移动开发技术丛书

Kotlin 移动和服务器端应用开发

[美] 布雷特·麦克劳克林(Brett McLaughlin) 著
任 强 译

清華大學出版社
北 京

北京市版权局著作权合同登记号　图字：01-2021-3640

图书在版编目(CIP)数据

Kotlin移动和服务器端应用开发 / (美)布雷特 • 麦克劳克林(Brett McLaughlin) 著；任强译. —北京：清华大学出版社，2022.8
(移动开发技术丛书)
书名原文：Programming Kotlin Applications，Building Mobile and Server-Side Applications with Kotlin
ISBN 978-7-302-61405-0

I. ①K… II. ①布… ②任… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2022)第 136596 号

责任编辑：王　军
装帧设计：孔祥峰
责任校对：成风进
责任印制：朱雨萌

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：小森印刷霸州有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张**：23.5　　**字　　数**：487 千字
版　　次：2022 年 10 月第 1 版　　**印　　次**：2022 年 10 月第 1 次印刷
定　　价：99.80 元

产品编号：091911-01

译 者 序

Kotlin 自 2011 年发布以来，备受关注并持续发展，时至今日已形成了强大的生态环境。在大家逐渐接受 Kotlin 的同时，Kotlin 能否取代 Java 也成为了饱受争议的话题。在 GoogleI/O 2019 会议上，Google 宣布将 Kotlin 作为 Android 开发的首选语言，这在 Android 移动开发领域再度掀起了学习 Kotlin 的热潮。

目前市面上关于 Kotlin 的书籍很多，相信广大读者都想在其中找到适合自己的入门书籍，但这并非易事。本书作者 Brett McLaughlin 在技术领域从业超过 20 年，近 10 年来出版了许多计算机编程书籍。《Kotlin 移动和服务器端应用开发》采用了一种全新的方式来教你如何学习 Kotlin 开发，着重于 Kotlin 语言的基础知识，带你进入 Kotlin 的世界；还教你如何在 Kotlin 中处理泛型、Lambda 表达式和作用域函数等，让你领略到 Kotlin 所带来的快乐与高效。

翻译本书的过程也是我重新审视并巩固 Kotlin 知识的过程。如果这是你的第一本 Kotlin 书，可以按照作者安排的章节顺序阅读；如果你对 Kotlin 已有所了解，那么可以完全自主选择感兴趣的章节阅读。无论你是 Kotlin 新手还是多年的老手，相信本书都会成为你的良师益友。

在这里要感谢清华大学出版社给我这个翻译本书的机会。编辑们为本书的翻译投入了巨大的热情并付出了很多心血，没有他们的帮助和鼓励，本书不可能顺利付梓。

在本书的翻译过程中虽力求“信、达、雅”，但鉴于本人水平，失误在所难免，如有任何意见和建议，请不吝指正。

最后，希望读者通过阅读本书能早日步入 Kotlin 的世界，领略 Kotlin 之美。

任 强

作 者 简 介

Brett McLaughlin 在技术工作和技术写作方面拥有超过 20 年的经验。目前，Brett 专注于云计算和企业计算。他是一名值得信赖的知名人士，能将繁杂的云概念转化为清晰的执行层愿景从而帮助公司实现云迁移，尤其是 Amazon Web Services。他的日常工作就是与需要了解云的关键决策者一起，建立并领导开发和运维团队，带领他们与不断变化的云计算空间进行交互。他最近领导了 NASA 的 Earth Science 计划和 RockCreek Group 集团的金融平台的大规模云迁移。Brett 目前还是电子商务平台提供商 Volusion 的首席技术官。

技术编辑简介

Jason Lee 是一位软件开发者，他精通各种编程语言，拥有超过 23 年的程序开发经验，主要编写从移动设备到大型计算机上的软件。在过去的 15 年里，他一直从事 Java/Jakarta EE 领域的工作，致力于应用服务器、框架以及面向用户的应用程序。最近，他作为后端工程师主要使用 Kotlin 语言通过 Quarkus 和 Spring Boot 等框架构建系统。他是 *Java 9 Programming Blueprints* 的作者、前 Java User Group 的主席、会议发言人和博客作者。业余时间，他喜欢和妻子以及两个儿子一起读书、弹贝司或吉他、跑步等。你可以通过 twitter.com/jasondlee 或他的博客 jasondl.ee 联系他。

致　　谢

过去我经常看电影，并惊讶于最后屏幕上滚动的数百个名字。一部电影怎么会涉及这么多人？

在我写完本书后，我终于明白了其中的原因。

Carole Jelen 是我在 Waterside 的经纪人，她回复了我的邮件并随即帮我联系出版社，在那时我真的需要有人帮助我找到重回出版业的路。我非常感谢她！

在 Wiley，Brad Jones 展现了他前所未有的耐心。谢谢你，Brad！Barath Kumar Rajasekaran 处理了无数个小细节，Pete Gaughan 和 Devon Lewis 则将一切保持在正轨上。Christine O'Connor 负责处理制作，Jason Lee 捕捉到了文本中的技术错误，让本书的质量上了一个台阶。说真的，尤其是 Jason，他用敏锐的眼光使本书的质量更上一层楼。

与往常一样，我的-家庭所付出的代价是最高的。漫长的一天又一天，而不只是几个周末和晚上，家人持续的支持让我继续前进。我的妻子 Leigh 是最棒的，而我的孩子 Dean、Robbie 和 Addie 总是会把我的漫长写作过程变成一种乐趣。

大家一起享用早午餐吧！香槟和玉米卷我请客。

Brett McLaughlin

前　言

数十年来，Java 编程语言一直是编译语言的主导力量。尽管有很多替代方案，但从桌面系统到服务器端再到移动端，Java 仍然是许多应用的核心。对于 Android 移动开发来说尤其如此。

不过最终出现了一位真正的竞争者：Kotlin。它是由 JetBrains (www.jetbrains.com) 设计并开发的一门现代编程语言。它不是 Java，但可以完全与之交互操作。Kotlin 十分类似于 Java，但它对 Java 进行了一些很好的改进，对于已经熟悉 Java 语言的开发者来说它很容易学习。

此外，Kotlin 是一门成熟的编程语言。它并不仅限于移动应用的开发，也并非专注于某个特定应用的可视化语言。Kotlin 支持：

- 继承、接口、实现以及类的层次结构
- 简单和复杂的控制及流程结构
- Lambda 和作用域函数
- 对泛型提供丰富支持的同时仍然保持强类型
- 惯用的开发方法，让 Kotlin 有了自己的“风格”

虽然 Kotlin 是一门新语言，但它并不让人感到陌生。这在很大程度上是因为它构建在 Java 之上，它反思并吸取了成千上万用 Java(和其他语言)编写代码的程序员所经历的教训，并使它们成为语言的一部分，强制执行强类型和严格的编译器也许会让用户需要一些时间才能适应，但通常会生成更干净、更安全的代码。

对继承的理解是学习 Kotlin 的一个重点，因此也是本书要讲解的一个重点。无论你是使用第三方的包，采用标准的 Kotlin 库，还是构建自己的程序，都需要对类之间的相互关系、子类化的工作方式以及如何使用抽象类和接口来定义行为并确保实现行为等主题有相当透彻的理解。当你读完本书后，将非常熟悉类、对象以及继承树的构建。

Kotlin 网站(kotlinlang.org)将 Kotlin 描述为“一门让开发者更快乐的现代编程语言”。有了 Kotlin 和本书，你将在 Kotlin 开发中更快乐、更高效。

本书内容

本书采用一种全面的方法来教你学习 Kotlin 编程语言，让你从一个初学者成长为自信、成熟的 Kotlin 开发者。当阅读完本书后，你将能在各种环境下(从桌面系统到服务器端再到移动端)编写 Kotlin 应用。

