综合进阶向导

Scala 编程

Martin Odersky
Lex Spoon
Bill Venners

artima

Scala 编程

Scala 编程

Martin Odersky, Lex Spoon, Bill Venners

artima

ARTIMA 印刷公司

芒廷维尤（Google 所在地），加利福尼亚

Scala 编程

第一发行版，第六版本

Martin Odersky 是 Scala 语言的缔造者同时也是瑞士洛桑 EPFL（洛桑联邦理工大学）的教授。Lex Spoon 做为博士后与 Martin Odersky 一起在 Scala 方面工作了 2 年。Bill Venners 是 Artima 公司的总裁。

Artima Press 是 Artima 公司拥有的商标。

加利福尼亚 94039，芒廷维尤市，390122 信箱

版权©2007，2008 Martin Odersky，Lex Spoon，Bill Venners。

版权所有。

第一版以预印刷版形式发布于 2007 年

第一版发布于 2008 年

美国制造

12 11 10 09 08　5 6 7 8 9

ISBN-10: 0-9815316-1-X

ISBN-13: 978-0-9815316-1-8

本书的任何部分未经 Artima 公司的书面授权不得以商业的或非商业的任何形式复制，修改，分发，储存于检索系统，再版，展览或表演。

本书中的所有信息仅以“依原件”形式提供，并不负有任何形式的保证责任。

专有名词“Artima”和 Artima 的标志图案是 Artima 公司注册的商业标志。所有其他公司和/或产品名称均为它们拥有者所注册的商标。

to Nastaran - M.O.

to Fay - L.S.

to Siew - B.V.

概要

内容

图释

表格

代码

前导

Martin Odersky 用他定义的匹萨语言给了 Java 世界一个很大的冲击。尽管匹萨本身没有流行过，但它展现了当把面向对象和函数型语言两种风格，技术地且很有品地混搭在一起时，就形成了一个自然和强有力的组合。匹萨的设计成为了 Java 泛型的基础，马丁的 GJ（Generic Java）编译器从 Java 1.3 开始成为了 Sun 微系统的标准编译器（尽管关闭了泛型）。我有幸能够维护这个编译器多年，因此我能通过第一手经验从语言设计到语言的实现方面（向大家）报告马丁的技术。

那时候我们还在 Sun 公司，尝试用一些零打碎敲的特定问题解决方案来扩展语言，如 for-each 循环，枚举，自动装包，去简化程序开发的时候，马丁则继续着他在更强大的正交语言原语方面的工作以帮助程序员用库来提供解决方案。

后来，静态类型语言受到了冲击。Java 的经验说明了静态语言编程会导致大量的固定写法的代码。通常认为我们应该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。这种认知被马丁最近的作品，Scala，的出现打破。

Scala 是一种很有**品味**的类型语言：它是静态类型的，但仅需在必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，然后用一些新奇的点子把它们优美地整合成一体。它的语法是如此的轻量级，而原语又如此富有表达力，以至于根本可以认为 API 的使用不须负担语法开销。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。从这点上看，Scala 是一种支持**内嵌的领域特化：*embedded domain-specific*** 的语言。

Scala 会成为下一个伟大的语言吗？时间可以说明一切。Martin Odersky 的小组绝对有这样的品味和技术做到这一点。有一件事可以确信：Scala 语言设定了一个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

许多人持续地关注本书及其相关的资料。在这里表示感谢。

Scala 语言本身已经成为了许多人努力的集合。版本 1.0 的设计和实现受到了 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努力。

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，从而使得语言的设计成型。Scala 语言电邮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和它的工具。

George Berger 付出了极大努力于建造过程和本书流畅的 Web 体验。令人欣慰的结果就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为助益的审校：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，and Jonathan Andrew Wolter。我们还要感谢 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。

