Broadview
www.broadview.com.cn

MANNING

Kotlin 实战

[俄] Dmitry Jemerov
Svetlana Isakova 著

覃宇 罗丽 李思阳 蒋扬海 译

Kotlin in Action



中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Kotlin实战

阜阳师范学院图书馆
藏书

[俄] Dmitry Jemerov
Svetlana Isakova 著

覃宇 罗丽 李思阳 蒋扬海 译

Kotlin in Action

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书将从语言的基本特性开始，逐渐覆盖其更多的高级特性，尤其注重讲解如何将 Koltin 集成到已有 Java 工程实践及其背后的原理。本书分为两个部分。第一部分讲解如何开始使用 Kotlin 现有的库和 API，包括基本语法、扩展函数和扩展属性、数据类和伴生对象、lambda 表达式，以及数据类型系统（着重讲解了可空性和集合的概念）。第二部分教你如何使用 Kotlin 构建自己的 API，以及一些深层次特性——约定和委托属性、高阶函数、泛型、注解和反射，以及领域特定语言的构建。本书适合广大移动开发者及入门学习者，尤其是紧跟主流趋势的前沿探索者。

Original English Language edition published by Manning Publications, USA. Copyright © 2017 by Manning Publications. Simplified Chinese-language edition copyright © 2017 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由 Manning Publications 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2017-4424

图书在版编目（CIP）数据

Kotlin 实战 /（俄罗斯）德米特里•詹莫瑞福（Dmitry Jemerov），（俄罗斯）斯维特拉娜•伊凡诺沃（Svetlana Isakova）著；覃宇等译 . —北京：电子工业出版社，2017.8
书名原文：Kotlin in Action
ISBN 978-7-121-32158-0

Ⅰ . ① K… Ⅱ . ①德… ②斯… ③覃… Ⅲ . ① JAVA 语言－程序设计 Ⅳ . ① TP312.8

中国版本图书馆 CIP 数据核字 (2017) 第 165490 号

责任编辑：张春雨
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：22.5　字数：496 千字
版　　次：2017 年 8 月第 1 版
印　　次：2017 年 8 月第 2 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

译者序

当收到这本书的翻译邀请时，我们的内心是激动的，终于有机会将自己喜爱的语言系统地介绍给中国的开发者，而且是通过口碑颇佳的实战系列。此时，正值2017年度的Google I/O召开前夕，接下来重磅消息大家都知道了：在Google I/O大会上，Kotlin正式成为了官方的Android开发语言，迅速占据了国内各大技术媒体的头条。一夜之间，所有的Android开发者都迫切地想搞清楚它的来龙去脉。Kotlin究竟是何方神圣，为什么是它？

这一点儿也不奇怪。对于资深Android开发者来说，Kotlin早已不是新鲜的概念了。早在2015年1月，Android开发者社区大神Jake Wharton就发布了一篇使用Kotlin来进行Android开发的总结。那时开始，不少顶尖的开发者和公司就开始尝试在正式的Android项目中使用Kotlin语言；我们也从2015年开始在多个项目上使用了Kotlin语言。它带给我们的体验，和带给所有其他实践过Kotlin语言的开发者的一样：它的发明者JetBrains所言非虚，这是一门简洁、安全、实用的语言，用了就停不下来，就忍不住地想推荐给周围的人。我们理所当然地把Kotlin放在了今年第一季度ThoughtWorks技术雷达的评估象限：https://www.thoughtworks.com/radar/languages-and-frameworks/kotlin。

Kotlin让人爱不释手的最重要原因就是来自JetBrains的基因。作为最负盛名的IDE创造者，JetBrains深谙开发者的需求，孜孜不倦地追求给开发者提供最实用、最高效的IDE，包括Android Studio、IntelliJ、RubyMine等。由这样想开发者之所想的公司创造出来的语言，又怎么会不受开发者热捧呢？所以Gradle、Spring，以

及越来越多的库、框架和工具也陆续加入到了支持 Kotlin 的阵营。

本书深入浅出地介绍了 Kotlin 语言的方方面面，从最基础的语言要素到如何定制自己的 DSL 都有涉及。相信读者阅读本书并尝试之后一定会爱上这门语言，但把 Kotlin 应用到自己的项目中会不会有什么风险呢？读者们大可不必担心，以往的经验告诉我们，整个过程无缝无痛。首先，Kotlin 足够简单，对于初学者来说掌握也不算困难，两三天就可以上手；其次，Kotlin 和 Java 可以无缝地衔接，可以在遗留项目上和 Java 混用；最后，编译器的静态检查和 IDE（必须是 JetBrains 出品的 IntelliJ IDEA 或者 Android Studio）强大的辅助功能，可以帮你发现很多问题（例如空指针异常）并将其自动消除在摇篮之中。有的读者会说，但我还没有用过这些 IDE 啊？那你还在犹豫什么，请立即使用它们来提高你的生产力吧！这也算是使用 Kotlin 带来的额外收获。

从 Kotlin 成为 Android 开发语言的那一刻开始，我们热情高涨地投入了几乎全部业余时间到本书的翻译工作，终于在最短的时间内把它呈现在广大读者面前。这一切还要感谢本书的编辑和所有译者家人在背后的默默付出。由于译者水平所限，难免出现谬误遗漏，还望读者海涵斧正。

覃宇、罗丽、李思阳、蒋扬海

2017 年 6 月于 ThoughtWorks 成都

序

当我在2010年春季第一次拜访JetBrains的时候，我相当确定世界上不需要另一种通用编程语言了。我认为现有的JVM上的语言已经足够好了，谁会有想法去创建一门新语言呢？在经过大约一个小时的关于大规模代码库上产品问题的讨论后我被完全说服了，并且后来成为Kotlin一部分的最初想法就已经被描绘在白板上。很快我就加入了JetBrains来主导这门语言的设计与编译器的开发工作。

到今天，六年多的时光过去了，我们也快要发布第二个版本。我们已经拥有超过30人的团队和数以千计的活跃用户，还有很多让我们难以轻易实现的精彩的设计理念。但是不要担心，这些想法在进入这门语言之前还必须经过缜密的考察。我们希望这本书的篇幅依然能够容得下Kotlin的未来。

学习一门编程语言是一个令人兴奋而且常常是回报颇丰的尝试。如果它是你的第一门语言，通过它你能学到整个编程的新世界。如果不是，它会使你以新的术语来思考熟悉的东西，从而以更高层次的抽象来更深入地了解它们。本书主要针对后者，即面向已经熟悉Java的读者。

从头开始设计一门语言可能是一项具有挑战性的任务，但是使其与另一门语言融洽的工作就是另一回事了——尤其是那门语言还包含了许多的愤怒的食人魔，以及一些阴暗的地牢（在这一点上你如果不相信可以去问C++的创造者Bjarne Stroustrup）。与Java的互操作性（这就是Java与Kotlin之间是如何互相混合调用的）是Kotlin的基石之一，本书也投入了很多的注意力在这一点上。互操作性对于在一个已有的Java代码库中逐步地引入Kotlin非常重要。即使从头开始开发一个新项目

时，也必须考虑到能够将这门语言融入一个拥有更大图景的平台中去，而以 Java 编写的所有函数库就是这样的一个平台。

当我在编写本书时，两个新的目标平台正在开发：Kotlin 现在可以在 JavaScript 虚拟机上运行以支持全栈 web 开发，并且还将很快能够直接编译成原生代码，从而在需要的时候能够脱离任何的虚拟机来运行。[1] 所以，虽然本书是面向 JVM 的，但是你从中学到的很多东西也是可以应用于其他运行环境的。

本书作者从项目伊始就已经是 Kotlin 团队的成员，所以他们对语言本身和内部实现非常熟悉。他们在会议演讲、研讨会及 Kotlin 课程方面的经验使他们能够对预期的常见问题及可能的陷阱，提供良好的阐述。本书既阐释了语言特征背后的高级概念，也提供了足够深入的细节。

希望你能享受与我们的语言及本书相处的时光。正如我经常在我们社区的帖子中说的那样：使用 Kotlin 愉快！

ANDREY BRESLAV
JetBrains Kotlin 首席设计师

1 截止本书翻译时，对 JavaScript 的支持已经随着 Kotlin 1.1 发布了，参见：https://blog.jetbrains.com/kotlin/2017/03/kotlin-1-1/。而对 Native 的支持也发布了技术预览版，参见：https://blog.jetbrains.com/kotlin/2017/04/kotlinnative-tech-preview-kotlin-without-a-vm/。——译者注

前言

关于 Kotlin 想法的构思 2010 年诞生于 JetBrains。当时，JetBrains 已经是许多程序语言开发工具的知名供应商，包括 Java、C#、JavaScript、Python、Ruby 和 PHP。Java IDE——IntelliJ IDEA，Groovy 和 Scala 的插件，都是我们的旗舰产品。

为各种程序语言构建开发工具的经验给了我们对语言设计领域全面的理解和独特的观点。而基于 IntelliJ 平台的 IDE，包括 IntelliJ IDEA，仍然是用 Java 开发的。我们甚至都有点羡慕在 .NET 团队中的同事，他们使用 C#，一种现代、强大、迅速进化的语言进行开发。但是我们没有看到任何一种可以用来取代 Java 的语言。

对于这样的一门语言我们有哪些要求呢？首要而且最明确的要求就是它必须是静态类型的。我们想象不到其他任何一种——开发一个拥有数百万行代码的代码库许多年后——还不把人逼疯的办法。其次，我们需要与现有的 Java 代码完全兼容。这样的代码库是 JetBrains 的一笔巨大财富，我们承受不起失去它或是因为互操作性的难度而使其贬值的损失。再次，我们不愿意在工具质量方面接受任何的妥协。开发者的生产力是 JetBrains 作为一个公司最重要的价值，而强大的工具是达到这一目的的必要条件。最后，我们需要的是一种易于学习和理解的语言。

当看到一个我们公司未能满足的需要时，我们知道其他公司也处在一个相似的境地，我们希望我们的解决方案能够在 JetBrains 之外找到许多用户。带着这样的初心，我们决定走上一条创建一门新语言：Kotlin 的道路。事实上，这个项目花费了超出我们预期的时间，在 Kotlin 1.0 最终诞生时，距离第一行代码提交到代码库中已经过去了超过五年；但是现在我们可以确信，这门语言找到了它的受众并且这些

人都留了下来。

Kotlin 以靠近俄罗斯圣彼得堡的一座岛屿命名，Kotlin 的大部分开发团队就在那里。在使用岛屿命名这件事上，我们遵循了 Java 和 Ceylon 确立的先例，但我们决定选用一处靠近我们家乡的地方（在英语中，这个名称通常的发音是“cot-lin”，而不是“coat-lin”或者“caught-lin”）。

在这门语言临近发布之际，我们意识到一本由参与了语言设计决策人员撰写的关于 Kotlin 的书籍是有价值的，他们可以自信地解释为什么 Kotlin 中的事物是以它们的方式运行的。本书就是这种努力的结果，我们希望它能帮助你学习和理解 Kotlin 语言。祝你好运，并愿你一直能愉快地进行开发。

致谢

首先，我们想感谢 Sergey Dmitriev 和 Max Shafirov，感谢他们相信一门新语言的想法并决定投入 JetBrains 的资源。没有他们，这门语言和这本书将不会存在。

我们要特别感谢 Andrey Breslav，他是负责设计这门令人能够愉快地书写（以及编码）的语言的主要人物。Andrey 在领导这支持续增长的 Kotlin 团队之余，也给了我们很多有效的反馈，这让我们非常感激。另外，你可以放心，本书收到了来自这位首席设计师的认可，他爽快地答应为本书撰写序。

我们非常感谢 Manning 团队，他们引导我们完成这本书的编写过程，并为文字可读性和结构合理性提供了帮助——具体包括，我们的开发编辑 Dan Maharry，尽管我们日程安排非常忙碌，他仍然勇于力争寻找时间跟我们讨论，同样还有 Michael Stephens、Helen Stergius、Kevin Sullivan、Tiffany Taylor、Elizabeth Martin 和 Marija Tudor。来自我们技术审核专员 Brent Watson 和 Igor Wojda 的反馈也是非常宝贵的，在开发过程中阅读原稿的审稿人的意见也很重要：Alessandro Campeis、Amit Lamba、Angelo Costa、Boris Vasile、Brendan Grainger、Calvin Fernandes、Christopher Bailey、Christopher Bortz、Conor Redmond、Dylan Scott、Filip Pravica、Jason Lee、Justin Lee、Kevin Orr、Nicolas Frankel、Paweł Gajda、Ronald Tischliar 和 Tim Lavers。感谢在 MEAP 计划和书籍论坛中提交反馈的所有人，我们已经根据你们的意见改进了文字。

我们要感谢整个 Kotlin 团队，在我们撰写这本书的过程中，他们每天都要听“又一个章节完成了”这样的日常报告。我们想感谢帮助我们计划这本书并在草案

上给出反馈的同事，尤其是 Ilya Ryzhenkov、Hadi Hariri、Michael Glukhikh 和 Ilya Gorbunov。我们也想感谢那些不仅给予帮助也同样阅读文本并提供反馈（有时在度假期间的滑雪胜地）的朋友：Lev Serebryakov、Pavel Nikolaev 和 Alisa Afonina。

最后，我们还想感谢我们的家人和使这个世界变得更美好的猫咪们。

关于本书

《*Kotlin* 实战》会教你 Kotlin 编程语言，以及如何使用它构建运行在 Java 虚拟机和 Android 平台的应用程序。这本书开始部分讲解了这门语言的基本特性，并逐渐覆盖更多 Kotlin 与众不同的方面，比如它对构建高级抽象和领域特定语言的支持。这本书很注重将 Kotlin 与已有的 Java 工程的集成，帮助你将 Kotlin 引入到你现在的工作环境。

这本书涵盖了 Kotlin 1.0，在编写这本书的同时 Kotlin 1.1 已经在开发过程中了，所以在一些可能的地方，我们提示了 1.1 版本中做出的更改。但是由于在写这本书的时候新的版本还没有完成，我们并不能在书中包含所有的内容。对于进行中的新特性和改变的更新，请参考线上的官方文档 https://kotlinlang.org。

哪些人应该阅读这本书

《*Kotlin* 实战》主要面向有一定 Java 经验的开发者。Kotlin 的构建基于 Java 中的许多概念和技术，这本书争取通过使用你现有的知识快速上手。如果你只是刚开始学习 Java，又或者你有诸如 C# 或者 JavaScript 这些编程语言的经验，你可能需要参考其他的信息源以理解 Kotlin 中与 JVM 交互的那些错综复杂的方面，但你还是可以通过这本书学习 Kotlin。我们致力于将 Kotlin 打造成全领域语言，而不是只针对某些特定的问题领域，所以这本书同样对服务端、Android，以及其他任何以构建基于 JVM 的工程为目标的开发人员都有用。

这本书是如何组织的

这本书被分成了两个部分。第 1 部分解释了如何开始使用 Kotlin 现有的库和 API：

- 第 1 章讲述了 Kotlin 的关键目标、价值和应用的领域，它将向你展示运行 Kotlin 代码的所有可能的途径。
- 第 2 章解释了 Kotlin 编程的基本元素，包括控制结构、变量和函数声明。
- 第 3 章讲解了 Kotlin 中关于函数声明的细节并介绍了扩展函数和扩展属性的概念。
- 第 4 章集中在类的声明上，并介绍了数据类和伴生对象的概念。
- 第 5 章介绍了 Kotlin 中 lambda 的使用并展示了一些 Kotlin 标准库中使用 lambda 的函数。
- 第 6 章描述 Kotlin 的数据类型系统，并特别关注了可空性和集合的话题。

第 2 部分会教你如何使用 Kotlin 构建你自己的 API 和抽象，并涵盖了这门语言的一些深层次的特性。

- 第 7 章讲到了约定原则，它利用特定的名字赋予函数和属性特殊的意义，还介绍了委托属性的概念。
- 第 8 章展示了如何声明高阶函数——以其他函数作为参数或者返回值的函数，还介绍了内联函数的概念。
- 第 9 章深入探讨 Kotlin 中泛型的话题，先讲了基本语法然后是更高级的领域，比如实化类型参数和变型。
- 第 10 章包括注解和反射的使用，并以 JKid 为中心。JKid 是大量使用了这些概念的一个小而真实的 JSON 序列化库。
- 第 11 章介绍了领域特定语言的概念，描述用来构建它的 Kotlin 工具，并演示了许多 DSL 示例。

还有三个附录。附录 A 说明了如何用 Gradle、Maven 和 Ant 构建 Kotlin 代码；附录 B 着重于编写文件注释和为 Kotlin 模块生成 API 文档；附录 C 是一个探索 Kotlin 生态圈和发现最新的在线信息的指南。

最好是按顺序通读本书，但是也完全可以只查阅感兴趣的包含特定主题的单个章节，在遇到不熟悉的概念的时候再参考其他章节。

编码规范和下载

这本书从头到尾都使用了以下的排版规范：

- 斜体字体用于介绍新的术语
- 等宽字体用于表示代码样本，还有函数名、类和其他的标识符
- 代码注释伴随着许多代码清单并强调重要概念

书中的许多代码清单还展示了代码的输出。在这些例子中，我们用 >>> 作为产生这些输出的代码行的前缀，而输出本身如下所示：

```
>>> println("Hello World")
Hello World
```

一些例子是完整的可运行的程序，而另外一些则是用于演示某些概念的代码片段，它们可能会包含省略部分（用……表示）或者语法错误（在书中的正文或者示例中被描述出来）。那些可运行的例子从发布者的网站 www.manning.com/books/kotlin-in-action 上以 zip 文件的格式下载。书中的这些例子也提前被加载到了在线环境中（http://try.kotlinlang.org），所以只需要在浏览器中点几下就能运行任何例子。

其他在线资源

Kotlin 有一个活跃的在线社区，所以如果你有问题或者想跟同样的 Kotlin 用户聊聊，可以使用以下资源：

- *Kotlin 官方论坛*——https://discuss.kotlinlang.org
- *Slack chat*——http://kotlinlang.slack.com（可以在 http://kotlinslack.herokuapp.com 得到邀请函）
- *Stack Overflow* 上的 Kotlin 标签——http://stackoverflow.com/questions/tagged/kotlin
- *Kotlin Reddit*——www.reddit.com/r/Kotlin

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与我们交流**：在页面下方【读者评论】处留下您的疑问或观点，与我们和其他读者一同学习交流。



页面入口：http://www.broadview.com.cn/32158

关于作者

Dmitry Jemerov 从 2003 年起就在 JetBrains 工作并参与了许多产品的开发，包括 IntelliJ IDEA、PyCharm 和 WebStorm。他是 Kotlin 最早的贡献者之一，创建了最初版本的 Kotlin JVM 字节码生成器，并且还在世界各地的活动上做了很多关于 Kotlin 的演示。目前他带领了进行 Kotlin IntelliJ IDEA 插件开发的团队。

Svetlana Isakova 从 2011 年成为 Kotlin 团队的一员。她从事编译器类型推导和重载解析子系统的工作。现在她是一名技术布道者，在各种会议上进行 Kotlin 相关讨论，并从事 Kotlin 在线课程的相关工作。

关于封面插图

《*Kotlin* 实战》封面上的画像名为“1764 年一位俄罗斯瓦尔达女士的着装”。瓦尔达城位于诺夫哥罗德州地区，在莫斯科与圣彼得堡之间的通道上。这幅图选自伦敦的 Thomas Jefferys 的《*A Collection of the Dresses of Different Nations, Ancient and Modern*》，出版于 1757 年到 1772 年之间。书籍首页说明了这幅画的制作工艺是铜版雕刻，手工上色，外层用阿拉伯胶做保护。Thomas Jefferys（1719–1771）被称为“地理界的乔治三世国王”。他是一名英语地图制作者，是那个时代的主要地图提供者。他为政府和其他官方机构雕刻印刷地图，制作了范围广泛的商业地图和体图册，尤其是北美地区。在作为一个地图制作者的工作中，他对那些他调查和绘制过的地区的服饰习俗产生了兴趣，它们被华丽地展示在了这部总共四卷的收藏中。

在 18 世纪，痴迷于遥远的土地，在旅行中寻找乐趣，相对来说还是一种新现象。这类收藏品曾经很流行，因为它向观光客和空想旅行家介绍了其他国家的居民。Jefferys 画卷里的绘画丰富多彩，生动地讲述了几个世纪前世界各民族的独特个性。如今，衣着风格已经发生了改变，曾经一度如此丰富的国家和地区的多样性却逐渐消失了。现在常常很难分辨不同大陆的居民。也许可以尝试积极地看待这种变化，我们用文化和视觉的多样性换来了更加多样化的个人生活——或者是更加多样有趣的智能与科技生活。

在曾经很难区分不同的计算机书籍的时代，Manning 出版社为了赞扬计算机行业的创造性和进取精神，使用了基于三个世纪前民族服饰的丰富多样性的书籍封面，这些多样性被 Jefferys 的画作复活了。

目录

序 XV

前言 XVII

致谢 XIX

关于本书 XXI

关于作者 XXIV

关于封面插图 XXV

第 1 部分 Kotlin 简介 1

1 Kotlin：定义和目的 3

1.1 Kotlin 初体验 3

1.2 Kotlin 的主要特征 4

1.2.1 目标平台：服务器端、Android 及任何 Java 运行的地方 4

1.2.2 静态类型 5

1.2.3 函数式和面向对象 6

1.2.4 免费并开源 7

1.3 Kotlin 应用 8

1.3.1 服务器端的 Kotlin 8

1.3.2 Android 上的 Kotlin 9

1.4 Kotlin 的设计哲学 10
1.4.1 务实 10
1.4.2 简洁 11
1.4.3 安全 12
1.4.4 互操作性 13
1.5 使用 Kotlin 工具 14
1.5.1 编译 Kotlin 代码 14
1.5.2 IntelliJ IDEA 和 Android Studio 插件 15
1.5.3 交互式 shell 15
1.5.4 Eclipse 插件 15
1.5.5 在线 playground 15
1.5.6 Java 到 Kotlin 的转换器 16
1.6 小结 16

2 Kotlin 基础 17

2.1 基本要素：函数和变量 17
2.1.1 Hello,world! 18
2.1.2 函数 18
2.1.3 变量 20
2.1.4 更简单的字符串格式化：字符串模板 22
2.2 类和属性 23
2.2.1 属性 24
2.2.2 自定义访问器 25
2.2.3 Kotlin 源码布局：目录和包 26
2.3 表示和处理选择：枚举和“when” 28
2.3.1 声明枚举类 28
2.3.2 使用“when”处理枚举类 29
2.3.3 在“when”结构中使用任意对象 30
2.3.4 使用不带参数的“when” 31
2.3.5 智能转换：合并类型检查和转换 32
2.3.6 重构：用“when”代替“if” 34
2.3.7 代码块作为“if”和“when”的分支 35
2.4 迭代事物：“while”循环和“for”循环 36
2.4.1 “while”循环 36

2.4.2 迭代数字：区间和数列 37
2.4.3 迭代 map 38
2.4.4 使用“in”检查集合和区间的成员 39
2.5 Kotlin 中的异常 41
2.5.1 “try”“catch”和“finally” 41
2.5.2 “try”作为表达式 42
2.6 小结 44

3 函数的定义与调用 45

3.1 在 Kotlin 中创建集合 45
3.2 让函数更好调用 47
3.2.1 命名参数 48
3.2.2 默认参数值 49
3.2.3 消除静态工具类：顶层函数和属性 50
3.3 给别人的类添加方法：扩展函数和属性 53
3.3.1 导入和扩展函数 54
3.3.2 从 Java 中调用扩展函数 54
3.3.3 作为扩展函数的工具函数 55
3.3.4 不可重写的扩展函数 56
3.3.5 扩展属性 58
3.4 处理集合：可变参数、中缀调用和库的支持 59
3.4.1 扩展 Java 集合的 API 59
3.4.2 可变参数：让函数支持任意数量的参数 60
3.4.3 键值对的处理：中缀调用和解构声明 60
3.5 字符串和正则表达式的处理 62
3.5.1 分割字符串 62
3.5.2 正则表达式和三重引号的字符串 63
3.5.3 多行三重引号的字符串 64
3.6 让你的代码更整洁：局部函数和扩展 66
3.7 小结 68

4 类、对象和接口 69

4.1 定义类继承结构 70
4.1.1 Kotlin 中的接口 70

4.1.2 open、final 和 abstract 修饰符：默认为 final 72
4.1.3 可见性修饰符：默认为 public 75
4.1.4 内部类和嵌套类：默认是嵌套类 76
4.1.5 密封类：定义受限的类继承结构 79
4.2 声明一个带非默认构造方法或属性的类 80
4.2.1 初始化类：主构造方法和初始化语句块 80
4.2.2 构造方法：用不同的方式来初始化父类 83
4.2.3 实现在接口中声明的属性 85
4.2.4 通过 getter 或 setter 访问支持字段 87
4.2.5 修改访问器的可见性 88
4.3 编译器生成的方法：数据类和类委托 89
4.3.1 通用对象方法 89
4.3.2 数据类：自动生成通用方法的实现 92
4.3.3 类委托：使用“by”关键字 93
4.4 “object”关键字：将声明一个类与创建一个实例结合起来 95
4.4.1 对象声明：创建单例易如反掌 95
4.4.2 伴生对象：工厂方法和静态成员的地盘 98
4.4.3 作为普通对象使用的伴生对象 100
4.4.4 对象表达式：改变写法的匿名内部类 102
4.5 小结 104

5 Lambda 编程 105

5.1 Lambda 表达式和成员引用 105
5.1.1 Lambda 简介：作为函数参数的代码块 106
5.1.2 Lambda 和集合 107
5.1.3 Lambda 表达式的语法 108
5.1.4 在作用域中访问变量 111
5.1.5 成员引用 114
5.2 集合的函数式 API 116
5.2.1 基础：filter 和 map 116
5.2.2 “all”“any”“count”和“find”：对集合应用判断式 118
5.2.3 groupBy：把列表转换成分组的 map 119
5.2.4 flatMap 和 flatten：处理嵌套集合中的元素 120
5.3 惰性集合操作：序列 121

5.3.1 执行序列操作：中间和末端操作 123
5.3.2 创建序列 125
5.4 使用 Java 函数式接口 126
5.4.1 把 lambda 当作参数传递给 Java 方法 127
5.4.2 SAM 构造方法：显式地把 lambda 转换成函数式接口 129
5.5 带接收者的 lambda："with" 与 "apply" 131
5.5.1 "with" 函数 131
5.5.2 "apply" 函数 133
5.6 小结 135

6 Kotlin 的类型系统 137

6.1 可空性 137
6.1.1 可空类型 138
6.1.2 类型的含义 140
6.1.3 安全调用运算符："?." 141
6.1.4 Elvis 运算符："?:" 143
6.1.5 安全转换："as?" 145
6.1.6 非空断言："!!" 146
6.1.7 "let" 函数 148
6.1.8 延迟初始化的属性 149
6.1.9 可空类性的扩展 151
6.1.10 类型参数的可空性 153
6.1.11 可空性和 Java 153
6.2 基本数据类型和其他基本类型 157
6.2.1 基本数据类型：Int、Boolean 及其他 158
6.2.2 可空的基本数据类型：Int?、Boolean? 及其他 159
6.2.3 数字转换 160
6.2.4 "Any" 和 "Any?"：根类型 162
6.2.5 Unit 类型：Kotlin 的 "void" 163
6.2.6 Nothing 类型："这个函数永不返回" 164
6.3 集合与数组 164
6.3.1 可空性和集合 165
6.3.2 只读集合与可变集合 167
6.3.3 Kotlin 集合和 Java 168

6.3.4 作为平台类型的集合 171
6.3.5 对象和基本数据类型的数组 173
6.4 小结 175

第 2 部分 拥抱 Kotlin 177

7 运算符重载及其他约定 179

7.1 重载算术运算符 180
7.1.1 重载二元算术运算 180
7.1.2 重载复合赋值运算符 183
7.1.3 重载一元运算符 184
7.2 重载比较运算符 186
7.2.1 等号运算符："equals" 186
7.2.2 排序运算符：compareTo 187
7.3 集合与区间的约定 188
7.3.1 通过下标来访问元素："get" 和 "set" 188
7.3.2 "in" 的约定 190
7.3.3 rangeTo 的约定 191
7.3.4 在 "for" 循环中使用 "iterator" 的约定 192
7.4 解构声明和组件函数 193
7.4.1 解构声明和循环 194
7.5 重用属性访问的逻辑：委托属性 195
7.5.1 委托属性的基本操作 196
7.5.2 使用委托属性：惰性初始化和 "by lazy()" 197
7.5.3 实现委托属性 198
7.5.4 委托属性的变换规则 202
7.5.5 在 map 中保存属性值 203
7.5.6 框架中的委托属性 204
7.6 小结 205

8 高阶函数：Lambda 作为形参和返回值 207

8.1 声明高阶函数 207
8.1.1 函数类型 208
8.1.2 调用作为参数的函数 209

8.1.3 在 Java 中使用函数类 211
8.1.4 函数类型的参数默认值和 null 值 212
8.1.5 返回函数的函数 214
8.1.6 通过 lambda 去除重复代码 216
8.2 内联函数：消除 lambda 带来的运行时开销 218
8.2.1 内联函数如何运作 219
8.2.2 内联函数的限制 221
8.2.3 内联集合操作 222
8.2.4 决定何时将函数声明成内联 223
8.2.5 使用内联 lambda 管理资源 223
8.3 高阶函数中的控制流 225
8.3.1 lambda 中的返回语句：从一个封闭的函数返回 225
8.3.2 从 lambda 返回：使用标签返回 226
8.3.3 匿名函数：默认使用局部返回 228
8.4 小结 229

9 泛型 231

9.1 泛型类型参数 232
9.1.1 泛型函数和属性 232
9.1.2 声明泛型类 234
9.1.3 类型参数约束 235
9.1.4 让类型形参非空 237
9.2 运行时的泛型：擦除和实化类型参数 238
9.2.1 运行时的泛型：类型检查和转换 238
9.2.2 声明带实化类型参数的函数 241
9.2.3 使用实化类型参数代替类引用 243
9.2.4 实化类型参数的限制 244
9.3 变型：泛型和子类型化 245
9.3.1 为什么存在变型：给函数传递实参 245
9.3.2 类、类型和子类型 246
9.3.3 协变：保留子类型化关系 248
9.3.4 逆变：反转子类型化关系 252
9.3.5 使用点变型：在类型出现的地方指定变型 254
9.3.6 星号投影：使用 * 代替类型参数 257

9.4 小结 261

10 注解与反射 263

10.1 声明并应用注解 264
10.1.1 应用注解 264
10.1.2 注解目标 265
10.1.3 使用注解定制 JSON 序列化 267
10.1.4 声明注解 269
10.1.5 元注解：控制如何处理一个注解 270
10.1.6 使用类做注解参数 271
10.1.7 使用泛型类做注解参数 272
10.2 反射：在运行时对 Kotlin 对象进行自省 273
10.2.1 Kotlin 反射 API：KClass、KCallable、KFunction 和 KProperty 274
10.2.2 用反射实现对象序列化 278
10.2.3 用注解定制序列化 279
10.2.4 JSON 解析和对象反序列化 283
10.2.5 反序列化的最后一步：callBy() 和使用反射创建对象 287
10.3 小结 291

11 DSL 构建 293

11.1 从 API 到 DSL 293
11.1.1 领域特定语言的概念 295
11.1.2 内部 DSL 296
11.1.3 DSL 的结构 297
11.1.4 使用内部 DSL 构建 HTML 298
11.2 构建结构化的 API:DSL 中带接收者的 lambda 299
11.2.1 带接收者的 lambda 和扩展函数类型 299
11.2.2 在 HTML 构建器中使用带接收者的 lambda 303
11.2.3 Kotlin 构建器：促成抽象和重用 307
11.3 使用“invoke”约定构建更灵活的代码块嵌套 310
11.3.1 “invoke”约定：像函数一样可以调用的对象 310
11.3.2 “invoke”约定和函数式类型 311
11.3.3 DSL 中的“invoke”约定：在 Gradle 中声明依赖 312

11.4 实践中的 Kotlin DSL 314
11.4.1 把中缀调用链接起来：测试框架中的“should” 314
11.4.2 在基本数据类型上定义扩展：处理日期 316
11.4.3 成员扩展函数：为 SQL 设计的内部 DSL 317
11.4.4 Anko：动态创建 Android UI 320
11.5 小结 322

A 构建 Kotlin 项目 323

B Kotlin 代码的文档化 327

C Kotlin 生态系统 331

第1部分

Kotlin简介

本书第一部分的目标是让你高效地编写使用现有 API 的 Kotlin 代码。第 1 章将介绍 Kotlin 的主要特征。在第 2 章～第 4 章，你将会学习到最基本的 Java 编程概念（语句、函数、类和类型）是如何映射到 Kotlin 代码中的，以及 Kotlin 是如何丰富这些概念使得编程变得更加愉悦的。你完全可以依靠对现有的 Java 知识的理解，借助 IDE 的编码辅助功能和 Java 到 Kotlin 的转换器来加速掌握这些知识。在第 5 章中，你会了解 lambda 是如何帮助你有效地解决编程中最普遍的一些问题的，比如对集合的操作。最后，在第 6 章，你将会熟悉 Kotlin 的一项重要特性：对 `null` 值处理的支持。

1 Kotlin：定义和目的

本章内容包括

- Kotlin 的基本示范
- Kotlin 语言的主要特征
- Android 和服务器端开发的可能性
- Kotlin 与其他语言的区别
- 用 Kotlin 编写并运行代码

Kotlin 到底是什么？它是一种针对 Java 平台的新编程语言。Kotlin 简洁、安全、务实，并且专注于与 Java 代码的互操作性。它几乎可以用在现在 Java 使用的任何地方：服务器端开发、Android 应用，等等。Kotlin 可以很好地和所有现存的 Java 库和框架一起工作，而且性能水平和 Java 旗鼓相当。在这一章中，我们将详细地探讨 Kotlin 的主要特征。

1.1 Kotlin初体验

让我们从一个小例子开始，来看看 Kotlin 代码长什么样子。这个例子定义了一个 Person 类来表示“人”，创建一个“人”的集合，查找其中年纪最大的人，并打印结果。尽管这是非常小的一段代码，从中也可以看到 Kotlin 许多有趣的特性。

我们对其中的一些特性做了标记，以便你可以方便地在本书后续的内容中找到它们。代码简要地进行了解释，但是如果有些内容你现在还无法理解，请不要担心，稍后我们会详细讨论。

如果你想尝试运行这个例子，最简单的方法是使用 http://try.kotl.in 的在线 Playground。输入示例代码并单击 Run 按钮，代码将会执行。

代码清单 1.1 Kotlin 初体验

```
data class Person(val name: String,
                  val age: Int? = null)

fun main(args: Array<String>) {
    val persons = listOf(Person("Alice"),
                         Person("Bob", age = 29))

    val oldest = persons.maxBy { it.age ?: 0 }
    println("The oldest is: $oldest")
}

// The oldest is: Person(name=Bob, age=29)
```

“数据”类

可空类型（Int?）；实参的默认值

顶层函数

命名参数

lambda 表达式；Elvis 运算符

字符串模板

自动生成的 toString 方法

你声明了一个简单的数据类，它包括了两个属性：name 和 age。age 属性默认为 null（如果没有指定）。在创建“人”的列表时，你省略了 Alice 的年龄，所以这里年龄使用了默认值 null。然后你调用了 maxBy 函数来查找列表中年纪最大的那个“人”。传递给这个函数的 lambda 表达式需要一个参数，使用 it 作为这个参数的默认名称。如果 age 属性为 null，*Elvis 运算符*（?:）会返回零。因为 Alice 的年龄没有指定，Elvis 运算符使用零代替了它，所以 Bob 幸运地成了年纪最大的人。

喜欢这样的代码吗？继续读下去，你将会学习到更多，并成为一名 Kotlin 专家。我们希望不久之后，在你自己的项目中也能看到这样的代码，而不只是在书上。

1.2 Kotlin的主要特征

你大概已经知道了 Kotlin 是一种怎样的语言，让我们更加深入地了解一下它的关键属性。首先，我们来看看你能用 Kotlin 创造哪些种类的应用程序。

1.2.1 目标平台：服务器端、Android 及任何 Java 运行的地方

Kotlin 的首要目标是提供一种更简洁、更高效、更安全的替代 Java 的语言，并且适用于现今使用 Java 的所有环境。Java 是一门非常受欢迎的语言，它广泛地应用于不同的环境：小到智能卡 (JavaCard 技术)，大到 Google、Twitter、LinkedIn 和其

他这种规模的互联网公司运行的最大的数据中心。在这些地方，使用 Kotlin 可以帮助开发者在实现目标的同时减少代码并避免麻烦。

Kotlin 最常见的应用场景有：

- 编写服务器端代码（典型的代表是 Web 应用后端）
- 创建 Android 设备上运行的移动应用

但 Kotlin 还有其他用武之地。例如，可以使用 Intel Multi-OS Engine（(https://software.intel.com/en-us/multi-os-engine）让 Kotlin 代码运行在 iOS 设备上。还可以使用 Kotlin 和 TornadoFX（https://github.com/edvin/tornadofx）以及 JavaFX[1] 一起来构建桌面应用程序。

除了 Java 之外，Kotlin 还可以编译成 JavaScript，允许你在浏览器中运行 Kotlin 代码。但截止本书撰写时，对 JavaScript 的支持仍在 JetBrains 内部探索并进行原型开发，这超出了本书的范围，而其他一些平台也在考虑支持 Kotlin 的未来版本。

正如你所看到的，Kotlin 的目标平台是相当广泛的。Kotlin 并没有被限制在单一的问题域，也没有被限制在解决软件开发者面临的某一类型的挑战。相反，对所有开发过程中涌现的任务，Kotlin 都提供了全面的生产力提升。它借助支持特定领域或编程范式的库，提供了卓越的集成水准。接下来让我们来看看 Kotlin 作为一种编程语言的关键特质。

1.2.2 静态类型

Kotlin 和 Java 一样是一种静态类型的编程语言。这意味着所有表达式的类型在编译期已经确定了，而编译器就能验证对象是否包含了你想访问的方法或者字段。

这与动态类型的编程语言形成了鲜明的对比，后者在 JVM 上的代表包括 Groovy 和 JRuby。这些语言允许你定义可以存储任何数据类型的变量，或者返回任何数据类型的函数，并在运行时才解析方法和字段引用。这会减少代码量并增加创建数据结构的灵活性。但它的缺点是，在编译期不能发现像名字拼写错误这样的问题，继而导致运行时的错误。

另一方面，与 Java 不同的是，Kotlin 不需要你在源代码中显式地声明每个变量的类型。很多情况下，变量类型可以根据上下文来自动判断，这样就可以省略类型声明。这里有一个可能是最简单的例子：

```
val x = 1
```

在声明这个变量时，由于变量初始化为整型值，Kotlin 自动判断出它的类型是

1 JavaFX：Getting Started with JavaFX，Oracle，http://mng.bz/500y。——译者注

Int。编译器这种从上下文推断变量类型的能力被称作类型推导。

下面罗列了一些静态类型带来的好处：

- 性能——方法调用速度更快，因为不需要在运行时才来判断调用的是哪个方法。
- 可靠性——编译器验证了程序的正确性，因而运行时崩溃的概率更低。
- 可维护性——陌生代码更容易维护，因为你可以看到代码中用到的对象的类型。
- 工具支持——静态类型使 IDE 能提供可靠的重构、精确的代码补全以及其他特性。

得益于 Kotlin 对类型推导的支持，你不再需要显式地声明类型，因此大部分关于静态类型的额外冗长代码也就不复存在了。

当你检视 Kotlin 类型系统的细节时，你会发现许多熟悉的概念。类、接口以及泛型和 Java 非常接近，所以大部分的 Java 知识可以很容易地转移到 Kotlin。然而，也会有一些新概念出现。

其中最重要的概念是 Kotlin 对可空类型的支持，通过在编译期检测可能存在的空指针异常，它让你可以写出更可靠的程序。本章后面我们将回顾可空类型，并在第 6 章中详细讨论。

另一个 Kotlin 类型系统的新概念是对函数类型的支持。要搞清楚这一点，我们先要了解函数式编程的主要思想，以及 Kotlin 是如何支持这种编程风格的。

1.2.3 函数式和面向对象

作为一个 Java 开发者，你一定对面向对象编程的核心概念烂熟于胸，但函数式编程对你来说却可能很新鲜。函数式编程的核心概念如下：

- 头等函数——把函数（一小段行为）当作值使用，可以用变量保存它，把它当作参数传递，或者当作其他函数的返回值。
- 不可变性——使用不可变对象，这保证了它们的状态在其创建之后不能再变化。
- 无副作用——使用的是纯函数。此类函数在输入相同时会产生同样的结果，并且不会修改其他对象的状态，也不会和外面的世界交互。

函数式编程风格的代码能给你带来什么好处？首先，简洁。函数式风格的代码比相应的命令式风格的代码更优雅、更简练，因为把函数当作值可以让你获得更强大的抽象能力，从而避免重复代码。

假设你有两段类似的代码，实现相似的任务（例如，在集合中寻找一个匹配的元素）但具体细节略有不同（如何判断元素是匹配的）。可以轻易地将这段逻辑中公共的部分提取到一个函数中，并将其他不同的部分作为参数传递给它。这些参数本

身也是函数，但你可以使用一种简洁的语法来表示这些匿名函数，它被称作 *lambda* 表达式：

```
fun findAlice() = findPerson { it.name == "Alice" }
fun findBob() = findPerson { it.name == "Bob" }
```

findPerson() 包含了寻找一个人的公共逻辑

花括号中的代码块识别出你要找的特定的人

函数式编程风格的代码带来的第二个好处是多线程安全。多线程程序中最大的错误来源之一就是，在没有采用适当同步机制的情况下，在不同的线程上修改同一份数据。如果你使用的是不可变数据结构和纯函数，就能保证这样不安全的修改根本不会发生，也就不需要考虑为其设计复杂的同步方案。

最后，函数式编程意味着测试更加容易。没有副作用的函数可以独立地进行测试，因为不需要写大量的设置代码来构造它们所依赖的整个环境。

一般来说，函数式编程风格可以在任何编程语言中使用（包括 Java），它的很多主张都被认为是良好的编程风格。然而并不是所有的语言都提供了语法和库支持，让我们可以毫不费力地使用这种风格。例如，Java 8 之前的 Java 版本都缺少了这种支持。Kotlin 拥有丰富的特性集从一开始就支持函数式编程风格，包括：

- 函数类型，允许函数接受其他函数作为参数，或者返回其他函数。
- *lambda* 表达式，让你用最少的样板代码方便地传递代码块
- 数据类，提供了创建不可变值对象的简明语法
- 标准库中包括了丰富的 *API* 集合，让你用函数式编程风格操作对象和集合

Kotlin 允许你使用函数式编程风格但并没有强制你使用它。当你需要的时候，可以使用可变数据，也可以编写带副作用的函数，而且不需要跳过任何多余的步骤。然后，毫无疑问的是，在 Kotlin 中使用基于接口和类层次结构的库就像 Java 一样简单。当编写 Kotlin 代码的时候，可以结合使用面向对象编程和函数式编程风格，并使用最合适的工具来对付亟待解决的问题。

1.2.4 免费并开源

Kotlin 语言（包括编译器、库和所有相关工具）是完全开源的，并且可以自由使用。它采用 Apache 2 许可证；其开发过程完全公开在 GitHub (http://github.com/jetbrains/kotlin) 上，并且欢迎来自社区的贡献。如果你要开发 Kotlin 应用程序，有三种开源

IDE 供你选择：IntelliJ IDEA Community[2] 版、Android Studio 以及 Eclipse，它们都完全支持 Kotlin（当然，IntelliJ IDEA Ultimate 也支持 Kotlin。）

现在你明白了 Kotlin 是什么语言，让我们看看 Kotlin 在具体的实际应用中会给你带来哪些好处。

1.3 Kotlin应用

如前所述，Kotlin 使用的两个主要的领域是服务器端和 Android 开发。接下来我们分别看看这两个领域，以及为什么 Kotlin 非常适合它们。

1.3.1 服务器端的 Kotlin

服务器端编程是一个非常大的概念，它包含了所有下列的应用程序类型甚至更多：

- 返回 HTML 页面给浏览器的 Web 应用程序
- 通过 HTTP 暴露 JSON API 的移动应用后端服务
- 通过 RPC 协议互相通信的微服务

多年以来，开发者一直在构建这些类型的应用，并且积累了大量的框架和技术来帮助他们构建这些应用。这些应用通常并不是孤立地开发或者从零开始的，它们几乎总是对现有的系统进行扩展、改进或者替换，新的代码必须和系统中现有部分进行集成，而这些部分可能很多年之前就写成了。

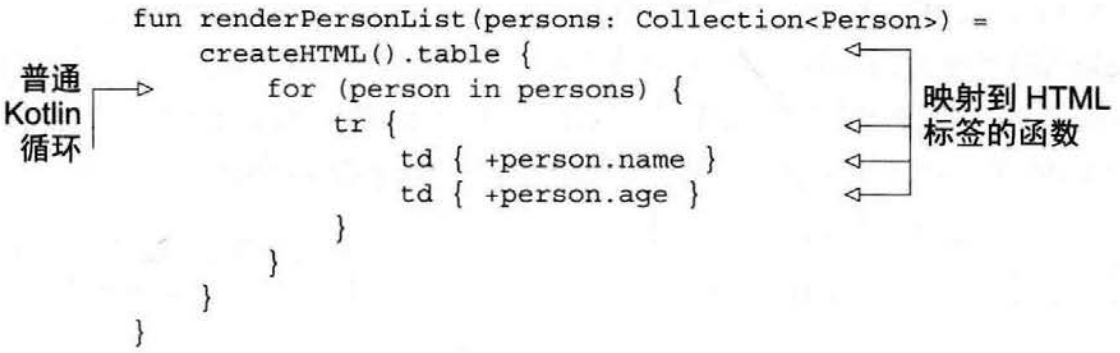
这种环境下 Kotlin 的一大优势就是它与现有的 Java 代码无缝的互操作性。无论是要编写一个全新的组件还是移植一个现有服务的代码，Kotlin 都毫无压力。不管你需要在 Kotlin 中继承 Java 类，还是以某种方式注解一个类的方法或字段，都不会遇到任何问题。它带来的优点是系统的代码会更紧凑、更可靠、更易于维护。

与此同时，Kotlin 还引入了许多用于开发这类系统的新技术。例如，对构建器模式的支持让你可以使用更简洁的语法来创建任何对象图，同时保留了语言中全套的抽象机制和代码重用工具。

这个特性的一个最简单的用例就是 HTML 生成库，它可以把一个外部模板语言替换成简洁且完全类型安全的解决方案。这里有一个例子：

---

2 此处参照JetBrains官方中文主页：http://www.jetbrains.com.cn/idea.html，主页中并没有使用中文“社区”来代替“Community”，因此这里保留了“Community”，下面的“Ultimate”也是同样保留英文原文。——译者注




```
fun renderPersonList(persons: Collection<Person>) =
    createHTML().table {
        for (person in persons) {
            tr {
                td { +person.name }
                td { +person.age }
            }
        }
    }
```

可以轻松地把映射到 HTML 标签的函数和常规的 Kotlin 语言结构组合起来。你不再需要使用一门独立的模板语言，也不需要学习新的语法，仅仅使用循环就可以生成 HTML 页面。

另一个能用上 Kotlin 干净和简洁的 DSL 的用例是持久化框架。例如，Exposed 框架（https://github.com/jetbrains/exposed）就提供了易读的 DSL，可以完全使用 Kotlin 代码来描述 SQL 数据库的结构并执行查询操作，并且有全面的类型检查。下面这个小例子展示了可行的做法：

```
object CountryTable : IdTable() {
    val name = varchar("name", 250).uniqueIndex()
    val iso = varchar("iso", 2).uniqueIndex()
}

class Country(id: EntityID) : Entity(id) {
    var name: String by CountryTable.name
    var iso: String by CountryTable.iso
}

val russia = Country.find {
    CountryTable.iso.eq("ru")
}.first()

println(russia.name)
```

描述数据库中的一张表

创建一个数据库实体对应的类

使用纯 Kotlin 代码查询数据库

本书后面的 7.5 节和第 11 章将会详细地剖析这些技术。

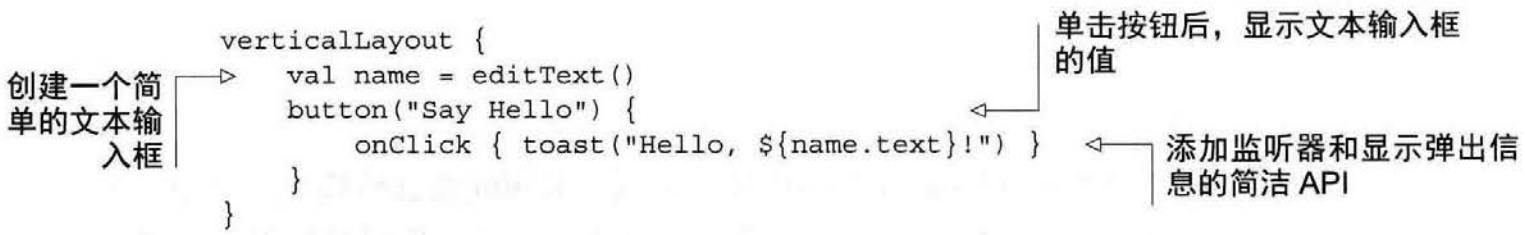
1.3.2 Android 上的 Kotlin

一个典型的移动应用和一个典型的企业应用完全不同。它更小，更少地依赖与现有的代码集成，通常需要快速交付，同时需要保证在大量的设备上能够可靠地运行。这类项目 Kotlin 也能胜任。

Kotlin 的语言特性，加上支持 Android 框架的特殊编译器插件，让 Android 的开发体验变得高效和愉悦。常见的开发任务，比如给控件添加监听器或是把布局元素绑定到字段，可以用更少的代码完成，有时甚至根本不用写任何代码（编译器会

帮你生成）。同样由 Kotlin 团队打造的库 Anko（https://github.com/kotlin/anko）给许多标准 Android API 添加了 Kotlin 友好的适配器，进一步提升了 Android 的开发体验。

下面是 Anko 的一个简单例子，可以品尝到使用 Kotlin 进行 Android 开发的滋味。只要把这段代码放在一个 `Activity` 中，一个简单的 Android 应用就做好了！




```
verticalLayout {
    val name = editText()
    button("Say Hello") {
        onClick { toast("Hello, ${name.text}!") }
    }
}
```

使用 Kotlin 带来的另一优势就是更好的应用可靠性。如果你有开发 Android 应用的经验，你一定对“Unfortunately, Process Has Stopped”对话框深恶痛绝。如果你的应用有未处理的异常，这个对话框就会出现，而这种异常一般是 `NullPointerException`（空指针异常）。Kotlin 的类型系统通过精确地跟踪 `null` 值，大大减轻了空指针异常问题带来的压力。大部分 Java 中会导致 `NullPointerException` 的代码在 Kotlin 中无法编译成功，以确保这些错误在应用到达用户手中之前得到修正。

同时，由于 Kotlin 完全兼容 Java 6，使用它并不会带来任何新的编译问题。你可以享受所有 Kotlin 的酷炫新特性，而你的用户仍然可以在他们的设备上使用你的应用，即使他们的设备并没有运行最新版本的 Android 系统。

说到性能，Kotlin 也没有带来任何负面影响。Kotlin 编译器生成的代码执行起来和普通的 Java 代码效率一样。Kotlin 使用的运行时（库）体积相当小，所以编译出来的应用程序包体积也不会增加多少。当你使用 lambda 的时候，它们会被许多 Kotlin 标准库函数内联。lambda 的内联确保不会创建新对象，因此应用程序也不必忍受额外的 GC 暂停。

看过了和 Java 相比 Kotlin 的优势之后，我们再来看看 Kotlin 的设计哲学——那些把 Kotlin 和其他面向 JVM 的现代语言区分开的主要特性。

1.4 Kotlin的设计哲学

当谈起 Kotlin 的时候，我们喜欢说它是一门务实、简洁和安全的语言，专注于互操作性。这里的每个词语究竟是什么含义？我们逐个来看看。

1.4.1 务实

务实对我们来说意味着一件简单的事情：Kotlin 就是一门设计出来解决现实世

界问题的实用语言。它的设计基于多年创建大型系统的工业经验，它的特性也是为解决许多软件开发者遇到的场景而选择的。此外，来自 JetBrains 内部和社区的开发者已经使用 Kotlin 的早期版本很多年，他们的反馈也被融合进了这门语言公开发布的版本中。所以我们才能自信地说，Kotlin 能够帮助解决实际项目的问题。

Kotlin 也不是一门研究性的语言。我们没有试图提升编程语言设计领域目前的技术水平，也没有尝试探索计算机科学的创新理念。反而，我们会尽可能地依赖已经出现在其他编程语言中并被证明是成功的那些特性和解决方案。这降低了语言的复杂性，也让它更容易学习，因为你可以仰仗那些熟悉的概念。

此外，Kotlin 也没有强制使用某种特定的编程风格和范式。当你开始学习这门语言的时候，可以使用熟悉的来自 Java 经验的风格。然后，你会渐渐地发现更多强大的 Kotlin 特性，并学习把它们应用到你的代码中，让代码更简洁、更符合语言习惯。

Kotlin 的实用主义的另一个重要体现是对于工具的专注。对开发者的生产力而言，一个智能的开发环境和一门设计良好的语言同样重要。因此，事后再来考虑对 IDE 进行支持就是马后炮。而 Kotlin 的情况是，IntelliJ IDEA 的插件是和编译器同步开发的，并且在设计语言特性时始终牢记着对工具的支持。

IDE 支持对帮助你探索 Kotlin 的特性也发挥着重要作用。许多情况下，工具会发现那些可以用更简洁的结构来替换的通用代码模式，并给你提供修正这些代码的选择。通过研究自动修正所使用的语言特性，你就能学习如何在自己的代码中应用这些特性。

1.4.2 简洁

和编写新代码相比，开发人员会耗费更多的时间来阅读现有代码，这已经是常识。想象一下你所在的团队正在开发一个大项目，而你的工作是添加一个新特性或者修改 bug。第一步会干什么？首先要找到需要改变的那段代码，然后才能实现你的修改。要阅读很多代码才能知道你要做什么。这些代码可能最近刚完成，由你的同事或者是那些已经离开的同事编写，或者是你自己很久之前写好的。只有搞懂了周围的代码你才能做出正确的改动。

代码越简单越简洁，你就能越快地了解发生了什么。当然，良好的设计和达意的命名在这里起着重要的作用。但语言的选择及其简洁性也很重要。如果语言的语法清晰地表达了被阅读的代码的意图，没有因为达成意图所需的样板代码而晦涩难懂，那么它就是*简洁*的。

在 Kotlin 中，我们努力地保证你写的代码都具有实际的意义，而不是仅仅为了满足代码结构的需要。许多标准的 Java 样板代码，例如 getter、setter 以及将构造方法的参数赋值给字段的逻辑，在 Kotlin 中都是隐式的，并不会使你的源代码变得

混乱。

另外一个导致代码变得不必要的冗长的原因是编写显式的代码来完成常见的任务，比如定位集合中的元素。和许多其他现代语言一样，Kotlin 有丰富的标准库，让你用库方法调用来代替这些冗长重复的代码段。Kotlin 对 lambda 的支持，让小代码块可以轻松地传递给库函数。这让你可以把公共的那部分代码全部封装在库中，而在用户代码中仅保留特定的针对任务的那部分。

与此同时，Kotlin 并没有尝试把源代码压缩到最小可能的长度。例如，即使 Kotlin 支持运算符重载，用户也不能定义自己的运算符。因此，库开发者不能用神秘的标点符号序列来代替方法名字。单词比标点符号显然更易读，也更容易找到相关的文档。

越简洁的代码写起来花的时间越短，更重要的是，读起来耗费的时间也更短。这会提高你的生产力并让你更快地达成目标。

1.4.3 安全

通常，我们说一门编程语言是安全的，我们的意思是它的设计可以防止程序出现某些类型的错误。当然，这并不意味着绝对的高质量，没有任何语言可以阻止所有可能出现的错误。此外，预防错误是需要成本的。需要给编译器提供程序有关预期操作更多的信息，这样编译器才能验证这些信息是否和程序的功能匹配。因此，你要在得到的安全级别和因为增加更多细节注解而造成的生产力损失之间权衡利弊。

使用 Kotlin，我们试图实现比 Java 更高的安全级别，同时保持更低的总体成本。在 JVM 上运行已经提供了许多的安全保证：例如，内存安全，防止了缓冲区溢出以及其他错误的动态内存分配造成的问题。作为面向 JVM 的静态类型语言，Kotlin 还保证了应用程序的类型安全。这比使用 Java 的成本要更低：不需要指定所有的类型声明，因为许多情况下编译器会自动地推断出类型。

Kotlin 所做的不止这些，这意味着更多的原本在运行时失败的错误在编译期的检查中就被阻止了。最重要的一点是，Kotlin 努力地从你的程序中消除 `NullPointerException`。Kotlin 的类型系统跟踪那些可以或不可以为 `null` 的值，并且禁止那些运行时可能导致 `NullPointerException` 的操作。这所带来的额外的成本是极小的：把类型标记为可空的只要一个字符，就是类型尾部的一个问号：

```
val s: String? = null          <—— 可以为 null
val s2: String = ""      <—— 不能为 null
```

除此之外，Kotlin 提供了许多便利的方法来处理可空数据。这非常有助于消灭应用程序的崩溃。

Kotlin 有助于避免的另一种异常类型就是 `ClassCastException`。当你把一个对象转换成一种类型，而没有事先检查它是否是正确的类型时，就会发生这个异常。在 Java 中，开发者常常省略了这类检查，因为必须反复地在检查和其后的转换中写明类型名称。另一方面，Kotlin 中的检查和转换被组合成了一次操作：一旦检查过类型，不需要额外的转换就能直接引用属于这个类型的成员。这样，开发者就没有借口跳过检查，也不会给错误留下可乘之机。下面展示了它是如何工作的：

```
if (value is String)                    <-- 检查类型
    println(value.toUpperCase())        <-- 调用该类型的方法
```

1.4.4 互操作性

关于互操作性，你的第一个问题可能是："我是不是可以继续使用现有的库？"Kotlin 给出的回答是："当然可以。"无论需要使用哪种库提供的 API，都可以在 Kotlin 中使用它们。可以调用 Java 的方法，继承 Java 的类和实现 Java 的接口，在 Kotlin 类上应用 Java 的注解，等等。

与其他一些 JVM 语言不同，Kotlin 在互操作性上更上一层楼，让 Java 代码也可以毫不费力地调用 Kotlin 的代码。无须取巧：Kotlin 的类和方法可以像常规的 Java 类和方法一样被调用。这带来了无限的灵活性，在项目的任何地方都可以混合使用 Java 和 Kotlin。当你刚开始在自己的 Java 项目中引入 Kotlin 时，可以在代码库中的任意一个类上运行 Java 到 Kotlin 的转换器，剩下的代码不需要任何修改就可以继续编译和工作。不管你所转换的类是什么角色，这都是可行的。

另一个 Kotlin 专注于互操作性的领域是在最大程度上使用现有的 Java 库。例如，Kotlin 没有自己的集合库，它完全依赖 Java 标准库中的类，使用额外的函数来扩展它们，让它们在 Kotlin 中用起来更方便（我们会在 3.3 节中了解这种机制更多的细节）。这意味着在 Kotlin 中调用 Java API 时，永远不需要包装或者转换这些 Java 对象，反之亦然。所有这些 Kotlin 提供的丰富的 API 在运行时没有任何的额外开销。

Kotlin 工具也对跨语言项目提供了全面支持。它可以编译任意混合的 Java 和 Kotlin 源码，不管它们之间是怎样互相依赖的。IDE 的特性也能跨语言工作，允许：

- 自由地在 Java 和 Kotlin 源码文件之间切换
- 调试混合语言的项目，可以在不同语言编写的代码之中单步调试
- 重构 Java 方法的时候，Kotlin 代码中的对它们的调用也会得到正确的更新，

反之亦然

希望我们已经说服你尝试一下 Kotlin。现在，你要如何开始使用它？在接下来的一节中，我们将从命令行和其他不同工具的使用两方面讨论编译和运行 Kotlin 代码的过程。

1.5 使用Kotlin工具

和 Java 一样，Kotlin 也是编译型语言。这意味着你必须先编译，然后才能执行 Kotlin 代码。让我们来讨论一下编译过程，然后看看帮你完成这个过程需要的不同工具。如果你需要关于如何搭建开发环境的信息，请参考 Kotlin 官方网站的“Tutorials”（教程）一节 (https://kotlinlang.org/docs/tutorials)。

1.5.1 编译 Kotlin 代码

Kotlin 的源代码存放在后缀名为 .kt 的文件中。Kotlin 编译器会分析源代码并生成 .class 文件，这和 Java 编译器做的没什么不同。然后按照你正在处理的应用程序类型的标准过程打包和执行生成的 .class 文件。最简单的情况下，只需要使用 `kotlinc` 命令就可以从命令行编译代码，然后就可以用 `java` 命令执行你的代码：

```
kotlinc <source file or directory> -include-runtime -d <jar name>
java -jar <jar name>
```

图 1.1 展示了 Kotlin 构建过程的简单描述。

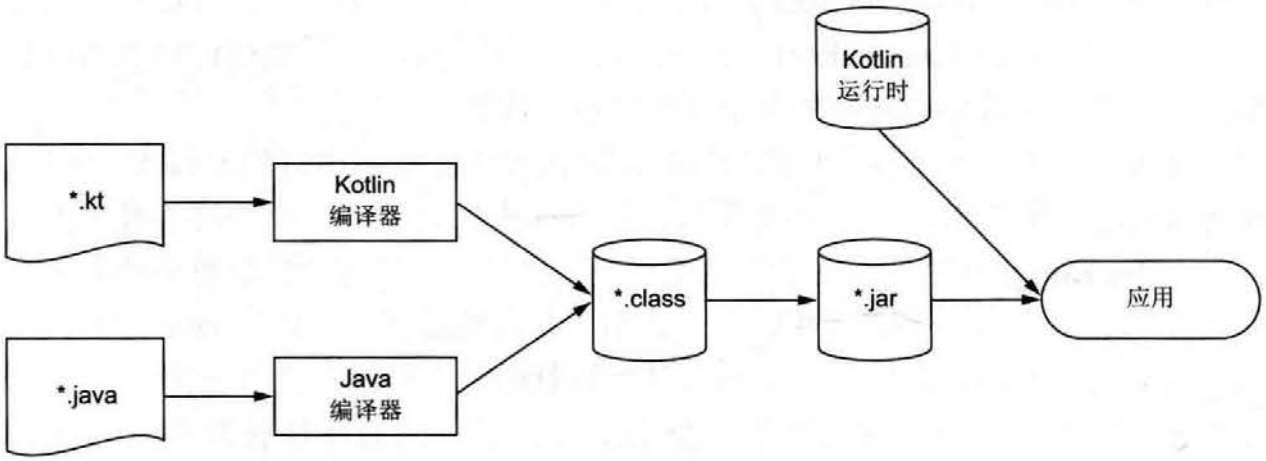


图 1.1 Kotlin 构建过程

用 Kotlin 编译器编译的代码依赖 *Kotlin 运行时库*。它包括了 Kotlin 自己的标准库类的定义，以及 Kotlin 对标准 Java API 的扩展。运行时库需要和你的应用程序一起分发。

在大多数实际工作的例子中，你会使用像 Maven、Gradle 或者 Ant 这样的构建系统来编译你的代码。Kotlin 和所有这些构建系统都兼容，我们会在附录 A 中讨论相关细节。所有这些构建系统也支持在同一个代码库中既有 Kotlin 也有 Java 的混合语言项目。此外，Maven 和 Gradle 还会帮你把 Kotlin 运行时库作为依赖加入到你的应用程序中。

1.5.2 IntelliJ IDEA 和 Android Studio 插件

IntelliJ IDEA 的 Kotlin 插件是和语言同步开发的，它是 Kotlin 可用的功能最全面的开发环境。它成熟且稳定，提供了 Kotlin 开发所需的全套工具。

IntelliJ IDEA 15 及其后续版本不需要额外的设置，Kotlin 插件就可以开箱即用。可以选择免费开源的 IntelliJ IDEA Community 版，也可以选择 IntelliJ IDEA Ultimate。在“New Project”（新建项目）对话框中选择“Kotlin”，然后就可以开始工作了。

如果你用的是 Android Studio，可以从“plug-in manager”（插件管理器）中安装 Kotlin 插件。打开“Settings”（设置）对话框，选择“Plugins”（插件），单击“Install JetBrains Plugin”（安装 JetBrains 插件）按钮，然后从列表中选择“Kotlin”。

1.5.3 交互式 shell

如果你想快速地尝试小段的 Kotlin 代码，可以使用交互式 shell(也叫 *REPL*[3])。在 REPL 中，可以逐行地输入 Kotlin 代码并立即看到其执行结果，可以使用不带任何参数的 `kotlinc` 命令启动 REPL，也可以从 IntelliJ IDEA 的“Kotlin”菜单中选择启动 REPL。

1.5.4 Eclipse 插件

如果你是 Eclipse 用户，同样可以选择在你的 IDE 中使用 Kotlin。Kotlin 的 Eclipse 插件提供了必要的 IDE 功能，如导航和代码补全。该插件可以在 Eclipse Marketplace 中找到。要安装它，请选择“Help > Eclipse Marketplace”菜单项，然后在列表中搜索“Kotlin”。

1.5.5 在线 playground

尝试 Kotlin 的最简单的方式，是不需要任何安装和配置。可以在 http://try.kotl.in 找到在线的 playground，可以在上面编写、编译及运行 Kotlin 的小程序。Playground

3 交互式解释器，http://en.wikipedia.org/wiki/Read-eval-print_loop。——译者注

上还展示了 Kotlin 特性的代码示例，其中包括了本书中的所有例子，还有一系列交互式学习 Kotlin 的练习。

1.5.6 Java 到 Kotlin 的转换器

要熟练掌握一门新语言总是要费点力气的。幸运的是，我们开辟了一条很棒的小捷径，让你可以借助现有的 Java 知识来加快学习和运用 Kotlin 的速度。这个工具就是 Java 到 Kotlin 的自动转换器。

当你开始学习 Kotlin 的时候，如果你还没有记住准确的语法，转换器能帮你表达一些内容。可以先用 Java 写出相应的代码片段，然后把它粘贴到 Kotlin 文件中，转换器会自动地将代码转换成 Kotlin。转换的结果不一定总是符合语言习惯，但是它一定是可以工作的代码，这样就可以让你的任务更进一步了。

在现有的 Java 项目中引入 Kotlin 时，转换器也很好用。当你写一个新类时，可以从一开始就用 Kotlin。但是如果你要在一个现有的类上做重大的更改时，可能也想在这个过程中使用 Kotlin，这时转换器就派上用场了。首先把这个类转换成 Kotlin，然后就可以利用现代编程语言的所有优势来添加更改了。

在 IntelliJ IDEA 中使用转换器再简单不过了。要么复制一段 Java 代码粘贴到 Kotlin 文件中，要么触发“Convert Java File to Kotlin File”（转换 Java 文件到 Kotlin 文件）动作来转换整个文件。也可以在 Eclipse 中或者线上使用转换器。

1.6 小结

- Kotlin 是静态类型语言并支持类型推导，允许维护正确性与性能的同时保持源代码的简洁。
- Kotlin 支持面向对象和函数式两种编程风格，通过头等函数使更高级别的抽象成为可能，通过支持不可变值简化了测试和多线程开发。
- 在服务器端应用程序中它工作得很好，全面支持所有现存的 Java 框架，为常见的任务提供了新工具，如生成 HTML 和持久化。
- 在 Android 上它也可以工作，这得益于紧凑的运行时、对 Android API 特殊的编译器支持以及丰富的库，为常见 Android 开发任务提供了 Kotlin 友好的函数。
- 它是免费和开源的，全面支持主流的 IDE 和构建系统。
- Kotlin 是务实的、安全的、简洁的，与 Java 可互操作，意味着它专注于使用已经证明过的解决方案处理常见任务，防止常见的像 `NullPointerException` 这样的错误，支持紧凑和易读的代码，以及提供与 Java 无限制的集成。

2 Kotlin 基础

本章内容包括

- 声明函数、变量、类、枚举以及属性
- Kotlin 中的控制结构
- 智能转换
- 抛出和处理异常

在这一章我们将学习怎样用 Kotlin 声明任何程序都存在的基本要素：变量、函数和类，顺便熟悉 Kotlin 的属性概念。

我们会学习怎样在 Kotlin 中使用不同的控制结构。这些结构大部分都和那些你熟知的 Java 结构相似，但在一些重要的方面增强了。

我们会介绍智能转换的概念，它把类型检查和类型转换合并成了一次操作。最后，我们会谈到异常处理。在学习完本章的内容后，你就能掌握这门语言的基础知识，并利用这些知识编写出可以工作的 Kotlin 代码，哪怕有些写法还不是最符合语言习惯的。

2.1 基本要素：函数和变量

这一节会向你介绍组成每个 Kotlin 程序的基本要素：函数和变量。你将看到

Kotlin 如何让你省略许多类型的声明，以及它如何鼓励你使用不可变的数据而不是可变的数据。

2.1.1 Hello,world!

让我们从最经典的例子开始：一个打印“Hello, world!”的程序。在 Kotlin 中，这只需要一个函数就可以实现：

代码清单 2.1 Kotlin 的“Hello, world!”

```
fun main(args: Array<String>) {
    println("Hello, world!")
}
```

你能从这样简单的一小段代码中观察到哪些特性和语法？看看下面这个列表：

- 关键字 fun 用来声明一个函数。没错，Kotlin 编程有很多乐趣（fun）！
- 参数的类型写在它的名称后面。稍后你会看到，变量的声明也是这样。
- 函数可以定义在文件的最外层，不需要把它放在类中。
- 数组就是类。和 Java 不同，Kotlin 没有声明数组类型的特殊语法。
- 使用 println 代替了 System.out.println。Kotlin 标准库给 Java 标准库函数提供了许多语法更简洁的包装，而 println 就是其中一个。
- 和许多其他现代语言一样，可以省略每行代码结尾的分号。

目前为止感觉还不错！我们稍后会更详细地讨论其中一些话题。现在，我们先来研究一下函数声明语法。

2.1.2 函数

你已经看到了怎样声明一个没有返回任何东西的函数。但是如果函数有一个有意义的结果，返回类型应该放在哪里呢？你可能会猜到它应该位于参数列表之后的某处：

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

>>> println(max(1, 2))
2
```

函数的声明以关键字 fun 开始，函数名称紧随其后：这个例子中函数名称是 max，接下来是括号括起来的参数列表。参数列表的后面跟着返回类型，它们之间

用一个冒号隔开。

图 2.1 向你展示了一个函数的基本结构。注意在 Kotlin 中，if 是有结果值的表达式。它和 Java 中的三元运算符相似：(a>b) ?a: b。

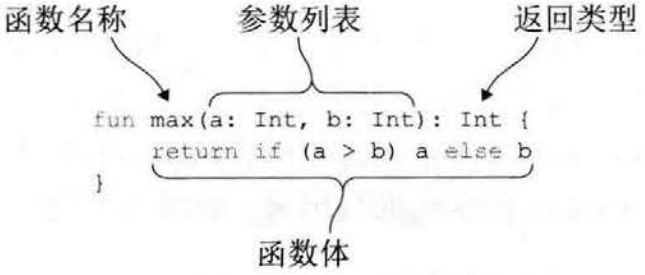


图 2.1 Kotlin 函数声明

语句和表达式

在 Kotlin 中，if 是表达式，而不是语句。语句和表达式的区别在于，表达式有值，并且能作为另一个表达式的一部分使用；而语句总是包围着它的代码块中的顶层元素，并且没有自己的值。在 Java 中，所有的控制结构都是语句。而在 Kotlin 中，除了循环（for、do 和 do/while）以外大多数控制结构都是表达式。这种结合控制结构和其他表达式的能力让你可以简明扼要地表示许多常见的模式，稍后你会在本书中看到这些内容。

另一方面，Java 中的赋值操作是表达式，在 Kotlin 中反而变成了语句。这有助于避免比较和赋值之间的混淆，而这种混淆是常见的错误来源。

表达式函数体

可以让前面的函数变得更简单。因为它的函数体是由单个表达式构成的，可以用这个表达式作为完整的函数体，并去掉花括号和 return 语句：

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

如果函数体写在花括号中，我们说这个函数有代码块体。如果它直接返回了一个表达式，它就有表达式体。

> INTELLIJ IDEA 小贴士 IntelliJ IDEA 提供了在两种函数风格之间转换的 intention actions（意向动作）："Convert to expression body"（转换成表达式函数体）和 "Convert to block body"（转换成代码块函数体）。

在 Kotlin 代码中会常常看到表达式体的函数。这种风格不光用在一些简单的单行函数中，也会用在对更复杂的单个表达式求值的函数中，比如 if、when 以及

try。你会在本章后面介绍 when 结构的内容中看到这样的函数。

还可以进一步简化 max 函数，省掉返回类型：

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

为什么有些函数可以不声明返回类型？作为一门静态类型语言，Kotlin 不是要求每个表达式都应该在编译期具有类型吗？事实上，每个变量和表达式都有类型，每个函数都有返回类型。但是对表达式体函数来说，编译器会分析作为函数体的表达式，并把它的类型作为函数的返回类型，即使没有显式地写出来。这种分析通常被称作类型推导。

注意，只有表达式体函数的返回类型可以省略。对于有返回值的代码块体函数，必须显式地写出返回类型和 return 语句。这是刻意的选择。真实项目中的函数一般很长且可以包含多条 return 语句，显式地写出返回类型和 return 语句能帮助你快速地理解函数能返回的是什么。接下来我们看看声明变量的语法。

2.1.3 变量

在 Java 中声明变量的时候会以类型开始。在 Kotlin 中这样是行不通的，因为许多变量声明的类型都可以省略。所以在 Kotlin 中以关键字开始，然后是变量名称，最后可以加上类型（不加也可以）：

```
val question =
    "The Ultimate Question of Life, the Universe, and Everything"

val answer = 42
```

这个例子省略了类型声明，但是如果需要也可以显式地指定变量的类型：

```
val answer: Int = 42
```

和表达式体函数一样，如果你不指定变量的类型，编译器会分析初始化器表达式的值，并把它的类型作为变量的类型。在前面这个例子中，变量的初始化器 42 的类型的是 Int，那么变量就是这个类型。

如果你使用浮点数常量，那么变量就是 Double 类型：

```
val yearsToCompute = 7.5e6          <—— 7.5 * 10^6 = 7500000.0
```

第 6.2 节会更深入地介绍算术类型。

如果变量没有初始化器，需要显式地指定它的类型：

```
val answer: Int
answer = 42
```

如果不能提供可以赋给这个变量的值的信息，编译器就无法推断出它的类型。

可变变量和不可变量

声明变量的关键字有两个：

- val（来自 *value*）——不可变引用。使用 val 声明的变量不能在初始化之后再次赋值。它对应的是 Java 的 final 变量。
- var（来自 *variable*）——可变引用。这种变量的值可以被改变。这种声明对应的是普通（非 final）的 Java 变量。

默认情况下，应该尽可能地使用 val 关键字来声明所有的 Kotlin 变量，仅在必要的时候换成 var。使用不可变引用、不可变对象及无副作用的函数让你的代码更接近函数式编程风格。第 1 章中简要地介绍了这种风格的优点，在第 5 章中会再次回到这个话题。

在定义了 val 变量的代码块执行期间，val 变量只能进行唯一一次初始化。但是，如果编译器能确保只有唯一一条初始化语句会被执行，可以根据条件使用不同的值来初始化它：

```
val message: String
if (canPerformOperation()) {
    message = "Success"
    // ... 进行操作
}
else {
    message = "Failed"
}
```

注意，尽管 val 引用自身是不可变的，但是它指向的对象可能是可变的。例如，下面这段代码是完全有效的：

```
val languages = arrayListOf("Java")
languages.add("Kotlin")
```

声明不可变引用

改变引用指向的对象

第 6 章中，我们会更详细地讨论可变对象和不可变对象。

即使 var 关键字允许变量改变自己的值，但它的类型却是改变不了的。例如，下面这段代码是不会编译的：

```
var answer = 42
answer = "no answer"
```

错误：类型不匹配

使用字符串字面值会发生错误，因为它的类型（String）不是期望的类型（Int）。编译器只会根据初始化器来推断变量的类型，在决定类型的时候不会考虑

后续的赋值操作。

如果需要在变量中存储不匹配类型的值，必须手动把值转换或强制转换到正确的类型。我们会在 6.2.3 节中讨论基本数据类型之间的转换。

现在你学会了怎样定义变量，是时候看看更多引用变量值的新技巧了。

2.1.4 更简单的字符串格式化：字符串模板

让我们回到本章开始的“Hello World”的例子。下面展示了如何完成这个经典练习的下一步：用 Kotlin 的方式来和人打招呼：

代码清单 2.2 使用字符串模板

```
fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Kotlin"
    println("Hello, $name!")
}
```

打印的是“Hello, Kotlin”，如果你把“Bob”作为实参传进来，打印的就是“Hello, Bob”

这个例子介绍了一个新特性，叫作*字符串模板*。在代码中，你声明了一个变量 name，并在后面的字符串字面值中使用了它。和许多脚本语言一样，Kotlin 让你可以在字符串字面值中引用局部变量，只需要在变量名称前面加上字符 $。这等价于 Java 中的字符串连接("Hello, " + name + "!")，效率一样但是更紧凑[1]。当然，表达式会进行静态检查，如果你试着引用一个不存在的变量，代码根本不会编译。

如果要在字符串中使用 $ 字符，你要对它转义：println("\$x") 会打印 $x，并不会把 x 解释成变量的引用。

还可以引用更复杂的表达式，而不是仅限于简单的变量名称，只需要把表达式用花括号括起来：

```
fun main(args: Array<String>) {
    if (args.size > 0) {
        println("Hello, ${args[0]}!")
    }
}
```

使用 ${} 的语法插入 args 数组中的第一个元素

还可以在双引号中直接嵌套双引号，只要它们处在某个表达式的范围内（即花括号内）：

```
fun main(args: Array<String>) {
    println("Hello, ${if (args.size > 0) args[0] else "someone"}!")
}
```

1 编译后的代码创建了一个StringBuilder对象，并把常量部分和变量附加上去。——译者注

稍后，在 3.5 节中，我们会再次回到字符串的话题，更多地讨论你能用它们做些什么。

现在你知道了如何定义函数和变量，让我们更上一层楼，来看看类。这一次，你会借助 Java 到 Kotlin 的转换器来开始运用新的语言特性。

2.2 类和属性

面向对象编程对你来说可能不是什么新鲜的概念，你也许非常熟悉类的抽象机制。Kotlin 这方面的概念你也会觉得似曾相识，但是你会发现许多常见的任务使用更少的代码就可以完成。这一节会向你介绍声明类的基本语法，在第 4 章中我们再深入细节。

首先，来看一个简单的 JavaBean 类 `Person`，目前它只有一个属性，`name`。

代码清单 2.3 简单的 Java 类 Person

```
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

在 Java 中，构造方法的方法体常常包含完全重复的代码：它把参数赋值给有着相同名称的字段。在 Kotlin 中，这种逻辑不用这么多的样板代码就可以表达。

在 1.5.6 节中，我们介绍了 Java 到 Kotlin 的转换器：一个把 Java 代码替换成功能相同的 Kotlin 代码的工具。接下来我们看看这个转换器的实战作用，用它把 `Person` 类转换成 Kotlin。

代码清单 2.4 转换成 Kotlin 的 Person 类

```
class Person(val name: String)
```

看起来不错，不是吗？如果你试过其他一些现代 JVM 语言，你也许见过类似的事情。这种类（只有数据没有其他代码）通常被叫作值对象，许多语言都提供简明语法来声明它们。

注意从 Java 到 Kotlin 的转换过程中 public 修饰符消失了。在 Kotlin 中，public 是默认的可见性，所以你能省略它。

2.2.1 属性

你肯定知道，类的概念就是把数据和处理数据的代码封装成一个单一的实体。在 Java 中，数据存储在字段中，通常还是私有的。如果想让类的使用者访问到数据，得提供*访问器方法*：一个 getter，可能还有一个 setter。在 Person 类中你已经看到了访问器的例子。setter 还可以包含额外的逻辑，包括验证传给它的值、发送关于变化的通知等。

在 Java 中，字段和其访问器的组合常常被叫作*属性*，而许多框架严重依赖这个概念。在 Kotlin 中，属性是头等的语言特性，完全代替了字段和访问器方法。在类中声明一个属性和声明一个变量一样：使用 val 和 var 关键字。声明成 val 的属性是只读的，而 var 属性是可变的。

代码清单 2.5 在类中声明可变属性

```
class Person(
    val name: String,
    var isMarried: Boolean
)
```

只读属性：生成一个字段和一个简单的 getter

可写属性：一个字段、一个 getter 和一个 setter

基本上，当你声明属性的时候，你就声明了对应的访问器（只读属性有一个 getter，而可写属性既有 getter 也有 setter）。访问器的默认实现非常简单：创建一个存储值的字段，以及返回值的 getter 和更新值的 setter。但是如果有需要，可以声明自定义的访问器，使用不同的逻辑来计算和更新属性的值。

代码清单 2.5 中简洁的 Person 类声明隐藏了和原始 Java 代码相同的底层实现：它是一个字段都是私有的类，这些字段在构造方法中初始化并能通过对应的 getter 访问。这意味着在 Java 或者 Kotlin 中都能以同样的方式使用这个类，无论它是在哪里声明的，用法看起来是完全一样的。下面的代码清单展示了你可以怎样在 Java 代码中使用 Person。

代码清单 2.6 在 Java 中使用 Person 类

```
/* Java */
>>> Person person = new Person("Bob", true);
>>> System.out.println(person.getName());
Bob
>>> System.out.println(person.isMarried());
true
```

注意，不管 Person 是定义在 Java 还是 Kotlin 中，这段代码看起来是一样的。Kotlin 的属性 name 把一个名称为 getName 的 getter 方法暴露给 Java。getter 和 setter 的命名规则有一个例外：如果属性的名称以 is 打头，getter 不会增加任何的前缀；而它的 setter 名称中的 is 会被替换成 set。所以在 Java 中，你调用的将是 isMarried()。

如果把代码清单 2.6 中的代码转换成 Kotlin 代码，会得到下面的结果。

代码清单 2.7 在 Kotlin 中使用 Person 类

```
>>> val person = Person("Bob", true)
>>> println(person.name)
Bob
>>> println(person.isMarried)
true
```

调用构造方法不需要关键字“new”

可以直接访问属性，但调用的是 getter

现在，可以直接引用属性，不再需要调用 getter。逻辑没有变化，但代码更简洁了。可变属性的 setter 也是这样：在 Java 中，使用 person.setMarried(false) 来表示离婚，而在 Kotlin 中，可以这样写 person.isMarried = false。

> **小贴士** 对于那些在 Java 中定义的类，一样可以使用 Kotlin 的属性语法。Java 类中的 getter 可以被当成 val 属性在 Kotlin 中访问，而一对 getter/setter 可以被当成 var 属性访问。例如，如果一个 Java 类定义了两个名称为 getName 和 setName 的方法，就把它们当作名称为 name 的属性访问。如果类定义了 isMarried 和 setMarried 方法，对应的 Kotlin 属性的就是 isMarried。

大多数情况下，属性有一个对应的支持字段来保存属性的值。但是如果这个值可以即时计算——例如，根据其他属性计算——可以用自定义的 getter 来表示。

2.2.2 自定义访问器

这一节将向你展示怎样写一个属性访问器的自定义实现。假设你声明这样一个矩形，它能判断自己是否是正方形。不需要一个单独的字段来存储这个信息（是否是正方形），因为可以随时通过检查矩形的长宽是否相等来判断：

```
class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
        get() {
            return height == width
        }
}
```

声明属性的 getter

属性 isSquare 不需要字段来保存它的值。它只有一个自定义实现的 getter。它的值是每次访问属性的时候计算出来的。

注意，不需要使用带花括号的完整语法，也可以这样写 get() = height == width。对这个属性的调用依然不变：

```
>>> val rectangle = Rectangle(41, 43)
>>> println(rectangle.isSquare)
false
```

如果要在 Java 中访问这个属性，可以像前面提到的那样调用 isSquare() 方法。

你可能会问，声明一个没有参数的函数是否比声明带自定义 getter 的属性更好。两种方式几乎一样：实现和性能都没有差别，唯一的差异是可读性。通常来说，如果描述的是类的特征（属性），应该把它声明成属性。

在第 4 章中，我们会看到更多使用类和属性的例子，还会看到显式地声明构造方法的语法。如果你已经迫不及待，还可以使用 Java 到 Kotlin 的转换器。在继续讨论其他的语言特性之前，我们先来简要地探索一下 Kotlin 的代码在磁盘中是怎样组织的。

