Kotlin 语言文档

概述

使用 Kotlin 进行服务器端开发

Kotlin 非常适合开发服务器端应用程序，可以让你编写简明且表现力强的代码，同时保持与现有基于 Java 的技术栈的完全兼容性以及平滑的学习曲线：

- **表现力**：Kotlin 的革新式语言功能，例如支持类型安全的构建器和委托属性，有助于构建强大而易于使用的抽象。
- **可伸缩性**：Kotlin 对协程的支持有助于构建服务器端应用程序，伸缩到适度的硬件要求以应对大量的客户端。
- **互操作性**：Kotlin 与所有基于 Java 的框架完全兼容，可以让你保持熟悉的技术栈，同时获得更现代化语言的优势。
- **迁移**：Kotlin 支持大型代码库从 Java 到 Kotlin 逐步迁移。你可以开始用 Kotlin 编写新代码，同时系统中较旧部分继续用 Java。
- **工具**：除了很棒的 IDE 支持之外，Kotlin 还为 IntelliJ IDEA Ultimate 的插件提供了框架特定的工具（例如 Spring）。
- **学习曲线**：对于 Java 开发人员，Kotlin 入门很容易。包含在 Kotlin 插件中的自动 Java 到 Kotlin 的转换器有助于迈出第一步。Kotlin 心印 通过一系列互动练习提供了语言主要功能的指南。

使用 Kotlin 进行服务器端开发的框架

- Spring 利用 Kotlin 的语言功能提供更简洁的 API，从版本 5.0 开始。在线项目生成器可以让你用 Kotlin 快速生成一个新项目。
- Vert.x 是在 JVM 上构建响应式 Web 应用程序的框架，为 Kotlin 提供了 专门支持，包括完整的文档。
- Ktor 是 JetBrains 为在 Kotlin 中创建 Web 应用程序而构建的框架，利用协程实现高可伸缩性，并提供易于使用且合乎惯用法的 API。
- kotlinx.html 是可在 Web 应用程序中用于构建 HTML 的 DSL。它可以作为传统模板系统（如JSP和FreeMarker）的替代品。
- Micronaut 是基于 JVM 的现代全栈框架，用于构建模块化、易于测试的微服务与无服务器应用程序。它带有许多内置的便捷功能。
- Javalin 是用于 Kotlin 与 Java 的非常轻量级的 Web 框架，支持 WebSockets、HTTP2 与异步请求。
- 通过相应 Java 驱动程序进行持久化的可用选项包括直接 JDBC 访问、JPA 以及使用 NoSQL 数据库。对于 JPA，kotlin-jpa 编译器插件使 Kotlin 编译的类适应框架的要求。

部署 Kotlin 服务器端应用程序

Kotlin 应用程序可以部署到支持 Java Web 应用程序的任何主机，包括 Amazon Web Services、Google Cloud Platform 等。

要在 Heroku 上部署 Kotlin 应用程序，可以按照 Heroku 官方教程来做。

AWS Labs 提供了一个示例项目，展示了 Kotlin 编写 AWS Lambda 函数的使用。

谷歌云平台（Google Cloud Platform）提供了一系列将 Kotlin 应用程序部署到 GCP 的教程，包括 Ktor 与 App Engine 应用及 Spring 与 App engine 应用。此外，还有一个交互式代码实验室（interactive code lab）用于部署 Kotlin Spring 应用程序。

Kotlin 用于服务器端的用户

Corda 是一个开源的分布式分类帐平台，由各大银行提供支持，完全由 Kotlin 构建。

JetBrains 账户，负责 JetBrains 整个许可证销售和验证过程的系统 100% 由 Kotlin 编写，自 2015 年生产运行以来，一直没有重大问题。

下一步

- 使用 Http Servlet 创建 Web 应用程序及使用 Spring Boot 创建 RESTful Web 服务教程将向你展示如何在 Kotlin 中构建和运行非常小的 Web 应用程序。
- 关于更深入的介绍，请查看本站的参考文档及 Kotlin 心印。
- Micronaut 还提供了很多详细的指南，展示了如何使用 Kotlin 构建微服务。

使用 Kotlin 进行 Android 开发

自 2019 年 Google I/O 以来，Kotlin 就成为了 Android 移动开发的首选。

使用 Kotlin 进行 Android 开发，可以受益于：

- **代码更少、可读性更强**。花更少的时间来编写代码与理解他人的代码。
- **成熟的语言与环境**。自 2011 年创建以来，Kotlin 不仅通过语言而且通过强大的工具在整个生态系统中不断发展。现在，它已无缝集成到 Android Studio 中，并被许多公司积极用于开发 Android 应用程序。
- **Android Jetpack 与其他库中的 Kotlin 支持**。KTX 扩展 为现有的 Android 库添加了 Kotlin 语言特性，如协程、扩展函数、lambdas 与命名参数。
- **与 Java 的互操作性**。可以在应用程序中将 Kotlin 与 Java 编程语言一起使用，而无需将所有代码迁移到 Kotlin。
- **支持多平台开发**。不仅可以使用 Kotlin 开发 Android，还可以开发 iOS、后端与 Web 应用程序。享受在平台之间共享公共代码的好处。
- **代码安全**。更少的代码与更好的可读性导致更少的错误。Kotlin 编译器检测这些剩余的错误，从而使代码安全。
- **易学易用**。Kotlin 非常易于学习，尤其是对于 Java 开发人员而言。
- **大社区**。Kotlin 得到了社区的大力支持与许多贡献，该社区在全世界范围内都在增长。根据 Google 的说法，Play 商店前 1000 个应用中有 60% 以上使用 Kotlin。

许多初创公司与财富 500 强公司已经使用 Kotlin 开发了 Android 应用程序——详情请见 面向 Kotlin 开发者的谷歌网站。

如果想开始使用 Kotlin 进行 Android 开发，请参阅 在 Android 开发中开始使用 Kotlin。

如果是 Android 的新手，并且想学习使用 Kotlin 创建应用程序，请查看 这门 Udacity 课程。

Kotlin JavaScript 概述

Kotlin 提供了 JavaScript 作为目标平台的能力。它通过将 Kotlin 转换为 JavaScript 来实现。目前的实现目标是 ECMAScript 5.1，但也有最终目标为 ECMAScript 2015 的计划。

当你选择 JavaScript 目标时，作为项目一部分的任何 Kotlin 代码以及 Kotlin 附带的标准库都会转换为 JavaScript。然而，这不包括使用的 JDK 和任何 JVM 或 Java 框架或库。任何不是 Kotlin 的文件会在编译期间忽略掉。

Kotlin 编译器努力遵循以下目标：

- 提供最佳大小的输出
- 提供可读的 JavaScript 输出
- 提供与现有模块系统的互操作性
- 在标准库中提供相同的功能，无论是 JavaScript 还是 JVM 目标（尽最大可能程度）。

如何使用

你可能希望在以下情景中将 Kotlin 编译为 JavaScript：

- 创建面向客户端 JavaScript 的 Kotlin 代码
  - **与 DOM 元素交互**。Kotlin 提供了一系列静态类型的接口来与文档对象模型（Document Object Model）交互，允许创建和更新 DOM 元素。
  - **与图形如 WebGL 交互**。你可以使用 Kotlin 在网页上用 WebGL 创建图形元素。
- 创建面向服务器端 JavaScript 的 Kotlin 代码
  - **使用服务器端技术**。你可以使用 Kotlin 与服务器端 JavaScript（如 Node.js）进行交互

Kotlin 可以与现有的第三方库和框架（如 jQuery 或 ReactJS）一起使用。要使用强类型 API 访问第三方框架，可以使用 dukat 工具将 TypeScript 定义从 Definitely Typed 类型定义仓库转换为 Kotlin。或者，你可以使用动态类型访问任何框架，而无需强类型。

Kotlin 兼容 CommonJS、AMD 和 UMD，直截了当与不同的模块系统交互。

Kotlin/JS 今天与明天

想进一步了解 Kotlin/JS 吗？

在这个短片中，Kotlin 开发者布道师 Sebastian Aigner 将为你解释 Kotlin/JS 的主要优点、分享一些技巧与使用场景，并介绍 Kotlin/JS 的计划与即将发布的特性。

https://www.youtube.com/watch?v=fZUL8_kgHXg

Kotlin/JS 入门

要了解如何开始使用 Kotlin 用于 JavaScript 开发，请参阅搭建 Kotlin/JS 项目。

Kotlin/JS 实践实验室

实践实验室是一种长形式的教程，通过一个与特定主题相关的独立项目来帮助了解一种技术。

它们包括示例项目，这些示例项目可用作自己的项目的起点，并包含有用的代码片段与模式。

对于 Kotlin/JS，当前有以下实践实验：

— 使用 React 与 Kotlin/JS 构建 Web 应用程序将指导完成使用 React 框架构建简单 Web 应用程序的过程，展示用于 HTML 的类型安全的 Kotlin DSL 如何使构建响应式 DOM 元素更加方便，并说明了如何使用第三方 React 组件，以及如何从 API 获取信息，同时使用纯 Kotlin/JS 编写整个应用程序逻辑。

— 使用 Kotlin Multiplatform 构建全栈 Web 应用通过构建使用通用代码、序列化与其他多平台范式的客户端服务器应用程序，讲授了构建针对 Kotlin/JVM 与 Kotlin/JS 的应用程序的概念。它还简要介绍了如何将 Ktor 作为服务器与客户端框架使用。

加入 Kotlin/JS 社区

还可以在官方 Kotlin Slack 中加入 #javascript 频道，并与社区和团队聊天。

Kotlin/Native 用于原生开发

Kotlin/Native 是一种将 Kotlin 代码编译为无需虚拟机就可运行的原生二进制文件的技术。它是一个基于 LLVM 的 Kotlin 编译器后端以及 Kotlin 标准库的原生实现。

为什么选用 Kotlin/Native?

Kotlin/Native 的主要设计目标是让 Kotlin 可以为不希望或者不可能使用 *虚拟机* 的平台（例如嵌入式设备或者 iOS）编译。它解决了开发人员需要生成无需额外运行时或虚拟机的自包含程序的情况。

目标平台

Kotlin/Native 支持以下平台：

- iOS（arm32、arm64、模拟器 x86_64）
- macOS（x86_64）
- watchOS（arm32、arm64、x86）
- tvOS（arm64、x86_64）
- Android（arm32、arm64、x86、x86_64）
- Windows（mingw x86_64、x86）
- Linux（x86_64、arm32、arm64、MIPS、MIPS 小端次序）
- WebAssembly（wasm32）

互操作

Kotlin/Native 支持与原生世界的双向互操作。一方面，编译器可创建：

- 用于多个平台的可执行文件
- 用于 C/C++ 项目的静态库或动态库以及 C 语言头文件
- 用于Swift 与 Objective-C 项目的 Apple 框架

另一方面，支持直接在 Kotlin/Native 中使用以下现有库的互操作：

- 静态或动态 C 语言库
- C 语言、Swift 以及 Objective-C 框架

将编译后的 Kotlin 代码包含进用 C、C++、Swift、Objective-C 以及其他语言编写的现有项目中会很容易。直接在 Kotlin/Native 中使用现有原生代码、静态或动态 C 语言库、Swift/Objective-C 框架、图形引擎以及任何其他原生内容也很容易。

Kotlin/Native 库有助于在多个项目之间共享 Kotlin 代码。POSIX、gzip、OpenGL、Metal、Foundation 以及许多其他流行库与 Apple 框架都已预先导入并作为 Kotlin/Native 库包含在编译器包中。

在多个平台之间共享代码

不同目标平台的 Kotlin 与 Kotlin/Native 之间支持多平台项目。这是在多个平台之间共享公共 Kotlin 代码的方式，这些平台包括 Android、iOS、服务器端、JVM、客户端、JavaScript、CSS 以及原生平台。

多平台库为公共 Kotlin 代码提供了必要的 API，并有助于在 Kotlin 代码中一次性开发项目的共享部分，从而将其与所有目标平台共享。

如何开始

教程与文档

Kotlin 新手?可以看看入门页。

建议的文档页:

— C 语言互操作
— Swift/Objective-C 互操作

推荐的教程:

— Hello Kotlin/Native
— 多平台项目:iOS 与 Android
— C 语言 Kotlin/Native 之间的类型映射
— Kotlin/Native 开发动态库
— 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)

在 GitHub 上还有更多示例。

Kotlin 用于数据科学

From building data pipelines to productionizing machine learning models, Kotlin can be a great choice for working with data:

- Kotlin is concise, readable and easy to learn.
- Static typing and null safety help create reliable, maintainable code that is easy to troubleshoot.
- Being a JVM language, Kotlin gives you great performance and an ability to leverage an entire ecosystem of tried and true Java libraries.

Interactive editors

Notebooks such as Jupyter Notebook and Apache Zeppelin provide convenient tools for data visualization and exploratory research. Kotlin integrates with these tools to help you explore data, share your findings with colleagues, or build up your data science and machine learning skills.

Jupyter Kotlin kernel

The Jupyter Notebook is an open-source web application that allows you to create and share documents (aka "notebooks") that can contain code, visualizations, and markdown text. Kotlin-jupyter is an open source project that brings Kotlin support to Jupyter Notebook.

Check out Kotlin kernel's GitHub repo for installation instructions, documentation, and examples.

Zeppelin Kotlin interpreter

Apache Zeppelin is a popular web-based solution for interactive data analytics. It provides strong support for the Apache Spark cluster computing system, which is particularly useful for data engineering. Starting from version 0.9.0, Apache Zeppelin comes with bundled Kotlin interpreter.

Libraries

The ecosystem of libraries for data-related tasks created by the Kotlin community is rapidly expanding. Here are some libraries that you may find useful:

Kotlin 库

- kotlin-statistics 是一个为探索性统计与生产统计中提供扩展函数的库。它支持基本的数字列表/序列/数组函数（从 `sum` 到 `skewness`）、切片操作符（诸如 `countBy`、`simpleRegressionBy`）、分箱（binning）操作符、离散 PDF 采样、朴素贝叶斯分类器、聚类、线性回归等等。
- kmath 是一个受 NumPy 启发的库。这个库支持代数结构与运算、类数组结构、数学表达式、直方图、流运算、commons-math 与 koma 的包装等等。
- krangl 是一个受 R 语言的 dplyr 与 Python 的 pandas 启发的库。这个库提供了采用函数式风格 API 进行数据操作的功能；它还包括过滤、转换、聚合与重塑表格数据的函数。
- lets-plot 是一个用 Kotlin 编写的统计数据绘图库。Lets-Plot 是多平台的，不仅可以用于 JVM，还可以用于 JS 与 Python。
- kravis 是另一个用于表格数据可视化的库，其灵感来自于 Python 的 ggplot。

Java 库

因为 Kotlin 提供了与 Java 互操作的头等支持，所以也可以在用于数据科学的 Kotlin 代码中使用 Java 库。以下是这些库的一些示例：

- DeepLearning4J——一个 Java 深度学习库
- ND4J——用于 JVM 的高效矩阵数学库
- Dex——一个基于 Java 的数据可视化工具
- Smile——一个全面的机器学习、自然语言处理、线性代数、图、插值与可视化系统。除了 Java API，Smile 还提供了函数式的 Kotlin API 以及 Scala 与 Clojure API。
  - Smile-NLP-kt——以 Kotlin 扩展函数与接口格式重写了 Smile 的自然语言处理部分的 Scala 隐式内容。
- Apache Commons Math——一个 Java 通用数学、统计与机器学习库
- OptaPlanner——一个用于优化规划问题的求解器实用程序
- Charts——一个正在开发中的科学 JavaFX 图表库
- CoreNLP——一个自然语言处理工具包
- Apache Mahout——一个回归、聚类与推荐的分布式框架
- Weka——一组用于数据挖掘任务的机器学习算法

如果这个列表还不能满足需求，可以在 Thomas Nield 的 **Kotlin 数据科学资源**摘要中找到更多选项。

用于异步编程等场景的协程

异步或非阻塞程序设计是新的现实。无论我们创建服务端应用、桌面应用还是移动端应用，都很重要的一点是，我们提供的体验不仅是从用户角度看着流畅，而且还能在需要时伸缩（scalable，可扩充/缩减规模）。

这个问题有很多方法，在 Kotlin 中我们采用非常灵活的方法，在语言级提供协程支持，而将大部分功能委托给库，这与 Kotlin 的理念非常一致。

额外收益是，协程不仅打开了异步编程的大门，还提供了大量其他的可能性，例如并发、参与者（actor）等。

如何开始

教程与文档

Kotlin 新手?可以看看入门页。

精选文档页：

- 协程指南
- 基础
- 通道
- 协程上下文与调度器
- 共享的可变状态与并发
- 异步流

推荐的教程：

- 你的第一个 Kotlin 协程程序
- 异步程序设计
- 协程与通道简介动手实验室

示例项目

- kotlinx.coroutines 示例与源代码
- KotlinConf app

在 GitHub 上还有更多示例

多平台程序设计

> 多平台项目是 Kotlin 1.2 与 1.3 中的实验性特性。本文档中描述的所有的语言与工具特性在未来的版本中都可能会有所变化。

在所有平台上都能用是 Kotlin 的一个明确目标，但我们将其视为一个更重要的目标——在多个平台之间共享代码的前提。有了对 JVM、Android、JavaScript、iOS、Linux、Windows、 Mac 甚至像 STM32 这样的嵌入式系统的支持，Kotlin 可以处理现代应用程序的任何组件与所有组件。这为代码与专业知识的复用带来了宝贵的收益，节省了工作量去完成更具挑战任务，而不是将所有东西都实现两次或多次。

它是如何工作的

总得来说，多平台并不是为所有平台编译全部代码。这个模型有其明显的局限性，我们知道现代应用程序需要访问其所运行平台的独有特性。Kotlin 并不会限制你只使用其中所有 API 的公共子集。每个组件都可以根据需要与其他组件共享尽可能多的代码，而通过语言所提供的 `expect/actual` 机制可以随时访问平台 API。

以下是在极简版日志框架中公共逻辑与平台逻辑之间代码共享与交互的示例。其公共代码如下所示：

```
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)
```

├ 为所有平台编译

├ 预期的平台相关 API

├ 可在公共代码中使用预期的 API

它期待目标平台为 `writeLogMessage` 提供平台相关实现，然后公共代码就可以使用此声明而无需考虑它是如何实现的。

在 JVM 上，可以提供一个将日志写到标准输出的实现：

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

在 JavaScript 世界中可用的是一组完全不同的 API，因此可以实现为将日志记录到控制台：

```
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 中我们重新设计了整个多平台模型。我们用于描述多平台 Gradle 项目的新版 DSL 更加灵活，我们会继续努力使项目配置更加简单。

多平台库

公共代码可以依赖于一组涵盖日常任务的库，例如 HTTP、serialization 以及协程管理。此外，丰富的标准库在所有平台上都可用。

你可以随时编写自己的库，提供一个公共的 API，而在每个平台上以不同的方式实现。

使用场景

Android——iOS

移动平台之间共享代码是 Kotlin 多平台的主要使用场景之一，现在可以通过在 Android 与 iOS 之间共享部分代码（如业务逻辑、连接等）来构建移动应用。

参见：

— 移动端多平台特性、案例研究以及一些示例
— 搭建移动端多平台项目

客户端——服务端

代码共享可以带来收益的另一个场景是互联应用，其中的逻辑可以在服务器与运行在浏览器中的客户端中复用。Kotlin 多平台也覆盖了这个场景。

Ktor 框架适用于在互联系统中构建异步的服务器与客户端。

如何开始

教程与文档

Kotlin 新手?可以看看[入门]Getting Started页。

建议的文档页：

— 搭建一个多平台项目
— 平台相关声明

推荐的教程：

— 多平台 Kotlin 库
— 多平台项目：iOS 与 Android

示例项目

— KotlinConf app
— KotlinConf Spinner app

在 GitHub 上还有更多示例

新特性

Kotlin 1.3 的新特性

协程正式发布

历经了漫长而充足的的测试，协程终于正式发布了！这意味着自 Kotlin 1.3 起，协程的语言支持与 API 已完全稳定。参见新版协程概述。

Kotlin 1.3 引入了挂起函数的可调用引用以及在反射 API 中对协程的支持。

Kotlin/Native

Kotlin 1.3 继续改进与完善原生平台。详情请参见 Kotlin/Native 概述。

多平台项目

在 1.3 中，我们完全修改了多平台项目的模型，以提高表现力与灵活性，并使共享公共代码更加容易。此外，多平台项目现在也支持 Kotlin/Native！

与旧版模型的主要区别在于：

- 在旧版模型中，需要将公共代码与平台相关代码分别放在独立的模块中，以 `expectedBy` 依赖项链接。现在，公共代码与平台相关代码放在相同模块的不同源根（source root）中，使项目更易于配置。
- 对于不同的已支持平台，现在有大量的预设的平台配置。
- 更改了依赖配置；现在为每个源根分别指定依赖项。
- 源集（source set）现在可以在任意平台子集之间共享（例如，在一个目标平台为 JS、Android 与 iOS 的模块中，可以有一个只在 Android 与 iOS 之间共享的源集）。
- 现在支持发布多平台库了。

更多相关信息，请参考多平台程序设计文档。

契约

Kotlin 编译器会做大量的静态分析工作，以提供警告并减少模版代码。其中最显著的特性之一就是智能转换——能够根据类型检测自动转换类型。

```
fun foo(s: String?) {
    if (s != null) s.length // 编译器自动将“s”转换为“String”
}
```

然而，一旦将这些检测提取到单独的函数中，所有智能转换都立即消失了：

```
fun String?.isNotNull(): Boolean = this != null

fun foo(s: String?) {
    if (s.isNotNull()) s.length // 没有智能转换 :(
}
```

为了改善在此类场景中的行为，Kotlin 1.3 引入了称为 *契约* 的实验性机制。

契约 让一个函数能够以编译器理解的方式显式描述其行为。目前支持两大类场景：

— 通过声明函数调用的结果与所传参数值之间的关系来改进智能转换分析：

```
fun require(condition: Boolean) {
    // 这是一种语法格式，告诉编译器：
    // “如果这个函数成功返回，那么传入的‘condition’为 true”
    contract { returns() implies condition }
    if (!condition) throw IllegalArgumentException(...)
}

fun foo(s: String?) {
    require(s is String)
    // s 在这里智能转换为“String”，因为否则
    // “require”会抛出异常
}
```

— 在存在高阶函数的情况下改进变量初始化的分析：

```
fun synchronize(lock: Any?, block: () -> Unit) {
    // 告诉编译器：
    // “这个函数会在此时此处调用‘block’，并且刚好只调用一次”
    contract { callsInPlace(block, EXACTLY_ONCE) }
}

fun foo() {
    val x: Int
    synchronize(lock) {
        x = 42 // 编译器知道传给“synchronize”的 lambda 表达式刚好
               // 只调了一次，因此不会报重复赋值错
    }
    println(x) // 编译器知道一定会调用该 lambda 表达式而执行
               // 初始化操作，因此可以认为“x”在这里已初始化
}
```

标准库中的契约

`stdlib`（kotlin 标准库）已经利用契约带来了如上所述的对代码分析的改进。这部分契约是**稳定版的**，这意味着你现在就可以从改进的代码分析中受益，而无需任何额外的 opt-ins：

```
fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // 哇，已经智能转换为非空!
    }
}
```

自定义契约

可以为自己的函数声明契约，不过这个特性是**实验性的**，因为目前的语法尚处于早期原型状态，并且很可能还会更改。另外还要注意，目前 Kotlin 编译器并不会验证契约，因此程序员有责任编写正确合理的契约。

通过调用标准库（stdlib）函数 `contract` 来引入自定义契约，该函数提供了 DSL 作用域：

```
fun String?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }
    return this == null || isEmpty()
}
```

请参见 KEEP 中关于语法与兼容性注意事项的详细信息。

将 when 主语捕获到变量中

在 Kotlin 1.3 中，可以将 `when` 表达式主语捕获到变量中：

```
fun Request.getBody() =
        when (val response = executeRequest()) {
            is Success -> response.body
            is HttpError -> throw HttpException(response.status)
        }
```

虽然已经可以在 `when` 表达式前面提取这个变量，但是在 `when` 中的 `val` 使其作用域刚好限制在 `when` 主体中，从而防止命名空间污染。关于 `when` 表达式的完整文档请参见这里。

接口中伴生对象的 @JvmStatic 与 @JvmField

对于 Kotlin 1.3，可以使用注解 `@JvmStatic` 与 `@JvmField` 标记接口的 `companion` 对象成员。在类文件中会将这些成员提升到相应接口中并标记为 `static`。

例如，以下 Kotlin 代码：

```
interface Foo {
    companion object {
        @JvmField
        val answer: Int = 42

        @JvmStatic
        fun sayHello() {
            println("Hello, world!")
        }
    }
}
```

相当于这段 Java 代码：

```
interface Foo {
    public static int answer = 42;
    public static void sayHello() {
        // ……
    }
}
```

注解类中的内嵌声明

在 Kotlin 1.3 中，注解可以有内嵌的类、接口、对象与伴生对象：

```
annotation class Foo {
    enum class Direction { UP, DOWN, LEFT, RIGHT }

    annotation class Bar

    companion object {
        fun foo(): Int = 42
        val bar: Int = 42
    }
}
```

无参的 main

按照惯例，Kotlin 程序的入口点是一个签名类似于 `main(args: Array<String>)` 的函数，其中 `args` 表示传给该程序的命令行参数。然而，并非每个应用程序都支持命令行参数，因此这个参数往往到最后都没有用到。

Kotlin 1.3 引入了一种更简单的无参 `main` 形式。现在 Kotlin 版的 `Hello,World` 缩短了19个字符！

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

更多元的函数

在 Kotlin 中用带有不同数量参数的泛型类来表示函数类型：`Function0<R>`、`Function1<P0, R>`、`Function2<P0, P1, R>` ……这种方式有一个问题是这个列表是有限的，目前只到 `Function22`。

Kotlin 1.3 放宽了这一限制，并添加了对具有更多元数（参数个数）的函数的支持：

```
fun trueEnterpriseComesToKotlin(block: (Any, Any, …… /* 42 个 */, Any) -> Any) {
    block(Any(), Any(), ……, Any())
}
```

渐进模式

Kotlin 非常注重代码的稳定性与向后兼容性：Kotlin 兼容性策略提到“破坏性变更”（例如，会使以前编译正常的代码现在不能通过编译的变更）只能在主版本（1.2、1.3 等）中引入。

我们相信很多用户可以使用更快的周期，其中关键的编译器修复会即时生效，从而使代码更安全、更正确。因此 Kotlin 1.3 引入了*渐进*编译器模式，可以通过将参数 `-progressive` 传给编译器来启用。

在渐进模式下，语言语义中的一些修复可以即时生效。所有这些修复都有以下两个重要特征：

- 保留了源代码与旧版编译器的向后兼容性，这意味着可以通过非渐进式编译器编译所有可由渐进式编译器编译的代码。
- 只是在某种意义上使代码*更安全*——例如，可以禁用某些不安全的智能转换，可以将所生成代码的行为更改为更可预测、更稳定，等等。

启用渐进模式可能需要重写一些代码，但不会太多——所有在渐进模式启用的修复都已经过精心挑选、通过审核并提供迁移辅助工具。我们希望对于任何积极维护、即将快速更新到最新语言版本的代码库来说，渐进模式都是一个不错的选择。

内联类

> 内联类在 Kotlin 1.3 起才可用，并且目前是**实验性**的。详见其参考文档。

Kotlin 1.3 引入了一种新的声明方式—— `inline class`。内联类可以看作是普通类的受限版，尤其是内联类必须有且只有一个属性：

```
inline class Name(val s: String)
```

Kotlin 编译器会使用此限制来积极优化内联类的运行时表示，并使用底层属性的值替换内联类的实例，其中可能会移除构造函数调用、GC 压力，以及启用其他优化：

```
fun main() {
    // 下一行不会调用构造函数，并且
    // 在运行时，“name”只包含字符串 "Kotlin"
    val name = Name("Kotlin")
    println(name.s)
}
```

详见内联类的参考文档。

无符号整型

> 无符号整数仅在 Kotlin 1.3 起才可用，并且目前是**实验性**的。详见其参考文档。

Kotlin 1.3 引入了无符号整型类型：

- `kotlin.UByte`：无符号 8 比特整数，范围是 0 到 255
- `kotlin.UShort`：无符号 16 比特整数，范围是 0 到 65535
- `kotlin.UInt`：无符号 32 比特整数，范围是 0 到 2^32 - 1
- `kotlin.ULong`：无符号 64 比特整数，范围是 0 到 2^64 - 1

无符号类型也支持其对应有符号类型的大多数操作：

```
// 可以使用字面值后缀定义无符号类型：
val uint = 42u
val ulong = 42uL
val ubyte: UByte = 255u

// 通过标准库扩展可以将有符号类型转换为无符号类型，反之亦然：
val int = uint.toInt()
val byte = ubyte.toByte()
val ulong2 = byte.toULong()

// 无符号类型支持类似的操作符：
val x = 20u + 22u
val y = 1u shl 8
val z = "128".toUByte()
val range = 1u..5u
```

详见其参考文档。

@JvmDefault

> `@JvmDefault` 仅在 Kotlin 1.3 起才可用，并且目前是**实验性**的。详见其参考文档页。

Kotlin 兼容很多 Java 版本，其中包括不支持默认方法的 Java 6 与 Java 7。为了方便起见，Kotlin 编译器可以变通突破这个限制，不过这个变通方法与 Java 8 引入的 `default` 方法并不兼容。

这可能会是 Java 互操作性的一个问题，因此 Kotlin 1.3 引入了 `@JvmDefault` 注解。以此注解标注的方法会生成为 JVM 平台的 `default` 方法：

```
interface Foo {
    // 会生成为“default”方法
    @JvmDefault
    fun foo(): Int = 42
}
```

> 警告！以 `@JvmDefault` 注解标注的 API 会对二进制兼容性产生严重影响。在生产中使用 `@JvmDefault` 之前，请务必仔细阅读其参考文档页。

标准库

多平台 Random

在 Kotlin 1.3 之前没有在所有平台生成随机数的统一方式——我们不得不借助平台相关的解决方案，如 JVM 平台的 `java.util.Random`。这个版本通过引入在所有平台都可用的 `kotlin.random.Random` 类来解决这一问题。

```
val number = Random.nextInt(42)  // 数字在区间 [0, 上限) 内
println(number)
```

isNullOrEmpty 与 orEmpty 扩展

一些类型的 `isNullOrEmpty` 与 `orEmpty` 扩展已经存在于标准库中。如果接收者是 `null` 或空容器，第一个函数返回 `true`；而如果接收者是 `null`，第二个函数回退为空容器实例。Kotlin 1.3 为集合、映射以及对象数组提供了类似的扩展。

在两个现有数组间复制元素

为包括无符号整型数组在内的现有数组类型新增的函数 `array.copyInto(targetArray, targetOffset, startIndex, endIndex)` 使在纯 Kotlin 中实现基于数组的容器更容易。

```
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())
```

associateWith

一个很常见的情况是，有一个键的列表，希望通过将其中的每个键与某个值相关联来构建映射。以前可以通过 `associate { it to getValue(it) }` 函数来实现，不过现在我们引入了一种更高效、更易读的替代方式：`keys.associateWith { getValue(it) }`。

```
val keys = 'a'..'f'
val map = keys.associateWith { it.toString().repeat(5).capitalize() }
map.forEach { println(it) }
```

ifEmpty 与 ifBlank 函数

集合、映射、对象数组、字符序列以及序列现在都有一个 `ifEmpty` 函数，它可以指定一个备用值，当接收者为空容器时以该值代替接收者使用：

```
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"))
```

此外，字符序列与字符串还有一个 `ifBlank` 扩展，它与 `ifEmpty` 类似，只是会检测字符串是否全部都是空白符而不只是空串。

```
val s = "    \n"
println(s.ifBlank { "<blank>" })
println(s.ifBlank { null })
```

反射中的密封类

我们在 `kotlin-reflect` 中添加了一个新的 API，可以列出密封（`sealed`）类的所有直接子类型，即 `KClass.sealedSubclasses`。

小改动

- `Boolean` 类型现在有伴生对象了。
- `Any?.hashCode()` 扩展函数对 `null` 返回 0。
- `Char` 现在提供了 `MIN_VALUE` / `MAX_VALUE` 常量。
- 原生类型伴生对象中的常量 `SIZE_BYTES` 与 `SIZE_BITS`。

工具

IDE 中的代码风格支持

Kotlin 1.3 在 IDE 中引入了对推荐代码风格的支持。其迁移指南参见这里。

kotlinx.serialization

kotlinx.serialization 是一个在 Kotlin 中为（反）序列化对象提供多平台支持的库。以前，它是一个独立项目，不过自 Kotlin 1.3 起，它与其他编译器插件一样随 Kotlin 编译器发行版一起发行。其主要区别在于，无需人为关注序列化 IDE 插件与正在使用的 Kotlin IDE 插件是否兼容了：现在 Kotlin IDE 插件已经包含了序列化支持！

详见这里。

> 请注意，尽管 kotlinx.serialization 现在随 Kotlin 编译器发行版一起发行，但它仍是一个**实验性的**特性。

脚本更新

> 请注意，脚本支持是一项**实验性的**特性，这意味着不会对 API 提供兼容性保证。

Kotlin 1.3 继续发展与改进脚本 API，为脚本定制引入了一些实验性支持，例如添加外部属性、提供静态或动态依赖等等。

关于其他细节，请参考 KEEP-75。

草稿文件支持

Kotlin 1.3 引入了对可运行的*草稿文件 (scratch files)* 的支持。*草稿文件*是一个扩展名为 .kts 的 kotlin 脚本文件，可以在编辑器中直接运行并获取求值结果。

更多细节请参考通用草稿文件文档。

Kotlin 1.2 的新特性

目录

多平台项目（实验性的）

多平台项目是 Kotlin 1.2 中的一个新的**实验性的**特性，允许你在支持 Kotlin 的目标平台——JVM、JavaScript 以及（将来的）Native 之间重用代码。在多平台项目中，你有三种模块：

— 一个*公共*模块包含平台无关代码，以及无实现的依赖平台的 API 声明。

— *平台*模块包含通用模块中的平台相关声明在指定平台的实现，以及其他平台相关代码。

— 常规模块面向指定的平台，既可以是平台模块的依赖，也可以依赖平台模块。

当你为指定平台编译多平台项目时，既会生成公共代码也会生成平台相关代码。

多平台项目支持的一个主要特点是可以通过**预期声明与实际声明**来表达公共代码对平台相关部分的依赖关系。一个预期声明指定一个 API（类、接口、注解、顶层声明等）。一个实际声明要么是该 API 的平台相关实现，要么是一个引用到在一个外部库中该 API 的一个既有实现的别名。这是一个示例：

在公共代码中：

```
// 预期平台相关 API:
expect fun hello(world: String): String

fun greet() {
    // 该预期 API 的用法:
    val greeting = hello("multi-platform world")
    println(greeting)
}

expect class URL(spec: String) {
    open fun getHost(): String
    open fun getPath(): String
}
```

在 JVM 平台代码中：

```
actual fun hello(world: String): String =
    "Hello, $world, on the JVM platform!"

// 使用既有平台相关实现:
actual typealias URL = java.net.URL
```

关于构建多平台项目的详细信息与步骤，请参见其文档。

其他语言特性

注解中的数组字面值

自 Kotlin 1.2 起，注解的数组参数可以通过新的数组字面值语法传入，而无需使用 `arrayOf` 函数：

```
@CacheConfig(cacheNames = ["books", "default"])
public class BookRepositoryImpl {
    // ……
}
```

该数组字面值语法仅限于注解参数。

lateinit 顶层属性与局部变量

`lateinit` 修饰符现在可以用于顶层属性与局部变量了。例如，后者可用于当一个 lambda 表达式作为构造函数参数传给一个对象时，引用另一个必须稍后定义的对象：

```
class Node<T>(val value: T, val next: () -> Node<T>)

fun main(args: Array<String>) {
    // 三个节点的环:
    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() }}, ...")
}
```

检测 lateinit 变量是否已初始化

现在可以通过属性引用的 `isInitialized` 来检测该 lateinit var 是否已初始化：

```
    println("isInitialized before assignment: " + this::lateinitVar.isInitialized)
    lateinitVar = "value"
    println("isInitialized after assignment: " + this::lateinitVar.isInitialized)
```

内联函数带有默认函数式参数

内联函数现在允许其内联函式数参数具有默认值：

```
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)" }
```

源自显式类型转换的信息会用于类型推断

Kotlin 编译器现在可将类型转换信息用于类型推断。如果你调用一个返回类型参数 `T` 的泛型方法并将返回值转换为指定类型 `Foo`，那么编译器现在知道对于本次调用需要绑定类型为 `Foo`。

这对于 Android 开发者来说尤为重要，因为编译器现在可以正确分析 Android API 级别 26中的泛型 `findViewById` 调用：

```
val button = findViewById(R.id.button) as Button
```

智能类型转换改进

当一个变量有安全调用表达式与空检测赋值时，其智能转换现在也可以应用于安全调用接收者：

```
val firstChar = (s as? CharSequence)?.firstOrNull()
if (firstChar != null)
return s.count { it == firstChar } // s: Any 会智能转换为 CharSequence

val firstItem = (s as? Iterable<*>)?.firstOrNull()
if (firstItem != null)
return s.count { it == firstItem } // s: Any 会智能转换为 Iterable<*>
```

智能转换现在也允许用于在 lambda 表达式中局部变量，只要这些局部变量仅在 lambda 表达式之前修改即可：

```
val flag = args.size == 0
var x: String? = null
if (flag) x = "Yahoo!"

run {
    if (x != null) {
        println(x.length) // x 会智能转换为 String
    }
}
```

支持 ::foo 作为 this::foo 的简写

现在写绑定到 `this` 成员的可调用引用可以无需显式接收者，即 `::foo` 取代 `this::foo`。这也使在引用外部接收者的成员的 lambda 表达式中使用可调用引用更加方便。

破坏性变更：try 块后可靠智能转换

Kotlin 以前将 `try` 块中的赋值语句用于块后的智能转换，这可能会破坏类型安全与空安全并引发运行时故障。这个版本修复了该问题，使智能转换更加严格，但可能会破坏一些依靠这种智能转换的代码。

如果要切换到旧版智能转换行为，请传入回退标志 `-Xlegacy-smart-cast-after-try` 作为编译器参数。该参数会在 Kotlin 1.3中弃用。

弃用：数据类弃用 copy

当从已具有签名相同的 `copy` 函数的类型派生数据类时，为数据类生成的 `copy` 实现使用超类型的默认值，这导致反直觉行为，或者导致运行时失败，如果超类型中没有默认参数的话。

导致 `copy` 冲突的继承在 Kotlin 1.2 中已弃用并带有警告，而在 Kotlin 1.3中将会是错误。

弃用：枚举条目中的嵌套类型

由于初始化逻辑的问题，已弃用在枚举条目内部定义一个非 `inner class` 的嵌套类。这在 Kotlin 1.2 中会引起警告，而在 Kotlin 1.3中会成为错误。

弃用：vararg 单个具名参数

为了与注解中的数组字面值保持一致，向一个具名参数形式的 vararg 参数传入单个项目的用法（`foo(items = i)`）已被弃用。请使用伸展操作符连同相应的数组工厂函数：

```
foo(items = *intArrayOf(1))
```

在这种情况下有一项防止性能下降的优化可以消除冗余的数组创建。单参数形式在 Kotlin 1.2 中会产生警告，而在 Kotlin 1.3中会放弃。

弃用：扩展 Throwable 的泛型类的内部类

继承自 `Throwable` 的泛型类的内部类可能会在 throw-catch 场景中违反类型安全性，因此已弃用，在 Kotlin 1.2 中会是警告，而在 Kotlin 1.3中会是错误。

弃用：修改只读属性的幕后字段

通过在自定义 getter 中赋值 `field = ……` 来修改只读属性的幕后字段的用法已被弃用，在 Kotlin 1.2 中会是警告，而在 Kotlin 1.3中会是错误。

标准库

Kotlin 标准库构件与拆分包

Kotlin 标准库现在完全兼容 Java 9 的模块系统，它禁止拆分包（多个 jar 文件声明的类在同一包中）。为了支持这点，我们引入了新的 `kotlin-stdlib-jdk7` 与 `kotlin-stdlib-jdk8`，它们取代了旧版的 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8`。

在 Kotlin 看来新的构件中的声明在相同的包名内，而在 Java看来有不同的包名。因此，切换到新的构件无需修改任何源代码。

确保与新的模块系统兼容的另一处变更是在 `kotlin-reflect` 库中删除了 `kotlin.reflect` 包中弃用的声明。如果你正在使用它们，你需要切换到使用 `kotlin.reflect.full` 包中的声明，自 Kotlin 1.1 起就支持这个包了。

windowed、chunked、zipWithNext

用于 `Iterable<T>`、`Sequence<T>` 与 `CharSequence` 的新的扩展覆盖了这些应用场景：缓存或批处理（`chunked`）、滑动窗口与计算滑动均值（`windowed`）以及处理成对的后续条目（`zipWithNext`）：

```
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 }
```

fill、replaceAll、shuffle/shuffled

添加了一些用于操作列表的扩展函数：`MutableList` 的 `fill`、`replaceAll` 与 `shuffle`，以及只读 `List` 的 `shuffled`：

```
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")
```

kotlin-stdlib 中的数学运算

为满足由来已久的需求，Kotlin 1.2 添加了 JVM 与 JS 公用的用于数学运算的 `kotlin.math` API，包含以下内容：

- 常量：`PI` 与 `E`；
- 三角函数：`cos`、`sin`、`tan` 及其反函数：`acos`、`asin`、`atan`、`atan2`；
- 双曲函数：`cosh`、`sinh`、`tanh` 及其反函数：`acosh`、`asinh`、`atanh`
- 指数函数：`pow`（扩展函数）、`sqrt`、`hypot`、`exp`、`expm1`；
- 对数函数：`log`、`log2`、`log10`、`ln`、`ln1p`；
- 取整：
  - `ceil`、`floor`、`truncate`、`round`（奇进偶舍）函数；
  - `roundToInt`、`roundToLong`（四舍五入）扩展函数；
- 符号与绝对值：
  - `abs` 与 `sign` 函数；
  - `absoluteValue` 与 `sign` 扩展属性；
  - `withSign` 扩展函数；
- 两个数的最值函数：`max` 与 `min`；
- 二进制表示：
  - `ulp` 扩展属性；
  - `nextUp`、`nextDown`、`nextTowards` 扩展函数；
  - `toBits`、`toRawBits`、`Double.fromBits`（这些在 `kotlin` 包中）。

这些函数同样也有 `Float` 参数版本（但不包括常量）。

用于 BigInteger 与 BigDecimal 的操作符与转换

Kotlin 1.2 引入了一些使用 `BigInteger` 与 `BigDecimal` 运算以及由其他数字类型创建它们的函数。具体如下：

- `toBigInteger` 用于 `Int` 与 `Long`；
- `toBigDecimal` 用于 `Int`、`Long`、`Float`、`Double` 以及 `BigInteger`；
- 算术与位运算操作符函数：
  - 二元操作符 `+`、`-`、`*`、`/`、`%` 以及中缀函数 `and`、`or`、`xor`、`shl`、`shr`；
  - 一元操作符 `-`、`++`、`--` 以及函数 `inv`。

浮点数到比特的转换

添加了用于将 `Double` 及 `Float` 与其比特表示形式相互转换的函数：

- `toBits` 与 `toRawBits` 对于 `Double` 返回 `Long` 而对于 `Float` 返回 `Int`；
- `Double.fromBits` 与 `Float.fromBits` 用于有相应比特表示形式创建浮点数。

正则表达式现在可序列化

`kotlin.text.Regex` 类现在已经是 `Serializable` 的了并且可用在可序列化的继承结构中。

如果可用, Closeable.use 会调用 Throwable.addSuppressed

当在其他异常之后关闭资源期间抛出一个异常，`Closeable.use` 函数会调用 `Throwable.addSuppressed` 。

要启用这个行为，需要依赖项中有 `kotlin-stdlib-jdk7`。

JVM 后端

构造函数调用规范化

自 1.0 版起，Kotlin 就已支持带有复杂控制流的表达式，诸如 try-catch 表达式以及内联函数。根据 Java 虚拟机规范这样的代码是有效的。不幸的是，当这样的表达式出现在构造函数调用的参数中时，一些字节码处理工具不能很好地处理这种代码 。

为了缓解这种字节码处理工具用户的这一问题，我们添加了一个命令行选项（`-Xnormalize-constructor-calls=模式`），告诉编译器为这样的构造过程生成更接近 Java 的字节码 。其中 `模式` 是下列之一：

- `disable`（默认）——以与 Kotlin 1.0 即 1.1 相同的方式生成字节码；
- `enable` ——为构造函数调用生成类似 Java 的字节码。这可能会改变类加载与初始化的顺序；
- `preserve-class-initialization` ——为构造函数调用生成类似 Java 的字节码，并确保类初始化顺序得到保留。这可能会影响应用程序的整体性能；仅用在多个类之间共享一些复杂状态并在类初始化时更新的场景中。

“人工”解决办法是将具有控制流的子表达式的值存储在变量中，而不是直接在调用参数内对其求值。这与 `-Xnormalize-constructor-calls=enable` 类似。

Java 默认方法调用

在 Kotlin 1.2 之前，面向 JVM 1.6 的接口成员覆盖 Java 默认方法会产生一个关于超类型调用的警告：`Super calls to Java default methods are deprecated in JVM target 1.6. Recompile with '-jvm-target 1.8'`（“面向 JVM 1.6 的 Java 默认方法的超类型调用已弃用，请使用 ‘-jvm-target 1.8’ 重新编译”）。在 Kotlin 1.2 中，这是一个**错误 **，因此这样的代码都需要面向 JVM 1.8 编译。

破坏性变更：平台类型 x.equals(null) 的一致行为

在映射到 Java 原生类型（`Int!`、`Boolean!`、`Short!`、`Long!`、`Float!`、`Double!`、`Char!`）的平台类型上调用 `x.equals(null)`，当 `x` 为 `null` 时错误地返回了 `true`。自 Kotlin 1.2 起，在平台类型的空值上调用 `x.equals(……)` 都会**抛出 NPE**（但 `x == ...` 不会）。

要返回到 1.2 之前的行为，请将标志 `-Xno-exception-on-explicit-equals-for-boxed-null` 传给编译器。

破坏性变更：修正平台 null 透过内联扩展接收者逃逸

在平台类型的空值上调用内联扩展函数并没有检测接收者是否为 null，因而允许 null 逃逸到其他代码中。Kotlin 1.2 在调用处强制执行这项检测，如果接收者为空就抛出异常。

要切换到旧版行为，请将回退标志 `-Xno-receiver-assertions` 传给编译器。

JavaScript 后端

默认启用 TypedArray 支持

将 Kotlin 原生数组（如 `IntArray`、`DoubleArray` 等）翻译为 JavaScript 有类型数组的 JS 有类型数组支持之前是选择性加入的功能，现在已默认启用。

工具

警告作为错误

编译器现在提供一个将所有警告视为错误的选项。可在命令行中使用 `-Werror`，或者在 Gradle 中使用以下代码片段：

```
compileKotlin {
    kotlinOptions.allWarningsAsErrors = true
}
```

Kotlin 1.1 的新特性

目录

JavaScript

从 Kotlin 1.1 开始，JavaScript 目标平台不再当是实验性的。所有语言功能都支持，并且有许多新的工具用于与前端开发环境集成。更详细改动列表，请参见下文 。

协程（实验性的）

Kotlin 1.1 的关键新特性是 *协程*，它带来了 `future` / `await`、`yield` 以及类似的编程模式的支持。Kotlin 的设计中的关键特性是协程执行的实现是语言库的一部分，而不是语言的一部分，所以你不必绑定任何特定的编程范式或并发库。

协程实际上是一个轻量级的线程，可以挂起并稍后恢复。协程通过 *挂起函数* 支持：对这样的函数的调用可能会挂起协程，并启动一个新的协程，我们通常使用匿名挂起函数（即挂起 lambda 表达式）。

我们来看看在外部库 kotlinx.coroutines 中实现的 `async` / `await`：

```
// 在后台线程池中运行该代码
fun asyncOverlay() = async(CommonPool) {
    // 启动两个异步操作
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // 然后应用叠加到两个结果
    applyOverlay(original.await(), overlay.await())
}

// 在 UI 上下文中启动新的协程
launch(UI) {
    // 等待异步叠加完成
    val image = asyncOverlay().await()
    // 然后在 UI 中显示
    showImage(image)
}
```

这里，`async { …… }` 启动一个协程，当我们使用 `await()` 时，挂起协程的执行，而执行正在等待的操作，并且在等待的操作完成时恢复（可能在不同的线程上）。

标准库通过 `yield` 与 `yieldAll` 函数使用协程来支持 *惰性生成序列*。在这样的序列中，在取回每个元素之后挂起返回序列元素的代码块，并在请求下一个元素时恢复。这里有一个例子：

```
val seq = buildSequence {
  for (i in 1..5) {
      // 产生一个 i 的平方
      yield(i * i)
  }
  // 产生一个区间
  yieldAll(26..28)
}

// 输出该序列
intln(seq.toList())
```

运行上面的代码以查看结果。随意编辑它并再次运行！

更多信息请参见协程文档及教程。

请注意，协程目前还是一个**实验性的特性**，这意味着 Kotlin 团队不承诺在最终的 1.1 版本时保持该功能的向后兼容性。

其他语言特性

类型别名

类型别名允许你为现有类型定义备用名称。这对于泛型类型（如集合）以及函数类型最有用。这里有几个例子：

```
typealias OscarWinners = Map<String, String>

fun countLaLaLand(oscarWinners: OscarWinners) =
        oscarWinners.count { it.value.contains("La La Land") }

// 请注意，类型名称（初始名与类型别名）是可互换的：
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
        oscarWinners["Best picture"] == "La La Land"
```

更详细信息请参阅其 KEEP。

已绑定的可调用引用

现在可以使用 `::` 操作符来获取指向特定对象实例的方法或属性的成员引用。以前这只能用 lambda 表达式表示。这里有一个例子：

```
val numberRegex = "\\d+".toRegex()
val numbers = listOf("abc", "123", "456").filter(numberRegex::matches)
```

更详细信息请参阅其 KEEP。

密封类与数据类

Kotlin 1.1 删除了一些对 Kotlin 1.0 中已存在的密封类与数据类的限制。现在你可以在同一个文件中的任何地方定义一个密封类的子类，而不只是以作为密封类嵌套类的方式。数据类现在可以扩展其他类。这可以用来友好且清晰地定义一个表达式类的层次结构：

```
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)))
```

更详细信息请参阅其文档或者密封类及数据类的 KEEP。

lambda 表达式中的解构

现在可以使用解构声明语法来解开传递给 lambda 表达式的参数。这里有一个例子：

```
val map = mapOf(1 to "one", 2 to "two")
// 之前
println(map.mapValues { entry ->
  val (key, value) = entry
  "$key -> $value!"
})
// 现在
println(map.mapValues { (key, value) -> "$key -> $value!" })
```

更详细信息请参阅其文档及其 KEEP。

下划线用于未使用的参数

对于具有多个参数的 lambda 表达式，可以使用 `_` 字符替换不使用的参数的名称：

```
map.forEach { _, value -> println("$value!") }
```

这也适用于解构声明：

```
val (_, status) = getResult()
```

更详细信息请参阅其 KEEP。

数字字面值中的下划线

正如在 Java 8 中一样，Kotlin 现在允许在数字字面值中使用下划线来分隔数字分组：

```
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

更详细信息请参阅其 KEEP。

对于属性的更短语法

对于没有自定义访问器、或者将 getter 定义为表达式主体的属性，现在可以省略属性的类型：

```
data class Person(val name: String, val age: Int) {
val isAdult get() = age >= 20 // 属性类型推断为 “Boolean”
```

内联属性访问器

如果属性没有幕后字段，现在可以使用 `inline` 修饰符来标记该属性访问器。这些访问器的编译方式与内联函数相同。

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

你也可以将整个属性标记为 `inline` ——这样修饰符应用于两个访问器。

更详细信息请参阅其文档及其 KEEP。

局部委托属性

现在可以对局部变量使用委托属性语法。一个可能的用途是定义一个延迟求值的局部变量：

```
val answer by lazy {
    println("Calculating the answer...")
    42
}
if (needAnswer()) {                     // 返回随机值
    println("The answer is $answer.")   // 此时计算出答案
}
else {
    println("Sometimes no answer is the answer...")
}
```

更详细信息请参阅其 KEEP。

委托属性绑定的拦截

对于委托属性，现在可以使用 `provideDelegate` 操作符拦截委托到属性之间的绑定 。例如，如果我们想要在绑定之前检测属性名称，我们可以这样写：

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, property: KProperty<*>): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, property.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)
}
```

`provideDelegate` 方法在创建 `MyUI` 实例期间将会为每个属性调用，并且可以立即执行必要的验证。

更详细信息请参阅其文档。

泛型枚举值访问

现在可以用泛型的方式来对枚举类的值进行枚举：

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

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}
```

对于 DSL 中隐式接收者的作用域控制

`@DslMarker` 注解允许限制来自 DSL 上下文中的外部作用域的接收者的使用。考虑那个典型的 HTML 构建器示例：

```
table {
    tr {
        td { + "Text" }
    }
}
```

在 Kotlin 1.0 中，传递给 `td` 的 lambda 表达式中的代码可以访问三个隐式接收者：传递给 `table`、`tr` 与 `td` 的。这允许你调用在上下文中没有意义的方法——例如在 `td` 里面调用 `tr`，从而在 `<td>` 中放置一个 `<tr>` 标签。

在 Kotlin 1.1 中，你可以限制这种情况，以使只有在 `td` 的隐式接收者上定义的方法会在传给 `td` 的 lambda 表达式中可用。你可以通过定义标记有 `@DslMarker` 元注解的注解并将其应用于标记类的基类。

更详细信息请参阅其文档及其 KEEP。

`rem` 操作符

`mod` 操作符现已弃用，而使用 `rem` 取代。动机参见这个问题。

标准库

字符串到数字的转换

在 String 类中有一些新的扩展，用来将它转换为数字，而不会在无效数字上抛出异常：`String.toIntOrNull(): Int?`、`String.toDoubleOrNull(): Double?` 等。

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

还有整数转换函数，如 `Int.toString()`、`String.toInt()`、`String.toIntOrNull()`，每个都有一个带有 `radix` 参数的重载，它允许指定转换的基数（2 到 36）。

onEach()

`onEach` 是一个小、但对于集合与序列很有用的扩展函数，它允许对操作链中的集合/序列的每个元素执行一些操作，可能带有副作用。对于迭代其行为像 `forEach` 但是也进一步返回可迭代实例。对于序列它返回一个包装序列，它在元素迭代时延迟应用给定的动作。

```
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() 与 takeUnless()

这些是适用于任何接收者的三个通用扩展函数。

`also` 就像 `apply`：它接受接收者、做一些动作、并返回该接收者。二者区别是在 `apply` 内部的代码块中接收者是 `this`，而在 `also` 内部的代码块中是 `it`（并且如果你想的话，你可以给它另一个名字）。当你不想掩盖来自外部作用域的 `this` 时这很方便：

```
fun Block.copy() = Block().also {
    it.content = this.content
}
```

`takeIf` 就像单个值的 `filter`。它检测接收者是否满足该谓词，并在满足时返回该接收者否则不满足时返回 `null`。结合 elvis-操作符与及早返回，可以编写如下结构：

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?: return false
// 对现有的 outDirFile 做些事情
```

```
val index = input.indexOf(keyword).takeIf { it >= 0 } ?: error("keyword not found")
// 对输入字符串中的关键字索引做些事情，鉴于它已找到
```

`takeUnless` 与 `takeIf` 相同，只是它采用了反向谓词。当它 *不* 满足谓词时返回接收者，否则返回 `null`。因此，上面的示例之一可以用 `takeUnless` 重写如下：

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?: error("keyword not found")
```

当你有一个可调用的引用而不是 lambda 时，使用也很方便：

```
val result = string.takeUnless(String::isEmpty)
```

groupingBy()

此 API 可以用于按照键对集合进行分组，并同时折叠每个组。例如，它可以用于计算文本中字符的频率：

```
val frequencies = words.groupingBy { it.first() }.eachCount()
```

Map.toMap() 与 Map.toMutableMap()

这俩函数可以用来简易复制映射：

```
class ImmutablePropertyBag(map: Map<String, Any>) {
    private val mapCopy = map.toMap()
}
```

Map.minus(key)

运算符 `plus` 提供了一种将键值对添加到只读映射中以生成新映射的方法，但是没有一种简单的方法来做相反的操作：从映射中删除一个键采用不那么直接的方式如 `Map.filter()` 或 `Map.filterKeys()`。现在运算符 `minus` 填补了这个空白。有 4 个可用的重载：用于删除单个键、键的集合、键的序列与键的数组。

```
val map = mapOf("key" to 42)
val emptyMap = map - "key"
```

minOf() 与 maxOf()

这些函数可用于查找两个或三个给定值中的最小与最大值，其中值是原生数字或 `Comparable` 对象。每个函数还有一个重载，它接受一个额外的 `Comparator` 实例，如果你想比较自身不可比的对象的话。

```
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 })
```

类似数组的列表实例化函数

类似于 `Array` 构造函数，现在有创建 `List` 与 `MutableList` 实例的函数，并通过调用 lambda 表达式来初始化每个元素：

```
val squares = List(10) { index -> index * index }
val mutable = MutableList(10) { 0 }
```

Map.getValue()

`Map` 上的这个扩展函数返回一个与给定键相对应的现有值，或者抛出一个异常，提示找不到该键。如果该映射是用 `withDefault` 生成的，这个函数将返回默认值，而不是抛异常。

```
val map = mapOf("key" to 42)
// 返回不可空 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") // <- 这将抛出 NoSuchElementException
```

抽象集合

这些抽象类可以在实现 Kotlin 集合类时用作基类。对于实现只读集合，有 `AbstractCollection`、`AbstractList`、`AbstractSet` 与 `AbstractMap`，而对于可变集合，有 `AbstractMutableCollection`、`AbstractMutableList`、`AbstractMutableSet` 与 `AbstractMutableMap`。在 JVM 上，这些抽象可变集合从 JDK 的抽象集合继承了大部分的功能。

数组处理函数

标准库现在提供了一组用于逐个元素操作数组的函数：比较（`contentEquals` 与 `contentDeepEquals`），哈希码计算（`contentHashCode` 与 `contentDeepHashCode`），以及转换成一个字符串（`contentToString` 与 `contentDeepToString`）。它们都支持 JVM（它们作为 `java.util.Arrays` 中的相应函数的别名）与 JS（在 Kotlin 标准库中提供实现）。

```
val array = arrayOf("a", "b", "c")
println(array.toString())  // JVM 实现：类型及哈希乱码
println(array.contentToString())  // 良好格式化为列表
```

JVM 后端

Java 8 字节码支持

Kotlin 现在可以选择生成 Java 8 字节码（命令行选项 `-jvm-target 1.8` 或者 Ant/Maven/Gradle 中的相应选项）。目前这并不改变字节码的语义（特别是，接口与 lambda 表达式中的默认方法的生成与 Kotlin 1.0 中完全一样），但我们计划在以后进一步使用它。

Java 8 标准库支持

现在有支持在 Java 7 与 8 中新添加的 JDK API 的标准库的独立版本。如果你需要访问新的 API，请使用 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8` maven 构件，而不是标准的 `kotlin-stdlib`。这些构件是在 `kotlin-stdlib` 之上的微小扩展，它们将它作为传递依赖项带到项目中。

字节码中的参数名

Kotlin 现在支持在字节码中存储参数名。这可以使用命令行选项 `-java-parameters` 启用。

常量内联

编译器现在将 `const val` 属性的值内联到使用它们的位置。

可变闭包变量

用于在 lambda 表达式中捕获可变闭包变量的装箱类不再具有 volatile 字段。此更改提高了性能，但在一些罕见的使用情况下可能导致新的竞争条件。如果受此影响，你需要提供自己的同步机制来访问变量。

javax.scripting 支持

Kotlin 现在与javax.script API（JSR-223）集成。其 API 允许在运行时求值代码段：

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

关于使用 API 的示例项目参见这里 。

kotlin.reflect.full

为 Java 9 支持准备，在 `kotlin-reflect.jar` 库中的扩展函数与属性已移动到 `kotlin.reflect.full` 包中。旧包（`kotlin.reflect`）中的名称已弃用，将在 Kotlin 1.2 中删除。请注意，核心反射接口（如 `KClass`）是 Kotlin 标准库（而不是 `kotlin-reflect`）的一部分，不受移动影响。

JavaScript 后端

统一的标准库

Kotlin 标准库的大部分目前可以从代码编译成 JavaScript 来使用。特别是，关键类如集合（`ArrayList`、`HashMap` 等）、异常（`IllegalArgumentException` 等）以及其他几个关键类（`StringBuilder`、`Comparator`）现在都定义在 `kotlin` 包下。在 JVM 平台上，一些名称是相应 JDK 类的类型别名，而在 JS 平台上，这些类在 Kotlin 标准库中实现。

更好的代码生成

JavaScript 后端现在生成更加可静态检测的代码，这对 JS 代码处理工具（如 minifiers、optimisers、linters 等）更加友好。

`external` 修饰符

如果你需要以类型安全的方式在 Kotlin 中访问 JavaScript 实现的类，你可以使用 `external` 修饰符写一个 Kotlin 声明。（在 Kotlin 1.0 中，使用了 `@native` 注解。）与 JVM 目标平台不同，JS 平台允许对类与属性使用 external 修饰符。例如，可以按以下方式声明 DOM `Node` 类：

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // 等等
}
```

改进的导入处理

现在可以更精确地描述应该从 JavaScript 模块导入的声明。如果在外部声明上添加 `@JsModule("<模块名>")` 注解，它会在编译期间正确导入到模块系统（CommonJS或AMD）。例如，使用 CommonJS，该声明会通过 `require(……)` 函数导入。此外，如果要将声明作为模块或全局 JavaScript 对象导入，可以使用 `@JsNonModule` 注解。

例如，以下是将 JQuery 导入 Kotlin 模块的方法：

```
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
```

在这种情况下，JQuery 将作为名为 `jquery` 的模块导入。或者，它可以用作 $-对象，这取决于Kotlin编译器配置使用哪个模块系统。

你可以在应用程序中使用如下所示的这些声明：

```
fun main(args: Array<String>) {
    jquery(".toggle-button").click {
        jquery(".toggle-panel").toggle(300)
    }
}
```

开始

基本语法

包的定义与导入

包的声明应处于源文件顶部：

```
package my.demo

import kotlin.text.*

// ……
```

目录与包的结构无需匹配：源代码可以在文件系统的任意位置。

参见包。

程序入口点

Kotlin 应用程序的入口点是 `main` 函数。

```
fun main() {
    println("Hello world!")
}
```

函数

带有两个 `Int` 参数、返回 `Int` 的函数：

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

将表达式作为函数体、返回值类型自动推断的函数：

```
fun sum(a: Int, b: Int) = a + b
```

函数返回无意义的值：

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

`Unit` 返回类型可以省略：

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

参见函数。

变量

定义只读局部变量使用关键字 `val` 定义。只能为其赋值一次。

```
val a: Int = 1  // 立即赋值
val b = 2   // 自动推断出 `Int` 类型
val c: Int  // 如果没有初始值类型不能省略
c = 3       // 明确赋值
```

可重新赋值的变量使用 `var` 关键字：

```
var x = 5 // 自动推断出 `Int` 类型
x += 1
```

顶层变量：

```
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
```

参见属性与字段。

注释

与大多数现代语言一样，Kotlin 支持单行（或*行末*）与多行（*块*）注释。

```
// 这是一个行注释

/* 这是一个多行的
   块注释。 */
```

Kotlin 中的块注释可以嵌套。

```
/* 注释从这里开始
/* 包含嵌套的注释 */
并且在这里结束。 */
```

参见编写 Kotlin 代码文档 查看关于文档注释语法的信息。

字符串模板

```
var a = 1
// 模板中的简单名称:
val s1 = "a is $a"

a = 2
// 模板中的任意表达式:
val s2 = "${s1.replace("is", "was")}, but now is $a"
```

参见字符串模板。

条件表达式

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

在 Kotlin 中，`if` 也可以用作表达式：

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

参见`if` 表达式。

空值与 `null` 检测

当某个变量的值可以为 `null` 的时候，必须在声明处的类型后添加 `?` 来标识该引用可为空。

如果 `str` 的内容不是数字返回 `null`：

```
fun parseInt(str: String): Int? {
    // ……
}
```

使用返回可空值的函数:

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // 直接使用 `x * y` 会导致编译错误，因为它们可能为 null
    if (x != null && y != null) {
        // 在空检测后，x 与 y 会自动转换为非空值（non-nullable）
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
```

或者

```
// ……
if (x == null) {
    println("Wrong number format in arg1: '$arg1'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '$arg2'")
    return
}

// 在空检测后，x 与 y 会自动转换为非空值
println(x * y)
```

参见空安全。

类型检测与自动类型转换

`is` 运算符检测一个表达式是否某类型的一个实例。如果一个不可变的局部变量或属性已经判断出为某类型，那么检测后的分支中可以直接当作该类型使用，无需显式转换：

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` 在该条件分支内自动转换成 `String`
        return obj.length
    }

    // 在离开类型检测分支后，`obj` 仍然是 `Any` 类型
    return null
}
```

或者

```
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` 在这一分支自动转换为 `String`
    return obj.length
}
```

甚至

```
fun getStringLength(obj: Any): Int? {
    // `obj` 在 `&&` 右边自动转换成 `String` 类型
    if (obj is String && obj.length > 0) {
      return obj.length
    }

    return null
}
```

参见类以及类型转换。

for 循环

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

或者

```
val items = listOf("apple", "banana", "kiwifruit")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}
```

参见 for 循环。

while 循环

```
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}
```

参见 while 循环。

when 表达式

```
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
```

参见 when 表达式。

使用区间(range)

使用 in 运算符来检测某个数字是否在指定区间内:

```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

检测某个数字是否在指定区间外:

```
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")
}
```

区间迭代:

```
for (x in 1..5) {
    print(x)
}
```

或数列迭代:

```
for (x in 1..10 step 2) {
    print(x)
}
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

参见区间。

集合

对集合进行迭代:

```
for (item in items) {
    println(item)
}
```

使用 in 运算符来判断集合内是否包含某实例:

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

使用 lambda 表达式来过滤（filter）与映射（map）集合：

```
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
fruits
  .filter { it.startsWith("a") }
  .sortedBy { it }
  .map { it.toUpperCase() }
  .forEach { println(it) }
```

参见集合概述。

创建基本类及其实例

```
val rectangle = Rectangle(5.0, 2.0)
val triangle = Triangle(3.0, 4.0, 5.0)
```

参见类以及对象与实例。

习惯用法

一些在 Kotlin 中广泛使用的语法习惯，如果你有更喜欢的语法习惯或者风格，建一个 pull request 贡献给我们吧！

创建 DTOs（POJOs/POCOs）

```
data class Customer(val name: String, val email: String)
```

会为 `Customer` 类提供以下功能：

- 所有属性的 getters（对于 `var` 定义的还有 setters）
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- 所有属性的 `component1()`、`component2()`……等等（参见数据类）

函数的默认参数

```
fun foo(a: Int = 0, b: String = "") { …… }
```

过滤 list

```
val positives = list.filter { x -> x > 0 }
```

或者可以更短:

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

检测元素是否存在于集合中

```
if ("john@example.com" in emailsList) { …… }

if ("jane@example.com" !in emailsList) { …… }
```

字符串内插

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

类型判断

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

遍历 map/pair型list

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

`k`、`v` 可以改成任意名字。

使用区间

```
for (i in 1..100) { …… }  // 闭区间：包含 100
for (i in 1 until 100) { …… } // 半开区间：不包含 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
```

延迟属性

```
val p: String by lazy {
    // 计算该字符串
}
```

扩展函数

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

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

创建单例

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

If not null 缩写

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

println(files?.size)
```

If not null and else 缩写

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

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

if null 执行一个语句

```
val values = ……
val email = values["email"] ?: throw IllegalStateException("Email is missing!")
```

在可能会空的集合中取第一元素

```
val emails = …… // 可能会是空集合
val mainEmail = emails.firstOrNull() ?: ""
```

if not null 执行代码

```
val value = ……

value?.let {
    …… // 代码会执行到此处, 假如data不为null
}
```

映射可空值(如果非空的话)

```
val value = ……

val mapped = value?.let { transformValue(it) } ?: defaultValue
// 如果该值或其转换结果为空，那么返回 defaultValue。
```

返回 when 表达式

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

“try/catch”表达式

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // 使用 result
}
```

“if”表达式

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

返回类型为 Unit 的方法的 Builder 风格用法

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

单表达式函数

```
fun theAnswer() = 42
```

等价于

```
fun theAnswer(): Int {
    return 42
}
```

单表达式函数与其它惯用法一起使用能简化代码，例如和 when 表达式一起使用：

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

对一个对象实例调用多个方法（with）

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { // 画一个 100 像素的正方形
    penDown()
    for (i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

配置对象的属性（apply）

```
val myRectangle = Rectangle().apply {
    length = 4
    breadth = 5
    color = 0xFAFAFA
}
```

这对于配置未出现在对象构造函数中的属性非常有用。

Java 7 的 try with resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

对于需要泛型信息的泛型函数的适宜形式

```
//  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)
```

使用可空布尔

```
val b: Boolean? = ……
if (b == true) {
    ……
} else {
    // `b` 是 false 或者 null
}
```

交换两个变量

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

TODO()：将代码标记为不完整

Kotlin 的标准库有一个 `TODO()` 函数，该函数总是抛出一个 `NotImplementedError`。其返回类型为 `Nothing`，因此无论预期类型是什么都可以使用它。还有一个接受原因参数的重载：

```
fun calcTaxes(): BigDecimal = TODO("Waiting for feedback from accounting")
```

IntelliJ IDEA 的 kotlin 插件理解 `TODO()` 的语言，并且会自动在 TODO 工具窗口中添加代码指示。

编码规范

本页包含当前 Kotlin 语言的编码风格。

- 源代码组织
- 命名规则
- 格式化
- 文档注释
- 避免重复结构
- 语言特性的惯用法
- 库的编码规范

应用风格指南

如需根据本风格指南配置 IntelliJ 格式化程序，请安装 Kotlin 插件 1.2.20 或更高版本，转到 **Settings | Editor | Code Style | Kotlin**，点击右上角的 **Set from…** 链接，并从菜单中选择 **Predefined style | Kotlin style guide**。

如需验证代码已按风格指南格式化，请转到探查设置（Inspections）并启用 **Kotlin | Style issues | File is not formatted according to project settings** 探查项。验证风格指南中描述的其他问题（如命名约定）的附加探查项默认已启用。

源代码组织

目录结构

在纯 Kotlin 项目中，推荐的目录结构遵循省略了公共根包的包结构。例如，如果项目中的所有代码都位于 `org.example.kotlin` 包及其子包中，那么 `org.example.kotlin` 包的文件应该直接放在源代码根目录下，而 `org.example.kotlin.network.socket` 中的文件应该放在源代码根目录下的 `network/socket` 子目录中。

> **对于 JVM 平台**：Kotlin 源文件应当与 Java 源文件位于同一源文件根目录下，并遵循相同的目录结构（每个文件应存储在与其 package 语句对应的目录中。

源文件名称

如果 Kotlin 文件包含单个类（以及可能相关的顶层声明），那么文件名应该与该类的名称相同，并追加 .kt 扩展名。如果文件包含多个类或只包含顶层声明，那么选择一个描述该文件所包含内容的名称，并以此命名该文件。使用首字母大写的驼峰风格（例如 `ProcessDeclarations.kt`）。

文件的名称应该描述文件中代码的作用。因此，应避免在文件名中使用诸如“Util”之类的无意义词语。

源文件组织

鼓励多个声明（类、顶级函数或者属性）放在同一个 Kotlin 源文件中，只要这些声明在语义上彼此紧密关联并且文件保持合理大小（不超过几百行）。

特别是在为类定义与类的所有客户都相关的扩展函数时，请将它们放在与类自身定义相同的地方。而在定义仅对指定客户有意义的扩展函数时，请将它们放在紧挨该客户代码之后。不要只是为了保存“Foo 的所有扩展函数”而创建文件。

类布局

通常，一个类的内容按以下顺序排列：

— 属性声明与初始化块
— 次构造函数
— 方法声明
— 伴生对象

不要按字母顺序或者可见性对方法声明排序，也不要将常规方法与扩展方法分开。而是要把相关的东西放在一起，这样从上到下阅读类的人就能够跟进所发生事情的逻辑。选择一个顺序（高级别优先，或者相反）并坚持下去。

将嵌套类放在紧挨使用这些类的代码之后。如果打算在外部使用嵌套类，而且类中并没有引用这些类，那么把它们放到末尾，在伴生对象之后。

接口实现布局

在实现一个接口时，实现成员的顺序应该与该接口的成员顺序相同（如果需要，还要插入用于实现的额外的私有方法）

重载布局

在类中总是将重载放在一起。

命名规则

在 Kotlin 中，包名与类名的命名规则非常简单：

— 包的名称总是小写且不使用下划线（`org.example.project`）。通常不鼓励使用多个词的名称，但是如果确实需要使用多个词，可以将它们连接在一起或使用驼峰风格（`org.example.myProject`）。

— 类与对象的名称以大写字母开头并使用驼峰风格：

```
open class DeclarationProcessor { /*……*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /*……*/ }
```

函数名

函数、属性与局部变量的名称以小写字母开头、使用驼峰风格而不使用下划线：

```
fun processDeclarations() { /*……*/ }
var declarationCount = 1
```

例外：用于创建类实例的工厂函数可以与抽象返回类型具有相同的名称：

```
interface Foo { /*……*/ }

class FooImpl : Foo { /*……*/ }

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

测试方法的名称

当且仅当在测试中，可以使用反引号括起来的带空格的方法名。（请注意，Android 运行时目前不支持这样的方法名。）测试代码中也允许方法名使用下划线。

```
class MyTestCase {
     @Test fun `ensure everything works`() { /*...*/ }

     @Test fun ensureEverythingWorks_onAndroid() { /*...*/ }
}
```

属性名

常量名称(标有 `const` 的属性,或者保存不可变数据的没有自定义 `get` 函数的顶层/对象 `val` 属性)应该使用大写、下划线分隔的名称:

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

保存带有行为的对象或者可变数据的顶层/对象属性的名称应该使用驼峰风格名称:

```
val mutableCollection: MutableSet<String> = HashSet()
```

保存单例对象引用的属性的名称可以使用与 `object` 声明相同的命名风格:

```
val PersonComparator: Comparator<Person> = /*...*/
```

对于枚举常量,可以使用大写、下划线分隔的名称(`enum class Color { RED, GREEN }`)也可使用首字母大写的常规驼峰名称,具体取决于用途。

幕后属性的名称

如果一个类有两个概念上相同的属性,一个是公共 API 的一部分,另一个是实现细节,那么使用下划线作为私有属性名称的前缀:

```
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
         get() = _elementList
}
```

选择好名称

类的名称通常是用来解释类*是*什么的名词或者名词短语:`List`、`PersonReader`。

方法的名称通常是动词或动词短语,说明该方法*做*什么:`close`、`readPersons`。修改对象或者返回一个新对象的名称也应遵循建议。例如 `sort` 是对一个集合就地排序,而 `sorted` 是返回一个排序后的集合副本。

名称应该表明实体的目的是什么,所以最好避免在名称中使用无意义的单词(`Manager`、`Wrapper` 等)。

当使用首字母缩写作为名称的一部分时,如果缩写由两个字母组成,就将其大写(`IOStream`);而如果缩写更长一些,就只大写其首字母(`XmlFormatter`、`HttpInputStream`)。

格式化

使用 4 个空格缩进。不要使用 tab。

对于花括号,将左花括号放在结构起始处的行尾,而将右花括号放在与左括结构横向对齐的单独一行。

```
if (elements != null) {
    for (element in elements) {
        // ……
    }
}
```

（注意：在 Kotlin 中，分号是可选的，因此换行很重要。语言设计采用 Java 风格的花括号格式，如果尝试使用不同的格式化风格，那么可能会遇到意外的行为。）

横向空白

在二元操作符左右留空格（`a + b`）。例外：不要在“range to”操作符（`0..i`）左右留空格。

不要在一元运算符左右留空格（`a++`）

在控制流关键字（`if`、`when`、`for` 以及 `while`）与相应的左括号之间留空格。

不要在主构造函数声明、方法声明或者方法调用的左括号之前留空格。

```
class A(val x: Int)

fun foo(x: Int) { …… }

fun bar() {
    foo(1)
}
```

绝不在 `(`、`[` 之后或者 `]`、`)` 之前留空格。

绝不在 `.` 或者 `?.` 左右留空格：`foo.bar().filter { it > 2 }.joinToString()`，`foo?.bar()`

在 `//` 之后留一个空格：`// 这是一条注释`

不要在用于指定类型参数的尖括号前后留空格：`class Map<K, V> { …… }`

不要在 `::` 前后留空格：`Foo::class`、`String::length`

不要在用于标记可空类型的 `?` 前留空格：`String?`

作为一般规则，避免任何类型的水平对齐。将标识符重命名为不同长度的名称不应该影响声明或者任何用法的格式。

冒号

在以下场景中的 `:` 之前留一个空格：

- 当它用于分隔类型与超类型时；
- 当委托给一个超类的构造函数或者同一类的另一个构造函数时；
- 在 `object` 关键字之后。

而当分隔声明与其类型时，不要在 `:` 之前留空格。

在 `:` 之后总要留一个空格。

```
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 { /*……*/ }
}
```

类头格式化

具有少数主构造函数参数的类可以写成一行：

```
class Person(id: Int, name: String)
```

具有较长类头的类应该格式化，以使每个主构造函数参数都在带有缩进的独立的行中。另外，右括号应该位于一个新行上。如果使用了继承，那么超类的构造函数调用或者所实现接口的列表应该与右括号位于同一行：

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) { /*……*/ }
```

对于多个接口，应该将超类构造函数调用放在首位，然后将每个接口应放在不同的行中：

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker { /*……*/ }
```

对于具有很长超类型列表的类，在冒号后面换行，并横向对齐所有超类型名：

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne {

    fun foo() { /*...*/ }
}
```

为了将类头与类体分隔清楚，当类头很长时，可以在类头后放一空行（如上例所示）或者将左花括号放在独立行上：

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne
{
    fun foo() { /*...*/ }
}
```

构造函数参数使用常规缩进（4 个空格）。

> 理由：这确保了在主构造函数中声明的属性与 在类体中声明的属性具有相同的缩进。

修饰符

如果一个声明有多个修饰符，请始终按照以下顺序安放：

```
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
```

将所有注解放在修饰符前：

```
@Named("Foo")
private val foo: Foo
```

除非你在编写库，否则请省略多余的修饰符（例如 `public`）。

注解格式化

注解通常放在单独的行上，在它们所依附的声明之前，并使用相同的缩进：

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

无参数的注解可以放在同一行：

```
@JsonExclude @JvmField
var x: String
```

无参数的单个注解可以与相应的声明放在同一行：

```
@Test fun foo() { /*……*/ }
```

文件注解

文件注解位于文件注释（如果有的话）之后、`package` 语句之前，并且用一个空白行与 `package` 分开（为了强调其针对文件而不是包）。

```
/** 授权许可、版权以及任何其他内容 */
@file:JvmName("FooBar")

package foo.bar
```

函数格式化

如果函数签名不适合单行，请使用以下语法：

```
fun longMethodName(
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType
): ReturnType {
    // 函数体
}
```

函数参数使用常规缩进（4 个空格）。

> 理由：与构造函数参数一致

对于由单个表达式构成的函数体，优先使用表达式形式。

```
fun foo(): Int {     // 不良
    return 1
}

fun foo() = 1        // 良好
```

表达式函数体格式化

如果函数的表达式函数体与函数声明不适合放在同一行，那么将 `=` 留在第一行。将表达式函数体缩进 4 个空格。

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

属性格式化

对于非常简单的只读属性，请考虑单行格式：

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

对于更复杂的属性，总是将 `get` 与 `set` 关键字放在不同的行上：

```
val foo: String
    get() { /*……*/ }
```

对于具有初始化器的属性，如果初始化器很长，那么在等号后增加一个换行并将初始化器缩进四个空格：

```
private val defaultCharset: Charset? =
    EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)
```

格式化控制流语句

如果 `if` 或 `when` 语句的条件有多行，那么在语句体外边总是使用大括号。将该条件的每个后续行相对于条件语句起始处缩进 4 个空格。将该条件的右圆括号与左花括号放在单独一行：

```
if (!component.isSyncing &&
    !hasAnyKotlinRuntimeInScope(module)
) {
    return createKotlinNotConfiguredPanel(module)
}
```

> 理由：对齐整齐并且将条件与语句体分隔清楚

将 `else`、`catch`、`finally` 关键字以及 do/while 循环的 `while` 关键字与之前的花括号放在相同的行上：

```
if (condition) {
    // 主体
} else {
    // else 部分
}

try {
    // 主体
} finally {
    // 清理
}
```

在 `when` 语句中，如果一个分支不止一行，可以考虑用空行将其与相邻的分支块分开：

```
private fun parsePropertyValue(propName: String, token: Token) {
    when (token) {
        is Token.ValueToken ->
            callback.visitValue(propName, token.value)

        Token.LBRACE -> { // ……
        }
    }
}
```

将短分支放在与条件相同的行上，无需花括号。

```
when (foo) {
    true -> bar() // 良好
    false -> { baz() } // 不良
}
```

方法调用格式化

在较长参数列表的左括号后添加一个换行符。按 4 个空格缩进参数。将密切相关的多个参数分在同一行。

```
drawSquare(
    x = 10, y = 10,
    width = 100, height = 100,
    fill = true
)
```

在分隔参数名与值的 `=` 左右留空格。

链式调用换行

当对链式调用换行时，将 `.` 字符或者 `?.` 操作符放在下一行，并带有单倍缩进：

```
val anchor = owner
    ?.firstChild!!
    .siblings(forward = true)
    .dropWhile { it is PsiComment || it is PsiWhiteSpace }
```

调用链的第一个调用通常在换行之前，当然如果能让代码更有意义也可以忽略这点。

Lambda 表达式格式化

在 lambda 表达式中，应该在花括号左右以及分隔参数与代码体的箭头左右留空格。如果一个调用接受单个 lambda 表达式，应该尽可能将其放在圆括号外边传入。

```
list.filter { it > 10 }
```

如果为 lambda 表达式分配一个标签，那么不要在该标签与左花括号之间留空格：

```
fun foo() {
    ints.forEach lit@{
        // ……
    }
}
```

在多行的 lambda 表达式中声明参数名时，将参数名放在第一行，后跟箭头与换行符：

```
appendCommaSeparated(properties) { prop ->
    val propertyValue = prop.get(obj)  // ……
}
```

如果参数列表太长而无法放在一行上，请将箭头放在单独一行：

```
foo {
   context: Context,
   environment: Env
   ->
   context.configureEnv(environment)
}
```

文档注释

对于较长的文档注释，将开头 `/**` 放在一个独立行中，并且后续每行都以星号开头：

```
/**
 * 这是一条多行
 * 文档注释。
 */
```

简短注释可以放在一行内：

```
/** 这是一条简短文档注释。 */
```

通常，避免使用 `@param` 与 `@return` 标记。而是将参数与返回值的描述直接合并到文档注释中，并在提到参数的任何地方加上参数链接。只有当需要不适合放进主文本流程的冗长描述时才应使用 `@param` 与 `@return`。

```
// 避免这样：

/**
 * Returns the absolute value of the given number.
 * @param number The number to return the absolute value for.
 * @return The absolute value.
 */
fun abs(number: Int) { /*……*/ }

// 而要这样：

/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: Int) { /*……*/ }
```

避免重复结构

一般来说，如果 Kotlin 中的某种语法结构是可选的并且被 IDE 高亮为冗余的，那么应该在代码中省略之。为了清楚起见，不要在代码中保留不必要的语法元素 。

Unit

如果函数返回 Unit，那么应该省略返回类型：

```
fun foo() { // 这里省略了“: Unit”

}
```

分号

尽可能省略分号。

字符串模版

将简单变量传入到字符串模版中时不要使用花括号。只有用到更长表达式时才使用花括号。

```
println("$name has ${children.size} children")
```

语言特性的惯用法

不可变性

优先使用不可变（而不是可变）数据。初始化后未修改的局部变量与属性，总是将其声明为 `val` 而不是 `var` 。

总是使用不可变集合接口（`Collection`，`List`，`Set`，`Map`）来声明无需改变的集合。使用工厂函数创建集合实例时，尽可能选用返回不可变集合类型的函数：

```
// 不良：使用可变集合类型作为无需改变的值
fun validateValue(actualValue: String, allowedValues: HashSet<String>) { …… }

// 良好：使用不可变集合类型
fun validateValue(actualValue: String, allowedValues: Set<String>) { …… }

// 不良：arrayListOf() 返回 ArrayList<T>，这是一个可变集合类型
val allowedValues = arrayListOf("a", "b", "c")

// 良好：listOf() 返回 List<T>
val allowedValues = listOf("a", "b", "c")
```

默认参数值

优先声明带有默认参数的函数而不是声明重载函数。

```
// 不良
fun foo() = foo("a")
fun foo(a: String) { /*……*/ }

// 良好
fun foo(a: String = "a") { /*……*/ }
```

类型别名

如果有一个在代码库中多次用到的函数类型或者带有类型参数的类型，那么最好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

在简短、非嵌套的 lambda 表达式中建议使用 `it` 用法而不是显式声明参数。而在有参数的嵌套 lambda 表达式中，始终应该显式声明参数。

在 lambda 表达式中返回

避免在 lambda 表达式中使用多个返回到标签。请考虑重新组织这样的 lambda 表达式使其只有单一退出点。如果这无法做到或者不够清晰，请考虑将 lambda 表达式转换为匿名函数。

不要在 lambda 表达式的最后一条语句中使用返回到标签。

具名参数

当一个方法接受多个相同的原生类型参数或者多个 `Boolean` 类型参数时，请使用具名参数语法，除非在上下文中的所有参数的含义都已绝对清楚。

```
drawSquare(x = 10, y = 10, width = 100, height = 100, fill = true)
```

使用条件语句

优先使用 `try`、`if` 与 `when` 的表达形式。例如：

```
return if (x) foo() else bar()

return when(x) {
    0 -> "zero"
    else -> "nonzero"
}
```

优先选用上述代码而不是：

```
if (x)
    return foo()
else
    return bar()

when(x) {
    0 -> return "zero"
    else -> return "nonzero"
}
```

`if` 还是 `when`

二元条件优先使用 `if` 而不是 `when`。不要使用

```
when (x) {
    null -> // ……
    else -> // ……
}
```

而应使用 `if (x == null) …… else ……`

如果有三个或多个选项时优先使用 `when`。

在条件中使用可空的 `Boolean` 值

如果需要在条件语句中用到可空的 `Boolean`，使用 `if (value == true)` 或 `if (value == false)` 检测。

使用循环

优先使用高阶函数（`filter`、`map` 等）而不是循环。例外：`forEach`（优先使用常规的 `for` 循环，除非 `forEach` 的接收者是可空的或者 `forEach` 用做长调用链的一部分。）

当在使用多个高阶函数的复杂表达式与循环之间进行选择时，请了解每种情况下所执行操作的开销并且记得考虑性能因素。

区间上循环

使用 `until` 函数在一个开区间上循环：

```
for (i in 0..n - 1) { /*……*/ }  // 不良
for (i in 0 until n) { /*……*/ }  // 良好
```

使用字符串

优先使用字符串模板而不是字符串拼接。

优先使用多行字符串而不是将 `\n` 转义序列嵌入到常规字符串字面值中。

如需在多行字符串中维护缩进，当生成的字符串不需要任何内部缩进时使用 `trimIndent`，而需要内部缩进时使用 `trimMargin`：

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

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

函数还是属性

在某些情况下，不带参数的函数可与只读属性互换。虽然语义相似，但是在某种程度上有一些风格上的约定。

底层算法优先使用属性而不是函数：

- 不会抛异常
- 计算开销小（或者在首次运行时缓存）
- 如果对象状态没有改变，那么多次调用都会返回相同结果

使用扩展函数

放手去用扩展函数。每当你有一个主要用于某个对象的函数时，可以考虑使其成为一个以该对象为接收者的扩展函数。为了尽量减少 API 污染，尽可能地限制扩展函数的可见性。根据需要，使用局部扩展函数、成员扩展函数或者具有私有可视性的顶层扩展函数。

使用中缀函数

一个函数只有用于两个角色类似的对象时才将其声明为中缀函数。良好示例如：`and`、`to`、`zip`。不良示例如：`add`。

如果一个方法会改动其接收者，那么不要声明为中缀形式。

工厂函数

如果为一个类声明一个工厂函数，那么不要让它与类自身同名。优先使用独特的名称，该名称能表明为何该工厂函数的行为与众不同。只有当确实没有特殊的语义时，才可以使用与该类相同的名称。

例如：

```
class Point(val x: Double, val y: Double) {
    companion object {
        fun fromPolar(angle: Double, radius: Double) = Point(...)
    }
}
```

如果一个对象有多个重载的构造函数，它们并非调用不同的超类构造函数，并且不能简化为具有默认参数值的单个构造函数，那么优先用工厂函数取代这些重载的构造函数。

平台类型

返回平台类型表达式的公有函数/方法必须显式声明其 Kotlin 类型：

```
fun apiCall(): String = MyJavaApi.getProperty("name")
```

任何使用平台类型表达式初始化的属性（包级别或类级别）必须显式声明其 Kotlin 类型：

```
class Person {
    val name: String = MyJavaApi.getProperty("name")
}
```

使用平台类型表达式初始化的局部值可以有也可以没有类型声明：

```
fun main() {
    val name = MyJavaApi.getProperty("name")
    println(name)
}
```

使用作用域函数 apply/with/run/also/let

Kotlin 提供了一系列用来在给定对象上下文中执行代码块的函数：`let`、`run`、`with`、`apply` 以及 `also`。关于不同情况下选择正确作用域函数的准则，请参考作用域函数。

库的编码规范

在编写库时，建议遵循一组额外的规则以确保 API 的稳定性：

— 总是显式指定成员的可见性（以避免将声明意外暴露为公有 API）

— 总是显式指定函数返回类型以及属性类型（以避免当实现改变时意外更改返回类型）

— 为所有公有成员提供 KDoc 注释，不需要任何新文档的覆盖成员除外（以支持为该库生成文档）

基础

基本类型

在 Kotlin 中，所有东西都是对象，在这个意义上讲我们可以在任何变量上调用成员函数与属性。一些类型可以有特殊的内部表示——例如，数字、字符以及布尔值可以在运行时表示为原生类型值，但是对于用户来说，它们看起来就像普通的类。在本节中，我们会描述 Kotlin 中使用的基本类型：数字、字符、布尔值、数组与字符串。

数字

Kotlin 提供了一组表示数字的内置类型。对于整数，有四种不同大小的类型，因此值的范围也不同。

类型	大小（比特数）	最小值	最大值
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
Long	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

所有以未超出 `Int` 最大值的整型值初始化的变量都会推断为 `Int` 类型。如果初始值超过了其最大值，那么推断为 `Long` 类型。如需显式指定 `Long` 型值，请在该值后追加 `L` 后缀。

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

对于浮点数，Kotlin 提供了 `Float` 与 `Double` 类型。根据 IEEE 754 标准，两种浮点类型的*十进制位数*（即可以存储多少位十进制数）不同。`Float` 反映了 IEEE 754 *单精度*，而 `Double` 提供了*双精度*。

类型	大小（比特数）	有效数字比特数	指数比特数	十进制位数
Float	32	24	8	6-7
Double	64	53	11	15-16

对于以小数初始化的变量，编译器会推断为 `Double` 类型。如需将一个值显式指定为 `Float` 类型，请添加 `f` 或 `F` 后缀。如果这样的值包含多于 6～7 位十进制数，那么会将其舍入。

```
val pi = 3.14 // Double
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float, 实际值为 2.7182817
```

请注意，与一些其他语言不同，Kotlin 中的数字没有隐式拓宽转换。例如，具有 `Double` 参数的函数只能对 `Double` 值调用，而不能对 `Float`、`Int` 或者其他数字值调用。

```
fun main() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.1
    val f = 1.1f

    printDouble(d)
//    printDouble(i) // 错误: 类型不匹配
//    printDouble(f) // 错误: 类型不匹配
}
```

如需将数值转换为不同的类型，请使用显示转换。

字面常量

数值常量字面值有以下几种:

- 十进制: `123`
  - Long 类型用大写 `L` 标记: `123L`
- 十六进制: `0x0F`
- 二进制: `0b00001011`

注意: 不支持八进制

Kotlin 同样支持浮点数的常规表示方法:

- 默认 double：`123.5`、`123.5e10`
- Float 用 `f` 或者 `F` 标记: `123.5f`

数字字面值中的下划线（自 1.1 起）

你可以使用下划线使数字常量更易读：

```
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
```

表示方式

在 Java 平台数字是物理存储为 JVM 的原生类型，除非我们需要一个可空的引用（如 `Int?`）或泛型。后者情况下会把数字装箱。

注意数字装箱不一定保留同一性:

```
val a: Int = 100
val boxedA: Int? = a
val anotherBoxedA: Int? = a

val b: Int = 10000
val boxedB: Int? = b
val anotherBoxedB: Int? = b

println(boxedA === anotherBoxedA) // true
println(boxedB === anotherBoxedB) // false
```

另一方面，它保留了相等性：

```
val a: Int = 10000
println(a == a) // 输出“true”
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA == anotherBoxedA) // 输出“true”
```

显式转换

由于不同的表示方式，较小类型并不是较大类型的子类型。如果它们是的话，就会出现下述问题：

```
// 假想的代码，实际上并不能编译：
val a: Int? = 1 // 一个装箱的 Int (java.lang.Integer)
val b: Long? = a // 隐式转换产生一个装箱的 Long (java.lang.Long)
print(b == a) // 惊！这将输出“false”鉴于 Long 的 equals() 会检测另一个是否也为 Long
```

所以相等性会在所有地方悄无声息地失去，更别说同一性了。

因此较小的类型**不能**隐式转换为较大的类型。这意味着在不进行显式转换的情况下我们不能把 `Byte` 型值赋给一个 `Int` 变量。

```
val b: Byte = 1 // OK, 字面值是静态检测的
val i: Int = b // 错误
```

我们可以显式转换来拓宽数字

```
val i: Int = b.toInt() // OK：显式拓宽
print(i)
```

每个数字类型支持如下的转换：

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

缺乏隐式类型转换很少会引起注意，因为类型会从上下文推断出来，而算术运算会有重载做适当转换，例如：

```
val l = 1L + 3 // Long + Int => Long
```

运算

Kotlin支持数字运算的标准集（`+` `-` `*` `/` `%`），运算被定义为相应的类成员（但编译器会将函数调用优化为相应的指令）。参见运算符重载。

整数除法

请注意，整数间的除法总是返回整数。会丢弃任何小数部分。例如：

```
val x = 5 / 2
//println(x == 2.5) // ERROR: Operator '==' cannot be applied to 'Int' and 'Double'
println(x == 2)
```

对于任何两个整数类型之间的除法来说都是如此。

```
val x = 5L / 2
println(x == 2L)
```

如需返回浮点类型，请将其中的一个参数显式转换为浮点类型。

```
val x = 5 / 2.toDouble()
println(x == 2.5)
```

位运算

对于位运算，没有特殊字符来表示，而只可用中缀方式调用具名函数，例如：

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

这是完整的位运算列表（只用于 `Int` 与 `Long`）：

- `shl(bits)` – 有符号左移
- `shr(bits)` – 有符号右移
- `ushr(bits)` – 无符号右移
- `and(bits)` – 位**与**
- `or(bits)` – 位**或**
- `xor(bits)` – 位**异或**
- `inv()` – 位非

浮点数比较

本节讨论的浮点数操作如下：

- 相等性检测：`a == b` 与 `a != b`
- 比较操作符：`a < b`、`a > b`、`a <= b`、`a >= b`
- 区间实例以及区间检测：`a..b`、`x in a..b`、`x !in a..b`

当其中的操作数 `a` 与 `b` 都是静态已知的 `Float` 或 `Double` 或者它们对应的可空类型（声明为该类型，或者推断为该类型，或者智能类型转换的结果是该类型），两数字所形成的操作或者区间遵循 IEEE 754 浮点运算标准。

然而，为了支持泛型场景并提供全序支持，当这些操作数**并非**静态类型为浮点数（例如是 `Any`、`Comparable<……>`、类型参数）时，这些操作使用为 `Float` 与 `Double` 实现的不符合标准的 `equals` 与 `compareTo`，这会出现：

— 认为 `NaN` 与其自身相等

— 认为 `NaN` 比包括正无穷大（`POSITIVE_INFINITY`）在内的任何其他元素都大

— 认为 `-0.0` 小于 `0.0`

字符

字符用 `Char` 类型表示。它们不能直接当作数字

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

字符字面值用单引号括起来: `'1'`。特殊字符可以用反斜杠转义。支持这几个转义序列: `\t`、`\b`、`\n`、`\r`、`\'`、`\"`、`\\` 与 `\$`。编码其他字符要用 Unicode 转义序列语法: `'\uFF00'`。

我们可以显式把字符转换为 `Int` 数字:

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

当需要可空引用时，像数字、字符会被装箱。装箱操作不会保留同一性。

布尔

布尔用 `Boolean` 类型表示，它有两个值: `true` 与 `false`。

若需要可空引用布尔会被装箱。

内置的布尔运算有:

— `||` – 短路逻辑或

— `&&` – 短路逻辑与

— `!` - 逻辑非

数组

数组在 Kotlin 中使用 `Array` 类来表示，它定义了 `get` 与 `set` 函数（按照运算符重载约定这会转变为 `[]`）以及 `size` 属性，以及一些其他有用的成员函数:

```
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>
    // ……
}
```

我们可以使用库函数 `arrayOf()` 来创建一个数组并传递元素值给它，这样 `arrayOf(1, 2, 3)` 创建了 array `[1, 2, 3]`。或者，库函数 `arrayOfNulls()` 可以用于创建一个指定大小的、所有元素都为空的数组。

另一个选项是用接受数组大小以及一个函数参数的 `Array` 构造函数，用作参数的函数能够返回给定索引的每个元素初始值：

```
// 创建一个 Array<String> 初始化为 ["0", "1", "4", "9", "16"]
val asc = Array(5) { i -> (i * i).toString() }
asc.forEach { println(it) }
```

如上所述，`[]` 运算符代表调用成员函数 `get()` 与 `set()`。

Kotlin 中数组是*不型变的 (invariant)*。这意味着 Kotlin 不让我们把 `Array<String>` 赋值给 `Array<Any>`，以防止可能的运行时失败（但是你可以使用 `Array<out Any>`，参见类型投影）。

原生类型数组

Kotlin 也有无装箱开销的专门的类来表示原生类型数组: `ByteArray`、`ShortArray`、`IntArray` 等等。这些类与 `Array` 并没有继承关系，但是它们有同样的方法属性集。它们也都有相应的工厂方法:

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

```
// 大小为 5、值为 [0, 0, 0, 0, 0] 的整型数组
val arr = IntArray(5)

// 例如：用常量初始化数组中的值
// 大小为 5、值为 [42, 42, 42, 42, 42] 的整型数组
val arr = IntArray(5) { 42 }

// 例如：使用 lambda 表达式初始化数组中的值
// 大小为 5、值为 [0, 1, 2, 3, 4] 的整型数组（值初始化为其索引值）
var arr = IntArray(5) { it * 1 }
```

无符号整型

> 无符号类型自 Kotlin 1.3 起才可用，并且目前是*实验性的*。详见下文

Kotlin 为无符号整数引入了以下类型：

- `kotlin.UByte`：无符号 8 比特整数，范围是 0 到 255
- `kotlin.UShort`：无符号 16 比特整数，范围是 0 到 65535
- `kotlin.UInt`：无符号 32 比特整数，范围是 0 到 2^32 - 1
- `kotlin.ULong`：无符号 64 比特整数，范围是 0 到 2^64 - 1

无符号类型支持其对应有符号类型的大多数操作。

> 请注意，将类型从无符号类型更改为对应的有符号类型（反之亦然）是*二进制不兼容*变更

无符号类型是使用另一个实验性特性（即内联类）实现的。

特化的类

与原生类型相同，每个无符号类型都有相应的为该类型特化的表示数组的类型：

— `kotlin.UByteArray`：无符号字节数组

— `kotlin.UShortArray`：无符号短整型数组

— `kotlin.UIntArray`：无符号整型数组

— `kotlin.ULongArray`：无符号长整型数组

与有符号整型数组一样，它们提供了类似于 `Array` 类的 API 而没有装箱开销。

此外，区间与数列也支持 `UInt` 与 `ULong`（通过这些类 `kotlin.ranges.UIntRange`、`kotlin.ranges.UIntProgression`、`kotlin.ranges.ULongRange`、`kotlin.ranges.ULongProgression`）

字面值

为使无符号整型更易于使用，Kotlin 提供了用后缀标记整型字面值来表示指定无符号类型（类似于 Float/Long）：

— 后缀 `u` 与 `U` 将字面值标记为无符号。确切类型会根据预期类型确定。如果没有提供预期的类型，会根据字面值大小选择 `UInt` 或者 `ULong`

```
val b: UByte = 1u  // UByte，已提供预期类型
val s: UShort = 1u // UShort，已提供预期类型
val l: ULong = 1u  // ULong，已提供预期类型

val a1 = 42u // UInt：未提供预期类型，常量适于 UInt
val a2 = 0xFFFF_FFFF_FFFFu // ULong：未提供预期类型，常量不适于 UInt
```

— 后缀 `uL` 与 `UL` 显式将字面值标记为无符号长整型。

```
val a = 1UL // ULong，即使未提供预期类型并且常量适于 UInt
```

无符号整型的实验性状态

无符号类型的设计是实验性的，这意味着这个特性改进很快并且没有给出兼容性保证。当在 Kotlin 1.3+ 中使用无符号算术时，会报出警告表明这个特性是实验性的。如需移除警告，必须选择加入（opt-in）无符号类型的实验性使用。

选择加入无符号整型有两种可行的方式：将 API 标记为实验性的，或者无需标记。

— 如需传播实验性，请以 `@ExperimentalUnsignedTypes` 标注使用了无符号整型的声明。

— 如需选择加入而不传播实验性，要么使用 `@OptIn(ExperimentalUnsignedTypes::class)` 注解标注声明，要么将 `-Xopt-in=kotlin.ExperimentalUnsignedTypes` 传给编译器。

你的客户是否必须选择使用你的 API 取决于你，不过请记住，无符号整型是一个实验性特性，因此使用它们的 API 可能会因语言的变更而发生突然破坏。

技术细节也参见实验性 API KEEP。

深入探讨

关于技术细节与深入探讨请参见无符号类型的语言提案。

字符串

字符串用 `String` 类型表示。字符串是不可变的。字符串的元素——字符可以使用索引运算符访问: `s[i]`。可以用 `for` 循环迭代字符串:

```
for (c in str) {
    println(c)
}
```

可以用 `+` 操作符连接字符串。这也适用于连接字符串与其他类型的值，只要表达式中的第一个元素是字符串：

```
val s = "abc" + 1
println(s + "def")
```

请注意，在大多数情况下，优先使用字符串模板或原始字符串而不是字符串连接。

字符串字面值

Kotlin 有两种类型的字符串字面值: 转义字符串可以有转义字符，以及原始字符串可以包含换行以及任意文本。以下是转义字符串的一个示例:

```
val s = "Hello, world!\n"
```

转义采用传统的反斜杠方式。参见上面的 字符 查看支持的转义序列。

原始字符串 使用三个引号（`"""`）分界符括起来，内部没有转义并且可以包含换行以及任何其他字符:

```
val text = """
    for (c in "foo")
        print(c)
"""
```

你可以通过 `trimMargin()` 函数去除前导空格：

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

默认 `|` 用作边界前缀，但你可以选择其他字符并作为参数传入，比如 `trimMargin(">")`。

字符串模板

字符串字面值可以包含 *模板表达式*，即一些小段代码，会求值并把结果合并到字符串中。模板表达式以美元符（`$`）开头，由一个简单的名字构成:

```
val i = 10
println("i = $i") // 输出“i = 10”
```

或者用花括号括起来的任意表达式:

```
val s = "abc"
println("$s.length is ${s.length}") // 输出“abc.length is 3”
```

原始字符串与转义字符串内部都支持模板。如果你需要在原始字符串中表示字面值 `$` 字符（它不支持反斜杠转义），你可以用下列语法：

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

包

源文件通常以包声明开头:

```
package org.example

fun printMessage() { /*……*/ }
class Message { /*……*/ }

// ……
```

源文件所有内容(无论是类还是函数)都包含在声明的包内。所以上例中 `printMessage()` 的全名是 `org.example.printMessage`,而 `Message` 的全名是 `org.example.Message`。

如果没有指明包,该文件的内容属于无名字的默认包。

默认导入

有多个包会默认导入到每个 Kotlin 文件中:

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.* (自 1.1 起)
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

根据目标平台还会导入额外的包:

- JVM:
  - java.lang.*
  - kotlin.jvm.*
- JS:
  - kotlin.js.*

导入

除了默认导入之外,每个文件可以包含它自己的导入指令。导入语法在语法中讲述。

可以导入一个单独的名字,如.

