



# Kotlin 语言参考文档

## 中文版 (2019 年 03 月, 第 8 次更新)

原作者: JetBrains 公司  
中文版译者: 李 颖(liying.cn.2010@gmail.com)

# 目录

前言 .....	5
关于翻译 .....	5
参考文档 .....	5
相关书籍 .....	5
概述 .....	7
在服务器端开发中使用 Kotlin .....	7
使用 Kotlin 进行 Android 开发 .....	9
Kotlin JavaScript 概述 .....	10
使用Kotlin/Native 进行原生(Native)程序开发 .....	11
用协程(Coroutine)实现异步程序开发 .....	13
跨平台程序开发 .....	14
Kotlin 1.1 的新增特性 .....	16
Kotlin 1.2 的新增特性 .....	28
Kotlin 1.3 的新增特性 .....	35
标准库 .....	40
工具 .....	41
入门 .....	43
基本语法 .....	43
惯用法 .....	54
编码规约 .....	59
基础 .....	73
基本类型 .....	73
包 .....	81
控制流: if, when, for, while .....	83
返回与跳转 .....	87
类与对象 .....	90
类与继承 .....	90
属性(Property)与域(Field) .....	97
接口 .....	101
可见度修饰符 .....	103
扩展 .....	105
数据类 .....	110
泛型 .....	113
嵌套类(Nested Class)与内部类(Inner Class) .....	119
枚举类(Enum Class) .....	120
对象表达式(Object Expression)与对象声明(Object Declaration) .....	122
内联类 .....	126
委托(Delegation) .....	130

委托属性(Delegated Property) .....	132
<b>函数与 Lambda 表达式</b> .....	138
函数 .....	138
高阶函数与 Lambda 表达式 .....	144
内联函数(Inline Function) .....	150
<b>跨平台程序开发</b> .....	154
与平台相关的声明 .....	154
使用 Gradle 编译跨平台项目 .....	156
<b>其他</b> .....	186
解构声明(Destructuring Declaration) .....	186
集合(Collection): List, Set, Map .....	188
值范围(Range) .....	190
类型检查与类型转换: ‘is’ 与 ‘as’ .....	193
this 表达式 .....	196
相等判断 .....	197
操作符重载(Operator overloading) .....	198
Null 值安全性 .....	202
异常(Exception) .....	205
注解(Annotation) .....	207
反射 .....	211
作用域函数(Scope Function) .....	216
类型安全的构建器(Type-Safe Builder) .....	226
实验性 API 标记(Experimental API Marker) .....	232
<b>核心库</b> .....	236
标准库(Standard Library) .....	236
测试库(kotlin.test) .....	237
<b>参考</b> .....	238
关键字(Keyword)与操作符(Operator) .....	238
语法 .....	242
<b>与 Java 的互操作性</b> .....	243
在 Kotlin 中调用 Java 代码 .....	243
在 Java 中调用 Kotlin .....	254
<b>JavaScript</b> .....	261
动态类型(Dynamic Type) .....	261
在 Kotlin 中调用 JavaScript .....	263
在 JavaScript 中调用 Kotlin .....	266
JavaScript 模块(Module) .....	268
JavaScript 中的反射功能 .....	271
JavaScript DCE .....	272
示例 .....	272

<b>原生(Native)程序开发</b>	274
Kotlin/Native 中的并发	274
Kotlin/Native 中的不变性(Immutability)	277
Kotlin/Native 库	278
高级问题	279
平台库	281
Kotlin/Native 的互操作性	282
Kotlin/Native 与 Swift/Objective-C 的交互能力	290
Kotlin/Native Gradle 插件	294
代码覆盖率(Code Coverage)	304
<b>协程(Coroutine)</b>	308
章节目录	308
其他参考文档	308
协程的基本概念	309
取消与超时	314
通道(Channel) (实验性功能)	319
挂起函数(Suspending Function)的组合	328
协程上下文与派发器(Dispatcher)	334
异常处理	344
监控	349
选择表达式(Select expression) (实验性功能)	353
共享的可变状态值与并发	361
<b>工具</b>	371
为 Kotlin 代码编写文档	371
在 Kotlin 中处理注解	374
使用 Gradle	378
使用 Maven	386
使用 Ant	393
Kotlin 与 OSGi	396
编译器插件	397
代码风格迁移指南	403
<b>Kotlin 的演化</b>	405
Kotlin 的演化	405
各部分组件的稳定性	409
Kotlin 1.3 兼容性指南	410
<b>FAQ</b>	419
FAQ	419
与 Java 语言的比较	422

# 前言

## 关于翻译

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/>

代码库: <https://github.com/JetBrains/kotlin-web-site>

中文翻译版

网址: <http://www.liying-cn.net/kotlin/docs/reference/>

代码库: <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

- 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

## 参考文档

关于 Kotlin 语言和 [标准库](#) 的完整参考文档。

## 从哪里开始

本参考文档的目标是帮助你在几个小时之内很容易地学习 Kotlin。请从 [基本语法](#) 开始, 然后继续阅读更高级的内容。阅读本文档时, 你可以在 [online IDE](#) 中试验文档中的例子程序。

当你大致了解 Kotlin 之后, 你可以前往 [Kotlin Koans](#), 这里是一些交互式编程练习题, 你可以试着自己解决这些问题。如果你不清楚如何解决某个练习题, 或者想要寻找更优雅的方式, 可以参考 [Kotlin 惯用法](#)。

## 离线阅读

你可以下载本参考文档的 [PDF 版](#)。

## 相关书籍

## Kotlin 实战(Kotlin in Action)

▫ [Kotlin 实战\(Kotlin in Action\)](#) 是一本关于 Kotlin 的书, 作者是 Dmitry Jemerov 和 Svetlana Isakova, 他们是 Kotlin 开发组的成员. 本书目前可以通过 MEAP 计划获取, 通过这种方式, 你可以在本书撰写过程中就提前阅读到各章内容, 并在本书最终完成之后得到完整版.

购买本书时使用 Coupon 号码 ‘39jemerov’, 可获得 39% 的折扣.

## 面向 Android 的 Kotlin 手册(Kotlin for Android Developers)

▫ [面向 Android 的 Kotlin 手册\(Kotlin for Android Developers\)](#) 由 Antonio Leiva 撰写, 本书向你展示如何使用 Kotlin 语言从零开始创建一个 Android 应用程序.

## 使用 Kotlin 开发现代化 Web 程序(Modern Web Development with Kotlin)

▫ [使用 Kotlin 开发现代化 Web 程序\(Modern Web Development with Kotlin\)](#) 由 Denis Kalinin 撰写, 介绍如何使用 Kotlin 进行 Web 开发. 本书包含了足够多的基本信息, 指导初学者起步, 但主要集中于 Kotlin 语言的实际应用. 具体来说, 本书会向你介绍 Web 应用程序开发的整个过程, 指导你创建一个包含大量技术内容的 Web 应用程序, 其中使用到许多流行的后端和前端技术.

## Kotlin 程序开发(Programming Kotlin)

▫ [Kotlin 程序开发\(Programming Kotlin\)](#) 由 Stephen Samuel 和 Stefan Bocutiu 撰写, 介绍如何在 JVM 环境中使用 Kotlin. 本书内容包含 Kotlin 语言的各个方面, 但主要集中介绍服务器端开发. 本书的目标读者是 Java 开发者, 帮助他们学习 Kotlin 语言, 尤其是帮助他们了解 Kotlin 相对于 Java 的进步之处.

# 概述

## 在服务器端开发中使用 Kotlin

Kotlin 非常适用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\(type-safe builder\)](#) 以及 [委托属性\(delegated property\)](#), 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\(coroutine\)](#) 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 Koans](#) 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

- [Spring](#) 从 5.0 版本开始, 使用 Kotlin 的语言特性实现了 [更加简洁的 API](#). [在线工程生成器](#) 可以帮助你使用 Kotlin 语言快速生成新的工程.
- [Vert.x](#), 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](#), 包含 [完整的文档](#).
- [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.
- [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.
- 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](#) 可以使 Kotlin 编译的 class文件符合 JPA 框架的要求.

## 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](#) 上发布 Kotlin 应用程序, 你可以参照 [Heroku 官方教程](#).

AWS Labs 提供了一个 [示例工程](#), 演示如何使用 Kotlin 来编写 [AWS 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](#) 和 [在 Google App Engine 上运行 Kotlin Spring 应用程序](#). 此外还有一篇 [向导式代码文档](#) 介绍如何发布 Kotlin Spring 应用程序.

## 使用 Kotlin 进行服务端开发的使用者

[Corda](#) 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](#), 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

- [使用 Http Servlet 创建 Web 应用程序](#) 以及 [使用 Spring Boot 创建 REST Web Service](#) 教程将向你演示如何使用 Kotlin 来创建并运行小型 Web 应用程序.
- 关于对 Kotlin 语言的更加深入介绍, 请参加本站的 [参考文档](#), 以及 [Kotlin Koans](#).



## 使用 Kotlin 进行 Android 开发

Kotlin 非常适合于开发 Android 应用程序, 它可以将一种现代化语言的所有优势带入 Android 平台, 同时又没有引入任何新的限制:

- **兼容性:** Kotlin 完全兼容于 JDK 6, 因此可以保证 Kotlin 应用程序能够在较旧的 Android 设备上运行, 不会产生问题. Android Studio 完全支持各种 Kotlin 开发工具, 并且兼容于 Android 构建系统.
- **性能:** 由于编译产生的字节码结构非常类似, 因此 Kotlin 应用程序可以达到等价的 Java 程序同样的运行速度. 由于 Kotlin 对内联函数 (inline function) 的支持, 使用 lambda 表达式的代码通常可以比等价的 Java 代码运行得更快.
- **互操作性:** Kotlin 100% 支持与 Java 的互操作, 因此在 Kotlin 应用程序中可以使用既有的 Android 库. 这种支持也包括注解处理 (annotation processing), 因此数据绑定, Dagger 都可以正常工作.
- **大小(Footprint):** Kotlin 的运行库非常紧凑, 而且还可以通过使用 ProGuard 来进一步缩减. 在一个 [真实的应用程序](#)中, Kotlin 运行库仅仅增加了几百个方法, 以及不到 100K 的 .apk 文件大小.
- **编译时间:** Kotlin 支持高效率的增量编译(incremental compilation), 因此, 虽然对于全新的编译(clean build)会存在一些额外的开销, 但[增量编译通常与 Java 同样快, 甚至更快](#).
- **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 Koans](#) 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.

## 使用 Kotlin 进行 Android 开发的案例研究

已有许多大公司成功地采用了 Kotlin, 其中一些公司分享了他们的经验:

- Pinterest 成功地 [将 Kotlin 引入到他们的应用程序中](#), 每月有 1.5 亿用户使用这些应用程序.
- Basecamp 的 Android 应用程序 [100% 使用 Kotlin 编写](#), 他们报告说程序员们更加快乐, 并且产品质量和开发速度都有了巨大的改善.
- Keepsafe 的 App Lock 应用程序也 [100% 转换为 Kotlin](#), 代码行数减少了 30%, 方法数量减少了 10%.

## Android 开发的相关工具

除 Kotlin 语言本身的特性之外, Kotlin 开发团队还提供了一系列用于 Android 的开发工具:

- [Kotlin Android Extensions](#) 是一个编译器扩展, 可以帮助你消除 `findViewById()` 调用, 替换为编译器生成的伪属性.
- [Anko](#) 是一个库, 针对 Android API 提供了一系列便于 Kotlin 使用的包装函数, 还提供了一种 DSL, 可以帮助你将在 .xml 布局文件替换为 Kotlin 代码.

## 下一步

- 下载并安装 [Android Studio 3.0](#), 其中已包含了对 Kotlin 的支持.
- 学习 [Android 与 Kotlin 入门](#) 教程, 创建你的第一个 Kotlin 应用程序.
- 关于对 Kotlin 语言的更加深入介绍, 请参加本站的 [参考文档](#), 以及 [Kotlin Koans](#).
- 还有一个很好的学习资源是 [针对 Android 开发者的 Kotlin 教程](#), 这本书会一步步地指导你使用 Kotlin 创建一个真实的 Android 应用程序.
- 查看 Google 的 [使用 Kotlin 编写的示例工程](#).

## Kotlin JavaScript 概述

Kotlin 提供了在 JavaScript 平台上运行的能力. 实现方法是将 Kotlin 代码转换为 JavaScript. 目前的实现针对 ECMAScript 5.1 标准, 但我们计划最终实现 ECMAScript 2015 标准.

当你选择编译目标为 JavaScript 时, 所有的 Kotlin 代码, 包括你的工程内的代码, 以及 Kotlin 自带的标准库的代码, 全部都会转换为 JavaScript. 但是不包括 JDK, 以及你使用到的任何 JVM, 或其他 Java 框架, 或其他库. Kotlin 代码之外的一切文件都会在编译期间被忽略掉.

Kotlin 编译器在编译时会尝试实现以下目标:

- 保证编译输出的 JavaScript 代码在尺寸方面最优化
- 保证编译输出的 JavaScript 代码是易读的
- 实现与现有的模块系统(module system)的互操作性
- 无论是编译到 JavaScript 还是JVM, (最大限度地)保证标准库的功能等同.

### 如何使用

在以下场景中, 你可能会希望将 Kotlin 代码编译为 JavaScript:

- 编写 Kotlin 代码, 用于客户端 JavaScript
  - **与 DOM 元素交互.** Kotlin 提供了一系列静态类型的接口, 可以与文档对象模型(Document Object Model)进行交互, 可以用来创建或修改 DOM 元素.
  - **与图形交互, 比如 WebGL.** 你可以编写 Kotlin 代码, 在 Web 页面中使用 WebGL 来创建图形元素 .
- 编写 Kotlin 代码, 用于服务器端 JavaScript
  - **使用服务器端技术.** 你可以使用 Kotlin 来与服务器端 JavaScript 交互, 比如 Node.js

Kotlin 可以与既有的第三方库和框架共同工作, 比如 jQuery 或 React. 要使用强类型 API 的第三方框架, 你可以使用 [ts2kt](#) 工具, 将 [Definitely Typed](#) 类型定义库中的 TypeScript 定义转换为 Kotlin. 或者, 你也可以使用 [动态类型\(Dynamic Type\)](#) 功能来访问任何没有强类型的框架.

JetBrains 专门针对 React 社区开发了几种工具: [React bindings](#), 以及 [Create React Kotlin App](#). 其中, Create React Kotlin App 可以帮助你使用 Kotlin 来创建 React 应用程序, 而不需要编译配置.

Kotlin 兼容于 CommonJS, AMD 以及 UMD, 因此可以直接[与不同的模块系统交互](#).

### 使用 Kotlin 进行 JavaScript 开发入门

关于如何使用 Kotlin 进行 JavaScript 开发, 请参见以下 [教程](#).

## 使用Kotlin/Native 进行原生(Native)程序开发



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. 它可以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:

- iOS (arm32, arm64, 以及 x86\_64 上的模拟器平台)
- MacOS (x86\_64)
- Android (arm32, arm64)
- Windows (mingw x86\_64)
- Linux (x86\_64, arm32, MIPS, MIPS 小尾序(little endian))
- WebAssembly (wasm32)

### 互操作性

Kotlin/Native 支持与原生代码的双向互操作. 一方面, 编译器会创建:

- 各种 [平台](#) 的可执行文件
- 静态库, 或 [动态](#) 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](#)

另一方面, 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](#)
- C, [Swift 和 Objective-C 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](#), Swift/Objective-C [框架](#), 图形引擎, 以及其他任何东西.

Kotlin/Native 的 [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

通过不同的 Kotlin 和 Kotlin/Native 编译目标, 我们支持 [跨平台项目](#). 这是一种在各种平台上共享 Kotlin 源代码的方式, 包括 Android, iOS 服务器端, JVM, 客户端, JavaScript, CSS, 以及原生平台.

[跨平台库](#) 为共通的 Kotlin 代码提供了必要的 API, 帮助我们以 Kotlin 代码的方式编写项目中共通的部分, 这些代码只需要编写一次, 然后就可以在所有的目标平台上共用.

### 如何开始



[教程和文档](#)

如果你是 Kotlin 新手, 请先阅读 [入门](#).

推荐的文档:

- [与 C 代码交互](#)

- [与 Swift/Objective-C 代码交互](#)

推荐的教程:

- [一个基本的 Kotlin/Native 应用程序](#)
- [跨平台项目: iOS 与 Android](#)
- [C 与 Kotlin/Native 之间的类型映射](#)
- [使用 Kotlin/Native 开发动态链接库](#)
- [使用 Kotlin/Native 开发 Apple 框架](#)



#### 示例项目

- [Kotlin/Native 源代码与示例](#)
- [KotlinConf App](#)
- [KotlinConf Spinner App](#)
- [Kotlin/Native 源代码与示例 \(.tgz\)](#)
- [Kotlin/Native 源代码与示例 \(.zip\)](#)

更多示例请参见 [GitHub](#).

## 用协程(Coroutine)实现异步程序开发

异步(Asynchronous)程序开发, 或者叫非阻塞(non-blocking)程序开发, 是一种相对来说比较新的程序开发方式. 无论我们开发的是服务器端程序, 桌面程序, 还是移动设备程序, 我们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.

为了解决这类问题, 有很多种不同的方法, 在 Kotlin 中, 我们采用一种灵活的方式, 在语言层我们提供了 [协程](#) 的支持, 然后将大部分具体功能交给运行库来实现, 这种方式非常符合 Kotlin 的理念.

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, 比如可以用来实现并发型模式(Concurrency), Actor模式, 等等.

### 如何开始



#### 教程和文档

如果你是 Kotlin 新手, 请先阅读 [入门](#).

精选文档:

- [协程简介](#)
- [基本概念](#)
- [频道\(Channel\)](#)
- [协程上下文与派发器\(Dispatcher\)](#)
- [共享的可变状态与并发](#)

推荐的教程:

- [使用 Kotlin 开发你的第一个协程](#)
- [异步程序开发](#)



#### 示例项目

- [kotlinx.coroutines 示例和源代码](#)
- [KotlinConf App](#)

更多示例请参见 [GitHub](#)

## 跨平台程序开发

⚠️ 跨平台项目是 Kotlin 1.2 和 1.3 版中新增的实验性特性. 本文档描述的所有语言特新和工具特性, 在未来的 Kotlin 版本中都有可能发生变更.

Kotlin 的一个明确目标就是希望能运行在所有平台上, 但是我们认为这个能力是为了实现一个更重要的目标: 在各种不同的平台上共用代码. 有了对各种平台的支持, 比如 JVM, Android, JavaScript, iOS, Linux, Windows, Mac, 甚至 STM32 这样的嵌入式系统, Kotlin 语言可以用来开发一个现代应用程序的任何组成部分. 而且, 由于可以重用源代码, 可以重用程序开发者的专业能力, 我们可以不必在不同的平台上重复实现同样的功能, 而将我们的精力用于实现更重要的功能, 这是无法估量的巨大益处.

### 基本工作原理

总体来说, 跨平台项目并不是简单地针对所有的平台编译所有的源代码. 这种模式存在明显的限制, 而且我们认识到, 现代应用程序需要使用它所运行的平台上的独有功能. Kotlin 并不限制你只能访问所有 API 的一个共通子集. 应用程序的各个部分之间都可以按照需要共用尽源代码, 同时也随时可以通过 Kotlin 语言提供的 [expect/actual 机制](#) 访问平台的 API.

下面的例子是一个极简单的日志框架, 演示如何共用代码, 以及在共通代码与平台逻辑之间如何交互. 共通代码大概是这样的:

```
enum class LogLevel {
    DEBUG, WARN, ERROR
}

internal expect fun writeLogMessage(message: String, logLevel: LogLevel)

fun logDebug(message: String) = writeLogMessage(message, LogLevel.DEBUG)
fun logWarn(message: String) = writeLogMessage(message, LogLevel.WARN)
fun logError(message: String) = writeLogMessage(message, LogLevel.ERROR)
```

└ 针对所有平台编译

└ 与平台相关的预期 API

└ 预期 API 可以在共通代码中使用

上面的代码预期各个平台的编译目标会为 `writeLogMessage` 函数提供平台相关的实现, 因此共通代码可以使用这个函数, 而不必考虑它具体如何实现.

在 JVM 平台, 你可以为这个函数提供实现, 将日志写到标准输出:

```
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    println("[${logLevel}]: $message")
}
```

在 JavaScript 中, 可以使用的是另一套完全不同的 API, 因此你可以在这个函数的实现中使用 console:

```
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    when (logLevel) {
        LogLevel.DEBUG -> console.log(message)
        LogLevel.WARN -> console.warn(message)
        LogLevel.ERROR -> console.error(message)
    }
}
```

在 1.3 版中, 我们重构了整个跨平台项目的模型. 我们用来描述跨平台的 Gradle 项目的 [新 DSL](#) 比以前更加灵活, 我们还在继续改进它, 以使项目的配置更加直观.

### 跨平台的库

共通代码可能依赖于一系列的库, 用来实现各种常见任务, 比如 [HTTP](#), [序列化](#), 以及 [管理协程](#). 而且, 所有的平台都提供了内容广泛的标准库.

你也可以编写你自己的库, 提供共通的 API, 并在各个平台上提供不同的实现.

## 用例

### Android — iOS

在不同的移动平台上共用代码, 是 Kotlin 跨平台项目的主要使用场景之一, 现在, 为了创建移动设备上的应用程序, 我们可以在 Android 和 iOS 平台上共用一部分代码, 比如业务逻辑, 网络连接, 等等.

See: [Multiplatform Project: iOS and Android](#)

### Client — Server

另一种使用场景是联网的应用程序, 一部分逻辑既可以用在服务器端, 也可以用在浏览器内运行的客户端, 因此代码共用也可以带来很大的益处. 这种情况的代码共用也可以通过 Kotlin 跨平台项目来实现.

[Ktor 框架](#) 适合于在联网的应用程序中创建异步的服务器端程序和客户端程序.

## 如何开始



### 教程和文档

如果你是 Kotlin 新手, 请先阅读 [入门](#).

推荐的文档:

- [设置跨平台项目](#)
- [与平台相关的声明](#)

推荐的教程:

- [跨平台的 Kotlin 库](#)
- [跨平台项目: iOS 与 Android](#)



### 示例项目

- [KotlinConf App](#)
- [KotlinConf Spinner App](#)

更多示例请参见 [GitHub](#)

# Kotlin 1.1 的新增特性

## 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\(suspending function\)](#); 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](#):

```
// 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>) {
    //sampleStart
    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())
    //sampleEnd
}
```

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](#) 以及 [教程](#).

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 [参考文档](#) 以及 [KEEP 文档](#).

## 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](#), 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](#) 以及 [KEEP 文档](#).

## 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 [参考文档](#), 或参见 [封闭类\(sealed class\)](#) 以及 [数据类\(data class\)](#) 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](#) 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 [参考文档](#) 以及 [KEEP 文档](#).

### 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](#) 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](#).

### 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](#).

### 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](#) 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 [参考文档](#) 以及 [KEEP 文档](#).

## 局部的委托属性

你现在可以对局部变量使用 [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](#).

## 委托属性绑定的拦截

对于 [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, 如果我們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](#).

### 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](#) 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](#):

```

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 [参考文档](#) 以及 [KEEP 文档](#).

### 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](#).

## 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")
}
```

## groupBy()

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.groupBy { 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 'groupBy' 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](#) (JSR-223)。The API allows to evaluate snippets of code at runtime:

```
val engine = ScriptEngineManager().getEngineByExtension("kts")!!
engine.eval("val x = 3")
println(engine.eval("x + 2")) // 输出结果为: 5
```

[这里](#) 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](#), `kotlin-reflect.jar` 库中的扩展函数和扩展属性已被移动到 `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 1.2 的新增特性

### 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 **预期声明与实际声明(expected and actual declarations)** 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("multi-platform 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 [相关文档](#).

### 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 = *intArrayOf(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

对于一个平台数据类型(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: fix for platform null escaping through an inlined extension receiver

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 JavaScript 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](#), 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 1.3 的新增特性

### 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](#) 状态. 请参见新的 [协程概述](#) 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](#).

### 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](#).
- [依赖项配置](#) 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](#).

更多详细信息, 请参见 [跨平台程序开发文档](#).

### 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](#).

## 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, 请参见 [这里](#).

## 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

 内联类从 Kotlin 1.3 开始可用, 目前还处于 *实验性* 阶段. 详情请参见 [参考文档](#).

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, 也不能更好:

```
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](#).

## 无符号整数

🚧 无符号整数从 Kotlin 1.3 开始可用, 目前还处于 实验性阶段. 详情请参见 [参考文档](#).

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](#) for details.

## @JvmDefault 注解

⚠️ @JvmDefault 从 Kotlin 1.3 开始可用, 目前还处于 实验性 阶段. 详情请参见 [参考文档](#).

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](#).

## 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の解决方案, 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/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
}
```

## isEmpty 和 ifBlank 函数

对于集合(Collection), Map, 对象数组, 字符序列, 以及值序列(sequence), 现在有了 `isEmpty` 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 `isEmpty` 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
}
```

## 在反射中使用密封类

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.

## 小变更

- `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` / `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 IDE 中支持 [推荐的代码风格](#). 关于代码迁移的方法, 请参见 [参考文档](#).


## kotlinx.serialization

[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](#).

 注意, 虽然现在 `kotlinx.serialization` 与 Kotlin 编译器一起发布, 但它仍然是一个实验性功能.

## 脚本 API 升级

 注意, 脚本是一个实验性功能, 也就是说, 目前提供的 API 不保证任何兼容性.

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](#).

## 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](#).

# 入门

## 基本语法

### 定义包

包的定义应该在源代码文件的最上方:

```
package my.demo

import java.util.*

// ...
```

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](#).

### 定义函数

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](#).

## 定义变量

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")
}
```

顶级(top level) 变量:

```
//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\(Property\)与域\(Field\)](#).

## 注释

与 Java 和 JavaScript 一样, Kotlin 支持行末注释, 也支持块注释.

```
// 这是一条行末注释

/* 这是一条块注释
   可以包含多行内容. */
```

与 Java 不同, Kotlin 中的块注释允许嵌套.

关于文档注释的语法, 详情请参见 [Kotlin 代码中的文档](#).

## 使用字符串模板

```
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)
}
```

参见 [字符串模板](#).

## 使用条件表达式

```
//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)}")
}
```

以表达式形式使用 `if`:

```
//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](#).

## 使用可为 `null` 的值, 以及检查 `null`

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`.

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("either '$arg1' or '$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](#).

## 使用类型检查和自动类型转换

`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("$obj' string length is ${getStringLength(obj) ?: "... err, not a string"}")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf())
}
```

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("$obj' string length is ${getStringLength(obj) ?: "... err, not a string"}")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf())
}
```

甚至可以



```
//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("$obj' string length is ${getStringLength(obj)} ?: "... err, is empty or not a string at all")
    }
    printLength("Incomprehensibilities")
    printLength("")
    printLength(1000)
}
```

参见 [类](#) 和 [类型转换](#).

## 使用 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](#).

## 使用 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](#).

## 使用 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 expression](#).

## 使用值范围(Range)

使用 `in` 运算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](#).

## 使用集合(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, 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.toUpperCase() }
        .forEach { println(it) }
    //sampleEnd
}

```

参见 [高阶函数与 Lambda 表达式](#).

创建基本的类, 及其实例:

```

fun main() {
//sampleStart
    val rectangle = Rectangle(5.0, 2.0) // 不需要 'new' 关键字
    val triangle = Triangle(3.0, 4.0, 5.0)
//sampleEnd
    println("Area of rectangle is ${rectangle.calculateArea()}, its perimeter is ${rectangle.perimeter}")
    println("Area of triangle is ${triangle.calculateArea()}, its perimeter is ${triangle.perimeter}")
}

abstract class Shape(val sides: List<Double>) {
    val perimeter: Double get() = sides.sum()
    abstract fun calculateArea(): Double
}

interface RectangleProperties {
    val isSquare: Boolean
}

class Rectangle(
    var height: Double,
    var length: Double
): Shape(listOf(height, length, height, length)), RectangleProperties {
    override val isSquare: Boolean get() = length == height
    override fun calculateArea(): Double = height * length
}

class Triangle(
    var sideA: Double,
    var sideB: Double,
    var sideC: Double
): Shape(listOf(sideA, sideB, sideC)) {
    override fun calculateArea(): Double {
        val s = perimeter / 2
        return Math.sqrt(s * (s - sideA) * (s - sideB) * (s - sideC))
    }
}

```

详情请参见 [类](#) 以及 [对象和实例](#).

## 惯用法

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](#))

### 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 }
```

甚至可以写得 shorter:

```
val positives = list.filter { it > 0 }
```

### 在字符串内插入变量值

```
println("Name $name")
```

### 类型实例检查

```
when (x) {  
    is Foo -> ...  
    is Bar -> ...  
    else -> ...  
}
```

使用成对变量来遍历 Map, 这种语法也可以用来遍历 Pair 组成的 List

```
for ((k, v) in map) {  
    println("$k -> $v")  
}
```

上例中的 `k`, `v` 可以使用任何的变量名.

### 使用数值范围

```
for (i in 1..100) { ... } // 闭区间: 包括 100
for (i in 1 until 100) { ... } // 半开(half-open)区间: 不包括 100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

### 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

```
println(map["key"])
map["key"] = value
```

### 延迟计算(Lazy)属性

```
val p: String by lazy {
    // compute the string
}
```

### 扩展函数

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

### 创建单例(Singleton)

```
object Resource {
    val name = "Name"
}
```

### If not null 的简写表达方式

```
val files = File("Test").listFiles()

println(files?.size)
```

### If not null ... else 的简写表达方式

```
val files = File("Test").listFiles()

println(files?.size ?: "empty")
```

### 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() ?: ""
```

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) } ?: defaultValueIfValueIsNull
```

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 语句)

```
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()
}
```

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())
}
```

对于需要泛型类型信息的函数, 可以使用这样的简便形式

```
// 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)
```

使用可为 null 的布尔值

```
val b: Boolean? = ...
if (b == true) {
    ...
} else {
    // `b` 为 false 或为 null
}
```

交换两个变量的值

```
var a = 1
var b = 2
a = b.also { b = a }
```

## 编码规约

本章介绍 Kotlin 语言目前的编码风格。

- [源代码组织](#)
- [命名规约](#)
- [代码格式化](#)
- [文档注释](#)
- [避免冗余的结构](#)
- [各种语言特性的惯用法](#)
- [针对库开发的编码规约](#)

### 在 IDE 中应用本编码规约

如果要按照本编码规约来配置 IntelliJ 的代码格式化规则, 请安装 Kotlin plugin 1.2.20 或更高版本, 进入菜单 Settings | Editor | Code Style | Kotlin, 点击右上方的 “Set from...” 链接, 然后在菜单中选择 “Predefined style / Kotlin style guide”。

如果要验证你的代码是否已经按照本编码规约格式化完成, 可以进入代码检查的设置, 然后启用 “Kotlin | Style issues | File is not formatted according to project settings” 检查项目。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

在混合语言项目中, 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)。

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.foo.bar” 包下的文件应该保存在源代码根目录下的 “foo/bar” 子目录下)。

#### 源代码文件名

如果 Kotlin 源代码文件只包含单个类 (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。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, 并将首字母大写 (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”。

#### 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, 如果与到这个类的所有使用者都有关系, 那么应该将它们放在与类本身相同的源代码文件内。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 Foo 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

通常来说, 类的内容按以下顺序排列:

- 属性声明, 以及初始化代码端
- 次构造器
- 方法声明
- 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則.

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 遵循 Java 的命名规约. 具体来说:

包名称总是使用小写字母, 并且不使用下划线( `org.example.myproject` ). 通常不鼓励使用多个单词的名称, 但如果确实需要, 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 = ...
```

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类相同的名称:

```
abstract class Foo { ... }

class FooImpl : Foo { ... }

fun Foo(): Foo { return FooImpl(...) }
```

### 测试方法名称

在测试代码中 (而且仅仅在测试代码中), 可以使用用反引号括起的, 带空格的方法名. (注意, 这样的方法名目前在 Android 运行环境下不支持.) 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, 那么应该使用通常的驼峰式大小写名称:

```
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( `enum class Color { RED, GREEN }` ), 也可以使用大写字母开头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` 等等).

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

大多数情况下, Kotlin 遵循 Java 的编码规范.

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) {
        // ...
    }
}
```

(注意: 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: `// This is a comment`

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 { ... }
}
```

## 类头部格式化

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. 如果我们用到了类的继承, 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).

原因: 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

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
companion
inline
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`).

### 注解格式化

注解通常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
```

无参数的单个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
```

### 函数格式化

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).

原因: 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
```

## 表达式体格式化

如果函数体表达式太长, 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. 表达式函数体放在下一行, 缩进 4 个空格.

```
fun f(x: String) =
    x.length
```

## 属性格式化

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(file)
```

## 控制流语句的格式化

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)
}
```

原因: 这是为了将条件部分与主体部分对其, 并且分隔清楚

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) {
    // body
} else {
    // else part
}

try {
    // body
} finally {
    // cleanup
}
```

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
}
```

## 方法调用的格式化

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 表达式格式化

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)  
}
```

## 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) = ...
```

// 应该这样:

```
/**
 * 对于给定的 [number], 返回其绝对值.
 */
fun abs(number: 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

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: arrayOf() 的返回类型为 ArrayList<T>, 这是一个可变的集合类型
val allowedValues = arrayOf("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>
```

## 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最为最后一条语句.

## 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)
```

## 使用条件语句

尽量使用 `try`, `if` 以及 `when` 的表达式形式. 示例:

```
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 vs when

对于二元的条件分支, 尽量使用 `if` 而不是 `when` . 比如, 下面的代码是不好的:

```
when (x) {
    null -> ...
    else -> ...
}
```

应该改用 `if (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.

### 使用循环

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

如果数值范围是一个开区间(不包含其末尾元素), 那么应该使用 `until` 函数进行循环:

```
for (i in 0..n - 1) { ... } // 不好的风格
for (i in 0 until n) { ... } // 比较好的风格
```

### 使用字符串

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:

```
assertEquals(
    """
    Foo
    Bar
    """.trimIndent(),
    value
)

val a = """if(a > 1) {
    |   return a
    |}""".trimMargin()
```

## 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.

当以下条件成立时, 应该选择使用只读属性, 而不是使用函数:

- 不会抛出异常
- 计算过程消耗的资源不多(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

## 使用扩展函数

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.

## 使用中缀函数

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中缀函数. 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中缀方法.

## 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.

示例:

```
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)  
}
```

### 使用范围函数 `apply`/`with`/`run`/`also`/`let`

Kotlin 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, 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\(Scope Function\)](#).

### 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 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



# 基础

## 基本类型

在 Kotlin 中, 一切都是对象, 这就意味着, 我们可以对任何变量访问它的成员函数和属性. 有些数据类型有着特殊的内部表现形式, 比如, 数值, 字符, 布尔值在运行时可以使用 Java 的基本类型(Primitive Type)来表达, 但对于使用者来说, 它们就和通常的类一样. 本章我们将介绍 Kotlin 中使用的基本类型: 数值, 字符, 布尔值, 数组, 以及字符串.

### 数值

Kotlin 处理数值的方式与 Java 类似, 但并不完全相同. 比如, 对数值不会隐式地扩大其值范围, 而且在某些情况下, 数值型的字面值(literal)也与 Java 存在轻微的不同.

Kotlin 提供了以下内建类型来表达数值(与 Java 类似):

类型	位宽度
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

注意, 字符在 Kotlin 中不是数值类型.

### 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`

注意: 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`

在数字字面值中使用下划线 (从 Kotlin 1.1 开始支持)

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

```

## 内部表达

在 Java 平台中, 数值的物理存储使用 JVM 的基本类型来实现, 但当我们需表达一个可为 null 的数值引用时(比如, `Int?`), 或者涉及到泛型时, 我们就不能使用基本类型了. 这种情况下数值会被装箱(box)为数值对象.

注意, 数值对象的装箱(box)并不一定会保持对象的同一性(identity):

```

fun main() {
//sampleStart
    val a: Int = 10000
    println(a === a) // 打印结果为 'true'
    val boxedA: Int? = a
    val anotherBoxedA: Int? = a
    println(boxedA === anotherBoxedA) // !!!打印结果为 'false'!!!
//sampleEnd
}

```

但是, 装箱(box)会保持对象内容相等(equality):

```

fun main() {
//sampleStart
    val a: Int = 10000
    println(a == a) // 打印结果为 'true'
    val boxedA: Int? = a
    val anotherBoxedA: Int? = a
    println(boxedA == anotherBoxedA) // 打印结果为 'true'
//sampleEnd
}

```

## 显式类型转换

由于数据类型内部表达方式的差异, 较小的数据类型不会被看作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), 而且还在所有发生隐式类型转换的地方, 保持内容相等(equality)的能力也静悄悄地消失了.

由于存在以上问题, Kotlin 不会将较小的数据类型隐式地转换为较大的数据类型. 也就是说, 如果不进行显式类型转换, 我们就不能将一个 `Byte` 类型值赋给一个 `Int` 类型的变量.

```

fun main() {
//sampleStart
    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 // 这是错误的
//sampleEnd
}

```

我们可以使用显式类型转换, 来将数值变为更大的类型

```
fun main() {  
    val b: Byte = 1  
    //sampleStart  
    val i: Int = b.toInt() // 这是 OK 的: 我们明确地扩大了数值的类型  
    print(i)  
    //sampleEnd  
}
```

所有的数值类型都支持以下类型转换方法:

- toByte(): Byte
- toShort(): Short
- toInt(): Int
- toLong(): Long
- toFloat(): Float
- toDouble(): Double
- toChar(): Char

Kotlin 语言中缺少了隐式类型转换的能力, 这个问题其实很少会引起使用者的注意, 因为类型可以通过代码上下文自动推断出来, 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
```

## 运算符(Operation)

Kotlin 对数值类型支持标准的数学运算符(operation), 这些运算符都被定义为相应的数值类上的成员函数(但编译器会把对类成员函数的调用优化为对应的运算指令). 参见 [运算符重载](#).

对于位运算符, 没有使用特别的字符来表示, 而只是有名称的普通函数, 但调用这些函数时, 可以将函数名放在运算数的中间(即中缀表示法), 比如:

```
val x = (1 shl 2) and 0x000FF000
```

以下是位运算符的完整列表(只适用于 Int 类型和 Long 类型):

- shl(bits) - 带符号左移 (等于 Java 的 <<)
- shr(bits) - 带符号右移 (等于 Java 的 >>)
- ushr(bits) - 无符号右移 (等于 Java 的 >>>)
- 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](#)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.

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(比如, `Any`, `Comparable<...>`, 或者类型参数), 此时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, 这些方法不符合 IEEE 754 浮点数值运算标准, 因此:

- `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`

## 字符

字符使用 `Char` 类型表达. 字符不能直接当作数值使用

```
fun check(c: Char) {
    if (c == 1) { // 错误: 类型不兼容
        // ...
    }
}
```

字符的字面值(literal)使用单引号表达: `'1'`. 特殊字符使用反斜线转义表达. Kotlin 支持的转义字符包括: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` 以及 `\$`. 其他任何字符, 都可以使用 Unicode 转义表达方式: `'\uFF00'`.

我们可以将字符显式地转换为 `Int` 型数值:

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() // 显式转换为数值
}
```

与数值型一样, 当需要一个可为 `null` 的字符引用时, 字符会被装箱(box)为对象. 装箱操作不保持对象的同一性(identity).

## 布尔值

`Boolean` 类型用来表示布尔值, 有两个可能的值: `true` 和 `false`.

当需要一个可为 `null` 的布尔值引用时, 布尔值也会被装箱(box).

布尔值的内建运算符有

- `||` - 或运算(会进行短路计算)
- `&&` - 与运算(会进行短路计算)
- `!` - 非运算

## 数组

Kotlin 中的数组通过 `Array` 类表达, 这个类拥有 `get` 和 `set` 函数(这些函数通过运算符重载转换为 `[]` 运算符), 此外还有 `size` 属性, 以及其他一些有用的成员函数:

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

要创建一个数组, 我们可以使用库函数 `arrayOf()`, 并向这个函数传递一些参数来指定数组元素的值, 所以 `arrayOf(1, 2, 3)` 将创建一个数组, 其中的元素为 `[1, 2, 3]`. 或者, 也可以使用库函数 `arrayOfNulls()` 来创建一个指定长度的数组, 其中的元素全部为 `null` 值.

另一种方案是使用 `Array` 构造函数, 第一个参数为数组大小, 第二个参数是另一个函数, 这个函数接受数组元素下标作为自己的输入参数, 然后返回这个下标对应的数组元素的初始值:

```
fun main() {
    //sampleStart
    // 创建一个 Array<String>, 其中的元素为 ["0", "1", "4", "9", "16"]
    val asc = Array(5, { i -> (i * i).toString() })
    asc.forEach { println(it) }
    //sampleEnd
}
```


我们在前面提到过, `[]` 运算符可以用来调用数组的成员函数 `get()` 和 `set()`.