2.2.3 Kotlin 源码布局：目录和包

你知道 Java 把所有的类组织成包。Kotlin 也有和 Java 相似的包的概念。每一个 Kotlin 文件都能以一条 package 语句开头，而文件中定义的所有声明（类、函数及属性）都会被放到这个包中。如果其他文件中定义的声明也有相同的包，这个文件可以直接使用它们；如果包不相同，则需要导入它们。和 Java 一样，导入语句放在文件的最前面并使用关键字 import。下面这个源码文件的例子展示了包声明和导入语句的语法。

代码清单 2.8 把类和函数的声明放在包中

包声明

```
package geometry.shapes

import java.util.Random

class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
        get() = height == width
}

fun createRandomRectangle(): Rectangle {
    val random = Random()
    return Rectangle(random.nextInt(), random.nextInt())
}
```

导入标准 Java 库的类

Kotlin 不区分导入的是类还是函数，而且，它允许使用 import 关键字导入任何种类的声明。可以直接导入顶层函数的名称。

代码清单 2.9 导入其他包中的函数

```
package geometry.example

import geometry.shapes.createRandomRectangle

fun main(args: Array<String>) {
    println(createRandomRectangle().isSquare)
}
```

导入函数名称

极少数会打印“true”

也可以在包名称后加上 .* 来导入特定包中定义的所有声明。注意这种星号导入不仅让包中定义的类可见，也会让顶层函数和属性可见。在代码清单 2.9 的例子中，用 import geometry.shapes.* 的写法代替显式的导入也能让代码成功编译。

在 Java 中，要把类放到和包结构相匹配的文件与目录结构中。例如，如果你有一个包含若干类的名为 shapes 的包，必须把每一个类都放在一个有着和类相同名字的单独文件中，然后把这些文件放在一个名字为 shapes 的目录中。图 2.2 展示了 geometry 包以及它的子包在 Java 中是怎样组织的。假设 createRandomRectangle 函数位于另外一个单独的类 RectangleUtil。

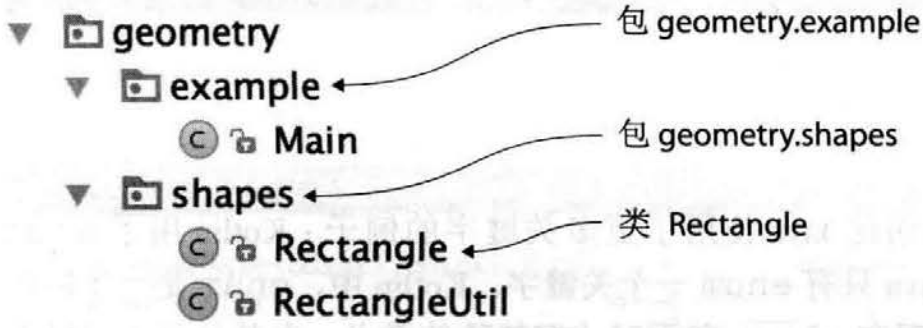


图 2.2 在 Java 中，目录层级结构照搬了包层级结构

在 Kotlin 中，可以把多个类放在同一个文件中，文件的名字还可以随意选择。Kotlin 也没有对磁盘上源文件的布局强加任何限制。比如，可以把包 geometry.shapes 所有的内容都放在文件 shapes.kt 中，并把这个文件直接放在目录 geometry 中，而不需要再创建一个独立的 shapes 文件夹（如图 2.3 所示）。

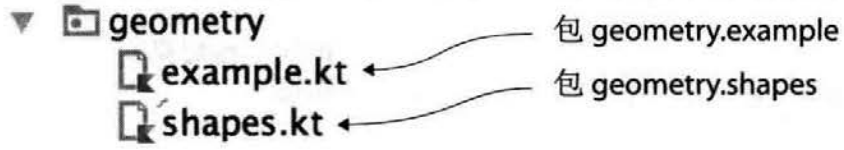


图 2.3 包层级结构不需要遵循目录层级结构

不管怎样，大多数情况下，遵循 Java 的目录布局并根据包结构把源码文件放到目录中，依然是个不错的实践。在 Kotlin 和 Java 混用的项目中坚持这样的结构尤为重要，因为这样做可以让你逐步地迁移代码，而不会和一些错误不期而遇。但是你应该毫不犹豫地把多个类放进同一个文件中，特别是那些很小的类（在 Kotlin 中，类通常很小）。

现在了解了程序的结构是怎样的，让我们继续学习基本概念，来看看 Kotlin 的控制结构。

2.3 表示和处理选择：枚举和“when”

这一节中，我们将会讨论 when 结构。它可以被认为是 Java 中 switch 结构的替代品，但是它更强大，也使用得更频繁。顺便我们会给你展示一个 Kotlin 声明枚举的例子并讨论*智能转换*的概念。

2.3.1 声明枚举类

让我们先给这本正儿八经的书增加一些充满想象力的明快的图片，来看看色彩的枚举。

代码清单 2.10 声明一个简单的枚举类

```
enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

这是极少数 Kotlin 声明比 Java 使用了更多关键字的例子：Kotlin 用了 enum class 两个关键字，而 Java 只有 enum 一个关键字。Kotlin 中，enum 是一个所谓的*软关键字*：只有当它出现在 class 前面时才有特殊的意义，在其他地方可以把它当作普通的名称使用。与此不同的是，class 仍然是一个关键字，要继续使用名称 clazz 和 aClass 来声明变量。

和 Java 一样，枚举并不是值的列表：可以给枚举类声明属性和方法。下面的代码清单展示了这种方式。

代码清单 2.11 声明一个带属性的枚举类

```
enum class Color(
        val r: Int, val g: Int, val b: Int
) {
```

声明枚举常量的属性

```
    RED(255, 0, 0), ORANGE(255, 165, 0),
    YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),
    INDIGO(75, 0, 130), VIOLET(238, 130, 238);

    fun rgb() = (r * 256 + g) * 256 + b
}
>>> println(Color.BLUE.rgb())
255
```

在每个常量创建的时候指定属性值

这里必须要有分号

给枚举类定义一个方法

枚举常量用的声明构造方法和属性的语法与之前你看到的常规类一样。当你声明每个枚举常量的时候，必须提供该常量的属性值。注意这个例子向你展示了 Kotlin 语法中唯一必须使用分号的地方：如果要在枚举类中定义任何方法，就要使用分号把枚举常量列表和方法定义分开。现在我们来看看一些在代码中处理枚举常量的超酷的方式。

2.3.2 使用“when”处理枚举类

你还记得小孩子怎样通过口诀来记住彩虹的颜色吗？这里就有一个：“Richard Of York Gave Battle In Vain!”[2]。假设你需要一个函数来告诉你每种颜色在这个口诀中对应的单词（而你并不想把这些信息存储在枚举内部）。在 Java 中可以用 `switch` 语句完成，而 Kotlin 对应的结构是 `when`。

和 `if` 相似，`when` 是一个有返回值的表达式，因此可以写一个直接返回 `when` 表示式的表达式体函数。在本章开始介绍函数的时候，我们曾约定要给出一个多行的表达式体函数的例子，现在就看看这个例子。

代码清单 2.12 使用 when 来选择正确的枚举值

```
fun getMnemonic(color: Color) =
    when (color) {
        Color.RED -> "Richard"
        Color.ORANGE -> "Of"
        Color.YELLOW -> "York"
        Color.GREEN -> "Gave"
        Color.BLUE -> "Battle"
        Color.INDIGO -> "In"
        Color.VIOLET -> "Vain"
    }

>>> println(getMnemonic(Color.BLUE))
Battle
```

直接返回一个“when”表达式

如果颜色和枚举常量相等就返回对应的字符串

2 口诀中单词首字母和彩虹颜色的英文单词首字母一致（Red、Orange、Yellow、Green、Blue、Indigo、Violet），口诀本身也是一个典故：韦克菲尔德战役，http://en.wikipedia.org/wiki/Battle_of_Wakefield。——译者注

上面的代码根据传进来的 color 值找到对应的分支。和 Java 不一样，你不需要在每个分支都写上 break 语句（在 Java 中遗漏 break 通常会导致 bug）。如果匹配成功，只有对应的分支会执行。也可以把多个值合并到同一个分支，只需要用逗号隔开这些值。

代码清单 2.13 在一个 when 分支上合并多个选项

```
fun getWarmth(color: Color) = when(color) {
    Color.RED, Color.ORANGE, Color.YELLOW -> "warm"
    Color.GREEN -> "neutral"
    Color.BLUE, Color.INDIGO, Color.VIOLET -> "cold"
}

>>> println(getWarmth(Color.ORANGE))
warm
```

这些例子用的都是枚举常量的完整名称，即指定了枚举类的名称 Color。可以通过导入这些常量值来简化代码。

代码清单 2.14 导入枚举常量后不用限定词就可以访问

```
import ch02.colors.Color
import ch02.colors.Color.*

fun getWarmth(color: Color) = when(color) {
    RED, ORANGE, YELLOW -> "warm"
    GREEN -> "neutral"
    BLUE, INDIGO, VIOLET -> "cold"
}
```

导入其他包中定义的 Color 类

显式地导入枚举常量就可以使用它们的名称

使用导入的常量的名称

2.3.3 在“when”结构中使用任意对象

Kotlin 中的 when 结构比 Java 中的 switch 强大得多。switch 要求必须使用常量（枚举常量、字符串或者数字字面值）作为分支条件，和它不一样，when 允许使用任何对象。我们写这样一个函数来混合两种颜色，如果它们在我们这个小小的调色板中是能够混合的。你只有很少的选项，可以简单地把所有组合列举出来。

代码清单 2.15 在 when 分支中使用不同的对象

```
fun mix(c1: Color, c2: Color) =
        when (setOf(c1, c2)) {
            setOf(RED, YELLOW) -> ORANGE
            setOf(YELLOW, BLUE) -> GREEN
            setOf(BLUE, VIOLET) -> INDIGO
```

“when”表达式的实参可以是任何对象，它被检查是否与分支条件相等

列举出能够混合的颜色对

```
        else -> throw Exception("Dirty color")
    }
```

如果没有任何其他分支匹配这里就会执行

如果颜色 c1 和 c2 分别是 RED 和 YELLOW（反过来也行），它们混合后的结果就是 ORANGE，以此类推。你使用了 set 比较来实现这个调色板。Kotlin 标准函数库中有一个 setOf 函数可以创建出一个 Set，它会包含所有指定为函数实参的对象。*set* 这种集合的条目顺序并不重要，只要两个 set 中包含一样的条目，它们就是相等的。所以如果 setOf(c1,c2) 和 setOf(RED,YELLOW) 是相等的，意味着 c1 是 RED 和 c2 是 YELLOW 的时候相等，反过来也成立。这正是你想要检查的条件。

when 表达式把它的实参依次和所有分支匹配，直到某个分支满足条件。这里 setOf(c1,c2) 被用来检查是否和分支条件相等：先和 setOf(RED,YELLOW) 比较，然后是其他颜色的 set，一个接一个。如果没有其他的分支满足条件，else 分支会执行。

能使用任何表达式做 when 的分支条件，很多情况下会让你写的代码既简洁又漂亮。这个例子中，分支条件是等式检查，接下来你会看到条件还可以是任意的布尔表达式。

2.3.4 使用不带参数的“when”

你可能已经注意到代码清单 2.15 的效率多少有些低。每次调用这个函数的时候，它都会创建一些 Set 实例，仅仅用来检查两种给定的颜色是否和另外两种颜色匹配。一般这不是什么大问题，但是如果这个函数调用很频繁，它就非常值得用另一种方式重写，来避免创建额外的垃圾对象。代码可读性会变差，但这是为了达到更好性能而必须付出的代价。

代码清单 2.16 不带参数的 when

```
fun mixOptimized(c1: Color, c2: Color) =
    when {
        (c1 == RED && c2 == YELLOW) ||
        (c1 == YELLOW && c2 == RED) ->
            ORANGE

        (c1 == YELLOW && c2 == BLUE) ||
        (c1 == BLUE && c2 == YELLOW) ->
            GREEN

        (c1 == BLUE && c2 == VIOLET) ||
        (c1 == VIOLET && c2 == BLUE) ->
            INDIGO

        else -> throw Exception("Dirty color")
```

没有实参传给“when”

```
    }
>>> println(mixOptimized(BLUE, YELLOW))
GREEN
```

如果没有给 when 表达式提供参数，分支条件就是任意的布尔表达式。mixOptimized 函数和前面的 mix 函数做的事情一样。这种写法的优点是不会创建额外的对象，但代价是它更难理解。

让我们继续，看看智能转换在 when 结构中发挥作用的例子。

2.3.5 智能转换：合并类型检查和转换

你会写一个函数来作为这一小节的例子，这个函数是对像 (1+2)+4 这样简单的算术表达式求值。这个表达式只包含一种运算：对两个数字求和。其他的算术运算（减法、乘法、除法）都可以用相似的方式实现，可以把这些作为练习。

首先，你会用怎样的形式编码这种表达式？把它们存储在一个树状结构中，结构中每个节点要么是一次求和（Sum）要么是一个数字（Num）。Num 永远都是叶子节点，而 Sum 节点有两个子节点：它们是求和运算的两个参数。下面的代码清单展示了一种简单的类结构来表示这种表达式编码方式：一个叫作 Expr 的接口和它的两个实现类 Num 和 Sum。注意 Expr 接口没有声明任何方法，它只是一个标记接口，用来给不同种类的表达式提供一个公共的类型。声明类的时候，使用一个冒号（:）后面跟上接口名称，来标记这个类实现了这个接口。

代码清单 2.17 表达式类层次结构

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr
```

简单的值对象类，只有一个属性 value，实现了 Expr 接口

Sum 运算的实参可以是任何 Expr: Num 或者另一个 Sum

Sum 存储了 Expr 类型的实参 left 和 right 的引用；在这个小例子中，它们要么是 Num 要么是 Sum。为了存储前面提到的表达式 (1+2)+4，你会创建这样一个对象 Sum(Sum(Num(1),Num(2)),Num(4))。图 2.4 展示了它的树状表示法。

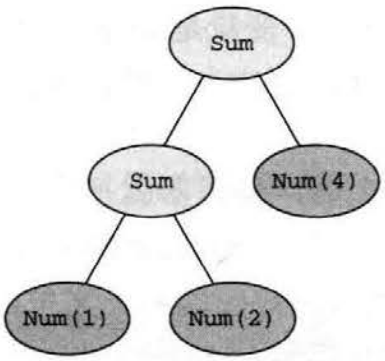


图 2.4 表达式 Sum(Sum(Num(1), Num(2)), Num(4)) 的表示法

现在我们来看看怎样计算表达式的值。例子中表达式的运算结果应该是 7:

```
>>> println (eval(Sum(Sum(Num(1), Num(2)), Num (4))))
7
```

Expr 接口有两种实现，所以为了计算出表达式的结果值，得尝试两种选项：

- 如果表达式是一个数字，直接返回它的值
- 如果是一次求和，得先计算左右两个表达式的值，再返回它们的和。

首先我们看看这个函数用普通的 Java 方式怎样写，然后我们把它重构成 Kotlin 风格的写法。在 Java 中，很可能会用一连串 if 语句来检查这些选项，所以我们先用 Kotlin 按照这种方式实现。

代码清单 2.18 使用 if 层叠对表达式求值

```
fun eval(e: Expr): Int {
    if (e is Num) {
        val n = e as Num
        return n.value
    }
    if (e is Sum) {
        return eval(e.right) + eval(e.left)
    }
    throw IllegalArgumentException("Unknown expression")
}
>>> println(eval(Sum(Sum(Num(1), Num(2)), Num(4))))
7
```

显式地转换成类型 Num 是多余的

变量 e 被智能地转换了类型

在 Kotlin 中，你要使用 is 检查来判断一个变量是否是某种类型。如果你曾经使用 C# 写过代码，这种表示法应该不会陌生。is 检查和 Java 中的 instanceOf 相似。但是在 Java 中，如果你已经检查过一个变量是某种类型并且要把它当作这种类型来访问其成员时，在 instanceOf 检查之后还需要显式地加上类型转换。如果最初的变量会使用超过一次，常常选择把类型转换的结果存储在另一个单独的变

量里。在 Kotlin 中，编译器帮你完成了这些工作。如果你检查过一个变量是某种类型，后面就不再需要转换它，可以就把它当作你检查过的类型使用。事实上编译器为你执行了类型转换，我们把这种行为称为*智能转换*。

在 eval 函数中，在你检查过变量 e 是否为 Num 类型之后，编译器就把它当成 Num 类型的变量解释。于是你不需要显式转换就可以像这样访问 Num 的属性 value：e.value。Sum 的属性 left 和 right 也是这样：在对应的上下文中，只需要写 e.right 和 e.left。在 IDE 中，这种智能转换过的值会用不同的背景颜色着重表示，这样更容易发现这个值的类型是事先检查过的，如图 2.5 所示。

```
if (e is Sum) {
    return eval(e.right) + eval(e.left)
}
```

图 2.5 IDE 使用不同背景色高亮显示智能转换

智能转换只在变量经过 is 检查且之后不再发生变化的情况下有效。当你对一个类的属性进行智能转换的时候，就像这个例子中一样，这个属性必须是一个 val 属性，而且不能有自定义的访问器。否则，每次对属性的访问是否都能返回同样的值将无从验证。

使用 as 关键字来表示到特定类型的显式转换：

```
val n = e as Num
```

接下来看看怎样把 eval 函数重构成更符合 Kotlin 语言习惯的风格。

2.3.6 重构：用“when”代替“if”

Kotlin 和 Java 中的 if 有什么不同，你已经看到了。本章开始的时候，你见过 if 表达式用在适用 Java 三元运算符的上下文中：if(a>b)a else b 和 a>b ? a :b 效果一样。Kotlin 没有三元运算符，因为 if 表达式有返回值，这一点和 Java 不同。这意味着你可以用表达式体语法重写 eval 函数，去掉 return 语句和花括号，使用 if 表达式作为函数体。

代码清单 2.19 使用有返回值的 if 表达式

```
fun eval(e: Expr): Int =
    if (e is Num) {
        e.value
    } else if (e is Sum) {
        eval(e.right) + eval(e.left)
    } else {
        throw IllegalArgumentException("Unknown expression")
    }
```

```
>>> println(eval(Sum(Num(1), Num(2))))
3
```

如果 if 分支中只有一个表达式，花括号是可以省略的。如果 if 分支是一个代码块，代码块中的最后一个表达式会被作为结果返回。

让我们进一步打磨代码，使用 when 来重写它。

代码清单 2.20 使用 when 代替 if 层叠

```
fun eval(e: Expr): Int =
    when (e) {
        is Num ->
            e.value
        is Sum ->
            eval(e.right) + eval(e.left)
        else ->
            throw IllegalArgumentException("Unknown expression")
    }
```

检查实参类型的“when”分支

这里应用了智能转换

when 表达式并不仅限于检查值是否相等，那是之前你看到的。而这里使用了另外一种 when 分支的形式，允许你检查 when 实参值的类型。和代码清单 2.19 中 if 的例子一样，类型检查应用了一次智能转换，所以不需要额外的转换就可以访问 Num 和 Sum 的成员。

比较最后两个 Kotlin 版本的 eval 函数，想一想你应该怎样在自己的代码中也使用 when 代替连串的 if 表达式。当分支逻辑太过复杂时，可以使用代码块作为分支体。我们来看看这种用法。

2.3.7 代码块作为“if”和“when”的分支

if 和 when 都可以使用代码块作为分支体。这种情况下，代码块中的最后一个表达式就是结果。如果要在例子函数中加入日志，可以在代码块中实现它并像之前一样返回最后的值。

代码清单 2.21 使用分支中含有混合操作的 when

```
fun evalWithLogging(e: Expr): Int =
    when (e) {
        is Num -> {
            println("num: ${e.value}")
            e.value
        }
        is Sum -> {
            val left = evalWithLogging(e.left)
```

代码块中最后的表达式，如果 e 的类型是 Num 就会返回它

```
        val right = evalWithLogging(e.right)
        println("sum: $left + $right")
        left + right                          ← 如果 e 的类型是 Sum 就会返回这个表达式
    }
    else -> throw IllegalArgumentException("Unknown expression")
}
```

现在可以看到 evalWithLogging 函数打印出来的日志，并跟踪计算的顺序：

```
>>> println(evalWithLogging(Sum(Sum(Num(1), Num(2)), Num(4))))
num: 1
num: 2
sum: 1 + 2
num: 4
sum: 3 + 4
7
```

规则——“代码块中最后的表达式就是结果”，在所有使用代码块并期望得到一个结果的地方成立。你会在本章末尾看到，同样的规则对 try 主体和 catch 子句也有效，而第 5 章还会讨论该规则在 lambda 表达式中的应用。但是在 2.2 节我们提到，这个规则对常规函数不成立。一个函数要么具有不是代码块的表达式函数体，要么具有包含显式 return 语句的代码块函数体。

现在熟悉了 Kotlin 从众多选项中做出正确选择的方式，是时候看看怎样迭代事物了。

2.4 迭代事物：“while”循环和“for”循环

在本章讨论的所有特性中，Kotlin 的迭代应该是和 Java 最接近的。when 循环和 Java 完全一样，本节开头会一笔带过。for 循环仅以唯一一种形式存在，和 Java 的 for-each 循环一致。其写法 for <item> in <elements> 和 C# 一样。和 Java 一样，循环最常见的应用就是迭代集合。我们也会探索它是怎样覆盖其他使用循环的场景的。

2.4.1 “while”循环

Kotlin 有 while 循环和 do-while 循环，它们的语法和 Java 中相应的循环没有什么区别：

```
while (condition) {                    ← 当 condition 为 true 时执行循环体
    /*...*/
}

do {                                   ← 循环体第一次会无条件地执行。此后，当 condition 为 true 时才执行
    /*...*/
} while (condition)
```

Kotlin 并没有给这些简单的循环带来任何新东西，所以不必停留。我们继续讨论 for 循环的各种用法。

2.4.2 迭代数字：区间和数列

正如我们刚刚提到的那样，在 Kotlin 中没有常规的 Java for 循环。在这种循环中，先初始化变量，在循环的每一步更新它的值，并在值满足某个限制条件时退出循环。为了替代这种最常见的循环用法，Kotlin 使用了区间的概念。

区间本质上就是两个值之间的间隔，这两个值通常是数字：一个起始值，一个结束值。使用 .. 运算符来表示区间：

```
val oneToTen = 1..10
```

注意 Kotlin 的区间是包含的或者闭合的，意味着第二个值始终是区间的一部分。

你能用整数区间做的最基本的事情就是循环迭代其中所有的值。如果你能迭代区间中所有的值，这样的区间被称作*数列*。

让我们用整数迭代来玩 Fizz-Buzz 游戏。这是一种用来打发长途驾驶旅程的不错方式，还能帮你回忆起被遗忘的除法技巧。游戏玩家轮流递增计数，遇到能被 3 整除的数字就用单词 *fizz* 代替，遇到能被 5 整除的数字则用单词 *buzz* 代替。如果一个数字是 3 和 5 的公倍数，你得说"FizzBuzz"。

下面的代码清单打印出了游戏中 1 到 100 之间所有数字的正确答案。注意你是怎样用不带参数的 when 表达式来检查可能的条件的。

代码清单 2.22 使用 when 实现 Fizz-Buzz 游戏

```
fun fizzBuzz(i: Int) = when {
    i % 15 == 0 -> "FizzBuzz "
    i % 3 == 0 -> "Fizz "
    i % 5 == 0 -> "Buzz "
    else -> "$i "
}
>>> for (i in 1..100) {
...     print(fizzBuzz(i))
... }
}
1 2 Fizz 4 Buzz Fizz 7 ...
```

如果 i 能被 3 整除，返回 Fizz

否则返回数字自己

如果 i 能被 5 整除，返回 Buzz

在整数区间 1..100 迭代

如果 i 能被 15 整除，返回 FizzBuzz。和 Java 一样 % 是取模运算符

假设一个小时的驾驶之后，你已经厌倦了这些规则，想把游戏变得复杂一点，那我们可以从 100 开始倒着计数并且只计偶数。

代码清单 2.23 迭代带步长的区间

```
>>> for (i in 100 downTo 1 step 2) {
...     print(fizzBuzz(i))
... }
Buzz 98 Fizz 94 92 FizzBuzz 88 ...
```

现在你在迭代一个带步长的数列，它允许跳过一些数字。步长也可以是负数，这种情况下数列是递减而不是递增的。在这个例子中，`100 downTo 1` 是递减的数列（步长为 -1）。然后 `step` 把步长的绝对值变成了 2，但方向保持不变（事实上，步长被设置成了 -2）。

如前所述，`..` 语法始终创建的是包含结束值（`..` 右边的值）的区间。许多情况下，迭代不包含指定结束值的半闭合区间更方便。使用 `until` 函数可以创建这样的区间。例如，循环 `for (x in 0 until size)` 虽然等同于 `for (x in 0..size - 1)`，但是更清晰地表达了意图。稍后，在 3.4.3 节，你会学习更多关于这些例子中 `downTo`、`step` 和 `until` 的语法。

可以看到使用区间和数列是怎样帮助你应付 FizzBuzz 游戏的进阶规则的。现在让我们看看其他使用 `for` 循环的例子。

2.4.3 迭代 map

我们提到了使用 `for...in` 循环的最常见的场景是迭代集合。这和 Java 中的用法一样，所以我们不会讲太多关于它的内容。让我们来看看你可以怎样迭代 map。

作为例子，我们看看这个打印字符二进制表示的小程序。你会把这些二进制表示保存在一个 map 中（仅做说明之用）。下面的代码创建了一个 map，把某些字母的二进制表示填充进去，最后打印出 map 的内容。

代码清单 2.24 初始化并迭代 map

```
val binaryReps = TreeMap<Char, String>()

for (c in 'A'..'F') {
    val binary = Integer.toBinaryString(c.toInt())
    binaryReps[c] = binary
}

for ((letter, binary) in binaryReps) {
    println("$letter = $binary")
}
```

使用 TreeMap 让键排序

把 ASCII 码转换成二进制

使用字符区间迭代从 A 到 F 之间的字符

根据键 c 把值存储到 map 中

迭代 map，把键和值赋给两个变量

`..` 语法不仅可以创建数字区间，还可以创建字符区间。这里使用它迭代从 *A* 开始到 *F* 的所有字符，包括 *F*。

代码清单 2.24 展示了 for 循环允许展开迭代中的集合的元素（在这个例子中，展开的是 map 的键值对集合）。把展开的结果存储到了两个独立的变量中：letter 是键，binary 是值。稍后，在 7.4.1 节，你将学到更多的展开语法。

代码清单 2.24 还用到了另外一个实用的小技巧，根据键来访问和更新 map 的简明语法。可以使用 map[key] 读取值，并使用 map[key] = value 设置它们，而不需要调用 get 和 put。下面这段代码

```
binaryReps[c] = binary
```

等价于它的 Java 版本：

```
binaryReps.put(c, binary)
```

例子中的输出就像这样（我们把一列结果排版成了两列）：

```
A = 1000001     D = 1000100
B = 1000010     E = 1000101
C = 1000011     F = 1000110
```

可以用这样的展开语法在迭代集合的同时跟踪当前项的下标。不需要创建一个单独的变量来存储下标并手动增加它：

```
val list = arrayListOf("10", "11", "1001")
for ((index, element) in list.withIndex()) {
    println("$index: $element")
}
```

迭代集合时使用下标

这段代码打印出了你期待的内容：

```
0: 10
1: 11
2: 1001
```

下一章我们会深入探索关于 withIndex 的内容。

我们已经看过了如何使用关键字 in 来迭代区间或者集合，还可以用 in 来检查区间或者集合是否包含了某个值。

2.4.4 使用"in"检查集合和区间的成员

使用 in 运算符来检查一个值是否在区间中，或者它的逆运算，!n，来检查这个值是否不在区间中。下面展示了如何使用 in 来检查一个字符是否属于一个字符区间。

代码清单 2.25 使用 in 检查区间的成员

```
fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun isNotDigit(c: Char) = c !in '0'..'9'

>>> println(isLetter('q'))
true
>>> println(isNotDigit('x'))
true
```

这种检查字符是否是英文字母的技巧看起来很简单。在底层，没有什么特殊处理：你依然会检查字符的编码是否位于第一个字母编码和最后一个字母编码之间的某个位置。但是这个逻辑被简洁地隐藏到了标准库中的区间类实现中：

```
c in 'a'..'z'
```

变换成 a <= c && c <= z

in 运算符和 !in 也适用于 when 表达式。

代码清单 2.26 用 in 检查作为 when 分支

```
fun recognize(c: Char) = when (c) {
    in '0'..'9' -> "It's a digit!"
    in 'a'..'z', in 'A'..'Z' -> "It's a letter!"
    else -> "I don't know…"
}
>>> println(recognize('8'))
It's a digit!
```

检查值是否在 0 到 9 的区间之内

可以组合多种区间

区间也不仅限于字符。假如有一个支持实例比较操作的任意类（实现了 java.lang.Comparable 接口），就能创建这种类型的对象的区间。如果是这样的区间，并不能列举出这个区间中的所有对象。想想这种情况：例如，是否可以列举出“Java”和“Kotlin”之间所有的字符串？答案是不能。但是仍然可以使用 in 运算符检查一个其他的对象是否属于这个区间：

```
>>> println("Kotlin" in "Java".."Scala")
true
```

结果和“Java” <= “Kotlin” && “Kotlin” <= “Scala” 一样

注意，这里字符串是按照字母表顺序进行比较的，因为 String 就是这样实现 Comparable 接口的。

in 检查也同样适用于集合：

```
>>> println("Kotlin" in setOf("Java", "Scala"))
false
```

这个集合不包含字符串“Kotlin”

稍后在 7.3.2 节，你将看到怎样使用你自己的数据类型的区间和数列，和通常情况下可以对哪些对象使用 in 检查。

本章我们还想介绍另外一组 Java 语句：处理异常的语句。

2.5 Kotlin中的异常

Kotlin 的异常处理和 Java 以及其他许多语言的处理方式相似。一个函数可以正常结束，也可以在出现错误的情况下抛出异常。方法的调用者能捕获到这个异常并处理它；如果没有被处理，异常会沿着调用栈再次抛出。

Kotlin 中异常处理语句的基本形式和 Java 类似，抛出异常的方式也不例外：

```
if (percentage !in 0..100) {
    throw IllegalArgumentException(
        "A percentage value must be between 0 and 100: $percentage")
}
```

和所有其他的类一样，不必使用 new 关键字来创建异常实例。

和 Java 不同的是，Kotlin 中 throw 结构是一个表达式，能作为另一个表达式的一部分使用：

```
val percentage =
    if (number in 0..100)
        number
    else
        throw IllegalArgumentException(
            "A percentage value must be between 0 and 100: $number")
```

“throw”是一个表达式

在这个例子中，如果条件满足，程序的行为是正确的，而 percentage 变量会用 number 初始化。否则，异常将会被抛出，而变量也不会初始化。在 6.2.6 节中我们会讨论 throw 作为其他表达式的一部分的技术细节。

2.5.1 “try”“catch”和“finally”

和 Java 一样，使用带有 catch 和 finally 子句的 try 结构来处理异常。你会在下面这个代码清单中看到这个结构，这个例子从给定的文件中读取一行，尝试把它解析成一个数字，返回这个数字；或者当这一行不是一个有效数字时返回 null。

代码清单 2.27 像在 Java 中一样使用 try

```
fun readNumber(reader: BufferedReader): Int? {
    try {
        val line = reader.readLine()
        return Integer.parseInt(line)
```

不必显式地指定这个函数可能抛出的异常

```
    }
    catch (e: NumberFormatException) {
        return null
    }
    finally {
        reader.close()
    }
}
```

异常类型在右边

“finally”的作用和Java中的一样

```
>>> val reader = BufferedReader(StringReader("239"))
>>> println(readNumber(reader))
239
```

和Java最大的区别就是throws子句没有出现在代码中：如果用Java来写这个函数，你会显式地在函数声明后写上throws IOException。你需要这样做的原因是IOException是一个受检异常。在Java中，这种异常必须显式地处理。必须声明你的函数能抛出的所有受检异常。如果调用另外一个函数，需要处理这个函数的受检异常，或者声明你的函数也能抛出这些异常。

和其他许多现代JVM语言一样，Kotlin并不区分受检异常和未受检异常。不用指定函数抛出的异常，而且可以处理也可以不处理异常。这种设计是基于Java中使用异常的实践做出的决定。经验显示这些Java规则常常导致许多毫无意义的重新抛出或者忽略异常的代码，而且这些规则不能总是保护你免受可能发生的错误。

例如，在代码清单2.27中，NumberFormatException就不是受检异常。因此，Java编译器并不会强迫你捕获它，在运行时很容易看到这个异常发生。这很令人沮丧，因为无效的输入数据是常见的情况，应该能被优雅地处理。与此同时，BufferedReader.close可能抛出需要处理的受检异常IOException。如果流关闭失败，大多数程序都不会采取什么有意义的行动，所以捕获来自close方法的异常所需的代码就是冗余的样板代码。

Java7的try-with-resources又是什么情况？Kotlin并没有对应的特殊语法：它被实现为一个库函数。在8.2.5节中，你会看到它是如何实现的。

2.5.2 “try”作为表达式

为了了解Java和Kotlin之间另外一个显著的差异，我们修改一下这个例子。让我们去掉finally部分（因为你已经看过它是怎样工作的），并添加一些代码，用来打印从文件中读取的数字。

代码清单2.28 把try当作表达式使用

```
fun readNumber(reader: BufferedReader) {
    val number = try {
```

```
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        return
    }

    println(number)
}

>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
```

变成“try”表达式的值

没有任何输出

Kotlin 中的 try 关键字就像 if 和 when 一样，引入了一个表达式，可以把它的值赋给一个变量。不同于 if，你总是需要用花括号把语句主体括起来。和其他语句一样，如果其主体包含多个表达式，那么整个 try 表达式的值就是最后一个表达式的值。

这个例子将 return 语句放在 catch 代码块中，因此该函数的执行在 catch 代码块之后不会继续。如果你想继续执行，catch 子句也需要有一个值，它将是子句中最后一个表达式的值。下面展示了这是怎么回事。

代码清单 2.29 在 catch 中返回值

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        null
    }

    println(number)
}

>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
null
```

没有任何异常发生时使用这个值

发生异常的情况下使用 null

抛出了一个异常，所以函数打印了“null”

如果一个 try 代码块执行一切正常，代码块中最后一个表达式就是结果。如果捕获到了一个异常，相应 catch 代码块中最后一个表达式就是结果。在代码清单 2.29 中，如果捕获到了 NumberFormatException，结果值就是 null。

现在，如果你已经按捺不住，完全可以开始用 Kotlin 编写程序了，只不过用的还是和 Java 代码类似的方式。继续阅读本书，你会不断学习怎样改变你习惯的思考方式，发挥出新语言的全部能量。

2.6 小结

- fun 关键字用来声明函数。val 关键字和 var 关键字分别用来声明只读变量和可变变量。
- 字符串模板帮助你避免烦琐的字符串连接。在变量名称前加上 $ 前缀或者用 ${ } 包围一个表达式，来把值注入到字符串中。
- 值对象类在 Kotlin 中以简洁的方式表示。
- 熟悉的 if 现在是带返回值的表达式。
- when 表达式类似于 Java 中的 switch 但功能更强大。
- 在检查过变量具有某种类型之后不必显式地转换它的类型：编译器使用智能转换自动帮你完成。
- for、while 和 do-while 循环与 Java 类似，但是 for 循环现在更加方便，特别是当你需要迭代 map 的时候，又或是迭代集合需要下标的时候。
- 简洁的语法 1..5 会创建一个区间。区间和数列允许 Kotlin 在 for 循环中使用统一的语法和同一套抽象机制，并且还可以使用 in 运算符和 !in 运算符来检查值是否属于某个区间。
- Kotlin 中的异常处理和 Java 非常相似，除了 Kotlin 不要求你声明函数可以抛出的异常。

3 函数的定义与调用

本章内容包括

- 用于处理集合、字符串和正则表达式的函数
- 使用命名参数、默认参数，以及中缀调用的语法
- 通过扩展函数和属性来适配 Java 库
- 使用顶层函数、局部函数和属性架构代码

至此，就像使用 Java 一样，你应该可以自如地使用 Kotlin 了。可以看到，从 Java 到 Kotlin，它们的很多概念都是相似的，而往往 Kotlin 可以让它们更加简洁并易读。

在这一章中，你将看到Kotlin改进每个程序的一个重要环节：函数的声明和调用。我们还将研究，如何通过扩展函数将 Java 库转换为 Kotlin 风格，以在混合语言的项目中获得 Kotlin 的全部好处。

为了让讨论更有用和具体，我们将把 Kotlin 集合、字符串和正则表达式作为重点问题领域。作为例子，我们来看看如何在 Kotlin 中创建集合。

3.1 在Kotlin中创建集合

在开始学习对集合的各种有趣的操作之前，需要先学会怎样创建它们。在 2.3.3

节，你已经接触到了怎样使用 setOf 函数创建一个 set。当时你创建了一组颜色，现在，让我们保持它简单的同时，也支持数字。

```
val set = hashSetOf(1, 7, 53)
```

可以用类似的方法创建一个 list 或者 map：

```
val list = arrayListOf(1, 7, 53)
val map = hashMapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

注意，to 并不是一个特殊的结构，而是一个普通函数。在本章的后面会探讨它。

你能猜到这里创建的对象类型吗？亲自运行下面的代码来看一下：

```
>>> println(set.javaClass)
class java.util.HashSet

>>> println(list.javaClass)
class java.util.ArrayList

>>> println(map.javaClass)
class java.util.HashMap
```

Kotlin 的 javaClass 等价于 Java 的 getClass()

如你所见，Kotlin 没有采用它自己的集合类，而是采用的标准的 Java 集合类，这对 Java 开发者是一个好消息。你现在所掌握的所有 Java 集合的知识在这里依然适用。

为什么 Kotlin 没有自己专门的集合类呢？那是因为使用标准的 Java 集合类，Kotlin 可以更容易与 Java 代码交互。当从 Kotlin 中调用 Java 函数的时候，不用转换它的集合类来匹配 Java 的类，反之亦然。

尽管 Kotlin 的集合类和 Java 的集合类完全一致，但 Kotlin 还不止于此。举个例子，可以通过以下方式来获取一个列表中的最后一个元素，或者是得到一个数字列表的最大值：

```
>>> val strings = listOf("first", "second", "fourteenth")

>>> println(strings.last())
fourteenth

>>> val numbers = setOf(1, 14, 2)

>>> println(numbers.max())
14
```

本章将会仔细探究它的工作原理，以及 Java 类上新增的方法从何而来。

在后续的章节中，当我们开始讨论 lambda 时，你将见识到更多的对于集合的操作，但是目前，就继续保持采用 Java 标准的集合类。在 6.3 节，你将学习到集合类

在 Kotlin 类型系统中的表示。

在开始讨论如何操作 Java 集合的 last 和 max 这两个神奇的函数之前，让我们来学习一下函数声明。

3.2 让函数更好调用

现在你已经知道了如何创建一个集合，让我们再来做点别的：打印它的内容。是不是看起来太简单了，别着急。在这个过程中，你将会接触到一堆重要的概念。

Java 的集合都有一个默认的 toString 实现，但是它格式化的输出是固定的，而且往往不是你需要的样子：

```
>>> val list = listOf(1, 2, 3)
>>> println(list)                    <—— 触发 toString() 的调用
[1, 2, 3]
```

假设你需要用分号来分隔每一个元素，然后用括号括起来，而不是采用默认实现用的方括号：(1; 2; 3)。要解决这个问题，Java 项目会使用第三方的库，比如 Guava 和 Apache Commons，或者是在这个项目中重写打印函数。在 Kotlin 中，它的标准库中有一个专门的函数来处理这种情况。

本节你将自己实现这个函数。不借助 Kotlin 的工具来简化函数声明，从直接重写实现函数开始，然后再过渡到 Kotlin 更惯用的方法来重写。

下面的 joinToString 函数就展现了通过在元素中间添加分割符号，在最前面添加前缀，在最末尾添加后缀的方式把集合的元素逐个添加到一个 StringBuilder 的过程。

代码清单 3.1 joinToString() 的基本实现

```
fun <T> joinToString(
        collection: Collection<T>,
        separator: String,
        prefix: String,
        postfix: String
): String {

    val result = StringBuilder(prefix)

    for ((index, element) in collection.withIndex()) {
        if (index > 0) result.append(separator)      <—— 不用在第一个元素前添加分隔符
        result.append(element)
    }

    resuIt.append(postfix)
    return result.toString()
}
```

这个函数是泛型：它可以支持元素为任意类型的集合。这里泛型的语法和 Java 类似（在第 9 章会对泛型有更详尽的论述）。

让我们来验证一下，这个函数运行起来是不是像我们设想的那样：

```
>>> val list = listOf(1, 2, 3)
>>> println(joinToString(list, "; ", "(", ")"))
(1; 2; 3)
```

看来这个函数是可行的，你差不多可以把它丢下了。接下来，我们会聚焦到它的声明：要怎样修改，让这个函数的调用更加简洁呢？也许我们可以避免每次调用的时候，都传入 4 个参数。来看看我们能做些什么。

3.2.1 命名参数

我们要关注的第一个问题就是函数的可读性。举个例子，来看看 joinToString 的调用：

```
joinToString(collection, " ", " ", ".")
```

你能看出这些 String 都对应的是什么参数吗？这个集合的元素是用空格还是点号来分割？如果不去查看函数的声明，我们很难回答这些问题。或许你记住了这些声明，又或者你可以借助你的 IDE，但从调用代码来看，这依然很隐晦。

对于 Boolean 类型的标志，这个问题尤其明显。为解决这个问题，一些 Java 编程风格，推荐创建 enum 类型而不是采用 Boolean；而另外一些风格，会要求你通过添加注释，在注释中指明参数的名称，像这个样子：

```
/* Java */
joinToString(collection, /* separator */ " ",  /* prefix */ " ",
    /* postfix */ ".");
```

在 Kotlin 中，可以做得更优雅：

```
joinToString(collection, separator = " ", prefix = " ", postfix = ".")
```

当调用一个 Kotlin 定义的函数时，可以显式地标明一些参数的名称。如果在调用一个函数时，指明了一个参数的名称，为了避免混淆，那它之后的所有参数都需要标明名称。

小贴士 当你在重命名函数的参数时，IntelliJ IDEA 可以帮你在调用该函数的地方，一同更新命名参数。不过需要注意的是，要确保在重命名的时候，是采用的 IDEA 自带的 Rename（重命名）或者 Change Signature（改变函数签名）来处理，而不是手动地修改参数名称。

警告 不幸的是，当你调用 Java 的函数时，不能采用命名参数，不管是 JDK 中的函数，或者是 Android 框架的函数，都不行。把参数名称存到 .class 文件是 Java 8 及其更高版本的一个可选功能，而 Kotlin 需要保持和 Java 6 的兼容性。所以，编译器不能识别出调用函数的参数名称，然后把这些参数名对应到函数的定义的地方。

当我们在处理默认参数值的时候，命名参数特别有用。接下来让我们来看看。

3.2.2 默认参数值

Java 的另一个普遍存在的问题是，一些类的重载函数实在太多了。只要看一眼 `java.lang.Thread` 以及它对应的 8 个构造方法（http://mng.bz/4KZC），就让人够受的了！这些重载，原本是为了向后兼容，方便这些 API 的使用者，又或者出于别的原因，但导致的最终结果是一致的：重复。这些参数名和类型被重复了一遍又一遍，如果你是一个良好公民，还必须在每次重载的时候重复大部分的文档。与此同时，当你调用一个省略了部分参数的重载函数时，你可能会搞不清它们到底用的是哪个。

在 Kotlin 中，可以在声明函数的时候，指定参数的默认值，这样就可以避免创建重载的函数。让我们尝试改进一下前面的 `joinToString` 函数。在大多数情况下，字符串可以不加前缀或者后缀并用逗号分隔。所以，我们把这些设置为默认值。

代码清单 3.2 声明带默认参数值的 joinToString()

```
fun <T> joinToString(
        collection: Collection<T>,
        separator: String = ", ",
        prefix: String = "",
        postfix: String = ""
): String
```

有默认值的参数

现在可以用所有参数来调用这个函数，或者省略掉部分参数：

```
>>> joinToString(list, ", ", "", "")
1, 2, 3
>>> joinToString(list)
1, 2, 3
>>> joinToString(list, "; ")
1; 2; 3
```

当使用常规的调用语法时，必须按照函数声明中定义的参数顺序来给定参数，可以省略的只有排在末尾的参数。如果使用命名参数，可以省略中间的一些参数，也可以以你想要的任意顺序只给定你需要的参数：

```
>>> joinToString(list, suffix = ";", prefix = "# ")
# 1, 2, 3;
```

注意，参数的默认值是被编码到被调用的函数中，而不是调用的地方。如果你改变了参数的默认值并重新编译这个函数，没有给参数重新赋值的调用者，将会开始使用新的默认值。

默认值和 Java

考虑到 Java 没有参数默认值的概念，当你从 Java 中调用 Kotlin 函数的时候，必须显式地指定所有参数值。如果需要从 Java 代码中做频繁的调用，而且希望它能对 Java 的调用者更简便，可以用 @JvmOverloads 注解它。这个指示编译器生成 Java 重载函数，从最后一个开始省略每个参数。

举个例子，如果用 @JvmOverloads 注解了 joinToString，编译器就会生成如下的重载函数：

```
/* Java */
String joinToString(Collection<T> collection, String separator,
    String prefix, String postfix);

String joinToString(Collection<T> collection, String separator,
    String prefix);

String joinToString(Collection<T> collection, String separator);

String joinToString(Collection<T> collection);
```

每个重载函数的默认参数值都会被省略。

至此，你一直在做你的工具函数，而没有怎么关注相关的上下文。当然，这里肯定还存在一些类的函数，没有在示例中表现出来，对吧？事实上，Kotlin 已经不再需要它们了。

3.2.3 消除静态工具类：顶层函数和属性

我们都知道，Java 作为一门面向对象的语言，需要所有的代码都写作类的函数。大多数情况下，这种方式还能行得通。但事实上，几乎所有的大型项目，最终都有很多的代码并不能归属到任何一个类中。有时一个操作对应两个不同的类的对象，而且重要性相差无几。有时存在一个基本的对象，但你不想通过实例函数来添加操作，让它的 API 继续膨胀。

结果就是，最终这些类将不包含任何的状态或者实例函数，而是仅仅作为一堆静态函数的容器。在 JDK 中，最适合的例子应该就是 Collections 了。看看你自

己的代码，是不是也有一些类本身就以 Util 作为后缀命名。

在 Kotlin 中，根本就不需要去创建这些无意义的类。相反，可以把这些函数直接放到代码文件的顶层，不用从属于任何的类。这些放在文件顶层的函数依然是包内的成员，如果你需要从包外访问它，则需要 import，但不再需要额外包一层。

让我们来把 joinToString 直接放到 strings 的包中试一下。依照下面这样，创建一个名为 join.kt 的文件：

代码清单 3.3 声明 joinToString() 作为顶层函数

```
package strings

fun joinToString(...): String { ... }
```

这会怎么运行呢？你知道，当你编译这个文件的时候，会生成一些类，因为 JVM 只能执行类中的代码。当你在使用 Kotlin 的时候，知道这些就够了。但是，如果需要从 Java 中来调用这些函数，你就必须理解它将会怎样被编译。为方便理解，我们来看一段 Java 代码，这里它会编译成相同的类：

```
/* Java */
package strings;

public class JoinKt {
    public static String joinToString(...) { ... }
}
```

对应代码清单 3.3 的文件名称，join.kt

可以看到 Kotlin 编译生成的类的名称，对应于包含函数的文件的名称。这个文件中的所有顶层函数编译为这个类的静态函数。因此，当从 Java 调用这个函数的时候，和调用任何其他静态函数一样非常简单：

```
/* Java */
import strings.JoinKt;

...

JoinKt.joinToString(list, ", ", "", "");
```

修改文件类名

要改变包含 Kotlin 顶层函数的生成的类的名称，需要为这个文件添加 @JvmName 的注解，将其放到这个文件的开头，位于包名的前面：

```
@file:JvmName("StringFunctions")

package strings

fun joinToString(...): String { ... }
```

注解指定类名

包的声明跟在文件注解之后

现在我们可以这样调用这个函数：

```
/* Java */
import strings.StringFunctions;
StringFunctions.joinToString(list, ", ", "", "");
```

关于注解语法，我们将会在第 10 章做更详尽的讨论。

顶层属性

和函数一样，属性也可以放到文件的顶层。在一个类的外面保存单独的数据片段虽然不常用，但还是有它的价值。

举个例子，可以用 var 属性来计算一些函数被执行的次数：

```
var opCount = 0                          <-- 声明一个顶层属性

fun performOperation() {
    opCount++                            <-- 改变该属性的值
    // ...
}

fun reportOperationCount() {
    println("Operation performed $opCount times")   <-- 读取属性值
}
```

像这个值就会被存储到一个静态的字段中。

也可以在代码中用顶层属性来定义常量：

```
val UNIX_LINE_SEPARATOR = "\n"
```

默认情况下，顶层属性和其他任意的属性一样，是通过访问器暴露给 Java 使用的（如果是 val 就只有一个 getter，如果是 var 就对应一对 getter 和 setter）。为了方便使用，如果你想要把一个常量以 public static final 的属性暴露给 Java，可以用 const 来修饰它（这个适用于所有的基本数据类型的属性，以及 String 类型）。

```
const val UNIX_LINE_SEPARATOR = "\n"
```

这个等同于下面的 Java 代码：

```
/* Java */
public static final String  UNIX_LINE_SEPARATOR = "\n";
```

对于 joinToString 这个工具函数我们已经改进了很多。现在，让我们来看看，还能怎样让它更好用。

3.3 给别人的类添加方法：扩展函数和属性

Kotlin 的一大特色，就是可以平滑地与现有代码集成。甚至，纯 Kotlin 的项目都可以基于 Java 库构建，如 JDK、Android 框架，以及其他的第三方框架。当你在一个现有的 Java 项目中集成 Kotlin 的时候，依然需要面临现有代码目前不能转成 Kotlin，甚至将来也不会转成 Kotlin 的局面。当使用这些 API 的时候，如果不用重写，就能使用到 Kotlin 为它带来的方便，岂不是更好？这里，可以用扩展函数来实现。

理论上来说，*扩展函数*非常简单，它就是一个类的成员函数，不过定义在类的外面。为了方便阐释，让我们添加一个方法，来计算一个字符串的最后一个字符：

```
package strings

fun String.lastChar(): Char = this.get(this.length - 1)
```

你所有要做的，就是把你要扩展的类或者接口的名称，放到即将添加的函数前面。这个类的名称被称为*接收者类型*；用来调用这个扩展函数的那个对象，叫作*接收者对象*，如图 3.1 所示。

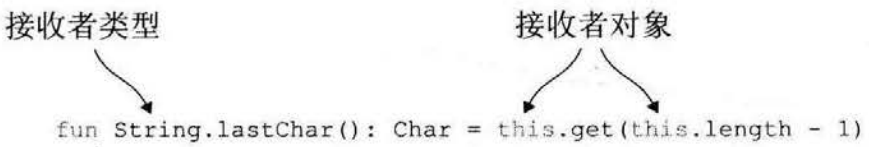


图 3.1 接收者类型是由扩展函数定义的，接收者对象是该类型的一个实例

可以像调用类的普通成员函数一样去调用这个函数：

```
>>> println("Kotlin".lastChar())
n
```

在这个例子中，String 就是接收者类型，而 "Kotlin" 就是接收者对象。

从某种意义上说，你已经为 String 类添加了自己的方法。即使字符串不是代码的一部分，也没有类的源代码，你仍然可以在自己的项目中根据需要去扩展方法。不管 String 类是用 Java、Kotlin，或者像 Groovy 的其他 JVM 语言编写的，只要是它会编译为 Java 类，你就可以为这个类添加自己的扩展。

在这个扩展函数中，可以像其他成员函数一样用 this。而且也可以像普通的成员函数一样，省略它。

```
package strings

fun String.lastChar(): Char = get(length - 1)
```

接收者对象成员可以不用 this 来访问

在扩展函数中，可以直接访问被扩展的类的其他方法和属性，就好像是在这个

类自己的方法中访问它们一样。注意，扩展函数并不允许你打破它的封装性。和在类内部定义的方法不同的是，扩展函数不能访问私有的或者是受保护的成员。

后面，我们将为类的成员函数和扩展函数使用术语*方法*。例如，可以说，在扩展函数的内部，可以在接收者上调用该类的任意方法，即，无论成员函数还是扩展函数，都可以调用。在调用的一方，扩展函数与成员函数没有区别，一般情况下，无论这个方法是成员函数还是扩展函数都不重要。

3.3.1 导入和扩展函数

对于你定义的一个扩展函数，它不会自动地在整个项目范围内生效。相反，如果你要使用它，需要进行导入，就像其他任何的类或者函数一样。这是为了避免偶然性的命名冲突。Kotlin 允许用和导入类一样的语法来导入单个的函数：

```
import strings.lastChar
val c = "Kotlin".lastChar()
```

当然，用 * 来导入也是可以的：

```
import strings.*
val c = "Kotlin".lastChar()
```

可以使用关键字 as 来修改导入的类或者函数名称：

```
import strings.lastChar as last
val c = "Kotlin".last()
```

当你在不同的包中，有一些重名的函数时，在导入时给它重新命名就显得很有必要了，这样可以在同一个文件中去使用它们。在这种情况下，对于一般的类和函数，还有另一个选择：可以选择用全名来指出这个类或者函数。对于扩展函数，Kotlin 的语法要求你用简短的名称，所以，在导入声明的时候，关键字 as 就是你解决命名冲突问题的唯一方式。

3.3.2 从 Java 中调用扩展函数

实质上，扩展函数是静态函数，它把调用对象作为了它的第一个参数。调用扩展函数，不会创建适配的对象或者任何运行时的额外消耗。

这使得从 Java 中调用 Kotlin 的扩展函数变得非常简单：调用这个静态函数，然后把接收者对象作为第一个参数传进去即可。和其他顶层函数一样，包含这个函数的 Java 类的名称，是由这个函数声明的文件名称决定的。假设它声明在一个叫作 StringUtil.kt 的文件中：

```
/* Java */
char c = StringUtilKt.lastChar("Java");
```

这个扩展函数被声明为顶层函数，所以，它将会被编译为一个静态函数。在 Java 中静态导入 lastChar 函数，就可以直接使用它了，如 `lastChar("Java")`。和 Kotlin 版本比较起来，这个写法显得可读性略微差一点，但从 Java 的角度来看，这个也很正常。

3.3.3 作为扩展函数的工具函数

现在，可以写一个 `joinToString` 函数的终极版本了，它和你在 Kotlin 标准库中看到的一模一样。

代码清单 3.4 声明扩展函数 joinToString()

```
fun <T> Collection<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = ""
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex())
        if (index > 0) result.append(separator)
        result.append(element)
    }

    result.append(postfix)
    return result.toString()
}
```

为 Collection<T> 声明一个扩展函数

为参数赋默认值

this 指向接收者对象：T 的集合

```
>>> val list = listOf(1, 2, 3)
>>> println(list.joinToString(separator = "; ",
...         prefix = "(", postfix = ")"))
(1; 2; 3)
```

可以给元素的集合类添加一个扩展函数，然后给所有的参数添加一个默认值。这样，就可以像使用一个类的成员函数一样，去调用 `joinToString` 了：

```
>>> val list = arrayListOf(1, 2, 3)
>>> println(list.joinToString(" "))
1 2 3
```

因为扩展函数无非就是静态函数的一个高效的语法糖，可以使用更具体的类型来作为接收者类型，而不是一个类。假设你想要一个 `join` 函数，只能由字符串的集合来触发。

```
fun Collection<String>.join(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = ""
) = joinToString(separator, prefix, postfix)

>>> println(listOf("one", "two", "eight").join(" "))
one two eight
```

如果是用其他类型的对象列表来调用，将会报错：

```
>>> listOf(1, 2, 8).join()
Error: Type mismatch: inferred type is List<Int> but Collection<String>
was expected.
```

扩展函数的静态性质也决定了扩展函数不能被子类重写。让我们来看一个例子。

3.3.4 不可重写的扩展函数

在 Kotlin 中，重写成员函数是很平常的一件事情。但是，不能重写扩展函数。假设这里有两个类，View 和它的子类 Button，然后 Button 重写了父类的 click 函数。

代码清单 3.5 重写成员函数

```
open class View {
    open fun click() = println("View clicked")
}

class Button: View() {
    override fun click() = println("Button clicked")
}
```

Button 继承 View

当你声明了类型为 View 的变量，那它可以被赋值为 Button 类型的对象，因为 Button 是 View 的一个子类。当你在调用这个变量的一般函数，比如 click 的时候，如果这个函数被 Button 重写了，那么这里将会调用到 Button 中重写的函数：

```
>>> val view: View = Button()
>>> view.click()
Button clicked
```

具体调用哪个方法，由实际的 view 的值来决定

但是对于扩展函数来说，并不是这样的，如图 3.2 所示。

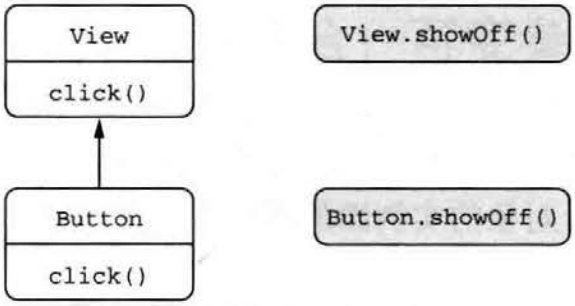


图 3.2 扩展函数声明在类的外部

扩展函数并不是类的一部分，它是声明在类之外的。尽管可以给基类和子类都分别定义一个同名的扩展函数，当这个函数被调用时，它会用到哪一个呢？这里，它是由该变量的静态类型所决定的，而不是这个变量的运行时类型。

下面的例子就展示了两个分别声明在 View 和 Button 的 showOff 扩展函数。

代码清单 3.6 不能重写扩展函数

```
fun View.showOff() = println("I'm a view!")
fun Button.showOff() = println("I'm a button!")

>>> val view: View = Button()
>>> view.showOff()
I'm a view!
```

扩展函数被静态地解析

当你在调用一个类型为 View 的变量的 showOff 函数时，对应的扩展函数会被调用，尽管实际上这个变量现在是一个 Button 的对象。

回想一下，扩展函数将会在 Java 中被编译为静态函数，同时接收值将会作为第一个参数，对于此你应该很清楚，因为 Java 会执行相同的函数：

```
/* Java */
>>> View view = new Button();
>>> ExtensionsKt.showOff(view);
I'm a view!
```

showOff 函数声明在 extensions.kt 文件中

如你所见，扩展函数并不存在重写，因为 Kotlin 会把它们当作静态函数对待。

注意 如果一个类的成员函数和扩展函数有相同的签名，成员函数往往会被优先使用。你应该牢记，当在扩展 API 类的时候：如果添加一个和扩展函数同名的成员函数，那么对应类定义的消费者将会重新编译代码，这将会改变它的意义并开始指向新的成员函数。

我们已经讨论了怎样给现有的类添加扩展函数，现在让我们来看看，该怎样去给它添加扩展属性。

3.3.5 扩展属性

扩展属性提供了一种方法，用来扩展类的 API，可以用来访问属性，用的是属性语法而不是函数的语法。尽管它们被称为*属性*，但它们可以没有任何状态，因为没有合适的地方来存储它，不可能给现有的 Java 对象的实例添加额外的字段。但有时短语法仍然是便于使用的。

在上一节，我们又定义了一个 `lastChar` 的函数，现在让我们把它转换成一个属性试试。

代码清单 3.7 声明一个扩展属性

```
val String.lastChar: Char
    get() = get(length - 1)
```

可以看到，和扩展函数一样，扩展属性也像接收者的一个普通的成员属性一样。这里，必须定义 getter 函数，因为没有支持字段，因此没有默认 getter 的实现。同理，初始化也不可以：因为没有地方存储初始值。

如果在 `StringBuilder` 上定义一个相同的属性，可以置为 `var`，因为 `StringBuilder` 的内容是可变的。

代码清单 3.8 声明一个可变的扩展属性

```
var StringBuilder.lastChar: Char
    get() = get(length - 1)                    <—— getter 属性
    set(value: Char) {
        this.setCharAt(length - 1, value)      <—— setter 属性
    }
```

可以像访问使用成员属性一样访问它：

```
>>> println("Kotlin".lastChar)
n
>>> val sb = StringBuilder("Kotlin?")
>>> sb.lastChar = '!'
>>> println(sb)
Kotlin!
```

注意，当你需要从 Java 中访问扩展属性的时候，应该显式地调用它的 getter 函数：`StringUtilKt.getLastChar("Java")`。

关于扩展函数和属性的概念我们已经讨论得差不多了，现在，让我们回到集合的话题，看一些库提供的能帮助你处理集合的函数，以及伴随而来的语言特性。

3.4 处理集合：可变参数、中缀调用和库的支持

这一节将会展示 Kotlin 标准库中用来处理集合的一些方法。另外，也会涉及几个相关的语言特性：

- 可变参数的关键字 vararg，可以用来声明一个函数将可能有任意数量的参数
- 一个*中缀*表示法，当你在调用一些只有一个参数的函数时，使用它会让代码更简练
- *解构声明*，用来把一个单独的组合值展开到多个变量中

3.4.1 扩展 Java 集合的 API

我们开始本章的前提，是基于 Kotlin 中的集合与 Java 的类相同，但对 API 做了扩展。可以看一个示例，用来获取列表中最后一个元素并找到数字集合中的最大值：

```
>>> val strings: List<String> = listOf("first", "second", "fourteenth")
>>> strings.last()
fourteenth

>>> val numbers: Collection<Int> = setOf(1, 14, 2)
>>> numbers.max()
14
```

我们感兴趣的是它是怎么工作的：尽管它们是 Java 库类的实例，为什么在 Kotlin 中能对集合有这么多丰富的操作。现在答案很明显了：因为函数 last 和 max 都被声明成了扩展函数。

last 函数不会比 String 的 lastChar 更复杂，在上一节中讨论过：它是 List 类的一个扩展函数。对于 max，我们做一个简单的声明（真正的库函数不仅用于 Int 数字，而且适用于任何可比较的元素）：

```
fun <T> List<T>.last(): T { /* 返回最后一个元素 */ }
fun Collection<Int>.max(): Int { /* 找到集合的最大值 */ }
```

许多扩展函数在 Kotlin 标准库中都有声明，在这里，我们不会列出所有这些方法。你可能会想知道，在 Kotlin 标准库中学习所有内容的最佳方式。这个并没有必要，在你需要操作集合或任何其他对象的时候，IDE 中的代码补全功能，将为你列出所有可用于该类型对象的方法，不管是普通函数或者是扩展函数，都会有显示，你可以选择所需的方法。除此之外，标准库的引用会列出库中每个类的所有可用的函数，包括成员函数及扩展函数。

在本章的开头，你已经看过了创建集合的函数。这些函数都有一些共同特征，就是它们可以被任意数量的参数调用。在下一节中，你将看到声明这些函数的语法。

3.4.2 可变参数：让函数支持任意数量的参数

当你在调用一个函数来创建列表的时候，可以传递任意个数的参数给它：

```
val list = listOf(2, 3, 5, 7, 11)
```

如果看看这个函数在库当中的声明，你会将会发现：

```
fun listOf<T>(vararg values: T): List<T> { ... }
```

你可能对 Java 的可变参数已经很熟悉了，通过这个功能，可以把任意个数的参数值打包到数组中传给函数。Kotlin 的可变参数与 Java 类似，但语法略有不同：Kotlin 在该类型之后不会再使用三个点，而是在参数上使用 vararg 修饰符。

Kotlin 和 Java 之间的另一个区别是，当需要传递的参数已经包装在数组中时，调用该函数的语法。在 Java 中，可以按原样传递数组，而 Kotlin 则要求你显式地解包数组，以便每个数组元素在函数中能作为单独的参数来调用。从技术的角度来讲，这个功能被称为*展开运算符*，而使用的时候，不过是在对应的参数前面放一个 * ：

```
fun main(args: Array<String>) {
    val list = listOf("args: ", *args)
    println(list)
}
```

展开运算符展开数组内容

这个示例展示了，通过展开运算符，可以在单个调用中组合来自数组的值和某些固定值。这在 Java 中并不支持。

接下来，我们再来看看 map，我们将简要讨论另一种提高 Kotlin 函数调用的可读性的方法：*中缀调用*。

3.4.3 键值对的处理：中缀调用和解构声明

可以使用 mapOf 函数来创建 map:

```
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

在本章的开头说过会提供另一个解释，现在时候到了。这行代码中的单词 to 不是内置的结构，而是一种特殊的函数调用，被称为*中缀调用*。

在中缀调用中，没有添加额外的分隔符，函数名称是直接放在目标对象名称和参数之间的。以下两种调用方式是等价的：

```
1.to("one")
1 to "one"
```

一般 to 函数的调用

使用中缀符号调用 to 函数

中缀调用可以与只有一个参数的函数一起使用，无论是普通的函数还是扩展函数。要允许使用中缀符号调用函数，需要使用 infix 修饰符来标记它。下面是一个简单的 to 函数的声明：

```
infix fun Any.to(other: Any) = Pair(this, other)
```

to 函数会返回一个 Pair 类型的对象，Pair 是 Kotlin 标准库中的类，不出所料，它会用来表示一对元素。Pair 和 to 的声明都用到了泛型，简单起见，这里我们省略了泛型。

注意，可以直接用 Pair 的内容来初始化两个变量：

```
val (number, name) = 1 to "one"
```

这个功能称为解构声明。图 3.3 展现了它如何与 Pair 一起使用。

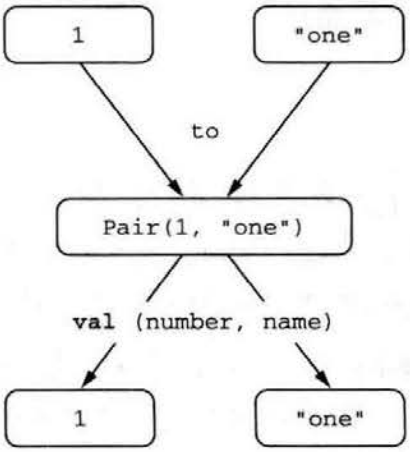


图 3.3 用 to 函数创建一个 pair，然后用解构声明来展开

解构声明特征不止用于 pair。例如，还可以使用 map 的 key 和 value 内容来初始化两个变量。

这也适用于循环，正如你在使用 withIndex 函数的 joinToString 实现中看到的：

```
for ((index, element) in collection.withIndex()) {
    println("$index: $element")
}
```

7.4 节将描述解构表达式的一般规则，并使用它来初始化几个变量。

to 函数是一个扩展函数，可以创建一对任何元素，这意味着它是泛型接收者的扩展：可以使用 1 to "one"、"one" to 1、list to list.size() 等写法。我们来看看 mapOf 函数的声明：

```
fun <K, V> mapOf(vararg values: Pair<K, V>): Map<K, V>
```

像 listOf 一样，mapOf 接收可变数量的参数，但是这次它们应该是键值对。

尽管在 Kotlin 中创建新的 map 可能看起来像特殊的解构，而它不过是一个具有简明语法的常规函数。接下来，我们来讨论扩展函数如何简化字符串和正则表达式的操作。

3.5 字符串和正则表达式的处理

Kotlin 字符串与 Java 字符串完全相同。可以将在 Kotlin 代码中创建的字符串传递给任何 Java 函数，也可以把任何 Kotlin 标准库函数应用到从 Java 代码接收的字符串上，而不用转换，也不用创建附加的包装对象。

Kotlin 通过提供一系列有用的扩展函数，使标准 Java 字符串使用起来更加方便。此外，它还隐藏了一些令人费解的函数，添加了一些更清晰易用的扩展。作为体现 API 差异化的第一个例子，我们来看看 Kotlin 是如何分割字符串的。

3.5.1 分割字符串

你可能已经对 String 的 spilt 方法很熟悉了。每个人都在使用它，但有时候人们在 Stack Overflow（http://stackoverflow.com）上抱怨："Java 中的 Split 方法不适用于一个点号。"当代码写为 "12.345-6.A".split(".") 的时候，我们期待的结果是得到一个 [12, 345-6, A] 数组。但是 Java 的 split 方法竟然返回一个空数组！这是因为它将一个正则表达式作为参数，并根据表达式将字符串分割成多个字符串。这里点号（.）是表示任何字符的正则表达式。

Kotlin 把这个令人费解的函数隐藏了，作为替换，它提供了一些名为 split 的，具有不同参数的重载的扩展函数。用来承载正则表达式的值需要一个 Regex 类型，而不是 String。这样确保了当有一个字符串传递给这些函数的时候，不会被当作正则表达式。

这里这样可以用一个点号或者破折号来分割字符串：

```
>>> println("12.345-6.A".split("\\.|-".toRegex()))
[12, 345, 6, A]
```

显式地创建一个正则表达式

Kotlin 使用与 Java 中完全相同的正则表达式语法。这里的模式匹配一个点（我们对它进行转义来表示我们指的是字面量，而不是通配符）或破折号。使用正则表达式的 API 也类似于标准 Java 库 API，但它们更为通用。例如，在 Kotlin 中，可以使用扩展函数 toRegex 将字符串转换为正则表达式。

但是对于一些简单的情况，就不需要使用正则表达式了。Kotlin 中的 split 扩展函数的其他重载支持任意数量的纯文本字符串分隔符：

```
>>> println("12.345-6.A".split(".", "-"))
[12, 345, 6, A]
```

指定多个分隔符

注意，你可以指定字符参数，并写入 `"12.345-6.A".split('.', '-')`，这样的结果是相同的。这个方法将替换 Java 中类似的，只能使用一个字符作为分隔符的方法。

3.5.2 正则表达式和三重引号的字符串

我们来看另一个例子的两种不同实现：一个将使用扩展函数来处理字符串，另一个将使用正则表达式。你的任务是将解析文件的完整路径名称到对应的组件：目录、文件名和扩展名。Kotlin 标准库中包含了一些可以用来获取在给定分隔符第一次（或最后一次）出现之前（或之后）的子字符串的函数。可以这样使用它们来解决这个问题（见图 3.4）。

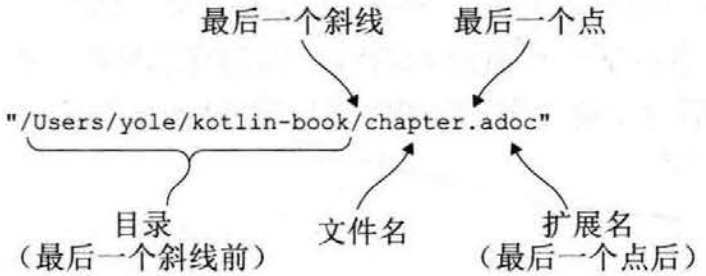


图 3.4 使用 `substringBeforeLast` 和 `substringAfterLast` 函数将一个路径分割为目录、文件名和扩展名

代码清单 3.9 使用 `String` 的扩展函数来解析文件路径

```
fun parsePath(path: String) {
    val directory = path.substringBeforeLast("/")
    val fullName = path.substringAfterLast("/")

    val fileName = fullName.substringBeforeLast(".")
    val extension = fullName.substringAfterLast(".")

    println("Dir: $directory, name: $fileName, ext: $extension")
}
>>> parsePath("/Users/yole/kotlin-book/chapter.adoc")
Dir: /Users/yole/kotlin-book, name: chapter, ext: adoc
```

path 字符串中的最后的一个斜线之前的部分，是目录的路径；点号之后的部分，是文件的扩展名；而文件名称，介于两者之间。

解析字符串在 Kotlin 中变得更加容易，而且不需要使用正则表达式，这个功能非常强大，但有时这样的代码会让人费解。如果你确实想要使用正则表达式，也可

以使用 Kotlin 标准库。下面展示的是怎样使用正则表达式完成相同的事情：

代码清单 3.10 使用正则表达式解析文件路径

```
fun parsePath(path: String) {
    val regex = """(.+)/(.+)\.(.+)""".toRegex()
    val matchResult = regex.matchEntire(path)
    if (matchResult != null) {
        val (directory, filename, extension) = matchResult.destructured
        println("Dir: $directory, name: $filename, ext: $extension")
    }
}
```

在这个例子中，正则表达式写在一个三重引号的字符串中。在这样的字符串中，不需要对任何字符进行转义，包括反斜线，所以可以用 \. 而不是 \\. 来表示点，正如写一个普通的字符串字面值（见图 3.5）一样。这个正则表达式将一个路径分为三个由斜线和点分隔的组。这个 . 模式从字符串的一开始就进行匹配，所以第一组（.+）包含最后一个斜线之前的子串。这个子串包括所有前面的斜线，因为它们匹配“任何字符”的模式。同理，第二组包含最后一个点之前的子串，第三组包含剩余部分。

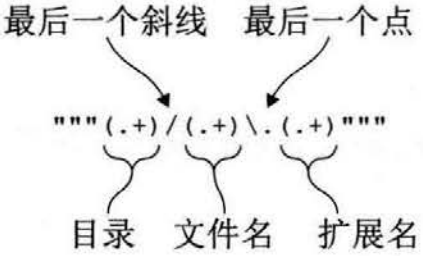


图 3.5 使用正则表达式将一个路径分割为目录、文件名和扩展名

现在我们来研究一下，上个例子中 parsePath 函数的实现：创建一个正则表达式，并将它与输入路径进行匹配。如果匹配成功（结果不为 null），则将它 destructured 属性赋给相应的变量。这个和使用 Pair 初始化两个变量的语法是相同的，将在 7.4 节将详细介绍。

3.5.3 多行三重引号的字符串

三重引号字符串的目的，不仅在于避免转义字符，而且使它可以包含任何字符，包括换行符。另外，它提供了一种更简单的方法，从而可以简单地把包含换行符的文本嵌入到程序中。例如，可以用 ASCII 码来画点东西：

```
val kotlinLogo = """| //
                   .|//
                   .|/ \"""
>>> println(kotlinLogo.trimMargin("."))
| //
|//
|/ \
```

多行字符串包含三重引号之间的所有字符，包括用于格式化代码的缩进。如果要更好地表示这样的字符串，可以去掉缩进（左边距）。为此，可以向字符串内容添加前缀，标记边距的结尾，然后调用 trimMargin 来删除每行中的前缀和前面的空格。在前面的例子中，它就是使用了点来作为前缀。

一个三重引号的字符串可以包含换行，而不用专门的字符，比如 \n。另一方面，可以不必转义字符 \，所以 Windows 风格的路径 "C:\\Users\\yole\\kotlin-book" 可以写成 """C:\Users\yole\kotlin-book"""。

还可以在多行字符串中使用字符串模板。因为多行字符串不支持转义序列，如果需要在字符串的内容中使用美元符号的字面量，则必须使用嵌入式表达式，像这样：val price ="""${'$'} 99.9"""。

除了使用 ASCII 的画图游戏外，需要使用多行字符串的一个地方是测试。在测试中，往往需要执行产生多行文本（例如，网页片段）的操作并将结果与预期输出进行比较。多行字符串完美地解决了将预期输出作为测试的一部分的问题。不需要笨拙的转义或从外部文件中加载文本，只需放入一些引号并将预期的 HTML 或其他输出放在它们中间。为了更好的格式化，请使用前面提到的 trimMargin 函数，这是扩展函数的另一个例子。

> **注意** 可以看到，扩展函数非常强大的地方，是可以扩展现有库的 API 并将其适应新语言，有时候被称为 Pimp My Library 模式。[1] 事实上，大部分 Kotlin 标准库是由标准 Java 类的扩展函数组成的。由 JetBrains 构建的 Anko 库（https://github.com/kotlin/anko）提供的扩展函数，使 Android API 对Kotlin更加友好。你还可以找到很多社区开发的库，对主要的第三方库（如 Spring）提供了友好的 Kotlin 包装。

现在，你已经看到了 Kotlin 是如何提供更好的 API 的，让我们把注意力回到你的代码。接下来，将介绍扩展函数的一些新用途，还将讨论一个新的概念：局部函数。

---

1 Martin Odersky，“Pimp My Library”，*Artima Developer*，2006年9月9日，http://mng.bz/86Qh。

3.6 让你的代码更整洁：局部函数和扩展

许多开发人员认为，好代码的重要标准之一是减少重复代码，甚至还给这个原则起了一个特殊的名字：不要重复你自己（DRY）。但是当你写 Java 代码的时候，有时候要做到这一点就不那么容易了。在许多情况下，可以使用 IDE 的 Extract Method（抽取方法）的重构方法把长的方法分解成更小的代码块，然后重用这些代码。但是，这样可能让代码更费解，因为你以一个包含许多小方法的类告终，而且它们之间并没有明确的关系。可以更进一步地将提取的函数组合成一个内部类，这样就可以保持结构，但是这种函数需要用到大量的样板代码。

Kotlin 提供了一个更整洁的方案：可以在函数中嵌套这些提取的函数。这样，既可以获得所需的结构，也无须额外的语法开销。

让我们来看看，怎样使用局部函数，来解决常见的代码重复问题。在下面的例子中，saveUser 函数用于将 user 的信息保存到数据库，并且确保 user 的对象包含有效数据。

代码清单 3.11 带重复代码的函数

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

fun saveUser(user: User) {
    if (user.name.isEmpty()) {
        throw IllegalArgumentException(
            "Can't save user ${user.id}: empty Name")
    }

    if (user.address.isEmpty()) {
        throw IllegalArgumentException(
            "Can't save user ${user.id}: empty Address")
    }

    // 保存 user 到数据库
}

>>> saveUser(User(1, "", ""))
java.lang.IllegalArgumentException: Can't save user 1: empty Name
```

重复的字段检查

这里的重复代码是很少的，你可能不想要在类中一个面面俱到的方法中，去验证用户字段的每一种特殊情况。但是，如果将验证代码放到局部函数中，可以摆脱重复，并保持清晰的代码结构，可以这样做：

代码清单 3.12 提取局部函数来避免重复

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

fun saveUser(user: User) {
```

```
    fun validate(user: User,                                    ← 声明一个局部函数来验证所有字段
                 value: String,
                 fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user ${user.id}: empty $fieldName")
        }
    }

    validate(user, user.name, "Name")                           ← 调用局部函数来验证特定字段
    validate(user, user.address, "Address")

    // 保存 user 到数据库
}
```

这看起来好多了，不用重复验证逻辑，如果在项目演进时需要向 User 添加其他字段，也可以轻松添加更多验证。但是，把 User 对象传递给验证函数看起来还是有点难看。好消息是，这完全不必要，因为局部函数可以访问所在函数中的所有参数和变量。我们可以利用这一点，去掉冗余的 User 参数。

代码清单 3.13 在局部函数中访问外层函数的参数

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

fun saveUser(user: User) {
    fun validate(value: String, fieldName: String) {           ← 现在不用在 saveUser 函数中重复 user 参数了
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user ${user.id}: " +               ← 可以直接访问外部函数的参数
                    "empty $fieldName")
        }
    }

    validate(user.name, "Name")
    validate(user.address, "Address")

    // 保存 user 到数据库
}
```

我们可以继续改进，把验证逻辑放到 User 类的扩展函数中。

代码清单 3.14 提取逻辑到扩展函数

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

fun User.validateBeforeSave() {
    fun validate(value: String, fieldName: String) {
        if (value.isEmpty()) {
            throw IllegalArgumentException(
                "Can't save user $id: empty $fieldName")       ← 可以直接访问 User 的属性
        }
    }
```

```
    validate(name, "Name")
    validate(address, "Address")
}

fun saveUser(user: User) {
    user.validateBeforeSave()          <-- 调用扩展函数
    // 保存 user 到数据库
}
```

将一段代码提取到扩展函数中，看起来相当有效。即使 User 属于当前代码库而不是库类的一部分，你也不会希望将这段逻辑放到 User 的方法当中，因为它和其他用到 User 的地方没有关系。如果你能遵循，类的 API 只能包含必需的方法，那么就可以让类保持精炼的同时，也让你的思路更加清晰。另一方面，主要处理单个对象，并且不需要访问其私有数据的函数，可以访问它的成员而无须额外验证，如代码清单 3.14 所示。

扩展函数也可以被声明为局部函数，所以，这里可以进一步将 User.validateBeforeSave 作为局部函数放在 saveUser 中。但是深度嵌套的局部函数往往让人费解。因此，一般我们不建议使用多层嵌套。

在了解了函数的各种有趣操作之后，在下一章，我们将看看类的操作。

3.7 小结

- Kotlin 没有定义自己的集合类，而是在 Java 集合类的基础上提供了更丰富的 API。
- Kotlin 可以给函数参数定义默认值，这样大大降低了重载函数的必要性，而且命名参数让多参数函数的调用更加易读。
- Kotlin 允许更灵活的代码结构：函数和属性都可以直接在文件中声明，而不仅仅是在类中作为成员。
- Kotlin 可以用扩展函数和属性来扩展任何类的 API，包括在外部库中定义的类，而不需要修改其源代码，也没有运行时开销。
- 中缀调用提供了处理单个参数的，类似调用运算符方法的简明语法。
- Kotlin 为普通字符串和正则表达式都提供了大量的方便字符串处理的函数。
- 三重引号的字符串提供了一种简洁的方式，解决了原本在 Java 中需要进行大量啰唆的转义和字符串连接的问题。
- 局部函数帮助你保持代码整洁的同时，避免重复。

4 类、对象和接口

本章内容包括

- 类和接口
- 非默认属性和构造方法
- 数据类
- 类委托
- 使用 object 关键字

本章会让你加深对如何使用 Kotlin 的类的理解。在第 2 章已经见过了声明一个类的基本语法，也知道了如何使用简单的主构造方法来声明方法和属性（是不是很好用？），以及使用枚举。但是还有更多可以看的内容。

Kotlin 的类和接口与 Java 的类和接口还是有一点区别的。例如，接口可以包含属性声明。与 Java 不同，Kotlin 的声明默认是 final 和 public 的。此外，嵌套的类默认并不是内部类：它们并没有包含对其外部类的隐式引用。

对于构造方法来说，简短的主构造方法语法在大多数情况下都工作得很好，但是依然有完整的语法可以让你声明带有重要初始化逻辑的构造方法。对于属性来说也是一样的：简洁的语法非常好用，但是你还是可以方便地定义你自己的访问器实现。

Kotlin 编译器能够生成有用的方法来避免冗余。将一个类声明为 data 类可以

让编译器为这个类生成若干标准方法。同样可以避免动手书写委托方法，因为委托模式是 Kotlin 原生支持的。

本章还介绍了一个新的可以声明类并创建这个类的一个实例的 object 关键字。这个关键字用来表示单例对象、伴生对象和对象表达式（类似于 Java 的匿名类）。让我们从类和接口以及 Kotlin 中定义类继承结构的精妙之处开始谈起吧。

4.1 定义类继承结构

本节内容会将 Kotlin 的类继承结构定义与 Java 做比较。我们会关注到 Kotlin 的可见性和访问修饰符，它们与 Java 中的类似但还是有一些不一样的默认行为。你还会学到新的用于限制一个类可能存在的子类的 sealed 修饰符。

4.1.1 Kotlin 中的接口

我们会以关注接口的定义和实现来作为开始。Kotlin 的接口与 Java 8 中的相似：它们可以包含抽象方法的定义以及非抽象方法的实现（与 Java 8 中的默认方法类似），但它们不能包含任何状态。

使用 interface 关键字而不是 class 来声明一个 Kotlin 的接口。

代码清单 4.1 声明一个简单的接口

```
interface Clickable {
    fun click()
}
```

这声明了一个拥有名为 click 的单抽象方法的接口。所有实现这个接口的非抽象类都需要提供这个方法的一个实现。接下来就是你该如何实现这个接口。

代码清单 4.2 实现一个简单接口

```
class Button : Clickable {
    override fun click() = println("I was clicked")
}

>>> Button().click()
I was clicked
```

Kotlin 在类名后面使用冒号来代替了 Java 中的 extends 和 implements 关键字。和 Java 一样，一个类可以实现任意多个接口，但是只能继承一个类。

与 Java 中的 @Override 注解类似，override 修饰符用来标注被重写的父类或者接口的方法和属性。与 Java 不同的是，在 Kotlin 中使用 *override* 修饰符是

强制要求的。这会避免先写出实现方法再添加抽象方法造成的意外重写；你的代码将不能编译，除非你显式地将这个方法标注为 override 或者重命名它。

接口的方法可以有一个默认实现。与 Java 8 不同的是，Java 8 中需要你在这样的实现上标注 default 关键字，对于这样的方法，Kotlin 没有特殊的注解：只需要提供一个方法体。让我们来给 Clickable 接口添加一个带默认实现的方法。

代码清单 4.3 在接口中定义一个带方法体的方法

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable!")
}
```

普通的方法声明

带默认实现的方法

如果你实现了这个接口，你需要为 click 提供一个实现。可以重新定义 showOff 方法的行为，或者如果你对默认行为感到满意也可以直接省略它。

现在让我们假设存在同样定义了一个 showOff 方法并且有如下实现的另一个接口。

代码清单 4.4 定义另一个实现了同样方法的接口

```
interface Focusable {
    fun setFocus(b: Boolean) =
        println("I ${if (b) "got" else "lost"} focus.")

    fun showOff() = println("I'm focusable!")
}
```

如果需要在你的类中实现这两个接口会发生什么？它们每一个都包含了带默认实现的 showOff 方法；将会使用哪一个实现？答案是，任何一个都不会使用。取而代之的是，如果你没有显式实现 showOff，你会得到如下的编译错误：

```
The class 'Button' must
override public open fun showOff() because it inherits
many implementations of it.
```

Kotlin 编译器强制要求你提供你自己的实现。

代码清单 4.5 调用继承自接口方法的实现

```
class Button : Clickable, Focusable {
    override fun click() = println("I was clicked")
```

```
    override fun showOff() {
        super<Clickable>.showOff()
        super<Focusable>.showOff()
    }
}
```

如果同样的继承成员有不止一个实现，必须提供一个显式实现

使用尖括号加上父类型名字的“super”表明了你想要调用哪一个父类的方法

现在 Button 类实现了两个接口。通过调用继承的两个父类型中的实现来实现 showOff()。要调用一个继承的实现，可以使用与 Java 相同的关键字：super。但是选择一个特定实现的语法是不同的。在 Java 中可以把基类的名字放在 super 关键字的前面，就像 Clickable.super.showOff() 这样，在 Kotlin 中需要把基类的名字放在尖括号中：super<Clickable>.showOff()。

如果只需要调用一个继承的实现，可以这样写：

```
override fun showOff() = super<Clickable>.showOff()
```

可以创建一个这个类的实例来验证所有继承的方法都可以被调用到。

```
fun main(args: Array<String>) {
    val button = Button()
    button.showOff()
    button.setFocus(true)
    button.click()
}
```

I'm clickable!
I'm focusable!