```
import org.example.Message // 现在 Message 可以不用限定符访问
```

也可以导入一个作用域下的所有内容(包、类、对象等):

```
import org.example.* // “org.example”中的一切都可访问
```

如果出现名字冲突,可以使用 `as` 关键字在本地重命名冲突项来消歧义:

```
import org.example.Message // Message 可访问
import org.test.Message as testMessage // testMessage 代表“org.test.Message”
```

关键字 `import` 并不仅限于导入类;也可用它来导入其他声明:

— 顶层函数及属性;

— 在对象声明中声明的函数和属性;

— 枚举常量。

顶层声明的可见性

如果顶层声明是 `private` 的,它是声明它的文件所私有的(参见 可见性修饰符)。

控制流：if、when、for、while

If 表达式

在 Kotlin 中，if是一个表达式，即它会返回一个值。因此就不需要三元运算符（条件 ? 然后 : 否则），因为普通的 if 就能胜任这个角色。

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

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

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

if 的分支可以是代码块，最后的表达式作为该块的值：

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

如果你使用 if 作为表达式而不是语句（例如：返回它的值或者把它赋给变量），该表达式需要有 `else` 分支。

参见 if 语法。

When 表达式

when 取代了类 C 语言的 switch 操作符。其最简单的形式如下：

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // 注意这个块
        print("x is neither 1 nor 2")
    }
}
```

when 将它的参数与所有的分支条件顺序比较，直到某个分支满足条件。when 既可以被当做表达式使用也可以被当做语句使用。如果它被当做表达式，符合条件的分支的值就是整个表达式的值，如果当做语句使用，则忽略个别分支的值。（像 if 一样，每一个分支可以是一个代码块，它的值是块中最后的表达式的值。）

如果其他分支都不满足条件将会求值 else 分支。如果 when 作为一个表达式使用，则必须有 else 分支，除非编译器能够检测出所有的可能情况都已经覆盖了 [例如，对于 枚举（enum）类条目与密封（sealed）类子类型]。

如果很多分支需要用相同的方式处理，则可以把多个分支条件放在一起，用逗号分隔：

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

我们可以用任意表达式(而不只是常量)作为分支条件

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

我们也可以检测一个值在(in)或者不在(!in)一个区间或者集合中:

```
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")
}
```

另一种可能性是检测一个值是(is)或者不是(!is)一个特定类型的值。注意:由于智能转换,你可以访问该类型的方法与属性而无需任何额外的检测。

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

when 也可以用来取代 if-else if链。如果不提供参数,所有的分支条件都是简单的布尔表达式,而当一个分支的条件为真时则执行该分支:

```
when {
    x.isOdd() -> print("x is odd")
    y.isEven() -> print("y is even")
    else -> print("x+y is even.")
}
```

自 Kotlin 1.3 起,可以使用以下语法将 when 的主语(subject,译注:指 `when` 所判断的表达式)捕获到变量中:

```
fun Request.getBody() =
        when (val response = executeRequest()) {
            is Success -> response.body
            is HttpError -> throw HttpException(response.status)
        }
```

在 when 主语中引入的变量的作用域仅限于 when 主体。

参见 when 语法。

For 循环

for 循环可以对任何提供迭代器(iterator)的对象进行遍历,这相当于像 C# 这样的语言中的 `foreach` 循环。语法如下:

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

循环体可以是一个代码块。

```
for (item: Int in ints) {
    // ……
}
```

如上所述，`for` 可以循环遍历任何提供了迭代器的对象。即：

- 有一个成员函数或者扩展函数 `iterator()`，它的返回类型
  - 有一个成员函数或者扩展函数 `next()`，并且
  - 有一个成员函数或者扩展函数 `hasNext()` 返回 `Boolean`。

这三个函数都需要标记为 `operator`。

如需在数字区间上迭代，请使用区间表达式：

```
for (i in 1..3) {
    println(i)
}
for (i in 6 downTo 0 step 2) {
    println(i)
}
```

对区间或者数组的 `for` 循环会被编译为并不创建迭代器的基于索引的循环。

如果你想要通过索引遍历一个数组或者一个 list，你可以这么做：

```
for (i in array.indices) {
    println(array[i])
}
```

或者你可以用库函数 `withIndex`：

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

参见 `for` 语法。

While 循环

`while` 与 `do`..`while` 照常使用

```
while (x > 0) {
    x--
}

do {
  val y = retrieveData()
} while (y != null) // y 在此处可见
```

参见 `while` 语法.

循环中的 Break 与 continue

在循环中 Kotlin 支持传统的 `break` 与 `continue` 操作符。参见返回与跳转。

返回和跳转

Kotlin 有三种结构化跳转表达式：

- `return`。默认从最直接包围它的函数或者匿名函数返回。
- `break`。终止最直接包围它的循环。
- `continue`。继续下一次最直接包围它的循环。

所有这些表达式都可以用作更大表达式的一部分：

```
val s = person.name ?: return
```

这些表达式的类型是 Nothing 类型。

Break 与 Continue 标签

在 Kotlin 中任何表达式都可以用标签（`label`）来标记。标签的格式为标识符后跟 `@` 符号，例如：`abc@`、`fooBar@` 都是有效的标签（参见语法）。要为一个表达式加标签，我们只要在其前加标签即可。

```
loop@ for (i in 1..100) {
    // ……
}
```

现在，我们可以用标签限制 `break` 或者`continue`：

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (……) break@loop
    }
}
```

标签限制的 break 跳转到刚好位于该标签指定的循环后面的执行点。`continue` 继续标签指定的循环的下一次迭代。

返回到标签

Kotlin 有函数字面量、局部函数和对象表达式。因此 Kotlin 的函数可以被嵌套。标签限制的 `return` 允许我们从外层函数返回。最重要的一个用途就是从 lambda 表达式中返回。回想一下我们这么写的时候：

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return // 非局部直接返回到 foo() 的调用者
        print(it)
    }
    println("this point is unreachable")
}
```

这个 `return` 表达式从最直接包围它的函数即 `foo` 中返回。（注意，这种非局部的返回只支持传给内联函数的 lambda 表达式。）如果我们需要从 lambda 表达式中返回，我们必须给它加标签并用以限制 `return`。

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit // 局部返回到该 lambda 表达式的调用者，即 forEach 循环
        print(it)
    }
    print(" done with explicit label")
}
```

现在，它只会从 lambda 表达式中返回。通常情况下使用隐式标签更方便。该标签与接受该 lambda 的函数同名。

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return@forEach // 局部返回到该 lambda 表达式的调用者，即 forEach 循环
        print(it)
    }
    print(" done with implicit label")
}
```

或者，我们用一个匿名函数替代 lambda 表达式。匿名函数内部的 `return` 语句将从该匿名函数自身返回

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach(fun(value: Int) {
        if (value == 3) return  // 局部返回到匿名函数的调用者，即 forEach 循环
        print(value)
    })
    print(" done with anonymous function")
}
```

请注意，前文三个示例中使用的局部返回类似于在常规循环中使用 `continue`。并没有 `break` 的直接等价形式，不过可以通过增加另一层嵌套 lambda 表达式并从其中非局部返回来模拟：

```
fun foo() {
    run loop@{
        listOf(1, 2, 3, 4, 5).forEach {
            if (it == 3) return@loop // 从传入 run 的 lambda 表达式非局部返回
            print(it)
        }
    }
    print(" done with nested loop")
}
```

当要返一个回值的时候，解析器优先选用标签限制的 return，即

```
return@a 1
```

意为“返回 `1` 到 `@a`”，而不是“返回一个标签标注的表达式 `(@a 1)`”。

类与对象

类与继承

类

Kotlin 中使用关键字 `class` 声明类

```
class Invoice { /*……*/ }
```

类声明由类名、类头（指定其类型参数、主构造函数等）以及由花括号包围的类体构成。类头与类体都是可选的；如果一个类没有类体，可以省略花括号。

```
class Empty
```

构造函数

在 Kotlin 中的一个类可以有一个**主构造函数**以及一个或多个**次构造函数**。主构造函数是类头的一部分：它跟在类名（与可选的类型参数）后。

```
class Person constructor(firstName: String) { /*……*/ }
```

如果主构造函数没有任何注解或者可见性修饰符，可以省略这个 `constructor` 关键字。

```
class Person(firstName: String) { /*……*/ }
```

主构造函数不能包含任何的代码。初始化的代码可以放到以 `init` 关键字作为前缀的**初始化块（initializer blocks）**中。

在实例初始化期间，初始化块按照它们出现在类体中的顺序执行，与属性初始化器交织在一起：

```
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}")
    }
}
```

请注意，主构造的参数可以在初始化块中使用。它们也可以在类体内声明的属性初始化器中使用：

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

事实上，声明属性以及从主构造函数初始化属性，Kotlin 有简洁的语法：

```
class Person(val firstName: String, val lastName: String, var age: Int) { /*……*/ }
```

与普通属性一样，主构造函数中声明的属性可以是可变的（var）或只读的（val）。

如果构造函数有注解或可见性修饰符，这个 constructor 关键字是必需的，并且这些修饰符在它前面：

```
class Customer public @Inject constructor(name: String) { /*……*/ }
```

更多详情，参见可见性修饰符

次构造函数

类也可以声明前缀有 constructor的**次构造函数**：

```
class Person {
    var children: MutableList<Person> = mutableListOf<>()
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

如果类有一个主构造函数，每个次构造函数需要委托给主构造函数，可以直接委托或者通过别的次构造函数间接委托。委托到同一个类的另一个构造函数用 this 关键字即可：

```
class Person(val name: String) {
    var children: MutableList<Person> = mutableListOf<>()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

请注意，初始化块中的代码实际上会成为主构造函数的一部分。委托给主构造函数会作为次构造函数的第一条语句，因此所有初始化块与属性初始化器中的代码都会在次构造函数体之前执行。即使该类没有主构造函数，这种委托仍会隐式发生，并且仍会执行初始化块：

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor")
    }
}
```

如果一个非抽象类没有声明任何（主或次）构造函数，它会有一个生成的不带参数的主构造函数。构造函数的可见性是 public。如果你不希望你的类有一个公有构造函数，你需要声明一个带有非默认可见性的空的主构造函数：

```
class DontCreateMe private constructor () { /*……*/ }
```

注意：在 JVM 上，如果主构造函数的所有的参数都有默认值，编译器会生成 一个额外的无参构造函数，它将使用默认值。这使得 Kotlin 更易于使用像 Jackson 或者 JPA 这样的通过无参构造函数创建类的实例的库。

```
class Customer(val customerName: String = "")
```

创建类的实例

要创建一个类的实例，我们就像普通函数一样调用构造函数：

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

注意 Kotlin 并没有 new 关键字。

创建嵌套类、内部类与匿名内部类的类实例在嵌套类中有述。

类成员

类可以包含：

- 构造函数与初始化块
- 函数
- 属性
- 嵌套类与内部类
- 对象声明

继承

在 Kotlin 中所有类都有一个共同的超类 `Any`，这对于没有超类型声明的类是默认超类：

```
class Example // 从 Any 隐式继承
```

`Any` 有三个方法：`equals()`、`hashCode()` 与 `toString()`。因此，为所有 Kotlin 类都定义了这些方法。

默认情况下，Kotlin 类是最终（final）的：它们不能被继承。要使一个类可继承，请用 `open` 关键字标记它。

```
open class Base // 该类开放继承
```

如需声明一个显式的超类型，请在类头中把超类型放到冒号之后：

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

如果派生类有一个主构造函数，其基类可以（并且必须）用派生类主构造函数的参数就地初始化。

如果派生类没有主构造函数，那么每个次构造函数必须使用 super 关键字初始化其基类型，或委托给另一个构造函数做到这一点。注意，在这种情况下，不同的次构造函数可以调用基类型的不同的构造函数：

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

覆盖方法

我们之前提到过，Kotlin 力求清晰显式。因此，Kotlin 对于可覆盖的成员（我们称之为*开放*）以及覆盖后的成员需要显式修饰符：

```
open class Shape {
    open fun draw() { /*……*/ }
    fun fill() { /*……*/ }
}

class Circle() : Shape() {
    override fun draw() { /*……*/ }
}
```

`Circle.draw()` 函数上必须加上 `override` 修饰符。如果没写，编译器将会报错。如果函数没有标注 `open` 如 `Shape.fill()`，那么子类中不允许定义相同签名的函数，不论加不加 **override**。将 `open` 修饰符添加到 final 类（即没有 `open` 的类）的成员上不起作用。

标记为 `override` 的成员本身是开放的，也就是说，它可以在子类中覆盖。如果你想禁止再次覆盖，使用 `final` 关键字：

```
open class Rectangle() : Shape() {
    final override fun draw() { /*……*/ }
}
```

覆盖属性

属性覆盖与方法覆盖类似；在超类中声明然后在派生类中重新声明的属性必须以 `override` 开头，并且它们必须具有兼容的类型。每个声明的属性可以由具有初始化器的属性或者具有 `get` 方法的属性覆盖。

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
    override val vertexCount = 4
}
```

你也可以用一个 `var` 属性覆盖一个 `val` 属性，但反之则不行。这是允许的，因为一个 `val` 属性本质上声明了一个 `get` 方法，而将其覆盖为 `var` 只是在子类中额外声明一个 `set` 方法。

请注意，你可以在主构造函数中使用 `override` 关键字作为属性声明的一部分。

```
interface Shape {
    val vertexCount: Int
}

class Rectangle(override val vertexCount: Int = 4) : Shape // 总是有 4 个顶点

class Polygon : Shape {
    override var vertexCount: Int = 0  // 以后可以设置为任何数
}
```

派生类初始化顺序

在构造派生类的新实例的过程中，第一步完成其基类的初始化（在之前只有对基类构造函数参数的求值），因此发生在派生类的初始化逻辑运行之前。

```
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") }
}
```

这意味着，基类构造函数执行时，派生类中声明或覆盖的属性都还没有初始化。如果在基类初始化逻辑中（直接或通过另一个覆盖的 `open` 成员的实现间接）使用了任何一个这种属性，那么都可能导致不正确的行为或运行时故障。设计一个基类时，应该避免在构造函数、属性初始化器以及 `init` 块中使用 `open` 成员。

调用超类实现

派生类中的代码可以使用 `super` 关键字调用其超类的函数与属性访问器的实现：

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

在一个内部类中访问外部类的超类，可以通过由外部类名限定的 `super` 关键字来实现：`super@Outer`：

```
class FilledRectangle: Rectangle() {
    fun draw() { /* …… */ }
    val borderColor: String get() = "black"

    inner class Filler {
        fun fill() { /* …… */ }
        fun drawAndFill() {
            super@FilledRectangle.draw() // 调用 Rectangle 的 draw() 实现
            fill()
            println("Drawn a filled rectangle with color ${super@FilledRectangle.borderColor}") // 使用
Rectangle 所实现的 borderColor 的 get()
        }
    }
}
```

覆盖规则

在 Kotlin 中，实现继承由下述规则规定：如果一个类从它的直接超类继承相同成员的多个实现，它必须覆盖这个成员并提供其自己的实现（也许用继承来的其中之一）。为了表示采用从哪个超类型继承的实现，我们使用由尖括号中超类型名限定的 `super`，如 `super<Base>`：

```
open class Rectangle {
    open fun draw() { /* …… */ }
}

interface Polygon {
    fun draw() { /* …… */ } // 接口成员默认就是“open”的
}

class Square() : Rectangle(), Polygon {
    // 编译器要求覆盖 draw():
    override fun draw() {
        super<Rectangle>.draw() // 调用 Rectangle.draw()
        super<Polygon>.draw() // 调用 Polygon.draw()
    }
}
```

可以同时继承 `Rectangle` 与 `Polygon`，但是二者都有各自的 `draw()` 实现，所以我们必须在 `Square` 中覆盖 `draw()`，并提供其自身的实现以消除歧义。

抽象类

类以及其中的某些成员可以声明为 `abstract`。抽象成员在本类中可以不用实现。需要注意的是，我们并不需要用 `open` 标注一个抽象类或者函数——因为这不言而喻。

我们可以用一个抽象成员覆盖一个非抽象的开放成员

```
open class Polygon {
    open fun draw() {}
}

abstract class Rectangle : Polygon() {
    abstract override fun draw()
}
```

伴生对象

如果你需要写一个可以无需用一个类的实例来调用、但需要访问类内部的函数（例如，工厂方法），你可以把它写成该类内对象声明中的一员。

更具体地讲，如果在你的类内声明了一个伴生对象，你就可以访问其成员，只是以类名作为限定符。

属性与字段

声明属性

Kotlin 类中的属性既可以用关键字 `var` 声明为可变的，也可以用关键字 `val` 声明为只读的。

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

要使用一个属性，只要用名称引用它即可：

```
fun copyAddress(address: Address): Address {
    val result = Address() // Kotlin 中没有“new”关键字
    result.name = address.name // 将调用访问器
    result.street = address.street
    // ……
    return result
}
```

Getters 与 Setters

声明一个属性的完整语法是

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

其初始器（initializer）、getter 和 setter 都是可选的。属性类型如果可以从初始器（或者从其 getter 返回值，如下文所示）中推断出来，也可以省略。

例如:

```
var allByDefault: Int? // 错误：需要显式初始化器，隐含默认 getter 和 setter
var initialized = 1 // 类型 Int、默认 getter 和 setter
```

一个只读属性的语法和一个可变的属性的语法有两方面的不同：1、只读属性的用 `val` 开始代替 `var` 2、只读属性不允许 setter

```
val simple: Int? // 类型 Int、默认 getter、必须在构造函数中初始化
val inferredType = 1 // 类型 Int 、默认 getter
```

我们可以为属性定义自定义的访问器。如果我们定义了一个自定义的 getter，那么每次访问该属性时都会调用它（这让我们可以实现计算出的属性）。以下是一个自定义 getter 的示例：

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

如果我们定义了一个自定义的 setter，那么每次给属性赋值时都会调用它。一个自定义的 setter 如下所示：

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // 解析字符串并赋值给其他属性
    }
```

按照惯例，setter 参数的名称是 `value`，但是如果你喜欢你可以选择一个不同的名称。

自 Kotlin 1.1 起，如果可以从 getter 推断出属性类型，则可以省略它：

```
val isEmpty get() = this.size == 0  // 具有类型 Boolean
```

如果你需要改变一个访问器的可见性或者对其注解，但是不需要改变默认的实现，你可以定义访问器而不定义其实现：

```
var setterVisibility: String = "abc"
    private set // 此 setter 是私有的并且有默认实现

var setterWithAnnotation: Any? = null
    @Inject set // 用 Inject 注解此 setter
```

幕后字段

在 Kotlin 类中不能直接声明字段。然而，当一个属性需要一个幕后字段时，Kotlin 会自动提供。这个幕后字段可以使用 `field` 标识符在访问器中引用：

```
var counter = 0 // 注意：这个初始器直接为幕后字段赋值
    set(value) {
        if (value >= 0) field = value
    }
```

`field` 标识符只能用在属性的访问器内。

如果属性至少一个访问器使用默认实现，或者自定义访问器通过 `field` 引用幕后字段，将会为该属性生成一个幕后字段。

例如，下面的情况下，就没有幕后字段：

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

幕后属性

如果你的需求不符合这套“隐式的幕后字段”方案，那么总可以使用 *幕后属性 (backing property)*：

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // 类型参数已推断出
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

对于 JVM 平台：通过默认 getter 和 setter 访问私有属性会被优化，所以本例不会引入函数调用开销。

编译期常量

如果只读属性的值在编译期是已知的，那么可以使用 `const` 修饰符将其标记为*编译期常量*。这种属性需要满足以下要求：

— 位于顶层或者是 `object` 声明 或 `companion object` 的一个成员

— 以 `String` 或原生类型值初始化

— 没有自定义 getter

这些属性可以用在注解中：

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { …… }
```

延迟初始化属性与变量

一般地，属性声明为非空类型必须在构造函数中初始化。然而，这经常不方便。例如：属性可以通过依赖注入来初始化，或者在单元测试的 setup 方法中初始化。这种情况下，你不能在构造函数内提供一个非空初始器。但你仍然想在类体中引用该属性时避免空检测。

为处理这种情况，你可以用 `lateinit` 修饰符标记该属性：

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // 直接解引用
    }
}
```

该修饰符只能用于在类体中的属性（不是在主构造函数中声明的 `var` 属性，并且仅当该属性没有自定义 getter 或 setter 时），而自 Kotlin 1.2 起，也用于顶层属性与局部变量。该属性或变量必须为非空类型，并且不能是原生类型。

在初始化前访问一个 `lateinit` 属性会抛出一个特定异常，该异常明确标识该属性被访问及它没有初始化的事实。

检测一个 lateinit var 是否已初始化（自 1.2 起）

要检测一个 `lateinit var` 是否已经初始化过，请在该属性的引用上使用 `.isInitialized`：

```
if (foo::bar.isInitialized) {
    println(foo.bar)
}
```

此检测仅对可词法级访问的属性可用，即声明位于同一个类型内、位于其中一个外围类型中或者位于相同文件的顶层的属性。

覆盖属性

参见覆盖属性

委托属性

最常见的一类属性就是简单地从幕后字段中读取（以及可能的写入）。另一方面，使用自定义 getter 和 setter 可以实现属性的任何行为。介于两者之间，属性如何工作有一些常见的模式。一些例子：惰性值、通过键值从映射读取、访问数据库、访问时通知侦听器等等。

这些常见行为可以通过使用*委托属性*实现为库。

接口

Kotlin 的接口可以既包含抽象方法的声明也包含实现。与抽象类不同的是，接口无法保存状态。它可以有属性但必须声明为抽象或提供访问器实现。

使用关键字 `interface` 来定义接口

```
interface MyInterface {
    fun bar()
    fun foo() {
      // 可选的方法体
    }
}
```

实现接口

一个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。在接口中声明的属性要么是抽象的，要么提供访问器的实现。在接口中声明的属性不能有幕后字段（backing field），因此接口中声明的访问器不能引用它们。

```
interface MyInterface {
    val prop: Int // 抽象的

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

接口继承

一个接口可以从其他接口派生，从而既提供基类型成员的实现也声明新的函数与属性。很自然地，实现这样接口的类只需定义所缺少的实现：

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // 不必实现“name”
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

解决覆盖冲突

实现多个接口时，可能会遇到同一方法继承多个实现的问题。例如

```
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()
    }
}
```

上例中，接口 *A* 和 *B* 都定义了方法 *foo()* 和 *bar()*。两者都实现了 *foo()*，但是只有 *B* 实现了 *bar()*（*bar()* 在 *A* 中没有标记为抽象，因为没有方法体时默认为抽象）。因为 *C* 是一个实现了 *A* 的具体类，所以必须要重写 *bar()* 并实现这个抽象方法。

然而，如果我们从 *A* 和 *B* 派生 *D*，我们需要实现我们从多个接口继承的所有方法，并指明 *D* 应该如何实现它们。这一规则既适用于继承单个实现（*bar()*）的方法也适用于继承多个实现（*foo()*）的方法。

可见性修饰符

类、对象、接口、构造函数、方法、属性和它们的 setter 都可以有 *可见性修饰符*。(getter 总是与属性有着相同的可见性。) 在 Kotlin 中有这四个可见性修饰符：`private`、`protected`、`internal` 和 `public`。如果没有显式指定修饰符的话，默认可见性是 `public`。

在本页可以学到这些修饰符如何应用到不同类型的声明作用域。

包

函数、属性和类、对象和接口可以在顶层声明，即直接在包内：

```
// 文件名: example.kt
package foo

fun baz() { …… }
class Bar { …… }
```

- 如果你不指定任何可见性修饰符，默认为 `public`，这意味着你的声明将随处可见；
- 如果你声明为 `private`，它只会在声明它的文件内可见；
- 如果你声明为 `internal`，它会在相同模块内随处可见；
- `protected` 不适用于顶层声明。

注意：要使用另一包中可见的顶层声明，仍需将其导入进来。

例如:

```
// 文件名: example.kt
package foo

private fun foo() { …… } // 在 example.kt 内可见

public var bar: Int = 5 // 该属性随处可见
    private set         // setter 只在 example.kt 内可见

internal val baz = 6    // 相同模块内可见
```

类和接口

对于类内部声明的成员：

- `private` 意味着只在这个类内部（包含其所有成员）可见；
- `protected` —— 和 `private` 一样 + 在子类中可见。
- `internal` —— 能见到类声明的 *本模块内* 的任何客户端都可见其 `internal` 成员；
- `public` —— 能见到类声明的任何客户端都可见其 `public` 成员。

请注意在 Kotlin 中，外部类不能访问内部类的 private 成员。

如果你覆盖一个 `protected` 成员并且没有显式指定其可见性，该成员还会是 `protected` 可见性。

例子:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4  // 默认 public

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a 不可见
    // b、c、d 可见
    // Nested 和 e 可见

    override val b = 5   // “b”为 protected
}

class Unrelated(o: Outer) {
    // o.a、o.b 不可见
    // o.c 和 o.d 可见（相同模块）
    // Outer.Nested 不可见，Nested::e 也不可见
}
```

构造函数

要指定一个类的的主构造函数的可见性，使用以下语法（注意你需要添加一个显式 `constructor` 关键字）：

```
class C private constructor(a: Int) { …… }
```

这里的构造函数是私有的。默认情况下，所有构造函数都是 `public`，这实际上等于类可见的地方它就可见（即 一个 `internal` 类的构造函数只能在相同模块内可见）.

局部声明

局部变量、函数和类不能有可见性修饰符。

模块

可见性修饰符 `internal` 意味着该成员只在相同模块内可见。更具体地说，一个模块是编译在一起的一套 Kotlin 文件：

- 一个 IntelliJ IDEA 模块；
- 一个 Maven 项目；
- 一个 Gradle 源集（例外是 `test` 源集可以访问 `main` 的 internal 声明）；
- 一次 `<kotlinc>` Ant 任务执行所编译的一套文件。

扩展

Kotlin 能够扩展一个类的新功能而无需继承该类或者使用像装饰者这样的设计模式。这通过叫做 *扩展* 的特殊声明完成。例如，你可以为一个你不能修改的、来自第三方库中的类编写一个新的函数。这个新增的函数就像那个原始类本来就有的函数一样，可以用普通的方法调用。这种机制称为 *扩展函数* 。此外，也有 *扩展属性* ，允许你为一个已经存在的类添加新的属性。

扩展函数

声明一个扩展函数，我们需要用一个 *接收者类型* 也就是被扩展的类型来作为他的前缀。下面代码为 `MutableList<Int>` 添加一个 `swap` 函数：

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // “this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

这个 `this` 关键字在扩展函数内部对应到接收者对象（传过来的在点符号前的对象）现在，我们对任意 `MutableList<Int>` 调用该函数了：

```
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // “swap()”内部的“this”会保存“list”的值
```

当然，这个函数对任何 `MutableList<T>` 起作用，我们可以泛化它：

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // “this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

为了在接收者类型表达式中使用泛型，我们要在函数名前声明泛型参数。参见泛型函数。

扩展是静态解析的

扩展不能真正的修改他们所扩展的类。通过定义一个扩展，你并没有在一个类中插入新成员，仅仅是可以通过该类型的变量用点表达式去调用这个新函数。

我们想强调的是扩展函数是静态分发的，即他们不是根据接收者类型的虚方法。这意味着调用的扩展函数是由函数调用所在的表达式的类型来决定的，而不是由表达式运行时求值结果决定的。例如：

```
open class Shape

class Rectangle: Shape()

fun Shape.getName() = "Shape"

fun Rectangle.getName() = "Rectangle"

fun printClassName(s: Shape) {
    println(s.getName())
}

printClassName(Rectangle())
```

这个例子会输出 "*Shape*"，因为调用的扩展函数只取决于参数 `s` 的声明类型，该类型是 `Shape` 类。

如果一个类定义有一个成员函数与一个扩展函数，而这两个函数又有相同的接收者类型、相同的名字，并且都适用给定的参数，这种情况**总是取成员函数**。例如：

```
class Example {
    fun printFunctionType() { println("Class method") }
}

fun Example.printFunctionType() { println("Extension function") }

Example().printFunctionType()
```

这段代码输出“*Class method*”。

当然，扩展函数重载同样名字但不同签名成员函数也完全可以：

```
class Example {
    fun printFunctionType() { println("Class method") }
}

fun Example.printFunctionType(i: Int) { println("Extension function") }

Example().printFunctionType(1)
```

可空接收者

注意可以为可空的接收者类型定义扩展。这样的扩展可以在对象变量上调用，即使其值为 null，并且可以在函数体内检测 `this == null`，这能让你在没有检测 null 的时候调用 Kotlin 中的toString()：检测发生在扩展函数的内部。

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // 空检测之后，“this”会自动转换为非空类型，所以下面的 toString()
    // 解析为 Any 类的成员函数
    return toString()
}
```

扩展属性

与函数类似，Kotlin 支持扩展属性：

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

注意：由于扩展没有实际的将成员插入类中，因此对扩展属性来说幕后字段是无效的。这就是为什么**扩展属性不能有初始化器**。他们的行为只能由显式提供的 getters/setters 定义。

例如:

```
val House.number = 1 // 错误：扩展属性不能有初始化器
```

伴生对象的扩展

如果一个类定义有一个伴生对象，你也可以为伴生对象定义扩展函数与属性。就像伴生对象的常规成员一样，可以只使用类名作为限定符来调用伴生对象的扩展成员：

```
class MyClass {
    companion object { }  // 将被称为 "Companion"
}

fun MyClass.Companion.printCompanion() { println("companion") }

fun main() {
    MyClass.printCompanion()
}
```

扩展的作用域

大多数时候我们在顶层定义扩展——直接在包里：

```
package org.example.declarations

fun List<String>.getLongestString() { /*……*/}
```

要使用所定义包之外的一个扩展，我们需要在调用方导入它：

```
package org.example.usage

import org.example.declarations.getLongestString

fun main() {
    val list = listOf("red", "green", "blue")
    list.getLongestString()
}
```

更多信息参见导入

扩展声明为成员

在一个类内部你可以为另一个类声明扩展。在这样的扩展内部，有多个 *隐式接收者* —— 其中的对象成员可以无需通过限定符访问。扩展声明所在的类的实例称为 *分发接收者*，扩展方法调用所在的接收者类型的实例称为 *扩展接收者* 。

```
class Host(val hostname: String) {
    fun printHostname() { print(hostname) }
}

class Connection(val host: Host, val port: Int) {
     fun printPort() { print(port) }

     fun Host.printConnectionString() {
         printHostname()   // 调用 Host.printHostname()
         print(":")
         printPort()   // 调用 Connection.printPort()
     }

     fun connect() {
         /*……*/
         host.printConnectionString()   // 调用扩展函数
     }
}

fun main() {
    Connection(Host("kotl.in"), 443).connect()
    //Host("kotl.in").printConnectionString(443)  // 错误，该扩展函数在 Connection 外不可用
}
```

```
}
```

对于分发接收者与扩展接收者的成员名字冲突的情况，扩展接收者优先。要引用分发接收者的成员你可以使用 限定的 this 语法。

```
class Connection {
    fun Host.getConnectionString() {
        toString()         // 调用 Host.toString()
        this@Connection.toString()  // 调用 Connection.toString()
    }
}
```

声明为成员的扩展可以声明为 `open` 并在子类中覆盖。这意味着这些函数的分发对于分发接收者类型是虚拟的，但对于扩展接收者类型是静态的。

```
open class Base { }

class Derived : Base() { }

open class BaseCaller {
    open fun Base.printFunctionInfo() {
        println("Base extension function in BaseCaller")
    }

    open fun Derived.printFunctionInfo() {
        println("Derived extension function in BaseCaller")
    }

    fun call(b: Base) {
        b.printFunctionInfo()   // 调用扩展函数
    }
}

class DerivedCaller: BaseCaller() {
    override fun Base.printFunctionInfo() {
        println("Base extension function in DerivedCaller")
    }

    override fun Derived.printFunctionInfo() {
        println("Derived extension function in DerivedCaller")
    }
}

fun main() {
    BaseCaller().call(Base())   // “Base extension function in BaseCaller”
    DerivedCaller().call(Base())  // “Base extension function in DerivedCaller”——分发接收者虚拟解析
    DerivedCaller().call(Derived())  // “Base extension function in DerivedCaller”——扩展接收者静态解析
}
```

关于可见性的说明

扩展的可见性与相同作用域内声明的其他实体的可见性相同。例如：

— 在文件顶层声明的扩展可以访问同一文件中的其他 `private` 顶层声明；

— 如果扩展是在其接收者类型外部声明的，那么该扩展不能访问接收者的 `private` 成员。

数据类

我们经常创建一些只保存数据的类。在这些类中，一些标准函数往往是从数据机械推导而来的。在 Kotlin 中，这叫做 *数据类* 并标记为 `data`：

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

编译器自动从主构造函数中声明的所有属性导出以下成员：

— `equals()` / `hashCode()` 对；

— `toString()` 格式是 `"User(name=John, age=42)"`；

— `componentN()` 函数 按声明顺序对应于所有属性；

— `copy()` 函数（见下文）。

为了确保生成的代码的一致性以及有意义的行为，数据类必须满足以下要求：

— 主构造函数需要至少有一个参数；

— 主构造函数的所有参数需要标记为 `val` 或 `var`；

— 数据类不能是抽象、开放、密封或者内部的；

— （在1.1之前）数据类只能实现接口。

此外，成员生成遵循关于成员继承的这些规则：

— 如果在数据类体中有显式实现 `equals()`、`hashCode()` 或者 `toString()`，或者这些函数在父类中有 `final` 实现，那么不会生成这些函数，而会使用现有函数；

— 如果超类型具有 `open` 的 `componentN()` 函数并且返回兼容的类型，那么会为数据类生成相应的函数，并覆盖超类的实现。如果超类型的这些函数由于签名不兼容或者是 final 而导致无法覆盖，那么会报错；

— 从一个已具 `copy(……)` 函数且签名匹配的类型派生一个数据类在 Kotlin 1.2 中已弃用，并且在 Kotlin 1.3 中已禁用。

— 不允许为 `componentN()` 以及 `copy()` 函数提供显式实现。

自 1.1 起，数据类可以扩展其他类（示例请参见密封类）。

在 JVM 中，如果生成的类需要含有一个无参的构造函数，则所有的属性必须指定默认值。（参见构造函数）。

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

在类体中声明的属性

请注意，对于那些自动生成的函数，编译器只使用在主构造函数内部定义的属性。如需在生成的实现中排除一个属性，请将其声明在类体中：

```
data class Person(val name: String) {
    var age: Int = 0
}
```

在 `toString()`、`equals()`、`hashCode()` 以及 `copy()` 的实现中只会用到 `name` 属性，并且只有一个 component 函数 `component1()`。虽然两个 `Person` 对象可以有不同的年龄，但它们会视为相等。

```
val person1 = Person("John")
val person2 = Person("John")
person1.age = 10
person2.age = 20
```

复制

在很多情况下，我们需要复制一个对象改变它的一些属性，但其余部分保持不变。`copy()` 函数就是为此而生成。对于上文的 `User` 类，其实现会类似下面这样：

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

这让我们可以写：

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

数据类与解构声明

为数据类生成的 *Component 函数* 使它们可在解构声明中使用：

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // 输出 "Jane, 35 years of age"
```

标准数据类

标准库提供了 `Pair` 与 `Triple`。尽管在很多情况下具名数据类是更好的设计选择，因为它们通过为属性提供有意义的名称使代码更具可读性。

密封类

密封类用来表示受限的类继承结构：当一个值为有限几种的类型、而不能有任何其他类型时。在某种意义上，他们是枚举类的扩展：枚举类型的值集合也是受限的，但每个枚举常量只存在一个实例，而密封类的一个子类可以有可包含状态的多个实例。

要声明一个密封类，需要在类名前面添加 `sealed` 修饰符。虽然密封类也可以有子类，但是所有子类都必须在与密封类自身相同的文件中声明。（在 Kotlin 1.1 之前，该规则更加严格：子类必须嵌套在密封类声明的内部）。

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

（上文示例使用了 Kotlin 1.1 的一个额外的新功能：数据类扩展包括密封类在内的其他类的可能性。）

一个密封类是自身抽象的，它不能直接实例化并可以有抽象（`abstract`）成员。

密封类不允许有非-`private` 构造函数（其构造函数默认为 `private`）。

请注意，扩展密封类子类的类（间接继承者）可以放在任何位置，而无需在同一个文件中。

使用密封类的关键好处在于使用 when 表达式 的时候，如果能够验证语句覆盖了所有情况，就不需要为该语句再添加一个 `else` 子句了。当然，这只有当你用 `when` 作为表达式（使用结果）而不是作为语句时才有用。

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

泛型

与 Java 类似，Kotlin 中的类也可以有类型参数：

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

一般来说，要创建这样类的实例，我们需要提供类型参数：

```
val box: Box<Int> = Box<Int>(1)
```

但是如果类型参数可以推断出来，例如从构造函数的参数或者从其他途径，允许省略类型参数：

```
val box = Box(1) // 1 具有类型 Int，所以编译器知道我们说的是 Box<Int>。
```

型变

Java 类型系统中最棘手的部分之一是通配符类型（参见 Java Generics FAQ）。而 Kotlin 中没有。相反，它有两个其他的东西：声明处型变（declaration-site variance）与类型投影（type projections）。

首先，让我们思考为什么 Java 需要那些神秘的通配符。在《Effective Java》第三版 解释了该问题——第 31 条：*利用有限制通配符来提升 API 的灵活性*。首先，Java 中的泛型是**不型变的**，这意味着 `List<String>` 并**不是** `List<Object>` 的子类型。为什么这样？如果 List 不是**不型变的**，它就没比 Java 的数组好到哪去，因为如下代码会通过编译然后导致运行时异常：

```
// Java
List<String> strs = new ArrayList<String>();
List<Object> objs = strs; // ！！！ 此处的编译器错误让我们避免了之后的运行时异常
objs.add(1); // 这里我们把一个整数放入一个字符串列表
String s = strs.get(0); // ！！！ ClassCastException：无法将整数转换为字符串
```

因此，Java 禁止这样的事情以保证运行时的安全。但这样会有一些影响。例如，考虑 `Collection` 接口中的 `addAll()` 方法。该方法的签名应该是什么？直觉上，我们会这样：

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

但随后，我们就无法做到以下简单的事情（这是完全安全）：

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
  to.addAll(from);
  // ！！！ 对于这种简单声明的 addAll 将不能编译：
  // Collection<String> 不是 Collection<Object> 的子类型
}
```

（在 Java 中，我们艰难地学到了这个教训，参见《Effective Java》第三版，第 28 条：*列表优先于数组*）

这就是为什么 `addAll()` 的实际签名是以下这样：

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

通配符类型参数 `? extends E` 表示此方法接受 `E` *或者 E 的一些子类型*对象的集合，而不只是 `E` 自身。这意味着我们可以安全地从其中（该集合中的元素是 E 的子类的实例）**读取** `E`，但**不能写入**，因为我们不知道什么对象符合那个未知的 `E` 的子类型。反过来，该限制可以让 `Collection<String>` 表示为 `Collection<? extends Object>` 的子类型。简而言之，带 **extends** 限定（**上界**）的通配符类型使得类型是**协变的（covariant）**。

理解为什么这个技巧能够工作的关键相当简单：如果只能从集合中获取项目，那么使用 `String` 的集合，并且从其中读取 `Object` 也没问题。反过来，如果只能向集合中 *放入* 项目，就可以用 `Object` 集合并向其中放入 `String`：在 Java 中有 `List<? super String>` 是 `List<Object>` 的一个**超类**。

后者称为**逆变性（contravariance）**，并且对于 `List <? super String>` 你只能调用接受 String 作为参数的方法（例如，你可以调用 `add(String)` 或者 `set(int, String)`），当然如果调用函数返回 `List<T>` 中的 `T`，你得到的并非一个 `String` 而是一个 `Object`。

Joshua Bloch 称那些你只能从中**读取**的对象为**生产者**，并称那些你只能**写入**的对象为**消费者**。他建议："*为了灵活性最大化，在表示生产者或消费者的输入参数上使用通配符类型*"，并提出了以下助记符：

PECS 代表生产者-Extens，消费者-Super（Producer-Extends, Consumer-Super）。

注意：如果你使用一个生产者对象，如 `List<? extends Foo>`，在该对象上不允许调用 `add()` 或 `set()`。但这并不意味着该对象是**不可变的**：例如，没有什么阻止你调用 `clear()` 从列表中删除所有项目，因为 `clear()` 根本无需任何参数。通配符（或其他类型的型变）保证的唯一的事情是**类型安全**。不可变性完全是另一回事。

声明处型变

假设有一个泛型接口 `Source<T>`，该接口中不存在任何以 `T` 作为参数的方法，只是方法返回 `T` 类型值：

```
// Java
interface Source<T> {
  T nextT();
}
```

那么，在 `Source <Object>` 类型的变量中存储 `Source <String>` 实例的引用是极为安全的——没有消费者-方法可以调用。但是 Java 并不知道这一点，并且仍然禁止这样操作：

```
// Java
void demo(Source<String> strs) {
  Source<Object> objects = strs; // ! ! ! 在 Java 中不允许
  // ……
}
```

为了修正这一点，我们必须声明对象的类型为 `Source<? extends Object>`，这是毫无意义的，因为我们可以像以前一样在该对象上调用所有相同的方法，所以更复杂的类型并没有带来价值。但编译器并不知道。

在 Kotlin 中，有一种方法向编译器解释这种情况。这称为**声明处型变**：我们可以标注 `Source` 的**类型参数** `T` 来确保它仅从 `Source<T>` 成员中**返回**（生产），并从不被消费。为此，我们提供 **out** 修饰符：

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // 这个没问题，因为 T 是一个 out-参数
    // ……
}
```

一般原则是：当一个类 `C` 的类型参数 `T` 被声明为 **out** 时，它就只能出现在 `C` 的成员的**输出**-位置，但回报是 `C<Base>` 可以安全地作为 `C<Derived>` 的超类。

简而言之，他们说类 `C` 是在参数 `T` 上是**协变的**，或者说 `T` 是一个**协变的**类型参数。你可以认为 `C` 是 `T` 的**生产者**，而不是 `T` 的**消费者**。

out修饰符称为**型变注解**，并且由于它在类型参数声明处提供，所以我们称之为**声明处型变**。这与 Java 的**使用处型变**相反，其类型用途通配符使得类型协变。

另外除了 **out**，Kotlin 又补充了一个型变注释：**in**。它使得一个类型参数**逆变**：只可以被消费而不可以被生产。逆变类型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 拥有类型 Double，它是 Number 的子类型
    // 因此，我们可以将 x 赋给类型为 Comparable <Double> 的变量
    val y: Comparable<Double> = x // OK!
}
```

我们相信 **in** 和 **out** 两词是自解释的（因为它们已经在 C# 中成功使用很长时间了），因此上面提到的助记符不是真正需要的，并且可以将其改写为更高的目标：

[存在性（The Existential）](#) **转换：消费者 in，生产者 out!** :-)

类型投影

使用处型变：类型投影

将类型参数 T 声明为 *out* 非常方便，并且能避免使用处子类型化的麻烦，但是有些类实际上**不能**限制为只返回 `T`！一个很好的例子是 Array：

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { …… }
    fun set(index: Int, value: T) { …… }
}
```

该类在 `T` 上既不能是协变的也不能是逆变的。这造成了一些不灵活性。考虑下述函数：

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

这个函数应该将项目从一个数组复制到另一个数组。让我们尝试在实践中应用它：

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any)
//   ^ 其类型为 Array<Int> 但此处期望 Array<Any>
```

这里我们遇到同样熟悉的问题：`Array <T>` 在 `T` 上是**不型变的**，因此 `Array <Int>` 和 `Array <Any>` 都不是另一个的子类型。为什么？再次重复，因为 copy **可能**做坏事，也就是说，例如它可能尝试**写**一个 String 到 `from`，并且如果我们实际上传递一个 `Int` 的数组，一段时间后将会抛出一个 `ClassCastException` 异常。

那么，我们唯一要确保的是 `copy()` 不会做任何坏事。我们想阻止它**写**到 `from`，我们可以：

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

这里发生的事情称为**类型投影**：我们说 `from` 不仅仅是一个数组，而是一个受限制的（**投影的**）数组：我们只可以调用返回类型为类型参数 `T` 的方法，如上，这意味着我们只能调用 `get()`。这就是我们的**使用处型变**的用法，并且是对应于 Java 的 `Array<? extends Object>`、但使用更简单些的方式。

你也可以使用 **in** 投影一个类型：

```
fun fill(dest: Array<in String>, value: String) { …… }
```

`Array<in String>` 对应于 Java 的 `Array<? super String>`，也就是说，你可以传递一个 `CharSequence` 数组或一个 `Object` 数组给 `fill()` 函数。

星投影

有时你想说，你对类型参数一无所知，但仍然希望以安全的方式使用它。这里的安全方式是定义泛型类型的这种投影，该泛型类型的每个具体实例化将是该投影的子类型。

Kotlin 为此提供了所谓的**星投影**语法：

- 对于 `Foo <out T : TUpper>`，其中 `T` 是一个具有上界 `TUpper` 的协变类型参数，`Foo <*>` 等价于 `Foo <out TUpper>`。这意味着当 `T` 未知时，你可以安全地从 `Foo <*>` *读取* `TUpper` 的值。
- 对于 `Foo <in T>`，其中 `T` 是一个逆变类型参数，`Foo <*>` 等价于 `Foo <in Nothing>`。这意味着当 `T` 未知时，没有什么可以以安全的方式*写入* `Foo <*>`。
- 对于 `Foo <T : TUpper>`，其中 `T` 是一个具有上界 `TUpper` 的不型变类型参数，`Foo<*>` 对于读取值时等价于 `Foo<out TUpper>` 而对于写值时等价于 `Foo<in Nothing>`。

如果泛型类型具有多个类型参数，则每个类型参数都可以单独投影。例如，如果类型被声明为 `interface Function <in T, out U>`，我们可以想象以下星投影：

- `Function<*, String>` 表示 `Function<in Nothing, String>`；
- `Function<Int, *>` 表示 `Function<Int, out Any?>`；
- `Function<*, *>` 表示 `Function<in Nothing, out Any?>`。

注意：星投影非常像 Java 的原始类型，但是安全。

泛型函数

不仅类可以有类型参数。函数也可以有。类型参数要放在函数名称**之前**：

```
fun <T> singletonList(item: T): List<T> {
    // ……
}

fun <T> T.basicToString(): String {  // 扩展函数
    // ……
}
```

要调用泛型函数，在调用处函数名**之后**指定类型参数即可：

```
val l = singletonList<Int>(1)
```

可以省略能够从上下文中推断出来的类型参数，所以以下示例同样适用：

```
val l = singletonList(1)
```

泛型约束

能够替换给定类型参数的所有可能类型的集合可以由**泛型约束**限制。

上界

最常见的约束类型是与 Java 的 *extends* 关键字对应的 **上界**：

```
fun <T : Comparable<T>> sort(list: List<T>) {  …… }
```

冒号之后指定的类型是**上界**：只有 `Comparable<T>` 的子类型可以替代 `T`。例如：

```
sort(listOf(1, 2, 3)) // OK。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
```

默认的上界（如果没有声明）是 `Any?`。在尖括号中只能指定一个上界。如果同一类型参数需要多个上界，我们需要一个单独的 **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() }
}
```

所传递的类型必须同时满足 `where` 子句的所有条件。在上述示例中，类型 `T` 必须*既*实现了 `CharSequence` *也*实现了 `Comparable`。

类型擦除

Kotlin 为泛型声明用法执行的类型安全检测仅在编译期进行。运行时泛型类型的实例不保留关于其类型实参的任何信息。其类型信息称为被*擦除*。例如，`Foo<Bar>` 与 `Foo<Baz?>` 的实例都会被擦除为 `Foo<*>`。

因此，并没有通用的方法在运行时检测一个泛型类型的实例是否通过指定类型参数所创建，并且编译器禁止这种 `is` 检测。

类型转换为带有具体类型参数的泛型类型，如 `foo as List<String>` 无法在运行时检测。当高级程序逻辑隐含了类型转换的类型安全而无法直接通过编译器推断时，可以使用这种非受检类型转换。编译器会对非受检类型转换发出警告，并且在运行时只对非泛型部分检测（相当于 `foo as List<*>`）。

泛型函数调用的类型参数也同样只在编译期检测。在函数体内部，类型参数不能用于类型检测，并且类型转换为类型参数（`foo as T`）也是非受检的。然而，内联函数的具体化的类型参数会由调用处内联函数体中的类型实参所代入，因此可以用于类型检测与转换，与上述泛型类型的实例具有相同限制。

嵌套类与内部类

类可以嵌套在其他类中：

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

内部类

标记为 `inner` 的嵌套类能够访问其外部类的成员。内部类会带有一个对外部类的对象的引用：

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

参见限定的 `this` 表达式以了解内部类中的 `this` 的消歧义用法。

匿名内部类

使用对象表达式创建匿名内部类实例：

```
window.addMouseListener(object : MouseAdapter() {

    override fun mouseClicked(e: MouseEvent) { …… }

    override fun mouseEntered(e: MouseEvent) { …… }
})
```

注: 对于 JVM 平台, 如果对象是函数式 Java 接口（即具有单个抽象方法的 Java 接口）的实例，你可以使用带接口类型前缀的lambda表达式创建它：

```
val listener = ActionListener { println("clicked") }
```

枚举类

枚举类的最基本的用法是实现类型安全的枚举：

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

每个枚举常量都是一个对象。枚举常量用逗号分隔。

初始化

因为每一个枚举都是枚举类的实例，所以他们可以是这样初始化过的：

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}
```

匿名类

枚举常量还可以声明其带有相应方法以及覆盖了基类方法的匿名类。

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

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

如果枚举类定义任何成员，那么使用分号将成员定义中的枚举常量定义分隔开。

枚举条目不能包含内部类以外的嵌套类型（已在 Kotlin 1.2 中弃用）。

在枚举类中实现接口

一个枚举类可以实现接口（但不能从类继承），可以为所有条目提供统一的接口成员实现，也可以在相应匿名类中为每个条目提供各自的实现。只需将接口添加到枚举类声明中即可，如下所示：

```
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)
}
```

使用枚举常量

Kotlin 中的枚举类也有合成方法允许列出定义的枚举常量以及通过名称获取枚举常量。这些方法的签名如下（假设枚举类的名称是 `EnumClass`）：

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

如果指定的名称与类中定义的任何枚举常量均不匹配，`valueOf()` 方法将抛出 `IllegalArgumentException` 异常。

自 Kotlin 1.1 起，可以使用 `enumValues<T>()` 与 `enumValueOf<T>()` 函数以泛型的方式访问枚举类中的常量：

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

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}

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

每个枚举常量都具有在枚举类声明中获取其名称与位置的属性：

```
val name: String
val ordinal: Int
```

枚举常量还实现了 Comparable 接口，其中自然顺序是它们在枚举类中定义的顺序。

对象表达式与对象声明

有时候，我们需要创建一个对某个类做了轻微改动的类的对象，而不用为之显式声明新的子类。Kotlin 用*对象表达式*和*对象声明*处理这种情况。

对象表达式

要创建一个继承自某个（或某些）类型的匿名类的对象，我们会这么写：

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

    override fun mouseEntered(e: MouseEvent) { /*……*/ }
})
```

如果超类型有一个构造函数，则必须传递适当的构造函数参数给它。多个超类型可以由跟在冒号后面的逗号分隔的列表指定：

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { /*……*/ }

val ab: A = object : A(1), B {
    override val y = 15
}
```

任何时候，如果我们只需要“一个对象而已”，并不需要特殊超类型，那么我们可以简单地写：

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

请注意，匿名对象可以用作只在本地和私有作用域中声明的类型。如果你使用匿名对象作为公有函数的返回类型或者用作公有属性的类型，那么该函数或属性的实际类型会是匿名对象声明的超类型，如果你没有声明任何超类型，就会是 `Any`。在匿名对象中添加的成员将无法访问。

```
class C {
    // 私有函数，所以其返回类型是匿名对象类型
    private fun foo() = object {
        val x: String = "x"
    }

    // 公有函数，所以其返回类型是 Any
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x        // 没问题
        val x2 = publicFoo().x  // 错误：未能解析的引用“x”
    }
}
```

对象表达式中的代码可以访问来自包含它的作用域的变量。

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ……
}
```

对象声明

单例模式在一些场景中很有用，而 Kotlin（继 Scala 之后）使单例声明变得很容易：

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ……
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ……
}
```

这称为*对象声明*。并且它总是在 `object` 关键字后跟一个名称。就像变量声明一样，对象声明不是一个表达式，不能用在赋值语句的右边。

对象声明的初始化过程是线程安全的并且在首次访问时进行。

如需引用该对象，我们直接使用其名称即可：

```
DataProviderManager.registerDataProvider(……)
```

这些对象可以有超类型：

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { …… }

    override fun mouseEntered(e: MouseEvent) { …… }
}
```

注意：对象声明不能在局部作用域（即直接嵌套在函数内部），但是它们可以嵌套到其他对象声明或非内部类中。

伴生对象

类内部的对象声明可以用 `companion` 关键字标记：

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

该伴生对象的成员可通过只使用类名作为限定符来调用：

```
val instance = MyClass.create()
```

可以省略伴生对象的名称，在这种情况下将使用名称 `Companion`：

```
class MyClass {
    companion object { }
}

val x = MyClass.Companion
```

其自身所用的类的名称（不是另一个名称的限定符）可用作对该类的伴生对象（无论是否具名）的引用：

```
class MyClass1 {
    companion object Named { }
}

val x = MyClass1

class MyClass2 {
    companion object { }
}

val y = MyClass2
```

请注意，即使伴生对象的成员看起来像其他语言的静态成员，在运行时他们仍然是真实对象的实例成员，而且，例如还可以实现接口：

```
interface Factory<T> {
    fun create(): T
}

class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}

val f: Factory<MyClass> = MyClass
```

当然，在 JVM 平台，如果使用 `@JvmStatic` 注解，你可以将伴生对象的成员生成为真正的静态方法和字段。更详细信息请参见Java 互操作性一节 。

对象表达式和对象声明之间的语义差异

对象表达式和对象声明之间有一个重要的语义差别：

— 对象表达式是在使用他们的地方**立即**执行（及初始化）的；

— 对象声明是在第一次被访问到时**延迟**初始化的；

— 伴生对象的初始化是在相应的类被加载（解析）时，与 Java 静态初始化器的语义相匹配。

类型别名

类型别名为现有类型提供替代名称。如果类型名称太长，你可以另外引入较短的名称，并使用新的名称替代原类型名。

它有助于缩短较长的泛型类型。例如，通常缩减集合类型是很有吸引力的：

```
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

你可以为函数类型提供另外的别名：

```
typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

你可以为内部类和嵌套类创建新名称：

```
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类型。它们等效于相应的底层类型。当你在代码中添加 `typealias Predicate<T>` 并使用 `Predicate<Int>` 时，Kotlin 编译器总是把它扩展为 `(Int) -> Boolean`。因此，当你需要泛型函数类型时，你可以传递该类型的变量，反之亦然：

```
typealias Predicate<T> = (T) -> Boolean

fun foo(p: Predicate<Int>) = p(42)

fun main() {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // 输出 "true"

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // 输出 "[1]"
}
```

内联类

> 内联类仅在 Kotlin 1.3 之后版本可用，目前还是*实验性的*。关于详情请参见下文

有时候，业务逻辑需要围绕某种类型创建包装器。然而，由于额外的堆内存分配问题，它会引入运行时的性能开销。此外，如果被包装的类型是原生类型，性能的损失是很糟糕的，因为原生类型通常在运行时就进行了大量优化，然而他们的包装器却没有得到任何特殊的处理。

为了解决这类问题，Kotlin 引入了一种被称为 `内联类` 的特殊类，它通过在类的前面定义一个 `inline` 修饰符来声明：

```
inline class Password(val value: String)
```

内联类必须含有唯一的一个属性在主构造函数中初始化。在运行时，将使用这个唯一属性来表示内联类的实例（关于运行时的内部表达请参阅下文）：

```
// 不存在 'Password' 类的真实实例对象
// 在运行时，'securePassword' 仅仅包含 'String'
val securePassword = Password("Don't try this in production")
```

这就是内联类的主要特性，它灵感来源于 “inline” 这个名称：类的数据被 “内联” 到该类使用的地方（类似于内联函数中的代码被内联到该函数调用的地方）。

成员

内联类支持普通类中的一些功能。特别是，内联类可以声明属性与函数：

```
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() // `greet` 方法会作为一个静态方法被调用
    println(name.length) // 属性的 get 方法会作为一个静态方法被调用
}
```

然而，内联类的成员也有一些限制：

- 内联类不能含有 `init` 代码块
- 内联类不能含有幕后字段
  - 因此，内联类只能含有简单的计算属性（不能含有延迟初始化/委托属性）

继承

内联类允许去继承接口

```
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()) // 仍然会作为一个静态方法被调用
}
```

禁止内联类参与到类的继承关系结构中。这就意味着内联类不能继承其他的类而且必须是 `final`。

表示方式

在生成的代码中，Kotlin 编译器为每个内联类保留一个包装器。内联类的实例可以在运行时表示为包装器或者基础类型。这就类似于 `Int` 可以表示为原生类型 `int` 或者包装器 `Integer`。

为了生成性能最优的代码，Kotlin 编译更倾向于使用基础类型而不是包装器。然而，有时候使用包装器是必要的。一般来说，只要将内联类用作另一种类型，它们就会被装箱。

```
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)    // 拆箱操作: 用作 Foo 本身
    asGeneric(f)   // 装箱操作: 用作泛型类型 T
    asInterface(f) // 装箱操作: 用作类型 I
    asNullable(f)  // 装箱操作: 用作不同于 Foo 的可空类型 Foo?

    // 在下面这里例子中, 'f' 首先会被装箱（当它作为参数传递给 'id' 函数时）然后又被拆箱（当它从'id'函数中被返回时）
    // 最后， 'c' 中就包含了被拆箱后的内部表达(也就是 '42')， 和 'f' 一样
    val c = id(f)
}
```

因为内联类既可以表示为基础类型有可以表示为包装器，引用相等对于内联类而言毫无意义，因此这也是被禁止的。

名字修饰

由于内联类被编译为其基础类型，因此可能会导致各种模糊的错误，例如意想不到的平台签名冲突：

```
inline class UInt(val x: Int)

// 在 JVM 平台上被表示为'public final void compute(int x)'
fun compute(x: Int) { }

// 同理，在 JVM 平台上也被表示为'public final void compute(int x)'!
fun compute(x: UInt) { }
```

为了缓解这种问题，一般会通过在函数名后面拼接一些稳定的哈希码来重命名函数。因此，`fun compute(x: UInt)` 将会被表示为 `public final void compute-<hashcode>(int x)`，以此来解决冲突的问题。

> 请注意在 Java 中 - 是一个 *无效的* 符号，也就是说在 Java 中不能调用使用内联类作为形参的函数。

内联类与类型别名

初看起来，内联类似乎与类型别名非常相似。实际上，两者似乎都引入了一种新的类型，并且都在运行时表示为基础类型。

然而，关键的区别在于类型别名与其基础类型(以及具有相同基础类型的其他类型别名)是 *赋值兼容* 的，而内联类却不是这样。

换句话说，内联类引入了一个真实的新类型，与类型别名正好相反，类型别名仅仅是为现有的类型取了个新的替代名称(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：传递别名类型的实参替代函数中基础类型的形参
    acceptString(nameInlineClass) // 错误：不能传递内联类的实参替代函数中基础类型的形参

    // And vice versa:
    acceptNameTypeAlias(string) // 正确：传递基础类型的实参替代函数中别名类型的形参
    acceptNameInlineClass(string) // 错误：不能传递基础类型的实参替代函数中内联类类型的形参
}
```

内联类的实验性状态

内联类的设计目前是实验性的，这就是说此特性是正在 *快速变化*的，并且不保证其兼容性。在 Kotlin 1.3+ 中使用内联类时，将会得到一个警告，来表明此特性还是实验性的。

如需移除警告，必须通过指定编译器参数 `-Xinline-classes` 来选择使用这项实验性的特性。

在 Gradle 中启用内联类

```
compileKotlin {
    kotlinOptions.freeCompilerArgs += ["-Xinline-classes"]
}
```

```
tasks.withType<KotlinCompile> {
    kotlinOptions.freeCompilerArgs += "-Xinline-classes"
}
```

关于详细信息，请参见编译器选项。关于多平台项目的设置，请参见使用 Gradle 构建多平台项目章节。

在 Maven 中启用内联类

```
<configuration>
    <args>
        <arg>-Xinline-classes</arg>
    </args>
</configuration>
```

关于详细信息，请参见指定编译器选项。

进一步讨论

关于其他技术详细信息和讨论，请参见内联类的语言提议

委托

属性委托

属性委托在单独一页中讲：属性委托。

由委托实现

委托模式已经证明是实现继承的一个很好的替代方式，而 Kotlin 可以零样板代码地原生支持它。`Derived` 类可以通过将其所有公有成员都委托给指定对象来实现一个接口 `Base`：

```
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` 的超类型列表中的 by-子句表示 `b` 将会在 `Derived` 中内部存储，并且编译器将生成转发给 `b` 的所有 `Base` 的方法。

覆盖由委托实现的接口成员

覆盖符合预期：编译器会使用 `override` 覆盖的实现而不是委托对象中的。如果将 `override fun printMessage() { print("abc") }` 添加到 `Derived`，那么当调用 `printMessage` 时程序会输出“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()
}
```

但请注意，以这种方式重写的成员不会在委托对象的成员中调用，委托对象的成员只能访问其自身对接口成员实现：

```
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` 实现中不会访问到这个属性
    override val message = "Message of Derived"
}

fun main() {
    val b = BaseImpl(10)
    val derived = Derived(b)
    derived.print()
    println(derived.message)
}
```

在 JVM 平台：当使用带有 `default` 方法的接口（包括带有 `@JvmDefault` 注解的 Kotlin 接口）进行委托时，即使实际的委托类型提供了其自身的实现也会调用默认实现。详细信息请参见在 Java 中调用 Kotlin。

委托属性

有一些常见的属性类型，虽然我们可以在每次需要的时候手动实现它们，但是如果能够为大家把他们只实现一次并放入一个库会更好。例如包括：

— 延迟属性（lazy properties）：其值只在首次访问时计算；
— 可观察属性（observable properties）：监听器会收到有关此属性变更的通知；
— 把多个属性储存在一个映射（map）中，而不是每个存在单独的字段中。

为了涵盖这些（以及其他）情况，Kotlin 支持 *委托属性*：

```
class Example {
    var p: String by Delegate()
}
```

语法是：`val/var <属性名>: <类型> by <表达式>`。在 by 后面的表达式是该 *委托*，因为属性对应的 `get()`（与 `set()`）会被委托给它的 `getValue()` 与 `setValue()` 方法。属性的委托不必实现任何的接口，但是需要提供一个 `getValue()` 函数（与 `setValue()`——对于 var 属性）。例如：

```
import kotlin.reflect.KProperty

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

当我们从委托到一个 `Delegate` 实例的 `p` 读取时，将调用 `Delegate` 中的 `getValue()` 函数，所以它第一个参数是读出 `p` 的对象、第二个参数保存了对 `p` 自身的描述（例如你可以取它的名字）。例如：

```
val e = Example()
println(e.p)
```

输出结果：

```
Example@33a17727, thank you for delegating ‘p’ to me!
```

类似地，当我们给 `p` 赋值时，将调用 `setValue()` 函数。前两个参数相同，第三个参数保存将要被赋予的值：

```
e.p = "NEW"
```

输出结果：

```
NEW has been assigned to ‘p’ in Example@33a17727.
```

委托对象的要求规范可以在下文找到。

请注意，自 Kotlin 1.1 起你可以在函数或代码块中声明一个委托属性，因此它不一定是类的成员。你可以在下文找到其示例。

标准委托

Kotlin 标准库为几种有用的委托提供了工厂方法。

延迟属性 Lazy

lazy() 是接受一个 lambda 并返回一个 `Lazy <T>` 实例的函数，返回的实例可以作为实现延迟属性的委托：第一次调用 `get()` 会执行已传递给 `lazy()` 的 lambda 表达式并记录结果，后续调用 `get()` 只是返回记录的结果。

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

默认情况下，对于 lazy 属性的求值是**同步锁的（synchronized）**：该值只在一个线程中计算，并且所有线程会看到相同的值。如果初始化委托的同步锁不是必需的，这样多个线程可以同时执行，那么将 `LazyThreadSafetyMode.PUBLICATION` 作为参数传递给 `lazy()` 函数。而如果你确定初始化将总是发生在与属性使用位于相同的线程，那么可以使用 `LazyThreadSafetyMode.NONE` 模式：它不会有任何线程安全的保证以及相关的开销。

可观察属性 Observable

Delegates.observable() 接受两个参数：初始值与修改时处理程序（handler）。每当我们给属性赋值时会调用该处理程序（在赋值*后*执行）。它有三个参数：被赋值的属性、旧值与新值：

```
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"
}
```

如果你想截获赋值并“否决”它们，那么使用 vetoable() 取代 `observable()`。在属性被赋新值生效*之前*会调用传递给 `vetoable` 的处理程序。

把属性储存在映射中

一个常见的用例是在一个映射（map）里存储属性的值。这经常出现在像解析 JSON 或者做其他“动态”事情的应用中。在这种情况下，你可以使用映射实例自身作为委托来实现委托属性。

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

在这个例子中，构造函数接受一个映射参数：

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

委托属性会从这个映射中取值(通过字符串键——属性的名称):

```
println(user.name) // Prints "John Doe"
println(user.age)  // Prints 25
```

这也适用于 `var` 属性,如果把只读的 `Map` 换成 `MutableMap` 的话:

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

局部委托属性(自 1.1 起)

你可以将局部变量声明为委托属性。例如,你可以使一个局部变量惰性初始化:

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

`memoizedFoo` 变量只会在第一次访问时计算。如果 `someCondition` 失败,那么该变量根本不会计算。

属性委托要求

这里我们总结了委托对象的要求。

对于一个**只读**属性(即 `val` 声明的),委托必须提供一个操作符函数 `getValue()`,该函数具有以下参数:

- `thisRef` —— 必须与 *属性所有者* 类型(对于扩展属性——指被扩展的类型)相同或者是其超类型。
- `property` —— 必须是类型 `KProperty<*>` 或其超类型。

`getValue()` 必须返回与属性相同的类型(或其子类型)。

```
class Resource

class Owner {
    val valResource: Resource by ResourceDelegate()
}

class ResourceDelegate {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return Resource()
    }
}
```

对于一个**可变**属性(即 `var` 声明的),委托必须额外提供一个操作符函数 `setValue()`,该函数具有以下参数:

- `thisRef` —— 必须与 *属性所有者* 类型(对于扩展属性——指被扩展的类型)相同或者是其超类型。
- `property` —— 必须是类型 `KProperty<*>` 或其超类型。
- `value` — 必须与属性类型相同(或者是其超类型)。

```
class Resource

class Owner {
    var varResource: Resource by ResourceDelegate()
}

class ResourceDelegate(private var resource: Resource = Resource()) {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return resource
    }
    operator fun setValue(thisRef: Owner, property: KProperty<*>, value: Any?) {
        if (value is Resource) {
            resource = value
        }
    }
}
```

`getValue()` 或/与 `setValue()` 函数可以通过委托类的成员函数提供或者由扩展函数提供。当你需要委托属性到原本未提供的这些函数的对象时后者会更便利。两函数都需要用 `operator` 关键字来进行标记。

委托类可以实现包含所需 `operator` 方法的 `ReadOnlyProperty` 或 `ReadWriteProperty` 接口之一。这俩接口是在 Kotlin 标准库中声明的：

```
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)
}
```

翻译规则

在每个委托属性的实现的背后，Kotlin 编译器都会生成辅助属性并委托给它。例如，对于属性 `prop`，生成隐藏属性 `prop$delegate`，而访问器的代码只是简单地委托给这个附加属性：

```
class C {
    var prop: Type by MyDelegate()
}

// 这段是由编译器生成的相应代码：
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 编译器在参数中提供了关于 `prop` 的所有必要信息：第一个参数 `this` 引用到外部类 `C` 的实例而 `this::prop` 是 `KProperty` 类型的反射对象，该对象描述 `prop` 自身。

请注意，直接在代码中引用绑定的可调用引用的语法 `this::prop` 自 Kotlin 1.1 起才可用。

提供委托（自 1.1 起）

通过定义 `provideDelegate` 操作符，可以扩展创建属性实现所委托对象的逻辑。如果 `by` 右侧所使用的对象将 `provideDelegate` 定义为成员或扩展函数，那么会调用该函数来创建属性委托实例。

`provideDelegate` 的一个可能的使用场景是在创建属性时（而不仅在其 getter 或 setter 中）检测属性一致性。

例如，如果要在绑定之前检测属性名称，可以这样写：

```
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` 的参数与 `getValue` 相同：

- `thisRef` —— 必须与 *属性所有者* 类型（对于扩展属性——指被扩展的类型）相同或者是它的超类型；
- `property` —— 必须是类型 `KProperty<*>` 或其超类型。

在创建 `MyUI` 实例期间，为每个属性调用 `provideDelegate` 方法，并立即执行必要的验证。

如果没有这种拦截属性与其委托之间的绑定的能力，为了实现相同的功能，你必须显式传递属性名，这不是很方便：

```
// 检测属性名称而不使用“provideDelegate”功能
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
}
```

在生成的代码中，会调用 `provideDelegate` 方法来初始化辅助的 `prop$delegate` 属性。比较对于属性声明 `val prop: Type by MyDelegate()` 生成的代码与上面（当 `provideDelegate` 方法不存在时）生成的代码：

```
class C {
    var prop: Type by MyDelegate()
}

// 这段代码是当“provideDelegate”功能可用时
// 由编译器生成的代码:
class C {
    // 调用“provideDelegate”来创建额外的“delegate”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)
}
```

请注意，`provideDelegate` 方法只影响辅助属性的创建，并不会影响为 getter 或 setter 生成的代码。

函数与 Lambda 表达式

函数

函数声明

Kotlin 中的函数使用 fun 关键字声明：

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

函数用法

调用函数使用传统的方法：

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

调用成员函数使用点表示法：

```
Stream().read() // 创建类 Stream 实例并调用 read()
```

参数

函数参数使用 Pascal 表示法定义，即 *name*: *type*。参数用逗号隔开。每个参数必须有显式类型：

```
fun powerOf(number: Int, exponent: Int) { /*……*/ }
```

默认参数

函数参数可以有默认值，当省略相应的参数时使用默认值。与其他语言相比，这可以减少重载数量：

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size) { /*……*/ }
```

默认值通过类型后面的 = 及给出的值来定义。

覆盖方法总是使用与基类型方法相同的默认参数值。当覆盖一个带有默认参数值的方法时，必须从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) { /*……*/ }  // 不能有默认值
}
```

如果一个默认参数在一个无默认值的参数之前，那么该默认值只能通过使用具名参数调用该函数来使用：

```
fun foo(bar: Int = 0, baz: Int) { /*……*/ }

foo(baz = 1) // 使用默认值 bar = 0
```

如果在默认参数之后的最后一个参数是 lambda 表达式，那么它既可以作为具名参数在括号内传入，也可以在括号外传入：

```
fun foo(bar: Int = 0, baz: Int = 1, qux: () -> Unit) { /*……*/ }

foo(1) { println("hello") }     // 使用默认值 baz = 1
foo(qux = { println("hello") }) // 使用两个默认值 bar = 0 与 baz = 1
foo { println("hello") }        // 使用两个默认值 bar = 0 与 baz = 1
```

具名参数

可以在调用函数时使用具名的函数参数。当一个函数有大量的参数或默认参数时这会非常方便。

给定以下函数：

```
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)
```

然而，当使用非默认参数调用它时，该调用看起来就像：

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

使用具名参数我们可以使代码更具有可读性：

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

并且如果我们不需要所有的参数：

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

当一个函数调用混用位置参数与具名参数时，所有位置参数都要放在第一个具名参数之前。例如，允许调用 `f(1, y = 2)` 但不允许 `f(x = 1, 2)`。

可以通过使用**星号**操作符将可变数量参数（vararg）以具名形式传入：

```
fun foo(vararg strings: String) { /*……*/ }

foo(strings = *arrayOf("a", "b", "c"))
```

> **对于 JVM 平台**：在调用 Java 函数时不能使用具名参数语法，因为 Java 字节码并不总是保留函数参数的名称。

返回 Unit 的函数

如果一个函数不返回任何有用的值，它的返回类型是 `Unit`。`Unit` 是一种只有一个值——`Unit` 的类型。这个值不需要显式返回：

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // `return Unit` 或者 `return` 是可选的
}
```

`Unit` 返回类型声明也是可选的。上面的代码等同于：

```
fun printHello(name: String?) { …… }
```

单表达式函数

当函数返回单个表达式时，可以省略花括号并且在 = 符号之后指定代码体即可：

```
fun double(x: Int): Int = x * 2
```

当返回值类型可由编译器推断时，显式声明返回类型是可选的：

```
fun double(x: Int) = x * 2
```

显式返回类型

具有块代码体的函数必须始终显式指定返回类型，除非他们旨在返回 `Unit`，在这种情况下它是可选的。Kotlin 不推断具有块代码体的函数的返回类型，因为这样的函数在代码体中可能有复杂的控制流，并且返回类型对于读者（有时甚至对于编译器）是不明显的。

可变数量的参数（Varargs）

函数的参数（通常是最后一个）可以用 `vararg` 修饰符标记：

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

允许将可变数量的参数传递给函数：

```
val list = asList(1, 2, 3)
```

在函数内部，类型 `T` 的 `vararg` 参数的可见方式是作为 `T` 数组，即上例中的 `ts` 变量具有类型 `Array <out T>`。

只有一个参数可以标注为 `vararg`。如果 `vararg` 参数不是列表中的最后一个参数，可以使用具名参数语法传递其后的参数的值，或者，如果参数具有函数类型，则通过在括号外部传一个 lambda。

当我们调用 `vararg`-函数时，我们可以一个接一个地传参，例如 `asList(1, 2, 3)`，或者，如果我们已经有一个数组并希望将其内容传给该函数，我们使用**伸展（spread）**操作符（在数组前面加 `*`）：

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

中缀表示法

标有 `infix` 关键字的函数也可以使用中缀表示法（忽略该调用的点与圆括号）调用。中缀函数必须满足以下要求：

- 它们必须是成员函数或扩展函数；
- 它们必须只有一个参数；
- 其参数不得接受可变数量的参数且不能有默认值。

```
infix fun Int.shl(x: Int): Int { …… }

// 用中缀表示法调用该函数
1 shl 2

// 等同于这样
1.shl(2)
```

> 中缀函数调用的优先级低于算术操作符、类型转换以及 `rangeTo` 操作符。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操作符 `&&` 与 `||`、`is-` 与 `in-` 检测以及其他一些操作符。这些表达式也是等价的：
>
> - `a && b xor c` 等价于 `a && (b xor c)`
> - `a xor b in c` 等价于 `(a xor b) in c`
>
> 完整的优先级层次结构请参见其语法参考。

请注意，中缀函数总是要求指定接收者与参数。当使用中缀表示法在当前接收者上调用方法时，需要显式使用 `this`；不能像常规方法调用那样省略。这是确保非模糊解析所必需的。

```
class MyStringCollection {
    infix fun add(s: String) { /*……*/ }

    fun build() {
        this add "abc"   // 正确
        add("abc")       // 正确
        //add "abc"        // 错误：必须指定接收者
    }
}
```

函数作用域

在 Kotlin 中函数可以在文件顶层声明，这意味着你不需要像一些语言如 Java、C# 或 Scala 那样需要创建一个类来保存一个函数。此外除了顶层函数，Kotlin 中函数也可以声明在局部作用域、作为成员函数以及扩展函数。

局部函数

Kotlin 支持局部函数，即一个函数在另一个函数内部：

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}
```

局部函数可以访问外部函数（即闭包）的局部变量，所以在上例中，*visited* 可以是局部变量：

```
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

成员函数是在类或对象内部定义的函数：

```
class Sample {
    fun foo() { print("Foo") }
}
```

成员函数以点表示法调用：

```
Sample().foo() // 创建类 Sample 实例并调用 foo
```

关于类和覆盖成员的更多信息参见类和继承。

泛型函数

函数可以有泛型参数，通过在函数名前使用尖括号指定：

```
fun <T> singletonList(item: T): List<T> { /*……*/ }
```

关于泛型函数的更多信息参见泛型。

内联函数

内联函数在这里讲述。

扩展函数

扩展函数在其自有章节讲述。

高阶函数和 Lambda 表达式

高阶函数和 Lambda 表达式在其自有章节讲述。

尾递归函数

Kotlin 支持一种称为尾递归的函数式编程风格。这允许一些通常用循环写的算法改用递归函数来写，而无堆栈溢出的风险。当一个函数用 `tailrec` 修饰符标记并满足所需的形式时，编译器会优化该递归，留下一个快速而高效的基于循环的版本：

```
val eps = 1E-10 // "good enough", could be 10^-15

tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

这段代码计算余弦的不动点（fixpoint of cosine），这是一个数学常数。它只是重复地从 1.0 开始调用 Math.cos，直到结果不再改变，对于这里指定的 `eps` 精度会产生 0.7390851332151611 的结果。最终代码相当于这种更传统风格的代码：

```
val eps = 1E-10 // "good enough", could be 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` 修饰符的条件的话，函数必须将其自身调用作为它执行的最后一个操作。在递归调用后有更多代码时，不能使用尾递归，并且不能用在 try/catch/finally 块中。目前在 Kotlin for JVM 与 Kotlin/Native 中支持尾递归。

高阶函数与 lambda 表达式

Kotlin 函数都是*头等的*，这意味着它们可以存储在变量与数据结构中、作为参数传递给其他高阶函数以及从其他高阶函数返回。可以像操作任何其他非函数值一样操作函数。

为促成这点，作为一门静态类型编程语言的 Kotlin 使用一系列函数类型来表示函数并提供一组特定的语言结构，例如 lambda 表达式。

高阶函数

高阶函数是将函数用作参数或返回值的函数。

一个不错的示例是集合的函数式风格的 `fold`，它接受一个初始累积值与一个接合函数，并通过将当前累积值与每个集合元素连续接合起来代入累积值来构建返回值：

```
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
}
```

在上述代码中，参数 `combine` 具有函数类型 `(R, T) -> R`，因此 `fold` 接受一个函数作为参数，该函数接受类型分别为 `R` 与 `T` 的两个参数并返回一个 `R` 类型的值。在 `for`-循环内部调用该函数，然后将其返回值赋值给 `accumulator`。

为了调用 `fold`，需要传给它一个函数类型的实例作为参数，而在高阶函数调用处，(下文详述的) lambda 表达 式广泛用于此目的。

```
val items = listOf(1, 2, 3, 4, 5)

// Lambdas 表达式是花括号括起来的代码块。
items.fold(0, {
    // 如果一个 lambda 表达式有参数，前面是参数，后跟“->”
    acc: Int, i: Int ->
    print("acc = $acc, i = $i, ")
    val result = acc + i
    println("result = $result")
    // lambda 表达式中的最后一个表达式是返回值：
    result
})

// lambda 表达式的参数类型是可选的，如果能够推断出来的话：
val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })

// 函数引用也可以用于高阶函数调用：
val product = items.fold(1, Int::times)
```

以下各节会更详细地解释上文提到的这些概念。

函数类型

Kotlin 使用类似 `(Int) -> String` 的一系列函数类型来处理函数的声明：`val onClick: () -> Unit = ……`。

这些类型具有与函数签名相对应的特殊表示法，即它们的参数和返回值：

— 所有函数类型都有一个圆括号括起来的参数类型列表以及一个返回类型：`(A, B) -> C` 表示接受类型分别为 `A` 与 `B` 两个参数并返回一个 `C` 类型值的函数类型。参数类型列表可以为空，如 `() -> A`。Unit 返回类型不可省略。

— 函数类型可以有一个额外的*接收者*类型，它在表示法中的点之前指定：类型 `A.(B) -> C` 表示可以在 `A` 的接收者对象上以一个 `B` 类型参数来调用并返回一个 `C` 类型值的函数。带有接收者的函数字面值通常与这些类型一起使用。

— 挂起函数属于特殊种类的函数类型，它的表示法中有一个 `suspend` 修饰符，例如 `suspend () -> Unit` 或者 `suspend A.(B) -> C`。

函数类型表示法可以选择性地包含函数的参数名：`(x: Int, y: Int) -> Point`。这些名称可用于表明参数的含义。

> 如需将函数类型指定为可空，请使用圆括号：`((Int, Int) -> Int)?`。
>
> 函数类型可以使用圆括号进行接合：`(Int) -> ((Int) -> Unit)`
>
> 箭头表示法是右结合的，`(Int) -> (Int) -> Unit` 与前述示例等价，但不等于 `((Int) -> (Int)) -> Unit`。

还可以通过使用类型别名给函数类型起一个别称：

```
typealias ClickHandler = (Button, ClickEvent) -> Unit
```

函数类型实例化

有几种方法可以获得函数类型的实例：

— 使用函数字面值的代码块，采用以下形式之一：
  — lambda 表达式：`{ a, b -> a + b }`，
  — 匿名函数：`fun(s: String): Int { return s.toIntOrNull() ?: 0 }`

  带有接收者的函数字面值可用作带有接收者的函数类型的值。

— 使用已有声明的可调用引用：
  — 顶层、局部、成员、扩展函数：`::isOdd`、`String::toInt`，
  — 顶层、成员、扩展属性：`List<Int>::size`，
  — 构造函数：`::Regex`

  这包括指向特定实例成员的绑定的可调用引用：`foo::toString`。

— 使用实现函数类型接口的自定义类的实例：

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

如果有足够信息，编译器可以推断变量的函数类型：

```
val a = { i: Int -> i + 1 } // 推断出的类型是 (Int) -> Int
```

带与不带接收者的函数类型*非字面*值可以互换，其中接收者可以替代第一个参数，反之亦然。例如，`(A, B) -> C` 类型的值可以传给或赋值给期待 `A.(B) -> C` 的地方，反之亦然：

```
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
```

> 请注意，默认情况下推断出的是没有接收者的函数类型，即使变量是通过扩展函数引用来初始化的。如需改变这点，请显式指定变量类型。

函数类型实例调用

函数类型的值可以通过其 invoke(……) 操作符调用：`f.invoke(x)` 或者直接 `f(x)`。

如果该值具有接收者类型，那么应该将接收者对象作为第一个参数传递。调用带有接收者的函数类型值的另一个方式是在其前面加上接收者对象，就好比该值是一个扩展函数：`1.foo(2)`，

例如：

```
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)) // 类扩展调用
```

内联函数

有时使用内联函数可以为高阶函数提供灵活的控制流。

Lambda 表达式与匿名函数

lambda 表达式与匿名函数是“函数字面值”，即未声明的函数，但立即做为表达式传递。考虑下面的例子：

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

函数 `max` 是一个高阶函数，它接受一个函数作为第二个参数。其第二个参数是一个表达式，它本身是一个函数，即函数字面值，它等价于以下具名函数：

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Lambda 表达式语法

Lambda 表达式的完整语法形式如下：

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

lambda 表达式总是括在花括号中，完整语法形式的参数声明放在花括号内，并有可选的类型标注，函数体跟在一个 `->` 符号之后。如果推断出的该 lambda 的返回类型不是 `Unit`，那么该 lambda 主体中的最后一个（或可能是单个）表达式会视为返回值。

如果我们把所有可选标注都留下，看起来如下：

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

传递末尾的 lambda 表达式

在 Kotlin 中有一个约定：如果函数的最后一个参数是函数，那么作为相应参数传入的 lambda 表达式可以放在圆括号之外：

```
val product = items.fold(1) { acc, e -> acc * e }
```

这种语法也称为*拖尾 lambda 表达式*。

如果该 lambda 表达式是调用时唯一的参数，那么圆括号可以完全省略：

```
run { println("...") }
```

`it`：单个参数的隐式名称

一个 lambda 表达式只有一个参数是很常见的。

如果编译器自己可以识别出签名，也可以不用声明唯一的参数并忽略 `->`。该参数会隐式声明为 `it`：

```
ints.filter { it > 0 } // 这个字面值是“(it: Int) -> Boolean”类型的
```

从 lambda 表达式中返回一个值

我们可以使用限定的返回语法从 lambda 显式返回一个值。否则，将隐式返回最后一个表达式的值。

因此，以下两个片段是等价的：

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

这一约定连同在圆括号外传递 lambda 表达式一起支持 LINQ-风格 的代码：

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

下划线用于未使用的变量（自 1.1 起）

如果 lambda 表达式的参数未使用，那么可以用下划线取代其名称：

```
map.forEach { _, value -> println("$value!") }
```

在 lambda 表达式中解构（自 1.1 起）

在 lambda 表达式中解构是作为解构声明的一部分描述的。

匿名函数

上面提供的 lambda 表达式语法缺少的一个东西是指定函数的返回类型的能力。在大多数情况下，这是不必要的。因为返回类型可以自动推断出来。然而，如果确实需要显式指定，可以使用另一种语法：*匿名函数*。

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

匿名函数看起来非常像一个常规函数声明，除了其名称省略了。其函数体可以是表达式（如上所示）或代码块：

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

参数和返回类型的指定方式与常规函数相同，除了能够从上下文推断出的参数类型可以省略：

```
ints.filter(fun(item) = item > 0)
```

匿名函数的返回类型推断机制与正常函数一样：对于具有表达式函数体的匿名函数将自动推断返回类型，而具有代码块函数体的返回类型必须显式指定（或者已假定为 `Unit`）。

请注意，匿名函数参数总是在括号内传递。允许将函数留在圆括号外的简写语法仅适用于 lambda 表达式。

Lambda表达式与匿名函数之间的另一个区别是非局部返回的行为。一个不带标签的 `return` 语句总是在用 `fun` 关键字声明的函数中返回。这意味着 lambda 表达式中的 `return` 将从包含它的函数返回，而匿名函数中的 `return` 将从匿名函数自身返回。

闭包

Lambda 表达式或者匿名函数（以及局部函数和对象表达式）可以访问其 *闭包*，即在外部作用域中声明的变量。在 lambda 表达式中可以修改闭包中捕获的变量：

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

带有接收者的函数字面值

带有接收者的函数类型，例如 `A.(B) -> C`，可以用特殊形式的函数字面值实例化—— 带有接收者的函数字面值。

如上所述，Kotlin 提供了调用带有接收者（提供*接收者对象*）的函数类型实例的能力。

在这样的函数字面值内部，传给调用的接收者对象成为*隐式*的`this`，以便访问接收者对象的成员而无需任何额外的限定符，亦可使用 `this` 表达式 访问接收者对象。

这种行为与扩展函数类似，扩展函数也允许在函数体内部访问接收者对象的成员。

这里有一个带有接收者的函数字面值及其类型的示例，其中在接收者对象上调用了 `plus`：

```
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收者类型。如果你需要使用带接收者的函数类型声明一个变量，并在之后使用它，这将非常有用。

```
val sum = fun Int.(other: Int): Int = this + other
```

当接收者类型可以从上下文推断时，lambda 表达式可以用作带接收者的函数字面值。One of the most important examples of their usage is type-safe builders:

```
class HTML {
    fun body() { …… }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()  // 创建接收者对象
    html.init()        // 将该接收者对象传给该 lambda
    return html
}

html {       // 带接收者的 lambda 由此开始
    body()   // 调用该接收者对象的一个方法
}
```

内联函数

使用高阶函数会带来一些运行时的效率损失：每一个函数都是一个对象，并且会捕获一个闭包。即那些在函数体内会访问到的变量。内存分配（对于函数对象和类）和虚拟调用会引入运行时间开销。

但是在许多情况下通过内联化 lambda 表达式可以消除这类的开销。下述函数是这种情况的很好的例子。即 `lock()` 函数可以很容易地在调用处内联。考虑下面的情况：

```
lock(l) { foo() }
```

编译器没有为参数创建一个函数对象并生成一个调用。取而代之，编译器可以生成以下代码：

```
l.lock()
try {
    foo()
}
finally {
    l.unlock()
}
```

这个不是我们从一开始就想要的吗？

为了让编译器这么做，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` 修饰符影响函数本身和传给它的 lambda 表达式：所有这些都将内联到调用处。

内联可能导致生成的代码增加；不过如果我们使用得当（即避免内联过大函数），性能上会有所提升，尤其是在循环中的“超多态（megamorphic）”调用处。

禁用内联

如果希望只内联一部分传给内联函数的 lambda 表达式参数，那么可以用 `noinline` 修饰符标记不希望内联的函数参数：

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) { …… }
```

可以内联的 lambda 表达式只能在内联函数内部调用或者作为可内联的参数传递，但是 `noinline` 的可以以任何我们喜欢的方式操作：存储在字段中、传送它等等。

需要注意的是，如果一个内联函数没有可内联的函数参数并且没有具体化的类型参数，编译器会产生一个警告，因为内联这样的函数很可能并无益处（如果你确认需要内联，则可以用 `@Suppress("NOTHING_TO_INLINE")` 注解关掉该警告）。

非局部返回

在 Kotlin 中，我们只能对具名或匿名函数使用正常的、非限定的 return 来退出。这意味着要退出一个 lambda 表达式，我们必须使用一个标签，并且在 lambda 表达式内部禁止使用裸 `return`，因为 lambda 表达式不能使包含它的函数返回：

```
fun foo() {
    ordinaryFunction {
        return // 错误：不能使 `foo` 在此处返回
    }
}
```

但是如果 lambda 表达式传给的函数是内联的，该 return 也可以内联，所以它是允许的：

inline fun inlined(block: () -> Unit) { println("hi!") }

```
fun foo() {
    inlined {
        return // OK：该 lambda 表达式是内联的
    }
}
```

这种返回（位于 lambda 表达式中，但退出包含它的函数）称为*非局部*返回。我们习惯了在循环中用这种结构，其内联函数通常包含：

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // 从 hasZeros 返回
    }
    return false
}
```

请注意，一些内联函数可能调用传给它们的不是直接来自函数体、而是来自另一个执行上下文的 lambda 表达式参数，例如来自局部对象或嵌套函数。在这种情况下，该 lambda 表达式中也不允许非局部控制流。为了标识这种情况，该 lambda 表达式参数需要用 `crossinline` 修饰符标记：

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ……
}
```

> `break` 和 `continue` 在内联的 lambda 表达式中还不可用，但我们也计划支持它们。

具体化的类型参数

有时候我们需要访问一个作为参数传给我们的一个类型：

```
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?
}
```

在这里我们向上遍历一棵树并且检测每个节点是不是特定的类型。这都没有问题，但是调用处不是很优雅：

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

我们真正想要的只是传一个类型给该函数，即像这样调用它：

```
treeNode.findParentOfType<MyTreeNode>()
```

为能够这么做，内联函数支持*具体化的类型参数*，于是我们可以这样写：

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

我们使用 `reified` 修饰符来限定类型参数，现在可以在函数内部访问它了，几乎就像是一个普通的类一样。由于函数是内联的，不需要反射，正常的操作符如 `!is` 和 `as` 现在都能用了。此外，我们还可以按照上面提到的方式调用它：`myTree.findParentOfType<MyTreeNodeType>()`。

虽然在许多情况下可能不需要反射，但我们仍然可以对一个具体化的类型参数使用它：

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

普通的函数（未标记为内联函数的）不能有具体化参数。不具有运行时表示的类型（例如非具体化的类型参数或者类似于 `Nothing` 的虚构类型）不能用作具体化的类型参数的实参。

相关底层描述，请参见规范文档。

内联属性（自 1.1 起）

`inline` 修饰符可用于没有幕后字段的属性的访问器。你可以标注独立的属性访问器：

```
val foo: Foo
    inline get() = Foo()

var bar: Bar
    get() = ……
    inline set(v) { …… }
```

你也可以标注整个属性，将它的两个访问器都标记为内联：

```
inline var bar: Bar
    get() = ……
    set(v) { …… }
```

在调用处，内联访问器如同内联函数一样内联。

公有 API 内联函数的限制

当一个内联函数是 `public` 或 `protected` 而不是 `private` 或 `internal` 声明的一部分时，就会认为它是一个模块级的公有 API。可以在其他模块中调用它，并且也可以在调用处内联这样的调用。

这带来了一些由模块做这样变更时导致的二进制兼容的风险——声明一个内联函数但调用它的模块在它修改后并没有重新编译。

为了消除这种由**非**公有 API 变更引入的不兼容的风险，公有 API 内联函数体内不允许使用非公有声明，即，不允许使用 `private` 与 `internal` 声明以及其部件。

一个 `internal` 声明可以由 `@PublishedApi` 标注，这会允许它在公有 API 内联函数中使用。当一个 `internal` 内联函数标记有 `@PublishedApi` 时，也会像公有函数一样检测其函数体。

集合

Kotlin 集合概述

Kotlin 标准库提供了一整套用于管理*集合*的工具，集合是可变数量（可能为零）的一组条目，各种集合对于解决问题都具有重要意义，并且经常用到。

集合是大多数编程语言的常见概念，因此如果熟悉像 Java 或者 Python 语言的集合，那么可以跳过这一介绍转到详细部分。

集合通常包含相同类型的一些（数目也可以为零）对象。集合中的对象称为*元素*或*条目*。例如，一个系的所有学生组成一个集合，可以用于计算他们的平均年龄。以下是 Kotlin 相关的集合类型：

- *List* 是一个有序集合，可通过索引（反映元素位置的整数）访问元素。元素可以在 list 中出现多次。列表的一个示例是一句话：有一组字、这些字的顺序很重要并且字可以重复。
- *Set* 是唯一元素的集合。它反映了集合（set）的数学抽象：一组无重复的对象。一般来说 set 中元素的顺序并不重要。例如，字母表是字母的集合（set）。
- *Map*（或者*字典*）是一组键值对。键是唯一的，每个键都刚好映射到一个值。值可以重复。map 对于存储对象之间的逻辑连接非常有用，例如，员工的 ID 与员工的位置。

Kotlin 让你可以独立于所存储对象的确切类型来操作集合。换句话说，将 `String` 添加到 `String` list 中的方式与添加 `Int` 或者用户自定义类的到相应 list 中的方式相同。因此，Kotlin 标准库为创建、填充、管理任何类型的集合提供了泛型的（通用的，双关）接口、类与函数。

这些集合接口与相关函数位于 kotlin.collections 包中。我们来大致了解下其内容。

集合类型

Kotlin 标准库提供了基本集合类型的实现：set、list 以及 map。一对接口代表每种集合类型：

- 一个 *只读* 接口，提供访问集合元素的操作。
- 一个 *可变* 接口，通过写操作扩展相应的只读接口：添加、删除和更新其元素。

请注意，更改可变集合不需要它是以 `var` 定义的变量：写操作修改同一个可变集合对象，因此引用不会改变。但是，如果尝试对 `val` 集合重新赋值，你将收到编译错误。

```
val numbers = mutableListOf("one", "two", "three", "four")
numbers.add("five")   // 这是可以的
//numbers = mutableListOf("six", "seven")      // 编译错误
```

只读集合类型是型变的。这意味着，如果类 `Rectangle` 继承自 `Shape`，则可以在需要 `List <Shape>` 的任何地方使用 `List <Rectangle>`。换句话说，集合类型与元素类型具有相同的子类型关系。map 在值（value）类型上是型变的，但在键（key）类型上不是。

反之，可变集合不是型变的；否则将导致运行时故障。如果 `MutableList <Rectangle>` 是 `MutableList <Shape>` 的子类型，你可以在其中插入其他 `Shape` 的继承者（例如，`Circle`），从而违反了它的 `Rectangle` 类型参数。

下面是 Kotlin 集合接口的图表：

让我们来看看接口及其实现。

Collection

`Collection<T>` 是集合层次结构的根。此接口表示一个只读集合的共同行为：检索大小、检测是否为成员等等。`Collection` 继承自 `Iterable <T>` 接口，它定义了迭代元素的操作。可以使用 `Collection` 作为适用于不同集合类型的函数的参数。对于更具体的情况，请使用 `Collection` 的继承者：`List` 与 `Set`。

```
fun printAll(strings: Collection<String>) {
        for(s in strings) print("$s ")
        println()
    }

fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet)
}
```

`MutableCollection` 是一个具有写操作的 `Collection` 接口，例如 `add` 以及 `remove`。

```
fun List<String>.getShortWordsTo(shortWords: MutableList<String>, maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // throwing away the articles
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split(" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords)
}
```

List

`List<T>` 以指定的顺序存储元素，并提供使用索引访问元素的方法。索引从 0 开始 – 第一个元素的索引 – 直到 `最后一个元素的索引` 即 `(list.size - 1)`。

```
val numbers = listOf("one", "two", "three", "four")
println("Number of elements: ${numbers.size}")
println("Third element: ${numbers.get(2)}")
println("Fourth element: ${numbers[3]}")
println("Index of element \"two\" ${numbers.indexOf("two")}")
```

List 元素（包括空值）可以重复：List 可以包含任意数量的相同对象或单个对象的出现。如果两个 List 在相同的位置具有相同大小和相同结构的元素，则认为它们是相等的。

```
val bob = Person("Bob", 31)
val people = listOf(Person("Adam", 20), bob, bob)
val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
println(people == people2)
bob.age = 32
println(people == people2)
```

`MutableList<T>` 是可以进行写操作的 `List`，例如用于在特定位置添加或删除元素。

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

如你所见，在某些方面，List 与数组（Array）非常相似。但是，有一个重要的区别：数组的大小是在初始化时定义的，永远不会改变；反之，List 没有预定义的大小；作为写操作的结果，可以更改 List 的大小：添加，更新或删除元素。

在 Kotlin 中，`List` 的默认实现是 `ArrayList`，可以将其视为可调整大小的数组。

Set

`Set<T>` 存储唯一的元素；它们的顺序通常是未定义的。`null` 元素也是唯一的：一个 `Set` 只能包含一个 `null`。当两个 `set` 具有相同的大小并且对于一个 `set` 中的每个元素都能在另一个 `set` 中存在相同元素，则两个 `set` 相等。

```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")
if (numbers.contains(1)) println("1 is in the set")

val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal: ${numbers == numbersBackwards}")
```

`MutableSet` 是一个带有来自 `MutableCollection` 的写操作接口的 `Set`。

`Set` 的默认实现 - `LinkedHashSet` – 保留元素插入的顺序。因此，依赖于顺序的函数，例如 `first()` 或 `last()`，会在这些 `set` 上返回可预测的结果。

```
val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet is the default implementation
val numbersBackwards = setOf(4, 3, 2, 1)

println(numbers.first() == numbersBackwards.first())
println(numbers.first() == numbersBackwards.last())
```

另一种实现方式 – `HashSet` – 不声明元素的顺序，所以在它上面调用这些函数会返回不可预测的结果。但是，`HashSet` 只需要较少的内存来存储相同数量的元素。

Map

`Map<K, V>` 不是 `Collection` 接口的继承者；但是它也是 Kotlin 的一种集合类型。`Map` 存储 *键-值* 对（或 *条目*）；键是唯一的，但是不同的键可以与相同的值配对。`Map` 接口提供特定的函数进行通过键访问值、搜索键和值等操作。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

println("All keys: ${numbersMap.keys}")
println("All values: ${numbersMap.values}")
if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
if (1 in numbersMap.values) println("The value 1 is in the map")
if (numbersMap.containsValue(1)) println("The value 1 is in the map") // 同上
```

无论键值对的顺序如何，包含相同键值对的两个 `Map` 是相等的。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)

println("The maps are equal: ${numbersMap == anotherMap}")
```

MutableMap 是一个具有写操作的 `Map` 接口，可以使用该接口添加一个新的键值对或更新给定键的值。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
numbersMap["one"] = 11

println(numbersMap)
```

`Map` 的默认实现 – LinkedHashMap – 迭代 Map 时保留元素插入的顺序。反之，另一种实现 – HashMap – 不声明元素的顺序。

构造集合

由元素构造

创建集合的最常用方法是使用标准库函数 listOf<T>()、setOf<T>()、mutableListOf<T>()、mutableSetOf<T>()。如果以逗号分隔的集合元素列表作为参数，编译器会自动检测元素类型。创建空集合时，须明确指定类型。

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

同样的，Map 也有这样的函数 mapOf() 与 mutableMapOf()。映射的键和值作为 `Pair` 对象传递（通常使用中缀函数 `to` 创建）。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
```

注意，`to` 符号创建了一个短时存活的 `Pair` 对象，因此建议仅在性能不重要时才使用它。为避免过多的内存使用，请使用其他方法。例如，可以创建可写 Map 并使用写入操作填充它。apply() 函数可以帮助保持初始化流畅。

```
val numbersMap = mutableMapOf<String, String>().apply { this["one"] = "1"; this["two"] = "2" }
```

空集合

还有用于创建没有任何元素的集合的函数：emptyList()、emptySet() 与 emptyMap()。创建空集合时，应指定集合将包含的元素类型。

```
val empty = emptyList<String>()
```

list 的初始化函数

对于 List，有一个接受 List 的大小与初始化函数的构造函数，该初始化函数根据索引定义元素的值。

```
val doubled = List(3, { it * 2 })  // 如果你想操作这个集合，应使用 MutableList
println(doubled)
```

具体类型构造函数

要创建具体类型的集合，例如 `ArrayList` 或 `LinkedList`，可以使用这些类型的构造函数。类似的构造函数对于 `Set` 与 `Map` 的各实现中均有提供。

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))
val presizedSet = HashSet<Int>(32)
```

复制

要创建与现有集合具有相同元素的集合，可以使用复制操作。标准库中的集合复制操作创建了具有相同元素引用的 **浅复制集合**。因此，对集合元素所做的更改会反映在其所有副本中。

在特定时刻通过集合复制函数，例如toList()、toMutableList()、toSet() 等等。创建了集合的快照。结果是创建了一个具有相同元素的新集合 如果在源集合中添加或删除元素，则不会影响副本。副本也可以独立于源集合进行更改。

```
val sourceList = mutableListOf(1, 2, 3)
val copyList = sourceList.toMutableList()
val readOnlyCopyList = sourceList.toList()
sourceList.add(4)
println("Copy size: ${copyList.size}")

//readOnlyCopyList.add(4)             // 编译异常
println("Read-only copy size: ${readOnlyCopyList.size}")
```

这些函数还可用于将集合转换为其他类型，例如根据 List 构建 Set，反之亦然。

```
val sourceList = mutableListOf(1, 2, 3)
val copySet = sourceList.toMutableSet()
copySet.add(3)
copySet.add(4)
println(copySet)
```

或者，可以创建对同一集合实例的新引用。使用现有集合初始化集合变量时，将创建新引用。因此，当通过引用更改集合实例时，更改将反映在其所有引用中。

```
val sourceList = mutableListOf(1, 2, 3)
val referenceList = sourceList
referenceList.add(4)
println("Source size: ${sourceList.size}")
```

集合的初始化可用于限制其可变性。例如，如果构建了一个 `MutableList` 的 `List` 引用，当你试图通过此引用修改集合的时候，编译器会抛出错误。

```
val sourceList = mutableListOf(1, 2, 3)
val referenceList: List<Int> = sourceList
//referenceList.add(4)            // 编译错误
sourceList.add(4)
println(referenceList) // 显示 sourceList 当前状态
```

调用其他集合的函数

可以通过其他集合各种操作的结果来创建集合。例如，过滤列表会创建与过滤器匹配的新元素列表：

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)
```

映射生成转换结果列表：

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { idx, value -> value * idx })
```

关联生成 Map:

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
```

有关 Kotlin 中集合操作的更多信息，参见集合操作概述.

迭代器

对于遍历集合元素，Kotlin 标准库支持 *迭代器* 的常用机制——对象可按顺序提供对元素的访问权限，而不会暴露集合的底层结构。当需要逐个处理集合的所有元素（例如打印值或对其进行类似更新）时，迭代器非常有用。

`Iterable<T>` 接口的继承者（包括 `Set` 与 `List`）可以通过调用 `iterator()` 函数获得迭代器。一旦获得迭代器它就指向集合的第一个元素；调用 `next()` 函数将返回此元素，并将迭代器指向下一个元素（如果下一个元素存在）。一旦迭代器通过了最后一个元素，它就不能再用于检索元素；也无法重新指向到以前的任何位置。要再次遍历集合，请创建一个新的迭代器。

```
val numbers = listOf("one", "two", "three", "four")
val numbersIterator = numbers.iterator()
while (numbersIterator.hasNext()) {
    println(numbersIterator.next())
}
```

遍历 `Iterable` 集合的另一种方法是众所周知的 `for` 循环。在集合中使用 `for` 循环时，将隐式获取迭代器。因此，以下代码与上面的示例等效：

```
val numbers = listOf("one", "two", "three", "four")
for (item in numbers) {
    println(item)
}
```

最后，有一个好用的 `forEach()` 函数，可自动迭代集合并为每个元素执行给定的代码。因此，等效的示例如下所示：

```
val numbers = listOf("one", "two", "three", "four")
numbers.forEach {
    println(it)
}
```

List 迭代器

对于列表，有一个特殊的迭代器实现：`ListIterator` 它支持列表双向迭代：正向与反向。反向迭代由 `hasPrevious()` 和 `previous()` 函数实现。此外，`ListIterator` 通过 `nextIndex()` 与 `previousIndex()` 函数提供有关元素索引的信息。

```
val numbers = listOf("one", "two", "three", "four")
val listIterator = numbers.listIterator()
while (listIterator.hasNext()) listIterator.next()
println("Iterating backwards:")
while (listIterator.hasPrevious()) {
    print("Index: ${listIterator.previousIndex()}")
    println(", value: ${listIterator.previous()}")
}
```

具有双向迭代的能力意味着 `ListIterator` 在到达最后一个元素后仍可以使用。

可变迭代器

为了迭代可变集合，于是有了 `MutableIterator` 来扩展 `Iterator` 使其具有元素删除函数 `remove()` 。因此，可以在迭代时从集合中删除元素。

```
val numbers = mutableListOf("one", "two", "three", "four")
val mutableIterator = numbers.iterator()

mutableIterator.next()
mutableIterator.remove()
println("After removal: $numbers")
```

除了删除元素，MutableListIterator 还可以在迭代列表时插入和替换元素。

```
val numbers = mutableListOf("one", "four", "four")
val mutableListIterator = numbers.listIterator()

mutableListIterator.next()
mutableListIterator.add("two")
mutableListIterator.next()
mutableListIterator.set("three")
println(numbers)
```

区间与数列

Kotlin 可通过调用 `kotlin.ranges` 包中的 `rangeTo()` 函数及其操作符形式的 `..` 轻松地创建两个值的区间。通常，`rangeTo()` 会辅以 `in` 或 `!in` 函数。

```
if (i in 1..4) {  // 等同于 1 <= i && i <= 4
    print(i)
}
```

整数类型区间（`IntRange`、`LongRange`、`CharRange`）还有一个拓展特性：可以对其进行迭代。这些区间也是相应整数类型的等差数列。这种区间通常用于 `for` 循环中的迭代。

```
for (i in 1..4) print(i)
```

要反向迭代数字，请使用 `downTo` 函数而不是 `..` 。

```
for (i in 4 downTo 1) print(i)
```

也可以通过任意步长（不一定为 1）迭代数字。这是通过 `step` 函数完成的。

```
for (i in 1..8 step 2) print(i)
println()
for (i in 8 downTo 1 step 2) print(i)
```

要迭代不包含其结束元素的数字区间，请使用 `until` 函数：

```
for (i in 1 until 10) {       // i in [1, 10), 10被排除
    print(i)
}
```

区间

区间从数学意义上定义了一个封闭的间隔：它由两个端点值定义，这两个端点值都包含在该区间内。区间是为可比较类型定义的：具有顺序，可以定义任意实例是否在两个给定实例之间的区间内。区间的主要操作是 `contains`，通常以 `in` 与 `!in` 操作符的形式使用。

要为类创建一个区间，请在区间起始值上调用 `rangeTo()` 函数，并提供结束值作为参数。`rangeTo()` 通常以操作符 `..` 形式调用。

```
val versionRange = Version(1, 11)..Version(1, 30)
println(Version(0, 9) in versionRange)
println(Version(1, 20) in versionRange)
```

数列

如上个示例所示，整数类型的区间（例如 `Int`、`Long` 与 `Char`）可视为等差数列。在 Kotlin 中，这些数列由特殊类型定义：`IntProgression`、`LongProgression` 与 `CharProgression`。

数列具有三个基本属性：`first` 元素、`last` 元素和一个非零的 `step`。首个元素为 `first`，后续元素是前一个元素加上一个 `step`。以确定的步长在数列上进行迭代等效于 Java/JavaScript 中基于索引的 `for` 循环。

```
for (int i = first; i <= last; i += step) {
  // ……
}
```

通过迭代数列隐式创建区间时，此数列的 `first` 与 `last` 元素是区间的端点，`step` 为 1。

```
for (i in 1..10) print(i)
```

要指定数列步长，请在区间上使用 `step` 函数。

```
for (i in 1..8 step 2) print(i)
```

数列的 `last` 元素是这样计算的：

— 对于正步长：不大于结束值且满足 `(last - first) % step == 0` 的最大值。

— 对于负步长：不小于结束值且满足 `(last - first) % step == 0` 的最小值。

因此，`last` 元素并非总与指定的结束值相同。

```
for (i in 1..9 step 3) print(i) // 最后一个元素是 7
```

要创建反向迭代的数列，请在定义其区间时使用 `downTo` 而不是 `..`。

```
for (i in 4 downTo 1) print(i)
```

数列实现 `Iterable<N>`，其中 `N` 分别是 `Int`、`Long` 或 `Char`，因此可以在各种集合函数（如 `map`、`filter` 与其他）中使用它们。

```
println((1..10).filter { it % 2 == 0 })
```

序列

除了集合之外，Kotlin 标准库还包含另一种容器类型——*序列*（`Sequence<T>`）。序列提供与 `Iterable` 相同的函数，但实现另一种方法来进行多步骤集合处理。

当 `Iterable` 的处理包含多个步骤时，它们会优先执行：每个处理步骤完成并返回其结果——中间集合。在此集合上执行以下步骤。反过来，序列的多步处理在可能的情况下会延迟执行：仅当请求整个处理链的结果时才进行实际计算。

操作执行的顺序也不同：`Sequence` 对每个元素逐个执行所有处理步骤。反过来，`Iterable` 完成整个集合的每个步骤，然后进行下一步。

因此，这些序列可避免生成中间步骤的结果，从而提高了整个集合处理链的性能。但是，序列的延迟性质增加了一些开销，这些开销在处理较小的集合或进行更简单的计算时可能很重要。因此，应该同时考虑使用 `Sequence` 与 `Iterable`，并确定在哪种情况更适合。

构造

由元素

要创建一个序列，请调用 `sequenceOf()` 函数，列出元素作为其参数。

```
val numbersSequence = sequenceOf("four", "three", "two", "one")
```

由 `Iterable`

如果已经有一个 `Iterable` 对象（例如 `List` 或 `Set`），则可以通过调用 `asSequence()` 从而创建一个序列。

```
val numbers = listOf("one", "two", "three", "four")
val numbersSequence = numbers.asSequence()
```

由函数

创建序列的另一种方法是通过使用计算其元素的函数来构建序列。要基于函数构建序列，请以该函数作为参数调用 `generateSequence()`。（可选）可以将第一个元素指定为显式值或函数调用的结果。当提供的函数返回 `null` 时，序列生成停止。因此，以下示例中的序列是无限的。

```
val oddNumbers = generateSequence(1) { it + 2 } // `it` 是上一个元素
println(oddNumbers.take(5).toList())
//println(oddNumbers.count())     // 错误：此序列是无限的。
```

要使用 `generateSequence()` 创建有限序列，请提供一个函数，该函数在需要的最后一个元素之后返回 `null`。

```
val oddNumbersLessThan10 = generateSequence(1) { if (it < 10) it + 2 else null }
println(oddNumbersLessThan10.count())
```

由组块

最后，有一个函数可以逐个或按任意大小的组块生成序列元素——`sequence()` 函数。此函数采用一个 lambda 表达式，其中包含 `yield()` 与 `yieldAll()` 函数的调用。它们将一个元素返回给序列使用者，并暂停 `sequence()` 的执行，直到使用者请求下一个元素。`yield()` 使用单个元素作为参数；`yieldAll()` 中可以采用 `Iterable` 对象、`Iterable` 或其他 `Sequence`。`yieldAll()` 的 `Sequence` 参数可以是无限的。当然，这样的调用必须是最后一个：之后的所有调用都永远不会执行。

```
val oddNumbers = sequence {
    yield(1)
    yieldAll(listOf(3, 5))
    yieldAll(generateSequence(7) { it + 2 })
}
println(oddNumbers.take(5).toList())
```

序列操作

关于序列操作，根据其状态要求可以分为以下几类：

- *无状态* 操作不需要状态，并且可以独立处理每个元素，例如 `map()` 或 `filter()`。无状态操作还可能需要少量常数个状态来处理元素，例如 `take()` 与 `drop()`。
- *有状态* 操作需要大量状态，通常与序列中元素的数量成比例。

如果序列操作返回延迟生成的另一个序列，则称为 *中间序列*。否则，该操作为 *末端* 操作。末端操作的示例为 `toList()` 或 `sum()`。只能通过末端操作才能检索序列元素。

序列可以多次迭代；但是，某些序列实现可能会约束自己仅迭代一次。其文档中特别提到了这一点。

序列处理示例

我们通过一个示例来看 `Iterable` 与 `Sequence` 之间的区别。

Iterable

假定有一个单词列表。下面的代码过滤长于三个字符的单词，并打印前四个单词的长度。

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)
```

运行此代码时，会看到 `filter()` 与 `map()` 函数的执行顺序与代码中出现的顺序相同。首先，将看到 `filter`：对于所有元素，然后是 `length`：对于在过滤之后剩余的元素，然后是最后两行的输出。列表处理如下图：

Sequence

现在用序列写相同的逻辑：

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
// 将列表转换为序列
val wordsSequence = words.asSequence()

val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars")
// 末端操作：以列表形式获取结果。
println(lengthsSequence.toList())
```

此代码的输出表明，仅在构建结果列表时才调用 `filter()` 与 `map()` 函数。因此，首先看到文本 `“Lengths of..”` 的行，然后开始进行序列处理。请注意，对于过滤后剩余的元素，映射在过滤下一个元素之前执行。当结果大小达到 4 时，处理将停止，因为它是 `take(4)` 可以返回的最大大小。

序列处理如下图：

在此示例中，序列处理需要 18 个步骤，而不是 23 个步骤来执行列表操作。

集合操作概述

Kotlin 标准库提供了用于对集合执行操作的多种函数。这包括简单的操作，例如获取或添加元素，以及更复杂的操作，包括搜索、排序、过滤、转换等。

扩展与成员函数

集合操作在标准库中以两种方式声明：集合接口的成员函数和扩展函数。

成员函数定义了对于集合类型是必不可少的操作。例如，`Collection` 包含函数 `isEmpty()` 来检查其是否为空；`List` 包含用于对元素进行索引访问的 `get()`，等等。

创建自己的集合接口实现时，必须实现其成员函数。为了使新实现的创建更加容易，请使用标准库中集合接口的框架实现：`AbstractCollection`、`AbstractList`、`AbstractSet`、`AbstractMap` 及其相应可变抽象类。

其他集合操作被声明为扩展函数。这些是过滤、转换、排序和其他集合处理功能。

公共操作

公共操作可用于只读集合与可变集合。常见操作分为以下几类：

- 集合转换
- 集合过滤
- plus 与 minus 操作符
- 分组
- 取集合的一部分
- 取单个元素
- 集合排序
- 集合聚合操作

这些页面中描述的操作将返回其结果，而不会影响原始集合。例如，一个过滤操作产生一个_新集合_，其中包含与过滤谓词匹配的所有元素。此类操作的结果应存储在变量中，或以其他方式使用，例如，传到其他函数中。

```
val numbers = listOf("one", "two", "three", "four")
numbers.filter { it.length > 3 }  // `numbers` 没有任何改变，结果丢失
println("numbers are still $numbers")
val longerThan3 = numbers.filter { it.length > 3 } // 结果存储在 `longerThan3` 中
println("numbers longer than 3 chars are $longerThan3")
```

对于某些集合操作，有一个选项可以指定 *目标* 对象。目标是一个可变集合，该函数将其结果项附加到该可变对象中，而不是在新对象中返回它们。对于执行带有目标的操作，有单独的函数，其名称中带有 `To` 后缀，例如，用 `filterTo()` 代替 `filter()` 以及用 `associateTo()` 代替 `associate()`。这些函数将目标集合作为附加参数。

```
val numbers = listOf("one", "two", "three", "four")
val filterResults = mutableListOf<String>()  // 目标对象
numbers.filterTo(filterResults) { it.length > 3 }
numbers.filterIndexedTo(filterResults) { index, _ -> index == 0 }
println(filterResults) // 包含两个操作的结果
```

为了方便起见，这些函数将目标集合返回了，因此您可以在函数调用的相应参数中直接创建它：

```
// 将数字直接过滤到新的哈希集中，
// 从而消除结果中的重复项
val result = numbers.mapTo(HashSet()) { it.length }
println("distinct item lengths are $result")
```

具有目标的函数可用于过滤、关联、分组、展平以及其他操作。有关目标操作的完整列表，请参见 Kotlin collections reference。

写操作

对于可变集合，还存在可更改集合状态的 *写操作*。这些操作包括添加、删除和更新元素。写操作在集合写操作以及 List 写操作与 Map 写操作的相应部分中列出。

对于某些操作，有成对的函数可以执行相同的操作：一个函数就地应用该操作，另一个函数将结果作为单独的集合返回。例如，`sort()` 就地对可变集合进行排序，因此其状态发生了变化；`sorted()` 创建一个新集合，该集合包含按排序顺序相同的元素。

```
val numbers = mutableListOf("one", "two", "three", "four")
val sortedNumbers = numbers.sorted()
println(numbers == sortedNumbers)  // false
numbers.sort()
println(numbers == sortedNumbers)  // true
```

集合转换

Kotlin 标准库为集合 *转换* 提供了一组扩展函数。这些函数根据提供的转换规则从现有集合中构建新集合。在此页面中，我们将概述可用的集合转换函数。

映射

映射 转换从另一个集合的元素上的函数结果创建一个集合。基本的映射函数是 `map()`。它将给定的 lambda 函数应用于每个后续元素，并返回 lambda 结果列表。结果的顺序与元素的原始顺序相同。如需应用还要用到元素索引作为参数的转换，请使用 `mapIndexed()`。

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { idx, value -> value * idx })
```

如果转换在某些元素上产生 `null` 值，则可以通过调用 `mapNotNull()` 函数取代 `map()` 或 `mapIndexedNotNull()` 取代 `mapIndexed()` 来从结果集中过滤掉 `null` 值。

```
val numbers = setOf(1, 2, 3)
println(numbers.mapNotNull { if ( it == 2) null else it * 3 })
println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value * idx })
```

映射转换时，有两个选择：转换键，使值保持不变，反之亦然。要将指定转换应用于键，请使用 `mapKeys()`；反过来，`mapValues()` 转换值。这两个函数都使用将映射条目作为参数的转换，因此可以操作其键与值。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
println(numbersMap.mapKeys { it.key.toUpperCase() })
println(numbersMap.mapValues { it.value + it.key.length })
```

双路合并

双路合并 转换是根据两个集合中具有相同位置的元素构建配对。在 Kotlin 标准库中，这是通过 `zip()` 扩展函数完成的。在一个集合（或数组）上以另一个集合（或数组）作为参数调用时，`zip()` 返回 `Pair` 对象的列表（`List`）。接收者集合的元素是这些配对中的第一个元素。如果集合的大小不同，则 `zip()` 的结果为较小集合的大小；结果中不包含较大集合的后续元素。`zip()` 也可以中缀形式调用 `a zip b` 。

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")
println(colors zip animals)

val twoAnimals = listOf("fox", "bear")
println(colors.zip(twoAnimals))
```

也可以使用带有两个参数的转换函数来调用 `zip()`：接收者元素和参数元素。在这种情况下，结果 `List` 包含在具有相同位置的接收者对和参数元素对上调用的转换函数的返回值。

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")

println(colors.zip(animals) { color, animal -> "The ${animal.capitalize()} is $color"})
```

当拥有 `Pair` 的 `List` 时，可以进行反向转换 *unzipping*——从这些键值对中构建两个列表：

- 第一个列表包含原始列表中每个 `Pair` 的键。
- 第二个列表包含原始列表中每个 `Pair` 的值。

要分割键值对列表，请调用 unzip()。

```
val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
println(numberPairs.unzip())
```

关联

关联 转换允许从集合元素和与其关联的某些值构建 Map。在不同的关联类型中，元素可以是关联 Map 中的键或值。

基本的关联函数 associateWith() 创建一个 `Map`，其中原始集合的元素是键，并通过给定的转换函数从中产生值。如果两个元素相等，则仅最后一个保留在 Map 中。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
```

为了使用集合元素作为值来构建 Map，有一个函数 associateBy()。它需要一个函数，该函数根据元素的值返回键。如果两个元素相等，则仅最后一个保留在 Map 中。还可以使用值转换函数来调用 `associateBy()`。

```
val numbers = listOf("one", "two", "three", "four")

println(numbers.associateBy { it.first().toUpperCase() })
println(numbers.associateBy(keySelector = { it.first().toUpperCase() }, valueTransform = { it.length
}))
```

另一种构建 Map 的方法是使用函数 associate()，其中 Map 键和值都是通过集合元素生成的。它需要一个 lambda 函数，该函数返回 `Pair`：键和相应 Map 条目的值。

请注意，`associate()` 会生成临时的 `Pair` 对象，这可能会影响性能。因此，当性能不是很关键或比其他选项更可取时，应使用 `associate()`。

后者的一个示例：从一个元素一起生成键和相应的值。

```
val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")
println(names.associate { name -> parseFullName(name).let { it.lastName to it.firstName } })
```

此时，首先在一个元素上调用一个转换函数，然后根据该函数结果的属性建立 Pair。

打平

如需操作嵌套的集合，则可能会发现提供对嵌套集合元素进行打平访问的标准库函数很有用。

第一个函数为 flatten()。可以在一个集合的集合（例如，一个 `Set` 组成的 `List`）上调用它。该函数返回嵌套集合中的所有元素的一个 `List`。

```
val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1, 2))
println(numberSets.flatten())
```

另一个函数——flatMap() 提供了一种灵活的方式来处理嵌套的集合。它需要一个函数将一个集合元素映射到另一个集合。因此，`flatMap()` 返回单个列表其中包含所有元素的值。所以，`flatMap()` 表现为 `map()`（以集合作为映射结果）与 `flatten()` 的连续调用。

```
val containers = listOf(
    StringContainer(listOf("one", "two", "three")),
    StringContainer(listOf("four", "five", "six")),
    StringContainer(listOf("seven", "eight"))
)
println(containers.flatMap { it.values })
```

字符串表示

如果需要以可读格式检索集合内容，请使用将集合转换为字符串的函数：`joinToString()` 与 `joinTo()`。

`joinToString()` 根据提供的参数从集合元素构建单个 `String`。`joinTo()` 执行相同的操作，但将结果附加到给定的 `Appendable` 对象。

当使用默认参数调用时，函数返回的结果类似于在集合上调用 `toString()`：各元素的字符串表示形式以空格分隔而成的 `String`。

```
val numbers = listOf("one", "two", "three", "four")

println(numbers)
println(numbers.joinToString())

val listString = StringBuffer("The list of numbers: ")
numbers.joinTo(listString)
println(listString)
```

要构建自定义字符串表示形式，可以在函数参数 `separator`、`prefix` 与 `postfix` 中指定其参数。结果字符串将以 `prefix` 开头，以 `postfix` 结尾。除最后一个元素外，`separator` 将位于每个元素之后。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": end"))
```

对于较大的集合，可能需要指定 `limit` ——将包含在结果中元素的数量。如果集合大小超出 `limit`，所有其他元素将被 `truncated` 参数的单个值替换。

```
val numbers = (1..100).toList()
println(numbers.joinToString(limit = 10, truncated = "<...>"))
```

最后，要自定义元素本身的表示形式，请提供 `transform` 函数。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.joinToString { "Element: ${it.toUpperCase()}"})
```

过滤

过滤是最常用的集合处理任务之一。在Kotlin中，过滤条件由 *谓词* 定义——接受一个集合元素并且返回布尔值的 lambda 表达式：`true` 说明给定元素与谓词匹配，`false` 则表示不匹配。

标准库包含了一组让你能够通过单个调用就可以过滤集合的扩展函数。这些函数不会改变原始集合，因此它们既可用于可变集合也可用于只读集合。为了操作过滤结果，应该在过滤后将其赋值给变量或链接其他函数。

按谓词过滤

基本的过滤函数是 `filter()`。当使用一个谓词来调用时，`filter()` 返回与其匹配的集合元素。对于 `List` 和 `Set`，过滤结果都是一个 `List`，对 `Map` 来说结果还是一个 `Map`。

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
println(filteredMap)
```

`filter()` 中的谓词只能检查元素的值。如果想在过滤中使用元素在集合中的位置，应该使用 `filterIndexed()`。它接受一个带有两个参数的谓词：元素的索引和元素的值。

如果想使用否定条件来过滤集合，请使用 `filterNot()`。它返回一个让谓词产生 `false` 的元素列表。

```
val numbers = listOf("one", "two", "three", "four")

val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5)  }
val filteredNot = numbers.filterNot { it.length <= 3 }

println(filteredIdx)
println(filteredNot)
```

还有一些函数能够通过过滤给定类型的元素来缩小元素的类型：

- `filterIsInstance()` 返回给定类型的集合元素。在一个 `List<Any>` 上被调用时，`filterIsInstance<T>()` 返回一个 `List<T>`，从而让你能够在集合元素上调用 `T` 类型的函数。

```
val numbers = listOf(null, 1, "two", 3.0, "four")
println("All String elements in upper case:")
numbers.filterIsInstance<String>().forEach {
    println(it.toUpperCase())
}
```

- `filterNotNull()` 返回所有的非空元素。在一个 `List<T?>` 上被调用时，`filterNotNull()` 返回一个 `List<T: Any>`，从而让你能够将所有元素视为非空对象。

```
val numbers = listOf(null, "one", "two", null)
numbers.filterNotNull().forEach {
    println(it.length)   // 对可空的 String 来说长度不可用
}
```

划分

另一个过滤函数 – `partition()` – 通过一个谓词过滤集合并且将不匹配的元素存放在一个单独的列表中。因此，你得到一个 `List` 的 `Pair` 作为返回值：第一个列表包含与谓词匹配的元素并且第二个列表包含原始集合中的所有其他元素。

```
val numbers = listOf("one", "two", "three", "four")
val (match, rest) = numbers.partition { it.length > 3 }

println(match)
println(rest)
```

检验谓词

最后，有些函数只是针对集合元素简单地检测一个谓词：

- 如果至少有一个元素匹配给定谓词，那么 `any()` 返回 `true`。
- 如果没有元素与给定谓词匹配，那么 `none()` 返回 `true`。
- 如果所有元素都匹配给定谓词，那么 `all()` 返回 `true`。注意，在一个空集合上使用任何有效的谓词去调用 `all()` 都会返回 `true`。这种行为在逻辑上被称为 *vacuous truth*。

```
val numbers = listOf("one", "two", "three", "four")

println(numbers.any { it.endsWith("e") })
println(numbers.none { it.endsWith("a") })
println(numbers.all { it.endsWith("e") })

println(emptyList<Int>().all { it > 5 })   // vacuous truth
```

`any()` 和 `none()` 也可以不带谓词使用：在这种情况下它们只是用来检查集合是否为空。如果集合中有元素，`any()` 返回 `true`，否则返回 `false`；`none()` 则相反。

```
val numbers = listOf("one", "two", "three", "four")
val empty = emptyList<String>()

println(numbers.any())
println(empty.any())

println(numbers.none())
println(empty.none())
```

`plus` 与 `minus` 操作符

在 Kotlin 中，为集合定义了 `plus`（`+`）和 `minus`（`-`）操作符。它们把一个集合作为第一个操作数；第二个操作数可以是一个元素或者是另一个集合。返回值是一个新的只读集合：

- `plus` 的结果包含原始集合 *和* 第二个操作数中的元素。
- `minus` 的结果包含原始集合中的元素，但第二个操作数中的元素 *除外*。如果第二个操作数是一个元素，那么 `minus` 移除其在原始集合中的 *第一次* 出现；如果是一个集合，那么移除其元素在原始集合中的 *所有* 出现。

```
val numbers = listOf("one", "two", "three", "four")

val plusList = numbers + "five"
val minusList = numbers - listOf("three", "four")
println(plusList)
println(minusList)
```

有关 map 的 `plus` 和 `minus` 操作符的详细信息，请参见 Map 相关操作。也为集合定义了广义赋值操作符 `plusAssign`（`+=`）和 `minusAssign`（`-=`）。然而，对于只读集合，它们实际上使用 `plus` 或者 `minus` 操作符并尝试将结果赋值给同一变量。因此，它们仅在由 `var` 声明的只读集合中可用。对于可变集合，如果它是一个 `val`，那么它们会修改集合。更多详细信息请参见集合写操作。

分组

Kotlin 标准库提供用于对集合元素进行分组的扩展函数。基本函数 `groupBy()` 使用一个 lambda 函数并返回一个 `Map`。在此 Map 中，每个键都是 lambda 结果，而对应的值是返回此结果的元素 `List`。例如，可以使用此函数将 `String` 列表按首字母分组。

还可以使用第二个 lambda 参数（值转换函数）调用 `groupBy()`。在带有两个 lambda 的 `groupBy()` 结果 Map 中，由 `keySelector` 函数生成的键映射到值转换函数的结果，而不是原始元素。

```
val numbers = listOf("one", "two", "three", "four", "five")

println(numbers.groupBy { it.first().toUpperCase() })
println(numbers.groupBy(keySelector = { it.first() }, valueTransform = { it.toUpperCase() }))
```

如果要对元素进行分组，然后一次将操作应用于所有分组，请使用 `groupingBy()` 函数。它返回一个 `Grouping` 类型的实例。通过 `Grouping` 实例，可以以一种惰性的方式将操作应用于所有组：这些分组实际上是刚好在执行操作前构建的。

即，`Grouping` 支持以下操作：

- `eachCount()` 计算每个组中的元素。
- `fold()` 与 `reduce()` 对每个组分别执行 fold 与 reduce 操作，作为一个单独的集合并返回结果。
- `aggregate()` 随后将给定操作应用于每个组中的所有元素并返回结果。这是对 `Grouping` 执行任何操作的通用方法。当折叠或缩小不够时，可使用它来实现自定义操作。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.groupingBy { it.first() }.eachCount())
```

取集合的一部分

Kotlin 标准库包含用于取集合的一部分的扩展函数。这些函数提供了多种方法来选择结果集合的元素：显式列出其位置、指定结果大小等。

Slice

slice() 返回具有给定索引的集合元素列表。索引既可以是作为区间传入的也可以是作为整数值的集合传入的。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.slice(1..3))
println(numbers.slice(0..4 step 2))
println(numbers.slice(setOf(3, 5, 0)))
```

Take 与 drop

要从头开始获取指定数量的元素，请使用 take() 函数。要从尾开始获取指定数量的元素，请使用 takeLast()。当调用的数字大于集合的大小时，两个函数都将返回整个集合。

要从头或从尾去除给定数量的元素，请调用 drop() 或 dropLast() 函数。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.take(3))
println(numbers.takeLast(3))
println(numbers.drop(1))
println(numbers.dropLast(5))
```

还可以使用谓词来定义要获取或去除的元素的数量。有四个与上述功能相似的函数：

- takeWhile() 是带有谓词的 `take()`：它将不停获取元素直到排除与谓词匹配的首个元素。如果首个集合元素与谓词匹配，则结果为空。
- takeLastWhile() 与 `takeLast()` 类似：它从集合末尾获取与谓词匹配的元素区间。区间的首个元素是与谓词不匹配的最后一个元素右边的元素。如果最后一个集合元素与谓词匹配，则结果为空。
- dropWhile() 与具有相同谓词的 `takeWhile()` 相反：它将首个与谓词不匹配的元素返回到末尾。
- dropLastWhile() 与具有相同谓词的 `takeLastWhile()` 相反：它返回从开头到最后一个与谓词不匹配的元素。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.takeWhile { !it.startsWith('f') })
println(numbers.takeLastWhile { it != "three" })
println(numbers.dropWhile { it.length == 3 })
println(numbers.dropLastWhile { it.contains('i') })
```

Chunked

要将集合分解为给定大小的“块”，请使用 chunked() 函数。`chunked()` 采用一个参数（块的大小），并返回一个 `List` 其中包含给定大小的 `List`。第一个块从第一个元素开始并包含 `size` 元素，第二个块包含下一个 `size` 元素，依此类推。最后一个块的大小可能较小。

```
val numbers = (0..13).toList()
println(numbers.chunked(3))
```

还可以立即对返回的块应用转换。为此，请在调用 `chunked()` 时将转换作为 lambda 函数提供。lambda 参数是集合的一块。当通过转换调用 `chunked()` 时，这些块是临时的 `List`，应立即在该 lambda 中使用。

```
val numbers = (0..13).toList()
println(numbers.chunked(3) { it.sum() })  // `it` 为原始集合的一个块
```

Windowed

可以检索给定大小的集合元素中所有可能区间。获取它们的函数称为 `windowed()`：它返回一个元素区间列表，比如通过给定大小的滑动窗口查看集合，则会看到该区间。与 `chunked()` 不同，`windowed()` 返回从*每个*集合元素开始的元素区间（*窗口*）。所有窗口都作为单个 `List` 的元素返回。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.windowed(3))
```

`windowed()` 通过可选参数提供更大的灵活性：

- `step` 定义两个相邻窗口的第一个元素之间的距离。默认情况下，该值为 1，因此结果包含从所有元素开始的窗口。如果将 step 增加到 2，将只收到以奇数元素开头的窗口：第一个、第三个等。
- `partialWindows` 包含从集合末尾的元素开始的较小的窗口。例如，如果请求三个元素的窗口，就不能为最后两个元素构建它们。在本例中，启用 `partialWindows` 将包括两个大小为2与1的列表。

最后，可以立即对返回的区间应用转换。为此，在调用 `windowed()` 时将转换作为 lambda 函数提供。

```
val numbers = (1..10).toList()
println(numbers.windowed(3, step = 2, partialWindows = true))
println(numbers.windowed(3) { it.sum() })
```

要构建两个元素的窗口，有一个单独的函数——`zipWithNext()`。它创建接收器集合的相邻元素对。请注意，`zipWithNext()` 不会将集合分成几对；它为*每个*元素创建除最后一个元素外的对，因此它在 `[1, 2, 3, 4]` 上的结果为 `[[1, 2], [2, 3], [3, 4]]`，而不是 `[[1, 2], [3, 4]]`。`zipWithNext()` 也可以通过转换函数来调用；它应该以接收者集合的两个元素作为参数。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.zipWithNext())
println(numbers.zipWithNext() { s1, s2 -> s1.length > s2.length})
```

取单个元素

Kotlin 集合提供了一套从集合中检索单个元素的函数。此页面描述的函数适用于 list 和 set。

正如 list 的定义所言，list 是有序集合。因此，list 中的每个元素都有其位置可供你引用。除了此页面上描述的函数外，list 还提供了更广泛的一套方法去按索引检索和搜索元素。有关更多详细信息，请参见 List 相关操作。

反过来，从定义来看，set 并不是有序集合。但是，Kotlin 中的 `Set` 按某些顺序存储元素。这些可以是插入顺序（在 `LinkedHashSet` 中）、自然排序顺序（在 `SortedSet` 中）或者其他顺序。一组元素的顺序也可以是未知的。在这种情况下，元素仍会以某种顺序排序，因此，依赖元素位置的函数仍会返回其结果。但是，除非调用者知道所使用的 `Set` 的具体实现，否则这些结果对于调用者是不可预测的。

按位置取

为了检索特定位置的元素，有一个函数 `elementAt()`。用一个整数作为参数来调用它，你会得到给定位置的集合元素。第一个元素的位置是 `0`，最后一个元素的位置是 `(size - 1)`。

`elementAt()` 对于不提供索引访问或非静态已知提供索引访问的集合很有用。在使用 `List` 的情况下，使用索引访问操作符（`get()` 或 `[]`）更为习惯。

```
val numbers = linkedSetOf("one", "two", "three", "four", "five")
println(numbers.elementAt(3))

val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
println(numbersSortedSet.elementAt(0)) // 元素以升序存储
```

还有一些有用的别名来检索集合的第一个和最后一个元素：`first()` 和 `last()`。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.first())
println(numbers.last())
```

为了避免在检索位置不存在的元素时出现异常，请使用 `elementAt()` 的安全变体：

- 当指定位置超出集合范围时，`elementAtOrNull()` 返回 null。
- `elementAtOrElse()` 还接受一个 lambda 表达式，该表达式能将一个 `Int` 参数映射为一个集合元素类型的实例。当使用一个越界位置来调用时，`elementAtOrElse()` 返回对给定值调用该 lambda 表达式的结果。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.elementAtOrNull(5))
println(numbers.elementAtOrElse(5) { index -> "The value for index $index is undefined"})
```

按条件取

函数 `first()` 和 `last()` 还可以让你在集合中搜索与给定谓词匹配的元素。当你使用测试集合元素的谓词调用 `first()` 时，你会得到对其调用谓词产生 `true` 的第一个元素。反过来，带有一个谓词的 `last()` 返回与其匹配的最后一个元素。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.first { it.length > 3 })
println(numbers.last { it.startsWith("f") })
```

如果没有元素与谓词匹配，两个函数都会抛异常。为了避免它们，请改用 `firstOrNull()` 和 `lastOrNull()`：如果找不到匹配的元素，它们将返回 `null`。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.firstOrNull { it.length > 6 })
```

或者，如果别名更适合你的情况，那么可以使用别名：

— 使用 find() 代替 `firstOrNull()`

— 使用 findLast() 代替 `lastOrNull()`

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.find { it % 2 == 0 })
println(numbers.findLast { it % 2 == 0 })
```

随机取元素

如果需要检索集合的一个随机元素，那么请调用 random() 函数。你可以不带参数或者使用一个 Random 对象作为随机源来调用它。

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.random())
```

检测存在与否

如需检查集合中某个元素的存在，可以使用 contains() 函数。如果存在一个集合元素等于（`equals()`）函数参数，那么它返回 `true`。你可以使用 `in` 关键字以操作符的形式调用 `contains()`。

如需一次检查多个实例的存在，可以使用这些实例的集合作为参数调用 containsAll()。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.contains("four"))
println("zero" in numbers)

println(numbers.containsAll(listOf("four", "two")))
println(numbers.containsAll(listOf("one", "zero")))
```

此外，你可以通过调用 isEmpty() 和 isNotEmpty() 来检查集合中是否包含任何元素。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.isEmpty())
println(numbers.isNotEmpty())

val empty = emptyList<String>()
println(empty.isEmpty())
println(empty.isNotEmpty())
```

集合排序

元素的顺序是某些集合类型的一个重要方面。例如，如果拥有相同元素的两个列表的元素顺序不同，那么这两个列表也不相等。

在 Kotlin 中，可以通过多种方式定义对象的顺序。

首先，有 *自然* 顺序。它是为 `Comparable` 接口的继承者定义的。当没有指定其他顺序时，使用自然顺序为它们排序。

大多数内置类型是可比较的：

— 数值类型使用传统的数值顺序：`1` 大于 `0`；`-3.4f` 大于 `-5f`，以此类推。

— `Char` 和 `String` 使用字典顺序：`b` 大于 `a`；`world` 大于 `hello`。

如需为用户定义的类型定义一个自然顺序，可以让这个类型继承 `Comparable`。这需要实现 `compareTo()` 函数。`compareTo()` 必须将另一个具有相同类型的对象作为参数并返回一个整数值来显示哪个对象更大：

— 正值表明接收者对象更大。

— 负值表明它小于参数。

— 0 说明对象相等。

下面是一个类，可用于排序由主版本号和次版本号两部分组成的版本。

```
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        } else if (this.minor != other.minor) {
            return this.minor - other.minor
        } else return 0
    }
}

fun main() {
    println(Version(1, 2) > Version(1, 3))
    println(Version(2, 0) > Version(1, 5))
}
```

自定义 顺序让你可以按自己喜欢的方式对任何类型的实例进行排序。特别是，你可以为不可比较类型定义顺序，或者为可比较类型定义非自然顺序。如需为类型定义自定义顺序，可以为其创建一个 `Comparator`。`Comparator` 包含 `compare()` 函数：它接受一个类的两个实例并返回它们之间比较的整数结果。如上所述，对结果的解释与 `compareTo()` 的结果相同。

```
val lengthComparator = Comparator { str1: String, str2: String -> str1.length - str2.length }
println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))
```

有了 `lengthComparator`，你可以按照字符串的长度而不是默认的字典顺序来排列字符串。

定义一个 `Comparator` 的一种比较简短的方式是标准库中的 `compareBy()` 函数。`compareBy()` 接受一个 lambda 表达式，该表达式从一个实例产生一个 `Comparable` 值，并将自定义顺序定义为生成值的自然顺序。使用 `compareBy()`，上面示例中的长度比较器如下所示：

```
tln(listOf("aaa", "bb", "c").sortedWith(compareBy { it.length }))
```

Kotlin 集合包提供了用于按照自然顺序、自定义顺序甚至随机顺序对集合排序的函数。在此页面上，我们将介绍适用于只读集合的排序函数。这些函数将它们的结果作为一个新集合返回，集合里包含了按照请求顺序排序的来自原始集合的元素。如果想学习就地对可变集合排序的函数，请参见 List 相关操作。

自然顺序

基本的函数 `sorted()` 和 `sortedDescending()` 返回集合的元素，这些元素按照其自然顺序升序和降序排序。这些函数适用于 `Comparable` 元素的集合。

```
val numbers = listOf("one", "two", "three", "four")

println("Sorted ascending: ${numbers.sorted()}")
println("Sorted descending: ${numbers.sortedDescending()}")
```

自定义顺序

为了按照自定义顺序排序或者对不可比较对象排序，可以使用函数 `sortedBy()` 和 `sortedByDescending()`。它们接受一个将集合元素映射为 `Comparable` 值的选择器函数，并以该值的自然顺序对集合排序。

```
val numbers = listOf("one", "two", "three", "four")

val sortedNumbers = numbers.sortedBy { it.length }
println("Sorted by length ascending: $sortedNumbers")
val sortedByLast = numbers.sortedByDescending { it.last() }
println("Sorted by the last letter descending: $sortedByLast")
```

如需为集合排序定义自定义顺序，可以提供自己的 `Comparator`。为此，调用传入 `Comparator` 的 `sortedWith()` 函数。使用此函数，按照字符串长度排序如下所示：

```
val numbers = listOf("one", "two", "three", "four")
println("Sorted by length ascending: ${numbers.sortedWith(compareBy { it.length })}")
```

倒序

你可以使用 `reversed()` 函数以相反的顺序检索集合。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.reversed())
```

`reversed()` 返回带有元素副本的新集合。因此，如果你之后改变了原始集合，这并不会影响先前获得的 `reversed()` 的结果。

另一个反向函数——`asReversed()`——返回相同集合实例的一个反向视图，因此，如果原始列表不会发生变化，那么它会比 `reversed()` 更轻量，更合适。

```
val numbers = listOf("one", "two", "three", "four")
val reversedNumbers = numbers.asReversed()
println(reversedNumbers)
```

如果原始列表是可变的，那么其所有更改都会反映在其反向视图中，反之亦然。

```
val numbers = mutableListOf("one", "two", "three", "four")
val reversedNumbers = numbers.asReversed()
println(reversedNumbers)
numbers.add("five")
println(reversedNumbers)
```

但是，如果列表的可变性未知或者源根本不是一个列表，那么 `reversed()` 更合适，因为其结果是一个未来不会更改的副本。

随机顺序

最后，`shuffled()` 函数返回一个包含了以随机顺序排序的集合元素的新的 `List`。你可以不带参数或者使用 `Random` 对象来调用它。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.shuffled())
```

集合聚合操作

Kotlin 集合包含用于常用的 *聚合操作*（基于集合内容返回单个值的操作）的函数。其中大多数是众所周知的，并且其工作方式与在其他语言中相同。

- `min()` 与 `max()` 分别返回最小和最大的元素；
- `average()` 返回数字集合中元素的平均值；
- `sum()` 返回数字集合中元素的总和；
- `count()` 返回集合中元素的数量；

```
val numbers = listOf(6, 42, 10, 4)

println("Count: ${numbers.count()}")
println("Max: ${numbers.max()}")
println("Min: ${numbers.min()}")
println("Average: ${numbers.average()}")
println("Sum: ${numbers.sum()}")
```

还有一些通过某些选择器函数或自定义 `Comparator` 来检索最小和最大元素的函数。

- `maxBy()`/`minBy()` 接受一个选择器函数并返回使选择器返回最大或最小值的元素。
- `maxWith()`/`minWith()` 接受一个 `Comparator` 对象并且根据此 `Comparator` 对象返回最大或最小元素。

```
val numbers = listOf(5, 42, 10, 4)
val min3Remainder = numbers.minBy { it % 3 }
println(min3Remainder)

val strings = listOf("one", "two", "three", "four")
val longestString = strings.maxWith(compareBy { it.length })
println(longestString)
```

此外，有一些高级的求和函数，它们接受一个函数并返回对所有元素调用此函数的返回值的总和：

- `sumBy()` 使用对集合元素调用返回 `Int` 值的函数。
- `sumByDouble()` 与返回 `Double` 的函数一起使用。

```

val numbers = listOf(5, 42, 10, 4)
println(numbers.sumBy { it * 2 })
println(numbers.sumByDouble { it.toDouble() / 2 })
```

Fold 与 reduce

对于更特定的情况，有函数 `reduce()` 和 `fold()`，它们依次将所提供的操作应用于集合元素并返回累积的结果。操作有两个参数：先前的累积值和集合元素。

这两个函数的区别在于：`fold()` 接受一个初始值并将其用作第一步的累积值，而 `reduce()` 的第一步则将第一个和第二个元素作为第一步的操作参数。

```
val numbers = listOf(5, 2, 10, 4)

val sum = numbers.reduce { sum, element -> sum + element }
println(sum)
val sumDoubled = numbers.fold(0) { sum, element -> sum + element * 2 }
println(sumDoubled)

//val sumDoubledReduce = numbers.reduce { sum, element -> sum + element * 2 } //错误：第一个元素在结果中没有加倍
//println(sumDoubledReduce)
```

上面的实例展示了区别：`fold()` 用于计算加倍的元素之和。如果将相同的函数传给 `reduce()`，那么它会返回另一个结果，因为在第一步中它将列表的第一个和第二个元素作为参数，所以第一个元素不会被加倍。

如需将函数以相反的顺序应用于元素，可以使用函数 `reduceRight()` 和 `foldRight()` 它们的工作方式类似于 `fold()` 和 `reduce()`，但从最后一个元素开始，然后再继续到前一个元素。记住，在使用 foldRight 或 reduceRight 时，操作参数会更改其顺序：第一个参数变为元素，然后第二个参数变为累积值。

```
val numbers = listOf(5, 2, 10, 4)
val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum + element * 2 }
println(sumDoubledRight)
```

你还可以使用将元素索引作为参数的操作。为此，使用函数 `reduceIndexed()` 和 `foldIndexed()` 传递元素索引作为操作的第一个参数。

最后，还有将这些操作从右到左应用于集合元素的函数——`reduceRightIndexed()` 与 `foldRightIndexed()`。

```
val numbers = listOf(5, 2, 10, 4)
val sumEven = numbers.foldIndexed(0) { idx, sum, element -> if (idx % 2 == 0) sum + element else sum }
println(sumEven)

val sumEvenRight = numbers.foldRightIndexed(0) { idx, element, sum -> if (idx % 2 == 0) sum + element
else sum }
println(sumEvenRight)
```

集合写操作

可变集合支持更改集合内容的操作，例如添加或删除元素。在此页面上，我们将描述实现 `MutableCollection` 的所有写操作。有关 `List` 与 `Map` 可用的更多特定操作，请分别参见 List 相关操作与 Map 相关操作。

添加元素

要将单个元素添加到列表或集合，请使用 `add()` 函数。指定的对象将添加到集合的末尾。

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
println(numbers)
```

`addAll()` 将参数对象的每个元素添加到列表或集合中。参数可以是 `Iterable`、`Sequence` 或 `Array`。接收者的类型和参数可能不同，例如，你可以将所有内容从 `Set` 添加到 `List`。

当在列表上调用时，`addAll()` 会按照在参数中出现的顺序添加各个新元素。你也可以调用 `addAll()` 时指定一个元素位置作为第一参数。参数集合的第一个元素会被插入到这个位置。其他元素将跟随在它后面，将接收者元素移到末尾。

```
val numbers = mutableListOf(1, 2, 5, 6)
numbers.addAll(arrayOf(7, 8))
println(numbers)
numbers.addAll(2, setOf(3, 4))
println(numbers)
```

你还可以使用 plus 运算符 - `plusAssign`（`+=`）添加元素。当应用于可变集合时，`+=` 将第二个操作数(一个元素或另一个集合)追加到集合的末尾。

```
val numbers = mutableListOf("one", "two")
numbers += "three"
println(numbers)
numbers += listOf("four", "five")
println(numbers)
```

删除元素

若要从可变集合中移除元素，请使用 `remove()` 函数。`remove()` 接受元素值，并删除该值的一个匹配项。

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.remove(3)                    // 删除了第一个 `3`
println(numbers)
numbers.remove(5)                    // 什么都没删除
println(numbers)
```

要一次删除多个元素，有以下函数：

- `removeAll()` 移除参数集合中存在的所有元素。或者，你可以用谓词作为参数来调用它；在这种情况下，函数移除谓词产生 `true` 的所有元素。
- `retainAll()` 与 `removeAll()` 相反：它移除除参数集合中的元素之外的所有元素。当与谓词一起使用时，它只留下与之匹配的元素。
- `clear()` 从列表中移除所有元素并将其置空。

```
val numbers = mutableListOf(1, 2, 3, 4)
println(numbers)
numbers.retainAll { it >= 3 }
println(numbers)
numbers.clear()
println(numbers)

val numbersSet = mutableSetOf("one", "two", "three", "four")
numbersSet.removeAll(setOf("one", "two"))
println(numbersSet)
```

从集合中移除元素的另一种方法是使用 `minusAssign` (`-=`) ——原地修改版的 `minus` 操作符。`minus` 操作符。第二个参数可以是元素类型的单个实例或另一个集合。右边是单个元素时，`-=` 会移除它的第一个匹配项。反过来，如果它是一个集合，那么它的所有元素的每次出现都会删除。例如，如果列表包含重复的元素，它们将被同时删除。第二个操作数可以包含集合中不存在的元素。这些元素不会影响操作的执行。

```
val numbers = mutableListOf("one", "two", "three", "three", "four")
numbers -= "three"
println(numbers)
numbers -= listOf("four", "five")
//numbers -= listOf("four")    // 与上述相同
println(numbers)
```

更新元素

list 和 map 还提供更新元素的操作。它们在 List 相关操作与 Map 相关操作中有所描述。对于 set 来说，更新没有意义，因为它实际上是移除一个元素并添加另一个元素。

List 相关操作

List 是 Kotlin 标准库中最受欢迎的集合类型。对列表元素的索引访问为 List 提供了一组强大的操作。

按索引取元素

List 支持按索引取元素的所有常用操作：`elementAt()`、`first()`、`last()` 与取单个元素中列出的其他操作。List 的特点是能通过索引访问特定元素，因此读取元素的最简单方法是按索引检索它。这是通过 `get()` 函数或简写语法 `[index]` 来传递索引参数完成的。

如果 List 长度小于指定的索引，则抛出异常。另外，还有两个函数能避免此类异常：

— `getOrElse()` 提供用于计算默认值的函数，如果集合中不存在索引，则返回默认值。

— `getOrNull()` 返回 `null` 作为默认值。

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.get(0))
println(numbers[0])
//numbers.get(5)                        // exception!
println(numbers.getOrNull(5))             // null
println(numbers.getOrElse(5, {it}))       // 5
```

取列表的一部分

除了取集合的一部分中常用的操作，List 还提供 `subList()` 该函数将指定元素范围的视图作为列表返回。因此，如果原始集合的元素发生变化，则它在先前创建的子列表中也会发生变化，反之亦然。

```
val numbers = (0..13).toList()
println(numbers.subList(3, 6))
```

查找元素位置

线性查找

在任何列表中，都可以使用 `indexOf()` 或 `lastIndexOf()` 函数找到元素的位置。它们返回与列表中给定参数相等的元素的第一个或最后一个位置。如果没有这样的元素，则两个函数均返回 `-1`。

```
val numbers = listOf(1, 2, 3, 4, 2, 5)
println(numbers.indexOf(2))
println(numbers.lastIndexOf(2))
```

还有一对函数接受谓词并搜索与之匹配的元素：

— `indexOfFirst()` 返回与谓词匹配的*第一个元素的索引*，如果没有此类元素，则返回 `-1`。

— `indexOfLast()` 返回与谓词匹配的*最后一个元素的索引*，如果没有此类元素，则返回 `-1`。

```
val numbers = mutableListOf(1, 2, 3, 4)
println(numbers.indexOfFirst { it > 2})
println(numbers.indexOfLast { it % 2 == 1})
```

在有序列表中二分查找

还有另一种搜索列表中元素的方法——二分查找算法。它的工作速度明显快于其他内置搜索功能，但*要求该列表按照一定的顺序*（自然排序或函数参数中提供的另一种排序）按升序排序过。否则，结果是不确定的。

要搜索已排序列表中的元素，请调用 `binarySearch()` 函数，并将该值作为参数传递。如果存在这样的元素，则函数返回其索引；否则，将返回 `(-insertionPoint - 1)`，其中 `insertionPoint` 为应插入此元素的索引，以便列表保持排序。如果有多个具有给定值的元素，搜索则可以返回其任何索引。

还可以指定要搜索的索引区间：在这种情况下，该函数仅在两个提供的索引之间搜索。

```
val numbers = mutableListOf("one", "two", "three", "four")
numbers.sort()
println(numbers)
println(numbers.binarySearch("two"))  // 3
println(numbers.binarySearch("z")) // -5
println(numbers.binarySearch("two", 0, 2))  // -3
```

Comparator 二分搜索

如果列表元素不是 `Comparable`，则应提供一个用于二分搜索的 `Comparator`。该列表必须根据此 `Comparator` 以升序排序。来看一个例子：

```
val productList = listOf(
    Product("WebStorm", 49.0),
    Product("AppCode", 99.0),
    Product("DotTrace", 129.0),
    Product("ReSharper", 149.0))

println(productList.binarySearch(Product("AppCode", 99.0), compareBy<Product> { it.price }.thenBy {
it.name }))
```

这是一个不可 `排序` 的 `Product` 实例列表，以及一个定义排序的 `Comparator`：如果 `p1` 的价格小于 `p2` 的价格，则产品 `p1` 在产品 `p2` 之前。因此，按照此顺序对列表进行升序排序后，使用 `binarySearch()` 查找指定的 `Product` 的索引。

当列表使用与自然排序不同的顺序时（例如，对 `String` 元素不区分大小写的顺序），自定义 Comparator 也很方便。

```
val colors = listOf("Blue", "green", "ORANGE", "Red", "yellow")
println(colors.binarySearch("RED", String.CASE_INSENSITIVE_ORDER)) // 3
```

比较函数二分搜索

使用 *比较* 函数的二分搜索无需提供明确的搜索值即可查找元素。取而代之的是，它使用一个比较函数将元素映射到 `Int` 值，并搜索函数返回 0 的元素。该列表必须根据提供的函数以升序排序；换句话说，比较的返回值必须从一个列表元素增长到下一个列表元素。

```
data class Product(val name: String, val price: Double)

fun priceComparison(product: Product, price: Double) = sign(product.price - price).toInt()

fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch { priceComparison(it, 99.0) })
}
```

Comparator 与比较函数二分搜索都可以针对列表区间执行。

List 写操作

除了集合写操作中描述的集合修改操作之外，可变列表还支持特定的写操作。这些操作使用索引来访问元素以扩展列表修改功能。

添加

要将元素添加到列表中的特定位置，请使用 `add()` 或 `addAll()` 并提供元素插入的位置作为附加参数。位置之后的所有元素都将向右移动。

```
val numbers = mutableListOf("one", "five", "six")
numbers.add(1, "two")
numbers.addAll(2, listOf("three", "four"))
println(numbers)
```

更新

列表还提供了在指定位置替换元素的函数——`set()` 及其操作符形式 `[]`。`set()` 不会更改其他元素的索引。

```
val numbers = mutableListOf("one", "five", "three")
numbers[1] =  "two"
println(numbers)
```

`fill()` 简单地将所有集合元素的值替换为指定值。

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.fill(3)
println(numbers)
```

删除

要从列表中删除指定位置的元素，请使用 `removeAt()` 函数，并将位置作为参数。在元素被删除之后出现的所有元素索引将减 1。

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.removeAt(1)
println(numbers)
```

排序

在集合排序中，描述了按特定顺序检索集合元素的操作。对于可变列表，标准库中提供了类似的扩展函数，这些扩展函数可以执行相同的排序操作。将此类操作应用于列表实例时，它将更改指定实例中元素的顺序。

就地排序函数的名称与应用于只读列表的函数的名称相似，但没有 `ed/d` 后缀：

- `sort*` 在所有排序函数的名称中代替 `sorted*`：`sort()`、`sortDescending()`、`sortBy()` 等等。
- `shuffle()` 代替 `shuffled()`。
- `reverse()` 代替 `reversed()`。

`asReversed()` 在可变列表上调用会返回另一个可变列表，该列表是原始列表的反向视图。在该视图中的更改将反映在原始列表中。以下示例展示了可变列表的排序函数：

```
val numbers = mutableListOf("one", "two", "three", "four")

numbers.sort()
println("Sort into ascending: $numbers")
numbers.sortDescending()
println("Sort into descending: $numbers")

numbers.sortBy { it.length }
println("Sort into ascending by length: $numbers")
numbers.sortByDescending { it.last() }
println("Sort into descending by the last letter: $numbers")

numbers.sortWith(compareBy<String> { it.length }.thenBy { it })
println("Sort by Comparator: $numbers")

numbers.shuffle()
println("Shuffle: $numbers")

numbers.reverse()
println("Reverse: $numbers")
```

Set 相关操作

Kotlin 集合包中包含 set 常用操作的扩展函数：查找交集、并集或差集。

要将两个集合合并为一个（并集），可使用 `union()` 函数。也能以中缀形式使用 `a union b`。注意，对于有序集合，操作数的顺序很重要：在结果集合中，左侧操作数在前。

要查找两个集合中都存在的元素（交集），请使用 `intersect()`。要查找另一个集合中不存在的集合元素（差集），请使用 `subtract()`。这两个函数也能以中缀形式调用，例如，`a intersect b`。

```
val numbers = setOf("one", "two", "three")

println(numbers union setOf("four", "five"))
println(setOf("four", "five") union numbers)

println(numbers intersect setOf("two", "one"))
println(numbers subtract setOf("three", "four"))
println(numbers subtract setOf("four", "three")) // 相同的输出
```

注意，`List` 也支持 Set 操作。但是，对 List 进行 Set 操作的结果仍然是 `Set`，因此将删除所有重复的元素。

Map 相关操作

在 map 中，键和值的类型都是用户定义的。对基于键的访问启用了各种特定于 map 的处理函数，从键获取值到对键和值进行单独过滤。在此页面上，我们提供了来自标准库的 map 处理功能的描述。

取键与值

要从 Map 中检索值，必须提供其键作为 `get()` 函数的参数。还支持简写 `[key]` 语法。如果找不到给定的键，则返回 `null` 。还有一个函数 `getValue()` ，它的行为略有不同：如果在 Map 中找不到键，则抛出异常。此外，还有两个选项可以解决键缺失的问题：

- `getOrElse()` 与 list 的工作方式相同：对于不存在的键，其值由给定的 lambda 表达式返回。
- `getOrDefault()` 如果找不到键，则返回指定的默认值。

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap.get("one"))
println(numbersMap["one"])
println(numbersMap.getOrDefault("four", 10))
println(numbersMap["five"])               // null
//numbersMap.getValue("six")      // exception!
```

要对 map 的所有键或所有值执行操作，可以从属性 `keys` 和 `values` 中相应地检索它们。`keys` 是 Map 中所有键的集合，`values` 是 Map 中所有值的集合。

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap.keys)
println(numbersMap.values)
```

过滤

可以使用 `filter()` 函数来过滤 map 或其他集合。对 map 使用 `filter()` 函数时，`Pair` 将作为参数的谓词传递给它。它将使用谓词同时过滤其中的键和值。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
println(filteredMap)
```

还有两种用于过滤 map 的特定函数：按键或按值。这两种方式，都有对应的函数：`filterKeys()` 和 `filterValues()` 。两者都将返回一个新 Map ，其中包含与给定谓词相匹配的条目。`filterKeys()` 的谓词仅检查元素键，`filterValues()` 的谓词仅检查值。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
val filteredValuesMap = numbersMap.filterValues { it < 10 }

println(filteredKeysMap)
println(filteredValuesMap)
```

`plus` 与 `minus` 操作

由于需要访问元素的键，`plus`（`+`）与 `minus`（`-`）运算符对 map 的作用与其他集合不同。`plus` 返回包含两个操作数元素的 `Map` ：左侧的 Map 与右侧的 Pair 或另一个 Map 。当右侧操作数中有左侧 `Map` 中已存在的键时，该条目将使用右侧的值。

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap + Pair("four", 4))
println(numbersMap + Pair("one", 10))
println(numbersMap + mapOf("five" to 5, "one" to 11))
```

`minus` 将根据左侧 `Map` 条目创建一个新 `Map`，右侧操作数带有键的条目将被剔除。因此，右侧操作数可以是单个键或键的集合：list、set 等。

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap - "one")
println(numbersMap - listOf("two", "four"))
```

关于在可变 Map 中使用 `plusAssign`（`+=`）与 `minusAssign`（`-=`）运算符的详细信息，请参见 Map 写操作。

Map 写操作

Mutable Map（可变 Map）提供特定的 Map 写操作。这些操作使你可以使用键来访问或更改 Map 值。

Map 写操作的一些规则：

- 值可以更新。反过来，键也永远不会改变：添加条目后，键是不变的。
- 每个键都有一个与之关联的值。也可以添加和删除整个条目。

下面是对可变 Map 中可用写操作的标准库函数的描述。

添加与更新条目

要将新的键值对添加到可变 Map，请使用 `put()`。将新条目放入 `LinkedHashMap`（Map的默认实现）后，会添加该条目，以便在 Map 迭代时排在最后。在 Map 类中，新元素的位置由其键顺序定义。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
println(numbersMap)
```

要一次添加多个条目，请使用 `putAll()`。它的参数可以是 `Map` 或一组 `Pair`：`Iterable`、`Sequence` 或 `Array`。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
numbersMap.putAll(setOf("four" to 4, "five" to 5))
println(numbersMap)
```

如果给定键已存在于 Map 中，则 `put()` 与 `putAll()` 都将覆盖值。因此，可以使用它们来更新 Map 条目的值。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
val previousValue = numbersMap.put("one", 11)
println("value associated with 'one', before: $previousValue, after: ${numbersMap["one"]}")
println(numbersMap)
```

还可以使用快速操作符将新条目添加到 Map。有两种方式：

- `plusAssign`（`+=`）操作符。
- `[]` 操作符为 `put()` 的别名。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap["three"] = 3     // 调用 numbersMap.put("three", 3)
numbersMap += mapOf("four" to 4, "five" to 5)
println(numbersMap)
```

使用 Map 中存在的键进行操作时，将覆盖相应条目的值。

删除条目

要从可变 Map 中删除条目，请使用 `remove()` 函数。调用 `remove()` 时，可以传递键或整个键值对。如果同时指定键和值，则仅当键值都匹配时，才会删除此的元素。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
numbersMap.remove("one")
println(numbersMap)
numbersMap.remove("three", 4)            //不会删除任何条目
println(numbersMap)
```

还可以通过键或值从可变 Map 中删除条目。在 Map 的 `.keys` 或 `.values` 中调用 `remove()` 并提供键或值来删除条目。在 `.values` 中调用时，`remove()` 仅删除给定值匹配到的的第一个条目。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain" to 3)
numbersMap.keys.remove("one")
println(numbersMap)
numbersMap.values.remove(3)
println(numbersMap)
```

`minusAssign`（`-=`）操作符也可用于可变 Map 。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
numbersMap -= "two"
println(numbersMap)
numbersMap -= "five"             //不会删除任何条目
println(numbersMap)
```

协程

Kotlin 是一门仅在标准库中提供最基本底层 API 以便各种其他库能够利用协程的语言。与许多其他具有类似功能的语言不同，`async` 与 `await` 在 Kotlin 中并不是关键字，甚至都不是标准库的一部分。此外，Kotlin 的 *挂起函数* 概念为异步操作提供了比 future 与 promise 更安全、更不易出错的抽象。

`kotlinx.coroutines` 是由 JetBrains 开发的功能丰富的协程库。它包含本指南中涵盖的很多启用高级协程的原语，包括 `launch`、`async` 等等。

本文是关于 `kotlinx.coroutines` 核心特性的指南，包含一系列示例，并分为不同的主题。

为了使用协程以及按照本指南中的示例演练，需要添加对 `kotlinx-coroutines-core` 模块的依赖，如项目中的 README 文件所述。

目录

其他参考资料

— 使用协程进行 UI 编程指南

— 协程设计文档（KEEP）

— 完整的 kotlinx.coroutines API 参考文档

目录

协程基础

这一部分包括基础的协程概念。

第一个协程程序

运行以下代码：

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // 在后台启动一个新的协程并继续
        delay(1000L) // 非阻塞的等待 1 秒钟（默认时间单位是毫秒）
        println("World!") // 在延迟后打印输出
    }
    println("Hello,") // 协程已在等待时主线程还在继续
    Thread.sleep(2000L) // 阻塞主线程 2 秒钟来保证 JVM 存活
}
```

> 可以在这里获取完整代码。

代码运行的结果：

```
Hello,
World!
```

本质上，协程是轻量级的线程。它们在某些 CoroutineScope 上下文中与 launch *协程构建器* 一起启动。这里我们在 GlobalScope 中启动了一个新的协程，这意味着新协程的生命周期只受整个应用程序的生命周期限制。

可以将 `GlobalScope.launch { …… }` 替换为 `thread { …… }`，并将 `delay(……)` 替换为 `Thread.sleep(……)` 达到同样目的。试试看（不要忘记导入 `kotlin.concurrent.thread`）。

如果你首先将 `GlobalScope.launch` 替换为 `thread`，编译器会报以下错误：

```
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*，它不会造成线程阻塞，但是会 *挂起* 协程，并且只能在协程中使用。

桥接阻塞与非阻塞的世界

第一个示例在同一段代码中混用了 *非阻塞的* `delay(……)` 与 *阻塞的* `Thread.sleep(……)`。这容易让我们记混哪个是阻塞的、哪个是非阻塞的。让我们显式使用 runBlocking 协程构建器来阻塞：

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // 在后台启动一个新的协程并继续
        delay(1000L)
        println("World!")
    }
    println("Hello,") // 主线程中的代码会立即执行
    runBlocking {     // 但是这个表达式阻塞了主线程
        delay(2000L)  // ……我们延迟 2 秒来保证 JVM 的存活
    }
}
```

可以在这里获取完整代码。

结果是相似的，但是这些代码只使用了非阻塞的函数 delay。调用了 `runBlocking` 的主线程会一直 *阻塞* 直到 `runBlocking` 内部的协程执行完毕。

这个示例可以使用更合乎惯用法的方式重写，使用 `runBlocking` 来包装 main 函数的执行：

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> { // 开始执行主协程
    GlobalScope.launch { // 在后台启动一个新的协程并继续
        delay(1000L)
        println("World!")
    }
    println("Hello,") // 主协程在这里会立即执行
    delay(2000L)      // 延迟 2 秒来保证 JVM 存活
}
```

可以在这里获取完整代码。

这里的 `runBlocking<Unit> { …… }` 作为用来启动顶层主协程的适配器。我们显式指定了其返回类型 `Unit`，因为在 Kotlin 中 `main` 函数必须返回 `Unit` 类型。

这也是为挂起函数编写单元测试的一种方式：

```
class MyTest {
    @Test
    fun testMySuspendingFunction() = runBlocking<Unit> {
        // 这里我们可以使用任何喜欢的断言风格来使用挂起函数
    }
}
```

等待一个作业

延迟一段时间来等待另一个协程运行并不是一个好的选择。让我们显式（以非阻塞方式）等待所启动的后台 Job 执行结束：

```
val job = GlobalScope.launch { // 启动一个新协程并保持对这个作业的引用
    delay(1000L)
    println("World!")
}
println("Hello,")
job.join() // 等待直到子协程执行结束
```

> 可以在这里获取完整代码。

现在，结果仍然相同，但是主协程与后台作业的持续时间没有任何关系了。好多了。

结构化的并发

协程的实际使用还有一些需要改进的地方。当我们使用 `GlobalScope.launch` 时，我们会创建一个顶层协程。虽然它很轻量，但它运行时仍会消耗一些内存资源。如果我们忘记保持对新启动的协程的引用，它还会继续运行。如果协程中的代码挂起了会怎么样（例如，我们错误地延迟了太长时间），如果我们启动了太多的协程并导致内存不足会怎么样？必须手动保持对所有已启动协程的引用并 join 之很容易出错。

有一个更好的解决办法。我们可以在代码中使用结构化并发。我们可以在执行操作所在的指定作用域内启动协程，而不是像通常使用线程（线程总是全局的）那样在 GlobalScope 中启动。

在我们的示例中，我们使用 runBlocking 协程构建器将 `main` 函数转换为协程。包括 `runBlocking` 在内的每个协程构建器都将 CoroutineScope 的实例添加到其代码块所在的作用域中。我们可以在这个作用域中启动协程而无需显式 `join` 之，因为外部协程（示例中的 `runBlocking`）直到在其作用域中启动的所有协程都执行完毕后才会结束。因此，可以将我们的示例简化为：

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch { // 在 runBlocking 作用域中启动一个新协程
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
```

> 可以在这里获取完整代码。

作用域构建器

除了由不同的构建器提供协程作用域之外，还可以使用 coroutineScope 构建器声明自己的作用域。它会创建一个协程作用域并且在所有已启动子协程执行完毕之前不会结束。

runBlocking 与 coroutineScope 可能看起来很类似，因为它们都会等待其协程体以及所有子协程结束。主要区别在于，runBlocking 方法会*阻塞*当前线程来等待，而 coroutineScope 只是挂起，会释放底层线程用于其他用途。由于存在这点差异，runBlocking 是常规函数，而 coroutineScope 是挂起函数。

可以通过以下示例来演示：

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch {
        delay(200L)
        println("Task from runBlocking")
    }

    coroutineScope { // 创建一个协程作用域
        launch {
            delay(500L)
            println("Task from nested launch")
        }

        delay(100L)
        println("Task from coroutine scope") // 这一行会在内嵌 launch 之前输出
    }

    println("Coroutine scope is over") // 这一行在内嵌 launch 执行完毕后才输出
}
```

可以在这里获取完整代码。

请注意，(当等待内嵌 launch 时) 紧挨“Task from coroutine scope”消息之后，就会执行并输出“Task from runBlocking”——尽管 coroutineScope 尚未结束。

提取函数重构

我们来将 `launch { …… }` 内部的代码块提取到独立的函数中。当你对这段代码执行“提取函数”重构时，你会得到一个带有 `suspend` 修饰符的新函数。这是你的第一个*挂起函数*。在协程内部可以像普通函数一样使用挂起函数，不过其额外特性是，同样可以使用其他挂起函数（如本例中的 `delay`）来*挂起*协程的执行。

```
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!")
}
```

可以在这里获取完整代码。

但是如果提取出的函数包含一个在当前作用域中调用的协程构建器的话，该怎么办？在这种情况下，所提取函数上只有 `suspend` 修饰符是不够的。为 `CoroutineScope` 写一个 `doWorld` 扩展方法是其中一种解决方案，但这可能并非总是适用，因为它并没有使 API 更加清晰。惯用的解决方案是要么显式将 `CoroutineScope` 作为包含该函数的类的一个字段，要么当外部类实现了 `CoroutineScope` 时隐式取得。作为最后的手段，可以使用 CoroutineScope(coroutineContext)，不过这种方法结构上不安全，因为你不能再控制该方法执行的作用域。只有私有 API 才能使用这个构建器。

协程很轻量

运行以下代码：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(100_000) { // 启动大量的协程
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

可以在这里获取完整代码。

它启动了 10 万个协程，并且在 5 秒钟后，每个协程都输出一个点。

现在，尝试使用线程来实现。会发生什么？（很可能你的代码会产生某种内存不足的错误）

全局协程像守护线程

以下代码在 GlobalScope 中启动了一个长期运行的协程，该协程每秒输出“I'm sleeping”两次，之后在主函数中延迟一段时间后返回。

```
GlobalScope.launch {
    repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
    }
}
delay(1300L) // 在延迟后退出
```

可以在这里获取完整代码。

你可以运行这个程序并看到它输出了以下三行后终止：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
```

在 GlobalScope 中启动的活动协程并不会使进程保活。它们就像守护线程。

目录

取消与超时

这一部分包含了协程的取消与超时。

取消协程的执行

在一个长时间运行的应用程序中，你也许需要对你的后台协程进行细粒度的控制。比如说，一个用户也许关闭了一个启动了协程的界面，那么现在协程的执行结果已经不再被需要了，这时，它应该是可以被取消的。该 launch 函数返回了一个可以被用来取消运行中的协程的 Job：

```
val job = launch {
    repeat(1000) { i ->
        println("job: I'm sleeping $i ...")
        delay(500L)
    }
}
delay(1300L) // 延迟一段时间
println("main: I'm tired of waiting!")
job.cancel() // 取消该作业
job.join() // 等待作业执行结束
println("main: Now I can quit.")
```

> 可以在这里获取完整代码。

程序执行后的输出如下：

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
main: Now I can quit.
```

一旦 main 函数调用了 `job.cancel`，我们在其它的协程中就看不到任何输出，因为它被取消了。这里也有一个可以使 Job 挂起的函数 cancelAndJoin 它合并了对 cancel 以及 join 的调用。

取消是协作的

协程的取消是 *协作* 的。一段协程代码必须协作才能被取消。所有 `kotlinx.coroutines` 中的挂起函数都是 *可被取消的*。它们检查协程的取消，并在取消时抛出 CancellationException。然而，如果协程正在执行计算任务，并且没有检查取消的话，那么它是不能被取消的，就如如下示例代码所示：

```
val startTime = System.currentTimeMillis()
val job = launch(Dispatchers.Default) {
    var nextPrintTime = startTime
    var i = 0
    while (i < 5) { // 一个执行计算的循环，只是为了占用 CPU
        // 每秒打印消息两次
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm sleeping ${i++} ...")
            nextPrintTime += 500L
        }
    }
}
delay(1300L) // 等待一段时间
println("main: I'm tired of waiting!")
job.cancelAndJoin() // 取消一个作业并且等待它结束
println("main: Now I can quit.")
```

可以在这里获取完整代码。

运行示例代码，并且我们可以看到它连续打印出了“I'm sleeping”，甚至在调用取消后，作业仍然执行了五次循环迭代并运行到了它结束为止。

使计算代码可取消

我们有两种方法来使执行计算的代码可以被取消。第一种方法是定期调用挂起函数来检查取消。对于这种目的 yield 是一个好的选择。另一种方法是显式的检查取消状态。让我们试试第二种方法。

将前一个示例中的 `while (i < 5)` 替换为 `while (isActive)` 并重新运行它。

```
val startTime = System.currentTimeMillis()
val job = launch(Dispatchers.Default) {
    var nextPrintTime = startTime
    var i = 0
    while (isActive) { // 可以被取消的计算循环
        // 每秒打印消息两次
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm sleeping ${i++} ...")
            nextPrintTime += 500L
        }
    }
}
delay(1300L) // 等待一段时间
println("main: I'm tired of waiting!")
job.cancelAndJoin() // 取消该作业并等待它结束
println("main: Now I can quit.")
```

可以在这里获取完整代码。

你可以看到，现在循环被取消了。isActive 是一个可以被使用在 CoroutineScope 中的扩展属性。

在 `finally` 中释放资源

我们通常使用如下的方法处理在被取消时抛出 CancellationException 的可被取消的挂起函数。比如说，`try {……} finally {……}` 表达式以及 Kotlin 的 `use` 函数一般在协程被取消的时候执行它们的终结动作：

```
val job = launch {
    try {
        repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
            delay(500L)
        }
    } finally {
        println("job: I'm running finally")
    }
}
delay(1300L) // 延迟一段时间
println("main: I'm tired of waiting!")
job.cancelAndJoin() // 取消该作业并且等待它结束
println("main: Now I can quit.")
```

可以在这里获取完整代码。

join 和 cancelAndJoin 等待了所有的终结动作执行完毕，所以运行示例得到了下面的输出：

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
job: I'm running finally
main: Now I can quit.
```

运行不能取消的代码块

在前一个例子中任何尝试在 `finally` 块中调用挂起函数的行为都会抛出 CancellationException，因为这里持续运行的代码是可以被取消的。通常，这并不是一个问题，所有良好的关闭操作（关闭一个文件、取消一个作业、或是关闭任何一种通信通道）通常都是非阻塞的，并且不会调用任何挂起函数。然而，在真实的案例中，当你需要挂起一个被取消的协程，你可以将相应的代码包装在 `withContext(NonCancellable) {……}` 中，并使用 withContext 函数以及 NonCancellable 上下文，见如下示例所示：

```
val job = launch {
    try {
        repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
            delay(500L)
        }
    } finally {
        withContext(NonCancellable) {
            println("job: I'm running finally")
            delay(1000L)
            println("job: And I've just delayed for 1 sec because I'm non-cancellable")
        }
    }
}
delay(1300L) // 延迟一段时间
println("main: I'm tired of waiting!")
job.cancelAndJoin() // 取消该作业并等待它结束
println("main: Now I can quit.")
```

可以在这里获取完整代码。

超时

在实践中绝大多数取消一个协程的理由是它有可能超时。当你手动追踪一个相关 Job 的引用并启动了一个单独的协程在延迟后取消追踪，这里已经准备好使用 withTimeout 函数来做这件事。来看看示例代码：

```
withTimeout(1300L) {
    repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
    }
}
```

可以在这里获取完整代码。

运行后得到如下输出：

```
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 抛出了 `TimeoutCancellationException`，它是 CancellationException 的子类。我们之前没有在控制台上看到堆栈跟踪信息的打印。这是因为在被取消的协程中 `CancellationException` 被认为是协程执行结束的正常原因。然而，在这个示例中我们在 `main` 函数中正确地使用了 `withTimeout`。

由于取消只是一个例外，所有的资源都使用常用的方法来关闭。如果你需要做一些各类使用超时的特别的额外操作，可以使用类似 withTimeout 的 withTimeoutOrNull 函数，并把这些会超时的代码包装在 `try {...} catch (e: TimeoutCancellationException) {...}` 代码块中，而 withTimeoutOrNull 通过返回 `null` 来进行超时操作，从而替代抛出一个异常：

```
val result = withTimeoutOrNull(1300L) {
    repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
    }
    "Done" // 在它运行得到结果之前取消它
}
println("Result is $result")
```

可以在这里获取完整代码。

运行这段代码时不再抛出异常：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Result is null
```

目录

组合挂起函数

本节介绍了将挂起函数组合的各种方法。

默认顺序调用

假设我们在不同的地方定义了两个进行某种调用远程服务或者进行计算的挂起函数。我们只假设它们都是有用的，但是实际上它们在这个示例中只是为了该目的而延迟了一秒钟：

```
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了一些有用的事
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了一些有用的事
    return 29
}
```

如果需要按 *顺序* 调用它们，我们接下来会做什么——首先调用 `doSomethingUsefulOne` *接下来* 调用 `doSomethingUsefulTwo`，并且计算它们结果的和吗？实际上，如果我们要根据第一个函数的结果来决定是否我们需要调用第二个函数或者决定如何调用它时，我们就会这样做。

我们使用普通的顺序来进行调用，因为这些代码是运行在协程中的，只要像常规的代码一样 *顺序* 都是默认的。下面的示例展示了测量执行两个挂起函数所需要的总时间：

```
val time = measureTimeMillis {
    val one = doSomethingUsefulOne()
    val two = doSomethingUsefulTwo()
    println("The answer is ${one + two}")
}
println("Completed in $time ms")
```

> 可以在这里获取完整代码。

它的打印输出如下：

```
The answer is 42
Completed in 2017 ms
```

使用 async 并发

如果 `doSomethingUsefulOne` 与 `doSomethingUsefulTwo` 之间没有依赖，并且我们想更快的得到结果，让它们进行 *并发* 吗？这就是 async 可以帮助我们的地方。

在概念上，async 就类似于 launch。它启动了一个单独的协程，这是一个轻量级的线程并与其它所有的协程一起并发的工作。不同之处在于 `launch` 返回一个 Job 并且不附带任何结果值，而 `async` 返回一个 Deferred —— 一个轻量级的非阻塞 future，这代表了一个将会在稍后提供结果的 promise。你可以使用 `.await()` 在一个延期的值上得到它的最终结果，但是 `Deferred` 也是一个 `Job`，所以如果需要的话，你可以取消它。

```
val time = measureTimeMillis {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    println("The answer is ${one.await() + two.await()}")
}
println("Completed in $time ms")
```

> 可以在这里获取完整代码。

它的打印输出如下：

```
The answer is 42
Completed in 1017 ms
```

这里快了两倍，因为两个协程并发执行。请注意，使用协程进行并发总是显式的。

惰性启动的 async

可选的，async 可以通过将 `start` 参数设置为 CoroutineStart.LAZY 而变为惰性的。在这个模式下，只有结果通过 await 获取的时候协程才会启动，或者在 `Job` 的 start 函数调用的时候。运行下面的示例：

```
val time = measureTimeMillis {
    val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }
    val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }
    // 执行一些计算
    one.start() // 启动第一个
    two.start() // 启动第二个
    println("The answer is ${one.await() + two.await()}")
}
println("Completed in $time ms")
```

> 可以在这里获取完整代码。

它的打印输出如下：

```
The answer is 42
Completed in 1017 ms
```

因此，在先前的例子中这里定义的两个协程没有执行，但是控制权在于程序员准确的在开始执行时调用 start。我们首先 调用 `one`，然后调用 `two`，接下来等待这个协程执行完毕。

注意，如果我们只是在 `println` 中调用 await，而没有在单独的协程中调用 start，这将会导致顺序行为，直到 await 启动该协程 执行并等待至它结束，这并不是惰性的预期用例。在计算一个值涉及挂起函数时，这个 `async(start = CoroutineStart.LAZY)` 的用例用于替代标准库中的 `lazy` 函数。

async 风格的函数

我们可以定义异步风格的函数来 *异步* 的调用 `doSomethingUsefulOne` 和 `doSomethingUsefulTwo` 并使用 async 协程建造器并带有一个显式的 GlobalScope 引用。我们给这样的函数的名称中加上“……Async”后缀来突出表明：事实上，它们只做异步计算并且需要使用延期的值来获得结果。

```
// 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。它们可以在任何地方使用。然而，它们总是在调用它们的代码中意味着异步（这里的意思是 *并发*）执行。

