Broadview®
www.broadview.com.cn 博文视点

综合进阶向导

Scala编程

Programming in Scala

黄海旭　高宇翔　译

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

Scala 编程

Programming in Scala

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 介 绍

本书介绍了一种新的编程语言，它把面向对象和函数式编程概念有机地结合为整体，从而形成一种完整统一、语义丰富的新思维体系。本书循序渐进，由浅入深，经作者精心组织、仔细编排，将语言中的各种概念自然地铺陈在字里行间。除此之外，本书还包含了大量富有针对性和趣味性的示例，它们除了提供对语言各个方面的具体演示之外，还从侧面说明了如何将函数式编程的理念切实并广泛地应用到面向对象编程中。本书面向的读者是有一定编程经验的开发人员，他们希望能够开拓眼界，并致力于提高在软件开发各方面的技能。

版权贸易合同登记号　图字：01-2010-4536

图书在版编目（CIP）数据

Scala编程 / 奥德斯基（Odersky, M.），莱斯彭（Spoon, L.），凡纳斯（Venners, B.）著；
黄海旭，高宇翔译．—北京：电子工业出版社，2010.11
书名原文：Programming in Scala
ISBN 978-7-121-12119-7
Ⅰ．①S… Ⅱ．①奥… ②莱… ③凡… ④黄… ⑤高…
Ⅲ．①JAVA语言－程序设计 Ⅳ．①TP312
中国版本图书馆CIP数据核字（2010）第212228号

策划编辑：卢鹓翔
责任编辑：杨绣国
印　　刷：北京天宇星印刷厂
装　　订：三河市皇庄路通装订厂
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：787×1092　1/16　**印张**：33　**字数**：782千字
印　　次：2010年12月第1次印刷
定　　价：89.00元

译者序

最初接触 Scala 是在 2008 年的 9 月。当时刚刚换了公司，工作上出现了一段“空档期”，因此非常想找一些有技术含量的事情来做。而编程语言的设计开发，或者更确切地说只是对于“银弹”的幻想，这始终是我的爱好之一。那时总会感觉自己应该为这么多年的开发工作总结一些什么，让它们变成之后工作的基础，或者即使没有那么基础也可以是一种参考，即便连参考都不算，至少也是一种思考和总结吧！

然后我重新审视了自己曾做过的项目，分析了项目中使用过的编程语言（Delphi / C++ / C# / Java），总结其中的优缺点，想象着如果是我做的话会有什么样的改善，甚至开始着手策划语言的语法结构，希望能够自圆其说，从只言片语中建立起完整的逻辑体系。在这样的过程中，我时常会发现自己钻进了死胡同，似乎不断地修改只是在不断地重复，甚至唯一的改进也只是“语法糖”而已。没有发现与其他语言间的标志性差异，不能建立独有的思维方式。

于是我在互联网上寻找有没有什么现成的、别出一格的新观点或想法，能够启发我步出自己的圈子，带给我一种“啊哈，原来是这样！”的感觉。不过真没想到，很快，我就找到了。而且是这么彻底，甚至让我打消了“从轮子造起”的念头。

不得不说，Scala 语言的确具有很新颖的创意。这在之前所用的语言中是不多见的。

相信大多数商业软件的程序员都是从 Fortran、Pascal、C/C++/Java 或 COBOL 语言一路学起的。本书中，这一大类的语言被称为是指令式语言，意指以变量和过程语句构造指令的方式对需要实现的目的进行编码的语言。与之相对的是被称为函数式语言的另一个大类，这种语言把程序的目的转换为函数和函数应用，是以函数计算这种不同的视角来处理一般问题的。

直到现在，Lisp、Scheme 或者 Haskell 这样的函数式语言也仅仅还是“小众”语言。这在 TIOBE 语言排行榜上也可见一斑。上榜的前 20 位几乎被面向对象的或者面向过程的指令式语言占满。尤其是前三甲则几乎始终被 Java、C、和 C++所把持。反观函数式语言，Lisp 在第 13 位，是最靠前的了，Scheme 在第 25 位，Haskell 在第 40 位左右。但并不是说函数式语言对于指令式语言就不具有可比性，相反，在很多地方函数式语言都体现了它们独到的优势，如：代码的简洁性、值的不可改变、函数的无副作用、值和函数替换的指称透明等。

但函数式语言与指令式语言总显得有点儿“格格不入”。一个被指令式语言锻炼出来的头脑很难理解函数式语言的精髓，已经用惯了循环语句的方式解决问题，想要把它转换为递归语句也不是很容易的事情。莫非，函数式语言始终只能作为学术派的思维游戏，而不能适应于现代商业软件开发的需

要，因此无缘商业软件开发浪潮的洗礼吗？

所以说，Scala 语言的确具有很新颖的创意。因为它能很巧妙地把指令式语言与函数式语言结合在一起。用 Martin Odersky 的话说，就是：指令式语言与函数式语言本就是一枚硬币的两面，它们不应是互相排斥、非此即彼的，而是可以互相协调，共同发挥出更大力量的。

Scala 语言还从许多其他的语言中汲取了长处，构造出了自己完整而优雅的思想体系。从语言的各个方面来看，似乎都存在一些有别于其他语言的特点，但从语言的整体性来看，它的每个特点都不是孤立的，而是作为一个有机的整体自然而然生发出来的功能。你可以发现，Scala 语言中的基本概念并不多，甚至与其他的语言相比，原生数据类型和语法结构还“显得”颇为不足。但 Scala 具有强大的自发展能力，能够构建出“乱真”的自定义语法库，这使得 Scala 语言成为了一种可以依赖于有限的构件创造出几乎无限可能的语言。

（写到这里我眼前总能立刻浮现出 DNA 结构的四个基本构成部分，谁能想到几乎所有复杂生命体的程序都仅仅使用 4 个单词就能够拼装出来呢？）

不过本书还不止是一部编程语言的教材，它还是一部编程方法论的说明。由于本书三位作者同时也是 Scala 语言的设计和开发者，所以想要了解个中秘密真是再也找不到更好的透露者了。通过本书你不但能够学习到这种语言的语义语法结构、应用场景、用法、实例，还能够发现语言作者在这些结构之后的考量、动机、权衡和折中，从中可以领悟到许多编程专家的思维模式，学习到他们处理问题的方法，从而开拓你的眼界，让你不仅在软件代码开发方面，而且还在软件的架构设计思想上面有新的感悟。

我必须承认，在接触 Scala 之前，我只是一个普通的 Java 开发人员，是完全“指令式”的软件从业者。是 Scala 语言开始让我领略到了函数式编程思想的奥妙。因此，借此机会想向语言的创造者表示自己的敬意，是你们非凡的工作给了我另一种全新的思考问题的方式，同时也“断送”了我“制造轮子”的梦想（笑）。

本书的翻译工作分为两部分，其中，前 27 章由本人完成，第 28 至 33 章由高宇翔翻译。

另外，还要感谢电子工业出版社不遗余力地引进国外优秀 IT 类书籍，感谢博文视点的徐定翔编辑在本书筹划阶段给予我参与的机会；感谢卢鹗翔编辑为译者在本书翻译阶段提供的大力支持；感谢刘唯一编辑和杨绣国编辑为本书的统稿和排版不辞辛劳的工作。翻译本书的过程中，译者虽已尽最大努力确保专业术语的统一和准确，也尽最大努力将原作者的意境用朴实的中文展现给读者，但囿于个人的水平，书中的问题和疏漏之处在所难免，敬请读者朋友给予批评指正。

黄海旭

2010 年 10 月于上海浦东

致 Nastaran - M.O.

致 Fay - L.S.

致 Siew - B.V.

序

Martin Odersky 设计的 Pizza 语言曾经震惊了 Java 世界。尽管 Pizza 语言本身并不流行，但它巧妙地把面向对象和函数型语言两种风格融合在了一起，形成了自然而又强有力的组合。Pizza 语言的设计成为了 Java 泛型的基础，Martin 的 GJ（Generic Java）编译器从 Java 1.3 开始成为了 Sun 公司的标准编译器（尽管关闭了泛型）。我有幸维护这个编译器多年，因此对 Martin 设计与实现语言的能力有非常直接的体会。

那时候我们还在 Sun 公司，尝试用一些零打碎敲的、针对特定问题的解决方案来扩展语言，比如 for-each 循环、枚举、自动装包，以简化程序的开发，而 Martin 则继续着他在更强大的正交语言原语方面的工作，帮助程序员用库来提供解决方案。

近年来，静态类型语言受到了冲击。Java 的广泛应用暴露了静态语言编程会导致大量固定写法的弊病。常见的看法是我们应避免静态类型从而消除这种代码，于是人们对动态语言如 Python，Ruby 和 Groovy 的兴趣开始增加。这种看法被 Martin 最近的构想，Scala 的出现打破。

Scala 是一种类型优雅的语言：它是静态类型的，但仅在需要的地方显式定义类型。Scala 从面向对象和函数式语言两方面获得了强大的特性，然后用全新的理念把它们完美地整合成一体。它的语法是如此的轻量级，而原语（primitive）又如此富有表达力，以至于根本可以认为 API 的使用不会产生语法开销。我们可以在标准库中，如拆分器、组合器和执行器中发现例子。而这也说明 Scala 是支持内嵌的特定领域语言。

Scala 会成为下一个伟大的语言吗？只有时间可以证明一切。我相信 Martin Odersky 的小组绝对有这样的能力和水平做到这一点。不过有一件事是确定无疑的：Scala 语言建立了衡量未来语言的新标准。

Neal Gafter
圣约瑟，加利福尼亚
2008 年 9 月 3 日

致谢

许多人持续关注本书及其相关资料，在这里表示感谢。

Scala 语言本身已经成为许多人努力的集合。版本 1.0 的设计和实现得到了 Philippe Altherr、Vincent Cremet、Gilles Dubochet、Burak Emir、Stéphane Micheloud、Nikolay Mihaylov、Michel Schinz、Erik Stenman 和 Matthias Zenger 等人的帮助。Iulian Dragos、Gilles Dubochet、Philipp Haller、Sean McDirmid、Ingo Maier 和 Adriaan Moors 参与了第二版和当前版语言及工具开发的工作。

Gilad Bracha、Craig Chambers、Erik Ernst、Matthias Felleisen、Shriram Krishnamurti、Gary Leavens、Sebastian Maneth、Erik Meijer、David Pollak、Jon Pretty、Klaus Ostermann、Didier Rémy、Vijay Saraswat、Don Syme、Mads Torgersen、PhilipWadler、Jamie Webb 和 JohnWilliams 通过生动和启发性的讨论，或者对此篇文稿早期版本的评注，热情地与我们分享了他们的观点，从而使语言的设计得以成型。Scala 语言电子邮件列表的建设者们同样提供了非常有用的反馈信息来帮助我们改善语言及相关工具。

George Berger 为本书更为流畅的制作过程和 Web 体验做出了巨大的贡献。令人欣慰的是这个项目没有变成一个技术大杂烩。

许多人给我们的早期版本提供了反馈信息。在这里感谢 Eric Armstrong、George Berger、Gilad Bracha、William Cook、Bruce Eckel、Stéphane Micheloud、Todd Millstein、David Pollak、Frank Sommers、Philip Wadler 和 Matthias Zenger。同样感谢硅谷模式组成员他们大有助益的审校：Dave Astels、Tracy Bialik、John Brewer、Andrew Chase、Bradford Cross、Raoul Duke、John P. Eurich、Steven Ganz、Phil Goodwin、Ralph Jocham、Yan-Fa Li、Tao Ma、Jeffery Miller、Suresh Pai、Russ Rufer、Dave W. Smith、Scott Turnquest、Walter Vannini、Darlene Wallach 和 Jonathan Andrew Wolter。我们同样还要感谢 Dewayne Johnson 和 Kim Leedy 在封面设计上的帮助，还有 Frank Sommers 在索引上的工作。

我们要特别感谢所有那些曾经提供给我们建设性意见的读者。你们的建议对我们把本书做得更好提供了非常大的帮助。我们没办法印出所有参与者的名字，但以下是在 eBook 预印刷阶段通过点击建议链接提供了至少五条评论的读者名单，首先以最高评论数排序，然后是字母顺序，感谢这些人：David Biesack、Donn Stephan、Mats Henricson、Rob Dickens、Blair Zajac、Tony Sloane、Nigel Harrison、Javier Diaz Soto、William Heelan、Justin Forder、Gregor Purdy、Colin Perkins、Bjarte S. Karlsen、Ervin Varga、Eric Willigers、Mark Hayes、Martin Elwin、Calum MacLean、Jonathan Wolter、Les Pruszynski、Seth Tisue、Andrei Formiga、Dmitry Grigoriev、George Berger、Howard Lovatt、John P. Eurich、Marius Scurtescu、Jeff Ervin、Jamie Webb、Kurt Zoglmann、Dean Wampler、Nikolaj Lindberg、Peter McLain、Arkadiusz Stryjski、Shanky Surana、Craig Bordelon、Alexandre Patry、Filip Moens、Fred Janon、Jeff Heon、Boris

Lorbeer、Jim Menard、Tim Azzopardi、Thomas Jung、Walter Chang、Jeroen Dijkmeijer、Casey Bowman、Martin Smith、Richard Dallaway、Antony Stubbs、Lars Westergren、Maarten Hazewinkel、Matt Russell、Remigiusz Michalowski、Andrew Tolopko、Curtis Stanford、Joshua Cough、Zemian Deng、Christopher Rodrigues Macias、Juan Miguel Garcia Lopez、Michel Schinz、PeterMoore、Randolph Kahle、Vladimir Kelman、Daniel Gronau、Dirk Detering、Hiroaki Nakamura、Ole Hougaard、Bhaskar Maddala、David Bernard、DerekMahar、George Kollias、Kristian Nordal、Normen Mueller、Rafael Ferreira、Binil Thomas、John Nilsson、Jorge Ortiz、Marcus Schulte、Vadim Gerassimov、Cameron Taggart、 Jon-Anders Teigen、Silvestre Zabala、Will McQueen、还有 Sam Owen。

最后，Bill 还要感谢 Gary Cornell、Greg Doench、Andy Hunt、Mike Leonard、Tyler Ortman、Bill Pollock、Dave Thomas 和 Adam Wright 对本书出版方面提供的观点和建议。

简介

本书是 Scala 编程语言的教程。写给那些直接参与 Scala 开发的人群。我们的目标是通过阅读此书，你能够学会一切所需，成为多产的 Scala 程序员。本书中所有的例子都能在 Scala 版本 2.7.2 下面编译通过。

谁应该阅读此书

本书的主要目标读者是那些想要学习使用 Scala 编程的程序员。如果你打算用 Scala 做下一个软件项目，那么本书是为你准备的。而且，本书还会传授一些新概念以满足那些希望拓展视野的程序员们。比如说，如果你是 Java 程序员，阅读本书将使你领略从函数型编程到高级面向对象思想的许多概念。我们相信学习 Scala 及它隐含的理念，通常能使你成为更好的程序员。

本书假设你已经有了基本的编程知识。当然 Scala 也可以很好地作为首次学习的编程语言，但这不是学习如何编程的书。

从另一方面来说，阅读本书并不需要特定的编程语言知识。尽管大多数人在 Java 平台上使用 Scala 语言，但本书并不设定你具备 Java 基础知识。然而，我们希望读者能够熟悉 Java，这样我们可以在某些时候通过比较 Scala 和 Java 来帮助这些读者明白其中的差别。

如何使用本书

本书的主要目的是作为教材，所以推荐的阅读方式是按照章节的次序，从头到尾完成阅读。我们尽力一次介绍一个话题，并且仅用已经介绍过的话题来说明新的话题。因此，如果你跳到后面想先睹为快，就可能会发现有些东西用了不太明白的概念来解释。我们认为按照章节的顺序阅读，这种一步一个脚印的方式将引导你顺利地获得 Scala 开发的技能。

如果你发现一个不懂的术语，请一定查找一下术语表和索引。许多读者会略过书中的某些部分，这不是问题，术语表和索引有助于你返回到你略过的某些东西。

读完一遍之后，本书还可以作为语言参考书。Scala 语言有正式的定义，但是语言的定义是以可读性变差为代价换取精确性好的文档。尽管本书并未涵盖 Scala 的所有细节，但在你更好地掌控 Scala 编程之前，它已经足够作为一本通信易懂的语言参考书了。

如何学习 Scala

简单地通读本书，你将学到 Scala 的许多东西。但如果再稍微多付出一点努力，就能更快更全面地

了解 Scala。

首先，你可以充分利用本书中包含的许多编程例子。尝试自己输入能强迫你的大脑思考每一行代码。而尝试各种各样的变化能让它们变得更有趣也能让你确信已真正明白它们如何工作。

其次，与多个在线论坛保持联系。采用这种方式，你和其他 Scala 爱好者就能够互相帮助。你还可以访问更多的电子邮件列表、讨论论坛、聊天室、维基百科和多个特别为 Scala 准备的文档资料更新点。花些时间来查找包含你需要的信息的地方。这样，花更少的时间在小问题上，就能花更多的时间在更深入和更重要的地方。

最后，一旦你已经掌握了部分内容，请把它用在你自己的编程项目上。从草案开始开发一个小程序，或大一点儿程序的附加部分。仅仅看书不能学到更多东西。

印刷体变化

首次使用的术语（term），将使用括号演示原来的英文词语。小代码例子，如 `x+1`，将用等宽字体演示在文档段落中。大段的代码例子将放在等宽字体的段落中演示：

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

在演示交互式 shell 的时候，shell 的回应将演示为深红色字体。

```
scala> 3 + 4
res0: Int = 7
```

内容概要

- 第 1 章，“可伸展的语言”，给出了 Scala 的设计，和它后面的理由，历史的概要。
- 第 2 章，“Scala 的入门初探”，展示给你如何使用 Scala 完成若干种基本编程任务，而不牵涉过多关于如何工作的细节。本章的目的是让你的手指开始敲击并执行 Scala 代码。
- 第 3 章，“Scala 的入门再探”，演示更多的基本编程任务来帮助你更快速地上手 Scala。本章之后，你将能够开始在简单的脚本任务中使用 Scala。
- 第 4 章，“类和对象”，通过描述面向对象语言的基本建设模块和如何编译及运行 Scala 程序的教程开始有深度地覆盖 Scala 语言。
- 第 5 章，“基本类型和操作”，覆盖了 Scala 的基本类型，它们的字面量，可执行的操作，优先级和关联性是如何工作的，还有什么是富包装器。
- 第 6 章，“函数式对象”，进入了 Scala 面向对象特征的更深层次，使用函数式（即不可变）有理数作为例子。

- 第 7 章，“内建控制结构”，演示了如何使用 Scala 的内建控制结构，如 `if`，`while`，`for`，`try` 和 `match`。
- 第 8 章，“函数和闭包”，深度讨论了函数式语言的基础建设模块，函数。
- 第 9 章，“控制抽象”，演示了如何通过定义你自己的控制抽象来增强 Scala 的基本控制结构。
- 第 10 章，“组合与继承”，讨论了更多 Scala 对面向对象编程的支持。这个话题并不像在第 4 章中那样基础，但它们在实践中经常出现。
- 第 11 章，“Scala 的层级”，解释了 Scala 的继承层级并讨论了其全体方法及底层类型。
- 第 12 章，“特质”（trait），演示了 Scala 在混入组成（mixin composition）中的机制。本章演示了特质如何工作，描述了通常的用法，还解释了为什么特质改善了传统的多继承。
- 第 13 章，“包和引用”，讨论了大项目编程中的事务，包括顶层包，引用语句，还有访问控制修饰符如，`protected` 和 `private`。
- 第 14 章，“断言和单元测试”，演示了 Scala 的断言机制并大致学习了各种可以为 Scala 编写测试的工具。
- 第 15 章，“样本类和模式匹配”，介绍了样本类和模式匹配，这对你在编写正规的非封装的数据结构时用到的工具，尤其对树型递归数据很有用。
- 第 16 章，“使用列表”，详细解释了列表。它或许是在 Scala 程序中最常用到的数据结构。
- 第 17 章，“集合类型”，演示了如何使用基础的 Scala 集合类型，如：列表、数组、元组集（tuple）及映射表。
- 第 18 章，“有状态的对象”，解释了什么是有状态（即可变）的对象，Scala 提供的语法层面表达它们的术语。本章包括了一个在离散事件模拟上的案例研究，用来演示一些有状态对象的动作。
- 第 19 章，“类型参数化”，用具体的例子：纯函数队列类的设计，解释了第 13 章介绍过的一些信息隐藏技术。本章建立了关于各种类型参数的描述，以及它如何与信息隐藏实现交互。
- 第 20 章，“抽象成员”，描述了所有 Scala 支持的抽象成员。能够声明为抽象的不仅是方法，还包括字段和类型。
- 第 21 章，“隐式转换和参数”，描述了这两个特性有助于程序员忽略掉源码中那些能由编译器推导出来的繁琐的细节的特性。
- 第 22 章，“实现列表”，描述了 `List` 类的实现。弄明白在 Scala 里面列表如何工作是很重要的，而且，实现本身展示了若干 Scala 特性的应用。
- 第 23 章，“重访 For 表达式”，解释了 `for` 表达式是如何翻译成对 `map`、`flatMap`、`filter`

和 foreach 的访问。

- 第 24 章（Extractors），“抽取器”，展示了如何使用模式匹配任何类，而不仅仅是用例类。
- 第 25 章，“注解”，演示了如何通过注解使用语言的扩展部分。本章示范了若干标准注解，也示范了如何建立你自己的注解。
- 第 26 章，“使用 XML”，演示了在 Scala 中如何处理 XML。包括 XML 的创建、解析，以及解析之后的处理等一系列惯用方式。
- 第 27 章，“使用对象的模块化编程”，演示了说 Scala 的对象已足够丰富的原因，从而消除了分离式模块系统的使用需求。
- 第 28 章，“对象相等性”，指出若干在编写 equals 方法时要考虑的情况。说明了若干应避免的误区。
- 第 29 章，“结合 Scala 和 Java”，描述了若干在同一个项目中捆绑使用 Java 和 Scala 时会碰到的状况，以及建议的解决方法。
- 第 30 章，“Actor 和并发”，展示如何使用 Scala 的 actor 并发库。尽管你使用 Java 平台的同步原语和来自于 Scala 程序的库，但 actor 能帮你避免死锁和资源竞争这些影响着传统并发的问题。
- 第 31 章，“连结符解析”，演示了如何使用 Scala 的解析器连结符库来创建解析器。
- 第 32 章，“GUI 编程”，展示了使用 Scala 库简化基于 Swing 的 GUI 编程的快速旅程。
- 第 33 章，“SCells 试算表”，通过展示一个完整的试算表的实现，集中演示了 Scala 的一切。

资源

在 Scala 的主网站，*http://www.scala-lang.org*，你能找到 Scala 最近的发布版和文档、社区资源的链接。Scala 资源链接更全的页面，请访问本书网站：*http://booksites.artima.com/programming_in_scala*。与本书其他读者交流，请访问：*http://www.artima.com/forums/forum.jsp?forum=282*。

源码

你可以从本书的网站下载包含本书源码的 ZIP 文件，它是以 Apache 2.0 开源许可方式发布的：*http://booksites.artima.com/programming_in_scala*。

勘误

尽管本书已复审检查多次，仍不可避免错误的发生。要查阅本书的勘误列表，请访问：*http://booksites.artima.com/programming_in_scala/errata*。如果你发现错误，也请在上述网址报告，这样我们可以确保在本书将来的印刷或发行版中修正它。

目录概览

目录

图示清单

表格清单

代码清单

Programming in Scala

```
println("Hello, reader!")
```


Scala 编程

第 1 章

可伸展的语言

Scala[1]语言的名称来自于“可伸展的语言”。之所以这样命名，是因为它被设计成可以随着使用者的需求而扩展。Scala 的应用范围很广，从编写简单脚本，到建立大型系统。

Scala 很容易上手。它运行在标准的 Java 平台上，可以与所有的 Java 库无缝交互。它可以用来编写脚本，把 Java 组件组合起来。不过更能够发挥它力量的地方，在于用它来建立大型系统或可重用控件的架构。

从技术层面上来说，Scala 是一种把面向对象和函数式编程理念加入静态类型语言中的混合体。

它在许多方面体现了面向对象和函数式编程的熔合；或许这种熔合比其他那些广泛使用的语言体现得还要深入。在可伸展性方面，这两种编程风格具有互补的力量。Scala 的函数式编程简化了用简单部件搭建实际应用的过程。它的面向对象特性又使它便于构造大型系统并使它们适应新的需求。Scala 中这两种风格的组合使得表达新的编程模式和新的组件抽象成为可能。同样也让这种易读、简洁的编程风格成为顺理成章的事。而且由于 Scala 良好的延展性，因此用它来编程也充满乐趣。

本章作为最初的章节，回答了“为什么使用 Scala”的问题，宏观地描述了 Scala 的设计思想及背后的原因。读完本章之后你将对 Scala 是什么和它将帮助你完成什么样的任务形成基本的认识。尽管本书是一部 Scala 教程，但本章不是教程的一部分。如果你想立刻动手写 Scala 代码，请直接阅读第 2 章。

1.1 与你一同成长的语言

不同规模的程序可能需要不同的编程结构。例如下面的 Scala 程序：

```
var capital = Map("US"->"Washington", "France" ->"Paris")
capital += ("Japan" ->"Tokyo")
println(capital("France"))
```

这段程序建立了国家和首都之间的映射关系表，并填加了一个新的映射关系（"Japan"->"Tokyo"），然后打印了与法国相关的首都[2]。本例中的每行代码都是高级的，直切要点，而且也没有被额外的分号或类型注释将其弄得乱糟糟的。实际上，这种感觉就好像是那种流行的“脚本

[1]注：Scala 的发音是 *skah-lah*。

[2]注：先不要急于了解程序中的所有细节，下两个章节中将对此加以说明。

化”语言，比如、Perl、Python 或 Ruby。上面的这个例子体现了这些语言的一个普遍特征，就是它们都在语法层面上支持“关联映射”。

关联映射非常有用，因为它有助于使程序保持清晰易懂。然而，有些时候你或许不赞成它们的这种“均码”哲学，因为你需要以更细粒度的方式控制程序中映射的属性。Scala 可以在你需要的时候提供这种细粒度的控制，因为它的映射并不是语法特性。它们是库抽象，可以扩展或改造。

上面的程序里使用的是默认的 Map 实现，改变它也很容易。比方说，采用特定的实现，如 HashMap 或 TreeMap，或者混入（mix-in）SynchronizedMap 特质（trait），指定这个映射必须是线程安全的。或者还可以给映射指定默认值，或重载新建映射的任意方法。但不论哪种情况，你都可以使用前面看到的那样简单的映射访问语法。

以上的例子展示了 Scala 的方便性和灵活性。它有全套的便捷组件来帮助你快速启动并让你保持着一种愉悦清晰的状态来编程。与此同时，你有信心不会对语言过度定制。你始终可以裁剪程序以符合所需，因为所有的东西都是基于库模块的，可以依照需要选择和修改。

增加新的类型

Eric Raymond 把大教堂和集市作为软件开发的两个隐喻[3]。大教堂是几近于完美的建筑物，要花很长的时间建设。而一旦建成了就长时间保持不变。相对来说，集市则天天在被工作其中的人调整和扩展。Raymond 的文章中，集市是对于开源软件开发的隐喻。Guy Steele 在他的讲话“发展一门语言”中提到，同样的差别也可以应用在语言定义中[4]。Scala 更像一个集市而不是大教堂，因为它被设计为可以让使用它的人扩展和修改。Scala 并没有提供那种“完美齐全”语言中可能需要的所有东西，而是把制作这些东西的工具放在了你的手中。

这里举个例子。许多程序都需要一种能够任意大且不溢出或由于数学操作而“绕回”的整数类型。Scala 在库类 scala.BigInt 中定义了这样的类型。以下是使用了该类型的方法定义，用以计算传入整数的阶乘值：[5]

```
def factorial(x: BigInt): BigInt =
  if (x == 0) 1 else x * factorial(x - 1)
```

现在，如果调用 factorial(30)，将得到：

```
265252859812191058636308480000000
```

BigInt 看上去就像是内建的类型，因为这种类型值的操作符（如*和–）可以操作整数值。然而它只是碰巧定义在 Scala 标准库中的类[6]。即使没有这个类，也可以直接由任何 Scala 程序员实现出来，比如说可以包装 Java 的类 java.math.BigInteger（实际上，Scala 的 BigInt 就是这么实现的）。

---

[3]注：Raymond，“The Cathedral and the Bazaar”。[Ray99]

[4]注：Steele，“Growing a language”。[Ste99]

[5]注：factorial(x)，或者 x!是一种数学表达，就是计算 1*2*...*x 的值，并且定义 0!的值为 1。

[6]注：Scala 附带的标准库，本书将介绍其中的一些内容。若需要得到更多信息，可以在 Scala 发布包里查找，或者浏览 *http://www.scala-lang.org* 查询库的 scaladoc 文档。

当然也可以直接使用 Java 的类库。但结果却不容乐观，因为尽管 Java 允许创建新的类，但这些类总感觉不像原生的语言支持那么方便。

```
import java.math.BigInteger

def factorial(x:BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
```

BigInt 代表了许多其他数值类型——大十进制数、复数、有理数、置信区间、多项式——诸如此类。有些编程语言原生实现了其中的部分类型。举例来说，Lisp、Haskell 和 Python 实现了大整数；Fortran 和 Python 实现了复数。但是任何语言想要尝试同时实现所有的这些抽象类型的话，都将很容易变得太大而难以管理。更进一步，即使有这样的语言，也总会有些应用会使用其他不被支持的数值类型。所以尝试在一种语言里提供所有东西的解决之道不可能具有良好的延展性。取而代之的是，Scala 允许用户通过定义感觉像原生语言支持一样的易用库在他们需要的方向上发展和改造语言。

增加新的控制结构

前面的例子演示了如何在 Scala 中增加新的类型，使得它们用起来方便得像内建类型一样。同样的扩展理念也应用在了控制结构上。这种扩展可以由 Scala 的“基于 actor”的并发编程 API 阐明。

随着近年来多核处理器的激增，为了获取可接受的性能，应用中必须运用更多的并行机制。这常常就意味着须重写代码以使计算分布到若干并发线程上。不幸的是，创建可依赖的多线程程序经实践证明非常具有挑战性。Java 的线程模型是围绕着共享内存和锁建立的，当系统在规模和复杂度都不断变大的时候，这种模型会越发变得难以理解。很难说程序里面没有资源竞争或潜藏的死锁——有些东西不是能经测试检验出的，或许只在投入生产后才会表现出来。目前可以认为比较安全的可选方案是消息传递架构，例如在 Erlang 编程语言中应用的“actor”方案。

Java 带了一个丰富的、基于线程的并发库。Scala 也可以像使用其他 Java API 那样用它编程。不过，Scala 还提供了一个实质上实现了 Erlang 的 actor 模型的附加库。

actor 是能够实现于线程模型之上的并发抽象。可以通过彼此间传递消息来实现通信。actor 可以执行两种基本操作，消息的发送和接受。发送操作，用惊叹号表示，可以把消息发送给 actor。这里用名为 recipient 的 actor 举例，如下。

```
recipient ! msg
```

发送是异步的；就是说，发送 actor 可以在一瞬间完成，而不需要等待消息被接收和处理。每个 actor 都有信箱（mailbox）把进入的消息排成队列。actor 通过 receive 代码块处理信箱中收到的消息：

```
receive {
  case Msg1 =>...//处理 Msg1
  case Msg2 =>...//处理 Msg2
  // ...
}
```

接收代码块由许多 case 语句组成，每一个都用消息模板查询信箱。信箱中第一个符合某个 case 的消息被选中，并且执行相应的动作。如果信箱中不含有任何能够匹配任意 case 的消息，actor 将休眠等待新进的消息。

这里举一个用 Scala 的 actor 实现校验和（cheksum）计算器服务的简单例子：

```
actor {
  var sum = 0
  loop {
    receive {
      case Data(bytes)        => sum += hash(bytes)
      case GetSum(requester) => requester ! sum
    }
  }
}
```

这个 actor 首先定义了一个名为 sum 的本地变量，并赋初值为零。然后就等待在消息循环中，不断地用 receive 段落接收消息。如果收到了 Data 消息，就把发送的 bytes 取哈希值加到 sum 变量上。如果收到了 GetSum 消息，就用 requester!sum 把当前 sum 值发回给 requester。requester 字段嵌入在 GetSum 消息里，它通常指向创建请求的 actor。

我们目前并不指望你能完全明白 actor 例子。实际上，对于可扩展性这个话题来说这个例子里面最重要的是，无论 actor、loop、receive 还是发送消息的符号(!)，这些都不是 Scala 内建的操作符。尽管 actor、loop 和 receive 看上去或表现上都如此接近于 while 或 for 循环这样的控制结构，但实际上它们只是定义在 Scala 的 actor 库里面的方法。同样，尽管‘!’看上去像是个内建的操作符，它也不过是定义在 actor 库里面的方法。所有这四个构件都是完全独立于 Scala 语言的。

receive 代码块和发送(!)语法让 Scala 看上去更像 Erlang 里的样子，但是在 Erlang 里面，这些构件是内建在语言中的。Scala 同时也实现了 Erlang 里的大多数其他并发编程组件，诸如监控失败 actor 和超时类。总体来说，actor 已变成表达并发和分布式计算的非常好的办法。尽管它们是定义在库里的，但给人的感觉就像 actor 是 Scala 语言整体的一部分。

这个例子表明了 Scala 语言可以往新的方向，例如向并发编程这样的特定方向上发展。当然这样做的前提是，定义良好的架构及素质可靠的程序员。但这里所讲的重点是这样做的确可行——一方面可以在 Scala 里面设计和实现抽象结构，从而快速投入新的应用领域，而另一方面感觉却仍然像是原生的语言支持。

1.2 是什么让 Scala 具有可扩展性?

可扩展性受到从语法细节到控件的抽象构造等许多因素的影响。如果一定要说出 Scala 中有助于可扩展性的一个方面，我们认为是面向对象和函数式编程的组合（呵呵，有作弊嫌疑，这的确是两个方面，但是它们交织在了一起）。

与其他常见语言相比，Scala 在把面向对象和函数式编程熔合为一体的语言设计方面要做得更多。

比方说，其他语言或许分别有对象和方法这样两个不同的概念，而在 Scala 里，函数就是对象。函数类型是能够被子类继承的类。看上去这似乎只是出于学术美感的考虑，但它从深层次上影响了可扩展性。实际上，如果没有这种函数和对象的联合，前面演示的 actor 的设想也将无从实现。本节将浏览 Scala 融合面向对象和函数概念的方法。

Scala 是面向对象的

面向对象编程已经取得了极大的成功。它始于（20 世纪）60 年代中期的 Simula 和 70 年代的 Smalltalk，到现在支持它的语言比不支持的还要更多。甚至某些领域已经完全是面向对象编程的天下了。不过面向对象的含义究竟是什么并没有明确定义，很明显有些事情还是程序员说了算的。

原则上，面向对象编程的动机非常简单：绝大多数的程序都需要某种结构。而最直接的办法就是把数据和操作放进某种形式的容器中。面向对象编程里最伟大的思想是让这些容器完全地通用化，这样就能像保存数据那样保存操作，并且还可以把这些容器作为值存储到其他容器里，或作为参数传递给操作。这样的容器就被称为对象。Alan Kay，Smalltalk 的发明者评论说，这种方法可以让最简单的对象与完整的计算机有同样的架构原则：用形式化的接口绑定数据和操作[7]。于是对象在语言的可扩展性方面起了很大作用：构造小程序和大程序都可以应用同样的技术。

尽管面向对象编程已经在很长一段时间里成为主流，但仍然鲜有语言能在 Smalltalk 之后推动这种构造原则到逻辑结论的转化。举例来说，许多语言允许非对象值的存在，如 Java 里面的原始值。或者它们允许不隶属于任何对象的静态字段和方法。这些对纯理想化面向对象编程的背叛最初看起来完全无害，但令人头疼的事情还在后面，情况会变得复杂并限制了可扩展性。

相反，Scala 是纯粹的面向对象语言：每个值都是对象，每个操作都是方法调用。例如，如果用 Scala 描述 `1 + 2`，实际上等于调用定义在 `Int` 类里的+方法。方法名可以用像操作符一样的名字定义，以便于 API 的使用者像使用操作符那样使用方法。这就是为什么 Scala 的 actor API 的设计者能够在前面的例子里面显示类似 `requester!sum` 这样的表达式：‘!’是 `actor` 的方法。

如果说到组装对象，Scala 比多数别的语言更胜一筹。Scala 的特质（trait）就是其中一例。特质就像 Java 的接口，但可以有方法实现及字段。可以通过混入组装（mixin composition）构造对象，从而带有了类的成员并加入了若干特质的成员。这样的构造方式，可以让不同用途的类包装不同的特质。这看上去有点儿像多重继承，但在细节上是有差异的。与类不同，特质可以把一些新的功能加入还未定义的超类中。这使得特质比类更具有“可加性”。尤为重要的是，它可以避免在多重继承里，通过若干不同渠道继承相同类时发生的经典的“菱形继承”问题。

Scala 是函数式的

除了作为一种纯面向对象的语言，Scala 还是一种成熟的函数式语言。函数式语言的思想比（电子）计算机的诞生还要早。其基础建立在 Alonzo Church 于 1930 年代发展的 λ 算子（lambda calculus）上。第一个函数式编程语言是 50 年代后期的 Lisp。其他流行的函数式语言有 Scheme、SML、Erlang、Haskell、

[7]注：Kay，“The Early History of Smalltalk”。[Kay96]

OCaml 和 F#。很长一段时间，函数式语言处于边缘地带，在学府里流行，但没有广泛应用于业界。然而，近几年对函数式语言和技术的热情开始持续升温。

函数式编程有两种指导理念，第一种理念是函数是头等值。在函数式语言中，函数也是值，与整数或字符串处于同一个地位。函数可以被当作参数传递给其他函数，可以当作结果从函数中返回或保存在变量里。可以在函数里定义其他函数，就好像在函数里定义整数一样。还可以定义匿名函数，并随意地插入到代码的任何地方，就好像使用 42 这样的整数字面量一样。

函数作为头等值这种理念简化了操作符的抽象和新控制结构的创建。这种函数的泛化具有很强的表现力，是程序保持清晰易懂的保证，而且它还在可扩展性上扮演了重要的角色。例如之前在 actor 例子里演示的 `receive` 结构就是把函数当作参数调用的方法，结构里面包含了一段未被执行而直接传入 `receive` 方法的函数。

相反，在多数传统语言中，函数不是值。确实有函数值的语言则又常常把它们贬到二类值的地位。举例来说，C 和 C++的函数指针就不能拥有与非函数指针在语言中同等的地位：函数指针仅能指向全局函数，环境中不能定义可以引用值的头等嵌套函数，也不能定义匿名函数字面量。

函数式编程的第二种理念是程序的操作应该把输入值映射为输出值而不是就地修改数据。其中的差别可以参考 Ruby 和 Java 对字符串的实现。在 Ruby 里，字符串是一个字符数组。字符串中的字符可以被独立的改变，例如可以把字符串对象的分号直接改成句号。而另一方面，在 Java 和 Scala 里，字符串是一种数学意义上的字符序列。使用 `s.replace(';', '.')`这样的表达式替换字符串里的字符会产生一个新的、不同于原字符串 s 的对象。换句话说就是在 Java 里字符串是不可变的（immutable）而在 Ruby 里则是可变的。因此单就字符串来说，Java 是函数式的语言，而 Ruby 不是。不可变数据结构是函数式语言的一块基石。Scala 库在 Java API 之上定义了更多的不可变数据类型。例如，Scala 有不可变的列表、元组、映射表和集。

函数式编程第二种理念的另一种解释是，方法不应有任何副作用（side effect）。方法与其所在环境交流的唯一方式应该是获得参数和返回结果。比如 Java 里 `String` 类的 `replace` 方法就符合这种说法。它的参数包括一个字符串和两个字符，返回的是一个新的字符串，其中的指定字符都替换成了另一个。调用 `replace` 不会有其他的结果。类似于 `replace` 这样的方法被称为指称透明（referentially transparent），就是说对于任何输入来讲，都可以用方法的结果替代对它的调用，而不影响程序的语义。

函数式语言鼓励使用不可变数据结构和指称透明的方法。甚至有些函数式语言必须依赖于它们。Scala 不强迫使用函数式的风格。必要的情况下，可以写成指令形式（imperative），用可变数据或有副作用的方法调用。但是 Scala 有更好的函数式编程方式做替代，因此通常可以轻松地避免使用它们。

1.3 为什么选择 Scala?

Scala 适合你吗？你必须自己搞清楚。除了可扩展性之外，我们发现喜欢用 Scala 编程实际上还有很多理由。本节将讨论其中最重要的四个：兼容性、简洁、高层抽象和高级的静态类型化。

Scala 是兼容的

Scala 不需要你脱离 Java 平台重新学习开发。它可以让你保全现存的代码并添加新东西因为它被设计成可以与 Java 实施无缝的互操作[8]。Scala 程序会被编译成 JVM 的字节码。其运行时的性能通常与 Java 程序的不分上下。Scala 代码可以调用 Java 方法，访问 Java 字段，继承自 Java 类和实现 Java 接口。这些都不需要特别的语法，外部接口描述或胶水代码。实际上，程序员们根本都没有意识到，几乎所有 Scala 代码都大量使用了 Java 库代码。

与 Java 的全交互操作性的另一个方面是 Scala 大量重用了 Java 类型。Scala 的 `Int` 类型代表了 Java 的原始整数类型 `int`，Float 代表了 `float`，`Boolean` 代表 `boolean`，等等。Scala 的数组被映射为 Java 数组。Scala 还重用了许多标准 Java 库类型。例如，Scala 里的字符串字面量 `"abc"` 是 `java.lang.String`，而抛出的异常也必须是 `java.lang.Throwable` 的子类。

Scala 不仅重用了 Java 的类型，而且还把它们“打扮”得更好看。如 Scala 的字符串支持类似于 `toInt` 和 `toFloat` 的方法，可以把字符串转换成整数或浮点数。因此可以用 `str.toInt` 替代 `Integer.parseInt(str)` 方法。不过，它在不打破互操作性的基础上是怎么做到这点的呢？Java 的 `String` 类可没有 `toInt` 方法。实际上，Scala 有一个通用方案可以解决这种高级库设计和互操作性相对立的问题。Scala 允许定义类型失配或者选用不存在的方法时使用的隐式转换（implicit conversion）。在上面的例子里，当在字符串中寻找 `toInt` 方法时，Scala 编译器会发现 `String` 类里没有这种方法，但它会发现把 Java 的 `String` 对象转换为 Scala 的 `RichString` 类实例的隐式转换，而 `RichString` 类中定义了这么个方法。于是在执行 `toInt` 操作之前，转换被隐式应用了。

Scala 代码同样可以由 Java 代码调用。不过由于 Scala 的语义比 Java 更为丰富，因此有时这种情况也更为微妙，有些更先进的 Scala 特性在映射到 Java 前需要先被编码一下。细节将在第 29 章中说明。

Scala 是简洁的

Scala 程序一般都很短。Scala 程序员曾报告说与 Java 比起来，它的代码行数可以减少到 Java 的 1/10。这有可能是极端的例子。较保守的估计大概标准的 Scala 程序的代码行在 Java 写的同样程序的一半左右。更少的行数不仅意味着打更少的字，同样意味着更少的对程序的阅读和理解，以及更少出错的可能。以下的许多因素都有助于减少代码行。

首先，Scala 的语法避免了一些束缚 Java 程序的固定写法。例如，Scala 里的分号是可选的，且通常不写。Scala 语法里还在其他很多地方做了省略。比方说，比较一下在 Java 和 Scala 里分别是如何写类及构造函数的。Java 里，带构造函数的类常是这个样子：

```
// 在 Java 里
class MyClass {
  privateint index;
  private String name;
```

[8]注：Scala 还有.NET 平台的版本，不过目前 JVM 平台的版本支持的最好。

```
  public MyClass(int index, String name) {
      this.index = index;
      this.name = name;
  }
}
```

Scala 里，可以写成这样：

```
class MyClass(index: Int, name: String)
```

根据这段代码，Scala 编译器制造的类将会含有两个私有字段，一个叫 index，Int 类型；另一个叫 name，String 类型，以及用这些变量作为参数获得初始值的构造函数。这个构造函数还将使用传入的参数值初始化这两个字段。一句话，这段代码具有与比这冗长得多的 Java 版本的代码同样的功能[9]。Scala 类写起来更快，读起来更容易，最重要的是，与 Java 类相比更不容易犯错。

另一个有助于 Scala 简洁性的因素是类型推断。重复的类型信息可以被忽略，因此程序变得更有条理和易读。

但减少代码的关键或许是现成的代码库。Scala 包含许多可以定义强有力库的工具以便抽取和提炼通用行为。例如库类的不同功能切面可以被分成若干特质，而这些特质可以被灵活地组合。还有，库方法可以被操作参数化，从而可以实际上自定义控制结构。这些结构组合在一起，就能够让库的定义既是高级的又能灵活使用。

14

Scala 是高级的

程序员总是在与代码的复杂性搏斗。只有对工作的代码了如指掌，才能有的放矢的编程开发。过度复杂的代码成了很多软件项目崩溃的原因。不幸的是，重要的软件通常需求复杂。既然复杂性不可避免，那么就必须加强管理。

Scala 可以通过帮助提升接口的抽象级别来帮助管理复杂性。例如，设想有一个 String 类型的变量 name，想知道它是否含有大写字符。Java 里可以这么写：

```
// 在 Java 里
boolean nameHasUpperCase = false;
for (int i = 0; i < name.length(); ++i) {
  if (Character.isUpperCase(name.charAt(i))) {
      nameHasUpperCase = true;
      break;
  }
}
```

Scala 里，可以写成：

```
val nameHasUpperCase = name.exists(_.isUpperCase)
```

Java 代码把字符串当作可以在循环中逐字符步进的低级实体。而 Scala 代码把同样的字符串当作能用论断（predicate）查询的更高级字符序列。明显 Scala 代码更短并且——对训练有素的眼睛来说——

[9]注：唯一的差别在于 Scala 例子里的实例字段是不变的（final）值。10.6 节里将学习如何把它们改为变量（non-final）。

比 Java 代码更容易懂。因此 Scala 代码在整体复杂度上降低很多，也更少有犯错的机会。

_.isUpperCase 论断是 Scala 函数字面量的例子[10]，它描述了带一个字符参量（用下划线字符代表）的函数，并测试其是否为大写字母[11]。

原则上，这种控制的抽象在 Java 中也是可能的。不过需要为此定义一个包含抽象功能的方法接口。例如，如果想要支持字符串查询，就需要为仅有的方法 hasProperty 定义接口 CharacterProperty：

```
// 在 Java 里
interface CharacterProperty {
  boolean hasProperty(char ch);
}
```

然后在 Java 里用这个接口组建方法 exists：参数分别为字符串和 CharacterProperty，并返回字符串中是否有满足要求的字符的检查结果。最后这样调用 exists 方法：

```
// 在 Java 里
exists(name, new CharacterProperty() {
    public boolean hasProperty(char ch) {
        return Character.isUpperCase(ch);
    }
});
```

然而，这样做过于复杂。实际上多数 Java 程序员都不会惹这个麻烦。他们宁愿写个循环并漠视代码里复杂性的增加。另一方面，Scala 里的函数字面量非常简洁，所以就可以被常常用到。随着对 Scala 的逐步了解，你会发现有越来越多地使用自定义抽象控制结构的机会，而且你将发现这有助于减少代码重叠并因此保持程序的简短和清晰。

Scala 是静态类型的

静态类型系统可以根据保存和计算的值的类型认定变量和表达式类型。Scala 是一种具有非常高级的静态类型系统的语言。它以 Java 的内嵌类型系统为基础，允许使用泛型（generics）参数化类型，用交集（intersection）组合类型及抽象类型（abstract type）隐藏类型细节[12]。这都为自建类型打下了坚实的基础，从而能够设计出即安全又能灵活使用的接口。

如果你喜欢动态语言如 Perl、Python、Ruby 或 Groovy，或许你会发现 Scala 把静态类型系统列为其优点之一有些奇怪。毕竟，很多人都认为动态语言的优点之一就是抛弃了静态类型系统。针对静态类型的最常见的批评就是认为它们使得程序过度冗长，这阻止了程序员用他们希望的方式表达自己，并阻碍了某些可以动态修改软件系统的模式的实现。然而，这些批评经常针对的不是静态类型的思想，而是指责过于冗长或太不灵活的类型系统实现。例如，Alan Kay，Smalltalk 语言的发明者，有一次评论说："我并非批评类型，不过我不知道还有哪个类型系统最终不是惨痛的教训，所以我还是喜欢动态类型。"[13]

[10]注：返回类型为 Boolean 的函数字面量被称作论断。
[11]注：这种使用下划线作为参数占位符的做法会在 8.5 节中描述。
[12]注：第 19 章讨论泛型，第 12 章讨论交集，第 20 章讨论抽象类型。
[13]注：Kay 在关于面向对象编程的含义的电子邮件中如此评价。[Kay03]

我们希望能在书里说服你，Scala 的类型系统是根本谈不上会变成“惨痛的教训”。实际上，它很好地诠释了两个静态类型问题的解决之道：通过类型推断避免了冗余性及通过模式匹配和一些新的编写和组织类型的办法获得了灵活性。把这些绊脚石搬掉后，静态类型系统的经典优越性就体现得更为明显。其中最重要的包括程序抽象的可检验属性、安全的重构，以及更好的文档。

可检验属性。静态类型系统可以保证消除某些运行时的错误。例如可以保证：布尔型不会与整数型相加；私有变量不会从类的外部被访问；用正确数量的参数调用了函数；字符串集只能加入字符串。

不过当前的静态类型系统还不能查到其他类型的错误。比方说，通常查不到无法终结的函数，数组越界，或者除零错误。同样也查不到你的程序不符合规格说明书（假设有这么一份规格说明书）。因此有些人认为静态类型系统不太有用而忽视它。批评说既然这种类型系统只能发现简单错误，而单元测试能提供更广泛的覆盖，那又为何自寻烦恼使用静态类型呢？我们认为这种说法不正确。尽管静态类型系统确实不能替代单元测试，但是却能减少用来测试这些属性的单元测试的数量。同样，单元测试也不能替代静态类型。总而言之，如 Edsger Dijkstra 所说，测试只能证明存在错误，而非不存在[14]。因此，静态类型能给的保证或许很简单，但这些保证无论多少测试都给不了。

安全的重构。静态类型系统让你可以非常有信心地去变动代码基础的安全网。试想一个给方法新增参数的重构实例。在静态类型语言中，你可以完成修改，重编译你的系统并简单地修改所有引起类型错误的代码行。一旦完成了这些，你可以确信已经发现了所有需要修改的地方。这种信心对于其他的简单重构，如改变方法名或把方法从一个类移到另一个，都会有效。静态类型检查会在所有的例子中提供足够的确信，表明新系统和旧系统可以一样的工作。

文档。静态类型是被编译器检查过正确性的程序文档。不像普通的注释，类型标注永远都不会过期（至少如果包含它的源文件近期刚刚通过编译就不会）。更进一步说，编译器和集成开发环境可以利用类型标注提供更好的上下文帮助。举例来说，集成开发环境可以通过判定选中表达式的静态类型，找到类型的所有成员，并全部显示出来。

虽然静态类型对程序文档来说通常很有用，但在它们弄乱程序时也会显得很讨厌。标准意义上来说，有用的文档是那些程序的读者自己不可能很容易地从程序中想出来的。在如下的方法定义中：

```
def f(x: String) = ...
```

知道 f 的变量应该是 String 是有用的。另一方面，以下例子中两个标注至少有一个是讨厌的：

```
val x: HashMap[Int, String] = new HashMap[Int, String]()
```

很明显，x 是以 Int 为键，String 为值的 HashMap 这句话说一遍就够了；同样的句子没必要重复两遍。

Scala 有非常精于此道的类型推断系统，能让你省略几乎所有的通常被认为是讨厌的类型信息。前面的例子可以改写成以下两种方式：

[14]注：Dijkstra，“Notes on Structured Programming”，第 7 段。[Dij70]

```
val x = new HashMap[Int, String]()
val x: Map[Int, String] = new HashMap()
```

Scala 里的类型推断不止于此。实际上，就算用户代码丝毫没有显式类型也不稀奇。因此，Scala 程序经常看上去有点像是动态类型脚本语言写出来的程序。尤其在客户应用代码中作为预编译代码库控件的胶水代码时，表现得更为显著。而对于库控件来说不是这么回事，因为它们常常用到相当精妙的类型去使其适于灵活使用的模式。这是由代码的实际情况决定的。毕竟，构成可重用控件接口的成员的类型符号应该是显式给出的，因为它们构成了控件和它的使用者间契约的重要部分。

1.4 Scala 的根源

Scala 的设计受许多编程语言和研究思想的影响。事实上，全新的 Scala 的特点并不多，大多数都已经被以另外的形式用在其他语言中了。Scala 的革新主要在于这些东西是怎么放在一起的。在这部分里，我们罗列了对 Scala 设计的主要影响。列表并不全——因为围绕着编程语言的设计有太多的好点子，没办法全都列举在这里。

在最表层，Scala 采用了 Java 和 C#语法的大部分，而它们大部分借自于对 C 和 C++句法的改变。表达式、句子和代码块多数和 Java 一样，同样还有类、包和引用的语法[15]。除语法之外，Scala 还采用了 Java 的其他元素，如它的基本类型、类库和它的执行模式。

Scala 也从其他语言中借鉴了许多地方。它的统一对象模型由 Smalltalk 建立，后来又被 Ruby 发扬光大。他的通用嵌套的思想（几乎所有的 Scala 里的结构都能被嵌套进其他结构）还出现在 Algol、Simula 和最近的 Beta 与 gbeta 中。它的方法调用和字段选择的统一访问原则来自于 Eiffel。它函数式编程的处理方式在骨子里与以 SML、OCaml 和 F#为代表的 ML 家族语言很接近。许多 Scala 标准库里面的高阶函数同样也出现在 ML 或 Haskell 中。Scala 的隐式参数灵感激发自 Haskell 的类型类；它们用一种更经典的面向对象设定获得了类似的结果。Scala 基于 actor 的并发库几乎全是 Erlang 的思想。

Scala 不是第一种强调可扩展性的语言。能够横跨不同应用领域的可扩展语言的历史根源是 Peter Landin 在 1966 年的论文“The Next 700 Programming Languages”[16]（这篇论文中描述的语言 Iswim，与 Lisp 一同为开先河的函数式语言）中提出的。把前缀的操作符视为函数的奇思妙想可以回溯到 Iswim 和 Smalltalk。另一个重要的思想是允许将函数字面量（或代码块）作为参数，从而能让库定义控制结构。同样可以回溯到 Iswim 和 Smalltalk。Smalltalk 和 Lisp 两者都具有灵活的语法，广泛应用在建造内部的领域特化的语言里。C++是另一种能通过操作符重载和他的模板系统被改造和扩展的可扩展语言；与 Scala 相较，它建立在低级的、更面向系统的内核上。

Scala 也不是第一个集成函数式和面向对象编程的，尽管也许在这个方向上它走得最远。其他在 OOP

[15]注：对 Java 最大的背离在于类型标注的语法——是“variable: Type”而不是 Java 里的“Type variable”。Scala 的后修饰类型语法类似于 Pascal、Modula-2 或 Eiffel。主要原因与类型推断有关，它常可以让你忽略变量的类型或方法的返回类型。如果使用“variable: Type”这种语法，只要省略冒号和类型即可。但是在 C 风格中“Type variable”语法里类型没法去掉——没有定义开始的标记了。需要可选的关键字作为占位符替代缺少的类型（C# 3.0，为了实现某些类型推断，使用 var）。这种替代的关键字让人感觉更扎眼也有违 Scala 规规矩矩的目标。

[16]注：Landin，“The Next 700 Programming Languages”。[Lan66]

里同时具有函数式编程思想的包括 Ruby、Smalltalk 和 Python。在 Java 平台上，Pizza、Nice 和 Multi-Java 都用函数式思想扩展了类似 Java 的内核。同样也有一些接受了对象系统的基于函数式的语言；OCaml、F#和 PLT-Scheme 是其中的例子。

Scala 同时也为编程语言领域贡献了一些革新。举例来说，它的抽象类型提供了对泛型类型来说更面向对象的替代，它的特质允许灵活的控件组合，还有它的拆分器提供了独立于表达方式的模式匹配。这些革新已在近年编程语言会议中阐述在论文里了[17]。

1.5 小结

本章里，我们走马观花地了解了什么是 Scala 和它怎样对编程提供帮助。说句实话，Scala 不是一颗能魔法般提高生产力的银弹。进一步说，你需要更技巧地用 Scala，而这需要一些学习和练习。如果你是从 Java 转到 Scala，那么学习 Scala 最具挑战的部分就是深入到 Scala 的类型系统里（这部分比 Java 丰富得多）和它对函数式编程的支持。本书的目的是通过对你的引导，让你有平滑 Scala 的学习曲线，一次一小步。我们相信你会发现这是有回报的智力体验，它会扩展你的知识面并让你有一些不同的想法。希望你能同时获得使用 Scala 编程的动力和快乐。

下一章，我们将带领你开始学写 Scala 代码。

[17]注：更多信息请在参考文献中查阅[Ode03]，[Ode05]和[Emi07]。

第 2 章

Scala 入门初探

开始用 Scala 写点儿代码吧。在深入 Scala 教程之前，我们将使用两个章节来说明 Scala 的概况，并且更重要的是，带你写一些代码。我们鼓励你亲自测试所有本章及后续章节的代码例子，因为学习 Scala 最好的方法就是用它编程。

执行本章的例子之前应该先完成 Scala 语言发布包的安装。可以到 *http://www.scala-lang.org/downloads* 下载并根据你的操作系统平台选择安装向导。当然还可以使用 Eclipse、IntelliJ，或者 NetBeans 的 Scala 插件，不过本章我们假设你用的是从 `scala-lang.org` 拿到的 Scala 发布包[1]。

如果你是一位新接触 Scala 的有经验的程序员，那么接下来的两章可以让你知道如何使用 Scala 编写程序。但如果你缺乏编程经验，那么本章的内容对你来说或许会稍显有些神秘，为了加快你上手的速度，我们隐藏了一些细节说明。不过不用着急，所有的东西将在后续章节中详加解释（less “firehose”）。而且，在这两章里我们插入了相当多的脚注以提供更多的信息，这可以指引你到本书后续段落中去发现更详细的解释。

2.1 第一步　学习使用 Scala 解释器

开始学习 Scala 最简单的方法就是使用 Scala 解释器，它是编写 Scala 表达式和程序的交互式“shell”。只要在解释器里输入表达式，它就能计算并打印结果值。Scala 的交互式 shell 就叫做 `scala`。可以在命令提示符里输入 `scala`，执行效果如下[2]：

```
$ scala
Welcome to Scala version 2.7.2.
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

输入表达式，如 `1 + 2`，并键入回车之后：

```
scala> 1 + 2
```

解释器会打印：

```
res0: Int = 3
```

[1]注：本书的所有例子在 Scala 版本 2.7.2 之下测试通过。

[2]注：Windows 操作系统中，`scala` 命令需输入到“命令提示符”的 DOS 窗口。

这行输出信息依次为：

- 自动产生的或用户定义的名称说明计算的值（res0，表示结果0）；
- 冒号（:）及表达式的类型（Int）；
- 等号（=）；
- 表达式经计算得到的结果（3）。

Int类型指代了scala包的类Int。Scala的包与Java的类似：都可以对全局命名空间进行分区并提供了信息隐藏机制[3]。类Int的值对应着Java的int值。从更广泛意义上来说，所有的Java原始类型在scala包里都有对应的类。例如，scala.Boolean对应着Java的boolean。scala.Float对应着Java的float。在把Scala代码编译成Java字节码的时候，Scala编译器会代以使用Java的原语类型以便优化性能。

resX识别符还能使用在后续代码行中。例如，由于res0已被预定义为3，因此res0 * 3就返回9：

```
scala> res0 * 3
res1: Int = 9
```

接下来，按照惯例，向世界问个好，输入：

```
scala> println("Hello, world!")
Hello, world!
```

println函数在标准输出上打印传给它的字符串，就跟Java里的System.out.println一样。

2.2 第二步 变量定义

Scala有两种变量，val和var。val类似于Java里的final变量。一旦初始化了，val就不能再被赋值。相反，var如同Java里面的非final变量，可以在它的生命周期中被多次赋值。下面演示了如何定义val变量：

```
scala> val msg = "Hello, world!"
msg: java.lang.String = Hello, world!
```

这个语句引入了msg作为字符串"Hello, world!"的名字。它的类型是java.lang.String，因为Scala的字符串是由Java的String类实现的。

如果曾经声明过Java变量，你会发现一个很显著的差别：无论是java.lang.String还是String都没有出现在对val变量的定义中。这个例子演示了被称为类型推断（type inference）的能力，它能让Scala自动理解你省略了的类型。这里，你用字符串字变量初始化msg，因此Scala推断msg的类型是String。对于可以由Scala解释器（或编译器）自动推断类型的情况，就没有必要非得写出类型标注不可。当然这也并不绝对，你可以根据自己的想法写明定义的类型，而且或许某些时候就应该这么做。写

[3]注：如果你不熟悉Java的包，你可以认为包是提供类全名的机制。因为Int是scala包的成员，所以“Int”是这个类的简化名，“scala.Int”是它的全名。包的细节在第13章中描述。

明类型标注不但可以确保 Scala 编译器使用你想要用的类型，还有助于之后代码的阅读。Java 的变量类型写在名称之前，相反，Scala 的变量类型写在其名称之后，用冒号分隔。如：

```
scala> val msg2: java.lang.String = "Hello again, world!"
msg2: java.lang.String = Hello again, world!
```

或者，因为在 Scala 程序里 java.lang 类型的简化名[4]也是可见的，所以可以简写为：

```
scala> val msg3: String = "Hello yet again, world!"
msg3: String = Hello yet again, world!
```

回到原来的那个 msg，现在它已经定义好了，你可以任意使用它，如：

```
scala> println(msg)
Hello, world!
```

不过由于 msg 不是 var 类型变量，因此唯一不能做的就是再给它赋值[5]。例如，可以看看做如下尝试的时候编译器是怎么报错的：

```
scala> msg = "Goodbye cruel world!"
<console>:5: error: reassignment to val
       msg = "Goodbye cruel world!"
           ^
```

如果需要的是可以被重新赋值的变量，你应该使用 var，如下：

```
scala> var greeting = "Hello, world!"
greeting: java.lang.String = Hello, world!
```

由于 greeting 是 var 类型的变量，因此可以在之后重新对它赋值。比如说，如果你之后心情压抑，就可以把 greeting 修改为：

```
scala> greeting = "Leave me alone, world!"
greeting: java.lang.String = Leave me alone, world!
```

想在解释器中跨行输入语句的话，只要一行行写进去即可。如果输到行尾还没结束，解释器将在下一行回应一个竖线。

```
scala> val multiLine =
     |   "This is the next line."
multiLine: java.lang.String = This is the next line.
```

如果你发现了一些错误而解释器仍在等着更多的输入，你可以通过按两次回车键取消掉：

```
scala> val oops =
     |
     |
You typed two blank lines. Starting a new command.

scala>
```

本书后续部分，我们将省略竖线以便让代码更易于阅读（并易于从 PDF 电子书中复制粘贴到解释器里）。

[4]注：java.lang.String 的简化名是 String。

[5]注：然而在解释器中，你可以用一个之前已经使用了的名字定义新的 val。这种机制将在 7.7 节中解释。

2.3 第三步 函数定义

我们已经学过了 Scala 变量的使用方法，接下来学习如何写 Scala 的函数。如下：

```
scala> def max(x: Int, y: Int): Int = {
         if (x > y)x
         else y
       }
max: (Int,Int)Int
```

函数的定义以 def 开始。然后是函数名，本例中是 max，跟着是括号里带有冒号分隔的参数列表。函数的每个参数都必须带有前缀冒号的类型标注，因为 Scala 编译器（以及解释器，但之后我们将只说编译器）无法推断函数的参数类型。本例中，名叫 max 的函数带有两个参数，x 和 y，都是 Int 类型。在 max 参数列表的括号之后你会看到另一个“: Int”类型标注。这是 max 函数的结果类型（result type）[6] 定义。跟在函数结果类型之后的是一个等号和一对包含了函数体的花括号。本例中，函数仅包含一个条件表达式，选择 x 或 y 之中较大的值作为 max 函数的结果。正如我们在这里看到的，Scala 的条件表达式可以像 Java 的三元操作符那样生成结果值。例如，Scala 表达式“if (x > y) x else y”可以与 Java 的“(x > y) ? x : y”产生同样的结果。函数花括号前的等号可以提示我们，按照函数式编程的风格来看，函数是对能产生值的表达式的定义。函数的基本结构演示在图 2.1 里。

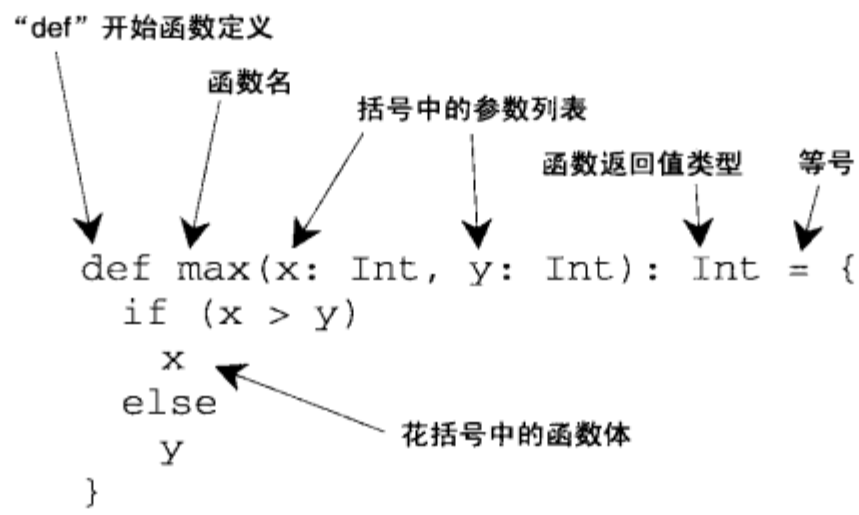


图 2.1 Scala 函数的基本构成

有时候 Scala 编译器需要函数结果类型的定义。比方说，如果函数是递归的[7]，那么函数结果类型就必须被明确地说明。然而在 max 的例子里，即使不写结果类型编译器也能够推断它[8]。同样，如果函数仅包含一个语句，那么连花括号都可以选择不写。这样 max 函数就可以写成：

```
scala> def max2(x: Int, y: Int) = if (x > y) x else y
max2: (Int,Int)Int
```

一旦函数完成定义，就可以通过函数名调用，如：

```
scala> max(3, 5)
res6: Int = 5
```

[6]注：在 Java 里，从方法里返回的值的类型被称为返回类型。在 Scala 里，同样的概念被叫做结果类型。

[7]注：如果一个方法调用自身，就称为递归。

[8]注：尽管如此，就算编译器不需要，显式说明函数结果类型也经常是个好主意，这种类型标注可以使代码便于阅读，因为读者不用研究了函数体之后再去猜结果类型。

以下是既不带参数也不返回有用结果的函数定义：

```
scala> def greet() = println("Hello, world!")
greet: ()Unit
```

函数 greet()定义之后，解释器输出返回 greet: ()Unit。“greet”当然是函数名，空白的括号说明函数不带参数。Unit 是 greet 的结果类型，指的是函数没有有效的返回值。Scala 的 Unit 类型比较类似于 Java 的 void 类型，而且实际上 Java 里返回 void 的方法都会被映射为返回 Unit 的 Scala 方法。因此结果类型为 Unit 的方法，并非是为了得到返回值，而是为了其他的运行效果（side effect）。greet()的例子里，效果就是在标准输出上打印一句话。

下一步，需要把 Scala 代码写在文件中并作为脚本执行。如果想要离开解释器，输入:quit 或者:q。

```
scala> :quit
$
```

2.4 第四步 编写 Scala 脚本

尽管 Scala 的设计目的是帮助程序员建造大型系统，但它也能适应于制造小型的脚本。脚本就是一种经常会被执行的放在文件中的句子序列。例如把以下代码放在 hello.scala 文件中：

```
println("Hello, world, from a script!")
```

然后运行[9]：

```
$ scala hello.scala
```

于是系统又会输出：

```
Hello, world, from a script!
```

Scala 脚本的命令行参数保存在名为 args 的 Scala 数组中。Scala 里，数组以零开始，可以通过在括号里指定索引值来访问数组元素。Scala 里数组 steps 的第一个元素是 steps(0)，而不是像 Java 那样的 steps[0]。现在，把以下内容写到新文件 helloarg.scala 中测试一下：

```
// 向第一个参数打问好
println("Hello, " + args(0) + "!")
```

然后运行：

```
$ scala helloarg.scala planet
```

这条命令里，命令行参数"planet"被传递给脚本，并通过访问 args(0)获得。因此，你会看到：

```
Hello, planet!
```

请注意这个脚本包含了一条注释。Scala 编译器忽略从//开始到行尾截止的以及在/*和*/之间的字符。本例还演示了 String 使用+操作符的连接。就像预期的那样，表达式"Hello, "+"world!"将得到字符串"Hello, world!" [10]。

---

[9]注：通过使用制式（pound-bang）语法，在 Unix 和 Windows 里你可以不用输入“scala”就运行脚本，附录 A 中对此作了说明。

[10]注：空格可以放在加号的旁边，如"Hello, " + "world!"。然而本书中，我们去掉了‘+’和字符串字面量的空格。

2.5 第五步 用while做循环；用if做判断

请在printargs.scala文件里输入以下代码测试while：

```
var i = 0
while (i < args.length) {
  println(args(i))
  i += 1
}
```

30

> **注意**
>
> 虽然本节的例子有助于解释while循环，但这并不是Scala推荐的代码风格。下一段中，你将看到避免用索引枚举数组的改善方式。

这个脚本以变量定义var i=0开始。类型推断根据初始值0的类型认定i是scala.Int。下一行的while结构将重复执行代码块（花括号之间的代码），直到布尔表达式i<args.length检查结果为否。args.length给出了args数组的长度。代码块包含两行语句，每行都缩进两个空格，这是Scala推荐的缩进风格。第一行，println(args(i))，输出第i个命令行参数。第二行，i+=1，让i变量值自增。请注意Java的++i和i++在Scala里不起作用，要在Scala里得到同样效果，必须要么写成i=i+1，要么写成i+=1。用下列命令运行这个脚本：

```
$ scala printargs.scala Scala is fun
```

你将看到：

```
Scala
is
fun
```

为了让输出变得更有意思，可以把下列代码输入到新文件echoargs.scala：

```
var i = 0
while (i < args.length) {
  if (i != 0)
    print(" ")
  print(args(i))
  i += 1
}
println()
```

31

新版本里，print调用替代了println调用，这样参数就可以在同一行里输出。为了更适合于阅读习惯，可以用if(i != 0)做检查，在除第一个参数之外的每个参数前插入空格。由于首次做while循环时i != 0不成立，因此头一个参数之前不会有空格输出。最后，应该在程序末尾多加一个println，用以在所有参数的输出之后执行换行。最后运行脚本：

```
$ scala echoargs.scala Scala is even more fun
```

就能得到：

```
Scala is even more fun
```

这样的输出结果就非常漂亮了。注意Scala和Java一样，必须把while或if的布尔表达式放在括

号里。（换句话说，就是不能像 Ruby 语言那样写成：`if i < 10`，而是必须写成 `if (i < 10)`。）另一个与 Java 相似的地方是，如果代码块仅有一行语句，就像 `echoargs.scala` 里的那个 `if` 语句，那么花括号就可以不写。并且尽管 Scala 也和 Java 一样用分号分隔语句，但是 Scala 的分号经常是可选的，因此你的右小手指可以不用那么累。不过如果你喜欢，也可以把所有可选部分都加上：

```
var i = 0;
while (i < args.length) {
  if (i != 0) {
    print(" ");
  }
  print(args(i));
  i += 1;
}
println();
```

2.6 第六步 用 foreach 和 for 做枚举

你有可能不知道，前一步里写的 `while` 循环的编码风格被称为是指令式（imperative）编程。这种风格常见于 Java、C++和 C 这些语言，或者逐条执行指令，或者循环枚举，并经常改变不同函数之间的共享状态。Scala 里允许使用指令式的编程风格，但随着你对 Scala 的深入了解，你可能会发现自己逐渐倾向于一种更为函数式（functional）的编程风格。实际上，本书的主旨之一就是帮助你学习并掌握使用这种函数式的编程风格。

函数式语言的主要特征之一就是，函数是头等结构，而这也正是 Scala 的主要特征。举例来说，另一种更为简洁的打印每个命令行参数的方法是：

```
args.foreach(arg => println(arg))
```

这行代码对 `args` 调用了 `foreach` 方法，并把函数作为参数传入，在这里是名为 `arg` 的函数字面量（function literal）。函数体是 `println(arg)`。把上述代码输入到新文件 `pa.scala`，并执行：

```
$ scala pa.scala Concise is nice
```

将会看到：

```
Concise
is
nice
```

在这个例子中，Scala 解释器可以推断 `arg` 的类型是 `String`，因为 `String` 是调用 `foreach` 的那个数组的元素类型。如果想要表达得更明确的话，可以给 `arg` 加上类型名，不过这样做需要把参数部分写在括号里（总之这是语法的正规形式）：

```
args.foreach((arg: String) => println(arg))
```

这个脚本的运行结果与前一个相同。

对于喜欢简单扼要的人，这段代码还有一种更为紧凑有效的写法。如果函数字面量只有一行语句并

只带一个参数，那么甚至连指代参数都不需要[11]。可以写成：

33

```
args.foreach(println)
```

综上所述，函数字面量的语法构成包含括号及命名参数列表，右箭头以及函数体。如图2.2中所示。

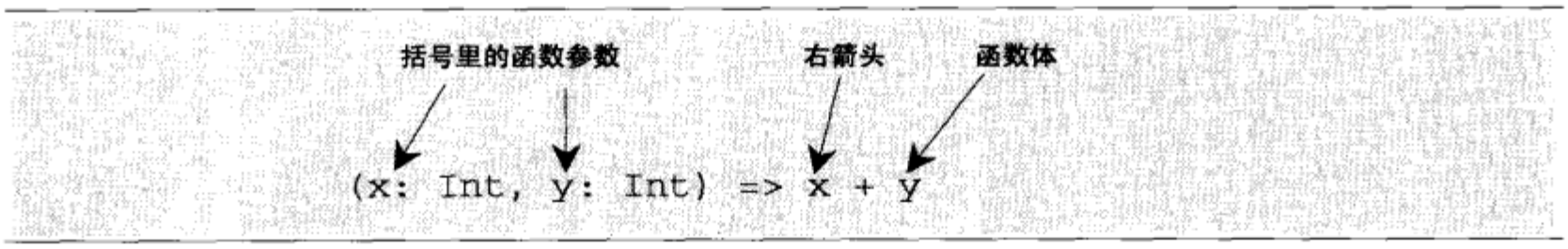


图2.2 Scala函数字面量的语法

现在，你或许奇怪那些指令式语言如Java或C里常用的for循环在哪里？为了努力引导你学习和使用函数式的编码风格，Scala只有一个指令式for（称为for表达式（“for expression”））的函数型与之近似。第73节将详细说明它的使用方法和适用范围，这里仅演示一个例子。创建新文件forargs.scala，输入以下代码：

```
for (arg <- args)
  println(arg)
```

例子中“for”之后的括号包含arg<-args[12]。<-右侧的是已经在前面见过的args数组。<-左侧的“arg”，是val的名称（不是var）。（这里一定是val，因此只写arg即可，无须写成val arg。）尽管arg可能感觉像var，因为每次枚举都会得到新的值，但这的确是val，因为它不能在for表达式的函数体中被重新赋值。所以，对于args数组的每个元素，枚举的时候都会创建并初始化新的arg值，然后调用执行for的函数体。

运行forargs.scala脚本：

34

```
$ scala forargs.scala for arg in args
```

可以看到：

```
for
arg
in
args
```

虽然Scala的for表达式还有更多用法，但是这个例子足以说明问题。我们将在73节和第23章中对for表达式做更详细的介绍。

2.7 小结

35

本章我们学习了Scala的基础知识并开始尝试编写代码。下一章将继续Scala的概况介绍和讨论一些更为深入的话题。

[11]注：这种简写被称为偏函数（partially applied function），将在8.6节里描述。
[12]注：你可以认为<-符号代表“其中”。如果要读for(arg<-args)，就读做“对于args中的arg”。

第 3 章

Scala 入门再探

本章除了继续前一章对 Scala 的概况介绍之外，我们还将介绍一些更先进的特征。读完本章，你将能够使用 Scala 编写有用的脚本。和前一章一样，我们建议你边阅读边测试附带的例子。掌握 Scala 的最好办法还是不断使用它。

3.1 第七步 使用类型参数化数组（Array）

Scala 里使用 new 实例化对象（或者叫类实例）。实例化过程中，可以用值和类型使对象参数化（parameterize）。参数化的意思是指在创建实例的同时完成对它的“设置”。使用值参数化实例可以通过把值传递给构造器的圆括号来实现。例如，下面的代码将创建值参数为 "12345" 的 java.math.BigInteger 实例：

```
val big = new java.math.BigInteger("12345")
```

使用类型参数化实例可以通过把一个或更多类型指定到基础类型后的方括号里来实现。具体参见清单 3.1。这个例子中，greetStrings 的类型是 Array[String]（字符串数组），并且由于创建数组的值参数为 3（第一行代码中），因此其初始长度为 3。执行清单 3.1 将再次得到 Hello, world!祝词。注意，当同时用类型和值参数化实例的时候，应该首先写方括号和类型参数，然后再写圆括号和值参数。

```
val greetStrings = new Array[String](3)

greetStrings(0) = "Hello"
greetStrings(1) = ","
greetStrings(2) = "world!\n"

for (i <- 0 to 2)
  print(greetStrings(i))
```

清单 3.1 用类型参数化数组

> **注意**
>
> 尽管我们使用了清单 3.1 做重要概念的演示，但并不代表这是 Scala 里创建和初始化数组的推荐方式。详情参见清单 3.2。

对于喜欢循规蹈矩的人来说，清单 3.1 的第一行里 greetStrings 的类型可以完整定义为：

```
val greetStrings: Array[String] = new Array[String](3)
```

得益于 Scala 的类型推断，这两种写法在语义上完全一致。不过完整定义的形式能够更有效地说明

类型参数（方括号包含的类型名）是实例类型的组成部分，而值参数（圆括号包含的值）不是：greetStrings的类型是Array[String]，不是Array[String](3)。

接下来的三行初始化了greetStrings数组的每个元素：

```
greetStrings(0) = "Hello"
greetStrings(1) = ","
greetStrings(2) = "world!\n"
```

正如前面提到的，Scala的数组是通过索引值加圆括号进行的访问，而不是像Java那样把索引值放在方括号里。所以数组的首个元素是greetStrings(0)，而不是greetStrings[0]。

这三行代码同时也指明了Scala中val含义的重要概念。如果用val定义变量，那么这个变量就不能被重新赋值，但变量指向的对象内部却仍可以改变。所以本例中，greetStrings对象不能被重新赋值成别的数组；它将永远指向初始化时指定的那个Array[String]实例。但是Array[String]的内部元素始终能被修改，因此数组本身是可变的。

清单3.1的最后两行包含一个for表达式，用来依次输出每个greetStrings数组元素。

```
for (i <- 0 to 2)
  print(greetStrings(i))
```

这个for表达式的第一行代码说明了Scala的另一个基本规则：方法若只有一个参数，调用的时候就可以省略点及括号。本例中的to实际上是仅带一个Int参数的方法。代码0 to 2被转换成方法调用(0).to(2)[1]。请注意这个语法只有在明确指定方法调用的接受者时才有效。例如不可以写成“println 10”，但是可以写成“Console println 10”。

从技术层面上来说，Scala没有操作符重载，因为它根本没有传统意义上的操作符。取而代之的是，诸如+，-，*和/这样的字符，可以用来做方法名。因此，当我们在第一步中往Scala解释器里输入1 + 2的时候，实际是在Int对象1上调用名为+的方法，并把2当作参数传给它，如图3.1所示。当然1 + 2更为传统的语法格式可以写成(1).+(2)。

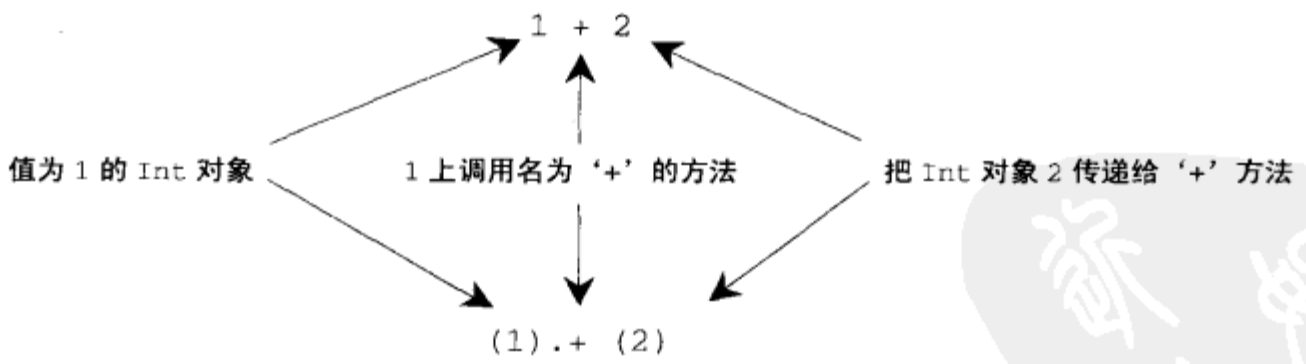


图3.1 Scala里所有的操作符都是方法调用

这里要说明的另一问题是为什么在Scala里要用括号访问数组。与Java相比，Scala鲜有特例。与Scala其他的类一样，数组也只是类的实例。用括号传递给变量一个或多个值参数时，Scala会把它转换

[1]注：这个to方法实际上返回的不是数组而是包含了值0，1和2的，可以让for表达式遍历的序列。序列和其他集合类将在第17章描述。

成对 apply 方法的调用。于是 greetStrings(i)转换成 greetStrings.apply(i)。所以 Scala 里访问数组的元素也只不过是跟其他的一样的方法调用。这个原则不是只对于数组：任何对于对象的值参数应用将都被转换为对 apply 方法的调用。当然前提是这个类型实际定义过 apply 方法。所以这不是特例，而是通用法则。

与之相似的是，当对带有括号并包括一到若干参数的变量赋值时，编译器将使用对象的 update 方法对括号里的参数（索引值）和等号右边的对象执行调用。例如：

```
greetStrings(0) = "Hello"
```

将被转化为：

```
greetStrings.update(0,"Hello")
```

因此，下列 Scala 代码与清单 3.1 语义一致：

```
val greetStrings = new Array[String](3)

greetStrings.update(0,"Hello")
greetStrings.update(1,",")
greetStrings.update(2,"world!\n")

for (i <- 0.to(2))
  print(greetStrings.apply(i))
```

Scala 对任何事都追求概念的简洁性，从数组到表达式，包括带有方法的对象。你不必记住太多特例，如 Java 里存在于原始类型和相应的包装类之间的，或者数组和普通对象之间的差别。而且这种统一并不是以严重损害性能为代价的。Scala 编译器会尽可能在编译完成的代码里利用 Java 数组、原始类型和原生的数字计算方法。

到目前为止，尽管这个例子能够完成编译且运行良好，但它不够简洁。Scala 提供了通常可以用于生产环境的更为简洁的创造和初始化数组的方法，如清单 3.2 所示。仅用一行代码就创建了长度为 3 的新数组，并用字符串"zero"，"one"和"two"实现初始化。编译器根据传递的值类型（字符串）推断数组的类型是 Array[String]。

```
val numNames = Array("zero", "one", "two")
```

清单 3.2 创造和初始化数组

清单 3.2 实际就是调用了创造并返回新数组的 apply 工厂方法。apply 方法可以有不定个数的参数[2]，定义在 Array 的伴生对象（companion object）中。第 4.3 节里会学到更多关于伴生对象的知识。对于 Java 程序员来说，可以把它理解为调用 Array 类的静态方法 apply。完整的写法是：

```
val numNames2 = Array.apply("zero","one","two")
```

3.2 第八步 使用列表（List）

方法没有副作用是函数式风格编程的重要理念，计算并返回值应该是方法唯一的目的。这样做的好

[2]注：变长参数列表，也可以称为重复参数（repeated parameters），在 8.8 节中描述。

处之一是方法之间耦合度降低，因此更加可靠和易于重用；好处之二（在静态类型语言里）是方法的参数和返回值都经过类型检查器的检查，因此可以比较容易地根据类型错误推断其中隐含的逻辑错误。而这种函数式编程思想在面向对象编程中的应用也就意味着对象的不可改变性。

41

正如我们之前所见的，Scala数组是可变的同类对象序列，例如，Array[String]的所有对象都是String。而且尽管数组在实例化之后长度固定，但它的元素值却是可变的。所以说数组是可变的。

至于不可变的同类对象序列，Scala的列表类（List）才是。和数组一样，List[String]仅包含String。但Scala的scala.List不同于Java的java.util.List，一旦创建了就不可改变。实际上，Scala的列表是为了实现函数式风格的编程而设计的。创建的方法如清单3.3所示：

```
val oneTwoThree = List(1, 2, 3)
```

清单3.3 创造和初始化列表

以上的代码建立了名为oneTwoThree的值，并初始化或整数元素为1、2和3的新List[Int][3]。列表本身不可变，因此看上去有些像Java的String：对某个列表调用方法时，似乎这个列表发生了改变，而实际上只是用新的值重建了列表然后再返回。例如，列表类中定义了“:::”方法实现叠加功能。用法如下：

```
val oneTwo = List(1,2)
val threeFour = List(3,4)
val oneTwoThreeFour = oneTwo ::: threeFour
println("" + oneTwo + " and " + threeFour + " were not mutated.")
println("Thus," + oneTwoThreeFour + " is a new List.")
```

执行之后可以得到如下返回：

```
List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new List.
```

列表类最常用的操作符或许是‘::’，发音为“cons”。它可以把新元素组合到现有列表的最前端，然后返回作为执行结果的新列表。执行代码：

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```

42

将得到：

```
List(1,2,3)
```

> **注意**
>
> 表达式“1 :: twoThree”中，::是右操作数twoThree的方法。你或许会质疑是否::方法的关联性有错误，不过请记住这只是个简单的规则：如果方法使用操作符来标注，如a * b，那么左操作数是方法的调用者，可以改写成a.*(b)——除非——方法名以冒号结尾。这种情况下，方

[3]注：这里不写成new List，因为“List.apply()”是定义在scala.List伴生对象中的工厂方法。第十一步里有更多关于伴生对象的说明。

法被右操作数调用。因此，1 :: twoThree 里，::方法的调用者是 twoThree，1 是方法的传入参数，因此可以改写成：twoThree.::(1)。

5.8 节中将具体说明操作符关联性的细节。

因为 Nil 是空列表的简写，所以可以使用 cons 操作符把所有元素都串起来，并以 Nil 作结尾来定义新列表[4]。例如可以用以下的方法产生与上文同样的输出，“List(1, 2, 3)”：

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

Scala 的 List 类定义了很多有用的方法，其中的部分列举在表 3.1 中。完整说明请参见第 16 章。

为什么列表不支持添加（append）操作？

List 类没有提供 append 操作，因为随着列表变长，append 的耗时将呈线性增长，而使用::做前缀则仅耗用固定的时间。如果你想通过添加元素来构造列表，你的选择是先把它们前缀进去，完成之后再调用 reverse；或使用 ListBuffer，一种提供 append 操作的可变列表，完成之后调用 toList。ListBuffer 将在 22.2 节中描述。

表 3.1 List 的一些方法和作用

方法名	方法作用
List()或 Nil	空 List
List("Cool", "tools", "rule")	创建带有三个值"Cool", "tools"和"rule"的新 List[String]
val thrill = "Will"::"fill"::"until"::Nil	创建带有三个值"Will", "fill"和"until"的新 List[String]
List("a", "b") ::: List("c", "d")	叠加两个列表（返回带"a", "b", "c", "d"的新 List[String]）
thrill(2)	返回在 thrill 列表上索引为 2（基于 0）的元素（"until"）
thrill.count(s => s.length == 4)	计算长度为 4 的 String 元素个数（2）
thrill.drop(2)	返回去掉前两个元素的 thrill 列表（List("until")）
thrill.dropRight(2)	返回去掉后两个元素的 thrill 列表（List("Will")）
thrill.exists(s => s == "until")	判断是否有值为"until"的字符串元素在 thrill 里（true）
thrill.filter(s => s.length == 4)	返回长度为 4 的元素依次组成的新列表（List("Will", "fill")）
thrill.forall(s => s.endsWith("1"))	判断是否 thrill 列表里所有元素都以"1"结尾（true）
thrill.foreach(s => print(s))	对 thrill 列表每个字符串执行 print 语句（"Willfilluntil"）
thrill.foreach(print)	与前相同，不过更简洁（同上）

[4]注：要在最后用到 Nil 的理由是::是定义在 List 类上的方法。如果你想只是写成 1 :: 2 :: 3，由于 3 是 Int 类型，没有::方法，因此会导致编译失败。

续表

方法名	方法作用
thrill.head	返回 thrill 列表的第一个元素（"Will"）
thrill.init	返回 thrill 列表除最后一个以外其他元素组成的列表（List("Will", "fill")）
thrill.isEmpty	返回 thrill 列表是否为空（false）
thrill.last	返回 thrill 列表的最后一个元素（"until"）
thrill.length	返回 thrill 列表的元素数量（3）
thrill.map(s => s + "y")	返回由 thrill 列表里每一个 String 元素都加了"y"构成的列表（List("Willy", "filly", "untily")）
thrill.mkString(", ")	返回用列表的元素组成的字符串（"will, fill, until"）
thrill.remove(s => s.length == 4)	返回去除了 thrill 列表中长度为 4 的元素后的元素依次组成的新列表（List("until")）
thrill.reverse	返回由 thrill 列表的元素逆序组成的新列表（List("until", "fill", "Will")）
thrill.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)	返回 thrill 列表元素按照第一个字符的字母小写排序之后依次组成的新列表（List("fill", "until", "Will")）
thrill.tail	返回 thrill 列表中除第一个元素之外依次组成的新列表（List("fill", "until")）

3.3 第九步 使用元组（Tuple）

元组也是很有用的容器对象。与列表一样，元组也是不可变的；但与列表不同，元组可以包含不同类型的元素。例如列表只能写成 List[Int]或 List[String]，但元组可以同时拥有 Int 和 String。元组适用场景很多，比方说，如果需要在方法里返回多个对象。Java 里的做法是创建 JavaBean 以包含多个返回值，Scala 里可以仅返回元组。而且做起来也很简单：只要把元组实例化需要的对象放在括号里，并用逗号分隔即可。元组实例化之后，可以用点号、下划线和基于 1 的索引访问其中的元素。如清单 3.4 所示：

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

清单 3.4 创造和使用元组

第一行创建了元组，第一个元素是 Int 类型的值 99，第二个是 String 类型的值"luftballons"。Scala 推断元组类型为 Tuple2[Int, String]，并赋给值 pair。第二行访问_1 字段，获得元组第一个元素的输出 99。第二行的点号“.”与访问字段或调用方法的点没有区别。本例用来访问名叫_1 的字段。这段代码执行的结果为：

```
99
Luftballons
```

元组的实际类型取决于它含有的元素数量和这些元素的类型。因此，(99, "Luftballons")的类型是 Tuple2[Int, String]。('u', 'r', 'the', 1, 4, "me")是 Tuple6[Char, Char, String, Int, Int, String]。[5]

46

访问元组的元素

你或许想知道为什么不能用访问列表的方法来访问元组，如 pair(0)。那是因为列表的 apply 方法始终返回同样的类型，但元组里的类型不尽相同。_1 的结果类型可能与_2 的不一致，诸如此类，因此两者的访问方法也不一样。另外，这些_N 的索引是基于 1 的，而不是基于 0 的，这是因为对于拥有静态类型元组的其他语言，如 Haskell 和 ML，从 1 开始是传统的设定。

3.4 第十步 使用集（set）和映射（map）

Scala 致力于充分利用函数式和指令式风格两方面的好处，因此它的集合（collection）库区分为可变类型和不可变类型。例如，array 具有可变性，而 list 保持不变。对于 set 和 map 来说，Scala 同样有可变的和不可变的，不过并非各提供两种类型，而是通过类继承的差别把可变性差异蕴含其中。

例如，Scala 的 API 包含了 set 的基本特质（trait），特质这个概念接近于 Java 的接口（interface）（第 12 章有更多说明）。Scala 还提供了两个子特质，分别为可变 set 和不可变 set。如图 3.2 所示（见下页），这三个特质都共享同样的简化名，set。然而它们的全称不一样，每个特质都在不同的包里。Scala 的 API 里具体的 set 类，如图 3.2 的 HashSet 类，各有一个扩展了可变的和另一个扩展不可变的 Set 特质。（Java 里面称为“实现”了接口，而在 Scala 里面称为“扩展”或“混入（mix in）”了特质。）因此，使用 HashSet 的时候，可以根据需要选择可变的或不可变的类型。set 的基本构造方法如清单 3.5 所示：

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
```

清单 3.5 创造、初始化和使用不可变集

47

第一行定义了名为 jetSet 的新变量，并初始化为包含两个字符串"Boeing"和"Airbus"的不可变集。如代码所示，scala 中创建 set 的方法与创建 list 和 array 的类似：通过调用 Set 伴生对象的 apply 工厂方法。在上面的例子里，对 scala.collection.immutable.Set 的伴生对象调用了 apply 方法，返回了默认的不可变 Set 的实例。Scala 编译器推断其类型为不可变 Set[String]。

要加入新变量，可以对 jetset 调用+，并传入新元素。可变的和不可变的集都提供了+方法，但结果不同。可变集把元素加入自身，而不可变集则创建并返回包含了添加元素的新集。清单 3.5 中使用的是不可变集，因此+调用将产生一个全新集。所以只有可变集提供的才是真正的+=方法，不可变集不是。例子中的第二行，“jetSet += "Lear"”，实质上是下面写法的简写：

[5]注：尽管理论上你可以创建任意长度的元组，然而当前 Scala 库仅支持到 Tupe22。

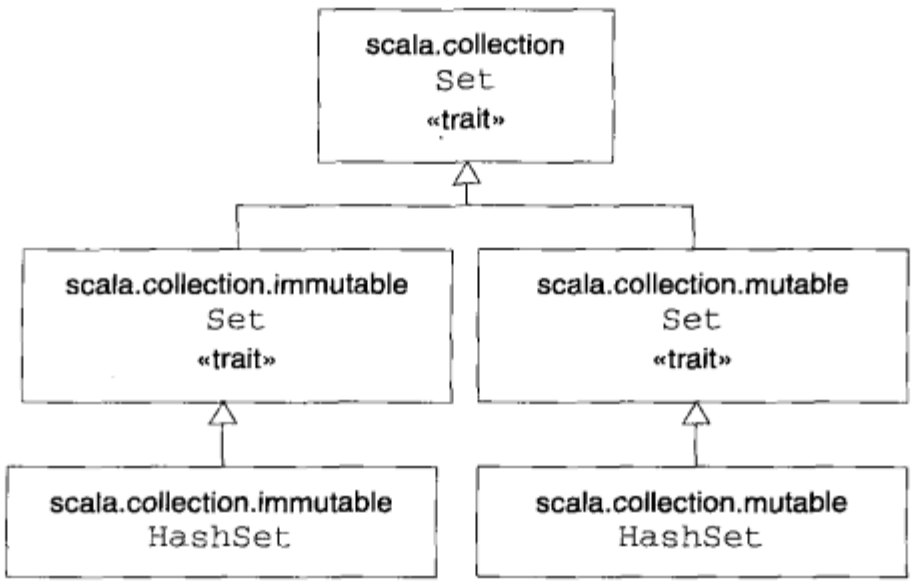


图 3.2 Scala 的 set 类层级

```
jetSet = jetSet + "Lear"
```

48

因此这里实际是用包含了"Boeing"，"Airbus"和"Lear"的新集重新赋值了 jetSet 变量。最后，代码打印输出这个集是否包含字符串"Cessna"。（诚如所料，输出 false。）

如果要定义不可变 Set，就需要加入引用（import），如清单 3.6 所示：

```
import scala.collection.mutable.Set

val movieSet = Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

清单 3.6 创建、初始化和使用可变集

第一行里引用了可变 Set。与 Java 类似，引用语句有助于用简单名，如 Set，替代较长的全标识名。所以，编译器知道第三行的 Set 是指 scala.collection.mutable.Set。然后再用包含"Hitch"和"Poltergeist"的新可变集对 movieSet 做了初始化。第四行对 movieSet 调用+=方法添加了"Shrek"。这在前面也刚提到过，只有可变集提供的才是真正的+=方法。这行代码还可以写成 movieSet.+=("Shrek")[6]。

尽管对于大多数情况来说，使用可变的或不可变的 Set 工厂方法构造对象就已经足够了，但偶尔也难免需要指定 set 的具体类型。幸运的是，构造的语法相同：只要引用所指定的类，并使用伴生对象的工厂方法即可。例如，要使用不可变的 HashSet，可以这么做：

```
import scala.collection.immutable.HashSet

val hashSet = HashSet("Tomatoes", "Chilies")
println(hashSet + "Coriander")
```

49

另一种 Scala 里常用的集合类是 Map。和 set 一样，Scala 采用了类继承机制提供了可变的和不可变的两种版本的 Map，参见图 3.3。map 的类继承机制看上去和 set 的很像。scala.collection 包

[6]注：因为清单 3.6 里的集是可变的，所以不需要对 movieSet 重新赋值，所以定义为值（val）。相对的，在清单 3.5 中对可变集使用+=需要对 jetSet 重新赋值，因此定义为变量（var）。

里面有一个基础 Map 特质和两个子特质 Map：可变的 Map 在 scala.collection.mutable 里，不可变的在 scala.collection.immutable 里。

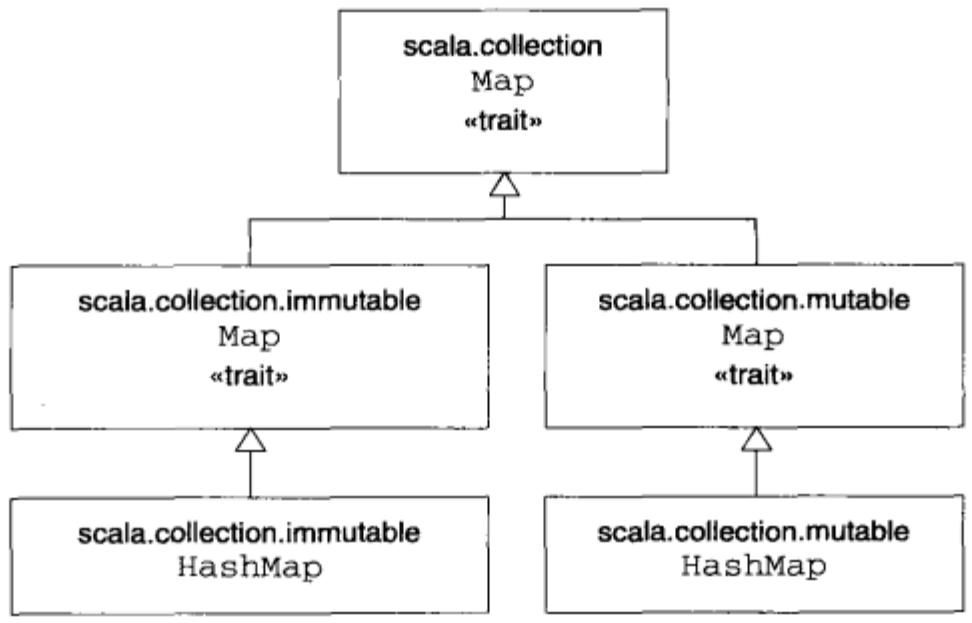


图 3.3 Scala 的 map 类层级

Map 的实现类，例如类继承图 3.3 里的 HashMap，扩展了可变或不可变特质之一。而且同样可以使用 Array、List 或 Set 同样的工厂方法构造和初始化。如清单 3.7 所示：

```
import scala.collection.mutable.Map

val treasureMap = Map[Int, String]()
treasureMap += (1 ->"Go to island.")
treasureMap += (2 ->"Find big X on ground.")
treasureMap += (3 ->"Dig.")
println(treasureMap(2))
```

清单 3.7 创造、初始化和使用可变映射

代码中首先引用了可变的 Map。然后定义了值 treasureMap，并初始化为以整数为键和以字符串为值的可变 Map。因为没有向工厂方法传递任何值（“Map[Int, String]()”的括号里面是空的），所以 Map 为空[7]。之后三行用->和+=方法向 Map 里添加键值对。像之前所演示的，Scala 编译器把二元操作符表达式 1 -> "Go to island"转换为(1).->("Go to island.")。因此，对于输入 1 -> "Go to island."来说，实际是在对 Int 值 1 调用->方法，并传入参数 String 值"Go to island."。Scala 的任何对象都能调用->方法，并返回包含键值对的二元组[8]。然后这个二元组被交给 treasureMap 对象的+=方法。代码最终输出打印 treasureMap 中与键 2 对应的值。执行以上这段代码可以得到：

```
Find big X on ground.
```

对于不可变 map 来说，不必引用任何类，因为不可变 map 是默认的，清单 3.8 展示了这个例子：

[7]注：在这里，类型参数化“[Int, String]”，对清单 3.7 来说是必须的，因为没有任何值被传递给工厂方法，编译器无法推断映射的类型参数。与之相对的是，清单 3.8 之中，编译器可以根据传递给 map 工厂方法的值推断参数类型，因此就不需要类型参数了。

[8]注：任何对象都能调用->的机制被称为隐式转换，将在第 21 章里涉及。

```
val romanNumeral = Map(
  1 ->"I", 2 ->"II", 3 ->"III", 4 ->"IV", 5 ->"V"
)
println(romanNumeral(4))
```

清单3.8 创造、初始化和使用不可变映射

由于没有引用，因此代码第一行里使用了默认的 Map: scala.collection.immutable.Map 定义对象。并把五个键值元组传给工厂方法，得到返回包含这些键值对的不可变 Map。执行以上代码，得到打印输出“IV”。

51

3.5 第十一步 学习识别函数式风格

第 1 章里提到过，Scala 允许指令式的编程风格，但是鼓励采用函数式的风格。如果你是从指令式的背景转到 Scala 来的——例如，如果你是 Java 程序员——那么学习 Scala 将面对的主要挑战就是理解怎样用函数式的风格编程。我们明白这种转变会很困难，在本书中我们将竭尽所能引领你。不过，俗语说“师傅领进门，修行在个人”，这同样需要你不懈的努力。如果你来自于指令式语言的环境，我们相信学习用函数式风格编程将不仅让你成为更好的 Scala 程序员，而且还能拓展你的视野并使你成为更广泛意义上的好程序员。

我们的首要工作是识别这两种风格在代码上的差异。大致可以说，如果代码包含了任何 var 变量，那它可能就是指令式的风格。如果代码根本就没有 var——就是说仅仅包含 val——那它或许是函数式的风格。因此向函数式风格转变的方式之一，就是尝试不用任何 var 编程。

如果你来自于指令式的语言环境，如 Java、C++，或者 C#，你或许会认为 var 是很正统的而 val 比较特殊。相反，如果你来自于函数式语言环境，如 Haskell、OCamel 或 Erlang，或许你会认为 val 很正常而对于 var 就有一种抵触情绪了。然而在 Scala 看来，val 和 var 只不过是工具箱里两种不同的工具。它们都很有用，没有一个天性邪恶。Scala 鼓励你学习 val，但最终做到“物尽其用”的原则。而且就算你赞同这种平衡哲学，也还是会发现想要搞明白如何从代码中去掉 var 在最初是很有挑战的事情。

下面是改编于第 2 章的 while 循环例子，它使用了 var，因此属于指令式风格：

```
def printArgs(args: Array[String]): Unit = {
  var i = 0
  while (i < args.length) {
    println(args(i))
    i += 1
  }
}
```

52

你可以通过去掉 var 的办法把这个代码变得更函数式风格，例如，像这样：

```
def printArgs(args: Array[String]): Unit = {
  for (arg <- args)
    println(arg)
}
```

或这样：

```
def printArgs(args: Array[String]): Unit = {
  args.foreach(println)
}
```

这个例子说明了减少使用 var 的一个好处。重构后（更函数式）的代码比原来（更指令式）的代码更简洁、明白，也更少有机会犯错。Scala 鼓励函数式风格的原因，实际上也就是因为函数式风格可以帮助你写出更易读懂，同样也是更不易犯错的代码。

当然，这段代码仍有修改的余地。重构后的 printArgs 方法并不是纯函数式的，因为它有副作用——本例中的副作用就是打印到标准输出流。识别函数是否有副作用的地方就在于其结果类型是否为 Unit。如果某个函数不返回任何有用的值，也就是说如果返回类型为 Unit，那么这个函数唯一能产生的作用就只能是通过某种副作用。而函数风格的方式应该是定义对需打印的 arg 进行格式化的方法，不过仅返回格式化之后的字符串，如清单 3.9 所示：

```
def formatArgs(args: Array[String]) = args.mkString("\n")
```

清单 3.9 没有副作用或 var 的函数

现在才是真正函数式风格的了：完全没有副作用或 var 的 mkString 方法，能在任何可枚举的集合类型（包括数组，列表，集和映射）上调用，返回由每个数组元素调用 toString，并把传入字符串做分隔符组成的字符串。因此如果 args 包含了三个元素，"zero"、"one"和"two"，formatArgs 将返回"zero\none\ntwo"。当然，这个函数并不像 printArgs 方法那样能够实际完成打印输出，但可以简单地把它的结果传递给 println 来实现：

```
println(formatArgs(args))
```

每个有用的程序都会有某种形式的副作用，否则就不可能向程序之外提供什么有价值的东西。我们提倡无副作用的方法是为了鼓励你尽量设计出没有副作用代码的程序。这种方式的好处之一是可以有助于你的程序更容易测试。举例来说，要测试前面给出的任何一个有副作用的 printArgs 方法，你将需要重定义 println，捕获传递给它的输出，再检查结果。相反，对于 formatArgs 来说你可以直接检查它的返回结果：

```
val res = formatArgs(Array("zero", "one", "two"))
assert(res == "zero\none\ntwo")
```

Scala 的 assert 方法检查传入的 Boolean 表达式，如果结果为假，抛出 AssertionError；否则 assert 就什么也不做，安静地返回。第 14 章将学习更多关于断言和测试的东西。

说了这么多，不过还是请牢记：不管是 var 还是副作用都不是天生邪恶的。Scala 不是只能使用函数式风格编程的纯函数式语言。它是这两种风格的混合式语言。甚至有时你会发现指令式风格能更有效地解决手中的问题，那就使用指令式风格，别犹豫。不过，为了帮助你学习如何不使用 var 编程，第 7 章中我们会演示若干把含有 var 的代码转换为 val 的特例。

Scala程序员的平衡感

崇尚 val，不可变对象和没有副作用的方法。

首先想到它们。只有在特定需要和并加以权衡之后才选择 var，可变对象和有副作用的方法。

54

3.6 第十二步 从文件里读取文本行

日常使用的脚本经常需要处理文件。本节中，你将完成从文件中读取文本行，并打印输出每行字符个数和文本的脚本。清单3.10为第一个版本的实现：

```
import scala.io.Source
if (args.length >0) {
  for (line <- Source.fromFile(args(0)).getLines)
    print(line.length + " " + line)
}
else
  Console.err.println("Please enter filename")
```

清单3.10 从文件中读入行

脚本首先引用包 scala.io 的 Source 类。然后检查命令行里是否至少包含一个参数。若是，则第一个参数被解释为要打开和处理的文件名。表达式 Source.fromFile(args(0))尝试打开指定的文件并返回 Source 对象。之后对它调用 getLines 函数，返回 Iterator[String]。枚举每次提供一行文本，包括行结束符。for 表达式枚举这些行并打印每行的长度、空格和这行文本。如果命令行里没有提供参数，最后的 else 子句将在标准错误流中输出错误信息。把以上代码写入文件 contchars1.scala，并运行它调用脚本自身：

```
$ scala countchars1.scala countchars1.scala
```

可以看到：

```
23 import scala.io.Source
1
23 if (args.length > 0) {
1
50   for (line <- Source.fromFile(args(0)).getLines)
36     print(line.length + " " + line)
2 }
5 else
47   Console.err.println("Please enter filename")
```

55

尽管脚本的第一版能够完成既定需求，但还不够美观。我们希望能让数字右对齐，并加上管道符号，输出成如下的格式：

```
23 | import scala.io.Source
 1 |
23 | if (args.length > 0) {
 1 |
50 |   for (line <- Source.fromFile(args(0)).getLines)
```

```
34 |     print(line.length + " " + line)
 2 | }
 5 | else
47 |   Console.err.println("Please enter filename")
```

为了达到这个目的，我们可以对所有文本行枚举两次。第一次决定字符计数的最大宽度。第二次根据计算的最大宽度调整格式打印输出。为此，我们为枚举创建值如下：

```
val lines = Source.fromFile(args(0)).getLines.toList
```

最后的 toList 是必须加的，因为 getLines 方法返回的是枚举器。一旦完成遍历，枚举器就失效了；而通过调用 toList 把它转换为 List，我们把文件中的所有行全部贮存在内存中，因此可以随时使用。lines 变量因此就指向着包含了指定文件的文本字符串列表。

接下来，因为每行文本要计算两次，每次枚举各计数一次，所以可以把字符串的长度计算功能提炼成函数：

```
def widthOfLength(s: String) = s.length.toString.length
```

这样就可以计算字符串的最大长度：

```
var maxWidth = 0
for (line <- lines)
  maxWidth = maxWidth.max(widthOfLength(line))
```

首先，定义了初始值为 0 的整数变量 maxWidth。然后，使用 for 表达式每次枚举一行文本，调用 max 方法（任何 Int 上都可以调用，返回被调用者和传入参数中较大的值）计算行的宽度，并把最大值赋给 maxWidth。如果不想用 var 来实现，也可以首先找到最长的一行，如：

```
val longestLine = lines.reduceLeft(
  (a, b) => if (a.length > b.length) a else b
)
```

reduceLeft 方法把传入的方法参数应用于 lines 的第一、二个元素，产生结果，然后再应用于这个结果和 lines 接下去的一个元素，以此类推直至整个列表。每次应用产生的结果都是已经调用过的参数中最长的一行，因为传入的函数(a, b) => if (a.length > b.length) a else b，返回两个传入字符串中长的那个。reduceLeft 将返回最后一次应用的结果，也就是本例 lines 中包含的最长字符串。

之后，可以把这行字符串传给 widthOfLength 计算最大宽度：

```
val maxWidth = widthOfLength(longestLine)
```

剩下的就是用合适的格式打印输出。如下：

```
for (line <- lines) {
  val numSpaces = maxWidth - widthOfLength(line)
  val padding = " " * numSpaces
  print(padding + line.length + " | " + line)
}
```

for 表达式再次枚举了全部文本行。对于每一行，首先计算行长度前所需的空格并把它赋给

numSpaces。然后用表达式：" " * numSpaces 创建包含 numSpaces 个空格的字符串。最终，打印出格式化的信息。全部的脚本在清单 3.11 中：

```
import scala.io.Source

def widthOfLength(s: String) = s.length.toString.length

if (args.length > 0) {

  val lines = Source.fromFile(args(0)).getLines.toList

  val longestLine = lines.reduceLeft(
     (a, b) => if (a.length > b.length) a else b
  )
  val maxWidth = widthOfLength(longestLine)

  for (line <- lines) {
     val numSpaces = maxWidth widthOfLength(line)
     val padding = " " * numSpaces
     print(padding + line.length +" | "+ line)
  }
}
else
  Console.err.println("Please enter filename")
```

清单 3.11 对文件的每行记录打印格式化的字符数量

3.7 小结

学习了本章讲解的知识之后，你应该已经具备了使用 Scala 编写简单任务（尤其是脚本）的能力。下一章，我们将对已经学习过的内容做更深入的讨论，并介绍其他尚未涉及的内容。

第 4 章

类和对象

前两章介绍了 Scala 中类和对象的基础知识。本章我们将深入这些话题，学习更多类、字段和方法的相关知识，并初步了解什么是分号推断。我们还将学习单例对象的定义及用法。熟悉 Java 的人会发现 Scala 里的某些概念虽然相似，但不完全相同。所以即使是 Java 高手，读一下本章也是有益的。

4.1 类、字段和方法

类是对象的蓝图。一旦你定义了类，就可以用关键字 new 根据类的蓝图创建对象。比方说，有如下的类定义：

```
class ChecksumAccumulator {
  // 此处为类定义
}
```

就能创建 ChecksumAccumulator 对象：

```
new CheckSumAccumulator
```

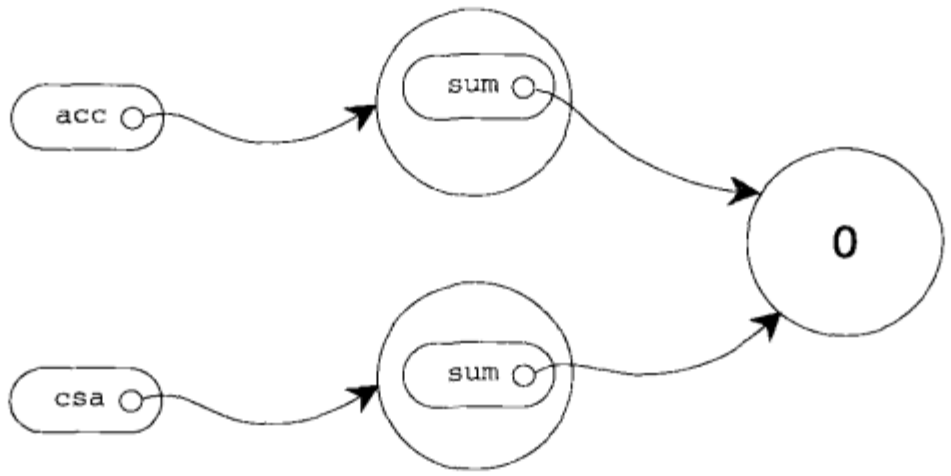
类定义里，可以放置字段和方法，这些被笼统地称为**成员**（member）。字段，不管是用 val 还是用 var 定义的，都是指向对象的变量。方法，用 def 定义，包含了可执行的代码。字段保留了对象的状态或数据，而方法使用这些数据执行对象的运算工作。当类被实例化的时候，运行时环境会预留一些内存来保留对象的状态映像——即变量的内容。例如，定义了 ChecksumAccumulator 类并内置名为 sum 的 var 字段：

```
class ChecksumAccumulator {
  var sum = 0
}
```

然后实例化两次：

```
val acc = new ChecksumAccumulator
val csa = new ChecksumAccumulator
```

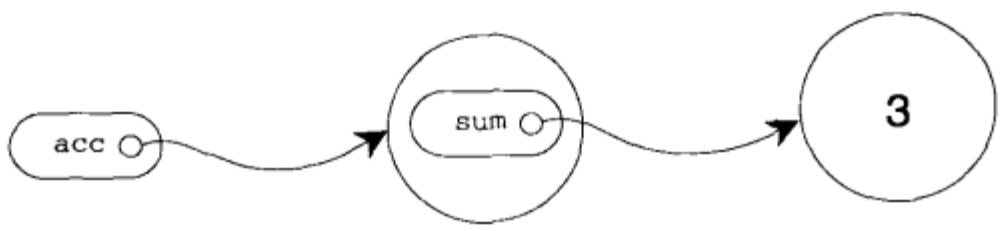
那么内存里对象的状态映像看上去大概是这样：

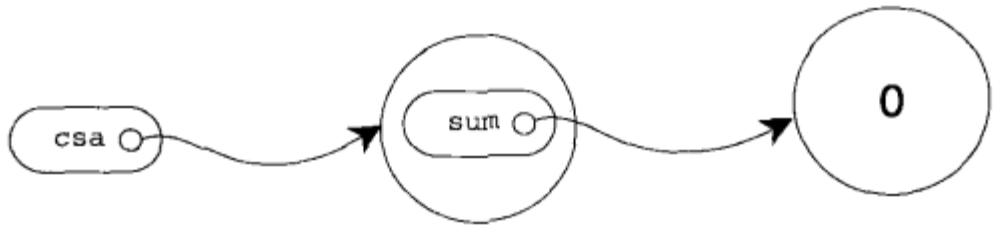


由于在类 ChecksumAccumulator 里面定义的字段 sum 是 var，而不是 val，因此之后可以重新赋给不同的 Int 值，如：

```
acc.sum = 3
```

现在，映像会变成：





注意图中同时存在两个 sum 变量，一个在 acc 指向的对象里，另一个在 csa 指向的对象里。字段的另一种说法是实例变量（instance variable），因为每一个实例都有自己的变量集。总体来说，对象实例的变量组成了对象的内存映像。我们不只可以从图上发现这点，同样也可以通过改变其中一个时，另一个不变来证实。

另外还需注意的是，尽管 acc 是 val，但仍可以修改 acc 指向的对象。val 类型对象对 acc（或 csa）的限制仅在于不可以把它们再次赋值为其他对象。例如，下面的尝试将会失败：

```
// 编译不过，因为 acc 是 val
acc = new ChecksumAccumulator
```

因此，我们可以得出结论，acc 将始终指向初始化时的 ChecksumAccumulator 对象，但是对象包含的字段可以随时改动。

保持对象健壮性的重要方法之一就是保证对象的状态——即实例变量的值——在对象整个生命周期中持续有效。第一步就是通过把字段变为私有的（private）以阻止外界直接对它的访问，因为私有字段只能被定义成在同一个类里的方法访问，所有能更新字段的代码将被锁定在类里。要声明字段是私有

的，可以把访问修饰符 private 放在字段的前面，就像这样：

```
class ChecksumAccumulator {
  private var sum = 0
}
```

有了这个 ChecksumAccumulator 的定义，任何从类外部访问 sum 的尝试都会失败：

```
val acc = new ChecksumAccumulator
acc.sum = 5//编译不过，因为 sum 是私有的
```

> **注意**
>
> 在 Scala 里把成员公开的方法是不显式地指定任何访问修饰符。换句话说，在 Java 里要写上“public”的地方，在 Scala 里只要什么都不要写就成。Public 是 Scala 的默认访问级别。

现在 sum 是私有的，所以唯一能访问 sum 的代码都定义在类的内部。这样，除非我们定义什么方法，否则 ChecksumAccumulator 对任何人都没什么用处：

```
class ChecksumAccumulator {

  private var sum = 0

  def add(b: Byte): Unit = {
    sum += b
  }
  def checksum(): Int = {
    return ~(sum &0xFF) + 1
  }
}
```

现在 ChecksumAccumulator 有两个方法了，add 和 checksum，两个都以基本的函数定义方式展示，参见图 2.1。

传递给方法的任何参数都可以在方法内部使用。Scala 里方法参数的一个重要特征是它们都是 val，不是 var[1]。如果你想在方法里面给参数重新赋值，结果是编译失败：

```
def add(b: Byte): Unit = {
  b = 1  // 编译不过，因为 b 是 val
  sum += b
}
```

尽管 ChecksumAccumulator 在这个版本里的 add 和 checksum 方法正确地实现了预期的功能，但还不够简洁。首先，checksum 方法最后的 return 语句是多余的可以去掉。如果没有发现任何显式的返回语句，Scala 方法将返回方法中最后一次计算得到的值。

方法的推荐风格是尽量避免使用返回语句，尤其是多条返回语句。代之以把每个方法当作是创建返回值的表达式。这种逻辑鼓励你分层简化方法，把较大的方法分解为多个更小的方法。另一方面，内容决定形式，如果确实需要，Scala 也能够很容易地编写具有多个显式的 return 的方法。

在这里，checksum 只要计算值，不用 return。所以这个方法有另一种简写方式：假如某个方法

[1]注：参数是 val 的理由是这样更易于理清头绪。你不需要分析更多的代码以确定是否 val 被重新赋值，而 var 则不然。

仅计算单个结果表达式，则可以去掉花括号。如果结果表达式很短，甚至可以把它放在 def 的同一行里。这样改动之后，类 ChecksumAccumulator 看上去像这样：

```
class ChecksumAccumulator {
  private var sum = 0
  def add(b: Byte): Unit = sum += b
  def checksum(): Int = ~(sum & 0xFF) + 1
}
```

对于像 ChecksumAccumulator 的 add 方法那样的结果类型为 Unit 的方法来说，执行的目的就是为了它的副作用。通常我们定义副作用为能够改变方法之外的某处状态或执行 I/O 活动的方法。比方说，在 add 这个例子里，副作用就是 sum 被重新赋值了。它的另一种表达方式是去掉结果类型和等号，把方法体放在花括号里。在这种形式下，方法看上去很像过程（procedure），一种仅为了副作用而执行的方法。清单 4.1 的 add 方法可以改写如下：

```
// 文件 ChecksumAccumulator.scala
class ChecksumAccumulator {
  private var sum = 0
  def add(b: Byte) { sum += b }
  def checksum(): Int = ~(sum &0xFF) + 1
}
```

清单 4.1 类 ChecksumAccumulator 的最终版

比较容易出错的地方是如果去掉方法体前面的等号，那么方法的结果类型就必定是 Unit。这种说法不论方法体里面包含什么都成立，因为 Scala 编译器可以把任何类型转换为 Unit。例如，如果方法的最后结果是 String，但结果类型被声明为 Unit，那么 String 将被转变为 Unit 并丢弃原值：

```
scala> def f(): Unit = "this String gets lost"
f: ()Unit
```

例子里，函数 f 声明了结果类型 Unit，因此 String 被转变为 Unit。Scala 编译器会把定义的像过程的方法，就是说，带有花括号但没有等号的，本质上当作 Unit 结果类型的方法。例如：

```
scala> def g() { "this String gets lost too" }
g: ()Unit
```

因此，对于本想返回非 Unit 值的方法却忘记加等号时，错误就出现了。所以为了得到想要的结果，等号是必不可少的：

```
scala> def h() = { "this String gets returned!" }
h: ()java.lang.String

scala> h
res0: java.lang.String = this String gets returned!
```

4.2 分号推断

Scala 程序里，语句末尾的分号通常是可选的。愿意可以加，若一行里仅有一个语句也可以不加。不过，如果一行包含多条语句时，分号则是必须的：

```
val s = "hello"; println(s)
```

输入跨越多行的语句时，多数情况无须特别处理，Scala 将在正确的位置分隔语句。例如，下面的代码被当成跨四行的一条语句：

```
if (x <2)
  println("too small")
else
  println("ok")
```

然而，偶尔 Scala 也许并不如你所愿，把句子分拆成两部分：

```
x
+ y
```

这会被当做两个语句 x 和 +y。如果希望把它作为一个语句 x + y，可以把它放在括号里：

```
(x
+ y)
```

或者也可以把 + 放在行末。也正源于此，串接类似于+这样的中缀操作符的时候，Scala 通常的风格是把操作符放在行尾而不是行头：

```
x +
y +
z
```

分号推断的规则

分割语句的具体规则既出人意料地简单又非常有效。那就是，除非以下情况的一种成立，否则行尾被认为是一个分号：

1. 疑问行由一个不能合法作为语句结尾的字结束，如句点或中缀操作符。
2. 下一行开始于不能作为语句开始的词。
3. 行结束于括号(...)或方框[...]内部，因为这些符号不可能容纳多个语句。

4.3 Singleton 对象

第 1 章提到过，Scala 比 Java 更为面向对象的特点之一是 Scala 不能定义静态成员，而是代之以定义单例对象（singleton object）。除了用 object 关键字替换了 class 关键字以外，单例对象的定义看上去与类定义一致。参见清单 4.2：

```
// 文件 ChecksumAccumulator.scala
import scala.collection.mutable.Map

object ChecksumAccumulator {

  private val cache = Map[String, Int]()

  def calculate(s: String): Int =
    if (cache.contains(s))
```

```
      cache(s)
    else {
      val acc = new ChecksumAccumulator
      for (c <- s)
        acc.add(c.toByte)
      val cs = acc.checksum()
      cache += (s -> cs)
      cs
    }
  }
```

清单 4.2 类 ChecksumAccumulator 的伴生对象

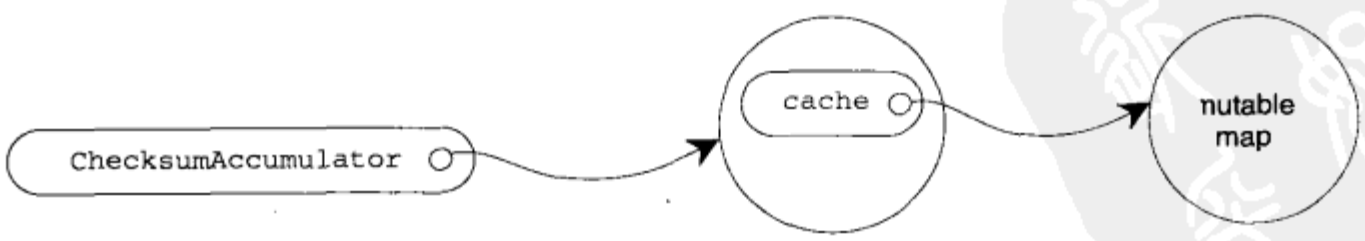
表中的单例对象叫做 ChecksumAccumulator，与前一个例子里的类同名。当单例对象与某个类共享同一个名称时，它就被称为是这个类的伴生对象（companion object）。类和它的伴生对象必须定义在一个源文件里。类被称为是这个单例对象的伴生类（companion class）。类和它的伴生对象可以互相访问其私有成员。

ChecksumAccumulator 单例对象包含了方法 calculate，用来计算方法参数 String 的字符校验和；以及私有字段 cache，缓存之前计算过的校验和的可变映射[2]。方法的第一行，“if (cache.contains(s))”，检查缓存，看看是否字符串参数已经作为键存在于映射当中。如果是，就返回映射的值“cache(s)”；否则，执行 else 子句计算校验和。else 子句的第一行定义了一个叫 acc 的 val 并初始化为新建的 ChecksumAccumulator 实例[3]。下一行是个 for 表达式，枚举传入字符串的字符，并依次调用 toByte 把字符转换成 Byte，然后传递给 acc 所指的 ChecksumAccumulator 实例的 add 方法。完成了 for 表达式后，下一行的方法对 acc 调用 checksum，获得传入字符串的校验和，存入名为 cs 的 val。下一行，“cache += (s -> cs)”，建立传入字符串到整数校验和的键值对映射，然后加入 cache 映射表。方法的最后一个表达式“cs”，确定校验和为方法结果。

对于 Java 程序员来说，可以把单例对象当作是 Java 中可能会用到的静态方法工具类。也可以用类似的语法做方法调用：单例对象名，点，方法名。例如，可以调用 ChecksumAccumulator 单例对象的 calculate 方法如下：

```
ChecksumAccumulator.calculate("Every value is an object.")
```

然而单例对象不只是静态方法的工具类。它同样是头等的对象。因此单例对象的名字可以被看作是贴在对象上的“名签”：



[2]注：这里我们使用了缓存例子来展示带有字段的单例对象。像这样的缓存是通过内存换计算时间的方式做到性能的优化。通常意义上说，只有遇到了缓存能解决的性能问题时，才可能用到这样的例子，而且应该使用弱映射（weak map），如 scala.Collection.jcl 的 WeakHashMap，这样如果内存稀缺的话，缓存里的条目就会被垃圾回收机制回收掉。

[3]注：因为关键字 new 只用来实例化类，所以这里创造的新对象是 ChecksumAccumulator 类的一个实例，而不是同名的单例对象。

定义单例对象并没有定义类型（在 Scala 的抽象层次上说）。如果只有 ChecksumAccumulator 对象的定义，就不能建立 ChecksumAccumulator 类型的变量。或者可以认为，ChecksumAccumulator 类型是由单例对象的伴生类定义的。然而，单例对象扩展了父类并可以混入特质。因此，可以使用类型调用单例对象的方法，或者用类型的实例变量指代单例对象，并把它传递给需要类型参数的方法。我们将在第 12 章展示一些继承自类和特质的单例对象的例子。

类和单例对象间的差别是，单例对象不带参数，而类可以。因为单例对象不是用 new 关键字实例化的，所以没机会传递给它实例化参数。每个单例对象都被实现为虚构类（synthetic class）的实例，并指向静态的变量，因此它们与 Java 静态类有着相同的初始化语义[4]。特别要指出的是，单例对象在第一次被访问的时候才会被初始化。

不与伴生类共享名称的单例对象被称为独立对象（standalone object）。它可以用在很多地方，例如作为相关功能方法的工具类，或者定义 Scala 应用的入口点。下面会说明这种用法。

4.4 Scala 程序

想要编写能够独立运行的 Scala 程序，就必须创建有 main 方法（仅带一个参数 Array[String]，且结果类型为 Unit）的单例对象。任何拥有合适签名的 main 方法的单例对象都可以用来作为程序的入口点。示例参见清单 4.3：

```
// 文件 Summer.scala
import ChecksumAccumulator.calculate
object Summer {
  def main(args: Array[String]) {
    for (arg <- args)
      println(arg + ": " + calculate(arg))
  }
}
```

清单 4.3 程序 Summer

这里单例对象的名字是 Summer。它的 main 方法具有合适的签名，所以可以用作程序。文件中的第一个语句是对定义在前面 ChecksumAccumulator 对象中 calculate 方法的引用。它允许你在后面的文件里使用方法的简化名[5]。main 方法体打印输出每个参数和参数的校验和，用冒号分隔。

> **注意**
>
> Scala 的每个源文件都隐含了对包 java.lang、包 scala，以及单例对象 Predef 的成员引用。包 scala 中的 Predef 对象包含了许多有用的方法。例如，Scala 源文件中写下 println 语句，实际调用的是 Predef 的 println（Predef.println 转而调用 Console.println，完成真正的工作）。写下 assert，实际是在调用 Predef.assert。

[4]注：虚构类的名字是对象名加上一个美元符号。因此单例对象 ChecksumAccumulator 的虚构类是 ChecksumAccumulator$。

[5]注：对于 Java 程序员来说，可以认为这种引用类似于 Java 5 引入的静态引用特性。然而 Scala 里的不同点在于，任何成员引用可以来自于任何对象，而不只是单例对象。

要执行 Summer 应用程序，需要把以上的代码写入文件 Summer.scala 中。因为 Summer 使用了 ChecksumAccumulator，所以还要把 ChecksumAccumulator 的代码，包括清单 4.1 的类和清单 4.2 里它的伴生对象，放在文件 ChecksumAccumulator.scala 中。

Scala 和 Java 之间有一点不同，Java 需要你把公共类放在以这个类命名的源文件中——如类 SpeedRacer 要放在文件 SpeedRacer.java 里——Scala 对于源文件的命名没有硬性规定。然而通常情况下如果不是脚本，推荐的风格是像在 Java 里那样按照所包含的类名来命名文件，这样程序员就可以比较容易地根据文件名找到类。本例中我们对文件 ChecksumAccumulator.scala 和 Summer.scala 使用这一原则命名。

ChecksumAccumulator.scala 和 Summer.scala 都不是脚本，因为它们都以定义结尾。反过来说，脚本必须以结果表达式结束。因此如果你尝试以脚本方式执行 Summer.scala，Scala 解释器将会报错说 Summer.scala 不是以结果表达式结束的（当然前提是你没有在 Summer 对象定义之后加上任何你自己的表达式）。正确的做法是，你需要用 Scala 编译器真正地编译这些文件，然后执行输出的类文件。方式之一是使用 Scala 的基本编译器，scalac。输入：