I got focus.

I was clicked.

setFocus 的实现是在 Focusable 接口中声明的并且被 Button 类自动继承了。

在 Java 中实现包含方法体的接口

Kotlin 1.0 是以 Java 6 为目标设计的，其并不支持接口中的默认方法。因此它会把每个带默认方法的接口编译成一个普通接口和一个将方法体作为静态函数的类的结合体。接口中只包含声明，类中包含了以静态方法存在的所有实现。因此，如果需要在 Java 类中实现这样一个接口，必须为所有的方法，包括在 Kotlin 中有方法体的方法定义你自己的实现。

现在你已经看过 Kotlin 是怎么让你实现在接口中定义的方法的，让我们来看看故事的第二部分：重写在基类中定义的成员。

4.1.2 open、final 和 abstract 修饰符：默认为 final

正如你所知，Java 允许你创建任意类的子类并重写任意方法，除非显式地使用了 final 关键字进行标注。这通常很方便，但是也造成了一些问题。

对基类进行修改会导致子类不正确的行为，这就是所谓的*脆弱的基类问题*，因为基类代码的修改不再符合在其子类中的假设。如果类没有提供子类应该怎么实现的明确规则（哪些方法需要被重写及如何重写），当事人可能会有按基类作者预期之外的方式来重写方法的风险。因为不可能分析所有的子类，这种情况下基类是如此“脆弱”，任何修改都有可能导致子类出现预期之外的行为改变。

为了防止这种问题，作为优秀 Java 编程风格最为知名的图书之一，Joshua Bloch 的《*Effective Java*》（Addison-Wesley，2008）建议你“要么为继承做好设计并记录文档，要么禁止这么做”。这意味着所有没有特别需要在子类中被重写的类和方法应该被显式地标注为 final。

Kotlin 采用了同样的哲学思想。Java 的类和方法默认是 open 的，而 Kotlin 中默认都是 final 的。

如果你想允许创建一个类的子类，需要使用 open 修饰符来标示这个类。此外，需要给每一个可以被重写的属性或方法添加 open 修饰符。

代码清单 4.6 声明一个带一个 open 方法的 open 类

```
open class RichButton : Clickable {
    fun disable() {}
    open fun animate() {}
    override fun click() {}
}
```

这个类是 open 的：其他类可以继承它

这个函数是 final 的：不能在子类中重写它

这个函数是 open 的：可以在子类中重写它

这个函数重写了一个 open 函数并且它本身同样是 open 的

注意，如果你重写了一个基类或者接口的成员，重写了的成员同样默认是 open 的。如果你想改变这一行为，阻止你的类的子类重写你的实现，可以显式地将重写的成员标注为 final。

代码清单 4.7 禁止重写

```
open class RichButton : Clickable {
    final override fun click() {}
}
```

在这里“final”并没有被删减是因为没有“final”的“override”意味着是 open 的

open 类和智能转换

类默认为 final 带来了一个重要的好处就是这使得在大量场景中的智能转换成为可能。正如我们在 2.3.5 节中提到的，智能转换只能在进行类型检查后没有改变过的变量上起作用。对于一个类来说，这意味着智能转换只能在 val 类型并且没有自定义访问器的类属性上使用。这个前提意味着属性必须是 final 的，否则如果一个子类可以重写属性并定义一个自定义的访问器将会打破智能转换的关键前提。因为属性默认是 final 的，可以在大多数属性上不加思考地使用智能转换，这提高了你的代码表现力。

在 Kotlin 中，同 Java 一样，可以将一个类声明为 abstract 的，这种类不能被实例化。一个抽象类通常包含一些没有实现并且必须在子类重写的抽象成员。抽象成员始终是 open 的，所以不需要显式地使用 open 修饰符。接下来就是示例。

代码清单 4.8 声明一个抽象类

```
abstract class Animated {

    abstract fun animate()

    open fun stopAnimating() {
    }

    fun animateTwice() {
    }
}
```

这个类是抽象的：不能创建它的实例

这个函数是抽象的：它没有实现必须被子类重写

抽象类中的非抽象函数并不是默认 open 的，但是可以标注为 open 的

表 4.1 列出了 Kotlin 中的访问修饰符。表中的评注适用于类中的修饰符；在接口中，不能使用 final、open 或者是 abstract。接口中的成员始终是 open 的，不能将其声明为 final 的。如果它没有函数体它就是 abstract 的，但是这个关键字并不是必需的。

表 4.1 类中访问修饰符的意义

修饰符	相关成员	评注
final	不能被重写	类中成员默认使用
open	可以被重写	需要明确地表明
abstract	必须被重写	只能在抽象类中使用；抽象成员不能有实现
override	重写父类或接口中的成员	如果没有使用 final 表明，重写的成员默认是开放的

讨论完控制继承的修饰符，现在让我们转向另一种类型的修饰符：可见性修饰符。

4.1.3 可见性修饰符：默认为 public

可见性修饰符帮助控制对代码库中声明的访问。通过限制类中实现细节的可见性，可以确保在修改它们时避免破坏依赖这个类的代码的风险。

总的来说，Kotlin 中的可见性修饰符与 Java 中的类似。同样可以使用 public、protected 和 private 修饰符。但是默认的可见性是不一样的：如果省略了修饰符，声明就是 public 的。

Java 中的默认可见性——包私有，在 Kotlin 中并没有使用。Kotlin 只把包作为在命名空间里组织代码的一种方式使用，并没有将其用作可见性控制。

作为替代方案，Kotlin 提供了一个新的修饰符，internal，表示“只在模块内部可见”。一个模块就是一组一起编译的 Kotlin 文件。这有可能是一个 Intellij IDEA 模块、一个 Eclipse 项目、一个 Maven 或 Gradle 项目或者一组使用调用 Ant 任务进行编译的文件。

internal 可见性的优势在于它提供了对模块实现细节的真正封装。使用 Java 时，这种封装很容易被破坏，因为外部代码可以将类定义到与你代码相同的包中，从而得到访问你的包私有声明的权限。

另一个区别就是 Kotlin 允许在顶层声明中使用 private 可见性，包括类、函数和属性。这些声明就会只在声明它们的文件中可见。这就是另外一种隐藏子系统实现细节的非常有用的方式。表 4.2 总结了所有的可见性修饰符。

表 4.2 Kotlin 的可见性修饰符

修饰符	类成员	顶层声明
public（默认）	所有地方可见	所有地方可见
internal	模块中可见	模块中可见
protected	子类中可见	—
private	类中可见	文件中可见

让我们来看个例子。giveSpeech 函数的每一行都试图违反可见性规则。在编译时就会发生错误。

```
internal open class TalkativeButton : Focusable {
    private fun yell() = println("Hey!")
    protected fun whisper() = println("Let's talk!")
}

fun TalkativeButton.giveSpeech() {
    yell()

    whisper()
}
```

错误：“public”成员暴露了其“internal”接收者类型 TalkativeButton

错误：不能访问“yell”：它在“TalkativeButton”中是“private”的

错误：不能访问“whisper”：它在“TalkativeButton”中是“protected”的

Kotlin 禁止从 public 函数 giveSpeech 去引用低可见的类型 TalkativeButton（这个例子中是 internal）。一个通用的规则是：类的基础类型和类型参数列表中用到的所有类，或者函数的签名都有与这个类或者函数本身相同的可见性。这个规则可以确保你在需要调用函数或者继承一个类时能够始终访问到所有的类型。要解决上面例子中的问题，既可以把函数改为 internal 的，也可以把类改成 public 的。

注意，protected 修饰符在 Java 和 Kotlin 中不同的行为。在 Java 中，可以从同一个包中访问一个 protected 的成员，但是 Kotlin 不允许这样做。在 Kotlin 中可见性规则非常简单，protected 成员只在类和它的子类中可见。同样还要注意的是类的扩展函数不能访问它的 private 和 protected 成员。

Kotlin 的可见性修饰符和 Java

Kotlin 中的 public、protected 和 private 修饰符在编译成 Java 字节码时会被保留。你从 Java 代码使用这些 Kotlin 声明时就如同他们在 Java 中声明了同样的可见性。唯一的例外是 private 类：在这种情况下它会被编译成包私有声明（在 Java 中你不能把类声明为 private）。

但是，你可能会问，internal 修饰符将会发生什么？Java 中并没有直接与之类似的东西。包私有可见性是一个完全不同的东西：一个模块通常会由多个包组成，并且不同模块可能会包含来自同一个包的声明。因此 internal 修饰符在字节码中会变成 public。

这些 Kotlin 声明和它们 Java 翻版（或者说它们的字节码呈现）的对应关系解释了为什么有时你能从 Java 代码中访问一些你不能从 Kotlin 中访问的东西。例如，可以从另一个模块的 Java 代码中访问 internal 类或顶层声明，抑或从同一个包的 Java 代码中访问一个 protected 的成员（与你在 Java 中做的相似）。

但是注意类的 internal 成员的名字会被破坏。从技术上讲，internal 成员也是可以在 Java 中使用的，但是它们在 Java 代码中看起来很难看。当你从另一个模块继承类时，可以帮助避免在重写时出现出乎意料的冲突，并且避免意外使用 internal 类。

另一个 Kotlin 与 Java 之间可见性规则的区别就是在 Kotlin 中一个外部类不能看到其内部（或者嵌套）类中的 private 成员。让我们接下来讨论 Kotlin 中的内部和嵌套类并看一个例子。

4.1.4 内部类和嵌套类：默认是嵌套类

像 Java 一样，在 Kotlin 中可以在另一个类中声明一个类。这样做在封装一个辅助类或者把一些代码放到靠近它被使用的地方时非常有用。区别是 Kotlin 的嵌套类

不能访问外部类的实例，除非你特别地做出了要求。让我们通过一个例子来展示为什么这很重要。

设想一下你想定义一个 View 元素，它的状态是可以序列化的。想要序列化一个视图可能并不容易，但是可以把所有需要的数据复制到另一个辅助类中去。你声明了 State 接口去实现 Serializable。View 接口声明了可以用来保存视图状态的 getCurrentState 和 restoreState 方法。

代码清单 4.9 声明一个包含可序列化状态的视图

```
interface State: Serializable

interface View {
    fun getCurrentState(): State
    fun restoreState(state: State) {}
}
```

可以方便地定义一个保存按钮状态的 Button 类。让我们来看看在 Java 中是怎么做的（相似的 Kotlin 代码一会儿展示）。

代码清单 4.10 用带内部类的 Java 代码来实现 View

```
/* Java */
public class Button implements View {
    @Override
    public State getCurrentState() {
        return new ButtonState();
    }

    @Override
    public void restoreState(State state) { /*...*/ }

    public class ButtonState implements State { /*...*/ }
}
```

你定义了实现 State 接口的 ButtonState 类，并且持有 Button 的特定信息。在 getCurrentState 方法中，你创建了这个类的一个新的实例。在真实情况下，你需要使用所有需要的数据来初始化 ButtonState。

这个代码有什么问题？为什么你会得到一个 java.io.NotSerializable Exception: Button 异常，如果你试图序列化声明的按钮的状态？这最开始可能会看起来奇怪：你序列化的变量是 ButtonState 类型的 state，并不是 Button 类型。

当你想起来这是在 Java 中时所有的事情都清楚了，当你在另一个类中声明一个类时，它会默认变成内部类。这个例子中的 ButtonState 类隐式地存储了它的外

部 Button 类的引用。这就解释了为什么 ButtonState 不能被序列化：Button 不是可序列化的，并且它的引用破坏了 ButtonState 的序列化。

要修复这个问题，你需要声明 ButtonState 类是 static 的。将一个嵌套类声明为 static 会从这个类中删除包围它的类的隐式引用。

在 Kotlin 中，内部类的默认行为与我们刚刚描述的是相反的，就像接下来的例子。

代码清单 4.11 在 Kotlin 中使用嵌套类来实现 View

```
class Button : View {
    override fun getCurrentState(): State = ButtonState()

    override fun restoreState(state: State) { /*...*/ }

    class ButtonState : State { /*...*/ }
}
```

这个类与 Java 中的静态嵌套类类似

Kotlin 中没有显式修饰符的嵌套类与 Java 中的 static 嵌套类是一样的。要把它变成一个内部类来持有一个外部类的引用的话需要使用 inner 修饰符。表 4.3 描述了 Java 和 Kotlin 在这个行为上的不同之处；嵌套类和内部类的区别在图 4.1 中呈现。

表 4.3 嵌套类和内部类在 Java 与 Kotlin 中的对应关系

类 A 在另一个类 B 中声明	在 Java 中	在 Kotlin 中
嵌套类（不存储外部类的引用）	static class A	class A
内部类（存储外部类的引用）	class A	inner class A

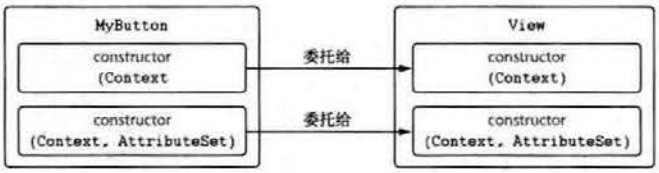


图 4.1 嵌套类不持有外部类的引用，而内部类持有

在 Kotlin 中引用外部类实例的语法也与 Java 不同。需要使用 this@Outer 从 Inner 类去访问 Outer 类：

```
class Outer {
    inner class Inner {
        fun getOuterReference(): Outer = this@Outer
    }
}
```

到此，你已经学习了 Java 和 Kotlin 中内部类和嵌套类的区别。现在让我们来讨论另一个可能在 Kotlin 中很有用的嵌套类使用场景：创建一个包含有限数量的类的继承结构。

4.1.5 密封类：定义受限的类继承结构

回想一下 2.3.5 节中关于继承结构表达式的例子。父类 Expr 有两个子类：表示数字的 Num，以及表示两个表达式之和的 Sum。在 when 表达式中处理所有可能的子类固然很方便，但是必须提供一个 else 分支来处理没有任何其他分支能匹配的情况：

代码清单 4.12 作为接口实现的表达式

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr

fun eval(e: Expr): Int =
    when (e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        else ->
            throw IllegalArgumentException("Unknown expression")
    }
```

必须检查“else”分支

当使用 when 结构来执行表达式的时候，Kotlin 编译器会强制检查默认选项。在这个例子中，不能返回一个有意义的值，所以直接抛出一个异常。

总是不得不添加一个默认分支很不方便。更重要的是，如果你添加了一个新的子类，编译器并不能发现有地方改变了。如果你忘记了添加一个新分支，就会选择默认的选项，这有可能导致潜在的 bug。

Kotlin 为这个问题提供了一个解决方案：sealed 类。为父类添加一个 sealed 修饰符，对可能创建的子类做出严格的限制。所有的直接子类必须嵌套在父类中。

代码清单 4.13 作为密封类的表达式

```
sealed class Expr {
    class Num(val value: Int) : Expr()
    class Sum(val left: Expr, val right: Expr) : Expr()
}

fun eval(e: Expr): Int =
    when (e) {
        is Expr.Num -> e.value
        is Expr.Sum -> eval(e.right) + eval(e.left)
    }
```

将基类标记为密封的……

……将所有可能的类作为嵌套类列出

“when”表达式涵盖了所有可能的情况，所以不再需要“else”分支

如果你在 when 表达式中处理所有 sealed 类的子类，你就不再需要提供默认分支。注意，sealed 修饰符隐含的这个类是一个 open 类，你不再需要显式地添加 open 修饰符。密封类的行为如图 4.2 所示。

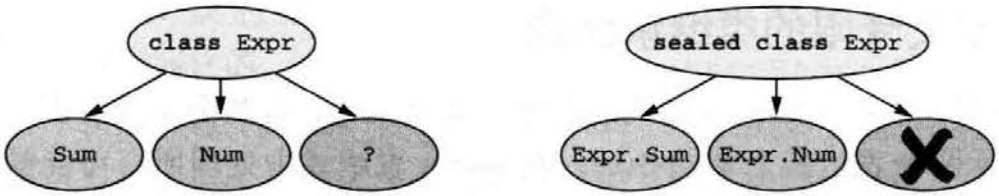


图 4.2 密封类不能在类外部拥有子类

当你在 when 中使用 sealed 类并且添加一个新的子类的时候，有返回值的 when 表达式会导致编译失败，它会告诉你哪里的代码必须要修改。

在这种情况下，Expr 类有一个只能在类内部调用的 private 构造方法。你也不能声明一个 sealed 接口。为什么？如果你能这样做，Kotlin 编译器不能保证任何人都不能在 Java 代码中实现这个接口。

> **注意** 在 Kotlin 1.0 中，sealed 功能是相当严格的。例如，所有的子类必须是嵌套的，并且子类不能创建为 data 类（data 类会在本章后面部分提到）。Kotlin 1.1 解除了这些限制并允许在同一文件的任何位置定义 sealed 类的子类。

你应该还记得，在 Kotlin 中，冒号既可以用来继承一个类也可以实现一个接口。那我们再来仔细看看一个子类的声明：

```
class Num(val value: Int) : Expr()
```

这个简单的例子应该已经很清楚了，除了 Expr() 中类名后面的括号的含义。我们会在下一节讲到它，下一节将包含 Kotlin 中类初始化相关的内容。

4.2 声明一个带非默认构造方法或属性的类

正如你所知，在 Java 中一个类可以声明一个或多个构造方法。Kotlin 也是类似的，只是做出了一点修改：区分了主构造方法（通常是主要而简洁的初始化类的方法，并且在类体外部声明）和从构造方法（在类体内部声明）。同样也允许在初始化语句块中添加额外的初始化逻辑。首先我们会关注声明主构造方法和初始化语句块的语法，然后我们会阐明如何声明多个构造方法。最后，我们还会再谈谈属性。

4.2.1 初始化类：主构造方法和初始化语句块

在第 2 章中已经见过了怎么声明一个简单类：

```
class User(val nickname: String)
```

通常来讲，类的所有声明都在花括号中。你可能会好奇为什么这个类没有花括

号而是只包含了声明在括号中。这段被括号围起来的语句块就叫作*主构造方法*。它主要有两个目的：表明构造方法的参数，以及定义使用这些参数初始化的属性。让我们揭示它是怎么工作的，并看看你可以编写的用来完成同样事情的最明确的代码会是什么样子：

```
class User constructor(_nickname: String) {        <-- 带一个参数的主构造方法
    val nickname: String

    init {                                          <-- 初始化语句块
        nickname = _nickname
    }
}
```

在这个例子中，可以看到两个新的 Kotlin 关键字：constructor 和 init。constructor 关键字用来开始一个主构造方法或从构造方法的声明。init 关键字用来引入一个*初始化语句块*。这种语句块包含了在类被创建时执行的代码，并会与主构造方法一起使用。因为主构造方法有语法限制，不能包含初始化代码，这就是为什么要使用初始化语句块的原因。如果你愿意，也可以在一个类中声明多个初始化语句块。

构造方法参数 _nickname 中的下画线用来区分属性的名字和构造方法参数的名字。另一个可选方案是使用同样的名字，通过 this 来消除歧义，就像 Java 中的常用做法一样：this.nickname = nickname。

在这个例子中，不需要把初始化代码放在初始化语句块中，因为它可以与 nickname 属性的声明结合。如果主构造方法没有注解或可见性修饰符，同样可以去掉 constructor 关键字。如果你这样做，会得到如下代码：

```
                                                    带一个参数的主构造方法
class User(_nickname: String) {                     <--
    val nickname = _nickname                        <-- 用参数来初始化属性
}
```

这就是声明同样的类的另一种方法。请注意如何在属性的初始化器中，以及初始化语句块中引用主构造方法的参数。

前两个例子在类体中使用 val 关键字声明了属性。如果属性用相应的构造方法参数来初始化，代码可以通过把 val 关键字加在参数前的方式来进行简化。这样可以替换类中的属性定义：

```
class User(val nickname: String)        <-- "val" 意味着相应的属性会用构造方法的参数来初始化
```

所有 User 类的声明都是等价的，但是最后一个使用了最简洁的语法。

可以像函数参数一样为构造方法参数声明一个默认值：

```
class User(val nickname: String,
           val isSubscribed: Boolean = true)
```

为构造方法参数提供一个默认值

要创建一个类的实例，只需要直接调用构造方法，不需要 new 关键字：

```
>>> val alice = User("Alice")
>>> println(alice.isSubscribed)
true
>>> val bob = User("Bob", false)
>>> println(bob.isSubscribed)
false
>>> val carol = User("Carol", isSubscribed = false)
>>> println(carol.isSubscribed)
false
```

为 isSubscribed 参数使用默认值“true”

可以按照声明顺序写明所有的参数

可以显式地为某些构造方法参数标明名称

看起来 Alice 默认订阅了邮件列表，而 Bob 仔细阅读了条款和条件后取消了默认选项。

注意 如果所有的构造方法参数都有默认值，编译器会生成一个额外的不带参数的构造方法来使用所有的默认值。这可以让 Kotlin 使用库时变得更简单，因为可以通过无参构造方法来实例化类。

如果你的类具有一个父类，主构造方法同样需要初始化父类。可以通过在基类列表的父类引用中提供父类构造方法参数的方式来做到这一点：

```
open class User(val nickname: String) { ... }

class TwitterUser(nickname: String) : User(nickname) { ... }
```

如果没有给一个类声明任何的构造方法，将会生成一个不做任何事情的默认构造方法：

```
open class Button
```

将会生成一个不带任何参数的默认构造方法

如果继承了 Button 类并且没有提供任何的构造方法，必须显式地调用父类的构造方法，即使它没有任何的参数：

```
class RadioButton: Button()
```

这就是为什么在父类名称后面还需要一个空的括号。注意与接口的区别：接口没有构造方法，所以在你实现一个接口的时候，不需要在父类型列表中它的名称后面再加上括号。

如果你想要确保你的类不被其他代码实例化，必须把构造方法标记为 private。下面就是你该怎样把主构造方法标记为 private：

```
class Secretive private constructor() {}
```

这个类有一个 private 构造方法

因为 Secretive 类只有一个 private 的构造方法，这个类外部的代码不能实例化它。在本章后面的部分，我们会讲到伴生对象，那里是调用这样的构造方法的好地方。

private 构造方法的替代方案

在 Java 中，可以通过使用 private 构造方法禁止实例化这个类来表示一个更通用的意思：这个类是一个静态实用工具成员的容器或者是单例的。Kotlin 针对这种目的具有内建的语言级别的功能。可以使用顶层函数（已经在 3.2.3 节中出现过）作为静态实用工具。要想表示单例，可以使用对象声明，将会在 4.4.1 节中见到。

在大多数真实的场景中，类的构造方法是非常简明的：它要么没有参数或者直接把参数与对应的属性关联。这就是为什么 Kotlin 有为主构造方法设计的简洁的语法：在大多数的情况下都能很好地工作。但是生活并不总是那么容易，所以 Kotlin 允许为你的类定义足够多的构造方法。让我们来看看这是怎么工作的。

4.2.2 构造方法：用不同的方式来初始化父类

通常来讲，使用多个构造方法的类在 Kotlin 代码中不如在 Java 中常见。大多数在 Java 中需要重载构造方法的场景都被 Kotlin 支持参数默认值和参数命名的语法涵盖了。

小贴士 不要声明多个从构造方法用来重载和提供参数的默认值。取而代之的是，应该直接标明默认值。

但是还是会有需要多个构造方法的情景。最常见的一种就来自于当你需要扩展一个框架类来提供多个构造方法，以便于通过不同的方式来初始化类的时候。设想一下一个在 Java 中声明的具有两个构造方法的 View 类（如果你是一个 Android 开发者也许已经知道这种定义）。Kotlin 中相似的声明如下：

```
open class View {
    constructor(ctx: Context) {
        // some code
    }
    constructor(ctx: Context, attr: AttributeSet) {
        // some code
    }
}
```

从构造方法

这个类没有声明一个主构造方法（就如你所见，因为在类头部的类名后面并没有括号），但是它声明了两个从构造方法。从构造方法使用 constructor 关键字引出。只要需要，可以声明任意多个从构造方法。

如果你想扩展这个类，可以声明同样的构造方法：

```
class MyButton : View {
    constructor(ctx: Context)
        : super(ctx) {
        // ...
    }
    constructor(ctx: Context, attr: AttributeSet)
        : super(ctx, attr) {
        // ...
    }
}
```

调用父类构造方法

这里定义了两个构造方法，它们都使用 super() 关键字调用了对应的父类构造方法。这些在图 4.3 中展示了出来，箭头显示了构造方法委托的目标。

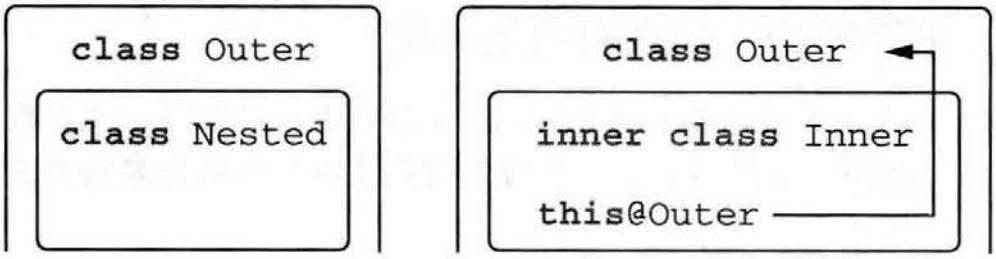


图 4.3 使用不同的父类构造方法

就像在 Java 中一样，也可以使用 this() 关键字，从一个构造方法中调用你自己的类的另一个构造方法。下面展示了这是如何使用的：

```
class MyButton : View {
    constructor(ctx: Context): this(ctx, MY_STYLE) {
        // ...
    }
    constructor(ctx: Context, attr: AttributeSet): super(ctx, attr) {
        // ...
    }
}
```

委托给这个类的另一个构造方法

可以修改 MyButton 类使得一个构造方法委托给同一个类的另一个构造方法（使用 this），为参数传入默认值，就像图 4.4 展示的那样。第二个构造方法继续调用 super()。

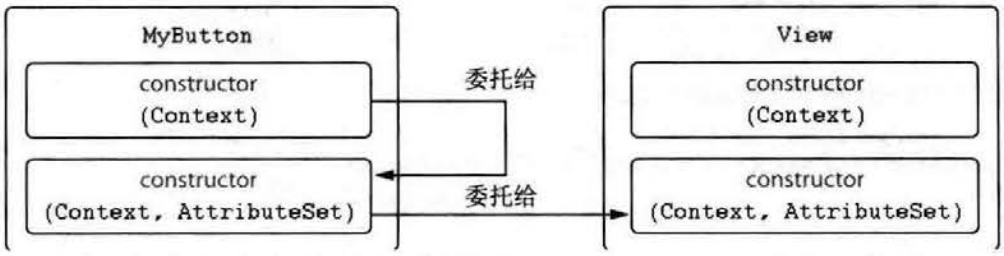


图 4.4 委托给同一个类的构造方法

如果类没有主构造方法，那么每个从构造方法必须初始化基类或者委托给另一个这样做了的构造方法。从图 4.3 来看，每个从构造方法必须以一个朝外的箭头开始并且结束于任意一个基类构造方法。

Java 的互操作性是你需要使用从构造方法的主要使用场景。但是还有另一个可能的情况：当你使用不同的参数列表，以多种方法创建类的实例时，使用不同的参数列表。我们会在 4.4.2 节中讨论一个例子。

我们讨论了如何定义非默认构造方法。现在让我们将注意力转向非默认属性。

4.2.3 实现在接口中声明的属性

在 Kotlin 中，接口可以包含抽象属性声明。这里有一个具有这样声明的接口定义的例子：

```
interface User {
    val nickname: String
}
```

这意味着实现 User 接口的类需要提供一个取得 nickname 值的方式。接口并没有说明这个值应该存储到一个支持字段还是通过 getter 来获取。接口本身并不包含任何状态，因此只有实现这个接口的类在需要的情况下会存储这个值。

让我们来看看这个类的一些可能的实现：PrivateUser，表示只填写了昵称的用户；SubscribingUser，表示显然被迫提供 email 进行注册的用户；FacebookUser，表示轻率地共享了他们的 Facebook 账号的用户。所有的这些类都以不同的方式实现了接口中的抽象属性。

代码清单 4.14 实现一个接口属性

```
class PrivateUser(override val nickname: String) : User
class SubscribingUser(val email: String) : User {
```

主构造方法属性

```
    override val nickname: String
        get() = email.substringBefore('@') )
}

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId)
}

>>> println(PrivateUser("test@kotlinlang.org").nickname)
test@kotlinlang.org
>>> println(SubscribingUser("test@kotlinlang.org").nickname)
test
```

自定义getter

属性初始化

对于 PrivateUser 来说，你使用了简洁的语法直接在主构造方法中声明了一个属性。这个属性实现了来自于 User 的抽象属性，所以你将其标记为 override。

对于 SubscribingUser 来说，nickname 属性通过一个自定义 getter 实现。这个属性没有一个支持字段来存储它的值，它只有一个 getter 在每次调用时从 email 中得到昵称。

对于 FacebookUser 来说，在初始化时将 nickname 属性与值关联。你使用了被认为可以通过账号 ID 返回 Facebook 用户名称的 getFacebookName 函数（假设这个函数是在别的地方定义的）。这个函数开销巨大：它需要与 Facebook 建立连接来获取想要的数据。这也就是为什么你决定只在初始化阶段调用一次的原因。

请注意 nickname 在 SubscribingUser 和 FacebookUser 中的不同实现。即使它们看起来很相似，第一个属性有一个自定义 getter 在每次访问时计算 substringBefore，然而 FacebookUser 中的属性有一个支持字段来存储在类初始化时计算得到的数据。

除了抽象属性声明外，接口还可以包含具有 getter 和 setter 的属性，只要它们没有引用一个支持字段（支持字段需要在接口中存储状态，而这是不允许的）。让我们来看一个例子：

```
interface User {
    val email: String
    val nickname: String
        get() = email.substringBefore('@')
}
```

属性没有支持字段：结果值在每次访问时通过计算得到

这个接口包含抽象属性 email，同时 nickname 属性有一个自定义的 getter。第一个属性必须在子类中重写，而第二个是可以被继承的。

不像在接口中实现的属性，在类中实现的属性具有对支持字段的完全访问权限。让我们看看怎样从访问器中引用它们。

4.2.4 通过 getter 或 setter 访问支持字段

你已经看过一些关于两种属性的例子：存储值的属性和具有自定义访问器在每次访问时计算值的属性。现在让我们来看看怎么结合这两种来实现一个既可以存储值又可以在值被访问和修改时提供额外逻辑的属性。要支持这种情况，需要能够从属性的访问器中访问它的支持字段。

我们假设你想在任何对存储在属性中的数据进行修改时输出日志，你声明了一个可变属性并且在每次 setter 访问时执行额外的代码。

代码清单 4.15 在 setter 中访问支持字段

```
class User(val name: String) {
    var address: String = "unspecified"
        set(value: String) {
            println("""
                Address was changed for $name:
                "$field" -> "$value".""".trimIndent())
            field = value
        }
}
```

读取支持字段的值

更新支持字段的值

```
>>> val user = User("Alice")
>>> user.address = "Elsenheimerstrasse 47, 80687 Muenchen"
Address was changed for Alice:
"unspecified" -> "Elsenheimerstrasse 47, 80687 Muenchen".
```

可以像平常一样通过使用 `user.address = "new value"` 来修改一个属性的值，这其实在底层调用了 setter。在这个例子中，setter 被重新定义了，所以额外的输出日志的代码被执行了（简单起见，这里直接将其打印了出来）。

在 setter 的函数体中，使用了特殊的标识符 `field` 来访问支持字段的值。在 getter 中，只能读取值；而在 setter 中，既能读取它也能修改它。

注意，可以只重定义可变属性的一个访问器。代码清单 4.15 中的 getter 是默认的并且只返回字段的值，所以没有必要重定义它。

你可能会好奇，有支持字段的属性和没有的有什么区别。访问属性的方式不依赖于它是否含有支持字段。如果你显式地引用或者使用默认的访问器实现，编译器会为属性生成支持字段。如果你提供了一个自定义的访问器实现并且没有使用 `field`（如果属性是 `val` 类型，就是 getter；而如果是可变属性，则是两个访问器），支持字段将不会被呈现出来。

有时候不需要修改访问器的默认实现，但是需要修改它的可见性。让我们来看看怎样做到这一点。

4.2.5 修改访问器的可见性

访问器的可见性默认与属性的可见性相同。但是如果需要可以通过在 get 和 set 关键字前放置可见性修饰符的方式来修改它。要想知道怎么使用它，我们来看一个例子。

代码清单 4.16 声明一个具有 private setter 的属性

```
class LengthCounter {
    var counter: Int = 0
        private set                    <-- 不能在类外部修改这个属性
    fun addWord(word: String) {
        counter += word.length
    }
}
```

这个类用来计算单词加在一起的总长度。持有总长度的属性是 public 的，因为它是这个类提供给客户的 API 的一部分。但是你需要确保它只能在类中被修改，否则外部代码有可能会修改它并存储一个不正确的值。因此，你让编译器生成一个默认可见性的 getter 方法，并且将 setter 的可见性修改为 private。

下面就是如何使用这个类：

```
>>> val lengthCounter = LengthCounter()
>>> lengthCounter.addWord("Hi!")
>>> println(lengthCounter.counter)
3
```

创建了 LengthCounter 的一个实例，并且添加了一个长度为 3 的词语“Hi!”。现在 counter 属性存储的是 3。

接下来关于属性的更多话题

在本书接下来的部分，我们还会继续关于属性的讨论。这里有一些参考话题：

- 在非空属性上使用的 lateinit 修饰符表明这个属性会将初始化推迟到构造方法被调用过后，这是一些框架的常见用法。这些功能将在第 6 章中涵盖到。
- 惰性初始化属性，作为更通用的**委托属性**的一部分，将会在第 7 章中涵盖到。为了与 Java 框架的兼容性，可以在 Kotlin 中使用注解来模仿 Java 的功能。例如，属性上的 @JvmField 注解用来在没有访问器的情况下暴露一个 public 的字段。你会在第 10 章学到更多关于注解的内容。
- const 修饰符使得使用注解更加方便，并且允许使用基本数据类型或者 String 的属性作为注解参数。第 10 章会给出相关的细节。

关于在 Kotlin 中书写非默认构造方法和属性的讨论到此就告一段落了。接下来，你会看到如何通过使用 data 类的概念以更友好的方式创建值对象类。

4.3 编译器生成的方法：数据类和类委托

Java 平台定义了一些需要在许多类中呈现的方法，并且通常是以一种很机械的方式实现的，譬如 equals、hashCode 及 toString。幸运的是，Java IDE 可以将这些方法的生成自动化，所以你通常不需要手动书写它们。但是这种情况下，你的代码库包含了样板代码。Kotlin 的编译器就领先一步了：它能将这些呆板的代码生成放到幕后，并不会因为自动生成的结果导致源代码文件变得混乱。

你已经看过了默认类构造方法和属性访问器是怎样工作的，让我们来看看更多 Kotlin 编译器生成典型方法的例子，它们对简单数据类很有用并且极大简化了类委托模式。

4.3.1 通用对象方法

就像 Java 中的情况一样，所有的 Kotlin 类也有许多也许你想重写的方法：toString、equals 和 hashCode。让我们来看看这些方法是什么，Kotlin 又是怎么帮助你自动生成它们的实现的。作为起点，你会看到一个简单的用来存储客户名字和邮编的 Client 类。

代码清单 4.17 Client 类的最初声明

```
class Client(val name: String, val postalCode: Int)
```

让我们看看类实例是怎么作为字符串表示的。

字符串表示：toString()

Kotlin 中的所有类同 Java 一样，提供了一种方式来获取类对象的字符串表示形式，虽然你也能在其他的上下文中使用这个功能，但是这还是主要用在调试和日志输出中。默认来说，一个对象的字符串表示形如 Client@5e9f23b4，这并不十分有用。要想改变它，需要重写 toString 方法。

代码清单 4.18 为 Client 实现 toString()

```
class Client(val name: String, val postalCode: Int) {
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

现在，一个客户的表示就看起来是这个样子：

```
>>> val client1 = Client("Alice", 342562)
>>> println(client1)
Client(name=Alice, postalCode=342562)
```

更表意了，不是吗？

对象相等性：equals()

所有关于 Client 类的计算都发生在其外部，这个类只用来存储数据。这意味着简单和透明。尽管如此，也许还是会有一些针对这种类行为的需求。例如，假设想要将包含相同数据的对象视为相等：

```
>>> val client1 = Client("Alice", 342562)
>>> val client2 = Client("Alice", 342562)
>>> println(client1 == client2)
false
```

在 Kotlin 中，== 检查对象是否相等，而不是比较引用。这里会编译成调用“equals”

正如你所见，对象并不相等。这意味着必须为 Client 类重写 equals。

== 表示相等性

在 Java 中，可以使用 == 运算符来比较基本数据类型和引用类型。如果应用在基本数据类型上，Java 的 == 比较的是值，然而在引用类型上 == 比较的是引用。因此，在 Java 中，众所周知的实践是总是调用 equals，如果忘记了这样做当然也会导致众所周知的问题。

在 Kotlin 中，== 运算符是比较两个对象的默认方式：本质上说它就是通过调用 equals 来比较两个值的。因此，如果 equals 在你的类中被重写了，你能够很安全地使用 == 来比较实例。要想进行引用比较，可以使用 === 运算符，这与 Java 中的 == 比较对象引用的效果一模一样。

让我们看看修改后的 Client 类。

代码清单 4.19 为 Client 类实现 equals()

检查“other”是不是一个 Client

检查对应的属性是否相等

“Any”是 java.lang.Object 的模拟：Kotlin 中所有类的父类。可空类型“Any?”意味着“other”是可以为空的

```
class Client(val name: String, val postalCode: Int) {
    override fun equals(other: Any?): Boolean {
        if (other == null || other !is Client)
            return false
        return name == other.name &&
                postalCode == other.postalCode
    }
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

提醒你一下，Kotlin 中的 is 检查是 Java 中 instanceof 的模拟，用来检查一个值是否为一个指定的类型。就像 !in 运算符是 in 检查的逆运算一样（已经在 2.4.4 节中对这两种运算符做过讨论），!is 运算符表示 is 检查的非运算。这些运算符让你的代码更易读。在第 6 章中会深入了解更多关于可空类型的细节，以及探讨为什么条件语句 other==null||other !is Client 可以简化为 other !is Client。

因为在 Kotlin 中 override 修饰符是强制使用的，你会在意外书写 fun equals(other: Client) 时得到保护，这样会添加一个新的方法而不是重写 equals。在你重写了 equals 后，也许会期望有着相同属性值的客户会是相等的。的确，前面例子中的相等性检查 client1 == client2 现在会返回 true 了。但是如果你想用客户来做更多复杂的事情就办不到了。通常的面试问题就是："什么东西被破坏了，问题在于哪儿？"你可能会说问题就是 hashCode 缺失了。事实确实如此，现在我们来谈谈为什么这十分重要。

Hash 容器：hashCode()

hashCode 方法通常与 equals 一起被重写。这一小节将会解释为什么这样做。

让我们创建有一个元素的 set：一个名为 Alice 的客户。接着，创建一个新的包含相同数据的 Client 实例并检查它是否包含在 set 中。你期望检查会返回 true，因为这两个实例是相等的，但事实上返回的是 false:

```
>>> val processed = hashSetOf(Client("Alice", 342562))
>>> println(processed.contains(Client("Alice", 342562)))
false
```

原因就是 Client 类缺少了 hashCode 方法。因此它违反了通用的 hashCode 契约：如果两个对象相等，它们必须有着相同的 hash 值。processed set 是一个 HashSet。在 HashSet 中值是以一种优化过的方式来比较的：首先比较它们的 hash 值，然后只有当它们相等时才会去比较真正的值。前一个例子中 Client 类的两个不同的实例有着不同的 hash 值，所以 set 认为它不包含第二个对象，即使 equals 会返回 true。因此，如果不遵循规则，HashSet 不能在这样的对象上正常工作。

要修复这个问题，可以向类中添加 hashCode 的实现。

代码清单 4.20 为 Client 实现 hashCode()

```
class Client(val name: String, val postalCode: Int) {
    ...
    override fun hashCode(): Int = name.hashCode() * 31 + postalCode
}
```

现在这个类在所有的情境下都能按预期来工作了——但是请注意有多少代码你需要书写。幸运的是，Kotlin 编译器能够帮助你自动生成这些方法。让我们看看怎样让编译器做到这一点。

4.3.2 数据类：自动生成通用方法的实现

如果想要你的类是一个方便的数据容器，你需要重写这些方法：toString、equals 和 hashCode。通常来说这些方法的实现十分简单，并且像 Intellij IDEA 这样的 IDE 能够帮助你自动地生成它们，并确保它们的实现是正确且一致的。

好消息是，在 Kotlin 中你不必再去生成这些方法了。如果为你的类添加 data 修饰符，必要的方法将会自动生成好。

代码清单 4.21 数据类 Client

```
data class Client(val name: String, val postalCode: Int)
```

很简单，是吧？现在就得到了一个重写了所有标准 Java 方法的类：

- equals 用来比较实例
- hashCode 用来作为例如 HashMap 这种基于哈希容器的键
- toString 用来为类生成按声明顺序排列的所有字段的字符串表达形式

equals 和 hashCode 方法会将所有在主构造方法中声明的属性纳入考虑。生成的 equals 法会检测所有的属性的值是否相等。hashCode 方法会返回一个根据所有属性生成的哈希值。请注意没有在主构造方法中声明的属性将不会加入到相等性检查和哈希值计算中去。

这些并不是为 data 类生成有用方法的完整列表。下一小节会揭示更多，而 7.4 节会介绍剩下的其他部分。

数据类和不可变性：copy() 方法

请注意，虽然数据类的属性并没有要求是 val——同样可以使用 var——但还是强烈推荐只使用只读属性，让数据类的实例不可变。如果你想使用这样的实例作为 HashMap 或者类似容器的键，这会是必需的要求，因为如果不这样，被用作键的对象在加入容器后被修改了，容器可能会进入一种无效的状态。不可变对象同样更容易理解，特别是在多线程代码中：一旦一个对象被创建出来，它会一直保持初始状态，也不用担心在你的代码工作时其他线程修改了对象的值。

为了让使用不可变对象的数据类变得更容易，Kotlin 编译器为它们多生成了一个方法：一个允许 *copy* 类的实例的方法，并在 copy 的同时修改某些属性的值。创

建副本通常是修改实例的好选择：副本有着单独的生命周期而且不会影响代码中引用原始实例的位置。下面就是手动实现 copy 方法后看起来的样子：

```
class Client(val name: String, val postalCode: Int) {
    ...
    fun copy(name: String = this.name,
             postalCode: Int = this.postalCode) =
        Client(name, postalCode)
}
```

这里就是 copy 方法是怎样使用的：

```
>>> val bob = Client("Bob", 973293)
>>> println(bob.copy(postalCode = 382555))
Client(name=Bob, postalCode=382555)
```

现在你已经看过 data 修饰符是如何让值对象类的使用更方便的，现在让我们探讨一下其他能够让你避免 IDE 生成的样板代码的 Kotlin 功能：类委托。

4.3.3 类委托：使用“by”关键字

设计大型面向对象系统的一个常见问题就是由继承的实现导致的脆弱性。当你扩展一个类并重写某些方法时，你的代码就变得依赖你自己继承的那个类的实现细节了。当系统不断演进并且基类的实现被修改或者新方法被添加进去时，你做出的关于类行为的假设会失效，所以你的代码也许最后就以不正确的行为而告终。

Kotlin 的设计就识别了这样的问题，并默认将类视作 final 的。这确保了只有那些设计成可扩展的类可以被继承。当使用这样的类时，你会看见它是开放的，就会注意这些修改需要与派生类兼容。

但是你常常需要向其他类添加一些行为，即使它并没有被设计为可扩展的。一个常用的实现方式以装饰器模式闻名。这种模式的本质就是创建一个新类，实现与原始类一样的接口并将原来的类的实例作为一个字段保存。与原始类拥有同样行为的方法不用被修改，只需要直接转发到原始类的实例。

这种方式的一个缺点是需要相当多的样板代码（像 Intellij IDEA 一样的众多 IDE 都有专门生成这样代码的功能）。例如，下面就是你需要多少代码来实现一个简单得如 Collection 的接口的装饰器，即使你不需要修改任何的行为：

```
class DelegatingCollection<T> : Collection<T> {
    private val innerList = arrayListOf<T>()

    override val size: Int get() = innerList.size
    override fun isEmpty(): Boolean = innerList.isEmpty()
    override fun contains(element: T): Boolean = innerList.contains(element)
    override fun iterator(): Iterator<T> = innerList.iterator()
```

```
    override fun containsAll(elements: Collection<T>): Boolean =
            innerList.containsAll(elements)
}
```

好消息是 Kotlin 将委托作为一个语言级别的功能做了头等支持。无论什么时候实现一个接口，你都可以使用 by 关键字将接口的实现委托到另一个对象。下面就是怎样通过推荐的方式来重写前面的例子：

```
class DelegatingCollection<T>(
        innerList: Collection<T> = ArrayList<T>()
) : Collection<T> by innerList {}
```

类中所有的方法实现都消失了。编译器会生成它们，并且实现与 DelegatingCollection 的例子是相似的。因为代码中没有太多有意思的内容，所以当编译器能够自动为你做同样事情的时候没有必要去手写这些代码。

现在，当你需要修改某些方法的行为时，你可以重写它们，这样你的方法就会被调用而不是使用生成的方法。可以保留感到满意的委托给内部的实例中的默认实现。

让我们来看看怎样使用这种技术来实现一个集合，它可以计算向它添加元素的尝试次数。例如，你在执行某种去重操作，可以使用这样的集合，通过比较添加元素的尝试次数和集合的最终大小来评判这种处理的效率。

代码清单 4.22 使用类委托

```
class CountingSet<T>(
        val innerSet: MutableCollection<T> = HashSet<T>()
) : MutableCollection<T> by innerSet {                    <-- 将 MutableCollection 的实现委托给 innerSet

    var objectsAdded = 0

    override fun add(element: T): Boolean {               <-- 不使用委托，提供一个不同的实现
        objectsAdded++
        return innerSet.add(element)
    }

    override fun addAll(c: Collection<T>): Boolean {
        objectsAdded += c.size
        return innerSet.addAll(c)
    }
}

>>> val cset = CountingSet<Int>()
>>> cset.addAll(listOf(1, 1, 2))
>>> println("${cset.objectsAdded} objects were added, ${cset.size} remain")
3 objects were added, 2 remain
```

正如你所见，通过重写 add 和 addAll 方法来计数，并将 MutableCollection

接口剩下的实现委托给被你包装的容器。

重要的部分是你没有对底层集合的实现方式引入任何的依赖。例如，你不用关心集合是不是通过在循环中调用 add 来实现的 addAll，或者是否是针对特定情况进行了优化的另一种实现。你对客户端代码调用你的类时会发生什么有完全的控制，并且只依赖底层集合文档列出的 API 来实现你的操作，所以你可以依赖它来继续工作。

我们已经完成了关于 Kotlin 编译器怎么生成类中有用方法的讨论，让我们继续 Kotlin 类的故事的最后一大部分：object 关键字和它发生作用的不同情况。

4.4 “object”关键字：将声明一个类与创建一个实例结合起来

Kotlin 中 object 关键字在多种情况下出现，但是它们都遵循同样的核心理念：这个关键字定义一个类并同时创建一个实例（换句话说就是一个对象）。让我们来看看使用它的不同场景：

- 对象声明是定义单例的一种方式。
- 伴生对象可以持有工厂方法和其他与这个类相关，但在调用时并不依赖类实例的方法。它们的成员可以通过类名来访问。
- 对象表达式用来替代 Java 的匿名内部类。

现在我们会详细讨论这些 Kotlin 功能。

4.4.1 对象声明：创建单例易如反掌

在面向对象系统设计中一个相当常见的情形就是只需要一个实例的类。在 Java 中，这通常通过单例模式来实现：定义一个使用 private 构造方法并且用静态字段来持有这个类仅有的实例。

Kotlin 通过使用对象声明功能为这一切提供了最高级的语言支持。对象声明将类声明与该类的单一实例声明结合到了一起。

例如，可以使用一个对象声明来表示一个组织的工资单。你也许不会有多个工资单，所以使用一个对象来表示看起来是明智的：

```
object Payroll {
    val allEmployees = arrayListOf<Person>()

    fun calculateSalary() {
        for (person in allEmployees) {
            ...
        }
```

```
    }
}
```

对象声明通过 object 关键字引入。一个对象声明可以非常高效地以一句话来定义一个类和一个该类的变量。

与类一样，一个对象声明也可以包含属性、方法、初始化语句块等的声明。唯一不允许的就是构造方法（包括主构造方法和从构造方法）。与普通类的实例不同，对象声明在定义的时候就立即创建了，不需要在代码的其他地方调用构造方法。因此，为对象声明定义一个构造方法是没有意义的。

与变量一样，对象声明允许你使用对象名加 . 字符的方式来调用方法和访问属性：

```
Payroll.allEmployees.add(Person(...))

Payroll.calculateSalary()
```

对象声明同样可以继承自类和接口。这通常在你使用的框架需要去实现一个接口，但是你的实现并不包含任何状态的时候很有用。例如，来看看 java.util.Comparator 接口。Comparator 的实现接收两个对象并返回一个整数来标示哪个对象更大。比较器通常来说都不存储任何数据，所以通常只需要一个单独的 Comparator 实例来以特定的方式比较对象。这是一个非常完美的对象声明使用场景。

作为一个完整的例子，让我们实现一个忽略大小写比较文件路径的比较器：

代码清单 4.23 使用对象来实现 Comparator

```
object CaseInsensitiveFileComparator : Comparator<File> {
    override fun compare(file1: File, file2: File): Int {
        return file1.path.compareTo(file2.path,
                ignoreCase = true)
    }
}

>>> println(CaseInsensitiveFileComparator.compare(
...     File("/User"), File("/user")))
0
```

可以在任何可以使用普通对象的地方使用单例对象。例如，可以将这个对象传入一个接收 Comparator 的函数：

```
>>> val files = listOf(File("/Z"), File("/a"))
>>> println(files.sortedWith(CaseInsensitiveFileComparator))
[/a, /Z]
```

这里使用了 sortedWith 函数，它返回一个根据特定的比较器排序过的列表。

单例和依赖注入

就像单例模式一样，在大型软件系统中使用对象声明也并不总是理想的。它们在少部分只有少量依赖或没有依赖的代码中非常好用，但在与系统中其他部分有着非常多的交互的大型组件中却不然。主要的原因就是你对对象实例化没有任何控制，并且不能通过构造方法指定特定的参数。

这意味着你不能在单元测试或软件系统的不同配置中替换掉对象自身的实现，或对象依赖的其他类。如果你需要那样的能力，你需要将依赖注入框架（譬如 Guice, https://github.com/google/guice）与普通的 Kotlin 类一起使用，就像在 Java 中那样。

同样可以在类中声明对象。这样的对象同样只有一个单一实例；它们在每个容器类的实例中并不具有不同的实例。例如，在类中放置一个用来比较特定对象的比较器是合乎逻辑的。

代码清单 4.24 使用嵌套类实现 Comparator

```
data class Person(val name: String) {
    object NameComparator : Comparator<Person> {
        override fun compare(p1: Person, p2: Person): Int =
            p1.name.compareTo(p2.name)
    }
}

>>> val persons = listOf(Person("Bob"), Person("Alice"))
>>> println(persons.sortedWith(Person.NameComparator))
[Person(name=Alice), Person(name=Bob)]
```

在 Java 中使用 Kotlin 对象

Kotlin 中的对象声明被编译成了通过静态字段来持有它的单一实例的类，这个字段名字始终都是 INSTANCE。如果在 Java 中实现单例模式，你也许也会顺手做同样的事。因此，要从 Java 代码使用 Kotlin 对象，可以通过访问静态的 INSTANCE 字段：

```
/* Java */
CaseInsensitiveFileComparator.INSTANCE.compare(file1, file2);
```

在这个例子中，INSTANCE 字段的类型是 CaseInsensitiveFileComparator。

接下来让我们来看看嵌套在类中对象的一种特殊情况：伴生对象。

4.4.2 伴生对象：工厂方法和静态成员的地盘

Kotlin 中的类不能拥有静态成员；Java 的 static 关键字并不是 Kotlin 语言的一部分。作为替代，Kotlin 依赖包级别函数（在大多数情形下能够替代 Java 的静态方法）和对象声明（在其他情况下替代 Java 的静态方法，同时还包括静态字段）。在大多数情况下，还是推荐使用顶层函数。但是顶层函数不能访问类的 private 成员，就像图 4.5 画出的那样。因此如果你需要写一个可以在没有类实例的情况下调用但是需要访问类内部的函数，可以将其写成那个类中的对象声明的成员。这种函数的一个例子就是工厂方法。

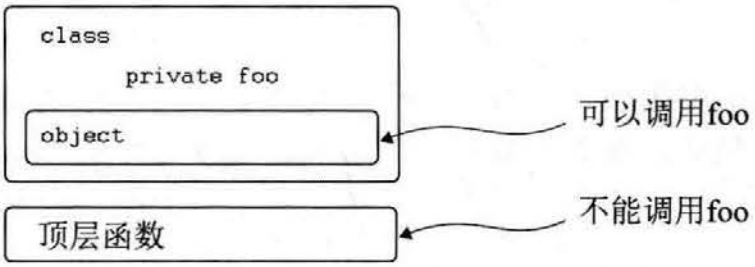


图 4.5 私有成员不能在类外部的顶层函数中使用

在类中定义的对象之一可以使用一个特殊的关键字来标记：companion。如果这样做，就获得了直接通过容器类名称来访问这个对象的方法和属性的能力，不再需要显式地指明对象的名称。最终的语法看起来非常像 Java 中的静态方法调用。下面就是展示这种语法的基础示例：

```
class A {
    companion object {
        fun bar() {
            println("Companion object called")
        }
    }
}

>>> A.bar()
Companion object called
```

还记得我们承诺过有一个调用 private 构造方法的好地方吗？那就是伴生对象。伴生对象可以访问类中的所有 private 成员，包括 private 构造方法，它是实现工厂模式的理想选择。

让我们来看看声明了两个构造方法的例子并将其改成使用在伴生对象中声明的工厂方法。我们将使用代码清单 4.14 中使用过的 FacebookUser 和 SubscribingUser 来举例。在此之前，这些实体是实现了通用接口 User 的不同的类。现在决定只用一个类来管理，但是在创建时提供不同的意义。

代码清单 4.25 定义一个拥有多个从构造方法的类

```
class User {
    val nickname: String

    constructor(email: String) {
        nickname = email.substringBefore('@')
    }

    constructor(facebookAccountId: Int) {
        nickname = getFacebookName(facebookAccountId)
    }
}
```

从构造方法

表示相同逻辑的另一种方法，就是使用工厂方法来创建类实例，这有多方面的好处。User 实例就是通过工厂方法来创建的，而不是通过多个构造方法。

代码清单 4.26 使用工厂方法来替代从构造方法

```
class User private constructor(val nickname: String) {
    companion object {
        fun newSubscribingUser(email: String) =
            User(email.substringBefore('@'))

        fun newFacebookUser(accountId: Int) =
            User(getFacebookName(accountId))
    }
}
```

将主构造方法标记为私有

声明伴生对象

用工厂方法通过 Facebook 账号来创建一个新用户

可以通过类名来调用 companion object 的方法：

```
>>> val subscribingUser = User.newSubscribingUser("bob@gmail.com")
>>> val facebookUser = User.newFacebookUser(4)

>>> println(subscribingUser.nickname)
bob
```

工厂方法是非常有用的。它们可以根据它们的用途来命名，就像示例中展示的那样。此外，工厂方法能够返回声明这个方法的类的子类，就像例子中的 SubscribingUser 和 FacebookUser 类。你还可以在不需要的时候避免创建新的对象。例如，你可以确保每一个 email 都与一个唯一的 User 实例对应，并且如果 email 在缓存中已经存在，那么在调用工厂方法时就返回这个存在的实例而不是创建一个新的。但是如果你需要扩展这样的类，使用多个构造方法也许是一个更好的方案，因为伴生对象成员在子类中不能被重写。

4.4.3 作为普通对象使用的伴生对象

伴生对象是一个声明在类中的普通对象。它可以有名字，实现一个接口或者有扩展函数或属性。在这一小节，我们会看到一个例子。

假设你正工作在一个公司工资单的网站服务上，并且需要在对象和 JSON 之间序列化和反序列化，可以将序列化的逻辑放在伴生对象中。

代码清单 4.27 声明一个命名伴生对象

```
class Person(val name: String) {
    companion object Loader {
        fun fromJSON(jsonText: String): Person = ...
    }
}

>>> person = Person.Loader.fromJSON("{name: 'Dmitry'}")
>>> person.name
Dmitry
>>> person2 = Person.fromJSON("{name: 'Brent'}")
>>> person2.name
Brent
```

可以通过两种方式来调用 fromJSON

在大多数情况下，通过包含伴生对象的类的名字来引用伴生对象，所以不必关心它的名字。但是如果需要你也可以指明，就像在代码清单 4.27 中那样：`companion object Loader`。如果你省略了伴生对象的名字，默认的名字将会分配为 `Companion`。接下来当我们讲到伴生对象扩展时，你将会看到使用这个名字的一些例子。

在伴生对象中实现接口

就像其他对象声明一样，伴生对象也可以实现接口。正如你即将看到的，可以直接将包含它的类的名字当作实现了该接口的对象实例来使用。

假定你的系统中有许多种对象，包括 `Person`，你想要提供一个通用的方式来创建所有类型的对象。假设你有一个 `JSONFactory` 接口可以从 JSON 反序列化对象，并且你的系统中的所有对象都通过这个工厂来创建。你可以为你的 `Person` 类提供一个这种接口的实现。

代码清单 4.28 在伴生对象中实现接口

```
interface JSONFactory<T> {
    fun fromJSON(jsonText: String): T
}

class Person(val name: String) {
    companion object : JSONFactory<Person> {
```

```
        override fun fromJSON(jsonText: String): Person = ...
    }
}
```

实现接口的伴生对象

这时，如果你有一个函数使用抽象方法来加载实体，可以传给它 Person 对象。

```
fun loadFromJSON<T>(factory: JSONFactory<T>): T {
    ...
}

loadFromJSON(Person)
```

将伴生对象实例传入函数中

注意，Person 类的名字被当作 JSONFactory 的实例。

Kotlin 的伴生对象和静态成员

类的伴生对象会同样被编译成常规对象：类中的一个引用了它的实例的静态字段。如果伴生对象没有命名，在 Java 代码中它可以通过 Companion 引用来访问：

```
/* Java */
Person.Companion.fromJSON("...");
```

如果伴生对象有名字，那就用这个名字替代 Companion。

但是你也许要和这样的 Java 代码一起工作，它需要类中的成员是静态的。可以在对应的成员上使用 @JvmStatic 注解来达到这个目的。如果你想声明一个 static 字段，可以再在一个顶层属性或者声明在 object 中的属性上使用 @JvmField 注解。这些功能专门为互操作性而存在，严格地讲，并不是核心语言的一部分。我们会在第 10 章详细讲到注解。

注意，Kotlin 可以访问在 Java 类中声明的静态方法和字段，使用与 Java 相同的语法。

伴生对象扩展

就像在 3.3 节看到的那样，扩展函数允许你定义可以通过代码库中其他地方定义的类实例调用的方法。但是如果你需要定义可以通过类自身调用的方法，就像伴生对象方法或者是 Java 静态方法该怎么办呢？如果类有一个伴生对象，可以通过在其上定义扩展函数来做到这一点。具体来说，如果类 C 有一个伴生对象，并且在 C.Companion 上定义了一个扩展函数 func，可以通过 C.func() 来调用它。

例如，假设你希望你的 Person 类有一个清晰的关注点分离，这个类本身会是核心业务逻辑模块的一部分，但是你并不想将这个模块与任何特定的数据格式耦合起来。正因为如此，反序列化函数需要定义在模块中用来负责客户端 / 服务器通信。

可以使用扩展函数来做到这一点。注意，你该怎样使用默认名字（Companion）来引用没有显式地定义名字的伴生对象。

代码清单 4.29 为伴生对象定义一个扩展函数

```
// business logic module
class Person(val firstName: String, val lastName: String) {
    companion object {
    }
}

// client/server communication module
fun Person.Companion.fromJSON(json: String): Person {
    ...
}

val p = Person.fromJSON(json)
```

声明一个空的伴生对象

声明一个扩展函数

你调用 fromJSON 就好像它是一个伴生对象定义的方法一样，但是实际上它是作为扩展函数在外部定义的。正如之前的扩展函数一样，看起来像是一个成员，但实际并不是。但是请注意，为了能够为你的类定义扩展，必须在其中声明一个伴生对象，即使是一个空的。

你已经见过了伴生对象是多么有用，现在让我们来看看 Kotlin 的下一个使用同样的 object 关键字表示的功能：对象表达式。

4.4.4 对象表达式：改变写法的匿名内部类

object 关键字不仅仅能用来声明单例式的对象，还能用来声明*匿名对象*。匿名对象替代了 Java 中匿名内部类的用法。例如，让我们来看看怎样将一个典型的 Java 匿名内部类用法——事件监听器——转换成 Kotlin：

代码清单 4.30 使用匿名对象来实现事件监听器

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

        override fun mouseEntered(e: MouseEvent) {
            // ...
        }
    }
)
```

声明一个继承 MouseAdapter 的匿名对象

重写 MouseAdapter 方法

除了去掉了对象的名字外，语法是与对象声明相同的。对象表达式声明了一个

类并创建了该类的一个实例，但是并没有给这个类或是实例分配一个名字。通常来说，它们都是不需要名字的，因为你会将这个对象用作一个函数调用的参数。如果你需要给对象分配一个名字，可以将其存储到一个变量中：

```
val listener = object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }
    override fun mouseEntered(e: MouseEvent) { ... }
}
```

与 Java 匿名内部类只能扩展一个类或实现一个接口不同，Kotlin 的匿名对象可以实现多个接口或者不实现接口。

注意 与对象声明不同，匿名对象不是单例的。每次对象表达式被执行都会创建一个新的对象实例。

与 Java 的匿名类一样，在对象表达式中的代码可以访问创建它的函数中的变量。但是与 Java 不同，访问并没有被限制在 final 变量，还可以在对象表达式中修改变量的值。例如，我们来看看怎样使用监听器对窗口点击计数。

代码清单 4.31 从匿名对象访问局部变量

```
fun countClicks(window: Window) {
    var clickCount = 0                                  <-- 声明局部变量
    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++                                <-- 更新变量的值
        }
    })
    // ...
}
```

注意 对象表达式在需要在匿名对象中重写多个方法时是最有用的。如果只需要实现一个单方法的接口（就像 Runnable，可以将你的实现写作函数字面值（lambda）并依靠 Kotlin 的 SAM 转换（把函数字面值转换成单抽象函数接口的实现）。我们会在第 5 章讨论 lambda 和 SAM 转换的更多细节。

我们已经完成了关于类、接口和对象的讨论。在接下来的一章，我们将会转到 Kotlin 最有趣的一部分之一：lambda 和函数式编程。

4.5 小结

- Kotlin 的接口与 Java 的相似，但是可以包含默认实现（Java 从第 8 版才开始支持）和属性。
- 所有的声明默认都是 final 和 public 的。
- 要想使声明不是 final 的，将其标记为 open。
- internal 声明在同一模块中可见。
- 嵌套类默认不是内部类。使用 inner 关键字来存储外部类的引用。
- sealed 类的子类只能嵌套在自身的声明中（Kotlin 1.1 允许将子类放置在同一文件的任意地方）。
- 初始化语句块和从构造方法为初始化类实例提供了灵活性。
- 使用 field 标识符在访问器方法体中引用属性的支持字段。
- 数据类提供了编译器生成的 equals、hashCode、toString、copy 和其他方法。
- 类委托帮助避免在代码中出现许多相似的委托方法。
- 对象声明是 Kotlin 中定义单例类的方法。
- 伴生对象（与包级别函数和属性一起）替代了 Java 静态方法和字段定义。
- 伴生对象与其他对象一样，可以实现接口，也可以拥有有扩展函数和属性。
- 对象表达式是 Kotlin 中针对 Java 匿名内部类的替代品，并增加了诸如实现多个接口的能力和修改在创建对象的作用域中定义的变量的能力等功能。

5 Lambda 编程

本章内容包括

- Lambda 表达式和成员引用
- 以函数式风格使用集合
- 序列：惰性地执行集合操作
- 在 Kotlin 中使用 Java 函数式接口
- 使用带接收者的 lambda

Lambda 表达式，或简称 *lambda*，本质上就是可以传递给其他函数的一小段代码。有了 lambda，可以轻松地把通用的代码结构抽取成库函数，Kotlin 标准库就大量地使用了它们。最常见的一种 lambda 用途就是和集合一起工作，在这一章中你会看到许多将 lambda 传给标准库函数从而替换掉常见的集合访问模式的例子。你还会看到 lambda 可以与任意 Java 库一起使用，即使是那些原本设计时没有考虑 lambda 的库。最后，我们会看看带接收者的 lambda，这是一种特殊的 lambda，它的函数体在不同于包围它代码的上下文中执行。

5.1 Lambda表达式和成员引用

把 lambda 引入 Java 8 的是 Java 这门语言演变过程中让人望眼欲穿的变化之一。为什么它是如此重要？这一节中，你会发现为何 lambda 这么好用，以及 Kotlin 的

lambda 语法看起来是什么样子的。

5.1.1 Lambda 简介：作为函数参数的代码块

在你的代码中存储和传递一小段行为是常有的任务。例如，你常常需要表达像这样的想法："当一个事件发生的时候运行这个事件处理器"又或者是"把这个操作应用到这个数据结构中所有的元素上"。在老版本的 Java 中，可以用匿名内部类来实现。这种技巧可以工作但是语法太啰唆了。

函数式编程提供了另外一种解决问题的方法：把函数当作值来对待。可以直接传递函数，而不需要先声明一个类再传递这个类的实例。使用 lambda 表达式之后，代码会更加简洁。都不需要声明函数了：相反，可以高效地直接传递代码块作为函数参数。

我们来看一个例子。假设你要定义一个点击按钮的行为，添加一个负责处理点击的监听器。监听器实现了相应的接口 `OnClickListener` 和它的一个方法 `onClick`。

代码清单 5.1 用匿名内部类实现监听器

```
/* Java */
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View view) {
        /* 点击后执行的动作 */
    }
});
```

这些啰唆的声明匿名内部类的写法重复多次之后让人烦躁。这种行为的表示法——点击的时候要做什么——有助于消除冗余的代码。在 Kotlin 中，可以像 Java 8 一样使用 lambda。

代码清单 5.2 用 lambda 实现监听器

```
button.setOnClickListener { /* 点击后执行的动作 */ }
```

这段 Kotlin 代码和 Java 的匿名内部类做的事情一模一样，但是更简洁易读。本节后面部分我们会讨论这个例子的细节。

看过了 lambda 如何被当作只有一个方法的匿名对象的替代品使用之后，现在我们继续看看另一种 lambda 表达式的经典用途：和集合一起工作。

5.1.2 Lambda 和集合

良好编程风格的主要原则之一是避免代码中的任何重复。我们对集合执行的大部分任务都遵循几个通用的模式，所以实现这几个模式的代码应该放在一个库里。但是没有 lambda 的帮助，很难为使用集合提供一个好用方便的库。因此如果你用 Java（Java 8 之前）编写代码，很有可能已经养成了什么东西都要自己实现的习惯。在 Kotlin 中这个习惯必须纠正。

我们来看一个例子。你会用到一个 Person 类，它包含了这个人的名字和年龄信息。

```
data class Person(val name: String, val age: Int)
```

假设现在你有一个人的列表，需要找到列表中年龄最大的那个人。如果完全不了解 lambda，你可能会急急忙忙地手动实现这个搜索功能。你会先引入两个中间变量，一个用来保存最大的年龄，而另一个用来保存找到的此年龄的第一个人。然后迭代整个列表，不断更新这两个变量。

代码清单 5.3 手动在集合中搜索

```
fun findTheOldest(people: List<Person>) {
    var maxAge = 0                          // 存储最大年龄
    var theOldest: Person? = null           // 存储年龄最大的人
    for (person in people) {
        if (person.age > maxAge) {          // 如果下一个人比现在年龄最大的人还要大，改变最大值
            maxAge = person.age
            theOldest = person
        }
    }
    println(theOldest)
}
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> findTheOldest(people)
Person(name=Bob, age=31)
```

如果你经验丰富，可以非常快地就写出这样的循环。但这里代码不少，而且很容易犯错。例如，你可能用了错误的比较，找到的是最小的元素而不是最大的那个。

Kotlin 有更好的方法。可以使用库函数，如下所示。

代码清单 5.4 用 lambda 在集合中搜索

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })      // 比较年龄找到最大的元素
Person(name=Bob, age=31)
```

maxBy 函数可以在任何集合上调用，且只需要一个实参：一个函数，指定比

较哪个值来找到最大元素。花括号中的代码 {it.age} 就是实现了这个逻辑的 lambda。它接收一个集合中的元素作为实参(使用 it 引用它)并且返回用来比较的值。这个例子中，集合元素是 Person 对象，用来比较的是存储在其 age 属性中的年龄。

如果 lambda 刚好是函数或者属性的委托，可以用成员引用替换。

代码清单 5.5 用成员引用搜索

```
people.maxBy(Person::age)
```

代码清单 5.5 和前面的例子含义一样。5.1.5 节会涵盖更多其中的细节。

我们通常能对 Java（Java 8 之前）中的集合做的大多数事情可以通过使用 lambda 或成员引用的库函数来更好地表达。这样产生的代码要短得多，也容易理解得多。为了帮助你开始习惯它，我们来看看 lambda 表达式的语法。

5.1.3 Lambda 表达式的语法

如前所述，一个 lambda 把一小段行为进行编码，你能把它当作值到处传递。它可以被独立地声明并存储到一个变量中。但是更常见的还是直接声明它并传递给函数。图 5.1 展示了声明 lambda 表达式的语法。

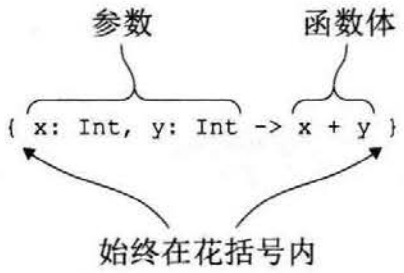


图 5.1 Lambda 表达式的语法

Kotlin 的 lambda 表达式始终用花括号包围。注意实参并没有用括号括起来。箭头把实参列表和 lambda 的函数体隔开。

可以把 lambda 表达式存储在一个变量中，把这个变量当作普通函数对待（即通过相应实参调用它）：

```
>>> val sum = { x: Int, y: Int -> x + y }
>>> println(sum(1, 2))
3
```

调用保存在变量中的 lambda

如果你乐意，还可以直接调用 lambda 表达式：

```
>>> { println(42) }()
42
```

但是这样的语法毫无可读性，也没有什么意义（它等价于直接执行 lambda 函数体中的代码）。如果你确实需要把一小段代码封闭在一个代码块中，可以使用库函数 run 来执行传给它的 lambda：

```
>>> run { println(42) }
42
```

运行 lambda 中的代码

在 8.2 节中，我们将会了解到这种调用和内建语言结构一样高效且不会带来额外运行时开销，以及背后的原因。我们回到代码清单 5.4 中寻找列表中年龄最大的人的代码：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

如果不用任何简明语法来重写这个例子，你会得到下面的代码：

```
people.maxBy({ p: Person -> p.age })
```

这段代码一目了然：花括号中的代码片段是 lambda 表达式，把它作为实参传给函数。这个 lambda 接收一个类型为 Person 的参数并返回它的年龄。

但是这段代码有点啰唆。首先，过多的标点符号破坏了可读性。其次，类型可以从上下文推断出来并可以省略。最后，这种情况下不需要给 lambda 的参数分配一个名称。

让我们来改进这些地方，先拿花括号开刀。Kotlin 有这样一种语法约定，如果 lambda 表达式是函数调用的最后一个实参，它可以放到括号的外边。这个例子中，lambda 是唯一的实参，所以可以放到括号的后边：

```
people.maxBy() { p: Person -> p.age }
```

当 lambda 是函数唯一的实参时，还可以去掉调用代码中的空括号对：

```
people.maxBy { p: Person -> p.age }
```

三种语法形式含义都是一样的，但最后一种最易读。如果 lambda 是唯一的实参，你当然愿意在写代码的时候省掉这些括号。而当你有多个实参时，既可以把 lambda 留在括号内来强调它是一个是实参，也可以把它放在括号的外面，两种选择都是可行的。如果你想传递两个或更多的 lambda，不能把超过一个的 lambda 放到外面。这时使用常规语法来传递它们通常是更好的选择。

要看看这些选项在更复杂的调用中是怎样的，我们回顾一下第 3 章中大量使用的 joinToString 函数。Kotlin 标准库中也有定义它，标准库中的这个版本的不同之处在于它可以接收一个附加的函数参数。这个函数可以用 toString 函数以外

的方法来把一个元素转换成字符串。下面的例子展示了你可以用它只打印出人的名字。

代码清单 5.6 把 lambda 作为命名实参传递

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> val names = people.joinToString(separator = " ",
...                                 transform = { p: Person -> p.name })
>>> println(names)
Alice Bob
```

下面的例子展示了可以怎样重写这个调用，把 lambda 放在放在括号外。

代码清单 5.7 把 lambda 放在括号外传递

```
people.joinToString(" ") { p: Person -> p.name }
```

代码清单 5.6 使用了一个命名实参来传递 lambda，清楚地表示了 lambda 应用到了哪里。代码清单 5.7 更简洁，但没有显式地表明 lambda 应用到了哪里，所以不熟悉被调用的函数的那些人可能更难理解。

INTELLIJ IDEA 小贴士 想要在不同的语法形式之间切换，可以使用这些动作："Move lambda expression out of parentheses"（把 lambda 表达式移到括号外）和 "Move lambda expression into parentheses"（把 lambda 表达式移到括号内）。

让我们继续简化语法，移除参数的类型。

代码清单 5.8 省略 lambda 参数类型

```
people.maxBy { p: Person -> p.age }
people.maxBy { p -> p.age }
```

显式地写出参数类型

推导出参数类型

和局部变量一样，如果 lambda 参数的类型可以被推导出来，你就不需要显式地指定它。以这里的 maxBy 函数为例，其参数类型始终和集合的元素类型相同。编译器知道你是对一个 Person 对象的集合调用 maxBy 函数，所以它能推断 lambda 参数也会是 Person 类型。

也存在编译器不能推断出 lambda 参数类型的情况，但这里我们暂不讨论。可以遵循这样一条简单的规则：先不声明类型，等编译器报错后再指定它们。

可以指定部分实参的类型，而剩下的实参只用名称。如果编译器不能推导出其

中一种类型，又或是显式的类型可以提升可读性，这种做法或许更方便。

这个例子你能做的最后简化是使用默认参数名称 it 代替命名参数。如果当前上下文期望的是只有一个参数的 lambda 且这个参数的类型可以推断出来，就会生成这个名称。

代码清单 5.9 使用默认参数名称

```
people.maxBy { it.age }
```

"it" 是自动生成的参数名称

仅在实参名称没有显式地指定时这个默认的名称才会生成。

注意 it 约定能大大缩短你的代码，但你不应该滥用它。尤其是在嵌套 lambda 的情况下，最好显式地声明每个 lambda 的参数。否则，很难搞清楚 it 引用的到底是哪个值。如果上下文中参数的类型或意义都不是很明朗，显式声明参数的方法也很有效。

如果你用变量存储 lambda，那么就没有可以推断出参数类型的上下文，所以你必须显式地指定参数类型：

```
>>> val getAge = { p: Person -> p.age }
>>> people.maxBy(getAge)
```

迄今为止，你看到的例子都是由单个表达式或语句构成的 lambda。但是 lambda 并没有被限制在这样小的规模，它可以包含更多的语句。下面这种情况，最后一个表达式就是（lambda 的）结果：

```
>>> val sum = { x: Int, y: Int ->
...     println("Computing the sum of $x and $y...")
...     x + y
... }
>>> println(sum(1, 2))
Computing the sum of 1 and 2...
3
```

接下来，我们谈谈和 lambda 表达式形影不离的概念：从上下文中捕捉变量。

5.1.4 在作用域中访问变量

当在函数内声明一个匿名内部类的时候，能够在这个匿名类内部引这个函数的参数和局部变量。也可以用 lambda 做同样的事情。如果在函数内部使用 lambda，也可以访问这个函数的参数，还有在 lambda 之前定义的局部变量。

我们用标准库函数 forEach 来展示这种行为。它是最基本的集合操作函

数之一；它所做的全部事情就是在集合中的每一个元素上都调用给定的 lambda。forEach 函数比普通 for 循环更简洁一些，但它并没有更多其他优势，所以你不必急于把所有的循环换成 lambda。

下面的代码清单接收一个消息列表，把每条消息都加上相同的前缀打印出来。

代码清单 5.10 在 lambda 中使用函数参数

```
fun printMessagesWithPrefix(messages: Collection<String>, prefix: String) {
    messages.forEach {
        println("$prefix $it")
    }
}

>>> val errors = listOf("403 Forbidden", "404 Not Found")
>>> printMessagesWithPrefix(errors, "Error:")
Error: 403 Forbidden
Error: 404 Not Found
```

在 lambda 中访问 “prefix” 参数

接收 lambda 作为实参，指定对每个元素的操作

这里 Kotlin 和 Java 的一个显著区别就是，在 Kotlin 中不会仅限于访问 final 变量，在 lambda 内部也可以修改这些变量。下面这个代码清单对给定的响应状态码 set 中的客户端与服务器错误分别计数。

代码清单 5.11 在 lambda 中改变局部变量

```
fun printProblemCounts(responses: Collection<String>) {
    var clientErrors = 0
    var serverErrors = 0
    responses.forEach {
        if (it.startsWith("4")) {
            clientErrors++
        } else if (it.startsWith("5")) {
            serverErrors++
        }
    }
    println("$clientErrors client errors, $serverErrors server errors")
}

>>> val responses = listOf("200 OK", "418 I'm a teapot",
...                        "500 Internal Server Error")
>>> printProblemCounts(responses)
1 client errors, 1 server errors
```

声明将在 lambda 内部访问的变量

在 lambda 中修改变量

和 Java 不一样，Kotlin 允许在 lambda 内部访问非 final 变量甚至修改它们。从 lambda 内访问外部变量，我们称这些变量被 lambda 捕捉，就像这个例子中的 prefix、clientErrors 以及 serverErrors 一样。

注意，默认情况下，局部变量的生命期被限制在声明这个变量的函数中。但是

如果它被 lambda 捕捉了，使用这个变量的代码可以被存储并稍后再执行。你可能会问这是什么原理。当你捕捉 final 变量时，它的值和使用这个值的 lambda 代码一起存储。而对非 final 变量来说，它的值被封装在一个特殊的包装器中，这样你就可以改变这个值，而对这个包装器的引用会和 lambda 代码一起存储。

捕捉可变变量：实现细节

Java 只允许你捕捉 final 变量。当你想捕捉可变变量的时候，可以使用下面两种技巧：要么声明一个单元素的数组，其中存储可变值；要么创建一个包装类的实例，其中存储要改变的值的引用。如果你在 Kotlin 中显式地使用这些技术，代码看起来是这样的：

```
class Ref<T>(var value: T)
>>> val counter = Ref(0)
>>> val inc = { counter.value++ }
```

模拟捕捉可变变量的类

形式上是不变量被捕捉了，但是存储在字段中的实际值是可以修改的

在实际代码中，你不需要创建这样的包装器，可以直接修改这个变量：

```
var counter = 0
val inc = { counter++ }
```

这是什么原理？第一例子展示的就是第二个例子背后的原理。任何时候你捕捉了一个 final 变量（val），它的值被拷贝下来，这和 Java 一样。而当你捕捉了一个可变变量（var）时，它的值被作为 Ref 类的一个实例被存储下来。Ref 变量是 final 的能轻易地被捕捉，然而实际值被存储在其字段中，并且可以在 lambda 内修改。

这里有一个重要的注意事项，如果 lambda 被用作事件处理器或者用在其他异步执行的情况，对局部变量的修改只会在 lambda 执行的时候发生。例如，下面这段代码并不是记录按钮点击次数的正确方法：

```
fun tryToCountButtonClicks(button: Button): Int {
    var clicks = 0
    button.onClick { clicks++ }
    return clicks
}
```

这个函数始终返回 0。尽管 onClick 处理器可以修改 clicks 的值，你并不能观察到值发生了变化，因为 onClick 处理器是在函数返回之后调用的。这个函

数正确的实现需要把点击次数存储在函数外依然可以访问的地方——例如类的属性，而不是存储在函数的局部变量中。

我们已经讨论了声明 lambda 的语法，也讨论了 lambda 是怎样捕捉变量的。现在我们谈谈成员引用，这种特性让你可以轻松地传递既有函数的引用。

5.1.5 成员引用

你已经看到 lambda 是如何让你把代码块作为参数传递给函数的。但是如果你想要当作参数传递的代码已经被定义成了函数，该怎么办？当然可以传递一个调用这个函数的 lambda，但这样做有点多余。那么你能直接传递函数吗？

Kotlin 和 Java 8 一样，如果把函数转换成一个值，你就可以传递它。使用 :: 运算符来转换：

```
val getAge = Person::age
```

这种表达式称为成员引用，它提供了简明语法，来创建一个调用单个方法或者访问单个属性的函数值。双冒号把类名称与你要引用的成员（一个方法或者一个属性）名称隔开，如图 5.2 所示。



图 5.2 成员引用语法

这是一个更简洁的 lambda 表达式，它做同样的事情：

```
val getAge = { person: Person -> person.age }
```

注意，不管你引用的是函数还是属性，都不要在成员引用的名称后面加括号。

成员引用和调用该函数的 lambda 具有一样的类型，所以可以互换使用：

```
people.maxBy(Person::age)
```

还可以引用顶层函数（不是类的成员）：

```
fun salute() = println("Salute!")
>>> run(::salute)
Salute!
```

引用顶层函数

这种情况下，你省略了类名称，直接以 :: 开头。成员引用 ::salute 被当作实参传递给库函数 run，它会调用相应的函数。

如果 lambda 要委托给一个接收多个参数的函数，提供成员引用代替它将会非常方便：

```
val action = { person: Person, message: String ->
    sendEmail(person, message)
}
val nextAction = ::sendEmail
```

这个 lambda 委托给 sendEmail 函数

可以用成员引用代替

可以用构造方法引用存储或者延期执行创建类实例的动作。构造方法引用的形式是在双冒号后指定类名称：

```
data class Person(val name: String, val age: Int)

>>> val createPerson = ::Person
>>> val p = createPerson("Alice", 29)
>>> println(p)
Person(name=Alice, age=29)
```