下面的例子展示了它们在协程的外面是如何使用的：

```
// 注意，在这个示例中我们在 `main` 函数的右边没有加上 `runBlocking`
fun main() {
    val time = measureTimeMillis {
        // 我们可以在协程外面启动异步执行
        val one = somethingUsefulOneAsync()
        val two = somethingUsefulTwoAsync()
        // 但是等待结果必须调用其它的挂起或者阻塞
        // 当我们等待结果的时候，这里我们使用 `runBlocking { …… }` 来阻塞主线程
        runBlocking {
            println("The answer is ${one.await() + two.await()}")
        }
    }
    println("Completed in $time ms")
}
```

> 可以在这里获取完整代码。

> 这种带有异步函数的编程风格仅供参考，因为这在其它编程语言中是一种受欢迎的风格。在 Kotlin 的协程中使用这种风格是**强烈不推荐**的，原因如下所述。

考虑一下如果 `val one = somethingUsefulOneAsync()` 这一行和 `one.await()` 表达式这里在代码中有逻辑错误，并且程序抛出了异常以及程序在操作的过程中中止，将会发生什么。通常情况下，一个全局的异常处理者会捕获这个异常，将异常打印成日记并报告给开发者，但是反之该程序将会继续执行其它操作。但是这里我们的 `somethingUsefulOneAsync` 仍然在后台执行，尽管如此，启动它的那次操作也会被终止。这个程序将不会进行结构化并发，如下一小节所示。

使用 async 的结构化并发

让我们使用使用 async 的并发这一小节的例子并且提取出一个函数并发的调用 `doSomethingUsefulOne` 与 `doSomethingUsefulTwo` 并且返回它们两个的结果之和。由于 async 被定义为了 CoroutineScope 上的扩展，我们需要将它写在作用域内，并且这是 coroutineScope 函数所提供的：

```
suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}
```

这种情况下，如果在 `concurrentSum` 函数内部发生了错误，并且它抛出了一个异常，所有在作用域中启动的协程都会被取消。

```
val time = measureTimeMillis {
    println("The answer is ${concurrentSum()}")
}
println("Completed in $time ms")
```

> 可以在这里获取完整代码。

从上面的 `main` 函数的输出可以看出，我们仍然可以同时执行这两个操作：

```
The answer is 42
Completed in 1017 ms
```

取消始终通过协程的层次结构来进行传递：

```
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) // 模拟一个长时间的运算
            42
        } finally {
            println("First child was cancelled")
        }
    }
    val two = async<Int> {
        println("Second child throws an exception")
        throw ArithmeticException()
    }
    one.await() + two.await()
}
```

> 可以在这里获取完整代码。

请注意，如果其中一个子协程（即 `two`）失败，第一个 `async` 以及等待中的父协程都会被取消：

```
Second child throws an exception
First child was cancelled
Computation failed with ArithmeticException
```

目录

协程上下文与调度器

协程总是运行在一些以 CoroutineContext 类型为代表的上下文中，它们被定义在了 Kotlin 的标准库里。

协程上下文是各种不同元素的集合。其中主元素是协程中的 Job，我们在前面的文档中见过它以及它的调度器，而本文将对它进行介绍。

调度器与线程

协程上下文包含一个 *协程调度器* (参见 CoroutineDispatcher) 它确定了哪些线程或与线程相对应的协程执行。协程调度器可以将协程限制在一个特定的线程执行，或将它分派到一个线程池，亦或是让它不受限地运行。

所有的协程构建器诸如 launch 和 async 接收一个可选的 CoroutineContext 参数，它可以被用来显式的为一个新协程或其它上下文元素指定一个调度器。

尝试下面的示例：

```
launch { // 运行在父协程的上下文中，即 runBlocking 主协程
    println("main runBlocking      : I'm working in thread ${Thread.currentThread().name}")
}
launch(Dispatchers.Unconfined) { // 不受限的——将工作在主线程中
    println("Unconfined            : I'm working in thread ${Thread.currentThread().name}")
}
launch(Dispatchers.Default) { // 将会获取默认调度器
    println("Default               : I'm working in thread ${Thread.currentThread().name}")
}
launch(newSingleThreadContext("MyOwnThread")) { // 将使它获得一个新的线程
    println("newSingleThreadContext: I'm working in thread ${Thread.currentThread().name}")
}
```

> 可以在这里获取完整代码。

它执行后得到了如下输出（也许顺序会有所不同）：

```
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
```

当调用 `launch { …… }` 时不传参数，它从启动了它的 CoroutineScope 中承袭了上下文（以及调度器）。在这个案例中，它从 `main` 线程中的 `runBlocking` 主协程承袭了上下文。

Dispatchers.Unconfined 是一个特殊的调度器且似乎也运行在 `main` 线程中，但实际上，它是一种不同的机制，这会在后文中讲到。

当协程在 GlobalScope 中启动时，使用的是由 Dispatchers.Default 代表的默认调度器。默认调度器使用共享的后台线程池。所以 `launch(Dispatchers.Default) { …… }` 与 `GlobalScope.launch { …… }` 使用相同的调度器。

newSingleThreadContext 为协程的运行启动了一个线程。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两者都必须被释放，当不再需要的时候，使用 close 函数，或存储在一个顶层变量中使它在整个应用程序中被重用。

非受限调度器 vs 受限调度器

Dispatchers.Unconfined 协程调度器在调用它的线程启动了一个协程，但它仅仅只是运行到第一个挂起点。挂起后，它恢复线程中的协程，而这完全由被调用的挂起函数来决定。非受限的调度器非常适用于执行不消耗 CPU 时间的任务，以及不更新局限于特定线程的任何共享数据（如UI）的协程。

另一方面，该调度器默认继承了外部的 CoroutineScope。runBlocking 协程的默认调度器，特别是，当它被限制在了调用者线程时，继承自它将会有效地限制协程在该线程运行并且具有可预测的 FIFO 调度。

```
launch(Dispatchers.Unconfined) { // 非受限的——将和主线程一起工作
    println("Unconfined      : I'm working in thread ${Thread.currentThread().name}")
    delay(500)
    println("Unconfined      : After delay in thread ${Thread.currentThread().name}")
}
launch { // 父协程的上下文，主 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}")
}
```

> 可以在这里获取完整代码。

执行后的输出：

```
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
```

所以，该协程的上下文继承自 `runBlocking {...}` 协程并在 `main` 线程中运行，当 delay 函数调用的时候，非受限的那个协程在默认的执行者线程中恢复执行。

> 非受限的调度器是一种高级机制，可以在某些极端情况下提供帮助而不需要调度协程以便稍后执行或产生不希望的副作用，因为某些操作必须立即在协程中执行。非受限调度器不应该在通常的代码中使用。

调试协程与线程

协程可以在一个线程上挂起并在其它线程上恢复。甚至一个单线程的调度器也是难以弄清楚协程在何时何地正在做什么事情。使用通常调试应用程序的方法是让线程在每一个日志文件的日志声明中打印线程的名字。这种特性在日志框架中是普遍受支持的。但是在使用协程时，单独的线程名称不会给出很多协程上下文信息，所以 `kotlinx.coroutines` 包含了调试工具来让它更简单。

使用 `-Dkotlinx.coroutines.debug` JVM 参数运行下面的代码：

```
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()}")
```

> 可以在这里获取完整代码。

这里有三个协程，包括 `runBlocking` 内的主协程 (#1)，以及计算延期的值的另外两个协程 `a` (#2) 和 `b` (#3)。它们都在 `runBlocking` 上下文中执行并且被限制在了主线程内。这段代码的输出如下：

```
[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
```

这个 `log` 函数在方括号种打印了线程的名字，并且你可以看到它是 `main` 线程，并且附带了当前正在其上执行的协程的标识符。这个标识符在调试模式开启时，将连续分配给所有创建的协程。

> 当 JVM 以 `-ea` 参数配置运行时，调试模式也会开启。你可以在 DEBUG_PROPERTY_NAME 属性的文档中阅读有关调试工具的更多信息。

在不同线程间跳转

使用 `-Dkotlinx.coroutines.debug` JVM 参数运行下面的代码（参见调试）：

```
newSingleThreadContext("Ctx1").use { ctx1 ->
    newSingleThreadContext("Ctx2").use { ctx2 ->
        runBlocking(ctx1) {
            log("Started in ctx1")
            withContext(ctx2) {
                log("Working in ctx2")
            }
            log("Back to ctx1")
        }
    }
}
```

> 可以在这里获取完整代码。

它演示了一些新技术。其中一个使用 runBlocking 来显式指定了一个上下文，并且另一个使用 withContext 函数来改变协程的上下文，而仍然驻留在相同的协程中，正如可以在下面的输出中所见到的：

```
[Ctx1 @coroutine#1] Started in ctx1
[Ctx2 @coroutine#1] Working in ctx2
[Ctx1 @coroutine#1] Back to ctx1
```

注意，在这个例子中，当我们不再需要某个在 newSingleThreadContext 中创建的线程的时候，它使用了 Kotlin 标准库中的 `use` 函数来释放该线程。

上下文中的作业

协程的 Job 是上下文的一部分，并且可以使用 `coroutineContext [Job]` 表达式在上下文中检索它：

```
println("My job is ${coroutineContext[Job]}")
```

> 可以在这里获取完整代码。

在调试模式下，它将输出如下这些信息：

```
My job is "coroutine#1":BlockingCoroutine{Active}@6d311334
```

请注意，CoroutineScope 中的 isActive 只是 `coroutineContext[Job]?.isActive == true` 的一种方便的快捷方式。

子协程

当一个协程被其它协程在 CoroutineScope 中启动的时候，它将通过 CoroutineScope.coroutineContext 来承袭上下文，并且这个新协程的 Job 将会成为父协程作业的 *子* 作业。当一个父协程被取消的时候，所有它的子协程也会被递归的取消。

然而，当使用 GlobalScope 来启动一个协程时，则新协程的作业没有父作业。因此它与这个启动的作用域无关且独立运作。

```
// 启动一个协程来处理某种传入请求（request）
val request = launch {
    // 孵化了两个子作业, 其中一个通过 GlobalScope 启动
    GlobalScope.launch {
        println("job1: I run in GlobalScope and execute independently!")
        delay(1000)
        println("job1: I am not affected by cancellation of the request")
    }
    // 另一个则承袭了父协程的上下文
    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() // 取消请求（request）的执行
delay(1000) // 延迟一秒钟来看看发生了什么
println("main: Who has survived request cancellation?")
```

> 可以在这里获取完整代码。

这段代码的输出如下：

```
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

一个父协程总是等待所有的子协程执行结束。父协程并不显式的跟踪所有子协程的启动，并且不必使用 Job.join 在最后的时候等待它们：

```
// 启动一个协程来处理某种传入请求（request）
val request = launch {
    repeat(3) { i -> // 启动少量的子作业
        launch  {
            delay((i + 1) * 200L) // 延迟 200 毫秒、400 毫秒、600 毫秒的时间
            println("Coroutine $i is done")
        }
    }
    println("request: I'm done and I don't explicitly join my children that are still active")
}
request.join() // 等待请求的完成，包括其所有子协程
println("Now processing of the request is complete")
```

可以在这里获取完整代码。

结果如下所示：

```
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
```

命名协程以用于调试

当协程经常打印日志并且你只需要关联来自同一个协程的日志记录时，则自动分配的 id 是非常好的。然而，当一个协程与特定请求的处理相关联时或做一些特定的后台任务，最好将其明确命名以用于调试目的。CoroutineName 上下文元素与线程名具有相同的目的。当调试模式开启时，它被包含在正在执行此协程的线程名中。

下面的例子演示了这一概念：

```
log("Started main coroutine")
// 运行两个后台值计算
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()}")
```

可以在这里获取完整代码。

程序执行使用了 `-Dkotlinx.coroutines.debug` JVM 参数，输出如下所示：

```
[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

有时我们需要在协程上下文中定义多个元素。我们可以使用 `+` 操作符来实现。比如说，我们可以显式指定一个调度器来启动协程并且同时显式指定一个命名：

```
launch(Dispatchers.Default + CoroutineName("test")) {
    println("I'm working in thread ${Thread.currentThread().name}")
}
```

可以在这里获取完整代码。

这段代码使用了 `-Dkotlinx.coroutines.debug` JVM 参数，输出如下所示：

```
I'm working in thread DefaultDispatcher-worker-1 @test#2
```

协程作用域

让我们将关于上下文，子协程以及作业的知识综合在一起。假设我们的应用程序拥有一个具有生命周期的对象，但这个对象并不是一个协程。举例来说，我们编写了一个 Android 应用程序并在 Android 的 activity 上下文中启动了一组协程来使用异步操作拉取并更新数据以及执行动画等等。所有这些协程必须在这个 activity 销毁的时候取消以避免内存泄漏。当然，我们也可以手动操作上下文与作业，以结合 activity 的生命周期与它的协程，但是 `kotlinx.coroutines` 提供了一个封装：CoroutineScope 的抽象。你应该已经熟悉了协程作用域，因为所有的协程构建器都声明为在它之上的扩展。

我们通过创建一个 CoroutineScope 实例来管理协程的生命周期，并使它与 activity 的生命周期相关联。`CoroutineScope` 可以通过 CoroutineScope() 创建或者通过MainScope() 工厂函数。前者创建了一个通用作用域，而后者为使用 Dispatchers.Main 作为默认调度器的 UI 应用程序 创建作用域：

```
class Activity {
    private val mainScope = MainScope()

    fun destroy() {
        mainScope.cancel()
    }
    // 继续运行……
```

Now, we can launch coroutines in the scope of this `Activity` using the defined `scope`. For the demo, we launch ten coroutines that delay for a different time:

```
    // 在 Activity 类中
    fun doSomething() {
        // 在示例中启动了 10 个协程，且每个都工作了不同的时长
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // 延迟 200 毫秒、400 毫秒、600 毫秒等等不同的时间
                println("Coroutine $i is done")
            }
        }
    }
} // Activity 类结束
```

在 main 函数中我们创建 activity，调用测试函数 `doSomething`，并且在 500 毫秒后销毁这个 activity。这取消了从 `doSomething` 启动的所有协程。我们可以观察到这些是由于在销毁之后，即使我们再等一会儿，activity 也不再打印消息。

```
val activity = Activity()
activity.doSomething() // 运行测试函数
println("Launched coroutines")
delay(500L) // 延迟半秒钟
println("Destroying activity!")
activity.destroy() // 取消所有的协程
delay(1000) // 为了在视觉上确认它们没有工作
```

可以在这里获取完整代码。

这个示例的输出如下所示：

```
Launched coroutines
Coroutine 0 is done
Coroutine 1 is done
Destroying activity!
```

你可以看到，只有前两个协程打印了消息，而另一个协程在 `Activity.destroy()` 中单次调用了 `job.cancel()`。

注意，Android 在所有具有生命周期的实体中都对协程作用域提供了一等的支持。请查看相关文档。

线程局部数据

有时，能够将一些线程局部数据传递到协程与协程之间是很方便的。然而，由于它们不受任何特定线程的约束，如果手动完成，可能会导致出现样板代码。

`ThreadLocal`，asContextElement 扩展函数在这里会充当救兵。它创建了额外的上下文元素，且保留给定 `ThreadLocal` 的值，并在每次协程切换其上下文时恢复它。

它很容易在下面的代码中演示：

```
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()}'")
```

可以在这里获取完整代码。

在这个例子中我们使用 Dispatchers.Default 在后台线程池中启动了一个新的协程，所以它工作在线程池中的不同线程中，但它仍然具有线程局部变量的值，我们指定使用 `threadLocal.asContextElement(value = "launch")`，无论协程执行在哪个线程中都是没有问题的。因此，其输出如（调试）所示：

```
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'
```

这很容易忘记去设置相应的上下文元素。如果运行协程的线程不同，在协程中访问的线程局部变量则可能会产生意外的值。为了避免这种情况，建议使用 ensurePresent 方法并且在不正确的使用时快速失败。

`ThreadLocal` 具有一流的支持，可以与任何 `kotlinx.coroutines` 提供的原语一起使用。但它有一个关键限制，即：当一个线程局部变量变化时，则这个新值不会传播给协程调用者（因为上下文元素无法追踪所有 `ThreadLocal` 对象访问），并且下次挂起时更新的值将丢失。使用 withContext 在协程中更新线程局部变量，详见 asContextElement。

另外，一个值可以存储在一个可变的域中，例如 `class Counter(var i: Int)`，是的，反过来，可以存储在线程局部的变量中。然而，在这个案例中你完全有责任来进行同步可能的对这个可变的域进行的并发的修改。

对于高级的使用，例如，那些在内部使用线程局部传递数据的用于与日志记录 MDC 集成，以及事务上下文或任何其它库，请参见需要实现的 ThreadContextElement 接口的文档。

目录

异步流

挂起函数可以异步的返回单个值,但是该如何异步返回多个计算好的值呢?这正是 Kotlin 流(Flow)的用武之地。

表示多个值

在 Kotlin 中可以使用集合来表示多个值。比如说,我们有一个 `simple` 函数,它返回一个包含三个数字的 List,然后使用 forEach 打印它们:

```
fun simple(): List<Int> = listOf(1, 2, 3)

fun main() {
    simple().forEach { value -> println(value) }
}
```

> 可以在这里获取完整代码。

这段代码输出如下:

```
1
2
3
```

序列

如果使用一些消耗 CPU 资源的阻塞代码计算数字(每次计算需要 100 毫秒)那么我们可以使用 Sequence 来表示数字:

```
fun simple(): Sequence<Int> = sequence { // 序列构建器
    for (i in 1..3) {
        Thread.sleep(100) // 假装我们正在计算
        yield(i) // 产生下一个值
    }
}

fun main() {
    simple().forEach { value -> println(value) }
}
```

> 可以在这里获取完整代码。

这段代码输出相同的数字,但在打印每个数字之前等待 100 毫秒。

挂起函数

然而，计算过程阻塞运行该代码的主线程。当这些值由异步代码计算时，我们可以使用 `suspend` 修饰符标记函数 `simple`，这样它就可以在不阻塞的情况下执行其工作并将结果作为列表返回：

```
suspend fun simple(): List<Int> {
    delay(1000) // 假装我们在这里做了一些异步的事情
    return listOf(1, 2, 3)
}

fun main() = runBlocking<Unit> {
    simple().forEach { value -> println(value) }
}
```

> 可以在这里获取完整代码。

这段代码将会在等待一秒之后打印数字。

流

使用 List 结果类型，意味着我们只能一次返回所有值。为了表示异步计算的值流（stream），我们可以使用 Flow 类型（正如同步计算值会使用 Sequence 类型）：

```

simple(): Flow<Int> = flow { // 流构建器
for (i in 1..3) {
    delay(100) // 假装我们在这里做了一些有用的事情
    emit(i) // 发送下一个值
}


main() = runBlocking<Unit> {
// 启动并发的协程以验证主线程并未阻塞
launch {
    for (k in 1..3) {
        println("I'm not blocked $k")
        delay(100)
    }
}
// 收集这个流
simple().collect { value -> println(value) }
```

> 可以在这里获取完整代码。

这段代码在不阻塞主线程的情况下每等待 100 毫秒打印一个数字。在主线程中运行一个单独的协程每 100 毫秒打印一次 “I'm not blocked” 已经经过了验证。

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

注意使用 Flow 的代码与先前示例的下述区别：

— 名为 flow 的 Flow 类型构建器函数。

— `flow { ... }` 构建块中的代码可以挂起。

— 函数 `simple` 不再标有 `suspend` 修饰符。

— 流使用 emit 函数 *发射* 值。
— 流使用 collect 函数 *收集* 值。

我们可以在 `simple` 的 `flow { ... }` 函数体内使用 delay 代替 `Thread.sleep` 以观察主线程在本案例中被阻塞了。

流是冷的

Flow 是一种类似于序列的冷流 — 这段 flow 构建器中的代码直到流被收集的时候才运行。这在以下的示例中非常明显：

```
simple(): Flow<Int> = flow {
println("Flow started")
for (i in 1..3) {
    delay(100)
    emit(i)
}


main() = runBlocking<Unit> {
println("Calling simple function...")
val flow = simple()
println("Calling collect...")
flow.collect { value -> println(value) }
println("Calling collect again...")
flow.collect { value -> println(value) }
```

可以在这里获取完整代码。

打印如下：

```
Calling simple function...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

这是返回一个流的 `simple` 函数没有标记 `suspend` 修饰符的主要原因。通过它自己，`simple()` 调用会尽快返回且不会进行任何等待。该流在每次收集的时候启动，这就是为什么当我们再次调用 `collect` 时我们会看到“Flow started”。

流取消基础

流采用与协程同样的协作取消。像往常一样，流的收集可以在当流在一个可取消的挂起函数（例如 delay）中挂起的时候取消。以下示例展示了当 withTimeoutOrNull 块中代码在运行的时候流是如何在超时的情况下取消并停止执行其代码的：

```
simple(): Flow<Int> = flow {
for (i in 1..3) {
    delay(100)
    println("Emitting $i")
    emit(i)
}


main() = runBlocking<Unit> {
withTimeoutOrNull(250) { // 在 250 毫秒后超时
    simple().collect { value -> println(value) }
}
println("Done")
```

可以在这里获取完整代码。

注意，在 `simple` 函数中流仅发射两个数字，产生以下输出：

```
Emitting 1
1
Emitting 2
2
Done
```

See Flow cancellation checks section for more details.

流构建器

先前示例中的 `flow { ... }` 构建器是最基础的一个。还有其他构建器使流的声明更简单：

- flowOf 构建器定义了一个发射固定值集的流。
- 使用 `.asFlow()` 扩展函数，可以将各种集合与序列转换为流。

因此，从流中打印从 1 到 3 的数字的示例可以写成：

```
// 将一个整数区间转化为流
(1..3).asFlow().collect { value -> println(value) }
```

可以在这里获取完整代码。

过渡流操作符

可以使用操作符转换流，就像使用集合与序列一样。过渡操作符应用于上游流，并返回下游流。这些操作符也是冷操作符，就像流一样。这类操作符本身不是挂起函数。它运行的速度很快，返回新的转换流的定义。

基础的操作符拥有相似的名字，比如 map 与 filter。流与序列的主要区别在于这些操作符中的代码可以调用挂起函数。

举例来说，一个请求中的流可以使用 map 操作符映射出结果，即使执行一个长时间的请求操作也可以使用挂起函数来实现：

```
end fun performRequest(request: Int): String {
delay(1000) // 模仿长时间运行的异步工作
return "response $request"


main() = runBlocking<Unit> {
(1..3).asFlow() // 一个请求流
    .map { request -> performRequest(request) }
    .collect { response -> println(response) }
```

可以在这里获取完整代码。

它产生以下三行，每一行每秒出现一次：

```
response 1
response 2
response 3
```

转换操作符

在流转换操作符中，最通用的一种称为 transform。它可以用来模仿简单的转换，例如 map 与 filter，以及实施更复杂的转换。使用 `transform` 操作符，我们可以 发射 任意值任意次。

比如说，使用 `transform` 我们可以在执行长时间运行的异步请求之前发射一个字符串并跟踪这个响应：

```
(1..3).asFlow() // 一个请求流
    .transform { request ->
        emit("Making request $request")
        emit(performRequest(request))
    }
    .collect { response -> println(response) }
```

可以在这里获取完整代码。

这段代码的输出如下：

```
Making request 1
response 1
Making request 2
response 2
Making request 3
response 3
```

限长操作符

限长过渡操作符（例如 take）在流触及相应限制的时候会将它的执行取消。协程中的取消操作总是通过抛出异常来执行，这样所有的资源管理函数（如 `try {...} finally {...}` 块）会在取消的情况下正常运行：

```
fun numbers(): Flow<Int> = flow {
    try {
        emit(1)
        emit(2)
        println("This line will not execute")
        emit(3)
    } finally {
        println("Finally in numbers")
    }
}

fun main() = runBlocking<Unit> {
    numbers()
        .take(2) // 只获取前两个
        .collect { value -> println(value) }
}
```

可以在这里获取完整代码。

这段代码的输出清楚地表明，`numbers()` 函数中对 `flow {...}` 函数体的执行在发射出第二个数字后停止：

```
1
2
Finally in numbers
```

末端流操作符

末端操作符是在流上用于启动流收集的*挂起函数*。collect 是最基础的末端操作符，但是还有另外一些更方便使用的末端操作符：

- 转化为各种集合，例如 toList 与 toSet。
- 获取第一个（first）值与确保流发射单个（single）值的操作符。
- 使用 reduce 与 fold 将流规约到单个值。

举例来说：

```
val sum = (1..5).asFlow()
    .map { it * it } // 数字 1 至 5 的平方
    .reduce { a, b -> a + b } // 求和（末端操作符）
println(sum)
```

可以在这里获取完整代码。

打印单个数字：

```
55
```

流是连续的

流的每次单独收集都是按顺序执行的，除非进行特殊操作的操作符使用多个流。该收集过程直接在协程中运行，该协程调用末端操作符。默认情况下不启动新协程。从上游到下游每个过渡操作符都会处理每个发射出的值然后再交给末端操作符。

请参见以下示例，该示例过滤偶数并将其映射到字符串：

```
(1..5).asFlow()
    .filter {
        println("Filter $it")
        it % 2 == 0
    }
    .map {
        println("Map $it")
        "string $it"
    }.collect {
        println("Collect $it")
    }
```

可以在这里获取完整代码。

执行：

```
Filter 1
Filter 2
Map 2
Collect string 2
Filter 3
Filter 4
Map 4
Collect string 4
Filter 5
```

流上下文

流的收集总是在调用协程的上下文中发生。例如，如果有一个流 `simple`，然后以下代码在它的编写者指定的上下文中运行，而无论流 `simple` 的实现细节如何：

```
withContext(context) {
    simple().collect { value ->
        println(value) // 运行在指定上下文中
    }
}
```

流的该属性称为 *上下文保存*。

所以默认的，`flow { ... }` 构建器中的代码运行在相应流的收集器提供的上下文中。举例来说，考虑打印线程的一个 `simple` 函数的实现，它被调用并发射三个数字：

```
fun simple(): Flow<Int> = flow {
    log("Started simple flow")
    for (i in 1..3) {
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> log("Collected $value") }
}
```

可以在这里获取完整代码。

运行这段代码：

```
[main @coroutine#1] Started simple flow
[main @coroutine#1] Collected 1
[main @coroutine#1] Collected 2
[main @coroutine#1] Collected 3
```

由于 `simple().collect` 是在主线程调用的，则 `simple` 的流主体也是在主线程调用的。这是快速运行或异步代码的理想默认形式，它不关心执行的上下文并且不会阻塞调用者。

withContext 发出错误

然而，长时间运行的消耗 CPU 的代码也许需要在 Dispatchers.Default 上下文中执行，并且更新 UI 的代码也许需要在 Dispatchers.Main 中执行。通常，withContext 用于在 Kotlin 协程中改变代码的上下文，但是 `flow {...}` 构建器中的代码必须遵循上下文保存属性，并且不允许从其他上下文中发射（emit）。

尝试运行下面的代码：

```
fun simple(): Flow<Int> = flow {
    // 在流构建器中更改消耗 CPU 代码的上下文的错误方式
    kotlinx.coroutines.withContext(Dispatchers.Default) {
        for (i in 1..3) {
            Thread.sleep(100) // 假装我们以消耗 CPU 的方式进行计算
            emit(i) // 发射下一个值
        }
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> println(value) }
}
```

> 可以在这里获取完整代码。

这段代码产生如下的异常：

```
Exception in thread "main" java.lang.IllegalStateException: Flow invariant is violated:
  Flow was collected in [CoroutineId(1), "coroutine#1":BlockingCoroutine{Active}@5511c7f8,
BlockingEventLoop@2eac3323],
  but emission happened in [CoroutineId(1), "coroutine#1":DispatchedCoroutine{Active}@2dae0000,
Dispatchers.Default].
  Please refer to 'flow' documentation or use 'flowOn' instead
 at ...
```

flowOn 操作符

例外的是 flowOn 函数，该函数用于更改流发射的上下文。以下示例展示了更改流上下文的正确方法，该示例还通过打印相应线程的名字以展示它们的工作方式：

```
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        Thread.sleep(100) // 假装我们以消耗 CPU 的方式进行计算
        log("Emitting $i")
        emit(i) // 发射下一个值
    }
}.flowOn(Dispatchers.Default) // 在流构建器中改变消耗 CPU 代码上下文的正确方式

fun main() = runBlocking<Unit> {
    simple().collect { value ->
        log("Collected $value")
    }
}
```

可以在这里获取完整代码。

注意，当收集发生在主线程中，`flow { ... }` 是如何在后台线程中工作的：

这里要观察的另一件事是 flowOn 操作符已改变流的默认顺序性。现在收集发生在一个协程中（“coroutine#1”）而发射发生在运行于另一个线程中与收集协程并发运行的另一个协程（“coroutine#2”）中。当上游流必须改变其上下文中的 CoroutineDispatcher 的时候，flowOn 操作符创建了另一个协程。

缓冲

从收集流所花费的时间来看，将流的不同部分运行在不同的协程中将会很有帮助，特别是当涉及到长时间运行的异步操作时。例如，考虑一种情况，一个 `simple` 流的发射很慢，它每花费 100 毫秒才产生一个元素；而收集器也非常慢，需要花费 300 毫秒来处理元素。让我们看看从该流收集三个数字要花费多长时间：

```
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 假装我们异步等待了 100 毫秒
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        simple().collect { value ->
            delay(300) // 假装我们花费 300 毫秒来处理它
            println(value)
        }
    }
    println("Collected in $time ms")
}
```

可以在这里获取完整代码。

它会产生这样的结果，整个收集过程大约需要 1200 毫秒（3 个数字，每个花费 400 毫秒）：

```
1
2
3
Collected in 1220 ms
```

我们可以在流上使用 buffer 操作符来并发运行这个 `simple` 流中发射元素的代码以及收集的代码，而不是顺序运行它们：

```
val time = measureTimeMillis {
    simple()
        .buffer() // 缓冲发射项，无需等待
        .collect { value ->
            delay(300) // 假装我们花费 300 毫秒来处理它
            println(value)
        }
}
println("Collected in $time ms")
```

可以在这里获取完整代码。

它产生了相同的数字，只是更快了，由于我们高效地创建了处理流水线，仅仅需要等待第一个数字产生的 100 毫秒以及处理每个数字各需花费的 300 毫秒。这种方式大约花费了 1000 毫秒来运行：

```
1
2
3
Collected in 1071 ms
```

> 注意，当必须更改 CoroutineDispatcher 时，flowOn 操作符使用了相同的缓冲机制，但是我们在这里显式地请求缓冲而不改变执行上下文。

合并

当流代表部分操作结果或操作状态更新时，可能没有必要处理每个值，而是只处理最新的那个。在本示例中，当收集器处理它们太慢的时候，conflate 操作符可以用于跳过中间值。构建前面的示例：

```
val time = measureTimeMillis {
    simple()
        .conflate() // 合并发射项，不对每个值进行处理
        .collect { value ->
            delay(300) // 假装我们花费 300 毫秒来处理它
            println(value)
        }
}
println("Collected in $time ms")
```

> 可以在这里获取完整代码。

我们看到，虽然第一个数字仍在处理中，但第二个和第三个数字已经产生，因此第二个是 *conflated*，只有最新的（第三个）被交付给收集器：

```
1
3
Collected in 758 ms
```

处理最新值

当发射器和收集器都很慢的时候，合并是加快处理速度的一种方式。它通过删除发射值来实现。另一种方式是取消缓慢的收集器，并在每次发射新值的时候重新启动它。有一组与 `xxx` 操作符执行相同基本逻辑的 `xxxLatest` 操作符，但是在新值产生的时候取消执行其块中的代码。让我们在先前的示例中尝试更换 conflate 为 collectLatest：

```
val time = measureTimeMillis {
    simple()
        .collectLatest { value -> // 取消并重新发射最后一个值
            println("Collecting $value")
            delay(300) // 假装我们花费 300 毫秒来处理它
            println("Done $value")
        }
}
println("Collected in $time ms")
```

> 可以在这里获取完整代码。

由于 collectLatest 的函数体需要花费 300 毫秒，但是新值每 100 秒发射一次，我们看到该代码块对每个值运行，但是只收集最后一个值：

```
Collecting 1
Collecting 2
Collecting 3
Done 3
Collected in 741 ms
```

组合多个流

组合多个流有很多种方式。

Zip

就像 Kotlin 标准库中的 Sequence.zip 扩展函数一样，流拥有一个 zip 操作符用于组合两个流中的相关值：

```
val nums = (1..3).asFlow() // 数字 1..3
val strs = flowOf("one", "two", "three") // 字符串
nums.zip(strs) { a, b -> "$a -> $b" } // 组合单个字符串
    .collect { println(it) } // 收集并打印
```

> 可以在这里获取完整代码。

示例打印如下：

```
1 -> one
2 -> two
3 -> three
```

Combine

当流表示一个变量或操作的最新值时（请参阅相关小节 conflation），可能需要执行计算，这依赖于相应流的最新值，并且每当上游流产生值的时候都需要重新计算。这种相应的操作符家族称为 combine。

例如，先前示例中的数字如果每 300 毫秒更新一次，但字符串每 400 毫秒更新一次，然后使用 zip 操作符合并它们，但仍会产生相同的结果，尽管每 400 毫秒打印一次结果：

> 我们在本示例中使用 onEach 过渡操作符来延时每次元素发射并使该流更具说明性以及更简洁。

```
val nums = (1..3).asFlow().onEach { delay(300) } // 发射数字 1..3，间隔 300 毫秒
val strs = flowOf("one", "two", "three").onEach { delay(400) } // 每 400 毫秒发射一次字符串
val startTime = System.currentTimeMillis() // 记录开始的时间
nums.zip(strs) { a, b -> "$a -> $b" } // 使用“zip”组合单个字符串
    .collect { value -> // 收集并打印
        println("$value at ${System.currentTimeMillis() - startTime} ms from start")
    }
```

> 可以在这里获取完整代码。

然而，当在这里使用 combine 操作符来替换 zip：

```
val nums = (1..3).asFlow().onEach { delay(300) } // 发射数字 1..3，间隔 300 毫秒
val strs = flowOf("one", "two", "three").onEach { delay(400) } // 每 400 毫秒发射一次字符串
val startTime = System.currentTimeMillis() // 记录开始的时间
nums.combine(strs) { a, b -> "$a -> $b" } // 使用“combine”组合单个字符串
    .collect { value -> // 收集并打印
        println("$value at ${System.currentTimeMillis() - startTime} ms from start")
    }
```

可以在这里获取完整代码。

我们得到了完全不同的输出，其中，`nums` 或 `strs` 流中的每次发射都会打印一行：

```
1 -> one at 452 ms from start
2 -> one at 651 ms from start
2 -> two at 854 ms from start
3 -> two at 952 ms from start
3 -> three at 1256 ms from start
```

展平流

流表示异步接收的值序列，所以很容易遇到这样的情况：每个值都会触发对另一个值序列的请求。比如说，我们可以拥有下面这样一个返回间隔 500 毫秒的两个字符串流的函数：

```
fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 毫秒
    emit("$i: Second")
}
```

现在，如果我们有一个包含三个整数的流，并为每个整数调用 `requestFlow`，如下所示：

```
(1..3).asFlow().map { requestFlow(it) }
```

然后我们得到了一个包含流的流（`Flow<Flow<String>>`），需要将其进行*展平*为单个流以进行下一步处理。集合与序列都拥有 flatten 与 flatMap 操作符来做这件事。然而，由于流具有异步的性质，因此需要不同的展平*模式*，为此，存在一系列的流展平操作符。

flatMapConcat

连接模式由 flatMapConcat 与 flattenConcat 操作符实现。它们是相应序列操作符最相近的类似物。它们在等待内部流完成之前开始收集下一个值，如下面的示例所示：

```
val startTime = System.currentTimeMillis() // 记录开始时间
(1..3).asFlow().onEach { delay(100) } // 每 100 毫秒发射一个数字
    .flatMapConcat { requestFlow(it) }
    .collect { value -> // 收集并打印
        println("$value at ${System.currentTimeMillis() - startTime} ms from start")
    }
```

可以在这里获取完整代码。

在输出中可以清楚地看到 flatMapConcat 的顺序性质：

```
1: First at 121 ms from start
1: Second at 622 ms from start
2: First at 727 ms from start
2: Second at 1227 ms from start
3: First at 1328 ms from start
3: Second at 1829 ms from start
```

flatMapMerge

另一种展平模式是并发收集所有传入的流，并将它们的值合并到一个单独的流，以便尽快的发射值。它由 flatMapMerge 与 flattenMerge 操作符实现。他们都接收可选的用于限制并发收集的流的个数的 `concurrency` 参数（默认情况下，它等于 DEFAULT_CONCURRENCY）。

```
val startTime = System.currentTimeMillis() // 记录开始时间
(1..3).asFlow().onEach { delay(100) } // 每 100 毫秒发射一个数字
    .flatMapMerge { requestFlow(it) }
    .collect { value -> // 收集并打印
        println("$value at ${System.currentTimeMillis() - startTime} ms from start")
    }
```

可以在这里获取完整代码。

flatMapMerge 的并发性质很明显：

```
1: First at 136 ms from start
2: First at 231 ms from start
3: First at 333 ms from start
1: Second at 639 ms from start
2: Second at 732 ms from start
3: Second at 833 ms from start
```

注意，flatMapMerge 会顺序调用代码块（本示例中的 `{ requestFlow(it) }`），但是并发收集结果流，相当于执行顺序是首先执行 `map { requestFlow(it) }` 然后在其返回结果上调用 flattenMerge。

flatMapLatest

与 collectLatest 操作符类似（在"处理最新值" 小节中已经讨论过），也有相对应的“最新”展平模式，在发出新流后立即取消先前流的收集。这由 flatMapLatest 操作符来实现。

```
val startTime = System.currentTimeMillis() // 记录开始时间
(1..3).asFlow().onEach { delay(100) } // 每 100 毫秒发射一个数字
    .flatMapLatest { requestFlow(it) }
    .collect { value -> // 收集并打印
        println("$value at ${System.currentTimeMillis() - startTime} ms from start")
    }
```

可以在这里获取完整代码。

该示例的输出很好的展示了 flatMapLatest 的工作方式：

```
1: First at 142 ms from start
2: First at 322 ms from start
3: First at 425 ms from start
3: Second at 931 ms from start
```

> 注意，flatMapLatest 在一个新值到来时取消了块中的所有代码 (本示例中的 `{ requestFlow(it) }`)。这在该特定示例中不会有什么区别，由于调用 `requestFlow` 自身的速度是很快的，不会发生挂起，所以不会被取消。然而，如果我们要在块中调用诸如 `delay` 之类的挂起函数，这将会被表现出来。

流异常

当运算符中的发射器或代码抛出异常时，流收集可以带有异常的完成。有几种处理异常的方法。

收集器 try 与 catch

收集者可以使用 Kotlin 的 `try/catch` 块来处理异常：

```
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value ->
            println(value)
            check(value <= 1) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

> 可以在这里获取完整代码。

这段代码成功的在末端操作符 collect 中捕获了异常，并且，如我们所见，在这之后不再发出任何值：

```
Emitting 1
1
Emitting 2
2
Caught java.lang.IllegalStateException: Collected 2
```

一切都已捕获

前面的示例实际上捕获了在发射器或任何过渡或末端操作符中发生的任何异常。例如，让我们修改代码以便将发出的值映射为字符串，但是相应的代码会产生一个异常：

```
fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // 发射下一个值
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

可以在这里获取完整代码。

仍然会捕获该异常并停止收集：

```
Emitting 1
string 1
Emitting 2
Caught java.lang.IllegalStateException: Crashed on 2
```

异常透明性

但是，发射器的代码如何封装其异常处理行为？

流必须*对异常透明*，即在 `flow { ... }` 构建器内部的 `try/catch` 块中发射值是违反异常透明性的。这样可以保证收集器抛出的一个异常能被像先前示例中那样的 `try/catch` 块捕获。

发射器可以使用 catch 操作符来保留此异常的透明性并允许封装它的异常处理。catch 操作符的代码块可以分析异常并根据捕获到的异常以不同的方式对其做出反应：

- 可以使用 `throw` 重新抛出异常。
- 可以使用 catch 代码块中的 emit 将异常转换为值发射出去。
- 可以将异常忽略，或用日志打印，或使用一些其他代码处理它。

例如，让我们在捕获异常的时候发射文本：

```
simple()
    .catch { e -> emit("Caught $e") } // 发射一个异常
    .collect { value -> println(value) }
```

可以在这里获取完整代码。

即使我们不再在代码的外层使用 `try/catch`，示例的输出也是相同的。

透明捕获

catch 过渡操作符遵循异常透明性，仅捕获上游异常（`catch` 操作符上游的异常，但是它下面的不是）。如果 `collect { ... }` 块（位于 `catch` 之下）抛出一个异常，那么异常会逃逸：

```
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple()
        .catch { e -> println("Caught $e") } // 不会捕获下游异常
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
```

可以在这里获取完整代码。

尽管有 `catch` 操作符，但不会打印“Caught …”消息：

声明式捕获

我们可以将 catch 操作符的声明性与处理所有异常的期望相结合，将 collect 操作符的代码块移动到 onEach 中，并将其放到 `catch` 操作符之前。收集该流必须由调用无参的 `collect()` 来触发：

```
simple()
    .onEach { value ->
        check(value <= 1) { "Collected $value" }
        println(value)
    }
    .catch { e -> println("Caught $e") }
    .collect()
```

可以在这里获取完整代码。

现在我们可以看到已经打印了“Caught …”消息，并且我们可以在没有显式使用 `try/catch` 块的情况下捕获所有异常：

流完成

当流收集完成时（普通情况或异常情况），它可能需要执行一个动作。你可能已经注意到，它可以通过两种方式完成：命令式或声明式。

命令式 finally 块

除了 `try/catch` 之外，收集器还能使用 `finally` 块在 `collect` 完成时执行一个动作。

```
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } finally {
        println("Done")
    }
}
```

可以在这里获取完整代码。

这段代码打印出 `simple` 流产生的三个数字，后面跟一个“Done”字符串：

```
1
2
3
Done
```

声明式处理

对于声明式，流拥有 onCompletion 过渡操作符，它在流完全收集时调用。

可以使用 onCompletion 操作符重写前面的示例，并产生相同的输出：

```
simple()
    .onCompletion { println("Done") }
    .collect { value -> println(value) }
```

> 可以在这里获取完整代码。

onCompletion 的主要优点是其 lambda 表达式的可空参数 `Throwable` 可以用于确定流收集是正常完成还是有异常发生。在下面的示例中 `simple` 流在发射数字 1 之后抛出了一个异常：

```
fun simple(): Flow<Int> = flow {
    emit(1)
    throw RuntimeException()
}

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> if (cause != null) println("Flow completed exceptionally") }
        .catch { cause -> println("Caught exception") }
        .collect { value -> println(value) }
}
```

> 可以在这里获取完整代码。

如你所期望的，它打印了：

```
1
Flow completed exceptionally
Caught exception
```

onCompletion 操作符与 catch 不同，它不处理异常。我们可以看到前面的示例代码，异常仍然流向下游。它将被提供给后面的 `onCompletion` 操作符，并可以由 `catch` 操作符处理。

成功完成

与 catch 操作符的另一个不同点是 onCompletion 能观察到所有异常并且仅在上游流成功完成（没有取消或失败）的情况下接收一个 `null` 异常。

```
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> println("Flow completed with $cause") }
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
```

可以在这里获取完整代码。

我们可以看到完成时 cause 不为空，因为流由于下游异常而中止：

```
1
Flow completed with java.lang.IllegalStateException: Collected 2
Exception in thread "main" java.lang.IllegalStateException: Collected 2
```

命令式还是声明式

现在我们知道如何收集流，并以命令式与声明式的方式处理其完成及异常情况。这里有一个很自然的问题是，哪种方式应该是首选的?为什么？作为一个库，我们不主张采用任何特定的方式，并且相信这两种选择都是有效的，应该根据自己的喜好与代码风格进行选择。

启动流

使用流表示来自一些源的异步事件是很简单的。在这个案例中，我们需要一个类似 `addEventListener` 的函数，该函数注册一段响应的代码处理即将到来的事件，并继续进行进一步的处理。onEach 操作符可以担任该角色。然而，`onEach` 是一个过渡操作符。我们也需要一个末端操作符来收集流。否则仅调用 `onEach` 是无效的。

如果我们在 `onEach` 之后使用 collect 末端操作符，那么后面的代码会一直等待直至流被收集：

```
// 模仿事件流
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }

fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .collect() // <--- 等待流收集
    println("Done")
}
```

可以在这里获取完整代码。

你可以看到它的输出：

```
Event: 1
Event: 2
Event: 3
Done
```

launchIn 末端操作符可以在这里派上用场。使用 `launchIn` 替换 `collect` 我们可以在单独的协程中启动流的收集，这样就可以立即继续进一步执行代码：

```
fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .launchIn(this) // <--- 在单独的协程中执行流
    println("Done")
}
```

可以在这里获取完整代码。

它打印了：

```
Done
Event: 1
Event: 2
Event: 3
```

`launchIn` 必要的参数 CoroutineScope 指定了用哪一个协程来启动流的收集。在先前的示例中这个作用域来自 runBlocking 协程构建器，在这个流运行的时候，runBlocking 作用域等待它的子协程执行完毕并防止 main 函数返回并终止此示例。

在实际的应用中，作用域来自于一个寿命有限的实体。在该实体的寿命终止后，相应的作用域就会被取消，即取消相应流的收集。这种成对的 `onEach { ... }.launchIn(scope)` 工作方式就像 `addEventListener` 一样。而且，这不需要相应的 `removeEventListener` 函数，因为取消与结构化并发可以达成这个目的。

注意，launchIn 也会返回一个 Job，可以在不取消整个作用域的情况下仅取消相应的流收集或对其进行 join。

Flow cancellation checks

For convenience, the flow builder performs additional ensureActive checks for cancellation on each emitted value. It means that a busy loop emitting from a `flow { ... }` is cancellable:

```
foo(): Flow<Int> = flow {
for (i in 1..5) {
    println("Emitting $i")
    emit(i)
}


main() = runBlocking<Unit> {
foo().collect { value ->
    if (value == 3) cancel()
    println(value)
}
```

You can get the full code from here.

We get only numbers up to 3 and a CancellationException after trying to emit number 4:

```
Emitting 1
1
Emitting 2
2
Emitting 3
3
Emitting 4
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@6d7b4f4c
```

However, most other flow operators do not do additional cancellation checks on their own for performance reasons. For example, if you use IntRange.asFlow extension to write the same busy loop and don't suspend anywhere, then there are no checks for cancellation:

```
main() = runBlocking<Unit> {
(1..5).asFlow().collect { value ->
    if (value == 3) cancel()
    println(value)
}
```

You can get the full code from here.

All numbers from 1 to 5 are collected and cancellation gets detected only before return from `runBlocking`:

```
1
2
3
4
5
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@3327bd23
```

Making busy flow cancellable

In the case where you have a busy loop with coroutines you must explicitly check for cancellation. You can add `.onEach { currentCoroutineContext().ensureActive() }`, but there is a ready-to-use cancellable operator provided to do that:

```
main() = runBlocking<Unit> {
(1..5).asFlow().cancellable().collect { value ->
    if (value == 3) cancel()
    println(value)
}
```

You can get the full code from here.

With the `cancellable` operator only the numbers from 1 to 3 are collected:

```
1
2
3
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCoroutine was
cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@5ec0a365
```

流（Flow）与响应式流（Reactive Streams）

对于熟悉响应式流（Reactive Streams）或诸如 RxJava 与 Project Reactor 这样的响应式框架的人来说，Flow 的设计也许看起来会非常熟悉。

确实，其设计灵感来源于响应式流以及其各种实现。但是 Flow 的主要目标是拥有尽可能简单的设计，对 Kotlin 以及挂起友好且遵从结构化并发。没有响应式的先驱及他们大量的工作，就不可能实现这一目标。你可以阅读 Reactive Streams and Kotlin Flows 这篇文章来了解完成 Flow 的故事。

虽然有所不同，但从概念上讲，Flow *依然是*响应式流，并且可以将它转换为响应式（规范及符合 TCK）的发布者（Publisher），反之亦然。这些开箱即用的转换器可以在 `kotlinx.coroutines` 提供的相关响应式模块（`kotlinx-coroutines-reactive` 用于 Reactive Streams，`kotlinx-coroutines-reactor` 用于 Project Reactor，以及 `kotlinx-coroutines-rx2` / `kotlinx-coroutines-rx3` 用于 RxJava2/RxJava3）中找到。集成模块包含 `Flow` 与其他实现之间的转换，与 Reactor 的 `Context` 集成以及与一系列响应式实体配合使用的挂起友好的使用方式。

目录

通道

延期的值提供了一种便捷的方法使单个值在多个协程之间进行相互传输。通道提供了一种在流中传输值的方法。

通道基础

一个 Channel 是一个和 `BlockingQueue` 非常相似的概念。其中一个不同是它代替了阻塞的 `put` 操作并提供了挂起的 send，还替代了阻塞的 `take` 操作并提供了挂起的 receive。

```
val channel = Channel<Int>()
launch {
    // 这里可能是消耗大量 CPU 运算的异步逻辑，我们将仅仅做 5 次整数的平方并发送
    for (x in 1..5) channel.send(x * x)
}
// 这里我们打印了 5 次被接收的整数:
repeat(5) { println(channel.receive()) }
println("Done!")
```

> 可以在这里获取完整代码。

这段代码的输出如下：

```
1
4
9
16
25
Done!
```

关闭与迭代通道

和队列不同，一个通道可以通过被关闭来表明没有更多的元素将会进入通道。在接收者中可以定期的使用 `for` 循环来从通道中接收元素。

从概念上来说，一个 close 操作就像向通道发送了一个特殊的关闭指令。这个迭代停止就说明关闭指令已经被接收了。所以这里保证所有先前发送出去的元素都在通道关闭前被接收到。

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x * x)
    channel.close() // 我们结束发送
}
// 这里我们使用 `for` 循环来打印所有被接收到的元素（直到通道被关闭）
for (y in channel) println(y)
println("Done!")
```

可以在这里获取完整代码。

构建通道生产者

协程生成一系列元素的模式很常见。这是 *生产者——消费者* 模式的一部分，并且经常能在并发的代码中看到它。你可以将生产者抽象成一个函数，并且使通道作为它的参数，但这与必须从函数中返回结果的常识相违悖。

这里有一个名为 produce 的便捷的协程构建器，可以很容易的在生产者端正确工作，并且我们使用扩展函数 consumeEach 在消费者端替代 `for` 循环：

```
val squares = produceSquares()
squares.consumeEach { println(it) }
println("Done!")
```

可以在这里获取完整代码。

管道

管道是一种一个协程在流中开始生产可能无穷多个元素的模式：

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // 在流中开始从 1 生产无穷多个整数
}
```

并且另一个或多个协程开始消费这些流，做一些操作，并生产了一些额外的结果。在下面的例子中，对这些数字仅仅做了平方操作：

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

主要的代码启动并连接了整个管道：

```
val numbers = produceNumbers() // 从 1 开始生成整数
val squares = square(numbers) // 整数求平方
repeat(5) {
    println(squares.receive()) // 输出前五个
}
println("Done!") // 至此已完成
coroutineContext.cancelChildren() // 取消子协程
```

可以在这里获取完整代码。

所有创建了协程的函数被定义在了 CoroutineScope 的扩展上，所以我们可以依靠结构化并发来确保没有常驻在我们的应用程序中的全局协程。

使用管道的素数

让我们来展示一个极端的例子——在协程中使用一个管道来生成素数。我们开启了一个数字的无限序列。

```
fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // 开启了一个无限的整数流
}
```

在下面的管道阶段中过滤了来源于流中的数字，删除了所有可以被给定素数整除的数字。

```
fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

现在我们开启了一个从 2 开始的数字流管道，从当前的通道中取一个素数，并为每一个我们发现的素数启动一个流水线阶段：

```
numbersFrom(2) -> filter(2) -> filter(3) -> filter(5) -> filter(7) ……
```

下面的例子打印了前十个素数，在主线程的上下文中运行整个管道。直到所有的协程在该主协程 runBlocking 的作用域中被启动完成。我们不必使用一个显式的列表来保存所有被我们已经启动的协程。我们使用 cancelChildren 扩展函数在我们打印了前十个素数以后来取消所有的子协程。

```
var cur = numbersFrom(2)
repeat(10) {
    val prime = cur.receive()
    println(prime)
    cur = filter(cur, prime)
}
coroutineContext.cancelChildren() // 取消所有的子协程来让主协程结束
```

> 可以在这里获取完整代码。

这段代码的输出如下：

```
2
3
5
7
11
13
17
19
23
29
```

注意，你可以在标准库中使用 `iterator` 协程构建器来构建一个相似的管道。使用 `iterator` 替换 `produce`、`yield` 替换 `send`、`next` 替换 `receive`、`Iterator` 替换 `ReceiveChannel` 来摆脱协程作用域，你将不再需要 `runBlocking`。然而，如上所示，如果你在 Dispatchers.Default 上下文中运行它，使用通道的管道的好处在于它可以充分利用多核心 CPU。

不过，这是一种非常不切实际的寻找素数的方法。在实践中，管道调用了另外的一些挂起中的调用（就像异步调用远程服务）并且这些管道不能内置使用 `sequence`/`iterator`，因为它们不被允许随意的挂起，不像 `produce` 是完全异步的。

扇出

多个协程也许会接收相同的管道，在它们之间进行分布式工作。让我们启动一个定期产生整数的生产者协程（每秒十个数字）：

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // 从 1 开始
    while (true) {
        send(x++) // 产生下一个数字
        delay(100) // 等待 0.1 秒
    }
}
```

接下来我们可以得到几个处理器协程。在这个示例中，它们只是打印它们的 id 和接收到的数字：

```
fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

现在让我们启动五个处理器协程并让它们工作将近一秒。看看发生了什么：

```
val producer = produceNumbers()
repeat(5) { launchProcessor(it, producer) }
delay(950)
producer.cancel() // 取消协程生产者从而将它们全部杀死
```

> 可以在这里获取完整代码。

该输出将类似于如下所示，尽管接收每个特定整数的处理器 id 可能会不同：

```
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，取消生产者协程将关闭它的通道，从而最终终止处理器协程正在执行的此通道上的迭代。

还有，注意我们如何使用 `for` 循环显式迭代通道以在 `launchProcessor` 代码中执行扇出。与 `consumeEach` 不同，这个 `for` 循环是安全完美地使用多个协程的。如果其中一个处理器协程执行失败，其它的处理器协程仍然会继续处理通道，而通过 `consumeEach` 编写的处理器始终在正常或非正常完成时消耗（取消）底层通道。

扇入

多个协程可以发送到同一个通道。比如说，让我们创建一个字符串的通道，和一个在这个通道中以指定的延迟反复发送一个指定字符串的挂起函数：

```
suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

现在，我们启动了几个发送字符串的协程，让我们看看会发生什么（在示例中，我们在主线程的上下文中作为主协程的子协程来启动它们）：

```
val channel = Channel<String>()
launch { sendString(channel, "foo", 200L) }
launch { sendString(channel, "BAR!", 500L) }
repeat(6) { // 接收前六个
    println(channel.receive())
}
coroutineContext.cancelChildren() // 取消所有子协程来让主协程结束
```

> 可以在这里获取完整代码。

输出如下：

```
foo
foo
BAR!
foo
foo
BAR!
```

带缓冲的通道

到目前为止展示的通道都是没有缓冲区的。无缓冲的通道在发送者和接收者相遇时传输元素（也称“对接”）。如果发送先被调用，则它将被挂起直到接收被调用，如果接收先被调用，它将被挂起直到发送被调用。

Channel() 工厂函数与 produce 建造器通过一个可选的参数 `capacity` 来指定 *缓冲区大小*。缓冲允许发送者在被挂起前发送多个元素，就像 `BlockingQueue` 有指定的容量一样，当缓冲区被占满的时候将会引起阻塞。

看看如下代码的表现：

```
val channel = Channel<Int>(4) // 启动带缓冲的通道
val sender = launch { // 启动发送者协程
    repeat(10) {
        println("Sending $it") // 在每一个元素发送前打印它们
        channel.send(it) // 将在缓冲区被占满时挂起
    }
}
// 没有接收到东西……只是等待……
delay(1000)
sender.cancel() // 取消发送者协程
```

> 可以在这里获取完整代码。

使用缓冲通道并给 capacity 参数传入 *四* 它将打印“sending” *五* 次：

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

前四个元素被加入到了缓冲区并且发送者在试图发送第五个元素的时候被挂起。

通道是公平的

发送和接收操作是 *公平的* 并且尊重调用它们的多个协程。它们遵守先进先出原则，可以看到第一个协程调用 `receive` 并得到了元素。在下面的例子中两个协程“乒”和“乓”都从共享的“桌子”通道接收到这个“球”元素。

```
data class Ball(var hits: Int)

fun main() = runBlocking {
    val table = Channel<Ball>() // 一个共享的 table（桌子）
    launch { player("ping", table) }
    launch { player("pong", table) }
    table.send(Ball(0)) // 乒乓球
    delay(1000) // 延迟 1 秒钟
    coroutineContext.cancelChildren() // 游戏结束，取消它们
}

suspend fun player(name: String, table: Channel<Ball>) {
    for (ball in table) { // 在循环中接收球
        ball.hits++
        println("$name $ball")
        delay(300) // 等待一段时间
        table.send(ball) // 将球发送回去
    }
}
```

可以在这里得到完整代码

“乒”协程首先被启动，所以它首先接收到了球。甚至虽然“乒” 协程在将球发送会桌子以后立即开始接收，但是球还是被“乓” 协程接收了，因为它一直在等待着接收球：

```
ping Ball(hits=1)
pong Ball(hits=2)
ping Ball(hits=3)
pong Ball(hits=4)
```

注意，有时候通道执行时由于执行者的性质而看起来不那么公平。点击这个提案来查看更多细节。

计时器通道

计时器通道是一种特别的会合通道，每次经过特定的延迟都会从该通道进行消费并产生 `Unit`。虽然它看起来似乎没用，它被用来构建分段来创建复杂的基于时间的 produce 管道和进行窗口化操作以及其它时间相关的处理。可以在 select 中使用计时器通道来进行“打勾”操作。

使用工厂方法 ticker 来创建这些通道。为了表明不需要其它元素，请使用 ReceiveChannel.cancel 方法。

现在让我们看看它是如何在实践中工作的：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
    val tickerChannel = ticker(delayMillis = 100, initialDelayMillis = 0) //创建计时器通道
    var nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Initial element is available immediately: $nextElement") // no initial delay

    nextElement = withTimeoutOrNull(50) { tickerChannel.receive() } // all subsequent elements have
100ms delay
    println("Next element is not ready in 50 ms: $nextElement")

    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 100 ms: $nextElement")

    // 模拟大量消费延迟
    println("Consumer pauses for 150ms")
    delay(150)
    // 下一个元素立即可用
    nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Next element is available immediately after large consumer delay: $nextElement")
    // 请注意，`receive` 调用之间的暂停被考虑在内，下一个元素的到达速度更快
    nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
    println("Next element is ready in 50ms after consumer pause in 150ms: $nextElement")

    tickerChannel.cancel() // 表明不再需要更多的元素
}
```

> 可以在这里获取完整代码。

它的打印如下：

```
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
```

请注意，ticker 知道可能的消费者暂停，并且默认情况下会调整下一个生成的元素如果发生暂停则延迟，试图保持固定的生成元素率。

给可选的 `mode` 参数传入 TickerMode.FIXED_DELAY 可以保持固定元素之间的延迟。

目录

异常处理

本节内容涵盖了异常处理与在异常上取消。我们已经知道取消协程会在挂起点抛出 CancellationException 并且它会被协程的机制所忽略。在这里我们会看看在取消过程中抛出异常或同一个协程的多个子协程抛出异常时会发生什么。

异常的传播

协程构建器有两种形式：自动传播异常（launch 与 actor）或向用户暴露异常（async 与 produce）。当这些构建器用于创建一个*根*协程时，即该协程不是另一个协程的*子*协程，前者这类构建器将异常视为**未捕获**异常，类似 Java 的 `Thread.uncaughtExceptionHandler`，而后者则依赖用户来最终消费异常，例如通过 await 或 receive（produce 与 receive 的相关内容包含于通道章节）。

可以通过一个使用 GlobalScope 创建根协程的简单示例来进行演示：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = GlobalScope.launch { // launch 根协程
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException() // 我们将在控制台打印 Thread.defaultUncaughtExceptionHandler
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async { // async 根协程
        println("Throwing exception from async")
        throw ArithmeticException() // 没有打印任何东西，依赖用户去调用等待
    }
    try {
        deferred.await()
        println("Unreached")
    } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
    }
}
```

> 可以在这里获取完整代码。

这段代码的输出如下（调试）：

```
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

It is possible to customize the default behavior of printing **uncaught** exceptions to the console. CoroutineExceptionHandler context element on a *root* coroutine can be used as generic `catch` block for this root coroutine and all its children where custom exception handling may take place. It is similar to `Thread.uncaughtExceptionHandler`. You cannot recover from the exception in the `CoroutineExceptionHandler`. The coroutine had already completed with the corresponding exception when the handler is called. Normally, the handler is used to log the exception, show some kind of error message, terminate, and/or restart the application.

在 JVM 中可以重定义一个全局的异常处理者来将所有的协程通过 `ServiceLoader` 注册到 CoroutineExceptionHandler。全局异常处理者就如同 `Thread.defaultUncaughtExceptionHandler` 一样，在没有更多的指定的异常处理者被注册的时候被使用。在 Android 中，`uncaughtExceptionPreHandler` 被设置在全局协程异常处理者中。

`CoroutineExceptionHandler` is invoked only on **uncaught** exceptions — exceptions that were not handled in any other way. In particular, all *children* coroutines (coroutines created in the context of another Job) delegate handling of their exceptions to their parent coroutine, which also delegates to the parent, and so on until the root, so the `CoroutineExceptionHandler` installed in their context is never used. In addition to that, async builder always catches all exceptions and represents them in the resulting Deferred object, so its `CoroutineExceptionHandler` has no effect either.

> Coroutines running in supervision scope do not propagate exceptions to their parent and are excluded from this rule. A further Supervision section of this document gives more details.

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("CoroutineExceptionHandler got $exception")
}
val job = GlobalScope.launch(handler) { // root coroutine, running in GlobalScope
    throw AssertionError()
}
val deferred = GlobalScope.async(handler) { // also root, but async instead of launch
    throw ArithmeticException() // 没有打印任何东西，依赖用户去调用 deferred.await()
}
joinAll(job, deferred)
```

> 可以在这里获取完整代码。

这段代码的输出如下：

```
CoroutineExceptionHandler got java.lang.AssertionError
```

取消与异常

取消与异常紧密相关。协程内部使用 `CancellationException` 来进行取消，这个异常会被所有的处理者忽略，所以那些可以被 `catch` 代码块捕获的异常仅仅应该被用来作为额外调试信息的资源。当一个协程使用 Job.cancel 取消的时候，它会被终止，但是它不会取消它的父协程。

```
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()
```

> 可以在这里获取完整代码。

这段代码的输出如下：

```
Cancelling child
Child is cancelled
Parent is not cancelled
```

如果一个协程遇到了 `CancellationException` 以外的异常，它将使用该异常取消它的父协程。这个行为无法被覆盖，并且用于为结构化的并发（structured concurrency）提供稳定的协程层级结构。CoroutineExceptionHandler 的实现并不是用于子协程。

> 在本例中，CoroutineExceptionHandler 总是被设置在由 GlobalScope 启动的协程中。将异常处理者设置在 runBlocking 主作用域内启动的协程中是没有意义的，尽管子协程已经设置了异常处理者，但是主协程也总是会被取消的。

当父协程的所有子协程都结束后，原始的异常才会被父协程处理，见下面这个例子。

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("CoroutineExceptionHandler got $exception")
}
val job = GlobalScope.launch(handler) {
    launch { // 第一个子协程
        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 { // 第二个子协程
        delay(10)
        println("Second child throws an exception")
        throw ArithmeticException()
    }
}
job.join()
```

可以在这里获取完整代码。

这段代码的输出如下：

```
Second child throws an exception
Children are cancelled, but exception is not handled until all children terminate
The first child finished its non cancellable block
CoroutineExceptionHandler got java.lang.ArithmeticException
```

异常聚合

当协程的多个子协程因异常而失败时，一般规则是“取第一个异常”，因此将处理第一个异常。在第一个异常之后发生的所有其他异常都作为被抑制的异常绑定至第一个异常。

```
import kotlinx.coroutines.*
import java.io.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception with suppressed
${exception.suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
                delay(Long.MAX_VALUE) // 当另一个同级的协程因 IOException  失败时，它将被取消
            } finally {
                throw ArithmeticException() // 第二个异常
            }
        }
        launch {
            delay(100)
            throw IOException() // 首个异常
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}
```

可以在这里获取完整代码。

注意：上面的代码将只在 JDK7 以上支持 suppressed 异常的环境中才能正确工作。

这段代码的输出如下：

```
CoroutineExceptionHandler got java.io.IOException with suppressed [java.lang.ArithmeticException]
```

注意，这个机制当前只能在 Java 1.7 以上的版本中使用。在 JS 和原生环境下暂时会受到限制，但将来会取消。

取消异常是透明的，默认情况下是未包装的：

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("CoroutineExceptionHandler got $exception")
}
val job = GlobalScope.launch(handler) {
    val inner = launch { // 该栈内的协程都将被取消
        launch {
            launch {
                throw IOException() // 原始异常
            }
        }
    }
    try {
        inner.join()
    } catch (e: CancellationException) {
        println("Rethrowing CancellationException with original cause")
        throw e // 取消异常被重新抛出，但原始 IOException 得到了处理
    }
}
job.join()
```

可以在这里获取完整代码。

这段代码的输出如下：

```
Rethrowing CancellationException with original cause
CoroutineExceptionHandler got java.io.IOException
```

监督

正如我们之前研究的那样，取消是在协程的整个层次结构中传播的双向关系。让我们看一下需要单向取消的情况。

此类需求的一个良好示例是在其作用域内定义作业的 UI 组件。如果任何一个 UI 的子作业执行失败了，它并不总是有必要取消（有效地杀死）整个 UI 组件，但是如果 UI 组件被销毁了（并且它的作业也被取消了），由于它的结果不再被需要了，它有必要使所有的子作业执行失败。

另一个例子是服务进程孵化了一些子作业并且需要 *监督* 它们的执行，追踪它们的故障并在这些子作业执行失败的时候重启。

监督作业

SupervisorJob 可以用于这些目的。它类似于常规的 Job，唯一的不同是：SupervisorJob 的取消只会向下传播。这是很容易用以下示例演示：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        // 启动第一个子作业——这个示例将会忽略它的异常（不要在实践中这么做！）
        val firstChild = launch(CoroutineExceptionHandler { _, _ ->  }) {
            println("The first child is failing")
            throw AssertionError("The first child is cancelled")
        }
        // 启动第二个子作业
        val secondChild = launch {
            firstChild.join()
            // 取消了第一个子作业且没有传播给第二个子作业
            println("The first child is cancelled: ${firstChild.isCancelled}, but the second one is
still active")
            try {
                delay(Long.MAX_VALUE)
            } finally {
                // 但是取消了监督的传播
                println("The second child is cancelled because the supervisor was cancelled")
            }
        }
        // 等待直到第一个子作业失败且执行完成
        firstChild.join()
        println("Cancelling the supervisor")
        supervisor.cancel()
        secondChild.join()
    }
}
```

可以在这里获取完整代码。

这段代码的输出如下：

```
The first child is failing
The first child is cancelled: true, but the second one is still active
Cancelling the supervisor
The second child is cancelled because the supervisor was cancelled
```

监督作用域

对于*作用域*的并发，可以用 supervisorScope 来替代 coroutineScope 来实现相同的目的。它只会单向的传播并且当作业自身执行失败的时候将所有子作业全部取消。作业自身也会在所有的子作业执行结束前等待，就像 coroutineScope 所做的那样。

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("The child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("The child is cancelled")
                }
            }
            // 使用 yield 来给我们的子作业一个机会来执行打印
            yield()
            println("Throwing an exception from the scope")
            throw AssertionError()
        }
    } catch(e: AssertionError) {
        println("Caught an assertion error")
    }
}
```

可以在这里获取完整代码。

这段代码的输出如下：

```
The child is sleeping
Throwing an exception from the scope
The child is cancelled
Caught an assertion error
```

监督协程中的异常

常规的作业和监督作业之间的另一个重要区别是异常处理。监督协程中的每一个子作业应该通过异常处理机制处理自身的异常。这种差异来自于子作业的执行失败不会传播给它的父作业的事实。这意味着在 supervisorScope 内部直接启动的协程*确实*使用了设置在它们作用域内的 CoroutineExceptionHandler，与父协程的方式相同（参见 CoroutineExceptionHandler 小节以获知更多细节）。

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("The child throws an exception")
            throw AssertionError()
        }
        println("The scope is completing")
    }
    println("The scope is completed")
}
```

可以在这里获取完整代码。

这段代码的输出如下：

```
The scope is completing
The child throws an exception
CoroutineExceptionHandler got java.lang.AssertionError
The scope is completed
```

目录

共享的可变状态与并发

协程可用多线程调度器(比如默认的 Dispatchers.Default)并发执行。这样就可以提出所有常见的并发问题。主要的问题是同步访问**共享的可变状态**。协程领域对这个问题的一些解决方案类似于多线程领域中的解决方案，但其它解决方案则是独一无二的。

问题

我们启动一百个协程，它们都做一千次相同的操作。我们同时会测量它们的完成时间以便进一步的比较：

```
suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程重复执行同一动作的次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用域
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

我们从一个非常简单的动作开始：使用多线程的 Dispatchers.Default 来递增一个共享的可变变量。

```
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

> 可以在这里获取完整代码。

这段代码最后打印出什么结果?它不太可能打印出“Counter = 100000”，因为一百个协程在多个线程中同时递增计数器但没有做并发处理。

volatile 无济于事

有一种常见的误解：volatile 可以解决并发问题。让我们尝试一下：

```
@Volatile // 在 Kotlin 中 `volatile` 是一个注解
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

> 可以在这里获取完整代码。

这段代码运行速度更慢了，但我们最后仍然没有得到“Counter = 100000”这个结果，因为 volatile 变量保证可线性化（这是“原子”的技术术语）读取和写入变量，但在大量动作（在我们的示例中即“递增”操作）发生时并不提供原子性。

线程安全的数据结构

一种对线程、协程都有效的常规解决方法，就是使用线程安全（也称为同步的、可线性化、原子）的数据结构，它为需要在共享状态上执行的相应操作提供所有必需的同步处理。在简单的计数器场景中，我们可以使用具有 `incrementAndGet` 原子操作的 `AtomicInteger` 类：

```
val counter = AtomicInteger()

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter.incrementAndGet()
        }
    }
    println("Counter = $counter")
}
```

> 可以在这里获取完整代码。

这是针对此类特定问题的最快解决方案。它适用于普通计数器、集合、队列和其他标准数据结构以及它们的基本操作。然而，它并不容易被扩展来应对复杂状态、或一些没有现成的线程安全实现的复杂操作。

以细粒度限制线程

限制线程 是解决共享可变状态问题的一种方案：对特定共享状态的所有访问权都限制在单个线程中。它通常应用于 UI 程序中：所有 UI 状态都局限于单个事件分发线程或应用主线程中。这在协程中很容易实现，通过使用一个单线程上下文：

```
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // 将每次自增限制在单线程上下文中
            withContext(counterContext) {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
```

可以在这里获取完整代码。

这段代码运行非常缓慢，因为它进行了 *细粒度* 的线程限制。每个增量操作都得使用 [withContext(counterContext)] 块从多线程 Dispatchers.Default 上下文切换到单线程上下文。

以粗粒度限制线程

在实践中，线程限制是在大段代码中执行的，例如：状态更新类业务逻辑中大部分都是限于单线程中。下面的示例演示了这种情况，在单线程上下文中运行每个协程。

```
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    // 将一切都限制在单线程上下文中
    withContext(counterContext) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
```

可以在这里获取完整代码。

这段代码运行更快而且打印出了正确的结果。

互斥

该问题的互斥解决方案：使用永远不会同时执行的 *关键代码块* 来保护共享状态的所有修改。在阻塞的世界中，你通常会为此目的使用 `synchronized` 或者 `ReentrantLock`。在协程中的替代品叫做 Mutex。它具有 lock 和 unlock 方法，可以隔离关键的部分。关键的区别在于 `Mutex.lock()` 是一个挂起函数，它不会阻塞线程。

还有 withLock 扩展函数，可以方便的替代常用的 `mutex.lock(); try { …… } finally { mutex.unlock() }` 模式：

```
val mutex = Mutex()
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // 用锁保护每次自增
            mutex.withLock {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
```

可以在这里获取完整代码。

此示例中锁是细粒度的，因此会付出一些代价。但是对于某些必须定期修改共享状态的场景，它是一个不错的选择，但是没有自然线程可以限制此状态。

Actors

一个 actor 是由协程、被限制并封装到该协程中的状态以及一个与其它协程通信的 *通道* 组合而成的一个实体。一个简单的 actor 可以简单的写成一个函数，但是一个拥有复杂状态的 actor 更适合由类来表示。

有一个 actor 协程构建器，它可以方便地将 actor 的邮箱通道组合到其作用域中（用来接收消息）、组合发送 channel 与结果集对象，这样对 actor 的单个引用就可以作为其句柄持有。

使用 actor 的第一步是定义一个 actor 要处理的消息类。Kotlin 的密封类很适合这种场景。我们使用 `IncCounter` 消息（用来递增计数器）和 `GetCounter` 消息（用来获取值）来定义 `CounterMsg` 密封类。后者需要发送回复。CompletableDeferred 通信原语表示未来可知（可传达）的单个值，这里被用于此目的。

```
// 计数器 Actor 的各种类型
sealed class CounterMsg
object IncCounter : CounterMsg() // 递增计数器的单向消息
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg() // 携带回复的请求
```

接下来我们定义一个函数，使用 actor 协程构建器来启动一个 actor：

```
// 这个函数启动一个新的计数器 actor
fun CoroutineScope.counterActor() = actor<CounterMsg> {
    var counter = 0 // actor 状态
    for (msg in channel) { // 即将到来消息的迭代器
        when (msg) {
            is IncCounter -> counter++
            is GetCounter -> msg.response.complete(counter)
        }
    }
}
```

main 函数代码很简单：

```
fun main() = runBlocking<Unit> {
    val counter = counterActor() // 创建该 actor
    withContext(Dispatchers.Default) {
        massiveRun {
            counter.send(IncCounter)
        }
    }
    // 发送一条消息以用来从一个 actor 中获取计数值
    val response = CompletableDeferred<Int>()
    counter.send(GetCounter(response))
    println("Counter = ${response.await()}")
    counter.close() // 关闭该actor
}
```

> 可以在这里获取完整代码。

actor 本身执行时所处上下文（就正确性而言）无关紧要。一个 actor 是一个协程，而一个协程是按顺序执行的，因此将状态限制到特定协程可以解决共享可变状态的问题。实际上，actor 可以修改自己的私有状态，但只能通过消息互相影响（避免任何锁定）。

actor 在高负载下比锁更有效，因为在这种情况下它总是有工作要做，而且根本不需要切换到不同的上下文。

> 注意，actor 协程构建器是一个双重的 produce 协程构建器。一个 actor 与它接收消息的通道相关联，而一个 producer 与它发送元素的通道相关联。

目录

select 表达式（实验性的）

select 表达式可以同时等待多个挂起函数，并 *选择* 第一个可用的。

> Select 表达式在 `kotlinx.coroutines` 中是一个实验性的特性。这些 API 在 `kotlinx.coroutines` 库即将到来的更新中可能会发生改变。

在通道中 select

我们现在有两个字符串生产者：`fizz` 和 `buzz` 。其中 `fizz` 每 300 毫秒生成一个“Fizz”字符串：

```
fun CoroutineScope.fizz() = produce<String> {
    while (true) { // 每 300 毫秒发送一个 "Fizz"
        delay(300)
        send("Fizz")
    }
}
```

接着 `buzz` 每 500 毫秒生成一个 “Buzz!” 字符串：

```
fun CoroutineScope.buzz() = produce<String> {
    while (true) { // 每 500 毫秒发送一个"Buzz!"
        delay(500)
        send("Buzz!")
    }
}
```

使用 receive 挂起函数，我们可以从两个通道接收 *其中一个* 的数据。但是 select 表达式允许我们使用其 onReceive 子句 *同时* 从两者接收：

```
suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>) {
    select<Unit> { // <Unit> 意味着该 select 表达式不返回任何结果
        fizz.onReceive { value ->  // 这是第一个 select 子句
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // 这是第二个 select 子句
            println("buzz -> '$value'")
        }
    }
}
```

让我们运行代码 7 次：

```
val fizz = fizz()
val buzz = buzz()
repeat(7) {
    selectFizzBuzz(fizz, buzz)
}
coroutineContext.cancelChildren() // 取消 fizz 和 buzz 协程
```

可以在这里获取完整代码。

这段代码的执行结果如下：

```
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
buzz -> 'Buzz!'
```

通道关闭时 select

select 中的 onReceive 子句在已经关闭的通道执行会发生失败，并导致相应的 `select` 抛出异常。我们可以使用 onReceiveOrNull 子句在关闭通道时执行特定操作。以下示例还显示了 `select` 是一个返回其查询方法结果的表达式：

```
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'"
        }
    }
```

注意，onReceiveOrNull 是一个仅在用于不可空元素的通道上定义的扩展函数，以使关闭的通道与空值之间不会出现意外的混乱。

现在有一个生成四次“Hello”字符串的 `a` 通道，和一个生成四次“World”字符串的 `b` 通道，我们在这两个通道上使用它：

```
val a = produce<String> {
    repeat(4) { send("Hello $it") }
}
val b = produce<String> {
    repeat(4) { send("World $it") }
}
repeat(8) { // 打印最早的八个结果
    println(selectAorB(a, b))
}
coroutineContext.cancelChildren()
```

可以在这里获取完整代码。

这段代码的结果非常有趣，所以我们将在细节中分析它：

```
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
```

有几个结果可以通过观察得出。

首先，`select` *偏向于* 第一个子句，当可以同时选到多个子句时，第一个子句将被选中。在这里，两个通道都在不断地生成字符串，因此 `a` 通道作为 select 中的第一个子句获胜。然而因为我们使用的是无缓冲通道，所以 `a` 在其调用 send 时会不时地被挂起，进而 `b` 也有机会发送。

第二个观察结果是，当通道已经关闭时，会立即选择 onReceiveOrNull。

Select 以发送

Select 表达式具有 onSend 子句，可以很好的与选择的偏向特性结合使用。

我们来编写一个整数生成器的示例，当主通道上的消费者无法跟上它时，它会将值发送到 `side` 通道上：

```
fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // 生产从 1 到 10 的 10 个数值
        delay(100) // 延迟 100 毫秒
        select<Unit> {
            onSend(num) {} // 发送到主通道
            side.onSend(num) {} // 或者发送到 side 通道
        }
    }
}
```

消费者将会非常缓慢，每个数值处理需要 250 毫秒：

```
val side = Channel<Int>() // 分配 side 通道
launch { // 对于 side 通道来说，这是一个很快的消费者
    side.consumeEach { println("Side channel has $it") }
}
produceNumbers(side).consumeEach {
    println("Consuming $it")
    delay(250) // 不要着急，让我们正确消化消耗被发送来的数字
}
println("Done consuming")
coroutineContext.cancelChildren()
```

> 可以在这里获取完整代码。

让我们看看会发生什么：

```
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
```

Select 延迟值

延迟值可以使用 onAwait 子句查询。让我们启动一个异步函数，它在随机的延迟后会延迟返回字符串：

```
fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}
```

让我们随机启动十余个异步函数，每个都延迟随机的时间。

```
fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}
```

现在 main 函数在等待第一个函数完成，并统计仍处于激活状态的延迟值的数量。注意，我们在这里使用 `select` 表达式事实上是作为一种 Kotlin DSL，所以我们可以用任意代码为它提供子句。在这种情况下，我们遍历一个延迟值的队列，并为每个延迟值提供 `onAwait` 子句的调用。

```
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")
```

> 可以在这里获取完整代码。

该输出如下：

```
Deferred 4 produced answer 'Waited for 128 ms'
11 coroutines are still active
```

在延迟值通道上切换

我们现在来编写一个通道生产者函数，它消费一个产生延迟字符串的通道，并等待每个接收的延迟值，但它只在下一个延迟值到达或者通道关闭之前处于运行状态。此示例将 onReceiveOrNull 和 onAwait 子句放在同一个 `select` 中：

```
fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = produce<String> {
    var current = input.receive() // 从第一个接收到的延迟值开始
    while (isActive) { // 循环直到被取消或关闭
        val next = select<Deferred<String>?> { // 从这个 select 中返回下一个延迟值或 null
            input.onReceiveOrNull { update ->
                update // 替换下一个要等待的值
            }
            current.onAwait { value ->
                send(value) // 发送当前延迟生成的值
                input.receiveOrNull() // 然后使用从输入通道得到的下一个延迟值
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // 跳出循环
        } else {
            current = next
        }
    }
}
```

为了测试它，我们将用一个简单的异步函数，它在特定的延迟后返回特定的字符串：

```
fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}
```

main 函数只是启动一个协程来打印 `switchMapDeferreds` 的结果并向它发送一些测试数据：

```
val chan = Channel<Deferred<String>>() // 测试使用的通道
launch { // 启动打印协程
    for (s in switchMapDeferreds(chan))
        println(s) // 打印每个获得的字符串
}
chan.send(asyncString("BEGIN", 100))
delay(200) // 充足的时间来生产 "BEGIN"
chan.send(asyncString("Slow", 500))
delay(100) // 不充足的时间来生产 "Slow"
chan.send(asyncString("Replace", 100))
delay(500) // 在最后一个前给它一点时间
chan.send(asyncString("END", 500))
delay(1000) // 给执行一段时间
chan.close() // 关闭通道……
delay(500) // 然后等待一段时间来让它结束
```

> 可以在这里获取完整代码。

这段代码的执行结果：

```
BEGIN
Replace
END
Channel was closed
```

多平台程序设计

平台相关声明

> 多平台项目是 Kotlin 1.2 与 Kotlin 1.3 中的一个实验性特性。本文档中描述的所有语言与工具特性在未来的 Kotlin 版本中都可能会有所变化。

Kotlin 多平台代码的一个关键功能是让公共代码能够依赖平台相关声明的一种方式。在其他语言中，这通常可以通过在公共代码中构建一组接口并在平台相关模块中实现这些接口来完成。然而，当在其中某个平台上有一个实现所需功能的库，并且希望直接使用该库的 API 而无需额外包装器时，这种方法并不理想。此外，它需要以接口表示公共声明，这无法覆盖所有可能情况。

作为替代方案，Kotlin 提供了一种 *预期声明与实际声明* 的机制。利用这种机制，公共模块可以定义 *预期声明*，而平台模块可以提供与预期声明相对应的 *实际声明*。为了了解其工作机制，我们先来看一个示例。此代码是公共模块的一部分：

```
package org.jetbrains.foo

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

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

而这是相应的 JVM 模块：

```
package org.jetbrains.foo

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

这阐明了几个要点：

— 公共模块中的预期声明与其对应的实际声明始终具有完全相同的完整限定名。

— 预期声明标有 `expect` 关键字；实际声明标有 `actual` 关键字。

— 与预期声明的任何部分匹配的所有实际声明都需要标记为 `actual`。

— 预期声明决不包含任何实现代码。

请注意，预期声明并不限于接口与接口成员。在本例中，预期的类有一个构造函数，于是可以直接在公共代码中创建该类。还可以将 `expect` 修饰符应用于其他声明，包括顶层声明以及注解：

```
// 公共
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
```

编译器会确保每个预期声明在实现相应公共模块的所有平台模块中都具有实际声明，并且如果缺少任何实际声明，编译器都会报错。IDE 提供了帮助创建所缺失实际声明的工具。

如果有一个希望用在公共代码中的平台相关的库，同时为其他平台提供自己的实现，那么可以将现有类的别名作为实际声明：

```
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 构建多平台项目

> 多平台项目是 Kotlin 1.2 与 Kotlin 1.3 中的一个实验性特性。本文档中描述的所有语言与工具特性在未来的 Kotlin 版本中都可能会有所变化。

本文档解释了 Kotlin 多平台项目的结构，并描述了如何使用 Gradle 配置与构建这些项目。

目录

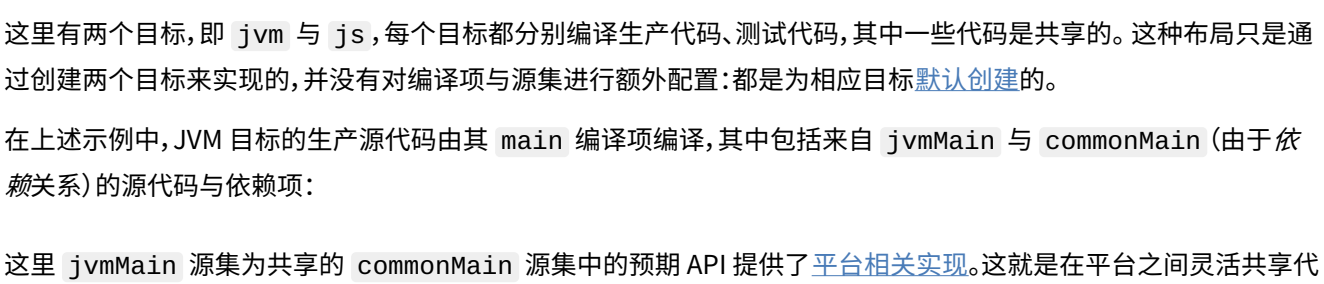
项目结构

Kotlin 多平台项目的布局由以下构建块构成：

- 目标是构建的一部分，负责构建、测试及打包其中一个平台的完整软件。因此，多平台项目通常包含多个目标。
- 构建每个目标涉及一到多次编译 Kotlin 源代码。换句话说，一个目标可能有一到多个编译项。例如，一个编译项用于编译生产代码，另一个用于编译测试代码。
- Kotlin 源代码会放到源集中。除了 Kotlin 源文件与资源外，每个源集都可能有自己的依赖项。源集之间以 *“依赖”*关系构成了层次结构。源集本身是平台无关的，但是如果一个源集只面向单一平台编译，那么它可能包含平台相关代码与依赖项。

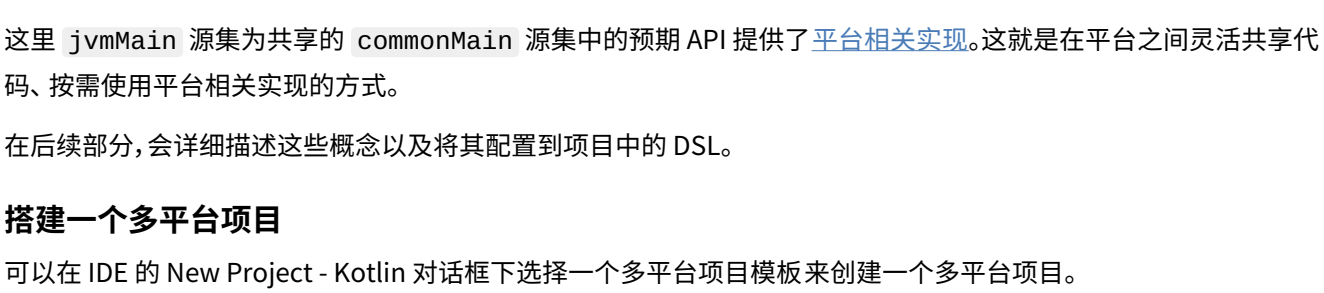
每个编译项都有一个默认源集，是放置该编译项的源代码与依赖项的地方。默认源集还用于通过“依赖”关系将其他源集引到该编译项中。

以下是一个面向 JVM 与 JS 的项目的图示：



这里有两个目标，即 `jvm` 与 `js`，每个目标都分别编译生产代码、测试代码，其中一些代码是共享的。这种布局只是通过创建两个目标来实现的，并没有对编译项与源集进行额外配置：都是为相应目标默认创建的。

在上述示例中，JVM 目标的生产源代码由其 `main` 编译项编译，其中包括来自 `jvmMain` 与 `commonMain`（由于*依赖*关系）的源代码与依赖项：



这里 `jvmMain` 源集为共享的 `commonMain` 源集中的预期 API 提供了平台相关实现。这就是在平台之间灵活共享代码、按需使用平台相关实现的方式。

在后续部分，会详细描述这些概念以及将其配置到项目中的 DSL。

搭建一个多平台项目

可以在 IDE 的 New Project - Kotlin 对话框下选择一个多平台项目模板来创建一个多平台项目。

例如，如果选择了“Kotlin (Multiplatform Library)”，会创建一个包含三个目标的库项目，其中一个用于 JVM，一个用于 JS，还有一个用于您正在使用的原生平台。这些是在 `build.gradle` 脚本中以下列方式配置的：

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.72'
}

repositories {
    mavenCentral()
}

kotlin {
    jvm() // 使用默认名称 “jvm” 创建一个 JVM 目标
    js()  // 使用名称 “js” 创建一个 JS 目标
    mingwX64("mingw") // 使用名称 “mingw” 创建一个 Windows (MinGW X64) 目标

    sourceSets { /* …… */ }
}
```

```
plugins {
    kotlin("multiplatform") version "1.3.72"
}

repositories {
    mavenCentral()
}

kotlin {
    jvm() // 使用默认名称 “jvm” 创建一个 JVM 目标
    js()  // 使用名称 “js” 创建一个 JS 目标
    mingwX64("mingw") // 使用名称 “mingw” 创建一个 Windows (MinGW X64) 目标

    sourceSets { /* …… */ }
}
```

这三个目标是通过预设函数 `jvm()`、`js()` 与 `mingwX64()` 创建的，它们提供了一些默认项目布局。每个已支持平台都有预设的函数。

然后配置源集及其依赖，如下所示：

```
plugins { /* …… */ }

kotlin {
    /* 省略目标声明 */

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

        // 仅用于 JVM 的源码及其依赖的默认源集
        // 或者使用 jvmMain { …… }
        jvm().compilations.main.defaultSourceSet {
            dependencies {
                implementation kotlin('stdlib-jdk8')
            }
        }
        // 仅用于 JVM 的测试及其依赖
        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 {
    /* 省略目标声明 */

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

        // 仅用于 JVM 的源码及其依赖的默认源集
        jvm().compilations["main"].defaultSourceSet {
            dependencies {
                implementation(kotlin("stdlib-jdk8"))
            }
        }
        // 仅用于 JVM 的测试及其依赖
        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 { /* …… */ }
    }
}
```

这些在上面配置的目标的生产与测试的源码都有各自的默认源集名称。源集 `commonMain` 与 `commonTest` 将被分别包含在所有目标的生产与测试编译项中。需要注意的是，公共源集 `commonMain` 与 `commonTest` 的依赖使用的都是公共构件，而平台库将转到特定目标的源集。

Gradle 插件

Kotlin 多平台项目需要 Gradle 4.7 及以上版本，不支持旧版本的 Gradle。

如需在 Gradle 项目中从头开始设置多平台项目，首先要将 `kotlin-multiplatform` 插件应用到项目中，即在 `build.gradle` 文件的开头添加以下内容：

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.72'
}
```

```
plugins {
    kotlin("multiplatform") version "1.3.72"
}
```

这会在顶层创建 `kotlin` 扩展。然后可以在构建脚本中访问该扩展，来：

— 为多个平台设置目标（默认不会创建目标）；

— 配置源集及其依赖；

设置目标

目标是构建的一部分，负责编译，测试与打包针对一个已支持平台的软件。

所有的目标可能共享一些源代码，也可能拥有平台专用的源代码。

由于平台的不同，目标也以不同的方式构建，并且拥有各个平台专用的设置。Gradle 插件捆绑了一些已支持平台的预设。

要创建一个目标，请使用其中一个预设函数，这些预置函数根据目标平台命名，并可选择接收一个目标名称与一个配置代码块：

```
kotlin {
    jvm() // 用默认名称 “jvm” 创建一个 JVM 目标
    js("nodeJs") // 用自定义名称 “nodeJs” 创建一个 JS 目标

    linuxX64("linux") {
        /* 在此处指定 “linux” 的其他设置 */
    }
}
```

如果存在，这些预置函数将返回一个现有的目标。这可以用于配置一个现有的目标：

```
kotlin {
    /* …… */

    // 配置 “jvm6” 目标的属性
    jvm("jvm6").attributes { /* …… */ }
}
```

注意目标平台与命名都很重要：如果一个目标作为 `jvm('jvm6')` 创建，使用 `jvm()` 将会创建一个单独的目标（使用默认名称 `jvm`）。如果用于创建该名称下的预设函数不同，将会报告一个错误。

从预置函数创建的目标将被添加到域对象集合 `kotlin.targets` 中，这可以用于通过名称访问它们或者配置所有目标：

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

    println(targets.names) // 打印：[jvm, metadata, nodeJs]

    // 配置所有的目标，包括稍后添加的目标
    targets.all {
        compilations["main"].defaultSourceSet { /* …… */ }
    }
}
```

要从动态创建或访问多个预设中的多个目标，你可以使用 `targetFromPreset` 函数，它接收一个接收预设（那些被包含在 `kotlin.presets` 域对象集合中的），以及可选的目标名称与配置的代码块。

例如，要为每一个 Kotlin/Native 支持的平台（见下文）创建目标，使用以下代码：

```
kotlin {
    presets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTargetPreset).each {
        targetFromPreset(it) {
            /* 配置每个已创建的目标 */
        }
    }
}
```

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

/* …… */

kotlin {
    presets.withType<KotlinNativeTargetPreset>().forEach {
        targetFromPreset(it) {
            /* 配置每个已创建的目标 */
        }
    }
}
```

已支持平台

如上所示，对于以下目标平台，可以使用预设函数应用目标平台预设：

- `jvm` 用于 Kotlin/JVM；
- `js` 用于 Kotlin/JS；
- `android` 用于 Android 应用程序与库。请注意在创建目标之前，应该应用其中之一的 Android Gradle 插件；
- Kotlin/Native 目标平台预设（参见下文备注）：
  - `androidNativeArm32` 与 `androidNativeArm64` 用于 Android NDK；
  - `iosArm32`、`iosArm64`、`iosX64` 用于 iOS；
  - `watchosArm32`、`watchosArm64`、`watchosX86` 用于 watchOS；
  - `tvosArm64`、`tvosX64` 用于 tvOS；
  - `linuxArm64`、`linuxArm32Hfp`、`linuxMips32`、`linuxMipsel32`、`linuxX64` 用于 Linux；
  - `macosX64` 用于 MacOS；
  - `mingwX64` 与 `mingwX86` 用于 Windows；
  - `wasm32` 用于 WebAssembly。

  请注意，某些 Kotlin/Native 目标平台需要适宜的主机来构建。

某些目标平台可能需要附加配置。Android 与 iOS 示例请参见多平台项目：iOS 与 Android 教程。

配置编译项

构建目标需要一次或多次编译 Kotlin。目标的每次 Kotlin 编译项都可以用于不同的目的（例如生产代码，测试），并包含不同的源集。可以在 DSL 中访问目标的编译项，例如，配置Kotlin 编译器选项或者获取依赖项文件和编译项输出用来获取任务。

```
kotlin {
    jvm {
        compilations.main.kotlinOptions {
            // 为“main”编译项设置 Kotlin 编译器选项：
            jvmTarget = "1.8"
        }

        compilations.main.compileKotlinTask // 获取 Kotlin 任务“compileKotlinJvm”
        compilations.main.output // 获取 main 编译项输出
        compilations.test.runtimeDependencyFiles // 获取测试运行时路径
    }

    // 配置所有目标的所有编译项：
```

```
    targets.all {
        compilations.all {
            kotlinOptions {
                allWarningsAsErrors = true
            }
        }
    }
}
```

```
kotlin {
    jvm {
        val main by compilations.getting {
            kotlinOptions {
                // 为“main”编译项设置 Kotlin 编译器选项:
                jvmTarget = "1.8"
            }

            compileKotlinTask // 获取 Kotlin 任务“compileKotlinJvm”
            output // 获取 main 编译项输出
        }

        compilations["test"].runtimeDependencyFiles // 获取测试运行时路径
    }

    // 配置所有目标的所有编译项:
    targets.all {
        compilations.all {
            kotlinOptions {
                allWarningsAsErrors = true
            }
        }
    }
}
```

每个编译项都附带一个默认源集，该默认源集存储特定于该编译项的源和依赖项。目标 `bar` 的编译项 `foo` 的默认源集的名称为 `barFoo`。也可以使用 `defaultSourceSet` 从编译项中访问它：