本书会教我如何用 Kotlin 编写移动应用吗

会的，但要想用 Kotlin 构建丰富的移动应用，仅靠本书还不够。Kotlin 是一门丰富的语言，有许多图书主要介绍构建移动应用所需的各种包，但本书基本上是学习 Kotlin 的入门级图书。你将了解 Kotlin 如何处理泛型、继承和 Lambda，这些都是移动开发的关键概念。

然后，你可以将这些概念扩展到移动应用开发中。可以轻松地将 Android 相关软件包的细节补充到你的 Kotlin 基础知识中，具备 Kotlin 的基础知识后你将能更有效地使用这些移动软件包。

如果你渴望更迅速地开始你的移动应用开发旅程，不妨再选择一本专注于 Kotlin 移动应用开发的书，然后在这两本书之间来回切换。阅读完本书的第 1 章后，你可以对侧重移动应用开发的书重复同样的过程。你将多次切换上下文，但这样做能够同时学到基础知识和特定的移动开发技术。

本书涵盖以下内容。

第 1 章　对象的世界

本章介绍如何安装 Kotlin，以及如何编写第一个 Kotlin 程序。你将从一开始就了解函数，以及如何通过“Hello, World!”应用与命令行交互。还将立即了解 Kotlin 中对象和类的作用，并进一步了解类、对象，以及对象实例的概念。

第 2 章　Kotlin 很难出错

本章深入探讨 Kotlin 的一个显著特点：对类型安全的严格立场。你将了解 Kotlin 的类型，并学习如何为正确的任务选择正确的类型。还将熟悉 val 和 var，以及 Kotlin 是如何允许更改的。

第 3 章　Kotlin 非常优雅

像任何面向对象的语言一样，使用 Kotlin 编程的大部分工作都是编写类。本章深入研究 Kotlin 中的类，并查看所有 Kotlin 对象的基本构建块。还将覆盖一些函数，并深入了解一些最基本的 Kotlin 函数：equals()和 hashCode()。

第 4 章　继承很重要

本章开启学习 Kotlin 中的继承之旅。你将了解 Kotlin 的构造函数以及相对独特的

次构造函数的概念。还将了解更多关于 Any 类的知识，知道继承对于所有 Kotlin 开发来说确实是必不可少的，并理解为什么编写好的超类是你要培养的最重要的技能之一。

第 5 章　List、Set 和 Map

本章(简要地)从类和继承转移到 Kotlin 集合。你将在开发中反复使用这些集合类，因此了解 Set 与 Map 的区别以及它们与 List 的区别非常重要。你还将深入研究 Kotlin 的可变性和不可变性(数据可以更改或不能更改)，以及迭代所有类型集合的各种方法。

第 6 章　Kotlin 的未来是泛型

泛型在大多数编程语言中都是一个难点。了解它们需要对语言的构建方式有深刻的理解。本章将深入探讨这些问题，了解为什么使用泛型为你构建可以在各种上下文中使用的类提供了更大的灵活性。本章还将介绍协变、逆变和不变。这些可能不是热门话题，但它们将是正确使用泛型构建程序的关键，也将加深你对继承以及子类的理解。

第 7 章　控制结构

控制结构是大多数编程语言的基础。本章将详细介绍 if/else、when、while 和 do 控制结构。在这一过程中，你将重点控制应用程序或应用程序集的流程，同时学会处理这些结构的语义和机制。

第 8 章　数据类

本章将介绍数据类，这是另一个非常酷的 Kotlin 概念。虽然不仅仅针对 Kotlin，但是你会发现数据类提供了一个快速而灵活的选项，比老旧的语言更能有效地表示数据。你还将真正推进数据类，超越简单的数据对象，探究构造函数、重写属性，以及使用数据类进行子类化和扩展。

第 9 章　枚举和密封类，以及更多专业类

本章将介绍枚举，这是一种远胜于字符串常量的方法。你将了解为什么将字符串用于常量值是非常糟糕的，以及枚举如何为你提供更大的灵活性和类型安全性，以及如何使代码更易于编写。你还将从枚举转而学习密封类，这是 Kotlin 的一个特别炫酷的特性，它可以进一步增强你对枚举概念的理解。你还将深入研究相关的对象和工厂，所有这些都有助于你使用一种健壮的类型安全的编程方法，而以往只能使用字符串类型。

第 10 章　函数

本书到此才用一章篇幅重点讨论函数，这似乎有些奇怪。然而，与任何学科中的大多数基础知识一样，你必须一次又一次地重温基础知识，弥补弱点，了解细微差别。本章仅通过函数来实现这一点。你将更深入地了解参数的工作方式，以及 Kotlin 在处理函数的输入输出数据时提供了多少可选项。

第 11 章　编写地道的 Kotlin 代码

像所有编程语言一样，Kotlin 提供了一些经验丰富的程序员能反复使用的固定模

式。本章讨论了这些模式以及 Kotlin 的一些习语。刚开始你会以 Kotlin 规定的样式编写 Kotlin 程序，慢慢地你会了解你可灵活选择如何让 Kotlin 程序更符合“你的风格”。

第 12 章　再次体会继承

本章实际上又是关于继承的介绍。将介绍你已经了解的抽象类和超类，并介绍接口和实现。还将介绍委托模式，这是一种常见的 Kotlin 模式，它可以帮助你进一步利用继承，能够提供比继承本身更大的灵活性。

第 13 章　学习 Kotlin 的下一步

没有一本书能教会你一切，本书当然也不例外。不过，在你的 Kotlin 开发之旅中，有一些现成的资源可作为你学习的下一站。本章为你提供了一些新的起点，以帮助你继续了解 Kotlin 的特定领域。

如何获取源代码

可以通过扫描本书封底的二维码来下载运行本书中的示例所需的项目文件。

目　录

```
fun main() {
  val brian = User("Brian", "Truesby")
  val rose = User("Rose", "Bushnell")

  val attendees: MutableList<User> = mutableListOf(brian, rose)

  attendees.forEach {
    user -> println("$user is attending!")
  }
}
```

请花几分钟的时间仔细阅读这段代码。即使你从来没有见过任何 Kotlin 代码，也能大致了解这段代码的作用。首先，它定义了一个 User 类(实际上这是一种特殊类，即数据类，稍后再详细说明)。然后定义了 main()函数，这是标准内容。接下来定义了两个变量(val)，它们都是之前定义的 User 类的实例。接着创建了一个列表 attendees，其中放入了刚才创建的两个 User 对象。最后是遍历这个列表的循环，用于打印列表中的每一项。

运行该代码，你会获得类似下面的结果：

```
User(firstName=Brian, lastName=Truesby) is attending!
User(firstName=Rose, lastName=Bushnell) is attending!
```

无论你是刚开始编写代码的新手还是经验丰富的 Java 老手，都可能会觉得其中的部分代码看起来有点奇怪。更重要的是，很有可能你不知道如何编译并运行这段代码。没关系，我们稍后会介绍。

注意：

不必非得理解代码清单 1.1 中的代码。虽然本书假设你具备一定的编程基础(当然，如果具备 Java 基础就能更快地学会 Kotlin)，但是没有任何编程基础也没关系，通过持续阅读本书并且推敲代码示例也能逐渐理解代码清单 1.1 中的所有内容，甚至更多。努力学习并持之以恒，你很快就能使用 Kotlin 编写程序。

不过，就目前而言，重点是：Kotlin 实为一门易于理解、简洁明了并且使用起来相当有趣的语言。基于这一点，下面我们首先了解一些基础知识，这样就可以开始编写代码了，而不仅仅是阅读代码。

1.2 什么是 Kotlin

Kotlin 是一门开源的编程语言。它是一门非常显著的、静态类型的和面向对象的语言。静态类型是指在你编写代码并编译时变量类型就已确定，并且这些类型是固定

```
fun main() {
    var car = Honda("Accord", "blue")
    car.start()
    car.drive("W", 60)
    car.stop()
}
```