```
$ scalac ChecksumAccumulator.scala Summer.scala
```

开始编译源文件，不过在编译完成之前或许会稍微停顿一下。这是因为每次编译器启动时，都要花一些时间扫描 jar 文件内容，并在开始编译你提交的源文件之前完成更多其他的初始化工作。因此，Scala 的发布包里还包括了一个叫做 fsc（快速 Scala 编译器，fast Scala compiler）的 Scala 编译器后台服务（daemon）。使用方法如下：

```
$ fsc ChecksumAccumulator.scala Summer.scala
```

第一次执行 fsc 时，会创建一个绑定在你计算机端口上的本地服务器后台进程。然后它就会把文件列表通过端口发送给后台进程，由后台进程完成编译。下一次执行 fsc 时，检测到后台进程已经在运行了，于是 fsc 将只把文件列表发给后台进程，它会立刻开始编译文件。使用 fsc，只须在首次运行的时候等待 Java 运行时环境的启动。如果想停止 fsc 后台进程，可以执行 fsc -shutdown。

不论执行 scalac 还是 fsc 命令，最终都将完成 Java 类文件的创建，然后可以用 scala 命令，就像之前的例子里调用解释器那样运行它。不过，不是像前面的例子里那样解释执行带有 .scala 扩展名的源代码文件[6]，这里用的是包含了正确签名的 main 方法的独立对象名。因此，Summer 应用程序的运行方法如下：

```
$ scala Summer of love
```

输出结果是两个命令行参数的校验和：

```
of: -213
love: -182
```

[6]注：scala 程序用来“解释”Scala 源文件的真正机制是，它把 Scala 源码编译成字节码，然后立刻通过类装载器装载它们，并执行它们。

4.5 Application 特质

Scala 提供了特质 scala.Application，可以减少一些输入工作。尽管现在所学的知识还不足以弄懂它的工作原理，不过我们认为了解一下还是有必要的。参见清单 4.4：

```
import ChecksumAccumulator.calculate

object FallWinterSpringSummer extends Application {

  for (season <- List("fall", "winter", "spring"))
    println(season +": "+ calculate(season))
}
```

清单 4.4 使用 Application 特质

使用方法是，首先在单例对象名后面写上“extends Application”。然后代之以 main 方法，你可以把想要执行的代码直接放在单例对象的花括号之间。如此而已。之后可以正常的编译和运行。

能这么做，是因为特质 Application 声明了带有合适签名的 main 方法，并被你写的单例对象继承，使它可以像 Scala 程序那样。花括号之间的代码被收集进了单例对象的主构造器（primary constructor），并在类被初始化时执行。如果你暂时不能明白所有这些指的是什么也不用着急。之后的章节会解释，目前只要知道有这种做法即可。

继承自 Application 比编写完整的 main 方法要方便，不过它也有些缺点。首先，如果想访问命令行参数的话就不能用它，因为 args 数组不可访问。比如，因为 Summer 程序使用了命令行参数，所以它必须编写完整的 main 方法，如清单 4.3 所示；第二，因为某些 JVM 线程模型里的局限，如对于多线程的程序需要自行编写 main 方法；最后，某些 JVM 的实现没有优化被 Application 特质执行的对象的初始化代码。因此只有当程序相对简单并且是单线程的情况下才可以继承 Application 特质。

71

4.6 小结

本章介绍了 Scala 里的类和对象的基础知识，并举例说明了如何编译和运行程序。下一章，我们将学习 Scala 的基本类型及使用方式。

72

第 5 章

基本类型和操作

现在你已经看过了真正的类和对象，到了更进一步了解 Scala 的基本类型和操作的时候。如果熟悉 Java，你会很开心地发现 Java 的基本类型和操作符在 Scala 里有同样的意思。然而即使对于资深 Java 开发者来说，这里也仍然有一些值得关注的差别。对于 Scala 里实质上与 Java 相同的部分，我们插入了注释，Java 开发者看到后可以安全跳过，以加快你的进程。

本章里，你将了解到 Scala 的基本类型，包括 `String` 和值类型 `Int`、`Long`、`Short`、`Byte`、`Float`、`Double`、`Char`，还有 `Boolean`。你将学到这些类型的相关操作，以及 Scala 表达式里的操作符优先级的工作原理。你还会学到隐式转换是如何让这些基本类型的操作变得更为“丰富”的，并由此带给你那些 Java 无法提供支持的附加操作。

5.1 基本类型

表 5.1 显示了 Scala 的基本类型和实例值域范围。总体来说、类型 `Byte`、`Short`、`Int`、`Long` 和 `Char` 被称为整数类型（integral type）。整数类型加上 `Float` 和 `Double` 被称为数类型（numeric type）。

表 5.1 一些基本类型

值类型	范围
`Byte`	8 位有符号补码整数（$-2^7 \sim 2^7-1$）
`Short`	16 位有符号补码整数（$-2^{15} \sim 2^{15}-1$）
`Int`	32 位有符号补码整数（$-2^{31} \sim 2^{31}-1$）
`Long`	64 位有符号补码整数（$-2^{63} \sim 2^{63}-1$）
`Char`	16 位无符号 Unicode 字符（$0 \sim 2^{16}-1$）
`String`	char 序列
`Float`	32 位 IEEE754 单精度浮点数
`Double`	64 位 IEEE754 单精度浮点数
`Boolean`	`true` 或 `false`

除了 `String` 归于 `java.lang` 包之外，其余所有的基本类型都是包 `scala` 的成员[1]。如，`Int` 的全名是 `scala.Int`。然而，由于包 `scala` 和 `java.lang` 的所有成员都被每个 Scala 源文件自动引用，因此可以在任何地方只用简化名（就是说，直接写成 `Boolean`、`Char` 或 `String`）。

[1]注：包的概念在第 2 章的第二步作了简要说明，将在第 13 章做更深入介绍。

> **注意**
>
> 目前实际上 Scala 值类型可以使用与 Java 的原始类型一致的小写化名称。比如，Scala 程序里可以用 `int` 替代 `Int`。但请记住它们都是一回事：`scala.Int`。Scala 社区实践提出的推荐风格是一直使用大写形式，这也是我们在这本书里做的。为了纪念这个社区推动的选择，将来 Scala 的版本可能不再支持乃至移除小写化名称，因此跟随社区的趋势，在 Scala 代码中使用 `Int` 而非 `int` 才是明智之举。

敏锐的 Java 开发者会注意到 Scala 的基本类型与 Java 的对应类型范围完全一样。这样可以让 Scala 编译器直接把 Scala 的值类型（value type）实例，如 `Int` 或 `Double`，对应转译为 Java 原始类型。

5.2 字面量

74

所有在表 5.1 里列出的基本类型都可以写成字面量（literal）。字面量就是直接写在代码里的常量值。

> **Java 程序员的快速通道**
>
> 本节里多数字面量的语法和在 Java 里完全一致，因此对于 Java 达人来说，可以安心地跳过本节的多数内容。唯一需要关注的两个差异分别是 Scala 的原字符串和符号字面量，将在原书第 59 页描述。

整数字面量

类型 `Int`、`Long`、`Short` 和 `Byte` 的整数字面量有三种格式：十进制、十六进制和八进制。整数字面量的开头方式说明了数值的进制。如果数开始于 `0x` 或 `0X`，那它是十六进制（基于 16），并且可能包含从 `0` 到 `9`，及大写或小写的从 `A` 到 `F` 的数字。举例如下：

```
scala> val hex = 0x5
hex: Int = 5

scala> val hex2 = 0x00FF
hex2: Int = 255

scala> val magic = 0xcafebabe
magic: Int = -889275714
```

请注意，不论用什么形式的整数字面量初始化，Scala 的 shell 始终打印输出基于 10 的整数值。因此解释器会把用字面量 `0x00FF` 初始化的 `hex2` 变量的值显示为十进制的 255（不要只是一味地对我们的话深信不疑。学习语言的方式是一边阅读一边尝试在解释器里执行这些代码）。如果数开始于零，就是八进制（基于 8）的，并且只可以包含数字 0 到 7。下面是一些例子：

```
scala> val oct = 035 // （八进制 35 是十进制 29）
oct: Int = 29

scala> val nov = 0777
nov: Int = 511

scala> val dec = 0321
dec: Int = 209
```

75

如果数开始于非零数字，并且没有被修饰过，就是十进制（基于 10）的。例如：

```
scala> val dec1 = 31
dec1: Int = 31

scala> val dec2 = 255
dec2: Int = 255

scala> val dec3 = 20
dec3: Int = 20
```

如果整数字面量结束于 L 或者 l，就是 Long 类型，否则就是 Int 类型。一些 Long 类型的整数字面量有：

```
scala> val prog = 0XCAFEBABEL
prog: Long = 3405691582

scala> val tower = 35L
tower: Long = 35

scala> val of = 31l
of: Long = 31
```

如果 Int 类型的字面量被赋值给 Short 或 Byte 类型的变量，字面量就会被当做是被赋值的类型，以便让字面量值处于有效范围内。如：

```
scala> val little: Short = 367
little: Short = 367

scala> val littler: Byte = 38
littler: Byte = 38
```

浮点数字面量

浮点数字面量是由十进制数字、可选的小数点、可选的 E 或 e 及指数部分组成的。下面是一些浮点数字面量的例子：

```
scala> val big = 1.2345
big: Double = 1.2345

scala> val bigger = 1.2345e1
bigger: Double = 12.345

scala> val biggerStill = 123E45
biggerStill: Double = 1.23E47
```

请注意指数部分表示的是乘上以 10 为底的幂级数。因此，1.2345e1 就是 1.2345 乘以 10^1，等于 12.345。如果浮点数字面量以 F 或 f 结束，就是 Float 类型的，否则就是 Double 类型的。可选的，Double 浮点数字面量也可以 D 或 d 结尾。Float 字面量举例如下：

```
scala> val little = 1.2345F
little: Float = 1.2345

scala> val littleBigger = 3e5f
littleBigger: Float = 300000.0
```

最后一个值可以用以下（或其他）格式表示为 Double 类型：

```
scala> val anotherDouble = 3e5
anotherDouble: Double = 300000.0

scala> val yetAnother = 3e5D
yetAnother: Double = 300000.0
```

字符字面量

字符字面量可以是在单引号之间的任何 Unicode 字符，如：

```
scala> val a = 'A'
a: Char = A
```

单引号之间除了可以摆放字符之外，还可以提供一个前缀反斜杠的八进制或十六进制的表示字符编码号的数字。八进制数必须在'\0'和'\377'之间。例如字母 A 的 Unicode 字符编码是八进制 101。因此：

```
scala> val c = '\101'
c: Char = A
```

字符字面量同样可以以前缀\u 的四位十六进制数字的通用 Unicode 字符方式给出，如：

```
scala> val d = '\u0041'
d: Char = A
scala> val f = '\u0044'
f: Char = D
```

实际上，这种 unicode 字符可以出现在 Scala 程序的任何地方。例如你可以这样写一个标识符：

```
scala> val B\u0041\u0044 = 1
BAD: Int = 1
```

这个标识符被当作 BAD，这是上面代码里的两个 unicode 字符转义（expand）之后的结果。通常，这样命名标识符是个坏主意，因为它太难读。然而，这种语法能够允许含有非 ASCII 的 Unicode 字符的 Scala 源文件用 ASCII 来代表。

最后，还有一些字符字面量被表示成特殊的转义序列，参见表 5.2。例如：

表 5.2 特殊字符字面量转义序列

字面量	含义
\n	换行（\u000A）
\b	回退（\u0008）
\t	制表符（\u0009）
\f	换页（\u000C）
\r	回车（\u000D）
\"	双引号（\u0022）
\'	单引号（\u0027）
\\	反斜杠（\u005C）

```
scala> val backslash = '\\'
backslash: Char = \
```

字符串字面量

字符串字面量是由双引号（"）包括的字符组成：

```
scala> val hello = "hello"
hello: java.lang.String = hello
```

引号内的字符语法与字符字面量相同，如：

```
scala> val escapes = "\\\"\'"
escapes: java.lang.String = \"'
```

由于这种语法对于包含大量转义序列或跨越若干行的字符串很笨拙。因此 Scala 为原始字符串（raw string）引入了一种特殊的语法。它以同一行里的三个引号（"""）作为开始和结束。内部的原始字符串可以包含无论何种任意字符，包括新行、引号和特殊字符，当然同一行的三个引号除外。举例来说，下面的程序使用原始字符串打印输出了一条消息：

```
println("""Welcome to Ultamix 3000.
           Type "HELP" for help.""")
```

运行这段代码不会产生完全符合所需的东西，而是：

```
Welcome to Ultamix 3000.
           Type "HELP" for help.
```

原因是第二行前导的空格被包含在了字符串里。为了解决这个常见情况，字符串类引入了 stripMargin 方法。使用的方式是，把管道符号（|）放在每行前面，然后对整个字符串调用 stripMargin：

```
println("""|Welcome to Ultamix 3000.
           |Type "HELP" for help.""".stripMargin)
```

这样，输出结果就令人满意了：

```
Welcome to Ultamix 3000.
Type "HELP" for help.
```

符号字面量

符号字面量被写成'<标识符>，这里<标识符>可以是任何字母或数字的标识符。这种字面量被映射成预定义类 scala.Symbol 的实例。具体地说，就是字面量'cymbal 将被编译器扩展为工厂方法调用：Symbol("cymbal")。符号字面量典型的应用场景是在动态类型语言中使用一个标识符。比方说，定义更新数据库记录的方法：

```
scala> def updateRecordByName(r: Symbol, value: Any) {
          // 此处为代码
       }
updateRecordByName: (Symbol,Any)Unit
```

方法有两个参数，符号参数表示记录的字段名，Any 类型的参数表示应该更新至记录的值。动态类型语言可以通过向方法传入未声明字段标识符调用这个操作，但在 Scala 里这样会编译不过：

```
scala> updateRecordByName(favoriteAlbum, "OK Computer")
<console>:6: error: not found: value favoriteAlbum
          updateRecordByName(favoriteAlbum, "OK Computer")
```

替代方案也同样简洁，你可以传递一个符号字面量：

```
scala> updateRecordByName('favoriteAlbum, "OK Computer")
```

符号字面量除了显示名字之外，什么都不能做：

```
scala> val s = 'aSymbol
s: Symbol = 'aSymbol

scala> s.name
res20: String = aSymbol
```

还有就是符号是被限定（interned）的。如果同一个符号字面量出现两次，那么两个字面量指向的是同一个 Symbol 对象。

80

布尔型字面量

布尔类型有两个字面量，true 和 false：

```
scala> val bool = true
bool: Boolean = true

scala> val fool = false
fool: Boolean = false
```

字面量的知识就这些。现在你简直（literally）[2]可以称得上是 Scala 的专家了。

5.3 操作符和方法

Scala 为基本类型提供了丰富的操作符集。如前几章里描述的，这些操作符实际只是普通方法调用的另一种表现形式。例如，1 + 2 与(1).+(2)其实是一回事。换句话说，就是 Int 类包含了叫做+的方法，它传入一个 Int 参数并返回一个 Int 结果，在两个 Int 值相加时被调用：

```
scala> val sum = 1 + 2 // Scala 调用了(1).+(2)
sum: Int = 3
```

为了证实这点，可以把表达式写成显式的方法调用：

```
scala> val sumMore = (1).+(2)
sumMore: Int = 3
```

实际上，Int 包含了+的各种类型参数的重载（overload）方法[3]。例如，另一个也叫+的方法的参数和返回类型都是 Long。如果把 Long 加到 Int 上，就将转而调用这个+方法：

```
scala> val longSum = 1 + 2L // Scala 调用了(1).+(2L)
longSum: Long = 3
```

81

符号+是操作符——更明确地说，是中缀操作符。操作符标注不仅限于像+这种其他语言里看上去

[2]注：象征意义的说法。（译注：双关语）

[3]注：重载的方法有同样的名称和不同的参数类型。第6章会对方法重载做更多说明。

像操作符的东西，任何方法都可以被当作操作符来标注。例如，类 String 有个带 Char 参数的方法 indexOf。它搜索 String 里第一次出现的指定字符，并返回它的索引；如果没有找到就返回-1。indexOf 可以被当作中缀操作符使用，就像这样：

```
scala> val s = "Hello, world!"
s: java.lang.String = Hello, world!

scala> s indexOf 'o'      // Scala 调用了 s.indexOf('o')
res0: Int = 4
```

另外，String 还提供了重载的 indexOf 方法，带两个参数，分别是要搜索的字符和从哪个索引开始搜索（前一个 indexOf 方法开始于索引零，也就是 String 开始的地方）。尽管这个 indexOf 方法带两个参数，你仍然可以用操作符标注的方式使用它。不过这样用的时候，这些参数必须放在括号内。例如，以下是把另一种形式的 indexOf 当作操作符使用的例子（接前例）：

```
scala> s indexOf ('o', 5) // Scala 调用了 s.indexOf('o', 5)
res1: Int = 8
```

任何方法都可以是操作符

Scala 里的操作符不是特殊的语法：任何方法都可以是操作符。到底是方法还是操作符取决于你如何使用它。如果写成 s.indexOf('o')，indexOf 就不是操作符。但如果写成 s indexOf 'o'，那么 indexOf 就是操作符，因为你以操作符标注方式使用它。

到目前为止，你已经看到了中缀（infix）操作符标注的例子，也就是说调用的方法位于对象和传递给方法的参数或若干参数之间，如“7 + 2”。Scala 还有另外两种操作符标注方式，分别是前缀标注和后缀标注。前缀标注中，方法名被放在调用的对象之前，如，-7 里的“-”。后缀标注中，方法放在对象之后，如，“7 toLong”里的“toLong”。

与中缀操作符——两个操作数，分别在操作符的左右两侧——相反，前缀和后缀操作符都是一元（unary）的：它们仅有一个操作数。前缀方式中，操作数在操作符的右边。前缀操作符的例子有-2.0、!found 和~0xFF。这些前缀操作符与中缀操作符一致，是值类型对象调用方法的简写形式。然而这种情况下，方法名在操作符字符上前缀“unary_”。例如，Scala 会把表达式-2.0 转换成方法调用“(2.0).unary_-”。这可以通过分别输入操作符和显式方法名两种方式调用方法加以证实：

```
scala> - 2.0              // Scala 调用了 (2.0).unary_-
res2: Double = - 2.0

scala> (2.0).unary_-
res3: Double = - 2.0
```

标识符中能作为前缀操作符用的只有+、-、!和~。因此，如果对类型定义了名为 unary_!的方法，就可以对值或变量用!p 这样的前缀操作符方式调用方法。但是即使定义了名为 unary_*的方法，也没办法将其用成前缀操作符了，因为*不是四种可以当作前缀操作符用的标识符之一。你可以像平常那用调用它，如 p.unary_*，但如果尝试像*p 这么调用，Scala 就会把它理解为*.p，这或许就不是你所期望的了[4]！

[4]注：然而，不是一点儿希望都没有。仍然有极微弱的机会，让你的带有*p 的程序或许能像 C++那样被编译。

后缀操作符是不用点或括号调用的不带任何参数的方法。在 Scala 里，方法调用的空括号可以省略。惯例是如果方法带有副作用就加上括号，如 println()；如果没有副作用就去掉括号，如 String 的 toLowerCase 方法：

```
scala> val s = "Hello, world!"
s: java.lang.String = Hello, world!

scala> s.toLowerCase
res4: java.lang.String = hello, world!
```

例子里的方法没带参数，因此还可以去掉点，采用后缀操作符标注方式：

```
scala> s toLowerCase
res5: java.lang.String = hello, world!
```

这里，toLowerCase 被当作操作数 s 的后缀操作符。

由此可知，要想知道 Scala 的值类型有哪些操作符，只要在 Scala 的 API 文档里查询定义在值类型上的方法即可。不过由于本书是 Scala 的教程，我们会在之后列举其中的大多数方法。

> **Java 程序员的快速通道**
>
> 本章后续部分描述的 Scala 的很多方面与 Java 相同。对于只想一探究竟的 Java 专家来说，可以安心地跳到 5.7 节，那里描述了在对象相等性方面 Scala 与 Java 的差异。

5.4 数学运算

任何数类型都可以使用中缀操作符，加号（+）、减号（-）、乘号（*）、除号（/）和余数（%）做数学计算。举例如下：

```
scala> 1.2 + 2.3
res6: Double = 3.5

scala> 3 - 1
res7: Int = 2

scala> 'b' - 'a'
res8: Int = 1

scala> 2L * 3L
res9: Long = 6

scala> 11 / 4
res10: Int = 2

scala> 11 % 4
res11: Int = 3

scala> 11.0f / 4.0f
res12: Float = 2.75

scala> 11.0 % 4.0
res13: Double = 3.0
```

当左右两个操作数都是整数类型时（Int、Long、Byte、Short 或 Char），/操作符将返回给你

商的整数部分，去掉余数部分。%操作符指明它的余数。

用%符号得到的浮点数余数部分并不遵循 IEEE754 标准的定义。IEEE754 在计算余数时使用四舍五入除法，而不是截尾除法，因此余数的计算与整数的余数操作会有很大的不同。如果想要得到符合 IEEE754 标准的余数，可以调用 scala.Math 里的 IEEEremainder，例如：

```
scala> Math.IEEEremainder(11.0, 4.0)
res14: Double = -1.0
```

数类型还提供了一元的前缀+和-操作符（方法 unary_+和 unary_-），以便指示数值字面量是正的还是负的，如-3 或+4.0。没有指定的情况下，数值字面量被解释为正值。+的存在只是为了与-相协调，没有任何效果。-还可以用来使变量变成负值。举例如下：

```
scala> val neg = 1 + -3
neg: Int = -2

scala> val y = +3
y: Int = 3

scala> -neg
res15: Int = 2
```

5.5 关系和逻辑操作

关系方法包括了大于（>）、小于（<）、大于等于（>=）和小于等于（<=），可以用来比较数类型并产生 Boolean 结果。还可以使用一元操作符!（unary_!方法）改变 Boolean 值。举例如下：

```
scala> 1 > 2
res16: Boolean = false

scala> 1 < 2
res17: Boolean = true

scala> 1.0 <= 1.0
res18: Boolean = true

scala> 3.5f >= 3.6f
res19: Boolean = false

scala> 'a' >= 'A'
res20: Boolean = true

scala> val thisIsBoring = !true
thisIsBoring: Boolean = false

scala> !thisIsBoring
res21: Boolean = true
```

逻辑方法、逻辑与（&&）和逻辑或（||），以中缀方式调用 Boolean 操作数并产生 Boolean 结果。如：

```
scala> val toBe = true
toBe: Boolean = true

scala> val question = toBe || !toBe
```

```
question: Boolean = true

scala> val paradox = toBe && !toBe
paradox: Boolean = false
```

与 Java 里一样，逻辑与和逻辑或有短路（short-circuit）的概念：用这些操作符建造的表达式仅评估能够决定结果的最少部分。换句话说，在逻辑与和逻辑或表达式的左侧部分能决定结果时右侧部分就不再被评估。举个例子，如果逻辑与表达式的左侧计算结果为 false，那么表达式的结果将注定是 false，因此右侧部分不再评估。与之类似，如果逻辑或表达式的左侧部分计算结果为 true，那么表达式的结果将必然是 true，于是右侧部分不再计算。下面是一些例子：

```
scala> def salt() = { println("salt"); false }
salt: ()Boolean

scala> def pepper() = { println("pepper"); true }
pepper: ()Boolean

scala> pepper() && salt()
pepper
salt
res22: Boolean = false

scala> salt() && pepper()
salt
res23: Boolean = false
```

第一个表达式中，pepper 和 salt 都被调用，但第二个里，只有 salt 被调用。因为 salt 返回 false，所以就没必要调用 pepper 了。

注意

或许你会想知道如果操作符都只是方法的话短路机制是怎么工作的呢。通常，进入方法之前会评估所有的参数，因此方法怎么可能选择不评估它的第二个参数呢？答案是因为所有的 Scala 方法都有延迟其参数评估乃至取消评估的机制，被称为传名参数（by-name parameter），将在 9.5 节中讨论。

5.6 位操作符

Scala 包含了可以对整数类型的单个位执行操作的若干位操作方法，有按位与运算（&）、按位或运算（|）和按位异或运算（^）[5]。一元按位取补操作符（~，方法 unary_~），反转它的操作数的每一位。例如：

```
scala> 1 & 2
res24: Int = 0

scala> 1 | 2
res25: Int = 3

scala> 1 ^ 3
res26: Int = 2

scala> ~1
res27: Int = -2
```

[5]注：按位异或方法对操作数执行互斥或（exclusive or）操作。相同位产生 0。差异位产生 1。因此 0011 ^ 0101 产生 0110。

第一个表达式，1 & 2，对 1（0001）和 2（0010）的每一位进行与运算，产生 0（0000）；第二个表达式，1 | 2，对同样的操作数的每一位执行或运算，产生 3（0011）；第三个表达式，1 ^ 3，异或 1（0001）和 3（0011）的每一位，产生 2（0010）。最后的表达式，~1，翻转 1（0001）的所有位，产生了-2，二进制看起来是 1111 1111 1111 1111 1111 1111 1111 1111。

Scala 整数类型还提供了三个位移方法，分别是左移（<<），右移（>>）和无符号右移（>>>）。以中缀操作符方式使用时，位移方法会按照右侧指定的整数值次数逐位移动左侧的整数。左移和无符号右移在移动的时候填入零。右移则在移动时填入左侧整数的最高位（符号位）。举例如下：

```
scala> -1 >> 31
res28: Int = -1

scala> -1 >>> 31
res29: Int = 1

scala> 1 << 2
res30: Int = 4
```

二进制的-1 是 1111 1111 1111 1111 1111 1111 1111 1111。第一个例子里，-1 >> 31，-1 被右移了 31 个位。由于 Int 包括 32 位，这个操作实际就是把最左侧的一位移到了最右侧[6]。由于>>方法在不断右移的时候填入的是 1，-1 最左侧的一位是 1，导致结果与原来左侧的数字一模一样，32 位个 1，或者说是-1。第二个例子里，-1 >>> 31，最左侧的位再一次不断向右移直至最右侧的位置，但是这次填入的是 0。因此这次的结果是二进制的 0000 0000 0000 0000 0000 0000 0000 0001，或者说是 1。最后一个例子里，1 << 2，左操作数 1，被向左移动 2 个位置（填入 0），产生结果是二进制的 0000 0000 0000 0000 0000 0000 0000 0100，或者说是 4。

5.7 对象相等性

如果想要比较一下看看两个对象是否相等，可以使用==，或它的反义!=。下面举几个例子：

```
scala> 1 == 2
res31: Boolean = false

scala> 1 != 2
res32: Boolean = true

scala> 2 == 2
res33: Boolean = true
```

这些操作对所有对象都起作用，而不仅仅是基本类型。例如可以用它来比较列表：

```
scala> List(1, 2, 3) == List(1, 2, 3)
res34: Boolean = true

scala> List(1, 2, 3) == List(4, 5, 6)
res35: Boolean = false
```

进一步，还可以用来比较不同类型的两个对象：

[6]注：数字类型的最左侧位是符号位。如果最左侧位是 1，数字就是负的，如果是 0，数字就是正的。

```
scala> 1 == 1.0
res36: Boolean = true

scala> List(1, 2, 3) == "hello"
res37: Boolean = false
```

甚至可以比较null或任何可能是null的东西。不会有任何异常被抛出：

```
scala> List(1, 2, 3) == null
res38: Boolean = false

scala> null == List(1, 2, 3)
res39: Boolean = false
```

如上所示，==已经被仔细地加工过，因此在多数情况下都可以实现合适的相等性比较。这种比较遵循一种非常简单的规则：首先检查左侧是否为null，如果不是，调用左操作数的equals方法。而精确的比较取决于左操作数的equals方法定义。由于有了自动的null检查，因此不需要手动再检查一次了[7]。

这种比较即使发生在不同的对象之间也会产生true，只要比较的两者内容相同并且equals方法是基于内容编写的。例如，以下是恰好都有五个同样字母的两个字符串的比较：

```
scala> ("he" + "llo") == "hello"
res40: Boolean = true
```

Scala的==与Java的有何差别

Java里==既可以比较原始类型也可以比较引用类型。对于原始类型，Java的==比较值的相等性，与Scala一致。而对于引用类型，Java的==比较了引用相等性（reference equality），也就是说比较的是这两个变量是否都指向JVM堆里的同一个对象。Scala也提供了这种机制，名字是eq。不过，eq和它的反义词ne，仅仅应用于可以直接映射到Java的对象。eq和ne将在11.1节和11.2节说明。还有，可以看一下第28章，了解如何编写好的equals方法。

5.8 操作符的优先级和关联性

操作符的优先级决定了表达式的哪个部分先于其他部分被评估。举例来说，表达式2 + 2 * 7计算得16，而不是28，因为*操作符比+操作符有更高的优先级。因此表达式的乘法部分先于加法部分被评估。当然你也可以在表达式里使用括号来厘清评估次序或覆盖优先级。例如，如果你实际上希望上面表达式的值是28，可以这么写：

```
(2 + 2) * 7
```

由于Scala没有操作符，实际上，操作符只是方法的一种表达方式，你或许想知道操作符优先级是怎么做到的。对于以操作符形式使用的方法，Scala根据操作符的第一个字符判断方法的优先级（这个规则有个例外，稍后再说）。比方说，如果方法名开始于*，那么就比开始于+的方法有更高的优先级。因此2+2*7将被评估为2+(2*7)，而a+++b***c（这里a，b和c是值或变量，而+++和***是方法）将被看作是a+++(b***c)，因为***方法比+++方法有更高的优先级。

[7]注：自动检查机制不会检查右侧的参数，但是任何合理的equals方法都应在参数为null的时候返回false。

表 5.3 操作符优先级

（所有其他的特殊字符）
* / %
+ -
:
= !
<>
&
^
|
（所有字母）
（所有赋值操作符）

表 5.3 以降序方式列举了以方法第一个字符判定的优先级，同一行的字符具有同样的优先级。表中字符的位置越高，以这个字符开始的方法具有的优先级就越高。举例如下：

```
scala> 2 << 2 + 2
res41: Int = 32
```

<<方法开始于字符<，在表 5.3 里的位置比+（+方法的第一个也是唯一的字符）要低。因此<<比+的优先级低，表达式也要在先调用了+方法之后再调用<<方法，如 2 << (2 + 2)。我们可以算一下，2 + 2 得 4，2 << 4 得 32。下面给出另一个例子：

```
scala> 2 + 2 << 2
res42: Int = 16
```

由于第一个字符与前面的例子里一样，因此调用的方法顺序也没有不同。首先+方法被调用，然后是<<方法。因此 2 + 2 得 4，4 << 2 得 16。

除以上提到的优先级规则外，还有以等号结束的赋值操作符（assignment operator）。如果操作符以等号字符（=）结束，且操作符并非比较操作符<=，>=，==，或=，那么这个操作符的优先级与赋值符（=）相同。也就是说，它比任何其他操作符的优先级都低。例如：

```
x *= y + 1
```

与下面的相同：

```
x *= (y + 1)
```

因为*=被当作赋值操作符，它的优先级低于+，尽管操作符的第一个字符是*，似乎其优先级高于+。

当同样优先级的多个操作符并列出现在表达式里时，操作符的关联性（associativity）决定了操作符分组的方式。Scala 里操作符的关联性取决于它的最后一个字符。正如第 3 章里提到的，任何以‘:’字符结尾的方法由它的右操作数调用，并传入左操作数。以其他字符结尾的方法与之相反。它们都是被左操作数调用，并传入右操作数的。因此 a * b 变成 a.*(b)，但是 a:::b 变成 b.:::(a)。

然而，不论操作符具有什么样的关联性，它的操作数总是从左到右评估的。因此如果 b[8]是一个表达式而不仅仅是一个不可变值的引用，那么更精确的意义上说，a:::b 将会被当作是：

92

```
{ val x = a; b.:::(x) }
```

这个代码块中，a 仍然在 b 之前被评估，然后评估结果被当作操作数传给 b 的:::方法。

这种关联性规则在同时使用多个具有同优先级的操作符时也会起作用。如果方法结束于:，它们就被自右向左分组；反过来，就是自左向右分组。例如，a ::: b ::: c 会被当作 a ::: (b ::: c)。而 a * b * c 被当作(a * b) * c。

操作符优先级也是 Scala 语言的一部分。你不用怕它。但无论如何，使用括号去厘清什么操作符作用在哪个表达式上都是好的风格。或许你唯一可以确信其他人不用查书就知道的优先级关系就是乘除法操作符（*，/和%）比加减法（+和-）的要高。因此即使 a + b << c 不用括号也能产生你想要的结果，写成(a + b) << c 而得到的简洁性也可能会降低你的同事为了表示不满在操作符注释里写你名字的频率，“bills!*&^%~code!”[9]。

5.9 富包装器

Scala 基本类型的可调用的方法远多于前几段里面讲到的。表 5.4 里罗列了几个例子。这些方法的使用要通过隐式转换（implicit conversion），一种在第 21 章才会提到的技术。现在所有要知道的就是本章介绍过的每个基本类型，都对应着一个“富包装器”提供许多额外的方法。因此，想要看到基本类型的所有可用方法，还应该查看一下每个基本类型的富包装器的 API 文档。这些类可参见表 5.5。

93

表 5.4 一些富操作

代码	结果
0 max 5	5
0 min 5	0
-2.7 abs	2.7
-2.7 round	-3L
1.5 isInfinity	false
(1.0/0)isInfinity	true
4 to 6	Range(4,5,6)
"bob" capitalize	"Bob"
"robert" drop 2	"bert"

表 5.5 富包装类

基本类型	富包装
Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Int	scala.runtime.RichInt
Long	scala.runtime.RichLong
Char	scala.runtime.RichChar
String	scala.runtime.RichString
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble
Boolean	scala.runtime.RichBoolean

5.10 小结

本章的便当包（take-away）里主要放了这么几样小菜，Scala 的操作符就是方法调用，Scala 的基本类型的富变体的隐式转换可以增加更多有用的方法。下一章里，我们会告诉你用那种让操作符改头换面的函数式风格来设计对象到底是什么意思。

94

[8]译注：a?

[9]注：现在你应该已经知道对于这段代码，Scala 编译器会调用成(bills.!*&^%~(code)).!()。

第 6 章

函数式对象

有了从前几章获得的 Scala 基础知识，你已经为探索如何在 Scala 里设计出拥有更全面特征的对象做好了准备。本章的重点在于定义函数式对象，也就是说，不具有任何可改变状态的对象的类。为了便于说明和演示，我们将创造若干以有理数作为不可变对象来建模的类版本。在此过程中，我们会学习 Scala 面向对象编程的更多知识：类参数和构造函数，方法和操作符，私有成员，子类方法重载，先决条件检查，同类方法重载和自指向。

6.1 类 Rational 的规格说明书

有理数（rational number）是一种可以表达为比率 $\frac{n}{d}$ 的数字，这里的 n 和 d 是数字，其中 d 不能为零。n 被称作分子（*numerator*），d 被称作分母（denominator）。有理数的例子有：$\frac{1}{2}$，$\frac{2}{3}$，$\frac{112}{239}$ 和 $\frac{2}{1}$。与浮点数相比较，有理数的优势是小数部分得到了完全表达，没有舍入或估算。

本章我们将要设计的类必须建模有理数的行为，包括允许它们执行加、减、乘还有除运算。有理数的加法运算，首先是求出公分母，然后才能把两个分子相加。例如，要计算 $\frac{1}{2}+\frac{2}{3}$，先把左操作数的上下部分都乘上 3，右操作数的两部分都乘上 2，得到了 $\frac{3}{6}+\frac{4}{6}$。把两个分子相加产生结果，$\frac{7}{6}$。有理数的乘法比较简单，只要把两个分子相乘，然后再把两个分母相乘即可。因此，$\frac{1}{2}\times\frac{2}{5}$ 得到了 $\frac{2}{10}$，还可以约分表示成它的“归元化（normalized）”形式 $\frac{1}{5}$。除法是把右操作数分子分母调换，然后做乘法。例如 $\frac{1}{2}/\frac{3}{5}$ 与 $\frac{1}{2}\times\frac{5}{3}$ 相同，结果是 $\frac{5}{6}$。

这里有一个或许不怎么重要的发现是，数学上有理数不具有可变的状态。有理数可以相加，但结果是新的有理数，原来的数不会被“改变”。我们本章设计的不可变的 Rational 类将秉承这一属性。每个有理数都能被表示成 Rational 对象。当两个 Rational 对象相加时，将创建新的带着累加结果的 Rational 对象。

本章还将捎带提一些能让你写出感觉像 Scala 原生语言库支持的方法。例如，本章结尾你将能如此使用 Rational 类：

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> (oneHalf / 7) + (1 twoThirds)
res0: Rational = 17/42
```

6.2 创建 Rational

着手设计 Rational 类之前应先考虑客户程序员将如何创建新的 Rational 对象。假设我们已决定让 Rational 对象是不可变的，我们将需要那个客户在创建实例时提供所有需要的数据（本例中，是分子和分母）。因此，我们应该这么开始设计：

```
class Rational(n: Int, d: Int)
```

这行代码里首先应当注意到的是如果类没有主体，就不需要指定一对空的花括号（当然你如果想的话也可以）。在类名 Rational 之后的括号里的 n 和 d，被称为类参数（class parameter）。Scala 编译器会收集这两个类参数并创造出带同样的两个参数的主构造器（primary constructor）。

96

不可变对象的权衡

不可变对象提供了若干强于可变对象的优点和一个潜在的缺点。首先，不可变对象常常比可变对象更易理清头绪，因为它们没有会随着时间变化的复杂的状态空间。其次，你可以很自由地传递不可变对象，但对于可变对象来说，传递给其他代码之前，需要先建造个以防万一的副本。第三，一旦不可变对象完成构造之后，就不会有线程因为并发访问而破坏对象内部状态，因为根本没有线程可以改变不可变对象的状态。第四，不可变对象让哈希表键值更安全。比方说，如果可变对象在进入 HashSet 之后被改变，那么你下一次查找这个 HashSet 时就找不到这个对象了。

不可变对象唯一的缺点就是有时需要复制很大的对象表而可变对象的更新可以在原址发生。有些情况下这会变得难以快速完成而可能产生性能瓶颈。因此，要求功能库提供不可变类的可变版本也并非是一件稀奇的事情。例如，类 StringBuilder 是不可变的 String 的可变版本。第 18 章中我们会更详细地说明如何在 Scala 里设计可变对象。

> **注意**
>
> 这个最初的 Rational 例子凸显了 Java 和 Scala 之间的不同。Java 类具有可以带参数的构造器，而 Scala 类可以直接带参数。Scala 的写法更简洁——类参数可以直接在类的主体中使用；没必要定义字段然后写赋值函数把构造器的参数复制到字段里。这在无形中省略了很多固定写法，尤其是对小类来说。

Scala 编译器将把类内部的任何既不是字段也不是方法定义的代码编译至主构造器中。例如，可以这样打印输出除错消息：

97

```
class Rational(n: Int, d: Int) {
  println("Created "+n+"/"+d)
}
```

Scala 编译器将把这段代码的 println 调用放进 Rational 的主构造器。因此，println 调用将在每次创建新的 Rational 实例时打印这条除错信息：

```
scala> new Rational(1, 2)
Created 1/2
res0: Rational = Rational@90110a
```

6.3 重新实现 toString 方法

Rational 实例创建时，解释器打印输出“Rational@a0b0f5”。解释器通过调用 Rational 对象的 toString 方法得到这个貌似无稽的字符串。默认情况下，Rational 类继承了 java.lang.Object 类的 toString 实现，只会打印类名、@符号和十六进制数。toString 的结果主要可以用在除错打印语句、日志消息、测试错误报告，经由解释器或除错器输出的信息，以辅助程序员分析问题。目前 toString 提供的结果不能给出 Rational 数值的任何特征信息，因此没什么大用。为了能够打印出 Rational 的分子和分母。我们可以重载（override）Rational 类的 toString 方法实现，如下：

```
class Rational(n: Int, d: Int) {
  override def toString = n +"/"+ d
}
```

方法定义前的 override 修饰符说明这是对原方法定义的重载，第 10 章会更进一步说明。现在有理数可以较为美观地显示，所以我们去掉了前一版本里 Rational 类的 println 除错语句。并在解释器里对 Rational 的新版本测试如下：

```
scala> val x = new Rational(1, 3)
x: Rational = 1/3

scala> val y = new Rational(5, 7)
y: Rational = 5/7
```

6.4 检查先决条件

下一步，我们将把视线转向当前主构造器行为里的一些问题上。如本章早些时候提到的，有理数的分母不能为零。然而目前主构造器会接受把零传递给 d：

```
scala> new Rational(5, 0)
res1: Rational = 5/0
```

面向对象编程的优点之一就是它允许你把数据封装在对象之内以确保数据在整个生命周期中的有效性。像 Rational 这样的不可变对象，这就意味着你必须确保对象创建时数据的有效性（并且，确保对象的确是不可变的，这样数据就不会在之后变成无效的状态）。由于零做分母对 Rational 来说是无效状态，因此在把零传递给 d 的时候，务必不能让 Rational 被构建出来。

解决这个问题的最佳方案是为主构造器定义先决条件（precondition）说明 d 必须为非零值。先决条件是对传递给方法或构造器的值的限制，是调用者必须满足的需求。一种方式是使用 require 方法[1]，如：

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  override def toString = n +"/"+ d
}
```

require 方法带一个布尔型参数。如果传入的值为真，require 将正常返回。反之，require 将抛出 IllegalArgumentException 阻止对象被构造。

[1]注：require 方法定义在 scala 包的孤立对象 Predef 上。4.4 节中提到过，Predef 对象的成员都已被自动引入到每个 Scala 源文件中了。

6.5 添加字段

现在主构造器已经有了先决条件检查，我们继续关注加法功能的实现。首先在类 Rational 上定义公开的 add 方法，参数为另一个 Rational 对象。其次为了保持 Rational 的不可变性，add 方法不能把传入的有理数加到自己身上，而是必须创建并返回全新的 Rational 累加值。你或许以为可以这么定义 add：

```
class Rational(n: Int, d: Int) { // 编译不过
  require(d != 0)
  override def toString = n +"/"+ d
  def add(that: Rational): Rational =
    new Rational(n * that.d + that.n * d, d * that.d)
}
```

但很不幸，编译器会提示说：

```
<console>:11: error: value d is not a member of Rational
               new Rational(n * that.d + that.n * d, d * that.d)
                                    ^
<console>:11: error: value d is not a member of Rational
               new Rational(n * that.d + that.n * d, d * that.d)
                                                           ^
```

尽管类参数 n 和 d 都在 add 代码可引用的范围内，但是 add 方法仅能访问调用对象自身的值。因此，当 add 实现访问 n 或 d 的时候，编译器能够提供给你这些类参数值。但 that.n 或 that.d 绝对无法访问，因为 that 并不是调用 add 的 Rational 对象[2]。要想访问 that 的 n 和 d，需要把它们放在字段中，方法说明见清单 6.1[3]。

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  override def toString = numer +"/"+ denom
  def add(that: Rational): Rational =
   new Rational(
     numer * that.denom + that.numer * denom,
     denom * that.denom
   )
}
```

清单 6.1 带字段的 Rational

清单 6.1 里，我们增加了两个字段，分别是 numer 和 denom，并用类参数 n 和 d 初始化[4]。我们还改变了 toString 和 add 的实现，让它们使用字段，而不是类参数。这个版本能够编译通过，并且实现了加法功能：

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2
```

[2]注：实际上，如果 that 指的是同一个 add 的调用者时，Rational 可以对自身执行累加。但是因为有可能传递给 add 方法任意 Rational 对象，所以编译器仍然不允许使用 that.n 这种说法。

[3]注：10.6 节介绍的参数化字段（parametric field），为编写类似代码提供了捷径。

[4]注：尽管 n 和 d 在类范围内有效，但因为它们只是构造器的一部分，所以 Scala 编译器不会为它们自动构造字段。所以有了以上的代码，Scala 编译器才会产生包含两个 Int 字段（分别是 numer 和 denom）的类。

```
scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> oneHalf add twoThirds
res3: Rational = 7/6
```

另外，我们之前不能在对象外部直接访问有理数的分子和分母，现在可以了。只要访问公共的 numer 和 denom 字段即可：

```
scala> val r = new Rational(1, 2)
r: Rational = 1 / 2

scala> r.numer
res4: Int = 1

scala> r.denom
res5: Int = 2
```

6.6 自指向

关键字 this 指向当前执行方法被调用的对象实例，或者如果使用在构造器里的话，就是正被构建的对象实例。例如，如果要添加方法 lessThan，测试当前有理数是否小于传入的参数，可以写成：

```
def lessThan(that: Rational) =
  this.numer * that.denom < that.numer * this.denom
```

这里，this.numer 指向 lessThan 被调用的那个对象的分子。你也可以去掉 this 前缀而只写 numer，两种写法意义相同。

下面是不能省略 this 的例子，在 Rational 类里添加 max 方法返回指定有理数和参数中的较大者：

```
def max(that: Rational) =
  if (this.lessThan(that)) that else this
```

这里，第一个 this 可写可不写，写成(lessThan(that))也是一样；但第二个 this 表示当测试为假的时候的方法结果；如果你省略它，就什么都返回不了了。

6.7 辅助构造器

有些时候一个类里需要多个构造器。Scala 里主构造器之外的构造器被称为辅助构造器（auxiliary constructor）。比方说，分母为 1 的有理数只写分子的话就更为简洁。如，对于$\frac{5}{1}$来说，可以只写成 5。因此，代之以写成 Rational(5, 1)，如果类的二次开发者可以简写成 Rational(5)或许感觉会更好些。而这就需要给 Rational 添加只传入分子的辅助构造器并预先设定分母为 1。具体参见清单 6.2：

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  val numer: Int = n
  val denom: Int = d

  def this(n: Int) = this(n, 1) // 辅助构造器

  override def toString = numer +"/"+ denom
```

```
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

清单 6.2 带有从构造器的 Rational

Scala 的辅助构造器定义开始于 def this(...)。函数主体几乎完全是主构造器的调用，并传递了参数 n 做分子和 1 做分母。输入下列代码到解释器里可以实际感受辅助构造器的效果：

```
scala> val y = new Rational(3)
y: Rational = 3/1
```

102

Scala 里的每个辅助构造器的第一个动作都是调用同类的别的构造器。换句话说就是，每个 Scala 类的每个辅助构造器都是以“this(...)”形式开头的。被调用的构造器既可以是主构造器(像 Rational 这个例子)，也可以是源文件中早于调用构造器定义的其他辅助构造器。规则的根本结果就是每个 Scala 的构造器调用终将结束于对主构造器的调用。因此主构造器是类的唯一入口点。

> **注意**
>
> 若熟悉 Java，你或许会奇怪为什么 Scala 构造器的规则比 Java 的还要严格。Java 里构造器的第一个动作只有两个选择：要么调用同类的其他构造器，要么直接调用超类的构造器。Scala 的类里面只有主构造器可以调用超类的构造器。Scala 里这种严格的限制实际上是为了让 Scala 的构造器与 Java 构造器相比更为清晰和简洁而付出的代价。超类，构造器调用和继承交互的细节将在第 10 章解释。

103

6.8 私有字段和方法

上一个版本的 Rational 里，我们只是分别用 n 初始化了 numer，用 d 初始化了 denom。结果，Rational 的分子和分母可能比实际需要的要大。例如有理数 $\frac{66}{42}$，可以约简化为无差别的最简形式，$\frac{11}{7}$，但目前 Rational 的主构造器没有这个功能：

```
scala> new Rational(66, 42)
res6:  Rational = 66/42
```

有理数的约简化，需要把分子和分母都除以最大公约数（greatest common divisor）。如：66 和 42 的最大公约数是 6。（另一种说法就是，6 是能够同时除尽 66 和 42 的最大整数。）$\frac{66}{42}$ 的分子和分母都除以 6 就产生它的最简形式，$\frac{11}{7}$。清单 6.3 展示代码如下：

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
```

```
    denom * that.denom
  )

  override def toString = numer +"/"+ denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

清单 6.3 带私有字段和方法的 Rational

这里，我们添加了私有字段 g，并修改了 numer 和 denom 的初始化器（initializer——是对变量进行初始化，如 numer 的“n / g”的代码）。因为 g 是私有的，所以只能在类的主体之内被访问，外部不可见。我们还添加了私有方法 gcd 用来计算传入的两个 Int 的最大公约数。比方说，gcd(12, 8)是 4。正如 4.1 节所见，把 private 关键字放在字段或方法定义之前可以让定义私有化。私有的“助手方法”gcd 的目的是把有用的代码从类的其他部分（这里是主构造器）中分离出来。为了确保 g 始终是正的，我们传入 n 和 d 的绝对值，调用 abs 即可获得任意整数的绝对值。

Scala 编译器将把 Rational 的三个字段的初始化器依照它们在源代码中出现的次序放入主构造器。所以 g 的初始化代码 gcd(n.abs, d.abs)将在另外两个之前执行，因为它在源文件中出现得最早。g 将被初始化为类参数 n 和 d 的绝对值的最大公约数。然后再被用于 numer 和 denom 的初始化。通过把 n 和 d 整除它们的最大公约数 g，每个 Rational 都将以最简形式构造：

```
scala> new Rational(66, 42)
res7: Rational = 11/7
```

6.9 定义操作符

当前 Rational 加法的实现只是可用，但不够好用。你或许会困惑为什么整数或浮点数可以写成：

```
x + y
```

但有理数就必须写成：

```
x.add(y)
```

或至少是：

```
x add y
```

没有理由可以解释说明为什么就该如此。有理数和别的数应该是一样的。数学的角度上看它们甚至比浮点数更接近于自然的本质。为什么就不能使用自然的数学操作符呢？Scala 里面你做得到。本章后续部分，我们会告诉你怎么做。

第一步是用通常的数学的符号替换 add 方法。这可以直接做到，因为 Scala 里 + 是合法的标识符。我们可以用 + 定义方法名。既然已经做了这么多，不妨再多做一点，类似地定义 * 方法以实现乘法，结果在清单 6.4 中：

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
```

```
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def +(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def *(that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  override def toString = numer +"/"+ denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

清单 6.4 带操作符方法的 Rational

有了这种方式定义的 Rational 类，你现在可以这么写了：

```
scala> val x = new Rational(1, 2)
x: Rational = 1/2

scala> val y = new Rational(2, 3)
y: Rational = 2/3

scala> x + y
res8: Rational = 7/6
```


与以往一样，最后一行的操作符语法等同于方法调用。你也可以写成：

```
scala> x.+(y)
res9: Rational = 7/6
```

不过这样写可读性不佳。

另外要提的是基于 5.8 节中曾经说过的 Scala 的操作符优先级规则，Rational 里面的*方法要比+方法优先级更高。或者说，Rational 涉及+和*操作的表达式会按照预期的方式那样执行。例如，x + x * y 会当作 x + (x * y) 而不是 (x + x) * y：

```
scala> x + x * y
res10: Rational = 5/6

scala> (x + x) * y
res11: Rational = 2/3

scala> x + (x * y)
res12: Rational = 5/6
```

6.10 Scala 的标识符

现在我们已经看到了 Scala 里两种最重要的标识符构成方式：字母数字式和操作符。Scala 在构成标识符方面有非常灵活的规则。除了这两种之外你会看到还有其他的两种。本节将说明所有的这四种标识

符的构成方式。

字母数字标识符（alphanumeric identifier）以字母或下划线开始，之后可以跟字母、数字或下划线。‘$’字符也被当作是字母，但被保留作为 Scala 编译器产生的标识符之用。用户程序里的标识符不应该包含‘$’字符，尽管能够编译通过；但这样做有可能导致与 Scala 编译器产生的标识符发生名称冲撞。

Scala 遵循 Java 的驼峰式[5]标识符习惯，例如 `toString` 和 `HashSet`。尽管下划线在标识符内是合法的，但在 Scala 程序里并不常用，部分原因是为了保持与 Java 一致，同样也由于下划线在 Scala 代码里有许多其他非标识符用法。因此，最好避免使用像 `to_string`、`__init__`或 `name_`这样的标识符。字段、方法参数、本地变量，还有函数的驼峰式名称，应该以小写字母开始，如：`length`、`flatMap`，还有 `s`。类和特质的驼峰式名称应该以大写字母开始，如：`BigInt`、`List`，还有 `UnbalancedTreeMap`[6]。

> **注意**
>
> 不建议在标识符结尾使用下划线。比如说，如果尝试如下定义，“`val name_: Int = 1`”，将导致编译器错误。编译器会认为你正在定义名为“`name_:`”的变量。为了使它编译通过，你将需要在冒号之前插入额外的空格，如：“`val name_ : Int = 1`”。

Scala 与 Java 的习惯不一致的地方在于常量名。Scala 里，constant 这个词并不等同于 `val`。尽管 `val` 在被初始化之后的确保持不变，但它仍然是变量。比方说，方法参数是 `val`，但是每次方法被调用的时候这些 `val` 都可以代表不同的值。而常量更持久。比方说，`scala.Math.Pi` 被定义为很接近实数π的双精度值，表示圆周和它直径的比值。这个值不太可能改变，因此 `Pi` 显然是个常量。常量还可以用来命名代码里的幻数（magic number）：字变量没有含义，如果出现在多个地方将会变得极度糟糕。你还可能会需要定义用在模式匹配里的常量，15.2 节将给出相关的应用场景。在 Java 里，习惯上常量名全都是大写的，用下划线分隔单词，如 `MAX_VALUE` 或 `PI`。Scala 里，习惯只是第一个字母必须大写。因此，Java 风格的常量名（如 `X_OFFSET`）在 Scala 里也可以用，但是 Scala 的惯例是常数也用驼峰式风格，如 `XOffset`。

操作符标识符（operator identifier）由一个或多个操作符字符组成。操作符字符是一些如+，:，?，~或#的可打印的 ASCII 字符[7]。以下是一些操作符标识符的例子：

```
+  ++  :::  <?>  :->
```

Scala 编译器将在内部“粉碎”操作符标识符以转换成合法的内嵌‘$’的 Java 标识符。例如，标识符`:->`将被内部表达为`$colon$minus$greater`。若你想从 Java 代码访问这个标识符，就应使用这种内部表达方式。

Scala 里的操作符标识符可以变得任意长，因此在 Java 和 Scala 间有一些小差别。Java 里，输入 `x<-y` 将会被拆分成四个词汇符号，所以写成 `x < - y` 也没什么不同。Scala 里，`<-`将被作为一个标识符拆分，而得到 `x <- y`。所以如果你想要以第一种方式解释，需要在‘<’和‘-’字符间加一个空格。这大概不会是实际应用中的问题，因为没什么人会在 Java 里不加空格或括号的写成 `x<-y`。

混合标识符（mixed identifier）由字母数字组成，后面跟着下划线和一个操作符标识符。如 `unary_+`

---

[5]注：这种风格被称为驼峰式（camel case），因为标识符由若干首字母大写的内嵌单词组成。

[6]注：在 16.5 节里，你将看到有些时候你需要给一种被称为样本类（case class）的特殊的类一个仅用操作符字符组成的名字。例如，Scala 的 API 包含了一种叫做`::`的类，用来方便 `List` 里的模式匹配。

[7]注：更精确地说，操作符字符属于数学符号（Sm）或其他符号（So）的 Unicode 集，或者不包含字母、数字、括号、方括号、花括号、单或双引号，或者下划线、句号、分号、冒号、回退字符的 7 位 ASCII 字符。

被用做定义一元的‘+’操作符的方法名。或myvar_=被用做定义赋值操作符的方法名。多说一句，混合标识符格式myvar_=是由Scala编译器产生的用来支持属性（property）的，第18章进一步说明。

字面量标识符（literal identifier）是用反引号`...`包括的任意字符串。如：

```
`x` `<clinit>` `yield`
```

思路是你可以把运行时环境认可的任意字符串放在反引号之间当作标识符。结果总被当作是Scala标识符。即使包含在反引号间的名称是Scala保留字，这个规则也有效。在Java的Thread类中访问静态的yield方法是它的典型用例。你不能写Thread.yield()，因为yield是Scala的保留字。然而可以在反引号里引用方法的名称，例如Thread.`yield`()。

109

6.11 方法重载

回到类Rational上来。在最近一次改变之后，有理数的加法和乘法有了自然风格的表示方法。但别忘了还有混合运算。例如，有理数和整数之间不能做乘法，因为‘*’的操作数只能是有理数。所以对于有理数r不能写r * 2，只能写成r * new Rational(2)，很不美观。

为了让Rational用起来更方便，可以在类上增加能够执行有理数和整数之间的加法和乘法的新方法。同时，还可以再加上减法和除法。结果展示在清单6.5中：

```
class Rational(n: Int, d: Int) {

  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)

  def - (that: Rational): Rational =
    new Rational(
      numer * that.denom - that.numer * denom,
      denom * that.denom
    )

  def - (i: Int): Rational =
    new Rational(numer - i* denom, denom)

  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  def * (i: Int): Rational =
    new Rational(numer * i, denom)
```

```
  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)

  override def toString = numer+"/"+denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

清单 6.5 含有重载方法的 Rational

现在每种数学方法都有两个版本：一个带有理数做参数，另一个带整数。或者可以说，这些方法名都被重载（overload）了，因为每个名字现在都被多个方法使用。例如，+这个名字分别被带 Rational 参数和另一个带 Int 参数的方法使用。方法调用时，编译器会拣出正确地匹配了参数类型的重载方法版本。例如，如果 x.+(y)的参数 y 是 Rational，编译器就会拣带有 Rational 参数的+方法来用。相反如果参数是整数，编译器就会拣带有 Int 参数的+方法做替代。如果输入：

```
scala> val x = new Rational(2, 3)
x: Rational = 2/3

scala> x * x
res13: Rational = 4/9

scala> x * 2
res14: Rational = 4/3
```

你会看到*方法的调用取决于例子中的右操作数类型。

> **注意**
>
> Scala 分辨重载方法的过程与 Java 极为相似。任何情况下，被选中的重载版本都是最符合参数静态类型的那个。如果发现不止一个最符合的版本，编译器会抛出“参考模糊”的错误。

6.12 隐式转换

现在可以写 r * 2 了，或许可以交换一下操作数，就像 2 * r 这样。不幸的是这样做还不可以：

```
scala>2 * r
<console>:7: error: overloaded method value * with
alternatives (Double)Double <and> (Float)Float <and>
(Long)Long <and> (Int)Int <and> (Char)Int <and> (Short)Int
<and> (Byte)Int cannot be applied to (Rational)
      2 * r
        ^
```

这里的问题是 2 * r 等同于 2.*(r)，因此这是在整数 2 上的方法调用。但 Int 类没有带 Rational 参数的乘法——没办法，因为类 Rational 不是 Scala 库的标准类。

不过 Scala 有另外的方法解决这个问题：可以创建在需要的时候自动把整数转换为有理数的隐式转换。试着把这行代码加入解释器：

```
scala> implicit def intToRational(x: Int) = new Rational(x)
```

这行代码定义了从 Int 到 Rational 的转换方法。方法前面的 implicit 修饰符告诉编译器可以在一些情况下自动调用。定义了转换之后，你现在可以重试之前失败的例子了：

```
scala> val r = new Rational(2,3)
r: Rational = 2/3

scala>2 * r
res16: Rational = 4/3
```

请注意要隐式转换起作用，需要定义在作用范围之内。如果你把隐式方法定义放在类 Rational 之内，它就不在解释器的作用范围。目前，你要在解释器内直接定义它。

如你所见，隐式转换是把库变得更灵活和更方便的非常强大的技术。不过正因它们如此强大，所以也很容易被误用。第 21 章里将介绍隐式转换的更多细节，包括在需要时把它们带入作用范围的方式[8]。

112

6.13 一番告诫

如本章所演示的，用操作符名称来创建方法并定义隐式转换能帮助你设计出让客户代码更简洁和易于理解的库。Scala 给了你大量的设计这种易于使用库的能力，不过请牢记：能力越大责任越大。

如果无技巧性地使用，操作符方法和隐式转换都会让客户代码变得难以阅读和理解。因为隐式转换是由编译器隐式地应用的，而不是显式地写在源代码中，对于二次开发者来说正在应用哪个隐式转换并非显而易见。而且尽管操作符方法通常会使得客户代码更简洁，但它只会在客户程序员能够识别和记住每个操作符的意思的程度上让程序变得更易读。

在设计库的时候你应记在脑袋里的目标并不是仅仅让客户代码简洁，而是让它变得更可读，更易懂。简洁性经常是可读性的重要部分，但不能简洁得过了头。通过设计出有助于简洁、可读、易懂的客户代码的库，你将帮助客户程序员更多产地工作。

6.14 小结

本节中，你看到了 Scala 类中更多的方面。你看到了如何向类添加参数，如何定义若干构造函数，如何像方法那样定义操作符，以及如何把让类使用起来更自然。或许最重要的是，你在本章中发现定义和使用不可变状态对象在 Scala 里是一种非常自然的方式。

尽管本章显示的 Rational 最终版本满足了开始时候的需求说明，但它仍有改善的空间。我们将在后续的书里回到这个例子上。例如在第 28 章，你会学到如何重载 equals 和 hashcode 来允许 Rational 在用==比较或放入到哈希表时表现得更好。在第 21 章你会学到如何把隐式方法定义放在 Rational 的伴生对象中，这样当二次开发者在使用 Rational 时就可以更容易地把它们置于作用范围中。

113 114

[8]译注：实际上如果定义了隐式转换，前面的 Rational 类甚至可以不用定义两套方法：仅定义对 Rational 的操作即可，隐式转换会自动把操作数转换为 Rational 类的实例。

第 7 章

内建控制结构

Scala 内建的控制结构屈指可数，仅有 if、while、for、try、match 和函数调用而已。如此之少的理由是，Scala 从语法层面支持函数字面量。因此，代之以在基本语法之上逐一添加高级的内建控制结构，Scala 可以把它们实现在函数库中。第 9 章将更细致地描述其实现方式。本章则主要说明这仅有的几个内建控制结构。

可能你已经发现，几乎所有的 Scala 的控制结构都会产生某个值。这是函数式语言所采用的方式，程序被看成是计算值的活动，因此程序的各个部件也不应例外。你也可以把这种方式看做指令式语言发展趋势的逻辑推演，函数调用可以返回值（或者更新被当作参数传入的输出变量）。另外，指令式语言还有三元操作符（如 C、C++和 Java 的?:操作符），其行为与 if 一致，但结果却是值。Scala 采用了这种三元操作符模型，但仍称为 if。换句话说，Scala 的 if 可以产生值。于是 Scala 让这种趋势继续发展，使得 for、try 和 match 也产生值。

程序员能够用结果值简化代码，就如同用函数的返回值那样。如果没有这种机制，程序员就必须创建临时变量来保存控制结构中的计算结果。去掉这些临时变量能让代码更简洁并避免类似于多个分支内变量设置不一致而导致的问题。

总而言之，Scala 的基础控制结构虽然少，但也足以支持指令式语言里所有的实质内容。进一步说，由于它们都能产生结果值，因此有助于缩短代码。为了说明这些结构的工作原理，本章将首先对 Scala 的基础控制结构建立感性认识。

7.1 If 表达式

Scala 的 if 语句与其他语言相比没有什么不同。它可以检测条件并根据其是否为真，执行两个分支中的一个。下面是常见的以指令式风格编写的例子：

```
var filename = "default.txt"
if (!args.isEmpty)
  filename = args(0)
```

这段代码声明了变量 filename，并初始化为默认值。然后使用 if 表达式检查程序是否带有参数。如果是，就把变量改成参数列表中定义的值；如果没有参数，就保持变量的默认值设定。

这段代码还有优化的余地，就像第 2 章的第三步曾经提到过，Scala 的 if 是能返回值的表达式。清

单 7.1 的效果与之前的相同，但没有使用任何 var：

```
val filename =
  if (!args.isEmpty) args(0)
  else "default.txt"
```

清单 7.1 在 Scala 里根据条件做初始化的惯例

这一次，if 有了两个分支。如果 args 不为空，那么选择初始化元素 args(0)；否则，选择默认值。if 表达式返回被选中的值，然后用这个值初始化 filename 变量。这段代码稍微短了点儿，不过它的实际优点在于使用了 val 而不是 var。使用 val 体现了函数式的风格，并且具有与 Java 的 final 变量类似的效果。它可以告诉读者，代码变量将不再改变，从而节省了他们审查变量作用域的所有代码，以检查它是否改变的工作。

116

使用 val 而不是 var 的第二点好处是它能更好地支持等效推论（equational reasoning）。在表达式没有副作用的前提下，引入的变量等效于计算它的表达式。因此，无论何时都可以用表达式替代变量名。如，要替代 println(filename)，可以这么写：

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

选择权在你，怎么写都行。使用 val 有助于安全地执行这类重构以不断革新代码。

尽可能寻找使用 val 的机会。它们能让你的代码既容易阅读又容易重构。

7.2 While 循环

Scala 的 while 循环与其他语言中一致。包括状态判断和循环体，只要状态保持为真，循环体就一遍遍被执行。清单 7.2 演示如下：

```
def gcdLoop(x: Long, y: Long): Long = {
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

清单 7.2 用 while 循环计算最大公约数

117

Scala 同样也有 do-while 循环，它与 while 循环基本没有区别，只是把状态检查移到了循环体之后。清单 7.3 演示的 Scala 脚本可以使用 do-while 回显从标准输入设备读入的文本行，直到读入空行为止：

```
var line = ""
do {
```

```
    line = readLine()
    println("Read: " + line)
  } while (line != "")
```

清单 7.3 用 do-while 从标准输入读取信息

while 和 do-while 结构之所以被称为“循环”，而不是表达式，是因为它们不能产生有意义的结果。结果的类型是 Unit，是表明存在并且唯一存在类型为 Unit 的值，称为 unit value，写成()。()的存在是 Scala 的 Unit 不同于 Java 的 void 的地方。可以在解释器里尝试输入下列代码：

```
scala> def greet() { println("hi") }
greet: ()Unit

scala> greet() == ()
hi
res0: Boolean = true
```

由于方法体之前没有等号，greet 被定义为结果类型为 Unit 的过程。因此，greet 返回 unit 值，()。这被下一行确证：比较 greet 的结果和 unit 值()的相等性，可以得到返回 true。

对 var 再赋值等式本身也是 unit 值，这是另一种与此相类似的架构。比如，假设尝试把下面这种 Java（或者 C 或 C++）里 while 循环的惯用方法用在 Scala 里读取文本行，你会遇到麻烦：

```
var line = ""
while ((line = readLine()) != "") // 不起作用
  println("Read: "+ line)
```

编译这段代码时，Scala 会警告你使用!=比较类型为 Unit 和 String 的值将永远返回 true。虽然在 Java 里，赋值语句可以返回被赋予的那个值，本例中是标准输入读取的文本行，但同样的情况下 Scala 的赋值语句只能得到 unit 值()。因此，赋值语句“line = readLine()”的值将永远返回()而不是""。结果，这个 while 循环的状态将永远不会是假，于是循环永远无法结束。

由于 while 循环不产生值，因此它经常被纯函数式语言所舍弃。这种语言只有表达式，没有循环。尽管如此，Scala 仍然包含了 while 循环，因为有些时候指令式的解决方案更容易读懂，尤其是对那些以指令式背景为主导的程序员来说尤其如此。例如，如果你想编码重复执行某进程直到某种状态改变的算法，那么 while 循环可以直接表达；而使用函数式风格编写的话，那大概要用递归实现，或许对某些代码的读者来说这就不是那么显而易见的了。

如，清单 7.4 演示了另一种计算两个数的最大公约数的方式[1]。给定同样的值 x 和 y，清单 7.4 里的 gcd 函数能够返回与清单 7.2 的 gcdLoop 函数相同的结果。两者的差异在于 gcdLoop 是指令式风格，使用了 var 和 while 循环，而 gcd 则是函数式风格的体现，它采用了递归（gcd 调用自身）并且不需要 var：

```
def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

清单 7.4 使用递归计算最大公约数

[1]注：清单 7.4 的 gcd 函数使用的方法与之前在清单 6.3 中的同名函数相同，目的是为类 Rational 计算最大公约数，主要的差别在于清单 7.4 的 gcd 的参数使用的是 Long 而不是 Int。

通常情况下，我们建议你质疑对 while 循环的使用，如同质疑对 var 的使用那样。实际上，while 循环和 var 经常是结对出现的。因为 while 循环不产生值，为了让程序发挥作用，while 循环通常不是更新 var 就是执行 I/O。前面的 gcdLoop 例子里可以看到：在 while 循环工作的时候，更新了 a 和 b 两个 var。因此，我们建议你在代码中更为审慎地使用 while 循环。如果对某个 while 或 do 循环没有非用不可的理由，那么请尝试用别的方式实现同样的功能。

7.3 For 表达式

Scala 的 for 表达式是枚举操作的“瑞士军刀”。它可以让你用不同的方式把若干简单的成分组合起来以表达各种各样的枚举。简单使用适合于通常的任务，例如枚举整数序列。高级表达式可以同时枚举不同类型的多个集合，可以用任意条件过滤元素，还可以制造新的集合。

枚举集合类

for 能做的最简单的事情就是把集合中所有元素都枚举一遍。如，清单 7.5 演示了打印当前目录所有文件名的例子。I/O 操作使用了 Java 的 API。首先，我们创建指向当前目录"."的 java.io.File 对象，然后调用它的 listFiles 方法。方法返回 File 对象数组，每个都代表当前目录包含的目录或文件。我们把结果数组保存在 filesHere 变量。

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere)
  println(file)
```

清单 7.5 用 for 表达式列举目录中的文件

通过使用被称为发生器（generator）的语法“file <- filesHere”，我们遍历了 filesHere 的元素。每一次枚举，名为 file 的新的 val 就被元素值初始化。编译器能够推断出 file 的类型是 File，因为 filesHere 是 Array[File]。对于每一次枚举，for 表达式的函数体 println(file)，将被执行一次。由于 File 的 toString 方法生成文件或目录的名称，因此当前目录的所有文件和目录的名称都会被打印出来。

for 表达式语法对任何种类的集合类都有效，而不只是数组[2]。比方说，表 5.4 中曾经出现的 Range 类型就是其中一例，你可以使用类似于“1 to 5”这样的语法很方便地创建 Range，然后用 for 做枚举。用法如下：

```
scala> for (i <- 1 to 4)
         println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

[2]注：更精确地说，<-符号右侧的表达式可以是具有合适签名的某种方法（这里是 foreach）的任意类型。Scala 编译器对于 for 表达式的处理细节将在第 23 章描述。

如果不想包括被枚举的 Range 的上边界，还可以用 until 替代 to：

```
scala> for (i <- 1 until 4)
         println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
```

像这样的整数枚举在 Scala 里是很平常的，但在其他语言中就没有那么容易了。其他语言中，你或许要采用如下方式遍历数组：

```
// Scala 中不常见...
for (i <- 0 to filesHere.length - 1)
  println(filesHere(i))
```

这个 for 表达式引入了变量 i，依次把它设成从 0 到 filesHere.length - 1 的整数值，然后对 i 的每次修改执行一次 for 表达式的循环体。对应于每一个 i 的值，filesHere 的第 i 个元素被取出并处理。

Scala 里不常见到这种枚举的原因，是集合对象本身可以直接被枚举。这样，代码可以变得更短并规避了许多枚举数组时频繁出现的超位溢出（off-by-one error）。枚举该从 0 开始还是从 1 开始？最后的索引值应该加-1，+1，还是什么都不加？这些问题很容易回答，但也很容易答错。因此还是避免碰到为佳。

121

过滤

有时你并不想枚举集合的全部元素，而只想过滤出某个子集。这可以通过在 for 表达式的括号中添加过滤器（filter），即 if 子句，实现。如清单 7.6 仅列表当前目录中以“.scala”结尾的文件名：

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere if file.getName.endsWith(".scala"))
  println(file)
```

清单 7.6　用带过滤器的 for 发现.scala 文件

或者也可以采用如下写法：

```
for (file <- filesHere)
  if (file.getName.endsWith(".scala"))
    println(file)
```

这段代码与清单 7.6 的输出结果相同，而且对于指令式背景的程序员来说看上去更熟悉一些。然而指令式的格式只是可选的实现方法之一，因为这个 for 表达式的执行目的只是打印的副作用并返回 unit 值()。正如我们将在本节后面看到的，for 表达式之所以被称为“表达式”是因为它能产生有意义的值，其类型取决于 for 表达式<-子句的集合。

需要的话，你可以包含更多的过滤器。只要不断添加 if 子句即可。例如，为了提高代码的健壮性，清单 7.7 仅仅打印输出文件而不是目录。这可以通过增加过滤器检查 file 的 isFile 方法做到：

```
for (
  file <- filesHere
  if file.isFile;
  if file.getName.endsWith(".scala")
) println(file)
```

122

清单7.7 在for表达式中使用多个过滤器

注意

如果在发生器中加入超过一个过滤器，if 子句必须用分号分隔。这也是清单 7.7 中的“if file.isFile”过滤器之后带着分号的原因。

嵌套枚举

如果加入多个<-子句，你就得到了嵌套的“循环”。比如，清单 7.8 的 for 表达式有两个嵌套循环。外层的循环枚举 filesHere，内层的枚举所有以.scala 结尾文件的 fileLines(file)。

```
def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines.toList

def grep(pattern: String) =
  for (
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    if line.trim.matches(pattern)
  ) println(file + ": " + line.trim)

grep(".*gcd.*")
```

清单7.8 在for表达式中使用多个发生器

如果愿意的话，你可以使用花括号代替小括号包裹发生器和过滤器。使用花括号的好处是可以省略使用小括号时必须加的分号。

流间（mid-stream）变量绑定

请注意前面的代码段中重复出现的表达式 line.trim。这是个不可忽略的计算，因此你或许希望只算一遍。这可以通过用等号（=）把结果绑定到新变量实现。绑定的变量被当作 val 引入和使用，不过不带关键字 val。清单 7.9 演示如下：

123

```
def grep(pattern: String) =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(pattern)
  } println(file + ": " + trimmed)

grep(".*gcd.*")
```

清单7.9 在for表达式里的流间赋值

代码中，名为 trimmed 的变量被从半路引入 for 表达式，并被初始化为 line.trim 的结果值。于是之后的 for 表达式在两个地方使用了新的变量，一次在 if 中，另一次在 println 中。

制造新集合

到现在为止所有的例子都只是对枚举值进行操作然后就释放，除此之外，你还可以创建一个值去记住每一次的迭代，只要在 for 表达式之前加上关键字 yield。比如，下面的函数鉴别出.scala 文件并保存在数组里：

```
def scalaFiles =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
  } yield file
```

for 表达式在每次执行的时候都会产生一个新值，本例中是 file。当 for 表达式完成的时候，结果将是包含了所有产生值的集合对象。对象的类型基于枚举子句处理的集合类型。本例中结果为 Array[File]，因为 filesHere 是数组并且产生的表达式类型是 File。

另外，请注意放置 yield 关键字的地方。对于 for-yield 表达式的语法是这样的：

```
for {子句} yield {循环体}
```

yield 在整个循环体之前。即使循环体是被花括号包围的代码块，也一定要把 yield 放在左括号之前，而不是代码块的最后一个表达式之前。写成如下的格式是不对的：

```
for (file <- filesHere if file.getName.endsWith(".scala")) {
  yield file // 语法错误!
}
```

清单 7.10 中，for 表达式首先把包含了当前目录所有文件的类型为 Array[File]的 filesHere 对象，转换成仅包含.scala 文件的数组。对于每一个对象，产生 Iterator[String]对象（fileLines 方法的结果，定义在清单 7.8 中），Iterator 对象提供了集合元素的枚举方法 next 和 hasNext。这个原始的枚举器又被过滤器转换为另一个 Iterator[String]，剩下的每一行元素都含有子字符串"for"。最终，对每行元素计算长度。这个 for 表达式的结果就是一个包含了这些长度的 Array[Int]数组。

```
val forLineLengths =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(".*for.*")
  } yield trimmed.length
```

清单 7.10　用 for 把 Array[File]转换为 Array[Int]

截止到目前，你已经看过了 Scala 的 for 表达式的所有主要特征。不过这一段过得实在是快了些，关于 for 表达式更透彻的介绍将在第 23 章给出。

7.4 使用 try 表达式处理异常

125

Scala 的异常和许多其他语言的一样。方法除了能以通常的方式返回值以外，还可以通过抛出异常中止执行。方法的调用者要么可以捕获并处理这个异常，或者也可以只是简单地中止掉，并把异常上升到调用者的调用者处。异常以这种方式上升，逐层释放调用堆栈，直到某个方法接手处理或不再剩下其他的方法。

抛出异常

异常的抛出看上去与 Java 的一模一样。首先创建一个异常对象然后用 throw 关键字抛出：

```
throw new IllegalArgumentException
```

尽管可能感觉有些出乎意料，但在 Scala 里，throw 也是有结果类型的表达式。下面举个有关结果类型的例子：

```
val half =
  if (n % 2 == 0)
    n / 2
  else
    throw new RuntimeException("n must be even")
```

这段代码的意思是，如果 n 是偶数，half 将被初始化为 n 的一半。如果 n 不是偶数，那么异常将在 half 被初始化为任何值之前被抛出。因此，无论怎么说，把抛出的异常当作任何类型的值都是安全的。任何使用经 throw 返回值的尝试都不会起作用，因此这样做不会有害处。

从技术角度上来说，抛出异常的类型是 Nothing。尽管 throw 不实际产生任何值，你还是可以把它当作表达式。这种小技巧或许看上去很怪异，但像在上面这样的例子里却常常很有用。if 的一个分支计算值，另一个抛出异常并得出 Nothing。整个 if 表达式的类型就是那个实际计算值的分支的类型。Nothing 类型将在 11.3 节中讨论。

捕获异常

126

清单 7.11 演示了捕获异常的语法。选择 catch 子句这种语法的原因是为了与 Scala 很重要的部分模式匹配（pattern matching）保持一致。模式匹配是一种很强大的特征，将在本章概述并在第 15 章详述。

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

try {
  val f = new FileReader("input.txt")
  // 使用并关闭文件
} catch {
  case ex: FileNotFoundException => // 处理丢失的文件
  case ex: IOException => // 处理其他 I/O 错误
}
```

清单 7.11 Scala 的 try-catch 子句

这个 try-catch 表达式的处理方式与其他语言中的异常处理一致。首先执行程序体，如果抛出异常，则依次尝试每个 catch 子句。本例中，如果异常是 FileNotFoundException，那么第一个子句将被执行。如果是 IOException 类型，第二个子句将被执行。如果都不是，那么 try-catch 将终结并把异常上升出去。

> **注意**
>
> 你将很快发现与 Java 的差别存在于 Scala 里不需要捕获检查异常（checked exception），或把它们声明在 throws 子句中。如果愿意，你可以用@throws 注解声明 throws 子句，但这不是必须的。@throws 的更多细节参见 29.2 节。

finally 子句

如果想让某些代码无论方法如何中止都要执行的话，可以把表达式放在 finally 子句里。例如，你或许想让打开的文件即使在方法抛出异常退出的情况下也要确保被关闭。清单 7.12 演示如下：

```
import java.io.FileReader

val file = new FileReader("input.txt")
try {
  // 使用文件
} finally {
  file.close() // 确保关闭文件
}
```

清单 7.12 Scala 的 try-finally 子句

> **注意**
>
> 清单 7.12 演示了确保非内存资源，如文件、套接字，或者数据库链接被关闭的惯例方式。首先占有资源；然后开始 try 代码块使用资源；最后，在 finally 代码块中关闭资源。这种惯例与在 Java 里的一样。然而，Scala 里还可以使用另一种被称为出借模式（loan pattern）的技巧更简洁地达到同样的目的。出借模式将在 9.4 节描述。

生成值

和其他大多数 Scala 控制结构一样，try-catch-finally 也产生值。如，清单 7.13 演示了如何尝试拆分 URL，但如果 URL 格式错误就使用默认值。返回的结果是，如果没有异常抛出，则对应于 try 子句；如果抛出异常并被捕获，则对应于相应的 catch 子句。如果异常被抛出但没被捕获，表达式就没有返回值。由 finally 子句计算得到的值，即使有也会被抛弃。通常 finally 子句做一些诸如关闭文件之类的清理工作，它们不应该修改主函数体或 catch 子句中计算的值。

```
import java.net.URL
import java.net.MalformedURLException

def urlFor(path: String) =
  try {
    new URL(path)
  } catch {
    case e: MalformedURLException =>
```

```
        new URL("http://www.scala-lang.org")
    }
```

清单 7.13 能够产生值的 catch 子句

如果熟悉 Java，不说你也知道，Scala 的行为与 Java 的差别仅在于 Java 的 try-finally 不产生值。在 Java 里，如果 finally 子句包含了返回语句，或抛出一个异常，这个返回值或异常将“凌驾”于任何之前在 try 代码块或某个 catch 子句里产生的值或异常之上。例如，如果函数有如下定义：

128

```
def f(): Int = try { return 1 } finally { return 2 }
```

调用 f()将返回 2。相反：

```
def g(): Int = try { 1 } finally { 2 }
```

调用 g()将返回 1。这两个例子足以令大多数程序员震惊，因此通常最好还是避免用 finally 子句返回值，而是把它理解为确保某些操作发生的途径，如关闭打开的文件。

7.5 匹配（match）表达式

Scala 的 match 表达式类似于其他语言中的 switch 语句，它可以提供给你在多个备选项（alternative）中做选择。基本上 match 表达式可以让你使用任意的模式（pattern）做选择，第 15 章会介绍。基本情况稍后再说。目前，只要考虑使用 match 在若干备选项中做选择。

作为演示，清单 7.14 里的脚本从参数列表读入食物名然后打印食物配料。match 表达式检查参数列表的第一个参数 firstArg。如果是字符串"salt"，就打印"pepper"，如果是"chips"，就打印"salsa"，如此递推。默认情况用下划线（_）说明，这是常用在 Scala 里作为占位符来表达未知值的通配符。

129

```
val firstArg = if (args.length >0) args(0) else ""

firstArg match {
  case "salt" => println("pepper")
  case "chips" => println("salsa")
  case "eggs" => println("bacon")
  case _ => println("huh?")
}
```

清单 7.14 有副作用的 match 表达式

与 Java 的 switch 语句比，匹配表达式还有一些重要的差别。其中之一是任何类型的常量，或其他什么东西，都能当成 Scala 里做比较用的样本（case），而不只是 Java 的 case 语句里面的整数类型和枚举常量，在这个例子里，备选项是字符串。另一个区别是在每个备选项的最后并没有 break。取而代之的是，break 是隐含的，也就是说，不允许从上一个备选项落入到下一个里面去的情况发生。通常这可以使代码变得更短，并且避免了一些错误产生的根源，程序员不再会因为疏忽而从一个选项落入到另一个。

然而，与 Java 的 switch 相比，最显著的差别或许是 match 表达式也能产生值。上面的例子里，

match 表达式的每个备选项不但可以通过打印输出值，还可以只生成返回值而不打印，具体见清单 7.15，match 表达式生成返回值并储存在 friend 变量里。这除了能让代码变得更短之外（至少减少了几个指令），还解开了两个原本应分离的关注点：首先是选择食物名，其次才是打印它。

```
val firstArg = if (!args.isEmpty) args(0) else ""

val friend =
  firstArg match {
    case "salt" =>"pepper"
    case "chips" =>"salsa"
    case "eggs" =>"bacon"
    case _ =>"huh?"
  }
println(friend)
```

清单 7.15 生成值的 match 表达式

7.6 不再使用 break 和 continue

你可能注意到了这里没有提到过 break 和 continue。Scala 去掉了这些命令因为它们与函数式字面量（下一章会谈到这个特征）啮合得不好。continue 在 while 循环中的意思很清楚，但是在函数式字面量中表示什么呢？虽然 Scala 既支持指令式风格也支持函数式风格，但在这点上它略微倾向于函数式编程，以换取语法上的简洁性。不过尽管如此也请不要着急，break 和 continue 并非编程之中不可或缺的，只要能充分利用函数字面量加以替代，甚至可以写出比原来更简洁的代码。

用 if 替换每个 continue 和用布尔变量替换每个 break 是最简单的方式。布尔变量用来说明 while 循环是否应该继续。比如说，假设你正搜索一个参数列表去查找以“.scala”结尾但不以连号开头的字符串。Java 里你可以（如果喜欢 while 循环，break 和 continue）如此写：

```
int i = 0;        // 在 Java 中...
boolean foundIt = false;
while (i < args.length) {
  if (args[i].startsWith("-")){
    i = i + 1;
    continue;
  }
  if (args[i].endsWith(".scala")) {
    foundIt = true;
    break;
  }
  i = i + 1;
}
```

如果要根据字面直译成 Scala 的代码，if 之后的 continue 这块可以改写成用 if 包裹 while 余下的全部内容。要去掉 break，只须增加一个布尔变量提示是否继续做下去即可，不过这里可以直接复用 foundIt。运用了这两点技巧之后，完工的代码参见清单 7.16：

```
var i = 0
var foundIt = false
```

```
while (i < args.length && !foundIt) {
  if (!args(i).startsWith("-")) {
    if (args(i).endsWith(".scala"))
      foundIt = true
  }
  i = i + 1
}
```

清单 7.16 不带 break 或 continue 的循环

这个版本与原来的 Java 代码非常像。所有的主要段落仍然存在并保持原顺序，有两个可重新赋值的变量及一个 while 循环。循环测试 i 是否小于 args.length，再检查"-"，最后检查".scala"。

如果要去掉清单 7.16 里面的 var，可以尝试的方案是把循环重写为递归函数。比方说，可以定义 searchFrom 函数，以整数值做输入，从该值向前搜索，并返回想要的参数的索引。若采用这种方式，代码看上去如清单 7.17 中这样：

```
def searchFrom(i: Int): Int =
  if (i >= args.length) -1
  else if (args(i).startsWith("-")) searchFrom(i + 1)
  else if (args(i).endsWith(".scala")) i
  else searchFrom(i + 1)

val i = searchFrom(0)
```

清单 7.17 不用 var 做循环的递归替代方法

这个版本给了函数一个能够看得懂的名字说明它在做什么，用递归替换了循环。每个 continue 都被带有 i+1 做参数的递归调用替换掉，并快速跳转处理下一个整数。许多人都发现一旦真正开始使用递归，这种编程风格更易于理解。

132

> **注意**
>
> Scala 编译器不会实际对清单 7.17 所示的代码生成递归函数。因为所有的递归调用都在尾调用（tail-call）位置，编译器会产生出与 while 循环类似的代码。每个递归调用将被实现为回到函数开始位置的跳转。尾调用优化将在 8.9 节讨论。

7.7 变量范围

到目前为止已经了解了 Scala 的内建控制结构，我们将在本节中使用它们举例说明 Scala 的变量范围是如何起作用的。

> **Java 程序员的快速通道**
>
> 如果你是 Java 程序员，你会发现 Scala 的范围规则几乎是 Java 的翻版。然而，两者之间仍然有一个差别，Scala 允许在嵌套范围内定义同名变量。因此即使是 Java 程序员，或许至少快速浏览一下也是有益的。

Scala 程序里所有的变量定义都存在有效作用范围（scope）。最常见的情况如，花括号通常引入了新

的作用范围，所以任何定义在花括号里的东西超出括号之后就脱离了范围[3]。作为演示，请看一下清单7.18里的函数：

```
def printMultiTable() {

  var i = 1
  // 这里只有 i 在范围内
  while (i <= 10) {

    var j = 1
    // 这里 i 和 j 在范围内
    while (j <= 10) {

      val prod = (i * j).toString
      // 这里 i，j 和 prod 在范围内

      var k = prod.length
      // 这里 i，j，prod 和 k 在范围内

      while (k <4) {
        print(" ")
        k += 1
      }

      print(prod)
      j += 1
    }
    // i 和 j 仍在范围内；prod 和 k 脱离范围

    println()
    i += 1
  }
  // i 仍在范围内；j，prod 和 k 脱离范围
}
```

清单7.18 打印乘法表时的变量范围

printMultiTable函数打印输出了乘法表[4]。函数的第一个语句引入了变量i并初始化为整数1。然后变量名i可以在函数余下的部分里使用。

接下去是while循环语句：

```
while (i <= 10) {

  var j = 1
  ...
}
```

这里可以使用i因为它仍在范围内。while循环的第一个语句里引入了另一个变量，叫做j，并再次初始化为1。因为变量j定义在while循环的花括号内，所以只能在循环内部使用。如果想尝试在while循环的花括号之后，比如说注释“prod和k脱离范围”的位置，再使用变量j，程序就不能编译通过了。

[3]注：这条规则有几个例外，因为在Scala里有时候可以用花括号代替小括号。for表达式的可选语法是这种用法的一个例子，已在7.3节描述。

[4]注：清单7.18演示的printMultiTable函数是用指令式风格写的。我们将在下一节中以函数式风格重构。

本例中定义的所有变量——i，j，prod 和 k——都是本地变量（local variable），即，对于定义它们的函数来说是“局部”的。每次函数被调用的时候都会使用一整套全新的本地变量。

一旦变量被定义了，在同一个范围内就无法定义同样的名字了。比如，下面的脚本不能编译通过：

```
val a = 1
val a = 2// 编译不过
println(a)
```

然而，你可以在内部范围内定义与外部范围里名称相同的变量。下列脚本将能够编译通过并运行：

```
val a = 1;
{
  val a = 2// 编译通过
  println(a)
}
println(a)
```

执行时，这个脚本会先打印 2，然后打印 1，因为定义在内部花括号里的 a 是不同的变量，所以仅在花括号内部有效[5]。Scala 和 Java 间要注意的差异是，Java 不允许在内部范围内创建与外部范围变量同名的变量。在 Scala 程序里，内部变量被认为遮蔽（shadow）了同名的外部变量，因为在内部范围中外部变量变得不可见。

135

或许你已经注意到了某些在解释器里看上去像是遮蔽的东西：

```
scala> val a = 1
a: Int = 1

scala> val a = 2
a: Int = 2

scala> println(a)
2
```

解释器里，你可以随意重用变量名。撇开别的不说，如果你在解释器中首次定义变量的时候犯了什么错误，也允许你改变主意。能这么做的理由在于，概念上，解释器为每次输入的新语句都创建了新的嵌套范围。因此，可以把之前解释器里的代码虚拟化为：

```
val a = 1;
{
  var a = 2;
  {
    println(a)
  }
}
```

这段代码可以作为 Scala 脚本编译和执行，而且像输入到解释器里的代码那样，打印输出 2。不过请记住这样的代码对读者来说是很混乱的，因为在嵌套范围中变量名称拥有了新的涵义。通常，选择新的、有意义的变量名比不时遮蔽外部变量的做法更为妥当。

[5]注：另外，本例中 a 的第一个定义之后需要加分号，因为 Scala 的分号推断机制不会在这里加上分号。

7.8 重构指令式风格的代码

为了帮助你对函数式风格有更多的认识，本节我们将重构清单 7.18 中以指令式风格打印乘法表的代码。我们的函数式风格的可选方案展示在清单 7.19 中展示。

清单 7.18 中的代码从两方面暴露出了指令式风格的痕迹。首先，调用 printMultiTable 方法有副作用：在标准输出上打印乘法表。在清单 7.19 中，我们重构了函数，让它把乘法表作为字符串返回。由于函数不再执行打印，我们把它重命名为 multiTable。正如前面提到过的，没有副作用的函数的优点之一是可以方便做单元测试。要测试 printMultiTable，你需要重定义 print 和 println 以便检查输出的正确性，而测试 multiTable 就简单多了，只要检查返回结果即可。

136

```
// 以序列形式返回一行乘法表
def makeRowSeq(row: Int) =
  for (col <- 1 to 10) yield {
    val prod = (row * col).toString
    val padding = " " * (4 - prod.length)
    padding + prod
  }

// 以字符串形式返回一行乘法表
def makeRow(row: Int) = makeRowSeq(row).mkString

// 以字符串形式返回乘法表，每行记录占一行字符串
def multiTable() = {
  val tableSeq = // 行记录字符串的序列
    for (row <- 1 to 10)
      yield makeRow(row)
  tableSeq.mkString("\n")
}
```

清单 7.19 创建乘法表的函数式方法

printMultiTable 函数的另一个暗示其指令式风格的信号在于它的 while 循环和 var。与之相对，multiTable 函数使用了 val 类型变量、for 表达式及帮助函数（helper function），并调用了 mkString 方法。

我们提炼出两个帮助函数，makeRow 和 makeRowSeq，使代码容易阅读。函数 makeRowSeq 使用了可以从 1 到 10 枚举列数的 for 表达式，其函数体计算行和列的乘积，决定乘积前占位的空格，并生成由占位空格、乘积合并成的字符串。for 表达式返回以这些生成字符串作为元素的序列（scala.Seq 的某个子类）。而另一个帮助函数 makeRow，只是对 makeRowSeq 返回的结果调用 mkString 函数，把序列中的字符串合并成一个字符串返回。

multiTable 方法首先使用 for 表达式的结果初始化 tableSeq，这个 for 表达式从 1 到 10 枚举行数，对每行调用 makeRow 获得该行的字符串。因为字符串前缀 yield 关键字，所以表达式的结果就是行字符串的序列。现在仅剩的工作就是把字符串序列转变为单一字符串。mkString 的调用完成这项工作，并且由于我们传递进去"\n"，因此每个字符串结尾插入了换行符。如果把 multiTable 返回

137

的字符串传递给 println，你将看到与调用 printMultiTable 所生成的输出结果相同：

```
 1    2    3    4    5    6    7    8    9   10
 2    4    6    8   10   12   14   16   18   20
 3    6    9   12   15   18   21   24   27   30
 4    8   12   16   20   24   28   32   36   40
 5   10   15   20   25   30   35   40   45   50
 6   12   18   24   30   36   42   48   54   60
 7   14   21   28   35   42   49   56   63   70
 8   16   24   32   40   48   56   64   72   80
 9   18   27   36   45   54   63   72   81   90
10   20   30   40   50   60   70   80   90  100
```

7.9 小结

Scala 的内建控制结构极少，不过刚好够使用。它们与对应的指令式控制结构功能接近，但因为它们能够产生值，所以也支持函数式风格。同样重要的是，它们很仔细地做了取舍，因此为 Scala 最强大的特征之一——函数式字面量留余地，下一章将对此详加说明。

第 8 章

函数和闭包

当程序变得庞大时，需要将其分割成更小、更易管理的片段。为分割控制流，Scala 为有经验的程序员提供了熟悉的方式：把代码分割成函数。Scala 提供了许多 Java 中没有的定义函数的方式。除了作为对象成员函数的方法之外，还有内嵌在函数中的函数，函数字面量和函数值。本章将介绍这些函数。

8.1 方法

定义函数最通用的方法是作为某个对象的成员。这种函数被称为方法（method）。作为例子，清单 8.1 演示了两个方法，它们共同读取指定文件名的文件并打印输出所有长度超过设定宽度的文本行，每个打印输出的文本行前缀所在文件的文件名：

```
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) {
    val source = Source.fromFile(filename)
    for (line <- source.getLines)
      processLine(filename, width, line)
  }

  private def processLine(filename:String, width:Int, line:String) {

    if (line.length > width)
      println(filename+": "+line.trim)
  }
}
```

清单 8.1 带私有的 processLine 方法的 LongLines 对象

processFile 方法以 filename 和 width 为参数。文件名用来创建 Source 对象，并通过 for 表达式的发生器调用 source 的 getLines 方法。第 3 章的第十二步曾经提到，getLines 返回的枚举器能在每次枚举中从文件里取出一行，包括换行符。然后 for 表达式通过调用 processLine 帮助方法处理所有的文件行。processLine 方法带三个参数：filename、width 和 line。它检查是否文件行的长度超过给定长度，如果是，就打印文件名，冒号，然后是文件行。

为了从命令行里使用 LongLines，我们需要创建一个应用，把第一个命令行参数当作行长度，并把后续的参数解释为文件名[1]：

[1]注：本书通常不会在例子程序中检查命令行参数的合法性，这既是为了保护林木资源（节省用纸），也是为了减少会模糊例子重点的套路代码。代价就是，如果有输入错误，我们的例子程序将抛出异常，而非产生有帮助的错误信息。

```
object FindLongLines {
  def main(args: Array[String]) {
    val width = args(0).toInt
    for (arg <- args.drop(1))
      LongLines.processFile(arg, width)
  }
}
```

下面是如何使用这个应用程序去发现 LongLines.scala 里超过 45 字符长度的行（只有一行）：

```
$ scala FindLongLines 45 LongLines.scala
LongLines.scala: def processFile(filename: String, width: Int) {
```


到这里为止，这些与面向对象语言能做的类似。然而 Scala 里的函数概念比方法更宽泛。后续章节中将详细说明函数的其他描述方法。

8.2 本地函数

上节中 processFile 方法的建立演示了函数式编程风格的重要设计原则：程序应该被解构成若干小的函数，每块实现一个定义完备的任务，每块都非常小。这种风格的好处是它让程序员能够把这些小模块灵活组装成更复杂的事物。每个小块应该充分简化到足以单独理解。

这种方式的问题是：所有这些帮助函数的名称可能污染程序的命名空间。在解释器里这不太成问题，但是一旦函数被打包成可复用的类和对象，就最好让类的使用者不可见。因为它们经常不能独立表达什么意思，并且如果之后用其他方式重写类的话，也常会希望能确保足以删掉这些帮助方法的灵活度。

Java 里，达成这个目的的主要工具是私有方法，这种方式在 Scala 里同样有效，如清单 8.1 里描述的，但是 Scala 提供了另一种方式：你可以把函数定义在别的函数之内。就好像本地变量那样，这种本地函数仅在包含它的代码块中可见。举例如下：

```
def processFile(filename: String, width: Int) {

  def processLine(filename:String,
      width:Int, line:String) {

    if (line.length > width)
      print(filename+": "+line)
  }
  val source = Source.fromFile(filename)
  for (line <- source.getLines) {
    processLine(filename, width, line)
  }
}
```


例中通过把私有方法 processLine 转换为局部方法 processFile，重构了清单 8.1 中原本的 LongLines 版本。为了做到这点，我们去掉了 private 修饰符，它仅能应用于方法（并且仅被方法需要），然后把 processLine 的定义放在 processFile 的定义里。作为本地函数，processLine 的范围局限于 processFile 之内，外部无法访问。

既然 processLine 被定义在 processFile 里，另一项改善有了可能性。注意 filename 和 width 是不加改变地传入帮助函数中，这不是必须的。因为本地函数可以访问包含其函数的参数。你可

以直接使用外部 processLine 函数的参数，如清单 8.2 所示：

```
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) {
    def processLine(line: String) {
      if (line.length > width)
        print(filename +": "+ line)
    }

    val source = Source.fromFile(filename)
    for (line <- source.getLines)
      processLine(line)
  }
}
```

清单 8.2 带局部 processLine 方法的 LongLines

更简单了，不是吗？这种对外层函数入参的直接使用是 Scala 里通用的嵌套思想的平常但也很有用的例子。7.7 节描述的嵌套和作用域原则可以应用于所有的 Scala 架构，包括函数。这是一个简单的原则，不过非常强大，尤其在函数是头等值（first-class）的语言中。

8.3 头等函数

Scala 的函数是头等函数（first-class function）。你不仅可以定义和调用函数，还可以把它们写成匿名的字面量（literal），并把它们作为值（value）传递。我们在第 2 章介绍了函数字面量并在图 2.2 里演示了基本语法。

函数字面量被编译进类，并在运行期实例化为函数值（function value）[2]。因此函数字面量和值的区别在于函数字面量存在于源代码，而函数值作为对象存在于运行期。这个区别很像类（源代码）和对象（运行期）之间的关系。

以下是对数执行递增操作的函数字面量的简单例子：

```
(x: Int) => x + 1
```

=>指明这个函数把左边的东西（任何整数 x）转变成右边的东西（x+1）。所以，这个函数可以把任意整数 x 映射为 x+1。

函数值是对象，所以如果你愿意，可以将其存入变量。它们也是函数，所以你可以使用通常的括号函数调用写法调用它们。以下是这两种操作的例子：

```
scala> var increase = (x: Int) => x + 1
increase: (Int) => Int = <function>

scala> increase(10)
res0: Int = 11
```

[2]注：任何函数值都是某个扩展了 scala 包的若干 FunctionN 特质之一的类的实例，如 Function0 是没有参数的函数，Function1 是有一个参数的函数等等。每个 FunctionN 特质有一个 apply 方法用来调用函数。

本例中，因为 increase 是 var，你可以随后再重新赋给它不同的函数值。

```
scala> increase = (x: Int) => x + 9999
increase: (Int) => Int = <function>

scala> increase(10)
res2: Int = 10009
```

143

如果你想让函数字面量包含多条语句，可以用花括号包住函数体，一行放一条语句，这样就组成了代码块。与方法一样，当函数值被调用时，所有的语句将被执行，而函数的返回值就是最后一行表达式产生的值。

```
scala> increase = (x: Int) => {
         println("We")
         println("are")
         println("here!")
         x + 1
       }
increase: (Int) => Int = <function>

scala> increase(10)
We
are
here!
res4: Int = 11
```

于是现在你已经看到了有如螺丝和螺帽的函数字面量和函数值。许多 Scala 库给你使用它们的机会。例如，所有的集合类都能用到 foreach 方法[3]。它以函数作为入参，并对每个元素调用该函数。下面是如何用它打印输出所有列表元素的代码：

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
10
```

144

另举一个例子，关于集合类型的 filter 方法。这个方法选择集合类型里可以通过用户提供的测试的元素。测试是通过调用作为入参的函数实现的。例如，函数(x: Int) => x > 0 可以用来作过滤，它把正整数映射为真，其他为假。下面说明如何把它当作 filter：

```
scala> someNumbers.filter((x: Int) => x > 0)
res6: List[Int] = List(5, 10)
```

类似于 foreach 和 filter 这样的方法将在本书后面进一步说明。第 16 章讨论了它们在类 List 中的使用。第 17 章讨论了它们在其他集合类型中的使用。

[3]注：foreach 方法被定义在特质 Iterable 中，它是 List，Set，Array，和 Map 的共有超特质。参见第 17 章相关细节。

8.4 函数字面量的短格式

Scala 提供了许多方法去除冗余信息并把函数字面量写得更简短。注意留意这些机会，因为它们能让你去掉代码里乱七八糟的东西。

一种让函数字面量更简短的方式是去除参数类型。因此，前面带过滤器的例子可以写成这样：

```
scala> someNumbers.filter((x) => x > 0)
res7: List[Int] = List(5, 10)
```

Scala 编译器知道 x 一定是整数，因为它看到你立刻使用了这个函数过滤整数列表（由 someNumbers 暗示）。这被称为目标类型化：target typing，因为表达式的目标使用——本例中 someNumbers.filter()的参数——影响了表达式的类型化——本例中决定了 x 参数的类型。目标类型化的精确细节并不重要。你可以简单地从编写一个不带参数类型的函数字面量开始，并且，如果编译器不能识别，再加上类型。几次之后你就对什么情况下编译器能否解开谜题有感觉了。

某些参数的类型是被推断的，省略其外的括号，是第二种去除无用字符的方式。前面例子里，x 两边的括号不是必须的：

```
scala> someNumbers.filter(x => x > 0)
res8: List[Int] = List(5, 10)
```

8.5 占位符语法

如果想让函数字面量更简洁，可以把下划线当做一个或更多参数的占位符，只要每个参数在函数字面量内仅出现一次。比如，_ > 0 对于检查值是否大于零的函数来说就是非常短的标注：

```
scala> someNumbers.filter(_ > 0)
res9: List[Int] = List(5, 10)
```

你可以把下划线看作表达式里需要被“填入”的“空白”。这个空白在每次函数被调用的时候用函数的参数填入。例如，由于 someNumbers 在第 96 页被初始化为值 List(-11, -10, -5, 0, 5, 10)，filter 方法会把_ > 0 里的空格首先用-11 替换，就如-11 > 0，然后用-10 替换，如-10 > 0，然后用-5，如-5 > 0，这样直到 List 的最后一个值。因此，函数字面量_ > 0 与稍微冗长一点儿的 x => x > 0 相同，演示如下：

```
scala> someNumbers.filter(x => x > 0)
res10: List[Int] = List(5, 10)
```

有时你把下划线当作参数的占位符，编译器可能无法推断缺失的参数类型。例如，假设你只是写_ + _：

```
scala> val f = _ + _
<console>:4: error: missing parameter type for expanded
function ((x$1, x$2) => x$1.$plus(x$2))
       val f = _ + _
                   ^
```

这种情况下，你可以使用冒号指定类型，如下：

```
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function>

scala> f(5, 10)
res11: Int = 15
```

146

请注意_ + _将扩展成带两个参数的函数字面量。这样也解释了为何仅当每个参数在函数字面量中最多出现一次时，你才能使用这种短格式。多个下划线指代多个参数，而不是单个参数的重复使用。第一个下划线代表第一个参数，第二个下划线代表第二个，第三个，……，如此类推。

8.6 部分应用函数

尽管前面的例子里下划线替代的只是单个参数，你还可以使用单个下划线替换整个参数列表。例如，写成println(_)，或者更好的方法你还可以写成println_。下面是一个例子：

```
someNumbers.foreach(println _)
```

Scala把这种短格式直接看作是你输入了下列代码：

```
someNumbers.foreach(x => println(x))
```

因此，这个例子中的下划线不是单个参数的占位符。它是整个参数列表的占位符。请记住要在函数名和下划线之间留一个空格，因为不这样做编译器会认为你是在说明一个不同的符号，比方说是，似乎不存在的名为println_的方法。

以这种方式使用下划线时，你就正在写一个部分应用函数（partially applied function）。Scala里，当你调用函数，传入任何需要的参数，实际是在把函数应用到参数上。如，给定下列函数：

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (Int,Int,Int)Int
```

你就可以把函数sum应用到参数1，2和3上，如下：

```
scala> sum(1, 2, 3)
res12: Int = 6
```

147

部分应用函数是一种表达式，你不需要提供函数需要的所有参数。代之以仅提供部分，或不提供所需参数。比如，要创建调用sum的部分应用表达式，而不提供任何3个所需参数，只要在“sum”之后放一个下划线即可。然后可以把得到的函数存入变量。举例如下：

```
scala> val a = sum _
a: (Int, Int, Int) => Int = <function>
```

有了这个代码，Scala编译器以部分应用函数表达式，sum_，实例化一个带3个缺失整数参数的函数值，并把这个新的函数值的索引赋给变量a。当你把这个新函数值应用于3个参数之上时，它就转而调用sum，并传入这3个参数：

```
scala> a(1, 2, 3)
res13: Int = 6
```

实际发生的事情是这样的：名为a的变量指向一个函数值对象。这个函数值是由Scala编译器依

照部分应用函数表达式 sum_，自动产生的类的一个实例。编译器产生的类有一个 apply 方法带 3 个参数[4]。之所以带 3 个参数是因为 sum_表达式缺少的参数数量为 3。Scala 编译器把表达式 a(1,2,3) 翻译成对函数值的 apply 方法的调用，传入 3 个参数 1，2，3。因此 a(1,2,3)是下列代码的短格式：

```
scala> a.apply(1, 2, 3)
res14: Int = 6
```

Scala 编译器根据表达式 sum_自动产生的类里的 apply 方法，简单地把这 3 个缺失的参数前转到 sum，并返回结果。本例中 apply 调用了 sum(1,2,3)，并返回 sum 返回的 6。

这种一个下划线代表全部参数列表的表达式的另一种用途，就是把它当作转换 def 为函数值的方式。例如，如果你有一个本地函数，如 sum(a: Int, b: Int, c: Int): Int，你可以把它“包装”成与 apply 方法具有同样的参数列表和结果类型的函数值。当你把这个函数值应用到某些参数上时，它依次把 sum 应用到同样的参数，并返回结果。尽管不能把方法或嵌套函数赋值给变量，或当作参数传递给其他方法，但是如果你通过在名称后面加下划线的方式把方法或嵌套函数包装在函数值中，就可以做到了。

现在，尽管 sum_确实是一个偏函数，或许对你来说为什么这么称呼并不是很熟悉。这个名字源自于函数未被应用于它所有的参数。在 sum _的例子里，它没有应用于任何参数。不过还可以通过提供某些但不是全部需要的参数表达一个偏函数。举例如下：

```
scala> val b = sum(1, _: Int, 3)
b: (Int) => Int = <function>
```

这个例子里，你提供了第一个和最后一个参数给 sum，但中间参数缺失。因为仅有一个参数缺失，Scala 编译器会产生一个新的函数类，其 apply 方法带一个参数。在使用一个参数调用的时候，这个产生的函数的 apply 方法调用 sum，传入 1，传递给函数的参数，还有 3。如下：

```
scala> b(2)
res15: Int = 6
```

这个例子里，b.apply 调用了 sum(1,2,3)。

```
scala> b(5)
res16: Int = 9
```

这个例子里，b.apply 调用了 sum(1,5,3)。

如果你正在写一个省略所有参数的偏程序表达式，如 println _或 sum _，而且在代码的那个地方正需要一个函数，你可以去掉下划线从而表达得更简明。例如，代之以打印输出 someNumbers 里的每一个数字（定义在第 92 页）的这种写法：

```
someNumbers.foreach(println _)
```

你可以只是写成：

```
someNumbers.foreach(println)
```

[4]注：产生的类扩展了特质 Function3，定义了 3 个参数的 apply 方法。

最后一种格式仅在需要写函数的地方，如例子中的 foreach 调用，才能使用。编译器知道这种情况需要一个函数，因为 foreach 需要一个函数作为参数传入。在不需要函数的情况下，尝试使用这种格式将引发一个编译错误。举例如下：

```
scala> val c = sum
<console>:5: error: missing arguments for method sum...
follow this method with `_' if you want to treat it as
   a partially applied function
       val c = sum
               ^

scala> val d = sum _
d: (Int, Int, Int) => Int = <function>

scala> d(10, 20, 30)
res17: Int = 60
```

为什么要使用尾下划线?

Scala 的部分应用函数语法凸显了 Scala 与经典函数式语言，如 Haskell 或 ML 之间设计折中的差异。在经典函数式语言中，部分应用函数被当作普通的例子。更进一步，这些语言拥有非常严格的静态类型系统能够暴露出你在部分应用中可能犯的所有错误。Scala 与指令式语言如 Java 关系近得多，在这些语言中没有应用所有参数的方法会被认为是错误的。进一步说，子类型推断的面向对象的传统和全局的根类型能接受一些被经典函数式语言认为是错误的程序。

举例来说，如果你误以为 List 的 drop(n: Int) 方法是 tail()，那么你会忘记需要传递给 drop 一个数字。你或许会写，"println(drop)"。如果 Scala 采用部分应用函数在哪儿都 OK 的经典函数式传统，这个代码就将通过类型检查。然而，你会惊奇地发现这个 println 语句打印的输出将总是<function>！可能发生的事情是表达式 drop 将被看作是函数对象。因为 println 可以带任何类型对象，这个代码可以编译通过，但会产生出乎意料的结果。

为了避免这样的情况，Scala 需要你指定显示省略的函数参数，尽管标志简单到仅用一个 '_'。也只有在需要函数类型的地方，Scala 才允许你省略这个仅用的_。

8.7 闭包

到这里为止，所有函数字面量的例子仅参考了传入的参数。例如，(x: Int) => x > 0 里，函数体 x > 0 用到的唯一变量 x，被定义为函数参数。然而也可以参考定义在其他地方的变量：

```
(x: Int) => x + more // more是多少?
```

函数把"more"加入参考，但什么是 more 呢？从这个函数来看，more 是个自由变量：free variable，因为函数字面量自身没有给出其含义。相对地，x 变量是一个绑定变量：bound variable，因为它在函数的上下文中有明确意义：被定义为函数的唯一参数是 Int。如果你尝试独立使用这个函数字面量，范围内没有任何 more 的定义，编译器会报错说：

```
scala> (x: Int) => x + more
<console>:5: error: not found: value more
       ( x: Int) => x + more
                        ^
```

另一方面，只要有一个叫做 more 的某种东西，同样的函数字面量将工作正常：

```
scala> var more = 1
more: Int = 1

scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function>

scala> addMore(10)
res19: Int = 11
```

依照这个函数字面量在运行时创建的函数值（对象）被称为闭包：closure。名称源自于通过“捕获”自由变量的绑定，从而对函数字面量执行的“关闭”行动。不带自由变量的函数字面量，如(x: Int) => x + 1，被称为封闭项（closed term），这里项（term）指的是一小部分源代码。因此依照这个函数字面量在运行时创建的函数值严格意义上来讲就不是闭包，因为(x: Int) => x + 1 在编写的时候就已经封闭了。但任何带有自由变量的函数字面量，如(x: Int) => x + more，都是开放项（open term）。因此，任何以(x: Int) => x + more 为模板在运行期创建的函数值将必须捕获对自由变量，more，的绑定。因此得到的函数值将包含指向捕获的 more 变量的索引。又由于函数值是关闭这个开放项(x: Int) => x + more 的行动的最终产物，因此被称为闭包。

这个例子带来一个问题：如果 more 在闭包创建之后被改变了会发生什么事？Scala 里，答案是变化如下：

```
scala> more = 9999
more: Int = 9999

scala> addMore(10)
res21: Int = 10009
```

直觉上，Scala 的闭包捕获了变量本身，而不是变量指向的值[5]。就像前面演示的例子，依照(x: Int) => x + more 创建的闭包看到了闭包之外做出的对 more 的变化。反过来也同样。闭包对捕获变量作出的改变在闭包之外也可见。下面是一个例子：

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> var sum = 0
sum: Int = 0

scala> someNumbers.foreach(sum += _)
scala> sum
res23: Int = -11
```

例子用一个循环的方式计算 List 的累加和。变量 sum 处于函数字面量 sum += _的外围，函数字面量把数累加到 sum 上。尽管这是一个在运行期改变 sum 的闭包，作为结果的累加值，-11，仍然在闭包之外可见。

如果闭包访问了某些在程序运行时有若干不同备份的变量，例如闭包使用了某个函数的本地变量，

[5]注：相对地，Java 的内部类根本不允许你访问外围范围内可以改变的变量，因此到底是捕获了变量还是捕获了它当前具有的值就没有差别了。

并且函数被调用很多次会怎样？每一次访问使用的是变量的哪个实例？

如下的说法通用于所有情况：使用的实例是那个在闭包被创建的时候活跃的。例如，以下是创建和返回“递增”闭包的函数：

```
def makeIncreaser(more: Int) = (x: Int) => x + more
```

每次函数被调用时都会创建一个新闭包。每个闭包都会访问闭包创建时活跃的 more 变量。

```
scala> val inc1 = makeIncreaser(1)
inc1: (Int) => Int = <function>

scala> val inc9999 = makeIncreaser(9999)
inc9999: (Int) => Int = <function>
```

调用 makeIncreaser(1)时，捕获值 1 当作 more 的绑定的闭包被创建并返回。相似地，调用 makeIncreaser(9999)，捕获值 9999 当作 more 的闭包被返回。当你把这些闭包应用到参数上（本例中，只有一个参数，x，必须被传入），回来的结果依赖于闭包被创建时 more 是如何定义的：

```
scala> inc1(10)
res24: Int = 11

scala> inc9999(10)
res25: Int = 10009
```

尽管本例中 more 是已经返回方法的调用参数也没有区别。在这样的情况下，Scala 编译器重新做了布局以使得捕获的参数继续存在于堆中，而不是堆栈中，因此可以保留在创建它的方法调用之外。这种重新布局的工作都是自动完成的，因此不需要你操心，只须关注捕获你想要的变量：val、var 或参数即可。

153

8.8 重复参数

Scala 中，你可以指明函数的最后一个参数是重复的。从而允许客户向函数传入可变长度参数列表。想要标注一个重复参数，可在参数的类型之后放一个星号。例如：

```
scala> def echo(args: String*) =
         for (arg <- args) println(arg)
echo: (String*)Unit
```

这样定义，echo 可以被零个至多个 String 参数调用：

```
scala> echo()

scala> echo("one")
one

scala> echo("hello", "world!")
hello
world!
```

函数内部，重复参数的类型是声明参数类型的数组。因此，echo 函数里被声明为类型“String*”的 args 的类型实际上是 Array[String]。然而，如果你有一个合适类型的数组，并尝试把它当作重

复参数传入，你会得到一个编译器错误：

```
scala> val arr = Array("What's", "up", "doc?")
arr: Array[java.lang.String] = Array(What's, up, doc?)

scala> echo(arr)
<console>:7: error: type mismatch;
 found   : Array[java.lang.String]
 required: String
       echo(arr)
            ^
```


要实现这个做法，你需要在数组参数后添加一个冒号和一个_*符号，像这样：

```
scala> echo(arr: _*)
What's
up
doc?
```

这个标注告诉编译器把 arr 的每个元素当作参数，而不是当作单一的参数传给 echo。

8.9 尾递归

7.2 节中提到过，如果想把更新 var 的 while 循环转换成仅使用 val 这种更函数式的风格的话，有时候你可以使用递归。下面的例子是通过不断改善猜测数字来逼近一个值的递归函数：

```
def approximate(guess: Double): Double =
  if (isGoodEnough(guess)) guess
  else approximate(improve(guess))
```

这样的函数，带合适的 isGoodEnough 和 improve 的实现，经常用在查找问题中。如果想要 approximate 函数执行得更快，你或许会被诱惑使用 while 循环编写以尝试加快它的速度，如：

```
def approximateLoop(initialGuess: Double): Double = {
  var guess = initialGuess
  while (!isGoodEnough(guess))
    guess = improve(guess)
  guess
}
```

两种 approximate 版本哪个更好？就简洁性和避免 var 而言，第一个，函数式的胜出。但指令式的方式是否更有效率呢？实际上，如果我们测量执行的时间，就会发现它们几乎完全相同！这可能很令人惊奇，因为相比简单地从循环结尾跳到开头而言，递归调用看上去要更花时间。

然而，在上面 approximate 的例子里，Scala 编译器可以应用一个重要的优化。注意递归调用是 approximate 函数体执行的最后一件事。像 approximate 这样，在它们最后一个动作调用自己的函数，被称为尾递归：tail recursive。Scala 编译器检测到尾递归就用新值更新函数参数，然后把它替换成一个回到函数开头的跳转。

事实上你不必刻意回避使用递归算法去解决问题。递归经常是比基于循环的更优美和简明的方案。

如果方案是尾递归，就无须付出任何运行期开销。

尾递归函数追踪

尾递归函数将不会为每个调用制造新的堆栈结构；所有的调用将在一个结构内执行。这可能会让检查程序的堆栈跟踪信息并失败的程序员感到惊奇。例如，这个函数调用自身若干次之后抛出了异常：

```
def boom(x: Int): Int =
  if (x == 0) throw new Exception("boom!")
  else boom(x - 1) + 1
```

这个函数不是尾递归，因为在递归调用之后执行了递增操作。如果执行它，你会得到预期的：

```
scala> boom(3)
java.lang.Exception: boom!
      at .boom(<console>:5)
      at .boom(<console>:6)
      at .boom(<console>:6)
      at .boom(<console>:6)
      at .<init>(<console>:6)
...
```

如果你现在修改了 boom 从而让它变成尾递归：

```
def bang(x: Int): Int =
  if (x == 0) throw new Exception("bang!")
  else bang(x - 1)
```

156

你会得到：

```
scala> bang(5)
java.lang.Exception: bang!
      at .bang(<console>:5)
      at .<init>(<console>:6)
...
```

157

这回，你仅看到了 bang 的一个堆栈结构。或许你会认为 bang 在调用自己之前就崩溃了，但事实并非如此。如果你认为你会在看到堆栈跟踪时被尾调用优化搞糊涂，你可以用开关项关掉它：

```
-g:notailcalls
```

把这个参数传给 scala 的 shell 或者 scalac 编译器。定义了这个选项，你就能得到一个长长的堆栈跟踪：

```
scala> bang(5)
java.lang.Exception: bang!
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .<init>(<console>:6)
...
```

尾调用优化

approximate 的编译后代码实质上与 approximateLoop 的编译后代码相同。两个函数编译后都是同样的 13 个 Java 字节码指令。如果你看一下 Scala 编译器对尾递归方法 approximate 产生的字节码，你会看到尽管 isGoodEnough 和 improve 都被方法体调用 approximate 却没有。Scala 编译器优化了递归调用：

```
public double approximate(double);
  Code:
   0:   aload_0
   1:   astore_3
   2:   aload_0
   3:   dload_1
   4:   invokevirtual#24; //Method isGoodEnough:(D)Z
   7:   ifeq12
   10:  dload_1
   11:  dreturn
   12:  aload_0
   13:  dload_1
   14:  invokevirtual#27; //Method improve:(D)D
   17:  dstore_1
   18:  goto2
```

尾递归的局限

Scala 里尾递归的使用局限很大，因为 JVM 指令集使实现更加先进的尾递归形式变得很困难。Scala 仅优化了直接递归调用使其返回同一个函数。如果递归是间接的，就像在下面的例子里两个互相递归的函数，就没有优化的可能性了：

```
def isEven(x: Int): Boolean =
  if (x == 0) true else isOdd(x - 1)
def isOdd(x: Int): Boolean =
  if (x == 0) false else isEven(x - 1)
```

同样，如果最后一个调用是一个函数值你也不能获得尾调用优化。请考虑下列递归代码的实例：

```
val funValue = nestedFun _
def nestedFun(x: Int) {
  if (x != 0) { println(x); funValue(x - 1) }
}
```

funValue 变量指向一个实质是包装了 nestedFun 的调用的函数值。当你把这个函数值应用到参数上，它会转向把 nestedFun 应用到同一个参数，并返回结果。因此你或许希望 Scala 编译器能执行尾调用优化，但在这个例子里做不到。因此，尾调用优化限定了方法或嵌套函数必须在最后一个操作调用本身，而不是转到某个函数值或什么其他的中间函数的情况。（如果你还不能完全明白什么是尾递归，请参见 8.9 节）。

8.10 小结

本章带你全面浏览了 Scala 里的函数。除了方法之外，Scala 还提供本地函数、函数字面量及函数值。除了普通的函数调用之外，Scala 还提供了部分应用函数和带有重复参数的函数。如果可能，函数调用应被实现为优化的尾调用，这样许多漂亮的递归函数就能执行得像手动优化版本的 `while` 循环那样快。下一章将建立在这些函数的基础上，并展示 Scala 对函数丰富的支持，进而帮助你在控制上实现抽象。

159

第 9 章

控制抽象

在第 7 章里，我们指出 Scala 没有太多的内建控制抽象，因为它为你提供了创建自己的控制抽象的能力。前一章里，你已经学习了函数值。本章将展示给你如何把函数值应用到创建新的控制抽象。同时，你还将学习柯里化和传名参数。

9.1 减少代码重复

所有的函数都可以被分成通用部分（它们在每次函数调用中都相同），以及非通用部分（在不同的函数调用中可能会变化）。通用部分是函数体，而非通用部分必须由参数提供。当你把函数值用做参数时，算法的非通用部分就是它代表的某些其他算法。在这种函数的每一次调用中，你都可以把不同的函数值作为参数传入，于是被调用函数将在每次选用参数的时候调用传入的函数值。这种高阶函数（higher-order function）——带其他函数做参数的函数——给了你额外的机会去组织和简化代码。

高阶函数的好处之一是它们能让你创造控制抽象从而减少代码重复。例如，假设你正在编写文件浏览器，并且想要提供 API，以便使用者搜索匹配某些标准的文件。首先，你加入了搜索文件名结束于特定字符串的机制。这能让你的用户发现，比方说，所有扩展名为“.scala”的文件。你可以通过在单例对象中定义公开的 filesEnding 方法提供这样的 API，如：

```
object FileMatcher {
  private def filesHere = (new java.io.File(".")).listFiles

  def filesEnding(query: String) =
    for (file <- filesHere; if file.getName.endsWith(query))
      yield file
}
```

filesEnding 方法通过使用私有帮助方法 filesHere 接受当前目录所有文件的列表，然后基于是否每个文件名以用户特定的查询结尾来过滤它们。由于 filesHere 是私有的，filesEnding 方法是定义在提供给用户的 API，FilesMatcher 中唯一可以访问的方法。

目前为止还挺好，没有重复的代码。然而后来，你决定让别人可以基于文件名的任何部分做查询。这个功能可以良好地用于以下情况：你的用户记不住他们是以 phb-important.doc，stupid-phb-report.doc，may2003salesdoc.phb，或什么完全不同的名字来命名文件的，但他们认为“phb”出现在文件的什么地方。你回到工作并把这个函数加到你的 API，FileMatcher 中：

```
def filesContaining(query: String) =
  for (file <- filesHere; if file.getName.contains(query))
    yield file
```

这段函数与 filesEnding 很像。它搜索 filesHere，检查名称，并且如果名称匹配则返回文件。唯一的差别是这个函数使用了 contains 替代 endsWith。

随着时间的推移，程序变得更加成功。最后，你屈服于几个强势用户的需求，他们想要基于正则表达式搜索。这些马虎的家伙拥有数千个文件的超大目录，他们希望能做到像发现所有在题目中什么地方包含“oopsla”的“pdf”文件这样的事。为了支持他们，你写了这个函数：

```
def filesRegex(query: String) =
  for (file <- filesHere; if file.getName.matches(query))
    yield file
```

162

有经验的程序员会注意到所有的这些重复并想知道是否能从中提炼出通用的帮助函数。然而，显而易见的方式不起作用。你希望能做的是这样的：

```
def filesMatching(query: String, method) =
  for (file <- filesHere; if file.getName.method(query))
    yield file
```

这种方式在某些动态语言中能起作用，但 Scala 不允许在运行期这样粘合代码。那么该怎么做呢？

函数值提供了一个答案。虽然你不能把方法名当作值传递，但你可以通过传递为你调用方法的函数值达到同样的效果。在这个例子里，你可以给方法添加一个 matcher 参数，其唯一的目的就是针对查询检查文件名：

```
def filesMatching(query: String,
    matcher: (String, String) =>Boolean) = {

  for (file <- filesHere; if matcher(file.getName, query))
    yield file
}
```

方法的这个版本中，if 子句现在使用 matcher 针对查询检查文件名。更精确的说法是这个检查不依赖于 matcher 定义了什么。现在看一下 matcher 的类型。它是一个函数，因此类型中有个=>。这个函数带两个字符串参数——文件名和查询——并返回布尔值，因此这个函数的类型是(String, String) => Boolean。

有了这个新的 filesMatching 帮助方法，你可以通过让三个搜索方法调用它，并传入合适的函数来简化它们：

```
def filesEnding(query: String) =
  filesMatching(query, _.endsWith(_))

def filesContaining(query: String) =
  filesMatching(query, _.contains(_))

def filesRegex(query: String) =
  filesMatching(query, _.matches(_))
```

163

这个例子中展示的函数字面量使用了前一章中介绍的占位符语法，对你来说可能感觉不是非常自然。因此，以下阐明例子里是如何使用占位符的。用在 filesEnding 方法里的函数字面量 _.endsWith(_)，与下面的是一回事：

```
(fileName: String, query: String) => fileName.endsWith(query)
```

原因是 filesMatching 带一个函数，这个函数需要两个 String 参数，不过无须指定参数类型。因此，你也可以写成(fileName, query) => fileName.endsWith(query)。由于第一个参数 fileName 在方法体中被第一个使用，第二个参数 query 被第二个使用，你也可以使用占位符语法：_.endsWith(_)。第一个下划线是第一个参数文件名的占位符，第二个下划线是第二个参数查询字符串的占位符。

代码已经被简化了，但它实际还能更短。注意到 query 传递给了 filesMatching，但 filesMatching 没有用查询做任何事，只是把它传回给传入的 matcher 函数。这个传来传去的过程不是必需的，因为调用者在前面就已经知道了 query 的内容。你可以同样从 filesMatching 和 matcher 中简单地去除 query 参数，因此简化后的代码如清单 9.1 中所示。

164

```
object FileMatcher {
  private def filesHere = (new java.io.File(".")).listFiles

  private def filesMatching(matcher: String =>Boolean) =
    for (file <- filesHere; if matcher(file.getName))
      yield file

  def filesEnding(query: String) =
    filesMatching(_.endsWith(query))

  def filesContaining(query: String) =
    filesMatching(_.contains(query))

  def filesRegex(query: String) =
    filesMatching(_.matches(query))
}
```

清单 9.1 使用闭包减少代码重复

这个例子演示了函数作为头等值帮助你减少代码重复的方式，如果没有它们，这将变得很困难。比方说在 Java 里，你可以创建包括带一个 String 并返回 Boolean 的方法的接口，然后创建并传递实现这个接口的匿名内部类实例给 filesMatching。尽管这个方式能去除你尝试简化掉的代码重复，但同时它增加了许多乃至更多的新代码。因此好处就不值这个开销了，于是你或许也就安于重复代码的现状了。

再者，这个例子还演示了闭包是如何能帮助你减少代码重复的。前面一个例子里用到的函数字面量，如_.endsWith(_)和_.contains(_)，都是在运行期实例化成函数值而不是闭包，因为它们没有捕获任何自由变量。举例来说，表达式_.endsWith(_)里用的两个变量，都是用下划线代表的，也就是说它们都是从传递给函数的参数获得的。因此，_.endsWith(_)使用了两个绑定变量，而不是自由变量。相对地，最近的例子里面用到的函数字面量_.endsWith(query)包含一个绑定变量，下划线代表的参数，和一个名为 query 的自由变量。仅仅因为 Scala 支持闭包才使得你可以在最近的这个例子里从

filesMatching 中去掉 query 参数，从而更进一步简化了代码。

9.2 简化客户代码

前一个例子演示了高阶函数能在你实现 API 的同时，帮助减少代码重复。高阶函数的另一个重要应用是把它们放在 API 里使客户代码更简洁。Scala 的集合类型的特定用途循环方法提供了一个很好的例子[1]。很多已经在第 3 章的表 3.1 中列了出来。不过现在请注意其中的一个例子来看看为什么这些方法如此有用。

165

考虑 exists，一个判断传入的值是否包含在集合中的方法。当然你也可以初始化一个 var 为假，循环遍历集合类型，检查每个元素，并且如果你找到了要寻找的就把 var 设置为真，通过这样的方式寻找元素。以下是使用了这种方式的方法去判断是否传入的 List 包含了负数的例子：

```
def containsNeg(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num < 0 )
      exists = true
  exists
}
```

假如你在解释器里定义了这个方法，就可以这样调用：

```
scala> containsNeg(List(1, 2, 3, 4))
res0: Boolean = false

scala> containsNeg(List(1, 2, -3, 4))
res1: Boolean = true
```

不过更简洁的定义方式是对传入的 List 调用高阶函数 exists，如：

```
def containsNeg(nums: List[Int]) = nums.exists(_ < 0)
```

这个版本的 containsNeg 能产生和前面的那个一样的结果：

```
scala> containsNeg(Nil)
res2: Boolean = false

scala> containsNeg(List(0, -1, -2))
res3: Boolean = true
```

exists 方法代表了控制抽象。是 Scala 库提供的特定用途循环架构而不是像 while 或 for 那样内建在 Scala 语言里的。上节中，高阶函数 filesMatching 在对象 FileMatcher 的实现中减少了代码重复。exists 方法提供了类似的好处，但因为 exists 是公开在 Scala 的集合类型 API 里的，所以它减少的是 API 的客户代码中的重复。exists 不存在的话，如果你想要写一个 containsOdd 方法，检测列表是否包含了奇数，你或许会写成这样：

166