注意: 与 Java 不同, Kotlin 中数组的类型是不可变的. 所以 Kotlin 不允许将一个 `Array<String>` 赋值给一个 `Array<Any>`, 否则可能会导致运行时错误(但你可以使用 `Array<out Any>`, 参见 [类型投射](#)).

Kotlin 中也有专门的类来表达基本数据类型的数组: `ByteArray`, `ShortArray`, `IntArray` 等等, 这些数组可以避免数值对象装箱带来的性能损耗. 这些类与 `Array` 类之间不存在继承关系, 但它们的方法和属性是一致的. 各个基本数据类型的数组类都有对应的工厂函数:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```


## 无符号整数

 无符号类型只在 Kotlin 1.3 以上版本有效, 目前还处于 *试验性阶段*. 详情请参见 [下文](#)

针对无符号整数, Kotlin 引入了以下数据类型:

- `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.UIInt`: 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  $2^{64} - 1$

无符号整数支持有符号整数的大多数运算符.

 注意, 将无符号类型变为有符号类型 (或者反过来) 是一种 *二进制不兼容的变换*

无符号整数的实现使用到了另一种试验性功能, 名为 [内嵌类](#).

### 为无符号整数服务的专用类

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, 特定的数组类专用于表达特定的无符号整数类型:

- `kotlin.UByteArray`: 无符号 byte 构成的数组

- `kotlin.UShortArray` : 无符号 short 构成的数组
- `kotlin.UIntArray` : 无符号 int 构成的数组
- `kotlin.ULongArray` : 无符号 long 构成的数组

与有符号的整数数组类一样, 这些无符号整数的数组类提供了与 `Array` 类近似的 API, 并且不会产生数值对象装箱带来的性能损耗.

此外, 还提供了 `kotlin.ranges.UIntRange`, `kotlin.ranges.UIntProgression`, `kotlin.ranges.ULongRange`, `kotlin.ranges.ULongProgression` 类, 来支持 `UInt` 和 `ULong` 类型的 [值范围与数列](#) 功能.

### 无符号整数的字面值(literal)

为了便利无符号整数的使用, 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/Long` 的标记方式类似):

- `u` 和 `U` 后缀将一个字面值标记为无符号整数. 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, 那么将根据整数值的大小来选择使用 `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` 后缀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
```

### 无符号整数功能目前的状况

无符号整数类型功能的设计还处于实验性阶段, 也就是说, 这个功能目前还在快速变化, 并且不保证兼容性. 在 Kotlin 1.3 以上版本中使用无符号整数运算时, 编译器会提示警告信息, 表示这项功能还处于实验性阶段. 如果要去掉这些警告, 你需要明确地表示自己确定要使用无符号整数的实验性功能.

具体的做法有两种方式: 一种会把你的 API 也变成实验性 API, 另一种不会.

- 如果愿意把你的 API 也变成实验性 API, 可以对使用无符号整数的 API 声明添加 `@ExperimentalUnsignedTypes` 注解, 或者对编译器添加 `-Xexperimental=kotlin.ExperimentalUnsignedTypes` 选项 (注意, 这个编译选项会将被编译的模块中的 *所有* API 声明变成实验性 API)
- 如果不愿意把你的 API 变成实验性 API, 可以对你的 API 声明使用 `@UseExperimental(kotlin.ExperimentalUnsignedTypes::class)` 注解, 或者对编译器添加 `-Xuse-experimental=kotlin.ExperimentalUnsignedTypes` 选项

你的 API 使用者是否需要明确地表示自己确定要使用你的实验性 API, 这个选择由你来决定. 但是请时刻记得, 无符号整数目前还是实验性功能, 因此由于 Kotlin 语言未来版本的变化, 使用无符号整数的 API 可能会突然崩溃.

关于更多技术细节, 请参见实验性 API 的 [KEEP](#).

### 更深入地讨论

关于更多技术细节和更深入的讨论, 请参见 [关于 Kotlin 语言支持无符号数据类型的建议](#).

## 字符串

字符串由 `String` 类型表示. 字符串的内容是不可变的. 字符串中的元素是字符, 可以通过下标操作符来访问: `s[i]`. 可以使用 `for` 循环来遍历字符串:

```
fun main() {
    val str = "abcd"
    //sampleStart
    for (c in str) {
        println(c)
    }
    //sampleEnd
}
```

你可以使用 `+` 操作符来拼接字符串. 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")
    //sampleEnd
}
```

注意, 大多数情况下, 字符串的拼接处理应该使用 [字符串模板](#) 或 原生字符串(raw string).

### 字符串的字面值(literal)

Kotlin 中存在两种字符串面值: 一种称为转义字符串(escaped string), 其中可以包含转义字符, 另一种成为原生字符串(raw string), 其内容可以包含换行符和任意文本. 转义字符串(escaped string) 与 Java 的字符串非常类似:

```
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, 请参见上文的 [字符](#) 小节.

原生字符串(raw string)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)
    """
```

你可以使用 [trimMargin\(\)](#) 函数来删除字符串的前导空白(leading whitespace):

```
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 `|` 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

字符串内可以包含模板表达式, 也就是说, 可以包含一小段代码, 这段代码会被执行, 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简单的变量名:

```
fun main() {  
    //sampleStart  
    val i = 10  
    println("i = $i") // 打印结果为 "i = 10"  
    //sampleEnd  
}
```

\$ 符号之后也可以是任意的表达式, 由大括号括起:

```
fun main() {  
    //sampleStart  
    val s = "abc"  
    println("$s.length is ${s.length}") // 打印结果为 "abc.length is 3"  
    //sampleEnd  
}
```

原生字符串(raw string)和转义字符串(escaped string)内都支持模板. 由于原生字符串无法使用反斜线转义表达方式, 如果你想在字符串内表示 \$ 字符本身, 可以使用以下语法:

```
val price = ""  
${'$'}9.99  
""
```



## 包

源代码文件的开始部分可以是包声明:

```
package foo.bar

fun baz() { ... }
class Goo { ... }

// ...
```

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, `baz()` 函数的完整名称将是 `foo.bar.baz`, `Goo` 类的完整名称将是 `foo.bar.Goo`.

如果没有指定包, 那么源代码文件中的内容将属于 “default” 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.\\*](#)
- [kotlin.annotation.\\*](#)
- [kotlin.collections.\\*](#)
- [kotlin.comparisons.\\*](#) (since 1.1)
- [kotlin.io.\\*](#)
- [kotlin.ranges.\\*](#)
- [kotlin.sequences.\\*](#)
- [kotlin.text.\\*](#)

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.\\*](#)
- JavaScript 平台:
  - [kotlin.js.\\*](#)

## 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の `import` 指令. `import` 指令的语法请参见 [语法](#).

我们可以导入一个单独的名称, 比如

```
import foo.Bar // 导入后 Bar 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(包, 类, 对象, 等等)之内所有可访问的内容:

```
import foo.* // 导入后 'foo' 内的一切都可以访问了
```

如果发生了名称冲突, 我们可以使用 `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 foo.Bar // 导入后 Bar 可以访问了
import bar.Bar as bBar // 可以使用新名称 bBar 来访问 'bar.Bar'
```

`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](#) 中定义的函数和属性;
- [枚举常数](#).

与 Java 不同, Kotlin 没有单独的 [“import static”](#) 语法; 所有这些声明都使用通常的 `import` 关键字来表达.

## 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](#)).

## 控制流: if, when, for, while

### if 表达式

在 Kotlin 中, `if` 是一个表达式, 也就是说, 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.

```
// if 的传统用法
var max = a
if (a < b) max = b

// 使用 else 分支的方式
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// if 作为表达式使用
val max = if (a > b) a else b
```

`if` 的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), 而不是用作普通的流程控制语句, 这种情况下 `if` 表达式必须有 `else` 分支.

参见 [if 语法](#).

### when 表达式

`when` 替代了各种 C 风格语言中的 `switch` 操作符. 最简单的形式如下例:

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // 注意, 这里是代码段
        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, 满足条件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, 则必须有 `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 [枚举类](#) 的常数 或 [封闭类](#) 的子类型).

如果对多种条件需要进行相同的处理, 那么可以对一个分支指定多个条件, 用逗号分隔:

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

在分支条件中, 我们可以使用任意的表达式(而不仅仅是常数值)

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

我们还可以使用 `in` 或 `!in` 来检查一个值是否属于一个 [范围](#), 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](#) 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")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

从 Kotlin 1.3 开始, 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.

参见 [when 语法](#).

## 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. 语法如下:

```
for (item in collection) print(item)
```

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:
  - 存在一个成员函数- 或扩展函数 `next()`, 并且
  - 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](#):

```
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  
}
```

参见 [for 语法](#)。

## while 循环

`while` 和 `do..while` 的功能与其他语言一样:

```
while (x > 0) {  
    x--  
}  
  
do {  
    val y = retrieveData()  
} while (y != null) // y is visible here!
```

参见 [while 语法](#).

## 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. 参见 [返回与跳转](#).

## 返回与跳转

Kotlin 中存在 3 种跳出程序流程的表达式:

- `return`. 默认行为是, 从最内层的函数或 [匿名函数](#) 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](#).

### 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@` 都是合法的标签(参见 [标签语法](#)). 要给一个表达式标记标签, 我们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), 允许实现函数的嵌套. 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:

```
//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()  
}
```

这里的 `return` 会从最内层的函数中返回, 也就是, 从 `foo` 函数返回. (注意, 这种非局部的返回(non-local return), 仅对传递给 [内联函数](#) ([inline function](#)) 的 Lambda 表达式有效.) 如果需要从 Lambda 表达式返回, 我们必须对它标记一个标签, 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, 也就是, 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, 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, 也就是, 返回到 forEach 循环
        print(it)
    }
    print(" done with implicit label")
}
//sampleEnd

fun main() {
    foo()
}
```

或者, 我们也可以使用 [匿名函数](#) 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, 也就是, 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, 也就是说:

```
return@a 1
```

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)” .

# 类与对象

## 类与继承

### 类

Kotlin 中的类使用 `class` 关键字定义:

```
class Invoice { ... }
```

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, 等等.), 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 中的类可以有一个 **主构造器** (primary constructor), 以及一个或多个 **次构造器** (secondary constructor). 主构造器是类头部的一部分, 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) { ... }
```

主构造器中不能包含任何代码. 初始化代码可以放在 **初始化代码段** (initializer block) 中, 初始化代码段使用 `init` 关键字作为前缀.

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")
}
```

注意, 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.toUpperCase()
}
```

实际上, 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) { ... }
```

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](#).

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 {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

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) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

注意, 初始化代码段中的代码实际上会成为主构造器的一部分. 对主构造器的委托调用, 会作为次级构造器的第一条语句来执行, 因此所有初始化代码段中代码, 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")
    }
}
//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`. 如果你不希望你的类带有 `public` 的构造器, 你需要声明一个空的构造器, 并明确设置其可见度:

```
class DontCreateMe private constructor () { ... }
```

注意: 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, 我们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")
```

注意, 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, 请参见 [嵌套类\(Nested Class\)](#).

## 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](#)
- [属性](#)
- [嵌套类和内部类](#)
- [对象声明](#)

## 继承

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
```

注意: `Any` 不是 `java.lang.Object`; 尤其要注意, 除 `equals()`, `hashCode()` 和 `toString()` 之外, 它没有任何成员. 详情请参见 [与 Java 的互操作性](#).

要明确声明类的超类, 我们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

我们在前面提到过, 我们很注意让 Kotlin 中的一切都明白无误. 而且与 Java 不同, Kotlin 要求使用明确的修饰符来标识允许被子类覆盖的成员(我们称之为 *open*), 而且也要求使用明确的修饰符来标识对超类成员的覆盖:

```
open class Base {
    open fun v() { ... }
    fun nv() { ... }
}
class Derived() : Base() {
    override fun v() { ... }
}
```

对于 `Derived.v()` 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 `Base.nv()`, 那么在子类中声明一个同名同参的方法将是非法的, 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, 也就是说, 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 AnotherDerived() : Base() {
    final override fun v() { ... }
}
```

## 属性的覆盖

属性的覆盖方式与方法覆盖类似; 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. 可以使用带初始化器的属性来覆盖超类属性, 也可以使用带取值方法(getter)的属性来覆盖.

```
open class Foo {
    open val x: Int get() { ... }
}

class Bar1 : Foo() {
    override val x: Int = ...
}
```

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 Foo {
    val count: Int
}

class Bar1(override val count: Int): Foo

class Bar2: Foo {
    override var count: Int = 0
}
```

### 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 Base") }

    open val size: Int =
        name.length.also { println("Initializing size in Base: $it") }
}

class Derived(
    name: String,
    val lastName: String
): Base(name.capitalize().also { println("Argument for Base: $it") }) {

    init { println("Initializing Derived") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in Derived: $it") }
}
//sampleEnd

fun main() {
    println("Constructing Derived(\"hello\", \"world\")")
    val d = 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 Foo {
    open fun f() { println("Foo.f()") }
    open val x: Int get() = 1
}

class Bar : Foo() {
    override fun f() {
        super.f()
        println("Bar.f()")
    }

    override val x: Int get() = super.x + 1
}
```

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:

```
class Bar : Foo() {
    override fun f() { /* ... */ }
    override val x: Int get() = 0

    inner class Baz {
        fun g() {
            super@Bar.f() // 调用 Foo 类中的 f() 函数实现
            println(super@Bar.x) // 使用 Foo 类中的 x 属性取值方法实现
        }
    }
}
```

## 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重实现, 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重实现中的某一个). 为了表示使用的方法是从哪个超类继承得到的, 我们使用 `super` 关键字, 将超类名称放在尖括号类, 比如, `super<Base>`:

```
open class A {
    open fun f() { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // 接口的成员默认是 'open' 的
    fun b() { print("b") }
}

class C() : A(), B {
    // 编译器要求 f() 方法必须覆盖:
    override fun f() {
        super<A>.f() // 调用 A.f()
        super<B>.f() // 调用 B.f()
    }
}
```

同时继承 `A` 和 `B` 是合法的, 而且函数 `a()` 和 `b()` 的继承也不存在问题, 因为对于这两个函数, `C` 类都只继承得到了唯一的一个实现. 但对函数 `f()` 的继承就发生了问题, 因为 `C` 类从超类中继承得到了两个实现, 因此在 `C` 类中我们必须覆盖函数 `f()`, 并提供我们自己的实现, 这样才能消除歧义.

## 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. 注意, 我们不必对抽象类或抽象成员标注 `open` 修饰符 – 因为它显然必须是 `open` 的.

我们可以使用抽象成员来覆盖一个非抽象的 `open` 成员:

```
open class Base {  
    open fun f() {}  
}  
  
abstract class Derived : Base() {  
    override abstract fun f()  
}
```

## 同伴对象(Companion Object)

与 Java 或 C# 不同, Kotlin 的类没有静态方法(static method). 大多数情况下, 建议使用包级函数(package-level function)替代静态方法.

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](#) 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](#), 那么只需要使用类名作为限定符就可以调用同伴对象的成员了, 语法与 Java/C# 中调用类的静态方法一样.



## 属性(Property)与域(Field)

### 声明属性

Kotlin 中的类可以拥有属性. 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 = ...  
    var street: String = ...  
    var city: String = ...  
    var state: String? = ...  
    var zip: String = ...  
}
```

使用属性时, 只需要简单地通过属性名来参照它, 和使用 Java 中的域变量(field)一样:

```
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, 详情见下文), 则属性类型的声明也可以省略.

示例:

```
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  
var initialized = 1 // 属性类型为 Int, 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
```

我们可以为属性定义自定义的访问方法. 如果我们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(因此我们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:

```
val isEmpty: Boolean  
    get() = this.size == 0
```

如果我们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. 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.

从 Kotlin 1.1 开始, 如果属性类型可以通过取值方法推断得到, 那么你可以在属性的定义中省略类型声明:

```
val isEmpty get() = this.size == 0 // 属性类型为 Boolean
```

如果你需要改变属性访问方法的可见度, 或者需要对其添加注解, 但又不需要修改它的默认实现, 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)

Kotlin 的类不能直接声明域变量. 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 // 注意: 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
    }
}
```

`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, 下面的情况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")
    }
```

不管从哪方面看, 这种方案都与 Java 中完全相同, 因为后端私有属性的取值方法与设值方法都使用默认实现, 我们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, 因此不会发生不必要的函数调用, 导致性能损失.

## 编译期常数值

如果属性值在编译期间就能确定, 则可以使用 `const` 修饰符, 将属性标记为 *编译期常数值(compile time constants)*. 这类属性必须满足以下所有条件:

- 必须是顶级属性, 或者是一个 [object 声明](#) 的成员, 或者是一个 [companion object](#) 的成员.
- 值被初始化为 `String` 类型, 或基本类型(primitive type)
- 不存在自定义的取值方法

这类属性可以用在注解内:

```
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). 从 Kotlin 1.2 开始, 这个修饰符也可以用于顶级(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.

### 检查延迟初始化的变量是否已经初始化完成(从 Kotlin 1.2 开始支持)

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](#) 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, 也就是说, 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](#)

## 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, 我们可以实现属性任意复杂的行为. 在这两种极端情况之间, 还存在一些非常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, 等等.

这些常见行为可以使用 [委托属性\(delegated property\)](#) 以库的形式实现.

## 接口

Kotlin 中的接口与 Java 8 非常类似。接口中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, 这时, 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)

当我们为一个类指定了多个超类, 可能会导致我们对同一个方法继承得到了多个实现. 比如:

```

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, 如果我们从 *A* 派生一个实体类 *C*, 显然, 我们必须覆盖函数 *bar()*, 并提供一个实现.

然而, 如果我们从 *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.

## 可见度修饰符

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`. 如果没有明确指定修饰符, 则使用默认的可见度 `public`.

在不同的范围内, 这些可见度的含义是不同的, 详情请看下文.

### 包

函数, 属性, 类, 对象, 接口, 都可以声明为”顶级的(top-level)”, 也就是说, 直接声明在包之内:

```
// file name: example.kt
package foo

fun baz() { ... }
class Bar { ... }
```

- 如果你不指定任何可见度修饰符, 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.

注意: 如果一个包中的顶级声明在另一个包中可以访问, 在使用它时, 仍然需要 [导入\(import\)](#) 它.

示例:

```
// file name: 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
```

### 类与接口

对于类内部声明的成员:

- `private` 表示只在这个类(以及它的所有成员)之内可以访问;
- `protected` — 与 `private` 一样, 另外在子类中也可以访问;
- `internal` — 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` — 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.

Java 使用者 请注意: 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` 成员, 并且没有明确指定可见度, 那么覆盖后成员的可见度也将是 `protected`.

示例:

```

open class Outer {
    private val a = 1
    protected open val b = 2
    internal 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
}

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(注意, 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, 因此使得凡是访问到类的地方都可以访问到类的构造器(也就是说, 一个 `internal` 类的构造器只能在同一个模块内访问).

## 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:

- 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

与 C# 和 Gosu 类似, Kotlin 提供了向一个类扩展新功能的能力, 而且不必从这个类继承, 也不必使用任何设计模式, 比如 Decorator 模式之类. 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)*. Kotlin 支持 *扩展函数(extension function)* 和 *扩展属性(extension property)*.

### 扩展函数(Extension Function)

要声明一个扩展函数, 我们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)*, 也就是说, 表明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, 我们可以对任意一个 `MutableList<Int>` 对象调用这个扩展函数:

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'swap()' 函数内的 'this' 将指向 'l' 的值
```

显然, 这个函数可以适用与任意元素类型的 `MutableList<T>`, 因此我们可以使用泛型, 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
}
```

我们在函数名之前声明了泛型的类型参数, 然后在接收者类型表达式中就可以使用泛型了. 参见 [泛型函数](#).

### 扩展函数是静态解析的

扩展函数并不会真正修改它所扩展的类. 定义扩展函数时, 其实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.

我们希望强调一下, 扩展函数的调用派发过程是**静态的**, 也就是说, 它并不是接收者类型的虚拟成员. 这就意味着, 调用扩展函数时, 具体被调用的函数是哪一个, 是通过调用函数的对象表达式的类型来决定的, 而不是在运行时刻表达式动态计算的最终结果类型决定的. 比如:

```
open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

fun printFoo(c: C) {
    println(c.foo())
}

printFoo(D())
```

这段示例程序的打印结果将是 “c”, 因为调用哪个函数, 仅仅是由参数 `c` 声明的类型决定, 这里参数 `c` 的类型为 `C` 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 总是会优先使用成员函数. 比如:

```
class C {  
    fun foo() { println("member") }  
}  
  
fun C.foo() { println("extension") }
```

如果我们对 `C` 类型的任意变量 `c` 调用 `c.foo()`, 结果会打印 “member”, 而不是 “extension” .

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:

```
class C {  
    fun foo() { println("member") }  
}  
  
fun C.foo(i: Int) { println("extension") }
```

调用 `C().foo(1)` 的打印结果将是 “extension” .

### 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, 在扩展函数的实现体之内, 可以通过 `this == null` 来检查接收者是否为 `null`. 在 Kotlin 中允许你调用 `toString()` 函数, 而不必检查对象是否为 `null`, 就是通过这个原理实现的: 对象是否为 `null` 的检查发生在扩展函数内部, 因此调用者不必再做检查.

```
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)

与扩展函数类似, Kotlin 也支持扩展属性:

```
val <T> List<T>.lastIndex: Int  
    get() = size - 1
```

注意, 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](#). 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 Foo.bar = 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](#), 你可以对这个同伴对象定义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.foo() { ... }
```

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:

```
MyClass.foo()
```

## 扩展的范围

大多数时候我们会在顶级位置定义扩展, 也就是说, 直接定义在包之下:

```
package foo.bar

fun Baz.goo() { ... }
```

要在扩展定义所在的包之外使用扩展, 我们需要在调用处 import 这个包:

```
package com.example.usage

import foo.bar.goo // 通过名称 "goo" 来导入扩展
// 或者
import foo.bar.* // 导入 "foo.bar" 包之下的全部内容

fun usage(baz: Baz) {
    baz.goo()
}
```

详情请参见 [导入](#).

## 将扩展定义为成员

在类的内部, 你可以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 D {
    fun bar() { ... }
}

class C {
    fun baz() { ... }

    fun D.foo() {
        bar() // 这里将会调用 D.bar
        baz() // 这里将会调用 C.baz
    }

    fun caller(d: D) {
        d.foo() // 这里将会调用扩展函数
    }
}
```

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](#).

```
class C {
    fun D.foo() {
        toString() // 这里将会调用 D.toString()
        this@C.toString() // 这里将会调用 C.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 D {}

class D1 : D() {}

open class C {
    open fun D.foo() {
        println("D.foo in C")
    }

    open fun D1.foo() {
        println("D1.foo in C")
    }

    fun caller(d: D) {
        d.foo() // 调用扩展函数
    }
}

class C1 : C() {
    override fun D.foo() {
        println("D.foo in C1")
    }

    override fun D1.foo() {
        println("D1.foo in C1")
    }
}

fun main() {
    C().caller(D()) // 打印结果为 "D.foo in C"
    C1().caller(D()) // 打印结果为 "D.foo in C1" - 派发接受者的解析过程是虚拟的
    C().caller(D1()) // 打印结果为 "D.foo in C" - 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](#) 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, 那么这样的扩展不能访问接受者的 `private` 成员.

## 使用扩展的动机

在 Java 中, 我们通常会使用各种名为 “\*Utils” 的工具类: `FileUtils`, `StringUtils` 等等. 著名的 `java.util.Collections` 也属于这种工具类. 这种工具类模式令人很不愉快的地方在于, 使用时代码会写成这种样子:

```
// java
Collections.swap(list, Collections.binarySearch(list,
Collections.max(otherList)),
Collections.max(list));
```

代码中反复出现的工具类类名非常烦人. 我们也可以使用静态导入(static import), 然后代码会变成这样:

```
// java
swap(list, binarySearch(list, max(otherList)), max(list));
```

这样略好了一点点, 但是没有了类名做前缀, 就导致我们无法利用 IDE 强大的代码自动补完功能. 如果我们能写下面这样的代码, 那不是很好吗:

```
// java
list.swap(list.binarySearch(otherList.max()), list.max());
```

但是我们又希望将一切可能出现的方法在 `List` 类之内全部都实现出来, 对不对? 这恰恰就是 Kotlin 的扩展机制可以帮助我们解决的问题.

## 数据类

我们经常会创建一些类, 其主要目的是用来保存数据. 在这些类中, 某些常见的功能和工具函数经常可以由类中保存的数据内容即可自动推断得到. 在 Kotlin 中, 我们将这样的类称为 *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](#), 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:

- 主构造器至少要有有一个参数;
- 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;
- (在 Kotlin 1.1 以前) 数据类只允许实现接口.

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;
- 从一个已经存在参数签名相同的 `copy(...)` 函数的父类继承一个新的数据类, 这种行为在 Kotlin 1.2 中已被废弃, 在 Kotlin 1.3 中已被禁止.
- 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).

从 Kotlin 1.1 开始, 数据类可以继承其他类 (示例请参见 [封闭类\(Sealed class\)](#)).

在 JVM 上, 如果自动生成的类需要拥有一个无参数的构造器, 那么需要为所有的属性指定默认值 (参见 [构造器](#)).

```
data class User(val name: String = "", val age: Int = 0)
```

### 在类主体部声明的属性

注意, 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 `toString()`, `equals()`, `hashCode()`, 和 `copy()` 函数的实现中, 只会使用属性 `name`, 而且只存在一个组建函数 `component1()`. 两个 `Person` 对象可以拥有不同的年龄, 但它们会被认为值相等.

```

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
    //sampleEnd
    println("person1 == person2: ${person1 == person2}")
    println("person1 with age ${person1.age}: ${person1}")
    println("person2 with age ${person2.age}: ${person2}")
}

```

## 对象复制

我们经常会需要复制一个对象, 然后修改它的一部分属性, 但保持其他属性不变. 这就是自动生成的 `copy()` 函数所要实现的功能. 对于前面示例中的 `User` 类, 自动生成的 `copy()` 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)

```

有了这个函数, 我们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\(destructuring declaration\)](#) 中使用数据类:

```

val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // 打印结果将是 "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. 但是, 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, 因此可以增加代码的可读性.

## 封闭类(Sealed Class)

封闭类(Sealed class)用来表示对类阶层的限制, 可以限定一个值只允许是某些指定的类型之一, 而不允许是其他类型. 感觉上, 封闭类是枚举类(enum class)的一种扩展: 枚举类的值也是有限的, 但每一个枚举值常数都只存在唯一的一个实例, 封闭类则不同, 它允许的子类类型是有限的, 但子类可以有多个实例, 每个实例都可以包含它自己的状态数据.

要声明一个封闭类, 需要将 `sealed` 修饰符放在类名之前. 封闭类可以有子类, 但所有的子类声明都必须定义在封闭类所在的同一个源代码文件内. (在 Kotlin 1.1 之前, 规则更加严格: 所有的子类声明都必须嵌套在封闭类的声明部分之内).

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()
```

(上面的示例使用了 Kotlin 1.1 的新增特性: 允许数据类继承其他类, 包括继承封闭类.)

封闭类本身是 [抽象\(abstract\)类](#), 不允许直接创建实例, 而且封闭类可以拥有 `abstract` 成员.

封闭类不允许拥有非 `private` 的构造器(封闭类的构造器默认都是 `private` 的).

注意, 从封闭类的子类再继承的子类(间接继承者)可以放在任何地方, 不必在封闭类所在的同一个源代码文件内.

使用封闭类的主要好处在于, 当使用 [when expression](#) 时, 可以验证分支语句覆盖了所有的可能情况, 因此就不必通过 `else` 分支来处理例外情况. 但是, 这种用法只适用于将 `when` 用作表达式(使用它的返回值)的情况, 而不能用于将 `when` 用作语句的情况.

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // 不需要 `else` 分支, 因为我们已经覆盖了所有的可能情况
}
```



## 泛型

与 Java 一样, Kotlin 中的类也可以有类型参数:

```
class Box<T>(t: T) {  
    var value = t  
}
```

通常, 要创建这样一个类的实例, 我们需要指定类型参数:

```
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, 或通过其他手段推断得到, 此时允许省略类型参数:

```
val box = Box(1) // 1 的类型为 Int, 因此编译器知道我们创建的实例是 Box<Int> 类型
```

### 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](#)). Kotlin 中不存在这样的通配符类型. 它使用另外的两种东西: 声明处类型变异(declaration-site variance), 以及类型投射(type projection).

首先, 我们来思考一下为什么 Java 需要那些神秘的通配符类型. 这个问题已有详细的解释, 请参见 [Effective Java, 第 3 版](#), 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). 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. 为什么会这样? 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, 然后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>();  
List<Object> objs = strs; // !!! 导致后面问题的原因就在这里. Java 会禁止这样的代码!  
objs.add(1); // 在这里, 我们向 String 组成的 List 添加了一个 Integer 类型的元素  
String s = strs.get(0); // !!! ClassCastException: 无法将 Integer 转换为 String
```

由于存在这种问题, Java 禁止上面示例中的做法, 以便保证运行时刻的类型安全. 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? 直觉地, 我们会将它定义为:

```
// Java  
interface Collection<E> ... {  
    void addAll(Collection<E> items);  
}
```

但是, 这样的定义会导致我们无法进行下面这种非常简单的操作(尽管这种操作是绝对安全的):

```
// Java  
void copyAll(Collection<Object> to, Collection<String> from) {  
    to.addAll(from);  
    // !!! 如果 addAll 方法使用前面那种简单的定义, 这里的调用将无法通过编译:  
    // 因为 Collection<String> 不是 Collection<Object> 的子类型  
}
```

(在 Java 语言中, 我们通过非常痛苦的方式才学到了这个教训, 详情请参见 [Effective Java, 第 3 版](#), 第 28 条: 尽量使用 `List`, 而不是数组)

正因为上面的问题, 所以 `addAll()` 的签名定义其实是这样的:

```
// 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` 的某种子类型, 而不同于 `E` 本身. 这就意味着, 我们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, 因为我们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, 我们得到了希望的功能: `Collection<String>` 是 `Collection<? extends Object>` 的子类型. 用“高级术语”来说, 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.

要理解这种技巧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, 我们可以使用 `List<? super String>`, 它是 `List<Object>` 的一个 **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)`), 而当你调用 `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, 而只是 `Object` 类型.

Joshua Bloch 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: 生产者(Producer)对应 Extends, 消费者(Consumer)对应 Super.*

**注意:** 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)

假设我们有一个泛型接口 `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, 我们不得不将对象类型声明为 `Source<? extends Object>`, 其实是毫无意义的, 因为我们在这样修改之后, 我们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)**: 我们可以对 `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, 我们可以对 `T` 添加 **out** 修饰符:

```
interface Source<out T> {
    abstract 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。

用”高级术语”来说, 我们将 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, 因此我们称之为 **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)**: 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, 我们可以将 x 赋值给 Comparable<Double> 类型的变量
    val y: Comparable<Double> = x // OK!
}
```

我们认为 in 和 out 关键字的意义是十分直观的(同样的关键字已经在 C# 中经常使用了), 因此, 前面提到的记忆口诀也没有必要了, 为了一种崇高的理念, 我们可以将它改写一下:

**存在主义** 变形法则: 消费者进去, 生产者出来! :-)

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) {
    fun get(index: Int): T { ... }
    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, 我们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. 为什么会这样? 原因与以前一样, 因为 `copy` 函数 **有可能** 会做一些不安全的操作, 也就是说, 这个函数可能会试图向 `from` 数组中 **写入**, 比如说, 一个 `String`, 这时假如我们传入的实际参数是一个 `Int` 的数组, 就会导致一个 `ClassCastException`.

所以, 我们需要确保的就是 `copy()` 函数不会做这类不安全的操作. 我们希望禁止这个函数向 `from` 数组 **写入** 数据, 我们可以这样声明:

```
fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

这种声明在 Kotlin 中称为 **类型投射(type projection)**: 我们声明的含义是, `from` 不是一个单纯的数组, 而是一个被限制(**投射**)的数组: 我们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, 我们只能调用 `get()` 方法. 这就是我们实现 **使用处的类型变异(use-site variance)** 的方案, 与 Java 的 `Array<? extends Object>` 相同, 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>`, 那么可以出现以下几种星号投射:

- `Function<*, String>`, 代表 `Function<in Nothing, String>`;
- `Function<Int, *>`, 代表 `Function<Int, out Any?>`;
- `Function<*, *>`, 代表 `Function<in Nothing, out Any?>`.

注意: 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)**, 与 Java 中的 *extends* 关键字相同:

```
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)**: 对于类型参数 `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, 这时就需要使用单独的 **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.

## 类型擦除

对于使用泛型声明的代码, Kotlin 只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<*>`.

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, 而且编译器 **禁止这样的 is 检查**.

把一种类型转换为带具体类型参数的泛型类型, 比如 `foo as List<String>`, 在运行时也无法进行类型安全性检查. 如果类型安全性不能通过编译器直接推断得到, 但是更高层次的程序逻辑可以保证, 那么可以使用这种 **未检查的类型转换**. 编译器会对未检查的类型转换报告一个警告, 在运行时, 只会针对泛型以外的部分进行类型检查 (前面的例子等价于 `foo as List<*>`).

泛型函数调用时的类型参数同样只在编译时进行类型检查. 在函数体内部, 不能对类型参数进行类型检查, 把一种类型转换为函数的类型参数类型 (比如 `foo as T`) 同样是未检查的类型转换. 但是, 内联函数的 [实体化的类型参数](#) 会在调用处被替换为内联函数体内部的实际类型参数, 因此这时可以用类型参数来进行类型检查和类型转换, 但这里的类型检查和类型转换, 也和前面讲到的泛型类的实例一样存在同样的限制.

## 嵌套类(Nested Class)与内部类(Inner Class)

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
```

### 内部类(Inner class)

类可以使用 `inner` 关键字来标记, 然后就可以访问外部类(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](#).

### 匿名内部类(Anonymous inner class)

匿名内部类的实例使用 [对象表达式\(object expression\)](#) 来创建:

```
window.addMouseListener(object : MouseAdapter() {  
    override fun mouseClicked(e: MouseEvent) { ... }  
  
    override fun mouseEntered(e: MouseEvent) { ... }  
})
```

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(Enum Class)

枚举类最基本的用法, 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.

### 初始化

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

枚举常数也可以定义它自己的匿名类:

```
enum class ProtocolState {  
    WAITING {  
        override fun signal() = TALKING  
    },  
  
    TALKING {  
        override fun signal() = WAITING  
    };  
  
    abstract fun signal(): ProtocolState  
}
```

枚举常数的匿名类可以有各自的方法, 还可以覆盖基类的方法. 注意, 与 Java 中一样, 如果枚举类中定义了任何成员, 你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.

除内部类(inner class)之外, 枚举常数中不能包含嵌套类型 (这个限制在 Kotlin 1.2 版中已被废除).

### 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), 可以对所有的枚举常数提供唯一的接口实现, 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.values()) {
        println("$f($a, $b) = ${f.apply(a, b)}")
    }
}

```

## 使用枚举常数

与 Java 中一样, Kotlin 中的枚举类拥有编译器添加的方法, 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.values(): Array<EnumClass>

```

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.

从 Kotlin 1.1 开始, 可以通过 `enumValues<T>()` 和 `enumValueOf<T>()` 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() {
    print(enumValues<T>().joinToString { it.name })
}

printAllValues<RGB>() // 打印结果为 RED, GREEN, BLUE

```

每个枚举常数值都拥有属性, 可以取得它的名称, 以及它在枚举类中声明的顺序:

```

val name: String
val ordinal: Int

```

枚举常数值还实现了 [Comparable](#) 接口, 枚举常数值之间比较时, 会使用枚举常数值在枚举类中声明的顺序作为自己的大小顺序.

## 对象表达式(Object Expression)与对象声明(Object Declaration)

有时我们需要创建一个对象, 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修改就明确地声明一个新类. Java 通过 *匿名内部类(anonymous inner class)* 来解决这种问题. Kotlin 使用 *对象表达式(object expression)* 和 *对象声明(object declaration)*, 对这个概念略做了一点泛化.

### 对象表达式(Object expression)

要创建一个继承自某个类(或多个类)的匿名类的对象, 我们需要写这样的代码:

```
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  
}
```

如果, 我们 “只需要对象”, 而不需要继承任何有价值的基类, 我们可以简单地写:

```
fun foo() {  
    val adHoc = object {  
        var x: Int = 0  
        var y: Int = 0  
    }  
    print(adHoc.x + adHoc.y)  
}
```

注意, 只有在局部并且私有的声明范围内, 匿名对象才可以被用作类型. 如果你将匿名对象用作公开函数的返回类型, 或者用作公开属性的类型, 那么这个函数或属性的真实类型会被声明为这个匿名对象的超类, 如果匿名对象没有超类, 则是 `Any`. 在匿名对象中添加的成员将无法访问.

```
class C {  
    // 私有函数, 因此它的返回类型为匿名对象类型  
    private fun foo() = object {  
        val x: String = "x"  
    }  
  
    // 公开函数, 因此它的返回类型为 Any  
    fun publicFoo() = object {  
        val x: String = "x"  
    }  
  
    fun bar() {  
        val x1 = foo().x // 正确  
        val x2 = publicFoo().x // 错误: 无法找到 'x'  
    }  
}
```

与 Java 的匿名内部类(anonymous inner class)类似, 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. (与 Java 不同的是, 被访问的变量不需要被限制为 final 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](#) 在有些情况下可能是很有用的, Kotlin (继 Scala 之后) 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).

要引用这个对象, 我们直接使用它的名称:

```
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) { ... }
}
```

**注意:** 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.

## 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
```

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](#).

## 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).

## 类型别名 (从 Kotlin 1.1 开始支持)

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]"
}
```

## 内联类

⚠ 内联类从 Kotlin 1.3 以后才可以使用, 而且目前还是 [实验性功能](#). 详情请参见 [下文](#)

对于业务逻辑来说, 有些时候会需要对某些类型创建一些包装类. 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), 声明内联类时需要在类名称之前添加 `inline` 修饰符:

```
inline class Password(val value: 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](#) 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:

```
inline class Name(val s: String) {
    val length: Int
    get() = s.length

    fun greet() {
        println("Hello, $s")
    }
}

fun main() {
    val name = Name("Kotlin")
    name.greet() // 方法 `greet` 会作为静态方法来调用
    println(name.length) // 属性的取值函数会作为静态方法来调用
}
```

但是, 对于内联类的成员存在一些限制:

- 内联类不能拥有 `init` 代码段
- 内联类的属性不能拥有 [后端域变量](#)
  - 因此, 内联类只能拥有简单的计算属性 (不能拥有延迟初始化属性或委托属性)

### 继承

内联类允许继承接口:

```

interface Printable {
    fun prettyPrint(): String
}

inline 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，而且它必须是 `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](#) 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

inline 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](#) 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。

## 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：

```
inline 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.

⚠ 注意, 在 Java 中 `-` 是一个 *无效的* 符号, 也就是说从 Java 中无法调用那些使用了内联类作为参数的函数.

## 内联类与类型别名

初看起来, 内联类好像非常象 [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
inline 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
}
```

## 内联类功能还在实验性阶段

内联类的设计目前还处于实验性阶段, 也就是说这个功能正在 *快速变化* 中, 不保证兼容性. 在 Kotlin 1.3+ 中使用内联类时, 编译器会报告警告信息, 指出这个功能是实验性的.

要删除这些警告, 你需要对 `kotlinc` 指定 `-XXLanguage:+InlineClasses` 选项, 来允许使用这个实验性功能.

## 在 Gradle 中启用内联类

```
compileKotlin {
    kotlinOptions.freeCompilerArgs += ["-XXLanguage:+InlineClasses"]
}
```

详情请参见 [Gradle 中的编译器选项](#). 关于 [跨平台项目](#) 的设置, 请参见 [使用 Gradle 编译跨平台项目](#).

## 在 Maven 中启用内联类



```
<configuration>
  <args>
    <arg>-XXLanguage:+InlineClasses</arg>
  </args>
</configuration>
```

详情请参见 [Maven 中的编译器选项](#) for details.

## 更深入地讨论

关于更多技术细节和讨论, 请参见 [关于 Kotlin 语言内联类的建议](#).

## 委托(Delegation)

### 属性的委托(Property Delegation)

委托属性(Delegated Property) 请参见单独的章节: [委托属性](#).

### 通过委托实现接口

[委托模式](#) 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例行公事的代码(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](#) 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. 如果我们在 `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)  
}
```

## 委托属性(Delegated Property)

有许多非常具有共性的属性, 虽然我们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, 供所有需要的类使用, 那将会好很多. 这样的例子包括:

- 延迟加载属性(lazy property): 属性值只在初次访问时才会计算;
- 可观察属性(observable property): 属性发生变化时, 可以向监听器发送通知;
- 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. 属性委托不必实现任何接口, 但必须提供 `getValue()` 函数(对于 `var` 属性, 还需要 `setValue()` 函数). 示例:

```
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, 那么当我们读取属性值时, 就会调用到 `Delegate` 的 `getValue()` 函数, 此时函数收到的第一个参数将是我们访问的属性 `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). For example:

```
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, 当我们向属性 `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](#).