12.1.4 子类履行通过抽象类编写的契约

现在你可以看到，一个抽象的类是没有实际价值的，直到它被子类化。但它确实有用：大多数抽象类都是为了用多个子类来子类化。除非你打算只创建 Honda 子类，否则你不会不创建 Car 类。

1. 子类应该改变行为

但是，当你开始看到多个子类时，有一些事情需要注意。首先，你应让子类具有不同的行为。要明白这点，请先看反例。代码清单 12.12 创建了另一个 Car 的子类，名为 Porsche。

代码清单 12.12 建造另一个 Car 的子类

```
package org.wiley.kotlin.car

class Porsche(model: String, color: String) : Car(model, color) {

    override var maxSpeed : Int = 212

    override fun start() {
       println("Starting up the Porsche ${model}!")
    }

    override fun stop() {
       println("Stopping the Porsche ${model}!")
    }

    override fun drive(direction: String, speed: Int) : Int {
        println("The Porsche ${model} is driving!")
        return speed / 60
    }
}
```

这实际上是一个糟糕的子类示例，不是因为 Porsche 类本身，而是要结合 Honda 类一起来看。Honda 和 Porsche 在 start()、stop()和 drive()函数中做了完全相同的事。

第1章
对象的世界

本章内容

- 初探 Kotlin 语法
- Kotlin 简史
- Kotlin 与 Java 的异同
- 设置、编写并运行 Kotlin
- 第一个 Kotlin 程序
- 为什么对象很有用且很重要

1.1 Kotlin：一门新兴的编程语言

归根到底，Kotlin 不过是另外一门编程语言。如果你已经在从事程序设计或编写代码的工作，那么你很快就可以学会 Kotlin，因为它和你已经在做的事情有很多共同之处。如果你使用过面向对象的编程语言，并且使用过 Java 编写代码，就会对 Kotlin 非常熟悉。虽然 Java 和 Kotlin 有一些区别，但都非常实用。

如果你刚接触 Kotlin，也没关系，它非常适合成为你的第一门编程语言。该语言非常清晰，没有太多古怪的习惯用法(比如 Ruby 或 LISP)，并且很有条理。你很快就能上手，并对它很满意。

事实上，Kotlin 语言十分简单明了。在此，我们将暂时抛开有关 Kotlin 的大量解释和历史，而是直接跳到基础的 Kotlin 代码(见代码清单 1.1)。

代码清单 1.1　一个使用了类和列表的简单 Kotlin 程序

```
data class User(val firstName: String, val lastName: String)
```

```
fun main() {
  val brian = User("Brian", "Truesby")
  val rose = User("Rose", "Bushnell")

  val attendees: MutableList<User> = mutableListOf(brian, rose)

  attendees.forEach {
    user -> println("$user is attending!")
  }
}
```

请花几分钟的时间仔细阅读这段代码。即使你从来没有见过任何 Kotlin 代码，也能大致了解这段代码的作用。首先，它定义了一个 User 类(实际上这是一种特殊类，即数据类，稍后再详细说明)。然后定义了 main()函数，这是标准内容。接下来定义了两个变量(val)，它们都是之前定义的 User 类的实例。接着创建了一个列表 attendees，其中放入了刚才创建的两个 User 对象。最后是遍历这个列表的循环，用于打印列表中的每一项。

运行该代码，你会获得类似下面的结果：

```
User(firstName=Brian, lastName=Truesby) is attending!
User(firstName=Rose, lastName=Bushnell) is attending!
```

无论你是刚开始编写代码的新手还是经验丰富的 Java 老手，都可能会觉得其中的部分代码看起来有点奇怪。更重要的是，很有可能你不知道如何编译并运行这段代码。没关系，我们稍后会介绍。

注意：

不必非得理解代码清单 1.1 中的代码。虽然本书假设你具备一定的编程基础(当然，如果具备 Java 基础就能更快地学会 Kotlin)，但是没有任何编程基础也没关系，通过持续阅读本书并且推敲代码示例也能逐渐理解代码清单 1.1 中的所有内容，甚至更多。努力学习并持之以恒，你很快就能使用 Kotlin 编写程序。

不过，就目前而言，重点是：Kotlin 实为一门易于理解、简洁明了并且使用起来相当有趣的语言。基于这一点，下面我们首先了解一些基础知识，这样就可以开始编写代码了，而不仅仅是阅读代码。

1.2 什么是 Kotlin

Kotlin 是一门开源的编程语言。它是一门非常显著的、静态类型的和面向对象的语言。静态类型是指在你编写代码并编译时变量类型就已确定，并且这些类型是固定

不变的。这也意味着 Kotlin 必须被编译。Kotlin 是面向对象的，这意味着该语言具有类和继承特性，使其成为 Java 和 C++开发者熟悉的语言。

Kotlin 实际上是由一群 JetBrains IDE 的开发者创建的，非常像 Java 的自然进化。它诞生自 2011 年，于 2016 年正式发布。这意味着它是一门新兴语言，这既可能是它的优势，也可能是劣势。Kotlin 是一门现代语言，可以在 Java 虚拟机(Java Virtual Machine，JVM)中运行，甚至可以编译为 JavaScript——这是一个很棒的功能，我们稍后介绍。

需要注意的是，Kotlin 也是开发 Android 应用程序的最佳语言。事实上，它对 Java 的许多增强功能都能找到对应的 Android 使用场景。即使你从未打算编写移动应用，也会发现 Kotlin 非常适合你的技术装备库，还非常适合服务器端的开发。

相比于 Java，Kotlin 增加了什么

这是一个非常好的问题，需要很长的篇幅来回答。事实上，我们将占用本书的大部分篇幅以多种形式来回答这个问题。但对大多数人来说，与 Java 相比，Kotlin 增加或更改了以下关键特性。

> **注意：**
> 如果你是 Kotlin 新手，或者没有 Java 背景，请跳过以下部分。

- Kotlin 在几乎所有场景下摒弃了 NullPointerException(以及可为空的变量类型)。
- Kotlin 支持扩展功能，而不必完全覆盖父类。
- Kotlin 不支持检测异常(你可能并不认同这是进步，因此值得一提)。
- Kotlin 增加了函数式编程的部分，例如，广泛使用的 Lambda 支持和惰性评价。
- Kotlin 定义了数据类，无须编写基本的 Getter 和 Setter。

当然，Kotlin 新增的特性远不止这些，你很快就可以了解到 Kotlin 不只是一个略有不同的 Java 版本。它追求不同与更好，在许多方面它都如此。

1.3　Kotlin 是面向对象的语言

至此，大多数书或教程会让你编写一个简单的“Hello, World”程序，但本书假设你想更进一步，因此，不妨从使用 Kotlin 创建一个对象开始。

简单来说，一个对象就是某事物的程序化表现。最理想的情况是，该事物就是一个真实世界中的对象，如一辆车、一个人或者一件产品。例如，可以创建一个对象来描述人，代码如下：

```
class Person {
```

```
  /* This class literally does nothing! */

}
```

就是这样。你现在可以新建一个 Person 类型的变量，代码如下：

```
fun main() {
    val jennifer = Person()
}
```

完整的代码如代码清单 1.2 所示。

代码清单 1.2　用 Kotlin 编写的一个空对象(和一个使用此空对象的 main()函数)

```
class Person {

  /* This class literally does nothing! */

}

fun main() {
    val jennifer = Person()
}
```

说真的，现在这段代码毫无用处。虽然它没有任何作用，但它是面向对象的。尽管如此，在我们对它进行改进之前，要先学会如何运行它。

1.4　设置 Kotlin 环境

运行 Kotlin 程序相对而言并不难，如果你是 Java 老手那就更容易。你需要安装 Java 虚拟机以及 Java 开发工具包(Java Development Kit，JDK)。接下来，还需要从众多支持 Kotlin 的 IDE 中选一个来安装。下面快速介绍一下这个安装过程。

1.4.1　安装 Kotlin(及 IDE)