我们要提出特别感谢给我们所有给我们建设性评价的读者。你们的评价非常有助于我们把本书做得更好。我们没办法印出所有提供了评论的名字，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，Peter Moore，Randolph Kahle，Vladimir Kelman，Daniel Gronau，Dirk Detering，Hiroaki Nakamura，Ole Hougaard，Bhaskar Maddala，David Bernard，Derek Mahar，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。

介绍

本书是一份 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。如果你想要用 Scala 做你的下一个软件项目，那么本书是为你准备的。而且，本书希望能使那些希望拓展视界的程序员们通过学习一些新概念而获得趣味。打个比方，如果你是一位 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，还有它隐含的理念，通常都能帮你成为一个更好的程序员。

本书假设你已经有了通常的编程知识。当然 Scala 也可以很好的做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，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，但本书并不预设你知道任何关于 Java 的事情。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

因为本书的主要目的是作为教材，所以推荐的阅读方式就是按照章节的次序，从头到尾。我们尽力一次介绍一个话题，并且仅以介绍过话题来说明新的话题。因此，如果你跳到后面去先睹为快，你可能会发现有些东西是用你不太明白的概念解释的。如果按照章节的顺序阅读，我们认为你将发现一步一个脚印的方式将引导你顺利地获得 Scala 开发的能力。

如果你发现一个你不懂的术语，请一定查找一下术语表和索引。许多读者会简单略过书中的某些部分，这也可以。术语表和索引有助于你返回到你略过的某些东西。

在你读完一遍之后，本书还可以做为一份语言参考书。Scala 语言有一个正式的定义，但是语言的定义是以可读性为代价要求精确性的文档。尽管本书并未涵盖 Scala 的所有细节，但它在你更好地掌控 Scala 编程之前，作为一本平易近人的语言参考书已足够全面。

如何学习 Scala

简单地通读本书，你将学到 Scala 的许多东西。但如果再只多做很少的事情，你将更快更全面地了解 Scala。

首先，你可以好好地利用本书中包括的许多编程例子。尝试自己输入是一个强迫你的大脑思考每一行代码的方式。尝试各种各样的变化是让它们变得更有趣也是让你确信你已真正明白它们如何工作的方法。

第二点，与多个在线论坛保持联系。采用这种方式，你和其他 Scala 痴迷者能够互相帮助。还有许多的电邮列表，讨论论坛，和聊天室，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，一旦你已经读得够多了，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。仅仅看书只能走到这么远。

电子书的特点

本书有纸面和 PDF 电子书两种形式。电子书并不仅仅是纸版书的可打印版本。虽然其内容与纸版书没有差别，但电子书已经（被）为在电脑屏幕上阅读做了仔细的设计和优化。

第一件要提的事就是书里面的大多数参考是超链接的。如果你选择一个到某个章节、图片、或者术语表的参考，你的 PDF 阅览器将立刻带你到选中的条目，从而避免你为了找到它翻遍全书。

另外，每页的底部有许多导引链接。“封面”，“概要”和“内容”链接将带你到本书的主要入口。“术语表”和“索引”链接将带你到本书的参考部分。最后，“讨论”链接将带你到在线论坛和其他读者、作者以及更大的 Scala 社区讨论问题。若你发现了一处印刷，或者什么东西你认为能够解释得更好的地方，请点击“建议”链接，带你到在线 Web 应用，并反馈给作者。

电子书里的页面除了空白页面被移除，剩余的页面重新排列编码外和打印出来的书没什么差别。页面的计数不同，这样当你想打印电子书的某些部分时，可以很容易地决定 PDF 页面的号码。所以，电子书的每一页的页码都和你在 PDF 阅读器中看到的页码一样。

印刷体变化

首次使用的**术语：** ***term***，将被倾斜显示（加粗）。小代码例子，如 `x+1`，将用等宽字体显示在文档段落中。大段的代码例子将放在等宽字体的段落中显示：