注意, 从 Kotlin 1.1 开始, 你可以在函数内, 或者一个代码段内定义委托属性, 委托属性不需要一定是类的成员. 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)

[lazy\(\)](#) 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, 实现延迟加载属性 (lazy property): 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, 延迟加载属性(lazy property)的计算是 **同步的(synchronized)**: 属性值只会在唯一一个线程内计算, 然后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。相反, 如果你确信初始化计算只可能发生在一个线程内, 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。

### 可观察属性(Observable)

[Delegates.observable\(\)](#) 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)。每次我们向属性赋值时, 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"
}
```

如果你希望能够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\(\)](#) 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。

### 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) (从 Kotlin 1.1 开始支持)

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

下面我们总结一下对属性委托对象的要求.

对于一个 只读 属性 (也就是说, val 属性), 它的委托必须提供一个名为 getValue 的函数, 这个函数接受以下参数:

- thisRef — 这个参数的类型必须与 属性所属的类 相同, 或者是它的基类(对于扩展属性 — 这个参数的类型必须与被扩展的类型相同, 或者是它的基类);
- property — 这个参数的类型必须是 KProperty<\*>, 或者是它的基类.

这个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).

对于一个 值可变(mutable) 属性(也就是说, var 属性), 除 getValue 函数之外, 它的委托还必须 另外再 提供一个名为 setValue 的函数, 这个函数接受以下参数:

- thisRef — 与 getValue() 函数的参数相同;
- property — 与 getValue() 函数的参数相同;
- new value — 这个参数的类型必须与属性类型相同, 或者是它的基类.

`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`.

委托类可以选择实现 `ReadOnlyProperty` 接口或 `ReadWriteProperty` 接口, 其中包含了需要的 `operator` 方法. 这些接口定义在 Kotlin 标准库内:

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}
```

### 编译器对委托属性的翻译规则

委托属性的底层实现是, 对每个委托属性, Kotlin 编译器会生成一个辅助属性, 并将目标属性的存取操作委托给它. 比如, 对于属性 `prop`, 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.

注意, `this::prop` 语法是一种 [与对象实例绑定的可调用的引用](#), 这种语法从 Kotlin 1.1 开始支持.

### 控制属性委托的创建逻辑 (从 Kotlin 1.1 开始支持)

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, 是在属性创建时检查属性的一致性, 而不仅仅是在属性的取值函数或设值函数中检查.

比如, 如果你希望在(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` — 这个参数的类型必须与 *属性所属的类* 相同, 或者是它的基类(对于扩展属性 — 这个参数的类型必须与被扩展的类型相同, 或者是它的基类);
- `property` — 这个参数的类型必须是 `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.

# 函数与 Lambda 表达式

## 函数

### 函数声明

Kotlin 中使用 `fun` 关键字定义函数:

```
fun double(x: Int): Int {  
    return 2 * x  
}
```

### 函数使用

函数的调用使用传统的方式:

```
val result = double(2)
```

调用类的成员函数时, 使用点号标记法(dot notation):

```
Sample().foo() // 创建一个 Sample 类的实例, 然后调用这个实例的 foo 函数
```

### 参数

函数参数的定义使用 Pascal 标记法, 也就是, *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) { ... }
```

### 默认参数

函数参数可以指定默认值, 当参数省略时, 就会使用默认值. 与其他语言相比, 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: Array<Byte>, 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, 总是会使用与基类中方法相同的默认参数值. 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](#), 那么这个 lambda 表达式可以使用命名参数的方式传递, 也可以 [在括号之外传递](#):

```
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, 可以通过参数名来指定参数. 当函数参数很多, 或者存在默认参数时, 指定参数名是一种非常便利的功能.

比如, 对于下面这个函数:

```
fun reformat(str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ') {
    ...
}
```

我们可以使用默认参数来调用它:

```
reformat(str)
```

但是, 如果需要使用非默认的参数值调用它, 那么代码会成为这样:

```
reformat(str, true, true, false, '_')
```

如果使用命名参数, 我们的代码可读性可以变得更好一些:

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

而且, 如果我们不需要指定所有的参数, 那么可以这样:

```
reformat(str, wordSeparator = '_')
```

如果在调用函数时混合使用位置参数和命名参数, 那么所有的位置参数必须放置在第一个命名参数之前. 比如, `f(1, y = 2)` 是允许的, 但是 `f(x = 1, 2)` 是错误的.

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"))
```

注意, 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任何有意义的结果值, 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)

如果一个函数返回单个表达式, 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)

一个函数的一个参数 (通常是参数中的最后一个) 可以标记为 `vararg`:

```
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
}
```

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, 也就是说, 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, 我们可以逐个传递参数值, 比如, `asList(1, 2, 3)`, 或者, 如果我们已经有了一个数组, 希望将它的内容传递给函数, 我们可以使用 **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)
```

## 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](#);
- 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`

关于运算符优先级的完整信息, 请参见 [语法参考](#)。

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`; 与调用其他方法不同, 这时 `this` 不能省略。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

在 Kotlin 中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 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 中的函数也可以定义为局部函数, 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: Set<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(也就是, 闭包), 因此, 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](#) 和 [继承](#).

## 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](#).

## 内联函数(Inline Function)

内联函数的详细解释在 [这里](#).

## 扩展函数

扩展函数的详细解释在 [单独的章节](#).

## 高阶函数(Higher-Order Function) 与 Lambda 表达式

高阶函数(Higher-Order Function) 与 Lambda 表达式的详细解释在 [单独的章节](#).

### 尾递归函数(Tail recursive function)

Kotlin 支持一种称为 [尾递归\(tail recursion\)](#) 的函数式编程方式. 这种方式使得某些本来需要使用循环来实现的算法, 可以改用递归函数来实现, 但同时不会存在栈溢出(stack overflow)的风险. 当一个函数标记为 `tailrec`, 并且满足要求的形式, 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, 而且你不能将尾递归用在 `try/catch/finally` 结构内. 尾递归目前只能用在 JVM 环境内.

## 高阶函数与 Lambda 表达式

在 Kotlin 中函数是 [一级公民](#), 也就是说, 函数可以保存在变量和数据结构中, 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, 任意操作函数.

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\(fold\)](#), 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, 我们需要向它传递一个 [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")
}
```

在这里我们提到了一些新的概念, 我们会在后面的章节中进行详细解释.

### 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 使用了一系列的函数类型(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, 也就是说, 用于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](#) 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 [带接受者的函数数字面值](#).
- [挂起函数\(Suspending function\)](#) 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](#) 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](#), 局部[函数](#), 成员[函数](#), 或扩展[函数](#), 比如: `::isOdd`, `String::toInt`,
  - 顶级[属性](#), 成员[属性](#), 或扩展[属性](#), 比如: `List<Int>::size`,
  - [构造器](#), 比如: `::Regex`

以上几种形式都包括 [绑定到实例的可调用的引用](#), 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](#): `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](#) 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](#) 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“函数数字面值(function literal)”, 也就是, 这个函数本身没有类型声明, 而是立即作为表达式传递出去. 我们来看看下面的示例:

```

max(strings, { a, b -> a.length < b.length })

```

函数 `max` 是一个高阶函数, 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, 也就是说, 它是一个函数数字面值. 这个函数数字面值, 等价于下面的这个有名称的函数:

```

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 = { 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.

如果我们把所有可选的内容都去掉, 那么剩余的部分如下:

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

### 将 Lambda 表达式用作函数调用的最后一个参数

在 Kotlin 中有一种规约, 如果函数的最后一个参数接受一个函数类型的值, 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 }
```

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.

如果编译器能够自行识别出 Lambda 表达式的参数定义, 那么我们可以不必声明这个唯一的参数, 并省略 `->` 符号, 这个参数会隐含地声明为 `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](#) 语法, 我们可以在 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](#) 的程序:

```
strings.filter { it.length == 5 }.sortedBy { it }.map { it.toUpperCase() }
```

### 使用下划线代替未使用的参数 (从 Kotlin 1.1 开始支持)

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 (从 Kotlin 1.1 开始支持)

关于在 Lambda 表达式中使用解构声明, 请参见 [解构声明\(destructuring declaration\)](#).

### 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, 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`).

注意, 匿名函数参数一定要在圆括号内传递. 允许将函数类型参数写在圆括号之外语法, 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\(non-local return\)](#) 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](#), [对象表达式](#)) 可以访问它的 *闭包*, 也就是, 定义在外层范围中的变量. 与 Java 不同, 闭包中捕获的变量是可以修改的 (译注: Java 中必须为 `final` 变量):

```
var sum = 0  
ints.filter { it > 0 }.forEach {  
    sum += it  
}  
print(sum)
```

### 带有接受者的函数数字面值

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数字面值来创建它的实例, 也就是带接受者的函数数字面值.

上文中我们讲到, Kotlin 提供了一种能力, 在 [调用带接受者的函数类型的实例](#) 时, 可以指定一个 *接收者对象(receiver object)*.

在这个函数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](#) 来访问接受者对象.

这种行为与 [扩展函数](#) 很类似, 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数字面值, 以及这个函数数字面值的类型, 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\(Type-Safe Builder\)](#):

```
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)

使用 [高阶函数](#) 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, 也就是, 在函数体内部访问的那些外层变量. 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. 也就是说, `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()
}
```

这不就是我们最初期望的东西吗?

为了让编译器做到这点, 我们需要使用 `inline` 修饰符标记 `lock()` 函数:

```
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, 但是, 只要我们合理使用(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](#))

### 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, 你可以对函数的一部分参数添加 `noinline` 标记:

```
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, 但 `noinline` 的 Lambda 表达式可以按照我们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.

注意. 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. 这就意味着, 要退出一个 Lambda 表达式, 我们必须使用一个 [标签](#), 无标签的 `return` 在 Lambda 表达式内是禁止使用的, 因为 Lambda 表达式不允许强制包含它的函数返回:

```

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!")
}
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. 我们在循环中经常用到这样的结构, 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. 为了标识这一点, Lambda 表达式参数需要添加 `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)

有些时候我们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, 我们向上遍历一颗树, 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)
```

我们真正需要的, 只是简单地将一个类型传递给这个函数, 也就是说, 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)*, 使用这个功能我们可以将代码写成:

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

我们给类型参数添加了 `reified` 修饰符, 现在, 它可以在函数内部访问了, 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, 通常的操作符, 比如 `!is` 和 `as` 都可以正常工作了. 此外, 我们可以象前面提到的那样来调用这个函数:

```
myTree.findParentOfType<MyTreeNodeType>().
```

虽然很多情况下并不需要, 但我们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.

关于实体化类型参数的更底层的介绍, 请参见 [规格文档](#).

## 内联属性(Inline property) (从 Kotlin 1.1 开始支持)

对于不存在后端变量的属性, 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, 并且不属于 `private` 或 `internal` 类型的一部分, 这个函数将被认为是一个 [模块\(module\)](#) 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.

# 跨平台程序开发

## 与平台相关的声明

⚠️ 跨平台项目是 Kotlin 1.2 和 1.3 版中的实验性特性. 本文档描述的所有语言特性和工具特性, 在未来的 Kotlin 版本中都有可能发生变更.

Kotlin 跨平台代码的关键特性之一就是, 允许共通代码依赖到与平台相关的声明. 在其他语言中, 要实现这一点, 通常是在共通代码中创建一系列的接口, 然后在与平台相关的代码中实现这些接口. 但是, 如果你已经有一个库在某个平台上实现了你需要的功能, 而你希望不通过额外的包装直接使用这个库的 API, 这种方式就并不理想了. 而且, 这种方式要求共通声明必须以接口的形式来表达, 而这并不能满足所有可能的使用场景.

作为一种替代的方案, Kotlin 提供了 *预期声明与实际声明(expected and actual declaration)* 机制. 通过这种机制, *common* 模块可以定义 *预期声明(expected declaration)*, 而 *platform* 模块则提供与预期声明相对应的 *实际声明(actual declaration)*. 为了理解这种机制的工作原理, 我们先来看一个示例程序. 这段代码是一个 *common* 模块的一部分:

```
package org.jetbrains.foo

expect class Foo(bar: String) {
    fun frob()
}

fun main() {
    Foo("Hello").frob()
}
```

下面是对应的 JVM 模块:

```
package org.jetbrains.foo

actual class Foo actual constructor(val bar: String) {
    actual fun frob() {
        println("Frobbing the $bar")
    }
}
```

上面的示例演示了几个要点:

- *common* 模块中的预期声明, 以及它与它对应的实际声明, 总是拥有完全相同的完整限定名(fully qualified name).
- 预期声明使用 `expect` 关键字进行标记; 实际声明使用 `actual` 关键字进行标记.
- 与预期声明中的任何一个部分对应的实际声明, 都必须标记为 `actual`.
- 预期声明绝不包含任何实现代码.

注意, 预期声明并不局限于接口和接口的成员. 在上面的示例中, 预期类有一个构造函数, 而且在共通代码中可以直接创建这个类的实例. 你也可以将 `expect` 标记符用在其他声明上, 包括顶层声明, 以及注解:

```
// Common
expect fun formatString(source: String, vararg args: Any): String

expect annotation class Test

// JVM
actual fun formatString(source: String, vararg args: Any) =
    String.format(source, *args)

actual typealias Test = org.junit.Test
```

编译器会保证 *common* 模块中的每一个预期声明, 在所有实现这个 *common* 模块的 *platform* 模块中, 都存在对应的实际声明, 如果缺少实际声明, 则会报告错误. IDE 提供了工具, 可以帮助你创建缺少的实际声明.

如果你已经有了一个依赖于平台的库, 希望在共通代码中使用, 同时对其他平台则提供你自己的实现, 这时你可以为已存在的类定义一个类型别名, 以此作为实际声明:

```
expect class AtomicRef<V>(value: V) {
    fun get(): V
    fun set(value: V)
    fun getAndSet(value: V): V
    fun compareAndSet(expect: V, update: V): Boolean
}

actual typealias AtomicRef<V> = java.util.concurrent.atomic.AtomicReference<V>
```

# 使用 Gradle 编译跨平台项目

⚠️ 跨平台项目是 Kotlin 1.2 和 1.3 版中的实验性特性. 本文档描述的所有语言特性和工具特性, 在未来的 Kotlin 版本中都有可能发生变更.

本文档介绍 [Kotlin 跨平台项目](#) 的结构, 并解释如何使用 Gradle 来配置和编译跨平台项目.

## 目录

- [项目结构](#)
- [设置跨平台项目](#)
- [Gradle Plugin](#)
- [设置编译目标](#)
  - [支持的平台](#)
  - [配置编译任务](#)
- [配置源代码集](#)
  - [源代码集之间的关联](#)
  - [添加依赖项目](#)
  - [语言设置](#)
- [默认的项目结构](#)
- [运行测试程序](#)
- [发布跨平台的库](#)
  - [元数据发布模式\(metadata publishing mode\)](#)
  - [对编译目标消除歧义](#)
- [Android 支持](#)
  - [发布 Android 库](#)
- [使用 Kotlin/Native 编译目标](#)
  - [编译最终的原生二进制文件](#)

## 项目结构

Kotlin 跨平台项目由以下几个编译模块组成:

- [编译目标](#) 是整个编译的一部分, 它负责在某个特定的目标平台上编译, 测试, 并打包整个软件. 因此, 一个跨平台项目通常会包括多个编译目标.
- 对每个编译目标的编译会包括对 Kotlin 源代码进行一次或多次编译. 也就是说, 一个编译目标可以包含一个或多个 [编译任务](#). 比如, 一个针对产品源代码的编译任务, 以及一个针对测试源代码的编译任务.
- Kotlin 源代码通过 [源代码集](#) 进行组织. 除 Kotlin 源代码文件和资源文件外, 每个源代码集还包含它自己的依赖项. 多个源代码集之间相互存在 “*依赖(depends on)*” 关系, 构成一个层级结构. 源代码集本身是平台无关的, 但如果它只针对单一的平台进行编译, 那么其中可以包含平台相关的代码和依赖项.

每个编译任务都有自己的默认源代码集, 也就是这个编译任务所使用的源代码和依赖项. 默认源代码集还可以通过 “*依赖*” 关系, 将其他源代码集引入到编译任务中.

下图是一个针对 JVM 和 JS 平台的项目结构:



这个项目中存在两个编译目标, `jvm` 和 `js`, 每个编译目标都会编译产品源代码和测试源代码, 其中有部分源代码是共用的. 为了实现这样的结果, 我们只需要创建这两个编译目标即可, 并不需要额外地配置编译任务和源代码集, 编译任务和源代码集都是针对各个编译目标 [默认创建](#) 的.

在上面的示例中, JVM 编译目标的产品源代码集由它的 `main` 编译任务来编译, 因此包含了 `jvmMain` 和 `commonMain` 源代码集中的源代码和依赖项 (因为这两个源代码集之间存在 [依赖](#) 关系):



在这个例子中, `commonMain` 源代码集的共用源代码定义了一些 API, 而 `jvmMain` 源代码集则为这些 API 提供了 [平台相关的实现](#). 这是一种灵活的方式, 我们可以在不同的平台之间共用一部分代码, 并在需要的时候为不同的平台提供各自的实现代码.

在后面的章节中, 会更详细地介绍这些概念, 并解释如何使用 DSL 来对它们进行配置.

## 设置跨平台项目

你可以在 IDE 中创建一个跨平台项目, 方法是在 New Project 对话框中选择 “Kotlin” 中的某个跨平台项目模板.

比如, 如果你选择 “Kotlin (Multiplatform Library)”, 会创建一个 Library 项目, 其中包含 3 个 [编译目标](#), 一个是 JVM, 一个是 JS, 另一个是你当前正在使用的本地平台. 这些设置保存在 `build.gradle` 脚本中, 如下:

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.21'
}

repositories {
    mavenCentral()
}

kotlin {
    jvm() // 创建 JVM 编译平台, 使用默认名称 'jvm'
    js() // 创建 JS 编译平台, 使用默认名称 'js'
    mingwX64("mingw") // 创建 Windows (MinGW X64) 编译平台, 名称指定为 'mingw'

    sourceSets { /* ... */ }
}
```

```
plugins {
    kotlin("multiplatform") version "1.3.21"
}

repositories {
    mavenCentral()
}

kotlin {
    jvm() // 创建 JVM 编译平台, 使用默认名称 'jvm'
    js() // 创建 JS 编译平台, 使用默认名称 'js'
    mingwX64("mingw") // 创建 Windows (MinGW X64) 编译平台, 名称指定为 'mingw'

    sourceSets { /* ... */ }
}
```

这 3 个编译目标是使用预定义函数 `jvm()`, `js()`, 和 `mingwX64()` 来创建的, 这些函数都提供了 [默认配置](#), 对于每个 [支持的平台](#) 都提供了这种预定义的默认配置.

[源代码集](#) 以及它们的 [依赖项目](#) 配置如下:

```

plugins { /* ... */ }

kotlin {
    /* 编译目标的相关配置, 略 */

    sourceSets {
        commonMain {
            dependencies {
                implementation kotlin('stdlib-common')
            }
        }
        commonTest {
            dependencies {
                implementation kotlin('test-common')
                implementation kotlin('test-annotations-common')
            }
        }
    }

    // 针对 JVM 平台相关代码的默认源代码集, 以及依赖项.
    // 这里也可以使用 jvmMain { ... }:
    jvm().compilations.main.defaultSourceSet {
        dependencies {
            implementation kotlin('stdlib-jdk8')
        }
    }
    // JVM 平台相关的测试代码, 以及依赖项:
    jvm().compilations.test.defaultSourceSet {
        dependencies {
            implementation kotlin('test-junit')
        }
    }

    js().compilations.main.defaultSourceSet { /* ... */ }
    js().compilations.test.defaultSourceSet { /* ... */ }

    mingwX64('mingw').compilations.main.defaultSourceSet { /* ... */ }
    mingwX64('mingw').compilations.test.defaultSourceSet { /* ... */ }
    }
}

```

```

plugins { /* ... */ }

kotlin {
    /* 编译目标的相关配置, 略 */

    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation(kotlin("stdlib-common"))
            }
        }
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test-common"))
                implementation(kotlin("test-annotations-common"))
            }
        }
    }

    // 针对 JVM 平台相关代码的默认源代码集, 以及依赖项:
    jvm().compilations["main"].defaultSourceSet {
        dependencies {
            implementation(kotlin("stdlib-jdk8"))
        }
    }
    // JVM 平台相关的测试代码, 以及依赖项:
    jvm().compilations["test"].defaultSourceSet {
        dependencies {
            implementation(kotlin("test-junit"))
        }
    }

    js().compilations["main"].defaultSourceSet { /* ... */ }
    js().compilations["test"].defaultSourceSet { /* ... */ }

    mingwX64("mingw").compilations["main"].defaultSourceSet { /* ... */ }
    mingwX64("mingw").compilations["test"].defaultSourceSet { /* ... */ }
}

```

在上面配置的编译目标中, 对产品源代码和测试源代码都有 [默认的源代码集名称](#). 对于所有的编译目标, 源代码集 `commonMain` 和 `commonTest` 分别包含在产品编译和测试编译之中. 注意, 共通源代码集 `commonMain` 和 `commonTest` 只依赖共通库, 特定平台相关的库由这个平台相关的源代码依赖.

## Gradle Plugin

Kotlin 跨平台项目需要使用 Gradle 4.7 以上版本, 不支持旧的 Gradle 版本.

如果想要从 Gradle 项目从头开始设置一个跨平台项目, 首先, 请在你的项目中使用 `kotlin-multiplatform` plugin, 方法是在 `build.gradle` 文件的最头部中添加以下代码:

```

plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.21'
}

```

```

plugins {
    kotlin("multiplatform") version "1.3.21"
}

```

这样会在最顶层添加一个 `kotlin` 扩展. 然后你可以在编译脚本中引用它, 以便:

- 对跨平台项目 [设置编译目标](#) (因为默认不会创建编译目标);
- [设置源代码集](#) 以及以及它们的 [依赖项目](#);

## 设置编译目标

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.

各个编译目标可以共用一部分源代码, 也可以包含平台相关的代码.

由于平台是不同的, 因此编译目标不过不同的方式编译, 而且各自带有平台专有的一些设定信息. Gradle plugin 对支持的平台带有很多预定义的设置.

要创建一个编译目标, 我们只需要使用某个预定义的函数, 函数名称对应于编译的目标平台, 另外还可以通过参数指定编译目标的名称, 并指定配置代码:

```
kotlin {
    jvm() // 创建一个 JVM 编译目标, 使用默认名称 'jvm'
    js("nodejs") // 创建一个 JS 编译目标, 指定编译名称为 'nodejs'

    linuxX64("linux") {
        /* 可以在这里对 'linux' 编译目标指定更多的设定项目 */
    }
}
```

如果编译目标已经存在, 那么函数直接返回已存在的编译目标. 因此可以利用这个特性来对已经存在的编译目标进行配置:

```
kotlin {
    /* ... */

    // 设置 'jvm6' 编译目标的属性:
    jvm("jvm6").attributes { /* ... */ }
}
```

注意, 调用这些函数时, 目标平台和编译目标的名称都必须指定正确: 如果使用 `jvm('jvm6')` 创建了一个编译目标, 那么调用 `jvm()` 会创建另一个编译目标 (使用默认名称 `jvm`). 如果使用目标平台A对应的函数创建了一个编译目标, 然后再使用另一个目标平台B对应的函数来创建相同名称的编译目标, 那么会发生错误.

使用默认配置创建的编译目标会被加入到 `kotlin.targets` 集合中, 可以使用名称在这个集合内查找编译目标, 也可以使用这个集合对所有的编译目标进行配置:

```
kotlin {
    jvm()
    js("nodejs")

    println(targets.names) // 打印结果为: [jvm, metadata, nodejs]

    // 对所有的编译目标进行配置, 包括将来添加的编译目标:
    targets.all {
        compilations["main"].defaultSourceSet { /* ... */ }
    }
}
```

要使用多个预定义配置动态地创建编译目标, 或者访问编译目标, 可以使用 `targetFromPreset` 函数, 它接受的参数是一个预定义配置(包含在 `kotlin.presets` 集合中), 另外还可以接受可选的参数, 编译目标名称, 以及配置代码段.

比如, 要对 Kotlin/Native 支持的所有目标平台(参见后文)分别创建编译目标, 可以使用这样的代码:



```
kotlin {
    presets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTargetPreset).each {
        targetFromPreset(it) {
            /* 对创建的某个编译目标进行配置 */
        }
    }
}
```

```
import org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTargetPreset

/* ... */

kotlin {
    presets.withType<KotlinNativeTargetPreset>().forEach {
        targetFromPreset(it) {
            /* */
        }
    }
}
```

## 支持的平台

Kotlin 支持以下目标平台, 针对每个目标平台, 都分别提供了预定义配置, 可以通过上面的例子那样使用:

- `jvm` 用于 Kotlin/JVM 平台. 注意: `jvm` 编译目标不会编译 Java 源代码;
- `js` 用于 Kotlin/JS 平台;
- `android` 用于 Android 应用程序和库. 注意, 在创建编译目标之前, 需要应用 Android Gradle plugin;
- Kotlin/Native 平台的预定义编译目标 (详情请参见后面的 [注意](#)):
  - `androidNativeArm32` 和 `androidNativeArm64` 用于 Android NDK 平台;
  - `iosArm32`, `iosArm64`, `iosX64` 用于 iOS 平台;
  - `linuxArm32Hfp`, `linuxMips32`, `linuxMipsel32`, `linuxX64` 用于 Linux 平台;
  - `macosX64` 用于 MacOS 平台;
  - `mingwX64` 用于 Windows 平台;
  - `wasm32` 用于 WebAssembly 平台.

注意, Kotlin/Native 的一部分编译目标需要针对一个 [适当的目标机器](#) 来编译.

某些目标平台可能需要额外的配置. 比如 Android 和 iOS 平台, 请参见教程 [跨平台项目: iOS 与 Android](#).

## 配置编译任务

编译目标的编译需要进行一次或多次 Kotlin 编译任务. 每次 Kotlin 编译任务可能是为了不同的目的(比如, 编译产品代码, 或测试代码), 也可能处理不同的 [源代码集](#). 可以在 DSL 内访问编译目标中的各个 Kotlin 编译任务, 比如, 可以得到编译任务, 配置 [Kotlin 编译器选项](#), 得到依赖项目文件名, 以及编译输出路径:

```

kotlin {
    jvm {
        compilations.main.kotlinOptions {
            // 对 'main' 编译任务设置 Kotlin 编译器选项:
            jvmTarget = "1.8"
        }

        compilations.main.compileKotlinTask // 得到 Kotlin 编译任务 'compileKotlinJvm'
        compilations.main.output // 得到 main 编译任务的输出路径
        compilations.test.runtimeDependencyFiles // 得到 test 编译任务的运行时 classpath
    }

    // 对所有编译目标的所有编译任务进行配置:
    targets.all {
        compilations.all {
            kotlinOptions {
                allWarningsAsErrors = true
            }
        }
    }
}

```

```

kotlin {
    jvm {
        val main by compilations.getting {
            kotlinOptions {
                // 对 'main' 编译任务设置 Kotlin 编译器选项:
                jvmTarget = "1.8"
            }

            compileKotlinTask // 得到 Kotlin 编译任务 'compileKotlinJvm'
            output // 得到 main 编译任务的输出路径
        }

        compilations["test"].runtimeDependencyFiles // 得到 test 编译任务的运行时 classpath
    }

    // 对所有编译目标的所有编译任务进行配置:
    targets.all {
        compilations.all {
            kotlinOptions {
                allWarningsAsErrors = true
            }
        }
    }
}

```

每个编译任务都存在一个默认 [源代码集](#), 默认源代码集是自动创建的, 用来保存这个编译任务独有的源代码文件和依赖项. 对于编译目标 `bar` 的编译任务 `foo`, 默认源代码集的名称为 `barFoo`. 也可以通过编译任务的 `defaultSourceSet` 属性访问默认源代码集:

```

kotlin {
    jvm() // 创建 JVM 编译目标, 使用默认名称 'jvm'

    sourceSets {
        // 'jvm' 编译目标的 'main' 编译任务的默认源代码集:
        jvmMain {
            /* ... */
        }
    }

    // 或者, 也可以使用编译目标的编译任务来访问对应的默认源代码集:
    jvm().compilations.main.defaultSourceSet {
        /* ... */
    }
}

```

```

kotlin {
    jvm() // 创建 JVM 编译目标, 使用默认名称 'jvm'

    sourceSets {
        // 'jvm' 编译目标的 'main' 编译任务的默认源代码集:
        val jvmMain by getting {
            /* ... */
        }
    }

    // 或者, 也可以使用编译目标的编译任务来访问对应的默认源代码集:
    jvm().compilations["main"].defaultSourceSet {
        /* ... */
    }
}

```

要得到一个编译任务的所有源代码集, 包括那些通过依赖关系添加的源代码集, 我们可以使用 `allKotlinSourceSets` 属性.

对于某些特殊的场景, 可能需要创建自定义的编译任务. 可以使用编译目标的 `compilations` 集合来实现. 注意, 对于所有的自定义编译任务, 依赖项需要手动设置, 而且自定义编译任务的输出文件的使用要由编译脚本的编写者自行实现. 比如, 假定我们要为 `jvm()` 编译目标的集成测试(Integration Test)创建自定义编译任务:

```

kotlin {
    jvm() {
        compilations.create('integrationTest') {
            defaultSourceSet {
                dependencies {
                    def main = compilations.main
                    // 使用 main 编译任务的编译时 classpath , 以及它的编译输出:
                    implementation(main.compileDependencyFiles + main.output.classesDirs)
                    implementation kotlin("test-junit")
                    /* ... */
                }
            }
        }

        // 创建测试任务, 运行这个编译任务编译产生的测试类:
        tasks.create('jvmIntegrationTest', Test) {
            // 运行测试程序, 使用的 classpath 包括: 编译期依赖项(含 'main'), 运行期依赖项, 以及当前编译任务的输出:
            classpath = compileDependencyFiles + runtimeDependencyFiles + output.allOutputs

            // 只在当前编译任务的输出中寻找需要运行的测试类:
            testClassesDirs = output.classesDirs
        }
    }
}

```

```

kotlin {
    jvm() {
        compilations {
            val main by getting

            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
                        // 使用 main 编译任务的编译时 classpath , 以及它的编译输出:
                        implementation(main.compileDependencyFiles + main.output.classesDirs)
                        implementation(kotlin("test-junit"))
                        /* ... */
                    }
                }
            }

            // 创建测试任务, 运行这个编译任务编译产生的测试类:
            tasks.create<Test>("integrationTest") {
                // 运行测试程序, 使用的 classpath 包括: 编译期依赖项(含 'main'), 运行期依赖项, 以及当前编译任务的输出:
                classpath = compileDependencyFiles + runtimeDependencyFiles + output.allOutputs

                // 只在当前编译任务的输出中寻找需要运行的测试类:
                testClassesDirs = output.classesDirs
            }
        }
    }
}

```

还需要注意, 默认情况下, 自定义编译任务的默认源代码集不会依赖于 `commonMain` 和 `commonTest` .

## 配置源代码集

Kotlin 源代码集是指一组 Kotlin 源代码, 相关的资源文件, 依赖项目, 以及语言设定, 在一个或多个 [编译目标](#) 的 Kotlin 编译中, 将会使用到这些源代码集.


源代码集并不一定是平台相关的, 也并不一定是多平台共用代码; 源代码集能够包含什么样的内容由它的使用方法来决定: 源代码集如果被加入到多个编译任务, 那么它只能使用这些编译任务共通支持的语言特性和依赖项, 而只被单个编译目标使用的源代码集, 则可以使用平台相关的依赖项, 其中的源代码也可以使用编译目标对应的目标平台上独有的语言特性。

有些源代码集是默认创建并配置好的: `commonMain`, `commonTest`, 以及各个编译任务默认的源代码集. 详情请参见 [默认的项目布局](#).

源代码集在 `kotlin { ... }` 扩展的 `sourceSets { ... }` 代码段中进行设置:

```
kotlin {
    sourceSets {
        foo { /* ... */ } // 使用名称 'foo' 创建或配置一个源代码集
        bar { /* ... */ }
    }
}
```

```
kotlin {
    sourceSets {
        val foo by creating { /* ... */ } // 使用名称 'foo' 创建一个新的源代码集
        val bar by getting { /* ... */ } // 配置一个已存在的源代码集 'bar'
    }
}
```

 注意: 创建一个源代码集不会将它关联到任何的编译目标. 有些源代码集 [预定义的](#), 因此默认会编译它. 但是, 自定义的源代码集需要明确地指定它属于哪些编译任务. 详情请参见: [源代码集之间的关联](#).

源代码集的名称是大小写敏感的. 使用名称来引用一个默认的源代码集时, 一定要注意源代码集的名称前缀要与编译目标的名称一致, 比如, 编译目标 `iosX64` 的主源代码集名称是 `iosX64Main`.

源代码集本身是与平台无关的, 但是如果它只编译到单个平台, 我们可以认为它是与这个平台相关的. 因此, 源代码集可以包括各个平台之间共用的共通代码, 也可以包含平台相关的代码.

某个源代码集都有一个默认的源代码目录, 用于存放 Kotlin 源代码文件: `src/<source set name>/kotlin`. 要向源代码集添加 Kotlin 源代码目录和资源, 请使用它的 `kotlin`, `resources` `SourceDirectorySet` 属性:

```
kotlin {
    sourceSets {
        commonMain {
            kotlin.srcDir('src')
            resources.srcDir('res')
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            kotlin.srcDir("src")
            resources.srcDir("res")
        }
    }
}
```

## 源代码集之间的关联

Kotlin 源代码集可以通过 ‘[依赖](#)’ 关系发生连接, 如果源代码集 `foo` 依赖于另一个源代码集 `bar`, 那么:

- 如果针对某个编译目标需要编译 `foo`, 那么 `bar` 也会被编译, 并且输出为同样的编译目标形式, 比如 JVM 平台的 class 文件, 或者 JS 代码;
- `foo` 中的源代码可以 ‘看见’ `bar` 中的声明, 包括 `internal` 声明, 也可以 ‘看见’ `bar` 中的 [依赖项目](#), 即使是标记为 `implementation` 的依赖项目也可以看见;
- 对于 `bar` 中的预期声明, `foo` 可以包含对应的 [平台相关的实现代码](#);
- `bar` 中的资源会与 `foo` 中的资源一起处理, 复制;
- `foo` 与 `bar` 的 [语言设定](#) 应该保持一致;

源代码集之间的循环依赖是禁止的.

源代码集之间的这些关联关系可以使用源代码集的 DSL 来定义:

```
kotlin {
    sourceSets {
        commonMain { /* ... */ }
        allJvm {
            dependsOn commonMain
            /* ... */
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting { /* ... */ }
        val allJvm by creating {
            dependsOn(commonMain)
            /* ... */
        }
    }
}
```

除[默认的源代码集](#)之外, 自定义创建的源代码集需要明确地加入到依赖关系中, 然后才可以在其他源代码中使用它, 而且更重要的, 比如加入到依赖关系中, 然后它才会被编译. 通常来说, 可以使用 `dependsOn(commonMain)` 或 `dependsOn(commonTest)` 语句, 某些默认的平台相关源代码集需要直接或者间接地依赖到自定义源代码集:

```

kotlin {
    mingwX64()
    linuxX64()

    sourceSets {
        // 自定义源代码集, 包括针对这两个编译目标的测试程序
        desktopTest {
            dependsOn commonTest
            /* ... */
        }
        // 让 'windows' 的默认测试源代码集依赖于 'desktopTest'
        mingwX64().compilations.test.defaultSourceSet {
            dependsOn desktopTest
            /* ... */
        }
        // 对其他编译目标也做同样的设定:
        linuxX64().compilations.test.defaultSourceSet {
            dependsOn desktopTest
            /* ... */
        }
    }
}

```

```

kotlin {
    mingwX64()
    linuxX64()

    sourceSets {
        // 自定义源代码集, 包括针对这两个编译目标的测试程序
        val desktopTest by creating {
            dependsOn(getByName("commonTest"))
            /* ... */
        }
        // 让 'windows' 的默认测试源代码集依赖于 'desktopTest'
        mingwX64().compilations["test"].defaultSourceSet {
            dependsOn(desktopTest)
            /* ... */
        }
        // 对其他编译目标也做同样的设定:
        linuxX64().compilations["test"].defaultSourceSet {
            dependsOn(desktopTest)
            /* ... */
        }
    }
}

```

## 添加依赖项目

要对源代码集添加依赖项目, 请使用源代码集 DSL 的 `dependencies { ... }` 语句段. 支持 4 种类型的依赖:

- `api` 依赖, 同时用于编译期和运行期, 而且会被导出给你的库的使用者. 如果在当前模块的 Public API 中使用了从依赖库得到的任何类型, 那么这个类型所属的库必须是 `api` 类型的依赖项目;
- `implementation` 依赖, 同时用于编译期和运行期, 但只供当前模块使用, 而不会暴露给依赖于本模块的其他模块. 当前模块的内部实现逻辑所需要的依赖项目, 应该使用 `implementation` 类型依赖. 如果一个模块是一个最终应用程序, 本身不对外公布 API, 那么它应该使用 `implementation` 依赖, 而不是 `api` 依赖.
- `compileOnly` 依赖, 只用于当前模块的编译期, 除此之外, 对于当前模块的运行期, 以及依赖本模块的其他模块的编译期, 这些依赖都不可使用. 对于运行期使用第三方实现的 API, 应该使用这种依赖.

— `runtimeOnly` 依赖, 只在运行期可用, 对任何模块的编译期, 这些依赖都不可使用.

依赖项目以源代码集为单位分别指定, 如下:

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                api 'com.example:foo-metadata:1.0'
            }
        }
        jvm6Main {
            dependencies {
                api 'com.example:foo-jvm6:1.0'
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                api("com.example:foo-metadata:1.0")
            }
        }
        val jvm6Main by getting {
            dependencies {
                api("com.example:foo-jvm6:1.0")
            }
        }
    }
}
```

注意, 为了使 IDE 能够正确解析共通代码的依赖项目, 除了在平台相关源代码集中指定平台相关的依赖项目之外, 共通代码的源代码集还需要指定对应的 Kotlin metadata 依赖项. 通常, 使用公开发布的库时, 需要使用带 `-common` 后缀(比如 `kotlin-stdlib-common`) 或 `-metadata` 后缀的依赖项 (除非它发布时带有 Gradle metadata, 详情请参见下文).

然而, 使用 `project('...')` 依赖到另一个跨平台项目时, 就可以自动解析到正确的依赖项目. 因此, 只需要在一个源代码集的依赖项中使用一个 `project('...')` 依赖就可以了, 只要这个被依赖的项目中包含了兼容的编译目标, 那么包含这个源代码集的编译都会依赖到这个项目中正确的平台相关依赖项:

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                // 所有包含源代码集 'commonMain' 的编译任务,
                // 如果存在这个依赖项, 都会被解析为正确的平台相关依赖项:
                api project(':foo-lib')
            }
        }
    }
}
```



```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                // 所有包含源代码集 'commonMain' 的编译任务,
                // 如果存在这个依赖项, 都会被解析为正确的平台相关依赖项:
                api(project(":foo-lib"))
            }
        }
    }
}
```

同样的, 如果使用实验性的 [Gradle metadata 发布模式](#) 发布一个跨平台库, 并且使用这个库的项目也设置为使用 metadata, 那么只需要对共通源代码集指定一次依赖项就足够了. 否则, 对每一个平台相关的源代码集, 除了需要依赖到共通模块之外, 还需要依赖库中对应平台的模块, 就象上面的示例那样.

另外一种方式是, 在最顶层使用 Gradle 内建的 DSL, 通过 `<sourceSetName><DependencyKind>` 格式的配置名称来指定依赖项目:

```
dependencies {
    commonMainApi 'com.example:foo-common:1.0'
    jvm6MainApi 'com.example:foo-jvm6:1.0'
}
```

```
dependencies {
    "commonMainApi"("com.example:foo-common:1.0")
    "jvm6MainApi"("com.example:foo-jvm6:1.0")
}
```

Gradle 的一些内建依赖项, 比如 `gradleApi()`, `localGroovy()`, 以及 `gradleTestKit()`, 在源代码集的依赖项 DSL 内是不可用的. 但是, 你可以在最顶层的依赖项配置代码段中添加它们, 方法和上面的例子一样.

对 Kotlin 模块(比如 `kotlin-stdlib` 或 `kotlin-reflect`)的依赖项, 可以使用 `kotlin("stdlib")` 来添加, 这是 `"org.jetbrains.kotlin:kotlin-stdlib"` 的简写.