支持 Kotlin 的最简单 IDE 就是 IntelliJ IDEA，从版本 15 开始，IntelliJ 就与 Kotlin 捆绑在一起。另外，由于 IntelliJ 实际上来自于 JetBrains，因此你将得到一个由 Kotlin 设计者开发的 IDE。

1. 安装 IntelliJ

可以从 www.jetbrains.com/idea/download 下载 IntelliJ。该网页会根据你的操作系统

跳转到不同的下载页面，如图 1.1 所示，本书中的多数示例都是基于 Mac OS X 的。下载免费的社区版本不需要任何费用。下载可能需要一些时间，完成下载后就可以进行安装，如图 1.2 所示，安装中包含了 Java 运行环境(Java Runtime Environment，JRE)和 JDK。

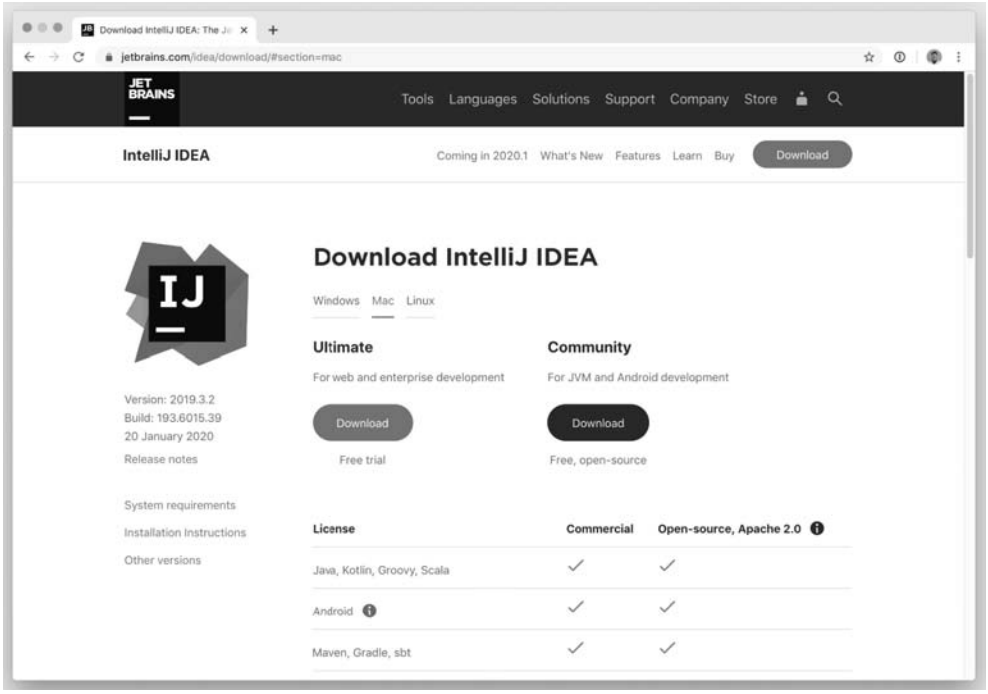

图 1.1　从 JetBrains 下载页面下载 IntelliJ

图 1.2　IntelliJ 预先打包了与系统对应的安装过程

注意：

IntelliJ 并不是唯一支持 Kotlin 的 IDE，支持 Kotlin 的 IDE 还在不断增加。其中值得一提的是 Android Studio (developer.android.com/studio/preview/index.html)和 Eclipse (www.eclipse.org/downloads)。尤其是 Eclipse 非常受欢迎，但 IntelliJ 仍是一个很好的选择，毕竟它和 Kotlin 都是源于 JetBrains。

注意：

Mac OS X 上的 IntelliJ “安装流程”非常简单：只需要将安装包(显示为一个图标)拖到 Applications 文件夹中。你需要将安装包移到该文件夹下运行或拖动安装包图标至你的程序坞(Dock)，我就是这么做的。

若是在 Windows 上，可以下载可执行文件并运行。如果愿意，可以选择在桌面上创建快捷方式。

在这两种情况下，都可以使用 JetBrainsToolbox(随 JetBrains Kotlin 附带)确保安装的是最新版本，并在软件有可用更新时升级。

有很多选项可用于设置 IDE。对于 IntelliJ，可以选择一个 UI 主题、一个启动脚本(建议接受默认选项并让它创建脚本)、默认插件，以及一组特色插件。你可以快速单击这些选项，然后重启 IDE。你将看到一个如图 1.3 所示的欢迎界面，然后单击其中的 Create New Project 选项。

图 1.3　通常可以从头创建项目，或者从代码仓库(如 GitHub)导入

警告：

如果你接受了 Mac OS X 上的默认位置，则可能需要使用高级权限来安装 IntelliJ 创建的启动脚本(Launcher Script)。

确保在创建项目时选择了 Kotlin/JVM 选项，如图 1.4 所示。

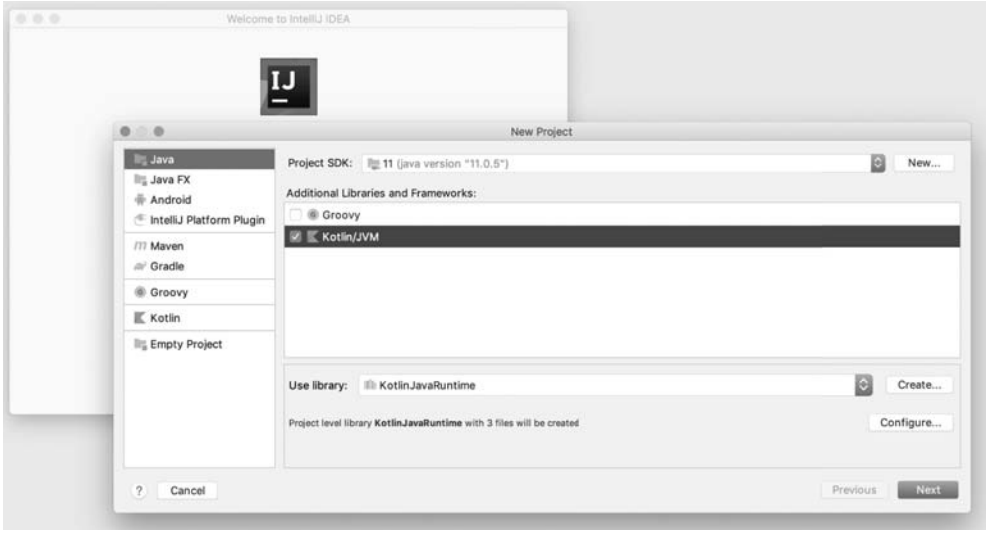

图 1.4　IntelliJ 让 Kotlin 开发变得简单，会提示你创建一个包含 Kotlin 库的新项目

2. 创建 Kotlin 程序

项目创建并运行后，创建一个新的 Kotlin 文件。在左侧导航窗格中找到 src/文件夹，右击，然后选择 New | Kotlin File/Class，如图 1.5 所示。你可以输入代码清单 1.2 所示的代码，IntelliJ 会自动格式化代码并添加合理的语法高亮显示，如图 1.6 所示(感谢 IntelliJ)。