```
kotlin {
    jvm() // 使用默认名称“jvm”创建一个 JVM 目标

    sourceSets {
        // “jvm”目标的“main”编译项的默认源集:
        jvmMain {
            /* …… */
        }
    }

    // 或者，从目标的编译项中访问它:
    jvm().compilations.main.defaultSourceSet {
        /* …… */
    }
}
```

```
kotlin {
    jvm() // 使用默认名称“jvm”创建一个 JVM 目标

    sourceSets {
        // “jvm”目标的“main”编译项的默认源集：
        val jvmMain by getting {
            /* …… */
        }
    }

    // 或者，从目标的编译项中访问它：
    jvm().compilations["main"].defaultSourceSet {
        /* …… */
    }
}
```

为了收集参与编译项的所有源集，包括通过依赖关系添加的源集，可以使用属性 `allKotlinSourceSets`。

对于某些特定用例，可能需要创建自定义编译项。这可以在目标的 `compilations` 领域对象集合中完成。请注意，需要为所有自定义编译项手动设置依赖项，并且自定义编译项输出的使用取决于构建所有者。例如，对目标 `jvm()` 的集成测试的自定义编译项：

```
kotlin {
    jvm() {
        compilations.create('integrationTest') {
            defaultSourceSet {
                dependencies {
                    def main = compilations.main
                    // 根据 main 编译项的编译类路径和输出进行编译：
                    implementation(main.compileDependencyFiles + main.output.classesDirs)
                    implementation kotlin('test-junit')
                    /* …… */
                }
            }

            // 创建一个测试任务来运行此编译项产生的测试：
            tasks.create('jvmIntegrationTest', Test) {
                // 使用包含编译依赖项（包括“main”）的类路径运行测试，
                // 运行时依赖项以及此编译项的输出：
                classpath = compileDependencyFiles + runtimeDependencyFiles + output.allOutputs

                // 仅运行此编译项输出中的测试：
                testClassesDirs = output.classesDirs
            }
        }
    }
}
```

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

            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
                        // 根据 main 编译项的编译类路径和输出进行编译：
                        implementation(main.compileDependencyFiles + main.output.classesDirs)
                        implementation(kotlin("test-junit"))
                        /* …… */
                    }
                }

                // 创建一个测试任务来运行此编译项产生的测试：
                tasks.create<Test>("integrationTest") {
                    // 使用包含编译依赖项（包括“main”）的类路径运行测试，
                    // 运行时依赖项以及此编译项的输出：
                    classpath = compileDependencyFiles + runtimeDependencyFiles + output.allOutputs

                    // 仅运行此编译项输出中的测试：
                    testClassesDirs = output.classesDirs
                }
            }
        }
    }
}
```

还要注意，默认情况下，自定义编译项的默认源集既不依赖于 `commonMain` 也不依赖于 `commonTest`。

配置源集

Kotlin 源集是 Kotlin 源代码及其资源、依赖关系以及语言设置的集合，一个源集可能会参与一个或多个目标的 Kotlin 编译项。

源集不限于平台特定的或“共享的”；允许包含的内容取决于其用法：添加到多个编译项中的源集仅限于通用语言特性及依赖项，仅由单个目标使用的源集可以具有平台特定的依赖项，并且其代码可能使用目标平台特定的语言特性。

默认情况下会创建并配置一些源集：`commonMain`、`commonTest` 和编译项的默认源集。请参见默认项目布局。

源集在 `kotlin { ... }` 扩展的 `sourceSets { ... }` 块内配置：

```
kotlin {
    sourceSets {
        foo { /* …… */ } // 创建或配置名称为 “foo” 的源集
        bar { /* …… */ }
    }
}
```

```
kotlin {
    sourceSets {
        val foo by creating { /* …… */ } // 创建一个名为 “foo” 的新源集
        val bar by getting { /* …… */ } // 使用名称 “bar” 配置现有的源集
    }
}
```

> 注意：创建源集不会将其链接到任何目标。一些源集是预定义的 因此默认情况下进行编译。但是，始终需要将自定义源集明确地定向到编译项。请参见：关联源集。

源集名称区分大小写。在通过名称引用默认源集时，请确保源集的名称前缀与目标名称匹配，例如，目标 `iosX64` 的源集 `iosX64Main`。

源集本身是平台无关的，但是如果仅针对单个平台进行编译，则可以将其视为特定于平台的。因此，源集可以包含平台之间共享的公共代码或平台特定的代码。

每个源集都有 Kotlin 源代码的默认源目录：`src/<源集名称>/kotlin`。要将 Kotlin 源目录以及资源添加到源集中，请使用其 `kotlin` 与 `resources` `SourceDirectorySet`：

默认情况下，源集的文件存储在以下目录中：

- 源文件：`src/<source set name>/kotlin`
- 资源文件：`src/<source set name>/resources`

应该手动创建这些目录。

要将自定义 Kotlin 源目录和资源添加到源集中，请使用其 `kotlin` 与 `resources` `SourceDirectorySet`：

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

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

关联源集

Kotlin 源集可能与 *“depends on”* 关系有关，因而如果一个源集 `foo` 依赖于一个源集 `bar`，那么：

- 每当为特定目标编译 `foo` 时，`bar` 也参与到编译中，并且还会编译成相同的目标二进制格式，例如 JVM 类文件或者 JS 代码；
- `foo` 源中的代码能 “看到” `bar` 的定义，包括 `internal` 的以及 `bar` 的依赖，即使是被指定为 `implementation` 的依赖；
- `foo` 可能包含针对 `bar` 的预期定义的特定平台的实现；
- `bar` 的资源总是与 `foo` 的资源一起处理与复制；
- `foo` 与 `bar` 的语言应该是一致的；

不允许源集间循环依赖。

源集 DSL 可以用于定义两个源集之间的联系：

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

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

除了默认源集外，还应将创建的自定义源集显式地包含在依赖关系层次结构中，以便于能够使用其他源集的定义，并且最重要的是能够参与到编译中。大多数时候，它们需要 `dependsOn(commonMain)` 或 `dependsOn(commonTest)` 声明，并且一些默认的特定平台的源集应该直接或间接地依赖于自定义的源集

```
kotlin {
    mingwX64()
    linuxX64()

    sourceSets {
        // 带有两个目标测试的自定义源集
        desktopTest {
            dependsOn commonTest
            /* …… */
        }
        // 将 “windows” 的默认测试源集设置为依赖于 “desktopTest”
        mingwX64().compilations.test.defaultSourceSet {
            dependsOn desktopTest
            /* …… */
        }
        // 并且为其他目标做同样的工作：
        linuxX64().compilations.test.defaultSourceSet {
            dependsOn desktopTest
            /* …… */
        }
    }
}
```

```
kotlin {
    mingwX64()
    linuxX64()

    sourceSets {
        // 带有两个目标测试的自定义源集
        val desktopTest by creating {
            dependsOn(getByName("commonTest"))
            /* …… */
        }
        // 将 “windows” 的默认测试源集设置为依赖于 “desktopTest”
        mingwX64().compilations["test"].defaultSourceSet {
            dependsOn(desktopTest)
            /* …… */
        }
        // 并且为其他目标做同样的工作：
        linuxX64().compilations["test"].defaultSourceSet {
            dependsOn(desktopTest)
            /* …… */
        }
    }
}
```

添加依赖

为了添加依赖到源集中，需要在源集 DSL 中使用 `dependencies { …… }` 块，支持以下四种依赖：

- `api` 依赖在编译项与运行时均会使用，并导出到库使用者。如果当前模块的公共 API 中使用了依赖中的任何类型，那么它应该是一个 `api` 依赖；
- `implementation` 依赖在当前模块的编译项与运行时均会使用，但不暴露给其他具有 `implementation` 依赖的模块的编译项。对于那种内部逻辑实现所需要的依赖，应该使用 `implementation` 依赖类型。如果模块是一个未发布的 endpoint 应用，它或许该使用 `implementation` 依赖而不是 `api` 依赖。
- `compileOnly` 依赖仅用于当前模块的编译项，并且在运行时与<!— ->其他模块的编译项均不可用。这些依赖应该用于运行时具有第三方实现 API 中。
- `runtimeOnly` 依赖在运行时可用，但在任何模块的编译项都是不可见的。

每个源集都可以通过以下方式指定依赖：

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

请注意，为了 IDE 能够正确地识别公共源的依赖，除了特定平台源集构件的依赖外，公共源集还需要在 Kotlin 元数据包中具有相应的依赖。通常，在使用一个已发布的库时（除非它与 Gradle 元数据一起发布，如下所述），需要有一个后缀为 `-common`（如 `kotlin-stdlib-common`）或 `-metadata` 的构件。

然而，在另一个多平台项目中的 `project('……')` 依赖会被自动处理成一个合适的目标。在源集的依赖中指定单个 `project('……')` 依赖就足够了，并且包含在源集中的编译将会接收到其项目的合适的特定平台的构件，鉴于它具有兼容的目标：

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                // 包含源集 “commonMain” 的所有编译项
                // 会将依赖项解析为兼容的目标（如果有）：
                api project(':foo-lib')
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                // 包含源集 “commonMain” 的所有编译项
                // 会将依赖项解析为兼容的目标（如果有）：
                api(project(":foo-lib"))
            }
        }
    }
}
```

同样的，如果发布了带有 Gradle 元数据的一个多平台库，那么只需要为公共源集指定一次依赖。除此以外，应该为每个特定平台的源集提供库的相应平台模块（除了公共模块），如上所示。

指定依赖的另一种方式是在顶层使用 Gradle 内置 DSL，其配置名称遵循模式 `<源集名称><依赖类型>`：

```
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")
}
```

一些 Gradle 内置依赖（例如 `gradleApi()`、`localGroovy()`、或 `gradleTestKit()`）在源集依赖 DSL 中是不可用的。但是，你可以将它们添加到顶级依赖块中，如上所示。

可以使用 `kotlin("stdlib")` 表示法添加对 Kotlin 模块（例如 `kotlin-stdlib` 或 `kotlin-reflect`）的依赖，这是 `"org.jetbrains.kotlin:kotlin-stdlib"` 的简写。

语言设置

源集的语言设置可以通过以下方式指定：

```
kotlin {
    sourceSets {
        commonMain {
            languageSettings {
                languageVersion = '1.3' // 可填：“1.0”、“1.1”、“1.2”、“1.3”
                apiVersion = '1.3' // 可填：“1.0”、“1.1”、“1.2”、“1.3”
                enableLanguageFeature('InlineClasses') // 语言特性名称
                useExperimentalAnnotation('kotlin.ExperimentalUnsignedTypes') // 注解的全限定名
                progressiveMode = true // 默认为 false
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            languageSettings.apply {
                languageVersion = "1.3" // 可填：“1.0”、“1.1”、“1.2”、“1.3”
                apiVersion = "1.3" // 可填：“1.0”、“1.1”、“1.2”、“1.3”
                enableLanguageFeature("InlineClasses") // 语言特性名称
                useExperimentalAnnotation("kotlin.ExperimentalUnsignedTypes") // 注解的全限定名
                progressiveMode = true // 默认为 false
            }
        }
    }
}
```

可以一次性为所有源集配置语言设置：

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

源集的语言设置会影响 IDE 识别源代码的方式。由于当前的限制，在 Gradle 构建中，只有构建的默认源集的语言设置会被使用，并且应用于参与编译的所有源代码。

检查语言设置是否相互依赖，以确保源集之间的一致性。即如果 `foo` 依赖于 `bar`：

- `foo` 需设置高于或等于 `bar` 的 `languageVersion`；
- `foo` 需要启用所有 `bar` 启用的非稳定语言特性（对于错误修复特性则没有这种要求）；
- `foo` 需要使用所有 `bar` 使用的实验性注解；
- `apiVersion`、错误修复的语言特性 和 `progressiveMode` 可以被任意设置；

默认项目布局

默认情况下，每个项目都包含了两个源集，`commonMain` 与 `commonTest`，在其中可以放置应在所有目标平台之间共享的所有代码。这些源集会被分别添加到每个生产和测试编译项。

之后，当目标被添加时，将为其创建默认编译项：

- 针对 JVM、JS 和原生目标的 `main` 与 `test` 编译项；
- 针对每个 Android 构建版本的编译项；

对于每个编译项，在由 `<目标名称><编译项名称>` 组成的名称下都有一个默认源集。这个默认源集参与编译，因此它应用于特定平台的代码与依赖，并且以依赖的方式将其他源集添加到编译项中。例如，一个有着 `jvm6`（JVM）与 `nodeJs`（JS）目标的项目将拥有源集：`commonMain`、`commonTest`、`jvm6Main`、`jvm6Test`、`nodeJsMain` 以及 `nodeJsTest`。

仅仅是默认源集就涵盖了很多用例，因此不需要自定义源集。

每个源集都默认拥有在 `src/<源集名称>/kotlin` 目录下的 Kotlin 源代码与在 `src/<源集名称>/resources` 目录下的资源。

在 Android 项目中，将为每个 Android 源集创建额外的 Kotlin 源集. 如果其 Android 目标的名称为 `foo`，那么其 Android 源集 `bar` 将获得一个对应的 Kotlin 源集 `fooBar`。然而，Kotlin 编译项能够使用来自所有 `src/bar/java`、`src/bar/kotlin` 以及 `src/fooBar/kotlin` 目录的 Kotlin 源代码。而 Java 源代码则只能从上述第一个目录读取。

运行测试

目前默认支持 JVM、Android、Linux、Windows 以及 macOS 在 Gradle 构建中运行测试；JS 与其他 Kotlin/Native 目标需要手动配置以在适当的环境、模拟器或测试框架下运行测试。

将为每个适合测试的目标创建名为 `<目标名称>Test` 的测试任务。运行 `check` 任务以为所有目标运行测试。

由于 `commonTest` 默认源集被添加到所有测试编译项中，所以会将所有目标平台上所需的测试和测试工具放在此处。

`kotlin.test` API对于多平台测试是可用的。添加 `kotlin-test-common` 与 `kotlin-test-annotations-common` 依赖到 `commonTest` 以在公共测试中使用断言函数（例如 `kotlin.test.assertTrue(……)` 以及 `@Test` / `@Ignore` / `@BeforeTest` / `@AfterTest` 注解）

对于 JVM 目标，将 `kotlin-test-junit` 或 `kotlin-test-testng` 用于相应的断言器实现和注解映射。

对于 Kotlin/JS 目标，把 `kotlin-test-js` 添加为测试依赖。至此，将创建针对 Kotlin/JS 的测试任务，但默认情况下并不会运行测试；应该手动配置它们以使用 JavaScript 测试框架运行测试。

Kotlin/Native 目标不需要额外测试依赖，并且内置了 `kotlin.test` API 的实现。

发布多平台库

> 目标平台集合由多平台库作者定义，并且他们应该为库提供所有特定平台的实现。不支持为多平台库添加用户端的新目标。

来自多平台项目的库构建可以通过 `maven-publish` Gradle 插件发布到 Maven 仓库，这个插件可通过以下方式应用：

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

一个库也需要在项目中设置 `group` 与 `version` 字段：

```
plugins { /* …… */ }

group = "com.example.my.library"
version = "0.0.1"
```

与发布一个普通的 Kotlin/JVM 或 Java 项目相比，并没有必要通过 `publishing { …… }` DSL 来手动创建一个发布项。将为可在当前主机构建的每个目标自动创建发布项，但 Android 目标除外，Android 目标需要额外的步骤来配置发布，参见发布 Android 库。

通过在 `publishing { …… }` DSL 中的 `repositories` 块添加将被发布的库的仓库。如 Maven Publish Plugin. Repositories 所述。

默认构件 ID 遵循模式 `<项目名称>-<小写的目标名称>`，例如对于项目 `sample-lib` 中名为 `nodeJs` 的目标，为 `sample-lib-nodejs`。

默认情况下，将为每个发布项中添加源代码 JAR（除了它的主构件）。源代码 JAR 包含了目标主编译项所使用的源代码。如果你还需要发布文档构件（例如 Javadoc JAR），则需要手动配置其构建并且将其作为构件添加到相关发布项中，如下所示。

此外，会默认添加名为 `metadata` 的额外发布项，它包含序列化的 Kotlin 定义并且被 IDE 用于分析多平台库。这个发布项的默认构件 ID 的形式为 `<项目名称>-metadata`。

可以更改 Maven 坐标，并且可以为在 `targets { …… }` 块或 `publishing { …… }` DSL 中的发布项添加额外的构件文件：

```
kotlin {
    jvm('jvm6') {
        mavenPublication { // 为目标 “jvm6” 设置发布项
            // 默认的 artifactId 为 “foo-jvm6”，修改它：
            artifactId = 'foo-jvm'
            // 添加 docs JAR 构件（应是一个自定义任务）：
            artifact(jvmDocsJar)
        }
    }
}

// 使用 `publishing { …… }` DSL 来配置发布项是可选的：
publishing {
    publications {
        jvm6 { /* 为目标 “jvm6” 设置发布项 */ }
        metadata { /* 为 Kotlin 元数据设置发布项 */ }
    }
}
```

```
kotlin {
    jvm("jvm6") {
        mavenPublication { // 为目标 “jvm6” 设置发布项
            // 默认的 artifactId 为 “foo-jvm6”，修改它：
            artifactId = "foo-jvm"
            // 添加 docs JAR 构件（应是一个自定义任务）：
            artifact(jvmDocsJar)
        }
    }
}

// 使用 `publishing { …… }` DSL 来配置发布项是可选的：
publishing {
    publications.withType<MavenPublication>().apply {
        val jvm6 by getting { /* 为目标 “jvm6” 设置发布项 */ }
        val metadata by getting { /* 为 Kotlin 元数据设置发布项 */ }
    }
}
```

由于 Kotlin/Native 的汇编构件需要多次构建才能在不同的主机平台运行，所以发布包含 Kotlin/Native 目标的多平台库需要使用同一套主机进行。为了避免重复发布能在多个平台（例如 JVM、JS、Kotlin 元数据以及 WebAssembly）上构建的模块，可以将这些模块的发布任务配置为有条件地运行。

这个简化的例子确保了 JVM、JS 与 Kotlin 元数据的发布仅在命令行中传递 `-PisLinux=true` 到构建时上传：

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

    // 注意 Kotlin 元数据也在这里。
    // 由于 mingwx64() 目标在 Linux 构建中不兼容而被自动跳过。
    configure([targets["metadata"], jvm(), js()]) {
        mavenPublication { targetPublication ->
            tasks.withType(AbstractPublishToMaven)
                .matching { it.publication == targetPublication }
                .all { onlyIf { findProperty("isLinux") == "true" } }
        }
    }
}
```

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

    // 注意 Kotlin 元数据也在这里。
    // 由于 mingwx64() 目标在 Linux 构建中不兼容而被自动跳过。
    configure(listOf(metadata(), jvm(), js())) {
        mavenPublication {
            val targetPublication = this@mavenPublication
            tasks.withType<AbstractPublishToMaven>()
                .matching { it.publication == targetPublication }
                .all { onlyIf { findProperty("isLinux") == "true" } }
        }
    }
}
```

元数据发布

Gradle 模块元数据提供了丰富的发布与解析依赖项的特性，这些特性用于 Kotlin 多平台项目来为构建作者简化依赖配置。特别是多平台库的发布项包含一个特殊的“根”模块，它基于整个库，并且在添加为依赖项时自动解析到适当的特定平台构件中，如下所述。

In Gradle 6.0 and above, the module metadata is always used during dependency resolution and included in publications.

In earlier Gradle versions starting from 5.3，依赖项解析期间使用模块元数据，但在默认情况下，发布项不会包含任何模块元数据。为了启用发布模块元数据，需要添加 `enableFeaturePreview("GRADLE_METADATA")` 到根项目的 `settings.gradle` 文件。

When publications include module metadata，一个额外的名为 `kotlinMultiplatform` 的“根”发布项将添加到项目的发布项中。这个发布项的默认构件 ID 与没有任何额外后缀的项目名称相匹配。为了配置这个发布项，可以通过 `maven-publish` 插件的 `publishing { …… }` DSL 访问：

```
kotlin { /* …… */ }

publishing {
    publications {
        kotlinMultiplatform {
            artifactId = "foo"
        }
    }
}
```

```
kotlin { /* …… */ }

publishing {
    publications {
        val kotlinMultiplatform by getting {
            artifactId = "foo"
        }
    }
}
```

这个发布项没有包含任何构件并且仅将其他发布项引用为它的变体。然而，如果仓库需要，则可能需要提供源代码和文档构件。在这种情况下，在发布项的 scope 中通过使用 `artifact(...)` 添加那些构件，如上所示访问。

如果库拥有一个“根”发布项，用户可以在公共源集中指定对整个库的单个依赖，并且将为每个包含这个依赖项的编译项（如果有）选择一个合适的特定平台版本。考虑一个为 JVM 与 JS 编译并且与“根”发布项一起发布的 `sample-lib` 库：

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

    sourceSets {
        commonMain {
            dependencies {
                // 这单个依赖将解析到适当的目标模块，
                // 例如，对于 JVM 解析为 `sample-lib-jvm6`，而对于 JS 解析为 `sample-lib-js`：
                api 'com.example:sample-lib:1.0'
            }
        }
    }
}
```

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

    sourceSets {
        val commonMain by getting {
            dependencies {
                // 这单个依赖将解析到适当的目标模块，
                // 例如，对于 JVM 解析为 `sample-lib-jvm6`，而对于 JS 解析为 `sample-lib-js`：
                api("com.example:sample-lib:1.0")
            }
        }
    }
}
```

目标消歧义

在一个多平台库中，对于单个平台可能拥有多个目标。例如，这些目标可能提供了相同的 API，并且在运行时调用的实现库中有所不同，例如测试框架或日志解决方案。

然而，对这种多平台库的依赖可能存在歧义，并且可能因为没有充足的信息来决定选择哪个目标，从而导致解析的失败。

解决的方法是用自定义属性标记目标，Gradle 会根据它来解析依赖项。但是，库的作者与用户必须同时给目标加上自定义属性，并且库作者有责任将属性与可能的值传达给使用者。

添加属性对库作者和用户来说是对称的。例如，考虑一个在两个目标中分别支持了 JUnit 和 TestNG 的测试库。库作者需要为这两个目标添加属性，如下：

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

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

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

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

用户可能只需要给产生歧义的单个目标添加属性。

如果将依赖项被添加到自定义的配置项中（而不是通过插件创建的配置项之一）时出现了相同的歧义，你可以通过相同的方式将属性添加到配置项中：

```
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")
    }
}
```

JVM 目标平台中的 Java 支持

这个特性自 Kotlin 1.3.40 可用。

默认情况下，JVM 目标将忽略 Java 源代码，并且只编译 Kotlin 源文件。为了将 Java 源代码包含入 JVM 目标的编译项中，或是为了应用需要 `java` 插件才能工作的 Gradle 插件，你需要为目标显式地启用 Java 支持：

```
kotlin {
    jvm {
        withJava()
    }
}
```

这将会应用 Gradle `java` 插件，并配置目标以与它协作。注意，在 JVM 目标中仅应用 Java 插件但没有指定 `withJava()`，将不会对目标有任何影响。

Java 源代码的文件系统位置与 `java` 插件的默认值不同。Java 源文件需要被放置在 Kotlin 源代码根目录的同级目录中。例如，如果 JVM 目标有一个默认名称 `jvm`，则路径为：

```
src
├── jvmMain
│   ├── java // production Java sources
│   ├── kotlin
│   └── resources
├── jvmTest
│   ├── java // test Java sources
│   ├── kotlin
…   └── resources
```

公共源集不能包含 Java 源代码。

由于当前的限制，一些由 Java 插件配置的任务将被禁用，并且 Kotlin 插件添加了相应的任务来代替它们：

- `jar` 被禁用，取而代之的是目标的 JAR 任务（例如 `jvmJar`）
- `test` 被禁用，并且使用目标的测试任务（例如 `jvmTest`）
- `*ProcessResources` 任务被禁用，并且资源将由编译项的等价任务处理

这个目标的发布项将由 Kotlin 插件处理，并且不需要特定于 Java 插件的步骤，例如手动创建发布项并配置它为 `from(components.java)`。

Android 支持

Kotlin 多平台项目通过提供 `android` 内置函数支持 Android 平台。创建 Android 目标需要 Android Gradle 插件之一，例如手动应用 `com.android.application` 或 `com.android.library` 到项目中。每个 Gradle 子项目仅可能创建一个 Android 目标：

```
plugins {
    id("com.android.library")
    id("org.jetbrains.kotlin.multiplatform").version("1.3.72")
}

android { /* …… */ }

kotlin {
    android { // 创建 Android 目标
        // 提供必要的附加配置
    }
}
```

```
plugins {
    id("com.android.library")
    kotlin("multiplatform").version("1.3.72")
}

android { /* …… */ }

kotlin {
    android { // 创建 Android 目标
        // 提供必要的附加配置
    }
}
```

默认创建的 Android 目标编译项与 Android 构建变体相关联：对于每个构建变体，将会以相同的名称创建 Kotlin 构建项。

然后，对于每个通过变体编译的 Android 源集，将在目标名称前面的那个源集名称下创建 Kotlin 源集，例如 Kotlin 源集 `androidDebug` 用于 Android 源集 `debug` 与名为 `android` 的 Kotlin 目标。这些 Kotlin 源集将相应地添加到变体编译项中。

默认源集 `commonMain` 将添加到每个生产项（应用或库）变体的编译项中。类似地，`commonTest` 源集也将添加到单元测试的编译项，以及 instrumented 测试变体中。

使用 kapt 进行注解处理也是受支持的，但，由于当前的限制，它要求 Android 目标需要在配置 `kapt` 依赖之前创建，`kapt` 依赖需要在顶级 `dependencies { …… }` 代码块（而不是 Kotlin 源集依赖）中完成。

```
// ...

kotlin {
    android { /* …… */ }
}

dependencies {
    kapt("com.my.annotation:processor:1.0.0")
}
```

发布 Android 库

为了将 Android 库发布为多平台库的一部分，需要为库设定发布项，并且为 Android 库目标提供额外的配置项。

默认情况下，不会发布 Android 库的构件。为了发布 Android 变体生成的一系列构件，需要在 Android 目标代码块中指定变体名称，如下所示：

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

上面的例子将在没有产品类型的 Android 库上工作。对于有产品类型的库，变体名称也要包含产品类型，例如 `fooBarDebug` 或是 `fooBazRelease`。

注意，如果库用户定义了库中缺失的变体，则他们需要提供备用的匹配。例如，如果库不具有，或者没有发布 `staging` 构建类型，那么有必要为拥有这种构建类型的使用者提供备用的匹配，至少指定库发布项的一个构建类型：

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

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

类似地，如果库发布项中缺失某些备用的匹配，那么库用户也许需要为自定义产品类型提供它们。

你可以选择发布按产品类型分组的变体，以便将不同构建类型的输出放置在单独的模块中，并使构建类型成为构建的分类器（release 构建类型不通过分类器发布）。这个模式默认是禁用的，不过可以通过以下方式启用：

```
kotlin {
    android {
        publishLibraryVariantsGroupedByFlavor = true
    }
}
```

不推荐发布按产品类型分组的变体，以免它们拥有不同的依赖项，因为这些将被合并到一个依赖项列表中。

使用 Kotlin/Native 目标平台

重要的是，注意某些 Kotlin/Native 目标仅能在适当的主机上被编译：

- Linux MIPS 目标（`linuxMips32` 与 `linuxMipsel32`）需要一台 Linux 主机。其他 Linux 目标则可以在任意受支持的主机上编译。
- Windows 目标需要一台 Windows 主机；
- macOS 与 iOS 目标只能在 macOS 主机上编译；
- 64 位的 Android 原生目标需要一台 Linux 或 macOS 主机。32 位的 Android 原生目标则可以在任意受支持的主机上编译。

当前主机不支持的目标在构建期间会被忽略，因此也不会发布。库作者可能希望在目标库平台所需的不同主机上进行构建和发布。

目标快捷方式

一些原生目标经常一同创建，并且使用相同的源代码。例如，iOS 设备与模拟器的构建由不同的目标（分别是 `iosArm64` 与 `iosX64`）表示，但它们的源代码通常是相同的。多平台项目模型中来表示这种共享代码的一个经典方式是创建一个中间源集（`iosMain`），并且在它和平台源集之间配置链接：

```
sourceSets{
    iosMain {
        dependsOn(commonMain)
        iosDeviceMain.dependsOn(it)
        iosSimulatorMain.dependsOn(it)
    }
}
```

```
val commonMain by sourceSets.getting
val iosDeviceMain by sourceSets.getting
val iosSimulatorMain by sourceSets.getting

val iosMain by sourceSets.creating {
    dependsOn(commonMain)
    iosDeviceMain.dependsOn(this)
    iosSimulatorMain.dependsOn(this)
}
```

自 1.3.60 起，`kotlin-multiplaform` 插件提供了自动化这些配置的快捷方式：它们使用户可以通过单个 DSL 方法来创建一组目标以及公共源集。

可用快捷方式有这些：

— `ios` 为 `iosArm64` 与 `iosX64` 创建目标。

— `watchos` 为 `watchosArm32`、`watchosArm64` 以及 `watchosX86` 创建目标。

— `tvos` 为 `tvosArm64` 与 `tvosX64` 创建目标。

```
// 为 iOS 创建两个目标。
// 创建公共源集：iosMain 与 iosTest。
ios {
    // 配置目标。
    // 注意：将会为每个目标调用这个 lambda。
}

// 你也可以指定一个名称前缀来创建目标。
// 公共源集也将会有一个前缀：
// anotherIosMain 与 anotherIosTest。
ios("anotherIos")
```

构建最终原生二进制文件

默认情况下，Kotlin/Native 目标将被编译为 `*.klib` 库构件，它可以被 Kotlin/Native 自身作为依赖项使用，但并不能被执行，或是用作原生库。为了声明像可执行文件或是链接库的最终原生二进制文件，需要使用原生目标的 `binaries` 属性。除默认 `*.klib` 构建外，这个属性还代表一个为这个目标构建的原生二进制文件集合，并且提供了一系列声明和配置它们的方法。

注意，`kotlin-multiplaform` 插件默认不会创建任何生产二进制文件。默认情况下，唯一可用的二进制文件是调试可执行文件，它允许运行来自 `test` 编译项的测试。

声明二进制文件

`binaries` 集合的元素通过一套工厂方法声明。这些方法允许指定要创建的二进制类型并对其进行配置。以下是受支持的二进制类型（注意，并不是所有类型都可用于所有原生平台）：

工厂方法	二进制类型	可用于
`executable`	产品可执行文件	所有原生目标
`test`	测试可执行文件	所有原生目标
`sharedLib`	链接原生库	除了 `wasm32` 以外的所有原生目标
`staticLib`	静态原生库	除了 `wasm32` 以外的所有原生目标
`framework`	Objective-C framework	仅 macOS、iOS、watchOS 与 tvOS 目标

每个工厂方法都有多个版本。通过 `executable` 方法的示例考虑他们。所有相同的版本对所有其他的工厂方法都是可用的。

最简单的版本不需要任何额外的参数，并且会为每个构建类型都创建二进制文件。目前有两个可用的构建类型：`DEBUG`（生成带有调试信息的、未优化的二进制文件）与 `RELEASE`（生成不带有调试信息的、经过优化的二进制文件）。

```
kotlin {
    linuxX64 { // 更改为你所使用的目标。
        binaries {
            executable {
                // 二进制配置。
            }
        }
    }
}
```

在上面例子中的 `executable` 方法接受的 lambda 表达式将应用于创建的每个二进制文件，并且允许配置二进制文件。（参见相应部分）。注意，如果不需要额外的配置，则可以删除这个 lambda：

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

可以指定哪些构建类型将用于创建二进制文件，哪些将不创建。以下的示例仅创建了调试可执行文件。

```
binaries {
    executable([DEBUG]) {
        // 二进制配置。
    }
}
```

```
binaries {
    executable(listOf(DEBUG)) {
        // 二进制配置。
    }
}
```

最终，最后一个工厂方法版本允许自定义二进制文件名称。

```
binaries {
    executable('foo', [DEBUG]) {
        // 二进制配置。
    }

    // 可以删除构建类型的列表（这种情况下，将使用所有可用的构建类型）。
    executable('bar') {
        // 二进制配置。
    }
}
```

```
binaries {
    executable("foo", listOf(DEBUG)) {
        // 二进制配置。
    }

    // 可以删除构建类型的列表（这种情况下，将使用所有可用的构建类型）。
    executable("bar") {
        // 二进制配置。
    }
}
```

在这个示例中的第一个参数允许为创建的二进制文件设置名称前缀，该前缀用于在构建脚本中访问它们（参见 “访问二进制文件” 部分）。这个前缀也用作二进制文件的默认名称。例如在 Windows 平台上，上面的示例将生产出 `foo.exe` 与 `bar.exe` 文件。

访问二进制文件

binaries DSL 不仅允许创建二进制文件，还可以访问已经创建的二进制文件以配置它们或获取它们的属性（例如输出文件的路径）。`binaries` 集合实现了 `DomainObjectSet` 接口，并提供了类似 `all` 或 `matching` 这些允许配置元素组的方法。

也可以获取集合中的某些元素。有两种方式可以做到。第一种方式，每个库拥有独有的名字。这个名称基于名称的前缀（如果已指定），构建类型和二进制种类根据以下模式：`<可选名称前缀><构建类型><二进制种类>`，例如 `releaseFramework` 或 `testDebugExecutable`。

> 注意：静态库和共享库分别有 `static` 与 `shared` 后缀，例如 `fooDebugStatic` 或 `barReleaseShared`

这个名称可以用于访问库：

```
// 如果没有这样的库将会导致错误。
binaries['fooDebugExecutable']
binaries.fooDebugExecutable
binaries.getByName('fooDebugExecutable')

 // 如果没有这样的库将返回 null。
binaries.findByName('fooDebugExecutable')
```

```
// 如果没有这样的库将会导致错误。
binaries["fooDebugExecutable"]
binaries.getByName("fooDebugExecutable")

 // 如果没有这样的库将返回 null。
binaries.findByName("fooDebugExecutable")
```

第二种方式是使用标记过类型的 getter。这些 getter 允许通过它的名称前缀与构建类型访问某种类型的二进制文件。

```
// 如果没有这样的库将会导致错误。
binaries.getExecutable('foo', DEBUG)
binaries.getExecutable(DEBUG)          // 如果没有设置名称前缀则会跳过第一个参数。
binaries.getExecutable('bar', 'DEBUG') // 你也可以使用字符串作为构建类型。

// 类似的 getter 对其他二进制种类也是可以用的：
// getFramework、getStaticLib 与 getSharedLib。

// 如果没有这样的库将返回 null。
binaries.findExecutable('foo', DEBUG)

// 类似的 getter 对其他二进制种类也是可以用的：
// findFramework、findStaticLib 与 findSharedLib。
```

```
// 如果没有这样的库将会导致错误。
binaries.getExecutable("foo", DEBUG)
binaries.getExecutable(DEBUG)          // 如果没有设置名称前缀则会跳过第一个参数。
binaries.getExecutable("bar", "DEBUG") // 你也可以使用字符串作为构建类型。

// 类似的 getter 对其他二进制种类也是可以用的：
// getFramework、getStaticLib 与 getSharedLib。

// 如果没有这样的库将返回 null。
binaries.findExecutable("foo", DEBUG)

// 类似的 getter 对其他二进制种类也是可以用的：
// findFramework、findStaticLib 与 findSharedLib。
```

> 1.3.40 之前，测试和产品可执行文件均由相同的二进制类型表示。因此，要访问通过插件创建的默认测试二进制文件，请使用下行：
>
> ```
> binaries.getExecutable("test", "DEBUG")
> ```
>
> 自 1.3.40 起，测试可执行文件由单独的二进制类型表示，并且拥有自己的 getter。要访问默认的测试库，请使用：
>
> ```
> binaries.getTest("DEBUG")
> ```

配置二进制文件

二进制文件具有一套属性，允许配置它们。可用的选项有这些：

- **编译项。**每个二进制都是基于相同目标中一些可用的编译项构建的。这个参数的默认值依赖于二进制类型：`Test` 二进制文件基于 `test` 编译项，而其他二进制文件基于 `main` 编译项。
- **链接器选项。**选项将在二进制文件的构建期间被传递到系统链接器中。可以使用这个设置链接到某些原生库。
- **输出文件名称。**默认情况下，输出文件名称基于二进制文件名称前缀，如果没有指定前缀，则基于项目名称。但可以使用 `baseName` 属性来单独配置输出文件名称。注意，最终文件名称将通过添加系统相关的前缀与后缀到这个基础名称形成。例如 Linux 共享库与基础名称 `foo` 将产出 `libfoo.so`。
- **入口点**（仅用于可执行的二进制文件）。默认情况下，Kotlin/Native 程序的入口点是位于根包的 `main` 函数。这个设置允许改变这个默认值，并使用自定义的函数作为入口点。例如它可以用于将 `main` 函数从根包中移出。
- **访问输出文件。**
- **访问链接任务。**
- **访问运行任务**（仅用于可执行的二进制文件）。`kotlin-multiplatform` 插件为主机平台（Windows、Linux 与 macOS）的所有可执行二进制文件创建运行任务。这些任务的名称基于二进制文件名称，例如

`runReleaseExecutable<目标名称>` 或 `runFooDebugExecutable<目标名称>`。可以使用可执行二进制文件的 `runTask` 属性来访问运行任务。

— **Framework 类型**（仅用于 Objective-C frameworks）。默认情况下，通过 Kotlin/Native 构建的 framework 包含动态库。但可以把它替换为静态库。

下面的例子演示了如何使用这些设置。

```
binaries {
    executable('my_executable', [RELEASE]) {
        // 在测试编译项的基础上构建二进制文件。
        compilation = compilations.test

        // 为链接器自定义命令行选项。
        linkerOpts = ['-L/lib/search/path', '-L/another/search/path', '-lmylib']

        // 用于输出文件的基础名称。
        baseName = 'foo'

        // 自定义入口函数。
        entryPoint = 'org.example.main'

        // 访问输出文件。
        println("Executable path: ${outputFile.absolutePath}")

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

        // 访问运行任务。
        // 注意，对于非本机的平台，runTask 为 null。
        runTask?.dependsOn(prepareForRun)
    }

    framework('my_framework' [RELEASE]) {
        // 在框架中包含静态库，而不是动态库。
        isStatic = true
    }
}
```

```
binaries {
    executable("my_executable", listOf(RELEASE)) {
        // 在测试编译项的基础上构建二进制文件。
        compilation = compilations["test"]

        // 为链接器自定义命令行选项。
        linkerOpts = mutableListOf("-L/lib/search/path", "-L/another/search/path", "-lmylib")

        // 用于输出文件的基础名称。
        baseName = "foo"

        // 自定义入口函数。
        entryPoint = "org.example.main"

        // 访问输出文件。
        println("Executable path: ${outputFile.absolutePath}")

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

        // 访问运行任务。
        // 注意，对于非本机的平台，runTask 为 null。
        runTask?.dependsOn(prepareForRun)
    }

    framework("my_framework" listOf(RELEASE)) {
        // 在框架中包含静态库，而不是动态库。
        isStatic = true
    }
}
```

导出依赖项到二进制文件

当构建 Objective-C framework 或原生库（共享或静态）时，经常不仅要打包当前项目的 class，还需要打包其某些依赖项的 class。binaries DSL 允许使用 `export` 方法指定将哪些依赖项将导出到二进制文件。注意，仅有相应源集的 API 依赖项可以被导出。

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // 将被导出。
            api project(':dependency')
            api 'org.example:exported-library:1.0'

            // 将不被导出。
            api 'org.example:not-exported-library:1.0'
        }
    }

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

        sharedLib {
            // 可以将不同的依赖项集合导出到不同的二进制文件。
            export project(':dependency')
        }
    }
}
```

```
kotlin {
    sourceSets {
        macosMain.dependencies {
            // 将被导出。
            api(project(":dependency"))
            api("org.example:exported-library:1.0")

            // 将不被导出。
            api("org.example:not-exported-library:1.0")
        }
    }

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

        sharedLib {
            // 可以将不同的依赖项集合导出到不同的二进制文件。
            export(project(':dependency'))
        }
    }
}
```

如这个示例所展示的，maven 依赖项也可以被导出。但由于 Gradle 元数据的当前限制，这种依赖项应该是平台依赖（例如 `kotlinx-coroutines-core-native_debug_macos_x64` 而不是 `kotlinx-coroutines-core-native`）或被传递地导出（参见下文）。

默认情况下，导出工作是非传递性的。如果导出了依赖于库 `bar` 的库 `foo`，那么仅有 `foo` 的方法将被添加到输出 framework。这个行为可以通过 `transitiveExport` 标志来改变。

```
binaries {
    framework {
        export project(':dependency')
        // 过渡地导出。
        transitiveExport = true
    }
}
```

```
binaries {
    framework {
        export(project(":dependency"))
        // 过渡地导出。
        transitiveExport = true
    }
}
```

构建通用 framework

默认情况下，仅支持一个平台通过 Kotlin/Native 产出 Objective-C framework。然而，这种 framework 可以使用 `lipo` 工具将其合并到一个单独的、通用的（fat）二进制文件中。特别的，这种操作对于 32 位与 64 位的 iOS framework 是有意义的。在这种情况下，最终通用 framework 可以在 32 位与 64 位的设备上使用。

Gradle 插件提供了一个单独的任务，该任务从多个常规目标为 iOS 目标创建通用 framework。下面的示例展示了如何使用这个任务。注意，fat framework 必须具有与初始 framework 相同的基础名称。

```
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // 创建并配置目标。
    targets {
        iosArm32("ios32")
        iosArm64("ios64")

        configure([ios32, ios64]) {
            binaries.framework {
                baseName = "my_framework"
            }
        }
    }

    // 创建一个任务，用于构建 fat framework。
    task debugFatFramework(type: FatFrameworkTask) {
        // fat framework 必须具有与初始 framework 相同的基础名称。
        baseName = "my_framework"

        // 默认目标目录是 “<build 目录>/fat-framework”。
        destinationDir = file("$buildDir/fat-framework/debug")

        // 指定要合并的 framework。
        from(
            targets.ios32.binaries.getFramework("DEBUG"),
            targets.ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

```
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // 创建并配置目标。
    val ios32 = iosArm32("ios32")
    val ios64 = iosArm64("ios64")

    configure(listOf(ios32, ios64)) {
        binaries.framework {
            baseName = "my_framework"
        }
    }

    // 创建一个任务，用于构建 fat framework。
    tasks.create("debugFatFramework", FatFrameworkTask::class) {
        // fat framework 必须具有与初始 framework 相同的基础名称。
        baseName = "my_framework"

        // 默认目标目录是 “<build 目录>/fat-framework”。
        destinationDir = buildDir.resolve("fat-framework/debug")

        // 指定要合并的 framework。
        from(
            ios32.binaries.getFramework("DEBUG"),
            ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

C 互操作支持

自 Kotlin/Native 提供了与原生语言互操作，就有 DSL 允许为特定编译项配置这个特性。

编译项可以与多个原生库交互。它们的互操作性可以在 compilation 的 `cinterops` 块中配置：

```
kotlin {
    linuxX64 { // 替换为你所需要的目标
        compilations.main {
            cinterops {
                myInterop {
                    // Def-file 描述原生 API。
                    // 默认路径是 src/nativeInterop/cinterop/<互操作名称>.def
                    defFile project.file("def-file.def")

                    // 用于放置生成的 Kotlin API 的包。
                    packageName 'org.sample'

                    // 通过 cinterop 工具传递给编译器的选项
                    compilerOpts '-Ipath/to/headers'

                    // 用于头文件搜索的目录（类似于编译器选项 -I<路径>）。
                    includeDirs.allHeaders("path1", "path2")

                    // 搜索在 “headerFilter” def-file 选项中列出的头文件的额外目录。
                    // 类似于命令行参数 -headerFilterAdditionalSearchPrefix。
                    includeDirs.headerFilterOnly("path1", "path2")

                    // includeDirs.allHeaders 的快捷方式。
                    includeDirs("include/directory", "another/directory")
                }

                anotherInterop { /* …… */ }
            }
        }
    }
}
```

```
kotlin {
    linuxX64 {  // 替换为你所需要的目标
        compilations.getByName("main") {
            val myInterop by cinterops.creating {
                // Def-file 描述原生 API。
                // 默认路径是 src/nativeInterop/cinterop/<互操作名称>.def
                defFile(project.file("def-file.def"))

                // 用于放置生成的 Kotlin API 的包。
                packageName("org.sample")

                // 通过 cinterop 工具传递给编译器的选项
                compilerOpts("-Ipath/to/headers")

                // 用于寻找头文件的目录。
                includeDirs.apply {
                    // 用于头文件搜索的目录（类似于编译器选项 -I<路径>）。
                    allHeaders("path1", "path2")

                    // 搜索在 “headerFilter” def-file 选项中列出的头文件的额外目录。
                    // 类似于命令行参数 -headerFilterAdditionalSearchPrefix。
                    headerFilterOnly("path1", "path2")
                }
                // includeDirs.allHeaders 的快捷方式。
                includeDirs("include/directory", "another/directory")
            }

            val anotherInterop by cinterops.creating { /* …… */ }
        }
    }
}
```

经常需要为使用了原生库的二进制文件指定特定于目标的链接器选项。可以通过使用二进制文件的 `linkerOpts` 属性来完成。参见配置二进制文件部分获取更多详细内容。

Kotlin 多平台 Gradle DSL 参考

> 在 Kotlin 1.2 与 1.3 中，多平台项目是实验性特性。所有在这篇文档中描述的所有语言与工具特性都可能在未来的 Kotlin 版本中改变。

Kotlin 多平台 Gradle 插件是用于创建 Kotlin 多平台 项目的工具。这里我们提供了它的参考；在为 Kotlin 多平台项目编写 Gradle 构建脚本时，用它作提醒。关于 Kotlin 多平台项目的概念与使用插件编写构建脚本的说明，参见使用 Gradle 构建多平台项目。

目录

id 与版本

Kotlin 多平台 Gradle 插件的全限定名是 `org.jetbrains.kotlin.multiplatform`。如果你使用 Kotlin Gradle DSL，那么你可以通过 `kotlin("multiplatform")` 来应用插件。插件版本与 Kotlin 发行版本相匹配。最新的版本是：1.3.72。

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.72'
}
```

```
plugins {
    kotlin("multiplatform") version "1.3.72"
}
```

顶层块

`kotlin` 是在 Gradle 构建脚本中用于配置多平台项目的顶层块。`kotlin` 块内，你可以使用以下块：

块	介绍
<目标名称>	声明项目的特定目标，所有可用的目标名称已陈列在目标部分中.
`targets`	项目的所有目标。
`presets`	所有预定义的目标。使用这个同时配置多个预定义目标。
`sourceSets`	配置预定义和声明自定义项目的源集。

目标

目标 是构建的一部分，负责构建编译、测试、以及针对某个已支持平台打包软件。多平台项目的目标在 `kotlin` 块中的相应代码块中描述，例如：`jvm`、`android` 以及 `iosArm64`。以下是可用目标的完整列表：

名称	描述
`jvm`	Java 虚拟机
`js`	JavaScript
`android`	Android (APK)
`androidNativeArm32`	Android NDK 基于 ARM (ARM32) 平台
`androidNativeArm64`	Android NDK 基于 ARM64 平台
`androidNativeX86`	Android NDK 基于 x86 平台
`androidNativeX64`	Android NDK 基于 x86_64 平台
`iosArm32`	Apple iOS 基于 ARM (ARM32) 平台 (Apple iPhone 5 或更早)
`iosArm64`	Apple iOS 基于 ARM64 平台 (Apple iPhone 5s 或更新)
`iosX64`	Apple iOS 64-bit 模拟器
`watchosArm32`	Apple watchOS 基于 ARM (ARM32) 平台 (Apple Watch Series 3 或更早)
`watchosArm64`	Apple watchOS 基于 ARM64_32 平台 (Apple Watch Series 4 或更新)
`watchosX86`	Apple watchOS 模拟器
`tvosArm64`	Apple tvOS 基于 ARM64 平台 (Apple TV 4th generation 或更新)
`tvosX64`	Apple tvOS 模拟器
`linuxArm64`	Linux 基于 ARM64 平台，例如：树莓派
`linuxArm32Hfp`	Linux 基于 hard-float ARM (ARM32) 平台
`linuxMips32`	Linux 基于 MIPS 平台
`linuxMipsel32`	Linux 基于 little-endian MIPS (mipsel) 平台
`linuxX64`	Linux 基于 x86_64 平台
`macosX64`	Apple macOS
`mingwX64`	64-bit 微软 Windows
`mingwX86`	32-bit 微软 Windows
`wasm32`	WebAssembly

```
kotlin {
    jvm()
    iosX64()
    macosX64()
    js().browser()
}
```

目标的配置项可以包含这两个部分：

— 可用于所有目标的公共目标配置。

— 目标特定的配置项。

公共目标配置

In any target block, you can use the following declarations:

Name	Description
`attributes`	Attributes used for disambiguating targets for a single platform.
`preset`	The preset that the target has been created from, if any.
`platformType`	Designates the Kotlin platform of this target. Avaiable values: `jvm`, `androidJvm`, `js`, `native`, `common`.
`artifactsTaskName`	The name of the task that builds the resulting artifacts of this target.
`components`	The components used to setup Gradle publications.

JVM 目标

In addition to common target configuration, jvm targets have a specific function:

Name	Description
`withJava()`	Includes Java sources into the JVM target' s compilations.

Use this function for projects that contain both Java and Kotlin source files. Note that the default source directories for Java sources don't follow the Java plugin's defaults. Instead, they are derived from the Kotlin source sets. For example, if the JVM target has the default name `jvm`, the paths are `src/jvmMain/java` (for production Java sources) and `src/jvmTest/java` for test Java sources. For more information, see Java support in JVM targets.

```
kotlin {
    jvm {
        withJava()
    }
}
```

JavaScript 目标

The `js` block describes the configuration of JavaScript targets. It can contain one of two blocks depending on the target execution environment:

Name	Description
`browser`	Configuration of the browser target.
`nodejs`	Configuration of the Node.js target.

For details about configuring Kotlin/JS projects, see Setting up a Kotlin/JS project.

Browser

`browser` can contain the following configuration blocks:

Name	Description
`testRuns`	Configuration of test execution.
`runTask`	Configuration of project running.
`webpackTask`	Configuration of project bundling with Webpack.
`dceTask`	Configuration of Dead Code Elimination.
`distribution`	Path to output files.

```
kotlin {
    js().browser {
        webpackTask { /* ... */ }
        testRuns { /* ... */ }
        dceTask {
            keep("myKotlinJsApplication.org.example.keepFromDce")
        }
        distribution {
            directory = File("$projectDir/customdir/")
        }
    }
}
```

Node.js

`nodejs` can contain configurations of test and run tasks:

Name	Description
`testRuns`	Configuration of test execution.
`runTask`	Configuration of project running.

```
kotlin {
    js().nodejs {
        runTask { /* ... */ }
        testRuns { /* ... */ }
    }
}
```

Native 目标

For native targets, the following specific blocks are available:

Name	Description
`binaries`	Configuration of binaries to produce.
`cinterops`	Configuration of interop with C libraries.

Binaries

There are the following kinds of binaries:

Name	Description
`executable`	Product executable.
`test`	Test executable.
`sharedLib`	Shared library.
`staticLib`	Static library.
`framework`	Objective-C framework.

```
kotlin {
    linuxX64 { // Use your target instead.
        binaries {
            executable {
                // Binary configuration.
            }
        }
    }
}
```

For binaries configuration, the following parameters are available:

Name	Description
`compilation`	The compilation from which the binary is built. By default, `test` binaries are based on the `test` compilation while other binaries - on the `main` compilation.
`linkerOpts`	Options passed to a system linker during binary building.
`baseName`	Custom base name for the output file. The final file name will be formed by adding system-dependent prefix and postfix to this base name.
`entryPoint`	The entry point function for executable binaries. By default, it's `main()` in the root package.
`outputFile`	Access to the output file.
`linkTask`	Access to the link task.
`runTask`	Access to the run task for executable binaries. For targets other than `linuxX64`, `macosX64`, or `mingwX64` the value is `null`.
`isStatic`	For Objective-C frameworks. Includes a static library instead of a dynamic one.

```
binaries {
    executable('my_executable', [RELEASE]) {
        // Build a binary on the basis of the test compilation.
        compilation = compilations.test

        // Custom command line options for the linker.
        linkerOpts = ['-L/lib/search/path', '-L/another/search/path', '-lmylib']

        // Base name for the output file.
        baseName = 'foo'

        // Custom entry point function.
        entryPoint = 'org.example.main'

        // Accessing the output file.
        println("Executable path: ${outputFile.absolutePath}")

        // Accessing the link task.
        linkTask.dependsOn(additionalPreprocessingTask)

        // Accessing the run task.
        // Note that the runTask is null for non-host platforms.
        runTask?.dependsOn(prepareForRun)
    }

    framework('my_framework' [RELEASE]) {
        // Include a static library instead of a dynamic one into the framework.
        isStatic = true
    }
}
```

```
binaries {
    executable("my_executable", listOf(RELEASE)) {
        // Build a binary on the basis of the test compilation.
        compilation = compilations["test"]

        // Custom command line options for the linker.
        linkerOpts = mutableListOf("-L/lib/search/path", "-L/another/search/path", "-lmylib")

        // Base name for the output file.
        baseName = "foo"

        // Custom entry point function.
        entryPoint = "org.example.main"

        // Accessing the output file.
        println("Executable path: ${outputFile.absolutePath}")

        // Accessing the link task.
        linkTask.dependsOn(additionalPreprocessingTask)

        // Accessing the run task.
        // Note that the runTask is null for non-host platforms.
        runTask?.dependsOn(prepareForRun)
    }

    framework("my_framework" listOf(RELEASE)) {
        // Include a static library instead of a dynamic one into the framework.
        isStatic = true
    }
}
```

For more information on configuring binaries, see Building final native binaries.

CInterops

`cinterops` is a collection of descriptions for interop with native libraries. To provide an interop with a library, add an entry to `cinterops` and define its parameters:

Name	Description
`defFile`	`def` file describing the native API.
`packageName`	Package prefix for the generated Kotlin API.
`compilerOpts`	Options to pass to the compiler by the cinterop tool.
`includeDirs`	Directories to look for headers.

For more information on Kotlin interop with C libraries, see CInterop support.

```
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.main {
            cinterops {
                myInterop {
                    // Def-file describing the native API.
                    // The default path is src/nativeInterop/cinterop/<interop-name>.def
                    defFile project.file("def-file.def")

                    // Package to place the Kotlin API generated.
                    packageName 'org.sample'

                    // Options to be passed to compiler by cinterop tool.
                    compilerOpts '-Ipath/to/headers'

                    // Directories for header search (an analogue of the -I<path> compiler option).
                    includeDirs.allHeaders("path1", "path2")
```

```
                    // A shortcut for includeDirs.allHeaders.
                    includeDirs("include/directory", "another/directory")
                }

                anotherInterop { /* ... */ }
            }
        }
    }
}
```

```
kotlin {
    linuxX64 {  // Replace with a target you need.
        compilations.getByName("main") {
            val myInterop by cinterops.creating {
                // Def-file describing the native API.
                // The default path is src/nativeInterop/cinterop/<interop-name>.def
                defFile(project.file("def-file.def"))

                // Package to place the Kotlin API generated.
                packageName("org.sample")

                // Options to be passed to compiler by cinterop tool.
                compilerOpts("-Ipath/to/headers")

                // Directories for header search (an analogue of the -I<path> compiler option).
                includeDirs.allHeaders("path1", "path2")

                // A shortcut for includeDirs.allHeaders.
                includeDirs("include/directory", "another/directory")
            }

            val anotherInterop by cinterops.creating { /* ... */ }
        }
    }
}
```

Android 目标

The Kotlin multiplatform plugin contains two specific functions for android targets. Two functions help you configure build variants:

Name	Description
`publishLibraryVariants()`	Specifies build variants to publish. For usage instructions, see Publishing Android libraries.
`publishAllLibraryVariants()`	Publishes all build variants.

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

For more details about configuring Android targets of multiplatform projects, see Android Support.

> Note that the `android` configuration inside `kotlin` doesn't replace the build configuration of any Android project. For information on writing build scripts for Android projects, see the Android developer documentation.

源集

The `sourceSets` block describes source sets of the project. A source set contains Kotlin source files that participate in compilations together, along with their resources, dependencies, and language settings.

A multiplatform project contains predefined source sets for its targets; developers can also create custom source sets for their needs. For instructions on creating and configuring source sets, see Configuring source sets.

预定义源集

Predefined source sets are set up automatically upon creation of a multiplatform project. Available predefined source sets are the following:

Name	Description
`commonMain`	Code and resources shared between all platforms. Available in all multiplatform projects. Used in all main compilations of a project.
`commonTest`	Test code and resources shared between all platforms. Available in all multiplatform projects. Used in all test compilations of a project.
<targetName> *<compilationName>*	Target-specific sources for a compilation. *<targetName>* is the name of a predefined target and *<compilationName>* is the name of a compilation for this target. Examples: `jsTest`, `jvmMain`.

With Kotlin Gradle DSL, the sections of predefined source sets should be marked `by getting`.

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

```
kotlin {
    sourceSets {
        val commonMain by getting { /* ... */ }
    }
}
```

For more information about the predefined source sets, see Default Project Layout.

自定义源集

Custom source sets are created by the project developers manually. To create a custom source set, add a section with its name inside the `sourceSets` section. If using Kotlin Gradle DSL, mark custom source sets `by creating`.

```
kotlin {
    sourceSets {
        myMain { /* ... */ } // create or configure a source set by the name 'myMain'
    }
}
```

```
kotlin {
    sourceSets {
        val myMain by creating { /* ... */ } // create a new source set by the name 'MyMain'
    }
}
```

Note that a newly created source set isn't connected to other ones. To use it in the project's compilations, connect it with other source sets as described in Connecting source sets.

源集参数

Configurations of source sets are stored inside the corresponding blocks of `sourceSets`. A source set has the following parameters:

Name	Description
`kotlin.srcDir`	Location of Kotlin source files inside the source set directory.
`resources.srcDir`	Location of resources inside the source set directory.
`dependsOn`	Connection with another source set. The instructions on connecting source sets are provided in Connecting source sets.
`dependencies`	依赖项 of the source set.
`languageSettings`	语言设置 applied to the source set.

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

            dependencies {
                /* ... */
            }
        }
    }
}
```

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

            dependencies {
                /* ... */
            }
        }
    }
}
```

编译项

A target can have one or more compilations, for example, for production or testing. There are predefined compilations that are added automatically upon target creation. Developers can additionally create custom compilations.

To refer to all or some particular compilations of a target, use the `compilations` object collection. From `compilations`, you can refer to a compilation by its name.

预定义编译项

Predefined compilations are created automatically for each target of a project except for Android targets. Available predefined compilations are the following:

Name	Description
`main`	Compilation for production sources.
`test`	Compilation for tests.

```
kotlin {
    jvm {
        compilations.main.output // get the main compilation output
        compilations.test.runtimeDependencyFiles // get the test runtime classpath
    }
}
```

```
kotlin {
    jvm {
        val main by compilations.getting {
            output // get the main compilation output
        }

        compilations["test"].runtimeDependencyFiles // get the test runtime classpath
    }
}
```

自定义编译项

In addition to predefined compilations, developers can create their own custom compilations. To create a custom compilation, add a new item into the `compilations` collection. If using Kotlin Gradle DSL, mark custom compilations `by creating`.

```
kotlin {
    jvm() {
        compilations.create('integrationTest') {
            defaultSourceSet {
                dependencies {
                    /* ... */
                }
            }

            // Create a test task to run the tests produced by this compilation:
            tasks.create('jvmIntegrationTest', Test) {
                /* ... */
            }
        }
    }
}
```

```
kotlin {
    jvm() {
        compilations {
            val integrationTest by compilations.creating {
                defaultSourceSet {
                    dependencies {
                        /* ... */
                    }
                }

                // Create a test task to run the tests produced by this compilation:
                tasks.create<Test>("integrationTest") {
                    /* ... */
                }
            }
        }
    }
}
```

编译项参数

A compilation has the following parameters:

Name	Description
`defaultSourceSet`	The compilation's default source set.
`kotlinSourceSets`	Source sets participating in the compilation.
`allKotlinSourceSets`	Source sets participating in the compilation and their connections via `dependsOn()`.
`kotlinOptions`	Compiler options applied to the compilation. For the list of available options, see Compiler options.
`compileKotlinTask`	Gradle task for compiling Kotlin sources.
`compileKotlinTaskName`	Name of `compileKotlinTask`.
`compileAllTaskName`	Name of the Gradle task for compiling all sources of a compilation.
`output`	The compilation output.
`compileDependencyFiles`	Compile-time dependency files (classpath) of the compilation.
`runtimeDependencyFiles`	Runtime dependency files (classpath) of the compilation.

```
kotlin {
    jvm {
        compilations.main.kotlinOptions {
            // Setup the Kotlin compiler options for the 'main' compilation:
            jvmTarget = "1.8"
        }

        compilations.main.compileKotlinTask // get the Kotlin task 'compileKotlinJvm'
        compilations.main.output // get the main compilation output
        compilations.test.runtimeDependencyFiles // get the test runtime classpath
    }

    // Configure all compilations of all targets:
    targets.all {
        compilations.all {
            kotlinOptions {
                allWarningsAsErrors = true
            }
        }
    }
}
```

```
kotlin {
    jvm {
        val main by compilations.getting {
            kotlinOptions {
                // Setup the Kotlin compiler options for the 'main' compilation:
                jvmTarget = "1.8"
            }

            compileKotlinTask // get the Kotlin task 'compileKotlinJvm'
            output // get the main compilation output
        }

        compilations["test"].runtimeDependencyFiles // get the test runtime classpath
    }

    // Configure all compilations of all targets:
    targets.all {
        compilations.all {
            kotlinOptions {
                allWarningsAsErrors = true
            }
        }
    }
}
```

依赖项

The dependencies block of the source set declaration contains the dependencies of this source set. There are four kinds of dependencies:

Name	Description
`api`	Dependencies used in the API of the current module.
`implementation`	Dependencies used in the module but not exposed outside it.
`compileOnly`	Dependencies used only for compilation of the current module.
`runtimeOnly`	Dependencies available at runtime but not visible during compilation of any module.

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

Additionally, source sets can depend on each other. In this case, the dependsOn() function is used. Source set dependencies can also be declared in the top-level `dependencies` block of the build script. In this case, their declarations follow the pattern `<sourceSetName><DependencyKind>`, for example, `commonMainApi`.

```
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")
}
```

语言设置

The languageSettings block of a source set defines certain aspects of project analysis and build. The following language settings are available:

Name	Description
`languageVersion`	Provides source compatibility with the specified version of Kotlin.
`apiVersion`	Allows using declarations only from the specified version of Kotlin bundled libraries.
`enableLanguageFeature`	Enables the specified language feature. The available values correspond to the language features that are currently experimental or have been introduced as such at some point.
`useExperimentalAnnotation`	Allows using the specified opt-in annotation.
`progressiveMode`	Enables the progressive mode.

```
kotlin {
    sourceSets {
        commonMain {
            languageSettings {
                languageVersion = '1.3' // possible values: '1.0', '1.1', '1.2', '1.3'
                apiVersion = '1.3' // possible values: '1.0', '1.1', '1.2', '1.3'
                enableLanguageFeature('InlineClasses') // language feature name
                useExperimentalAnnotation('kotlin.ExperimentalUnsignedTypes') // annotation FQ-name
                progressiveMode = true // false by default
            }
        }
    }
}
```

```
kotlin {
    sourceSets {
        val commonMain by getting {
            languageSettings.apply {
                languageVersion = "1.3" // possible values: '1.0', '1.1', '1.2', '1.3'
                apiVersion = "1.3" // possible values: '1.0', '1.1', '1.2', '1.3'
                enableLanguageFeature("InlineClasses") // language feature name
                useExperimentalAnnotation("kotlin.ExperimentalUnsignedTypes") // annotation FQ-name
                progressiveMode = true // false by default
            }
        }
    }
}
```

更多语言结构

解构声明

有时把一个对象 *解构* 成很多变量会很方便，例如：

```
val (name, age) = person
```

这种语法称为 *解构声明*。一个解构声明同时创建多个变量。我们已经声明了两个新变量：`name` 和 `age`，并且可以独立使用它们：

```
println(name)
println(age)
```

一个解构声明会被编译成以下代码：

```
val name = person.component1()
val age = person.component2()
```

其中的 `component1()` 和 `component2()` 函数是在 Kotlin 中广泛使用的 *约定原则* 的另一个例子。（参见像 `+` 和 `*`、`for`-循环等操作符）。任何表达式都可以出现在解构声明的右侧，只要可以对它调用所需数量的 component 函数即可。当然，可以有 `component3()` 和 `component4()` 等等。

请注意，`componentN()` 函数需要用 `operator` 关键字标记，以允许在解构声明中使用它们。

解构声明也可以用在 `for`-循环中：当你写：

```
for ((a, b) in collection) { …… }
```

变量 `a` 和 `b` 的值取自对集合中的元素上调用 `component1()` 和 `component2()` 的返回值。

例：从函数中返回两个变量

让我们假设我们需要从一个函数返回两个东西。例如，一个结果对象和一个某种状态。在 Kotlin 中一个简洁的实现方式是声明一个 *数据类* 并返回其实例：

```
data class Result(val result: Int, val status: Status)
fun function(……): Result {
    // 各种计算

    return Result(result, status)
}

// 现在，使用该函数：
val (result, status) = function(……)
```

因为数据类自动声明 `componentN()` 函数，所以这里可以用解构声明。

注意：我们也可以使用标准类 `Pair` 并且让 `function()` 返回 `Pair<Int, Status>`，但是让数据合理命名通常更好。

例：解构声明和映射

可能遍历一个映射（map）最好的方式就是这样：

```
for ((key, value) in map) {
    // 使用该 key、value 做些事情
}
```

为使其能用，我们应该

— 通过提供一个 `iterator()` 函数将映射表示为一个值的序列；

— 通过提供函数 `component1()` 和 `component2()` 来将每个元素呈现为一对。

当然事实上，标准库提供了这样的扩展：

```
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 `for`-循环中对映射（以及数据类实例的集合等）自由使用解构声明。

下划线用于未使用的变量（自 1.1 起）

如果在解构声明中你不需要某个变量，那么可以用下划线取代其名称：

```
val (_, status) = getResult()
```

对于以这种方式跳过的组件，不会调用相应的 `componentN()` 操作符函数。

在 lambda 表达式中解构（自 1.1 起）

你可以对 lambda 表达式参数使用解构声明语法。如果 lambda 表达式具有 `Pair` 类型（或者 `Map.Entry` 或任何其他具有相应 `componentN` 函数的类型）的参数，那么可以通过将它们放在括号中来引入多个新参数来取代单个新参数：

```
map.mapValues { entry -> "${entry.value}!" }
map.mapValues { (key, value) -> "$value!" }
```

注意声明两个参数和声明一个解构对来取代单个参数之间的区别：

```
{ a //-> …… } // 一个参数
{ a, b //-> …… } // 两个参数
{ (a, b) //-> …… } // 一个解构对
{ (a, b), c //-> …… } // 一个解构对以及其他参数
```

如果解构的参数中的一个组件未使用，那么可以将其替换为下划线，以避免编造其名称：

```
map.mapValues { (_, value) -> "$value!" }
```

你可以指定整个解构的参数的类型或者分别指定特定组件的类型：

```
map.mapValues { (_, value): Map.Entry<Int, String> -> "$value!" }

map.mapValues { (_, value: String) -> "$value!" }
```

类型检测与类型转换：“is”与“as”

is 与 !is 操作符

我们可以在运行时通过使用 `is` 操作符或其否定形式 `!is` 来检测对象是否符合给定类型：

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // 与 !(obj is String) 相同
    print("Not a String")
}
else {
    print(obj.length)
}
```

智能转换

在许多情况下，不需要在 Kotlin 中使用显式转换操作符，因为编译器跟踪不可变值的 `is`-检测以及显式转换，并在需要时自动插入（安全的）转换：

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x 自动转换为字符串
    }
}
```

编译器足够聪明，能够知道如果反向检测导致返回那么该转换是安全的：

```
if (x !is String) return

print(x.length) // x 自动转换为字符串
```

或者在 `&&` 和 `||` 的右侧：

```
// `||` 右侧的 x 自动转换为字符串
if (x !is String || x.length == 0) return

// `&&` 右侧的 x 自动转换为字符串
if (x is String && x.length > 0) {
    print(x.length) // x 自动转换为字符串
}
```

这些 *智能转换* 用于 `when`-表达式 和 `while`-循环 也一样：

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

请注意，当编译器不能保证变量在检测和使用之间不可改变时，智能转换不能用。更具体地，智能转换能否适用根据以下规则：

- `val` 局部变量——总是可以，局部委托属性除外；
- `val` 属性——如果属性是 private 或 internal，或者该检测在声明属性的同一模块中执行。智能转换不适用于 open 的属性或者具有自定义 getter 的属性；
- `var` 局部变量——如果变量在检测和使用之间没有修改、没有在会修改它的 lambda 中捕获、并且不是局部委托属

性；

— `var` 属性——决不可能（因为该变量可以随时被其他代码修改）。

“不安全的”转换操作符

通常，如果转换是不可能的，转换操作符会抛出一个异常。因此，我们称之为*不安全的*。Kotlin 中的不安全转换由中缀操作符 `as`（参见operator precedence）完成：

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

请注意，`null` 不能转换为 `String` 因该类型不是可空的，即如果 `y` 为空，上面的代码会抛出一个异常。为了让这样的代码用于可空值，请在类型转换的右侧使用可空类型：

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

“安全的”（可空）转换操作符

为了避免抛出异常，可以使用*安全*转换操作符 `as?`，它可以在失败时返回 `null`：

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

请注意，尽管事实上 `as?` 的右边是一个非空类型的 `String`，但是其转换的结果是可空的。

类型擦除与泛型检测

Kotlin 在编译时确保涉及泛型操作的类型安全性，而在运行时，泛型类型的实例并未带有关于它们实际类型参数的信息。例如，`List<Foo>` 会被擦除为 `List<*>`。通常，在运行时无法检测一个实例是否属于带有某个类型参数的泛型类型。

为此，编译器会禁止由于类型擦除而无法执行的 `is` 检测，例如 `ints is List<Int>` 或者 `list is T`（类型参数）。当然，你可以对一个实例检测星投影的类型：

```
if (something is List<*>) {
    something.forEach { println(it) } // 这些项的类型都是 `Any?`
}
```

类似地，当已经让一个实例的类型参数（在编译期）静态检测，就可以对涉及非泛型部分做 `is` 检测或者类型转换。请注意，在这种情况下，会省略尖括号：

```
fun handleStrings(list: List<String>) {
    if (list is ArrayList) {
        // `list` 会智能转换为 `ArrayList<String>`
    }
}
```

省略类型参数的这种语法可用于不考虑类型参数的类型转换：`list as ArrayList`。

带有具体化的类型参数的内联函数使其类型实参在每个调用处内联，这就能够对类型参数进行 `arg is T` 检测，但是如果 `arg` 自身是一个泛型实例，**其**类型参数还是会被擦除。例如：

```
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>>() // 破坏类型安全!
```

非受检类型转换

如上所述，类型擦除使运行时不可能对泛型类型实例的类型实参进行检测，并且代码中的泛型可能相互连接不够紧密，以致于编译器无法确保类型安全。

即便如此，有时候我们有高级的程序逻辑来暗示类型安全。例如：

```
fun readDictionary(file: File): Map<String, *> = file.inputStream().use {
    TODO("Read a mapping of strings to arbitrary elements.")
}

// 我们已将存有一些 `Int` 的映射保存到该文件
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>
```

编译器会对最后一行的类型转换产生一个警告。该类型转换不能在运行时完全检测，并且不能保证映射中的值是“Int”。

为避免未受检类型转换，可以重新设计程序结构：在上例中，可以使用具有类型安全实现的不同接口 `DictionaryReader<T>` 与 `DictionaryWriter<T>`。可以引入合理的抽象，将未受检的类型转换从调用代码移动到实现细节中。正确使用泛型型变也有帮助。

对于泛型函数，使用具体化的类型参数可以使诸如 `arg as T` 这样的类型转换受检，除非 `arg` 对应类型的*自身*类型参数已被擦除。

可以通过在产生警告的语句或声明上用注解 `@Suppress("UNCHECKED_CAST")` 标注来禁止未受检类型转换警告：

```
inline fun <reified T> List<*>.asListOfType(): List<T>? =
    if (all { it is T })
        @Suppress("UNCHECKED_CAST")
        this as List<T> else
        null
```

在 JVM 平台中，数组类型（`Array<Foo>`）会保留关于其元素被擦除类型的信息，并且类型转换为一个数组类型可以部分受检：元素类型的可空性与类型实参仍然会被擦除。例如，如果 `foo` 是一个保存了任何 `List<*>`（无论可不可空）的数组的话，类型转换 `foo as Array<List<String>?>` 都会成功。

This 表达式

为了表示当前的 *接收者* 我们使用 `this` 表达式：

— 在类的成员中，`this` 指的是该类的当前对象。

— 在扩展函数或者带有接收者的函数字面值中，`this` 表示在点左侧传递的 *接收者* 参数。

如果 `this` 没有限定符，它指的是最内层的包含它的作用域。要引用其他作用域中的 `this`，请使用 *标签限定符*：

限定的 `this`

要访问来自外部作用域的`this`（一个类 或者扩展函数，或者带标签的带有接收者的函数字面值）我们使用 `this@label`，其中 `@label` 是一个代指 `this` 来源的标签：

```
class A { // 隐式标签 @A
    inner class B { // 隐式标签 @B
        fun Int.foo() { // 隐式标签 @foo
            val a = this@A // A 的 this
            val b = this@B // B 的 this

            val c = this // foo() 的接收者，一个 Int
            val c1 = this@foo // foo() 的接收者，一个 Int

            val funLit = lambda@ fun String.() {
                val d = this // funLit 的接收者
            }


            val funLit2 = { s: String ->
                // foo() 的接收者，因为它包含的 lambda 表达式
                // 没有任何接收者
                val d1 = this
            }
        }
    }
}
```

Implicit `this`

当对 `this` 调用成员函数时，可以省略 `this.` 部分。但是如果有一个同名的非成员函数时，请谨慎使用，因为在某些情况下会调用同名的非成员：

```
fun printLine() { println("Top-level function") }

class A {
    fun printLine() { println("Member function") }

    fun invokePrintLine(omitThis: Boolean = false)  {
        if (omitThis) printLine()
        else this.printLine()
    }
}

A().invokePrintLine() // Member function
A().invokePrintLine(omitThis = true) // Top-level function
```

相等性

Kotlin 中有两种类型的相等性：

— 结构相等（用 `equals()` 检测）；

— 引用相等（两个引用指向同一对象）。

结构相等

结构相等由 `==`（以及其否定形式 `!=`）操作判断。按照惯例，像 `a == b` 这样的表达式会翻译成：

```
a?.equals(b) ?: (b === null)
```

也就是说如果 `a` 不是 `null` 则调用 `equals(Any?)` 函数，否则（即 `a` 是 `null`）检测 b 是否与 `null` 引用相等。

请注意，当与 `null` 显式比较时完全没必要优化你的代码：`a == null` 会被自动转换为 `a === null`。

如需提供自定义的相等检测实现，请覆盖 `equals(other: Any?): Boolean` 函数。名称相同但签名不同的函数，如 `equals(other: Foo)` 并不会影响操作符 `==` 与 `!=` 的相等性检测。

结构相等与 `Comparable<……>` 接口定义的比较无关，因此只有自定义的 `equals(Any?)` 实现可能会影响该操作符的行为。

浮点数相等性

当相等性检测的两个操作数都是静态已知的（可空或非空的）`Float` 或 `Double` 类型时，该检测遵循 IEEE 754 浮点数运算标准。

否则会使用不符合该标准的结构相等性检测，这会导致 `NaN` 等于其自身，而 `-0.0` 不等于 `0.0`。

参见：浮点数比较。

引用相等

引用相等由 `===`（以及其否定形式 `!==`）操作判断。`a === b` 当且仅当 `a` 与 `b` 指向同一个对象时求值为 true。对于运行时表示为原生类型的值（例如 `Int`），`===` 相等检测等价于 `==` 检测。

操作符重载

Kotlin 允许我们为自己的类型提供预定义的一组操作符的实现。这些操作符具有固定的符号表示（如 `+` 或 `*`）和固定的优先级。为实现这样的操作符，我们为相应的类型（即二元操作符左侧的类型和一元操作符的参数类型）提供了一个固定名字的成员函数或扩展函数。重载操作符的函数需要用 `operator` 修饰符标记。

另外，我们描述为不同操作符规范操作符重载的约定。

一元操作

一元前缀操作符

表达式	翻译为
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

这个表是说，当编译器处理例如表达式 `+a` 时，它执行以下步骤：

— 确定 `a` 的类型，令其为 `T`；

— 为接收者 `T` 查找一个带有 `operator` 修饰符的无参函数 `unaryPlus()`，即成员函数或扩展函数；

— 如果函数不存在或不明确，则导致编译错误；

— 如果函数存在且其返回类型为 `R`，那就表达式 `+a` 具有类型 `R`；

注意 这些操作以及所有其他操作都针对基本类型做了优化，不会为它们引入函数调用的开销。

以下是如何重载一元减运算符的示例：

```
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)  // 输出“Point(x=-10, y=-20)”
}
```

递增与递减

表达式	翻译为
`a++`	`a.inc()` + 见下文
`a--`	`a.dec()` + 见下文

`inc()` 和 `dec()` 函数必须返回一个值，它用于赋值给使用 `++` 或 `--` 操作的变量。它们不应该改变在其上调用 `inc()` 或 `dec()` 的对象。

编译器执行以下步骤来解析*后缀*形式的操作符，例如 `a++`：

— 确定 `a` 的类型，令其为 `T`；

— 查找一个适用于类型为 `T` 的接收者的、带有 `operator` 修饰符的无参数函数 `inc()`；

— 检测函数的返回类型是 `T` 的子类型。

计算表达式的步骤是：

— 把 `a` 的初始值存储到临时存储 `a0` 中；

— 把 `a.inc()` 结果赋值给 `a`；

— 把 `a0` 作为表达式的结果返回。

对于 `a--`，步骤是完全类似的。

对于*前缀*形式 `++a` 和 `--a` 以相同方式解析，其步骤是：

— 把 `a.inc()` 结果赋值给 `a`；

— 把 `a` 的新值作为表达式结果返回。

二元操作

算术运算符

表达式	翻译为
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`、`a.mod(b)`（已弃用）
`a..b`	`a.rangeTo(b)`

对于此表中的操作，编译器只是解析成*翻译为*列中的表达式。

请注意，自 Kotlin 1.1 起支持 `rem` 运算符。Kotlin 1.0 使用 `mod` 运算符，它在 Kotlin 1.1 中被弃用。

示例

下面是一个从给定值起始的 Counter 类的示例，它可以使用重载的 `+` 运算符来增加计数：

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

“In”操作符

表达式	翻译为
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

对于 `in` 和 `!in`，过程是相同的，但是参数的顺序是相反的。

索引访问操作符

表达式	翻译为
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ……, i_n]`	`a.get(i_1, ……, i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ……, i_n] = b`	`a.set(i_1, ……, i_n, b)`

方括号转换为调用带有适当数量参数的 `get` 和 `set`。

调用操作符

表达式	翻译为
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ……, i_n)`	`a.invoke(i_1, ……, i_n)`

圆括号转换为调用带有适当数量参数的 `invoke`。

广义赋值

表达式	翻译为
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)`,`a.modAssign(b)`(已弃用)

对于赋值操作,例如 `a += b`,编译器执行以下步骤:

— 如果右列的函数可用
  — 如果相应的二元函数(即 `plusAssign()` 对应于 `plus()`)也可用,那么报告错误(模糊),
  — 确保其返回类型是 `Unit`,否则报告错误,
  — 生成 `a.plusAssign(b)` 的代码;
— 否则试着生成 `a = a + b` 的代码(这里包含类型检测:`a + b` 的类型必须是 `a` 的子类型)。

注意:赋值在 Kotlin 中 *不是*表达式。

相等与不等操作符

表达式	翻译为
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

这些操作符只使用函数 `equals(other: Any?): Boolean`,可以覆盖它来提供自定义的相等性检测实现。不会调用任何其他同名函数(如 `equals(other: Foo)`)。

注意: `===` 和 `!==`（同一性检测）不可重载，因此不存在对他们的约定。

这个 `==` 操作符有些特殊：它被翻译成一个复杂的表达式，用于筛选 `null` 值。`null == null` 总是 true，对于非空的 `x`，`x == null` 总是 false 而不会调用 `x.equals()`。

比较操作符

表达式	翻译为
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

所有的比较都转换为对 `compareTo` 的调用，这个函数需要返回 `Int` 值

属性委托操作符

`provideDelegate`、`getValue` 以及 `setValue` 操作符函数已在委托属性中描述。

具名函数的中缀调用

我们可以通过中缀函数的调用 来模拟自定义中缀操作符。

空安全

可空类型与非空类型

Kotlin 的类型系统旨在消除来自代码空引用的危险，也称为《十亿美元的错误》。

许多编程语言（包括 Java）中最常见的陷阱之一，就是访问空引用的成员会导致空引用异常。在 Java 中，这等同于 `NullPointerException` 或简称 `NPE`。

Kotlin 的类型系统旨在从我们的代码中消除 `NullPointerException`。NPE 的唯一可能的原因可能是：

- 显式调用 `throw NullPointerException()`；
- 使用了下文描述的 `!!` 操作符；
- 有些数据在初始化时不一致，例如当：
  - 传递一个在构造函数中出现的未初始化的 `this` 并用于其他地方（“泄漏 `this`”）；
  - 超类的构造函数调用一个开放成员，该成员在派生中类的实现使用了未初始化的状态；
- Java 互操作：
  - 企图访问平台类型的 `null` 引用的成员；
  - 用于具有错误可空性的 Java 互操作的泛型类型，例如一段 Java 代码可能会向 Kotlin 的 `MutableList<String>` 中加入 `null`，这意味着应该使用 `MutableList<String?>` 来处理它；
  - 由外部 Java 代码引发的其他问题。

在 Kotlin 中，类型系统区分一个引用可以容纳 `null`（可空引用）还是不能容纳（非空引用）。例如，String 类型的常规变量不能容纳 `null`：

```
var a: String = "abc" // 默认情况下，常规初始化意味着非空
a = null // 编译错误
```

如果要允许为空，我们可以声明一个变量为可空字符串，写作 `String?`：

```
var b: String? = "abc" // 可以设置为空
b = null // ok
print(b)
```

现在，如果你调用 `a` 的方法或者访问它的属性，它保证不会导致 `NPE`，这样你就可以放心地使用：

```
val l = a.length
```

但是如果你想访问 `b` 的同一个属性，那么这是不安全的，并且编译器会报告一个错误：

```
val l = b.length // 错误：变量“b”可能为空
```

但是我们还是需要访问该属性，对吧?有几种方式可以做到。

在条件中检测 `null`

首先，你可以显式检测 `b` 是否为 `null`，并分别处理两种可能：

```
val l = if (b != null) b.length else -1
```

编译器会跟踪所执行检测的信息，并允许你在 `if` 内部调用 `length`。同时，也支持更复杂（更智能）的条件：

```
val b: String? = "Kotlin"
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
```

请注意，这只适用于 `b` 是不可变的情况（即在检测和使用之间没有修改过的局部变量，或者不可覆盖并且有幕后字段的 `val` 成员），因为否则可能会发生在检测之后 `b` 又变为 `null` 的情况。

安全的调用

你的第二个选择是安全调用操作符，写作 `?.`：

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length) // 无需安全调用
```

如果 `b` 非空，就返回 `b.length`，否则返回 `null`，这个表达式的类型是 `Int?`。

安全调用在链式调用中很有用。例如，如果一个员工 Bob 可能会（或者不会）分配给一个部门，并且可能有另外一个员工是该部门的负责人，那么获取 Bob 所在部门负责人（如果有的话）的名字，我们写作：

```
bob?.department?.head?.name
```

如果任意一个属性（环节）为空，这个链式调用就会返回 `null`。

如果要只对非空值执行某个操作，安全调用操作符可以与 `let` 一起使用：

```
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
     item?.let { println(it) } // 输出 Kotlin 并忽略 null
}
```

安全调用也可以出现在赋值的左侧。这样，如果调用链中的任何一个接收者为空都会跳过赋值，而右侧的表达式根本不会求值：

```
// 如果 `person` 或者 `person.department` 其中之一为空，都不会调用该函数：
person?.department?.head = managersPool.getManager()
```

Elvis 操作符

当我们有一个可空的引用 `b` 时，我们可以说“如果 `b` 非空，我使用它；否则使用某个非空的值”：

```
val l: Int = if (b != null) b.length else -1
```

除了完整的 `if`-表达式，这还可以通过 Elvis 操作符表达，写作 `?:`：

```
val l = b?.length ?: -1
```

如果 `?:` 左侧表达式非空，elvis 操作符就返回其左侧表达式，否则返回右侧表达式。请注意，当且仅当左侧为空时，才会对右侧表达式求值。

请注意，因为 `throw` 和 `return` 在 Kotlin 中都是表达式，所以它们也可以用在 elvis 操作符右侧。这可能会非常方便，例如，检测函数参数：

```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ……
}
```

!! 操作符

第三种选择是为 NPE 爱好者准备的：非空断言运算符（`!!`）将任何值转换为非空类型，若该值为空则抛出异常。我们可以写 `b!!`，这会返回一个非空的 `b` 值（例如：在我们例子中的 `String`）或者如果 `b` 为空，就会抛出一个 `NPE` 异常：

```
val l = b!!.length
```

因此，如果你想要一个 NPE，你可以得到它，但是你必须显式要求它，否则它不会不期而至。

安全的类型转换

如果对象不是目标类型，那么常规类型转换可能会导致 `ClassCastException`。另一个选择是使用安全的类型转换，如果尝试转换不成功则返回 `null`：

```
val aInt: Int? = a as? Int
```

可空类型的集合

如果你有一个可空类型元素的集合，并且想要过滤非空元素，你可以使用 `filterNotNull` 来实现：

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

异常

异常类

Kotlin 中所有异常类都是 `Throwable` 类的子孙类。每个异常都有消息、堆栈回溯信息以及可选的原因。

使用 `throw`-表达式来抛出异常：

```
throw Exception("Hi There!")
```

使用 `try`-表达式来捕获异常：

```
try {
    // 一些代码
}
catch (e: SomeException) {
    // 处理程序
}
finally {
    // 可选的 finally 块
}
```

可以有零到多个 `catch` 块。`finally` 块可以省略。但是 `catch` 与 `finally` 块至少应该存在一个。

Try 是一个表达式

`try` 是一个表达式，即它可以有一个返回值：

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

`try`-表达式的返回值是 `try` 块中的最后一个表达式或者是（所有）`catch` 块中的最后一个表达式。`finally` 块中的内容不会影响表达式的结果。

受检的异常

Kotlin 没有受检的异常。这其中有很多原因，但我们会提供一个简单的例子。

以下是 JDK 中 `StringBuilder` 类实现的一个示例接口：

```
Appendable append(CharSequence csq) throws IOException;
```

这个签名是什么意思？它是说，每次我追加一个字符串到一些东西（一个 `StringBuilder`、某种日志、一个控制台等）上时我就必须捕获那些 `IOException`。为什么？因为它可能正在执行 IO 操作（`Writer` 也实现了 `Appendable`）…… 所以它导致这种代码随处可见的出现：

```
try {
    log.append(message)
}
catch (IOException e) {
    // 必须要安全
}
```

这并不好，参见《Effective Java》第三版 第 77 条：*不要忽略异常*。

Bruce Eckel says about checked exceptions:

> 通过一些小程序测试得出的结论是异常规范会同时提高开发者的生产力与代码质量，但是大型软件项目的经验表明一个不同的结论——生产力降低、代码质量很少或没有提高。

其他相关引证：

— 《Java 的受检异常是一个错误》（Java's checked exceptions were a mistake）（Rod Waldhoff）

— 《受检异常的烦恼》（The Trouble with Checked Exceptions）（Anders Hejlsberg）

If you want to alert callers of possible exceptions when calling Kotlin code from Java, Swift, or Objective-C, you can use the `@Throws` annotation. Read more about using this annotation for Java as well as for Swift and Objective-C.

Nothing 类型

在 Kotlin 中 `throw` 是表达式，所以你可以使用它（比如）作为 Elvis 表达式的一部分：

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

`throw` 表达式的类型是特殊类型 `Nothing`。该类型没有值，而是用于标记永远不能达到的代码位置。在你自己的代码中，你可以使用 `Nothing` 来标记一个永远不会返回的函数：

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

当你调用该函数时，编译器会知道在该调用后就不再继续执行了：

```
val s = person.name ?: fail("Name required")
println(s)     // 在此已知“s”已初始化
```

可能会遇到这个类型的另一种情况是类型推断。这个类型的可空变体 `Nothing?` 有一个可能的值是 `null`。如果用 `null` 来初始化一个要推断类型的值，而又没有其他信息可用于确定更具体的类型时，编译器会推断出 `Nothing?` 类型：

```
val x = null           // “x”具有类型 `Nothing?`
val l = listOf(null)   // “l”具有类型 `List<Nothing?>
```

Java 互操作性

与 Java 互操作性相关的信息，请参见 Java 互操作性章节中的异常部分。

注解

注解声明

注解是将元数据附加到代码的方法。要声明注解，请将 `annotation` 修饰符放在类的前面：

```
annotation class Fancy
```

注解的附加属性可以通过用元注解标注注解类来指定：

- `@Target` 指定可以用该注解标注的元素的可能的类型（类、函数、属性、表达式等）；
- `@Retention` 指定该注解是否存储在编译后的 class 文件中，以及它在运行时能否通过反射可见（默认都是 true）；
- `@Repeatable` 允许在单个元素上多次使用相同的该注解；
- `@MustBeDocumented` 指定该注解是公有 API 的一部分，并且应该包含在生成的 API 文档中显示的类或方法的签名中。

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

用法

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
        return (@Fancy 1)
    }
}
```

如果需要对类的主构造函数进行标注，则需要在构造函数声明中添加 `constructor` 关键字，并将注解添加到其前面：

```
class Foo @Inject constructor(dependency: MyDependency) { …… }
```

你也可以标注属性访问器：

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

构造函数

注解可以有接受参数的构造函数。

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

允许的参数类型有：

- 对应于 Java 原生类型的类型（Int、Long等）；
- 字符串；

— 类（`Foo::class`）；
— 枚举；
— 其他注解；
— 上面已列类型的数组。

注解参数不能有可空类型，因为 JVM 不支持将 `null` 作为注解属性的值存储。

如果注解用作另一个注解的参数，则其名称不以 @ 字符为前缀：

```
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"))
```

如果需要将一个类指定为注解的参数，请使用 Kotlin 类（KClass）。Kotlin 编译器会自动将其转换为 Java 类，以便 Java 代码能够正常访问该注解与参数 。

```
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

注解也可以用于 lambda 表达式。它们会被应用于生成 lambda 表达式体的 `invoke()` 方法上。这对于像 Quasar 这样的框架很有用，该框架使用注解进行并发控制。

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

注解使用处目标

当对属性或主构造函数参数进行标注时，从相应的 Kotlin 元素生成的 Java 元素会有多个，因此在生成的 Java 字节码中该注解有多个可能位置 。如果要指定精确地指定应该如何生成该注解，请使用以下语法：

```
class Example(@field:Ann val foo,    // 标注 Java 字段
              @get:Ann val bar,      // 标注 Java getter
              @param:Ann val quux)   // 标注 Java 构造函数参数
```

可以使用相同的语法来标注整个文件。要做到这一点，把带有目标 `file` 的注解放在文件的顶层、package 指令之前或者在所有导入之前（如果文件在默认包中的话）：

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

如果你对同一目标有多个注解，那么可以这样来避免目标重复——在目标后面添加方括号并将所有注解放在方括号内：

```
class Example {
     @set:[Inject VisibleForTesting]
     var collaborator: Collaborator
}
```

支持的使用处目标的完整列表为：

- `file`；
- `property`（具有此目标的注解对 Java 不可见）；
- `field`；
- `get`（属性 getter）；
- `set`（属性 setter）；
- `receiver`（扩展函数或属性的接收者参数）；
- `param`（构造函数参数）；
- `setparam`（属性 setter 参数）；
- `delegate`（为委托属性存储其委托实例的字段）。

要标注扩展函数的接收者参数，请使用以下语法：

```
fun @receiver:Fancy String.myExtension() { ... }
```

如果不指定使用处目标，则根据正在使用的注解的 `@Target` 注解来选择目标。如果有多个适用的目标，则使用以下列表中的第一个适用目标：

- `param`；
- `property`；
- `field`。

Java 注解

Java 注解与 Kotlin 100% 兼容：

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
import org.junit.rules.*

class Tests {
    // 将 @Rule 注解应用于属性 getter
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

因为 Java 编写的注解没有定义参数顺序，所以不能使用常规函数调用语法来传递参数。相反，你需要使用具名参数语法：

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

就像在 Java 中一样，一个特殊的情况是 `value` 参数；它的值无需显式名称指定：

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

数组作为注解参数

如果 Java 中的 `value` 参数具有数组类型，它会成为 Kotlin 中的一个 `vararg` 参数：

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

对于具有数组类型的其他参数，你需要显式使用数组字面值语法（自 Kotlin 1.2 起）或者 `arrayOf(……)`：

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```
// Kotlin 1.2+:
@AnnWithArrayMethod(names = ["abc", "foo", "bar"])
class C

// 旧版本 Kotlin:
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar"))
class D
```

访问注解实例的属性

注解实例的值会作为属性暴露给 Kotlin 代码：

```
// Java
public @interface Ann {
    int value();
}
```

```
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

反射

反射是这样的一组语言和库功能，它允许在运行时自省你的程序的结构。Kotlin 让语言中的函数和属性做为一等公民、并对其自省（即在运行时获悉一个名称或者一个属性或函数的类型）与简单地使用函数式或响应式风格紧密相关。

> 在 Java 平台上，使用反射功能所需的运行时组件作为单独的 JAR 文件（`kotlin-reflect.jar`）分发。这样做是为了减少不使用反射功能的应用程序所需的运行时库的大小。如果你需要使用反射，请确保该 .jar文件添加到项目的 classpath 中。

类引用

最基本的反射功能是获取 Kotlin 类的运行时引用。要获取对静态已知的 Kotlin 类的引用，可以使用 *类字面值* 语法：

```
val c = MyClass::class
```

该引用是 KClass 类型的值。

请注意，Kotlin 类引用与 Java 类引用不同。要获得 Java 类引用，请在 `KClass` 实例上使用 `.java` 属性。

绑定的类引用（自 1.1 起）

通过使用对象作为接收者，可以用相同的 `::class` 语法获取指定对象的类的引用：

```
val widget: Widget = ……
assert(widget is GoodWidget) { "Bad widget: ${widget::class.qualifiedName}" }
```

你可以获取对象的精确类的引用，例如 `GoodWidget` 或 `BadWidget`，尽管接收者表达式的类型是 `Widget`。

可调用引用

函数、属性以及构造函数的引用，除了作为自省程序结构外，还可以用于调用或者用作函数类型的实例。

所有可调用引用的公共超类型是 `KCallable<out R>`，其中 `R` 是返回值类型，对于属性是属性类型，对于构造函数是所构造类型。

函数引用

当我们有一个具名函数声明如下：

```
fun isOdd(x: Int) = x % 2 != 0
```

我们可以很容易地直接调用它（`isOdd(5)`），但是我们也可以将其作为一个函数类型的值，例如将其传给另一个函数。为此，我们使用 `::` 操作符：

```
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd))
```

这里 `::isOdd` 是函数类型 `(Int) -> Boolean` 的一个值。

函数引用属于 `KFunction<out R>` 的子类型之一，取决于参数个数，例如 `KFunction3<T1, T2, T3, R>`。

当上下文中已知函数期望的类型时，`::` 可以用于重载函数。例如：

```
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)) // 引用到 isOdd(x: Int)
```

或者，你可以通过将方法引用存储在具有显式指定类型的变量中来提供必要的上下文：

```
val predicate: (String) -> Boolean = ::isOdd   // 引用到 isOdd(x: String)
```

如果我们需要使用类的成员函数或扩展函数，它需要是限定的，例如 `String::toCharArray`。

请注意，即使以扩展函数的引用初始化一个变量，其推断出的函数类型也会没有接收者（它会有一个接受接收者对象的额外参数）。如需改为带有接收者的函数类型，请明确指定其类型：

```
val isEmptyStringList: List<String>.() -> Boolean = List<String>::isEmpty
```

示例：函数组合

考虑以下函数：

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

它返回一个传给它的两个函数的组合：`compose(f, g) = f(g(*))`。现在，你可以将其应用于可调用引用：

```
fun length(s: String) = s.length

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength))
```

属性引用

要把属性作为 Kotlin中 的一等对象来访问，我们也可以使用 `::` 运算符：

```
val x = 1

fun main() {
    println(::x.get())
    println(::x.name)
}
```

表达式 `::x` 求值为 `KProperty<Int>` 类型的属性对象，它允许我们使用 `get()` 读取它的值，或者使用 `name` 属性来获取属性名。更多信息请参见关于 KProperty 类的文档。

对于可变属性，例如 `var y = 1`，`::y` 返回 `KMutableProperty<Int>` 类型的一个值，该类型有一个 `set()` 方法。

```
var y = 1

fun main() {
    ::y.set(2)
    println(y)
}
```

属性引用可以用在预期具有单个泛型参数的函数的地方：

```
val strs = listOf("a", "bc", "def")
println(strs.map(String::length))
```

要访问属于类的成员的属性，我们这样限定它：

```
class A(val p: Int)
val prop = A::p
println(prop.get(A(1)))
```

对于扩展属性：

```
val String.lastChar: Char
    get() = this[length - 1]

fun main() {
    println(String::lastChar.get("abc"))
}
```

与 Java 反射的互操作性

在Java平台上，标准库包含反射类的扩展，它提供了与 Java 反射对象之间映射（参见 `kotlin.reflect.jvm` 包）。例如，要查找一个用作 Kotlin 属性 getter 的 幕后字段或 Java方法，可以这样写：

```
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main() {
    println(A::p.javaGetter) // 输出 "public final int A.getP()"
    println(A::p.javaField)  // 输出 "private final int A.p"
}
```

要获得对应于 Java 类的 Kotlin 类，请使用 `.kotlin` 扩展属性：

```
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

构造函数引用

构造函数可以像方法和属性那样引用。他们可以用于期待这样的函数类型对象的任何地方：它与该构造函数接受相同参数并且返回相应类型的对象。通过使用 `::` 操作符并添加类名来引用构造函数。考虑下面的函数，它期待一个无参并返回 `Foo` 类型的函数参数：

```
class Foo

fun function(factory: () -> Foo) {
    val x: Foo = factory()
}
```

使用 `::Foo`，类 Foo 的零参数构造函数，我们可以这样简单地调用它：

```
function(::Foo)
```

构造函数的可调用引用的类型也是 KFunction<out R> 的子类型之一，取决于其参数个数。

绑定的函数与属性引用（自 1.1 起）

你可以引用特定对象的实例方法：

```
val numberRegex = "\\d+".toRegex()
println(numberRegex.matches("29"))

val isNumber = numberRegex::matches
println(isNumber("29"))
```

取代直接调用方法 `matches` 的是我们存储其引用。这样的引用会绑定到其接收者上。它可以直接调用（如上例所示）或者用于任何期待一个函数类型表达式的时候：

```
val numberRegex = "\\d+".toRegex()
val strings = listOf("abc", "124", "a70")
println(strings.filter(numberRegex::matches))
```

比较绑定的类型和相应的未绑定类型的引用。绑定的可调用引用有其接收者“附加”到其上，因此接收者的类型不再是参数：

```
val isNumber: (CharSequence) -> Boolean = numberRegex::matches

val matches: (Regex, CharSequence) -> Boolean = Regex::matches
```

属性引用也可以绑定：

```
val prop = "abc"::length
println(prop.get())
```

自 Kotlin 1.2 起，无需显式指定 `this` 作为接收者：`this::foo` 与 `::foo` 是等价的。

绑定的构造函数引用

inner 类的构造函数的绑定的可调用引用可通过提供外部类的实例来获得：

```
class Outer {
    inner class Inner
}

val o = Outer()
val boundInnerCtor = o::Inner
```

作用域函数

Kotlin 标准库包含几个函数，它们的唯一目的是在对象的上下文中执行代码块。当对一个对象调用这样的函数并提供一个 lambda 表达式时，它会形成一个临时作用域。在此作用域中，可以访问该对象而无需其名称。这些函数称为*作用域函数*。共有以下五种：`let`、`run`、`with`、`apply` 以及 `also`。

这些函数基本上做了同样的事情：在一个对象上执行一个代码块。不同的是这个对象在块中如何使用，以及整个表达式的结果是什么。

下面是作用域函数的典型用法：

```
Person("Alice", 20, "Amsterdam").let {
    println(it)
    it.moveTo("London")
    it.incrementAge()
    println(it)
}
```

如果不使用 `let` 来写这段代码，就必须引入一个新变量，并在每次使用它时重复其名称。

```
val alice = Person("Alice", 20, "Amsterdam")
println(alice)
alice.moveTo("London")
alice.incrementAge()
println(alice)
```

作用域函数没有引入任何新的技术，但是它们可以使你的代码更加简洁易读。

由于作用域函数的相似性质，为你的案例选择正确的函数可能有点棘手。选择主要取决于你的意图和项目中使用的一致性。下面我们将详细描述各种作用域函数及其约定用法之间的区别。

区别

由于作用域函数本质上都非常相似，因此了解它们之间的区别很重要。每个作用域函数之间有两个主要区别：

- 引用上下文对象的方式
- 返回值

上下文对象：`this` 还是 `it`

在作用域函数的 lambda 表达式里，上下文对象可以不使用其实际名称而是使用一个更简短的引用来访问。每个作用域函数都使用以下两种方式之一来访问上下文对象：作为 lambda 表达式的接收者（`this`）或者作为 lambda 表达式的参数（`it`）。两者都提供了同样的功能，因此我们将针对不同的场景描述两者的优缺点，并提供使用建议。

```
fun main() {
    val str = "Hello"
    // this
    str.run {
        println("The receiver string length: $length")
        //println("The receiver string length: ${this.length}") // 和上句效果相同
    }

    // it
    str.let {
        println("The receiver string's length is ${it.length}")
    }
}
```

this

`run`、`with` 以及 `apply` 通过关键字 `this` 引用上下文对象。因此，在它们的 lambda 表达式中可以像在普通的类函数中一样访问上下文对象。在大多数场景，当你访问接收者对象时你可以省略 `this`，来让你的代码更简短。相对地，如果省略了 `this`，就很难区分接收者对象的成员及外部对象或函数。因此，对于主要对对象成员进行操作（调用其函数或赋值其属性）的 lambda 表达式，建议将上下文对象作为接收者（`this`）。

```
val adam = Person("Adam").apply {
    age = 20                       // 和 this.age = 20 或者 adam.age = 20 一样
    city = "London"
}
println(adam)
```

it

反过来，`let` 及 `also` 将上下文对象作为 lambda 表达式参数。如果没有指定参数名，对象可以用隐式默认名称 `it` 访问。`it` 比 `this` 简短，带有 `it` 的表达式通常更容易阅读。然而，当调用对象函数或属性时，不能像 `this` 这样隐式地访问对象。因此，当上下文对象在作用域中主要用作函数调用中的参数时，使用 `it` 作为上下文对象会更好。若在代码块中使用多个变量，则 `it` 也更好。

```
fun getRandomInt(): Int {
    return Random.nextInt(100).also {
        writeToLog("getRandomInt() generated value $it")
    }
}

val i = getRandomInt()
```

此外，当将上下文对象作为参数传递时，可以为上下文对象指定在作用域内的自定义名称。

```
fun getRandomInt(): Int {
    return Random.nextInt(100).also { value ->
        writeToLog("getRandomInt() generated value $value")
    }
}

val i = getRandomInt()
```

返回值

根据返回结果，作用域函数可以分为以下两类：

— `apply` 及 `also` 返回上下文对象。

— `let`、`run` 及 `with` 返回 lambda 表达式结果.

