 的 ksp 频道 (https://kotlinlang.slack.com/archives/C013BA8EQSE) 与我们联络 (得到邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up)). 也欢迎提交 GitHub issue 或 Feature Request (https://github.com/google/ksp/issues) 或 Pull Request!

学习资料概述

最终更新: 2024/09/10

你可以通过以下资料学习 Kotlin:

- 基本语法 (基本语法) – Kotlin 语法简要介绍.
- 惯用法 (惯用法) – 针对常见场景, 学习如何编写符合 Kotlin 习惯代码.
  - Java 代码向 Kotlin 迁移指南: 字符串 (Java 和 Kotlin 中的字符串) – 学习如何执行 Java 与 Kotlin 中的常见字符串处理任务.
  - Java 代码向 Kotlin 迁移指南: 集合(Collection) (Java 和 Kotlin 中的集合(Collection)) — 学习如何执行 Java 与 Kotlin 中的常见集合(Collection)处理任务.
  - Java 代码向 Kotlin 迁移指南: 可空性(Nullability) (Java 和 Kotlin 中的可空性(Nullability)) — 学习如何处理 Java 与 Kotlin 中的可空性(Nullability).
- Kotlin Koan (Kotlin Koan) – 学习 Kotlin 语法的完整练习题. 每个练习从一个失败的 unit test 开始, 你的任务是让测试通过. 推荐有 Java 经验的开发者学习.
- 通过示例学习 Kotlin (https://play.kotlinlang.org/byExample/overview) – 一系列小而简单的示例程序, 演示 Kotlin 语法.
- JetBrains Academy 的 Kotlin 核心课程 (https://hyperskill.org/tracks?category=4&utm_source=jbkotlin_hs&utm_medium=referral&utm_campaign=kotlinlang-docs&utm_content=button_1&utm_term=22.03.23) – 一步一步创建可真正工作的应用程序, 学习 Kotlin 的所有基本知识.
- Kotlin 书籍 (Kotlin 书籍) – 我们审阅并推荐的 Kotlin 学习书籍.
- Kotlin 小技巧 (Kotlin 小技巧) – 观看简短视频, Kotlin 开发组会向你演示使用 Kotlin 的更加高效更加符合惯用法的方式, 让你的编码工作更加有趣.
- Advent of Code 编程题 (使用 Kotlin 惯用法的 Advent of Code) – 完成小而有趣的任务, 学习 Kotlin 惯用法, 并测验你的语言技能.
- Kotlin 实际动手(hands-on)教程 (Kotlin 动手课程(hands-on)) – 完成比较长的教程, 完整掌握 Kotlin 技术. 这些教程围绕一个主题, 指导你完成独立的项目.

- 针对 Java 开发者的 Kotlin 课程 (https://www.coursera.org/learn/kotlin-for-java-developers) – 通过 Coursera 的课程, 学习 Java 与 Kotlin 之间的相似之处与区别.
- Kotlin 文档 (PDF 格式) (Kotlin 文档 (PDF 格式)) – 可供离线阅读的文档.

Kotlin Koan

最终更新: 2024/09/10

Kotlin Koan 是一系列练习, 主要针对 Java 开发者, 帮助你熟悉 Kotlin 语法. 每个练习从一个失败的 unit test 开始, 你的任务是让测试通过. 你可以通过以下任何一种方式来完成 Kotlin Koan 任务:

- 你可以使用 Koans online (https://play.kotlinlang.org/koans).
- 你也可以安装 JetBrains Academy plugin (https://plugins.jetbrains.com/plugin/10081-jetbrains-academy/docs/install-jetbrains-academy-plugin.html), 并 选择 Kotlin Koan 课程 (https://plugins.jetbrains.com/plugin/10081-jetbrains-academy/docs/learner-start-guide.html?section=Kotlin%20Koans), 直接在 IntelliJ IDEA 或 Android Studio 之内完成这些任务 .

无论使用哪种方式解答 Koan 练习, 你可以都看到每个任务的答案:

- 在 online 版中, 可以点击 **Show answer**.
- 在 JetBrains Academy plugin 中, 先尝试完成任务, 如果你的答案不正确, 可以选择 **Peek solution**.

我们推荐你在完成任务之后再查看答案, 比较你的解答与系统提供的答案之间的差别. 注意不要作弊!

Kotlin 动手课程(hands-on)

最终更新: 2024/09/10

这里是一系列动手教程, 你可以使用 Kotlin 中各种不同的技术, 针对多种平台, 创建应用程序. 练习题分解为一系列的步骤, 引导你完成每个步骤.

使用 Kotlin coroutine 和 RSocket 创建响应式(Reactive) Spring Boot 应用程序

使用 Spring Boot 和 Kotlin 创建一个简单的聊天应用程序, 从语法的角度, 学习 Kotlin 用于服务器端开发的好处.

开始 (https://spring.io/guides/tutorials/spring-webflux-kotlin-rsocket/)

使用 React 和 Kotlin/JS 创建 Web 应用程序

使用 Kotlin/JS 创建一个 React 应用程序, 了解如何利用 Kotlin 的类型系统, 库生态环境, 以及互操作功能.

开始 (教程 - 使用 React 和 Kotlin/JS 创建 Web 应用程序)

使用 Spring Boot 和 Kotlin 创建 Web 应用程序

结合 Spring Boot 和 Kotlin 的力量, 创建一个简单的 Blog 应用程序.

开始 (https://spring.io/guides/tutorials/spring-boot-kotlin/)

使用 Ktor 创建 HTTP API

为你的应用程序创建后端 API, 响应 HTTP 请求.

开始 (https://ktor.io/docs/creating-http-apis.html)

使用 Ktor 创建 WebSocket 聊天程序

使用 Ktor 创建一个简单的聊天应用程序, 包括 JVM 服务器和 JVM 客户端.

开始 (https://ktor.io/docs/creating-web-socket-chat.html)

使用 Ktor 创建一个交互式网站

学习如何提供文件服务, 使用模板引擎, 例如 Freemarker 和 kotlinx.html DSL, 处理 Ktor 的表单输入.

开始 (https://ktor.io/docs/creating-interactive-website.html)

介绍 Kotlin coroutine 和 channel

学习 Kotlin 中的 coroutine, 以及如何使用 channel 在 coroutine 之间进行通信.

开始 (教程 - 协程与通道(Channel))

介绍 Kotlin/Native

使用 Kotlin/Native 和 libcurl 创建一个简单的 HTTP 客户端, 能够在多个平台上作为原生代码运行.

开始 (教程 - 使用 C Interop 和 libcurl 创建应用程序)

Kotlin Multiplatform: 网络与数据存储

学习如何使用 Kotlin Multiplatform, Ktor 以及 SQLDelight, 为 Android 和 iOS 创建移动应用程序.

开始 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-ktor-sqldelight.html)

使用 Kotlin Multiplatform 针对 iOS 和 Android 平台的开发

学习如何使用 Kotlin Multiplatform 创建移动应用程序, 能够同时在 iOS 和 Android 上运行.

开始 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-create-first-app.html)

Kotlin 小技巧

最终更新: 2024/09/10

Kotlin 小技巧是一系列的短视频, Kotlin 开发组成员向你演示, 使用 Kotlin 编写代码时更加高效, 更加符合惯用法, 更加有趣的方式.

不要错过新的 Kotlin 小技巧视频, 订阅我们的 YouTube 频道 (https://www.youtube.com/channel/UCP7uiEZIqci43m22KDl0sNw).

Kotlin 中的 null + null

在 Kotlin 中执行 `null + null` 会发生什么情况, 返回结果是什么? 在我们最新的一个 Kotlin 小技巧视频中, Sebastian Aigner 将解开这个谜题. 此外, 他还会演示为什么我们不用害怕可为 null 的值:

去除集合中的重复元素

得到了一个包含重复元素的 Kotlin 集合吗? 需要只包含唯一元素的集合? 请看这个 Kotlin 小技巧视频, Sebastian Aigner 向你演示如何从你的 List 中删除重复元素, 或者转换为 Set:

挂起(Suspend)与内联(Inline) 之谜

为什么 `repeat()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/repeat.html), `map()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map.html) 和 `filter()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html) 之类的函数在它们的 Lambda 表达式参数中接受挂起函数, 尽管它们的方法签名并没有标记为与协程相关? 在这一期的 Kotlin 小技巧视频中, Sebastian Aigner 会解开这个谜题: 与 inline 修饰符有关:

使用完整限定名称(fully qualified name)解决声明的遮盖(Shadowing)问题

遮盖(Shadowing)是指在同一个作用域内出现 2 个相同名称的声明. 这时, 会使用哪一个? 在这一期的 Kotlin 小技巧视频中, Sebastian Aigner 向你演示一个简单的 Kotlin 技巧, 使用完整限定名称(fully qualified name), 来正确调用你需要的那个函数:

在 Elvis 操作符中返回或抛出异常

Elvis 操作符 ("Elvis 操作符" in "Null 值安全性") 再次进入我们的视野! Sebastian Aigner 向你解释为什么这个操作符使用那位著名歌手的名字来命名, 以及在 Kotlin 中如何使用 `?:` 进行返回, 或抛出

异常. 幕后的神奇实现是什么呢? 请参见 Nothing 类型
(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-nothing.html).

解构声明(Destructuring Declaration)

使用 Kotlin 中的 解构声明(Destructuring Declaration) (解构声明), 你可以从单个对象一次性创建多个变量. 在这个视频中, Sebastian Aigner 向你演示各种可以解构的对象 – Pair, List, Maps, 等等. 你自己的对象又如何呢? Kotlin 的组件函数对此也提供了答案:

可为 null 的值的操作符函数

在 Kotlin 中, 你可以对你的类覆盖操作符, 比如加和减, 并提供你自己的逻辑. 但如果你想要在操作符的左侧和右侧都允许 null 值, 这时该怎么办? 在这个视频中, Sebastian Aigner 解答了这个问题:

测量代码执行时间

请看 Sebastian Aigner 介绍 `measureTimedValue()`
(https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/measure-timed-value.html) 函数, 学习如何测量你的代码的执行时间:

循环的改进

在这个视频中, Sebastian Aigner 会演示如何改进 循环 ("for 循环" in "条件与循环"), 让你的代码更加易读, 易懂, 简洁:

字符串

在这一期中, Kate Petrova 演示 3 个小技巧, 帮助你在 Kotlin 中处理 字符串 (字符串):

Elvis 操作符的复杂运用

在这个视频中, Sebastian Aigner 演示如何向 Elvis 操作符 ("Elvis 操作符" in "Null 值安全性") 添加更多逻辑, 比如对操作符右侧输出日志:

Kotlin 集合

在这一期中, Kate Petrova 演示 3 个小技巧, 帮助你处理 Kotlin 集合 (集合(Collection)概述):

下一步做什么?

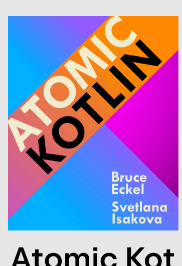
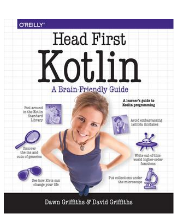
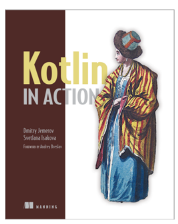
* 在我们的 YouTube 播放列表 (https://youtube.com/playlist?list=PLlFc5cFwUnmyDrc-mwwAL9cYFkSHoHHz7) 中查看 Kotlin 小技巧的完整列表
* 学习如何编写 针对常见问题的 Kotlin 惯用代码 (惯用法)

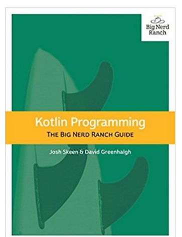
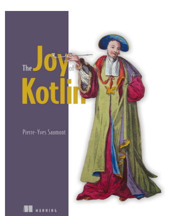
Kotlin 书籍

最终更新: 2024/09/10

越来越多的作者在撰写关于学习 Kotlin 的各种语言的书籍. 我们非常感谢所有这些作者, 并感谢他们的努力, 帮助我们增加了 Kotlin 专业开发者的数量.

以下只是我们审阅过, 并推荐用来学习 Kotlin 的少部分书籍. 你可以在 我们的社区网站 (https://kotlin.link/) 找到更多书籍.

 Atomic Kotlin	**Atomic Kotlin (https://www.atomickotlin.com/atomickotlin/) 既适合于初学者, 也适合于有经验的程序员!** **作者是 Bruce Eckel, 他还撰写过获得许多奖项的《Thinking in C++》和《Thinking in Java》, 以及 Svetlana Isakova, JetBrains 公司的 Kotlin 开发者 Advocate, 本书将语言的概念分解过小的, 易于理解的 "原子(atom)", 以及由许多练习组成的自由课程, 这些练习可以直接在 IntelliJ IDEA 之内得到提示和解答!**
 Head First Kotlin	Head First Kotlin (https://www.oreilly.com/library/view/head-first-kotlin/9781491996683/) 完整的介绍 Kotlin 编程. 这本手册超越语法讲解和如何解决问题的手册之外, 通过独特的方法帮助你学习 Kotlin 语言, 教导你如何象一个伟大的 Kotlin 开发者那样思考. 你将学到一切知识, 从语言基础, 到集合, 泛型, Lambda 表达式, 以及高阶函数. 在这个过程中, 你将接触到面向对象编程以及函数式编程. 如果你希望真正理解 Kotlin, 这本书很适合你.
 Kotlin in Action	Kotlin in Action (https://manning.com/books/kotlin-in-action) 教你使用 Kotlin 语言开发真正产品程度的应用程序. 本书针对有经验的 Java 开发者, 包含大量示例, 内容比大多数编程语言书籍更加丰富, 涵盖有趣的主题, 比如使用自然语言语法构建 DSL. 本书作者是 Dmitry Jemerov 和 Svetlana Isakova, Kotlin 开发组的成员. 第 6 章, 讲解 Kotlin 类型系统, 以及第 11 章, 讲解 DSL, 可以在 出版社网站 (https://www.manning.com/books/kotlin-in-action#downloads) 免费预览.

 Kotlin Programming: The Big Nerd Ranch Guide	Kotlin Programming: The Big Nerd Ranch Guide (https://www.amazon.com/Kotlin-Programming-Nerd-Ranch-Guide/dp/0135161630) 通过这本书你将会学到任何高效的使用 Kotlin 语言, 它通过仔细考虑的示例, 教导你 Kotlin 优雅的编程风格和功能特性. 从第 1 原则开始, 你将学习 Kotlin 的高级用法, 学会如何使用更少的代码创建更加可靠的程序.
 Programming Kotlin	Programming Kotlin (https://pragprog.com/book/vskotlin/programming-kotlin) 由 Venkat Subramaniam 撰写. 程序员不仅仅只是使用 Kotlin, 他们热爱 Kotlin. 甚至 Google 也采用它作为 Android 开发的第一语言. 使用 Kotlin, 你可以混合使用命令式, 函数式程, 以及面向对象风格的编程方式, 利用最适合目前问题的方式. 在本书中, 你可以通过易于理解的示例, 学习这个高度简洁, 流程, 优雅, 表达能力强的静态类系语言, 学会使用它的各种功能特性. 学习编写易于维护的, 高性能的 JVM 和 Android 应用程序, 创建 DSL, 异步程序开发, 等等.
 The Joy of Kotlin	The Joy of Kotlin (https://www.manning.com/books/the-joy-of-kotlin) 教你正确的 Kotlin 编程方式. 在这本内涵丰富的书中, 你将通过学习编程技术掌握 Kotlin 语言, 使你成为更好的开发者, 无论你使用什么语言. Kotlin 自然的支持函数式编程, 因此 seasoned 作者 Pierre-Yves Saumont 首先回顾函数式编程的原则, 包括不可变性(Immutability), 参照透明度 (Referential transparency), 以及隔离函数与效果. 然后, 你将会深入到在真实世界使用 Kotlin 的情况, 学习正确的处理错误以及数据, 包括共享的可变状态, 以及使用延迟加载. 本书将会改变你编程的方式 — 还会在你开始阅读之后享受到一些乐趣.

使用 Kotlin 惯用法的 Advent of Code

最终更新: 2024/09/10

Advent of Code (https://adventofcode.com/) 是每年 12 月举办一次的活动, 从 12 月 1 日开始到 25 日, 每天发布一道节日主题的谜题. 经 Advent of Code 创建者 Eric Wastl (http://was.tl/) 授权, 我们演示如何使用 Kotlin 风格的惯用法来解决这些谜题:

- Advent of Code 2023 (https://www.youtube.com/playlist?list=PLlFc5cFwUnmzk0wvYW4aTl57F2VNkFisU)
- Advent of Code 2022
- Advent of Code 2021
- Advent of Code 2020

准备进入 Advent of Code

我们会讲解使用 Kotlin 解决 Advent of Code 问题的基本技巧:

- 阅读我们的 关于 Advent of Code 2021 的 Blog (https://blog.jetbrains.com/kotlin/2021/11/advent-of-code-2021-in-kotlin/)
- 使用 这个 GitHub 模板 (https://github.com/kotlin-hands-on/advent-of-code-kotlin-template) 来创建项目
- 观看 Kotlin Developer Advocate, Sebastian Aigner 的入门视频:

Advent of Code 2022

第 1 天: Calorie counting

学习 Kotlin Advent of Code 模板 (https://github.com/kotlin-hands-on/advent-of-code-kotlin-template), 以及在 Kotlin 中处理字符串和集合的便利函数, 例如 `maxOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-of.html) 和 `sumOf()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum-of.html). 了解扩展函数如何帮助你以更好的方式构建解决方案.

- 在 Advent of Code (https://adventofcode.com/2022/day/1) 阅读题目内容
- 观看视频中的解答:

▶ Advent of Code 2022, 第 1 天 | Kotlin (https://www.youtube.com/watch?v=ntbsbqLCKDs)

第 2 天: Rock paper scissors

理解 Kotlin 中对 `Char` 类型的操作，了解在模式匹配中如何使用 `Pair` 类型和 `to` 构造器. 理解如何使用 `compareTo()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparable/compare-to.html) 函数对你自己的对象排序.

- 在 Advent of Code (https://adventofcode.com/2022/day/2) 阅读题目内容
- 观看视频中的解答:

▶ Advent of Code 2022, 第 2 天 | Kotlin (https://www.youtube.com/watch?v=Fn0SY2yGDSA)

第 3 天: Rucksack reorganization

学习 kotlinx.benchmark (https://github.com/Kotlin/kotlinx-benchmark) 库如何帮助你理解你的代码的性能特性. 了解 `intersect` 等 Set 操作如何帮助你选择重叠的数据, 查看同一解决方案的不同具体实现之间的性能比较.

- 在 Advent of Code (https://adventofcode.com/2022/day/3) 阅读题目内容
- 观看视频中的解答:

▶ Advent of Code 2022, 第 3 天 | Kotlin (https://www.youtube.com/watch?v=IPLfo4zXNjk)

第 4 天: Camp cleanup

`infix` 和 `operator` 函数如何提升你的代码的表现能力, 以及 `String` 和 `IntRange` 类型的扩展函数如何简化输入解析的工作.

- 在 Advent of Code (https://adventofcode.com/2022/day/4) 阅读题目内容
- 观看视频中的解答:

▶ Advent of Code 2022, 第 4 天 | Kotlin (https://www.youtube.com/watch?v=dBIbr55YS0A)

第 5 天: Supply stacks

了解如何使用工厂函数构建更加复杂的对象, 如何使用正规表达式, 以及双向的(Double-Ended) `ArrayDeque` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-array-deque/) 类型.

- 在 Advent of Code (https://adventofcode.com/2022/day/5) 阅读题目内容
- 观看视频中的解答:

▶ Advent of Code 2022, 第 5 天 | Kotlin (https://www.youtube.com/watch?v=lKq6r5Nt8Yo)

第 6 天: Tuning trouble

查看如何使用 kotlinx.benchmark (https://github.com/Kotlin/kotlinx-benchmark) 库进行更加深入的性能调查, 比较同一个解决方案的16种不同的遍体的性能特性.

- 在 Advent of Code (https://adventofcode.com/2022/day/6) 阅读题目内容
- 观看视频中的解答:

▶ Advent of Code 2022, 第 6 天 | Kotlin (https://www.youtube.com/watch?v=VbBhaQhW0zk)

第 7 天: No space left on device

学习如何构建树结构模型, 查看一个示例程序, 演示如何通过编程方式生成 Kotlin 代码.

- 在 Advent of Code (https://adventofcode.com/2022/day/7) 阅读题目内容
- 观看视频中的解答:

▶ Advent of Code 2022, 第 7 天 | Kotlin (https://www.youtube.com/watch?v=Q819VW8yxFo)

第 8 天: Treetop tree house

学习 `sequence` 构建器的实际使用, 以及一个程序最初的草稿和符合 Kotlin 惯用法的解决方案之间能有多大的差异 (和特邀嘉宾 Roman Elizarov 一起!).

- 在 Advent of Code (https://adventofcode.com/2022/day/8) 阅读题目内容
- 观看视频中的解答:

Advent of Code 2022, 第 8 天 | Kotlin (https://www.youtube.com/watch?v=6d6FXFh-UdA)

第 9 天: Rope bridge

学习 `run` 函数, 带标签的返回(Labeled Return), 以及便利的标准库函数, 例如 `coerceIn`, 或 `zipWithNext`. 学习如何使用 `List` 和 `MutableList` 构建器构建指定大小的 List, 查看这个题目基于 Kotlin 的可视化.

- 在 Advent of Code (https://adventofcode.com/2022/day/9) 阅读题目内容
- 观看视频中的解答:

Advent of Code 2022, 第 9 天 | Kotlin (https://www.youtube.com/watch?v=ShU9dNUa_3g)

第 10 天: Cathode-ray tube

学习值范围和 `in` 操作符如何让数值范围的检查变得更加自然, 如何将函数参数转换为接受者, 简要的探索 `tailrec` 修饰符.

- 在 Advent of Code (https://adventofcode.com/2022/day/10) 阅读题目内容
- 观看视频中的解答:

Advent of Code 2022, 第 10 天 | Kotlin (https://www.youtube.com/watch?v=KVyeNmFHoL4)

第 11 天: Monkey in the middle

学习如何从可变的、命令式(imperative)的代码转变为更加函数式的方案, 这种方案使用不可变的、只读的数据结构. 学习上下文接受者(Context Receiver), 以及我们的嘉宾如何为 Advent of Code 构建他自己的可视化库.

* 在 Advent of Code (https://adventofcode.com/2022/day/11) 阅读题目内容
* 观看视频中的解答:

▶ Advent of Code 2022, 第 11 天 | Kotlin (https://www.youtube.com/watch?v=1eBSyPe_9j0)

第 12 天: Hill Climbing algorithm

使用队列, `ArrayDeque`, 函数引用, 以及 `tailrec` 修饰符, 用 Kotlin 解决路径寻找问题.

* 在 Advent of Code (https://adventofcode.com/2022/day/12) 阅读题目内容
* 观看视频中的解答:

▶ Advent of Code 2022, 第 12 天 | Kotlin (https://www.youtube.com/watch?v=tJ74hi_3sk8)

Advent of Code 2021

> ⚠ 阅读我们的 关于 Advent of Code 2021 的 Blog (https://blog.jetbrains.com/kotlin/2021/11/advent-of-code-2021-in-kotlin/)

第 1 天: Sonar sweep

使用窗口和计数函数, 来处理整数的对(Pair)和三元组(Triplet).

* 在 Advent of Code (https://adventofcode.com/2021/day/1) 阅读题目内容
* 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/12/advent-of-code-2021-in-kotlin-day-1) 查看 Anton Arhipov 的解答, 或观看这个视频:

▶ Advent of Code 2021 in Kotlin, 第 1 天: Sonar Sweep (https://www.youtube.com/watch?v=76IzmtOyiHw)

第 2 天: Dive!

学习解构声明和 `when` 表达式.

* 在 Advent of Code (https://adventofcode.com/2021/day/2) 阅读题目内容

- 在 GitHub (https://github.com/asm0dey/aoc-2021/blob/main/src/Day02.kt) 查看 Pasha Finkelshteyn 的解答, 或观看这个视频:

Advent of Code 2021 in Kotlin, 第 2 天: Dive! (https://www.youtube.com/watch?v=4A2WwniJdNc)

第 3 天: Binary diagnostic

学习处理二进制数值的不同方式.

- 在 Advent of Code (https://adventofcode.com/2021/day/3) 阅读题目内容
- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/12/advent-of-code-2021-in-kotlin-day-3/) 查看 Sebastian Aigner 的解答, 或观看这个视频:

Advent of Code 2021 in Kotlin, 第 3 天: Binary Diagnostic (https://www.youtube.com/watch?v=mF2PTnnOi8w)

第 4 天: Giant squid

学习如何解析输入, 介绍用于更加便利的处理的一些领域类(Domain Class).