注意：

你的 IDE 设置可能与我的完全不同。如果找不到 src/文件夹，可能需要单击 IDE 左侧的 Project 来显示文件夹，还可能需要单击项目名称。

注意：

从现在开始，通常不再将代码放在任一个 IDE 中显示。这样就可以使用自己选择的 IDE(或命令行)，因为无论使用什么 IDE 都应得到同样的结果。

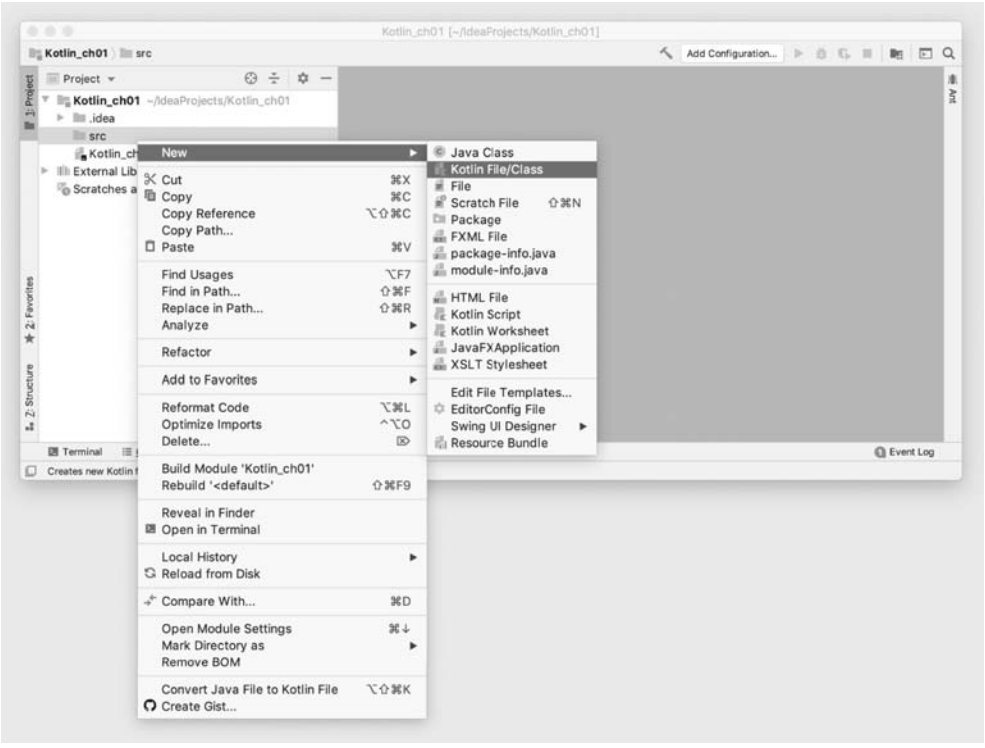

图 1.5　Kotlin 代码应该放在 src/文件夹中

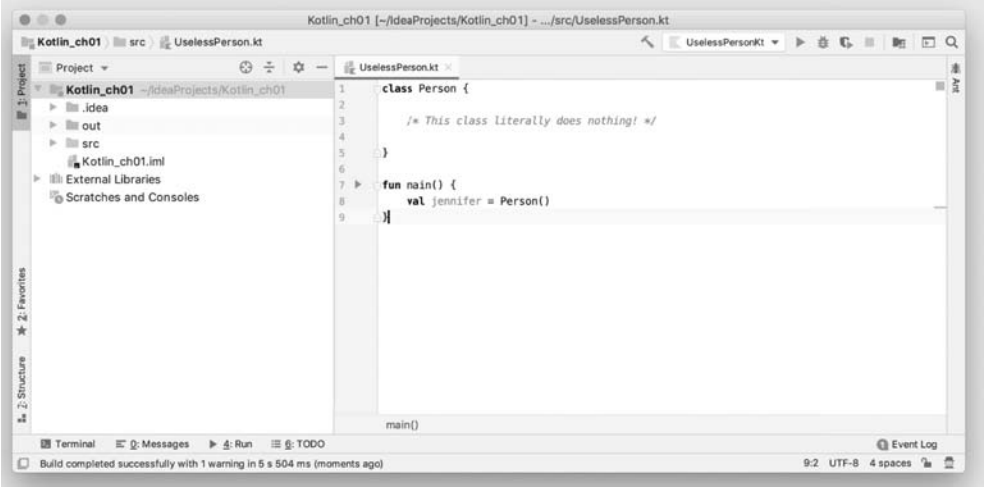

图 1.6　IntelliJ 会自动格式化代码并添加合理的语法高亮显示

3. 编译并运行 Kotlin 程序

接下来就是自己编译并运行程序了。这很简单，当文件中定义了 main()函数时，IntelliJ 会提供一个便捷的绿色小箭头供单击。将鼠标光标放在上面并单击即可(如图 1.7 所示)。然后你可以选择 Run 按钮以及文件名(我用的文件名是 UselessPerson)，这样程序将被编译并运行，输出将显示在 IDE 底部的新窗格中，如图 1.8 所示。

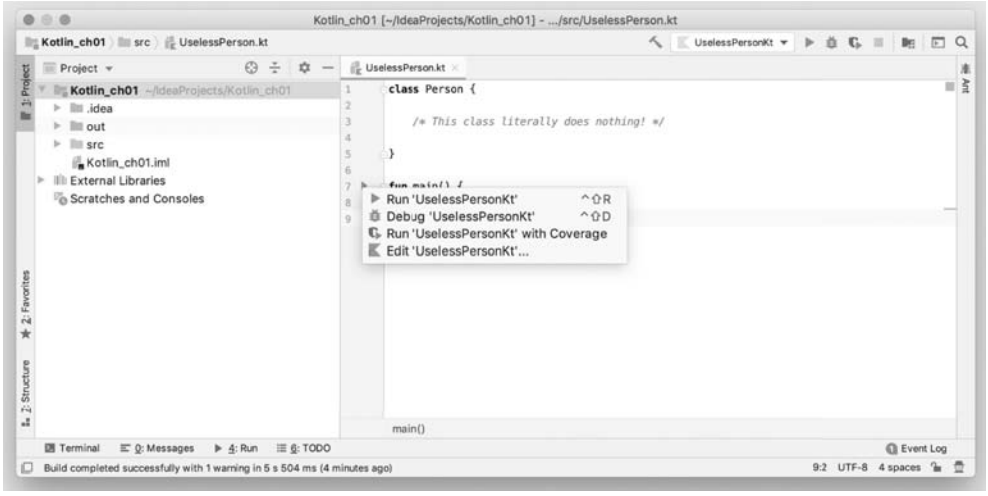

图 1.7　可单击绿色的 Run 按钮，并选择第一个选项编译并运行代码

图 1.8　你的程序在相应的窗口中没有输出结果(很快就会有结果输出)

在这种情况下，你应该不会遇到任何错误，但也不会得到任何输出。没关系，我们很快会对该程序进行修复。

4. 修复出现的任何错误

在继续改进前面那个无用的 Person 类之前，还有最后一个注意事项。IntelliJ 和其他 IDE 都非常擅长在代码出问题时提供可视化的提示。例如，图 1.9 展示了 IntelliJ 在编译同一程序出错时的情况。在此例中，第 8 行缺少了左右圆括号。你将在代码编辑器中看到一个橙色指示器，在输出窗口中看到一条错误信息，提示第 8 行(第 20 列)有错误。

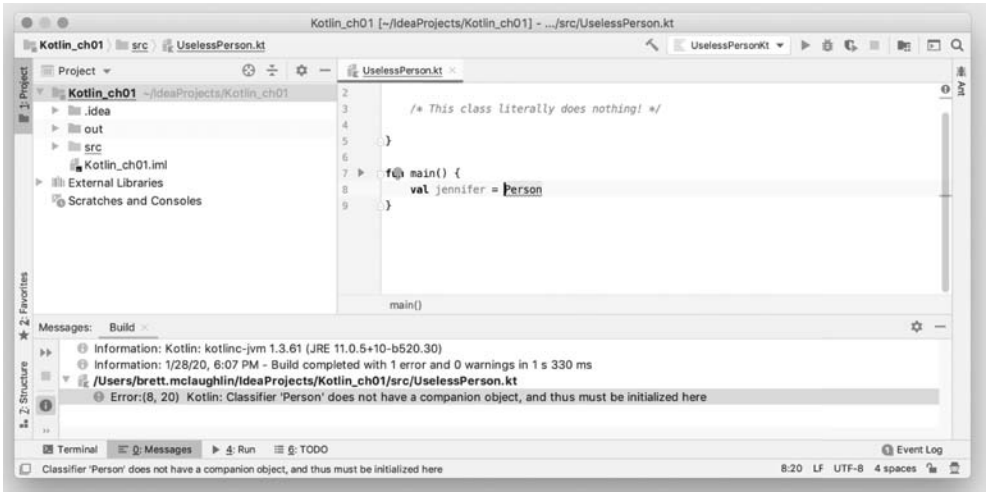

图 1.9　出色的 IDE 有助于快速发现和修复错误

然后就可以轻松地修复错误并重新编译程序。

1.4.2　安装 Kotlin(并使用命令行)