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

在显示交互式 shell 的时候，shell 的回应将显示为较亮的字体。

```
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 的第一步”，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 的下一步”，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，它们的文本，你可以执行的操作，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）分数作为例子。
- 第 7 章 “内建控制结构”，显示了如何使用 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”，显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 章之中那样基础，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），显示了 Scala 在混入组成（mixin composition）中的机制。本章显示了特质如何工作，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”，显示了 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 “case 类和模式匹配”，介绍了 case 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”，显示了如何使用基础的 Scala 集合类型，如：列表，数组，元组（tuple），集以及映射表。
- 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，用来显示一些有状态对象的活动。
- 第 19 章，“类型参数化”，用一个具体的例子：纯函数队列类的设计，解释了第十三章之中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，“抽象成员和属性”，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 里面列表是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，“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，“标注”，显示了如何通过标注使用语言的扩展部分。本章示范了若干标准标注，也示范了如何建立你自己的标注。
- 第 26 章，“使用 XML”，显示了在 Scala 中如何处理 XML。包括创建 XML，拆解，以及拆解之后的处理等一系列惯用方式。
- 第 27 章，“对象用作模块”，显示了为什么说 Scala 的对象已足够丰富，从而消除了分离式模块系统的使用需求。
- 第 28 章，“对象等价”，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，“捆绑使用 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，及建议的解决方法。
- 第 30 章 “行动类和并发”，展示如何使用 Scala 的行动类：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，但行动类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，“组合子解析”，显示了如何使用 Scala 的解析器组合子库来创建解析器。
- 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 “SCell 电子表”，通过展示一个完整的电子表的实现，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。与本书其他读者交互，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，这样我们可以确信在本书将来的印刷或发行版中修正它。

Scala 编程

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

第1章

可伸展的语言

Scala语言的名称来自于“可伸展的语言”。之所以这样命名，是因为他被设计成随着使用者的需求而成长。你可以把Scala应用在很大范围的编程任务上，从写个小脚本到建立个大系统。[1]

Scala 是很容易进入的语言。它跑在标准的 Java 平台上，可以与所有的 Java 库实现无缝交互。它也是用来编写脚本把 Java 控件链在一起的很好的语言。但是用它来建立大系统和可重用控件的架构将更能够发挥它的力量。

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到静态类型语言中的混血儿。Scala 的许多不同的方面都展现了面向对象和函数式编程的熔合；或许它比其他那些广泛使用的语言更有渗透性。在可伸展性方面，这两种编程风格具有互补的力量。Scala的函数式编程使得它便于快速地从简单的碎片开始建立一些有趣的东西。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于新的需求。Scala 中这两种风格的组合使得它有可能表达新的编程模式和控件抽象。并产生了易读、简洁的编程风格。由于它良好的延展性，用 Scala 编程将会有很多的乐趣。

本章作为最初的章节，回答了“为什么使用Scala？”的问题，给出了Scala设计高层级的视图和它背后的原因。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有一个基础认识。尽管本书是一部Scala教程，但本章不在其内。如果你急于开始写一些Scala代码，你可以跳到 第 2 章。

1.1 与你一同成长的语言

不同尺寸的程序倾向于需要不同的编程结构。举例来说，考虑以下的 Scala 程序：

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

这段程序建立了一个国家和它们的首都之间的映射表，增加了一个新的绑定("Japan"->"Tokyo")，然后打印了与法国相关的首都。[2]本例中的声明都是高层次的，也就是说，没有被外加的分号或者类型注释弄得乱糟糟的。实际上，这种感觉就好像那种现代的“脚本化”语言，比如，Perl，Python或者Ruby。这些语言的一个普遍特征，与上例有关的，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，因为它能让程序易读和清晰。然而，有些时候你或许不赞成它们的这种“均码”哲学，因为你需要用一种更加细粒度地去控制在你程序中用到的映射的属性。Scala 可以在你需要的时候提供这种细粒度的控制，因为映射在 Scala 里并不是语法特性。

---

[1] Scala 的发音是 *skah-la*。

[2] 若你不能明白程序中的所有细节也请忍耐。在下两个章节中将解释它们。

它们是库抽象，你可以扩展或者改造。

在上面的程序里，你将获得一个缺省的 Map 实现，不过你也可以很轻松地改变它。比方说，你可以定义个特别的实现，如 HashMap 或 TreeMap，或者你可以特定这个映射必须是线程安全的，**混入：*mix-in*** 个 SynchronizedMap **特色：*trait***。你还可以给映射特定一个缺省值，或你可以重载你创建的映射的任意方法。每个例子里，你都可以如上例所示那样使用同样简单的映射访问语法。

这个例子显示了 Scala 带给你的方便性和灵活性。Scala 有一整套的方便构件来帮助你快速启动及让你用一种愉悦清晰的状态编程。与此同时，你有信心你不会让语言过度发育。你总可以把程序按你的需要裁剪，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。

培育新的类型

Eric Raymond把大教堂和杂货铺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。一旦建成了，就长时间保持不变。相对来说，杂货铺则天天在被工作其中的人调整和扩展。Raymond的文章中，杂货铺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更像一个杂货铺而不是大教堂，因为它被设计为让用它编程的人扩展和修改的。Scala并没有提供所有你在一种“完美齐全”语言中可能需要的东西，而是把制作这些东西的工具放在了你的手中。

这儿有个例子。许多程序需要一个能够变得任意大都不会溢出或者由于数学操作而“绕回”的整数类型。Scala在库类Scala.BigInt中定义了这样一个类型。这里有一个使用了那个类型的方法定义，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，如果你调用了 factorial(30)，你将得到：