## 语言设置

源代码集的语言设置可以通过以下方式指定:

```
kotlin {
    sourceSets {
        commonMain {
            languageSettings {
                languageVersion = '1.3' // 可指定值是: '1.0', '1.1', '1.2', '1.3'
                apiVersion = '1.3' // 可指定值是: '1.0', '1.1', '1.2', '1.3'
                enableLanguageFeature('InlineClasses') // 语言特性名称
                useExperimentalAnnotation('kotlin.ExperimentalUnsignedTypes') // 注解的完整限定名
                progressiveMode = true // 默认为 false
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            languageSettings.apply {
                languageVersion = "1.3" // 可指定值是: '1.0', '1.1', '1.2', '1.3'
                apiVersion = "1.3" // 可指定值是: '1.0', '1.1', '1.2', '1.3'
                enableLanguageFeature("InlineClasses") // 语言特性名称
                useExperimentalAnnotation("kotlin.ExperimentalUnsignedTypes") // 注解的完整限定名
                progressiveMode = true // 默认为 false
            }
        }
    }
}
```

也可以一次性指定所有源代码集的语言设置:

```
kotlin.sourceSets.all {
    languageSettings.progressiveMode = true
}
```

源代码集的语言设置会影响到 IDE 解析源代码的方式. 由于目前的限制, 在 Gradle 编译中, 只有编译任务的默认源代码集的语言设置会被使用, 并被应用于这个编译任务相关的所有源代码.

在存在依赖关系的源代码集之间, 会检查语言设置的一致性. 举例来说, 如果源代码集 `foo` 依赖于 `bar`:

- `foo` 的 `languageVersion` 设置要高于或等于 `bar`;
- 如果 `bar` 启用了不稳定的语言特征, 那么所有这些语言特征在 `foo` 中也应该启用 (但对 bug 修正的语言特征没有这种限制);
- 如果 `bar` 使用了实验性的注解, 那么所有这些注解在 `foo` 中也应该使用;
- `apiVersion`, bug 修正的语言特性, 以及 `progressiveMode` 可以任意设置;

## 默认的项目结构

默认情况下, 每个项目包含两个源代码集, `commonMain` 和 `commonTest`, 你可以将所有目标平台共用的共通代码放在这两个源代码集中. 这两个源代码集将会被分别添加到所有的产品编译和测试编译中.

然后, 只要添加一个编译目标, 默认就会为它创建以下编译任务:

- `main` 和 `test` 编译任务, 针对 JVM, JS, 以及 Native 目标;
- 对每个 Android 编译目标中的每个 [Android 编译变体](#), 创建一个编译任务;

对于每个编译任务, 都有一个默认的源代码集, 名称是 `<targetName><CompilationName>`. 这个默认的源代码集会参与编译, 因此应该使用它来编写平台相关的代码, 以及指定平台相关的依赖项, 还可以通过它来添加源代码集之间的'依赖', 来将其他源代码集加入到编译任务中. 比如, 如果一个项目包含 `jvm6` (JVM) 和 `nodejs` (JS) 编译目标, 那么将包括以下源代码集: `commonMain`, `commonTest`, `jvm6Main`, `jvm6Test`, `nodejsMain`, `nodejsTest`.

大多数使用场景只需要通过默认的源代码集就可以解决了, 不必使用自定义的源代码集.

每个源代码集的默认设置是, Kotlin 源代码放在 `src/<sourceSetName>/kotlin` 文件夹下, 资源文件放在 `src/<sourceSetName>/resources` 文件夹下.

在 Android 项目中, 还会对每个 [Android 源代码集](#) 创建 Kotlin 源代码集. 如果 Android 编译目标的名称是 `foo`, 那么 Android 源代码集 `bar` 会产生一个对应的 Kotlin 源代码集, 名为 `fooBar`. 但是 Kotlin 编译任务可以从所有的 `src/bar/java`, `src/bar/kotlin`, 和 `src/fooBar/kotlin` 目录中得到 Kotlin 源代码. 而 Java 源代码只会从 `src/bar/java` 目录得到.

## 运行测试程序

目前在 JVM, Android, Linux, Windows 和 macOS 平台上, 默认支持在 Gradle 编译脚本中运行测试程序; 对于 JS 以及其他 Kotlin/Native 编译目标, 需要手工配置, 以便使用适当的环境运行来测试程序, 比如使用模拟器, 或者测试框架.

对每个可以进行测试的编译目标, 都会生成测试任务, 名称是 `<targetName>Test`. 执行 `check` 任务, 可以对所有的编译目标运行测试程序.

由于 `commonTest` 任务的 [默认源代码集](#) 被添加到了所有的测试任务中, 因此可以在这里添加那些需要在所有的目标平台上运行的测试程序和测试工具.


跨平台测试可以使用 [kotlin.test API](#). 对 `commonTest` 添加 `kotlin-test-common` 和 `kotlin-test-annotations-common` 依赖项, 就可以在共通的测试代码中使用断言函数, 比如 `kotlin.test.assertTrue(...)`, 以及 `@Test` / `@Ignore` / `@BeforeTest` / `@AfterTest` 注解.

对于 JVM 编译目标, 应该添加 `kotlin-test-junit` 或 `kotlin-test-testng` 依赖项, 来使用对应的断言函数和注解.

对于 Kotlin/JS 编译目标, 应该添加 `kotlin-test-js` 作为测试依赖项. 目前来说, 会创建 Kotlin/JS 的测试任务, 但默认不会运行; 需要手工配置这些测试任务, 以便使用适当的 JavaScript 测试框架来运行来测试程序.

Kotlin/Native 编译目标不需要添加额外的测试依赖项, `kotlin.test` API 的实现是默认附带的.

## 发布跨平台的库

 跨平台库的作者可以定义一组目标平台, 这些目标平台都应该提供这个库所需要的所有平台相关实现. 作为库的使用者, 不允许对一个跨平台的库添加新的目标平台.

通过跨平台项目编译得到的库, 可以使用 [maven-publish Gradle plugin](#) 发布到 Maven 仓库, 这个 plugin 可以通过以下方法引入:

```
plugins {  
    /* ... */  
    id("maven-publish")  
}
```

引入这个 plugin 之后, 会对所有能够在当前机器上编译的目标平台创建默认的发布任务. 但是需要对当前项目设置 `group` 和 `version` 属性:

```
plugins { /* ... */ }  
  
group = "com.example.my.library"  
version = "0.0.1"
```

但是, Android 库文件编译目标, 默认不会发布任何库文件, 需要额外的步骤来配置发布任务, 详情请参见 [发布 Android 库](#).

默认的库文件 ID 使用的命名方式是 `<projectName>-<targetNameToLowerCase>`, 比如对 `sample-lib` 项目的 `nodejs` 编译目标, 发布的库文件名为 `sample-lib-nodejs`.

默认情况下, 对每个发布任务, 除库文件之外, 还会包含源代码 JAR 文件. 源代码 JAR 文件包含编译目标的 `main` 编译任务中用到的源代码文件. 如果你还需要发布文档库文件 (比如一个 Javadoc JAR), 你需要手动配置它的编译脚本, 并把它作为一个库文件添加到相关的发布任务中, 详情请参见下文.

此外还会默认创建一个额外的发布任务, 名为 “metadata”, 其中包含序列化后的 Kotlin 声明信息, 将被 IDE 用来分析跨平台库.

可以修改 Maven 仓库中的发布位置, 也可以对发布任务添加新的库文件, 方法是在 `targets { ... }` 代码段中, 或者使用 `publishing { ... }` DSL, 进行如下设置:

```

kotlin {
    jvm('jvm6') {
        mavenPublication { // 对 'jvm6' 编译目标设置发布任务
            // 默认的库文件 ID 是 'foo-jvm6', 我们需要修改它:
            artifactId = 'foo-jvm'
            // 添加一个文档 JAR 库文件 (它应该是一个自定义编译任务):
            artifact(jvmDocsJar)
        }
    }
}

// 或者也可以使用 `publishing { ... }` DSL 来配置发布任务:
publishing {
    publications {
        jvm6 { /* 对 'jvm6' 编译目标设置发布任务 */ }
        metadata { /* 对 Kotlin metadata 设置发布任务 */ }
    }
}

```

```

kotlin {
    jvm("jvm6") {
        mavenPublication { // 对 'jvm6' 编译目标设置发布任务
            // 默认的库文件 ID 是 'foo-jvm6', 我们需要修改它:
            artifactId = "foo-jvm"
            // 添加一个文档 JAR 库文件 (它应该是一个自定义编译任务):
            artifact(jvmDocsJar)
        }
    }
}

// 或者也可以使用 `publishing { ... }` DSL 来配置发布任务:
publishing {
    publications.withType<MavenPublication>().apply {
        val jvm6 by getting { /* 对 'jvm6' 编译目标设置发布任务 */ }
        val metadata by getting { /* 对 Kotlin metadata 设置发布任务 */ }
    }
}

```

Kotlin/Native 库文件的组装需要在多个目标平台上分别进行编译, 因此, 如果一个跨平台库包含 Kotlin/Native 编译目标, 那么它的发布任务就同样需要在多台主机上运行. 有些模块可能会在多个平台上编译多次(比如 JVM, JS, Kotlin metadata, WebAssembly), 为了避免重复发布这样的模块, 这些模块的发布任务可以配置为有条件执行, 比如:

```

kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()

    // 在 Linux 环境运行时, 会收到命令行参数 ` -PisLinux=true `,
    // 因此这些编译目标只会通过 Linux 环境的编译器发布.
    // 注意, 这里也包含了 Kotlin metadata 编译目标.
    // mingwX64() 编译目标会被自动跳过, 因为与 Linux 平台的编译任务不兼容.
    configure([targets["metadata"], jvm(), js()]) {
        mavenPublication { linuxOnlyPublication ->
            tasks.withType(AbstractPublishToMaven).all {
                onlyIf {
                    publication != linuxOnlyPublication || findProperty("isLinux") == "true"
                }
            }
        }
    }
}

```

```

kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()


    // 在 Linux 环境运行时, 会收到命令行参数 ` -PisLinux=true `,
    // 因此这些编译目标只会通过 Linux 环境的编译器发布.
    // 注意, 这里也包含了 Kotlin metadata 编译目标.
    // mingwX64() 编译目标会被自动跳过, 因为与 Linux 平台的编译任务不兼容.
    configure(listOf(metadata(), jvm(), js())) {
        mavenPublication {
            val linuxOnlyPublication = this@mavenPublication
            tasks.withType<AbstractPublishToMaven>().all {
                onlyIf {
                    publication != linuxOnlyPublication || findProperty("isLinux") == "true"
                }
            }
        }
    }
}

```

### 元数据发布模式(metadata publishing mode) (实验性功能)

在根项目的 `settings.gradle` 文件中添加 `enableFeaturePreview("GRADLE_METADATA")` 设置项, 可以对库的发布和依赖计算启用 Gradle metadata 模式, 这个功能目前还是实验性的.

使用 Gradle metadata 模式时, 会添加一个新的发布任务 `kotlinMultiplatform`, 这个任务会引用所有的编译目标的发布任务作为它的变体. 这个发布任务的默认库文件 ID 与项目名相同.

 Gradle metadata 发布模式是 Gradle 的一个实验性功能, 不保证向后兼容性. 如果一个库使用当前版本的 Gradle metadata 模式发布, 那么 Gradle 的未来版本可能会无法解析对这个库的依赖. 在这个功能稳定之前, 建议库的作者只使用这个模式来发布库的实验性的版本, 同时使用稳定的发布机制来发布库的正式版本.

如果一个库使用 Gradle metadata 模式发布, 并且库的使用这也启用了这个模式, 那么库的使用者可以在一个共通源代码集中只指定一次对这个库的依赖, 然后它的各平台的编译目标都可以自动选择到这个库在对应的平台上的依赖项目. 比如, 一个 `sample-lib` 库针对 JVM 和 JS 进行编译, 并使用 Gradle metadata 模式发布. 那么使用者只需要添加 `enableFeaturePreview("GRADLE_METADATA")` 设定项, 然后添加一次依赖项:

```
kotlin {
    jvm('jvm6')
    js('nodejs')

    sourceSets {
        commonMain {
            dependencies {
                // 这个依赖项会解析为适当的目标模块,
                // 比如, 对 JVM 平台会解析为 `sample-lib-jvm6`, 对 JS 平台会解析为 `sample-lib-js`:
                api 'com.example:sample-lib:1.0'
            }
        }
    }
}
```

```
kotlin {
    jvm("jvm6")
    js("nodejs")

    sourceSets {
        val commonMain by getting {
            dependencies {
                // 这个依赖项会解析为适当的目标模块,
                // 比如, 对 JVM 平台会解析为 `sample-lib-jvm6`, 对 JS 平台会解析为 `sample-lib-js`:
                api("com.example:sample-lib:1.0")
            }
        }
    }
}
```

## 对编译目标消除歧义

在跨平台的库中, 对单个目标平台有可能存在多个编译目标. 比如, 这些编译目标可能提供相同的 API, 但在运行期使用不同的库来实现, 比如可以选择不同的测试框架, 或日志输出框架.

但是, 对这样的跨平台的库的依赖可能会发生歧义, 会导致解析失败, 因为没有足够的信息决定应该选择哪个编译目标.

解决方法是, 对编译目标标记一个自定义属性, Gradle 在解析依赖项的过程中会使用这个属性. 但是, 必须在库的作者和使用者双方都进行同样的标记, 而且库的作者需要负责维护这个属性的名称和可选的值, 并将这些信息公开给使用者;

需要在库本身的项目和使用库的项目中都加入相同的自定义属性. 比如, 假设一个测试库, 它的两个编译目标分别支持 JUnit 和 TestNG. 库的作者需要向这两个编译目标都添加属性, 如下:

```
def testFrameworkAttribute = Attribute.of('com.example.testFramework', String)

kotlin {
    jvm('junit') {
        attributes.attribute(testingFrameworkAttribute, 'junit')
    }
    jvm('testng') {
        attributes.attribute(testingFrameworkAttribute, 'testng')
    }
}
```

```

val testFrameworkAttribute = Attribute.of("com.example.testFramework", String::class.java)

kotlin {
    jvm("junit") {
        attributes.attribute(testingFrameworkAttribute, "junit")
    }
    jvm("testng") {
        attributes.attribute(testingFrameworkAttribute, "testng")
    }
}

```

库的使用者可以只向发生依赖项歧义的编译目标添加相同的属性。

如果同样的依赖项歧义发生在自定义编译配置, 而不是 plugin 创建的编译配置, 你可以通过同样的方式向编译配置添加属性:

```

def testFrameworkAttribute = Attribute.of('com.example.testFramework', String)

configurations {
    myConfiguration {
        attributes.attribute(testFrameworkAttribute, 'junit')
    }
}

```

```

val testFrameworkAttribute = Attribute.of("com.example.testFramework", String::class.java)

configurations {
    val myConfiguration by creating {
        attributes.attribute(testFrameworkAttribute, "junit")
    }
}

```

另一种办法是, 如果依赖项的库文件发布时使用了 Gradle metadata 发布模式(实验性功能), 但是你仍然可以将单一的依赖项替换为特定平台上的无歧义的依赖项, 就好像并没有使用 Gradle metadata 那样. 这种方式下, 依赖项仍然会通过 Gradle metadata 模式发布, 因此使用者不会遇到依赖项歧义。

## Android 支持

Kotlin 跨平台项目提供了预定义的 `android` 编译目标, 支持 Android 平台. 创建 Android 编译目标需要在项目中手动适用某个 Android Gradle plugin, 比如 `com.android.application` 或 `com.android.library`. 对于每个 Gradle 子项目, 只能创建一个 Android 编译目标:

```

plugins {
    id("com.android.library")
    id("org.jetbrains.kotlin.multiplatform").version("1.3.21")
}

android { /* ... */ }

kotlin {
    android { // 创建 Android 编译目标
        // 如果需要, 在这里指定额外的配置信息
    }
}

```

```

plugins {
    id("com.android.library")
    kotlin("multiplatform").version("1.3.21")
}

android { /* ... */ }

kotlin {
    android { // 创建 Android 编译目标
        // 如果需要, 在这里指定额外的配置信息
    }
}

```

Android 编译目标会默认创建编译任务, 并绑定到 [Android 编译变体](#): 对每个编译变体, 都会创建一个与它相同名称的编译任务。

然后, 对于编译变体编译的每个 [Android 源代码集](#), 都会创建一个 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变体编译任务中。

默认的源代码集 `commonMain` 会被添加到所有的产品编译变体(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)的编译变体中。

也支持使用 [kapt](#) 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")
}

```

## 发布 Android 库

如果要在发布跨平台项目时包含 Android 库, 你需要 [设置库的发布任务](#), 并为 Android 库编译目标指定额外的配置信息。

Android 库文件默认不会发布。如果要发布各个 [Android 编译变体](#) 产生的库文件, 请在 Android 编译目标的脚本中指定需要发布的编译变体名称, 如下:

```

kotlin {
    android {
        publishLibraryVariants("release", "debug")
    }
}

```

上面的示例适用于没有产品风格(Product Flavor)的 Android 库。对于有产品风格(Product Flavor)的库, 变体名称中还需要包含产品风格名称, 比如 `fooBarDebug` 或 `fooBazRelease`。

注意, 如果库的使用者定义了某个编译变体, 但在库中不存在这个变体, 那么需要为编译变体指定 [匹配回退\(Matching Fallback\)](#)。比如, 如果一个库不包含, 或者没有发布, 名为 `staging` 的编译变体, 那么对于库的使用者, 如果使用了这个编译变体, 就需要提供一个匹配回退机制, 指定库发布的一个以上的编译变体, 作为后备选项:



```

android {
    buildTypes {
        staging {
            // ...
            matchingFallbacks = ['release', 'debug']
        }
    }
}

```

```

android {
    buildTypes {
        val staging by creating {
            // ...
            matchingFallbacks = listOf("release", "debug")
        }
    }
}

```

类似的, 如果库的使用者使用了自定义的产品风格(Product Flavor), 而在库中不存在这个产品风格(Product Flavor), 那么库的使用者同样需要提供匹配回退.

有一个选项, 可以将各个编译变体以产品风格为单位分组发布, 使得不同的编译变体的输出文件可以放在同一个模块内, 编译变体成为库文件 ID 中的一个分类符 (release 编译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, 那么不推荐使用模式, 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.

## 使用 Kotlin/Native 编译目标

注意, 某些 [Kotlin/Native 编译目标](#) 可能只能在适当的机器上编译:

- Linux 编译目标可能只能在 Linux 主机上编译;
- Windows 编译目标需要 Windows 主机;
- macOS 和 iOS 编译目标只能在 macOS 主机上编译;
- Android Native 编译目标需要 Linux 或 macOS 主机;

在编译时, 当前主机不能支持的编译目标会被忽略, 因此也不会发布. 库的作者可能希望根据库的目标平台的需要来设置编译脚本, 并通过不同的主机来发布.

## 编译最终的原生二进制文件

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 Native 编译目标的 `binaries` 属性. 这个属性值是二进制文件类型的列表, 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二进制文件类型.

注意, `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测试程序.

### 二进制编译输出的声明

有一组工厂方法可以用来声明 `binaries` 中的元素. 这些方法允许开发者指定输出什么类型的二进制文件, 并对其进行配置. 支持的二进制文件类型如下(注意, 并不是所有的原生平台都支持所有的二进制类型):

工厂方法	二进制文件类型	支持的原生平台
executable	可执行程序	所有的原生平台
sharedLib	共享的原生库文件	所有的原生平台, wasm32 除外
staticLib	静态的原生库文件	所有的原生平台, wasm32 除外
framework	Objective-C 框架	只支持 macOS 和 iOS

每个工厂方法都存在几个版本. 这里我们以 `executable` 为例进行说明, 对于其他工厂方法也存在完全相同的其他版本.ds.

最简单的版本不需要任何额外参数, 并对每一个编译类型创建一个二进制文件. 现在我们有 2 个编译类型: `DEBUG` (产生一个未经优化的, 带调试信息的二进制文件), 和 `RELEASE` (产生优化过的, 无调试信息的二进制文件). 因此, 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.
            }
        }
    }
}
```

上例中的 `executable` 方法接受一个 Lambda 表达式参数, 这个 Lambda 表达式会对每个二进制文件执行, 因此开发者可以在这里对二进制文件进行配置(参见 [配置二进制文件](#)). 注意, 如果不需要额外的配置信息, 那么这个 Lambda 表达式可以省略:

```
binaries {
    executable()
}
```

还可以指定使用哪些编译类型来创建二进制文件, 以及忽略哪些编译类型. 下面的示例只创建 `debug` 版的二进制文件.

```
binaries {
    executable([DEBUG]) {
        // 这里指定二进制文件的配置信息.
    }
}
```

```
binaries {
    executable(listOf(DEBUG)) {
        // 这里指定二进制文件的配置信息.
    }
}
```

工厂方法的最后一个版本, 可以定制二进制文件的名称.

```
binaries {
    executable('foo', [DEBUG]) {
        // 这里指定二进制文件的配置信息.
    }

    // 可以省略编译类型 (这时会使用所有的编译类型).
    executable('bar') {
        // 这里指定二进制文件的配置信息.
    }
}
```

```

binaries {
    executable("foo", listOf(DEBUG)) {
        // 这里指定二进制文件的配置信息.
    }

    // 可以省略编译类型 (这时会使用所有的编译类型).
    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允许我们对被创建的二进制文件指定一个名称前缀, 在编译脚本中可以使用这个名称来访问二进制文件 (参见[“访问二进制文件”](#) 小节). 而且这个前缀还被用于二进制文件的默认文件名. 比如, 在 Windows 平台, 上例的会输出两个文件 `foo.exe` 和 `bar.exe`.

使用二进制文件声明 API (自 1.3 版开始引入)

除二进制文件配置的 DSL 之外, 从 1.3 版开始, 还可以使用二进制文件声明 API. 通过这些 API, 你可以对编译任务指定一个或多个 `outputKinds`. 可用的输出类型包括:

- `executable`, 用于输出可执行程序;
- `dynamic`, 用于输出动态链接库;
- `static`, 用于输出静态库;
- `framework`, 用于输出 Objective-C 框架 (只对 macOS 和 iOS 编译目标支持这种输出)

输出类型的添加方法如下:

```

kotlin {
    linuxX64 { // 这里请修改为你的编译目标
        compilations.main.outputKinds("executable") // 也可以使用 "static", "dynamic" 或 "framework".
    }
}

```

```

kotlin {
    linuxX64 { // 这里请修改为你的编译目标
        compilations["main"].outputKinds("executable") // 也可以使用 "static", "dynamic" 或 "framework".
    }
}

```

这段代码会使用对应的类型创建二进制文件, 并使用编译任务的名称作为二进制名称前缀, 并将二进制文件添加到 `binaries` 列表内. 但请注意, 这样的二进制文件只有在项目执行之后才会被创建, 因此只有在 `afterEvaluate` 代码段内它们才可用.

尽管支持这种方法在 1.3.20 版中是支持的, 但未来它会被废弃. 因此建议开发者改为使用二进制文件 DSL.

## 访问二进制文件

二进制文件配置的 DSL 不仅可以用来创建二进制文件, 还可以访问已创建的二进制文件, 对其进行配置, 或者获取其属性(比如, 输出文件的路径). `binaries` 集合实现了 [DomainObjectSet](#) 接口, 提供了 `all`, `matching` 之类的方法, 可以用来对一组元素进行配置.

此外还可以从这个集合中获取特定的元素. 有两种方法. 第一种方法是, 使用二进制文件的唯一名称. 这个名称由名称前缀(如果有指定的话), 编译类型, 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`.

注意: 静态库和共享库分别带有 `static` 和 `shared` 后缀, 比如, `fooDebugStatic` 或 `barReleaseShared`

通过这个名称, 我们可以访问二进制文件:

```
// 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")
```

```
// 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")
```

第二种方法是使用有类型的 get 方法. 通过这些 get 方法, 我们可以使用名称前缀和编译类型访问某个特定类型的二进制文件.

```
// 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, 使用空字符串.
```

```
// 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.
```

```
// 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, 使用空字符串.
```

```
// 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.
```

## 配置二进制文件

二进制文件有一组属性, 可以对其进行配置. 目前支持的属性如下:

- **编译任务.** 每个二进制文件都是基于同一个编译目标中的一组编译任务构建的. 默认会使用 `main` 编译任务, 但你也可以使用其他编译任务.
- **链接参数.** 二进制文件编译时 传递给操作系统链接程序(system linker)的命令行参数. 你可以使用这个设置, 来将你的二进制文件链接到某些原生库.
- **输出文件名.** 默认情况下, 编译输出文件名由二进制文件的名称前缀决定, 如果没有指定名称前缀, 则由项目名称决定. 但也可以使用 `baseName` 属性来独立配置输出文件名. 注意, 最终输出的文件名会在 `baseName` 之上添加前缀和后缀, 前缀和后缀的具体内容与操作系统相关. 比如, 对于 Linux 共享库, `baseName` 设置为 `foo` 时, 最终输出文件名会是 `libfoo.so`.
- **入口点(Entry Point)** (只对可执行二进制文件有效). 默认情况下, Kotlin/Native 可执行程序入口点是位于最顶层包中的 `main` 函数. 使用这个设置, 我们可以改变这个默认设置, 使用一个自定义的函数作为可执行程序的入口点. 比如, 可以用来将 `main` 函数从最顶层包移动到其他包.

— 访问输出文件.

— 访问链接任务.

— 访问运行任务 (只对可执行二进制文件有效). `kotlin-multiplatform` 会对当前编译主机(Windows, Linux 和 macOS)上所有的可执行二进制文件创建运行任务. 这些运行任务的名称由二进制文件名称决定, 比如, `runReleaseExecutable<target-name>` 或 `runFooDebugExecutable<target-name>`. 对于可执行的二进制文件, 可以使用 `runTask` 属性来访问它的运行任务.

下面的例子演示如何使用这些设定.

```
binaries {
    executable('test', [RELEASE]) {
        // 使用 test 编译任务创建一个二进制文件.
        compilation = compilations.test

        // 设置传递给链接程序的命令行选项.
        linkerOpts = ['-L/lib/search/path', '-L/another/search/path', '-lmylib']

        // 定义最终输出文件的 base name.
        baseName = 'foo'

        // 设置入口函数.
        entryPoint = 'org.example.main'

        // 访问最终输出文件.
        println("Executable path: ${outputFile.absolutePath}")

        // 访问链接任务.
        linkTask.dependsOn(additionalPreprocessingTask)

        // 访问运行任务.
        // 注意, 在不支持这些二进制文件运行的编译平台上, runTask 会为 null.
        runTask?.dependsOn(prepareForRun)
    }
}

// 注意, 默认创建的测试任务同时也是一个运行任务.
// 因此, 你可以使用同样的属性访问它.
def testTask = binaries.getExecutable("test", DEBUG).runTask
task processTests {
    dependsOn(testTask)
}
```

```

binaries {
    executable("test", listOf(RELEASE)) {
        // 使用 test 编译任务创建一个二进制文件.
        compilation = compilations["test"]

        // 设置传递给链接程序的命令行选项.
        linkerOpts = mutableListOf("-L/lib/search/path", "-L/another/search/path", "-lmylib")

        // 定义最终输出文件的 base name.
        baseName = "foo"

        // 设置入口函数.
        entryPoint = "org.example.main"

        // 访问最终输出文件.
        println("Executable path: ${outputFile.absolutePath}")

        // 访问链接任务.
        linkTask.dependsOn(additionalPreprocessingTask)

        // 访问运行任务.
        // 注意, 在不支持这些二进制文件运行的编译平台上, runTask 会为 null.
        runTask?.dependsOn(prepareForRun)
    }
}

// 注意, 默认创建的测试任务同时也是一个运行任务.
// 因此, 你可以使用同样的属性访问它.
val testTask = binaries.getExecutable("test", DEBUG).runTask
val processTests by tasks.creating
processTests.dependsOn(testTask)

```

## 导出 Framework 中的依赖项

编译 Objective-C 框架时, 经常会出现一种需要, 不仅要打包当前项目的类文件, 同时还要打包当前项目依赖的其他类. 二进制文件配置的 DSL 提供了 `export` 方法, 我们可以用来指定需要导出哪些依赖项到我们的框架中. 注意, 只能导出对应的源代码集的 API 依赖项.

```

kotlin {
    sourceSets {
        macosMain.dependencies {
            // 这些依赖项会被导出到框架中.
            api project(':dependency')
            api 'org.example:exported-library:1.0'

            // 这个依赖项不会被导出到框架中.
            api 'org.example:not-exported-library:1.0'
        }
    }

    macosX64("macos").binaries {
        framework {
            export project(':dependency')
            export 'org.example:exported-library:1.0'
        }
    }
}

```

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // 这些依赖项会被导出到框架中.
            api(project(":dependency"))
            api("org.example:exported-library:1.0")

            // 这个依赖项不会被导出到框架中.
            api("org.example:not-exported-library:1.0")
        }
    }

    macosX64("macos").binaries {
        framework {
            export(project(":dependency"))
            export("org.example:exported-library:1.0")
        }
    }
}
```

如上面的例子所示, maven 依赖项也可以导出. 但由于 Gradle metadata 目前的限制, 这样的依赖项必须是一个平台相关的文件(比如, 应该是 `kotlinx-coroutines-core-native_debug_macos_x64`, 而不是 `kotlinx-coroutines-core-native`), 否则的话, 必须是传递性 (transitively) 导出(详情请参见下文).

默认情况下, 导出是非传递性的(non-transitively). 如果被导出的库 `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` 标记来修改.

```
binaries {
    framework {
        export project(':dependency')
        // 使用传递性导出.
        transitiveExport = true
    }
}
```

```
binaries {
    framework {
        export(project(":dependency"))
        // 使用传递性导出.
        transitiveExport = true
    }
}
```

## 对 CInterop 的支持

由于 Kotlin/Native 提供了 [与原生语言的交互能力](#), 因此编译脚本中也提供了一种 DSL 用来为编译任务配置这个功能.

编译任务可以与几种不同的原生语言交互. 在编译任务的 `cinterop` 代码段中可以配置与每种语言的交互能力:

```

kotlin {
    linuxX64 { // 这里请修改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")
                }
            }
        }
    }
}

```



```

kotlin {
    linuxX64 { // 这里请修改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 { /* ... */ }
        }
    }
}

```

对于使用到原生库的二进制输出文件, 经常会需要指定编译目标特定的链接参数. 可以使用二进制文件的 `linkerOpts` 属性来实现. 详情请参见 [配置二进制文件](#) 小节.

# 其他

## 解构声明(Destructuring Declaration)

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. 上例中我们声明了两个变量: `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, 等等.). 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.

注意, `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: 当我说:

```
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

举例来说, 假如我们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*, 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.

注意: 我们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, 我们应该:

- 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 (从 Kotlin 1.1 开始支持)

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 (从 Kotlin 1.1 开始支持)

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!" }
```

## 集合(Collection): List, Set, Map

与很多其他语言不同, Kotlin 明确地区分可变的和不可变的集合(list, set, map, 等等). 明确地控制集合什么时候可变什么时候不可变, 对于消除 bug 是很有帮助的, 也有助于设计出良好的 API.

理解可变集合的只读 *视图(view)* 与一个不可变的集合之间的差别, 是很重要的. 这两者都可以很容易地创建, 但类型系统无法区分这二者的差别, 因此记住哪个集合可变哪个不可变, 这是你自己的责任.

Kotlin 的 `List<out T>` 类型是一个只读的接口, 它提供的操作包括 `size`, `get` 等等. 与 Java 一样, 这个接口继承自 `Collection<T>`, `Collection<T>` 又继承自 `Iterable<T>`. 能够修改 List 内容的方法是由 `MutableList<T>` 接口添加的. 对于 `Set<out T>/MutableSet<T>` 以及 `Map<K, out V>/MutableMap<K, V>` 也是同样的模式.

通过下面的例子, 我们可以看看 list 和 set 类型的基本用法:

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)    // 打印结果为: "[1, 2, 3]"
numbers.add(4)
println(readOnlyView) // 打印结果为: "[1, 2, 3, 4]"
readOnlyView.clear() // -> 无法编译

val strings = hashSetOf("a", "b", "c", "c")
assert(strings.size == 3)
```

Kotlin 没有专门的语法用来创建 list 和 set. 你可以使用标准库中的方法, 比如 `listOf()`, `mutableListOf()`, `setOf()`, `mutableSetOf()`. 在并不极端关注性能的情况下, 创建 map 可以使用一个简单的 [惯用法](#): `mapOf(a to b, c to d)`.

注意, `readOnlyView` 变量指向的其实是同一个 list 实例, 因此它的内容会随着后端 list 一同变化. 如果指向 list 的只有唯一一个引用, 而且这个引用是只读的, 那么我们可以这个集合完全是不可变的. 创建一个这样的集合的简单办法如下:

```
val items = listOf(1, 2, 3)
```

目前, `listOf` 方法是使用 array list 实现的, 但在将来, 这个方法会返回一个内存效率更高的, 完全不可变的集合类型, 以便尽量利用集合内容不可变这个前提.

注意, 只读集合类型是 [协变的\(covariant\)](#). 也就是说, 假设 `Rectangle` 继承自 `Shape`, 你可以将一个 `List<Rectangle>` 类型的值赋给一个 `List<Shape>` 类型变量(集合类型的继承关系与元素类型的继承关系相同). 但这对于可变的集合类型是不允许的, 因为可能导致运行时错误: 你可能会将一个 `Circle` 类型的实例添加到一个 `List<Shape>` 内, 于是导致在你的程序的其他地方出现一个 `List<Rectangle>`, 其中包含一个 `Circle` 类型的实例.

有时候, 你希望向调用者返回集合在某个时刻的一个快照, 而且这个快照保证不会变化:

```
class Controller {
    private val _items = mutableListOf<String>()
    val items: List<String> get() = _items.toList()
}
```

`toList` 扩展方法只是单纯地复制 list 内的元素, 因此, 返回的 list 内容可以确保不会变化.

list 和 set 还有一些有用的扩展方法, 值得我们熟悉一下:

```

val items = listOf(1, 2, 3, 4)
items.first() == 1
items.last() == 4
items.filter { it % 2 == 0 } // 返回值为: [2, 4]

val rwList = mutableListof(1, 2, 3)
rwList.requireNotNulls() // 返回值为: [1, 2, 3]
if (rwList.none { it > 6 }) println("No items above 6") // 打印结果为: "No items above 6"
val item = rwList.firstOrNull()

```

… 此外还有你所期望的各种工具方法, 比如 sort, zip, fold, reduce 等等.

需要注意的是, 针对只读集合进行某种操作并返回一个可修改的集合 (比如 `+`, `filter`, `drop`, 等等操作.), 这时结果集合的创建过程本身不是一个原子操作(atomically), 因此, 在另一个线程中, 如果不经适当的同步控制, 直接访问这个结果集合是不安全的.

Map 也遵循相同的模式. 可以很容易地创建和访问, 比如:

```

val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
println(readWriteMap["foo"]) // 打印结果为: "1"
val snapshot: Map<String, Int> = HashMap(readWriteMap)

```

## 值范围(Range)

值范围表达式使用 `rangeTo` 函数来构造, 这个函数的操作符形式是 `..`, 另外还有两个相关操作符 `in` 和 `lin`. 任何可比较大小的数据类型 (comparable type) 都可以定义值范围, 但对于整数性的基本类型, 值范围的实现进行了特殊的优化. 下面是使用值范围的一些示例:

```
if (i in 1..10) { // 等价于: 1 <= i && i <= 10
    println(i)
}
```

整数性的值范围( `IntRange`, `LongRange`, `CharRange` )还有一种额外的功能: 可以对这些值范围进行遍历. 编译器会负责将这些代码变换为 Java 中基于下标的 `for` 循环, 不会产生不必要的性能损耗:

```
fun main() {
    //sampleStart
    for (i in 1..4) print(i)
    //sampleEnd
}
```

如果你需要按反序遍历整数, 应该怎么办? 很简单. 你可以使用标准库中的 `downTo()` 函数:

```
fun main() {
    //sampleStart
    for (i in 4 downTo 1) print(i)
    //sampleEnd
}
```

可不可以使用 1 以外的任意步长来遍历整数? 没问题, `step()` 函数可以帮你实现:

```
fun main() {
    //sampleStart
    for (i in 1..4 step 2) print(i)

    for (i in 4 downTo 1 step 2) print(i)
    //sampleEnd
}
```

要创建一个不包含其末尾元素的值范围, 可以使用 `until` 函数:

```
fun main() {
    //sampleStart
    for (i in 1 until 10) {
        // i in [1, 10), 不包含 10
        println(i)
    }
    //sampleEnd
}
```

## 值范围的工作原理

值范围实现了标准库中的一个共通接口: `ClosedRange<T>`.

`ClosedRange<T>` 表示数学上的一个闭区间(closed interval), 由可比较大小的数据类型(comparable type)构成. 这个区间包括两个端点: `start` 和 `endInclusive`, 这两个端点的值都包含在值范围内. 主要的操作是 `contains`, 主要通过 `in/lin` 操作符的形式来调用.

整数性类型的数列( `IntProgression`, `LongProgression`, `CharProgression` )代表算术上的一个整数数列. 数列由 `first` 元素, `last` 元素, 以及一个非 0 的 `step` 来定义. 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`. 除非数列为空, 否则遍历数列时一定会到达 `last` 元素.

数列是 `Iterable<N>` 的子类型, 这里的 `N` 分别代表 `Int`, `Long` 和 `Char`, 因此数列可以用在 `for` 循环内, 还可以用于 `map` 函数, `filter` 函数, 等等. 在 `step` 为正数的 `Progression` 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) {  
    // ...  
}
```

对于整数性类型, `..` 操作符将会创建一个实现了 `ClosedRange<T>` 和 `*Progression` 接口的对象. 比如, `IntRange` 实现了 `ClosedRange<Int>`, 并继承 `IntProgression` 类, 因此 `IntProgression` 上定义的所有操作对于 `IntRange` 都有效. `downTo()` 和 `step()` 函数的结果永远是一个 `*Progression`.

要构造一个数列, 可以使用对应的类的同伴对象中定义的 `fromClosedRange` 函数:

```
IntProgression.fromClosedRange(start, end, step)
```

数列的 `last` 元素会自动计算, 对于 `step` 为正数的情况, 会求得一个不大于 `end` 的最大值, 对于 `step` 为负数的情况, 会求得一个不小于 `end` 的最小值, 并且使得 `(last - first) % step == 0`.

## 工具函数

### `rangeTo()`

整数性类型上定义的 `rangeTo()` 操作符只是简单地调用 `*Range` 类的构造器, 比如:

```
class Int {  
    //...  
    operator fun rangeTo(other: Long): LongRange = LongRange(this, other)  
    //...  
    operator fun rangeTo(other: Int): IntRange = IntRange(this, other)  
    //...  
}
```

浮点型数值(`Double`, `Float`) 的 `rangeTo` 操作符返回一个值范围, 这个值范围在将一个数值与自己的范围边界进行比较时, [遵照 IEEE-754 标准](#). 这个操作符返回的值范围不是一个数列, 不能用来遍历.

### `downTo()`

`downTo()` 扩展函数可用于一对整数类型值, 下面是两个例子:

```
fun Long.downTo(other: Int): LongProgression {  
    return LongProgression.fromClosedRange(this, other.toLong(), -1L)  
}  
  
fun Byte.downTo(other: Int): IntProgression {  
    return IntProgression.fromClosedRange(this.toInt(), other, -1)  
}
```

### `reversed()`

对每个 `*Progression` 类都定义了 `reversed()` 扩展函数, 所有这些函数都会返回相反的数列:

```
fun IntProgression.reversed(): IntProgression {  
    return IntProgression.fromClosedRange(last, first, -step)  
}
```

### `step()`

对每个 `*Progression` 类都定义了 `step()` 扩展函数, 所有这些函数都会返回使用新 `step` 值(由函数参数指定)的数列. 步长值参数要求永远是正数, 因此这个函数不会改变数列遍历的方向:

```
fun IntProgression.step(step: Int): IntProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return IntProgression.fromClosedRange(first, last, if (this.step > 0) step else -step)
}

fun CharProgression.step(step: Int): CharProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return CharProgression.fromClosedRange(first, last, if (this.step > 0) step else -step)
}
```

注意, 函数返回的数列的 `last` 值可能会与原始数列的 `last` 值不同, 这是为了保证 `(last - first) % step == 0` 原则. 下面是一个例子:

```
(1..12 step 2).last == 11 // 数列中的元素为 [1, 3, 5, 7, 9, 11]
(1..12 step 3).last == 10 // 数列中的元素为 [1, 4, 7, 10]
(1..12 step 4).last == 9  // 数列中的元素为 [1, 5, 9]
```



## 类型检查与类型转换: ‘is’ 与 ‘as’

### is 与 !is 操作符

我们可以使用 `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, 或者使用与它相反的 `!is` 操作符:

```
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

很多情况下, 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
```

在 `&&` 和 `||` 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
}
```

这种 [智能类型转换\(smart cast\)](#) 对于 [when 表达式](#) 和 [while 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, 假如编译器无法确保变量不会改变, 此时智能类型转换是无效的. 更具体地说, 必须满足以下条件时, 智能类型转换才有效:

- 局部的 `val` 变量 - 永远有效, 但 [局部委托属性](#) 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\(module\)](#) 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;

- 局部的 `var` 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, 我们称之为 *不安全的(unsafe)*. 在 Kotlin 中, 不安全的类型转换使用中缀操作符 `as` (参见 [操作符优先顺序](#)):

```
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\(nullable\)](#), 也就是说, 如果 `y` 为 `null`, 上例中的代码将抛出一个异常. 为了实现在 Java 相同的类型转换, 我们需要在类型转换操作符的右侧使用可为 `null` 的类型, 比如:

```
val x: String? = y as String?
```

## “安全的” (nullable) 类型转换操作

为了避免抛出异常, 你可以使用 *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.