行家通常有一种倾向，即无论做什么都想使用命令行。Kotlin 当然有命令行。因为它在某种意义上“就是 Java”，它通过 JVM 和 JDK 运行，使用它可以达到事半功倍的效果。

1. 在 Windows 上安装 Kotlin 命令行

对于 Windows 用户，首先需要一个 JDK。可以从 Oracle Java 下载页面(www.oracle.com/technetwork/java/javase/downloads)下载 JDK。不同版本的下载文件都附有易于操作的安装说明。

有了 Java 后，你需要从 GitHub 获得最新的 Kotlin 版本。可以从 github.com/JetBrains/kotlin/releases/latest(该链接将重定向到最新的版本)进行下载。下载最新版本并按照说明进行操作，就可以顺利完成 Kotlin 的安装。

注意：

这里仅简略说明这些命令。因为如果你已经在使用命令行，那么不需要很多手把手的教程。而对于大多数人来说，使用 IDE 确实是最佳方法。另外，你还可以使用 IntelliJ 作为编译器的代理，所以与其在命令行上花费时间，不如将时间投入到 Kotlin 开发中去！

2. 在 Mac OS X 上安装 Kotlin 命令行

让 Kotlin 在 Mac OS X 上工作的最简单方式是使用 Mac 上流行的包管理器：Homebrew (brew.sh)或者 MacPorts (www.macports.org)。这两个包管理器都能让 Kotlin 的开发和运行变得简单。

对于 MacPorts，仅运行以下命令：

```
brett $ sudo port install kotlin
```

这需要提升权限，但运行完以上命令，一切都会准备妥当。

对于 Homebrew，首先执行更新：

```
brett $ brew update
```

接下来，安装 Kotlin：

```
brett $ brew install kotlin
```

3. 在 UNIX 系统上安装 Kotlin 命令行

如果你使用的不是 Mac OS X，仍是 UNIX 类的操作系统，则可以使用 SDKMAN! (sdkman.io)来安装 Kotlin。

注意：

严格来说，Mac OS X 也是基于 UNIX 的操作系统，因此可以在 Mac 上用 SDKMAN! 来替代 Homebrew 或者 MacPorts。

首先，获取 SDKMAN!：

```
brett $ curl -s https://get.sdkman.io | bash
```

完成后，需要打开一个新的终端或 Shell 窗口，或者用 source 命令执行已更新的文件(在安装过程结束时指定)。

现在，安装 Kotlin：

```
brett $ sdk install kotlin
```

4. 验证命令行的安装

既然已选择安装 Kotlin，完成后应使用如下命令验证安装过程：

```
brett $ kotlinc
```

警告：

此时，可能会提示安装 Java 运行时。这应该让你的系统来处理，接受提示即可，而不必担心。找到对应系统的 JDK 或 JRE，下载并运行。然后再回来试着运行一下 kotlinc。

如果你的系统已经完成了 Java 的适当配置，则应返回如下内容：

```
brett $ kotlinc

Java HotSpot(TM) 64-Bit Server VM warning: Options -Xverify:none and -noverify
were deprecated in JDK 13 and will likely be removed in a future release.

Welcome to Kotlin version 1.3.61 (JRE 13.0.2+8)

Type :help for help, :quit for quit

>>>
```

这是 Kotlin REPL (Read-Eval-Print Loop)，一个快速评估 Kotlin 语句的工具。稍后将详细介绍它，现在输入:quit 命令退出 REPL。

还可以使用以下命令验证你的 Kotlin 版本：

```
brett $ kotlin -version

Kotlin version 1.3.61-release-180 (JRE 13.0.2+8)
```

现在，我们已准备出发！

1.5 创建有用的对象

有了可以工作的 Kotlin 环境，是时候充实代码清单 1.2 中的 Person 类了。前面提到过，对象应是现实世界中的事物。具体来说，如果一个对象代表世界中的某个“事物”(或者一种正式的说法是事物通常为“名词”)，那么它就应该具有那个事物的属性或特征。

一个人最基本的属性是他的姓名，具体来说是名字和姓氏。这些可以表现为对象的属性。而且，这些是必需的属性，你不会想要创建一个没有名字或姓氏的人。

对于必需的属性，最好在创建对象的新实例时要求提供这些属性。实例只是对象的特定版本，所以你可以有多个 Person 类的实例，每个实例代表不同的人。

警告：

你可能已经发现，我们提到了很多与类相关的专业词汇。遗憾的是，后面还要介绍一些这样的词汇：实例(instance)、实例化(instantiation)和构造函数(constructor)等。不必急于理解这些词汇，继续阅读，你很快就能适应这些词汇。在第 3 章中，还将深入地理解这些概念，事实上，你会在整本书中不断地重新审视和扩展与类相关的知识。

1.5.1　使用构造函数将值传递给对象

Kotlin 中的一个对象可以包含一个或多个构造函数。顾名思义，构造函数能构造对象。更具体而言，它是一种特殊的方法，在创建对象时运行。它也可以接收属性值，如必需的名字和姓氏。

可以将构造函数放在类定义的后面，如下所示：

```
class Person constructor(firstName: String, lastName: String) {
```

在本示例中，构建函数具有两个属性：firstName 和 lastName，它们都为 String 类型。代码清单 1.3 在上下文中列出了完整的程序，并通过传递 firstName 和 lastName 的值来创建 Person 的实例。

代码清单 1.3　用 Kotlin 编写的一个对象及其构造函数

```
class Person constructor (firstName: String, lastName: String) {

    /* This class still doesn ' t do much! */

}

fun main() {
    val brian = Person( "Brian" , "Truesby" )
}
```

警告：

你有时会听到属性或属性值被称为参数。这并没有错，参数通常是指被传递给另一个事物的事物；在本示例中，一个事物(名字或姓氏)被传递给了另一个事物(构造函数)。但是一旦它们被分配到对象实例中，就不再是参数了。此时，它们就是对象的属性(或更准确地说，是属性值)。所以，从一开始就称之为属性值会更容易理解一些。

看到没？术语会令人困惑。不过，还是那句话，时间能解决一切。让我们继续。

现在，该类需要一些有用的属性。但正如大多数开发者所知，程序员往往有一种缩减代码的倾向。人们普遍倾向于少打字，而不是多打字(注意，本书作者无此倾向)。因此，代码清单 1.3 也可以被缩减：你可以丢弃“constructor”一词，并且毫无影响。代码清单 1.4 展示了仅有细微差别的缩减版本(以及有争议的改进)。

代码清单 1.4　去除关键字 constructor

```
class Person(firstName: String, lastName: String) {

   /* This class still doesn' t do much! */

}

fun main() {
    val brian = Person( "Brian" , "Truesby" )
}
```

1.5.2　使用 toString ()方法打印对象

这个 Person 类确实比原来的好一些。但它依然没有输出，这个类基本上仍毫无用处。不过 Kotlin 无偿提供了一些方法可用于输出，即每个类都会自动获得一个 toString()方法。你可以先创建该类的实例，然后调用该方法，具体如下：

```
val brian = Person("Brian" , "Truesby" )

println (brian.toString())
```

对 main()函数进行此更改。创建一个新的 Person 类(向它传递任何你想要的姓名)，然后用 println 打印对象实例，向 println 传入 toString()的结果即可。