```
def containsOdd(nums: List[Int]): Boolean = {
  var exists = false
```

[1]注：这些特殊目的的循环方法定义在特质 Iterable 中，被 List、Set、Array、还有 Map 扩展。说明见第 17 章。

```
  for (num <- nums)
    if (num % 2 == 1)
      exists = true
  exists
}
```

若你比较了 containsNeg 和 containsOdd 的函数体，会发现除了 if 表达式之外，其他东西都是重复的。使用 exists，你就可以这么写：

```
def containsOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

这个版本的代码体再一次与相应的 containsNeg 方法的保持一致（使用了 exists 的版本），除了搜索的条件不同。然而代码重复的量却少得多，因为所有的循环架构都被提取成 exists 方法本身了。

Scala 的标准库中还有许多其他循环方法。如果你能发现使用它们的机会，那么像 exists 一样，它们经常能缩短你的代码。

9.3 柯里化（currying）

在第 1 章，我们说过 Scala 允许你创建新的“感觉像是原生语言支持”的控制抽象。尽管到目前你已经看到的例子都的确是控制抽象，不过没有人会误以为它们是原生语言支持的。为了搞明白如何让控制抽象感觉更像语言的扩展，你首先需要明白称为柯里化的函数式编程技巧。

柯里化的函数被应用于多个参数列表，而不是仅仅一个。清单 9.2 展示了一个规整的，未被柯里化的函数，它实现对两个 Int 型参数 x 和 y 做加法。

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (Int,Int)Int

scala> plainOldSum(1, 2)
res4: Int = 3
```

清单 9.2 定义和调用“陈旧的”函数

相对的，清单 93 展示了柯里化后的同一个函数。代之以一个列表的两个 Int 参数，你把这个函数应用于两个列表的各一个参数。

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (Int)(Int)Int

scala> curriedSum(1)(2)
res5: Int = 3
```

清单 9.3 定义和调用柯里化的函数

这里发生的事情是当你调用 curriedSum 时，实际上接连调用了两个传统函数。第一个函数调用带单个的名为 x 的 Int 参数，并返回第二个函数的函数值。第二个函数带 Int 参数 y。下面的名为 first 的函数实质上执行了 curriedSum 的第一个传统函数调用会做的事情：

```
scala> def first(x: Int) = (y: Int) => x + y
first: (Int)(Int) => Int
```

在第一个函数上应用1——换句话说，调用第一个函数并传入1——会产生第二个函数：

```
scala> val second = first(1)
second: (Int) => Int = <function>
```

在第二个函数上应用2产生结果：

168

```
scala> second(2)
res6: Int = 3
```

first和second函数只是柯里化过程的一个演示。他们并不直接连接在curriedSum函数上。尽管如此，仍然有一个方式获得实际指向curriedSum的“第二个”函数的参考。你可以用部分应用函数表达式方式，把占位符标注用在curriedSum里，如：

```
scala> val onePlus = curriedSum(1)_
onePlus: (Int) => Int = <function>
```

curriedSum(1)_里的下划线是第二个参数列表的占位符[2]。结果就是指向一个函数的参考，这个函数在被调用的时候，对它唯一的Int参数加1并返回结果：

```
scala> onePlus(2)
res7: Int = 3
```

然后以下是你如何获得对唯一的Int参数加2函数的方式：

```
scala> val twoPlus = curriedSum(2)_
twoPlus: (Int) => Int = <function>

scala> twoPlus(2)
res8: Int = 4
```

9.4 编写新的控制结构

在拥有头等函数的语言中，即使语言的语法是固定的，你也可以有效地制作新的控制结构。所有你需要做的就是创建带函数做参数的方法。

例如，下面是“双倍”控制结构，能够重复一个操作两次并返回结果：

```
scala> def twice(op: Double =>Double, x: Double) = op(op(x))
twice: ((Double) => Double,Double)Double
```

169

```
scala> twice(_ + 1, 5)
res9: Double = 7.0
```

这个例子里op的类型是Double => Double，就是说它是带一个Double做参数并返回另一个Double的函数。

任何时候，当你发现你的代码中多个地方有重复的控制模式时，就应该考虑把它实现为一个新的控制结构。本章早些时候你看到了filesMatching，一个极度特化了的控制模式。现在考虑一个更宽泛使用的代码模式：打开一个资源，对它进行操作，然后关闭资源。你可以使用如下的方法将其捕

[2]注：前一章里，当占位符标注用在传统方法上时，如println_，你必须在名称和下划线之间留一个空格。在这个例子里不需要，因为println_是Scala里合法的标识符，curriedSum(1)_不是。

获并放入控制抽象：

```
def withPrintWriter(file: File, op: PrintWriter => Unit) {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

有了这个方法，你就可以这样使用：

```
withPrintWriter(
  new File("date.txt"),
  writer => writer.println(new java.util.Date)
)
```

使用这个方法的好处是，由 withPrintWriter 而并非用户代码，确认文件在结尾被关闭。因此忘记关闭文件是不可能的。这个技巧被称为借贷模式（loan pattern），因为控制抽象函数，如 withPrintWriter，打开了资源并“贷出”给函数。例如，前面例子里的 withPrintWriter 把 PrintWriter 借给函数 op。当函数完成的时候，它发出信号说明它不再需要“借”的资源。于是资源被关闭在 finally 块中，以确信其确实被关闭，而忽略函数是正常结束返回还是抛出了异常。

让客户代码看上去更像内建控制结构的一种方式是使用花括号代替小括号包围参数列表。Scala 的任何方法调用，如果你确实只传入一个参数，就能可选地使用花括号替代小括号包围参数。

例如，代之以：

```
scala> println("Hello, world!")
Hello, world!
```

你可以写成：

```
scala> println { "Hello, world!" }
Hello, world!
```

在第二个例子里，你使用了花括号替代小括号包围 println 的参数。然而，这个花括号技巧仅在你传入一个参数时有效。下面是破坏这个规则的尝试：

```
scala> val g = "Hello, world!"
g: java.lang.String = Hello, world!

scala> g.substring { 7, 9 }
<console>:1: error: ';' expected but ',' found.
  g.substring { 7, 9 }
                   ^
```

因为你正打算把两个参数传入 substring，当你尝试用花括号包围这些参数的时候产生了错误。为了纠正错误，你需要使用小括号：

```
scala> g.substring(7, 9)
res12: java.lang.String = wo
```

在传入一个参数时，可以用花括号替代小括号的机制，其目的是让客户程序员能写出包围在花括号内的函数字面量。这可以让方法调用感觉更像控制抽象。以前面例子里定义的 withPrintWriter 方法举例。在它最近的形式里，withPrintWriter 带了两个参数，因此你不能使用花括号。虽然如此，因为传递给 withPrintWriter 的函数是列表的最后一个参数，你可以使用柯里化把第一个参数，File 拖入分离的参数列表。这将使函数仅剩下列表的第二个参数作为唯一的参数。清单 9.4 展示了你要怎样重新定义 withPrintWriter。

```
def withPrintWriter(file: File)(op: PrintWriter => Unit) {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

清单 9.4 使用借贷模式写文件

新版本不同于旧版本的地方仅在于用两个参数列表的各有一个参数替代了原来的一个参数列表里面的两个参数。仅比较这两个参数的差异。展示在第 109 页的 withPrintWriter 的前一个版本里，你看到了...,File, op.... 但在这个版本里，你看到了...File)(op.... 有了上述的定义，你就可以用更赏心悦目的语法格式调用这个方法：

```
val file = new File("date.txt")

withPrintWriter(file) {
  writer => writer.println(new java.util.Date)
}
```

在这个例子里，第一个参数列表包含了一个 File 参数，被写成包围在小括号中。第二个参数列表包含了一个函数参数，被包围在花括号中。

9.5 传名参数（by-name parameter）

上节展示的 withPrintWriter 方法不同于语言的内建控制结构，如 if 和 while，在于花括号之间的代码带了参数。withPrintWriter 方法需要一个类型为 PrintWriter 的参数。这个参数以“writer =>”方式显示出来：

```
withPrintWriter(file) {
  writer => writer.println(new java.util.Date)
}
```

然而如果你想要实现某些更像 if 或 while 的东西，根本没有值要传入花括号之间的代码，那该怎么做呢？为了适应这种情况，Scala 提供了传名参数。

为了举一个有现实意义的例子，请设想你需要实现一个称为 myAssert 的断言架构[3]。myAssert

[3]注：你只能称其为 myAssert，而不是 assert，因为 Scala 提供了它自己的 assert，将在 14.1 节描述。

函数将带一个函数值作为输入并参考一个标志位来决定该做什么。如果标志位被设置了，myAssert 将调用传入的函数并证实其返回了 true。如果标志位被关闭了，myAssert 将什么都不做。

如果没有传名参数，你可以这样写 myAssert：

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError
```

这个定义是正确的，但使用它会有点儿难看：

```
myAssert(() => 5 > 3)
```

你或许很想省略函数字面量里的空参数列表和=>符号，写成如下形式：

```
myAssert(5 > 3) // 不会有效，因为缺少() =>
```

传名函数恰好为了实现你的愿望而出现。要实现一个传名函数，要定义参数的类型开始于=>而不是() =>。例如，你可以通过改变其类型，“() => Boolean”，为“=> Boolean”，把 myAssert 的 predicate 参数改为传名参数。清单 9.5 展示了它的样子：

```
def byNameAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

清单 9.5 使用传名参数

现在你可以在需要断言的属性里省略空的参数了。使用 byNameAssert 的结果看上去就好象使用了内建控制结构：

```
byNameAssert(5 > 3)
```

传名类型中，空的参数列表，()，被省略，它仅在参数中被允许。没有什么传名变量或传名字段这样的东西。

现在，你或许想知道为什么你不能简化 myAssert 的编写，使用陈旧的 Boolean 作为它参数的类型，如：

```
def boolAssert(predicate: Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

当然这种格式同样合法，并且使用这个版本 boolAssert 的代码看上去仍然与前面的一样：

```
boolAssert(5 > 3)
```

虽然如此，这两种方式之间存在一个非常重要的差别须指出。因为 boolAssert 的参数类型是 Boolean，在 boolAssert(5 > 3)里括号中的表达式先于 boolAssert 的调用被评估。表达式 5 > 3 产生 true，被传给 boolAssert。相对的，因为 byNameAssert 的 predicate 参数的类型是=> Boolean，byNameAssert(5 > 3)里括号中的表达式不是先于 byNameAssert 的调用被评估的。而是

代之以先创建一个函数值，其 apply 方法将评估 5 > 3，而这个函数值将被传递给 byNameAssert。

因此这两种方式之间的差别，在于如果断言被禁用，你会看到 boolAssert 括号里的表达式的某些副作用，而 byNameAssert 却没有。例如，如果断言被禁用，boolAssert 的例子里尝试对“x / 0 == 0”的断言将产生一个异常：

```
scala> var assertionsEnabled = false
assertionsEnabled: Boolean = false

scala> boolAssert(x / 0 == 0)
java.lang.ArithmeticException: / by zero
         at .<init>(<console>:8)
        at .<clinit>(<console>)
        at RequestResult$.<init>(<console>:3)
        at RequestResult$.<clinit>(<console>)...
```

174

但在 byNameAssert 的例子里尝试同样代码的断言将不产生异常：

```
scala> byNameAssert(x / 0 == 0)
```

9.6 小结

本章展示给你如何基于 Scala 的丰富的函数支持建造控制抽象。你可以在你的代码中使用函数提取通用的控制模式，并且你可以利用 Scala 库里的高阶函数去复用所有程序源代码中都常见的控制模式。本章还展示了如何使用柯里化和传名参数以便你自己的高阶函数能以一种简洁的语法形式使用。

175

在前一章和这一章里，你已经看到了太多关于函数的信息。后续的章节中将回过头讨论语言中更加面向对象的特征。

第 10 章

组合与继承

第 6 章介绍了一些 Scala 面向对象的概貌。本章将继续第 6 章省略的部分并深入到 Scala 面向对象编程的更多细节中。我们将比较类之间的两种基本关系：组合与继承。组合指一个类持有另一个的引用，借助被引用的类完成任务。继承是超类/子类的关系。除此之外，还将讨论抽象类，无参数方法，扩展类，重写方法和字段，参数化字段，调用超类构造器，多态和动态绑定，final 成员和类，还有工厂对象和方法。

10.1 二维布局库

我们将建立一个创建和渲染二维布局元素的库作为本章运行的实例。每个元素将显示一个由文字填充的矩形。为方便起见，库将提供名为“elem”的工厂方法来通过传入的数据构造新的元素。例如，你将能通过工厂方法采用下面的写法创建带有字符串的元素：

```
elem(s: String): Element
```

正如你所见，元素将以名为 Element 的类型建模。你可以对元素调用 above 或 beside 方法，并传入第二个元素，来获取一个合并了前两个元素的新元素。例如，下面的表达式将构建一个包含两列，列高为 2 的大一些的元素：

```
val column1 = elem("hello") above elem("****")
val column2 = elem("****") above elem("world")
column1 beside column2
```

该表达式的显示结果为：

```
hello ***
 *** world
```

在通过简单部件及组合操作符的帮助下构建对象的系统中，布局元素是个不错的例子。本章里，我们将定义类使得元素对象能够由简单部件，包括数组，行，以及长方形进行构建。我们还将定义组合操作符 above 和 beside。这种组合操作符也经常被称为连结符（combinator），因为它们能把某些区域的元素组合成新的元素。

以连结符的方式思考问题通常是实现库的设计的好方法：可以让你只要考虑以基本的方式在应用领域中构建对象即可。什么是简单对象？用什么方式能让更多有趣的对象通过简单对象构造出来？连符是怎样结合在一起的？什么是最通用的组合？它们满足一些有趣的规律吗？如果对这些问题都有好的答案，那么你的库的设计就可以说是踏上正轨了。

10.2 抽象类

我们的第一项任务就是定义代表布局元素的类型 Element。因为元素是二维的字符矩形，所以顺理成章应包含一个成员 contents 指向布局元素的内容。内容可以由字符串数组表示，这里每个字符串代表一行。因此，contents 返回的结果类型为 Array[String]。如清单 10.1 所示。

```
abstract class Element {
  def contents: Array[String]
}
```

清单 10.1 定义抽象方法和类

178

这个类里，contents 被声明为没有实现的方法。换句话说，这个方法是 Element 类的抽象成员。具有抽象成员的类本身必须被声明为抽象的，只要在 class 关键字之前加上 abstract 修饰符即可：

```
abstract class Element ...
```

abstract 修饰符说明类可能有未实现的成员，因此不能实例化一个抽象类。如果尝试这么做，编译器会报错：

```
scala> new Element
<console>:5: error: class Element is abstract;
    cannot be instantiated
       new Element
           ^
```

本章后面你会看到如何创建 Element 类的子类，这些类由于包含了之前缺少的 contents 方法的定义，所以可以被实例化。

注意 Element 类的 contents 方法并没带 abstract 修饰符。一个方法只要没有实现（即没有等号或方法体），它就是抽象的。不像 Java，方法的声明中不需要（也不允许）有抽象修饰符。拥有实现的方法被称为具体的（concrete）方法。

另外需要分辨术语声明（declaration）和定义（definition）之间的区别。Element 类声明了抽象方法 contents，但当前没有定义具体方法。在下一节，我们将通过定义一些具体方法来加强 Element。

10.3 定义无参数方法

179

接下来，我们将向 Element 添加显示宽度和高度的方法，如清单 10.2 所示。height 方法返回 contents 里的行数。width 方法返回第一行的长度，如果元素没有行则返回零。（即，不能定义一个高度为零但宽度不为零的元素。）

```
abstract class Element {
  def contents: Array[String]
  def height: Int = contents.length
  def width: Int = if (height == 0) 0 else contents(0).length
}
```

清单 10.2 定义无参数方法 width 和 height

注意 Element 的三个方法全都没有参数列表，甚至连空列表都没有。例如，不同于方法：

```
def width(): Int
```

以下的方法定义中不含括号：

```
def width: Int
```

这种无参数方法在 Scala 里是非常普遍的。相对地，带有空括号的方法定义，如 def height(): Int，被称为空括号方法（empty-paren method）。推荐的惯例是无论何时，只要方法中没有参数**并且**方法仅能通过读取所包含对象的属性去访问可变状态（特指方法不能改变可变状态），就使用无参数方法。这个惯例支持统一访问原则（uniform access principle）[1]，就是说客户代码不应由属性是通过字段实现还是方法实现而受到影响。例如，我们可以选择把 width 和 height 作为字段而不是方法来实现，只要简单地在每个实现里把 def 修改成 val 即可：

```
abstract class Element {
  def contents: Array[String]
  val height = contents.length
  val width =
    if(height == 0) 0 else contents(0).length
}
```

两组定义从客户的观点来看是完全相同的。唯一的差别是访问字段比调用方法略快，因为字段值在类初始化的时候被预计算，而方法调用在每次调用的时候都要计算。另一方面，使用字段需要为每个 Element 对象分配更多的内存空间。因此一个属性是用字段表示更好还是用方法表示更好与类的使用情况有关，并且类的使用情况可能会随着时间变化。重要的是 Element 类的客户不应在其内部实现改变的时候受影响。

特别地，如果类的字段变为了访问方法，只要访问方法是纯的，即没有副作用且不依赖于可变状态，那么 Element 类的客户代码就不需重写。客户代码不应该关心到底是用哪种方式实现的。

到目前为止一切良好。但要协同 Java 处理事情的方式，仍然有些琐碎复杂的东西要做。问题在于 Java 没有实现统一访问原则。因此 Java 里是 string.length()，而不是 string.length（尽管是 array.length，而不是 array.length()）。不用说，这让人很困惑。

为了在这道鸿沟上架起一座桥梁，Scala 在遇到混合了无参数和空括号方法的情况时很自由。特别是，你可以用空括号方法重写无参数方法，并且反之亦可。你还可以在调用任何不带参数的方法时省略空的括号。例如，下面两行在 Scala 里都是合法的：

```
Array(1, 2, 3).toString
"abc".length
```

原则上，Scala 的函数调用中可以省略所有的空括号。然而，在调用的方法超出其调用者对象的属性时，推荐仍然写一对空的括号。例如，如果方法执行了 I/O，或写入可重新赋值的变量（var），或读取不是调用者字段的 var，总之无论是直接的还是非直接的使用可变对象，都应该添加空括号。这种方

[1]注：Meyer，“Object-Oriented Software Construction”。[Mey00]

式是让参数列表扮演一个可见的线索来说明调用触发了某些有趣的计算。例如：

```
"hello".length    // 没有副作用，所以无需()
println()         // 最好别省略()
```

总结起来，Scala 鼓励使用将不带参数且没有副作用的方法定义为无参数方法的风格，即省略空括号。但是永远不要定义没有括号的带副作用的方法，因为那样的话方法调用看上去会像在使用字段，所以你的客户会很奇怪为什么它带有副作用。据此，无论何时当调用有副作用的方法时应确保包含一对空括号。另一种考虑这个问题的方式是，如果你调用的函数执行了操作就使用括号，但如果仅提供了对某个属性的访问，那么省略括号。

181

10.4 扩展类

我们仍然需要创建新的元素对象。你已经看到了因为 Element 类是抽象的，所以“new Element”做不了这件事。因此，为了实例化一个元素，我们需要创建一个扩展了 Element 类并实现抽象的 contents 方法的子类。清单 10.3 是可能的方式之一：

```
class ArrayElement(conts: Array[String]) extends Element {
  def contents: Array[String] = conts
}
```

清单 10.3 定义 ArrayElement 为 Element 的子类

ArrayElement 类的定义扩展了 Element 类。与 Java 类似，在类名之后使用 extends 子句表示：

```
... extends Element ...
```

这样的 extends 子句有两个效果：使得 ArrayElement 类继承 Element 类的所有非私有的成员，并且让 ArrayElement 类型成为 Element 类型的子类型。因为 ArrayElement 扩展了 Element，所以 ArrayElement 类被称为 Element 类的子类。反过来，Element 是 ArrayElement 的超类。

如果你省略 extends 子句，Scala 编译器将隐式地假设你的类扩展自 scala.AnyRef，这与 Java 平台上的 java.lang.Object 相同。因此，Element 类隐式地扩展了 AnyRef 类。你可以在图 10.1 上看到这些继承关系。

182

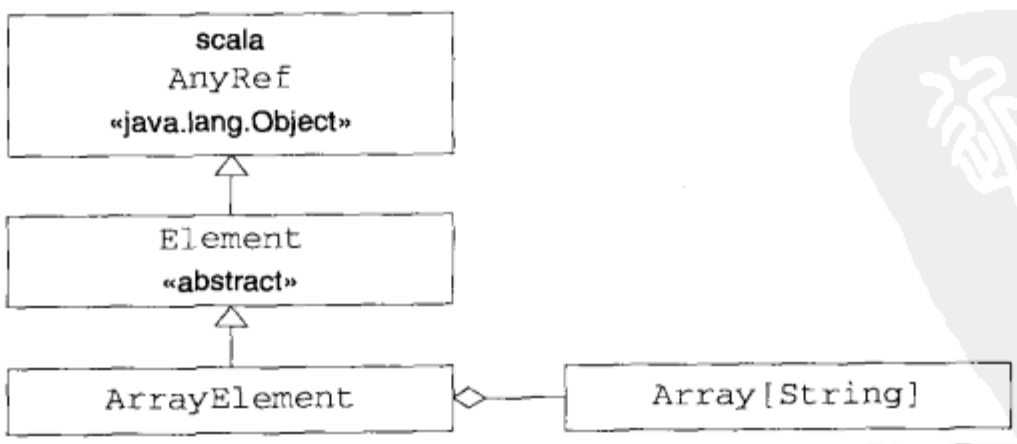


图 10.1 ArrayElement 的类关系图

继承（inheritance）表示超类的所有成员也是子类的成员，但以下两种情况例外。第一，超类的私有成员不会被子类继承。第二，超类中的成员若与子类中实现的成员具有相同名称和参数则不会被子类继承。这种情况被称为子类的成员重写（override）了超类的成员。如果子类中的成员是具体的而超类中的是抽象的，我们也可以说具体的成员实现（implement）了抽象的成员。

例如，ArrayElement 类的 contents 方法重写（或者说实现）了 Element 类的抽象方法 contents。[2]相应地，ArrayElement 类从 Element 类继承了 width 和 height 方法。例如，给定一个 ArrayElement 的对象 ae，可以使用 ae.width 查询其宽度，就如同 width 是定义在 ArrayElement 类中：

```
scala> val ae = new ArrayElement(Array("hello", "world"))
ae: ArrayElement = ArrayElement@d94e60

scala> ae.width
res1: Int = 5
```


子类型化（subtyping）是指子类的值可以在任何需要其超类的值的地方使用。例如：

```
val e: Element = new ArrayElement(Array("hello"))
```

变量 e 被定义为 Element 类型，所以其初始化的值也应当是 Element。但实际上初始化值的类型是 ArrayElement。可以这样写的原因是 ArrayElement 类扩展了 Element 类，所以 ArrayElement 类型与 Element 类型是兼容的。[3]

图 10.1 还显示出 ArrayElement 和 Array[String]之间的组合（composition）关系。这种关系之所以被称为组合是因为 ArrayElement 类是由 Array[String]“组合”出来的，Scala 编译器将在它生成的 ArrayElement 二进制类中设置一个字段来保留传入的 conts 数组的引用。我们将在本章后续内容中讨论一些关于组合和继承的设计理念，详见 10.11 节。

10.5 重写方法和字段

统一访问原则只是 Scala 在对待字段和方法上比 Java 更统一的一个方面。另一个差异是 Scala 里的字段和方法属于相同的命名空间。这让字段可以重写无参数方法。例如，你可以通过改变 ArrayElement 类中 contents 的实现将其从一个方法变为一个字段，而无需修改类 Element 中 contents 的抽象方法定义，如清单 10.4 所示：

```
class ArrayElement(conts: Array[String]) extends Element {
  val contents: Array[String] = conts
}
```

清单 10.4 用字段重写无参数方法

[2]注：这个设计的漏洞之一是因为返回的数组是可变的，所以客户端能改变它。本书中我们会让事情保持简单，但当 ArrayElement 是真实项目中的部分时，你也许应该考虑改为返回一个数组的防御性拷贝。另一个问题是我们现在不能保证 contents 数组中的每一个 String 元素都具有同样的长度。这可以通过在主构造器中检查前提条件来解决这个问题，一旦前提条件不成立则抛出异常。

[3]注：想了解更多子类和子类型之间的差异，请参考词汇表中的 subtype。

这个版本的 ArrayElement 中的 contents 字段（用 val 定义）为 Element 类中的无参数方法 contents（用 def 定义）提供了非常好的实现。

另一方面，Scala 里禁止在同一个类里用同样的名称定义字段和方法，尽管 Java 允许这样做。例如，下面的 Java 类可以顺利通过编译：

```
// 在 Java 里的代码
class CompilesFine {
  private int f = 0;
  public int f() {
    return 1;
  }
}
```

但是相应的 Scala 类将不能通过编译：

```
class WontCompile {
  private var f = 0       // 不能编译通过，因为字段和方法重名
  def f = 1
}
```

与 Java 为定义准备了四个命名空间（分别是字段，方法，类型和包）相对，一般来说，Scala 仅有两个命名空间：

- 值（字段，方法，包还有单例对象）。
- 类型（类和特质名）。

Scala 把字段和方法放进同一个命名空间的理由很明确，因为这样你就可以实现使用 val 重写无参数方法这种你在 Java 里做不到的事情。[4]

10.6 定义参数化字段

再次考虑上一节中的 ArrayElement 类的定义。它有一个参数 conts，其唯一作用是复制到 contents 字段中。选择 conts 这个参数的名称只是为了让它看上去更像字段名 contents 而不会与它发生实际冲突。这是一种“代码味道”，表明在代码中或许存在某些不必要的累赘和重复。

你可以通过在单一的参数化字段（parametric field）定义中组合参数和字段来避免这种代码味道，如清单 10.5 所示：

```
class ArrayElement(
  val contents: Array[String]
) extends Element
```

清单 10.5 定义 contents 为参数化字段

[4]注：Scala 里包与字段和方法共享相同命名空间是为了让你能够引入包，而不仅仅是引入类型名以及单例对象的字段和方法。这同样是你在 Java 中无法做到的。详见 13.2 节。

注意现在 contents 参数的前缀是 val。这是同时定义同名的参数和字段的一个简写方式。特别地，ArrayElement 类现在拥有了一个可以从类外部访问的（不能重新赋值的）字段 contents。字段使用参数值初始化。与下述写法类似，其中 x123 是任意未曾用过的参数名称：

```
class ArrayElement(x123: Array[String]) extends Element {
  val contents: Array[String] = x123
}
```

类的参数同样也可以使用 var 做前缀，这种情况下相应的字段可以被重新赋值。最后，与其他类成员相同，这些参数化字段还可以添加如 private，protected，[5]或 override 这类的修饰符。例如，对于下列类定义：

```
class Cat {
  val dangerous = false
}
class Tiger(
  override val dangerous: Boolean,
  private var age: Int
) extends Cat
```

Tiger的定义是下述另一种类定义方式的简写，其中包含了重写成员dangerous和私有成员age：

```
class Tiger(param1: Boolean, param2: Int) extends Cat {
  override val dangerous = param1
  private var age = param2
}
```

两个成员由各自对应的参数初始化。我们随意选择了参数名 param1 和 param2，重点是它们不会与作用域内的任何其他名称冲突。

10.7 调用超类构造器

现在你有了由两个类组成的完整系统：抽象类 Element 和扩展它的具体类 ArrayElement。或许你还在预想其他表示元素的方式。比方说，客户可能想要创造由给定的单行字符串构成的布局元素。面向对象编程让使用新的数据类型来扩展系统变得容易。加入子类即可。例如，清单 10.6 说明了扩展 ArrayElement 的 LineElement 类：

```
class LineElement(s: String) extends ArrayElement(Array(s)) {
  override def width = s.length
  override def height = 1
}
```

清单 10.6 调用超类构造器

由于 LineElement 扩展了 ArrayElement，并且 ArrayElement 的构造器带一个参数（Array[String]），因此 LineElement 需要传给超类的主构造器一个参数。要调用超类构造器，只要简单地把要传递的参数或参数列表放在超类名之后的括号里即可。例如，LineElement 类传递参

[5]注：protected 修饰符，可以授权给子类访问，将在第 13 章详细描述。

数 Array(s)到 ArrayElement 的主构造器，把它放在超类 ArrayElement 的类名后面的括号里：

187

```
... extends ArrayElement(Array(s)) ...
```

包含新子类的布局元素的继承层级如图 10.2 中所示。

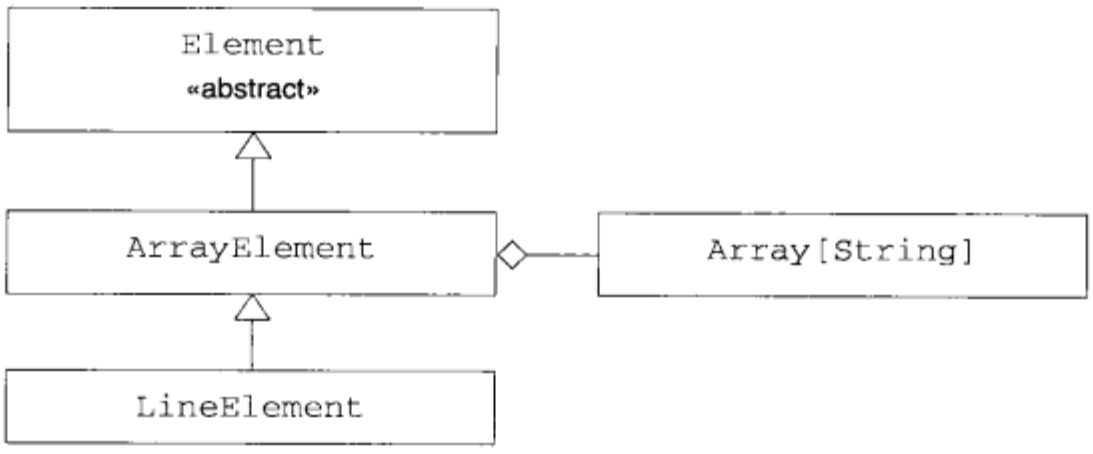


图 10.2 LineElement 的类关系图

10.8 使用 override 修饰符

注意 LineElement 里的 width 和 height 的定义都带着 override 修饰符，6.3 节 toString 方法的定义中我们也看到过这一修饰符。Scala 要求，若子类成员所有重写了父类的具体成员则必须带有这个修饰符；若成员实现的是同名的抽象成员时，则这个修饰符是可选的；若成员并未重写或实现什么其他基类（译注：通常我们称为超类）里的成员则禁用这个修饰符。由于 LineElement 类的 height 和 width 重写了 Element 类的具体成员定义，因此需要 override 修饰符。

这条规则给编译器提供了有用的信息来帮助避免某些难以捕捉的错误并使得对系统的改进更加安全。例如，如果你碰巧拼错了方法名或偶尔传递给它不同的参数列表，编译器会返回报错信息：

```
$ scalac LineElement.scala
.../LineElement.scala:50:
error: method hight overrides nothing
  override def hight = 1
               ^
```

188

改进系统的时候，使用 override 的习惯显得更重要。假设你定义了一个 2D 画图方法库。把它公开，使它得以被广泛使用。在下个版本的库中，你想为基类 Shape 增加一个带签名的新方法：

```
def hidden(): Boolean
```

你的新方法将被许多画图方法调用以确定是否需要画出形状，这会产生显著的提速。但与此同时你不得不冒破坏客户代码的风险。毕竟客户可能已经定义了使用不同的 hidden 实现的 Shape 的子类。或许客户的方法实际上是让对象消失而不是检测对象是否隐藏。因为这两个版本的 hidden 互相重写，画图的方法可能最终将使对象消失，这肯定不是你想要的！这些“意外的重写”就是被称为“脆基类”问题的最通常的表现。即如果你在类的层次结构中，为基类添加了新的成员，你会有破坏客户代码的风险。

Scala 不能完全解决脆基类问题，不过它相比 Java 有所改善。[6]如果画图库和它的客户代码是用 Scala 写的，那么客户 hidden 方法的原始实现就不会有 override 修饰符，因为这时候还没有其他的同名方法。一旦你在第二版中为 Shape 类添加了 hidden 方法，那么再次编译客户代码时将给出下列错误：

```
.../Shapes.scala:6: error: error overriding method
    hidden in class Shape of type ()Boolean;
method hidden needs 'override' modifier
def hidden(): Boolean =
^
```

也就是说，通过提供给客户将编译期错误取代错误的执行结果，这常常是更可取的。

189

10.9 多态和动态绑定

在 10.4 节中你看到了 Element 类型的变量可以指向 ArrayElement 类型的对象。这种现象叫做多态（polymorphism），意即“许多形状”或“许多形式”。这种情况下，Element 对象可以有许多形式[7]。目前为止，你已经看到了两种形式：ArrayElement 和 LineElement。你可以通过定义新的 Element 子类创造 Element 的更多形式。例如，下面给出了如何定义拥有给定长度和高度并充满指定字符的新的 Element 形式：

```
class UniformElement(
  ch: Char,
  override val width: Int,
  override val height: Int
) extends Element {
  private val line = ch.toString * width
  def contents = Array.make(height, line)
}
```

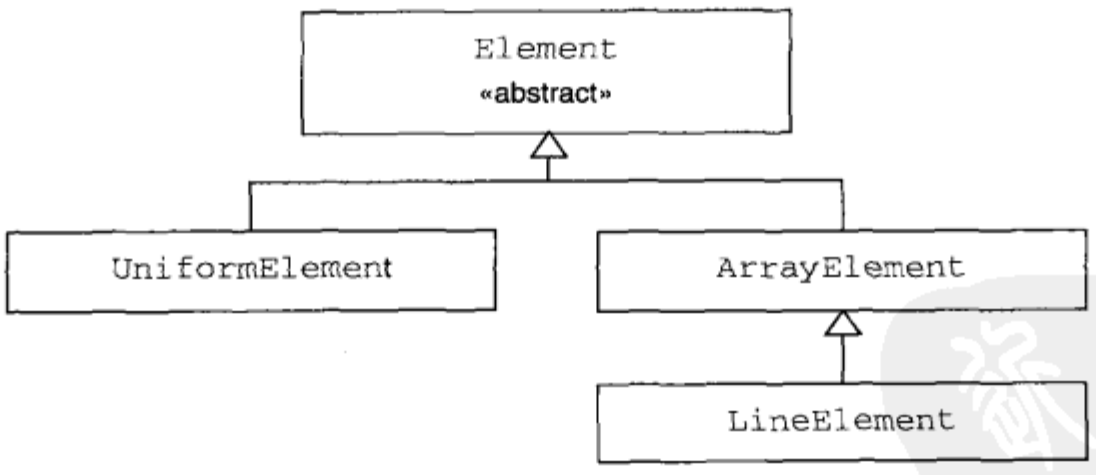


图 10.3 布局元素的类层级

现在 Element 类的继承层级如图 10.3 所示。结果是 Scala 将接受所有下列赋值，因为赋值表达式

[6]注：Java1.5 中引入的@Override 标注与 Scala 的 override 修饰符效果相似，不过不像 Scala 的 override，它不是必需的。

[7]注：这种类型的多态被称为子类型化多态：subtyping polymorphism。Scala 里另一种类型的多态，称为统一多态：universal polymorphism，详情将在第 19 章介绍。

的类型符合定义的变量类型：

```
val e1: Element = new ArrayElement(Array("hello", "world"))
val ae: ArrayElement = new LineElement("hello")
val e2: Element = ae
val e3: Element = new UniformElement('x', 2, 3)
```

检查继承层级会发现以上每一个 val 的定义里，等号右侧表达式的类型都在等号左侧被初始化的 val 类型之下。

然而，另一方面，变量和表达式上的方法调用是动态绑定（dynamically bound）的。这意味着被调用的实际方法实现取决于运行期对象基于的类，而不是变量或表达式的类型。为了证明这种行为，我们会从我们的 Element 类中临时移除所有存在的成员并添加一个名为 demo 的方法。我们会在 ArrayElement 和 LineElement 中重写 demo，但 UniformElement 除外：

190

```
abstract class Element {
  def demo() {
    println("Element's implementation invoked")
  }
}
class ArrayElement extends Element {
  override def demo() {
    println("ArrayElement's implementation invoked")
  }
}
class LineElement extends ArrayElement {
  override def demo() {
    println("LineElement's implementation invoked")
  }
}
// 继承了 Element 的 demo 方法
class UniformElement extends Element
```

把这些代码输入解释器，你就可以定义参数带 Element 的方法并在其中调用它的 demo 方法：

191

```
def invokeDemo(e: Element) {
  e.demo()
}
```

如果传递 ArrayElement 给 invokeDemo，会看到一条消息指明调用了 ArrayElement 中的 demo 实现，尽管调用的是变量 e 中的 demo，而变量 e 的类型是 Element：

```
scala> invokeDemo(new ArrayElement)
ArrayElement's implementation invoked
```

类似地，如果传递 LineElement 给 invokeDemo，也会看到一条指明调用了 LineElement 中的 demo 实现的消息：

```
scala> invokeDemo(new LineElement)
LineElement's implementation invoked
```

乍一看，传递 UniformElement 的执行结果似乎有些奇怪，但它是正确的：

```
scala> invokeDemo(new UniformElement)
Element's implementation invoked
```

因为在 UniformElement 中没有重写 demo，所以它继承了超类 Element 中 demo 的实现。因此，当对象是 UniformElement 类时，Element 的实现就是应该被调用的 demo 的正确实现。

10.10 定义 final 成员

有时在设计一个继承层级时，你想要确保一个成员不被子类重写。这在 Scala 中可以和 Java 一样通过给成员添加 final 修饰符来实现。例如，你可以在 ArrayElement 的 demo 方法前放置一个 final 修饰符，如清单 10.7 所示。

```
class ArrayElement extends Element {
  final override def demo() {
    println("ArrayElement's implementation invoked")
  }
}
```

清单 10.7 声明 final 方法

在这个版本的 ArrayElement 中，尝试重写子类 LineElement 的 demo 方法将无法通过编译：

```
elem.scala:18: error: error overriding method demo
  in class ArrayElement of type ()Unit;
method demo cannot override final member
    override def demo() {
                 ^
```

你有时或许还想确保整个类都不会有子类。要做到这点只要简单地在类的声明上添加 final 修饰符来把整个类声明为 final 即可。清单 10.8 说明了如何将 ArrayElement 声明为 final 的：

```
final class ArrayElement extends Element {
  override def demo() {
    println("ArrayElement's implementation invoked")
  }
}
```

清单 10.8 声明 final 类

在这个版本的 ArrayElement 中，任何定义子类的尝试都将失败：

```
elem.scala: 18: error: illegal inheritance from final class
    ArrayElement
  class LineElement extends ArrayElement {
                            ^
```

现在我们去掉 final 修饰符和 demo 方法，并回归之前实现的 Element 家族。本章的剩余部分将把注意力集中在完成布局库的可用版本上。

10.11 使用组合与继承

组合与继承是利用其他现存类定义新类的两个方法。如果你追求的是根本上的代码重用，那么通常更推荐采用组合而不是继承。只有继承才受累于脆基类问题，因为你可能会在更改超类时无意中破坏子类。

你可以问自己一个关于继承关系的问题，它是否建模了一种“是个”（is-a）的关系。[8]例如，说 ArrayElement 是一个（is-an）Element 是合理的。另一个问题则是，客户是否想要把子类类型当作超类类型来用。[9]在 ArrayElement 的例子里，我们的确期望客户想要把 ArrayElement 当作 Element 使用。

如果你对图 10.3 的继承关系问了这些问题，那么是否感觉到其中的一些关系有点可疑？尤其是，你认为 LineElement 明显是一个（is-an）ArrayElement 吗？你是否认为客户会把 LineElement 当作 ArrayElement 使用？实际上，我们把 LineElement 定义为 ArrayElement 主要是希望重用 ArrayElement 的 contents 定义。因此或许把 LineElement 定义为 Element 的直接子类会更好一些，就像这样：

```
class LineElement(s: String) extends Element {
  val contents = Array(s)
  override def width = s.length
  override def height = 1
}
```

在前一个版本中，LineElement 与 ArrayElement 间是继承关系，LineElement 从 ArrayElement 那里继承了 contents。现在它与 Array 是组合关系：在它自己的 contents 字段中持有一个字符串数组的引用。[10]依据 LineElement 的这个实现，Element 的继承层级现在如图 10.4 所示。

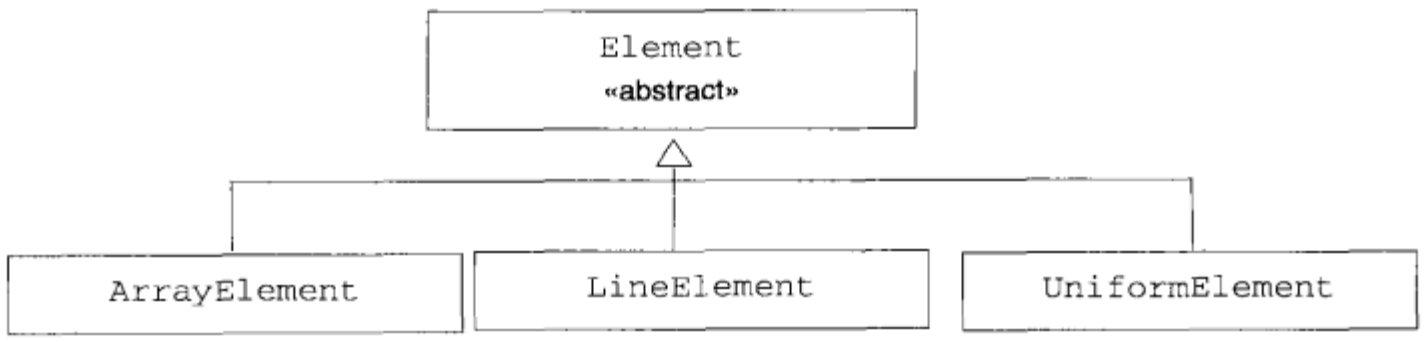


图 10.4 修改了 LineElement 后的类层级

10.12 实现 above、beside 和 toString

接下来一步，我们将在 Element 类中实现 above 方法。把一个元素放在另一个上面是指连接这两个元素的 contents 值。因此 above 方法的简单实现可以这样：

[8]注：Meyers，“Effective C++”。[Mey91]

[9]注：Eckel，“Thinking in Java”。[Eck98]

[10]注：类 ArrayElement 也与 Array 有组合关系，因为它的参数化字段 contents 持有字符串数组的引用。ArrayElement 的代码展示在清单 10.5 中。其组合关系用一个菱形表现在类图中，如图 10.1 所示。

```
def above(that: Element): Element =
  new ArrayElement(this.contents ++ that.contents)
```

++操作符连接两个数组。Scala 中的数组用 Java 数组表示，但是支持的方法更多。具体地说，Scala 中的数组继承自 scala.Seq 类，因此能够表示类似序列的结构并包含了许多访问和转换序列的方法。本章还会讲解一些其它数组的方法，在第 17 章将进行更加详尽的讨论。

实际上，前面展示的代码并不够用，因为它不允许你把不同长度的元素堆叠在一起。但在本节中为简单起见，我们先不管这个问题，只把长度相同的元素传给 above。在 10.14 节里，我们会改进 above，这样客户就能用它组合不同长度的元素了。

下一个要实现的方法是 beside。把两个元素靠在一起后，我们将创造一个新的元素，新元素的每一行都来自两个原始元素的相应行的串连。如前所述，为简单起见，我们先假设两个元素高度相同。这样就可以如下设计 beside 方法：

```
def beside(that: Element): Element = {
  val contents = new Array[String](this.contents.length)
  for(i <- 0 until this.contents.length)
    contents(i) = this.contents(i) + that.contents(i)
  new ArrayElement(contents)
}
```

beside 方法首先分配了一个新数组 contents，并以连接 this.contents 和 that.contents 中相应的数组元素来填充。这样最终产生了包含了新的 contents 的 ArrayElement。

尽管 beside 的这个实现可以工作，但它是指令式风格的，索引数组的循环可以说明问题。这个方法可以被缩减成一个表达式：

```
new ArrayElement(
  for(
    (line1, line2) <- this.contents zip that.contents
  ) yield line1 + line2
)
```

这里，this.contents 和 that.contents 两个数组用 zip 操作符转换为一个二元对的数组（称为 Tuple2）。zip 方法从它的两个参数中取出相应的元素并生成二元对数组。例如，表达式：

```
Array(1, 2, 3) zip Array("a", "b")
```

将生成：

```
Array((1, "a"), (2, "b"))
```

如果两个数组中的一个比另一个长，zip 将舍弃多余的元素。在上面的表达式中，左操作元的第三个元素 3 不在生成的结果中，因为它在右操作元中没有相对的元素。

然后 zip 运算的结果数组在 for 表达式被遍历。这里，表达式“for ((line1, line2) <- ...”允许你在一个模式（pattern）中命名二元对的两个元素，也就是说，line1 现在代表二元对的第一个元素，line2 代表第二个。Scala 的模式匹配系统将在第 15 章详细说明。现在，你可以考虑在迭代的每一步中定义两个 val，line1 和 line2。

for 表达式的 yield 部分可以产生结果。结果与枚举遍历的表达式类型一致，也就是说，是一个数组。数组的每个元素都是相应行，即 line1 和 line2 串联的结果。因此这段代码最终的结果与前一个版本的 beside 一样，不过它避免了显式的数组索引，因此出错的机会更少。

我们还需要一个显示元素的方式。可以和之前一样通过定义 toString 方法返回元素格式化成的字符串。下面是它的定义：

```
override def toString = contents mkString "\n"
```

toString 的实现使用了 mkString，mkString 定义在包括数组在内的所有序列中。正如你在 7.8 节中看到的，像“arr mkString sep”这样的表达式能返回由数组 arr 所有元素组成的字符串。每个元素都通过调用 toString 方法被映射为字符串。分隔符字符串 seq 被插入连续的元素字符串中间。因此表达式“contents mkstring "\n"”将 contents 数组格式化为字符串，其中每个数组元素占据一行。

注意 toString 没有带空参数列表。这遵循了统一访问原则的建议，因为 toString 是一个不带任何参数的纯方法。

加入这三个方法后，Element 类如清单 10.9 所示。

```
abstract class Element {

  def contents: Array[String]

  def width: Int =
    if(height == 0) 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
    new ArrayElement(this.contents ++ that.contents)

  def beside(that: Element): Element =
    new ArrayElement(
      for(
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )

  override def toString = contents mkString "\n"
}
```

清单 10.9 带有 above、beside 和 toString 的类 Element

10.13 定义工厂对象

你现在有了布局元素的类层级。这个层级可以“依原样”展现给你的客户。但是你也可能选择把层级隐藏在工厂对象之后。工厂对象包含了构建其他对象的方法。客户将使用这些工厂方法构造对象而不是直接使用 new 构造对象。这种方式的一个好处是可以将对象的创建集中化并且隐藏对象实际代表的类的细节。这种隐藏一方面可以让客户更容易理解你的库，因为暴露的细节更少了，另一方面给了你更多的机会可以在不破坏客户代码的前提下改变库的实现。

为布局元素构建工厂方法的首要任务是选择工厂方法应该放在何处。它们应该是单例对象成员还是

类成员？应该怎么调用包含它们的对象或类？这里有许多种可能性。一个直接的方法是创建 Element 类的伴生对象并把它作为布局元素的工厂方法。使用这种方式，你唯一要暴露给客户的就是 Element 的类/对象的组合，隐藏 ArrayElement，LineElement 和 UniformElement 三个类的实现。

清单 10.10 是遵循这个方案的设计。Element 伴生对象包含了三个重载的 elem 方法变体。每一个变体构建一种不同的布局对象。

```
object Element {

  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)

  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

  def elem(line: String): Element =
    new LineElement(line)
}
```

清单 10.10 带有工厂方法的工厂对象

198

这些工厂方法使得改变 Element 类的实现通过使用 elem 工厂方法而非显式地创建新的 ArrayElement 实例成为可能。为了直接调用工厂方法而不必使用单例对象的名称 Element 做限定，我们将在源文件的开始引用 Element.elem。换句话说，代之以在 Element 类内部以 Element.elem 方式调用工厂方法，我们将引入 Element.elem，这样只要使用简化的 elem 就可以调用工厂方法。清单 10.11 展示了如上更改后的 Element 类。

```
import Element.elem

abstract class Element {

  def contents: Array[String]

  def width: Int =
    if(height == 0) 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
    elem(this.contents ++ that.contents)

  def beside(that: Element): Element =
    elem(
      for(
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )

  override def toString = contents mkString "\n"
}
```

清单 10.11 重构以使用工厂方法的类 Element

而且，有了工厂方法之后，子类 ArrayElement，LineElement 和 UniformElement 现在可以是私有的，因为它们不再需要直接被客户访问。在 Scala 中，你可以在类和单例对象的内部定义其他的类和单例对象。因此一种让 Element 的子类私有化的方式就是把它们放在 Element 单例对象中并在

那里声明它们为私有。需要的时候，这些类将仍然能被三个 elem 工厂方法访问。清单 10.12 展示了其中的细节。

```
object Element {

  private class ArrayElement(
    val contents: Array[String]
  ) extends Element

  private class LineElement(s: String) extends Element {
    val contents = Array(s)
    override def width = s.length
    override def height = 1
  }

  private class UniformElement(
    ch: Char,
    override val width: Int,
    override val height: Int
  ) extends Element {
    private val line = ch.toString * width
    def contents = Array.make(height, line)
  }

  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)

  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

  def elem(line: String): Element =
    new LineElement(line)
}
```

清单 10.12 用私有类隐藏实现

10.14 变高变宽

我们现在需要最后一个改进。清单 10.11 中的 Element 的版本并不完善，因为它不允许客户把不同宽度的元素堆叠在一起，或者把不同高度的元素靠在一起。比方说，由于组合后的元素第二行比第一行长，下面的表达式将不能正常工作：

```
new ArrayElement(Array("hello")) above
new ArrayElement(Array("world!"))
```

与之类似，下面的表达式也不能正常工作，因为第一个 ArrayElement 高度为 2，而第二个的高度只有 1：

```
new ArrayElement(Array("one", "two")) beside
new ArrayElement(Array("one"))
```

清单 10.13 展示了一个私有助手方法 widen，能够以指定宽度为参数并返回该宽度的 Element。返回结果包括这个 Element 的居中显示的内容和为满足宽度要求在左侧和右侧留的空格。清单 10.13 还展示了一个类似的方法 heighten，能在竖直方向执行同样的功能。widen 方法被 above 调用以确保

Element 堆叠在一起后有同样的宽度。类似的，heighten 方法被 beside 调用以确保靠在一起的元素具有同样的高度。有了这些改变后，就可以准备使用布局库了。

200

```
import Element.elem

abstract class Element {
  def contents: Array[String]

  def width: Int = contents(0).length
  def height: Int = contents.length

  def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    elem(this1.contents ++ that1.contents)
  }

  def beside(that: Element): Element = {
    val this1 = this heighten that.height
    val that1 = that heighten this.height
    elem(
      for((line1, line2) <- this1.contents zip that1.contents)
      yield line1 + line2
    )
  }

  def widen(w: Int): Element =
    if(w <= width) this
    else{
      val left = elem(' ', (w - width) / 2, height)
      var right = elem(' ', w - width - left.width, height)
      left beside this beside right
    }

  def heighten(h: Int): Element =
    if(h <= height) this
    else {
      val top = elem(' ', width, (h - height) / 2)
      var bot = elem(' ', width, h - height - top.height)
      top above this above bot
    }

  override def toString = contents mkString "\n"
}
```

202

清单 10.13 有了 widen 和 heighten 方法的 Element

10.15 把代码都放在一起

能用到布局库的几乎所有元素的有趣方法就是写一个能画出给定数量边界的螺旋（Spiral）的程序。这个 Spiral 程序如清单 10.14 所示：

```
import Element.elem

object Spiral {

  val space = elem(" ")
```

```
  val corner = elem("+")

  def spiral(nEdges: Int, direction: Int): Element = {
    if (nEdges == 1)
      elem("+")
    else{
      val sp = spiral(nEdges - 1, (direction + 3) % 4)
      def verticalBar = elem('|', 1, sp.height)
      def horizontalBar = elem('-', sp.width, 1)
      if (direction == 0)
         (corner beside horizontalBar) above (sp beside space)
      else if (direction == 1)
        (sp above space) beside (corner above verticalBar)
      else if (direction == 2)
        (space beside sp) above (horizontalBar beside corner)
      else
        (verticalBar above corner) beside (space above sp)
    }
  }

  def main(args: Array[String]) {
    val nSides = args(0).toInt
    println(spiral(nSides, 0))
  }
}
```

清单 10.14 Spiral 程序

203

因为 Spiral 是个带有合适签名的 main 方法的独立对象，所以它是个 Scala 程序。Spiral 需要一个整数作为命令行参数，并且以特定数量的边界画一个螺旋。例如，可以像展示在下面的左边那样画一个六条边界的螺旋，或者右边的那种更大的螺旋：

```
$ scala Spiral 6
+-----
|
| +-+
| + |
|   |
+---+
```

```
$ scala Spiral 11
+----------
|
| +------+
| |      |
| | +--+ |
| | |  | |
| | ++ | |
| |    | |
| +----+ |
|        |
+--------+
```

```
$ scala Spiral 17
+----------------
|
| +------------+
| |            |
| | +--------+ |
| | |        | |
| | | +----+ | |
| | | |    | | |
| | | | ++ | | |
| | | |  | | | |
| | | +--+ | | |
| | |      | | |
| | +------+ | |
| |          | |
| +----------+ |
|              |
+--------------+
```

10.16 小结

本章中，你看到了更多 Scala 中与面向对象编程有关的概念。其中，你遇到了抽象类、继承和子类化、类层级、参数化字段及方法重写。你应当已经建立了在 Scala 里构造不太小的类层级的感觉。我们会在第 14 章重新介绍布局库。

204

第 11 章

Scala 的层级

你已经在前一章看过了类继承的细节，是时候退回一步整体看看 Scala 的类层级了。Scala 里，每个类都继承自通用的名为 Any 的超类。因为所有的类都是 Any 的子类，所以定义在 Any 中的方法就是“共同的”方法：它们可以被任何对象调用。Scala 还在层级的底端定义了一些有趣的类，如 Null 和 Nothing，扮演通用的子类。例如，如同 Any 是所有其他类的超类，Nothing 是所有其他类的子类。本章中，我们将带你一览 Scala 的类层级。

11.1 Scala 的类层级

图 11.1（见下一页）展示了 Scala 的类层级的大纲。层级的顶端是 Any 类，定义了下列的方法：

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def hashCode: Int
def toString: String
```

因为每个类都继承自 Any，所以 Scala 程序里的每个对象都能用==、!=或 equals 比较，用 hashCode 做散列（hash），以及用 toString 格式化。Any 类里的等号和不等号方法、==和!=，被声明为 final，因此它们不能在子类里重写。实际上，==总是与 equals 相同，!=总是与 equals 相反。因此独立的类可以通过重写 equals 方法改变==或!=的意义。我们会在本章后面展示一个例子。

根类 Any 有两个子类：AnyVal 和 AnyRef。AnyVal 是 Scala 里每个内建值类的父类。有 9 个这样的值类：Byte、Short、Char、Int、Long、Float、Double、Boolean 和 Unit。其中的前 8 个都对应到 Java 的基本类型，它们的值在运行时表示成 Java 的基本类型的值。Scala 里这些类的实例都写成字面量。例如，42 是 Int 的实例，'x'是 Char 的实例，false 是 Boolean 的实例。你不能使用 new 创造这些类的实例。这一点由一个“小技巧（trick）”保证，值类都被定义为既是抽象的又是 final 的。因此如果你写了：

```
scala> new Int
```

就会得到：

```
<console>:5: error: class Int is abstract; cannot be
instantiated
       new Int
       ^
```

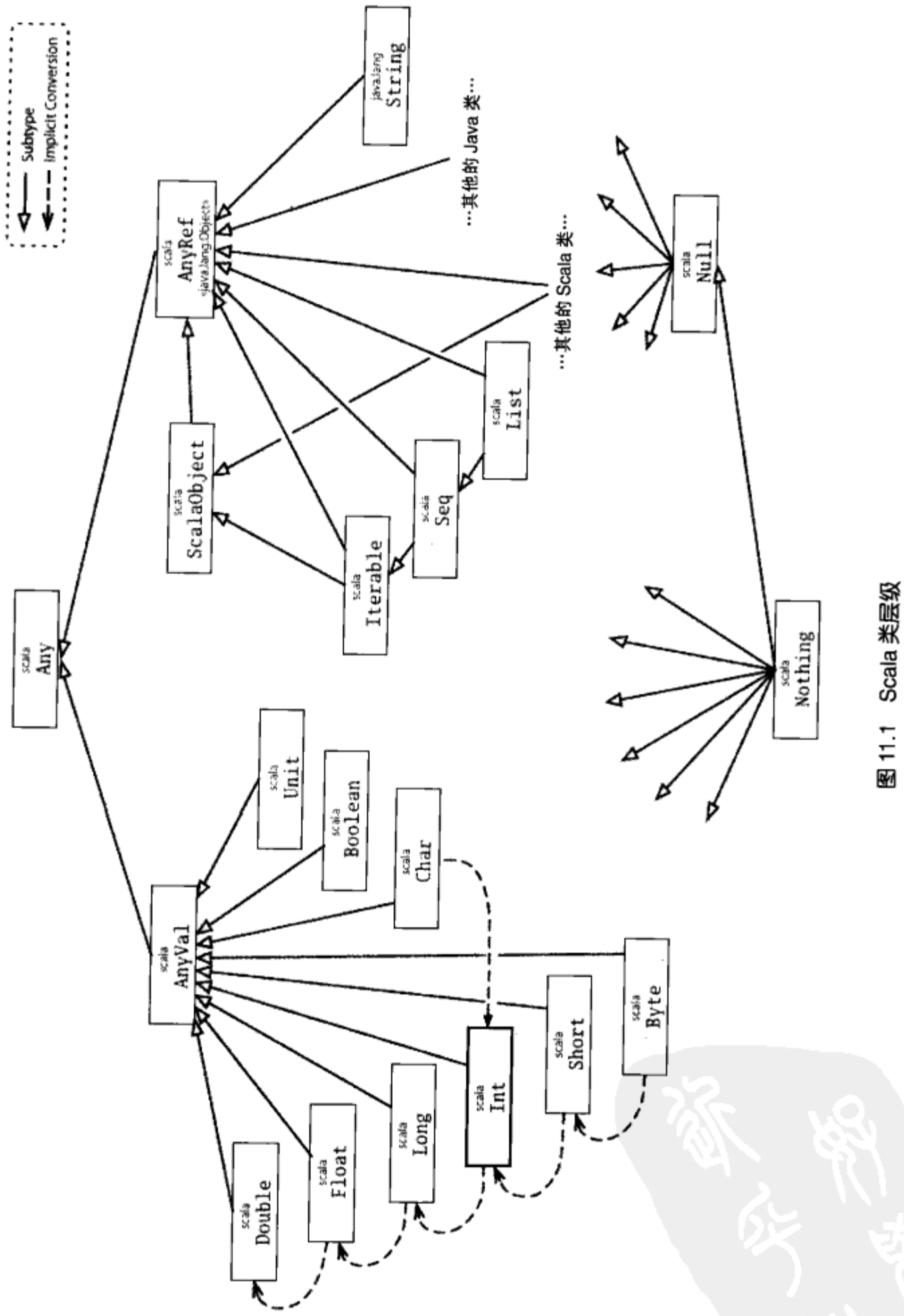


图 11.1 Scala 类层级

另一个值类，Unit，大约对应于 Java 的 void 类型；被用作不返回任何有趣结果的方法的结果类型。Unit 只有一个实例值，写成()，这在 7.2 节中讨论过。

正如第五章解释过的，值类以方法的形式支持通用的数学和布尔操作符。例如，Int 有名为+和*的方法，Boolean 有名为||和&&的方法。值类也从类 Any 继承所有的方法。你可以在解释器里测试：

```
scala> 42.toString
res1: java.lang.String = 42

scala> 42.hashCode
res2: Int = 42

scala> 42 equals 42
res3: Boolean = true
```


注意，值类的空间是平坦的；所有的值类都是 scala.AnyVal 的子类型，但是它们不是其他类的子类。但是不同的值类类型之间可以隐式地互相转换。例如，需要的时候，类 scala.Int 的实例可以自动放宽（通过隐式转换）到类 scala.Long 的实例。

正如 5.9 节中提到过的，隐式转换还用来为值类型添加更多的功能。例如，类型 Int 支持以下所有的操作：

```
scala> 42 max 43
res4: Int = 43

scala> 42 min 43
res5: Int = 42

scala> 1 until 5
res6: Range = Range(1, 2, 3, 4)

scala> 1 to 5
res7: Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 3.abs
res8: Int = 3

scala> (-3).abs
res9: Int = 3
```

这里解释其工作原理：方法 min、max、until、to 和 abs 都定义在类 scala.runtime.RichInt 里，并且有一个从类 Int 到 RichInt 的隐式转换。当你在 Int 上调用的方法没有定义在 Int 中，但定义在 RichInt 中时，就应用这个转换。类似的“支持类”和隐式转换也存在于其他的值类。我们将在第 21 章具体讨论隐式转换。

类 Any 的另一个子类是类 AnyRef。这个是 Scala 里所有引用类（reference class）的基类。正如前面提到的，在 Java 平台上 AnyRef 实际就是类 java.lang.Object 的别名。因此 Java 里写的类和 Scala 里写的都继承自 AnyRef[1]。你可以认为 java.lang.Object 是 Java 平台上实现 AnyRef 的方式。因此，尽管在 Java 平台上的 Scala 程序里 Object 和 AnyRef 的使用是可交换的，推荐的风格是在任何地方都只使用 AnyRef。

[1]注：使用 AnyRef 别名代替 java.lang.Object 名称的理由是 Scala 被设计成可以同时工作在 Java 和.Net 平台上。在.NET 平台上，AnyRef 是 System.Object 的别名。

Scala 类与 Java 类的不同在于它们还继承自一个名为 ScalaObject 的特别的记号特质[2]。是想要通过 ScalaObject 包含的 Scala 编译器定义和实现的方法让 Scala 程序的执行更高效。到现在为止，Scala object 只包含了一个方法，名为$tag，在内部使用以加速模式匹配。

11.2 原始类型是如何实现的

这些都是怎么实现的？实际上，Scala 以与 Java 同样的方式存储整数：把它当作 32 位的字。这对在 JVM 上的效率及与 Java 库的互操作性方面来说都很重要。标准的操作如加法或乘法都被实现为基本操作。然而，当整数需要被当作（Java）对象看待的时候，Scala 使用了“备份”类 java.lang.Integer。如在整数上调用 toString 方法或者把整数赋值给 Any 类型的变量时，就会这么做。需要的时候，Int 类型的整数能被透明转换为 java.lang.Integer 类型的“装箱整数（boxed integer）”。

所有这些听上去都类似 Java5 里的自动装箱，当然它们的确很像。不过有一个关键的差异，就是 Scala 里的装箱比 Java 里的更少见。尝试下面的 Java 代码：

```
// Java 代码
boolean isEqual(int x,int y) {
  return x == y;
}
System.out.println(isEqual(421,421));
```

你当然会得到 true。现在，把 isEqual 的参数类型变为 java.lang.Integer（或 Object，结果都一样）：

```
// Java 代码
boolean isEqual(Integer x,Integer y){
  return x==y;
}
System.out.println(isEqual(421,421));
```

209

你得到了 false！原因是数字 421 被装箱了两次，因此参数 x 和 y 是两个不同的对象。

因为在引用类型上==表示引用相等，而 Integer 是引用类型，所以结果是 false。这说明了 Java 不是纯粹面向对象语言的一个方面。我们能清楚观察到基本类型和引用类型之间的差别。

现在在 Scala 里尝试同样的实验：

```
scala> def isEqual(x:Int, y:Int) = x == y
isEqual:(Int,Int)Boolean

scala> isEqual(421,421)
res10:Boolean = true

scala> def isEqual(x:Any, y:Any) = x == y
isEqual:(Any,Any)Boolean

scala> isEqual(421,421)
res11:Boolean = true
```

[2]译注：special marker trait。

实际上 Scala 里的相等操作==被设计为对类型表达透明。对值类型来说，就是自然的（数学或布尔）相等。对于引用类型，==被视为继承自 Object 的 equals 方法的别名。这个方法被初始地定义为引用相等，但被许多子类重写以实现它们自然理念上的相等性。这也意味着在 Scala 里你永远也不会落入 Java 知名的关于字符串比较的陷阱。在 Scala 中，字符串比较以其应有的方式工作：

```
scala> val x = "abcd".substring(2)
x:java.lang.String = cd

scala> val y = "abcd".substring(2)
y:java.lang.String = cd

scala> x == y
res12:Boolean = true
```

Java 里，x 与 y 的比较结果将是 false。程序员在这种情况应该用 equals，不过它容易被忘记。

然而，有些情况你需要使用引用相等代替用户定义的相等。例如，某些时候效率是首要因素，你想要把某些类散列合并（hash cons）然后通过引用相等比较它们的实例[3]。为解决这类问题，AnyRef 类定义了附加的 eq 方法，它不能被重写并且实现为引用相等（也就是说，它表现得就像 Java 里对于引用类型的==那样）。同样也有一个 eq 的反义词，被称为 ne。例如：

```
scala> val x = new String("abc")
x:java.lang.String = abc

scala> val y = new String("abc")
y:java.lang.String = abc

scala> x == y
res13:Boolean = true

scala> x eq y
res14:Boolean = false

scala> x ne y
res15:Boolean = true
```

Scala 的相等性会在第 28 章中讨论。

11.3 底层类型

在图 11.1 类型层级的底部你看到了两个类 scala.Null 和 Scala.Nothing。它们是用统一的方式处理 Scala 面向对象类型系统的某些“边界情况”的特殊类型。

Null 类是 null 引用对象的类型，它是每个引用类（就是说，每个继承自 AnyRef 的类）的子类。Null 不兼容值类型。例如，你不能把 null 值赋给整数变量：

```
scala> val i: Int = null
<console>:4: error:type mismatch;
```

[3]注：类实例的散列合并是指把创建的所有实例缓存在弱集合中。然后，一旦需要类的新实例，首先检查缓存。如果缓存中已经有一个元素与要创建的相等，就可以重用存在的实例。这样安排的结果是，任何 equals() 方法判定相等的两个实例同样在引用相等性上一致。

```
found    : Null(null)
required: Int
```

Nothing 类型在 Scala 的类层级的最底端；它是任何其他类型的子类型。然而，根本没有这个类型的任何值。要一个没有值的类型有什么意思呢？7.4 节中讨论过，Nothing 的一个用处是它标明了不正常的终止。例如 Scala 的标准库中的 Predef 对象有一个 error 方法，如下定义：

```
def error(message: String): Nothing =
  throw new RuntimeException(message)
```

error 的返回类型是 Nothing，告诉用户方法不是正常返回的（代之以抛出异常）。因为 Nothing 是任何其他类型的子类，所以你可以非常灵活地使用像 error 这样的方法。例如：

```
def divide(x: Int, y: Int): Int =
  if(y != 0) x / y
  else error("can't divide by zero")
```

“那么”（then）状态分支，x / y，类型为 Int，而“否则”（else）分支，调用了 error，类型为 Nothing。因为 Nothing 是 Int 的子类型，所以整个状态语句的类型是 Int，正如需要的那样。

11.4 小结

本章中我们展示了在 Scala 类层级的顶端和底端的类。现在你对 Scala 里类继承的理解打下了良好的基础，做好了理解混入组合的准备。下一章，你会学到关于特质的内容。

第 12 章

特质

特质（trait）是 Scala 里代码复用的基础单元。特质封装了方法和字段的定义，并可以通过混入到类中重用它们。与类的继承时每个类都只能继承唯一的超类不同，类可以混入任意多个特质。本章将告诉你特质是如何工作的并展示它们最常用到的两种方式：拓宽瘦接口为胖接口和定义可堆叠的改变。本章还将说明如何使用 Ordered 特质，以及特质和其他语言的多继承的比较。

12.1 特质是如何工作的

特质的定义除了使用关键字 trait 之外，与类定义无异。清单 12.1 举例如下：

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
```

清单 12.1 Philosophical 特质的定义

这个特质名为 Philosophical。它没有声明超类，因此和类一样，有个默认的超类 AnyRef。它定义了一个具体的方法 philosophize。这是个简单的特质，刚好能说明特质如何工作。

一旦特质被定义了，就可以使用 extends 或 with 关键字，把它混入类中。Scala 程序员“混入”特质而不是继承它们，因为特质的混入与那些其他语言中的多重继承有重要的差别。这部分将在 12.6 节中讨论。例如，清单 12.2 展示了使用 extends 混入 Philosophical 特质的类：

```
class Frog extends Philosophical {
  override def toString = "green"
}
```

清单 12.2 使用 extends 混入特质

你可以使用 extends 关键字混入特质；这种情况下隐式地继承了特质的超类。举例来说，在清单 12.2 中，Frog 类是 AnyRef（Philosophical 的超类）的子类并混入了 Philosophical。从特质继承的方法可以像从超类继承的方法那样使用。样例如下：

```
scala> val frog = new Frog
frog: Frog = green
```

```
scala> frog.philosophize()
I consume memory, therefore I am!
```

特质同样也是类型。以下是把 Philosophical 用作类型的例子：

```
scala> val phil: Philosophical = frog
phil: Philosophical = green

scala> phil.philosophize()
I consume memory, therefore I am!
```

phil 的类型是 Philosophical，是特质。因此，变量 phil 可以被初始化为任何混入了 Philosophical 特质的类的对象。

如果想把特质混入显式扩展超类的类里，可以用 extends 指明待扩展的超类，用 with 混入特质。清单 12.3 就是这样的例子。如果想混入多个特质，都加在 with 子句里就可以了。例如，假设存在 HasLegs 特质，你还可以把 Philosophical 和 HasLegs 都混入到 Frog 类中，参见清单 12.4。

214

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
}
```

清单 12.3 使用 with 混入特质

```
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
}
```

清单 12.4 混入多个特质

目前为止你看到的例子中，类 Frog 都继承了 Philosophical 的 philosophize 实现。或者，Frog 也可以重写 philosophize 方法。语法与重写超类中定义的方法一样。举例如下：

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
  override def philosophize() {
    println("It ain't easy being "+ toString +"!")
  }
}
```

因为 Frog 的这个新定义仍然混入了特质 Philosophize，你仍然可以把它当作这种类型的变量使用。但是由于 Frog 重写了 Philosophical 的 philosophize 实现，当你调用它的时候，你会得到新的回应：

```
scala> val phrog: Philosophical = new Frog
phrog: Philosophical = green

scala> phrog.philosophize()
It ain't easy being green!
```

215

这时你或许可以得出以下结论：特质就像是带有具体方法的 Java 接口。不过它其实能做更多的事情。例如，特质可以声明字段和维持状态值。实际上，你可以用特质的定义做任何用类定义能做的事，并且除了以下两点之外连语法都是一样的。第一点，特质不能有任何“类”参数，即传递给类的主构造器的参数。换句话说，尽管你可以定义如下的类：

```
class Point(x: Int, y: Int)
```

但是尝试定义下面的特质将会失败：

```
trait NoPoint(x: Int, y: Int) // 不能编译通过
```

你将在 20.5 节中找到如何规避这条限制的方法。

类和特质的另一个差别在于不论在类的哪个角落，super 调用都是静态绑定的，而在特质中，它们是动态绑定的。如果你在类中写下“super.toString”，你很明确哪个方法实现将被调用。然而如果你在特质中写了同样的东西，在你定义特质的时候 super 调用的方法实现尚未被定义。调用的实现将在每一次特质被混入到具体类的时候才被决定。这种处理 super 的有趣的行为是使得特质能以可堆叠的改变（stackable modifications）方式工作的关键，它将在 12.5 节中描述。Super 调用的规则将在 12.6 节给出。

12.2 瘦接口对阵胖接口

特质的一种主要应用方式是可以根据类已有的方法自动为类添加方法。也就是说，特质可以丰富一个瘦接口，把它变成胖接口。

瘦接口与胖接口的对阵体现了面向对象设计中常会面临的在实现者与接口用户之间的权衡。胖接口有更多的方法，对于调用者来说更便捷。客户可以选一个完全符合他们功能需要的方法。另一方面瘦接口有较少的方法，对于实现者来说更简单。然而调用瘦接口的客户因此要写更多的代码。由于没有更多可选的方法调用，他们或许不得不选一个不太完美匹配他们所需的方法并为了使用它写一些额外的代码。

Java 的接口常常是过瘦而非过胖。例如，从 Java 1.4 开始引入的 CharSequence 接口，是对于字符串类型的类来说通用的瘦接口，它持有一个字符序列。下面是把它看作 Scala 特质的定义：

```
trait CharSequence {
  def charAt(index: Int): Char
  def length: Int
  def subSequence(start: Int, end: Int): CharSequence
  def toString(): String
}
```

尽管 String 类成打的方法中的大多数都可以用在任何 CharSequence 上，但是 Java 的 CharSequence 接口定义仅提供了 4 个方法。如果 CharSequence 包含全部的 String 接口，那它将为 CharSequence 的实现者压上沉重的负担。任何实现 Java 里的 CharSequence 接口的程序员将不得不定义一大堆方法。因为 Scala 特质可以包含具体方法，这使得创建胖接口大为便捷。

在特质中添加具体方法使得胖瘦对阵的权衡大大倾向于胖接口。不像在 Java 里那样，在 Scala 中添加具体方法是一次性的劳动。你只要在特质中实现一次方法，而不再需要在每个混入特质的方法中重新实现它。因此，与没有特质的语言相比，Scala 里的胖接口没什么工作要做。

要使用特质丰富接口，只要简单地定义一个具有少量抽象方法的特质——特质接口的瘦部分——和潜在的大量具体方法，所有的都实现在抽象方法之上。然后你就可以把丰富了的特质混入到类中，实现接口的瘦部分，并最终获得具有全部胖接口内容的类。

12.3 样例：长方形对象

217

图形库总有许多表达为某些长方形的不同的类。其中的例子包括窗口、位图，以及鼠标选中的区域。为了使这些长方形对象便于使用，如果库能够提供诸如 `width`、`height`、`left`、`right`、`topLeft` 等等的几何查询会比较好。然而，许多这种最好能有的方法却会变成实现所有长方形对象的 Java 库作者的沉重负担。相反，如果这个库使用 Scala 编写，那么库作者就可以使用特质来方便地为所有他想要给的类提供所有这些便利方法。

要知道怎么做，先设想一下没有特质的代码是什么样的。首先会有一些基本的集合类如 `Point` 和 `Rectangle`:

```
class Point(val x: Int, val y: Int)

class Rectangle(val topLeft: Point, val bottomRight: Point) {
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // 以及其他更多的几何方法……
}
```

这个 `Rectangle` 类在它的主构造器中带两个点，分别是左上角和右下角的坐标。然后它通过对这两个点执行简单的计算实现了许多便捷方法诸如 `left`、`right` 和 `width`。

图库应该有的另一个类是 2-D 图像工具：

```
abstract class Component {
  def topLeft: Point
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // 以及其他更多的几何方法……
}
```

请注意 `left`、`right` 和 `width` 在两个类中的定义是一模一样。除了少许的变动外，他们将在任何其他的长方形对象的类中保持一致。

218

使用丰满了的特质可以消除这种重复。这个特质应该具有两个抽象方法：一个返回对象的左上角坐标，另一个返回右下角的坐标。然后它就可以应用到所有其他的几何查询的具体实现中。代码如清单 12.5 所示：

```
trait Rectangular {
  def topLeft: Point
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // 以及其他更多的几何方法……
}
```

清单 12.5 定义丰满了的特质

Component 类可以混入这个特质并获得 Rectangular 提供的所有的几何方法：

```
abstract class Component extends Rectangular {
  // 其他方法……
}
```

与之类似，Rectangle 本身也可以混入特质：

```
class Rectangle(val topLeft: Point, val bottomRight: Point)
    extends Rectangular {

  // 其他方法……
}
```

有了这些定义，你可以创建 Rectangle 对象并调用它的 width 或 left 等几何方法：

```
scala> val rect = new Rectangle(new Point(1, 1), new Point(10, 10))
rect: Rectangle = Rectangle@3536fd

scala> rect.left
res2: Int = 1

scala> rect.right
res3: Int = 10

scala> rect.width
res4: Int = 9
```


12.4 Ordered 特质

比较是另一个胖接口显得更便捷的领域。当你比较两个排序对象时，如果用一个方法调用就能获知精确的比较结果将非常便利。如果你想要“小于”，你会调用<，如果你想要“小于等于”，你会调用<=。对于瘦比较接口来说，你或许只有<方法，所以或许什么时候你会不得不写出类似于“(x < y) || (x == y)”这样的东西。胖接口将能够提供给你所有通用的比较操作符，从而让你能直接写出像“x <= y”这样的东西。

在看到 Ordered 之前，假想如果没有它你该怎么做。假设你从第 6 章取来了 Rational 类并且打算加入比较操作。你会最终得到这些东西[1]：

[1]注：本例基于的 Rational 类的全部代码在清单 6.5 中。

```
class Rational(n: Int, d: Int) {
  // ...
  def < (that: Rational) =
    this.numer * that.denom > that.numer * this.denom
  def > (that: Rational) = that < this
  def <= (that: Rational) = (this < that) || (this == that)
  def >= (that: Rational) = (this > that) || (this == that)
}
```

这个类定义了4个比较操作符(<，>，<=，和>=)，并且这是一个对定义胖接口代价的经典的演示。首先，注意到三个比较操作符都定义在使用第一个的基础上。例如，>被定义为<的反转，<=被定义为句法上的“小于或等于”。另外，还可以注意到所有这三个方法对于任何可比较的类来说都是一样的。讨论<=的时候不会有任何对于分数来说特别的东西。在比较的上下文中，<=永远表示着“小于或等于”。总而言之，这个类的代码中存在着与任何其他实现了比较操作的类中一样的大量的固定格式写法。

220

这个问题如此常见以至于Scala专门提供了一个特质解决它。这个特质就是Ordered。要使用它，你首先要用一个compare方法替换所有独立的比较方法。然后Ordered特质就会利用这个方法为你定义<、>、<=和>=。从而，Ordered特质让你可以通过仅仅实现一个方法，compare，使你的类具有了全套的比较方法。

以下是使用Ordered特质为Rational定义比较操作的例子：

```
class Rational(n: Int, d: Int) extends Ordered[Rational] {
  // ...
  def compare(that: Rational) =
    (this.numer * that.denom) - (that.numer * this.denom)
}
```

只有两件事要做。首先，这个版本的Rational混入了Ordered特质。不像你之前看到过的特质，Ordered需要你在混入的时候设定类型参数：type parameter。直到第19章我们才会详细讨论类型参数，现在你需要知道的就是当你混入Ordered的时候，你必须实际混入Ordered[*C*]，这里的*C*是你比较的元素的类。在本例中，Rational混入了Ordered[Rational]。

你要做的第二件事就是定义compare方法来比较两个对象。这个方法应该能比较方法的接收者this，和当作方法参数传入的对象。如果对象相同应该返回一个整数零，如果this小于参数返回负数，如果this大于参数返回正数。本例中，Rational的比较方法使用了基于把分数转换成公分母的公式然后再做结果分子的相减。有了混入和compare的定义，类Rational现在具有了所有4种比较方法：

```
scala> val half = new Rational(1, 2)
half: Rational = 1/2

scala> val third = new Rational(1, 3)
third: Rational = 1/3

scala> half < third
res5: Boolean = false

scala> half > third
res6: Boolean = true
```

221

在你想要实现通过某种比较排序的类的任何时候，你都应该考虑混入 Ordered 特质。如果这样做了，你将能给类的使用者提供丰富的比较方法集。

请当心 Ordered 特质并没有为你定义 equals 方法，因为它无法做到。问题在于要通过使用 compare 实现 equals 需要检查传入对象的类型，但是因为类型擦除，Ordered 本身无法做这种测试。因此，即使你继承了 Ordered，也还是需要自己定义 equals。你将在第 28 章了解该怎么做。

12.5 特质用来做可堆叠的改变

你已经看到了特质的一个主要用法：把瘦接口转变成胖接口。现在将转向第二个主要用法：为类提供可堆叠的改变。特质让你改变类的方法，它们能够通过堆叠这些改动的方式做到这点。

作为例子，考虑一下对一个整数队列堆叠改动。队列有两种操作：put，把整数放入队列，和 get，从尾部取出它们。队列是先进先出的，因此 get 应该依整数进入队列时的顺序把它们取出来。

假设有一个类实现了这样的队列，你可以定义特质执行如下的改动：

- Doubling：把所有放入到队列的数字加倍。
- Incrementing：把所有放入到队列的数字增值。
- Filtering：从队列中过滤掉负整数。

这三种特质代表了改动，因为它们改变了原始队列类的行为而并非定义了全新的队列类。这三种同样也是可堆叠的。你可以任选三者中的若干，把它们混入类中，并获得你所需改动的新类。

222

抽象的 IntQueue 类展示在清单 12.6 中。IntQueue 有一个 put 方法把整数添加到队列中，和一个 get 方法移除并返回它们。使用了 ArrayBuffer 的 IntQueue 基本实现参见清单 12.7。

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
```

清单 12.6 抽象类 IntQueue

```
import scala.collection.mutable.ArrayBuffer

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
```

清单 12.7 使用 ArrayBuffer 实现 BasicIntQueue

BasicIntQueue 类有一个私有字段持有数组缓存。get 方法从缓存的一端移除成员，put 方法从另一端加入元素。下面是这个实现运行时的样子：

```
scala> val queue = new BasicIntQueue
queue: BasicIntQueue = BasicIntQueue@24655f

scala> queue.put(10)

scala> queue.put(20)

scala> queue.get()
res9: Int = 10

scala> queue.get()
res10: Int = 20
```

223 到目前为止运行良好。现在看一下使用特质改变它的行为。清单 12.8 展示了在把整数放入队列的时候对其加倍。Doubling 特质有两件有趣的事情。第一个是它定义了超类 IntQueue。这个定义意味着特质只能混入扩展了 IntQueue 的类中。因此你可以把 Doubling 混入到 BasicIntQueue，但是不能混入 Rational。

```
trait Doubling extends IntQueue {
  abstract override def put(x: Int) { super.put(2 * x) }
}
```

清单 12.8 Doubling 可堆叠改动特质

第二件有趣的事情是特质在声明为抽象的方法中有一个 super 调用。这种调用对于普通的类来说是非法的，因为他们在执行时将必然失败。然而对于特质来说，这样的调用实际能够成功。因为特质里的 super 调用是动态绑定的，特质 Doubling 的 super 调用将直到被混入另一个特质或类之后，有了具体的方法定义时才工作。

这种安排对于实现可堆叠改动的特质来说是常常要用到的。为了告诉编译器你的目的，你必须对这种方法打上 abstract override 的标志。这种标识符的组合仅在特质成员的定义中被认可，在类中则不行，它意味着特质必须被混入某个具有期待方法的具体定义的类中。

这么简单的特质竟有这么多技巧！以下是使用这个特质的效果：

```
scala> class MyQueue extends BasicIntQueue with Doubling
defined class MyQueue

scala> val queue = new MyQueue
queue: MyQueue = MyQueue@91f017

scala> queue.put(10)

scala> queue.get()
res12: Int = 20
```

在这个解释器会话的第一行中，我们定义了类 MyQueue，它扩展了 BasicIntQueue 并混入了 Doubling。然后我们把 10 放在队列中，但是因为 Doubling 被混入了，10 被加倍。当我们从队列中 224 取出整数的时候，它变成了 20。

注意 MyQueue 没有定义一行新代码。只是简单地指明了一个类并混入了一个特质。这种情况下，你甚至可以直接 new 一个“BasicIntQueue with Doubling”以替代命名类。如清单 12.9 所示：

```
scala> val queue = new BasicIntQueue with Doubling
queue: BasicIntQueue with Doubling = $anon$1@5fa12d

scala> queue.put(10)

scala> queue.get()
res14: Int = 20
```

清单 12.9 在使用 new 实例化的时候混入特质

要想看到如何堆叠改动，我们需要定义另两个改动特质，Incrementing 和 Filtering。这两个特质的实现展示在清单 12.10 中：

```
trait Incrementing extends IntQueue {
  abstract override def put(x: Int) { super.put(x + 1) }
}
trait Filtering extends IntQueue {
  abstract override def put(x: Int) {
    if(x >= 0) super.put(x)
  }
}
```

清单 12.10 可堆叠改动特质 Incrementing 和 Filtering

有了这些改动，你现在可以挑选想要的组成特定的队列。比方说，这里有一个队列能够即过滤负数又对每个进队列的数字增量：

```
scala> val queue = (new BasicIntQueue
           with Incrementing with Filtering)
queue: BasicIntQueue with Incrementing with Filtering...

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res15: Int = 1

scala> queue.get()
res16: Int = 2
```

混入的次序非常重要[2]。准确的规则将在下一节给出，但是，粗略地说，越靠近右侧的特质越先起作用。当你调用带混入的类的方法时，最右侧特质的方法首先被调用。如果那个方法调用了 super，它调用其左侧特质的方法，以此类推。前面的例子里，Filtering 的 put 首先被调用，因此它移除了开始的负整数。Incrementing 的 put 第二个被调用，因此它对剩下的整数增量。

如果你逆转特质的次序，那么整数首先会加 1，然后如果仍然是负的才会被抛弃：

```
scala> val queue = (new BasicIntQueue
           with Filtering with Incrementing)
queue: BasicIntQueue with Filtering with Incrementing...

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res17: Int = 0
```

[2]注：一旦特质混入了类，你也可以称其为一个混入（mixin）。

```
scala> queue.get()
res18: Int = 1

scala> queue.get()
res19: Int = 2
```

总而言之，这种风格的代码能带给你极大的灵活性。通过以不同的组合和次序混入这三个特质，你可以定义 16 个不同的类。这对于这么少量的代码来说是非常灵活了，因此你应时刻关注是否有机会以可堆叠的改变方式安排代码。

12.6 为什么不是多重继承?

226

特质是一种继承多个类似于类的结构的方式，但是它与许多语言中的多重继承有很重要的差别。其中的一个尤为重要：super 的解释。对于多重继承来说，super 调用导致的方法调用可以在调用发生的地方明确决定。而对于特质来说，方法调用是由类和被混入到类的特质的线性化（linearization）所决定的。这种差别让前一节所描述的改动的堆叠成为可能。

在关注线性化之前，请花一点儿时间考虑一下在传统的多重继承语言中如何堆叠改动。假想有下列的代码，但是这次解释为多重继承而不是特质混入：

```
// 多重继承的思考实验
val q = new BasicIntQueue with Incrementing with Doubling
q.put(42) // 哪个 put 会被调用?
```

第一个问题是，哪个 put 方法会在这个调用中被引用？或许规则会决定最后一个超类胜出，本例中的 Doubling 将被调用。Doubling 将加倍它的参数并调用 super.put，大概就是这样。增量操作将不会发生！同样，如果规则决定第一个超类胜出，那么结果队列将增量整数但不会加倍它们。因此怎么排序都不会有效。

或许你会满足于允许程序员显式地指定在他们说 super 的时候他们想要的到底是哪个超类方法。比方说，假设下列 Scala 类似代码，super 似乎被显式地指定为 Incrementing 和 Doubling 调用：

```
// 多重继承思考实验
trait MyQueue extends BasicIntQueue
    with Incrementing with Doubling {

  def put(x: Int) {
    Incrementing.super.put(x) //（并非 Scala 真实代码）
    Doubling.super.put(x)
  }
}
```

227

这种方式将带给我们新的问题。这种尝试的繁冗几乎不算是问题。实际会发生的是基类的 put 方法将被调用两次——一次带了增量的值另一次带了加倍的值，但是没有一次是带了增量加倍的值。

显然使用多重继承对这个问题来说没有好的方案。你不得不返回到你的设计并分别提炼出代码。相反，Scala 里的特质方案很直接。你只要简单地混入 Incrementing 和 Doubling，Scala 对 super 的特别照顾让它迎刃而解。这与传统的多重继承相比必然有不同的地方，但这是什么呢?

就像在前面暗示的，答案就是线性化。当你使用 new 实例化一个类的时候，Scala 把这个类和所有它继承的类还有它的特质以线性（linear）的次序放在一起。然后，当你在其中的一个类中调用 super，被调用的方法就是方法链的下一节。除了最后一个调用 super 之外的方法，其净结果就是可堆叠的行为。

线性化的精确次序由语言的规格说明书描述。虽然有一点儿复杂，但你需要知道的主旨就是，在任何的线性化中，某个类总是被线性化在其所有超类和混入特质之前。因此，当你写了一个调用 super 的方法时，这个方法必将改变超类和混入特质的行为，没有其他路可走。

> **注意**
>
> 本节余下的部分描述了线性化的细节。如果现在没有兴趣了解这些细节，你可以跳过这部分。

Scala 的线性化的主要属性可以用下面的例子演示：假设你有一个类 Cat，继承自超类 Animal 以及两个特质 Furry 和 FourLegged。FourLegged 又扩展了另一个特质 HasLegs：

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Cat 类的继承层级和线性化次序展示在图 12.1 中。继承次序使用传统的 UML 标注指明：白色三角箭头表明继承，箭头指向超类型。黑底非三角箭头说明线性化次序，箭头指向 super 调用解决的方向。

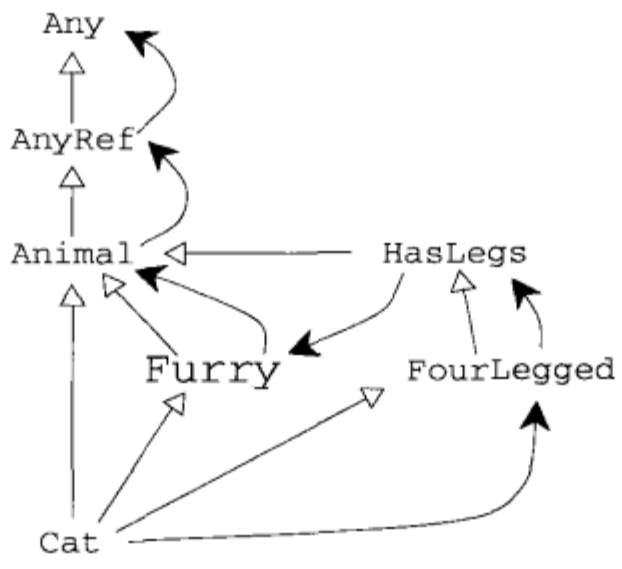


图 12.1 Cat 类的继承层级和线性化次序

Cat 的线性化次序以下列的从后向前的顺序计算。Cat 线性化的最后部分是它的超类，Animal 的线性化。这个线性化被无损地复制过来。（这些类型每一个的线性化次序展示在表 12.1 中。）因为 Animal 没有显式扩展超类或混入任何超特质，因此它默认地扩展了 AnyRef，并随之扩展了 Any。因此，Animal 的线性化，看上去是这样的：

表 12.1 Cat 层级中类型的线性化

类型	线性化
Animal	Animal, AnyRef, Any
Furry	Furry, Animal, AnyRef, Any
FourLegged	FourLegged, HasLegs, Animal, AnyRef, Any
HasLegs	HasLegs, Animal, AnyRef, Any
Cat	Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any

Animal → AnyRef → Any

倒数第二部分是第一个混入，特质 Furry 的线性化，但是所有已经在 Animal 的线性化之中的类现在被排除在外，因此 Cat 的线性化中每个类仅出现 1 次。结果是：

Furry → Animal → AnyRef → Any

它之前是 FourLegged 的线性化，任何已被复制到线性化中的超类及第一个混入再次被排除在外：

FourLegged → HasLegs → Furry → Animal → AnyRef → Any

229

最后，Cat 线性化的第一个类是 Cat 自己：

Cat → FourLegged → HasLegs → Furry → Animal → AnyRef → Any

当这些类和特质中的任何一个通过 super 调用了方法，那么被调用的实现将是它线性化的右侧的第一个实现。

12.7 特质，用还是不用?

当你实现了一个可重用的行为集合时，你将必须决定是使用特质还是抽象类。这里没有固定的规律，但是本节包含了几条可供考虑的规则。

如果行为不会被重用，那么就把它做成具体类。具体类没有可重用的行为。

如果要在多个不相关的类中重用，就做成特质。只有特质可以混入到不同的类层级中。

如果你希望从 Java 代码中继承它，就使用抽象类。因为特质和它的代码没有近似的 Java 模拟，在 Java 类里继承特质是很笨拙的。而继承 Scala 的类和继承 Java 的类完全一样。除了一个例外，只含有抽象成员的 Scala 特质将直接翻译成 Java 接口，因此即使你想用 Java 代码继承，也可以随心地定义这样的特质。要了解让 Java 和 Scala 一起工作的更多信息请看第 29 章。

如果你计划以编译后的方式发布它，并且你希望外部组织能够写一些继承自它的类，你应更倾向于使用抽象类。原因是当特质获得或失去成员，所有继承自它的类就算没有改变也都要被重新编译。如果外边客户仅需要调用行为，而不是继承自它，那么使用特质没有问题。

如果效率非常重要，则应该倾向于使用类。大多数 Java 运行时都能让类成员的虚方法调用快于接口方法调用。特质被编译成接口，因此会付出微小的性能代价。然而，仅当你知道哪个存疑的特质构成了性能瓶颈，并且有证据说明使用类代替确实能解决问题，才做这样的选择。

如果你还是不知道，在考虑了上面的这些情况之后，那么就开始尝试做成特质吧。你总可以在之后改变它，并且通常使用特质可以始终保持选择的可能性。

12.8 小结

本章为你展示了特质是如何工作的，并且在若干常见成例中如何使用它们。你可以发现特质与多重继承很像，但因为它使用了线性化解释 `super`，因此既避免了传统多重继承的困难，又能够让你堆叠行为。你还看到了 `Ordered` 特质并且学习了如何写你自己丰满的特质。

现在你已经看到了方方面面，有必要退回一步以整体的方式再看一眼特质。特质并不只是支持本章描述的成例。它们是通过继承让代码变为可重用的基础单元。因为这种天然的属性，许多有经验的 Scala 程序员在实现的早期阶段就开始使用特质。一个特质能够持有的并非一个完全的概念，只是概念的一个片段。随着设计的不断完整，这些片段可以通过特质的混入组合成更完整的概念。

Scala 编程

第 13 章

包和引用

做程序的时候，尤其是很大的程序，使耦合（coupling，所谓耦合是指程序的各个部分依赖于其他部分的程度）最小化是很重要的。低耦合能降低程序一部分的细微改变影响到另一部分的正常执行这样的风险。减少耦合性的方式之一是使用模块化风格编写代码。把程序分解成若干比较小的模块，把每块分成内部和外部。在模块的内部（即，模块的实现部分）工作时，你只需要和同样工作于这个模块的程序员交互；只有当你必须改变模块的外部（即模块的接口）时，才需要和工作于其他模块的开发人员交互。

本章展示了能够帮助你以模块化风格编程的若干构造。说明了如何把东西放在包里，通过引用让名字可见，并且通过访问修饰符控制定义的可见性。这些构造与 Java 的构造在思想上一致，但是也有一些区别，在 Scala 中通常能保持更好的一致性。所以，即便你已经从 Java 中学会了这些知识，读一下本章也是有价值的。

13.1 包

Scala 的代码采用了 Java 平台完整的包机制。本书到目前为止你已经看到的例子代码都是在未命名的（unnamed）包中。你可以用两种方式把代码放在命名包中。一种是通过把 package 子句放在文件顶端的方式把整个文件内容放进包里，如清单 13.1 所示。

```
package bobsrockets.navigation
class Navigator
```

清单 13.1　把文件的全部内容放进包里

清单 13.1 的 package 子句把 Navigator 类放在了名为 bobsrockets.navigation 的包里。想必这是 Bob's Rockets, Inc.公司开发的浏览软件（Navigation software）吧。

> **注意**
>
> 由于 Scala 代码是 Java 生态系统（ecosystem）的一部分，在你发布到公开场合的时候，推荐遵循 Java 的反域名习惯设置 Scala 包名。因此，更好的 Navigator 包名应该是 com.bobsrockets.navigation。然而本章里为了让例子容易理解我们去掉了“com.”。

Scala 里另一种把代码放进包里的方式更像 C#的命名空间。可以在 package 子句之后把要放到包里的定义用花括号括起来。除此之外，这种语法还能让你把文件的不同部分放在不同的包里。例如，你或许会把类的测试与原始代码一起放在同一个文件，但在不同的包里。如清单 13.2 所示：

```
package bobsrockets {
  package navigation {
    // 在 bobsrockets.navigation 包中
    class Navigator
    package tests {
      // 在 bobsrockets.navigation.tests 包中
      class NavigatorSuite
    }
  }
}
```

清单 13.2 同一个文件嵌入不同的包

234

清单 13.1 中类似于 Java 的语法实际上只是清单 13.2 中更一般性的嵌套语法的语法糖。实际上，如果一个包只是用来嵌入另一个包的话，你可以使用清单 13.3 展示的方式省去一个缩进：

```
package bobsrockets.navigation {
  // 在 bobsrockets.navigation 包里
  class Navigator

  package tests {

    // 在 bobsrockets.navigation.tests 包里
    class NavigatorSuite
  }
}
```

清单 13.3 较少缩进的嵌入包

正如注释所言，Scala 的包的确是嵌套的。也就是说，包 navigation 从语义上讲在包 bobsrockets 的内部。Java 包，尽管是分级的，却不是嵌套的。在 Java 里，当你命名一个包的时候，你必须从包层级的根开始。Scala 为了简化语言使用了更一致的规则。

看看清单 13.4。在 Booster 类中，引用 Navigator 不必以 bobsrockets.navigation.Navigator 这种全称的方式。由于包是嵌套的，所以可以简单地表示为 navigation.Navigator。这样的短名称之所以可行是因为 Booster 类包含在 bobsrockets 包中，而这个包又含有 navigation 成员。因此可以直接写 navigation，无须使用前缀，就好像类方法里的代码可以直接用类的其他方法而不用前缀。

235

```
package bobsrockets {
  package navigation {
    class Navigator
  }
  package launch {
    class Booster {
      // 不用写 bobsrockets.navigation.Navigator
      val nav = new navigation.Navigator
    }
  }
}
```

清单 13.4 Scala 的包确实是嵌套的

Scala 的作用域划分规则产生的另一个结果就是内部作用域的包可以隐匿被定义在外部作用域的同名包。举个例子，清单 13.5 有三个名为 launch 的包。一个 launch 在包 bobsrockets.navigation，一个在 bobsrockets，还有一个在顶级域（与那两个不在同一个文件中）。这些重复的名字都有效，毕竟这就是使用包的主要原因。但是这也意味着你必须为准确地访问到你想要访问的包多花一些心思。

想知道如何选择你要访问的包，看一下清单 13.5 的 MissionControl。如何引用 Booster1、Booster2 还有 Booster3？访问第一个最简单。直接使用 launch 引用就能让你访问到 bobsrockets.navigation.launch 包。因为 launch 包定义在最近的包含作用域中。所以，launch.Booster1 即可指到第一个推进器（booster）类。第二个同样也不麻烦。只要写下 bobrockets.launch.Booster2 就足以说明你所指的是哪个。不过第三个推进器类是个问题。在内嵌的 launch 包遮盖了顶层包的情况下，怎样才能访问到 Booster3 呢?

```
// 文件 launch.scala
package launch {
  class Booster3
}

// 文件 bobsrockets.scala
package bobsrockets {
  package navigation {
    package launch {
      class Booster1
    }
    class MissionControl1 {
      val booster1 = new launch.Booster1
      val booster2 = new bobsrockets.launch.Booster2
      val booster3 = new _root_.launch.Booster3
    }
  }
  package launch {
    class Booster2
  }
}
```

清单 13.5 访问隐藏的包名

为了处理这种情况，Scala 在所有用户可创建的包之外提供了名为_root_的包。换句话就是，任何你能写出来的顶层包都被当作是_root_包的成员。例如，清单 13.5 中的 launch 和 bobsrockets 都是 _root_ 包的成员。因此，_root_.launch 让你能访问顶层的 launch 包，_root_.launch.Booster3 指向的就是最外面的 booster 类。

13.2 引用

Scala 里，包和其成员可以用 import 子句来引用。之后引用的项目就可以用 File 这样的简单名访问，否则就要用 java.io.File 这样的全称。比如，考虑一下清单 13.6 所示的例子：

```
package bobsdelights

abstract class Fruit(
```

```
    val name: String,
    val color: String
    )

object Fruits {
  object Apple extends Fruit("apple", "red")
  object Orange extends Fruit("orange", "orange")
  object Pear extends Fruit("pear", "yellowish")
  val menu = List(Apple, Orange, Pear)
}
```

清单 13.6 鲍勃最爱的水果，已为引用做好准备

import 子句让包或对象的成员可以直接通过名称访问而不用通过前缀包或对象的名称。下面是一些简单的例子：

```
// 易于访问 Fruit
import bobsdelights.Fruit

// 易于访问 bobsdelights 的所有成员
import bobsdelights._

// 易于访问 Fruits 的所有成员
import bobsdelights.Fruits._
```

其中，第一个与 Java 的单类型引用一致，第二个对应于 Java 的按需（on-demand）引用，唯一的差别是 Scala 的按需引用写成以下划线（_）而不是星号（*）结尾（毕竟*是合法的 Scala 标识符！）。上面的第三个引用子句与 Java 的静态类字段引用一致。

这三个引用语句让你浅尝了引用的功能，但 Scala 引用实际上可以更为普遍。比方说，Scala 引用可以出现在任何地方，而不是仅仅在编译单元的开始处。同样，它们可以指向任意值。例如，清单 13.7 所示的引用是合法的：

```
def showFruit(fruit: Fruit) {
  import fruit._
  println(name +"s are "+ color)
}
```

清单 13.7 引用规范的（不是单例）对象的成员

方法 showFruit 引用了它的参数（fruit，Fruit 类型）的所有成员。之后的 println 语句就可以直接使用 name 和 color 了。这两个引用值等价于 fruit.name 和 fruit.color。当你把对象当作模块使用时这种语法尤其有用，这将在第 27 章描述。

另一个体现了 Scala 引用灵活性的方面是它们可以引用包自身，而不只是除了包之外的其他成员。这只有你把内嵌包想象成包含在围绕着它包之内才是自然的。例如，清单 13.8 里，包 java.util.regex 被引用。这使得 regex 可以用作简单名。要访问 java.util.regex 包的 Pattern 单例对象，你可以只是写成 regex.Pattern，如清单 13.8 所示：

```
import java.util.regex

class AStarB {
```

```
  // 访问 java.util.regex.Pattern
  val pat = regex.Pattern.compile("a*b")
}
```

清单 13.8 引用包名

Scala 的灵活的引用

Scala 的 import 子句比 Java 的更为灵活。在它们之间存在三点主要差异。在 Scala 中，引用：

- 可以出现在任何地方。
- 可以指的是（单例或正统的）对象及包。
- 可以重命名或隐藏一些被引用的成员。

Scala 的引用还可以重命名或隐藏成员。这可以在被引用成员的对象之后加上括号里的引用选择器子句（import selector clause）来做到。下面是一些例子：

```
import Fruits.{Apple, Orange}
```

此例只引用了对象 Fruits 的 Apple 和 Orange 成员。

```
import Fruits.{Apple => McIntosh, Orange}
```

此例从对象 Fruits 引用了 Apple 和 Orange 两个成员。不过，Apple 对象重命名为 McIntosh。因此这个对象可以用 Fruits.Apple 或 McIntosh 访问。重命名子句的格式是“<原始名> => <新名>”。

```
import java.sql.{Date => SDate}
```

此例以 SDate 的名字引用了 SQL 的日期类，以便同时以 Date 的名字引用普通的 Java 日期类。

```
import java.{sql => S}
```

此例以名称 S 引用了 java.sql 包，这样你就可以写成 S.Date。

```
import Fruits.{_}
```

此例引用了对象 Fruits 的所有成员。这与 import Fruits._同义。

```
import Fruits.{Apple => McIntosh, _}
```

此例从 Fruits 对象引用所有成员，不过重命名 Apple 为 McIntosh。

```
import Fruits.{Pear => _, _}
```

此例引用了 Fruits 的所有成员，Pear 除外。“<原始名> => _”格式的子句会从被引用的名字中排除<原始名>。从某种意义上来说，把某样东西重命名为‘_’就是表示把它隐藏掉。这对避免出现混淆的局面有所帮助。比方说你有两个包，Fruits 和 Notebooks，它们都定义了类 Apple。如果你只是想得到名为 Apple 的笔记本，而不是水果，你仍然能以如下方式按需使用两个引用：

```
import Notebooks._
import Fruits.{Apple => _, _}
```

此例将引用所有的 Notebooks 和除了 Apple 之外所有的水果。

这些例子演示了有选择性地及以不同的名称引用成员时，Scala 提供的极大的灵活性。总而言之，引用选择器可以包括下列模式：

- 简单名 x。把 x 包含进引用名集。
- 重命名子句 x => y。让名为 x 的成员以名称 y 出现。
- 隐藏子句 x => _。把 x 排除在引用名集之外。
- 全包括‘_’。引用除了前面子句提到的之外的全体成员。如果存在全包括，那么必须是引用选择的最后一个。

本节最初展示的比较简单的引用子句可以被视为带有选择子句的简写。例如，“import p._”等价于“import p.{_}”，并且“import p.n”等价于“import p.{n}”。

13.3 隐式引用

Scala 为每个程序隐式地添加了一些引用。本质上，就好像每个以“.scala”为扩展名的源文件的顶端都加载了下列三个引用子句：

```
import java.lang._    // java.lang 包的所有东西
import scala._        // scala 包的所有东西
import Predef._       // Predef 对象的所有东西
```

java.lang 包里面是标准 Java 类。它始终被隐式包含在 Scala 的 JVM 实现中。.NET 实现将代以引用 system 包，它是 java.lang 的.NET 模拟。因为 java.lang 是隐式引用的，所以说你就可以直接使用 Thread 而不需要写成 java.lang.Thread。

现在你已经毫无疑问地意识到，scala 包里面是标准的 Scala 库，包括许多通用的类和对象。因为 scala 被隐式引用，你可以直接用 List 而不需要写成 scala.List。

Predef 对象包含了许多 Scala 程序中常用到的类型、方法和隐式转换的定义。比方说，因为 Predef 是隐式引用，所以你可以直接写 assert 而不用写成 Predef.assert。

上面的这三个引用子句与其他的稍有不同，出现在靠后位置的引用将覆盖靠前的引用。例如，StringBuilder 类被定义在 scala 包及包 java.lang 中（从 Java 版本 1.5 开始）。因为 scala 引用覆盖了 java.lang 引用，所以 StringBuilder 简单名将被看作 scala.StringBuilder，而不是 java.lang.StringBuilder。

13.4 访问修饰符

包、类或对象的成员可以用访问修饰符 private 和 protected 做标记。这些修饰符把对成员的访问限制在代码确定的区域中。Scala 大体上遵守 Java 对访问修饰符的对待方式，但也有一些重要的差异，本节将对此加以说明。

私有成员

私有成员的处理方式与 Java 的相同。标记为 private 的成员仅在包含了成员定义的类或对象内部可见。Scala 里，同样的规则还应用到了内部类上。这种处理方式更具有一致性，但与 Java 形成了差别。参见清单 13.9 的例子：

```
class Outer {
  class Inner {
    private def f() { println("f") }
    class InnerMost {
      f() // OK
    }
  }
  (new Inner).f() // 错误：f 不可访问
}
```

清单 13.9 Scala 和 Java 的 private 访问差异

Scala 里，(new Inner).f()访问非法，因为 f 在 Inner 中被声明为 private 而访问不在类 Inner 之内。相反，在类 InnerMost 里访问 f 没有问题，因为这个访问包含在 Inner 类之内。Java 允许这两种访问，因为它允许外部类访问其内部类的私有成员。

保护成员

在 Scala 里，对保护成员的访问也同样比 Java 严格一些。Scala 里，保护成员只在定义了成员的类的子类中可以被访问。而 Java 中，还允许在同一个包的其他类中进行这种访问。在 Scala 中，另有途径达到这种效果，下面会对此进行说明，所以 protected 这样就可以了。清单 13.10 的例子演示了受保护的访问：

```
package p {
  class Super {
    protected def f() { println("f") }
  }
  class Sub extends Super {
    f()
  }
  class Other {
    (new Super).f() // 错误：f 不可访问
  }
}
```

清单 13.10 Scala 和 Java 的 protected 访问差异

例子中，Sub 类对 f 的访问没有问题，因为 f 在 Super 中被声明为 protected，而 Sub 是 Super 的子类。相反 Other 对 f 的访问不被允许，因为 Other 没有继承自 Super。Java 里，后者同样会被认可，因为 Other 与 Sub 在同一个包里。

公开成员

任何没有标记为 private 或 protected 的成员都是公开的。公开成员没有显式的修饰符。这样的成员可以在任何地方被访问。

保护的作用域

Scala 里的访问修饰符可以通过使用限定词强调。格式为 private[X] 或 protected[X] 的修饰符表示“直到” X 的私有或保护，这里 X 指代某个所属的包、类或单例对象。

243

带限定的访问修饰符提供给你非常细粒度的可见度控制。尤其是它们能让你表达诸如包私有、包保护，或者直到最外层类的私有这些 Java 的可访问性标注，而这些都不是用 Scala 的简单修饰符能直接表达的。除此之外，它们还能让你表达 Java 无法表达的访问规则。清单 13.11 的例子中用到了许多访问限定词。在这段代码中，类 Navigator 被标记为 private[bobsrockets]。就是说这个类对包含在 bobsrockets 包里的所有的类和对象可见。比方说，从 Vehicle 对象里对 Navigator 的访问是被允许的，因为 Vehicle 包含在包 launch 中，而 launch 包在 bobsrockets 中。相反，所有在 bobsrockets 包之外的代码都不能访问类 Navigator。

这种技巧在横跨了若干包的大型项目中非常有用。它允许你定义一些在你项目的若干子包中可见但对于项目外部的客户却始终不可见的东西。同样的技巧在 Java 里是不可能的。在那里，一旦定义超出了它所在包的边界，那就大白于全世界了。

```
package bobsrockets {
  package navigation {
    private[bobsrockets] class Navigator {
      protected[navigation] def useStarChart() {}
      class LegOfJourney {
        private[Navigator] val distance = 100
      }
      private[this] var speed = 200
    }
  }
  package launch {
    import navigation._
    object Vehicle {
      private[launch] val guide = new Navigator
    }
  }
}
```

清单 13.11 使用访问修饰词的灵活的保护作用域

244

当然，private 限定词也可以是它直接所属的包。清单 13.11 里对象 Vehicle 的 guide 访问修饰符就是这样的例子。这种访问修饰符等价于 Java 的包私有访问。

所有的限定词也可以用于 protected，与 private 的意思相同。也就是说，C 类里的 protected[X] 修饰符允许 C 的所有子类及修饰符所属的包、类或对象 X 访问带有此标记的定义。例

如，清单 13.11 里的 useStarChart 方法能被 Navigator 所有子类及包含在 navigation 包里的所有代码访问。这与 Java 的 protected 意思完全一致。

private 的限定词还能指向所属类或对象。例如清单 13.11 中 LegOfJourney 里的 distance 变量被标记为 private[Navigator]，因此它在类 Navigator 的任何地方都可见。这种访问能力与 Java 里的内部类的私有成员一致。private[C]里的 C 如果是最外层类，那么 private 的意思和 Java 的一致。

最后，Scala 还具有一种比 private 更严格的访问修饰符。被 private[this]标记的定义仅能在包含了定义的同一个对象中被访问。这种定义被称为对象私有（object-private）。例如，清单 13.11 中，类 Navigator 的 speed 定义就是对象私有的。这意味着所有的访问必须不仅是在 Navigator 类里，而且还要是同一个 Navigator 实例内发生的。因此在 Navigator 内访问“speed”和“this.speed”是合法的。然而以下的访问，即使发生在 Navigator 类之中也不被允许：

```
val other = new Navigator
other.speed // 此行不能编译
```

把成员标记为 private[this]可以保证它不能被同一个类中其他对象访问。这在把代码看作（程序员之间的契约）文档时比较有用。有时它也能让你写出更通用的变体注释（参见第 19.7 节相关细节）。

总结一下，表 13.1 罗列了 private 限定字的效果。每一行说明了一个被限定的私有修饰符及如果这个修饰符被附加在清单 13.11 的 LegOfJourney 类里声明的 distance 变量上意味着什么。

表 13.1 LegOfJourney.distance 上的私有限定符效果

没有修饰符	公开访问
private[bobsrockets]	在外部包中访问
private[navigation]	与 Java 的包可见度相同
private[Navigator]	与 Java 的 private 相同
private[LegOfJourney]	与 Scala 的 private 相同
private[this]	仅在同一个对象中可以访问

可见性和伴生对象

Java 里，静态成员和实例成员属于同一个类，因此访问修饰符可以统一地应用在它们之上。你已经知道在 Scala 里没有静态成员；作为替代，可以拥有包含成员的单例的伴生对象。例如，清单 13.12 里的 Rocket 对象是 Rocket 类的伴生：

```
class Rocket {
  import Rocket.fuel
  private def canGoHomeAgain = fuel > 20
}
object Rocket {
  private def fuel = 10
  def chooseStrategy(rocket: Rocket) {
```

```
    if (rocket.canGoHomeAgain)
      goHome()
    else
      pickAStar()
  }
  def goHome() {}
  def pickAStar() {}
}
```

清单 13.12 访问伴生类和对象的私有成员

对于私有或保护访问来说，Scala 的访问规则给予了伴生对象和类一些特权。类的所有访问权限都对伴生对象开放，反过来也是如此。具体地说，就是对象可以访问所有它的伴生类的私有成员，就好像类也可以访问伴生它的对象的所有私有成员一样。

246

举个例子，前面提到的 `Rocket` 类可以访问 `fuel` 方法，而它在 `Rocket` 对象中是被声明为 `private` 的。类似地，`Rocket` 对象也可以访问 `Rocket` 类里面的私有方法 `canGetHome`。

`protected static` 成员对于 Scala 和 Java 在修饰符的相似性方面来说是一个例外。Java 中 C 类的保护静态成员可以被 C 的所有子类访问。相反，伴生对象的 `protected` 成员没有意义，因为单例对象没有任何子类。

13.5 小结

本章里，你看到了可以把程序分割到包里的基本结构。它提供给你简单并有用的模块化方法，这样你就可以工作于非常大的代码块中却不受不同代码部分相互干扰的影响。这一系统在思想上与 Java 的包是相同的，但是由于 Scala 对一致性和通用性的选择使得其中存在了一些差别。

之后，第 27 章描述了比分割成包更灵活的模块系统。它除了让你能够把代码分成若干命名空间以外，还能让模块被参数化，以及相互继承。下一章，我们将把注意力转向断言和单元测试。

247

第 14 章

断言和单元测试

断言和单元测试是检查软件的行为是否符合需要的两种重要方式。本章，我们将向你展示在 Scala 里编写和执行它们的若干选择。

14.1 断言

Scala 里，断言被写为对预定义方法 assert 的调用[1]。表达式 assert(condition)将在 condition 条件不成立的时候抛出 AssertionError。assert 还有带两个参数的版本。表达式 assert(condition, explanation)会测试 condition，并且如果条件不成立，会抛出含有指定 explanation 作为说明的 AssertionError。explanation 的类型是 Any，因此你可以把任何对象当作说明参数。assert 方法会对传入的参数调用 toString，以获得可以放在 AssertionError 中的字符串说明。

例如，在清单 10.13 的 Element 类名为“above”的方法中，可以在调用 widen 之后放一个 assert 以确保变宽的元素具有相等的宽度。如清单 14.1 所示。

```
def above(that: Element): Element = {
  val this1 = this widen that.width
  val that1 = that widen this.width
  assert(this1.width == that1.width)
  elem(this1.contents ++ that1.contents)
}
```

清单 14.1　使用断言

另一种或许会使用断言的方式是在 widen 方法的结束处，在返回结果值之前，检查一下宽度是否正确。你可以这么做，首先把结果存在 val 里，然后对这个结果执行断言，并且如果断言成功，则返回 val 值。不过，你也可以使用 Predef 里的名为 ensuring 的方法来简化这些操作，该用法如清单 14.2 所示。

```
private def widen(w: Int): Element =
  if (w <= width)
    this
  else {
```

[1]注：assert 方法被定义在 Predef 单例对象中，该单例对象的所有成员将在每个 Scala 源文件中被自动引用。

```
    val left = elem(' ', (w - width) / 2, height)
    var right = elem(' ', w - width - left.width, height)
    left beside this beside right
  } ensuring (w <= _.width)
```

清单 14.2 使用 ensuring 断言函数结果

由于存在隐式转换，因此 ensuring 方法能被用在任何结果类型上。尽管这段代码看上去好像是对 widen 的结果（类型 Element）调用 ensuring，但实际上是对 Element 隐式转换成的类型调用了 ensuring。ensuring 方法带一个函数做参数，该函数是接受一个结果类型对象并返回 Boolean 类型的论断函数（predicate function）。ensuring 会把结果传给这个函数。如果函数返回 true，ensuring 将返回结果，否则，ensuring 将抛出 AssertionError。

在这个例子中，论断函数是“w <= _.width”。下划线是传递给论断参数的占位符，即 widen 方法的结果 Element。如果传给 widen 的宽度（即 w）小于或等于结果 Element 的 width，论断将返回 true，于是 ensuring 将返回被调用的 Element 作为结果。因为这是 widen 方法的最后一个表达式，所以 widen 本身将返回 Element 作为结果。

250

断言（以及 ensuring 检查）可以使用 JVM 的-ea 和-da 命令行标志开放和禁止。开放的时候，每个断言被当作对使用软件运行时产生的实际数据进行的小测试。本章余下的部分，我们将集中在编写外部单元测试上，它们将提供它们自己的测试数据并且独立于应用运行。

14.2 Scala 里的单元测试

Scala 的单元测试可以有许多选择，从 Java 实现的工具，如 JUnit 和 TestNG，到 Scala 编写的新工具，如 ScalaTest、specs，还有 ScalaCheck。本章余下的部分，我们会快速浏览这些工具。首先看的是 ScalaTest。

ScalaTest 提供了若干编写测试的方法，最简单的就是创建扩展 org.scalatest.Suite 的类并在这些类中定义测试方法。Suite 代表一个测试集。测试方法名以“test”开头。示例见清单 14.3：

```
import org.scalatest.Suite
import Element.elem

class ElementSuite extends Suite {

  def testUniformElement() {
    val ele = elem('x', 2, 3)
    assert(ele.width == 2)
  }
}
```

清单 14.3 使用 Suite 编写测试方法

尽管 ScalaTest 包含了 Runner 应用，你也还是可以直接在 Scala 解释器中通过调用 execute 方法运行 Suite。特质 Suite 的 execute 方法使用反射发现测试方法并调用它们。举例如下：

```
scala> (new ElementSuite).execute()
Test Starting - ElementSuite.testUniformElement
Test Succeeded - ElementSuite.testUniformElement
```

251

由于 execute 可以在 Suite 子类型中重载，因此 ScalaTest 为不同风格的测试提供了便利。比方说，ScalaTest 提供了名为 FunSuite 的特质，重载了 execute，从而可以让你以函数值的方式而不是方法定义测试。清单 14.4 示例如下：

```
import org.scalatest.FunSuite
import Element.elem
class ElementSuite extends FunSuite {
  test("elem result should have passed width") {
    val ele = elem('x', 2, 3)
    assert(ele.width == 2)
  }
}
```

清单 14.4 使用 FunSuite 编写测试函数

FunSuite 里的“Fun”表示函数。“test”是定义在 FunSuite 中的方法，它将被 ElementSuite 的主构造器调用。圆括号里的字符串指定了测试的名称，花括号之间的是测试代码。测试代码是被作为传名参数传递给 test 的函数，并由 test 函数注册之后运行。FunSuite 的好处之一是你不必以“test”开头命名所有的测试。而且，你可以更容易地给测试起很长的名称，因为你不需要把它们编码为驼峰形式，而对于测试方法你就必须这么做了。[2]

14.3 翔实的失败报告

前面的两个例子尝试创建宽度为 2 的元素并且断言结果元素的宽度的确是 2。如果这个断言失败了，你会看到一条指明断言失败的消息。你会得到行号，但是无法知道不相等的两个值。你可以在断言中放一个包含了这两个值的字符串信息得到这两个值，不过还有一种更清晰的方式，就是使用三等号操作符，这是 ScalaTest 为了这个目的专门提供的：

```
assert(ele.width === 2)
```

如果断言失败了，你会在失败报告中看到“3 did not equal 2”这样的信息。它能告诉你 ele.width 错误地返回了 3。三等号操作符不能区分实际结果和希望结果。它只是说明左侧的操作元不等于右侧的操作元。如果你希望强调这种区分，你可以改用 ScalaTest 的 expect 方法，如下：

```
expect(2) {
  ele.width
}
```

这个表达式可以说明你希望在花括号之间的代码返回的结果是 2。如果返回的结果是 3，你会在测试失败报告中看到“Expected 2, but got 3”的消息。

如果你想要检查方法是否抛出了期待的异常，可以使用 ScalaTest 的 intercept 方法，如下：

```
intercept(classOf[IllegalArgumentException]) {
  elem('x', -2, 3)
}
```

[2]注：你可以从 *http://www.scalatest.org/* 下载 ScalaTest。

如果花括号之间的代码被一个参数指定的异常类的实例突然中断，intercept 将返回捕获的异常，以便于你在之后进行检查。绝大多数情况下，你可以只关心期待的异常被抛出，并忽略 intercept 的结果，就像上面的例子中的那样。另一方面，如果代码没有抛出异常，或抛出了不同的异常，intercept 将抛出 AssertionError，并且你将在失败报告中得到有帮助的错误消息，如：

```
Expected IllegalArgumentException to be thrown,
  but NegativeArraySizeException was thrown.
```

ScalaTest 的===符号和它的 expect，以及 intercept 方法的目的是帮助你编写简明清晰的基于断言的测试。下一节，我们将说明如何在 Scala 里使用这种语法编写 JUnit 和 TestNG 测试。

253

14.4 使用 JUnit 和 TestNG

Java 平台上最著名的单元测试架构就是 JUnit，它是 Kent Bech 和 Erich Gamma 编写的开源工具。Scala 里编写 JUnit 测试非常容易。以下是使用 JUnit 3.8.1 的例子：

```
import junit.framework.TestCase
import junit.framework.Assert.assertEquals
import junit.framework.Assert.fail
import Element.elem

class ElementTestCase extends TestCase {

  def testUniformElement() {
    val ele = elem('x', 2, 3)
    assertEquals(2, ele.width)
    assertEquals(3, ele.height)
    try {
      elem('x', -2, 3)
      fail()
    }
    catch {
      case e: IllegalArgumentException => // 预期的
    }
  }
}
```

一旦你完成了这个类的编译，JUnit 就会像运行其他任何 TestCase 那样运行它。JUnit 不会在意它是用 Scala 写的。不过，如果你希望在 JUnit 3 测试中使用 ScalaTest 的断言语法，你可以换成 JUnit3Suite 的子类，参见清单 14.5。

JUnit3Suite 特质扩展了 TestCase，因此一旦你完成了这个类的编译之后，即使它使用了 ScalaTest 的更简洁的断言语法，JUnit 也能很好地运行它。不只如此，因为 JUnit3Suite 混入了 ScalaTest 的特质 Suite，你还能使用 ScalaTest 的运行器运行这个测试。其目的是提供一条柔和的迁移途径，能够让 JUnit 使用者开始在 Scala 中编写 JUnit 测试，还能得益于 Scala 的清晰性。ScalaTest 同样具有 JUnit3WrapperSuite，能让你使用 ScalaTest 的运行器运行用 Java 编写的现存 JUnit 测试。

254

```
import org.scalatest.junit.JUnit3Suite
import Element.elem
```

```
class ElementSuite extends JUnit3Suite {

  def testUniformElement() {
    val ele = elem('x', 2, 3)
    assert(ele.width === 2)
    expect(3) { ele.height }
    intercept(classOf[IllegalArgumentException]) {
      elem('x', -2, 3)
    }
  }
}
```

清单 14.5 使用 JUnit3Suite 编写 JUnit 测试

ScalaTest 为 JUnit 4 和 TestNG 提供了相似的集成类，两者都大量地使用了注解。我们将展示一个使用由 Cédric Beust 和 Alexandru Popescu 编写的开源架构 TestNG 的例子。与使用 JUnit 一样，用 Scala 编写 TestNG 测试非常容易，编译，然后使用 TestNG 的运行器运行它们即可。举例如下：

```
import org.testng.annotations.Test
import org.testng.Assert.assertEquals
import Element.elem

class ElementTests {
  @Test def verifyUniformElement() {
    val ele = elem('x', 2, 3)
    assertEquals(ele.width, 2)
    assertEquals(ele.height, 3)
  }
  @Test {
    val expectedExceptions =
      Array(classOf[IllegalArgumentException])
  }
  def elemShouldThrowIAE() { elem('x', -2, 3) }
}
```

不过如果你喜欢在 TestNG 测试中使用 ScalaTest 的断言语法，可以扩展 TestNGSuite 特质，清单 14.6 如下：

```
import org.scalatest.testng.TestNGSuite
import org.testng.annotations.Test
import Element.elem

class ElementSuite extends TestNGSuite {

  @Test def verifyUniformElement() {
    val ele = elem('x', 2, 3)
    assert(ele.width === 2)
    expect(3) { ele.height }
    intercept(classOf[IllegalArgumentException]) {
      elem('x', -2, 3)
    }
  }
}
```

清单 14.6 使用 TestNGSuite 编写 TestNG 测试

正如使用 JUnit3Suite 那样，你既可以用 TestNG，也可以用 ScalaTest 运行 TestNGSuite，ScalaTest 也同样提供了 TestNGWrapperSuite，以便你能够用 ScalaTest 运行用 Java 写的现存 TestNG 测试。关于用 Scala 写的 JUnit 4 测试的例子，请参见 29.2 节。

14.5 规格测试

行为驱动开发（behavior-driven development，BDD）测试风格中，重点放在了编写人类可读的预期代码行为的规格说明上，并辅以验证代码具有规定行为的测试。ScalaTest 包含了特质 Spec，以便于进行这种风格的测试。例子如清单 14.7 所示。

```
import org.scalatest.Spec

class ElementSpec extends Spec {

  describe("A UniformElement") {

    it("should have a width equal to the passed value") {
      val ele = elem('x', 2, 3)
      assert(ele.width === 2)
    }
    it("should have a height equal to the passed value") {
      val ele = elem('x', 2, 3)
      assert(ele.height === 3)
    }
    it("should throw an IAE if passed a negative width") {
      intercept(classOf[IllegalArgumentException]) {
        elem('x', -2, 3)
      }
    }
  }
}
```

清单 14.7 使用 ScalaTest 的 Spec 规格化和测试行为

Spec 包含了“描述部分”和“规格部分”。描述部分写为 describe，然后是带括号的字符串和代码块，它描述了要被规格化和测试的“目标”。规格部分写为 it，然后是带括号的字符串和代码块，（在字符串中）规格化了目标的一小块行为并（在代码块中）提供了验证这种行为的代码。在 Spec 被执行之后，它将把规格部分当作 ScalaTest 的测试逐一运行。Spec 在执行之后会产生输出，读起来很像规格说明。例如，以下是在解释器中执行清单 14.7，ElementSpec 产生的输出效果：

```
scala> (new ElementSpec).execute()
A UniformElement
- should have a width equal to the passed value
- should have a height equal to the passed value
- should throw an IAE if passed a negative width
```

由 Eric Torreborre 用 Scala 写的开源工具 specs 测试框架也同样支持 BDD 风格测试，不过语法不同。例如，清单 14.8 展示了使用 specs 编写测试的例子：

```
import org.specs._

object ElementSpecification extends Specification {
  "A UniformElement" should {
    "have a width equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.width must be_==(2)
    }
    "have a height equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.height must be_==(3)
    }
    "throw an IAE if passed a negative width" in {
      elem('x', -2, 3) must
        throwA(IllegalArgumentException)
    }
  }
}
```

清单 14.8 使用 specs 架构规格化和测试行为

specs的一个目标是让你能够编写读起来更像自然语言的断言，并且产生详尽的失败消息。为此specs提供了大量的匹配（matcher）。你也可以创建自己的匹配。清单 14.8 中，在包含了“must be_==”和“must throwA”的行中，你可以看到一些实际的匹配例子。你还可以通过把 org.specs.SpecsMatchers 特质混入到你的测试类的方式在用 Scala 写的 ScalaTest、JUnit 或 TestNG 测试中使用 specs 匹配。你可以单独使用 specs，不过它也可以与 ScalaTest 和 JUnit 集成在一起，所以你也可以使用这些工具运行 specs 测试。[3]

257

14.6 基于属性的测试

Scala 的另一个有用的测试工具是 ScalaCheck，是由 Rickard Nilsson 编写的开源架构。ScalaCheck 能让你指定待测代码须遵循的属性。对于每个属性，ScalaCheck 将产生测试数据并运行测试以验证其是否正确。清单 14.9 展示了在 ScalaTest 测试集里使用 ScalaCheck 的例子。

258

```
import org.scalatest.prop.FunSuite
import org.scalacheck.Prop._
import Element.elem

class ElementSuite extends FunSuite {

  test("elem result should have passed width", (w: Int) =>
    w > 0 ==> (elem('x', w, 3).width == w)
  )

  test("elem result should have passed height", (h: Int) =>
    h > 0 ==> (elem('x', 2, h).height == h)
  )
}
```

清单 14.9 使用 ScalaCheck 编写基于属性的测试

[3]注：你可以从 *http://code.google.com/p/specs/* 下载 specs。

这个例子中，我们检查了 elem 工厂应该遵循的两个属性。ScalaCheck 属性被表示为用所需的测试数据做参数的函数值，这些所需的测数据将由 ScalaCheck 产生。清单 14.9 展示的第一个属性中，测试数据是名为 w 的整数，代表了宽度。在函数体里可以看到：

```
w > 0 ==> (elem('x', w, 3).width == w)
```

==>符号是含义操作符（implication operator）。说明当左手边的表达式为真时，那么右侧的表达式也必须为真。因此在这个例子中，==>右侧的表达式在 w 大于 0 的时候必须为真。本例中若传递给 elem 工厂的宽度与工厂返回的 Element 宽度一致的话，右侧的表达式将产生真值。

有了这一小段代码，ScalaCheck 将有可能产生几百个 w 值并测试每一个，查看有哪个值使得属性失败。如果对于每个 ScalaCheck 尝试的值来说属性始终为真，那么测试通过。反之，测试就会被包含了引起失败的值的信息的 AssertionError 中断掉。

在清单 14.9 中，每个测试都由单个属性组成。然而有些时候，你或许希望在单个测试中检查多个属性，或者在同一个测试中既执行属性检查又包含断言。用 Scala 的 Checkers 测试特质就简单了。只要把 Checkers 混入你的测试类，然后把属性传递给几个“check”方法之一即可。例如，清单 14.10 里演示了 JUnit3Suite 执行了前面的例子里看到过的两个相同的 ScalaCheck 属性检查，但这次它们包含在一个测试中。与所有其他的 JUnit3Suite 一样，这个类也是 JUnit 的 TestCase，因此使用 ScalaTest 或 JUnit 都可以运行。[4]