- 在 Advent of Code (https://adventofcode.com/2021/day/4) 阅读题目内容
- 在 GitHub (https://github.com/antonarhipov/advent-of-code-2021/blob/main/src/Day04.kt) 查看 Anton Arhipov 的解答, 或观看这个视频:

Advent of Code 2021 in Kotlin, 第 4 天: Giant Squid (https://www.youtube.com/watch?v=wL6sEoLezPQ)

Advent of Code 2020

你可以在我们的 GitHub 代码仓库 (https://github.com/kotlin-hands-on/advent-of-code-2020/) 找到Advent of Code 2020 谜题的所有解答.

第 1 天: Report repair

学习输入处理, 遍历列表, 通过不同的方法构建 Map, 使用 `let` ("let 函数" in "作用域函数(Scope Function)") 函数简化你的代码.

- 在 Advent of Code (https://adventofcode.com/2020/day/1) 阅读题目内容
- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/07/advent-of-code-in-idiomatic-kotlin/) 查看 Svetlana Isakova 的解答, 或观看这个视频:

▶ 和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #1 (https://www.youtube.com/watch?v=o4emra1xm88)

第 2 天: Password philosophy

学习字符串工具函数, 正规表达式, 集合上的操作, 以及如何使用 `let` ("let 函数" in "作用域函数 (Scope Function)") 函数变换你的表达式.

- 在 Advent of Code (https://adventofcode.com/2020/day/2) 阅读题目内容
- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/07/advent-of-code-in-idiomatic-kotlin-day2/) 查看 Svetlana Isakova 的解答, 或观看这个视频:

▶ 和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #2 (https://www.youtube.com/watch?v=MyvJ7G6aErQ)

第 3 天: Toboggan trajectory

比较命令式编程与函数式编程风格, 使用 pair 和 `reduce()` (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce.html) 函数, 在列选择模式(Column Selection Mode)下编辑代码, 修正整数溢出问题.

- 在 Advent of Code (https://adventofcode.com/2020/day/3) 阅读题目内容
- 在 GitHub (https://github.com/kotlin-hands-on/advent-of-code-2020/blob/master/src/day03/day3.kt) 查看 Mikhail Dvorkin 的解答, 或观看这个视频:

▶ 和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #3 (https://www.youtube.com/watch?v=ounCIclwOAw)

第 4 天: Passport processing

使用 `when` ("when 表达式" in "条件与循环") 表达式, 学习如何进行输入校验: 工具函数, 使用数值范围, 检查成员是否属于集合, 匹配特定的正规表达式.

- 在 Advent of Code (https://adventofcode.com/2020/day/4) 阅读题目内容

- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/09/validating-input-advent-of-code-in-kotlin/) 查看 Sebastian Aigner 的解答, 或观看这个视频:

和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #4 (https://www.youtube.com/watch?v=-kltG4Ztv1s)

第 5 天: Binary boarding

使用 Kotlin 标准库函数 (`replace()`, `toInt()`, `find()`) 处理数值的二进制表达, 学习强大的局部函数, 学习如何使用 Kotlin 1.5 的 `max()` 函数.

- 在 Advent of Code (https://adventofcode.com/2020/day/5) 阅读题目内容
- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/09/idiomatic-kotlin-binary-representation/) 查看 Svetlana Isakova 的解答, 或观看这个视频:

和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #5 (https://www.youtube.com/watch?v=XEFna3xyxeY)

第 6 天: Custom customs

学习如何分组并统计字符串和集合中的字符, 使用标准库函数: `map()`, `reduce()`, `sumOf()`, `intersect()`, 和 `union()`.

- 在 Advent of Code (https://adventofcode.com/2020/day/6) 阅读题目内容
- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/09/idiomatic-kotlin-set-operations/) 查看 Anton Arhipov 的解答, 或观看这个视频:

和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #6 (https://www.youtube.com/watch?v=QLAB0kZ-Tqc)

第 7 天: Handy haversacks

学习如何使用正规表达式, 在 Kotlin 代码中 使用 Java 的 HashMap 的 `compute()` 方法, 动态计算 Map 中的值, 使用 `forEachLine()` 函数读取文件, 比较两种查找算法: 深度优先查找和广度优先查找.

- 在 Advent of Code (https://adventofcode.com/2020/day/7) 阅读题目内容

- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/09/idiomatic-kotlin-traversing-trees/) 查看 Pasha Finkelshteyn 的解答, 或观看这个视频:

▶ 和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #7 (https://www.youtube.com/watch?v=KyZiveDXWHw)

第 8 天: Handheld halting

使用封闭类和 Lambda 表达式来表达指令, 使用 Kotlin Set 在程序执行中查找循环, 使用序列和 `sequence { }` 构建函数, 创建延迟计算的集合, 试验试验性的 `measureTimedValue()` 函数来检查性能统计指标.

- 在 Advent of Code (https://adventofcode.com/2020/day/8) 阅读题目内容
- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/10/idiomatic-kotlin-simulating-a-console/) 查看 Sebastian Aigner 的解答, 或观看这个视频:

▶ 和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #8 (https://www.youtube.com/watch?v=0GWTTSMatO8)

第 9 天: Encoding error

学习 Kotlin 中的不同方式操纵 List, 使用 `any()`, `firstOrNull()`, `firstNotNullOfOrNull()`, `windowed()`, `takeIf()`, 和 `scan()` 函数, 这些函数是 Kotlin 编程风格的典型例子.

- 在 Advent of Code (https://adventofcode.com/2020/day/9) 阅读题目内容
- 在 Kotlin Blog (https://blog.jetbrains.com/kotlin/2021/10/idiomatic-kotlin-working-with-lists/) 查看 Svetlana Isakova 的解答, 或观看这个视频:

▶ 和 Kotlin Team 一起学习 Kotlin: Advent of Code 2020 #9 (https://www.youtube.com/watch?v=vj3J9MuF1ml)

下一步做什么?

- 在 Kotlin Koans (Kotlin Koan) 中完成更多任务

- 通过 JetBrains Academy 的 Kotlin 核心教程 (https://hyperskill.org/tracks?category=4&utm_source=jbkotlin_hs&utm_medium=referral&utm_campaign=kotlinlang-docs&utm_content=button_1&utm_term=22.03.23) 创建真实工作的应用程序

使用 JetBrains Academy plugin 学习 Kotlin

最终更新: 2024/09/10

JetBrains Academy plugin (https://plugins.jetbrains.com/plugin/10081-jetbrains-academy) 可用于 Android Studio (https://developer.android.com/studio) 和 IntelliJ IDEA (https://www.jetbrains.com/idea/), 使用它你可以通过代码实践来学习 Kotlin.

请阅读 学习者入门指南 (https://plugins.jetbrains.com/plugin/10081-jetbrains-academy/docs/learner-start-guide.html?section=Kotlin%20Koans), 这篇指南将会指导你在 IntelliJ IDEA 之内学习 Kotlin Koans 课程. 你可以解决这些交互式编码问题, 并在 IDE 中立即得到结果反馈.

如果你希望使用 JetBrains Academy plugin 进行教学, 请参见 使用 JetBrains Academy plugin 教授 Kotlin (使用 JetBrains Academy plugin 教授 Kotlin).

使用 JetBrains Academy plugin 教授 Kotlin

最终更新: 2024/09/10

JetBrains Academy plugin (https://plugins.jetbrains.com/plugin/10081-jetbrains-academy), 可用于 Android Studio (https://developer.android.com/studio) 和 IntelliJ IDEA (https://www.jetbrains.com/idea/), 使用它你可以通过代码实践来教授 Kotlin.

请阅读 教授者入门指南 (https://plugins.jetbrains.com/plugin/10081-jetbrains-academy/docs/educator-start-guide.html?section=Kotlin) 学习如何创建一个简单的 Kotlin 课程, 包括一组编程任务和集成测试.

如果你希望使用 JetBrains Academy plugin 学习 Kotlin, 请参见 使用 JetBrains Academy plugin 学习 Kotlin (使用 JetBrains Academy plugin 学习 Kotlin).

参加 Kotlin EAP 项目

最终更新: 2024/09/10

你可以参加 Kotlin 早期预览(EAP) 项目, 试用 Kotlin 还未发布的最新功能.

在每个功能发布版 (*1.x*) 和增量发布版 (*1.x.y*)之前, 我们会发布少量的 Beta (*Beta*) 和 Release Candidate (*RC*) 版本.

如果你发现并报告 bug 到我们的问题追踪系统 YouTrack (https://kotl.in/issue), 我们非常感谢. 我们很可能会在最终发布版之前修正这些 bug, 因此为了解决你的问题, 不需要等到 Kotlin 的下个发布版.

参加早期预览(EAP) 项目并报告 bug, 你可以向 Kotlin 作出贡献, 帮助我们改进它, 为 不断增长的 Kotlin 社区 (https://kotlinlang.org/community/) 每个成员带来利益. 我们非常感谢你的帮助!

如果你有任何问题, 希望参与讨论, 欢迎加入 Kotlin Slack 的 #eap 频道 (https://app.slack.com/client/T09229ZC6/C0KLZSCHF). 在这个频道中, 你还可以收到关于新的 EAP 版本的通知.

在 IDEA 和 Android Studio 中安装 Kotlin EAP Plugin (安装 Kotlin EAP Plugin)

> 参与 EAP 项目, 表示你明确了解 EAP 版本并不可靠, 可能不会象期待的那样正常工作, 并且可能包含错误. 请注意, 对于 EAP 和某些版本的最终发布版之间兼容性, 我们并不提供任何保证.

如果你已经安装了 EAP 版, 并希望在之前创建的项目中使用, 请参见 如何针对 EAP 版配置你的构建 (针对 EAP 进行构建配置).

EAP 版能够如何帮助你更加高效的使用 Kotlin

- **为使用稳定发布版做好准备**.

  如果你在开发一个复杂的多模块项目, 参加 EAP 可以让你采用稳定发布版时的使用经验更加流畅. 越早更新到稳定发布版, 你就可以越快从它的性能改进和新的语言特性中获益.

  巨大而且复杂的项目的迁移工作可能需要耗费一些时间, 不仅因为项目的规模, 而且因为 Kotlin 开发组可能还没有考虑到某些特定的使用场景. 通过参加 EAP, 并持续的测试 Kotlin 的新版本, 对于你的特定的使用场景, 你可以快速向我们提供反馈意见. 因此可以帮助我们尽可能多的追查问

题, 并保证当稳定版正式发布时, 你能够安全的更新. 请参见 Slack 通过测试 Android, Kotlin, 以及 Gradle 的发布前预览版本带来的益处 (https://slack.engineering/shadow-jobs/).

- **保证使用最新版本的库.**

  如果你是库的开发者, 更新到新的 Kotlin 版本是极其重要的. 如果你使用旧的版本, 可能导致你的使用者无法在他们的项目中更新 Kotlin 版本. 如果使用 EAP 版本, 那么在稳定版发布时, 你可以几乎立即在你的库中支持最新的 Kotlin 版本, 因此可以让你的使用者更加满意, 也让你的库获得更多使用者.

- **分享使用经验.**

  如果你是一个 Kotlin 爱好者, 并且喜欢创作教育性的内容来向 Kotlin 生态系统做贡献, 那么试用 Kotlin EAP 中的新功能特性, 你就有机会第一个向 Kotlin 社区分享新的酷炫功能的使用经验.

EAP 版本

版本	重要功能
2.0.0-Beta5 发布日期: **2024/03/20** GitHub 上的 Release 页面 (https://github.com/JetBrains/kotlin/releases/tag/v2.0.0-Beta4)	Kotlin K2 编译器的稳定发布版. 包括 Gradle 构建工具的改进. Kotlin/Wasm 与 JavaScript 交互能力的改进. 更多详情, 请参见 changelog (https://github.com/JetBrains/kotlin/releases/tag/v2.0.0-Beta5) 或 Kotlin 2.0.0-Beta5 中的新功能 (Kotlin 2.0.0-Beta5 版中的新功能).

> 如果 Kotlin EAP plugin 无法找到最新的 EAP 版, 请检查你使用的是不是最新版的 IntelliJ IDEA (https://www.jetbrains.com/help/idea/update.html) 或 Android Studio (https://developer.android.com/studio/intro/update).

安装 Kotlin EAP Plugin

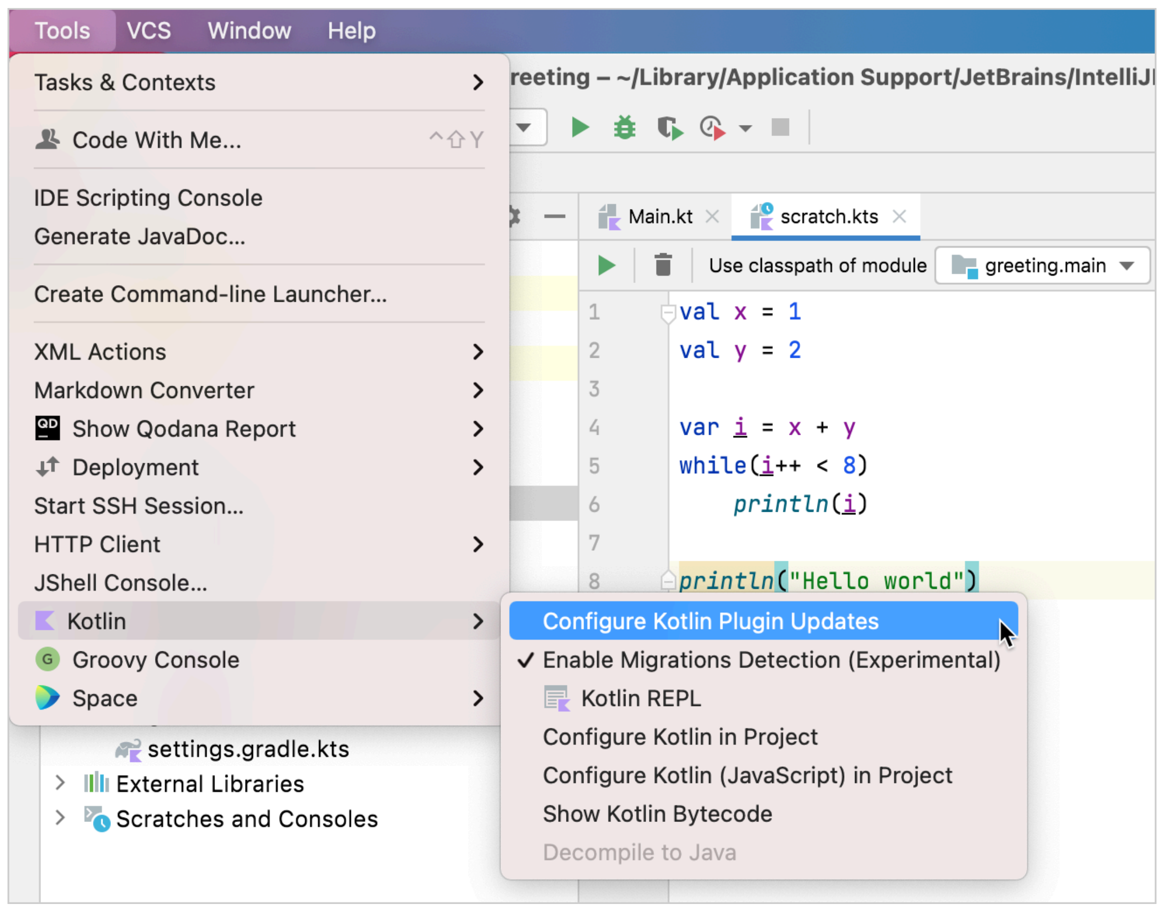
最新的 Kotlin EAP 版本: **2.0.0-Beta5**
查看 Kotlin EAP 版 ("EAP 版本" in "参加 Kotlin EAP 项目")

最终更新: 2024/09/10

> 从 IntelliJ IDEA 2023.3 和 Android Studio Iguana (2023.2.1) Canary 15 开始, 不再需要单独设置 Kotlin plugin. 你只需要在你的构建脚本中 修改 Kotlin 版本 (针对 EAP 进行构建配置).

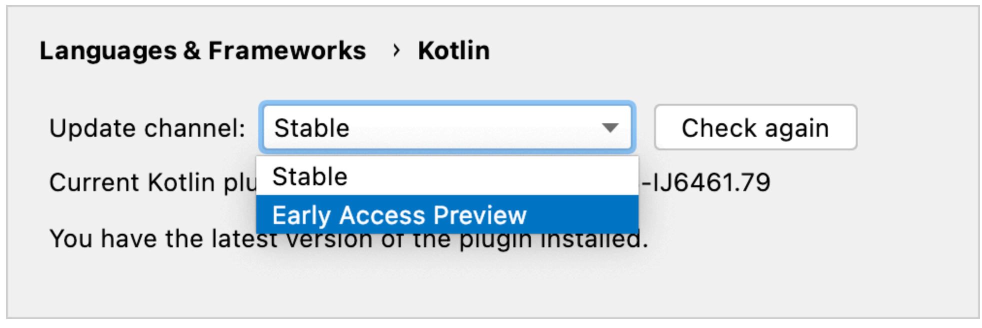
请按照以下步骤来安装 IntelliJ IDEA 和 Android Studio 的 Kotlin Plugin 预览版.

1. 选择 **Tools** | **Kotlin** | **Configure Kotlin Plugin Updates.**



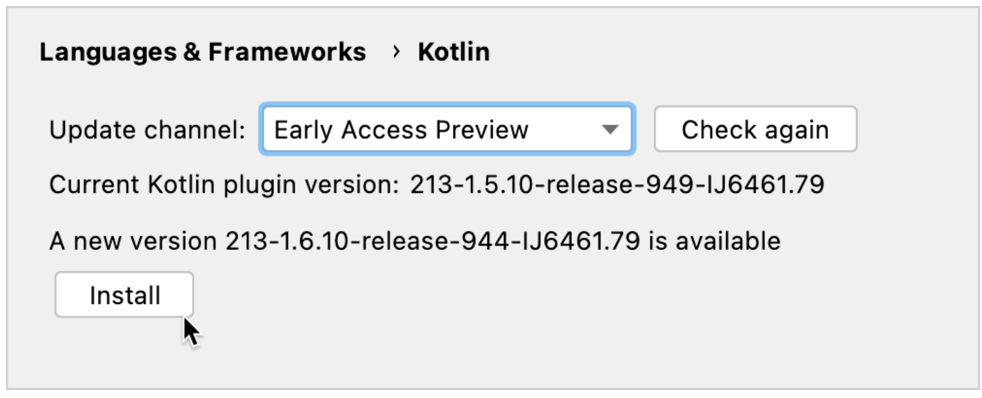
选择 Kotlin Plugin 更新

2. 在 **Update channel** 列表中, 选择 **Early Access Preview** 频道.



选择 EAP 更新频道

3. 点击 **Check again**. 会出现最新的 EAP 版本.



安装 EAP 版

> 如果 Kotlin EAP plugin 找不到最新的 EAP 版本, 请检查你使用的是不是 IntelliJ IDEA (https://www.jetbrains.com/help/idea/update.html) 或 Android Studio (https://developer.android.com/studio/intro/update) 的最新版本.

4. 点击 **Install**.

如果你希望在 EAP 版安装之前已创建的项目中使用 EAP 版, 你需要 为 EAP 版配置你的构建 (针对 EAP 进行构建配置).

如果你遇到任何问题

- 到 我们的问题追踪系统, YouTrack (https://kotl.in/issue) 报告问题.

* 到 Kotlin Slack 的 #eap 频道 (https://app.slack.com/client/T09229ZC6/C0KLZSCHF) 寻求帮助 (获得邀请 (https://surveys.jetbrains.com/s3/kotlin-slack-sign-up)).
* 回退到最新的稳定版: 在菜单 **Tools | Kotlin | Configure Kotlin Plugin Updates** 中, 选择 **Stable** 更新频道, 然后点击 **Install**.

针对 EAP 进行构建配置

最终更新: 2024/09/10

如果你使用 Kotlin 的 EAP 版创建新的项目, 你不需要进行任何额外的设置. Kotlin Plugin (安装 Kotlin EAP Plugin) 会为你配置好一切!

你只需要为既有的项目手动配置你的构建 — 在安装 EAP 版之前创建的那些项目.

要配置你的构建, 使它使用 Kotlin 的 EAP 版, 你需要:

- 指定 Kotlin 的 EAP 版. 详情请参见 可用的 EAP 版 ("EAP 版本" in "参加 Kotlin EAP 项目").
- 将依赖项版本修改为 EAP 版. Kotlin 的 EAP 版可能无法与前一个正式发布版的库共同工作.

下文解释如何在 Gradle 和 Maven 中配置你的构建:

- 在 Gradle 中配置
- 在 Maven 中配置

在 Gradle 中配置

本节介绍如何:

- 调整 Kotlin 版本
- 调整依赖项中的版本

调整 Kotlin 版本

在 build.gradle(.kts) 的 plugins 部分, 将 KOTLIN-EAP-VERSION 修改为实际的 EAP 版本, 比如 2.0.0-Beta5. 详情请参见 可用的 EAP 版本 ("EAP 版本" in "参加 Kotlin EAP 项目").

或者, 你也可以在 settings.gradle(.kts) 的 pluginManagement 部分指定 EAP 版本 – 详情请参见 Gradle 文档 (https://docs.gradle.org/current/userguide/plugins.html#sec:plugin_version_management).

下面是 Multiplatform 项目的示例.

Kotlin

```
plugins {
    java
    kotlin("multiplatform") version "KOTLIN-EAP-VERSION"
}

repositories {
    mavenCentral()
}
```

Groovy

```
plugins {
    id 'java'
    id 'org.jetbrains.kotlin.multiplatform' version 'KOTLIN-EAP-
VERSION'
}

repositories {
    mavenCentral()
}
```

调整依赖项中的版本

如果在你的项目中使用 kotlinx 库, 你的库版本可能与 Kotlin 的 EAP 版不兼容.

要解决这个问题, 你需要在依赖项中指定兼容的库版本. 兼容的库一览表, 请参见 EAP 版本详细列表 ("EAP 版本" in "参加 Kotlin EAP 项目").

> 大多数情况下, 我们只为某个发布版的第一个 EAP 版创建库, 这些库兼容这个发布版的后续 EAP 版. 如果在后面的 EAP 版中发生了不兼容的变更, 我们会发布新版本的库.

下面是示例.

对于 **kotlinx.coroutines** 库, 请添加版本号 – `1.7.3` – 这个版本兼容于 Kotlin 的 EAP 版 `2.0.0-Beta5`.

Kotlin

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.7.3")
}
```

Groovy

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.7.3"
}
```

在 Maven 中配置

在下面的 Maven 项目定义示例中, 请将 `KOTLIN-EAP-VERSION` 替换为实际的版本, 比如 `2.0.0-Beta5`. 详情请参见 可用的 EAP 版本 ("EAP 版本" in "参加 Kotlin EAP 项目").

```
<project ...>
    <properties>
        <kotlin.version>KOTLIN-EAP-VERSION</kotlin.version>
    </properties>

    <repositories>
        <repository>
           <id>mavenCentral</id>
           <url>https://repo1.maven.org/maven2/</url>
        </repository>
    </repositories>

    <pluginRepositories>
       <pluginRepository>
          <id>mavenCentral</id>
          <url>https://repo1.maven.org/maven2/</url>
       </pluginRepository>
    </pluginRepositories>
```

```xml
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-stdlib</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>${kotlin.version}</version>
                ...
            </plugin>
        </plugins>
    </build>
</project>
```

FAQ

最终更新: 2024/09/10

什么是 Kotlin?

Kotlin 是一种开源的, 静态类型的编程语言, 针对的目标平台是 JVM, Android, JavaScript, Wasm, 以及 Native 应用. Kotlin 由 JetBrains 公司 (https://www.jetbrains.com) 开发. Kotlin 项目开始于 2010 年, 并在很早的阶段开源. 第一次正式发布的 1.0 版是在 2016 年 2 月.

Kotlin 的当前版本是多少?

当前发布的版本是 1.9.23, 发布日期是 2024/03/07. 更多详情请参见 GitHub (https://github.com/jetbrains/kotlin).

Kotlin 是免费的吗?

是的. Kotlin 是免费的, 现在是免费的, 以后也会继续免费. 它使用 Apache 2.0 许可协议, 源代码托管在 GitHub (https://github.com/jetbrains/kotlin) 上.

Kotlin 是面向对象式语言, 还是函数式语言?

Kotlin 既有面向对象的部分, 也有函数式的部分. 你可以以面向对象的方式使用它, 也可以以函数式的方式使用它, 或者也可以混合使用. 由于它对高阶函数, 函数类型, lambda 表达式等等特性的一级支持, 如果你在进行函数式编程, 或者正在学习的话, Kotlin 是一个很好的选择.

Kotlin 能够向我提供哪些超出 Java 语言的功能?