这两个选项使你可以根据在代码中的后续操作来选择适当的函数。

上下文对象

`apply` 及 `also` 的返回值是上下文对象本身。因此，它们可以作为辅助步骤包含在调用链中：你可以继续在同一个对象上进行链式函数调用。

```
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()
```

它们还可以用在返回上下文对象的函数的 return 语句中。

```
fun getRandomInt(): Int {
    return Random.nextInt(100).also {
        writeToLog("getRandomInt() generated value $it")
    }
}

val i = getRandomInt()
```

Lambda 表达式结果

`let`、`run` 及 `with` 返回 lambda 表达式的结果。所以，在需要使用其结果给一个变量赋值，或者在需要对其结果进行链式操作等情况下，可以使用它们。

```
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.")
```

此外，还可以忽略返回值，仅使用作用域函数为变量创建一个临时作用域。

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    val firstItem = first()
    val lastItem = last()
    println("First item: $firstItem, last item: $lastItem")
}
```

几个函数

为了帮助你为你的场景选择合适的作用域函数，我们会详细地描述它们并且提供一些使用建议。从技术角度来说，作用域函数在很多场景里是可以互换的，所以这些示例展示了定义通用使用风格的约定用法。

`let`

上下文对象作为 lambda 表达式的参数（`it`）来访问。**返回值**是 lambda 表达式的结果。

`let` 可用于在调用链的结果上调用一个或多个函数。例如，以下代码打印对集合的两个操作的结果：

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
val resultList = numbers.map { it.length }.filter { it > 3 }
println(resultList)
```

使用 `let`，可以写成这样：

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let {
    println(it)
    // 如果需要可以调用更多函数
}
```

若代码块仅包含以 `it` 作为参数的单个函数，则可以使用方法引用(`::`)代替 lambda 表达式：

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let(::println)
```

`let` 经常用于仅使用非空值执行代码块。如需对非空对象执行操作，可对其使用安全调用操作符 `?.` 并调用 `let` 在 lambda 表达式中执行操作。

```
val str: String? = "Hello"
//processNonNullString(str)       // 编译错误: str 可能为空
val length = str?.let {
    println("let() called on $it")
    processNonNullString(it)      // 编译通过: 'it' 在 '?.let { }' 中必不为空
    it.length
}
```

使用 `let` 的另一种情况是引入作用域受限的局部变量以提高代码的可读性。如需为上下文对象定义一个新变量，可提供其名称作为 lambda 表达式参数来替默认的 `it`。

```
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'")
```

with

一个非扩展函数：**上下文对象**作为参数传递，但是在 lambda 表达式内部，它可以作为接收者（`this`）使用。**返回值**是 lambda 表达式结果。

我们建议使用 `with` 来调用上下文对象上的函数，而不使用 lambda 表达式结果。在代码中，`with` 可以理解为“*对于这个对象，执行以下操作。*”

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    println("'with' is called with argument $this")
    println("It contains $size elements")
}
```

`with` 的另一个使用场景是引入一个辅助对象，其属性或函数将用于计算一个值。

```
val numbers = mutableListOf("one", "two", "three")
val firstAndLast = with(numbers) {
    "The first element is ${first()}," +
    " the last element is ${last()}"
}
println(firstAndLast)
```

run

上下文对象 作为接收者（`this`）来访问。**返回值** 是 lambda 表达式结果。

`run` 和 `with` 做同样的事情，但是调用方式和 `let` 一样——作为上下文对象的扩展函数.

当 lambda 表达式同时包含对象初始化和返回值的计算时，`run` 很有用。

```
val service = MultiportService("https://example.kotlinlang.org", 80)

val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}

// 同样的代码如果用 let() 函数来写:
val letResult = service.let {
    it.port = 8080
    it.query(it.prepareRequest() + " to port ${it.port}")
}
```

除了在接收者对象上调用 `run` 之外，还可以将其用作非扩展函数。非扩展 `run` 可以使你在需要表达式的地方执行一个由多个语句组成的块。

```
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)
}
```

apply

上下文对象 作为接收者（`this`）来访问。**返回值** 是上下文对象本身。

对于不返回值且主要在接收者（`this`）对象的成员上运行的代码块使用 `apply`。`apply` 的常见情况是对象配置。这样的调用可以理解为“*将以下赋值操作应用于对象*”。

```
val adam = Person("Adam").apply {
    age = 32
    city = "London"
}
println(adam)
```

将接收者作为返回值，你可以轻松地将 `apply` 包含到调用链中以进行更复杂的处理。

also

上下文对象作为 lambda 表达式的参数（`it`）来访问。**返回值**是上下文对象本身。

`also` 对于执行一些将上下文对象作为参数的操作很有用。对于需要引用对象而不是其属性与函数的操作，或者不想屏蔽来自外部作用域的 `this` 引用时，请使用 `also`。

当你在代码中看到 `also` 时，可以将其理解为“*并且用该对象执行以下操作*”。

```
val numbers = mutableListOf("one", "two", "three")
numbers
    .also { println("The list elements before adding new one: $it") }
    .add("four")
```

函数选择

为了帮助你选择合适的作用域函数，我们提供了它们之间的主要区别表。

函数	对象引用	返回值	是否是扩展函数
`let`	`it`	Lambda 表达式结果	是
`run`	`this`	Lambda 表达式结果	是
`run`	-	Lambda 表达式结果	不是：调用无需上下文对象
`with`	`this`	Lambda 表达式结果	不是：把上下文对象当做参数
`apply`	`this`	上下文对象	是
`also`	`it`	上下文对象	是

以下是根据预期目的选择作用域函数的简短指南：

- 对一个非空（non-null）对象执行 lambda 表达式：`let`
- 将表达式作为变量引入为局部作用域中：`let`
- 对象配置：`apply`
- 对象配置并且计算结果：`run`
- 在需要表达式的地方运行语句：非扩展的 `run`
- 附加效果：`also`
- 一个对象的一组函数调用：`with`

不同函数的使用场景存在重叠，你可以根据项目或团队中使用的特定约定选择函数。

尽管作用域函数是使代码更简洁的一种方法，但请避免过度使用它们：这会降低代码的可读性并可能导致错误。避免嵌套作用域函数，同时链式调用它们时要小心：此时很容易对当前上下文对象及 `this` 或 `it` 的值感到困惑。

`takeIf` 与 `takeUnless`

除了作用域函数外，标准库还包含函数 `takeIf` 及 `takeUnless`。这俩函数使你可以将对象状态检查嵌入到调用链中。

当以提供的谓词在对象上进行调用时，若该对象与谓词匹配，则 `takeIf` 返回此对象。否则返回 `null`。因此，`takeIf` 是单个对象的过滤函数。反之，`takeUnless` 如果不匹配谓词，则返回对象，如果匹配则返回 `null`。该对象作为 lambda 表达式参数（`it`）来访问。

```
val number = Random.nextInt(100)

val evenOrNull = number.takeIf { it % 2 == 0 }
val oddOrNull = number.takeUnless { it % 2 == 0 }
println("even: $evenOrNull, odd: $oddOrNull")
```

当在 `takeIf` 及 `takeUnless` 之后链式调用其他函数，不要忘记执行空检查或安全调用（`?.`），因为他们的返回值是可为空的。

```
val str = "Hello"
val caps = str.takeIf { it.isNotEmpty() }?.toUpperCase()
/val caps = str.takeIf { it.isNotEmpty() }.toUpperCase() // 编译错误
println(caps)
```

`takeIf` 及 `takeUnless` 与作用域函数一起特别有用。一个很好的例子是用 `let` 链接它们，以便在与给定谓词匹配的对象上运行代码块。为此，请在对象上调用 `takeIf`，然后通过安全调用（`?.`）调用 `let`。对于与谓词不匹配的对象，`takeIf` 返回 `null`，并且不调用 `let`。

```
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")
```

没有标准库函数时，相同的函数看起来是这样的：

```
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")
```

类型安全的构建器

通过使用命名得当的函数作为构建器，结合带有接收者的函数字面值，可以在 Kotlin 中创建类型安全、静态类型的构建器。

类型安全的构建器可以创建基于 Kotlin 的适用于采用半声明方式构建复杂层次数据结构领域专用语言（DSL）。以下是构建器的一些示例应用场景：

— 使用 Kotlin 代码生成标记语言，例如 HTML 或 XML；

— 以编程方式布局 UI 组件：Anko；

— 为 Web 服务器配置路由：Ktor。

一个类型安全的构建器示例

考虑下面的代码：

```
import com.example.html.* // 参见下文声明

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"}

            // 一个具有属性和文本内容的元素
            a(href = "http://kotlinlang.org") {+"Kotlin"}

            // 混合的内容
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "http://kotlinlang.org") {+"Kotlin"}
                +"project"
```

```
            }
            p {+"some text"}

            // 以下代码生成的内容
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

这是完全合法的 Kotlin 代码。你可以在这里在线运行上文代码(修改它并在浏览器中运行)。

实现原理

让我们来看看 Kotlin 中实现类型安全构建器的机制。首先,我们需要定义我们想要构建的模型,在本例中我们需要建模 HTML 标签。用一些类就可以轻易完成。例如,`HTML` 是一个描述 `<html>` 标签的类,也就是说它定义了像 `<head>` 和 `<body>` 这样的子标签。(参见下文它的声明。)

现在,让我们回想下为什么我们可以在代码中这样写:

```
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。该函数定义如下:

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

这个函数接受一个名为 `init` 的参数,该参数本身就是一个函数。该函数的类型是 `HTML.() -> Unit`,它是一个*带接收者的函数类型*。这意味着我们需要向函数传递一个 HTML 类型的实例(*接收者*),并且我们可以在函数内部调用该实例的成员。该接收者可以通过 `this` 关键字访问:

```
html {
    this.head { …… }
    this.body { …… }
}
```

(`head` 和 `body` 是 `HTML` 的成员函数。)

现在,像往常一样,`this` 可以省略掉了,我们得到的东西看起来已经非常像一个构建器了:

```
html {
    head { …… }
    body { …… }
}
```

那么,这个调用做什么?让我们看看上面定义的 `html` 函数的主体。它创建了一个 `HTML` 的新实例,然后通过调用作为参数传入的函数来初始化它(在我们的示例中,归结为在HTML实例上调用 `head` 和 `body`),然后返回此实例。这正是构建器所应做的。

`HTML` 类中的 `head` 和 `body` 函数的定义与 `html` 类似。唯一的区别是,它们将构建的实例添加到包含 `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同样的事情，所以我们可以有一个泛型版本，`initTag`：

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

所以，现在我们的函数很简单：

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

并且我们可以使用它们来构建 `<head>` 和 `<body>` 标签。

这里要讨论的另一件事是如何向标签体中添加文本。在上例中我们这样写到：

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ……
}
```

所以基本上，我们只是把一个字符串放进一个标签体内部，但在它前面有一个小的 `+`，所以它是一个函数调用，调用一个前缀 `unaryPlus()` 操作。该操作实际上是由一个扩展函数 `unaryPlus()` 定义的，该函数是 `TagWithText` 抽象类（`Title` 的父类）的成员：

```
operator fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

所以，在这里前缀 `+` 所做的事情是把一个字符串包装到一个 `TextElement` 实例中，并将其添加到 `children` 集合中，以使其成为标签树的一个适当的部分。

所有这些都在上面构建器示例顶部导入的包 `com.example.html` 中定义。在最后一节中，你可以阅读这个包的完整定义。

作用域控制：@DslMarker（自 1.1 起）

使用 DSL 时，可能会遇到上下文中可以调用太多函数的问题。我们可以调用 lambda 表达式内部每个可用的隐式接收者的方法，因此得到一个不一致的结果，就像在另一个 `head` 内部的 `head` 标记那样：

```
html {
    head {
        head {} // 应该禁止
    }
    // ……
}
```

在这个例子中，必须只有最近层的隐式接收者 `this@head` 的成员可用；`head()` 是外部接收者 `this@html` 的成员，所以调用它一定是非法的。

为了解决这个问题，在 Kotlin 1.1 中引入了一种控制接收者作用域的特殊机制。

为了使编译器开始控制标记，我们只是必须用相同的标记注解来标注在 DSL 中使用的所有接收者的类型。例如，对于 HTML 构建器，我们声明一个注解 `@HTMLTagMarker`：

```
@DslMarker
annotation class HtmlTagMarker
```

如果一个注解类使用 `@DslMarker` 注解标注，那么该注解类称为 DSL 标记。

在我们的 DSL 中，所有标签类都扩展了相同的超类 `Tag`。只需使用 `@HtmlTagMarker` 来标注超类就足够了，之后，Kotlin 编译器会将所有继承的类视为已标注：

```
@HtmlTagMarker
abstract class Tag(val name: String) { …… }
```

我们不必用 `@HtmlTagMarker` 标注 `HTML` 或 `Head` 类，因为它们的超类已标注过：

```
class HTML() : Tag("html") { …… }
class Head() : Tag("head") { …… }
```

在添加了这个注解之后，Kotlin 编译器就知道哪些隐式接收者是同一个 DSL 的一部分，并且只允许调用最近层的接收者的成员：

```
html {
    head {
        head { } // 错误：外部接收者的成员
    }
    // ……
}
```

请注意，仍然可以调用外部接收者的成员，但是要做到这一点，你必须明确指定这个接收者：

```
html {
    head {
        this@html.head { } // 可能
    }
    // ……
}
```

`com.example.html` 包的完整定义

这就是 `com.example.html` 包的定义（只有上面例子中使用的元素）。它构建一个 HTML 树。代码中大量使用了扩展函数和带有接收者的 lambda 表达式。

请注意，`@DslMarker` 注解在 Kotlin 1.1 起才可用。

```
package com.example.html
```

```
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
}
```

选择加入的要求

> 要求选择加入的注解 `@RequiresOptIn` 与 `@OptIn` 是 *实验性的*。请参阅以下用法详细信息。

> 1.3.70 中引入了 `@RequireOptIn` 与 `@OptIn` 注解以取代先前使用的 `@Experimental` 与 `@UseExperimental`；同时 `-Xopt-in` 编译器选项也取代了 `-Xuse-experimental`。

Kotlin 标准库提供了一种机制，用于要求并明确同意使用 API 的某些元素。通过这种机制，库开发人员可以将使用其 API 需要选择加入的特定条件告知用户，例如，如果某个 API 处于实验状态，并且将来可能会更改。

为了避免潜在的问题，编译器会向此类 API 的用户发出警告，告知他们这些条件，并要求他们在使用 API 之前选择加入。

选择使用 API

如果库作者将一个库的 API 声明标记为 *要求选择加入* 你应该明确同意在代码中使用它。有多种方式可以选择加入使用此类 API，所有方法均不受技术限制。你可以自由选择最适合自己的方式。

传播选择加入

在使用供第三方（库）使用的 API 时，你也可以把其选择加入的要求传播到自己的 API。为此，请在你的 API 主体声明中添加注解 *要求选择加入的注解*。这可以让你使用带有此注解的 API 元素。

```
// 库代码
@RequiresOptIn(message = "This API is experimental. It may be changed in the future without notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // 要求选择加入的注解

@MyDateTime
class DateProvider // 要求选择加入的类
```

```
// 客户端代码
fun getYear(): Int {
    val dateProvider: DateProvider // 错误：DateProvider 要求选择加入
    // ...
}

@MyDateTime
fun getDate(): Date {
    val dateProvider: DateProvider // OK：该函数也需要选择加入
    // ...
}

fun displayDate() {
    println(getDate()) // 错误：getDate() 需要选择加入
}
```

如本例所示，带注释的函数看起来是 `@MyDateTime` API 的一部分。因此，这种选择加入会将选择加入的要求传播到客户端代码；其客户将看到相同的警告消息，并且也必须同意。要使用多个需要选择加入的API，请在声明中标记所有需要选择加入的注解。

非传播的用法

在不公开其自身API的模块（例如应用程序）中，你可以选择使用 API 而无需将选择加入的要求传播到代码中。这种情况下，请使用 @OptIn 标记你的声明，并以要求选择加入的注解作为参数：

```
// 库代码
@RequiresOptIn(message = "This API is experimental. It may be changed in the future without notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // 要求选择加入的注解

@MyDateTime
class DateProvider // 要求选择加入的类
```

```
//客户端代码
@OptIn(MyDateTime::class)
fun getDate(): Date { // 使用 DateProvider；不传播选择加入的要求
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    println(getDate()) // OK：不要求选择加入
}
```

当有人调用函数 `getDate()` 时，不会通知他们函数主体中使用的选择加入 API 的要求。

要在一个文件的所有函数和类中使用要求选择加入的 API，请在文件的顶部，文件包说明和导入声明前添加文件级注释 `@file:OptIn`。

```
 //客户端代码
 @file:OptIn(MyDateTime::class)
```

模块范围的选择加入

如果你不想在使用要求选择加入 API 的每个地方都添加注解，则可以为整个模块选择加入这些 API。要选择在模块中使用 API，请使用参数 `-Xopt-in` 进行编译，使用 `-Xopt-in = org.mylibrary.OptInAnnotation` 指定该 API 使用的要求选择加入注解的标准名称。使用此参数进行编译的效果与模块中每个声明都有注解 `@OptIn(OptInAnnotation::class)` 的效果相同。

如果使用 Gradle 构建模块，可以添加如下参数：

```
compileKotlin {
    kotlinOptions {
        freeCompilerArgs += "-Xopt-in=org.mylibrary.OptInAnnotation"
    }
}
```

```
tasks.withType<KotlinCompile>().all {
    kotlinOptions.freeCompilerArgs += "-Xopt-in=org.mylibrary.OptInAnnotation"
}
```

如果你的 Gradle 模块是多平台模块，请使用 `useExperimentalAnnotation` 方法：

```
sourceSets {
    all {
        languageSettings {
            useExperimentalAnnotation('org.mylibrary.OptInAnnotation')
        }
    }
}
```

```
sourceSets {
    all {
        languageSettings.useExperimentalAnnotation("org.mylibrary.OptInAnnotation")
    }
}
```

对于 Maven，它将是：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>...</executions>
            <configuration>
                <args>
                    <arg>-Xopt-in=org.mylibrary.OptInAnnotation</arg>
                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

要在模块级别选择加入多个 API，请为每个要求选择加入的 API 添加以上描述的参数之一。

要求选择加入 API

要求选择加入的注解

如果想获得使用者使用你的模块 API 的明确同意，请创建一个注解类，作为_要求选择加入的注解_。这个类必须使用 @RequiresOptIn 注解：

```
@RequiresOptIn
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime
```

要求选择加入的注解必须满足以下几个要求：

— `BINARY` retention

— targets中没有 `EXPRESSION` 与 `FILE`

— 没有参数

选择加入的要求可以具有以下两个严格级别之一：

— `RequiresOptIn.Level.ERROR`。选择加入是强制性的。否则，使用标记 API 的代码将无法编译。默认级别。

— `RequiresOptIn.Level.WARNING`。选择加入不是强制性的，而是建议使用的。没有它，编译器会发出警告。

要设置所需的级别，请指定 `@RequiresOptIn` 注解的 `level` 参数。

另外，你可以提供一个 `message` 来通知用户有关使用该 API 的特定条件。编译器会将其显示给使用该 API 但未选择加入的用户。

```
@RequiresOptIn(level = RequiresOptIn.Level.WARNING, message = "This API is experimental. It can be
incompatibly changed in the future.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime
```

如果你发布了多个需要选择加入的独立功能，请为每个功能声明一个注解。这使你的用户可以更安全地使用 API：他们只能使用其明确接受的功能。这也使你可以独立地从功能中删除选择加入的要求。

标记 API 元素

要在使用 API 时要求选择加入，请给它的声明添加要求选择加入的注解。

```
@MyDateTime
class DateProvider

@MyDateTime
fun getTime(): Time {}
```

实验 API 的选择加入要求

如果要求选择加入实验状态的特性，请仔细处理 API 由实验状态到稳定状态的转换，以避免破坏客户端代码。

API 结束实验并以稳定状态发布后，请从声明中删除其要求选择加入的注解。客户端将可以不受限制地使用它们。但是，你应该将注解类留在模块中，以便与现有的客户端代码保持兼容。

为了让 API 用户相应地更新其模块（从代码中删除注解并重新编译），请将注解标记为 `@Deprecated` 并在弃用 message 中提供说明。

```
@Deprecated("This opt-in requirement is not used anymore. Remove its usages from your code.")
@RequiresOptIn
annotation class ExperimentalDateTime
```

选择加入要求的实验状态

选择加入要求的机制在 Kotlin 1.3 中是实验性的。这意味着在将来的版本中，可能会以不兼容的方式进行更改。

为了让使用注解 `@OptIn` 和 `@RequiresOptIn` 的用户了解其实验状态，编译器会在编译代码时发出警告：

`This class can only be used with the compiler argument '-Xopt-in=kotlin.RequiresOptIn'`

要移除警告，请添加编译器参数 `-Xopt-in=kotlin.RequiresOptIn`。

参考

关键字与操作符

硬关键字

以下符号会始终解释为关键字，不能用作标识符：

- `as`
  - 用于类型转换
  - 为导入指定一个别名
- `as?` 用于安全类型转换
- `break` 终止循环的执行
- `class` 声明一个类
- `continue` 继续最近层循环的下一步
- `do` 开始一个 do/while 循环（后置条件的循环）
- `else` 定义一个 if 表达式条件为 false 时执行的分支
- `false` 指定布尔类型的“假”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`
  - 指定在 for 循环中迭代的对象
  - 用作中缀操作符以检测一个值属于一个区间、一个集合或者其他定义“contains”方法的实体
  - 在 when 表达式中用于上述目的
  - 将一个类型参数标记为逆变
- `!in`
  - 用作中缀操作符以检测一个值**不**属于一个区间、一个集合或者其他定义“contains”方法的实体
  - 在 when 表达式中用于上述目的
- `interface` 声明一个接口
- `is`
  - 检测一个值具有指定类型
    在 when 表达式中用于上述目的
    —
- `!is`

  - 检测一个值**不**具有指定类型
  - 在 when 表达式中用于上述目的
- `null` 是表示不指向任何对象的对象引用的常量
- `object` 同时声明一个类及其实例
- `package` 指定当前文件的包
- `return` 从最近层的函数或匿名函数返回
- `super`
  - 引用一个方法或属性的超类实现
  - 在次构造函数中调用超类构造函数
- `this`
  - 引用当前接收者
  - 在次构造函数中调用同一个类的另一个构造函数
- `throw` 抛出一个异常
- `true` 指定布尔类型的“真”值
- `try` 开始一个异常处理块
- `typealias` 声明一个类型别名
- `typeof` 保留以供未来使用
- `val` 声明一个只读属性或局部变量
- `var` 声明一个可变属性或局部变量
- `when` 开始一个 when 表达式（执行其中一个给定分支）
- `while` 开始一个 while 循环（前置条件的循环）

软关键字

以下符号在适用的上下文中充当关键字，而在其他上下文中可用作标识符：

- `by`
  - 将接口的实现委托给另一个对象
  - 将属性访问器的实现委托给另一个对象
- `catch` 开始一个处理指定异常类型的块
- `constructor` 声明一个主构造函数或次构造函数
- `delegate` 用作注解使用处目标
- `dynamic` 引用一个 Kotlin/JS 代码中的动态类型
- `field` 用作注解使用处目标
- `file` 用作注解使用处目标
- `finally` 开始一个当 try 块退出时总会执行的块
- `get`

  - 声明属性的 getter
  - 用作注解使用处目标
- `import` 将另一个包中的声明导入当前文件
- `init` 开始一个初始化块
- `param` 用作注解使用处目标
- `property` 用作注解使用处目标
- `receiver` 用作注解使用处目标
- `set`
  - 声明属性的 setter
  - 用作注解使用处目标
- `setparam` 用作注解使用处目标
- `where` 指定泛型类型参数的约束

修饰符关键字

以下符号作为声明中修饰符列表中的关键字，并可用作其他上下文中的标识符：

- `actual` 表示多平台项目中的一个平台相关实现
- `abstract` 将一个类或成员标记为抽象
- `annotation` 声明一个注解类
- `companion` 声明一个伴生对象
- `const` 将属性标记为编译期常量
- `crossinline` 禁止传递给内联函数的 lambda 中的非局部返回
- `data` 指示编译器为类生成典型成员
- `enum` 声明一个枚举
- `expect` 将一个声明标记为平台相关，并期待在平台模块中实现。
- `external` 将一个声明标记为不是在 Kotlin 中实现（通过 JNI 访问或者在 JavaScript 中实现）
- `final` 禁止成员覆盖
- `infix` 允许以中缀表示法调用函数
- `inline` 告诉编译器在调用处内联传给它的函数和 lambda 表达式
- `inner` 允许在嵌套类中引用外部类实例
- `internal` 将一个声明标记为在当前模块中可见
- `lateinit` 允许在构造函数之外初始化非空属性
- `noinline` 关闭传给内联函数的 lambda 表达式的内联
- `open` 允许一个类子类化或覆盖成员
- `operator` 将一个函数标记为重载一个操作符或者实现一个约定
- `out` 将类型参数标记为协变

- `override` 将一个成员标记为超类成员的覆盖
- `private` 将一个声明标记为在当前类或文件中可见
- `protected` 将一个声明标记为在当前类及其子类中可见
- `public` 将一个声明标记为在任何地方可见
- `reified` 将内联函数的类型参数标记为在运行时可访问
- `sealed` 声明一个密封类（限制子类化的类）
- `suspend` 将一个函数或 lambda 表达式标记为挂起式（可用做协程）
- `tailrec` 将一个函数标记为尾递归（允许编译器将递归替换为迭代）
- `vararg` 允许一个参数传入可变数量的参数

特殊标识符

以下标识符由编译器在指定上下文中定义，并且可以用作其他上下文中的常规标识符：

- `field` 用在属性访问器内部来引用该属性的幕后字段
- `it` 用在 lambda 表达式内部来隐式引用其参数

操作符和特殊符号

Kotlin 支持以下操作符和特殊符号：

- `+`、`-`、`*`、`/`、`%` —— 数学操作符
  - `*` 也用于将数组传递给 vararg 参数
- `=`
  - 赋值操作符
  - 也用于指定参数的默认值
- `+=`、`-=`、`*=`、`/=`、`%=` —— 广义赋值操作符
- `++`、`--` —— 递增与递减操作符
- `&&`、`||`、`!` —— 逻辑“与”、“或”、“非”操作符（对于位运算，请使用相应的中缀函数）
- `==`、`!=` —— 相等操作符（对于非原生类型会翻译为调用 `equals()`）
- `===`、`!==` —— 引用相等操作符
- `<`、`>`、`<=`、`>=` —— 比较操作符（对于非原生类型会翻译为调用 `compareTo()`）
- `[`、`]` —— 索引访问操作符（会翻译为调用 `get` 与 `set`）
- `!!` 断言一个表达式非空
- `?.` 执行安全调用（如果接收者非空，就调用一个方法或访问一个属性）
- `?:` 如果左侧的值为空，就取右侧的值（elvis 操作符）
- `::` 创建一个成员引用或者一个类引用
- `..` 创建一个区间
- `:` 分隔声明中的名称与类型

- `?` 将类型标记为可空
- `->`
  - 分隔 lambda 表达式的参数与主体
  - 分隔在函数类型中的参数类型与返回类型声明
  - 分隔 when 表达式分支的条件与代码体
- `@`
  - 引入一个注解
  - 引入或引用一个循环标签
  - 引入或引用一个 lambda 表达式标签
  - 引用一个来自外部作用域的 “this” 表达式
  - 引用一个外部超类
- `;` 分隔位于同一行的多个语句
- `$` 在字符串模版中引用变量或者表达式
- `_`
  - 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

Description

Notation

The notation used on this page corresponds to the ANTLR 4 notation with a few exceptions for better readability:

— omitted lexer rule actions and commands,
— omitted lexical modes.

Short description:

— operator `|` denotes *alternative*,
— operator `*` denotes *iteration* (zero or more),
— operator `+` denotes *iteration* (one or more),
— operator `?` denotes *option* (zero or one),
— operator `..` denotes *range* (from left to right),
— operator `~` denotes *negation*.

Grammar source files

Kotlin grammar source files (in ANTLR format) are located in the Kotlin specification repository:

— **KotlinLexer.g4** describes lexical structure;
— **UnicodeClasses.g4** describes the characters that can be used in identifiers (these rules are omitted on this page for better readability);
— **KotlinParser.g4** describes syntax.

The grammar on this page corresponds to the grammar files above.

Symbols and naming

Terminal symbol names start with an uppercase letter, e.g. Identifier.
Non-terminal symbol names start with a lowercase letter, e.g. kotlinFile.

Symbol definitions may be documented with *attributes*:

— `start` attribute denotes a symbol that represents the whole source file (see kotlinFile and script),
— `helper` attribute denotes a lexer fragment rule (used only inside other terminal symbols).

Also for better readability some simplifications are made:

— lexer rules consisting of one string literal element are inlined to the use site,
— new line tokens are excluded (new lines are not allowed in some places, see source grammar files for details).

Scope

The grammar corresponds to the latest stable version of the Kotlin compiler excluding lexer and parser rules for experimental features that are disabled by default.

Syntax grammar

General

Relevant pages: Packages

```
start
kotlinFile
 : shebangLine? fileAnnotation* packageHeader importList topLevelObject* EOF
 ;
start
script
 : shebangLine? fileAnnotation* packageHeader importList (statement semi)* EOF
 ;
shebangLine
(used by kotlinFile, script)
 : ShebangLine
 ;
fileAnnotation
(used by kotlinFile, script)
 : ANNOTATION_USE_SITE_TARGET_FILE (('[' unescapedAnnotation+ ']') | unescapedAnnotation)
 ;
```

See Packages

```
packageHeader
(used by kotlinFile, script)
 : ('package' identifier semi?)?
 ;
```

See Imports

```
importList
(used by kotlinFile, script)
 : importHeader*
 ;
importHeader
(used by importList)
 : 'import' identifier ((.' '*') | importAlias)? semi?
 ;
importAlias
(used by importHeader)
 : 'as' simpleIdentifier
 ;
topLevelObject
(used by kotlinFile)
 : declaration semis?
 ;
typeAlias
(used by declaration)
 : modifiers? 'typealias' simpleIdentifier typeParameters? '=' type
 ;
declaration
(used by topLevelObject, classMemberDeclaration, statement)
 : classDeclaration
 | objectDeclaration
 | functionDeclaration
 | propertyDeclaration
 | typeAlias
 ;
```

Classes

See Classes and Inheritance

```
classDeclaration
(used by declaration)
 : modifiers? ('class' | 'interface')
   simpleIdentifier typeParameters?
   primaryConstructor?
   (':' delegationSpecifiers)?
   typeConstraints?
   (classBody | enumClassBody)?
 ;
primaryConstructor
(used by classDeclaration)
 : (modifiers? 'constructor')? classParameters
 ;
```

classBody
(used by classDeclaration, companionObject, objectDeclaration, enumEntry, objectLiteral)
 : '{' classMemberDeclarations '}'
 ;
classParameters
(used by primaryConstructor)
 : '(' (classParameter (',' classParameter)*)? ')'
 ;
classParameter
(used by classParameters)
 : modifiers? ('val' | 'var')? simpleIdentifier ':' type ('=' expression)?
 ;
delegationSpecifiers
(used by classDeclaration, companionObject, objectDeclaration, objectLiteral)
 : annotatedDelegationSpecifier (',' annotatedDelegationSpecifier)*
 ;
delegationSpecifier
(used by annotatedDelegationSpecifier)
 : constructorInvocation
 | explicitDelegation
 | userType
 | functionType
 ;
constructorInvocation
(used by delegationSpecifier, unescapedAnnotation)
 : userType valueArguments
 ;
annotatedDelegationSpecifier
(used by delegationSpecifiers)
 : annotation* delegationSpecifier
 ;
explicitDelegation
(used by delegationSpecifier)
 : (userType | functionType) 'by' expression
 ;

See Generic classes

typeParameters
(used by typeAlias, classDeclaration, functionDeclaration, propertyDeclaration)
 : '<' typeParameter (',' typeParameter)* '>'
 ;
typeParameter
(used by typeParameters)
 : typeParameterModifiers? simpleIdentifier (':' type)?
 ;

See Generic constraints

typeConstraints
(used by classDeclaration, functionDeclaration, propertyDeclaration, anonymousFunction)
 : 'where' typeConstraint (',' typeConstraint)*
 ;
typeConstraint
(used by typeConstraints)
 : annotation* simpleIdentifier ':' type
 ;

Class members

classMemberDeclarations
(used by classBody, enumClassBody)
 : (classMemberDeclaration semis?)*
 ;
classMemberDeclaration
(used by classMemberDeclarations)
 : declaration
 | companionObject
 | anonymousInitializer
 | secondaryConstructor
 ;
anonymousInitializer
(used by classMemberDeclaration)
 : 'init' block
 ;
companionObject
(used by classMemberDeclaration)
 : modifiers? 'companion' 'object' simpleIdentifier?
   (':' delegationSpecifiers)?
   classBody?
 ;
functionValueParameters
(used by functionDeclaration, secondaryConstructor, anonymousFunction)
 : '(' (functionValueParameter (',' functionValueParameter)*)? ')'
 ;
functionValueParameter
(used by functionValueParameters)
 : modifiers? parameter ('=' expression)?

;
functionDeclaration
(used by declaration)
: modifiers? 'fun' typeParameters?
(receiverType '.')?
simpleIdentifier functionValueParameters
(':' type)? typeConstraints?
functionBody?
;
functionBody
(used by functionDeclaration, getter, setter, anonymousFunction)
: block
| '=' expression
;
variableDeclaration
(used by multiVariableDeclaration, propertyDeclaration, forStatement, lambdaParameter, whenSubject)
: annotation* simpleIdentifier (':' type)?
;
multiVariableDeclaration
(used by propertyDeclaration, forStatement, lambdaParameter)
: '(' variableDeclaration (',' variableDeclaration)* ')'
;

See Properties and Fields

propertyDeclaration
(used by declaration)
: modifiers? ('val' | 'var') typeParameters?
(receiverType '.')?
(multiVariableDeclaration | variableDeclaration)
typeConstraints?
(('=' expression) | propertyDelegate)? ';'?
((getter? (semi? setter)?) | (setter? (semi? getter)?))
;
propertyDelegate
(used by propertyDeclaration)
: 'by' expression
;
getter
(used by propertyDeclaration)
: modifiers? 'get'
| modifiers? 'get' '(' ')'
(':' type)?
functionBody
;
setter
(used by propertyDeclaration)
: modifiers? 'set'
| modifiers? 'set' '(' (annotation | parameterModifier)* setterParameter ')'
(':' type)?
functionBody
;
setterParameter
(used by setter)
: simpleIdentifier (':' type)?
;
parameter
(used by functionValueParameter, functionTypeParameters)
: simpleIdentifier ':' type
;

See Object expressions and Declarations

objectDeclaration
(used by declaration)
: modifiers? 'object' simpleIdentifier (':' delegationSpecifiers)? classBody?
;
secondaryConstructor
(used by classMemberDeclaration)
: modifiers? 'constructor' functionValueParameters
(':' constructorDelegationCall)? block?
;
constructorDelegationCall
(used by secondaryConstructor)
: 'this' valueArguments
| 'super' valueArguments
;

Enum classes

See Enum classes

enumClassBody
(used by classDeclaration)
: '{' enumEntries? (';' classMemberDeclarations)? '}'
;
enumEntries

(used by enumClassBody)
 : enumEntry (',' enumEntry)* ','?
 ;
enumEntry
(used by enumEntries)
 : modifiers? simpleIdentifier valueArguments? classBody?
 ;

Types

See Types

type
(used by typeAlias, classParameter, typeParameter, typeConstraint, functionDeclaration, variableDeclaration, getter, setter, setterParameter, parameter, typeProjection, functionType, functionTypeParameters, parenthesizedType, infixOperation, asExpression, lambdaParameter, anonymousFunction, superExpression, typeTest, catchBlock)
 : typeModifiers? (parenthesizedType | nullableType | typeReference | functionType)
 ;
typeReference
(used by type, nullableType, receiverType)
 : userType
 | 'dynamic'
 ;
nullableType
(used by type, receiverType)
 : (typeReference | parenthesizedType) quest+
 ;
quest
(used by nullableType)
 : '?'
 | QUEST_WS
 ;
userType
(used by delegationSpecifier, constructorInvocation, explicitDelegation, typeReference, parenthesizedUserType, unescapedAnnotation)
 : simpleUserType ('.' simpleUserType)*
 ;
simpleUserType
(used by userType)
 : simpleIdentifier typeArguments?
 ;
typeProjection
(used by typeArguments)
 : typeProjectionModifiers? type
 | '*'
 ;
typeProjectionModifiers
(used by typeProjection)
 : typeProjectionModifier+
 ;
typeProjectionModifier
(used by typeProjectionModifiers)
 : varianceModifier
 | annotation
 ;
functionType
(used by delegationSpecifier, explicitDelegation, type)
 : (receiverType '.')? functionTypeParameters '->' type
 ;
functionTypeParameters
(used by functionType)
 : '(' (parameter | type)? (',' (parameter | type))* ')'
 ;
parenthesizedType
(used by type, nullableType, receiverType)
 : '(' type ')'
 ;
receiverType
(used by functionDeclaration, propertyDeclaration, functionType, callableReference)
 : typeModifiers? (parenthesizedType | nullableType | typeReference)
 ;
parenthesizedUserType
(used by parenthesizedUserType)
 : '(' userType ')'
 | '(' parenthesizedUserType ')'
 ;

Statements

statements
(used by block, lambdaLiteral)
 : (statement (semis statement)* semis?)?
 ;
statement
(used by script, statements, controlStructureBody)

: (label | annotation)* (declaration | assignment | loopStatement | expression)
;

See Returns and jumps

label
(used by statement, unaryPrefix, annotatedLambda)
: IdentifierAt
;
controlStructureBody
(used by forStatement, whileStatement, doWhileStatement, ifExpression, whenEntry)
: block
| statement
;
block
(used by anonymousInitializer, functionBody, secondaryConstructor, controlStructureBody, tryExpression, catchBlock, finallyBlock)
: '{' statements '}'
;
loopStatement
(used by statement)
: forStatement
| whileStatement
| doWhileStatement
;
forStatement
(used by loopStatement)
: 'for'
'(' annotation* (variableDeclaration | multiVariableDeclaration) 'in' expression ')'
controlStructureBody?
;
whileStatement
(used by loopStatement)
: 'while' '(' expression ')' controlStructureBody
| 'while' '(' expression ')' ';'
;
doWhileStatement
(used by loopStatement)
: 'do' controlStructureBody? 'while' '(' expression ')'
;
assignment
(used by statement)
: directlyAssignableExpression '=' expression
| assignableExpression assignmentAndOperator expression
;
semi
(used by script, packageHeader, importHeader, propertyDeclaration, whenEntry)
: EOF
;
semis
(used by topLevelObject, classMemberDeclarations, statements)
: EOF
;

Expressions

Precedence	Title	Symbols
Highest	Postfix	`++`, `--`, `.`, `?.`, `?`
	Prefix	`-`, `+`, `++`, `--`, `!`, `label`
	Type RHS	`:`, `as`, `as?`
	Multiplicative	`*`, `/`, `%`
	Additive	`+`, `-`
	Range	`..`
	Infix function	`simpleIdentifier`
	Elvis	`?:`
	Named checks	`in`, `!in`, `is`, `!is`
	Comparison	`<`, `>`, `<=`, `>=`
	Equality	`==`, `!==`
	Conjunction	`&&`
	Disjunction	`||`
	Spread operator	`*`
Lowest	Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`

expression

(used by classParameter, explicitDelegation, functionValueParameter, functionBody, propertyDeclaration, propertyDelegate, statement, forStatement, whileStatement, doWhileStatement, assignment, indexingSuffix, valueArgument, parenthesizedExpression, collectionLiteral, lineStringExpression, multiLineStringExpression, ifExpression, whenSubject, whenCondition, rangeTest, jumpExpression)
 : disjunction
 ;
disjunction
(used by expression)
 : conjunction ('||' conjunction)*
 ;
conjunction
(used by disjunction)
 : equality ('&&' equality)*
 ;
equality
(used by conjunction)
 : comparison (equalityOperator comparison)*
 ;
comparison
(used by equality)
 : infixOperation (comparisonOperator infixOperation)?
 ;
infixOperation
(used by comparison)
 : elvisExpression ((inOperator elvisExpression) | (isOperator type))*
 ;
elvisExpression
(used by infixOperation)
 : infixFunctionCall (elvis infixFunctionCall)*
 ;
elvis
(used by elvisExpression)
 : '?' ':'
 ;
infixFunctionCall
(used by elvisExpression)
 : rangeExpression (simpleIdentifier rangeExpression)*
 ;
rangeExpression
(used by infixFunctionCall)
 : additiveExpression ('..' additiveExpression)*
 ;
additiveExpression
(used by rangeExpression)
 : multiplicativeExpression (additiveOperator multiplicativeExpression)*
 ;
multiplicativeExpression
(used by additiveExpression)
 : asExpression (multiplicativeOperator asExpression)*
 ;
asExpression
(used by multiplicativeExpression)
 : prefixUnaryExpression (asOperator type)?
 ;
prefixUnaryExpression
(used by asExpression, assignableExpression)
 : unaryPrefix* postfixUnaryExpression
 ;
unaryPrefix
(used by prefixUnaryExpression)
 : annotation
 | label
 | prefixUnaryOperator
 ;
postfixUnaryExpression
(used by prefixUnaryExpression, directlyAssignableExpression)
 : primaryExpression
 | primaryExpression postfixUnarySuffix+
 ;
postfixUnarySuffix
(used by postfixUnaryExpression)
 : postfixUnaryOperator
 | typeArguments
 | callSuffix
 | indexingSuffix
 | navigationSuffix
 ;
directlyAssignableExpression
(used by assignment)
 : postfixUnaryExpression assignableSuffix
 | simpleIdentifier
 ;
assignableExpression
(used by assignment)
 : prefixUnaryExpression
 ;
assignableSuffix
(used by directlyAssignableExpression)
 : typeArguments

```
 | indexingSuffix
 | navigationSuffix
 ;
indexingSuffix
(used by postfixUnarySuffix, assignableSuffix)
 : '[' expression (',' expression)* ']'
 ;
navigationSuffix
(used by postfixUnarySuffix, assignableSuffix)
 : memberAccessOperator (simpleIdentifier | parenthesizedExpression | 'class')
 ;
callSuffix
(used by postfixUnarySuffix)
 : typeArguments? valueArguments? annotatedLambda
 | typeArguments? valueArguments
 ;
annotatedLambda
(used by callSuffix)
 : annotation* label? lambdaLiteral
 ;
typeArguments
(used by simpleUserType, postfixUnarySuffix, assignableSuffix, callSuffix)
 : '<' typeProjection (',' typeProjection)* '>'
 ;
valueArguments
(used by constructorInvocation, constructorDelegationCall, enumEntry, callSuffix)
 : '(' ')'
 | '(' valueArgument (',' valueArgument)* ')'
 ;
valueArgument
(used by valueArguments)
 : annotation? (simpleIdentifier '=')? '*'? expression
 ;
primaryExpression
(used by postfixUnaryExpression)
 : parenthesizedExpression
 | simpleIdentifier
 | literalConstant
 | stringLiteral
 | callableReference
 | functionLiteral
 | objectLiteral
 | collectionLiteral
 | thisExpression
 | superExpression
 | ifExpression
 | whenExpression
 | tryExpression
 | jumpExpression
 ;
parenthesizedExpression
(used by navigationSuffix, primaryExpression)
 : '(' expression ')'
 ;
collectionLiteral
(used by primaryExpression)
 : '[' expression (',' expression)* ']'
 | '[' ']'
 ;
literalConstant
(used by primaryExpression)
 : BooleanLiteral
 | IntegerLiteral
 | HexLiteral
 | BinLiteral
 | CharacterLiteral
 | RealLiteral
 | 'null'
 | LongLiteral
 | UnsignedLiteral
 ;
stringLiteral
(used by primaryExpression)
 : lineStringLiteral
 | multiLineStringLiteral
 ;
lineStringLiteral
(used by stringLiteral)
 : '"' (lineStringContent | lineStringExpression)* '"'
 ;
multiLineStringLiteral
(used by stringLiteral)
 : '"""' (multiLineStringContent | multiLineStringExpression | '"')*
   TRIPLE_QUOTE_CLOSE
 ;
lineStringContent
(used by lineStringLiteral)
 : LineStrText
 | LineStrEscapedChar
```

```
 | LineStrRef
 ;
lineStringExpression
(used by lineStringLiteral)
 : '${' expression '}'
 ;
multiLineStringContent
(used by multiLineStringLiteral)
 : MultiLineStrText
 | '"'
 | MultiLineStrRef
 ;
multiLineStringExpression
(used by multiLineStringLiteral)
 : '${' expression '}'
 ;
lambdaLiteral
(used by annotatedLambda, functionLiteral)
 : '{' statements '}'
 | '{' lambdaParameters? '->' statements '}'
 ;
lambdaParameters
(used by lambdaLiteral)
 : lambdaParameter (',' lambdaParameter)*
 ;
lambdaParameter
(used by lambdaParameters)
 : variableDeclaration
 | multiVariableDeclaration (':' type)?
 ;
anonymousFunction
(used by functionLiteral)
 : 'fun' (type '.')? functionValueParameters
   (':' type)? typeConstraints?
   functionBody?
 ;
functionLiteral
(used by primaryExpression)
 : lambdaLiteral
 | anonymousFunction
 ;
objectLiteral
(used by primaryExpression)
 : 'object' ':' delegationSpecifiers classBody
 | 'object' classBody
 ;
thisExpression
(used by primaryExpression)
 : 'this'
 | THIS_AT
 ;
superExpression
(used by primaryExpression)
 : 'super' ('<' type '>')? ('@' simpleIdentifier)?
 | SUPER_AT
 ;
ifExpression
(used by primaryExpression)
 : 'if' '(' expression ')'
   (controlStructureBody | ';')
 | 'if' '(' expression ')'
   controlStructureBody? ';'? 'else' (controlStructureBody | ';')
 ;
whenSubject
(used by whenExpression)
 : '(' (annotation* 'val' variableDeclaration '=')? expression ')'
 ;
whenExpression
(used by primaryExpression)
 : 'when' whenSubject? '{' whenEntry* '}'
 ;
whenEntry
(used by whenExpression)
 : whenCondition (',' whenCondition)* '->' controlStructureBody semi?
 | 'else' '->' controlStructureBody semi?
 ;
whenCondition
(used by whenEntry)
 : expression
 | rangeTest
 | typeTest
 ;
rangeTest
(used by whenCondition)
 : inOperator expression
 ;
typeTest
(used by whenCondition)
 : isOperator type
```

```
 ;
tryExpression
(used by primaryExpression)
 : 'try' block ((catchBlock+ finallyBlock?) | finallyBlock)
 ;
catchBlock
(used by tryExpression)
 : 'catch' '(' annotation* simpleIdentifier ':' type ')' block
 ;
finallyBlock
(used by tryExpression)
 : 'finally' block
 ;
jumpExpression
(used by primaryExpression)
 : 'throw' expression
 | ('return' | RETURN_AT) expression?
 | 'continue'
 | CONTINUE_AT
 | 'break'
 | BREAK_AT
 ;
callableReference
(used by primaryExpression)
 : (receiverType? '::' (simpleIdentifier | 'class'))
 ;
assignmentAndOperator
(used by assignment)
 : '+='
 | '-='
 | '*='
 | '/='
 | '%='
 ;
equalityOperator
(used by equality)
 : '!='
 | '!=='
 | '=='
 | '==='
 ;
comparisonOperator
(used by comparison)
 : '<'
 | '>'
 | '<='
 | '>='
 ;
inOperator
(used by infixOperation, rangeTest)
 : 'in'
 | NOT_IN
 ;
isOperator
(used by infixOperation, typeTest)
 : 'is'
 | NOT_IS
 ;
additiveOperator
(used by additiveExpression)
 : '+'
 | '-'
 ;
multiplicativeOperator
(used by multiplicativeExpression)
 : '*'
 | '/'
 | '%'
 ;
asOperator
(used by asExpression)
 : 'as'
 | 'as?'
 ;
prefixUnaryOperator
(used by unaryPrefix)
 : '++'
 | '--'
 | '-'
 | '+'
 | excl
 ;
postfixUnaryOperator
(used by postfixUnarySuffix)
 : '++'
 | '--'
 | '!' excl
 ;
excl
```

(used by prefixUnaryOperator, postfixUnaryOperator)
 : '!'
 | EXCL_WS
 ;
memberAccessOperator
(used by navigationSuffix)
 : '.'
 | safeNav
 | '::'
 ;
safeNav
(used by memberAccessOperator)
 : '?' '.'
 ;

Modifiers

modifiers
(used by typeAlias, classDeclaration, primaryConstructor, classParameter, companionObject, functionValueParameter, functionDeclaration, propertyDeclaration, getter, setter, objectDeclaration, secondaryConstructor, enumEntry)
 : annotation
 | modifier+
 ;
modifier
(used by modifiers)
 : classModifier
 | memberModifier
 | visibilityModifier
 | functionModifier
 | propertyModifier
 | inheritanceModifier
 | parameterModifier
 | platformModifier
 ;
typeModifiers
(used by type, receiverType)
 : typeModifier+
 ;
typeModifier
(used by typeModifiers)
 : annotation
 | 'suspend'
 ;
classModifier
(used by modifier)
 : 'enum'
 | 'sealed'
 | 'annotation'
 | 'data'
 | 'inner'
 ;
memberModifier
(used by modifier)
 : 'override'
 | 'lateinit'
 ;
visibilityModifier
(used by modifier)
 : 'public'
 | 'private'
 | 'internal'
 | 'protected'
 ;
varianceModifier
(used by typeProjectionModifier, typeParameterModifier)
 : 'in'
 | 'out'
 ;
typeParameterModifiers
(used by typeParameter)
 : typeParameterModifier+
 ;
typeParameterModifier
(used by typeParameterModifiers)
 : reificationModifier
 | varianceModifier
 | annotation
 ;
functionModifier
(used by modifier)
 : 'tailrec'
 | 'operator'
 | 'infix'
 | 'inline'
 | 'external'
 | 'suspend'
 ;
propertyModifier

(used by modifier)
 : 'const'
 ;
inheritanceModifier
(used by modifier)
 : 'abstract'
 | 'final'
 | 'open'
 ;
parameterModifier
(used by setter, modifier)
 : 'vararg'
 | 'noinline'
 | 'crossinline'
 ;
reificationModifier
(used by typeParameterModifier)
 : 'reified'
 ;
platformModifier
(used by modifier)
 : 'expect'
 | 'actual'
 ;

Annotations

annotation
(used by annotatedDelegationSpecifier, typeConstraint, variableDeclaration, setter, typeProjectionModifier, statement, forStatement, unaryPrefix, annotatedLambda, valueArgument, whenSubject, catchBlock, modifiers, typeModifier, typeParameterModifier)
 : singleAnnotation
 | multiAnnotation
 ;
singleAnnotation
(used by annotation)
 : annotationUseSiteTarget unescapedAnnotation
 | '@' unescapedAnnotation
 ;
multiAnnotation
(used by annotation)
 : annotationUseSiteTarget '[' unescapedAnnotation+ ']'
 | '@' '[' unescapedAnnotation+ ']'
 ;
annotationUseSiteTarget
(used by singleAnnotation, multiAnnotation)
 : ANNOTATION_USE_SITE_TARGET_FIELD
 | ANNOTATION_USE_SITE_TARGET_PROPERTY
 | ANNOTATION_USE_SITE_TARGET_GET
 | ANNOTATION_USE_SITE_TARGET_SET
 | ANNOTATION_USE_SITE_TARGET_RECEIVER
 | ANNOTATION_USE_SITE_TARGET_PARAM
 | ANNOTATION_USE_SITE_TARGET_SETPARAM
 | ANNOTATION_USE_SITE_TARGET_DELEGATE
 ;
unescapedAnnotation
(used by fileAnnotation, singleAnnotation, multiAnnotation)
 : constructorInvocation
 | userType
 ;

Identifiers

simpleIdentifier
(used by importAlias, typeAlias, classDeclaration, classParameter, typeParameter, typeConstraint, companionObject, functionDeclaration, variableDeclaration, setterParameter, parameter, objectDeclaration, enumEntry, simpleUserType, infixFunctionCall, directlyAssignableExpression, navigationSuffix, valueArgument, primaryExpression, superExpression, catchBlock, callableReference, identifier)
 : Identifier
 | 'abstract'
 | 'annotation'
 | 'by'
 | 'catch'
 | 'companion'
 | 'constructor'
 | 'crossinline'
 | 'data'
 | 'dynamic'
 | 'enum'
 | 'external'
 | 'final'
 | 'finally'
 | 'get'
 | 'import'
 | 'infix'
 | 'init'
 | 'inline'

| 'inner'
| 'internal'
| 'lateinit'
| 'noinline'
| 'open'
| 'operator'
| 'out'
| 'override'
| 'private'
| 'protected'
| 'public'
| 'reified'
| 'sealed'
| 'tailrec'
| 'set'
| 'vararg'
| 'where'
| 'expect'
| 'actual'
| 'const'
| 'suspend'
;
identifier
(used by packageHeader, importHeader)
: simpleIdentifier ('.' simpleIdentifier)*
;

Lexical grammar

General

ShebangLine
(used by shebangLine)
: '#!' ~[\r\n]*
;
DelimitedComment
(used by DelimitedComment, Hidden)
: ('/*' (DelimitedComment | .)*? '*/')
;
LineComment
(used by Hidden)
: ('//' ~[\r\n]*)
;
WS
(used by Hidden)
: [\u0020\u0009\u000C]
;
helper
Hidden
(used by EXCL_WS, QUEST_WS, NOT_IS, NOT_IN)
: DelimitedComment
| LineComment
| WS
;

Separators and operations

RESERVED
: '...'
;
EXCL_WS
(used by excl)
: '!' Hidden
;
DOUBLE_ARROW
: '=>'
;
DOUBLE_SEMICOLON
: ';;'
;
HASH
: '#'
;
QUEST_WS
(used by quest)
: '?' Hidden
;
SINGLE_QUOTE
: '\''
;

Keywords

```
RETURN_AT
(used by jumpExpression)
 : 'return@' Identifier
 ;
CONTINUE_AT
(used by jumpExpression)
 : 'continue@' Identifier
 ;
BREAK_AT
(used by jumpExpression)
 : 'break@' Identifier
 ;
THIS_AT
(used by thisExpression)
 : 'this@' Identifier
 ;
SUPER_AT
(used by superExpression)
 : 'super@' Identifier
 ;
ANNOTATION_USE_SITE_TARGET_FILE
(used by fileAnnotation)
 : '@file' ':'
 ;
ANNOTATION_USE_SITE_TARGET_FIELD
(used by annotationUseSiteTarget)
 : '@field' ':'
 ;
ANNOTATION_USE_SITE_TARGET_PROPERTY
(used by annotationUseSiteTarget)
 : '@property' ':'
 ;
ANNOTATION_USE_SITE_TARGET_GET
(used by annotationUseSiteTarget)
 : '@get' ':'
 ;
ANNOTATION_USE_SITE_TARGET_SET
(used by annotationUseSiteTarget)
 : '@set' ':'
 ;
ANNOTATION_USE_SITE_TARGET_RECEIVER
(used by annotationUseSiteTarget)
 : '@receiver' ':'
 ;
ANNOTATION_USE_SITE_TARGET_PARAM
(used by annotationUseSiteTarget)
 : '@param' ':'
 ;
ANNOTATION_USE_SITE_TARGET_SETPARAM
(used by annotationUseSiteTarget)
 : '@setparam' ':'
 ;
ANNOTATION_USE_SITE_TARGET_DELEGATE
(used by annotationUseSiteTarget)
 : '@delegate' ':'
 ;
TYPEOF
 : 'typeof'
 ;
NOT_IS
(used by isOperator)
 : '!is' Hidden
 ;
NOT_IN
(used by inOperator)
 : '!in' Hidden
 ;
```

Literals

```
helper
DecDigit
(used by DecDigitOrSeparator, DecDigits, IntegerLiteral)
 : '0'..'9'
 ;
helper
DecDigitNoZero
(used by IntegerLiteral)
 : '1'..'9'
 ;
helper
DecDigitOrSeparator
(used by DecDigits, IntegerLiteral)
 : DecDigit
 | '_'
 ;
helper
DecDigits
```

(used by DoubleExponent, FloatLiteral, DoubleLiteral)
 : DecDigit DecDigitOrSeparator* DecDigit
 | DecDigit
 ;
helper
DoubleExponent
(used by DoubleLiteral)
 : [eE] [+-]? DecDigits
 ;
RealLiteral
(used by literalConstant)
 : FloatLiteral
 | DoubleLiteral
 ;
FloatLiteral
(used by RealLiteral)
 : DoubleLiteral [fF]
 | DecDigits [fF]
 ;
DoubleLiteral
(used by RealLiteral, FloatLiteral)
 : DecDigits? '.' DecDigits DoubleExponent?
 | DecDigits DoubleExponent
 ;
IntegerLiteral
(used by literalConstant, UnsignedLiteral, LongLiteral)
 : DecDigitNoZero DecDigitOrSeparator* DecDigit
 | DecDigit
 ;
helper
HexDigit
(used by HexDigitOrSeparator, HexLiteral, UniCharacterLiteral)
 : [0-9a-fA-F]
 ;
helper
HexDigitOrSeparator
(used by HexLiteral)
 : HexDigit
 | '_'
 ;
HexLiteral
(used by literalConstant, UnsignedLiteral, LongLiteral)
 : '0' [xX] HexDigit HexDigitOrSeparator* HexDigit
 | '0' [xX] HexDigit
 ;
helper
BinDigit
(used by BinDigitOrSeparator, BinLiteral)
 : [01]
 ;
helper
BinDigitOrSeparator
(used by BinLiteral)
 : BinDigit
 | '_'
 ;
BinLiteral
(used by literalConstant, UnsignedLiteral, LongLiteral)
 : '0' [bB] BinDigit BinDigitOrSeparator* BinDigit
 | '0' [bB] BinDigit
 ;
UnsignedLiteral
(used by literalConstant)
 : (IntegerLiteral | HexLiteral | BinLiteral) [uU] 'L'?
 ;
LongLiteral
(used by literalConstant)
 : (IntegerLiteral | HexLiteral | BinLiteral) 'L'
 ;
BooleanLiteral
(used by literalConstant)
 : 'true'
 | 'false'
 ;
CharacterLiteral
(used by literalConstant)
 : '\'' (EscapeSeq | ~[\n\r'\\]) '\''
 ;

Identifiers

helper
UnicodeDigit
(used by Identifier)
 : UNICODE_CLASS_ND
 ;
Identifier
(used by simpleIdentifier, RETURN_AT, CONTINUE_AT, BREAK_AT, THIS_AT, SUPER_AT, IdentifierOrSoftKey)

```
 : (Letter | '_') (Letter | '_' | UnicodeDigit)*
 | '`' ~([\r\n] | '`' | '.' | ';' | ':' | '\\' | '/' | '[' | ']' | '<' | '>')+ '`'
 ;
```

Depending on the target and publicity of the declaration, the set of allowed symbols in identifiers is different. This rule contains the union of allowed symbols from all targets. Thus, the code for any target can be parsed using the grammar.

The allowed symbols in identifiers corresponding to the target and publicity of the declaration are given below.

```
Kotlin/JVM (any declaration publicity)
~ ( [\r\n] | '`' | '.' | ';' | ':' | '\' | '/' | '[' | ']' | '<' | '>' )
```

```
Kotlin/Android (any declaration publicity)
```

The allowed symbols are different from allowed symbols for Kotlin/JVM and correspond to the Dalvik Executable format.

```
Kotlin/JS (private declarations)
~ ( [\r\n] | '`' )
```

```
Kotlin/JS (public declarations)
```

The allowed symbols for public declarations correspond to the ECMA specification (section 7.6) except that ECMA reserved words is allowed.

```
Kotlin/Native (any declaration publicity)
~ ( [\r\n] | '`' )
```

```
IdentifierOrSoftKey
(used by IdentifierAt, FieldIdentifier)
 : Identifier
 | 'abstract'
 | 'annotation'
 | 'by'
 | 'catch'
 | 'companion'
 | 'constructor'
 | 'crossinline'
 | 'data'
 | 'dynamic'
 | 'enum'
 | 'external'
 | 'final'
 | 'finally'
 | 'get'
 | 'import'
 | 'infix'
 | 'init'
 | 'inline'
 | 'inner'
 | 'internal'
 | 'lateinit'
 | 'noinline'
 | 'open'
 | 'operator'
 | 'out'
 | 'override'
 | 'private'
 | 'protected'
 | 'public'
 | 'reified'
 | 'sealed'
 | 'tailrec'
 | 'set'
 | 'vararg'
 | 'where'
 | 'expect'
 | 'actual'
 | 'const'
 | 'suspend'
 ;
IdentifierAt
(used by label)
 : IdentifierOrSoftKey '@'
 ;
FieldIdentifier
(used by LineStrRef, MultiLineStrRef)
 : '$' IdentifierOrSoftKey
 ;
helper
UniCharacterLiteral
(used by EscapeSeq, LineStrEscapedChar)
 : '\\' 'u' HexDigit HexDigit HexDigit HexDigit
 ;
```

```
helper
EscapedIdentifier
(used by EscapeSeq, LineStrEscapedChar)
 : '\\' ('t' | 'b' | 'r' | 'n' | '\'' | '"' | '\\' | '$')
 ;
helper
EscapeSeq
(used by CharacterLiteral)
 : UniCharacterLiteral
 | EscapedIdentifier
 ;
```

Characters

```
helper
Letter
(used by Identifier)
 : UNICODE_CLASS_LL
 | UNICODE_CLASS_LM
 | UNICODE_CLASS_LO
 | UNICODE_CLASS_LT
 | UNICODE_CLASS_LU
 | UNICODE_CLASS_NL
 ;
```

Strings

```
LineStrRef
(used by lineStringContent)
 : FieldIdentifier
 ;
```

See String templates

```
LineStrText
(used by lineStringContent)
 : ~('\\' | '"' | '$')+
 | '$'
 ;
LineStrEscapedChar
(used by lineStringContent)
 : EscapedIdentifier
 | UniCharacterLiteral
 ;
TRIPLE_QUOTE_CLOSE
(used by multiLineStringLiteral)
 : ('"'? '"""')
 ;
MultiLineStrRef
(used by multiLineStringContent)
 : FieldIdentifier
 ;
MultiLineStrText
(used by multiLineStringContent)
 : ~('"' | '$')+
 | '$'
 ;
ErrorCharacter
 : .
 ;
```

代码风格迁移指南

Kotlin 编码规范与 IntelliJ IDEA 格式化程序

Kotlin 编码规范影响了编写地道 Kotlin 代码的几个方面，其中包括一组旨在提高 Kotlin 代码可读性的格式建议。

遗憾的是，IntelliJ IDEA 中内置的代码格式化工具在这篇文档发布很久之前就已经在使用了，并且现在具有默认设置，该默认设置产生的格式不同于现在建议格式。

接下来，通过改变 IntelliJ IDEA 中的默认设置并使格式与 Kotlin 编码规范一致来消除这种隔阂似乎是符合逻辑的。但这意味着所有现有 Kotlin 项目将在安装 Kotlin 插件后启用新的代码格式。这并不是插件更新的预期结果，对不对？

这就是为什么我们有以下迁移计划的原因：

- 从 Kotlin 1.3 开始，默认情况下启用官方代码格式，并且仅对新项目启用（旧风格可以手动启用）
- 现有项目的作者可以选择迁移到现有的 Kotlin 编码规范
- 现有项目的作者可以选择进行显式声明在项目中使用旧代码风格（这样，将来不会因改变默认值而影响项目）
- 切换到默认格式，使其与 Kotlin 1.4 中的 Kotlin 编码规范一致

“Kotlin 编码规范”与“IntelliJ IDEA 默认代码风格”之间的差异

最显着的变化是延续缩进策略。使用双缩进来显示多行表达式尚未在前一行结束是一个好主意。这是一个非常简单且通用的规则，但是以这种方式格式化时，一些 Kotlin 构造看起来有些尴尬。在 Kotlin 编码规范中，建议在之前强制使用长延续缩进的场景中使用单个缩进

实际上，很多代码都会受到影响，因此可以将其视为重大的代码风格更新。

迁移到新的代码风格讨论

如果没有使用旧风格的代码，那么从新项目开始就采用新的代码风格应该是很自然的过程。因此，从 1.3 版开始，Kotlin IntelliJ 插件使用默认情况下启用的编码规范文档中的格式创建新项目。

在现有项目中更改格式是一项更加艰巨的任务，应该与团队讨论所有注意事项然后一起开始。

更改现有项目中的代码风格的主要缺点是，blame/annotate 版本控制系统特性将更频繁地指向无关的提交。尽管每种版本控制系统都有某种方式可以解决此问题（IntelliJ IDEA 中可以使用“Annotate Previous Revision”），但重要的是确定新风格是否值得所有努力。将修改格式的提交与有意义的更改分开的做法可以为以后的调查提供很大帮助。

对于大型团队来说，迁移也可能会比较困难，因为在多个子系统中提交大量文件可能会在个人的分支中产生合并冲突。尽管每个冲突解决方案通常都很琐碎，但明智的做法是知道当前是否正在使用大型功能分支。

通常，对于小型项目，建议一次转换所有文件。

对于大中型项目，决定可能会很艰难。如果还没有准备好立即更新许多文件，则可以决定逐模块迁移，或者继续只对已修改文件逐步迁移。

迁移到新的代码风格

可以在 `Settings → Editor → Code Style → Kotlin` 对话框中切换 Kotlin 代码风格。将 Scheme 切换到 *Project* 并从下方选择 `Set from... → Predefined Style → Kotlin Style Guide`。

为了向所有项目开发人员共享这些更改，必须将 `.idea/codeStyle` 文件夹提交给版本控制系统。

如果使用外部构建系统来配置项目，并且已决定不共享 `.idea/codeStyle` 文件夹，那么可以通过附加属性强制使用 Kotlin 编码规范：

在 Gradle 中

在项目根目录的 **gradle.properties** 文件中添加 **kotlin.code.style=official** 属性，并将其提交到版本控制系统。

在 Maven 中

在项目根目录的 **pom.xml** 文件中添加 **kotlin.code.style official** 属性。

```
<properties>
  <kotlin.code.style>official</kotlin.code.style>
</properties>
```

警告： 设置 **kotlin.code.style** 选项可能会在项目导入期间修改代码风格方案，并且可能会更改代码风格设置。

更新代码风格设置后，在所需范围的项目视图中选择“Reformat Code”。

对于逐步迁移，可以启用 *“File is not formatted according to project settings”（文件未根据项目设置格式化）* 探查项。这将突出显示应修改格式的地方。启用 *“Apply only to modified files”（仅应用于修改后的文件）* 选项后，检查将仅在修改后的文件中显示格式问题。无论如何，此类文件应该尽快修改并提交。

在项目中存储旧的代码风格

随时可以将 IntelliJ IDEA 代码风格明确设置为项目的正确代码风格。为此，请在 `Settings → Editor → Code Style → Kotlin` 中将 Scheme 切换到 *Project*，然后在 *Load/Save* 选项卡中的 *"Use defaults from:"* 中选择 *"Kotlin obsolete IntelliJ IDEA codestyle"*。

为了在项目开发人员的 `.idea/codeStyle` 文件夹中共享更改，必须将其提交给版本控制系统。另外，**kotlin.code.style=obsolete** 可以用于配置了 Gradle 或 Maven 的项目。

Java 互操作

在 Kotlin 中调用 Java 代码

Kotlin 在设计时就考虑了 Java 互操作性。可以从 Kotlin 中自然地调用现存的 Java 代码，并且在 Java 代码中也可以很顺利地调用 Kotlin 代码。在本节中我们会介绍从 Kotlin 中调用 Java 代码的一些细节。

几乎所有 Java 代码都可以使用而没有任何问题：

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // “for”-循环用于 Java 集合：
    for (item in source) {
        list.add(item)
    }
    // 操作符约定同样有效：
    for (i in 0..source.size - 1) {
        list[i] = source[i] // 调用 get 和 set
    }
}
```

Getter 和 Setter

遵循 Java 约定的 getter 和 setter 的方法（名称以 `get` 开头的无参数方法和以 `set` 开头的单参数方法）在 Kotlin 中表示为属性。`Boolean` 访问器方法（其中 getter 的名称以 `is` 开头而 setter 的名称以 `set` 开头）会表示为与 getter 方法具有相同名称的属性。例如：

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) {  // 调用 getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY      // 调用ll setFirstDayOfWeek()
    }
    if (!calendar.isLenient) {                         // 调用 isLenient()
        calendar.isLenient = true                      // 调用 setLenient()
    }
}
```

请注意，如果 Java 类只有一个 setter，它在 Kotlin 中不会作为属性可见，因为 Kotlin 目前不支持只写（set-only）属性。

返回 void 的方法

如果一个 Java 方法返回 void，那么从 Kotlin 调用时中返回 `Unit`。万一有人使用其返回值，它将由 Kotlin 编译器在调用处赋值，因为该值本身是预先知道的（是 `Unit`）。

将 Kotlin 中是关键字的 Java 标识符进行转义

一些 Kotlin 关键字在 Java 中是有效标识符：`in`、`object`、`is` 等等。如果一个 Java 库使用了 Kotlin 关键字作为方法，你仍然可以通过反引号（`）字符转义它来调用该方法：

```
foo.`is`(bar)
```

空安全与平台类型

Java 中的任何引用都可能是 `null`，这使得 Kotlin 对来自 Java 的对象要求严格空安全是不现实的。Java 声明的类型在 Kotlin 中会被特别对待并称为*平台类型*。对这种类型的空检测会放宽，因此它们的安全保证与在 Java 中相同（更多请参见下文）。

考虑以下示例：

```
val list = ArrayList<String>() // 非空（构造函数结果）
list.add("Item")
val size = list.size // 非空（原生 int）
val item = list[0] // 推断为平台类型（普通 Java 对象）
```

当我们调用平台类型变量的方法时，Kotlin 不会在编译时报告可空性错误，但在运行时调用可能会失败，因为空指针异常或者 Kotlin 生成的阻止空值传播的断言：

```
item.substring(1) // 允许，如果 item == null 可能会抛出异常
```

平台类型是*不可标示*的，意味着不能在语言中明确地写下它们。当把一个平台值赋值给一个 Kotlin 变量时，可以依赖类型推断（该变量会具有推断出的的平台类型，如上例中 `item` 所具有的类型），或者我们可以选择我们期望的类型（可空或非空类型均可）：

```
val nullable: String? = item // 允许，没有问题
val notNull: String = item // 允许，运行时可能失败
```

如果我们选择非空类型，编译器会在赋值时触发一个断言。这防止 Kotlin 的非空变量保存空值。当我们把平台值传递给期待非空值等的 Kotlin 函数时，也会触发断言。总的来说，编译器尽力阻止空值通过程序向远传播（尽管鉴于泛型的原因，有时这不可能完全消除）。

平台类型表示法

如上所述，平台类型不能在程序中显式表述，因此在语言中没有相应语法。然而，编译器和 IDE 有时需要（在错误信息中、参数信息中等）显示他们，所以我们用一个助记符来表示他们：

— `T!` 表示“`T` 或者 `T?`”，

— `(Mutable)Collection<T>!` 表示“可以可变或不可变、可空或不可空的 `T` 的 Java 集合”，

— `Array<(out) T>!` 表示“可空或者不可空的 `T`（或 `T` 的子类型）的 Java 数组”

可空性注解

具有可空性注解的Java类型并不表示为平台类型，而是表示为实际可空或非空的 Kotlin 类型。编译器支持多种可空性注解，包括：

— JetBrains（`org.jetbrains.annotations` 包中的 `@Nullable` 和 `@NotNull`）

— Android（`com.android.annotations` 和 `android.support.annotations`）

— JSR-305（`javax.annotation`，详见下文）

— FindBugs（`edu.umd.cs.findbugs.annotations`）

— Eclipse（`org.eclipse.jdt.annotation`）

— Lombok（`lombok.NonNull`）。

你可以在 Kotlin 编译器源代码中找到完整的列表。

注解类型参数

可以标注泛型类型的类型参数，以便同时为其提供可空性信息。例如，考虑这些 Java 声明的注解：

```
@NotNull
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) { …… }
```

在 Kotlin 中可见的是以下签名：

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { …… }
```

请注意 `String` 类型参数上的 `@NotNull` 注解。如果没有的话，类型参数会是平台类型：

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { …… }
```

标注类型参数适用于面向 Java 8 或更高版本环境，并且要求可空性注解支持 `TYPE_USE` 目标（`org.jetbrains.annotations` 15 或以上版本支持）。

> 注：由于当前的技术限制，IDE 无法正确识别用作依赖的已编译 Java 库中类型参数上的这些注解。

JSR-305 支持

已支持 JSR-305 中定义的 `@Nonnull` 注解来表示 Java 类型的可空性。

如果 `@Nonnull(when = ...)` 值为 `When.ALWAYS`，那么该注解类型会被视为非空；`When.MAYBE` 与 `When.NEVER` 表示可空类型；而 `When.UNKNOWN` 强制类型为平台类型。

可针对 JSR-305 注解编译库，但不需要为库的消费者将注解构件（如 `jsr305.jar`）指定为编译依赖。Kotlin 编译器可以从库中读取 JSR-305 注解，并不需要该注解出现在类路径中。

自 Kotlin 1.1.50 起，也支持自定义可空限定符（KEEP-79）（见下文）。

类型限定符别称（自 1.1.50 起）

如果一个注解类型同时标注有 `@TypeQualifierNickname` 与 JSR-305 `@Nonnull`（或者它的其他别称，如 `@CheckForNull`），那么该注解类型自身将用于 检索精确的可空性，且具有与该可空性注解相同的含义：

```java
@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
@CheckForNull // 另一个类型限定符别称的别称
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@MyNonnull String x);
    // 在 Kotlin (严格模式) 中: `fun foo(x: String): String?`

    String bar(List<@MyNonnull String> x);
    // 在 Kotlin (严格模式) 中: `fun bar(x: List<String>!): String!`
}
```

类型限定符默认值(自 1.1.50 起)

`@TypeQualifierDefault` 引入应用时在所标注元素的作用域内定义默认可空性的注解。

这些注解类型应自身同时标注有 `@Nonnull`(或其别称)与 `@TypeQualifierDefault(...)` 注解,后者带有一到多个 `ElementType` 值:

- `ElementType.METHOD` 用于方法的返回值;
- `ElementType.PARAMETER` 用于值参数;
- `ElementType.FIELD` 用于字段;以及
- `ElementType.TYPE_USE`(自 1.1.60 起)适用于任何类型,包括类型参数、类型参数的上界与通配符类型。

当类型并未标注可空性注解时使用默认可空性,并且该默认值是由最内层标注有带有与所用类型相匹配的 `ElementType` 的类型限定符默认注解的元素确定。

```java
@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); // fun foo(x: String?): String?

    @NotNullApi // 覆盖来自接口的默认值
    String bar(String x, @Nullable String y); // fun bar(x: String, y: String?): String

    // 由于 `@NullableApi` 具有 `TYPE_USE` 元素类型,
    // 因此认为 List<String> 类型参数是可空的:
    String baz(List<String> x); // fun baz(List<String?>?): String?