创建“Person”实例的动作被保存成了值

注意，还可以用同样的方式引用扩展函数：

```
fun Person.isAdult() = age >= 21
val predicate = Person::isAdult
```

尽管 isAdult 不是 Person 类的成员，还是可以通过引用访问它，这和访问实例的成员没什么两样：person.isAdult()。

绑定引用

在 Kotlin 1.0 中，当接收一个类的方法或属性引用时，你始终需要提供一个该类的实例来调用这个引用。Kotlin 1.1 计划支持绑定成员引用，它允许你使用成员引用语法捕捉特定实例对象上的方法引用。

```
>>> val p = Person("Dmitry", 34)
>>> val personsAgeFunction = Person::age
>>> println(personsAgeFunction(p))
34
>>> val dmitrysAgeFunction = p::age
>>> println(dmitrysAgeFunction())
34
```

在 Kotlin 1.1 中可以使用绑定成员引用

注意，personsAgeFunction 是一个单参数函数（返回给定人的年龄），而 dmitrysAgeFunction 是一个零参数的函数（返回已经指定好的人的年龄）。在 Kotlin 1.1 之前，你需要显式地写出 lambda{p.age}，而不是使用绑定成员引用 p::age。

在接下来的一节中，我们会看到许多和 lambda 表达式及成员引用相得益彰的库函数。

5.2 集合的函数式API

函数式编程风格在操作集合时提供了很多优势。大多数任务都可以通过库函数完成，来简化你的代码。在这一节中，我们将讨论 Kotlin 标准库中和集合有关的一些函数。我们先从 filter 和 map 这类函数及它们背后的概念开始。我们还将介绍其他有用的函数，并且给你一些提示，告诉你如何避免过度使用它们，以及如何写出清晰易懂的代码。

注意所有这些函数都是不是 Kotlin 的设计者发明的。这些函数或者类似的函数存在于所有支持 lambda 的语言中，包括 C#、Groovy 和 Scala。如果这些概念你已经烂熟于心，可以快速浏览下面的例子，跳过解释的部分。

5.2.1 基础：filter 和 map

filter 和 map 函数形成了集合操作的基础，很多集合操作都是借助它们来表达的。

每个函数我们都会给出两个例子，一个使用数字，另一个使用熟悉的 Person 类：

```
data class Person(val name: String, val age: Int)
```

filter 函数遍历集合并选出应用给定 lambda 后会返回 true 的那些元素：

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.filter { it % 2 == 0 })
[2, 4]
```

只有偶数留了下来

上面的结果是一个新的集合，它只包含输入集合中那些满足判断式的元素，如图 5.3 所示。

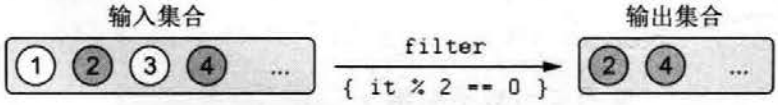


图 5.3 filter 函数选出了匹配给定判断式的元素

如果你想留下那些超过 30 岁的人，可以用 filter：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.filter { it.age > 30 })
[Person(name=Bob, age=31)]
```

filter 函数可以从集合中移除你不想要的元素，但是它并不会改变这些元素。元素的变换是 map 的用武之地。

map 函数对集合中的每一个元素应用给定的函数并把结果收集到一个新集合。可以把数字列表变换成它们平方的列表，比如：

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.map { it * it })
[1, 4, 9, 16]
```

结果是一个新集合，包含的元素个数不变，但是每个元素根据给定的判断式做了变换，如图 5.4 所示。

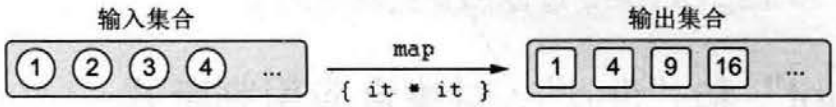


图 5.4 map 对集合中的每个元素都应用 lambda

如果你想打印的只是一个姓名列表，而不是人的完整信息列表，可以用 map 来变换列表：

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.map { it.name })
[Alice, Bob]
```

注意，这个例子可以用成员引用漂亮地重写：

```
people.map(Person::name)
```

可以轻松地把多次这样的调用链接起来。例如，打印出年龄超过 30 岁的人的名字：

```
>>> people.filter { it.age > 30 }.map(Person::name)
[Bob]
```

现在，如果说需要这个分组中所有年龄最大的人的名字，可以先找到分组中人的最大年龄，然后返回所有这个年龄的人。很容易就用 lambda 写出如下代码：

```
people.filter { it.age == people.maxBy(Person::age).age }
```

但是请注意，这段代码对每个人都会重复寻找最大年龄的过程，假设集合中有 100 个人，寻找最大年龄的过程就会执行 100 遍！

下面的解决方法做出了改进，只计算了一次最大年龄：

```
val maxAge = people.maxBy(Person::age).age
people.filter { it.age == maxAge }
```

如果没有必要就不要重复计算！使用 lambda 表达式的代码看起来简单，有时候却掩盖了底层操作的复杂性。始终牢记你写的代码在干什么。

还可以对 map 应用过滤和变换函数：

```
>>> val numbers = mapOf(0 to "zero", 1 to "one")
>>> println(numbers.mapValues { it.value.toUpperCase() })
{0=ZERO, 1=ONE}
```

键和值分别由各自的函数来处理。filterKeys 和 mapKeys 过滤和变换 map 的键，而另外的 filterValues 和 mapValues 过滤和变换对应的值。

5.2.2 “all”“any”“count”和“find”：对集合应用判断式

另一种常见的任务是检查集合中的所有元素是否都符合某个条件（或者它的变种，是否存在符合的元素）。Kotlin 中，它们是通过 all 和 any 函数表达的。count 函数检查有多少元素满足判断式，而 find 函数返回第一个符合条件的元素。

为了演示这些函数，我们先定义一个判断式 canBeInClub27，来检查一个人是否还没有到 28 岁：

```
val canBeInClub27 = { p: Person -> p.age <= 27 }
```

如果你对是否所有元素都满足判断式感兴趣，应该使用 all 函数：

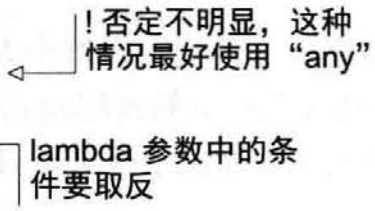
```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.all(canBeInClub27))
false
```

如果你需要检查集合中是否至少存在一个匹配的元素，那就用 any：

```
>>> println(people.any(canBeInClub27))
true
```

注意，!all（“不是所有”）加上某个条件，可以用 any 加上这个条件的取反来替换，反之亦然。为了让你的代码更容易理解，应该选择前面不需要否定符号的函数：

```
>>> val list = listOf(1, 2, 3)
>>> println(!list.all { it == 3 })
true
>>> println(list.any { it != 3 })
true
```



第一行检查是保证不是所有的元素都等于3。这和至少有一个元素不是3是一个意思，这正是你在第二行用any做的检查。

如果你想知道有多少个元素满足了判断式，使用count：

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.count(canBeInClub27))
1
```

使用正确的函数完成工作："count" VS. "size"

count方法容易被遗忘，然后通过过滤集合之后再取大小来实现它：

```
>>> println(people.filter(canBeInClub27).size)
1
```

在这种情况下，一个中间集合会被创建并用来存储所有满足判断式的元素。而另一方面，count方法只是跟踪匹配元素的数量，不关心元素本身，所以更高效。

一般的规则是，尝试找到适合你需求的最合适的操作。

要找到一个满足判断式的元素，使用find函数：

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.find(canBeInClub27))
Person(name=Alice, age=27)
```

如过有多个匹配的元素就返回其中第一个元素；或者返回null，如果没有一个元素能满足判断式。find还有一个同义方法firstOrNull，可以使用这个方法更清楚地表述你的意图。

5.2.3 groupBy：把列表转换成分组的map

假设你需要把所有元素按照不同的特征划分成不同的分组。例如，你想把人按年龄分组，相同年龄的人放在一组。把这个特征直接当作参数传递十分方便。groupBy函数可以帮你做到这一点：

```
>>> val people = listOf(Person("Alice", 31),
...         Person("Bob", 29), Person("Carol", 31))
>>> println(people.groupBy { it.age })
```

这次操作的结果是一个map，是元素分组依据的键（这个例子中是age）和元素分组（persons）之间的映射，如图5.5所示。

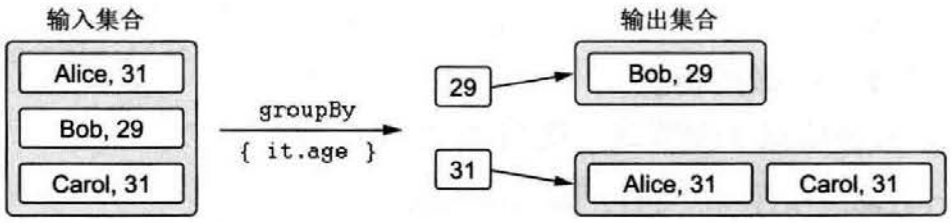


图 5.5 应用函数 groupBy 的结果

这个例子的输出会是这样的：

```
{29=[Person(name=Bob, age=29)],
 31=[Person(name=Alice, age=31), Person(name=Carol, age=31)]}
```

每一个分组都是存储在一个列表中，结果的类型就是 Map<Int, List<Person>>。可以使用像 mapKeys 和 mapValues 这样的函数对这个 map 做进一步的修改。

我们再来看另外一个例子，如何使用成员引用把字符串按照首字母分组：

```
>>> val list = listOf("a", "ab", "b")
>>> println(list.groupBy(String::first))
{a=[a, ab], b=[b]}
```

注意，这里 first 并不是 String 类的成员，而是一个扩展。然而，可以把它当作成员引用访问。

5.2.4 flatMap 和 flatten：处理嵌套集合中的元素

现在让我们先把人的例子的讨论放在一边，换一个书籍的例子。假设你有一堆藏书，使用 Book 类表示：

```
class Book(val title: String, val authors: List<String>)
```

每本书都可能有一个或者多个作者，可以统计出图书馆中的所有作者的 set：

```
books.flatMap { it.authors }.toSet()
```

包含撰写“books”集合中书籍的所有作者的 set

flatMap 函数做了两件事情：首先根据作为实参给定的函数对集合中的每个元素做变换（或者说映射），然后把多个列表合并（或者说平铺）成一个列表。下面这个字符串的例子很好地阐明了这个概念，如图 5.6 所示。

```
>>> val strings = listOf("abc", "def")
>>> println(strings.flatMap { it.toList() })
[a, b, c, d, e, f]
```

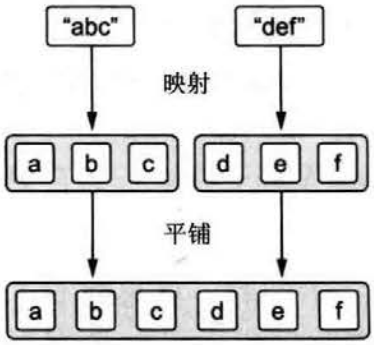


图 5.6 应用 flatMap 函数之后的结果

字符串上的 toList 函数把它转换成字符列表。如果和 toList 一起使用的是 map 函数，你会得到一个字符列表的列表，就如同图中的第二行。flatMap 函数还会执行后面的步骤，并返回一个包含所有元素（字符）的列表。

让我们回到书籍作者的例子：

```
>>> val books = listOf(Book("Thursday Next", listOf("Jasper Fforde")),
...                    Book("Mort", listOf("Terry Pratchett")),
...                    Book("Good Omens", listOf("Terry Pratchett",
...                                              "Neil Gaiman")))
>>> println(books.flatMap { it.authors }.toSet())
[Jasper Fforde, Terry Pratchett, Neil Gaiman]
```

每一本书都可能有多位作者，属性 book.authors 存储了每本书籍的作者集合。flatMap 函数把所有书籍的作者合并成了一个扁平的列表。toSet 调用移除了结果集合中的所有重复元素。所以这个例子中，Terry Pratchett 在输出中只出现了一次。

当你卡壳在元素集合的集合不得不合并成一个的时候，你可能会想起 flatMap 来。注意，如果你不需要做任何变换，只是需要平铺一个集合，可以使用 flatten 函数：listOfLists.flatten()。

我们已经着重介绍了 Kotlin 标准库中的一些集合操作函数。受限于篇幅，我们不会介绍集合全部的操作函数，此外，展示一大串函数的清单也挺无聊的。当你编写关于集合的代码时，我们一般的建议是：想一想这个操作如何用一个通用的变换来表达，然后寻找能执行这种变换的库函数。你很有可能会找到这样的函数并使用它更快地解决问题，而不是手动去实现它。

现在让我们更深入地研究链式集合操作的代码的性能。在下一节中，我们会看到这些操作不同的执行方法。

5.3 惰性集合操作：序列

在前面一节中，你看到了许多链式集合函数调用的例子，比如 map 和 filter。

这些函数会及早地创建中间集合，也就是说每一步的中间结果都被存储在一个临时列表。序列给了你执行这些操作的另一种选择，可以避免创建这些临时中间对象。

先来看个例子：

```
people.map(Person::name).filter { it.startsWith("A") }
```

Kotlin 标准库参考文档有说明，filter 和 map 都会返回一个列表。这意味着上面例子中的链式调用会创建两个列表：一个保存 filter 函数的结果，另一个保存 map 函数的结果。如果源列表只有两个元素，这不是什么问题，但是如果有一百万个元素，（链式）调用就会变得十分低效。

为了提高效率，可以把操作变成使用序列，而不是直接使用集合：

```
people.asSequence()
    .map(Person::name)
    .filter { it.startsWith("A") }
    .toList()
```

把初始集合转换成序列

序列支持和集合一样的 API

把结果序列转换回列表

应用这次操作后的结果和前面的例子一模一样：一个以字母 A 开头的人名列表。但是第二个例子没有创建任何用于存储元素的中间集合，所以元素数量巨大的情况下性能将显著提升。

Kotlin 惰性集合操作的入口就是 Sequence 接口。这个接口表示的就是一个可以逐个列举元素的元素序列。Sequence 只提供了一个方法，iterator，用来从序列中获取值。

Sequence 接口的强大之处在于其操作的实现方式。序列中的元素求值是惰性的。因此，可以使用序列更高效地对集合元素执行链式操作，而不需要创建额外的集合来保存过程中产生的中间结果。

可以调用扩展函数 asSequence 把任意集合转换成序列，调用 toList 来做反向的转换。

为什么需要把序列转换回集合？用序列代替集合不是更方便吗？特别是它们还有这么多优点。答案是：有的时候是这样。如果你只需要迭代序列中的元素，可以直接使用序列。如果你要用其他的 API 方法，比如用下标访问元素，那么你需要把序列转换成列表。

> **注意** 通常，需要对一个**大型**集合执行链式操作时要使用序列。在 8.2 节中，我们将讨论 Kotlin 常规集合的及早操作高效的原因，尽管它会创建中间集合。但是如果集合拥有数量巨大的元素，元素为中间结果进行重新分配开销巨大，所以惰性求值是更好的选择。

因为序列的操作是惰性的，为了执行它们，你需要直接迭代序列元素，或者把序列转换成一个集合。接下来的这一小节会做出解释。

5.3.1 执行序列操作：中间和末端操作

序列操作分为两类：中间的和末端的。一次中间操作返回的是另一个序列，这个新序列知道如何变换原始序列中的元素。而一次末端操作返回的是一个结果，这个结果可能是集合、元素、数字，或者其他从初始集合的变换序列中获取的任意对象（如图 5.7 所示）。



图 5.7 序列的中间和末端操作

中间操作始终都是惰性的。先看看下面这个缺少了末端操作的例子：

```
>>> listOf(1, 2, 3, 4).asSequence()
...         .map { print("map($it) "); it * it }
...         .filter { print("filter($it) "); it % 2 == 0 }
```

执行这段代码并不会在控制台上输出任何内容。这意味着 map 和 filter 变换被延期了，它们只有在获取结果的时候才会被应用（即末端操作被调用的时候）：

```
>>> listOf(1, 2, 3, 4).asSequence()
...         .map { print("map($it) "); it * it }
...         .filter { print("filter($it) "); it % 2 == 0 }
...         .toList()
map(1) filter(1) map(2) filter(4) map(3) filter(9) map(4) filter(16)
```

末端操作触发执行了所有的延期计算。

这个例子中另外一件值得注意的重要事情是计算执行的顺序。一个笨办法是先在每个元素上调用 map 函数，然后在结果序列的每个元素上再调用 filter 函数。map 和 filter 对集合就是这样做的，而序列不一样。对序列来说，所有操作是按顺序应用在每一个元素上：处理完第一个元素（先映射再过滤），然后完成第二个元素的处理，以此类推。

这种方法意味着部分元素根本不会发生任何变换，如果在轮到它们之前就已经取得了结果。我们来看一个 map 和 find 的例子。首先把一个数字映射成它的平方，然后找到第一个比数字 3 大的条目：

```
>>> println(listOf(1, 2, 3, 4).asSequence()
                              .map { it * it }.find { it > 3 })
4
```

如果同样的操作被应用在集合而不是序列上时，那么 map 的结果首先被求出来，即变换初始集合中的所有元素。第二步，中间集合中满足判断式的一个元素会被找出来。而对于序列来说，惰性方法意味着你可以跳过处理部分元素。图 5.8 阐明了这段代码两种求值方式之间的区别，一种是及早求值（使用集合），一种是惰性求值（使用序列）。

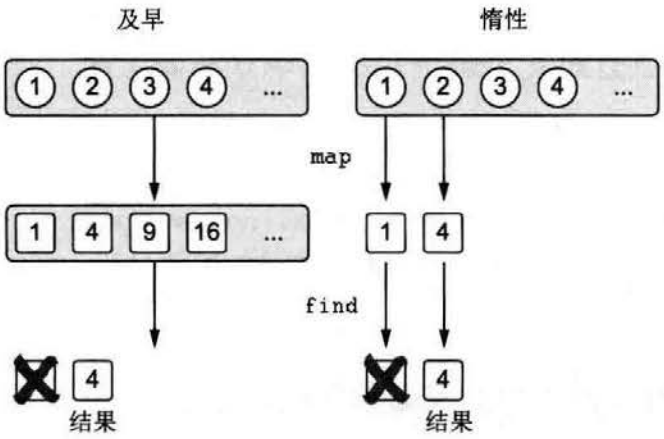


图 5.8 及早求值在整个集合上执行每个操作；惰性求值则逐个处理元素

第一种情况，当你使用集合的时候，列表被变换成了另一个列表，所以 map 变换应用到每一个元素上，包括了数字 3 和 4。然后，第一个满足判断式的元素被找到了：数字 2 的平方。

第二种情况，find 调用一开始就逐个地处理元素。从原始序列中取一个数字，用 map 变换它，然后再检查它是否满足传给 find 的判断式。当进行到数字 2 时，发现它的平方已经比数字 3 大，就把它作为 find 操作结果返回了。不再需要继续检查数字 3 和 4，因为这之前你已经找到了结果。

在集合上执行操作的顺序也会影响性能。假设你有一个人的集合，想要打印集合中那些长度小于某个限制的人名。你需要做两件事：把每个人映射成他们的名字，然后过滤掉其中那些不够短的名字。这种情况可以用任何顺序应用 map 和 filter 操作。两种顺序得到的结果一样，但它们应该执行的变换总次数是不一样的，如图 5.9 所示。

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31),
...         Person("Charles", 31), Person("Dan", 21))
>>> println(people.asSequence().map(Person::name)
...          .filter { it.length < 4 }.toList())
[Bob, Dan]
```

先“map”后“filter”

```
>>> println(people.asSequence().filter { it.name.length < 4 }
...         .map(Person::name).toList())
[Bob, Dan]
```

先“filter”后“map”

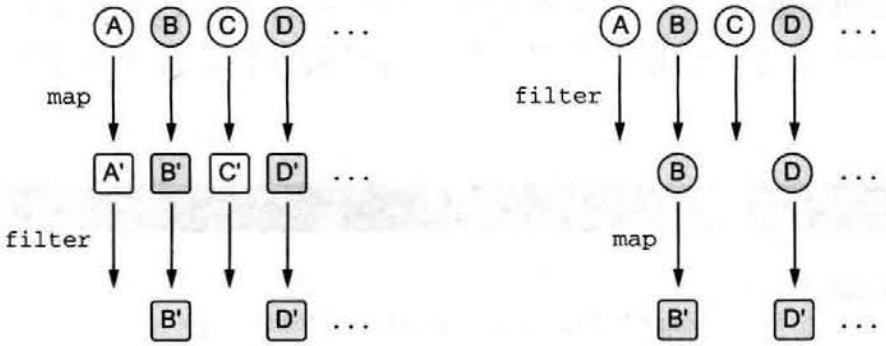


图 5.9 先应用 filter 有助于减少变换的总次数

如果 map 在前，每个元素都被变换。而如果 filter 在前，不合适的元素会被尽早地过滤掉且不会发生变换。

流 vs. 序列

如果你很熟悉 Java 8 中的流这个概念，你会发现序列就是它的翻版。Kotlin 提供了这个概念自己的版本，原因是 Java 8 的流并不支持那些基于 Java 老版本的平台，例如 Android。如果你的目标版本是 Java 8，流提供了一个 Kotlin 集合和序列目前还没有实现的重要特性：在多个 CPU 上并行执行流操作（比如 map 和 filter）的能力。可以根据 Java 的目标版本和你的特殊要求在流和序列之间做出选择。

5.3.2 创建序列

前面的例子都是使用同一个方法创建序列：在集合上调用 asSequence()。另一种可能性是使用 generateSequence 函数。给定序列中的前一个元素，这个函数会计算出下一个元素。下面这个例子就是如何使用 generateSequence 计算 100 以内所有自然数之和。

代码清单 5.12 生成并使用自然数序列

```
>>> val naturalNumbers = generateSequence(0) { it + 1 }
>>> val numbersTo100 = naturalNumbers.takeWhile { it <= 100 }
>>> println(numbersTo100.sum())
5050
```

当获取结果“sum”时，所有被推迟的操作都被执行

注意，这个例子中的 naturalNumbers 和 numbersTo100 都是有延期操作的序列。这些序列中的实际数字直到你调用末端操作(这里是 sum)的时候才会求值。

另一种常见的用例是父序列。如果元素的父元素和它的类型相同（比如人类或者 Java 文件），你可能会对它所有祖先组成的序列的特质感兴趣。下面这个例子可以查询文件是否放在隐藏目录中，通过创建一个其父目录的序列并检查每个目录的属性来实现。

代码清单 5.13 创建并使用父目录的序列

```
fun File.isInsideHiddenDirectory() =
        generateSequence(this) { it.parentFile }.any { it.isHidden }

>>> val file = File("/Users/svtk/.HiddenDir/a.txt")
>>> println(file.isInsideHiddenDirectory())
true
```

又一次，你生成了一个序列，通过提供第一个元素和获取每个后续元素的方式来实现。如果把 any 换成 find，你还可以得到想要的那个目录（对象）。注意，使用序列允许你找到需要的目录之后立即停止遍历父目录。

我们彻底地讨论了一个频繁出现的 lambda 表达式的应用：使用它们来简化集合操作。现在我们继续看下一个重要的主题：和现存的 Java API 一起使用 lambda。

5.4 使用Java函数式接口

和 Kotlin 库一起使用 lambda 感觉不错，但是你用到的大部分 API 很有可能还是用 Java 而不是 Kotlin 写的。好消息是 Kotlin 的 lambda 可以无缝地和 Java API 互操作。在这一节中，你将了解它是怎么回事。

在本章开头，你看过一个把 lambda 传给 Java 方法的例子：

```
button.setOnClickListener { /* 点击之后的动作 */ }
```

把 lambda 作为实参传递

Button 类通过接收类型为 OnClickListener 的实参的 setOnClickListener 方法给按钮设置一个新的监听器：

```
/* Java */
public class Button {
    public void setOnClickListener(OnClickListener l) { ... }
}
```

OnClickListener 接口只声明了一个方法，onClick：

```
/* Java */
public interface OnClickListener {
    void onClick(View v);
}
```

在 Java 中（Java 8 之前），你不得不创建一个匿名类的实例来作为实参传递给 setOnClickListener 方法：

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        ...
    }
}
```

在 Kotlin 中，可以传递一个 lambda，代替这个实例：

```
button.setOnClickListener { view -> ... }
```

这个 lambda 用来实现 OnClickListener，它有一个类型为 View 的参数，和 onClick 方法一样。图 5.10 展示了这个映射。

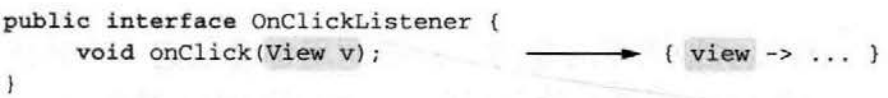


图 5.10 Lambda 的参数和方法参数对应

这种方式可以工作的原因是 OnClickListener 接口只有一个抽象方法。这种接口被称为函数式接口，或者 SAM 接口，SAM 代表单抽象方法。Java API 中随处可见像 Runnable 和 Callable 这样的函数式接口，以及支持它们的方法。Kotlin 允许你在调用接收函数式接口作为参数的方法时使用 lambda，来保证你的 Kotlin 代码既整洁又符合习惯。

注意 和 Java 不同，Kotlin 拥有完全的函数类型。正因为这样，需要接收 lambda 作为参数的 Kotlin 函数应该使用函数类型而不是函数式接口类型，作为这些参数的类型。Kotlin 不支持把 lambda 自动转换成实现 Kotlin 接口的对象。我们会在 8.1 节讨论在声明函数时函数类型的用法。

让我们先看看当你把一个 lambda 传给一个期望函数式接口类型实参的方法时到底会发生什么。

5.4.1 把 lambda 当作参数传递给 Java 方法

可以把 lambda 传给任何期望函数式接口的方法。例如，下面这个方法，它有一

个 Runnable 类型的参数：

```
/* Java */
void postponeComputation(int delay, Runnable computation);
```

在 Kotlin 中，可以调用它并把一个 lambda 作为实参传给它。编译器会自动把它转换成一个 Runnable 的实例：

```
postponeComputation(1000) { println(42) }
```

注意，当我们说“一个 Runnable 的实例”时，指的是“一个实现了 Runnable 接口的匿名类的实例”。编译器会帮你创建它，并使用 lambda 作为单抽象方法——这个例子中是 run 方法——的方法体。

通过显式地创建一个实现了 Runnable 的匿名对象也能达到同样的效果：

```
postponeComputation(1000, object : Runnable {
    override fun run() {
        println(42)
    }
})
```

把对象表达式作为函数式接口的实现传递

但是这里有一点不一样。当你显式地声明对象时，每次调用都会创建一个新的实例。使用 lambda 的情况不同：如果 lambda 没有访问任何来自定义它的函数的变量，相应的匿名类实例可以在多次调用之间重用：

```
postponeComputation(1000) { println(42) }
```

整个程序只会创建一个 Runnable 的实例

因此，完全等价的实现应该是下面这段代码中的显式 object 声明，它把 Runnable 实例存储在一个变量中，并且每次调用的时候都使用这个变量：

```
val runnable = Runnable { println(42) }
fun handleComputation() {
    postponeComputation(1000, runnable)
}
```

编译成全局变量；程序中仅此一个实例

每次 postponeComputation 调用时用的是一个对象

如果 lambda 从包围它的作用域中捕捉了变量，每次调用就不再可能重用同一个实例了。这种情况下，每次调用时编译器都要创建一个新对象，其中存储着被捕捉的变量的值。例如下面这个函数，每次调用都会使用一个新的 Runnable 实例，把 id 值存储在它的字段中：

```
fun handleComputation(id: String) {
    postponeComputation(1000) { println(id) }
}
```

lambda 会捕捉" id "这个变量

每次 handleComputation 调用时都创建一个 Runnable 的新实例

Lambda 的实现细节

自 Kotlin 1.0 起，每个 lambda 表达式都会被编译成一个匿名类，除非它是一个内联 lambda。后续版本计划支持生成 Java 8 字节码。一旦实现，编译器就可以避免为每一个 lambda 表达式都生成一个独立的 .class 文件。如果 lambda 捕捉了变量，每个被捕捉的变量会在匿名类中有对应的字段，而且每次（对 lambda 的）调用都会创建一个这个匿名类的新实例。否则，一个单例就会被创建。类的名称由 lambda 声明所在的函数名字称加上后缀衍生出来：这个例子中就是 HandleComputation$1。如果你反编译之前 lambda 表达式的代码，就会看到：

```
class HandleComputation$1(val id: String) : Runnable {
    override fun run() {
        println(id)
    }
}
fun handleComputation(id: String) {
    postponeComputation(1000, HandleComputation$1(id))
}
```

底层创建的是一个特殊类的实例，而不是一个 lambda

如你所见，编译器给每个被捕捉的变量生成了一个字段和一个构造方法参数。

请注意这里讨论的为 lambda 创建一个匿名类，以及该类的实例的方式只对期望函数式接口的 Java 方法有效，但是对集合使用 Kotlin 扩展方法的方式并不适用。如果你把 lambda 传给了标记成 `inline` 的 Kotlin 函数，是不会创建任何匿名类的。而大多数的库函数都标记成了 `inline`。这里的细节将在 8.2 节讨论。

如你所见，大多数情况下，从 lambda 到函数式接口实例的转换都是自动发生的，不需要你做什么。但是也存在需要显式地执行转换的情况。下面我们看看如何进行转换。

5.4.2 SAM 构造方法：显式地把 lambda 转换成函数式接口

*SAM 构造方法*是编译器生成的函数，让你执行从 lambda 到函数式接口实例的显式转换。可以在编译器不会自动应用转换的上下文中使用它。例如，如果有一个方法返回的是一个函数式接口的实例，不能直接返回一个 lambda，要用 SAM 构造方法把它包装起来。这里有一个简单的例子。

代码清单 5.14 使用 SAM 构造方法来返回值

```
fun createAllDoneRunnable(): Runnable {
    return Runnable { println("All done!") }
}

>>> createAllDoneRunnable().run()
All done!
```

SAM 构造方法的名称和底层函数式接口的名称一样。SAM 构造方法只接收一个参数——一个被用作函数式接口单抽象方法体的 lambda——并返回实现了这个接口的类的一个实例。

除了返回值外，SAM 构造方法还可以用在需要把从 lambda 生成的函数式接口实例存储在一个变量中的情况。假设你要在多个按钮上重用同一个监听器，就像下面的代码清单一样（在 Android 应用中，这段代码可以作为 `Activity.onCreate` 方法的一部分）。

代码清单 5.15 使用 SAM 构造方法来重用 `listener` 实例

```
val listener = OnClickListener { view ->
    val text = when (view.id) {
        R.id.button1 -> "First button"
        R.id.button2 -> "Second button"
        else -> "Unknown button"
    }
    toast(text)
}
button1.setOnClickListener(listener)
button2.setOnClickListener(listener)
```

使用 view.id 来判断点击的是哪个按钮

把“text”的值显示给用户

`listener` 会检查哪个按钮是点击的事件源并做出相应的行为。可以使用实现了 `OnClickListener` 的对象声明来定义监听器，但是 SAM 构造方法给你更简洁的选择。

Lambda 和添加 / 移除监听器

注意 lambda 内部没有匿名对象那样的 `this`：没有办法引用到 lambda 转换成的匿名类实例。从编译器的角度来看，lambda 是一个代码块，不是一个对象，而且也不能把它当成对象引用。Lambda 中的 `this` 引用指向的是包围它的类。

如果你的事件监听器在处理事件时还需要取消它自己，不能使用 lambda 这样做。这种情况使用实现了接口的匿名对象。在匿名对象内，`this` 关键字指向该对象实例，可以把它传给移除监听器的 API。

还有，尽管方法调用中的 SAM 转换一般都自动发生，但是当把 lambda 作为参数传给一个重载方法时，也有编译器不能选择正确的重载的情况。这时，使用显式的 SAM 构造方法是解决编译器错误的好方法。

我们来看看带接收者的 lambda，以及如何使用它们定义那些看起来就像内建结构的方便的库函数，来给 lambda 语法和用途的讨论收尾。

5.5 带接收者的lambda："with"与"apply"

这一节会展示 Kotlin 标准库中的 `with` 函数和 `apply` 函数。这是非常方便的两个函数，即便不了解它们是如何声明的，你也可以发现它们的很多用途。稍后在 11.2.1 节，你会看到如何声明类似的函数来满足自己的需求。而这一节的讲解将帮助你逐步熟悉 Kotlin 的 lambda 的独特功能：在 lambda 函数体内可以调用一个不同对象的方法，而且无须借助任何额外限定符；这种能力在 Java 中是找不到的。这样的 lambda 叫作*带接收者的 lambda*。我们从 `with` 函数开始，它就用到了带接收者的 lambda。

5.5.1 "with"函数

很多语言都有这样的语句，可以用它对同一个对象执行多次操作，而不需要反复把对象的名称写出来。Kotlin 也不例外，但它提供的是一个叫 `with` 的库函数，而不是某种特殊的语言结构。

要理解这种用法，我们先看看下面这个例子，稍后你会用 `with` 来重构它。

代码清单 5.16 构建字母表

```
fun alphabet(): String {
    val result = StringBuilder()
    for (letter in 'A'..'Z') {
        result.append(letter)
    }
    result.append("\nNow I know the alphabet!")
    return result.toString()
}
>>> println(alphabet())
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Now I know the alphabet!
```

上面这个例子中，你调用了 `result` 实例上好几个不同的方法，而且每次调用都要重复 `result` 这个名称。这里情况还不算太糟，但是如果你用到的表达式更长或者重复得更多，该怎么办？

下面的例子展示了如何使用 with 来重写这段代码。

代码清单 5.17 使用 with 构建字母表

```
fun alphabet(): String {
    val stringBuilder = StringBuilder()
    return with(stringBuilder) {
        for (letter in 'A'..'Z') {
            this.append(letter)
        }
        append("\nNow I know the alphabet!")
        this.toString()
    }
}
```

指定接收者的值，你会调用它的方法

通过显式的“this”来调用接收者值的方法

省掉“this”也可以调用方法

从 lambda 返回值

with 结构看起来像是一种特殊的语法结构，但它实际上是一个接收两个参数的函数：这个例子中两个参数分别是 stringBuilder 和一个 lambda。这里利用了把 lambda 放在括号外的约定，这样整个调用看起来就像是内建的语言功能。当然你可以选择把它写成 with(stringBuilder,{ ... })，但可读性就会差很多。

with 函数把它的第一个参数转换成作为第二个参数传给它的 lambda 的接收者。可以显式地通过 this 引用来访问这个接收者。或者，按照惯例，可以省略 this 引用，不用任何限定符直接访问这个值的方法和属性。

代码清单 5.17 中，this 指向了 stringBuilder，这是传给 with 的第一个参数。可以通过显式的 this 引用来访问 stringBuilder 的方法，就像 this.append(letter) 这样；也可以像 append("\nNow...") 这样直接访问。

带接收者的 lambda 和扩展函数

你可能回想起曾经见过的相似概念，this 指向的是函数接收者。在扩展函数体内部，this 指向了这个函数扩展的那个类型的实例，而且也可以被省略掉，让你直接访问接收者的成员。注意一个扩展函数某种意义上来说就是带接收者的函数。可以做下面的类比：

普通函数	普通 lambda
扩展函数	带接收者的 lambda

Lambda 是一种类似普通函数的定义行为的方式。而带接收者的 lambda 是类似扩展函数的定义行为的方式。

让我们进一步重构初始的 alphabet 函数，去掉额外的 stringBuilder 变量。

代码清单 5.18 使用 with 和一个表达式函数体来构建字母表

```
fun alphabet() = with(StringBuilder()) {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
    toString()
}
```

现在这个函数只返回一个表达式，所以使用表达式函数体语法重写了它。可以创建一个新的 StringBuilder 实例直接当作实参传给这个函数，然后在 lambda 中不需要显式的 this 就可以引用到这个实例。

方法名称冲突

如果你当作参数传给 with 的对象已经有这样的方法，该方法的名称和你正在使用 with 的类中的方法一样，怎么办？这种情况下，可以给 this 引用加上显式的标签来表明你要调用的是哪个方法。假设函数 alphabet 是类 OuterClass 的一个方法。如果你想引用的是定义在外部类的 toString 方法而不是 StringBuilder，可以用下面这种语法：

```
this@OuterClass.toString()
```

with 返回的值是执行 lambda 代码的结果，该结果就是 lambda 中的最后一个表达式（的值）。但有时候你想返回的是接收者对象，而不是执行 lambda 的结果。这时 apply 库函数就派上用场了。

5.5.2 "apply"函数

apply 函数几乎和 with 函数一模一样，唯一的区别是 apply 始终会返回作为实参传递给它的对象（换句话说，接收者对象）。让我们再一次重构 alphabet 函数，这一次用的是 apply。

代码清单 5.19 使用 apply 构建字母表

```
fun alphabet() = StringBuilder().apply {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}.toString()
```

apply被声明成一个扩展函数。它的接收者变成了作为实参的lambda的接收者。执行 apply 的结果是 StringBuilder，所以接下来你可以调用 toString 把它转换成 String。

许多情况下 apply 都很有效，其中一种是在创建一个对象实例并需要用正确的方式初始化它的一些属性的时候。在 Java 中，这通常是通过另外一个单独的 Builder 对象来完成的；而在 Kotlin 中，可以在任意对象上使用 apply，完全不需要任何来自定义该对象的库的特别支持。

通过下面这个使用一些自定义属性创建 Android TextView 的例子，来看看这种情况下 apply 是如何使用的。

代码清单 5.20 使用 apply 初始化一个 TextView

```
fun createViewWithCustomAttributes(context: Context) =
    TextView(context).apply {
        text = "Sample Text"
        textSize = 20.0
        setPadding(10, 0, 0, 0)
    }
```

apply 函数允许你使用紧凑的表达式函数体风格。新的 TextView 实例创建之后立即被传给了 apply。在传给 apply 的 lambda 中，TextView 实例变成了（lambda 的）接收者，你就可以调用它的方法并设置它的属性。Lambda 执行之后，apply 返回已经初始化过的接收者实例，它变成了 createViewWithCustomAttributes 函数的结果。

with 函数和 apply 函数是最基本和最通用的使用带接收者的 lambda 的例子。更多具体的函数也可以使用这种模式。例如，你可以使用标准库函数 buildString 进一步简化 alphabet 函数，它会负责创建 StringBuilder 并调用 toString。buildString 的实参是一个带接收者的 lambda，接收者就是 StringBuilder。

代码清单 5.21 使用 buildString 创建字母表

```
fun alphabet() = buildString {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}
```

buildString 函数优雅地完成了借助 StringBuilder 创建 String 的任务。

第 11 章中当我们开始讨论领域特定语言的时候，你还会看到更多有意思的例

子。带接收者的 lambda 是构建 DSL 的好工具，我们会向你展示如何使用它们来构建 DSL，以及如何定义你自己的函数，来调用带接收者的 lambda。

5.6 小结

- Lambda 允许你把代码块当作参数传递给函数。
- Kotlin 可以把 lambda 放在括号外传递给函数，而且可以用 `it` 引用单个的 lambda 参数。
- lambda 中的代码可以访问和修改包含这个 lambda 调用的函数中的变量。
- 通过在函数名称前加上前缀 `::`，可以创建方法、构造方法及属性的引用，并用这些引用代替 lambda 传递给函数。
- 使用像 `filter`、`map`、`all`、`any` 等函数，大多数公共的集合操作不需要手动迭代元素就可以完成。
- 序列允许你合并一个集合上的多次操作，而不需要创建新的集合来保存中间结果。
- 可以把 lambda 作为实参传给接收 Java 函数式接口（带单抽象方法的接口，也叫作 SAM 接口）作为形参的方法。
- 带接收者的 lambda 是一种特殊的 lambda，可以在这种 lambda 中直接访问一个特殊接收者对象的方法。
- `with` 标准库函数允许你调用同一个对象的多个方法，而不需要反复写出这个对象的引用。`apply` 函数让你使用构建者风格的 API 创建和初始化任何对象。

6 Kotlin 的类型系统

本章内容包括

- 处理 null 的可空类型和语法
- 基本数据类型和它们对应的 Java 类型
- Kotlin 的集合，以及它们和 Java 的关系

到目前为止，你已经见过了很大一部分实战中的 Kotlin 语法。你已经度过了使用 Kotlin 编写和 Java 等价的代码的阶段，已经准备好开始享受 Kotlin 那些让代码变得更紧凑、更易读的高效特性了。

我们先放慢一点速度，仔细观察一下 Kotlin 中最重要的一部分：类型系统。与 Java 相比，Kotlin 引入了一些新特性，它们是提升代码可读性的基本要素，比如：对可空的类型和只读集合的支持。与此同时，Kotlin 去掉了一些 Java 类型系统中不必要的或者有问题的特性，比如把数组作为头等公民来支持。让我们来看看这其中的细节。

6.1 可空性

可空性是 Kotlin 类型系统中帮助你避免 NullPointerException 错误的特性。作为一个程序的用户，你很可能见过像这样干巴巴的错误信息“An error has occurred: java.lang.NullPointerException”（发生了错误：java.lang.NullPointerException）。这条

信息还有另外一个版本“Unfortunately, the application X has stopped”（对不起，X 应用程序已停止），其背后的隐藏原因往往也是 NullPointerException。这样的错误让用户和开发者都觉得厌烦。

现代编程语言包括 Kotlin 解决这类问题的方法是把运行时的错误转变成编译期的错误。通过支持作为类型系统的一部分的可空性，编译器就能在编译期发现很多潜在的错误，从而减少运行时抛出异常的可能性。

这一节我们会讨论 Kotlin 中的可空类型：Kotlin 怎样表示允许为 null 的值，以及 Kotlin 提供的处理这些值的工具。除此之外，我们还要讨论混合使用 Kotlin 和 Java 代码时关于可空类型的细节。

6.1.1 可空类型

Kotlin 和 Java 的类型系统之间第一条也可能是最重要的一条区别是，Kotlin 对可空类型的显式的支持。这意味着什么呢？这是一种指出你的程序中哪些变量和属性允许为 null 的方式。如果一个变量可以为 null，对变量的方法的调用就是不安全的，因为这样会导致 NullPointerException。Kotlin 不允许这样的调用，因而可以阻止许多可能的异常。想搞清楚实践中这种方式是如何工作的，我们先看看下面的 Java 函数：

```
/* Java */
int strLen(String s) {
    return s.length();
}
```

这个函数是安全的吗？如果这个函数被调用的时候，传给它的是一个 null 实参，它就会抛出 NullPointerException。那么你是否需要在方法中增加对 null 的检查呢？这取决于使用该函数的意图。

我们试着用 Kotlin 重写这个函数。第一个必须回答的问题是，你期望这个函数被调用的时候传给它的实参可以为 null 吗？无论是语句 strLen(null) 中这样直接的字面值 null，还是任何在运行时可能为 null 的变量或者表达式。

如果你不期望这种情况发生，在 Kotlin 中要像下面这样声明这个函数：

```
fun strLen(s: String) = s.length
```

使用可能为 null 的实参调用 strLen 是不允许的，在编译期就会被标记成错误：

```
>>> strLen(null)
ERROR: Null can not be a value of a non-null type String
```

这个函数中的参数被声明成 String 类型，在 Kotlin 中这表示它必须包含一个 String 实例。这一点由编译器强制实施，所以你不能传给它一个包含 null 的实参。这样就保证了 strLen 函数永远不会在运行时抛出 NullPointerException。

如果你允许调用这个方法的时候传给它所有可能的实参，包括那些可以为 null 的实参，需要显式地在类型名称后面加上问号来标记它：

```
fun strLenSafe(s: String?) = ...
```

问号可以加在任何类型的后面来表示这个类型的变量可以存储 null 引用：String?、Int?、MyCustomType?，等等（如图 6.1 所示）。

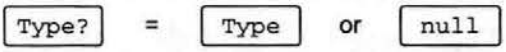


图 6.1 可空类型的变量可以存储 null 引用

重申一下，没有问号的类型表示这种类型的变量不能存储 null 引用。这说明所有常见类型默认都是非空的，除非显式地把它标记为可空。

一旦你有一个可空类型的值，能对它进行的操作也会受到限制。例如，不能再调用它的方法：

```
>>> fun strLenSafe(s: String?) = s.length()
ERROR: only safe (?.) or non-null asserted (!!.) calls are allowed
 on a nullable receiver of type kotlin.String?
```

也不能把它赋值给非空类型的变量：

```
>>> val x: String? = null
>>> var y: String = x
ERROR: Type mismatch: inferred type is String? but String was expected
```

也不能把可空类型的值传给拥有非空类型参数的函数：

```
>>> strLen(x)
ERROR: Type mismatch: inferred type is String? but String was expected
```

那么你可以对它做什么呢？最重要的操作就是和 null 进行比较。而且一旦你进行了比较操作，编译器就会记住，并且在这次比较发生的作用域内把这个值当作非空来对待。例如，下面这段代码是完全合法的：

代码清单 6.1 使用 if 检查处理 null

```
fun strLenSafe(s: String?): Int =
    if (s != null) s.length else 0

>>> val x: String? = null
```

增加了 null 检查后，这段代码就可以编译了

```
>>> println(strLenSafe(x))
0
>>> println(strLenSafe("abc"))
3
```

如果 if 检查只是唯一处理可空性的工具，你的代码很快将会变得冗长。幸运的是，Kotlin 还提供了其他一些工具来帮助我们用更简洁的方式来处理可空值。在了解这些工具之前，我们先花些时间讨论一下可空性的含义，以及什么是变量类型。

6.1.2 类型的含义

我们先思考一下最普遍的问题：什么是类型，为什么变量拥有类型？维基百科上这篇关于类型的条目（http://en.wikipedia.org/wiki/Data_type）给出了相当不错的答案："类型就是数据的分类……决定了该类型可能的值，以及该类型的值上可以完成的操作。"

我们试试在 Java 的一些类型上套用这个定义，先从 double 类型开始。正如你知道的，double 是 64 位的双精度浮点数。可以对 double 类型的值进行标准的算术运算，所有的功能都可以一视同仁地运用到所有 double 类型的值上。因此，如果你有一个类型为 double 的变量，那么你就能确定编译器允许的该变量值上的任何操作，都可以被成功地执行。

现在我们把它和 String 类型的变量对比一下。在 Java 中，这样的变量可以持有两种值，分别是 String 的实例和 null。这两种值完全不一样：就连 Java 自己的 instanceof 运算符都会告诉你 null 不是 String。这两种值的操作也完全不一样：真实的 String 实例允许你调用它的任何方法，而 null 值只允许非常有限的操作。

这说明 Java 的类型系统在这种情况下不能很好地工作。即使变量拥有声明的类型 String 你依然无法知道能对该变量的值做些什么，除非做额外的检查。你往往会跳过这些检查，因为你觉得你了解程序中大概的数据流动，并确定在某个点上这个值不可能为 null。有时候你想错了，而你的程序就会因为 NullPointerException 而崩溃。

其他应对 NullPointerException 错误的方法

Java 有一些帮助解决 NullPointerException 问题的工具。比如，有些人会使用注解（如 @Nullable 和 @NotNull）来表达值的可空性。有些工具（例如，IntelliJ IDEA 内置的代码检查）可以利用这些注解来发现可能抛出 NullPointerException 的位置。但这些工具都不是标准 Java 编译过程的一部分，所以很难保证它们自始至终都被应用。而且在整个代码库中（包括项

目用到的库）很难使用注解标记所有可能发生错误的地方，让它们能被探测到。我们 JetBrains 自己的经验显示，就算在 Java 代码中大面积地使用可空性注解依然不能彻底地解决 NPE 的问题。

另外一种解决这个问题的思路是永远不要在代码中使用 null 值，而是使用像 Java 8 中引入的 Optional 类型这样的特殊包装类型，来表示这个值可能没有被定义。这种方式有一些缺点：代码变得更冗长，额外的包装接口还会影响运行时的性能，在整个生态圈中也没有被广泛地采用。即使在代码中到处都使用了 Optional，仍然需要处理 JDK、Android 框架，以及其他第三方库中的方法返回的 null 值。

Kotlin 的可空类型为这类问题提供了全面的解决方案。区分开可空类型和非空类型使事情变得明朗：哪些对值的操作是允许的，哪些操作有会导致运行时异常并因此被禁止。

注意 可空的和非空的对象在运行时没有什么区别；可空类型并不是非空类型的包装。所有的检查都发生在编译期。这意味着使用 Kotlin 的可空类型并不会在运行时带来额外的开销。

现在我们来看看在 Kotlin 中如何使用可空类型，以及为什么和它们打交道并不令人厌烦。我们从能够安全访问可空值的特殊运算符说起。

6.1.3 安全调用运算符："?."

Kotlin 的弹药库中最有效的一种工具就是安全调用运算符：?.，它允许你把一次 null 检查和一次方法调用合并成一个操作。例如，表达式 s?.toUpperCase() 等同于下面这种烦琐的写法：if (s!=null) s.toUpperCase() else null。

换句话说，如果你试图调用一个非空值的方法，这次方法调用会被正常地执行。但如果值是 null，这次调用不会发生，而整个表达式的值为 null，如图 6.2 所示。

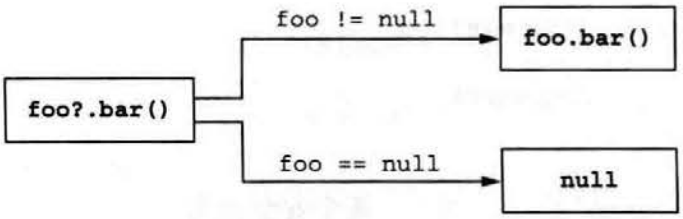


图 6.2 安全调用运算符只会调用非空值的方法

注意，这次调用的结果类型也是可空的。尽管 String.toUpperCase() 会返回 String 类型的值，但 s 是可空的时候，表达式 s?.toUpperCase() 的结果

类型是 String?：

```
fun printAllCaps(s: String?) {
    val allCaps: String? = s?.toUpperCase()
    println(allCaps)
}
>>> printAllCaps("abc")
ABC
>>> printAllCaps(null)
null
```

allCaps 可能是 null

安全调用不光可以调用方法，也能用来访问属性。下面这个例子展示了一个具有可空属性的简单 Kotlin 类，以及访问这个属性时安全调用运算符的用法。

代码清单 6.2 使用安全调用处理可空属性

```
class Employee(val name: String, val manager: Employee?)

fun managerName(employee: Employee): String? = employee.manager?.name

>>> val ceo = Employee("Da Boss", null)
>>> val developer = Employee("Bob Smith", ceo)
>>> println(managerName(developer))
Da Boss
>>> println(managerName(ceo))
null
```

如果你的对象图中有多个可空类型的属性，通常可以在同一个表达式中方便地使用多个安全调用。假如你要使用不同的类来保存关于个人的信息、他们的公司，以及公司的地址，而公司和地址都可以省略。使用 ?. 运算符，不需要任何额外的检查，就可以在一行代码中访问到 Person 的 country 属性。

代码清单 6.3 链接多个安全调用

```
class Address(val streetAddress: String, val zipCode: Int,
              val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun Person.countryName(): String {
   val country = this.company?.address?.country
   return if (country != null) country else "Unknown"
}
>>> val person = Person("Dmitry", null)
>>> println(person.countryName())
Unknown
```

多个安全调用链接在一起

带 null 检查的方法调用序列在 Java 代码中太常见了，现在你看到了 Kotlin 可以让它们变得更简洁。但是代码清单 6.3 中还有不必要的重复代码：你用一个值和 null 比较，如果这个值不为空就返回这个值，否则返回其他的值。接下来我们看看 Kotlin 是否能帮助去掉这些重复代码。

6.1.4 Elvis 运算符："?:"

Kotlin 有方便的运算符来提供代替 null 的默认值。它被称作 *Elvis 运算符*（或者 *null 合并运算符*，如果你喜欢听起来更严肃的名称）。它看起来就像这样：?:（把书顺时针旋转 90°，它看起来就像猫王 Elvis 一样，有图有真相：https://i.stack.imgur.com/hQlrps.png）。下面展示了它是如何使用的：

```
fun foo(s: String?) {
    val t: String = s ?: ""
}
```

如果"s"为 null，结果是一个空的字符串

Elvis 运算符接收两个运算数，如果第一个运算数不为 null，运算结果就是第一个运算数；如果第一个运算数为 null，运算结果就是第二个运算数。图 6.3 展示了它是如何运行的。

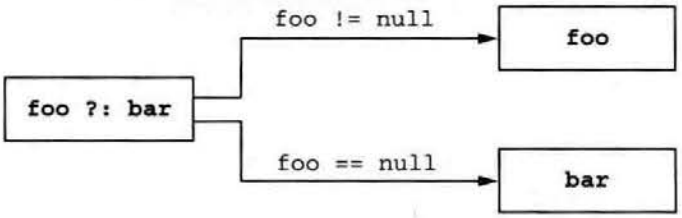


图 6.3 Elvis 运算符用其他值代替 null

Elvis 运算符经常和安全调用运算符一起使用，用一个值代替对 null 对象调用方法时返回的 null。下面展示了如何使用这种模式来简化代码清单 6.1。

代码清单 6.4 使用 Elvis 运算符处理 null 值

```
fun strLenSafe(s: String?): Int = s?.length ?: 0

>>> println(strLenSafe("abc"))
3
>>> println(strLenSafe(null))
0
```

代码清单 6.3 中的函数 countryName 现在也可以使用一行代码完成。

```
fun Person.countryName() =
    company?.address?.country ?: "Unknown"
```

在 Kotlin 中有一种场景下 Elvis 运算符会特别顺手，像 return 和 throw 这样的操作其实是表达式，因此可以把它们写在 Elvis 运算符的右边。这种情况下，如果 Elvis 运算符左边的值为 null，函数就会立即返回一个值或者抛出一个异常。如果函数中需要检查先决条件，这个方式特别有用。

我们来看看如何使用这个运算符来实现一个打印包含个人公司地址的出货标签的函数。下面这个代码清单重复了所有类的声明——Kotlin 中它们是如此简洁，这不是一个问题。

代码清单 6.5 同时使用 throw 和 Elvis 运算符

```
class Address(val streetAddress: String, val zipCode: Int,
              val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun printShippingLabel(person: Person) {
    val address = person.company?.address
      ?: throw IllegalArgumentException("No address")
    with (address) {
        println(streetAddress)
        println("$zipCode $city, $country")
    }
}
```

如果缺少 address 就抛出异常

“address”不为空

```
>>> val address = Address("Elsestr. 47", 80687, "Munich", "Germany")
>>> val jetbrains = Company("JetBrains", address)
>>> val person = Person("Dmitry", jetbrains)

>>> printShippingLabel(person)
Elsestr. 47
80687 Munich, Germany

>>> printShippingLabel(Person("Alexey", null))
java.lang.IllegalArgumentException: No address
```

如果一切正常，函数 printShippingLabel 会打印出标签。如果地址不存在，它不会只是抛出一个带行号的 NullPointerException，相反，它会报告一个有意义的错误。如果地址存在，标签会包含街道地址、邮编、城市和国家。留意前一章中见过的 with 函数是如何被用来避免在这一行中重复四次 address 的。

现在，你了解了 Kotlin 进行“if 非空”检查的方式，我们接下来介绍 Kotlin 中 instanceof 检查的安全版本：常常和安全调用及 Elvis 运算符一起出现的安全转换运算符。

6.1.5 安全转换："as?"

在第 2 章中，我们学习了用来转换类型的常规 Kotlin 运算符：as 运算符。和常规的 Java 类型转换一样，如果被转换的值不是你试图转换的类型，就会抛出 ClassCastException 异常。当然可以结合 is 检查来确保这个值拥有合适的类型。但是作为一种安全简洁的语言，Kotlin 没有更优雅的解决方案吗？当然有。

as? 运算符尝试把值转换成指定的类型，如果值不是合适的类型就返回 null，如图 6.4 所示。

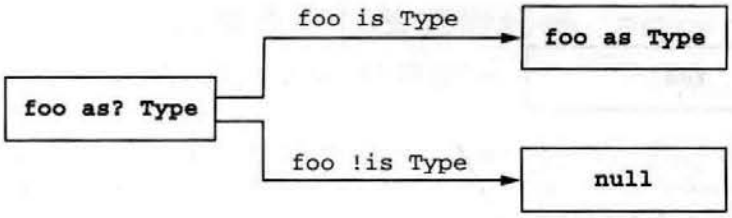


图 6.4 安全转换运算符尝试把值转换成给定的类型，如果类型不合适就返回 null

一种常见的模式是把安全转换和 Elvis 运算符结合使用。例如，实现 equals 方法的时候这样的用法非常方便。

代码清单 6.6 使用安全转换实现 equals

```
class Person(val firstName: String, val lastName: String) {
    override fun equals(o: Any?): Boolean {
        val otherPerson = o as? Person ?: return false

        return otherPerson.firstName == firstName &&
                otherPerson.lastName == lastName
    }

    override fun hashCode(): Int =
        firstName.hashCode() * 37 + lastName.hashCode()
}

>>> val p1 = Person("Dmitry", "Jemerov")
>>> val p2 = Person("Dmitry", "Jemerov")
>>> println(p1 == p2)
true
>>> println(p1.equals(42))
false
```

检查类型，如果不匹配就返回 false

在安全转换之后，变量 otherPerson 被智能地转换为 Person 类型

== 运算符会调用“equals”方法

使用这种模式，可以非常容易地检查实参是否是适当的类型，转换它，并在它的类型不正确时返回 false，而且这些操作全部在同一个表达式中。当然，这种场景下智能转换也会生效：当你检查过类型并拒绝了 null 值，编译器就确定了变量 otherPerson 值的类型是 Person 并让你能够相应地使用它。

安全调用、安全转换和 Elvis 运算符都非常有用，它们出现在 Kotlin 代码中的频率非常高。但有时你并不需要 Kotlin 的这些支持来处理 null 值，你只需要直接告诉编译器这个值实际上并不是 null。接下来我们看看你如何做到这一点。

6.1.6 非空断言："!!"

非空断言是 Kotlin 提供给你的最简单直率的处理可空类型值的工具。它使用双感叹号表示，可以把任何值转换成非空类型。如果对 null 值做非空断言，则会抛出异常。图 6.5 展示了这个逻辑。

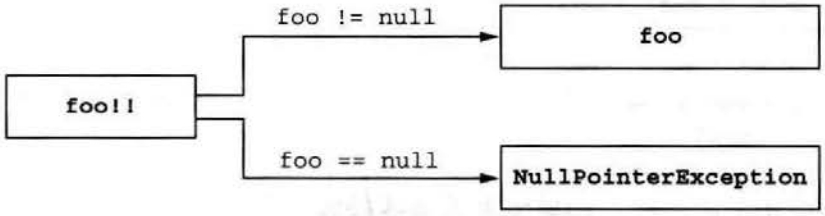


图 6.5 通过使用非空断言，如果值为 null，可以显式地抛出异常

下面这个小例子是一个函数，它使用这种断言来把可空的实参转换成非空。

代码清单 6.7 使用非空断言

```
fun ignoreNulls(s: String?) {
    val sNotNull: String = s!!
    println(sNotNull.length)
}
```

异常指向这一行

```
>>> ignoreNulls(null)
Exception in thread "main" kotlin.KotlinNullPointerException
    at <...>.ignoreNulls(07_NotnullAssertions.kt:2)
```

如果上面函数中 s 为 null 会发生什么？Kotlin 没有其他选择，它会在运行时抛出一个异常（一种特殊的 NullPointerException）。但是注意异常抛出的位置是非空断言所在的那一行，而不是接下来试图使用那个值的一行。本质上，你在告诉编译器："我知道这个值不为 null，如果我错了我准备好了接收这个异常。"

注意 你可能注意到双感叹号看起来有点儿粗暴：就像你在冲着编译器咆哮。这是有意为之。Kotlin 的设计者试图说服你思考更好的解决方案，这些方案不会使用断言这种编译器无法验证的方式。

但是确实存在这样的情况，某些问题适合用非空断言来解决。当你在一个函数中检查一个值是否为 null，而在另一个函数中使用这个值时，这种情况下编译器无法识别这种用法是否安全。如果你确信这样的检查一定在其他某个函数中存在，

你可能不想在使用这个值之前重复检查，这时你就可以使用非空断言。

实践中在许多像 Swing 这样的 UI 框架中出现的 action 类会发生这种情况。在一个 action 类中，存在不同的方法分别更新 action 的状态（启用或是禁止它）和执行 action。在 update 方法中执行的检查确保了 execute 方法在条件不满足时并不会被调用，这一点编译器是无法知道的。

让我们看一个 Swing Action 的例子，这种情况下该 action 使用了非空断言。CopyRowAction 期望把一个列表中被选中的行拷贝到剪贴板中。我们省略了全部无谓的细节，只保留了负责检查是否有行被选中（意味着 action 可以执行）的代码和获取被选中行的值的代码。Action API 暗示 actionPerformed 只会在 isEnabled 为 true 时被调用。

代码清单 6.8 在 Swing action 中使用非空断言

```
class CopyRowAction(val list: JList<String>) : AbstractAction() {
    override fun isEnabled(): Boolean =
        list.selectedValue != null

    override fun actionPerformed(e: ActionEvent) {
        val value = list.selectedValue!!
        // copy value to clipboard
    }
}
```

只会在 isEnabled 返回“true”时被调用

注意，这种情况下如果你不想使用 !!，还可以这样写来获取一个非空类型的值：val value = list.selectedValue ?: return。如果你使用这种写法，list.selectedValue 的可空值会导致函数提前返回，因此 value 永远都是非空的。尽管这里使用 Elvis 运算符进行非空检查是多余的，但是如果 isEnabled 将来变得更复杂，这可能是一种有效的保护。

还有一个需要牢记的注意事项：当你使用 !! 并且它的结果是异常时，异常调用栈的跟踪信息只表明异常发生在哪一行代码，而不会表明异常发生在哪一个表达式。为了让跟踪信息更清晰精确地表示哪个值为 null，最好避免在同一行中使用多个 !! 断言：

```
person.company!!.address!!.country
```

不要写这样的代码！

如果上面这一行代码中发生了异常，你不能分辨出到底 company 的值为 null，还是 address 的值为 null。

到目前为止，我们讨论的都是如何访问可空类型的值。但是如果你要将一个可空值作为实参传递一个只接收非空值的函数时，应该怎么办？编译器不允许在没有

检查的情况下这样做，因为这样不安全。Kotlin 语言并没有对这种使用场景的特殊支持，但是标准库函数可以帮到你：这个函数叫作 let。

6.1.7 “let”函数

let 函数让处理可空表达式变得更容易。和安全调用运算符一起，它允许你对表达式求值，检查求值结果是否为 null，并把结果保存为一个变量。所有这些动作都在同一个简洁的表达式中。

可空参数最常见的一种用法应该就是被传递给一个接收非空参数的函数。比如说下面这个 sendEmailTo 函数，它接收一个 String 类型的参数并向这个地址发送一封邮件。这个函数在 Kotlin 中是这样写的，它需要一个非空的参数：

```
fun sendEmailTo(email: String) { /*...*/ }
```

不能把可空类型的值传给这个函数：

```
>>> val email: String? = ...
>>> sendEmailTo(email)
ERROR: Type mismatch: inferred type is String? but String was expected
```

必须显式地检查这个值不为 null：

```
if (email != null) sendEmailTo(email)
```

但你还有另外一种处理方式：使用 let 函数，并通过安全调用来调用它。let 函数做的所有事情就是把一个调用它的对象变成 lambda 表达式的参数。如果结合安全调用语法，它能有效地把调用 let 函数的可空对象，转变成非空类型（如图 6.6 所示）。

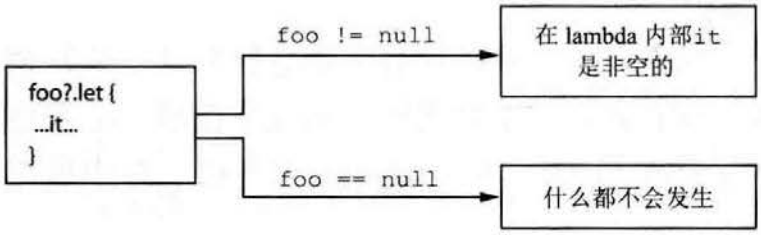


图 6.6 安全调用“let”只在表达式不为 null 时执行 lambda

let 函数只在 email 的值非空时才被调用，所以你就能在 lambda 中把 email 当作非空的实参使用。

```
email?.let { email -> sendEmailTo(email) }
```

使用自动生成的名字 it 这种简明语法之后，上面的代码就更短了：email?.let{ sendEmailTo(it) }。下面这个更完整的例子展示了这种模式。

代码清单 6.9 使用 let 调用一个接受非空参数的函数

```
fun sendEmailTo(email: String) {
    println("Sending email to $email")
}

>>> var email: String? = "yole@example.com"
>>> email?.let { sendEmailTo(it) }
Sending email to yole@example.com
>>> email = null
>>> email?.let { sendEmailTo(it) }
```

注意，如果有一些很长的表达式结果不为 null，而你又要使用这些结果时，let 表示法特别方便。这种情况下你不必创建一个单独的变量。对比一下显式的 if 检查

```
val person: Person? = getTheBestPersonInTheWorld()
if (person != null) sendEmailTo(person.email)
```

和功能相同但没有额外变量的代码：

```
getTheBestPersonInTheWorld()?.let { sendEmailTo(it.email) }
```

这个函数返回 null，所以 lambda 中的代码永远不会执行：

```
fun getTheBestPersonInTheWorld(): Person? = null
```

当你需要检查多个值是否为 null 时，可以用嵌套的 let 调用来处理。但在大多数情况下，这种代码相当啰唆又难以理解。用普通的 if 表达式来一次性检查所有值通常更简单。

另一种常见的情况是，属性最终是非空的，但不能使用非空值在构造方法中初始化。接下来我们看看 Kotlin 如何让你能够处理这种情况。

6.1.8 延迟初始化的属性

很多框架会在对象实例创建之后用专门的方法来初始化对象。例如，在 Android 中，Activity 的初始化就发生在 onCreate 方法中。而 JUnit 则要求你把初始化的逻辑放在用 @Before 注解的方法中。

但是你不能在构造方法中完全放弃非空属性的初始化器，仅仅在一个特殊的方法里初始化它。Kotlin 通常要求你在构造方法中初始化所有属性，如果某个属性是非空类型，你就必须提供非空的初始化值。否则，你就必须使用可空类型。如果你这样做，该属性的每一次访问都需要 null 检查或者 !! 运算符。

代码清单 6.10 使用非空断言访问可空属性

```
class MyService {
    fun performAction(): String = "foo"
}

class MyTest {
    private var myService: MyService? = null

    @Before fun setUp() {
        myService = MyService()
    }

    @Test fun testAction() {
        Assert.assertEquals("foo",
            myService!!.performAction())
    }
}
```

声明一个可空类型的属性并初始化为 null

在 setUp 方法中提供真正的初始化器

必须注意可空性：要么用 !!, 要么用 ?.

这段代码很难看，尤其是你要反复使用这个属性的时候。为了解决这个问题，可以把 myService 属性声明成可以*延迟初始化*的，使用 lateinit 修饰符来完成这样的声明。

代码清单 6.11 使用延迟初始化属性

```
class MyService {
    fun performAction(): String = "foo"
}

class MyTest {
    private lateinit var myService: MyService

    @Before fun setUp() {
        myService = MyService()
    }

    @Test fun testAction() {
        Assert.assertEquals("foo",
            myService.performAction())
    }
}
```

声明一个不需要初始化器的非空类型的属性

像之前的例子一样在 setUp 方法中初始化属性

不需要 null 检查直接访问属性

注意，延迟初始化的属性都是 var，因为需要在构造方法外修改它的值，而 val 属性会被编译成必须在构造方法中初始化的 final 字段。尽管这个属性是非空类型，但是你不需要在构造方法中初始化它。如果在属性被初始化之前就访问了它，会得到这个异常“lateinit property myService has not been initialized”(lateinit 的属性 myService 没有被初始化)。该异常清楚地说明了发生了什么，

比一般的 NullPointerException 要容易理解得多。

> **注意** lateinit 属性常见的一种用法是依赖注入。在这种情况下，lateinit 属性的值是被依赖注入框架从外部设置的。为了保证和各种 Java（依赖注入）框架的兼容性，Kotlin 会自动生成一个和 lateinit 属性具有相同可见性的字段。如果属性的可见性是 public，生成字段的可见性也是 public。

现在我们看看如何通过定义可空类型的扩展函数，来丰富 Kotlin 处理 null 值的工具集。

6.1.9 可空类性的扩展

为可空类型定义扩展函数是一种更强大的处理 null 值的方式。可以允许接收者为 null 的（扩展函数）调用，并在该函数中处理 null，而不是在确保变量不为 null 之后再调用它的方法。只有扩展函数才能做到这一点，普通成员方法的调用是通过对象实例来分发的，因此实例为 null 时（成员方法）永远不能被执行。

Kotlin 标准库中定义的 String 的两个扩展函数 isEmpty 和 isBlank 就是这样的例子。第一个函数判断字符串是否是一个空的字符串 ""。第二个函数则判断它是否是空的或者它只包含空白字符。通常用这些函数来检查字符串是有价值的，以确保对它的操作是有意义的。你可能意识到了，像处理无意义的空字符串和空白字符串这样处理 null 也很有用。事实上，你的确可以这样做：函数 isEmptyOrNull 和 isNullOrBlank 就可以由 String? 类型的接收者调用。

代码清单 6.12 用可空接收者调用扩展函数

```
fun verifyUserInput(input: String?) {
    if (input.isNullOrBlank()) {                    <-- 这里不需要安全调用
        println("Please fill in the required fields")
    }
}

>>> verifyUserInput(" ")
Please fill in the required fields
>>> verifyUserInput(null)
Please fill in the required fields
```

这个接收者调用 isNullOrBlank 并不会导致任何异常

不需要安全访问，可以直接调用为可空接收者声明的扩展函数（如图 6.7 所示）。这个函数会处理可能的 null 值。

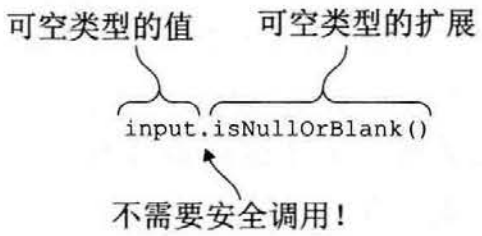


图 6.7 不需要安全调用就可以访问可空类型的扩展

函数 isNullOrBlank 显式地检查了 null，这种情况下返回 true，然后调用 isBlank，它只能在非空 String 上调用：

```
fun String?.isNullOrBlank(): Boolean =
        this == null || this.isBlank()
```

可空字符串的扩展

第二个“this”使用了智能转换

当你为一个可空类型（以？结尾）定义扩展函数时，这意味着你可以对可空的值调用这个函数；并且函数体中的 this 可能为 null，所以你必须显式地检查。在 Java 中，this 永远是非空的，因为它引用的是当前你所在这个类的实例。而在 Kotlin 中，这并不永远成立：在可空类型的扩展函数中，this 可以为 null。

注意我们之前讨论的 let 函数也能被可空的接收者调用，但它并不检查值是否为 null。如果你在一个可空类型直接上调用 let 函数，而没有使用安全调用运算符，lambda 的实参将会是可空的：

```
>>> val person: Person? = ...
>>> person.let { sendEmailTo(it) }
ERROR: Type mismatch: inferred type is Person? but Person was expected
```

没有安全调用，所以“it”是可空类型

因此，如果想要使用 let 来检查非空的实参，你就必须使用安全调用运算符 ?.，就像之前看到的代码一样：person?.let { sendEmailTo(it) }。

> **注意** 当你定义自己的扩展函数时，需要考虑该扩展是否需要为可空类型定义。默认情况下，应该把它定义成非空类型的扩展函数。如果发现大部分情况下需要在可空类型上使用这个函数，你可以稍后再安全地修改它（不会破坏其他代码）。

这一节展示了一些意外的状况。如果你没有使用额外的检查来解引用一个变量，比如 s.isNullOrBlank()，它并不会立即意味着变量是非空的：这个函数有可能是非空类型的扩展函数。下一节，我们还会讨论另外一种让你惊讶的情况：即使不用问号结尾，类型参数也能是可空的。

6.1.10 类型参数的可空性

Kotlin 中所有泛型类和泛型函数的类型参数默认都是可空的。任何类型，包括可空类型在内，都可以替换类型参数。这种情况下，使用类型参数作为类型的声明都允许为 null，尽管类型参数 T 并没有用问号结尾。参考下面这个例子。

代码清单 6.13 处理可空的类型参数

```
fun <T> printHashCode(t: T) {
    println(t?.hashCode())
}
>>> printHashCode(null)
null
```

因为“t”可能为 null，所以必须使用安全调用

“T”被推导成“Any?”