## 类型擦除(Type erasure) 与泛型类型检查

涉及 [泛型](#) 的情况下, Kotlin 会在编译期间确保代码的类型安全性, 而在运行期间, 泛型类型的实例中并不保存它们的实际类型参数的信息. 比如, `List<Foo>` 的类型信息会被擦除, 变成 `List<*>`. 通常情况下, 在运行期间没有办法检查一个实例是不是属于带有某个类型参数的泛型类型.

因此, 由于类型擦除导致的, 在运行期间无法正确执行的 `is` 检查, 编译器会禁止使用, 比如 `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: List<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, 省略类型参数: `list as ArrayList`.

使用 [实体化的类型参数\(Reified type parameter\)](#) 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 `arg is T` 检查, 但是如果 `arg` 本身是一个泛型类型的实例, 它自己的类型参数仍然会被擦除. 示例:

```
//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>>() // 此处破坏了类型安全型!
//sampleEnd

fun main() {
    println("stringToSomething = " + stringToSomething)
    println("stringToInt = " + stringToInt)
    println("stringToList = " + stringToList)
    println("stringToStringList = " + stringToStringList)
}
```

## 未检查的类型转换

如上文所述, 类型擦除使得在运行期间无法检查一个泛型类型的实例的实际类型参数, 而且, 代码中的泛型类型相互之间的关系可能会不够紧密, 使得编译无法确保类型安全性。

即便如此, 有时我们还是可能通过更高级别的程序逻辑来暗示类型安全性。比如:

```
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")

// Warning: Unchecked cast: `Map<String, *>` to `Map<String, Int>`
val intsDictionary: Map<String, Int> = readDictionary(intsFile) as Map<String, Int>
```

编译器会对代码最后一行中的类型转换提示一个警告。这个类型转换在运行期无法完整地检查, 也不能保证 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, 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\(eneric variance\)](#) 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\(Reified type parameter\)](#) 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](#), 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](#) (`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。

## this 表达式

为了表示当前函数的 *接收者(receiver)*, 我们使用 `this` 表达式:

- 在 [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](#) 中, 或 [带接收者的函数数字面值\(function literal\)](#) 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](#), 或 [扩展函数](#), 或有标签的 [带接受者的函数数字面值](#))内的 `this`, 我们使用 `this@label`, 其中的 `@label` 是一个 [标签](#), 代表我们想要访问的 `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
      }

      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
      }
    }
  }
}
```

## 相等判断

在 Kotlin 中存在两种相等判断:

- 结构相等 ( `equals()` 判断);
- 引用相等 (两个引用指向同一个对象).

### 结构相等

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)
```

也就是说, 如果 `a` 不为 `null`, 将会调用 `equals(Any?)` 函数, 否则(也就是如果 `a` 为 `null`) 将会检查 `b` 是否指向 `null`.

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`.

如果需要实现自定义的相等判断, 请覆盖 `equals(other: Any?): Boolean` 函数. 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比较操作符的结果.

### 浮点数值的相等比较

如果相等比较的对象值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.

否则, 将会使用结构相等判定, 这种判定不遵循 IEEE 754 浮点数运算标准, 因此 `NaN` 不等于它自己, 而且 `-0.0` 不等于 `0.0`.

详情请参见: [浮点值的比较](#).

### 引用相等

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`. 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.

## 操作符重载(Operator overloading)

Kotlin 允许我们对数据类型的一组预定义的操作符提供实现函数。这些操作符的表达符号是固定的(比如 `+` 或 `*`), [优先顺序](#)也是固定的。要实现这些操作符, 我们需要对相应的数据类型实现一个固定名称的 [成员函数](#) 或 [扩展函数](#), 这里所谓”相应的数据类型”, 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。用于实现操作符重载的函数应该使用 `operator` 修饰符进行标记。

下面我们介绍与各种操作符的重载相关的约定。

### 一元操作符

#### 一元前缀操作符

表达式	翻译为
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>

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`;

注意, 这些操作符, 以其其它所有操作符, 都对 [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
<code>a++</code>	<code>a.inc()</code> (参见下文)
<code>a--</code>	<code>a.dec()</code> (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 `a.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
<code>a + b</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b)</code> , <code>a.mod(b)</code> (已废弃)
<code>a..b</code>	<code>a.rangeTo(b)</code>

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.

注意, `rem` 操作符函数从 Kotlin 1.1 版开始支持. Kotlin 1.0 版使用的是 `mod` 操作符函数, 在 Kotlin 1.1 版中已经废弃了.

### 示例

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
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

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
<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, j]</code>	<code>a.get(i, j)</code>
<code>a[i_1, ..., i_n]</code>	<code>a.get(i_1, ..., i_n)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, j] = b</code>	<code>a.set(i, j, b)</code>
<code>a[i_1, ..., i_n] = b</code>	<code>a.set(i_1, ..., i_n, b)</code>

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
<code>a()</code>	<code>a.invoke()</code>
<code>a(i)</code>	<code>a.invoke(i)</code>
<code>a(i, j)</code>	<code>a.invoke(i, j)</code>
<code>a(i_1, ..., i_n)</code>	<code>a.invoke(i_1, ..., i_n)</code>

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
<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.remAssign(b), a.modAssign(b)</code> (已废弃)

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, 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).

注意: 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
<code>a == b</code>	<code>a?.equals(b) ?: (b === null)</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: (b === null))</code>

这些操作符是通过 `equals(other: Any?): Boolean` 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.

注意: `===` 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
<code>a &gt; b</code>	<code>a.compareTo(b) &gt; 0</code>
<code>a &lt; b</code>	<code>a.compareTo(b) &lt; 0</code>
<code>a &gt;= b</code>	<code>a.compareTo(b) &gt;= 0</code>
<code>a &lt;= b</code>	<code>a.compareTo(b) &lt;= 0</code>



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](#).

### 对命名函数的中缀式调用

使用 [中缀式函数调用](#), 我们可以模拟自定义的中缀操作符.

## Null 值安全性

### 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代码中 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](#)。

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。

Kotlin 的类型系统致力于从我们的代码中消除 `NullPointerException`。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;
- 初始化过程中存在某些数据不一致, 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](#), 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](#)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不正确, 比如, 一段 Java 代码可能向一个 Kotlin `MutableList<String>` 中添加一个 `null` 值, 也就是说, 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"  
    a = null // 编译错误  
    //sampleEnd  
}
```

要允许 null 值, 我们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"  
    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
```

但我们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 `null` 和非 `null` 的两种情况分别处理:

```
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` 就有可能在检查之后被修改为 `null` 值.

## 安全调用

第二个选择方案是使用安全调用操作符, 写作 `?.`:

```
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, 假如雇员 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, 那么, 为了取得 Bob 所属部门的主管的名字, (如果存在的话), 我们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](#):

```
fun main() {
    //sampleStart
    val listWithNulls: List<String?> = listOf("Kotlin", null)
    for (item in listWithNulls) {
        item?.let { println(it) } // 打印 A, 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()
```

## Elvis 操作符

假设我们有一个可为 null 的引用 `r`, 我们可以用说, “如果 `r` 不为 null, 那么就使用它, 否则, 就使用某个非 null 的值 `x`”:

```
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, 还可以使用 Elvis 操作符来表达, 写作 `?:`:

```
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.

注意, 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. 这种用法可以带来很大的方便, 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. 我们可以写 `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无声无息地出现.

## 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()
```

## 异常(Exception)

### 异常类

Kotlin 中所有的异常类都是 `Throwable` 的后代类。每个异常都带有一个错误消息, 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` 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 { parseInt(input) } 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;
```

这个方法签名代表什么意思? 它说, 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? 因为这个对象有可能会执行 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, 参见 [Effective Java, 第 3 版](#), 第 77 条: *不要忽略异常*。

Bruce Eckel 在 [Java 需要受控异常吗?](#) 一文中说:

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。

另外还有其他一些这类讨论文章:

- [Java 的受控异常是一个错误](#) (Rod Waldhoff)
- [受控异常带来的问题](#) (Anders Hejlsberg)(译注, 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)

## 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` 表达式的类型是一个特殊的类型 `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
```

另一种用到这个类型的情况是类型推断. 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](#) 中关于异常的小节.

## 注解(Annotation)

### 注解的声明

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` 指定这个注解可被用于哪些元素(类, 函数, 属性, 表达式, 等等.);
- `@Retention` 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` 允许在单个元素上多次使用同一个注解;
- `@MustBeDocumented` 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.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](#)). Kotlin 编译器会将它自动转换为 Java 类, 因此 Java 代码可以正常地访问到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
```

## 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](#) 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)

当你为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, 那么可以在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(从 Kotlin 1.2 开始支持), 或使用 `arrayOf` 函数:

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```
// Kotlin 1.2+ 版本:
@AnnWithArrayMethod(names = ["abc", "foo", "bar"])
class C
```

```
// Kotlin 旧版本:
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar"))
class D
```

### 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
}
```

## 反射

反射是语言与库中的一组功能, 可以在运行时刻获取程序本身的信息. Kotlin 将函数和属性当作语言中的一等公民(first-class citizen), 而且, 通过反射获取它们的信息(也就是说, 在运行时刻得到一个函数或属性的名称和数据类型) 可以通过简单的函数式, 或交互式的编程方式实现.

⚠ 在 Java 平台上, 使用反射功能所需要的运行时组件是作为一个单独的 JAR 文件发布的(kotlin-reflect.jar). 这是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. 如果你需要使用反射, 请注意将这个 .jar 文件添加到你的项目的 classpath 中.

### 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:

```
val c = MyClass::class
```

类引用是一个 [KClass](#) 类型的值.

注意, Kotlin 的类引用不是一个 Java 的类引用. 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 (从 Kotlin 1.1 开始支持)

`::class` 语法同样可以用于取得某个对象实例的类的引用:

```
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`, 但你将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, 除了用来反射程序结构之外, 还可以被调用, 或用作 [函数类型](#) 的实例.

所有可调用的引用的共同的超类是 [KCallable<out R>](#), 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)

假设我们有一个有名称的函数, 声明如下:

```
fun isOdd(x: Int) = x % 2 != 0
```

我们可以很容易地直接调用它(`isOdd(5)`), 但我们也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, 我们使用 `::` 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>](#) 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
```

如果我们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, 比如, `String::toCharArray` .

注意, 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(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(*))` . 现在, 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 `KProperty<Int>`, 通过它 `get()` 方法可以得到属性值, 通过它的 `name` 属性可以得到属性名称. 详情请参见 [KProperty 类的 API 文档](#).

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 `KMutableProperty<Int>`, 它有一个 `set()` 方法:

```
var y = 1

fun main() {
    ::y.set(2)
    println(y)
}
```

属性引用可以用在所有使用单个参数函数的地方:

```
fun main() {
    //sampleStart
    val strs = listOf("a", "bc", "def")
    println(strs.map(String::length))
    //sampleEnd
}
```

要访问类的成员属性, 我们需要使用限定符:

```
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. 构造器引用可以用于使用函数类型对象的地方, 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, 我们可以很简单地调用上面的函数:

```
function(::Foo)
```

指向构造器的引用的类型是 [KFunction<out R>](#) 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 (从 Kotlin 1.1 开始支持)

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
}
```

我们将 `matches` 方法保存在一个指向它的引用变量内, 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表达式的地方:

```
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
}
```

从 Kotlin 1.2 开始, 不再需要明确地指定 `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 \(inner class\)](#) 的构造器引用:

```
class Outer {
    inner class Inner
}

val o = Outer()
val boundInnerCtor = o::Inner
```

## 作用域函数(Scope Function)

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](#), 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`, `run`, `with`, `apply`, 以及 `also`.

基本上, 这些函数所做的事情都是一样的: 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.

下面是作用域函数的典型使用场景:

```
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.

```
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.

由于所有的作用域函数都很类似, 因此选择一个适合你需求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. 下面我们将会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.

### 作用域函数之间的区别

由于所有的作用域函数都很类似, 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:

- 访问上下文对象的方式
- 返回值.

**访问上下文对象: 使用 `this` 或使用 `it`**



在作用域函数的 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](#)( `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情况下的优点和缺点, 并提供一些使用建议.

```
fun main() {
    val str = "Hello"
    // 使用 this
    str.run {
        println("The receiver string length: $length")
        //println("The receiver string length: ${this.length}") // 这种写法的功能与上面一样
    }

    // 使用 it
    str.let {
        println("The receiver string's length is ${it.length}")
    }
}
```

#### 使用 this

`run`, `with`, 和 `apply` 函数将上下文函数作为 Lambda 表达式的接受者 - 通过 `this` 关键字来访问. 因此, 在这些函数的 Lambda 表达式内, 可以向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, 或者 adam.age = 20
        city = "London"
    }
    //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. 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了. 因此, 把上下文对象作为 `it` 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()
    //sampleEnd
}
```

另外, 如果把上下文对象作为参数传递, 你还可以在作用域内为它指定一个自定义的名称.

```
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()
    //sampleEnd
}
```

## 返回值

各种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.

这两种方式, 允许你根据你的代码下面需要做什么, 来选择适当的作用域函数.

## 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, 下面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, 很多情况下各个函数是可以互换的, 因此这里的示例只演示常见的使用风格.

### 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 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, 这段代码可以改写为:

```

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 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 ?. , 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
}

```

let 函数的另一个使用场景是, 在一个比较小的作用域内引入局部变量, 以便提高代码的可读性. 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 '$firstItem'")
        if (firstItem.length >= 5) firstItem else "!" + firstItem + "!"
    }.toUpperCase()
    println("First item after modifications: '$modifiedFirstItem'")
//sampleEnd
}

```

## with 函数

这是一个非扩展函数: 上下文对象 作为参数传递, 但在 Lambda 表达式内部, 它是一个接受者 ( `this` ). 返回值 是 Lambda 表达式的结果值.

我们推荐使用 `with` 函数, 用来在上下文对象上调用函数, 而不返回 Lambda 表达式结果值. 在源代码中, `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
}

```

`with` 函数的另一种使用场景是, 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 的功能与 with 一样, 但调用它的方式与 let 一样 - 作为上下文对象的扩展函数来调用.

如果你的 Lambda 表达式既包含对象的初始化处理, 也包含结果值的计算处理, 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)
}
```

除了对接受者对象调用 run 函数之外, 也可以把它作为非扩展函数来使用. 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.

```
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("+1234 -FFFF not-a-number")) {
        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.

如果代码段没有返回值, 并且主要操作接受者对象的成员, 那么适合使用 apply 函数. apply 函数的常见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"
    }
    //sampleEnd
}
```

由于返回值是接受者, 因此你可以很容易地将 `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` 函数适合于执行一些将上下文对象作为参数的操作. 可以使用 `also` 函数来执行一些不改变上下文对象的额外操作, 比如输出日志, 打印调试信息. 通常, 你可以从链式调用中删除 `also` 的调用, 而不会改变程序的逻辑。

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
}
```

## 选择作用域函数

为了帮助你选择适合需要的作用域函数, 我们整理了这些函数之间关键区分的比较表格。

函数	上下文对象的引用方式	返回值	是否扩展函数
<code>let</code>	<code>it</code>	Lambda 表达式的结果值	是
<code>run</code>	<code>this</code>	Lambda 表达式的结果值	是
<code>run</code>	-	Lambda 表达式的结果值	不是: 不使用上下文对象来调用
<code>with</code>	<code>this</code>	Lambda 表达式的结果值	不是: 上下文对象作为参数传递.
<code>apply</code>	<code>this</code>	上下文对象本身	是
<code>also</code>	<code>it</code>	上下文对象本身	是

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`

不同的函数的使用场景是有重叠的, 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代码变得更简洁, 但也要注意不要过度使用: 可能会降低你的代码的可读性, 造成错误. 不要在作用域函数内部再嵌套作用域函数, 对作用域函数的链式调用要特别小心: 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.

## takeIf 函数和 takeUnless 函数

除作用域函数外, 标准库还提供了 `takeIf` 函数和 `takeUnless` 函数. 这些函数允许你在链式调用中加入对象的状态检查.

如果对一个对象调用 `takeIf` 函数, 并给定一个检查条件, 这个函数会在对象满足检查条件时返回这个对象, 否则返回 `null`. 因此, `takeIf` 函数可以作为单个对象的过滤函数. 类似的, `takeUnless` 函数会在对象不满足检查条件时返回这个对象, 满足条件时返回 `null`. 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() }?.toUpperCase()
    //val caps = str.takeIf { it.isNotEmpty() }.toUpperCase() // 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. 一个很好的例子就是, 将这些函数与 `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
}
```

如果没有这些标准库函数的帮助, 上面的代码会变成这样:



```
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  
}
```

## 类型安全的构建器(Type-Safe Builder)

通过将恰当命名的函数用做构建器, 结合 [带接受者的函数数字面值](#), 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](#) 或 XML;
- 以程序方式构建 UI 组件布局: [Anko](#)
- 为 Web 服务器配置路由: [Ktor](#).

### 类型安全的构建器的示例

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 = "http://kotlinlang.org") {+"Kotlin"}

            // 混合内容
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "http://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)。

### 工作原理

我们来看看 Kotlin 中类型安全的构建器的实现机制。首先, 我们要对我们想要构建的东西定义一组模型, 在这个示例中, 我们需要对 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, 也就是说, 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, 回忆一下为什么我们可以写这样的代码:

```
html {
    // ...
}
```

html 实际上是一个函数调用, 它接受一个 [Lambda 表达式](#) 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, 我们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, 我们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, 省略之后我们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 (在我们的示例中, 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, 因此我们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, 这两个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)
```

现在我们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, 我们如何在标签内部添加文本. 在上面的示例程序中, 我们写了这样的代码:

```
html {
    head {
        title {"XML encoding with Kotlin"}
    }
    // ...
}
```

我们所作的, 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 (从 Kotlin 1.1 开始支持)

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, 我们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, Kotlin 1.1 中引入了一种特殊机制, 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, 我们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 构建器我们可以定义一个注解 `@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) { ... }
```

我们不必对 `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](#) 和 [带接受者的 Lambda 表达式](#).

注意, `@DslMarker` 注解要在 Kotlin 1.1 之后的版本才可用.

```
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  
}
```

## 实验性 API 标记(Experimental API Marker)

⚠️ 用于标记和使用实验性 API 的注解(@Experimental 和 @UseExperimental) 在 Kotlin 1.3 中是 [实验性功能](#). 详情请参见 [下文](#).

Kotlin 标准库为开发者提供了一种机制, 用来创建和使用 [实验性 API](#). 这种机制允许库的开发者告知使用者, 库 API 中的某些部分, 比如某些类或者某些函数, 目前还未稳定, 将来可能发生变更. 这样的变更可能会要求库的使用者重新编写并重新编译他们的代码. 为了防止潜在的兼容性问题, 编译器对使用实验性 API 的代码会提示警告, 并会要求开发者明确同意使用实验性 API.

### 使用实验性 API

如果库中的一个类或一个函数被作者标记为实验性 API, 那么在你的代码中使用它会导致编译警告甚至编译错误, 除非你明确同意接受使用这个实验性 API. 有几种方法来同意接受使用实验性 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.

### 传递式使用(Propagating Use)

如果你在代码中使用了实验性 API, 而你的代码本身又打算给第三方使用(是一个库), 那么你也可以将你的 API 标记为实验性 API. 为了做到这一点, 需要把你的函数体中使用到的 API 的 [实验性 API 标记注解](#) 添加到你的函数的声明部分. 这样你就可以使用这个标记标注过的 API 元素了.

```
// 库代码
@Experimental
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime // 实验性 API 标注

@ExperimentalDateTime
class DateProvider // 实验性类
```

```
// 库的使用者代码
fun getYear(): Int {
    val dateProvider: DateProvider // 错误: DateProvider 是实验性 API
    // ...
}

@ExperimentalDateTime
fun getDate(): Date {
    val dateProvider: DateProvider // OK: 这个函数已被标注为实验性 API
    // ...
}

fun displayDate() {
    println(getDate()) // 错误: getDate() 是实验性 API, 需要明确表示同意使用实验性 API
}
```

在上面的示例中我们可以看到, 被注解的函数变得像是 @ExperimentalDateTime 实验性 API 的一部分. 因此, 实验性 API 的状态传递到了使用实验性 API 的代码中; 这段代码又会要求使用它的代码必须明确同意使用实验性 API. 如果要使用多个实验性 API, 请在你的函数声明中分别添加它们的注解.

### 非传递式使用(Non-propagating Use)

在并不对外提供 API 的模块内部, 比如应用程序模块, 你可以使用实验性 API, 而不必将这种状态传递到你的代码中. 这种情况下, 请使用 [@UseExperimental\(Marker::class\)](#) 注解来标注你的代码, 并在这个注解中指明实验性 API 的标记注解:



```
// 库代码
@Experimental
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime // 实验性 API 标注

@ExperimentalDateTime
class DateProvider // 实验性类
```

```
// 库的使用者代码
@UseExperimental(ExperimentalDateTime::class)
fun getDate(): Date { // 使用 DateProvider; 但不对外导出实验性 API 状态
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    println(getDate()) // OK: getDate() 不是实验性 API
}
```

当使用者调用 `getDate()` 函数时, 他们不会被警告说这个函数内部使用了实验性 API.

如果想要一个源代码文件的所有类和所有函数内使用某个实验性 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注解 `@file:UseExperimental`.

```
// 库的使用者代码
@file:UseExperimental(ExperimentalDateTime::class)
```

### 模块范围内使用(Module-wide Use)

如果你不希望在你的代码中每次使用实验性 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实验性 API. 模块范围内使用实验性 API, 本身也可以是传递式的, 或者非传递式的:

- 如果希望使用实验性 API, 但不对外传递, 可以使用参数 `-Xuse-experimental` 来编译模块, 并指定你所使用的实验性 API 的标注的完全限定名称: `-Xuse-experimental=org.mylibrary.ExperimentalMarker`. 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 `@UseExperimental(ExperimentalMarker::class)` 注解.
- 如果希望使用实验性 API, 并且让你的整个模块都成为实验性的状态, 可以使用参数 `-Xexperimental=org.mylibrary.ExperimentalMarker` 来编译模块. 这种情况下, 模块内的 每一个声明 都会变成实验性 API. 使用这个模块会也要求使用者明确同意使用实验性 API.

如果使用 Gradle 来编译模块, 你可以这样添加编译参数:

```
compileKotlin {
    kotlinOptions {
        freeCompilerArgs += "-Xuse-experimental=org.mylibrary.ExperimentalMarker"
    }
}
```

```
tasks.withType<KotlinCompile>().all {
    kotlinOptions.freeCompilerArgs += "-Xuse-experimental=org.mylibrary.ExperimentalMarker"
}
```

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>
      <executions>...</executions>
      <configuration>
        <args>
          <arg>-Xuse-experimental=org.mylibrary.ExperimentalMarker</arg>
        </args>
      </configuration>
    </plugin>
  </plugins>
</build>

```

如果希望在模块级别上使用多个实验性 API, 请对你的模块中使用到的每一个实验性 API 的标注, 逐个使用上述编译参数。

## 标记实验性 API

### 创建标记用的注解

如果你希望将你的模块 API 声明为实验性 API, 需要创建一个注解, 用来作为这个 API 的 *实验性 API* 标注. 这个注解类本身必须标注 [@Experimental](#) 注解:

```

@Experimental
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime

```

用作实验性 API 标注的注解类, 必须满足几个要求:

- [retention](#) 为 `BINARY`
- [targets](#) 不含 `EXPRESSION` 和 `FILE`
- 没有参数.

实验性 API 标注的注解类, 需要标注它的 [严重级别](#), 用来告知使用者, 严重级别包括以下两个值:

- `Experimental.Level.ERROR`. 对实验性 API 的明确同意是必须的. 否则, 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.
- `Experimental.Level.WARNING`. 对实验性 API 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 `@Experimental` 注解的 `level` 参数来设置你希望的严重级别.

```

@Experimental(level = Experimental.Level.WARNING)
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime

```

如果你需要对外公布多个实验性的功能, 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实验性功能时更加安全: 他们可以只使用他们需要的那部分功能. 同时也使你自己能够控制各个功能, 将它们分别升级到稳定状态.

## 标记实验性 API

想要将 API 标记为实验性 API, 请在它的声明部分添加你定义的那个实验性 API 标注注解:

```
@ExperimentalDateTime
class DateProvider

@ExperimentalDateTime
fun getTime(): Time {}
```

### 模块范围内的标记(Module-wide Marker)

如果你认为你的整个模块内的所有的 API 都是实验性 API, 那么可以对整个模块进行标注, 使用编译参数 `-Xexperimental`, 详情请参见 [模块范围内使用\(Module-wide Use\)](#).

### 实验性 API 升级为正式 API

一旦你的实验性 API 升级为正式 API, 可以按照它最终的状态发布了, 这时请从它的声明中删除实验性 API 标记注解, 以便允许使用者不受限制地使用它. 但是, 你还需要将实验性 API 标记注解本身继续留在模块内, 以免破坏使用者代码的兼容性. 为了让 API 的使用者相应地升级他们的模块(也就是从他们的代码中删除实验性 API 标记注解, 并重新编译代码), 请将注解标注为 `@Deprecated`, 并在这个注解的 `message` 参数中给出解释.

```
@Deprecated("This experimental API marker is not used anymore. Remove its usages from your code.")
@Experimental
annotation class ExperimentalDateTime
```

### 实验性 API 标注本身的实验性状态

本章中我们介绍了实验性 API 使用时的标注机制, 但在 Kotlin 1.3 中, 这种机制本身也是实验性的功能. 也就是说, 在未来的发布版中, 可能会发生变化, 并导致不兼容. 为了让 `@Experimental` 和 `UseExperimental` 注解的使用者意识到这两个注解的实验性状态, 对于使用这些注解的代码, 编译器会提示警告信息:

```
This class can only be used with the compiler argument '-Xuse-experimental=kotlin.Experimental'
```

To remove the warnings, add the compiler argument `-Xuse-experimental=kotlin.Experimental`.

# 核心库

标准库(Standard Library)

参见 [Kotlin 官方网站](#)

## 测试库(kotlin.test)

参见 [Kotlin 官方网站](#)

# 参考

## 关键字(Keyword)与操作符(Operator)

### 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](#)
  - 为 `import` 指定一个别名
- `as?` 用于 [安全的类型转换](#)
- `break` [结束一个循环](#)
- `class` 声明一个 [类](#)
- `continue` [跳转到最内层循环的下次执行](#)
- `do` 开始一个 [do/while 循环](#) (loop with postcondition)
- `else` 定义 [if 表达式](#) 的一个分支, 这个分支在条件为 `false` 时执行
- `false` 指定 [布尔类型](#) 的 ‘false’ 值
- `for` 开始一个 [for 循环](#)
- `fun` 声明一个 [函数](#)
- `if` 开始一个 [if 表达式](#)
- `in`
  - 指定 [for 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](#) 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](#) 的实体
  - 在 [when 表达式](#) 中做同样的判断
  - 将一个类型参数标记为 [反向类型变异](#)
- `!in`
  - 用作操作符, 判断一个值是否 [不属于 一个值范围](#), 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](#) 的实体
  - 在 [when 表达式](#) 中做同样的判断
- `interface` 声明一个 [接口](#)
- `is`
  - 判断 [一个值是不是某个类型](#)
  - 在 [when 表达式](#) 中做同样的判断
- `!is`
  - 判断 [一个值是否不是某个类型](#)
  - 在 [when 表达式](#) 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](#)
- `package` 指定 [当前源代码文件的包](#)
- `return` [从最内层的函数或匿名函数中返回](#)
- `super`
  - [引用一个方法或属性在超类中的实现](#)
  - [在次级构造器中调用超类构造器](#)
- `this`
  - 引用 [当前接受者](#)
  - [在次级构造器中调用同一个类的另一个构造器](#)
- `throw` [抛出一个异常](#)
- `true` 指定 [布尔类型](#) 的 ‘true’ 值
- `try` [开始一个异常处理代码段](#)
- `typealias` 声明一个 [类型别名](#)
- `val` 声明一个只读的 [属性](#), 或者一个只读的 [局部变量](#)
- `var` 声明一个可变的 [属性](#), 或者一个可变的 [局部变量](#)
- `when` 开始一个 [when 表达式](#) (执行其中一个分支)
- `while` 开始一个 [while 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](#)
  - [将一个属性的访问器函数实现委托给另一个对象](#)
- `catch` 开始一个 [处理特定的异常类型](#) 的代码段
- `constructor` 声明一个 [主构造器, 或次级构造器](#)
- `delegate` 用作一种 [注解的使用目标\(target\)](#)
- `dynamic` 引用一个 [动态类型](#) in Kotlin/JS code
- `field` 用作一种 [注解的使用目标\(target\)](#)
- `file` 用作一种 [注解的使用目标\(target\)](#)
- `finally` 开始一个 [try 代码段结束时始终会被执行](#) 的代码段
- `get`
  - 声明 [属性的取值方法](#)
  - 用作一种 [注解的使用目标\(target\)](#)
- `import` [从另一个包中将一个声明导入到当前源代码文件](#)
- `init` 开始一个 [初始化代码段](#)
- `param` 用作一种 [注解的使用目标\(target\)](#)
- `property` 用作一种 [注解的使用目标\(target\)](#)
- `receiver` 用作一种 [注解的使用目标\(target\)](#)
- `set`
  - 声明 [属性的设值方法](#)
  - 用作一种 [注解的使用目标\(target\)](#)
- `setparam` 用作一种 [注解的使用目标\(target\)](#)
- `where` 指定 [泛型类型参数的约束](#)

## 标识符关键字(Modifier Keyword)

以下符号在声明的标识符列表中用作关键字, 在其他场合可以用作标识符:

- `actual` 在 [跨平台项目](#) 中, 表示某个特定平台上的具体实现
- `abstract` 将一个类或一个成员标注为 [抽象元素](#)
- `annotation` 声明一个 [注解类](#)
- `companion` 声明一个 [同伴对象](#)
- `const` 将一个属性标注为 [编译时常数值](#)
- `crossinline` 禁止 [传递给内联函数的 lambda 表达式中的非局部的返回](#)
- `data` 指示编译器, [为类生成常用的成员函数](#)
- `enum` 声明一个 [枚举类](#)
- `expect` 标注一个 [与平台相关的声明](#), 在各个平台模块中, 需要存在对应的具体实现.
- `external` 标注一个声明不是由 Kotlin 语言实现的 (可以通过 [JNI](#) 实现, 或者用 [JavaScript](#) 实现)
- `final` 禁止 [覆盖成员](#)
- `infix` 允许使用 [中缀标记法](#) 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](#)
- `inner` 允许在 [嵌套内](#) 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](#)
- `lateinit` 允许 [在构造器之外初始化非 null 的属性](#)
- `noinline` 关闭 [对传递给内联函数的 lambda 表达式的内联](#)
- `open` 允许 [继承类, 或者覆盖成员](#)
- `operator` 将函数标记为 [操作符重载, 或实现一个规约](#)
- `out` 将类型参数标记为 [协变的](#)
- `override` 将成员标记为 [对超类成员的覆盖](#)
- `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](#)
- `protected` 将声明标记为 [只在当前类, 以及它的子类中可以访问](#)
- `public` 将声明标记为 [在任何位置都可以访问](#)
- `reified` 将内联函数的类型参数标记为 [在运行时刻可以访问](#)
- `sealed` 声明一个 [封闭类](#) (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](#) 中使用)
- `tailrec` 将一个函数标注为 [尾递归](#) (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](#)

## 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](#)
- `it` 在 lambda 表达式内部, 用来 [引用 lambda 表达式的隐含参数](#)

## 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](#)
- `=`
  - 赋值操作符



- 用来指定 [参数的默认值](#)
- `+=`, `-=`, `*=`, `/=`, `%=` - [计算并赋值](#)
- `++`, `--` - [递增与递减操作符](#)
- `&&`, `||`, `!` - ‘与’, ‘或’, ‘非’ 逻辑运算符 (用于位运算, 使用对应的 [中缀函数](#))
- `==`, `!=` - [相等和不等比较操作符](#) (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](#)
- `<`, `>`, `<=`, `>=` - [比较操作符](#) (对非基本类型, 会翻译为对 `compareTo()` 函数的调用)
- `[`, `]` - [下标访问操作符](#) (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](#)
- `?.` 执行一个 [安全调用](#) (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](#))
- `::` 创建一个 [成员的引用](#), 或者一个 [类引用](#)
- `..` 创建一个 [值范围](#)
- `:` 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](#)
- `->`
  - 在 [lambda 表达式](#) 中, 用作参数与函数体之间的分隔符
  - 在 [函数类型](#) 中, 用作参数与返回类型之间的分隔符
  - 在 [when 表达式](#) 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](#)
  - 定义, 或者引用一个 [循环标签](#)
  - 定义, 或者引用一个 [lambda 表达式标签](#)
  - 引用一个 [外层范围的 ‘this’ 表达式](#)
  - 引用一个 [外部类的超类](#)
- `;` 用于在同一行中分隔多条语句
- `$` 在 [字符串模板](#) 中引用变量或表达式
- `_`
  - 在 [lambda 表达式](#) 中代替未使用的参数
  - 在 [解构声明](#) 中代替未使用的参数

## 语法

参见 [Kotlin 官方网站](#)

# 与 Java 的互操作性

## 在 Kotlin 中调用 Java 代码

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. 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.

比如:

```
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()
    }
}
```

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 目前还不支持只写(set-only)的属性.

## 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, 当我们调用它的方法时, 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 则可能抛出异常
```

平台数据类型是 *无法指示的(non-denotable)*, 也就是说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, 我们可以依靠类型推断(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), 或者我们也可以选择我们期望的数据类型(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
```

如果我们选择使用非 `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. 当我们把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. 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, 等等), 因此, 我们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` ( `org.jetbrains.annotations` 包中定义的 `@Nullable` 和 `@NotNull` 注解)
- `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` ).

完整的列表请参见 [Kotlin 编译器源代码](#).

### 对类型参数添加注解

也可以对泛型的类型参数添加注解, 标记它是否可以为 null. 比如, 我们先来看看在 Java 中如何添加这些注解:

```
@NotNull
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) { ... }
```


这段 Java 程序在 Kotlin 中看到的函数签名如下:

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { ... }
```

请注意在类型参数 `String` 上的 `@NotNull` 注解. 如果没有这些注解, 我们得到的类型参数就只能是平台数据类型:

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { ... }
```

对类型参数的注解需要 Java 8 或更高版本的编译环境, 还要求可否为 null(Nullability) 注解支持 `TYPE_USE` (从版本 15 开始, `org.jetbrains.annotations` 支持 `TYPE_USE`).

 注意: 由于目前的技术限制, 对于那些通过依赖项引入的已编译过的 Java 库, IDE 不能正确识别这种类型参数上的注解.

### 对 JSR-305 规范的支持

[JSR-305 规范](#) 中定义了 [@Nonnull](#) 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.

从 Kotlin 1.1.50 开始, 还支持 [自定义可空限定符 \(KEEP-79\)](#) (详情请见下文).

类型限定符别名(Type qualifier nickname) (从 Kotlin 1.1.50 开始支持)

如果一个注解, 同时标注了 [@TypeQualifierNickname](#) 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) (从 Kotlin 1.1.50 开始支持)

[@TypeQualifierDefault](#) 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` (从 1.1.60 版开始支持) 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?
}

```

注意: 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 (从 1.1.60 版开始有效)

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;
- 使用 `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;
- 此外还有 `MigrationStatus.IGNORE`, 会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 {}

```

注意: 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} (从 1.1.60 开始支持) 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同的看法: 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} (从 1.1.60 开始支持) 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, @Nullable 以及 @CheckForNull 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。

对于 Kotlin 1.1.50+/1.2 版, 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	kotlin.Annotation!
java.lang.Deprecated	kotlin.Deprecated!
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>!

注意: 这些 Java 类型的静态成员, 无法通过 Kotlin 类型的 [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](#)). 将 Java 类型导入 Kotlin 时, 我们进行以下变换:

- 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, 也就是说, 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, 所以, `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 不允许我们将 `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, 这是允许(通过使用 `Array<(out) String>!` 形式的[平台数据类型](#)).

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)
}
```

即使我们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)
```

对于使用可变长参数的 Java 方法, 目前无法向它传递 `null` 参数.

## 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), 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](#).

### 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](#) 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](#), 这个功能从 Kotlin 1.1 开始支持. 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](#), 第 13 条: *要正确地覆盖 clone 方法*.

## finalize()

要覆盖 `finalize()` 方法, 你只需要声明它既可, 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)”. 我们不能将这样的”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.

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

与 Java 8 一样, Kotlin 支持 SAM(Single Abstract Method) 转换. 也就是说如果一个 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") })
```

注意, 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.

还应当注意, 这个功能只在 Kotlin 与 Java 互操作时有效; 由于 Kotlin 本身已经有了专门的函数类型, 因此没有必要将函数自动转换为 Kotlin 接口的实现者, Kotlin 也不支持这样的转换.

## 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.

## 在 Java 中调用 Kotlin

在 Java 中可以很容易地调用 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 `example.kt` 的 `org.foo.bar` 包内声明的所有函数和属性, 包括扩展函数, 都会被编译成为 Java 类 `org.foo.bar.ExampleKt` 的静态方法。

```
// example.kt
package demo

class Foo

fun bar() { ... }
```

```
// Java
new demo.Foo();
demo.ExampleKt.bar();
```

编译生成的 Java 类的名称, 可以通过 `@JvmName` 注解来改变:

```
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() { ... }
```

```
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

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` 注解) 这样的情况通常会被认为是错误. 但是, 编译器可以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` 注解.

```
// oldutils.kt
@file:jvmName("Utils")
@file:jvmMultifileClass

package demo

fun foo() { ... }
```

```
// newutils.kt
@file:jvmName("Utils")
@file:jvmMultifileClass

package demo

fun bar() { ... }
```

```
// Java
demo.Utils.foo();
demo.Utils.bar();
```

## 实例的域

如果希望将一个 Kotlin 属性公开为 Java 中的一个域, 你需要对它添加 `@JvmField` 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, 需要满足以下条件: 属性应该拥有后端域变量(backing field), 不是 `private` 属性, 没有 `open`, `override` 或 `const` 修饰符, 并且不是委托属性(delegated property).

```
class C(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(C c) {
        return c.ID;
    }
}
```

[延迟初始化属性](#) 也会公开为 Java 中的域. 域的可见度将与属性的 `lateinit` 的设置方法可见度一样.

## 静态域

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` 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
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](#)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
Singleton.provider = new Provider();
// 这里访问的是 Singleton 类中的 public static 非 final 域
```

使用 `const` 修饰符的属性(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:

```
// file example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

在 Java 中:

```
int c = Obj.CONST;
int d = ExampleKt.MAX;
int v = C.VERSION;
```

## 静态方法

上文中我们提到, 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` 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 foo() {}
        fun bar() {}
    }
}
```



现在, `foo()` 在 Java 中是一个静态方法, 而 `bar()` 不是:

```
C.foo(); // 正确
C.bar(); // 错误: 不是静态方法
C.Companion.foo(); // 实例上的方法仍然存在
C.Companion.bar(); // 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 foo() {}
    fun bar() {}
}
```

在 Java 中:

```
Obj.foo(); // 正确
Obj.bar(); // 错误
Obj.INSTANCE.bar(); // 正确, 这是对单体实例的一个方法调用
Obj.INSTANCE.foo(); // 也正确
```

`@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.

## 可见度

Kotlin 中的可见度会根据以下规则映射到 Java:

- `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 包内局部声明(package-local declaration);
- `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>
```

这两个函数是无法同时定义的, 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` 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` 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 Foo @JvmOverloads constructor(x: Int, y: Double = 0.0) {
    @JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") { ... }
}
```

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:

```
// Constructors:
Foo(int x, double y)
Foo(int x)

// Methods
void f(String a, int b, String c) {}
void f(String a, int b) {}
void f(String a) {}
```

注意, 在 [次级构造器](#) 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)

我们在上文中提到过, 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, 假如我们有一个这样的 Kotlin 函数:

```
// example.kt
package demo

fun foo() {
    throw IOException()
}
```

然后我们希望在 Java 中调用它, 并捕获异常:

```
// Java
try {
    demo.Example.foo();
}
catch (IOException e) { // 错误: foo() 没有声明抛出 IOException 异常
    // ...
}
```

这时 Java 编译器会报告错误, 因为 `foo()` 没有声明抛出 `IOException` 异常. 为了解决这个问题, 我们可以在 Kotlin 中使用 `@Throws` 注解:

```
@Throws(IOException::class)
fun foo() {
    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\(declaration-site variance\)](#), 那么这些类在 Java 代码中看到的形式存在两种可能. 假设我们有下面这样的类, 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 中我们可以这样: `unboxBase(boxDerived("s"))`, 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, 我们必须将 `unboxBase` 函数定义成这样:

```
Base unboxBase(Box<? extends Base> box) { ... }
```

这里我们使用了 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, `Box<Super>` 会成为 Java 的 `Box<? extends Super>` (对于定义了类型参数反向协变的 `Foo` 类, 会生成 Java 的 `Foo<? super Bar>`). 当类被用作返回值时, 编译产生的结果不会使用类型通配符, 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) { ... }
```

注意: 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.