    // “x”参数仍然是平台类型,因为有显式
    // UNKNOWN 标记的可空性注解:
    String qux(@Nonnull(when = When.UNKNOWN) String x); // fun baz(x: String!): String?
}
```

> 注意：本例中的类型只在启用了严格模式时出现，否则仍是平台类型。参见 @UnderMigration 注解与编译器配置两节。

也支持包级的默认可空性：

```
// 文件：test/package-info.java
@NonNullApi // 默认将“test”包中所有类型声明为不可空
package test;
```

@UnderMigration 注解（自 1.1.60 起）

库的维护者可以使用 `@UnderMigration` 注解（在单独的构件 `kotlin-annotations-jvm` 中提供）来定义可为空性类型限定符的迁移状态。

`@UnderMigration(status = ...)` 中的状态值指定了编译器如何处理 Kotlin 中注解类型的不当用法（例如，使用 `@MyNullable` 标注的类型值作为非空值）：

- `MigrationStatus.STRICT` 使注解像任何纯可空性注解一样工作，即对不当用法报错并影响注解声明内的类型在 Kotlin 中的呈现；
- 对于 `MigrationStatus.WARN`，不当用法报为警告而不是错误；但注解声明内的类型仍是平台类型；而
- `MigrationStatus.IGNORE` 则使编译器完全忽略可空性注解。

库的维护者还可以将 `@UnderMigration` 状态添加到类型限定符别称与类型限定符默认值：

```
@Nonnull(when = When.ALWAYS)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
@UnderMigration(status = MigrationStatus.WARN)
public @interface NonNullApi {
}

// 类中的类型是非空的，但是只报警告
// 因为 `@NonNullApi` 标注了 `@UnderMigration(status = MigrationStatus.WARN)`
@NonNullApi
public class Test {}
```

注意：可空性注解的迁移状态并不会从其类型限定符别称继承，而是适用于默认类型限定符的用法。

如果默认类型限定符使用类型限定符别称，并且它们都标注有 `@UnderMigration`，那么使用默认类型限定符的状态。

编译器配置

可以通过添加带有以下选项的 `-Xjsr305` 编译器标志来配置 JSR-305 检测：

- `-Xjsr305={strict|warn|ignore}` 设置非 `@UnderMigration` 注解的行为。自定义的可空性限定符，尤其是 `@TypeQualifierDefault` 已经在很多知名库中流传，而用户更新到包含 JSR-305 支持的 Kotlin 版本时可能需要平滑迁移。自 Kotlin 1.1.60 起，这一标志只影响非 `@UnderMigration` 注解。
- `-Xjsr305=under-migration:{strict|warn|ignore}`（自 1.1.60 起）覆盖 `@UnderMigration` 注解的行为。用户可能对库的迁移状态有不同的看法：他们可能希望在官方迁移状态为 `WARN` 时报错误，反之亦然，他们可能希望推迟错误报告直到他们完成迁移。

— `-Xjsr305=@<fq.name>:{strict|warn|ignore}`（自 1.1.60 起）覆盖单个注解的行为，其中 `<fq.name>` 是该注解的完整限定类名。对于不同的注解可以多次出现。这对于管理特定库的迁移状态非常有用。

其中 `strict`、`warn` 与 `ignore` 值的含义与 `MigrationStatus` 中的相同，并且只有 `strict` 模式会影响注解声明中的类型在 Kotlin 中的呈现。

> 注意：内置的 JSR-305 注解 `@Nonnull`、`@Nullable` 与 `@CheckForNull` 总是启用并影响所注解的声明在 Kotlin 中呈现，无论如何配置编译器的 `-Xjsr305` 标志。

例如，将 `-Xjsr305=ignore -Xjsr305=under-migration:ignore -Xjsr305=@org.library.MyNullable:warn` 添加到编译器参数中，会使编译器对由 `@org.library.MyNullable` 标注的不当用法生成警告，而忽略所有其他 JSR-305 注解。

对于 kotlin 1.1.50+/1.2 版本，其默认行为等同于 `-Xjsr305=warn`。`strict` 值应认为是实验性的（以后可能添加更多检测）。

已映射类型

Kotlin 特殊处理一部分 Java 类型。这样的类型不是“按原样”从 Java 加载，而是 *映射* 到相应的 Kotlin 类型。映射只发生在编译期间，运行时表示保持不变。Java 的原生类型映射到相应的 Kotlin 类型（请记住平台类型）：

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

一些非原生的内置类型也会作映射：

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.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Java 的装箱原始类型映射到可空的 Kotlin 类型：

Java type	Kotlin type
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?

请注意，用作类型参数的装箱原始类型映射到平台类型：例如，`List<java.lang.Integer>` 在 Kotlin 中会成为 `List<Int!>`。

集合类型在 Kotlin 中可以是只读的或可变的，因此 Java 集合类型作如下映射：（下表中的所有 Kotlin 类型都驻留在 `kotlin.collections` 包中）：

Java 类型	Kotlin 只读类型	Kotlin 可变类型	加载的平台类型
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 的数组按下文所述映射：

Java 类型	Kotlin 类型
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

注意：这些 Java 类型的静态成员不能在相应 Kotlin 类型的伴生对象中直接访问。要调用它们，请使用 Java 类型的完整限定名，例如 `java.lang.Integer.toHexString(foo)`。

Kotlin 中的 Java 泛型

Kotlin 的泛型与 Java 有点不同（参见泛型）。当将 Java 类型导入 Kotlin 时，我们会执行一些转换：

- Java 的通配符转换成类型投影，
  - `Foo<? extends Bar>` 转换成 `Foo<out Bar!>!`，
  - `Foo<? super Bar>` 转换成 `Foo<in Bar!>!`；
- Java的原始类型转换成星投影，
  - `List` 转换成 `List<*>!`，即 `List<out Any?>!`。

和 Java 一样，Kotlin 在运行时不保留泛型，即对象不携带传递到他们构造器中的那些类型参数的实际类型。即 `ArrayList<Integer>()` 和 `ArrayList<Character>()` 是不能区分的。这使得执行 `is`-检测不可能照顾到泛型。Kotlin 只允许 `is`-检测星投影的泛型类型：

```
if (a is List<Int>) // 错误：无法检测它是否真的是一个 Int 列表
// but
if (a is List<*>) // OK：不保证列表的内容
```

Java 数组

与 Java 不同，Kotlin 中的数组是不型变的。这意味着 Kotlin 不允许我们把一个 `Array<String>` 赋值给一个 `Array<Any>`，从而避免了可能的运行时故障。Kotlin 也禁止我们把一个子类的数组当做超类的数组传递给 Kotlin 的方法，但是对于 Java 方法，这是允许的（通过 `Array<(out) String>!` 这种形式的平台类型）。

Java 平台上，数组会使用原生数据类型以避免装箱/拆箱操作的开销。由于 Kotlin 隐藏了这些实现细节，因此需要一个变通方法来与 Java 代码进行交互。对于每种原生类型的数组都有一个特化的类（`IntArray`、`DoubleArray`、`CharArray` 等等）来处理这种情况。它们与 `Array` 类无关，并且会编译成 Java 原生类型数组以获得最佳性能。

假设有一个接受 int 数组索引的 Java 方法：

```
public class JavaArrayExample {

    public void removeIndices(int[] indices) {
        // 在此编码……
    }
}
```

在 Kotlin 中你可以这样传递一个原生类型的数组：

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // 将 int[] 传给方法
```

当编译为 JVM 字节代码时，编译器会优化对数组的访问，这样就不会引入任何开销：

```
val array = arrayOf(1, 2, 3, 4)
array[1] = array[1] * 2 // 不会实际生成对 get() 和 set() 的调用
for (x in array) { // 不会创建迭代器
    print(x)
}
```

即使当我们使用索引定位时，也不会引入任何开销：

```
for (i in array.indices) {// 不会创建迭代器
    array[i] += 2
}
```

最后，in-检测也没有额外开销：

```
if (i in array.indices) { // 同 (i >= 0 && i < array.size)
    print(array[i])
}
```

Java 可变参数

Java 类有时声明一个具有可变数量参数（varargs）的方法来使用索引：

```
public class JavaArrayExample {

    public void removeIndicesVarArg(int... indices) {
        // 在此编码……
    }
}
```

在这种情况下，你需要使用展开运算符 `*` 来传递 `IntArray`：

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

目前无法传递 `null` 给一个声明为可变参数的方法。

操作符

由于 Java 无法标记用于运算符语法的方法，Kotlin 允许具有正确名称和签名的任何 Java 方法作为运算符重载和其他约定（`invoke()` 等）使用。不允许使用中缀调用语法调用 Java 方法。

受检异常

在 Kotlin 中，所有异常都是非受检的，这意味着编译器不会强迫你捕获其中的任何一个。因此，当你调用一个声明受检异常的 Java 方法时，Kotlin 不会强迫你做任何事情：

```
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
    }
}
```

对象方法

当 Java 类型导入到 Kotlin 中时，类型 `java.lang.Object` 的所有引用都成了 `Any`。而因为 `Any` 不是平台指定的，它只声明了 `toString()`、`hashCode()` 和 `equals()` 作为其成员，所以为了能用到 `java.lang.Object` 的其他成员，Kotlin 要用到扩展函数。

wait()/notify()

类型 `Any` 的引用没有提供 `wait()` 与 `notify()` 方法。通常不鼓励使用它们，而建议使用 `java.util.concurrent`。如果确实需要调用这两个方法的话，那么可以将引用转换为 `java.lang.Object`：

```
(foo as java.lang.Object).wait()
```

getClass()

要取得对象的 Java 类，请在类引用上使用 `java` 扩展属性：

```
val fooClass = foo::class.java
```

上面的代码使用了自 Kotlin 1.1 起支持的绑定的类引用。你也可以使用 `javaClass` 扩展属性：

```
val fooClass = foo.javaClass
```

clone()

要覆盖 `clone()`，需要继承 `kotlin.Cloneable`：

```
class Example : Cloneable {
    override fun clone(): Any { …… }
}
```

不要忘记《Effective Java》第三版 的第 13 条: *谨慎地改写clone*。

finalize()

要覆盖 `finalize()`，所有你需要做的就是简单地声明它，而不需要 `override` 关键字：

```
class C {
    protected fun finalize() {
        // 终止化逻辑
    }
}
```

根据 Java 的规则，`finalize()` 不能是 `private` 的。

从 Java 类继承

在 kotlin 中，类的超类中最多只能有一个 Java 类（以及按你所需的多个 Java 接口）。

访问静态成员

Java 类的静态成员会形成该类的“伴生对象”。我们无法将这样的“伴生对象”作为值来传递，但可以显式访问其成员，例如：

```
if (Character.isLetter(a)) { …… }
```

要访问已映射到 Kotlin 类型的 Java 类型的静态成员，请使用 Java 类型的完整限定名：`java.lang.Integer.bitCount(foo)`。

Java 反射

Java 反射适用于 Kotlin 类，反之亦然。如上所述，你可以使用 `instance::class.java`，`ClassName::class.java` 或者 `instance.javaClass` 通过 `java.lang.Class` 来进入 Java 反射。

其他支持的情况包括为一个 Kotlin 属性获取一个 Java 的 getter/setter 方法或者幕后字段、为一个 Java 字段获取一个 `KProperty`、为一个 `KFunction` 获取一个 Java 方法或者构造函数，反之亦然。

SAM 转换

就像 Java 8 一样，Kotlin 支持 SAM 转换。这意味着 Kotlin 函数字面值可以被自动的转换成只有一个非默认方法的 Java 接口的实现，只要这个方法的参数类型能够与这个 Kotlin 函数的参数类型相匹配。

你可以这样创建 SAM 接口的实例：

```
val runnable = Runnable { println("This runs in a runnable") }
```

……以及在方法调用中：

```
val executor = ThreadPoolExecutor()
// Java 签名: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

如果 Java 类有多个接受函数式接口的方法，那么可以通过使用将 lambda 表达式转换为特定的 SAM 类型的适配器函数来选择需要调用的方法。这些适配器函数也会按需由编译器生成：

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

请注意，SAM 转换只适用于接口，而不适用于抽象类，即使这些抽象类也只有一个抽象方法。

还要注意，此功能只适用于 Java 互操作；因为 Kotlin 具有合适的函数类型，所以不需要将函数自动转换为 Kotlin 接口的实现，因此不受支持。

在 Kotlin 中使用 JNI

要声明一个在本地（C 或 C++）代码中实现的函数，你需要使用 `external` 修饰符来标记它：

```
external fun foo(x: Int): Double
```

其余的过程与 Java 中的工作方式完全相同。

Java 中调用 Kotlin

Java 可以轻松调用 Kotlin 代码。例如，可以在 Java 方法中无缝创建与操作 Kotlin 类的实例。然而，在将 Kotlin 代码集成到 Java 中时，需要注意 Java 与 Kotlin 之间的一些差异。在本页，我们会描述定制 Kotlin 代码与其 Java 客户端的互操作的方法。

属性

Kotlin 属性会编译成以下 Java 元素：

- 一个 getter 方法，名称通过加前缀 `get` 算出；
- 一个 setter 方法，名称通过加前缀 `set` 算出（只适用于 `var` 属性）；
- 一个私有字段，与属性名称相同（仅适用于具有幕后字段的属性）。

例如，`var firstName: String` 编译成以下 Java 声明：

```
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

如果属性的名称以 `is` 开头，则使用不同的名称映射规则：getter 的名称与属性名称相同，并且 setter 的名称是通过将 `is` 替换为 `set` 获得。例如，对于属性 `isOpen`，其 getter 会称做 `isOpen()`，而其 setter 会称做 `setOpen()`。这一规则适用于任何类型的属性，并不仅限于 `Boolean`。

包级函数

在 `org.example` 包内的 `app.kt` 文件中声明的所有的函数和属性，包括扩展函数，都编译成一个名为 `org.example.AppKt` 的 Java 类的静态方法。

```
// app.kt
package org.example

class Util

fun getTime() { /*……*/ }
```

```
// Java
new org.example.Util();
org.example.AppKt.getTime();
```

可以使用 `@JvmName` 注解修改生成的 Java 类的类名：

```
@file:JvmName("DemoUtils")

package org.example

class Util

fun getTime() { /*……*/ }
```

```
// Java
new org.example.Util();
org.example.DemoUtils.getTime();
```

如果多个文件中生成了相同的 Java 类名（包名相同并且类名相同或者有相同的 `@JvmName` 注解）通常是错误的。然而，编译器能够生成一个单一的 Java 外观类，它具有指定的名称且包含来自所有文件中具有该名称的所有声明。要启用生成这样的外观，请在所有相关文件中使用 `@JvmMultifileClass` 注解。

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getTime() { /*……*/ }
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getDate() { /*……*/ }
```

```
// Java
org.example.Utils.getTime();
org.example.Utils.getDate();
```

实例字段

如果需要在 Java 中将 Kotlin 属性作为字段暴露，那就使用 `@JvmField` 注解对其标注。该字段将具有与底层属性相同的可见性。如果一个属性有幕后字段（backing field）、非私有、没有 `open` / `override` 或者 `const` 修饰符并且不是被委托的属性，那么你可以用 `@JvmField` 注解该属性。

```
class User(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(User user) {
        return user.ID;
    }
}
```

延迟初始化的属性（在Java中）也会暴露为字段。该字段的可见性与 `lateinit` 属性的 setter 相同。

静态字段

在具名对象或伴生对象中声明的 Kotlin 属性会在该具名对象或包含伴生对象的类中具有静态幕后字段。

通常这些字段是私有的，但可以通过以下方式之一暴露出来：

— `@JvmField` 注解；

— `lateinit` 修饰符；

— `const` 修饰符。

使用 `@JvmField` 标注这样的属性使其成为与属性本身具有相同可见性的静态字段。

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// Java
Key.COMPARATOR.compare(key1, key2);
// Key 类中的 public static final 字段
```

在具名对象或者伴生对象中的一个延迟初始化的属性具有与属性 setter 相同可见性的静态幕后字段。

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java
Singleton.provider = new Provider();
// 在 Singleton 类中的 public static 非-final 字段
```

(在类中以及在顶层)以 `const` 声明的属性在 Java 中会成为静态字段:

```
// 文件 example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

在 Java 中:

```
int const = Obj.CONST;
int max = ExampleKt.MAX;
int version = C.VERSION;
```

静态方法

如上所述,Kotlin 将包级函数表示为静态方法。Kotlin 还可以为具名对象或伴生对象中定义的函数生成静态方法,如果你将这些函数标注为 `@JvmStatic` 的话。如果你使用该注解,编译器既会在相应对象的类中生成静态方法,也会在对象自身中生成实例方法。例如:

```
class C {
    companion object {
        @JvmStatic fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

现在,`callStatic()` 在 Java 中是静态的,而 `callNonStatic()` 不是:

```
C.callStatic(); // 没问题
C.callNonStatic(); // 错误：不是一个静态方法
C.Companion.callStatic(); // 保留实例方法
C.Companion.callNonStatic(); // 唯一的工作方式
```

对于具名对象也同样：

```
object Obj {
    @JvmStatic fun callStatic() {}
    fun callNonStatic() {}
}
```

在 Java 中：

```
Obj.callStatic(); // 没问题
Obj.callNonStatic(); // 错误
Obj.INSTANCE.callNonStatic(); // 没问题，通过单例实例调用
Obj.INSTANCE.callStatic(); // 也没问题
```

自 Kotlin 1.3 起，`@JvmStatic` 也适用于在接口的伴生对象中定义的函数。这类函数会编译为接口中的静态方法。请注意，接口中的静态方法是 Java 1.8 中引入的，因此请确保使用相应的编译目标。

```
interface ChatBot {
    companion object {
        @JvmStatic fun greet(username: String) {
            println("Hello, $username")
        }
    }
}
```

`@JvmStatic` 注解也可以应用于对象或伴生对象的属性，使其 getter 和 setter 方法在该对象或包含该伴生对象的类中是静态成员。

接口中的默认方法

> 默认方法仅适用于面向 JVM 1.8 及更高版本。

> `@JvmDefault` 注解在 Kotlin 1.3 中是实验性的。其名称与行为都可能发生变化，导致将来不兼容。

自 JDK 1.8 起，Java 中的接口可以包含默认方法。可以将 Kotlin 接口的非抽象成员为实现它的 Java 类声明为默认。如需将一个成员声明为默认，请使用 `@JvmDefault` 注解标记之。这是一个带有默认方法的 Kotlin 接口的一个示例：

```
interface Robot {
    @JvmDefault fun move() { println("~walking~") }
    fun speak(): Unit
}
```

默认实现对于实现该接口的 Java 类都可用。

```
//Java 实现
public class C3PO implements Robot {
    // 来自 Robot 的 move() 实现隐式可用
    @Override
    public void speak() {
        System.out.println("I beg your pardon, sir");
    }
}
```

```
C3PO c3po = new C3PO();
c3po.move(); // 来自 Robot 接口的默认实现
c3po.speak();
```

接口的实现者可以覆盖默认方法。

```
//Java
public class BB8 implements Robot {
    //自己实现默认方法
    @Override
    public void move() {
        System.out.println("~rolling~");
    }

    @Override
    public void speak() {
        System.out.println("Beep-beep");
    }
}
```

为了让 `@JvmDefault` 生效，编译该接口必须带有 `-Xjvm-default` 参数。根据添加注解的情况，指定下列值之一：

— `-Xjvm-default=enabled` 只添加带有 `@JvmDefault` 注解的新方法时使用。这包括为 API 添加整个接口。

— `-Xjvm-default=compatibility` 将 `@JvmDefault` 添加到以往 API 中就有的方法时使用。这种模式有助于避免兼容性破坏：为先前版本编写的所有接口实现都会与新版本完全兼容。然而，兼容模式可能会增大生成字节码的规模并且影响性能。

关于兼容性的更多详情请参见 `@JvmDefault` 参考页。

在委托中使用

请注意，如果将带有 `@JvmDefault` 的方法的接口用作委托，那么即是实际的委托类型提供了自己的实现，也会调用默认方法的实现。

```
interface Producer {
    @JvmDefault fun produce() {
        println("interface method")
    }
}

class ProducerImpl: Producer {
    override fun produce() {
        println("class method")
    }
}

class DelegatedProducer(val p: Producer): Producer by p {
}

fun main() {
    val prod = ProducerImpl()
    DelegatedProducer(prod).produce() // 输出“interface method”
}
```

关于 Kotlin 中接口委托的更多详情，请参见委托。

可见性

Kotlin 的可见性以下列方式映射到 Java：

— `private` 成员编译成 `private` 成员；

— `private` 的顶层声明编译成包级局部声明；

— `protected` 保持 `protected`（注意 Java 允许访问同一个包中其他类的受保护成员，而 Kotlin 不能，所以 Java 类会访问更广泛的代码）；

— `internal` 声明会成为 Java 中的 `public`。`internal` 类的成员会通过名字修饰，使其更难以在 Java 中意外使用到，并且根据 Kotlin 规则使其允许重载相同签名的成员而互不可见；

— `public` 保持 `public`。

KClass

有时你需要调用有 `KClass` 类型参数的 Kotlin 方法。因为没有从 `Class` 到 `KClass` 的自动转换，所以你必须通过调用 `Class<T>.kotlin` 扩展属性的等价形式来手动进行转换：

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

用 @JvmName 解决签名冲突

有时我们想让一个 Kotlin 中的具名函数在字节码中有另外一个 JVM 名称。最突出的例子是由于*类型擦除*引发的：

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

这两个函数不能同时定义，因为它们的 JVM 签名是一样的：`filterValid(Ljava/util/List;)Ljava/util/List;`。如果我们真的希望它们在 Kotlin 中用相同名称，我们需要用 `@JvmName` 去标注其中的一个（或两个），并指定不同的名称作为参数：

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

在 Kotlin 中它们可以用相同的名称 `filterValid` 来访问，而在 Java 中，它们分别是 `filterValid` 和 `filterValidInt`。

同样的技巧也适用于属性 `x` 和函数 `getX()` 共存：

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

如需在没有显式实现 getter 与 setter 的情况下更改属性生成的访问器方法的名称，可以使用 `@get:JvmName` 与 `@set:JvmName`：

```
@get:JvmName("x")
@set:JvmName("changeX")
var x: Int = 23
```

生成重载

通常，如果你写一个有默认参数值的 Kotlin 函数，在 Java 中只会有一个所有参数都存在的完整参数签名的方法可见，如果希望向 Java 调用者暴露多个重载，可以使用 `@JvmOverloads` 注解。

该注解也适用于构造函数、静态方法等。它不能用于抽象方法，包括在接口中定义的方法。

```
class Circle @JvmOverloads constructor(centerX: Int, centerY: Int, radius: Double = 1.0) {
    @JvmOverloads fun draw(label: String, lineWidth: Int = 1, color: String = "red") { /*……*/ }
}
```

对于每一个有默认值的参数，都会生成一个额外的重载，这个重载会把这个参数和它右边的所有参数都移除掉。在上例中，会生成以下代码：

```
// 构造函数:
Circle(int centerX, int centerY, double radius)
Circle(int centerX, int centerY)

// 方法
void draw(String label, int lineWidth, String color) { }
void draw(String label, int lineWidth) { }
void draw(String label) { }
```

请注意，如次构造函数中所述，如果一个类的所有构造函数参数都有默认值，那么会为其生成一个公有的无参构造函数。这就算没有 `@JvmOverloads` 注解也有效。

受检异常

如上所述，Kotlin 没有受检异常。所以，通常 Kotlin 函数的 Java 签名不会声明抛出异常。于是如果我们有一个这样的 Kotlin 函数：

```
// example.kt
package demo

fun writeToFile() {
    /*……*/
    throw IOException()
}
```

然后我们想要在 Java 中调用它并捕捉这个异常：

```
// Java
try {
  demo.Example.writeToFile();
}
catch (IOException e) { // 错误: writeToFile() 未在 throws 列表中声明 IOException
  // ……
}
```

因为 `writeToFile()` 没有声明 `IOException`，我们从 Java 编译器得到了一个报错消息。为了解决这个问题，要在 Kotlin 中使用 `@Throws` 注解。

```
@Throws(IOException::class)
fun writeToFile() {
    /*……*/
    throw IOException()
}
```

空安全性

当从 Java 中调用 Kotlin 函数时，没人阻止我们将 `null` 作为非空参数传递。这就是为什么 Kotlin 给所有期望非空参数的公有函数生成运行时检测。这样我们就能在 Java 代码里立即得到 `NullPointerException`。

型变的泛型

当 Kotlin 的类使用了声明处型变，有两种选择可以从 Java 代码中看到它们的用法。让我们假设我们有以下类和两个使用它的函数：

```
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
```

一种看似理所当然地将这俩函数转换成 Java 代码的方式可能会是：

```
Box<Derived> boxDerived(Derived value) { …… }
Base unboxBase(Box<Base> box) { …… }
```

问题是，在 Kotlin 中我们可以这样写 `unboxBase(boxDerived("s"))`，但是在 Java 中是行不通的，因为在 Java 中类 `Box` 在其泛型参数 `T` 上是*不型变的*，于是 `Box<Derived>` 并不是 `Box<Base>` 的子类。要使其在 Java 中工作，我们按以下这样定义 `unboxBase`：

```
Base unboxBase(Box<? extends Base> box) { …… }
```

这里我们使用 Java 的*通配符类型*（`? extends Base`）来通过使用处型变来模拟声明处型变，因为在 Java 中只能这样。

当它*作为参数*出现时，为了让 Kotlin 的 API 在 Java 中工作，对于协变定义的 `Box` 我们生成 `Box<Super>` 作为 `Box<? extends Super>`（或者对于逆变定义的 `Foo` 生成 `Foo<? super Bar>`）。当它是一个返回值时，我们不生成通配符，因为否则 Java 客户端将必须处理它们（并且它违反常用 Java 编码风格）。因此，我们的示例中的对应函数实际上翻译如下：

```
// 作为返回类型—没有通配符
Box<Derived> boxDerived(Derived value) { …… }

// 作为参数—有通配符
Base unboxBase(Box<? extends Base> box) { …… }
```

当参数类型是 final 时，生成通配符通常没有意义，所以无论在什么地方 `Box<String>` 始终转换为 `Box<String>`。

如果我们在默认不生成通配符的地方需要通配符，我们可以使用 `@JvmWildcard` 注解：

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// 将被转换成
// Box<? extends Derived> boxDerived(Derived value) { …… }
```

另一方面，如果我们根本不需要默认的通配符转换，我们可以使用 `@JvmSuppressWildcards`

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// 会翻译成
// Base unboxBase(Box<Base> box) { …… }
```

`@JvmSuppressWildcards` 不仅可用于单个类型参数，还可用于整个声明（如函数或类），从而抑制其中的所有通配符。

`Nothing` 类型翻译

类型 `Nothing` 是特殊的，因为它在 Java 中没有自然的对应。确实，每个 Java 引用类型，包括 `java.lang.Void` 都可以接受 `null` 值，但是 Nothing 不行。因此，这种类型不能在 Java 世界中准确表示。这就是为什么在使用 `Nothing` 参数的地方 Kotlin 生成一个原始类型：

```
fun emptyList(): List<Nothing> = listOf()
// 会翻译成
// List emptyList() { …… }
```

JavaScript

搭建 Kotlin/JS 项目

Kotlin/JS 项目使用 Gradle 作为构建系统。为了开发者轻松管理其 Kotlin/JS 项目，我们提供了 Kotlin/JS Gradle 插件。该插件提供项目配置工具以及用以自动执行 JavaScript 开发中常用的例程的帮助程序。举个例子，该插件会下载 Yarn 软件包管理器，在后台管理 npm 依赖，并使用 webpack 从 Kotlin 项目构建 JavaScript 包。

要在 IntelliJ IDEA 中创建 Kotlin/JS 项目，请转至 **文件(File) | 新建(New) | 项目(Project)**，并勾选 **Gradle | Kotlin/JS for browser** 或 **Kotlin/JS for Node.js**。请不要勾选 **Java** 复选框。

另外，你可以在 Gradle build file (`build.gradle` or `build.gradle.kts`) 中手动将 `org.jetbrains.kotlin.js` 插件应用于 Gradle 项目。如果你使用 Gradle Kotlin DSL，则可以使用插件 `kotlin("js")`。

```
plugins {
    id 'org.jetbrains.kotlin.js' version '1.3.72'
}
```

```
plugins {
     kotlin("js") version "1.3.72"
}
```

Kotlin/JS 插件可让你在构建脚本的 `kotlin` 部分中管理项目的各个方面。

```
kotlin {
    //...
}
```

在 `kotlin` 部分中，你可以管理以下方面：

- 选择执行环境: 浏览器或 Node.js
- 管理依赖: Maven 和 npm
- 配置 run 任务
- 配置 test 任务
- 配置 webpack 绑定 针对于浏览器项目
- 分发目标目录

选择执行环境

Kotlin/JS 项目可以针对两个不同的执行环境：

- Browser，用于浏览器中客户端脚本
- Node.js，用于在浏览器外部运行 JavaScript 代码，例如，用于服务器端脚本。

要定义 Kotlin/JS 项目的目标执行环境，请在 `target` 部分添加 `browser {}` 或 `nodejs {}`。

```
kotlin {
    target {
        browser {
        }
    }
}
```

或者

```
kotlin.target.browser {
}
```

Kotlin/JS 插件会自动配置其任务与所选环境配合工作。这项操作包括下载与安装运行和测试应用程序所需的依赖。因此，开发者无需额外配置就可以构建、运行和测试简单项目。

管理依赖

就像其他任何的 Gradle 项目一样，Kotlin/JS 项目支持位于构建脚本的 `dependencies` 部分的传统 Gradle 依赖声明。

```
dependencies {
    implementation 'org.example.myproject:1.1.0'
}
```

```
dependencies {
    implementation("org.example.myproject", "1.1.0")
}
```

Kotlin/JS Gradle 插件还支持构建脚本的 `kotlin` 部分中特定 `sourceSets` 的依赖声明。

```
kotlin {
    sourceSets {
        main {
            dependencies {
                implementation 'org.example.myproject:1.1.0'
            }
        }
    }
}
```

```
kotlin {
  sourceSets["main"].dependencies {
    implementation("org.example.myproject", "1.1.0")
  }
}
```

Kotlin 标准库

所有 Kotlin/JS 项目都必须依赖 Kotlin/JS 标准库。如果你的项目包含用 Kotlin 编写的测试，则还应在添加 kotlin.test 依赖项。

```
dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib-js'
    testImplementation 'org.jetbrains.kotlin:kotlin-test-js'
}
```

```
dependencies {
    implementation(kotlin("stdlib-js"))
    testImplementation(kotlin("test-js"))
}
```

npm 依赖

在 JavaScript 中，管理依赖项的常用方法是 npm。它提供了最大的 JavaScript 模块公共存储库以及用于下载它们的工具。

Kotlin/JS 插件使你可以在 Gradle 构建脚本中声明 npm 依赖项以及其他依赖项，并自动执行其他所有操作。它安装了 Yarn 程序包管理器，并使用它来将依赖项从 npm 存储库下载项目的 `node_modules` 目录 —— JavaScript 项目的 npm 依赖项的一般位置。

要声明 npm 依赖项，将其名称与版本传给依赖项声明内的 `npm()` 函数。

```
dependencies {
    implementation npm('react', '16.12.0')
}
```

```
dependencies {
    implementation(npm("react", "16.12.0"))
}
```

安装 npm 依赖项后，你可以按照在 Kotlin 中调用 JS 中所述，在代码中使用其 API。

配置 run 任务

Kotlin/JS 插件提供了一个 `run` 任务，使你无需额外配置即可运行项目。它使用 webpack DevServer 来运行 Kotlin/JS 项目。如果要自定义 DevServer 配置，请更改其端口，请使用 webpack 配置文件。

要运行项目，请执行标准生命周期的 `run` 任务：

```
./gradlew run
```

要在浏览器中查看源文件更改而不想重新启动 DevServer，请使用 Gradle 连续构建（continuous build）：

```
./gradlew run --continuous
```

或者

```
./gradlew run -t
```

配置 test 任务

Kotin/JS Gradle 插件会自动为项目设置测试基础结构。对于浏览器项目，它将下载并安装具有其他必需依赖的 Karma 测试运行程序；对于 NodeJS 项目，使用 Mocha 测试框架。

该插件还提供了有用的测试功能，例如：

— 源代码映射文件生成
— 测试报告生成
— 在控制台中测试运行结果

默认情况下，该插件使用 Headless Chrome 来运行浏览器测试。你还可以通过在构建脚本中的 `useKarma` 部分中添加相应的条目，从而在其他浏览器中运行它们：

```
kotlin.target.browser {
    testTask {
        useKarma {
            useIe()
            useSafari()
            useFirefox()
            useChrome()
            useChromeCanary()
            useChromeHeadless()
            usePhantomJS()
            useOpera()
        }
    }
}
```

如果要跳过测试，请将 `enabled = false` 这一行添加到 `testTask` 中。

```
kotlin.target.browser {
    testTask {
        enabled = false
    }
}
```

要运行测试，请执行标准生命周期 `check` 任务：

```
./gradlew check
```

配置 Webpack 绑定

对于浏览器目标，Kotlin/JS 插件使用众所周知的 Webpack 模块捆绑器。

Kotlin/JS Gradle 插件会在构建时自动生成一个标准的 webpack 配置文件，可以在 `build/js/packages/projectName/webpack.config.js` 中找到该文件。

最常见的 webpack 调整可以直接通过 Gradle 构建文件中的 `kotlin.target.browser.webpackTask` 配置块进行。

如果要进一步调整 webpack 配置，请将其他配置文件放在项目根目录中名为 `webpack.config.d` 的目录中。在构建项目时，所有 JS 配置文件都会自动被合并到 `build/js/packages/projectName/webpack.config.js` 文件中。例如，要添加新的 webpack loader，请将以下内容添加到 `webpack.config.d` 中的 `.js` 文件中：

```
config.module.rules.push({
    test: /\.extension$/,
    loader: 'loader-name'
});
```

所有 webpack 配置功能在其 文档 中都有详细说明。

为了通过 webpack 构建可执行的 JavaScript 构件，Kotlin/JS 插件包含 `browserDevelopmentWebpack` 以及 `browserProductionWebpack` Gradle 任务。执行它们分别获得用于开发或生产的构件：

```
./gradlew browserProductionWebpack
```

配置 Yarn

要配置其他 Yarn 特性，请将 `.yarnrc` 文件放在项目的根目录中。在构建时，它会被自动拾取。

例如，要将自定义 registry 用于 npm 软件包，请将以下行添加到项目根目录中名为 `.yarnrc` 的文件中：

```
registry "http://my.registry/api/npm/"
```

要了解有关 `.yarnrc` 的更多信息，请访问 Yarn 官方文档。

分发目标目录

默认情况下，Kotlin/JS 项目构建的结果位于项目根目录下的 `/build/distribution` 目录中。

要为项目分发文件设置另一个位置，请在构建脚本中的 `browser` 里添加 `distribution`，然后为它的 `directory` 属性赋值。运行项目构建任务后，Gradle 会将输出的内容和项目资源一起保存在此位置。

```
kotlin.target.browser {
    distribution {
        directory = file("$projectDir/output/")
    }
}
```

```
kotlin.target.browser {
    distribution {
        directory = File("$projectDir/output/")
    }
}
```

动态类型

> 在面向 JVM 平台的代码中不支持动态类型

作为一种静态类型的语言，Kotlin仍然需要与无类型或松散类型的环境（例如 JavaScript生态系统）进行互操作。为了方便这些使用场景，语言中有 `dynamic` 类型可用：

```
val dyn: dynamic = ……
```

`dynamic` 类型基本上关闭了 Kotlin 的类型检测系统：

- 该类型的值可以赋值给任何变量或作为参数传递到任何位置；
- 任何值都可以赋值给 `dynamic` 类型的变量，或者传递给一个接受 `dynamic` 作为参数的函数；
- `null`-检测对这些值是禁用的。

`dynamic` 最特别的特性是，我们可以对 `dynamic` 变量调用**任何**属性或以任意参数调用**任何**函数：

```
dyn.whatever(1, "foo", dyn) // “whatever”在任何地方都没有定义
dyn.whatever(*arrayOf(1, 2, 3))
```

在 JavaScript 平台上，该代码将按照原样编译：在生成的 JavaScript 代码中，Kotlin中的 `dyn.whatever(1)` 变为 `dyn.whatever(1)` 。

当在 `dynamic` 类型的值上调用 Kotlin 写的函数时，请记住由 Kotlin 到 JavaScript 编译器执行的名字修饰。你可能需要使用 @JsName 注解为要调用的函数分配明确的名称。

动态调用总是返回 `dynamic` 作为结果，所以我们可以自由地这样链接调用：

```
dyn.foo().bar.baz()
```

当我们把一个 lambda 表达式传给一个动态调用时，它的所有参数默认都是 `dynamic` 类型的：

```
dyn.foo {
    x -> x.bar() // x 是 dynamic
}
```

使用 `dynamic` 类型值的表达式会按照原样转换为 JavaScript，并且不使用 Kotlin 运算符约定。支持以下运算符：

- 二元：`+`、`-`、`*`、`/`、`%`、`>`、`<`、`>=`、`<=`、`==`、`!=`、`===`、`!==`、`&&`、`||`
- 一元
  - 前置：`-`、`+`、`!`
  - 前置及后置：`++`、`--`
- 赋值：`+=`、`-=`、`*=`、`/=`、`%=`
- 索引访问：
  - 读：`d[a]`，多于一个参数会出错
  - 写：`d[a1] = a2`，`[]` 中有多于一个参数会出错

`in`、`!in` 以及 `..` 操作对于 `dynamic` 类型的值是禁用的。

更多技术说明请参见规范文档。

Kotlin 中调用 JavaScript

Kotlin 已被设计为能够与 Java 平台轻松互操作。它将 Java 类视为 Kotlin 类，并且 Java 也将 Kotlin 类视为 Java 类。但是，JavaScript 是一种动态类型语言，这意味着它不会在编译期检测类型。你可以通过动态类型在 Kotlin 中自由地与 JavaScript 交流，但是如果你想要 Kotlin 类型系统的全部威力，你可以为 JavaScript 库创建 Kotlin 头文件。

内联 JavaScript

你可以使用 js("……") 函数将一些 JavaScript 代码嵌入到 Kotlin 代码中。例如：

```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
```

`js` 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") // 此处报错
}
fun getTypeof() = "typeof"
```

`external` 修饰符

要告诉 Kotlin 某个声明是用纯 JavaScript 编写的，你应该用 `external` 修饰符来标记它。当编译器看到这样的声明时，它假定相应类、函数或属性的实现由开发人员提供，因此不会尝试从声明中生成任何 JavaScript 代码。这意味着你应该省略 `external` 声明内容的代码体。例如：

```
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
```

请注意，嵌套的声明会继承 `external` 修饰符，即在 `Node` 类中，我们在成员函数和属性之前并不放置 `external`。

`external` 修饰符只允许在包级声明中使用。你不能声明一个非 `external` 类的 `external` 成员。

声明类的（静态）成员

在 JavaScript 中，你可以在原型或者类本身上定义成员。即：

```
function MyClass() { …… }
MyClass.sharedMember = function() { /* 实现 */ };
MyClass.prototype.ownMember = function() { /* 实现 */ };
```

Kotlin 中没有这样的语法。然而，在 Kotlin 中我们有伴生（`companion`）对象。Kotlin 以特殊的方式处理 `external` 类的伴生对象：替代期待一个对象的是，它假定伴生对象的成员就是该类自身的成员。要描述来自上例中的 `MyClass`，你可以这样写：

```
external class MyClass {
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

声明可选参数

一个外部函数可以有可选参数。JavaScript 实现实际上如何计算这些参数的默认值，是 Kotlin 所不知道的，因此在 Kotlin 中不可能使用通常的语法声明这些参数。你应该使用以下语法：

```
external fun myFunWithOptionalArgs(x: Int,
    y: String = definedExternally,
    z: Long = definedExternally)
```

这意味着你可以使用一个必需参数和两个可选参数来调用 `myFunWithOptionalArgs`（它们的默认值由一些 JavaScript 代码算出）。

扩展 JavaScript 类

你可以轻松扩展 JavaScript 类，因为它们是 Kotlin 类。只需定义一个 `external` 类并用非 `external` 类扩展它。例如：

```
external open class HTMLElement : Element() {
    /* 成员 */
}

class CustomElement : HTMLElement() {
    fun foo() {
        alert("bar")
    }
}
```

有一些限制：

1. 当一个外部基类的函数被签名重载时，不能在派生类中覆盖它。
2. 不能覆盖一个使用默认参数的函数。

请注意，你无法用外部类扩展非外部类。

`external` 接口

JavaScript 没有接口的概念。当函数期望其参数支持 `foo` 和 `bar` 方法时，只需传递实际具有这些方法的对象。对于静态类型的 Kotlin，你可以使用接口来表达这点，例如：

```
external interface HasFooAndBar {
    fun foo()

    fun bar()
}

external fun myFunction(p: HasFooAndBar)
```

外部接口的另一个使用场景是描述设置对象。例如：

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

外部接口有一些限制：

1. 它们不能在 `is` 检测的右侧使用。
2. `as` 转换为外部接口总是成功(并在编译时产生警告)。
3. 它们不能作为具体化类型参数传递。
4. 它们不能用在类的字面值表达式(即 `I::class`)中。

JavaScript 中调用 Kotlin

Kotlin 编译器生成正常的 JavaScript 类，可以在 JavaScript 代码中自由地使用的函数和属性 。不过，你应该记住一些微妙的事情。

用独立的 JavaScript 隔离声明

为了防止损坏全局对象，Kotlin 创建一个包含当前模块中所有 Kotlin 声明的对象 。所以如果你把模块命名为 `myModule`，那么所有的声明都可以通过 `myModule` 对象在 JavaScript 中可用。例如：

```
fun foo() = "Hello"
```

可以在 JavaScript 中这样调用：

```
alert(myModule.foo());
```

这不适用于当你将 Kotlin 模块编译为 JavaScript 模块时（关于这点的详细信息请参见 JavaScript 模块）。在这种情况下，不会有一个包装对象，而是将声明作为相应类型的 JavaScript 模块对外暴露。例如，对于 CommonJS 的场景，你应该写：

```
alert(require('myModule').foo());
```

包结构

Kotlin 将其包结构暴露给 JavaScript，因此除非你在根包中定义声明，否则必须在 JavaScript 中使用完整限定的名称。例如：

```
package my.qualified.packagename

fun foo() = "Hello"
```

可以在 JavaScript 中这样调用：

```
alert(myModule.my.qualified.packagename.foo());
```

`@JsName` 注解

在某些情况下（例如为了支持重载），Kotlin 编译器会修饰（mangle）JavaScript 代码中生成的函数和属性的名称。要控制生成的名称，可以使用 `@JsName` 注解：

```
// 模块“kjs”
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
```

现在，你可以通过以下方式在 JavaScript 中使用这个类：

```
var person = new kjs.Person("Dmitry");   // 引用到模块“kjs”
person.hello();                          // 输出“Hello Dmitry!”
person.helloWithGreeting("Servus");      // 输出“Servus Dmitry!”
```

如果我们没有指定 `@JsName` 注解，相应函数的名称会包含从函数签名计算而来的后缀，例如 `hello_61zpoe$`。

请注意，Kotlin 编译器不会对 `external` 声明应用这种修饰，因此你不必在其上使用 `@JsName`。值得注意的另一个例子是从外部类继承的非外部类。在这种情况下，任何被覆盖的函数也不会被修饰。

`@JsName` 的参数需要是一个常量字符串字面值，该字面值是一个有效的标识符。任何尝试将非标识符字符串传递给 `@JsName` 时，编译器都会报错。以下示例会产生编译期错误：

```
@JsName("new C()")   // 此处出错
external fun newC()
```

在 JavaScript 中表示 Kotlin 类型

- 除了 `kotlin.Long` 的 Kotlin 数字类型映射到 JavaScript Number。
- `kotlin.Char` 映射到 JavaScript Number 来表示字符代码。
- Kotlin 在运行时无法区分数字类型（`kotlin.Long` 除外），即以下代码能够工作：

  ```
  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。
- JavaScript 中没有 64 位整数，所以 `kotlin.Long` 没有映射到任何 JavaScript 对象，它是由一个 Kotlin 类模拟的。
- `kotlin.String` 映射到 JavaScript String。
- `kotlin.Any` 映射到 JavaScript Object（即 `new Object()`、`{}` 等）。
- `kotlin.Array` 映射到 JavaScript Array。
- Kotlin 集合（即 `List`、`Set`、`Map` 等）没有映射到任何特定的 JavaScript 类型。
- `kotlin.Throwable` 映射到 JavaScript Error。
- Kotlin 在 JavaScript 中保留了惰性对象初始化。
- Kotlin 不会在 JavaScript 中实现顶层属性的惰性初始化。

自 1.1.50 版起，原生数组转换到 JavaScript 时采用 TypedArray：

- `kotlin.ByteArray`、`-.ShortArray`、`-.IntArray`、`-.FloatArray` 以及 `-.DoubleArray` 会相应地映射为 JavaScript 中的 Int8Array、Int16Array、Int32Array、Float32Array 以及 Float64Array。
- `kotlin.BooleanArray` 会映射为 JavaScript 中具有 `$type$ == "BooleanArray"` 属性的 Int8Array
- `kotlin.CharArray` 会映射为 JavaScript 中具有 `$type$ == "CharArray"` 属性的 UInt16Array
- `kotlin.LongArray` 会映射为 JavaScript 中具有 `$type$ == "LongArray"` 属性的 `kotlin.Long` 的数组。

JavaScript 模块

Kotlin 允许你将 Kotlin 项目编译为热门模块系统的 JavaScript 模块。以下是可用选项的列表：

1. 无模块（Plain）。不为任何模块系统编译。像往常一样，你可以在全局作用域中以其名称访问模块。默认使用此选项。
2. 异步模块定义（AMD，Asynchronous Module Definition），它尤其为 require.js 库所使用。
3. CommonJS 约定，广泛用于 node.js/npm（`require` 函数和 `module.exports` 对象）
4. 统一模块定义（UMD，Unified Module Definitions），它与 *AMD* 和 *CommonJS* 兼容，并且当在运行时 *AMD* 和 *CommonJS* 都不可用时，作为“plain”使用。

面向浏览器

If you're targeting the browser, you can specify the desired module type in the `webpackTask` configuration block:

```
import org.jetbrains.kotlin.gradle.targets.js.webpack.KotlinWebpackOutput.Target.COMMONJS

kotlin {
    target {
        browser {
            webpackTask {
                output.libraryTarget = COMMONJS
                //output.libraryTarget = "commonjs" // alternative
             }
        }
    }
}
```

This way, you'll get a single JS file with all dependencies included.

创建库与 node.js 文件

If you're creating a JS library or a node.js file, define the module kind as described below.

选择目标模块系统

To select module kind, set the `moduleKind` compiler option in the Gradle build script.

```
compileKotlinJs.kotlinOptions.moduleKind = "commonjs"
```

```
tasks.named("compileKotlinJs") {
    this as KotlinJsCompile
    kotlinOptions.moduleKind = "commonjs"
}
```

Available values are: `plain`, `amd`, `commonjs`, `umd`.

In Kotlin Gradle DSL, there is also a shortcut for setting the CommonJS module kind:

```
kotlin {
    target {
         useCommonJs()
    }
}
```

`@JsModule` 注解

要告诉 Kotlin 一个 `external` 类、包、函数或者属性是一个 JavaScript 模块，你可以使用 `@JsModule` 注解。考虑你有以下 CommonJS 模块叫“hello”：

```
module.exports.sayHello = function(name) { alert("Hello, " + name); }
```

你应该在 Kotlin 中这样声明：

```
@JsModule("hello")
external fun sayHello(name: String)
```

将 @JsModule 应用到包

一些 JavaScript 库导出包（命名空间）而不是函数和类。从 JavaScript 角度讲，它是一个具有一些成员的对象，这些成员*是*类、函数和属性。将这些包作为 Kotlin 对象导入通常看起来不自然。编译器允许使用以下助记符将导入的 JavaScript 包映射到 Kotlin 包：

```
@file:JsModule("extModule")
package ext.jspackage.name

external fun foo()

external class C
```

其中相应的 JavaScript 模块的声明如下：

```
module.exports = {
    foo:  { /* 此处一些代码 */ },
    C:  { /* 此处一些代码 */ }
}
```

重要提示：标有 `@file:JsModule` 注解的文件无法声明非外部成员。下面的示例会产生编译期错误：

```
@file:JsModule("extModule")
package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // 此处报错
```

导入更深的包层次结构

在前文示例中，JavaScript 模块导出单个包。但是，一些 JavaScript 库会从模块中导出多个包。Kotlin 也支持这种场景，尽管你必须为每个导入的包声明一个新的 `.kt` 文件。

例如，让我们的示例更复杂一些：

```
module.exports = {
    mylib: {
        pkg1: {
            foo: function() { /* 此处一些代码 */ },
            bar: function() { /* 此处一些代码 */ }
        },
        pkg2: {
            baz: function() { /* 此处一些代码 */ }
        }
    }
}
```

要在 Kotlin 中导入该模块，你必须编写两个 Kotlin 源文件：

```
@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

当一个声明具有 `@JsModule`、当你并不把它编译到一个 JavaScript 模块时，你不能在 Kotlin 代码中使用它。通常，开发人员将他们的库既作为 JavaScript 模块也作为可下载的 `.js` 文件分发，你可以将这些文件复制到项目的静态资源，并通过 `<script>` 元素包含。要告诉 Kotlin，可以在非模块环境中使用一个 `@JsModule` 声明，你应该放置 `@JsNonModule` 声明。例如，给定 JavaScript 代码：

```
function topLevelSayHello(name) { alert("Hello, " + name); }
if (module && module.exports) {
    module.exports = topLevelSayHello;
}
```

可以这样描述：

```
@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)
```

备注

Kotlin 以 `kotlin.js` 标准库作为单个文件分发，该文件本身被编译为 UMD 模块，因此你可以使用上述任何模块系统。在 NPM 上作为 kotlin 包 提供

JavaScript 反射

目前，JavaScript 不支持完整的 Kotlin 反射 API。唯一支持的该 API 部分是 `::class` 语法，它允许你引用一个实例的类或者与给定类型相对应的类。一个 `::class` 表达式的值是一个只能支持 simpleName 和 isInstance 成员的精简版 KClass 实现。

除此之外，你可以使用 KClass.js 访问与 JsClass 类对应的实例。该 `JsClass` 实例本身就是对构造函数的引用。这可以用于与期望构造函数的引用的 JS 函数进行互操作。

示例：

```
class A
class B
class C

inline fun <reified T> foo() {
    println(T::class.simpleName)
}

val a = A()
println(a::class.simpleName)  // 获取一个实例的类；输出“A”
println(B::class.simpleName)  // 获取一个类型的类；输出“B”
println(B::class.js.name)     // 输出“B”
foo<C>()                      // 输出“C”
```

JavaScript 无用代码消除（DCE）

Kotlin/JS Gradle 插件包含一个 *无用代码消除*（*DCE*）工具。无用代码消除通常也称为 *摇树*。通过删除未使用的属性、函数和类，它减小了大小或生成的 JavaScript 代码。

在以下情况下会出现未使用的声明：

- 函数是内联的，永远不会直接调用（除了少数情况总是发生）。
- 模块使用共享库。没有 DCE 的情况下，未使用的组件仍会进入结果包。例如，Kotlin 标准库中包含用于操作列表、数组、字符序列、DOM 适配器的函数。它们总共构成了大约 1.3MB 的文件。一个简单的 "Hello, world" 应用程序仅需要控制台例程，整个程序只有几 KB。

Kotlin/JS Gradle 插件在构建生产包时会自动处理 DCE，例如：使用 `browserProductionWebpack` 任务。开发捆绑任务不包括 DCE。

从 DCE 排除的声明

有时，即使未在模块中使用函数或类，也可能需要在结果 JavaScript 代码中保留一个函数或一个类，例如：如果要在客户端 JavaScript 代码中使用它，则可能会保留该函数或类。

为了避免某些声明被删除，请将 `dceTask` 代码块添加到 Gradle 构建脚本中，并将这些声明列为 `keep` 函数的参数。参数必须是声明的完整限定名，并且模块名称为前缀：`moduleName.dot.separated.package.name.declarationName`

```
kotlin.target.browser {
    dceTask {
        keep 'myKotlinJSModule.org.example.getName', 'myKotlinJSModule.org.example.User'
    }
}
```

```
kotlin.target.browser {
    dceTask {
        keep("myKotlinJSModule.org.example.getName", "myKotlinJSModule.org.example.User" )
    }
}
```

请注意，带有参数的函数名称在生成的 JavaScript 代码中会被修饰。为了避免消除这些函数，请在 `keep` 参数中使用修饰的名称。

已知问题：DCE 与 ktor

在 Kotlin 1.3.72 中，存在一个在 Kotlin/JS 项目中使用 ktor 的已知问题。在某些情况下，可能会遇到类型错误，例如：`<something> is not a function` 这是来自 `io.ktor:ktor-client-js:1.3.0` 或 `io.ktor:ktor-client-core:1.3.0` 构件。为避免此问题，请添加以下 DCE 配置：

```
kotlin.target.browser {
    dceTask {
        keep 'ktor-ktor-io.\$\$importsForInline\$\$.ktor-ktor-io.io.ktor.utils.io'
    }
}
```

```
kotlin.target.browser {
    dceTask {
        keep("ktor-ktor-io.\$\$importsForInline\$\$.ktor-ktor-io.io.ktor.utils.io")
    }
}
```

原生

Kotlin/Native 中的并发

Kotlin/Native 运行时并不鼓励带有互斥代码块与条件变量的经典线程式并发模型，因为已知该模型易出错且不可靠。相反，我们建议使用一系列替代方法，让你可以使用硬件并发并实现阻塞 IO。这些方法如下，并且分别会在后续各部分详细阐述：

— 带有消息传递的 worker
— 对象子图所有权转移
— 对象子图冻结
— 对象子图分离
— 使用 C 语言全局变量的原始共享内存
— 用于阻塞操作的协程（本文档未涉及）

Worker

Kotlin/Native 运行时提供了 worker 的概念来取代线程：并发执行的控制流以及与其关联的请求队列。Worker 非常像参与者模型中的参与者。一个 worker 可以与另一个 worker 交换 Kotlin 对象，从而在任何时刻每个可变对象都隶属于单个 worker，不过所有权可以转移。请参见对象转移与冻结部分。

一旦以 `Worker.start` 函数调用启动了一个 worker，就可以使用其自身唯一的整数 worker id 来寻址。其他 worker 或者非 worker 的并发原语（如 OS 线程）可以使用 `execute` 调用向 worker 发消息。

```
val future = execute(TransferMode.SAFE, { SomeDataForWorker() }) {
   // 第二个函数参数所返回的数据
   // 作为“input”参数进入 worker 例程
   input ->
   // 这里我们创建了一个当有人消费结果 future 时返回的实例
   WorkerResult(input.stringParam + " result")
}

future.consume {
  // 这里我们查看从上文例程中返回的结果。请注意 future 对象或
  // id 都可以转移给另一个 worker，所以并不是必须要在
  // 获得 future 的同一上下文中消费之。
  result -> println("result is $result")
}
```

调用 `execute` 会使用作为第二个参数传入的函数来生成一个对象子图（即一组相互引用的对象）然后将其作为一个整体传给该 worker，之后发出该请求的线程不可以再使用该对象子图。如果第一个参数是 `TransferMode.SAFE`，那么会通过图遍历来检测这一属性；而如果第一个参数是 `TransferMode.UNSAFE` 那么直接假定为 true。`execute` 的最后一个参数是一个特殊 Kotlin lambda 表达式，不可以捕获任何状态，并且实际上是在目标 worker 的上下文中调用。一旦处理完毕，就将结果转移给将会消费它地方，并将其附加到该 worker/线程的对象图中。

如果一个对象以 `UNSAFE` 模式转移，并且依然在多个并发执行子中访问，那么该程序可能会意外崩溃，因此考虑将 `UNSAFE` 作为最后的优化手段而不是通用机制来使用。

更完整的示例请参考 Kotlin/Native 版本库中的 worker 示例。

对象转移与冻结

Kotlin/Native 运行时维护的一个重要的不变式是，对象要么归单个线程/worker 所有，要么不可变（*共享 XOR 可变*）。这确保了同一数据只有一个修改方，因此不需要锁定。为了实现这个不变式，我们使用了非外部引用的对象子图的概念。这是一个没有来自子图以外的外部引用的子图，（在 ARC 系统中）可由 O(N) 复杂度进行算法检测，其中 N 是这种子图中元素的数量。这种子图通常是作为 lambda 表达式的结果而产生的（例如某些构建器），并且可能不含外部引用的对象。

冻结是一种运行时操作，通过修改对象头使给定的对象子图不可变，这样之后的修改企图都会抛出 `InvalidMutabilityException`。它是深度冻结，因此如果一个对象有指向其他对象的指针——这些对象的传递闭包也都会被冻结。冻结是单向转换，冻结的对象不能解冻。冻结的对象有一个很好的属性，由于其不可变性，它们可以在多个 worker/线程之间自由共享，而不会破坏“可变 XOR 共享”不变式。

一个对象是否已冻结，可以使用扩展属性 `isFrozen` 来检测，如果冻结了就可以共享。目前，Kotlin/Native 运行时只能在枚举对象创建后进行冻结，尽管将来可能实现自动冻结某些可证明不可变的对象。

对象子图分离

没有外部引用的对象子图可以使用 `DetachedObjectGraph<T>` 断开到 `COpaquePointer` 值的连接，该值可以存储在 `void*` 数据中，因此断开连接的对象子图可以存储在 C 语言数据结构中，并且之后还能在任意线程或 worker 中通过 `DetachedObjectGraph<T>.attach()` 加回。如果 worker 机制不足以完成特定任务，那么可以将对象子图分离与原始共享内存相结合，能够在并发线程之间进行旁路对象传输。

原始共享内存

考虑到 Kotlin/Native 与 C 语言之间通过互操作性的紧密联系，结合上文中提到的其他机制，可以构建流行的数据结构，如并发的 hashmap 或者与 Kotlin/Native 共享缓存。可以依赖共享的 C 语言数据，并在其中存储分离的对象子图的引用。考虑以下 .def 文件：

```
package = global

---
typedef struct {
  int version;
  void* kotlinObject;
} SharedData;

SharedData sharedData;
```

在运行 cinterop 工具之后，可以在版本化的全局结构中共享 Kotlin 数据，并通过自动生成的 Kotlin 代码在 Kotlin 中与其透明交互，如下所示：

```
class SharedData(rawPtr: NativePtr) : CStructVar(rawPtr) {
    var version: Int
    var kotlinObject: COpaquePointer?
}
```

因此，结合上文声明的顶层变量，可以让不同的线程看到相同的内存，并使用平台相关的同步原语来构建传统的并发结构。

全局变量与单例

全局变量常常是非预期并发问题的根源，因此 *Kotlin/Native* 实现了以下机制来防止意外通过全局对象共享状态：

- 全局变量（除非特别标记过）都只能在主线程（即首次初始化 *Kotlin/Native* 运行时的线程）中访问，如果其他线程访问这样的全局变量就会抛出 `IncorrectDereferenceException`
- 对于标有 `@kotlin.native.ThreadLocal` 注解的全局变量，每个线程都保留线程局部副本，因此变更在线程之间并不可见
- 对于标有 `@kotlin.native.SharedImmutable` 注解的变量，其值是共享的，但是在发布之前会被冻结，因此每个线程都会看到相同的值
- 单例对象（除非标有 `@kotlin.native.ThreadLocal`）都是冻结且共享的，允许惰性值（除非企图创建循环冻结结构）
- 枚举总是冻结的

结合起来，这些机制允许在多平台（MPP）项目中跨平台复用代码的自然竞态冻结编程。

Kotlin/Native 中的不可变性

Kotlin/Native 实现了严格的可变性检测，确保了重要的不变式：对象要么不可变，要么在同一时刻只在单个线程中访问（`mutable XOR global`）。

在 Kotlin/Native 中不可变性是运行时属性，可以将 `kotlin.native.concurrent.freeze` 函数应用到任意对象子图。它使从给定的对象递归可达的所有对象都不可变，这样的转换是单向操作（即这些对象之后不能解冻）。一些天然不可变的对象（如 `kotlin.String`、`kotlin.Int` 与其他原生类型，以及 `AtomicInt` 与 `AtomicReference`）默认就是冻结的。如果对已冻结对象应用了修改操作，那么会抛出 `InvalidMutabilityException` 异常。

为了实现 `mutable XOR global` 不变式，所有全局可见状态（目前有 `object` 单例与枚举）都会自动冻结。如果对象无需冻结，可以使用 `kotlin.native.ThreadLocal` 注解，这会使该对象状态成为线程局部的，因此可修改（但是变更后的状态对其他线程不可见）。

非基本类型的顶层/全局变量默认只能在主线程（即首先初始化 *Kotlin/Native* 运行时的线程）中访问。在其他线程中访问会引发 `IncorrectDereferenceException` 异常。如需其他线程可访问这种变量，可以使用 `@ThreadLocal` 注解将该值标记为线程局部，或者使用 `@SharedImmutable` 注解，它会冻结该值并使其他线程可访问。

`AtomicReference` 类可用于将变更后的冻结状态发布给其他线程，从而构建共享缓存等模式。

Kotlin/Native 库

Kotlin 编译器细节

用 Kotlin/Native 编译器生成一个库，请使用 `-produce library` 或者 `-p library` 标志。例如：

```
$ kotlinc foo.kt -p library -o bar
```

上述命令会生成一个带有 `foo.kt` 编译后的内容的库 `bar.klib`。

链接到一个库请使用 `-library <库名>` or `-l <库名>` 标志。例如：

```
$ kotlinc qux.kt -l bar
```

上述命令会由 `qux.kt` 与 `bar.klib` 生成 `program.kexe`

cinterop 工具细节

cinterop 工具为原生库生成 `.klib` 包装作为其主要输出。例如，使用 Kotlin/Native 发行版中提供的简单 `libgit2.def` 原生库定义文件

```
$ cinterop -def samples/gitchurn/src/nativeInterop/cinterop/libgit2.def -compiler-option -
I/usr/local/include -o libgit2
```

会得到 `libgit2.klib`。

更多详情请参见 INTEROP.md

klib 实用程序

klib 库管理实用程序可以探查与安装库。

可以使用以下命令。

列出库的内容：

```
$ klib contents <库名>
```

探查库的簿记细节

```
$ klib info <库名>
```

将库安装到默认位置，使用

```
$ klib install <库名>
```

将库从默认存储库中删除，使用

```
$ klib remove <库名>
```

上述所有命令都接受一个额外的 `-repository <目录>` 参数，用于指定与默认不同的存储库。

```
$ klib <命令> <库名> -repository <目录>
```

几个示例

首先创建一个库。将微型库的源代码写到 `kotlinizer.kt` 中：

```
package kotlinizer
val String.kotlinized
    get() = "Kotlin $this"
```

```
$ kotlinc kotlinizer.kt -p library -o kotlinizer
```

该库已在当前目录中创建：

```
$ ls kotlinizer.klib
kotlinizer.klib
```

现在来看看库的内容：

```
$ klib contents kotlinizer
```

可以将 `kotlinizer` 安装到默认存储库：

```
$ klib install kotlinizer
```

从当前目录中删除它的任何痕迹：

```
$ rm kotlinizer.klib
```

创建一个非常短的程序并写到 `use.kt` 中：

```
import kotlinizer.*

fun main(args: Array<String>) {
    println("Hello, ${"world".kotlinized}!")
}
```

现在编译该程序链接到刚刚创建的库：

```
$ kotlinc use.kt -l kotlinizer -o kohello
```

并运行该程序：

```
$ ./kohello.kexe
Hello, Kotlin world!
```

乐在其中！

高级主题

库搜索顺序

当给出 `-library foo` 标志时，编译器按照以下顺序搜索 `foo` 库：

* 当前编译目录或者一个绝对路径。

* 以 `-repo` 标志指定的所有存储裤。

* 安装在默认存储库（目前默认为 `~/.konan`，不过可以通过设置 **KONAN_DATA_DIR** 来更改）中的库。

* 安装在 `$installation/klib` 目录中的库。

库格式

Kotlin/Native 是包含预定义目录结构的 zip 文件，具有以下布局：

foo.klib 当解压为 **foo/** 时会有：

```
- foo/
  - targets/
    - $platform/
      - kotlin/
        - Kotlin 编译为 LLVM 位码（bitcode）。
      - native/
        - 附加原生对象的位码文件。
    - $another_platform/
      - 可以有几个平台相关的 kotlin 与原生对。
  - linkdata/
    - 一组带有序列化的链接元数据的 ProtoBuf 文件。
  - resources/
    - 图像等普通资源。(尚未使用)。
  - manifest——描述库的 *java 属性*格式文件。
```

可以在安装的 `klib/stdlib` 目录中找到示例布局。

平台库

概述

为了提供对用户原生操作系统服务的访问，`Kotlin/Native` 发行版包含了一组为每个目标平台预构建的库。我们称之为**平台库**。

POSIX 绑定

对于所有基于 `Unix` 或 `Windows` 的目标平台（包括 `Android` 与 `iPhone`），我们提供了 `posix` 平台库。它包含对 `POSIX` 标准的平台实现的绑定。

使用该库只需

```
import platform.posix.*
```

唯一不可用的目标平台是 WebAssembly。

请注意，`platform.posix` 的内容在不同平台上并不相同，就像不同的 `POSIX` 实现一样略有不同。

热门原生库

还有很多平台库可用于所在主机以及交叉编译目标。`Kotlin/Native` 发行版可以在适用的平台上访问 `OpenGL`、`zlib` 以及其他热门原生库。

在苹果平台上提供了 `objc` 库，用来与 Objective-C 进行互操作。

详细信息请核查发行版的 `dist/klib/platform/$target` 的内容。

默认可用

来自平台库的包都默认可用。使用时无需指定特殊的链接标志。`Kotlin/Native` 编译器会自动检测访问了哪些平台库，并自动链接所需的库。

另一方面，发行版中的平台库仅仅是对原生库的包装与绑定。这意味着计算机上需要已经安装了原生库自身（`.so`、`.a`、`.dylib`、`.dll` 等）。

示例

`Kotlin/Native` 安装包中提供了大量的示例演示平台库的使用。详见样例。

Kotlin/Native 互操作

Introduction

Kotlin/Native follows the general tradition of Kotlin to provide excellent existing platform software interoperability. In the case of a native platform, the most important interoperability target is a C library. So *Kotlin/Native* comes with a `cinterop` tool, which can be used to quickly generate everything needed to interact with an external library.

The following workflow is expected when interacting with the native library.

— create a `.def` file describing what to include into bindings

— use the `cinterop` tool to produce Kotlin bindings

— run *Kotlin/Native* compiler on an application to produce the final executable

The interoperability tool analyses C headers and produces a "natural" mapping of the types, functions, and constants into the Kotlin world. The generated stubs can be imported into an IDE for the purpose of code completion and navigation.

Interoperability with Swift/Objective-C is provided too and covered in a separate document OBJC_INTEROP.md.

Platform libraries

Note that in many cases there's no need to use custom interoperability library creation mechanisms described below, as for APIs available on the platform standardized bindings called platform libraries could be used. For example, POSIX on Linux/macOS platforms, Win32 on Windows platform, or Apple frameworks on macOS/iOS are available this way.

Simple example

Install libgit2 and prepare stubs for the git library:

```
cd samples/gitchurn
../../dist/bin/cinterop -def src/main/c_interop/libgit2.def \
 -compiler-option -I/usr/local/include -o libgit2
```

Compile the client:

```
../../dist/bin/kotlinc src/main/kotlin \
 -library libgit2 -o GitChurn
```

Run the client:

```
./GitChurn.kexe ../..
```

Creating bindings for a new library

To create bindings for a new library, start by creating a `.def` file. Structurally it's a simple property file, which looks like this:

```
headers = png.h
headerFilter = png.h
package = png
```

Then run the `cinterop` tool with something like this (note that for host libraries that are not included in the sysroot search paths, headers may be needed):

```
cinterop -def png.def -compiler-option -I/usr/local/include -o png
```

This command will produce a `png.klib` compiled library and `png-build/kotlin` directory containing Kotlin source code for the library.

If the behavior for a certain platform needs to be modified, you can use a format like `compilerOpts.osx` or `compilerOpts.linux` to provide platform-specific values to the options.

Note, that the generated bindings are generally platform-specific, so if you are developing for multiple targets, the bindings need to be regenerated.

After the generation of bindings, they can be used by the IDE as a proxy view of the native library.

For a typical Unix library with a config script, the `compilerOpts` will likely contain the output of a config script with the `--cflags` flag (maybe without exact paths).

The output of a config script with `--libs` will be passed as a `-linkedArgs` `kotlinc` flag value (quoted) when compiling.

Selecting library headers

When library headers are imported to a C program with the `#include` directive, all of the headers included by these headers are also included in the program. So all header dependencies are included in generated stubs as well.

This behavior is correct but it can be very inconvenient for some libraries. So it is possible to specify in the `.def` file which of the included headers are to be imported. The separate declarations from other headers can also be imported in case of direct dependencies.

Filtering headers by globs

It is possible to filter headers by globs. The `headerFilter` property value from the `.def` file is treated as a space-separated list of globs. If the included header matches any of the globs, then the declarations from this header are included into the bindings.

The globs are applied to the header paths relative to the appropriate include path elements, e.g. `time.h` or `curl/curl.h`. So if the library is usually included with `#include <SomeLbrary/Header.h>`, then it would probably be correct to filter headers with

```
headerFilter = SomeLibrary/**
```

If a `headerFilter` is not specified, then all headers are included.

Filtering by module maps

Some libraries have proper `module.modulemap` or `module.map` files in its headers. For example, macOS and iOS system libraries and frameworks do. The module map file describes the correspondence between header files and modules. When the module maps are available, the headers from the modules that are not included directly can be filtered out using the experimental `excludeDependentModules` option of the `.def` file:

```
headers = OpenGL/gl.h OpenGL/glu.h GLUT/glut.h
compilerOpts = -framework OpenGL -framework GLUT
excludeDependentModules = true
```

When both `excludeDependentModules` and `headerFilter` are used, they are applied as an intersection.

C compiler and linker options

Options passed to the C compiler (used to analyze headers, such as preprocessor definitions) and the linker (used to link final executables) can be passed in the definition file as `compilerOpts` and `linkerOpts` respectively. For example

```
compilerOpts = -DFOO=bar
linkerOpts = -lpng
```

Target-specific options, only applicable to the certain target can be specified as well, such as

```
 compilerOpts = -DBAR=bar
 compilerOpts.linux_x64 = -DFOO=foo1
 compilerOpts.mac_x64 = -DFOO=foo2
```

and so, C headers on Linux will be analyzed with `-DBAR=bar -DFOO=foo1` and on macOS with `-DBAR=bar -DFOO=foo2`. Note that any definition file option can have both common and the platform-specific part.

Adding custom declarations

Sometimes it is required to add custom C declarations to the library before generating bindings (e.g., for macros). Instead of creating an additional header file with these declarations, you can include them directly to the end of the `.def` file, after a separating line, containing only the separator sequence `---`:

```
headers = errno.h

---

static inline int getErrno() {
    return errno;
}
```

Note that this part of the `.def` file is treated as part of the header file, so functions with the body should be declared as `static`. The declarations are parsed after including the files from the `headers` list.

Including static library in your klib

Sometimes it is more convenient to ship a static library with your product, rather than assume it is available within the user's environment. To include a static library into `.klib` use `staticLibrary` and `libraryPaths` clauses. For example:

```
headers = foo.h
staticLibraries = libfoo.a
libraryPaths = /opt/local/lib /usr/local/opt/curl/lib
```

When given the above snippet the `cinterop` tool will search `libfoo.a` in `/opt/local/lib` and `/usr/local/opt/curl/lib`, and if it is found include the library binary into `klib`.

When using such `klib` in your program, the library is linked automatically.

Using bindings

Basic interop types

All the supported C types have corresponding representations in Kotlin:

- Signed, unsigned integral, and floating point types are mapped to their Kotlin counterpart with the same width.
- Pointers and arrays are mapped to `CPointer<T>?`.
- Enums can be mapped to either Kotlin enum or integral values, depending on heuristics and the definition file hints.
- Structs / unions are mapped to types having fields available via the dot notation, i.e. `someStructInstance.field1`.
- `typedef` are represented as `typealias`.

Also, any C type has the Kotlin type representing the lvalue of this type, i.e., the value located in memory rather than a simple immutable self-contained value. Think C++ references, as a similar concept. For structs (and `typedef`s to structs) this representation is the main one and has the same name as the struct itself, for Kotlin enums it is named `${type}Var`, for `CPointer<T>` it is `CPointerVar<T>`, and for most other types it is `${type}Var`.

For types that have both representations, the one with a "lvalue" has a mutable `.value` property for accessing the value.

Pointer types

The type argument `T` of `CPointer<T>` must be one of the "lvalue" types described above, e.g., the C type `struct S*` is mapped to `CPointer<S>`, `int8_t*` is mapped to `CPointer<int_8tVar>`, and `char**` is mapped to `CPointer<CPointerVar<ByteVar>>`.

C null pointer is represented as Kotlin's `null`, and the pointer type `CPointer<T>` is not nullable, but the `CPointer<T>?` is. The values of this type support all the Kotlin operations related to handling `null`, e.g. `?:`, `?.`, `!!` etc.:

```
val path = getenv("PATH")?.toKString() ?: ""
```

Since the arrays are also mapped to `CPointer<T>`, it supports the `[]` operator for accessing values by index:

```
fun shift(ptr: CPointer<BytePtr>, length: Int) {
    for (index in 0 .. length - 2) {
        ptr[index] = ptr[index + 1]
    }
}
```

The `.pointed` property for `CPointer<T>` returns the lvalue of type `T`, pointed by this pointer. The reverse operation is `.ptr`: it takes the lvalue and returns the pointer to it.

`void*` is mapped to `COpaquePointer` – the special pointer type which is the supertype for any other pointer type. So if the C function takes `void*`, then the Kotlin binding accepts any `CPointer`.

Casting a pointer (including `COpaquePointer`) can be done with `.reinterpret<T>`, e.g.:

```
val intPtr = bytePtr.reinterpret<IntVar>()
```

or

```
val intPtr: CPointer<IntVar> = bytePtr.reinterpret()
```

As is with C, these reinterpret casts are unsafe and can potentially lead to subtle memory problems in the application.

Also there are unsafe casts between `CPointer<T>?` and `Long` available, provided by the `.toLong()` and `.toCPointer<T>()` extension methods:

```
val longValue = ptr.toLong()
val originalPtr = longValue.toCPointer<T>()
```

Note that if the type of the result is known from the context, the type argument can be omitted as usual due to the type inference.

Memory allocation

The native memory can be allocated using the `NativePlacement` interface, e.g.

```
val byteVar = placement.alloc<ByteVar>()
```

or

```
val bytePtr = placement.allocArray<ByteVar>(5)
```

The most "natural" placement is in the object `nativeHeap`. It corresponds to allocating native memory with `malloc` and provides an additional `.free()` operation to free allocated memory:

```
val buffer = nativeHeap.allocArray<ByteVar>(size)
<use buffer>
nativeHeap.free(buffer)
```

However, the lifetime of allocated memory is often bound to the lexical scope. It is possible to define such scope with `memScoped { ... }`. Inside the braces, the temporary placement is available as an implicit receiver, so it is possible to allocate native memory with `alloc` and `allocArray`, and the allocated memory will be automatically freed after leaving the scope.

For example, the C function returning values through pointer parameters can be used like

```
val fileSize = memScoped {
    val statBuf = alloc<stat>()
    val error = stat("/", statBuf.ptr)
    statBuf.st_size
}
```

Passing pointers to bindings

Although C pointers are mapped to the `CPointer<T>` type, the C function pointer-typed parameters are mapped to `CValuesRef<T>`. When passing `CPointer<T>` as the value of such a parameter, it is passed to the C function as is. However, the sequence of values can be passed instead of a pointer. In this case the sequence is passed "by value", i.e., the C function receives the pointer to the temporary copy of that sequence, which is valid only until the function returns.

The `CValuesRef<T>` representation of pointer parameters is designed to support C array literals without explicit native memory allocation. To construct the immutable self-contained sequence of C values, the following methods are provided:

— `${type}Array.toCValues()`, where `type` is the Kotlin primitive type

— `Array<CPointer<T>?>.toCValues()`, `List<CPointer<T>?>.toCValues()`

— `cValuesOf(vararg elements: ${type})`, where `type` is a primitive or pointer

For example:

C:

```
void foo(int* elements, int count);
...
int elements[] = {1, 2, 3};
foo(elements, 3);
```

Kotlin:

```
foo(cValuesOf(1, 2, 3), 3)
```

Working with the strings

Unlike other pointers, the parameters of type `const char*` are represented as a Kotlin `String`. So it is possible to pass any Kotlin string to a binding expecting a C string.

There are also some tools available to convert between Kotlin and C strings manually:

— `fun CPointer<ByteVar>.toKString(): String`

— `val String.cstr: CValuesRef<ByteVar>`.

  To get the pointer, `.cstr` should be allocated in native memory, e.g.

  ```
  val cString = kotlinString.cstr.getPointer(nativeHeap)
  ```

In all cases, the C string is supposed to be encoded as UTF-8.

To skip automatic conversion and ensure raw pointers are used in the bindings, a `noStringConversion` statement in the `.def` file could be used, i.e.

```
noStringConversion = LoadCursorA LoadCursorW
```

This way any value of type `CPointer<ByteVar>` can be passed as an argument of `const char*` type. If a Kotlin string should be passed, code like this could be used:

```
memScoped {
    LoadCursorA(null, "cursor.bmp".cstr.ptr)   // for ASCII version
    LoadCursorW(null, "cursor.bmp".wcstr.ptr)  // for Unicode version
}
```

Scope-local pointers

It is possible to create a scope-stable pointer of C representation of `CValues<T>` instance using the `CValues<T>.ptr` extension property, available under `memScoped { ... }`. It allows using the APIs which require C pointers with a lifetime bound to a certain `MemScope`. For example:

```
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)
    ...
}
```

In this example, all values passed to the C API `new_menu()` have a lifetime of the innermost `memScope` it belongs to. Once the control flow leaves the `memScoped` scope the C pointers become invalid.

Passing and receiving structs by value

When a C function takes or returns a struct / union `T` by value, the corresponding argument type or return type is represented as `CValue<T>`.

`CValue<T>` is an opaque type, so the structure fields cannot be accessed with the appropriate Kotlin properties. It should be possible, if an API uses structures as handles, but if field access is required, there are the following conversion methods available:

— `fun T.readValue(): CValue<T>`. Converts (the lvalue) `T` to a `CValue<T>`. So to construct the `CValue<T>`, `T` can be allocated, filled, and then converted to `CValue<T>`.

— `CValue<T>.useContents(block: T.() -> R): R`. Temporarily places the `CValue<T>` to memory, and then runs the passed lambda with this placed value `T` as receiver. So to read a single field, the following code can be used:

```
val fieldValue = structValue.useContents { field }
```

Callbacks

To convert a Kotlin function to a pointer to a C function, `staticCFunction(::kotlinFunction)` can be used. It is also able to provide the lambda instead of a function reference. The function or lambda must not capture any values.

If the callback doesn't run in the main thread, it is mandatory to init the *Kotlin/Native* runtime by calling `kotlin.native.initRuntimeIfNeeded()`.

Passing user data to callbacks

Often C APIs allow passing some user data to callbacks. Such data is usually provided by the user when configuring the callback. It is passed to some C function (or written to the struct) as e.g. `void*`. However, references to Kotlin objects can't be directly passed to C. So they require wrapping before configuring the callback and then unwrapping in the callback itself, to safely swim from Kotlin to Kotlin through the C world. Such wrapping is possible with `StableRef` class.

To wrap the reference:

```
val stableRef = StableRef.create(kotlinReference)
val voidPtr = stableRef.asCPointer()
```

where the `voidPtr` is a `COpaquePointer` and can be passed to the C function.

To unwrap the reference:

```
val stableRef = voidPtr.asStableRef<KotlinClass>()
val kotlinReference = stableRef.get()
```

where `kotlinReference` is the original wrapped reference.

The created `StableRef` should eventually be manually disposed using the `.dispose()` method to prevent memory leaks:

```
stableRef.dispose()
```

After that it becomes invalid, so `voidPtr` can't be unwrapped anymore.

See the `samples/libcurl` for more details.

Macros

Every C macro that expands to a constant is represented as a Kotlin property. Other macros are not supported. However, they can be exposed manually by wrapping them with supported declarations. E.g. function-like macro `FOO` can be exposed as function `foo` by adding the custom declaration to the library:

```
headers = library/base.h

---

static inline int foo(int arg) {
    return FOO(arg);
}
```

Definition file hints

The `.def` file supports several options for adjusting the generated bindings.

— `excludedFunctions` property value specifies a space-separated list of the names of functions that should be ignored. This may be required because a function declared in the C header is not generally guaranteed to be really callable, and it is often hard or impossible to figure this out automatically. This option can also be used to workaround a bug in the interop itself.

— `strictEnums` and `nonStrictEnums` properties values are space-separated lists of the enums that should be generated as a Kotlin enum or as integral values correspondingly. If the enum is not included into any of these lists, then it is generated according to the heuristics.

— `noStringConversion` property value is space-separated lists of the functions whose `const char*` parameters shall not be autoconverted as Kotlin string

Portability

Sometimes the C libraries have function parameters or struct fields of a platform-dependent type, e.g. `long` or `size_t`. Kotlin itself doesn't provide neither implicit integer casts nor C-style integer casts (e.g. `(size_t) intValue`), so to make writing portable code in such cases easier, the `convert` method is provided:

```
fun ${type1}.convert<${type2}>(): ${type2}
```

where each of `type1` and `type2` must be an integral type, either signed or unsigned.

`.convert<${type}>` has the same semantics as one of the `.toByte`, `.toShort`, `.toInt`, `.toLong`, `.toUByte`, `.toUShort`, `.toUInt` or `.toULong` methods, depending on `type`.

The example of using `convert`:

```
fun zeroMemory(buffer: COpaquePointer, size: Int) {
    memset(buffer, 0, size.convert<size_t>())
}
```

Also, the type parameter can be inferred automatically and so may be omitted in some cases.

Object pinning

Kotlin objects could be pinned, i.e. their position in memory is guaranteed to be stable until unpinned, and pointers to such objects inner data could be passed to the C functions. For example

```
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
            }
            // Now `buffer` has raw data obtained from the `recv()` call.
        }
    }
}
```

Here we use service function `usePinned`, which pins an object, executes block and unpins it on normal and exception paths.

Kotlin/Native 与 Swift/Objective-C 互操作

This document covers some details of Kotlin/Native interoperability with Swift/Objective-C.

Usage

Kotlin/Native provides bidirectional interoperability with Objective-C. Objective-C frameworks and libraries can be used in Kotlin code if properly imported to the build (system frameworks are imported by default). See e.g. "Using cinterop" in Gradle plugin documentation. A Swift library can be used in Kotlin code if its API is exported to Objective-C with `@objc`. Pure Swift modules are not yet supported.

Kotlin modules can be used in Swift/Objective-C code if compiled into a framework (see "Targets and output kinds" section in Gradle plugin documentation). See calculator sample for an example.

Mappings

The table below shows how Kotlin concepts are mapped to Swift/Objective-C and vice versa.

Kotlin	Swift	Objective-C	Notes
`class`	`class`	`@interface`	note
`interface`	`protocol`	`@protocol`	
`constructor/create`	Initializer	Initializer	note
Property	Property	Property	note
Method	Method	Method	note note
`@Throws`	`throws`	`error:(NSError**)error`	note
Extension	Extension	Category member	note
`companion` member <-	Class method or property	Class method or property	
`null`	`nil`	`nil`	
`Singleton`	`Singleton()`	`[Singleton singleton]`	note
Primitive type	Primitive type / `NSNumber`		note
`Unit` return type	`Void`	`void`	
`String`	`String`	`NSString`	
`String`	`NSMutableString`	`NSMutableString`	note
`List`	`Array`	`NSArray`	
`MutableList`	`NSMutableArray`	`NSMutableArray`	
`Set`	`Set`	`NSSet`	
`MutableSet`	`NSMutableSet`	`NSMutableSet`	note
`Map`	`Dictionary`	`NSDictionary`	
`MutableMap`	`NSMutableDictionary`	`NSMutableDictionary`	note
Function type	Function type	Block pointer type	note
Suspend functions	Unsupported	Unsupported	note
Inline classes	Unsupported	Unsupported	note

Name translation

Objective-C classes are imported into Kotlin with their original names. Protocols are imported as interfaces with `Protocol` name suffix, i.e. `@protocol Foo` -> `interface FooProtocol`. These classes and interfaces are placed into a package specified in build configuration (`platform.*` packages for preconfigured system frameworks).

The names of Kotlin classes and interfaces are prefixed when imported to Objective-C. The prefix is derived from the framework name.

Initializers

Swift/Objective-C initializers are imported to Kotlin as constructors and factory methods named `create`. The latter happens with initializers declared in the Objective-C category or as a Swift extension, because Kotlin has no concept of extension constructors.

Kotlin constructors are imported as initializers to Swift/Objective-C.

Top-level functions and properties

Top-level Kotlin functions and properties are accessible as members of special classes. Each Kotlin file is translated into such a class. E.g.

```
// MyLibraryUtils.kt
package my.library

fun foo() {}
```

can be called from Swift like

```
MyLibraryUtilsKt.foo()
```

Method names translation

Generally Swift argument labels and Objective-C selector pieces are mapped to Kotlin parameter names. Anyway these two concepts have different semantics, so sometimes Swift/Objective-C methods can be imported with a clashing Kotlin signature. In this case the clashing methods can be called from Kotlin using named arguments, e.g.:

```
[player moveTo:LEFT byMeters:17]
[player moveTo:UP byInches:42]
```

in Kotlin it would be:

```
player.moveTo(LEFT, byMeters = 17)
player.moveTo(UP, byInches = 42)
```

Errors and exceptions

Kotlin has no concept of checked exceptions, all Kotlin exceptions are unchecked. Swift has only checked errors. So if Swift or Objective-C code calls a Kotlin method which throws an exception to be handled, then the Kotlin method should be marked with a `@Throws` annotation. In this case all Kotlin exceptions (except for instances of `Error`, `RuntimeException` and subclasses) are translated into a Swift error/ `NSError`.

Note that the opposite reversed translation is not implemented yet: Swift/Objective-C error-throwing methods aren't imported to Kotlin as exception-throwing.

Category members

Members of Objective-C categories and Swift extensions are imported to Kotlin as extensions. That's why these declarations can't be overridden in Kotlin. And the extension initializers aren't available as Kotlin constructors.

Kotlin singletons

Kotlin singleton (made with an `object` declaration, including `companion object`) is imported to Swift/Objective-C as a class with a single instance. The instance is available through the factory method, i.e. as `[MySingleton mySingleton]` in Objective-C and `MySingleton()` in Swift.

NSNumber

Kotlin primitive type boxes are mapped to special Swift/Objective-C classes. For example, `kotlin.Int` box is represented as `KotlinInt` class instance in Swift (or `${prefix}Int` instance in Objective-C, where `prefix` is the framework names prefix). These classes are derived from `NSNumber`, so the instances are proper `NSNumber`s supporting all corresponding operations.

`NSNumber` type is not automatically translated to Kotlin primitive types when used as a Swift/Objective-C parameter type or return value. The reason is that `NSNumber` type doesn't provide enough information about a wrapped primitive value type, i.e. `NSNumber` is statically not known to be a e.g. `Byte`, `Boolean`, or `Double`. So Kotlin primitive values should be cast to/from `NSNumber` manually (see below).

NSMutableString

`NSMutableString` Objective-C class is not available from Kotlin. All instances of `NSMutableString` are copied when passed to Kotlin.

Collections

Kotlin collections are converted to Swift/Objective-C collections as described in the table above. Swift/Objective-C collections are mapped to Kotlin in the same way, except for `NSMutableSet` and `NSMutableDictionary`. `NSMutableSet` isn't converted to a Kotlin `MutableSet`. To pass an object for Kotlin `MutableSet`, you can create this kind of Kotlin collection explicitly by either creating it in Kotlin with e.g. `mutableSetOf()`, or using the `KotlinMutableSet` class in Swift (or `${prefix}MutableSet` in Objective-C, where `prefix` is the framework names prefix). The same holds for `MutableMap`.

Function types

Kotlin function-typed objects (e.g. lambdas) are converted to Swift functions / Objective-C blocks. However there is a difference in how types of parameters and return values are mapped when translating a function and a function type. In the latter case primitive types are mapped to their boxed representation. Kotlin `Unit` return value is represented as a corresponding `Unit` singleton in Swift/Objective-C. The value of this singleton can be retrieved in the same way as it is for any other Kotlin `object` (see singletons in the table above). To sum the things up:

```
fun foo(block: (Int) -> Unit) { ... }
```

would be represented in Swift as

```
func foo(block: (KotlinInt) -> KotlinUnit)
```

and can be called like

```
foo {
    bar($0 as! Int32)
    return KotlinUnit()
}
```

Generics

Objective-C supports "lightweight generics" defined on classes, with a relatively limited feature set. Swift can import generics defined on classes to help provide additional type information to the compiler.

Generic feature support for Objc and Swift differ from Kotlin, so the translation will inevitably lose some information, but the features supported retain meaningful information.

To Use

Generics are currently not enabled by default. To have the framework header written with generics, add an experimental flag to the compiler config:

```
compilations.main {
    outputKinds("framework")
    extraOpts "-Xobjc-generics"
}
```

Limitations

Objective-C generics do not support all features of either Kotlin or Swift, so there will be some information lost in the translation.

Generics can only be defined on classes, not on interfaces (protocols in Objc and Swift) or functions.

Nullability

Kotlin and Swift both define nullability as part of the type specification, while Objc defines nullability on methods and properties of a type. As such, the following:

```
class Sample<T>(){
  fun myVal():T
}
```

will (logically) look like this:

```
class Sample<T>(){
  fun myVal():T?
}
```

In order to support a potentially nullable type, the Objc header needs to define `myVal` with a nullable return value.

To mitigate this, when defining your generic classes, if the generic type should *never* be null, provide a non-null type constraint:

```
class Sample<T:Any>(){
  fun myVal():T
}
```

That will force the Objc header to mark `myVal` as non-null.

Variance

Objective-C allows generics to be declared covariant or contravariant. Swift has no support for variance. Generic classes coming from Objective-C can be force-cast as needed.

```
data class SomeData(val num:Int = 42):BaseData()
class GenVarOut<out T:Any>(val arg:T)
```

```
let variOut = GenVarOut<SomeData>(arg: sd)
let variOutAny : GenVarOut<BaseData> = variOut as! GenVarOut<BaseData>
```

Constraints

In Kotlin you can provide upper bounds for a generic type. Objective-C also supports this, but that support is unavailable in more complex cases, and is currently not supported in the Kotlin - Objective-C interop. The exception here being a non-null upper bound will make Objective-C methods/properties non-null.

Casting between mapped types

When writing Kotlin code, an object may need to be converted from a Kotlin type to the equivalent Swift/Objective-C type (or vice versa). In this case a plain old Kotlin cast can be used, e.g.

```
val nsArray = listOf(1, 2, 3) as NSArray
val string = nsString as String
val nsNumber = 42 as NSNumber
```

Subclassing

Subclassing Kotlin classes and interfaces from Swift/Objective-C

Kotlin classes and interfaces can be subclassed by Swift/Objective-C classes and protocols.

Subclassing Swift/Objective-C classes and protocols from Kotlin

Swift/Objective-C classes and protocols can be subclassed with a Kotlin `final` class. Non-`final` Kotlin classes inheriting Swift/Objective-C types aren't supported yet, so it is not possible to declare a complex class hierarchy inheriting Swift/Objective-C types.

Normal methods can be overridden using the `override` Kotlin keyword. In this case the overriding method must have the same parameter names as the overridden one.

Sometimes it is required to override initializers, e.g. when subclassing `UIViewController`. Initializers imported as Kotlin constructors can be overridden by Kotlin constructors marked with the `@OverrideInit` annotation:

```
class ViewController : UIViewController {
    @OverrideInit constructor(coder: NSCoder) : super(coder)

    ...
}
```

The overriding constructor must have the same parameter names and types as the overridden one.

To override different methods with clashing Kotlin signatures, you can add a `@Suppress("CONFLICTING_OVERLOADS")` annotation to the class.

By default the Kotlin/Native compiler doesn't allow calling a non-designated Objective-C initializer as a `super(...)` constructor. This behaviour can be inconvenient if the designated initializers aren't marked properly in the Objective-C library. Adding a `disableDesignatedInitializerChecks = true` to the `.def` file for this library would disable these compiler checks.

C features

See INTEROP.md for an example case where the library uses some plain C features (e.g. unsafe pointers, structs etc.).

Unsupported

Some features of Kotlin programming language are not yet mapped into respective features of Objective-C or Swift. Currently, following features are not properly exposed in generated framework headers:

— suspend functions
— inline classes (arguments are mapped as either underlying primitive type or `id`)
— custom classes implementing standard Kotlin collection interfaces (`List`, `Map`, `Set`) and other special classes
— Kotlin subclasses of Objective-C classes

CocoaPods integration

Starting with 1.3.30, an experimental integration with CocoaPods is added to Kotlin/Native. This feature allows you to represent a Kotlin/Native Gradle-project as a CocoaPods dependency. Such a representation provides the following advantages:

— Such a dependency can be included in a Podfile of an Xcode project and automatically built (and rebuilt) along with this project. As a result, importing to Xcode is simplified since there is no need to write corresponding Gradle tasks and Xcode build steps manually.

— When building from Xcode, you can use CocoaPods libraries without writing .def files manually and setting cinterop tool parameters. In this case, all required parameters can be obtained from the Xcode project configured by CocoaPods.

For an example of CocoaPods integration, refer to the `cocoapods` sample.

CocoaPods Gradle plugin

The CocoaPods support is implemented in a separate Gradle plugin: `org.jetbrains.kotlin.native.cocoapods`.

> **Note:** The plugin is based on the multiplatform project model and requires applying the `org.jetbrains.kotlin.multiplatform` plugin. See details about the multiplatform plugin at the corresponding page.

When applied, the CocoaPods plugin does the following:

1. Adds both `debug` and `release` frameworks as output binaries for all iOS and macOS targets.
2. Creates a `podspec` task which generates a podspec file for the given project.

The podspec generated includes a path to an output framework and script phases which automate building this framework during a build process of an Xcode project. Some fields of the podspec file can be configured using the `kotlin.cocoapods { ... }` code block.

```
// Apply plugins.
plugins {
    id("org.jetbrains.kotlin.multiplatform") version "1.3.30"
    id("org.jetbrains.kotlin.native.cocoapods") version "1.3.30"
}

// CocoaPods requires the podspec to have a version.
version = "1.0"

kotlin {
    cocoapods {
        // Configure fields required by CocoaPods.
        summary = "Some description for a Kotlin/Native module"
        homepage = "Link to a Kotlin/Native module homepage"
    }
}
```

The following podspec fields are required by CocoaPods:

— `version`

— `summary`

— `homepage`

A version of the Gradle project is used as a value for the `version` field. Fields `summary` and `homepage` can be configured using the `cocoapods` code block.

This podspec file can be referenced from a Podfile of an Xcode project. After that the framework built from the Kotlin/Native module can be used from this Xcode project. If necessary, this framework is automatically rebuilt during Xcode build process.

Workflow

To import a Kotlin/Native module in an existing Xcode project:

1. Make sure that you have CocoaPods installed. We recommend using CocoaPods 1.6.1 or later.
2. Configure a Gradle project: apply the `org.jetbrains.kotlin.native.cocoapods` plugin, add and configure the targets, and specify the required podspec fields.
3. Run the `podspec` task. The podspec file described above will be generated.

   > In order to avoid compatibility issues during an Xcode build, the plugin requires using a Gradle wrapper. To generate the wrapper automatically during execution of the podspec task, run it with the parameter `-Pkotlin.native.cocoapods.generate.wrapper=true`.

4. Add a reference to the generated podspec in a Podfile of the Xcode project.

   ```
   target 'my-ios-app' do
       pod 'my_kotlin_library', :path => 'path/to/my-kotlin-library'
   end
   ```

5. Run `pod install` for the Xcode project.

After completing these steps, you can open the generated workspace (see CocoaPods documentation) and run an Xcode build.

Interoperability

The CocoaPods plugin also allows using CocoaPods libraries without manual configuring cinterop parameters (see the corresponding section of the multiplatform plugin documentation). The `cocoapods { ... }` code block allows you to add dependencies on CocoaPods libraries.

```
kotlin {
    cocoapods {
        // Configure a dependency on AFNetworking.
        // The CocoaPods version notation is supported.
        // The dependency will be added to all macOS and iOS targets.
        pod("AFNetworking", "~> 3.2.0")
    }
}
```

To use these dependencies from a Kotlin code, import a package `cocoapods.<library-name>`. In the example above, it's `cocoapods.AFNetworking`.

The dependencies declared in this way are added in the podspec file and downloaded during the execution of `pod install`.

Important: To correctly import the dependencies into the Kotlin/Native module, the Podfile must contain either `use_modular_headers!` or `use_frameworks!` directive.

Search paths for libraries added in the Kotlin/Native module in this way are obtained from properties of the Xcode projects configured by CocoaPods. Thus if the module uses CocoaPods libraries, it can be build **only from Xcode**.

Current Limitations

— If a Kotlin/Native module uses a CocoaPods library, you can build this module only from an Xcode project. Otherwise the CocoaPods library cannot be resolved by the Kotlin/Native infrastructure.

— Subspecs are not supported.

Kotlin/Native Gradle 插件

Since 1.3.40, a separate Gradle plugin for Kotlin/Native is deprecated in favor of the `kotlin-multiplatform` plugin. This plugin provides an IDE support along with support of the new multiplatform project model introduced in Kotlin 1.3.0. Below you can find a short list of differences between `kotlin-platform-native` and `kotlin-muliplatform` plugins. For more information see the `kotlin-muliplatform` documentation page. For `kotlin-platform-native` reference see the corresponding section.

Applying the multiplatform plugin

To apply the `kotlin-multiplatform` plugin, just add the following snippet into your build script:

```
plugins {
    id("org.jetbrains.kotlin.multiplatform") version '1.3.40'
}
```

Managing targets

With the `kotlin-platform-native` plugin a set of target platforms is specified as a list in properties of the main component:

```
components.main {
    targets = ['macos_x64', 'linux_x64', 'mingw_x64']
}
```

With the `kotlin-multiplatform` plugin target platforms can be added into a project using special methods available in the `kotlin` extension. Each method adds into a project one **target** which can be accessed using the `targets` property. Each target can be configured independently including output kinds, additional compiler options etc. See details about targets at the corresponding page.

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

kotlin {
    // These targets are declared without any target-specific settings.
    macosX64()
    linuxX64()

    // You can specify a custom name used to access the target.
    mingwX64("windows")

    iosArm64 {
        // Additional settings for ios_arm64.
    }

    // You can access declared targets using the `targets` property.
    println(targets.macosX64)
    println(targets.windows)

    // You also can configure all native targets in a single block.
    targets.withType(KotlinNativeTarget) {
        // Native target configuration.
    }
}
```

Each target includes two **compilations**: `main` and `test` compiling product and test sources respectively. A compilation is an abstraction over a compiler invocation and described at the corresponding page.

Managing sources

With the `kotlin-platform-native` plugin source sets are used to separate test and product sources. Also you can specify different sources for different platforms in the same source set:

```
sourceSets {
    // Adding target-independent sources.
    main.kotlin.srcDirs += 'src/main/mySources'

    // Adding Linux-specific code.
    main.target('linux_x64').srcDirs += 'src/main/linux'
}
```

With the `kotlin-multiplatform` plugin **source sets** are also used to group sources but source files for different platforms are located in different source sets. For each declared target two source sets are created: `<target-name>Main` and `<target-name>Test` containing product and test sources for this platform. Common for all platforms sources are located in `commonMain` and `commonTest` source sets created by default. More information about source sets can be found here.

```
kotlin {
    sourceSets {
        // Adding target-independent sources.
        commonMain.kotlin.srcDirs += file("src/main/mySources")

        // Adding Linux-specific code.
        linuxX64Main.kotlin.srcDirs += file("src/main/linux")
    }
}
```

Managing dependencies

With the `kotlin-platform-native` plugin dependencies are configured in a traditional for Gradle way by grouping them into configurations using the project `dependencies` block:

```
dependencies {
    implementation 'org.sample.test:mylibrary:1.0'
    testImplementation 'org.sample.test:testlibrary:1.0'
}
```

The `kotlin-multiplatform` plugin also uses configurations under the hood but it also provides a `dependencies` block for each source set allowing configuring dependencies of this sources set:

```
kotlin.sourceSets {
    commonMain {
        dependencies {
            implementation("org.sample.test:mylibrary:1.0")
        }
    }

    commonTest {
        dependencies {
            implementation("org.sample.test:testlibrary:1.0")
        }
    }
}
```

Note that a module referenced by a dependency declared for `commonMain` or `commonTest` source set must be published using the `kotlin-multiplatform` plugin. If you want to use libraries published by the `kotlin-platform-native` plugin, you need to declare a separate source set for common native sources.

```
kotlin.sourceSets {
    // Create a common source set used by native targets only.
    nativeMain {
        dependsOn(commonMain)
        dependencies {
            // Depend on a library published by the kotlin-platform-naive plugin.
            implementation("org.sample.test:mylibrary:1.0")
        }
    }

    // Configure all native platform sources sets to use it as a common one.
    linuxX64Main.dependsOn(nativeMain)
    macosX64Main.dependsOn(nativeMain)
    //...
}
```

See more info about dependencies at the corresponding page.

Output kinds

With the `kotlin-platform-native` plugin output kinds are specified as a list in properties of a component:

```
components.main {
    // Compile the component into an executable and a Kotlin/Native library.
    outputKinds = [EXECUTABLE, KLIBRARY]
}
```

With the `kotlin-multiplatform` plugin a compilation always produces a `*.klib` file. A separate `binaries` block is used to configure what final native binaries should be produced by each target. Each binary can be configured independently including linker options, executable entry point etc.

```
kotlin {
    macosX64 {
        binaries {
            executable {
                // Binary configuration: linker options, name, etc.
            }
            framework {
                // ...
            }

        }
    }
}
```

See more about native binaries declaration at the corresponding page.

Publishing

Both `kotlin-platform-native` and `kotlin-multiplatform` plugins automatically set up artifact publication when the `maven-publish` plugin is applied. See details about publication at the corresponding page. Note that currently only Kotlin/Native libraries (`*.klib`) can be published for native targets.

Cinterop support

With the `kotlin-platform-native` plugin interop with a native library can be declared in component dependencies:

```
components.main {
    dependencies {
        cinterop('mystdio') {
            // Cinterop configuration.
        }
    }
}
```

With the `kotlin-multiplatform` plugin interops are configured as a part of a compilation (see details here). The rest of an interop configuration is the same as for the `kotlin-platform-native` plugin.

```
kotlin {
    macosX64 {
        compilations.main.cinterops {
            mystdio {
                // Cinterop configuration.
            }
        }
    }
}
```

`kotlin-platform-native` reference

Overview

You may use the Gradle plugin to build *Kotlin/Native* projects. Builds of the plugin are available at the Gradle plugin portal, so you can apply it using Gradle plugin DSL:

```
plugins {
    id "org.jetbrains.kotlin.platform.native" version "1.3.0-rc-146"
}
```

You also can get the plugin from a Bintray repository. In addition to releases, this repo contains old and development versions of the plugin which are not available at the plugin portal. To get the plugin from the Bintray repo, include the following snippet in your build script:

```
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'
```

By default the plugin downloads the Kotlin/Native compiler during the first run. If you have already downloaded the compiler manually you can specify the path to its root directory using `org.jetbrains.kotlin.native.home` project property (e.g. in `gradle.properties`).

```
org.jetbrains.kotlin.native.home=/home/user/kotlin-native-0.8
```

In this case the compiler will not be downloaded by the plugin.

Source management

Source management in the `kotlin.platform.native` plugin is uniform with other Kotlin plugins and is based on source sets. A source set is a group of Kotlin/Native source which may contain both common and platform-specific code. The plugin provides a top-level script block `sourceSets` allowing you to configure source sets. Also it creates the default source sets `main` and `test` (for production and test code respectively).

By default the production sources are located in `src/main/kotlin` and the test sources - in `src/test/kotlin`.

```
sourceSets {
    // Adding target-independent sources.
    main.kotlin.srcDirs += 'src/main/mySources'

    // Adding Linux-specific code. It will be compiled in Linux binaries only.
    main.target('linux_x64').srcDirs += 'src/main/linux'
}
```

Targets and output kinds

By default the plugin creates software components for the main and test source sets. You can access them via the `components` container provided by Gradle or via the `component` property of a corresponding source set:

```
// Main component.
components.main
sourceSets.main.component

// Test component.
components.test
sourceSets.test.component
```

Components allow you to specify:

— Targets (e.g. Linux/x64 or iOS/arm64 etc)

— Output kinds (e.g. executable, library, framework etc)

— Dependencies (including interop ones)

Targets can be specified by setting a corresponding component property:

```
components.main {
    // Compile this component for 64-bit MacOS, Linux and Windows.
    targets = ['macos_x64', 'linux_x64', 'mingw_x64']
}
```

The plugin uses the same notation as the compiler. By default, test component uses the same targets as specified for the main one.

Output kinds can also be specified using a special property:

```
components.main {
    // Compile the component into an executable and a Kotlin/Native library.
    outputKinds = [EXECUTABLE, KLIBRARY]
}
```

All constants used here are available inside a component configuration script block. The plugin supports producing binaries of the following kinds:

— `EXECUTABLE` - an executable file;

— `KLIBRARY` - a Kotlin/Native library (*.klib);

— `FRAMEWORK` - an Objective-C framework;

— `DYNAMIC` - shared native library;

— `STATIC` - static native library.

Also each native binary is built in two variants (build types): `debug` (debuggable, not optimized) and `release` (not debuggable, optimized). Note that Kotlin/Native libraries have only `debug` variant because optimizations are preformed only during compilation of a final binary (executable, static lib etc) and affect all libraries used to build it.

Compile tasks

The plugin creates a compilation task for each combination of the target, output kind, and build type. The tasks have the following naming convention:

`compile<ComponentName><BuildType><OutputKind><Target>KotlinNative`

For example `compileDebugKlibraryMacos_x64KotlinNative`, `compileTestDebugKotlinNative`.

The name contains the following parts (some of them may be empty):

— `<ComponentName>` - name of a component. Empty for the main component.

— `<BuildType>` - `Debug` or `Release`.

— `<OutputKind>` - output kind name, e.g. `Executabe` or `Dynamic`. Empty if the component has only one output kind.

— `<Target>` - target the component is built for, e.g. `Macos_x64` or `Wasm32`. Empty if the component is built only for one target.

Also the plugin creates a number of aggregate tasks allowing you to build all the binaries for a build type (e.g. `assembleAllDebug`) or all the binaries for a particular target (e.g. `assembleAllWasm32`).

Basic lifecycle tasks like `assemble`, `build`, and `clean` are also available.

Running tests

The plugin builds a test executable for all the targets specified for the `test` component. If the current host platform is included in this list the test running tasks are also created. To run tests, execute the standard lifecycle `check` task:

```
./gradlew check
```

Dependencies

The plugin allows you to declare dependencies on files and other projects using traditional Gradle's mechanism of configurations. The plugin supports Kotlin multiplatform projects allowing you to declare the `expectedBy` dependencies

```
dependencies {
    implementation files('path/to/file/dependencies')
    implementation project('library')
    testImplementation project('testLibrary')
    expectedBy project('common')
}
```

It's possible to depend on a Kotlin/Native library published earlier in a maven repo. The plugin relies on Gradle's metadata support so the corresponding feature must be enabled. Add the following line in your `settings.gradle`:

```
enableFeaturePreview('GRADLE_METADATA')
```

Now you can declare a dependency on a Kotlin/Native library in the traditional `group:artifact:version` notation:

```
dependencies {
    implementation 'org.sample.test:mylibrary:1.0'
    testImplementation 'org.sample.test:testlibrary:1.0'
}
```

Dependency declaration is also possible in the component block:

```
components.main {
    dependencies {
        implementation 'org.sample.test:mylibrary:1.0'
    }
}

components.test {
    dependencies {
        implementation 'org.sample.test:testlibrary:1.0'
    }
}
```

Using cinterop

It's possible to declare a cinterop dependency for a component:

```
components.main {
    dependencies {
        cinterop('mystdio') {
            // src/main/c_interop/mystdio.def is used as a def file.

            // Set up compiler options
            compilerOpts '-I/my/include/path'

            // It's possible to set up different options for different targets
            target('linux') {
                compilerOpts '-I/linux/include/path'
            }
        }
    }
}
```

Here an interop library will be built and added in the component dependencies.

Often it's necessary to specify target-specific linker options for a Kotlin/Native binary using an interop. It can be done using the `target` script block:

```
components.main {
    target('linux') {
        linkerOpts '-L/path/to/linux/libs'
    }
}
```

Also the `allTargets` block is available.

```
components.main {
    // Configure all targets.
    allTargets {
        linkerOpts '-L/path/to/libs'
    }
}
```

Publishing

In the presence of `maven-publish` plugin the publications for all the binaries built are created. The plugin uses Gradle metadata to publish the artifacts so this feature must be enabled (see the dependencies section).