在 printHashCode 调用中，类型参数 T 推导出的类型是可空类型 Any?。因此，尽管没有用问号结尾，实参 t 依然允许持有 null。

要使类型参数非空，必须要为它指定一个非空的上界，那样泛型会拒绝可空值作为实参。

代码清单 6.14 为类型参数声明非空上界

```
fun <T: Any> printHashCode(t: T) {
    println(t.hashCode())
}
>>> printHashCode(null)
Error: Type parameter bound for `T` is not satisfied
>>> printHashCode(42)
42
```

现在“T”就不是可空的

这段代码是无法编译的：你不能传递 null，因为期望的是非空值

第 9 章会介绍 Kotlin 中的泛型，而在 9.1.4 节中会介绍这个话题的更多细节。

注意必须使用问号结尾来标记类型为可空的，没有问号就是非空的。类型参数是这个规则唯一的例外。下一节展示了另外一种可空性的特例：来自 Java 代码的类型。

6.1.11 可空性和 Java

之前的讨论介绍的都是在 Kotlin 世界里和 null 打交道的工具。但是 Kotlin 引以为豪的是和 Java 的互操作性，而你知道 Java 的类型系统是不支持可空性的。那么当你混合使用 Kotlin 和 Java 时会发生什么？会不会失去所有的安全性？或者每个值都必须检查是否为 null？还有更好的办法吗？让我们来弄清楚。

首先，正如之前所述，有些时候 Java 代码包含了可空性的信息，这些信息使

用注解来表达。当代码中出现了这样的信息时，Kotlin 就会使用它。因此 Java 中的 `@Nullable String` 被 Kotlin 当作 `String?`，而 `@NotNull String` 就是 `String`（如图 6.8 所示）。

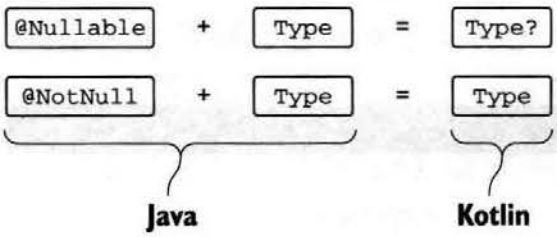


图 6.8 根据 Java 类型的注解，Java 类型会在 Kotlin 中表示为可空类型和非空类型

Kotlin 可以识别多种不同风格的可空性注解，包括 JSR-305 标准的注解（在 `javax.annotation` 包之中）、Android 的注解（`android.support.annotation`）和 JetBrains 工具支持的注解（`org.jetbrains.annotations`）。这里有一个有意思的问题，如果这些注解不存在会发生什么？这种情况下，Java 类型会变成 Kotlin 中的平台类型。

平台类型

平台类型本质上就是 Kotlin 不知道可空性信息的类型。既可以把它当作可空类型处理，也可以当作非空类型处理（如图 6.9 所示）。这意味着，你要像在 Java 中一样，对你在这个类型上做的操作负有全部责任。编译器将会允许所有的操作，它不会把对这些值的空安全操作高亮成多余的，但它平时却是这样对待非空类型值上的空安全操作的。如果你认为这个值为 `null`，在使用它之前可以用它和 `null` 比较。如果你认为它不为 `null`，就直接使用它。就像在 Java 中一样，如果你错误地理解了这个值，使用的时候就会遇到 `NullPointerException`。

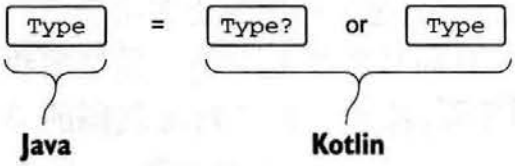


图 6.9 Java 类型在 Kotlin 中表示为平台类型，既可以把它当作可空类型也可以当作非空类型来处理

比方说，我们在 Java 中定义了一个 `Person` 类。

代码清单 6.15 没有可空性注解的 Java 类

```
/* Java */
public class Person {
    private final String name;
```

```
    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

getName 能不能返回 null？这种情况下 Kotlin 编译器完全不知道 String 类型的可空性，所以你必须自己处理它。如果你确定 name 不为 null，就可以像 Java 中一样按照通常的方式对它解引用，不需要额外的检查。但是这种情况下请准备好接收异常。

代码清单 6.16 不使用 null 检查访问 Java 类

```
fun yellAt(person: Person) {
    println(person.name.toUpperCase() + "!!!")
}

>>> yellAt(Person(null))
java.lang.IllegalArgumentException: Parameter specified as non-null
 is null: method toUpperCase, parameter $receiver
```

toUpperCase() 调用的接收者 person.name 为 null，所以这里会抛出异常

注意，这里你看到的不是一个干巴巴的 NullPointerException，而是一条更详细的错误消息，告诉你方法 toUpperCase 不能在 null 的接收者上调用。

事实上，对于公有的 Kotlin 函数，编译器会生成对每个非空类型的参数（和接收者）的检查，所以，使用不正确的参数的调用尝试都会立即被报告为异常。注意，这种值检查在函数调用的时候就执行了，而不是等到这些参数被使用的时候。这确保了不正确的调用会被尽早发现，那些由于 null 值被传给代码不同层次的多个函数之后，并被这些函数访问时而产生难以理解的异常就能被避免。

另外一个选择是把 getName() 的返回类型解释为可空的并安全地访问它。

代码清单 6.17 使用 null 检查来访问 Java 类

```
fun yellAtSafe(person: Person) {
    println((person.name ?: "Anyone").toUpperCase() + "!!!")
}

>>> yellAtSafe(Person(null))
ANYONE!!!
```

上面这个例子中，null 值被正确地处理了，没有抛出运行时异常。

使用 Java API 时要特别小心。大部分的库都没有（可空性）注解，所以可以把

所有类型都解释为非空，但那样会导致错误。为了避免错误，你应该阅读要用到的 Java 方法的文档（必要时还要查看它的实现），搞清楚它们什么时候会返回 null，并给那些方法加上检查。

为什么需要平台类型？

对 Kotlin 来说，把来自 Java 的所有值都当成可空的是不是更安全？这种设计也许可行，但是这需要对永远不为空的值做大量冗余的 null 检查，因为 Kotlin 编译器无法了解到这些信息。

涉及泛型的话这种情况就更糟糕了。例如，在 Kotlin 中，每个来自 Java 的 ArrayList<String> 都被当作 ArrayList<String?>?，每次访问或者转换类型都需要检查这些值是否为 null，这将抵消掉安全性带来的好处。编写这样的检查非常令人厌烦，所以 Kotlin 的设计者做出了更实用的选择，让开发者负责正确处理来自 Java 的值。

在 Kotlin 中不能声明一个平台类型的变量，这些类型只能来自 Java 代码，但你可能会在 IDE 的错误消息中见到它们：

```
>>> val i: Int = person.name
ERROR: Type mismatch: inferred type is String! but Int was expected
```

String! 表示法被 Kotlin 编译器用来表示来自 Java 代码的平台类型。你不能在自己的代码中使用这种语法。而且感叹号通常与问题的来源无关，所以通常可以忽略它。它只是强调类型的可空性是未知的。

如前所述，你可以用你喜欢的方式来解释平台类型，既可以是可空的也可以是非空的，所以下面两种声明都是有效的：

```
>>> val s: String? = person.name
>>> val s1: String = person.name
```

Java 的属性可以被当作可空……

……或者非空

这种情况下，和调用方法一样，你需要确保正确地理解了可空性。如果你试图用来自 Java 的 null 值给一个非空的 Kotlin 变量赋值，在赋值的瞬间你就会得到异常。

我们已经讨论了 Kotlin 怎样看待 Java 的类型。下面我们说说创建混合的 Kotlin 和 Java 类层级关系时会遇到的一些陷阱。

继承

当在 Kotlin 中重写 Java 的方法时，可以选择把参数和返回类型定义成

可空的，也可以选择把它们定义成非空的。例如，我们来看一个 Java 中的 StringProcessor 接口。

代码清单 6.18 使用 String 参数的 Java 接口

```
/* Java */
interface StringProcessor {
    void process(String value);
}
```

Kotlin 中下面的两种实现编译器都可以接收。

代码清单 6.19 实现 Java 接口时使用不同的参数可空性

```
class StringPrinter : StringProcessor {
    override fun process(value: String) {
        println(value)
    }
}

class NullableStringPrinter : StringProcessor {
    override fun process(value: String?) {
        if (value != null) {
            println(value)
        }
    }
}
```

注意，在实现 Java 类或者接口的方法时一定要搞清楚它的可空性。因为方法的实现可以在非 Kotlin 的代码中被调用，Kotlin 编译器会为你声明的每一个非空的参数生成非空断言。如果 Java 代码传给这个方法一个 null 值，断言将会触发，你会得到一个异常，即便你从没有在你的实现中访问过这个参数的值。

让我们总结一下关于可空性的讨论。我们讨论了可空类型和非空类型，以及处理它们的手段，安全操作的运算符（安全调用运算符 ?.、Elvis 运算符 ?: 和安全转换运算符 as?），还有非安全解析的运算符（非空断言 !!）。你了解了库函数 let 怎样帮助你完成简洁的非空检查，也了解了可空类型的扩展能帮你把非空检查移动到函数内部。我们也讨论了 Kotlin 中表示 Java 类型的平台类型。

现在我们已经介绍了可空性的话题，下面我们讨论一下 Kotlin 中基本数据类型是怎么表示的。这些可空性的知识对理解 Kotlin 如何处理 Java 的包装类型非常重要。

6.2 基本数据类型和其他基本类型

这一节将描述程序中使用的基本类型，例如 Int、Boolean 和 Any。与 Java

不一样的是，Kotlin 并不区分基本数据类型和它们的包装类。稍后你就会了解到背后的原因和底层的运作机制，你还将看到 Kotlin 类型和像 Object 和 Void 这样的 Java 类型之间的对应关系。

6.2.1 基本数据类型：Int、Boolean 及其他

正如你知道的，Java 把基本数据类型和引用类型做了区分。一个基本数据类型（如 int）的变量直接存储了它的值，而一个引用类型（如 String）的变量存储的是指向包含该对象的内存地址的引用。

基本数据类型的值能够更高效地存储和传递，但是你不能对这些值调用方法，或是把它们存放在集合中。Java 提供了特殊的包装类型（比如 java.lang.Integer），在你需要对象的时候对基本数据类型进行封装。因此，你不能用 Collection<int> 来定义一个整数的集合，而必须用 Collection<Integer> 来定义。

Kotlin 并不区分基本数据类型和包装类型，你使用的永远是同一个类型（比如：Int）：

```
val i: Int = 1
val list: List<Int> = listOf(1, 2, 3)
```

这样很方便。此外，你还能对一个数字类型的值调用方法。例如下面这段代码中，使用了标准库的函数 coerceIn 来把值限制在特定范围内：

```
fun showProgress(progress: Int) {
    val percent = progress.coerceIn(0, 100)
    println("We're ${percent}% done!")
}

>>> showProgress(146)
We're 100% done!
```

如果基本数据类型和引用类型是一样的，是不是意味着 Kotlin 使用对象来表示所有的数字？这样不是非常低效吗？确实低效，所以 Kotlin 并没有这样做。

在运行时，数字类型会尽可能地使用最高效的方式来表示。大多数情况下——对于变量、属性、参数和返回类型——Kotlin 的 Int 类型会被编译成 Java 基本数据类型 int。唯一不可行的例外是泛型类，比如集合。用作泛型类型参数的基本数据类型会被编译成对应的 Java 包装类型。例如，Int 类型被用作集合类的类型参数时，集合类将会保存对应包装类型 java.lang.Integer 的实例。

对应到 Java 基本数据类型的类型完整列表如下：

- 整数类型——Byte、Short、Int、Long

- 浮点数类型——Float、Double
- 字符类型——Char
- 布尔类型——Boolean

像 Int 这样的 Kotlin 类型在底层可以轻易地编译成对应的 Java 基本数据类型，因为两种类型都不能存储 null 引用。反过来也差不多：当你在 Kotlin 中使用 Java 声明时，Java 基本数据类型会变成非空类型（而不是平台类型），因为它们不能持有 null 值。现在我们讨论一下这些类型的可空版本。

6.2.2 可空的基本数据类型：Int?、Boolean? 及其他

Kotlin 中的可空类型不能用 Java 的基本数据类型表示，因为 null 只能被存储在 Java 的引用类型的变量中。这意味着任何时候只要使用了基本数据类型的可空版本，它就会编译成对应的包装类型。

为了说明可空类型的使用，我们回到本书最开始的例子，回忆一下 Person 类的定义。这个类表示一个名字永远已知但年龄可能已知或未指定的人。我们来增加一个函数，检查一个人是否比另一个更年长。

代码清单 6.20 使用可空的基本数据类型

```
data class Person(val name: String,
                  val age: Int? = null) {

    fun isOlderThan(other: Person): Boolean? {
        if (age == null || other.age == null)
            return null
        return age > other.age
    }
}

>>> println(Person("Sam", 35).isOlderThan(Person("Amy", 42)))
false
>>> println(Person("Sam", 35).isOlderThan(Person("Jane")))
null
```

注意，普通的可空性规则如何在这里应用。你不能就这样比较 Int? 类型的两个值，因为它们之中任何一个都可能为 null。相反，你必须检查两个值都不为 null。然后，编译器才允许你正常地比较它们。

在 Person 类中声明的 age 属性的值被当作 java.lang.Integer 存储。但是只有在你使用来自 Java 的类时这些细节才有意义。为了在 Kotlin 中选出正确的类型，你只需要考虑对变量或者属性来说，null 是否是它们可能的值。

如前所述，泛型类是包装类型应用的另一种情况。如果你用基本数据类型作为泛型类的类型参数，那么 Kotlin 会使用该类型的包装形式。例如，下面这段代码

创建了一个 Integer 包装类的列表，尽管你从来没有指定过可空类型或者用过 null 值：

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

这是由 Java 虚拟机实现泛型的方式决定的。JVM 不支持用基本数据类型作为类型参数，所以泛型类（Java 和 Kotlin 一样）必须始终使用类型的包装表示。因此，假如你要高效地存储基本数据类型元素的大型集合，要么使用支持这种集合的第三方库（如 Trove4J, http://trove.starlight-systems.com），要么使用数组来存储。本章末尾我们会讨论数组的细节。

下面我们来看看值在不同的基本数据类型之间是如何转换的。

6.2.3 数字转换

Kotlin 和 Java 之间一条重要的区别就是处理数字转换的方式。Kotlin 不会自动地把数字从一种类型转换成另外一种，即便是转换成范围更大的类型。例如，Kotlin 中下面这段代码不会编译：

```
val i = 1
val l: Long = i          <—— 错误：类型不匹配
```

相反，必须显式地进行转换：

```
val i = 1
val l: Long = i.toLong()
```

每一种基本数据类型（Boolean 除外）都定义有转换函数：toByte()、toShort()、toChar() 等。这些函数支持双向转换：既可以把小范围的类型括展到大范围，比如 Int.toLong()，也可以把大范围的类型截取到小范围，比如 Long.toInt()。

为了避免意外情况，Kotlin 要求转换必须是显式的，尤其是在比较装箱值的时候。比较两个装箱值的 equals 方法不仅会检查它们存储的值，还要比较装箱类型。所以，在 Java 中 new Integer(42).equals(new Long(42)) 会返回 false。假设 Kotlin 支持隐式转换，你也许能这样写：

```
val x = 1                          <—— Int 变量
val list = listOf(1L, 2L, 3L)      <—— Long 值的列表
x in list                          <—— 假设 Kotlin 支持隐式转换的话仍是 false
```

这个表达式的结果将会是 false，这与所有人的期望背道而驰。因此，上面例

子中的这行代码 x in list 根本不会编译。Kotlin 要求你显式地转换类型，这样只有类型相同的值才能比较：

```
>>> val x = 1
>>> println(x.toLong() in listOf(1L, 2L, 3L))
true
```

如果在代码中同时用到了不同的数字类型，你就必须显式地转换这些变量，来避免意想不到的行为。

基本数据类型字面值

Kotlin 除了支持简单的十进制数字之外，还支持下面这些在代码中书写数字字面值的方式：

- 使用后缀 L 表示 Long 类型（长整型）字面值：123L。
- 使用标准浮点数表示 Double（双精度浮点数）字面值：0.12、2.0、1.2e10、1.2e-10。
- 使用后缀 F 表 Float 类型（浮点数）字面值：123.4f、.456F、1e3f。
- 使用前缀 0x 或者 0X 表示十六进制字面值：0xCAFEBABE 或者 0xbcdL。
- 使用前缀 0b 或者 0B 表示二进制字面值：0b000000101。

注意，Kotlin 1.1 才开始支持数字字面值中的下画线。对字符字面值来说，可以使用和 Java 几乎一样的语法。把字符写在单引号中，必要时还可以使用转义序列。下面是几个例子都是有效的 Kotlin 字符字面值：'1'、'\t'（制表符）、'\u0009'（使用 Unicode 转义序列表示的制表符）。

注意，当你书写数字字面值的时候，一般不需要使用转换函数。一种可能性是用这种特殊的语法来显式地标记常量的类型，比如 42L 或者 42.0f。而且即使你没有用这种语法，当你使用数字字面值去初始化一个类型已知的变量时，又或是把字面值作为实参传给函数时，必要的转换会自动地发生。此外，算术运算符也被重载了，它们可以接收所有适当的数字类型。例如，下面这段代码并没有任何显式转换，但可以正确地工作：

```
fun foo(l: Long) = println(l)

>>> val b: Byte = 1
>>> val l = b + 1L
>>> foo(42)
42
```

常量有正确的类型

+ 可以进行字节类型和长整型参数的计算

编译器认为 42 是一个长整型

注意，Kotlin 算术运算符关于数值范围溢出的行为和 Java 完全一致；Kotlin 并没有引入由溢出检查带来的额外开销。

字符串转换

Kotlin 标准库提供了一套相似的扩展方法，用来把字符串转换成基本数据类型（toInt、toByte、toBoolean 等）：

```
>>> println("42".toInt())
42
```

每个这样的函数都会尝试把字符串的内容解析成对应的类型，如果解析失败则抛出 NumberFormatException。

在继续学习更多的类型之前，我们需要介绍三种特殊的类型：Any、Unit 和 Nothing。

6.2.4 “Any”和“Any?”：根类型

和 Object 作为 Java 类层级结构的根差不多，Any 类型是 Kotlin 所有非空类型的超类型（非空类型的根）。但是在 Java 中，Object 只是所有引用类型的超类型（引用类型的根），而基本数据类型并不是类层级结构的一部分。这意味着当你需要 Object 的时候，不得不使用像 java.lang.Integer 这样的包装类型来表示基本数据类型的值。而在 Kotlin 中，Any 是所有类型的超类型（所有类型的根），包括像 Int 这样的基本数据类型。

和 Java 一样，把基本数据类型的值赋给 Any 类型的变量时会自动装箱：

```
val answer: Any = 42
```

Any 是引用类型，所以值 42 会被装箱

注意 Any 是非空类型，所以 Any 类型的变量不可以持有 null 值。在 Kotlin 中如果你需要可以持有任何可能值的变量，包括 null 在内，必须使用 Any? 类型。

在底层，Any 类型对应 java.lang.Object。Kotlin 把 Java 方法参数和返回类型中用到的 Object 类型看作 Any（更确切地是当作平台类型，因为其可空性是未知的）。当 Kotlin 函数使用 Any 时，它会被编译成 Java 字节码中的 Object。

正如你在第 4 章中看到的，所有 Kotlin 类都包含下面三个方法：toString、equals 和 hashCode。这些方法都继承自 Any。Any 并不能使用其他 java.lang.Object 的方法（比如 wait 和 notify），但是可以通过手动把值转换成 java.lang.Object 来调用这些方法。

6.2.5 Unit 类型：Kotlin 的“void”

Kotlin 中的 Unit 类型完成了 Java 中的 void 一样的功能。当函数没什么有意思的结果要返回时，它可以用作函数的返回类型：

```
fun f(): Unit { ... }
```

语法上，这和写一个带有代码块体但不带类型声明的函数没有什么不同：

```
fun f() { ... }
```

显式的 Unit 声明被省略了

大多数情况下，你不会留意到 void 和 Unit 之间的区别。如果你的 Kotlin 函数使用 Unit 作为返回类型并且没有重写泛型函数，在底层它会被编译成旧的 void 函数。如果你要在 Java 代码中重写这个函数，新的 Java 函数需要返回 void。

那么 Kotlin 的 Unit 和 Java 的 void 到底有什么不一样呢？ Unit 是一个完备的类型，可以作为类型参数，而 void 却不行。只存在一个值是 Unit 类型，这个值也叫作 Unit，并且（在函数中）会被隐式地返回。当你在重写返回泛型参数的函数时这非常有用，只需要让方法返回 Unit 类型的值：

```
interface Processor<T> {
    fun process(): T
}

class NoResultProcessor : Processor<Unit> {
    override fun process() {
        // do stuff
    }
}
```

返回 Unit，但可以省略类型说明

这里不需要显式的 return

接口签名要求 process 函数返回一个值；而且，因为 Unit 类型确实有值，所以从方法中返回它并没有问题。然而你不需要在 NoResultProcessor.process 函数中写上显式的 return 语句，因为编译器会隐式地加上 return Unit。

和 Java 对比一下，Java 中为了解决使用“没有值”作为类型参数的任何一种可能解法，都没有 Kotlin 的解决方案这样漂亮。一种选择是使用分开的接口定义来分别表示需要和不需要返回值的接口（如 Callable 和 Runnable）。另一种是用特殊的 java.lang.Void 类型作为类型参数。即便你选择了后面这种方式，你还是需要加入一个 return null; 语句来返回唯一能匹配这个类型的值，因为只要返回类型不是 void，你就必须始终有显式的 return 语句。

你也许会奇怪为什么我们选择使用一个不一样的名字 Unit 而不是把它叫作 Void。在函数式编程语言中，Unit 这个名字习惯上被用来表示“只有一个实例”，

这正是 Kotlin 的 Unit 和 Java 的 void 的区别。我们本可以沿用 Void 这个名字，但是 Kotlin 还有一个叫作 Nothing 的类型，它有着完全不同的功能。Void 和 Nothing 两种类型的名字含义如此相近，会令人困惑。那么这种 Nothing 类型是什么？我们来看看吧。

6.2.6 Nothing 类型："这个函数永不返回"

对某些 Kotlin 函数来说，"返回类型"的概念没有任何意义，因为它们从来不会成功地结束。例如，许多测试库都有一个叫作 fail 的函数，它通过抛出带有特定消息的异常来让当前测试失败。一个包含无限循环的函数也永远不会成功地结束。

当分析调用这样函数的代码时，知道函数永远不会正常终止是很有帮助的。Kotlin 使用一种特殊的返回类型 Nothing 来表示：

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}
>>> fail("Error occurred")
java.lang.IllegalStateException: Error occurred
```

Nothing 类型没有任何值，只有被当作函数返回值使用，或者被当作泛型函数返回值的类型参数使用才会有意义。在其他所有情况下，声明一个不能存储任何值的变量没有任何意义。

注意，返回 Nothing 的函数可以放在 Elvis 运算符的右边来做先决条件检查：

```
val address = company.address ?: fail("No address")
println(address.city)
```

上面这个例子展示了在类型系统中拥有 Nothing 为什么极其有用。编译器知道这种返回类型的函数从不正常终止，然后在分析调用这个函数的代码时利用这个信息。在上面这个例子中，编译器会把 address 的类型推断成非空，因为它为 null 时的分支处理会始终抛出异常。

我们已经讨论完了 Kotlin 中的基本类型：基本数据类型、Any、Unit 和 Nothing。现在我们来看看集合类型，以及它们和 Java 对应部分的区别。

6.3 集合与数组

我们已经看到过许多使用不同集合 API 的代码例子，而且你知道 Kotlin 以 Java 集合库为基础构建，并通过扩展函数增加的特性来增强它。还有更多关于 Kotlin 如何支持集合的故事，以及 Java 和 Kotlin 集合之间的对应关系，现在就是深入细节的好时机。

6.3.1 可空性和集合

在本章前面，我们讨论了可空类型的概念，但仅仅简略地谈到类型参数的可空性。对前后一致的类型系统来说有一点十分关键：知道集合是否可以持有 null 元素，和知道变量值是否可以为 null 同等重要。好消息是 Kotlin 完全支持类型参数的可空性。就像变量的类型可以加上？字符来表示变量可以持有 null 一样，类型在被当作类型参数时也可以用同样的方式标记。我们通过下面这个函数的例子来看看它是如何工作的，这个函数从一个文件中读取文本行的列表，并尝试把每一行文本解析成一个数字。

代码清单 6.21 创建一个包含可空值的集合

```
fun readNumbers(reader: BufferedReader): List<Int?> {
    val result = ArrayList<Int?>()
    for (line in reader.lineSequence()) {
        try {
            val number = line.toInt()
            result.add(number)
        }
        catch(e: NumberFormatException) {
            result.add(null)
        }
    }
    return result
}
```

创建包含可空 Int 值的列表

向列表添加整数（非空值）

向列表添加 null，因为当前行不能被解析成整数

List<Int?> 是能持有 Int? 类型值的列表：换句话说，可以持有 Int 或者 null。如果一行文本可以被解析，那么就向 result 列表中添加一个整数，否则添加 null。注意从 Kotlin 1.1 开始，可以用函数 String.toIntOrNull 来简化这个例子，字符串不能被解析的时候它会返回 null。

注意，变量自己类型的可空性和用作类型参数的类型的可空性是有区别的。包含可空 Int 的列表和包含 Int 的可空列表之间的区别如图 6.10 所示。

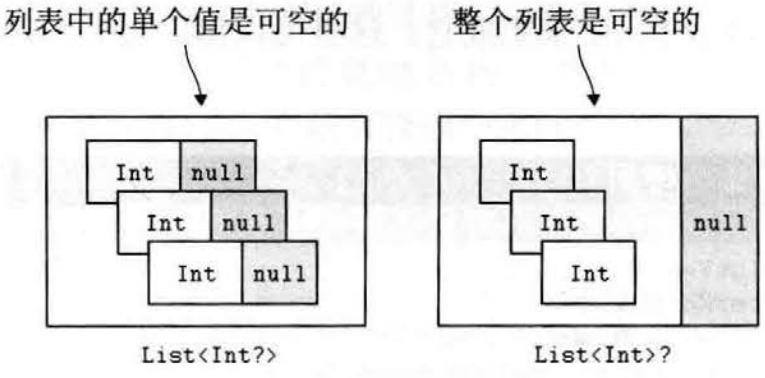


图 6.10 要小心决定什么是可空的：集合的元素还是集合本身？

在第一种情况下，列表本身始终不为 null，但列表中的每个值都可以为 null。第二种类型的变量可能包含空引用而不是列表实例，但列表中的元素保证是非空的。

在另外一种上下文中，你可能需要声明一个变量持有可空的列表，并且包含可空的数字。Kotlin 中的写法是 List<Int?>?，有两个问号。使用变量自己的值的时候，以及使用列表中每个元素的值的时候，都需要使用 null 检查。

要搞清楚如何使用包含可空值的列表，我们写了一个函数来计算列表中有效数字之和，并单独地对无效数字计数。

代码清单 6.22 使用可空值的集合

```
fun addValidNumbers(numbers: List<Int?>) {
    var sumOfValidNumbers = 0
    var invalidNumbers = 0
    for (number in numbers) {
        if (number != null) {
            sumOfValidNumbers += number
        } else {
            invalidNumbers++
        }
    }
    println("Sum of valid numbers: $sumOfValidNumbers")
    println("Invalid numbers: $invalidNumbers")
}
```

从列表中读取可空值

检查值是否为 null

```
>>> val reader = BufferedReader(StringReader("1\nabc\n42"))
>>> val numbers = readNumbers(reader)
>>> addValidNumbers(numbers)
Sum of valid numbers: 43
Invalid numbers: 1
```

这里并没有发生什么特殊的事情。当你访问一个列表中的元素时，你得到的是一个类型为 Int? 的值，而且要在用它进行算术运算之前检查它是否为 null。

遍历一个包含可空值的集合并过滤掉 null 是一个非常常见的操作，因此 Kotlin 提供了一个标准库函数 filterNotNull 来完成它。这里可以用它来大大简化前面的例子。

代码清单 6.23 对包含可空值的集合使用 filterNotNull

```
fun addValidNumbers(numbers: List<Int?>) {
    val validNumbers = numbers.filterNotNull()
    println("Sum of valid numbers: ${validNumbers.sum()}")
    println("Invalid numbers: ${numbers.size - validNumbers.size}")
}
```

当然，这种过滤也影响了集合的类型。validNumbers 的类型是 List<Int>，因为过滤保证了集合不会再包含任何为 null 的元素。

现在你了解了 Kotlin 怎样区分持有可空类型元素和持有非空类型元素的集合，现在我们看看 Kotlin 带来的另外一项重要的差别：只读集合与可变集合。

6.3.2 只读集合与可变集合

Kotlin 的集合设计和 Java 不同的另一项重要特质是，它把访问集合数据的接口和修改集合数据的接口分开了。这种区别存在于最基础的使用集合的接口之中：kotlin.collections.Collection。使用这个接口，可以遍历集合中的元素、获取集合大小、判断集合中是否包含某个元素，以及执行其他从该集合中读取数据的操作。但这个接口没有任何添加或移除元素的方法。

使用 kotlin.collections.MutableCollection 接口可以修改集合中的数据。它继承了普通的 kotlin.collections.Collection 接口，还提供了方法来添加和移除元素、清空集合等。图 6.11 展示了这两个接口中定义的关键方法。

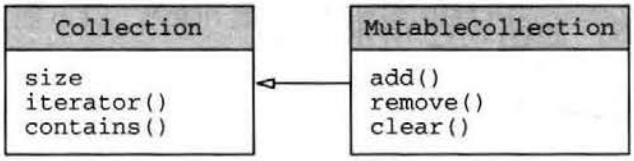


图 6.11 MutableCollection 继承了 Collection 并增加了修改集合内容的方法。

一般的规则是在代码的任何地方都应该使用只读接口，只在代码需要修改集合的地方使用可变接口的变体。

就像 val 和 var 之间的分离一样，只读集合接口与可变集合接口的分离能让程序中的数据发生的事情更容易理解。如果函数接收 Collection 而不是 MutableCollection 作为形参，你就知道它不会修改集合，而只是读取集合中的数据。如何函数要求你传递给它 MutableCollection，可以认为它将会修改数据。如果你用了集合作为组件部状态的一部分，可能需要把集合先拷贝一份再传递给这样的函数（这种模式通常称为*防御式拷贝*）。

例如，可以清楚地看到，下面清单中的 copyElements 函数仅仅修改了 target 集合，而没有修改 source 集合。

代码清单 6.24 使用只读集合接口与可变集合接口

```
fun <T> copyElements(source: Collection<T>,
                     target: MutableCollection<T>) {
    for (item in source) {
```

在 source 集合中的所有元素中循环

```
        target.add(item)                    <-- 向可变的 target 集合中添加元素
    }
}
```

```
>>> val source: Collection<Int> = arrayListOf(3, 5, 7)
>>> val target: MutableCollection<Int> = arrayListOf(1)
>>> copyElements(source, target)
>>> println(target)
[1, 3, 5, 7]
```

不能把只读集合类型的变量作为 target 参数传给函数，即便它的值是一个可变集合：

```
>>> val source: Collection<Int> = arrayListOf(3, 5, 7)
>>> val target: Collection<Int> = arrayListOf(1)
>>> copyElements(source, target)          <-- "target" 参数错误
Error: Type mismatch: inferred type is Collection<Int>
  but MutableCollection<Int> was expected
```

使用集合接口时需要牢记的一个关键点是*只读集合不一定是不可变的*。[1] 如果你使用的变量拥有一个只读接口类型，它可能只是同一个集合的众多引用中的一个。任何其他的引用都可能拥有一个可变接口类型，如图 6.12 所示。

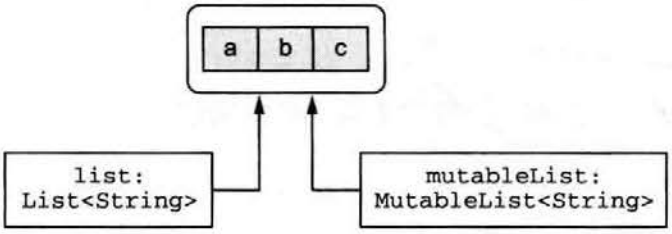


图 6.12 两个不同的引用，一个只读，另一个可变，指向同一个集合对象

如果你调用了这样的代码，它持有其他指向你集合的引用，或者并行地运行了这样的代码。你依然会遇到这样的状况，你正在使用集合的时候它被其他代码修改了，这会导致 concurrentModificationException 错误和其他一些问题。因此，必须了解*只读集合并不总是线程安全的*。如果你在多线程环境下处理数据，你需要保证代码正确地同步了对数据的访问，或者使用支持并发访问的数据结构。

只读集合与可变集合之间的分离是怎么做到的？我们之前不是说 Kotlin 集合和 Java 集合是一样的吗？这不是自相矛盾吗？我们来看看这里究竟发生了什么。

6.3.3 Kotlin 集合和 Java

每一个 Kotlin 接口都是其对应 Java 集合接口的一个实例，这种说法并没有错。在 Kotlin 和 Java 之间转移并不需要转换；不需要包装器也不需要拷贝数据。但是每

1 未来Kotlin标准库计划加入不可变集合。

一种 Java 集合接口在 Kotlin 中都有两种表示：一种是只读的，另一种是可变的，如图 6.13 所示。

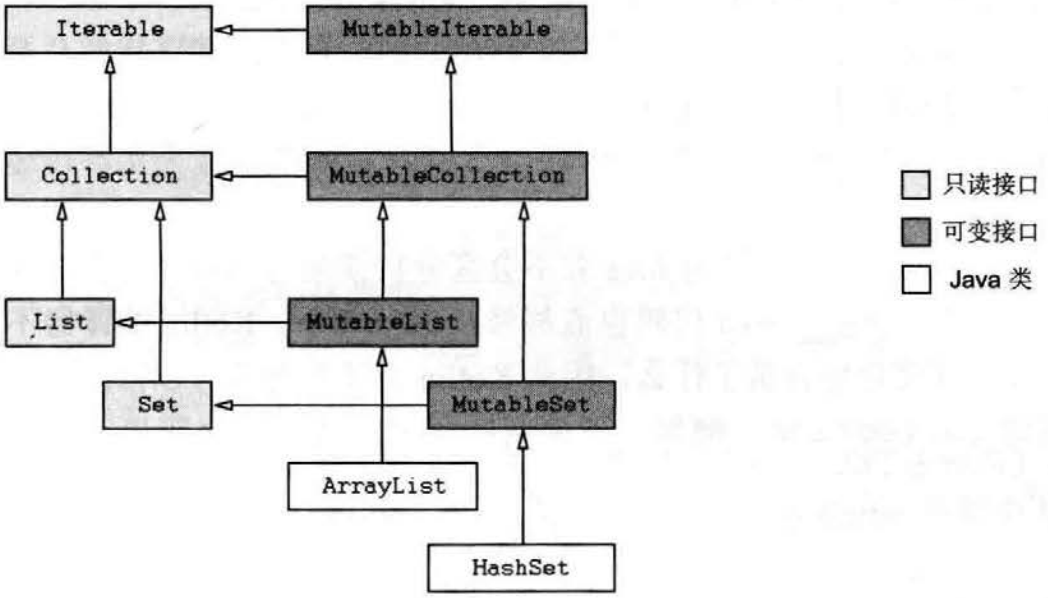


图 6.13 Kotlin 集合接口的层级结构，Java 类 ArrayList 和 HashSet 都继承了 Kotlin 可变接口

图 6.13 中的所有集合接口都是在 Kotlin 中声明的。Kotlin 中只读接口和可变接口的基本结构与 java.util 中的 Java 集合接口的结构是平行的。可变接口直接对应 java.util 包中的接口，而它们的只读版本缺少了所有产生改变的方法。

图 6.13 中还包含了 Java 类 java.util.ArrayList 和 java.util.HashSet，展示了 Kotlin 是怎样对待 Java 标准类的。在 Kotlin 看来，它们分别继承自 MutableList 接口和 MutableSet 接口。这里没有展示其他 Java 集合库的实现（LinkedList、SortedSet 等），但从 Kotlin 的角度来看，它们都有相似的超类型。这样你可以鱼与熊掌兼得，既得到了兼容性，也得到了可变接口和只读接口之间清晰的分离。

除了集合之外，Kotlin 中 Map 类（它并没有继承 Collection 或是 Iterable）也被表示成了两种不同的版本：Map 和 MutableMap。表 6.1 展示了可以用来创建不同类型集合的函数。

表 6.1 集合创建函数

集合类型	只读	可变
List	listOf	mutableListOf、arrayListOf
Set	setOf	mutableSetOf、hashSetOf、linkedSetOf、sortedSetOf
Map	mapOf	mutableMapOf、hashMapOf、linkedMapOf、sortedMapOf

注意，setOf() 和 mapOf() 返回的是 Java 标准类库中类的实例（至少在 Kotlin 1.0 中是这样），在底层它们都是可变的[2]。但你不能完全信赖这一点：Kotlin 的未来版本可能会使用真正不可变的实现类作为 setOf 和 mapOf 的返回值。

当你需要调用一个 Java 方法并把集合作为实参传给它时，可以直接这样做，不需要任何额外的步骤。例如，你有一个使用 java.util.Collection 做形参的 Java 方法，可以把任意 Collection 或 MutableCollection 的值作为实参传递给这个形参。

这对集合的可变性有重要影响。因为 Java 并不会区分只读集合与可变集合，即使 Kotlin 中把集合声明成只读的，Java 代码也能够修改这个集合。Kotlin 编译器不能完全地分析 Java 代码到底对集合做了什么，因此 Kotlin 无法拒绝向可以修改集合的 Java 代码传递只读 Collection。例如，下面两段代码组成了一个跨语言兼容的 Kotlin/Java 程序：

```
/* Java */
// CollectionUtils.java
public class CollectionUtils {
    public static List<String> uppercaseAll(List<String> items) {
        for (int i = 0; i < items.size(); i++) {
            items.set(i, items.get(i).toUpperCase());
        }
        return items;
    }
}

// Kotlin
// collections.kt
fun printInUppercase(list: List<String>) {
    println(CollectionUtils.uppercaseAll(list))
    println(list.first())
}

>>> val list = listOf("a", "b", "c")
>>> printInUppercase(list)
[A, B, C]
A
```

声明只读的参数

调用可以修改集合的 Java 函数

打印被修改过的集合

因此，如果你写了一个 Kotlin 函数，使用了集合并传递给了 Java，你有责任使用正确的参数类型，这取决于你调用的 Java 代码是否会修改集合。

留意此注意事项也适用于包含非空类型元素的集合类。如果你向 Java 方法转递了这样的集合，该方法就可能在其中写入 null 值；Kotlin 没有办法在不影响性能的情况下，禁止它的发生，或者察觉到已经发生的改变。因此，当你向可以修改集合的 Java 代码传递集合的时候，你需要采取特别的预防措施，来确保 Kotlin 类型正

2 包装成Collection.unmodifiable会带来间接的开销，所以并没有实现。

确地反映出集合上所有可能的修改。

现在，我们来看看 Kotlin 如何处理 Java 代码中声明的集合。

6.3.4 作为平台类型的集合

如果你还记得这一章前面关于可空性的讨论，你应该也记得 Kotlin 把那些定义在 Java 代码中的类型看成平台类型。Kotlin 没有任何关于平台类型的可空性信息，所以编译器允许 Kotlin 代码将其视为可空或者非空。同样，Java 中声明的集合类型的变量也被视为平台类型。一个平台类型的集合本质上就是可变性未知的集合——Kotlin 代码将其视为只读的或者可变的。通常这并不重要，因为，实际上你想要执行的所有操作都能正常工作。

当你重写或者实现签名中有集合类型的 Java 方法时这种差异才变得重要。这里，像平台类型的可空性一样，你需要决定使用哪一种 Kotlin 类型来表示这个 Java 类型，它来自你要重写或实现的方法。

这种情况下，你要做出多种选择，它们都会反映在产生的 Kotlin 参数类型中：

- 集合是否可空？
- 集合中的元素是否可空？
- 你的方法会不会修改集合？

为了看到差异，考虑下面这些情况。在第一个例子中，一个 Java 接口表示一个能处理文件中文本的对象。

代码清单 6.25 使用集合参数的 Java 接口

```
/* Java */
interface FileContentProcessor {
    void processContents(File path,
        byte[] binaryContents,
        List<String> textContents);
}
```

这个接口的 Kotlin 实现需要做出下面的选择：

- 列表将是可空的，因为有些文件是二进制格式，它们的内容不能被表示成文本。
- 列表中的元素将会是非空的，因为文件中每一行都永远不为 `null`。
- 列表将是只读的，因为它表示的是文件的内容，而且这些内容不会被修改。

下面就是这种实现看起来的样子。

代码清单 6.26 FileContentProcessor 的 Kotlin 实现

```
class FileIndexer : FileContentProcessor {
    override fun processContents(path: File,
        binaryContents: ByteArray?,
        textContents: List<String>?) {

        // ...
    }
}
```

把它和另外一个接口对比。这里接口的实现从文本表单中解析出数据并放到一个对象列表中，再把这些对象附加到输出列表中。当发现解析错误时，就把错误信息添加到另一个单独的列表中，作为错误日志。

代码清单 6.27 另一个使用集合参数的 Java 接口

```
/* Java */
interface DataParser<T> {
    void parseData(String input,
        List<T> output,
        List<String> errors);
}
```

这种情况下的选择是不同的：

- List<String> 将是非空的，因为调用者总是需要接收错误消息。
- 列表中的元素将是可空的，因为不是每个输出列表中的条目都有关联的错误信息。
- List<String> 将是可变的，因为实现代码需要向其中添加元素。

下面是在 Kotlin 中如何实现这个接口。

代码清单 6.28 DataParser 的 Kotlin 实现

```
class PersonParser : DataParser<Person> {
    override fun parseData(input: String,
        output: MutableList<Person>,
        errors: MutableList<String?>) {

        // ...
    }
}
```

注意，同样的 Java 类型——List<String>——如何表示成了两种不同的 Kotlin 类型：一种是 List<String>?（包含字符串的可空列表），另一种是

MutableList<String?>（包含可空字符串的可变列表）。为了做出正确的选择，你必须知道 Java 接口或类必须遵守的确切契约。基于你的实现要做的事情这通常很容易理解。

现在，我们讨论了集合，是时候看看数组了。如前所述，默认情况下，你应该优先使用集合而不是数组。但是因为还有大量 Java API 仍然在使用数组，我们将介绍如何在 Kotlin 中使用它们。

6.3.5 对象和基本数据类型的数组

Kotlin 数组的语法出现在了每个例子中，因为数组是 Java main 函数标准签名的一部分。下面给你一些关于它的提示：

代码清单 6.29 使用数组

```
fun main(args: Array<String>) {
    for (i in args.indices) {
        println("Argument $i is: ${args[i]}")
    }
}
```

使扩展属性 array.indices 在下标的范围内迭代

通过下标使用 array[index] 访问元素

Kotlin 中的一个数组是一个带有类型参数的类，其元素类型被指定为相应的类型参数。

要在 Kotlin 中创建数组，有下面这些方法供你选择：

- arrayOf 函数创建一个数组，它包含的元素是指定为该函数的实参
- arrayOfNulls 创建一个给定大小的数组，包含的是 null 元素。当然，它只能用来创建包含元素类型可空的数组。
- Array 构造方法接收数组的大小和一个 lambda 表达式，调用 lambda 表达式来创建每一个数组元素。这就是使用非空元素类型来初始化数组，但不用显式地传递每个元素的方式。

这里有一个简单的例子，展示了如何使用 Array 函数来创建从 "a" 到 "z" 的字符串数组。

代码清单 6.30 创建字符数组

```
>>> val letters = Array<String>(26) { i -> ('a' + i).toString() }
>>> println(letters.joinToString(""))
abcdefghijklmnopqrstuvwxyz
```

Lambda 接收数组元素的下标并返回放在数组下标位置的值。这里你把字符 'a'

加上下标并把结果转换成字符串来计算出数组元素的值。为了清楚起见，这里显示了数组元素的类型，但在真实的代码中可以省略。因为编译器可以推导出它的类型。

说到这里，Kotlin 代码中最常见的创建数组的情况之一是需要调用参数为数组的 Java 方法时，或是调用带有 vararg 参数的 Kotlin 函数时。在这些情况下，通常已经将数据存储在集合中，只需将其转换为数组即可。可以使用 toTypedArray 方法来执行此操作。

代码清单 6.31 向 vararg 方法传递集合

```
>>> val strings = listOf("a", "b", "c")
>>> println("%s/%s/%s".format(*strings.toTypedArray()))
a/b/c
```

期望 vararg 参数时使用展开运算符（*）传递数组

和其他类型一样，数组类型的类型参数始终会变成对象类型。因此，如果你声明了一个 Array<Int>，它将会是一个包含装箱整型的数组（它的 Java 类型将是 java.lang.Integer[]）。如果你需要创建没有装箱的基本数据类型的数组，必须使用一个基本数据类型数组的特殊类。

为了表示基本数据类型的数组，Kotlin 提供了若干独立的类，每一种基本数据类型都对应一个。例如，Int 类型值的数组叫作 IntArray。Kotlin 还提供了 ByteArray、CharArray、BooleanArray 等给其他类型。所有这些类型都被编译成普通的 Java 基本数据类型数组，比如 int[]、byte[]、char[] 等。因此这些数组中的值存储时并没有装箱，而是使用了可能的最高效的方式。

要创建一个基本数据类型的数组，你有如下选择：

- 该类型的构造方法接收 size 参数并返回一个使用对应基本数据类型默认值（通常是 0）初始化好的数组。
- 工厂函数（IntArray 的 intArrayOf，以及其他数组类型的函数）接收变长参数的值并创建存储这些值的数组。
- 另一种构造方法，接收一个大小和一个用来初始化每个元素的 lambda。

下面是创建存储了 5 个 0 的整型数组的两种选择：

```
val fiveZeros = IntArray(5)
val fiveZerosToo = intArrayOf(0, 0, 0, 0, 0)
```

下面是接收 lambda 的构造方法的例子：

```
>>> val squares = IntArray(5) { i -> (i+1) * (i+1) }
>>> println(squares.joinToString())
1, 4, 9, 16, 25
```

或者，假如你有一个持有基本数据类型装箱后的值的数组或者集合，可以用对应的转换函数把它们转换成基本数据类型的数组，比如 toIntArray。

接下来，我们来看一下你可以对数组做的事情。除了那些基本操作（获取数组的长度，获取或者设置元素）外，Kotlin 标准库支持一套和集合相同的用于数组的扩展函数。第 5 章中你看到的全部函数（filter、map 等）也适用于数组，包括基本数据类型的数组（注意，这些方法的返回值是列表而不是数组）。

我们来看看如何使用 forEachIndexed 函数加上 lambda 来重写代码清单 6.30 中的代码。

代码清单 6.32 对数组使用 forEachIndexed

```
fun main(args: Array<String>) {
    args.forEachIndexed { index, element ->
        println("Argument $index is: $element")
    }
}
```

现在你知道了如何在代码中使用数组。在 Kotlin 中使用它们就像使用集合一样简单。

6.4 小结

- Kotlin 对可空类型的支持，可以帮助我们在编译期，检测出潜在的 NullPointerException 错误。
- Kotlin 提供了像安全调用（?.）、Elvis 运算符（?:）、非空断言（!!）及 let 函数这样的工具来简洁地处理可空类型。
- as? 运算符提供了一种简单的方式来把值转换成一个类型，以及处理当它拥有不同类型时的情况。
- Java 中的类型在 Kotlin 中被解释成平台类型，允许开发者把它们当作可空或非空来对待。
- 表示基本数字的类型（如 Int）看起来用起来都像普通的类，但通常会被编译成 Java 基本数据类型。
- 可空的基本数据类型（如 Int?）对应着 Java 中的装箱基本数据类型（如 java.lang.Integer）。
- Any 类型是所有其他类型的超类型，类似于 Java 的 Object。而 Unit 类比于 void。
- 不会正常终止的函数使用 Nothing 类型作为返回类型。

- Kotlin 使用标准 Java 集合类，并通过区分只读和可变集合来增强它们。
- 当你在 Kotlin 中继承 Java 类或者实现 Java 接口时，你需要仔细考虑参数的可空性和可变性。
- Kotlin 的 `Array` 类就像普通的泛型类，但它会被编译成 Java 数组。
- 基本数据类型的数组使用像 `IntArray` 这样的特殊类来表示。

第2部分

拥抱Kotlin

至此，使用 Kotlin 来访问现有的 API 你应该已经非常熟悉了。在本书的这一部分，你将学习到如何在 Kotlin 中构建自己的 API。务必要记住的是，构建 API 不只是库开发者的事情：在程序中每次有两个类交互时，其中一个类就在给另一个提供 API。

在第 7 章中，你将学习到 Kotlin 中约定的原理，用来实现运算符重载和其他抽象的技术（如委托属性）。在第 8 章，我们将剖析 lambda，学习如何声明函数，来接收 lambda 作为参数。你也将学习到 Kotlin 对 Java 的一些高级概念的处理，例如泛型（在第 9 章）、注解和反射（第 10 章）。另外在第 10 章中，你将学习到现实世界中的一个大型 Kotlin 项目：JKid，一个 JSON 序列化和反序列化的库。最后，在第 11 章中，你将接触到 Kotlin 的“王冠明珠”之一：支持构建特定的领域语言。

7 运算符重载及其他约定

本章内容包括

- 运算符重载
- 约定：支持各种运算的特殊命名函数
- 委托属性

如你所知，Java 在标准库中有一些与特定的类相关联的语言特性。例如，实现了 `java.lang.Iterable` 接口的对象可以在 `for` 循环中使用，实现了 `java.lang.AutoCloseable` 接口的对象可以在 try-with-resources 语句中使用。

Kotlin 也有许多特性的原理非常类似，通过调用自己代码中定义的函数，来实现特定语言结构。但是，在 Kotlin 中，这些功能与特定的函数命名相关，而不是与特定的类型绑定。例如，如果在你的类中定义了一个名为 `plus` 的特殊方法，那么按照约定，就可以在该类的实例上使用 + 运算符。因此，在 Kotlin 中，我们把这种技术称为约定。在这一章我们将学习 Kotlin 支持的各种约定，以及它们的使用。

Kotlin 使用约定的原则，并不像 Java 那样依赖类型，因为它允许开发人员适应现有的 Java 类，来满足 Kotlin 语言特性的要求。由类实现的接口集是固定的，而 Kotlin 不能为了实现其他接口而修改现有的类。另一方面，Kotlin 可以通过扩展函数的机制来为现有的类增添新的方法。可以把任意约定方法定义为扩展函数，从而适应任何现有的 Java 类而不用修改其代码。

作为这一章的例子，我们将实现一个简单的 Point 类，用来表示屏幕上的一个点。这样的类存在于大多数的 UI 框架中，也可以简单地把这里的定义用到你的环境中去：

```
data class Point(val x: Int, val y: Int)
```

我们就从给 Point 类定义一些算术运算符开始吧。

7.1 重载算术运算符

在 Kotlin 中使用约定的最直接的例子就是算术运算符。在 Java 中，全套的算术运算只能用于基本数据类型，+ 运算符可以与 String 值一起使用。但是，这些运算在其他一些情况下用起来也很方便。例如，在使用 BigInteger 类处理数字的时候，使用 + 号就比调用 add 方法显得更为优雅；给集合添加元素的时候，你可能也在想要是能用 += 运算符就好了。在 Kotlin 中，你就可以这样做，在这一节我们就来看一下到底它是如何工作的。

7.1.1 重载二元算术运算

我们要支持的第一个运算，就是把两个点加到一起。这个运算需要把点的 (X, Y) 坐标分别加到一起。可以这样来实现：

代码清单 7.1 定义一个 plus 运算符

```
data class Point(val x: Int, val y: Int) {
    operator fun plus(other: Point): Point {
        return Point(x + other.x, y + other.y)
    }
}

>>> val p1 = Point(10, 20)
>>> val p2 = Point(30, 40)
>>> println(p1 + p2)
Point(x=40, y=60)
```

定义一个名为“plus”的方法

坐标分别相加，然后返回一个新的点

通过使用 + 号来调用“plus”方法

注意，如何使用 operator 关键字来声明 plus 函数。用于重载运算符的所有函数都需要用该关键字标记，用来表示你打算把这个函数作为相应的约定的实现，并且不是碰巧地定义一个同名函数。

在使用了 operator 修饰符声明了 plus 函数之后，你就可以直接使用 + 号来求和了。事实上，如图 7.1 所示，这里它调用的是 plus 函数。

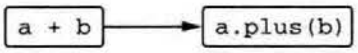


图 7.1 + 号运算将会转换为 plus 函数的调用

除了把这个运算符声明为一个成员函数外，也可以把它定义为一个扩展函数。

代码清单 7.2 把运算符定义为扩展函数

```
operator fun Point.plus(other: Point): Point {
    return Point(x + other.x, y + other.y)
}
```

这样实现是一样的。后续的示例中将会使用扩展函数的语法来写，这是给第三方库的类定义约定扩展函数的常用模式。而且，对于你自己的类这种语法也同样适用。

和其他一些语言相比，在 Kotlin 中不管是定义还是使用重载运算符都更为简单，因为你不能定义自己的运算符。Kotlin 限定了你能重载哪些运算符，以及你需要在你的类中定义的对应名字的函数。表 7.1 列举了你能定义的二元运算符，以及对应的函数名称。

表 7.1 可重载的二元算术运算符

表达式	函数名
a * b	times
a / b	div
a % b	mod
a + b	plus
a - b	minus

自定义类型的运算符，基本上和与标准数字类型的运算符有着相同的优先级。例如，如果是 a+ b * c，乘法将始终在添加之前执行，即使你已经自己定义了这些运算符。运算符 *、/ 和%具有相同的优先级，高于 + 和 - 运算符的优先级。

运算符函数和 Java

从 Java 调用 Kotlin 运算符非常容易：因为每个重载的运算符都被定义为一个函数，可以像普通函数那样调用它们。当从 Kotlin 调用 Java 的时候，对于与 Kotlin 约定匹配的函数都可以使用运算符语法来调用。由于 Java 没有定义任何用于标记运算符函数的语法，所以使用 operator 修饰符的要求对它不适用，唯一的约束是，参数需要匹配名称和数量。如果 Java 类定义了一个满足需求的函数，但是起了一个不同的名称，可以通过定义一个扩展函数来修正这个函数名，用来代替现有的 Java 方法。

当你在定义一个运算符的时候，不要求两个运算数是相同的类型。例如，让我们定义一个运算符，它允许你用一个数字来缩放一个点，可以用它在不同坐标系之间做转换。

代码清单 7.3 定义一个运算数类型不同的运算符

```
operator fun Point.times(scale: Double): Point {
    return Point((x * scale).toInt(), (y * scale).toInt())
}

>>> val p = Point(10, 20)
>>> println(p * 1.5)
Point(x=15, y=30)
```

注意，Kotlin 运算符不会自动支持交换性（交换运算符的左右两边）。如果希望用户能够使用 `1.5 * p` 以外，还能使用 `p * 1.5`，你需要为它定义一个单独的运算符：`operator fun Double.times(p: Point): Point`。

运算符函数的返回类型也可以不同于任一运算数类型。例如，可以定义一个运算符，通过多次重复单个字符来创建字符串。

代码清单 7.4 定义一个返回结果不同的运算符

```
operator fun Char.times(count: Int): String {
    return toString().repeat(count)
}

>>> println('a' * 3)
aaa
```

这个运算符，接收一个 Char 作为左值，Int 作为右值，然后返回一个 String 类型。这样的运算数和结果类型的组合是允许的。

注意，和普通的函数一样，可以重载 operator 函数：可以定义多个同名的，但参数类型不同的方法。

没有用于位运算的特殊运算符

Kotlin 没有为标准数字类型定义任何位运算符，因此，也不允许你为自定义类型定义它们。相反，它使用支持中缀调用语法的常规函数，可以为自定义类型定义相似的函数。

以下是 Kotlin 提供的，用于执行位运算的完整函数列表：

- shl——带符号左移
- shr——带符号右移

- ushr——无符号右移
- and——按位与
- or——按位或
- xor——按位异或
- inv——按位取反

下面的示例展示了这些函数的使用方法：

```
>>> println(0x0F and 0xF0)
0
>>> println(0x0F or 0xF0)
255
>>> println(0x1 shl 4)
16
```

下面，我们来研究一下类似 += 这样的，合并了两步操作的运算符：赋值，以及对应的算术运算。

7.1.2 重载复合赋值运算符

通常情况下，当你在定义像 plus 这样的运算符函数时，Kotlin 不止支持 + 号运算，也支持 +=。像 +=、-= 等这些运算符被称为复合赋值运算符。看这个例子：

```
>>> var point = Point(1, 2)
>>> point += Point(3, 4)
>>> println(point)
Point(x=4, y=6)
```

这等同于 point = point + Point(3, 4) 的写法。当然，这个只对于可变变量有效。

在一些情况下，定义 += 运算可以修改使用它的变量所引用的对象，但不会重新分配引用。将一个元素添加到可变集合，就是一个很好的例子：

```
>>> val numbers = ArrayList<Int>()
>>> numbers += 42
>>> println(numbers[0])
42
```

如果你定义了一个返回值为 Unit，名为 plusAssign 的函数，Kotlin 将会在用到 += 运算符的地方调用它。其他二元算术运算符也有命名相似的对应函数：如 minusAssign、timesAssign 等。

Kotlin 标准库为可变集合定义了 plusAssign 函数，在前面的例子中可以这样使用：

```
operator fun <T> MutableCollection<T>.plusAssign(element: T) {
    this.add(element)
}
```

当你在代码中用到 += 的时候，理论上 plus 和 plusAssign 都可能被调用（如图 7.2 所示）。如果在这种情况下，两个函数都有定义且适用，编译器会报错。一种可行的解决办法是，替换运算符的使用为普通函数调用。另一个办法是，用 val 替换 var，这样 plusAssign 运算就不再适用。但一般来说，最好在设计新类时保持（可变性）一致：尽量不要同时给一个类添加 plus 和 plusAssign 运算。如果像前面的一个示例中的 Point，这个类是不可变的，那么就应该只提供返回一个新值（如 plus）的运算。如果一个类是可变的，比如构建器，那么只需要提供 plusAssign 和类似的运算就够了。

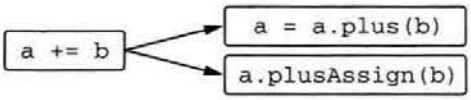


图 7.2 运算符 += 可以被转换为 plus 或者 plusAssign 函数的调用

Kotlin 标准库支持集合的这两种方法。+ 和 - 运算符总是返回一个新的集合。+= 和 -= 运算符用于可变集合时，始终就地修改它们；而它们用于只读集合时，会返回一个修改过的副本（这意味着只有当引用只读集合的变量被声明为 var 的时候，才能使用 += 和 -=）。作为它们的运算数，可以使用单个元素，也可以使用元素类型一致的其他集合：

```
>>> val list = arrayListOf(1, 2)
>>> list += 3
>>> val newList = list + listOf(4, 5)
>>> println(list)
[1, 2, 3]
>>> println(newList)
[1, 2, 3, 4, 5]
```

+= 修改“list”

+ 返回一个包含所有元素的新列表

至此，我们已经讨论了二元运算符的重载，即可以应用到两个值的运算符，比如 a+b。另外，Kotlin 也允许重载只能用于单个值运算的一元运算符，比如 -a。

7.1.3 重载一元运算符

重载一元运算符的过程与你在前面看到的方式相同：用预先定义的一个名称来声明函数（成员函数或扩展函数），并用修饰符 operator 标记。我们来看一个例子。

代码清单 7.5 定义一个一元运算符

```
operator fun Point.unaryMinus(): Point {
    return Point(-x, -y)
}

>>> val p = Point(10, 20)
>>> println(-p)
Point(x=-10, y=-20)
```

一元运算符
无参数

坐标取反，
然后返回

用于重载一元运算符的函数，没有任何参数。一元的 plus 运算符调用如图 7.3 所示。表 7.2 列举了所有可以重载的一元运算符。

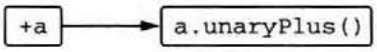


图 7.3 一元运算符 + 被转换为 unaryPlus 函数的调用

表 7.2 可重载的一元算法的运算符

表达式	函数名
+a	unaryPlus
-a	unaryMinus
!a	not
++a, a++	inc
--a, a--	dec

当你定义 inc 和 dec 函数来重载自增和自减的运算符时，编译器自动支持与普通数字类型的前缀和后缀自增运算符相同的语义。考虑一下用来重载 BigDecimal 类的 ++ 运算符的这个例子。

代码清单 7.6 定义一个自增运算符

```
operator fun BigDecimal.inc() = this + BigDecimal.ONE

>>> var bd = BigDecimal.ZERO
>>> println(bd++)
0
>>> println(++bd)
2
```

在第一个 println 语句执行后增加

在第二个 println 语句执行前增加

后缀运算 ++ 首先返回 bd 变量的当前值，然后执行 ++，这个和前缀运算相反。打印的值与使用 Int 类型的变量所看到的相同，不需要额外做什么特别的事情就能支持。

7.2 重载比较运算符

与算术运算符一样，在 Kotlin 中，可以对任何对象使用比较运算符（==、!=、>、< 等），而不仅仅限于基本数据类型。不用像 Java 那样调用 equals 或 compareTo 函数，可以直接使用比较运算符。在这一节中，我们将介绍用于支持这些运算符的约定。

7.2.1 等号运算符："equals"

我们在 4.3.1 节中就曾谈到过等式比较的话题，也已经看到，如果在 Kotlin 中使用 == 运算符，它将被转换成 equals 方法的调用。这只是我们要讨论的约定原则中的一个。

使用 != 运算符也会被转换成 equals 函数的调用，明显的差异在于，它们的结果是相反的。注意，和所有其他运算符不同的是，== 和 != 可以用于可空运算数，因为这些运算符事实上会检查运算数是否为 null。比较 a == b 会检查 a 是否为非空，如果不是，就调用 a.equals(b)（如图 7.4 所示）；否则，只有两个参数都是空引用，结果才是 true。

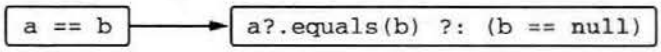


图 7.4 等式校验 == 被转换为 equals 函数的调用，以及 null 的校验

对于 Point 类，因为已经被标记为数据类（4.3.2 节有详细说明），equals 的实现将会由编译器自动生成。但如果要手动实现，那么代码可以是这样的。

代码清单 7.7 实现 equals 函数

```
class Point(val x: Int, val y: Int) {
    override fun equals(obj: Any?): Boolean {
        if (obj === this) return true
        if (obj !is Point) return false
        return obj.x == x && obj.y == y
    }
}

>>> println(Point(10, 20) == Point(10, 20))
true
>>> println(Point(10, 20) != Point(5, 5))
true
>>> println(null == Point(1, 2))
false
```

重写在 Any 中定义的方法

优化：检查参数是否与 this 是同一个对象

检查参数类型

智能转换为 Point 来访问 x、y 属性

这里使用了恒等运算符（===）来检查参数与调用 equals 的对象是否相同。

恒等运算符与 Java 中的 == 运算符是完全相同的：检查两个参数是否是同一个对象的引用（如果是基本数据类型，检查它们是否是相同的值）。在实现了 equals（方法）之后，通常会使用这个运算符来优化调用代码。注意，=== 运算符不能被重载。

equals 函数之所以被标记为 override，那是因为与其他约定不同的是，这个方法的实现是在 Any 类中定义的（Kotlin 中的所有对象都支持等式比较）。这也解释了为什么你不需要将它标记为 operator：Any 中的基本方法就已经标记了，而且函数的 operator 修饰符也适用于所有实现或重写它的方法。还要注意，equals 不能实现为扩展函数，因为继承自 Any 类的实现始终优先于扩展函数。

这个例子显示 != 运算符的使用也会转换为 equals 方法的调用。编译器会自动对返回值取反，因此你不需要再做别的事情，就可以正常运行。

那其他比较运算符呢？

7.2.2 排序运算符：compareTo

在 Java 中，类可以实现 Comparable 接口，以便在比较值的算法中使用，例如在查找最大值或排序的时候。接口中定义的 compareTo 方法用于确定一个对象是否大于另一个对象。但是在 Java 中，这个方法的调用没有简明语法。只有基本数据类型能使用 < 和 > 来比较，所有其他类型都需要明确写为 element1.compareTo(element2)。

Kotlin 支持相同的 Comparable 接口。但是接口中定义的 compareTo 方法可以按约定调用，比较运算符（<，>，<= 和 >=）的使用将被转换为 compareTo，如图 7.5 所示。compareTo 的返回类型必须为 Int。p1 < p2 表达式等价于 p1.compareTo(p2)< 0。其他比较运算符的运算方式也是完全一样的。



图 7.5 两个对象的比较被转换为 compareTo 的函数调用，然后结果与零比较

因为没有明显的正确方式来比较两个点，所以让我们用旧的 Person 类作为例子，来看看如何实现这个方法。这个实现将对地址簿排序（先比较名字中的姓氏，如果姓氏相同，再比较名字）。

代码清单 7.8 实现 compareTo 方法

```
class Person(
        val firstName: String, val lastName: String
) : Comparable<Person> {
    override fun compareTo(other: Person): Int {
        return compareValuesBy(this, other,          <-- 按顺序调用给定的方法，
            Person::lastName, Person::firstName)         并比较它们的值
```

```
    }
}
>>> val p1 = Person("Alice", "Smith")
>>> val p2 = Person("Bob", "Johnson")
>>> println(p1 < p2)
false
```

在这种情况下，可以实现 Comparable 接口，这样 Person 对象不仅可以在 Kotlin 代码中用来比较，还可以被 Java 函数（比如用于对集合进行排序的功能）进行比较。与 equals 一样，operator 修饰符已经被用在了基类的接口中，因此在重写该接口时无须再重复。

要注意如何使用 Kotlin 标准库中的 compareValuesBy 函数来简洁地实现 compareTo 方法。这个函数接收用来计算比较值的一系列回调，按顺序依次调用回调方法，两两一组分别做比较，并返回结果。如果值不同，则返回比较结果；如果它们相同，则继续调用下一个；如果没有更多回调来调用，则返回 0。这些回调函数可以像 lambda 一样传递，或者像这里做的一样，作为属性引用传递。

注意，尽管自己直接实现字段的比较会运行得更快一点，然而这样会包含更多的代码。一般情况下，更推荐使用简洁的写法，不用过早地担心性能问题，除非你知道这个实现将会被频繁调用。

所有 Java 中实现了 Comparable 接口的类，都可以在 Kotlin 中使用简洁的运算符语法，不用再增加扩展函数：

```
>>> println("abc" < "bac")
true
```

7.3 集合与区间的约定

处理集合最常见的一些操作是通过下标来获取和设置元素，以及检查元素是否属于当前集合。所有的这些操作都支持运算符语法：要通过下标获取或设置元素，可以使用语法 a[b]（称为下标运算符）。可以使用 in 运算符来检查元素是否在集合或区间内，也可以迭代集合。可以作为集合的自定义类。让我们来看看用于支持这些操作的约定。

7.3.1 通过下标来访问元素："get" 和 "set"

我们已经知道，在 Kotlin 中，可以用类似 Java 中数组的方式来访问 map 中的元素——使用方括号：

```
val value = map[key]
```

也可以用同样的运算符来改变一个可变 map 的元素：

```
mutableMap[key] = newValue
```

来看看它是如何工作的。在 Kotlin 中，下标运算符是一个约定。使用下标运算符读取元素会被转换为 get 运算符方法的调用，并且写入元素将调用 set。Map 和 MutableMap 的接口已经定义了这些方法。让我们看看如何给自定义的类添加类似的方法。

可以使用方括号来引用点的坐标：p[0] 访问 X 坐标，p[1] 访问 Y 坐标。下面是它的实现和调用：

代码清单 7.9 实现 get 约定

```
operator fun Point.get(index: Int): Int {
    return when(index) {
        0 -> x
        1 -> y
        else ->
            throw IndexOutOfBoundsException("Invalid coordinate $index")
    }
}
```

定义一个名为“get”的运算符函数

根据给出的 index 返回对应的坐标

```
>>> val p = Point(10, 20)
>>> println(p[1])
20
```

你只需要定义一个名为 get 的函数，并标记 operator。之后，像 p[1] 这样的表达式，其中 p 具有类型 Point，将被转换为 get 方法的调用，如图 7.6 所示。

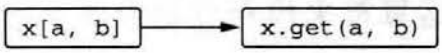


图 7.6 方括号的访问会被转换为 get 函数的调用

注意，get 的参数可以是任何类型，而不只是 Int。例如，当你对 map 使用下标运算符时，参数类型是键的类型，它可以是任意类型。还可以定义具有多个参数的 get 方法。例如，如果要实现一个类来表示二维数组或矩阵，你可以定义一个方法，例如 operator fun get(rowIndex: Int, colIndex: Int)，然后用 matrix [row,col] 来调用。如果需要使用不同的键类型访问集合，也可以使用不同的参数类型定义多个重载的 get 方法。

我们也可以用类似的方式定义一个函数，这样就可以使用方括号语法更改给定下标处的值。Point 类是不可变的，所以定义 Point 的这种方法是没有意义的。作为例子，我们来定义另一个类来表示一个可变的点。

代码清单 7.10 实现 set 的约定方法

```
data class MutablePoint(var x: Int, var y: Int)

operator fun MutablePoint.set(index: Int, value: Int) {
    when(index) {
        0 -> x = value
        1 -> y = value
        else ->
            throw IndexOutOfBoundsException("Invalid coordinate $index")
    }
}

>>> val p = MutablePoint(10, 20)
>>> p[1] = 42
>>> println(p)
MutablePoint(x=10, y=42)
```

定义一个名为“set”的运算符函数

根据给出的 index 修改对应的坐标

这个例子也很简单：只需要定义一个名为 set 的函数，就可以在赋值语句中使用下标运算符。set 的最后一个参数用来接收赋值语句中（等号）右边的值，其他参数作为方括号内的下标，如图 7.7 所示。



图 7.7 方括号的赋值操作将会转换为 set 函数的调用

7.3.2 “in”的约定

集合支持的另一个运算符是 in 运算符，用于检查某个对象是否属于集合。相应的函数叫作 contains。我们来实现一下，使用 in 运算符来检查点是否属于一个矩形。

代码清单 7.11 实现 in 的约定

```
data class Rectangle(val upperLeft: Point, val lowerRight: Point)

operator fun Rectangle.contains(p: Point): Boolean {
    return p.x in upperLeft.x until lowerRight.x &&
           p.y in upperLeft.y until lowerRight.y
}

>>> val rect = Rectangle(Point(10, 20), Point(50, 50))
>>> println(Point(20, 30) in rect)
true
>>> println(Point(5, 5) in rect)
false
```

构建一个区间，检查坐标“x”是否属于这个区间

使用“until”函数来构建一个开区间

in 右边的对象将会调用 contains 函数，in 左边的对象将会作为函数入参。

在 Rectangle.contains 的实现中，我们用到了的标准库的 until 函数，来构建一个开区间，然后使用运算符 in 来检查某个点是否属于这个区间。

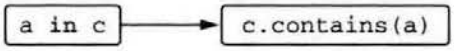


图 7.8 in 操作将会转换为 contains 函数的调用

开区间是不包括最后一个点的区间。例如，如果用 10..20 构建一个普通的区间（闭区间），该区间则包括 10 到 20 的所有数字，包括 20。开区间 10 until 20 包括从 10 到 19 的数字，但不包括 20。矩形类通常定义成这样，它的底部和右侧坐标不是矩形的一部分，因此在这里使用开区间是合适的。

7.3.3 rangeTo 的约定

要创建一个区间，请使用 .. 语法：举个例子，1..10 代表所有从 1 到 10 的数字。在 2.4.2 节你已经看到过区间的使用，现在让我们来研究一下创建它的约定。.. 运算符是调用 rangeTo 函数的一个简洁方法（如图 7.9 所示）。

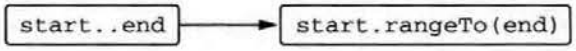


图 7.9 .. 运算符将被转换为 rangeTo 函数的调用

rangeTo 函数返回一个区间。你可以为自己的类定义这个运算符。但是，如果该类实现了 Comparable 接口，那么不需要了：你可以通过 Kotlin 标准库创建一个任意可比较元素的区间，这个库定义了可以用于任何可比较元素的 rangeTo 函数：

```
operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>
```

这个函数返回一个区间，可以用来检测其他一些元素是否属于它。

作为例子，我们用 LocalDate（Java8 标准库中有定义）来构建一个日期的区间。

代码清单 7.12 处理日期的区间

```
>>> val now = LocalDate.now()
>>> val vacation = now..now.plusDays(10)
>>> println(now.plusWeeks(1) in vacation)
true
```

创建一个从今天开始的 10 天的区间

检测一个特定的日期是否属于这个区间

now..now.plusDays(10) 表达式将会被编译器转换为 now.rangeTo(now.plusDays(10))。rangeTo 并不是 LocalDate 的成员函数，而是 Comparable 的一个扩展函数，如前面所展现的那样。

rangeTo 运算符的优先级低于算术运算符，但是最好把参数括起来以免混淆：

```
>>> val n = 9
>>> println(0..(n + 1))
0..10
```

可以写成 0..n + 1，但括起来更清晰一点

还要注意，表达式 0..n.forEach{} 不会被编译，因为必须把区间表达式括起来才能调用它的方法：

```
>>> (0..n).forEach { print(it) }
0123456789
```

把区间括起来，来调用它的方法

现在让我们来讨论一下如何使用约定，来遍历一个集合或是区间。

7.3.4 在“for”循环中使用“iterator”的约定

正如我们在第 2 章中讨论的，在 Kotlin 中，for 循环中也可以使用 in 运算符，和做区间检查一样。但是在这种情况下它的含义是不同的：它被用来执行迭代。这意味着一个诸如 for(x in list) {...} 将被转换成 list.iterator() 的调用，然后就像在 Java 中一样，在它上面重复调用 hasNext 和 next 方法。

请注意，在 Kotlin 中，这也是一种约定，这意味着 iterator 方法可以被定义为扩展函数。这就解释了为什么可以遍历一个常规的 Java 字符串：标准库已经为 CharSequence 定义了一个扩展函数 iterator，而它是 String 的父类：

```
operator fun CharSequence.iterator(): CharIterator

>>> for (c in "abc") {}
```

这个库函数让迭代字符串成为可能

可以为自己的类定义 iterator 方法。例如，可以这样定义方法来遍历日期。

代码清单 7.13 实现日期区间的迭代器

```
operator fun ClosedRange<LocalDate>.iterator(): Iterator<LocalDate> =
        object : Iterator<LocalDate> {
            var current = start

            override fun hasNext() =
                current <= endInclusive

            override fun next() = current.apply {
                current = plusDays(1)
            }
        }

>>> val newYear = LocalDate.ofYearDay(2017, 1)
>>> val daysOff = newYear.minusDays(1)..newYear
```

这个对象实现了遍历 LocalDate 元素的 Iterator

注意，这里日期用到了 compareTo 约定

在修改前返回当前日期作为结果

把当前日期增加一天

```
>>> for (dayOff in daysOff) { println(dayOff) }
2016-12-31
2017-01-01
```

对应的 iterator 函数实现后，遍历 daysOff

请注意如何在自定义区间类型上定义 iterator 方法：使用 LocalDate 作为类型参数。如上一节所示，rangeTo 库函数返回一个 ClosedRange 的实例，并且 ClosedRange<LocalDate> 的 iterator 扩展允许在 for 循环中使用区间的实例。

7.4 解构声明和组件函数

当我们在 4.3.2 节中讨论数据类时，我们提到过之后会研究它的功能。现在你已经熟悉了约定的原则，我们来看一下最后的功能：解构声明。这个功能允许你展开单个复合值，并使用它来初始化多个单独的变量。

来看看它是怎样工作的：

```
>>> val p = Point(10, 20)
>>> val (x, y) = p
>>> println(x)
10
>>> println(y)
20
```

声明变量 x、y，然后用 p 的组件来初始化

一个解构声明看起来像一个普通的变量声明，但它在括号中有多个变量。

事实上，解构声明再次用到了约定的原理。要在解构声明中初始化每个变量，将调用名为 componentN 的函数，其中 N 是声明中变量的位置。换句话说，前面的例子可以被转换成如图 7.10 所示的样子。

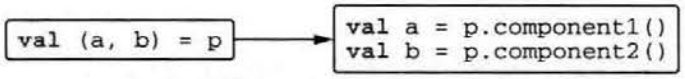


图 7.10 解构声明被转换为 componentN 函数的调用

对于数据类，编译器为每个在主构造方法中声明的属性生成一个 componentN 函数。下面的例子显示了如何手动为非数据类声明这些功能：

```
class Point(val x: Int, val y: Int) {
    operator fun component1() = x
    operator fun component2() = y
}
```

解构声明主要使用场景之一，是从一个函数返回多个值，这个非常有用。如果要这样做，可以定义一个数据类来保存返回所需的值，并将它作为函数的返回类型。

在调用函数后，可以用解构声明的方式，来轻松地展开它，使用其中的值。举个例子，让我们编写一个简单的函数，来将一个文件名分割成名字和扩展名。

代码清单 7.14 使用解构声明来返回多个值

```
data class NameComponents(val name: String,
                          val extension: String)

fun splitFilename(fullName: String): NameComponents {
    val result = fullName.split('.', limit = 2)
    return NameComponents(result[0], result[1])
}

>>> val (name, ext) = splitFilename("example.kt")
>>> println(name)
example
>>> println(ext)
kt
```

声明一个数据类来持有值

返回一个数据类型的实例

使用解构声明来展开这个类

如果你注意到 componentN 函数在数组和集合上也有定义，可以进一步改进这个代码。当你在处理已知大小的集合时，这是非常有用的。一个例子就是，用 split 来返回两个元素的列表。

代码清单 7.15 使用解构声明来处理集合

```
data class NameComponents(
        val name: String,
        val extension: String)

fun splitFilename(fullName: String): NameComponents {
    val (name, extension) = fullName.split('.', limit = 2)
    return NameComponents(name, extension)
}
```

当然，不可能定义无限数量的 componentN 函数，这样这个语法就可以与任意数量的集合一起工作，但这也没用。标准库只允许使用此语法来访问一个对象的前五个元素。

让一个函数能返回多个值有更简单的方法，是使用标准库中的 Pair 和 Triple 类。在语义表达上这种方式会差一点，因为这些类也不知道它返回的对象中包含什么，但因为不需要定义自己的类所以可以少写代码。

7.4.1 解构声明和循环

解构声明不仅可以用作函数中的顶层语句，还可以用在其他可以声明变量的地方，例如 in 循环。一个很好的例子，是枚举 map 中的条目。下面是一个小例子，

使用这个语法打印给定 map 中的所有条目。

代码清单 7.16 用解构声明来遍历 map

```
fun printEntries(map: Map<String, String>) {
    for ((key, value) in map) {
        println("$key -> $value")
    }
}

>>> val map = mapOf("Oracle" to "Java", "JetBrains" to "Kotlin")
>>> printEntries(map)
Oracle -> Java
JetBrains -> Kotlin
```

在 in 循环中用解构声明

这个简单的例子用到了两个 Kotlin 约定：一个是迭代一个对象，另一个是用于解构声明。Kotlin 标准库给 map 增加了一个扩展的 `iterator` 函数，用来返回 map 条目的迭代器。因此，与 Java 不同的是，可以直接迭代 map。它还包含 `Map.Entry` 上的扩展函数 `component1` 和 `component2`，分别返回它的键和值。实际上，前面的循环被转换成了这样的代码：

```
for (entry in map.entries) {
    val key = entry.component1()
    val value = entry.component2()
    // ...
}
```

这个例子再次印证了扩展函数对于约定的重要性。

7.5 重用属性访问的逻辑：委托属性

让我们用一个依赖于约定的功能来结束本章，Kotlin 中最独特和最强大的功能之一：委托属性。这个功能可以让你轻松实现这样的属性，它们处理起来比把值存储在支持字段中更复杂，却不用在每个访问器中都重复这样的逻辑。例如，这些属性可以把它们的值存储在数据库表中，在浏览器会话中，在一个 map 中等。

这个功能的基础是委托，这是一种设计模式，操作的对象不用自己执行，而是把工作委托给另一个辅助的对象。我们把辅助对象称为委托。当我们讨论类的委托的时候，你之前应该在 4.3.3 节中看到过这种模式。当这个模式应用于一个属性时，它也可以将访问器的逻辑委托给一个辅助对象。你可以手动实现它（稍后我们会有示例）或使用更好的解决方案：利用 Kotlin 的语言支持。我们先从大概的介绍开始，然后再看一些具体的例子。

7.5.1 委托属性的基本操作

委托属性的基本语法是这样的：

```
class Foo {
    var p: Type by Delegate()
}
```

属性 p 将它的访问器逻辑委托给了另一个对象：这里是 Delegate 类的一个新的实例。通过关键字 by 对其后的表达式求值来获取这个对象，关键字 by 可以用于任何符合属性委托约定规则的对象。

编译器创建一个隐藏的辅助属性，并使用委托对象的实例进行初始化，初始属性 p 会委托给该实例。为了简单起见，我们把它称为 delegate：

```
class Foo {
    private val delegate = Delegate()
    var p: Type
        set(value: Type) = delegate.setValue(..., value)
        get() = delegate.getValue(...)
}
```

编译器会自动生成一个辅助属性

“p”的访问都会调用对应的“delegate”的 getValue 和 setValue 方法

按照约定，Delegate 类必须具有 getValue 和 setValue 方法（后者仅适用于可变属性）。像往常一样，它们可以是成员函数，也可以是扩展函数。为了让例子看起来更简洁，这里我们省略掉了参数。准确的（函数）签名将在将在本章后面介绍。Delegate 类的简单实现差不多应该是这样的：

```
class Delegate {
    operator fun getValue(...) { ... }

    operator fun setValue(..., value: Type) { ... }
}
```

getValue 包含了实现 getter 的逻辑

setValue 包含了实现 setter 的逻辑

```
class Foo {
    var p: Type by Delegate()
}
```

关键字“by”把属性关联上委托对象

```
>>> val foo = Foo()
>>> val oldValue = foo.p
>>> foo.p = newValue
```

通过调用 delegate.getValue(...)来实现属性的修改

通过调用 delegate.setValue(..., newValue)来实现属性的修改

可以把 foo.p 作为普通的属性使用，事实上，它将调用 Delegate 类型的辅助属性的方法。为了研究这种机制如何在实践中使用，我们首先看一个委托属性展示威力的例子：库对惰性初始化的支持。之后，我们再探讨如何定义自己的委托属性，

以及什么时候用它。

7.5.2 使用委托属性：惰性初始化和“by lazy()”

*惰性初始化*是一种常见的模式，直到在第一次访问该属性的时候，才根据需要创建对象的一部分。当初始化过程消耗大量资源并且在使用对象时并不总是需要数据时，这个非常有用。

举个例子，一个 Person 类，可以用来访问一个人写的邮件列表。邮件存储在数据库中，访问比较耗时。你希望只有在首次访问时才加载邮件，并只执行一次。假设你已经有函数 loadEmails，用来从数据库中检索电子邮件：

```
class Email { /*...*/ }
fun loadEmails(person: Person): List<Email> {
    println("Load emails for ${person.name}")
    return listOf(/*...*/)
}
```

下面展示了如何使用额外的 _emails 属性来实现惰性加载，在没有加载之前为 null，然后加载为邮件列表。

代码清单 7.17 使用支持属性来实现惰性初始化

```
class Person(val name: String) {
    private var _emails: List<Email>? = null          <-- "_emails"属性用来保存数据，关联委托

    val emails: List<Email>
        get() {
            if (_emails == null) {                    <-- 访问时加载邮件
                _emails = loadEmails(this)
            }
            return _emails!!                          <-- 如果已经加载，就直接返回
        }
}
>>> val p = Person("Alice")
>>> p.emails                                          <-- 第一次访问会加载邮件
Load emails for Alice
>>> p.emails
```

这里使用了所谓的*支持属性*技术。你有一个属性，_emails，用来存储这个值，而另一个 emails，用来提供对属性的读取访问。你需要使用两个属性，因为属性具有不同的类型：_emails 可以为空，而 emails 为非空。这种技术经常会使用到，值得熟练掌握。

但这个代码有点啰唆：要是有几个惰性属性那得有多长。而且，它并不总是正常运行：这个实现不是线程安全的。Kotlin 提供了更好的解决方案。

使用委托属性会让代码变得简单得多，可以封装用于存储值的支持属性和确保该值只被初始化一次的逻辑。在这里可以使用标准库函数 lazy 返回的委托。

代码清单 7.18 用委托属性来实现惰性初始化

```
class Person(val name: String) {
    val emails by lazy { loadEmails(this) }
}
```

lazy 函数返回一个对象，该对象具有一个名为 getValue 且签名正确的方法，因此可以把它与 by 关键字一起使用来创建一个委托属性。lazy 的参数是一个 lambda，可以调用它来初始化这个值。默认情况下，lazy 函数是线程安全的，如果需要，可以设置其他选项来告诉它要使用哪个锁，或者完全避开同步，如果该类永远不会在多线程环境中使用。

在下一节中，我们将详细介绍委托属性机制是怎样工作的，并讨论这里发挥作用的约定。

7.5.3 实现委托属性

要了解委托属性的实现方式，让我们来看另一个例子：当一个对象的属性更改时通知监听器。这在许多不同的情况下都很有用：例如，当对象显示在 UI 时，你希望在对象变化时 UI 能自动刷新。Java 具有用于此类通知的标准机制：PropertyChangeSupport 和 PropertyChangeEvent 类。让我们看看在 Kotlin 中在不使用委托属性的情况下，该如何使用它们，然后我们再将代码重构为用委托属性的方式。

PropertyChangeSupport 类维护了一个监听器列表，并向它们发送 PropertyChangeEvent 事件。要使用它，你通常需要把这个类的一个实例存储为 bean 类的一个字段，并将属性更改的处理委托给它。

为了避免要在每个类中去添加这个字段，你需要创建一个小的工具类，用来存储 PropertyChangeSupport 的实例并监听属性更改。之后，你的类会继承这个工具类，以访问 changeSupport。

代码清单 7.19 使用 PropertyChangeSupport 的工具类

```
open class PropertyChangeAware {
    protected val changeSupport = PropertyChangeSupport(this)

    fun addPropertyChangeListener(listener: PropertyChangeListener) {
        changeSupport.addPropertyChangeListener(listener)
    }
```

```
    fun removePropertyChangeListener(listener: PropertyChangeListener) {
        changeSupport.removePropertyChangeListener(listener)
    }
}
```

现在我们来写一个 Person 类，定义一个只读属性（作为一个人的名称，一般不会随时更改）和两个可写属性：年龄和工资。 当这个人的年龄或工资发生变化时，这个类将通知它的监听器。

代码清单 7.20 手工实现属性修改的通知

```
class Person(
        val name: String, age: Int, salary: Int
) : PropertyChangeAware() {

    var age: Int = age
        set(newValue) {
            val oldValue = field
            field = newValue
            changeSupport.firePropertyChange(
                    "age", oldValue, newValue)
        }
    var salary: Int = salary
        set(newValue) {
            val oldValue = field
            field = newValue
            changeSupport.firePropertyChange(
                    "salary", oldValue, newValue)
        }
}

>>> val p = Person("Dmitry", 34, 2000)
>>> p.addPropertyChangeListener(
...     PropertyChangeListener { event ->
...         println("Property ${event.propertyName} changed " +
...                 "from ${event.oldValue} to ${event.newValue}")
...     }
... )
>>> p.age = 35
Property age changed from 34 to 35
>>> p.salary = 2100
Property salary changed from 2000 to 2100
```

“field”标识符允许你访问属性背后的支持字段

当属性变化时，通知监听器

关联监听器，用于监听属性修改

注意，这里的代码是如何使用 field 标识符来访问 age 和 salary 属性的支持字段的，与 4.2.4 节所讨论的一样。

setter 中有很多重复的代码。我们来尝试提取一个类，用来存储这个属性的值并发起通知。

代码清单 7.21 通过辅助类来实现属性变化的通知

```
class ObservableProperty(
    val propName: String, var propValue: Int,
    val changeSupport: PropertyChangeSupport
) {
    fun getValue(): Int = propValue
    fun setValue(newValue: Int) {
        val oldValue = propValue
        propValue = newValue
        changeSupport.firePropertyChange(propName, oldValue, newValue)
    }
}

class Person(
    val name: String, age: Int, salary: Int
) : PropertyChangeAware() {

    val _age = ObservableProperty("age", age, changeSupport)
    var age: Int
        get() = _age.getValue()
        set(value) { _age.setValue(value) }

    val _salary = ObservableProperty("salary", salary, changeSupport)
    var salary: Int
        get() = _salary.getValue()
        set(value) { _salary.setValue(value) }
}
```

现在，你应该已经差不多理解了在 Kotlin 中，委托属性是如何工作的。你创建了一个保存属性值的类，并在修改属性时自动触发更改通知。你删除了重复的逻辑代码，但是需要相当多的样板代码来为每个属性创建 `ObservableProperty` 实例，并把 getter 和 setter 委托给它。Kotlin 的委托属性功能可以让你摆脱这些样板代码。但是在此之前，你需要更改 `ObservableProperty` 方法的签名，来匹配 Kotlin 约定所需的方法。

代码清单 7.22 `ObservableProperty` 作为属性委托

```
class ObservableProperty(
    var propValue: Int, val changeSupport: PropertyChangeSupport
) {
    operator fun getValue(p: Person, prop: KProperty<*>): Int = propValue

    operator fun setValue(p: Person, prop: KProperty<*>, newValue: Int) {
        val oldValue = propValue
        propValue = newValue
        changeSupport.firePropertyChange(prop.name, oldValue, newValue)
    }
}
```

与之前的版本相比，这次代码做了一些更改：

- 现在，按照约定的需要，getValue 和 setValue 函数被标记了 operator。
- 这些函数加了两个参数：一个用于接收属性的实例，用来设置或读取属性，另一个用于表示属性本身。这个属性类型为 KProperty。我们将在 10.2 节中详细介绍它；现在，你就可以使用 KProperty.name 的方式来访问该属性的名称。
- 把 name 属性从主构造方法中删除了，因为现在已经可以通过 KProperty 访问属性名称。

终于，你可以见识 Kotlin 委托属性的神奇了。来看看代码变短多少？

代码清单 7.23 使用委托属性来绑定更改通知

```
class Person(
    val name: String, age: Int, salary: Int
) : PropertyChangeAware() {

    var age: Int by ObservableProperty(age, changeSupport)
    var salary: Int by ObservableProperty(salary, changeSupport)
}
```

通过关键字 by，Kotlin 编译器会自动执行之前版本的代码中手动完成的操作。如果把这份代码与以前版本的 Person 类进行比较：使用委托属性时生成的代码非常类似。右边的对象被称为委托。Kotlin 会自动将委托存储在隐藏的属性中，并在访问或修改属性时调用委托的 getValue 和 setValue。

你不用手动去实现可观察的属性逻辑，可以使用 Kotlin 标准库，它已经包含了类似于 ObservableProperty 的类。标准库类和这里使用的 PropertyChangeSupport 类没有耦合，因此你需要传递一个 lambda，来告诉它如何通知属性值的更改。可以这样做：

代码清单 7.24 使用 Delegates.observable 来实现属性修改的通知

```
class Person(
    val name: String, age: Int, salary: Int
) : PropertyChangeAware() {

    private val observer = {
        prop: KProperty<*>, oldValue: Int, newValue: Int ->
        changeSupport.firePropertyChange(prop.name, oldValue, newValue)
    }

    var age: Int by Delegates.observable(age, observer)
    var salary: Int by Delegates.observable(salary, observer)
}
```

by 右边的表达式不一定是新创建的实例，也可以是函数调用、另一个属性或任何其他表达式，只要这个表达式的值，是能够被编译器用正确的参数类型来调用 getValue 和 setValue 的对象。与其他约定一样，getValue 和 setValue 可以是对象自己声明的方法或扩展函数。

注意，为了让示例保持简单，我们只展示了如何使用类型为 Int 的委托属性。委托属性机制其实是通用的，适用于任何其他类型。

7.5.4 委托属性的变换规则

让我们来总结一下委托属性是怎么工作的，假设你已经有了一个具有委托属性的类：

```
class C {
    var prop: Type by MyDelegate()
}

val c = C()
```

MyDelegate 实例会被保存到一个隐藏的属性中，它被称为 <delegate>。编译器也将用一个 KProperty 类型的对象来代表这个属性，它被称为 <property>。

编译器生成的代码如下：

```
class C {
    private val <delegate> = MyDelegate()

    var prop: Type
       get() = <delegate>.getValue(this, <property>)
       set(value: Type) = <delegate>.setValue(this, <property>, value)
}
```

因此，在每个属性访问器中，编译器都会生成对应的 getValue 和 setValue 方法，如图 7.11 所示。

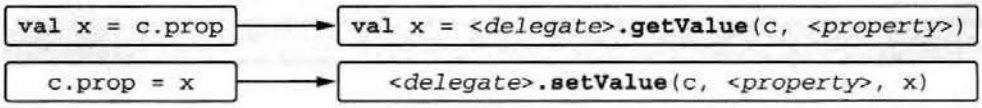


图 7.11 当访问属性时，会调用 <delegate> 的 getValue 和 setValue 函数

这个机制非常简单，但它可以实现许多有趣的场景。你可以自定义存储该属性值的位置（map、数据库表或者用户会话的 Cookie 中），以及在访问该属性时做点什么（比如添加验证、更改通知等）。所有这一切都可以用紧凑的代码完成。我们再来看看标准库中委托属性的另一个用法，然后看看如何在自己的框架中使用它们。

7.5.5 在 map 中保存属性值

委托属性发挥作用的另一种常见用法，是用在有动态定义的属性集的对象中。这样的对象有时被称为自订 (*expando*) 对象。例如，考虑一个联系人管理系统，可以用来存储有关联系人的任意信息。系统中的每个人都有一些属性需要特殊处理(例如名字)，以及每个人特有的数量任意的额外属性（例如，最小的孩子的生日）。

实现这种系统的一种方法是将人的所有属性存储在 map 中，不确定提供属性，来访问需要特殊处理的信息。来看个例子：

代码清单 7.25 定义一个属性，把值存到 map

```
class Person {
    private val _attributes = hashMapOf<String, String>()
    fun setAttribute(attrName: String, value: String) {
        _attributes[attrName] = value
    }

    val name: String
        get() = _attributes["name"]!!
}

>>> val p = Person()
>>> val data = mapOf("name" to "Dmitry", "company" to "JetBrains")
>>> for ((attrName, value) in data)
...     p.setAttribute(attrName, value)
>>> println(p.name)
Dmitry
```

从 map 手动检索属性

这里使用了一个通用的 API 来把数据加载到对象中（在实际项目中，可以是 JSON 反序列化或类似的方法），然后使用特定的 API 来访问一个属性的值。把它改为委托属性非常简单，可以直接将 map 放在 by 关键字后面。

代码清单 7.26 使用委托属性把值存到 map 中

```
class Person {
    private val _attributes = hashMapOf<String, String>()

    fun setAttribute(attrName: String, value: String) {
        _attributes[attrName] = value
    }

    val name: String by _attributes
}
```

把 map 作为委托属性

因为标准库已经在标准 Map 和 MutableMap 接口上定义了 getValue 和 setValue 扩展函数，所以这里可以直接这样用。属性的名称将自动用作在

map 中的键，属性值作为 map 中的值。在代码清单 7.25 中，p.name 隐藏了 _attributes.getValue(p, prop) 的调用，这里变为 _attributes [prop.name]。

7.5.6 框架中的委托属性

更改存储和修改属性的方式对框架的开发人员非常有用。在 1.3.1 节中，已经展示了数据库框架使用委托属性的示例。这一节将展示一个类似的例子，并阐述它的工作原理。

假设数据库中 Users 的表包含两列数据：字符串类型的 name 和整型的 age。可以在 Kotlin 中定义 Users 和 User 类。在 Kotlin 代码中，所有存储在数据库中的用户实体的加载和更改都可以通过 User 类的实例来操作。

代码清单 7.27 使用委托属性来访问数据库列

```
object Users : IdTable() {
    val name = varchar("name", length = 50).index()
    val age = integer("age")
}

class User(id: EntityID) : Entity(id) {
    var name: String by Users.name
    var age: Int by Users.age
}
```