259

```
import org.scalatest.junit.JUnit3Suite
import org.scalatest.prop.Checkers
import org.scalacheck.Prop._
import Element.elem

class ElementSuite extends JUnit3Suite with Checkers {

  def testUniformElement() {
    check((w: Int) => w > 0 ==> (elem('x', w, 3).width == w))
    check((h: Int) => h > 0 ==> (elem('x', 2, h).height == h))
  }
}
```

清单 14.10 使用 Checkers 在 JUnit 的 TestCase 里检查属性

14.7 组织和运行测试

本章提到的每个结构都提供了一些组织和运行测试的机制。在这一节，我们将快速浏览 ScalaTest 的方案。不过如果想对上述的任一架构有完整的认识，你需要参考它们的完整文档。

在 ScalaTest 中，可以通过在 Suite 内部嵌套 Suite 来管理较大的测试集。一个 Suite 在执行的时候，会把内嵌的 Suite 当做它的测试执行。以此类推，内嵌的 Suite 会依次执行它们的内嵌 Suite。因此一个大的测试集被表达成 Suite 对象树。当你执行树的根节点 Suite 时，树上的所有 Suite 都将被执行。

[4]注：你可以在 *http://code.google.com/p/scalacheck/* 下载到 ScalaCheck。

你可以手动或自动嵌套测试集。手动嵌套的话，要么在你的 Suite 里重载 nestedSuites 方法，要么采用 ScalaTest 专为此提供的方案，把你想要嵌套的 Suite 传递给 SuperSuite 类的构造器。自动嵌套的话，准备一个与 ScalaTest 的 Runner 同名的包，保存需要自动发现的 Suite，把它们内嵌到根 Suite 下，然后执行根 Suite。

ScalaTest 的 Runner 应用可以在命令行或 ant 任务中调用。不过必须指定要运行的测试集，方式包括显式说明测试集名称或说明你想要 Runner 执行自动发现的测试集名称前缀。你可以选择性地指定运行路径（runpath），目录列表和装载测试类的 JAR 文件及它们测试的代码[5]。你还可以指定一个或多个报表器，以格式化测试结果的外观。

例如，ScalaTest 的发布包里包含了测试 ScalaTest 自身的测试集。你可以使用下列命令执行其中的测试集 SuiteSuite[6]：

```
$ scala -cp scalatest-0.9.4.jar org.scalatest.tools.Runner
      -p "scalatest-0.9.4-tests.jar" -s org.scalatest.SuiteSuite
```

-cp 的目的是把 ScalaTest 的 JAR 文件放在类路径中。下一项，org.scalatest.tools.Runner，是 Runner 应用的全称。Scala 将运行这个应用并把剩余的项当做命令行参数传递给应用。-p 指定了运行路径，这里是包含了测试集类的 JAR 文件：scalatest-0.9.4-tests.jar。-s 说明了 SuiteSuite 是要执行的测试集。因为没有显示指定报表器，默认将采用图形报表器。结果展示在图 14.1 中。

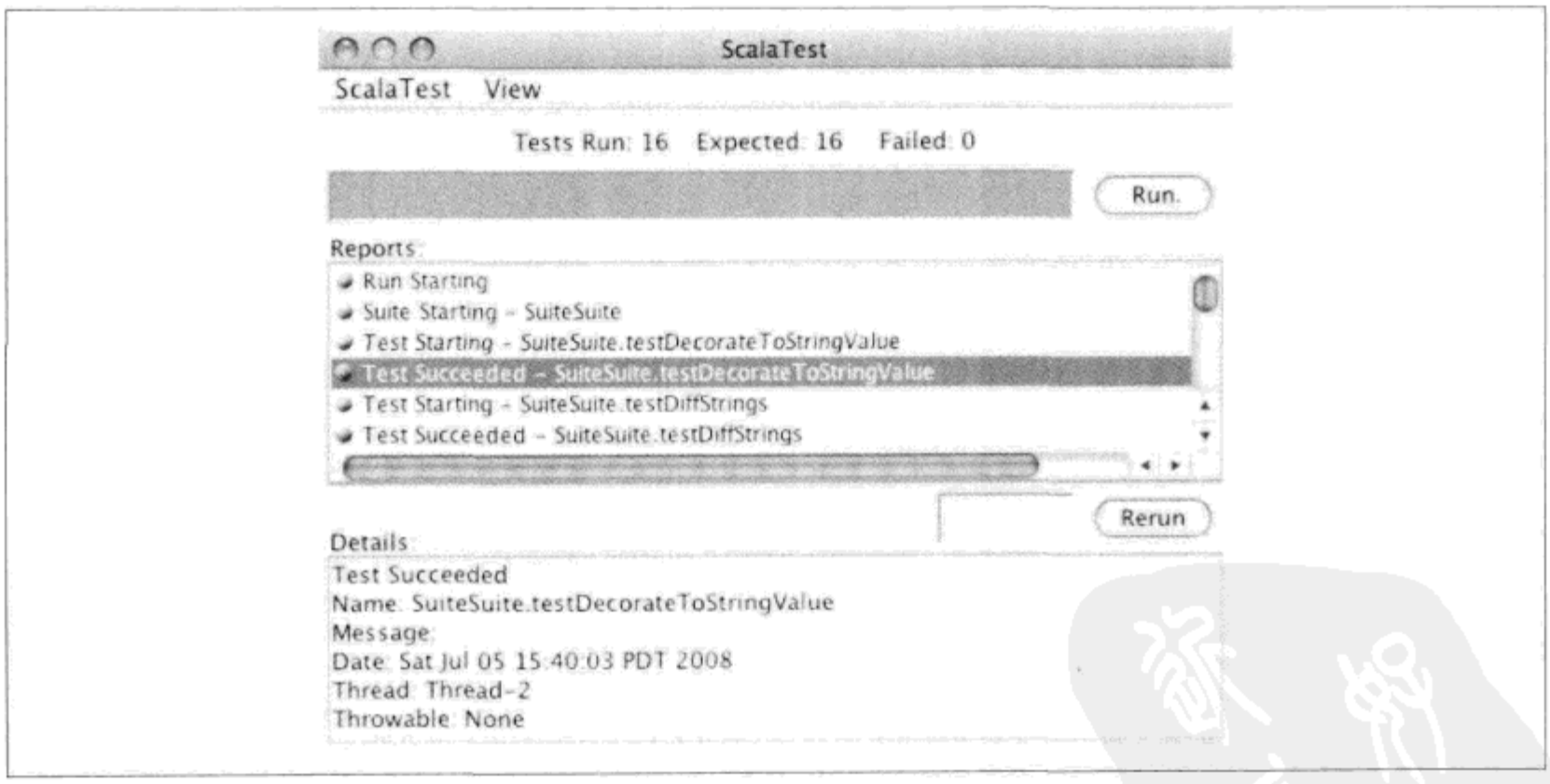


图 14.1 ScalaTest 的图形报表器

[5]注：测试可以在运行路径或类路径的任何地方，不过原则上你应保持测试代码与生产代码的隔离，比如说采用一个分离的能够映射你源代码树的目录层级结构。

[6]注：SuiteSuite 如此得名的原因是它是测试 Suite 特质的测试集。

14.8 小结

本章中我们看到了把断言直接混合在生产代码中的例子，以及把它们放在外部的单元测试的例子。你可以发现作为 Scala 程序员，不但可以利用 Java 社区中流行的测试工具，如 JUnit 和 TestNG，还可以使用专为 Scala 设计的新工具，如 ScalaTest、ScalaCheck 还有 specs。不论是代码内的断言还是单元测试，都能帮助你提高软件的质量。我们感觉本章描述的技术非常重要，即使是作为 Scala 的教程也不应缺少。不过下一章，我们将回归语言的教程，介绍 Scala 一个非常有用的方面：模式匹配。

第 15 章

样本类和模式匹配

本章介绍样本类（case class）和模式匹配（pattern matching），这两种在编写规范的、无封装数据结构时会用到的构件。它们对于树型递归数据尤其有用。

如果之前曾经使用过函数式语言编程，你或许知道什么是模式匹配。不过样本类是新概念。它是 Scala 避免在对象上使用模式匹配时需要大量的固定写法而采用的方式。通常，你要知道的只是在那些你希望用来做模式匹配的每个类前加一个 `case` 关键字即可。

本章以一个样本类和模式匹配的简单例子起头，然后了解所有支持的模式种类，谈论一下封闭（sealed）类的角色，研讨一下 `Option` 类型，然后演示一些语言中应用模式匹配的并非显而易见的地方。最后，以一个更大一些并且更实际一些的模式匹配例子收尾。

15.1 简单例子

在深入所有的规则及模式匹配之间的细微差别之前，最好先来看一个简单的例子以获得基础的印象。我们假设你想要写一个操控数学表达式的库，或许是把它作为正在设计的域特化语言的一部分。

处理这个问题的第一步是定义输入数据。为了保持简洁性，我们将集中注意力在由变量、数字、一元及二元操作符组成的数学表达式上。这可以通过清单 15.1 展示的 Scala 类层级来表达。


```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
  left: Expr, right: Expr) extends Expr
```

清单 15.1　定义样本类

层级包括一个抽象基类 `Expr` 和四个子类，每个代表一种表达式[1]。所有的五个类都没有类结构体。就像之前提到的，Scala 里可以去掉围绕空类结构体的花括号，因此 `class C` 与 `class C {}` 相同。

样本类

清单 15.1 的声明中另一个值得关注的事情就是每个子类都有一个 `case` 修饰符。带有这种修饰符的

[1]注：除了抽象类之外，我们可以等价地选择把类层级的根类模型化为特质。不过抽象类或许会稍微更有效一些。

类被称为样本类(case class)。这种修饰符可以让 Scala 编译器自动为你的类添加一些句法上的便捷设定。

首先，它会添加与类名一致的工厂方法。也就是你可以，比方说，写成 Var("x")来构造 Var 对象以替代稍长一些的 new Var("x")：

```
scala> val v = Var("x")
v: Var = Var(x)
```

尤其在你把工厂方法嵌套在一起的时候，这种方式显得极为有用。因为代码里不会到处都是讨厌的 new 关键字，可以看一下如下表达式的结构：

```
scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

264

第二个句法便捷设定是样本类参数列表中的所有参数隐式获得了 val 前缀，因此它被当作字段维护：

```
scala> v.name
res0: String = x

scala> op.left
res1: Expr = Number(1.0)
```

第三个，是编译器为你的类添加了方法 toString、hashCode 和 equals 的“自然”实现。它们能够打印，哈希和比较由类及（递归地得到）其所有参数组成的整棵树。因为 Scala 里的==始终直接转到 equals，这也就特别意味着样本类的元素一直是在做结构化的比较：

```
scala> println(op)
BinOp(+,Number(1.0),Var(x))

scala> op.right == Var("x")
res3: Boolean = true
```

所有这些转换以极低的代价带来了大量的便利。代价就是必须写 case 修饰符并且你的类和对象都会变得稍微大一点儿。变大的原因是因为产生了附加的方法及对于每个构造器参数添加了隐含的字段。不过，样本类最大的好处还在于它们能够支持模式匹配。

模式匹配

假如说你想要简化之前看到的数学表达式。可能有许多的简化规则。以下三条规则只是作为演示：

```
UnOp("–", UnOp("–", e))      => e  // 双重负号
BinOp("+", e, Number(0))     => e  // 加 0
BinOp("*", e, Number(1))     => e  // 乘 1
```

使用模式匹配，这些规则可以用几乎同样的形式作为 Scala 简化函数的核心，如清单 15.2 所示。函数 simplifyTop 可以如下方式使用：

265

```
scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
res4: Expr = Var(x)
```

```
def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) => e  //双重负号
  case BinOp("+", e, Number(0)) => e  //加 0
  case BinOp("*", e, Number(1)) => e  //乘 1
```

```
    case _ => expr
  }
```

清单 15.2 simplifyTop 函数，使用了模式匹配

simplifyTop 右侧的部分组成了 match 表达式。match 对应于 Java 里的 switch，不过它写在选择器表达式之后。也就是说：

```
选择器 match { 备选项 }
```

取代了：

```
switch (选择器) { 备选项 }
```

一个模式匹配包含了一系列备选项（alternative），每个都开始于关键字 case。每个备选项都包含了一个模式（pattern）及一到多个表达式，它们将在模式匹配过程中被计算。箭头符号=>隔开了模式和表达式。

match 表达式通过以代码编写的先后次序尝试每个模式来完成计算。第一个匹配的模式被选中，然后跟在箭头后的部分被选中并被执行。

类似于"+"或 1 这样的常量模式（constant pattern）匹配的值等于用==判断相等的常量。类似于 e 这样的变量模式（variable pattern）匹配所有值。之后在 case 子句的右侧，这个变量指代了被匹配的值。本例中，可以注意到前三个例子的计算结果 e，就是相关的模式里绑定的变量。通配模式（wildcard pattern）（_）同样也匹配所有值，不过没有引入指向那个值的变量名。清单 15.2 中，可以注意到 match 是如何以一个默认的 case 结束的，它不对表达式做任何改变，而仅返回用来做匹配的表达式 expr。

构造器模式（constructor pattern）类似于 UnOp("-", e)。这种模式匹配所有类型为 UnOp，并且第一个参数匹配"-"，第二个参数匹配 e 的值。请注意传递给构造器的参数本身也是模式。这就能够让你用清晰的编码方式写出深度模式。以下是一个例子：

```
UnOp("-", UnOp("-", e))
```

可以想象一下为了尝试实现同样的功能性需要如何使用访问者设计模式吧[2]！同样还可以想象一下以一长串 if 语句、类型测试及类型转换去实现它有多么的笨拙吧。

match 与 switch 的比较

匹配表达式可以被看作 Java 风格 switch 的泛化。当每个模式都是常量并且最后一个模式可以是通配（表示为 switch 的 default 情况）的时候，Java 风格的 switch 可以被自然地表达为 match 表达式。不过有三点不同要牢记在心。首先，match 是 Scala 的表达式，也就是说，它始终以值作为结果；第二，Scala 的备选项表达式永远不会“掉到”下一个 case；第三，如果没有模式匹配，MatchError 异常会被抛出。这意味着你必须始终确信所有的情况都考虑到了，或者至少意味着可以添加一个默认情况什么事都不做。清单 15.3 举例如下：

```
expr match {
  case BinOp(op, left, right) =>
```

[2]注：Gamma 等人，“Design Patterns”。[Gam95]

```
      println(expr +" is a binary operation")
    case _ =>
  }
```

清单 15.3 带有空“默认”情况的模式匹配

清单 15.3 中的第二个情况是必须的，因为否则的话 match 表达式将在每个 expr 参数不是 BinOp 的时候抛出 MatchError。这个例子里，对于第二个情况没有指定代码，因此如果跑到了这里就什么都不做。每个情况的结果都是 unit 值‘()’，因此这也就是整个 match 表达式的结果。

267

15.2 模式的种类

前一个例子先后展示了若干种模式。现在花点时间把每一种都看一下。

模式的语法很简单，因此不要对此过分担心。所有的模式看上去都与相应的表达式一样。例如，以清单 15.1 的层级关系为前提，模式 Var(x) 能够匹配任何变量表达式，并绑定 x 为变量名。如果 Var(x) 以完全相同的语法形式被用作表达式，并假设 x 已经被绑定到变量名，那它就能够重建出相当的对象。由于模式的语法是如此透明，因此只要关心可能有些什么种类的模式即可。

通配模式

通配模式（_）匹配任意对象。你已经看过它被用作默认的“全匹配（catch-all）”的备选项，如下：

```
expr match {
  case BinOp(op, left, right) =>
    println(expr +"is a binary operation")
  case _ =>
}
```

通配模式还可以用来忽略对象中你不关心的部分。比如说，前一个例子实际上并不关心二元操作符的元素是什么。只是检查是否为二元操作符。因此用通配符指代 BinOp 的元素也是可以的，展示在清单 15.4 中：

```
expr match {
  case BinOp(_, _, _) => println(expr +"is a binary operation")
  case _ => println("It's something else")
}
```

268

清单 15.4 使用通配模式的模式匹配

常量模式

常量模式仅匹配自身。任何字面量都可以用作常量。例如，5、true，还有"hello"都是常量模式。另外，任何的 val 或单例对象也可以被用作常量。例如，单例对象 Nil 是只匹配空列表的模式。清单 15.5 展示了一些常量模式的例子：

```
def describe(x: Any) = x match {
  case 5 => "five"
  case true => "truth"
  case "hello" => "hi!"
```

```
  case Nil => "the empty list"
  case _ => "something else"
}
```

清单 15.5 使用常量模式的模式匹配

以下是清单 15.5 展示的模式匹配的实际效果：

```
scala> describe(5)
res5: java.lang.String = five

scala> describe(true)
res6: java.lang.String = truth

scala> describe("hello")
res7: java.lang.String = hi!

scala> describe(Nil)
res8: java.lang.String = the empty list

scala> describe(List(1,2,3))
res9: java.lang.String = something else
```

变量模式

变量模式类似于通配符，可以匹配任意对象。不过与通配符不同的地方在于，Scala 把变量绑定在匹配的对象上。因此之后你可以使用这个变量操作对象。例如，清单 15.6 展示了一个对零特殊处理，对其他值默认处理的模式匹配。默认的情况使用了变量模式，从而使这个值有了名称，而不管这个值到底是什么。


```
expr match {
  case 0 => "zero"
  case somethingElse => "not zero: "+ somethingElse
}
```

清单 15.6 使用变量模式的模式匹配

变量，还是常量?

常量模式可以有符号名。我们使用 Nil 作为模式时就是这样的情况。以下是一个与之相关的例子，这里模式匹配采用了常量 E（2.71828...）和 Pi（3.14159...）：

```
scala> import Math.{E, Pi}
import Math.{E, Pi}

scala> E match {
         case Pi => "strange math? Pi = "+ Pi
         case _ => "OK"
       }
res10: java.lang.String = OK
```

正如预期的那样，E 不能匹配 Pi，因此“strange math”的情况不会出现。

Scala 编译器是如何知道 Pi 是从 java.lang.Math 对象引入的常量，而不是代表选择器自身值的变量？Scala 使用了一个简单的文字规则对此加以区分：用小写字母开始的简单名被当作是模式变量；所有其他的引用被认为是常量。为了体会其间的差别，你可以创建一个小写字母的别名 pi 并做如下尝试：

```
scala> val pi = Math.Pi
pi: Double = 3.141592653589793
```

```
scala> E match {
         case pi => "strange math? Pi = "+ pi
       }
res11: java.lang.String = strange math? Pi = 2.7182818...
```


在这里编译器甚至都不会让你添加默认状况。因为 pi 是变量模式，它可以匹配任意输入，因此之后的情况没有可访问到的：

```
scala> E match {
         case pi => "strange math? Pi = "+ pi
         case _ => "OK"
       }
<console>:9: error: unreachable code
         case _ => "OK"
                   ^
```

如果有必要的话，你仍然可以通过以下两种手法之一给模式常量使用小写字母名，首先，如果常量是某个对象的字段，可以在其之上用限定符前缀。例如，pi 是变量模式，但是 this.pi 或 obj.pi 虽然都开始于小写字母但都是常量。如果这不起作用（比如说，因为 pi 是本地变量），还可以用反引号包住变量名，例如，`pi`会再次被解释为常量，而不是变量：

```
scala> E match {
         case `pi` => "strange math? Pi = "+ pi
         case _ => "OK"
       }
res13: java.lang.String = OK
```

正如你所见的，Scala 里标识符的反引号语法有两处不同目的的用法以帮助你在非常规的情况下解决编码问题。这里你见到了它可以被用来处理小写字母标识符当作模式匹配常量的问题。更早的，在 6.10 节中，你已经看到了它同样也被用来处理关键字当作普通的标识符的问题，如，Thread.`yield`() 里的 yield 被当作标识符而不是关键字。

构造器模式

构造器（模式）的存在使得模式匹配真正变得强大。构造器模式看上去就像“BinOp("+", e, Number(0))”。它由名称（BinOp）及若干括号之内的模式："+"、e 和 Number(0)构成。假如这个名称指定了一个样本类，那么这个模式就是表示首先检查对象是该名称的样本类的成员，然后检查对象的构造器参数是符合额外提供的模式的。

这些额外的模式意味着 Scala 模式支持深度匹配（deep match）。这种模式不只检查顶层对象是否一致，还会检查对象的内容是否匹配内层的模式。由于额外的模式自身可以形成构造器模式，因此可以使用它们检查到对象内部的任意深度。例如，清单 15.7 的模式检查了对象的顶层是 BinOp，及它的第三个构造器参数是 Number，以及它的值为数字 0。这个模式仅有一行但却能检查三层深度。

```
expr match {
  case BinOp("+", e, Number(0)) => println("a deep match")
  case _ =>
}
```

清单 15.7 带有构造器模式的模式匹配

序列模式

你也可以像匹配样本类那样匹配如 List 或 Array 这样的序列类型。不过同样的语法现在可以指定模式内任意数量的元素。例如，清单 15.8 展示了检查开始于零的三元素列表的模式：

```
expr match {
  case List(0, _, _) => println("found it")
  case _ =>
}
```

清单 15.8 有固定长度的序列模式

如果你想匹配一个不指定长度的序列，可以指定_*作为模式的最后元素。这种古怪的模式能匹配序列中零到任意数量的元素。清单 15.9 展示了匹配由零开始，不计长度的任意列表的模式。

```
expr match {
  case List(0, _*) => println("found it")
  case _ =>
}
```

清单 15.9 匹配任意长度的序列模式

元组模式

你还可以匹配元组。类似(a, b, c)这样的模式可以匹配任意的 3-元组。清单 15.10 是一个例子：

```
def tupleDemo(expr: Any) =
  expr match {
    case (a, b, c) => println("matched "+ a + b + c)
    case _ =>
  }
```

清单 15.10 带有元组模式的模式匹配

把清单 15.10 展示的 tupleDemo 方法加载到解释器里，并传给它有三个元素的元组，就能看到：

```
scala> tupleDemo(("a ", 3, -"tuple"))
matched a 3-tuple
```

类型模式

你可以把类型模式（typed pattern）当做类型测试和类型转换的简易替代。清单 15.11 举例如下：

```
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => 1
}
```

清单 15.11 带有类型化模式的模式匹配

以下是几个在解释器中使用 generalSize 方法的例子：

```
scala> generalSize("abc")
res14: Int = 3
```

```
scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res15: Int = 2

scala> generalSize(Math.Pi)
res16: Int = -1
```

generalSize 方法返回各种类型对象的 size 或 length。它的参数是 Any 类型，因此任何值都可接受。如果参数是 String，方法返回 String 的 length。模式“s: String”是类型模式；匹配所有的（非空）String 实例。模式变量 s 指代了字符串。

请注意，尽管 s 和 x 指代了同样的值，不过 x 是 Any，而 s 是 String。因此可以在模式对应的备选项表达式中写成 s.length，但不能写成 x.length，因为 Any 类型没有 length 成员。

能够得到与类型模式匹配相同效果但更为曲折的方式需要使用类型测试及类型转换。Scala 使用了与 Java 不同的语法。比如说，为了测试表达式 expr 是 String 类型的，你得这么写：

```
expr.isInstanceOf[String]
```

要转换同样的表达式为类型 String，你要写成：

```
expr.asInstanceOf[String]
```

使用类型测试和转换，你就可以重写之前 match 表达式的第一个例子，展示在清单 15.12 中。

```
if (x.isInstanceOf[String]) {
  val s = x.asInstanceOf[String]
  s.length
} else ...
```

清单 15.12 使用 isInstanceOf 和 asInstanceOf（不好的风格）

操作符 isInstanceOf 和 asInstanceOf 被当作了带了类型参数（方括号内）的 Any 类预定义方法。实际上，x.asInstanceOf[String]是一个使用显式类型参数 String 的方法调用的具体例子。

到现在你应该已经注意到，Scala 里类型测试和转换的代码真的很冗长。这是故意的，因为我们不鼓励这么做。使用带有类型模式的模式匹配通常就能满足你的要求，尤其在你需要同时做类型测试和转换的场景时，因为这两种操作都被融入一个模式匹配之中了。

上述 match 表达式的第二个例子包含了类型模式“m: Map[_, _]”。这个模式匹配各种包含任意键/值类型的 Map 类的值并用 m 指代这个值。因此，m.size 的写法正确并将返回映射的 size。类型模式里的下划线就像其他模式里的通配符一样。你也可以使用（小写字母）类型变量替代。

类型擦除

特定元素类型的映射能匹配吗？这会更有用，比如说测试给定值是否是从 Int 到 Int 的映射。让我们试一下：

```
scala> def isIntIntMap(x: Any) = x match {
         case m: Map[Int, Int] => true
         case _ => false
       }
warning: there were unchecked warnings; re-run with
```

```
  -unchecked for details[3]
isIntIntMap: (Any)Boolean
```

解释器发出了“未检查警告”。你可以使用-unchecked 命令行选项重新开始解释器来寻求更多细节说明：

```
scala> :quit
$ scala -unchecked
Welcome to Scala version 2.7.2
(Java HotSpot(TM) Client VM, Java 1.5.0_13).
Type in expressions to have them evaluated.
Type :help for more information.
scala> def isIntIntMap(x: Any) = x match {
         case m: Map[Int, Int] => true
         case _ => false
       }
  <console>:5: warning: non variable type-argument Int in
  type pattern is unchecked since it is eliminated by erasure[4]
         case m: Map[Int, Int] => true
                 ^
```

Scala 使用了泛型的擦除（erasure）模式，就如 Java 那样。也就是说类型参数信息没有保留到运行期。因此，运行期没有办法判断给定的 Map 对象创建时带了两个 Int 参数还是其他的什么类型。系统所能做的只是判断这个值是某种任意类型参数的 Map。你可以通过对 isIntIntMap 调用不同的 Map 实例来证实这点：

```
scala> isIntIntMap(Map(1 -> 1))
res17: Boolean = true

scala> isIntIntMap(Map("abc" -> "abc"))
res18: Boolean = true
```

第一个应用返回 true，看上去很正确，但是第二个同样返回 true，就有点儿让人惊讶了。为了警告你可能存在的这种非直观的运行期行为，编译器就会如之前看到的那样发出未检查警告。

擦除规则的唯一例外就是数组，因为在 Scala 里和 Java 里，它们都被特殊处理了。数组的元素类型与数组值保存在一起，因此它可以做模式匹配。如下所示：

```
scala> def isStringArray(x: Any) = x match {
         case a: Array[String] => "yes"
         case _ => "no"
       }
isStringArray: (Any)java.lang.String

scala> val as = Array("abc")
as: Array[java.lang.String] = Array(abc)

scala> isStringArray(as)
res19: java.lang.String = yes

scala> val ai = Array(1, 2, 3)
ai: Array[Int] = Array(1, 2, 3)

scala> isStringArray(ai)
res20: java.lang.String = no
```

[3]译注：有未检查的警告；使用-uncheck 重新运行可以得到细节信息。

[4]译注：警告：未检查类型模式中的非变量类型参数 Int，因为它将在擦除过程中被抹去。

变量绑定

除了独立的变量模式之外，你还可以对任何其他模式添加变量。只要简单地写上变量名、一个@符号，以及这个模式。这种写法创造了变量绑定模式。这种模式的意义在于它能像通常的那样做模式匹配，并且如果匹配成功，则把变量设置成匹配的对象，就像使用简单的变量模式那样。

举个例子，清单 15.13 展示了为寻找一行中使用了两遍绝对值操作符的模式匹配。这样的表达式可以被简化为仅使用一次绝对值操作。

```
expr match {
  case UnOp("abs", e @ UnOp("abs", _)) => e
  case _ =>
}
```

清单 15.13 带有变量绑定（用@符号）的模式

清单 15.13 里有一个用 e 作为变量及 UnOp("abs", _) 作为模式的变量绑定模式。如果整个模式匹配成功，那么符合 UnOp("abs", _) 的部分就可以使用 e 指代。正如代码中写的，之后 e 就会保持原样被返回。

15.3 模式守卫

有些时候，语法的模式匹配还不够精确。例如，假如说你被指派了一份工作去制定一个简化规则以通过乘二运算，也就是说，e*2 替代两个相同操作元的相加，如 e+e。表示成 Expr 树的语言，就是表达式：

```
BinOp("+", Var("x"), Var("x"))
```

将通过此规则被转换为：

```
BinOp("*", Var("x"), Number(2))
```

或许你会尝试以如下的方式定义该规则：

```
scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, x) => BinOp("*", x, Number(2))
         case _ => e
       }
<console>:10: error: x is already defined as value x
         case BinOp("+", x, x) => BinOp("*", x, Number(2))
                            ^
```

失败了，因为 Scala 要求模式是线性的：模式变量仅允许在模式中出现一次。不过，你可以使用模式守卫（pattern guard）重新制定这个匹配规则，清单 15.14 展示了做法：

```
scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, y) if x == y =>
           BinOp("*", x, Number(2))
         case _ => e
       }
simplifyAdd: (Expr)Expr
```

清单 15.14 带有模式守卫的 match 表达式

模式守卫接在模式之后，开始于 if。守卫可以是任意的引用模式中变量的布尔表达式。如果存在模式守卫，那么只有在守卫返回 true 的时候匹配才成功。因此，上面的第一个例子只会匹配带有两个相同操作元的二元操作。

带守卫的模式的其他例子包括：

```
// 仅匹配正整数
case n: Int if 0 < n => ...

// 仅匹配以字母'a'开始的字符串
case s: String if s(0) == 'a' => ...
```

278

15.4 模式重叠

模式以代码编写的先后次序尝试。在清单 15.15 中 simplify 的这个版本给出了须考虑样本次序的例子：

```
def simplifyAll(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) =>
    simplifyAll(e)   // '-'是自身的反转
  case BinOp("+", e, Number(0)) =>
    simplifyAll(e)   // '0'对于'+'来说不改变结果
  case BinOp("*", e, Number(1)) =>
    simplifyAll(e)   // '1'对于'*'来说不改变结果
  case UnOp(op, e) =>
    UnOp(op, simplifyAll(e))
  case BinOp(op, l, r) =>
    BinOp(op, simplifyAll(l), simplifyAll(r))
  case _ => expr
}
```

清单 15.15 样本依次被调用的匹配表达式

清单 15.15 里的这个 simplify 版本将对表达式的任何地方应用简化规则，而不仅仅是如 simplifyTop 做的在最顶层。它发源于 simplifyTop，不过多添加了两个样本处理通常的一元和二元表达式（清单 15.15 中的样本四和五）。

第四个样本包含了模式 UnOp(op, e)；等内容，它匹配任何一元操作。操作符和操作元任选，它们相应地绑定为模式变量 op 和 e。这个样本中的可选表达式对操作元 e 递归调用了 simplifyAll 方法并使用（可能是）简化了的操作元重建同样的一元操作。第五个样本对 BinOp 是类似的：它是任意二元操作的“全匹配”，并且对两个操作元递归调用了简化方法。

这个例子中有一点很重要，就是全匹配的样本要跟在更具体的简化方法之后。如果写成其他次序，那么全匹配样本将比特定规则获得更高的优先级。在许多情况下，编译器将在你做这种尝试的时候发出警告。

279

例如，下面的 match 表达式不会编译成功，因为第一个样本匹配任何能匹配第二个样本的东西：

```
scala> def simplifyBad(expr: Expr): Expr = expr match {
         case UnOp(op, e) => UnOp(op, simplifyBad(e))
         case UnOp("-", UnOp("-", e)) => e
       }
```

```
<console>:17: error: unreachable code
         case UnOp("-", UnOp("-", e)) => e
                                         ^
```

15.5 封闭类

一旦你写好了模式匹配，你就需要确信已经考虑到了所有可能的情况。有些时候你可以通过在匹配的最后添加默认处理做到这点，不过这仅仅在的确有一个合理的默认行为的情况下有效。如果没有默认的情况该怎么办？你怎样才能保证包括了所有的情况呢？

实际上，你可以让 Scala 编译器帮助你检测 `match` 表达式中遗漏的模式组合。要做到这点，编译器就需要搞清楚哪些情况是可能的。通常，这在 Scala 里是不可能的，因为新的样本类可以定义在任何时刻及任意编译单元中。比方说，什么都无法阻止你在不同于 `Expr` 其他四个已定义样本的编译单元中，向 `Expr` 的类层级新加第五个样本类。

可选方案就是让样本类的超类被封闭（sealed）。封闭类除了类定义所在的文件之外不能再添加任何新的子类。这对于模式匹配来说是非常有用的，因为这意味着你仅需要关心你已经知道的子类即可。这还意味着你可以获得更好的编译器帮助。如果你使用继承自封闭类的样本类做匹配，编译器将通过警告信息标志出缺失的模式组合。

因此，如果要写打算做模式匹配的类层级，你应当考虑封闭它们。只要把关键字 `sealed` 放在最顶层类的前边即可。使用你的类层级的程序员于是就能放心地使用它做模式匹配。程序清单 15.16 展示了 `Expr` 变为封闭类的例子。

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

清单 15.16 封闭了的样本类层级

现在定义一个丢失了若干可能样本的模式匹配：

```
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_)    => "a variable"
}
```

你会得到如下的编译器警告：

```
warning: match is not exhaustive!
missing combination          UnOp
missing combination          BinOp
```

这样的警告向你表明你的代码会有产生 `MatchError` 异常的风险，因为某些可能的模式（`UnOp`，`BinOp`）没有被处理。警告指出了潜在的运行期故障的源头，因此它通常能够帮助你正确的编程。

然而，有些时候你或许会碰到这样的情况，编译器弹出太过挑剔的警告。例如，或许可以通过上下文知道你只会把上面的 `describe` 方法应用在仅仅是 `Number` 或 `Var` 的表达式上。因此你知道实际不

会产生 MatchError。要让这些警告不再发生，你可以为方法添加用作全匹配的第三个样本，如下：

```
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_) => "a variable"
  case _ => throw new RuntimeException // 不会发生
}
```

这样做的确有效，但不理想。你或许对被迫添加不会被执行（或至少你这么认为），而只是让编译器闭嘴的代码感到失望。

更轻量级的做法是给匹配的选择器表达式添加@unchecked 注解。如下：

```
def describe(e: Expr): String = (e: @unchecked) match {
  case Number(_) => "a number"
  case Var(_)    => "a variable"
}
```

注解将在第 25 章描述。通常，你可以用类似于添加类型的方式给表达式添加注解：表达式后跟一个冒号及注解的名称（前缀@符号）。比方说，本例中使用“e: @unchecked”方法为变量 e 添加了@unchecked 注解。@unchecked 注解对于模式匹配来说有特定的意思。如果 match 的选择器表达式带有这个注解，那么对于随后的模式的穷举性检查将被抑制掉。

15.6 Option 类型

Scala 为可选值定义了一个名为 Option 的标准类型。这种值可以有两种形式。可以是 Some(x) 的形式，其中 x 是实际值，或者也可以是 None 对象，代表缺失的值。

Scala 集合类的某些标准操作会产生可选值。例如，Scala 的 Map 的 get 方法会在发现了指定键的情况下产生 Some(value)，在没有找到指定键的时候产生 None。举例如下：

```
scala> val capitals =
         Map("France" -> "Paris", "Japan" -> "Tokyo")
capitals:
  scala.collection.immutable.Map[java.lang.String,    ,
  java.lang.String] = Map(France -> Paris, Japan -> Tokyo)

scala> capitals get "France"
res21: Option[java.lang.String] = Some(Paris)

scala> capitals get "North Pole"
res22: Option[java.lang.String] = None
```

分离可选值最通常的办法是通过模式匹配。例如：

```
scala> def show(x: Option[String]) = x match {
         case Some(s) => s
         case None => "?"
       }
show: (Option[String])String

scala> show(capitals get "Japan")
res23: String = Tokyo

scala> show(capitals get "France")
res24: String = Paris
```

```
scala> show(capitals get "North Pole")
res25: String = ?
```

Option 类型在 Scala 编程中常会用到。与之相较，Java 里最常用的是代表没有值的 null。例如，java.util.HashMap 的 get 方法要么返回存储在 HashMap 里的值，要么没找到返回 null。这种方式对 Java 起效，不过可能会隐藏错误，因为很难实际记得住程序中哪个变量可以允许是 null。如果变量允许是 null，那你就必须得记住每次使用它的时候检查是否为 null。一旦你忘记了检查，就难以避免运行时发生 NullPointerException 异常。又因为这种异常可能不是经常发生，所以想要通过测试发现故障是非常困难的。对于 Scala 来说，这种方式根本不起作用，因为可以在哈希映射（hashmap）中存储值类型，而 null 不是值类型的合法元素。举例来说，HashMap[Int, Int]不能返回 null 以表明“没有元素”。

相反，Scala 鼓励对 Option 的使用以说明值是可选的。这种处理可选值的方式有若干超越 Java 的优点。首先，对于代码读者来说，Option[String]类型的变量是可选的 String，这比 String 类型的变量或可能有时是 null 来说要更为明显。但最重要的是，之前描述的因为使用可能为 null 而没有首先检查是否为 null 的变量产生的编程错误在 Scala 里变为类型错误。如果变量是 Option[String] 类型的，而你打算当做 String 使用，你的 Scala 程序就不会编译通过。

283

15.7 模式无处不在

在 Scala 中模式可以出现在很多地方，而不单单在 match 表达式里。现在看一下你可以使用模式的其他地方。

模式在变量定义中

你在定义 val 或 var 的任何时候，都可以使用模式替代简单的标识符。例如，你可以使用模式拆分元组并把其中的每个值分配给变量，如清单 15.17 所示：

```
scala> val myTuple = (123, "abc")
myTuple: (Int, java.lang.String) = (123,abc)

scala> val (number, string) = myTuple
number: Int = 123
string: java.lang.String = abc
```

清单 15.17 用一个赋值语句定义多个变量

使用样本类时这种构造非常有用。如果你知道正在用的样本类的精确结构，那就可以使用模式解构它。举例如下：

```
scala> val exp = new BinOp("*", Number(5), Number(1))
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))

scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
```

用作偏函数的样本序列

花括号内的样本序列（就是说，备选项）可以用在能够出现函数字面量的任何地方。实质上，样本序列就是函数字面量，而且只有更普遍。函数字面量只有一个入口点和参数列表，但样本序列可以有多个入口点，每个都有自己的参数列表。每个样本都是函数的一个入口点，参数也被模式所特化。每个入口点的函数体都在样本的右侧。

284

举例如下：

```
val withDefault: Option[Int] => Int = {
  case Some(x) => x
  case None => 0
}
```

这个函数的函数体有两个样本。第一个样本匹配 Some，并返回 Some 内的数字。第二个样本匹配 None，返回零作为默认值。以下是这个函数的用法：

```
scala> withDefault(Some(10))
res25: Int = 10

scala> withDefault(None)
res26: Int = 0
```

这种设置对于 Actor 库非常有用，将在第 30 章描述。以下是一些典型的 Actor 的代码。它直接把模式匹配传递给了 react 方法：

```
react {
  case (name: String, actor: Actor) => {
    actor ! getip(name)
    act()
  }
  case msg => {
    println("Unhandled message: "+ msg)
    act()
  }
}
```

另一种通用化方案就不值一提了：样本序列可以用作这样的偏（partial）函数，如果你把函数应用在它不支持的值上，将会产生一个运行期异常。例如，下面的偏函数能够返回整数列表的第二个元素：

```
val second: List[Int] => Int = {
  case x :: y :: _ => y
}
```

285

如果你编译它，编译器会正确地提示说匹配并不全面：

```
<console>:17: warning: match is not exhaustive!
missing combination            Nil
```

如果你传给函数一个三元素的列表，它的执行没有问题，但是如果传给它一个空列表就不行了：

```
scala> second(List(5,6,7))
res24: Int = 6

scala> second(List())
scala.MatchError: List()
      at $anonfun$1.apply(<console>:17)
      at $anonfun$1.apply(<console>:17)
```

如果你想要检查是否一个偏函数有定义，你必须首先告诉编译器你知道正在使用的是偏函数。类型 List[Int] => Int 包含了不管是否为偏函数的，从整数列表到整数的所有函数。仅包含从整数列表到整数的偏函数的，应该写成 PartialFunction[List[Int], Int]。下面还是 second 函数，这次写成了使用偏函数类型：

```
val second: PartialFunction[List[Int],Int] = {
  case x :: y :: _ => y
}
```

偏函数有一个 isDefinedAt 方法，可以用来测试是否函数对某个特定值有定义。本例中，函数对任何有至少两个元素的列表有定义：

```
scala> second.isDefinedAt(List(5,6,7))
res27: Boolean = true

scala> second.isDefinedAt(List())
res28: Boolean = false
```

286

典型的偏函数例子是前面例子里的那种模式匹配函数字面量。实际上，Scala 编译器把这样的表达式转译成偏函数的时候， 会对模式执行两次翻译——其中一次是真实函数的实现，另一次是测试函数是否（对特定参数）有定义的实现。例如，上面的函数字面量{case x :: y :: _ => y}会被翻译成下列的偏函数值：

```
new PartialFunction[List[Int], Int] {
  def apply(xs: List[Int]) = xs match {
    case x :: y :: _ => y
  }
  def isDefinedAt(xs: List[Int]) = xs match {
    case x :: y :: _ => true
    case _ => false
  }
}
```

这种翻译只有在函数字面量的声明类型为 PartialFunction 的时候才起效。如果声明的类型只是 Function1，或根本不存在，那么函数字面量就会代而转译为完整的函数。

通常，在可能的情况下你应该尝试使用完整函数，因为使用偏函数的话，编译器没办法帮你避免运行期故障的出现。不过有些时候偏函数的确很有用。你应该确信永远不会对它调用未处理的值。或者，你可能正在使用一个需要偏函数的架构，并且如果是这样的话，它会在调用函数之前始终检查 isDefinedAt。后者的一个例子是前面给出的 react 例子，那里的参数是一个部分定义的函数，仅精确定义了那些调用者想要处理的消息。

for 表达式里的模式

for 表达式里也可以使用模式。如清单 15.18 所示。这个 for 表达式从 capitals 映射中获得所有的键/值对。每一对都匹配于模式(country, city)，并定义了两个变量 country 和 city。

```
scala> for ((country, city) <- capitals)
         println("The capital of "+ country +" is "+ city)
The capital of France is Paris
The capital of Japan is Tokyo
```

清单 15.18 带有元组模式的 for 表达式

清单 15.18 演示的配对模式是特殊的，因为对它的匹配不会失败。实际上，capitals 生成了一系列的对偶，因此你可以确信产生的每个对偶都能够匹配这个配对模式。

不过模式同样也可能无法匹配产生的值。清单 15.19 展示了这种情况的一个例子：

```
scala> val results = List(Some("apple"), None,
           Some("orange"))
results: List[Option[java.lang.String]] = List(Some(apple),
    None, Some(orange))
scala> for (Some(fruit) <- results) println(fruit)
apple
orange
```

清单 15.19 从列表中拣出匹配于模式的元素

如你在这个例子中所见，产生出来的不能匹配于模式的值被丢弃。例如，results 列表里第二个元素 None 不匹配模式 Some(fruit)，因此就没有出现在输出中。

15.8 一个更大的例子

学习了模式的不同形式之后，你或许会对看到它们应用在一个更大的例子里感兴趣。值得推荐的任务就是写一个表达式的格式化类，能够以二维的布局显示一个数学表达式。如“x / x + 1”这样的除法，应该能够通过把被除数放在除数的顶上这种方式，被垂直打印如下：

```
  x
-----
x + 1
```

另一个例子是，表达式((a / (b * c) + 1 / n) / 3)的二维布局：

```
    a     1
  ----- + -
  b * c   n
 ---------
     3
```

这个例子看上去好像类（我们称它为 ExprFormatter）会做大量的布局判断，因此明智的做法是使用第 10 章开发的布局库。我们还会使用本章之前看过的样本类 Expr 家族，并把第 10 章的布局库和本章的表达式格式化器放在命名包里。例子的完整代码展示在清单 15.20 和清单 15.21 里。

第一步应该是集中注意力在水平布局上。结构化的表达式如：

```
BinOp("+",
      BinOp("*",
            BinOp("+", Var("x"), Var("y")),
            Var("z")),
      Number(1))
```

应该打印成(x+y)*z+1。注意 x+y 两边的小括号是必加的，不过(x+y)*z 的两边就是可选的了。为了保证布局最好的可读性，你的目标是应当去掉冗余的括号，并保留所有必须存在的括号。

想要知道哪里应该加括号，代码必须知道每个操作符相对的优先级，所以首先处理它是个好主意。你可以用如下形式的映射字面量直接表达相对优先级：

```
Map(
  "|" -> 0, "||" -> 0,
  "&" -> 1, "&&" -> 1, ...
)
```

289

然而，这会在你这部分代码中引入大量对优先级的预计算。更方便的方式是只定义递增优先级的操作符组然后通过它计算每个操作符的优先级。展示在清单 15.20 中。

precedence 变量是从操作符到它们的优先级的映射，是开始于 0 的整数。它是通过使用带有两个生成器的 for 表达式计算出来的。第一个生成器产生 opGroups 数组的每个索引 i。第二个生成器产生 opGroups(i) 里的每个操作符 op。for 表达式对每个这样的操作符创造一个从 op 操作符到它的索引 i 的关联。因此，操作符在数组里的相对位置就被作为它的优先级取了出来。关联被写作内联的箭头就是，op -> i。到这里你看到的只是这些关联作为映射结构的部分，但是它们同样也是它们自身的值。实际上，关联 op -> i 不外乎就是对偶(op, i)。

现在你已经补全了除了/之外所有二元操作符的优先级关系，是时候把这个概念推广到同时包含一元操作符的情况下了。一元操作符的优先级高于任何的二元操作符。因此我们可以把 opGroups 数组的长度设为 unaryPrecedence（参见清单 15.20），它比*和%操作元的优先级大 1。

分数的优先级与其他操作符的处理方式不一致，因为分数使用了垂直布局。然而，把除法操作符的优先级设定为-1 将会被证明是很方便的，因此我们初始化 fractionPrecedence 为-1（参见清单 15.20）。

290

```
    case Ver(name) =>
      elem(name)
```

```
package org.stairwaybook.expr
import layout.Element.elem

sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr

class ExprFormatter {

  // 包含了递增优先级的组中的操作符
  private val opGroups =
    Array(
      Set("|", "||"),
      Set("&", "&&"),
      Set("^"),
      Set("==", "!="),
      Set("<", "<=", ">", ">="),
      Set("+", "-"),
      Set("*", "%")
    )
  // 操作符到优先级的映射
```

```
private val precedence = {
  val assocs =
    for {
      i <- 0 until opGroups.length
      op <- opGroups(i)
    } yield op -> i
  Map() ++ assocs
}

private val unaryPrecedence = opGroups.length
private val fractionPrecedence = -1
// 清单 15.21 中继续...
```

清单 15.20 表达式格式化器的上半截

经过了这些准备之后，你现在可以开始编写主方法 format 了。这个方法带有两个参数：表达式 e，类型为 Expr，以及紧贴表达式 e 的操作符的优先级 enclPrec（如果没有这样的操作符，enclPrec 就应该是零）。这个方法可以产生表达字符的二维数组的布局元素。

清单 15.21 展示了类 ExprFormatter 的剩余部分，其中包含了三个方法。第一个方法 stripDot 是一个助手方法。下一个方法是私有的 format 方法，它完成了格式化表达式的大部分工作。最后的方法，同样也叫 format，是库唯一的公开方法，它以需要格式化的表达式为参数。

```
// ...上接清单 15.20

private def format(e: Expr, enclPrec: Int): Element =
  e match {
  case Var(name) =>
    elem(name)

  case Number(num) =>
    def stripDot(s: String) =
      if (s endsWith ".0") s.substring(0, s.length -2)
      else s
    elem(stripDot(num.toString))

  case UnOp(op, arg) =>
    elem(op) beside format(arg, unaryPrecedence)

  case BinOp("/", left, right) =>
    val top = format(left, fractionPrecedence)
    val bot = format(right, fractionPrecedence)
    val line = elem('-', top.width max bot.width, 1)
    val frac = top above line above bot
    if (enclPrec != fractionPrecedence) frac
    else elem(" ") beside frac beside elem(" ")

  case BinOp(op, left, right) =>
    val opPrec = precedence(op)
    val l = format(left, opPrec)
    val r = format(right, opPrec + 1)
    val oper = l beside elem(" "+ op +" ") beside r
    if (enclPrec <= opPrec) oper
    else elem("(") beside oper beside elem(")")
}
```

```
    def format(e: Expr): Element = format(e, 0)
  }
```

292

清单 15.21 表达式格式化器的下半截

私有的 format 方法通过对表达式的类型执行模式匹配来完成它的工作。match 表达式有五个样本。我们会逐一讨论每个样本。第一个是：

```
case Var(name) =>
  elem(name)
```

如果表达式是变量，结果是由变量名组成的元素。

第二个样本是：

```
case Number(num) =>
  def stripDot(s: String) =
    if (s endsWith ".0") s.substring(0, s.length -2)
    else s
  elem(stripDot(num.toString))
```

如果表达式是数字，结果就是由数字的值组成的元素。stripDot 函数通过从字符串中除去任何".0"后缀的方式清理浮点数的显示。

第三个样本是：

```
case UnOp(op, arg) =>
  elem(op) beside format(arg, unaryPrecedence)
```

如果表达式是一元操作 UnOp(op, arg)，结果就由操作 op 和最高环境优先级格式化参数 arg 的结果组成。[5]这意味着如果 arg 是二元操作（但不是分数），它就始终会显示在括号中。

第四个样本是：

```
case BinOp("/", left, right) =>
  val top = format(left, fractionPrecedence)
  val bot = format(right, fractionPrecedence)
  val line = elem('-', top.width max bot.width, 1)
  val frac = top above line above bot
  if (enclPrec != fractionPrecedence) frac
  else elem(" ") beside frac beside elem(" ")
```

293

如果表达式是分数，中间结果 frac 就由格式化了的操作元 left 和 right 垂直放置，中间用一条水平线元素分隔来构成。水平线的宽度是格式化的操作元宽度的最大值。除非这个分数还是另外一个分数的参数，否则这个中间结果也就是最终结果。在最后提到的情况里，frac 的两边各加一个空格。为了说明这么做的理由，请考虑表达式“(a / b) / c”。没有变宽的纠正，格式化这个表达式就会给出：

```
a
-
b
-
c
```

[5]注：unaryPrecedence 的值是可能出现的最高优先级，因为它被初始化为比*和%操作符的优先级高 1。

这个布局的问题很明显——不清楚哪一个是主分数线。上面的表达式既可以是“(a/b)/c”也可以是“a/(b/c)”。为了消除这种含糊的语义，内嵌的分数“a/b”的布局两边应该各加一个空格。这样一来布局的语义就清晰了：

```
  a
  -
  b
- - -
  c
```

第五个也是最后一个样本：

```
case BinOp(op, left, right) =>
  val opPrec = precedence(op)
  val l = format(left, opPrec)
  val r = format(right, opPrec + 1)
  val oper = l beside elem(" "+ op +" ") beside r
  if (enclPrec <= opPrec) oper
  else elem("(") beside oper beside elem(")")
```

这个样本可以应用到所有其他的二元操作。因此它在这个样本之后：

```
case BinOp("/", left, right) => ...
```

你应该知道模式 BinOp(op, left, right)里的操作符 op 不会是除法。要格式化这种二元操作，首先要格式化它的操作元 left 和 right。格式化左操作元的优先级参数是操作符 op 的 opPrec，而格式化右操作元是这个优先级加 1。这种设计保证了括号也同样反映正确的联合性。如操作：

```
BinOp("-", Var("a"), BinOp("-", Var("b"), Var("c")))
```

将被正确地分划成“a - (b - c)”。之后中间结果 oper 由格式化了的左右操作元依次放好，中间加操作符构成。如果当前操作符的优先级小于外围操作符的优先级，那么 oper 就被放在括号当中，否则就按原样返回。

这里结束了私有函数 format 的设计。唯一剩下的是可以让客户程序员不提供优先级参数就格式化顶层表达式的公开方法 format。清单 15.22 展示了试运行 ExprFormatter 的演示程序：

```
import org.stairwaybook.expr._

object Express extends Application {

  val f = new ExprFormatter

  val e1 = BinOp("*", BinOp("/", Number(1), Number(2)),
                      BinOp("+", Var("x"), Number(1)))
  val e2 = BinOp("+", BinOp("/", Var("x"), Number(2)),
                      BinOp("/", Number(1.5), Var("x")))
  val e3 = BinOp("/", e1, e2)

  def show(e: Expr) = println(f.format(e)+ "\n\n")

  for (val e <- Array(e1, e2, e3)) show(e)
}
```

清单 15.22 输出格式化了的表达式的应用程序

注意，尽管这个程序没有定义 main 方法，但它仍然是可以运行的应用程序，因为它继承自 Application 特质。正如在 4.5 节中提到的，Application 特质只定义了一个空的 main 方法，被 Express 对象继承。Express 对象的实际工作在对象初始化过程中就完成了，还早于 main 方法的运行。这也就是只有你的程序不带任何命令行参数的时候才能使用这一技巧的原因。一旦有了参数，你就需要显式地写出 main 方法。你可以使用以下命令执行 Express 程序：

```
scala Express
```

它会产生如下的输出：

```
  1
- * (x + 1)
  2

    x   1.5
  - + ---
    2    x

  1
- * (x + 1)
  2
-----------
    x   1.5
  - + ---
    2    x
```

15.9 小结

本章中，你学到了 Scala 的样本类和模式匹配的细节。有了它们，你得以使用若干通常不会存在于面向对象语言的简洁用法。然而 Scala 的模式匹配比本章中所描述的更为深远。如果你想在你的一个类里使用模式匹配，但又不想采用样本类所做的方式那样开放你代码的访问权限，那么你可以使用第 24 章描述的抽取器（extractor）。不过下一章的内容，我们将会把注意力转移到列表上。

第 16 章

使用列表

列表或许是 Scala 程序中最常用到的数据结构了。本章说明了列表的细节。并列举了许多可以在列表上执行的常用操作。而且还传授了一些使用列表编程过程中要注意的重要设计原则。

16.1 列表字面量

在之前的章节中你已经看到过列表，因此你应该知道包含了元素'a'，'b'，'c'的列表写成了List('a', 'b', 'c')。以下是另外一些例子：

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty = List()
```

列表与数组非常相似，不过有两点重要的差别。首先，列表是不可变的。也就是说，不能通过赋值改变列表的元素。其次，列表具有递归结构（例如，链接列表：linked list）[1]，而数组是连续的。

16.2 List 类型

就像数组一样，列表是同质（homogeneous）的：列表的所有元素都具有相同的类型。元素类型为T的列表类型写成List[T]。例如，以下四个列表的例子都明确指定了类型：

```
val fruit: List[String] = List("apples", "oranges", "pears")
val nums: List[Int] = List(1, 2, 3, 4)
val diag3: List[List[Int]] =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty: List[Nothing] = List()
```

[1]注：List 的图形描述，参见图 22.2。

Scala 里的列表类型是协变（covariant）的。这意味着对于每一对类型 S 和 T 来说，如果 S 是 T 的子类型，那么 List[S]是 List[T]的子类型。比如说，List[String]是 List[Object]的子类型。这很自然，因为每个字符串列表都同样可以被看作是对象列表[2]。

注意空列表的类型为 List[Nothing]。你应该在 11.3 节看到过 Nothing 是 Scala 的类层级的底层类型。它是每个 Scala 类型的子类。因为列表是协变的，所以对于任意类型 T 的 List[T]来说，List[Nothing]都是其子类。因此类型为 List[Nothing]的空列表对象，还可以被当作是其他任何形式为 List[T]的列表类型的对象。这也就是如下代码完全合法的原因：

```
// List()同样也是List[String]的!
val xs: List[String] = List()
```

16.3 构造列表

所有的列表都是由两个基础构造块 Nil 和::（发音为“cons”）构造出来的。Nil 代表空列表。中缀操作符::，表示列表从前端扩展。也就是说，x :: xs 代表了第一个元素为 x，后面跟着列表 xs（的元素）的列表。因此，之前的列表值也可以用如下的方式定义：

298

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
            (0 :: (1 :: (0 :: Nil))) ::
            (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

实际上之前以 List(...)形式对 fruit、nums、diag3 和 empty 的定义只不过是扩展为这些定义的包装。例如，List(1, 2, 3)创建了列表 1 :: (2 :: (3 :: Nil))。

由于以冒号结尾，::操作遵循右结合规则：A :: B :: C 等同于 A :: (B :: C)。因此，你可以去掉前面定义里用到的括号。例如：

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

与前面 nums 的定义一致。

16.4 列表的基本操作

对于列表的所有操作都可以表达为以下三种形式：

head	返回列表的第一个元素。
tail	返回除第一个之外所有元素组成的列表。
isEmpty	如果列表为空，则返回真。

这些操作都定义为 List 类的方法。表 16.1 中包含了一部分例子。

[2]注：第 19 章提供了更多关于协变及其他类型变化的细节。

表 16.1 基本列表操作

操作	行为
empty.isEmpty	返回 true
fruit.isEmpty	返回 false
fruit.head	返回"apples"
fruit.tail.head	返回"oranges"
diag3.head	返回 List(1, 0, 0)

head 和 tail 方法仅能够作用在非空列表上。如果执行在空列表上，会抛出异常。如：

```
scala> Nil.head
java.util.NoSuchElementException: head of empty list
```

作为列表处理操作的例子，可以考虑把数值列表以升序方式排列，较简单的做法是插入排序（insertion sort）。工作的方式如下：排序非空列表 x :: xs，可以先排序列表 xs，然后把 x 插入正确的地方；而对空列表的排序结果还是空列表。用 Scala 代码来表达的话，插入排序算法大致如下：

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]): List[Int] =
  if (xs.isEmpty || x <= xs.head) x :: xs
  else xs.head :: insert(x, xs.tail)
```


16.5 列表模式

列表还可以使用模式匹配做拆分，这时列表模式需逐一匹配要拆分的列表表达式。你既可以用 List(...)形式的模式对列表所有的元素做匹配，也可以用::操作符和 Nil 常量组成的模式逐位拆分列表。

下面的例子演示了第一种模式类型：

```
scala> val List(a, b, c) = fruit
a: String = apples
b: String = oranges
c: String = pears
```

模式 List(a, b, c)与列表的长度（3）匹配，并把它的三个元素分别绑定到模式变量 a，b 和 c 上。如果起先并不知道列表元素的数量，那么最好还是使用::做匹配。例如，模式 a :: b :: rest 可以匹配长度至少为 2 的列表：

```
scala> val a :: b :: rest = fruit
a: String = apples
b: String = oranges
rest: List[String] = List(pears)
```

List 的模式匹配

如果你回顾了第 15 章里解释过的模式所有可能形式的话，你或许会发现无论是 List(...)还是::看上去都不符合之前定义的模式类型。实际上，List(...)是由开发库定义的抽取器（extractor）模式的实例。这种模式将在第 24 章中说明。“cons”模式 x :: xs 是中缀操作符模式的特例。你应该已经知道，如果被看做是表达式，那么中缀操作与方法调用等价。但对于模式来说规则有些不同：如果被当作模式，那么类似于 p op q 这样的中缀操作符等价于 op(p, q)。也就是说，中缀操作符 op 被当作构造器模式。在这里，x :: xs 这样的 cons 模式被看作::(x, xs)。这么说，是否会有符合这种构造器模式的名为::的类呢？实际上也的确存在这个类。全称是 scala.::，它是可以创建非空列表的类。所以 Scala 中存在两个::，一个存在于 scala 包中，另一个是 List 类的方法。::方法的目的是实例化 scala.::的对象。你将在第 22 章发现更多列表类的实现细节。

采用模式匹配拆分列表是与采用基本方法 head、tail 和 isEmpty 做拆分一致的可选方案。比如说，以下的代码同样实现了插入排序，但这次使用了模式匹配：

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List()   => List()
  case x :: xs1 => insert(x, isort(xs1))
}

def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List()   => List(x)
  case y :: ys  => if (x <= y) x :: xs
                   else y :: insert(x, ys)
}
```

通常，采用模式匹配做拆分会比使用那些基本方法更为清晰，因此你的列表处理工具箱中应该加入模式匹配这样工具。

以上就是为了要正确使用 Scala 的列表所需的全部知识。然而，在日常使用时，仍然还有大量方法需要掌握。这些方法能让列表的处理更为简明和清晰。以下的两节说明了 List 类定义中最重要的方法。

16.6 List 类的一阶方法

本节将解释 List 类中定义的绝大多数一阶（first-order）方法。一阶方法是指不以函数做入参的方法。本节还将通过两个例子的说明推荐一些构造列表操作程序的技术。

连接列表

连接操作是与::接近的一种操作，写做“:::”。不过不像::，:::的两个操作元都是列表。xs ::: ys 的结果是依次包含 xs 和 ys 所有元素的新列表。举例如下：

```
scala> List(1, 2) ::: List(3, 4, 5)
res0: List[Int] = List(1, 2, 3, 4, 5)

scala> List() ::: List(1, 2, 3)
res1: List[Int] = List(1, 2, 3)
```

```
scala> List(1, 2, 3) ::: List(4)
res2: List[Int] = List(1, 2, 3, 4)
```

与 cons 一样，列表的连接操作也是右结合的。像这样的表达式：

```
xs ::: ys ::: zs
```

等价于：

```
xs ::: (ys ::: zs)
```

302

分治原则

连接操作（:::）被实现为 List 类的方法。不过我们也可以通过对列表使用模式匹配来“手工”实现。能够自己完成这件事非常具有教育意义，因为这样能让你了解利用列表实现算法的通用方式。首先，我们需要决定连接方法（称其为 append）的参数格式。为了不让事情变得太麻烦，我们假设 append 方法是定义在 List 类之外的。因此它的参数应该是两个待连接的列表。这两个列表的元素可以是任意类型，但必须保持一致。这可以通过对 append 指定代表两个输入列表的元素类型的类型参数[3]来表达：

```
def append[T](xs: List[T], ys: List[T]): List[T]
```

在设计 append 方法的实现细节之前，有必要先回忆一下对递归类数据结构（如列表）进行编程时的“分治”设计原则。列表的许多算法首先是用模式匹配把输入列表拆分为更简单的样本，这是原则里所说的“分”；然后根据每个样本构建结果。如果结果是非空列表，那么一块块部件将通过同样的递归遍历算法构建出来。这就是原则里所说的“治”。

把这个原则应用到 append 方法的实现上，第一个问题就是需要匹配哪个列表。与其他的方法相比，在 append 例子中这个问题可能要多费一番心思，因为我们现在有了两个选择。然而，后续“治”的过程需要把两个输入列表的所有元素组合成一个列表。由于列表是从后向前构造的，因此明智的做法是让 ys 保持完整而 xs 则需要被拆分并放到 ys 前面，所以我们只能把注意力集中到 xs 的模式匹配上。最普通的模式匹配只是简单地把列表区分为空的和非空的，因此可以得到一个简要的 append 方法，如下：

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => // ??
    case x :: xs1 => // ??
  }
```

303

剩下的，就是把两个标志了“??”的地方填上代码。第一个地方是当输入列表 xs 为空时的可选项。这种情况下，连接操作应该生成第二个列表：

```
case List() => ys
```

留空的第二个地方是当输入列表 xs 由某个头元素 x 和尾列表 xs1 组成时的可选项，这种情况下结果也是非空列表。而要构建非空列表，你需要知道头元素和尾列表究竟是什么。你现在知道结果列表的

[3]注：类型参数将在第 19 章描述。

第一个元素是 x。对于余下的元素来说，可以通过合并第一个列表的余下部分 xs1 和第二个列表 ys 做出来。因此可以完成设计如下：

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
}
```

第二个可选项的计算过程充分演示了分治原则的“治”的部分：首先需设想所要的输出应该有的形状；然后计算其中独立的部分，使用此处适用的算法做递归遍历；最终根据这些部分构建成输出。

计算列表的长度：length 方法

length 方法能够计算列表的长度。

```
scala> List(1, 2, 3).length
res3: Int = 3
```

相对于数组来说，列表的 length 方法是较费时的操作。为了找到尾部，需要遍历整个列表，因此其花费的时间与列表元素数量成正比。这也是在判断列表是否为空时，应当采用 xs.isEmpty 方法，而不采用 xs.length == 0 的理由。虽然两种测试的结果一致，但第二种更慢，尤其是列表 xs 较长的时候。

304

访问列表的尾部：init 方法和 last 方法

你已经知道了 head 和 tail 基本操作，相应可以获得列表的第一个元素及除了第一个元素之外余下的列表。它们都有成对的操作：last 返回（非空）列表的最后一个元素，init 返回除了最后一个元素之外余下的列表：

```
scala> val abcde = List('a', 'b', 'c', 'd', 'e')
abcde: List[Char] = List(a, b, c, d, e)

scala> abcde.last
res4: Char = e

scala> abcde.init
res5: List[Char] = List(a, b, c, d)
```

与 head 和 tail 一样的地方是，对空列表调用这些方法的时候，会抛出异常：

```
scala> List().init
java.lang.UnsupportedOperationException: Nil.init
      at scala.List.init(List.scala:544)
      at ...
scala> List().last
java.util.NoSuchElementException: Nil.last
      at scala.List.last(List.scala:563)
      at ...
```

不一样的是，head 和 tail 运行的时间都是常量，但 init 和 last 需要遍历整个列表以计算结

果。因此所耗的时间与列表长度成正比。

组织好数据，以便让所有的访问都集中在列表的头部，而不是尾部。

反转列表：reverse 方法

如果出于某种原因，某种算法需要频繁地访问列表的尾部，那么可以首先把列表反转过来然后再处理。反转方法如下：

```
scala> abcde.reverse
res6: List[Char] = List(e, d, c, b, a)
```

请注意，与所有其他列表操作都一样，reverse 创建了新的列表而不是就地改变被操作列表。由于列表是不可变的，因此这种改变无论如何也不会发生。验证的方法是，可以检查 abcde 的原始值在 reverse 操作之后并未改变：

```
scala> abcde
res7: List[Char] = List(a, b, c, d, e)
```

reverse、init 和 last 操作满足一些为了计算合理性或简化编程而设的规则。

1. reverse 是其自身的反转：

```
xs.reverse.reverse  等价于  xs
```

2. reverse 把 init 转为 tail，把 last 转为 head，除了其中的元素是反的：

```
xs.reverse.init  等价于  xs.tail.reverse
xs.reverse.tail  等价于  xs.init.reverse
xs.reverse.head  等价于  xs.last
xs.reverse.last  等价于  xs.head
```

反转操作可以用连接（:::）实现，如以下方法，rev：

```
def rev[T](xs: List[T]): List[T] = xs match {
  case List() => xs
  case x :: xs1 => rev(xs1) ::: List(x)
}
```


不过，这个方法的效率不够理想。我们假设列表 xs 的长度为 n，以研究 rev 方法的复杂性。注意到这里有 n 次对 rev 的递归调用。除了最后一次之外，每个调用执行了一次列表连接。而列表连接 xs ::: ys 所用的时间与它的第一个参数 xs 的长度成正比。因此，rev 的总复杂度是：

$$n + (n-1) + \cdots + 1 = (1+n) * n/2$$

换句话说，rev 的复杂度为入参列表长度的平方。如果与可变的链接列表的标准反转方法相比较，这一结果令人失望。然而在所有可能的实现中，当前的 rev 实现并非最佳。在 16.7 节你将看到如何加速反转过程。

前缀与后缀：drop、take 和 splitAt

drop 和 take 操作泛化了 tail 和 init，它们可以返回列表任意长度的前缀或后缀。表达式“xs take n”返回 xs 列表的前 n 个元素。如果 n 大于 xs.length，则返回整个 xs。操作“xs drop n”返回 xs 列表除了前 n 个元素之外的所有元素。如果 n 大于 xs.length，则返回空列表。

splitAt 操作在指定位置拆分列表，并返回对偶（pair）列表[4]。它的定义符合以下等式：

```
xs splitAt n 等价于 (xs take n, xs drop n)
```

然而，splitAt 避免了对列表 xs 的二次遍历。以上三种方法的例子如下：

```
scala> abcde take 2
res8: List[Char] = List(a, b)

scala> abcde drop 2
res9: List[Char] = List(c, d, e)

scala> abcde splitAt 2
res10: (List[Char], List[Char]) = (List(a, b),List(c, d, e))
```

元素选择：apply 方法和 indices 方法

apply 方法实现了随机元素的选择；不过与数组中的同名方法相比，它使用得并不广泛。

```
scala> abcde apply 2 // Scala 中罕见
res11: Char = c
```

与其他类型一样，当对象出现在应该是方法调用的函数位置上时，就会隐式地插入 apply 方法，因此上面的代码可以缩写为：

```
scala> abcde(2) // Scala 中罕见
res12: Char = c
```

在列表中使用随机元素访问比在数组中要少得多，原因之一在于 xs(n) 花费的时间与索引值 n 成正比。实际上，apply 仅简单地定义为 drop 和 head 的组合：

```
xs apply n 等价于 (xs drop n).head
```

这个定义同时也说明了列表的索引范围是从 0 到列表长度减一。indices 方法可以返回指定列表的所有有效索引值组成的列表：

```
scala> abcde.indices
res13: List[Int] = List(0, 1, 2, 3, 4)
```

啮合列表：zip

zip 操作可以把两个列表组成一个对偶列表：

[4]注：正如 10.12 节提到过的，术语 pair 是 Tuple2 的非正规名。

```
scala> abcde.indices zip abcde
res14: List[(Int, Char)] = List((0,a), (1,b), (2,c), (3,d), (4,e))
```

如果两个列表的长度不一致，那么任何不能匹配的元素将被丢掉：

```
scala> val zipped = abcde zip List(1, 2, 3)
zipped: List[(Char, Int)] = List((a,1), (b,2), (c,3))
```

常用到的情况是把列表元素与索引值啮合在一起，这时使用 zipWithIndex 方法会更为有效，它能把列表的每个元素与元素在列表中的位置值组成一对。

308

```
scala> abcde.zipWithIndex
res15: List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3), (e,4))
```

显示列表：toString 方法和 mkString 方法

toString 操作返回列表的标准字符串表达形式：

```
scala> abcde.toString
res16: String = List(a, b, c, d, e)
```

如果你需要其他表达方式，可以使用 mkString 方法。xs mkString (pre, sep, post)操作有四个操作元：待显示的列表 xs，需要显示在所有元素之前的前缀字符串 pre，需要显示在每两个元素之间的分隔符字符串 sep，以及显示在最后面的后缀字符串 post。操作的结果就是字符串：

```
pre + xs(0) + sep + . . . + sep + xs(xs.length - 1) + post
```

mkString 方法有两个重载的变体以便让你可以忽略部分乃至全部参数。第一个变体仅带有分隔符字符串：

```
xs mkString sep  等价于  xs mkString ("", sep, "")
```

第二个变体让你可以忽略所有的参数：

```
xs.mkString  等价于  xs mkString ""
```

这里举例如下：

```
scala> abcde mkString ("[", ",", "]")
res17: String = [a,b,c,d,e]

scala> abcde mkString ""
res18: String = abcde

scala> abcde.mkString
res19: String = abcde

scala> abcde mkString ("List(", ", ", ")")
res20: String = List(a, b, c, d, e)
```

309

mkString 方法还有名为 addString 的变体，它可以把构建好的字符串添加到 StringBuilder

对象[5]中，而不是作为结果返回：

```
scala> val buf = new StringBuilder
buf: StringBuilder =

scala> abcde addString (buf, "(", ";", ")")
res21: StringBuilder = (a;b;c;d;e)
```

mkString 和 addString 方法都继承自 List 的超特质 Iterable，因此它们可以应用到各种可枚举的集合类上。

转换列表：elements、toArray、copyToArray

想要让数据存储格式在连续存放的数组和递归存放的列表之间进行转换，可以使用 List 类的 toArray 方法和 Array 类的 toList 方法：

```
scala> val arr = abcde.toArray
arr: Array[Char] = Array(a, b, c, d, e)

scala> arr.toString
res22: String = Array(a, b, c, d, e)

scala> arr.toList
res23: List[Char] = List(a, b, c, d, e)
```

另外还有一个方法叫 copyToArray，可以把列表元素复制到目标数组的一段连续空间。操作如下：

```
xs copyToArray (arr, start)
```

这将把列表 xs 的所有元素复制到数组 arr 中，填入位置开始为 start。必须确保目标数组 arr 有足够的空间可以全部放下列表元素。举例如下：

```
scala> val arr2 = new Array[Int](10)
arr2: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> List(1, 2, 3) copyToArray (arr2, 3)

scala> arr2.toString
res25: String = Array(0, 0, 0, 1, 2, 3, 0, 0, 0, 0)
```

最后，如果你需要用枚举器访问列表元素，可以使用 elements 方法：

```
scala> val it = abcde.elements
it: Iterator[Char] = non-empty iterator

scala> it.next
res26: Char = a

scala> it.next
res27: Char = b
```

举例：归并排序

前面看到的插入排序编写起来较为简洁，但效率不高。它的平均复杂度与输入列表的长度的平方成

[5]注：这里是 scala.StringBuilder 类，而不是 java.lang.StringBuilder 类。

正比。归并排序（merge sort）是更为有效的排序算法。

> **快速通道**
>
> 除了深入讨论了算法的复杂度问题之外，这个例子还对分治原则及柯里化（currying）作了演示。不过如果你希望在本书的第一轮通读进程中加快步伐的话，可以直接跳到 16.7 节。

归并排序的工作原理如下：首先如果列表长度为零或仅有一个元素，它就已经是排好序的了，因此可以不加改变地返回。长列表可以拆分为两个子列表，每个包含大概一半的原列表元素。每个子列表采用对排序函数的递归调用完成排序，然后再用归并操作把产生的两个排好序的列表合并在一起。

对于归并排序的通用实现来说，你可能希望对于待排序列表元素的类型不做限制，并同样希望对于用来做元素比较的函数也不做限制。你可以通过把这两项作为传入参数以获得具有最大通用性的函数。这样我们就得到了清单 16.1 中实现的代码：

```
def msort[T](less: (T, T) => Boolean)
    (xs: List[T]): List[T] = {

  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (less(x, y)) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }

  val n = xs.length / 2
  if (n == 0) xs
  else {
    val (ys, zs) = xs splitAt n
    merge(msort(less)(ys), msort(less)(zs))
  }
}
```

清单 16.1　List 类的归并排序

msort 的复杂度为(n log(n))级，这里的 n 是输入列表的长度。理由如下，列表拆分与两个有序列表的归并，耗费的时间都与参数列表的长度成正比。每次对 msort 的递归调用都会把输入的元素数量折半，因此大概要连续递归调用 log(n)次才能达到列表长度为 1 这种基本的情况。然而，对于较长的列表来说，每次调用都会衍生出下两次递归调用。把这些统加起来我们就得到了对于每个 log(n)调用级别，原始列表的每个元素都参加了一次拆分操作和一次合并操作。因此，每个调用级别的总开销与 n 成正比。因为有 log(n)个调用级别，所以全体操作的总开销与(n log(n))成正比。而这个开销与列表中元素的初始分布并无关系，因此最坏情况的开销与平均情况的开销一致。这种属性使得归并排序成为很具吸引力的列表排序算法。

msort 函数的用法如下：

```
scala> msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
res28: List[Int] = List(1, 3, 5, 7)
```

msort 函数是我们曾经在 9.3 节中讨论过的柯里化概念的经典例子。柯里化可以让它更易于指定比

较函数。举例如下：

312

```
scala> val intSort = msort((x: Int, y: Int) => x < y) _
intSort: (List[Int]) => List[Int] = <function>
```

intSort 变量所指的函数的入参是整数列表，能够按照顺序排列。正如在 8.6 节曾经描述过的，下划线代表的是没有写的参数列表。在这个例子中，没有写的参数是需要排序的列表。另举一例，下面定义的代码可以把整数列表以倒序排列：

```
scala> val reverseIntSort = msort((x: Int, y: Int) => x > y) _
reverseIntSort: (List[Int]) => List[Int] = <function>
```

因为已经通过柯里化提供了比较函数，所以当你在调用 intSort 或 reverseIntSort 函数的时候，仅须提供待排序的列表即可。举例如下：

```
scala> val mixedInts = List(4, 1, 9, 0, 5, 8, 3, 6, 2, 7)
mixedInts: List[Int] = List(4, 1, 9, 0, 5, 8, 3, 6, 2, 7)

scala> intSort(mixedInts)
res0: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> reverseIntSort(mixedInts)
res1: List[Int] = List(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

16.7 List 类的高阶方法

对于列表的许多操作都具有相同的结构，模式重复不断地出现。例如：用某种方式转变列表的所有元素，检查列表的所有元素是否都具有某种性质，取出列表中满足特定条件的元素，或使用某种操作符合并列表的元素。Java 中，这样的模式通常需要用 for 或 while 循环的固定组合表达。Scala 中，可以通过使用以 List 类的方法实现的高阶操作符[6]来更为简洁和直接地表达这些模式。本节将讨论这些高阶操作符。

313

列表间映射：map、flatMap 和 foreach

xs map f 操作以类型为 List[T] 的列表 xs 和类型为 T => U 的函数 f 为操作元。返回把函数 f 应用在 xs 的每个列表元素之后由此组成的新列表。如：

```
scala> List(1, 2, 3) map (_ + 1)
res29: List[Int] = List(2, 3, 4)

scala> val words = List("the", "quick", "brown", "fox")
words: List[java.lang.String] = List(the, quick, brown, fox)

scala> words map (_.length)
res30: List[Int] = List(3, 5, 5, 3)

scala> words map (_.toList.reverse.mkString)
res31: List[String] = List(eht, kciuq, nworb, xof)
```

flatMap 操作符与 map 类似，不过它的右操作元是能够返回元素列表的函数。它对列表的每个元

[6]注：我们这里所说的高阶操作符（higher-order operator），就是以操作符形式出现的高阶函数。9.1 节我们曾经提到过，高阶函数就是以其他函数做参数的函数。

素调用该方法，然后连接所有方法的结果并返回。map 和 flatMap 的差异举例说明如下：

```
scala> words map (_.toList)
res32: List[List[Char]] = List(List(t, h, e), List(q, u, i, c, k),
    List(b, r, o, w, n), List(f, o, x))

scala> words flatMap (_.toList)
res33: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x)
```

可以发现 map 返回的是包含列表的列表，而 flatMap 返回的是把所有元素列表连接之后的单个列表。

map 与 flatMap 之间的差异和互相协作同样也可以用下面的例子说明。它可以构建出所有 1≤j<i<5 的（i,j）对偶：

```
scala> List.range(1, 5) flatMap (
         i => List.range(1, i) map (j => (i, j))
       )
res34: List[(Int, Int)] = List((2,1), (3,1), (3,2), (4,1), (4,2), (4,3))
```

List.range 是可以创建某范围内所有整数列表的工具方法。在这里使用了两次：一次产生从 1（包含）到 5（排除）的整数列表，第二次产生了从 1 到 i 的整数列表，i 的每个值都取自前一个列表。这个表达式里的 map 产生了元组（i, j）的列表，这里 j<i。外部的 flatMap 对每个 1 到 5 之间的 i 产生列表，并连接所有列表得到结果。

请注意，使用 for 表达式也能得到同样的列表：

```
for (i <- List.range(1, 5); j <- List.range(1, i)) yield (i, j)
```

在第 23 章你将学到更多有关 for 表达式与列表的交互操作的知识。

foreach 是第三种与映射类似的操作。然而不像 map 和 flatMap，foreach 的右操作元是过程（返回类型为 Unit 的函数）。它只是对每个列表元素都调用一遍过程。操作的结果仍然是 Unit，不会产生结果列表。例如，以下的代码可以把列表的所有数字都累加起来：

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res36: Int = 15
```

列表过滤：filter、partition、find、takeWhile、dropWhile 和 span

“xs filter p”操作把类型为 List[T]的列表 xs 和类型为 T => Boolean 的论断函数作为操作元。产生 xs 中符合 p(x)为 true 的所有元素 x 组成的列表。如：

```
scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
res37: List[Int] = List(2, 4)

scala> words filter (_.length == 3)
res38: List[java.lang.String] = List(the, fox)
```

partition 方法与 filter 类似，不过返回的是列表对。其中一个包含所有论断为真的元素，另一个包含所有论断为假的元素。它的定义与以下等式相同：

```
xs partition p    等价于    (xs filter p, xs filter (!p(_)))
```

举例如下：

```
scala> List(1, 2, 3, 4, 5) partition (_ % 2 == 0)
res39: (List[Int], List[Int]) = (List(2, 4),List(1, 3, 5))
```

find 方法同样与 filter 类似，不过返回的是第一个满足给定论断的元素，而并非全部。xs find p 操作以列表 xs 和论断 p 为操作元。返回可选值。如果 xs 中存在元素 x 使得 p(x) 为真，Some(x) 将返回。否则，若 p 对所有元素都不成立，None 将返回。举例如下：

```
scala>  List(1, 2, 3, 4, 5) find (_ % 2 == 0)
res40: Option[Int] = Some(2)

scala>  List(1, 2, 3, 4, 5) find (_ <= 0)
res41: Option[Int] = None
```

takeWhile 和 dropWhile 操作符同样带有论断做右操作元。xs takeWhile p 操作返回列表 xs 中最长的能够满足 p 的前缀。类似地，xs dropWhile p 操作移除最长的能够满足 p 的前缀。举例如下：

```
scala> List(1, 2, 3, -4, 5) takeWhile (_ > 0)
res42: List[Int] = List(1, 2, 3)

scala> words dropWhile (_ startsWith "t")
res43: List[java.lang.String] = List(quick, brown, fox)
```

span 方法把 takeWhile 和 dropWhile 组合成一个操作，就好像 splitAt 组合了 take 和 drop 一样。它返回一对列表，定义与下列等式一致：

```
xs  span p    等价于    (xs takeWhile p, xs dropWhile p)
```

与 splitAt 一样，span 可以避免对列表 xs 的二次访问：

```
scala> List(1, 2, 3, -4, 5) span (_ > 0)
res44: (List[Int], List[Int]) = (List(1, 2, 3),List(-4, 5))
```

列表的论断：forall 和 exists

操作 xs forall p 以列表 xs 和论断 p 为参数。如果列表的所有元素满足 p 则返回 true。与之相对，在 xs exists p 操作里，xs 中只要有一个值满足论断 p 就返回 true。例如，要找出以列表的列表形式表达的矩阵是否有全零行元素：

```
scala> def hasZeroRow(m: List[List[Int]]) =
         m exists (row => row forall (_ == 0))
hasZeroRow: (List[List[Int]])Boolean

scala> hasZeroRow(diag3)
res45: Boolean = false
```

折叠列表：/:和:\

其他常用的操作会对列表元素始终执行某种操作。例如：

```
sum(List(a, b, c))    等价于    0 + a + b + c
```

这可以认为是折叠操作的一个具体实例：

```
scala> def sum(xs: List[Int]): Int = (0 /: xs) (_ + _)
sum: (List[Int])Int
```

类似地：

```
product(List(a, b, c))    等价于    1 * a * b * c
```

也是折叠操作的另一个具体实例：

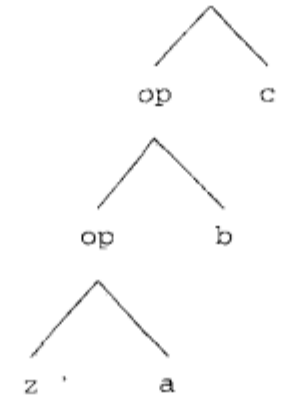
```
scala> def product(xs: List[Int]): Int = (1 /: xs) (_ * _)
product: (List[Int])Int
```

317

左折叠（fold left）操作“(z /: xs) (op)”与三个对象有关：开始值 z，列表 xs，以及二元操作 op。折叠的结果是 op 应用到前缀值 z 及每个相邻元素上，如下：

```
(z /: List(a, b, c)) (op)    等价于    op(op(op(z, a), b), c)
```

或图形化为：



以下是另外一个演示/:用法的例子。用空格连接所有字符串列表中的单词，可以写成：

```
scala>  ("" /: words) (_ +" "+ _)
res46: java.lang.String = the quick brown fox
```

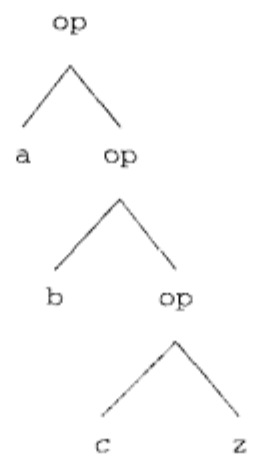
结果在最开始的地方多了一个空格。要去掉它，可以稍微改变为：

```
scala> (words.head /: words.tail) (_ +" "+ _)
res47: java.lang.String = the quick brown fox
```

/:操作符产生向左侧倾斜的操作树（语法中反斜杠“/”的前进方向也有意反映了这一点）。与之相类似，操作符:\产生向右倾斜的操作树。如：

```
(List(a, b, c) :\ z) (op)    等价于    op(a, op(b, op(c, z)))
```

或者，图形化为：



:\操作符被称为右折叠（fold right）。与左折叠一样带有三个操作元，不过前两个的出现次序相反：第一个操作元是待折叠的列表，第二个是开始值。

对于组合操作来说，左折叠与右折叠是等价的，不过在效率上稍有差别。这里可以参考一个与 List.flatten 方法[7]有关的例子，该方法可以把列表的列表中的所有元素连接在一起。它既可以用左折叠实现，也可以用右折叠实现：

```
def flattenLeft[T](xss: List[List[T]]) =
    (List[T]() /: xss) (_ ::: _)

def flattenRight[T](xss: List[List[T]]) =
    (xss :\ List[T]()) (_ ::: _)
```

由于列表的连接 xs ::: ys 耗时与第一个参数 xs 的长度成正比，因此以右折叠形式完成的实现 flattenRight 比左折叠实现 flattenLeft 更有效率。原因在于 flattenLeft(xss)需要复制第一个元素列表 xss.head 一共 n-1 次，这里 n 是指列表 xss 的长度。

请注意 flatten 的两个版本都需要对作为折叠开始值的空列表做类型标注。这是由于 Scala 的类型推断的局限性，它不能自动地推断出列表的正确类型。如果不做标注的话，会得到如下错误：

```
scala> def flattenRight[T](xss: List[List[T]]) =
           (xss :\ List()) (_ ::: _)
<console>:15: error: type mismatch;
 found : List[T]
 required: List[Nothing]
           (xss :\ List()) (_ ::: _)
                           ^
```

想要知道为什么类型推断会失败，就需要知道折叠方法的类型及实现方式。更多内容参见第 22 章。

最后，尽管/:和:\操作符已经从斜杠的方向上描绘了它们的操作树的倾斜方向，并且冒号字符的结合性也保证了开始值在操作树和表达式中处于同样的位置，但还是有很多人感觉产生的结果与直觉相去甚远。因此如果你喜欢，也可以使用名为 foldLeft 和 foldRight 的方法，它们同样定义在 List 类中。

[7]注：List.flatten 将在本章的下一节说明。

例子：使用折叠操作完成列表反转

本章早些时候曾经完成过 reverse 方法的实现，称为 rev，其运行时间为待反转的列表长度的平方。下面是另一种耗时为线性关系（与列表长度成正比）的 reverse 实现。其思想是通过左折叠操作的调用，方案如下：

```
def reverseLeft[T](xs: List[T]) = (startvalue /: xs)(operation)
```

现在剩下的只有在 startvalue 和 operation 处填入正确的部分。实际上，你可以用一些简单的例子来推导这些部分。要推导 startvalue 的正确值，可以从可能出现的最小列表 List() 开始，并作如下推导计算：

```
List()
  等价于 （根据 reverseLeft 的属性）
reverseLeft(List())
  等价于 （根据 reverseLeft 的模板）
(startvalue /: List())(operation)
  等价于 （根据/:的定义）
startvalue
```

因此，startvalue 必然是 List()。要推导第二个操作元，可以另选一个次小的列表作为样例。你已经知道 startvalue 是 List()，因此有推导计算如下：

```
List(x)
  等价于 （根据 reverseLeft 的属性）
reverseLeft(List(x))
  等价于 （根据 reverseLeft 的模板，并 startvalue = List()）
(List() /: List(x)) (operation)
  等价于 （根据/:的定义）
operation(List(), x)
```

因此，operation(List(), x)等价于 List(x)，还可以写成 x :: List()。这暗示了可以用::操作符及交换操作元位置来替换 operation（这种操作有时也被称为“snoc”，以暗示被称为“cons”的::）。最终我们获得了 reverseLeft 的如下实现：

```
def reverseLeft[T](xs: List[T]) =
  (List[T]() /: xs) {(ys, y) => y :: ys}
```

（同样，为了使得类型推断正常工作，List[T]()中的类型标注是不可或缺的）。如果分析了 reverseLeft 的复杂度，你会发现它 n 倍于常数时间操作（“snoc”），这里 n 是参数列表的长度。因此，reverseLeft 的复杂度是线性的，正如我们所希望的。

列表排序：sort

对列表 xs 的操作，xs sort before，可以对列表的元素执行排序，其中“before”是比较元素的方法。表达式 x before y 在 x 应按照顺序处于 y 之前的时候要能够返回 true。举例如下：

```
scala> List(1, -3, 4, 2, 6) sort (_ < _)
res48: List[Int] = List(-3, 1, 2, 4, 6)
```

```
scala> words sort (_.length > _.length)
res49: List[java.lang.String] = List(quick, brown, fox, the)
```

请注意，sort 执行的是与曾经在上一节提到过的与 msort 算法相似的归并排序，不过它是 List 类的方法而 msort 定义在列表之外。

16.8 List 对象的方法

目前为止，你在本章看到的所有操作都实现为 List 类的方法，因此你是在独立的类对象上调用它们。还有些方法是定义在全局可访问对象 scala.List 上的，它是 List 类的伴生对象。其中的一些操作是创建列表的工厂方法。另外一些是对某些特定类型列表的操作。两种类型的方法都将在本节介绍。

通过元素创建列表：List.apply

你已经在许多场合看到过了以 List(1, 2, 3)形式出现的列表字面量。它的语法没有什么特别的。类似于 List(1, 2, 3)这样的字面量只是 List 对象对元素 1、2、3 的简单应用。也就是说，它等价于 List.apply(1, 2, 3)：

```
scala> List.apply(1, 2, 3)
res50: List[Int] = List(1, 2, 3)
```

创建数值范围：List.range

本章早些时候在讨论 map 和 flatmap 时曾简略地提过 range 方法，它可以创建数值范围的列表。最简单的形式是 List.range(from, until)，可以创建从 from 开始到 until 减一的所有数值的列表。因此尾部值 until 不在范围之内。

还有另外一个版本的 range 可以带 step 值作为第三参数。这个操作可以产生从 from 开始的，间隔为 step 的列表元素。step 可以为正，也可以为负：

```
scala> List.range(1, 5)
res51: List[Int] = List(1, 2, 3, 4)

scala> List.range(1, 9, 2)
res52: List[Int] = List(1, 3, 5, 7)

scala> List.range(9, 1, -3)
res53: List[Int] = List(9, 6, 3)
```

创建统一的列表：List.make

make 方法创建由相同元素的零份或多份拷贝组成的列表。它带两个参数：待创建列表的长度，需重复的元素：

```
scala> List.make(5, 'a')
res54: List[Char] = List(a, a, a, a, a)
```

```
scala> List.make(3, "hello")
res55: List[java.lang.String] = List(hello, hello, hello)
```

322

解除啮合列表：List.unzip

unzip 操作是 zip 的相反。zip 把两个列表组成对偶（pair）列表，unzip 把对偶列表拆分还原为两个列表，其中一个列表由每对对偶的第一个元素组成，另一个由第二个元素组成：

```
scala> val zipped = "abcde".toList zip List(1, 2, 3)
zipped: List[(Char, Int)] = List((a,1), (b,2), (c,3))

scala> List.unzip(zipped)
res56: (List[Char], List[Int]) = (List(a, b, c),
    List(1, 2, 3))
```

> **注意**
>
> 你或许奇怪为什么 unzip 要成为全局 List 对象的方法，而不是 List 类的内建方法。原因在于 unzip 并不处理除了结对列表之外的其他类型列表，而 Scala 的类型系统需要类的所有方法对所有类实例都是可用的。因此，unzip 不能放在 List 类中。将来有可能会扩展 Scala 的类型系统以便让它能够接受只能应用于某些类实例的方法，但到目前为止还不能做到。

连接列表：List.flatten、List.concat

flatten 方法以列表的列表做参数，并把所有的元素列表连接在一起。举例如下：

```
scala> val xss =
         List(List('a', 'b'), List('c'), List('d', 'e'))
xss: List[List[Char]] = List(List(a, b), List(c), List(d, e))

scala> List.flatten(xss)
res57: List[Char] = List(a, b, c, d, e)
```

> **注意**
>
> flatten 方法被打包入全局的 List 对象中，理由与 unzip 一致：它不能处理任意的列表，仅能用于包含了列表的列表，因此不能作为通用的 List 类方法。

323

concat 方法与 flatten 类似，它能够连接多个元素列表。将多个列表以重复参数的形式直接传递给方法。数量不限：

```
scala> List.concat(List('a', 'b'), List('c'))
res58: List[Char] = List(a, b, c)

scala> List.concat(List(), List('b'), List('c'))
res59: List[Char] = List(b, c)

scala> List.concat()
res60: List[Nothing] = List()
```

映射及测试配对列表：List.map2、List.forall2、List.exists2

map2 方法与 map 相似，不过它同时带两个列表及能够把两个元素值映射为结果的函数做参数。函数会应用到两个列表相关的元素上，然后把这些结果值变为列表：

```
scala> List.map2(List(10, 20), List(3, 4, 5)) (_ * _)
res61: List[Int] = List(30, 80)
```

exist2 和 forall2 方法同样类似于 exists 及 forall，不过它们也同样带两个列表及输入两个参数的布尔型论断。论断也是应用于相应的参数上：

```
scala> List.forall2(List("abc", "de"),
           List(3, 2)) (_.length == _)
res62: Boolean = true

scala> List.exists2(List("abc", "de"),
            List(3, 2)) (_.length != _)
res63: Boolean = false
```

> **快速通道**
>
> 本章的下一节（也是最后一节）中，我们将揭秘 Scala 的类型推断算法。如果你目前还没有对这些细节产生兴趣的话，可以安全地跳过这部分。直接转到第 204 页的小结。

324

16.9 了解 Scala 的类型推断算法

sort 与 msort 两种方法在之前的用法上存在着一个差别，这个差别与比较函数采用的句法形式有关。请试着比较：

```
scala> msort((x: Char, y: Char) => x > y)(abcde)
res64: List[Char] = List(e, d, c, b, a)
```

与：

```
scala> abcde sort (_ > _)
res65: List[Char] = List(e, d, c, b, a)
```

两种表达式是等价的，但第一种的比较函数使用了长格式，有命名的参数及明确的类型定义，而第二种使用了简洁格式（_ > _），命名参数被替换成了下划线。当然，你也可以让 sort 使用第一种较长的比较形式。不过 msort 却不能使用短格式：

```
scala> msort(_ > _)(abcde)
<console>:12: error: missing parameter type for expanded
function ((x$1, x$2) => x$1.$greater(x$2))
       msort(_ > _)(abcde)
             ^
```

要想知道为什么，就需要知道 Scala 的类型推断算法的细节。Scala 里的类型推断是基于流的。在 m(args) 的方法调用中，（类型）推断器首先检查方法 m 是否有已知类型。如果有，那么这个类型将被用来做参数预期类型的推断。例如，在 abcde.sort(_ > _) 中，abcde 的类型是 List[Char]，因此就明确了 sort 的参数类型为 (Char, Char) => Boolean 并且产生的结果类型为 List[Char]。

由于函数入参的参数类型已知，因此就不必再明确给出了。根据对 sort 已知的内容，推断器能够推断出(_ > _)可以被扩展为((x: Char, y: Char) => x > y)，这里 x 和 y 是任意的未被使用过的名称。

现在考虑一下第二个例子，msort(_ > _)(abcde)。msort 的类型是柯里化的、多态的方法类型，该方法类型的参数类型是(T, T) => Boolean，并返回从 List[T]到 List[T]的函数，其中 T 是未知的类型。因此 msort 方法需要在对它的参数执行调用之前，用类型参数做实例化。因为并不知道应用中 msort 精确的实例类型，所以无法用来推断它的第一个参数的类型。这种情况下类型推断器改变了策略，它首先做方法参数的类型检查以确定方法合适的实例类型。然而，当打算对缩写形式的函数字面量(_ > _)做类型检查的时候，由于用下划线指代的隐式函数参数没有提供类型信息，所以会造成检查失败。

325

唯一的解决之道只能是明确地把类型参数传递给 msort，如：

```
scala> msort[Char](_ > _)(abcde)
res66: List[Char] = List(e, d, c, b, a)
```

msort 的正确实例类型已知，因此现在可以用来做参数类型的推断了。

另一种可能的解决方案是重写 msort 方法以交换它的参数顺序：

```
def msortSwapped[T](xs: List[T])(less:
    (T, T) => Boolean): List[T] = {
  // 与 msort 同样的实现，不过交换了参数的次序
}
```

现在就可以执行类型推断了：

```
scala> msortSwapped(abcde)(_ > _)
res67: List[Char] = List(e, d, c, b, a)
```

这里的改变在于推断器能够使用第一个参数 abcde 的已知类型决定 msortSwapped 方法的类型参数。一旦 msortSwapped 的类型被明确，它就可以转而用来推断第二个参数(_ > _)的类型。

通常，一旦有需要推断多态方法类型参数的任务时，类型推断器就只会参考第一个参数列表中所有的值参数类型，但不会参考之后其他的参数。因为 msortSwapped 是柯里化的方法，带两个参数列表，所以第二个参数（也就是说，那个函数值）将不会用来做决定方法类型参数的参考。

326

因此这种类型推断的方案也隐含了如下的库方法设计原则：如果需要把参数设计为若干非函数值及一个函数值的某种多态方法，需要把函数参数独自放在柯里化参数列表的最后面。这样一来，方法的正确实例类型就可以通过非函数参数推断出来，并且这个类型可以转而用来完成函数参数的类型检查。从而使得方法的使用者可以避免提供更多的类型信息并能编写出更为简洁的函数字面量。

现在是时候处理更为复杂的折叠（fold）操作了。为什么在表达式中需要提供明确的类型参数，就像之前曾看到过的 flattenRight 方法那样呢?

```
(xss :\ List[T]()) (_ ::: _)
```

因为右折叠操作的类型取决于两个类型变量，对于表达式：

```
(xs :\ z) (op)
```

来说，xs 的类型必然是某种任意类型 A 的列表，比如说，写成 xs: List[A]。而开始值 z 可能是另一种类型 B。而操作 op 必须以类型 A 和 B 的值为参数并返回类型 B 的结果，也就是说，op: (A, B) => B。因为 z 的类型不见得与列表 xs 有关系，所以类型推断就不会有 z 的相关信息。现在如果以错误的方式定义 flattenRight 方法的话：

```
(xss :\ List()) (_ ::: _) // 编译失败
```

折叠操作的开始值 z 是空列表 List()，没有任何附加类型信息，所以它的类型就被推断为 List[Nothing]。这样，推断器会认为折叠操作的类型 B 是 List[Nothing]。结果，折叠的(_ ::: _)操作被预期为如下类型：

```
(List[T], List[Nothing]) => List[Nothing]
```

这的确是折叠操作的可能类型但却没有丝毫意义。这意味着操作的输入和输出始终都是空列表。换句话说，就是类型推断只处理到 List()就结束还为时过早，它应该至少等到发现了 op 操作的类型才结束。因此（另一个非常有用的）规则——在柯里化的方法中，方法类型仅取决于第一段参数——即源于这个问题。另一方面，即使规则不被执行，推断器仍然无法获得 op 的类型，因为没有被明确给出它的参数类型。如此一来，这里就陷入了第 22 条军规般左右为难的情况，也只能以程序员明确指定类型标注的方式来解决。

327

Scala 采用局部的、基于流的类型推断方案，其局限性再次显露无遗。函数式语言如 ML 或 Haskell 则采用的是更为全局化的 Hindley-Milner 类型推断方式。然而，Scala 的局部类型推断对于面向对象的分支类型的处理比 Hindley-Milner 方式要完成得更为优雅。幸运的是，上面看到的局限性仅存在于某些极端的例子中，并且通常可以很容易地通过明确的类型标注来解决。

另外，在对于多态方法的类型错误信息迷惑不解时，添加类型标注同样也是一种有用的除错技术。如果你不能确定某种类型错误的原因时，只要加上一些你认为是正确的类型参数或类型标注，这样你很快就能发现真正的问题是什么了。

16.10 小结

现在你已经学习了许多使用列表的方法。学习了它的基本操作，如 head 和 tail；一阶操作，如 reverse；高阶操作，如 map；以及 List 对象中的工具方法。顺道还学习了 Scala 的类型推断的工作原理。

列表是 Scala 中的重头戏，这些知识会让你获益匪浅。因此，本章深入地说明了列表的使用方法。然而列表只是 Scala 所支持的集合类型中的一种。下一章将更为广泛，而并非深入地向你说明如何使用 Scala 中各种各样的集合类型。

328

第 17 章

集合类型

Scala 拥有丰富的集合类型库。前面的章节中你已经看过了最常用到的集合类型——数组（Array）、列表（List）、集（Set）、还有映射（Map）——但可讲的还有很多。本章，我们一开始先来大致浏览一下这些类型在集合继承层次中是如何相互联系的。之后，我们还将大致描述这些以及其他你可能会偶尔用到的集合类型，包括对它们的速度、空间，以及输入数据的限制需满足的条件之间的权衡的讨论。

17.1 集合库概览

Scala 的集合库包含了许多类和特质。因此，想要通过浏览 Scaladoc 文档，以求对这个集合库有完整的认识，也变成了很麻烦的事情。图 17.1 中，我们仅展示了认识整体结构所需的特质。

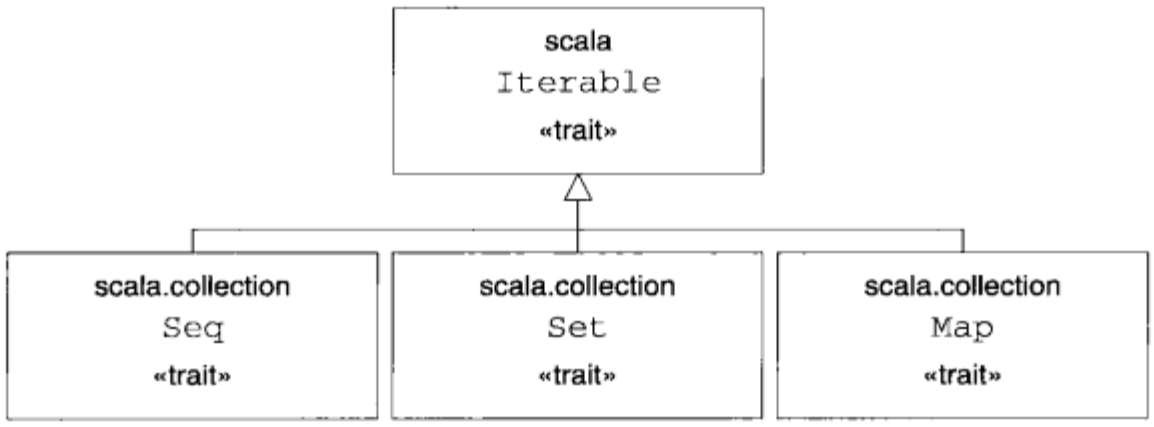


图 17.1 Scala 集合类型的类层级

Iterable 是主要特质，它同时还是可变和不可变序列（Seq）、集（Set），以及映射（Map）的超特质。序列是有序的集合，例如数组和列表。集可以通过==方法确定对每个对象最多只包含一个。映射则包含了键值映射关系的集合。

命名为 Iterable 是为了说明集合对象可以通过名为 elements 的方法产生 Iterator（枚举器）：

```
def elements: Iterator[A]
```

例子中的 A 是 Iterator 的类型参数，它指代集合中包含的元素的类型。elements 返回的 Iterator 被参数化为同样的类型。例如，Iterable[Int]的 elements 方法将创建 Iterator[Int]。

Iterable 包含几十个有用的具体方法，所有这些方法都是使用了 elements 返回的 Iterator 实现的，而 elements 是 Iterable 唯一的抽象方法。Iterable 定义的方法中，许多是高阶方法，

多数都已经在前面的章节中出现过。其中包含 map、flatMap、filter、exists 及 find。这些高阶方法提供了根据特定需求枚举集合的简洁方式，诸如修改所有的元素并创建新的集合（map 方法）或找到符合给定条件的第一个元素（find 方法）。

Iterator 有许多与 Iterable 相同的方法，包括那些高阶方法，但它们不属于同一层级。如图 17.2 所见，特质 Iterator 扩展了 AnyRef。Iterable 与 Iterator 之间的差异在于特质 Iterable 指代的是可以被枚举的类型（如集合类型），而特质 Iterator 是用来执行枚举操作的机制。尽管 Iterable 可以被枚举若干次，但 Iterator 仅能使用一次。一旦你使用 Iterator 枚举遍历了集合对象，你就不能再使用它了。如果你需要再次枚举该集合对象，你需要对它调用 elements 方法获得新的 Iterator。

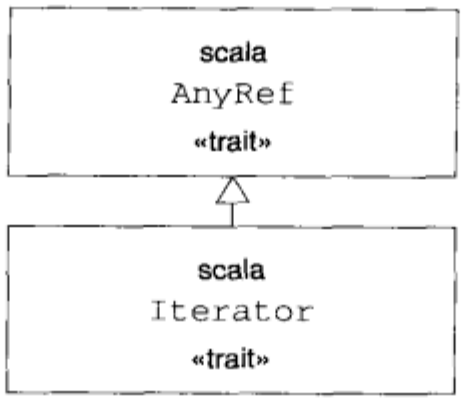


图 17.2 Iterator 的类层级

Iterator 提供的具体方法都使用了 next 和 hasNext 抽象方法实现：

```
def hasNext: Boolean
def next: A
```

hasNext 方法指代是否在枚举中仍剩余未遍历的元素。next 方法返回下一个元素。

尽管你有可能碰到的多数的 Iterable 实现将代表的都是有限空间的集合，Iterable 也同样可以用来代表无限的集合。例如，可以把π值的推演计算变成一种无限集合，并让从这种无限集合返回的 Iterator 在每次调用 next 方法的时候计算并返回π的下一个数字。

17.2 序列

序列是继承自特质 Seq 的类，它可以让你处理一组线性分布的数据。因为元素是有序的，所以你可以请求第一个元素，第二个元素，第 103 个元素，乃至更多。本节，我们将快速浏览最重要的序列。

列表

或许应知应会的最重要的序列类型就是 List 类了，上一章中详细描述了不可变的链接列表。列表可以支持在列表的头部快速添加和删除，但不能提供对任意索引值的快速访问，因为这种实现需要线性枚举整个列表。

这种特征的组合可能令人感觉很古怪，但对于许多算法来说却恰是最佳的发力点。头元素的快速添加和删除意味着可以很好地用于模式匹配，参见第 15 章。列表的不可变性也能帮助你开发正确和高效的算法，因为你永远无须创建列表的拷贝。下面的短例演示了列表的初始化及分别对其首、尾的访问：

```
scala> val colors = List("red", "blue", "green")
colors: List[java.lang.String] = List(red, blue, green)

scala> colors.head
res0: java.lang.String = red

scala> colors.tail
res1: List[java.lang.String] = List(blue, green)
```

列表的介绍可以参见第 3 章的第八步，使用细节参见第 16 章。列表还将在第 22 章讨论，届时将提供更多 Scala 中列表的实现细节。

数组

数组能够让你保留一组元素序列并可以用基于零的索引高效访问（无论是获取还是添加）处于任意位置的元素。下列代码说明了如何创建长度已知但内容未知的数组：

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

以下的代码说明了如何根据已知元素初始化数组：

```
scala> val fiveToOne = Array(5, 4, 3, 2, 1)
fiveToOne: Array[Int] = Array(5, 4, 3, 2, 1)
```

正如之前提到的，Scala 中数组的访问方式是通过把索引值放在圆括号里，而不是像 Java 里那样放在方括号里。下面的例子访问和更新了数组元素：

```
scala> fiveInts(0) = fiveToOne(4)
scala> fiveInts
res1: Array[Int] = Array(1, 0, 0, 0, 0)
```

Scala 的数组与 Java 数组的表达方式一致。因此，你可以无缝地使用现存的返回数组的 Java 方法。[1]

前几章中你已经在实战里多次看到了数组的使用。基本内容参见第 3 章的第七步。使用 for 表达式对数组元素做枚举的若干例子参见 7.3 节。在第 10 章的二维布局库中数组也有令人印象深刻的表现。

列表缓存

List 类能够提供对列表头部，而非尾部的快速访问。因此，如果需要通过向结尾添加对象的方式建造列表，你应该考虑先以对表头前缀元素的方式反向构造列表，完成之后再调用 reverse 使得元素反转为你需要的顺序。

另一种方式是使用 ListBuffer，这可以避免 reverse 操作。ListBuffer 是可变对象（包含在 scala.collection.mutable 包中），它可以更高效地通过添加元素的方式构建列表。ListBuffer 能够支持常量时间的添加和前缀操作。元素的添加使用+=操作符，前缀使用+:操作符。完成之后，可以通过对 ListBuffer 调用 toList 方法获得 List。举例如下：

```
scala> import scala.collection.mutable.ListBuffer
```

[1]注：Scala 的数组变化型（译注：类型参数化的数组）与 Java 数组的差别——即 Array[String]是否为 Array[AnyRef]的子类型——将在 19.3 节讨论。

```
import scala.collection.mutable.ListBuffer
scala> val buf = new ListBuffer[Int]
buf: scala.collection.mutable.ListBuffer[Int] = ListBuffer()

scala> buf += 1

scala> buf += 2

scala> buf
res11: scala.collection.mutable.ListBuffer[Int]
  = ListBuffer(1, 2)

scala> 3 +: buf
res12: scala.collection.mutable.Buffer[Int]
  = ListBuffer(3, 1, 2)

scala> buf.toList
res13: List[Int] = List(3, 1, 2)
```

使用 ListBuffer 替代 List 的另一个理由是为了避免栈溢出的风险。即使你能够使用前缀的方式以正确的次序构建列表，但所需的递归算法不是尾递归，那么你也可以使用 for 表达式或 while 循环及 ListBuffer 做替代。你将在 22.2 节看到 ListBuffer 的这种用法。

333

数组缓存

ArrayBuffer 与数组类似，只是额外还允许你在序列的开始或结束的地方添加和删除元素。所有的 Array 操作都被保留，只是由于实现中的包装层导致执行得稍微有些慢。新的添加和删除操作平均为常量时间，但偶尔会因为实现申请分配新的数组以保留缓存内容而需要线性的处理时间。

在使用 ArrayBuffer 之前，你必须首先从可变集合包中引用它：

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer
```

创建 ArrayBuffer 的时候，你必须指定它的类型参数，但可以不用指定长度。ArrayBuffer 可以自动调整分配的空间：

```
scala> val buf = new ArrayBuffer[Int]()
buf: scala.collection.mutable.ArrayBuffer[Int] =
  ArrayBuffer()
```

ArrayBuffer 还能用+=操作添加元素：

```
scala> buf += 12
scala> buf += 15

scala> buf
res16: scala.collection.mutable.ArrayBuffer[Int] =
  ArrayBuffer(12, 15)
```

所有数组能使用的方法数组缓存都能用。如可以获得 ArrayBuffer 的长度，或通过索引访问元素：

```
scala> buf.length
res17: Int = 2

scala> buf(0)
res18: Int = 12
```

334

队列（Queue）

如果你需要先进先出序列，可以使用 Queue。Scala 的集合库提供了可变和不可变的 Queue。以下方法创建空的不可变队列：

```
scala> import scala.collection.immutable.Queue
import scala.collection.immutable.Queue

scala> val empty = new Queue[Int]
empty: scala.collection.immutable.Queue[Int] = Queue()
```

你可以使用 enqueue 为不可变队列添加元素：

```
scala> val has1 = empty.enqueue(1)
has1: scala.collection.immutable.Queue[Int] = Queue(1)
```

如果要添加多个元素的话，可以把集合当作 enqueue 调用的参数：

```
scala> val has123 = has1.enqueue(List(2, 3))
has123: scala.collection.immutable.Queue[Int] = Queue(1,2,3)
```

从队列的头部移除元素，可以使用 dequeue：

```
scala> val (element, has23) = has123.dequeue
element: Int = 1
has23: scala.collection.immutable.Queue[Int] = Queue(2,3)
```

对于不可变队列来说，dequeue 方法将返回由队列头部元素和移除该元素之后的剩余队列组成的对偶（Tuple2）。

可变队列的使用方式与不可变队列一样，只是代之以 enqueue 方法，你可以使用+=及++=操作符添加元素。还有，对于可变队列来说，dequeue 方法将只从队列移除元素头并返回。举例如下：

```
scala> import scala.collection.mutable.Queue
import scala.collection.mutable.Queue

scala> val queue = new Queue[String]
queue: scala.collection.mutable.Queue[String] = Queue()

scala> queue += "a"

scala> queue ++= List("b", "c")

scala> queue
res21: scala.collection.mutable.Queue[String] = Queue(a, b, c)

scala> queue.dequeue
res22: String = a

scala> queue
res23: scala.collection.mutable.Queue[String] = Queue(b, c)
```

栈

如果需要的是后进先出的序列，你可以使用 Stack，它同样在 Scala 的集合库中也有可变和不可变版本。元素的推入使用 push，弹出使用 pop，只获取栈顶的元素而不移除可以使用 top。下面是使用可变栈的例子：

```
scala> import scala.collection.mutable.Stack
import scala.collection.mutable.Stack

scala> val stack = new Stack[Int]
stack: scala.collection.mutable.Stack[Int] = Stack()

scala> stack.push(1)

scala> stack
res1: scala.collection.mutable.Stack[Int] = Stack(1)

scala> stack.push(2)

scala> stack
res3: scala.collection.mutable.Stack[Int] = Stack(1, 2)

scala> stack.top
res8: Int = 2

scala> stack
res9: scala.collection.mutable.Stack[Int] = Stack(1, 2)

scala> stack.pop
res10: Int = 2

scala> stack
res11: scala.collection.mutable.Stack[Int] = Stack(1)
```

336

字符串（经 RichString 隐式转换）

RichString 也是应该知道的序列，它的类型是 Seq[Char]。因为 Predef 包含了从 String 到 RichString 的隐式转换，所以你可以把任何字符串当作 Seq[Char]。举例如下：

```
scala> def hasUpperCase(s: String) = s.exists(_.isUpperCase)
hasUpperCase: (String)Boolean

scala> hasUpperCase("Robert Frost")
res14: Boolean = true

scala> hasUpperCase("e e cummings")
res15: Boolean = false
```

本例的 hasUpperCase 方法体中，字符串 s 调用了 exists 方法。而 String 类本身并没有定义名为“exists”的方法，因此 Scala 编译器会把 s 隐式转换为含有这个方法的 RichString 类。exists 方法把字符串看作 Seq[Char]，并且如果所有的字符都是大写字母则返回真[2]。

17.3 集（Set）和映射（Map）

从前面第 3 章的第八步开始，你已经陆续看到了集和映射的基本操作。本节，我们将带给你更多的例子以深入剖析它们的用法。

正如前面提到过的，Scala 的集合库对集和映射提供了可变的和不可变的版本。集的层级已经展示在图 3.2 中，映射的层级展示在了图 3.3 中。就像在图中看到的，Set 和 Map 的简化名被用在三个特质

[2]注：第 1 章第 10 页的代码给出了同样的例子。

中，每个都位于不同的包里。

默认情况下在你使用“Set”或“Map”的时候，获得的都是不可变对象。如果需要的是可变版本，你需要先写明引用。Scala 让你更易于使用不可变的版本，期望能够以此方式鼓励你更倾向于使用它们而并非相对的可变版本。这种易于访问性来自于 Predef 对象的支持，它被每个 Scala 源文件隐含引用。清单 17.1 说明了相关的定义。

```
object Predef {
  type Set[T] = scala.collection.immutable.Set[T]
  type Map[K, V] = scala.collection.immutable.Map[K, V]
  val Set = scala.collection.immutable.Set
  val Map = scala.collection.immutable.Map
  // ...
}
```

清单 17.1 Predef 中定义的默认映射和集

Predef 中的“type”关键字用来把 Set 和 Map 定义为不可变集和映射特质的全引用名的别名[3]。Set 和 Map 的 val 分别被初始化为指向不可变 Set 和 Map 的单例对象。因此这里的 Map 就是 Predef.Map，根据定义还是 scala.collection.immutable.Map，对象和类型都是这样。

如果同一个源文件中既要用到可变版本，也要用到不可变版本的集合或映射，方式之一是引用包含了可变版本的包名：

```
scala> import scala.collection.mutable
import scala.collection.mutable
```

与以往一样，不可变集可以用 Set 指代，但现在还可以用 mutable.Set 指代可变集。举例如下：

```
scala> val mutaSet = mutable.Set(1, 2, 3)
mutaSet: scala.collection.mutable.Set[Int] = Set(3, 1, 2)
```

使用集

集的关键特性在于它可以使用对象的==操作检查，确保任何时候每个对象只在集中保留最多一个副本。例如，我们可以用集对字符串不同单词计数。

如果指定空格和标点符号为单词的分隔符，String 的 split 方法可以把字符串分割成单词。正则表达式“[!,.]+”足以完成该功能：它表明字符串应该在每个存在一到多个空格和/或标点符号的地方分割开来：

```
scala> val text = "See Spot run. Run, Spot. Run!"
text: java.lang.String = See Spot run. Run, Spot. Run!

scala> val wordsArray = text.split("[ !,.]+")
wordsArray: Array[java.lang.String] =
   Array(See, Spot, run, Run, Spot, Run)
```

要对不同的单词计数，你可以先把它们变为同样的大小写然后加到集中。由于集不能重复添加，因

[3]注：20.6 节将详细说明 type 关键字。

此不同的单词将只在集中出现一次。首先，可以使用 Set 伴生对象提供的 empty 方法创建空集：

```
scala> val words = mutable.Set.empty[String]
words: scala.collection.mutable.Set[String] = Set()
```

然后，用 for 表达式枚举所有的单词，分别把它们转换为小写字母形式，然后使用+=操作符添加到可变集中：

```
scala> for (word <- wordsArray)
         words += word.toLowerCase

scala> words
res25: scala.collection.mutable.Set[String] =
  Set(spot, run, see)
```

从而我们可以发现，文本中只含有 3 个不同的单词：spot、run 和 see。可变和不可变集最常用的方法见表 17.1。

339

表 17.1 集的常用操作

操作	行为
val nums = Set(1, 2, 3)	创建不可变集（nums.toString 返回 Set(1,2,3)）
nums + 5	添加元素（返回 Set(1, 2, 3, 5)）
nums - 3	删除元素（返回 Set(1, 2)）
nums ++ List(5, 6)	添加多个元素（返回 Set(1, 2, 3, 5, 6)）
nums -- List(1, 2)	删除多个元素（返回 Set(3)）
nums ** Set(1, 3, 5, 7)	获得交集（返回 Set(1, 3)）
nums.size	返回集中包含的对象数量（返回 3）
nums.contains(3)	检查是否包含（返回 true）
import scala.collection.mutable	引用可变集合类型
val words = mutable.Set.empty[String]	创建空可变集（words.toString 返回 Set()）
words += "the"	添加元素（words.toString 返回 Set(the)）
words -= "the"	如果存在元素，则删除（words.toString 返回 Set()）
words ++= List("do", "re", "mi")	添加多个元素（words.toString 返回 Set(do, re, mi)）
words --= List("do", "re")	删除多个元素（words.toString 返回 Set(mi)）
words.clear	删除所有元素（words.toString 返回 Set()）

340

使用映射

映射可以用来把值与集合中的元素联系起来。映射的使用与数组接近，只是你可以用任何类型的键，而不仅是使用从 0 开始计数的整数做索引。如果引用了 scala.collection.mutable 包，你还可以创建空的可变映射：

```
scala> val map = mutable.Map.empty[String, Int]
map: scala.collection.mutable.Map[String,Int] = Map()
```

请注意，在创建映射的时候，你须指定两个类型。第一个类型是用来定义映射的键（key），第二个用来定义值（value）。在这个例子中，键是字符串，值是整数。

映射中的条目设置与数组中的操作类似：

```
scala> map("hello") = 1

scala> map("there") = 2

scala> map
res28: scala.collection.mutable.Map[String,Int] =
  Map(hello -> 1, there -> 2)
```

类似地，读取映射和读取数组也是一样：

```
scala> map("hello")
res29: Int = 1
```

把上面的例子放在一起，就有了下面对字符串中单词出现次数做计数的方法：

```
scala> def countWords(text: String) = {
         val counts = mutable.Map.empty[String, Int]
         for (rawWord <- text.split("[ ,!.]+")) {
           val word = rawWord.toLowerCase
           val oldCount =
             if (counts.contains(word)) counts(word)
             else 0
           counts += (word -> (oldCount + 1))
         }
         counts
       }
countWords: (String)scala.collection.mutable.Map[String,Int]

scala> countWords("See Spot run! Run, Spot. Run!")
res30: scala.collection.mutable.Map[String,Int] =
  Map(see -> 1, run -> 3, spot -> 2)
```

经过计数之后，你就能发现这段文本说了许多“跑”的事，但没说太多“看”的事。

这段代码的工作原理如下：可变映射 counts 把每个单词映射到它在文本中出现的次数。对于每个文本中的单词，首先找到它的原始数量，加一，然后存回 counts。注意这里使用了 contains 检查是否已经有过这个单词。如果 counts.contains(word)返回结果为假，就说明这个单词没有出现过，那么它的原始数量就是零。

表 17.2 列举了可变和不可变映射的多数常用方法：

表 17.2 映射的常用操作

操作	行为
val nums = Map("i"->1, "ii"->2)	创建不可变映射（nums.toString 返回 Map(i->1, ii->2)）
nums +("vi"->6)	添加条目（返回 Map(i->1, ii->2, vi->6)）
nums - "ii"	删除条目（返回 Map(i -> 1)）

续表

操作	行为
`nums ++ List("iii"->3, "v"->5)`	添加多个条目（返回 `Map(i->1, ii->2, iii->3, v->5)`）
`nums -- List("i", "ii")`	删除多个条目（返回 `Map()`）
`nums.size`	返回映射的条目数量（返回 2）
`nums.contains("ii")`	检查是否包含（返回 `true`）
`nums("ii")`	获取指定键的关联值（返回 2）
`nums.keys`	返回键枚举器（返回字符串"i"和"ii"的 Iterator）
`nums.keySet`	返回键集（返回 `Set(i, ii)`）
`nums.values`	返回值枚举器（返回整数 1 和 2 的 Iterator）
`nums.isEmpty`	指明映射是否为空（返回 `false`）
`import scala.collection.mutable`	引用可变集合类型
`val words = mutable.Map.empty[String, Int]`	创建空可变映射
`words += ("one"->1)`	添加从"one"到 1 的映射条目（`words.toString` 返回 `Map(one->1)`）
`words -= "one"`	若存在映射条目，则删除（`words.toString` 返回 `Map()`）
`words ++= List("one"->1, "two"->2, "three"->3))`	添加多个映射条目（`words.toString` 返回 `Map(one->1, two->2, three->3)`）
`words --= List("one", "two")`	删除多个对象（`words.toString` 返回 `Map(three->3)`）

342

默认的（Default）集和映射

对于多数应用来说，由 `Set()`、`scala.collection.mutable.Map()`等的工厂方法提供的可变及不可变的集和映射的实现已经能够满足需求。这些工厂方法提供的实现都使用了快速查找算法，通常都涉及哈希表，因此它们能够快速反应对象是否存在于集合中。

如 `scala.collection.mutable.Set()` 工厂方法返回 `scala.collection.mutable.HashSet`，则其在内部使用了哈希表。类似地，`scala.collection.mutable.Map()` 工厂方法返回了 `scala.collection.mutable.HashMap`。

不可变集和映射的情况更为复杂一些。例如，`scala.collection.immutable.Set()` 工厂方法返回的类，取决于你传递给它多少元素，具体说明参见表 173。对于少于 5 个元素的集，类型完全取决于它的元素数量，以获得最优的性能。然而一旦你请求的集包含了 5 个元素以上，工厂方法返回的将是不可变的 `HashSet`。

343

表 17.3 默认的不可变集实现

元素数量	实现
0	scala.collection.immutable.EmptySet
1	scala.collection.immutable.Set1
2	scala.collection.immutable.Set2
3	scala.collection.immutable.Set3
4	scala.collection.immutable.Set4
5 或更多	scala.collection.immutable.HashSet

与之类似，scala.collection.immutable.Map()工厂方法返回的类取决于传递进去的键值对数量，参见表 17.4。对于少于 5 个元素的不可变映射，类型完全取决于其键值对数量，以获得最优的性能。但如果包含了 5 个或以上的键值对，则使用的是不可变的 HashMap。

表 17.4 默认的不可变映射实现

元素数量	实现
0	scala.collection.immutable.EmptyMap
1	scala.collection.immutable.Map1
2	scala.collection.immutable.Map2
3	scala.collection.immutable.Map3
4	scala.collection.immutable.Map4
5 或更多	scala.collection.immutable.HashMap

表 17.3 和表 17.4 中的不可变实现类的默认实现之目的就是为了能够带给你最优的性能。例如，如果你向 EmptySet 中添加元素，它将返回给你 Set1。如果向 Set1 添加元素，将返回 Set2。之后如果从 Set2 中移除元素，将返回另一个 Set1。

有序的（Sorted）集和映射

有时，可能你需要集或映射的枚举器能够返回按特定顺序排列的元素。为此，Scala 的集合库提供了 SortedSet 和 SortedMap 特质。这两个特质分别由类 TreeSet 和 TreeMap 实现，它们都使用了红黑树有序地保存元素（TreeSet 类）或键（TreeMap 类）。具体的顺序取决于 Ordered 特质，集的元素类型或映射的键类型必须要么混入，要么能够隐式地转换成 Ordered 特质。这些类只有不可变类型的版本。下面是 TreeSet 的例子：

```
scala> import scala.collection.immutable.TreeSet
import scala.collection.immutable.TreeSet

scala> val ts = TreeSet(9, 3, 1, 8, 0, 2, 7, 4, 6, 5)
ts: scala.collection.immutable.SortedSet[Int] =
  Set(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> val cs = TreeSet('f', 'u', 'n')
cs: scala.collection.immutable.SortedSet[Char] = Set(f, n, u)
```

下面是 TreeMap 的例子：

```
scala> import scala.collection.immutable.TreeMap
import scala.collection.immutable.TreeMap

scala> var tm = TreeMap(3 -> 'x', 1 -> 'x', 4 -> 'x')
tm: scala.collection.immutable.SortedMap[Int,Char] =
  Map(1 -> x, 3 -> x, 4 -> x)

scala> tm += (2 -> 'x')

scala> tm
res38: scala.collection.immutable.SortedMap[Int,Char] =
. Map(1 -> x, 2 -> x, 3 -> x, 4 -> x)
```

同步的（Synchronized）集和映射

在 1.1 节，我们曾经提到过如果你需要线程安全的映射，可以把 SynchronizedMap 特质混入到你想要的特定类实现中。例如，把 SynchronizedMap 混入 HashMap，见清单 17.2。这个例子首先引入两个特质，Map 和 SynchronizedMap，以及一个类 HashMap，它们全部引用自 scala.collection.mutable 包。例子余下的部分是单例对象 MapMaker 的定义，声明了方法 makeMap。该方法声明了返回类型为字符串键到字符串值的可变映射。

345

```
import scala.collection.mutable.{Map,
    SynchronizedMap, HashMap}

object MapMaker {

  def makeMap: Map[String, String] = {

      new HashMap[String, String] with
          SynchronizedMap[String, String] {
        override def default(key: String) =
          "Why do you want to know?"
      }
  }
}
```

清单 17.2 混入 SynchronizedMap 特质

makeMap 方法体的第一个语句构造了新的可变 HashMap 并混入了 SynchronizedMap 特质：

```
new HashMap[String, String] with
  SynchronizedMap[String, String]
```

根据这段代码，Scala 编译器将产生混入了 SynchronizedMap 的 HashMap 合成子类，并创建（以及返回）它的实例。根据以下代码，这个合成子类还将重载名为 default 的方法：

```
override def default(key: String) =
  "Why do you want to know?"
```

如果你请求映射返回与特定键关联的值，而该键的映射实际不存在，默认你将得到 NoSuchElementException。然而如果你定义了新的映射类并重载了 default 方法，那么这个新映射将在查询不存在的键时返回 default 方法的返回值。因此，根据清单 17.2 由编译器构造的合成 HashMap 子类将在查询不存在的键时返回一句稍显无理的回应字符串，"Why do you want to know?"[4]。

[4]译注：为什么你想知道？

由于 makeMap 方法返回的可变映射混入了 SynchronizedMap 特质，因此可以立即用于多线程环境。每次对映射的访问都被同步操作。下面的例子演示了在解释器中，单线程访问映射的情况：

346

```
scala> val capital = MapMaker.makeMap
capital: scala.collection.mutable.Map[String,String] = Map()

scala> capital ++ List("US" -> "Washington",
           "Paris" -> "France", "Japan" -> "Tokyo")
res0: scala.collection.mutable.Map[String,String] =
  Map(Paris -> France, US -> Washington, Japan -> Tokyo)

scala> capital("Japan")
res1: String = Tokyo

scala> capital("New Zealand")
res2: String = Why do you want to know?

scala> capital += ("New Zealand" -> "Wellington")

scala> capital("New Zealand")
res3: String = Wellington
```

你可以按照与创建同步映射相类似的方法创建同步集。例如可以通过混入 SynchronizedSet 特质创建同步的 HashSet，见下：

```
import scala.collection.mutable

val synchroSet =
  new mutable.HashSet[Int] with
      mutable.SynchronizedSet[Int]
```

最后，在需要考虑使用同步集合的时候，你也可以考虑使用 java.util.concurrent 的并发集合。又或者，还可以使用非同步的集合及 Scala 的 actor。actor 将在第 30 章详细介绍。

17.4 可变（mutable）集合 vs.不可变（immutable）集合

对于某些问题来说，可变集合能够处理得更好；而另外一些，不可变集合更为适合。如果拿不定主意，可以首先使用不可变集合，之后如果实在需要也可以再转为可变的，因为不可变集合比可变集合更易于理清头绪。

347

有的时候也可以反过来，如果你发现使用可变集合的代码复杂的像一团乱麻时，你就应该考虑是否适当改为使用不可变集合会更有帮助一些。特别是如果你发现一直在担忧是否要在合适的地方复制可变集合的拷贝，或者思考很多关于谁“主宰”或“拥有”可变集合的时候，那么请考虑把集合替换成不可变类型的可能性。

除了更易于掌控之外，如果集合中元素数量不多，不可变集合的数据存储还常常比可变集合更为紧凑。例如 HashMap 默认实现的空可变映射占用大约 80 字节，而且每添加一个元素就多占用 16 字节。而空的不可变 Map 是可以在所有引用中共享的单例对象，因此指向它本质上仅需一个指针字段。还有，Scala 集合库当前保存了单个对象所含元素不超过 4 个的不可变映射和集的定义，它们分别占用 16 到 40 字节不等，这取决于集合中的元素数量[5]。因此对于较小的映射或集来说，不可变版本比可变版本的对

[5]注：“单个对象”是指从 Set1 到 Set4，或 Map1 到 Map4 的实例，参见表 17.3 和表 17.4。

象更为紧凑。由于多数集合都比较小，所以把它们转换为不可变类型将节省大量空间并提高工作性能。

为了更易于完成不可变集合到可变集合的转换，或反向转换，Scala 提供了一些语法糖。纵使不可变集和映射并不支持真正的+=方法，Scala 还是为此提供了+=的语法解释。如果你写了 a += b，而 a 不支持名为+=的方法，Scala 将尝试把它解释为 a = a + b。例如，不可变集虽然不支持+=操作符：