Kotlin 更简洁. 粗略的估算显示, 代码行数可以减少大约 40%. Kotlin 在类型安全方面也更强, 比如, 它支持 非 null 类型, 可以减少应用程序的空指针异常. 其他特性包括, 智能类型转换, 高阶函数, 扩展函数, 以及带接受者的 lambda 表达式, 可以编写出表达能力更高的代码, 此外还有创建 DSL 的能力.

Kotlin 与 Java 语言兼容吗?

是的. Kotlin 100% 可以与 Java 语言交互, 而且重点保证你的既有代码可以与 Kotlin 正确交互. 你可以很容易地在 Java 中调用 Kotlin 代码 (在 Java 中调用 Kotlin 代码), 也可以反过来在 Kotlin 中调用 Java 代码 (在 Kotlin 中调用 Java 代码). 这个能力使得采用 Kotlin 变得更容易, 更低风险. 另外还有

IDE 中内置的 Java 到 Kotlin 源代码自动转换器 ("使用 J2K 将既有的 Java 文件转换为 Kotlin" in "教程 - 在同一个项目中混合使用 Java 和 Kotlin"), 可以大大简化既有代码的迁移工作.

我可以用 Kotlin 来做什么?

Kotlin 可以用来做任何类型的开发, 可以用在 Web 服务器端, Web 客户端, 以及 Android 环境. 通过 Kotlin/Native 功能(目前正在开发的), 未来还将支持其他平台, 比如嵌入式系统, macOS 以及 iOS. 目前已有开发者使用 Kotlin 开发移动应用程序, 服务端应用程序, JavaScript 或 JavaFX 的客户端应用程序, 以及数据科学, 这只是少部分例子.

我可以使用 Kotlin 进行 Android 开发吗?

是的. Kotlin 在 Android 中已受到一级支持. Android 环境中已经有几百中应用程序使用 Kotlin 开发, 比如 Basecamp, Pinterest, 等等. 详情请参照 Android 开发的相关资源 (使用 Kotlin 进行 Android 开发).

我可以使用 Kotlin 进行服务器端开发吗?

是的. Kotlin 与 JVM 100% 兼容, 因此你可以使用任何既有的框架, 比如 Spring Boot, vert.x 或 JSF. 此外, 还有使用 Kotlin 编写的框架, 比如 Ktor (https://github.com/kotlin/ktor). 详情请参见 服务端端开发的相关资源 (使用 Kotlin 进行服务器端开发).

我可以使用 Kotlin 进行 Web 开发吗?

是的. 除了用于 Web 后端开发之外, 你还可以使用 Kotlin/Wasm 来开发 Web 客户端. 参见 Kotlin/Wasm 入门 (使用 IntelliJ IDEA 开发 Kotlin/Wasm 入门).

我可以使用 Kotlin 进行桌面开发吗?

是的. 你可以使用任何 Java UI 框架, 比如 JavaFx, Swing, 或者其他框架. 此外, 还有专门的 Kotlin 框架, 比如 TornadoFX (https://github.com/edvin/tornadofx).

我可以使用 Kotlin 进行原生(Native)程序开发吗?

是的. Kotlin 项目包括了 Kotlin/Native. 它可以将 Kotlin 代码编译为原生代码, 运行时无需 VM. 你可以试用它, 用于流行的桌面和移动设备平台, 甚至还可以用于一部分 IoT 设备. 详情请参见 Kotlin/Native 文档 (使用 Kotlin/Native 进行原生(Native)程序开发).

有哪些 IDE 支持 Kotlin?

通过 JetBrains 开发的官方 Kotlin plugin, Kotlin 完全支持 IntelliJ IDEA (https://www.jetbrains.com/idea/download/), Android Studio (https://developer.android.com/kotlin/get-started), 以及 JetBrains Fleet (https://www.jetbrains.com/help/fleet/getting-started-with-kotlin-in-fleet.html).

其他 IDE 和源代码编辑器, 比如 Eclipse, Visual Studio Code, 和 Atom, 也有 Kotlin 社区支持的 plugin.

你也可以试用 Kotlin Playground (https://play.kotlinlang.org), 在你的浏览器中编写, 运行, 并共享 Kotlin 代码.

此外, 还有一个 命令行编译器 (Kotlin 命令行编译器), 可以编译并运行应用程序.

有哪些编译工具支持 Kotlin?

在 JVM 平台, 主流编译工具都支持 Kotlin, 包括 Gradle (Gradle), Maven (Maven), Ant (Ant), 以及 Kobalt (https://beust.com/kobalt/home/index.html). 此外还有一些针对 JavaScript 平台的编译工具.

Kotlin 编译输出的是什么?

在 JVM 平台, Kotlin 产生与 Java 兼容的字节码.

在 JavaScript 平台, Kotlin 产生 ES5.1 代码, 生成的代码兼容于 JavaScript 模块系统, 包括 AMD 和 CommonJS.

在 Native 平台, Kotlin 将(通过 LLVM)产生目标平台特有的代码.

Kotlin 支持 JVM 的哪些版本?

Kotlin 允许你选择运行时的 JVM 版本. 默认情况下, Kotlin/JVM 编译器产生与 Java 8 兼容的字节码. 如果你希望利用更高版本 Java 中的优化功能, 你可以明确指定编译目标的 Java 版本, 可选的版本是从 9 到 21. 注意, 这时编译产生的字节码在低版本的 Java 环境可能无法运行. 从 Kotlin 1.5 ("新的默认 JVM 编译目标: 1.8" in "Kotlin 1.5.0 版中的新功能") 开始, 编译器不再产生与低于 Java 8 的版本兼容的字节码.

Kotlin 难吗?

Kotlin 受到各种既有语言的启发, 比如 Java, C#, JavaScript, Scala 以及 Groovy. 我们努力确保 Kotlin 易于学习, 帮助开发者更容易转向 Kotlin, 可以在几天时间之内便能够读懂, 能够编写 Kotlin 代码. 学习 Kotlin 的惯用法, 使用某些高级特性可能会花费稍微长一点的时间, 但总的来说, Kotlin 不是一种复杂的语言. 详情请参见 我们的学习资料 (学习资料概述).

哪些公司在使用 Kotlin?

使用 Kotlin 的公司非常多, 难以全部列举, 但有些大公司已经通过 blog, 通过 GitHub 库, 或通过演讲, 公开宣布使用 Kotlin, 包括 Square (https://medium.com/square-corner-blog/square-open-source-loves-kotlin-c57c21710a17), Pinterest (https://www.youtube.com/watch?v=mDpnc45WwlI), Basecamp (https://m.signalvnoise.com/how-we-made-basecamp-3s-android-app-100-kotlin-35e4e1c0ef12) 以及 Corda (https://docs.corda.net/releases/release-M9.2/further-notes-on-kotlin.html).

Kotlin 的开发者是谁?

Kotlin 主要是由 JetBrains (https://www.jetbrains.com/) 公司的一个工程师团队(目前 100+ 人)开发的. 语言设计的领导者是 Michail Zarečenskij. 除了这个核心团队之外, 在 GitHub 上还有超过 250 人的外部贡献者.

在哪里可以得到 Kotlin 的更多信息?

最好从 我们的网站 (https://kotlinlang.org) 开始. 在这里, 你可以下载编译器 (Kotlin 命令行编译器), 在线试运行代码 (https://play.kotlinlang.org), 并访问各种资源.

是否有关于 Kotlin 的书籍?

关于 Kotlin 有很多书籍. 其中一些经过我们的审核, 并推荐大家从这些书籍开始学习. 这些书籍已经列在 Kotlin 书籍 (Kotlin 书籍) 页面. 其他更多书籍, 请参见由社区维护的书籍列表, 位于 kotlin.link (https://kotlin.link/) 网站.

是否有关于 Kotlin 的在线课程?

你可以通过 JetBrains Academy 的 Kotlin 核心教程 (https://hyperskill.org/tracks?category=4&utm_source=jbkotlin_hs&utm_medium=referral&utm_campaign=kotlinlang-docs&utm_content=button_1&utm_term=22.03.23) 来学习创建应用程序所需要的全部 Kotlin 基础知识.

你还可以学习这些课程:

- Kevin Jones 著: Pluralsight 课程: Kotlin 入门 (https://www.pluralsight.com/courses/kotlin-getting-started)
- Hadi Hariri 著: O'Reilly 课程: Kotlin 编程介绍

(https://www.oreilly.com/library/view/introduction-to-kotlin/9781491964125/)

- Peter Sommerhoff 著: Udemy 课程: 面向初学者的 10 个 Kotlin 教程
  (https://petersommerhoff.com/dev/kotlin/kotlin-beginner-tutorial/)

也可以通过我们的 YouTube 频道 (https://www.youtube.com/c/Kotlin) 查看其他教程和内容.

有 Kotlin 开发者社区吗?

是的! Kotlin 有一个很活跃的社区. Kotlin 开发者聚集在 Kotlin 论坛
(https://discuss.kotlinlang.org), StackOverflow
(https://stackoverflow.com/questions/tagged/kotlin), 以及更活跃的 Kotlin Slack
(https://slack.kotlinlang.org) (到 2020 年 4 月, 成员接近 30000 人).

有 Kotlin 开发者活动吗?

是的! 有很多专注于 Kotlin 的用户组, 以及聚会活动. 你可以 在这个网站
(https://kotlinlang.org/user-groups/user-group-list.html) 找到这类活动的列表. 此外, 还有
Kotlin 开发者社区在世界各地组织的 Kotlin 之夜
(https://kotlinlang.org/community/events.html) 活动.

有 Kotlin 开发者大会吗?

是的! Kotlin 开发者大会 (https://kotlinconf.com/) 由 JetBrains 公司每年举办一次, 这个大会聚集了来自全世界的开发者, 爱好者, 以及专家, 分享他们关于 Kotlin 的知识和经验.

除技术演讲和研讨之外, Kotlin 开发者大会还提供了的机会让大家建立关系网络, 组织社区活动, 以及社会活动, 参加者可以与其他 Kotlin 爱好者保持联系, 并交换思想. Kotlin 开发者大会是培养和促进 Kotlin 生态系统内协作, 以及社区建设的平台.

在世界各地的各种开发者大会中也会涉及到 Kotlin. 你可以 在这个网站
(https://kotlinlang.org/community/talks.html?time=upcoming) 找到即将举行的演讲列表.

Kotlin 是否有社交媒体帐号?

是的. 请订阅 Kotlin YouTube 频道 (https://www.youtube.com/c/Kotlin), 并追随 Kotlin 的
Twitter 帐号 (https://twitter.com/kotlin).

是否有关于 Kotlin 的其他在线资源?

在各种网站上有很多 在线资源 (https://kotlinlang.org/community/), 包括社区成员编写的 Kotlin Digests (https://kotlin.link), 一份 通讯 (http://kotlinweekly.net), 一个 博客 (https://talkingkotlin.com), 等等.

在哪里可以得到高分辨率的 Kotlin Logo?

可以在 这个地址 (https://resources.jetbrains.com/storage/products/kotlin/docs/kotlin_logos.zip) 下载 Logo. 使用 Logo 时请注意遵守使用规则, 具体请参见压缩包中的 `guidelines.pdf` 文件包含的简单规则, 以及 Kotlin 商标使用指南 (https://kotlinfoundation.org/guidelines/).

详情请参见 Kotlin 品牌资产 (Kotlin 品牌资产).

Kotlin 的演化

最终更新: 2024/09/10

务实的演化原则

> *语言设计就象用石头做雕像,*
>
> *但是这块石头还算比较柔软,*
>
> *付出一些努力之后, 我们以后还可以改造它.*
>
> *Kotlin 设计组*

Kotlin 被设计为一种为程序员服务的务实的工具. 当语言发生演化时, 我们通过以下原则来保证它的务实性:

- 随着时间的发展, 持续保证语言本身的现代化.
- 与使用者持续不断的反馈循环.
- 语言版本升级对使用者来说应该平滑, 便利.

我们来解释一下这些原则, 因为这是理解 Kotlin 演化的关键.

保证语言的现代化. 我们认识到, 任何系统随着时间的发展都会积累很多历史遗产. 过去曾经是非常前沿的技术, 到了今天可能无可挽救地变得非常过时. 我们必须让语言本身不断演进, 让它适应使用者的需求, 象使用者期望的那样, 永远保持最新状态. 这不仅包括增加新的功能, 也需要淘汰那些不再适合用于生产环境, 已经变成历史包袱的旧功能.

语言版本升级平滑便利. 不兼容的变更, 比如从语言中删除某个特性, 如果不经过适当的注意, 可能会导致语言版本升级时出现非常痛苦的迁移工作. *在这类变化发生之前*, 我们总是会提前发布公告, 将未来会被删除的功能标注为已废弃, 还会提供自动迁移工具. 当语言变化真正发生时, 我们希望大多数代码都已经更新过了, 因此迁移到新版本时不会发生问题.

来自使用者的反馈循环. 需要付出很大的努力, 才能完成语言中某个功能的废弃过程, 因此我们希望尽量减少将来出现的不兼容变化. 除了依靠我们自己尽力作出最好的判断之外, 我们相信, 对某个设计进行验证的做好办法就是, 在真正的软件开发过程中去试用它. 在将某个设计雕刻到石头上之前, 我们希望它经过实战考验. 因此我们会努力在语言的生产版本中发布一些新设计的早期版本, 但会将它设定为某种 *未稳定* 状态: 实验性, Alpha, 或 Beta (Kotlin 各部分组件的稳定性). 这些功能还没有

稳定下来, 随时都可能改变, 使用者需要明确地指明自己确定要使用这些未稳定功能, 以及自己愿意面对未来可能发生的迁移问题. 这些使用者会在使用过程中向我们提供宝贵的反馈信息, 我们收集这些反馈信息后, 会将他们的意见反映到后面的设计中, 并确定最终的功能设计.

不兼容的变更

如果由于从一个版本更新到另一个版本, 导致过去曾经正确工作的代码不再正确, 那么成为语言的*不兼容变更* (有时也称作 "破坏性变更"). 所谓"不再正确工作", 对它的精确定义有时可能会有一些不同意见, 但肯定包括一下情况:

- 过去能够正确编译并正常运行的代码, 现在出现了编译错误 (在编译时, 或者在链接时). 这种情况包括语言中删除了某些概念, 或者添加了新的限制.
- 过去能过正常运行的代码, 现在抛出了异常.

其他不那么明显的情况(或者叫 "灰色地带")包括, 对某些边界条件的处理发生了变化, 抛出和以前不同类型的异常, 某些只有通过反射才会出现的行为发生了变化, 没有公开文档或者没有明确定义的行为发生了变化, 改变了二进制库文件的名字, 等等. 这样的变更有时会非常重要, 并导致巨大的代码迁移工作, 有时变化只是非常细微的, 并没有什么影响.

不兼容的变更不包括以下情况:

- 增加新的警告.
- 启用一个新的语言概念, 或者放松了过去曾经存在的某个限制.
- 改变私有或内部的 API, 以及其他实现细节.

由于 "保证语言的现代化" 原则和 "语言版本升级平滑便利" 原则的存在, 因此不兼容的变更有时候是必须的, 但需要非常小心地引入这种变更. 我们的目标是, 让使用者能够提前察觉到即将发生的变更, 使他们有机会以比较容易地方式迁移代码.

理想情况下, 每一个不兼容的变更都会在编译时对有问题的代码给出警告(通常是 *功能已废弃警告*), 以这种方式通知用户, 并且还会发布自动迁移工具. 因此, 理想的代码迁移过程如下:

- 升级到版本 A (这里我们会提前宣布不兼容的变更)
  - 看到警告信息, 提示即将发生的变更
  - 在工具的帮助下迁移代码

- 升级到版本 B (不兼容的变更会在这里发生)
  - 完全不发生任何问题

实际引用中, 某些变更在编译期可能无法精确地检测出来, 因此无法提示警告信息, 但至少在版本 A 的发布公告中我们会通知使用者, 在版本 B 中会发生某个变更.

处理编译器 bug

编译器是个非常复杂的软件, 尽管开发者们付出了最大的努力, 但是编译器还是会有 bug. 有些 bug 会导致编译器本身崩溃, 或者报告不正确的编译错误, 或者编译产生明显不正确的代码, 这样的 bug 尽管很烦人, 而且很丢脸, 但其实是容易修复的, 因为修复这类 bug 不会造成不兼容的变更. 其他的 bug 可能导致编译器编译产生不争取的代码, 但不会崩溃: 比如, 忽略了源代码中的某些错误, 或者编译产生了不正确的指令. 对这类 bug 的修复技术上来说也属于不兼容的变更 (有些代码过去可以编译, 但修复编译器 bug 之后就不能编译了), 但是我们倾向于尽可能快地修复这些 bug, 以免不好的编程风格在使用者的源代码中扩散开. 我们的意见是, 这也符合 "语言版本升级平滑便利" 原则, 因为可以让更少的使用者遇到这些问题. 当然, 这只适用于正式发布版中出现的 bug 很快被发现的情况.

决策方式

Kotlin 的原始创建者, JetBrains 公司 (https://jetbrains.com), 在开发者社区的帮助下, 并通过与 Kotlin 基金会 (https://kotlinfoundation.org/) 的协调, 正在不断推动 Kotlin 的开发.

Kotlin 编程语言的所有变更都在首席语言设计师 (https://kotlinfoundation.org/structure/) (目前是 Michail Zarečenskij) 的监督之下. 所有与语言演进相关的问题, 首席设计师拥有最终决定权. 此外, 对已经完全稳定的组件的不兼容变更, 必须经过 Kotlin 基金会 (https://kotlinfoundation.org/structure/) 任命的 语言委员会 (https://kotlinfoundation.org/structure/) 的批准. (语言委员会目前由 Jeffrey van Gogh, Werner Dietl, 和 Michail Zarečenskij 组成).

语言委员会最终决定作出哪些不兼容的变更, 应该采取哪些步骤让使用者平滑地升级. 在作出这些决策时, 语言委员会依靠一组指导原则, 详情请参见 这里 (https://kotlinfoundation.org/language-committee-guidelines/).

功能性发布版(Feature Release)与增量发布版(Incremental Release)

稳定发布版, 比如版本号 1.2, 1.3, 等等. 通常是一次 *功能发布版*, 带来大的语言变更. 通常, 在功能发布版之间我们会发布一些 *增量发布版*, 比如版本号 1.2.20, 1.2.30, 等等.

增量发布版会带来工具更新(通常包含新功能), 性能改进, 以及 bug 修正. 我们会努力让这些版本之间相互兼容, 因此编译器的变更通常只是代码优化, 警告信息的增加/删除. 当然, 未稳定功能可能会在任何时候发生增加, 删除, 或变更.

功能性发布版通常会增加新的功能, 也可能会删除或变更以前废弃掉的功能. 某个功能从未稳定状态升级到稳定状态, 也会发生在功能性发布版中.

早期预览版(Early Access Program (EAP))

在发布稳定版本之前, 我们通常会发布许多个预览版, 称为早期预览版 (Early Access Program (EAP)), 我们使用这种方式来更加快速地迭代我们的版本, 并从开发者社区收集使用者的反馈信息. 功能性发布版的 EAP 输出的二进制文件, 通常会被将来的稳定版编译器拒绝, 以保证预览版输出的二进制文件中可能存在的 bug 不会长期存在. 最终的发布候选版(Final Release Candidate)通常不会存在这个限制.

未稳定功能

根据我们上面介绍过的 "来自使用者的反馈循环" 原则, 我们会在语言的预览版和发布版中快速迭代和改进我们的设计, 这些版本中某些功能可能会处于某种 *未稳定* 状态, 并且 *预期会被改变*. 这些功能可能会在任何时候增加, 修改, 或者删除, 而且不会有任何警告. 我们会尽力确保使用者不会意外地使用到未稳定功能. 这些功能通常会需要某种明确的设置才能启用, 要么在源代码中, 要么在项目配置中.

经过一系列的迭代改进后, 未稳定功能通常会升级到稳定状态.

各部分组件的稳定性状态

Kotlin 包含很多组件(Kotlin/JVM, JS, Native, 各种库, 等等), 关于各部分组件的稳定性状态, 请参见参考文档 (Kotlin 各部分组件的稳定性).

库

离开了它的生态系统, 一个编程语言就毫无用处了, 因此我们付出了很大的努力, 来确保 Kotlin 库的平滑演进.

理想情况下, 库的新版本应该可以直接替代旧版本. 也就是说, 对一个二进制依赖项的版本升级, 应该不造成任何破坏, 即使应用程序没有重新编译 (这是通过动态链接来实现的).

然而, 为了实现这个目标, 编译器就必须在各自独立的不同编译之间, 保证某种程度的二进制接口 (Application Binary Interface, ABI) 稳定性. 这就是为什么 每一次语言变更都需要通过二进制兼容性的观点进行审查.

另一方面, 很大程度上我们依赖于库的作者来仔细判断那些变更是安全的. 因此, 库作者需要正确理解源代码的变更会如何影响二进制的兼容性, 并遵循某种好的实践原则, 来确保他们的库在 API 和

ABI 两方面的稳定性. 从库的演进的角度考虑语言的变更, 我们设想了以下原则:

- 库的代码应该明确指定 public/protected 函数和属性的值类型, 因此, 对于 public API 不应该通过类型推断来决定值类型. 类型推断的细微变化可能会导致返回值类型的变化, 而且这种变化是很难察觉的, 因此会发生二进制兼容性问题.
- 由同一个库提供的重载(overload)的函数和属性, 本质上应该做完全相同的工作. 类型推断的变更可能导致在函数调用处得到更加精确的静态类型, 因此会导致对重载函数的调用解析为不同的结果.

库的作者可以使用 @Deprecated 和 @RequiresOptIn (明确要求使用者同意的功能(Opt-in Requirement)) 注解来控制他们的 API 接口的演进. 注意, 即使是已经从 API 中删除的声明, 也可以使用 @Deprecated(level=HIDDEN) 注解来保护二进制兼容性.

而且, 按照通常的规约, 命名为 "internal" 的包不应该看作 public API. 命名为 "experimental" 的包内所有的 API 都应该被看作未稳定功能, 随时可以发生变化.

对稳定的平台的 Kotlin 标准库 (kotlin-stdlib), 我们按照上述原则进行维护. 对标准库的 API 的变更, 需要经过与语言变更相同的流程.

编译器参数

编译器接受的命令行参数也是一种 public API, 因此对它们也适用同样的原则. 编译器接受的参数 (不带 "-X" 前缀或 "-XX" 前缀的那些) 只能在功能发布版中增加, 而且在删除之前, 需要先标记为废弃. "-X" 和 "-XX" 参数是实验性的, 随时可以添加, 删除.

兼容性工具

由于遗留的旧功能被删除, bug 被修复, 因此源代码的语言变更时, 如果旧的源代码没有适当地迁移, 可能会无法正确编译. 通常的废弃流程使得使用者可以有一个平滑的代码迁移期间, 即使这个期间结束后, 语言的不兼容性变更已经随稳定版发布了, 我们仍然有办法可以编译未迁移的旧代码.

兼容性标记

我们提供了 `-language-version X.Y` 和 `-api-version X.Y` 标记, 用来让 Kotlin 的新版本模拟旧版本的行为, 以便维持兼容性. 为了给你留下更多的代码迁移时间, 除最新的稳定版之外, 我们还 支持 (兼容模式) 使用语言和 API 的前 3 个旧版本.

活跃维护中的代码库可以尽快升级到 bug 修复后的版本, 而不必等待整个升级周期完成. 目前, 这样的项目可以启用 `-progressive` 选项, 这样即使在增量发布的版本中, 也可以让这些 bug 修复有效.

所有这些标记都可以在命令行中使用, 也可以在 Gradle (Kotlin Gradle plugin 中的编译器选项) 和 Maven ("指定编译器选项" in "Maven") 中使用.

二进制格式的演化

即使在最糟糕的情况下, 源代码中的问题也可以手工修复, 但二进制文件的迁移就要困难得多了, 因此, 对二进制文件来说, 保证向后兼容是非常重要的. 二进制文件的不兼容变更可能导致版本升级非常痛苦, 因此, 与对语言的变更相比, 进行二进制文件的不兼容变更需要更加慎重.

对于完全稳定版本的编译器, 默认的二进制兼容性原则如下:

- 所有的二进制文件都是向后兼容的, 也就是说, 新版本的编译器可以读取旧版本的二进制文件 (比如, 1.3 版可以正确理解 1.0 到 1.2 版的二进制文件),
- 旧版本的编译器会拒绝那些依赖新功能的二进制文件(比如, 1.0 版的编译器会拒绝那些使用了协程的二进制文件).
- 更进一步(但我们不能保证一定如此), 大多数二进制文件可以向前兼容下一个功能发布版, 但不兼容再下一个版本(如果没有使用新的功能, 比如, 1.3 版可以理解 1.4 版产生的大多数二进制文件, 但不能理解 1.5 版产生的二进制文件).