Users 对应数据库中的表

name 和 age 对应数据库表中的列

每个 User 实例对应表中的一个实体

“name” 的值是数据库中对应那个用户的值

Users 对象描述数据库的一个表；它被声明为一个对象，因为它对应整个表，所以只需要一个实例。对象的属性表示数据表的列。

User 类的基类 Entity，包含了实体的数据库列与值的映射。特定 User 的属性拥有这个用户在数据库中指定的值 name 和 age。

框架用起来会特别方便，因为访问属性会自动从 Entity 类的映射中检索相应的值，而修改过的对象会被标记成脏数据，在需要时可将其保存到数据库中。可以在 Kotlin 代码中编写 user.age += 1，数据库中的相应实体将自动更新。

现在，你已经充分了解了如何实现具有这种 API 的框架。每个实体属性（name，age）都实现为委托属性，使用列对象（Users.name，Users.age）作为委托：

```
class User(id: EntityID) : Entity(id) {
    var name: String by Users.name
    var age: Int by Users.age
}
```

Users.name 是 “name” 属性的委托

让我们来看看怎样显式地指定列的类型：

```
object Users : IdTable() {
    val name: Column<String> = varchar("name", 50).index()
    val age: Column<Int> = integer("age")
}
```

至于 Column 类，框架已经定义了 getValue 和 setValue 方法，满足 Kotlin 的委托约定：

```
operator fun <T> Column<T>.getValue(o: Entity, desc: KProperty<*>): T {
    // 从数据库中获取值
}
operator fun <T> Column<T>.setValue(o: Entity, desc: KProperty<*>, value: T) {
    // 更新值到数据库
}
```

可以使用 Column 属性（Users.name）作为被委托属性（name）的委托。当在代码中写入 user.age + = 1 时，代码的执行将类似于 user.ageDelegate.setValue(user.ageDelegate.getValue() + 1)（省略了属性和对象实例的参数）的操作。getValue 和 setValue 方法负责检索和更新数据库中的信息。

完整的代码示例可以在 Exposed 框架的源代码中找到（https://github.com/JetBrains/Exposed）。我们将在第 11 章中再次用到这个框架，在那里将会探讨使用到的 DSL 设计技术。

7.6 小结

- Kotlin 允许使用对应名称的函数来重载一些标准的数学运算，但是不能定义自己的运算符。
- 比较运算符映射为 equals 和 compareTo 方法的调用。
- 通过定义名为 get、set 和 contains 的函数，就可以让你自己的类与 Kotlin 的集合一样，使用 [] 和 in 运算符。
- 可以通过约定来创建区间，以及迭代集合和数组。
- 解构声明可以展开单个对象用来初始化多个变量，这可以方便地用来从函数返回多个值。它们可以自动处理数据类，可以通过给自己的类定义名为 componentN 的函数来支持。
- 委托属性可以用来重用逻辑，这些逻辑控制如何存储、初始化、访问和修改属性值，这是用来构建框架的一个强大的工具。
- lazy 标准库函数提供了一种实现惰性初始化属性的简单方法。
- Delegates.observable 函数可以用来添加属性更改的观察者。
- 委托属性可以使用任意 map 来作为属性委托，来灵活来处理具有可变属性集的对象。

8 高阶函数：Lambda作为形参和返回值

本章内容包括

- 函数类型
- 高阶函数及其在组织代码过程中的应用
- 内联函数
- 非局部返回和标签
- 匿名函数

在第 5 章我们已经初步认识了 lambda，探讨了 lambda 的基本概念和标准库中使用 lambda 的函数。lambda 是一个构建抽象概念的强大工具，当然它的能力并不局限于标准库中提供的集合和其他类。在这一章你将学到如何创建高阶函数——属于你自己的，使用 lambda 作为参数或者返回值的函数。你将会看到高阶函数如何帮助你简化代码，去除代码重复，以及构建漂亮的抽象概念。你也将认识内联函数——Kotlin 的一个强大特性，它能够消除 lambda 带来的性能开销，还能够使 lambda 内的控制流更加灵活。

8.1 声明高阶函数

这一章新的关键概念就是高阶函数。按照定义，高阶函数就是以另一个函数作为参数或者返回值的函数。在 Kotlin 中，函数可以用 lambda 或者函数引用来表示。

因此，任何以 lambda 或者函数引用作为参数的函数，或者返回值为 lambda 或函数引用的函数，或者两者都满足的函数都是高阶函数。例如，标准库中的 filter 函数将一个判断式函数作为参数，因此它就是一个高阶函数：

```
list.filter { x > 0 }
```

在第 5 章，我们已经见过很多 Kotlin 标准库中的其他高阶函数：map、with，等等。现在我们将学习如何在自己的代码中声明高阶函数。要达到这个目标，首先必须要知道什么是*函数类型*。

8.1.1 函数类型

为了声明一个以 lambda 作为实参的函数，你需要知道如何声明对应形参的类型。在这之前，我们先来看一个简单的例子，把 lambda 表达式保存在局部变量中。其实我们已经见过在不声明类型的情况下如何做到这一点，这依赖于 Kotlin 的类型推导：

```
val sum = { x: Int, y: Int -> x + y }
val action = { println(42) }
```

在这个例子中，编译器推导出 sum 和 action 这两个变量具有函数类型。现在我们来看看这些变量的显式类型声明是什么样的：

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
val action: () -> Unit = { println(42) }
```

有两个 Int 型参数和 Int 型返回值的函数

没有参数和返回值的函数

声明函数类型，需要将函数参数类型放在括号中，紧接着是一个箭头和函数的返回类型，如图 8.1 所示。

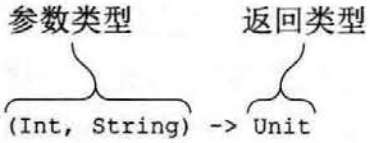


图 8.1 Kotlin 中的函数类型语法

你应该还记得，Unit 类型用于表示函数不返回任何有用的值。在声明一个普通的函数时，Unit 类型的返回值是可以省略的，但是一个函数类型声明总是需要一个显式的返回类型，所以在这种场景下 Unit 是不能省略的。

注意，在 lambda 表示式 { x, y -> x + y } 中是如何省略参数 x、y 的类型的。因为它们的类型已经在函数类型的变量声明部分指定了，不需要在 lambda 本身的定义当中再重复声明。

就像其他方法一样，函数类型的返回值也可以标记为可空类型：

```
var canReturnNull: (Int, Int) -> Int? = { null }
```

也可以定义一个函数类型的可空变量。为了明确表示是变量本身可空，而不是函数类型的返回类型可空，你需要将整个函数类型的定义包含在括号内并在括号后面添加一个问号：

```
var funOrNull: ((Int, Int) -> Int)? = null
```

注意这个例子和前一个例子的微妙区别。如果省略了括号，声明的将会是一个返回值可空的函数类型，而不是一个可空的函数类型的变量。

函数类型的参数名

可以为函数类型声明中的参数指定名字：

```
fun performRequest(
        url: String,
        callback: (code: Int, content: String) -> Unit
) {
    /*...*/
}

>>> val url = "http://kotl.in"
>>> performRequest(url) { code, content -> /*...*/ }
>>> performRequest(url) { code, page -> /*...*/ }
```

函数类型的参数现在有了名字

可以使用 API 中提供的参数名字作为 lambda 参数的名字……

……或者你可以改变参数的名字

参数名称不会影响类型的匹配。当你声明一个 lambda 时，不必使用和函数类型声明中一模一样的参数名称，但命名会提升代码可读性并且能用于 IDE 的代码补全。

8.1.2 调用作为参数的函数

知道了怎样声明一个高阶函数，现在我们来讨论如何去实现它。第一个例子会尽量简单并且使用之前的 lambda sum 同样的声明。这个函数实现两个数字 2 和 3 的任意操作，然后打印结果。

代码清单 8.1 定义一个简单高阶函数

```
fun twoAndThree(operation: (Int, Int) -> Int) {
```

定义一个函数类型的参数

```
    val result = operation(2, 3)
    println("The result is $result")
}

>>> twoAndThree { a, b -> a + b }
The result is 5
>>> twoAndThree { a, b -> a * b }
The result is 6
```

调用函数类型的参数

调用作为参数的函数和调用普通函数的语法是一样的：把括号放在函数名后，并把参数放在括号内。

来看一个更有趣的例子，我们来实现最常用的标准库函数：filter 函数。为了让事情简单一点，将实现基于 String 类型的 filter 函数，但和作用于集合的泛型版本的原理是相似的。函数声明如图 8.2 所示。

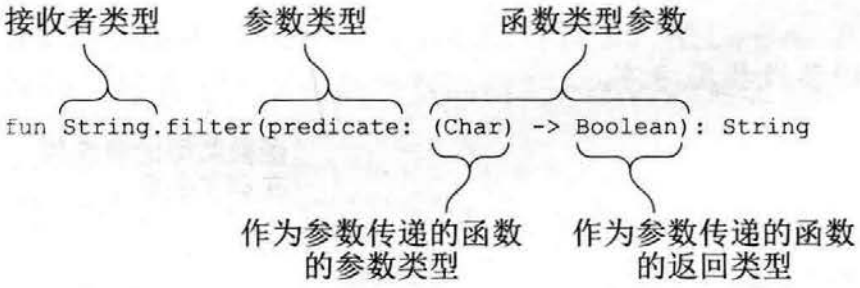


图 8.2 filter 函数的声明，以一个判断式作为参数

filter 函数以一个判断式作为参数。判断式的类型是一个函数，以字符作为参数并返回 boolean 类型的值。如果要让传递给判断式的字符出现在最终返回的字符串中，判断式需要返回 true，反之返回 false。以下是这个方法的实现。

代码清单 8.2 实现一个简单版本的 filter 函数

```
fun String.filter(predicate: (Char) -> Boolean): String {
    val sb = StringBuilder()
    for (index in 0 until length) {
        val element = get(index)
        if (predicate(element)) sb.append(element)
    }
    return sb.toString()
}

>>> println("ab1c".filter { it in 'a'..'z' })
abc
```

调用作为参数传递给“predicate”的函数

传递一个 lambda 作为“predicate”参数

filter 函数的实现非常简单明了。它检查每一个字符是否满足判断式，如果满足就将字符添加到包含结果的 StringBuilder 中。

IntelliJ IDEA 小贴士 IntelliJ IDE 调试器支持智能地单步步入 lambda 中的

代码。如果我们单步调试之前的例子，你会看到程序在 filter 函数体和传递给函数的 lambda 之间来回移动执行，因为函数会处理输入列表当中的每一个元素。

8.1.3 在 Java 中使用函数类

其背后的原理是，函数类型被声明为普通的接口：一个函数类型的变量是 FunctionN 接口的一个实现。Kotlin 标准库定义了一系列的接口，这些接口对应于不同参数数量的函数：Function0<R>（没有参数的函数）、Function1<P1,R>（一个参数的函数），等等。每个接口定义了一个 invoke 方法，调用这个方法就会执行函数。一个函数类型的变量就是实现了对应的 FunctionN 接口的实现类的实例，实现类的 invoke 方法包含了 lambda 函数体。

在 Java 中可以很简单地调用使用了函数类型的 Kotlin 函数。Java 8 的 lambda 会被自动转换为函数类型的值。

```
/* Kotlin 声明 */
fun processTheAnswer(f: (Int) -> Int) {
    println(f(42))
}

/* Java */
>>> processTheAnswer(number -> number + 1);
43
```

在旧版的 Java 中，可以传递一个实现了函数接口中的 invoke 方法的匿名类的实例：

```
/* Java */
>>> processTheAnswer(
...     new Function1<Integer, Integer>() {
...         @Override
...         public Integer invoke(Integer number) {
...             System.out.println(number);
...             return number + 1;
...         }
...     });
43
```

在 Java 代码中使用函数类型（Java 8 以前）

在 Java 中可以很容易地使用 Kotlin 标准库中以 lambda 作为参数的扩展函数。但是要注意它们看起来并没有 Kotlin 中那么直观——必须要显式地传递一个接收者对象作为第一个参数：

```
/* Java */
>>> List<String> strings = new ArrayList();
>>> strings.add("42");
>>> CollectionsKt.forEach(strings, s -> {
...     System.out.println(s);
...     return Unit.INSTANCE;
... });
```

可以在 Java 代码中使用 Kotlin 标准库中的函数

必须要显式地返回一个 Unit 类型的值

在 Java 中，函数或者 lambda 可以返回 Unit。但因为在 Kotlin 中 Unit 类型是有一个值的，所以需要显式地返回它。一个返回 void 的 lambda 不能作为返回 Unit 的函数类型的实参，就像之前的例子中的 (String) -> Unit。

8.1.4 函数类型的参数默认值和 null 值

声明函数类型的参数的时候可以指定参数的默认值。要知道默认值的用处，我们回头看一看第 3 章讨论过的 joinToString 函数。以下是它的最终实现。

代码清单 8.3 使用了硬编码 toString 转换的 joinToString 函数

```
fun <T> Collection<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = ""
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(element)
    }

    result.append(postfix)
    return result.toString()
}
```

使用默认的 toString 方法将对象转换为字符串

这个实现很灵活，但是它并没有让你控制转换的关键点：集合中的元素是如何转换为字符串的。代码中使用了 StringBuilder.append(o:Any?)，它总是使用 toString 方法将对象转换为字符串。在大多数情况下这样就可以了，但并不总是这样。我们现在已经知道可以传递一个 lambda 去指定如何将对象转换为字符串。但是要求所有调用者都传递 lambda 是比较烦人的事情，因为大部分调用者使用默认的行为就可以了。为了解决这个问题，可以定义一个函数类型的参数并用一个 lambda 作为它的默认值。

代码清单 8.4 给函数类型的参数指定默认值

```
fun <T> Collection<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = "",
        transform: (T) -> String = { it.toString() }
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(transform(element))
    }

    result.append(postfix)
    return result.toString()
}

>>> val letters = listOf("Alpha", "Beta")
>>> println(letters.joinToString())
Alpha, Beta
>>> println(letters.joinToString { it.toLowerCase() })
alpha, beta
>>> println(letters.joinToString(separator = "! ", postfix = "! ",
...         transform = { it.toUpperCase() }))
ALPHA! BETA!
```

声明一个以 lambda 为默认值的函数类型的参数

调用作为实参传递给“transform”形参的函数

使用默认的转换函数

传递一个 lambda 作为参数

使用命名参数语法传递几个参数，包括一个 lambda

注意这是一个泛型函数：它有一个类型参数 T 表示集合中的元素的类型。Lambda transform 将接收这个类型的参数。

声明函数类型的默认值并不需要特殊的语法——只需要把 lambda 作为值放在 = 号后面。上面的例子展示了不同的函数调用方式：省略整个 lambda（使用默认的 toString() 做转换），在括号以外传递 lambda，或者以命名参数形式传递。

另一种选择是声明一个参数为可空的函数类型。注意这里不能直接调用作为参数传递进来的函数：Kotlin 会因为检测到潜在的空指针异常而导致编译失败。一种可选的办法是显式地检查 null：

```
fun foo(callback: (() -> Unit)?) {
    // ...
    if (callback != null) {
        callback()
    }
}
```

还有一个更简单的版本，它利用了这样一个事实，函数类型是一个包含 invoke 方法的接口的具体实现。作为一个普通方法，invoke 可以通过安全调用

语法被调用：callback?.invoke()。

下面介绍使用这项技术重写 joinToString 函数。

代码清单 8.5 使用函数类型的可空参数

```
fun <T> Collection<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = "",
        transform: ((T) -> String)? = null
): String {
    val result = StringBuilder(prefix)
    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
        val str = transform?.invoke(element)
            ?: element.toString()
        result.append(str)
    }

    result.append(postfix)
    return result.toString()
}
```

声明一个函数类型的可空参数

使用安全调用语法调用函数

使用 Elvis 运算符处理回调没有被指定的情况

现在你已经知道了如何编写接收另外的函数作为参数的函数。接下来让我们看看另一种类型的高阶函数：返回值为其他函数的函数。

8.1.5 返回函数的函数

从函数中返回另一个函数并没有将函数作为参数传递那么常用，但它仍然非常有用。想象一下程序中的一段逻辑可能会因为程序的状态或者其他条件而产生变化——比如说，运输费用的计算依赖于选择的运输方式。可以定义一个函数用来选择恰当的逻辑变体并将它作为另一个函数返回。以下是具体的代码。

代码清单 8.6 定义一个返回函数的函数

```
enum class Delivery { STANDARD, EXPEDITED }

class Order(val itemCount: Int)

fun getShippingCostCalculator(
        delivery: Delivery): (Order) -> Double {
    if (delivery == Delivery.EXPEDITED) {
        return { order -> 6 + 2.1 * order.itemCount }
    }

    return { order -> 1.2 * order.itemCount }
}
```

声明一个返回函数的函数

返回 lambda

```
}

>>> val calculator =
...     getShippingCostCalculator(Delivery.EXPEDITED)
>>> println("Shipping costs ${calculator(Order(3))}")
Shipping costs 12.3
```

将返回的函数保存在变量中

调用返回的函数

声明一个返回另一个函数的函数，需要指定一个函数类型作为返回类型。在代码清单 8.6 中，getShippingCostCalculator 返回了一个函数，这个函数以 Order 作为参数并返回一个 Double 类型的值。要返回一个函数，需要写一个 return 表达式，跟上一个 lambda、一个成员引用，或者其他的函数类型的表达式，比如一个（函数类型的）局部变量。

来看另一个返回函数的函数非常实用的例子。假设你正在开发一个带 GUI 的联系人管理应用，你需要通过 UI 的状态来决定显示哪一个联系人。例如，可以在 UI 上输入一个字符串，然后只显示那些姓名以这个字符串开头的联系人；还可以隐藏没有电话号码的联系人。我们用 ContactListFilters 这个类来保存这些选项的状态。

```
class ContactListFilters {
    var prefix: String = ""
    var onlyWithPhoneNumber: Boolean = false
}
```

当用户输入 D 来查看姓或者名以 *D* 开头的联系人，prefix 的值会被更新。这里省略了必须要更改的代码（对于这本书的篇幅来说，一个带有完整 UI 的程序代码实在太多，所以我们只展示一个简化的示例）。

为了让展示联系人列表的逻辑代码和输入过滤条件的 UI 代码解耦，可以定义一个函数来创建一个判断式，用它来过滤联系人列表。判断式检查前缀，如果有需要也会检查电话号码是否存在。

代码清单 8.7 在 UI 代码中定义一个返回函数的函数

```
data class Person(
        val firstName: String,
        val lastName: String,
        val phoneNumber: String?
)

class ContactListFilters {
    var prefix: String = ""
    var onlyWithPhoneNumber: Boolean = false

    fun getPredicate(): (Person) -> Boolean {
```

声明一个返回函数的函数

```
        val startsWithPrefix = { p: Person ->
            p.firstName.startsWith(prefix) || p.lastName.startsWith(prefix)
        }
        if (!onlyWithPhoneNumber) {
            return startsWithPrefix
        }
        return { startsWithPrefix(it)
                    && it.phoneNumber != null }
    }
}
```

返回一个函数类型的变量

从这个函数返回一个 lambda

```
>>> val contacts = listOf(Person("Dmitry", "Jemerov", "123-4567"),
...                       Person("Svetlana", "Isakova", null))
>>> val contactListFilters = ContactListFilters()
>>> with (contactListFilters) {
>>>     prefix = "Dm"
>>>     onlyWithPhoneNumber = true
>>> }
>>> println(contacts.filter(
...     contactListFilters.getPredicate()))
[Person(firstName=Dmitry, lastName=Jemerov, phoneNumber=123-4567)]
```

将 getPredicate 返回的函数作为参数传递给“filter”函数

getPredicate 方法返回一个函数（类型）的值，这个值被传递给 filter 作为参数。Kotlin 的函数类型可以让这一切变得简单，就跟处理其他类型的值一样，比如字符串。

高阶函数是一个改进代码结构和减少重复代码的利器。我们来看 lambda 如何帮助我们从代码中抽取重复逻辑。

8.1.6 通过 lambda 去除重复代码

函数类型和 lambda 表达式一起组成了一个创建可重用代码的好工具。许多以前只能通过复杂笨重的结构来避免的重复代码，现在可以通过使用简洁的 lambda 表达式被消除。

我们来看一个分析网站访问的例子。SiteVisit 类用来保存每次访问的路径、持续时间和用户的操作系统。不同的操作系统使用枚举类型来表示。

代码清单 8.8 定义站点访问数据

```
data class SiteVisit(
    val path: String,
    val duration: Double,
    val os: OS
)

enum class OS { WINDOWS, LINUX, MAC, IOS, ANDROID }
```

```
val log = listOf(
    SiteVisit("/", 34.0, OS.WINDOWS),
    SiteVisit("/", 22.0, OS.MAC),
    SiteVisit("/login", 12.0, OS.WINDOWS),
    SiteVisit("/signup", 8.0, OS.IOS),
    SiteVisit("/", 16.3, OS.ANDROID)
)
```

想象一下如果你需要显示来自 Windows 机器的平均访问时间，可以用 average 函数来完成这个任务。

代码清单 8.9 使用硬编码的过滤器分析站点访问数据

```
val averageWindowsDuration = log
    .filter { it.os == OS.WINDOWS }
    .map(SiteVisit::duration)
    .average()

>>> println(averageWindowsDuration)
23.0
```

现在假设你要计算来自 Mac 用户的相同数据，为了避免重复，可以将平台类型抽象为一个参数。

代码清单 8.10 用一个普通方法去除重复代码

```
fun List<SiteVisit>.averageDurationFor(os: OS) =
        filter { it.os == os }.map(SiteVisit::duration).average()

>>> println(log.averageDurationFor(OS.WINDOWS))
23.0
>>> println(log.averageDurationFor(OS.MAC))
22.0
```

将重复代码抽取到函数中

注意将这个函数作为扩展函数增强了可读性。如果它只在局部的上下文中有用，你甚至可以将这个函数声明为局部扩展函数。

但这还远远不够。想象一下，如果你对来自移动平台（目前你识别出来的只有两种：iOS 和 Android）的访问的平均时间非常有兴趣。

代码清单 8.11 用一个复杂的硬编码函数分析站点访问数据

```
val averageMobileDuration = log
    .filter { it.os in setOf(OS.IOS, OS.ANDROID) }
    .map(SiteVisit::duration)
    .average()

>>> println(averageMobileDuration)
12.15
```

现在已经无法再用一个简单的参数表示不同的平台了。你可能还需要使用更加复杂的条件查询日志，比如“来自 iOS 平台对注册页面的访问的平均时间是多少？”Lambda 可以帮上忙。可以用函数类型将需要的条件抽取到一个参数中。

代码清单 8.12 用一个高阶函数去除重复代码

```
fun List<SiteVisit>.averageDurationFor(predicate: (SiteVisit) -> Boolean) =
        filter(predicate).map(SiteVisit::duration).average()

>>> println(log.averageDurationFor {
...     it.os in setOf(OS.ANDROID, OS.IOS) })
12.15
>>> println(log.averageDurationFor {
...     it.os == OS.IOS && it.path == "/signup" })
8.0
```

函数类型可以帮助去除重复代码。如果你禁不住复制粘贴了一段代码，那么很可能这段重复代码是可以避免的。使用 lambda，不仅可以抽取重复的数据，也可以抽取重复的行为。

> **注意** 一些广为人知的设计模式可以用函数类型和 lambda 表达式进行简化。比如策略模式。没有 lambda 表达式的情况下，你需要声明一个接口，并为每一种可能的策略提供实现类。使用函数类型，可以用一个通用的函数类型来描述策略，然后传递不同的 lambda 表达式作为不同的策略。

我们已经讨论了如何创建高阶函数。接下来，我们来看看它的性能。相比于编写老式的循环和条件语句，任何情况下都使用高阶函数难道不会使代码运行得更慢吗？下一节将讨论为什么并不总是这样，以及 `inline` 关键字如何提供帮助。

8.2 内联函数：消除lambda带来的运行时开销

你可能已经注意到，Kotlin 中传递 lambda 作为函数参数的简明语法与普通的表达式（例如 `if` 和 `for`）语法很相似。在第 5 章讨论 `with` 和 `apply` 这两个函数的时候我们已经见过。但是它的性能如何呢？要是定义了类似 Java 语句的函数但运行起来却慢得多，这难道不是出人意料的不爽吗？

在第 5 章中，我们解释了 lambda 表达式会被正常地编译成匿名类。这表示每调用一次 lambda 表达式，一个额外的类就会被创建。并且如果 lambda 捕捉了某个变量，那么每次调用的时候都会创建一个新的对象。这会带来运行时的额外开销，导致使用 lambda 比使用一个直接执行相同代码的函数效率更低。

有没有可能让编译器生成跟 Java 语句同样高效的代码，但还是能把重复的逻辑

抽取到库函数中呢？是的，Kotlin 的编译器能够做到。如果使用 inline 修饰符标记一个函数，在函数被使用的时候编译器并不会生成函数调用的代码，而是使用函数实现的真实代码替换每一次的函数调用。我们通过一个具体的例子来看看这到底是怎么运作的。

8.2.1 内联函数如何运作

当一个函数被声明为 inline 时，它的函数体是内联的——换句话说，函数体会被直接替换到函数被调用的地方，而不是被正常调用。来看一个例子以便理解生成的最终代码。

代码清单 8.13 中的函数用于确保一个共享资源不会并发地被多个线程访问。函数锁住一个 Lock 对象，执行代码块，然后释放锁。

代码清单 8.13 定义一个内联函数

```
inline fun <T> synchronized(lock: Lock, action: () -> T): T {
    lock.lock()
    try {
        return action()
    }
    finally {
        lock.unlock()
    }
}
val l = Lock()
synchronized(l) {
    // ...
}
```

调用这个函数的语法跟 Java 中使用 synchronized 语句完全一样。区别在于 Java 的 synchronized 语句可以用于任何对象，而这个函数则要求传入一个 Lock 实例。这里展示的定义只是一个示例，Kotlin 标准库中定义了一个可以接收任何对象作为参数的 synchronized 函数的版本。

但在同步操作时使用显式的对象锁能提高代码的可读性和可维护性。在 8.2.5 节中，我们会介绍 Kotlin 标准库中的 withLock 函数，在需要加锁执行给定的操作时，它将成为你的最佳选择。

因为已经将 synchronized 函数声明为 inline，所以每次调用它所生成的代码跟 Java 的 synchronized 语句是一样的。看看下面这个使用 synchronized() 的例子：

```
fun foo(l: Lock) {
    println("Before sync")
    synchronized(l) {
        println("Action")
    }
    println("After sync")
}
```

图 8.3 展示的是作用相同的代码，将会被编译成同样的字节码：

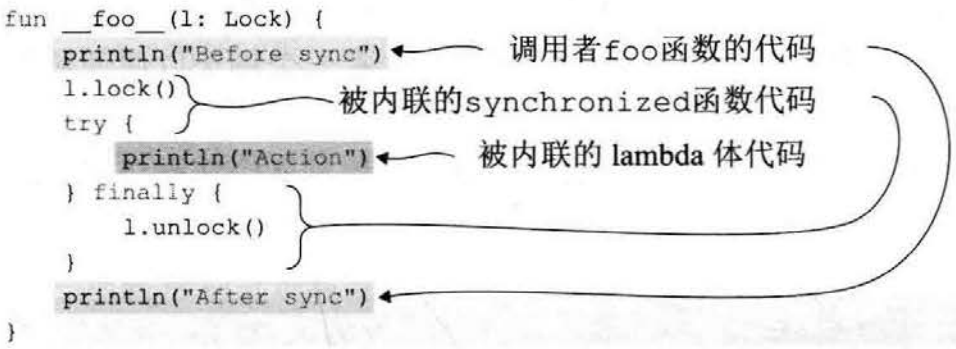


图 8.3 编译后的 foo 函数

注意 lambda 表达式和 synchronized 函数的实现都被内联了。由 lambda 生成的字节码成为了函数调用者定义的一部分，而不是被包含在一个实现了函数接口的匿名类中。

注意，在调用内联函数的时候也可以传递函数类型的变量作为参数：

```
class LockOwner(val lock: Lock) {
    fun runUnderLock(body: () -> Unit) {
        synchronized(lock, body)
    }
}
```

传递一个函数类型的变量作为参数，而不是一个 lambda

在这种情况下，lambda 的代码在内联函数被调用点是不可用的，因此并不会被内联。只有 synchronized 的函数体被内联了，lambda 才会被正常调用。runUnderLock 函数会被编译成类似于以下函数的字节码：

```
class LockOwner(val lock: Lock) {
    fun __runUnderLock__(body: () -> Unit) {
        lock.lock()
        try {
            body()
        }
        finally {
            lock.unlock()
        }
    }
}
```

这个函数类似于真正的 runUnderLock 被编译成的字节码

body 没有被内联，因为在调用的地方还没有 lambda

如果在两个不同的位置使用同一个内联函数，但是用的是不同的 lambda，那么内联函数会在每一个被调用的位置被分别内联。内联函数的代码会被拷贝到使用它的两个不同位置，并把不同的 lambda 替换到其中。

8.2.2 内联函数的限制

鉴于内联的运作方式，不是所有使用 lambda 的函数都可以被内联。当函数被内联的时候，作为参数的 lambda 表达式的函数体会被直接替换到最终生成的代码中。这将限制函数体中的对应（lambda）参数的使用。如果（lambda）参数被调用，这样的代码能被容易地内联。但如果（lambda）参数在某个地方被保存起来，以便后面可以继续使用，lambda 表达式的代码将不能被内联，因为必须要有一个包含这些代码的对象存在。

一般来说，参数如果被直接调用或者作为参数传递给另外一个 `inline` 函数，它是可以被内联的。否则，编译器会禁止参数被内联并给出错误信息“Illegal usage of inline-parameter”（非法使用内联参数）。

例如，许多作用于序列的函数会返回一些类的实例，这些类代表对应的序列操作并接收 lambda 作为构造方法的参数。以下是 `Sequence.map` 函数的定义：

```
fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}
```

`map` 函数没有直接调用作为 `transform` 参数传递进来的函数。而是将这个函数传递给一个类的构造方法，构造方法将它保存在一个属性中。为了支持这一点，作为 `transform` 参数传递的 lambda 需要被编译成标准的非内联的表示法，即一个实现了函数接口的匿名类。

如果一个函数期望两个或更多 lambda 参数，可以选择只内联其中一些参数。这是有道理的，因为一个 lambda 可能会包含很多代码或者以不允许内联的方式使用。接收这样的非内联 lambda 的参数，可以用 `noinline` 修饰符来标记它：

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
  // ...
}
```

注意，编译器完全支持内联跨模块的函数或者第三方库定义的函数。也可以在 Java 中调用绝大部分内联函数，但这些调用并不会被内联，而是被编译成普通的函数调用。

在本书的 9.2.4 节，你将会看到另外一个使用 `noinline` 的情况（然而，会受

到 Java 互操作性的一些约束）。

8.2.3 内联集合操作

我们来仔细看一看 Kotlin 标准库中操作集合的函数的性能。大部分标准库中的集合函数都带有 lambda 参数。相比于使用标准库函数，直接实现这些操作不是更高效吗？

例如，让我们来比较以下两个代码清单中用来过滤一个人员列表的方式：

代码清单 8.14 使用 lambda 过滤一个集合

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

>>> println(people.filter { it.age < 30 })
[Person(name=Alice, age=29)]
```

前面的代码不用 lambda 表达式也可以实现，代码如下：

代码清单 8.15 手动过滤一个集合

```
>>> val result = mutableListOf<Person>()
>>> for (person in people) {
>>>       if (person.age < 30) result.add(person)
>>> }
>>> println(result)
[Person(name=Alice, age=29)]
```

在 Kotlin 中，filter 函数被声明为内联函数。这意味着 filter 函数，以及传递给它的 lambda 的字节码会被一起内联到 filter 被调用的地方。最终，第一种实现所产生的字节码和第二种实现所产生的字节码大致是一样的。你可以很安全地使用符合语言习惯的集合操作，Kotlin 对内联函数的支持让你不必担心性能的问题。

想象一下现在你连续调用 filter 和 map 两个操作。

```
>>> println(people.filter { it.age > 30 }
...                 .map(Person::name))
[Bob]
```

这个例子使用了一个 lambda 表达式和一个成员引用。再一次，filter 和 map 函数都被声明为 inline 函数，所以它们的函数体会被内联，因此不会产生额外的类或者对象。但是上面的代码却创建了一个中间集合来保存列表过滤的结果，由 filter 函数生成的代码会向这个集合中添加元素，而由 map 函数生成的代码会读取这个集合。

如果有大量元素需要处理，中间集合的运行开销将成为不可忽视的问题，这时可以在调用链后加上一个 asSequence 调用，用序列来替代集合。但正如你在前一节中看到的，用来处理序列的 lambda 没有被内联。每一个中间序列被表示成把 lambda 保存在其字段中的对象，而末端操作会导致由每一个中间序列调用组成的调用链被执行。因此，即便序列上的操作是惰性的，你不应该总是试图在集合操作的调用链后加上 asSequence。这只在处理大量数据的集合时有用，小的集合可以用普通的集合操作处理。

8.2.4 决定何时将函数声明成内联

现在你已经知道了 inline 关键字带来的好处，可能已经想要开始在代码中使用 inline，试图让你的代码运行得更快。事实证明，这并不是一个好主意。使用 inline 关键字只能提高带有 lambda 参数的函数的性能，其他的情况需要额外的度量和研究。

对于普通的函数调用，JVM 已经提供了强大的内联支持。它会分析代码的执行，并在任何通过内联能够带来好处的时候将函数调用内联。这是在将字节码转换成机器代码时自动完成的。在字节码中，每一个函数的实现只会出现一次，并不需要跟 Kotlin 的内联函数一样，每个调用的地方都拷贝一次。再说，如果函数被直接调用，调用栈会更加清晰。

另一方面，将带有 lambda 参数的函数内联能带来好处。首先，通过内联避免的运行时开销更明显了。不仅节约了函数调用的开销，而且节约了为 lambda 创建匿名类，以及创建 lambda 实例对象的开销。其次，JVM 目前并没有聪明到总是能将函数调用内联。最后，内联使得我们可以使用一些不可能被普通 lambda 使用的特性，比如非局部返回，我们将在本章的后面部分讨论它。

但是在使用 inline 关键字的时候，你还是应该注意代码的长度。如果你要内联的函数很大，将它的字节码拷贝到每一个调用点将会极大地增加字节码的长度。在这种情况下，你应该将那些与 lambda 参数无关的代码抽取到一个独立的非内联函数中。你可以自己去验证一下，在 Kotlin 标准库中的内联函数总是很小的。

接下来，让我们看看高阶函数如何帮助我们改进代码。

8.2.5 使用内联 lambda 管理资源

Lambda 可以去除重复代码的一个常见模式是资源管理：先获取一个资源，完成一个操作，然后释放资源。这里的资源可以表示很多不同的东西：一个文件、一个锁、一个数据库事务等。实现这个模式的标准做法是使用 try/finally 语句。资源在 try 代码块之前被获取，在 finally 代码块中被释放。

在本节的前面部分你看到过一个例子，将 try/finally 的逻辑封装在一个函数中，然后将使用资源的代码作为 lambda 传递给这个方法。那个例子展示了 synchronized 函数，它跟 Java 中的 synchronized 语句语法一样：将一个锁对象作为参数。Kotlin 标准库定义了另一个叫作 withLock 的函数，它提供了实现同样功能的更符合语言习惯的 API: 它是 Lock 接口的扩展函数。下面来看如何使用它：

```
val l: Lock = ...
l.withLock {
    // access the resource protected by this lock
}
```

在加锁的情况下执行指定的操作

这是 Kotlin 库中 withLock 函数的定义：

```
fun <T> Lock.withLock(action: () -> T): T {
    lock()
    try {
        return action()
    } finally {
        unlock()
    }
}
```

需要加锁的代码被抽取到一个独立的方法中

文件是另一种可以使用这种模式的常见资源类型。Java 7 甚至为这种模式引入了特殊的语法：try-with-resource 语句。下面的代码清单展示了一个使用这个语句来读取文件第一行的 Java 方法。

代码清单 8.16 在 Java 中使用 try-with-resource 语句

```
/* Java */
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

Kotlin 中并没有等价的语法，因为通过使用一个带有函数类型参数的函数（接收 lambda 参数）可以无缝地完成相同的事情。这个 Kotlin 标准库中的函数叫作 use。现在使用 use 函数将代码清单 8.16 中的代码重写为 Kotlin 代码。

代码清单 8.17 使用 use 函数作资源管理

```
fun readFirstLineFromFile(path: String): String {
```

```
    BufferedReader(FileReader(path)).use { br ->
        return br.readLine()
    }
}
```

构建 BufferedReader，调用“use”函数，传递一个 lambda 执行文件操作

从函数中返回文件的一行

use 函数是一个扩展函数，被用来操作可关闭的资源，它接收一个 lambda 作为参数。这个方法调用 lambda 并且确保资源被关闭，无论 lambda 正常执行还是抛出了异常。当然，use 函数是内联函数，所以使用它并不会引发任何性能开销。

注意，在 lambda 的函数体中，使用了非局部 return 从 readFirstLineFromFile 函数中返回了一个值。我们来详细地讨论 lambda 中 return 表达式的细节。

8.3 高阶函数中的控制流

当你开始使用 lambda 去替换像循环这样的命令式代码结构时，很快便会遇到 return 表达式的问题。把一个 return 语句放在循环的中间是很简单的事情。但是如果将循环转换成一个类似 filter 的函数呢？在这种情况下 return 会如何工作？我们来看一些例子。

8.3.1 lambda 中的返回语句：从一个封闭的函数返回

来比较两种不同的遍历集合的方法。在下面的代码清单中，很明显如果一个人的名字是 Alice，就应该从函数 lookForAlice 返回。

代码清单 8.18 在一个普通循环中使用 return

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun lookForAlice(people: List<Person>) {
    for (person in people) {
        if (person.name == "Alice") {
            println("Found!")
            return
        }
    }
    println("Alice is not found")
}
```

如果“people”中没有 Alice，这一行就会被打印出来

```
>>> lookForAlice(people)
Found!
```

使用 forEach 迭代重写这段代码安全吗？return 语句还会是一样的表现吗？

是的，正如以下的代码所展示的，使用 forEach 是安全的。

代码清单 8.19 在传递给 forEach 的 lambda 中使用 return

```
fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name == "Alice") {
            println("Found!")
            return                                    <-- 和代码清单8.18中一样返回
        }
    }
    println("Alice is not found")
}
```

如果你在 lambda 中使用 return 关键字，它会从调用 *lambda* 的函数中返回，并不只是从 lambda 中返回。这样的 return 语句叫作非局部返回，因为它从一个比包含 return 的代码块更大的代码块中返回了。

为了理解这条规则背后的逻辑，想想 Java 函数中在 for 循环或者 synchronized 代码块中使用 return 关键字。显然会从函数中返回，而不是从循环或者代码块中返回。当使用以 lambda 作为参数的函数的时候 Kotlin 保留了同样的行为。

需要注意的是，*只有在以 lambda 作为参数的函数是内联函数的时候*才能从更外层的函数返回。在代码清单 8.19 中，forEach 的函数体和 lambda 的函数体一起被内联了，所以在编译的时候能很容易做到从包含它的函数中返回。在一个非内联函数的 lambda 中使用 return 表达式是不允许的。一个非内联函数可以把传给它的 lambda 保存在变量中，以便在函数返回以后可以继续使用，这个时候 lambda 想要去影响函数的返回已经太晚了。

8.3.2 从 lambda 返回：使用标签返回

也可以在 lambda 表达式中使用局部返回。lambda 中的局部返回跟 for 循环中的 break 表达式相似。它会终止 lambda 的执行，并接着从调用 lambda 的代码处执行。要区分局部返回和非局部返回，要用到标签。想从一个 lambda 表达式处返回你可以标记它，然后在 return 关键字后面引用这个标签。

代码清单 8.20 用一个标签实现局部返回

```
fun lookForAlice(people: List<Person>) {
```

```
lambda表达式加上标签 ──▷ people.forEach label@{
        if (it.name == "Alice") return@label
    }
    println("Alice might be somewhere")
}
```

return@label 引用了这个标签

这一行总是会被打印出来

```
>>> lookForAlice(people)
Alice might be somewhere
```

要标记一个 lambda 表达式，在 lambda 的花括号之前放一个标签名（可以是任何标识符），接着放一个 @ 符号。要从一个 lambda 返回，在 return 关键字后放一个 @ 符号，接着放标签名，如图 8.4 所示。

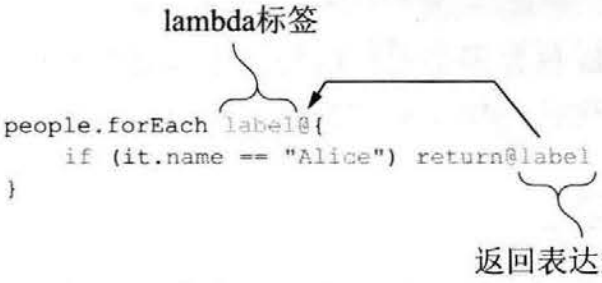


图 8.4 用“@”符号标记一个标签从一个 lambda 返回

另外一种选择是，使用 lambda 作为参数的函数的函数名可以作为标签。

代码清单 8.21 用函数名作为 return 标签

```
fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name == "Alice") return@forEach
    }
    println("Alice might be somewhere")
}
```

return@forEach 从 lambda 表达式返回

如果你显式地指定了 lambda 表达式的标签，再使用函数名作为标签没有任何效果。一个 lambda 表达式的标签数量不能多于一个。

带标签的“this”表达式

同样的规则也适用于 this 表达式的标签。在第 5 章我们讨论了带接收者的 lambda——包含一个隐式上下文对象的 lambda（第 11 章会解释如何写一个以带接收者的 lambda 作为参数的函数）可以通过一个 this 引用去访问。如果你给带接收者的 lambda 指定标签，就可以通过对应的带有标签的 this 表达式访问它的隐式接收者：

```
>>> println(StringBuilder().apply sb@{
...     listOf(1, 2, 3).apply {
...         this@sb.append(this.toString())
...     }
... })
[1, 2, 3]
```

这个 lambda 的隐式接收者可以通过 this@sb 访问

“this”指向作用域内最近的隐式接收者

所有隐式接收者都可以被访问，外层的接收者通过显式的标签访问

和 return 表达式中使用标签一样，可以显式地指定 lambda 表达式的标签，也可以使用函数名作为标签。

局部返回的语法相当冗长，如果一个 lambda 包含多个返回语句会变得更加笨重。解决方案是，可以使用另一种可选的语法来传递代码块：匿名函数。

8.3.3 匿名函数：默认使用局部返回

匿名函数是一种不同的用于编写传递给函数的代码块的方式。先来看一个示例：

代码清单 8.22 在匿名函数中使用 return

```
fun lookForAlice(people: List<Person>) {
    people.forEach(fun (person) {
        if (person.name == "Alice") return
        println("${person.name} is not Alice")
    })
}

>>> lookForAlice(people)
Bob is not Alice
```

使用匿名函数取代 lambda 表达式

“return”指向最近的函数：一个匿名函数

匿名函数看起来跟普通函数很相似，除了它的名字和参数类型被省略了外。这里有另外一个例子：

代码清单 8.23 在 filter 中使用匿名函数

```
people.filter(fun (person): Boolean {
    return person.age < 30
})
```

匿名函数和普通函数有相同的指定返回值类型的规则。正如代码清单 8.23 中的代码一样，代码块体匿名函数需要显式地指定返回类型，如果使用表达式函数体，就可以省略返回类型。

代码清单 8.24 使用表达式体匿名函数

```
people.filter(fun (person) = person.age < 30)
```

在匿名函数中，不带标签的 return 表达式会从匿名函数返回，而不是从包含匿名函数的函数返回。这条规则很简单：return 从最近的使用 fun 关键字声明的函数返回。lambda 表达式没有使用 fun 关键字，所以 lambda 中的 return 从最外层的函数返回。匿名函数使用了 fun，因此，在前一个例子中匿名函数是最近的符合规则的函数。所以，return 表达式从匿名函数返回，而不是从最外层的函数返回。图 8.5 阐述了它们之间的区别：

```
fun lookForAlice(people: List<Person>) {
    people.forEach(fun(person) {
        if (person.name == "Alice") return

    })
}

fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name == "Alice") return
    }
}
```

图 8.5 return 表达式从使用 fun 关键字声明的函数返回

注意，尽管匿名函数看起来跟普通函数很相似，但它其实是 lambda 表达式的另一种语法形式而已。关于 lambda 表达式如何实现，以及在内联函数中如何被内联的讨论同样适用于匿名函数。

8.4 小结

- 函数类型可以让你声明一个持有函数引用的变量、参数或者函数返回值。
- 高阶函数以其他函数作为参数或者返回值。可以用函数类型作为函数参数或者返回值的类型来创建这样的函数。
- 内联函数被编译以后，它的字节码连同传递给它的 lambda 的字节码会被插入到调用函数的代码中，这使得函数调用相比于直接编写相同的代码，不会产生额外的运行时开销。
- 高阶函数促进了一个组件内的不同部分的代码重用，也可以让你构建功能强大的通用库。

- 内联函数可以让你使用*非局部返回*——在 lambda 中从包含函数返回的返回表达式。
- 匿名函数给 lambda 表达式提供了另一种可选的语法，用不同的规则来解析 `return` 表达式。可以在需要编写有多个退出点的代码块的时候使用它们。

9 泛型

本章内容包括

- 声明泛型函数和类
- 类型擦除和实化类型参数
- 声明点变型和使用点变型

在本书中你已经见过一些使用泛型代码的例子。Kotlin 中声明和使用泛型类及泛型函数的基本概念和 Java 类似，所以之前的例子即使不用详细解释应该也很清晰。这一章我们会回顾其中的一些例子，探索它们的细节。

然后我们继续深入泛型的主题，探索 Kotlin 引入的新概念，比如实化类型参数和声明点变型。这些概念对你来说可能很新奇，但不要担心，本章会透彻地讲解这些概念。

*实化类型参数*允许你在运行时的内联函数调用中引用作为类型实参的具体类型（对普通的类和函数来说，这样行不通，因为类型实参在运行时会被擦除）。

*声明点变型*可以说明一个带类型参数的泛型类型，是否是另一个泛型类型的子类型或者超类型，它们的基础类型相同但类型参数不同。例如，它能调节是否可以把 `List<Int>` 类型的参数传给期望 `List<Any>` 的函数。*使用点变型*在具体使用一个泛型类型时做同样的事，达到和 Java 通配符一样的效果。

9.1 泛型类型参数

泛型允许你定义带类型形参的类型。当这种类型的实例被创建出来的时候，类型形参被替换成称为类型实参的具体类型。例如，如果有一个 List 类型的变量，弄清楚这个列表中可以存储哪种事物是很有意义的。类型形参可以准确清晰地进行描述，就像这样“这个变量保存了字符串列表”，而不是“这个变量保存了一个列表”。Kotlin 说明“字符串列表”的语法和 Java 看起来一样：List<String>。还可以给一个类声明多个类型形参。例如，Map 类就有键类型和值类型这两个类型形参：class Map<K, V>。我们可以用具体的类型实参来实例化它：Map<String, Person>。目前，所有概念都和 Java 没什么不一样。

和一般类型一样，Kotlin 编译器也常常能推导出类型实参：

```
val authors = listOf("Dmitry", "Svetlana")
```

因为传给 listOf 函数的两个值都是字符串，编译器推导出你正在创建一个 List<String>。另一方面，如果你想创建一个空的列表，这样就没有任何可以推导出类型实参的线索，你就得显式地指定它（类型形参）。就创建列表来说，既可以选择在变量声明中说明泛型的类型，也可以选择在创建列表的函数中说明类型实参。参看下面的例子：

```
val readers: MutableList<String> = mutableListOf()

val readers = mutableListOf<String>()
```

两种声明是等价的。注意创建集合的函数在 6.3 节中介绍过。

注意 和 Java 不同，Kotlin 始终要求类型实参要么被显式地说明，要么能被编译器推导出来。因为泛型是在 1.5 版本才引入到 Java 的，它必须保证和基于老版本 Java 编写的代码兼容，所以它允许使用没有类型参数的泛型类型——所谓的原生态类型。例如，在 Java 中，可以声明 List 类型的变量，而不需要说明它可以包含哪类事物。而 Kotlin 从一开始就有泛型，所以它不支持原生态类型，类型实参必须定义。

9.1.1 泛型函数和属性

如果要编写一个使用列表的函数，希望它可以在任何列表（通用的列表）上使用，而不是某个具体类型的元素的列表，需要编写一个泛型函数。泛型函数有它自己的类型形参。这些类型形参在每次函数调用时都必须替换成具体的类型实参。

大部分使用集合的库函数都是泛型的。来看看图 9.1 中的 slice 函数。这个函数返回一个只包含在指定下标区间内的元素。

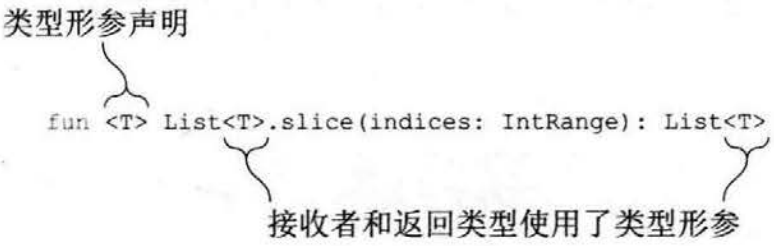


图 9.1 slice 泛型函数的类型形参为 T

接收者和返回类型用到了函数的类型形参 T，它们的类型都是 List<T>。当你在一个具体的列表上调用这个函数时，可以显式地指定类型实参。但是大部分情况下你不必这样做，因为编译器会推导出类型，如下所示。

代码清单 9.1 调用泛型函数

```
>>> val letters = ('a'..'z').toList()
>>> println(letters.slice<Char>(0..2))
[a, b, c]
>>> println(letters.slice(10..13))
[k, l, m, n]
```

显式地指定类型实参

编译器推导出这里的 T 是 Char

这两次调用的结果都是 List<Char>。编译器把函数返回类型 List<T> 中的 T 替换成了推导出来的类型 Char。

8.1 节中，你看过 filter 函数的声明，它接收了一个函数类型 (T) -> Boolean 的参数。我们看看如何把它用到前面例子中的变量 readers 和 authors 中。

代码清单 9.2 调用泛化的高阶函数

```
val authors = listOf("Dmitry", "Svetlana")
val readers = mutableListOf<String>(/* ... */)

fun <T> List<T>.filter(predicate: (T) -> Boolean): List<T>

>>> readers.filter { it !in authors }
```

这个例子中自动生成的 lambda 参数 it 的类型是 String。编译器必须把它推导出来：毕竟，在函数声明中 lambda 参数是泛型类型 T（即 (T) -> Boolean 函数的参数类型 T）。编译器推断 T 就是 String，因为它知道函数应该在 List<T> 上调用，而它的接收者 readers 的真实类型是 List<String>。

可以给类或接口的方法、顶层函数，以及扩展函数声明类型参数。在前面的例子中，类型参数用在了接收者和（lambda）参数的类型上，就像代码清单 9.1 和 9.2

那样：类型参数 T 是接收者类型 List<T> 的一部分，也用在了参数的函数类型 (T) -> Boolean 上。

还可以用同样的语法声明泛型的扩展属性。例如下面这个返回列表倒数第二个元素的扩展属性：

```
val <T> List<T>.penultimate: T
    get() = this[size - 2]

>>> println(listOf(1, 2, 3, 4).penultimate)
3
```

这个泛型扩展属性能在任何种类元素的列表上调用

在这次调用中，类型参数 T 被推导成 Int

不能声明泛型非扩展属性

普通（即非扩展）属性不能拥有类型参数，不能在一个类的属性中存储多个不同类型的值，因此声明泛型非扩展函数没有任何意义。你可以尝试一下，编译器会报告错误：

```
>>> val <T> x: T = TODO()
ERROR: type parameter of a property must be used in its receiver type
```

接下来我们概括一下如何声明泛型类。

9.1.2 声明泛型类

和 Java 一样，Kotlin 通过在类名称后加上一对尖括号，并把类型参数放在尖括号内来声明泛型类及泛型接口。一旦声明之后，就可以在类的主体内像其他类型一样使用类型参数。我们来看看标准 Java 接口 List 如何使用 Kotlin 来声明。我们省去了大部分的方法定义，让例子变得简单：

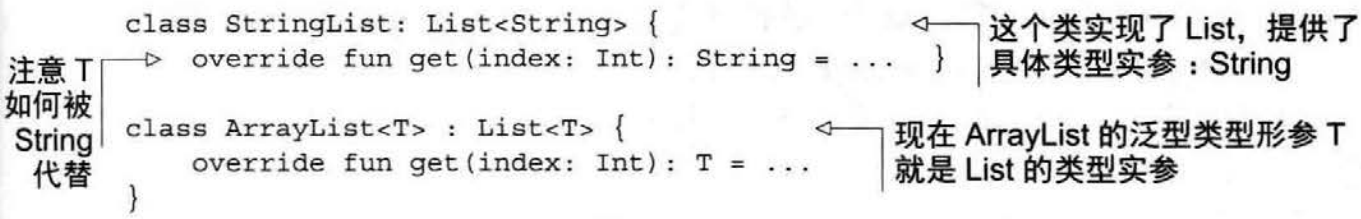
```
interface List<T> {
    operator fun get(index: Int): T
    // ...
}
```

List 接口定义了类型参数 T

在接口或类的内部，T 可以当作普通类型使用

本章稍后介绍变型的主题时，会改进这个例子并看到 Kotlin 标准库中 List 是如何声明的。

如果你的类继承了泛型类（或者实现了泛型接口），你就得为基础类型的泛型形参提供一个类型实参。它可以是具体类型或者另一个类型形参：



StringList 类被声明成只能包含 String 元素，所以它使用 String 作为基础类型的类型实参。子类中的任何函数都要用这个正确的类型换掉 T，所以在 StringList 中你会得到函数签名 get(Int): String，而不是 fun get(Int): T。

而类 ArrayList 定义了它自己的类型参数 T 并把它指定为父类的类型实参。注意 ArrayList<T> 中的 T 和 List<T> 中的 T 不一样，它是全新的类型形参，不必保留一样的名称。

一个类甚至可以把它自己作为类型实参引用。实现 Comparable 接口的类就是这种模式的经典例子。任何可以比较的元素都必须定义它如何与同样类型的对象比较：

```
interface Comparable<T> {
    fun compareTo(other: T): Int
}

class String : Comparable<String> {
    override fun compareTo(other: String): Int = /* ... */
}
```

String 类实现了 Comparable 泛型接口，提供类型 String 给类型实参 T。

迄今为止，泛型和 Java 中的看起来差不多。本章后面的 9.2 和 9.3 节我们会谈到两者之间的区别。现在我们先讨论另外一个和 Java 类似的概念：它允许你写出有用的处理可比较条目的函数。

9.1.3 类型参数约束

*类型参数约束*可以限制作为（泛型）类和（泛型）函数的类型实参的类型。以计算列表元素之和的函数为例。它可以用在 List<Int> 和 List<Double> 上，但不可以用在 List<String> 这样的列表上。可以定义一个类型参数约束，说明 sum 的类型形参必须是数字，来表达这个限制。

如果你把一个类型指定为泛型类型形参的上界约束，在泛型类型具体的初始化中，其对应的类型实参就必须是这个具体类型或者它的子类型（现在，你可以认为子类型和子类是同义词。9.3.2 节会着重介绍它们的区别）。

你是这样定义约束的，把冒号放在类型参数名称之后，作为类型形参上界的类

型紧随其后，如图 9.2 所示。在 Java 中，用的是关键字 extends 来表达一样的概念：<T extends Number> T sum(List<T> list)。

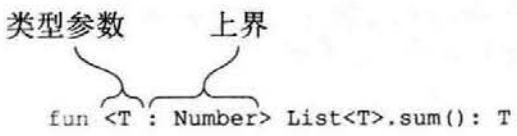


图 9.2 通过在类型参数后指定上界来定义约束

这次函数调用是允许的，因为具体类型实参（下面这个例子中是 Int）继承了 Number：

```
>>> println(listOf(1, 2, 3).sum())
6
```

一旦指定了类型形参 T 的上界，你就可以把类型 T 的值当作它的上界（类型）的值使用。例如，可以调用定义在上界类中的方法：

```
fun <T : Number> oneHalf(value: T): Double {
    return value.toDouble() / 2.0
}
```

指定 Number 为类型形参的上界

调用 Number 类中的方法

```
>>> println(oneHalf(3))
1.5
```

现在让我们编写一个找出两个条目中最大值的泛型函数。因为只有在可以相互比较的条目之中才能找出最大值，需要在函数签名中说明这一点。做法如下。

代码清单 9.3 声明带类型参数约束的函数

```
fun <T: Comparable<T>> max(first: T, second: T): T {
    return if (first > second) first else second
}
```

这个函数的实参必须是可比较的元素

```
>>> println(max("kotlin", "java"))
kotlin
```

字符串按字母表顺序比较

当你试图对不能比较的条目调用 max 方法时，代码不会编译：

```
>>> println(max("kotlin", 42))
ERROR: Type parameter bound for T is not satisfied:
 inferred type Any is not a subtype of Comparable<Any>
```

T 的上界是泛型类型 Comparable<T>。前面已经看到了，String 类继承了 Comparable<String>，这样使得 String 变成了 max 函数的有效类型实参。

记住，first > second 的简写形式会根据 Kotlin 的运算符约定被编译成 first.compareTo(second) > 0。这种比较之所以可行，是因为 first 的类型 T 继承自 Comparable<T>，这样你就可以比较 first 和另外一个类型 T 的元素。

极少数情况下，需要在一个类型参数上指定多个约束，这时你需要使用稍微不同的语法。例如下面这个代码清单用泛型的方式保证给定的 CharSequence 以句号结尾。标准 StringBuilder 类和 java.nio.CharBuffer 类都适用。

代码清单 9.4 为一个类型参数指定多个约束

```
fun <T> ensureTrailingPeriod(seq: T)
        where T : CharSequence, T : Appendable {
    if (!seq.endsWith('.')) {
        seq.append('.')
    }
}
```

类型参数约束的列表

调用为 CharSequence 接口定义的扩展函数

调用 Appendable 接口的方法

```
>>> val helloWorld = StringBuilder("Hello World")
>>> ensureTrailingPeriod(helloWorld)
>>> println(helloWorld)
Hello World.
```

这种情况下，可以说明作为类型实参的类型必须实现 CharSequence 和 Appendable 两个接口。这意味着该类型的值可以使用访问数据（endsWith）和修改数据（append）两种操作。

下面，我们将讨论另外一种常见的使用类型参数约束的情况：当需要声明非空类型形参的时候。

9.1.4 让类型形参非空

如果你声明的是泛型类或者泛型函数，任何类型实参，包括那些可空的类型实参，都可以替换它的类型形参。事实上，没有指定上界的类型形参将会使用 Any? 这个默认的上界。看看下面这个例子：

```
class Processor<T> {
    fun process(value: T) {
        value?.hashCode()
    }
}
```

“value”是可空的，所以要用安全调用

process 函数中，参数 value 是可空的，尽管 T 并没有使用问号标记。下面这种情况是因为 Processor 类具体初始化时 T 能使用可空类型：

```
val nullableStringProcessor = Processor<String?>()
nullableStringProcessor.process(null)
```

可空类型 String? 被用来替换 T

使用“null”作为“value”实参的代码可以编译

如果你想保证替换类型形参的始终是非空类型，可以通过指定一个约束来实现。如果你除了可空性之外没有任何限制，可以使用 Any 代替默认的 Any? 作为上界：

```
class Processor<T : Any> {
    fun process(value: T) {
        value.hashCode()
    }
}
```

指定非“空”上界

类型 T 的值现在是非“空”的

约束 <T : Any> 确保了类型 T 永远都是非空类型。编译器不会接收代码 Processor<String?>，因为类型实参 String? 不是 Any 的子类型（它是 Any? 的子类型，一种更普通的类型）：

```
>>> val nullableStringProcessor = Processor<String?>()
Error: Type argument is not within its bounds: should be subtype of 'Any'
```

注意，可以通过指定任意非空类型作为上界，来让类型参数非空，不光是类型 Any。

到目前为止，我们已经介绍了泛型的基础概念——那些和 Java 最接近的主题。如果你是一个 Java 开发者，现在我们讨论另一个熟悉的概念：运行时泛型的行为。

9.2 运行时的泛型：擦除和实化类型参数

你可能知道，JVM 上的泛型一般是通过类型擦除实现的，就是说泛型类实例的类型实参在运行时是不保留的。在本节中，我们将讨论类型擦除对 Kotlin 的实际影响，以及如何通过将函数声明为 inline 来解决其局限性。可以声明一个 inline 函数，使其类型实参不被擦除（或者，按照 Kotlin 术语，称作实化）。我们将详细讨论实化类型参数，并查看一些有用的例子。

9.2.1 运行时的泛型：类型检查和转换

和 Java 一样，Kotlin 的泛型在运行时也被擦除了。这意味着泛型类实例不会携带用于创建它的类型实参的信息。例如，如果你创建了一个 List<String> 并将一堆字符串放到其中，在运行时你只能看到它是一个 List，不能识别出列表本打算包含的是哪种类型的元素（当然，你可以获取一个元素然后检查它的类型，但即便检查通过了也不会有任何保证，因为其他的元素可能拥有不同的类型）。

想想执行下面的代码时这两个列表会发生什么（如图 9.3 所示）：

```
val list1: List<String> = listOf("a", "b")
val list2: List<Int> = listOf(1, 2, 3)
```

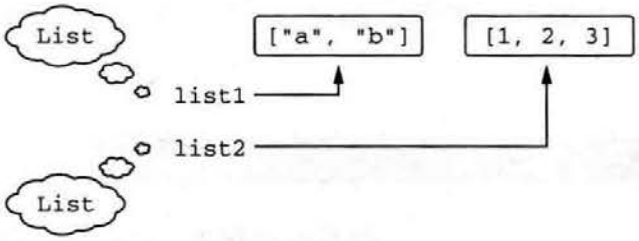


图 9.3 在运行时，你不会知道 list1 和 list2 是否声明成字符串或者整数列表。它们每个都只是 List

即使编译器看到的是两种完全不同类型的列表，在执行的时候它们看起来却完全一样。尽管如此，你通常可以确信 List<String> 只包含字符串，而 List<Int> 只包含整数。因为编译器知道类型实参，并确保每个列表中只存储正确类型的元素（可以通过类型转换或使用 Java 原生态类型访问列表，来欺骗编译器，但你需要特意这样做）。

接下来我们谈谈伴随着擦除类型信息的约束。因为类型实参没有被存储下来，你不能检查它们。例如，你不能判断一个列表是一个包含字符串的列表还是包含其他对象的列表。一般而言，在 is 检查中不可能使用类型实参中的类型。下面这样的代码不会编译：

```
>>> if (value is List<String>) { ... }
ERROR: Cannot check for instance of erased type
```

尽管在运行时可以完全断定这个值是一个 List，但你依然无法判断它是一个含有字符串的列表，还是含有人，或者含有其他什么：这些信息被擦除了。注意擦除泛型类型信息是有好处的：应用程序使用的内存总量较小，因为要保存在内存中的类型信息更少。

如前所述，Kotlin 不允许使用没有指定类型实参的泛型类型。那么你可能想知道如何检查一个值是否是列表，而不是 set 或者其他对象。可以使用特殊的星号投影语法来做这种检查：

```
if (value is List<*>) { ... }
```

实际上，泛型类型拥有的每个类型形参都需要一个 *。本章稍后我们会详细讨论星号投影（包括它被称为投影的原因）。现在，你可以认为它就是拥有未知类型实参的泛型类型（或者类比于 Java 的 List<?>）。前面的例子中，检查了 value 是否是 List，而并没有得到关于它的元素类型的任何信息。

注意，在 as 和 as? 转换中仍然可以使用一般的泛型类型。但是如果该类有正确的基础类型但类型实参是错误的，转换也不会失败，因为在运行时转换发生的时候类型实参是未知的。因此，这样的转换会导致编译器发出“unchecked cast”（未受检转换）的警告。这仅仅是一个警告，你仍然可以继续使用这个值，就当它拥有必要的类型，如下所示。

代码清单 9.5 对泛型类型做类型转换

```
fun printSum(c: Collection<*>) {
    val intList = c as? List<Int>
            ?: throw IllegalArgumentException("List is expected")
    println(intList.sum())
}

>>> printSum(listOf(1, 2, 3))
6
```

这里会有警告。Unchecked cast: List<*> to List<Int>

一切都符合预期

编译一切正常：编译器只是发出了一个警告，这意味着代码是合法的。如果在一个整型的列表或者 set 上调用 printSum 函数，一切都会如预期发生：第一种情况会打印出元素之和，而第二种情况则会抛出 IllegalArgumentException。但如果你传递了一个错误类型的值，运行时会得到一个 ClassCastException：

```
>>> printSum(setOf(1, 2, 3))
IllegalArgumentException: List is expected
>>> printSum(listOf("a", "b", "c"))
ClassCastException: String cannot be cast to Number
```

Set 不是列表，所以抛出了异常

类型转换成功，但后面抛出了另外的异常

我们来讨论一下在字符串列表上调用 printSum 函数时抛出的异常。你得到的并不是 IllegalArgumentException，因为你没有办法判断实参是不是一个 List<Int>。因此类型转换会成功，无论如何函数 sum 都会在这个列表上调用。在这个函数执行期间，异常抛出了。这是因为 sum 函数试着从列表中读取 Number 值然后把它们加在一起。把 String 当 Number 用的尝试会导致运行时的 ClassCastException。

注意，Kotlin 编译器是足够智能的，在编译期它已经知道相应的类型信息时，is 检查是允许的。

代码清单 9.6 对已知类型实参做类型转换

```
fun printSum(c: Collection<Int>) {
    if (c is List<Int>) {
        println(c.sum())
```

这次检查是合法的

```
    }
}

>>> printSum(listOf(1, 2, 3))
6
```

在代码清单 9.6 中，c 是否拥有类型 List<Int> 的检查是可行的，因为在编译期就确定了集合（不管它是列表还是其他类型的集合）包含的是整型数字。

通常，Kotlin 编译器会负责让你知道哪些检查是危险的（禁止 is 检查，以及发出 as 转换的警告），而哪些又是可行的。你要做的就是了解这些警告的含义并且了解哪些操作是安全的。

如前所述，Kotlin 有特殊的语法结构可以允许你在函数体中使用具体的类型实参，但只有 inline 函数可以。接下来我们就来看看这个特性。

9.2.2 声明带实化类型参数的函数

前面我们已经讨论过，Kotlin 泛型在运行时会被擦除，这意味着如果你有一个泛型类的实例，你无法弄清楚在这个实例创建时用的究竟是哪些类型实参。泛型函数的类型实参也是这样。在调用泛型函数的时候，在函数体中你不能决定调用它用的类型实参：

```
>>> fun <T> isA(value: Any) = value is T
Error: Cannot check for instance of erased type: T
```

通常情况下都是这样，只有一种例外可以避免这种限制：内联函数。内联函数的类型形参能够被实化，意味着你可以在运行时引用实际的类型实参。

在 8.2 节我们讨论过 inline 函数的细节，如果用 inline 关键字标记一个函数，编译器会把每一次函数调用都换成函数实际的代码实现。使用内联函数还可能提升性能，如果该函数使用了 lambda 实参：lambda 的代码也会内联，所以不会创建任何匿名类。这一节会展示 inline 函数大显身手的另一种场景：它们的类型参数可以被实化。

如果你把前面例子中的 isA 函数声明成 inline 并且用 reified 标记类型参数，你就能够用该函数检查 value 是不是 T 的实例。

代码清单 9.7 声明带实化类型参数的函数

```
inline fun <reified T> isA(value: Any) = value is T      <-- 现在代码可以编译了

>>> println(isA<String>("abc"))
true
>>> println(isA<String>(123))
false
```

接下来我们看看使用实化类型参数的一些稍微有意义的例子。一个实化类型参数能发挥作用的最简单的例子就是标准库函数 filterIsInstance。这个函数接收一个集合，选择其中那些指定类的实例，然后返回这些被选中的实例。下面展示了这个函数的用法。

代码清单 9.8 使用标准库函数 filterIsInstance

```
>>> val items = listOf("one", 2, "three")
>>> println(items.filterIsInstance<String>())
[one, three]
```

通过指定 <String> 作为函数的类型实参，你表明感兴趣的只是字符串。因此函数的返回类型是 List<String>。这种情况下，类型实参在运行时是已知的，函数 filterIsInstance 使用它来检查列表中的值是不是指定为该类型实参的类的实例。

下面是 Kotlin 标准库函数 filterIsInstance 声明的简化版本。

代码清单 9.9 filterIsInstance 的简化实现

```
inline fun <reified T>
        Iterable<*>.filterIsInstance(): List<T> {
    val destination = mutableListOf<T>()
    for (element in this) {
        if (element is T) {
            destination.add(element)
        }
    }
    return destination
}
```

“reified”声明了类型参数不会在运行时被擦除

可以检查元素是不是指定为类型实参的类的实例

为什么实化只对内联函数有效

这是什么原理？为什么在 inline 函数中允许这样写 element is T，而普通的类或函数却不行？

正如在 8.2 节中讨论的，编译器把实现内联函数的字节码插入每一次调用发生的地方。每次你调用带实化类型参数的函数时，编译器都知道这次特定调用中用作类型实参的确切类型。因此，编译器可以生成引用作为类型实参的具体类的字节码。实际上，对代码清单 9.8 中的 filterIsInstance<String> 调用来说，生成的代码和下面这段代码是等价的：

```
for (element in this) {
    if (element is String) {
        destination.add(element)
    }
}
```

引用了具体类

因为生成的字节码引用了具体类，而不是类型参数，它不会被运行时发生的类型参数擦除影响。

注意，带 reified 类型参数的 inline 函数不能在 Java 代码中调用。普通的内联函数可以像常规函数那样在 Java 中调用——它们可以被调用而不能被内联。带实化类型参数的函数需要额外的处理，来把类型实参的值替换到字节码中，所以它们必须永远是内联的。这样它们不可能用 Java 那样普通的方式调用。

一个内联函数可以有多个实化类型参数，也可以同时拥有非实化类型参数和实化类型参数。注意，filterIsInstance 函数虽然被标记成 inline，而它并不期望 lambda 作为实参。在 8.2.4 节中，我们提到把函数标记成内联只有在一种情况下有性能优势，即函数拥有函数类型的形参并且其对应的实参——lambda——和函数一起被内联的时候。但现在这个例子中，并不是因为性能的原因才把函数标记成 inline，这里这样做是为了能够使用实化类型参数。

为了保证良好的性能，你仍然需要跟踪了解标记为 inline 的函数的大小。如果函数变得庞大，最好把不依赖实化类型参数的代码抽取到单独的非内联函数中。

9.2.3 使用实化类型参数代替类引用

另一种实化类型参数的常见使用场景是为接收 java.lang.Class 类型参数的 API 构建适配器。一个这种 API 的例子是 JDK 中的 ServiceLoader，它接收一个代表接口或抽象类的 java.lang.Class，并返回实现了该接口（或继承了该抽象类）的类的实例。现在我们看看如何利用实化类型参数更容易地调用这些 API。

通过下面的调用来使用标准的 ServiceLoader Java API 加载一个服务：

```
val serviceImpl = ServiceLoader.load(Service::class.java)
```

::class.java 的语法展现了如何获取 java.lang.Class 对应的 Kotlin 类。这和 Java 中的 Service.class 是完全等同的。我们在 10.2 节中讨论反射的时候会更深入地涉及这个话题。

现在让我们用带实化类型参数的函数重写这个例子：

```
val serviceImpl = loadService<Service>()
```

代码是不是短了不少？要加载的服务类现在被指定成了 loadService 函数的类型实参。把一个类指定成类型实参要容易理解得多，因为它的代码比使用 ::class.java 语法更短。

下面，我们看看这个 loadService 函数是如何定义的：

```
inline fun <reified T> loadService() {
    return ServiceLoader.load(T::class.java)
}
```

类型参数标记成了“reified”

把 T::class 当成类型形参的类访问

这种用在普通类上的 ::class.java 语法也可以同样用在实化类型参数上。使用这种语法会产生对应到指定为类型参数的类的 java.lang.Class，你可以正常地使用它。

简化 Android 上的 startActivity 函数

如果你是 Android 开发者，你会觉得另一个例子更熟悉：显示 activity。也可以使用实化类型参数来代替传递作为 java.lang.Class 的 activity 类：

```
inline fun <reified T : Activity>
        Context.startActivity() {
    val intent = Intent(this, T::class.java)
    startActivity(intent)
}

startActivity<DetailActivity>()
```

类型参数标记成了“reified”

把 T::class 当成类型参数的类访问

调用方法显示 Activity

9.2.4 实化类型参数的限制

尽管实化类型参数是方便的工具，但它们也有一些限制。有一些是实化与生俱来的，而另外一些则是现有的实现决定的，而且可能在未来的 Kotlin 版本中放松这些限制。

具体来说，可以按下面的方式使用实化类型参数：

- 用在类型检查和类型转换中（is、!is、as、as?）
- 使用 Kotlin 反射 API，我们将在第 10 章讨论（::class）
- 获取相应的 java.lang.Class（::class.java）
- 作为调用其他函数的类型实参

不能做下面这些事情：

- 创建指定为类型参数的类的实例
- 调用类型参数类的伴生对象的方法
- 调用带实化类型参数函数的时候使用非实化类型形参作为类型实参
- 把类、属性或者非内联函数的类型参数标记成 reified

最后一条限制会带来有趣的后果：因为实化类型参数只能用在内联函数上，使用实化类型参数意味着函数和所有传给它的 lambda 都会被内联。如果内联函数使用 lambda 的方式导致 lambda 不能被内联，或者你不想 lambda 因为性能的关系被内联，可以使用 8.2.2 小节介绍的 noinline 修饰符把它们标记成非内联的。

现在我们已经讨论了作为语言特性的泛型的工作原理，接下来我们更仔细地查看每个 Kotlin 程序中都会出现的最常见的泛型类型：集合及它们的子类。我们会以它们为起点开始探索子类型化和变型的概念。

9.3 变型：泛型和子类型化

变型的概念描述了拥有相同基础类型和不同类型实参的（泛型）类型之间是如何关联的：例如，List<String> 和 List<Any> 之间如何关联。首先，我们会综合讨论为什么这种关系如此重要，然后我们将看到 Kotlin 怎样表达这种关系。当编写自己的泛型类或者泛型函数时，理解变型的概念是十分重要的：它有助于你创建出这样的 API，既不会以不方便的方式限制用户，也不会破坏用户所期望的类型安全。

9.3.1 为什么存在变型：给函数传递实参

假设你有一个接收 List<Any> 作为实参的函数。把 List<String> 类型的变量传给这个函数是否安全？毫无疑问，把一个字符串传给一个期望 Any 的函数是安全的，因为 String 类继承了 Any。但是当 String 和 Any 变成 List 接口的类型实参之后，情况就没有这么简单了。

例如，考虑一个打印出列表内容的函数。

```
fun printContents(list: List<Any>) {
    println(list.joinToString())
}

>>> printContents(listOf("abc", "bac"))
abc, bac
```

看起来这里字符串列表可以正常工作。函数把每个元素都当成 Any 对待，而且因为每一个字符串都是 Any，这是完全安全的。

现在来看另一个函数，它会修改列表（因此它接收一个 MutableList 作为参数）：

```
fun addAnswer(list: MutableList<Any>) {
    list.add(42)
}
```

如果把一个字符串列表传给这个函数，会有什么不好的事情发生吗？

```
>>> val strings = mutableListOf("abc", "bac")
>>> addAnswer(strings)
>>> println(strings.maxBy { it.length })
ClassCastException: Integer cannot be cast to String
```

如果这一行编译通过了……

……运行时就会产生一个异常。

你声明了一个类型为 MutableList<String> 的变量 strings，然后尝试把它传给这个函数。假设编译器接收了，你就能在字符串列表中添加一个整型，这会导致当你在运行时尝试访问列表中的字符串的时候出现异常。正因如此，这次调用不会通过编译。这个例子展示了当期望的是 MutableList<Any> 的时候把一个 MutableList<String> 当作实参传递是不安全的，Kotlin 编译器正确地禁止了它。

现在你可以回答刚才那个问题了，把一个字符串列表传给期望 Any 对象列表的函数是否安全。如果函数添加或者替换了列表中的元素就是不安全的，因为这样会产生类型不一致的可能性。否则它就是安全的（本节稍后我们将会更详细地讨论其原因）。在 Kotlin 中，这可以通过根据列表是否可变选择合适的接口来轻易地控制。如果函数接收的是只读列表，可以传递具有更具体的元素类型的列表。如果列表是可变的，你就不能这样做。

本节后面的内容会把同样的问题推广到任何泛型类，而不仅仅是 List。你将会看到为什么两种接口 List 和 MutableList 会因为它们的类型参数产生差异。但是在此之前，我们要讨论一下类型和子类型的概念。

9.3.2 类、类型和子类型

我们在 6.1.2 节讨论过，变量的类型规定了该变量的可能值。有时候我们会把类型和类当成同样的概念使用，但它们不一样，现在是时候看看它们的区别了。

最简单的例子就是非泛型类，类的名称可以直接当作类型使用。例如，如果你这样写 var x: String，就是声明了一个可以保存 String 类的实例的变量。但是注意，同样的类名称也可以用来声明可空类型：var x: String?。这意味着每一个 Kotlin 类都可以用于构造至少两种类型。

泛型类的情况就变得更复杂了。要得到一个合法的类型，需要用一个作为类型实参的具体类型替换（泛型）类的类型形参。List 不是一个类型（它是一个类），但是下面列举出来的所有替代品都是合法的类型：List<Int>、List<String?>、List<List<String>> 等。每一个泛型类都可能生成潜在的无限数量的类型。

为了讨论类型之间的关系，需要熟悉子类型这个术语。任何时候如果需要的是类型 A 的值，你都能够使用类型 B 的值（当作 A 的值），类型 B 就称为类型 A 的子类型。举例来说，Int 是 Number 的子类型，但 Int 不是 String 的子类型。这个定义还表明了任何类型都可以被认为是它自己的子类型，如图 9.4 所示。

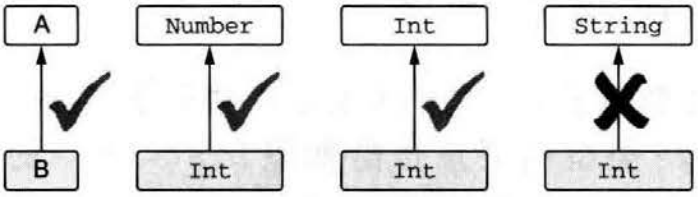


图 9.4 如果期望 A 的时候可以使用 B，B 就是 A 的子类型

术语超类型是子类型的反义词。如果 A 是 B 的子类型，那么 B 就是 A 的超类型。

为什么一个类型是否是另一个的子类型这么重要？编译器在每一次给变量赋值或者给函数传递实参的时候都要做这项检查。参考下面这个例子。

代码清单 9.10 检查一个类型是否是另一个的子类型

```
fun test(i: Int) {
    val n: Number = i

    fun f(s: String) { /*...*/ }
    f(i)
}
```

编译通过，因为 Int 是 Number 的子类型

不能编译，因为 Int 不是 String 的子类型

只有值的类型是变量类型的子类型时，才允许变量存储该值。例如，变量 n 的初始化器 i 的类型 Int 是变量的类型 Number 的子类型，所以 n 的声明是合法的。只有当表达式的类型是函数参数的类型的子类型时，才允许把该表达式传给函数。这个例子中 i 的类型 Int 不是函数参数的类型 String 的子类型，所以函数 f 的调用会编译失败。

简单的情况下，子类型和子类本质上意味着一样的事物。例如，Int 类是 Number 的子类，因此 Int 类型是 Number 类型的子类型。如果一个类实现了一个接口，它的类型就是该接口类型的子类型：String 是 CharSequence 的子类型。

可空类型提供了一个例子，说明子类型和子类不是同一个事物，如图 9.5 所示。

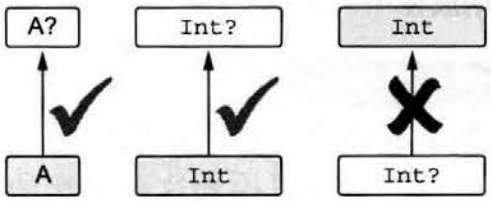


图 9.5 非空类型 A 是可空的 A? 的子类型，但反过来却不是

一个非空类型是它的可空版本的子类型，但它们都对应着同一个类。你始终能在可空类型的变量中存储非空类型的值，但反过来却不行（null 不是非空类型的变量可以接收的值）：

```
val s: String  = "abc"
val t: String? = s
```

这次赋值是合法的，因为 String 是 String? 的子类型

当我们开始涉及泛型类型时，子类型和子类之间的差异显得格外重要。前面一节的那个问题，把 List<String> 类型的变量传给期望 List<Any> 的函数是否安全，现在可以使用子类型化术语来重新组织：List<String> 是 List<Any> 的子类型吗？你已经了解了为什么把 MutableList<String> 当成 MutableList<Any> 的子类型对待是不安全的。显然，反过来也是不成立的：MutableList<Any> 肯定不是 MutableList<String> 的子类型。

一个泛型类——例如，MutableList——如果对于任意两种类型 A 和 B，MutableList<A> 既不是 MutableList<B> 的子类型也不是它的超类型，它就被称为在该类型参数上是*不变型的*。Java 中所有的类都是不变型的（尽管那些类具体的使用可以标记成可变型的，稍后你就会看到）。

在前一节中，你见过一个这样一个类，List，对它来说，子类型化规则不一样。Kotlin 中的 List 接口表示的是只读集合。如果 A 是 B 的子类型，那么 List<A> 就是 List<B> 的子类型。这样的类或者接口被称为*协变的*。下一小节会详细讨论协变的概念并解释什么时候才可以把类或者接口声明成协变的。

9.3.3 协变：保留子类型化关系

一个协变类是一个泛型类（我们以 Producer<T> 为例），对这种类来说，下面的描述是成立的：如果 A 是 B 的子类型，那么 Producer<A> 就是 Producer<B> 的子类型。我们说*子类型化被保留了*。例如，Producer<Cat> 是 Producer<Animal> 的子类型，因为 Cat 是 Animal 的子类型。

在 Kotlin 中，要声明类在某个类型参数上是可以协变的，在该类型参数的名称前加上 out 关键字即可：

```
interface Producer<out T> {
    fun produce(): T
}
```

类被声明成在 T 上协变

将一个类的类型参数标记为协变的，在该类型实参没有精确匹配到函数中定义的类型形参时，可以让该类的值作为这些函数的实参传递，也可以作为这些函数的返回值。例如，想象一下有这样一个函数，它负责喂养用类 Herd 代表的一群动物（畜

群），Herd 类的类型参数确定了畜群中动物的类型。

代码清单 9.11 定义一个不变型的类似集合的类

```
open class Animal {
    fun feed() { ... }
}

class Herd<T : Animal> {                    <-- 类型参数没有声明成协变的
    val size: Int get() = ...
    operator fun get(i: Int): T { ... }
}

fun feedAll(animals: Herd<Animal>) {
    for (i in 0 until animals.size) {
        animals[i].feed()
    }
}
```

假设这段代码的用户有一群猫需要照顾。

代码清单 9.12 使用一个不变型的类似集合的类

```
class Cat : Animal() {                      <-- Cat 是一个 Animal
    fun cleanLitter() { ... }
}

fun takeCareOfCats(cats: Herd<Cat>) {
    for (i in 0 until cats.size) {
        cats[i].cleanLitter()
        // feedAll(cats)                    <-- 错误：推导的类型是 Herd<Cat>，但期望的却是 Herd<Animal>
    }
}
```

很遗憾，这群猫要挨饿了：如果尝试把猫群传给 feedAll 函数，在编译期你就会得到类型不匹配的错误。因为 Herd 类中的类型参数 T 没有用任何变型修饰符，猫群不是畜群的子类。可以使用显式的类型转换来绕过这个问题，但是这种方法啰唆、易出错，而且几乎从来都不是解决类型不匹配问题的正确方式。

因为 Herd 类有一个类似 List 的 API，并且不允许它的调用者添加和改变畜群中的动物，可以把它变成协变的并相应地修改调用代码。

代码清单 9.13 使用一个协变的类似集合的类

```
class Herd<out T : Animal> {                <-- 类型参数 T 现在是协变的
    ...
}
```

```
fun takeCareOfCats(cats: Herd<Cat>) {
    for (i in 0 until cats.size) {
        cats[i].cleanLitter()
    }
    feedAll(cats)
}
```

不需要类型转换

你不能把任何类都变成协变的：这样不安全。让类在某个类型参数变为协变，限制了该类中对该类型参数使用的可能性。要保证类型安全，它只能用在所谓的 *out* 位置，意味着这个类只能生产类型 T 的值而不能消费它们。

在类成员的声明中类型参数的使用可以分为 *in* 位置和 *out* 位置。考虑这样一个类，它声明了一个类型参数 T 并包含了一个使用 T 的函数。如果函数是把 T 当成返回类型，我们说它在 out 位置。这种情况下，该函数生产类型为 T 的值。如果 T 用作函数参数的类型，它就在 in 位置。这样的函数消费类型为 T 的值，如图 9.6 所示。

```
interface Transformer<T> {
    fun transform(t: T): T
}
```

“in”位置 “out”位置

图 9.6 函数参数的类型叫作 in 位置，而函数返回类型叫作 out 位置

类的类型参数前的 out 关键字要求所有使用 T 的方法只能把 T 放在 out 位置而不能放在 in 位置。这个关键字约束了使用 T 的可能性，这保证了对应子类型关系的安全性。

以 Herd 类为例，它只在一个地方使用了类型参数 T：get 方法的返回值。

```
class Herd<out T : Animal> {
    val size: Int get() = ...
    operator fun get(i: Int): T { ... }
}
```

把 T 作为返回类型使用

这是一个 out 位置，可以安全地把类声明成协变的。如果 Herd<Animal> 类的 get 方法返回的是 Cat，任何调用该方法的代码都可以正常工作，因为 Cat 是 Animal 的子类型。

重申一下，类型参数 T 上的关键字 out 有两层含义：

- 子类型化会被保留（Producer<Cat> 是 Producer<Animal> 的子类型）
- T 只能用在 out 位置

现在我们看看 List<Interface> 接口。Kotlin 的 List 是只读的，所以它只有一个返回类型为 T 的元素的方法 get，而没有定义任何把类型为 T 的元素存储到列表中的方法。因此，它也是协变的。

```
interface List<out T> : Collection<T> {
    operator fun get(index: Int): T
    // ...
}
```

只读接口只定义了返回 T 的方法（所以 T 在“out”位置）

注意，类型形参不光可以直接当作参数类型或者返回类型使用，还可以当作另一个类型的类型实参。例如，List 接口就包含了一个返回 List<T> 的 subList 方法。

```
interface List<out T> : Collection<T> {
    fun subList(fromIndex: Int, toIndex: Int): List<T>
    // ...
}
```

这里 T 也在“out”位置

在这个例子中，函数 subList 中的 T 也用在 out 位置。这里我们不再继续深入细节，如果你对决定哪个位置是 out 哪个是 in 的精确算法感兴趣，可以在 Kotlin 语言文档中找到这些信息。

注意，不能把 MutableList<T> 在它的类型参数上声明成协变的，因为它既含有接收类型为 T 的值作为参数的方法，也含有返回这种值的方法（因此，T 出现在 in 和 out 两种位置上）。

```
interface MutableList<T>
        : List<T>, MutableCollection<T> {
    override fun add(element: T): Boolean
}
```

MutableList 不能在 T 上声明成协变的……

……因为 T 用在了“in”位置。

编译器强制实施了这种限制。如果这个类被声明成协变的，编译器会报告错误：Type parameter T is declared as 'out' but occurs in 'in' position（类型参数 T 声明为“out”但出现在“in”位置）。

注意，构造方法的参数既不在 in 位置，也不在 out 位置。即使类型参数声明成了 out，仍然可以在构造方法参数的声明中使用它：

```
class Herd<out T: Animal>(vararg animals: T) { ... }
```

如果把类的实例当成一个更泛化的类型的实例使用，变型会防止该实例被误用：不能调用存在潜在危险的方法。构造方法不是那种在实例创建之后还能调用的方法，因此它不会有潜在的危险。

然而，如果你在构造方法的参数上使用了关键字 val 和 var，同时就会声明一个 getter 和一个 setter（如果属性是可变的）。因此，对只读属性来说，类型参数用在了 out 位置，而可变属性在 out 位置和 in 位置都使用了它：

```
class Herd<T: Animal>(var leadAnimal: T, vararg animals: T) { ... }
```

上面这个例子中，T 不能用 out 标记，因为类包含属性 leadAnimal 的 setter，它在 in 位置用到了 T。

还需要留意的是，位置规则只覆盖了类外部可见的（public、protected 和 internal）API。私有方法的参数既不在 in 位置也不在 out 位置。变型规则只会防止外部使用者对类的误用但不会对类自己的实现起作用：

```
class Herd<out T: Animal>(private var leadAnimal: T, vararg animals: T) { ... }
```

现在可以安全地让 Herd 在 T 上协变，因为属性 leadAnimal 变成了私有的。

你可能会问如果类型参数只在 in 位置使用，类和接口会怎么样。这种情况下，逆向关系成立。接下来的一节展示了这种情况的细节。

9.3.4 逆变：反转子类型化关系

*逆变*的概念可以被看成是协变的镜像：对一个逆变类来说，它的子类型化关系与用作类型实参的类的子类型化关系是相反的。我们从 Comparator 接口的例子开始，这个接口定义了一个方法 compare 类，用于比较两个给定的对象：

```
interface Comparator<in T> {
    fun compare(e1: T, e2: T): Int { ... }
}
```

在“in”位置使用 T

如你所见，这个接口方法只是消费类型为 T 的值。这说明 T 只在 in 位置使用，因此它的声明之前用了 in 关键字。

一个为特定类型的值定义的比较器显然可以比较该类型任意子类型的值。例如，如果有一个 Comparator<Any>，可以用它比较任意具体类型的值。

```
>>> val anyComparator = Comparator<Any> {
...     e1, e2 -> e1.hashCode() - e2.hashCode()
... }
>>> val strings: List<String> = ...
>>> strings.sortedWith(anyComparator)
```

可以用任意对象的比较器比较具体对象，比如字符串

sortedWith 函数期望一个 Comparator<String>（一个可以比较字符串的比较器），传给它一个能比较更一般的类型的比较器是安全的。如果你要在特定类型的对象上执行比较，可以使用能处理该类型或者它的超类型的比较器。这说明 Comparator<Any> 是 Comparator<String> 的子类型，其中 Any 是 String 的超类型。不同类型之间的子类型关系和这些类型的比较器之间的子类型化关系截然相反。

现在你已经为完整的逆变定义做好了准备。一个在类型参数上*逆变*的类是这样的一个泛型类（我们以 Consumer<T> 为例），对这种类来说，下面的描述

是成立的：如果 B 是 A 的子类型，那么 Consumer<A> 就是 Consumer<B> 的子类型。类型参数 A 和 B 交换了位置，所以我们说子类型化被反转了。例如，Consumer<Animal> 就是 Consumer<Cat> 的子类型。

图 9.7 展示了在类型参数上协变和逆变的类之间子类型化关系的差异。可以看到对 Producer 类来说，子类型化关系复制了它的类型实参的子类型化关系，而对 Consumer 类来说，关系反转过来了。

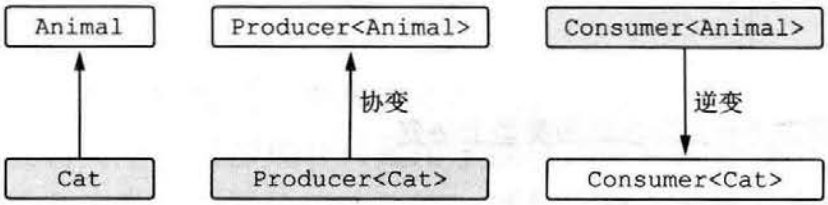


图 9.7 对协变类型 Producer<T> 来说，子类型化保留了，但对逆变类型 Consumer<T> 来说，子类型化反转了

in 关键字的意思是，对应类型的值是传递进来给这个类的方法的，并且被这些方法消费。和协变的情况类似，约束类型参数的使用将导致特定的子类型化关系。在类型参数 T 上的 in 关键字意味着子类型化被反转了，而且 T 只能用在 in 位置。表 9.1 总结了可能的变型选择之间的差异。

表 9.1 协变的、逆变的和不变型的类

协变	逆变	不变型
Producer<out T>	Consumer<in T>	MutableList<T>
类的子类型化保留了：Producer<Cat> 是 Producer<Animal> 的子类型	子类型化反转了：Consumer<Animal> 是 Consumer<Cat> 的子类型	没有子类型化
T 只能在 out 位置	T 只能在 in 位置	T 可以在任何位置

一个类可以在一个类型参数上协变，同时在另外一个类型参数上逆变。Function 接口就是一个经典的例子。下面是一个单个参数的 Function 的声明：

```
interface Function1<in P, out R> {
    operator fun invoke(p: P): R
}
```

Kotlin 的表示法 (P) -> R 是表达 Function<P, R> 的另一种更具可读性的形式。可以发现用 in 关键字标记的 P（参数类型）只用在 in 位置，而用 out 关键字标记的 R（返回类型）只用在 out 位置。这意味着对这个函数类型的第一个类型参数来说，子类型化反转了，而对于第二个类型参数来说，子类型化保留了。例如，你有一个高阶函数，该函数尝试对你所有的猫进行迭代，你可以把一个接收任意动物的 lambda 传给它。

```
fun enumerateCats(f: (Cat) -> Number) { ... }
fun Animal.getIndex(): Int = ...