```
scala> val people = Set("Nancy", "Jane")
people: scala.collection.immutable.Set[java.lang.String] =
  Set(Nancy, Jane)

scala> people += "Bob"
<console>:6: error: reassignment to val
       people += "Bob"
              ^
```

348

然而如果声明 people 为 var，而不是 val，那么集合就可以用+=操作完成“更新”，尽管它是不可变类型的。首先，新创建集合；然后，people 将被重新赋值为新集合：

```
scala> var people = Set("Nancy", "Jane")
people: scala.collection.immutable.Set[java.lang.String] =
  Set(Nancy, Jane)

scala> people += "Bob"

scala> people
res42: scala.collection.immutable.Set[java.lang.String] =
  Set(Nancy, Jane, Bob)
```

经过一系列操作之后，people 变量现在指向新的不可变集，其中包含了添加的字符串"Bob"。同样的理念可以应用于所有以=结尾的方法，而不仅是+=方法。以下的代码把该语法应用于-=操作符，从集中移除元素；以及++=操作符，把集合的所有元素添加到集里面：

```
scala> people -= "Jane"

scala> people ++= List("Tom", "Harry")

scala> people
res45: scala.collection.immutable.Set[java.lang.String] =
  Set(Nancy, Bob, Tom, Harry)
```

为了说明这种语法糖为何有用，请再次考虑 1.1 节出现过的 Map 例子：

```
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

这段代码使用了不可变集。如果你想尝试改用可变集合，仅需引用可变版本的 Map 即可，这样就可以重写对不可变 Map 的默认引用：

```
import scala.collection.mutable.Map // 唯一的改变!
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

349

并不是所有的例子都易于修改，但对于以等号结尾的方法的特别处理通常可减少需修改的代码量。

另外，这块语法糖不仅对集合有效，它还具有普适性。例如，下面是用在浮点数上的：

```
scala> var roughlyPi = 3.0
roughlyPi: Double = 3.0

scala> roughlyPi += 0.1

scala> roughlyPi += 0.04

scala> roughlyPi
res48: Double = 3.14
```

这种扩展的效果类似于 Java 的赋值操作符+=、-=、*=等，不过其更为通用，因为所有以=结尾的操作符都可以做转化。

17.5 初始化集合

正如之前所见，最常见的创建和初始化集合的办法是把初始值传递给要用的集合类型的伴生对象的工厂方法。你只须把元素放在伴生对象名后面的括号中，Scala 编译器就会把它转化为该伴生对象的 apply 方法调用：

```
scala> List(1, 2, 3)
res0: List[Int] = List(1, 2, 3)

scala> Set('a', 'b', 'c')
res1: scala.collection.immutable.Set[Char] = Set(a, b, c)

scala> import scala.collection.mutable
import scala.collection.mutable

scala> mutable.Map("hi" -> 2, "there" -> 5)
res2: scala.collection.mutable.Map[java.lang.String,Int] =
  Map(hi -> 2, there -> 5)

scala> Array(1.0, 2.0, 3.0)
res3: Array[Double] = Array(1.0, 2.0, 3.0)
```

尽管通常都可以让 Scala 编译器从传递给工厂方法的元素推断集合的元素类型，但有些时候或许你会希望指定以不同于编译器所选的类型创建集合。尤其对于可变集合来说更为如此。举例如下：

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val stuff = mutable.Set(42)
stuff: scala.collection.mutable.Set[Int] = Set(42)

scala> stuff += "abracadabra"
<console>:7: error: type mismatch;
 found   : java.lang.String("abracadabra")
 required: Int
       stuff += "abracadabra"
                ^
```

这里的问题在于 stuff 被指定元素类型为 Int。如果想要让它的类型为 Any，你需要明确地说明，把元素类型放在方括号中，像这样：

```
scala> val stuff = mutable.Set[Any](42)
stuff: scala.collection.mutable.Set[Any] = Set(42)
```

另一种特殊情况是，你想要把集合初始化为指定类型。例如，设想你要把列表中的元素保存在TreeSet中。列表如下：

```
scala> val colors = List("blue", "yellow", "red", "green")
colors: List[java.lang.String] =
  List(blue, yellow, red, green)
```

你不能把colors列表传递给TreeSet工厂方法：

```
scala> import scala.collection.immutable.TreeSet
import scala.collection.immutable.TreeSet

scala> val treeSet = TreeSet(colors)
<console>:6: error: no implicit argument matching
  parameter type (List[java.lang.String]) =>
    Ordered[List[java.lang.String]] was found.
       val treeSet = TreeSet(colors)
                     ^
```

实际上，需要创建空的TreeSet[String]对象并使用TreeSet的++操作符把列表元素加入其中：

```
scala> val treeSet = TreeSet[String]() ++ colors
treeSet: scala.collection.immutable.SortedSet[String] =
   Set(blue, green, red, yellow)
```

数组与列表之间的互转

另一方面，如果你需要用集合初始化列表或数组，非常简单，直接做就是了。正如你之前看到的，使用集合初始化列表，只须对集合调用toList方法：

```
scala> treeSet.toList
res54: List[String] = List(blue, green, red, yellow)
```

或者，如果需要的是数组，则调用toArray：

```
scala> treeSet.toArray
res55: Array[String] = Array(blue, green, red, yellow)
```

尽管原始的colors列表没有做排序，但对TreeSet调用toList产生的列表元素是按照字母顺序排列的。对集合调用toList或toArray方法时，结果列表或数组里的元素顺序与在集合上调用elements方法获得的枚举器产生的顺序一致。因为TreeSet[String]的枚举器产生的字符串是按照字母顺序排列的，所以对TreeSet调用toList产生的结果列表中的字符串也是按照字母顺序排列的。

然而，请牢记，转变为列表或数组同样需要复制集合的所有元素，因此对于大型集合来说可能比较慢。尽管如此，由于现存API的缘故有时你必须这么做。但更进一步说，许多集合只不过保存了少量元素，这种情况下速度只会慢一点点。

集和映射的可变与不可变互转

另一种偶尔发生的情况是把可变集或映射转换成不可变类型，或者反向转换。要做到这一点，你可以利用前面以列表元素初始化TreeSet时演示的技巧。比如说你想要把当前使用的可变集合转换成不可变集合，可以先创建空不可变集合，然后把可变集合的元素用++操作符添加进去。下面的代码说明了

如何把前面例子里不可变 TreeSet 转换成可变集，再转回不可变集：

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> treeSet
res5: scala.collection.immutable.SortedSet[String] =
  Set(blue, green, red, yellow)

scala> val mutaSet = mutable.Set.empty ++ treeSet
mutaSet: scala.collection.mutable.Set[String] =
  Set(yellow, blue, red, green)

scala> val immutaSet = Set.empty ++ mutaSet
immutaSet: scala.collection.immutable.Set[String] =
  Set(yellow, blue, red, green)
```

你可以使用同样的技巧实现可变映射与不可变映射之间的转换：

```
scala> val muta = mutable.Map("i" -> 1, "ii" -> 2)
muta: scala.collection.mutable.Map[java.lang.String,Int] =
    Map(ii -> 2, i -> 1)

scala> val immu = Map.empty ++ muta
immu: scala.collection.immutable.Map[java.lang.String,Int] =
    Map(ii -> 2, i -> 1)
```

17.6 元组

第 3 章的第九步曾提到过，元组可以把固定数量的条目组合在一起以便于作为整体传送。不像数组或列表，元组可以保存不同类型的对象。下面是可以作为整体保存整数、字符串和控制台的元组：

```
(1, "hello", Console)
```

元组节省了定义简单到只包含数据的类的时间。即使定义类再简单，但它的确还是需要很少量的有时还是无意义的工作。元组省掉你选择类名，选择类的定义域，以及选择类成员名的工作。如果类只须保存整数和字符串，那么定义 AnIntegerAndAString 这样的类丝毫无助于代码的清晰整洁。

由于元组可以组合不同类型的对象，因此它不能继承自 Iterable。如果你发现自己想要的是把“一个”整数和“一个”字符串组合在一起，那么你需要的就是元组，不是 List，也不是 Array。

元组常用来返回方法的多个值。例如，下面的方法找到集合中的最长单词并返回它的索引：

```
def longestWord(words: Array[String]) = {
  var word = words(0)
  var idx = 0
  for (i <- 1 until words.length)
    if (words(i).length > word.length) {
      word = words(i)
      idx = i
    }
  (word, idx)
}
```

以下是方法用例：

```
scala> val longest =
         longestWord("The quick brown fox".split(" "))
longest: (String, Int) = (quick,1)
```

354 这里的 longestWord 方法需计算两个条目：数组中最长的单词 word，以及该单词的索引 idx。为了简化设计，函数假定列表至少有一个单词，并且如果单词一样长，就选择列表中最早出现的。一旦选定了要返回的单词和索引，则使用元组语法(word, idx)把它们同时返回。

要访问元组的元素，可以使用方法_1 访问第一个元素，_2 访问第二个，以此类推：

```
scala> longest._1
res56: String = quick

scala> longest._2
res57: Int = 1
```

而且，你还可以把元组的每个元素赋值给它自己的变量[6]，如下：

```
scala> val (word, idx) = longest
word: String = quick
idx: Int = 1

scala> word
res58: String = quick
```

另外，如果你去掉了括号，将得到不同的结果：

```
scala> val word, idx = longest
word: (String, Int) = (quick,1)
idx: (String, Int) = (quick,1)
```

这种语法对多重定义（multiple definitions）指定了相同的表达式。每个变量被初始化为右侧表达式的单次执行结果。不过本例中表达式执行的结果为元组，并没有什么差别。两个变量分别被初始化为相互独立的元组。第 18 章可以看到一些多重定义有效发挥作用的例子。

唯一需要当心的就是，元组实在是太过易于使用了。当你想要把数据以无外乎“A 与 B”的意义组合起来的时候，元组是最好用的。然而，如果这种组合要加入实在的意义，或者你想要为这种组合添加一些方法，那么最好还是创建类吧。例如，请不要使用 3 元组组合月、日与年。请创建 Date 类。这样做能够表明你的目的，从而既有助于代码的阅读，又能够让语法及编译器帮你捕获错误。

355

17.7 小结

本章概览了 Scala 的集合类库，以及其中最为重要的类和特质。有了本章做基础，你将能够有效地使用 Scala 的集合，并且知道在需要更多信息的时候到哪里去查找 Scaladoc。下一章，我们将把注意力从 Scala 库转回语言本身，并讨论 Scala 对可变对象的支持。

356

[6]注：这种语法实际就是模式匹配的特例，详情参见 15.7 节。

第 18 章

有状态的对象

上一章里，为了着力说明对象不带可变状态的理念，我们把重点放在函数式（不可变）对象上。然而，Scala 中定义带可变状态的对象也是完全可能的。当你需要对真实世界中不断改变的对象建模时，自然而然地会想到这样的有状态对象。

本章解释什么是有状态的对象，以及 Scala 的表达语法。第二部分介绍了用于研究离散事件模拟的较大的用例，其中使用了有状态对象以建造内部的域特定语言（DSL）来定义仿真的数字电路。

18.1 什么让对象具有状态?

即使不了解对象的实现细节，你也能够观察到纯函数式对象与有状态对象之间的本质差别。如果对某个纯函数式对象调用方法或获取其字段的值，你将永远得到同样的结果。例如，对于如下字符列表：

```
val cs = List('a', 'b', 'c')
```

调用 cs.head 将永远返回 'a'。无论在 cs 的定义与调用 cs.head 方法之间对 cs 执行多少操作，都不能改变这个事实。

相反，对于有状态的对象来说，方法调用或字段访问的结果将取决于之前已经执行过的操作。银行账号就是很好的说明有状态对象的例子。清单 18.1 演示了银行账号的简化实现：

```
class BankAccount {
  private var bal: Int = 0

  def balance: Int = bal

  def deposit(amount: Int) {
    require(amount > 0)
    bal += amount
  }

  def withdraw(amount: Int): Boolean =
    if (amount > bal) false
    else {
      bal -= amount
      true
    }
}
```

清单 18.1 可变的银行账号类

BankAccount 类定义了私有变量 bal，以及三个公开方法：balance 返回当前余额；deposit 向 bal 添加指定 amount 的金额；以及 withdraw 尝试从 bal 减少指定 amout 的金额并须要确保操作之后的余额不能为负数。withdraw 的返回值为 Boolean 类型，说明请求的资金是否被成功提取。

即使你完全不了解 BankAccount 类内部的工作原理，你仍然可以断定 BankAccount 是有状态的对象：

```
scala> val account = new BankAccount
account: BankAccount = BankAccount@bf5bb7

scala> account deposit 100

scala> account withdraw 80
res1: Boolean = true

scala> account withdraw 80
res2: Boolean = false
```

358

注意上面交互的最后两次提款操作返回不同的值。第一次提款操作返回 true，因为银行账户包含了足够提取的资金；第二次操作，尽管与第一次相同，但却返回 false，因为账户余额已经减少，不够提取了。所以很清楚，银行账户有可变的状态，对于不同的时间，相同的操作将返回不同的结果。

你或许认为 BankAccount 具有状态的原因体现在它内部存在 var 变量定义。虽然状态与 var 类型变量通常携手而来，但它们的关系并非总是如此明确。例如，类即使没有定义或继承任何 var 变量，也可以由于把方法调用传递给其他有可变状态的对象而带有状态。反过来也同样有可能：类即使包含了 var 变量也可以仍然是纯函数的。例如，把高耗能操作的结果缓存在字段中，以优化为目的的类。为了说明这样的例子，可以假设有以下的未被优化的类 Keyed，以及高耗能操作 computeKey：

```
class Keyed {
  def computeKey: Int = ... // 这会消耗一定的时间
  ...
}
```

只要 computeKey 操作既不读取也不写入任何 var 类型变量，你就可以添加操作缓存以使 Keyed 类更为高效：

```
class MemoKeyed extends Keyed {
  private var keyCache: Option[Int] = None
  override def computeKey: Int = {
    if (!keyCache.isDefined) keyCache = Some(super.computeKey)
    keyCache.get
  }
}
```

359

使用 MemoKeyed 取代 Keyed 可以加快运行速度，第二次请求 computeKey 操作的结果时，keyCache 字段存储的值可以直接返回，无须再次运行 computeKey，除此之外，Keyed 与 MemoKeyed 完全一致。因此，如果 Keyed 是纯函数式的，那么 MemoKeyed 也是，尽管其含有可重新赋值的变量。

18.2 可重新赋值的变量和属性

你可以对可重新赋值的变量执行两种基本操作：获取它的值，或者把它设定为新值。在类似 JavaBeans

这样的库中，操作常被封装进明确定义的不同的 getter 及 setter 方法中。Scala 里，对象的每个非私有的 var 类型成员变量都隐含定义了 getter 和 setter 的方法。然而这些 getter 和 setter 方法的命名方式并没有沿袭 Java 的约定。var 变量 x 的 getter 方法命名为“x”，它的 setter 方法命名为“x_=”。

例如，如果类中存在 var 定义：

```
var hour = 12
```

则除了可重新赋值的字段之外，还会额外产生 getter 方法“hour”，以及 setter 方法“hour_=”。该字段始终被标记为 private[this]，表示它只能被包含它的对象访问。但是 getter 和 setter 方法获得了原本 var 变量的可见性。如果 var 定义为 public，则它的 getter 和 setter 与之相同，如果定义为 protected，那么它们也是 protected，以此类推。

例如，清单 18.2 演示的 Time 类，定义了两个公开 var 变量 hour 和 minute：

```
class Time {
  var hour = 12
  var minute = 0
}
```

清单 18.2 带有公开 var 变量的类

这种实现方式与清单 18.3 里的类定义完全相同。类定义里本地字段 h 和 m 的命名是随意选取的，目的是不与任何已经存在的名称冲突。

```
class Time {
  private[this] var h = 12
  private[this] var m = 0

  def hour: Int = h
  def hour_=(x: Int) { h = x }

  def minute: Int = m
  def minute_=(x: Int) { m = x }
}
```

清单 18.3 公开的 var 变量是如何扩展为 getter 和 setter 方法的

var 变量扩展为 getter 和 setter 方法的有趣之处在于，你也可以选择直接定义 getter 及 setter 方法以取代定义 var 变量。通过直接定义这些访问方法，你可以依照自己的意愿解释对变量的访问及赋值操作。例如，清单 18.4 中 Time 类的变体包含了捕获所有使用非法值赋给 hour 和 minute 的需求。

```
class Time {
  private[this] var h = 12
  private[this] var m = 12

  def hour: Int = h
  def hour_= (x: Int) {
    require(0 <= x && x < 24)
    h = x
  }

  def minute = m
  def minute_= (x: Int) {
```

```
    require(0 <= x && x < 60)
    m = x
  }
}
```

清单 18.4 直接定义 getter 和 setter 方法

某些语言对这些并非普通变量的类变量体提供了特别的语法结构，以便重定义它们的 getter 或 setter 方法。例如，C#有属性，足以充当这个角色。而 Scala 的始终把变量解释为一对 setter 和 getter 方法的约定带给你实际上与 C#的属性相同的能力，却不需要特殊的语法。属性能够满足许多不同的需要。在清单 18.4 的例子中，setter 方法始终强制执行合法性检查，从而保护变量不被赋为非法值。你也可以使用属性记录所有对变量的 getter 或 setter 方法的访问。或者你还可以把变量和事件集成在一起，例如可以在每次修改变量时通知相关的事件订阅者（你将在第 33 章看到这个例子）。

你也可以只定义 getter 和 setter 方法而不带有关联字段（见清单 18.5），这种做法不但可行，有时甚至很有必要。下面的 Thermometer 类就是这样的例子。它包装了可读可更新的温度变量，温度可以表达为摄氏度或华氏度。下面的类允许你以这两种测量标准获得或设置温度。

```
class Thermometer {

  var celsius: Float = _

  def fahrenheit = celsius * 9 / 5 + 32
  def fahrenheit_= (f: Float) {
    celsius = (f - 32) * 5 / 9
  }
  override def toString = fahrenheit +"F/"+ celsius +"C"
}
```

清单 18.5 定义不带关联字段的 getter 和 setter

类函数体的第一行定义了 var 变量 celsius，以保存摄氏度计量的温度值，celsius 变量初始化设置为缺省值‘_’，这个符号指定了变量的“初始化值”。更为精确地说，字段的初始化器“= _”把零值赋给该字段。这里的“零”的取值取决于字段的类型。对于数值类型来说是 0，布尔类型是 false，引用类型则是 null。

请注意，Scala 中不可以随意省略“= _”初始化器。如果写成：

```
var celsius: Float
```

这将定义为抽象变量[1]，而不是未初始化的变量[2]。

celsius 变量定义之后是 getter 方法“fahrenheit”，以及 setter 方法，“fahrenheit_=”，它们可以存取同样的温度变量，但以华氏度计量标准进行。并没有独立的字段保存当前华氏度的温值。不过 getter 和 setter 方法会自动把摄氏度转变为相对的华氏度。下面的例子说明了与 Thermometer 对象的交互操作：

```
scala> val t = new Thermometer
t: Thermometer = 32.0F/0.0C
```

[1]注：抽象变量将在第 20 章说明。

[2]译注：很希望建议 Scala 专家组对抽象变量在前面加上 abstract 以示区别。因为对于 Java 程序员来说，定义变量不加初始化值可能是个很习惯的动作。

```
scala> t.celsius = 100

scala> t
res3: Thermometer = 212.0F/100.0C

scala> t.fahrenheit = -40

scala> t
res4: Thermometer = -40.0F/-40.0C
```

18.3 案例研究：离散事件模拟

本章余下的部分将通过一个扩展的例子演示有状态的对象如何以有趣的方式与头等的函数值组合在一起。你将看到数字电路模拟器的设计和实现。这个任务被拆分为若干个子问题，每个都很有趣：首先，你将看到为数字电路定制的小语言。这种语言的定义将凸显在 Scala 这样的宿主语言中定义嵌入式域特定语言的通用方法。第二，我们将呈现一个简单但通用的离散事件模拟框架。这个框架的主要工作是保持对执行在模拟时间中的动作的跟踪。最后，我们将演示如何架构和建造离散模拟程序。这种模拟的理念在于通过对象的模拟对物理对象建模，并使用模拟框架对物理时间建模。

这个例子取自经典教材 “Structure and Interpretation of Computer Programs”，作者为 Abelson 和 Sussman [Abe96][3]。差别在于实现语言为 Scala 而不是 Scheme，并且例子的许多方面被构造为四层软件：模拟框架，基本电路模拟包，用户定义电路库，以及每个模拟电路本身。每一层都被表达为类，特化层（上层）继承自通用层（下层）。

363

快速通道

理解本章的离散事件模拟例子需要花一定的时间。如果你认为你宁愿用这些时间继续学习 Scala 的更多知识，也可以安全地前进到下一章。

18.4 为数字电路定制的语言

我们的工作可以开始于描述数字电路的 “小语言”。数字电路由线路和功能盒建造起来。线路上加载了信号，经功能盒发生转变。信号可以表达为布尔值：`true` 是有信号，`false` 是没有信号。

图 18.1 说明了三种基本的功能盒（或称为门）：

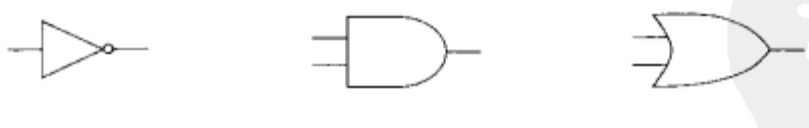


图 18.1 基本门电路

- 反转器（Inverter）[4]，输出反转信号。

[3]译注：中文版书名《计算机程序的构造和解释》，译者：裘宗燕。
[4]译注：或称为非门。

- 与门（and-Gate），输出其输入信号的合取值（conjunction）[5]。
- 或门（or-Gate），输出其输入信号的析取值（disjunction）[6]。

这些门足以建造所有其他的功能盒。门有延迟，因此门的输出改变必须在输入改变的一定时间之后才能发生。

364

我们将使用下列 Scala 类和函数描述数字电路的各个元件。首先 `Wire` 类指代线路。我们可以这样构造线路：

```
val a = new Wire
val b = new Wire
val c = new Wire
```

或者，更简洁地写成：

```
val a, b, c = new Wire
```

其次，我们需要的基本门电路由以下三个过程“制造”：

```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

考虑到 Scala 对于函数性的强调，这里不同寻常的地方在于，是由这些过程的副作用构造了门电路，而并非返回构造好的门电路作为结果。例如 `inverter(a, b)` 的调用在线路 `a` 和 `b` 之间放置了反转器。这是由于这种副作用的构造使得循序渐进地构造复杂电路变得更为容易。还有，尽管方法经常定义为动词名称，但这里的是名词名称，以便说明它们制造的是哪个门电路。这反映了 DSL 说明的本质：应该描述电路，而并非如何制造它。

复杂的功能盒可以用基本门电路制做。例如，清单 18.6 演示的方法可以构造半加法器（half-adder）。`halfAdder` 方法带两个输入，`a` 和 `b`，产生累加和 `s`，其定义为“`s = (a + b) % 2`”及进位 `c`，定义为“`c = (a + b) / 2`”。半加法器的图表展示在图 18.2 中。

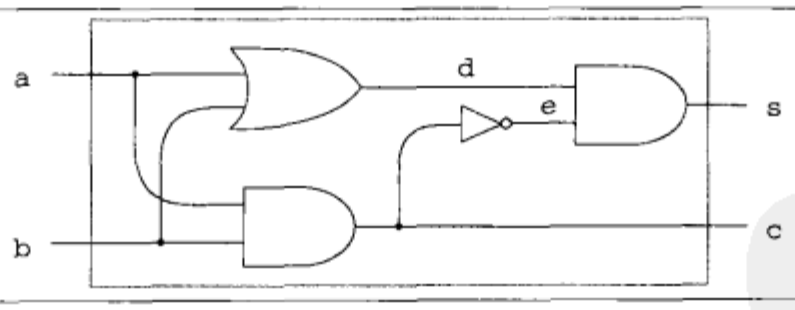


图 18.2 半加法电路

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {
  val d, e = new Wire
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)
```

[5]译注：即“交”或者称为“逻辑与”。
[6]译注：即“并”或者称为“逻辑或”。

```
    andGate(d, e, s)
  }
```

清单 18.6 halfAdder 方法

请注意 halfAdder 与前面的三个构造基本门电路的方法一样，是参数化的功能盒。你可以使用 halfAdder 方法构造更为复杂的电路。例如，清单 18.7 定义了一位全加法器，参见图 18.3。它带两个输入，a 和 b，及一个进位 cin，产生累加输出，定义为“sum = (a + b + cin) %2”，及进位输出，定义为“cout = (a + b + cin) / 2”。

```
  def fullAdder(a: Wire, b: Wire, cin: Wire,
      sum: Wire, cout: Wire) {

    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
```

清单 18.7 fullAdder 方法

图 18.3 全加法电路

Wire 类及函数 inverter、andGate 和 orGate 表达的小语言可以让用户实现数字电路的定义。这是内部 DSL 的很好的例子，通过使用宿主语言将域特定语言定义为库，而不是完全实现这种语言。

• 电路 DSL 的实现仍然需要大量的工作。既然用 DSL 定义电路的目的在于对电路的模拟，因此合理的想法是让这种 DSL 实现能够基于离散事件模拟的通用 API。后面的两节将首先带给你模拟 API 实现及在其之上的电路 DSL 实现。

18.5 Simulation API

模拟 API 参见清单 18.8。由 org.stairwaybook.simulation 包里的 Simulation 类组成。具体的模拟库将继承该类，并用域特定功能加以强化。Simulation 类的原件将在本节给出。

离散事件模拟在特定的时间执行用户定义的动作。动作由具体的模拟子类定义；均共享了常规类型：

```
type Action = () => Unit
```

这个语句把 Action 定义为带空参数列表并返回 Unit 的过程的别名。Action 是 Simulation 类的类型成员（type member）。你可以把它当作是类型 () => Unit 的更好读的名称。类型成员将在 20.6 节说明。

动作执行的时间是模拟时间；与实际的“挂钟”时间无关。模拟时间可以直接表达为整数。当前的模拟时间被保存在私有变量中：

```
private var curtime: Int = 0
```

该变量提供了公开的访问方法，可以获取当前时间：

```
def currentTime: Int = curtime
```

私有变量与公开访问方法的组合可以用来保证当前时间不被 Simulation 类外部的方法改变。毕竟通常不希望模拟对象能够操纵当前时间，除非说你在使用模拟器对时间旅行建模。

367

```
abstract class Simulation {

  type Action = () => Unit

  case class WorkItem(time: Int, action: Action)
  private var curtime = 0
  def currentTime: Int = curtime

  private var agenda: List[WorkItem] = List()

  private def insert(ag: List[WorkItem],
      item: WorkItem): List[WorkItem] = {

    if (ag.isEmpty || item.time < ag.head.time) item :: ag
    else ag.head :: insert(ag.tail, item)
  }

  def afterDelay(delay: Int)(block: => Unit) {
    val item = WorkItem(currentTime + delay, () => block)
    agenda = insert(agenda, item)
  }

  private def next() {
    (agenda: @unchecked) match {
      case item :: rest =>
        agenda = rest
        curtime = item.time
        item.action()
    }
  }

  def run() {
    afterDelay(0) {
      println("*** simulation started, time = "+
          currentTime +" ***")
    }
    while (!agenda.isEmpty) next()
  }
}
```

368

清单 18.8 Simulation 类

需要在特定时间执行的动作被称为工作条目（work item）。工作条目实现为如下类：

```
case class WorkItem(time: Int, action: Action)
```

我们把 WorkItem 类定义为样本类，因为通过继承将获得如下语法层面的便捷：你可以使用工厂方法 WorkItem 创建类实例，并且免费获得访问构造器参数 time 和 action 的方法。同样请注意 WorkItem 类内嵌在 Simulation 类中。Scala 的内嵌类的处理方式与 Java 相同。20.7 节会更详细地说明。

Simulation 类保存了所有剩余的尚未执行的工作条目的排程表（agenda）。工作条目以将要执行的模拟时间顺序排程：

```
private var agenda: List[WorkItem] = List()
```

agenda 列表将保持合适的顺序，并通过 insert 方法加以更新。你可以看到 afterDelay 调用了 insert，这是把工作条目加入排程表的唯一方式：

```
def afterDelay(delay: Int)(block: => Unit) {
  val item = WorkItem(currentTime + delay, () => block)
  agenda = insert(agenda, item)
}
```

正如名字所暗示的，这个方法把（由 block 引入的）动作插入到排程表，使得该动作被排在当前模拟时间之后的 delay 时间单位上执行。例如，以下的调用将创建新的工作条目，在 currentTime + delay 模拟时间点执行：

```
afterDelay(delay) { count += 1 }
```

将被执行的代码保存在方法的第二个参数中。该参数的标准类型为“=> Unit”，也就是说，这是通过传名调用的类型为 Unit 的计算。请回想一下，传名参数在传递给方法的时候是不做计算的。因此在上面的调用中，count 将只在模拟框架调用存储于工作条目中的动作时才完成自增值。注意到 afterDelay 是柯里化的方法。因此它还是 9.5 节曾经提到的“柯里化可以被用来使得方法调用看上去更像内建语法结构”原则的绝好脚注。

创建了的工作条目还需要插入到排程表中。这需要 insert 方法，它还可以让排程表始终保持有序：

```
private def insert(ag: List[WorkItem],
    item: WorkItem): List[WorkItem] = {
  if (ag.isEmpty || item.time < ag.head.time) item :: ag
  else ag.head :: insert(ag.tail, item)
}
```

Simulation 类的核心是 run 方法的定义：

```
def run() {
  afterDelay(0) {
    println("*** simulation started, time = "+
        currentTime +" ***")
  }
  while (!agenda.isEmpty) next()
}
```

该方法重复提取排程表的第一个条目，从中移除并执行，直到表中没有需要执行的条目为止。每一次的执行都须要调用 next 方法，定义如下：

```
private def next() {
  (agenda: @unchecked) match {
    case item :: rest =>
      agenda = rest
      curtime = item.time
```

```
        item.action()
    }
  }
```

next 方法把当前排程表解构为首部的条目 item，以及剩余工作条目列表 rest 的模式匹配。它从排程表移除条目，然后把模拟时间 curtime 设置为工作条目的时间，然后执行工作条目的动作。

370

请注意 next 只能在日程表非空的情况下调用，代码中没有处理空列表的情况，因此如果尝试对空日程表执行 next 方法将导致 MatchError 异常。

实际上，Scala 编译器通常会警告你，说你少加了一条列表可能的模式：

```
Simulator.scala:19: warning: match is not exhaustive!
missing combination            Nil

  agenda match {
  ^
one warning found
```

在这个例子里，遗漏的情况并不是问题，因为我们知道 next 只会对非空日程表执行调用。因此，你或许想关闭这个警告。之前在 15.5 节我们看到过可以通过给模式匹配的选择器表达式添加 @unchecked 注解做到这点。这也正是 Simulation 代码使用“(agenda: @unchecked) match”而非“agenda match”的原因。

好了。以上这些就是模拟框架的“稍嫌”短小的代码。你或许会感到惊讶：如果它所能做的只是执行工作条目的列表，那么这个框架真的能够支持那些有趣的模拟实验吗？实际上模拟框架的力量来自于这样一个事实：存储于工作条目中的动作自身可以在执行时把更多的工作条目安排到排程表中。这使得从简单的开始发展为长时间运行的模拟变为了可能。

18.6 电路模拟

下一步是使用模拟框架为 18.4 节提到过的电路实现域特定语言。回忆一下之前说过的，电路 DSL 由线路及创建与门、或门和反转器的方法类组成。这些都包含在扩展了模拟框架的 BasicCircuitSimulation 类中。这个类的代码参见清单 18.9 和清单 18.10。

371

BasicCircuitSimulation 类声明了三个抽象方法，代表三种基本门电路的延迟：InverterDelay，AndGateDelay 和 OrGateDelay。实际延迟在这个类的级别中还未知，因为它们依赖于被模拟的电路的技术参数。这也是类 BasicCircuitSimulation 把这些延迟定义为抽象函数的原因，因此它们的具体定义将推迟到子类中完成[7]。BasicCircuitSimulation 类的其他成员的实现在之后介绍。

```
package org.stairwaybook.simulation

abstract class BasicCircuitSimulation extends Simulation {
  def InverterDelay: Int
  def AndGateDelay: Int
```

[7]注：这些“延迟”方法的名称开始于大写字母，因为它们代表的是常量。又因为它们是方法，所以可以在子类中重载。你将在 20.3 节发现如何用 val 实现同样的功能。

```
def OrGateDelay: Int

class Wire {

  private var sigVal = false
  private var actions: List[Action] = List()

  def getSignal = sigVal

  def setSignal(s: Boolean) =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_ ())
    }

  def addAction(a: Action) = {
    actions = a :: actions
    a()
  }
}

def inverter(input: Wire, output: Wire) = {
  def invertAction() {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) {
      output setSignal !inputSig
    }
  }
  input addAction invertAction
}

// 在清单 18.10 中继续
```

清单 18.9 BasicCircuitSimulation 类的前半部分

```
// 接自清单 18.9
def andGate(a1: wire, a2: wire, output; wire) = {
  def andAction() = {
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig & a2Sig)
    }
  }
  a1 addAction andAction
  a2 addAction andAction
}

def orGate(o1: Wire, o2: Wire, output: Wire) {
  def orAction() {
    val o1Sig = o1.getSignal
    val o2Sig = o2.getSignal
    afterDelay(OrGateDelay) {
      output setSignal (o1Sig | o2Sig)
    }
  }
  o1 addAction orAction
  o2 addAction orAction
}
```

```
    def probe(name: String, wire: Wire) {
      def probeAction() {
        println(name +" "+ currentTime +
             " new-value = "+ wire.getSignal)
      }
      wire addAction probeAction
    }
  }
```

373

清单 18.10 BasicCircuitSimulation 类的后半部分

Wire 类

线路需要支持以下三种基本动作：

`getSignal: Boolean`	返回当前加载在线路上的信号。
`setSignal(sig: Boolean)`	设置线路信号为 `sig`。
`addAction(p: Action)`	把特定过程 `p` 附加到线路的动作中。思路是所有附加在某根线路上的动作过程将在每次线路信号改变时被执行。通过连接组件可以为线路添加该组件特有的功能。附加的动作将在被添加到线路，以及之后每次线路信号改变时被执行。

以下是 `Wire` 类的实现：

```
class Wire {
  private var sigVal = false
  private var actions: List[Action] = List()

  def getSignal = sigVal

  def setSignal(s: Boolean) =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_ ())
    }

  def addAction(a: Action) = {
    actions = a :: actions
    a()
  }
}
```

374

两个私有变量组合成线路的状态。变量 `sigVal` 代表当前信号，变量 `actions` 代表当前附加在线路上的动作过程。唯一值得关注的方法实现是 `setSignal`：当线路信号改变时，新的信号值首先被保存在变量 `sigVal` 中，然后再执行所有线路附加的动作。请注意缩写格式："`actions foreach(_())`"，对于每个 `actions` 列表的元素执行函数调用"`_ ()`"。8.5 节曾经描述过，函数"`_()`"是"`f => f()`"的缩写，也就是说，它可以带函数（我们称之为 `f`）作参数，并执行对它的空参数列表调用。

inverter 方法

创建反转器的唯一效果是在输入线路上安装了一个动作。该动作在安装及之后每次输入线路信号改

变时被调用。其效果是（通过 setSignal 方法）把反转器的输出值设置为输入值的反转。由于反转器具有延迟，因此仅当输入值发生改变，且动作执行 InverterDelay 单位模拟时间后，才会发生。这也意味着如下的实现：

```
def inverter(input: Wire, output: Wire) = {
  def invertAction() {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) {
      output setSignal !inputSig
    }
  }
  input addAction invertAction
}
```

inverter 方法的效果是向 input 线路添加了 invertAction。如果调用该动作，将获得输入信号并把另一个反转 output 信号的动作装入模拟日程表中。这个另外的动作将在模拟时间的 InverterDelay 单位时间之后执行。请注意方法是如何使用模拟框架的 afterDelay 方法以创建将在之后执行的新工作条目的。

375

andGate 和 orGate 方法

与门的实现类似于反转器的实现。与门的目的是输出所有输入信号的合取值（即，逻辑与）。这件事应该发生在它的两个输入之一改变的 AndGateDelay 单位模拟时间之后。因此有以下实现：

```
def andGate(a1: Wire, a2: Wire, output: Wire) = {
  def andAction() = {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig & a2Sig)
    }
  }
  a1 addAction andAction
  a2 addAction andAction
}
```

andGate 方法的效果是把 andAction 加到它的两根输入线路 a1 和 a2 上。调用动作时，获得两个输入信号并安装另一个设置 output 信号为两个输入信号的交的动作。这个另外的动作将在 AndGateDelay 单位模拟时间之后执行。注意只要有输入线路的信号改变输出就需要重新计算。这也是两根输入线路 a1 和 a2 都需要安装同样的 andAction 原因。orGate 方法实现近似，只是用逻辑或替代了逻辑与。

模拟输出

运行模拟器，你需要观察线路上信号改变的方法。为了达到这个目的，你可以模拟在线路上插探针（probe）的动作：

```
def probe(name: String, wire: Wire) {
  def probeAction() {
      println(name +" "+ currentTime +
           " new-value = "+ wire.getSignal)
    }
  wire addAction probeAction
}
```

376

probe 过程的效果是对指定线路安装 probeAction。一如往常，安装的动作会在每次线路信号改变时执行。这里它仅打印输出线路名称（probe 的第一个入参），以及当前模拟时间和线路的信号值。

运行模拟器

经过以上这一系列的准备活动之后，亲眼见证模拟器实际操作的时刻终于到了。要定义具体的模拟活动，你首先须要继承并扩展模拟框架类。为了看点儿有趣的效果，我们将创建扩展 BasicCircuitSimulation 的抽象模拟类并包含本章早些时候在清单 18.6 和清单 18.7 中呈现的半加法器和全加法器的方法定义。这个类，我们称为 CircuitSimulation，的代码在清单 18.11 中：

```
package org.stairwaybook.simulation

abstract class CircuitSimulation
  extends BasicCircuitSimulation {

  def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {
    val d, e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)
  }

  def fullAdder(a: Wire, b: Wire, cin: Wire,
      sum: Wire, cout: Wire) {

    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
}
```

377

清单 18.11 CircuitSimulation 类

具体的电路模拟实现是继承自 CircuitSimulation 类的对象。对象仍然需要解决基于被模拟电路实现技术的门电路延迟问题。最后，你还要定义被模拟的具体电路。你可以在 Scala 解释器中交互完成这些步骤：

```
scala> import org.stairwaybook.simulation._
import org.stairwaybook.simulation._
```

首先是门电路的延迟。定义能够提供这些数字的对象，MySimulation：

```
scala> object MySimulation extends CircuitSimulation {
         def InverterDelay = 1
         def AndGateDelay = 3
```

```
        def OrGateDelay = 5
    }
defined module MySimulation
```

因为你正打算访问重复 MySimulation 对象的成员，所以可以加一个对象引用以缩短后续的代码：

```
scala> import MySimulation._
import MySimulation._
```

接下来是电路。定义四根线路，并把探针放在其中的两根上：

```
scala> val input1, input2, sum, carry = new Wire
input1: MySimulation.Wire =
    simulator.BasicCircuitSimulation$Wire@111089b
input2: MySimulation.Wire =
    simulator.BasicCircuitSimulation$Wire@14c352e
sum: MySimulation.Wire =
    simulator.BasicCircuitSimulation$Wire@37a04c
carry: MySimulation.Wire =
    simulator.BasicCircuitSimulation$Wire@1fd10fa

scala> probe("sum", sum)
sum 0 newvalue = false

scala> probe("carry", carry)
carry 0 newvalue = false
```

378

注意探针会立刻打印输出结果。这是由于每个安装到线路上的动作都会在安装的第一时间被执行。

现在定义连接在线路上的半加法器：

```
scala> halfAdder(input1, input2, sum, carry)
```

最后，逐次设置两根输入线路的信号为 true，并执行模拟过程：

```
scala> input1 setSignal true

scala> run()
*** simulation started, time = 0 ***
sum 8 new-value = true

scala> input2 setSignal true

scala> run()
*** simulation started, time = 8 ***
carry 11 new-value = true
sum 15 new-value = false
```

18.7 小结

本章推出了两个一开始似乎完全不相干的技术：可变的状态及高阶函数。可变状态被用来模拟状态能够随时间改变的物理实体。高阶函数用在模拟框架中，在特定的模拟时间点执行一系列动作。它们共同被应用于电路模拟中，作为连结动作和状态改变的触发器（trigger）。随后，你看到了一种简单的方法可以把域特定语言定义为库。一章包含了这么多东西也真是够多的了！

如果你觉得还有兴趣的话，或许可以尝试更多模拟的例子。你可以把半加法器和全加法器组合成更大的电路，或者使用截止到目前已经定义了的基本的门电路设计新的电路并模拟它们。下一章，你将学习 Scala 中关于类型参数化的知识，并看到其他的函数式和指令式方法组合在一起所产生的好的解决方案。

379

第 19 章

类型参数化

本章，我们将解释 Scala 中类型参数化的细节。之后我们将通过实例：纯函数式队列类（Queue）的设计，演示第 13 章介绍过的一些信息隐藏技术。我们之所以要把类型参数化与信息隐藏摆在一起，是因为信息隐藏可以用来获得更为通用的类型参数化的变化型注解。

类型参数化让你能够编写泛型类和特质。例如，set 是泛型的，带类型参数：定义为 `Set[T]`。所以，任何特定的 set 实例都应该写成 `Set[String]`，或者 `Set[Int]`这样，必须是某种东西的 set。不像 Java，允许原始类型，Scala 需要你指定类型参数。而且变化型定义了参数化类型的继承关系，例如，`Set[String]`是 `Set[AnyRef]`的子类型。

本章包含三个部分。第一部分开发了纯函数式队列的数据结构。第二部分开发了该结构内部表现细节的技术。最后的部分解释了类型参数的变化型及它是如何与信息隐藏技术互相交互的。

19.1 queues 函数式队列

函数式队列是一种具有以下三种操作方式的数据结构：

`head` 返回队列的第一个元素。

`tail` 返回除第一个元素之外的队列。

`append` 返回尾部添加了指定元素的新队列。

不像可变队列，函数式队列在添加元素的时候不会改变其内容，而是返回包含了这个元素的新队列。本章的目的是创建一个名为 `Queue` 的类，其工作方式如下：

```
scala> val q = Queue(1, 2, 3)
q: Queue[Int] = Queue(1, 2, 3)

scala> val q1 = q append 4
q1: Queue[Int] = Queue(1, 2, 3, 4)

scala> q
res0: Queue[Int] = Queue(1, 2, 3)
```

如果 `Queue` 实现为可变类型，那么上面的第二行输入的 `append` 操作将改变 `q` 的内容；从而在操作之后使得结果 `q1`，及原始队列 `q`，都包含了序列 `1`，`2`，`3`，`4`。但对于函数式队列来说，添加的值仅出现在结果 `q1` 中，不会出现在被操作的队列 `q` 中。

纯函数式队列同样与 List 具有某些相似性。它们都被称为是完全持久的数据结构，即使在扩展或改变之后，旧版本的对象仍然可用。它们都支持 head 和 tail 操作。但 List 通常使用::操作在对象前端执行扩展，而队列使用 append 扩展在对象后段。

怎样的实现才更为高效呢？理想的情况下，函数式（不可变）队列不应该具有比指令式（可变）队列更高的基本开销。也就是说，三种操作，head，tail，以及 append 应该能在常量时间内完成。

实现函数式队列的一种简单方案是以 List 作为功能的表达类型。这样，head 和 tail 将转为 List 的相同操作，而 append 将变为连结操作。这将得到以下的实现代码：

```
class SlowAppendQueue[T](elems: List[T]) { // 效率不高
  def head = elems.head
  def tail = new SlowAppendQueue(elems.tail)
  def append(x: T) = new SlowAppendQueue(elems ::: List(x))
}
```

382

这种实现的问题在于 append 操作。它花的时间与存储于队列的元素数量成正比。如果你想要的是常量时间的 append，你也可以尝试把表达 List 里面的元素倒序排列，这样原本最后添加进来的元素现在出现在 List 的最前面。这样就有了以下的实现：

```
class SlowHeadQueue[T](smele: List[T]) { //效率不高
  //smele 是把 elems 反过来
  def head = smele.last
  def tail = new SlowHeadQueue(smele.init)
  def append(x: T) = new SlowHeadQueue(x :: smele)
}
```

现在 append 是常量时间了，但 head 和 tail 不是。它们现在所花费的时间与存储在队列中的元素数量成正比。

以上两个例子说明，想让这三种操作都保持常量时间完成并不是容易的事。甚至令人怀疑是否有这种可能！然而，通过两种操作的组合可以得到很接近这个目标的方案。其思想是用两个 List，我们称其为 leading 和 trailing，来表达队列。leading 包含了前段元素，而 trailing 包含了反向排列的后段元素。队列在任意时刻的所有内容都可以表达为“leading ::: trailing.reverse”。

现在，想要添加新的元素，你只要使用::操作符把它 cons 到 trailing，因此 append 是常量时间。这也意味着，当原始的空队列通过后继的 append 操作构建起来时，trailing 将不断增加而 leading 将始终空白。于是，在对空的 leading 第一次执行 head 或 tail 操作之前，trailing 应该被反转并复制给 leading。我们称这个操作为 mirror。清单 19.1 列出了以这种方式完成的队列实现。

383

这种队列实现的复杂度如何计算？mirror 操作花费的时间大概与队列的元素数量成正比，但仅在 leading 为空时。如果 leading 非空，它将直接返回。因为 head 和 tail 调用了 mirror，所以它们的复杂度与队列长度也呈线性关系。然而队列越长，mirror 被调用的次数就越以级数的方式递减。实际上，假设队列的长度为 n 且 leading 为空，那么 mirror 操作须反转并复制长度为 n 的 List。然而，下一次需要 mirror 工作时是在 leading 再次变空时，将在 n 次 tail 操作之后，也就意味着这 n 次 tail 操作平均每次“承担”mirror 操作时间复杂度的 n 分之一，这也就相当于常量的工作时间。假设 head，tail 及 append 操作出现的频率大致相等，那么经过缓冲的复杂度对于每次操作来

说就都是常量时间。所以函数式队列的效率将逐渐趋近于可变队列的。

```
class Queue[T](
  private val leading: List[T],
  private val trailing: List[T]
) {
  private def mirror =
    if (leading.isEmpty)
      new Queue(trailing.reverse, Nil)
    else
      this

  def head = mirror.leading.head

  def tail = {
    val q = mirror
    new Queue(q.leading.tail, q.trailing)
  }

  def append(x: T) =
    new Queue(leading, x :: trailing)
}
```

清单 19.1 基本的函数式队列

现在，要对以上讨论加上一些声明。首先，讨论的情况只是关于渐进的趋势，这和常量因子会有些不同。第二，讨论是基于 head，tail 及 append 的调用处于接近的频度这一事实。如果 head 调用次数远多于另外的两种操作，上面的讨论就无效了，因为每次调用 head 都有可能以调用 mirror 对列表做重新整理为代价。第二项声明可被避免；我们可以设计出这样的函数式队列，使得对于一系列的连续 head 操作来说，仅第一次须要做重新组织。你将在本章结尾发现具体实现方法。

19.2 信息隐藏

清单 19.1 的 Queue 实现目前已经非常好地考虑到了效率问题。但你仍然会质疑，这种效率是以暴露了本不该暴露的细节实现为代价的。全局可访问的 Queue 构造器，带了两个列表做参数，而且竟然还有一个是反序的——这简直就不能当作是可以直观地表达队列的形式。这里需要有一种能够对本节的客户代码隐藏这种构造器的方式，以下我们将说明 Scala 中可以用的几种办法。

私有构造器及工厂方法

Java 中，你可以把构造器声明为私有的使其不可见。Scala 中，主构造器无须明确定义；不过虽然它的定义隐含于类参数及类方法体中，还是可以通过把 private 修饰符添加在类参数列表的前边把主构造器隐藏起来，如清单 19.2 所示：

```
class Queue[T] private (
  private val leading: List[T],
  private val trailing: List[T]
)
```

清单 19.2 通过私有化隐藏主构造器

夹在类名与其参数之间的 private 修饰符表明 Queue 的构造器是私有的：它只能被类本身及伴生对象访问。类名 Queue 仍然是公开的，因此你可以继续使用这个类，但不能调用它的构造器：

```
scala> new Queue(List(1, 2), List(3))
<console> :6: error: constructor Queue cannot be accessed in object $iw[1]
       new Queue(List(1, 2), List(3))
       ^
```

现在客户代码将不再能够调用 Queue 类的主构造器，需要有创建新队列的其他方法。一种可能方案是添加辅助构造器，如下：

385

```
def this() = this(Nil, Nil)
```

上面看到的辅助构造器可以构建空队列。通过改良，它可以带上初始队列元素列表：

```
def this(elems: T*) = this(elems.toList, Nil)
```

其中的 T*是重复参数的注记，之前已经介绍过，具体内容参见 8.8 节。

另一种可能性是添加可以用初始元素序列创建队列的工厂方法。比较简洁的做法是定义与类同名的 Queue 对象及 apply 方法，如清单 19.3 所示：

```
object Queue {
  //用初始元素'xs'构造队列
  def apply[T](xs: T*) = new Queue[T](xs.toList, Nil)
}
```

清单 19.3 伴生对象的 apply 工厂方法

再把这个对象放在类 Queue 的同一个源文件中，你就把它变成了类的伴生对象。13.4 节已经看过，伴生对象与类具有相同的访问权。据此，即使 Queue 类的构造器是私有的，对象 Queue 的 apply 方法也可以创建新的 Queue 对象。

注意，工厂方法名为 apply，因此客户可以用类似于 Queue(1, 2, 3)这样的表达式创建队列。由于 Queue 是对象而不是函数，这个表达式会被扩展为 Queue.apply(1, 2, 3)。结果，对于客户来说，Queue 就好象是全局定义的工厂方法。实际上，Scala 没有全局可见的方法；每个方法都必须被包含在对象或类中。然而，使用定义在全局对象中的名为 apply 的方法，你就能够提供这种看上去好像是对全局方法调用的使用模式了。

可选方案：私有类

386

私有构造器和私有成员是隐藏类的初始化代码和表达代码的一种方式。另一种更为彻底的方式是直接把类本身隐藏掉，仅提供能够暴露类公共接口的特质。

```
trait Queue[T] {
  def head: T
  def tail: Queue[T]
  def append(x: T): Queue[T]
```

[1]译注：Queue 构造器不能在对象$iw 中被访问。

```
}

object Queue {
  def apply[T](xs: T*): Queue[T] =
    new QueueImpl[T](xs.toList, Nil)

  private class QueueImpl[T](
    private val leading: List[T],
    private val trailing: List[T]
  ) extends Queue[T] {

    def mirror =
      if (leading.isEmpty)
        new QueueImpl(trailing.reverse, Nil)
      else
        this

    def head: T = mirror.leading.head

    def tail: QueueImpl[T] = {
      val q = mirror
      new QueueImpl(q.leading.tail, q.trailing)
    }

    def append(x: T) =
      new QueueImpl(leading, x :: trailing)
  }
}
```

清单 19.4 函数式队列的类型抽象

清单 19.4 实现了这种设计方案。代码中定义了特质 Queue，声明了方法 head、tail 和 append。这三个方法都实现在子类 QueueImpl 中，而它本身是对象 Queue 的内部类。这个方案暴露给客户的信息与前面相同，但使用了不同的技术。代之以逐个隐藏构造器与方法，这个版本隐藏全体实现类。

19.3 变化型注解

定义在代码 19.4 中的 Queue 是特质，不是类型。因为它带类型参数。结果，你将不能创建类型为 Queue 的变量：

```
scala> def doesNotCompile(q: Queue) {}
<console> :5: error: trait Queue takes type parameters
       def doesNotCompile(q: Queue) {}
                          ^
```

取而代之，特质 Queue 能够指定参数化的类型，如 Queue[String]，Queue[Int]，或 Queue[AnyRef]：

```
scala> def doesCompile(q: Queue[AnyRef]) {}
doesCompile: (Queue[AnyRef])Unit
```

也就是说，Queue 是特质，Queue[String]是类型。Queue 也被称为类型构造器，因为有了它你就能通过指定参数类型构造新的类型。（这是对通过指定值参数使用单纯传统的（plain-old）构造器创建对象实例的模拟。）类型构造器“产生”了一个类型的家族，包括 Queue[Int]，Queue[String]，

以及 Queue[AnyRef]。

你也可以认为 Queue 是泛型的特质。（带类型参数的类和特质是“泛型的”，但它们产生的类型已被“参数化”，不再是泛型的。）术语“泛型”指的是通过一个能够广泛适用的类或特质定义了许多特定的类型。例如，清单 19.4 里的特质 Queue 定义了泛型的队列。Queue[Int]及 Queue[String]等，则是特定的队列。

类型参数的组合与子类型化产生了一些有趣的问题。例如，在 Queue[T]产生的类型家族的成员之间是否有任何特定的子类型关系？更具体地说就是，是否可以把 Queue[String]当作是 Queue[AnyRef]的子类型？

388

或从更广泛的意义上来说，如果 S 是类型 T 的子类型，那么是否可以把 Queue[S]当作 Queue[T]的子类型？如果是，你可以认为 Queue 特质是与它的类型参数 T 保持协变（covariant）的（或“有弹性的”）。或者，由于它只有一个类型参数，你可以简单地说 Queue 是协变的。协变的 Queue 将意味着，你可以把，比方说 Queue[String]，传递给之前看到过的值参数类型为 Queue[AnyRef]的 doesCompile 方法。

直观上这些看上去都很合理，因为 String 队列似乎就是 AnyRef 的特例。然而在 Scala 中，泛型类型缺省的是非协变的（nonvariant）（或称为“严谨的”）子类型化。也就是说，根据清单 19.4 中定义的 Queue，不同元素类型的队列之间没有子类型关系。Queue[String]对象不能被用作 Queue[AnyRef]。然而，可以用如下方式改变 Queue 类定义的第一行，以要求队列协变（弹性）的子类型化：

```
trait Queue[+T] { ... }
```

在正常的类型参数前加上+号表明这个参数的子类型化是协变（弹性）的。这个符号是向 Scala 说明你希望可以把，比方说 Queue[String]，当作 Queue[AnyRef]的子类型。编译器将检查 Queue 被这种子类型化所说的方式定义。

除了+号以外，还可以前缀加上–号，这表明是需要逆变的（contravariant）子类型化。如果 Queue 定义如下：

```
trait Queue[-T] { ... }
```

那么如果 T 是类型 S 的子类型，这将隐含 Queue[S]是 Queue[T]的子类型（这对于队列的这个例子来说是多么奇怪呀！）。无论类型参数是协变的，逆变的，还是非协变的，都被称为参数的变化型。可以放在类型参数前的+号和–号被称为变化型注解。

在纯函数式中，许多类型都是自然协变（弹性）的。然而，一旦引入了可变数据，情况就改变了。不明白的话，可以假想有这么一种简单的只能读写一个元素的单元格（Cell）类型，代码见清单 19.5。

```
class Cell[T](init: T) {
  private[this] var current = init
  def get = current
  def set(x: T) { current = x }
}
```

清单 19.5 非协变（严谨）的 Cell 类

代码中的 Cell 类型被声明为非协变（严谨）的。为了便于讨论，假设暂时声明为协变的——也就是说，声明为 Cell[+T]——并发送给 Scala 编译器。（实际没有，我们会在之后解释）。于是你将能构建如下存在问题的语句序列：

389

```
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get
```

这四条语句从自身来看，每条都很正常。第一行创建 String 单元格并保存在 c1 中。第二行定义 c2，类型是 Cell[Any]，并初始化为 c1。这没问题，因为前提设定 Cell 为协变的。第三行把 c2 的值设为 1。这也没问题，因为被赋予的值 1 是 c2 的元素类型 Any 的实例。最后，第四行把 c1 的元素值赋给字符串。这没什么奇怪的，两边的值都是同样的类型。但把它们放在一起，这四行代码要完成的是把整数 1 赋给字符串 s。这明显是对类型声明的破坏。

哪个操作应该对运行期的失败负责呢？一定是使用了协变子类型化的第二行。其他的语句都太简单和基础了。因此，String 类型的 Cell 并不就是 Any 类型的 Cell，因为有些事可以对 Any 的 Cell 做，但不能对 String 的 Cell 做。比如，不能以 Int 类型的入参调用 String 的 Cell 的 set 方法。

实际上，如果你把协变版的 Cell 类传递给 Scala 编译器，将返回编译时错误：

390

```
Cell.scala:7:error: covariant type T occurs in
contravariant position in type T[2] of value x
  def set(x: T) = current = x
```

变化型和数组

（如果拿单元格的这种行为）与 Java 数组比较会很有趣。基本上，除了数组可以有超过一个元素之外，它与单元格没什么差别。然而，Java 中数组被认为是协变的。例如下面是用 Java 数组模拟了上面的单元格交互操作：

```
// 在 Java 里
String[] a1 = { "abc" };
Object[] a2 = a1;
a2[0] = new Integer(17);
String s = a1[0];
```

试过这个例子，你会发现它能够通过编译，但执行程序会在把 Integer 对象赋值给 a2[0]时引发 ArrayStore 异常：

```
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
        at JavaArrays.main(JavaArrays.java:8)
```

这里的情况是 Java 在运行时保存了数组的元素类型。于是每次数组元素更新的时候，新元素值都会用储存的类型校验合法性。如果不是该类型的实例，就会抛出 ArrayStore 异常。

[2]译注：协变类型 T 出现在值 x 的类型 T 的逆变位置。

你或许会问为什么 Java 采纳了这样的设计，似乎既不安全又不经济。被问及这个问题的时候，James Gosling，Java 语言的主要发明者，回答说他们希望有一个通用处理数组的简单方法。例如，他们想要能够编写方法排序数组的所有元素，使用以下的方法签名带入对象数组：

```
void sort(Object[] a, Comparator cmp) { ... }
```

数组的协变被用来确保任意参考类型的数组都可以传入排序方法。当然，随着 Java 引入了泛型，这种排序方法现在可以带有类型参数，因此数组的协变不再有用。只是考虑到兼容的原因，直到今天它都存在于 Java 中。

391

Scala 不认为数组是协变的，以尝试保持比 Java 更高的纯粹性。下面的代码演示了如果你把数组例子中的前两行翻译成 Scala 时的情况：

```
scala> val a1 = Array("abc")
a1: Array[java.lang.String] = Array(abc)

scala> val a2: Array[Any] = a1
<console>:5: error: type mismatch;
 found   : Array[java.lang.String]
 required: Array[Any]
       val a2: Array[Any] = a1
                            ^
```

这里的情况是 Scala 把数组当作是非协变（严谨）的，因此 `Array[String]`对象不能被当作与 `Array[Any]`一致。然而，有时需要使用对象数组作为模拟泛型数组的手段与 Java 的遗留方法执行交互。例如，你或许希望调用类似于之前描述的，以字符串数组为参数的排序方法。为了满足这种情况，Scala 允许你把 T 类型的数组造型为任意 T 的超类型的数组：

```
scala> val a2: Array[Object] =
         a1.asInstanceOf[Array[Object]]
a2: Array[java.lang.Object] = Array(abc)
```

编译时的造型始终合法，并且将在运行时永远成功，因为 JVM 的内含运行时模型与 Java 语言一致，把数组当作是协变的。但是之后你或许还是会收到 `ArrayStore` 异常，就如你在 Java 中遇到的一样。

19.4 检查变化型注解

现在你已经看到了一些不适用变化型的例子，你或许想知道哪种类型的类定义是必须拒绝的，哪种是可以接受的。到目前为止，所有类型可靠性的冲突都与一些可重新赋值的字段或数组元素有关。相反，队列的纯函数式实现似乎看上去好像是协变很好的候选者。然而，下面的例子可以说明即使没有可重新赋值的字段，你仍然可以“设计”出不健全的情况。

392

为了举例说明，我们假设清单 19.4 定义的队列是协变的。然后，创建指定元素类型为 `Int` 的队列子类，并重载 `append` 方法：

```
class StrangeIntQueue extends Queue[Int] {
  override def append(x: Int) = {
    println(Math.sqrt(x))
```

```
    super.append(x)
  }
}
```

StrangeIntQueue 的 append 方法在执行本身的添加操作之前，首先打印输出它的（整数）参数的方根。现在，你可以使用以下两行编写反例：

```
val x: Queue[Any] = new StrangeIntQueue
x.append("abc")
```

第一行是有效的，因为 StrangeIntQueue 是 Queue[Int]的子类，并且假设队列是协变的，Queue[Int]是 Queue[Any]的子类型。第二行是有效的因为你可以对 Queue[Any]添加 String 对象。然而，这两行放在一起的效果就是取字符串的平方根，毫无意义。

很明显，不是只有可变字段会使协变类型出错。这种问题要更为普遍。只要泛型的参数类型被当作方法参数的类型，那么包含它的类或特质就有可能不能与这个类型参数一起协变。对于队列来说，append 方法违反了以下情况：

```
class Queue[+T] {
  def append(x: T) =
    ...
}
```

使用 Scala 编译器跑一遍上面修改后的队列类将产生：

```
Queues.scala:11: error: covariant type T occurs in
contravariant position in type T of value x
  def append(x: T) =
             ^
```

393

可重新赋值的字段是“不允许使用+号注解的类型参数用做方法参数类型”这条规则的具体例子。我们曾在 18.2 节提到过，可重新赋值的字段，“var x: T”在 Scala 里被看作一种 getter 方法，“def x: T”，和 setter 方法，“def x_= (y: T)”。正如你所见的，setter 方法带有字段类型为 T 的参数。因此类型将不是协变的。

> **快速通道**
>
> 本节余下的部分，我们将描述 Scala 编译器检查变化型注解的机制。如果你当前对这种细节不感兴趣，你可以安全地跳到 19.5 节。所需了解的最重要的事情就是 Scala 编译器将检查你放在类型参数里的每个变化型注解。例如，如果你尝试声明类型参数为协变的（通过添加+号），但它或许会引起潜在的运行时故障，那么你的程序就不会被编译通过。

为了核实变化型注解的正确性，Scala 编译器会把类或特质结构体的所有位置分类为正，负，或中立。所谓的“位置”是指类（或特质，但从此开始为我们只用“类”代表）的结构体内可能会用到类型参数的地方。例如，任何方法的值参数都是这种位置，因为方法值参数具有类型，所以类型参数可以出现在这个位置上。编译器检查类的类型参数的每一个用法。注解了+号的类型参数只能被用在正的位置上，而注解了–号的类型参数只能用在负的位置上。没有变化型注解的类型参数可以用于任何位置，因此它是唯一能被用在类结构体的中性位置上的类型参数。

为了对这些位置分类，编译器首先从类型参数的声明开始，然后进入更深的内嵌层。处于声明类的最顶层被划为正的位置。默认情况下，更深的内嵌层的位置的分类会与它的外层一致，不过仍有屈指可

数的几种例外会改变具体的分类。方法值参数位置是方法外部的位置的翻转类别，这里正的位置翻转为负的，负的位置翻转为正的，而中性位置仍然保持中性。

394

除了方法值参数位置外，方法的类型参数的当前类别也会同时被翻转。而类型的类型参数位置，如C[Arg]中的Arg，也有可能被翻转，这取决于对应类型参数的变化型。如果C的类型参数标注了+号，那么类别保持不变。如果C的类型参数标注了–号，那么当前类别被翻转。如果C的类型参数没有变化型注解，那么当前类别将改为中性。

下面是个显得有点儿生编硬造的例子，我们考虑如下的类型定义，其中若干位置的变化型被标注了⁺（正的）或⁻（负的）：

```
abstract class Cat[-T, +U] {
  def meow[W⁻](volume: T⁻, listener: Cat[U⁺, T⁻]⁻)
    : Cat[Cat[U⁺, T⁻]⁻, U⁺]⁺
}
```

类型参数W，以及两个值参数，volume和listener的位置都是负的。注意meow的结果类型，第一个Cat[U, T]参数的位置是负的，因为Cat的第一个类型参数T被标注了–号。这个参数中的类型U重新转为正的位置（两次翻转），而参数中的类型T仍然是负的位置。

从讨论中你可以发现想要跟上变化型位置的变化是非常困难的。因此Scala编译器自动帮你完成这项工作，从而减轻了你的负担。

计算变化型的过程中，编译器检查每个类型参数是否仅用于分类正确的位置上。这个例子中，T仅用在负的位置上，而U仅用于正的位置上。因此Cat类是类型正确的。

19.5 下界

回到Queue类中来。你已经看到了前面演示在清单19.4的Queue[T]定义中不能实现对T的协变，因为T作为参数类型出现在append方法中，而这里是负的位置。

395

很幸运，有一个办法可以打开这个结：可以通过把append变为多态以使其泛型化（即提供给append方法类型参数）并使用它的类型参数的下界。清单19.6演示了实现这个想法的Queue的新格式。

```
class Queue[+T] (private val leading: List[T],
    private val trailing: List[T] ) {
  def append[U>: T](x: U) =
    new Queue[U](leading, x :: trailing) // ...
}
```

清单19.6 带有下界的类型参数

新的定义指定了append的类型参数U，并通过语法“U >: T”，定义了T为U的下界。结果，U必须是T的超类型[3]。append的参数现在变为类型U而不是类型T，而方法的返回值现在也变为Queue[U]，取代了Queue[T]。

[3]注：超类型与子类型关系是自反的，也就是说类型对于自身来讲，既是超类型也是子类型。即使T是U的下界，你也仍然能够把T传递给append方法。

至于例子，可以假设存在类 Fruit 及两个子类，Apple 和 Orange。通过使用 Queue 类的新定义，现在可以把 Orange 对象加入到 Queue[Apple]。而返回的结果为 Queue[Fruit]类型。

append 的改进定义是类型正确的。直观地看，如果 T 比预期的类型更为特化（例如，用 Apple 替代 Fruit），那么 append 的调用就仍然正确，因为 U（Fruit）是 T（Apple）的超类型[4]。

可以认为 append 的新定义比旧的要更好，因为它更为普通。不像旧的版本，新定义允许你把队列元素类型 T 的任意超类型 U 的对象添加进来，结果为 Queue[U]。它与队列的协变性一起，以自然的方式提供了对不同元素类型队列建模的合适的弹性。

这也说明变化型注解与下界可以相互协作。这是很好的类型驱动设计的例子，由接口的类型导引其细节的设计和实现。在队列的例子中，你或许没有想到要使用下界重构 append 方法的实现，但你或许想要让队列是协变的。这种情况下，编译器将指出 append 变化型的错误。而通过添加下界来修正变化型错误将使得 append 更为通用并且也连带使得队列作为整体功能更为有用。

396

我们观察到的这种情况，其实也是 Scala 更推崇声明点的变化型，而非 Java 的通配符中可以发现的使用点的变化型的主要原因。如果是使用点的变化型，那你是在独立设计类。需要类的客户把正确的值带入通配符位置，并且如果他们做错了，则某些重要的实例方法将不再可用。变化型变成了棘手的工作，用户常常会搞错，他们会想要逃避，认为通配符和泛型过于复杂。而对于定义过程中的变化型，你是在向编译器解释你的意图，编译器会复查你是否的确实现了意图。

19.6 逆变

本章到目前为止，所有的例子要么是协变的，要么是非协变的。但还有一些情况，逆变才显得自然。例如，看一下清单 19.7 中的输出通道特质：

```
trait OutputChannel[-T] {
  def write(x: T)
}
```

清单 19.7 逆变的输出通道

这里，OutputChannel 被定义为 T 的逆变。因此，AnyRef 的输出通道，打个比方，将是 String 的输出通道的子类型。尽管这看起来不是很直观，但确实很有道理。如果想要明白这是为什么，可以考虑一下你能对 OutputChannel[String]做什么。唯一支持的操作就是写一个 String 给它。而同样的操作对于 OutputChannel[AnyRef]来说也支持。因此用 OutputChannel[AnyRef]替代 OutputChannel[String]是安全的。相反，在需要 OutputChannel[AnyRef]的地方替换成 OutputChannel[String] 却是不安全的。毕竟，你可以把任何对象发送给 OutputChannel[AnyRef]，而 OutputChannel[String]要求写入的值只能是字符串。

这也点出了类型系统设计的通用原则：如果你能在需要类型 U 的值的地方替换成类型 T 的值，那么类型 T 是类型 U 的子类型的假设就是安全的。这被称为里氏替代原则（Liskov Substitution Principle，LSP）。

[4]注：从技术的角度来看，这里的情况是发生了下界的翻转。类型参数 U 处于负的位置（1 次翻转），而下界（>: T）处于正的位置（2 次翻转）。

397

原则的前提是 T 要支持 U 所支持的操作并且对于 T 的所有操作来说，与 U 的相应操作比较，需求的要更少，而提供的要更多。

在输出通道的例子中，OutputChannel[AnyRef]可以是 OutputChannel[String]的子类型，因为它们都支持同样的 write 操作，并且这个操作对于 OutputChannel[AnyRef]来说要求的比 OutputChannel[String]更少。这里的“更少”是指前者的参数仅需要 AnyRef 对象即可，而后者则需要 String 对象才行。

有时，协变与逆变会同时混合在一个类型中出现。比较典型的例子就是 Scala 的函数特质。比方说，你在编写函数类型 A => B 的时候，Scala 会把它扩展为 Function1[A, B]。而 Function1 在标准库里的定义就同时使用了协变和逆变。Function1 特质的函数参数类型 S 是逆变的，而结果类型 T 是协变的，具体参见清单 19.8。这可以满足里氏替换原则，因为参数是需求的东西，而结果是提供的东西。

```
trait Function1[-S, +T] {
  def apply(x: S): T
}
```

清单 19.8 Function1 的协变和逆变

清单 19.9 中的应用程序给出了一个例子。其中，类 Publication 包含一个参数字段 title，类型为 String。类 Book 扩展了 Publication 并把它的字符串参数 title 前转给超类的构造器。Library 单例对象定义了书籍对象的集（set）及方法 printBookList，它带一个名为 info，类型为 Book => AnyRef 的方法参数。换句话说，也就是 printBookList 唯一的参数是带 Book 参数并返回 AnyRef 的方法。Customer 应用定义了方法 getTitle，输入唯一的参数 Publication，并返回 String，为传入参数的标题。

现在看一下 Customer 的最后一行。这行调用了 Library 的 printBookList 方法并把 getTitle 打包为函数值传递给方法：

```
Library.printBookList(getTitle)
```

尽管函数的结果类型 String 是 AnyRef 的子类型，这行代码仍然要对 printBookList 的 info 参数做类型检查。编译器通过了对代码的检查，因为函数结果类型被声明为协变的（清单 19.8 里的+T）。

398

如果你看过 printBookList 方法体的内部实现，你就可以清楚（编译器）这样做是有道理的。

```
class Publication(val title: String)
class Book(title: String) extends Publication(title)

object Library {
  val books: Set[Book] =
    Set(
      new Book("Programming in Scala"),
      new Book("Walden")
    )
  def printBookList(info: Book => AnyRef) {
    for (book <- books) println(info(book))
  }
}
```

```
object Customer extends Application {
  def getTitle(p: Publication): String = p.title
  Library.printBookList(getTitle)
}
```

清单 19.9 函数类型参数变化型的演示

printBookList 方法枚举了它的书籍列表，并对每一本书调用传入的函数。它把返回的 AnyRef 类型的结果传递给 println，由 println 对这个结果调用 toString 方法并打印。这个行为对于 String 及所有 AnyRef 子类的对象都有效，这也是函数结果类型的协变意义所在。

现在请考虑一下传递给 printBookList 方法的函数参数类型。尽管 printBookList 的参数类型被声明为 Book，而我们传递给它的 getTitle 带 publication 参数，是 Book 的超类型。可以这样做的原因是 printBookList 的参数类型是 Book，所以 printBookList 方法体将仅会传递 Book 对象给该函数。因为 getTitle 的参数类型为 Publication，函数体将只能访问其参数 p，是被声明为类 Publication 的成员。最后还由于任何声明在 Publication 内的方法在它的子类 Book 内也同样有效，因此所有环节都被打通了，这也就是函数参数类型的逆变的意义所在。详情参见图 19.1。

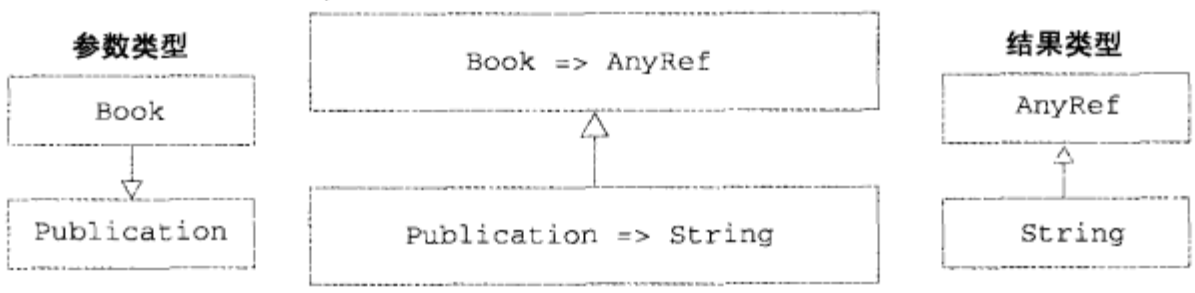


图 19.1 函数类型参数中的协变与逆变

正如图 19.1 所示，Publication => String 是 Book => AnyRef 的子类型，所以清单 19.9 的代码得以编译通过。由于 Function1 的结果类型被定义为协变的，所以可以看到在图右侧显示的两个结果类型的继承关系与在中间的显示的两个函数的关系方向是一致的。相反，由于 Function1 的参数类型被定义为逆变的，所以左侧显示的两个参数类型的继承关系与函数关系的方向相反。

19.7 对象私有数据

到目前为止，Queue 类仍然有一些问题存在。如果 head 被一遍遍地调用许多次，而 leading 列表为空，那么 mirror 操作可能会重复地把 trailing 复制到 leading 列表。这可以通过添加一些做判断的副作用代码对这些无用的复制工作加以避免。清单 19.10 呈现了一种新的 Queue 实现，可以对任意系列的 head 操作执行最多一次 trailing 到 leading 的调节操作。

与之前版本的不同之处在于现在的 leading 和 trailing 都是可以重新赋值的变量，而 mirror 从 trailing 反向复制到 leading 的操作是在当前队列上的副作用，不再返回新的队列。由于 leading 和 trailing 都是私有变量，因此这种副作用纯粹是 Queue 操作的内部实现，从而使得它对于 Queue 的客户来说是不可见的。因此根据第 18 章我们提到过的说法来看，新版本的 Queue 仍然定义了纯函数

式的对象，尽管实际上它们现在已经包含了可重新赋值的字段。

400

你或许想要知道是否这段代码能够通过 Scala 的类型检查。毕竟，队列现在包含两个协变参数类型T的可重新赋值的字段。这难道没有破坏变化型的规则吗？的确如此，原因就在于leading和trailing具有private[this]修饰符，从而都被声明为对象私有的了。

```
class Queue[+T] private (
  private[this] var leading: List[T],
  private[this] var trailing: List[T]
){

  private def mirror() =
    if(leading.isEmpty) {
      while(!trailing.isEmpty) {
        leading = trailing.head :: leading
        trailing = trailing.tail
      }
    }

  def head: T = {
    mirror()
    leading.head
  }

  def tail: Queue[T] = {
    mirror()
    new Queue(leading.tail, trailing)
  }

  def append[U >: T](x: U) =
    new Queue[U](leading, x :: trailing)
}
```

清单 19.10 优化了的函数式队列

13.4 节曾提到对象私有成员仅能在被定义的对象内部访问。而在它们被定义的同一个对象内访问这些变量并不会引起与变化型有关的问题。较直观的解释是，为了构造变化型能够导致类型错误的例子，需要比定义的对象类型更弱的静态类型的对象引用。然而对于对象的私有值的访问来说，这完全不可能。

401

Scala 的变化型检查规则包含了关于对象私有定义的特例。当检查到带有+/–号的类型参数只出现在具有相同变化型分类的位置上时，这种定义将被忽略。因此，清单 19.10 的代码将通过编译，没有错误。而另一方面，如果遗漏了两个 private 修饰符的[this]限定符，你将看到如下的两个类型错误：

```
Queues.scala:1: error: covariant type T occurs in
contravariant position in type List[T] of parameter of
setter leading_=
class Queue[+T] private (private var leading: List[T][5],
                                                  ^

Queues.scala:1: error: covariant type T occurs in
```

[5]译注：协变类型 T 出现在 setter 函数 leading_=参数类型 List[T]的逆变位置上。

```
contravariant position in type List[T] of parameter of
setter trailing_=[6]
                        private var trailing: List[T]) {
```

19.8 上界

在清单 16.1 中，我们演示了一个列表的归并排序函数，它的第一个参数是一个比较函数，第二个是 curry 化的参数，是待排序的列表。这种排序函数的另一种组织方式大概就是把列表的类型混入到 Ordered 特质中。12.4 节曾经提到过，通过把 Ordered 混入到类中，并实现 Ordered 唯一的抽象方法 compare，你能让客户使用<，>，<=，和>=做类实例的比较。例如，清单 19.11 演示了 Ordered 被混入到 Person 类。结果，你就能以如下方式比较两个人（Person）：

```
scala> val robert = new Person("Robert", "Jones")
robert: Person = Robert Jones

scala> val sally = new Person("Sally", "Smith")
sally: Person = Sally Smith

scala> robert < sally
res0: Boolean = true
```

```
class Person(val firstName: String, val lastName: String)
    extends Ordered[Person] {

  def compare(that: Person) = {
    val lastNameComparison =
      lastName.compareToIgnoreCase(that.lastName)
    if (lastNameComparison != 0)
      lastNameComparison
    else
      firstName.compareToIgnoreCase(that.firstName)
  }

  override def toString = firstName +" "+ lastName
}
```

清单 19.11 混入了 Ordered 特质的 Person 类

为了让传递给你的新排序函数的列表类型混入到 Ordered 中，你需要使用上界。除了使用<:符号代替用于下界定义的>:符号以外，上界的指定与下界类似，参见清单 19.12。使用“T <: Ordered[T]”语法，你指明了类型参数 T 具有上界，Ordered[T]。即传递给 orderedMergeSort 的列表的元素类型必须是 Ordered 的子类型。因此，你可以传递 List[Person]给 orderedMergeSort，因为 Person 混入了 Ordered。例如，请看如下列表：

```
def orderedMergeSort[T <: Ordered[T]](xs: List[T]): List[T] = {
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
```

[6]译注：协变类型 T 出现在 setter 函数 trailing_=参数类型 List[T]的逆变位置上。

```
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (x < y) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }
  val n = xs.length / 2
  if (n == 0) xs
  else {
    val (ys, zs) = xs splitAt n
    merge(orderedMergeSort(ys), orderedMergeSort(zs))
  }
}
```

403

清单 19.12 带有上界的归并排序

```
scala> val people = List(
         new Person("Larry", "Wall"),
         new Person("Anders", "Hejlsberg"),
         new Person("Guido", "van Rossum"),
         new Person("Alan", "Kay"),
         new Person("Yukihiro", "Matsumoto")
       )
people: List[Person] = List(Larry Wall, Anders Hejlsberg,
  Guido van Rossum, Alan Kay, Yukihiro Matsumoto)
```

因为这个列表的元素类型 Person 混入了 Ordered[People]（因此也是它的子类型），所以你可以把列表传给 orderedMergeSort：

```
scala> val sortedPeople = orderedMergeSort(people)
sortedPeople: List[Person] = List(Anders Hejlsberg, Alan Kay,
  Yukihiro Matsumoto, Guido van Rossum, Larry Wall)
```

现在，尽管清单 19.12 中的排序函数很好地诠释了上界的用法，但它实际上并非 Scala 中设计能够充分利用 Ordered 特质的排序函数的通用方式。比方说，你不能用 orderedMergeSort 函数对整数列表做排序，因为 Int 类不是 Ordered[Int]的子类型：

```
scala> val wontCompile = orderedMergeSort(List(3, 2, 1))
<console>:5: error: inferred type arguments [Int] do
  not conform to method orderedMergeSort's type
      parameter bounds [T <: Ordered[T]]
        val wontCompile = orderedMergeSort(List(3, 2, 1))
                                           ^
```

404 在 21.6 节，我们将展示如何使用隐式参数及查看约束获得更通用的解决方案。

19.9 小结

本章，你看到了若干信息隐藏的技巧：私有构造器、工厂方法、类型抽象以及对象私有成员。你还学习了如何指定数据类型的变化型以及它对于类的实现意味着什么。最后，你看到了能够帮助获得有弹
405 性的变化型注解的两种技巧：方法类型参数的下界以及本地字段和方法的 private[this]注解。

第 20 章

抽象成员

我们称不完全定义的类或特质的成员为抽象成员。抽象成员将被声明类的子类实现。这种思想在许多面向对象的语言中都可以看到。比方说，Java 允许你声明抽象方法。正如 10.2 节所见，Scala 也可以让你声明这种方法，但并不仅限于此，而是在更一般性的场合实现了这一思想：除了方法之外，你还可以声明抽象字段乃至抽象类型为类和特质的成员。

本章将描述所有的四种抽象成员：val、var、方法及类型。同时还将讨论预初始化字段、懒加载 val、路径依赖类型及枚举类型。

20.1 抽象成员的快速浏览

下面的特质对每种抽象成员各声明了一个例子，它们分别是，类型（T），方法（transform），val（initial），以及 var（current）：

```
trait Abstract {
  type T
  def transform(x: T): T
  val initial: T
  var current: T
}
```

Abstract 的具体实现需要对每种抽象成员填入定义。下面的例子是提供这些定义的实现：

```
class Concrete extends Abstract {
  type T = String
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
}
```

这个实现为类型名 T 提供了具体的含义，它被定义为类型 String 的别名。transform 被定义为参数字符串与其自身连接的操作，而 initial 和 current 值都被设置为"hi"。

这个例子让你对 Scala 中存在什么类型的抽象成员有了粗略的第一印象。本章后续部分将介绍其中的细节并说明，这些抽象成员的新形式与通常的类型成员相比，更适用在什么地方。

20.2 类型成员

就像你在前面一节的例子中所看到的，抽象类型这个术语在 Scala 中是指不带具体定义的，（由“type”关键字）声明为类或特质的成员的类型。类本身可以是抽象的，而特质本来就是抽象的，但不论哪种都不是 Scala 中所指的抽象类型。Scala 的抽象类型永远都是某个类或者特质的成员，就好像特质 Abstract 里的类型 T 那样。

你可以把非抽象的（或者说是“具体的”）类型成员，如 Concrete 类里的类型 T，想象成是为类型定义新的名称，或别名的方式。例如 Concrete 类中，类型 String 被指定了别名 T。因此，任何出现在 Concrete 定义中的 T 指的都是 String。这也包含了 transform 的参数和结果类型，initial，以及 current，这些在 Abstract 超特质中声明的时候提到了 T 的成员。因此，当 Concrete 类实现这些抽象成员[1]的时候，所有的 T 都被解释为 String。

使用类型成员的理由之一是为类型定义短小的、具有说明性的别名，因为类型的实际名称可能比别名要更冗长，或语义不清。这种类型成员有助于净化类或特质的代码。类型成员的另一种主要用途是声明必须被定义为子类的抽象类型。这种用法已经在前面的章节中演示到，将在本章后续部分详细描述。

20.3 抽象 val

抽象 val 以如下形式定义：

```
val initial: String
```

它指定了 val 的名称和类型，但不指定值。该值必须由子类的具体 val 定义提供。例如，Concrete 类以如下方式实现了 val：

```
val initial = "hi"
```

如果你不知道类中定义的值的确切内容，但知道该值对于每个类实例来说都不能改变的时候，可以在类中使用抽象的 val 声明。

抽象的 val 声明类似于抽象的无参数方法声明，如：

```
def initial: String
```

客户代码将使用完全一致的方式（即 obj.initial）引用 val 及方法。然而，如果 initial 是抽象 val，那么客户就获得了保证，每次引用都将得到同样的值。如果 initial 是抽象方法，就不会获得这样的保证，因为在这种情况下 initial 可以被实现为每次调用时都返回不同值的具体方法。

换句话说，抽象的 val 限制了合法的实现方式：任何实现都必须是 val 类型的定义；不可以是 var 或 def。另一方面，抽象方法声明可以被实现为具体的方法定义或者具体的 val 定义。所以，对于清单 20.1 的抽象类 Fruit 定义，类 Apple 是它的合法子类实现，而类 BadApple 不是。

[1]译注：原作方法不妥。

```
abstract class Fruit {
  val v: String //'v'代表value(值)
  def m: String //'m'代表method(方法)
}

abstract class Apple extends Fruit {
  val v: String
  val m: String //可以用'val'重写'def'
}

abstract class BadApple extends Fruit {
  def v: String //错误:不能用'def'重写'val'
  def m: String
}
```

清单 20.1 重写的抽象 val 及无参方法

20.4 抽象 var

与抽象 val 类似，抽象 var 只声明名称和类型，没有初始值。例如，清单 20.2 中的 AbstractTime 特质，声明了名为 hour 和 minute 的抽象变量：

```
trait AbstractTime {
  var hour: Int
  var minute: Int
}
```

清单 20.2 声明抽象 var

类似于 hour 和 minute 这样的抽象 var 表达的是什么意思呢？18.2 节你曾看到过声明为类成员的 var 实际配备了 getter 和 setter 方法。对于抽象 var 来说也是如此。比方说如果你声明了名为 hour 的抽象 var，实际上是隐式声明了抽象 getter 方法、hour 及抽象 setter 方法，hour_=。这里根本没有定义什么可重新赋值的字段——会在之后进入定义了抽象 var 的具体实现的子类。例如，清单 20.2 所示的 AbstractTime 定义完全等价于清单 20.3 所示的定义。

```
trait AbstractTime {
  def hour: Int   // 'hour'的getter方法
  def hour_=(x: Int)  // 'hour'的setter方法
  def minute: Int  // 'minute'的getter方法
  def minute_=(x: Int)  // 'minute'的setter方法
}
```

清单 20.3 抽象 var 是如何被扩展为 getter 和 setter 方法的

20.5 初始化抽象 val

抽象 val 有时会扮演类似于超类的参数这样的角色：它们能够让你在子类中提供超类缺少的细节信息。这对于特质来说尤其重要，因为特质缺少能够用来传递参数的构造器。因此通常参数化特质的方

式就是通过需要在子类中实现的抽象 val 完成。作为例子，可以考虑重新组织第 6 章的 Rational 类（清单 6.5）为特质：

```
trait RationalTrait {
  val numerArg: Int
  val denomArg: Int
}
```

第 6 章的 Rational 类有两个参数：n 是分数的分子，d 是分母。这里给出的 RationalTrait 特质代以用两个抽象 val：numerArg 和 denomArg 定义。为了实例化这个特质，你需要实现抽象的 val 定义。举例如下：

```
new RationalTrait {
  val numerArg = 1
  val denomArg = 2
}
```

这里关键字 new 出现在特质名称（RationalTrait）之前，然后是花括号包围的类结构体。这个表达式可以产生混入了特质并被结构体定义的匿名类（anonymous class）实例。这种特殊的匿名类实例化结果与使用 new Rational(1, 2)的方式创建的实例具有类似效果。不过这种类比并非完美，两者在表达式的初始化顺序方面存在着微妙的差别。当你写下：

411

```
new Rational(expr1, expr2)
```

两个表达式，expr1 和 expr2，会在类 Rational 初始化之前计算，因此在执行类的初始化操作时，expr1 和 expr2 的值已经可用。然而对于特质来说，情况正好相反。当你写下：

```
new RationalTrait {
  val numerArg = expr1
  val denomArg = expr2
}
```

表达式，expr1 和 expr2 被作为匿名类初始化的一部分计算，但匿名类的初始化在 RationalTrait 之后。因此 numerArg 和 denomArg 的值在 RationalTrait 初始化期间还没有准备好（更为精确地说，选用任何值都将得到 Int 类型的默认值，0）。对于之前的 RationalTrait 定义来说，这不是问题，因为特质的初始化没有使用 numerArg 或 denomArg 的值。然而，对于清单 20.4 所展示的 RationalTrait 变例来说，这将成为一个问题，因为其中定义了经过约分的分子和分母：

```
trait RationalTrait {
  val numerArg: Int
  val denomArg: Int
  require(denomArg != 0)
  private val g = gcd(numerArg, denomArg)
  val numer = numerArg / g
  val denom = denomArg / g
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  override def toString = numer +"/"+ denom
}
```

412

清单 20.4 使用了抽象 val 的特质

如果尝试使用某种分子和分母的表达式而并非简单的字面量实例化这个特质，你将得到以下异常信息：

```
scala> val x = 2
x: Int = 2

scala> new RationalTrait {
         val numerArg = 1 * x
         val denomArg = 2 * x
         }
java.lang.IllegalArgumentException: requirement failed
        at scala.Predef$.require(Predef.scala:107)
        at RationalTrait$class.$init$(<console>:7)
        at $anon$1.<init>(<console>:7)
        ....
```

例子中抛出异常的原因在于当类 RationalTrait 初始化的时候，denomArg 仍然为它的默认值 0，使得 require 调用失败。

这个例子演示了类参数和抽象字段的初始化顺序并不一致。类参数在被传递给类构造器之前计算（除非参数是传名的）。相反，子类对于 val 定义的实现，是在超类完成了初始化之后执行的。

现在你应该明白了为什么抽象 val 与参数的行为不一致，最好也应该了解一下对于这种情况我们可以做些什么。是否能定义出可以安全初始化的 RationalTrait，而不用害怕由于未初始化的字段导致的错误？实际上，Scala 为这个问题提供了两种可选方案，预初始化字段和懒加载 val。将在本节后续部分呈现。

fields 预初始化字段

第一种解决方案，预初始化字段，可以让你在调用超类之前初始化子类的字段。操作的方式是把字段定义加上花括号，放在超类构造器调用之前。清单 20.5 就是这样的例子，其中再次尝试了 RationalTrait 实例的创建。正如你所见，初始化段落被提到超特质 RationalTrait 之前。两者以 with 分隔。

```
scala> new {
         val numerArg = 1 * x
         val denomArg = 2 * x
       } with RationalTrait
res15: java.lang.Object with RationalTrait = 1/2
```

清单 20.5 匿名类表达式中的预初始化字段

预初始化字段并不仅限于匿名类；它们还可以被用于对象或有名称的子类。清单 20.6 和清单 20.7 分别是这样的两个例子。从中你可以发现，预初始化段落在每个例子中都被放在定义的对象或类的 extends 关键字之后。清单 20.7 的类 RationalClass 例示了如何在超特质的初始化过程中使用类参数的通用模式。

```
object twoThirds extends {
  val numerArg = 2
```

```
  val denomArg = 3
} with RationalTrait
```

清单 20.6 对象定义中的预初始化字段

由于预初始化的字段在超类构造器调用之前被初始化，因此它们的初始化器不能引用正被构造的对象。相应的结果是，如果有引用 this 的这种初始化器，那么实际指向的是包含了正被构造的类或对象的对象，而不是被构造对象本身。如下：

```
scala> new {
         val numerArg = 1
         val denomArg = this.numerArg * 2
       } with RationalTrait
<console>:8: error: value numerArg is not a
       member of object $iw
           val denomArg = this.numerArg * 2
                                  ^
```

414

例子不能编译通过的原因在于 this.numerArg 引用是在包含 new 的对象中寻找 numerArg 字段(这个例子中是指名为$iw 的合成对象，解释器会把用户输出的语句放在这个对象中)。预初始化字段与类构造器参数在这方面再次取得一致[2]。

```
class RationalClass(n: Int, d: Int) extends {
  val numerArg = n
  val denomArg = d
} with RationalTrait {
  def + (that: RationalClass) = new RationalClass(
    numer * that.denom + that.numer *
    denom, denom * that.denom
  )
}
```

清单 20.7 类定义中的预初始化字段

懒加载 val

你可以使用预初始化字段精确模拟类构造器参数的初始化行为。然而有时你或许更希望让系统本身管理初始化的顺序。这一目的可以通过让你的 val 定义变成懒惰的（lazy）来达成。如果你把 lazy 修饰符前缀在 val 定义上，那么右侧的初始化表达式将直到 val 第一次被使用的时候才计算。

例如，按照如下方式定义的带有 val 的对象 Demo：

```
scala> object Demo {
         val x = { println("initializing x"); "done" }
       }
defined module Demo
```

现在，首先引用 Demo，然后是 Demo.x：

```
scala> Demo
initializing x
res19: Demo.type = Demo$@97d1ff
```

[2]译注：前一次是指都在类初始化之前完成计算；这一次是指都不能引用正在构造的对象值。

```
scala> Demo.x
res20: java.lang.String = done
```

你将发现，用到 Demo 的时候，它的 x 字段就完成了初始化。x 的初始化成为了 Demo 初始化的一部分。然而如果你把 x 字段定义为 lazy 的话，情况就改变了：

```
scala> object Demo {
         lazy val x = { println("initializing x"); "done" }
       }
defined module Demo

scala> Demo
res21: Demo.type = Demo$@d81341

scala> Demo.x
initializing x
res22: java.lang.String = done
```

现在，初始化 Demo 不会执行初始化 x 的调用。x 的初始化将延迟到第一次使用 x 的时候。

这类似于使用 def 把 x 定义为无参方法的情况。然而，与 def 不同的是，懒加载的 val 绝对不会计算第二次。实际上，第一次计算懒加载 val 的时候结果就被保存了下来，以备同样的 val 后续使用。

从这个例子中还能发现，类似于 Demo 这样的对象本身的行为就很像懒加载 val，因为它们也是按需（在第一次使用时）做初始化的。的确如此。实际上 object 关键字定义的（单例）对象可以被看作是用匿名类描述对象内容的懒加载 val 定义的缩写。

使用懒加载 val，你可以把 RationalTrait 重新以清单 20.8 的形式表现。新的特质定义中，所有的具体字段都被定义为 lazy。与之前清单 20.4 的 RationalTrait 定义相比，还有一处变化在于 require 子句从特质的方法体移到了计算 numerArg 和 denomArg 的最大公约数的私有字段 g 的初始化器。经过这些改变，LazyRationalTrait 的初始化就已经没有要做的事了；所有的初始化代码现在都在各个懒加载 val 右侧的结构体中。因此，可以安全地在类被定义之后才执行 LazyRationalTrait 抽象字段的初始化操作。举例如下：

```
trait LazyRationalTrait {
  val numerArg: Int
  val denomArg: Int
  lazy val numer = numerArg / g
  lazy val denom = denomArg / g
  override def toString = numer +"/"+ denom
  private lazy val g = {
    require(denomArg != 0)
    gcd(numerArg, denomArg)
  }
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

清单 20.8 使用懒加载 val 初始化特质

```
scala> val x = 2
x: Int = 2

scala> new LazyRationalTrait {
         val numerArg = 1 * x
         val denomArg = 2 * x
       }
res1: java.lang.Object with LazyRationalTrait = 1/2
```

没有需要预初始化的操作。让我们跟踪分析上面的代码是如何完成初始化并打印输出字符串 1/2 的：

1. 首先，LazyRationalTrait 的新实例被创建出来，特质的初始化代码被运行。该初始化代码为空——没有任何字段被初始化。

2. 之后，由 new 表达式定义的匿名子类的主构造器被执行。它把 numerArg 初始化为 2，把 denomArg 初始化为 4。

417

3. 之后，解释器调用了构造对象的 toString 方法，结果值被打印出来。

4. 之后，numer 字段被特质 LazyRationalTrait 的 toString 方法首次访问，因此它的初始化器执行计算。

5. numer 的初始化器访问了私有字段 g，因此 g 接下来执行计算。这次计算访问了 numerArg 和 denomArg，它们定义在第二步。

6. 之后，toString 方法访问了 denom 值，引发 denom 的计算。这次计算访问了 denomArg 和 g 的值。g 字段的初始化器不再重新计算，因为它已经在第五步执行过。

7. 最终，结果字符串“1/2”被构造出来并被打印。

请注意，在 LazyRationalTrait 类中，g 的定义在代码文本中处于 numer 和 denom 定义之后。尽管如此，因为所有的三个值都是懒加载的，所以 g 将在 numer 和 denom 完成初始化之前被初始化。这说明了懒加载 val 的一个很重要的属性：定义的文本顺序不用多加考虑，因为初始化是按需的。从而，懒加载 val 可以免去你作为程序员不得不认真考虑的问题，即如何安排 val 定义顺序，以确保所有东西在需要的时候已经完成定义。

然而，这种好处仅存在于懒加载 val 的初始化不会产生副作用且更不能依赖于此的前提下。如果存在副作用，那么初始化的顺序就值得考虑了。并且在这种情况下，想要像前面的例子里演示的那样跟踪初始化的顺序将变得非常困难。因此懒加载 val 是对函数式对象的锦上添花，这里初始化的顺序问题无足轻重，只要最后所有的东西都完成初始化就好。它们并不适合于主要为指令式代码的情况。

懒加载函数式语言

Scala 并非第一种开发出懒加载定义与函数式代码完美匹配的语言。实际上，有许多“懒加载函数式编程语言”，它们的每个值及参数都是以懒加载形式完成初始化的。其中最有名的是 Haskell[SPJ02]。

418

20.6 抽象类型

在本章开始的地方，你看到过抽象类型声明“type T”。本章后续将讨论这种抽象类型声明的含义及其适用范围。与所有其他抽象声明一样，抽象类型声明也是将在子类中具体定义的事物的占位符。这里，它是将在之后的类层次中定义的类型。因此上文的 T 是对在声明点尚不可知的类型的引用。不同的子类可以提供不同的 T 实现。

下面是一个能够以自然的方式体现抽象类型的常识性例子。假设给了你一个为动物饮食习惯建模的任务。你或许会以 Food 类和带有 eat 方法的 Animal 类开始工作：

```
class Food
abstract class Animal {
  def eat(food: Food)
}
```

然后你或许会尝试把这两个类特化为 Cow 类吃 Grass 类（牛吃草）：

```
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) {} // 不能编译
}
```

然而，如果你尝试编译新的类，会得到下列编译错误：

```
BuggyAnimals.scala:7: error: class Cow needs to be
abstract, since method eat in class Animal of type
    (Food)Unit is not defined
class Cow extends Animal {
      ^

BuggyAnimals.scala:8: error: method eat overrides nothing
  override def eat(food: Grass) {}
               ^
```

这里的情况是 Cow 类的 eat 方法不能重写 Animal 类的 eat 方法，因为参数类型不同——Cow 类里是 Grass，而 Animal 类里是 Food。

某些人可能会争辩说对于类型系统来说，拒绝这些类显得过分严格了些。他们认为对子类中的方法参数做具体化应该是可行的。然而，如果可以允许类使用这种写法，你将很快失去安全保障。例如，下面的脚本将通过类型检查器：

```
class Food
abstract class Animal {
  def eat(food: Food)
}
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) {} //这将不能通过编译
}                                  //不过如果能的话，……
class Fish extends Food
val bessy: Animal = new Cow
bessy eat (new Fish)       //……你将能用鱼喂牛
```

若放松限制，程序将能够通过编译，因为 Cow 是 Animal，而 Animal 的 eat 方法可以接受任何种类的 Food，甚至包括 Fish。但明显让牛吃鱼是没有好处的！

你应该做的是采用更为精确的建模方式。Animal 的确吃 Food，但 Animal 具体吃什么类型的 Food 取决于 Animal。这可以使用抽象类型干净地表示出来，参见清单 20.9：

```
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}
```

清单 20.9 用抽象类型建模合适的食物

420

有了新的类定义，Animal 就可以只吃适合的食物了。不过到底什么食物合适，这并不在 Animal 类的层面决定。这也就是 SuitableFood 被建模为抽象类型的原因。类型具有上界约束，Food，表达为“<: Food”子句。说明任何（Animal 子类中的）SuitableFood 的具体实例化结果都必须是 Food 的子类。如，不能用 IOException 类实例化 SuitableFood。

```
class Grass extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  override def eat(food: Grass) {}
}
```

清单 20.10 在子类中实现抽象类型

有了以上的 Animal 定义，你现在可以如清单 20.10 所示的那样表达 Cow 类。其中 SuitableFood 被修整为 Grass 并同样为这种类型的食物定义了具体的 eat 方法。这些新的类定义不会出现编译错误。如果你尝试用新的类定义运行“吃_鱼_的_牛”反例，将得到如下编译报错：

```
scala> class Fish extends Food
defined class Fish

scala> val bessy: Animal = new Cow
bessy: Animal = Cow@674bf6

scala> bessy eat (new Fish)
<console> :10: error: type mismatch;
 found    : Fish
 required : bessy.SuitableFood
       bessy eat (new Fish)
                  ^
```

20.7 路径依赖类型

421

看一下最后的那条错误信息：有趣的地方在于 eat 方法所需的类型：bessy.SuitableFood。这个类型由对象引用（bessy）加对象的类型字段 SuitableFood 构成。因此这说明了 Scala 里类型可以做对象的成员。bessy.SuitableFood 的意思是“SuitableFood 类型是由 bessy 引用的对象的

成员”，或者也可以理解为适合于 bessy 的食物类型。类似于 bessy.SuitableFood 的类型被称为路径依赖类型。这里的“路径”是指对象的引用。它可以是单名（如 bessy）或更长的访问路径（如 farm.barn.bessu.SuitableFood），其中的 farm、barn 及 bessy 都是指向这个对象的变量（或单例对象名）。

正如术语“路径依赖类型”所要表述的，类型依赖于路径：通常情况下，不同的路径将产生不同的类型。例如，假定你定义了 DogFood 类和 Dog 类，如下：

```
class DogFood extends Food
class Dog extends Animal {
  type SuitableFood = DogFood
  override def eat(food: DogFood) {}
}
```

如果你尝试把适合牛的饲料用来喂狗，你的代码将无法通过编译：

```
scala> val bessy = new Cow
bessy: Cow = Cow@10cd6d

scala> val lassie = new Dog
bootsie: Dog = Dog@d11fa6

scala> lassie eat (new bessy.SuitableFood)
<console>: 13: error: type mismatch;
 found   : Grass
 required: DogFood
       lassie eat (new bessy.SuitableFood)
                   ^
```

问题在于传递给 eat 方法的 SuitableFood 对象的类型（bessy.SuitableFood），不能匹配 eat 的参数类型，lassie.SuitableFood。然而如果同样是 Dog 的话，情况会不一样。因为 Dog 的 SuitableFood 类型被定义为 DogFood 类的别名，所以对于两条 Dog 来说，它们的 SuitableFood 类型实际是一样的。结果，名为 lassie 的 Dog 实例当然可以吃其他 Dog 实例（我们称其为 bootsie）的食物：

```
scala> val bootsie = new Dog
bootsie: Dog = Dog@54ca71

scala> lassie eat (new bootsie.SuitableFood)
```

路径依赖类型会让我们想起 Java 中的内部类（inner class）语法，但两者有决定性的差别：路径依赖类型表达了外在的对象，而内部类表达了外在的类。Java 类型的内部类同样也可以在 Scala 中表达，不过写法不同。参考如下的两个类，Outer 和 Inner：

```
class Outer {
  class Inner
}
```

Scala 中，内部类的表达形式为 Outer#Inner，而不是 Java 的 Outer.Inner。‘.’语法保留给对象使用。例如，假设你实例化了类型 Outer 的两个对象：

```
val o1 = new Outer
val o2 = new Outer
```

这里 o1.Inner 和 o2.Inner 是两个路径依赖类型(并且是不同的类型)。两种类型都符合于(即,继承于)更一般的类型 Outer#Inner(该类型代表任意 Outer 类型外部对象的 Inner 类)。相反,o1.Inner 类型是指特定(o1 引用的)外部对象的 Inner 类。同理,类型 o2.Inner 指的是不同的特定(o2 引用的)外部对象的 Inner 类。

Scala 中与 Java 中一样,内部类实例持有外部类实例的引用。这使得内部类可以执行类似于访问其外部类的成员的操作。因此不能在没有通过某种方式指定外部类实例的情况下实例化内部类。实例化内部类的方式之一是直接在外部类的方法体中完成。这种情况下将使用当前外部类实例(this 引用的对象);方式之二是使用路径依赖类型。例如,因为类型 o1.Inner 命名了特定外部对象,所以你可以用如下方式实现实例化:

```
scala> new o1.Inner
res1: o1.Inner = Outer$Inner@13727f
```


返回的内部对象将包含其外部对象的引用,即 o1 的对象引用。相反,类型 Outer#Inner 没有指明任何特定 Outer 实例,因此你不能创建它的实例:

```
scala> new Outer#Inner
<console>:6: error: Outer is not a legal prefix for
 a constructor
       new Outer#Inner
                 ^
```

20.8 枚举

路径依赖类型的有趣应用可以在 Scala 的枚举支持中找到。其他语言,包括 Java 和 C#在内,都有语言内建的枚举结构以定义新的类型。Scala 不需要关于枚举的特别语法,而是代之以标准库中的类,scala.Enumeration。想要创建新的枚举,只需定义扩展这个类的对象即可,下面的例子定义了新的 Color 枚举:

```
object Color extends Enumeration {
  val Red = Value
  val Green = Value
  val Blue = Value
}
```

Scala 还能让你简化掉若干右侧一致的相连 val 或 var 定义。与上面等价,你也可以写成:

```
object Color extends Enumeration {
  val Red, Green, Blue = Value
}
```

这个对象定义提供了三个值:Color.Red、Color.Green 及 Color.Blue。你可以引用 Color 的全部内容:

```
import Color._
```

然后简单写成 Red、Green 和 Blue。但这些值的类型是什么？Enumeration 定义了内部类，名为 Value，以及同名的无参方法 Value 返回该类的新对象。也就是说诸如 Color.Red 类的值类型是 Color.Value。而 Color.Value 也正是定义在对象 Color 中的所有枚举值的类型。它是路径依赖类型，其中 Color 是路径，Value 是依赖类型。这里很重要的一点是它是全新的类型，与其他所有的类型都不一样。尤其是，即使你定义另外的枚举，例如：

424

```
object Direction extends Enumeration {
  val North, East, South, West = Value
}
```

那么 Direction.Value 也会与 Color.Value 不同，因为两种类型的路径部分不同。

Scala 的 Enumeration 类还提供了其他语言的枚举设计中所拥有的许多其他特征。你可以通过使用 Value 方法不同的重载变体把名称与枚举值联系起来：

```
object Direction extends Enumeration {
  val North = Value("North")
  val East = Value("East")
  val South = Value("South")
  val West = Value("West")
}
```

你可以使用 foreach 遍历枚举的所有值，或使用带 map、flatMap 和 filter（函数值）的 for 表达式：

```
scala> for (d <- Direction) print(d +" ")
North East South West
```

枚举值从 0 开始计数，你可以用枚举值的 id 方法获得它的计数值：

```
scala> Direction.East.id
res5: Int = 1
```

也可以反过来，通过非零的整数获得 id 为该数的枚举值：

```
scala> Direction(1)
res6: Direction.Value = East
```

以上应该足以引导你开始使用枚举了。你还可以在 Scaladoc 中找到更多关于 scala.Enumeration 类的信息。

425

20.9 案例研究：货币

本章剩余部分提供了一个解释 Scala 中如何使用抽象类型的案例研究。任务是设计 Currency 类。典型的 Currency 实例可以代表美元、欧元、日元或其他货币种类的一笔金额。而且还有可能需要一些关于货币方面的计算。比方说，相同货币的两笔金额应该能够相加。或金额应该能够乘上代表利率的因子。

这些想法产生了如下的 Currency 类的第一个设计：

```
//第一版（有问题的）Currency 类设计
abstract class Currency {
  val amount: Long
  def designation: String
  override def toString = amount +" "+ designation
  def + (that: Currency): Currency = ...
  def * (x: Double): Currency = ...
}
```

货币的 amout 是它所代表货币单位的数量。是 Long 类型的字段，以便可以表达非常大数量的金额，如 Google 或 Microsoft 的市值。可以先让它是抽象的，留待需要用到具体金额数量的子类去定义。designation 是标识货币的字符串。Currency 类的 toString 方法可以显示金额及货币名称。产生的结果如下：

```
79 USD
11000 Yen
99 Euro
```

最后，还有方法+，执行货币加法，及*，执行货币与浮点数的乘法。通过提供具体的 amount 及 designation 值，你可以创建具体的货币值：

```
new Currency {
  val amount = 79L
  def designation = "USD"
}
```


如果我们所要做的只是对单一货币建模，如：仅是美元或者仅是欧元，那么这种设计不存在问题。可是一旦我们需要处理若干货币种类，这样做就不行了。假设你对美元和欧元建模为货币类的两个子类：

```
abstract class Dollar extends Currency {
  def designation = "USD"
}
abstract class Euro extends Currency {
  def designation = "Euro"
}
```

头一眼看上去这么做很有道理。但是这将允许你执行美元和欧元的加法。这种加法的结果为 Currency 类型，但却是把美元和欧元混合在一起组成的古怪货币。你要的应该是+方法更具体化的版本：实现在 Dollar 类中的时候，它应该带 Dollar 参数并产生 Dollar 结果；实现在 Euro 类中的时候，应该带 Euro 参数并产生 Euro 结果。因此加法的类型应该依赖于所在类而改变。尽管如此，你还希望方法只写一次即可，而不是每次定义新的货币都要重写。

Scala 中有一种简单的技巧可以处理类似于这样的情况：如果在类定义的时候某些东西还未知，可以在类中定义它的抽象。这对值和类型都有用。在货币的例子中，加法确切的参数和结果类型未知，因此抽象类型正好可以用在这里。这将产生如下的 AbstractCurrency 类草案：

```
//第二版（仍然不完美的）Currency 类设计
abstract class AbstractCurrency {
  type Currency <: AbstractCurrency
  val amount: Long
  def designation: String
  override def toString = amount +" "+ designation
  def + (that: Currency): Currency = ...
  def * (x: Double): Currency = ...
}
```


它与前面的版本仅有的区别在于：类现在被称为 AbstractCurrency，并且包含了抽象类型 Currency，代表未知的真实货币种类。每种 AbstractCurrency 的具体子类将需要把 Currency 类型修改为这个类本身，从而（能够把两者）“结合在一起”[3]。

例如，下面是新版本的 Dollar 类，它现在扩展自 AbstractCurrency 类：

```
abstract class Dollar extends AbstractCurrency {
  type Currency = Dollar
  def designation = "USD"
}
```

这个设计可以工作，但仍不完美。问题之一是隐藏在 AbstractCurrency 类省略号中的方法定义 +和*。具体化就是，如何在这个类中实现加法？像 this.amout + that.amout 这样计算新货币的正确金额是非常简单的，但怎样才能把金额转变为正确类型的货币？你或许想做这样的尝试：

```
def + (that: Currency): Currency = new Currency {
  val amount = this.amount + that.amount
}
```

然而，这不能通过编译：

```
error: class type required
  def + (that: Currency): Currency = new Currency {
                                     ^
```

Scala 对待抽象类型的一种限制是你既不能创建抽象类型的实例，也不能把抽象类型当作其他类的超类型[4]。因此编译器将拒绝上面例子中的代码实例化 Currency 的尝试。

然而，你可以使用工厂方法绕过这个限制。代之以直接创建抽象类型的实例，可以声明抽象方法间接做到这点。于是，在抽象类型被修改为某种具体类型的地方，你还需要提供工厂方法的具体实现。对于 AbstractCurrency 类来说，应该看上去是下面的样子：


```
abstract class AbstractCurrency {
  type Currency <: AbstractCurrency  // 抽象类型
  def make(amount: Long): Currency   // 工厂方法
  ...                                // 类的其余部分
}
```

这样的设计能够工作，但看起来非常可疑。为什么要把工厂方法放在 AbstractCurrency 类中？

[3]译注：“tying the knot.”表示“系上婚姻的纽带”。

[4]注：最近对于虚拟类的研究有希望允许这种做法，但 Scala 目前还不支持虚拟类。

说它可疑，至少有两点原因。首先，如果你有一笔金额（比如说，一美元），那么你也就握有了能够挣到更多同种货币的能力，就像以下的代码所做的：

```
myDollar.make(100)    //一下就多挣了一百!
```

在彩色复印时代这可能是很有诱惑力的场景，但不用希望这是一桩可以长久干下去不被抓住的买卖。这段代码的问题之二是，如果已经有了 Currency 对象的引用，你就能做出更多的 Currency 对象，但如何得到指定 Currency 的第一个对象呢？你需要另一个创建方法，而它实质上与 make 所做的工作相同。所以这里存在代码重复的现象，这是坏味道代码的明确标志（a sure sign of a code smell）。

解决之道，显而易见，是把抽象类型和工厂方法从 AbstractCurrency 类中移出来。你需要创建另一个类，来包含 AbstractCurrency 类、Currency 类型及 make 工厂方法。我们称其为 CurrencyZone：

```
abstract class CurrencyZone {
  type Currency <: AbstractCurrency
  def make(x: Long): Currency
  abstract class AbstractCurrency {
    val amount: Long
    def designation: String
    override def toString = amount +" "+ designation
    def + (that: Currency): Currency =
      make(this.amount + that.amount)
    def * (x: Double): Currency =
      make((this.amount * x).toLong)
  }
}
```

CurrencyZone 的具体例子是 US。你可以采用以下的方法定义：

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency {
    def designation = "USD"
  }
  type Currency = Dollar
  def make(x: Long) = new Dollar { val amount = x }
}
```

这里，US 是扩展了 CurrencyZone 的对象。它定义了类 Dollar，是 AbstractCurrency 的子类。所以在这个区域内钞票的类型是 US.Dollar。US 对象同样也修整了类型 Currency 为 Dollar 的别名，并提供了 make 工厂方法的实现以返回美元金额。

这是一个可工作的设计。只剩不多的需添加的改进。第一项是有关子单位（subunit）的。目前为止，每种货币都仅用一个测量单位：美元，欧元，或日元。然而，大多数货币都有子单位：例如，在美国，有美元和美分。建模美分最直接的方式是让 US.Currency 的 amount 字段代表美分而不是美元。要想转回（代表）美元，可以向 CurrencyZone 类引入新的字段 CurrencyUnit，以包含此货币一个标准单位的金额。

```
class CurrencyZone {
  ...
```

```
    val CurrencyUnit: Currency
  }
```

US 对象可以定义 Cent，Dollar 的数量及 CurrencyUnit，如清单 20.11 所示。这里的定义与之前的 US 对象定义接近，只是添加了三个新字段。Cent 字段代表 1 US.Currency 的金额。它是一美分硬币的对象类比。Dollar 字段代表 100 US.Currency 的金额。因此 US 对象现在可以用两种方式定义 Dollar 名称。类型 Dollar（由名为 Dollar 的抽象内部类定义）代表在 US 货币区域中有效的 Currency 的泛化名称。相反，Dollar 值（由名为 Dollar 的 val 字段引用）代表一个美元，是一美元纸币的类比。第三个字段定义（CurrencyUnit）特指 US 区域的标准货币单位为 Dollar（即 Dollar 值，它是字段引用，并非 Dollar 类型）。

430

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency {
    def designation = "USD"
  }
  type Currency = Dollar
  def make(cents: Long) = new Dollar {
    val amount = cents
  }
  val Cent = make(1)
  val Dollar = make(100)
  val CurrencyUnit = Dollar
}
```

清单 20.11 美国货币区域

Currency 类的 toString 方法也需要适配为考虑到子单位的情况。例如，10 美元与 23 美分的总和应该打印成小数：10.23 USD。为了做到这点，你可以用如下方式实现 Currency 的 toString 方法：

```
override def toString =
  ((amount.toDouble / CurrencyUnit.amount.toDouble)
   formatted ("%."+ decimals(CurrencyUnit.amount) +"f")
   +" "+ designation)
```

这里，formatted 是 Scala 在若干类（包括 Double）上实现的方法[5]。formatted 方法返回调用该方法的原始字符串根据方法右侧入参的格式字符串完成格式化之后的字符串。传递给 formatted 的格式字符串的语法与 Java 的 String.format 方法中的一致。例如，格式字符串%.2f 可以把数格式化为带两位小数的字符串。上面看到的 toString 中使用的格式字符串是通过对 CurrencyUnit.amount 调用 decimals 方法组装成的。这个方法返回了十进制指数的十进制数字数量减一。例如，decimals(10)是 1，decimals(100)是 2，以此类推。decimals 方法是由一个简单的递归调用实现的：

431

```
private def decimals(n: Long): Int =
  if (n == 1) 0 else 1 + decimals(n / 10)
```

清单 20.12 演示了其他的一些货币区域：

[5]注：Scala 使用了富包装器，具体参见 5.9 节，使得 formatted 方法可用。

```
object Europe extends CurrencyZone {
  abstract class Euro extends AbstractCurrency {
    def designation = "EUR"
  }
  type Currency = Euro
  def make(cents: Long) = new Euro {
    val amount = cents
  }
  val Cent = make(1)
  val Euro = make(100)
  val CurrencyUnit = Euro
}

object Japan extends CurrencyZone {
  abstract class Yen extends AbstractCurrency {
    def designation = "JPY"
  }
  type Currency = Yen
  def make(yen: Long) = new Yen {
    val amount = yen
  }
  val Yen = make(1)
  val CurrencyUnit = Yen
}
```

清单 20.12 欧洲和日本的货币区域

另一项改进在于你可以在模型中添加货币转换的特征。首先，你可以编写 Converter 对象，以包含适用的货币间汇率，如清单 20.13 所示。

432

```
object Converter {
  var exchangeRate = Map(
    "USD" -> Map("USD" -> 1.0   , "EUR" -> 0.7596,
                 "JPY" -> 1.211 , "CHF" -> 1.223),
    "EUR" -> Map("USD" -> 1.316 , "EUR" -> 1.0  ,
                 "JPY" -> 1.594 , "CHF" -> 1.623),
    "JPY" -> Map("USD" -> 0.8257, "EUR" -> 0.6272,
                 "JPY" -> 1.0   , "CHF" -> 1.018),
    "CHF" -> Map("USD" -> 0.8108, "EUR" -> 0.6160,
                 "JPY" -> 0.982 , "CHF" -> 1.0 )
  )
}
```

清单 20.13 带有汇率表的转换器对象

然后，你可以添加转换方法 from 到 Currency 类中，它可以把指定源货币转换为当前 Currency 对象：

```
def from(other: CurrencyZone#AbstractCurrency): Currency =
  make(Math.round(
    other.amount.toDouble * Converter.exchangeRate
      (other.designation)(this.designation)))
```

from 方法可以带任意货币为参数。这个参数被表达为形式上的参数类型（CurrencyZone#

AbstractCurrency），以指明 other 传递的参数必须是某种未知但任意 CurrencyZone 的 AbstractCurrency 类型。它通过把 other 货币的金额乘以 other 与当前货币之间的汇率产生结果[6]。

CurrencyZone 类的最终版参见清单 20.14。你可以在 Scala 的命令 shell（command shell）中测试这个类。我们假设 CurrencyZone 类和所有的具体 CurrencyZone 对象都定义在 org.stairwaybook.currencies 包中。那么第一步就是把这个包的所有东西引用到命令 shell 中：

```
scala> import org.stairwaybook.currencies._
```

```
abstract class CurrencyZone {

  type Currency <: AbstractCurrency
  def make(x: Long): Currency

  abstract class AbstractCurrency {

    val amount: Long
    def designation: String

    def + (that: Currency): Currency =
      make(this.amount + that.amount)
    def * (x: Double): Currency =
      make((this.amount * x).toLong)
    def - (that: Currency): Currency =
      make(this.amount - that.amount)
    def / (that: Double) =
      make((this.amount / that).toLong)
    def / (that: Currency) =
      this.amount.toDouble / that.amount

    def from(other: CurrencyZone#AbstractCurrency): Currency =
      make(Math.round(
        other.amount.toDouble     * Converter.exchangeRate
          (other.designation)(this.designation)))

    private def decimals(n: Long): Int =
      if (n == 1) 0 else 1 + decimals(n / 10)

    override def toString =
      ((amount.toDouble/CurrencyUnit.amount.toDouble)
        formatted ("%."+ decimals(CurrencyUnit.amount)    +"f")
        +" "+ designation)
  }

  val CurrencyUnit: Currency
}
```

清单 20.14 CurrencyZone 的完整代码

接下来你可以完成货币的转换：

[6]注：顺便提一句，例子中你可能认为对日元是在做亏本买卖，但实际上汇率对货币的转换是基于 CurrencyZone 的金额。因此，1.211 是指美分到日元之间的汇率。

```
scala> Japan.Yen from US.Dollar * 100
res1: Japan.Currency = 12110 JPY

scala> Europe.Euro from res1
res2: Europe.Currency = 75.95 EUR

scala> US.Dollar from res2
res3: US.Currency = 99.95 USD
```

完成了三次转换之后我们几乎获得相同的金额，这一事实暗示了这些汇率（的精准性）真的很不错！

相同货币的币值还可以加在一起：

```
scala> US.Dollar * 100 + res3
res4: currencies.US.Currency = 199.95
```

相反，不同的货币金额不能相加：

```
scala> US.Dollar + Europe.Euro
<console>:7: error: type mismatch;
 found   : currencies.Europe.Euro
 required: currencies.US.Currency
        US.Dollar + Europe.Euro
                    ^
```

通过阻止不同单位（在这里，是指货币）的两个值的相加，类型抽象实现了它的功能。它阻止我们执行不合理的计算。在不同单位间的转换失败看上去似乎是微不足道的问题，但它们会引起许多严重的系统故障。其中的一个例子就是1999年9月23日火星气候轨道航天器的坠毁，起因只是由于一个工程组使用了公制单位而另一个使用了英制单位。如果单位被以同样的方式编码，就好像本章货币的编码这样，那么这种错误将在编译期被发现。结果，它却造成了飞行器在将近航行了十个月之后的坠毁（译注：按照事后的推算，应该是进入到火星的大气层而坠毁）。

20.10 小结

435

Scala为面向对象的抽象提供了系统化的并且是非常通用的支持。它不仅使你能在方法上使用抽象，还能在值、变量和类型上使用。本章演示了如何利用这些抽象成员的方法。它们支持系统架构简单但有效的原则：设计类的时候，把所有未知的东西都变成抽象成员。之后类型系统会驱动你的模型开发，就好像你在货币的案例研究中看到的那样。到底未知的是类型、方法、变量或者值，这无关紧要。在Scala

436

中，所有的这些都可以声明为抽象的。

第 21 章

隐式转换和参数

你的代码与其他人的库函数之间有项根本的差别：你可以随意改变或者扩展你自己的代码，但如果你想要使用别人的库函数，通常只能原封不动。

为了缓解这一问题，各种编程语言中涌现出了许多灵感：Ruby 引入了模块，Smalltalk 让不同的包能够加入各自的类中。这些都很有好处，但也非常危险，因为你改变了整个程序中类的行为，其中可能有你想不到的问题。C# 3.0 有静态扩展方法，它更为本地化，同时也更为局限，你只能对类添加方法，而不能添加字段，并且不能让类实现新的接口。

Scala 的办法是隐式转换和隐式参数。它们可以通过忽略那些掩盖了代码中有趣部分的冗长的、过于细节的内容，让现存的库函数处理起来更为得心应手。只要技巧性地使用，那么生成的代码就会更专注于程序的重点，而并非细枝末节。本章将展示给你隐式操作的工作原理，并带给你一些通常的用法。

21.1 隐式转换

在深入隐式转换的细节之前，我们首先看一下它们的典型使用例子。Scala 里的一个核心集合特质是 RandomAccessSeq[T]，它描述了建立在类型 T 的元素上的随机访问序列。RandomAccessSeq 具有大多数你知道的数组或列表的实用方法。take、drop、map、filter、exists，以及 mkString 只是其中的一部分例子。要定义新的随机访问序列，唯一的工作就是扩展 RandomAccessSeq 特质。你只须完成特质中的两个抽象方法的定义：length 和 apply，于是特质中所有其他有用的方法实现就都可以“免费”获得了。

目前为止，一切都很棒。如果你打算定义新的类可以这么做，但已经存在的怎么办呢？或许你会希望同样把别人的库当作随机访问序列使用，即使那些库的设计者没有想到让他们的类扩展 RandomAccessSeq 特质。例如，Java 的 String 可以很好地实现 RandomAccessSeq[Char]，只是很不幸 Java 的 String 类没有继承自 Scala 的 RandomAccessSeq 特质。

这种情况下，隐式操作可以提供帮助。为了让 String 表现为 RandomAccessSeq 的子类，你需要定义从 String 到实际为 RandomAccessSeq 子类的适配类隐式转换。

```
implicit def stringWrapper(s: String) =
  new RandomAccessSeq[Char]  {
    def length = s.length
```

```
    def apply(i: Int) = s.charAt(i)
}
```

这样就可以了[1]。隐式转换只是普通的方法。唯一特殊的地方是它以修饰符 implicit 开始。你也可以显式地通过应用这个方法把 String 转换为 RandomAccessSeq 子类：

```
scala> stringWrapper("abc123") exists (_.isDigit)
res0: Boolean = true
```

但你也可以省略这个转换并仍然获得同样的行为带来的好处：

```
scala> "abc123" exists (_.isDigit)
res1: Boolean = true
```

隐藏在这背后的实际是 Scala 编译器已经为你加入了 StringWrapper 转换。因此实际上它已将最后的表达式改写为之前的那个。但表面上，感觉好像 Java 的 String 已经获得了所有 RandomAccessSeq 特质的有用方法。

438

隐式操作这方面与 C#的扩展方法很相似，它也能让你在现存类中添加新方法。然而，隐式操作远比扩展方法更为简洁。比方说，我们仅需要在 stringWrapper 转换中定义 length 和 apply 方法，于是就可以免费获得所有其他 RandomAccessSeq 中定义的方法。而对于扩展方法来说，你需要再逐一定义。这种重复会使得代码变得更难写，并且更难维护。假设某人之后又为 RandomAccessSeq 添加了新的方法。如果之前你用的是扩展方法，你就需要一个接一个方法的追踪，模仿 RandomAccessSeq，并且须把每个方法都加上。如果你哪怕只是忘了一个方法，系统都不再能保持一致。这简直是陷入了一个维护的梦魇！相反，使用 Scala 的隐式操作，所有的转换将自动包含新增加的方法。

隐式转换的另一个优点是它们支持目标类型的转换。所谓目标类型，是指在代码的某个点需要的类型。例如，假设你在编写方法 printWithSpaces，它可以打印指定随机访问序列中的字符并用空格作字符间的分隔：

```
def printWithSpaces(seq: RandomAccessSeq[Char]) =
  seq mkString " "
```

由于 String 对象可以隐式转换为 RandomAccessSeq，因此可以把字符串传递给 printWithSpaces：

```
scala> printWithSpaces("xyz")
res2: String = x y z
```

这条表达式等同于下面的这条，以显式方式演示了转换的使用：

```
scala> printWithSpaces(stringWrapper("xyz"))
res3: String = x y z
```

本节向你演示了隐式转换的能力，并说明如何用它们“装扮”现存库。下一节你将学习决定何时尝试隐式转换及如何发现它们的规则。

439

[1]注：实际上，Predef 对象已经用类似的功能定义了 stringWrapper 转换，因此你可以在实践中直接使用这个转换，不用再重新定义一个。

21.2 隐式操作规则

隐式定义是指编译器为了修正类型错误而允许插入到程序中的定义。例如，如果 x + y 不能通过类型检查，那么编译器可能会把它改为 covert(x) + y，这里的 convert 是某个可用的隐式转换。如果 convert 可以把 x 改变为某种带了+方法的东西，那么这种转变有可能修复程序以便让它通过类型检查并正确地执行。如果 convert 只是一个非常简单的转换函数，那么从源代码中省略它将有助于代码的净化。

隐式转换由以下通用规则掌控：

标记规则：只有标记为 implicit 的定义才是可用的。implicit 关键字被用来标记编译器可以用于隐式操作的声明。你可以使用它标记任何变量，函数，或对象定义。以下是隐式函数定义的例子[2]：

```
implicit def intToString(x: Int) = x.toString
```

编译器将仅在 convert 被标记为 implicit 时把 x + y 转换为 convert(x) + y。这样，你可以避免编译器把碰巧在作用域之内的随便什么函数都拿来当作“转换器”，从而导致的混乱结果。编译器将从定义中仅选择你明确标记为隐式的操作。

作用域规则：插入的隐式转换必须以单一标识符的形式处于作用域中，或与转换的源或目标类型关联在一起。Scala 编译器将仅考虑处于作用域之内的隐式转换。从而，为了使隐式转换可用，你必须以某种方式把它带入作用域之内，此外，隐式转换就必须以单一标识符的形式（除了一种情况以外，见下文）进入作用域。编译器将不能插入形式为 someVariable.convert 的转换。例如，不可能把 x + y 扩展为 someVariable.convert(x) + y。因此如果你想让 someVariable.convert 可以用作隐式操作，就需要把它引用进来，以作为单一标识符使它可用。一旦引用进来之后，编译器将可以自由地以 convert(x) + y 这种方式应用。实际上，库定义常常包含了 Preamble 对象，以包含若干有用的隐式转换。使用库的代码就可以使用“import Premable._”访问库的隐式转换。

“单一标识符”规则有一个例外。编译器还将在源类型或转换的期望目标类型的伴生对象中寻找隐式定义。例如，如果你尝试传递 Dollar 对象给入参为 Euro 的方法，源类型是 Dollar，而目标类型是 Euro。因此你可以把从 Dollar 到 Euro 的隐式转换打包到 Dollar 或 Euro 类的伴生对象中。下面的例子里隐式转换被放在了 Dollar 的伴生对象中：

```
object Dollar {
  implicit def dollarToEuro(x: Dollar): Euro = ...
}
class Dollar { ... }
```

这里，dollarToEuro 转换被称为是与类型 Dollar 相关联的。编译器将在每次需要转换 Dollar 类型实例的时候实现这样的关联转换，这样就不需要在你的程序中单独地引用转换了。

作用域规则有助于模块推断。在你阅读一份文件的代码时，只有两种情况，要么是引用，要么是通过完全限定名的显式引用，才需要考虑其他的文件。这种好处对于隐式操作来说，至少与显式地把代码

[2]注：标记为隐式操作的变量和单例对象可以被用作隐式参数。这种用法将在本章后续部分描述。

写出来是同样重要的。如果隐式操作在全系统范围有效，那么想要明白一份文件你需要了解程序中任何地方引入的每一个隐式操作！

无歧义规则：隐式转换唯有不存在其他可插入转换的前提下才能插入。如果编译器有两个可选方法修正 `x + y`，比方说既可使用 `convert1(x) + y`，又可使用 `convert2(x) + y`，那么它会报错并拒绝在两者之间做出选择。虽然可以定义某种类型的“最佳匹配”规则让一种转换覆盖其他的。但是这样的选择会导致无法理解的代码。假设编译器选择了 `convert2`，但你对文件并不熟悉，只意识到了 `convert1` 的存在——结果你可能花了大量的时间在思考一个根本没被应用的转换。

441

这种情况下，一种选择是移除一个引用的隐式操作以便消除歧义性。如果你喜欢 `convert2`，那么就移除 `convert1` 的引用。或者，你也可以明确写明你想用的转换：`convert2(x) + y`。