这个原则是为了 "语言版本升级平滑便利" 而设计的, 因为即使项目本身在使用稍微旧一点的编译器, 它仍然可以升级它的依赖项目版本.

请注意, 并不是所有目标平台的稳定性都达到了这个程度(但 Kotlin/JVM 已经达到了).

Kotlin 各部分组件的稳定性

最终更新: 2024/09/10

Kotlin 语言和它的工具集分成很多组件, 比如针对 JVM, JS 和 Native 平台的编译器, 标准库, 大量的相关工具, 等等. 这些组件很多已经正式发布为 **稳定(Stable)** 版本, 也就是说它们会以向后兼容的方式演化, 遵循 *版本升级平滑便利* 以及 *保证语言的现代化* 的 原则 (Kotlin 的演化). 这些稳定组件包括, 比如, Kotlin 针对 JVM 的编译器, 标准库, 协程.

遵循 *反馈循环(Feedback Loop)* 原则, 出于试用的目的, 我们会向开发者社区提前发布很多功能, 因此很多组件还没有发布为 **稳定(Stable)** 版本. 其中一些功能还处于非常早期的阶段, 另一些已经比较成熟了. 根据各个组件的演化速度不同, 以及使用它们会给使用者带来的风险不同, 我们将这些功能标记为 **实验性(Experimental)**, **Alpha** 或 **Beta** 几种状态.

稳定性级别

下面简单介绍这些稳定性级别的含义:

实验性(Experimental) 代表 "请只在玩具项目中使用这些功能":

- 我们只是在实验某些想法, 并且希望某些使用者试用, 并提供意见反馈. 如果这些想法不成功, 我们随时可能抛弃它.

Alpha 代表 "使用时风险自负, 将来升级时可能会出现问题":

- 我们打算将这些想法变成产品, 但它还没有达到最终状态.

Beta 代表 "可以使用这些功能, 我们会尽力减少升级时的问题":

- 功能已经基本完成, 现在使用者的意见反馈非常重要.
- 但是, 它还没有 100% 完成, 因此还可能发生变化 (包括根据你的意见反馈产生的变化).
- 为了确保升级顺利, 请注意废弃声明.

我们将 *实验性(Experimental)*, *Alpha* 和 *Beta* 统称为 **未稳定(pre-stable)** 级别.

稳定(Stable) 代表 "即使是在最保守的场景也可以使用这些功能":

* 功能已开发完毕. 我们会继续改进它, 遵循我们严格的 向后兼容(backward compatibility) 规则 (https://kotlinfoundation.org/language-committee-guidelines/).

请注意, 稳定性级别并不代表组件会在什么时间发布为稳定版本. 同样, 也不代表组件在正式发布之前会发生多大的变化. 稳定性级别只代表组件会以多快的速度发生变化, 以及将来的版本升级问题会给使用者带来多大的风险.

Kotlin 组件的 GitHub 徽章

GitHub 上的 Kotlin 组织 (https://github.com/Kotlin) 存放着很多 Kotlin 相关项目. 有些项目我们在全职投入开发, 其他则只是业余项目.

每个 Kotlin 项目都有 2 个 GitHub 徽章描述它的稳定性和支持状态:

* **稳定性** 状态. 表示每个项目的演化速度, 以及用户使用它时的风险程度. 稳定性状态与 Kotlin 语言特性和各组件的稳定性级别 完全相符:
  * experimental 表示 **Experimental** 状态
  * alpha 表示 **Alpha** 状态
  * beta 表示 **Beta** 状态
  * stable 表示 **Stable** 状态
* **支持** 状态. 表示我们对维护这个项目以及帮助使用者解决相关问题的保证程度. 对于 JetBrains 的所有产品, 支持级别是一致的. 详情请参见 JetBrains 文档 (https://confluence.jetbrains.com/display/ALL/JetBrains+on+GitHub).

子组件的稳定性

一个稳定的组件也可以包含实验性的子组件, 比如:

* 稳定的编译器可以包含实验性的功能特性;
* 稳定的 API 可以包含实验性的类或函数;
* 稳定的命令行工具可能包含实验性的参数选项.

我们确保会对这些未稳定的子组件提供正确的文档. 我们也会尽可能警告使用者, 并要求使用者明确同意, 以避免无意中使用到这些未稳定发布的功能.

Kotlin 各部分组件当前的稳定性

组件	状态	进入这个状态的版本	备注
Kotlin/JVM	Stable	1.0	
Kotlin K2 (JVM, Native, Wasm, JS)	Beta	1.9.20	
kotlin 标准库 (JVM)	Stable	1.0	
协程	Stable	1.3	
kotlin 反射 (JVM)	Beta	1.0	
Kotlin/JS (基于传统后端)	Stable	1.3	从 1.8.0 开始已废弃, 参见 IR 迁移指南 (将 Kotlin/JS 项目迁移到 IR 编译器)
Kotlin/JVM (基于 IR)	Stable	1.5	
Kotlin/JS (基于 IR)	Stable	1.8	
Kotlin/Native 运行库	Stable	1.9.20	
Kotlin/Native 内存管理器	Stable	1.9.20	
klib 二进制文件	Stable	1.9.20	
Kotlin Multiplatform	Stable	1.9.20	
Kotlin/Native 与 C 代码和 Objective C 代码的交互	Beta	1.3	
CocoaPods 集成	Stable	1.9.20	

Android Studio 的 Kotlin Multiplatform Mobile plugin	Beta	0.8.0	版本与语言本身的版本不同 (Kotlin Multiplatform Mobile Plugin 的发布版本)
期待(expected)与实际(actual) 函数和属性	Stable	1.9.20	
期待(expected)与实际(actual) 类	Beta	1.7.20	
KDoc 语法	Stable	1.0	
Dokka	Beta	1.6	
脚本的语法和语义	Alpha	1.2	
脚本的内嵌和扩展 API	Beta	1.5	
脚本的 IDE 支持	Beta		从 IntelliJ IDEA 2023.1 之后版本开始可用
CLI 脚本	Alpha	1.2	
编译器插件 API	Experimental	1.0	
序列化编译器插件	Stable	1.4	
序列化核心库	Stable	1.0.0	版本与语言本身的版本不同
内联类(Inline class)	Stable	1.5	
无符号数运算	Stable	1.5	

标准库中的契约(Contract)	Stable	1.3	
用户定义的契约(Contract)	Experimental	1.3	
所有其他实验性组件, 默认状态	Experimental	N/A	

本章节在 1.4 以前的版本参见这个页面 (Kotlin 各部分组件的稳定性 (1.4 版以前)).

Kotlin 各部分组件的稳定性 (1.4 版以前)

最终更新: 2024/09/10

根据组件演进速度的不同, 可能存在几种不同的稳定性模式:

- **快速变化 (Moving fast, MF)**: 即使在 增量发布 ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化") 之间也不保证任何兼容性, 可能在没有警告的情况下增加, 删除, 或改变任何功能.
- **包括新功能的增量发布 (Additions in Incremental Releases, AIR)**: 在增量发布时可能增加新的功能, 尽量避免删除或改变功能, 如果确实需要, 应该在之前的增量发布时提前公告.
- **稳定的增量发布 (Stable Incremental Releases, SIR)**: 增量发布保证完全兼容, 只进行代码优化和 bug 修正. 任何其他变化都应该通过 功能发布 ("功能性发布版(Feature Release)与增量发布版(Incremental Release)" in "Kotlin 的演化") 来进行.

- **完全稳定 (Fully Stable, FS)**: 增量发布保证完全兼容, 只进行代码优化和 bug 修正. 功能发布保证向后兼容.

对于同一个组件, 源代码和二进制发布版可以有不同的稳定模式, 例如, 源代码可以比二进制版更早到达完全稳定状态, 或者反过来.

只对那些达到了完全稳定 (Fully Stable, FS) 的组件, 才完全适用 Kotlin 演进政策 (Kotlin 的演化) 的条款. 在此之后的一切导致不兼容的变更, 都必须经过 Kotlin 语言委员会的审批.

** 组件 **	** 进入该状态的版本 **	** 源代码稳定性 **	** 二进制发布版稳定性 **
Kotlin/JVM	1.0	FS	FS
kotlin 标准库 (JVM)	1.0	FS	FS
KDoc 语法	1.0	FS	N/A
协程	1.3	FS	FS
kotlin 反射 (JVM)	1.0	SIR	SIR
Kotlin/JS	1.1	AIR	MF
Kotlin/Native	1.3	AIR	MF
Kotlin 脚本 (*.kts)	1.2	AIR	MF
dokka	0.1	MF	N/A
Kotlin 脚本 API	1.2	MF	MF
编译器插件 API	1.0	MF	MF
序列化	1.3	MF	MF
跨平台项目	1.2	MF	MF
内联类	1.3	MF	MF
无符号数运算	1.3	MF	MF

所有其他实验性功能的默认稳定性	N/A	**MF**	**MF**

Kotlin 1.9 兼容性指南

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/compatibility-guide-19.html)

Kotlin 1.8 兼容性指南

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/compatibility-guide-18.html)

Kotlin 1.7.20 兼容性指南

最终更新: 2024/09/10

保证语言的现代化 (*Kotlin 的演化*) and *语言版本升级平滑便利* (*Kotlin 的演化*) 是 Kotlin 语言设计时的基本原则之一. 第一条原则认为, 阻碍语言演进的那些元素应该删除, 后一条原则则认为, 这些删除必须事先与使用者良好沟通, 以便让源代码的迁移尽量平滑.

不兼容的变更通常只出现在功能发布版中, 但这一次, 我们不得不在一个增量发布版中引入了 2 个这样的变更, 以便尽早解决由 Kotlin 1.7 的变更造成的一些问题.

本文档概述这些问题, 提供关于 Kotlin 1.7.0 和 1.7.10 向 Kotlin 1.7.20 迁移的参考.

基本术语

在本文档中, 我们介绍几种类型的兼容性:

- *源代码级兼容性*: 源代码级别的不兼容会导致过去能够正确编译(没有错误和警告)的代码变得不再能够编译
- *二进制级兼容性*: 如果交换两个二进制库文件, 不会导致程序的装载错误, 或链接错误, 那么我们称这两个文件为二进制兼容
- *行为级兼容性*: 如果在某个变更发生之前和之后, 程序表现出不同的行为, 那么这个变更称为行为不兼容

请记住, 这些兼容性定义只针对纯 Kotlin 程序. 从其他语言(比如, Java)的观点来看 Kotlin 代码的兼容性如何, 本文档不予讨论.

语言

回滚了对约束处理的修正

> **Issue**: KT-53813 (https://youtrack.jetbrains.com/issue/KT-53813)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 过去曾经尝试修正 1.7.0 版在类型推断约束处理中出现的问题, 本次发布实现了 KT-52668 (https://youtrack.jetbrains.com/issue/KT-52668) 描述的变更, 回滚了这个修

正. 在 1.7.10 中进行了这个修正, 但导致了新的问题.

废弃周期:

- 1.7.20: 回滚到 1.7.0 版的行为

禁止某些构建器推断情况, 以避免与多个 Lambda 表达式和解析之间的有问题的互动

Issue: KT-53797 (https://youtrack.jetbrains.com/issue/KT-53797)

组件: 核心语言

不兼容性类型: 源代码级

概述: Kotlin 1.7 引入了一个功能特性, 称为无限制的构建器推断, 使得即使传递给参数的 Lambda 表达式没有标注 `@BuilderInference` 注解, 也可以利用构建器推断功能. 但是, 如果在函数调用中出现多个这样的 Lambda 表达式, 可能导致几种问题.

如果多个 Lambda 函数对应的参数没有标注 `@BuilderInference` 注解, 而且在 Lambda 表达式内要求使用构建器推断来完成类型推断, Kotlin 1.7.20 会对这样的情况报告编译错误.

废弃周期:

- 1.7.20: 对这样的Lambda 函数报告编译错误, 可以使用 `-XXLanguage:+NoBuilderInferenceWithoutAnnotationRestriction` 来临时退回到 1.7.20 以前的行为.

Kotlin 1.7 兼容性指南

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/compatibility-guide-17.html)

Kotlin 1.6 兼容性指南

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/compatibility-guide-16.html)

Kotlin 1.5 兼容性指南

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/compatibility-guide-15.html)

Kotlin 1.4 兼容性指南

最终更新: 2024/09/10

保证语言的现代化 (*[Kotlin 的演化]*) 以及 *语言版本升级平滑便利* (*[Kotlin 的演化]*) 是 Kotlin 语言设计时的基本原则之一. 第一条原则认为, 阻碍语言演进的那些元素应该删除, 后一条原则则认为, 这些删除必须事先与使用者良好沟通, 以便让源代码的迁移尽量平滑.

尽管语言的大多数变化都通过其他途径进行了通知, 比如每次更新时的变更日志, 以及编译器的警告信息, 但我们还是在本文档中对这些变化进行一个总结, 提供一个 Kotlin 1.3 从迁移到 Kotlin 1.4 时的完整的参考列表.

基本术语

在本文档中, 我们介绍几种类型的兼容性:

- *源代码级兼容性*: 源代码级别的不兼容会导致过去能够正确编译(没有错误和警告)的代码变得不再能够编译
- *二进制级兼容性*: 如果交换两个二进制库文件, 不会导致程序的装载错误, 或链接错误, 那么我们称这两个文件为二进制兼容
- *行为级兼容性*: 如果在某个变更发生之前和之后, 程序表现出不同的行为, 那么这个变更称为行为不兼容

请记住, 这些兼容性定义只针对纯 Kotlin 程序. 从其他语言(比如, Java)的观点来看 Kotlin 代码的兼容性如何, 本文档不予讨论.

语言与标准库

ConcurrentHashMap 的 in 中缀操作符的不正常行为

> **Issue**: KT-18053 (https://youtrack.jetbrains.com/issue/KT-18053)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: Kotlin 1.4 将会禁止使用 Java 编写的 `java.util.Map` 的实现中的 `contains` 操作符
>
> **废弃周期**:

- < 1.4: 对问题的操作符, 在调用处产生编译警告
- >= 1.4: 将这个警告提升为错误, 可以暂时使用 `-XXLanguage:-ProhibitConcurrentHashMapContains` 回退到 1.4 以前的行为

禁止访问 public inline 成员之内的 protected 成员

Issue: KT-21178 (https://youtrack.jetbrains.com/issue/KT-21178)

组件: 核心语言

不兼容性类型: 源代码级

概述: Kotlin 1.4 将会禁止通过 public inline 成员来访问 protected 成员.

废弃周期:

- < 1.4: 对有问题的访问, 在调用处产生编译警告
- 1.4: 将这个警告提升为错误, 可以暂时使用 `-XXLanguage:-ProhibitProtectedCallFromInline` 回退到 1.4 以前的行为

带隐含接受者的调用的契约(Contract)

Issue: KT-28672 (https://youtrack.jetbrains.com/issue/KT-28672)

组件: 核心语言

不兼容性类型: 行为级

概述: 在 1.4 中, 对于带隐含接受者的调用, 可以根据契约(Contract)信息进行智能类型转换

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-ContractsOnCallsWithImplicitReceiver` 回退到 1.4 以前的行为

浮点数比较的行为不一致

> **Issues**: KT-22723 (https://youtrack.jetbrains.com/issue/KT-22723)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 行为级
>
> **概述**: 从 Kotlin 1.4 开始, Kotlin 编译器将会使用 IEEE 754 标准来比较浮点数
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-ProperIeee754Comparisons` 回退到 1.4 以前的行为

在泛型 Lambda 表达式中, 最后一条表达式没有智能类型转换

> **Issue**: KT-15020 (https://youtrack.jetbrains.com/issue/KT-15020)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 行为级
>
> **概述**: 从 1.4 开始, Lambda 表达式中对最后一条表达式的智能类型转换将会正确执行
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

不再根据 Lambda 表达式参数的顺序将类型强制解释为 Unit

> **Issue**: KT-36045 (https://youtrack.jetbrains.com/issue/KT-36045)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级

概述: 从 Kotlin 1.4 开始, Lambda 表达式参数将会分别独立解析, 不再隐含的强制解释为 `Unit`

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

raw 和 integer 字面类型的共通超类型错误, 导致代码错误

Issue: KT-35681 (https://youtrack.jetbrains.com/issue/KT-35681)

Components: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, raw `Comparable` 类型和 integer 字面类型的共通超类型将会更加明确

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

类型安全性问题: 几个相等类型的变量被初始化为不同的类型

Issue: KT-35679 (https://youtrack.jetbrains.com/issue/KT-35679)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, Kotlin 编译器将会禁止将相等类型的变量初始化为不同的类型

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)

- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

类型安全性问题: 对于类型交集, 子类型不正确

Issues: KT-22474 (https://youtrack.jetbrains.com/issue/KT-22474)

组件: 核心语言

不兼容性类型: 源代码级

概述: 在 Kotlin 1.4 中, 将会改进类型交集的子类型, 使其动作更正确

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

在 Lambda 表达式之内的空 when 表达式没有类型不匹配

Issue: KT-17995 (https://youtrack.jetbrains.com/issue/KT-17995)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 对于空 `when` 表达式, 如果用作 Lambda 表达式之内的最后一条表达式, 将会出现类型不匹配

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

如果 Lambda 表达式可能的返回值之一是使用 integer 字面类型的快速返回, 推断的 Lambda 表达式返回类型为 Any

> **Issue**: KT-20226 (https://youtrack.jetbrains.com/issue/KT-20226)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 从 Kotlin 1.4 开始, 如果 Lambda 表达式存在快速返回, 返回类型将会更加正确的推断为 integer
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

正确的捕捉带递归类型的星号投射(star projection)

> **Issue**: KT-33012 (https://youtrack.jetbrains.com/issue/KT-33012)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 从 Kotlin 1.4 开始, 可以使用更多类型选项, 因为递归类型的捕捉将会更加正确
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

不完整类型(non-proper type)与灵活类型(flexible type)的共通超类型计算导致错误的结果

> **Issue**: KT-37054 (https://youtrack.jetbrains.com/issue/KT-37054)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 行为级

概述: 从 Kotlin 1.4 开始, 灵活类型(flexible type)的共通超类型将会更加正确, 防止运行时错误

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

类型安全性问题: 可 null 的类型参数未能正确转换为捕获的类型

Issue: KT-35487 (https://youtrack.jetbrains.com/issue/KT-35487)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 捕获类型与可 null 类型的子类型将会更加正确, 防止运行时错误

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

对协变(covariant)类型, 在未检测的类型转换之后, 保留交叉类型(intersection type)

Issue: KT-37280 (https://youtrack.jetbrains.com/issue/KT-37280)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 对协变(covariant)类型的未检测的类型转换, 会产生用于智能类型转换的交叉类型(intersection type), 而不是未检测的类型转换的类型.

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)

- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

使用 this 表达式导致构造器推断中缺少类型变量

> **Issue**: KT-32126 (https://youtrack.jetbrains.com/issue/KT-32126)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 从 Kotlin 1.4 开始, 如果不存在其他适当的约束, 在 `sequence {}` 之类的构造器函数内使用 `this` 会被禁止
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

对带有可 null 类型参数的反向类型变异(contravariant)类型的 overload 解析结果错误

> **Issue**: KT-31670 (https://youtrack.jetbrains.com/issue/KT-31670)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 从 Kotlin 1.4 开始, 如果一个函数的 2 个 overload, 接受反向类型变异(contravariant)类型参数, 区别只有类型可否为 null (比如 `In<T>` 和 `In<T?>`), 这时将认为可 null 类型是更明确的 overload 解析结果.
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

带有非嵌套的递归约束的构造器推断

Issue: KT-34975 (https://youtrack.jetbrains.com/issue/KT-34975)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, `sequence {}` 之类的构造器函数, 如果其类型依赖于一个在传入的Lambda 表达式之内的递归约束, 会导致编译器错误.

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

类型变量的固定在及早计算(eager)模式下会导致矛盾的约束系统

Issue: KT-25175 (https://youtrack.jetbrains.com/issue/KT-25175)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 某些场景的类型推断不再使用及早计算(eager)模式, 以便寻找没有矛盾的约束系统.

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NewInference` 回退到 1.4 以前的行为. 注意, 这个标记还会同时关闭其他几个新语言特性.

对 open 函数禁止使用 tailrec 标识符

Issue: KT-18541 (https://youtrack.jetbrains.com/issue/KT-18541)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 函数不能同时存在 `open` 和 `tailrec` 标识符.

废弃周期:

- < 1.4: 对同时存在 `open` 和 `tailrec` 标识符的函数报告警告 (渐进模式(progressive mode) 下会报告错误).
- >= 1.4: 将这个警告提升为错误.

同伴对象的 INSTANCE 域可见度超过同伴对象的类本身

Issue: KT-11567 (https://youtrack.jetbrains.com/issue/KT-11567)

组件: Kotlin/JVM

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 如果同伴对象可见度为 private, 那么它的域 `INSTANCE` 可见度也将是 private

废弃周期:

- < 1.4: 编译器生成对象的 `INSTANCE` 带有废弃标记
- >= 1.4: 同伴对象的 `INSTANCE` 域带有正确的可见度

插入在 return 之前的外层 finally 代码段, 没有从不带 finally 的内层 try 代码段的 catch 分支中排除

Issue: KT-31923 (https://youtrack.jetbrains.com/issue/KT-31923)

组件: Kotlin/JVM

不兼容性类型: 行为级

概述: 从 Kotlin 1.4 开始, 对嵌套的 `try/catch` 代码段, catch 分支将被正确计算

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)

- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-ProperFinally` 回退到 1.4 以前的行为

对协变(covariant)和泛型专用的(generic-specialized)覆盖, 在返回类型位置的内联类(inline class)会使用装箱(box)版本

Issues: KT-30419 (https://youtrack.jetbrains.com/issue/KT-30419)

组件: Kotlin/JVM

不兼容性类型: 行为级

概述: 从 Kotlin 1.4 开始, 使用协变(covariant)和泛型专用的(generic-specialized)覆盖的函数, 将会返回内联类(inline class)的装箱(box)的值

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化

委托到 Kotlin 接口时, 在 JVM 字节码中不声明受控异常(Checked Exception)

Issue: KT-35834 (https://youtrack.jetbrains.com/issue/KT-35834)

组件: Kotlin/JVM

不兼容性类型: 源代码级

概述: Kotlin 1.4 中, 当接口委托给 Kotlin 接口时, 将不会生成受控异常(Checked Exception)

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-DoNotGenerateThrowsForDelegatedKotlinMembers` 回退到 1.4 以前的行为

对于带有单个不定数量参数(vararg parameter)的方法, 签名多态(signature-polymorphic)调用的行为有变化, 以避免将参数包装入另一个数组

> **Issue**: KT-35469 (https://youtrack.jetbrains.com/issue/KT-35469)
>
> **组件**: Kotlin/JVM
>
> **不兼容性类型**: 源代码级
>
> **概述**: Kotlin 1.4 中, 对签名多态(signature-polymorphic)的方法调用, 不会将参数包装入另一个数组
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化

当 KClass 用作泛型参数时, 注解中的泛型签名错误

> **Issue**: KT-35207 (https://youtrack.jetbrains.com/issue/KT-35207)
>
> **组件**: Kotlin/JVM
>
> **不兼容性类型**: 源代码级
>
> **概述**: Kotlin 1.4 中, 当 KClass 用作泛型参数时, 会修正注解中的错误的类型映射
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化

在签名多态(signature-polymorphic) 调用中禁止展开(spread)

> **Issue**: KT-35226 (https://youtrack.jetbrains.com/issue/KT-35226)
>
> **组件**: Kotlin/JVM
>
> **不兼容性类型**: 源代码级

概述: Kotlin 1.4 将在签名多态(signature-polymorphic)调用中禁止使用展开(spread)操作符(*)

废弃周期:

- < 1.4: 对于在签名多态(signature-polymorphic)调用中使用展开(spread)操作符, 会报告警告
- >= 1.5: 将这个警告提升为错误, 可以暂时使用 `-XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall` 回退到 1.4 以前的行为

对尾递归(tail-recursive)优化函数, 默认值的初始化顺序变更

Issue: KT-31540 (https://youtrack.jetbrains.com/issue/KT-31540)

组件: Kotlin/JVM

不兼容性类型: 行为级

概述: 从 Kotlin 1.4 开始, 对尾递归(tail-recursive)函数的初始化顺序, 将与通常的函数相同

废弃周期:

- < 1.4: 对于有问题的函数, 在声明处报告警告
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-ProperComputationOrderOfTailrecDefaultParameters` 回退到 1.4 以前的行为

对非 const 的 val, 不生成 ConstantValue 属性

Issue: KT-16615 (https://youtrack.jetbrains.com/issue/KT-16615)

组件: Kotlin/JVM

不兼容性类型: 行为级

概述: 从 Kotlin 1.4 开始, 对非 `const` 的 `val`, 编译器不会生成 `ConstantValue` 属性

废弃周期:

- < 1.4: 通过 IntelliJ IDEA 的代码审查报告警告

- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-NoConstantValueAttributeForNonConstVals` 回退到 1.4 以前的行为

对 open 方法上的 @JvmOverloads 生成的 overload 应该是 final

Issue: KT-33240 (https://youtrack.jetbrains.com/issue/KT-33240)

Components: Kotlin/JVM

不兼容性类型: 源代码级

概述: 对带有 `@JvmOverloads` 注解的函数, 生成的 overload 将是 `final`

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化, 可以暂时使用 `-XXLanguage:-GenerateJvmOverloadsAsFinal` 回退到 1.4 以前的行为

返回 kotlin.Result 的 Lambda 表达式, 现在会返回装箱(box)的值, 而不是未装箱(unbox)的值

Issue: KT-39198 (https://youtrack.jetbrains.com/issue/KT-39198)

组件: Kotlin/JVM

不兼容性类型: 行为级

概述: 从 Kotlin 1.4 开始, 返回值为 `kotlin.Result` 类型的 Lambda 表达式, 将会返回装箱(box)的值, 而不是未装箱(unbox)的值

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化

统一 null 检查的相关异常

> ⚠ **Issue**: KT-22275 (https://youtrack.jetbrains.com/issue/KT-22275)
>
> **组件**: Kotlin/JVM
>
> **不兼容性类型**: 行为级
>
> **概述**: 从 Kotlin 1.4 开始, 所有的运行时 null 检查将会抛出 `java.lang.NullPointerException` 异常
>
> **废弃周期**:
>
> - < 1.4: 运行时 null 检查抛出不同的异常, 比如 `KotlinNullPointerException`, `IllegalStateException`, `IllegalArgumentException`, 和 `TypeCastException`
> - >= 1.4: all 运行时 null 检查抛出 `java.lang.NullPointerException` 异常. 可以暂时使用 `-Xno-unified-null-checks` 回退到 1.4 以前的行为

在 array/list 的 contains, indexOf, lastIndexOf 操作中的浮点值比较: 使用 IEEE 754 标准还是全顺序(total order)标准

> ⚠ **Issue**: KT-28753 (https://youtrack.jetbrains.com/issue/KT-28753)
>
> **组件**: kotlin-stdlib (JVM)
>
> **不兼容性类型**: 行为级
>
> **概述**: 从 `Double/FloatArray.asList()` 返回的 `List` 实现, 将会实现 `contains`, `indexOf`, 和 `lastIndexOf`, 因此他们会使用全顺序相等性(total order equality)
>
> **废弃周期**:
>
> - < 1.4: 旧行为 (详情请参见 issue)
> - >= 1.4: 行为有变化

将集合的 min 和 max 函数返回类型逐渐改变为非 null

> ⚠ **Issue**: KT-38854 (https://youtrack.jetbrains.com/issue/KT-38854)
>
> **组件**: kotlin-stdlib (JVM)
>
> **不兼容性类型**: 源代码级

概述: 在 1.6 中, 集合的 `min` 和 `max` 函数返回类型将会变为非 null

废弃周期:

- 1.4: 引入 `...OrNull` 函数作为同义函数, 并废弃受影响的 API (详情请参见 issue)
- 1.5.x: 将受影响的 API 的废弃级别提升为错误
- >=1.6: 重新引入受影响的 API, 但返回类型为非 null

废弃 appendln, 改为使用 appendLine

Issue: KT-38754 (https://youtrack.jetbrains.com/issue/KT-38754)

组件: kotlin-stdlib (JVM)

不兼容性类型: 源代码级

概述: `StringBuilder.appendln()` 将被废弃, 改为使用 `StringBuilder.appendLine()`

废弃周期:

- 1.4: 引入 `appendLine` 函数, 替代 `appendln`, 并废弃 `appendln`
- >=1.5: 将废弃级别提升为错误

废弃浮点类型向 Short 和 Byte 的转换

Issue: KT-30360 (https://youtrack.jetbrains.com/issue/KT-30360)

组件: kotlin-stdlib (JVM)

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 浮点类型转换为 `Short` 和 `Byte` 将被废弃

废弃周期:

- 1.4: 废弃 `Double.toShort()/toByte()` 和 `Float.toShort()/toByte()`, 并提供替代函数
- >=1.5: 将废弃级别提升为错误

在 Regex.findAll 中使用不正确的 startIndex 时快速失败

Issue: KT-28356 (https://youtrack.jetbrains.com/issue/KT-28356)

组件: kotlin-stdlib

不兼容性类型: 行为级

概述: 从 Kotlin 1.4 开始, `findAll` 函数将会改进, 会在进入 `findAll` 时, 立即检查 `startIndex` 是否在输入的字符序列的正确下标范围之内, 如果不是则抛出 `IndexOutOfBoundsException` 异常

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化

删除废弃的 kotlin.coroutines.experimental

Issue: KT-36083 (https://youtrack.jetbrains.com/issue/KT-36083)

组件: kotlin-stdlib

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 废弃的 `kotlin.coroutines.experimental` API 将从标准库中删除

废弃周期:

- < 1.4: `kotlin.coroutines.experimental` 被废弃, 级别为 `ERROR`
- >= 1.4: `kotlin.coroutines.experimental` 被从标准库删除. 在 JVM 平台, 提供了一个单独的兼容库(详情请参见 issue).

删除废弃的 mod 操作符

Issue: KT-26654 (https://youtrack.jetbrains.com/issue/KT-26654)

组件: kotlin-stdlib

不兼容性类型: 源代码级

概述: 从 Kotlin 1.4 开始, 数值类型上的 `mod` 操作符被从标准库删除

废弃周期:

- < 1.4: `mod` 被废弃, 级别为 `ERROR`
- >= 1.4: `mod` 被从标准库删除

隐藏 Throwable.addSuppressed 成员函数, 改为使用扩展函数

Issue: KT-38777 (https://youtrack.jetbrains.com/issue/KT-38777)

组件: kotlin-stdlib

不兼容性类型: 行为级

概述: 建议使用 `Throwable.addSuppressed()` 扩展函数, 而不是 `Throwable.addSuppressed()` 成员函数

废弃周期:

- < 1.4: 旧行为 (详情请参见 issue)
- >= 1.4: 行为有变化

capitalize 应该将二合字母(Digraph)转换为标题格式(title case)

Issue: KT-38817 (https://youtrack.jetbrains.com/issue/KT-38817)

组件: kotlin-stdlib

不兼容性类型: 行为级

概述: `String.capitalize()` 函数现在能够将 Serbo-Croatian Gaj 式拉丁字母表 (https://en.wikipedia.org/wiki/Gaj%27s_Latin_alphabet) 中的二合字母(Digraph) 转换为标题格式(title case) (`ǅ` 而不是 `Ǆ`)

废弃周期:

- < 1.4: 二合字母的首字母大写转换结果为: 大写格式(upper case) (`Ǆ`)
- >= 1.4: 二二合字母的首字母大写转换结果为: 标题格式(title case) (`ǅ`)

工具

在 Windows 上，带分隔字符的编译器参数必须使用双引号括起

> **Issue**: KT-41309 (https://youtrack.jetbrains.com/issue/KT-41309)
>
> **组件**: CLI
>
> **不兼容性类型**: 行为级
>
> **概述**: 在 Windows 上，`kotlinc.bat` 包含分隔字符的参数 (空格, `=`, `;`, `,`) 现在必须使用双引号(`"`)括起
>
> **废弃周期**:
>
> - < 1.4: 所有的编译器参数都不使用引号
> - >= 1.4: 包含分隔字符的编译器参数 (空格, `=`, `;`, `,`) 需要双引号(`"`)括起

KAPT: 属性的合成 $annotations() 方法名称有变化

> **Issue**: KT-36926 (https://youtrack.jetbrains.com/issue/KT-36926)
>
> **组件**: KAPT
>
> **不兼容性类型**: 行为级
>
> **概述**: 在 1.4 中，KAPT 为属性生成的合成 `$annotations()` 方法的名称有变化
>
> **废弃周期**:
>
> - < 1.4: 属性的合成 `$annotations()` 方法的名称模式是 `<propertyName>@annotations()`
> - >= 1.4: 属性的合成 `$annotations()` 方法的名称包含 `get` 前缀: `get<PropertyName>@annotations()`

Kotlin 1.3 兼容性指南

最终更新: 2024/09/10

保证语言的现代化 ([Kotlin 的演化](#)) 以及 *语言版本升级平滑便利 ([Kotlin 的演化](#))* 是 Kotlin 语言设计时的基本原则之一. 第一条原则认为, 阻碍语言演进的那些元素应该删除, 后一条原则则认为, 这些删除必须事先与使用者良好沟通, 以便让源代码的迁移尽量平滑.

尽管语言的大多数变化都通过其他途径进行了通知, 比如每次更新时的变更日志, 以及编译器的警告信息, 但我们还是在本文档中对这些变化进行一个总结, 提供一个 Kotlin 1.2 从迁移到 Kotlin 1.3 时的完整的参考列表.

基本术语

在本文档中, 我们介绍几种类型的兼容性:

- *源代码级兼容性*: 源代码级别的不兼容会导致过去能够正确编译(没有错误和警告)的代码变得不再能够编译
- *二进制级兼容性*: 如果交换两个二进制库文件, 不会导致程序的装载错误, 或链接错误, 那么我们称这两个文件为二进制兼容
- *行为级兼容性*: 如果在某个变更发生之前和之后, 程序表现出不同的行为, 那么这个变更称为行为不兼容

请记住, 这些兼容性定义只针对纯 Kotlin 程序. 从其他语言(比如, Java)的观点来看 Kotlin 代码的兼容性如何, 本文档不予讨论.

不兼容的变化

调用 <clinit> 时的构造器参数计算顺序

> **Issue**: KT-19532 (https://youtrack.jetbrains.com/issue/KT-19532)
>
> **组件**: Kotlin/JVM
>
> **不兼容性类型**: 行为级
>
> **概述**: 在 1.3 版中, 类初始化时的计算顺序有变化
>
> **废弃周期**:

- <1.3: 旧行为 (详情请参见 Issue)
- >=1.3: 行为有变化, 可以使用 `-Xnormalize-constructor-calls=disable` 参数临时退回到 1.3 以前的行为. 到下一个主版本发布时, 将会删除这个参数.

注解的构造器参数的属性取值方法的注解丢失问题

Issue: KT-25287 (https://youtrack.jetbrains.com/issue/KT-25287)

组件: Kotlin/JVM

不兼容性类型: 行为级

概述: 从 1.3 版开始, 针对注解的构造器参数的属性取值方法的注解会被正确地写入到 class 文件中

废弃周期:

- <1.3: 针对注解的构造器参数的属性取值方法的注解不会正确标注
- >=1.3: 针对注解的构造器参数的属性取值方法的注解会正确地标注, 并写入到编译生成的代码中

类构造器的 @get: 注解的错误丢失问题

Issue: KT-19628 (https://youtrack.jetbrains.com/issue/KT-19628)

组件: 核心语言

不兼容性类型: 源代码级

概述: 在 1.3 中, 取值方法的注解会正确地报告编译错误

废弃周期:

- <1.2: 取值方法的注解有错误时, 不会报告编译错误, 导致错误的代码可以被编译.
- 1.2.x: 只会通过工具报告错误, 编译器本身仍然会编译这些代码, 没有任何警告
- >=1.3: 编译器也会报告错误, 不正确的代码会被编译器拒绝

访问 @NotNull 注解标注的 Java 类型时的可空性断言

> **Issue**: KT-20830 (https://youtrack.jetbrains.com/issue/KT-20830)
>
> **组件**: Kotlin/JVM
>
> **不兼容性类型**: 行为级
>
> **概述**: 对非空注解标注的 Java 类型, 会生成更加严格的可空性断言, 因此如果传递 `null`, 会更快地失败.
>
> **废弃周期**:
>
> - <1.3: 出现类型推断时, 编译器可能会丢失这些断言, 使得编译二进制文件时可能出现 `null` 值 (详情请参见 Issue).
> - >=1.3: 编译器会生成这些断言. 这会使得那些传递了 `null` 值的错误代码更快地失败. 可以使用 `-XXLanguage:-StrictJavaNullabilityAssertions` 参数临时退回到 1.3 以前的行为. 到下一个主版本发布时, 将会删除这个参数.

对枚举类成员的智能类型转换不正确

> **Issue**: KT-20772 (https://youtrack.jetbrains.com/issue/KT-20772)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 对一个枚举值的成员的智能类型转换, 将会只适用于这个枚举值
>
> **废弃周期**:
>
> - <1.3: 对枚举值的成员的智能类型转换, 可能导致对其他枚举值的同一个成员的不正确的智能类型转换.
> - >=1.3: 智能类型转换将会正确地, 只适用于这个枚举值的成员. 可以使用 `-XXLanguage:-Sound智能类型转换ForEnumEntries` 参数临时退回到 1.3 以前的行为. 到下一个主版本发布时, 将会删除这个参数.

在取值方法中对val 型属性的后端域变量再次赋值

> **Issue**: KT-16681 (https://youtrack.jetbrains.com/issue/KT-16681)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 对于 `val` 型属性, 禁止在取值方法中对后端域变量再次赋值
>
> **废弃周期**:
>
> - <1.2: Kotlin 编译器允许在 `val` 型属性的取值方法中修改后端域变量. 这不仅违反了 Kotlin 的语法, 而且还会生成不正常的 JVM 字节码, 试图对 `final` 域变量赋值.
> - 1.2.X: 对 `val` 型属性的后端域变量赋值的代码, 会产生废弃警告
> - >=1.3: 废弃警告升级为编译错误

在对数组的 for 循环之前捕获数组

> **Issue**: KT-21354 (https://youtrack.jetbrains.com/issue/KT-21354)
>
> **组件**: Kotlin/JVM
>
> **不兼容性类型**: 源代码级
>
> **概述**: 如果 for 循环的范围表达式是一个局部变量, 并且它的值在循环体内部被修改, 那么这个修改会影响循环的执行. 这样的行为与其他容器上的循环不一致, 比如值范围(Range), 字符串序列, 集合(Collection).
>
> **废弃周期**:
>
> - <1.2: 上面讲到的这类代码能够被正常编译, 但对局部变量值的修改会影响到循环的执行
> - 1.2.X: 如果 for 循环的范围表达式是一个基于数组的局部变量, 而且在循环体内被重新赋值, 那么编译器会产生废弃警告
> - 1.3: 对这里情况改变行为, 以便与其他容器上的循环行为保持一致

枚举值内的嵌套类型

> **Issue**: KT-16310 (https://youtrack.jetbrains.com/issue/KT-16310)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 从 Kotlin 1.3 开始, 枚举值内部禁止使用嵌套类型 (类, 对象, 接口, 注解类, 枚举类)
>
> **废弃周期**:
>
> - <1.2: 枚举值内部的嵌套类型可以正常编译, 但在运行期可能发生例外, 运行失败
> - 1.2.X: 对嵌套类型会产生废弃警告
> - >=1.3: 废弃警告升级为编译错误

数据类覆盖 copy 方法

> **Issue**: KT-19618 (https://youtrack.jetbrains.com/issue/KT-19618)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 从 Kotlin 1.3 开始, 数据类禁止覆盖 `copy()` 方法
>
> **废弃周期**:
>
> - <1.2: 覆盖 `copy()` 方法的数据类, 可以正常编译, 但在运行期可能运行失败, 或者产生怪异的行为
> - 1.2.X: 对于覆盖 `copy()` 方法的数据类, 产生废弃警告
> - >=1.3: 废弃警告升级为编译错误

继承 Throwable 的内部类从外部类中捕获泛型参数

> **Issue**: KT-17981 (https://youtrack.jetbrains.com/issue/KT-17981)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级

概述: 从 Kotlin 1.3 开始, 内部类禁止继承 `Throwable`

废弃周期:

- <1.2: 继承 `Throwable` 的内部类可以正常编译. 如果这样的内部类捕获了泛型参数, 可能会导致奇怪的代码, 运行期会失败.
- 1.2.X: 对继承 `Throwable` 的内部类, 产生废弃警告
- >=1.3: 废弃警告升级为编译错误

对于带有同伴对象的复杂的类继承的可见度规则

Issues: KT-21515 (https://youtrack.jetbrains.com/issue/KT-21515), KT-25333 (https://youtrack.jetbrains.com/issue/KT-25333)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.3 开始, 涉及同伴对象和内嵌类型的复杂的类继承, 使用短名称(short name)的可见度规则变得更加严格了.

废弃周期:

- <1.2: 使用旧的可见度规则 (详情请参见 Issue)
- 1.2.X: 对于未来将会变得不再可用的短名称, 产生废弃警告. 工具可以添加完整名称, 帮助你自动迁移代码.
- >=1.3: 废弃警告升级为编译错误. 违反规则的代码需要添加完整名称, 或者明确地 import

常数以外的 vararg 注解参数

Issue: KT-23153 (https://youtrack.jetbrains.com/issue/KT-23153)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.3 开始, 禁止对 vararg 注解参数设置常数以外的值

废弃周期:

- <1.2: 编译器允许对 vararg 注解参数设置常数以外的值, 但在生成时字节码其实会抛弃这些值, 因此会导致难以理解的行为
- 1.2.X: 对这类代码产生废弃警告
- >=1.3: 废弃警告升级为编译错误

局部的注解类

Issue: KT-23277 (https://youtrack.jetbrains.com/issue/KT-23277)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.3 开始, 不再支持局部的注解类

废弃周期:

- <1.2: 编译器能够正常编译局部的注解类
- 1.2.X: 对局部的注解类, 产生废弃警告
- >=1.3: 废弃警告升级为编译错误

对局部的委托属性的智能类型转换

Issue: KT-22517 (https://youtrack.jetbrains.com/issue/KT-22517)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.3 开始， 不再允许局部的委托属性的智能类型转换

废弃周期:

- <1.2: 编译器允许对局部的委托属性的智能类型转换, 如果委托本身的行为不正确, 可能会导致不正确的智能类型转换

- 1.2.X: 对局部的委托属性的智能类型转换, 将会被警告为已废弃 (编译器产生警告信息)
- >=1.3: 废弃警告升级为编译错误

mod 运算符规约

> **Issues**: KT-24197 (https://youtrack.jetbrains.com/issue/KT-24197)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 从 Kotlin 1.3 开始, 禁止声明 `mod` 运算符, 也禁止调用这类运算符
>
> **废弃周期**:
>
> - 1.1.X, 1.2.X: 对于 `operator mod` 声明产生警告, 也对这类运算符的调用产生警告
> - 1.3.X: 警告升级为编译错误, 但还是允许对 % 运算符的调用解析到 `operator mod` 声明
> - 1.4.X: 对 % 运算符的调用不再解析到 `operator mod` 声明

以命名参数的形式向 vararg 传递单个值

> **Issues**: KT-20588 (https://youtrack.jetbrains.com/issue/KT-20588), KT-20589 (https://youtrack.jetbrains.com/issue/KT-20589). See also KT-20171 (https://youtrack.jetbrains.com/issue/KT-20171)
>
> **组件**: 核心语言
>
> **不兼容性类型**: 源代码级
>
> **概述**: 在 Kotlin 1.3 中, 将单个元素赋值给 vararg 已被废弃, 应该改用连续展开 (consecutive spread)操作和数组构造函数.
>
> **废弃周期**:
>
> - <1.2: 以命名参数的形式将单个元素赋值给 vararg, 可以正常编译, 而且会被认为是将*单个*元素赋值给一个数组, 在将数组赋值给 vararg 时会预料之外的行为
> - 1.2.X: 对这样的赋值会产生废弃警告, 建议使用者改用连续展开和数组构造函数.

- 1.3.X: 警告升级为编译错误
- >=1.4: 将会改变将单个元素赋值给 vararg 的语法含义, 使得以数组赋值等价于以数组的展开赋值

目标为 EXPRESSION 的注解的 retention 设置

Issue: KT-13762 (https://youtrack.jetbrains.com/issue/KT-13762)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.3 开始, 对于目标为 `EXPRESSION` 的注解, 它的 retention 允许设置为 `SOURCE`

废弃周期:

- <1.2: 目标为 `EXPRESSION` 的注解, 如果 retention 设置为 `SOURCE` 以外的值, 是允许的, 但是使用时会被忽略, 并且不提示任何警告信息
- 1.2.X: 对这样的注解声明会产生废弃警告
- >=1.3: 警告升级为编译错误

目标为 PARAMETER 的注解不应该用在参数的类型上

Issue: KT-9580 (https://youtrack.jetbrains.com/issue/KT-9580)

组件: 核心语言

不兼容性类型: 源代码级

概述: 从 Kotlin 1.3 开始, 如果注解的目标为 `PARAMETER`, 但被用在参数的类型上, 会正确地产生警告

废弃周期:

- <1.2: 上述不正确的代码可以正常编译; 注解会被忽略, 没有任何警告信息, 并且不会出现在编译产生的字节码中
- 1.2.X: 对注解的这种错误使用会产生废弃警告

- >=1.3: 警告升级为编译错误

当下标越界时 Array.copyOfRange 抛出异常, 而不是扩大返回的数组大小

Issue: KT-19489 (https://youtrack.jetbrains.com/issue/KT-19489)

组件: kotlin-stdlib (JVM)

不兼容性类型: 行为级

概述: `Array.copyOfRange` 函数的 `toIndex` 参数表示数组复制范围的结束位置下标(不包含在复制范围内), 从 Kotlin 1.3 开始, 会确保它不能大于数组大小, 否则会抛出 `IllegalArgumentException` 异常.

废弃周期:

- <1.3: 如果调用 `Array.copyOfRange` 时的 `toIndex` 参数大于数组大小, 那么指定的复制范围内缺少的数组元素会被填充为 `null` 值, 这会违反 Kotlin 的类型系统规则.
- >=1.3: 检查 `toIndex` 是否在数组边界内, 否则会抛出异常

步长(step)为 Int.MIN_VALUE 和 Long.MIN_VALUE 的整数和长整数的数列(progression)会被判定为非法, 并禁止创建

Issue: KT-17176 (https://youtrack.jetbrains.com/issue/KT-17176)

组件: kotlin-stdlib (JVM)

不兼容性类型: 行为级

概述: 从 Kotlin 1.3 开始, 禁止整数数列的步长(step)值设置为对应的整数类型(`Long` or `Int`)的最小值, 因此如果调用 `IntProgression.fromClosedRange(0, 1, step = Int.MIN_VALUE)` 将会抛出 `IllegalArgumentException` 异常

废弃周期:

- <1.3: 可以创建步长为 `Int.MIN_VALUE` 的 `IntProgression`, 这个数列将会产生两个值: `[0, -2147483648]`, 这是一种预期之外的行为
- >=1.3: 如果步长值是整数类型的最小值, 将会抛出 `IllegalArgumentException` 异常

对非常长的序列的操作中，检查下标溢出

> **Issue**: KT-16097 (https://youtrack.jetbrains.com/issue/KT-16097)
>
> **组件**: kotlin-stdlib (JVM)
>
> **不兼容性类型**: 行为级
>
> **概述**: 从 Kotlin 1.3 开始，在对非常长的序列的操作中，会确保 `index`, `count` 以及其他类似方法不会发生整数溢出. 关于受影响的所有方法，请参见 Issue.
>
> **废弃周期**:
>
> - <1.3: 对非常长的序列，调用这些方法可能发生整数溢出，得到负数结果
> - >=1.3: 对这些方法检查整数溢出，并立即抛出异常

使用没有匹配结果的正规表达式来切分字符串时，在各个平台上得到一致的结果

> **Issue**: KT-21049 (https://youtrack.jetbrains.com/issue/KT-21049)
>
> **组件**: kotlin-stdlib (JVM)
>
> **不兼容性类型**: 行为级
>
> **概述**: 从 Kotlin 1.3 开始，使用没有匹配结果的正规表达式来调用 `split` 方法时，在各个平台上会得到一致的结果
>
> **废弃周期**:
>
> - <1.3: 这样的调用在 JS, JRE 6, JRE 7 以及 JRE 8+ 平台会得到不同的结果
> - >=1.3: 统一各个平台的结果

在编译器的发布中不再带有已废弃的库文件

> **Issue**: KT-23799 (https://youtrack.jetbrains.com/issue/KT-23799)
>
> **组件**: 其它
>
> **不兼容性类型**: 二进制级

概述: Kotlin 1.3 不再带有以下已废弃的二进制库文件:

- `kotlin-runtime`: 请改用 `kotlin-stdlib`
- `kotlin-stdlib-jre7/8`: 请改用 `kotlin-stdlib-jdk7/8`
- `kotlin-jslib`: 请改用 `kotlin-stdlib-js`

废弃周期:

- 1.2.X: 这些库文件被标记为已废弃, 使用这些库时编译器会产生警告
- >=1.3: 这些库文件不再随编译器一起发布

stdlib 中的注解

Issue: KT-21784 (https://youtrack.jetbrains.com/issue/KT-21784)

组件: kotlin-stdlib (JVM)

不兼容性类型: 二进制级

概述: Kotlin 1.3 从 stdlib 删除了 `org.jetbrains.annotations` 包内的注解, 移动到随编译器一起发布的其他库文件中: `annotations-13.0.jar` 和 `mutability-annotations-compat.jar`

废弃周期:

- <1.3: 这些注解随 stdlib 库文件一起发布
- >=1.3: 这些注解随其他库文件一起发布

兼容模式

最终更新: 2024/09/10

当一个大的开发组迁移到一个新的版本时, 某些时候可能会出现一种 "不一致状态", 一部分开发组已经更新, 但其他开发组还没有. 为了避免前一部分开发者编写并提交的代码, 导致其他人无法编译, 我们提供以下命令行选项 (在 IDE 和 Gradle (Kotlin Gradle plugin 中的编译器选项)/Maven ("指定编译器选项" in "Maven") 中也可以使用):

- `-language-version X.Y` - Kotlin 语言版本 X.Y 兼容模式, 对这个版本以后的所有语言功能报告错误.
- `-api-version X.Y` - Kotlin API 版本 X.Y 兼容模式, 所有使用 Kotlin 标准库更高版本 API 的代码报告错误(包括编译器生成的代码).