>>> enumerateCats(Animal::getIndex)
```

在 Kotlin 中这段代码是合法的。Animal 是 Cat 的超类型，而 Int 是 Number 的子类型

图 9.8 展示了前面这个例子中的子类型化关系。

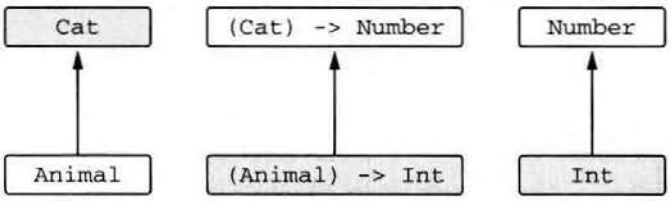


图 9.8 函数 (P) -> R 在它的参数类型上逆变而在返回类型上协变

9.3.5 使用点变型：在类型出现的地方指定变型

在类声明的时候就能够指定变型修饰符是很方便的，因为这些修饰符会应用到所有类被使用的地方。这被称作*声明点变型*。如果你熟悉 Java 的通配符类型（? extends 和 ? super），你会意识到 Java 用完全不同的方式处理变型。在 Java 中，每一次使用带类型参数的类型的时候，还可以指定这个类型参数是否可以用它的子类型或者超类型替换。这叫作*使用点变型*。

Kotlin 的声明点变型 vs. Java 通配符

声明点变型带来了更简洁的代码，因为只用指定一次变型修饰符，所有这个类的使用者都不用再考虑这些了。在 Java 中，库作者不得不一直使用通配符：Function<? super T, ? extends R>，来创建按照用户期望运行的 API。如果你查看 Java 8 标准库的源码，你会在每次用到 Function 接口的地方发现通配符。例如，下面是 Stream.map 方法的声明：

```
/* Java */
public interface Stream<T> {
    <R> Stream<R> map(Function<? super T, ? extends R> mapper);
}
```

在声明时一次性指定变型让代码变得简洁和优雅得多。

Kotlin 也支持使用点变型，允许在类型参数出现的具体位置指定变型，即使在类型声明时它不能被声明成协变或逆变的。我们看看它是如何工作的。

你已经见过许多像 MutableList 这样的接口，通常情况下既不是协变也不是逆变的，因为它同时生产和消费指定为它们类型参数的类型的值。但是对于这个类型的变量来说，在某个特定函数中只被当成其中一种角色使用的情况挺常见的：要

么是生产者要么是消费者。例如下面这个简单的函数。

代码清单 9.14 带不变型类型参数的数据拷贝函数

```
fun <T> copyData(source: MutableList<T>,
                 destination: MutableList<T>) {
    for (item in source) {
        destination.add(item)
    }
}
```

这个函数从一个集合中把元素拷贝到另一个集合中。尽管两个集合都拥有不变型的类型，来源集合只是用于读取，而目标集合只是用于写入。这种情况下，集合元素的类型不需要精确匹配。例如，把一个字符串的集合拷贝到可以包含任意对象的集合中一点儿问题也没有。

要让这个函数支持不同类型的列表，可以引入第二个泛型参数。

代码清单 9.15 带不变型类型参数的数据拷贝函数

```
fun <T: R, R> copyData(source: MutableList<T>,
                       destination: MutableList<R>) {
    for (item in source) {
        destination.add(item)
    }
}
>>> val ints = mutableListOf(1, 2, 3)
>>> val anyItems = mutableListOf<Any>()
>>> copyData(ints, anyItems)
>>> println(anyItems)
[1, 2, 3]
```

来源的元素类型应该是目标元素类型的子类型

可以调用这个函数，因为 Int 是 Any 的子类型

你声明了两个泛型参数代表来源列表和目标列表中的元素类型。为了能够把一个列表中的元素拷贝到另一个列表中，来源元素类型应该是目标列表中的元素的子类型，就像代码清单 9.15 中 Int 是 Any 的子类型。

但是 Kotlin 提供了一种更优雅的表达方式。当函数的实现调用了那些类型参数只出现在 out 位置（或只出现在 in 位置）的方法时，可以充分利用这一点，在函数定义中给特定用途的类型参数加上变型修饰符。

代码清单 9.16 带 out 投影类型参数的数据拷贝函数

```
fun <T> copyData(source: MutableList<out T>,
                 destination: MutableList<T>) {
    for (item in source) {
        destination.add(item)
    }
}
```

可以给类型的用法加上“out”关键字：没有使用那些 T 用在“in”位置的方法

可以为类型声明中类型参数任意的用法指定变型修饰符，这些用法包括：形参类型（就像代码清单 9.16 这样）、局部变量类型、函数返回类型，等等。这里发生的一切被称作类型投影：我们说 source 不是一个常规的 MutableList，而是一个投影（受限）的 MutableList。只能调用返回类型是泛型类型参数的那些方法，或者严格地讲，只在 out 位置使用它的方法。编译器禁止调用使用类型参数做实参（类型）的那些方法（在 in 位置使用类型参数）：

```
>>> val list: MutableList<out Number> = ...
>>> list.add(42)
Error: Out-projected type 'MutableList<out Number>' prohibits
the use of 'fun add(element: E): Boolean'
```

*不要为使用投影类型后不能调用某些方法而吃惊。*如果需要调用那些方法，你要用的是常规类型而不是投影。这可能要求你声明第二个类型参数，它依赖的是原本要进行投影的类型，就像代码清单 9.15 中那样。

当然，实现 copyData 函数的正确方式应该是使用 List<T> 作为 source 实参的类型，因为我们只用了声明在 List 中的方法，并没用到 MutableList 中的方法，而且 List 类型参数的变型在声明时就指定了。但这个例子对展示这个概念依然十分重要，尤其是要记住大多数的类并没有像 List 和 MutableList 这样分开的两个接口，一个是协变的读取接口，另一个是不变型的读取 / 写入接口。

如果类型参数已经有 out 变型，获取它的 out 投影没有任何意义。就像 List<out T> 这样。它和 List<T> 是一个意思，因为 List 已经声明成了 class List<out T>。Kotlin 编译器会发出警告，表明这样投影是多余的。

同理，可以对类型参数的用法使用 in 修饰符，来表明在这个特定的地方，相应的值担当的是消费者，而且类型参数可以使用它的任意子类型替换。下面展示了如何使用 in 投影来重写代码清单 9.16 中的代码。

代码清单 9.17 带 in 投影类型参数的数据拷贝函数

```
fun <T> copyData(source: MutableList<T>,
                 destination: MutableList<in T>) {
    for (item in source) {
        destination.add(item)
    }
}
```

允许目标元素的类型是来源元素类型的超类型

注意 Kotlin 的使用点变型直接对应 Java 的限界通配符。Kotlin 中的 MutableList<out T> 和 Java 中 的 MutableList<? extends T> 是一个意思。in 投影的 MutableList<in T> 对应到 Java 的 MutableList<? super T>。

使用点变型有助于放宽可接收的类型的范围。现在我们讨论一种极端情况：这种情况下（泛型）类型使用所有可能的类型实参，都是可以接受的。

9.3.6 星号投影：使用 * 代替类型参数

本章前面提到类型检查和转换的时候，我们提到了一种特殊的星号投影语法，可以用它来表明你不知道关于泛型实参的任何信息。例如，一个包含未知类型的元素的列表用这种语法表示为 List<*>。现在我们深入探讨星号投影的语义。

首先，需要注意的是 MutableList<*> 和 MutableList<Any?> 不一样（这里非常重要的是 MutableList<T> 在 T 上是不变型的）。你确信 MutableList<Any?> 这种列表包含的是任意类型的元素。而另一方面，MutableList<*> 是包含某种特定类型元素的列表，但是你不知道是哪个类型。这种列表被创建成一个包含某种特定类型元素的列表，比如 String（你无法创建一个 ArrayList<*>），而且创建它的代码期望只包含那种类型的元素。因为不知道是哪个类型，你不能向列表中写入任何东西，因为你写入的任何值都可能会违反调用代码的期望。但是从列表中读取元素是可行的，因为你心里有数，所有的存储在列表中的值都能匹配所有 Kotlin 类型的超类型 Any?：

```
>>> val list: MutableList<Any?> = mutableListOf('a', 1, "qwe")
>>> val chars = mutableListOf('a', 'b', 'c')
>>> val unknownElements: MutableList<*> =
...         if (Random().nextBoolean()) list else chars
>>> unknownElements.add(42)
Error: Out-projected type 'MutableList<*>' prohibits
the use of 'fun add(element: E): Boolean'
>>> println(unknownElements.first())
a
```

MutableList<*> 和 MutableList<Any?> 不一样。

编译器禁止调用这个方法。

读取元素是安全的：first() 返回一个类型为 Any? 的元素。

为什么编译器会把 MutableList<*> 当成 out 投影的类型？在这个例子的上下文中，MutableList<*> 投影成了 MutableList<out Any?>：当你没有任何元素类型信息的时候，读取 Any? 类型的元素仍然是安全的，但是向列表中写入元素是不安全的。谈到 Java 通配符，Kotlin 的 MyType<*> 对应于 Java 的 MyType<?>。

> **注意** 对像 Consumer<in T> 这样的逆变类型参数来说，星号投影等价于 <in Nothing>。实际上，在这种星号投影中无法调用用任何签名中有 T 的方法。如果类型参数是逆变的，它就只能表现为一个消费者，而且，我们之前讨论过，你不知道它可以消费的到底是什么。因此，不能让它消费任何东西。如果你对其中更多的细节感兴趣，请查看 Kotlin 在线文档（http://mng.bz/3Ed7）。

当类型实参的信息并不重要的时候，可以使用星号投影的语法：不需要使用任何在签名中引用类型参数的方法，或者只是读取数据而不关心它的具体类型。例如，可以实现一个接收 List<*> 做参数的 printFirst 函数：

```
fun printFirst(list: List<*>) {
    if (list.isNotEmpty()) {
        println(list.first())
    }
}

>>> printFirst(listOf("Svetlana", "Dmitry"))
Svetlana
```

每一种列表都是可能的实参。

isNotEmpty() 没有使用泛型类型参数。

first() 现在返回的是 Any?，但是这里足够了。

在使用点变型的情况下，你有一个替代方案——引入一个泛型类型参数：

```
fun <T> printFirst(list: List<T>) {
    if (list.isNotEmpty()) {
        println(list.first())
    }
}
```

再一次，每一种列表都是可能的实参。

first() 现在返回的是 T 的值。

星号投影的语法很简洁，但只能用在对泛型类型实参的确切值不感兴趣的地方：只是使用生产值的方法，而且不关心那些值的类型。

现在我们来看另外一个使用星号投影的类型的例子，以及使用这种方式时常见的会困扰你的陷阱。假设你需要验证用户的输入，并声明了一个接口 FieldValidator。它只包含在 in 位置的类型参数，所以声明成了逆变的。而且，事实上，当期望的是字符串验证器时使用可验证任意元素的验证器也是没有问题的（这正是把它声明成逆变带来的效果）。你还声明了两个验证器来分别处理 String 和 Int。

代码清单 9.18 输入验证的接口

```
interface FieldValidator<in T> {
    fun validate(input: T): Boolean
}

object DefaultStringValidator : FieldValidator<String> {
    override fun validate(input: String) = input.isNotEmpty()
}

object DefaultIntValidator : FieldValidator<Int> {
    override fun validate(input: Int) = input >= 0
}
```

接口定义成在 T 上逆变。

T 只在“in”位置使用（这个方法只是消费 T 的值）。

现在假设你想要把所有的验证器都存储到同一个容器中，并根据输入的类型来选出正确的验证器。你首先会想到用 map 来存储它们。你要存储的是任意类型的验证器，所以你声明了 KClass（代表一个 Kotlin 类——第 10 章会详细介绍 KClass）到 FieldValidator<*>（可以指向任意类型的验证器）的 map：

```
>>> val validators = mutableMapOf<KClass<*>, FieldValidator<*>>()
>>> validators[String::class] = DefaultStringValidator
>>> validators[Int::class] = DefaultIntValidator
```

如果你这样做了，尝试使用验证器的时候就会遇到困难。不能用类型为 FieldValidator<*> 的验证器来验证字符串。这是不安全的，因为编译器不知道它是哪种验证器：

```
>>> validators[String::class]!!.validate("")
Error: Out-projected type 'FieldValidator<*>' prohibits
the use of 'fun validate(input: T): Boolean'
```

存储在 map 中的值的类型是 FieldValidator<*>。

在前面尝试向 MutableList<*> 中写入元素的时候，你已经见过这个错误了。这种情况下，这个错误的意思是把具体类型的值传给未知类型的验证器是不安全的。一种修正的方法是把验证器显式地转换成需要的类型。这样做是不安全的，也是不推荐的。但我们还是把它作为让代码快速通过编译的技巧展示在这里，这样可以在后面来重构它。

代码清单 9.19 使用显式的转换获取验证器

```
>>> val stringValidator = validators[String::class]
                            as FieldValidator<String>
>>> println(stringValidator.validate(""))
false
```

警告：未受检的转换

编译器发出了未受检转换的警告。注意，尽管如此，这段代码只有在验证时可能失败，而不是在转换时，因为运行时所有的泛型信息都被擦除了。

代码清单 9.20 错误地获取验证器

```
>>> val stringValidator = validators[Int::class]
                            as FieldValidator<String>
>>> stringValidator.validate("")
java.lang.ClassCastException:
  java.lang.String cannot be cast to java.lang.Number
  at DefaultIntValidator.validate
```

得到了一个错误的验证器（可能是不小心），但代码可以编译。

这仅仅是一个警告。

直到使用验证器时才发现真正的错误。

错误的代码和代码清单 9.19 中的代码在两种情境下是相似的，都只会发出警告。

只转换那些类型正确的值是你的职责。

这种解决方法不是类型安全的，而且容易出错。所以，如果想要把不同类型的验证器存储在同一个地方，我们要研究一下其他的选择。

代码清单 9.21 中的解决方法使用了同样的 map `validators`，但是把所有对它的访问封装到了两个泛型方法中，它们负责保证只有正确的验证器被注册和返回。这段代码依然会发出未受检转换的警告（这之前的一样），但这里的 `Validators` 对象控制了所有对 map 的访问，保证了没有任何人会错误地改变 map。

代码清单 9.21 封装对验证器集合的访问

```
object Validators {
    private val validators =
            mutableMapOf<KClass<*>, FieldValidator<*>>()

    fun <T: Any> registerValidator(
            kClass: KClass<T>, fieldValidator: FieldValidator<T>) {
        validators[kClass] = fieldValidator
    }

    @Suppress("UNCHECKED_CAST")
    operator fun <T: Any> get(kClass: KClass<T>): FieldValidator<T> =
        validators[kClass] as? FieldValidator<T>
                ?: throw IllegalArgumentException(
                "No validator for ${kClass.simpleName}")
}
```

使用和之前一样的 map，但现在无法在外部访问它

只有正确的键值对被写入 map，即当验证器正好对应到类的时候

禁止关于未受检的转换到 FieldValidator<T> 的警告

```
>>> Validators.registerValidator(String::class, DefaultStringValidator)
>>> Validators.registerValidator(Int::class, DefaultIntValidator)

>>> println(Validators[String::class].validate("Kotlin"))
true
>>> println(Validators[Int::class].validate(42))
true
```

现在你拥有了一个类型安全的 API。所有不安全的逻辑都被隐藏在类的主体中，通过把这些逻辑局部化，保证了它不会被错误地使用。编译器禁止使用错误的验证器，因为 `Validators` 对象始终都会给出正确的验证器实现：

```
>>> println(Validators[String::class].validate(42))
Error: The integer literal does not conform to the expected type String
```

现在“get”方法返回的是 FieldValidator<String> 的实例

这种模式可以轻松地推广到任意自定义泛型类的存储。把不安全的代码局部化到一个分开的位置预防了误用，而且让容器的使用变得安全。注意这里描述的模式

并不只是针对 Kotlin，在 Java 中也可以使用同样的方法。

泛型和变型往往被认为是 Java 语言中最难处理的部分。在 Kotlin 中我们力图拿出一个易读且易用的设计方案，同时保留和 Java 之间的互操作性。

9.4 小结

- Kotlin 的泛型和 Java 相当接近：它们使用同样的方式声明泛型函数和泛型类。
- 和 Java 一样，泛型类型的类型实参只在编译期存在。
- 不能把带类型实参的类型和 `is` 运算符一起使用，因为类型实参在运行时将被擦除。
- 内联函数的类型参数可以标记成实化的，允许你在运行时对它们使用 `is` 检查，以及获得 `java.lang.Class` 实例。
- 变型是一种说明两种拥有相同基础类型和不同类型参数的泛型类型之间子类型化关系的方式，它说明了如果其中一个泛型类型的类型参数是另一个的类型参数的子类型，这个泛型类型就是另外一个泛型类型的子类型或者超类型。
- 可以声明一个类在某个类型参数上是协变的，如果该参数只是用在 `out` 位置。
- 逆变的情况正好相反：可以声明一个类在某个类型参数上是逆变的，如果该参数只是用在 `in` 位置。
- Kotlin 中的只读接口 `List` 声明成了协变的，这意味着 `List<String>` 是 `List<Any>` 的子类型。
- 函数接口声明成了在第一个类型参数上逆变而在第二个类型参数上协变，使 `(Animal)->Int` 成为 `(Cat)->Number` 的子类型。
- 在 Kotlin 中既可以为整个泛型类指定变型（声明点变型），也可以为泛型类型特定的使用指定变型（使用点变型）。
- 当确切的类型实参是未知的或者不重要的时候，可以使用星号投影语法。

10 注解与反射

本章内容包括

- 应用和定义注解
- 在运行时使用反射对类进行自省
- 一个真正的 Kotlin 项目实例

截止目前，你已经见过了许多关于类和函数的特性，但是它们全部要求在使用这些类和函数的时候说明它们的确切名称，作为程序代码的一部分。要调用一个函数，你需要知道它定义在哪个类中，还有它的名称和参数的类型。注解和反射给你超越这个规则的能力，并让你编写出使用事先未知的任意类的代码。可以使用注解赋予这些类库特定的语义，而反射允许你在运行时分析这些类的结构。

应用注解非常直截了当。但编写你自己的注解尤其是编写处理它们的代码，就没有这么简单了。使用注解的语法和 Java 完全一样，而声明自己注解类的语法却略有不同。反射 API 的大体结构与 Java 相仿，但细节存在差异。

作为注解和反射用法的演示，我们将会带你浏览一个真实项目的实现：一个叫作 JKid 的库，用来序列化和反序列化 JSON。这个库在运行时用反射访问任意的 Kotlin 对象，同时还根据 JSON 文件中提供的数据创建对象。注解则可以让你定制具体的类和属性是如何被这个库序列化和反序列化的。

10.1 声明并应用注解

绝大多数现代的 Java 框架都大量地使用了注解，所以你一定在开发 Java 应用程序的时候遇见过它们。Kotlin 中的核心概念是一样的。一个注解允许你把额外的元数据关联到一个声明上。然后元数据就可以被相关的源代码工具访问，通过编译好的类文件或是在运行时，取决于这个注解是如何配置的。

10.1.1 应用注解

在 Kotlin 中使用注解的方法和 Java 一样。要应用一个注解，以 @ 字符作为(注解)名字的前缀，并放在要注解的声明最前面。可以注解不同的代码元素，比如函数和类。

例如，如果你正在使用 JUnit 框架（http://junit.org/junit4/），可以用 @Test 标记一个测试方法：

```
import org.junit.*

class MyTest {
    @Test fun testTrue() {
        Assert.assertTrue(true)
    }
}
```

@Test 注解指引 JUnit 框架把这个方法当测试调用

我们再来看一个更有趣的例子，@Deprecated 注解。它在 Kotlin 中的含义和 Java 一样，但是 Kotlin 用 replaceWith 参数增强了它，让你可以提供一个替代者的（匹配）模式，以支持平滑地过渡到 API 的新版本。下面的例子向你展示了如何给该注解提供实参（一条不推荐使用的消息和一个替代者的模式）：

```
@Deprecated("Use removeAt(index) instead.", ReplaceWith("removeAt(index)"))
fun remove(index: Int) { ... }
```

实参在括号中传递，就和常规函数的调用一样。用了这种声明之后，如果有人使用了 remove 函数，IntelliJ IDEA 不仅会提示应该使用哪个函数来代替它（这个例子中是 removeAt），还会提供一个自动的快速修正。

注解只能拥有如下类型的参数：基本数据类型、字符串、枚举、类引用、其他的注解类，以及前面这些类型的数组。指定注解实参的语法与 Java 有些微小的差别：

- 要把一个类指定为注解实参，在类名后加上 ::class：@MyAnnotation(MyClass::class)。
- 要把另一个注解指定为一个实参，去掉注解名称前面的 @。例如，前面例子中的 ReplaceWith 是一个注解，但是你把它指定为 Deprecated 注解的实参时没有用 @。

- 要把一个数组指定为一个实参，使用 arrayOf 函数：@RequestMapping (path = arrayOf("/foo", "/bar"))。如果注解类是在 Java 中声明的，命名为 value 的形参按需自动地被转换成可变长度的形参，所以不用 arrayOf 函数就可以提供多个实参。

注解实参需要在编译期就是已知的，所以你不能引用任意的属性作为实参。要把属性当作注解实参使用，你需要用 const 修饰符标记它，来告知编译器这个属性是编译期常量。下面是一个 JUnit @Test 注解的例子，使用 timeout 参数指定测试超时时长，单位为毫秒：

```
const val TEST_TIMEOUT = 100L

@Test(timeout = TEST_TIMEOUT) fun testMethod() { ... }
```

正如 3.3.1 节讨论过的，用 const 标注的属性可以声明在一个文件的顶层或者一个 object 之中，而且必须初始化为基本数据类型或者 String 类型的值。如果你尝试使用普通属性作为注解实参，将会得到一个错误“Only ‘const val’ can be used in constant expressions.”（只有‘const val’才能用在常量表达式中）。

10.1.2 注解目标

许多情况下，Kotlin 源代码中的单个声明会对应成多个 Java 声明，而且它们每个都能携带注解。例如，一个 Kotlin 属性就对应了一个 Java 字段、一个 getter，以及一个潜在的 setter 和它的参数。而一个在主构造方法中声明的属性还多拥有一个对应的元素：构造方法的参数。因此，说明这些元素中哪些需要注解十分必要。

使用点目标声明被用来说明要注解的元素。使用点目标被放在 @ 符号和注解名称之间，并用冒号和注解名称隔开。图 10.1 中的单词 get 导致注解 @Rule 被应用到了属性的 getter 上。



图 10.1 说明使用点目标的语法

下面我们来看一个使用这个注解的例子。在 JUnit 中可以指定一个每个测试方法被执行之前都会执行的规则。例如，标准的 TemporaryFolder 规则用来创建文件和文件夹，并在测试结束后删除它们。

要指定一个规则，在 Java 中需要声明一个用 @Rule 注解的 public 字段或者方法。如果在你的 Kotlin 测试类中只是用 @Rule 注解了属性 folder，你会得到一

个 JUnit 异常："The (???) 'folder' must be public."（(???) 'folder' 必须是公有的）。这是因为 @Rule 被应用到了字段上，而字段默认是私有的。要把它应用到（公有的）getter 上，要显式地写出来，@get:Rule，就像下面这样：

```
class HasTempFolder {
    @get:Rule
    val folder = TemporaryFolder()

    @Test
    fun testUsingTempFolder() {
        val createdFile = folder.newFile("myfile.txt")
        val createdFolder = folder.newFolder("subfolder")
        // ...
    }
}
```

注解的是 getter，而不是属性。

如果你使用 Java 中声明的注解来注解一个属性，它会被默认地应用到相应的字段上。Kotlin 也可以让你声明被直接对应到属性上的注解。

Kotlin 支持的使用点目标的完整列表如下：

- property——Java 的注解不能应用这种使用点目标。
- field——为属性生成的字段。
- get——属性的 getter。
- set——属性的 setter。
- receiver——扩展函数或者扩展属性的接收者参数。
- param——构造方法的参数。
- setparam——属性 setter 的参数。
- delegate——为委托属性存储委托实例的字段。
- file——包含在文件中声明的顶层函数和属性的类。

任何应用到 file 目标的注解都必须放在文件的顶层，放在 package 指令之前。@JvmName 是常见的应用到文件的注解之一，它改变了对应类的名称。3.2.3 节中已经展示过一个例子：@file:JvmName("StringFunctions")。

注意，和 Java 不一样的是，Kotlin 允许你对任意的表达式应用注解，而不仅仅是类和函数的声明及类型。最常见的例子就是 @Suppress 注解，可以用它抑制被注解的表达式的上下文中的特定的编译器警告。下面就是一个注解局部变量声明的例子，抑制了未受检转换的警告：

```
fun test(list: List<*>) {
    @Suppress("UNCHECKED_CAST")
    val strings = list as List<String>
    // ...
}
```

注意，在 IntelliJ IDEA 中，在出现这个编译器警告的地方，按下 Alt+Enter 组合键并从意向选项菜单中选择 Suppress（抑制），IntelliJ IDEA 就会帮你插入这个注解。

用注解控制 Java API

Kotlin 提供了各种注解来控制 Kotlin 编写的声明如何编译成字节码并暴露给 Java 调用者。其中一些注解代替了 Java 语言中对应的关键字：比如，注解 `@Volatile` 和 `@Strictfp` 直接充当了 Java 的关键字 `volatile` 和 `strictfp` 的替身。其他的注解则是被用来改变 Kotlin 声明对 Java 调用者的可见性：

- `@JvmName` 将改变由 Kotlin 生成的 Java 方法或字段的名称。
- `@JvmStatic` 能被用在对象声明或者伴生对象的方法上，把它们暴露成 Java 的静态方法。
- `@JvmOverloads`，曾在 3.2.2 节中出现过，指导 Kotlin 编译器为带默认参数值的函数生成多个重载（函数）。
- `@JvmField` 可以应用于一个属性，把这个属性暴露成一个没有访问器的公有 Java 字段。

可以在这些注解的文档注释和在线文档中关于 Java 互操作的章节中找到更多它们用法的细节。

10.1.3 使用注解定制 JSON 序列化

注解的经典用法之一就是定制化对象的序列化。序列化就是一个过程，把对象转换成可以存储或者在网络上传输的二进制或者文本的表示法。它的逆向过程，反序列化，把这种表示法转换回一个对象。而最常见的一种用来序列化的格式就是 JSON。已经有很多广泛使用的库可以把 Java 对象序列化成 JSON，包括 Jackson（https://github.com/FasterXML/jackson）和 GSON（https://github.com/google/gson）。就和任何其他 Java 库一样，它们和 Kotlin 完全兼容。

在本章中，我们将会讨论一个满足此用途的名为 JKid 的纯 Kotlin 库。它足够小巧，你可以轻松地读完它的全部源码，我们也鼓励你在阅读本章的同时阅读它的源码。

JKid 库源码和练习

JKid 完整的实现是本书源代码的一部分，可以在线上找到：https://manning.com/books/kotlin-in-action 和 http://github.com/yole/jkid。要学习库的实现和例子，在 IDE 中把文件 ch10/jkid/build.gradle 作为 Gradle 项目打开。在项目的 src/test/kotlin/examples 目录下可以找到这些例子。这个库并不像 GSON 或者 Jackson 那样完善和灵活，但它的性能足够在真实项目中使用，如果它符合你的项目需求，欢迎你使用它。

JKid 项目有一系列的练习，在读完本章的内容之后可以把练习都做一遍，来巩固对所有概念的理解。可以在项目的 README.md 文件中找到关于练习的描述，也可以在 GitHub 的项目页面上阅读。

让我们从最简单的例子开始，测试一下这个库：序列化和反序列化一个 Person 类的实例。把实例传给 serialize 函数，然后它就会返回一个包含该实例 JSON 表示法的字符串：

```
data class Person(val name: String, val age: Int)

>>> val person = Person("Alice", 29)
>>> println(serialize(person))
{"age": 29, "name": "Alice"}
```

一个对象的 JSON 表示法由键值对组成：具体实例的属性名称和它们的值之间的键值对，比如："age": 29。

要从 JSON 表示法中取回一个对象，要调用 deserialize 函数：

```
>>> val json = """{"name": "Alice", "age": 29}"""
>>> println(deserialize<Person>(json))
Person(name=Alice, age=29)
```

当你从 JSON 数据中创建实例的时候，必须显式地指定一个类作为类型参数，因为 JSON 没有存储对象的类型。这种情况下，你要传递 Person 类。

图 10.2 展示了一个对象和它的 JSON 表示法之间的等价关系。注意序列化之后的类能包含的不仅是图中展示的这些基本数据类型或者字符串类型的值，还可以是集合，以及其他值对象类的实例。

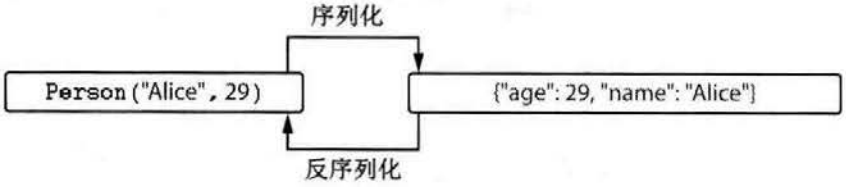


图 10.2 Person 实例的序列化和反序列化

你可以使用注解来定制对象序列化和反序列化的方式。当把一个对象序列化成JSON的时候，默认情况下这个库尝试序列化所有属性，并使用属性名称作为键。注解允许你改变默认的行为，这一节我们会讨论两个注解，@JsonExclude 和 @JsonName，本章稍后你就会看到它们的实现：

- @JsonExclude 注解用来标记一个属性，这个属性应该排除在序列化和反序列化之外。
- @JsonName 注解让你说明代表这个属性的（JSON）键值对之中的键应该是一个给定的字符串，而不是属性的名称。

参考下面这个例子：

```
data class Person(
    @JsonName("alias") val firstName: String,
    @JsonExclude val age: Int? = null
)
```

你注解了属性 firstName，来改变在 JSON 中用来表示它的键。而属性 age 也被注解了，在序列化和反序列化的时候会排除它。注意，你必须指定属性 age 的默认值。否则，在反序列化时你无法创建一个 Person 的新实例。图 10.3 展示了 Person 类实例的表示法发生了怎样的变化。

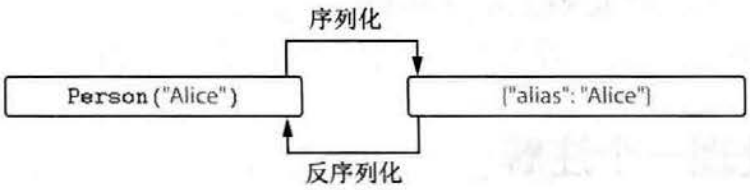


图 10.3 应用注解之后 Person 实例的序列化和反序列化

你已经见过了 JKid 中出现的大多数功能：serialize()、deserialize()、@JsonName 和 @JsonExclude。现在我们开始深入探索它的实现，就从注解声明开始。

10.1.4 声明注解

在这一节，你会以 JKid 库中的注解为例学习怎样声明它们。注解 @JsonExclude 有着最简单的形式，因为它没有任何参数：

```
annotation class JsonExclude
```

语法看起来和常规类的声明很像，只是在 class 关键字之前加上了 annotation 修饰符。因为注解类只是用来定义关联到声明和表达式的元数据的结

构，它们不能包含任何代码。因此，编译器禁止为一个注解类指定类主体。

对拥有参数的注解来说，在类的主构造方法中声明这些参数：

```
annotation class JsonName(val name: String)
```

你用的是常规的主构造方法的声明语法。对一个注解类的所有参数来说，val关键字是强制的。

作为对比，下面是如何在Java中声明同样的注解：

```
/* Java */
public @interface JsonName {
    String value();
}
```

注意，Java注解拥有一个叫作value的方法，而Kotlin注解拥有一个name属性。Java中value方法很特殊：当你应用一个注解时，你需要提供value以外所有指定特性的显式名称。而另一方面，在Kotlin中应用注解就是常规的构造方法调用。可以使用命名实参语法让实参的名称变成显式的，或者可以省略掉这些实参的名称：@JsonName(name = "first_name")和@JsonName("first_name")含义一样，因为name是JsonName构造方法的第一个形参（它的名称可以省略）。然而，如果你需要把Java中声明的注解应用到Kotlin元素上，必须对除了value以外的所有实参使用命名实参语法，而value也会被Kotlin特殊对待。

接下来，我们将讨论如何控制注解的使用，以及如何把注解应用到其他注解上。

10.1.5 元注解：控制如何处理一个注解

和Java一样，一个Kotlin注解类自己也可以被注解。可以应用到注解类上的注解被称作*元注解*。标准库中定义了一些元注解，它们会控制编译器如何处理注解。其他一些框架也会用到元注解——例如，许多依赖注入库使用了元注解来标记其他注解，表示这些注解用来识别拥有同样类型的不同的可注入对象。

标准库定义的元注解中最常见的就是@Target。JKid中@JsonExclude和@JsonName的声明使用它为这些注解指定有效的目标。下面展示了它是如何应用(在注解上）的：

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

@Target元注解说明了注解可以被应用的元素类型。如果不使用它，所有的声明都可以应用这个注解。这并不是JKid想要的，因为它只需要处理属性的注解。

AnnotationTarget枚举的值列出了可以应用注解的全部可能的目标。包括：

类、文件、函数、属性、属性访问器、所有的表达式，等等。如果需要，你还可以声明多个目标：@Target(AnnotationTarget.CLASS, AnnotationTarget.METHOD)。

要声明你自己的元注解，使用 ANNOTATION_CLASS 作为目标就好了：

```
@Target(AnnotationTarget.ANNOTATION_CLASS)
annotation class BindingAnnotation

@BindingAnnotation
annotation class MyBinding
```

注意，在 Java 代码中无法使用目标为 PROPERTY 的注解；要让这样的注解可以在 Java 中使用，可以给它添加第二个目标 AnnotationTarget.FIELD。这样注解既可以应用到 Kotlin 中的属性上，也可以应用到 Java 中的字段上。

@Retention 注解

你也许在 Java 中见过另一个重要的元注解：@Retention。它被用来说明你声明的注解是否会存储到 .class 文件，以及在运行时是否可以通过反射来访问它。Java 默认会在 .class 文件中保留注解但不会让它们在运行时被访问到。大多数注解确实需要在运行时存在，所以 Kotlin 的默认行为不同：注解拥有 RUNTIME 保留期。因此，JKid 中的注解没有显式地指定保留期。

10.1.6 使用类做注解参数

你已经见过了如何定义保存了作为其实参的静态数据的注解，但有时候你有不同的需求：能够引用类作为声明的元数据。可以通过声明一个拥有类引用作为形参的注解类来做到这一点。在 JKid 库中，这出现在 @DeserializeInterface 注解中，它允许你控制那些接口类型属性的反序列化。不能直接创建一个接口的实例，因此需要指定反序列化时哪个类作为实现被创建。

下面这个简单例子展示了这个注解如何使用：

```
interface Company {
    val name: String
}

data class CompanyImpl(override val name: String) : Company

data class Person(
    val name: String,
    @DeserializeInterface(CompanyImpl::class) val company: Company
)
```

当JKid读到一个Person类实例嵌套的company对象时，它创建并反序列化了一个CompanyImpl的实例，把它存储在company属性中。使用CompanyImpl::class作为@DeserializeInterface注解的实参来说明这一点。通常，使用类名称后面跟上::class关键字来引用一个类。

现在我们看看这个注解是如何声明的。它的单个实参是一个类引用，就像@DeserializeInterface(CompanyImpl::class)：

```
annotation class DeserializeInterface(val targetClass: KClass<out Any>)
```

KClass是Java的java.lang.Class类型在Kotlin中的对应类型。它用来保存Kotlin类的引用。在本章后面的“反射”一节中你将看到它能让你对这些类做些什么。

KClass的类型参数说明了这个引用可以指向哪些Kotlin类。例如，CompanyImpl::class的类型是KClass<CompanyImpl>，它是这个注解形参类型的子类型，如图10.4所示。

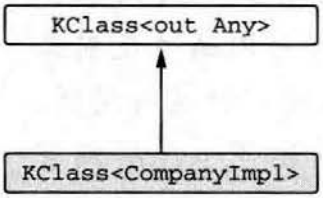


图10.4 注解实参类型CompanyImpl::class（KClass<CompanyImpl>）是注解形参类型（KClass<out Any>）的子类型

如果你只写出KClass<Any>而没有用out修饰符，就不能传递CompanyImpl::class作为实参：唯一允许的实参将是Any::class。out关键字说明允许引用那些继承Any的类，而不仅仅是引用Any自己。下一小节还会展示另一个这样的注解，它接收指向泛型类的引用作为形参。

10.1.7 使用泛型类做注解参数

默认情况下，JKid把非基本数据类型的属性当成嵌套的对象序列化。但是你可以改变这种行为并为某些值提供你自己的序列化逻辑。

@CustomSerializer注解接收一个自定义序列化器类的引用作为实参。这个序列化器类应该实现ValueSerializer接口：

```
interface ValueSerializer<T> {
    fun toJsonValue(value: T): Any?
    fun fromJsonValue(jsonValue: Any?): T
}
```

假设你需要支持序列化日期，而且已经为此创建了你自己的 DateSerializer 类，它实现了 ValueSerializer<Date> 接口（这个类是 JKid 源代码中的一个例子：http://mng.bz/73a7）。下面将展示如何在 Person 类上应用它：

```
data class Person(
    val name: String,
    @CustomSerializer(DateSerializer::class) val birthDate: Date
)
```

现在我们看看 @CustomSerializer 注解是如何声明的。ValueSerializer 类是泛型的而且定义了一个类型形参，所以在你引用该类型的时候需要提供一个类型实参值。因为你不知道任何关于那些应用了这个注解的属性类型的信息，可以使用星号投射（9.3.6 节中讨论过）作为（类型）实参：

```
annotation class CustomSerializer(
    val serializerClass: KClass<out ValueSerializer<*>>
)
```

图 10.5 审视了 serializerClass 参数的类型并解释了其中不同的部分。你需要保证注解只能引用实现了 ValueSerializer 接口的类。例如，@CustomSerializer(Date::class) 的写法是不允许的，因为 Date 没有实现 ValueSerializer 接口。

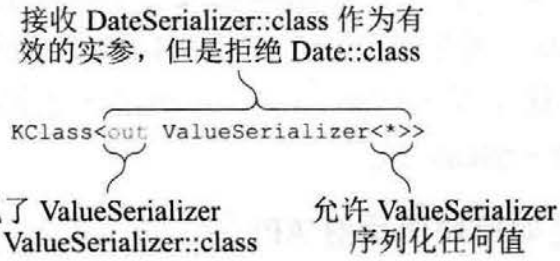


图 10.5 serializerClass 注解参数的类型。指向 ValueSerializer 实现类的类引用将会是有效的注解实参

是不是很麻烦？好消息是每一次需要使用类作为注解实参的时候都可以应用同样的模式。可以这样写 KClass<out YourClassName>，如果 YourClassName 有它自己的类型实参，就用 * 代替它们。

现在你已经了解了所有关于 Kotlin 中声明和应用注解的所有重要方面。下一步就要搞清楚如何访问存储在这些注解中的数据。你需要使用反射来做到这一点。

10.2 反射：在运行时对Kotlin对象进行自省

反射是，简单来说，一种在运行时动态地访问对象属性和方法的方式，而不需

要事先确定这些属性是什么。一般来说，当你访问一个对象的方法或者属性时，程序的源代码会引用一个具体的声明，编译器将静态地解析这个引用并确保这个声明是存在的。但有些时候，你需要编写能够使用任意类型的对象的代码，或者只能在运行时才能确定要访问的方法和属性的名称。JSON 序列化库就是这种代码绝好的例子：它要能够把任何对象都序列化成 JSON，所以它不能引用具体的类和属性。这时该反射大显身手了。

当在 Kotlin 中使用反射时，你会和两种不同的反射 API 打交道。第一种是标准的 Java 反射，定义在包 `java.lang.reflect` 中。因为 Kotlin 类会被编译成普通的 Java 字节码，Java 反射 API 可以完美地支持它们。实际上，这意味着使用了反射 API 的 Java 库完全兼容 Kotlin 代码。

第二种是 Kotlin 反射 API，定义在包 `kotlin.reflect` 中。它让你能访问那些在 Java 世界里不存在的概念，诸如属性和可空类型。但这一次它没有为 Java 反射 API 提供一个面面俱到的替身，而且不久你就会看到，有些情况下你仍然会回去使用 Java 反射。这里有个重要的提示，Kotlin 反射 API 没有仅限于 Kotlin 类：你能够使用同样的 API 访问用任何 JVM 语言写成的类。

> **注意** 在一些特别在意运行时库的大小的平台上，例如 Android，为了降低大小，Kotlin 反射 API 被打包成了单独的 .jar 文件，即 kotlin-reflect.jar。它不会被默认地添加到新项目的依赖中。如果你正在使用 Kotlin 反射 API，你要确保这个库作为依赖被添加（到了项目中）。IntelliJ IDEA 能够检测到缺失的依赖并协助你添加它。这个库的 Maven group/artifact 坐标是 `org.jetbrains.kotlin:kotlin-reflect`。

在这一节中，你将看到 JKid 是如何使用反射 API 的。首先我们会带你浏览序列化部分，因为我们能更直白、更容易地解释它，然后再继续进入 JSON 的解析和反序列化部分。但首先我们得仔细看看反射 API 的内容。

10.2.1 Kotlin 反射 API：KClass、KCallable、KFunction 和 KProperty

Kotlin 反射 API 的主要入口就是 `KClass`，它代表了一个类。`KClass` 对应的是 `java.lang.class`，可以用它列举和访问类中包含的所有声明，然后是它的超类中的声明，等等。`MyClass::class` 的写法会带给你一个 `KClass` 的实例。要在运行时取得一个对象的类，首先使用 `javaClass` 属性获得它的 Java 类，这直接等价于 Java 中的 `java.lang.Object.getClass()`。然后访问该类的 `.kotlin` 扩展属性，从 Java 切换到 Kotlin 的反射 API：

```
class Person(val name: String, val age: Int)

>>> val person = Person("Alice", 29)
>>> val kClass = person.javaClass.kotlin
>>> println(kClass.simpleName)
Person
>>> kClass.memberProperties.forEach { println(it.name) }
age
name
```

返回一个 KClass<Person> 的实例

这个简单的例子打印出了类的名称和它的属性的名称，并且使用 .memberProperties 来收集这个类，以及它的所有超类中定义的全部非扩展属性。

如果浏览一下 KClass 的声明，你会发现它包含大量方便的方法，用于访问类的内容：

```
interface KClass<T : Any> {
    val simpleName: String?
    val qualifiedName: String?
    val members: Collection<KCallable<*>>
    val constructors: Collection<KFunction<T>>
    val nestedClasses: Collection<KClass<*>>
    ...
}
```

KClass 的许多有用的特性，包括前面例子中用到的 memberProperties，都声明成了扩展。可以在标准库参考（http://mng.bz/em4i）中看到 KClass 的完整方法类列表（包括扩展）。

你可能已经发现了由类的所有成员组成的列表是一个 KCallable 实例的集合。KCallable 是函数和属性的超接口。它声明了 call 方法，允许你调用对应的函数或者对应属性的 getter：

```
interface KCallable<out R> {
    fun call(vararg args: Any?): R
    ...
}
```

你把（被引用）函数的实参放在 varargs 列表中提供给它。下面的代码展示了如何通过反射使用 call 来调用一个函数：

```
fun foo(x: Int) = println(x)
>>> val kFunction = ::foo
>>> kFunction.call(42)
42
```

在 5.1.5 节中你见过 ::foo 的语法，现在你可以发现这个表达式的值是来自

反射 API 的 KFunction 类的一个实例。你会使用 KCallable.call 方法来调用被引用的函数。这个例子中，你需要提供一个单独的实参，42。如果你用错误数量的实参去调用函数，比如 kFunction.call()，这将会抛出一个运行时异常："IllegalArgumentException: Callable expects 1 arguments, but 0 were provided."（IllegalArgumentException: Callable 期望的是 1 个参数，但只提供了 0 个）。

然而，这种情况下，你可以用一个更具体的方法来调用这个函数。::foo 表达式的类型是 KFunction1<Int, Unit>，它包含了形参类型和返回类型的信息。1 表示这个函数接收一个形参。你使用 invoke 方法通过这个接口来调用函数。它接收固定数量的实参（这个例子中是一个），而且这些实参的类型对应着 KFunction1 接口的类型形参。你也可以直接调用 kFunction[1]：

```
import kotlin.reflect.KFunction2

fun sum(x: Int, y: Int) = x + y
>>> val kFunction: KFunction2<Int, Int, Int> = ::sum
>>> println(kFunction.invoke(1, 2) + kFunction(3, 4))
10
>>> kFunction(1)
ERROR: No value passed for parameter p2
```

现在你无法用数量不正确的实参去调用 kFunction 的 invoke 方法：这连编译都不能通过。因此，如果你有这样一个具体类型的 KFunction，它的形参类型和返回类型是确定的，那么应该优先使用这个具体类型的 invoke 方法。call 方法是对所有类型都有效的通用手段，但是它不提供类型安全性。

KFunctionN 接口是如何定义的，又是在哪里定义的？

像 KFunction1 这样的类型代表了不同数量参数的函数。每一个类型都继承了 KFunction 并加上一个额外的成员 invoke，它拥有数量刚好的参数。例如，KFunction2 声明了 operator fun invoke(p1: P1, p2: P2): R，其中 P1 和 P2 代表着函数的参数类型，而 R 代表着函数的返回类型。

这些类型称为**合成的编译器生成类型**，你不会在包 kotlin.reflect 中找到它们的声明。这意味着你可以使用任意数量参数的函数接口。合成类型的方式减小了 kotlin-reflect.jar 的尺寸，同时避免了对函数类型参数数量的人为限制。

你也可以在一个 KProperty 实例上调用 call 方法，它会调用该属性的 getter。但是属性接口为你提供了一个更好的获取属性值的方式：get 方法。

要访问 get 方法，你需要根据属性声明的方式来使用正确的属性接口。顶层属

1 11.3节将会详细解释为什么不用显式的invoke就可以调用kFunction。

性表示为 KProperty0 接口的实例，它有一个无参数的 get 方法：

```
var counter = 0
>>> val kProperty = ::counter
>>> kProperty.setter.call(21)
>>> println(kProperty.get())
21
```

通过反射调用 setter，把 21 作为实参传递

通过调用“get”获取属性的值

一个成员属性由 KProperty1 的实例表示，它拥有一个单参数的 get 方法。要访问该属性的值，必须提供你需要的值所属的那个对象实例。下面这个例子在 memberProperty 变量中存储了一个指向属性的引用；然后调用 memberProperty.get(person) 来获取属于具体 person 实例的这个属性的值。所以，如果 memberProperty 指向了 Person 类的 age 属性，memberProperty.get(person) 就是动态获取 person.age 的值的一种方式：

```
class Person(val name: String, val age: Int)

>>> val person = Person("Alice", 29)
>>> val memberProperty = Person::age
>>> println(memberProperty.get(person))
29
```

注意，KProperty1 是一个泛型类。变量 memberProperty 的类型是 KProperty<Person, Int>，其中第一个类型参数表示接收者的类型，而第二个类型参数代表了属性的类型。这样你只能对正确类型的接收者调用它的 get 方法；而 memberProperty.get("Alice") 这样的调用不会通过编译。

还有一点值得注意，只能使用反射访问定义在最外层或者类中的属性，而不能访问函数的局部变量。如果你定义了一个局部变量 x 并试图使用 ::x 来获得它的引用，你会得到一个编译期的错误：“References to variables aren’t supported yet”（现在还不支持对变量的引用）。

图 10.6 展示了运行时你可以用来访问源码元素的接口的层级结构。因为所有的声明都能被注解，所以代表运行时声明的接口，比如 KClass、KFunction 和 KParameter，全部继承了 KAnnotatedElement。KClass 既可以用来表示类也可以表示对象。KProperty 可以表示任何属性，而它的子类 KMutableProperty 表示一个用 var 声明的可变属性。可以使用声明在 KProperty 和 KMutableProperty 中的特殊接口 Getter 和 Setter，把属性的访问器当成函数使用——例如，如果你需要取回它们的注解。两个访问器的接口都继承了 KFunction。简单起见，图中我们省略了像 KProperty0 这样的具体的属性接口。

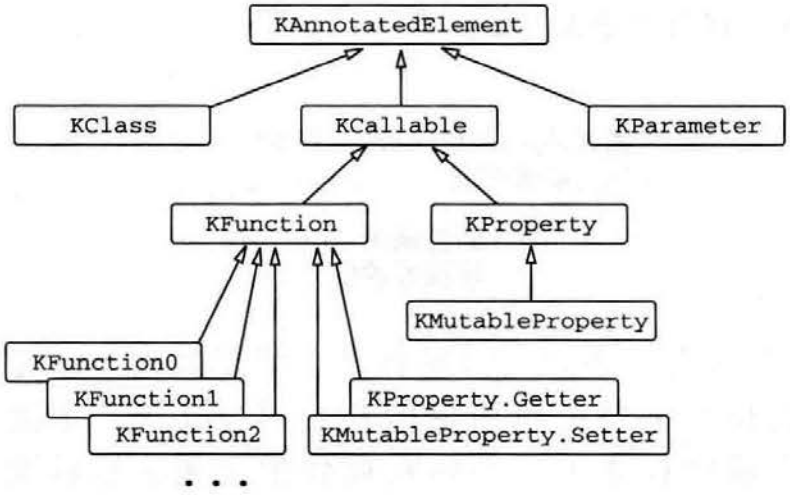


图 10.6 Kotlin 反射 API 中的接口层级结构

现在你已经熟悉了 Kotlin 反射 API 的基础，接下来我们研究一下 JKid 库是如何实现的。

10.2.2 用反射实现对象序列化

首先，我们回忆一下 JKid 中序列化函数的声明：

```
fun serialize(obj: Any): String
```

这个函数接收一个对象然后返回 JSON 表示法的字符串。它通过一个 StringBuilder 实例来构建 JSON 结果。这个函数在序列化对象属性和它们的值的同时，这些内容会被附加到这个 StringBuilder 对象之中。我们把实现放在 StringBuilder 的扩展函数中，好让 append 的调用更加简洁。这样，你不用限定符就可以方便地调用 append 方法：

```
private fun StringBuilder.serializeObject(x: Any) {
    append(...)
}
```

把一个函数参数转化成一个扩展函数的接收者是 Kotlin 代码中的常见模式，我们会在 11.2.1 节中详细讨论。注意 serializeObject 没有拓展 StringBuilder 的 API。它执行的操作在这个特殊的上下文之外毫无意义，所以它被标记成 private，以保证它不会在其他地方使用。它被声明成扩展以强调这个特殊对象是代码块的主要对象，让这个对象用起来更容易。

结果，serialize 函数把所有的工作委托给了 serializeObject：

```
fun serialize(obj: Any): String = buildString { serializeObject(obj) }
```

正如你在 5.5.2 节中看到的，buildString 会创建一个 StringBuilder，并

让你在 lambda 中填充它的内容。这个例子中，对 serializeObject(obj) 的调用提供了要填充的内容。

现在我们讨论一下序列化函数的行为。默认情况下，它将序列化对象的所有属性；基本数据类型和字符串将会被酌情序列化成 JSON 数值、布尔值和字符串值；集合将会被序列化成 JSON 数组；其他类型的属性将会被序列化成嵌套的对象。正如我们在上一节中讨论的，这种行为可以通过注解进行定制。

我们来看看 serializeObject 的实现，在这里可以在真实的场景中观察反射 API。

代码清单 10.1 序列化一个对象

```
private fun StringBuilder.serializeObject(obj: Any) {
    val kClass = obj.javaClass.kotlin
    val properties = kClass.memberProperties

    properties.joinToStringBuilder(
            this, prefix = "{", postfix = "}") { prop ->
        serializeString(prop.name)
        append(": ")
        serializePropertyValue(prop.get(obj))
    }
}
```

取得对象的 KClass

取得类的所有属性

取得属性名字

取得属性值

这个函数的实现应该很清晰：逐一序列化类的每一个属性。生成的 JSON 看起来会是这样：{prop1: value1, prop2: value2 }。joinToStringBuilder 函数保证属性与属性之间用逗号隔开。serializeString 函数按照 JSON 格式的要求对特殊的字符进行转义。serializeString 函数检查一个值是否是一个基本数据类型的值、字符串、集合或是嵌套对象，然后相应地序列化它的内容。

在前面的小节中，我们讨论过一种获取 KProperty 实例值的方式：get 方法。在那个例子中，你使用过类型为 KProperty1<Person,Int> 的成员引用 Person::age，它让编译器知道了接收者和属性值的确切类型。然而在这个例子中，确切类型是未知的，因为你列举了一个对象的类中的所有属性。因此，prop 变量拥有类型 KProperty1<Any, *>,，而 prop.get(obj) 返回一个 Any 类型的值。你不会得到任何针对接收者的编译期检查，但是因为你传递的对象和获取属性列表的对象是同一个，接收者的类型是不会错的。接下来，我们看看用来调整序列化的注解是如何实现的。

10.2.3 用注解定制序列化

在本章的前面部分，你见过了让你定制 JSON 序列化过程的注解定义。实际上，

我们讨论过 @JsonExclude、@JsonName 和 @CustomSerializer 这几个注解。现在是时候看看 serializeObject 函数是如何处理这些注解的。

我们从 @JsonExclude 开始，这个注解允许你在序列化的时候排除某些属性。让我们研究一下应该如何修改 serializeObject 函数的实现来支持它。

回忆一下，你使用了 KClass 实例的扩展属性 memberProperties，来取得类的所有成员属性。但现在的任务变得更加复杂：使用 @JsonExclude 注解的属性需要被过滤掉。我们来看看如何做到这一点。

KAnnotatedElement 接口定义了属性 annotations，它是一个由应用到源码中元素上的所有注解（具有运行时保留期）的实例组成的集合。因为 KProperty 继承了 KAnnotatedElement，可以用 property.annotations 这样的写法来访问一个属性的所有注解。

但这里的过滤并不会用到所有的注解，它只需要找到那个特定的注解（@JsonExclude）。辅助函数 findAnnotation 完成了这项工作：

```
inline fun <reified T> KAnnotatedElement.findAnnotation(): T?
        = annotations.filterIsInstance<T>().firstOrNull()
```

findAnnotation 函数将返回一个注解，其类型就是指定为类型实参的类型，如果这个注解存在。它用到了 9.2.3 节中我们讨论过的模式，让类型形参变成 reified，以期把注解类作为类型实参传递。

现在可以把 findAnnotation 和标准库函数 filter 一起使用，过滤掉那些带 @JsonExclude 注解的属性：

```
val properties = kClass.memberProperties
        .filter { it.findAnnotation<JsonExclude>() == null }
```

下一个注解是 @JsonName。我们把它的实现和使用它的例子重新放在这里作为提示：

```
annotation class JsonName(val name: String)

data class Person(
    @JsonName("alias") val firstName: String,
    val age: Int
)
```

这种情况下，你关心的不仅是注解存不存在，还要关心它的实参：被注解的属性在 JSON 中应该用的名称。幸运的是，findAnnotation 函数可以帮上忙：

```
val jsonNameAnn = prop.findAnnotation<JsonName>()
val propName = jsonNameAnn?.name ?: prop.name
```

取得 @JsonName 注解的实例，如果它存在

取得它的“name”实参或者备用的“prop.name”

如果属性没有用 @JsonName 注解，jsonNameAnn 就是 null，而你仍然需要使用 prop.name 作为属性在 JSON 中的名称。如果属性用 @JsonName 注解了，你就会使用在注解中指定的名称而不是属性自己的名称。

我们来看一下先前声明的 Person 类的一个实例的序列化过程。在属性 firstName 序列化期间，jsonNameAnn 包含了注解类 JsonName 对应的实例。所以，jsonNameAnn?.name 返回了非空的值 "alias"，将会用作 JSON 中的键。当属性 age 序列化时，没有找到这个注解，所以属性名称 age 被用作 JSON 中的键。

我们把目前为止所有讨论过的修改组合到一起，看看组合后形成的序列化逻辑的实现。

代码清单 10.2 使用属性过滤序列化对象

```
private fun StringBuilder.serializeObject(obj: Any) {
    obj.javaClass.kotlin.memberProperties
            .filter { it.findAnnotation<JsonExclude>() == null }
            .joinToStringBuilder(this, prefix = "{", postfix = "}") {
                serializeProperty(it, obj)
            }
}
```

现在用 @JsonExclude 注解的属性被过滤掉了。我们还把负责属性序列化的逻辑抽取到了一个单独的 serializeProperty 函数。

代码清单 10.3 序列化单个属性

```
private fun StringBuilder.serializeProperty(
        prop: KProperty1<Any, *>, obj: Any
) {
    val jsonNameAnn = prop.findAnnotation<JsonName>()
    val propName = jsonNameAnn?.name ?: prop.name
    serializeString(propName)
    append(": ")

    serializePropertyValue(prop.get(obj))
}
```

属性的名称根据之前讨论的 @JsonName 注解进行处理。

下一步，我们来实现剩下的注解 @CustomSerializer。它的实现基于 getSerializer 函数，该函数返回通过 @CustomSerializer 注解注册的 ValueSerializer 实例。例如，如果像下面展示的这样声明 Person 类，并在序列化 birthDate 属性的时候调用 getSerializer()，它会返回一个 DateSerializer 的实例：

```
data class Person(
    val name: String,
    @CustomSerializer(DateSerializer::class) val birthDate: Date
)
```

这里提示一下 @CustomSerializer 注解是如何声明的，帮助你更好地理解 getSerializer 的实现：

```
annotation class CustomSerializer(
    val serializerClass: KClass<out ValueSerializer<*>>
)
```

下面就是如何实现 getSerializer 函数。

代码清单 10.4 取回属性值的序列化器

```
fun KProperty<*>.getSerializer(): ValueSerializer<Any?>? {
    val customSerializerAnn = findAnnotation<CustomSerializer>() ?: return null
    val serializerClass = customSerializerAnn.serializerClass

    val valueSerializer = serializerClass.objectInstance
            ?: serializerClass.createInstance()
    @Suppress("UNCHECKED_CAST")
    return valueSerializer as ValueSerializer<Any?>
}
```

它是 KProperty 的扩展函数，因为属性是这个方法要处理的主要对象（接收者）。它调用了 findAnnotation 函数取得一个 @CustomSerializer 注解的实例，如果实例存在。它的实参 serializerClass 指定了你需要获取哪个类的实例。

处理作为 @CustomSerializer 注解的值的类和对象（Kotlin 的单例）的方式，是这里最有趣的部分。它们都用 KClass 类表示。不同的是，对象拥有非空值的 objectInstance 属性，可以用它来访问为 object 创建的单例实例。例如，DateSerializer 被声明成了一个 object，所以它的 objectInstance 属性存储了 DateSerializer 的单例实例。你将用这个实例序列化所用对象，而不会调用 createInstance。

如果 KClass 表示的是一个普通的类，可以通过调用 createInstance 来创建一个新的实例。这个函数和 java.lang.Class.newInstance 类似。

最终，你可以在 serializeProperty 的实现中用上 getSerializer。下面是这个函数的最终版本。

代码清单 10.5 序列化属性，支持自定义序列化器

```
private fun StringBuilder.serializeProperty(
        prop: KProperty1<Any, *>, obj: Any
```

```
) {
    val name = prop.findAnnotation<JsonName>()?.name ?: prop.name
    serializeString(name)
    append(": ")

    val value = prop.get(obj)
    val jsonValue =
            prop.getSerializer()?.toJsonValue(value)
            ?: value
    serializePropertyValue(jsonValue)
}
```

如果自定义序列化器存在就为属性使用它

否则像之前那样使用属性值

serializeProperty 通过调用序列化器的 toJsonValue，来把属性值转换成 JSON 兼容的格式。如果属性没有自定义序列化器，它就使用属性的值。

现在你已经见过了这个库 JSON 序列化部分的实现，我们的话题将转到解析和反序列化。反序列化部分需要更多的代码，所以我们不会审查所有的代码，但会看到实现的结构，并解释反射是如何用来反序列化对象的。

10.2.4 JSON 解析和对象反序列化

让我们从故事的第二部分开始：实现反序列化的逻辑。首先，回忆一下 API，它和序列化用到的 API 相似，包含一个单独的函数：

```
inline fun <reified T: Any> deserialize(json: String): T
```

这里有一个使用它的例子：

```
data class Author(val name: String)
data class Book(val title: String, val author: Author)

>>> val json = """{"title": "Catch-22", "author": {"name": "J. Heller"}}"""
>>> val book = deserialize<Book>(json)
>>> println(book)
Book(title=Catch-22, author=Author(name=J. Heller))
```

你把要被反序列化的对象的类型作为实化类型参数传给 deserialize 函数并拿回一个新的对象实例。

JSON 反序列化是比序列化更困难的任务，因为它涉及解析 JSON 字符串输入，还有使用反射访问对象的内部细节。JKid 中的 JSON 反序列化器使用相当普通的方式实现，由三个主要阶段组成：词法分析器（通常被称为 *lexer*）、语法分析器或解析器，以及反序列化组件本身。

词法分析把由字符组成的输入字符串切分成一个由标记组成的列表。这里有两类标记：代表 JSON 语法中具有特殊意义的字符（逗号、冒号、花括号和方括号）的字符标记；对应到字符串、数字、布尔值以及 null 常量的值标记。左花括号（{）、

字符串值（"Catch-22"）和整数值（42）是不同标记的例子。

解析器通常负责将无格式的标记列表转换为结构化的表示法。它在 JKid 中的任务是理解 JSON 的更高级别的结构，并将各个标记转换为 JSON 中支持的语义元素：键值对、对象和数组。

JsonObject 接口跟踪当前正在被反序列化的对象或数组。解析器在发现当前对象的新属性（简单值、复合属性或数组）时调用相应的方法。

代码清单 10.6 JSON 解析器回调接口

```
interface JsonObject {
    fun setSimpleProperty(propertyName: String, value: Any?)

    fun createObject(propertyName: String): JsonObject

    fun createArray(propertyName: String): JsonObject
}
```

这些方法中的参数 propertyName 接收到了 JSON 键。因此，当解析器遇到一个使用对象作为值的 author 属性时，createObject("author") 方法会被调用。简单值属性被报告为 setSimpleProperty 调用，实际的标记值作为 value 实参传递给这次调用。JsonObject 实现负责创建属性的新对象，并在外部对象中存储对它们的引用。

图 10.7 展示了反序列化一个样本字符串时，词法和语法分析每一阶段的输入和输出。再一次，词法分析将输入字符串切分成标记列表，然后句法分析（解析器）处理这个标记列表，并在每个有意义的新元素上调用 JsonObject 中适当的方法。

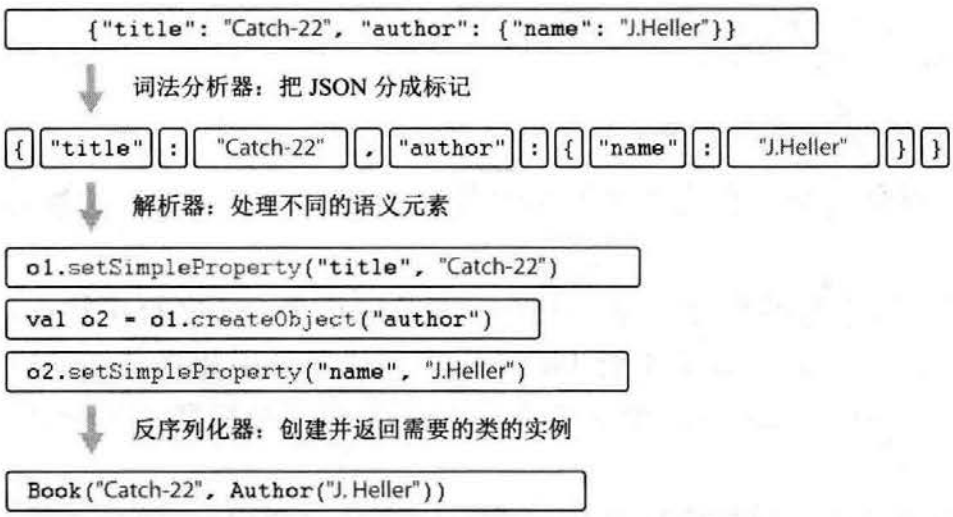


图 10.7 JSON 解析：词法分析器、解析器和反序列化器

然后，反序列化器为 JsonObject 提供一种实现，逐步构建相应类型的新实例。

它需要找到类属性和 JSON 键（图 10.7 中的 title、author 和 name）之间的对应关系，并构建嵌套对象的值（Author 的实例）。在这之后，它才可以创建一个最终需要的类的新实例（Book）。

JKid 库打算使用数据类，因此，它将从 JSON 文件加载的所有名称 - 值的配对作为参数传递给要被反序列化的类的构造方法。它不支持在对象实例创建后设置其属性。这意味着从 JSON 中读取数据时它需要将数据存储在某处，然后才能构建该对象。

在创建对象之前保存其组件的要求看起来与传统的构建器模式相似，区别在于构建器通常用于创建一种特定类型的对象，并且解决方案需要完全通用。我们在这个实现中使用了一个有趣的词语种子（*Seed*）。在 JSON 中，你需要构建不同类型的复合结构：对象、集合和 map。ObjectSeed、ObjectListSeed 和 ValueListSeed 类负责构建适当的对象、复合对象列表，以及简单值的列表。而 map 的构造就作为练习留给你了。

基本的 Seed 接口继承了 JsonObject，并在构建过程完成后提供了一个额外的 spawn 方法来获取生成的实例。它还声明了用于创建嵌套对象和嵌套列表的 createCompositeProperty 方法（它们使用相同的底层逻辑通过种子来创建实例）。

代码清单 10.7 从 JSON 数据创建对象的接口

```
interface Seed: JsonObject {
    fun spawn(): Any?

    fun createCompositeProperty(
        propertyName: String,
        isList: Boolean
    ): JsonObject

    override fun createObject(propertyName: String) =
        createCompositeProperty(propertyName, false)
    override fun createArray(propertyName: String) =
        createCompositeProperty(propertyName, true)

    // ...
}
```

你可以认为 spawn 就是返回结果值的 build 方法的翻版。它返回的是为 ObjectSeed 构造的对象，以及为 ObjectListSeed 或 ValueListSeed 生成的列表。我们不会详细讨论列表是如何反序列化的。我们将注意力集中于创建对象，它更复杂并有助于展示通用的思路。

但在此之前，我们先来研究一下 deserialize 的主要功能，它完成反序列化

一个值的所有工作。

代码清单 10.8 顶层反序列化函数

```
fun <T: Any> deserialize(json: Reader, targetClass: KClass<T>): T {
    val seed = ObjectSeed(targetClass, ClassInfoCache())
    Parser(json, seed).parse()
    return seed.spawn()
}
```

整个解析过程是这样的，一开始你会创建一个 ObjectSeed 来存储反序列化对象的属性，然后调用解析器并将输入字符流 json 传递给它。当达到输入数据的结尾时，你就可以调用 spawn 函数来构建最终对象。

现在我们聚焦 ObjectSeed 的实现，它存储了正在构造的对象的状态。ObjectSeed 接收了一个目标类的引用和一个 classInfoCache 对象，该对象包含缓存起来的关于该类属性的信息。这些缓存起来的信息稍后将被用于创建该类的实例。ClassInfoCache 和 ClassInfo 是我们将在下一节讨论的辅助类。

代码清单 10.9 反序列化一个对象

```
class ObjectSeed<out T: Any>(
        targetClass: KClass<T>,
        val classInfoCache: ClassInfoCache
) : Seed {

    private val classInfo: ClassInfo<T> =
            classInfoCache[targetClass]

    private val valueArguments = mutableMapOf<KParameter, Any?>()
    private val seedArguments = mutableMapOf<KParameter, Seed>()

    private val arguments: Map<KParameter, Any?>
        get() = valueArguments +
                seedArguments.mapValues { it.value.spawn() }

    override fun setSimpleProperty(propertyName: String, value: Any?) {
        val param = classInfo.getConstructorParameter(propertyName)
        valueArguments[param] =
                classInfo.deserializeConstructorArgument(param, value)
    }

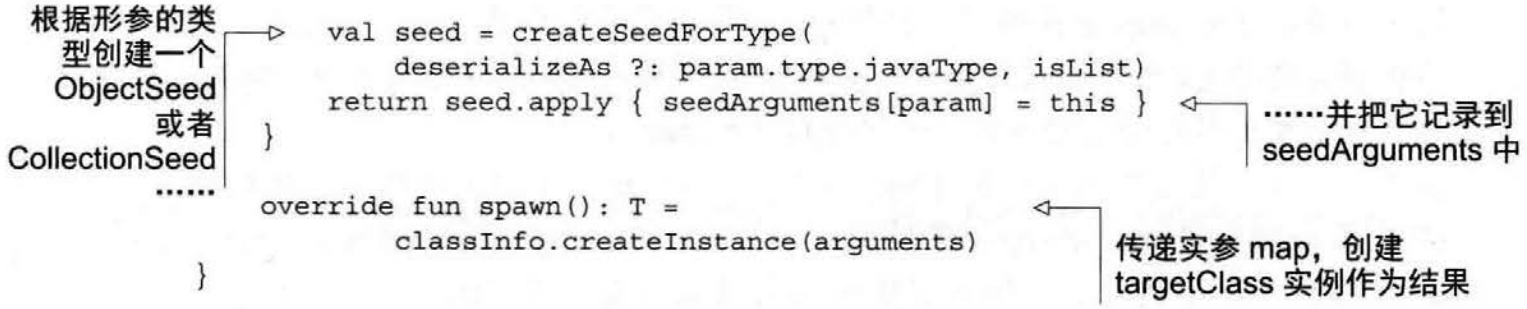
    override fun createCompositeProperty(
            propertyName: String, isList: Boolean
    ): Seed {
        val param = classInfo.getConstructorParameter(propertyName)
        val deserializeAs =
            classInfo.getDeserializeClass(propertyName)
```

缓存需要创建 targetClass 实例的信息

构建一个从构造方法参数到它们的值的映射

如果一个构造方法参数的值是简单值，把它记录下来

如果有的话加载属性 DeserializeInterface 注解的值



ObjectSeed 构建了一个构造方法形参和它们的值之间的映射。这用到了两个可变的 map：给简单值用的 valueArguments 和给复合属性用的 seedArguments。当结果开始构建时，新的实参通过 setSimpleProperty 调用被添加到 valueArguments，通过 createCompositeProperty 调用被添加到 seedArguments。新的复合种子被添加时状态是空的，然后被来自输入流的数据填充。最终，spawn 方法递归地调用每个种子的 spawn 方法来构建所有嵌套的种子。

注意，spawn 的方法体中 arguments 调用是怎样启动递归的复合（种子）实参的构建过程的：arguments 自定义的 getter 调用 seedArguments 中每一个元素的 spawn 方法。createSeedForType 函数分析形参的类型并根据形参是哪种集合来创建 ObjectSeed、ObjectListSeed 或者 ValueListSeed。我们把它实现的剩下部分交给你自己去研究。接下来，我们看看 ClassInfo.createInstance 函数是如何创建 targetClass 的实例的。

10.2.5 反序列化的最后一步：callBy() 和使用反射创建对象

最后一部分你要理解的就是 ClassInfo 类，它创建了作为结果的实例，还缓存了关于构造方法参数的信息。ObjectSeed 用到了它。但在我们一头扎进实现细节之前，我们先看看通过反射来创建对象的 API。

你已经见过了 KCallable.call 方法，它调用函数或者构造方法，并接收一个实参组成的列表。这个方法很多情况下都很好用，但它有一个限制：不支持默认参数值。这种情况下，如果用户试图用带默认参数值的构造方法来反序列化一个对象，绝对不想这些实参还需要在 JSON 中说明。因此，你需要使用另外一个支持默认参数值的方法：KCallable.callBy。

```
interface KCallable<out R> {
    fun callBy(args: Map<KParameter, Any?>): R
    ...
}
```

这个方法接收一个形参和它们对应值之间的 map，这个 map 将被作为参数传给

这个方法。如果 map 中缺少了一个形参，可行的话它的默认值将会被使用。还有一点特别方便的是，你不必按照顺序来写入形参；可以从 JSON 中读取名称 - 值的配对，找到每个实参名称对应的形参，把它的值写入 map 中。

有一点需要注意的是取得正确的类型。args map 中值的类型需要跟构造方法的参数类型相匹配，否则你将得到一个 IllegalArgumentException。这对算术类型来说特别重要：你需要知道参数接收的是一个 Int、一个 Long、一个 Double，还是一个其他的基本数据类型，并把来自 JSON 的算术值转换成正确的类型。可以使用 KParameter.type 属性来做到这一点。

这里的类型转换是通过 ValueSerializer 接口完成的，这个接口和定制序列化时用的 ValueSerializer 接口是同一个。如果属性没有 @CustomSerializer 注解，你会根据它的类型获取标准的实现。

代码清单 10.10 根据值类型取得序列化器

```
fun serializerForType(type: Type): ValueSerializer<out Any?>? =
        when(type) {
            Byte::class.java -> ByteSerializer
            Int::class.java -> IntSerializer
            Boolean::class.java -> BooleanSerializer
            // ...
            else -> null
        }
```

对应的 ValueSerializer 实现会执行必要的类型检查及转换。

代码清单 10.11 Boolean 值的序列化器

```
object BooleanSerializer : ValueSerializer<Boolean> {
    override fun fromJsonValue(jsonValue: Any?): Boolean {
        if (jsonValue !is Boolean) throw JKidException("Boolean expected")
        return jsonValue
    }

    override fun toJsonValue(value: Boolean) = value
}
```

callBy 方法给了你一种调用一个对象的主构造方法的方式，需要传给它一个形参和对应值之间的 map。ValueSerializer 机制保证了 map 中的值拥有正确的类型。现在我们看看如何调用这个 API。

ClassInfoCache 旨在减少反射操作的开销。回忆一下用来控制序列化和反序列化过程的注解（@JsonName 和 @CustomSerializer），它们是用在了属性上，而不是形参上。当你反序列化一个对象时，你打交道的是构造方法参数，而不是属性；

要获取注解，你需要先找到对应的属性。在读取每个（JSON）键值对的时候都执行一次这样的搜索将会极其缓慢，所以每个类只会做一次这样的搜索并且把信息缓存起来。下面是 ClassInfoCache 的完整实现。

代码清单 10.12 缓存的反射数据的存储

```
class ClassInfoCache {
    private val cacheData = mutableMapOf<KClass<*>, ClassInfo<*>>()

    @Suppress("UNCHECKED_CAST")
    operator fun <T : Any> get(cls: KClass<T>): ClassInfo<T> =
            cacheData.getOrPut(cls) { ClassInfo(cls) } as ClassInfo<T>
}
```

这里使用了在 9.3.6 节中我们讨论过的模式：在 map 中存储值的时候去掉类型信息，但 get 方法的实现保证了返回的 ClassInfo<T> 拥有正确的类型实现。注意 getOrPut 的用法：如果 mapcacheData 已经包含了一个 cls 的值，你就返回这个值。否则，调用传递进来的 lambda，它会计算出这个键对应的值并存储到 map 中，然后返回它。

ClassInfo 类负责按目标类创建新实例并缓存必要的信息。为了简化代码，我们省略了一些函数和默认初始化器的代码。还有，你可能注意到生产代码会抛出一个带有丰富信息的异常（这也是你的代码应该采用的良好模式），来代替这里的 !!。

代码清单 10.13 构造方法的参数及注解数据的缓存

```
class ClassInfo<T : Any>(cls: KClass<T>) {
    private val constructor = cls.primaryConstructor!!

    private val jsonNameToParamMap = hashMapOf<String, KParameter>()
    private val paramToSerializerMap =
        hashMapOf<KParameter, ValueSerializer<out Any?>>()
    private val jsonNameToDeserializeClassMap =
        hashMapOf<String, Class<out Any>?>()

    init {
        constructor.parameters.forEach { cacheDataForParameter(cls, it) }
    }
    fun getConstructorParameter(propertyName: String): KParameter =
            jsonNameToParam[propertyName]!!