单一调用规则：只会尝试一个隐式操作。编译器将不会把 `x+y` 重写成 `convert1(convert2(x))+y`。这样做会明显延长错误代码的编译时间，并加大程序员编写的代码和实际执行程序之间的差别。理智上，编译器不会尝试在某个隐式操作期间再添加隐式转换。然而，可以通过让隐式操作带隐式参数绕过这个限制，本章之后会对此加以描述。

显式操作先行规则：若编写的代码类型检查无误，则不会尝试任何隐式操作。编译器不会改变已经正常工作的代码。这条规则的推论就是你始终可以用显式内容替换隐式标识符，这样虽然使得代码变得更长但也减少了歧义性的出现。选择可以基于具体事例具体分析。一旦你发现代码似乎过于重复或冗长，那么隐式转换可以帮你减少冗余。一旦代码显得过于简洁以至于有些看不清，你可以插入显式的转换。留待编译器决定插入的隐式操作的多与少，最终只会影响代码的风格。

命名隐式转换。隐式转换可以任意命名。命名仅需要考虑两种情况：你是否需要在方法应用中明确写明，以及决定哪个隐式转换在程序的任何地方都有效。

为了演示第二点，假设你有一个对象带有两个隐式转换：

```
object MyConversions {
  implicit def stringWrapper(s: String):
      RandomAccessSeq[Char] = ...
  implicit def intToString(x: Int): String = ...
}
```

442

你在应用程序中想要使用 `stringWrapper` 转换，但又不希望整数使用 `intToString` 自动转换为字符串。这样你可以只引用一个转换，不要引用另一个：

```
import MyConversions.stringWrapper
... // 使用 stringWrapper 的代码
```

例子中，隐式转换具有名称是很重要的，因为只有用这种方式你才能选择性地只引用一个而不引用另一个。

隐式操作在哪里尝试。Scala 语言中能用到隐式操作的有三个地方：转换为期望类型、指定（方法）调用者的转换、隐式参数。隐式转换为期望类型可以让你在需要不同类型的时候直接使用语境中的类型。例如，你或许有个 `String` 对象并想把它传递给需要 `RandomAccessSeq[Char]` 的方法。（方法）调

用者的转换可以在方法不能应用在原本类型上时让你对方法的调用者（即调用方法的对象）做出适配。例如"abc".exists，将被转换为 StringWrapper("abc").exists，因为 exists 方法并非定义于 String 上，而是定义于 RandomAccessSeq 中。相反，隐式参数通常用于给被调用函数提供关于调用者所期待内容的更多的信息。隐式参数对泛型函数来说尤为有用，这种情况下被调用函数反而可能对于一个或多个入参的类型根本一无所知。以下三节将分别讨论这三种隐式操作

21.3 隐式转换为期望类型

隐式转换为期望类型是编译器会使用隐式操作的第一个地方。规则很简单。一旦编译器看到了 X，但需要 Y，就会检查从 X 到 Y 的隐式转换函数。例如，通常双精度数不能用作整数，因为这样会丢失精度：

```
scala> val i: Int = 3.5
<console>:5: error: type mismatch;
 found   : Double(3.5)
 required: Int
       val i: Int = 3.5
                    ^
```

然而，你可以定义隐式转换消除这一障碍：

```
scala> implicit def doubleToInt(x: Double) = x.toInt
doubleToInt: (Double)Int

scala> val i: Int = 3.5
i: Int = 3
```

这里的情况是，编译器在语境中看到了 Double（具体地说是 3.5），但它需要的是 Int。目前为止，编译器看到了一个普通的类型错误。但是在放弃之前，它搜索了从 Double 到 Int 的隐式转换。例子中，它发现了一个：doubleToInt，因为 doubleToInt 可以当作单一标识符处于作用域之中。（在解释器之外，你可以通过一个 import 语句或者有可能通过继承把 doubleToInt 带入作用域）编译器于是自动插入了 doubleToInt 调用，幕后，代码实际变成了：

```
val i: Int = doubleToInt(3.5)
```

这就是字面意义的隐式转换。你实际没有明确地要求做这种转换。而是把 doubleToInt 作为单一标识符带入作用域，从而把它标记为可用的隐式转换，于是编译器在需要把 Double 转换为 Int 的时候自动使用了它。

Double 到 Int 的转换可能会让人有些皱眉，因为这种为了做某件事而导致精度以不可见的方式降低的主意很令人怀疑。因此这不是我们所推荐的转换。更为合理的方式是反过来，从某些更受限制的类型转换为更为一般的类型。例如，Int 可以被转换成 Double 而没有精度损失，因此从 Int 到 Double 的隐式转换是合理的。实际上，这里就是这么做的。被所有 Scala 程序隐式引用的 scala.Predef 对象，定义了把“较小的”数值类型转变为“较大的”类型的隐式转换。例如，你可以在 Predef 中找到以下的转换：

```
implicit def int2double(x: Int): Double = x.toDouble
```

这也是为什么Scala的Int值可以被存储在Double类型变量中的原因。并非类型系统中有什么专门为此设计的特别规则；这只是应用了隐式转换的结果[3]。

444

21.4 转换（方法调用的）接收者

隐式转换同样还应用于方法调用的接收者，也就是方法调用的对象。这种类型的隐式转换有两种主要用途。第一种，接收者转换使得新的类可以更为平滑地集成到现存类层级中。以及第二种，它们支持在语言中编写域特定语言（DSL）。

要说明它是如何工作的，假设你编写了obj.doIt，但obj并没有名为doIt的成员。编译器将在放弃之前尝试插入转换代码。这里，转换需要应用于接收者obj。编译器会认为obj的期望“类型”应该“拥有名为doIt成员”。这个“有doIt”的类型不是普通的Scala类型，不过它在概念上是存在的，并且是编译器在这种情况下插入隐式转换的理由。

与新类型的交互操作

正如前面提到的，接收者转换的主要用途之一是使得新类型更为平滑地集成到现存类型中。特别地，它们使你能够让客户程序员把现存类型的实例当作你的新类型的实例使用。例如，清单6.5中所示的Rational类，以下是其中的一些片段：

```
class Rational(n: Int, d: Int) {
  ...
  def + (that: Rational): Rational = ...
  def + (that: Int): Rational = ...
}
```

Rational类有两个重载的+号方法变体，分别带Rational和Int为参数。因此你可以要么做有理数之间的加法，要么是有理数加上整数：

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> oneHalf + oneHalf
res4: Rational = 1/1

scala> oneHalf + 1
res5: Rational = 3/2
```

445

然而，如果是类似于1 + oneHalf的表达式呢？这个表达式比较棘手，因为调用者（1）没有合适的+方法。因此下面的调用抛出了错误：

```
scala> 1 + oneHalf
<console>:6: error: overloaded method value + with
```

[3]注：然而，Scala编译器后台将会以特别的方式处理这个转换，把它翻译为特别的“i2d”字节码。因此编译后的镜像与Java中的一致。

```
alternatives (Double)Double  <and>  ... cannot be applied
to (Rational)
    1 + oneHalf
      ^
```

为了允许这种类型的混合数学运算，你需要定义从 Int 到 Rational 的隐式转换：

```
scala> implicit def intToRational(x: Int) =
         new Rational(x, 1)
intToRational: (Int)Rational
```

随着转换方法的到位，对接受者的转换解决了这一棘手问题：

```
scala> 1 + oneHalf
res6: Rational = 3/2
```

这一场景背后的情况是，Scala 编译器首先依照表达式 1 + oneHalf 的原样尝试做类型检查。由于 Int 的若干 + 号方法中没有带 Rational 参数的，因此检查失败。下一步，编译器搜索了 Int 到其他具有+号方法并可以应用于 Rational 的类型转换。它发现了你的转换并应用了方法，结果产生：

```
intToRational(1) + oneHalf
```

在这个例子中，由于你把隐式转换函数的定义输入到了解释器中，使得它处于解释器会话后续部分的作用域之中，从而让编译器得以发现隐式转换函数。

模拟新的语法

隐式转换的另一种主要用途是模拟添加新的语法。请回忆一下你曾经使用如下语法创建 Map 对象：

```
Map(1 -> "one", 2 -> "two", 3 -> "three")
```


你是否曾经想要知道->符号是如何支持的？这不是内建语法！->只是定义在标准 Scala 序文（scala.Predef）中的类 ArrowAssoc 的方法。这个序文还定义了从 Any 到 ArrowAssoc 的隐式转换。在你编写 1 -> "one"的时候，编译器会插入从 1 到 ArrowAssoc 的转换以便找到->方法。以下是相关的定义：

```
package scala
object Predef {
  class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = Tuple2(x, y)
  }
  implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] =
    new ArrowAssoc(x)
  ...
}
```

这个“富包装器”模式在给语言提供类似于语法扩展的库中颇为常见，因此你应该准备好在看到这个模式的时候把它认出来。一旦你看到某人调用的方法不像是存在于接收者类中，那么他们就有可能在使用隐式操作。同样地，如果你看到形如 RichSomething 的类，如 RichInt 或 RichString，那么这个类就有可能是在给类型 Something 添加类似语法的方法。

你在第5章已经看过了用于基本类型的这种富包装器模式。正如你现在看到的，这些富包装器的用途更为宽广，通常让你用定义库的方式得到内部DSL，而这对于其他语言的程序员来说，或许感觉上是需要开发外部DSL的。

21.5 隐式参数

可以让编译器插入隐式操作的最后的地方是通过参数列表。编译器有时会用 someCall(a)(b) 替代 someCall(a)，或者用 new SomeClass(a)(b) 替代 new SomeClass(a)，从而通过添加缺失的参数列表以满足函数调用。被提供的是完整的最后一节柯里化参数，而不仅是最后的参数。例如，若 someCall 缺失的最后参数列表带三个参数，那么编译器会用 someCall(a)(b, c, d) 替代 someCall(a)。对于这种用法来说，不仅是被插入的标识符，如(b, c, d)中的 b，c 和 d，必须在定义的时候被标记为 implicit，连 someCall 或 someClass 定义的最后一个参数列表都必须被标记为 implicit。

以下是一个简单的例子。假设你有 PreferredPrompt 类，其中封装了用户喜欢的 shell 提示字符串（例如，"$ "或"> "）：

```
class PreferredPrompt(val preference: String)
```

同时，假设你有 Greeter 对象带 greet 方法，其中带两个参数列表。第一个参数列表是字符串的用户名，第二个参数列表是 PreferredPrompt：

```
object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt) {
    println("Welcome, "+ name +". The system is ready.")
    println(prompt.preference)
  }
}
```

后面的参数列表被标记为 implicit，也就是说它可以被隐式提供。不过你仍然可以显式提供 prompt，如下：

```
scala> val bobsPrompt = new PreferredPrompt("relax> ")
bobsPrompt: PreferredPrompt = PreferredPrompt@ece6e1

scala> Greeter.greet("Bob")(bobsPrompt)
Welcome, Bob. The system is ready.
relax>
```

为了让编译器隐式提供参数，你必须首先定义期望类型的变量，在这里是 PreferredPrompt。例如，你可以在参数设置对象中这样做：

```
object JoesPrefs {
  implicit val prompt = new PreferredPrompt("Yes, master> ")
}
```

注意 val 本身也被标记为 implicit。如果不是的话，编译器就不能用它来补充缺失的参数列表。而且如果它不以单一标识符处于作用域之内的话，也同样不能用它。例如：

```
scala> Greeter.greet("Joe")
<console>:7: error: no implicit argument matching parameter
  type PreferredPrompt was found.
       Greeter.greet("Joe")
                     ^
```

不过一旦你通过引用把它带入作用域，就可以用来补充缺失的参数列表了：

```
scala> import JoesPrefs._
import JoesPrefs._

scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
Yes, master>
```

注意 implicit 关键字是应用于全体参数列表，而不是单独的参数。清单 21.1 演示了一个例子，其中 Greeter 的 greet 方法的最后的参数列表再次被标记为 implicit，它有两个参数：prompt（PrefferredPrompt 类型）及 drink（PrefferedDrink 类型）：

```
class PreferredPrompt(val preference: String)
class PreferredDrink(val preference: String)

object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt,
      drink: PreferredDrink) {

    println("Welcome, "+ name +". The system is ready.")
    print("But while you work, ")
    println("why not enjoy a cup of "+ drink.preference +"?")
    println(prompt.preference)
  }
}

object JoesPrefs {
  implicit val prompt = new PreferredPrompt("Yes, master> ")
  implicit val drink = new PreferredDrink("tea")
}
```

清单 21.1 带多个参数的隐式参数列表

清单 21.1 中的 JoesPrefs 单例对象声明了两个隐式 val，分别为 PreferredPrompt 类型的 prompt 和 PreferredDrink 类型的 drink。不过与之前一样，只要它们不作为单一标识符处于作用域内，就不会被用于填充 greet 的缺失参数列表：

```
scala> Greeter.greet("Joe")
<console>:8: error: no implicit argument matching parameter
  type PreferredPrompt was found.
       Greeter.greet("Joe")
                     ^
```

你可以用一个 import 语句把两个隐式 val 全部带入作用域：

```
scala> import JoesPrefs._
import JoesPrefs._
```

因为 prompt 和 drink 现在都是以单一标识符的形式处于作用域之内，所以你现在可以用它们来显式地补充最后的参数列表，就像这样：

```
scala> Greeter.greet("Joe")(prompt, drink)
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
Yes, master>
```

并且由于现在已经符合隐式参数的所有规则，你也可以通过省略最后的参数列表的方式让 Scala 编译器为你补充 prompt 和 drink：

```
scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
Yes, master>
```

上面的这个例子中有一件事需要注意，就是我们没有使用 String 作为 prompt 或 drink 的类型，尽管最终它们每一个通过 preference 字段提供出来的实际还是 String 对象。因为编译器选择隐式参数的方式是通过匹配参数类型与作用域内的值类型，隐式参数往往是很“稀少”或者很“特殊”的类型，以至于不会被碰巧匹配。例如，清单 21.1 中分别定义了 PreferredPrompt 和 PreferredDrink 类型以用于隐式参数类型。结果，只要不是被用于 Greeter.greet 的隐式参数，就基本不会出现具有这些类型的隐式变量处于作用域之内的情况。

450

关于隐式参数的另一件需要知道的事情是，或许它们最常见的用法是为了提供前面（在隐式参数列表之前）显式标明的参数列表的类型信息，这与 Haskell 的类型类很类似。例如，考虑一下清单 21.2 里面的 maxListUpBound 函数，它可以返回传入列表的最大元素。

```
def maxListUpBound[T <: Ordered[T]](elements: List[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListUpBound(rest)
      if (x > maxRest) x
      else maxRest
}
```

清单 21.2 带有上界的函数

maxListUpBound 的函数签名与清单 19.12 的 orderedMergeSort 一致。它以 List[T] 为入参，并通过上界指定 T 必须是 Ordered[T] 的子类型。正如在 19.8 节的结尾处提到的，使用这种方式的弱点是你不能把这个函数用于元素类型并非已经是 Ordered 子类型的列表。例如，你不可以使用 maxListUpBound 函数发现整数列表的最大值，因为 Int 类不是 Ordered[Int] 的子类型。

另外，组织 maxListUpBound 的更为一般的方式应该是除 List[T] 入参之外，还需要一个分离的第二入参，一个能够把 T 转换为 Ordered[T] 的函数。清单 21.3 说明了这一方式。例子中，第二参数（orderer）被放在分离的入参列表中并被标记为 implicit。

例子中的 orderer 参数被用来描述 T 对象的排序。在 maxListImpParm 的方法体中，这种排序被用在两个地方：maxListImpParm 的递归调用，以及检查是否列表头大于列表剩余元素的最大值的 if 表达式。

451

```
def maxListImpParm[T](elements: List[T])
      (implicit orderer: T => Ordered[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListImpParm(rest)(orderer)
      if (orderer(x) > maxRest) x
      else maxRest
  }
```

清单 21.3 带有隐式参数的函数

清单 21.3 展示的 maxListImpParm 函数是使用隐式参数提供较早显式标明的参数列表的类型信息的例子。更明确地说，是指隐式参数 order（类型为 T => Ordered[T]），提供了更多的 T 类型的信息——例子里是指如何做 T 对象的排序。类型 T 由参数 elements 的类型（List[T]）标明，它出现在较早的参数列表中。因为 elements 必须永远在 maxListImpParm 函数的每次调用中显式提供，所以编译器将在编译时知道 T，并且从而判决是否 T => Ordered[T] 类型的隐式定义存在于作用域中。如果是，它将隐式地传入第二参数列表 orderer。

这种模式是如此的普通以至于标准 Scala 库为许多通用类型提供了隐式的“orderer”方法。因此可以把 maxListImpParm 方法用在许多类型上：

```
scala> maxListImpParm(List(1,5,10,3))
res10: Int = 10

scala> maxListImpParm(List(1.5, 5.2, 10.7, 3.14159))
res11: Double = 10.7

scala> maxListImpParm(List("one", "two", "three"))
res12: java.lang.String = two
```

在第一个例子中，编译器为 Int 对象插入了 orderer 函数；第二个例子中，是 Double 对象；第三个例子中，是 String。

452

隐式参数的样式规则。样式规则是说，最好对隐式参数的类型使用自定义命名的类型。例如，前面例子中的 prompt 和 drink 类型并非 String，而分别是 PreferredPrompt 和 PreferredDrink。作为反例，假设 maxListImpParm 函数就直接编写为使用如下的类型签名：

```
def maxListPoorStyle[T](elments: List[T])
      (implicit orderer: (T, T) => Boolean): T
```

不过为了使用函数的这个版本，调用者必须提供类型为 (T, T) => Boolean 的 orderer 参数。这种类型过于平凡，包含了任何以两个 T 对象产生 Boolean 值的函数。它根本没能说明这个类型的目

的；可以是相等性判断、小于判断、大于判断或者根本不清楚的什么东西。

清单21.3里，maxListImpParm的实际代码采用了较好的样式。它使用了类型为T => Ordered[T]的orderer参数。类型中的单词Ordered确切指明了隐式参数的用途：为了排序T类型的元素。这个orderer类型更为清晰，因此也扫清了把这种类型的隐式转换加入到标准库中的障碍。相反，可以设想一下如果把(T, T) => Boolean类型的隐式操作加入标准库，而编译器又将它撒播在他人的代码中，必然引发混乱。对你来说代码通过编译并正常运行就结束了，但（你不会发现）这段代码却会相当随意地发起对每对条目的测试！

因此有了样式规则：至少用一个角色确定的名称为隐式参数的类型命名。

21.6 视界

前面的例子中有一个使用隐式操作的机会但却没有使用。请注意，如果把implicit用在参数上，编译器不仅会尝试用隐式值补足这个参数，还会把这个参数当作可用的隐式操作而使用于方法体中！因此，方法体中orderer的两处应用都可以被省略。

453

当编译器检查清单21.4中的代码时，将发现类型不能匹配。例如，T类型的x不存在>方法，因此x > maxRest不起作用。然而编译器并不会立刻停止。

它会首先寻找能够修复这段代码的隐式转换。例子中，它将注意到orderer可用，因此可以把代码转换为orderer(x) > maxRest。并且同样处理了表达式maxList(rest)，把它转换为maxList(rest)(ordered)。经过了这两段隐式操作，方法现在能够通过类型检查了。

```
def maxList[T](elements: List[T])
      (implicit orderer: T => Ordered[T]): T =

  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxList(rest)    // 隐式转换出(orderer)
      if (x > maxRest) x             // 隐式转换出orderer(x)
      else maxRest
}
```

清单21.4 使用了内部隐式参数的函数

仔细看一下maxList，方法的文本中没有一处提到ordered参数的地方。所有对ordered的使用都是隐式的。虽然很令人惊奇，但这种代码模式实际上非常普通。隐式参数只是被用来做转换，因此它本身也可被隐式地使用。

现在，因为参数名从未被显式调用，所以名称可以任意指定。例如，只要不改变maxList的方法体，对于仅改变参数名称来说，方法的行为没有任何改变：

```
def maxList[T](elements: List[T])
      (implicit converter: T => Ordered[T]): T =
  // 相同的方法体
```

因此，即使定义如下也不会有问题：

```
def maxList[T](elements: List[T])
      (implicit iceCream: T => Ordered[T]): T =
  // 相同的方法体
```

由于这种模式很普通，因此 Scala 可以让你省略这个参数的名称并使用视界缩短方法头。通过使用视界，你可以写出如清单 21.5 所列 maxList 方法的签名。

```
def maxList[T <% Ordered[T]](elements: List[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxList(rest)      // 隐式转换出 (orderer)
      if (x > maxRest) x               // 隐式转换出 orderer(x)
      else maxRest
  }
```

清单 21.5 带有视界的函数

可以认为“T <% Ordered[T]”是在说，“任何的 T 都好，只要 T 能被当作 Ordered[T] 即可。”这与说 T 是 Ordered[T] 不一样，这是上界（T <: Ordered[T]）所要表达的意思。例如，尽管 Int 类不是 Ordered[Int] 的子类型，但只要 Int 到 Ordered[Int] 的隐式转换可用，仍然能把 List[Int] 传递给 maxList 方法。此外，如果类型 T 碰巧已经是 Ordered[T] 类型，也仍然可以把 List[T] 传给 maxList。编译器将调用声明在 Predef 中的隐式鉴别函数：

```
implicit def identity[A](x: A): A = x
```

这里，转换什么也没做；它只是简单地把传入的参数再返还出来。

视界与上界

清单 21.2 的 maxListUpBound 函数用上界，T <: Ordered[T]，指定 T 是 Ordered[T] 类型的。相反，清单 21.5 的 maxList 函数用视界，T <% Ordered[T]，指定 T 可以被当作 Ordered[T]。如果你比较了 maxListUpBound 和 maxList 的代码，会发现两者唯一的本质差别在于上界符号，<:，被替换成了视界符号，<%，但清单 21.5 的 maxList 可以用于更多的类型。

21.7 隐式操作调试

隐式操作是 Scala 的非常强大的特征，但有时很难用对也很难调试。本节包含了若干隐式操作调试的秘诀。

有时你或许好奇为什么编译器不能发现你认为应该可以用的隐式转换。这种情况下，把这些转换明确写出来有助于发现问题。如果这样还有错误信息，就应该知道为什么编译器不能应用隐式操作了。例如，假设错误地把 stringWrapper 当作是从 String 对象到 List 对象，而不是 RandomAccessSeq 对象的转换。你可能会不明白为什么下面的代码不能工作：

```
scala> val chars: List[Char] = "xyz"
<console>:12: error: type mismatch;
 found   : java.lang.String("xyz")
 required: List[Char]
       val chars: List[Char] = "xyz"
                               ^
```

这时，你可以明确写出 stringWrapper 转换，来发现到底错在哪里，这样，你就能发现错误的原因：stringWrapper 返回了错误的类型。另一方面，也有可能由于显式地插入了转换使得错误消失。这种情况下，你就知道有其他的什么规则（诸如作用域规则）正在阻止隐式转换的应用。

```
scala> val chars: List[Char] = stringWrapper("xyz")
<console>:12: error: type mismatch;
 found   : java.lang.Object with RandomAccessSeq[Char]
 required: List[Char]
       val chars: List[Char] = stringWrapper("xyz")
                               ^
```

调试程序的时候，如果能看到编译器正在插入的隐式转换可能会有一些帮助。编译器的 -Xprint:typer 选项可以用于这一目的。如果你使用这一选项运行 scalac，编译器就将展示给你在类型检查器添加了所有的隐式转换之后的代码看起来的样子。清单 21.6 和清单 21.7 演示了一个例子。如果观察了这两个清单的最后一行语句的话，你将发现被清单 21.6 中代码省略的 enjoy 的第二参数列表：

```
enjoy("reader")
```

已经在清单 21.7 中被编译器插进去了：

```
Mocha.this.enjoy("reader")(Mocha.this.pref)
```

如果你能做到勇于面对（自己的代码），可以尝试 scala -Xprint:typer 以获得能够打印输出内部使用的已输入源代码的交互式 shell。如果你真这么做了，请做好阅读大量围绕在你代码周围的样板写法的心理准备。

```
object Mocha extends Application {

  class PreferredDrink(val preference: String)

  implicit val pref = new PreferredDrink("mocha")

  def enjoy(name: String)(implicit drink: PreferredDrink) {
    print("Welcome, "+ name)
    print(". Enjoy a ")
    print(drink.preference)
    println("!")
  }
```

```
  enjoy("reader")
}
```

清单 21.6 使用隐式参数的样例代码

```
$ scalac -Xprint:typer mocha.scala
[[syntax trees at end of typer]]// Scala 源码: mocha.scala
package <empty> {
  final object Mocha extends java.lang.Object with Application
      with ScalaObject {
    // ...

    private[this] val pref: Mocha.PreferredDrink =
      new Mocha.this.PreferredDrink("mocha");
    implicit <stable> <accessor>
      def pref: Mocha.PreferredDrink = Mocha.this.pref;
    def enjoy(name: String)
        (implicit drink: Mocha.PreferredDrink): Unit = {
      scala.this.Predef.print("Welcome, ".+(name));
      scala.this.Predef.print(". Enjoy a ");
      scala.this.Predef.print (drink.preference);
      scala.this.Predef.println("!")
    };
    Mocha.this.enjoy("reader")(Mocha.this.pref)
  }
}
```

清单 21.7 完成类型检查和隐式内容插入之后的样例代码

456

21.8 小结

隐式操作是功能强大、代码凝练的 Scala 特性。本章展示了 Scala 的关于隐式操作的规则，并同时说明了若干适用于隐式操作的通用编程场景。

如果要给一些警告的话，隐式操作若过于频繁使用，会让代码变得晦涩难懂。因此，在考虑添加新的隐式转换之前，请首先自问是否能够通过其他手段，诸如继承、混入组合或方法重载，达到同样的目的。如果所有这些都不能成功，并且你感觉代码仍有一些繁复和冗余，那么隐式操作或许正好能帮到你。

458

第 22 章

实现列表

本书中列表简直无处不在。List 类或许是 Scala 里最常用到的结构化数据。第 16 章说明了如何使用列表。本章将“揭开谜底”，透露一下 Scala 中列表是如何实现的。

了解 List 类的内部实现在若干方面是有帮助的。能让人更清楚地知道相对效率更高的列表操作，这将有助于用列表编写快速紧凑的代码，还可以学到能够应用于你自己的库函数设计的一整套技巧。最终，List 类是对 Scala 的类型系统（具体来说，是它的泛型化概念）的精妙应用。因此，对 List 类的研究将巩固这方面的知识。

22.1 List 类原理

列表并非 Scala 的“内建”语言结构，它们由 List 抽象类定义在 scala 包之中，并且包含了两个子类——::和 Nil[1]。下面首先对 List 类作快速浏览。本节带来的是相对于 Scala 标准库中实现的简化版本，标准实现在 22.3 节介绍。

```
package scala
abstract class List[+T] {
```

List 是抽象类，所以不能通过调用空的 List 构造器定义元素。例如“new List”不是合法的表达式。

类具有类型参数 T。类型参数前的+号特指列表是协变的，这在第 19 章已经讨论过。因为这一属性，你可以打比方，如把 List[Int]类型的值赋给类型为 List[Any]的变量：

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> var ys: List[Any] = xs
ys: List[Any] = List(1, 2, 3)
```

所有的列表操作都可以用以下三种基本方法定义：

```
def isEmpty: Boolean
def head: T
def tail: List[T]
```

[1]译注：::是子类，Nil 是子对象，不过依照原文，译作子类。

这三种方法在 List 类中都是抽象的，它们被定义在子对象 Nil 和子类::中。List 的层级参见图 22.1。

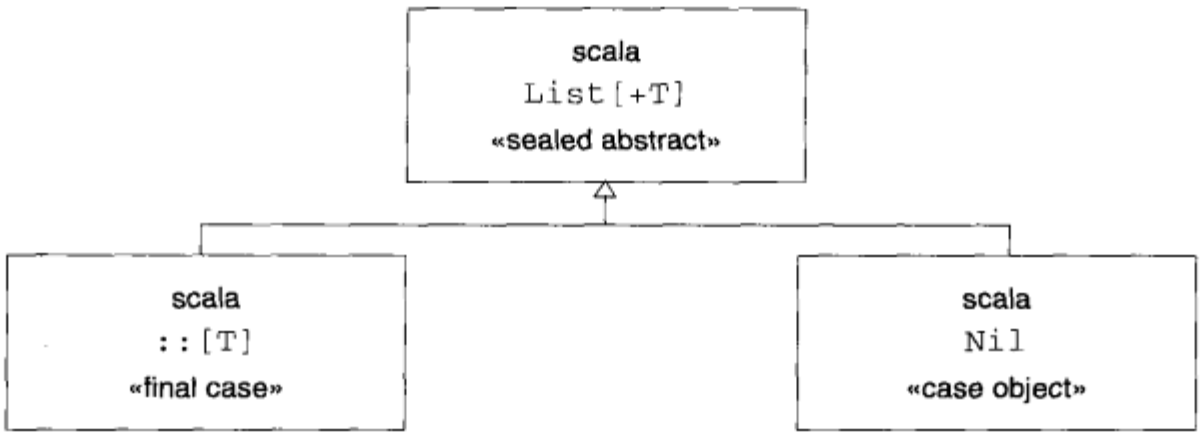


图 22.1 Scala 列表的类层级

Nil 对象

Nil 对象定义了空列表。具体定义参见清单 22.1。Nil 对象继承自 List[Nothing]类型。因为协变的存在，这意味着 Nil 可以兼容任何类型的 List 实例。

460

```
case object Nil extends List[Nothing] {
  override def isEmpty = true
  def head: Nothing =
    throw new NoSuchElementException("head of empty list")
  def tail: List[Nothing] =
    throw new NoSuchElementException("tail of empty list")
}
```

清单 22.1 Nil 单例对象的定义

Nil 对象对这三个抽象方法的实现都很直接：isEmpty 方法返回 true，head 和 tail 方法抛出异常。请注意对于 head 方法来说抛出异常不仅仅是合理的，而且实际上也是唯一可能的做法。因为 Nil 是 Nothing 的列表，所以 head 的结果类型必须为 Nothing。而这个类型的值根本不存在，也就意味着 head 无法返回常规值，因此只能以抛出异常的方式非正常地返回[2]。

::类

::类，发音为“cons”，意为“构造”，代表非空列表。如此命名是为了能够用中缀::支持模式匹配。在 16.5 节看到过，模式中的每个中缀操作都会被当作入参为操作符参数的操作符构造器调用。因此，模式 x :: xs 被当作样本类::的构造器调用，::(x, xs)。以下是::类的定义：

```
final case class ::[T](hd: T, tl: List[T]) extends List[T] {
  def head = hd
  def tail = tl
  override def isEmpty: Boolean = false
}
```

461

::类的实现同样很直接。它带两个参数 hd 和 tl，代表着被构造列表的头和尾部。head 和 tail

[2]注：更精确地说，类型也可以允许 head 操作进入无休止的循环以替代抛出异常，但这明显不符合函数的原意。

方法定义只是简单地返回相应的参数。实际上，这个模式可以被缩略为由参数直接实现超类 List 的 head 和 tail 方法，就好像下面具有等价功能但更为短小的::类定义那样：

```
final case class ::[T](head: T, tail: List[T])
    extends List[T] {

  override def isEmpty: Boolean = false
}
```

这样做之所以可行，是因为所有的样本类参数同时隐含着是类的字段（就好像是前缀 val 的参数声明）。20.3 节曾提到过 Scala 允许你用字段实现抽象的无参数方法，如 head 或 tail。因此，上面的代码直接使用了参数 head 和 tail 作为继承于 List 类的抽象方法 head 和 tail 的实现。

更多的类方法

所有其他的 List 方法都可以用这三种基本方法编写。例如：

```
def length: Int =
  if (isEmpty) 0 else 1 + tail.length
```

或：

```
def drop(n: Int): List[T] =
  if (isEmpty) Nil
  else if (n <= 0) this
  else tail.drop (n - 1)
```

或：

```
def map[U](f: T => U): List[U] =
  if (isEmpty) Nil
  else f(head) :: tail.map(f)
```

列表构建

列表的::和:::构建方法都很特殊，因为它们都以冒号结束，所以实际是绑定在右操作数上的。即，诸如 x::xs 这样的操作会被看作是 xs.::(x)，而不是 x.::(xs)的方法调用。实际上，x.::(xs)没有实际意义，因为 x 是列表元素的类型，可以是任意的，所以我们不能假设这种类型具有::方法。

也由于这个原因，::方法应该输入元素值并生成新列表。那么需要的元素值类型应该是什么呢？你或许会惯性地认为，它应该与列表的元素类型相同，但实际上没有这么局限。为了说明理由，可以参考以下的这种类层级关系：

```
abstract class Fruit
class Apple extends Fruit
class Orange extends Fruit
```

清单 22.2 演示了你在构建水果列表时的情况：

```
scala> val apples = new Apple :: Nil
apples: List[Apple] = List(Apple@585fa9)
```

```
scala> val fruits = new Orange :: apples
fruits: List[Fruit] = List(Orange@cd6798, Apple@585fa9)
```

清单 22.2 在子类型列表上前缀超类型元素

不出所料，apples 被处理成 Apple 的 List。然而，fruits 的例子说明也可以为列表添加不同类型的元素。结果列表的元素类型现在变为 Fruit，是原始列表元素类型（即 Apple）和被添加元素类型（即 Orange）的最精确公共超类型。这种弹性是以清单 22.3 这样的方式定义::（cons）方法而获得的。

```
def ::[U >: T](x: U): List[U] = new scala.::(x, this)
```

清单 22.3 List 类的::（cons）方法定义

463

请注意方法本身是多态的——它带有名为 U 的类型参数，U 被以[U >: T]的方式限定为列表元素 T 的超类型。被添加的元素必须是 U 类型，并且结果为 List[U]。

根据清单 22.3 所示的::格式，可以发现清单 22.2 中的 fruits 定义是如何正确地使用类型以通过检验：定义中::的类型参数 U 被实例化为 Fruit。于是 U 的下界约束得以满足，因为 apples 列表类型为 List[Apple]，并且 Fruit 是 Apple 的超类型。::的参数是 new Orange，符合类型 Fruit。所以，方法应用的结果类型为 List[Fruit]，是正确的。图 22.2 演示了执行清单 22.3 的代码所产生的结果列表。

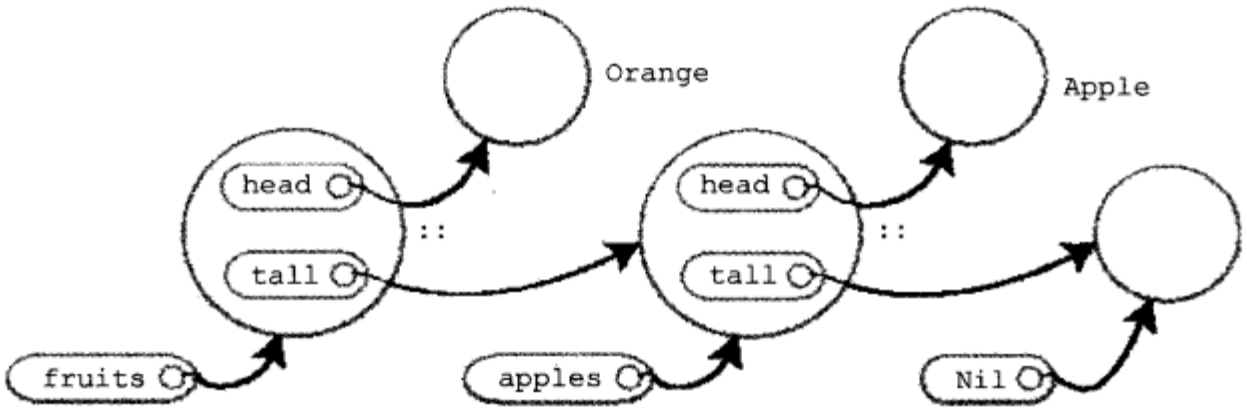


图 22.2 展示的 Scala 列表构造

实际上，带有下界 T 的::多态定义不只是为了方便；这也是为了以正确类型完成 List 类定义的必然结果。这是因为 List 被定义为协变的。假设一段时间里我们把::定义为如下形式：

```
//（不会成功的）脑力实验
def ::(x: T): List[T] = new scala.::(x, this)
```

你已经在第 19 章看到方法参数被认为是逆变位置，因此，上面定义里的列表元素类型 T 处于逆变位置。但这样一来，List 就不能被声明为与 T 保持协变了。因此，下界[U >: T]实在是一石二鸟之计：一方面移除了类型化错误，另一方面也使得::方法在使用的时候更具有弹性。

464

列表连结方法:::定义的方式与::类似，参见清单 22.4。

```
def :::[U >: T](prefix: List[U]): List[U] =
  if (prefix.isEmpty) this
  else prefix.head :: prefix.tail ::: this
```

清单 22.4 List 类的:::方法定义

与 cons 一样，连结方法是多态的。结果类型可以按照需要“放宽”以包含所有列表元素的类型。（同样注意，参数的次序再次在中缀操作与显示方法调用之间被交换）[3]。因为不论:::还是::都以冒号结尾，它们都是右绑定和右关联的。举例来说，清单 22.4 里的:::定义的“else”部分既包含了::也包含了:::。这些中缀操作可以被等价扩展为以下的方法调用：

```
prefix.head :: prefix.tail ::: this
  等价于（因为::和:::是右关联的）
prefix.head :: (prefix.tail ::: this)
  等价于（因为::绑定在右操作元上）
(prefix.tail ::: this).::(prefix.head)
  等价于（因为:::绑定在右操作元上）
this.:::(prefix.tail).::(prefix.head)
```

22.2 ListBuffer 类

列表的典型访问模式就是递归。例如，为了不使用 map 也能实现列表的元素自增，你可以这样写：

```
def incAll(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => x + 1 :: incAll(xs1)
}
```

这种程序模式的缺点在于它不是尾递归的。注意上面 incAll 的递归调用是在::操作之内。因此每次递归调用都需要新的堆栈帧。对于今天的抽象机器来说，这意味着你不能把 incAll 应用于超过大概 30 000 或 50 000 元素的列表上。真是遗憾！

那么如何编写新版本的 incAll 以便工作于任意尺寸（只要有足够的堆空间）的列表呢？方案之一是采用循环：

```
for (x <- xs) // ??
```

但循环体内放什么呢？注意到上面的 incAll 在通过前缀元素到递归调用结果的方式构造列表的地方，因此循环需要把新元素添加到结果列表的后面。一种非常低效的备选方案是使用列表添加操作符::::

```
var result = List[Int]() //非常低效的方法
for (x <- xs) result = result ::: List(x + 1)
result
```

但该方案的效率的确低到令人发指。因为:::的操作时间与它的第一个操作数成正比，所以整个操作所耗费的时间将与列表长度的平方成正比。这明显是不可接受的方案。

较好的可选方案是使用列表缓冲（list buffer），可以让你收集累积列表的元素。使用的方式是通过

[3]译注：就是说，参数次序是倒装的。

调用如“buf += elem”这样的操作，将元素 elem 添加到列表缓冲 buf 的后部。一旦你完成了元素添加的工作，就可以使用 toList 操作把缓冲转换为列表。

ListBuffer 是 scala.collection.mutable 包里的类。如果想仅使用简单名，你需要从这个包引用 ListBuffer：

```
import scala.collection.mutable.ListBuffer
```

使用了列表缓冲之后，incAll 的方法体现在可以写成：

```
val buf = new ListBuffer[Int]
for (x <- xs) buf += x + 1
buf.toList
```

这是一种效率很高的创建列表的方式。实际上，列表缓冲的实现经过了组织，以便于使它的添加操作（+=）及 toList 操作都只要（非常短的）常量时间即可完成。

466

22.3 实际的 List 类

22.1 节给出的列表的方法实现既简洁又清楚，但受制于堆栈溢出问题（与 incAll 的非尾递归实现相同）。因此，List 类真正实现的多数方法都会避免采用递归方案而转用列表缓冲的循环方案作为替代。例如，清单 22.5 演示了 List 类的 map 方法的真正实现：

```
final override def map[U](f: T => U): List[U] = {
  val b = new ListBuffer[U]
  var these = this
  while (!these.isEmpty) {
    b += f(these.head)
    these = these.tail
  }
  b.toList
}
```

清单 22.5 List 类的 map 方法定义

修订后的实现通过简单的循环遍历了列表，效率很高。尾递归实现会有类似的效率，但通常的递归实现会更慢且伸展性差。不过，最后的 b.toList 操作怎样呢？它的复杂度是什么？实际上，toList 方法的调用仅需要少量的循环，与列表长度无关。

想要知道理由的话，请再看一遍用来构造非空列表的::类，事实上，这个类与它在之前 22.1 节给出的理想化的定义并不太一致。真正的定义在清单 22.6 中。

其中有一点比较奇怪：tl 参数竟然是 var！这就意味着在列表构建之后还有机会改变列表的尾部。然而，因为变量 tl 带有 private[scala]修饰符，因此，它只能在 Scala 包中被访问。包外部的客户代码既不能读也不能写这个变量。

467

由于 ListBuffer 类被包含在 Scala 包的子包 scala.collection.mutable 内，因此 ListBuffer 可以访问 tl 字段。实际上列表缓冲的元素被表达为列表，而添加新的元素只是对列表中

最后一个::单元的 tl 字段的修改。以下是 ListBuffer 类的开始部分：

```
final case class ::[U](hd: U,
    private [scala] var tl: List[U]) extends List[U] {
  def head = hd
  def tail = tl
  override def isEmpty: Boolean = false
}
```

清单 22.6 List 子类::的定义

```
package scala.collection.immutable
final class ListBuffer[T] extends Buffer[T] {
  private var start: List[T] = Nil
  private var last0: ::[T] = _
  private var exported: Boolean = false
  ...
```

其中包含了能够说明 ListBuffer 特征的三个私有字段：

start	指向存储于缓冲的所有元素的列表
last0	指向列表中的最后一个::单元
exported	说明缓冲是否曾经通过使用 toList 操作转换为列表

toList 操作非常简单：

```
override def toList: List[T] = {
  exported = !start.isEmpty
  start
}
```

它返回由 start 指向的元素列表，并且如果列表非空，则设置 exported 为 true。所以 toList 非常高效，因为它并不复制存储于 ListBuffer 中的列表。但如果列表在 toList 操作之后又被扩展了，将会怎样呢？当然，一旦列表从 toList 函数返回，就应该保持不变。然而，向 last0 添加后续元素将改变由 start 指向的列表。

为了维护列表缓冲操作的正确性，你需要转向对一个新列表进行操作。这可以通过 += 操作的第一行语句实现：

```
override def += (x: T) {
  if (exported) copy()
  if (start.isEmpty) {
    last0 = new scala.::(x, Nil)
    start = last0
  } else {
    val last1 = last0
    last0 = new scala.::(x, Nil)
    last1.tl = last0
  }
}
```

如果 exported 状态为真，+=就复制 start 指向的列表。所以，最终，没有免费的午餐。如果你想要把可以在后端扩展的列表转变为不可变列表，就一定会有复制存在。然而，ListBuffer 的实现保

证了只有在列表缓冲被转换为列表之后还要扩展的情况下才需要复制。这种情况实际上非常少见。大多数列表缓冲的用例都是逐次添加元素并最后执行一次 toList 操作。这种情况下就无须复制了。

22.4 外在的函数式（风格）

上一节展示了 Scala 的 List 和 ListBuffer 类实现中的关键环节。可以发现，从“外部”来看列表是纯函数式的，但在“内部”则是使用了列表缓冲的指令式实现。这是 Scala 编程的典型策略：通过仔细地取舍非纯粹（函数式）操作的效果，以尝试将纯粹性与效率结合在一起。或许你会问，为什么强调这种
469 纯粹性？为什么不干脆放开列表的定义，让 tail 字段，或者甚至 head 字段成为可变的呢？这种方式的劣势在于它将使程序变得更为脆弱。注意使用::构造列表的时候重用了被构造列表的尾部。因此当编写：

```
val ys = 1 :: xs
val zs = 2 :: xs
```

时，ys 和 zs 列表的尾部是共享的；它们指向同一个数据结构。这是提高效率的要点；如果列表 xs 在每次添加新元素的时候都被复制，则会变慢很多。因为共享无所不在，所以如果改变列表元素变为可能，这将非常危险。例如，根据以上的代码，如果你想要截断 ys 列表为头两个元素：

```
ys.drop(2).tail = Nil    // Scala 中不能这么做！
```

导致的副作用就是：同样也截断了 zs 和 xs 列表。很清楚，想要跟踪到什么改变了将变得非常困难。这也是为什么 Scala 倾向于无所不在的共享以及列表的不可变性的原因。如果你希望的话，ListBuffer 类仍然能够让你以指令式和渐进的方式构建列表。但由于列表缓冲不是列表，因此以类型作为可变缓冲和不可变列表的区隔。

Scala 的 List 和 ListBuffer 设计与 Java 的 String 和 StringBuffer 这一对类的做法非常相像，这并非巧合。因为在两种情况下，设计者都同样希望维持纯粹的、不可变的数据结构，但也都希望提供能够高效、渐进地构造这一结构的方式。对于 Java 和 Scala 的字符串来说，StringBuffer（或者 Java 5 的 StringBuilder）提供了渐进构造字符串的方式。而对于 Scala 的列表来说，你拥有选择权：或者通过使用::添加元素到列表开头的方式渐进构造列表，或者使用列表缓冲把元素添加到结尾。可以根据情况来判断哪种更为适合。通常，在分治风格的递归算法中::用起来更方便，列表缓冲则经常用于更为传统的基于循环的风格中。

22.5 小结

在本章你看到了于 Scala 中是如何实现列表的。List 是 Scala 中最为常用的数据结构之一，而且它
470 的实现已经过了优化。List 的两个子类，Nil 和::，都是样本类。然而，许多核心列表方法都没有采用递归遍历结构的方式，而是使用了 ListBuffer 实现。而 ListBuffer 被细心打造，从而能够在避
471 免占用无关内存的情况下高效实现列表的构建。它的外在表现是函数式的，但内部使用了可变性，以加速通常情况下调用 toList 之后（即废弃缓冲）的表现。经过对所有这些的研究，你现在已经完全了解了列表类的实现原理，甚至已经学到了一两个实现方法的技巧。

第 23 章

重访 For 表达式

第 16 章演示了诸如 map、flatMap 以及 filter 等高阶函数在处理列表时强大的构造能力。不过有时候这些函数所需的抽象级别会让程序变得有些难于理解。举个例子，假定你有一个人员列表，其中每个元素都被定义为 Person 类的实例。Person 类包含了人员名称，是否为男性，以及他/她的孩子的字段。下面给出类的定义：

```
scala> case class Person(name: String,
                         isMale: Boolean,
                         children: Person*)
```

以及一些 person 例子：

```
val lara = Person("Lara", false)
val bob = Person("Bob", true)
val julie = Person("Julie", false, lara, bob)
val persons = List(lara, bob, julie)
```

现在，假定需要找出列表中所有的妈妈和孩子结对的名称。使用 map、flatMap 及 filter，你可以组成如下的查询语句：

```
scala> persons filter (p => !p.isMale) flatMap (p =>
             (p.children map (c => (p.name, c.name))))
res5: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

这个查询语句能够完成工作，但编写和阅读都很不容易。有没有更简单的办法？的确有。还记得 7.3 节的 for 表达式吗？使用 for 表达式，同样的例子可以写成如下形式：

```
scala> for (p <- persons; if !p.isMale; c <- p.children)
       yield (p.name, c.name)
res6: List[(String, String)] = List((Julie,Lara),
     (Julie,Bob))
```

这个表达式的结果与前面完全一致。而且，大多数读者将发现 for 表达式比之前用高阶函数 map、flatMap 及 filter 的查询要清楚很多。

然而，这两个查询的差别并非像看上去那样不同。实际上，Scala 编译器根本就是把第二个查询转译成了第一个。更为一般的说法是，所有的能够 yield（产生）结果的 for 表达式都会被编译器转译为高阶方法 map、flatMap 及 filter 的组合调用。所有的不带 yield 的 for 循环都会被转译为仅

对高阶函数 filter 和 foreach 的调用。

本章，你将首先掌握编写 for 表达式的准确规则。之后，你会看到它们如何使组合问题更易于解决。最后，你还将学到 for 表达是如何转译，并了解如何通过这种转译有助于在新的应用领域“拓展”Scala 语言。

23.1 For 表达式

通常，for 表达式的形式如下：

```
for ( seq ) yield expr
```

这里，seq 由生成器、定义及过滤器组成序列，以分号分隔。示例如下：

```
for (p <- persons; n = p.name; if (n startsWith "To"))
yield n
```


上面的 for 表达式分别包含了一个生成器、一个定义及一个过滤器。7.3 节曾经提到过，还可以用花括号替代小括号包围表达式序列，那样的话，分号就是可选的了：

```
for {
  p <- persons              //生成器
  n = p.name                //定义
  if (n startsWith "To") //过滤器
} yield n
```

生成器的形式如下：

```
pat <- expr
```

表达式 expr 典型的返回值是列表，不过你会在后面看到它可以是泛化的。模式 pat 一一匹配列表的所有元素。如果匹配成功，模式中的变量将绑定元素的相应部分，这与第 15 章所描述的一致。但即使匹配失败也不会抛出 MatchError，而只是在迭代中丢弃这个元素罢了。

最常见的情况下，模式 pat 仅为一个变量 x，如 x <- expr。这种情况下，变量 x 只会逐一迭代 expr 返回的所有元素。

定义的形式如下：

```
pat = expr
```

以上的定义把模式 pat 绑定到表达式 expr 的值。因此与以下的 val 定义效果相同：

```
val x = expr
```

这里模式最常见的情况同样也是简单的变量 x，即，x = expr。也就是定义了 x 为 expr 值的名称。

过滤器的形式如下：

```
if expr
```

这里，expr 是 Boolean 类型的表达式。过滤器能从迭代中滤除所有 expr 返回 false 的元素。

所有的 for 表达式都以生成器开始。如果 for 表达式中有若干生成器，那么后面的生成器比前面的变化的更快[1]。可以用下面简单的测试证实这一点：

```
scala> for (x <- List(1, 2); y <- List("one", "two"))
       yield (x, y)
res0: List[(Int, java.lang.String)] =
  List((1,one), (1,two), (2,one), (2,two))
```

23.2 皇后问题

组合谜题尤其适合于 for 表达式。8 皇后问题就是这样的谜题：要求在标准的棋盘上摆放八个皇后使得没有皇后处于可以被叫吃的位置上（皇后可以叫吃处于同列、同行、或相同对角线上的另一个子）。为了求解这个问题，把它推广为任意尺寸的棋盘实际上更为简单。这样，问题就是在 N×N 平方的棋盘上放 N 个皇后，而尺寸 N 为任意数。我们从 1 开始对格子计数，则 N × N 棋盘左上角的格子的坐标为 (1, 1)，右下角的个子的坐标就是(N, N)。

开始解决 N 皇后问题，首先应该注意需要把皇后放在每一行中。因此可以一行行连续地摆放皇后，并在每次检查新摆放的皇后不会被之前摆放的皇后叫吃。在搜索的过程中，有可能会遇到需要摆放在第 k 行的皇后将所有位置处于第 1 到 k – 1 行的皇后叫吃的局面。这种情况下，你需要放弃这部分的搜索以继续另外一种第 1 到 k – 1 行皇后的配置局面。

这一问题的指令式解决方案会一个接一个地摆放皇后，并把她们移动到棋盘的各个地方。但这看起来难以得出确保尝试了所有可能性的计划。

更为函数式的方法会直接以值的方式表达解决方案，由皇后在棋盘位置的坐标列表组成。然而请注意，完整的解决方案并非可以一步就找到，它需要逐步构建起来，通过各位皇后一行一行试探出来。

这让我们想到了以递归算法来实现。假设你已经产生了在 N × N 大小的棋盘上放置 k 个皇后的所有方案，这里 k 小于 N。那么每个方案都可以被表达为长度为 k 的坐标(row, column)列表，这里的 row 和 column 都是从 1 到 N 的数。把这些部分解决的方案列表当作堆栈会比较方便，可以让第 k 行皇后的坐标放在最上面，接下去是 k – 1 行的皇后坐标，依此类推，直到堆栈的最底部是棋盘第一行皇后放置的坐标。所有的方案在一起组成了列表的列表，其中每个元素都是一个解决方案。

现在，要把下一个皇后放在 k + 1 行，就是对之前的每个方案产生多加一个皇后的所有可能的扩展。这将产生另一个方案列表，这一次的长度是 k + 1。持续这一过程就能让你获得了尺寸为 N 的棋盘的所有解决方案。这一算法思想体现在下面的 placeQueens 函数：

```
def queens(n: Int): List[List[(Int, Int)]] = {
  def placeQueens(k: Int): List[List[(Int, Int)]] =
    if (k == 0)
      List(List())
```

[1]译注：即前面的每变化一次，后面的变化一轮。

```
      else
        for {
          queens <- placeQueens(k - 1)
          column <- 1 to n
          queen = (k, column)
          if isSafe(queen, queens)
        } yield queen :: queens

    placeQueens(n)
  }
```

上面程序中的外部方法 queens 仅以棋盘尺寸 n 作为参数调用了 placeQueens 方法。而函数应用 placeQueens(k)的任务是以列表方式产生所有的长度 k 的部分解决方案。列表的每个元素都是一个方案，用 k 长度的列表来表达。因此 placeQueens 返回的是列表的列表。

如果 placeQueens 的参数 k 为 0，这意味着它需要产生把第零个皇后放在第零行的所有方案。实际上只有一个：根本不要放皇后。这个方案表达为空列表。因此如果 k 为 0，则 placeQueens 返回 List(List())，一个以空列表为单一元素组成的列表。请注意这与空列表 List()非常不同。如果 placeQueens 返回 List()，这意味着没有方案而不是“不放皇后”这唯一的方案。

477

另一种情况下，k 不为 0，那么所有 placeQueens 的工作就都交给 for 表达式完成。for 表达式的第一生成器能够迭代遍历在棋盘上放置 k - 1 个皇后的所有方案。第二生成器迭代遍历第 k 个皇后可以被放置的所有可能的 column。for 表达式的第三个部分把新考虑的 queen 位置定义为列数 k 和每次产生的行数 column 的对偶。for 表达式的第四个部分是使用 isSafe 检查新皇后是否没有被所有之前的皇后叫吃的过滤器（isSafe 的定义将在后面讨论）。

如果新皇后没有被任何其他皇后叫吃，就可以用来组成部分方案，因此 placeQueens 用 queen :: queens 产生了新的方案。如果新皇后不能避免被叫吃，过滤器将返回 false，就不会产生方案。

唯一剩下的就是用来检查是否指定 queen 正被 queens 列表其他元素叫吃的 isSafe 方法。以下是它的定义：

```
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._1 == q2._1 ||  // 同一行
  q1._2 == q2._2 ||  // 同一列
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs       // 对角线
```

isSafe 方法表示如果 queen 没有被 queens 中的任何一个叫吃的话，queen 就是安全的。inCheck 方法表示皇后 q1 和 q2 没有互相叫吃。以下三种情况返回为 true:

1. 如果两个皇后处于同一行；
2. 如果两个皇后处于同一列；
3. 如果两个皇后处于同一对角线，即，行距与列距相等。

第一种情况，即两个皇后处于同一行，在实际应用中不会发生，因为 placeQueens 是把每个皇后分别置于不同的行中。所以即使不做检查也不会改变程序整体的功能性。

23.3 使用 for 表达式做查询

for 注记实质上等同于数据库查询语言的常用操作。例如，假定有一个名为 books 的数据库，代表书目列表，其中 Book 类定义如下：

```
case class Book(title: String, authors: String*)
```

下面是一个小的示例数据库，代表内存中的列表：

```
val books: List[Book] =
  List(
    Book(
      "Structure and Interpretation of Computer Programs",
      "Abelson, Harold", "Sussman, Gerald J."
    ),
    Book(
      "Principles of Compiler Design",
      "Aho, Alfred", "Ullman, Jeffrey"
    ), Book(
      "Programming in Modula-2",
      "Wirth, Niklaus"
    ),
    Book(
      "Elements of ML Programming",
      "Ullman, Jeffrey"
    ),
    Book(
      "The Java Language Specification", "Gosling, James",
      "Joy, Bill", "Steele, Guy", "Bracha, Gilad"
  )
)
```

然后，为了找到姓“Gosling”的作者的所有书名：

```
scala> for (b <- books; a <- b.authors
            if a startsWith "Gosling")
       yield b.title
res0: List[String] = List(The Java Language Specification)
```

或者，为了找到所有包含“Program”的书名：

```
scala> for (b <- books if (b.title indexOf "Program") >= 0)
       yield b.title
res1: List[String] = List(Structure and Interpretation of
  Computer Programs, Programming in Modula-2, Elements of ML Programming)
```

再或者，为了从数据库中找到编写了至少两本书作者：

```
scala> for (b1 <- books; b2 <- books if b1 != b2;
            a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
```

```
        yield a1
res2: List[String] = List(Ullman, Jeffrey, Ullman, Jeffrey)
```

最后一个解决方案不甚完美，因为作者会在结果列表中出现多次。你还需要把结果列表中重复的作者删掉。用以下函数完成：

```
scala> def removeDuplicates[A](xs: List[A]): List[A] = {
         if (xs.isEmpty) xs
         else
           xs.head :: removeDuplicates(
             xs.tail filter (x => x != xs.head)
           )
         }
removeDuplicates: [A](List[A])List[A]

scala> removeDuplicates(res2)
res3: List[java.lang.String] = List(Ullman, Jeffrey)
```

值得注意的是，方法 removeDuplicates 的最后一个表达式可以使用 for 表达式等价地表达为：

```
xs.head :: removeDuplicates(
  for (x <- xs.tail if x != xs.head) yield x
)
```

480

23.4 for 表达式的转译

每个 for 表达式都可以用三个高阶函数 map、flatMap 及 filter 表达。本节描述转移的方案，它也被用于 Scala 编译器。

转译带一个生成器的 for 表达式

首先，假设有一个简单的 for 表达式：

for (x <- $expr_1$) **yield** $expr_2$

这里 x 是变量。这种表达式可以被转译为：

$expr_1$ .map(x => $expr_2$)

转译以生成器和过滤器开始的 for 表达式

现在，考虑组合了带其他元素的前导生成器的 for 表达式。其形式为：

for (x <- $expr_1$ **if** $expr_2$) **yield** $expr_3$

可以转译为：

for (x <- $expr_1$ filter (x => $expr_2$)) **yield** $expr_3$

这次转译得到了另一个比原来的短一个元素的 for 表达式，因为第一个构造器表达式中的 if 元素被改变成 filter 的应用。转译工作于是继续处理第二个表达式，所以最终将获得：

```
expr1 filter (x => expr2 ) map (x => expr3)
```

如果在过滤器之后有更多的元素，同样的转译方案将再次应用。如果 seq 是任意序列的生成器、定义及过滤器，那么：

```
for (x <- expr1 if expr2 ; seq) yield expr3
```

会被转译为：

```
for (x <- expr1 filter expr2 ; seq) yield expr3
```

然后转译工作将继续处理第二表达式，会再缩短一个元素。

转译以两个生成器开始的 for 表达式

下一种情况需要处理以两个生成器[2]开始的 for 表达式，如：

```
for (x <- expr1 ; y <- expr2 ; seq) yield expr3
```

同样，假设 seq 是任意序列的生成器、定义及过滤器。实际上，seq 也可能是空的，在这种情况下 $expr_2$ 之后就没有分号了。不过转译方案在每种情况下都是相同的。上面的 for 表达式被转译为 flatMap 的应用：

```
expr1 .flatMap(x => for (y <- expr2 ; seq) yield expr3 )
```

这次，生成了另一个传递给 flatMap 的函数值形式的 for 表达式。它（已经比原来的表达式简化了一个元素）会在之后依照相同的规则被转换更简单的形式。

以上给出的三个转译方案已经足够处理仅包含生成器和过滤器，并且生成器仅绑定简单变量情况下的所有 for 表达式了。作为例子，23.3 节查询“找出所有出版过至少两本书的作者”

```
for (b1 <- books; b2 <- books if b1 != b2;
     a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
```

这个查询可以被转译为以下 map/flatMap/filter 的组合：

```
books flatMap (b1 =>
  books filter (b2 => b1 != b2) flatMap (b2 =>
    b1.authors flatMap (a1 =>
      b2.authors filter (a2 => a1 == a2) map (a2 =>
        a1))))
```

目前的转译方案还只能处理绑定为简单变量，而非完整模式的生成器。也还不能覆盖有定义的情况。这两个问题将在以下两小节中解释。

转译生成器中的模式

如果生成器的左侧是模式 pat 而不是简单变量，那么转译方案将变得复杂得多。相对处理起来仍然较

[2]译注：原文为 filter“过滤器”，根据上下文关系，应作 generator“生成器”。

为简单的情况是 for 表达式绑定了变量元组。这种情况下，几乎与应用于单个变量的方案一致。以下形式的 for 表达式：

```
for ((x1, ..., xn) <- expr1) yield expr2
```

将转译为：

```
expr.map { case (x1, ..., xn) => expr2 }
```

如果生成器的左侧是任意模式 pat 而不再是简单变量或元组，事情就变得更为复杂了。这种情况下：

```
for (pat <- expr1) yield expr2
```

将被转译为：

```
expr1 filter {
  case pat => true
  case _ => false
} map {
  case pat => expr2
}
```

即，生成的条目首先经过过滤并且仅有那些匹配于 pat 的才会被映射。因此，这也保证了模式匹配生成器不会抛出 MatchError。

上面的方案仅处理 for 表达式包含单个模式匹配生成器的情况。类似的规则可以应用于包含其他生成器、过滤器或定义的 for 表达式。因为这些附加规则没有新亮点，这里就不讨论了。如果你感兴趣，可以在 Scala 语言规格书[Ode08]中找到它们。

转译定义

483

最后还未讨论的情况是 for 表达式包含了内嵌的定义。以下为典型示例：

```
for (x <- expr1 ; y = expr2 ; seq) yield expr3
```

再次假设 seq 是（可能为空的）生成器、定义及过滤器序列。这个表达式可以被转译为：

```
for ((x, y) <- for (x <- expr1) yield (x, expr2); seq)
yield expr3
```

所以每次产生新的 x 值的时候 $expr_2$ 都被重新计算。这样做很有必要，因为 $expr_2$ 可能引用了 x 从而需要对改变的 x 值做重新计算。对于你，一个程序员来说，结论就是，让没有引用某些先前生成器绑定变量的定义内嵌在 for 表达式中或许不是好主意，因为这种表达式的重复计算是很浪费的。例如，代之以：

```
for (x <- 1 to 1000; y = expensiveComputationNotInvolvingX)
yield x * y
```

通常更好的写法是：

```
val y = expensiveComputationNotInvolvingX
for (x <- 1 to 1000) yield x * y
```

转译 for 循环

前面的小节演示了包含 yield 的 for 表达式是如何转译的。那么不返回任何值而仅仅执行了副作用的 for 循环又怎么样呢？它们的转译类似，不过比 for 表达式更简单。原则上，前面的转译方案用到了 map 或 flatMap，而 for 循环的转译方案只需用到 foreach。例如表达式：

```
for (x <- expr1) body
```

将转译为：

```
expr1 foreach (x => body)
```

更大的例子如表达式：

```
for (x <- expr1 ; if expr2 ; y <- expr3 ) body
```

将转译为：

```
expr1 filter (x => expr2 ) foreach (x =>
  expr3 foreach (y => body))
```

又比如，以下表达式把列表的列表形式的矩阵所有元素累加在一起：

```
var sum = 0
for (xs <- xss; x <- xs) sum += x
```

这个循环将被转译为两个嵌套 foreach 的应用：

```
var sum = 0
xss foreach (xs =>
  xs foreach (x =>
    sum += x))
```

23.5 反其道而行之

前面的章节展示了 for 表达式可以被转译为高阶函数 map、flatMap 和 filter 的应用。实际上，你也可以完全相等地反过来进行：map、flatMap 或 filter 的每个应用也可以表达为 for 表达式。以下是三个方法写为 for 表达式的实现方式。方法被放在 Demo 对象内，以区别于 List 的标准操作。在实现上，三种方法都带 List 为参数，但转译方案对于其他集合类型也是一样工作的：

```
object Demo {
  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)

  def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
    for (x <- xs; y <- f(x)) yield y

  def filter[A](xs: List[A], p: A => Boolean): List[A] =
    for (x <- xs if p(x)) yield x
}
```

毫无疑问，Demo.map 方法体中做转译用的 for 表达式将生成对 List 类的 map 调用。同样，

Demo.flatMap 和 Demo.filter 将转译为 List 类的 flatMap 和 filter。

因此这个小演示也说明了 for 表达式具有等价于组合应用 map、flatMap 和 filter 这三种函数的表达能力。

23.6 泛化的 for

因为 for 表达式的转译仅依赖于方法 map、flatMap 和 filter 的搭配，所以才可以把 for 注记应用于大批数据类型上。

你已经看过用在列表和数组上的 for 表达式。它们之所以受到支持是因为列表以及数组都定义了 map、flatMap 和 filter 操作。又因为它们定义了 foreach 方法，所以对这些数据类型的 for 循环也成为了可能。

列表和数组之外，Scala 标准库中还有许多其他类型支持这四种方法从而允许 for 表达式的存在。其中包括范围（Range），迭代器（Iterator），流（Stream）以及所有的集（Set）实现。同样如果你自己的数据类型定义了需要的方法也可以完美地支持 for 表达式。要完整支持 for 表达式和 for 循环，你需要定义 map、flatMap、filter 以及 foreach 为你数据类型的方法。但同样也可以只定义这些方法的子集，从而部分支持 for 表达式或循环。具体规则如下：

- 如果你的类型只定义了 map，可以允许单一生成器组成的 for 表达式。
- 如果定义了 flatMap 和 map，可以允许若干生成器组成的 for 表达式。
- 如果定义了 foreach，允许 for 循环（可以有单一或若干生成器）。
- 如果定义了 filter，for 表达式中允许以 if 开头的过滤器表达式。

486

for 表达式的转译发生在类型检查之前。这可以保持最大的灵活性，因为接下来只需 for 表达式展开的结果通过类型检查即可。Scala 没有定义 for 表达式自身的类型指定规则，并且也不需要 map、flatMap、filter 或 foreach 必须有任何特殊的类型签名。

尽管如此，还是有一个典型的配置可以涵盖 for 表达式所转移的高阶方法最常见的意图。假定有一个参数化的类 C，能够代表某种集合类。那么很自然 map、flatMap、filter 和 foreach 方法应该采用以下的类型签名：

```
abstract class C[A] {
  def map[B](f: A => B): C[B]
  def flatMap[B](f: A => C[B]): C[B]
  def filter(p: A => Boolean): C[A]
  def foreach(b: A => Unit): Unit
}
```

也就是说，map 函数的参数是可以把集合的元素类型 A 转换为某种另外的类型 B 的函数。它可以产生与 C 类型相同但元素类型为 B 的新集合。flatMap 方法的参数是从类型 A 转换到 B 的 C 集合的函数，并产

生 B 的 C 集合。filter 方法的参数是可以把元素类型 A 的集合转换为 Boolean 的论断函数。它产生与被调用对象类型一致的集合。最后，foreach 方法的参数是从 A 到 Unit 的函数，并产生 Unit 结果。

集中注意力到前三个函数上，以下的事实是值得注意的。在函数式编程中，有一种广泛的概念被称为单体（monad），它可以用来解释多种类型的计算，包括从集合、状态和 I/O 操作的计算、回溯计算以及交易等，不一而足。你可以在单体上定制 map、flatMap 及 filter 功能，如果这样做，它们最终具有的正是上文指定的类型。此外，可以用 map、flatMap 及 filter 特性化每个单体，添加可以从元素值产生单体的“单元”构造器。在面向对象的语言中，这种“单元”构造器就是实例构造器或工厂方法。因而，map、flatMap 及 filter 可以被视为函数式单体概念的面向对象版本。因为 for 表达式等同于这三种方法的应用，所以可以被当做是单体的语法。

所有这些都似乎在说明 for 表达式概念是比仅仅迭代集合更为一般性的东西，的确如此。例如，for 表达式同样也在异步 I/O 中扮演重要角色，或者可以作为可选值的替代注记。请注意 Scala 库中 map、flatMap 及 filter 出现的地方——看到它们就意味着可以用 for 表达式以更为简洁的方式操纵该类型的元素。

23.7 小结

本章，你有机会一窥 for 表达式和 for 循环背后的东西。你已经知道了它们会被转译为标准高阶方法集的应用。而这种应用的结果是，发现了 for 表达式的应用远比仅用于集合的迭代要更为广泛，并且可以设计自己类来支持它们。

第 24 章

抽取器（Extractors）

现在你或许已经习惯使用模式匹配这种简洁的方式来做数据的解构和分析。本章将展示如何能够更为深层次地推广这一概念。目前为止，构造器模式还是与样本类相关联，例如，Some(x)之所以是合法的模式是因为 Some 是样本类。有些情况下你或许希望能够在不创建关联的样本类的前提下编写出类似的模式，实际上，你甚至可能希望能够创建自己的模式类型。抽取器便是为此而产生。本章解释了抽取器的概念，以及如何用来定义与对象表达解耦的模式。

24.1 例子：抽取 email 地址

为了演示抽取器能够解决的问题，假想你需要分析一些代表 email 地址的字符串：对于给定的字符串，首先要判断它是否为 email 地址，如果是，则要分别访问地址的用户名和域。传统的方式需使用三个助手函数：

```
def isEMail(s: String): Boolean
def domain(s: String): String
def user(s: String): String
```

有了这些函数，可以用以下方式解析指定字符串 s：

```
if (isEMail(s)) println(user(s) +" AT "+ domain(s)
else println("not an email address")
```

这种做法可行，但有点儿笨手笨脚。而且要是把若干这样的检查组合在一起会使得事情变得过于复杂。比如想要在列表中找到连续的两条相同用户的 email 地址字符串，你可以用前面定义的访问方法自己试一下，看看会变得有多麻烦。

你已经在第 15 章看到过，模式匹配是这种问题的理想工具。让我们假设你可以使用以下模式做字符串匹配：

```
EMail(user, domain)
```

匹配成功的条件是字符串包含了内嵌的“at”符号（@）。这种情况下它将把 user 变量绑定到字符串@符号之前的部分，并把 domain 变量绑定到之后的部分。以此模式为基础，上面的表达式可以更清晰地写为：

```
s match {
  case EMail(user, domain) => println(user +" AT "+ domain)
```

```
  case _ => println("not an email address")
}
```

而“找到两个连续的同一用户的email地址”这一更为复杂的问题也可以被译做以下模式：

```
ss match {
  case EMail(u1, d1) :: EMail(u2, d2) :: _ if (u1 == u2) => ...
  ...
}
```

这远比使用访问函数写出的任何东西都要更为易读。然而，问题在于字符串不是样本类；它们不具有符合于Email(user, domain)这样的表达方式。这也是Scala的抽取器的来由：它们可以让你为已经存在的类型定义新模式，而这种模式不需要遵守类型的内部表达方式。

24.2 抽取器

Scala里的抽取器是具有名为unapply成员方法的对象。unapply方法的目的是为了匹配并分解值。通常抽取器对象还会定义可以构建值的对偶方法apply，但并非必须。例如，清单24.1演示了email地址的抽取器对象：

```
object EMail {

  //注入方法（可选的）
  def apply(user: String, domain: String) = user +"@"+ domain

  //抽取方法（规定的）
  def unapply(str: String): Option[(String, String)] = {
    val parts = str split "@"
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
```

清单24.1 EMail字符串抽取器对象

该对象定义了apply和unapply两种方法。apply方法的含义始终不变：它使得EMail对象可以直接使用括号应用方法的参数，所以可以写EMail(“John”, “epfl.ch”)构造字符串“John@epfl.ch”。为了让这些更为明显，还可以让EMail继承自Scala的函数类型，如：

```
object EMail extends (String, String) => String { ... }
```

> **注意**
>
> 这里对象声明中的“(String, String) => String”，与Function2[String, String, String]同义，是对EMail实现的抽象apply方法的声明，结果就是，比方说可以把EMail传递给需要Function2[String, String, String]的方法。

unapply方法是EMail成为抽取器的原因。从某种意义上来说，它逆转了apply的构建进程。apply是将两个传入字符串拼成email地址字符串返回，而unapply则是传入email地址字符串并返回可能存在的两个字符串：地址的用户和域。但unapply必须还能处理传入字符串并非email地址的情况，这也是

unapply 返回 Option 类型的字符串对偶的原因。结果要么是 Some(user, domain)，如果 str 字符串的确是由 user 和 domain 部分组成的 email 地址[1]，要么是 None，如果 str 不是 email 地址。举例如下：

```
unapply("John@epfl.ch")  等价于  Some("John", "epfl.ch")
unapply("John Doe")  等价于  None
```

现在，一旦模式匹配碰到的是抽取器对象所指的模式，它就会在选择器表达式中调用抽取器的 unapply 方法。例如，执行下列代码：

```
selectorString match { case EMail(user, domain) => ... }
```

将引发调用：

```
EMail.unapply(selectorString)
```

如之前所见，对 EMail.unapply 的调用返回的或是 None，或是 Some(u, d)。对于 Some 类型值来说，u 是地址的用户部分，d 是域部分；如果是 None，则说明模式不能匹配，然后系统会尝试其他的模式或失败返回 MatchError 异常。

在 EMail 模式匹配的例子中，选择器表达式 selectorString 的类型 String，与 unapply 的参数类型一致（在例子中也是 String）。这种情况很常见，但不是绝对的，EMail 抽取器也能用来匹配更广泛类型的选择器表达式。例如，为了检查任意值 x 是否为 email 地址字符串，可以这么写：

```
val x: Any = ...
x match { case EMail(user, domain) => ... }
```

根据这行代码，模式匹配器会首先检查给定值 x 是否符合于 EMail 的 unapply 方法的参数类型 String。如果符合，则 x 值会转型为 String，模式匹配照之前的那样进行。如果不符合，模式立刻失败。

EMail 对象中，apply 方法被称为注入方法，因为它可以传入一些参数并生成指定子集的元素（在我们的例子中：是指 email 地址这种字符串子集）。unapply 方法被称为抽取方法，因为它可以传入相同子集的元素并抽取其中的部分（在我们的例子中：是指用户和域字符串）。注入方法和抽取方法常常同处于一个对象中，因为如此一来你就可以把对象名既当作构造器用也当作模式用，这是对使用样本类作模式匹配的惯例的模仿。不过，也可以只在对象内定义抽取方法，不定义相对应的注入方法。对象本身被称为抽取器，与是否具有 apply 方法无关。

如果包含了注入方法，那么就应该与抽取方法呈对偶关系。例如，调用：

```
EMail.unapply(EMail.apply(user, domain))
```

应该返回：

```
Some(user, domain)
```

也就是说，是被 Some 包装的同一序列的参数。反过来就是首先执行 unapply，然后是 apply，如下：

[1]注：就像在这里演示的，当把元组作为唯一的参数传递给函数的时候，Some 对于元组“(user, domain)”的应用可以省略一对括号。所以，Some(user, domain)与 Some((user, domain))同义。

```
EMail.unapply(obj) match {
  case Some(u, d) => EMail.apply(u, d)
}
```

代码中，如果 obj 匹配成功，就能从 apply 重新得到与原来一样的对象。apply 与 unapply 对偶性的这两个条件是良好的设计原则，Scala 不强求你必须遵守，但建议在设计抽取器的时候保持。

24.3 0 或 1 个变量的模式

493

前面例子中的 unapply 方法在匹配成功的情况下返回了一对元素值。这可以很方便地推广到超过两个变量的模式上。要绑定 N 个变量，unapply 应该返回 N 元的元组，并包括在 Some 中。

不过，模式仅绑定一个变量的情况需要特别对待。Scala 中没有一元的元组。要返回唯一的模式元素，unapply 方法会简单地把它包装在 Some 中。例如，清单 24.2 里的抽取器对象为在一行中同样的字符串出现两次组成的字符串定义了 apply 和 unapply：

```
object Twice {
  def apply(s: String): String = s + s
  def unapply(s: String): Option[String] = {
    val length = s.length / 2
    val half = s.substring(0, length)
    if (half == s.substring(length)) Some(half) else None
  }
}
```

清单 24.2 Twice 字符串抽取器对象

还有一种可能是抽取器模式不绑定任何变量。这种情况下它的 unapply 方法返回布尔值——true 或 false 说明匹配成功或失败。例如，清单 24.3 的抽取器对象能够检查字符串是否全部由大写字符组成：

```
object UpperCase {
  def unapply(s: String): Boolean = s.toUpperCase == s
}
```

清单 24.3 UpperCase 字符串抽取器对象

这次，抽取器仅定义了 unapply，没有定义 apply。因为定义 apply 没有意义，根本没东西可以构造。

下面 userTwiceUpper 函数的模式匹配代码集中应用了所有之前定义的抽取器：

```
def userTwiceUpper(s: String) = s match {
  case EMail(Twice(x @ UpperCase()), domain) =>
    "match: "+ x +" in domain "+ domain
  case _ =>
    "no match"
}
```

494

函数的第一个模式匹配的字符串是 email 地址，并且用户部分需要由大写字母形式的相同字符串出现两次组成。

```
scala> userTwiceUpper("DIDI@hotmail.com")
res0: java.lang.String = match: DI in domain hotmail.com

scala> userTwiceUpper("DIDO@hotmail.com")
res1: java.lang.String = no match

scala> userTwiceUpper("didi@hotmail.com")
res2: java.lang.String = no match
```

注意函数 userTwiceUpper 功能里的 UpperCase 带着空参数列表。这里的()是不能被省略的，否则会执行与 UpperCase 对象的匹配检查！同样还要注意，尽管 UpperCase()本身没有绑定任何变量，但仍然可以把变量与匹配它的整个模式联系起来。如果要这么做，须使用到 15.2 节解释过的变量绑定的标准方案：以 x @ UpperCase()的形式把变量 x 与 UpperCase()匹配的模式联系起来。例如，在上面的第一个 userTwiceUpper 调用中，x 被绑定为"DI"，因为那是匹配于 UpperCase()模式的值。

24.4 变参抽取器

前面的 email 地址抽取方法返回的都是固定数目的元素值，这有些时候不够灵活。例如，你或许希望匹配代表域名的字符串，使得域的每一段都被拆分在不同的子模式中。这可以使你能以如下方式表达模式：

```
dom match {
  case Domain("org", "acm") => println("acm.org")
  case Domain("com", "sun", "java") => println("java.sun.com")
  case Domain("net", _*) => println("a .net domain")
}
```

例子中经过调整使得域以逆序展开——从顶级域一直到子域。这样做的好处是能够更好地使用序列模式。在 15.2 节你曾看到，参数列表末尾的序列通配模式，_*，匹配序列中所有剩下的元素。如果以顶级域摆在前面的方式排列，那么这一特性会更有用，因为这样就可以使用序列通配符匹配任意深度的子域了。

剩下的问题就是抽取器如何支持像之前例子里的那种变参匹配，让模式中可以含有可变数量的子模式。目前所碰到的 unapply 方法并不足以解决这个问题，因为它们每次成功匹配返回的都是固定数目的子元素。为了处理这一情况，Scala 允许定义特别为变参匹配准备的不同的抽取方法。被称为 unapplySeq。具体的写法参见清单 24.4，Domain 抽取器的实现：

```
object Domain {

  // 注入方法（可选）
  def apply(parts: String*): String =
    parts.reverse.mkString(".")

  // 抽取方法（不可选）
  def unapplySeq(whole: String): Option[Seq[String]] =
    Some(whole.split("\\.").reverse)
}
```

清单 24.4 Domain 字符串抽取器对象

Domain 对象定义的 unapplySeq 方法，首先会把字符串以句点为分隔做拆分。具体是通过 Java

的 split 方法，以正则表达式为参数实现。split 的结果是一个字符串数组。然后 unapplySeq 会把数组所有元素倒置并包装在 Some 中返回。

unapplySeq 的结果类型必须与 Option[Seq[T]]一致，这里元素类型 T 不能限制。正如你在 17.2 节看到过，Seq 是 Scala 的集合层次关系中很重要的一个类。它是若干描述各种序列的类（List、Array、RichString 等）的共有超类。

考虑到对称性，Domain 也提供了使用逆序的变参域名字段来构建域名字符串的 apply 方法。apply 方法依旧是可选的。

496

你可以使用 Domain 抽取器获取 email 字符串更多的细节信息。如，寻找在某个“.com”域中名为"tom"的 email 地址，可以写成以下函数：

```
def isTomInDotCom(s: String): Boolean = s match {
  case EMail("tom", Domain("com", _*)) => true
  case _ => false
}
```

并将返回期望的结果：

```
scala> isTomInDotCom("tom@sun.com")
res3: Boolean = true

scala> isTomInDotCom("peter@sun.com")
res4: Boolean = false

scala> isTomInDotCom("tom@acm.org")
res5: Boolean = false
```

同样也可以从 unapplySeq 及变化部分返回固定的元素。表达为包含所有元素的元组，变化部分照例还是处在最后。作为例子，清单 24.5 展示了 email 的新抽取器，其中域名部分已经扩展成序列了：

```
object ExpandedEMail {
  def unapplySeq(email: String)
      : Option[(String, Seq[String])] = {
    val parts = email split "@"
    if (parts.length == 2)
      Some(parts(0), parts(1).split("\\.").reverse)
    else
      None
  }
}
```

清单 24.5　ExpandedEMail 抽取器对象

497

ExpandedEMail 里的 unapplySeq 方法返回对偶（Tuple2 类型）可选值。对偶的第一个元素是用户部分，第二个是表示域名的名称序列。你可以像通常那样匹配它：

```
scala> val s = "tom@support.epfl.ch"
s: java.lang.String = tom@support.epfl.ch

scala> val ExpandedEMail(name, topdom, subdoms @ _*) = s
```

```
name: String = tom
topdom: String = ch
subdoms: Seq[String] = List(epfl, support)
```

24.5 抽取器和序列模式

在 15.2 节你已经知道可以使用序列模式访问列表或数组的元素：

```
List()
1List(x, y, _*)
Array(x, 0, 0, _)
```

实际上，这些序列模式都是使用标准 Scala 库的抽取器实现的。例如，`List(...)`形式的模式是由于 `scala.List` 的伴生对象是定义了 `unapplySeq` 方法的抽取器。清单 24.6 展示了相关的定义：

```
package scala
object List {
  def apply[T](elems: T*) = elems.toList
  def unapplySeq[T](x: List[T]): Option[Seq[T]] = Some(x)
  ...
}
```

清单 24.6 定义了 `unapplySeq` 方法的抽取器

`List` 对象包含了带可变数量参数的 `apply` 方法，从而允许编写如下的表达式：

```
List()
List(1, 2, 3)
```

它还包含了以序列形式返回列表所有元素的 `unapplySeq` 方法，从而对 `List(...)`模式提供了支持。`scala.Array` 对象的定义也非常类似，所以数组也支持注入和抽取方法。

24.6 抽取器 VS.样本类

尽管样本类非常有用，但仍然存在一个缺点：它们暴露了数据的具体表达方式。这是指构造器模式中的类名与选择器对象的具体表达类型是一致的。如果匹配：

```
case C(...)
```

成功，那么就知道了选择器表达式实际就是 C 类的实例。

抽取器截断了数据表达与模式之间的联系。在本节的例子中你已经看到它们可以让模式与被选择对象的数据类型之间毫无关系，这种属性被称为表征独立（representation independence）。在大型开发系统中，表征独立是非常重要的，因为这可以让修改部件的实现类型不会影响部件的使用者。

如果部件已经定义并导出了一套样本类的话，你就必须始终使用它们，因为客户代码可能已经包含了对这些样本类的模式匹配，重命名某些样本类或修改类层级都将影响客户代码。但抽取器不会出现类似的问题，因为它们代表了数据表现和客户观察方式之间的间接适配层，你仍然可以修改类型的具体实现，只要对抽取器做更新即可。

表征独立是抽取器对样本类的一大优势。另一方面，样本类同样有优于抽取器的地方。首先，它们更易于建立和定义、需要更少的代码。其次，它们通常能实现比抽取器更为高效的模式匹配，因为 Scala 编译器对于样本类模式的优化远胜于对抽取器模式的优化。这是由于样本类的机制是固定的，但抽取器的 `unapply` 或 `unapplySeq` 方法几乎能做任何事情。第三，如果样本类继承自 `sealed` 基类，那么 Scala 编译器将采用穷举法检查模式匹配并在模式没有覆盖某种可能的组合值情况下报错，但对于抽取器来说不存在这种穷举检查。

499

那么，对于你的模式匹配来说应该采用这两种方法的哪一种呢？这要分情况讨论。如果编写的代码是封闭的应用，那么样本类通常更为适合，因为它们更简洁，编写速度更快，而且有静态检查。如果你决定之后要修改类层级，应用程序需要重构，但这通常不是问题。另一方面，如果你需要把类型暴露给未知的客户，那么抽取器或许更合适，因为它们能保持表征独立。

幸运的是，你不需要立刻做决定。可以始终以样本类开始，然后如果有新需求，将其改为抽取器。因为抽取器模式与样本类模式在 Scala 中看上去完全一致，所以你的客户代码中的模式匹配还将能够继续工作。

当然，仍然有一些情况：在一开始就说你的模式架构与数据表现类型并不相符。本章讨论的 email 地址就是这样的例子。这种情况下，抽取器是唯一可能的选择。

24.7 正则表达式

正则表达式是非常能够体现抽取器价值的应用领域。类似于 Java、Scala 通过库提供了正则表达式，但抽取器使它能更好地执行交互操作。

形成正则表达式

Scala 从 Java 继承了正则表达式语法，而 Java 则是从 Perl 继承了其中的大多数特征。我们假设你已经知道了这种语法，如果不是的话，也有许多教程能够找到，可以先从 `java.util.regex.Pattern` 类的 Javadoc 文档开始看起。对于新手来说，以下的这些例子已经足够了：

`ab?`	‘a’，可能跟着‘b’。
`\d+`	由一个或多个数字（以`\d`表示）组成的数
`[a-dA-D]\w*`	以字母 a 到 d（包括大小写）开始，后缀为零到多个“单词字符”（表示为`\w`）的单词（所谓单词字符是指字母、数字或下划线）。
`(-)?(\d+)(\.\d*)?`	由可选负号，后跟一或多个数字，由可选的点号及零到多个数字组成的数。该数包含了三个组，即负号、小数点之前部分，以及包含小数点的小数部分。分组以括号封闭。

500

Scala 的正则表达式类放在 `scala.util.matching` 包里。

```
scala> import scala.util.matching.Regex
```

可以通过向 `Regex` 构造器传递字符串构造新的正则表达式值。如：

```
scala> val Decimal = new Regex("(-)?(\\d+)(\\.\\d*)?")
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

注意，与之前给的数值正则表达式相比，上面的字符串里每个反斜杠都出现两次。这是因为 Java 和 Scala 里的单个反斜杠是字符串文本的转义符，而并非出现在字符串中的正规字符。因此代之以“\”，你需要写成“\\”才能在字符串中得到一个反斜杠。

如果正则表达式包含了许多反斜杠的话，那么写和看都会有些痛苦。Scala 的原始字符串为此提供了可选方案。在 5.2 节已经看到，原始字符串是夹在三引号（"""）之间的字符序列。原始字符串和普通字符串之间的差别在于：原始字符串的所有字符都与输入完全一致，包括反斜杠在内，也不被当做转义符处理。因此你可以等价但略具可读性地写成：

```
scala> val Decimal = new Regex("""(-)?(\d+)(\.\d*)?""")
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

正如在解释器的输出中所看到的，产生的 Decimal 结果值与之前完全一致。

另外，Scala 里编写正则表达式更短的方式为：

```
scala> val Decimal = """(-)?(\d+)(\.\d*)?""".r
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?
```

换句话说，就是只要在字符串后添加.r 即可获得正则表达式。因为 RichString 类里包含了名为 r 的方法把字符串转为正则表达式。该方法的定义参见清单 24.7：

```
package scala.runtime
import scala.util.matching.Regex

class RichString(self: String) ... {
  ...
  def r = new Regex(self)
}
```

清单 24.7 RichString 是如何定义 r 方法的

用正则表达式做查找

你可以使用若干不同的操作符查找字符串中是否存在正则表达式（规定的模式）：

```
regex findFirstIn str
```

查找字符串 str 中正则表达式 regex 的首次出现，返回 Option 类型的结果。

```
regex findAllIn str
```

查找字符串 str 中正则表达式 regex 的所有出现，返回 Iterator 结果。

```
regex findPrefixOf str
```

查找字符串 str 开始位置正则表达式 regex 的出现，返回 Option 类型的结果。

例如，可以定义以下输入序列并在其中搜索数值：

```
scala> val Decimal = """(-)?(\d+)(\.\d*)?""".r
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d*)?

scala> val input = "for -1.0 to 99 by 3"
input: java.lang.String = for -1.0 to 99 by 3

scala> for (s <- Decimal findAllIn input)
```

```
println(s)
-1.0
99
3

scala> Decimal findFirstIn input
res1: Option[String] = Some(-1.0)

scala> Decimal findPrefixOf input
res2: Option[String] = None
```

用正则表达式抽取值

另外，Scala 中所有的正则表达式都定义了抽取器，这可以用来鉴别匹配于正则表达式分组的子字符串。例如，可以用如下方式解构数值字符串：

```
scala> val Decimal(sign, integerpart, decimalpart) = "-1.23"
sign: String = -
integerpart: String = 1
decimalpart: String = .23
```

在这个例子中，模式 Decimal(...)被用于 val 定义，如 15.7 节所示。解构的过程是：Decimal 正则表达式值定义了 unapplySeq 方法。这个方法把每个字符串同数值的正则表达式语法做匹配，如果成功，那么正则表达式(-)?(\d+)(\.\d*)?中三个分组对应的部分就以模式元素的形式返回并匹配到三个模式变量 sign，integerpart 及 decimalpart。如果有一个分组缺失，那么元素值被设为 null，参见下面的例子：

```
scala> val Decimal(sign, integerpart, decimalpart) = "1.0"
sign: String = null
integerpart: String = 1
decimalpart: String = .0
```

同样还可以在 for 表达式中混用抽取器和正则表达式做搜索。例如，下面的表达式可以解构从 input 字符串中发现的所有数值：

```
scala> for (Decimal(s, i, d) <- Decimal findAllIn input)
         println("sign: "+ s +", integer: "+
             i +", decimal: "+ d)
sign: -, integer: 1, decimal: .0
sign: null, integer: 99, decimal: null
sign: null, integer: 3, decimal: null
```

24.8 小结

本章你看到了用抽取器推广模式匹配的方式。抽取器可以让你定义模式类型，而不需要与选择的表达式类型相关联。它能让匹配模式的类型更为灵活。这实际上类似于对相同的数据具有了多种可能的视角。它还使你能够在类型表现和客户的观察方式之间插入一层适配层。最终能够让你在做模式匹配的时候维持表征的独立，这对于大型软件系统来说是非常有用的。

抽取器是能够让你定义更有弹性的库抽象的又一把工具。举例来说，它们被大量地使用在 Scala 库中，以方便正则表达式匹配的进行。

第 25 章

注解

注解是添加到程序源代码中的结构化信息。它们与注释很像，可以出现在程序的任何地方并附加到任意变量、方法、表达式或其他程序元素上。但与注释又不一样，它们具有结构性，因此更易于机器处理。

本章说明了 Scala 中如何使用注解。演示了它的一般语法及若干标准注解的使用方式。

但本章并不会说明如何编写新的注解处理工具，因为它不在本书的范围之内，第 29 章将演示一种，但这并非唯一的技巧。本章主要关注的是如何使用注解，因为这比定义新的注解处理器更为寻常。

25.1 为什么要有注解?

除了编译和运行之外，还可以对程序做许多事。例如：

1. 使用 Scaladoc 自动产生文档。
2. 漂亮地打印出符合你偏爱风格的代码。
3. 代码的常见错误检查，如：打开了文件却没（在全部逻辑分支中）关闭。
4. 实验性类型检查，例如副作用管理或所有权属性确认。

这类工具（程序）被称为元编程（meta-programming）工具，因为它们是把其他程序当做输入程序。注解通过让程序员把需要传递给工具的指令标示在他们的源代码中以提供对这些工具的支持。这种指令能让工具比在没有用户输入的情况下更为有效。例如，注解可以在以下方面增强前面所列的功能：

1. 文档生成器可以得到指示，把某些方法在文档中记为已经废除。
2. 排版打印器可以得到指示，跳过已被仔细手工格式化的程序部分。
3. 未关闭文件检查器可以得到指示，忽略已被人工验证为关闭的特定文件。
4. 副作用检查器可以得到指示，验证指定的方法是否有副作用。

所有的这些情况中，在理论上，编程语言提供插入额外信息的方式是可能的，而实际上，它们大多数都是直接从这种或那种语言中得到支持。然而对于一种语言来说实在有太多这样的工具能够直接支持所有的用例，以至于编译器已经完全忽略了所有的信息，毕竟只要代码能运行即可。

Scala 在这种问题上的哲学是，在核心语言中包含最少的、正交的支持以便于编写大量的元编程工

具。在这里，最少的支持就是注解系统。编译器只知道一个特性：注解，但并不把任何含义与注解个体联系在一起。每种元编程工具都可以以此为基础，定义并使用自己特定的注解。

25.2 注解语法

注解的典型使用方式如下：

```
@deprecated def bigMistake() = //...
```

注解为@deprecated 部分，它应用于 bigMistake[1]方法（省略——细节过于尴尬）的全体。这种情况下，方法被标记为 bigMistake 作者不希望你使用的东西。或许在代码的未来版本中 bigMistake 将被整体移除。

506

上面的例子里，是方法被加上了@deprecated 注解。注解还可以用在其他地方，它可以存在于任何种类的声明或定义之上，包括 val、var、def、class、object、trait，以及 type。并应用于跟随其后的声明或定义的整体：

```
@deprecated class QuickAndDirty {
  //...
}
```

注解还可以应用于表达式，就如在模式匹配中的@unchecked 注解（参见第 15 章）。这需要在表达式之后加上冒号（：），然后写上注解内容。依照语法构成来看，就好像注解被用作类型一样：

```
(e: @unchecked) match {
  // 非完备的样本（列表）
}
```

最后，注解还可以放在类型上。本章后续部分将会描述加注解的类型。

到目前为止，所见的注解都只是简单的 at 符号（@）跟着注解类。这种简单注解很普遍也很有用，但实际上注解具有更为丰富的一般形式：

$$@annot(exp_1, exp_2, ...)\ \{val\ name_1=const_1, ..., val\ name_n=const_n\}$$

annot 是指注解的类，这是所有的注解都必须包含的。exp 部分是注解的参数，类似于@deprecated 这样的注解是不需要任何参数的。你通常可以省略括号，但如果愿意的话也可以写成@deprecated()。而对于的确具有参数的注解来说，参数需要放在括号内，如@serial(1234)。

507

需要提供给注解的参数的具体形式取决于注解类。多数注解处理器只允许提供立即常量如 123 或 "hello"。不过只要类型检查可以通过，编译器本身支持任意的表达式。因此，有些注解类可以利用这一点以便让你做比如说引用作用域内的其他变量这样的事：

```
@cool val normal = "Hello"
@coolerThan(normal) val fonzy = "Heeyyy"
```

[1]译注：严重的错误。

一般语法内的 name = const 这种名值对形式可以用于更为复杂的、具有可选参数的注解。这些参数可选，并且可以按照任意次序指定。为了便于处理，等号右侧必须是常量。

25.3 标准注解

Scala 包含了若干个标准注解。它们是为了那些被广泛应用的，并足以将其放在语言规格说明书中的，但又不够基础到值得为其提供专属语法的特性而准备的。随着时间流逝，或许会有新的注解以同样的方式加入到标准中。

废弃

有些时候你可能会编写了宁愿没有写过的类或方法。尽管如此，一旦代码可用之后，其他人编写的代码就有可能会调用这些方法。因此，你不能简单的删除了事，因为这将导致其他人的代码编译失败。

废弃注解能够让你优雅地移除被证明出是错误的方法或类。如果你把方法或类标记为废弃，那么任何调用方法或类的人将收到废弃警告。他们最好能够注意到这个警告并更新他们的代码！经过了一段合适的时间之后，你感觉可以安全地假设所有理性的客户都已经停止了废弃类或方法的访问，就可以安全地删除这段代码。

要把方法标记为废弃，只要在方法之前加上@deprecated 即可。如：

```
@deprecated def bigMistake() =    //...
```

这个注解将在 Scala 代码访问该方法的时候让 Scala 编译器弹出废弃警告。

易变字段

并发编程不能很好地与共享的可变状态混用。为此，Scala 的并发重点关注于消息传递的支持，对共享的可变状态支持极少。细节参见第 30 章。

尽管如此，有些时候程序员仍然在并发编程中需要用到可变的状态。@volatile 注解对此提供了帮助。它可以通知编译器相关变量将被多个线程使用。实现这样的变量会使得它在读和写上变慢，但多线程的访问将表现得更具可预见性。

@volatile 关键字在不同平台上提供的保证也不同。不过，在 Java 平台上，它与你在 Java 代码中编写字段，并用 Java 的 volatile 修饰符对它做标记的行为是一致的。

二进制序列化

许多语言包含了二进制序列化的架构。序列化架构可以帮助你把对象变成字节流，反之亦然。如果你想要把对象保存到磁盘上或者通过网络发送，这会很有用。XML 有助于解决同一问题（参见第 26 章），但两者在速度、空间使用、灵活性以及可移植性方面有不同的侧重。

Scala 不具有自己的序列化框架。相反，你应该使用自己的基础平台上的框架。Scala 能够提供的是

为各种平台准备的三个注解。同样，Scala 的基于 Java 平台的编译器把这些注解以 Java 的方式作了解释（参见第 29 章）。

第一个注解指明类究竟是否可以序列化。多数类都可以序列化，但并非全部类。例如，socket 或 GUI 窗口句柄不能被序列化。默认情况下，类被认为是不可序列化的，你需要把`@serializable`注解添加到你希望序列化的类上。

第二个注解有助于处理随时间改变而改变的可序列化类。你可以通过添加如`@SerialVersionUID(1234)`这样的注解，把序号附加在类的当前版本上，这里 1234 应该用你选择的序号替代，而框架应该把这个数存在产生的字节流中。当你后来重载字节流并尝试把它转换为对象的时候，框架可以检查当前的类版本与字节流里的版本是否具有相同的版本号。如果想要对你的类做对于序列化不能相兼容的改变的话，你可以修改版本号，框架将自动拒绝加载类的旧实例。

509

最后，Scala 给不应被序列化的字段提供了`@transient`注解。如果你标记字段为`@transient`，那么框架就不应该在序列化该字段所在的对象时执行序列化。在对象被加载的时候，注解为`@transient`的字段也将根据其类型恢复为默认值。

自动的 get 和 set 方法

Scala 代码通常不需要为字段提供显式的 get 和 set 方法，因为 Scala 的字段访问和方法调用语法是混在一起的。但某些特定平台的框架需要 get 和 set 方法。为此，Scala 提供了`@scala.reflect.BeanProperty`注解。如果你把这个注解加到字段上，编译器就会为你自动产生 get 和 set 方法。如果你注解名为 crazy 的字段，那么 get 方法将被命名为 getCrazy，而 set 方法会被命名为 setCrazy。

产生的 get 和 set 方法只在完成编译之后才可用。因此，不能在同一时间编译的代码中调用注解字段的这些 get 和 set 方法。不过这在实际使用中不是问题，因为在 Scala 代码中你可以直接访问这些字段。这个特性只是为了支持那些需要有正规的 get 和 set 方法的架构，并且明显你不会在同一时间既编译框架又编译使用它的代码。

不检查

`@unchecked`注解由编译器在模式匹配的时候解释。它告诉编译器不要担心 match 表达式忽略了某些情况。细节参见 15.5 节。

25.4 小结

本章描述了你最应该了解的注解与平台无关的几方面知识。首先讲述了注解的语法，因为注解的使用远比定义新的注解更为寻常。其次演示了如何使用标准 Scala 编译器支持的若干注解，包括`@deprecated`、`@volatile`、`@serializable`、`@BeanProperty`及`@unchecked`。

510

第 29 章将提供更多特定于 Java 的注解信息。包括仅在 Java 中可用的注解，如何与基于 Java 的注解进行互操作，以及如何使用基于 Java 的机制在 Scala 中定义和处理注解。

511

第 26 章

使用 XML

本章将介绍 Scala 对 XML 的支持。在讨论一般意义上的半结构化数据之后，将展示 Scala 中操控 XML 的基础功能：如何构建 XML 字面量节点，如何通过文件保存及加载 XML，并且如何使用查询方法和模式匹配拆解 XML 节点。本章仅是对 XML 的概要介绍，但作为起步阶段是足够了。

26.1 半结构化数据

XML 是一种半结构化数据（semi-structured data）的形式。它比单纯的字符串更为结构化，因为它把数据内容组织成了树结构。尽管如此，单纯的 XML 的结构化程度仍然比不上编程语言的对象，因为它允许在标签之间存在自由格式的文本，并且它缺少类型系统[1]。

任何在你需要序列化程序数据以保存到文件或通过网络运输的时候，半结构化的数据都将非常有用。你无须把结构化的数据直接“降解”为字节，而只完成到半结构化数据之间的转换即可。然后你可以使用已有的库函数（library routines）实现半结构化数据与二进制数据之间的转换，从而节省了可以用来处理更重要问题的时间。

半结构化的数据有很多种格式，但 XML 是互联网上用得最广的。大多数操作系统上都有 XML 工具，大多数编程语言中也都有可用的 XML 库。其流行性不断自我强化。因 XML 流行性而开发的工具和库越多，软件工程师就越有可能选用 XML 格式。如果你要编写通过互联网通信的软件，那么迟早你会需要与使用 XML 的某些服务进行交互。

鉴于以上这些原因，Scala 包含了专门支持处理 XML 的工具。本章将展示 Scala 如何通过正规方法和模式匹配的方式处理结构化 XML 的支持。除了这些基本部件之外，本章还将在随后演示若干 XML 在 Scala 中的常见例子。

26.2 XML 概览

XML 的构成包含两种基本元素，文本和标签[2]。通常，文本是指任何字符序列。标签写法类似于 `<pod>`，由小于号、字母数字的标签名及大于号组成。包括开始标签和结束标签两种。结束标签除了在

[1]注：XML 也有类型系统，如 XML 结构定义（XML Schema），但它们不在本书的讨论范围之内。
[2]注：完整的说法更为复杂，但目前采用的说法足以概括 XML。

标签名之前多了斜杠外与开始标签一致，如：</pod>。

开始和结束标签必须互相匹配，就像一对括号。任何开始标签必须最终跟随同样标签名的结束标签。因此以下形式是不合法的：

```
//不合法的 XML
One <pod>, two <pod>, three <pod> zoo
```

更进一步说，任何两个匹配的标签之间的内容自身也必须是合法的 XML。两对匹配标签不能相互叠加：

```
//同样不合法
<pod>Three <peas> in the </pod></peas>
```

不过，你可以写成：

```
<pod>Three <peas></peas> in the </pod>
```

标签的这种匹配方式，使得 XML 以元素嵌套的方式结构化。每一对匹配的开始和结束标签组成了一个元素，元素可以嵌套在另一个元素中。在上面的例子中，<pod>Three <peas></peas> in the </pod>是一个元素，而<peas></peas>是内嵌在元素中的元素。

514

以上是 XML 的基础。还有两件事你应该知道：第一，对于开始标签紧跟匹配的结束标签的情况有一种简写方式，只要在标签的名称之后加上斜杠即可。这种标签包含了空的元素。使用这种空元素，前面的例子可以同样写成：

```
<pod>Three <peas/> in the </pod>
```

第二，开始标签可以有附加的属性。属性就是指 name=value 这种中间有等号的名值对。属性名本身是平凡的无结构文本，属性值则必须以双引号（""）或单引号（''）包围。举例如下：

```
<pod peas="3" strings="true"/>
```

26.3 XML 字面量

Scala 允许你在任何可以存在表达式的地方以字面量的形式键入 XML。只要简单地打上开始标签并连续编写 XML 内容即可。编译过程会进入 XML 输入模式并把内容读取为 XML，直到发现了匹配于前面开始标签的结束标签为止：

```
scala> <a>
         This is some XML.
         Here is a tag: <atag/>
       </a>
res0: scala.xml.Elem =
<a>
         This is some XML.
         Here is a tag: <atag></atag>
       </a>
```

以上表达式的结果类型为 Elem，说明它是有标签名（"a"），有内容（"This is some XML..."，等）的 XML 元素。其他的一些重要的 XML 类包括：

- Node 类，所有 XML 节点类的抽象超类。
- Text 类，只包含文本的节点。例如，<a>stuff的“stuff”部分是 Text 类的。
- NodeSeq 类，保存节点序列。XML 库中的许多方法，在你看来应该处理单个 Node 的，都是处理 NodeSeq 的。不过这些方法还是能够用在单个节点上，因为 Node 是从 NodeSeq 扩展来的。这可能听起来很怪，但能够很好地处理 XML。你可以把单个的 Node 看作是仅有一个元素的 NodeSeq。

你不仅可以一个字符一个字符地编写纯粹的 XML，还可以在 XML 字面量中使用花括号（{}）做转义符，执行 Scala 代码。举例如下：

```
scala> <a> {"hello"+", world"} </a>
res1: scala.xml.Elem = <a> hello, world </a>
```

转义括号可以包括任意的 Scala 内容，乃至更多的 XML 字面量。因此，随着内嵌级别的增加，代码可以在 XML 与普通 Scala 代码之间来回切换。举例如下：

```
scala> val yearMade = 1955
yearMade: Int = 1955

scala> <a> { if (yearMade < 2000) <old>{yearMade}</old>
             else xml.NodeSeq.Empty }
       </a>
res2: scala.xml.Elem =
<a> <old>1955</old>
</a>
```

如果花括号内的代码执行的结果是 XML 节点或节点序列的话，这些节点将依原样被直接插入。上面的例子中，如果 yearMade 小于 2 000，就被包裹在<old>标签中并被加入到<a>元素里。否则，就什么也不加。注意上面的例子中的“什么也不加”，以 XML 节点方式表示就是 xml.NodeSeq.Empty。

转义括号内的表达式并非一定要运行出 XML 节点才行。它可以返回任何 Scala 值。这种情况下，结果将被转变为字符串，然后以文本节点的形式插入：

```
scala> <a> {3 + 4} </a>
res3: scala.xml.Elem = <a> 7 </a>
```


如果以打印的方式输出返回节点，那么文本中所有的<，>以及&字符将被转义：

```
scala> <a> {"</a>potential security hole<a>"} </a>
res4: scala.xml.Elem = <a> &lt;/a&gt;potential security
hole&lt;a&gt; </a>
```

相反，如果使用低级字符串操作创建 XML，你或许会陷入以下的圈套之中：

```
scala> "<a>" + "</a>potential security hole<a>" + "</a>"
res5: java.lang.String = <a></a>potential security
hole<a></a>
```

这里发生的问题是用户提供的字符串包含了自己的 XML 标签，例子中是和<a>。这种行为可

能会让代码作者大惊失色，因为这使得用户可以在允许范围（<a>元素内部）之外修改结果 XML 树。你可以通过始终使用 XML 字面量，而不是字符串添加的方式构建 XML，来杜绝所有的这类错误。

26.4 序列化

现在看到的是已经足够支持用 Scala 的 XML 编写序列化器的第一部分：从内部数据结构到 XML 的转换。目前要用的只有 XML 字面量和转义括号。

举个例子，假设你正在实现数据库以便记录广泛收集的古董可口可乐温度计。你或许会构建如下的内部类以目录形式保存条目：

```
abstract class CCTherm {
  val description: String
  val yearMade: Int
  val dateObtained: String
  val bookPrice: Int        //美分
  val purchasePrice: Int    //美分
  val condition: Int        //1~10

  override def toString = description
}
```

这是一个简单明了的数据类，握有各种各样的信息，如温度计的生产时间，获得时间，以及购买费用。

要把这个类的实例转换为 XML，只要添加使用了 XML 字面量和转义括号的 toXML 方法，如下：

```
abstract class CCTherm {
  ...
  def toXML =
    <cctherm>
      <description>{description}</description>
      <yearMade>{yearMade}</yearMade>
      <dateObtained>{dateObtained}</dateObtained>
      <bookPrice>{bookPrice}</bookPrice>
      <purchasePrice>{purchasePrice}</purchasePrice>
      <condition>{condition}</condition>
    </cctherm>
}
```

实际操作中的方法执行效果如下：

```
scala> val therm = new CCTherm {
         val description = "hot dog #5"
         val yearMade = 1952
         val dateObtained = "March 14, 2006"
         val bookPrice = 2199
         val purchasePrice = 500
         val condition = 9
       }
therm: CCTherm = hot dog #5
```

```
scala> therm.toXML
res6: scala.xml.Elem =
<cctherm>
        <description>hot dog #5</description>
        <yearMade>1952</yearMade>
        <dateObtained>March 14, 2006</dateObtained>
        <bookPrice>2199</bookPrice>
        <purchasePrice>500</purchasePrice>
        <condition>9</condition>
        </cctherm>
```


注意

尽管 CCTherm 是抽象类，上面例子中的表达式“new CCTherm”仍能正常工作，因为这样的语法实际上实例化了 CCTherm 的匿名子类。匿名类详见 20.5 节。

顺便说一下，如果你想要在 XML 文本中包含花括号（‘{’或‘}’），而不是用于 Scala 代码的转义，只要在一行中写两次花括号即可：

```
scala> <a> {{{{brace yourself!}}}} </a>
res7: scala.xml.Elem = <a> {{brace yourself!}} </a>
```

26.5 拆解 XML

在 XML 类可用的许多方法中，你需要注意有三个很特殊。它们可以让你在不用考虑太多 XML 在 Scala 中的表现细节的情况下执行 XML 的拆解。这些方法都基于处理 XML 的 XPath 语言。就像通常在 Scala 中的那样，你可以直接在 Scala 代码中使用它们而不必调用外部工具。

抽取文本。通过对 XML 节点调用 text 方法你可以取回节点内去除了所有元素标签之后的全部文本。

```
scala> <a>Sounds <tag/> good</a>.text
res8: String = Sounds  good
```

所有的编码字符会自动解码：

```
scala> <a> input ---&gt; output </a>.text
res9: String =  input ---> output
```

抽取子元素。如果你想要通过标签名找到子元素，只要调用 \ 加标签名即可：

```
scala> <a><b><c>hello</c></b></a>\ "b"
res10: scala.xml.NodeSeq = <b><c>hello</c></b>
```

你还可以使用 \\ 替代 \ ，执行“深度搜索”寻找子元素等内容：

```
scala> <a><b><c>hello</c></b></a> \ "c"
res11: scala.xml.NodeSeq =

scala> <a><b><c>hello</c></b></a> \\ "c"
res12: scala.xml.NodeSeq = <c>hello</c>

scala> <a><b><c>hello</c></b></a> \ "a"
```

```
res13: scala.xml.NodeSeq =

scala> <a><b><c>hello</c></b></a>\\ "a"
res14: scala.xml.NodeSeq = <a><b><c>hello</c></b></a>
```

注意

Scala使用了 \ 和 \\ 代替XPath的 / 和 //。理由是 // 是Scala注释的开始！因此必须用其他的符号代替，而用另一种斜杠可以顺利工作。

抽取属性。你可以同样使用 \ 和 \\ 方法抽取标签属性。只要在属性名之前加上at符号（@）即可：

```
scala> val joe = <employee
          name="Joe"
          rank="code monkey"
          serial="123"/>
joe: scala.xml.Elem = <employee rank="code monkey" name="Joe"
 serial="123"></employee>

scala>  joe \ "@name"
res15: scala.xml.NodeSeq = Joe

scala>  joe \ "@serial"
res16: scala.xml.NodeSeq = 123
```

26.6 反序列化

520 通过使用前面的XML拆解方法，你现在可以编写序列化器的对偶部分，从XML转回你的内部数据结构的解析器了。例如，可以使用下列代码完成CCTerm实例的解析：

```
def fromXML(node: scala.xml.Node): CCTherm =
  new CCTherm {
    val description    = (node \ "description").text
    val yearMade       = (node \ "yearMade").text.toInt
    val dateObtained   = (node \ "dateObtained").text
    val bookPrice      = (node \ "bookPrice").text.toInt
    val purchasePrice  = (node \ "purchasePrice").text.toInt
    val condition      = (node \ "condition").text.toInt
}
```

这段代码搜索输入的名为node的XML节点，找到定义CCTherm所需的六块数据，就是用.text方法抽取的并且保持原样的文本。方法在实际使用中的效果如下：

```
scala> val node = therm.toXML
node: scala.xml.Elem =
<cctherm>
          <description>hot dog #5</description>
          <yearMade>1952</yearMade>
          <dateObtained>March 14, 2006</dateObtained>
          <bookPrice>2199</bookPrice>
          <purchasePrice>500</purchasePrice>
          <condition>9</condition>
```

```
        </cctherm>

scala> fromXML(node)
res15: CCTherm = hot dog #5
```

26.7 加载和保存

数据序列化器还有最后一部分要编写：XML 与字节流之间的转换。这最后的部分是最容易的，因为可以由库函数帮你完成。你只要对正确的数据调用正确的函数即可。

要把 XML 转换为字符串，仅须调用 toString 方法。而正是因为存在了可用的 toString 方法才使得你可以在 Scala 的 shell 中能够完成 XML 的实验。不过，最好还能使用库函数直接把它们转换为字节。那样的话，结果 XML 就可以包含指定所用字符编码的指令。如果你自己实现从字符串到字节的编码，那就需要你亲自过问字符编码的问题了。

要把 XML 转换为字节文件，你可以使用 XML.saveFull 命令。需要选择的部分包括：文件名、要保存的节点及字符编码。第四个参数是决定是否在文件头写上包含字符编码的 XML 声明。第五个参数是这个 XML 的“文档类型”，这个主题超出了本章的范围。你可以指定为 null，任由文档类型为未指定即可：

```
scala.xml.XML.saveFull("therm1.xml", node, "UTF-8", true, null)
```

运行了以上命令之后，结果文件 therm1.xml 看上去为如下形式：

```
<?xml version='1.0' encoding='UTF-8'?>
<cctherm>
              <description>hot dog #5</description>
              <yearMade>1952</yearMade>
              <dateObtained>March 14, 2006</dateObtained>
              <bookPrice>2199</bookPrice>
              <purchasePrice>500</purchasePrice>
              <condition>9</condition>
            </cctherm>
```

加载比保存更简单，因为文件包含了所有加载器需要知道的东西。只要调用 XML.loadFile 和文件名即可：

```
scala> val loadnode = xml.XML.loadFile("therm1.xml")
loadnode: scala.xml.Elem =
<cctherm>
      <description>hot dog #5</description>
      <yearMade>1952</yearMade>
      <dateObtained>March 14, 2006</dateObtained>
      <bookPrice>2199</bookPrice>
      <purchasePrice>500</purchasePrice>
      <condition>9</condition>
    </cctherm>

scala> fromXML(loadnode)
res14: CCTherm = hot dog #5
```

这些就是你需要知道的基本方法了。加载和保存方法有许多的变化，包括针对各种读写对象（reader / writer）及输入输出流（input / output stream）的读写方法。

26.8 XML的模式匹配

目前为止你已经看到了如何使用text和类似于Xpath的方法，\ 和 \\ 切分XML。在你确切知道待拆解的是何种XML结构的情况下，这些方法很好用。但有些时候，一个XML可能有几种结构。或许因为数据内含有多种记录，例如你扩展了温度计收集以包含钟表和三明治盘子。或许只是想要跳过标签之间的空白部分。不管理由是什么，你都可以使用模式匹配对这些情况进行筛选。

XML模式看上去很像XML字面量。主要差别在于如果你插入转义括号{}，那么{}内的代码不是表达式而是模式。{}内嵌的模式可以使用所有的Scala模式语言，包括绑定新变量，执行类型检查，以及使用_和_*忽略内容。举例如下：

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents}</a> => "It's an a: "+ contents
    case <b>{contents}</b> => "It's a b: "+ contents
    case _ => "It's something else."
}
```

函数有三种情况的模式匹配。第一种情况寻找内容为单个子节点的<a>元素。它把这些内容绑定到名为contents的变量上，然后运行箭头（=>）右侧的代码。第二种情况几乎完全相同，只是把寻找的元素用<b>替代了<a>，第三种情况匹配除前两种之外所有的情况。函数的使用效果如下：

```
scala> proc(<a>apple</a>)
res16: String = It's an a: apple

scala> proc(<b>banana</b>)
res17: String = It's a b: banana scala> proc(<c>cherry</c>)

scala> proc(<c>cherry</c>)
res18: String = It's something else.
```

很可能这个函数不是你想要的，因为它只能寻找<a>或<b>中单个子节点的内容。因此它将不能匹配以下的情况：

```
scala> proc(<a>a <em>red</em> apple</a>)
res19: String = It's something else.

scala> proc(<a/>)
res20: String = It's something else.
```

如果你想要函数能够匹配这些情况，可以执行对节点序列而不是单个节点的匹配。“任意序列”XML节点的模式写做‘_*’。外表上，这个序列很像通配模式（_）后跟正则表达式风格的克莱尼星号（*，Kleene star）。以下是更新后的函数，它以子元素序列的匹配取代了对单个子元素的匹配：

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents @ _*}</a> => "It's an a: "+ contents
```

```
    case <b>{contents @ _*}</b> => "It's a b: "+ contents
    case _ => "It's something else."
}
```

可以注意到_*的结果是通过使用 15.2 节描述过的@模式被绑定到了 contents 变量。以下是实际使用中新版本函数的执行效果：

```
scala> proc(<a>a <em>red</em> apple</a>)
res21: String = It's an a: ArrayBuffer(a ,
  <em>red</em>,  apple)
scala> proc(<a/>)
res22: String = It's an a: Array()
```

最后要说的就是，请注意 XML 模式能够很好地与 for 表达式一起工作，通过这种方式你可以迭代遍历 XML 树结构的某些局部而忽略其他部分。例如，假设你希望在下列的 XML 结构中跳过记录之间的空白：

524

```
val catalog =
  <catalog>
    <cctherm>
      <description>hot dog #5</description>
      <yearMade>1952</yearMade>
      <dateObtained>March 14, 2006</dateObtained>
      <bookPrice>2199</bookPrice>
      <purchasePrice>500</purchasePrice>
      <condition>9</condition>
    </cctherm>
    <cctherm>
      <description>Sprite Boy</description>
      <yearMade>1964</yearMade>
      <dateObtained>April 28, 2003</dateObtained>
      <bookPrice>1695</bookPrice>
      <purchasePrice>595</purchasePrice>
      <condition>5</condition>
    </cctherm>
  </catalog>
```

看起来，好似<catalog>元素中有两个节点。但实际上，有五个。还有前面的，后面的，以及夹在两个元素之间的空白！如果你没有注意到这些空白，或许会错误地以如下方式处理温度计的记录：

```
catalog match {
  case <catalog>{therms @ _*}</catalog> =>
    for (therm <- therms)
      println("processing: "+
              (therm \ "description").text)
}


processing:
processing: hot dog #5
processing:
processing: Sprite Boy
processing:
```

525

可以发现所有的代码行都在尝试把空白当作真正的温度计记录处理。而你实际想要的是忽略空白并只处理那些在<cctherm>元素之内的子节点。你可以使用模式<cctherm>{_*}</cctherm>描述这个子集，并限制 for 表达式枚举能够匹配这个模式的条目：

```
catalog match {
  case <catalog>{therms @ _*}</catalog> =>
    for (therm @ <cctherm>{_*}</cctherm> <-  therms)
      println("processing: "+
              (therm \ "description").text)
}

processing: hot dog #5
processing: Sprite Boy
```

26.9 小结

本章只谈到了 XML 使用的皮毛。还有很多值得学习的扩展、库和工具，其中的一些是为 Scala 定制的，另一些是为 Java 制作的但也适用于 Scala，还有一些则是语言原生的。你应该能够从本章了解到如何把半结构化数据应用于信息交换，以及如何通过 Scala 的 XML 支持访问半结构化的数据。

第 27 章

使用对象的模块化编程

第 1 章里，我们曾经声称 Scala 是可伸展的语言。理由之一就是，它是一种既适合于构造小程序，也适合于构造大程序的技术。而这本书里直到现在，我们还是主要关注于小规模编程：设计和实现可以用于构造大程序的小片段[1]。反过来就是大规模编程：把小片段组织并汇总为更大的程序、应用或系统。我们在第 13 章讨论包和访问修饰符的时候曾经接触过这个题目。简言之，包和访问修饰符能让你把包用作模块来组织大程序。这里的模块是指具有良好定义的接口和隐藏实现的“小程序片段”。

尽管把程序以包的形式分割已经非常有助于编程，但这种帮助仍然很有限。因为它不能提供对抽象的支持。你不能在同一个程序中以两种不同的方式重新配置包，也不能在包之间引入继承。包总是包含了其内容的明确列表，而且除非你改变代码，否则列表是固定的。

本章，我们将讨论如何利用 Scala 的面向对象特性使得程序更为模块化。我们首先会说明如何把简单的单例对象用作模块，然后再说明如何把特质和类用作模块上的抽象。这种抽象可以在多个模块中重新配置，乃至在同样的程序中做多次的配置。最后，我们会说明一种用特质把模块分割到多个文件中的实用技术。

27.1 问题

随着程序规模的增大，以模块化的方式对其加以组织变得尤为重要。首先，能够编译不同的模块以分别建造系统的方式来帮助不同的小组互不干扰的工作。其次，允许模块实现的插拔和替换是有益的，因为这使得系统的不同配置可以应用于不同的环境，诸如程序员桌面的单元测试、集成测试、分阶段（持续集成）及部署。

例如，假设你需要实现一个使用数据库和消息服务的应用程序。在编写代码的时候，你或许希望能够在桌面上运行使用了仿制版的数据库和消息服务的单元测试，它们模拟的这些服务足以应付测试而无须与共享资源进行网络通信。集成测试中，你或许希望使用仿制的消息服务但却要求真实的开发者数据库。分阶段期间及发布过程中，你的公司会希望使用真实版本的数据库和消息服务。

任何致力于这种模块性需要的技巧都需要至少满足以下几点基本要求。首先，应该有一个能够很好地分离接口和实现的模块化构造。其次，应该有方式可以替换具有相同接口的模块，但不用改变或重新

[1]注：术语来自于 DeRemer 等人的“Programming-in-the-large versus programming-in-the-small”。[DeR75]

编译依赖被替换模块的模块。最后，应该有方式可以把模块衔接在一起。这种衔接的任务可以认为是系统配置。

解决这个问题的方法之一是依赖注入，这是一种通过框架（如企业 Java 社区较为流行的 Spring 或 Guice）支持在 Java 平台之上的技术[2]。比方说 Spring，本质上是让你能够用 Java 的接口表达模块的接口，以及用 Java 的类实现模块。你可以指定模块之间的依赖关系并通过外部 XML 配置文件把应用程序“衔接”在一起。尽管你也可以在 Scala 里使用 Spring，从而以 Spring 的方式让 Scala 程序能够达到系统级的模块性，但 Scala 也提供给你了通过语言自身实现的备选方案。本章后续部分，我们将说明如何把对象用作模块以获得期望的“大规模”的模块性而无须用到外部框架。

528

27.2 食谱应用

设想你正在建造一个可以让用户管理食谱的企业 Web 应用。你想要把软件划分为不同的层次，包括领域层及应用层。在领域层，你会定义领域对象，用来保存业务概念和规则并包装将被持久化到外部关系数据库的状态信息。在应用层（包含了用户接口层），你会提供以客户服务形式组织的 API。应用层会通过协调任务及派发工作给领域层对象的方式实现这些服务[3]。

同时设想你需要让每一层都可以插入某些对象真实的或仿制的版本，这样你就可以更容易地为应用编写单元测试。为了达到这个目标，你可以把想要仿制的对象当成模块。Scala 里，对象不一定就是“小”东西，像模块这样的“大”东西也不需要使用其他种类的构造。Scala 是可伸展的语言的原因之一就是，同样的构造既可以使用在大结构中，也可以使用在小结构中。例如，既然代表关系数据库的对象是你在领域层想要仿制的“东西”之一，那么你就应该把它做成模块。在应用层，你应该把“数据库浏览器”对象当作模块。数据库将保留收集到的所有食谱。浏览器可以帮助搜索和浏览数据库，比方说，查找包含了你手头上食材的所有食谱。

要做的第一件事就是对食物和食谱建模。为了简化问题，食物仅包含名称，如清单 27.1 所示。食谱也只有名称、食材列表及一些做法，如清单 27.2 所示。

529

```
package org.stairwaybook.recipe

abstract class Food(val name: String) {
  override def toString = name
}
```

清单 27.1 简化的 Food 实体类

```
package org.stairwaybook.recipe

class Recipe(
  val name: String,
  val ingredients: List[Food],
  val instructions: String
) {
```

[2]注：Fowler，“Inversion of control containers and the dependency injection pattern”. [Fow04]

[3]注：这些层的命名方式来自于 Evans，“Domain-Driven Design”。[Eva03]

```
  override def toString = name
}
```

清单 27.2 简化的 Recipe 实体类

清单 27.1 和清单 27.2 所示的 Food 和 Recipe 类代表了将要被持久化到数据库的实体[4]。清单 27.3 演示了这两个类的一些单例对象，可以在编写测试代码的时候使用：

```
package org.stairwaybook.recipe

object Apple extends Food("Apple")
object Orange extends Food("Orange")
object Cream extends Food("Cream")
object Sugar extends Food("Sugar")

object FruitSalad extends Recipe(
  "fruit salad",
  List(Apple, Orange, Cream, Sugar),
  "Stir it all together."
)
```

清单 27.3 可以用于测试的 Food 和 Recipe 例子

Scala 可以把对象用作模块，所以你可以通过创建测试期间用于仿制数据库和浏览器模块的两个单例对象开始模块化程序。因为是仿制的，所以数据库模块只用简单的内存列表作支撑。这些对象的实现参见清单 27.4。数据库和浏览器使用方式如下：

```
package org.stairwaybook.recipe

object SimpleDatabase {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def foodNamed(name: String): Option[Food] =
    allFoods.find(_.name == name)

  def allRecipes: List[Recipe] = List(FruitSalad)
}

object SimpleBrowser {
  def recipesUsing(food: Food) =
    SimpleDatabase.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))
}
```

清单 27.4 仿制的数据库和浏览器模块

```
scala> val apple = SimpleDatabase.foodNamed("Apple").get
apple: Food = Apple

scala> SimpleBrowser.recipesUsing(apple)
```

[4]注：这些实体类都已经被简化过，以便样例代码不要过多牵扯真实世界中的细节。抛开这些不谈，把这些类转换为可以被例如 Hibernate 或 Java 持久化架构持久化的实体，仅需要很少的改变，如添加私有的 Long id 字段和无参构造器，把 scala.reflect.BeanProperty 注解加到字段上，通过注解或独立的 XML 文件指定合适的映射方法，等等。

```
res0: List[Recipe] = List(fruit salad)
```

为了增加一点儿趣味性，假设数据库对食物进行了分类。要实现这一点，你可以在数据库中添加FoodCategory类（表示食物类别）及所有（食物）类别的列表，参见清单27.5。注意后面的例子中的private关键字，对于类的实现来说很有用，对于模块的实现来说也同样有用。标记为private的条目是模块实现的一部分，从而很容易在不影响其他模块的情况下进行改动。

在这一点上，我们还可以加入更多的机制，不过只要你明白意思就行了。程序可以被分切成单例对象，你可以认为它是模块。这不是什么重要消息，不过当你考虑抽象概念的时候，它会变得非常有用。

531

```
package org.stairwaybook.recipe

object SimpleDatabase {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def foodNamed(name: String): Option[Food] =
    allFoods.find(_.name == name)

  def allRecipes: List[Recipe] = List(FruitSalad)

  case class FoodCategory(name: String, foods: List[Food])

  private var categories = List(
    FoodCategory("fruits", List(Apple, Orange)),
    FoodCategory("misc", List(Cream, Sugar)))

  def allCategories = categories
}

object SimpleBrowser {
  def recipesUsing(food: Food) =
    SimpleDatabase.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))

  def displayCategory(category:  SimpleDatabase.FoodCategory) {
    println(category)
  }
}
```

清单27.5 添加了类别的数据库和浏览器模块

27.3 抽象概念

尽管目前为止看到的例子的确把你的程序划分为分离的数据库和浏览器模块，但这个设计还不够“模块化”。问题在于浏览器模块实质上是“硬连接”到数据库模块的：

```
SimpleDatabase.allRecipes.filter(recipe => ...
```

因为SimpleBrowser模块是通过名称提到SimpleDatabase模块，所以你不能在不修改和重新编译浏览器模块的前提下插入数据库模块的不同实现。而且，尽管SimpleDatabase模块没有到

532

SimpleBrowser 模块的硬连接[5]，但也没有什么清晰的方式，比如说让用户接口层配置来使用不同的浏览器模块实现。

不过，在把这些模块变得更具可插拔性的时候，很重要的一点是避免代码重复，因为似乎有大量的代码可以在相同模块的不同实现之间共享。举例来说，假设你想要创建能访问这些数据库的单独的浏览器，你可能会希望为每个实例重用浏览器代码。因为这些浏览器之间唯一的差别仅在于它们引用的是哪个数据库。除了数据库实现之外，剩下的代码每个字符都是可重用的。如何让程序的重复代码被压缩到最少？如何让代码变得可重新配置，以便于让你配置它使用哪个数据库实现？

答案似曾相识：如果模块是对象，那么模块的模板就是类。就好像类描述了所有实例的公共部分一样，类也可以描述模块中它所有可能的配置中的公共部分。

因此浏览器定义变成了类，不再是对象，所用的数据库被指定为类的抽象成员，参见清单 27.6。数据库也变成了类，包括所有数据库中公共的部分，并声明了数据库必须定义的缺失部分。在这里，所有的数据库模块必须定义的方法有 allFoods，allRecipes，以及 allCategories，但由于它们可以任意定义，因此这些方法必须在 Database 类中保留为抽象的。相反，foodNamed 方法可以定义在抽象 Database 类里，参见清单 27.7：

```
abstract class Browser {
  val database: Database

  def recipesUsing(food: Food) =
    database.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))

  def displayCategory(category:  database.FoodCategory) {
    println(category)
  }
}
```

清单 27.6　带抽象数据库 val 字段的 Browser 类

```
abstract class Database {
  def allFoods: List[Food]
  def allRecipes: List[Recipe]

  def foodNamed(name: String) =
    allFoods.find(f => f.name == name)

  case class FoodCategory(name: String, foods: List[Food])
  def allCategories: List[FoodCategory]
}
```

清单 27.7　带抽象方法的 Database 类

SimpleDatabase 对象必须更新以继承抽象的 Database 类，参见清单 27.8：

[5]注：这很好，因为这种架构的每一层只应该依赖它下面的层。

```
object SimpleDatabase extends Database {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def allRecipes: List[Recipe] = List(FruitSalad)

  private var categories = List(
    FoodCategory("fruits", List(Apple, Orange)),
    FoodCategory("misc", List(Cream, Sugar)))

  def allCategories = categories
}
```

清单 27.8 Database 的子类对象 SimpleDatabase

然后，通过实例化 Browser 类并指定使用的数据库，创建特定的浏览器模块，参见清单 27.9。

你仍然可以像以前那样使用这些更具可插拔性的模块：

```
scala> val apple = SimpleDatabase.foodNamed("Apple").get
apple: Food = Apple
```



```
object SimpleBrowser extends Browser {
  val database = SimpleDatabase
}
```

清单 27.9 Browser 子类对象 SimpleBrowser