目前, 除最新的稳定版之外, 我们还支持至少 3 个旧的语言和 API 版本的开发.

跨平台移动应用程序开发

最终更新: 2024/09/10

目前, 很多公司面临一种挑战, 需要为多个平台创建移动 App, 尤其是 Android 和 iOS. 这导致了跨平台移动开发解决方案成为最流行的软件开发趋势之一.

根据 Statista 的统计, 在 2022 年第 3 季度, Google Play Store 有 355 万移动 App, App Store 有 160 万 App, Android 和 iOS 合计现在已经占有 全世界移动操作系统市场的 99% (https://gs.statcounter.com/os-market-share/mobile/worldwide).

你如何才能创建一个移动 App, 得到 Android 和 iOS 用户? 在本文中, 你将会看到为什么越来越多的移动开发工程师选择跨平台的, 或者叫做多平台的, 移动开发方案.

跨平台移动开发: 定义与解决方案

多平台移动开发是这样一种方案, 它让你能够创建单个移动应用程序, 平滑的运行在多个操作系统上. 在跨平台 App 中, 一部分甚至全部的源代码都可以共用. 这就意味着, 开发者可以创建并部署移动应用程序, 同时在 Android 和 iOS 上工作, 不需要为各个平台重复编写代码.

移动 App 开发的各种不同策略

要为 Android 和 iOS 创建应用程序, 主要有 4 种方式.

1. 为每个操作系统创建独立的原生 App

在创建原生 App 时, 开发者会为一个特定的操作系统构建应用程序, 并且依赖于为这个平台专门设计的工具和编程语言: 对 Android 是 Kotlin 或 Java, 对 iOS 是 Objective-C 或 Swift.

这些工具和语言让你能够访问某个 OS 的功能特性, 还能够创造出 UI 符合直觉、反应灵敏的 App. 但如果你想要同时得到 Android 和 iOS 用户, 你就不得不为这两个平台分别创建应用程序, 这会耗费很多时间和精力.

2. 渐进式 Web App(Progressive Web App, PWA)

渐进式 Web App(Progressive Web App) 将移动 App 的功能与 Web 开发中使用的解决方案结合在一起. 粗略的说, 渐进式 Web App 提供了网站和移动应用程序的一种混合. 开发者使用 Web 技术来创建 PWA, 例如 JavaScript, HTML, CSS, 以及 WebAssembly.

Web 应用程序不需要分别打包和发布, 而且可以在线发布. 可以通过你的计算机, 智能手机, 以及平板上的浏览器来访问, 不需要通过 Google Play 或 App Store 来安装.

缺点是, 使用者在使用 App 时不能利用他们设备上的全部功能, 例如, 通讯录, 日历, 电话, 以及其他资源, 因此导致用户体验比较差. 从 App 性能来说, 原生 App 是最好的.

3. 跨平台 App

前面提到过, 多平台 App 会在不同的移动平台上以相同的方式运行. 跨平台框架允许你为开发这类 App 编写可共用的代码.

这种方案有很多优点, 例如在时间和成本方面都有很高效率. 我们会早后面的章节详细介绍跨平台移动开发的优点和弱点.

4. 混合 App

在浏览网站和论坛时, 你可能注意到有些人用 *"跨平台移动开发"* 和 *"混合移动开发"* 这样的词汇来代表相同的意思. 但实际上这样的看法并不完全正确.

对于跨平台 App, 移动开发工程师只需要编写一次代码, 然后可以在不同的平台重用这些代码. 混合 App 开发则不同, 它是一种结合了原生和 Web 技术的方案. 它要求你将 Web 开发语言编写的代码, 例如 HTML, CSS, 或 JavaScript, 嵌入到原生 App 之内. 你可以通过框架的帮助来实现, 例如 Ionic Capacitor 和 Apache Cordova, 并使用额外的 plugin 来使用原生平台的功能.

跨平台和混合 开发之间唯一相似的地方是代码共用. 从性能方面看, 混合应用程序不如原生 App. 因为混合 App 使用单一的代码库, 有些功能可能限定于特定的 OS, 在其他 OS 上无法正确运行.

应该选择原生还是跨平台 App 开发: 长久的争论

关于原生开发和跨平台开发的争论 (原生(Native)应用程序开发与跨平台(cross-platform)移动应用程序开发: 如何选择?) 在技术社区始终未能结局. 这两种技术都在不断演化, 而且都有各自的优点, 也有各自的局限.

有些专家仍然偏向于原生移动开发, 而不是多平台解决方案, 他们认为原生 App 更好的性能和更好的用户体验, 是它最重要的优点.

但是, 很多现代化的业务需要减少发布上市的时间, 以及每个平台开发的成本, 还要同时提供 Android 和 iOS 版本. 这就是 Kotlin Multiplatform (KMP) (https://kotlinlang.org/lp/multiplatform/) 之类的跨平台开发技术能够帮助你的地方, 正如 Netflix 的资深软件工程师 David Henry 和 Mel Yahya, 注意到 (https://netflixtechblog.com/netflix-android-and-ios-studio-apps-kotlin-multiplatform-d6d4d8d25d23):

>
> 由于网络连接质量不佳的可能性很高, 这就导致我们倾向于移动解决方案, 以便实现更健壮的客户端持久化存储和离线支持. 对于快速生产发布的需求, 导致我们尝试多平台架构. 现在我们正在更进一步, 使用 Kotlin Multiplatform, 用 Kotlin 来一次性编写平台无关的业务逻辑, 然后将其编译为 Kotlin 库, 供 Android 使用, 以及原生的 Universal 框架, 供 iOS 使用.

> Kotlin Multiplatform 入门 (https://www.jetbrains.com/kotlin-multiplatform/)

跨平台移动开发是否适合你?

选择一种适合于你的移动开发方案取决于很多因素, 例如业务需求, 目标, 任务. 和其它任何解决方案一样, 跨平台移动开发也有它的优点和弱点.

跨平台开发的优点

选择这种方案而不是其它方案的理由有很多.

1. 代码可重用

通过跨平台编程, 移动开发工程师不需要为每个操作系统编写新的代码. 使用单一的代码库让开发者可以减少在重复工作上耗费的时间, 例如 API 调用, 数据存储, 数据序列化, 以及分析实现.

Kotlin Multiplatform 这样的技术让你能够只需要一次实现你的应用程序的数据, 业务, 以及表现层. 或者, 你可以逐渐的采用 KMP: 选择一小块频繁变化, 经常会不同步的逻辑, 例如数据校验, 过滤, 或排序; 让它跨平台; 然后将它以微型库的形式连接到你的项目.

在 JetBrains, 我们定期进行 Kotlin Multiplatform 调查, 询问我们的社区成员, 他们在不同平台之间共用哪些部分的代码.

2. 节省时间

由于能够重用代码, 跨平台应用程序需要编写的代码较少, 对于编码工作来说, 代码越少越好. 可以节省开发时间, 因为你不需要编写太多代码. 而且, 代码行数越少, 发生 bug 的可能性越低, 因此测试和维护你的代码所花费的时间也更少.

3. 高效的资源管理

构建不同的应用程序的代价是很高的. 使用单一的代码库可以帮助你更加有效的管理你的资源. 你的 Android 和 iOS 开发组都可以学习如何编写和使用共通的代码.

4. 对开发者更有吸引力的工作机会

很多移动开发工程师将现代化的跨平台技术看作产品的技术栈中有吸引力的元素. 开发者可能会对那些重复的日常任务感到厌烦, 例如 JSON 解析. 但是, 新的技术和任务能够给他们带来对工作任务的激情, 积极性, 以及乐趣. 通过这样的方式, 采用现代化的技术栈, 实际上能够让你更容易为你的移动开发团队配备人员, 并使他们更长时间的保持参与和热情.

5. 获得更多用户的机会

你可以不必在不同的平台之间做选择. 由于你的 App 能够兼容于多个操作系统, 因此你能够同时满足 Android 和 iOS 用户的需求, 获得最多的用户.

6. 更快的上市时间和定制化时间

由于你不需要为不同的平台创建不同的 App, 因此你可以更快的开发并发布你的产品. 此外, 如果你的应用程序需要定制或移植, 对程序员来说, 对你的代码库的某个部分进行小的修改会容易得多. 这也可以帮助你更迅速的根据用户的反馈意见进行改进.

跨平台开发策略的难点

所有的解决方案都有自己的局限性. 技术社区的有些人认为, 跨平台开发仍然面临性能方面的问题. 而且, 项目主管可能担心, 如果他们集中努力优化开发过程, 可能对应用程序的用户体验产生不利的影响.

但是, 随着底层技术的进步, 跨平台解决方案正在变得更加 稳定 (https://blog.jetbrains.com/kotlin/2023/11/kotlin-multiplatform-stable/), 更加适用, 更加灵活.

这里是关于框架使用情况的两次 Kotlin Multiplatform 用户调查的结果, 这 2 次调查相隔 6 个月:

另一个普遍担心的问题是, 跨平台开发无法无缝支持平台的原生功能. 然而, 使用 Kotlin Multiplatform, 你可以使用 Kotlin 的 预期声明与实际声明(Expected and Actual Declarations) (预期声明与实际声明), 让你的跨平台应用程序能够访问平台相关的 API 预期声明与实际声明允许你在共同代码中定义你 "预期" 在多个平台上能够调用的相同的函数, 并提供 "实际" 实现, 通过 Kotlin 与 Java 和 Objective-C/Swift 的交互能力, 这些实际实现可以利用任何平台相关的库.

由于现代化的跨平台框架一直在持续进步, 它们不断的增强能力, 让移动开发工程师能够创建出类似于原生 App 的使用体验. 如果一个应用程序编写得足够好, 使用者将不会注意到区别. 但是, 你的产品的质量会严重依赖于你选择的跨平台 App 开发工具.

最流行的跨平台解决方案

最流行的跨平台框架 (跨平台应用程序开发最流行的 6 种框架) 包括 Flutter, React Native, 以及 Kotlin Multiplatform. 这些框架的每一种都有它的长处. 取决于你使用的工具, 你的开发过程和成果会很不一样.

Flutter

Flutter 由 Google 创建, 是一个跨平台开发框架, 使用 Dart 编程语言. Flutter 支持原生功能, 例如定位服务, 摄像头功能, 以及硬盘访问. 如果你需要创建某个 Flutter 不支持的 App 功能, 你可以使用 Platform Channel 技术 (https://brightmarbles.io/blog/platform-channel-in-flutter-benefits-and-limitations/), 编写平台相关的代码.

使用 Flutter 构建的 App 需要共用它们所有的 UX 和 UI 层, 因此它们可能并不会 100% 感觉象原生 App. 这个框架最好的功能之一, 是它的热加载(Hot Reload)功能, 可以让开发者修改代码, 并立即看到结果.

对于以下情况, 这个框架可能是最好的选择:

- 你想要在你的 App 之间共用 UI 组件, 但你希望你的应用程序看起来接近原生 App.
- App 工作时会产生很重的 CPU/GPU 负载, 而且需要性能优化.
- 你需要开发 MVP(Minimum Viable Product, 最简可行产品) 应用程序.

使用 Flutter 构建的最流行的 App 包括 Google Ads, Alibaba 公司的 Xianyu, eBay Motors, 以及 Hamilton.

React Native

Facebook 于 2015 年将 React Native 发布为开源框架, 它用于帮助移动开发工程师构建原生/跨平台 混合 App. 它基于 ReactJS – 一个用于构建 UI 的 JavaScript 库. 换句话说, 它使用 JavaScript 来为 Android 和 iOS 系统构建移动 App.

React Native 能够访问几种第三方 UI 库, 其中包含可以立即使用的组件, 帮助移动开发工程师在开发过程中节约时间. 与 Flutter 类似, 它提供了 Fast Refresh 功能, 你修改过代码后, 可以立即看到的结果.

对于以下情况, 你应该考虑为你的 App 使用 React Native:

- 你的应用程序相对简单, 轻量.
- 开发团队熟悉 JavaScript 或 React.

使用 React Native 构建的 App 包括 Facebook, Instagram, Skype, 以及 Uber Eats.

Kotlin Multiplatform

Kotlin Multiplatform 是由 JetBrains 提供的开源技术, 它允许开发者跨平台共用代码, 同时又保留原生编程的优点. 它的关键优点包括:

- 能够在 Android, iOS, Web, Desktop, 以及 Server 端重用代码, 同时, 如果需要, 也可以保持原生代码.
- 与既有项目平滑集成. 你可以使用平台相关的 API, 充分利用原生开发和跨平台开发.
- 完全的代码共用灵活性, 能够共用逻辑和 UI, 这要归功于 Compose Multiplatform (https://www.jetbrains.com/lp/compose-multiplatform/), 由 JetBrains 创建的一个现代化的声明式跨平台 UI 框架.

* 如果你已经在 Android 开发中使用了 Kotlin, 那么不需要向你的代码库引入新的语言. 你可以继续沿用你的 Kotlin 代码和技能, 因此与其他技术相比, 迁移到 Kotlin Multiplatform 的风险更低.

> Kotlin Multiplatform 入门 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html).

McDonald's, Netflix, 9GAG, VMware, Cash App, Philips, 以及其他很多公司都已经在利用 Kotlin Multiplatform 的逐步集成能力, 以及它比较低的采用风险. 其中一些公司选择共用他们既有的 Kotlin 代码的关键部分, 来增强应用程序稳定性. 另一些公司的目标是, 在不影响应用程序质量的情况下最大化代码的重用, 并在移动, 桌面, Web, 以及 TV 平台, 共用所有的应用程序逻辑, 同时在每个平台保留原生 UI. 从采用了这种方案的公司的成功故事来看, 它的优势是很明显的.

> 查看所有的 全球公司和初创企业使用 Kotlin Multiplatform 的案例 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/case-studies.html)

结论

由于跨平台开发解决方案一直在持续进化, 与它们提供的好处相比, 它们的弱点已经开始变少了. 目前市场上有很多技术可供选择, 各种技术适用于不同的工作流程和需求. 对开发团队来说, 本文讨论的每一种工具都代表一些参考信息, 鼓励他们考虑试验一下跨平台开发方案.

根本上来说, 你需要仔细考虑你自己的业务需求, 目标, 以及任务, 想清楚你通过你的 App 想要达到什么目标, 这将会有助于找到对你来说最好的解决方案.

原生(Native)应用程序开发与跨平台(cross-platform)移动应用程序开发: 如何选择?

最终更新: 2024/09/10

人们醒着的时间大量用在他们的移动设备上. 他们还 每天在移动应用程序上花费 4.8 小时 (https://www.data.ai/en/insights/market-data/state-of-mobile-2022/), 因此移动平台对任何商业类型都非常具有吸引力.

移动 App 开发技术还在持续进化中, 每年都会出现新的技术和框架. 由于市场上存在许多解决方案, 因此经常会难以作出选择. 你可能已经听说过 "原生 vs 跨平台" 的长期争论.

在创建一个 App 之前有很多因素需要考虑, 例如开发成本, 时间, 以及 App 的功能. 如果你想要同时针对 Android 和 iOS 用户, 那么这些因素尤其重要. 对你的某个项目来说, 使用哪种移动开发策略才是最好的, 可能会很难决定. 为了帮助你在原生和跨平台 App 开发之间进行选择, 我们创建了一个列表, 包括值得注意的 6 个基本因素.

什么是原生移动 App 开发?

原生移动开发意思是说, 你针对某个移动操作系统来创建一个 App – 大多数情况下是 Android 或 iOS. 开发原生应用程序时, 开发者会使用特定的编程语言和工具. 例如, 你可以使用 Kotlin 或 Java 创建一个原生 Android 应用程序, 或者使用 Objective-C 或 Swift 创建一个 iOS App.

下面是原生移动 App 的主要优势和不利点.

优势	不利点
高性能. 用来创建原生 App 使用的核心编程语言和 API 使得这些 App 快速而且反应灵敏.	**高成本.** 原生 App 开发需要 2 组独立的开发队伍, 分别具备不同的技能, 因此增加了开发过程的时间和成本.
用户体验直观. 移动开发工程师使用原生 SDK 来开发原生 App, 因此 UI 风格一致. 原生 App 的界面被设计得更加适应于在特定的平台, 因此它们感觉象与设备融合在一起的一部分, 而且提供更加直观的用户体验.	**开发团队大.** 管理很多专业人士组成的大团队可能会非常困难. 为一个项目工作的人越多, 为沟通和协作而耗费的精力就越多.
能够使用特定设备的全部功能. 针对特定操作系统创建的原生 App 能够直接访问设备的硬件, 例如摄像头, 麦克风, 以及 GPS 定位功能.	**代码中的错误更多.** 更多行数的代码意味着更多 bug 存在的风险.
	在 Android 和 iOS App 中存在业务逻辑差异的风险. 使用原生 App 开发方案, 针对一个移动平台编写的代码无法在另一个平台上使用. 例如, 由于折扣计算方式的错误, Android 和 iOS App 可能对同一个商品显出不同的价格.

什么是跨平台 App 开发?

跨平台 App 开发, 也叫做多平台开发, 意思是说创建兼容多个操作系统的移动 App. 移动开发工程师能够在多个平台之间共用一部分, 甚至全部的源代码, 而不是为 iOS 和 Android 创建不同的应用程序. 通过这种方式, 应用程序在 iOS 和 Android 的工作结果是一样的.

现在已经有了很多用于 跨平台移动 App 开发 (跨平台移动应用程序开发) 的开源框架. 其中最流行的一些是 Flutter, React Native, 以及 Kotlin Multiplatform Mobile. 下面是这种方式的主要优势和不利点.

优势	不利点
代码共用. 开发者可以创建单一的代码库, 不必为每个 OS 编写新代码.	**性能问题.** 有些开发者 认为多平台应用程序与原生 App 相比性能比较低.
开发更快速. 你不需要编写或测试更多代码, 因此可以帮助你加快开发速度.	**难于使用移动设备的原生功能.** 创建一个需要访问平台专有 API 的跨平台 App, 需要耗费更多精力.
低成本. 对于初创企业和预算有限的公司, 跨平台解决方案是一个很好的选择, 因为可以降低开发成本.	**UI 一致性不佳.** 使用跨平台开发框架使得你可以共用 UI, 应用程序的外观和使用体验可能不太象原生程序.
新的工作机会. 在你的产品技术栈中包含现代化的跨平台技术, 可以吸引新的人才加入你的团队. 很多开发者想要在工作中探索新的挑战, 所以新的技术和任务有助于提升开发者的热情, 并让他们在工作时感到更加有趣.	**招聘困难.** 与原生 App 开发者相比, 找到能够创建多平台 App 的专家可能更加困难. 例如, 撰写本文时, 我们在 Glassdoor 上找到大约 2,400 个 Android 开发者职位, 而 Flutter 开发者职位只有 348 个. 但是, 随着跨平台技术的持续进步, 并吸引更多的移动开发工程师, 这样的情况可能会发生改变.
共用业务逻辑. 由于这个方案使用单一的代码库, 你可以确保在不同的平台的应用程序逻辑是完全相同的.	

这只是跨平台 App 开发的优势中很少的一部分. 关于它的优势, 以及在全球各大公司的使用场景, 请参见我们关于 跨平台移动开发 (跨平台移动应用程序开发) 的文章. 关于这个方案的问题 – 我们在下面的章节中进行讨论.

关于跨平台 App 开发的一些常见的误解

跨平台技术一直在持续演进. 有些跨平台开发框架, 例如 Kotlin Multiplatform Mobile (https://kotlinlang.org/lp/multiplatform/) 提供了构建跨平台和原生 App 两种方案的优势, 解决了跨平台方案的一些常见限制.

1. 跨平台 App 的性能低于原生 App.

性能低下长期被认为是多平台应用程序的主要劣势. 但是, 你的产品的性能和质量 很大程度上依赖于你用来构建 App 的工具. 最新的跨平台框架提供了开发出类似于原生程序一样的用户体验的 App 所需要的所有工具.

通过使用不同的编译器后端, Kotlin ("在不同平台间共用代码" in "Kotlin Multiplatform") 代码被编译为平台格式 – 在 Android 平台是 JVM 字节码, 在 iOS 平台是原生二进制代码. 因此, 你的共用代码的性能, 和你使用原生代码编写它们是一样的.

2. 跨平台框架不安全.

有一个常见的误解是, 原生 App 更加安全更加可靠. 但是, 现代化的跨平台开发工具可以帮助开发者创建安全的 App, 提供可靠的数据保护. 移动开发工程师只需要 添加额外的指标来提升他们的 App 的安全性 (https://appstronauts.co/blog/are-cross-platform-apps-as-fast-and-secure-as-native-apps/#:~:text=Unsecurity%20of%20cross%2Dplatform%20apps,a%20cross%2Dplatform%20app%27s%20code.).

3. 跨平台 App 不能访问移动设备的全部原生功能.

的确, 并不是所有的跨平台框架都能允许你创建 App 来访问设备的全部功能. 但是, 有些现代化的多平台框架能够帮助你解决这个问题. 例如, Kotlin Multiplatform Mobile 可以很容易的访问 Android 和 iOS SDK. 它提供了一个 Kotlin 预期声明与实际声明(expected and actual declarations) 机制 (预期声明与实际声明), 可以帮助你访问设备的功能特性.

4. 管理跨平台项目很困难.

实际上, 恰恰相反. 跨平台解决方案能够帮助你更加有效的管理资源. 你的开发团队可以学习如何编写和复用共用的代码. Android 和 iOS 开发者通过互相沟通和分享知识, 可以达到很高的效率和高度的信息透明.

帮助你在跨平台 App 开发和原生方案之间作出选择的 6 个关键因素

下面, 我们来看看你为移动 App 开发选择原生和跨平台解决方案时需要考虑的一些重要因素.

1. 你未来的 App 的类型和目的

第一步应该是理解你要创建什么样的 App, 包括它的功能和目的. 一个带有很多功能的复杂的应用程序需要很多编程工作, 尤其是如果它是一个全新的 App、没有太多现成的模板可以借鉴.

你的 App 的用户界面有多重要? 你想要非常漂亮的视觉效果吗, 或者 UI 并不重要? 它是否需要任何特定的硬件功能, 是否需要访问摄像头和 GPS 定位功能? 你需要确保你选择的移动开发策略提供了必要的工具来构建你需要的 App, 而且提供了很好的用户体验.

2. 你的团队在编程语言和工具方面的经验

你团队中的开发者需要具有足够的经验和专业技能来使用某个特定的框架. 需要仔细考虑开发工具

需要什么样的编程技能和编程语言.

例如, 开发者需要懂得 Objective-C 或 Swift 才能创建 iOS 的原生 App, 需要懂得 Kotlin 或 Java 才能创建 Android 的原生 App. 跨平台框架 Flutter 需要懂得 Dart 知识. 如果你使用 Kotlin Multiplatform Mobile, 对 iOS 开发者来说 Kotlin 语法是很容易学习的, 因为它遵循与 Swift 相似的概念.

3. 各种技术在未来是否能够长期存在

选择不同的方案和框架时, 你需要确信平台供应商能够在未来长期支持它. 你可以挖掘供应商的细节, 他们的开发社区规模, 被全球各大公司采用的程度. 例如, Kotlin Multiplatform Mobile 的开发者是 JetBrains, Flutter 的开发者是 Google, React Native 的开发者是 Facebook.

4. 开发成本和你的预算

上面提到过, 不同的移动开发解决方案和工具会带来不同的开支. 根据你的预算灵活度, 你可以为你的项目选择适当的解决方案.