    fun deserializeConstructorArgument(
            param: KParameter, value: Any?): Any? {
        val serializer = paramToSerializer[param]
        if (serializer != null) return serializer.fromJsonValue(value)

        validateArgumentType(param, value)
```

```
        return value
    }

    fun createInstance(arguments: Map<KParameter, Any?>): T {
        ensureAllParametersPresent(arguments)
        return constructor.callBy(arguments)
    }

    // ...
}
```

在初始化时，这段代码找到了每个构造方法参数对应的属性并取回了它们的注解。它把这些数据存储在三个 map 中：jsonNameToParamMap 说明了 JSON 文件中的每个键对应的形参，paramToSerializerMap 存储了每个形参对应的序列化器，还有 jsonNameToDeserializeClassMap 存储了指定为 @DeserializeInterface 注解的实参的类，如果有的话。然后 ClassInfo 就能根据属性名称提供构造方法的形参，并调用使用形参的代码，这些代码中这个形参将作为形参和实参之间 map 的键使用。

cacheDataForParameter、validateArgumentType 和 ensureAllParametersPresent 是这个类的私有函数。下面是 ensureAllParametersPresent 的实现，可以自己浏览其他函数的代码。

代码清单 10.14 验证需要的参数被提供了

```
private fun ensureAllParametersPresent(arguments: Map<KParameter, Any?>) {
    for (param in constructor.parameters) {
        if (arguments[param] == null &&
                !param.isOptional && !param.type.isMarkedNullable) {
            throw JKidException("Missing value for parameter ${param.name}")
        }
    }
}
```

这个函数检查你是不是提供了全部需要的参数的值。注意这里反射 API 是如何帮助你的。如果一个参数有默认值，那么 param.isOptional 是 true，你就可以为它省略一个实参；反之，默认值就会被使用。如果一个参数类型是可空的（param.type.isMarkedNullable 会告知你这一点），null 将会被作为默认的参数值使用。对所有的形参来说，你都必须提供对应的实参；否则就会抛出异常。反射缓存保证了只会搜索一次那些定制反序列化过程的注解，而不会为 JSON 数据中出现的每一个属性都执行搜索。

我们关于 JKid 库实现的讨论到此为止。在本章的课程中，我们探索了 JSON 序列化和反序列化库的实现，它基于反射 API 并使用了注解来定制其行为。当然，本

章展示的所有技术和方法都能在你自己的框架中使用。

10.3 小结

- Kotlin 中应用注解的语法和 Java 几乎一模一样。
- 在 Kotlin 中可以让你应用注解的目标的范围比 Java 更广，其中包括了文件和表达式。
- 一个注解的参数可以是一个基本数据类型、一个字符串、一个枚举、一个类引用、一个其他注解类的实例，或者前面这些元素组成的数组。
- 如果单个 Kotlin 声明产生了多个字节码元素，像 `@get:Rule` 这样指定一个注解的使用点目标，允许你选择注解如何应用。
- 注解类的声明是这样的，它是一个拥有主构造方法且没有类主体的类，其构造方法中所有参数都被标记成 `val` 属性。
- 元注解可以用来指定（使用点）目标、保留期模式和其他注解的特性。
- 反射 API 让你在运行时动态地列举和访问一个对象的方法和属性。它拥有许多接口来表示不同种类的声明，例如类（`KClass`）、函数（`KFunction`）等。
- 要获取一个 `KClass` 的实例，如果类是静态已知的，可以使用 `ClassName::class`；否则，使用 `obj.javaClass.kotlin` 从对象实例上取得类。
- `KFunction` 接口和 `KProperty` 接口都继承了 `KCallable`，它提供了一个通用的 `call` 方法。
- `KCallable.callBy` 方法能用来调用带默认参数值的方法。
- `KFunction0`、`KFunction1` 等这种不同参数数量的函数可以使用 `invoke` 方法调用。
- `KProperty0` 和 `KProperty1` 是接收者数量不同的属性，支持用 `get` 方法取回值。`KMutableProperty0` 和 `KMutableProperty1` 继承了这些接口，支持通过 `set` 方法来改变属性的值。

11 DSL构建

本章内容包括

- 构建领域特定语言
- 使用带接收者的 lambda
- 应用 invoke 约定
- 已有的 Kotlin DSL 示例

在这一章，我们会讨论怎样使用领域特定语言（DSL）为你的 Kotlin 类设计更有表现力、更符合语言习惯的 API。我们会对比传统 API 和 DSL 风格的 API 的不同，你也将会看到 DSL 风格的 API 被广泛地应用于各个领域的许多不同的实际问题当中，诸如数据库访问、HTML 生成、测试、编写编译脚本、定义 Android UI 布局，等等。

Kotlin DSL 设计依赖于许多语言特性，其中的两个我们还没有完整地探讨过。一个在第 5 章有过简单的介绍：带接收者的 lambda，它可以让你通过改变代码块中的命名解析规则来创建一个 DSL 结构。另外一个是全新的：invoke 约定，它使得 DSL 代码中 lambda 和属性赋值的结合更加灵活。我们将在这一章详细地学习这些特性。

11.1 从API到DSL

在我们深入讨论 DSL 之前，让我们更好地了解想要解决的问题。归根结底，我

们的目标是尽可能地让代码具有最佳的可读性和可维护性。要想达到这个目标，只关注单独的类是不够的。类中的大部分代码都是会与其他类交互的，所以我们需要关注交互发生的接口——换句话说就是类的 API。

重要的是要记住，构建良好 API 的挑战并不是留给库作者的；相反，这是每个开发者必须要做的。就像库会提供编程接口供使用一样，应用程序中的每个类都提供了其他类与之交互的可能性。确保这些交互易于理解并可以清楚地表达，对保持项目的可维护性至关重要。

在本书的课程中，你已经看到了许多可以让你为类构建整洁 API 的 Kotlin 功能示例。当我们说一个 API 是整洁的时候我们指的又是什么呢？其实就是两点：

- 它需要能够让读者清楚地知道在代码中发生了什么。这可以通过选择良好的名称和概念来做到，这对任何语言来说都很重要。
- 代码需要看起来整洁，极少使用浮夸的代码且不存在多余的语法。这一章就主要聚焦在如何达到这个目标上。整洁的 API 甚至很难从语言的内建功能中区分出来。

Kotlin 允许你构建整洁 API 的功能的代表包括：扩展函数、中缀调用、lambda 简明语法和运算符重载。表 11.1 展示了这些功能如何帮助减少代码中的语法噪声。

表 11.1 Kotlin 对整洁语法的支持

常规语法	整洁语法	用到的功能
`StringUtil.capitalize(s)`	`s.capitalize()`	扩展函数
`1.to("one")`	`1 to "one"`	中缀调用
`set.add(2)`	`set += 1`	运算符重载
`map.get("key")`	`map["key"]`	get 方法的约定
`file.use({ f -> f.read() } )`	`file.use { it.read() }`	括号外的 lambda
`sb.append("yes") sb.append("no")`	`with (sb) {append("yes") append("no") }`	带接收者的 lambda

在本章，我们将会在整洁 API 上更进一步，来看看 Kotlin 对于构建 DSL 的支持。Kotlin 的 DSL 建立在那些整洁语法的特性上，利用由多个方法调用创建出结构的能力来扩展它们。因此，DSL 比由单独方法构造出的 API 更具表现力并且更适宜工作。

Kotlin 的 DSL 是完全静态类型的，就像语言的其他功能一样。这意味着静态类型的所有优势，比如编译时错误的检测，以及更好的 IDE 支持，在你的 API 上使用 DSL 模式时仍然生效。

我们来一次快速体验，这里有几个例子展示了 Kotlin 的 DSL 能做什么。这个表达式会在时间上回溯并返回前一天（好啦，就仅仅是昨天的日期）：

```
val yesterday = 1.days.ago
```

而下面这个函数用来生成一个 HTML 表格：

```
fun createSimpleTable() = createHTML().
    table {
        tr {
            td { +"cell" }
        }
    }
```

通过本章的课程，你将学会这些例子是如何构建的。但在开始更深入的讨论之前，让我们来看看什么是 DSL。

11.1.1 领域特定语言的概念

DSL 的总体理念几乎和编程语言想法存在的时间一样长。我们将*通用编程语言*（有一系列足够完善的能力来解决几乎所有能被计算机解决的问题）与*领域特定语言*（专注在特定任务，或者说领域上，并放弃与该领域无关的功能）区分开来。

毫无疑问你熟悉的最常见的 DSL 就是 SQL 和正则表达式。它们分别很好地解决了操作数据库和文本字符串的特定任务，但是你不能用它们来开发整个应用程序（至少，我们希望你没有这样做。整个应用程序都使用正则表达式来构建，这样的方式我们想想都觉得酸爽）。

注意，这些语言可以通过减少它们提供的功能来有效地完成它们的目标。当你需要执行一个 SQL 语句时，不用从声明一个类或者方法开始。相反，在每一条 SQL 语句中最开始的关键字表明了需要执行的操作的类型，每种类型的操作都有自己独特的语法和针对特定任务的关键字集合。正则表达式的语法甚至更少：通过使用紧凑的标点符号语法说明文本的变化，程序直接描述了被匹配的文字。通过这样紧凑的语法，DSL 能够比通用编程语言中的等价代码更简洁地表达特定领域的操作。

另一个重点是 DSL 更趋向于声明式，和通用编程语言相反，它们大部分是命令式的。*命令式语言*描述了执行操作所需步骤的确切序列，而*声明式语言*描述了想要的结果并将执行细节留给了解释它的引擎。这通常会让执行更有效率，因为必要的优化只用在执行引擎上实现一次；而另一方面，命令式的方式要求每一个操作的实现都被独立优化。

这种类型的 DSL 有一个缺点，制衡了所有这些优势：它们很难与使用通用编程语言的宿主应用程序结合起来使用。它们有自己的语法并且不能直接嵌入到使用

不同语言的程序中去。因此，要调用使用 DSL 编写的程序，你要么需要将它存入一个单独的文件，要么将它嵌入一个字符串字面值。这样许多事情变得很不容易，包括在编译时验证 DSL 与宿主语言的正确交互、调试 DSL 程序，以及在编写时提供 IDE 代码辅助。同样，单独的语法需要独立地学习，也经常会让代码变得更难阅读。

要解决这些问题，同时还得保留 DSL 的其他优点，内部 DSL 的概念最近越来越受欢迎。让我们来看看这到底是什么。

11.1.2 内部 DSL

与有着自己独立语法的外部 DSL 不同，内部 DSL 是用通用编程语言编写的程序的一部分，使用了和通用编程语言完全一致的语法。实际上，内部 DSL 不是完全独立的语言，而是使用主要语言的特定方式，同时保留具有独立语法的 DSL 的主要优点。

为了比较两种方法，让我们来看看如何使用外部 DSL 和内部 DSL 来实现相同的任务。假设你有两个数据库表，Customer 和 Country，Customer 的每一条记录都有一个消费者居住国家的引用。要完成的任务是在数据库中查询并找出主要消费者群体居住的国家。下面将会使用的外部 DSL 是 SQL；而内部 DSL 是 Exposed 框架提供的（https://github.com/JetBrains/Exposed），这是一个使用 Kotlin 编写的用来访问数据库的框架。下面就是如何使用 SQL 来完成：

```
SELECT Country.name, COUNT(Customer.id)
    FROM Country
    JOIN Customer
      ON Country.id = Customer.country_id
GROUP BY Country.name
ORDER BY COUNT(Customer.id) DESC
   LIMIT 1
```

直接使用 SQL 来写代码也许不是那么方便：你必须提供一种在主要程序语言（这里是 Kotlin）和查询语言之间进行交互的方法。通常来讲，你能做到的最好方式就是把 SQL 放到一个字符串字面值中并寄希望于你的 IDE 能够帮你编写和验证它。

作为比较，下面就是使用 Kotlin 和 Exposed 构建的相同的查询：

```
(Country join Customer)
    .slice(Country.name, Count(Customer.id))
    .selectAll()
    .groupBy(Country.name)
    .orderBy(Count(Customer.id), isAsc = false)
    .limit(1)
```

你能看到这两个版本中的相似之处。事实上，第二个版本会生成并运行的 SQL 查询，与手动编写的一模一样。但是第二个版本是普通的 Kotlin 代码，并且

selectAll、groupBy、orderBy 等都是普通的 Kotlin 方法。此外，你不需要花任何的精力去把 SQL 查询结果集里的数据转成 Kotlin 对象——查询的执行结果就是直接以原生 Kotlin 对象交付的。因此我们将其称作一个内部 DSL：实现为通用编程语言（Kotlin）的库，旨在完成特定任务（构建 SQL 查询）的代码。

11.1.3 DSL 的结构

通常来讲，在 DSL 与普通的 API 之间并没有明确定义的边界，判断的标准常常主观到就像这样“当我看见它的时候我就知道它是一个 DSL”。DSL 通常会依赖在其他上下文中也会广泛使用的语言特性，比如说中缀调用和运算符重载。但是 DSL 中经常会出现一个通常在其他 API 中不存在的特征：*结构*或者说是*文法*。

一个典型的库由许多方法组成，客户端通过逐个调用方法来使用这个库。调用序列并没有内在的结构，而且在两个调用之间并没有维护上下文。这样的 API 有时候被叫作*命令查询* API。与之不同的是，DSL 的方法调用存在于由 DSL 文法定义的更大的结构中。在 Kotlin DSL 中，结构通常是通过嵌套的 lambda 表达式或链式方法调用来创建的。在前面的 SQL 例子中你能清楚地看到这一点：执行查询需要用描述所需结果集的不同因素的方法调用进行组合，而且和带上所有查询参数的单方法调用相比，组合查询太易读了。

这种文法使得我们能够将一个内部 DSL 称作一门*语言*。在一门像英语这样的自然语言中，句子就是用单词构成的，而文法规则控制着这些单词之间如何组合。与之类似，在 DSL 中，单一操作可以通过多个函数调用组成，而类型检查确保了调用会以一种有意义的方式组合起来。实际上，函数名称通常使用动词（groupBy、orderBy），它们的参数扮演名词的角色（Country.name）。

DSL 结构的一个好处就是允许你在多个函数调用之间重用一个上下文，而不是在每一次调用时都去重复它。这在接下来这个例子中有所体现，展示了用于描述 Gradle 构建脚本中依赖关系的 Kotlin DSL（https://github.com/gradle/gradle-script-kotlin）：

```
dependencies {
    compile("junit:junit:4.11")
    compile("com.google.inject:guice:4.1.0")
}
```

通过 lambda 嵌套表示的结构

比较一下下面这种通过普通的命令查询 API 实现的相同操作。注意在这段代码中的重复代码要多得多：

```
project.dependencies.add("compile", "junit:junit:4.11")
project.dependencies.add("compile", "com.google.inject:guice:4.1.0")
```

链式方法调用是另一种在 DSL 中创建结构的方法。例如，它们通常用于测试框架中，用来将断言分解为多个方法调用。这样的断言会更易读，特别是用了中缀调用语法之后。下面的例子来自于 kotlintest (https://github.com/kotlintest/kotlintest)，一个 Kotlin 的第三方测试框架，我们将会在 11.4.1 节中讨论更多的细节。

```
str should startWith("kot")
```

通过链式方法调用表示的结构

注意，同样的例子通过普通的 JUnit API 来表示是非常冗余的且不那么可读的：

```
assertTrue(str.startsWith("kot"))
```

现在我们来更细致地看看一个内部 DSL 的例子。

11.1.4 使用内部 DSL 构建 HTML

在本章开始部分的难题之一就是构建 HTML 页面的 DSL。在这一节，我们会讨论更多的细节。这里用到的 API 来自于 kotlinx.html 库（https://github.com/Kotlin/kotlinx.html）。这里有一个代码小片段，创建只有一个单元格的表格：

```
fun createSimpleTable() = createHTML().
    table {
        tr {
            td { +"cell" }
        }
    }
```

与这一结构对应的 HTML 是很清晰的：

```
<table>
  <tr>
    <td>cell</td>
  </tr>
</table>
```

`createSimpleTable` 函数返回了包含这个 HTML 片段的字符串。

为什么你要用 Kotlin 代码来构建这段 HTML，而不是用纯文本来编写它呢？首先，Kotlin 编写的版本是类型安全的：只能在 `tr` 中使用 `td` 标签，否则代码将不能编译。更为重要的是它是普通的代码，可以在其中使用任何语言结构。这意味着你可以在定义表格的位置动态地生成表格的单元格（例如，单元格对应着 map 中的元素）：

```
fun createAnotherTable() = createHTML().table {
    val numbers = mapOf(1 to "one", 2 to "two")
    for ((num, string) in numbers) {
        tr {
            td { +"$num" }
            td { +string }
        }
    }
}
```

生成的 HTML 包含期望的数据：

```
<table>
  <tr>
    <td>1</td>
    <td>one</td>
  </tr>
  <tr>
    <td>2</td>
    <td>two</td>
  </tr>
</table>
```

HTML 是标记语言的典型例子，这让它非常适合用来展示概念；但是对于具有类似结构的任何语言来说，比如 XML，都可以使用相同的方法。很快我们会讨论这样的代码如何在 Kotlin 中工作。

既然现在你已经知道了什么是 DSL，以及为什么要建立一个 DSL，那我们来看看 Kotlin 如何帮助你去完成它。首先，我们将更加深入了解*带接收者的 lambda*：这是帮助建立 DSL 文法的关键功能。

11.2 构建结构化的API:DSL中带接收者的lambda

带接收者的 lambda 是 Kotlin 的一个强大特性，它可以让你使用一个结构来构建 API。就像我们已经讨论过的，拥有结构是区分 DSL 和普通 API 的关键特征。让我们来仔细研究一下这个特性，看看一些用到它的 DSL。

11.2.1 带接收者的 lambda 和扩展函数类型

在 5.5 节已经简单地了解了带接收者的 lambda 这个概念，当时介绍了 `buildString`、`with` 和 `apply` 这些标准库函数。现在我们来看看它们是如何实现的，以 `buildString` 为例。这个函数将添加到一个中间 `StringBuilder` 中的几个字符串片段构建成一个字符串。

讨论之前，我们先定义一个以普通 lambda 作为参数的 `buildString` 函数。

在第 8 章已经讲过如何实现，这里还是熟悉的味道。

代码清单 11.1 定义以 lambda 为参数的 buildString()

```
fun buildString(
        builderAction: (StringBuilder) -> Unit
): String {
    val sb = StringBuilder()
    builderAction(sb)
    return sb.toString()
}

>>> val s = buildString {
...     it.append("Hello, ")
...     it.append("World!")
... }
>>> println(s)
Hello, World!
```

定义一个函数类型的参数

传递一个 StringBuilder 对象作为 lambda 的参数

使用“it”指向 StringBuilder 实例

这段代码很好理解，但却比我们期待的要难用一些。注意，我们必须要用 lambda 函数体中的 it 去引用 StringBuilder 实例（可以定义自己的参数名取代 it，但还是需要显式地定义它）。这个 lambda 的主要目的是用文本填充 StringBuilder，所以我们想去掉 it. 前缀并直接调用 StringBuilder 的方法，用 append 替换 it.append。

要做到这一点，需要将 lambda 转换成带接收者的 *lambda*。实际上，可以将接收者的特殊状态赋予 lambda 参数中的一个，让你不需要任何修饰符就能直接调用它的成员。下面的代码清单演示了该如何去实现。

代码清单 11.2 重新定义以带接收者的 lambda 为参数的 buildString()

```
fun buildString(
        builderAction: StringBuilder.() -> Unit
) : String {
    val sb = StringBuilder()
    sb.builderAction()
    return sb.toString()
}

>>> val s = buildString {
...     this.append("Hello, ")
...     append("World!")
... }
>>> println(s)
Hello, World!
```

定义带接收者的函数类型的参数

传递一个 StringBuilder 实例作为 lambda 的接收者

用“this”关键字指向 StringBuilder 实例

也可以省略“this”，隐式地引用 StringBuilder

注意代码清单 11.1 和 11.2 的区别。首先，想想使用 buildString 的方式是如何改进的。现在传递一个带接收者的 lambda 作为参数，因此可以去掉 lambda 函数

体中的 it，用 append() 替换 it.append() 做函数调用。完整的格式是 this.append()。但对于类的普通成员，一般只有歧义需要澄清的时候才需要显式地使用 this。

然后我们来讨论 buildString 函数的定义发生了怎样的变化。可以用*扩展函数类型*取代普通函数类型来声明参数的类型。在声明扩展函数类型的时候，将函数类型签名中的一个参数（类型）移到括号前面，并用一个 . 将它与其他的（参数）类型分隔开。在代码清单 11.2 中，用 StringBuilder(). -> Unit 代替 (StringBuilder) -> Unit。这个特殊的类型（StringBuilder）就叫作*接收者类型*，传递给 lambda 的这个类型的值就叫作*接收者对象*。图 11.1 展示了一个更加复杂的扩展函数类型的声明。

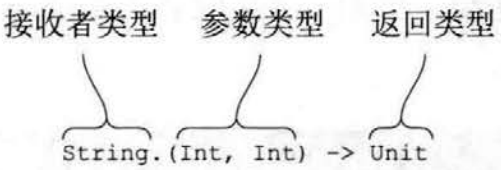


图 11.1 一个扩展函数类型，接收者类型是 String，两个参数类型是 Int，返回类型是 Unit

为什么要用*扩展函数类型*？不需要显式的修饰符就可以访问一个外部类型的成员让我们想起了扩展函数，它可以让我们为在代码其他地方定义的类定义自己的方法。扩展函数和带接收者的 lambda 都有一个*接收者对象*，当函数被调用的时候需要提供这个对象，它在函数体内是可用的。实际上，一个扩展函数类型描述了一个可以被当作扩展函数来调用的代码块。

当你将一个普通函数类型转换为扩展函数类型时，其调用方式也发生了变化。像调用一个扩展函数那样调用 lambda，而不是将对象作为参数传递给 lambda。在使用普通 lambda 时，我们使用这样的语法将一个 StringBuilder 实例作为参数给它：builderAction(sb)。但当你将它改成带接收者的 lambda 时，代码就变成了 sb.builderAction()。再次重申，这里的 builderAction 并不是 StringBuilder 类的方法，它是一个函数类型的参数，但可以用调用扩展函数一样的语法调用它。

图 11.2 展示了 buildString 函数的一个形参与一个实参的对应关系，同样也说明了用于调用 lambda 的接收者对象。

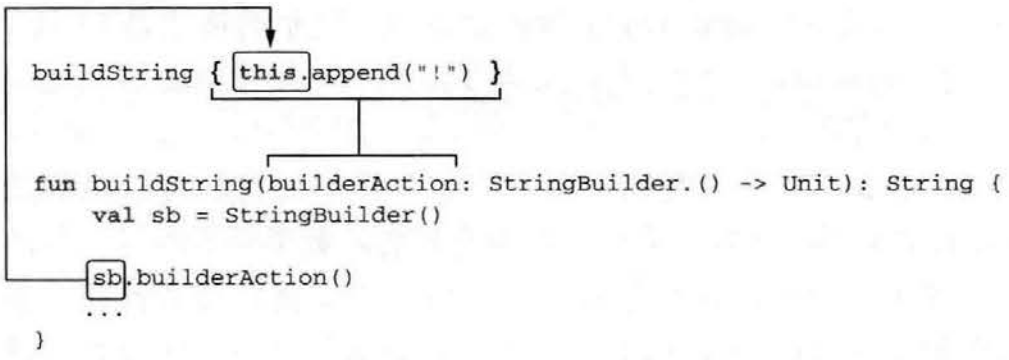


图 11.2 buildString 函数的实参（带接收者的 lambda）对应于扩展函数类型的形参（builderAction）。lambda 函数体被调用的时候接收者（sb）变成了一个隐式的接收者（this）

也可以声明一个扩展函数类型的变量，就像下面的代码一样。这样的话，就可以像扩展函数一样调用它，也可以将它作为参数传递给一个期望带接收者的 lambda 为参数的函数。

代码清单 11.3 用变量保存带接收者的 lambda

```
val appendExcl : StringBuilder.() -> Unit =
        { this.append("!") }

>>> val stringBuilder = StringBuilder("Hi")
>>> stringBuilder.appendExcl()
>>> println(stringBuilder)
Hi!

>>> println(buildString(appendExcl))
!
```

appendExcl 是一个扩展函数类型的值

可以像调用扩展函数一样调用 appendExcl

也可以将 appendExcl 作为参数传递

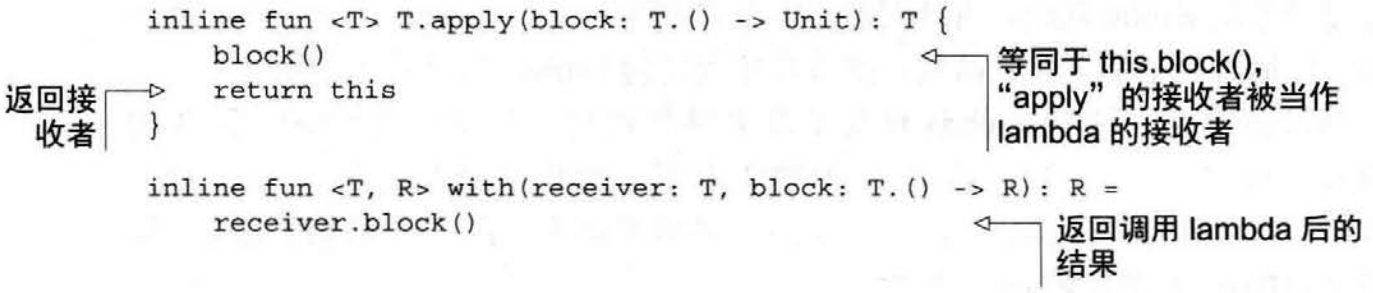
注意，在代码中带接收者的 lambda 跟普通的 lambda 看起来一模一样。要确定一个 lambda 是否有接收者，需要看 lambda 被传递给了什么函数：函数的签名会告诉你 lambda 是否有接收者，如果有，它的类型又是什么。例如，可以在 IDE 中查看 buildString 的文档或实现，可以看到它有一个 StringBuilder.() -> Unit 类型的 lambda 作为参数，可以推断出在 lambda 函数体中可以不需要修饰符即可调用 StringBuilder 的方法。

在标准库中 buildString 的实现比代码清单 11.2 中要简短一些。相比于显式地调用 builderAction，它被当作参数传递给 apply 函数（在 5.5 节介绍过）。这样就可以将函数缩减为一行：

```
fun buildString(builderAction: StringBuilder.() -> Unit): String =
        StringBuilder().apply(builderAction).toString()
```

apply 函数将调用它的对象（例子中全新的 StringBuilder 实例）作为隐

式接收者，对这个隐式接收者调用作为 apply 参数的函数或者 lambda（例子中的 builderAction）。在标准库中还有另外一个很有用的函数 :with。来研究一下它们的实现：



基本上 apply 和 with 所做的事情就是对提供给它的接收者调用扩展函数类型的参数。apply 函数被声明为这个接收者的扩展函数，而 with 函数把却把它作为第一个参数。另外，apply 返回接收者本身，而 with 返回调用 lambda 后的结果。

如果你并不关心返回值，这两个方式是可以互换使用的：

```
>>> val map = mutableMapOf(1 to "one")
>>> map.apply { this[2] = "two"}
>>> with (map) { this[3] = "three" }
>>> println(map)
{1=one, 2=two, 3=three}
```

apply 和 with 函数在 Kotlin 中很常用，希望你已经开始喜欢它们给代码带来的简洁性了。

我们已经复习了带接收者的 lambda，也讨论了扩展函数类型。现在是时候来看一看这些概念在 DSL 上下文中是如何被使用的。

11.2.2 在 HTML 构建器中使用带接收者的 lambda

用于 HTML 的 Kotlin DSL 通常被叫作 *HTML 构建器*，它代表了一个更普遍的概念叫作*类型安全的构建器*。最初，构建器的概念在 Groovy 社区（http://www.groovy-lang.org/dsls.html#_builders）很流行。构建器以声明式的方式构建对象层级结构，这种方式可以很方便地生成 XML 或者编排 UI 组件。

Kotlin 中使用了相同的理念，只不过在 Kotlin 中构建器是类型安全的。这使得 Kotlin 中的构建器比 Groovy 的动态构建器更方便、更安全、更有吸引力。我们来看看 Kotlin 中 HTML 构建器是怎么工作的。

代码清单 11.4 用 Kotlin HTML 构建器产生一个简单的 HTML

```
fun createSimpleTable() = createHTML().
    table {
```

```
        tr {
            td { +"cell" }
        }
    }
```

这是普通的 Kotlin 代码，并不是什么特别的模板语言：table、tr 和 td 都只是函数。它们每一个都是高阶函数，接收带接收者的 lambda 为参数。

值得注意的是，这些 lambda 改变了命名解析规则。在传递给 table 函数的 lambda 中，可以用 tr 函数创建 <tr> HTML 标签。而在 lambda 之外，tr 函数无法被解析。td 函数同样也只能在 tr 函数体内被解析使用（注意这里 API 的设计强制你遵守 HTML 语言的文法）。

每一个代码块中的命名解析上下文是由每一个 lambda 的接收者的类型定义的。传递给 table 的 lambda 的接收者的类型很特别，是 TABLE，它定义了 tr 函数。同理，tr 函数期望的 lambda 的接收者的类型是 TR。下面的代码清单是这些类和方法声明的简化版本。

代码清单 11.5 声明 HTML 构建器的标签类

```
open class Tag

class TABLE : Tag {
    fun tr(init : TR.() -> Unit)
}
class TR : Tag {
    fun td(init : TD.() -> Unit)
}
class TD : Tag
```

tr 函数所期望的 lambda 的接收者类型是 TR

td 函数所期望的 lambda 的接收者类型是 TD

TABLE、TR 和 TD 是工具类，不应该显式地出现在代码中，所以它们是以大写字母命名的。它们都继承自 Tag 这个基类。每个类都定义了方法，用来创建允许出现在它内部的标签：TABLE 类定义了 tr 方法，TR 类定义了 td 方法。

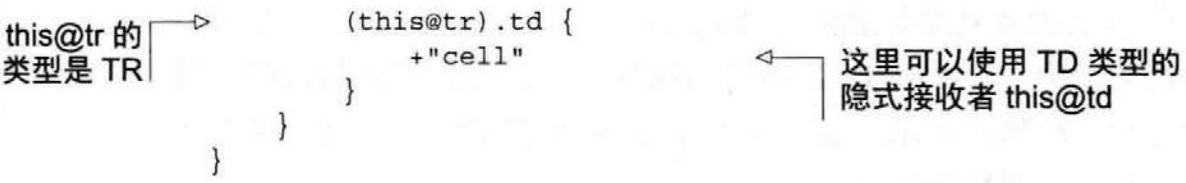
注意 tr 和 td 函数的 init 参数的类型：它们是扩展函数类型 TR.() -> Unit 和 TD.() -> Unit。它们分别决定了 lambda 参数的接收者类型：TR 和 TD。

为了更清楚地看到这里发生了什么，可以像清单 11.4 一样显式地使用所有接收者。提示一下，你可以像 this@foo 这样访问 foo 函数 lambda 参数的接收者。

代码清单 11.6 显式地使用 HTML 构建器的接收者

```
fun createSimpleTable() = createHTML().
    table {
        (this@table).tr {
```

this@table 的类型是 TABLE



如果在构建器中使用普通 lambda 代替带接收者的 lambda，语法会变得像这个例子一样难以阅读：你需要用 it 引用来调用标签创建方法，或者给每个 lambda 参数分配一个名字。隐藏 this 引用并隐式地使用接收者之后，构建器的语法变得漂亮，也更接近 HTML 语法。

注意，像代码清单 11.6 这样，如果一个带接收者的 lambda 放在另一个带接收者的 lambda 内，外部 lambda 定义的接收者在嵌套的 lambda 中也是可用的。比如，在作为 td 函数参数的 lambda 中，三个接收者（this@table、this@tr、this@td）都可用。但从 Kotlin1.1 开始，可以使用 @DslMarker 注解限制外部 lambda 的接收者的可用性。

我们已经解释了 HTML 构建器的语法是如何依赖于带接收者的 lambda 这个概念的。现在我们来讨论 HTML 是如何生成的。

代码清单 11.6 用到了 kotlinx.html 库中的函数。现在你将要实现一个简单得多的 HTML 构建器库版本：扩展 TABLE、TR 和 TD 标签的声明并添加生成最终 HTML 的支持。作为这个简化版本的入口，顶层的 table 函数用 <table> 标签创建了一个 HTML 片段作为最外层的标签。

代码清单 11.7 生成一个 HTML 字符串

```
fun createTable() =
    table {
        tr {
            td {
            }
        }
    }

>>> println(createTable())
<table><tr><td></td></tr></table>
```

table 函数创建了 TABLE 标签的一个新实例，初始化（调用作为 init 参数传递给它的函数）并返回它。

```
fun table(init: TABLE.() -> Unit) = TABLE().apply(init)
```

在函数 createTable 中，传递给 table 函数的 lambda 包含了 tr 函数的调用。可以重写代码以尽量显式地调用所有函数：table(init = { this.tr { ...

} })。tr 函数会对 TABLE 类的实例调用，就像 TABLE().tr { ... } 这样。

在这个玩具似的例子中，<table> 是顶层的标签，其他的标签嵌套在中。每一个标签持有了一个它的子标签的引用列表。因此，tr 函数不仅初始化 TR 标签的实例，还会将它添加到外层标签的子标签列表中。

代码清单 11.8 定义一个标签构建器函数

```
fun tr(init: TR.() -> Unit) {
    val tr = TR()
    tr.init()
    children.add(tr)
}
```

对于所有标签来说，初始化一个给定的标签并把它添加到外层标签的子标签列表中，这套逻辑都是一样的，所以可以把这套逻辑抽取成 Tag 基类中的 doInit 成员方法。doInit 函数负责两件事情：保存子标签的引用和调用作为参数的 lambda。然后不同的标签会调用它：比如 tr 函数创建一个 TR 类的新实例，然后和 init 的 lambda 参数一起传给 doInit 函数：doInit(TR(), init)。下面的代码清单完整地展示了如何生成目标 HTML。

代码清单 11.9 一个简单 HTML 构建器的完整实现

```
open class Tag(val name: String) {
    private val children = mutableListOf<Tag>()

    protected fun <T : Tag> doInit(child: T, init: T.() -> Unit) {
        child.init()
        children.add(child)
    }

    override fun toString() =
        "<$name>${children.joinToString("")}</$name>"
}

fun table(init: TABLE.() -> Unit) = TABLE().apply(init)

class TABLE : Tag("table") {
    fun tr(init: TR.() -> Unit) = doInit(TR(), init)
}
class TR : Tag("tr") {
    fun td(init: TD.() -> Unit) = doInit(TD(), init)
}
class TD : Tag("td")

fun createTable() =
    table {
```

保存所有嵌套标签

初始化子标签

保存子标签的引用

返回 HTML 字符串

创建、初始化 TR 标签的示例并添加到 TABLE 的子标签中

添加 TD 标签的一个实例到 TR 的子标签中

```
        tr {
            td {
            }
        }
    }
>>> println(createTable())
<table><tr><td></td></tr></table>
```

每个标签保存了一个嵌套标签的列表并相应地渲染它自己：先渲染它自己的名字然后递归渲染嵌套标签。这里的实现并不支持标签内的文本和标签属性，完整的实现请参考前面提到的 kotlinx.html 库。

注意，标签创建函数自己会把相应的标签加到它们父标签的子标签列表中，这么做可以动态地生成标签。

代码清单 11.10 用 HTML 构建器动态地生成标签

```
fun createAnotherTable() = table {
    for (i in 1..2) {
        tr {                          <-- 每一次调用“tr”都会创建一个新的 TR 标签实例，并将它添加到 TABLE 的子标签中
            td {
            }
        }
    }
}
>>> println(createAnotherTable())
<table><tr><td></td></tr><tr><td></td></tr></table>
```

如你所见，带接收者的 lambda 是构建 DSL 的好工具。可以在代码块中改变命名解析上下文，因此让你可以在 API 中创建结构，它是区分平铺的方法调用序列与 DSL 的关键特征之一。现在我们来讨论将 DSL 集成到静态类型编程语言中的好处。

11.2.3 Kotlin 构建器：促成抽象和重用

在程序中编写普通代码时，有许多工具可以用来避免重复代码或者让代码看起来更漂亮。其中，可以将重复代码抽取到新的函数中，并给它一个自解释的名称。用 SQL 或 HTML（提取重复代码）没有这么简单，甚至不可能做到。但使用 Kotlin 内部的 DSL 却可以做到同样的事情，将重复代码抽取到函数中并重用。

来看 Bootstrap 库（http://getbootstrap.com）中的一个例子，Bootstrap 是一个很流行的 HTML、CSS、JS 框架，用于开发响应式的，移动优先的 web 程序。先看一个具体的例子：在程序中添加下拉列表。要直接在 HTML 页面中添加这样一个列表，可以通过拷贝必要的代码段并粘贴到需要的地方，在按钮或者其他要显示列表的元素下方。你只需要添加必要的超链接和下拉菜单的标题。初始的 HTML 代码（简化版本，去掉了过多的样式属性）如下所示：

代码清单 11.11 用 Bootstrap 在 HTML 中生成一个下拉菜单

```
<div class="dropdown">
  <button class="btn dropdown-toggle">
    Dropdown
    <span class="caret"></span>
  </button>
  <ul class="dropdown-menu">
    <li><a href="#">Action</a></li>
    <li><a href="#">Another action</a></li>
    <li role="separator" class="divider"></li>
    <li class="dropdown-header">Header</li>
    <li><a href="#">Separated link</a></li>
  </ul>
</div>
```

在 Kotlin 中使用 kotlinx.html，可以用 div、button、ul、li 等函数复制同样的结构。

代码清单 11.12 用 Kotlin HTML 构建器构建一个下拉菜单

```
fun buildDropdown() = createHTML().div(classes = "dropdown") {
    button(classes = "btn dropdown-toggle") {
        +"Dropdown"
        span(classes = "caret")
    }
    ul(classes = "dropdown-menu") {
        li { a("#") { +"Action" } }
        li { a("#") { +"Another action" } }
        li { role = "separator"; classes = setOf("divider") }
        li { classes = setOf("dropdown-header"); +"Header" }
        li { a("#") { +"Separated link" } }
    }
}
```

但是你还可以做得更好。因为 div、button 等都是普通函数，可以将重复的逻辑抽取到独立的函数中，来提高代码的可读性。然后代码就变成下面这样。

代码清单 11.13 使用辅助函数构建下拉菜单

```
fun dropdownExample() = createHTML().dropdown {
    dropdownButton { +"Dropdown" }
    dropdownMenu {
        item("#", "Action")
        item("#", "Another action")
        divider()
        dropdownHeader("Header")
        item("#", "Separated link")
    }
}
```

现在不必要的细节都被隐藏了，代码看起来也整洁了很多。我们从 item 函数开始，来讨论这个实现的诀窍。这个函数有两个参数：超链接和对应菜单项的名称。这个函数的代码会添加一个列表项：li { a(href) { +name } }。唯一的问题在于，如何才能在函数体中调用 li。它应该是扩展函数吗？的确可以让它成为 UL 类的一个扩展，因为 li 本身就是 UL 类的扩展。在代码清单 11.13 中，item 在 UL 类型的一个隐式 this 上调用。

```
fun UL.item(href: String, name: String) = li { a(href) { +name } }
```

定义了 item 函数后，就可以在任何 UL 标签中调用它，它会添加一个 LI 标签的实例。通过抽取 item，就可以把最初版本的代码变成下面这样，而且不会改变生成的 HTML 代码。

代码清单 11.14 使用 item 函数构建下拉菜单

```
ul {
    classes = setOf("dropdown-menu")
    item("#", "Action")
    item("#", "Another action")
    li { role = "separator"; classes = setOf("divider") }
    li { classes = setOf("dropdown-header"); +"Header" }
    item("#", "Separated link")
}
```

这里用“item”函数代替“li”

另一些扩展函数以相似的方式添加到 UL 类中，用来替代剩下的 li 标签。

```
fun UL.divider() = li { role = "separator"; classes = setOf("divider") }

fun UL.dropdownHeader(text: String) =
    li { classes = setOf("dropdown-header"); +text }
```

来看看 dropdownMenu 的实现。它用指定的 dropdown-menu 类创建了一个 ul 标签，并以一个带接收者的 lambda 为实参，这个 lambda 用来填充标签的内容。

```
dropdownMenu {
    item("#", "Action")
    ...
}
```

因为用 dropdownMenu { ... } 的调用代替了 ul { ... } 代码块，所以 lambda 的接收者可以保持不变。dropdownMenu 函数可以接收 UL 类的扩展 lambda 作为参数，这个参数可以让你像之前一样调用 UL.item 这样的函数。这是它的函数声明：

```
fun DIV.dropdownMenu(block: UL.() -> Unit) = ul("dropdown-menu", block)
```

dropdownButton 函数也是以相似的方式实现的。这里省略它的实现，可以在 kotlinx.html 库的例子中找到它的完整实现。

最后，来看看 dropdown 函数。它比较重要，因为任何标签都可以调用它：下拉菜单可以放在代码中的任何位置。

代码清单 11.15 用于构建下拉菜单的顶层函数

```
fun StringBuilder.dropdown(
        block: DIV.() -> Unit
): String = div("dropdown", block)
```

这是一个简化版本，你可以使用它，如果你想将 HTML 作为字符串打印出来。在 kotlinx.html 库的完整实现中使用一个抽象类 TagConsumer 作为接收者，因此它支持生成的 HTML 的不同归宿。

这个例子阐述了抽象和重用是如何帮助我们改进代码并让代码更易于理解的。接下来我们来看另一个工具，它使得我们在 DSL 中支持更多灵活的结构：invoke 约定。

11.3 使用“invoke”约定构建更灵活的代码块嵌套

invoke 约定允许把自定义类型的对象当作函数一样调用。你已经见过函数类型的对象，它们可以作为函数调用；使用 invoke 约定，也可以定义支持同样语法的自己的对象。

注意这并不是一个日常使用的功能，因为这会写出一些难以理解的代码，比如 1()。但是有时候这在 DSL 中非常有用。我们将会告诉你为什么是这样，但首先我们还是来谈谈约定本身。

11.3.1 “invoke”约定：像函数一样可以调用的对象

在第 7 章中，我们深入讨论了 Kotlin 中约定的概念：使用与常规方法调用语法不同的、更简洁的符号，调用有着特殊命名的函数。提醒一句，get 就是我们讨论过的约定之一，它允许你使用下标运算符来访问一个对象。对于一个 Foo 类型的变量 foo，如果对应的 get 函数被定义为 Foo 类的成员或者 Foo 的扩展函数，那么 foo[bar] 调用就会被转换成 foo.get(bar)。

实际上，invoke 约定做了同样的事情，除了用括号替换了 get 约定中的方括号外。类如果定义了使用 operator 修饰符的 invoke 方法，就可以被当作函数一样调用。下面的例子说明了这是怎么工作的。

代码清单 11.16 在类中定义一个 invoke 方法

```
class Greeter(val greeting: String) {
    operator fun invoke(name: String) {
        println("$greeting, $name!")
    }
}

>>> val bavarianGreeter = Greeter("Servus")
>>> bavarianGreeter("Dmitry")
Servus, Dmitry!
```

在 Greeter 上定义“invoke”方法

像函数一样调用 Greeter 实例

这段代码在 Greeter 中定义了 invoke 方法，允许你将 Greeter 实例当作函数调用。在这种情况下，表达式 bavarianGreeter("Dmitry") 会被编译成方法调用 bavarianGreeter.invoke("Dmitry")。这里并没有任何神秘的地方。它就如同一个普通约定那样工作：提供了一种方法，用更简洁、清晰的表达式替换了冗长的表达式。

invoke 方法并没有限制任何特殊的签名。可以给它定义任意数量的参数和任意的返回类型。当你将类实例像函数那样调用时，可以使用所有这些签名。让我们来看看使用约定的实际场景吧，首先是在普通的程序上下文中，接着是在 DSL 中。

11.3.2 “invoke”约定和函数式类型

你也许会记得在本书前面部分见过 invoke。在 8.1.2 节中我们讨论过可以使用带 invoke 方法名称的安全调用语法，像 lambda?.invoke() 这样调用一个可空函数类型的变量。

现在你知道了 invoke 约定，你应该清楚调用 lambda 的普通方式（把括号放在它后面：lambda()）并没什么特别的，只是这种约定的一种应用而已。Lambda，除非是内联的，都是被编译成实现了函数式接口 (Function1 等) 的类，而这些接口定义了具有对应数量参数的 invoke 方法：

```
interface Function2<in P1, in P2, out R> {
    operator fun invoke(p1: P1, p2: P2): R
}
```

这个接口表示正好具有两个参数的函数

当你将 lambda 作为函数调用时，这种操作被转换成一次 invoke 方法调用。为什么了解这些会有用呢？它提供了一种这样的方式，将复杂 lambda 代码拆分成多个方法，同时仍然允许将它与接收函数类型参数的函数一起使用。要做到这一点，你可以定义一个实现了函数类型接口的类。你可以将基础接口声明为一个显式的 FunctionN 类型，或者像下面的代码清单那样，使用简明语法：(P1, P2) -> R。

这个例子使用复杂的条件来过滤问题列表，其中就用到了这样的类。

代码清单 11.17 扩展函数类型并重写 invoke()

```
data class Issue(
    val id: String, val project: String, val type: String,
    val priority: String, val description: String
)

class ImportantIssuesPredicate(val project: String)
        : (Issue) -> Boolean {

    override fun invoke(issue: Issue): Boolean {
        return issue.project == project && issue.isImportant()
    }

    private fun Issue.isImportant(): Boolean {
        return type == "Bug" &&
                (priority == "Major" || priority == "Critical")
    }
}

>>> val i1 = Issue("IDEA-154446", "IDEA", "Bug", "Major",
...                 "Save settings failed")
>>> val i2 = Issue("KT-12183", "Kotlin", "Feature", "Normal",
... "Intention: convert several calls on the same receiver to with/apply")
>>> val predicate = ImportantIssuesPredicate("IDEA")
>>> for (issue in listOf(i1, i2).filter(predicate)) {
...     println(issue.id)
... }
IDEA-154446
```

将函数类型用作基类

实现“invoke”方法

将判断式传入filter()

这里判断式的逻辑要是放到单个 lambda 中就太复杂了，所以你会把它拆分成多个方法，使每种检查的意义清晰。将 lambda 转换成一个实现了函数类型接口的类，并重写接口的 invoke 方法；是进行这种重构的一种方式。这种方法的优点是从 lambda 函数体中抽取的方法的作用域被尽可能缩小了；它们仅在判断式类内部可见。在判断式类和其周围代码都有很多逻辑时这是非常有价值的，并且将不同的关注点清楚地分离也是值得的。

现在我们来看看 invoke 约定怎样帮助你为自己的 DSL 创建更灵活的结构。

11.3.3 DSL 中的“invoke”约定：在 Gradle 中声明依赖

让我们再来回顾一下用来配置模块依赖的 Gradle DSL 的例子。这里就是前面展示过的例子：

```
dependencies {
    compile("junit:junit:4.11")
}
```

你常常会想在一个 API 中同时支持嵌套代码块结构（就像上面展示的一样）和扁平调用结构。换句话说，下面的两种都是你想要的：

```
dependencies.compile("junit:junit:4.11")

dependencies {
    compile("junit:junit:4.11")
}
```

通过这样的设计，有多个依赖项需要配置时，DSL 的用户可以使用嵌套代码块结构；而在只有一个依赖项需要配置时，用户可以使用扁平调用结构，来保持代码简洁。

第一种情况在 dependencies 变量上调用了 compile 方法。也可以为 dependencies 定义 invoke 方法，这样它（dependencies）就可以接收 lambda 作为参数，而你就可以使用第二种表示法了。这次调用的完整语法是 dependencies.invoke({...})。

dependencies 对象是 DependencyHandler 类的一个实例，它同时定义了 compile 和 invoke 方法。invoke 方法接受一个带接收者的 lambda 作为参数，并且这个方法的接收者的类型还是 DependencyHandler。Lambda 函数体中发生的一切你已经很熟悉了：你有一个作为接收者的 DependencyHandler 并且可以在上面直接调用 compile 这样的方法。下面的小例子展示了部分 DependencyHandler 的实现。

代码清单 11.18 使用 invoke 来支持灵活的 DSL 语法

```
class DependencyHandler {
    fun compile(coordinate: String) {
        println("Added dependency on $coordinate")
    }

    operator fun invoke(
            body: DependencyHandler.() -> Unit) {
        body()
    }
}
```

定义一个普通的命令 API

定义“invoke”来支持 DSL API

“this”变成了 body 函数的接收者：this.body()

```
>>> val dependencies = DependencyHandler()

>>> dependencies.compile("org.jetbrains.kotlin:kotlin-stdlib:1.0.0")
Added dependency on org.jetbrains.kotlin:kotlin-stdlib:1.0.0

>>> dependencies {
...     compile("org.jetbrains.kotlin:kotlin-reflect:1.0.0")
>>> }
Added dependency on org.jetbrains.kotlin:kotlin-reflect:1.0.0
```

当你添加第一个依赖时，直接调用了 compile 方法。第二个调用实际上被转换成了下面的代码：

```
dependencies.invoke({
    this.compile("org.jetbrains.kotlin:kotlin-reflect:1.0.0")
})
```

换句话说，你将 dependencies 当成函数调用并传入一个 lambda 作为它的参数。Lambda 参数的类型是带接收者的函数类型，并且接收者的类型同样是 DependencyHandler。invoke 方法调用了这个 lambda。因为它是 DependencyHandler 类的一个方法，所以该类的实例可以作为隐式接收者使用，因此在调用 body() 时不需要显式地指定它。

这样一段短小的代码片段，仅仅重定义了一个 invoke 方法，就显著地增加了 DSL API 的灵活性。这个模式很通用，稍加修改就能在你自己的 DSL 中重用。

现在你已经熟悉了能够帮助你构建 DSL 的 Kotlin 的两个新特性：带接收者的 lambda 和 invoke 约定。让我们来看看前面讨论的 Kotlin 特性在 DSL 上下文中如何发挥作用。

11.4 实践中的Kotlin DSL

到现在，你已经熟悉在构建 DSL 时使用的所有 Kotlin 特性。其中有一些，像扩展和中缀调用已经是你的老朋友了，另外的诸如带接收者的 lambda 则是在本章中第一次进行深入探讨。让我们把所有这些知识都用上并调研一系列实际的 DSL 构造实例。我们会涵盖到相当多样的主题：包括测试、富日期字面值、数据库查询和 Android UI 构建。

11.4.1 把中缀调用链接起来：测试框架中的“should”

像我们前面提到的那样，清晰的语法是内部 DSL 的一个关键特点，它可以通过在减少代码中的标点符号数量来达成。大多数内部 DSL 可以归结为方法调用的序列，所以任何让你减少方法调用中的语法噪声的特性都能够在这里大显身手。在 Kotlin 中，这些特性包括，前面详细讨论过的调用 lambda 的简明语法，还有中缀函数调用。在 3.4.3 节中已经讨论了中缀调用，这里我们将会关注在 DSL 中使用它们。

让我们来看一个使用 kotlintest（https://github.com/kotlintest/kotlintest）DSL 的例子，我们在本章的前面部分已经见过它了。

代码清单 11.19 使用 kotlintest DSL 来表示一句断言

```
s should startWith("kot")
```

如果 s 变量不是以“kot”开头的，这个断言将会失败。这段代码读起来就像英语：“s 字符串应该以这个常量打头”。为了做到这点，需要使用 infix 修饰符来声明 should 函数。

代码清单 11.20 实现 should 函数

```
infix fun <T> T.should(matcher: Matcher<T>) = matcher.test(this)
```

这个 should 函数期望接收一个 Matcher 的实例，这是一个在值上执行断言的泛型接口。startWith 实现了 Matcher 并检查字符串是否以给定的子字符串打头。

代码清单 11.21 为 kotlintest DSL 定义一个匹配器

```
interface Matcher<T> {
    fun test(value: T)
}
class startWith(val prefix: String) : Matcher<String> {
    override fun test(value: String) {
        if (!value.startsWith(prefix))
            throw AssertionError("String $value does not start with $prefix")
    }
}
```

注意在普通的代码中，你需要将 startWith 的类名以大写字母开头，但是 DSL 常常需要你偏离标准命名约定。代码清单 11.21 就展示了在 DSL 上下文中使用中缀调用非常简单，并且能够减少代码中的噪声。处理得更巧妙一些的话，你还可以更进一步地减少噪声。kotlintest DSL 还支持这样。

代码清单 11.22 kotlintest DSL 的链式调用

```
"kotlin" should start with "kot"
```

乍一看，这根本不像 Kotlin。要想了解它的原理，我们把中缀调用转成普通写法。

```
"kotlin".should(start).with("kot")
```

这显示了代码清单 11.22 是按两个中缀调用组成的序列，而 start 是第一个调用的参数。事实上，start 引用了一个对象声明，而 should 和 with 是使用中缀

调用表示法进行调用的函数。

should 函数有一个特殊的重载，它使用 start 对象作为一个参数类型并返回一个中间包装器，然后可以在其上调用 with 方法。

代码清单 11.23 定义 API 来支持链式中缀调用

```
object start

infix fun String.should(x: start): StartWrapper = StartWrapper(this)

class StartWrapper(val value: String) {
    infix fun with(prefix: String) =
        if (!value.startsWith(prefix))
            throw AssertionError(
                "String does not start with $prefix: $value")
}
```

注意，在 DSL 上下文之外，使用一个 object 作为参数类型几乎没有意义，因为它只有唯一一个实例，可以直接访问它而不是将它作为参数进行传递。而这里是有意义的：object 不是用来给函数传递任何数据的，而是 DSL 文法的一部分。通过将 start 作为参数传递，可以选择正确的 should 重载并获得一个作为结果的 StartWrapper 实例。StartWrapper 类有一个 with 成员，使用需要执行断言的实际值作为参数。

这个库同样还支持其他的匹配器，它们都读起来都像英语一样：

```
"kotlin" should end with "in"
"kotlin" should have substring "otl"
```

要支持这些，should 函数有还有更多的重载，分别接收像 end 和 have 这样的对象实例并返回 EndWrapper 和 HaveWrapper 的实例。

这是 DSL 结构中相当取巧的例子，但是最终结果是如此美妙，非常值得去理解这种模式的原理。中缀调用和 object 实例的结合能为你的 DSL 构建相当复杂的文法，并以清晰的语法来使用这些 DSL。当然，DSL 依然是完全静态类型的。错误的函数和对象结合将不能编译通过。

11.4.2 在基本数据类型上定义扩展：处理日期

现在我们来看看在本章开始时留下的难题：

```
val yesterday = 1.days.ago
val tomorrow = 1.days.fromNow
```

要使用 Java 8 java.time API 和 Kotlin 来实现这样的 DSL，只需要几行代码。

下面就是实现的相关内容。

代码清单 11.24 定义一个日期操作的 DSL

```
val Int.days: Period
    get() = Period.ofDays(this)

val Period.ago: LocalDate
    get() = LocalDate.now() - this

val Period.fromNow: LocalDate
    get() = LocalDate.now() + this

>>> println(1.days.ago)
2016-08-16
>>> println(1.days.fromNow)
2016-08-18
```

"this" 引用了数字常量的值

使用运算符语法调用 LocalDate.minus

使用运算符语法调用 LocalDate.plus

在这里，days 是一个 Int 类型的扩展属性。Kotlin 没有限制能被用作扩展函数的接收者的类型：你能很容易地在基本数据类型上定义扩展并在常量上调用它们。days 属性返回一个 Period 类型的值，它是 JDK 8 中用来表示两个日期之间间隔的类型。

要完成这个句子并支持单词 ago，你需要定义另一个扩展属性，这次是在 Period 类上。这个属性的类型是 LocalDate，表示一个日期。注意在 ago 属性实现中 - (减号) 运算符并不依赖任何 Kotlin 定义的扩展。LocalDate JDK 类定义了一个带一个参数名为 minus 的方法，恰好符合 Kotlin 关于 - 运算符的约定，所以 Kotlin 自动将这个运算符映射到了这个方法上。可以在 GitHub 上的 kxdate 库（https://github.com/yole/kxdate）中找到完整实现，它支持所有时间单位，而不仅仅是以天为单位。

现在你明白了 DSL 工作起来是多么简单，让我们转向更有挑战性的一些内容：数据库查询 DSL 的实现。

11.4.3 成员扩展函数：为 SQL 设计的内部 DSL

你已经见过扩展函数在 DSL 设计中扮演的重要角色。在这一小节，我们将学习之前提到的另一个技巧：在类中声明扩展函数和扩展属性。这样的函数或属性既是它容器类的成员，也是某些其他类型的扩展。我们把这样的函数和属性叫作成员扩展。

我们来看看使用了成员扩展的一系列示例。它们来自之前提到过的为 SQL 设计的内部 DSL——Exposed 框架。在开始之前，我们需要讨论 Exposed 是如何允许你去定义数据库结构的。

为了使用SQL表，Exposed框架需要你将它们声明为继承了Table类的对象。下面就是一个简单的带两列的Country表。

代码清单 11.25 在Exposed中声明一个表

```
object Country : Table() {
    val id = integer("id").autoIncrement().primaryKey()
    val name = varchar("name", 50)
}
```

这个声明与数据库中的一个表相对应。要创建这个表，需要调用SchemaUtils.create(Country)方法，它会基于声明的表结构生成需要的SQL语句：

```
CREATE TABLE IF NOT EXISTS Country (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    CONSTRAINT pk_Country PRIMARY KEY (id)
)
```

就像生成HTML一样，你可以看到原始Kotlin代码中的声明是如何变成生成的SQL语句的一部分的。

如果你去检查Country对象中属性的类型，你会看到它们都是带必要类型参数的Column类型：id的类型是Column<Int>，name的类型是Column<String>。

Exposed框架中的Table类定义了所有你在表中能够声明的类型，包括刚刚使用的那些：

```
class Table {
    fun integer(name: String): Column<Int>
    fun varchar(name: String, length: Int): Column<String>
    // ...
}
```

integer和varchar方法各自创建了用来存储整型和字符串的新列。

现在我们来看看如何指定列的属性。是成员扩展闪亮登场的时候了：

```
val id = integer("id").autoIncrement().primaryKey()
```

像autoIncrement和primaryKey这样的方法被用来指定每一列的属性。每个方法都能在Column上调用并返回调用发生的实例（就是自己），这样你就能把这些方法链接起来使用。下面是这些函数简化后的声明：

```
class Table {
    fun <T> Column<T>.primaryKey(): Column<T>
    fun Column<Int>.autoIncrement(): Column<Int>
    // ...
}
```

将该列设成表的主键

只有整型值能够自增

这些函数都是 Table 类的成员，这意味着你不能在这个类作用域之外去使用它们。现在你明白为什么将方法声明为成员扩展讲得通了吧：你限制了它们应用的作用域。你不能在表的上下文之外去指定一列的属性：必要的方法根本无法解析。

在这里还用到了扩展函数的另一个很棒的功能，就是限制接收者类型的能力。虽然表中任意一列都可以作为主键，但是只有数字列是可以自增的。通过将 autoIncrement 方法声明为 Column<Int> 的扩展，你就可以在 API 中表达这个意思。把其他类型的列标记为自增的尝试会导致编译失败。

此外，当你将一列标记为 primaryKey 时，这些信息存储在包含这一列的表中。将此函数声明为 Table 的成员能够允许你直接把这些信息存储到表的实例中。

成员扩展依然是成员

成员扩展还是有缺点的：那就是扩展性的缺失。它们是属于类的，所以你不能另外定义新的成员扩展。

例如，假设你想为 Exposed 添加对新数据库的支持，而这个数据库需要支持一些新的列属性。要想达到这个目标，你必须修改 Table 类的定义来为新的属性增添成员扩展函数。你不能像普通（非成员）扩展那样，在不改动原始类的情况下添加必要的声明，因为那样的扩展不能访问这些可以存储定义的 Table 实例。

让我们来看看另一个可以在简单的 SELECT 查询中找到的成员扩展函数。假设你声明了两个表，Customer 和 Country，并且每一条 Customer 的记录存储了消费者来自哪个国家的引用。下面的代码用来打印所有生活在美国的消费者的名字。

代码清单 11.26 在 Exposed 中连接两个表

```
val result = (Country join Customer)
    .select { Country.name eq "USA" }
result.forEach { println(it[Customer.name]) }
```

对应相应的 SQL 代码：WHERE Country.name = "USA"

select 方法能够在 Table 或者两个表的连接上调用。它的参数是一个 lambda，用来指定查询必要数据的条件。

eq 方法是从哪里来的呢？我们现在可以认出它是一个使用 "USA" 作为参数的

中缀函数，并且你也许能够猜到它是另一个成员扩展。

这里你再次遇到一个 Column 上的扩展函数，它同样也是一个成员，因此只能在适当的上下文中使用：例如，在指定 select 方法的条件时。简化过的 select 和 eq 方法声明如下：

```
fun Table.select(where: SqlExpressionBuilder.() -> Op<Boolean>) : Query

object SqlExpressionBuilder {
    infix fun<T> Column<T>.eq(t: T) : Op<Boolean>
    // ...
}
```

SqlExpressionBuilder 对象定义了很多种表示条件的方式：比较值、检查非 null、执行算数操作等。你永远不会在代码中显式引用它，但是在它是个隐式接收者的时候，你会经常调用它的方法。select 函数使用带接收者的 lambda 作为参数，并且 SqlExpressionBuilder 对象在这个 lambda 中是一个隐式接收者。这允许你在 lambda 函数体中使用所有这个对象中定义的能使用的扩展函数，比如 eq。

你已经见过了列上的两种扩展：用来声明一个 Table 的那些，以及用来在条件中比较值的那些。没有成员扩展的话，你必须将这些函数都声明为 Column 的扩展或成员，这将允许你在任何上下文中使用它们。成员扩展的方式给你提供了一种控制的方法。

> **注意** 在 7.5.6 节中，在讲到使用框架中的委托属性时我们看过一些使用 Exposed 的代码。委托属性经常出现在 DSL 中，Exposed 框架就说明了这一点。我们在这里就不再赘述委托属性，因为我们已经深入讨论过它们。但是如果你热切地想为你自己的需求创建一些 DSL 或是改进你的 API 来让它们变得更干净，请把这个功能牢记于心。

11.4.4 Anko：动态创建 Android UI

当我们讨论带接收者的 lambda 时，我们提到它们非常适合用来做 UI 组件的布局。让我们来看看 Anko 库（https://github.com/Kotlin/anko）是如何帮助你构建 Android 应用程序的 UI 的。

首先，我们来看看 Anko 是如何将熟悉的 Android API 包装成 DSL 样式的结构的。下面的代码清单定义了一个警告对话框，展示了一条有点讨厌的消息和两个选项（要继续处理还是停止操作）。

代码清单 11.27 使用 Anko 来展示一个 Android 警告对话框

```
fun Activity.showAreYouSureAlert(process: () -> Unit) {
    alert(title = "Are you sure?",
          message = "Are you really sure?") {
        positiveButton("Yes") { process() }
        negativeButton("No") { cancel() }
    }
}
```

你能指出这段代码中的三个 lambda 吗？首先是 alert 函数的第三个参数，另外两个则被作为参数传给 positiveButton 和 negativeButton。第一个（外层）lambda 的接收者的类型是 AlertDialogBuilder。同样的模式又出现了：AlertDialogBuilder 类的名字在代码中并没有直接出现，但你能访问它的成员给警告对话框添加元素。在代码清单 11.27 中成员的声明如下。

代码清单 11.28 alert API 的声明

```
fun Context.alert(
        message: String,
        title: String,
        init: AlertDialogBuilder.() -> Unit
)

class AlertDialogBuilder {
    fun positiveButton(text: String, callback: DialogInterface.() -> Unit)
    fun negativeButton(text: String, callback: DialogInterface.() -> Unit)
    // ...
}
```

你给警告对话框添加了两个按钮。如果用户点击了 Yes 按钮，process 动作将会被调用。如果用户还不确定，那么操作会被取消。cancel 方法是 DialogInterface 接口的成员，所以它在这个 lambda 的隐式接收者上被调用。

现在我们来看看更复杂的例子，在这里 Anko DSL 扮演了 XML 中布局定义的完全替身。下一个代码清单声明了一个带两个可编辑栏位的简单表格：一个用来输入 email 地址，另一个用来输入密码。最后，还有一个带点击处理器的按钮。

代码清单 11.29 使用 Anko 定义一个简单的 activity

```
verticalLayout {
    val email = editText {
        hint = "Email"
```

声明一个 EditText 视图元素并存储到一个引用上

这个 lambda 的隐式接收者是一个来自 Android API 的普通类：android.widget.EditText



带接收者的 lambda 是一个很棒的工具，提供了一种声明结构化 UI 元素的简洁方式。在代码中（与 XML 文件相比）声明它们让你能够抽出重复的逻辑并重用，就像你在 11.2.3 节中见到的那样。可以把 UI 和业务逻辑拆分到不同的组件中，但是所有的一切仍然是原汁原味的 Kotlin 代码。

11.5 小结

- 内部 DSL 是一种 API 设计模式，借助由多个方法调用组成的结构，可以使用这种模式来构建更表意的 API。
- 带接收者的 lambda 采用嵌套结构，来重新定义在 lambda 函数体中方法应该如何解析。
- 用作 (调用函数) 参数的带接收者的 lambda，其类型是扩展函数类型，并且这个调用函数在调用 lambda 时会为它提供一个接收者实例。
- 使用 Kotlin 内部 DSL 而不是外部模板或标记语言的好处是可以重用代码并创建抽象。
- 使用特殊命名的对象作为中缀调用的参数，允许你创建读起来就像英语的 DSL，而且不带额外的标点符号。
- 在基本数据类型上定义扩展让你能够为各种字面值（比如日期）创建可读的语法。
- 使用 `invoke` 约定，可以把任意对象当成函数一样调用。
- kotlin.html 库提供了用来构建 HTML 页面的内部 DSL，并且很容易扩展，用来支持各种前端开发框架。
- kotlintest 库提供的内部 DSL，支持在单元测试中使用可读断言语句。
- Exposed 库提供了使用数据库的内部 DSL。
- Anko 库为 Android 开发提供了各种工具，包括用来定义 UI 布局的内部 DSL。

附录A 构建Kotlin项目

本附录阐明了如何使用 Gradle、Maven 和 Ant 来构建 Kotlin 代码的项目，也涵盖了如何构建 Kotlin 的 Android 应用。

A.1 用Gradle构建Kotlin代码的项目

构建 Kotlin 项目的推荐系统是 Gradle。Gradle 是 Android 项目的标准构建系统，它还支持可以使用 Kotlin 的所有其他类型的项目。Gradle 具有灵活的项目模型，因为支持增量构建、长期构建过程（Gradle 守护进程）和其他高级技术，因此可以提供出色的构建性能。

Gradle 团队正在努力支持用 Kotlin 编写 Gradle 构建脚本，这将允许使用相同的语言编写应用程序及其构建脚本。在本文的编写过程中，这项工作仍在进行。可以在 https://github.com/gradle/gradle-script-kotlin 中找到有关的更多信息。在本书中，我们使用 Groovy 语法来编写 Gradle 构建脚本。

构建 Kotlin 项目的标准 Gradle 构建脚本如下所示：

```
buildscript {
    ext.kotlin_version = '1.0.6'

    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:" +
                  "kotlin-gradle-plugin:$kotlin_version"
```

指定要用的 Kotlin 的版本

给 Kotlin Gradle 插件增加构建脚本的依赖

```
    }
}

apply plugin: 'java'
apply plugin: 'kotlin'

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

使用 Kotlin Gradle 插件

给 Kotlin 标准库增加依赖

脚本在以下位置查找 Kotlin 源文件：

- 代码源文件位置：src/main/java 和 src/main/kotlin
- 测试源文件位置：src/test/java 和 src/test/kotlin

在大多数情况下，推荐将 Kotlin 和 Java 源文件放在同一目录中。尤其是当你把 Kotlin 引入现有项目时，使用单个源文件目录可以减少 Java 文件转换为 Kotlin 的阻力。

如果你使用了 Kotlin 反射，则需要另外添加一个依赖关系：Kotlin 反射库。为此，请在 Gradle 构建脚本的 dependencies 中添加以下内容：

```
compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
```

A.1.1 用 Gradle 来构建 Kotlin Android 应用

和普通的 Java 应用相比，Android 应用使用了不同的构建过程，所以需要使用不同的 Gradle 插件来构建。不是添加 apply plugin: 'kotlin'，需要把下面的代码添加到构建脚本中：

```
apply plugin: 'kotlin-android'
```

剩下的设置和其他非 Android 应用的设置一样。

如果你喜欢把 Kotlin 源代码放在特定目录下（如 src/main/kotlin），则需要注册它们，以便 Android Studio 识别它们为源目录。可以用以下代码段来实现：

```
android {
    ...

    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }
}
```

A.1.2 构建需要处理注解的项目

许多 Java 框架，特别是在 Android 开发中使用框架，都依赖注解处理在编译时生成代码。要在 Kotlin 中使用这些框架，需要在构建脚本中启用 Kotlin 注解处理。可以通过添加下面的代码来实现：

```
apply plugin: 'kotlin-kapt'
```

如果你试图引入 Kotlin 到现有的一个使用注解处理的 Java 项目中，那么需要删除 apt 工具的现有配置。Kotlin 注解处理工具包含了 Java 和 Kotlin 类的处理，如果同时有两个单独的注解处理工具会很多余。可以使用 kapt 依赖配置来配置注解处理所需的依赖关系：

```
dependencies {
    compile 'com.google.dagger:dagger:2.4'
    kapt 'com.google.dagger:dagger-compiler:2.4'
}
```

如果你对 androidTest 或 test 使用注解处理器，则对应的 kapt 配置应该分别为 kaptAndroidTest 和 kaptTest。

A.2 使用Maven来构建Kotlin项目

如果你喜欢使用 Maven 来构建项目，Kotlin 也是支持的。最简便的方式是使用 org.jetbrains.kotlin:kotlin-archetype-jvm 原型来创建 Kotlin 的 Maven 项目。对于现有的 Maven 项目，可以简单地通过在项目的 Kotlin IntelliJ IDEA 插件中选择 Tools > Kotlin > Configure Kotlin 添加对 Kotlin 的支持。

要手动给 Kotlin 项目添加 Maven 的支持，需要执行以下步骤：

1. 在 Kotlin 的标准库上添加依赖 (group ID：org.jetbrains.kotlin，artifact ID：kotlin-stdlib)。
2. 添加 Kotlin 的 Maven 的插件 (group ID：org.jetbrains.kotlin，artifact ID：kotlin-maven-plugin)，并配置它在 compile 和 test-compile 阶段执行。
3. 如果你喜欢把 Kotlin 代码和 Java 的源代码根目录分开，配置源文件目录。

由于篇幅的关系，在这里我们就不展示完整的 pom.xml 示例了，可以在在线文档中找到它们，网址为：https://kotlinlang.org/docs/reference/using-maven.html。

在混合的 Java / Kotlin 项目中，需要配置 Kotlin 插件，以便它在 Java 插件之前运行。这个很有必要。因为 Kotlin 插件可以解析 Java 源代码，而 Java 插件只能读

取 .class 文件。因此，需要在 Java 插件运行之前将 Kotlin 文件编译为 .class。可以在 http://mng.bz/73od 上找到如何配置的示例。

A.3 用Ant来构建Kotlin代码

Kotlin 提供了两种任务来使用 Ant 构建项目：<kotlinc> 任务用于编译纯 Kotlin 的模块，而 <withKotlin> 作为 <javac> 的扩展用于构建混合的 Kotlin/Java 模块。这里是使用 <kotlinc> 的一个最小示例：

```
<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="src"/>
        </kotlinc>
    </target>
</project>
```

定义 <kotlinc> 任务

用 <kotlinc> 构建一个单源代码目录，并打包结果到一个 jar 文件

Ant 任务 <kotlinc> 会自动添加标准库的依赖，所以你不必在配置时添加额外的参数。它也支持打包编译的 .class 文件到一个 jar 文件。这里是一个使用 <withKotlin> 任务来构建一个混合的 Java / Kotlin 模块的示例：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
             classpath="${kotlin.lib}/kotlin-ant.jar"/>
    <target name="build">
        <javac destdir="classes" srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

定义 <withKotlin> 任务

使用 <withKotlin> 任务来允许混合的 Kotlin/Java 编译

打包编译的 .class 文件到一个 jar 文件

和 <kotlinc> 不同的是，<withKotlin> 并不支持自动打包编译的类，所以这个示例中单独使用了 <jar> 任务来打包。

B Kotlin代码的文档化

本附录内容包括为 Kotlin 代码编写文档注释，并为 Kotlin 模块生成 API 文档。

B.1 给Kotlin代码写文档注释

为 Kotlin 声明编写文档注释的格式与 Java 的 Javadoc 类似，称为 KDoc。和 Javadoc 一样，KDoc 注释以 /** 开始，并使用以 @ 开头的标签来记录声明的特定部分。KDoc 和 Javadoc 的主要区别在于，用来写入注释的格式是 Markdown（https://daringfireball.net/projects/markdown）而不是 HTML。为了使写入文档注释更加容易，KDoc 还支持许多额外的约定，来引用文档元素，比如函数的形参。

这里是一个用 KDoc 给函数注释的简单示例。

代码清单 B.1　使用 KDoc 注释

```
/**
 * Calculates the sum of two numbers, [a] and [b]
 */
fun sum(a: Int, b: Int) = a + b
```

要在 KDoc 注释中引用一个声明，请把它们的名称放入括号内。该示例使用这个语法来引用正在记录的函数的参数，但也可以使用它来引用其他声明。如果你需要引用的声明是在包含 KDoc 注释的代码中导入，则可以直接使用它的名称。

否则，可以使用完全限定名称。如果需要为链接指定自定义标签，则使用两对括号，并将标签放在第一对中，声明名称放在第二对中：[an example] [com.mycompany.SomethingTest.simple]。

这里是一个更复杂的示例，展示了如何在注释中使用标签。

代码清单 B.2 在注释中使用标签

```
/**
 * Performs a complicated operation.
 *
 * @param remote If true, executes operation remotely
 * @return The result of executing the operation
 * @throws IOException if remote connnection fails
 * @sample com.mycompany.SomethingTest.simple
 */
fun somethingComplicated(remote: Boolean): ComplicatedResult { ... }
```

记录返回值

记录一个参数

记录一个可能的异常

包含指定函数的文本作为例子在文档中的文本

使用标签的一般语法与 Javadoc 完全相同。除了标准的 Javadoc 标签之外，KDoc 还支持许多额外的在 Java 中不存在的概念的标签，例如用于记录扩展函数或属性的接收者的 @receiver 标签。可以在 http://kotlinlang.org/docs/reference/kotlin-doc.html 找到所有的支持标签的列表。

@sample 标签可用于将指定函数的文本包含在文档文本中，作为使用正在记录的 API 的示例。标签的值是要包括的方法的完全限定名称。

一些 Javadoc 标记在 KDoc 中并不支持：

- @deprecated 被替换为 @Deprecated 注解。
- 在 Kotlin 中不支持 @inheritdoc，因为在 Kotlin 中，文档注释始终通过覆盖声明自动继承。
- @code、@literal 和 @link 将替换为相应的 Markdown 的格式。

请注意，Kotlin 团队首选的文档样式是直接在文档注释文本中记录函数的参数和返回值，如代码清单 B.1 所示。只有当参数或返回值具有复杂的语义并且需要从主要文档文本中清楚地分离时，才会使用标签，如代码清单 B.2 所示。

B.2 生成API文档

Kotlin的文档生成工具叫作Dokka：https://github.com/kotlin/dokka。和Kotlin一样，Dokka 完全支持跨语言的 Java/Kotlin 项目。它可以读取 Java 代码中的 Javadoc 注释，

以及 Kotlin 代码的 KDoc 注释，并生成覆盖模块的整个 API 的文档，而不管编写每个类的语言。Dokka 支持多种输出格式，包括纯 HTML、Javadoc 风格的 HTML（所有声明使用 Java 语法，并展示如何从 Java 访问 API），以及 Markdown。

可以从命令行运行 Dokka，或者把它作为 Ant、Maven 或 Gradle 构建脚本的一部分。推荐的运行 Dokka 的方式是把它添加到模块的 Gradle 构建脚本中。下面是 Gradle 构建脚本中 Dokka 的最起码的配置：

```
buildscript {
    ext.dokka_version = '0.9.13'          <-- 指定要用的 Dokka 的版本

    repositories {
        jcenter()
    }
    dependencies {
        classpath "org.jetbrains.dokka:dokka-gradle-plugin:${dokka_version}"
    }
}

apply plugin: 'org.jetbrains.dokka'
```

通过这样的配置，你就能通过运行 `./gradlew dokka` 来为模块生成 HTML 格式的文档。

可以在 Dokka 文档（https://github.com/Kotlin/dokka/blob/master/README.md）中找到有关指定其他生成选项的信息。该文档还展示了 Dokka 如何作为独立工具运行，以及如何集成到 Maven 和 Ant 构建脚本中。

C Kotlin生态系统

尽管 Kotlin 还非常年轻，但它已经有非常完整的生态系统了，包括库、框架和工具，这些绝大部分都是由外部开发社区创造的。在这个附录中，我们会给出一些帮助你继续探索 Kotlin 生态系统的方向。显然，一本书籍并不是描述快速增长的工具集合的完美媒介，所以我们首先给你介绍的就是能够找到更多最新信息的在线资源：https://kotlin.link/。

再一次提醒你，Kotlin 和整个 Java 库的生态系统完全兼容。在寻找能够解决问题的库的过程中，你不必只着眼于那些纯 Kotlin 编写的库——标准 Java 编写的库也应该在你的搜索范围内。现在我们来看看一些值得探索的库。有一些 Java 库特别为 Kotlin 提供了扩展，具有更简洁和更符合语言习惯的 API。只要有这样的扩展，你就应该毫不犹豫地使用它们。

C.1 测试

除了和 Kotlin 配合得不错的标准 JUnit 与 TestNG 外，下面两个框架提供了用 Kotlin 编写测试的更有表现力的 DSL.

- *KotlinTest*（https://github.com/kotlintest/kotlintest）——灵活的测试框架，它的灵感来自于 ScalaTest，在第 11 章提到过，支持多种不同的编写测试的样式
- *Spek*（https://github.com/jetbrains/spek）——属于 Kotlin 的 BDD 风格的测试

框架，由 JetBrains 发起，现在由社区维护

如果你对 JUnit 还算满意，只是对更有表现力的断言 DSL 感兴趣，可以试试 *Hamkrest*（https://github.com/npryce/hamkrest）。如果你在测试中还用到了 mock，绝对不应该错过 *Mockito-Kotlin*（https://github.com/nhaarman/mockito-kotlin），它解决了 mock Kotlin 类时出现的一些问题，还提供了一套更漂亮的用于 mock 的 DSL。

C.2 依赖注入

常见的 Java 依赖注入框架，比如 Spring、Guice 和 Dagger，都能很好地和 Kotlin 一起工作。如果你对原生的 Kotlin 方案感兴趣，试试 Kodein（https://github.com/SalomonBrys/Kodein），它提供了一套漂亮的 DSL 来配置依赖，而且它的实现也非常高效。

C.3 JSON序列化

如果你需要的是比第 10 章介绍的 JKid 的库更重量级的 JSON 序列化方案，你有很多选择。如果你更喜欢用 Jackson，*jackson-module-kotlin*（https://github.com/FasterXML/jackson-module-kotlin）提供了深度的 Kotlin 集成，包括了对数据类的支持。而 *Kotson*（https://github.com/SalomonBrys/Kotson）为 GSON 提供了一套漂亮的包装器。如果你追求的是轻量的纯 Kotlin 方案，可以试试 *Klaxon*（https://github.com/cbeust/klaxon）。

C.4 HTTP Clients

如果你需要用 Kotlin 构建 REST API 的 client，Retrofit（http://square.github.io/retrofit）是不二之选。它是完全兼容 Android 的 Java 库，能和 Kotlin 平滑对接。对于更底层的方案，试试 OKHttp（http://square.github.io/okhttp/）或者纯 Kotlin 的 HTTP 库 Fuel（https://github.com/kittinunf/Fuel）。

C.5 Web应用

如果你正在开发服务器端的 Web 应用，现在最成熟的选择莫过于像 Spring、Spark Java 及 vert.x 这样的 Java 框架。Spring 5.0 将会包括开箱即用的 Kotlin 支持和扩展。如果是在老版本的 Spring 项目上使用 Kotlin，可以在 Spring Kotlin（https://

github.com/sdeleuze/spring-kotlin）项目上找到额外的信息和一些辅助函数。vert.x 也提供了官方的 Kotlin 支持：https://github.com/vert-x3/vertx-lang-kotlin/。

而纯 Kotlin 的方案，可以考虑下面这些选项：

- *Ktor*（https://github.com/Kotlin/ktor）——JetBrains 的研究项目，探索如何借助符合语言习惯的 API 来构建一个现代的、全功能的 web 应用项目
- *Kara*（https://github.com/TinyMission/kara）——最初的 Kotlin Web 框架，JetBrains 和其他一些公司在正式产品中使用了它
- *Wasabi*（https://github.com/wasabifx/wasabi）——基于 Netty 构建的 HTTP 框架，具备表现力丰富的 Kotlin API
- *Kovert*（https://github.com/kohesive/kovert）——基于 vert.x 构建的 REST 框架

如果你需要生成 HTML，看看第 11 章中讨论过的 kotlinx.html（https://github.com/kotlin/kotlinx.html）。或者，如果你更喜欢传统的方式，可以使用像 Thymeleaf（www.thymeleaf.org）这样的 Java 模板引擎。

C.6 访问数据库

除了像 Hibernate 这样的传统的 Java 选项外，你有许多可以满足访问数据库需要的 Kotlin 特定的选择。我们经验最丰富的是 *Exposed*（https://github.com/jetbrains/Exposed），一个本书中多次讨论过的 SQL 生成框架。其他一些替代方案列在 https://kotlin.link 上。

C.7 工具和数据结构

现今最火热的新编程范式莫过于响应式编程。JVM 上响应式编程库的事实标准 RxJava（https://github.com/ReactiveX/RxJava）官方提供了 Kotlin 绑定（https://github.com/ReactiveX/RxKotlin）。

下面这些库提供的工具和数据结构，你可能会觉得对项目有帮助：

- *funKTionale*（https://github.com/MarioAriasC/funKTionale）——实现了各种函数式编程原语（比如偏函数应用）
- *Kovenant*（https://github.com/mplatvoet/kovenant）——Kotlin 和 Android 的 promise 实现

C.8 桌面应用编程

如果现在你还在 JVM 上构建桌面应用，最有可能用的就是 JavaFX。TornadoFX（https://github.com/edvin/tornadofx）为 JavaFX 提供了一套强大的 Kotlin 适配器，让你能自然地使用 Kotlin 完成桌面应用开发。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Kotlin实战

开发者想完成他们的工作——同时越省事越好。使用Kotlin编码就意味着省事。Kotlin编程语言提供了富有表现力的语法，强大直观的类型系统和美妙的工具支持，还有与现存Java代码、库及框架无缝的互操作性。Kotlin可以被编译成Java字节码，所以你可以在所有使用Java的地方使用它，包括Android在内。借助高效的编译器和标准库，Kotlin在运行时几乎不用承受任何额外开销。

《Kotlin实战》教会你使用Kotlin语言来开发达到产品级品质的应用。这本书为具备一定Java经验的开发者编写，包含了丰富的示例；与大多数介绍编程语言的书籍相比更加深入，涵盖了非常有趣的话题，例如怎样构建使用自然语言语法的DSL。两位作者是Kotlin的核心开发者，所以你完全可以相信书中最细枝末节的内容都无比的精确。

本书包括：

◎在JVM上进行函数式编程
◎编写整洁并符合语义习惯的代码
◎结合运用Kotlin和Java
◎领域特定语言

本书适合有一定Java经验的开发者。

作者Dmitry Jemerov和Svetlana Isakova是来自JetBrains的Kotlin核心开发者。

"既阐释了高级概念，也提供了足够深入的细节。"

—— 摘自 Kotlin 首席设计师 Andrey Breslav 为本书作的序

"这本书保持了 Manning 实战系列的水准，满足了你快速提高生成力的所有需要。"

——Kevin Orr, Sumus Solutions

"有这本书指导你，Kotlin学起来有趣又简单!"

——Filip Pravica, Info.nl

"写得非常全面，非常好，浅显易懂。"

——Jason Lee, NetSuite

MANNING

责任编辑：张春雨
封面设计：李　玲

上架建议：移动开发/Android

ISBN 978-7-121-32158-0

9 787121 321580 >

定价：89.00元