```
265252859812191058636308480000000
```

BigInt看上去就像一个内建的类型，因为你可以使用整数值和这种类型值的操作符如*和-。然而它只是凑巧定义在Scala标准库中的类。[6]如果这个类缺失了，可以直接由任意的Scala程序员写一个实现出来，举例来说，通过包装Java的类java.math.BigInteger（实际上，Scala的BigInt就是这么实现的）。

当然，你也可以直接使用 Java 的类库。但结果却不尽乐观，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。

```
import java.math.BigInteger
def factorial(x:BigInteger): BigInteger =
```

---

[3] Raymond, 大教堂和杂货铺【Ray99】
[4] Steele，“发展一门语言”【Ste99】
[5] 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] Scala 伴随着一个标准库，其中的某些东西将被本书所覆盖。若需要得到更多信息，你也可以在 Scala 发布包里，或者在线 http://www.scala-lang.org 查询库的 scaladoc 文档。

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

BigInt 代表了许多其他类似于数字的类型——大十进制数，复数，分数，置信区间，多项式——诸如此类。一些编程语言原生实现了其中的一些类型。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将很容易变得太大而难以管理。更进一步，即使如果有这样的语言，总有些应用会使用其他的没支持的数字类型。所以尝试在一种语言里提供所有东西的解决之道不可能很好地伸展。取而代之，Scala 允许用户在他们需要的方向上通过定义易用库来发展和改造语言，使得这些特性**感觉上**好像原生语言支持一样。

培育新的控制结构

前面的例子演示了 Scala 让你增加新的类型，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控制结构上。这种类型的扩展是由 Scala 的“基于行动类”的并发编程 API 阐明的。

随着近年多核处理器的激增，为了获取可接受的性能，你将必须在应用中运用更多的并行机制。常常这就意味着重写你的代码来让计算分布到若干并发线程上。不幸的是，创建依赖性的多线程程序在实践中被证明是非常具有挑战性的。Java 的线程模型是围绕着共享内存和锁建立的，尤其是当系统在大小和复杂度都得到提升的时候，这种模型常常是不可理喻的。很难说程序里面没有资源竞争或潜藏的死锁——有些东西不是能在测试里面检验得出，而或许只在投入生产后才表现出来。而大致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“行动类”方案。

Java 伴随着一个丰富的，基于线程的并发库。Scala 可以像其他 JavaAPI 那样使用它编程。然而，Scala 也提供了一个实质上实现了 Erlang 的行动类模型的附加库。

行动类是能够实现于线程之上的并发抽象。它们通过在彼此间发送消息实现通信。每个行动类都能实现两个基本操作，消息的发送和接受。发送操作，用一个惊叹号表示，发送消息给一个行动类。这里用一个命名为 `recipient` 的行动类举例如下：