Now you can publish the artifacts with the standard Gradle `publish` task:

```
./gradlew publish
```

Only `EXECUTABLE` and `KLIBRARY` binaries are published currently.

The plugin allows you to customize the pom generated for the publication with the `pom` code block available for every component:

```
components.main {
    pom {
        withXml {
            def root = asNode()
            root.appendNode('name', 'My library')
            root.appendNode('description', 'A Kotlin/Native library')
        }
    }
}
```

Serialization plugin

The plugin is shipped with a customized version of the `kotlinx.serialization` plugin. To use it you don't have to add new buildscript dependencies, just apply the plugins and add a dependency on the serialization library:

```
apply plugin: 'org.jetbrains.kotlin.platform.native'
apply plugin: 'kotlinx-serialization-native'

dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-serialization-runtime-native'
}
```

The the example project for details.

DSL example

In this section a commented DSL is shown. See also the example projects that use this plugin, e.g. Kotlinx.coroutines, MPP http client

```
plugins {
    id "org.jetbrains.kotlin.platform.native" version "1.3.0-rc-146"
}

sourceSets.main {
    // Plugin uses Gradle's source directory sets here,
    // so all the DSL methods available in SourceDirectorySet can be called here.
    // Platform independent sources.
    kotlin.srcDirs += 'src/main/customDir'

    // Linux-specific sources
    target('linux').srcDirs += 'src/main/linux'
}

components.main {

    // Set up targets
    targets = ['linux_x64', 'macos_x64', 'mingw_x64']

    // Set up output kinds
    outputKinds = [EXECUTABLE, KLIBRARY, FRAMEWORK, DYNAMIC, STATIC]

    // Specify custom entry point for executables
    entryPoint = "org.test.myMain"

    // Target-specific options
    target('linux_x64') {
```

```
    target('linux_x64') {
        linkerOpts '-L/linux/lib/path'
    }

    // Targets independent options
    allTargets {
        linkerOpts '-L/common/lib/path'
    }

    dependencies {

        // Dependency on a published Kotlin/Native library.
        implementation 'org.test:mylib:1.0'

        // Dependency on a project
        implementation project('library')

        // Cinterop dependency
        cinterop('interop-name') {
            // Def-file describing the native API.
            // The default path is src/main/c_interop/<interop-name>.def
            defFile project.file("deffile.def")

            // Package to place the Kotlin API generated.
            packageName 'org.sample'

            // Options to be passed to compiler and linker by cinterop tool.
            compilerOpts 'Options for native stubs compilation'
            linkerOpts 'Options for native stubs'

            // Additional headers to parse.
            headers project.files('header1.h', 'header2.h')

            // Directories to look for headers.
            includeDirs {
                // All objects accepted by the Project.file method may be used with both options.

                // Directories for header search (an analogue of the -I<path> compiler option).
                allHeaders 'path1', 'path2'

                // Additional directories to search headers listed in the 'headerFilter' def-file
option.
                // -headerFilterAdditionalSearchPrefix command line option analogue.
                headerFilterOnly 'path1', 'path2'
            }
            // A shortcut for includeDirs.allHeaders.
            includeDirs "include/directory" "another/directory"

            // Pass additional command line options to the cinterop tool.
            extraOpts '-verbose'

            // Additional configuration for Linux.
            target('linux') {
                compilerOpts 'Linux-specific options'
            }
        }
    }

    // Additional pom settings for publication.
    pom {
        withXml {
            def root = asNode()
            root.appendNode('name', 'My library')
            root.appendNode('description', 'A Kotlin/Native library')
        }
    }

    // Additional options passed to the compiler.
    extraOpts '-w -time'
```

```
    extraopts "--time"
}
```

调试

Currently the Kotlin/Native compiler produces debug info compatible with the DWARF 2 specification, so modern debugger tools can perform the following operations:

— breakpoints
— stepping
— inspection of type information
— variable inspection

Producing binaries with debug info with Kotlin/Native compiler

To produce binaries with the Kotlin/Native compiler it's sufficient to use the `-g` option on the command line.

Example:

```
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)
```

Breakpoints

Modern debuggers provide several ways to set a breakpoint, see below for a tool-by-tool breakdown:

lldb

— by name

```
(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 is optional, this flag is applied by default

— by location (filename, line number)

```
(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
```

— by address

```
(lldb) b -a 0x00000001000012e4
Breakpoint 2: address = 0x00000001000012e4
```

— by regex, you might find it useful for debugging generated artifacts, like lambda etc. (where used `#` symbol in name).

```
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

— by regex

```
(gdb) rbreak main(
Breakpoint 1 at 0x1000109b4
struct ktype:kotlin.Unit &kfun:main(kotlin.Array<kotlin.String>);
```

— by name **unusable**, because `:` is a separator for the breakpoint by location

```
(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
```

— by location

```
(gdb) b hello.kt:1
Breakpoint 2 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 1.
```

— by address

```
(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.
```

Stepping

Stepping functions works mostly the same way as for C/C++ programs

Variable inspection

Variable inspections for var variables works out of the box for primitive types. For non-primitive types there are custom pretty printers for lldb in `konan_lldb.py`:

```
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)
```

Getting representation of the object variable (var) could also be done using the built-in runtime function `Konan_DebugPrint` (this approach also works for gdb, using a module of command syntax):

```
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)
```

Known issues

— performance of Python bindings.

Note: Supporting the DWARF 2 specification means that the debugger tool recognizes Kotlin as C89, because before the DWARF 5 specification, there is no identifier for the Kotlin language type in specification.

Q: How do I run my program?

A: Define a top level function `fun main(args: Array<String>)` or just `fun main()` if you are not interested in passed arguments, please ensure it's not in a package. Also compiler switch `-entry` could be used to make any function taking `Array<String>` or no arguments and return `Unit` as an entry point.

Q: What is Kotlin/Native memory management model?

A: Kotlin/Native provides an automated memory management scheme, similar to what Java or Swift provides. The current implementation includes an automated reference counter with a cycle collector to collect cyclical garbage.

Q: How do I create a shared library?

A: Use the `-produce dynamic` compiler switch, or `binaries.sharedLib()` in Gradle, i.e.

```
targets {
    fromPreset(presets.iosArm64, 'mylib') {
        binaries.sharedLib()
    }
}
```

It will produce a platform-specific shared object (.so on Linux, .dylib on macOS, and .dll on Windows targets) and a C language header, allowing the use of all public APIs available in your Kotlin/Native program from C/C++ code. See `samples/python_extension` for an example of using such a shared object to provide a bridge between Python and Kotlin/Native.

Q: How do I create a static library or an object file?

A: Use the `-produce static` compiler switch, or `binaries.staticLib()` in Gradle, i.e.

```
targets {
    fromPreset(presets.iosArm64, 'mylib') {
        binaries.staticLib()
    }
}
```

It will produce a platform-specific static object (.a library format) and a C language header, allowing you to use all the public APIs available in your Kotlin/Native program from C/C++ code.

Q: How do I run Kotlin/Native behind a corporate proxy?

A: As Kotlin/Native needs to download a platform specific toolchain, you need to specify `-Dhttp.proxyHost=xxx -Dhttp.proxyPort=xxx` as the compiler's or `gradlew` arguments, or set it via the `JAVA_OPTS` environment variable.

Q: How do I specify a custom Objective-C prefix/name for my Kotlin framework?

A: Use the `-module-name` compiler option or matching Gradle DSL statement, i.e.

```
targets {
    fromPreset(presets.iosArm64, 'myapp') {
        binaries.framework()
        compilations.main.extraOpts '-module-name', 'TheName'
    }
}
```

Q: How do I rename the iOS framework? (default name is *<project name>*.framework)

A: Use the `baseName` option. This will also set the module name.

```
 targets {
     fromPreset(presets.iosArm64, 'myapp') {
        binaries {
           framework {
               baseName = "TheName"
           }
        }
     }
 }
```

Q: How do I enable bitcode for my Kotlin framework?

A: By default gradle plugin adds it on iOS target.

— For debug build it embeds placeholder LLVM IR data as a marker.

— For release build it embeds bitcode as data.

Or commandline arguments: `-Xembed-bitcode` (for release) and `-Xembed-bitcode-marker` (debug)

Setting this in a Gradle DSL:

```
targets {
    fromPreset(presets.iosArm64, 'myapp') {
        binaries {
            framework {
                // Use "marker" to embed the bitcode marker (for debug builds).
                // Use "disable" to disable embedding.
                embedBitcode "bitcode" // for release binaries.
            }
        }
    }
}
```

These options have nearly the same effect as clang's `-fembed-bitcode` / `-fembed-bitcode-marker` and swiftc's `-embed-bitcode` / `-embed-bitcode-marker`.

Q: Why do I see `InvalidMutabilityException`?

A: It likely happens, because you are trying to mutate a frozen object. An object can transfer to the frozen state either explicitly, as objects reachable from objects on which the `kotlin.native.concurrent.freeze` is called, or implicitly (i.e. reachable from `enum` or global singleton object - see the next question).

Q: How do I make a singleton object mutable?

A: Currently, singleton objects are immutable (i.e. frozen after creation), and it's generally considered good practise to have the global state immutable. If for some reason you need a mutable state inside such an object, use the `@konan.ThreadLocal` annotation on the object. Also the `kotlin.native.concurrent.AtomicReference` class could be used to store different pointers to frozen objects in a frozen object and automatically update them.

Q: How can I compile my project against the Kotlin/Native master?

A: One of the following should be done:

- For the CLI, you can compile using gradle as stated in the README (and if you get errors, you can try to do a `./gradlew clean`):
- For Gradle, you can use Gradle composite builds like this:

工具

使用 Gradle

为了用 Gradle 构建 Kotlin 项目，需要设置好 *kotlin-gradle* 插件，将其应用到你的项目中，并且添加 *kotlin-stdlib* 依赖。这些操作也可以在 IntelliJ IDEA 中通过调用 **Project action** 中的 **Tools | Kotlin | Configure Kotlin** 自动执行。

插件与版本

使用 Gradle 插件 DSL 应用 Kotlin Gradle 插件。Kotlin Gradle 插件 1.3.72 适用于 Gradle 4.9 及更高版本。

```
plugins {
    id 'org.jetbrains.kotlin.<……>' version '1.3.72'
}
```

```
plugins {
    kotlin("<……>") version "1.3.72"
}
```

需要将其中的占位符 `<……>` 替换为可在后续部分中找到的插件名之一。

或者通过将依赖项 `kotlin-gradle-plugin` 添加到构建脚本类路径来应用插件：

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.3.72"
    }
}

plugins {
    id "org.jetbrains.kotlin.<……>" version "1.3.72"
}
```

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath(kotlin("gradle-plugin", version = "1.3.72"))
    }
}
plugins {
    kotlin("<……>")
}
```

当通过 Gradle 插件 DSL 与 Gradle Kotlin DSL 使用 Kotlin Gradle 插件 1.1.1 及以上版本时，这不是必需的。

构建 Kotlin 多平台项目

使用 `kotlin-multiplatform` 插件构建多平台项目在以 Gradle 构建多平台项目中详述。

面向 JVM

如需面向 JVM 平台，请应用 Kotlin JVM 插件。自 Kotlin 1.1.1 起，可以使用 Gradle 插件 DSL 来应用该插件：

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.3.72"
}
```

```
plugins {
    kotlin("jvm") version "1.3.72"
}
```

在这个代码块中的 `version` 应该是字面值，并且不能在其他构建脚本中应用。

或者使用旧版 `apply plugin` 方式：

```
apply plugin: 'kotlin'
```

不建议在 Gradle Kotlin DSL 中以 `apply` 的方式应用 Kotlin 插件。详见下文。

Kotlin 源代码可以与同一个文件夹或不同文件夹中的 Java 源代码混用。默认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，那么应该更新相应的 *sourceSets* 属性：

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

```
sourceSets.main {
    java.srcDirs("src/main/myJava","src/main/myKotlin")}
}
```

面向 JavaScript

当面向 JavaScript 时，须应用不同的插件：

```
plugins {
    id 'org.jetbrains.kotlin.js' version '1.3.72'
}
```

```
plugins {
    kotlin("js") version "1.3.72"
}
```

这个插件只适用于 Kotlin 文件，因此建议将 Kotlin 和 Java 文件分开（如果是同一项目包含 Java 文件的情况）。与面向 JVM 一样，如果不使用默认约定，需要使用 *sourceSets* 来指定源代码文件夹：

```
kotlin {
    sourceSets {
        main.kotlin.srcDirs += 'src/main/myKotlin'
    }
}
```

```
kotlin {
    sourceSets["main"].apply {
        kotlin.srcDir("src/main/myKotlin")
    }
}
```

面向 Android

Android 的 Gradle 模型与普通 Gradle 有点不同，所以如果我们要构建一个用 Kotlin 编写的 Android 项目，我们需要用 *kotlin-android* 插件取代 *kotlin* 插件：

```
buildscript {
    ext.kotlin_version = '1.3.72'

    ……

    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 = "1.3.72"))
    }
}
plugins {
    id("com.android.application")
    kotlin("android")
}
```

Kotlin Gradle 插件 1.3.72 适用于 Android Gradle 插件 3.0 及更高版本。

不要忘记配置标准库依赖关系。

Android Studio

如果使用 Android Studio，那么需要在 android 下添加以下内容：

```
android {
  ……

  sourceSets {
    main.java.srcDirs += 'src/main/kotlin'
  }
}
```

```
android {
  ……

    sourceSets["main"].java.srcDir("src/main/kotlin")
}
```

这让 Android Studio 知道该 kotlin 目录是源代码根目录，所以当项目模型加载到 IDE 中时，它会被正确识别。或者，你可以将 Kotlin 类放在 Java 源代码目录中，该目录通常位于 `src/main/java`。

配置依赖

除了上面显示的 `kotlin-gradle-plugin` 依赖之外，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` 面向 Java 6 及以上版本。还有扩展版本的标准库，增加了对 JDK 7 与 JDK 8 中某些特性支持。如需使用这些版本，请添加下列依赖之一而不是 `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 中，请使用 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8`。

如果是面向 JavaScript，请使用依赖项 `stdlib-js` 。

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-js"
```

```
implementation(kotlin("stdlib-js"))
```

如果项目中用到了 Kotlin 反射或者测试设施，还需要添加相应的依赖项：

```
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"))
```

自 Kotlin 1.1.2 起，使用 `org.jetbrains.kotlin` group 的依赖项默认使用从已应用的插件获得的版本来解析。你可以用完整的依赖关系助记符：

```
implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
```

```
implementation(kotlin("stdlib", kotlinVersion))
```

注解处理

Kotlin 通过 *Kotlin 注解处理工具*（`kapt`）支持注解处理。kapt 同 Gradle 合用的用法已在 kapt 页描述。

增量编译

Kotlin Gradle 插件支持支持增量编译。增量编译会跟踪多次构建之间源文件的变更，因此只会编译这些变更所影响的文件。

Kotlin/JVM 与 Kotlin/JS 项目均支持增量编译。对于 Kotlin 1.1.1 起的 Kotlin/JVM 项目以及自 Kotlin 1.3.20 起的 Kotlin/JS 项目默认启用增量编译。

有几种方法可以覆盖默认设置：

- 在 Gradle 配置文件中：在 `gradle.properties` 或者 `local.properties` 中，对于 Kotlin/JVM 项目添加一行 `kotlin.incremental=＜值＞`，对于 Kotlin/JS 项目添加一行 `kotlin.incremental.js=＜值＞`。`＜值＞` 是一个反应增量编译用法的布尔值。
- 在 Gradle 命令行参数中：添加带有反应增量编译用法的布尔值的 `-Pkotlin.incremental` or `-Pkotlin.incremental.js` 参数。请注意，这样用法中，该参数必须添加到后续每个子构建，并且任何具有禁用增量编译的构建将使增量缓存失效。

请注意，任何情况下首次构建都不会是增量的。

Gradle 构建缓存支持（自 1.2.20 起）

Kotlin 插件支持 Gradle 构建缓存（需要 Gradle 4.3 及以上版本；低版本则禁用缓存）。

如需禁用所有 Kotlin 任务的缓存，请将系统属性标志 `kotlin.caching.enabled` 设置为 `false`（运行构建带上参数 `-Dkotlin.caching.enabled=false`）。

如果使用 kapt，请注意默认情况下不会缓存注解处理任务。不过，可以手动为它们启用缓存。详见 kapt 页。

编译器选项

要指定附加的编译选项，请使用 Kotlin 编译任务的 `kotlinOptions` 属性。

当面向 JVM 时，对于生产代码这些任务称为 `compileKotlin` 而对于测试代码称为 `compileTestKotlin`。对于自定义源文件集（source set）这些任务称呼取决于 `compile＜Name＞Kotlin` 模式。

Android 项目中的任务名称包含构建变体 名称，并遵循 `compile<BuildVariant>Kotlin` 的模式，例如 `compileDebugKotlin`、`compileReleaseUnitTestKotlin`。

当面向 JavaScript 时，这些任务分别称为 `compileKotlin2Js` 与 `compileTestKotlin2Js`，以及对于自定义源文件集称为 `compile＜Name＞Kotlin2Js`。

要配置单个任务，请使用其名称。示例：

```
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
```

请注意，对于 Gradle Kotlin DSL，首先从项目的 `tasks` 中获取任务。

相应地，为 JS 与 Common 目标使用类型 `Kotlin2JsCompile` 与 `KotlinCompileCommon`。

也可以在项目中配置所有 Kotlin 编译任务：

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile).all {
    kotlinOptions { ... }
}
```

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

tasks.withType<KotlinCompile>().configureEach {
    kotlinOptions.suppressWarnings = true
}
```

对于 Gradle 任务的完整选项列表如下：

JVM、JS 与 JS DCE 的公共属性

名称	描述	可能的值	默认值
`allWarningsAsErrors`	任何警告都报告为错误		false
`suppressWarnings`	不生成警告		false
`verbose`	启用详细日志输出		false
`freeCompilerArgs`	附加编译器参数的列表		[]

JVM 与 JS 的公共属性

Name	Description	Possible values	Default value
`apiVersion`	只允许使用来自捆绑库的指定版本中的声明	"1.0"、"1.1"、"1.2"、"1.3"、"1.4 (EXPERIMENTAL)"	
`languageVersion`	提供与指定 Kotlin 版本源代码级兼容	"1.0"、"1.1"、"1.2"、"1.3"、"1.4 (EXPERIMENTAL)"	

JVM 特有的属性

名称	描述	可能的值	默认值
`javaParameters`	为方法参数生成 Java 1.8 反射的元数据		false
`jdkHome`	将来自指定位置的自定义 JDK 而不是默认的 JAVA_HOME 包含到类路径中		
`jvmTarget`	生成的 JVM 字节码的目标版本（1.6、1.8、9、10、11、12 或 13），默认为 1.6	"1.6"、"1.8"、"9"、"10"、"11"、"12"、"13"	"1.6"
`noJdk`	不要自动在类路径中包含 Java 运行时		false
`noReflect`	不要自动在类路径中包含 Kotlin 反射实现		true
`noStdlib`	不要自动在类路径中包含 Kotlin 运行时与 Kotlin 反射		true

JS 特有的属性

名称	描述	可能的值	默认值
`friendModulesDisabled`	禁用内部声明导出		false
`main`	定义是否在执行时调用 `main` 函数	"call"、"noCall"	"call"
`metaInfo`	使用元数据生成 .meta.js 与 .kjsm 文件。用于创建库		true
`moduleKind`	编译器生成的 JS 模块类型	"plain"、"amd"、"commonjs"、"umd"	"plain"
`noStdlib`	不要自动将默认的 Kotlin/JS stdlib 包含到编译依赖项中		true
`outputFile`	编译结果的目标 *.js 文件		
`sourceMap`	生成源代码映射（source map）		false
`sourceMapEmbedSources`	将源代码嵌入到源代码映射中	"never"、"always"、"inlining"	
`sourceMapPrefix`	将指定前缀添加到源代码映射中的路径		
`target`	生成指定 ECMA 版本的 JS 文件	"v5"	"v5"
`typedArrays`	将原生数组转换为 JS 带类型数组		true

生成文档

要生成 Kotlin 项目的文档，请使用 Dokka；相关配置说明请参见 Dokka README 。Dokka 支持混合语言项目，并且可以生成多种格式的输出，包括标准 JavaDoc。

OSGi

关于 OSGi 支持请参见 Kotlin OSGi 页。

使用 Gradle Kotlin DSL

使用 Gradle Kotlin DSL 时，请使用 `plugins { …… }` 块应用 Kotlin 插件。如果使用 `apply { plugin(……) }` 来应用的话，可能会遇到未解析的到由 Gradle Kotlin DSL 所生成扩展的引用问题。为了解决这个问题，可以注释掉出错的用法，运行 Gradle 任务 `kotlinDslAccessorsSnapshot`，然后解除该用法注释并重新运行构建或者重新将项目导入到 IDE 中。

示例

以下示例显示了配置 Gradle 插件的不同可能性：

- Kotlin
- 混用 Java 与 Kotlin
- Android
- JavaScript

使用 Maven

插件与版本

kotlin-maven-plugin 用于编译 Kotlin 源代码与模块，目前只支持 Maven V3。

通过 *kotlin.version* 属性定义要使用的 Kotlin 版本：

```
<properties>
    <kotlin.version>1.3.72</kotlin.version>
</properties>
```

依赖

Kotlin 有一个广泛的标准库可用于应用程序。在 pom 文件中配置以下依赖关系：

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

如果是面向 JDK 7 或 JDK 8，那么可以使用扩展版本的 Kotlin 标准库，其中包含为新版 JDK 所增 API 而加的额外的扩展函数。使用 `kotlin-stdlib-jdk7` 或 `kotlin-stdlib-jdk8` 取代 `kotlin-stdlib`，这取决于你的 JDK 版本（对于 Kotlin 1.1.x 用 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8`，因为相应的 `jdk` 构件在 1.2.0 才引入）。

如果你的项目使用 Kotlin 反射 或者测试设施，那么你还需要添加相应的依赖项。其构件 ID 对于反射库是 `kotlin-reflect`，对于测试库是 `kotlin-test` 与 `kotlin-test-junit` 。

编译只有 Kotlin 的源代码

要编译源代码，请在 `<build>` 标签中指定源代码目录：

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

需要引用 Kotlin Maven 插件来编译源代码：

```
<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>
```

同时编译 Kotlin 与 Java 源代码

要编译混合代码应用程序，必须在 Java 编译器之前调用 Kotlin 编译器。按照 maven 的方式，这意味着应该使用以下方法在 `maven-compiler-plugin` 之前运行 `kotlin-maven-plugin`。确保 `pom.xml` 文件中的 `kotlin` 插件位于 `maven-compiler-plugin` 之前：

```xml
<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>${project.basedir}/src/test/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.5.1</version>
            <executions>
                <!-- 替换会被 maven 特别处理的 default-compile -->
                <execution>
                    <id>default-compile</id>
                    <phase>none</phase>
                </execution>
                <!-- 替换会被 maven 特别处理的 default-testCompile -->
                <execution>
                    <id>default-testCompile</id>
                    <phase>none</phase>
                </execution>
                <execution>
                    <id>java-compile</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>java-test-compile</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                    <configuration>
                        <skip>${maven.test.skip}</skip>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

增量编译

为了使构建更快，可以为 Maven 启用增量编译（从 Kotlin 1.1.2 起支持）。为了做到这一点，需要定义 `kotlin.compiler.incremental` 属性：

```
<properties>
    <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>
```

或者，使用 `-Dkotlin.compiler.incremental=true` 选项运行构建。

注解处理

请参见 Kotlin 注解处理工具（`kapt`）的描述。

协程支持

在 Kotlin 1.2 中协程支持是一项实验性的特性，因此当你在项目中使用协程时 Kotlin 编译器会报警告。可以将以下代码块添加到 `pom.xml` 文件中来关闭这一警告：

```
<configuration>
    <experimentalCoroutines>enable</experimentalCoroutines>
</configuration>
```

Jar 文件

要创建一个仅包含模块代码的小型 Jar 文件，请在 Maven pom.xml 文件中的 `build->plugins` 下面包含以下内容，其中 `main.class` 定义为一个属性，并指向主 Kotlin 或 Java 类：

```
<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>
```

独立的 Jar 文件

要创建一个独立的（self-contained）Jar 文件，包含模块中的代码及其依赖项，请在 Maven pom.xml 文件中的 `build->plugins` 下面包含以下内容其中 `main.class` 定义为一个属性，并指向主 Kotlin 或 Java 类：

```
<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>
```

这个独立的 jar 文件可以直接传给 JRE 来运行应用程序：

```
 java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

指定编译器选项

可以将额外的编译器选项与参数指定为 Maven 插件节点的 `<configuration>` 元素下的标签：

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <executions>……</executions>
    <configuration>
        <nowarn>true</nowarn>  <!-- 禁用警告 -->
        <args>
            <arg>-Xjsr305=strict</arg> <!-- 对 JSR-305 注解启用严格模式 -->
            ...
        </args>
    </configuration>
</plugin>
```

许多选项还可以通过属性来配置：

```
<project ……>
    <properties>
        <kotlin.compiler.languageVersion>1.0</kotlin.compiler.languageVersion>
    </properties>
</project>
```

支持以下属性：

JVM 和 JS 的公共属性

名称	属性名	描述	可能的值	默认值
nowarn		不生成警告	true、false	false
languageVersion	kotlin.compiler.languageVersion	提供与指定语言版本源代码兼容性	"1.0"、"1.1"、"1.2"、"1.3"、"1.4 (EXPERIMENTAL)"	
apiVersion	kotlin.compiler.apiVersion	只允许使用来自捆绑库的指定版本中的声明	"1.0"、"1.1"、"1.2"、"1.3"、"1.4 (EXPERIMENTAL)"	
sourceDirs		包含要编译源文件的目录		该项目源代码根目录
compilerPlugins		启用编译器插件		[]
pluginOptions		编译器插件的选项		[]
args		额外的编译器参数		[]

JVM 特有的属性

名称	属性名	描述	可能的值	默认值
jvmTarget	kotlin.compiler.jvmTarget	生成的 JVM 字节码的目标版本	"1.6"、"1.8"、"9"、"10"、"11"、"12"	"1.6"
jdkHome	kotlin.compiler.jdkHome	要包含到 classpath 中的 JDK 主目录路径，如果与默认 JAVA_HOME 不同的话		

JS 特有的属性

名称	属性名	描述	可能的值	默认值
outputFile		输出文件路径		
metaInfo		使用元数据生成 .meta.js 与 .kjsm 文件。用于创建库	true、false	true
sourceMap		生成源代码映射（source map）	true、false	false
sourceMapEmbedSources		将源代码嵌入到源代码映射中	"never"、"always"、"inlining"	"inlining"
sourceMapPrefix		源代码映射中路径的前缀		
moduleKind		编译器生成的模块类型	"plain"、"amd"、"commonjs"、"umd"	"plain"

生成文档

标准的 JavaDoc 生成插件（`maven-javadoc-plugin`）不支持 Kotlin 代码。要生成 Kotlin 项目的文档，请使用 Dokka；相关配置说明请参见 Dokka README 。Dokka 支持混合语言项目，并且可以生成多种格式的输出，包括标准 JavaDoc。

OSGi

对于 OSGi 支持，请参见 Kotlin OSGi 页。

示例

一个示例 Maven 项目可以从 Github 版本库直接下载

使用 Ant

获取 Ant 任务

Kotlin 为 Ant 提供了三个任务：

— kotlinc: 面向 JVM 的 Kotlin 编译器；

— kotlin2js: 面向 JavaScript 的 Kotlin 编译器；

— withKotlin: 使用标准 *javac* Ant 任务时编译 Kotlin 文件的任务。

这仨任务在 *kotlin-ant.jar* 库中定义，该库位于 Kotlin 编译器的 *lib* 文件夹中 需要 Ant 1.8.2+ 版本。

面向 JVM 只用 Kotlin 源代码

当项目由 Kotlin 专用源代码组成时，编译项目的最简单方法是使用 *kotlinc* 任务：

```
<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>
```

其中 `${kotlin.lib}` 指向解压缩 Kotlin 独立编译器所在文件夹。

面向 JVM 只用 Kotlin 源代码且多根

如果项目由多个源代码根组成，那么使用 *src* 作为元素来定义路径：

```
<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 使用 Kotlin 和 Java 源代码

如果项目由 Kotlin 和 Java 源代码组成，虽然可以使用 *kotlinc* 来避免任务参数的重复，但是建议使用 *withKotlin* 任务：

```
<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>
```

还可以将正在编译的模块的名称指定为 `moduleName` 属性：

```
<withKotlin moduleName="myModule"/>
```

面向 JavaScript 用单个源文件夹

```
<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 用 Prefix、PostFix 以及 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 用单个源文件夹以及 metaInfo 选项

如果要将翻译结果作为 Kotlin/JavaScript 库分发，那么 `metaInfo` 选项会很有用。如果 `metaInfo` 设置为 `true`，则在编译期间将创建具有二进制元数据的额外的 JS 文件。该文件应该与翻译结果一起分发：

```
<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，其中包含二进制元数据 -->
        <kotlin2js src="root1" output="out.js" metaInfo="true"/>
    </target>
</project>
```

参考

元素和属性的完整列表如下：

kotlinc 和 kotlin2js 的公共属性

名称	描述	必需	默认值
`src`	要编译的 Kotlin 源文件或目录	是	
`nowarn`	禁止所有编译警告	否	false
`noStdlib`	不要将 Kotlin 标准库包含进 classpath	否	false
`failOnError`	在编译期间检测到错误时，会导致构建失败	否	true

kotlinc 属性

名称	描述	必需	默认值
`output`	目标目录或 .jar 文件名	是	
`classpath`	编译类路径	否	
`classpathref`	编译类路径引用	否	
`includeRuntime`	Kotlin 运行时库是否包含在 jar 中，如果 `output` 是 .jar 文件的话	否	true
`moduleName`	编译的模块的名称	否	目标（如果指定的话）或项目的名称

kotlin2js 属性

名称	描述	必需
`output`	目标文件	是
`libraries`	Kotlin 库的路径	否
`outputPrefix`	生成的 JavaScript 文件所用前缀	否
`outputSuffix`	生成的 JavaScript 文件所用后缀	否
`sourcemap`	是否要生成 sourcemap 文件	否
`metaInfo`	是否要生成具有二进制描述符的元数据文件	否
`main`	编译器是否生成调用 main 函数的代码	否

传递原始编译器参数

如需传递原始编译器参数，可以使用带 `value` 或 `line` 属性的 `<compilerarg>` 元素。可以放在 `<kotlinc>`、`<kotlin2js>` 与 `<withKotlin>` 任务元素内，如下所示：

```
<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>
```

当运行 `kotlinc -help` 时，会显示可以使用的参数的完整列表。

Kotlin 编译器选项

每个 Kotlin 版本都包含支持目标的编译器：用于所支持平台的 JVM、JavaScript 与 Native 二进制文件。

当单击 Kotlin 项目的 **Compile** 或 **运行(Run)** 按钮时，IDE 会使用这些编译器。

还可以按照使用命令行编译器 教程中所述在命令行手动运行 Kotlin 编译器。例如：

```
$ kotlinc hello.kt -include-runtime -d hello.jar
```

编译器选项

Kotlin 编译器具有许多用于定制编译过程的选项。此页面列出了针对不同目标的编译器选项，并提供了每个选项的描述。

有几种方法可以设置编译器选项及其值（*编译器参数*）：

- 在 IntelliJ IDEA 的 **设置(Settings) | 构建、执行、部署(Build, Execution, Deployment) | 编译器(Compilers) | Kotlin Compiler** 窗口中，在 **附加命令行参数(Additional command-line parameters)** 文本框中输入编译器参数。
- 如果使用 Gradle，请在 Kotlin 编译任务的 `kotlinOptions` 属性中指定编译器参数。详情请参见使用 Gradle。
- 如果使用 Maven，请在 Maven 插件节点的 `<configuration>` 元素中指定编译器参数。详情请参见使用 Maven。
- 如果运行命令行编译器，则将编译器参数直接添加到工具函数调用中，或将其写入 argfile。

公共选项

以下选项对于所有 Kotlin 编译器都是通用的。

`-version`

显示编译器版本。

`-nowarn`

禁止编译器在编译期间显示警告。

`-Werror`

将所有警告转换为编译错误。

`-verbose`

启用详细的日志记录输出，其中包括编译过程的详细信息。

`-script`

运行 Kotlin 脚本文件。使用此选项调用时，编译器将执行给定参数中的第一个 Kotlin 脚本文件（`*.kts`）。

`-help` (`-h`)

显示用法信息并退出。仅显示标准选项。要显示高级选项，请使用 `-X`。

-X

显示有关高级选项的信息并退出。这些选项当前不稳定：它们的名称与行为可能会更改，恕不另行通知。

-kotlin-home <path>

指定用于发现运行时库的 Kotlin 编译器的自定义路径。

-P plugin:<pluginId>:<optionName>=<value>

将选项传递给 Kotlin 编译器插件。可用插件及其选项列在编译器插件中。

-language-version <version>

提供与指定版本的 Kotlin 的源代码兼容性。

-api-version <version>

仅允许使用 Kotlin 捆绑库指定版本中的声明。

-progressive

为编译器启用渐进模式。

在渐进模式下，针对不稳定代码的弃用与错误修复将立即生效，而无需经历正常的迁移周期。以渐进模式编写的代码是向后兼容的。但是，以非渐进模式编写的代码可能会在渐进模式下导致编译错误。

@<argfile>

从给定文件中读取编译器选项。这样的文件可以包含带有值与源文件路径的编译器选项。选项与路径应由空格分隔。例如：

```
-include-runtime -d hello.jar
hello.kt
```

要传递包含空格的值，请用单引号 (') 或双引号 (") 引起来。如果值中包含引号，请使用反斜杠 (\) 对其进行转义。

```
-include-runtime -d 'My folder'
```

还可以传递多个参数文件，例如，将编译器选项与源文件分开。

```
$ kotlinc @compiler.options @classes
```

如果文件位于与当前目录不同的位置，请使用相对路径。

```
$ kotlinc @options/compiler.options hello.kt
```

Kotlin/JVM 编译器选项

用于 JVM 的 Kotlin 编译器将 Kotlin 源文件编译为 Java 类文件。用于 Kotlin 到 JVM 编译的命令行工具是 `kotlinc` 与 `kotlinc-jvm`。也可以使用它们来执行 Kotlin 脚本文件。

除了公共选项外，Kotlin/JVM 编译器还具有以下列出的选项。

`-classpath <path>`(`-cp <path>`)

在指定的路径中搜索类文件。用系统路径分隔符将类路径的各个元素分开(在Windows上是 **;** 在macOS/Linux上是 **:**)。类路径可以包含文件和目录路径、ZIP 或 JAR 文件。

`-d <path>`

将生成的类文件放置到指定位置。该位置可以是目录、ZIP 或 JAR 文件。

`-include-runtime`

将 Kotlin 运行时包含在生成的 JAR 文件中。使生成的归档文件可在任何启用 Java 的环境中运行。

`-jdk-home <path>`

如果自定义 JDK 主目录与默认的 `JAVA_HOME` 不同,请使用它来将其包含在类路径中。

`-jvm-target <version>`

指定生成的 JVM 字节码的目标版本。可能的值为 `1.6`、`1.8`、`9`、`10`、`11`、`12`、`13`。默认值为 `1.6`。

`-java-parameters`

为 Java 1.8 反射方法参数生成元数据。

`-module-name <name>`

为生成的 `.kotlin_module` 文件设置自定义名称。

`-no-jdk`

不要自动将 Java 运行时包含在类路径中。

`-no-reflect`

不要自动将 Kotlin 反射(`kotlin-reflect.jar`)包含到类路径中。

`-no-stdlib`

不要自动将 Kotlin/JVM 标准库(`kotlin-stdlib.jar`)与 Kotlin 反射(`kotlin-reflect.jar`)包含到类路径中。

`-script-templates <classnames[,]>`

脚本定义模板类。使用完全限定的类名,并用逗号(**,**)分隔。

Kotlin/JS 编译器选项

JS 的 Kotlin 编译器将 Kotlin 源文件编译为 JavaScript 代码。Kotlin 到 JS 编译的命令行工具是 `kotlinc-js`。

除了公共选项外,Kotlin/JS 编译器还具有以下列出的选项。

-libraries <path>

具有 `.meta.js` 与 `.kjsm` 文件的 Kotlin 库的路径，由系统路径分隔符分隔。

-main {call|noCall}

定义是否应在执行时调用 `main` 函数。

-meta-info

使用元数据生成 `.meta.js` 与 `.kjsm` 文件。创建 JS 库时使用此选项。

-module-kind {plain|amd|commonjs|umd}

编译器生成的 JS 模块类型：

- `plain` ——普通的 JS 模块；
- `commonjs` ——CommonJS 模块；
- `amd` ——异步模块定义模块；
- `umd` ——通用模块定义模块。

要了解有关不同类型的 JS 模块及其之间区别的更多信息，请参见这篇文章。

-no-stdlib

不要自动将默认的 Kotlin/JS 标准库包含在编译依赖项中。

-output <filepath>

设置编译结果的目标文件。该值必须是包含其名称的 `.js` 文件的路径。

-output-postfix <filepath>

将指定文件的内容添加到输出文件的末尾。

-output-prefix <filepath>

将指定文件的内容添加到输出文件的开头。

-source-map

生成源代码映射。

-source-map-base-dirs <path>

使用指定的路径作为基本目录。基本目录用于计算源代码映射中的相对路径。

-source-map-embed-sources {always|never|inlining}

将源代码文件嵌入到源代码映射中。

`-source-map-prefix`

将指定的前缀添加到源代码映射中的路径。

Kotlin/Native 编译器选项

Kotlin/Native 编译器将 Kotlin 源代码文件编译为用于所支持平台的 Native 二进制文件。Kotlin/Native 编译的命令行工具是 `kotlinc-native`。

除了公共选项外，Kotlin/Native 编译器还具有以下列出的选项。

`-enable-assertions (-ea)`

在生成的代码中启用运行时断言。

`-g`

启用输出调试信息。

`-generate-test-runner (-tr)`

生成一个用于运行项目中的单元测试的应用程序。

`-generate-worker-test-runner (-trw)`

生成一个用于在 Worker 线程中运行单元测试的应用程序。

`-generate-no-exit-test-runner (-trn)`

生成一个用于运行单元测试的应用程序，而无需显式退出进程。

`-include-binary <path> (-ib <path>)`

在生成的 klib 文件中打包外部二进制文件。

`-library <path> (-l <path>)`

与库链接。要了解有关在 Kotlin/native 项目中使用库的信息，请参见 Kotlin/Native 库。

`-library-version <version> (-lv)`

设置库版本。

`-list-targets`

列出可用的硬件目标。

`-manifest <path>`

提供清单附加文件。

-module-name <name>

指定编译模块的名称。此选项还可用于为导出到 Objective-C 的声明指定名称前缀：如何为 Kotlin 框架指定自定义的 Objective-C 前缀/名称?

-native-library <path>(-nl <path>)

包含 Native Bitcode 库。

-no-default-libs

禁止将用户代码与编译器一起分发的默认平台库链接。

-nomain

假定要由外部库提供的 `main` 入口点。

-nopack

不要将库打包到 klib 文件中。

-linker-option

在二进制构建过程中，将参数传递给链接器。这可用于链接到某些 Native 库。

-linker-options <args>

在二进制构建过程中，将多个参数传递给链接器。用空格分隔参数。

-nostdlib

不要与标准库链接。

-opt

启用编译优化。

-output <name>(-o <name>)

设置输出文件的名称。

-entry <name>(-e <name>)

指定限定的入口点名称。

-produce <output>(-p)

指定输出文件的种类：

— `program`
— `static`
— `dynamic`
— `framework`
— `library`
— `bitcode`

`-repo <path>(-r <path>)`

库搜索路径。有关更多信息，请参见库搜索顺序。

`-target <target>`

设置硬件目标。要查看可用目标的列表，请使用 `-list-targets` 选项。

编译器插件

- 全开放编译器插件
- 无参编译器插件
- 带有接收者的 SAM 编译器插件
- `Parcelable` 实现生成器

全开放编译器插件

Kotlin 的类及其成员默认是 `final` 的，这使得像 Spring AOP 这样需要类为 `open` 的框架与库用起来很不方便。这个 *all-open* 编译器插件会适配 Kotlin 以满足那些框架的需求，并使用指定的注解标注类而其成员无需显式使用 `open` 关键字打开。

例如，当你使用 Spring 时，你不需要打开所有的类，而只需要使用特定的注解标注，如 `@Configuration` 或 `@Service`。*All-open* 允许指定这些注解。

我们为全开放插件提供 Gradle 与 Maven 支持并有完整的 IDE 集成。

注意：对于 Spring，你可以使用 `kotlin-spring` 编译器插件（见下文）。

在 Gradle 中使用

将插件构件添加到 buildscript 依赖中并应用该插件：

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-allopen"
```

另一种方式是使用 `plugins` 块启用之：

```
plugins {
  id "org.jetbrains.kotlin.plugin.allopen" version "1.3.72"
}
```

然后指定会打开类的注解的列表：

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation", "com.third.Annotation")
}
```

如果类（或任何其超类）标有 `com.my.Annotation` 注解，类本身及其所有成员会变为开放。

它也适用于元注解：

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // 将会全开放
```

`MyFrameworkAnnotation` 已由全开放元注解 `com.my.Annotation` 标注，所以它也成了一个全开放注解。

在 Maven 中使用

下面是全开放与 Maven 一起使用的用法：

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- 或者 "spring" 对于 Spring 支持 -->
            <plugin>all-open</plugin>
        </compilerPlugins>

        <pluginOptions>
            <!-- 每个注解都放在其自己的行上 -->
            <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>
```

关于全开放注解如何工作的详细信息，请参考上面的“在 Gradle 中使用”一节。

Spring 支持

如果使用 Spring，可以启用 *kotlin-spring* 编译器插件而不是手动指定 Spring 注解。kotlin-spring 是在全开放之上的一层包装，并且其运转方式也完全相同。

与全开放一样，将该插件添加到 buildscript 依赖中：

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-spring" // 取代 "kotlin-allopen"
```

或者使用 Gradle 插件 DSL：

```
plugins {
  id "org.jetbrains.kotlin.plugin.spring" version "1.3.72"
}
```

在 Maven 中，`kotlin-maven-allopen` 插件的依赖项提供了 `spring` 插件，因此可以这样启用：

```
<configuration>
    <compilerPlugins>
        <plugin>spring</plugin>
    </compilerPlugins>
</configuration>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-allopen</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

该插件指定了以下注解：@Component、@Async、@Transactional、@Cacheable 以及 @SpringBootTest。由于元注解的支持，标注有 @Configuration、@Controller、@RestController、@Service 或者 @Repository 的类会自动打开，因为这些注解标注有元注解 @Component。

当然，你可以在同一个项目中同时使用 `kotlin-allopen` 与 `kotlin-spring`。

请注意，如果使用 start.spring.io 服务生成的项目模板，那么默认会启用 `kotlin-spring` 插件。

在命令行中使用

全开放编译器插件的 JAR 包已随 Kotlin 编译器的二进制发行版分发。可以使用 kotlinc 选项 `Xplugin` 提供该 JAR 文件的路径来附加该插件：

```
-Xplugin=$KOTLIN_HOME/lib/allopen-compiler-plugin.jar
```

可以使用 `annotation` 插件选项或者启用“预设”来直接指定全开放注解。现在可用于全开放的唯一预设是 `spring`。

```
# The plugin option format is: "-P plugin:<plugin id>:<key>=<value>".
# Options can be repeated.

-P plugin:org.jetbrains.kotlin.allopen:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.allopen:preset=spring
```

无参编译器插件

*无参(no-arg)*编译器插件为具有特定注解的类生成一个额外的零参数构造函数。

这个生成的构造函数是合成的，因此不能从 Java 或 Kotlin 中直接调用，但可以使用反射调用。

这允许 Java Persistence API（JPA）实例化一个类，就算它从 Kotlin 或 Java 的角度看没有无参构造函数（参见下面的 `kotlin-jpa` 插件的描述）。

在 Gradle 中使用

其用法非常类似于全开放插件。

添加该插件并指定注解的列表，这些注解一定会导致被标注的类生成无参构造函数。

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-noarg"
```

或者使用 Gradle 插件 DSL：

```
plugins {
  id "org.jetbrains.kotlin.plugin.noarg" version "1.3.72"
}
```

然后指定无参注解列表：

```
noArg {
    annotation("com.my.Annotation")
}
```

如果希望该插件在合成的构造函数中运行其初始化逻辑，请启用 `invokeInitializers` 选项。由于在未来会解决的 KT-18667 及 KT-18668，自 Kotlin 1.1.3-2 起，它被默认禁用。

```
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>
            <!-- 或者对于 JPA 支持用 "jpa" -->
            <plugin>no-arg</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>no-arg:annotation=com.my.Annotation</option>
            <!-- 在合成的构造函数中调用实例初始化器 -->
            <!-- <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* 插件类似，*kotlin-jpa* 是在 *no-arg* 之上的一层包装。该插件自动指定了 `@Entity`、`@Embeddable` 与 `@MappedSuperclass` 这几个 *无参* 注解。

这是在 Gradle 中添加该插件的方法：

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-jpa"
```

或者使用 Gradle 插件 DSL：

```
plugins {
  id "org.jetbrains.kotlin.plugin.jpa" version "1.3.72"
}
```

在 Maven 中，则启用 `jpa` 插件：

```
<compilerPlugins>
    <plugin>jpa</plugin>
</compilerPlugins>
```

在命令行中使用

与全开放类似，将插件 JAR 文件添加到编译器插件类路径并指定注解或预设：

```
-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
```

带有接收者的 SAM 编译器插件

编译器插件 *sam-with-receiver* 使所注解的 Java“单抽象方法”接口方法的第一个参数成为 Kotlin 中的接收者。这一转换只适用于当 SAM 接口作为 Kotlin 的 lambda 表达式传递时，对 SAM 适配器与 SAM 构造函数均适用（详见其文档）。

这里有一个示例：

```
public @interface SamWithReceiver {}

@SamWithReceiver
public interface TaskRunner {
    void run(Task task);
}
```

```
fun test(context: TaskContext) {
    val runner = TaskRunner {
        // 这里的“this”是“Task”的一个实例

        println("$name is started")
        context.executeTask(this)
        println("$name is finished")
    }
}
```

在 Gradle 中使用

除了事实上 sam-with-receiver 没有任何内置预设、并且需要指定自己的特殊处理注解列表外，其用法与 all-open 及 no-arg 相同。

```
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 的注解列表：

```
samWithReceiver {
    annotation("com.my.SamWithReceiver")
}
```

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 CLI 中使用

只需将该插件的 JAR 文件添加到编译器插件类路径中，并指定 sam-with-receiver 注解列表即可：

```
-Xplugin=$KOTLIN_HOME/lib/sam-with-receiver-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.samWithReceiver:annotation=com.my.SamWithReceiver
```

`Parcelable` 实现生成器

Android 扩展插件提供了 `Parcelable` 实现生成器。

用 `@Parcelize` 标注该类，然后会自动生成一个 `Parcelable` 实现。

```
import kotlinx.android.parcel.Parcelize

@Parcelize
class User(val firstName: String, val lastName: String, val age: Int): Parcelable
```

`@Parcelize` 要求在主构造函数中声明所有序列化的属性。Android 扩展程序会针对每个属性发出警告，并在类主体中声明一个幕后字段。此外，如果某些主要构造函数参数不是属性，则无法应用 `@Parcelize`。

如果类需要更高级的序列化逻辑，请在伴生类中编写：

```
@Parcelize
data class User(val firstName: String, val lastName: String, val age: Int) : Parcelable {
    private companion object : Parceler<User> {
        override fun User.write(parcel: Parcel, flags: Int) {
            // Custom write implementation
        }

        override fun create(parcel: Parcel): User {
            // Custom read implementation
        }
    }
}
```

已支持类型

`@Parcelize` 支持多种类型：

- 基本类型（及其装箱版）；
- 对象与枚举；
- `String`、`CharSequence`；
- `Exception`；
- `Size`、`SizeF`、`Bundle`、`IBinder`、`IInterface`、`FileDescriptor`；
- `SparseArray`、`SparseIntArray`、`SparseLongArray`、`SparseBooleanArray`；
- 所有 `Serializable`（是的，`Date` 也受支持）与 `Parcelable` 实现；
- 所有已支持类型的集合：`List`（映射到 `ArrayList`），`Set` 映射到 `LinkedHashSet`，`Map` 映射到 `LinkedHashMap`；
  - 还有一些具体的实现：`ArrayList`、`LinkedList`、`SortedSet`、`NavigableSet`、`HashSet`、`LinkedHashSet`、`TreeSet`
- 所有受支持的数组类型；
- 所有受支持的类型的可空版本。

自定义 Parceler

即使不直接支持的类型，也可以为其编写 `Parceler` 映射对象。

```
class ExternalClass(val value: Int)

object ExternalClassParceler : Parceler<ExternalClass> {
    override fun create(parcel: Parcel) = ExternalClass(parcel.readInt())

    override fun ExternalClass.write(parcel: Parcel, flags: Int) {
        parcel.writeInt(value)
    }
}
```

可以使用 `@TypeParceler` 或 `@WriteWith` 标注应用外部包裹程序：

```
// Class-local parceler
@Parcelize
@TypeParceler<ExternalClass, ExternalClassParceler>()
class MyClass(val external: ExternalClass)

// Property-local parceler
@Parcelize
class MyClass(@TypeParceler<ExternalClass, ExternalClassParceler>() val external: ExternalClass)

// Type-local parceler
@Parcelize
class MyClass(val external: @WriteWith<ExternalClassParceler>() ExternalClass)
```

Kotlin 注解处理

> 译注：kapt 即 Kotlin annotation processing tool（Kotlin 注解处理工具）缩写。

在 Kotlin 中通过 *kapt* 编译器插件支持注解处理器（参见 JSR 269）。

简而言之，你可以在 Kotlin 项目中使用像 Dagger 或者 Data Binding 这样的库。

关于如何将 *kapt* 插件应用于 Gradle/Maven 构建中，请阅读下文。

在 Gradle 中使用

应用 `kotlin-kapt` Gradle 插件：

```
plugins {
    id "org.jetbrains.kotlin.kapt" version "1.3.72"
}
```

```
plugins {
    kotlin("kapt") version "1.3.72"
}
```

或者使用 `apply plugin` 语法：

```
apply plugin: 'kotlin-kapt'
```

然后在 `dependencies` 块中使用 `kapt` 配置添加相应的依赖项：

```
dependencies {
    kapt 'groupId:artifactId:版本'
}
```

```
dependencies {
    kapt("groupId:artifactId:版本")
}
```

如果你以前使用 Android 支持作为注解处理器，那么以 `kapt` 取代 `annotationProcessor` 配置的使用。如果你的项目包含 Java 类，`kapt` 也会顾全到它们。

如果为 `androidTest` 或 `test` 源代码使用注解处理器，那么相应的 `kapt` 配置名为 `kaptAndroidTest` 和 `kaptTest`。请注意 `kaptAndroidTest` 和 `kaptTest` 扩展了 `kapt`，所以你可以只提供 `kapt` 依赖而它对生产和测试源代码都可用。

注解处理器参数

使用 `arguments {}` 块将参数传给注解处理器：

```
kapt {
    arguments {
        arg("key", "value")
    }
}
```

Gradle 构建缓存支持（自 1.2.20 起）

默认情况下，kapt 注解处理任务就会在 Gradle 中缓存。注解处理器所运行的任意代码可能不一定将输入转换为输出、可能访问与修改 Gradle 未跟踪的文件等。如果无法正确缓存在构建中使用的注解处理器，那么可以通过在构建脚本中添加以下几行来完全禁用对 kapt 的缓存，以避免对 kapt 任务造成假阳性的缓存命中：

```
kapt {
    useBuildCache = false
}
```

并行运行 kapt 任务（自 1.2.60 起）

为了提高使用 kapt 的构建速度，可以为 kapt 任务启用 Gradle Worker API。使用 Worker API，Gradle 可以并行运行来自单个项目的独立注解处理任务，这在某些情况下会大大减少执行时间。但是，在启用 Gradle Worker API 的情况下运行 kapt 会由于并行执行而导致内存消耗增加。

要使用 Gradle Worker API 并行执行 kapt 任务，请将此行添加到 `gradle.properties` 文件中：

```
kapt.use.worker.api=true
```

kapt 的避免编译（自 1.3.20 起）

为了减少 kapt 增量构建的时间，可以使用 Gradle 避免编译。启用避免编译后，Gradle 可以在重新构建项目时跳过注解处理。特别是在以下情况下，将跳过注解处理：

- 项目的源文件未更改。
- 依赖项中的变更是 ABI 兼容的。例如，唯一的变化是方法主体。

但是，避免编译不能用于在编译类路径中发现的注解处理器，因为它们中的任何更改都需要运行注解处理任务。

要在避免编译的情况下运行 kapt：

- 如在 Gradle 中使用所述，将注解处理器依赖项手动添加到 `kapt*` 配置。
- 通过在 `gradle.properties` 文件中添加以下行，在编译类路径中关闭对注解处理器的发现：

```
kapt.include.compile.classpath=false
```

增量注解处理（自 1.3.30 起）

从 1.3.30 版开始，kapt 作为实验特性支持增量注解处理。当前，仅当所使用的所有注解处理器均为增量式时，注解处理才可以是增量式的。

从 1.3.50 版开始，默认情况下启用增量注解处理。要禁用增量注解处理，请将以下行添加到 `gradle.properties` 文件中：

```
kapt.incremental.apt=false
```

请注意，增量注解处理也需要启用增量编译。

Java 编译器选项

Kapt 使用 Java 编译器来运行注解处理器。以下是将任意选项传给 javac 的方式：

```
kapt {
    javacOptions {
        // 增加注解处理器的最大错误次数
        // 默认为 100。
        option("-Xmaxerrs", 500)
    }
}
```

非存在类型校正

一些注解处理器（如 `AutoFactory`）依赖于声明签名中的精确类型。默认情况下，Kapt 将每个未知类型（包括生成的类的类型）替换为 `NonExistentClass`，但你可以更改此行为。将额外标志添加到 `build.gradle` 文件以启用在存根（stub）中推断出的错误类型：

```
kapt {
    correctErrorTypes = true
}
```

在 Maven 中使用

在 `compile` 之前在 kotlin-maven-plugin 中添加 `kapt` 目标的执行：

```
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal>
    </goals>
    <configuration>
        <sourceDirs>
            <sourceDir>src/main/kotlin</sourceDir>
            <sourceDir>src/main/java</sourceDir>
        </sourceDirs>
        <annotationProcessorPaths>
            <!-- 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>
```

你可以在 Kotlin 示例版本库 中找到一个显示使用 Kotlin、Maven 和 Dagger 的完整示例项目。

请注意，IntelliJ IDEA 自身的构建系统目前还不支持 kapt。当你想要重新运行注解处理时，请从“Maven Projects”工具栏启动构建。

在命令行中使用

Kapt 编译器插件已随 Kotlin 编译器的二进制发行版分发。

可以使用 kotlinc 选项 `Xplugin` 提供该 JAR 文件的路径来附加该插件：

```
-Xplugin=$KOTLIN_HOME/lib/kotlin-annotation-processing.jar
```

以下是可用选项的列表：

— `sources`（*必需*）：所生成文件的输出路径。

— `classes`（*必需*）：所生成类文件与资源的输出路径。

- `stubs`（*必需*）：存根文件的输出路径。换句话说，一些临时目录。
- `incrementalData`：二进制存根的输出路径。
- `apclasspath`（*可重复*）：注解处理器 JAR 包路径。如果有的多个 JAR 包就传多个 `apclasspath` 选项。
- `apoptions`：注解处理器选项的 base64 编码列表。详见 AP/javac options encoding。
- `javacArguments`：传给 javac 的选项的 base64 编码列表。详见 AP/javac options encoding。
- `processors`：逗号分隔的注解处理器全类名列表。如果指定，kapt 就不会尝试在 `apclasspath` 中查找注解处理器。
- `verbose`：启用详细输出。
- `aptMode`（*必需*）
  - `stubs` ——只生成注解处理所需的存根；
  - `apt` ——只运行注解处理；
  - `stubsAndApt` ——生成存根并运行注解处理。
- `correctErrorTypes`：参见下文。默认未启用。

插件选项格式为：`-P plugin:<plugin id>:<key>=<value>`。选项可以重复。

一个示例：

```
-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 代码

Kapt 可生成 Kotlin 代码。是将生成的 Kotlin 源文件写入 `processingEnv.options["kapt.kotlin.generated"]` 所指定的目录，这些文件会与主源代码一起编译。

可以在 kotlin-examples Github 版本库中找到完整的示例。

请注意，对于所生成 Kotlin 文件，Kapt 不支持多轮处理。

AP/javac 选项编码

`apoptions` 与 `javacArguments` 命令行选项接受选项编码映射。这是自己编码选项的方式：

```
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())
}
```

编写 Kotlin 代码文档

用来编写 Kotlin 代码文档的语言（相当于 Java 的 JavaDoc）称为 **KDoc**。本质上 KDoc 是将 JavaDoc 的块标签（block tags）语法（扩展为支持 Kotlin 的特定构造）和 Markdown 的内联标记（inline markup）结合在一起。

生成文档

Kotlin 的文档生成工具称为 Dokka。其使用说明请参见 Dokka README。

Dokka 有 Gradle、Maven 和 Ant 的插件，因此你可以将文档生成集成到你的构建过程中。

KDoc 语法

像 JavaDoc 一样，KDoc 注释也以 `/**` 开头、以 `*/` 结尾。注释的每一行可以以星号开头，该星号不会当作注释内容的一部分。

按惯例来说，文档文本的第一段（到第一行空白行结束）是该元素的总体描述，接下来的注释是详细描述。

每个块标签都以一个新行开始且以 `@` 字符开头。

以下是使用 KDoc 编写类文档的一个示例：

```
/**
 * 一组*成员*。
 *
 * 这个类没有有用的逻辑；它只是一个文档示例。
 *
 * @param T 这个组中的成员的类型。
 * @property name 这个组的名称。
 * @constructor 创建一个空组。
 */
class Group<T>(val name: String) {
    /**
     * 将 [member] 添加到这个组。
     * @return 这个组的新大小。
     */
    fun add(member: T): Int { …… }
}
```

块标签

KDoc 目前支持以下块标签（block tags）：

`@param <名称>`

用于函数的值参数或者类、属性或函数的类型参数。为了更好地将参数名称与描述分开，如果你愿意，可以将参数的名称括在方括号中。因此，以下两种语法是等效的：

```
@param name 描述。
@param[name] 描述。
```

`@return`

用于函数的返回值。

`@constructor`

用于类的主构造函数。

`@receiver`

用于扩展函数的接收者。

`@property <名称>`

用于类中具有指定名称的属性。这个标签可用于在主构造函数中声明的属性，当然直接在属性定义的前面放置 doc 注释会很别扭。

`@throws <类>, @exception <类>`

用于方法可能抛出的异常。因为 Kotlin 没有受检异常，所以也没有期望所有可能的异常都写文档，但是当它会为类的用户提供有用的信息时，仍然可以使用这个标签。

`@sample <标识符>`

将具有指定限定的名称的函数的主体嵌入到当前元素的文档中，以显示如何使用该元素的示例。

`@see <标识符>`

将到指定类或方法的链接添加到文档的**另请参见**块。

`@author`

指定要编写文档的元素的作者。

`@since`

指定要编写文档的元素引入时的软件版本。

`@suppress`

从生成的文档中排除元素。可用于不是模块的官方 API 的一部分但还是必须在对外可见的元素。

> KDoc 不支持 `@deprecated` 这个标签。作为替代，请使用 `@Deprecated` 注解。

内联标记

对于内联标记，KDoc 使用常规 Markdown 语法，扩展了支持用于链接到代码中其他元素的简写语法。

链接到元素

要链接到另一个元素（类、方法、属性或参数），只需将其名称放在方括号中：

```
为此目的，请使用方法 [foo]。
```

如果要为链接指定自定义标签（label），请使用 Markdown 引用样式语法：

```
为此目的，请使用[这个方法][foo]。
```

你还可以在链接中使用限定的名称。请注意，与 JavaDoc 不同，限定的名称总是使用点字符来分隔组件，即使在方法名称之前：

```
使用 [kotlin.reflect.KClass.properties] 来枚举类的属性。
```

链接中的名称与正写文档的元素内使用该名称使用相同的规则解析。特别是，这意味着如果你已将名称导入当前文件，那么当你在 KDoc 注释中使用它时，不需要再对其进行完整限定。

请注意 KDoc 没有用于解析链接中的重载成员的任何语法。因为 Kotlin 文档生成工具将一个函数的所有重载的文档放在同一页面上，标识一个特定的重载函数并不是链接生效所必需的。

模块和包文档

作为一个整体的模块、以及该模块中的包的文档，由单独的 Markdown 文件提供，并且使用 `-include` 命令行参数或 Ant、Maven 和 Gradle 中的相应插件将该文件的路径传递给 Dokka。

在该文件内部，作为一个整体的模块和分开的软件包的文档由相应的一级标题引入。标题的文本对于模块必须是“Module `<模块名>`”，对于包必须是“Package `<限定的包名>`”。

以下是该文件的一个示例内容：

```
# Module kotlin-demo

该模块显示 Dokka 语法的用法。

# Package org.jetbrains.kotlin.demo

包含各种有用的东西。

## 二级标题

这个标题下的文本也是 `org.jetbrains.kotlin.demo` 文档的一部分。

# Package org.jetbrains.kotlin.demo2

另一个包中有用的东西。
```

Kotlin 与 OSGi

要启用 Kotlin OSGi 支持，你需要引入 `kotlin-osgi-bundle` 而不是常规的 Kotlin 库。建议删除 `kotlin-runtime`、`kotlin-stdlib` 和 `kotlin-reflect` 依赖，因为 `kotlin-osgi-bundle` 已经包含了所有这些。当引入外部 Kotlin 库时你也应该注意。大多数常规 Kotlin 依赖不是 OSGi-就绪的，所以你不应该使用它们，且应该从你的项目中删除它们。

Maven

将 Kotlin OSGi 包引入到 Maven 项目中：

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-osgi-bundle</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

从外部库中排除标准库（注意“星排除”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

将 `kotlin-osgi-bundle` 引入到 gradle 项目中：

```
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

要排除作为传递依赖的默认 Kotlin 库，你可以使用以下方法：

```
dependencies {
 compile (
   [group: 'some.group.id', name: 'some.library', version: 'someversion'],
   ……) {
  exclude group: 'org.jetbrains.kotlin'
}
```

FAQ

为什么不只是添加必需的清单选项到所有 Kotlin 库

尽管它是提供 OSGi 支持的最好的方式，遗憾的是现在做不到，是因为不能轻易消除的所谓的 “包拆分”问题并且这么大的变化不可能现在规划。有 `Require-Bundle` 功能，但它也不是最好的选择，不推荐使用。所以决定为 OSGi 做一个单独的构件。

演进

Kotlin 演进

实用主义演进原则

> 语言的设计是石头铸造的（译注：原文为“cast in stone”，意为“最终定论，板上钉钉”，此处双关），
> 但这块石头相当柔软，
> 经过一番努力，我们可以后期重塑它。
>
> Kotlin 设计团队

Kotlin 旨在成为程序员的实用工具。在语言演进方面，它的实用主义本质遵循以下原则：

— 一直保持语言的现代性。

— 与用户保持持续的反馈循环。

— 使版本更新对用户来说是舒适的。

由于这是理解 Kotlin 如何向前发展的关键，我们来展开来看这些原则。

保持语言现代性。我们承认系统随着时间的推移积累了很多遗留问题。曾经是尖端技术的东西如今可能已经无可救药地过时了。我们必须改进语言，使其与用户需求保持一致、与用户期望保持同步。这不仅包括添加新特性，还包括逐步淘汰不再推荐用于生产的旧特性，并且完全成为历史特性。

舒适的更新。如果没有适度谨慎地进行不兼容的变更（例如从语言中删除内容）可能会导致从一个版本到下一个版本的痛苦迁移过程。我们会始终提前公布这类变更，将相应内容标记为已弃用并 *在变更发生之前* 提供自动化的迁移工具。当语言发生变更之时，我们希望世界上绝大多数代码都已经更新，这样迁移到新版本就没有问题了。

反馈循环。通过弃用周期需要付出很大的努力，因此我们希望最大限度地减少将来不兼容变更的数量。除了使用我们的最佳判断之外，我们相信在现实生活中试用是验证设计的最佳方法。在最终定论之前，我们希望已经实战测试过。这就是为什么我们利用每个机会在语言的生产版本中提供我们早期版设计，只是带有_实验性_ 状态。实验性特性并不稳定，可以随时更改，选择使用它们的用户明确表示已准备好了应对未来的迁移问题。这些用户提供了宝贵的反馈，而我们收集这些反馈来迭代设计并使其坚如磐石。

不兼容的变更

如果从一个版本更新到另一个版本时，一些以前工作的代码不再工作，那么它是语言中的 *不兼容的变更*（有时称为“破坏性变更”）。在一些场景中“不再工作”的确切含义可能会有争议，但是它肯定包含以下内容：

— 之前编译运行正常的代码现在（编译或链接）失败并报错。这包括删除语言结构以及添加新的限制。

— 之前正常执行的代码现在抛异常了。

属于“灰色区域”的不太明显的情况包括以不同方式处理极端情况，抛出与以前不同类型的异常，仅通过反射可以观察到的行为更改，未记录/未定义的行为更改，重命名二进制文件等。有时这些更改非常重要，并且会极大地影响迁移体验，有时影响微不足道。

绝对不是不兼容的变更的一些示例包括

— 添加新的警告。

— 启用新的语言结构或放宽对现有语言结构的限制。
— 更改私有/内部 API 和其他实现细节。

保持语言现代化和舒适更新的原则表明，有时需要进行不兼容的更改，但应该详细介绍这些更改。我们的目标是使用户提前了解即将发生的更改，以便他们能够轻松地迁移代码。

理想情况下，应通过有问题的代码中报告的编译期警告（通常称为弃用警告）来声明每个不兼容的更改，并提供自动迁移辅助工具。因此，理想的迁移工作流程如下：

— 更新到版本 A（宣布更改）
  — 查看有关即将发生的更改的警告
  — 借助工具迁移代码
— 更新到版本 B（发生更改）
  — 完全没有问题

实际上，在编译期无法准确检测到某些更改，因此不会报告任何警告，但是至少会通过版本 A 的发行说明通知用户版本 B 中即将进行的更改。

处理编译器错误

编译器是复杂的软件，尽管开发人员尽了最大努力，但它们仍然存在 bug。导致编译器自身编译失败、或报告虚假错误、或生成明显编译失败的代码的 bug，虽然很烦人并且常常令人尴尬，但它们很容易修复，因为这些修复不构成不兼容的变更。其他 bug 可能会导致编译器生成不会编译失败的错误代码，例如：遗漏了源代码中的一些错误，或者只是生成了错误的指令。这些 bug 的修复是技术上不兼容的更改（某些代码过去可以正常编译，但现在编译失败），但是我们倾向于尽快修复它们，以防止不良代码模式在用户代码中传播。我们认为，这符合“舒适更新”的原则，因为较少的用户有机会遇到此问题。当然，这仅适用于在发行版本中出现后不久发现的 bug。

决策制定

JetBrains是 Kotlin 的原始创建者，它在社区的帮助下并根据 Kotlin 基金会来推动 kotlin 的发展。

首席语言设计师（现为 Andrey Breslav）负责监督 Kotlin 编程语言的所有更改。首席设计师在与语言发展有关的所有事务中拥有最终决定权。此外，对完全稳定的组件进行不兼容的更改必须完全由Kotlin 基金会指定的语言委员会（目前由 Jeffrey van Gogh，William R. Cook和Andrey Breslav组成）批准。

语言委员会对将进行哪些不兼容的更改以及应采取什么确切的措施使用户感到满意做出最终决定。为此，它依赖此处提供的一组准则。

特性发布与增量发布

类似 1.2、1.3 等版本的稳定版本通常被认为是对语言进行重大更改的特性版本。通常，在特性发布之间会发布增量发布，编号为 1.2.20、1.2.30 等。

增量版本带来了工具方面的更新（通常包括特性），性能改进和错误修复。我们试图使这些版本彼此兼容，因此对编译器的更改主要是优化和添加/删除警告。实验特性可以随时被添加、删除或更改。

特性发布通常会添加新特性，并且可能会删除或更改以前不推荐使用的特性。某项特性从试验版到稳定版的过渡也包含在特性版本的发布中。

早期预览版本

在发布稳定版本之前，我们通常会发布许多称为 EAP（“Early Access Preview”）的早期预览版本，这些版本使我们能够更快地进行迭代并从社区中收集反馈。特性版本的早期预览版本通常会生成二进制文件，这些二进制文件随后将被稳定的编译器拒绝，以确保二进制文件中可能存在的错误只在预览期出现。最终发布的二进制文件通常没有此限制。

实验特性

根据上述反馈环原则，我们在语言的开放和发行版本中对设计进行迭代，其中某些特性具有实验性并且可以更改。实验特性可以随时被添加、更改或删除，不会发出警告。我们确保实验特性不会被用户意外使用。此类特性通常需要在代码或项目配置中进行某种类型的显式选择。

实验特性通常会在经过几次迭代后逐渐达到稳定状态。

不同组件的状态

要查看 Kotlin 的不同组件（Kotlin/JVM、JS、Native、各种库等）的稳定性状态，请查阅链接。

Libraries

A language is nothing without its ecosystem, so we pay extra attention to enabling smooth library evolution.

Ideally, a new version of a library can be used as a "drop-in replacement" for an older version. This means that upgrading a binary dependency should not break anything, even if the application is not recompiled (this is possible under dynamic linking).

On the one hand, to achieve this, the compiler has to provide certain ABI stability guarantees under the constraints of separate compilation. This is why every change in the language is examined from the point of view of binary compatibility.

On the other hand, a lot depends on the library authors being careful about which changes are safe to make. Thus it's very important that library authors understand how source changes affect compatibility and follow certain best practices to keep both APIs and ABIs of their libraries stable. Here are some assumptions that we make when considering language changes from the library evolution standpoint:

— Library code should always specify return types of public/protected functions and properties explicitly thus never relying on type inference for public API. Subtle changes in type inference may cause return types to change inadvertently, leading to binary compatibility issues.
— Overloaded functions and properties provided by the same library should do essentially the same thing. Changes in type inference may result in more precise static types to be known at call sites causing changes in overload resolution.

Library authors can use the @Deprecated and @Experimental annotations to control the evolution of their API surface. Note that @Deprecated(level=HIDDEN) can be used to preserve binary compatibility even for declarations removed from the API.

Also, by convention, packages named "internal" are not considered public API. All API residing in packages named "experimental" is considered experimental and can change at any moment.

We evolve the Kotlin Standard Library (kotlin-stdlib) for stable platforms according to the principles stated above. Changes to the contracts for its API undergo the same procedures as changes in the language itself.

Compiler Keys

Command line keys accepted by the compiler are also a kind of public API, and they are subject to the same considerations. Supported flags (those that don't have the "-X" or "-XX" prefix) can be added only in feature releases and should be properly deprecated before removing them. The "-X" and "-XX" flags are experimental and can be added and removed at any time.

Compatibility Tools

As legacy features get removed and bugs fixed, the source language changes, and old code that has not been properly migrated may not compile any more. The normal deprecation cycle allows a comfortable period of time for migration, and even when it's over and the change ships in a stable version, there's still a way to compile unmigrated code.

Compatibility flags

We provide the -language-version and -api-version flags that make a new version emulate the behaviour of an old one, for compatibility purposes. Normally, at least one previous version is supported. This effectively leaves a time span of two full feature release cycles for migration (which usually amounts to about two years). Using an older kotlin-stdlib or kotlin-reflect with a newer compiler without specifying compatibility flags is not recommended, and the compiler will report a warning when this happens.

Actively maintained code bases can benefit from getting bug fixes ASAP, without waiting for a full deprecation cycle to complete. Currently such project can enable the -progressive flag and get such fixes enabled even in incremental releases.

All flags are available on the command line as well as Gradle and Maven.

Evolving the binary format

Unlike sources that can be fixed by hand in the worst case, binaries are a lot harder to migrate, and this makes backwards compatibility very important in the case of binaries. Incompatible changes to binaries can make updates very uncomfortable and thus should be introduced with even more care than those in the source language syntax.

For fully stable versions of the compiler the default binary compatibility protocol is the following:

- All binaries are backwards compatible, i.e. a newer compiler can read older binaries (e.g. 1.3 understands 1.0 through 1.2),
- Older compilers reject binaries that rely on new features (e.g. a 1.0 compiler rejects binaries that use coroutines).
- Preferably (but we can't guarantee it), the binary format is mostly forwards compatible with the next feature release, but not later ones (in the cases when new features are not used, e.g. 1.3 can understand most binaries from 1.4, but not 1.5).

This protocol is designed for comfortable updates as no project can be blocked from updating its dependencies even if it's using a slightly outdated compiler.

Please note that not all target platforms have reached this level of stability (but Kotlin/JVM has).

不同组件的稳定性

依据组件的发展速度，可以有不同的稳定性模式：

- **快速流转（MF, Moving fast）**：即使在增量版本之间也不要期待任何兼容性，任何功能都可以在没有警告的情况下添加、删除或者更改。
- **有功能添加的增量版本（AIR, Additions in Incremental Releases）**：可以在增量版本中添加内容，应避免删除与更改行为，而如果必须要删改的话，应在之前的增量版本中预告。
- **稳定增量版本（SIR, Stable Incremental Releases）**：增量版本完全兼容，只会有优化与 bug 修复。可以在特性版本中进行任何更改。

- **完全稳定（FS, Fully Stable）**：增量版本完全兼容，只会有优化与 bug 修复。特性版本兼容旧版。

对于相同的组件，源代码兼容性与二进制兼容性可以有不同的模式，例如，在二进制格式稳定之前，源代码语言可以达到完全稳定，反之亦然。

Kotlin 演进制度的条款只适用于已经达到完全稳定（FS）的组件。从那一刻起，不兼容的变更必须得到语言委员会的批注。

组件	进入该状态的版本	源代码兼容模式	二进制兼容模式
Kotlin/JVM	1.0	FS	FS
kotlin-stdlib（JVM）	1.0	FS	FS
KDoc 语法	1.0	FS	N/A
协程	1.3	FS	FS
kotlin-reflect（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
多平台项目	1.2	MF	MF
内联类	1.3	MF	MF
无符号算术	1.3	MF	MF
默认情况下，所有其他实验性特性	N/A	**MF**	**MF**

Compatibility Guide for Kotlin 1.3

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructions which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.2 to Kotlin 1.3

Basic terms

In this document we introduce several kinds of compatibility:

— Source: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore

— Binary: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors

— Behavioral: a change is said to be behavioral-incompatible if one and the same program demonstrates different behavior before and after applying the change

One has to remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (e.g. from Java) is out of the scope of this document.

Evaluation order of constructor arguments regarding `<clinit>` call

Issue: KT-19532

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: evaluation order with respect to class initialization is changed in 1.3

Deprecation cycle:

— <1.3: old behavior (see details in the Issue)

— >= 1.3: behavior changed, `-Xnormalize-constructor-calls=disable` can be used to temporarily revert to pre-1.3 behavior. Support for this flag is going to be removed in the next major release.

Missing getter-targeted annotations on annotation constructor parameters

Issue: KT-25287

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: getter-target annotations on annotations constructor parameters will be properly written to classfiles in 1.3

Deprecation cycle:

— <1.3: getter-target annotations on annotation constructor parameters are not applied

— >=1.3: getter-target annotations on annotation constructor parameters are properly applied and written to the generated code

Missing errors in class constructor's `@get:` annotations

Issue: KT-19628

Component: Core language

Incompatible change type: Source

Short summary: errors in getter-target annotations will be reported properly in 1.3

Deprecation cycle:

— <1.2: compilation errors in getter-target annotations were not reported, causing incorrect code to be compiled fine.

— 1.2.x: errors reported only by tooling, the compiler still compiles such code without any warnings

— >=1.3: errors reported by the compiler too, causing erroneous code to be rejected

Nullability assertions on access to Java types annotated with `@NotNull`

Issue: KT-20830

Component: Kotlin/JVM

Incompatible change type: Behavioral

Short summary: nullability assertions for Java-types annotated with not-null annotations will be generated more aggressively, causing code which passes `null` here to fail faster.

Deprecation cycle:

— <1.3: the compiler could miss such assertions when type inference was involved, allowing potential `null` propagation during compilation against binaries (see Issue for details).

— >=1.3: the compiler generates missed assertions. This can case code which was (erroneously) passing `null`s here fail faster.
`-XXLanguage:-StrictJavaNullabilityAssertions` can be used to temporarily return to the pre-1.3 behavior. Support for this flag will be removed in the next major release.

Unsound smartcasts on enum members

Issue: KT-20772

Component: Core language

Incompatible change type: Source

Short summary: a smartcast on a member of one enum entry will be correctly applied to only this enum entry

Deprecation cycle:

— <1.3: a smartcast on a member of one enum entry could lead to an unsound smartcast on the same member of other enum entries.

— >=1.3: smartcast will be properly applied only to the member of one enum entry.
`-XXLanguage:-SoundSmartcastForEnumEntries` will temporarily return old behavior. Support for this flag will be removed in the next major release.

`val` backing field reassignment in getter

Issue: KT-16681

Components: Core language

Incompatible change type: Source

Short summary: reassignment of the backing field of `val`-property in its getter is now prohibited

Deprecation cycle:

— <1.2: Kotlin compiler allowed to modify backing field of `val` in its getter. Not only it violates Kotlin semantic, but also generates ill-behaved JVM bytecode which reassigns `final` field.

— 1.2.X: deprecation warning is reported on code which reassigns backing field of `val`

— >=1.3: deprecation warnings are elevated to errors

Array capturing before the `for`-loop where it is iterated

Issue: KT-21354

Component: Kotlin/JVM

Incompatible change type: Source

Short summary: if an expression in for-loop range is a local variable updated in a loop body, this change affects loop execution. This is inconsistent with iterating over other containers, such as ranges, character sequences, and collections.

Deprecation cycle:

— <1.2: described code patterns are compiled fine, but updates to local variable affect loop execution

— 1.2.X: deprecation warning reported if a range expression in a for-loop is an array-typed local variable which is assigned in a loop body

— 1.3: change behavior in such cases to be consistent with other containers

Nested classifiers in enum entries

Issue: KT-16310

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, nested classifiers (classes, object, interfaces, annotation classes, enum classes) in enum entries are prohibited

Deprecation cycle:

— <1.2: nested classifiers in enum entries are compiled fine, but may fail with exception at runtime

— 1.2.X: deprecation warnings reported on the nested classifiers

— >=1.3: deprecation warnings elevated to errors

Data class overriding copy

Issue: KT-19618

Components: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, data classes are prohibited to override `copy()`

Deprecation cycle:

— <1.2: data classes overriding `copy()` are compiled fine but may fail at runtime/expose strange behavior

— 1.2.X: deprecation warnings reported on data classes overriding `copy()`

— >=1.3: deprecation warnings elevated to errors

Inner classes inheriting `Throwable` that capture generic parameters from the outer class

Issue: KT-17981

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, inner classes are not allowed to inherit `Throwable`

Deprecation cycle:

— <1.2: inner classes inheriting `Throwable` are compiled fine. If such inner classes happen to capture generic parameters, it could lead to strange code patterns which fail at runtime.

— 1.2.X: deprecation warnings reported on inner classes inheriting `Throwable`

— >=1.3: deprecation warnings elevated to errors

Visibility rules regarding complex class hierarchies with companion objects

Issues: KT-21515, KT-25333

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, rules of visibility by short names are stricter for complex class hierarchies involving companion objects and nested classifiers.

Deprecation cycle:

- <1.2: old visibility rules (see Issue for details)
- 1.2.X: deprecation warnings reported on short names which are not going to be accessible anymore. Tooling suggests automated migration by adding full name.
- >=1.3: deprecation warnings elevated to errors. Offending code should add full qualifiers or explicit imports

Non-constant vararg annotation parameters

Issue: KT-23153

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, setting non-constant values as vararg annotation parameters is prohibited

Deprecation cycle:

- <1.2: the compiler allows to pass non-constant value for vararg annotation parameter, but actually drops that value during bytecode generation, leading to non-obvious behavior
- 1.2.X: deprecation warnings reported on such code patterns
- >=1.3: deprecation warnings elevated to errors

Local annotation classes

Issue: KT-23277

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 local annotation classes are not supported

Deprecation cycle:

- <1.2: the compiler compiled local annotation classes fine
- 1.2.X: deprecation warnings reported on local annotation classes
- >=1.3: deprecation warnings elevated to errors

Smartcasts on local delegated properties

Issue: KT-22517

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 smartcasts on local delegated properties are not allowed

Deprecation cycle:

— <1.2: the compiler allowed to smartcast local delegated property, which could lead to unsound smartcast in case of ill-behaved delegates

— 1.2.X: smartcasts on local delegated properries are reported as deprecated (the compiler issues warnings)

— >=1.3: deprecation warnings elevated to errors

`mod` operator convention

Issues: KT-24197

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 declaration of `mod` operator is prohibited, as well as calls which resolve to such declarations

Deprecation cycle:

— 1.1.X, 1.2.X: report warnings on declarations of `operator mod`, as well as on calls which resolve to it

— 1.3.X: elevate warnings to error, but still allow to resolve to `operator mod` declarations

— 1.4.X: do not resolve calls to `operator mod` anymore

Passing single element to vararg in named form

Issues: KT-20588, KT-20589. See also KT-20171

Component: Core language

Incompatible change type: Source

Short summary: in Kotlin 1.3, assigning single element to vararg is deprecated and should be replaced with consecutive spread and array construction.

Deprecation cycle:

— <1.2: assigning one value element to vararg in named form compiles fine and is treated as assigning *single* element to array, causing non-obvious behavior when assigning array to vararg
— 1.2.X: deprecation warnings are reported on such assignments, users are suggested to switch to consecutive spread and array construction.
— 1.3.X: warnings are elevated to errors
— >= 1.4: change semantic of assigning single element to vararg, making assignment of array equivalent to the assignment of a spread of an array

Retention of annotations with target `EXPRESSION`

Issue: KT-13762

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, only `SOURCE` retention is allowed for annotations with target `EXPRESSION`

Deprecation cycle:

— <1.2: annotations with target `EXPRESSION` and retention other than `SOURCE` are allowed, but silently ignored at use-sites
— 1.2.X: deprecation warnings are reported on declarations of such annotations
— >=1.3: warnings are elevated to errors

Annotations with target `PARAMETER` shouldn't be applicable to parameter's type

Issue: KT-9580

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, error about wrong annotation target will be properly reported when annotation with target `PARAMETER` is applied to parameter's type

Deprecation cycle:

— <1.2: aforementioned code patterns are compiled fine; annotations are silently ignored and not present in the bytecode
— 1.2.X: deprecation warnings are reported on such usages
— >=1.3: warnings are elevated to errors

`Array.copyOfRange` throws an exception when indices are out of bounds instead of enlarging the returned array

Issue: KT-19489

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, ensure that the `toIndex` argument of `Array.copyOfRange`, which represents the exclusive end of the range being copied, is not greater than the array size and throw `IllegalArgumentException` if it is.

Deprecation cycle:

— <1.3: in case `toIndex` in the invocation of `Array.copyOfRange` is greater than the array size, the missing elements in range fill be filled with `null`s, violating soundness of the Kotlin type system.
— >=1.3: check that `toIndex` is in the array bounds, and throw exception if it isn't

Progressions of ints and longs with a step of `Int.MIN_VALUE` and `Long.MIN_VALUE` are outlawed and won't be allowed to be instantiated

Issue: KT-17176

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, prohibit step value for integer progressions being the minimum negative value of its integer type (`Long` or `Int`), so that calling `IntProgression.fromClosedRange(0, 1, step = Int.MIN_VALUE)` will throw `IllegalArgumentException`

Deprecation cycle:

— <1.3: it was possible to create an `IntProgression` with `Int.MIN_VALUE` step, which yields two values `[0, -2147483648]`, which is non-obvious behavior

— >=1.3: throw `IllegalArgumentException` if the step is the minimum negative value of its integer type

Check for index overflow in operations on very long sequences

Issue: KT-16097

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, make sure `index`, `count` and similar methods do not overflow for long sequences. See the Issue for the full list of affected methods.

Deprecation cycle:

— <1.3: calling such methods on very long sequences could produce negative results due to integer overflow

— >=1.3: detect overflow in such methods and throw exception immediately

Unify split by an empty match regex result across the platforms

Issue: KT-21049

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, unify behavior of `split` method by empty match regex across all platforms

Deprecation cycle:

— <1.3: behavior of described calls is different when comparing JS, JRE 6, JRE 7 versus JRE 8+

— >=1.3: unify behavior across the platforms

Discontinued deprecated artifacts in the compiler distribution

Issue: KT-23799

Component: other

Incompatible change type: Binary

Short summary: Kotlin 1.3 discontinues the following deprecated binary artifacts:

— `kotlin-runtime`: use `kotlin-stdlib` instead

— `kotlin-stdlib-jre7/8`: use `kotlin-stdlib-jdk7/8` instead

— `kotlin-jslib` in the compiler distribution: use `kotlin-stdlib-js` instead

Deprecation cycle:

— 1.2.X: the artifacts were marked as deprecated, the compiler reported warning on usage of those artifacts

— >=1.3: the artifacts are discontinued

Annotations in stdlib

Issue: KT-21784

Component: kotlin-stdlib (JVM)

Incompatible change type: Binary

Short summary: Kotlin 1.3 removes annotations from the package `org.jetbrains.annotations` from stdlib and moves them to the separate artifacts shipped with the compiler: `annotations-13.0.jar` and `mutability-annotations-compat.jar`

Deprecation cycle:

— <1.3: annotations were shipped with the stdlib artifact

— >=1.3: annotations ship in separate artifacts

常见问题

FAQ

Kotlin 是什么?

Kotlin 是一门面向 JVM、Android、JavaScript 以及原生平台的开源（OSS）静态类型编程语言。它是由 JetBrains 开发的。该项目开始于 2010 年并且很早就已开源。第一个官方 1.0 版发布于 2016 年 2 月。

Kotlin 的当前版本是多少?

目前发布的版本是 1.3.72，发布于 2020-04-15。

Kotlin 是免费的吗?

是。Kotlin 是免费的，已经免费并会保持免费。它是遵循 Apache 2.0 许可证开发的，其源代码可以在 GitHub 上获得。

Kotlin 是面向对象还是函数式语言?

Kotlin 既具有面向对象又具有函数式结构。你既可以按 OO 风格也可以按 FP 风格使用，还可以混合使用两种风格。通过对诸如高阶函数、函数类型和 lambda 表达式等功能的一等支持，Kotlin 是一个很好的选择，如果你正在进行或探索函数式编程的话。

Kotlin 能给我超出 Java 语言的哪些优点?

Kotlin 更简洁。粗略估计显示，代码行数减少约 40%。它也更安全，例如对不可空类型的支持使应用程序不易发生 NPE。其他功能包括智能类型转换、高阶函数、扩展函数和带接收者的 lambda 表达式，提供了编写富于表现力的代码的能力以及易于创建 DSL 的能力。

Kotlin 与 Java 语言兼容吗?

兼容。Kotlin 与 Java 语言可以 100% 互操作，并且主要强调确保你现有的代码库可以与Kotlin 正确交互。你可以轻松地在 Java 中调用 Kotlin 代码以及在 Kotlin 中调用 Java 代码。这使得采用 Kotlin 更容易、风险更低。内置于 IDE 的自动化 Java 到 Kotlin 转换器可简化现有代码的迁移。

我可以用 Kotlin 做什么?

Kotlin 可用于任何类型的开发，无论是服务器端、客户端 Web 还是 Android。随着原生 Kotlin（Kotlin/Native）目前的进展，对其他平台（如嵌入式系统、macOS 和 iOS）的支持即将就绪。人们将 Kotlin 用于移动端和服务器端应用程序、使用 JavaScript 或 JavaFX的客户端、以及数据科学，仅举这几例。

我可以用 Kotlin 进行 Android 开发吗?

可以。Kotlin 已作为 Android 平台的一等语言而支持。已经有数百种应用程序在使用 Kotlin 用于 Android 开发，比如 Basecamp、Pinterest 等等。更多信息请查看 Android 开发资源。

我可以用 Kotlin 进行服务器端开发吗?

可以。Kotlin 与 JVM 100% 兼容，因此你可以使用任何现有的框架，如 Spring Boot、vert.x 或 JSF。另外还有一些 Kotlin 写的特定框架，例如 Ktor。更多信息请查看服务器端开发资源。

我可以用 Kotlin 进行 web 开发吗?

可以。除了用于后端 Web，你还可以使用 Kotlin/JS 用于客户端 Web。Kotlin 可以使用 DefinitelyTyped 中的定义来获取常见 JavaScript 库的静态类型版，并且它与现有的模块系统（如 AMD 和 CommonJS）兼容。更多信息请查看客户端开发中的资源。

我可以用 Kotlin 进行桌面开发吗?

可以。你可以使用任何 Java UI 框架如 JavaFx、Swing 或其他框架。另外还有 Kotlin 特定框架，如 TornadoFX。

我可以用 Kotlin 进行原生开发吗?

可以。Kotlin/Native 是 Kotlin 项目的一部分。它将 Kotlin 编译成无需虚拟机（VM）即可运行的原生代码。仍处于 beta 测试阶段，不过已经可以在主流的桌面与移动端平台甚至某些物联网（IoT）设备上试用。更多详细信息请查阅 Kotlin/Native 文档。

哪些 IDE 支持 Kotlin?

所有主要的 Java IDE 都支持 Kotlin，包括 IntelliJ IDEA、Android Studio、Eclipse 和 NetBeans。另外，有一个命令行编译器可用，为编译和运行应用程序提供了直接的支持。

哪些构建工具支持 Kotlin?

在 JVM 端，主要构建工具包括 Gradle、Maven、Ant 和 Kobalt。还有一些可用于构建客户端 JavaScript 的构建工具。

Kotlin 会编译成什么?

当面向JVM 平台时，Kotlin 生成 Java 兼容的字节码。当面向 JavaScript 时，Kotlin 会转译到 ES5.1，并生成与包括 AMD 和 CommonJS 在内的模块系统相兼容的代码。当面向原生平台时，Kotlin 会（通过 LLVM）生成平台相关的代码。

Kotlin 面向哪些版本的 JVM?

Kotlin 会让你选择用于执行的 JVM 版本。默认情况下，Kotlin/JVM 编译器会生成兼容 Java 6 的字节码。如果要利用 Java 新版本中提供的优化功能，可以将目标 Java 版本显式指定为 8 到 13。请注意，这种情况下生成的字节码可能无法在较低版本中运行。

Kotlin 难吗?

Kotlin 是受 Java、C#、JavaScript、Scala 以及 Groovy 等现有语言的启发。我们已经努力确保 Kotlin 易于学习，所以人们可以在几天之内轻松转向、阅读和编写 Kotlin。学习惯用的 Kotlin 和使用更多它的高级功能可能需要一点时间，但总体来说这不是一个复杂的语言。

哪些公司使用 Kotlin?

有太多使用 Kotlin 的公司可列，而有些更明显的公司已经公开宣布使用 Kotlin，分别通过博文、Github 版本库或者演讲宣布，包括 Square、Pinterest、Basecamp 还有 Corda。

谁开发 Kotlin?

Kotlin 主要由 JetBrains 的一个工程师团队开发（目前团队规模为 100+）。其首席语言设计师是 Andrey Breslav。除了核心团队，GitHub 上还有 250 多个外部贡献者。

在哪里可以了解关于 Kotlin 更多?

最好的起始地方好是本网站（原文是英文官网）。从那里你可以下载编译器、在线尝试以及访问资源、参考文档和教程。

有没有关于 Kotlin 的书?

已经有一些关于 Kotlin 的图书。其中包括由 Kotlin 团队成员 Dmitry Jemerov 和 Svetlana Isakova 合著的 Kotlin in Action、面向 Android 开发人员的 Kotlin for Android Developers。

Kotlin 有没有在线课程?

有一些 Kotlin 的课程，包括 Kevin Jones 的 Pluralsight Kotlin Course、Hadi Hariri 的 O' Reilly Course 以及 Peter Sommerhoff 的 Udemy Kotlin Course。

在 YouTube 和 Vimeo 上也有许多 Kotlin 演讲 的录像。

有没有 Kotlin 社区?

有。Kotlin 有一个非常有活力的社区。Kotlin 开发人员常出现在 Kotlin 论坛、StackOverflow 上并且更积极地活跃在 Kotlin Slack（截至 2020 年 4 月有近 30000 名成员）上。

有没有 Kotlin 活动?

有。现在有很多用户组和集会组专注于 Kotlin。你可以在网站上找到一个列表。此外，还有世界各地的社区组织的 Kotlin 之夜活动。

有没有 Kotlin 大会?

有。官方的年度 KotlinConf 由 JetBrains 主办。分别于 2017 年 在旧金山、2018 年在阿姆斯特丹、2019 年在哥本哈根举行。Kotlin 也会在全球不同地方举行大会。你可以在网站上找到即将到来的会谈列表。

Kotlin 上社交媒体吗?

上。最活跃的 Kotlin 帐号是 Twitter 上的。

其他在线 Kotlin 资源呢?

网站上有一堆在线资源，包括社区成员的 Kotlin 文摘、通讯、播客等等。

在哪里可以获得高清 Kotlin 徽标?

徽标可以在这里下载。使用该徽标时，请遵循压缩包中的 `guidelines.pdf` 以及 Kotlin 品牌使用指南 中的简单规则。

与 Java 语言比较

Kotlin 解决了一些 Java 中的问题

Kotlin 通过以下措施修复了 Java 中一系列长期困扰我们的问题：

- 空引用由类型系统控制。
- 无原始类型
- Kotlin 中数组是不型变的
- 相对于 Java 的 SAM-转换，Kotlin 有更合适的函数类型
- 没有通配符的使用处型变
- Kotlin 没有受检异常

Java 有而 Kotlin 没有的东西

- 受检异常
- 不是类的原生类型 —— 字节码会尽可能试用原生类型，但不是显式可用的。
- 静态成员 —— 以 伴生对象、顶层函数、扩展函数 或者 @JvmStatic 取代。
- 通配符类型 —— 以 声明处协变 与 类型投影 取代。
- 三目操作符 a ? b : c —— 以 if 表达式取代。

Kotlin 有而 Java 没有的东西

- Lambda 表达式 + 内联函数 = 高性能自定义控制结构
- 扩展函数
- 空安全
- 智能类型转换
- 字符串模板
- 属性
- 主构造函数
- 一等公民的委托
- 变量与属性类型的类型推断
- 单例
- 声明处型变 & 类型投影
- 区间表达式
- 操作符重载
- 伴生对象
- 数据类
- 分离用于只读与可变集合的接口
- 协程

与 Scala 比较【官方已删除】

Kotlin 团队的主要目标是创建一种务实且高效的编程语言，而不是提高编程语言研究中的最新技术水平。考虑到这一点，如果你对 Scala 感到满意，那你很可能不需要 Kotlin。

Scala 有而 Kotlin没有的东西

- 隐式转换、参数……等等
  - 在 Scala 中，由于画面中有太多的隐式转换，有时不使用调试器会很难弄清代码中具体发生了什么
  - 在 Kotlin 中使用扩展函数来给类型扩充功能/函数（双关：functions）。
- 可覆盖的类型成员
- 路径依赖性类型
- 宏
- 存在类型
  - 类型投影是一种非常特殊的情况
- 特质（trait）初始化的复杂逻辑
  - 参见类与接口
- 自定义符号操作
  - 参见操作符重载
- 结构类型
- 值类型
  - 我们计划支持 Project Valhalla——当它作为 JDK 一部分发布时。
- yield 操作符与 actor
  - 参见协程
- 并行集合
  - Kotlin 支持 Java 8 stream，它提供了类似的功能

Kotlin 有而 Scala 没有的东西

- 零开销空安全
  - Scala 有 Option，它是一个语法糖以及运行时的包装器
- 智能转换
- Kotlin的内联函数便于非局部跳转
- 头等委托。也通过第三方插件 Autoproxy 实现