5. 业内采用的广泛度

你总是能够找到技术社区的其他专家们对各种不同方案的意见. Reddit, StackOverflow, 以及 Google Trends 是几个不错的消息源. 请看看以下两个词语的搜索趋势: "原生移动开发" vs "跨平台移动开发". 很多用户仍然对学习原生 App 开发感兴趣, 但似乎跨平台方案正在获得更多关注.

如果一种技术被专业人士广泛使用, 它会拥有一个很强的生态环境, 大量的库, 以及技术社区的最佳实践, 这些都可以让你的开发工作更加快速.

6. 流行度与学习资源

如果你在考虑试用跨平台 App 开发, 你需要考虑的一个因素是, 找到各个多平台框架的学习资料的难易程度. 请查看它们的官方文档, 书籍, 以及课程. 请确认它们提供了带有长期计划的 产品路线图 (https://blog.jetbrains.com/kotlin/2022/06/what-to-expect-from-the-kotlin-team-in-202223/).

什么情况下你应该选择跨平台 App 开发?

在针对 Android 和 iOS 构建应用程序时, 为移动 App 开发的跨平台解决方案可以节省你的时间和工作.

简单的说, 对于下面的情况, 你应该选择跨平台解决方案:

- 你需要创建一个同时运行于 Android 和 iOS 的 App.
- 你想要节约开发时间.

- 你想要对 App 的业务逻辑只保持单一的代码库, 同时又能够完全控制它们的 UI 元素. 并不是所有的跨平台框架都允许你实现这样的功能, 但是有一些框架, 例如 Kotlin Multiplatform Mobile, 提供了这样的能力.
- 你热切希望采用持续演进中的现代化技术.

> 请在你的 iOS 和 Android App 中共用业务逻辑. 参见 Kotlin Multiplatform (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html) 中的实际例子.

什么情况下你应该选择跨平台原生 App 开发?

对少数集中情况, 选择原生移动开发会更加合适. 对于下面的情况, 你应该选择这个方案:

- 你的 App 只针对某一种用户 – Android 或 iOS.
- 用户界面对你的应用程序非常重要. 但是, 即使你采用原生方案, 你也可以尝试使用多平台移动 App 开发方案, 它可以允许你对你的项目共用 App 的业务逻辑, 但不共用 UI.
- 你的团队拥有极高技能的 Android 和 iOS 开发者, 但你没有时间来学习新的技术.

结束语

请关注上面介绍的各种因素, 你的项目的目标, 以及最终用户. 无论你喜欢原生开发还是跨平台开发, 都取决于你独特的需求. 每个解决方案都有它的长处, 也有它的弱点.

无论如何, 需要关注技术社区正在发生什么事. 知道最新的移动开发趋势, 可以帮助你为你的项目作出最好的选择.

跨平台应用程序开发最流行的 6 种框架

最终更新: 2024/09/10

过去几年, 跨平台 App 开发已经成为创建移动应用程序的最流行方式. 一个跨平台方案, 或者叫多平台方案, 能够帮助开发者创建 App, 在不同的移动平台上以类似的方式运行.

Google 搜索趋势图显示, 自从 2010 以来, 开发者对跨平台方案的兴趣一直在稳定增长:

快速进步的 跨平台移动开发 ("Kotlin Multiplatform" in "跨平台移动应用程序开发") 技术的流行度增长, 导致市场上出现了更多的新工具. 面对这样多的选择, 要选择最适合你的一种可能会很困难. 为了帮助你找到正确的工具, 我们整理了一份列表, 包括 6 个最好的跨平台 App 开发框架, 以及它们各自的优秀功能. 在本文的最后, 你将看到为你的业务选择多平台开发框架时值得注意的几个关键因素.

什么是跨平台 App 开发框架?

移动开发工程师使用跨平台移动开发框架, 为多个平台(例如 Android 和 iOS)构建外观类似原生程序的应用程序, 只需要单个代码库. 与原生 App 开发相比, 代码共用是这种方案的关键优势之一. 只需要单个代码库意味着移动开发工程师不必为每个操作系统编写代码, 因此可以节省时间, 加快开发速度.

流行的跨平台 App 开发框架

这里列出的并不包括所有的框架; 现在市场上还有很多其他选择. 值得注意的是, 不存在完美的万能工具能够适合所有人. 对框架的选择 很大程度上取决于你的具体项目, 你的目标, 以及其他因素, 我们会在本文末尾进行介绍.

总之, 我们尽力列举出跨平台移动开发的一部分最好的框架, 作为你进行决策的参考.

Flutter

2017 年由 Google 发布, Flutter 是一个流行的框架, 可以使用单一代码库来构建移动 App, Web App, 以及桌面 App. 要使用 Flutter 构建应用程序, 你需要使用 Google 的编程语言 Dart.

编程语言: Dart.

移动应用程序示例: eBay, Alibaba, Google Pay, ByteDance App.

主要功能特性:

- Flutter 的热加载(Hot Reload) 功能可以让你在修改代码后立即看到应用程序如何变化, 你不需

要重新编译它.

- Flutter 支持 Google 的 Material Design, 这是一个设计系统, 帮助开发者构建数字化的用户体验. 在构建你的 App 时, 你可以使用很多可视化元件(Visual Widget)和行为元件(Behavioral Widget).
- Flutter 不依赖于 Web 浏览器技术. 相反, 它使用自己的渲染引擎来描绘元件.

Flutter 拥有遍及全世界、相对活跃的用户社区, 并被很多开发者广泛使用. 根据 Stack Overflow Trends (https://insights.stackoverflow.com/trends?tags=flutter%2Creact-native), 由于相应的 tag 的使用量不断增加, 可以看出 Flutter 的使用随着时间的推移呈增加趋势.

React Native

一个开源 UI 软件框架, React Native 2015 年 (比 Flutter 稍早) 由 Meta Platforms 开发, 以前叫做 Facebook. 它基于 Facebook 的 JavaScript 库 React, 允许开发者构建原生渲染的跨平台移动 App.

编程语言: JavaScript.

移动应用程序示例: React Native 被应用于 Microsoft 公司的 Office, Skype, 以及 Xbox Game Pass; Meta 公司的 Facebook, Desktop Messenger, 以及 Oculus. 更多案例请参见 React Native 案例展示 (https://reactnative.dev/showcase).

主要功能特性:

- 由于拥有 Fast Refresh 功能, 开发者可以立即看到他们在他们的 React 组件中的变更.
- React Native 优势之一是集中于 UI. React primitives 会渲染为原生平台 UI 组件, 因此你可以构建自定义的、响应式的用户界面.
- 在 0.62 及之后版本, 默认启用了 React Native 和移动 App 调试器 Flipper 之间的集成. Flipper 用来调试 Android, iOS, 和 React native App, 它提供了很多工具, 例如日志查看器, 交互式布局查看器 , 以及网络监控器.

React Native 是最流行的跨平台 App 开发框架之一, 它拥有强大的开发者社区, 分享他们的技术知识. 感谢这些社区的存在, 你在使用这个框架构建移动 App 时可以得到你需要的支持.

Kotlin Multiplatform

Kotlin Multiplatform (KMP) 是由 JetBrains 提供的开源技术, 它允许跨平台共用代码, 同时又保留原生编程的优点. 它允许开发者尽可能多的重用代码, 如果需要也可以编写原生代码, 并能够将共用的 Kotlin 代码无缝的集成到任何项目中.

编程语言: Kotlin.

移动应用程序示例: McDonald's, Netflix, Forbes, 9GAG, Cash App, Philips. 参见 Koltlin Multiplatform 使用案例 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/case-studies.html).

主要功能特性:

- 开发者能够在 Android, iOS, Web, Desktop, 以及 Server 端重用代码, 同时, 如果需要, 也可以保持原生代码.
- Kotlin Multiplatform 能够与任何项目无缝集成. 开发者可以使用平台相关的 API, 充分利用原生开发和跨平台开发.
- 感谢 Compose Multiplatform (https://www.jetbrains.com/lp/compose-multiplatform/), 由 JetBrains 创建的一个现代化的声明式跨平台 UI 框架, 开发者拥有完全的代码共用灵活性, 能够共用逻辑和 UI.
- 如果你已经在 Android 开发中使用了 Kotlin, 那么不需要向你的代码库引入新的语言. 你可以继续沿用你的 Kotlin 代码和技能, 因此与其他技术相比, 迁移到 Kotlin Multiplatform 的风险更低.

即使这个跨平台移动开发框架是我们的列表中最新的框架之一, 但它已经有了成熟的开发者社区. 2023年11月, JetBrains 将它提升到了 稳定版 (https://blog.jetbrains.com/kotlin/2023/11/kotlin-multiplatform-stable/). 这个框架正在快速成长, 已经给今天的市场留下了深刻的印象. 由于它持续更新的 文档 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/get-started.html) 和社区支持, 你遇到问题时总是能够找到答案. 此外, 很多 全球公司和初创企业已经在使用 Kotlin Multiplatform (https://www.jetbrains.com/help/kotlin-multiplatform-dev/case-studies.html) 来开发带有近似原生程序用户体验的多平台 App.

使用 Kotlin Multiplatform (https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-getting-started.html) 创建你的第一个跨平台移动 App.

Ionic

Ionic 是一个开源的移动 UI 工具库, 于 2013 发布. 它帮助开发者通过单一代码库创建跨平台移动应用程序, 它使用 Web 技术, 例如 HTML, CSS, 以及 JavaScript, 与 Angular, React, 以及 Vue 框架集成.

编程语言: JavaScript.

移动应用程序示例: T-Mobile, BBC (儿童与教育 App), EA Games.

主要功能特性:

- Ionic 基于一个 SaaS UI 框架, 专门针对移动 OS 设计, 提供了用于创建应用程序的很多 UI 组件.
- Ionic 框架 使用 Cordova 和 Capacitor plugin 来访问设备的内建功能, 比如摄像头, 手电筒, GPS, 以及录音机.
- Ionic 拥有自己的命令行, Ionic CLI, 它是构建 Ionic 应用程序的首选工具.

拥有持续活跃的 Ionic Framework 论坛, 社区成员在这里交流知识, 相互帮助, 解决他们在开发中遇到的问题.

.NET MAUI

.NET Multi-platform App UI (.NET MAUI) 是一个跨平台框架, 2022年5月发布, 由 Microsoft 拥有. 它允许开发者使用 C# 和 XAML 创建原生的移动应用程序和桌面应用程序. .NET MAUI 是 Xamarin.Forms 的后续, Xamarin.Forms 是 Xamarin 的功能之一, 它为 Xamarin 支持的平台提供原生控件.

编程语言: C#, XAML.

移动应用程序示例: NBC Sports Next, Escola Agil, Irth Solutions.

主要功能特性:

- .NET MAUI 提供跨平台 API 用于访问原生设备功能, 例如 GPS, 加速度计, 以及电池和网络状态.
- 有一个单一的项目系统, 可以使用多编译目标, 针对 Android, iOS, macOS, 以及 Windows 进行开发.
- 通过对 .NET 热重载(hot reload)的支持, 开发人员可以在应用程序正在运行时修改托管源代码 (managed source code).

尽管 .NET MAUI 仍然是一个相对比较新的框架, 它已经得到了开发人员的关注, 并在 Stack Overflow 和 Microsoft Q&A 拥有活跃的社区.

NativeScript

这个开源的移动应用程序开发框架初次发布于 2014 年. NativeScript 可以帮助你构建 Android 和 iOS 移动 App, 使用的语言是 JavaScript, 或能够翻译到 JavaScript 的其他语言, 例如 TypeScript, 使用的框架是 Angular 和 Vue.js.

编程语言: JavaScript, TypeScript.

移动应用程序示例: Daily Nanny, Strudel, Breethe.

主要功能特性:

- NativeScript 允许开发者容易的访问 Android 和 iOS 原生 API.
- 这个框架渲染为平台原生的 UI. 使用 NativeScript 构建的 App 直接运行在原生设备上, 不需要依赖于 WebView, WebView 是 Android OS 的一个系统组件, 供 Android 应用程序在 App 内显示 Web 内容.
- NativeScript 提供了很多 plugin 和预构建的 App 模板, 因此不需要第三方解决方案.

NativeScript 基于广泛流行的 Web 技术, 例如 JavaScript 和 Angular, 这是很多开发者选择这个框架的原因. 然而, 它通常由小公司或初创企业采用.

你应该如何为你的项目选择正确的跨平台 App 开发框架?

除了上面列举的之外, 还有其它跨平台框架, 而且新的工具还会不断出现. 有了这么多的选择, 你怎么样才能为你的下一个项目找到正确的方案? 第一步是要理解你的项目的需求和目标, 清楚的理解你希望你的 App 是怎么样的. 然后, 你需要考虑下面这些重要因素, 然后你就可以决定哪个方案最适合于你的业务.

1. 你的开发团队的专长

不同的跨平台移动开发框架基于不同的编程语言. 在采用一个框架之前, 首先要确认它要求的技能, 确认你的移动开发工程师团队具备了足够的知识和经验开使用这个框架.

例如, 如果你的开发团队拥有高度技能的 JavaScript 开发者, 而且你没有足够的资源来采用新的技术, 那么可能应该选择使用这个语言的框架, 例如 React Native.

2. 开发商的可靠程度和支持程度

要确认框架的维护者未来还会长期支持它, 这是很重要的问题. 对于你正在考虑的框架, 应该了解开发和支持它的公司, 还要调查一下使用这些框架创建的移动 App 有哪些.

3. UI 定制

根据 UI 对于你的 App 的重要性不同, 你可能需要了解使用某个框架时定制 UI 的难易程度. 例如, 通过 Compose Multiplatform (https://www.jetbrains.com/lp/compose-multiplatform/), 由 JetBrains 创建的一个现代化的声明式跨平台 UI 框架, Kotlin Multiplatform 提供了完全的代码共用灵活性. 它允许开发者在 Android, iOS, Web, 以及 Desktop (通过 JVM) 平台之间共用 UI, 它基于 Kotlin 和 Jetpack Compose 开发.

Compose Multiplatform 入门 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-multiplatform-getting-started.html)

4. 框架成熟度

要了解某个框架的 Public API 和工具的更新频度. 例如, 对原生操作系统组件的某些变更会破坏内部的跨平台行为. 使用移动 App 开发框架时, 最好了解你可能遇到的问题. 你也可以查看 GitHub, 看看这个框架目前还有多少 bug, 以及维护者如何处理这些 bug.

5. 框架的能力

每个框架都有它自己的能力和缺陷. 了解框架提供了什么样的功能特性和工具, 对于寻找最好的解决方案是至关重要的. 它是否拥有代码分析和单元测试框架? 你在构建, 调试, 以及测试你的 App 时, 有多快, 有多容易?

6. 安全性

在为商业创建重要的移动 App 时, 安全性和隐私是非常重要的, 例如, 包含支付系统的银行和电子商务 App. 根据 OWASP Mobile Top 10 (https://owasp.org/www-project-mobile-top-10/), 移动 App 的最严重安全风险包括不安全的数据存储, 以及身份认证(Authentication)/用户授权(Authorization).

你需要确认你选择的多平台移动开发框架提供了必要的安全等级. 一种方法是, 如果框架拥有可供公共查看的问题追踪系统, 那么可以去查看其中的安全性问题.

7. 教学资料

关于一个框架的学习资源的数量和质量, 也可以帮助你理解在使用这个框架时的体验会如何. 复杂的官方 文档 (https://www.jetbrains.com/help/kotlin-multiplatform-dev/get-started.html), 线上和线下的会议, 以及教学课程是很好的迹象, 表示在你需要的时候, 你能够找到关于这个产品的足够的必要信息.

结束语

不考虑这些因素, 选择最适合你的特定需求的跨平台移动开发框架, 那将会很困难. 应该仔细考察你的应用程序的需求, 并以此为根据来评估各个框架的能力. 这样, 你就可以找到能够帮助你开发出高质量 App 的正确的跨平台解决方案.

使用 Kotlin 参加 Google 编程夏令营 2024

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/gsoc-2024.html)

使用 Kotlin 参加 Google 编程夏令营 2023

最终更新: 2024/09/10

本章不翻译, 请阅读 原文 (https://kotlinlang.org/docs/gsoc-2023.html)

安全性

最终更新: 2024/09/10

我们尽力保证我们的产品不存在安全性漏洞. 为了减少发生安全性漏洞的风险, 请遵守以下原则:

- 总是使用 Kotlin 的最新发布版. 为了安全性目的, 我们将发布版公布到 Maven Central (https://central.sonatype.com/search?q=g:org.jetbrains.kotlin), 使用以下 PGP key:
  - Key ID: **kt-a@jetbrains.com**
  - Fingerprint: **2FBA 29D0 8D2E 25EE 84C1 32C3 0729 A0AF F899 9A87**
  - Key size: **RSA 3072**
- 对你的应用程序的依赖项使用最新版. 如果你需要使用一个依赖项的特定版本, 请定期检查是否发现了新的安全性漏洞. 你可以遵照
- GitHub 的依赖项安全管理指南 (https://docs.github.com/ja/enterprise-cloud@latest/code-security/supply-chain-security/managing-vulnerabilities-in-your-projects-dependencies), 或在 CVE 数据库 (https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=kotlin) 中查阅已知的安全性漏洞.

如果你能报告你发现的任何安全性问题, 我们非常期待并表示感谢. 要报告你在 Kotlin 中发现的安全性漏洞, 请直接在我们的 问题追踪系统 (https://youtrack.jetbrains.com/newIssue?project=KT&c=Type%20Security%20Problem) 中提交一条消息, 或者向我们发送 email (mailto:security@jetbrains.org).

关于我们对安全性问题的负责任的披露流程(Responsible Disclosure Process), 详情请参见 JetBrains 协调披露政策(Coordinated Disclosure Policy) (https://www.jetbrains.com/legal/docs/terms/coordinated-disclosure/).

Kotlin 文档 (PDF 格式)

最终更新: 2024/09/10

你可以在这里下载 Kotlin 文档的 PDF 版本, 其中包含除教程和 API 参考文档之外的所有内容.

下载 Kotlin 1.9.20 文档 (PDF) (https://kotlin.liying-cn.net/resources/kotlin-reference.pdf)

阅读 Kotlin 最新版文档 (在线) (Kotlin 语言参考文档)

为 Kotlin 项目贡献代码

最终更新: 2024/09/10

Kotlin 是一个开源项目, 使用 Apache 2.0 许可协议 (https://github.com/JetBrains/kotlin/blob/master/license/LICENSE.txt). 源代码, 工具, 文档, 已经本网站, 都在 GitHub (https://github.com/jetbrains/kotlin) 维护. 尽管 Kotlin 主要由 JetBrains 开发, 但 Kotlin 项目还有几百名外部贡献者, 而且我们始终希望更多的开发者帮助我们.

参加早期预览(Early Access Preview)项目

你可以 参加 Kotlin 早期预览(Early Access Preview, EAP) 项目 (参加 Kotlin EAP 项目), 并想我们提供宝贵的反馈意见, 帮助我们改进 Kotlin.

对每一个正式发布版, Kotlin 都会发布几个预览版, 你可以在最新功能正式发布之前进行试用. 你可以向我们的问题追踪系统 YouTrack (https://kotl.in/issue) 报告你发现的 bug, 我们会尝试在最终发布之前修复这些 bug. 通过这种方式, 你报告的 bug 可以比通常的 Kotlin 发布周期更快修复.

向编译器和标准库贡献代码

如果你想要向 Kotlin 编译器和标准库贡献代码, 可以访问 JetBrains/Kotlin GitHub (https://github.com/jetbrains/kotlin), 下载最新的 Kotlin 版本, 然后按照文档 如何向项目贡献代码 (https://github.com/JetBrains/kotlin/blob/master/docs/contributing.md) 中的步骤进行.

你可以帮助我们解决 未完成的任务 (https://youtrack.jetbrains.com/issues/KT?q=tag:%20%7BUp%20For%20Grabs%7D%20and%20State:%20Open). 请和我们保持联系, 因为我们可能会有一些疑问, 并对你提交的修改留下评论. 否则, 我们不能将你贡献的代码合并到项目中.

向 Kotlin IDE plugin 贡献代码

Kotlin IDE plugin 是 IntelliJ IDEA 代码仓库 (https://github.com/JetBrains/intellij-community/tree/master/plugins/kotlin) 的一部分.

要向 Kotlin IDE plugin 贡献代码, 请 clone IntelliJ IDEA 代码仓库 (https://github.com/JetBrains/intellij-community/), 然后按照文档 如何贡献代码 (https://github.com/JetBrains/intellij-community/blob/master/plugins/kotlin/CONTRIBUTING.md) 中的步骤进行.

向其他 Kotlin 库和工具贡献代码

除提供核心功能的标准库之外, Kotlin 还有很多额外的 (kotlinx) 库, 提供更多扩展功能. 每个 kotlinx 库都在单独的代码仓库中开发, 拥有自己的版本和发布周期.

如果你想要对某个 kotlinx 库 (比如 kotlinx.coroutines (https://github.com/Kotlin/kotlinx.coroutines) 或 kotlinx.serialization (https://github.com/Kotlin/kotlinx.serialization)) 和工具贡献代码, 请访问 Kotlin GitHub (https://github.com/Kotlin), 选择你感兴趣的代码仓库, 并 clone 它.

然后按照各个库和工具的文档中的步骤进行, 比如 kotlinx.serialization (https://github.com/Kotlin/kotlinx.serialization/blob/master/CONTRIBUTING.md), ktor (https://github.com/ktorio/ktor/blob/master/CONTRIBUTING.md) 等等.

如果你有一个库, 可能对其他开发者很有用, 请通过 feedback@kotlinlang.org (mailto:feedback@kotlinlang.org) 联系我们.

向文档贡献代码

如果你在 Kotlin 文档中发现了问题, 请 check out 文档的 GitHub 代码仓库 (https://github.com/JetBrains/kotlin-web-site/tree/master/docs/topics), 并向我们发送一个 pull request. 请遵守 关于风格和格式的指南 (https://docs.google.com/document/d/1mUuxK4xwzs3jtDGoJ5_zwYLaSEl13g_SuhODdFuh2Dc/edit?usp=sharing).

请和我们保持联系, 因为我们可能会有一些疑问, 并对你提交的修改留下评论. 否则, 我们不能将你贡献的代码合并到项目中.

创建教程或视频

如果你为 Kotlin 创建了教程或视频, 请通过 feedback@kotlinlang.org (mailto:feedback@kotlinlang.org) 分享给我们.

将文档翻译为其他语言

欢迎你将 Kotlin 文档翻译为你自己的语言, 并发布到你的网站. 但是, 我们不能将你的翻译存放到主代码仓库, 并发布到 kotlinlang.org (https://kotlinlang.org/).

这个网站是 Kotlin 语言的官方文档, 并且我们会确保这里的所有信息是正确的, 并且是最新的. 不幸的是, 我们不能审核其他语言的文档.

举办活动和演讲

如果已经或者在计划举办关于 Kotlin 的活动和演讲, 请填写 这个表格 (https://surveys.jetbrains.com/s3/Submit-a-Kotlin-Talk). 我们会将你的活动添加到 活动列表 (https://kotlinlang.org/docs/events.html).

KUG 指南

最终更新: 2024/09/10

Kotlin User Group, 简称 KUG, 是一个专注于 Kotlin 的开发社区, 你可以在这里与其他志趣相投的人分享你的 Kotlin 编程经验.

要成为 KUG, 你的开发社区需要满足每个 KUG 都必须的特定的条件. 包括:

- 提供 Kotlin 相关的内容, 主要的活动形式是定期会面交流.
- 主办定期的活动 (至少每 3 个月 1 次), 允许公开注册, 对参与活动没有限制.
- 由社区管理和组织, 不通过这些活动赚取金钱, 也不从成员和参加者获取任何其他商业利益.
- 保证遵守行为规范, 为来自任何背景任何经验的参加者提供一个友好的环境 (参见我们推荐的 行为规范 (https://confluence.jetbrains.com/display/ALL/JetBrains+Open+Source+and+Community+Code+of+Conduct)).

对 KUG 聚会的形式没有任何限制. 可以采用适合于开发社区的任何形式, 可以是演讲, 动手实验室, 讲座课程, 编程马拉松(hackathon), 或者非正式的啤酒聚会.

> 关于 Kotlin User Group 品牌资产, 请参见 Kotlin 品牌资产文档 ("Kotlin 用户组品牌资产" in "Kotlin 品牌资产").

如何运营一个 KUG?

- 为了提升团体的团结性, 防止沟通不畅, 我们推荐每个城市只限存在 1 个 KUG. 请查看 KUG 列表 (https://kotlinlang.org/community/user-groups), 确认在你的地区是否已有 KUG 存在.
- 使用官方的 KUG Logo 和品牌. 请查看 品牌指南 ("Kotlin 用户组品牌资产" in "Kotlin 品牌资产").
- 让你的用户组保持活跃. 定期举办聚会, 至少每 3 个月 1 次.
- 至少提前 2 周公布你的 KUG 聚会计划. 公告应该包含演讲题目和演讲者姓名的列表, 以及地点, 时间, 以及关于这个活动的其他任何重要信息.