如果我们需要类型通配符, 但默认没有生成, 那么可以使用 `@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, 但我们不需要它, 那么可以使用 `@JvmSuppressWildcards` 注解:

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// 将被翻译为
// Base unboxBase(Box<Base> box) { ... }
```

注意: `@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` 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() { ... }
```

# JavaScript

## 动态类型(Dynamic Type)

⚠ 当编译目标平台为 JVM 时, 不支持动态类型

Kotlin 虽然是一种静态类型的语言, 但它仍然可以与无类型或松散类型的环境互操作, 比如各种 JavaScript 环境. 为了为这样的使用场景提供帮助, Kotlin 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:

- 这个类型的值可以赋值给任意变量, 也可以作为参数传递给任何函数;
- 任何值都可以复制给 `dynamic` 类型的变量, 也可以传递给函数的 `dynamic` 类型参数;
- 对这些值不做 `null` 检查.

`dynamic` 类型最特殊的功能是, 允许我们对 `dynamic` 类型变量访问它的 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 平台上, 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](#) 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, 因此我们可以将这些调用自由地串联起来:

```
dyn.foo().bar.baz()
```

当我们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](#).

## 在 Kotlin 中调用 JavaScript

Kotlin 设计时很重视与 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, 因此它在编译时刻不做类型检查. 通过使用 [动态类型](#), 在 Kotlin 中你可以自由地与 JavaScript 交互, 但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 Kotlin 头文件.

### 内联 JavaScript

使用 `js("...")` 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")
}
```

`js` 函数的参数必须是字符串常量. 因此, 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"
```

### 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, 会由开发者提供, 因此它不会为这个声明生成 JavaScript 代码. 也就是说, 你应该省略 `external` 声明的 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, 也就是说, 在 `Node` 类的内部, 我们不需要在成员函数和属性之前添加 `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. 也就是:

```
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)` 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

`external` 函数可以拥有可选的参数. JavaScript 实现代码中具体如何为这些可选参数计算默认值, 对 Kotlin 是不可知的, 因此在 Kotlin 中, 我们无法使用通常的语法来定义这样的参数. 应该使用如下语法:

```
external fun myFunWithOptionalArgs(x: Int,
    y: String = definedExternally,
    z: Long = definedExternally)
```

这样, 你就可以使用一个必须参数和两个可选参数(其默认值由某些 JavaScript 代码计算得到)来调用 `myFunWithOptionalArgs` 函数.

### 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` 类, 然后通过非 `external` 类来扩展它. 比如:

```
external open class HTMLElement : Element() {
    /* 成员定义 */
}

class CustomElement : HTMLElement() {
    fun foo() {
        alert("bar")
    }
}
```

但存在以下限制:

1. 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.
2. 带默认参数的函数不能覆盖.

注意, 你不能使用 `external` 类来扩展非 `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` 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, 比如:

```
external interface HasFooAndBar {
    fun foo()

    fun bar()
}

external fun myFunction(p: HasFooAndBar)
```

`external` 接口的另一种使用场景, 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:

1. 它们不可以用在 `is` 检查语句的右侧.
2. 使用 `as` 将对象转换为 `external` 接口, 永远会成功 (并且在编译期间产生一个警告).
3. 它们不可以用作实体化的类型参数(reified type argument).
4. 它们不可以用在类的字面值表达式中(也就是 `I::class` ).

## 在 JavaScript 中调用 Kotlin

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, 你应该记住.

### 将声明隔离在独立的 JavaScript 对象内

为了避免破坏全局对象, Kotlin 会创建一个对象, 其中包含来自当前模块的所有 Kotlin 声明. 因此, 如果你将你的模块命名为 `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(详情请参见 [JavaScript 模块](#)), 就会出现兼容问题. 这时不会存在一个封装对象, 所有的声明会以相应的 JavaScript 模块的形式对外公开. 比如, 在 CommonJS 中你应该这样:

```
alert(require('myModule').foo());
```

### 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"
```

在 JavaScript 中应该这样访问:

```
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 中通过以下方式使用这个类:

```
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, Kotlin 编译器对 `external` 声明不会进行这样的名称混淆, 因此你不必对 `external` 声明使用 `@JsName` 注解. 另一种值得注意的情况是, 从 `external` 类继承来的非 `external` 类. 这种情况下, 所有被覆盖的函数名称都不会进行混淆.

`@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()
```

## Kotlin 类型在 JavaScript 中的表达

- 除 `kotlin.Long` 外, Kotlin 的数字类型都被映射为 JavaScript 的 `Number` 类型.
- `kotlin.Char` 被映射为 JavaScript 的 `Number`, 值为字符编码.
- Kotlin 在运行时无法区分数字类型(除 `kotlin.Long` 外), 也就是说, 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 保留了 `kotlin.Int`, `kotlin.Byte`, `kotlin.Short`, `kotlin.Char` 和 `kotlin.Long` 的溢出语义.
- 在 JavaScript 中没有 64 位整数, 所以 `kotlin.Long` 没有映射到任何 JavaScript 对象, 它会通过一个 Kotlin 类来模拟实现.
- `kotlin.String` 映射为 JavaScript 字符串.
- `kotlin.Any` 映射为 JavaScript 的 `Object` (也就是 `new Object()`, `{}`, 等等).
- `kotlin.Array` 映射为 JavaScript 的 `Array`.
- Kotlin 集合 (也就是 `List`, `Set`, `Map`, 等等) 不会映射为任何特定的 JavaScript 类型.
- `kotlin.Throwable` 映射为 JavaScript 的 `Error`.
- 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.

从 Kotlin 1.1.50 版开始, 基本类型的数组使用 JavaScript 的 `TypedArray` 来实现:

- `kotlin.ByteArray`, `-ShortArray`, `-IntArray`, `-FloatArray`, 以及 `-DoubleArray` 分别映射为 JavaScript 的 `Int8Array`, `Int16Array`, `Int32Array`, `Float32Array`, 和 `Float64Array`.
- `kotlin.BooleanArray` 映射为 JavaScript 的 `Int8Array`, 并且属性 `$type$ == "BooleanArray"`
- `kotlin.CharArray` 映射为 JavaScript 的 `UInt16Array`, 并且属性 `$type$ == "CharArray"`
- `kotlin.LongArray` 映射为 JavaScript 的 `kotlin.Long` 数组, 并且属性 `$type$ == "LongArray"`.

## JavaScript 模块(Module)

Kotlin 允许你将 Kotlin 工程编译为 JavaScript 模块(module), 支持各种常见的 JavaScript 模块系统. 以下是编译 JavaScript 模块的各种方式:

1. 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. 默认会使用这种方式.
2. [异步模块定义 \(Asynchronous Module Definition \(AMD\)\)](#), require.js 库使用的就是这个模块系统.
3. [CommonJS](#) 规约, 广泛使用于 node.js/npm ( `require` 函数和 `module.exports` 对象)
4. 统一模块定义(Unified Module Definitions (UMD)), 这种方式同时兼容于 *AMD* 和 *CommonJS*, 而且在运行时, 如果 *AMD* 和 *CommonJS* 都不可用, 则会以 “plain” 模式工作.

### 选择编译目标的模块系统

在各种编译环境中, 可以分别通过以下方法来选择编译目标的模块系统:

#### 使用 IDEA 时

对各个模块进行设置: 打开菜单 File -> Project Structure..., 找到你的模块, 然后在这个模块中选择 “Kotlin” facet. 在 “Module kind” 项中选择适当的模块系统.

对整个工程进行设置: 打开菜单 File -> Settings, 选择 “Build, Execution, Deployment” -> “Compiler” -> “Kotlin compiler”. 在 “Module kind” 项中选择适当的模块系统.

#### 使用 Maven 时

使用 Maven 编译时, 要选择模块系统, 你应该设置 `moduleKind` 配置属性, 也就是说, 你的 `pom.xml` 文件应该类似如下:

```
<plugin>
  <groupId>org.jetbrains.kotlin</groupId>
  <artifactId>kotlin-maven-plugin</artifactId>
  <version>${kotlin.version}</version>
  <executions>
    <execution>
      <id>compile</id>
      <goals>
        <goal>js</goal>
      </goals>
    </execution>
  </executions>
  <!-- 插入以下内容 -->
  <configuration>
    <moduleKind>commonjs</moduleKind>
  </configuration>
  <!-- 插入内容到此结束 -->
</plugin>
```

`moduleKind` 配置属性可以设置的值是: `plain`, `amd`, `commonjs`, `umd`.

#### 使用 Gradle 时

使用 Gradle 编译时, 要选择模块系统, 你应该设置 `moduleKind` 属性, 也就是:

```
compileKotlin2js.kotlinOptions.moduleKind = "commonjs"
```

这个属性可以设置的值与 Maven 中类似.

#### @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. 编译器允许将导入的 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 */ }
}
```

注意: 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, 让我们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

假如一个声明使用了 `@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, 以可以使用 `<script>` 元素来引用). 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, 你应该再加上 `@JsNonModule` 声明. 比如, JavaScript 代码如下:

```
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)
```

### 注意

Kotlin 将 `kotlin.js` 标准库作为一个单个的文件发布, 这个库本身作为 UMD 模块编译, 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. 而且, 在 NPM 中可以通过 [kotlin 包](#) 使用这个库.

## JavaScript 中的反射功能

目前, JavaScript 还不支持完整的 Kotlin 反射 API. 这部分 API 中唯一支持的是 `::class` 语法, 可以用来引用一个对象实例的类信息, 或者引用一个指定的类型的类信息. `::class` 表达式的值是 [KClass](#) 的一个简化版实现, 只支持 [simpleName](#) 和 [jsInstance](#) 成员函数.

除此之外, 你还可以使用 [KClass.js](#) 来获取某个类的 [JsClass](#) 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.

示例:

```
class A
class B
class C

inline fun <reified T> foo() {
    println(T::class.simpleName)
}

val a = A()
println(a::class.simpleName) // 获取对象实例的类信息; 打印结果为 "A"
println(B::class.simpleName) // 获取数据类型的类信息; 打印结果为 "B"
println(B::class.js.name)    // 打印结果为 "B"
foo<C>()                     // 打印结果为 "C"
```

## JavaScript DCE

从 1.1.4 版开始, Kotlin/JS 包含了一个死代码剔除(DCE, dead code elimination)工具. 这个工具可以从生成的 JS 中剔除未被使用的属性, 函数, 以及类. 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).
- 你使用了一个共享库, 其中提供的函数远远超过你实际需要的. 比如, Kotlin 标准库( `kotlin.js` ) 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, 文件大小总计 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.

死代码剔除通常也叫做 ‘tree shaking’ .

### 如何使用

目前可以通过 Gradle 使用 DCE 工具.

要激活 DCE 工具, 请将以下内容添加到 `build.gradle` :

```
apply plugin: 'kotlin-dce-js'
```

注意, 如果你正在使用多工程编译, 那么应该将这个 plugin 用在主工程上, 主工程就是你的应用程序入口.

默认情况下, 编译输出的 JavaScript 文件集 (你的应用程序加上所有的依赖项) 会位于 `$BUILD_DIR/min/` 路径下, 其中 `$BUILD_DIR` 是编译产生的 JavaScript 的输出路径(通常为, `build/classes/main` ).

### 配置

要对主代码集配置 DCE, 你可以使用 `runDceKotlinJs` 编译任务(其他的源代码集对应的编译任务名为 `runDce<sourceSetName>KotlinJs` ).

有时你会在 JavaScript 中直接使用 Kotlin 声明, 然而这个声明被 DCE 工具消除了. 你可能会希望保留这个声明. 为了实现这个目的, 你可以在 `build.gradle` 中使用以下语法:

```
runDceKotlinJs.keep "declarationToKeep", "declarationToKeep", ...]
```

其中 `declarationToKeep` 的语法如下:

`moduleName.dot.separated.package.name.declarationName`

比如, 假设一个名为 `kotlin-js-example` 的模块, 其中包含名为 `org.jetbrains.kotlin.examples` 的包, 包中包含一个名为 `toKeep` 的函数, 这时, 请使用以下声明:

```
runDceKotlinJs.keep "kotlin-js-example_main.org.jetbrains.kotlin.examples.toKeep"
```

注意, 如果你的函数有参数, 那么函数名会被混淆, 因此在 `keep` 命令中应该使用混淆后的函数名.

### 开发模式

运行 DCE 会导致每次编译时都消耗一些额外的时间, 而且在开发过程中我们并不在意编译输出的结果文件大小. 因此在开发过程中, 可以对 DCE 编译任务指定 `dceOptions.devMode` 选项, 让 DCE 工具跳过死代码剔除处理, 以便缩短编译时间.

比如, 如果想要对 `main` 源代码集按照某种自定义的条件来禁用 DCE 处理, 并且对 `test` 源代码始终禁用 DCE, 那么可以在编译脚本中加入以下设置:

```
runDceKotlinJs.dceOptions.devMode = isDevMode
runDceTestKotlinJs.dceOptions.devMode = true
```

## 示例



[这里](#) 是一个完整的示例, 演示如何集成 DCE 和 webpack, 将 Kotlin 代码编译产生一个尺寸很小的 JavaScript bundle.

## 注意事项

- 在 Kotlin 1.1.x 版中, DCE 工具是一个 实验性 功能. 这并不代表我们会删除它, 也不代表它不能用于生产环境. 而是说, 将来我们可能改变配置参数的名称, 默认值, 等等.
- 目前, 如果你的工程是一个共享库, 你不应该使用 DCE 工具. 只有当你在开发一个应用程序(可能使用到标准库)时, 才应该使用它. 理由是: DCE 不知道库的哪部分代码会被用户的应用程序使用到.
- DCE 不会通过删除不必要的空格, 缩短标识符名称等手段, 对你的代码进行最小化(丑化(uglifyfication))处理. 要完成这种任务, 你应该使用既有的工具, 比如 UglifyJS (<https://github.com/mishoo/UglifyJS2>), 或 Google Closure Compiler (<https://developers.google.com/closure/compiler/>).

# 原生(Native)程序开发

## Kotlin/Native 中的并发

Kotlin/Native 平台不鼓励使用经典的面向线程的并发模型, 包括互斥代码段, 以及有条件的变量, 因为这种模型易于出错, 可靠性差. 相反, 我们建议使用一组替代方案, 可以帮助你使用硬件并发, 并实现阻塞式 IO. 这些方案如下, 我们会在后面的章节中进行详细的介绍:

- 使用消息传递的 Worker
- 对象子图(Object subgraph)的所有权转换(ownership transfer)
- 对象子图的冻结(freezing)
- 对象子图的分离(detachment)
- 使用 C 全局变量的原生共享内存(Raw shared memory)
- 用于阻塞操作的协程 (本文档不作介绍)

### Worker

Kotlin/Native 平台不鼓励使用线程, 而是提供了工作者(Worker)的概念: 与请求队列相关联, 并且并发执行的一系列控制流. Worker 与 Actor 模型中的 actor 非常类似. 一个 worker 可以与另一个 worker 交换 Kotlin 对象, 因此任何一个时刻, 每个可变对象都只属于唯一的一个 worker, 但这种所有权可以转换. 详情请参见 [对象传输\(transfer\)与冻结\(freezing\)](#).

通过 `Worker.start` 函数调用启动一个 worker 之后, 可以通过它的唯一整数 worker id 来追踪它. 其他 worker, 或者非 worker 的并发元素, 比如 OS 线程, 可以使用 `execute` 函数调用来向它发送消息.

```
val future = execute(TransferMode.SAFE, { SomeDataForWorker() }) {
    // 第 2 个函数参数所返回的数据, 会被传入到 worker 程序内, 成为 'input' 参数
    input ->
    // 这里我们创建一个实例, 当某个外面的某段代码消费 worker 的结果值时, 就会返回这个实例.
    WorkerResult(input.stringParam + " result")
}

future.consume {
    // 在这里我们会看到上面的 worker 程序返回的结果.
    // 注意, future 对象或 id 可以被传输给另一个 worker,
    // 因此我们并不需要在得到这个 future 的同一个执行上下文内消费它.
    result -> println("result is $result")
}
```

调用 `execute` 时使用一个函数作为第 2 个参数, 这个函数负责生产一个对象子图(object subgraph)(也就是, 一组可变的引用对象), 这个对象子图再被整体传递给 worker, 然后, 在启动这个请求的线程内它就不再能够访问了. 如果第 1 个参数是 `TransferMode.SAFE`, 那么会在访问对象子图时检查这个属性, 如果第 1 个参数是 `TransferMode.UNSAFE`, 那么会假定这个属性为 true. `execute` 的最后一个参数是一个特殊的 Kotlin Lambda 表达式, 它不允许捕捉任何状态数据, 而且会在目标 worker 的上下文中调用它. 一旦处理完毕, 结果就会被传输给某个未来的消费者, 它会被绑定在这个 worker/线程 的对象图上.

如果使用 `UNSAFE` 模式来传输对象, 而且还可以在多个并发执行器中访问到它, 那么程序可能会意外崩溃, 因此这种方式只能作为程序优化时的最终手段, 而不要用作通常的编程机制.

关于更完整的示例程序, 请参见 Kotlin/Native 代码仓库中的 [worker 示例程序](#).

### 对象传输(transfer)与冻结(freezing)

Kotlin/Native 平台维持的一个重要的不变性(invariant)机制就是, 对象要么属于单个线程或 worker, 否则它就必须是不可变的 (要么是共享的不可变对象, 要么是不共享的可变对象). 这个机制保证同一个数据只可能被单个访问者修改, 因此不需要锁的存在. 为了实现这个不变性(invariant)机制, 我们使用一种概念, 就是不被外接引用的对象子图(object subgraph). 对象子图是指, 不存在来自这个子图外部的引用, 这一点可以使用(在 ARC 系统中)复杂度为  $O(N)$  的算法进行检查, 其中  $N$  是子图中元素的个数. 这种子图通常由一个 Lambda 表达式(比如某些构建器)的结果产生, 而且可能也不包含引用到外部对象的对象.

冻结(Freezing) 是一种运行期操作, 使得一个给定的对象子图不可变更, 通过修改对象头(object header), 因此将来试图修改对象将会抛出 `InvalidMutabilityException` 异常. 冻结是一种深度操作, 因此如果一个对象拥有指向其他对象的指针 - 那么由这些对象构成的一个传递性的包, 全部都会被冻结. 冻结是一种单方向的变换, 冻结后的对象不能被解冻. 由于冻结后的对象不可变更, 因此它有一个很好的性质, 它可以在多个 worker/线程 之间共享, 而不会破坏 “要么是共享的不可变对象, 要么是不共享的可变对象” 不变性原则.

对象是否被冻结, 可以通过扩展属性 `isFrozen` 来检查, 如果已被冻结, 那么对象的共享是被允许的. 目前, Kotlin/Native 平台只会在创建后冻结枚举对象, 但是将来可能会自动冻结其他确定不可变的对象.

### 对象子图的分离(detachment)

不带外部引用的对象子图可以使用 `DetachedObjectGraph<T>` 来解除连接, 变成一个 `COpaquePointer` 值, 这个值可以保存在 `void*` 数据中, 因此解除连接后的对象子图可以保存到 C 数据结构中, 然后将来在任意的线程或 worker 中, 使用

`DetachedObjectGraph<T>.attach()` 连接回来. 如果对于某个任务 worker 机制还不足以完成任务, 这个功能与 [原生的共享内存\(raw memory sharing\)](#) 结合在一起, 可以实现在并发的线程中传输旁路对象(side channel object).

### 原生的共享内存(Raw shared memory)

考虑到 Kotlin/Native 和 C 交互时极强的关联性, 将上文提到的那些机制结合在一起, 可以使用 Kotlin/Native 创建出常用的数据结构, 比如并发的 `hashmap`, 或共享的缓存. 这些数据结构可以依赖于共享的 C 数据, 并且将分离后的对象子图的引用保存在 C 数据中. 我们来看看下面的 .def 文件:

```
package = global

---
typedef struct {
    int version;
    void* kotlinObject;
} SharedData;

SharedData sharedData;
```

运行 `cinterop` 工具后, 这段代码可以使用一个带版本号的全局数据结构来共享 Kotlin 数据, 使用自动生成的 Kotlin 代码, 可以在 Kotlin 代码中与这段代码透明地互动, 如下:

```
class SharedData(rawPtr: NativePtr) : CStructVar(rawPtr) {
    var version: Int
    var kotlinObject: COpaquePointer?
}
```

因此, 结合上面声明的顶级变量, 可以用来从不同的线程访问同一块内存, 并使用平台相关的同步机制, 创建出传统的并发数据结构.

### 全局变量与单子(singleton)

很多时候, 全局变量会导致预料之外的并发错误, 因此 *Kotlin/Native* 实现了以下机制, 来避免预料之外的通过全局对象共享状态数据:

- 全局变量, 除非明确指定, 只能从主线程中访问 (也就是, *Kotlin/Native* 平台最早初始化的那个线程), 如果其他线程访问了这样的全局变量, 会抛出 `IncorrectDereferenceException` 异常
- 对标注了 `@kotlin.native.ThreadLocal` 注解的全局变量, 每个线程都会保存一份本线程内的 copy, 因此线程之间相互不会受影响
- 对标注了 `@kotlin.native.SharedImmutable` 注解的全局变量, 会被所有的线程共享, 但在此之前先会被冻结, 因此所有的线程都会读到相同的值
- 单子对象(singleton object), 如果没有标注 `@kotlin.native.ThreadLocal` 注解, 也会被冻结并共享, 允许使用延迟初始化的值(lazy value), 除非试图创建循环的冻结结构(cyclic frozen structures)

— 枚举值总是会被冻结

将这些机制结合在一起, 可以在多平台项目(MPP project)中实现跨平台的代码重用, 实现非常自然的 race-freeze (???意思不明???) 编程.

## Kotlin/Native 中的不变性(Immutability)

Kotlin/Native 实现了严格的可变性检查, 以确保重要的不变性(invariant)机制, 也就是, 对象在任何时刻, 要么是不可变的, 要么只能从单个线程访问 (要么是不共享的可变对象, 要么是共享的不可变对象)。

在 Kotlin/Native 中, 不变性是一种运行期特性, 可以通过 `kotlin.native.concurrent.freeze` 函数应用在任意的对象子图上。它会使得从某个给定的对象开始, 所能访问到的全部对象都成为不可变的对象, 这样的转换是一种单向转换 (也就是说, 冻结后的对象以后不能解冻)。有些天然的不可变对象, 比如 `kotlin.String`, `kotlin.Int`, 以及其他基本类型, 以及 `AtomicInt` 和 `AtomicReference` 默认就是冻结的。如果对一个冻结后的对象执行一个变更其值的操作, 会抛出一个 `InvalidMutabilityException` 异常。

为了实现 要么是不共享的可变对象, 要么是共享的不可变对象 机制, 所有的全局可见状态值 (目前包括, `object` 单子, 以及枚举值) 会自动冻结。如果不希望冻结某个对象, 可以使用 `kotlin.native.ThreadLocal` 注解, 它会将对象变成线程局部变量(thread local), 而且是可变的 (但它的值的变化对其他线程是不可见的)。

非基本类型的顶层或全局变量, 默认只能在主线程中访问(也就是, *Kotlin/Native* 平台最早初始化的那个线程)。如果其他线程访问了这样的变量, 会抛出 `IncorrectDereferenceException` 异常。如果想让这些变量可以在其他线程中访问, 你可以使用 `@ThreadLocal` 注解, 将它变成线程局部变量, 或者使用 `@SharedImmutable` 注解, 让对象冻结, 并允许从其他线程访问。

`AtomicReference` 类可以用来将变更后的冻结状态值公布给其他线程, 因此可以构建共享缓存这样的模式。

## Kotlin/Native 库

### 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 foo.kt -p library -o bar
```

上例的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 qux.kt -l bar
```

上例的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/git churn/src/nativeInterop/cinterop/libgit2.def -compilerOpts -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](#)

### 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.

列出库的内容:

```
$ klib contents <name>
```

查看库的内容细节:

```
$ klib info <name>
```

要把库安装到默认的位置, 可以使用:

```
$ klib install <name>
```

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 kotlinizer.kt -p library -o kotlinizer
```

库会被创建到当前目录下:

```
$ ls kotlinizer.klib
kotlinizer.klib
```

现在来看看库的内容:

```
$ klib contents kotlinizer
```

我们可以将 `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, 并链接我们刚才创建的库:

```
$ kotlinc 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(默认仓库现在是 `~/konan`` 目录, 但可以设置 `**KONAN_DATA_DIR**` 环境变量来修改这个路径).
- \* ``$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/` 后会得到以下内容:

```
- foo/
- 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.



# 平台库

## 概述

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 iPhone ) 我们提供了 `posix` 平台库. 其中包含对各平台的 POSIX 标准实现的绑定.

要使用这个库, 只需要:

```
import platform.posix.*
```

唯一不能使用这个库的平台是 [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](#) 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

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.

## 示例

Kotlin/Native 的发布版提供了大量的示例程序, 演示平台库的使用方法. 详情请参见 [示例程序](#).

## Kotlin/Native 的互操作性

### 简介

*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需要的一切.

与原生库交互时的工作流程如下.

- 创建一个 `.def` 文件, 描述需要绑定(binding)的内容
- 使用 `cinterop` 工具生成绑定
- 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](#).

### 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](#). 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/git churn
../../dist/bin/cinterop -def src/main/c_interop/libgit2.def \
  -compilerOpts -I/usr/local/include -o libgit2
```

编译客户端代码:

```
../../dist/bin/kotlinc src/main/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 -compilerOpts -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, 编译时可以作为 `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 glob 过滤头文件。`.def` 文件内的 `headerFilter` 属性值会被看作一个空格分隔的 glob 列表。如果包含的头文件与任何一个 glob 匹配, 那么这个头文件中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, 比如, `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代码来过滤头文件:

```
headerFilter = SomeLibrary/**
```

如果没有指定 `headerFilter`, 那么所有的头文件都会被引入。

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 \(module map file\)](#) 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.mac_x64 = -DFOO=foo2
```

这样, Linux 上的 C 头文件会使用 `-DBAR=bar -DFOO=foo1` 参数, macOS 上则会使用 `-DBAR=bar -DFOO=foo2` 参数。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。

## 添加自定义声明

在生成绑定之前, 有时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 `staticLibraries` 和 `libraryPaths` 语句. 比如:

```
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`, 库文件会被自动链接到你的程序内.

## 使用绑定

### 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, 或整数值, 由 heuristic 以及 [定义文件中的提示](#) 决定.
- 结构体(Struct)或联合体(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")?.toString() ?: ""
```

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)
```

## 使用字符串

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>.toString(): 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.

如果回调不在主线程中运行, 那么就必须通过调用 `kotlin.native.initRuntimeIfNeeded()` 来启动 *Kotlin/Native* 运行时.

## 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, 可以有符号整数, 也可以是无符号整数。

`.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.

## Kotlin/Native 与 Swift/Objective-C 的交互能力

本章介绍 Kotlin/Native 与 Swift/Objective-C 的交互能力的一些细节。

### 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 [Gradle 插件](#) 中的“使用 cinterop”小节。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 [Gradle 插件](#) 中的“目标平台与输出类型”小节)。我们提供了一个例子, 请参见 [calculator 示例程序](#)。

### 映射

下表展示了 Kotlin 中的各种概念与 Swift/Objective-C 的对应关系。

Kotlin	Swift	Objective-C	注意事项
class	class	@interface	<a href="#">名称翻译</a>
interface	protocol	@protocol	
constructor/create	Initializer	Initializer	<a href="#">初始化器</a>
Property	Property	Property	<a href="#">顶层函数和属性</a>
Method	Method	Method	<a href="#">顶层函数和属性 方法名称翻译</a>
@Throws	throws	error:(NSError**)error	<a href="#">错误与异常</a>
Extension	Extension	Category member	<a href="#">Category 成员</a>
companion 成员	Class method 或 property	Class method 或 property	
null	nil	nil	
Singleton	Singleton()	[Singleton singleton]	<a href="#">Kotlin 单子(singleton)</a>
基本类型	基本类型 / NSNumber		<a href="#">NSNumber</a>
Unit 类型返回值	Void	void	
String	String	NSString	
String	NSMutableString	NSMutableString	<a href="#">NSMutableString</a>
List	Array	NSArray	
MutableList	NSMutableArray	NSMutableArray	
Set	Set	NSSet	
MutableSet	NSMutableSet	NSMutableSet	<a href="#">集合</a>
Map	Dictionary	NSDictionary	
MutableMap	NSMutableDictionary	NSMutableDictionary	<a href="#">集合</a>
Function 类型	Function 类型	Block pointer 类型	<a href="#">Function 类型</a>

### 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。

### 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。

## 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的, 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)
```

## 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. 这种情况下所有的 Kotlin 异常( `Error` , `RuntimeException` 及其子类除外) 都被翻译为 Swift error 或 `NSError` .

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.

## Category 成员

Objective-C Category 的成员, 以及 Swift extension 的成员, 导入 Kotlin 时会变成扩展函数. 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.

## 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. 这个实例可以通过工厂方法访问, 在 Objective-C 中是 `[MySingleton mySingleton]` 方法, 在 Swift 中是 `MySingleton()` 方法.

## 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 Kotlin 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()  
}
```

## 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.

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](#), 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).

# Kotlin/Native Gradle 插件

## 注意

本章介绍的是实验性的 Kotlin/Native Gradle 插件, 不是 IDE 中支持的 Kotlin 插件, 也不是跨平台项目插件. 关于跨平台项目 Gradle 插件, 请参见 [相关文档](#).

## 概述

*Kotlin/Native* 项目的编译需要使用 Gradle 插件. 在 Gradle 插件库中可以找到 [这个插件](#), 因此你可以通过 Gradle plugin DSL 来应用这个插件:

```
plugins {  
    id "org.jetbrains.kotlin.platform.native" version "1.3.0-rc-146"  
}
```

也可以从二进制文件仓库中得到这个插件. 除了正式发布版之外, 仓库中还包含了这个插件的旧版本, 以及开发中的版本, 这些非正式发布版通过 Gradle 插件库是无法得到的. 要从二进制文件仓库中得到这个插件, 请在你的编译脚本中添加以下代码:

```
buildscript {  
    repositories {  
        mavenCentral()  
        maven {  
            url "https://dl.bintray.com/jetbrains/kotlin-native-dependencies"  
        }  
    }  
  
    dependencies {  
        classpath "org.jetbrains.kotlin:kotlin-native-gradle-plugin:1.3.0-rc-146"  
    }  
}  
  
apply plugin: 'org.jetbrains.kotlin.platform.native'
```

默认情况下, 插件第一次运行时会上下载 Kotlin/Native 编译器. 如果你已经手动下载过编译器, 你可以使用项目属性 `org.jetbrains.kotlin.native.home` 来指定编译器根路径 (比如, 在 `gradle.properties` 文件中指定).

```
org.jetbrains.kotlin.native.home=/home/user/kotlin-native-0.8
```

这种情况下, 插件不会再去下载编译器.

## 源代码管理

`kotlin.platform.native` 插件的源代码管理方式与其他 Kotlin 插件是一致的, 都是按源代码集(source set)为单位进行管理. 一个源代码集就是一组 Kotlin/Native 源代码文件, 其中包含共通代码, 也包含平台相关代码. 插件提供一个顶层的编译脚本代码段 `sourceSets`, 可以用来配置源代码集. 插件还会插件默认的源代码集 `main` 和 `test` (分别用于生产代码和测试代码).

生产代码默认放在 `src/main/kotlin` 目录下, 测试代码默认放在 `src/test/kotlin` 目录下.

```
sourceSets {  
    // 添加与编译目标平台无关的代码.  
    main.kotlin.srcDirs += 'src/main/mySources'  
  
    // 添加 Linux 平台独有的代码. 这些代码只会在 Linux 目标平台中编译.  
    main.target('linux_x64').srcDirs += 'src/main/linux'  
}
```

## 目标平台与输出类型

插件默认会为 main 和 test 源代码集创建组件。要访问这些组件, 你使用通过 Gradle 的 `components` 容器, 也可以使用对应的源代码集的 `component` 属性:

```
// main 组件.
components.main
sourceSets.main.component

// test 组件.
components.test
sourceSets.test.component
```

通过组件你可以指定:

- 目标平台 (比如 Linux/x64 或 iOS/arm64 等等)
- 输出类型 (比如 可执行文件, 库, 框架 等等)
- 依赖项目 (包括 interop 依赖项目)

可以通过对应的组件的属性来设置它的目标平台:

```
components.main {
    // 针对 64-bit MacOS, Linux 和 Windows 平台编译这个组件.
    targets = ['macos_x64', 'linux_x64', 'mingw_x64']
}
```

关于目标平台的名称, 编译插件使用与编译器相同的表达方式. test 组件默认使用与 main 组件相同的目标平台.

输出类型也可以通过专门的属性来执行:

```
components.main {
    // 将这个组件编译为可执行文件, 以及 Kotlin/Native 库.
    outputKinds = [EXECUTABLE, KLIBRARY]
}
```

这里用到的所有的常数, 都可以用在组件的配置脚本代码段中. 编译插件支持输出以下类型的二进制文件:

- `EXECUTABLE` - 可执行文件;
- `KLIBRARY` - Kotlin/Native 库 (\*.klib);
- `FRAMEWORK` - Objective-C 框架;
- `DYNAMIC` - 原生的共享库;
- `STATIC` - 原生的静态库.

每个原生二进制文件又被编译为两个版本 (编译类型): `debug` 版(可调试, 无代码优化) 和 `release` 版(不可调试, 有代码优化). 注意, Kotlin/Native 库只有 `debug` 版, 因为只在编译最终二进制文件(可执行文件, 静态库, 等等)时才会进行代码优化, 并且会对最终二进制文件所需要用到的全部库文件进行代码优化.

## 编译任务

编译插件会对所有目标平台, 输出类型, 以及编译类型的组合, 分别创建编译任务. 这些编译任务的命名规约如下:

`compile<ComponentName><BuildType><OutputKind><Target>KotlinNative`

比如 `compileDebugKlibraryMacos_x64KotlinNative`, `compileTestDebugKotlinNative`.

编译任务名称包含以下组成部分 (其中某些部分可能为空):

- `<ComponentName>` - 组件名称. 对 main 组件来说, 组件名称为空.
- `<BuildType>` - `Debug` 或 `Release`.
- `<OutputKind>` - 输出类型名称, 比如 `Executable` 或 `Dynamic`. 如果组件只有一种输出类型, 则输出类型名称为空.

— `<Target>` - 组件的编译目标平台名称, 比如 `Macos_x64` 或 `Wasm32` . 如果组件只对一种目标平台编译, 则目标平台名称为空.

编译插件还会创建一些复合任务, 可以用来对某个编译类型编译所有相关的二进制文件 (比如, `assembleAllDebug` ), 或者对某个目标平台编译所有的二进制文件 (比如, `assembleAllWasm32` ).

另外还创建了基本的编译环节任务, 比如 `assemble` , `build` , 以及 `clean` .

## 运行测试程序

编译插件会对 `test` 组件的所有目标平台编译产生测试用的可执行文件. 如果当前机器的平台也是 `test` 组件的目标平台之一, 那么还会创建测试的运行任务. 要运行测试, 你可以执行标准的编译环节 `check` 任务:

```
./gradlew check
```

## 依赖项目

编译插件允许你使用传统的 Gradle 配置机制, 声明针对其他文件或项目的依赖. 编译插件支持 Kotlin 跨平台项目, 允许你声明 `expectedBy` 依赖项:

```
dependencies {
    implementation files('path/to/file/dependencies')
    implementation project('library')
    testImplementation project('testLibrary')
    expectedBy project('common')
}
```

我们可以依赖于已经发布到 maven 仓库的 Kotlin/Native 库. 编译插件需要使用 Gradle 的 [metadata](#) 功能, 因此这个功能需要启用. 请将下面的设置添加到你的 `settings.gradle` 文件中:

```
enableFeaturePreview('GRADLE_METADATA')
```

然后, 你可以使用传统的 `group:artifact:version` 表达方式, 声明对 Kotlin/Native 库的依赖:

```
dependencies {
    implementation 'org.sample.test:mylibrary:1.0'
    testImplementation 'org.sample.test:testlibrary:1.0'
}
```

依赖声明也可以定义在组件的代码段内:

```
components.main {
    dependencies {
        implementation 'org.sample.test:mylibrary:1.0'
    }
}

components.test {
    dependencies {
        implementation 'org.sample.test:testlibrary:1.0'
    }
}
```

## 使用 cinterop 工具

可以对组件声明一个 `cinterop` 依赖:



```

components.main {
    dependencies {
        cinterop('mystdio') {
            // 会使用 src/main/c_interop/mystdio.def 作为 def 文件.

            // 设置编译器参数
            compilerOpts '-I/my/include/path'

            // 可以对不同的编译目标平台设置不同的编译参数
            target('linux') {
                compilerOpts '-I/linux/include/path'
            }
        }
    }
}

```

这样, 就会编译一个 interop 库, 然后将它添加为组件的依赖项.

对于使用 interop 的 Kotlin/Native 库, 我们经常会需要指定一些与平台相关的链接参数. 可以使用 `target` 代码段来实现:

```

components.main {
    target('linux') {
        linkerOpts '-L/path/to/linux/libs'
    }
}

```

`allTargets` 代码段也可以这样使用.

```

components.main {
    // 对所有目标平台进行配置.
    allTargets {
        linkerOpts '-L/path/to/libs'
    }
}

```

## 发布

当 `maven-publish` 插件存在时, 对所有二进制文件的编译任务都会创建相应的发布任务. 这个插件使用 Gradle 的 `metadata` 来发布文件, 因此这个功能需要启用 (详情请参见 [依赖项目](#) 小节).

然后你就可以使用标准的 Gradle `publish` 任务来发布二进制文件了:

```
./gradlew publish
```

目前只会发布 `EXECUTABLE` 和 `KLIBRARY` 类型的二进制文件.

发布插件允许你使用 `pom` 代码段, 自定义发布时生成的 pom 文件, 这个代码段可以在所有的组件内使用:

```

components.main {
    pom {
        withXml {
            def root = asNode()
            root.appendNode('name', 'My library')
            root.appendNode('description', 'A Kotlin/Native library')
        }
    }
}

```

## 序列化插件

这个插件随 `kotlin.serialization` 插件的一个定制版本一起发布. 要使用这个插件, 你不需要添加新的编译脚本依赖项, 只需要应用这个插件, 并添加对序列化库的依赖项:

```
apply plugin: 'org.jetbrains.kotlin.platform.native'
apply plugin: 'kotlinx-serialization-native'

dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-serialization-runtime-native'
}
```

详情请参见 [示例项目](#).

## DSL 示例

本节我们展示一个带详细注释的 DSL. 也请参考使用这个插件的示例工程, 比如, [Kotlinx.coroutines](#), [MPP http client](#)

```
plugins {
    id "org.jetbrains.kotlin.platform.native" version "1.3.0-rc-146"
}

sourceSets.main {
    // 插件使用的 Gradle 源代码目录在这里设置,
    // 因此这里可以调用 SourceDirectorySet 内所有可用的 DSL 方法.
    // 独立于平台的源代码.
    kotlin.srcDirs += 'src/main/customDir'

    // Linux 平台专有的源代码
    target('linux').srcDirs += 'src/main/linux'
}

components.main {

    // 设置编译的目标平台
    targets = ['linux_x64', 'macos_x64', 'mingw_x64']

    // 设置编译输出类型
    outputKinds = [EXECUTABLE, KLIBRARY, FRAMEWORK, DYNAMIC, STATIC]

    // 对可执行文件指定自定义的执行入口点
    entryPoint = "org.test.myMain"

    // 目标平台相关的选项
    target('linux_x64') {
        linkerOpts '-L/linux/lib/path'
    }

    // 与平台无关的选项
    allTargets {
        linkerOpts '-L/common/lib/path'
    }

    dependencies {

        // 依赖一个已发布的 Kotlin/Native 库.
        implementation 'org.test:mylib:1.0'

        // 依赖一个项目
        implementation project('library')

        // Cinterop 依赖
    }
}
```

```

cinterop('interop-name') {
    // 描述原生 API 的 def 文件.
    // 默认路径是 src/main/c_interop/<interop-name>.def
    defFile project.file("defFile.def")

    // 存放生产的 Kotlin API 的包
    packageName 'org.sample'

    // cinterop 工具传递给编译器和链接器的选项.
    compilerOpts 'Options for native stubs compilation'
    linkerOpts 'Options for native stubs'

    // 需要解析的其他头文件.
    headers project.files('header1.h', 'header2.h')

    // 头文件的查找路径.
    includeDirs {
        // Project.file 方法所能够接受的所有对象, 都可以用于这两个设置项.