注意：

你可能想要知道，toString()方法究竟从何而来。它的出现很神奇，但也不那么神奇，它实际上来自继承。继承是与对象密切相关的概念，我们将在第 3 章和第 5 章中更详细地介绍。

1. 术语：函数和方法

下面更细致地介绍一些术语。函数一般为一段代码。例如，main()就是一个函数。方法是附着在对象上的函数。换句话说，方法也是一段代码，但它并不像函数那样没

有“牵绊”而独立存在。方法定义在对象上，通过特定的对象实例运行。

注意：

在许多 Kotlin 官方文档中，函数与方法之间没有明确的区别。然而，我之所以选择这样区分，是因为通常这在面向对象的编程中很重要，如果你使用过或熟悉任何其他基于对象的语言，就会遇到这些术语。但你应该意识到，在“正确的”Kotlin 中，所有的方法都是函数。

注意上面一段中的最后一句很重要，所以可以再读一遍。重要是因为它意味着方法可以与对象实例交互。例如，方法可以使用对象实例的某个属性值(如名字和姓氏)。记住这一点，继续回到代码！

2. 打印对象(通过简略表达式)

可以随时运行 println 函数，仅传入需要打印的内容即可。因此可以使用如下代码：

```
println("How are you?")
```

然后就能在结果窗口中看到输出。也可以打印方法(如 toString())的返回结果，就像之前那样。但还有一条捷径。如果你传给 println()函数某个具有 toString()方法的实例，那么该方法会自动运行。因此，你可以将以下代码：

```
println(brian.toString())
```

缩减成如下代码：

```
println(brian)
```

在后一种情况下，Kotlin 在看到传给 println()函数一个对象后，就会自动运行 brian.toString()并打印结果。无论是哪种情况，都会得到类似下面的输出：

```
Person@7c30a502
```

这个很有用，不是吗？它本质上是一个标识符，是 Person 的特定实例的标识符，对 Kotlin 内部和 JVM 来说有用，但除此之外别无他用。下面修复这个问题。

1.5.3　覆盖 toString ()方法

关于类方法最炫酷的一点是，可以自定义该方法的功能。目前，我们还没有这样做，接下来试一下。但这里的情况略有不同：我们并没有给这个方法编写过代码(当然，它也没有做我们想做的事)。

在这种情况下，你可以对方法进行覆盖。这意味着用你自己的代码替换方法的代码。这正是我们要做的。

首先，需要使用关键字 override 告诉 Kotlin，你正在覆盖该方法。然后，使用另一个关键字 fun，之后是要覆盖的方法的名称，如下所示：

```
override fun toString()
```

注意：

之前你了解了函数和方法间的区别。在本例中，toString()绝对是 Person 类的一个方法。那么，为什么要用 fun 关键字呢？它看起来很像“function”，事实上的确如此。

官方的回答是，Kotlin 原则上将方法视为附于对象上的函数，并且对函数和方法使用相同的关键字会更容易一些。

如果你被这个问题所困扰，别担心，我也一样被困扰！但要明白一点，就 Kotlin 而言，均要使用 fun 来定义函数和方法。

但是，toString()方法有一个返回值。它返回一个 String 类型的值，用于打印。你需要告诉 Kotlin 这个方法会返回什么。需要在括号之后使用一个冒号，之后是返回类型，这里返回 String 类型，如下所示：

```
override fun toString(): String
```

现在你可以在花括号之间编写该方法的代码，如下所示：

```
class Person(firstName: String, lastName: String) {

    override fun toString(): String {
        return "$firstName $lastName"
    }
}
```

这看起来不错，你可能已经发现，在变量名之前放置一个美元符号($)可以让你访问该变量。因此，你将 firstName 和 lastName 变量传入 Person 类的构造函数并打印出来，可行否？

不完全可行。如果运行这段代码，实际上会得到图 1.10 所示的错误。

这是怎么回事？问题有点棘手。

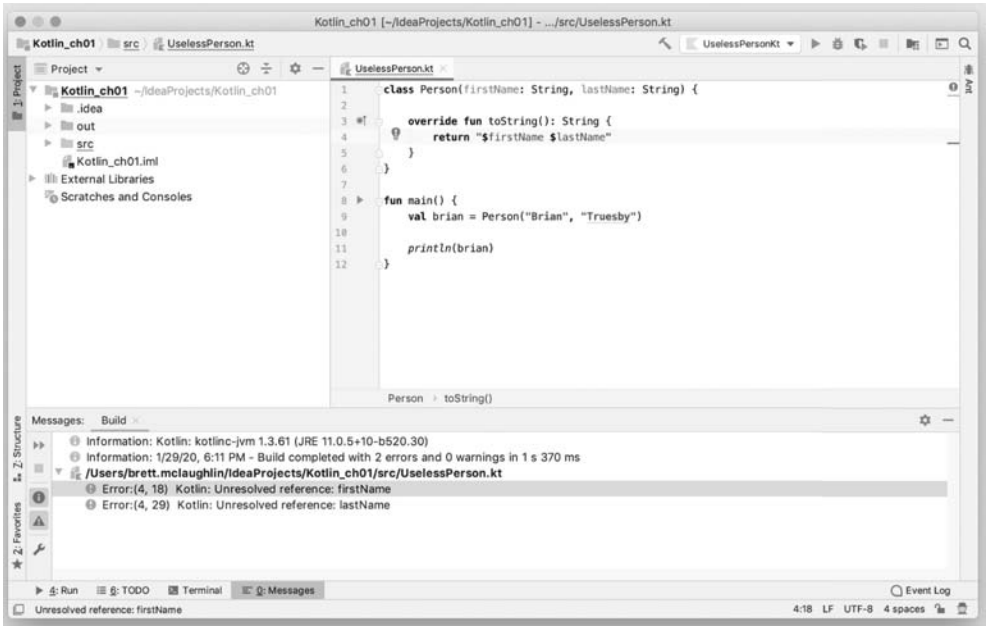

图 1.10　覆盖后的 toString()不起作用

1.5.4　数据并不都是属性值

比如你有一个构造函数，它接收 firstName 和 lastName 两个数据作为参数。这是由构造函数的声明语句决定的：

```
class Person(firstName: String, lastName: String) {
```

但问题是：仅仅接收这些值并不能真正把它们变成属性值。这就是产生图 1.10 所示错误的原因：Person 对象接收了名字和姓氏，但随后立即忽略了它们。它们在所覆盖的 toString()方法中无法使用。

你需要在每个数据前使用 val 关键字将该数据转换为属性值。以下是需要做出的更改：

```
class Person(val firstName: String, val lastName: String) {
```

具体点就是，通过使用 val 关键字(或者 var，稍后介绍)，你创建了变量，并将它们赋值给所创建的 Person 实例。然后允许对这些变量(也许称之为属性更准确)进行访问，例如，在你的 toString()方法中使用它们。

完成这些更改(参见代码清单 1.5)后，编译并运行程序。

代码清单 1.5　将数据转换为实际属性

```
class Person(val firstName: String, val lastName: String) {

    override fun toString(): String {
        return "$firstName $lastName"
    }
}

fun main() {
    val brian = Person( "Brian" , "Truesby" )

    println(brian)
}
```

应得到下面这行输出：

```
Brian Truesby
```

显然，如果你在名字和姓氏上传递不同的值，名字会有所不同，但结果是一样的，这很重要。现在你已经：

- 创建了一个新对象。
- 为对象定义了构造函数。
- 在该构造函数中接收两个数据，并将它们存储为与对象实例关联的属性
- 覆盖了一个方法，使其有用。
- 编写了一个 main()函数。
- 实例化了你的自定义对象并传入了值。
- 使用所覆盖的方法将对象打印出来。