```
scala> SimpleBrowser.recipesUsing(apple)
res1: List[Recipe] = List(fruit salad)
```

不过现在你可以创建第二个仿制数据库，并与相同的浏览器类一起工作，参见清单 27.10：

```
object StudentDatabase extends Database {
  object FrozenFood extends Food("FrozenFood")

  object HeatItUp extends Recipe(
    "heat it up",
    List(FrozenFood),
    "Microwave the 'food' for 10 minutes.")

  def allFoods = List(FrozenFood)
  def allRecipes = List(HeatItUp)
  def allCategories = List(
    FoodCategory("edible", List(FrozenFood)))
}

object StudentBrowser extends Browser {
  val database = StudentDatabase
}
```

清单 27.10 学生数据库与浏览器

27.4 把模块拆分为特质

模块常常过于庞大，而不适于放在单个文件中。如果发生这种情况，你可以使用特质把模块拆分为

多个文件。例如，假设你想要把执行分类操作的代码移到 Database 类所在文件以外成为独立的文件。你可以为这段代码创建一个特质，参见清单 27.11。

```
trait FoodCategories {
  case class FoodCategory(name: String, foods: List[Food])
  def allCategories: List[FoodCategory]
}
```

清单 27.11 食品分类的特质

现在 Database 类可以混入 FoodCategories 特质而无须自己定义 FoodCategory 和 allCategories，参见清单 27.12：

```
abstract class Database extends FoodCategories {
  def allFoods: List[Food]
  def allRecipes: List[Recipe]
  def foodNamed(name: String) =
    allFoods.find(f => f.name == name)
}
```

清单 27.12 混入了 FoodCategories 特质的 Database

继续这样做，你可以尝试把 SimpleDatabase 划分为两个特质，一个是食物，另一个是食谱。这可以让你能够定义如下的 SimpleDatabase，参见清单 27.13：

```
object SimpleDatabase extends Database
    with SimpleFoods with SimpleRecipes
```

清单 27.13 仅由混入组成的 SimpleDatabase 对象

SimpleFoods 特质定义可以参见清单 27.14：

```
trait SimpleFoods {
  object Pear extends Food("Pear")
  def allFoods = List(Apple, Pear)
  def allCategories = Nil
}
```

清单 27.14 SimpleFoods 特质

目前为止一切都好，但不幸的是，如果你尝试以如下方式定义 SimpleRecipes 特质，问题就发生了：

```
trait SimpleRecipes { //编译失败
  object FruitSalad extends Recipe(
    "fruit salad",
    List(Apple, Pear),   //啊哦……
    "Mix it all together."
  )
  def allRecipes = List(FruitSalad)
}
```

这里的问题是 Pear 没有处在使用它的特质中，所以超出了作用范围。编译器不知道 SimpleRecipes 只会与 SimpleFoods 混搭在一起。

不过有一个方法可以让你向编译器说明这点。Scala 对于恰好是这样的情况提供了自身类型(self type)。从技术角度上看，自身类型是在类中提到 this 时，对于 this 的假设性类型。从实用角度上看，自身类型指定了对于特质能够混入的具体类的需求。如果你的特质仅用于混入另一个或几个特质，那么可以指定那些假设性的特质。在当前的例子中，指定自身类型为 SimpleFoods 已经足够，参见清单 27.15：

```
trait SimpleRecipes {
  this: SimpleFoods =>

  object FruitSalad extends Recipe(
    "fruit salad",
    List(Apple, Pear),    //现在 Pear 在作用域内了
    "Mix it all together."
  )
  def allRecipes = List(FruitSalad)
}
```

清单 27.15 带有自身类型的 SimpleRecipes

有了新的自身类型，Pear 现在可以用了。Pear 的引用隐含地被认为是 this.Pear。这很安全，因为任何混入了 SimpleRecipes 的具体类都必须同时是 SimpleFoods 的子类型，也就是说 Pear 会是它的成员。抽象子类和特质不必遵守这个限制，但因为它们不能用 new 实例化，所以不存在 this.Pear 引用失败的风险。

537

27.5 运行期链接

Scala 模块的最后一个特性很值得强调：它们可以在运行时链接在一起，并且你还可以根据运行时的计算决定哪个模块将链接到哪个。例如，清单 27.16 展示了可以在运行时选择数据库并打印输出所有苹果食谱的小程序：

```
object GotApples {
  def main(args: Array[String]) {
    val db: Database =
      if(args(0) == "student")
        StudentDatabase
      else
        SimpleDatabase

    object browser extends Browser {
      val database = db
    }

    val apple = SimpleDatabase.foodNamed("Apple").get

    for(recipe <- browser.recipesUsing(apple))
      println(recipe)
  }
}
```

清单27.16 动态选择模块实现的应用程序

现在，如果你使用简单数据库，将能发现水果色拉的食谱；如果你使用学生数据库，会发现根本没有用得到苹果的食谱：

```
$ scala GotApples simple
fruit salad
$ scala GotApples student
$
```


用 Scala 代码做配置

你或许会好奇是否你又退回到了本章原始例子中的硬链接问题，因为清单 27.16 里的 GotApples 对象包含了 StudentDatabase 和 SimpleDatabase 的硬链接。这里的不同在于硬链接是处于可被替换的文件中的。

任何模块应用都需要以某种方式指定特殊情况下用到的实际模块实现。这种“配置”应用的活动将必然关系到具体模块实现的命名。例如，在 Spring 应用中，你通过在外部 XML 文件中命名实现来做配置。在 Scala 中，你可以通过 Scala 代码本身做配置。对于配置来说，使用 Scala 源代码与 XML 相比的好处之一是，通过 Scala 编译器运行配置文件的过程，会在它实际使用之前发现所有的拼写错误。

27.6 跟踪模块实例

尽管使用的是相同的代码，上一节创建的不同的浏览器和数据库模块仍然是分离的模块。这意味着每个模块都有自己的内容，包括内嵌类。比方说，SimpleDatabase 里的 FoodCategory 就与 StudentDatabase 里的 FoodCategory 不是同一个类！

```
scala> val category = StudentDatabase.allCategories.head
category: StudentDatabase.FoodCategory =
FoodCategory(edible,List(FrozenFood))

scala> SimpleBrowser.displayCategory(category)
<console> :12: error: type mismatch;
found     : StudentDatabase.FoodCategory
required  : SimpleBrowser.database.FoodCategory
       SimpleBrowser.displayCategory(category)
                                     ^
```


如果你更喜欢让所有的 FoodCategory 都相同，可以把 FoodCategory 定义移到类或特质之外。选择权在你，不过照之前的写法，每个 Database 类都有属于自己的、唯一的 FoodCategory 类。

上面例子中的两个 FoodCategory 类的确是不同的，所以编译器弹出错误信息的做法很正确。虽然如此，但有些时候你或许会碰到这样的情况！两个类型虽然相同但编译器却不能鉴别。你会看到编译器弹出错误信息说两个类型不同，尽管你作为程序员知道它们是完全一致的。

这种情况下，你可以用单例类型解决问题。例如，在 GotApples 程序里，类型检查器不知道 db 和 browser.database 是相同的。所以如果你尝试在两个对象之间传递分类信息的话，会引起类型错误：

```
object GotApples {
  //同样的定义……

  for (category <- db.allCategories)
    browser.displayCategory(category)

  // ...
}

GotApples2.scala:14: error: type mismatch;
 found   : db.FoodCategory
 required: browser.database.FoodCategory
        browser.displayCategory(category)
                                ^
one error found
```

要避免这个错误，你需要通知类型检查器它们是同一个对象。可以通过改变 browser.database 的定义实现，参见清单 27.17：

```
object browser extends Browser {
  val database: db.type = db
}
```

清单 27.17 使用单例类型

这个定义与之前的相同，只是 database 的类型很古怪，db.type。结尾的“.type”表示它是单例类型。单例类型极其特殊，只保存一个对象，在这里，是 db 指向的那个对象。通常这样的类型过于特殊没有什么用处，这也是为什么编译器很不希望自动插入它们的原因。但在这里，单例类型可以让编译器知道 db 和 browser.database 是同样的对象，这些信息足以消除类型错误了。

27.7 小结

本章展示了如何把 Scala 的对象用作模块。除了能简化静态模块之外，这种方式还能带给你创建抽象的、可重新配置模块的大量方法。实际上还有比已经看到的更多的抽象技术，因为任何可以应用在类上的东西同样也可以应用于实现模块的类上。同以往一样，你能用到什么程度，完全取决于个人喜好。

模块是大规模编程的部分，因此很难做实验。你需要大程序来体会其间的差异。不管怎么说，读过本章之后，你知道了当需要以模块化风格编程时须考虑哪种相应的 Scala 特性。在你写大程序的时候可以想想这些技术，当你在其他人的代码中看到它们时，也要能够认出来这些编码模式。

第 28 章

对象相等性

在编程活动中，比较两个值是否相等的需求无处不在。同时，这也比第一眼看上去要更具欺骗性。本章详细介绍对象相等性，并给出一些建议，供你在设计自己的相等性检测时参考。

28.1 Scala 中的相等性

正如在 11.2 节提到的，Scala 对相等的定义与 Java 不同。Java 有两种相等性比较：==操作符，对值类型而言这是自然的相等性，而对引用类型而言则是对象一致性；以及 equals 方法，是（用户定义的）引用类型的规约相等性。这样的约定俗成是有问题的，因为==符号更自然，但并不总是对应到自然意义上的相等性。Java 编程中，对初学者而言一个常见的陷阱是在该用 equals 的地方用==来比较对象。举例来说，在 Java 中，用“x == y”比较两个字符串 x 和 y，即便 x 和 y 拥有完全相同的字符和顺序，得到 false 也不奇怪。

Scala 也有一个相等性判断方法用来表示对象一致，不过用得并不多。此类相等性判断，写作“x eq y”，当 x 和 y 引用同一个对象时为 true。在 Scala 中，==相等性判断被用来表示每个类型“自然的”相等性。对值类型而言，==是对值的比较，和 Java 一样。而对于引用类型，==在 Scala 中相当于 equals。你可以重写新类型的 equals 方法从而重新定义==的行为，这个方法总是会从 Any 类继承下来。继承的 equals 方法除非被重写，默认是像 Java 那样判断对象是否一致。因此 equals 方法（以及==）默认和 eq 是一样的，不过你可以通过在你定义的类中重写 equals 方法的方式来改变其行为。我们没法直接重写==，因为它在 Any 类中定义为 final 方法。也就是说，Scala 对待==就如同它在 Any 类中被定义为如下的样子：

```
final def == (that: Any): Boolean =
  if(null eq this) {null eq that} else {this equals that}
```

28.2 编写相等性方法

我们该如何定义 equals 方法呢？我们发现在面向对象的语言中编写一个正确的相等性方法出人意料的困难。事实上，在研究了大量的 Java 代码后，2007 年一篇论文的作者们得出了这样的结论：几

乎所有的 equals 方法实现都有问题[1]。

这个问题很严重，因为很多其他代码逻辑都以相等性判断为基础。比如，如果一个类型 C 的相等性方法有问题，可能意味着你无法很有把握地将一个类型 C 的对象放到集合（collection）中。你可能有两个相等的类型 C 的元素 elem1 和 elem2，即“elem1 equals elem2”得出 true。尽管如此，由于经常会遇到 equals 方法实现有问题的情况，你还是可能会看到类似于如下的行为：

```
var hashSet: Set[C] = new collection.immutable.HashSet
hashSet += elem1
hashSet contains elem2    //返回 false!
```

重写 equals 方法时有四个常见的陷阱[2]可能会造成不一致：

1. 定义 equals 方法时采用了错误的方法签名。
2. 修改了 equals 方法但并没有同时修改 hashCode。
3. 用可变字段定义 equals 方法。
4. 未能按同关系定义 equals 方法。

544

本节的剩余部分将对这四个陷阱进行探讨。

陷阱#1：以错误的签名定义 equals 方法

考虑为以下这个简单点的 Point 类添加一个相等性判断方法：

```
class Point(val x: Int, val y: Int) { ... }
```

一个似乎显而易见但却是错误的方式是像这样来定义：

```
//完全错误的 equals 定义
def equals(other: Point): Boolean =
  this.x == other.x && this.y == other.y
```

这个方法有什么问题呢？初看上去，它似乎运转得还不错：

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@62d74e
p2: Point = Point@254de0

scala> val q = new Point(2, 3)
q: Point = Point@349f8a

scala> p1 equals p2
res0: Boolean = true

scala> p1 equals q
res1: Boolean = false
```

[1]注：Vaziri 等，“Declarative Object Identity Using Relation Types”。[Vaz07]
[2]注：除第三个之外其余陷阱都在 Joshua Block 的《Effective Java（第二版）》中以 Java 语言进行了描述。

但是，一旦你开始将点[3]放到集合中时，麻烦就来了：

```
scala> import scala.collection.mutable._
import scala.collection.mutable._

scala> val coll = HashSet(p1)
coll: scala.collection.mutable.Set[Point] =
Set(Point@62d74e)

scala> coll contains p2
res2: Boolean = false
```

如何解释 p1 已经被加进 coll，而 p1 和 p2 是相等的对象，但 coll 却不包含 p2 这件事呢？为了探明原因，我们遮住一个参与比较的点的精确类型，然后再做如下交互操作。我们把 p2a 定义为 p2 的别名，只不过是 Any 类型而不是 Point：

```
scala> val p2a: Any = p2
p2a: Any = Point@254de0
```

现在，如果你重复第一次的比较，不过这一次用别名 p2a 而不是 p2，你会得到：

```
scala> p1 equals p2a
res3: Boolean = false
```

出了什么问题？事实上，之前给出的 equals 方法并没有重写标准的 equals 方法，因为它的类型不同。根类 Any 中定义的 equals 方法所用的类型是[4]：

```
def equals(other: Any): Boolean
```

由于 Point 类的 equals 方法以 Point 而不是 Any 作为参数，它并没有重写 Any 类中的 equals 方法。相反，它只是一个重载的备选方法。目前，Scala 和 Java 中的重载都是根据参数的静态类型，而不是运行期类型来解析的。因此只要参数的静态类型是 Point，则调用的就是 Point 类中的 equals 方法。但是，一旦静态的参数是 Any 类型的，则调用的就是 Any 类的 equals 方法了。这个方法没有被重写，因此它仍然是通过比较对象是否一致来实现的。这就是为什么哪怕 p1 和 p2 两个点的 x 值和 y 值都相同，“p1 equals p2a”还是会得出 false。这也是为什么 HashSet 的 contains 方法返回 false。由于这个方法操作的是泛型的集合，它调用的是 Object 类的 equals 方法而不是 Point 中重载的变种。

更好的 equals 方法如下：

```
//更好一些的定义，但仍不完美
override def equals(other: Any) = other match {
  case that: Point => this.x == that.x && this.y == that.y
  case _ => false
}
```

现在 equals 有了正确的类型，它以一个类型为 Any 的值作为参数，返回 Boolean 类型的结果。这个方法的实现用到模式匹配。它首先检测 other 对象是否也是 Point 类型的。如果是，它就比较两

[3]译注：即 Point 类的实例。

[4]注：如果你编写大量的 Java 代码，你可能会预期这个方法的参数应为 Object 类型而非 Any。别担心，这还是你认为的那个 equals，编译器只是让它看上去是 Any 类型而已。

个点的坐标，然后返回结果，否则结果就是 false。

与此相关的一个陷阱是用错误的签名来定义==。通常，如果你尝试以正确的签名（即接受一个类型为 Any 的参数）来重新定义==，编译器会报错，因为这是要重写 Any 类中的 final 方法。然而，刚接触 Scala 不久的朋友有时会同时犯两个错：他们想重写==，但是又用了错误的签名。举例来说：

```
def ==(other: Point): Boolean = //别这样做!
```

这样，用户定义的==方法被当作 Any 类中同名方法重载的变种，程序因此通过了编译。但是，程序的行为就和你用错误的签名定义 equals 一样令人生疑。

陷阱#2：修改 equals 但没有同时修改 hashCode

如果你用先前最新定义的 Point 重复对 p1 和 p2a 的比较，你会得到 true，正如预期的那样。但是，如果你重复 HashSet.contains 检测，你可能还是会得到 false。

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@670f2b
p2: Point = Point@14f7c0

scala> HashSet(p1) contains p2
res4: Boolean = false
```

事实上，这个结果并非 100%确定。你也可能从这个试验中得到 true。如果遇到这种情况，你可以试试别的坐标为 1 和 2 的点，最终你会得到一个不被这个集合包含的实例。这里的问题是 Point 类重定义了 equals 方法，但是没有同时重定义 hashCode。

注意上例中用到的集合类是 HashSet。这意味着这个集合类中的元素会依据它们的哈希码被放进“哈希桶”中。contains 检测首先决定要找的桶，然后将给定的元素同该桶中所有的元素进行比较。现在的情况是，最后这个版本的 Point 类的确重定义了 equals，但并没有同时重定义 hashCode。因此 hashCode 仍然是它在 AnyRef 类中的样子：已分配对象地址的某种转换。p1 和 p2 的哈希码几乎肯定是不同的，尽管这两个点的字段值都相同。不同的哈希码意味着它们对应到集合中不同的哈希桶。contains 检测会在与 p2 的哈希码相称的桶中查找匹配的元素。在大多数情况下，p1 这个点会在另一个桶中，因此它永远也找不到。p1 和 p2 也可能偶然地被放在同一个哈希桶中，这种情况下测试就会返回 true。

问题在于 Point 最后这个实现违背了 Any 类中定义的 hashCode 方法的契约：[5]

如果两个对象根据 equals 方法是相等的，那么对它们每一个调用 hashCode 方法都必须产出相同的整型结果。

事实上，Java 中 hashCode 和 equals 应该总是一起重定义，这已是共识。除此之外，hashCode 只能依赖 equals 方法依赖的字段。对 Point 类而言，以下是合适的 hashCode 定义：

[5]注：Any 的 hashCode 契约这部分文字受到 java.lang.Object 类的 Javadoc 文档启发。

```
class Point(val x: Int, val y: Int) {
  override def hashCode = 41 * (41 + x) + y
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
  }
}
```

这只是许多种可用的 hashCode 实现的一种。给整型字段 x 加上一个常量 41，得到结果乘以质数 41，然后再加上另一个字段 y，这样我们就用较低的运行时间消耗和较少的代码得到一个合理的哈希码分布。我们将在本章稍后提供更多编写 hashCode 的指导。

增加 hashCode 解决了类似 Point 这样类定义的相等性问题。不过，还有其他一些麻烦点需要留意。

548

陷阱#3：用可变字段定义 equals

考虑如下简单调整后的 Point 类：

```
class Point(var x: Int, var y: Int) { // 有问题
  override def hashCode = 41 * (41 + x) + y
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ =>false
  }
}
```

唯一的区别是现在字段 x 和 y 是 var 而不是 val 了。equals 方法和 hashCode 方法现在是用这些可变的字段来定义，因此这些字段改变时，它们的结果也会改变。一旦你把点对象放入集合类，这就会带来奇怪的效果：

```
scala> val p = new Point(1, 2)
p: Point = Point@2b

scala> val coll = HashSet(p)
coll: scala.collection.mutable.Set[Point] = Set(Point@2b)

scala> coll contains p
res5: Boolean = true
```

现在，如果你改变点对象 p 中的一个字段，集合中还有这个点对象吗？我们来试试看：

```
scala> p.x += 1

scala> coll contains p
res7: Boolean = false
```

这看上去很奇怪啊。p 去哪儿了？如果你检查集合的 elements 迭代器是否包含 p，结果会更加奇怪：

```
scala> coll.elements contains p
res8: Boolean = true
```

549

如此说来，这是个不包含 p 的集合，但 p 却在集合的元素当中！当然，发生的事情是当 x 字段被修改后，点对象 p 相当于被放到了 coll 集合当中错误的哈希桶里。换句话说，原来那个哈希桶不再对应到这个点对象新的哈希值。某种意义上讲，p 这个点对象“离开了” coll 集合的“视野”，虽然它仍然是集合元素中的一员。

从这个示例中我们可以学到的是，如果 equals 和 hashCode 依赖于可变状态，对于潜在的用户这会带来问题。如果他们将这样的对象放入到集合中，他们必须要非常小心不去修改被依赖的状态，而这并不容易做到。如果需要的比较牵扯到对象当前的状态，你通常应该取别的名字，而不是 equals。对于最后这个 Point 的定义，我们不去重定义 hashCode，并且将相等性判断的方法命名为 equalsContent 或别的不同于 equals 的名字，这样可能会更好。如此一来，Point 就继承了默认的 equals 和 hashCode 实现，而 p 就能在它的 x 字段被修改后继续在 coll 中被定位到。

陷阱#4：未能按等同关系定义 equals 方法

根据 scala.Any 中的 equals 方法的契约约定，equals 方法必须对非 null 对象实现等同关系：[6]

- 它必须是自反射的：对任何非空值 x，表达式 x.equals(x)应返回 true。
- 它是对称的：对任何非空值 x 和 y，x.equals(y)当且仅当 y.equals(x)返回 true 时返回 true。
- 它是可传递的：对任何非空值 x、y 和 z，如果 x.equals(y)返回 true，且 y.equals(z)返回 true，则 x.equals(z)应返回 true。
- 它是一致的：对任何非空值 x 和 y，多次调用 x.equals(y)都应一致的返回 true 或一致的返回 false，只要用于对象的 equals 比较的信息没有被修改过。

550

- 对任何非空值 x，x.equals(null)应返回 false。

我们到目前为止为 Point 类开发的 equals 方法定义是满足 equals 的契约的。不过，当我们开始考虑子类时，事情就变得更复杂了。比方说有一个 Point 的子类 ColoredPoint，添加了一个类型为 Color 的字段 color。假定 Color 的定义是一个如 20.8 节所给出的枚举类：

```
object Color extends Enumeration {
  val Red, Orange, Yellow, Green, Blue, Indigo, Violet = Value
}
```

ColoredPoint 重写了 equals 方法，将新的 color 字段纳入到考虑范畴：

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) { //问题：equals 方法不对称

  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      this.color == that.color && super.equals(that)
```

[6]注：和 hashCode 类似，Any 的 equals 契约也是基于 java.lang.Object 的 equals 方法的契约。

```
      case _ => false
    }
  }
```

这是许多程序员可能会写出来的代码。注意在本例中，ColoredPoint类并不需要重写hashCode。因为 ColoredPoint 的新的 equals 定义比 Point 中被重写的定义更严格（意思是说它与更少的对象相等），hashCode 的契约依然合法。如果两个带颜色的点相等，它们必须有相同的坐标，因此它们的 hashCode 也一定会是相等的。

如果只考虑 ColoredPoint 自身，那么它的 equals 方法定义看上去还不错。不过，一旦点和带颜色的点混在一起，equals 的契约就失效了。考虑这样一个场景：

```
scala> val p = new Point(1, 2)
p: Point = Point@2b

scala> val cp = new ColoredPoint(1, 2, Color.Red)
cp: ColoredPoint = ColoredPoint@2b

scala> p equals cp
res8: Boolean = true

scala> cp equals p
res9: Boolean = false
```

比较判断“p equals cp”调用 p 的 equals 方法，该方法定义在 Point 类中。这个方法只关心两个点的坐标。因此，这个比较判断得出 true。然而，比较判断“cp equals p”调用的是 cp 的 equals 方法，定义在 ColoredPoint 类中。这个方法返回 false，因为 p 不是一个 ColoredPoint。因此 equals 方法定义出来的关系并不对称。

对集合类而言，失去对称性会带来无法预期的后果。有这样一个例子：

```
scala> HashSet[Point](p) contains cp
res10: Boolean = true

scala> HashSet[Point](cp) contains p
res11: Boolean = false
```

如此说来，就算 p 和 cp 是相等的，也只有一个 contains 测试成功，而另一个则会失败。如何修改 equals 定义让它变得对称呢？从本质上说有两种方式。要么让它们的关系变得更笼统，要么更明确。让关系变得更笼统的意思是 x 和 y 这样一对对象，不论将 x 与 y 比较，还是将 y 与 x 比较，如果得出 true，那么它们就是相等的。以下是实现这个逻辑的代码：

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) { // equals 不是可传递的

override def equals(other: Any) = other match {
  case that: ColoredPoint =>
    (this.color == that.color) && super.equals(that)
  case that: Point =>
    that equals this
  case _ =>
```

```
      false
    }
  }
```

ColoredPoint 中新的 equals 定义与原来的相比多了一个可能出现的情形：如果 other 对象是 Point 但不是 ColoredPoint，判断逻辑就会转到 Point 类的 equals 方法。这使得 equals 是对称的，是我们想要的效果。现在，不论是“cp equals p”还是“p equals cp”结果都是 true。不过，equals 的契约依然是不成立。现在的问题是，新的关系不再是可传递的了！这里有一组语句对此进行说明。定义一个点和两个不同颜色的带颜色的点，坐标都相同：

552

```
scala> val redp = new ColoredPoint(1, 2, Color.Red)
redp: ColoredPoint = ColoredPoint@2b

scala> val bluep = new ColoredPoint(1, 2, Color.Blue)
bluep: ColoredPoint = ColoredPoint@2b
```

单独来看，redp 与 p 相等，且 p 与 bluep 相等：

```
scala> redp == p
res12: Boolean = true

scala> p == bluep
res13: Boolean = true
```

但是，比对 redp 和 bluep 则得出 false：

```
scala> redp == bluep
res14: Boolean = false
```

因此，我们违反了 equals 契约中的可传递条款。

将 equals 关系变得更笼统看来走不下去了。我们再试试让它变得更明确看看。让 equals 更明确的方式之一是总是将不同类型的对象当作是不同的。这可以通过修改 Point 类和 ColoredPoint 类的 equals 方法来实现。在 Point 类中，你可以添加一个额外的比较，来检查是否另一个 Point 的运行期类确切的是 Point 的类：

```
// 一个技术上讲合法，但并不满足要求的 equals 方法
class Point(val x: Int, val y: Int) {
  override def hashCode = 41 * (41 + x) + y
  override def equals(other: Any) = other match {
    case that: Point =>
      this.x == that.x && this.y == that.y &&
      this.getClass == that.getClass
    case_ =>false
  }
}
```

553

然后你就可以将 ColoredPoint 类的实现回退到之前违反了对称性要求的版本[7]：

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) {
```

[7]注：有了新的 Point 类的 equals 实现，这个版本的 ColorPoint 就不再违背对称性的要求了。

```
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (this.color == that.color) && super.equals(that)
    case _ => false
  }
}
```

这里，Point 类的一个实例只有在其他实例与它有着相同坐标并且有相同的运行期类型（即不论哪个对象的.getClass 都返回相同的值）时才相等。新的定义既满足对称性又满足可传递性，因为现在不同类型的对象间的比较判断总是得出 false。因此一个带颜色的点永远不可能与一个点相等。这个约定俗成看似有理，但有人会说新的定义太严格了。

考虑如下这样一个变通的方式来定义坐标为(1, 2)的点：

```
scala> val pAnon = new Point(1, 1) { override val y = 2 }
pAnon: Point = $anon$1@2b
```

pAnon 和 p 相等吗？答案是不相等，因为与 p 和 pAnon 相关联的 java.lang.Class 对象不同。p 是 Point，而 pAnon 是匿名的 Point 类的（子）类。但很清楚的是，pAnon 只是另一个坐标在(1, 2)的点。将它当作是不同于 p 的点似乎并不合理。

这样说来我们似乎卡住了。有没有一种合理的方式来重新定义类继承关系中若干级别上的相等性，同时不违背其契约呢？事实上，有这样一种方式，不过它需要与 equals 和 hashCode 一道再多定义一个方法。我们的构想是一旦类重定义 equals（以及 hashCode），它应该同时明确指出该类的对象不与任何定义了不同相等性方法的超类的对象相等。

554

这是通过给每个重定义 equals 方法的类添加一个 canEqual 方法来实现的。这里是方法签名：

```
def canEqual(other: Any): Boolean
```

如果 other 对象是（重）定义了 canEqual 方法的类的实例，则该方法应返回 true，否则应返回 false。equals 方法中调用 canEqual 来确保对象可以双向进行比较。清单 28.1 用如下代码给出了一个新的（也是最终的）Point 类实现：

```
class Point(val x: Int, val y: Int) {
  override def hashCode = 41 * (41 + x) + y
  override def equals(other: Any) = other match {
    case that: Point =>
      (that canEqual this) &&
      (this.x == that.x) && (this.y == that.y)
    case _ => false
  }
  def canEqual(other: Any) = other.isInstanceOf[Point]
}
```

清单 28.1 调用 canEqual 的超类 equals 方法

在这个版本的 Point 类中的 equals 方法包含了额外的要求，那就是另外那个对象要能够与这个对象相等，而这取决于 canEqual 方法。根据 Point 的 canEqual 方法的实现，所有 Point 的实例都可以相等。

555

清单28.2显示了相应的ColoredPoint的实现。可以看到新的Point和ColoredPoint定义保持了equals的契约，相等性是对称的，也是可传递的。将Point和ColoredPoint相比较总是得出false。的确，对任何点p和带颜色的点cp而言，“p equals cp”将返回false，因为“cp canEqual p”会返回false。反过来的比较判断“cp equals p”也将返回false，因为p不是ColoredPoint的实例，因而ColoredPoint的equals方法体内的第一个模式匹配将会失败。

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) {

  override def hashCode = 41 * super.hashCode + color.hashCode
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (that canEqual this) &&
      super.equals(that) && this.color == that.color
    case _ =>
      false
  }
  override def canEqual(other: Any) =
    other.isInstanceOf[ColoredPoint]
}
```

清单28.2 调用canEqual的子类equals方法

另一方面，Point的不同子类的示例可以相等，只要这些类不重定义相等性方法。举例来说，采用新的类定义，p和pAnon的比较判断会得出true。示例如下：

```
scala> val p = new Point(1, 2)
p: Point = Point@6bc

scala> val cp = new ColoredPoint(1, 2, Color.Indigo)
cp: ColoredPoint = ColoredPoint@11421

scala> val pAnon = new Point(1, 1) { override val y = 2 }
pAnon: Point = $anon$1@6bc

scala> val coll = List(p)
coll: List[Point] = List(Point@6bc)

scala> coll contains p
res0: Boolean = true

scala> coll contains cp
res1: Boolean = false

scala> coll contains pAnon
res2: Boolean = true
```

这些示例展示了如果超类的equals实现定义并调用了canEqual，则实现子类的程序员可以决定他们的子类是否可以与超类的实例相等。比方说，由于ColoredPoint重写了canEquals方法，带颜色的点不可能和一个普通的点相等。但pAnon引用的匿名子类并未重写canEqual方法，它的实例可以与Point的示例相等。

一个潜在的对 canEqual 这种方式的批评是它违背了里氏替换原则（LSP）。举例来说，通过比较运行期类型来实现 equals 的技巧（让我们无法定义可以与超类实例相等的子类）是违反 LSP 的[8]。这背后的原因是 LSP 指出你应该能够在要求超类示例的地方用子类的示例（替代）。但是在前例中，“coll contains cp”返回了 false，尽管 cp 的 x 和 y 值与集合类中的点相匹配。这样说起来它似乎是违反了 LSP，因为你不能在预期 Point 的地方使用 ColoredPoint。但我们认为这是错误的解读，因为 LSP 并不要求子类的行为与其超类完全一致，而只是满足和实现超类的契约即可。

编写比对运行期类型的 equals 方法的问题不在于它违反了 LSP，而是它没有给你一种方式来创建其实例可以与超类实例相等的子类。举例来说，假如我们在前例中用了运行期类型的技巧，“coll contains pAnon”应该会返回 false，但这并是不是我们想要的。相反，我们的确是想要“coll contains cp”返回 false，因为通过在 ColoredPoint 中重写 equals 方法，我们基本上就是说，一个位于(1, 2)的靛蓝色的点和一个同样位于(1, 2)的无颜色的点并不是一回事。这样一来，在前例中我们可以向集合类的 contains 方法传递两个不同的 Point 子类的实例，并得到两个不同的答案，而且两个答案都是正确的。

28.3 定义带参数类型的相等性

前面例子中的 equals 方法都以模式匹配开始的，检测被操作对象的类型是否与包含 equals 方法的类型一致。当类带参数时，这样的设计须做一些调整。作为一个示例，我们来考虑下二叉树。

557

清单 28.3 所示的类层级定义了二叉树的抽象类 Tree，它有两个可选的实现：一个 EmptyTree 对象和一个 Branch 类用于表示非空树。非空树由某元素 elem 及 left 和 right 子树构成。元素的类型由类型参数 T 给出。

```
trait Tree[+T] {
  def elem: T
  def left: Tree[T]
  def right: Tree[T]
}

object EmptyTree extends Tree[Nothing] {
  def elem =
    throw new NoSuchElementException("EmptyTree.elem")
  def left =
    throw new NoSuchElementException("EmptyTree.left")
  def right =
    throw new NoSuchElementException("EmptyTree.right")
}

class Branch[+T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T]
```

清单 28.3 二叉树的类继承关系

[8]注：Bloch，《Effective Java 第二版》第 39 页。[Blo08]

我们现在将为这些类添加 equals 和 hashCode 方法。对 Tree 类自身而言，什么都不需要做，因为我们假定这些方法在这个抽象类的每个具体实现中都被单独地实现了。对 EmtyTree 对象而言，也不需要我们做什么，因为 EmptyTree 从 AnyRef 继承下来的 equals 和 hashCode 的默认实现就可以满足要求。毕竟，EmptyTree 只和自己相等，因此它的相等性就应该是引用相等，这也是从 AnyRef 继承下来就有的行为。

不过给 Branch 添加 equals 和 hashCode 方法需要费更多工夫。只有当两个 Branch 值拥有相等的 elem、left 和 right 字段时，它们才是相等的。自然而然我们就会想到用本章前面几节开发的 equals 方法的设计。这样就得到：

558

```
class Branch[T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {

  override def equals(other: Any) = other match {
    case that: Branch[T] => this.elem == that.elem &&
                            this.left == that.left &&
                            this.right == that.right

    case _ => false
  }
}
```

不过，编译这个示例，会遇到“unchecked（未检查）”警告。加上-unchecked 选项再次编译就揭示出如下问题：

```
fsc -unchecked Tree.scala
Tree.scala:14: warning: non variable type-argument T in type pattern is unchecked
since it is eliminated by erasure[9]
    case that: Branch[T] => this.elem == that.elem &&
                    ^
```

正如警告信息所说，这里有一个针对 Branch[T]类型的模式匹配，但系统只能检查当 other 引用的是（某种）Branch，而无法检查该树的元素类型是 T 的情况。你在第 19 章曾经见识过这背后的原因：参数化类型的元素类型在编辑器的擦除阶段被抹掉，这些信息在运行期无法被检查。

559

那么为此你能做些什么呢？所幸，你在比较两个 Branch 实例时并不一定须要检查它们是否有相同的元素类型。两个元素类型不同的 Branch 也很有可能相等，只要它们的字段是相同的。简单的例子是包含单个 Nil 元素和两个空子树的 Branch。考虑这样两个 Branch 为相等是说得通的，不论它们的静态类型是什么。

```
scala> val b1 = new Branch[List[String]](Nil, EmptyTree, EmptyTree)
b1: Branch[List[String]] = Branch@2f1eb9

scala> val b2 = new Branch[List[Int]](Nil, EmptyTree, EmptyTree)
b2: Branch[List[Int]] = Branch@be55d1
```

[9]译注：这里的命令行提示的意思是，类型模式中不可变的类型参数 T 未被检查，因为它（在编译过程中）被擦写。

```
scala> b1 == b2
res0: Boolean = true
```

以上比较的正面结果是通过之前所示的 Branch 的 equals 实现得到的。这说明 Branch 的元素类型没有被检查——如果检查了，结果应该是 false 才对。

注意，人们可能对比较的两种可能的结果中的哪一个更自然会有不同的看法。这最终取决于这些类是如何被表示的心理模型。在类型参数只在编译期有作用的模型中，自然会认为两个 Branch 值 b1 和 b2 是相等的。而在类型参数也是对象值的一部分的另一种模型中，同样自然地会认为它们不相等。由于 Scala 采用的是类型擦除模型，类型参数在运行期不被保留，因此 b1 和 b2 自然而然被认为是相等的。

为使 equals 方法不产生 unchecked 警告只需要一个小小的修改：不用元素类型 T，而是用一个小写字母，如 t：

```
case that: Branch[t] => this.elem == that.elem &&
                        this.left == that.left &&
                        this.right == that.right
```

还记得 15.2 节吧， 模式中以小写字母开始的类型参数代表未知的类型。因而，模式匹配：

```
case that: Branch[t] =>
```

对任何类型都会成功。类型参数 t 代表 Branch 的未知元素类型。它也可以被替换为下划线，如下所示，和之前的那一个是等效的：

```
case that: Branch[_] =>
```

剩下的最后一件事是为 Branch 类定义其余两个方法，hashCode 和 canEqual，它们须要随着 equals 一起修改。以下是 hashCode 的一个可能的实现：

```
override def hashCode: Int =
  41 * (
    41 * (
      41+ elem.hashCode
    ) + left.hashCode
  ) + right.hashCode
```

这只是许多实现可能中的一种。如之前所示，它的原理是拿到所有字段的 hashCode 值，然后用质数来进行加法和乘法，再把它们拼在一起。这里是 Branch 类的 canEqual 方法实现：

```
def canEqual(other: Any) = other match {
  case that: Branch[_] => true
  case _ => false
}
```

canEqual 方法的实现用到了类型模式匹配。我们也可以用 isInstanceOf 来实现它：

```
def canEqual(other: Any) = other.isInstanceOf[Branch[_]]
```

如果你想挑毛病的话（我们鼓励你这样做！），你可能会想，上面的类型中出现的这个下划线意味着什么。毕竟，Branch[_]技术上讲是方法的类型参数，而不是类型模式，因此怎么可能留有未定义的

部分呢？这个问题的答案在下一章中：Branch[_]是所谓的存在类型的简写，粗略的说这是一个有着未知部分的类型。因此尽管技术上讲下划线在模式匹配和方法调用的类型参数中代表两种不同的东西，本质上含义是相同的：它让你将某些东西标记为未知。最终版的 Branch 如清单 28.4 所示。

561

```
class Branch[T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {

  override def equals(other: Any) = other match {
    case that: Branch[_] => (that canEqual this) &&
                            this.elem == that.elem &&
                            this.left == that.left &&
                            this.right == that.right
    case _ => false
  }

  def canEqual(other: Any) = other.isInstanceOf[Branch[_]]

  override def hashCode: Int =
    41 * (
      41 *(
        41+ elem.hashCode
      ) + left.hashCode
    ) + right.hashCode
}
```

清单 28.4 带有 equals 和 hashCode 方法的参数化类型

28.4 equals 和 hashCode 的制作方法

本节，我们将提供分步骤创建 equals 和 hashCode 方法的指导，这对绝大多数情形而言是足够了。我们将使用清单 28.5 所示的 Rational 类的方法来进行说明。为了创建这个类，我们去除了清单 6.5 所示的 Rational 类的数学操作符方法。我们还对 toString 做了小小的增强，并修改了 numer 和 denom 的初始化方法来将所有分数进行规整[10]，让它们的分母为正数（也就是说将 $\frac{1}{-2}$ 转换为 $\frac{-1}{2}$ ）。这里是重写 equals 的方法：

562

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = (if (d <0) -n else n) / g
  val denom = d.abs / g
```

[10]译注：关于规整的含义，原文 normalize 有规范化的意思，用在分数上也指约分。

```
private def gcd(a: Int, b: Int): Int =
  if(b == 0) a else gcd(b, a % b)

override def equals(other: Any): Boolean =
  other match {
    case that: Rational =>
      (that canEqual this) &&
      numer == that.numer &&
      denom == that.denom
    case_ => false
  }

def canEqual(other: Any): Boolean =
  other.isInstanceOf[Rational]

override def hashCode: Int =
  41 * (
    41 + numer
  ) + denom

override def toString =
  if(denom == 1) numer.toString else numer +"/"+ denom
}
```

563

清单 28.5 带有 equals 和 hashCode 的 Rational 类

1. 如果要在非 final 的类中重写 equals 方法，你应该创建 canEqual 方法。如果 equals 的定义是继承自 AnyRef（即 equals 没有在类继承关系的上方被重新定义），则 canEqual 的定义将会是新的，否则它将重写之前同名方法的定义。需求中的唯一例外是关于重定义了继承自 AnyRef 的 equals 方法的 final 类的。对它们来说，28.2 节中描述的子类的问题并不会出现，因此它们并不需要定义 canEqual。传递给 canEqual 的对象类型应为 Any：

   ```
   def canEqual(other: Any): Boolean =
   ```

2. 如果参数对象是当前类的实例则 canEqual 方法应返回 true（即 canEqual 定义所在的类），否则应返回 false：

   ```
   other.isInstanceOf[Rational]
   ```

3. 在 equals 方法中，记得声明传入的对象类型为 Any：

   ```
   override def equals(other: Any): Boolean =
   ```

4. 将 equals 的方法体写为单个 match 表达式，而 match 的选择器应为传递给 equals 的对象：

   ```
   other match {
     // ...
   }
   ```

5. match 表达式应有两个 case，第一个应声明为你所定义 equals 方法的类的类型模式：

   ```
   case that: Rational =>
   ```

564

6. 在这个 case 的语句体中，编写一个表达式，把两个对象要相等必须为 true 的独立表达式以逻辑

与的方式结合起来。如果重写的 equals 方法并非是 AnyRef 的那一个，你很可能想要包含对超类的 equals 方法的调用：

```
super.equals(that) &&
```

如果为首个引入 canEqual 的类定义 equals 方法，你应该调用其参数的 canEqual 方法，将 this 作为参数传递进去：

```
(that canEqual this) &&
```

重写的 equals 方法也应该包含 canEqual 的调用，除非它们包含了对 super.equals 的调用。在后面这个情形中，canEqual 测试已经会在超类调用中完成。最后，对每个与相等性相关的字段，验证本对象的字段与传入对象的对应字段是相等的。

```
numer == that.numer &&
denom == that.denom
```

7. 对第二个 case，用一个通配的模式返回 false：

```
case _ => false
```

如果严格遵照前面的制作方法，相等性就能够被保证是等同关系，正如 equals 的契约所要求的那样。

对于 hashCode，如果你使用下面这个制作方法，通常也可以达到满意的结果，这和“Effective Java”[11]中针对 Java 类的推荐做法类似。将对象中用在 equals 方法里计算相等性的每个字段（是为“相关”字段）都包含进来。对每个相关字段，不管类型是什么，你都可以通过调用 hashCode 来计算出一个哈希码。为计算整个对象的哈希码，对第一个字段的哈希码加上 41，再乘以 41，再加上第二个字段的哈希码，再乘以 41，再加上第三个字段的哈希码，再乘以 41，如此继续下去，直到完成所有相关字段的累计。

举例来说，为实现有五个名为 a、b、c、d、e 的相关字段的对象的哈希码，你会这样写：

```
overridedef hashCode: Int =
  41 * (
    41 * (
      41 * (
        41 * (
          41 + a.hashCode
        ) + b.hashCode
      ) + c.hashCode
    ) + d.hashCode
  ) + e.hashCode
```

如果你愿意，你也可以不对类型为 Int、Short、Byte 和 Char 的字段调用 hashCode。Int 的哈希码是 Int 的值，而 Short、Byte 和 Char 的哈希码也会自动变宽为 Int。由于 numer 和 denom 都是 Int，我们实现的 Rational 的 hashCode 是这样的：

```
override def hashCode: Int =
  41 * (
    41 + numer
    ) + denom
```

[11] Bloch，《Effective Java（第二版）》。[Blo08]

之所以选择 41 来作为乘数是因为它是个奇（jī）质数。你也可以用别的数字，不过仍应该是奇质数，以防最小化溢出时潜在的信息丢失。我们给最里层的值加 41 是为了减少第一个乘法得到 0 的可能性，这是假定第一个字段更有可能是 0 而非-41。我们选择 41 这个数字只是为了好看，你也可以用任何非 0 的整数。

如果 equals 方法将 super.equals(that) 调用作为其计算的一部分，你应该以调用 super.hashCode 开始你的 hashCode 计算。举例来说，如果 Rational 的 equals 方法调用了 super.equals(that)，它的 hashCode 可能就是这个样子的：

```
override def hashCode: Int =
  41 * (
    41 *(
      super.hashCode
    ) + numer
  ) + denom
```

当你用这种方式编写 hashCode 方法时，需要记住的一点是，hashCode 好坏要看你构建它用到的哈希码的好坏，也就是说通过调用对象中相关字段的 hashCode 得到的值。有时你可能除了在字段上调用 hashCode 之外还须要做些额外的事情才能得到该字段有用的哈希码。举例来说，如果某个字段是集合，你或许希望这个字段的哈希码是基于集合类中的所有元素；如果字段是 List、Set、Map 或元组，你可以简单地调用字段的 hashCode，因为这些类的 equals 和 hashCode 方法被重写过，会考虑包含的元素。但是对 Array 而言并不是这样，它们在计算哈希码时并不会考虑元素。因此，对数组而言，你应该将每个元素当作是对象的字段，主动调用对每个元素的 hashCode，或者将数组传递给单例对象 java.util.Arrays 的某一个 hashCode 方法。

最后，如果你发现一个特定的哈希码计算影响到程序的性能，也可以考虑将哈希码缓存起来。如果对象是不可变的，可以在对象创建时计算哈希码并保存到一个字段中。你可以简单地通过用 val 而不是 def 重写 hashCode 来做到，就像这样：

```
override val hashCode: Int =
  41 * (
    41 + numer
  ) + denom
```

这种方法是用内存来换取计算时间，因为每个不可变类的实例将会多出一个字段来保留缓存的哈希码值。

28.5 小结

来回顾一下，定义一个正确的 equals 实现出乎我们意料的微妙。你必须很小心类型签名，你必须重写 hashCode，你要避免依赖可变字段，并且如果类不是 final 的，你应该实现并使用 canEqual 方法。既然实现一个正确的相等性方法如此困难，你可以将可比较的对象的类定义为样本类。这样，Scala 编辑器将会自动地添加正确的符合各项要求的 equals 和 hashCode 方法。

第 29 章

结合 Scala 和 Java

Scala 代码经常会和大型的 Java 程序及框架一起使用。由于 Scala 与 Java 高度兼容，大部分时间在结合这两种语言时并不需要太多顾虑。举例来说，标准的框架如 Swing、Servlet 和 JUnit 等可以很好的与 Scala 一起工作。尽管如此，时不时的你也会遇到一些结合 Java 和 Scala 的问题。

本章描述了结合 Java 和 Scala 的两个方面。首先，它会讨论 Scala 是如何被翻译成 Java 的，这在你从 Java 调用 Scala 代码时尤为重要。其次，它将讨论在 Scala 中使用 Java 注解，如果你想在现有的 Java 框架中使用 Scala，这将是一个十分重要的特性。

29.1 在 Java 中使用 Scala

大多数时候，你在源码层面考虑 Scala 就可以了。不过，如果你知道一些关于 Scala 到 Java 的翻译的细节，你会对这个系统如何运转有更全面的理解。进一步说，如果你从 Java 调用 Scala 代码，你将须要知道从 Java 的角度看 Scala 代码长什么样。

一般的原则

Scala 的实现方式是将代码翻译成标准的 Java 字节码。Scala 的特性尽可能地直接映射为相对等的 Java 特性。举例来说，Scala 的类、方法、字符串、异常等都和它们在 Java 中的对应概念一样编译成相同的 Java 字节码。

为了实现这一点，在设计 Scala 的过程中，有时需要做出艰难的抉择。例如，在运行期使用运行期类型解析确定重载方法，而不是在编译期决定，这也许是个不错的想法，不过这样的设计会破坏 Java 的重载解析，使得混用 Java 和 Scala 变得更加困难。在这个问题上，Scala 与 Java 的重载解析保持一致，因而 Scala 的方法和方法调用可以直接与 Java 的方法和方法调用相对应。

对于其他一些特性，Scala 有它自己的设计。举例来说，特质在 Java 中没有与之相当的对应。同样的，虽然 Scala 和 Java 都有泛型，这两个系统在细节上存在冲突。对于类似这样的语言特性，Scala 代码无法直接映射为 Java 的语法结构，因此它必须结合 Java 现有的特性来进行编码。

对于这些不能直接映射的特性，编码并不是固定的。目前有一个进行中的工作是让这个翻译尽可能的简单，因此当你读到这本书时，有些细节可能与本书写作之时不同了。你可以使用类似 `javap` 这样

的工具查看“.class”文件来获知当前的Scala编译器使用的翻译。

这些是一般性的原则。现在我们来考虑一些特例。

值类型

类似Int这样的值类型翻译成Java有两种不同的方式。只要可能，编译器会将Scala的Int翻译为Java的int以获得更好的性能。但有时这做不到，因为编译器不确定它在翻译的是一个Int还是另外某种数据类型。举例来说，某个特定的List[Any]可能只保有Int型的元素，但编译器没有办法确认这一点。

对于这样的情形，编译器不确定某个对象是不是值类型，而是会使用对象并依赖相应的包装类。举例来说，如java.lang.Integer这样的包装类允许一个值类型被包装在Java对象中，由需要对象的代码操作[1]。

单例对象

570

Java并没有单例对象的确切对应，不过它的确有静态方法。Scala对单例对象的翻译采用了静态和实例方法相结合的方式。对每一个Scala单例对象，编译器都会为这个对象创建一个名称后加美元符号的Java类。对一个名为App的单例对象，编译器产出一个名为App$的Java类。这个类拥有Scala单例对象的所有方法和字段，这个Java类同时还有一个名为MODULE$的静态字段，保存该类在运行期创建的一个实例。

作为一个完整的示例，假定你编译下面这个单例对象：

```
object App {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Scala将会生成一个Java类App$，其字段和方法如下：

```
$ javap App$
public final class App$ extends java.lang.Object
implements scala.ScalaObject{
    public static final App$ MODULE$;
    public static {};
    public App$();
    public void main(java.lang.String[]);
    publicint $tag();
}
```

一般情况下的翻译就是这样。一个重要的特例是当你面对的是一个“独立”（即没有同名的类与之对应）的单例对象时。举例来说，你可能有一个名为App的单例对象，且并没有任何名为App的类，在这种情况下，编译器将会创建一个名为App的Java类，这个类对于每个Scala单例对象的方法都有一

[1]注：值类型的实现在11.2节曾详细介绍过。

个静态的转发方法与之对应：

```
$ javap App
Compiled from "App.scala"
public final class App extends java.lang.Object{
    public static final int $tag();
    public static final void main(java.lang.String[]);
}
```


相反，如果你确实有一个名为 App 的类，Scala 会创建一个相对应的 Java 类 App 来保存你定义的 App 类的成员。在这种情况下，它就不会添加任何转发到同名单例对象的方法，Java 代码则必须通过 MODULE$字段来访问这个单例。

作为接口的特质

编译任何特质都会创建一个同名的 Java 接口。这个接口可以作为 Java 类型使用，你可以通过这个类型的变量来调用 Scala 对象的方法。

在 Java 中实现一个特质则完全是另一回事。通常的情况下，这样做并不实际。不过也有的特例须要注意。如果你制作的 Scala 特质只包含抽象方法，则这个特质会被直接翻译成 Java 接口，不须要关心任何其他代码。从本质上讲，这意味着只要你愿意，你可以用 Scala 的语法来编写 Java 接口。

29.2 注解

Scala 通用的注解系统在第 25 章讨论，本节讨论的是注解中 Java 特定的方面。

标准注解的额外效果

有一些注解编译器在针对 Java 平台编译时会产生额外的信息。当编译器看到这样的注解时，它会首先根据一般的 Scala 原则去处理，然后针对 Java 做一些额外的工作。

过期。对于任何标记为@deprecated 的方法或类，编译器会为产出的代码添加 Java 自己的过期注解。因此，Java 编译器能够在 Java 代码访问过期 Scala 方法时给出过期警告。

Volatile 字段。同理，Scala 中标记为@volatile 的字段会在产出的代码中添加 Java 的 volatile 修饰符。因而，Scala 中的 volatile 字段与 Java 的处理机制完全一致，而对 volatile 字段的访问也是完全根据 Java 内存模型所规定的 volatile 字段处理原则来进行排列的。

序列化。Scala 的三个标准序列化注解全部都被翻译成 Java 中对等的语法结构。@serializable 类会被加上 Java 的 Serializable 接口。@SerialVersionUID(1234L)会被转换成如下 Java 字段定义：

```
// 序列化版本标记
private final static long SerialVersionUID= 1234L
```

任何标记为@transient 的变量会被加上 Java 的 transient 修饰符。

抛出的异常

Scala 并不检查抛出的异常是否被代码捕获。也就是说，Scala 的方法并没有与 Java 中的 throws 声明相对应的定义。所有 Scala 方法都被翻译成没有声明任何抛出异常的 Java 方法[2]。

声明抛出异常这个特性之所以被 Scala 排除在外是因为在 Java 中人们对它的体验并不全是正面的。由于用 throws 语句注解方法是个沉重的负担，因此许多开发者都编写吃掉并能丢弃异常的代码，仅仅是为了在不增加所有这些 throws 语句的情况下让代码编译通过。他们可能也想在之后增强这部分的异常处理，但经验显示时间压力下的程序员们几乎从不回过头来增加正确的异常处理。这样带来的扭曲的结果就是这个本意很好的特定特性让代码不那么可靠了。大量生产环境的 Java 代码都吃掉并隐藏运行期的异常，而这样做的原因只是让编译器满意而已。

有时候我们与 Java 对接，可能会需要编写描述你的方法可能会抛出哪些异常的对 Java 友好的注解。举例来说，每个 RMI 远程接口中的方法都需要在 throws 语句中提到 java.io.RemoteException。因此，如果你希望用带有抽象方法定义的 Scala 特质编写 RMI 远程接口，会需要在这些方法的 throws 语句中列出 RemoteException。为了达成这个目的，你所要做的是用@throws 注解标记你的方法。例如，清单 29.1 中所示的 Scala 类就有一个标记为抛出 IOException 的方法。

573

```
import java.io._
class Reader(fname: String) {
  private val in =
    new BufferedReader(new FileReader(fname))

  @throws(classOf[IOException])
  def read() = in.read()
}
```

清单 29.1 声明了 Java throws 语句的 Scala 方法

以下是从 Java 看它是什么样子：

```
$ javap Reader
Compiled from "Reader.scala"
public class Reader extends java.lang.Object implements
scala.ScalaObject{
    public Reader(java.lang.String);
    public int read()       throws java.io.IOException;
    public int $tag();
}
$
```

注意 read 方法用 Java 的 throws 语句指出它可能会抛出 IOException。

Java 注解

Java 框架中的注解可以直接在 Scala 代码中使用。任何 Java 框架都会看到你编写的注解，就好像你是用 Java 编写的一样。

[2]注：这些能够工作的原因是反正 Java 字节码的验证器也不会检查这些声明！Java 编译器会检查，但验证器不会。

有相当多的 Java 包都使用注解，JUnit 4 就是其中一个例子。JUnit 是一个用于编写自动化测试和运行这些测试的框架。最新版 JUnit 4 使用注解来标明代码中哪些部分是测试代码。这背后的想法是为代码编写大量测试，然后只要你对源码有修改，就运行这些测试。这样，如果你的修改带来新的 bug，其中某一个测试会失败，你就可以立即找到问题所在。

编写测试很容易。只需要在顶级类中编写一个测试代码的方法，然后用一个注解来将它标记为测试。它的样子类似这样：

```
import org.junit.Test
import org.junit.Assert.assertEquals

class SetTest{

  @Test
  def testMultiAdd {
    val set = Set() + 1 + 2 + 3 + 1 + 2 + 3
    assertEquals(3, set.size)
  }
}
```

testMultiAdd 方法是一个测试。这个测试向集中添加多个项目，并确保每个项目只被添加了一次。assertEquals 方法是 JUnit API 的一部分，检查它的两个参数将会是相等的。如果它们不同，则测试失败。在本例中，测试检查了重复增加相同的数字并不会增加集的大小。

该测试用 org.junit.Test 注解标记。注意这个注解被引入过，因此可以直接通过@Test 引用，而无须写成更繁琐的@org.junit.Test。

这些就是全部的内容了。我们可以用任何 JUnit 测试运行器来运行这个测试。在这里它是通过命令行的测试运行器来执行的：

```
$ scala -cp junit-4.3.1.jar:. org.junit.runner.JUnitCoreSetTest
JUnit version 4.3.1
.
Time: 0.023

OK (1 test)
```

编写你自己的注解

为了让注解对 Java 反射可见，你必须用 Java 的语法编写并用 javac 编译。对这样的用例而言，用 Scala 来编写注解看上去并没有什么帮助，因此标准的编译器不支持这样做。这背后的原因是 Scala 的支持将不可避免地会无法实现 Java 注解的全部功能，而且 Scala 可能会在未来的某一天拥有自己的反射，但你可能想要使用 Scala 反射来访问 Scala 的注解。

这里有一个注解的示例：

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.METHOD)
public @interface Ignore { }
```

使用 javac 编译上述代码后，你可以像下面这样来使用该注解：

```
object Tests {
  @Ignore
  def testData = List(0, 1, -1, 5, -5)

  def test1 {
    assert(testData == (testData.head :: testData.tail))
  }

  def test2 {
    assert(testData.contains(testData.head))
  }
}
```

在本例中，test1 和 test2 应为测试方法，尽管 testData 以“test”打头，它实际上应该被忽略。

为了查看这些注解是否被用到，可以用 Java 反射 API。这里有一段示例代码显示这是如何工作的：

```
for {
  method <- Tests.getClass.getMethods
  if method.getName.startsWith("test")
  if method.getAnnotation(classOf[Ignore]) == null
} {
  println("found a test method: " + method)
}
```

在这里，我们用反射方法 getClass 和 getMethods 来检查输入对象的类的所有字段，这些只是普通的反射方法。与注解特定的部分是对 getAnnotation 方法的使用。就 Java 1.5 而言，许多反射对象都有一个 getAnnotation 方法来查找特定类型的注解。在本例中，代码查找的是一个我们新的 Ignore 类型的注解。由于这是 Java API，是否成功取决于结果是 null 还是实际的注解对象。

以下是代码运行起来的样子：

```
$ javac Ignore.java
$ scalac Tests.scala
$ scalac FindTests.scala
$ scala FindTests
found a test method: public void Tests$.test2()
found a test method: public void Tests$.test1()
```

附带提一句，请注意以 Java 反射的视角来看这些方法是位于 Test$类而非 Test 类中。如本章一开始讲到的那样，Scala 的单例对象的实现位于一个名称后增加美元符号的 Java 类中。对本例而言，Tests 的实现位于 Java 类 Tests$中。

请注意，在使用 Java 注解时你必须遵循它们所规定的限制。举例来说，在注解的参数中，只能使用常量，而不能使用表达式。你可以支持@serial(1234)但不是@serial(x * 2)，因为 x * 2 并不是常量。

29.3 存在类型

所有 Java 类型在 Scala 中都有对等的概念。这是必要的，因为只有这样 Scala 代码才能访问任何合法的 Java 类。大多数时候这个翻译很直截了当。Java 中的 `Pattern` 在 Scala 中就是 `Pattern`，而 Java 中的 `Iterator<Component>`在 Scala 中就是 `Iterator[Component]`。但是在某些情况下，到目前为止看到的 Scala 类型并没有满足要求。对于诸如 `Iterator<?>`或者 `Iterator<? extends Component>`的 Java 通配类型我们能做什么？对于没有类型参数的原始类型 `Iterator` 我们又能做什么？对于通配类型和原始类型，Scala 使用一种额外的叫做存在类型的类型来表示。

存在类型是 Scala 语言中被完整支持的一部分，不过实际上它们主要用于从 Scala 访问 Java 类型。本节给出存在类型的工作机制的概览，不过它的主要用途是当 Scala 代码访问 Java 时，你能够理解编译器报告的错误提示。

存在类型的通用形式如下：

```
type forSome{ declarations }
```

type 部分是任意的 Scala 类型，而 declarations 部分是一个抽象 `val` 和 `type` 的列表。这个定义解读为：声明的变量和类型是存在但未知的，正如类中的抽象成员那样。这个类型进而被允许引用这些声明的变量和类型，虽然编译器并不知道它们具体指向什么。

来看一些具体的例子。Java 中的 `Iterator<?>`可以在 Scala 中写为：

```
Iterator[T] forSome { type T }
```

从左往右看，这是一个 T 的 `Iterator`，而 T 是某种类型 `T`。类型 T 具体是什么未知，可以是任何类型，不过对这个特定的 `Iterator` 而言，它是固定不变的。同理，Java 中的 `Iterator<? extends Component>`可以在 Scala 中这样表示：

```
Iterator[T] forSome { type T <: Component }
```

这是一个 T 的 `Iterator`，针对某种为 `Component` 子类型的类型 `T`。在本例中 T 仍然未知，但它一定是 `Component` 的子类。

顺便说一下，有一种更简短的方式来写这些示例。如果你写下 `Iterator[_]`，它的含义与 `Iterator[T]for Some{type T}`相同。这是针对存在类型的占位符语法，在精神上与 8.5 节所描述的函数字面量的占位符语法相似。如果在可以使用表达式的位置使用下划线（_），Scala 则将它当作占位符并且做出一个函数字面量。对类型而言它也有类似处理。如果在可以使用类型的地方使用下划线，则 Scala 会为你做出一个存在类型。每个下划线在 `forSome` 语句中变成一个类型参数，因而如果你在同一个类型中使用两个下划线，则将得到与在 `forSome` 语句中使用两个类型相同的效果。

你还可以在使用占位符语法是插入上界和下界。只需要简单地给下划线加上它们，而不必在 `forSome` 语句中这样做。类型 `Iterator[_ <: Component]`和你刚看到的类是一样的：

```
Iterator[T] forSome { type T <: Component }
```

关于存在类型本身就介绍这么多。那实际上怎么使用它们呢？对于简单的用例而言，你可以当forSome不存在来使用存在类型。尽管forSome语句中的类型和值是未知的，Scala还是会检查程序是否完备。举例来说，假定你有如下这样一个Java类：

```
//这是一个用到通配的Java类
public class Wild {
  Collection<?> contents() {
    Collection<String> stuff = new Vector<String>();
    stuff.add("a");
    stuff.add("b");
    stuff.add("see");
    return stuff;
  }
}
```

如果在Scala中访问这个类，你将会看到它有一个存在类型：

```
scala> val contents = (new Wild).contents
contents: java.util.Collection[?0] forSome { type ?0 } =
   [a, b, see]
```

如果想知道这个集合类中有多少元素，你可以简单地忽略存在定义的部分，并像平常一样调用size方法：

```
scala> contents.size()
res0: Int = 3
```

对于更复杂的用例，存在类型可能会更笨拙一些，因为我们没有办法给存在类型命名。举例来说，假定你想要创建一个可变的Scala集合，并使用contents的元素初始化它：

```
import scala.collection.mutable.Set
val iter = (new Wild).contents.iterator
val set = Set.empty[???]      //这里该用什么类型呢?
while (iter.hasMore)
  set += iter.next()
```

579

问题出现在第三行。我们没有办法给出Java集合中的元素类型的名称，因此你无法写下set的满足各项要求的类型。为了绕开此类问题，你应该考虑如下两种技巧：

1. 将存在类型传入方法时，把类型参数从forSome语句移到方法的类型参数中。在方法体内，你可以用这个类型参数来指代本来在forSome语句中的类型。

2. 不要从方法返回存在类型，而是返回一个带有forSome语句中的每个类型的抽象成员的对象。（关于抽象成员，请参考第20章。）

使用这两个小技巧，之前的代码可以写成如下的样子：

```
import scala.collection.mutable.Set
import java.util.Collection

abstract class SetAndType{
  type Elem
```

```
  val set: Set[Elem]
}

def javaSet2ScalaSet[T](jset: Collection[T]): SetAndType= {
  val sset = Set.empty[T]  //现在我们可以用 T 这个名称了!

  val iter = jset.iterator
  while(iter.hasNext)
    sset += iter.next()

  return new SetAndType{
    type Elem = T
    val set = sset
  }
}
```

你现在能明白为什么 Scala 代码通常不使用存在类型了吧。为了用它们实现任何复杂些的东西，你会倾向于将它们转换成使用抽象成员，当然也可以从一开始就使用抽象成员。

29.4 小结

大多数时候，你可以忽略 Scala 是如何实现的，直接编写和运行代码就好了。不过有时候“看看盖子下面都有些什么”也不错，因此本章介绍了 Scala 在 Java 平台上的实现的三个方面：翻译是什么样子的，Scala 和 Java 的注解如何相互配合，以及如何使用 Scala 的存在类型访问 Java 的通配类型。当你需要同时使用 Scala 和 Java 时，这些话题是很重要的。

第 30 章

Actor 和并发

有时，指定事情相互独立、并行且同时发生，有助于我们更好设计程序。Java 包含了对并发的支持，虽然这样的支持足够满足基本的需要，在实际使用中发现，随着程序变得越来越大、越来越复杂，要把这件事做对并不容易。Scala 增加了 actor，对 Java 原生的并发支持进行了补充。Actor 提供了一种易于使用的并发模型，因而可以帮助你避免使用 Java 原生并发模型时会遇到的困难。本章将向你展示使用 Scala 的 actor 库的基础，并提供一个扩展的示例，将第 18 章中单线程的电路模拟代码转换成多线程的版本。

30.1 天堂中的烦恼

Java 平台自带了内建的基于共享数据和锁的线程模型。每个对象都关联了一个逻辑监视器(monitor)，可以用来控制对数据的多线程访问。使用这种模型需要由你来决定哪些数据将被多线程共享，并将访问共享数据或控制对这些数据访问的代码段标记为“synchronized”。Java 运行时将应用一种锁的机制来确保同一时间只有一个线程进入由同一个锁控制的同步的代码段，从而让你可以协同共享数据的多线程访问。

不幸的是，程序员们发现，要使用共享数据和锁模型来有把握地构建健壮的、多线程的应用程序十分困难。这当中的问题是，在程序中的每一点，你都必须推断出哪些是你正在修改或访问的数据可能会被其他线程修改或访问，以及在这一点上你握有哪些锁。每次方法调用，你都必须推断出它将会尝试握有哪些锁，并说服自己它这样做不会死锁。而在你推断中的这些锁，并不是在编译期就固定下来的，这让问题变得更加复杂，因为程序可以在运行期，在执行过程中任意创建新的锁。

更糟的是，对于多线程的代码而言，测试是不可靠的。由于线程是非确定性的，你可能测试 1000 次都是成功的，而程序第一次在客户的机器上运行就出问题。对共享数据和锁，你必须通过推断来把程序做对，别无他途。

不仅如此，你也无法通过过度的同步来解决问题。同步一切可能并不比什么都不同步更好。这中间的问题是尽管新的锁操作去掉了争用状况的可能，但同时也增加了死锁的可能。一个正确的使用锁的程序既不能存在争用状况，也不能有死锁，因此你不论往哪个方向做过头都是不安全的。

Java 5 引入了 `java.util.concurrent`，这是一组并发工具包，提供对并发编程更高级别的抽象。使用并发工具包来进行多线程编程比你用低级别的同步语法制作自己的抽象可能会带来的问题要少得多。

尽管如此，并发工具包也是基于共享数据和锁模型的，因而并没有从根本上解决使用这种模型的困难。

Scala的actor包则确实从根本上解决了这个问题，提供了另外一种不共享任何数据、依赖消息传递的模型，对程序员们而言，用这种模型，推断要容易得多。设计并发软件时，actor是首选的工具，因为它们能够帮助你避开死锁和争用状况，这两种情形都是在使用共享和锁模型时很容易遇到的。

30.2 actor和消息传递

actor是一个类似线程的实体，它有一个用来接收消息的邮箱。实现actor的方法是继承scala.actors.Actor并完成其act方法。清单30.1给出了一个示例。这个actor并不对它的邮箱做任何事，只是将一个消息打印五次然后退出。

584

```
import scala.actors._

object SillyActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("I'm acting!")
      Thread.sleep(1000)
    }
  }
}
```

清单30.1 一个简单的actor

你通过调用actor的start方法来启动它，这和你启动Java线程类似：

```
scala> SillyActor.start()
I'm acting!
res4: scala.actors.Actor = SillyActor$@1945696

scala> I'm acting!
I'm acting!
I'm acting!
I'm acting!
```

注意“I'm acting!”输出和Scala的命令行输出交错在了一起。这是因为SillyActor这个actor的运行独立于运行命令行的线程，actor在运行时也是相互独立的。举例来说，给定这样一个（第二版的）actor：

```
import scala.actors._

object SeriousActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("To be or not to be.")
      Thread.sleep(1000)
    }
  }
}
```

585

你可以同时运行两个actor，像这样：

```
scala> SillyActor.start(); SeriousActor.start()
res3: scala.actors.Actor = seriousActor$@1689405

scala> To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
```

你也可以用对象 scala.actor.Actor 中名为 actor 工具的方法来创建 actor：

```
scala> import scala.actors.Actor._

scala> val seriousActor2 = actor {
         for (i <- 1 to 5)
           println("That is the question.")
         Thread.sleep(1000)
       }

scala> That is the question.
That is the question.
That is the question.
That is the question.
That is the question.
```

上述 val 定义创建一个由执行 actor 方法后的代码块定义的 actor。这个 actor 在定义以后立即启动，无须另外调用 start 方法。

所有这些都没问题。你可以创建 actor，而它们也是独立运行的。但它们如何协作呢？它们如何在不使用共享内存和锁的前提下通信？Actor 通过相互发送消息的方式来通信。你可以使用 ! 方法来发送消息，就像这样：

```
scala> SillyActor ! "hi there"
```

这种情况下什么都不会发生，因为 SillyActor 忙于表演根本无暇顾及消息，因此“hi there”消息静静的躺在邮箱中，保持未读的状态。清单 30.2 给出了一个新的，更愿意与外界沟通的 actor，它等待邮箱收到消息，然后不论收到的是什么，都打印出来。它通过调用 receive 来接收消息，将局部函数作为参数传入[1]。

```
val echoActor = actor {
  while (true) {
    receive {
      case msg =>
        println("received message: "+ msg)
    }
```

[1]注：正如 15.7 节所讨论的，偏函数字面量用一系列的 match 备选项或“case”来表示。除了没有 match 关键字之外，它和 match 表达式长得很像。

```
    }
  }
```

清单30.2 调用receive的actor

当actor发送消息时，它并不会阻塞，而当actor接收到消息时，它也不会被打断。发送的消息在接收actor的邮箱中等待处理，直到actor调用receive方法。你可以从以下演示观察到这个行为：

```
scala> echoActor ! "hi there"
received message: hi there

scala> echoActor ! 15

scala> received message: 15
```

正如15.7节所讨论的，偏函数（PartialFunction特质的实例）并不是完整的函数——也就是说，它并不对所有输入值都有定义。除了接受单个参数的apply方法之外，偏函数还提供一个isDefinedAt方法，同样只接受单个参数。如果偏函数能够“处理”传给isDefinedAt的值，则这个方法返回true，这样的值传递给apply是安全的。如果你传递给apply的值会让isDefinedAt方法返回false，apply则会抛出一个异常。

actor只会处理传给receive方法的偏函数中的某个样本相匹配的消息。对邮箱中的每个消息，receive都会先调用传入的偏函数的isDefinedAt方法来决定它是否与某个样本匹配，然后处理该消息。receive方法将选定邮箱中第一个让isDefinedAt返回true的消息，将这个消息传递给偏函数的apply方法。而偏函数的apply方法将会处理这个消息。举例来说，echoActor的apply方法将打印“received message: ”，跟着是消息对象的toString结果。如果邮箱中没有让isDefinedAt返回true的消息，则被调用receive的actor将会阻塞，直到收到匹配的消息。

举例来说，以下是只处理Int类型的消息的actor：

```
scala> val intActor = actor {
         receive {
           case x: Int => // 我只要Int
             println("Got an Int: "+ x)
         }
       }
intActor: scala.actors.Actor =
  scala.actors.Actor$$anon$1@34ba6b
```

如果你传给它String或Double，intActor将会忽略这个消息，什么都不做：

```
scala> intActor ! "hello"
scala> intActor ! Math.Pi
```

但如果你传给它一个Int，你将会得到这样的响应，打印出：

```
scala> intActor ! 12
Got an Int: 12
```

30.3 将原生线程当作 actor

588

Actor 子系统会管理一个或多个原生线程供自己使用。只要你用的是你显式定义的 actor，就不需要关心它们和线程的对应关系是怎样的。

该子系统也支持反过来的情形：即每个原生线程也可以被当作 actor 来使用。不过，你不能直接使用 `Thread.current`，因为它并不具备必要的方法。你应该使用 `Actor.self` 来将当前的线程作为 actor 来查看。

这个设施对于在交互式命令行中调试 actor 尤为有用。这里有一个示例：

```
scala> import scala.actors.Actor._
import scala.actors.Actor._

scala> self ! "hello"

scala> self.receive { case x => x }
res6: Any = hello
```

这里的 `receive` 方法返回由传递给它的偏函数计算出的值。在本例中，偏函数返回消息本身，因此接收到的消息最终被解释器命令行打印出来。

如果你使用这项技巧，最好是用 `receive` 的变种 `receiveWithin`。以此方法你可以指定一个以毫秒计的超时时限。如果你在解释器命令行中使用 `receive` 的话，`receive` 将会阻塞命令行，直到有消息到来。对 `self.receive` 而言，这可能意味着永远等下去。你应该使用 `receiveWithin` 并给出超时值：

```
scala> self.receiveWithin(1000) { case x => x } // 等一秒钟!
res7: Any = TIMEOUT
```

30.4 通过重用线程获得更好的性能

Actor 是构建在普通 Java 线程之上的。就目前描述而言，实际上每个 actor 都必须得到自己的线程，这样每个 `act` 方法才能有机会运行。

不幸的是，虽然线程听上去很轻巧，Java 中线程的开销并不低。线程会消耗内存，对于一个典型的 Java 虚拟机，尽管可以有数以百万计的对象，却只可能有数千个线程。更糟的是，切换线程通常需要数百甚至数千条处理器指令。如果你想让程序尽可能高效，那么慎用线程的创建和切换就很重要了。

589

为帮助你节约线程，Scala 提供了一种通常的 `receive` 方法之外的备选方法，叫做 `react`。和 `receive` 一样，`react` 带一个偏函数。不同的是，`react` 在找到并处理消息后并不返回。它的返回类型是 Nothing。它对消息处理器求值，然后就结束了[2]。

[2]注：实际上在后台，`react` 将在完成时抛出一个异常。

由于react方法不需要返回，其实现不需要保留当前线程的调用栈。因此，actor库可以在下一个被唤醒的线程中重用当前的线程。这种方式十分高效，如果程序中的每个actor都使用react而不是receive的话，理论上只需要一个线程就能满足程序的全部actor的需要（确切地说，如果计算机有多个处理器核心，actor子系统将在可能的情况下使用足够多的线程来充分利用所有核心）。

> 实际当中，程序至少需要少量的几个用receive的actor，不过你应该尽可能使用react以节约线程。

由于react不返回，接收消息的消息处理器现在必须同时处理消息并执行actor所有余下的工作。通常的做法是用一个顶级的工作方法——比如act自身——供在息处理器在完成时调用。清单30.3给出了一个使用这种方式的示例。

清单30.3中所示的actor等待内容为主机名的字符串，如果有的话，返回那个主机名对应的IP地址。这里有一个示例：

```
scala> NameResolver.start()
res0: scala.actors.Actor = NameResolver$@90d6c5

scala> NameResolver ! ("www.scala-lang.org", self)

scala> self.receiveWithin(0) { case x => x }
res2: Any = Some(www.scala-lang.org/128.178.154.102)

scala> NameResolver ! ("wwwwww.scala-lang.org", self)

scala> self.receiveWithin(0) { case x => x }
res4: Any = None
```

590

```
object NameResolver extends Actor {
  import java.net.{InetAddress, UnknownHostException}

  def act() {
    react {
      case (name: String, actor: Actor) =>
        actor ! getIp(name)
        act()
      case"EXIT" =>
        println("Name resolver exiting.")
        // 退出
      case msg =>
        println("Unhandled message: "+ msg)
        act()
    }
  }

  def getIp(name: String): Option[InetAddress] = {
    try {
      Some(InetAddress.getByName(name))
    } catch {
```

```
        case _:UnknownHostException => None
      }
    }
  }
```

清单 30.3 调用 react 的 actor

编写使用 react 而非 receive 的 actor 颇具挑战，不过在性能上能够带来相当的回报。由于 react 不需要返回，作为调用方的 actor 的调用栈可以被销毁，将线程的资源释放供另一个 actor 使用。极端情况下，如果程序中所有 actor 都使用 react，则它们可以用单个线程实现。

这样一种编码模式对于基于事件的 actor 而言是如此常见，以至于 actor 库对此有特别的支持。Actor.loop 函数重复执行一个代码块，哪怕代码调用的是 react。我们可以用 loop 来重写 NameResolver 的 act 方法，如清单 30.4 所示。在行为上这与清单 30.3 的不同之处在于它并不以退出来响应“EXIT”消息 。相反，这个 actor 将会循环响应消息直到永远。

```
  def act() {
    loop {
      react {
        case (name: String, actor: Actor) =>
          actor ! getIp(name)
        case msg =>
          println("Unhandled message: " + msg)
      }
    }
  }
```

清单 30.4 使用 loop 的 actor 方法 act

30.5 良好的 actor 风格

至此你已经见识了所有编写你自己的 actor 所需要的内容。不过，简单地用这些方法也就只能带你走这么远。它们之所以被创建出来，是为了支持 actor 风格的并发编程。你的代码是否更容易调试并且死锁和争用状况更少，这取决于你能够把这种风格用到什么程度 。本节提供一些 actor 风格编程的指导意见。

Actor 不应阻塞

编写良好的 actor 在处理消息时并不阻塞。阻塞的问题是，在 actor 阻塞时，另一个 actor 可能会对其发起一个它能够处理的请求。如果 actor 在首个请求时阻塞了，那么它将不会留意到第二个请求。最坏的情形是这可能带来死锁，多个 actor 都在等待另一个阻塞的 actor 的响应。

与阻塞相反，actor 应该允许某种表示动作可以执行的消息发送给它。通常这样的安排将需要其他 actor 的帮助。举例来说，用调用 Thread.sleep 代替并阻塞当前的 actor，你可以创建一个助手 actor，这个 actor 睡眠并在一定时间后发回一个消息：

react的工作原理

Nothing这个返回类型表明这是一个永远不会正常返回的函数，相反，它总是以一个异常的方式完成。的确，对于react而言就是这么回事。实际上react的实现并非像下面描述的这样简单，而且还会有新的变化，但概念上讲，你可以认为react是这样工作的：

当你调用一个actor的start时，start方法会以某种方式来确保最终会有某个线程来调用那个actor的act方法。如果act方法调用了react，则react方法会在actor的邮箱中查找传递给偏函数的能够处理的消息。（和receive方法一样，传递待处理消息给偏函数的isDefinedAt方法。）如果找到一个可以处理的消息，react会安排一个在未来某个时间处理该消息的计划并抛出异常。如果它没有找到这样的消息，它会将actor置于“冷存储”状态，在它通过邮箱收到更多消息时重新激活，并抛出异常。不论是哪种情况，react都会以这个异常的方式完成其执行，act方法也随之结束。调用act的线程会捕获这个异常，忘掉这个actor，并继续处理其他事务。

这就是为什么你想要react在处理第一个消息之外做更多的事，你将需要在偏函数中再次调用act方法，或使用某种其他的手段来让react再次被调用。

```
actor {
  Thread.sleep(time)
  mainActor ! "WAKEUP"
}
```

这个助手actor的确阻塞了，但由于它永远不会收到消息，因此在这种情况下是OK的。主actor可以继续响应新的请求。清单30.5所示的emoteLater方法展示了这种处理方式的用法。它创建一个新的actor来执行sleep以便主actor不阻塞。以确保它向正确的actor发送“Emote”消息，我们必须小心地在主actor中对self求值而不是在助手actor中。

593

```
val sillyActor2 = actor {
  def emoteLater() {
    val mainActor = self
    actor {
      Thread.sleep(1000)
      mainActor ! "Emote"
    }
  }

  var emoted = 0
  emoteLater()

  loop {
    react {
      case "Emote" =>
        println("I'm acting!")
        emoted += 1
        if (emoted <5)
          emoteLater()
      case msg =>
        println("Received: "+ msg)
```

```
      }
    }
  }
```

清单 30.5 使用助手 actor 来避免阻塞自己的 actor

由于这个 actor 并不在 sleep 方法中阻塞——它的助手 actor 会阻塞——它可以在等待下次表演之前继续做其他事。与早先的蠢 actor[3]不同，这个 actor 会在等待下一个输出的同时继续打印出消息：

```
scala> sillyActor2 ! "hi there"
scala> Received: hi there
I'm acting!
I'm acting!
I'm acting!
```


只通过消息与 actor 通信

Actor 模型解决共享数据和锁的关键方法是提供一个安全的空间——actor 的 act 方法——在这里你可以顺序地思考。换个说法就是，actor 让你可以像一组独立的通过异步消息传递来进行交互的单线程的程序那样编写多线程的程序。不过，像这样的简化描述只在消息是你的 actor 的唯一通信途径的前提下才成立[4]。

举例来说，一个 GoodActor 可能会在发往 BadActor 的消息中包含一个指向自己的引用，来表明作为消息源的自己。如果 BadActor 调用了 GoodActor 的某个任意的方法而不是通过!发送消息的话，问题就来了。被调用的方法可能读到 GoodActor 的私有实例数据，而这些数据可能是由另一个线程写进去的。结果是，你需要确保 BadActor 线程对这些实例数据的读取和 GoodActor 线程对这些实例数据的写入是同步在同一个锁上的。GoodActor 的私有实例数据就成了必须由锁守卫起来的共享数据。因此，一旦你绕过了 actor 之间的消息传递机制，你就回到了共享数据和锁的模型中，所有那些你想用 actor 模型避开的困难又都回来了。

另一方面，这并不是说你应该完全避免绕开消息传递的做法。虽然共享数据和锁要做正确很难，但也并不是完全不可能。Scala 的 actor 和 Erlang 的 actor 实现方式的区别之一，实际上就是 Scala 让你可以在同一个程序中混用 actor 与共享数据和锁两种模型。

作为示例，想象你想要多个 actor 共享一个公用的可变映射。由于映射是可变的，纯 actor 方式可能是创建一个“拥有”可变映射的 actor 并定义一组允许其他 actor 访问这个映射的消息。你可以定义将键-值对放入共享映射的消息，获取给定键的值的消息等等，用来执行所有你需要对映射执行的操作。此外，你还会需要定义用于向发起查询请求的 actor 发送异步响应的消息。然而，另外一种选择是通过消息向多个 actor 传递一个线程安全的映射，比如 Java 并发包中的 ConcurrentHashMap，让这些 actor 直接使用那个映射。

尽管通过 actor 实现共享映射比你自己实现类似 ConcurrentHashMap 这样的类要简单和安全得

[3]译注：即 Silly Actor。

[4]注：另一个好处是消息发送确保了消息对象被安全地发布给其他线程，如 Goetz 等所著的“Java Concurrency in Practice”第 49 页所述。[Goe06]

多，由于ConcurrentHashMap已经存在，你可能会认为使用它比你自己通过actor实现共享映射更简单，风险也更低。这也意味着你的共享映射的响应是同步的，而用actor的话响应是异步的。Scala的actor类库让你可以有所选择。

如果你考虑用共享数据和锁

当你考虑是否要混用actor模型与共享数据和锁模型时，回想一下Clint Eastwood在1971年的电影Dirty Harry中扮演的Harrt Callahan的话是很有帮助的：

我知道你在想什么。“他开了六枪还是只开了五枪？”实话告诉你吧，我兴奋得自己也有点记不清了。不过这可是一把.44的Magnum，全世界威力最大的手枪，可以把你的头整个嘣掉，你得问问自己：我今天走运么？你走运么，人渣？

优选不可变消息

由于Scala的actor模型提供了在每个actor的act方法中的单线程环境，你不需要担心在这个方法的实现中使用的对象是否是线程安全的。例如你可以在act方法中放心的使用非同步的、可变的对象，因为每个act方法实际上被局限在一个线程中[5]。这就是为什么actor模型被称作什么都不共享（share-nothing）的模型——数据局限于一个线程中，而不是被多个线程共享。

不过这个什么都不共享的原则有一个例外：用于在actor间发送消息的对象中的数据由多个actor“共享”。结果是，你确实需要担心消息对象是否线程安全。通常来说，它们也应该是这样。

确保消息对象是线程安全的最佳途径是在消息中只使用不可变对象。任何只有val字段且这些字段只引用到不可变对象的类的实例都是不可变的。当然，一个简单的定义这样的消息类的方式是把它们定义为样本类。只要你不显式地给样本类添加var字段，并确保val字段均为不可变类型的，则样本类按照定义就是不可变的。这对于在传递给react或receive的偏函数中使用模式匹配而言也很方便。你也可以使用你自己定义的常规的（非样本类）的不可变类来作为消息。或者你也可以使用Scala API提供的各种不可变类的实例，比如元组、字符串、列表、不可变集、映射等。

话说回来，如果actor发送的是可变的、未同步的对象作为消息，并在此以后永不读或写这个对象，也没问题，但这摆明了是嫌麻烦还不够多。未来的维护者可能意识不到这是个共享的对象，然后对它进行写操作，这样就会产生难以找到的并发bug。

总体来说，最好将你的数据像这样来组织，使得每个非同步的、可变的对象只被一个actor“拥有”和访问。如果愿意，你可以让对象从一个actor转移到另一个actor，不过你需要确保在任何时间点上，哪个actor拥有这个对象并允许访问这个对象，都是清楚的。换句话说，当你设计一个基于actor的系统时，你需要决定哪部分的可变内存是指派给哪个actor的。其他需要访问可变数据结构的actor必须向这个数据结构的所有者发送消息并等待回复返回的消息。

[5]注：使用react时，不同的消息会被潜在的不同的线程处理，尽管这样，它们仍将被依次处理，并有足够多的同步机制来让你按照每个act方法只限于单个线程这样的简化的假设来进行编程。

如果你发现自己有一个可变的对象，你想继续使用它，同时也想用消息发送给另一个 actor，你应该制作并发送它的一个副本。这样做的时候，你可能想要让它变成不可变的。举例来说，由于数组是可变且非同步的，你使用的任何数组在同一时间都应该只被一个 actor 访问。如果你想继续使用某个数组同时将它发送给另一个 actor 的话，你应该发送一个副本。例如，如果数组自身只包含不可变对象，你可以用 `arr.clone` 来制作副本。不过你也应该考虑要不要使用 `arr.toList`，并发送由此得到的不可变列表。

不可变对象在很多情形下都很方便，不过让它们真正发光的是并行系统，因为它们是设计线程安全对象的最简单同时风险也最低的方式。当你设计一个未来可能会牵涉到并行计算的程序时，不论是否使用 actor，你都应该花大力气尽量使用不可变的数据结构。

让消息自包含

当你从方法中返回一个值时，调用者所处的位置最便于记住在调用该方法之前它正在做什么。它可以取走这个响应的值然后继续它在做的事。

对 actor 而言，事情变得更复杂了。当一个 actor 向另一个发起请求时，响应可能很长时间都不会到来。作为调用方的 actor 不应阻塞，而是在等待响应时继续做任何它能做的其他事。这样当响应最终回来之后要解释它就有难度了。这个 actor 还能记得当它发起请求时在做什么吗？

一种简化 actor 程序逻辑的方法是在消息中包含冗余的信息。如果请求是个不可变的对象，你甚至可以花费很小的代价在返回值中包含一个指向该请求的引用！举例来说，IP 查询 actor 如果在返回 IP 地址的同时也返回相应的主机名就更好。这可能让这个 actor 变得稍长一些，不过它应该能简化任何向它发起请求的 actor 的逻辑：

```
def act() {
  loop {
    react {
      case (name: String, actor: Actor) =>
        actor ! (name, getIp(name))
    }
  }
}
```

另一种在消息中增加冗余的方式是为每一类消息制作样本类。虽然在很多情况下这样一个包装类严格来讲并不是必须的，但它让 actor 程序理解起来简单很多。想象一下一个程序员看着这样一个字符串的发送过程，如：

```
lookerUpper ! ("www.scala-lang.org", self)
```

要搞明白代码中哪些 actor 可能会响应是很困难的。如果代码看上去是这样就简单多了：

```
case class LookupIP(hostname: String, requester: Actor)

lookerUpper ! LookupIP("www.scala-lang.org", self)
```

现在，程序员可以在源码中查找 LookupIP，但很可能只会找到很少几个响应者。清单 30.6 显示了一个更新后的名字解析 actor，它用样本类而不是元组来作为消息。

```
import scala.actors.Actor._
import java.net.{InetAddress, UnknownHostException}

case class LookupIP(name: String, respondTo: Actor)
case class LookupResult(
  name: String,
  address: Option[InetAddress]
)

object NameResolver2 extends Actor {

  def act() {
    loop {
      react {
        case LookupIP(name, actor) =>
          actor ! LookupResult(name, getIp(name))
      }
    }
  }

  def getIp(name: String): Option[InetAddress] = {
    // 和之前一样（同清单 30.3）
  }
}
```

清单 30.6 使用样本类作为消息的 actor

30.6 更长一些的示例：并行离散事件模拟

作为一个更长一些的示例，假定你想要将第 18 章的离散事件模拟并行化。模拟当中的每个参与者都可以运行为自己的 actor，从而让你可以通过使用多个处理器来加速模拟。本节将使用基于由 Philipp Haller 开发的并行电路模拟器代码带你完整地过一遍整个过程。

总体设计

第 18 章的大部分设计对于顺序的或并行的离散事件模拟都没问题。首先有事件，它们按照既定的时间发生，处理一个事件可能会造成新的事件被排程，如此往复。同样，电路模拟也可以按事件处理来实现，将门和线路作为模拟的参与者，而将线路的变化作为模拟中的事件。让事件并行地运行会是个不错的改变。我们该如何重组设计来实现它呢？

核心的想法是让每个被模拟的对象都成为 actor。每个事件都由 actor 处理，事件的大部分状态都位于 actor 中。对电路模拟而言，对门的输出的更新可以由那个门所对应的 actor 处理。这样，事件就可以很自然地被并行处理。

用代码来表示的话，很可能不同的被模拟的对象之间会有一些共通的行为。于是定义一个能混入任

何类的 Simulant 特质来做出一个模拟对象就很合理了。线路、门和其他模拟对象可以加上这个特质。

```
trait Simulant extends Actor
class Wire extends Simulant
```

目前一切都还好，不过我们需要解决一些设计问题，这些问题当中有不少都没有明显的最佳答案。对本章而言，我们针对这些设计问题做出的合理选择都是为了保持代码精简。不过也有其他一些可能的解决方案，尝试这些方案对于任何想要得到 actor 编程相关经验的人而言都是不错的实践。

第一个设计问题是搞明白如何让模拟的参与者保持与模拟的时间同步。也就是说，参与者 A 在所有其他 actor 完成时间点 99 之前不应该急着处理时间点 100 的事件。为了搞清楚为什么这一点很重要，想象一下模拟对象 A 还停留在时间点 90，而模拟对象 B 在时间点 100。有可能参与者 A 正打算发送一个改变 B 在时间点 91 的状态的消息。B 知晓这个消息时已经太晚了，因为它已经处理完了 92 到 99 的时间点。为避免这个问题发生，本章所采用的设计方式是任何模拟对象在所有其他模拟对象完成了对时间 n-1 的处理之前，都不应处理时间 n 的事件。

600

不过这个决定也带来了新的问题：模拟对象如何知道什么时候可以安全地往前走呢？直截了当的一种方式是用一个“时钟” actor 来跟踪当前的时间并在该前进的时候告诉模拟的参与者。为防止这个时钟在所有模拟对象准备好之前就前进，时钟 actor 可以在仔细选定的时间 ping 其他 actor 来确保它们收到并处理了当前时间点的所有消息。将会有时钟发给模拟对象的 Ping 消息，以及模拟对象在准备好让时钟前进时发回的 Pong 消息。

```
case class Ping(time: Int)
case class Pong(time: Int, from: Actor)
```

注意这些消息也可以被定义为不带字段的。不过，time 和 from 字段为系统增加了一些冗余。time 字段握有 ping 的时间，可以用来将 Pong 和与之相关联的 Ping 连接起来。from 字段是 Pong 的发送者。Ping 的发送者始终是时钟，因此它没有 from 字段。如果程序运转良好，所有这些信息都是不需要的，不过它能够在某些环节简化逻辑，如果程序遇到任何错误它也能在很大程度上帮助我们调试。

产生的一个问题是模拟对象如何知道它完成了当前时间点的工作。模拟对象在完成当前时间点的所有工作之前不应该响应 Ping 消息，不过它们怎么知道呢？可能另一个 actor 向它发出了请求但还没有送到，可能某个 actor 的消息被送给另一个 actor 还没有被处理。

增加两个限制条件会简化该问题的答案。首先，假定模拟对象从不直接相互发送消息，而只是相互安排事件日程。其次，它们从不向当前时间点提交事件，而是向未来至少比当前多 1 点的时间提交。这两个限制条件带来的影响不小，不过它们看上去对于典型的模拟而言是可以接受的。毕竟，通常每当系统中的两个组件相互交互时总会有些传递的延迟。更进一步讲，在最坏的情况下，时间点也可以有更短的时间间隔，未来需要的信息可以事先发送。

另一种处理方式也是可行的。我们可以允许模拟对象直接相互发消息。不过，如果它们这样做了，那么就需要一个更精巧的机制来决定一个 actor 何时可以安全的送回一个 Pong。每个模拟对象都应该延迟响应 Ping 消息直到它请求的所有其他模拟对象都完成了对请求的处理。为了保证这一点，你会需要模拟对象相互间传递一些额外的信息。就现在而言，假定模拟对象除了通过整个模拟的日程表外并不相

601

互通信。

有了这个决定，就意味着还有一个工作项的日程表，这个日程表可能也是由时钟 actor 所持有的。这样，时钟 actor 就可以等到当前时间点的所有工作项的请求都发送完成后再发出 Ping 消息。Actor 们则知道当它们收到 Ping 时，它们已经从时钟那里收到了所有当前时间点需要发生的工作项。因此当 actor 收到 Ping 时可以安全的立即回送一个 Pong，因为当前时间点不会再有更多的工作发过来了。采用这种方式，一个 Clock 有如下的状态：

```
class Clock extends Actor {
  private var running = false
  private var currentTime = 0
  private var agenda: List[WorkItem] = List()
}
```

最后一个需要解决的设计问题是如何设置好一个初始的模拟场景。一种自然而然的方式是在时钟停止的状态下创建模拟，添加所有的模拟对象，将它们连在一起，然后启动时钟。这当中微妙的地方在于你需要绝对的确认所有东西都连好了才能运行时钟！否则，模拟的某些部分将会在完全成形之前开始运行。

你如何知道模拟在什么时候被完全组装好并可以启动了呢？这一次仍然有多个途径来解决这个问题。本章所采用的简单办法是在设置好模拟的过程中不向 actor 发送消息。这样，一旦最后一个消息调用返回，你就知道模拟被完整的构建好了。最终的编码模式是你使用常规的方法调用将模拟设置好，然后你在模拟运行时使用 actor 发送消息。

有了前面的这些决定，剩下的设计就很直截了当了。WorkItem 可以基本像第 18 章那样定义，相通之处是它握有一个时间和一个动作。不过对并行模拟而言，动作本身有另外的编码方式。在第 18 章中，动作由无参数的函数表示，而在并行模拟中使用目标 actor 和发往该 actor 的消息会更加自然。

602

```
case class WorkItem(time: Int, msg: Any, target: Actor)
```

同理，用于安排新工作项的 afterDelay 方法变成了可以发往时钟的 AfterDelay 消息。正如 WorkItem 类那样，无参数的动作函数被替换为一个消息和一个目标 actor：

```
case class AfterDelay(delay: Int, msg: Any, target: Actor)
```

最后，用于要求模拟启动和停止的消息也会很有用：

```
case object Start
case object Stop
```

总体设计就是这些。有一个 Clock 类握有当前的时间和日程表，且时钟只在它 ping 了所有模拟对象并确保它们准备好以后才往前进。有一个 Simulant 特质来模拟参与者，它们通过向时钟发送工作项增加到日程表的方式来和其他模拟对象通信。下一节将会介绍如何实现这些核心类。

实现模拟框架

核心框架有两个东西需要实现：Clock 类和 Simulant 特质。首先来考虑 Clock 类。必要的时钟

状态如下：

```
class Clock extends Actor {
  private var running = false
  private var currentTime = 0
  private var agenda: List[WorkItem] = List()
  private var allSimulants: List[Actor] = List()
  private var busySimulants: Set[Actor] = Set.empty
```

一开始时钟的 running 被设为 false。一旦模拟全部初始化完成，时钟会收到 Start 消息，running 会变成 true。这样，模拟在其所有组成部分被按照要求连在一起之前都保持冻结状态。这也意味着，由于所有的模拟对象也都是冻结的，使用常规的方法调用来配置它们是安全的，而不需要发送 actor 消息。

时钟也可以在创建好之后就作为 actor 开始运行。这是安全的，因为它在收到 Strat 消息之前实际上不会做任何事。

```
start()
```

时钟也跟踪记录当前的时间（currentTime）、由这个时钟管理的参与者列表（allSimulants）、以及仍然在当前时间点工作的参与者列表（busySimulants）。之所以用列表来放 allSimulants，是因为它只会被遍历，而 busySimulant 之所以用集，是因为项目会以一种不可预期的顺序从中去掉。一旦模拟器开始运行，它只有当 busySimulants 为空时才会前进到新的时间，只要它将时钟前进一格，它都会将 busySimulants 设为 allSimulants 的内容。

为设置好模拟场景，将会需要有一个方法来为时钟添加新的模拟对象。现在就加上吧：

```
def add(sim: Simulant) {
  allSimulants = sim :: allSimulants
}
```

以上是时钟的状态。下面来看时钟的活动。它的主循环在两个职责间交替变换：让时钟前进，以及响应消息。一旦时钟前进，它只能在至少收到一个消息时才能再次前进，因此将主循环定义为这两个活动间的交替变换是安全的：

```
def act() {
  loop {
    if (running && busySimulants.isEmpty)
      advance()

    reactToOneMessage()
  }
}
```

时间的推进除了简单的 currentTime 步进之外还有一些额外的内容。首先，如果日程表是空的，且模拟并非刚开始，那么模拟需要退出。其次，假定日程表非空，则所有 currentTime 这个时间点的工作项需要现在发生。

再次，所有模拟对象都必须被放到 busySimulant 中，并被发送 Ping 消息。直到所有 Ping 被

响应前，时钟都不会再次前进：

```
def advance() {
  if (agenda.isEmpty && currentTime > 0) {
    println("** Agenda empty. Clock exiting at time "+
            currentTime+".")
    self ! Stop
    return
  }

  currentTime += 1
  println("Advancing to time "+currentTime)

  processCurrentEvents()
  for (sim <- allSimulants)
    sim ! Ping(currentTime)

  busySimulants = Set.empty ++ allSimulants
}
```

处理当前的事件只不过是简单地处理所有在日程表最上方，时间为 currentTime 的事件：

```
private def processCurrentEvents() {
  val todoNow = agenda.takeWhile(_.time <= currentTime)

  agenda = agenda.drop(todoNow.length)

  for (WorkItem(time, msg, target) <- todoNow) {
    assert(time == currentTime)
    target ! msg
  }
}
```

这个方法中有三步。首先，我们用 takeWhile 将所有在当前时间需要发生的条目选出并存入 val todoNow。然后，我们用 drop 将这些条目从日程表中去掉。最后，我们遍历现在要做的条目，并向它们发送目标消息。包含的 assert 只是为了确保排程器的逻辑没有问题。

有了这些基础工作，处理时钟可能会收到的消息就再简单不过了。AfterDelay 消息将新的条目添加到工作队列；Pong 消息从忙碌的模拟对象列表中去除一个模拟对象；Start 让模拟开始；Stop 则让时钟停止。

```
def reactToOneMessage() {
  react {
    case AfterDelay(delay, msg, target) =>
      val item = WorkItem(currentTime + delay, msg, target)
      agenda = insert(agenda, item)

    case Pong(time, sim) =>
      assert(time == currentTime)
      assert(busySimulants contains sim)
      busySimulants -= sim

    case Start => running = true

    case Stop =>
      for (sim <- allSimulants)
        sim ! Stop
```

```
        exit()
    }
  }
```

没有列出的 insert 方法和清单 18.8 完全一样。它将它的参数插入到日程表，并小心翼翼地保持日程表的排序不被打乱。

这些就是 Clock 的完整实现了，现在考虑如何实现 Simulant 吧。从本质上来说，一个 Simulant 可以是理解模拟消息 Stop 和 Ping 并与它们合作的任何 actor。它的 act 方法因而可以简单如下所示：

```
def act() {
  loop {
    react {
      case Stop => exit()
      case Ping(time) =>
        if (time == 1) simStarting()
        clock ! Pong(time, self)
      case msg =>handleSimMessage(msg)
    }
  }
}
```


每当模拟对象收到 Stop，它就退出。如果收到 Ping，它就响应一个 Pong。如果 Ping 的时间为 1，则在送回 Pong 之前调用 simStarting，让子类可以定义当模拟开始运行时应发生的行为。任何其他消息都必须由子类翻译，因此实际逻辑是推给抽象的 handleSimMessage 方法完成的。

模拟对象有两个抽象成员：handleSimMessage 和 clock。模拟对象必须知道它的时钟以便它可以回复 Ping 消息并安排新的工作项。所有内容放在一起，Simulant 特质如清单 30.7 所示。注意模拟对象在创建后直接启动运行。这是安全且方便的，因为它实际上在它的时钟发送消息给它之前不会做

```
trait Simulant extends Actor {
  val clock: Clock
  def handleSimMessage(msg: Any)
  def simStarting() { }
  def act() {
    loop {
      react {
        case Stop => exit()
        case Ping(time) =>
          if (time == 1) simStarting()
          clock ! Pong(time, self)
        case msg => handleSimMessage(msg)
      }
    }
  }
  start()
}
```

清单 30.7　Simulant 特质

任何事，且在模拟开始、时钟收到 Start 消息之前，这些[6]都不会发生。

以上就是并行事件模拟的框架了。和它在第 18 章的顺序版[7]一样，它所需要的代码出人意料的少。

607

实现一个电路模拟

既然模拟框架已经完成，是时候处理电路的实现了。电路中会有一些线路和门，它们是模拟对象，还有一个时钟用来管理这个模拟。线路握着一个布尔型的信号——可能是高（true）或者低（false）。门上则连接了多根线路，有些是输入，其他是输出。门基于输入线路的状态为输出线路计算出一个信号。

由于电路中的线路、门等只用于这个特定的电路，它们的类可以被定义为 Circuit 类的成员，就像 20.9 节的货币对象那样。因此总体来说 Circuit 类会有这样一些成员：

```
class Circuit {
  val clock = new Clock
  // 模拟消息
  // 延迟常量
  // Wire 类和 Gate 类及其方法
  // 其他工具方法
}
```

现在一组一组地看这些成员。首先是模拟消息。一旦模拟开始运行，线路和门就只能通过发送消息来通信，因此它们需要对每种想相互发送的信息都定义一个消息类型。这样的消息只有两种。门需要告诉它们的输出线路更改状态，而线路需要告诉那些以它们为输入线路的门它们的状态改变了：

```
case class SetSignal(sig: Boolean)
case class SignalChanged(wire: Wire, sig: Boolean)
```

接下来，有多种延迟需要选择。任何通过模拟框架排程的工作项——包括向线路上传出或从线路上获取的信号——都必须安排在未来的某个时间。确切的延迟并不明确，因此延迟不妨放在 val 中。这样，就容易在未来调整它们。

608

```
val WireDelay = 1
val InverterDelay = 2
val OrGateDelay = 3
val AndGateDelay = 3
```

至此，是时候看 Wire 类和 Gate 类了。先考虑线路。线路是拥有当前信号状态（高或低）和观察着该状态的门的列表的模拟对象。它混入了 Simulant 特质，因而也需要指定一个使用的时钟：

```
class Wire(name: String, init: Boolean) extends Simulant {
  def this(name: String) { this(name, false) }
  def this() { this("unnamed") }

  val clock = Circuit.this.clock
  clock.add(this)
```

[6]译注：指时钟发送消息给模拟对象。

[7]译注：指单线程版。

```
  private var sigVal = init
  private var observers: List[Actor] = List()
```

这个类也需要一个 handleSimMessage 方法来指定它应如何响应模拟消息。线路应该收到的唯一消息是 SetSignal，这个消息用来代表线路的信号。而响应应该是，当信号与当前的信号不同时，当前的状态改变，然后新的信号被发布出去：

```
def handleSimMessage(msg: Any) {
  msg match {
    case SetSignal(s) =>
      if (s != sigVal) {
        sigVal = s
        signalObservers()
      }
  }
}

def signalObservers() {
  for (obs <- observers)
    clock ! AfterDelay(
      WireDelay,
      SignalChanged(this, sigVal),
      obs)
}
```

上述代码显示了线路的信号改变是如何被传递到任何观察着它的门的。将线路的初始状态传给任何作为观察者的门也是很重要的。这只须在模拟启动时做一次。在那之后，门只须简单地保存它们收到的最近一次 SignalChanged 消息结果即可。而在模拟开始时发送初始信号也只须简单地提供一个 simStarting()方法：

```
override def simStarting() { signalObservers() }
```

现在关于线路只剩下一些小细节没讲到了。线路需要一个方法来连接新的门，另外它们可能会用到一个不错的 toString 方法：

```
def addObserver(obs: Actor) {
  observers = obs :: observers
}

override def toString = "Wire("+ name +")"
```

这些就是你所需要的关于线路的一切。现在来考虑门，这是电路中另一个主要的对象类别。总共有三种可以定义的基础门：与门（And）、或门（Or）和非门（Not）。所有这些都有很多行为是共通的，因此值得我们定义一个抽象的 Gate 类来保有这些共性。

定义这个 Gate 类的一个困难之处是某些门有两个输入线路（And、Or）而其他门只有一个（Not）。将这个区别显式地建模并非不可能，但是将所有门都想象为有两个输入会简化代码，非门只不过是忽略第二个输入罢了。被忽略的第二个输入可以设为某种假的线路，永远不会从 false 状态变成其他状态。

```
private object DummyWire extends Wire("dummy")
```

有了这个技巧，门的类就可以很简单直接做出来。它混入 Simulant 特质，且它唯一的构造方法会接受两个输入线路和一个输出线路：

```
abstract class Gate(in1: Wire, in2: Wire, out: Wire)
    extends Simulant {
```

610

Gate 有两个抽象成员，是它具体的子类必须填充的。最明显的是不同类型的门用不同的函数计算它们的输入。因此，应该要有一个抽象的方法用来基于输入计算输出：

```
def computeOutput(s1: Boolean, s2: Boolean): Boolean
```

其次，不同类型的门有不同的传递延迟。因此，门的延迟应该是一个抽象的 val：

```
val delay: Int
```

延迟可以是 def，但把它做成 val 表明这样一个事实：某个特定的门的延迟不应该被修改。

由于 Gate 加上了 Simulant 特质，我们必须指定它使用的时钟。和 Wire 类似，Gate 应指定包含它的 Circuit 的时钟。为了方便，Gate 可以在创建好之后将自己添加到时钟上。

```
val clock = Circuit.this.clock
clock.add(this)
```

同理，把门和两个输入线路连起来也是合理的，像下面这样，使用常规的方法调用：

```
in1.addObserver(this)
in2.addObserver(this)
```

门唯一的本地状态是它的每一个输入线路最近的一次信号。这个状态需要保存，因为线路只会在状态改变时发送信号。如果一个输出线路改变了，只有这个线路的状态会被发送给门，但新的输出需要从两条线路的状态来计算：

```
var s1, s2 = false
```

现在来看门如何响应模拟消息。它们只有一种消息需要处理，即表示某个输入线路已改变的 SignalChanged 消息。当 SignalChanged 消息到来时，我们要做两件事。首先，本地的线路状态需要根据这个改变更新。其次，新的输出需要计算并通过 SetSignal 消息将输出发送到输出线路：

611

```
def handleSimMessage(msg: Any) {
  msg match {
    case SignalChanged(w, sig) =>
      if (w == in1)
        s1 = sig
      if (w == in2)
        s2 = sig
      clock ! AfterDelay(delay,
          SetSignal(computeOutput(s1, s2)),
          out)
  }
}
```

有了这个抽象的 Gate 类，现在要定义特定的门的类型就很容易了。和第 18 章的顺序模拟类似，门的创建可以作为调用某个工具方法的副产物。方法需要做的就是创建一个 Gate 并填充合适的延迟和

输出计算。其他一切都是所有门共有的，在 Gate 类中处理掉了：

```
def orGate(in1: Wire, in2: Wire, output: Wire) =
  new Gate(in1, in2, output) {
    val delay = OrGateDelay
    def computeOutput(s1: Boolean, s2: Boolean) = s1 || s2
  }

def andGate(in1: Wire, in2: Wire, output: Wire) =
  new Gate(in1, in2, output) {
    val delay = AndGateDelay
    def computeOutput(s1: Boolean, s2: Boolean) = s1 && s2
  }
```

对非门而言，假的线路会被作为第二个输入值。这对创建非门的调用者而言是不需要关心的实现细节，因此 inverter 方法只接受一条输入线路而非两条：

```
def inverter(input: Wire, output: Wire) =
  new Gate(input, DummyWire, output) {
    val delay = InverterDelay
    def computeOutput(s1: Boolean, ignored: Boolean) = !s1
  }
```

612

至此我们的库已经可以模拟电路了，不过，正如第 18 章所述，增加一个线路状态查看工具是很有用的，这样你就可以观察电路的演进。没有这样的工具，模拟将不会知道哪条线路值得记录日志，而哪些又更像是实现细节。

定义一个接受 Wire 作为参数并在线路的信号改变时打印出一行文本的 probe 方法。这个方法可以通过简单地制作一个新的将自己连接到指定线路的模拟对象来实现。这个模拟对象可以通过打印出新的信号来响应 SignalChanged 消息：

```
def probe(wire: Wire) = new Simulant {
  val clock = Circuit.this.clock
  clock.add(this)
  wire.addObserver(this)
  def handleSimMessage(msg: Any) {
    msg match {
      case SignalChanged(w, s) =>
        println("signal "+ w +" changed to "+ s)
    }
  }
}
```

以上就是 Circuit 类的大体内容了。调用者应创建一个 Circuit 的实例，创建一组线路和门，对一些感兴趣的线路调用 probe，然后开始运行模拟。我们没有讲到的一块是如何启动模拟，它很简单，只是向时钟发送 Start 消息而已：

```
def start() { clock ! Start }
```

如之前在第 18 章解释过的，更复杂的电路组件也可以用这些方法构建出来。举例来说，清单 30.8 再次给出了当时介绍的半加法器和全加法器组件。它们的实现基本保持不变，但有小小的变化是它们现

613

在被打包在名为 Adders 的特质中，而在第 18 章它们是包含在一个抽象类中。由于特质被标记为扩展 Circuit，它可以直接访问 Circuit 的成员如 Wire 和 orGate。使用特质的方法则是类似这样：

```
trait Adders extends Circuit {
  def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {
    val d, e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)
  }

  def fullAdder(a: Wire, b: Wire, cin: Wire,
      sum: Wire, cout: Wire) {
    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
}
```

清单 30.8 加法器组件

```
val circuit = new Circuit with Adders
```

这个 circuit 变量保存的是一个拥有所有 Circuit 方法和所有 Adders 的方法的电路。注意通过这种编码模式，基于特质而非类，你搭好了提供多个组件集合的舞台。用户可以混入任何他们计划使用的组件集，就像这样：

```
val circuit =
  new Circuit
    with Adders
    with Multiplexers
    with FlipFlops
    with MultiCoreProcessors
```

将所有内容合在一起

614

以上就是整个框架了。包括一个模拟框架、一个电路模拟类和一个小型的标准加法器组件库。下面是用到它的简单的演示对象：

```
object Demo {
  def main(args: Array[String]) {
    val circuit = new Circuit with Adders
    import circuit._

    val ain = new Wire("ain", true)
    val bin = new Wire("bin", false)
    val cin = new Wire("cin", true)
    val sout = new Wire("sout")
    val cout = new Wire("cout")

    probe(ain)
```

```
    probe(bin)
    probe(cin)
    probe(sout)
    probe(cout)

    fullAdder(ain, bin, cin, sout, cout)

    circuit.start()
  }
}
```

这个示例创建一个包含 Adders 特质的电路。它立即引入了所有电路的成员，因而我们可以很容易访问诸如 probe 和 fullAdder 等方法。没有这些引入，我们就需要编写 circuit.probe(ain) 而不只是简单的 probe(ain)。

这个示例接下来创建了五条线路。其中三条用作输入（ain、bin 和 cin），两条用作输出（sout、cout）。三条输入线路被随意给定了初始值 true、false 和 true。这些输入对应的含义是给数值 0 增加 1，并且带入一个 1。

probe 方法被应用到所有五条外部可见的线路上，因此随着模拟的运行，它们的状态的任何修改都可以被观察到。最后这些线路被插到全加法器中，模拟开始。模拟的输出如下：

```
Advancing to time 1
Advancing to time 2
signal Wire(cout) changed to false
signal Wire(cin) changed to true
signal Wire(ain) changed to true
signal Wire(sout) changed to false
signal Wire(bin) changed to false
Advancing to time 3
Advancing to time 4
Advancing to time 5
Advancing to time 6
Advancing to time 7
Advancing to time 8
Advancing to time 9
Advancing to time 10
signal Wire(cout) changed to true
Advancing to time 11
Advancing to time 12
Advancing to time 13
Advancing to time 14
Advancing to time 15
Advancing to time 16
Advancing to time 17
Advancing to time 18
signal Wire(sout) changed to true
Advancing to time 19
Advancing to time 20
Advancing to time 21
signal Wire(sout) changed to false
** Agenda empty. Clock exiting at time 21.
```

正如我们预期的那样，对于1、0、1这样的输入（true、false和true），输出是一个进位1，以及为0的和（cout为true，sout为false）。

30.7 小结

并发编程带给你强大的能量。它让你能简化代码，能利用多个处理器。因此，看到大多数被广泛使用的并发基础语法结构、线程、锁和监视器等带来的是死锁和争用状况的雷区，这十分不幸。

616

Actor风格提供了一种走出雷区的途径，让你可以不面临这样大风险的死锁和争用状况的前提下编写并发程序。本章介绍了好几个用于在Scala中处理actor的基础语法结构，包括如何创建actor，如何发送和接收消息，如何用react节约线程，还有其他一些特性。最后还向你展示了如何在通用的actor风格的编程中使用这些语法结构。

617

第 31 章

连结符解析

你有时可能须要处理某种小型的、特殊用途的语言。举例来说，你可能要为软件读取配置文件，想让它们变得比 XML 更易于手工修改。或者，可能你想要在程序中支持一种输入语言，比如带有布尔型操作符的查询词句（计算机，给我找一部“有‘太空船’且没有‘爱情故事’的电影”）。不论背后的原因是什么，你将需要一个解析器。你需要一种方式来将输入语言转换成某种你的软件能够处理的数据结构。

从本质上讲，你只有少量的几个选择。一个选择是制作你自己的解析器（以及词法分析器）。如果你不是专家，这很难。如果你是专家，这仍然是很费时的一件事。

另一种选择是使用解析器生成器。有好几种这样的生成器。人们比较熟知的有生成 C 语言解析器的 Yacc 和 Bison，以及生成 Java 语言解析器的 ANTLR。你很可能还需要一个扫描器生成器，如 Lex、Flex 或 JFlex 与之配合。虽然有些不方便，这可能是最好的解决方案了。你需要学习新的工具，包括它们——有时很晦涩难懂——的错误提示。你还需要搞明白如何将这些工具的输出连接到你的程序中。这可能会限制你对编程语言的选择，并让你的工具链变得复杂起来。

本章给出的是第三种选择。与使用某个解析器生成器自成体系的领域特定语言不同，你将使用一种内部的领域特定语言，或简称为内部 DSL。内部 DSL 将由一个解析器连结符库组成——解析器连结符是用 Scala 定义的函数和操作符，它们是解析器的构建单元。这些构建单元将一一与上下文无关的语法结构对应起来，使得它们易于理解。

本章涉及到的语言特性当中，只有一个是之前没有解释过的：this 别名，这在 31.6 节中会讲到。不过，本章大量的使用到之前的章节中解释过的其他语言特性。其中起到重要作用的包括参数化类型、抽象类型、作为对象的函数、操作符重载、传名参数，以及隐式转换等。本章会介绍这些语言元素将如何被结合在一起从而设计出一个相当高级别的库。

本章介绍的概念相比之前的章节更高阶一些。如果你有良好的编译器构建基础，在阅读本章时将会受益于这部分知识，因为它帮助你更好地理解这些概念。不过，理解本章内容的唯一先决条件，是你要知道正则语法（regular grammar）和上下文无关语法（context-free grammar）。如果你对它们一无所知，那么本章的内容也可以跳过。

31.1 示例：算术表达式

我们从一个示例开始讲。比方说你想要构建一个针对由浮点数、括弧、及二元操作符+、-、*、/构成的算术表达式的解析器。首先第一步是将要解析的语言的语法写下来。以下是算术表达式的语法：

```
expr    ::= term {"+" term | "-" term}.
term    ::= factor {"*" factor | "/" factor}.
factor  ::= floatingPointNumber | "(" expr ")".
```

在这里，|表示备选产出，而{…}表示重复（0 次或更多次）。另外虽然本例中没有用到，[…]表示可选项。

这组上下文无关语法正式地定义了一个算术表达式的语言。每个表达式（由 expr 表示）都是一个词（term），后面可以跟一系列的+或-操作符及更多的词。词是一个因子（factor），后面可能跟着一组*或/操作符及更多的因子。而因子可以是数字字面量或用括弧括起来的表达式。注意这样的语法已经规定了操作符的相对优先次序。例如，*的绑定比+更紧，因为*操作得出一个词，而+操作得出一个表达式，表达式可以包含词，但词只有当 expr 被括弧括起来时才能包含它。

现在你已经定义好语法，接下来呢？如果你使用 Scala 的连结符解析器，你基本上已经做完了！你只需要做一些规律性的文本替换，并将解析器包装在一个类中，如清单 31.1 所示：

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

清单 31.1 一个算术表达式解析器

算术表达式的解析器被包含在一个继承自 `JavaTokenParsers` 特质的类中。这个特质提供了编写解析器的基本工具，并给出了一些最基础的能够识别一些词汇类别如标识符、字符串字面量和数字等的解析器。在清单 31.1 的示例中，你只需要用到基础的 `floatingPointNumber` 解析器即可，它就是从这个特质继承下来的。

`Arith` 类中的三个定义代表了算术表达式的产出。如你所见，它们与上下文无关的语法的产出十分接近。实际上，你可以从上下文无关语法自动地生成这部分内容，只需要执行一些简单的文本替换：

1. 每个产出都变成一个方法，因此你需要给它在前面加上 `def`。
2. 每个方法的结果类型都是 `Parser[Any]`，因此你需要将::=符号修改为“`: Parser[Any] =`”。在本章稍后会了解到类型 `Parser[Any]` 的含义是什么，以及如何让它变得更确切。
3. 在语法定义中，顺序的组合是隐含的，但在程序中，它由一个显式的操作符 ~ 表示。因此你需要在产出的每两个连续的符号间插入一个 ~ 。在清单 31.1 的示例中，我们决定不在 ~ 的前后加上空

格。这样，解析器代码与语法定义在视觉上就能保持很接近——它只是将空格用~字符替换掉了。

4. 重复项使用 rep(…)表示而不是{…}。同理（尽管没有在示例中显示），可选项使用 opt(…)表示而不是[…]。

5. 位于每个产出最后的句点（.）被去掉——不过如果你愿意，也可以写上一个分号（;）。

以上就是需要做的全部工作了。最终的 Arith 类定义了三个解析器：expr、term 和 factor，它们可以用来解析算术表达式和表达式的组成部分。

31.2 运行你的解析器

你可以用如下这个小程序来执行你的解析器：

```
object ParseExpr extends Arith {
  def main(args: Array[String]) {
    println("input : "+ args(0))
    println(parseAll(expr, args(0)))
  }
}
```

ParseExpr 对象定义了一个 main 方法来解析传给它的第一个命令行参数。它将原始的输入参数打印出来，然后打印它解析后的版本。解析是通过下面这个表达式来完成的：

```
parseAll(expr, input)
```

这个表达式将解析器 expr 应用到给定的 input 上。它预期所有的输入都可以匹配，也就是说，对于解析后的表达式，不应该有多余的字符跟在后面。还有一个 parse 方法，让你解析一个输入前缀，而剩下的部分先不管。你可以用如下的命令来运行这个算术解析器：

```
$ scala ParseExpr "2 * (3 + 7)"
input: 2 * (3 + 7)
[1.12] parsed: ((2~List((*~(((~((3~List())~List((+
~(7~List())))))~)))))~List())
```

这样的输出告诉你解析器成功地分析了输入字符串至位置[1.12]，意思是第一行的第十二列——换句话说，整个输入字符串——被解析。现在先不去管“parsed:”之后的结果。它的用处不大，你在后面会明白怎样能得到更具体的解析结果。

你也可以试着引入一些不合法的表达式字符串输入。例如，你可以写多出一个右括弧的表达式：

```
$ scala ParseExpr "2 * (3 + 7))"
input: 2 * (3 + 7))
[1.12] failure: `-' expected but `)' found

2 * (3 + 7))
           ^
```

在这里，expr 解析器解析了直到最后一个右括弧之前的所有内容，这个右括弧并不是（合法的）算术表达式的一部分。parseAll 方法于是发出一个错误消息，说它在有括弧的位置预期一个-操作符。

你将在本章稍后部分了解到它为什么产出这样一个特定的错误提示，以及如何改进它。

31.3 基本的正则表达式解析器

算术表达式的解析器用到了另一个名为 floatingPointNumber 的解析器。这个继承自 Arith 的超特质 JavaTokenParsers 的解析器按照 Java 的格式识别出浮点数。不过当你须要解析一个和 Java 的格式略有不同的数字时又该怎么办呢？在这种情况下，你可以使用正则表达式解析器。

这背后的想法是你可以将任何正则表达式当作解析器来使用。正则表达式可以解析所有它能匹配的字符串，而它的结果是解析后的字符串。举例来说，清单 31.2 所示的正则表达式解析器描述了 Java 的标识符应有的样子：

```
object MyParsers extends RegexParsers {
  val ident: Parser[String] = """[a-zA-Z_]\w*""".r
}
```

623

清单 31.2 针对 Java 标识符的正则表达式解析器

清单 31.2 中的 MyParsers 对象继承自 RegexParsers 特质，而 Arith 继承自 JavaTokenParsers。Scala 的连结符解析器是按照一组特质的继承关系来组织的，它们都包含在包 scala.util.parsing.combinator 中。最顶层的特质是 Parsers，它定义了一个十分通用的解析框架，应对各式各样的输入。往下一层是 RegexParsers，要求输入是一个字符序列，提供正则表达式方式的解析。再具体一点的特质是 JavaTokenParsers，实现了针对 Java 中定义的词（或语言符号）的基本类别的解析器。

31.4 另一个示例：JSON

JSON，即 JavaScript Object Notation（JavaScript 对象标记法），是一个十分流行的数据交换格式。在本节中，我们将向你展示如何为它编写解析器。以下是一组描述 JSON 句法的语法定义：

```
value   ::=  obj | arr | stringLiteral |
             floatingPointNumber |
             "null" | "true" | "false".
obj     ::=  "{" [ members ] "}".
arr     ::=  "[" [ values ] "]".
members ::=  member {"," member}.
member  ::=  stringLiteral ":" value.
values  ::=  value {"," value}.
```

JSON 值是一个对象、数组、字符串、数值或三个保留字 null、true、false 当中的一个。JSON 对象是由逗号分开、花括号括起来的成员（可能为空）的序列。每个成员都是一组字符串/值的对，其中字符串和值用冒号隔开。最后，JSON 数组是一个由逗号分开、用花括号括起来的系列值。作为示例，

清单 31.3 包含了一个格式为 JSON 对象的地址簿。

624

```
{
  "address book": {
    "name": "John Smith",
    "address" : {
      "street"   : "10 Market Street",
      "city"     : "San Francisco, CA",
      "zip"      : 94111
    },
    "phone numbers": [
      "408 338-4238",
      "408 111-6892"
      ]
  }
}
```

清单 31.3 JSON 格式的数据

使用 Scala 的解析器连结符来解析这样的数据很直截了当。完整的解析器如清单 31.4 所示。解析器遵循与算术解析器相同的结构。这一次同样也是对 JSON 语法产出的直接映射。产出使用了简化语法定义的一个快捷方式：repsep 连结符解析一个由给定分隔字符串分隔开（可能为空）的词的序列。举例来说，在清单 31.4 的示例中，repsep(member, ",") 解析的是一个逗号分割的 member 词的序列。否则就和算术表达式解析器一样，解析器的产出就会与语法定义的产出完全对应。

```
import scala.util.parsing.combinator._

class JSON extends JavaTokenParsers {

  def value : Parser[Any] = obj | arr |
                            stringLiteral |
                            floatingPointNumber |
                            "null" | "true" | "false"

  def obj    : Parser[Any] = "{"~repsep(member, ",")~"}"

  def arr    : Parser[Any] = "["~repsep(value, ",")~"]"

  def member : Parser[Any] = stringLiteral~":"~value
}
```

625

清单 31.4 一个简单的 JSON 解析器

为尝试使用 JSON 解析器，我们将对框架稍作修改，以便解析器可以操作文件而不是操作命令行：

```
import java.io.FileReader

objec tParseJSON extends JSON {
  def main(args: Array[String]) {
    val reader = new FileReader(args(0))
    println(parseAll(value, reader))
  }
}
```

该程序的 main 方法首先创建出一个 FileReader 对象。然后它将这个阅读器返回的字符根据 JSON 语法的 value 产出来进行解析。注意 parseAll 和 parse 也有重载的版本：它们都可以只接受字符序列或者同时将输入阅读器作为第二个参数。

如果你将清单 31.3 所示的“地址簿”对象保存到名为 address-book.json 的文件中，并对其运行 ParseJSON 程序，你应该会得到：

```
$ scala ParseJSON address-book.json
[13.4] parsed: (({~List((("address book"~:)~(({~List(((
"name"~:)~"John Smith"), (("address"~:)~(({~List(((
"street"~:)~"10 Market Street"), (("city"~:)~"San Francisco
,CA"), (("zip"~:)~94111)))~})), (("phone numbers"~:)~(([~
List("408 338-4238", "408 111-6892"))~])))~})))~})
```

31.5 解析器输出

ParseJSON 程序成功地解析了 JSON 地址簿。不过，解析器的输出看上去很奇怪，似乎是一个由输入的细碎片段用 List 和~组合起来的序列。这样的输出没多大意义。相对于输入，它对人类而言可读性更差，对机器而言它也过于无序，没法很容易地被计算机分析。是时候该对它做点什么了。

为搞清楚我们要做什么，你首先须要知道连结符框架中的每个解析器都返回什么样的结果（前提是它们成功地解析了输入值）。规则如下：

1. 每个写作字符串的解析器（比如：“{”或“:”或“null”）返回解析的字符串本身。
2. 正则表达式解析器，如"""[a-zA-Z_]\w*""".r 也返回解析后的字符串本身。同样的规则适用于诸如 stringLiteral 或 floatingPointNumber 等的正则表达式解析器，它们继承自 JavaTokenParsers 特质。
3. 顺序组合 P~Q 返回 P 和 Q 两个结果。这些结果通过同样写作~的样本类实例中返回。因此如果 P 返回“true”而 Q 返回“?”，那么顺序组合 P~Q 返回~("true", "?")，打印为(true~?)。
4. 备选组合 P | Q 返回 P 或者 Q 的成功结果。
5. 重复项 rep(P)或 repsep(P, separator)返回所有 P 的运行结果的列表。
6. 可选项 opt(P)返回一个 Scala 的 Option 类型的实例。如果 P 成功得到结果 R，它就返回 Some(R)，如果失败，则返回 None。

有了这些规则，你就可以推断出为什么之前的示例中解析器输出是那个样子了。不过，输出依然不那么方便。如果能把 JSON 对象映射为某种内部 Scala 表示方式来代表 JSON 值的含义就会好得多。更自然的表示方式可能是如下这个样子：

- JSON 对象由类型为 Map[String, Any]的 Scala 映射表示。每个成员都表示为映射中的键/值绑定。

- JSON 数组由类型为 List[Any] 的 Scala 列表表示。
- JSON 字符串由 Scala 的 String 表示。
- JSON 数值字面量由 Scala 的 Double 表示。
- 值 true、false 和 null 由 Scala 中同名的值表示。

627

为产出这样的表现形式，你还需要用到另一个解析器连结符形式：^^。

^^操作符对解析器的结果进行转型。使用这个操作符的表达式的格式为 P ^^ f，其中 P 是解析器而 f 是函数。P ^^ f 解析的句子和 P 没什么两样。只要 P 返回某个结果 R，P ^^ f 的结果就是 f(R)。

作为一个示例，这里有一个解析浮点数并将它转换为 Double 类型的 Scala 值的解析器：

```
floatingPointNumber ^^ (_.toDouble)
```

这里有一个解析字符串“true”并返回 Scala 的布尔值 true 的解析器：

```
"true" ^^ (x => true)
```

现在来看更高阶一些的转型。以下是 JSON 对象的新版解析器，返回一个 Scala 的 Map：

```
def obj: Parser[Map[String, Any]] = // 可以再完善
  "{"~repsep(member, ","}~"}" ^^
    { case"{"~ms~"}" =>Map() ++ ms }
```

你应该还记得~操作符产出的结果是同样名为~的样本类实例。这里是那个类的定义——它是 Parsers 特质的内部类：

```
case class ~[+A, +B](x: A, y: B) {
  override def toString = "("+ x +"~"+ y +")"
}
```

类名特意采用了与序列连结符方法~相同的名字。这样，你可以用遵循与解析器自己相同结构的模式来匹配解析结果。举例来说，模式"{"~ms~"}"匹配一个以“{”打头然后是 ms 变量最后是“}”的字符串。这样的模式严格与^^左边的解析器返回的结果相对应。在去掉了糖衣（即~先出现）的版本中，同样的模式写成这样：~(~("{", ms), "}")，不过这读起来要差得多。

628

模式"{"~ms~"}"的目的是去掉花括号，以便你可以得到从 repsep(member, ",")解析器返回的成员列表。对类似这样的情形，还有另一种方法可以避免产出不必要的、直接被模式匹配丢弃的解析器结果。这个方法使用了~>和<~解析器连结符。这两个连结符都表示和~一样的顺序组合，不过~>只保留它右操作元的结果，而<~只保留它左操作元的结果。通过这些连结符，JSON 对象解析器可以表达得更紧凑：

```
def obj: Parser[Map[String, Any]] =
  "{"~> repsep(member, ",") <~"}" ^^ (Map() ++ _)
```

清单 31.5 显示了完整的返回有意义结果的 JSON 解析器。如果你对文件 address-book.json 执

行该解析器，将会得到如下结果（添加了一些换行和缩进）：

```
$ scala JSON1Test address-book.json
[14.1] parsed: Map(
  address book -> Map(
    name -> John Smith,
    address -> Map(
      street -> 10 Market Street,
      city -> San Francisco, CA,
      zip -> 94111),
    phone numbers -> List(408 338-4238, 408 111-6892)
  )
)
```