        // 头文件的查找路径 (类似于编译器的 -I<path> 选项).
        allHeaders 'path1', 'path2'

        // def 文件的 'headerFilter' 设置项中列出的头文件的查找路径.
        // 类似于编译器的 -headerFilterAdditionalSearchPrefix 选项.
        headerFilterOnly 'path1', 'path2'
    }
    // includeDirs.allHeaders 设置项的一个缩写.
    includeDirs "include/directory" "another/directory"

    // 传递给 cinterop 工具的其他命令行选项.
    extraOpts '-verbose'

    // 对 Linux 平台的其他配置.
    target('linux') {
        compilerOpts 'Linux-specific options'
    }
}

// 二进制文件发布时的其他 pom 设置.
pom {
    withXml {
        def root = asNode()
        root.appendNode('name', 'My library')
        root.appendNode('description', 'A Kotlin/Native library')
    }
}

// 传递给编译器的其他选项.
extraOpts '--time'
}

```

## 调试

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

### 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, 只需要在命令行添加 `-g` 选项.

示例:

```
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 = Point(x=1, y=2)
(lldb) p p
(ObjHeader *) $2 = Point(x=1, y=2)
(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 -- Konan_DebugPrint(a_variable)
(a_variable) one is 1(KInt) $0 = 0
(lldb)

```

## 已知的问题

— Python 绑定的性能问题.

**注意:** 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.

## 代码覆盖率(Code Coverage)

Kotlin/Native 有一个代码覆盖率统计工具, 它是基于 Clang 的 [Source-based Code Coverage](#) 开发的.

注意:

1. 代码覆盖率统计工具还处于非常早期的阶段, 并在活跃开发中. 目前已知的问题和限制包括:
  - 代码覆盖率统计结果可能不准确.
  - 单行的执行次数统计结果可能不准确.
  - 只支持 macOS 和 iOS 模拟器二进制文件.
2. 本章介绍的大部分功能将来会被合并到 Gradle plugin 中.

### 使用方法

TL;DR

```
kotlinc-native main.kt -Xcoverage
./program.kexe
llvm-profdata merge program.kexe.profrac -o program.profrac
llvm-cov report program.kexe -instr-profile program.profrac
```

编译时打开代码覆盖率选项

有 2 个编译器选项, 可以生成代码覆盖率信息:

- `-Xcoverage`. 对当前项目源代码生成代码覆盖率信息.
- `-Xlibrary-to-cover=<path>`. 对指定的 `klib` 库生成代码覆盖率信息. 注意, 这个库本身链接时也需要使用 `-library/-l` 参数, 或者需要是一个传递性的依赖项.

运行带代码覆盖率信息的可执行文件

编译产生的二进制文件(比如, `program.kexe`)运行之后, 会生成 `program.kexe.profrac` 文件. 默认情况下这个文件会产生在二进制文件相同的路径下. 有两种方法可以改变这个文件的输出路径:

- `-Xcoverage-file=<path>` 编译选项.
- `LLVM_PROFILE_FILE` 环境变量. 因此, 如果你使用这样的方法运行你的程序:  
`LLVM_PROFILE_FILE=build/program.profrac ./program.kexe`

代码覆盖率信息将被存储到 `build` 目录的 `program.profrac` 文件内.

解析 \*.profrac 文件

生成的代码覆盖率信息文件可以使用 `llvm-profdata` 工具程序来解析. 基本用法是:

```
llvm-profdata merge default.profrac -o program.profrac
```

更多命令行选项请参见 [目录行指南](#).

创建报告

最后一步是使用 `program.profrac` 文件创建报告. 可以使用 `llvm-cov` 工具程序来完成 (具体用法请参见 [目录行指南](#)). 比如, 我们可以通过以下命令来查看基本报告:

```
llvm-cov report program.kexe -instr-profile program.profrac
```

也可以使用以下命令来生成 html 格式的, 各代码行的覆盖率报告:

```
llvm-cov show program.kexe -instr-profile program.profrac -format=html > report.html
```



## 示例

通常, 会在运行测试程序的过程中收集代码覆盖率信息. 具体做法请参见 [samples/coverage](#).

## 相关资料

- [LLVM 代码覆盖率映射格式](#)

**Q: 我要怎样运行我的程序?**

A: 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`.

**Q: Kotlin/Native 的内存管理机制是怎样的?**

A: Kotlin/Native 提供一种自动化的内存管理机制, 与 Java 和 Swift 类似. 目前的内存管理器的实现包括, 自动的引用计数器, 以及循环收集器, 可以回收循环引用的垃圾内存.

**Q: 我要怎样创建一个共享库?**

A: 可以使用编译器的 `-produce dynamic` 选项, 或在 Gradle 中使用 `compilations.main.outputKinds 'DYNAMIC'` 设置, 即:

```
targets {
    fromPreset(presets.iosArm64, 'mylib') {
        compilations.main.outputKinds 'DYNAMIC'
    }
}
```

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. 参见 [samples/python\\_extension](#), 这是一个例子, 演示如何使用这样的共享库来连接 Python 程序和 Kotlin/Native 程序.

**Q: 我要怎样创建静态库, 或 object 文件?**

A: 可以使用编译器的 `-produce static` 选项, 或在 Gradle 中使用 `compilations.main.outputKinds 'STATIC'` 设置, 即:

```
targets {
    fromPreset(presets.iosArm64, 'mylib') {
        compilations.main.outputKinds 'STATIC'
    }
}
```

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.

**Q: 我要怎样在企业的网络代理服务器之后运行 Kotlin/Native?**

A: 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.

**Q: 我要怎样为我的 Kotlin 框架指定自定义的 Objective-C 前缀?**

A: 可以使用编译器的 `-module-name` 选项, 或对应的 Gradle DSL 语句, 即:

```
targets {
    fromPreset(presets.iosArm64, 'myapp') {
        compilations.main.outputKinds 'FRAMEWORK'
        compilations.main.extraOpts '-module-name', 'TheName'
    }
}
```

**Q: 我要怎样对我的 Kotlin 框架启用 bitcode?**

A: 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:

```
targets {
    fromPreset(presets.iosArm64, 'myapp') {
        compilations.main.outputKinds 'FRAMEWORK'
        compilations.main.embedBitcode BitcodeEmbeddingMode.BITCODE // 对 release 版二进制文件请使用这个命令
        // 对 debug 版二进制文件请使用 BitcodeEmbeddingMode.MARKER
    }
}
```

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`.

#### Q: 为什么我会遇到 `InvalidMutabilityException` 异常?

A: 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).

#### Q: 我要怎样让一个单子对象可以被修改?

A: 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.

#### Q: 我要怎样使用 Kotlin/Native 的 master 分支上的最新版本来编译我的项目?

A: 请使用以下任何一种方法:

- 对于命令行环境, 你可以使用 gradle 来编译, 详细方法请参见 [README](#) (如果你遇到了错误, 可以试试运行一下 `./gradlew clean`):
- 对于 Gradle 环境, 你可以使用 [Gradle 复合编译](#), 如下:

# 协程(Coroutine)

Kotlin 语言只在它的标准库中提供了最少量的低层 API, 让其它各种不同的库来使用协程. 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 `promis` 更安全, 更不容易出错的抽象模型.

`kotlinx.coroutines` 是 JetBrains 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](#).

## 章节目录

- [基本概念](#)
- [取消与超时](#)
- [挂起函数\(Suspending Function\)的组合](#)
- [协程上下文与派发器\(Dispatcher\)](#)
- [异常处理与监视](#)
- [通道\(Channel\) \(实验性功能\)](#)
- [共享的可变状态与并发](#)
- [选择表达式 \(实验性功能\)](#)

## 其他参考文档

- [使用协程进行 UI 编程向导](#)
- [使用协程进行响应式流\(Reactive Stream\)编程向导](#)
- [协程功能设计文档 \(KEEP\)](#)
- [kotlinx.coroutines API 完整参考](#)

## 目录

- [协程的基本概念](#)
  - [你的第一个协程](#)
  - [联通阻塞与非阻塞的世界](#)
  - [等待一个任务完成](#)
  - [结构化的并发](#)
  - [作用范围构建器](#)
  - [抽取函数\(Extract Function\)的重构](#)
  - [协程是非常轻量的](#)
  - [全局协程类似于守护线程\(Daemon Thread\)](#)

## 协程的基本概念

本章我们介绍协程的基本概念。

### 你的第一个协程

请运行以下代码:

```
import kotlin.coroutines.*

fun main() {
    GlobalScope.launch { // 在后台启动新的协程, 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!") // 等待完成后打印信息
    }
    println("Hello,") // 当协程在后台等待时, 主线程继续执行
    Thread.sleep(2000L) // 阻塞主线程 2 秒, 保证 JVM 继续存在
}
```

完整的代码请参见 [这里](#)

你将看到以下运行结果:

```
Hello,
World!
```

本质上来说, 协程就是轻量级的线程. 在某个 [CoroutineScope](#) 的上下文环境内, 协程通过 [协程构建器 launch](#) 来启动. 在上面的示例中, 我们在 [GlobalScope](#) 内启动了新的协程, 也就是说, 新协程的生命周期只受整个应用程序的生命周期限制。

你可以把 `GlobalScope.launch { ... }` 替换为 `thread { ... }`, 把 `delay(...)` 替换为 `Thread.sleep(...)`, 运行的结果仍然一样. 请自己试验一下.

如果只把 `GlobalScope.launch` 替换为 `thread`, 编译器会报告以下错误:

Error: Kotlin: Suspend functions are only allowed to be called from a coroutine or another suspend function

这是因为 [delay](#) 是一个特殊的 *挂起函数(suspending function)*, 它不会阻塞进程, 但会 *挂起* 协程, 因此它只能在协程中使用。

### 联通阻塞与非阻塞的世界

在我们的第一个示例程序中, 我们在同一块代码中混合使用了 *非阻塞* 的 `delay(...)` 函数和 *阻塞* 的 `Thread.sleep(...)` 函数. 因此很容易搞不清楚哪个函数是阻塞的, 哪个函数不是阻塞的. 所以我们通过 [runBlocking](#) 协程构建器来明确指定使用阻塞模式:

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // 在后台启动新的协程, 然后继续执行当前程序
        delay(1000L)
        println("World!")
    }
    println("Hello,") // 主线程在这里立即继续执行
    runBlocking { // 但这个表达式会阻塞主线程
        delay(2000L) // ... 我们在这里等待 2 秒, 保证 JVM 继续存在
    }
}
```

完整的代码请参见 [这里](#)

这样修改后的执行结果是一样的, 但这段代码只使用非阻塞的 `delay` 函数. 在主线程中, 调用 `runBlocking` 会阻塞, 直到 `runBlocking` 内部的协程执行完毕.

我们也可以用更符合 Kotlin 语言编程习惯的方式重写这个示例程序, 用 `runBlocking` 来包装主函数的运行:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> { // 启动主协程
    GlobalScope.launch { // 在后台启动新的协程, 然后继续执行当前程序
        delay(1000L)
        println("World!")
    }
    println("Hello,") // 主协程在这里立即继续执行
    delay(2000L) // 等待 2 秒, 保证 JVM 继续存在
}
```

完整的代码请参见 [这里](#)

这里 `runBlocking<Unit> { ... }` 起一种适配器的作用, 用来启动最上层的主协程. 我们明确指定了返回值类型为 `Unit`, 因为 Kotlin 语言中, 语法正确的 `main` 函数必须返回 `Unit`.

对挂起函数编写单元测试也可以使用这种方式:

```
class MyTest {
    @Test
    fun testMySuspendingFunction() = runBlocking<Unit> {
        // 在这里, 我们可以根据需要通过任何断言的方式来使用挂起函数
    }
}
```

## 等待一个任务完成

当其他协程正在工作时, 等待一段固定的时间, 这是一种不太好的方案. 下面我们(以非阻塞的方式)明确地等待我们启动的后台 `Job` 执行完毕:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    //sampleStart
    val job = GlobalScope.launch { // 启动新的协程, 并保存它的执行任务的引用
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    job.join() // 等待, 直到子协程执行完毕
    //sampleEnd
}
```

完整的代码请参见 [这里](#)

这样修改后, 执行结果仍然完全一样, 但主协程的代码不必尝试等待一个确定的, 比后台任务运行时间更长的时间. 这样就好多了.

## 结构化的并发

实际使用协程时, 我们还需要一些其他的東西. 当我们使用 `GlobalScope.launch` 时, 我们创建了一个顶级的协程. 虽然它是轻量的, 但它运行时还是会消耗一些内存资源. 如果我们忘记保存一个新启动的协程的引用, 协程仍然会运行. 假如协程中的代码挂起(比如, 我们错误地等待了一个很长的时间), 那么会怎么样, 如果我们启动了太多的协程, 耗尽了内存, 那么会怎么样? 不得不手工保存所有启动的协程的引用, 然后 [join](#) 所有这些协程, 这样的编程方式是很容易出错的.

我们有更好的解决方案. 我们可以在代码中使用结构化的并发. 我们可以在协程需要工作的那个特定的作用范围内启动协程, 而不是象我们通常操作线程那样(线程总是全局的), 在 [GlobalScope](#) 内启动协程.

在我们的示例程序中, 有一个 `main` 函数, 它使用 `runBlocking` 协程构建器变换成了一个协程. 所有的协程构建器, 包括 `runBlocking`, 都会向它的代码段的作用范围添加一个 `CoroutineScope` 的实例. 我们在这个作用范围内启动协程, 而不需要明确地 `join` 它们, 因为外层协程(在我们的示例程序中就是 `runBlocking`) 会等待它的作用范围内启动的所有协程全部完成. 因此, 我们可以把示例程序写得更简单一些:

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch { // 在 runBlocking 的作用范围内启动新的协程
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
```

完整的代码请参见 [这里](#)

## 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` 构建器来自行声明作用范围. 这个构建器可以创建新的协程作用范围, 并等待在这个范围内启动的所有子协程运行结束. `runBlocking` 和 `coroutineScope` 之间的主要区别是, `coroutineScope` 在等待子协程运行时, 不会阻塞当前线程.

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch {
        delay(200L)
        println("Task from runBlocking")
    }

    coroutineScope { // 创建新的协程作用范围
        launch {
            delay(500L)
            println("Task from nested launch")
        }

        delay(100L)
        println("Task from coroutine scope") // 在嵌套的 launch 之前, 这一行会打印
    }

    println("Coroutine scope is over") // 直到嵌套的 launch 运行结束后, 这一行才会打印
}
```

完整的代码请参见 [这里](#)

### 抽取函数(Extract Function)的重构

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, 我们使用了 `delay` 函数, 来 *挂起* 当前协程的运行.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch { doWorld() }
    println("Hello,")
}

// 这是你的第一个挂起函数
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

完整的代码请参见 [这里](#)

但是如果抽取出来的函数包含一个协程构建器, 并且这个构建器需要在当前作用范围上调用, 那么怎么办? 这种情况下, 对于被抽取出来的函数来说只有 `suspend` 修饰符是不够的. 有一种解决办法是把 `doWorld` 变成 `CoroutineScope` 的扩展函数, 但这种办法有时候并不适用, 因为它会使得 API 难于理解. 理想的解决办法是, 要么明确地把 `CoroutineScope` 作为一个类的域变量, 再让这个类包含我们抽取的函数, 或者让外层类实现 `CoroutineScope` 接口, 于是就可以隐含的实现这个目的. 最后一种办法就是, 可以使用 [CoroutineScope\(coroutineContext\)](#), 但这种方法从结构上来说并不安全, 因为你不再能够控制当前方法运行时所属的作用范围. 只有私有 API 才能够使用这个构建器.

### 协程是非常轻量的

请试着运行一下这段代码:



```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(100_000) { // 启动非常多的协程
        launch {
            delay(1000L)
            print(".")
        }
    }
}
```

完整的代码请参见 [这里](#)

这个例子会启动 10 万个协程, 1 秒钟之后, 每个协程打印 1 个点. 现在试试用线程来实现同样的功能. 会发生什么结果? (你的程序很可能会发生某种内存耗尽的错误)

### 全局协程类似于守护线程(Daemon Thread)

下面的示例程序会在 [GlobalScope](#) 作用范围内启动一个长期运行的协程, 协程会每秒打印 “I’ m sleeping” 2次, 主程序等待一段时间后, 从 main 函数返回:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    //sampleStart
    GlobalScope.launch {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(1300L) // 等待一段时间后, 主程序直接退出
    //sampleEnd
}
```

完整的代码请参见 [这里](#)

你可以试着运行这段程序, 看到它会打印 3 行消息, 然后就结束了:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
```

在 [GlobalScope](#) 作用范围内启动的活跃的协程, 不会保持应用程序的整个进程存活. 它们的行为就象守护线程一样.

## 目录

- [取消与超时](#)
  - [取消协程的运行](#)
  - [取消是协作式的](#)
  - [使计算代码能够被取消](#)
  - [使用 finally 语句来关闭资源](#)
  - [运行无法取消的代码段](#)
  - [超时](#)

## 取消与超时

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](#) 函数会返回一个 [Job](#), 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("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](#)

这个示例的运行结果如下:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
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, 因为协程已经被取消了. 还有一个 [Job](#) 上的扩展函数 [cancelAndJoin](#), 它组合了 [cancel](#) 和 [join](#) 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](#) 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
            // 每秒打印信息 2 次
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("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](#)

运行一下这个示例, 我们会看到, 即使在取消之后, 协程还是继续打印 “I’ m sleeping” 信息, 直到循环 5 次之后, 协程才自己结束。

### 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](#) 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
            // 每秒打印信息 2 次
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("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](#)

你会看到, 现在循环变得能够被取消了. [isActive](#) 是一个扩展属性, 在协程内部的代码中可以通过 [CoroutineScope](#) 对象访问到.

### 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](#) 异常, 这个异常可以通过通常方式来处理. 比如, 可以使用 `try {...} finally {...}` 表达式, 或者 Kotlin 的 `use` 函数, 以便协程被取消时来执行结束处理:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    //sampleStart
    val job = launch {
        try {
            repeat(1000) { i ->
                println("I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("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](#)

[join](#) 和 [cancelAndJoin](#) 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:

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
main: I'm tired of waiting!
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](#) 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](#) 函数和 [NonCancellable](#) 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("I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            withContext(NonCancellable) {
                println("I'm running finally")
                delay(1000L)
                println("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](#)

## 超时

在实际应用中, 取消一个协程最明显的理由就是, 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](#), 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](#) 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](#)

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](#) 函数抛出的 `TimeoutCancellationException` 异常是 [CancellationException](#) 的子类. 我们在前面的例子中, 都没有看到过 [CancellationException](#) 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 `try {...} catch (e: TimeoutCancellationException) {...}` 代码块中, 也可以使用 [withTimeoutOrNull](#) 函数, 它与 [withTimeout](#) 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](#)

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
```

## 目录

- [通道\(Channel\) \(实验性功能\)](#)
  - [通道的基本概念](#)
  - [通道的关闭与迭代](#)
  - [构建通道的生产者\(Producer\)](#)
  - [管道\(Pipeline\)](#)
  - [使用管道寻找质数](#)
  - [扇出\(Fan-out\)](#)
  - [扇入\(Fan-in\)](#)
  - [带缓冲区的通道](#)
  - [通道是平等的](#)
  - [定时器\(Ticker\)通道](#)

## 通道(Channel) (实验性功能)

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.

通道是 `kotlinx.coroutines` 中的一个实验性功能. 在以后的 `kotlinx.coroutines` 新版本库中, 与通道相关的 API 将会发生变化, 可能带来一些不兼容的变更.

### 通道的基本概念

[Channel](#) 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](#) 操作, 没有阻塞的 `take` 操作, 而是提供挂起的 [receive](#) 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)
    }
    // 我们在这里打印收到的整数:
    repeat(5) { println(channel.receive()) }
    println("Done!")
    //sampleEnd
}
```

完整的代码请参见 [这里](#)

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` 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` 循环来打印接收到的数据 (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](#)

### 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`, 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`, 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](#)

### 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
    for (i in 1..5) println(squares.receive()) // 打印前 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](#)

所有创建协程的函数都被定义为 [CoroutineScope](#) 上的扩展函数, 因此我们可以依靠 [结构化的并发](#) 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) ...

下面的示例程序会打印前 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](#) 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. 我们可以在打印完前 10 个质数之后, 使用 [cancelChildren](#) 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)
    for (i in 1..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](#)

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 [buildIterator](#) 来创建相同的管道. 把 `produce` 函数替换为 `buildIterator`, 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](#) 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 `buildSequence` / `buildIterator` 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打印自己的 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](#)

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](#)

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()` 工厂函数和 `produce` 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, 先打印它
            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](#)

使用缓冲区大小为 4 的通道时, 这个示例程序会打印 “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](#)

“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](#).

### 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](#) 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](#) 中, 执行 “on tick” 动作.

可以使用 [ticker](#) 工厂函数来创建这种通道. 使用通道的 [ReceiveChannel.cancel](#) 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.*

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") // 初始指定的延迟时间还未过去

    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
}

```

完整的代码请参见 [这里](#)

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](#) 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](#), 可以指定产生元素时维持一个固定的间隔速度.

## 目录

- [挂起函数\(Suspending Function\)的组合](#)
  - [默认连续执行](#)
  - [使用 async 并发执行](#)
  - [延迟启动的\(Lazily started\) async](#)
  - [async 风格的函数](#)
  - [使用 async 的结构化并发](#)

## 挂起函数(Suspending Function)的组合

本章介绍将挂起函数组合起来的几种不同方式。

### 默认连续执行

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](#)

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, 以便更快地得到结果, 那么应该怎么做? 这时 `async` 可以帮助我们.

概念上来说, `async` 就好象 `launch` 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`, 其中不带有结果值, 而 `async` 返回一个 `Deferred` - 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](#)

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, 因为我们让两个协程并发执行. 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`, 可以让 `async` 延迟启动. 只有在通过 `await` 访问协程的计算结果的时候, 或者调用 `start` 函数的时候, 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](#)

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` 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`, 而省略了对各个协程的 `start` 调用, 那么两个协程的执行结果将会是连续的, 而不是并行的, 因为 `await` 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

我们可以定义一个 `async` 风格的函数, 它使用一个明确的 `GlobalScope` 引用, 通过 `async` 协程构建器来 *异步地* 调用 `doSomethingUsefulOne` 和 `doSomethingUsefulTwo`. 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.

```

// somethingUsefulOneAsync 函数的返回值类型是 Deferred<Int>
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

// somethingUsefulTwoAsync 函数的返回值类型是 Deferred<Int>
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 {
        // 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

fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

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](#)

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.

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` 协程构建器被定义为 `CoroutineScope` 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`, `coroutineScope` 函数可以为我们提供 `CoroutineScope`:

```
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.*

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](#)

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:

```

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](#)

注意, 当一个子协程失败时, 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

```

## 目录

- [协程上下文与派发器\(Dispatcher\)](#)
  - [派发器与线程](#)
  - [非受限派发器\(Unconfined dispatcher\)与受限派发器\(Confined dispatcher\)](#)
  - [协程与线程的调试](#)
  - [在线程间跳转](#)
  - [在上下文中的任务](#)
  - [协程的子协程](#)
  - [父协程的职责](#)
  - [为协程命名以便于调试](#)
  - [组合上下文中的元素](#)
  - [使用显式任务来取消协程](#)
  - [线程的局部数据](#)

## 协程上下文与派发器(Dispatcher)

协程执行时总是属于某个上下文环境, 上下文环境通过 [CoroutineContext](#) 类型的值来表示, 这个类型是 Kotlin 标准库中定义的。

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](#), 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](#)), 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](#) 和 [async](#), 都接受一个可选的 [CoroutineContext](#) 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](#)

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](#) 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](#) 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.

当协程在 [GlobalScope](#) 内启动时, 会使用默认派发器, 用 [Dispatchers.Default](#) 表示, 它会使用后台共享的线程池, 因此 `launch(Dispatchers.Default) { ... }` 会使用与 `GlobalScope.launch { ... }` 相同的派发器.

[newSingleThreadContext](#) 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, 当这样的线程, 必须在不再需要的时候使用 [close](#) 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](#) 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, 会继承外层 [CoroutineScope](#) 的派发器. 具体来说, 对于 `runBlocking` 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](#)

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](#) 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. 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. 调试多线程应用程序的常见办法是在日志文件的每一条日志信息中打印线程名称. 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](#)

上面的例子中会出现 3 个协程. 主协程 (#1) - `runBlocking` 协程, 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` 函数会在方括号内打印线程名称, 你可以看到, 是 `main` 线程, 但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.

关于调试工具的详情, 请参见 [newCoroutineContext](#) 函数的文档.

## 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](#)

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](#), 另一个技巧是使用 [withContext](#) 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](#) 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](#) 是协程上下文的一部分。协程可以通过自己的上下文来访问到 [Job](#), 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](#)

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](#) 中的 [isActive](#) 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](#) 内启动时, 它会通过 [CoroutineScope.coroutineContext](#) 继承这个协程的上下文, 并且新协程的 [Job](#) 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, 如果使用 [GlobalScope](#) 来启动一个协程, 那么这个协程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, 其中一个使用 GlobalScope
        GlobalScope.launch {
            println("job1: I run in GlobalScope 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
    delay(1000) // 延迟 1 秒, 看看结果如何
    println("main: Who has survived request cancellation?")
    //sampleEnd
}

```

完整的代码请参见 [这里](#)

这个示例程序的运行结果是:

```

job1: I run in GlobalScope and execute independently!
job2: I am a child of the request coroutine
job1: I am not affected by cancellation of the request
main: Who has survived request cancellation?

```

## 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](#) 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](#)

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](#) 起到与线程名类似的作用. 当 [调试模式](#) 开启时, 协程名称会出现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](#)

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](#)

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
```

### 使用显式任务来取消协程

下面我们把关于上下文, 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.

我们创建 [Job](#) 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. 当 activity 创建时, 使用工厂函数 [Job\(\)](#) 来创建任务的实例, 当 activity 销毁时取消这个任务, 如下例所示:

```
class Activity : CoroutineScope {
    lateinit var job: Job

    fun create() {
        job = Job()
    }

    fun destroy() {
        job.cancel()
    }
    // 待续 ...
}
```

我们还在这个 `Activity` 类中实现 [CoroutineScope](#) 接口. 我们只需要实现这个接口的 [CoroutineScope.coroutineContext](#) 属性, 来指明在这个作用范围内启动的协程所使用的上下文. 我们将希望使用的派发器(这个例子中使用 [Dispatchers.Default](#))与任务结合在一起:

```
// Activity 类的内容继续
override val coroutineContext: CoroutineContext
    get() = Dispatchers.Default + job
// 待续 ...
```

然后, 在这个 `Activity` 的作用范围内启动协程, 不必明确指定上下文. 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 ->
        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 会取消所有的协程, 如果我们继续等待, 就会注意到协程不再向屏幕打印信息, 因此我们知道协程已经被取消了:

```

import kotlin.coroutines.*
import kotlinx.coroutines.*

class Activity : CoroutineScope {
    lateinit var job: Job

    fun create() {
        job = Job()
    }

    fun destroy() {
        job.cancel()
    }
    // 待续 ...

    // Activity 类的内容继续
    override val coroutineContext: CoroutineContext
        get() = Dispatchers.Default + job
    // 待续 ...

    // 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 ->
            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.create() // 创建一个 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](#)

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 个协程打印出了信息, 由于 `Activity.destroy()` 中调用了 `job.cancel()`, 其他所有协程都被取消了。

### 线程的局部数据

有些时候, 如果能够传递一些线程局部的数据(thread-local data)将是一种很方便的功能, 但是对于协程来说, 它并没有关联到某个具体的线程, 因此, 不写大量的样板代码, 通常很难自己写代码来实现这个功能。

对于 [ThreadLocal](#), 有一个扩展函数 [asContextElement](#) 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](#)

在这个示例程序中, 我们使用 [Dispatchers.Default](#), 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'
```

`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。它只有一个关键的限制: 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](#) 来更新线程局部变量的值, 详情请参见 [asContextElement](#)。

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](#) 接口, 详情请参见这个接口的文档.

## 目录

- [异常处理](#)
  - [异常的传播\(propagation\)](#)
  - [CoroutineExceptionHandler](#)
  - [取消与异常](#)
  - [异常的聚合\(aggregation\)](#)
- [监控](#)
  - [监控任务](#)
  - [监控作用范围](#)
  - [被监控的协程中的异常](#)

## 异常处理

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](#), 而协程机制忽略会这个异常. 但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, 那么会怎么样?

### 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](#) 和 [actor](#) 构建器), 或者将异常交给使用者处理([async](#) 和 [produce](#) 构建器). 前一种方式部队异常进行处理, 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](#) 或 [receive](#) 来处理异常. (关于 [produce](#) 和 [receive](#) 请参见 [通道\(Channel\)](#)).

我们通过一个简单的示例程序来演示一下, 我们在 [GlobalScope](#) 内创建协程:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = GlobalScope.launch {
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException() // 这个异常会被 Thread.defaultUncaughtExceptionHandler 打印到控制台
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async {
        println("Throwing exception from async")
        throw ArithmeticException() // 这个异常不会被打印, 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")
    }
}
```

完整的代码请参见 [这里](#)

(使用 [调试模式](#)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

但是如果我們不想把所有的異常都輸出到控制台, 那麼應該怎麼辦呢? 協程的上下文元素 [CoroutineExceptionHandler](#) 會被作為協程的通用的 `catch` 塊, 我們可以在這裡實現自定義的日誌輸出, 或其他異常處理邏輯. 它的使用方法与 [Thread.uncaughtExceptionHandler](#) 类似.

在 JVM 平台, 可以通过 [ServiceLoader](#) 注册一个 [CoroutineExceptionHandler](#), 为所有的协程重定义全局的异常处理器. 全局的异常处理器类似于 [Thread.defaultUncaughtExceptionHandler](#), 如果没有注册更具体的异常处理器, 就会使用这个

`Thread.defaultUncaughtExceptionHandler` 异常处理器. 在 Android 平台, 默认安装的协程全局异常处理器是 `uncaughtExceptionHandler`.

有些异常是我们预计会被使用者处理的, 只有发生了这类异常以外的其他异常时, 才会调用 [CoroutineExceptionHandler](#), 因此, 对 `async` 或其他类似的协程构建器注册异常处理器, 不会产生任何效果.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    //sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception")
    }
    val job = GlobalScope.launch(handler) {
        throw AssertionError()
    }
    val deferred = GlobalScope.async(handler) {
        throw ArithmeticException() // 这个异常不会被打印, 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](#)

这个示例程序的输出结果是:

```
Caught 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](#) 来取消一个协程, 而且不指明任何原因, 那么协程会终止运行, 但不会取消它的父协程. 父协程可以使用不指明原因的取消机制, 来取消自己的子协程, 而不取消自己.

```
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](#)

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](#) 中的稳定的协程层级关系, 而不是依赖于 [CoroutineExceptionHandler](#) 的实现. 当所有的子协程全部结束后, 原始的异常会被父协程处理.

这也是为什么, 在这些示例程序中, 我们总是在 [GlobalScope](#) 内创建的协程上安装 [CoroutineExceptionHandler](#). 如果在 `main` [runBlocking](#) 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后之后, 主协程一定会被取消, 而忽略它上面安装的异常处理器.

```

import kotlinx.coroutines.*

fun main() = runBlocking {
    //sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $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](#)

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
Caught java.lang.ArithmeticException

```

### 异常的聚合(agggregation)

如果一个协程的多个子协程都抛出了异常, 那么会怎么样? 通常的规则是 “最先发生的异常优先”, 因此第 1 个发生的异常会被传递给异常处理器. 但是这种处理发生可能会导致丢失其他异常, 比如, 如果另一个协程在它的 `finally` 块中抛出了另一个异常. 为了解决这个问题, 我们将其他异常压制(suppress)到最先发生的异常内.

有一种解决办法是, 我们可以将各个异常分别向外抛出, 但是这样一来 [Deferred.await](#) 部分就必须实现相同的机制, 捕获多个异常, 以保持异常抛出与捕获的一致性. 这就会导致协程内部的具体实现细节(比如, 它是否将部分工作代理给了自己的子协程)暴露给了它的异常处理器.

```

import kotlinx.coroutines.*
import java.io.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception with suppressed ${exception.suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
                delay(Long.MAX_VALUE)
            } finally {
                throw ArithmeticException()
            }
        }
        launch {
            delay(100)
            throw IOException()
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}

```

完整的代码请参见 [这里](#)

注意: 上面的示例程序只能在支持 suppressed 异常的 JDK7+ 以上版本才能正常运行

这个示例程序的输出结果是:

```
Caught 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.*

fun main() = runBlocking {
    //sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught original $exception")
    }
    val job = GlobalScope.launch(handler) {
        val inner = launch {
            launch {
                launch {
                    throw IOException()
                }
            }
        }
    }
    try {
        inner.join()
    } catch (e: CancellationException) {
        println("Rethrowing CancellationException with original cause")
        throw e
    }
}
job.join()
//sampleEnd
}

```

完整的代码请参见 [这里](#)

这个示例程序的输出结果是:

```

Rethrowing CancellationException with original cause
Caught original 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. 但是如果我们需要单向的取消, 那么应该怎么办呢?

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几个子任务, 需要 *监控* 这些子任务的执行, 追踪它们是否失败, 并且重启那些失败的子任务.

## 监控任务

为了这类目的, 我们可以使用 [SupervisorJob](#). 它与通常的 [Job](#) 类似, 唯一的区别在于取消只向下方传播. 我们用一个示例程序来演示一下:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    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("First child is failing")
            throw AssertionError("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("First child is cancelled: ${firstChild.isCancelled}, but 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("Second child is cancelled because supervisor is cancelled")
            }
        }
        // 等待第 1 个子协程失败, 并结束运行
        firstChild.join()
        println("Cancelling supervisor")
        supervisor.cancel()
        secondChild.join()
    }
}
```

完整的代码请参见 [这里](#)

这个示例程序的输出结果是:

```
First child is failing
First child is cancelled: true, but second one is still active
Cancelling supervisor
Second child is cancelled because supervisor is cancelled
```

### 监控作用范围

对于 **带作用范围**的并发, 可以使用 [supervisorScope](#) 代替 [coroutineScope](#) 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. 它和 [coroutineScope](#) 一样, 在运行结束之前也会等待所有的子协程结束.

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("Child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("Child is cancelled")
                }
            }
            // 使用 yield, 给子协程一个机会运行, 并打印信息
            yield()
            println("Throwing exception from scope")
            throw AssertionError()
        }
    } catch (e: AssertionError) {
        println("Caught assertion error")
    }
}
```

完整的代码请参见 [这里](#)

这个示例程序的输出结果是:

```
Child is sleeping
Throwing exception from scope
Child is cancelled
Caught 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.

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("Child throws an exception")
            throw AssertionError()
        }
        println("Scope is completing")
    }
    println("Scope is completed")
}
```

完整的代码请参见 [这里](#)

这个示例程序的输出结果是:

Scope is completing  
Child throws an exception  
Caught java.lang.AssertionError  
Scope is completed



## 目录

- [选择表达式\(Select expression\) \(实验性功能\)](#)
  - [从通道中选择](#)
  - [在通道关闭时选择](#)
  - [发送时选择](#)
  - [选择延迟的值](#)
  - [在延迟值的通道上切换](#)

## 选择表达式(Select expression) (实验性功能)

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 300ms 产生一个 “Fizz” 字符串:

```
fun CoroutineScope.fizz() = produce<String> {  
    while (true) { // 每 300ms 发送一个 "Fizz"  
        delay(300)  
        send("Fizz")  
    }  
}
```

`buzz` 每 500ms 产生一个 “Buzz!” 字符串:

```
fun CoroutineScope.buzz() = produce<String> {  
    while (true) { // 每 500ms 发生一个 "Buzz!"  
        delay(500)  
        send("Buzz!")  
    }  
}
```

使用 [receive](#) 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](#) 表达式的 [onReceive](#) 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 300ms 发送一个 "Fizz"
        delay(300)
        send("Fizz")
    }
}

fun CoroutineScope.buzz() = produce<String> {
    while (true) { // 每 500ms 发生一个 "Buzz!"
        delay(500)
        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](#)

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 -> 'Buzz!'

```

### 在通道关闭时选择

如果通道已关闭, 那么 `select` 表达式的 `onReceive` 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` 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.onReceiveOrNull { value ->
            if (value == null)
                "Channel 'a' is closed"
            else
                "a -> '$value'"
        }
        b.onReceiveOrNull { value ->
            if (value == null)
                "Channel 'b' is closed"
            else
                "b -> '$value'"
        }
    }
}
```

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.onReceiveOrNull { value ->
            if (value == null)
                "Channel 'a' is closed"
            else
                "a -> '$value'"
        }
        b.onReceiveOrNull { value ->
            if (value == null)
                "Channel 'b' is closed"
            else
                "b -> '$value'"
        }
    }
}

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](#)

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集件事。

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` 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 `onReceiveOrNull` 子句.

### 发送时选择

选择表达式也可以使用 `onSend` 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](#)

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](#) 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](#)

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 `onReceiveOrNull` 子句和 `onAwait` 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.onReceiveOrNull { update ->
                update // 改为等待下一个延迟值
            }
            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.receiveOrNull() // 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打印 `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.onReceiveOrNull { update ->
                update // 改为等待下一个延迟值
            }
            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.receiveOrNull() // 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 { // 启动打印结果的协程
        for (s in switchMapDeferreds(chan))
            println(s) // 打印每个收到的字符串
    }
    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](#)

这个示例程序的运行结果是:

```

BEGIN
Replace
END
Channel was closed

```



## 目录

- [共享的可变状态值与并发](#)
  - [问题的产生](#)
  - [volatile 不能解决这个问题](#)
  - [线程安全的数据结构](#)
  - [细粒度的线程限定](#)
  - [粗粒度的线程限定](#)
  - [互斥](#)
  - [Actor](#)

## 共享的可变状态值与并发

使用多线程的派发器, 比如 [Dispatchers.Default](#), 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 CoroutineScope.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 {  
        val jobs = List(n) {  
            launch {  
                repeat(k) { action() }  
            }  
        }  
        jobs.forEach { it.join() }  
    }  
    println("Completed ${n * k} actions in $time ms")  
}
```

我们先来执行一个非常简单的操作, 把一个共享的可变变量加 1, 在 [GlobalScope](#) 中启动协程, 使用多线程的 [Dispatchers.Default](#).

```

import kotlinx.coroutines.*
import kotlin.system.*

suspend fun CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

var counter = 0

fun main() = runBlocking<Unit> {
    //sampleStart
    GlobalScope.massiveRun {
        counter++
    }
    println("Counter = $counter")
    //sampleEnd
}

```

完整的代码请参见 [这里](#)

这段代码的最终输出结果是什么? 很大概率它不会输出 “Counter = 100000”, 因为 100 个协程运行在多线程环境内, 它们同时给 counter 加 1, 但没有任何的同步机制.

注意: 如果你的系统太旧, 只有 2 个或更少的 CPU, 那么你 一定会 得到确定的结果 100000, 因为这时线程池只有一个线程. 为了重现我们的问题, 你需要对示例程序做如下修改:

```

import kotlinx.coroutines.*
import kotlin.system.*

suspend fun CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

val mtContext = newFixedThreadPoolContext(2, "mtPool") // 明确定义上下文, 使用 2 个线程
var counter = 0

fun main() = runBlocking<Unit> {
    //sampleStart
    CoroutineScope(mtContext).massiveRun { // 使用自定义的上下文, 而不是 Dispatchers.Default
        counter++
    }
    println("Counter = $counter")
    //sampleEnd
}

```

完整的代码请参见 [这里](#)

### 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:

```

import kotlinx.coroutines.*
import kotlin.system.*

suspend fun CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

@Volatile // 在 Kotlin 中, `volatile` 是注解
var counter = 0

fun main() = runBlocking<Unit> {
    GlobalScope.massiveRun {
        counter++
    }
    println("Counter = $counter")
}

```

完整的代码请参见 [这里](#)

代码运行变慢了,但最终我们还是得不到 “Counter = 100000”, 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需要的同步保障. 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 CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

var counter = AtomicInteger()

fun main() = runBlocking<Unit> {
    //sampleStart
    GlobalScope.massiveRun {
        counter.incrementAndGet()
    }
    println("Counter = ${counter.get()}")
    //sampleEnd
}

```

完整的代码请参见 [这里](#)

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的一个事件派发(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 CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking<Unit> {
    //sampleStart
    GlobalScope.massiveRun { // 使用 DefaultDispathcer 运行每个协程
        withContext(counterContext) { // 但把所有的加 1 操作都限定在单一线程的上下文中
            counter++
        }
    }
    println("Counter = $counter")
    //sampleEnd
}

```

完整的代码请参见 [这里](#)

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](#), 从多线程的 [Dispatchers.Default](#) 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. 这里我们使用 [CoroutineScope\(\)](#) 函数来将协程的上下文转换为 [CoroutineScope](#) 类型:

```

import kotlinx.coroutines.*
import kotlin.system.*

suspend fun CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking<Unit> {
    //sampleStart
    CoroutineScope(counterContext).massiveRun { // 在单一线程的上下文中运行每个协程
        counter++
    }
    println("Counter = $counter")
    //sampleEnd
}

```

完整的代码请参见 [这里](#)

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](#). 它的 [lock](#) 和 [unlock](#) 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](#), 它代表 `mutex.lock(); try { ... } finally { mutex.unlock() }` :

```

import kotlinx.coroutines.*
import kotlinx.coroutines.sync.*
import kotlin.system.*

suspend fun CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

val mutex = Mutex()
var counter = 0

fun main() = runBlocking<Unit> {
    //sampleStart
    GlobalScope.massiveRun {
        mutex.withLock {
            counter++
        }
    }
    println("Counter = $counter")
    //sampleEnd
}

```

完整的代码请参见 [这里](#)

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.