- KUG 活动应该免费, 或者, 如果你需要支付活动经费, 参加费应该限制在 10 美元以内.
- 你的用户小组应该有一份对所有成员公开的行为规范.

如果你的社区满足了所有的条件, 并且遵循这些规范, 那么你可以 申请成为一个新的 KUG (https://surveys.jetbrains.com/s3/submit-a-local-kotlin-user-group).

如果有问题, 可以 联系我们 (mailto:kug@jetbrains.com)

JetBrains 对 KUG 的支持

活跃的 KUG, 至少每 3 个月主办 1 次聚会, 可以申请我们的社区支持计划, 包括:

- 官方的 KUG 品牌.
- Kotlin 网站的特别访问权限.
- JetBrains 产品的免费 License, 供聚会时抽奖发送.
- Kotlin 活动和促销时的优先支持.
- 帮助你的活动招募 Kotlin 演讲者.

JetBrains 对其它技术社区的支持

如果你组织了其它任何技术社区, 你也可以申请支持. 这样做之后, 你可以得到:

- JetBrains 产品的免费 License, 供聚会时抽奖发送.
- 关于 Kotlin 官方活动和促销的信息.
- Kotlin 贴纸.
- 帮助你的活动招募 Kotlin 演讲者.

Kotlin Night 指南

最终更新: 2024/09/10

Kotlin Night 是一种聚会, 包含 3 到 4 次关于 Kotlin 或相关技术的演讲.

关于 Kotlin Night 的品牌资产, 请参见 Kotlin 品牌资产文档 ("Kotlin Night 品牌资产" in "Kotlin 品牌资产").

活动指南

- 请使用我们提供的 品牌素材 ("Kotlin Night 品牌资产" in "Kotlin 品牌资产"). 所有活动和素材使用相同的风格, 将会有助于让 Kotlin Night 的体验保持统一.
- Kotlin Night 应该是免费的活动. 可以收取最低费用来应付必须的开支, 但应该是非营利的活动.
- 活动应该公开宣布, 并且允许任何人参加, 没有任何歧视性的限制.
- 如果你在活动之后将演讲内容发布到网上, 那么必须免费, 可供任何人访问, 而且不需要帐号注册.
- 不要求录像, 但建议如此, 而且录像视频也应该可以供任何人访问. 如果你决定录下演讲的视频, 我们建议提前最好计划, 以确保视频质量.
- 演讲内容应该主要是关于 Kotlin, 不应该集中于产品营销或推广.
- 活动也可以提供食品和饮料.

活动要求

JetBrains 非常愿意支持你的 Kotlin Night 活动. 由于我们希望所有的活动都能达到相同的高质量体验, 我们需要组织者满足一些基本的活动要求, 然后才能得到 JetBrains 的支持. 作为组织者, 你需要对活动的以下方面负责:

1. 关于活动的举办场地以及其它需求, 包括预约一个舒适的场地. 请确保做到:
    - 所有参加者都知道确切的活动日期, 地点, 开始时间, 以及活动的日程安排.
    - 有足够的空间, 如果你提供食品和饮料的话, 要足够所有人食用.

* 要提前计划好演讲者. 包括日程, 演讲主题, 演讲内容摘要, 以及演示用的必要设备.

2. 内容与演讲者

   * 完全可以从你的当地社区, 从你的临近国家, 甚至可以从全世界任何地方, 邀请演讲者. 你不必邀请 JetBrains 公司的代表或演讲者出席你的活动. 但是, 我们很乐意听取关于 Kotlin Night 活动的信息, 因此请通知我们.

3. 宣传和推广

   * 在你的活动开始至少 3 周以前宣布.
   * 要在公告中包含日程, 主题, 演讲内容摘要, 以及演讲者简介.
   * 在社交媒体上扩散消息.

4. 在活动之后, 将活动素材提供给 JetBrains

   * 我们很乐意在 kotlinlang.org (https://kotlinlang.org/community/talks.html) 宣布你的活动, 如果你能够向我们提供演示文稿和视频素材, 用于后续报道, 我们会非常感谢.

JetBrains 支持

JetBrains 提供以下支持:

* 使用 Kotlin Night Branding, 包括名称和 Logo
* 向演讲者提供小商品, 例如贴纸和 T-恤, 以及向出席者提供小纪念品
* 在 Kotlin Talks 页面宣布活动列表
* 如果需要的话, 可以帮助联系演讲者参加你的活动
* 如果可能的话, 可以帮助寻找活动场地(通过我们的关系公司, 等等.), 还可以帮助在当地的商业公司中寻找可能的合作伙伴

Kotlin 品牌资产

最终更新: 2024/09/10

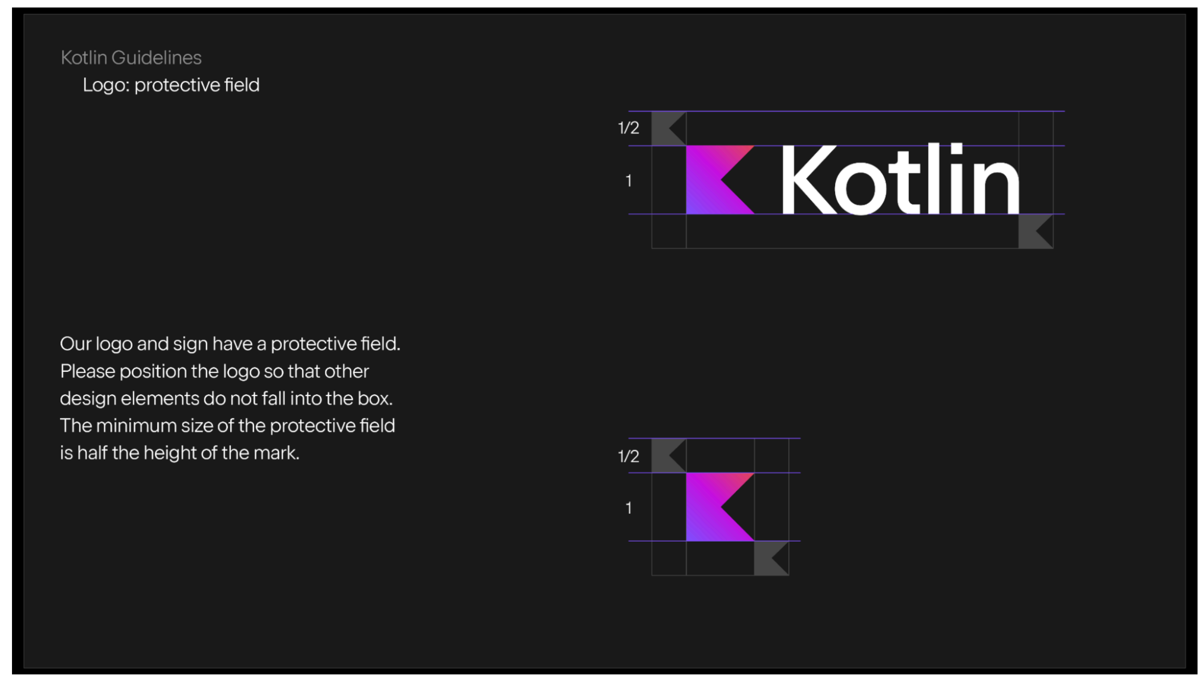
Kotlin Logo

我们的 Logo 包括一个标志和一个字体. 主 Logo 是全彩色版本, 大多数情况下应该使用这个版本.

下载所有版本
(https://resources.jetbrains.com/storage/products/kotlin/docs/kotlin_logos.zip)

Kotlin logo

我们的 Logo 和标志中包含保护区域. 请将 Logo 摆放在适当位置, 不要让其它图像元素遮挡保护区域. 保护区域的最小尺寸是标志高度的一半.



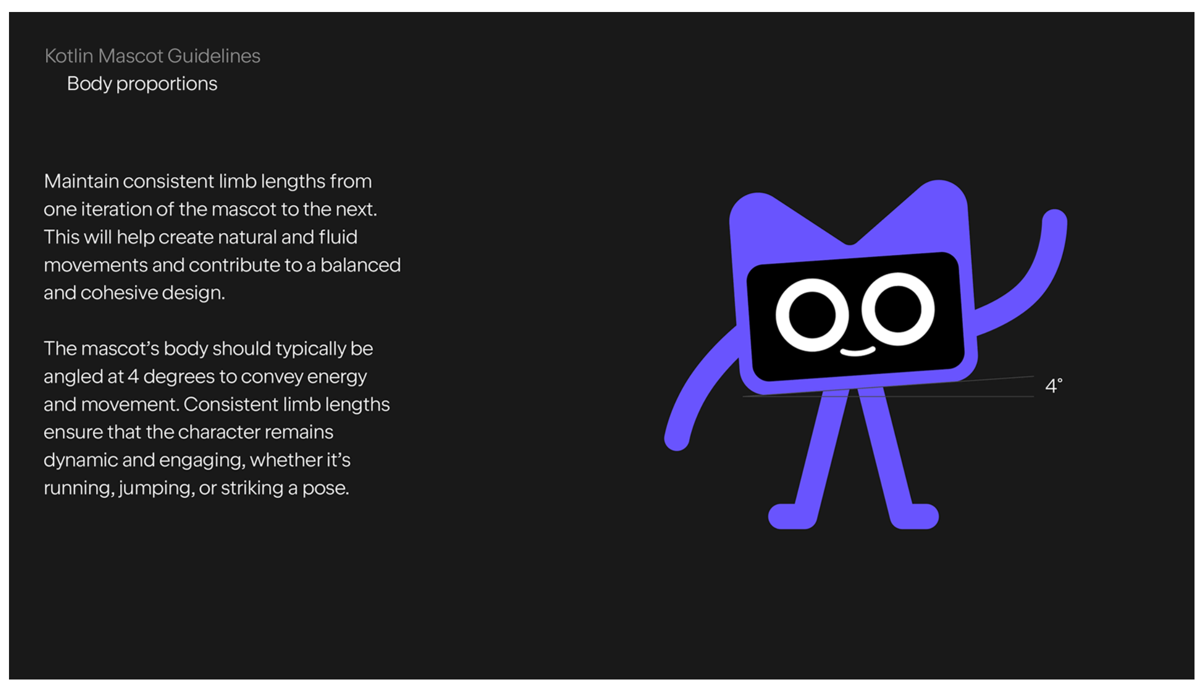
Kotlin Logo 的尺寸

使用 Logo 时请特别注意以下限制事项:

- 不要将标志与文字分离. 不要交换画面元素位置.
- 不要修改 Logo 的透明度.
- 不要突出 Logo 轮廓.
- 不要使用其他颜色重新绘制 Logo.
- 不要修改文字.
- 不要将 Logo 放在复杂的背景上. 不要将 Logo 放在亮色背景上.

Kotlin 吉祥物

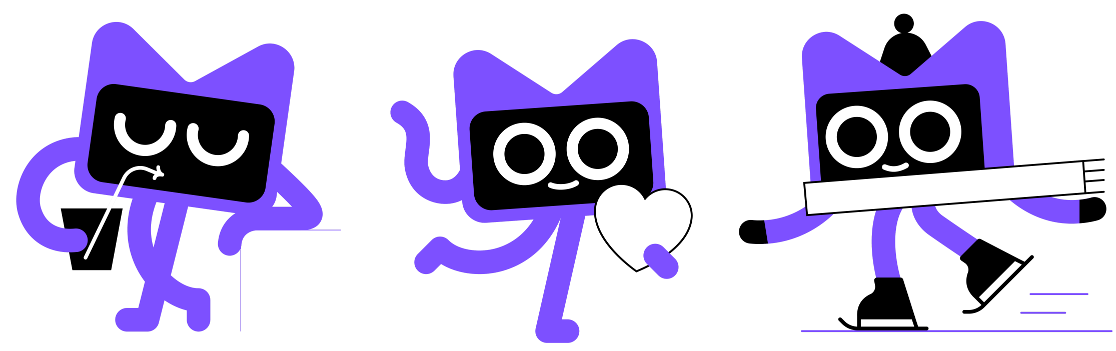
Kodee 是 Kotlin 重新构想的吉祥物. Kodee 不仅仅只是一个符号, 它也是你的好伙伴, 鼓励并激发你展现你的创造力. 在使用它时, 我们请求你遵循这些 简单的规则 (https://resources.jetbrains.com/storage/products/kotlin/docs/Kotlin_Mascot_Guidelines.pdf).



Kotlin 吉祥物 Kodee 的尺寸

你可以在你的数字化素材或打印素材中使用 Kodee. 为了这个用途, 我们准备了一系列的 Kotlin 吉祥物资产, 供你下载和使用.

下载全部资产 (https://resources.jetbrains.com/storage/products/kotlin/docs/kotlin_mascot_2.zip)



Kotlin mascot Kodee in action

Kotlin 用户组品牌资产

我们为 Kotlin 用户组提供特别定制的 Logo, 这个 Logo 易于辨认, 而且与 Kotlin 相关联.

- 官方的 Kotlin Logo 代表 Kotlin 语言本身. 它不应该使用在其它范围, 以免发生混淆. 它的相近的派生物也是如此.
- 用户组 Logo 也代表社区的观点和活动独立于 Kotlin 开发组.
- 你的观点不一定要与我们相同, 对于有创造力的、强大的社区, 我们认为这是最好的模式.

下载所有版本 (https://drive.google.com/drive/folders/0B3Zi34svOj1RZ2sxZExhblRJc1k)

用户组风格

自从 2017 年初 Kotlin 社区支持计划启动以来, 用户组的数量已经增长了很多倍, 每个月大约有 2 到 4 个新的用户组加入. 请在 **Kotlin 用户组** 小节查看用户组的完整列表, 找找你所在地区的用户组.

我们向新的 Kotlin 用户组提供用户组 Logo, 以及 Profile 图片.

品牌图片

我们这样做的主要原因有 2 点:

- 首先, 我们从社区收到很多请求, 要求特别的 Kotlin 风格品牌素材, 帮助他们标识自己是正式的专门用户组.
- 其次, 我们希望为用户组提供独特的风格, 以及社区内容, 以便明确区分哪些 Kotlin 相关的素材是来自官方开发组, 哪些是由社区自行创建的.

创建你的用户组的 Logo

要为你的用户组创建一个 Logo, 请执行以下步骤:

1. 将 Kotlin 用户组的 Logo 文件 (https://docs.google.com/drawings/d/1IcJp8Z2jAwEliXrHB-l9RNK_2LrqGTkNuPPtjrW1iIU/edit) 复制到你的 Google Drive (你需要登录到你的 Google 帐号).
2. 将 **Your City** 文字替换为你的用户组名称.
3. 下载图片, 将它用作用户组素材.

Belarusian Kotlin 用户组示例

Belarusian Kotlin 用户组 Profile 图片示例

你可以下载 一组图片 (https://drive.google.com/drive/folders/0B3Zi34svOj1RZ2sxZExhbIRJc1k), 包括矢量图, 以及用于社交网络的封面图片示例.

为你的用户组在不同的平台创建 Profile 图片

要创建你的用户组的 Profile 图片, 请执行以下步骤:

1. 将 Kotlin 用户组的 Profile 图片 file (https://docs.google.com/drawings/d/1buhwccmIlb7wFS0OIAub0WC4DIuSHRiDpjEQhB4tkPs/edit) 复制到你的 Google Drive (你需要登录到你的 Google 帐号).
2. 添加一个用户组位置的缩略名称(按照我们的默认示例, 最多 4 个首字母).
3. 下载图片, 在 Facebook, Twitter, 或任何其他平台, 将它用作你的 Profile.

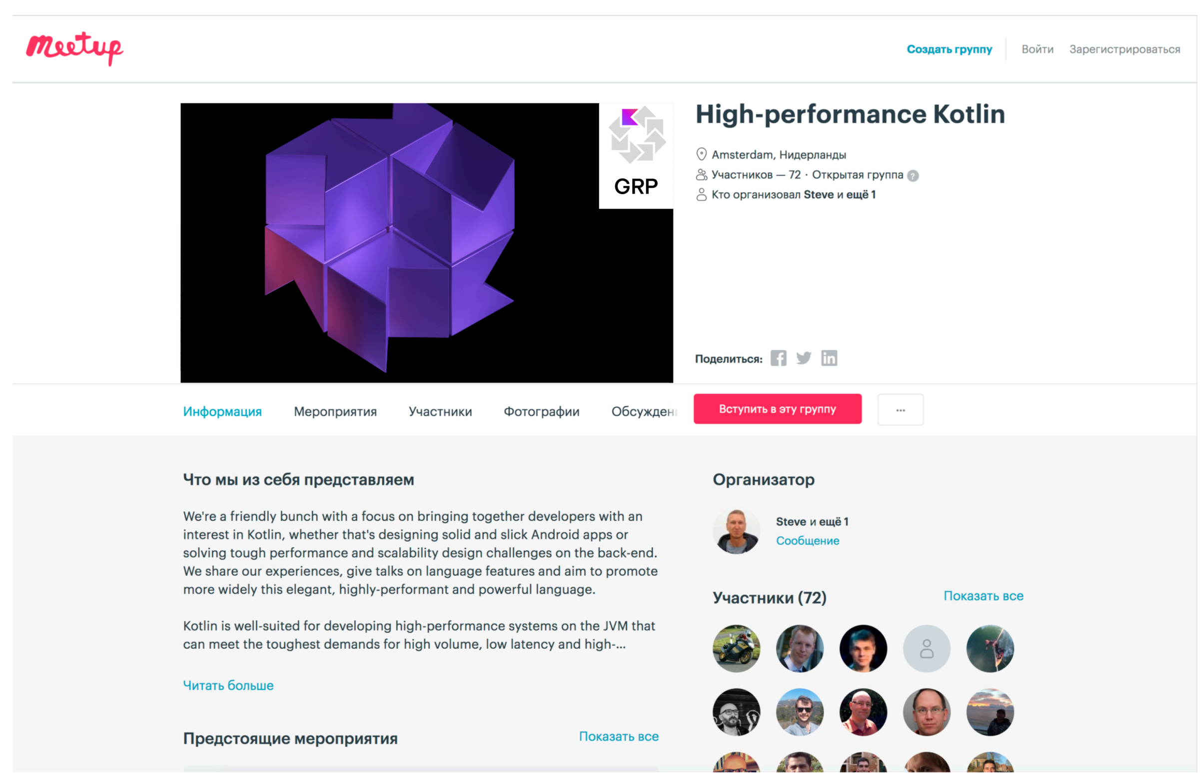
创建 meetup.com 封面照片

要为 meetup.com 创建一个带用户组 Logo 的封面照片, 请执行以下步骤:

1. 将 图片文件 (https://drive.google.com/file/d/1g_0PIf_do6vrXvy1R-Hx430vfV2CPVKN/view) 复制到你的 Google Drive (你需要登录到你的 Google 帐号).

2. 在 Logo 图片的右上角添加一个用户组位置的缩略名称. 如果你想要用自定义的图片来替换通常的模式, 请点击模式图片的背景, 选择 'Replace Image', 然后 'Upload from Computer', 或使用其他来源.

3. 下载图片, 将它用作你在 meetup.com (https://meetup.com) 上的 Profile.



用户组示例

Kotlin Night 品牌资产

JetBrains 为 Kotlin Night 活动提供品牌和素材. 我们的开发组会为活动的宣传准备数字资产, 并发送你的商品包, 包括贴纸和 T-恤. 请阅读相关文档, 看看我们能够如何让你的 Kotlin Night 活动更加有趣!

下载所有资产 (https://drive.google.com/drive/folders/1wTJ-PiO6VvbY6XdACGLsWZ_N8KHI0Nvr)

社交媒体

贴纸可以帖到任何对 Kotlin Night 活动需要的媒体. 你能够帖的地方全部都贴上吧. 很好玩!

Cover

Cover/Logo

Example of usage

Cover Social

品牌贴纸

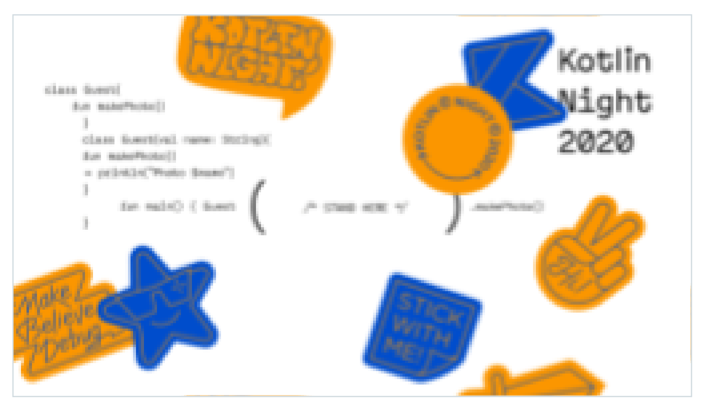
贴纸可以用作 Kotlin Night 的品牌资产. 你能够帖的地方全部都贴上吧. 很好玩!

Example of usage

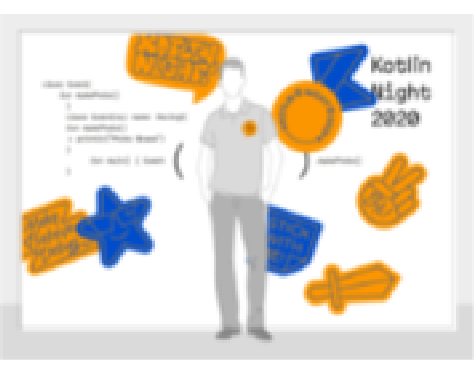
贴纸

Press-wall

你可以在你的宣传墙装饰上贴纸或难忘的活动照片.



Press-wall



Example of usage

宣传墙

贴纸徽章

可以把贴纸用作出席者的徽章,在活动上鼓励大家相互交流!

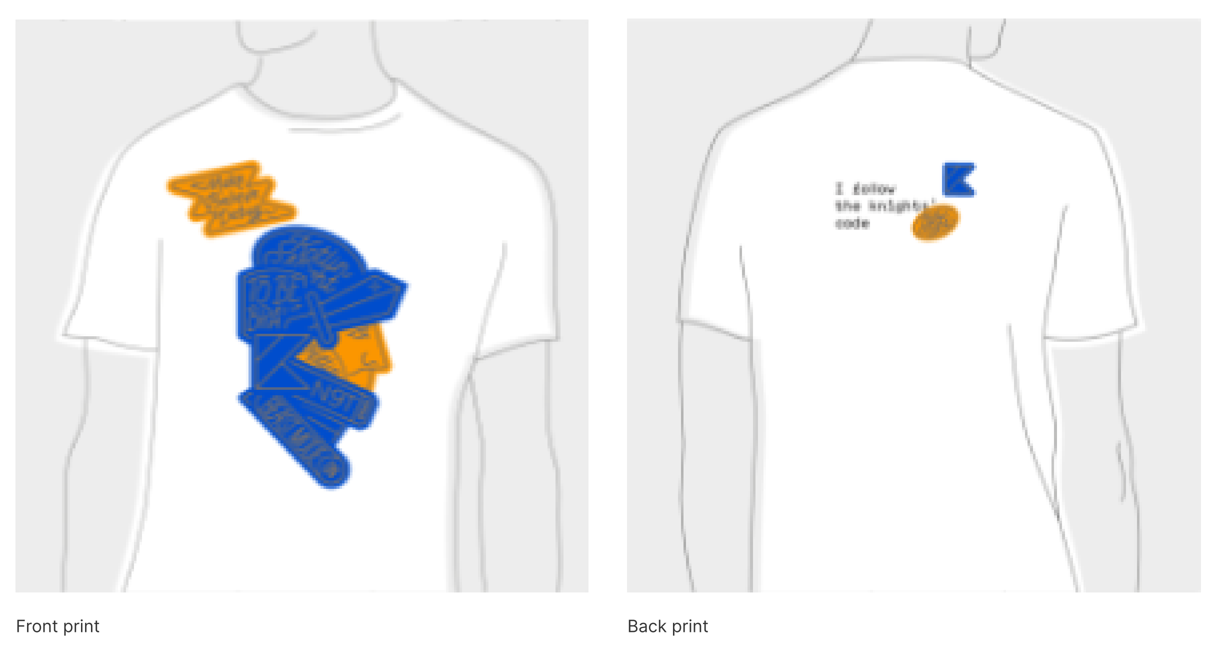
贴纸板

或者你也可以提供一面贴纸板, 让你的来宾能够贴上贴纸, 写下他们的感想, 反馈, 以及希望.

贴纸板

T-恤

活动的来宾能够在贴纸板帖上贴纸, 写下他们对聚会的感想. 对你来说意味着什么呢?



T-恤