以上就是你开始编写自己的解析器之前所需要知道的全部内容。为帮助记忆，表31.1列出了到目前为止讨论到的解析器连结符。

符号名称 vs. 由字母数字组成名称

表31.1（见下页）中的许多解析器连结符使用的是符号名称。这既有好处也有坏处。坏处是符号名称需要时间去学习了解。不熟悉Scala连结符解析库的用户可能会搞不清楚~、~>或^^的含义是什么。但这样做好处是符号名称较短，可以让它们有“正确”的优先级和相关性[1]。举例来说，连结符~、^^和|的优先级是从左到右递减的。典型的语法如清单31.5

629

```
import scala.util.parsing.combinator._

class JSON1 extends JavaTokenParsers {

  def obj: Parser[Map[String, Any]] =
    "{"~>repsep(member, ",") <~"}" ^^ (Map() ++ _)
  def arr: Parser[List[Any]] =
    "["~>repsep(value, ",") <~"]"
  def member: Parser[(String, Any)] =
    stringLiteral~":"~value ^^
      { case name~":"~value => (name, value) }
  def value: Parser[Any] = (
    obj
  | arr
  | stringLiteral
  | floatingPointNumber ^^ (_.toDouble)
  | "null"  ^^ (x => null)
  | "true"  ^^ (x => true)
  | "false" ^^ (x => false)
  )
}
```

清单31.5 返回有意义的结果的完整JSON解析器

[1] 译注：前关联或后关联。

表 31.1 解析器连结符汇总

"..."	字面量
"...".r	正则表达式
P~Q	顺序组合
P<~Q, P~>Q	顺序组合；只保留左或右
P|Q	备选项
opt(P)	可选项
rep(P)	重复项
repsep(P, Q)	交错在一起的重复项
P^^f	结果转换

关闭分号推导

注意清单 31.5 中的 value 解析器主体是包含在括弧中的。这是在解析器表达式中关闭分号推导的小技巧。你在 4.2 节曾经看到 Scala 假定任何从句法上是独立语句的两行之间都有一个分号，除非第一行以中缀结尾，或者两行被括在括弧或方括号中。因此，你本可以在每个备选项末尾写上|操作符而不是在下一个备选项的开始，就像这样：

```
def value: Parser[Any] =
  obj |
  arr |
  stringLiteral |
  ...
```

这种情况下，就不须要在 value 解析器主体之外加上括弧了。不过，有人更倾向于在第二个备选项的开始看到|操作符，而不是在第一个备选项的末尾。通常，这会造成两行间不需要的分号，像这样：

```
  obj ; // 分号被隐含着插入了
| arr
```

分号改变了代码的结构，造成它编译失败。将整个表达式放在括弧中就避免了分号的出现，从而让代码正确的编译。

产出由不同的解析部分和转型部分构成。解析部分通常包含多个由~操作符分隔开的顺序条目。根据给定的~、^^和|的优先级，你可以不需要任何括弧就能写出这样的语法产出。

除此之外，符号操作符看起来也比有字母和数字组成的操作符更轻量。对解析器而言这很重要，因为它让你关注于当前处理的语法，而不是连结符本身。为看清楚区别，想象一下如果顺序组合（~）叫做 andThen，而备选项（|）叫做 orElse，清单 31.1 中的算术表达式解析器就会是如下这个样子：

```
class ArithHypothetical extends JavaTokenParsers {
  def expr: Parser[Any] =
    term andThen rep(("+" andThen term) orElse
```

```
                    ("-" andThen term))
  def term: Parser[Any] =
    factor andThen rep(("*" andThen factor) orElse
                       ("/" andthen factor))
  def factor: Parser[Any]    =
    floatingPointNumber orElse
    ("(" andThen expr andThen ")")
}
```

你会注意到代码变得比以前长得多，并且很难在所有这些操作符和括弧当中“看”到语法定义。不过话说回来，初次接触连结符的人看到这样的代码可能会更容易搞清楚代码要做的是什么。

31.6 实现连结符解析器

从之前的各节我们可以看到，Scala 的连结符解析器提供了构建你自己的解析器的便利手段。由于它们和 Scala 库没什么两样，它们能够被无缝地整合到 Scala 程序中。因此将解析器和一些处理它产出的结果的代码结合在一起，或者临时做一个可以从某种特定的源（比如说一个文件、字符串或字符数组）接收其输入的解析器，都相当容易。

这是如何做到的呢？在本章剩余的部分你将会看到连结符解析器库“背后的”细节。你将会看到什么是解析器，以及在之前的各节中遇到的基础解析器和解析器连结符是如何实现的。如果你想要做的只是编写一些简单的连结符解析器，你可以放心地跳过。话虽如此，阅读本章剩余的部分将让你对连结符解析器，以及对更通用的连结符领域特定语言的设计原理有更深入的理解。

632

在符号名称与由字母和数字组成的名称之间进行选择

作为在符号名称与由字母和数字组成的名称之间进行选择的指导原则，我们建议如下：

- 在符号名称已经具备普遍含义的情况下使用符号名称。举例来说，不会有人建议用 add 而不是 +来表示数值的加法。
- 否则，如果你想要让代码对普通读者而言更易于理解，则优选由字母和数字组成的名称。
- 你仍然可以为特定领域的库选用符号名称，如果这样明显更可读，且你并不指望那些不具备该领域扎实基础的普通读者能够立即理解这些代码的话。

具体到解析器连结符，摆在我们面前的是一个高度领域特定的语言，哪怕我们使用由字母和数字组成的名称，普通读者理解起来还是会有困难。不仅如此，符号名称对专家而言很明显要更可读。因此我们认为对这个应用场景而言，使用符号名称是明智的。

Scala 的连结符解析框架的核心位于 scala.util.parsing.combinator.Parsers 特质中。该特质定义了 Parser 类型及所有基础的连结符。除非特别指出，接下来的两个小节中介绍的定义都位于该特质中。也就是说，它们都被包含在以如下内容开始的特质定义里：

```
package scala.util.parsing.combinator
trait Parsers {
  ... // 除非特别说明代码均出现在这里
}
```

本质上讲 Parser 只是一个从某种输入类型解析出结果的函数。作为第一次尝试，Parser 的类型定义可以被写成如下的样子：

```
type Parser[T] = Input =>ParseResult[T]
```

633

解析器输入

有时候，解析器读取的是语言符号的流而不是原始的字符序列。因此会用到单独的词法分析器来将原始的字符流转换成语言符号流。解析器输入的类型定义如下：

```
type Input = Reader[Elem]
```

Reader 类来自 scala.util.parsing.input 包。它和 Stream 很像，不过它同时还跟踪记录了所有它读取的元素的位置。Elem 类型代表了每个输入的元素。它是 Parsers 特质的一个抽象类型成员：

```
type Elem
```

这意味着 Parsers 的子类和子特质需要将 Elem 类具化为要解析的输入元素的类型。举例来说，RegexParsers 和 JavaTokenParsers 将 Elem 具化为 Char。不过我们也可以将 Elem 设置为某种别的类型，比如从单独的词法分析器（lexer）返回的语言符号的类型。

解析器结果

解析器对某个给定的输入可能成功，也可能失败。因此 ParseResult 类有两个子类分别表示成功和失败：

```
sealed abstract class ParseResult[+T]
case class Success[T](result: T, in: Input)
  extends ParseResult[T]
case class Failure(msg: String, in: Input)
  extends ParseResult[Nothing]
```

Success 用 result 参数保存解析器返回的结果。解析器返回的结果可以是任何类型的，这就是为什么 ParseResult、Success 和 Parser 都以一个类型参数 T 参数化。类型参数表示的是从给定解析器返回的结果的种类。Success 同时还接受第二个参数 in，它指的是紧跟在解析器已处理掉的部分后面的输入。这个字段用于串联解析器，以便解析器可以一个接一个的进行操作。注意这是进行解析的一种纯函数式的方式。输入并非作为副产物读取进来，而是保持在流中。解析器分析输入流的某个部分，然后在结果中返回剩余的部分。

634

ParseResult 的另一个子类是 Failure。这个类接受一个描述解析器为何失败的消息作为参数。和 Success 一样，Failure 也接受剩余的输入流作为第二个参数。这并非出于串联的需要（解析器在失败后不会继续），而是为了将错误提示放在输入流中正确的位置。

注意解析结果定义以便与类型参数 T 协变。也就是说，假如返回 String 作为结果的解析器，和返回 AnyRef 的解析器是兼容的。

Parser 类

之前将解析器描述为从输入解析出结果有些过于简单了。之前的示例显示解析器还实现了诸如顺序组合两个解析器的~方法和二选一的 | 方法。因此 Parser 实际上是一个继承自函数类型 Input =>

ParseResult[T]并额外定义了如下这些方法的类：

```
abstract class Parser[+T] extends (Input =>ParseResult[T])
{ p =>
  // 未给出的定义解析器行为的方法。
  def apply(in: Input): ParseResult[T]

  def ~ ...
  def | ...
  ...
}
```

由于解析器是(即继承自)函数,因此需要定义 apply 方法。你在 Parser 类中看到了抽象的 apply 方法，不过这只是用于文档目的，因为任何继承自父类型 Input => ParseResult[T]的 case 类都有同样的方法（你应该还记得这个类型是 scala.Function1[Input, ParseResult[T]]的简写）。apply 方法仍然需要在每个继承自抽象类 Parser 的解析器中实现。我们将在接下来的 this 别名这一节之后讨论这些解析器。

635

给 this 起别名

Parser 类的主体由一个有意思的表达式开始：

```
abstract class Parser[+T] extends ... { p =>
```

诸如"id =>"这样紧跟在类模板起始花括号之后的语句将标识符 id 在类中定义为 this 的别名。就好像你将：

```
val id = this
```

写在类定义中，只不过 Scala 编译器知道 id 是 this 的一个别名。举例来说，你可以使用 id.m 或 this.m 来访问类的对象级私有成员 m，这两种写法完全等同。如果 id 仅仅是把 this 定义为等式右边的 val，则第一个表达式将无法编译，因为在这种情况下 Scala 的编译器会把 id 当作普通的标识符来看待。

你在 27.4 节中看到过类似的语法，它被用来将自我类型提供给特质。当你须要访问外部类的 this 时，别名也可以拿来作为缩写。这里有个例子：

```
class Outer { outer =>
  class Inner {
    println(Outer.this eq outer) // 打印: true
  }
}
```

这个示例定义了两个嵌套类，Outer 和 Inner，在 Inner 中 Outer 类的 this 被引用两次，用的是不同的表达式。第一个表达式显示了 Java 的做事方式：你可以给保留字 this 前面加上外部类的名称和句点；这样的表达式就指向外面那个类的 this。第二个表达式显示了 Scala 提供给你的另一种选择。通过引入名为 outer 的 Outer 类的 this 的别名，你在内部类中也可以直接引用这个别名。如果你为别名选择的名称足够好，Scala 提供的这种方式更精简，也让代码更清晰。你将在第 413 页和第 414 页看到相关的示例。

636

单语言符号解析器

Parsers 类定义的通用解析器 elem 可以被用来解析任何单个语言符号：

```
def elem(kind: String, p: Elem =>Boolean) =
  new Parser[Elem] {
    def apply(in: Input) =
      if (p(in.first)) Success(in.first, in.rest)
      else Failure(kind +" expected", in)
  }
```

这个解析器接受两个参数：kind 字符串描述要解析的是哪一种语言符号，以及对 Elem 的一个论断 p，标明元素是否符合要解析的语言符号类别的要求。

给某个输入 in 应用解析器 elem(kind, p)时，输入流中的第一个元素被论断 p 检测。如果 p 返回 true，解析器成功返回元素本身作为结果，剩余的输入是紧接着已被解析的元素后开始的输入流。另一方面，如果 p 返回 false，则解析器失败，且会有错误提示表明预期的语言符号种类是什么。

顺序组合

elem 解析器只处理单个元素。为解析更多有意思的词句，你可以用顺序组合操作符~将解析器串在一起。正如你之前曾经看到过的，P~Q 是这样一个解析器：它首先将 P 解析器应用到给定的输入字符串，然后如果 P 成功了，则 Q 解析器被应用到 P 完成其工作后剩下的输入上。

~连结符是作为 Parser 类的方法实现的，其定义如清单 31.6 所示。这个方法是 Parser 类的成员。在这个类中，p 被“p =>”部分指定为 this 的别名，因此 p 表示~的左操作元（或：接收者）。它的右操作元由参数 q 表示。现在，如果 p~q 被应用到某个输入 in 时，首先会在 in 上运行 p，通过模式匹配分析出结果。如果 p 成功了，剩余的输入 in1 上会再运行 q。如果 q 也成功了，整个解析器就成功了。其结果是同时包含了 p 的结果（即 x）及 q 的结果（即 y）的~对象。与此相反，如果 p 或者 q 失败了，p~q 的结果则是由 p 或 q 返回的 Failure 对象。

```
abstract class Parser[+T] ... { p =>
...
  def ~ [U](q: =>Parser[U]) = new Parser[T~U] {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) =>
        q(in1) match {
          case Success(y, in2) =>Success(new ~(x, y), in2)
          case failure => failure
        }
      case failure => failure
    }
  }
```

清单 31.6 ~ 连结符方法

~的返回类型是一个解析器，该解析器返回的是元素类型为 T 和 U 的样本类~的实例。类型表达式 T~U 只不过是一个更可读的参数化类型~[T, U]的简写。总体而言，Scala 总是将二元的类型操作如 A op

B解释为参数化的类型op[A, B]。这与模式的情形是类似的，二元模式P op Q同样是被解释为一个函数应用，即op(P, Q)。

另外两个顺序组合操作符<~和~>也可以像~那样来定义，只是在结果如何计算的问题上需要做些小的调整。不过更优雅的方法是像如下这样直接用~来定义：

```
def <~ [U](q: =>Parser[U]): Parser[T] =
  (p~q) ^^ { case x~y => x }
def ~> [U](q: =>Parser[U]): Parser[U] =
  (p~q) ^^ { case x~y => y }
```

备选组合

备选组合P | Q对给定的输出应用P或Q。它首先尝试P，如果P成功，则整个解析器成功并返回P的结果，否则如果P失败，则在与P相同的输入上尝试Q，Q的结果就是整个解析器的结果。

这里有一个作为Parser类的方法|的定义：

```
def | (q: =>Parser[T]) = new Parser[T] {
  def apply(in: Input) = p(in) match {
    case s1 @ Success(_, _) => s1
    case failure => q(in)
  }
}
```

注意如果P和Q都失败了，则失败消息由Q决定。这个看似微妙的选择在稍后31.9节会讨论到。

处理递归

注意方法~和|的参数q是传名的——其类型声明之前标有=>。这意味着只有当q需要时实际的解析器参数才会被求值，只有当p运行之后才会需要q的值。这使得我们可以编写类似如下这样的递归解析器，它可以解析括在任意多的括弧内的数字：

```
def parens = floatingPointNumber | "("~parens~")"
```

如果|和~接受的是传值参数，则这个定义非但读不到任何信息，还会立即造成栈溢出，因为parens的值出现在等式右边的中间。

结果转换

Parser类的最后一个方法是对解析器的结果进行转换。当P成功时解析器P ^^ f就是成功的，它将会返回使用函数f转换后的P的结果。以下是这个方法的实现：

```
def ^^ [U](f: T => U): Parser[U] = new Parser[U] {
  def apply(in: Input) = p(in) match {
    case Success(x, in1) =>Success(f(x), in1)
    case failure => failure
  }
}
```

```
} // Parser 类到此结束
```

639

不读取任何输入的解析器

还有两个不处理任何输入的解析器：success 和 failure。success(result)解析器总是成功并返回给定的 result。failure(msg)解析器则总是失败并给出 msg 作为错误提示。它们都是在 Parsers 特质中作为方法实现的，Parsers 这个外部特质同样也包含了 Parser 类。

```
def success[T](v: T) = new Parser[T] {
  def apply(in: Input) = Success(v, in)
}
def failure(msg: String) = new Parser[Nothing] {
  def apply(in: Input) = Failure(msg, in)
}
```

可选项和重复项

同样定义在 Parsers 特质中的是可选项和重复项连结符 opt、rep 和 repsep。它们都是利用顺序组合、备选项及结果转换来实现的：

```
def opt[T](p: =>Parser[T]): Parser[Option[T]] = (
  p ^^ Some(_)
| success(None)
)

def rep[T](p: Parser[T]): Parser[List[T]] = (
  p~rep(p) ^^ { case x~xs => x :: xs }
| success(List())
)

def repsep[T, U](p: Parser[T],
      q: Parser[U]): Parser[List[T]] = (
    p~rep(q~> p) ^^ { case r~rs => r :: rs }
  | success(List())
  )

} // Parsers 特质到此结束
```

640

31.7 字符串字面量和正则表达式

到目前为止看到的解析器用字符串字面量和正则表达式来解析单个单词，是受到来自于 RegexParsers 的支持，它是 Parsers 的子特质：

```
trait RegexParsers extends Parsers {
```

这个特质比 Parsers 特质更具体，只处理字符序列的输入：

```
type Elem = Char
```

它定义了两个方法，literal 和 regex，它们的签名如下：

```
implicit def literal(s: String): Parser[String] = ...
implicit def regex(r: Regex): Parser[String] = ...
```

注意这两个方法都有一个 implicit 修饰符，因此它们可以在给定 String 或 Regex 但预期 Parser 时自动被应用。这就是为什么你可以直接在语法定义中编写字符串字面量和正则表达式，而不须要将它们用上述的两个方法包装起来。举例来说，解析器"("~expr~")"会自动展开为 literal("(")~expr~literal(")")。

RegexParsers 特质同时还帮我们处理掉符号间的空白。为此，它在执行 literal 或 regex 解析器之前会调用一个名为 handleWhiteSpace 的方法。handleWhiteSpace 方法会跳过符合 whiteSpace 正则表达式的最长输入序列[2]，默认定义如下：

```
  protected val whiteSpace = """\s+""".r
} // RegexParsers 至此结束
```

如果你倾向于使用对空白的不同处理方式，可以重写这个名为 whiteSpace 的 val。举例来说，如果你完全不想跳过空白，可以将 whiteSpace 重写为空白的正则表达式：

```
object MyParsers extends RegexParsers {
  override val whiteSpace = "".r
  ...
}
```

641

31.8 词法分析和解析

语法分析通常分为两个阶段。词法分析器（lexer）阶段识别出输入中的每个单词并将它们归类为不同的语言符号类别。这个阶段也被称作词法分析。在这之后是句法分析，分析语言符号的序列。句法分析有时也被称为解析，尽管这并不十分准确，因为词法分析也可以被看作是一个解析问题。

前一节提到的 Parsers 特质可以被用于上述的任何一个阶段，因为它的输入元素的类型是抽象类型 Elem。对词法分析而言，Elem 可以被具化为 Char，意思是说解析的是组成单词的各个字符。而句法分析器可以将 Elem 具化为词法分析器返回的语言符号的类型。

Scala 的解析连结符提供了用于词法和句法分析的多个工具类。它们包含在两个子包中，分别对应这两类分析：

```
scala.util.parsing.combinator.lexical
scala.util.parsing.combinator.syntactical
```

如果你想要将解析器拆分为单独的词法分析和句法分析器，你应该查阅一下这两个包的 Scaladoc 文档。不过对于简单的解析器而言，本章在前面介绍的基于正则表达式的方式通常就够用了。

[2]译注：尽可能多地匹配掉空白。

31.9 错误报告

还有最后一个话题没有讲到：解析器如何发出错误消息呢？解析器的错误报告从某种意义上讲是一个黑魔法。问题之一是当解析器拒绝某个输入时，它通常遇到了多个不同的错误。每个可选择的不同解析路径都失败了，对递归而言则是每个选择点都失败了。那么通常数量众多的错误中，哪些应该作为错误消息提示给用户呢？

Scala 的解析库实现了一个简单的启发式算法：在所有失败当中，选择出现在输入最后未知的那一个。换句话说，解析器选取仍然合法的最长的前缀，然后发出一个错误消息，描述为什么对这个前缀的解析不能再走得更远了。如果在最后这个位置存在多个失败点，则选择最后被访问到的那一个。

举例来说，考虑运行 JSON 解析器来解析一个有错的地址簿，从如下的行开始：

```
{ "name": John,
```

这个语句中最长的合法前缀为“{"name":”。因此 JSON 解析器将会把单词 John 标记为错误。JSON 在这个点上预期一个值，而 John 是一个标识符，不算值（假定该文档的作者忘记将名字用引号括起来了）。解析器对该文档发出的错误提示为：

```
[1.13] failure: "false" expected but identifier John found
  { "name": John,
              ^
```

预期“false”这个部分源自这样一个事实：“false”是 JSON 语法定义中 value 的最后一个可选择的产出，因此这是在这个点上最后一个失败。知道 JSON 语法定义的细节用户可以将错误提示重建出来，不过对于非专家而言，这个错误提示可能就很让人意外并且也十分容易被误导。

更好的错误提示可以通过添加一个“捕获所有”的失败点作为 value 产出的最后一个选项来实现：

```
def value: Parser[Any] =
  obj | arr | stringLit | floatingPointNumber | "null" |
  "true" | "false" | failure("illegal start of value")
```

加上这一段代码并不会改变作为合法文档被接受的输入集。它所做的是改进错误提示，因为现在这个特意加上的 failure 才是最后一个选项，因而被报告出来：

```
[1.13] failure: illegal start of value
  { "name": John,
              ^
```

这个错误报告的“最后一个可能”这样的设计实现用到 Parsers 特质中一个名为 lastFailure 的字段，用来标记在输入的最后位置发生的最后一个失败：


```
var lastFailure: Option[Failure] = None
```

这个字段被初始化为 None。它在 Failure 类的构造方法中被更新：

```
case class Failure(msg: String, in: Input)
    extends ParseResult[Nothing] {
  if(lastFailure.isDefined &&
```

```
        lastFailure.get.in.pos<= in.pos)
      lastFailure = Some(this)
  }
```

这个字段被 phrase 方法读取，如果解析器失败的话，这个方法将发出最终的错误提示。以下是 Parsers 特质中的 phrase 方法实现：

```
def phrase[T](p: Parser[T]) = new Parser[T] {
  lastFailure = None
  def apply(in: Input) = p(in) match {
    case s @ Success(out, in1) =>
      if (in1.atEnd) s
      else Failure("end of input expected", in1)
    case f : Failure =>
      lastFailure
  }
}
```

phrase 方法运行作为参数传递给它的解析器 p，如果 p 成功地完整处理掉输入，则返回 p 的成功结果。如果 p 成功但输入并没有被完整读取，则返回消息提示为“end of input expected（预期输入结尾）”的失败。如果 p 失败了，则返回 lastFailure 中保存的失败或错误。注意对 lastFailure 的处理并不是函数式的；它作为 Failure 的构造器和 phrase 方法自身的副产物被更新。同样设计的函数式版本并非不可能，但这将会要求把 lastFailure 值贯穿到每个解析器结果中，不论这个结果是 Success 还是 Failure。

31.10 回溯 vs. LL(1)

644

解析器连结符在选择不同的解析器时采用回溯的方式。在表达式 P | Q 中，如果 P 失败了，则 Q 会运行与 P 相同的输入。哪怕 P 在失败之前已经解析出了某些语言符号也是如此。这样同样的语言符号会再次被 Q 解析。

回溯只对如何公式化语法定义增加了少量限制，以便它能够被解析。本质上讲，你只需要避免左递归的产出。类似如下的产出：

expr ::= expr "+" term | term.

总是会失败，因为 expr 立即调用了自身因此不会继续往前[3]。另一方面，回溯也潜在会带来巨大的消耗，因为同样的输入可能被解析多次。考虑如下这样的产出：

expr ::= term "+" expr | term.

如果 expr 解析器应用到诸如(1+2)*3 这样的合法输入词句上会发生什么呢？首先会尝试第一个选择，当匹配+符号时会失败。然后会对同样的词句尝试第二个选择，结果成功了。到最后这个词句被解析了两次。

[3]译注：哪怕存在左递归，我们也有办法避免栈溢出，不过这需要更精细的解析连结符框架，这样的框架至今还未被实现。

通常我们可以改变语法定义来避免回溯。举例来说，对算术表达式而言，以下任何一个产出都能工作：

```
expr :: = term [ "+" expr ].
expr :: = term {"+" term}.
```

许多语言都认可所谓的“LL(1)”语法定义[4]。如果连结符解析器从这样一个语法构造出来，则它永远都不会回溯，也就是说，输入位置永远不会被重置到更早的值。举例来说，本章之前介绍的算术表达式和 JSON 词句均为 LL(1)的，因此来自这些语言的输入从不会动用到解析器连结符框架的回溯能力。

连结符解析框架让你可以用一个新的操作符~! 显式地指定某个语法定义是 LL(1)的。这个操作符就像顺序组合~不过永远不会回溯到“未读”但已经被解析过的输入元素上。使用这个操作符，算术表达式的解析器也可以写成下面这个样子：

```
def expr : Parser[Any] =
  term~! rep("+" ~! term | "-" ~! term)
def term : Parser[Any] =
  factor~! rep("*" ~! factor | "/" ~! factor)
def factor: Parser[Any] =
  "(" ~! expr~! ")" | floatingPointNumber
```

LL(1)解析器的一个优势是它可以使用更为简单的输入技巧。输入可以被顺序地读取出来，而输入元素可以随着它们被读取即被丢弃。这是 LL(1)解析器通常比回溯解析器更有效率的另一个原因。

31.11 小结

你现在已经看到了 Scala 的连结符解析框架的所有重要元素。像这样一个真正有用的东西，代码量却出人意料的少。利用这个框架你可以构建出大量上下文无关的语法定义。这个框架让你很快就能上手，不过对于新的语法种类和输入方法同样也可以进行定制。由于是 Scala 类库，它与语言的其他部分可以无缝集成。因此在较大的 Scala 程序中集成一个连结符解析器十分容易。

连结符解析器的一个弱点是它们并不十分高效，至少与那些具有特定用途的工具如 Yacc 或 Bison 生成的解析器相比是这样。这有两个原因：首先，连结符解析自身所用的回溯方法效率并不高。根据语法定义和解析输入的不同，它可能会由于重复的回溯带来指数级的变慢。这可以通过制作 LL(1)语法定义和使用专用的顺序组合操作符~!来解决。

影响连结符解析器性能的第二个问题是它们将解析器构建和输入分析混在同一组操作中。这样带来的效果是，对每个要解析的输入，解析器都是重新生成的。

这个问题可以被克服，不过需要不同的解析器连结符框架实现。在优化的框架中，解析器不再表示为从输入解析结果的函数，而是表示为一棵树，每个构建步骤都表示为样本类。举例来说，顺序组合可以用样本类 `Seq` 表示，备选项可以用 `Alt` 表示，等等。“最外面”的解析器方法 `phrase` 则可以将解析器的这个符号表示用标准的解析器生成算法，进而转换成高效的解析表。

[4]译注：Aho 等，“Compilers: Principles,Techniques, and Tools”。[Aho86]

这样做的好处是，从用户的角度看，与普通的连结符解析器相比没有任何变化。用户仍然可以用ident、floatingPointNumber、~、|等等来编写解析器。他们不须要知道这些方法生成的是解析器的符号化表示而不是解析器函数。由于 phrase 连结符将这些符号化表示转换成真正的解析器，所有的功能都和原先一样好用。

这样的设计对于性能而言优势是双重的。首先，你现在可以将解析器构建和输入分析分开。如果你本来要写：

```
val jsonParser = phrase(value)
```

然后再将 jsonParser 应用到不同输入，jsonParser 可以只创建一次，而不是在每次读取输入时都创建。

其次，解析器生成可以使用高效的解析算法如 LALR(1)[5]。这些算法通常带来比使用回溯快得多的解析器。

目前，Scala 中这样一个优化的解析器生成器还没有被写出来。不过完全可以做到这样。如果有人愿意贡献这样一个生成器，将它集成到标准 Scala 库中是很容易的。不过，尽管考虑到未来某个时候这样一个生成器可能会存在，我们也有理由保留现有的解析器连结符框架。它比解析器生成器更易于理解和适应，而实际工作中速度的差异通常并没有那么重要，除非你想要解析的输入十分庞大。

[5]注：Aho 等，“Compilers: Principles, Techniques, and Tools”。[Aho86]

第 32 章

GUI 编程

在本章中你将会了解到如何用 Scala 开发使用图形用户界面（GUI）的应用程序。我们将要开发的应用程序基于一个 Scala 库，该库提供了对 Java 的 Swing 框架的 GUI 类的访问。从概念上讲，这个 Scala 库和它下层的 Swing 类很像，不过隐藏了大部分的复杂度。你会发现使用这个框架来开发 GUI 应用程序实际上非常容易。

尽管 Scala 做了简化，像 Swing 这样的框架功能十分丰富，有许多不同的类，每个类中又有许多方法。为了在这样一个功能丰富的库中找到方向，使用类似于 Scala 的 Eclipse 插件这样的 IDE 会很有帮助，好处是 IDE 可以交互式地通过它的命令补全功能向你展示一个包中都有哪些类，以及你引用的对象上有哪些可用的方法。当你初次探索未知的库空间时，这能够大大加速你的学习进程。

32.1 第一个 Swing 应用

作为第一个 Swing 应用程序，我们将从包含单个按钮的窗体开始。为了用 Swing 进行编程，你需要从 Scala 的 Swing API 包引入各式各样的类：

```
import scala.swing._
```

清单 32.1 给出了第一个用 Scala 编写的 Swing 应用程序。如果编译并运行该文件，应该会看到如图 32.1 左侧所示的窗体。窗体还可以被拉大，如图 32.1 右侧所示。

```
import scala.swing._

object FirstSwingApp extends SimpleGUIApplication {
  def top = new MainFrame {
    title = "First Swing App"
    contents = new Button {
      text = "Click me"
    }
  }
}
```

清单 32.1　一个用 Scala 编写的简单 Swing 应用程序

如果你逐行分析清单 32.1 中的代码，会注意到如下元素：

```
object FirstSwingApp extends SimpleGUIApplication {
```

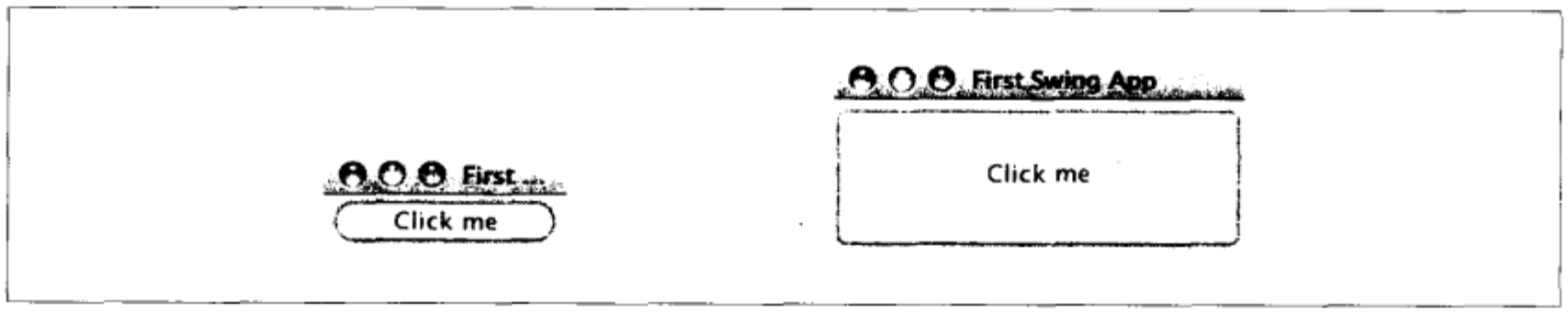


图32.1 一个简单的Swing应用程序：初始状态（左）和拉大之后的状态（右）

在引入声明后的第一行，是FirstSwingApp对象，它继承自scala.swing.SimpleGUIApplication。这与传统的命令行应用程序不同，那些应用程序可能继承自scala.Application。SimpleGUIApplication类已经定义了包含一些设置Java Swing框架代码的main方法。main方法随后继续调用top方法，而这个方法是由你来提供的：

```
def top = new MainFrame {
```

接下来的一行实现top方法。这个方法包含定义顶级GUI组件的代码。这通常是某种Frame——即可以包含任意数据的窗体。在清单32.1中，我们选定MainFrame作为顶级组件。MainFrame就像是个普通的Swing的Frame，只不过关闭它的同时也会关闭整个GUI应用程序。

```
title = "First Swing App"
```

框架(Frame)有一些属性，其中最重要的两个是：title，这将被写到标题栏；以及它的contents，这些内容将被显示在窗体当中。在Scala的Swing API中，这些属性被建模成属性值。你应该还记得在18.2节我们讲到，Scala中的属性值被编码成getter和setter方法对。举例来说，Frame对象的title属性被实现为一个getter方法：

```
def title: String
```

和一个setter方法：

```
def title_=(s: String)
```

上述对title的赋值调用的正是这个setter方法。赋值的效果是选定的标题被显示在窗体的顶部。如果不做这个赋值，窗体的标题将会是空白的。

```
contents = new Button {
```

top框架是这个Swing应用的顶级组件。它是一个Container，这意味着在它当中还可以定义更多的组件。每个Swing容器都有一个contents属性，让你获取和设置它包含的组件。这个属性的getter方法contents类型为Seq[Component]，说明组件一般而言可以拥有多个对象作为其内容。不过框架总是只有一个组件作为contents。这个组件用setter方法contents_=来设置或修改。举例来说，在清单32.1中，单个Button构成了top框架的contents。

```
text = "Click me"
```

按钮同样有标题，在本例中为“Click me”。

32.2 面板和布局

接下来，我们将添加一些文本，作为应用程序的 top 框架的另一个内容元素。图 32.2 的左侧显示了应用程序应有的样子。在上一节你看到了，框架只有一个子元素。因此，为制作出既有按钮又有标签的框架，你须要创建一个不同的容器来容纳它们。面板就是用来做这个的。Panel 是根据某种固定的布局规则显示所有它包含的组件的容器。Panel 类的各种子类实现了许多不同的、可用的布局，从简单的到很复杂的都有。实际上，复杂的 GUI 应用程序最难实现的部分之一可能就是把布局弄对——做出能在各种设备、各种窗体尺寸都显示得很好的布局并不容易。

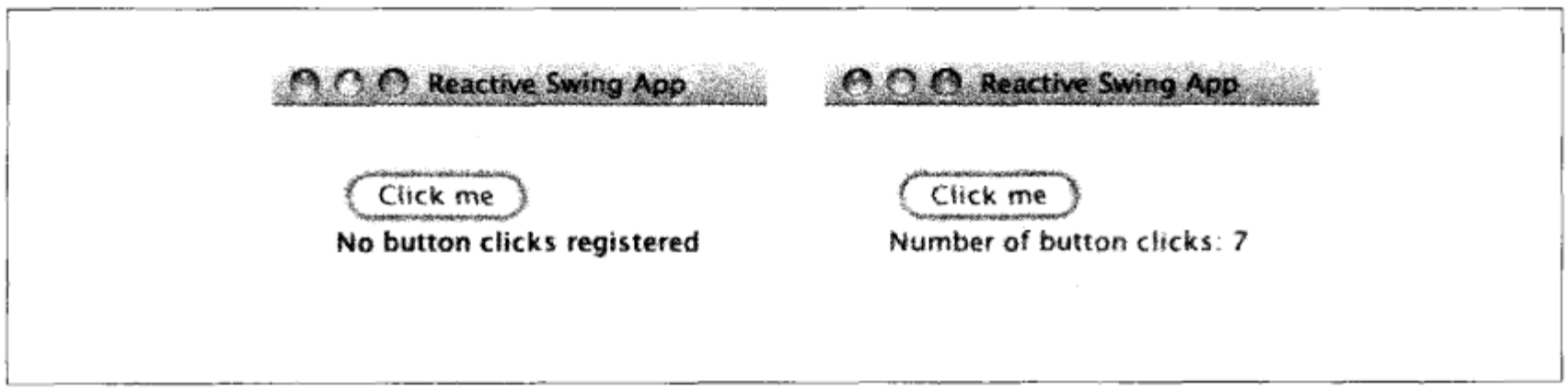


图 32.2 一个有反应的 Swing 应用程序：初始状态（左）及点击后的状态（右）

清单 32.2 给出了完整的实现。在这个类中，top 框架的两个子元素分别命名为 button 和 label。button 的定义和之前一样。label 是一个不可编辑的、只用来显示的文本字段：

```
val label = new Label {
  text = "No button clicks registered"
}
```

```
import scala.swing._
object SecondSwingApp extends SimpleGUIApplication {
  def top = new MainFrame {
    title = "Second Swing App"
    val button = new Button {
      text = "Click me"
    }
    val label = new Label {
      text = "No button clicks registered"
    }
    contents = new BoxPanel(Orientation.Vertical) {
      contents += button
      contents += label
      border = Swing.EmptyBorder(30, 30, 10, 30)
    }
  }
}
```

清单 32.2 在面板上组装组件

清单 32.2 中的代码选用了简单的垂直布局，组件在 BoxPanel 中被一个一个垒在一起。

```
contents = new BoxPanel(Orientation.Vertical) {
```

BoxPanel的contents属性是一个（初始为空的）缓冲器，button和label这些元素用+=操作符添加进去：

```
contents += button
contents += label
```

我们还通过将border属性赋值给面板的方式在这两个对象周围添加一个边界：

```
border = Swing.EmptyBorder(30, 30, 10, 30)
```

和其他GUI组件一样，边界也是由对象表示的。EmptyBorder是Swing对象的工厂方法，接受四个参数分别表示要绘制的对象边界在上、右、下和左边的宽度。

尽管简单，这个示例已经显示了结构化GUI应用程序的基本方式。它由组件构成，这些组件是scala.swing类如Frame、Panel、Label或Button的实例。组件有属性，可以被应用程序定制。Panel组件可以在contents属性中包含多个其他组件，因而最终GUI应用程序是由一棵组件树构成的。

653

32.3 处理事件

另一方面，我们的应用程序还缺少一个重要的特性。如果你运行清单32.2中的代码并点击显示出来的按钮，什么都不会发生。实际上，应用是完全静态的；除了top框架的关闭按钮会终止应用程序的运行以外，它不会以任何方式响应用户事件。因此接下来我们将进一步完善这个应用程序以使它在按钮旁边显示一个标签来表示按钮被点击的频次。图32.2的右侧显示了应用程序在几次按钮点击后应有的样子。

为此，你须要为用户输入事件（按钮被点击）关联一个动作。Java和Scala基本上用相同的“发布/订阅”方式来处理事件：组件可以是发布者，也可以是订阅者。发布者发布事件，订阅者向发布者订阅从而在任何发布的事件发生时得到通知。发布者又称作“事件源”，订阅者又称作“事件监听器”。举例来说，Button是一个事件源，发布了一个事件ButtonClicked，表示该按钮被点击。

在Scala中，订阅一个事件源source的方法是调用listenTo(source)。还有从一个事件源取消订阅的方法deafTo(source)。在当前这个示例应用程序中，首先要做的是让top框架监听它的按钮，以便它在按钮发出任何事件时得到通知。为此你需要在top框架的主体中增加如下调用：

```
listenTo(button)
```

收到事件通知只是整个故事的一半，另一半是处理它们。这里是Scala的Swing框架与Java的Swing API最不一样的地方（要简单很多）。在Java中，发出事件信号意味着调用一个必须实现Listener接口的对象上的“通知”方法。通常，这会牵扯到大量的抽象和样板代码，让事件处理应用程序难以编写和阅读 。相反，在Scala中，事件是真正的对象，发送给订阅的组件，就像消息发送给actor那样。举例来说，按下按钮将会创建一个事件，即如下样本类的实例：

654

```
case class ButtonClicked(source: Button)
```

这个样本类的参数指向被点击的按钮。和所有其他 Scala Swing 事件一样，这个事件类包含在名为 scala.swing.event 的包中。

为了让你的组件对传入的事件作出响应，你须要向名为 reactions 的属性添加一个处理器。以下是执行这个动作的代码：

```
var nClicks = 0
reactions += {
  case ButtonClicked(b) =>
    nClicks += 1
    label.text = "Number of button clicks: "+ nClicks
}
```

上面的第一行定义了一个变量 nClicks，其内容是按钮被点击的次数。剩余的代码行将花括号中的代码作为处理器添加到 top 框架的 reactions 属性中。而处理器是通过对事件模式匹配定义出来的函数，就像 actor 的接收代码块是通过对消息模式匹配定义出来的一样。上述处理器匹配形式为 ButtonClicked(b)的事件，即 ButtonClicked 类的实例。模式变量 b 指向实际被点击的按钮。上述代码中与此事件相关的动作将 nClicks 加大并更新标签的文本。

总体来说，处理器是一个匹配事件并执行动作的 PartialFunction。也可以在单个处理器中用多个样本来匹配多种事件。

reactions 属性实现了一个集合类，就像 contents 属性那样。有些组件自带了预定义的响应方法。举例来说，Frame 有一个预定义的响应，如果用户按下右上角的关闭按钮，则该框架会关闭。如果你通过+=安装自己的响应方法到 reactions 属性的话，你定义的响应方法会在标准的响应方法之外也被考虑进去。从概念上讲，reactions 中安装的处理器形成一个栈。在当前这个示例中，当 top 框架接收到一个事件时，第一个被尝试的处理器是匹配 ButtonClicked 的那一个，因为它是为该框架安装的最后一个处理器。如果收到的事件类型为 ButtonClicked，与这个模式相关联的代码将会被调用。该段代码执行完成之后，系统将会在事件栈中继续查找更多可能满足条件的处理器。如果收到的事件类型不是 ButtonClicked，事件则会立即被发往剩余的已安装处理器栈。我们也可以用-=操作符将处理器从 reactions 属性中移除。

清单 32.3 给出了完整的应用程序，包括响应方法。这段代码显示了用 Scala 的 Swing 框架编写的 GUI 应用程序的重要元素：应用程序由一棵组件树构成，从 top 框架开始。代码中显示的组件有 Frame、BoxPanel、Button 和 Lavel，不过在 Swing 库中还定义了许多其他类型的组件。每个组件都可以通过设置属性来定制。两个重要的属性分别是 contents，对应组件在树中的子节点，以及 reactions，决定了组件如何响应事件。

```
import scala.swing._
import scala.swing.event._

object ReactiveSwingApp extends SimpleGUIApplication {
  def top = new MainFrame {
    title = "Reactive Swing App"
    val button = new Button {
      text = "Click me"
```

```
import swing._
import event._

object TempConverter extends SimpleGUIApplication {
  def top = new MainFrame {
    title = "Celsius/Fahrenheit Converter"
    object celsius extends TextField { columns = 5 }
    object fahrenheit extends TextField { columns = 5 }
    contents = new FlowPanel{
      contents += celsius
      contents += new Label(" Celsius = ")
      contents += fahrenheit
      contents += new Label(" Fahrenheit")
      border = Swing.EmptyBorder(15, 10, 10, 10)
    }
    listenTo(celsius, fahrenheit)
    reactions += {
      case EditDone(`fahrenheit`) =>
        val f = fahrenheit.text.toInt
        val c = (f - 32) * 5 / 9
        celsius.text = c.toString
      case EditDone(`celsius`) =>
        val c = celsius.text.toInt
        val f = c * 9 / 5 + 32
        fahrenheit.text = f.toString
    }
  }
}
```

清单 32.4 温度转换器的一个实现

TempConverter 中的两个组件 celsius 和 fahrenheit 是 TextField 类的对象。在 Swing 中 TextField 是可以让你编辑单行文本的组件。它有默认的宽度，对应以字符计算的 columns 属性（两个对象中，均设为 5）。

TempConverter 的 contents 被装配为一个面板，包括了两个文本字段和两个解释字段是什么的标签。面板的类型是 FlowPanel，意思是它根据框架的宽度用一行或多行一个接一个地显示所有元素。

TempConverter 的 reactions 定义了一种包含两种情况的处理器。每种情况匹配对应到两个字段中的一个的 EditDone 事件。注意模式的形式，在元素名前后包括了反引号：

```
case EditDone(`celsius`)
```

我们在 15.2 节解释过，celsius 前后的反引号确保模式只当事件源为 celcius 对象时才匹配。如果你漏掉反引号，只写 case EditDone(celcius)的话，模式将匹配所有类型为 EditDone 的事件。而改变后的字段会被存放到模式变量 celsius 中。显然，这不是你想要的。或者你也可以将两个 TextField 对象定义成以大写字符母开头，即 Celsius 和 Fahrenheit。这样你就可以直接匹配它们而不需要反引号，如 case EditDone(Celsius)。

EditDone事件的两个动作将其中一个数量转换为另一个数量。每一个转换都从读取修改后的字段的内容开始，并将它转换成Int。然后用公式将一种温度转换成另一种，并将结果作为另一个文本字段的字符串存回去。

32.5 小结

本章让你初尝了GUI编程，其中用到了Scala对Swing框架的包装。它显示了如何组装GUI组件，如何定制它们的属性，以及如何处理事件。受空间所限，我们只能讨论少量简单的组件，还有很多其他种类的组件。你可以查阅scala.swing包的Scala文档找到更多与它们相关的信息。下面的章节将会开发一个更复杂的Swing应用程序的示例。

659

Scala包装是以原始的Java Swing框架为基础的，这个基础框架也有许多教程[1]。Scala包装和它下层的Swing类很相似，不过它们尽可能地简化概念并让这些概念更加统一。这个简化的过程大量地使用了Scala语言的属性。举例来说，Scala对属性的模拟和它的操作符重载让我们可以方便地用赋值和+=操作定义属性。它的“一切皆对象”的哲学也让我们可以继承GUI应用程序的主方法。因而这个方法可以在用户应用程序中隐藏掉，包括它用来设置Swing的样板代码。最后，同时也是最重要的一点，Scala的头等函数和模式匹配让我们可以将事件处理逻辑公式化组织到reactions组件属性中，这大大简化了应用程序开发人员的工作。

660

[1]注：“比如可以参考The Java Tutorials”。[Jav]

第 33 章

Scell 试算表

在之前的章节中你看到了 Scala 编程语言许多不同的语法结构。在本章，你将会看到这些语法结构如何在一起实现一个大一些的应用程序。我们的任务是编写一个试算表应用程序，这个应用程序将会被命名为 SCells。

为什么说这项任务有趣有如下几个原因：首先，每个人都知道试算表，因此很容易理解这个应用程序应该做什么；其次，试算表会执行大量的不同的计算任务。有可视化的方面，试算表被看作是富 GUI 应用程序；有符号化的方面，这与公式及如何解析和翻译它们有关；有计算的方面，如何增量地更新可能会很大的表格；有响应的方面，试算表被看作是以错综复杂的方式响应事件的程序；最后，还有组件的方面，这个应用程序由一组可重用的组件构建起来。所有这些都会在本章做深入的介绍。

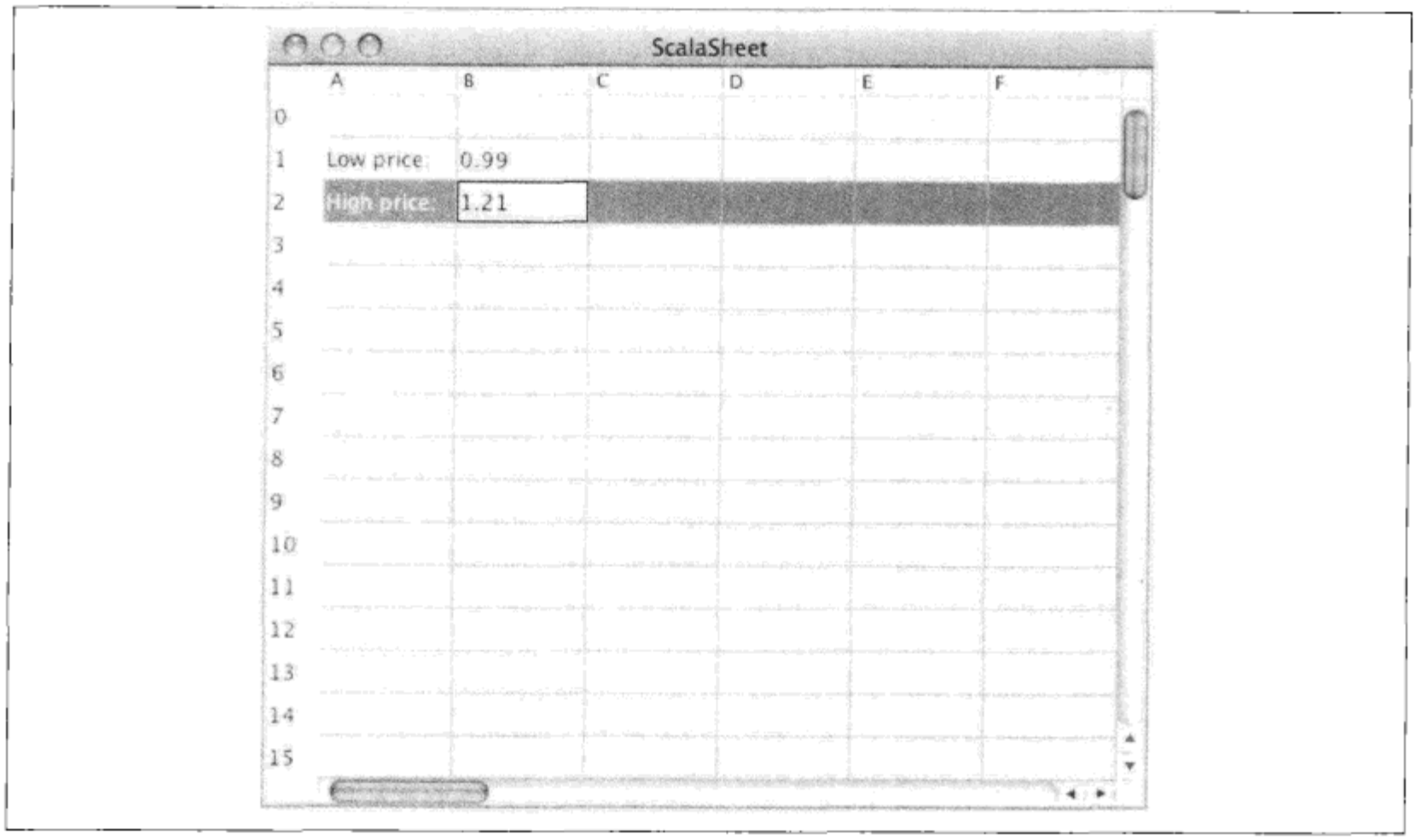
33.1 可视化框架

我们将从编写该应用程序基本的可视化框架开始。图 33.1 显示了第一版的用户界面。你可以看到试算表是一个可滚动的表格。它有从 0 到 99 的行及从 A 到 Z 的列。你可以通过将试算表定义为一个包含 Table 的 ScrollPane，以便在 Swing 中表达这个设计。清单 33.1 给出了代码。

清单 33.1 所示的试算表组件定义在 org.stairwaybook.scells 包中，这个包将会包含所有该应用程序所需的类、特质和对象。它从 scala.swing 包引入了 Scala 的 Swing 包中的重要元素。Spreadsheet 本身是一个将 height 和 width（以单元格数量计算）作为参数的类。这个类继承自 ScrollPane，带给它如图 33.1 所示的底部和右侧的滚动条。它包含了两个名为 table 和 rowHeader 的子组件。

table 组件是 scala.swing.Table 类的匿名子类的实例。定义体中的四行设置了它的一些属性：表示表格行以点数计算的高度 rowHeight、关闭表格自动调整大小功能的 autoResizeMode、显示单元格之间网格线的 showGrid、以及将网格的颜色设置为深灰色的 gridColor。

包含图 33.1 中的试算表左侧的行号头的 rowHeader 组件是一个 ListView，在其元素中显示从 0 到 99 的字符串。定义体中的两行（代码）将单元格的宽度固定为 30 点，这样高度与 table 的 rowHeight 相同。



662

图 33.1 一个简单的试算表表格

```
package org.stairwaybook.scells
import swing._

class Spreadsheet(val height: Int, val width: Int)
  extends ScrollPane {

  val table = new Table(height, width) {
    rowHeight = 25
    autoResizeMode = Table.AutoResizeMode.Off
    showGrid = true
    gridColor = new java.awt.Color(150, 150, 150)
  }

  val rowHeader =
    new ListView((0 until height) map (_.toString)) {
      fixedCellWidth = 30
      fixedCellHeight = table.rowHeight
    }

  viewportView = table
  rowHeaderView = rowHeader
}
```

清单 33.1 图 33.1 中的试算表的代码

整个试算表通过设置 ScrollPane 中的两个字段组装起来。viewportView 字段设置为 table，rowHeaderVew 设置为 rowHeader 列表。这两个视图的区别是滚动板的查看区是通过两个滚动条滚动的区域，而左边的行头在你移动水平滚动条时保持固定。通过一些技巧，Swing 默认已经在表格顶部提供了一个列头，因此不需要再显式地定义一个。

为尝试使用清单 33.1 中所示的初级试算表，你只需要定义一个创建 Spreadsheet 组件的主程序。清单 33.2 显示了这样一个程序。

Main 程序继承自 SimpleGUIApplication，负责所有需要在 Swing 应用程序运行之前设置好的底层细节。你只须在 top 方法中定义应用的顶级窗体即可。在我们的示例中，top 是定义了两个元素的 MainFrame：其 title 设置为“ScalaSheet”，而其 contents 设置为 100 行 26 列的类 Spreadsheet 的实例。这些就是全部的内容了。如果你用 scalaorg.stairwaybook.scells.Main 启动该应用程序，应该就能看到图 33.1 中的试算表。

```
package org.stairwaybook.scells
import swing._

object Main extends SimpleGUIApplication{
  def top = new MainFrame {
    title = "ScalaSheet"
    contents = new Spreadsheet(100, 26)
  }
}
```

清单 33.2 试算表应用程序的主程序

33.2 将数据录入和显示分开

如果你玩一玩目前写好的试算表，你很快就会注意到，单元格中显示的输出永远都和你在单元格中录入的一样。真正的试算表并不是这样的。在真正的试算表中，你可能会录入一个公式，然后看到的是它计算出的值。所以录入到单元格中的内容和显示的内容是不同的。

为了实现真正的试算表应用程序，你首先应该将注意力集中在如何将数据录入和显示分开的问题上。基本的显示机制包含在 Table 类的 rendererComponent 方法中。默认情况下，rendererComponent 总是显示录入的内容。如果你想改变这个逻辑，你需要重写 renderer-Component 来做不同的事。清单 33.3 显示了带有 rendererComponent 方法的新版 Spreadsheet。

这个 rendererComponent 方法重写了 Table 类中的一个默认方法。它接受四个参数。isSelected 和 hasFocus 参数是布尔型的，表示单元格是否被选中及是否有焦点，意味着键盘事件会进入该单元格。剩下的两个参数 row 和 column 给出单元格的坐标。


```
package org.stairwaybook.scells
import swing._

class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane{

  val cellModel = new Model(height, width)
  import cellModel._

  val table = new Table(height, width) {
    // 设置和之前一样……

    override def rendererComponent(isSelected: Boolean,
```

```
        hasFocus: Boolean, row: Int, column: Int): Component =
      if(hasFocus) new TextField(userData(row, column))
      else
        new Label(cells(row)(column).toString) {
          xAlignment = Alignment.Right
        }
    def userData(row: Int, column: Int): String = {
      val v = this(row, column)
      if (v == null) "" else v.toString
    }
  }
  // 其余内容和之前一样
}
```

清单33.3 带有rendererComponent方法的试算表

新的rendererComponent方法检查单元格是否有输入焦点，如果hasFocus为true，单元格就可以编辑。在这种情况下你会想要显示一个包含用户已录入数据的可编辑TextField。这个数据由助手方法userData返回，它显示表格中给定row和column的内容。内容通过调用this(row, column)获取[1]。userData方法同时也负责把null元素显示为空字符串而不是“null”。

到目前为止一切都好。不过如果单元格没有焦点应该显示什么呢？在真正的试算表中这会是单元格的值。因此，实际上工作中有两个表格。其中名为table的表格包含了用户录入的内容。另一个“影子”表格则包含了单元格的内部表示及应显示的内容。在试算表例子中，这个表格是一个名为cells的二维数组。如果给定row和column的单元格没有编辑焦点，rendererComponent方法将显示元素cells(row)(column)。元素不能被编辑，因此应该显示在Label而不是可编辑的TextField中。

剩下的是定义单元格的内部数组。你可以直接在Spreadsheet类中来做，不过通常更好的做法是让GUI组件的视图与内部模型分开。这就是为什么在上面的示例中cells数组定义在另一个名为Model的类中。我们通过定义类型为Model的cellModel值来将模型集成到Spreadsheet中。在这个val定义之后的import语句让cellModel的成员在Spreadsheet中直接可见而不需要加上前缀。清单33.4显示了Model类的第一个简化的版本。这个类定义了一个内部类Cell，以及一个二维数组cells存放Cell元素。每个元素都被初始化成全新的Cell。

```
package org.stairwaybook.scells

class Model(val height: Int, val width: Int) {

  case class Cell(row: Int, column: Int)

  val cells = new Array[Array[Cell]](height, width)

  for (i<- 0 until height; j <- 0 until width)
    cells(i)(j) = new Cell(i, j)
}
```

清单33.4 第一版的Model类

[1]注：虽然“this(row, column)”可能看上去与构造方法调用很像，但本例中调用的是当前Table实例的apply方法。

就是这样了。编译这个混入了 Model 特质的 Spreadsheet 类并运行 Main 应用程序，你会看到如图 33.2 所示的一个窗体。

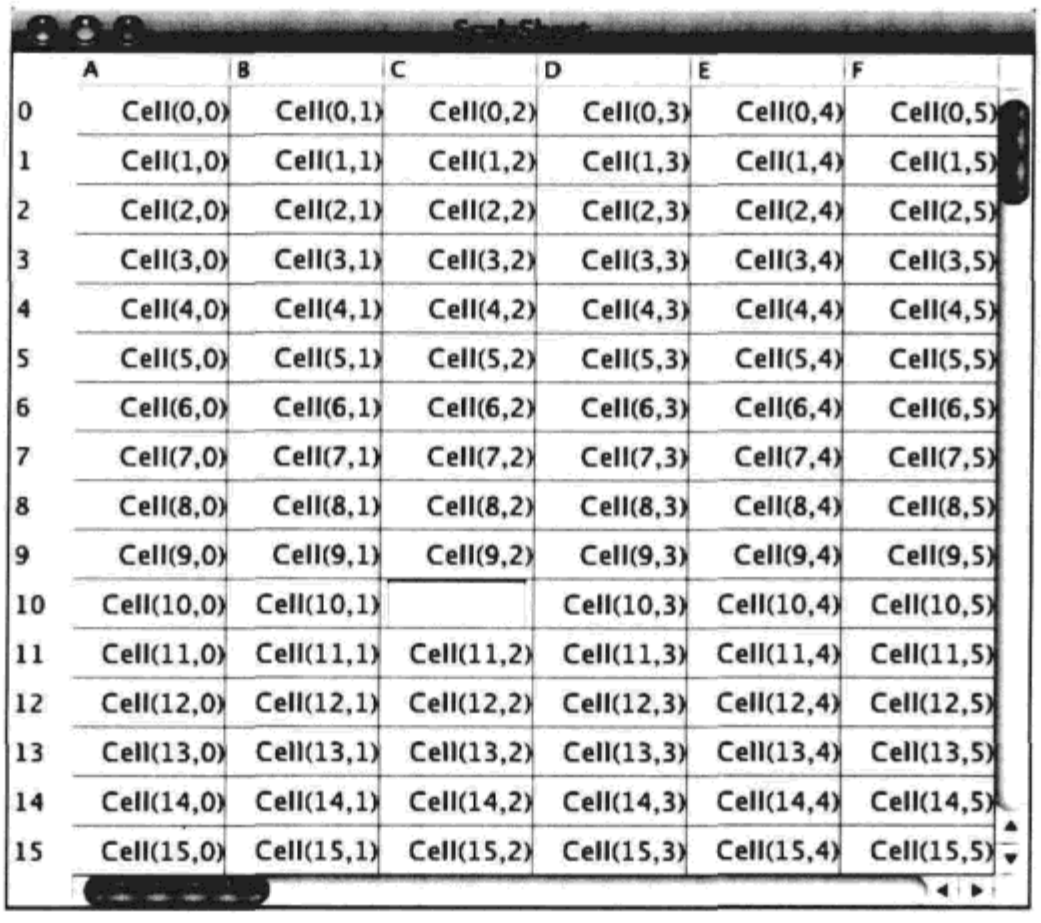


图 33.2 显示自身的单元格

本节的目标是做出这样一个设计：单元格显示的内容与录入进去的字符串不同。这个目标显然达成了，不过实现很粗糙。在新的试算表中，你可以在单元格中录入任何想录入的内容，但它一旦失去焦点将总是只显示它的坐标。显然，我们还有工作要做。

33.3 公式

现实生活中，试算表的单元格持有两个东西：实际的值和计算该值的公式。试算表中可能有三种类型的公式：

1. 数值如 1.22、-3 或 0。
2. 文本标签如 Annual sales、Deprecation 或 total。
3. 从单元格的内容计算出新值的公式，如“=add(A1,B2)”或“=sum(mul(2, A2), C1:D16)”。

计算值的公式总是以等号开始，然后是一个算术表达式。SCells 试算表有一个特别简单和统一的算术表达式规约：每个表达式都是对一组参数应用某个函数。函数名是一个标识符，如做二元加法的 add，或做任意多数量的操作元求和的 sum。函数的参数可以是数字、对某个单元格的引用，对某个区

间的单元格如 C1:D16 的引用，或者是另一个函数应用。你将在稍后看到 SCell 有一个开放式的架构，让你可以很容易通过混入组合安装你自己的函数。处理公式的第一步是写出代表它们的类型。你可能想到了，不同类型的公式由样本类表示。清单 33.5 给出了名为 Formulas.scala 的文件的内容，这些样本类就定义在这里：

```
package org.stairwaybook.scells

trait Formula

case class Coord(row: Int, column: Int) extends Formula {
  override def toString = ('A' + column).toChar.toString + row
}
case class Range(c1: Coord, c2: Coord) extends Formula {
  override def toString = c1.toString +":"+ c2.toString
}
case class Number(value: Double) extends Formula {
  override def toString = value.toString
}
case class Textual(value: String) extends Formula {
  override def toString = value
}
case class Application(function: String,
    arguments: List[Formula]) extends Formula {

  override def toString =
    function + arguments.mkString("(", ",", ")")
}
object Empty extends Textual("")
```

清单 33.5 代表公式的类

669

清单 33.5 所示的类继承关系的根是 Formula 特质。这个特质有如下五个样本类：

Coord	用于表示单元格坐标如 A3。
Range	用于表示单元格区间如 A3:B17。
Number	用于表示浮点数如 3.1415。
Textual	用于表示文本标签如 Deprecation。
Application	用于函数应用如 sum(A1, A2)。

每个样本类都重写了 toString 方法，因此它按照上述的标准方式显示其对应类型的公式。为方便起见还有一个 Empty 对象，表示空单元格的内容。Empty 对象是 Textual 类的实例，以一个空字符串作为参数。

33.4 解析公式

在前一节，你看到不同类型的公式及它们如何显示为字符串。在本节你将会看到如何将这个过程反过来：也就是说，如何将用户输入的字符串转换成 Formula 树。本节剩余的部分将逐一讲解

FormulaParsers 类的不同元素，这个类包含了执行转换的解析器。该类基于第 31 章给出的连结符框架构建。具体而言，公式解析器是在那一章讲到的 RegexParsers 类的实例：

```
package org.stairwaybook.scells
import scala.util.parsing.combinator._

object FormulaParsers extends RegexParsers{
```

FormulaParsers 类的头两个元素是针对标识符和十进制数的辅助解析器：

```
  def ident: Parser[String] = """[a-zA-Z_]\w*""".r
  def decimal: Parser[String] = """-?\d+(\.\d*)?""".r
```

从上面第一个正则表达式你可以看到，标识符由字母或下划线打头，紧跟着是任意数量的由正则表达式代码\w 表示的“词”，它能够识别出的内容是字母、数字或下划线。第二个正则表达式描述了十进制数，由可选的减号、一个或多个由正则表达式\d 表示的数字、以及一个可选的由句点和 0 到多个数字组成的小数部分构成。

FormulaParsers 类的下一个元素是 cell 解析器，它识别出单元格的坐标，如 C11 或 B2。它首先调用决定坐标的格式的正则表达式解析器，单个字母加上一个或多个数字。从这个解析器返回的字符串随后被转换成单元格的坐标，方法是先将字母和数字部分分开，然后将这两个部分分别转换成单元格的列和行的下标。

```
  def cell: Parser[Coord] =
    """[A-Za-z]\d+""".r ^^ { s =>
      val column = s.charAt(0) - 'A'
      val row = s.substring(1).toInt
      Coord(row, column)
    }
```

注意 cell 解析器限制较多，只允许列坐标有一个字母。因此试算表的列数量事实上被限制在最多 26 个，因为无法解析更多的列。将这个解析器变得更通用，接受多个起始字母是个不错的主意。这留给你作为练习吧。

range 解析器能够识别出一个区间的单元格。这样的一个区间由两个单元格的坐标组成，以冒号隔开：

```
  def range: Parser[Range] =
    cell~":"~cell ^^ {
      case c1~":"~c2 => Range(c1, c2)
    }
```

number 解析器能够识别十进制数，识别出的数字被转换成一个 Double 并包装在 Number 类的实例中：

```
  def number: Parser[Number] =
    decimal ^^ (d => Number(d.toDouble))
```

application 解析器能够识别出函数应用。这样的应用由标识符加上一组用括弧括起来的参数表达式组成：

```
def application: Parser[Application] =
  ident~"("~repsep(expr, ",")~")" ^^ {
    case f~"("~ps~")" => Application(f, ps)
  }
```

670

expr 解析器能够识别公式表达式——不论它是由‘=’打头的顶级公式还是某个函数的参数。这样一个公式表达式定义为一个单元格、一个单元格区间、一个数字或一个应用：

```
def expr: Parser[Formula] =
  range | cell | number | application
```

这个 expr 解析器的定义有一些过于简单，因为单元格区间只应作为函数参数出现；它们不应被允许为顶级公式。你可以修改公式的语法以便将表达式这两种用法分开，把区间在语法上从顶级公式中排除。在这里展示的试算表中，这样的错误在表达式被求值时才被检测到。

textual 解析器能够识别任意的输入字符串，只要它不以等号打头（你应该还记得‘=’打头的字符串被认为是公式）：

```
def textual: Parser[Textual] =
  """[^=].*""".r ^^ Textual
```

formula 解析器能够识别单元格的各种合法输入。公式可以是一个 number、一个 textual 条目、或者以等号开始的公式：

```
def formula: Parser[Formula] =
  number | textual | "="~>expr
```

以上就是试算表单元格的全部语法。最后的 parse 方法在一个将输入字符串转换为 Formula 树的方法中用到了这组语法：

```
  def parse(input: String): Formula =
    parseAll(formula, input) match {
      case Success(e, _) => e
      case f: NoSuccess=>Textual("["+ f.msg +"]")
    }
} // FormulaParsers 到此结束
```

671

parse 方法用 formula 解析器解析所有的输入。如果成功了，返回得到的公式。如果失败了，则返回一个带有错误提示的 Textual 对象。解析公式的全部内容见清单 33.6。

```
package org.stairwaybook.scells
import swing._
import event._

class Spreadsheet(val height: Int, val width: Int) ... {
  val table = new Table(height, width) {
    ...
```

```
      reactions += {
        case TableUpdated(table, rows, column) =>
          for (row <- rows)
            cells(row)(column).formula =
              FormulaParsers.parse(userData(row, column))
      }
    }
  }
```

清单 33.6 解析公式的试算表

唯一剩下的事情是将解析器集成到试算表中。为此，你可以在 Model 类中通过 formula 字段来丰富 Cell 类：

```
case class Cell(row: Int, column: Int) {
  var formula: Formula = Empty
  override def toString = formula.toString
}
```

在新版的 Cell 类中，我们定义了 toString 方法来显示单元格的 formula。这样你可以检查公式是否被正确的解析。

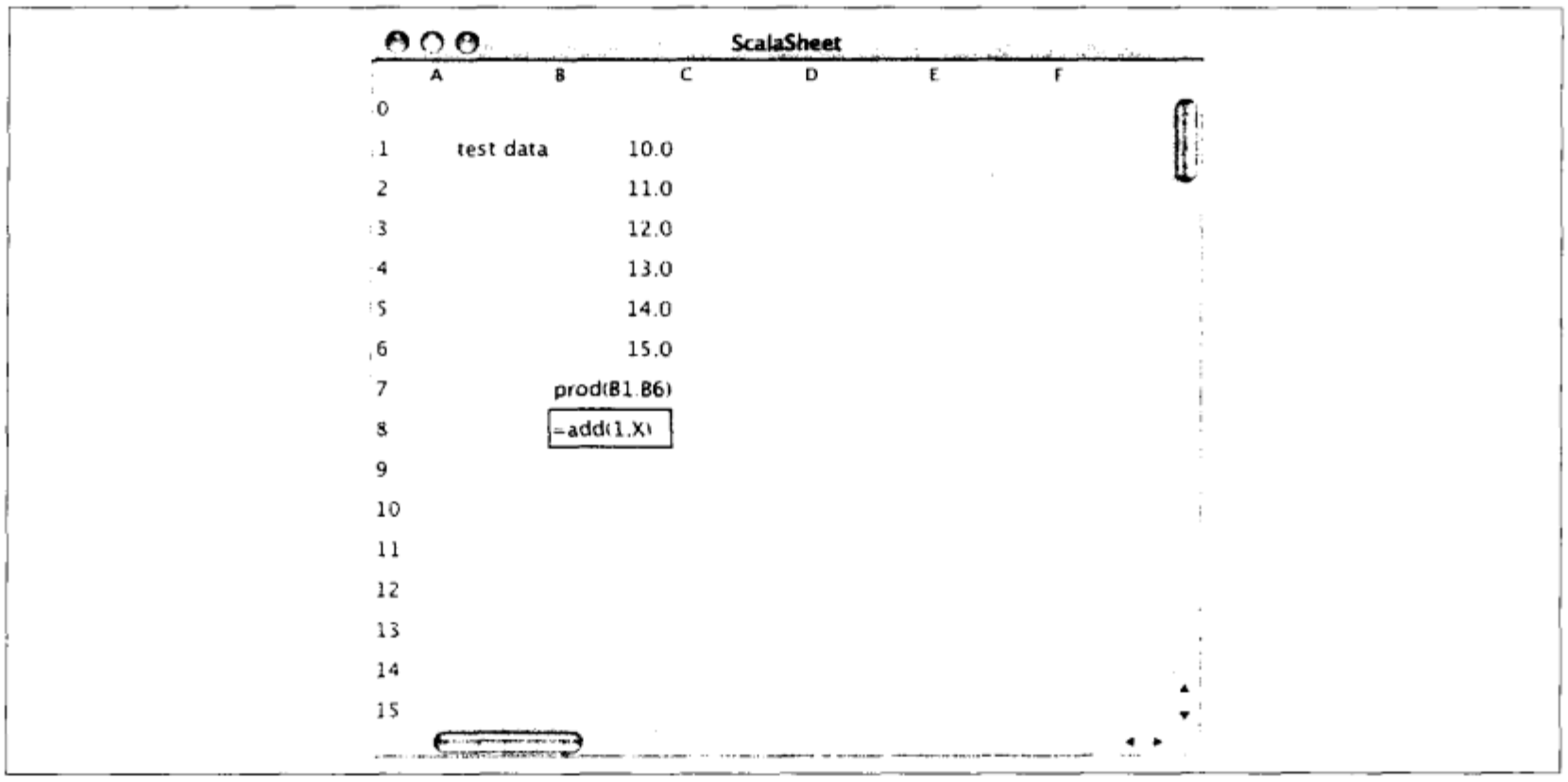
本节的最后一步是将解析器集成到试算表中。对公式的解析是作为对用户向单元格进行输入操作的响应发生的。完整的单元格输出在 Swing 库中通过 TableUpdated 事件来捕获的。TableUpdated 类包含在 scala.swing.event 包中。事件的格式为：

```
TableUpdated(table, rows, column)
```

它包含了被改变的 table，以及一组以 rows 和 column 给出的受到影响的单元格的坐标。参数 rows 是一组类型为 Range[Int][2]的值。参数 column 是一个整数。因此总体而言 TableUpdated 事件可以指向多个受影响的单元格，不过它们只能是在同一列的连续行这样的区间。

一旦表格被改变，受到影响的单元格需要被重新解析。为响应 TableUpdated 事件，你需要给 table 组件的 reactions 值增加一个样本，如清单 33.6 所示。现在，任何时候 table 被编辑，所有受影响的单元格都会被通过解析相关用户数据来更新。编译目前为止讨论到的类并启动 scells.Main 应用程序，你应该看到如图 33.3 所示那样的试算表应用程序。你可以通过在单元格键入内容来编辑它们。编辑完成后，单元格显示出它包含的公式。你还可以试着键入一些不合法的输入，比如在图 33.3 中有编辑焦点的字段中的=add(1, X)。非法的输入会显示为错误消息。举例来说，一旦你离开图 33.3 中编辑后的字段，就应该在单元格中看到这样的错误消息：[`(' expected]（为了看到全部的错误消息，你可能需要向右拖拽列头间的分隔线来拉宽该列）。

[2]注：Range[Int]也是诸如“1 to N”这样的 Scala 表达式的类型。



673

图 33.3 显示方程式的单元格

33.5 求值

当然，最终试算表应该对公式求值，而不只是简单地显示它们。在本节中，我们将添加必要的组件来完成这个任务。

我们需要的是一个 `evaluate` 方法，接受一个公式，并返回它在当前试算表中的值，以 `Double` 表示。我们将把这个方法放在新的特质 `Evaluator` 中。这个方法须要访问 `Model` 类中的 `cells` 字段来获知公式中引用到的单元格的当前值。另一方面，`Model` 类须要调用 `evaluate`。因此，`Model` 和 `Evaluator` 之间相互依赖。第 27 章给出了表达这样的类间相互依赖关系的一个好方法：在其中的一个方向上使用继承，而在另一个方向上使用自身类型。

在试算表的示例中，`Model` 类继承 `Evaluator` 从而得到对其 `evaluation` 方法的访问。反过来，`Evaluator` 类定义它的自身类型为 `Model`，就像下面这样：

```
package org.stairwaybook.scells
trait Evaluator { this: Model => ...
```

这样一来，`Evaluator` 类中的 `this` 值就被认为是 `Model`，而 `cells` 数组可以通过写出 `cells` 或 `this.cells` 访问到。

现在既然线路已经铺就，我们将集中精力来定义 `Evaluator` 类的内容。清单 33.7 给出了 `evaluate` 方法的实现。正如你可能会预期的那样，这个方法包含了对不同类型的公式的模式匹配。对坐标 `Coord(row, column)`，它返回 `cells` 数组中在那个坐标的值。对数字 `Number(v)`，它返回值 `v`；对文本标签 `Textual(s)`，它返回零。最后，对应用 `Application(function, arguments)`，它计算出所有参数的值；从 `operations` 表获取与 `function` 名称相对应的函数对象并将这个函数应用到所有参数值上。与 `function` 名称相对应的函数对象，并将这个函数应用到所有参数值上。

674

```
def evaluate(e: Formula): Double = try {
  e match {
    case Coord(row, column) =>
      cells(row)(column).value
    case Number(v) =>
      v
    case Textual(_) =>
      0
    case Application(function, arguments) =>
      val argvals = arguments flatMap evalList
      operations(function)(argvals)
  }
} catch {
  case ex: Exception => Math.NaN_DOUBLE
}
```

清单 33.7 Evalutor 特质的 evaluate 方法

operations 表将函数名映射到函数对象。它是这样定义的：

```
type Op = List[Double] => Double
val operations = new collection.mutable.HashMap[String, Op]
```

从这个定义中你可以看到，操作被建模为从值列表到值（对象）的函数。Op 类型引入了操作类型的一个便于使用的别名。

evaluate 的计算被包装在 try-catch 中以防止输入错误。实际上在对单元格求值时有相当多的环节可能会出问题：坐标可能超出区间了，函数名可能没有被定义，函数的参数个数可能不对，算术操作可能非法或者溢出。对任何这些错误的应对方式都是相同的：返回一个“not-a-number”的值。返回的值 Math.NaN_DOUBLE 是 IEEE 对没有可以用来表示它的浮点值的计算的表示方式。这之所以可能发生，是由于比方说出现了溢出或除以零的情况。清单 33.7 的 evaluate 方法选择对所有其他类型的错误也返回相同的这个值。这样做的好处是它理解起来很简单，也不需要很多代码就能实现。缺点是所有类型的错误都混在一起，因此试算表的用户并不会得到任何关于出了什么错的细节反馈。如果你想，也可以尝试一些更好的方式来表示 SCells 应用程序的错误。

对参数的求值不同于顶级公式的求值。参数可能是列表而顶级公式不是这样。举例来说，sum(A1:A3)的参数表达式 A1:A3 用列表返回单元格 A1、A2 和 A3 的值。这个列表被传递到 sum 操作中。在参数表达式中混用列表和单值也是可以的，例如操作 sum(A1:A3, 1.0, C7)，它会对五个值求和。为了处理可能被求值为列表的参数，有另一个名为 evalList 的求值函数。该函数接受一个公式，返回一个值的列表。它的定义如下：

```
private def evalList(e: Formula): List[Double] = e match {
  case Range(_, _) => references(e) map (_.value)
  case _ => List(evaluate(e))
}
```

如果传递给 evalList 的公式参数是一个 Range，则返回的值是一个由该区间引用到的所有单元格的值组成的列表。对其他的每个公式，结果都是由那个公式的单个返回值组成的列表。公式应用到的

675

单元格则是由另外一个函数 references 计算出来的。它的定义如下：

```
def references(e: Formula): List[Cell] = e match {
  case Coord(row, column) =>
    List(cells(row)(column))
  case Range(Coord(r1, c1), Coord(r2, c2)) =>
    for(row <- (r1 to r2).toList; column <- c1 to c2)
    yield cells(row)(column)
  case Application(function, arguments) =>
    arguments flatMap references
  case _ =>
    List()
}
} // Evaluator 到此结束
```

676

references 方法实际上比我们现在所需要的更通用，它能计算出任何类型的公式引用的单元格，而不只是 Range 公式。稍后我们会看到这个多出来的功能对于计算出需要更新的单元格组而言是有必要的。方法的主体是对公式类型的简单模式匹配。对坐标 Coord(row, column)，它返回包含那个坐标的单元格的单元素列表。对区间表达式 Range(coord1, coord2)，它返回两个坐标之间以一个 for 表达式计算出来的所有单元格。对函数应用 Application(function, arguments)，它返回每个参数表达式引用到的单元格，并通过 flatMap 拼接为单个列表。对于其他两类公式，Textual 和 Number，它返回空的列表。

33.6 操作库

Evaluator 类本身并没有定义可以在单元格上执行的操作：它的 operations 表在一开始是空的。这背后的想法是在其他特质中定义这些操作，然后这些特质再被混入到 Model 类中。清单 33.8 给出了一个实现了通用算术操作的示例特质：

```
package org.stairwaybook.scells
trait Arithmetic { this: Evaluator =>
  operations += (
    "add"    -> { case List(x, y) => x + y },
    "sub"    -> { case List(x, y) => x - y },
    "div"    -> { case List(x, y) => x / y },
    "mul"    -> { case List(x, y) => x * y },
    "mod"    -> { case List(x, y) => x % y },
    "sum"    -> { xs => (0.0 /: xs)(_ + _) },
    "prod"   -> { xs => (1.0 /: xs)(_ * _) }
  )
}
```

清单 33.8 一个算术操作库

677

有趣的是，这个特质没有输出的成员。它做的唯一一件事是在它初始化的过程中填充 operations

表。它用自身类型 Evaluator 访问那个表，也就是说，使用的是和 Arithmetic 类访问模型相同的技巧。

Arithmetic 特质定义的 7 个操作当中，有五个是二元操作，两个接受任意数量的参数。二元操作采用相同的结构。举例来说，加操作 add 由如下表达式定义：

```
{ case List(x, y) => x + y }
```

也就是说，它预期一个由两个元素 x 和 y 组成的列表，并返回 x 和 y 的和。如果参数列表包含的元素个数不是两个的话，会抛出 MatchError。这与 SCell 求值模型的“让它失败”的哲学有关，错误的输入会造成运行期异常，然后这个异常被 evaluate 方法内的 try-catch 捕获。

最后两个操作，sum 和 prod，接受任意长度的参数列表，然后在相邻的元素间插入二元操作。因此它们是“左合”的实例，在 List 类中表示为/:操作。举例来说，为了对数字列表 List(x, y, z) 做 sum，操作结算为 0 + x + y + z。第一个操作元 0 是列表为空时的结果。

你可以通过将 Arithmetic 特质混入到 Model 类的方式将这个操作库集成到试算表应用程序中，就像这样：

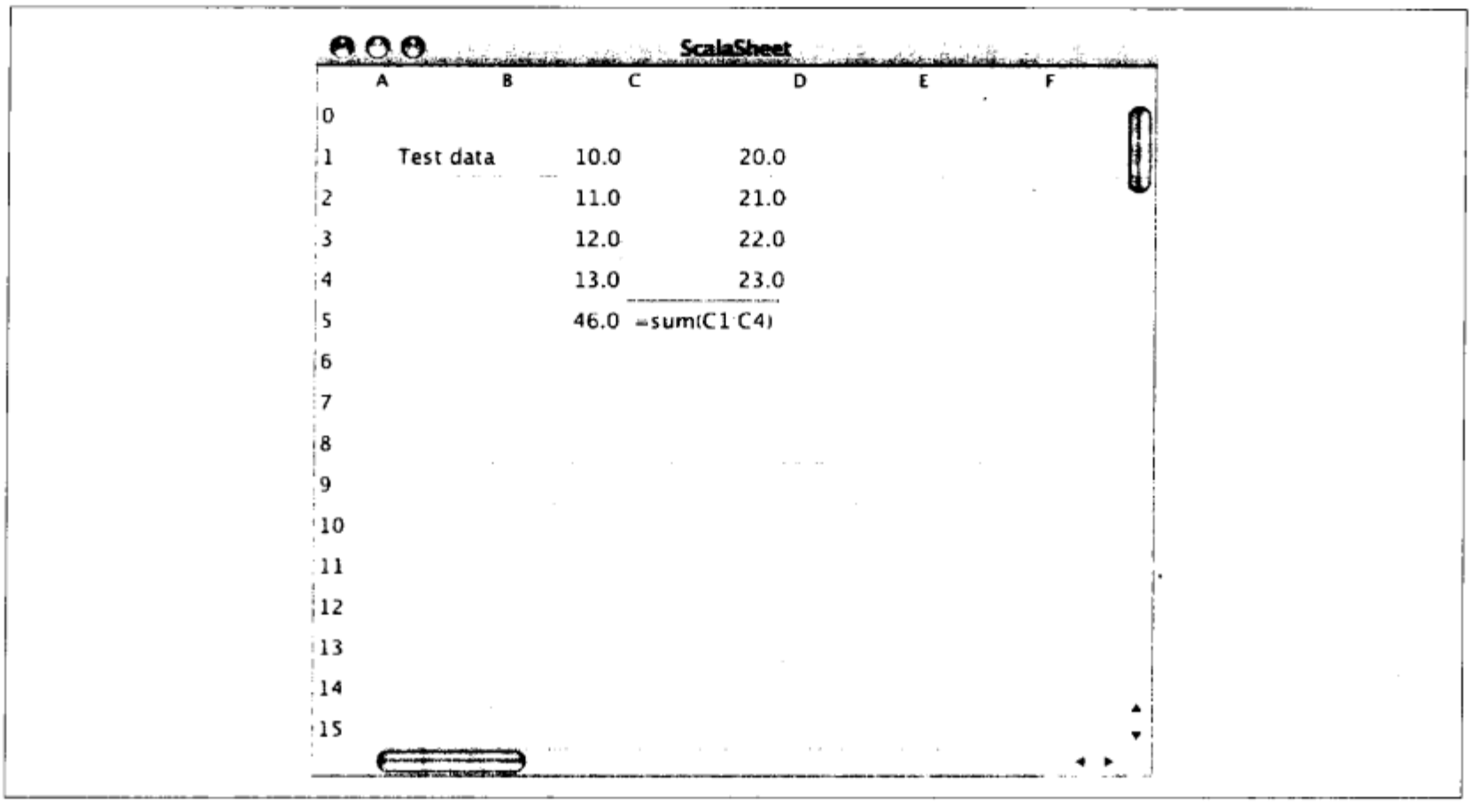
```
package org.stairwaybook.scells

class Model(val height: Int, val width: Int)
    extends Evaluator with Arithmetic {
  case class Cell(row: Int, column: Int) {
    var formula: Formula = Empty
    def value = evaluate(formula)

    override def toString = formula match {
      case Textual(s) => s
      case _ =>value.toString
    }
  }
  ... //其余和之前相同
}
```

另一个对 Model 类的修改是关于单元格如何显示它们自己。在新版中，单元格的显示值取决于它的公式。如果公式是一个 Textual 的字段，则字段内容按照字面显示。其余情况下公式都会被求值，求值得到的结果 value 会被显示出来。

如果你编译这些修改后的特质和类并重新启动 Main 程序，你得到的东西开始像个真的试算表了。图 33.4 显示了一个示例。你可以将公式录入到单元格然后让它们自己求值。举例来说，一旦你关闭图 33.4 中 C5 单元格的编辑焦点，你应该会看到 86.0，这是对公式 sum(C1:C4)求值的结果。



679

图 33.4 可以求值的单元格

不过，我们还缺一个重要的元素。如果你将图 33.4 中的单元格 C1 的值从 20 改成 100，单元格 C5 中的和并不会自动被更新到 166。你必须手工点击 C5 才能看到它的值的变化。仍然缺少的是，让单元格在某次修改后自动重新计算它们的值的方式。

33.7 修改传达

如果单元格的值被修改，所有依赖于这个值的单元格都应该重新计算它们的结果并重新显示出来。实现这个最简单的方式是在每次修改后重新计算试算表中的每一个单元格。不过随着试算表的规模越来越大，这样的方式伸缩性并不好。

更好的方式是只重新计算那些在它们的公式中引用到并被修改的单元格。我们的想法是用一个基于事件的发布/订阅框架来传达修改：一旦单元格被赋值了一个公式，它将会订阅该公式引用的所有单元格，在它们的值被修改时得到通知。某个单元格的值变化都会触发订阅单元格的重新求值。如果这样的重新求值造成了这个单元格的值的变化，它会转而通知所有依赖于它的单元格。这个过程不断继续直到所有单元格的值都稳定下来，也就是说，没有任何单元格有更多的值变化[3]。

这个发布/订阅框架在 Model 类中用 Scala Swing 框架标准的事件机制实现。以下是该类新的（最终的）版本：

```
package org.stairwaybook.scells
import swing._

class Model(val height: Int, val width: Int)
extends Evaluator with Arithmetic {
```

[3]注：这里假定单元格间没有循环依赖。我们在本章最后讨论如何去掉这个假定。

和之前版本的 Model 相比，这个版本增加了一个新的对 swing._的引入，让 Swing 的事件抽象直接可用。

Model 类的主要修改和嵌套类 Cell 有关。Cell 类现在继承自 Publisher，因此它可以发布事件。事件处理逻辑被完整地包含在两个属性的 setter 方法中：value 和 formula。以下是新版本的 Cell：

680

```
case class Cell(row: Int, column: Int) extends Publisher {
  override def toString = formula match {
    case Textual(s) => s
    case _ =>value.toString
  }
```

对外部而言，看上去 value 和 formula 是 Cell 类的两个变量。它们的实际实现是用装配有公共的 getter 方法 value 和 formula 及 setter 方法 value_=和 formula_=的私有字段实现的。以下是实现 value 属性的定义：

```
private var v: Double = 0
def value: Double = v
def value_=(w: Double) {
  if (!(v == w || v.isNaN && w.isNaN)) {
    v = w
    publish(ValueChanged(this))
  }
}
```

value_=这个 setter 方法将一个新的值 w 赋值给私有字段 v。如果新的值和原来的不同，它还会发布一个以单元格自身作为参数的事件 ValueChanged。注意对值是否有变化的测试比较微妙，因为它涉及到 NaN 这个值。根据 Java 规范，NaN 和其他所有值都不相同，包括它自己！因此，测试两个值是否相同需要对 NaN 进行特殊处理：两个值 v 和 w，如果它们按照==是相等的，或者它们的值均为 NaN，即 v.isNaN 和 w.isNaN 均得到 true，则它们是相同的。

value_=这个 setter 方法负责发布/订阅框架中的发布，formula_=这个 setter 方法则负责订阅：

```
private var f: Formula = Empty
  def formula: Formula = f
  def formula_=(f: Formula) {
    for (c <- references(formula)) deafTo(c)
    this.f = f
    for (c <- references(formula)) listenTo(c)
    value = evaluate(f)
  }
```

681

如果一个单元格被赋予新的公式，它首先用 deafTo 取消对之前的 formula 值引用的单元格的订阅。然后将新的公式存放到私有变量 f 中并用 listenTo 订阅所有它引用到的单元格。最后，它重新用新的公式计算它的值。

修改后的 Cell 的最后一段代码指定如何响应 ValueChanged 事件：

```
  reactions += {
    case ValueChanged(_) => value = evaluate(formula)
```

```
  }
} // 类到此结束
```

ValueChanged类同样位于Model类中：

```
case class ValueChanged(cell: Cell) extends event.Event
```

Model类其余部分和之前一样：

```
  val cells = new Array[Array[Cell]](height, width)
  for (i <- 0 until height; j <- 0 until width)
    cells(i)(j) = new Cell(i, j)
} // Model 类到此结束
```

现在试算表的代码基本上完整了。最后还缺失的一块是修改后的单元格的重显示。到目前为止，所有值传达都只关心到内部的Cell值，可见的表格没有收到影响。改变这个的一种方式是在value_=这个setter方法中加上redraw命令。不过，这会破坏你目前一路看下来的模型和视图间的严格区分。更模块化的解决方案是将所有ValueChanged事件通知表格，让它自己处理重新绘制的动作。清单33.9给出了最终的试算表组件，实现了这个设计。

清单33.9的Spreadsheet类只做了两处修改。首先，table组件现在用listenTo订阅模型中的所有单元格；其次，在表格的响应操作中增加了一个case：如果它接收到ValueChanged(cell)通知，它就会通过调用updateCell(cell.row, cell.column)要求相关的单元格进行重新绘制。

682

```
package org.stairwaybook.scells
import swing._, event._

class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane {

  val cellModel = new Model(height, width)
  import cellModel._

  val table = new Table(height, width) {
    ... // 设置同清单33.1

    override def rendererComponent(
        isSelected: Boolean, hasFocus: Boolean,
        row: Int, column: Int) =
      ... // 同清单33.3
    def userData(row: Int, column: Int): String =
      ... // 同清单33.3
    reactions += {
      case TableUpdated(table, rows, column) =>
        for (row <- rows)
          cells(row)(column).formula =
            FormulaParsers.parse(userData(row, column))
      case ValueChanged(cell) =>
        updateCell(cell.row, cell.column)
```

```
    }
    for (row <- cells; cell <- row) listenTo(cell)
  }

  val rowHeader = new ListView(0 until height) {
    fixedCellWidth = 30
    fixedCellHeight = table.rowHeight
  }

  viewportView = table
  rowHeaderView = rowHeader
}
```


清单 33.9 完成的试算表组件

33.8 小结

我们在本章中开发的试算表的功能是完整的，尽管有些地方它采用了最简单的方案来实现而不是对用户而言最方便的。这样，它可以在不到 200 行代码中完成。尽管如此，试算表的架构使得修改和扩展都很容易。以免你想要更进一步试验这个代码，以下是一些建议，关于什么是你可以试着修改或添加的：

1. 你让试算表可以调整大小，行数和列数都能被交互式地修改。

2. 你可以增加新的公式，比如二元操作或其他函数。

3. 你可以想想看如果单元格递归地引用到它们自己我们该怎么办。举例来说，如果单元格 A1 握有公式 add(B1, 1)而单元格 B1 握有公式 mul(A1, 2)，对它们当中任何一个进行求值都会触发栈溢出。显然，这不是一个很好的方案。作为备选，你可以不允许这种情况出现，或简单地对涉及到的单元格只做一次迭代的计算。

4. 你可以增强错误处理机制，给出更详细的信息，描述出了什么问题。

5. 你可以在试算表的顶部增加一个公式录入字段，以便长的公式可以被更方便地录入。

在本书的开始我们强调了 Scala 的可伸缩性。我们声称 Scala 对面向对象和函数式编程结构的结合让它适合于各种程序，从小型的脚本到十分庞大的系统。这里展示的试算表显然还只是一个小的系统，虽然在大多数别的语言中可能远不止 200 行代码。尽管如此，在这个应用程序中，你可以看到许多让 Scala 可伸缩的细节。

试算表用到了 Scala 的类和特质，并用它们的混入组合方式来灵活地装配其组件。组件间的递归依赖通过自身类型表述。对静态状态的需求完全消除了——唯一不是类的顶级组件是公式树和公式解析器，而它们都是纯函数式的。该应用还用到大量的高阶函数和模式匹配，既是为访问公式也是为事件处理。因此它是对函数式和面向对象编程可以如何平滑地结合在一起的很好的案例。

试算表应用程序之所以如此精简，一个很重要的原因是它以强大的类库作为基础。解析器连结符库

实际上提供了内部的领域特定语言类编写解析器。没有它，解析公式就会困难得多。Scala 的 Swing 库中的事件处理是流程控制抽象的强大能力的很好的例子。如果你了解 Java 的 Swing 库，你大概会对 Scala 的响应的概念大加赞赏，尤其是在与经典的发布/订阅设计模式编写通知，并实现监听器接口的冗长乏味相比较时。试算表展示了可扩展性带来的好处，使得高级别的库可以看上去就像是语言扩展一样。

685

附录 A

Unix 和 Windows 的 Scala 脚本

如果你用的是 Unix 的某个发行版，通过在文件顶端前缀一个“制式”标志（pound bang），你可以像运行 shell 脚本那样运行 Scala 脚本。例如，可以把以下内容输入文件 helloarg：

```
#!/bin/sh
exec scala "$0" "$@"
!#
// 对第一个参数打招呼
println("Hello, " + args(0) + "!")
```

头文字#!/bin/sh 必须在文件里最开头一行。一旦你设置了它的执行许可：

```
$ chmod +x helloarg
```

你就可以让 Scala 脚本像 shell 脚本那样运行，输入：

```
$ ./helloarg globe
```

如果你在 Windows 平台，可以把文件命名为 helloarg.bat，并把以下内容放在脚本头，这样就能达到同样效果：

```
::#!
@echo off
call scala %0 %*
goto :eof
::!#
```

术语表

代数数据类型（algebraic data type）。通过提供若干个含有独立构造器的备选项来定义的类型。通常可以辅助于通过模式匹配解构类型的方式。这个概念可以在规约语言和函数式编程语言中发现。代数数据类型在 Scala 里可以用样本类（case class）模拟。

备选项（alternative）。`match` 表达式的分支。形式为“`case` pattern => expression”。备选项的别名是样本（case）。

注解（annotation）。注解出现在源代码中，并被附加在某个语法部分上。注解可以用计算机处理，所以你可以用它们有效地添加 Scala 的扩展。

匿名类（anonymous class）。匿名类是由 Scala 编译器通过类或特质（trait）名跟花括号内的 `new` 表达式生成的合成子类。花括号包含了匿名子类的类结构体，可以为空。然而，如果 `new` 后面的名称指向的特质或类包含了抽象成员，则这些成员必须在定义匿名子类的类结构体的花括号内变为具体定义。

匿名函数（anonymous function）。函数字面量（function literal）的另一个叫法。

应用（apply）。你可以把方法、函数、或闭包应用于参数（argument），也就是说你对这些实参执行调用。

实参（argument）。调用函数的时候，实参被传递给函数的每个参数（parameter）。参数是指向实参的变量。实参是调用时传递的对象。另外，应用可以获得传递给单例对象（singleton object）`main` 方法的 `Array[String]` 类型的（命令行）实参。

赋值（assign）。你可以把对象赋值给变量。之后，变量将指向（refer to）对象。

辅助构造器（auxiliary constructor）。在类定义花括号内定义的附加构造器。看上去类似于名为 `this` 的方法定义，但没有结果类型。

块（block）。由花括号包围的一个或多个表达式和声明。运行块的时候，所有的表达式和声明会依次处理，之后块将把最后的表达式作为块自身的值返回。块通常被用作函数、`for` 表达式、`while` 循环的方法体，以及任何其他你想要把一些语句组合在一起的场合。较为正式的说法是，块是你只能从外部看到副作用和结果值的封装结构。因此定义类或对象的花括号并不能形成块，因为（定义在花括号之内的）字段和方法是从外部可见的。这种花括号形成的是模板。

绑定变量（bound variable）。表达式的绑定变量是指被定义和使用于表达式之内的变量。例如，在函数文本表达式`(x:Int)=>(x,y)`中，变量`x`和`y`都被使用，但仅有`x`被绑定，因为它是定义在表达式内的`Int`，并且是表达式所描述函数的唯一实参。

传名参数（by-name parameter）。是指参数类型的前部标记=>的参数，即，`(x: =>Int)`。相应于传名参数的实参不会在方法被调用之前做运算，而是在每次方法通过名称引用到参数的时候才执行。如果参数不是传名的，那它就是传值（by-value）的。

传值参数（by-value parameter）。是指参数类型的前部没有标记=>的形参，即，`(x: Int)`。相应于传值形参的实参在方法被调用之前做运算。传值形参是相对于传名形参的。

069

类（class）。使用`class`关键字定义，类既可以是抽象的也可以是具体的，还可以在实例化的时候用类型和值做参数化。在“`new Array[String](2)`”里，类被实例化为`Array`，产生的值类型是`Array[String]`。带类型参数（type parameter）的类被称为类型构造器（type constructor）。也可以说类型同样也具有类，好像类型`Array[String]`的类是`Array`。

闭包（closure）。能捕获自由变量（free variable），或者说是“关闭”函数创建时可见变量的函数对象（function object）。

伴生类（companion class）。与定义在相同源文件中的单例对象（singleton object）共享相同名称的类。这个类就是单例对象的伴生类。

伴生对象（companion object）。与定义在相同源文件中的类共享相同名称的单例对象。伴生对象与类可以互相访问私有成员。另外，定义于伴生对象的任何隐式转换都在任何用到了类的地方的作用域内。

逆变（contravariant）。可以通过把减号（-）放在类型参数之前，把逆变标注应用于类或特质的类型参数上。于是，类或特质的子类型将逆向——以相反的方向——协变于类型标注的参数。例如，`Function1`的第一个类型参数是逆变的，所以`Function1[Any, Any]`是`Function1[String, Any]`的子类型。

协变（covariant）。可以通过把加号（+）放在类型参数之前，把协变标注应用于类或特质的类型参数上。于是，类或特质的子类型将协变——以相同的方向——于类型标注的参数。例如，`List`的类型参数是协变的，所以`List[String]`是`List[Any]`的子类型。

柯里化（currying）。是一种编写带多个参数列表的函数的方式。例如`def f(x: Int)(y: Int)`是带有两个参数列表的柯里化函数。要应用柯里化的函数可以通过传递若干实参列表实现，如`f(3)(4)`。不过也可以编写柯里化函数的部分应用，如`f(3)`。

691

声明（declare）。声明抽象字段、方法或类型，是命名而不是实现实体。声明和定义（definition）的关键差别在于定义建立了命名实体的实现，但声明没有。

定义（define）。Scala 里定义某个东西是指给予它名称和实现。你可以定义类，特质，单例对象，字段

（field），方法（method），局部函数（local function），局部变量（local variable），等等。因为定义总是引入某种实现，所以抽象成员是被声明的而不是被定义的。

直接子类（direct subclass）。类是它的直接超类（direct superclass）的直接子类。

直接超类（direct superclass）。类或特质直接源于的类，是它的继承层级里在它之上的最近的类。如果类 `Child` 的可选扩展子句中提到了类 `Parent`，那么 `Parent` 就是 `Child` 的直接超类。如果 `Child` 的扩展子句提到了某个特质，那么特质的直接超类就是 `Child` 的直接超类。如果 `Child` 没有扩展子句，那么 `AnyRef` 就是 `Child` 的直接超类。如果类的直接超类带有类型参数，比如 `class Child extends Parent[String]`，那么 `Child` 的直接超类仍然是 `Parent`，不是 `Parent[String]`。另一方面，`Parent[String]` 将是 `Child` 的直接超类性（direct supertype）。参见超类型（supertype）以获得更多的关于类和类型之间差异的讨论。

相等性（equality）。在没有条件限制的情况下使用时，相等性是可以用‘==’表达的两个值之间的关系。参见引用相等性（reference equality）。

存在类型（existential type）。存在类型包含了对未知类型变量的引用。例如，`Array[T] forSome { type T }` 是存在类型。它是 `T` 的数组，这里 `T` 是某种完全未知的类型。所有对 `T` 的假设是它的确存在。这种假设很弱，但至少说明 `Array[T] forSome { type T }` 的确是数组，不是香蕉。

表达式（expression）。任何能产生结果的 Scala 代码，也可以说表达式运行能得到结果或产生值。

过滤器（filter）。是指 `for` 表达式中的 `if` 跟着一个布尔表达式。在 `for(i<-1 to 10;if i % 2==0)` 里，过滤器是“`if i % 2==0`”。

过滤表达式（filter expression）。过滤表达式是 `for` 表达式中跟在 `if` 后面的布尔表达式。在 `for(i<-1 to 10;if i % 2==0)` 里，过滤表达式是“`i % 2==0`”。

头等函数（first-class function）。Scala 支持头等函数，就是说你可以把函数表达为函数字面量语法，即，`(x:Int)=>x+1`，而且函数可以被表现为对象，这也被称为函数值（function value）。

for 推导式（for comprehension）。`for` 表达式的另一个叫法。

自由变量（free variable）。表达式的自由变量是指使用但不定义在表达式内的变量。例如，函数字面量表达式 `(x: Int) => (x, y)`，变量 `x` 和 `y` 都被使用到了，但只有 `y` 是自由变量，因为它没有定义在表达式内。

函数（function）。函数可以用实参列表调用并产生结果。函数具有参数列表、方法体，以及结果类型。函数作为类、特质或单例对象的成员时被称为方法（method）。定义在其他函数之内的函数被称为局部函数。结果类型为 Unit 的函数被称为过程（procedure）。源代码中的匿名函数被称为函数字面量（function literal）。运行时，函数文本被实例化为对象，称为函数值（function value）。

692

函数字面量（function literal）。Scala 源代码中没有名称的函数，用函数字面量语法加以特化。例如，`(x: Int,y: Int)=> x + y`。

函数值（function value）。可以像其他函数一样被调用的函数对象。函数值的类扩展了 scala 包的若干 `FunctionN` 特质（即 `Function0`，`Function1`）中的一个，并且通常是通过函数字面量语法的方式表达在源代码中的。当 `apply` 方法运行时，函数值被“调用”。能够捕获自由变量的函数值是闭包。

函数式风格（functional style）。函数式风格的编程强调函数和运行结果，而弱化操作发生的顺序。这种风格的特征是可以传递函数值给循环方法、不可变数据、无副作用的方法。其中主导语言包括 Haskell 和 Erlang，与指令式风格（imperative style）相对。

693

生成器（generator）。生成器在 `for` 表达式中定义了命名的 `val` 并把一系列的值赋给它。例如，在 `for(i<-1 to 10)` 中，生成器是“`i< -1 to 10`”。<–右侧的值是生成器表达式（generator expression）。

生成器表达式（generator expression）。生成器表达式在 `for` 表达式中产生一系列的值。例如，在 `for(i<-1 to 10)` 里，生成器表达式是“`1 to 10`”。

泛型类（generic class）。带类型参数的类。例如，因为 `scala.List` 带类型参数，所以 `scala.List` 是泛型类。

泛型特质（generic trait）。带类型参数的特质。例如，因为特质 `scala.collection.Set` 带类型参数，所以它是泛型特质。

助手函数（helper function）。以服务于一或多个临近其他函数为目的的函数。帮助函数通常实现为局部函数。

帮助方法（helper+ method）。作为类成员的帮助函数。通常是私有的。

不可变（immutable）。如果对象的值在对象创建之后不能以任何用户可见的方式改变，那么它就是不可变的。对象可以是，也可以不是不可变的。

指令式风格（imperative style）。编程的指令式风格强调对于操作精心的排列顺序以便于让它们的效果以正确的顺序发生。这种风格的特征为循环枚举，对数据适当的改变，以及带有副作用的方法。其中主导语言有 C，C++，C#和 Java，与函数式风格相对。

初始化（initialize）。一旦 Scala 源代码中定义了变量，你必须用对象对它执行初始化。

694

实例（instance）。实例，或类实例，就是对象，它是仅存在于运行时的概念。

实例化（instantiate）。类的实例化是指根据类创建新的对象，是仅发生于运行时的活动。

不变性/约束条件（invariant）。用在两个地方。它可以表示当数据结构良好的情况下始终保持为真的属性。例如，对于排序二叉树来说，每个节点如果存在右子节点，则在其右子节点之前完成排序，这是不变

的。有的时候它（invariant）也被用作非协变的同义词："在类型参数上 Array 类是不变的"[1]。

调用（invoke）。可以对实参调用方法，函数或闭包，即使用特定的实参执行方法体。

Java 虚拟机（JVM）。JVM 是 Java 虚拟机，或运行时，是运行 Scala 程序的宿主。

字面量（literal）。1，"one"，以及 (x:Int)=>x+1 都是字面量的例子。字面量是描述对象的简写，这个简写精确反映了被创建对象的结构。

局部函数（local function）。局部函数由 def 定义在块内部。相对地，如果 def 定义为类、特质或单例对象的成员，则被称为方法。

局部变量（local variable）。局部变量由 val 或 var 定义在块内部。尽管与局部变量相似，但函数的参数并不能被称为局部变量，而只是简称为参数或不带"局部"的"变量"。

成员（member）。成员是类、特质或单例对象模板的命名元素。成员可以通过所有者名称、点、它的简化名访问。比方说，定义在类中的顶层字段和方法是类的成员。定义在类里的特质是它包围类的成员。类里使用 type 关键字定义的类型是那个类的成员。类是定义于其中的包的成员。相对地，局部变量或局部函数不是围绕块的成员。

消息（message）。Actor 之间相互通信是通过相互发送消息实现的。发送消息不会中断接收者正在做的事。接收者可以一直等到结束当前活动并且它（对外）不变的部分被重新建立起来。

元编程（meta-programming）。元编程软件是指输入的本身也是软件的软件。编译器是元程序，也是像 scaladoc 这样的工具。使用元编程软件是为了能使用标注做一些事。

方法（method）。方法是某个类、特质或单例对象的成员函数。

混入（mixin）。混入是对被用于混入构成（mixin composition）的特质的称呼。换句话说，在"trait Hat"中，Hat 只是特质，但在"new Cat extends AnyRef with Hat"中，Hat 可以被称为混入。如果用在动词，"mix in"是两个单词。例如，你可以把特质混入到类或其他特质中。

混入构成（mixinconposition）。是把特质混入类或其他特质的过程。混入构成不同于传统的多继承，在于 super 所指的类型在定义特质时是未知的，直到每次特质被混入到类或其他特质时才被重新决定。

修饰符（modifier）。是以某种方式限制类、特质、字段或方法定义的关键字。例如，private 修饰符指明所定义的类、特质、字段或方法是私有的。

多重定义（multiple definitions）。同样的表达式能够以多重定义的方式赋值，如果你使用 val v1, v2, v3 = exp 这样的语法格式。

非协变（nonvariant）。类或特质的类型参数默认是非协变的。这样类或特质就不会在（类型）参数改变的情况下有子类型关系。例如，因为类 Array 对它的类型参数是非协变的，所以 Array[String]

[1]注：即 Array 类不与其类型参数发生协变。

既不是 Array[Any]的子类型，也不是它的超类型。

969

操作（operation）。Scala 里所有的操作都是方法调用。方法或许会以操作符标识的形式出现，如 b + 2，中的标识+。

参数（parameter）。函数可以带零个或多个参数。每个参数都有名称和类型。参数与实参之间的区别在于实参指的是调用函数时传递的实际对象，而参数则是指向这些传递的实参的变量[2]。

无参函数（parameterless function）。定义中没有空括号的、不带参数的函数。调用无参函数也可以不用提供括号。这是对统一访问原则的支持，从而可以在无须修改客户代码的情况下把 def 改变为 val。

无参方法（parameterless method）。无参方法是类、特质或单例对象的无参函数成员。

参数化字段（parametric field）。定义为类参数的字段。

偏函数（partially applied function）。用于表达式的、缺少了某些实参的函数。例如，如果函数 f 的类型为 Int=>Int=>Int，那么 f 和 f(1)是偏函数。

路径依赖类型（path-dependent type）。类似于 swiss.cow.Food 的类型。swiss.cow 部分是组成对象引用的路径。类型的含义对用来访问的路径敏感。比如说，swiss.cow.Food 和 fish.Food 是不同的类型。

模式（pattern）。match 表达式的可选部分中，模式是跟在 case 关键字之后，处于模式守卫（pattern guard）或=>符号之前的部分。

模式守卫（pattern guard）。match 表达式的可选部分中，可以在模式后加上模式守卫。例如，在“case x if x % 2==0=> x + 1”中，模式守卫是“if x % 2==0”。带有模式守卫的样本仅在模式匹配成功且模式守卫返回值为真的情况下才会被选中。

论断（predicate）。论断是指结果类型为 Boolean 的函数。

主构造器（primary constructor）。类的主要构造器，将调用超类的构造器，并在需要的情况下，初始化字段为传入值，以及执行所有定义在类的花括号内的顶层代码。字段仅由不被传递给超类构造器的值参数做初始化（但任何类结构体内不曾使用过的字段不会初始化并有可能因而被优化掉）。

697

过程（procedure）。过程是结果类型为 Unit 的函数，因此执行的目的仅为了得到它的副作用。

可重新赋值的（reassignable）。变量可以是也可以不是可重新赋值的。var 是可重新赋值的，val 不是。

递归（recursive）。递归函数是调用自身的函数。如果函数中唯一调用它自身的地方是函数的最后一个

[2]译注：虽然 parameter 对应于 argument 应分别译作形参和实参，但 parameter 一词用在很多地方，如：parameter type，by-name parameter，type parameter 等，若译为形参，则其他词应译为：形参类型，传名形参，类型形参，颇有些教条。所以就译为参数，而 argument 译为实参作对照。如果这两个词处于实际没有差别的语境中，统一作参数处理。

表达式，那么函数就是尾递归的（tail recursive）。

引用（reference）。引用是指针的 Java 抽象，可以唯一识别保存于 JVM 堆中的对象。引用类型变量保留了到对象的引用，因为引用类型（`AnyRef` 的实例）实现为保存在 JVM 堆的 Java 对象。相反，值类型变量可能有时保留的是引用（对于装箱类型来说），而有时不是（如果对象被表达为原始值）。广泛意义上说，Scala 变量指向（refer to）的是对象。术语“指向”比“保留引用”更为抽象。如果类型为 `scala.Int` 的变量当前代表的是 Java 原始的 `int` 值，那么那个变量仍然指向 `Int` 对象，但不涉及引用。

引用相等性（reference equality）。引用相等性是指两个引用识别的是同一个对象。引用相等性仅对于引用类型有意义，可以通过调用 `AnyRef` 的 `eq` 得到。（Java 程序中，引用相等性可以在 Java 引用类型上使用==获得。）

引用类型（reference type）。引用类型是 `AnyRef` 的子类。运行时，其实例始终保存于 JVM 的堆中。

指称透明性（referential transparency）。是指独立于临时语境并且没有副作用的函数属性，对于特定的输入，指称透明函数的调用可以用它的结果替换而不会改变程序语义。

指向（refer）。在运行中的 Scala 程序里的变量始终指向某个对象。即使变量被赋值为 `null`，概念上它也是指向 `Null` 对象的。运行时，对象可能实现为 Java 对象或原始类型值，但 Scala 能够让程序员在更高的抽象层次上想象代码以他们设想的方式运行。参见引用（reference）。

结果（result）。Scala 程序中的表达式会产生结果。Scala 中所有表达式的结果是对象。

结果类型（result type）。方法的结果类型是指通过调用方法产生的值的类型。（Java 中，这个概念被称为返回类型。）

返回（return）。Scala 程序的函数可以返回值。你可以把这个值称为函数的结果，也可以说函数产生了值。所有 Scala 函数的结果都是对象。

运行时（runtime）。指 Java 虚拟机或 JVM，即正在运行的 Scala 程序的宿主。运行时的概念包含了由 Java 虚拟机规格说明（Java Virtual Machine Specification）定义的虚拟机，Java API 的运行时函数库和标准 Scala API。运行时阶段（run 和 time 之间加空格），是指程序正在运行之中，与编译时相对。

运行期类型（runtime type）。是指运行期的对象类型。相对的，静态类型（static type）是表达式在编译期的类型。多数运行期类型都只是没有类型参数的裸类。例如，“`Hi`”的运行时类型是 `String`，而 `(x: Int) => x + 1` 的运行时类型是 `Function1`。运行时类型可以通过 `isInstanceOf` 测试。

脚本（script）。包含顶层定义和语句，可以直接用 `scala` 运行而无须执行编译过程的文件。脚本必须以表达式，而不是定义结束。

选择器（selector）。match 表达式中用来做匹配的值。例如，“`smatch{ case_=> }`”中，选择器是 `s`。

自身类型（self type）。特质的自身类型是用于特质中的方法接收者，`this` 的假设性类型，任何混入特

质的具体类都必须确保它的类型符合特质的自身类型。自身类型最通常的应用就是像第 27 章描述的那样为了把大类分成若干特质。

半结构化数据（semi-structured data）。XML 数据是半结构化的。它比平展的二进制文件或文本文件更为结构化，但它没有编程语言的数据结构那样的完整结构。

序列化（serialization）。对象可以被序列化为字节流，以便保存成文件或通过网络传递。之后可以甚至在另一台电脑上反序列化字节流，并获得与原来序列化的对象一致的（另一个）对象。

遮蔽（shadow）。局部变量的新声明会遮蔽处于外围作用域的同名变量。

签名（signature）。签名是类型签名（type signature）的简写。

单例对象（singleton object）。用 `object` 关键字定义的对象。每个单例对象都有且只有一个实例。若单例对象与类共享名称并定义在相同源文件中，则它被称为是类的伴生对象。类是它的伴生类。没有伴生类的单例对象是独立对象（standalone object）。

独立对象（standalone object）。没有伴生类的单例对象。

语句（statement）。表达式、定义或引用等，可以写进 Scala 源代码的模板或块的东西。

静态类型（static type）。参见类型。

子类（subclass）。类对于它所有的超类和超特质来说，是子类。

子特质（subtrait）。特质对于它的所有超特质来说，是子特质。

子类型（subtype）。Scala 编译器允许任何类型的子类型可以在需要该类型的时候被用作替代类型。对于不带类型参数的类和特质，子类型关系仿照子类关系。例如，如果类 `Cat` 是抽象类 `Animal` 的子类，并且它们都不带参数类型，那么 `Cat` 类型就是 `Animal` 类型的子类型。与之类似，如果特质 `Apple` 是特质 `Fruit` 的子特质，并且它们都不带参数类型，那么类型 `Apple` 就是类型 `Fruit` 的子类型。然而对于带类型参数的类和特质，变型开始发挥作用了。例如，因为抽象类 `List` 被声明为与它唯一的类型参数协变（即 `List` 被声明为 `List[+A]`），所以 `List[Cat]` 是 `List[Animal]` 的子类型，`List[Apple]` 是 `List[Fruit]` 的子类型。尽管这些类型的类都是 `List`，这些子类型关系也依然存在。相反，因为 `Set` 并没有声明为与它的类型参数协变（即 `Set` 被声明为 `Set[A]`，没有加号），所以 `Set[Cat]` 不是 `Set[Animal]` 的子类型。子类型应该正确地实现超类型的契约，以便能够应用里氏代换原则（Liskov Substitution Principle），但编译器仅在类型检查的级别核实这一属性。

超类（superclass）。类的超类包含它的直接超类，它直接超类的直接超类，依此类推，直到 `Any`。

超特质（supertrait）。类或特质的超特质，如果存在，则包含所要直接混入类或特质或任何超类的特质，以及任何这些特质的超特质。

超类型（supertype）。类型是所有其子类型的超类型。

合成类（synthetic class）。合成类是由编译器自动产生的，而不是由程序员手工编写的。

尾递归（tail recursive）。尾递归函数是指仅在函数的最后一个操作中调用了函数自身的函数。

目标类型化（targettyping）。目标类型化是一种参考所需类型进行类型推论的形式。例如，在`nums.filter((x) => x > 0)`中，Scala 编译器会推论 x 的类型为 nums 的元素类型，因为 `filter` 方法是对每个 nums 的元素调用的函数。

模板（template）。模板是类、特质或单例对象定义的结构体，它定义了类型签名、行为及类、特质或对象的初始化状态。

特质（trait）。特质，由 `trait` 关键字定义，类似于不能带任何值参数的抽象类，并且可以通过被称为是混入构成的过程被“混合进”类或其他特质。特质可以被一或多个类型参数化。在被类型参数化的时候，特质构造了类型。例如，`Set` 是带有单个类型参数的特质，然而 `Set[Int]`是类型。同样，`Set` 被说成是 `Set[Int]` “的特质”。

类型（type）。Scala 程序中的每个变量和表达式都有在编译时就知道的类型。类型限制了运行时变量可以指向的，或者表达式可以产生的值的可能性。若有必要的话，变量或表达式的类型也可以被称为是静态类型以区分于对象的运行时类型。换句话说，“类型”本身是指静态类型。类型之所以不同于类，是因为带有类型参数的类可以构造许多类型。例如，`List` 是类，但不是类型。`List[T]`是带有自由类型参数的类型。`List[Int]`和 `List[String]`也是类型（称为实类型（ground type）因为它们没有自由类型参数）。类型可以有“类”或“特质”。例如，类型 `List[Int]`的类是 `List`。类型 `Set[String]`的特质是 `Set`。

类型约束（type constraint）。某些标注是类型约束，即它们对类型可以包含的值添加了额外的限制或约束。例如，`@positive` 是对于类型 Int 的类型约束，限制 32 位的整数类型只能包含正数。类型约束不会受到标准 Scala 编译器的检查，但附加工具或编译器插件必须实现这种检查。

类型构造器（type constructor）。带类型参数的类或特质。

类型参数（type parameter）。必须填入类型的泛型类或泛型方法的参数。例如，`List` 类被定义为“`class List[T]{...`”，还有 `Predef` 对象的方法 `identity`，被定义为“`defidentity[T] (x:T)=x`”。这两种情况下的 `T` 都是类型参数。

类型签名（type signature）。方法的类型签名包含了它的名称、参数，如果有的话，数量、次序、类型，以及它的结果类型。类、特质或单例对象的类型签名包含了它的名称，所有成员和构造器的类型签名，以及它的继承和混入关系声明。

统一访问原则（uniform access principle）。统一访问原则规定，变量和无参函数应该可以用相同的语法形式访问。Scala 以不让括号出现在无参函数调用场合的方式支持这一原则。结果，无参函数定义可以在不影响客户代码的情况下被改变为 `val`，反之亦然。

不可到达的（unreachable）。在 Scala 的层次上，对象可以变成不可到达的，这时它们占据的内存可能

会被运行时收回。不可到达并非一定意味着未引用。引用类型（AnyRef 的实例）被实现为保存在 JVM 堆中的对象。一旦引用类型的实例变成不可到达的，那么它的确会变成未引用的，并且可以被垃圾回收。值类型（AnyVal 的实例）被实现为原始类型值及 Java 包装类型的实例（诸如 java.lang.Integer），存在于堆中。值类型实例可以在指向它们的变量的完整生命周期内被装箱（从原始值转变为包装对象）及拆箱（从包装对象转变为原始值）。如果当前代表了 JVM 堆中的包装对象的值类型实例变得不可到达了，它的确会变成未引用的，并且可以被垃圾回收。但如果当前代表原始值的值类型变得不可到达了，那么它不会变成未引用，因为在这个时候它不是对象，不存在于 JVM 的堆中。运行时会收回由不可到达对象占据的内存，但比方说如果 Int 在运行时被实现为原始的 Java 的 Int 值，占据了执行方法堆栈帧的一些内存，那么只有在方法完成之后堆栈帧被弹出才能“收回”对象的内存。引用类型的内存，如 String，会在它们变得不可到达之后由 JVM 的垃圾回收器收回。

未引用的（unreferenced）。参见不可到达（unreachable）。

703

值（value）。任何计算或表达式的结果在 Scala 里都是值，并且 Scala 里的所有值都是对象。值这个术语在本质上是指对象在内存（在 JVM 的堆或堆栈）中的镜像。

值类型（value type）。值类型是指 AnyVal 的任意子类，如 Int、Double 或 Unit。这个术语在 Scala 源代码级别有意义。在运行时，相应于 Java 原始类型的值类型的实例可以利用原始类型值或包装类型实例实现，如 java.lang.Integer。在值类型实例的整个生命周期中，运行时可能会把它们在原始类型和包装类型之间前后转换（即对它们执行装箱和拆箱）。

变量（variable）。指向对象的具名实体。变量要么是 val，要么是 var。它们都必须在定义的时候被初始化，但只有 var 可以在之后重新赋值以指向不同的对象。

变型（variance）。类或特质的类型参数可以用变型标注（协变，+；或者逆变，-）做标记。这些变型标注指示了如何进行泛型类或特质的子类型化操作。例如，泛型类 List 是与它的类型参数协变的，因此 List[String]是 List[Any]的子类型。默认情况下，也就是说没有+或-标注的时候，类型参数是非协变的。

704

生成（yield）。表达式可以生成结果。yield 关键字标示了 for 表达式的结果。

参考文献

[Abe96] Abelson, Harold and Gerald Jay Sussman. Structure and Interpretation of Computer Programs. The MIT Press, second edition, 1996.

[Aho86] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[Blo08] Bloch, Joshua. Effective Java Second Edition. Addison-Wesley, 2008.

[DeR75] DeRemer, Frank and Hans Kron. "Programming-in-the large versus programming-in-the-small." In Proceedings of the international conference on Reliable software, pages 114–121. ACM, New York, NY, USA, 1975. doi:*http://doi.acm.org/10.1145/800027.808431*.

[Dij70] Dijkstra, Edsger W. "Notes on Structured Programming.", April 1970. Circulated privately. Available at *http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF* as EWD249 (accessed June 6, 2008).

[Eck98] Eckel, Bruce. Thinking in Java. Prentice Hall, 1998.

[Emi07] Emir, Burak, Martin Odersky, and John Williams. "Matching Objects With Patterns." In Proc. ECOOP, Springer LNCS, pages 273–295. July 2007.

[Eva03] Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

[Fow04] Fowler, Martin. "Inversion of Control Containers and the Dependency Injection pattern." January 2004. Available on the web at *http://martinfowler.com/articles/injection.html* (accesssed August 6, 2008).

[Gam95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Goe06] Goetz, Brian, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Homes, and Doug Lea. Java Concurrency in Practice. Addison Wesley, 2006.

[Jav] The Java Tutorials: Creating a GUI with JFC/Swing. Available on the web at *http://java.sun.com/docs/books/tutorial/uiswing*.

[Kay96] Kay, Alan C. "The Early History of Smalltalk." In History of programming languages—II, pages 511–598. ACM, New York, NY, USA, 1996. ISBN 0-201-89502-1. doi:*http://doi.acm.org/10.1145/234286.1057828*.

[Kay03] Kay, Alan C. An email to Stefan Ram on the meaning of the term "object-oriented programming", July 2003. The email is published on the web at *http://www.purl.org/stefan_ram/pub/doc_kay_oop_en (accesssed June 6, 2008)*.

[Lan66] Landin, Peter J. "The Next 700 Programming Languages." Communications of the ACM, 9(3):157–166, 1966.

[Mey91] Meyers, Scott. Effective C++. Addison-Wesley, 1991.

[Mey00] Meyer, Bertrand. Object-Oriented Software Construction. Prentice Hall, 2000.

[Ode03] Odersky, Martin, Vincent Cremet, Christine Röckl, and Matthias Zenger. "A Nominal Theory of Objects with Dependent Types." In Proc. ECOOP'03, Springer LNCS, pages 201–225. July 2003.

[Ode05] Odersky, Martin and Matthias Zenger. "Scalable Component Abstractions." In Proceedings of OOPSLA, pages 41–58. October 2005.

[Ode08] Odersky, Martin. The Scala Language Specification, Version 2.7. EPFL, February 2008. *Http://www.scalalang.org/docu/manuals.html*.

[Ray99] Raymond, Eric. The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, 1999.

[Rum04] Rumbaugh, James, Ivar Jacobson, and Grady Booch. The Unified Modeling Language Reference Manual (2nd Edition). Addison-Wesley, 2004.

[SPJ02] Simon Peyton Jones, et.al. "Haskell 98 Language and Libraries, Revised Report." Technical report, *http://www.haskell.org/onlinereport*, 2002.

[Ste99] Steele, Jr., Guy L. "Growing a Language." Higher-Order and Symbolic Computation, 12:221–223, 1999. Transcript of a talk given at OOPSLA 1998.

[Vaz07] Vaziri, Mandana, Frank Tip, Stephen Fink, and Julian Dolby. "Declarative Object Identity Using Relation Types." In Proc. ECOOP 2007, pages 54–78. 2007.

关于作者

Martin Odersky 是 Scala 语言的创造者。作为瑞士洛桑联邦理工学院（EPFL）的教授，他主要从事编程语言领域的工作。更具体地说，是面向对象和函数式编程的语言。他研究的论题是，这两种编程模式是硬币的两面，应该被尽可能地统一在一起。为了证明这点，他已试验性地设计了大量的语言，从 Pizza 到 GJ 到 Functional Nets 语言[1]。他还作为 Java 泛型的联合设计师及当前 `javac` 参考编译器的原作者影响了 Java 的发展。从 2001 年起，他主要从事 Scala 编程语言的设计、实现及改进工作。

Lex Spoon 是 Google 的软件工程师。他以 EPFL 博士后身份在 Scala 方面工作了两年时间，从佐治亚理工学院（Georgia Tech）获得计算机科学的博士学位。那时他的主要工作是动态语言的静态分析。除了 Scala 之外，他还从事大量其他的编程语言工作，范围从动态语言 Smalltalk 到科学性语言 X10。他和他的妻子，两只猫、一条吉娃娃还有一只乌龟，现生活于亚特兰大。

Bill Venners 是 Artima 的总裁，兼 Artima 开发者网站（*www.artima.com*）的发行人。他是《深入 Java 虚拟机》（"Inside the Java Virtual Machine"）的作者，该书是定向为程序员的 Java 平台架构和内部组织的总体研究。他在 JavaWorld 杂志上有很受欢迎的专栏，内容涵盖 Java 内部机制，面向对象设计，还有 Jini。Bill 从 Jini 诞生伊始就活跃于 Jini 社区，他曾领导 Jini 社区的 ServiceUI 项目，而其中的 ServiceUI API 已经变成了联系用户界面和 Jini 服务之间的事实标准。Bill 还是 ScalaTest（Scala 和 Java 开发的开源测试工具）的首席开发者（lead developer）和设计者。

[1]译注：Petri-Nets 语言的研究性衍生版本。

索引

符号

B

D

E

F

G

H

I

J

M

N

O

P

Q

R

T

W

X

Y

Z

北京博文视点（www.broadview.com.cn）资讯有限公司成立于2003年，是工业和信息化部直属的中央一级科技与教育出版社——电子工业出版社（PHEI）下属旗舰级子公司，在六年的开拓、探索和成长中，已成为中国颇具影响力的专业IT图书策划和服务提供商。

六年来，博文视点以开发IT类图书选题为主业，励精图治、兢兢业业，打造了一支团结一心的专业队伍，并形成了自身独特的竞争优势。一直以来，博文视点始终以传播完美知识为己任，用诚挚之心奉献精品佳作，年组织策划图书达300个品种，同时开展相关信息和知识增值服务，赢得了众多才华横溢的作者朋友和肝胆相照的合伙伙伴，已经成为IT图书领域的高端品牌。

我们的理念： 创新专业图书服务体制；培养职业策划图书服务队伍；打造精品图书品牌；完善全面出版服务平台。

我们的目标： 面向IT专业人员的出版物提供相关服务。

我们的团队： 一个整合了专业技术人员和专业服务人员的团队；一个充满创新意识和创作激情的团队；一个不断进取、追求卓越的团队。

我们的服务： 善待作者　尊重作者　提升作者

我们的实力： 优秀的专业编辑队伍
全方位立体化的强大的市场推广平台
实力雄厚的电子工业出版社的渠道平台

“走出软件作坊独辟蹊经　人道编程之美，
追踪加密解密庖丁解牛　精雕夜读天书。”

路漫漫其修远，博文视点愿与所有曾经帮助、关心过我们的朋友、作者、合作伙伴携手奋斗。未来之路，不可限量！

地址： 北京市万寿路173信箱电子工业出版社博文视点资讯有限公司
邮编： 100036　**总机：** 010-88254356　**传真：** 010-88254356-802

武汉分部地址： 武汉市洪山区吴家湾湖北信息产业科技大厦1402室
邮编： 430074　**总机：** 027-87690813　**传真：** 027-87690013

欢迎投稿： bvtougao@gmail.com
读者邮箱： reader@broadview.com.cn

博文视点官方博客： http://blog.csdn.net/bvbook
博文视点官方网站： http://www.broadview.com.cn

反侵权盗版声明

Scala编程

Scala语言权威著作，Scala是Java平台的新语言，它将面向对象与函数式编程概念结合在一起，形成了独特的并且强大的开发工具。

如果今天我要选用Java之外的一门语言，我将会选择Scala。

——James Gosling，Java缔造者

由Scala语言的设计者共同执笔，这本权威书籍将逐步引导你学习Scala语言及其背后的思想。

本书经过精心编排以方便读者学习。开始的几章会有足够的基本知识做铺垫，让你能够立即上手用Scala实现简单的任务。全书经过组织，使每个新概念都建立于先前的概念之上——通过一系列的步骤确保帮助你掌握Scala语言及其所体现的重要编程理念。

作为Scala的综合教程和参考，本书覆盖了语言的全部内容及重要的库。

你将学到如何：

- 用Scala编写脚本和系统
- 从Scala调用Java标准库
- 使用Scala的面向对象特性
- 以函数式风格编程
- 使用Scala的集合库
- 使用特质和混入做设计
- 设计内嵌领域特定语言（DSL）
- 受益于模式匹配
- 利用函数和闭包
- 创建新的控制抽象
- 通过单元测试确保质量
- 用actor编写并行程序

Martin Odersky Scala语言的创始者。他是瑞士洛桑联邦理工学院（EPFL）教授，从2001年起他就带领小组致力于Scala语言、标准库和编译器的开发。他还是Java泛型的合作设计者及当前javac参考编译器的原作者。**Lex Spoon** 在EPFL为Scala工作了两年，现为Google软件工程师。**Bill Venners** Artima的总裁，撰写了许多关于Java的文章，是《深入Java虚拟机》的作者，以及ScalaTest测试框架的设计者。

www.artima.com
artima

图书分类 程序设计/编程语言

ISBN 978-7-121-12119-7

9 787121 121197 >

定价：89.00元

策划编辑：卢鹞翔
责任编辑：杨绣国