## Actor

[actor](#) 是一个实体, 其中包含一个线程, 一个限定并封装在这个线程上的状态值, 以及一个用来与其他协程通信的通道. 一个简单的 actor 可以写成一个函数, 但带有复杂状态的 actor 更适合写成一个类.

有一个 [actor](#) 协程构建器, 可以将 actor 的信箱通道(mailbox channel) 绑定在它的作用范围上, 用来接受消息, 并将它的送信通道绑定到线程构建器的结果任务对象上, 因此一个指向 actor 的引用就可以作为它的句柄来传递.

使用 actor 的第一步是定义 actor 将要处理的消息类. Kotlin 的 [封闭类\(Sealed Class\)](#) 非常适合于这个目的. 我们定义一个 CounterMsg 封闭类, 其中 IncCounter 消息用来对计数器加 1, GetCounter 消息用来获取计数器的值. 后一个消息还需要发送一个应答. 我们在这里使用通信原语 [CompletableDeferred](#) 来实现这个目的, 它表示一个会在未来得到(传来)的单个的值,.

```

// 供 counterActor 使用的消息类
sealed class CounterMsg
object IncCounter : CounterMsg() // 单向消息, 将计数器加 1
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg() // 一个带应答的请求

```

然后我们定义一个函数, 它使用 [actor](#) 协程构建器启动一个 actor:



```
// 这个函数启动一个新的计数器 actor
fun CoroutineScope.counterActor() = actor<CounterMsg> {
    var counter = 0 // actor 的状态值
    for (msg in channel) { // 遍历所有收到的消息
        when (msg) {
            is IncCounter -> counter++
            is GetCounter -> msg.response.complete(counter)
        }
    }
}
```

主代码非常简单:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlin.system.*

suspend fun CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

// 供 counterActor 使用的消息类
sealed class CounterMsg
object IncCounter : CounterMsg() // 单向消息, 将计数器加 1
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg() // 一个带应答的请求

// 这个函数启动一个新的计数器 actor
fun CoroutineScope.counterActor() = actor<CounterMsg> {
    var counter = 0 // actor 的状态值
    for (msg in channel) { // 遍历所有收到的消息
        when (msg) {
            is IncCounter -> counter++
            is GetCounter -> msg.response.complete(counter)
        }
    }
}

fun main() = runBlocking<Unit> {
    //sampleStart
    val counter = counterActor() // 创建 actor
    GlobalScope.massiveRun {
        counter.send(IncCounter)
    }
    // 发送一个消息, 从 actor 得到计数器值
    val response = CompletableDeferred<Int>()
    counter.send(GetCounter(response))
    println("Counter = ${response.await()}")
    counter.close() // 关闭 actor
    //sampleEnd
}
```

完整的代码请参见 [这里](#)

(对于结果的正确性来说) actor 本身在哪个上下文中运行是无关紧要的. actor 是一个协程, 协程是顺序执行的, 因此将状态值限定在特定的协程内, 可以解决共享的可变状态值问题. actor 确实可以修改它自己的私有状态值, 但它们之间只能通过消息来相互影响 (因此不必使用锁).

在高负载的情况下, actor 比锁的效率更高, 因为这种情况下 actor 总是有工作可做(不必挂起), 而且它完全不必在不同的上下文之间切换.

注意, [actor](#) 协程构建器 与 [produce](#) 协程构建器刚好相反. actor 绑定到一个通道, 从通道读取消息, 而生产者则绑定到一个通道, 向通道发送数据.

# 工具

## 为 Kotlin 代码编写文档

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).

### 生成文档

Kotlin 的文档生成工具叫做 [Dokka](#). 关于它的使用方法请阅读 [Dokka README](#).

Dokka 有 plugin 可用于 Gradle, Maven 以及 Ant 构建环境, 因此你可以将 Kotlin 代码的文档生成集成到你的构建过程之内.

### 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(checkedException), 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](#) 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, 只需要简单地将其名称放在方括号内:

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.

## 模块与包的文档

针对模块整体的文档, 以及针对模块内包的文档, 通过单独的 Markdown 文件的形式提供, 文件路径需要传递给 Dokka, 可以使用命令行参数 `-include` 来指定, 或者使用 Ant, Maven 以及 Gradle plugin 中的对应参数来指定.

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 <模块名>”, 对于包, 必须是 “Package <包的完整限定名>”.

下面是一个 Markdown 文件内容的示例:

```
# Module kotlin-demo
```

本模块演示 Dokka 语法的使用方法.

```
# Package org.jetbrains.kotlin.demo
```

包含各种有用的东西.

```
## 2 级标题
```

这个标题之后的文字也是 ``org.jetbrains.kotlin.demo`` 包的文档的一部分

```
# Package org.jetbrains.kotlin.demo2
```

另一个包中的有用的东西.

## 在 Kotlin 中处理注解

Kotlin 使用 *kapt* 编译器插件来支持注解处理器(参见 [JSR 269](#)). 注: *kapt* 是 “Kotlin annotation processing tool” 的缩写

大略地说, 你可以在你的 Kotlin 项目中使用 [Dagger](#) 或 [Data Binding](#) 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

应用 `kotlin-kapt` Gradle plugin:

```
plugins {  
    id "org.jetbrains.kotlin.kapt" version "1.3.21"  
}
```

```
plugins {  
    kotlin("kapt") version "1.3.21"  
}
```

你也可以使用 `apply plugin` 语法:

```
apply plugin: 'kotlin-kapt'
```

然后在你的 `dependencies` 块中使用 `kapt` 配置来添加对应的依赖:

```
dependencies {  
    kapt 'groupId:artifactId:version'  
}
```

```
dependencies {  
    kapt("groupId:artifactId:version")  
}
```

如果你以前对注解处理器使用过 [Android support](#), 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.

### 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 (从 1.2.20 版开始支持)

`kapt` 注解处理任务默认情况下不会 [被 Gradle 缓存](#). 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 = true
}
```

## 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` 文件中添加一个额外的标记, 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>
  </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>
```

在 [Kotlin 示例程序库](#) 中, 你可以找到一个完整的示例项目, 演示如何使用 Kotlin, Maven 和 Dagger.

请注意, IntelliJ IDEA 自有的编译系统目前还不支持 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.

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.

你可以在 [kotlin 示例](#) 的 Github 代码库中找到完整的示例.

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())  
}
```

## 使用 Gradle

要使用 Gradle 编译 Kotlin 代码, 你需要 [设置 `kotlin-gradle plugin`](#), 将它 [应用](#) 到你的工程, 然后 [添加 `kotlin-stdlib` 依赖](#). 在 IntelliJ IDEA 中, 在 Project action 内选择 Tools | Kotlin | Configure Kotlin 也可以自动完成这些操作.

### Plugin 与版本

要应用 Kotlin plugin, 可以使用 [the Gradle plugins DSL](#), 请将下面例子中的占位符替换为某个 plugin 名称, 具体的 plugin 名称请参照本章的后续小节:

```
plugins {  
    id 'org.jetbrains.kotlin.<...>' version '1.3.21'  
}
```

```
plugins {  
    kotlin("<...>") version "1.3.21"  
}
```

应用 Kotlin plugin 的另一种方法是, 在编译脚本的 classpath 中添加 `kotlin-gradle-plugin` 的依赖项目:

```
buildscript {  
    repositories {  
        mavenCentral()  
    }  
  
    dependencies {  
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.3.21"  
    }  
}  
  
plugins {  
    id "org.jetbrains.kotlin.<...>" version "1.3.21"  
}
```

```
buildscript {  
    repositories {  
        mavenCentral()  
    }  
  
    dependencies {  
        classpath(kotlin("gradle-plugin", version = "1.3.21"))  
    }  
}  
plugins {  
    kotlin("<...>")  
}
```

通过 [Gradle plugins DSL](#), 或 [Gradle Kotlin DSL](#), 使用 Kotlin Gradle plugin 1.1.1 及以上版本时, 不需要以上定义.

### 编译 Kotlin 多平台项目

关于使用 `kotlin-multiplatform` plugin 来编译 [跨平台项目](#) 的方法, 请参见 [使用 Gradle 编译跨平台项目](#).

### 编译到 JVM 平台

要编译到 JVM 平台, 需要应用(apply) Kotlin JVM plugin. 从 Kotlin 1.1.1 版开始, 可以使用 [Gradle plugins DSL](#) 来应用这个 plugin:

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.3.21"
}
```

```
plugins {
    kotlin("jvm") version "1.3.21"
}
```

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.

另一种方法是, 也可以使用旧的 `apply plugin` 模式:

```
apply plugin: 'kotlin'
```

在 Gradle Kotlin DSL 中, 不推荐使用 `apply` 来应用 Kotlin plugin. 详情请参见 [下文](#).

Kotlin 源代码可以与 Java 源代码共存在同一个文件夹下, 也可以放在不同的文件夹下. 默认的约定是使用不同的文件夹:

```
project
- src
  - main (root)
  - kotlin
  - java
```

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:

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

```
sourceSets["main"].java.srcDir("src/main/myJava")
sourceSets["main"].withConvention(KotlinSourceSet::class) {
    kotlin.srcDir("src/main/myKotlin")
}
```

如果使用 Gradle Kotlin DSL, 请用 `java.sourceSets { ... }` 来设置源代码集.

## 编译到 JavaScript

编译到 JavaScript 时, 需要应用(apply)另一个 plugin:

```
plugins {
    id 'kotlin2js' version '1.3.21'
}
```

```
plugins {
    id("kotlin2js") version "1.3.21"
}
```

注意, 使用这种方式来应用 Kotlin/JS plugin, 需要向 Gradle 设定文件( `settings.gradle` )添加以下代码:

```

pluginManagement {
    resolutionStrategy {
        eachPlugin {
            if (requested.id.id == "kotlin2js") {
                useModule("org.jetbrains.kotlin:kotlin-gradle-plugin:${requested.version}")
            }
        }
    }
}

```

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). 与编译到 JVM 平台时一样, 如果不使用默认约定的文件夹结构, 你应该使用 `sourceSets` 来指定文件夹目录:

```

sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
}

```

```

sourceSets["main"].withConvention(KotlinSourceSet::class) {
    kotlin.srcDir("src/main/myKotlin")
}

```

除了编译输出的 JavaScript 文件之外, plugin 默认还会创建一个带二进制描述符(binary descriptor)的 JS 文件. 如果你在编译一个被其他 Kotlin 模块依赖的可重用的库, 那么这个文件是必须的, 而且需要与编译结果一起发布. 这个文件的生成, 可以通过 `kotlinOptions.metaInfo` 选项来控制:

```

compileKotlin2Js {
    kotlinOptions.metaInfo = true
}

```

```

tasks {
    "compileKotlin2Js"(Kotlin2JsCompile::class) {
        kotlinOptions.metaInfo = true
    }
}

```

## 编译到 Android

Android 的 Gradle 模型与通常的 Gradle 略有区别, 因此如果我们想要编译一个使用 Kotlin 语言开发的 Android 工程, 就需要使用 `kotlin-android` plugin 而不是 `kotlin` plugin:

```

buildscript {
    ext.kotlin_version = '1.3.21'

    ...

    dependencies {
        classpath 'com.android.tools.build:gradle:3.2.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

plugins {
    id 'com.android.application'
    id 'kotlin-android'
}

```

```

buildscript {
    dependencies {
        classpath("com.android.tools.build:gradle:3.2.1")
        classpath(kotlin("gradle-plugin", version = "1.3.21"))
    }
}
plugins {
    id("com.android.application")
    id("kotlin-android")
}

```

此外不要忘记配置 [对标准库的依赖](#).

## Android Studio

如果使用 Android Studio, 需要在 android 之下添加以下内容:

```

android {
    ...

    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }
}

```

```

android {
    ...

    sourceSets["main"].java.srcDir("src/main/kotlin")
}

```

这些设置告诉 Android Studio, kotlin 目录是一个源代码根目录, 因此当工程模型装载进入 IDE 时, 就可以正确地识别这个目录. 或者, 你也可以将 Kotlin 类放在 Java 源代码目录内, 通常是 `src/main/java`.

## 配置依赖

除了上文讲到的 `kotlin-gradle-plugin` 依赖之外, 你还需要添加 Kotlin 标准库的依赖:

```

repositories {
    mavenCentral()
}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib"
}

```

```

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib"))
}

```

Kotlin 标准库 `kotlin-stdlib` 的编译目标是 Java 6 及以上版本. 此外还有标准库的扩展版本, 其中添加了对 JDK 7 和 JDK 8 的一些特性的支持. 要使用这些版本, 请添加以下依赖之一, 而不要使用标准的 `kotlin-stdlib`:

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7"
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
```

```
implementation(kotlin("stdlib-jdk7"))
implementation(kotlin("stdlib-jdk8"))
```

在 Kotlin 1.1.x 版本中, 请使用 `kotlin-stdlib-jre7` 和 `kotlin-stdlib-jre8` .

如果你的编译目标平台是 JavaScript, 请使用 `stdlib-js` 依赖项.

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-js"
```

```
implementation(kotlin("stdlib-js"))
```

如果你的项目使用了 [Kotlin 反射功能](#), 或测试功能, 那么还需要添加相应的依赖:

```
implementation "org.jetbrains.kotlin:kotlin-reflect"
testImplementation "org.jetbrains.kotlin:kotlin-test"
testImplementation "org.jetbrains.kotlin:kotlin-test-junit"
```

```
implementation(kotlin("reflect"))
testImplementation(kotlin("test"))
testImplementation(kotlin("test-junit"))
```

从 Kotlin 1.1.2 版开始, `org.jetbrains.kotlin` 组之下的依赖项, 默认会使用从 Kotlin plugin 得到的版本号. 你也可以使用以下的依赖项完整语法, 手动指定版本号:

```
implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
```

```
implementation(kotlin("stdlib", kotlinVersion))
```

## 处理注解

Kotlin 通过 *Kotlin 注解处理工具* (kapt) 支持注解的处理. kapt 在 Gradle 中的使用方法请参见 [kapt 章节](#).

## 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, 因此只会编译那些变更过的文件.

增量编译模式支持 Kotlin/JVM 和 Kotlin/JS 工程. 从 Kotlin 1.1.1 版开始, 对 Kotlin/JVM 工程默认开启, 从 1.3.20 开始, 对 Kotlin/JS 工程默认开启.

有以下几种方式可以覆盖默认的设置:

- 修改 Gradle 配置文件: 在 `gradle.properties` 或 `local.properties` 文件中, 对于 Kotlin/JVM 工程, 添加 `kotlin.incremental=<value>` 对 Kotlin/JS 工程, 添加 `kotlin.incremental.js=<value>` . `<value>` 是 boolean 值, 指定是否使用增量编译模式.
- 修改 Gradle 命令行参数: 添加参数 `-Pkotlin.incremental` 或 `-Pkotlin.incremental.js` , 参数值为 boolean 值, 指定是否使用增量编译模式. 注意, 这种情况下应该向所有后续的编译命令都添加这个参数, 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.

注意, 上述两种方式, 初次编译都不会是增量编译.

## 对 Gradle 编译缓存的支持 (从 1.2.20 版开始支持)

Kotlin 插件支持 [Gradle 编译缓存](#) (需要 Gradle 4.3 或更高版本; 对于 4.3 以下版本, 编译缓存会被禁用).

如果想要对所有的 Kotlin 编译任务禁用缓存, 可以将系统属性 `kotlin.caching.enabled` 设置为 `false` (也就是使用参数 `-Dkotlin.caching.enabled=false` 来执行编译).

如果你使用 [kapt](#), 请注意, 注解处理任务默认不会缓存. 但你可以手动启用缓存功能. 详情请参见 [kapt 章节](#).

## 编译选项

如果需要指定额外的编译选项, 请使用 Kotlin 编译任务的 `kotlinOptions` 属性.

当编译的目标平台为 JVM 时, 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\(build variant\)](#) 的名称, 完整名称是 `compile<BuildVariant>Kotlin`, 比如, `compileDebugKotlin`, `compileReleaseUnitTestKotlin`.

当编译的目标平台为 JavaScript 时, 编译任务名分别是 `compileKotlin2Js` 和 `compileTestKotlin2Js`, 针对自定义源代码集的编译任务名, 是 `compile<Name>Kotlin2Js`.

要对单个编译任务进行配置, 请使用它的名称. 示例如下:

```
compileKotlin {
    kotlinOptions.suppressWarnings = true
}

//或者

compileKotlin {
    kotlinOptions {
        suppressWarnings = true
    }
}
```

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile
// ...

val compileKotlin: KotlinCompile by tasks

compileKotlin.kotlinOptions.suppressWarnings = 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.

也可以对项目中的所有 Kotlin 编译任务进行配置:

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile).all {
    kotlinOptions { ... }
}
```

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

tasks.withType<KotlinCompile> {
    kotlinOptions.suppressWarnings = true
}
```

Gradle 任务所支持的编译选项完整列表如下:

## 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		false
freeCompilerArgs	指定额外的编译参数, 可以是多个		[]

## JVM 和 JS 任务支持的共通属性

Name	Description	Possible values	Default value
apiVersion	只允许使用指定的版本的运行库中的 API	“1.0”, “1.1”, “1.2”, “1.3”, “1.4 (实验性功能)”	
languageVersion	指定源代码所兼容的 Kotlin 语言版本	“1.0”, “1.1”, “1.2”, “1.3”, “1.4 (实验性功能)”	

## 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
jdkHome	如果 JDK home 目录路径与默认的 JAVA_HOME 值不一致, 这个参数可以指定 JDK home 目录路径, 这个路径将被添加到 classpath 内		
jvmTarget	指定编译输出的 JVM 字节码的版本 (1.6 或 1.8), 默认为 1.6	“1.6”, “1.8”	“1.6”
noJdk	不要将 Java 运行库包含到 classpath 内		false
noReflect	不要将 Kotlin 的反射功能实现库包含到 classpath 内		true
noStdlib	不要将 Kotlin 的运行库包含到 classpath 内		true

## 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	指定是否调用 main 函数	“call”, “noCall”	“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	指定编译器生成的模块类型	“plain”, “amd”, “commonjs”, “umd”	“plain”
noStdlib	不使用默认附带的 Kotlin 标准库(stdlib)		true
outputFile	指定输出文件的路径		
sourceMap	指定是否生成源代码映射文件(source map)		false
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”	
sourceMapPrefix	指定源代码映射文件中的路径前缀		
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 arrays		true

## 生成文档

要对 Kotlin 项目生成文档, 请使用 [Dokka](#); 相关的配置方法, 请参见 [Dokka README](#). 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.

## OSGi

关于对 OSGi 的支持, 请参见 [Kotlin 与 OSGi](#).



## 使用 Gradle Kotlin DSL

使用 [Gradle Kotlin DSL](#) 时, 请使用 `plugins { ... }` 来添加 Kotlin 插件. 如果你使用 `apply { plugin(...) }` 来添加插件, 可能会发生错误, 无法解析那些由 Gradle Kotlin DSL 生成的扩展. 为了解决这个问题, 可以将出错的代码注释掉, 执行 Gradle 的 `kotlinDslAccessorsSnapshot` 任务, 再将代码添加回来, 然后重新编译, 或者重新将工程导入到 IDE.

## 示例

以下示例演示了 Gradle plugin 的一些可能的配置:

- [Kotlin](#)
- [Java 代码与 Kotlin 代码的混合](#)
- [Android](#)
- [JavaScript](#)

## 使用 Maven

### 插件与版本

`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。

可以通过 `kotlin.version` 变量来指定你希望使用的 Kotlin 版本:

```
<properties>
  <kotlin.version>1.3.21</kotlin.version>
</properties>
```

### 依赖

Kotlin 有一个内容广泛的标准库, 可以在你的应用程序中使用. 请在 pom 文件中添加以下依赖设置:

```
<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 JDK 8, 你可以使用 Kotlin 标准库的扩展版本, 其中包含了针对 JDK 新版本中新增 API 的额外的扩展函数. 请使用 `kotlin-stdlib-jdk7` 或 `kotlin-stdlib-jdk8` 依赖(根据你的 JDK 版本决定), 而不是通常的 `kotlin-stdlib`. (对 `jdk` 包的依赖是 Kotlin 1.2.0 版本开始引入, 对于 Kotlin 1.1.x 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](#), 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 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</testSourceDirectory>
</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>
```

## 编译 Kotlin 和 Java 的混合源代码

要编译混合源代码的应用程序, 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>${kotlin.version}</version>
      <executions>
        <execution>
          <id>compile</id>
          <goals> <goal>compile</goal> </goals>
          <configuration>
            <sourceDirs>
              <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
              <sourceDir>${project.basedir}/src/main/java</sourceDir>
            </sourceDirs>
          </configuration>
        </execution>
        <execution>
          <id>test-compile</id>
          <goals> <goal>test-compile</goal> </goals>
          <configuration>
            <sourceDirs>
              <sourceDir>${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>

```

增量编译(Incremental compilation)

为了提高编译速度, 你可以打开 Maven 的增量编译模式(从 Kotlin 1.1.2 版开始支持). 方法是定义 `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.

## 处理注解

详情请参见 [Kotlin 注解处理工具](#) ( `kapt` ).

## Coroutines support

对 [协程](#) 的支持是从 Kotlin 1.2 开始新增的一个实验性功能, 因此如果你在项目中使用了协程, Kotlin 编译器会报告一个警告信息. 在你的 `pom.xml` 文件中添加以下代码, 可以关闭这个警告:

```
<configuration>
  <experimentalCoroutines>enable</experimentalCoroutines>
</configuration>
```

## 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:

```
<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>
```

## 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:

```

<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>
        </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
```

## 编译到 JavaScript

为了编译到 JavaScript, 你需要将 `compile` 的目标设置为 `js` 和 `test-js`:

```

<plugin>
  <groupId>org.jetbrains.kotlin</groupId>
  <artifactId>kotlin-maven-plugin</artifactId>
  <version>${kotlin.version}</version>
  <executions>
    <execution>
      <id>compile</id>
      <phase>compile</phase>
      <goals>
        <goal>js</goal>
      </goals>
    </execution>
    <execution>
      <id>test-compile</id>
      <phase>test-compile</phase>
      <goals>
        <goal>test-js</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

此外还需要修改标准库的依赖项:

```

<groupId>org.jetbrains.kotlin</groupId>
<artifactId>kotlin-stdlib-js</artifactId>
<version>${kotlin.version}</version>

```

要支持单元测试, 还需要添加 `kotlin-test-js` 依赖项.

更多详细信息, 请参见教程: [使用 Maven 编译 Kotlin 和 JavaScript 入门](#).

指定编译选项

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:

```
<plugin>
  <groupId>org.jetbrains.kotlin</groupId>
  <artifactId>kotlin-maven-plugin</artifactId>
  <version>${kotlin.version}</version>
  <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:

```
<project ...>
  <properties>
    <kotlin.compiler.languageVersion>1.0</kotlin.compiler.languageVersion>
  </properties>
</project>
```

支持的编译选项列表如下:

JVM 和 JS 支持的共通属性

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	“1.0” “1.1” “1.2” “1.3” , “1.4 (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	“1.0” “1.1” “1.2” “1.3” , “1.4 (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 <a href="#">编译器插件</a>		[]
pluginOptions		供编译器插件使用的选项		[]
args		额外的编译器参数		[]

JVM 独有的属性

名称	Maven 属性名	描述	可以选择的值	默认值
jvmTarget	kotlin.compiler.jvmTarget	指定编译输出的 JVM 字节码的版本	“1.6” , “1.8”	“1.6”
jdkHome	kotlin.compiler.jdkHome	如果 JDK home 目录路径与默认的 JAVA_HOME 值不一致, 这个参数可以指定 JDK home 目录路径, 这个路径将被添加到 classpath 内		

JS 独有的属性

名称	Maven 属性名	描述	可以选择的值	默认值
outputFile		指定输出文件的路径		
metaInfo		指定是否生成带有 metadata 的 .meta.js 和 .kjsm 文件. 用于创建库	true, false	true
sourceMap		指定是否生成源代码映射文件(source map)	true, false	false
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”	“inlining”
sourceMapPrefix		指定源代码映射文件中的路径前缀		
moduleKind		指定编译器生成的模块类型	“plain”, “amd”, “commonjs”, “umd”	“plain”

## 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](#); 相关的配置方法, 请参见 [Dokka README](#). 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.

## OSGi

关于对 OSGi 的支持, 请参见 [Kotlin 与 OSGi](#).

## 示例

我们提供了一个 Maven 工程示例, 可以 [通过 GitHub 仓库下载](#).



## 使用 Ant

### 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](#) 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

```

<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

```

<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:

```

<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.

## Kotlin 与 OSGi

要使用 Kotlin 的 OSGi 支持功能, 你需要使用 `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`:

```
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

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:

```
dependencies {
  compile (
    [group: 'some.group.id', name: 'some.library', version: '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 设置呢?

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, 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.

## 编译器插件

### All-open 编译器插件

Kotlin 默认会将类及其成员定义为 `final` 的, 而某些框架或库要求类为 `open`, 比如 Spring AOP, 因此导致使用这些框架或库时的不便. *all-open* 编译器插件可以帮助 Kotlin 满足这些框架或库的要求, 如果类添加了某个特定的注解, 插件会将这个类以及它的成员变为 `open`, 而不必明确地标记 `open` 关键字.

比如, 当使用 Spring 时, 你不需要所有的类都是 `open` 的, 而只需要添加了特定注解的类为 `open`, 比如 `@Configuration` 注解, 或 `@Service` 注解. *all-open* 插件可以指定这些注解.

对 *all-open* 插件, 我们提供 Gradle 和 Maven 的支持, 以及完全的 IDE 支持.

注意: 对于 Spring, 你可以使用 `kotlin-spring` 编译器插件(详情请 [参见下文](#)).

### 在 Gradle 中使用

将插件 artifact 添加到编译脚本的依赖项目中, 然后应用插件:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-allopen"
```

或者, 你也可以使用 `plugins` 代码段来应用这个插件:

```
plugins {
    id "org.jetbrains.kotlin.plugin.allopen" version "1.3.21"
}
```

然后指定将哪些注解标注的类变为 `open`:

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation", "com.third.Annotation")
}
```

如果类 (或者它的任何一个超类) 标注了 `com.my.Annotation` 注解, 那么类本身, 以及它的所有成员都会变为 `open` 的.

对于元注解(meta-annotation)也有效:

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // 这个类将会变为 open
```

由于 `MyFrameworkAnnotation` 被标注了 *all-open* 的元注解(meta-annotation) `com.my.Annotation`, 因此它也会成为一个 *all-open* 注解.

### 在 Maven 中使用

下面是在 Maven 中使用 *all-open* 插件的方法:

```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <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 注解工作原理的详细信息, 请参照上文的 “在 Gradle 中使用” 小节。

## Spring 支持

如果你使用 Spring, 你可以使用 *kotlin-spring* 编译器插件, 而不必手动指定 Spring 注解。 *kotlin-spring* 是在 *all-open* 之上的一层封装, 工作方式完全相同。

与 all-open 一样, 你需要将 *kotlin-spring* 插件添加到编译脚本的依赖项目中:

```

buildscript {
  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
  }
}

apply plugin: "kotlin-spring" // 而不是使用 "kotlin-allopen"

```

或者使用 Gradle plugin DSL 语法:

```

plugins {
  id "org.jetbrains.kotlin.plugin.spring" version "1.3.21"
}

```

在 Maven 中, 需要启用 `spring` 插件:

```

<compilerPlugins>
  <plugin>spring</plugin>
</compilerPlugins>

```

这个 plugin 将会指定以下注解: [@Component](#), [@Async](#), [@Transactional](#), [@Cacheable](#), [@SpringBootTest](#).

得益于元注解的支持, 使用 `@Configuration`, `@Controller`, `@RestController`, `@Service` 或 `@Repository` 标注的类, 会自动变为 `open`, 因为这些注解被标注了 `@Component` 注解.

当然, 你也可以在同一个工程中同时使用 `kotlin-allopen` 和 `kotlin-spring` 插件.

注意, 如果你使用了 [start.spring.io](https://start.spring.io) 服务生成的项目模板, 那么 `kotlin-spring` 插件默认会打开.

### 在命令行中使用

All-open 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/allopen-compiler-plugin.jar
```

你可以直接指定 all-open 注解, 使用插件的 `annotation` 选项, 或者使用”预先设定”的 all-open 注解. all-open 目前唯一可用的预选设定是 `spring`.

```
# 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.allopen:annotation=com.my.Annotation  
-P plugin:org.jetbrains.kotlin.allopen:preset=spring
```

### No-arg 编译器插件

*no-arg* 编译器插件会为标注了特定注解的类产生一个额外的无参数的构造器.

自动产生的构造器是合成的(synthetic), 因此在 Java 或 Kotlin 中不能直接调用它, 但可以通过反射来调用.

这个功能使得 Java Persistence API (JPA) 可以创建一个类的实例, 即使从 Kotlin 或 Java 的角度看来, 这个类并不存在 0 个参数的构造器 (关于 `kotlin-jpa` 插件, 请参见 [下文](#)).

### 在 Gradle 中使用

使用方法与 all-open 插件很类似.

你需要添加插件, 指定要对类产生无参数构造器的注解列表.

```
buildscript {  
    dependencies {  
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"  
    }  
}  
  
apply plugin: "kotlin-noarg"
```

或者使用 Gradle plugin DSL 语法:

```
plugins {  
    id "org.jetbrains.kotlin.plugin.noarg" version "1.3.21"  
}
```

然后指定要对类产生无参数构造器的注解列表:

```
noArg {  
    annotation("com.my.Annotation")  
}
```

如果你希望插件在合成的构造器中执行初始化逻辑, 可以打开 `invokeInitializers` 选项. 从 Kotlin 1.1.3-2 版开始, 这个选项默认是关闭的, 因为存在 bug: [KT-18667](#) 和 [KT-18668](#), 我们会在未来的版本中解决这些 bug.

```
noArg {
    invokeInitializers = true
}
```

## 在 Maven 中使用

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <configuration>
    <compilerPlugins>
      <!-- 或者使用 "jpa", 支持 JPA -->
      <plugin>no-arg</plugin>
    </compilerPlugins>

    <pluginOptions>
      <option>no-arg:annotation=com.my.Annotation</option>
      <!-- 在合成的构造器中, 调用实例的初始化逻辑 -->
      <!-- <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

与 *kotlin-spring* 类似, *kotlin-jpa* 是在 *no-arg* 之上的一层封装. 这个插件会将 [@Entity](#), [@Embeddable](#), 以及 [@MappedSuperclass](#) 注解指定为 *no-arg* 注解. 在 Gradle 中, 你需要添加以下代码:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:${kotlin_version}"
    }
}

apply plugin: "kotlin-jpa"
```

或者使用 Gradle plugin DSL 语法:

```
plugins {
    id "org.jetbrains.kotlin.plugin.jpa" version "1.3.21"
}
```

在 Maven 中, 需要启用 `jpa` 插件:



```
<compilerPlugins>
  <plugin>jpa</plugin>
</compilerPlugins>
```

### 在命令行中使用

与 *all-open* 类似, 你需要将这个插件的 JAR 文件添加到编译器的插件类路径中, 然后指定 *no-arg* 注解, 或者使用预先设定的注解:

```
-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

*sam-with-receiver* 编译器插件, 可以将被注解的 Java “单个抽象方法” (SAM: Single Abstract Method) 的第一个参数, 转换为 Kotlin 中的接受者. 只有当 SAM 接口被当作 Kotlin 的 Lambda 表达式来传递时, 这个转换才起作用. 对 SAM 方法调用适配, 以及 SAM 实例构造都是如此 (详情请参见 [SAM 转换](#)).

示例如下:

```
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' 的实例

        println("$name is started")
        context.executeTask(this)
        println("$name is finished")
    }
}
```

### 在 Gradle 中使用

这个插件的使用方法与 *all-open* 和 *no-arg* 插件基本相同, 区别是 *sam-with-receiver* 插件没有内建的默认设定, 因此你要指定需要特别处理的注解列表.

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-sam-with-receiver:$kotlin_version"
    }
}

apply plugin: "kotlin-sam-with-receiver"
```

然后指定 SAM-with-receiver 注解列表:

```
samWithReceiver {
    annotation("com.my.Annotation")
}
```

## 在 Maven 中使用

```
<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>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-maven-sam-with-receiver</artifactId>
      <version>${kotlin.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

## 在命令行中使用

只需要将插件的 JAR 文件添加到编译器的类路径, 然后指定 sam-with-receiver 注解列表:

```
-Xplugin=$KOTLIN_HOME/lib/sam-with-receiver-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.samWithReceiver:annotation=com.my.SamWithReceiver
```

# 代码风格迁移指南

## 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](#) 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 时期望的结果, 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.



在实际使用中, 很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”](#)), 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 → 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 `Settings → Editor → Code Style → Kotlin → From... → Predefined Style → 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>
```

警告: 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.



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. 首先在 `Settings → Editor → Code Style → Kotlin` 对话框中将 `scheme` 切换为 `Project`, 然后在 `Load` 标签页的 “*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`.

# Kotlin 的演化

## Kotlin 的演化

### 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 *实验性* 状态. 实验性的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实验性功能, 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, 使它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](#), 在开发者社区的帮助下, 并通过与 [Kotlin 基金会](#) 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](#) (目前是 Andrey Breslav) 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](#) 任命的 [语言委员会](#) 的批准.(语言委员会目前由 Jeffrey van Gogh, William R. Cook 和 Andrey Breslav 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](#).

## 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, 实验性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. 某个功能从实验性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.

## 实验性功能

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 *实验性* 状态, 并且 *预期会被改变*. 实验性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. 我们会确保使用者不会意外地使用到实验性功能. 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, 实验性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](#)。

## 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 @Experimental 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 都应该被看作实验性的功能, 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 和 -api-version 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。通常会至少支持一个旧版本。因此实际上为使用者留下了相当于两次完整功能发布周期的时间可以进行代码迁移 (通常相当于 2 年)。在新版本的编译器中使用旧版本的 kotlin-stdlib 或 kotlin-reflect, 但不指定兼容性标记, 这样的用法是不推荐的, 而且这种情况下编译器会报告 [警告](#)。

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中使用, 也可以在 [Gradle](#) 和 [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).



## 各部分组件的稳定性

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](#) 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](#) 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](#) 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.

<b>** 组件 **</b>	<b>** 进入该状态的版本 **</b>	<b>** 源代码稳定性 **</b>	<b>** 二进制发布版稳定性 **</b>
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 1.3 兼容性指南

在 Kotlin 语言设计的基本原则就是 *保证语言的现代化* 和 *语言版本升级平滑便利*. 第一条认为, 阻碍语言演进的那些元素应该删除, 后一条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. 本文档不讨论从其他语言(比如, Java)的观点来看 Kotlin 代码的兼容性如何.

### 调用 `<clinit>` 时的构造器参数计算顺序

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](#)

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](#)

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](#)

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. 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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#), [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](#)

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](#)

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](#)

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](#)

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:

- 1.1X, 1.2X: 对于 `operator mod` 声明产生警告, 也对这类运算符的调用产生警告
- 1.3X: 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
- 1.4X: 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](#), [KT-20589](#). See also [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
- 1.2X: 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.
- 1.3X: 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](#)

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
- 1.2X: 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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

**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-compatible.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

# FAQ

## FAQ

### 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 以及 Native 应用. Kotlin 由 [JetBrains 公司](#) 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 1.3.21, 发布日期是 2019 年 02 月 06 日.

### 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](#) 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大, 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, 也可以反过来在 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, 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, 可以用在服务器端, 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](#).

### 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](#). 详情请参见 [服务端端开发的相关资源](#).

### 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/JS 来开发 web 客户端. Kotlin 可以使用 [DefinitelyTyped](#) 中的定义, 为 JavaScript 共通库获取静态类型能力, 而且兼容于既有的 JavaScript 模块系统, 比如 AMD 和 CommonJS. 详情请参见 [客户端开发的相关资源](#).

### 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](#).

### 我可以使用 Kotlin 进行 native 开发吗?

Kotlin/Native 目前 [正在开发中](#). 它可以将 Kotlin 代码编译为原生代码, 运行时无需 VM. 已发布过一个技术预览版, 但还不能用于生产环境, 而且还不能支持我们期望在 1.0 版时支持的所有平台. 详情请参见 [宣布 Kotlin/Native 的博文](#).

### 有哪些 IDE 支持 Kotlin?

大多数主流 Java IDE 支持 Kotlin, 包括 [IntelliJ IDEA](#), [Android Studio](#), [Eclipse](#) 以及 [NetBeans](#). 此外, 还有一个 [命令行编译器](#), 可以用来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](#), [Maven](#), [Ant](#), 以及 [Kobalt](#). 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 只支持 Java 6 吗?

不是的. Kotlin 允许你选择生成 Java 6 和 Java 8 兼容的字节码. 对高版本的 Java 平台, 可以生成更加优化的代码.

### 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.

### 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](#), [Pinterest](#), [Basecamp](#) 以及 [Corda](#).

### Kotlin 的开发者是谁?

Kotlin 主要是由 JetBrains 公司的一个工程师团队(目前 50+ 人)开发的. 语言设计的领导者是 [Andrey Breslav](#). 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 [这个网站](#) 开始. 在这里, 你可以下载编译器, [在线试运行代码](#), 访问各种资源, 阅读 [参考文档](#) 和 [教程](#).

### 有关于 Kotlin 的书籍吗?

关于 Kotlin, 已经有了 [很多书籍](#), 包括 Kotlin 开发组成员 Dmitry Jemerov 和 Svetlana Isakova 编写的 [Kotlin 实战\(Kotlin in Action\)](#), 以及针对 Android 开发者的 [面向 Android 的 Kotlin 手册\(Kotlin for Android Developers\)](#).

### 有关于 Kotlin 的在线课程吗?

有一些关于 Kotlin 的课程, 包括 Kevin Jones 的 [Pluralsight Kotlin 课程](#), Hadi Hariri 的 [O'Reilly 课程](#), 以及 Peter Sommerhoff 的 [Udemy Kotlin 课程](#).

此外, 在 YouTube 和 Vimeo 上, 还有很多 [Kotlin 演讲](#) 录像.

#### 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](#), [StackOverflow](#), 以及更活跃的 [Kotlin Slack](#) (到 2018 年 10 月, 成员接近 20000 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](#) 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](#) 活动.

#### 有 Kotlin 开发者大会吗?

是的. 官方的 [Kotlin 开发者大会](#) 由 JetBrains 公司每年举办一次. [2017 年](#) 在 San-Francisco 举行, 2018 年在 Amsterdam 举行. 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](#) 找到即将举行的演讲列表.

#### Kotlin 有社交媒体帐号吗?

是的. 最活跃的 Kotlin 帐号是 [Twitter 帐号](#).

#### 还有关于 Kotlin 的其他在线资源吗?

在各种网站上有很多 [在线资源](#), 包括社区成员编写的 [Kotlin Digests](#), 一份 [通讯](#), 一个 [博客](#), 等等.

#### 在哪里可以得到高分辨率的 Kotlin logo?

可以在 [这个地址](#) 下载 Logo. 压缩包中的 `guidelines.pdf` 文件包含一些简单的规则, 使用 Logo 时请注意遵守.

## 与 Java 语言的比较

### 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](#).
- [没有原生类型\(raw type\)](#)
- Kotlin 中的数组是 [类型不可变的](#)
- 与 Java 中的 SAM 变换方案相反, Kotlin 中存在专门的 [函数类型\(function type\)](#)
- 不使用通配符的 [使用处类型变异\(Use-site variance\)](#)
- Kotlin 中不存在受控 [异常](#)

### Java 中有, 而 Kotlin 中没有的东西

- [受控异常](#)
- 不是类的 [基本数据类型](#)
- [静态成员](#)
- [非私有的域\(Non-private field\)](#)
- [通配符类型\(Wildcard-type\)](#)
- [条件\(三元\)运算符 a ? b : c](#)

### Kotlin 中有, 而 Java 中没有的东西

- [Lambda 表达式](#) + [内联函数](#) = 实现自定义的控制结构
- [扩展函数](#)
- [Null 值安全性](#)
- [类型智能转换](#)
- [字符串模板](#)
- [属性](#)
- [主构造器](#)
- [委托\(First-class delegation\)](#)
- [变量和属性的类型推断](#)
- [单例\(Singleton\)](#)
- [声明处类型变异\(Declaration-site variance\)](#) 和 [类型投射\(Type projection\)](#)
- [值范围表达式](#)
- [操作符重载](#)
- [同伴对象\(Companion object\)](#)
- [数据类](#)
- [集合的接口定义区分为只读集合与可变集合](#)
- [协程](#)