一切都还顺利！在结束 Kotlin 的首次尝试之前，还有一个细节问题需要解决。

1.6　初始化对象并更改变量

假设你想继续研究 Person 类。可尝试更新你的代码，如代码清单 1.6 所示(你对部分代码有困惑不要紧，先大致明白即可)。

代码清单 1.6　为 Person 类创建新属性

```
class Person(val firstName: String, val lastName: String) {
    val fullName: String

    // Set the full name when creating an instance
    init {
```

```
        fullName = "$firstName $lastName"
    }

    override fun toString(): String {
        return fullName
    }
}

fun main() {
    // Create a new person
    val brian = Person("Brian", "Truesby")

    // Create another person
    val rose = Person("Rose", "Bushnell")

    println(brian)
}
```

在此你可以看到一些新变化，但也没什么稀奇。首先，在 Person 对象内声明一个新的变量 fullName。在 main()函数中已经进行过这样的操作。不过这一次，由于是在 Person 对象内部声明变量，因此该变量会自动成为每个 Person 实例的一部分。

另一个小变化是在 main()函数中新增了一个 Person 实例：这一次是一个名为 rose 的变量。

然后，使用了一个新的关键字 init。后面将进一步介绍这个关键字。

1.6.1　使用代码块初始化类

在大多数编程语言(包括 Java)中，构造函数接收传值(就像在 Person 类中一样)，并做一些基本的逻辑处理。Kotlin 有所不同：它引入了初始化代码块(initializer block)。这个初始化代码块通过关键字 init 可以很方便地识别——你可以把每次创建对象时都应运行的代码放在其中。

这可能与你以往的操作有所不同：数据是通过构造函数输入的，但它与初始化代码是分隔开的，它位于初始化代码块中。

在下面的示例中，初始化代码块使用了新的 fullName 变量，并使用通过类构造函数传递的名字和姓氏属性来设置它的值：

```
// Set the full name when creating an instance
init {
    fullName = "$firstName $lastName"
}
```

然后，这个新变量被用于 toString()方法：

```
        override fun toString(): String {
    return fullName
}
```

警告：

正如本章所述，长远来看，这也许是你学到的最重要的东西。通过更改 toString()方法来使用 fullName 变量，而不是直接使用 firstName 和 lastName 变量，你正在实施一个名为 DRY(Don't Repeat Yourself)的原则：避免重复代码。为此，不用再重复组合名字和姓氏，因为已经在初始化代码块中完成了。你只需要将组合结果赋值给一个变量，从此以后，应该使用该变量而不是该变量实际引用的内容。稍后将对此做更多介绍，先记住一点：这很重要！

1.6.2 Kotlin 自动生成 getter 和 setter

到目前为止，事情进展很顺利。部分原因在于你添加的代码，另外很大一部分得益于 Kotlin 在幕后做了很多工作。它会自动运行代码(如初始化代码块)，并允许你覆盖方法。

Kotlin 还做了其他一些事：它在类中自动生成了一些额外的方法。因为你创建了 firstName 和 lastName 属性值(使用 val 关键字)，还定义了一个 fullName 属性，所以 Kotlin 创建了所有这些属性的 getter 和 setter 方法。

术语：getter、setter、更改器、访问器

getter 方法可用来获取一个值。例如，可以将以下代码添加到你的 main()函数中，它不仅会起作用，还可以打印出 Person 实例 brian 的名字：

```
// Create a new person
val brian = Person( "Brian", "Truesby" )
println (brian.firstName )
```

它能起作用是因为 Person 的 firstName 属性具有 getter 方法。当然，也可以让 fullName 和 lastName 属性拥有自己的 getter 方法。getter 的更正式说法是访问器(accessor)，它提供了对值的访问权限，在此它提供了 Person 类的一个属性。它是“免费的”，因为 Kotlin 会自动为你创建这个访问器。

Kotlin 还提供了一个 setter 方法，或者(更正式的称呼是)一个更改器(mutator)。更改器允许你更改值。因此，可以将如下代码添加到 main()函数中：

```
// Create a new person
val brian = Person( "Brian", "Truesby" )
```

```
println (brian.firstName )

// Create another person
val rose = Person( "Rose", "Bushnell" )
rose.lastName = "Bushnell-Truesby"
```

正如可以通过访问器获取数据一样，也可以通过更改器更新数据。

警告：

大多数情况下，我都称 getter 为“访问器”，称 setter 为“更改器”。这样的称呼不像 getter 或 setter 那么普遍。但正如我的一位好朋友和编辑曾经告诉我的那样：“setter 让我想到一只毛茸茸的可爱小狗，而更改器让我想到用它来更改类数据”。一转眼，这一习惯已沿用了 20 年。

现在，如果你继续并编译这段代码，就会遇到一个奇怪的错误，这将是深入探索对象之前要解决的最后一个问题。

1.6.3　常量变量不能改变

以下是导致问题的代码：

```
// Create another person
val rose = Person( "Rose", "Bushnell" )
rose.lastName = "Bushnell-Truesby"
```

如果尝试运行这段代码，将遇到如下错误：

```
Error: Kotlin: Val cannot be reassigned
```

Kotlin 的特性之一是它对变量的强硬立场。具体来说，Kotlin 不仅允许你声明变量的类型，还允许声明该变量是可变变量还是常量变量。

注意：

这里的术语有些令人困惑，所以请在此多花点时间。就像用关键字 fun 声明方法一样，你需要一些时间才能习惯常量变量。

在 Kotlin 中声明变量时，可以使用关键字 val，就像之前所做的那样：

```
val brian = Person("Brian", "Truesby")
```

但也可以使用关键字 var，如下所示：

```
var brian = Person("Brian", "Truesby")
```

首先，在这两种情况下，你最终都会完成一个变量的声明：例如，val 并不代表 value，

只是除了 var 的另一种声明变量的方式。当使用 val 时，表示正在创建一个常量变量。在 Kotlin 中，一个常量变量可以被赋值一次，并且只能赋值一次。然后，该变量是恒定的，永远无法更改。

可使用下面这行代码在 Person 类中创建 lastName 变量：

```
class Person(val firstName: String, val lastName: String) {
```

这将 lastName(和 firstName)定义为常量变量。创建 Person 实例时，它就会被传入并赋值，无法再对其值进行更改。因此以下语句是非法的：

```
rose.lastName = "Bushnell-Truesby"
```

为了清除之前的奇怪错误，需要让 lastName 成为一个可变变量，需要使它在初始赋值后是可更改的。

注意：

setter 会让人误以为是狗的名字，这或许是改称更改器的一个理由；更改器允许你更改一个可变变量。与使用“setter”相比，使用术语“更改器”显得更加合理。

因此，更改 Person 构造函数，使用 var 而不是 val。这表明 firstName 和 lastName 可以被更改：

```
class Person(var firstName: String, var lastName: String) {
```

现在，你应该能再次编译程序，不再会有任何错误。事实上，一旦完成了程序的编译，就可以做一些其他的调整。最终程序看起来如代码清单 1.7 所示。

代码清单 1.7　使用可变变量

```
class Person(var firstName: String, var lastName: String) {
    var fullName: String

    // Set the full name when creating an instance
    init {
        fullName = "$firstName $lastName"
    }

    override fun toString(): String {
        return fullName
    }
}

fun main() {
    // Create a new person
    val brian = Person( "Brian", "Truesby" )
```

```
    println(brian)

    // Create another person
    val rose = Person( "Rose", "Bushnell" )
    println(rose)

    // Change Rose' s last name
    rose.lastName = "Bushnell-Truesby"
    println(rose)
}
```

在这里，fullName 已经成为了可变变量，而且 main()函数中还有几条打印语句。现在程序应该可以毫无错误地编译和运行了。

但是等一下！你看到输出了吗？有问题！以下是运行代码时可能获得的结果：

```
Brian Truesby
Rose Bushnell
Rose Bushnell
```

尽管以上三行中有两行是正确的，但还不够完美。为什么 Rose 的名字在最后一个实例中没有打印出她的新姓氏？

要解决这个问题，请阅读第 2 章，该章更深入地介绍了 Kotlin 如何处理数据、类型，以及更多自动运行的代码。