```
recipient ! msg
```

发送是异步的；就是说，发送的行动类可以在一瞬间完成，而不需要等待消息被接受和处理。每一个行动类都有一个**信箱**：***mailbox*** 把进入的消息排成队列。行动类通过 `receive` 代码块处理信箱中受到的消息：

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

接收代码块由许多 case 语句组成，每一个都用一个消息模板查询信箱。信箱中第一个符合任何 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符合任何 case

的消息，行动类将休眠等待新进的消息。

这里举一个简单的 Scala 行动类实现检查值（cheksum）计算器服务的例子：

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

这个行动类首先定义了一个名为 sum 的本地变量，并赋了初值为零。然后就用 receive 段落重复等待在消息循环中。如果收到了 Data 消息，就把发送的 bytes 取哈希值加到 sum 变量中。如果收到了 GetSum 消息，就用消息发送 requester!sum 把当前 sum 值发回给 requester。requester 字段嵌入在 GetSum 消息里；它通常指出创建请求的行动类。

目前我们并不指望你能完全明白行动类例子。实际上，对于可伸展性这个话题来说这个例子里面最重要的是，不论是 actor 还是 loop 还是 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 看上去或者表现上都如此接近于控制结构如 while 或者 for 循环，实际上它们是定义在 Scala 的行动类库里面的方法。同样，尽管“!”看上去像是个内建的操作符，它也不过是定义在行动类库里面的方法。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 还实现了 Erlang 其他并发编程构件的大多数，诸如监控失败行动类和超时类。总体来说，行动类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，给人的感觉就像行动类是 Scala 语言整体的部分。

本例演示了你可以向新的方向“培养”Scala 语言乃至像并发编程这样的特化。前提是，你需要一个好的架构和程序员来做这样的事。但重要的事情是这的确可行——你可以在 Scala 里面设计和实现抽象结构，从而快速投入新的应用领域，却仍然感觉像是原生的语言支持。

1.2　什么使得 Scala 具有伸缩性？

伸缩性受许多因素影响，范围从语法细节到控件的抽象构造。如果我们一定要说出 Scala 中有助伸缩性的一个方面，我们会把面向对象和函数式编程的组合拣出来（呵呵，不厚道了一把，这的确是两个方面，但是纠缠在了一起）。

Scala 在把面向对象和函数式编程熔合成一套语言的设计方面比其他众所周知的语言都走得更远。比方说，其他语言或许把对象和方法作为两个不同的概念，但在 Scala 里，函数值就**是**对象。函数类型是能够被子类继承的类。这看上去似乎不外乎学术上的美感，但它从深层次上影响了可伸展性。实际上之前看到的行动类这个概念如果没有这种函数和对象的联合将无法实现。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

面向对象编程已经无与伦比地成功了。它开始于（20 世纪）60 年代中期的 Simula 和 70 年代的 Smalltalk，现在 支持它的语言比不支持的更多。某些领域已经被对象完全接管了。然而并没有面向对象意味着什么的明确定义，很明显对象的某些东西是程序员说了算的。

原则上，面向对象编程的动机非常简单：除了最琐碎的程序之外的绝大多数都需要某些结构。做的这点最直接的办法就是把数据和操作放进某种形式上的容器。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，这样它们就能像保存数据那样保存操作，并且它们是自己的值，可以存储到其他容器里，或作为参数传递给操作。这样的容器就被叫做对象。Alan Kay，Smalltalk的发明者，评论说，用这样的方法最简单的对象可以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]于是对象在语言伸缩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。

尽管很长一段时间面向对象编程已经成为主流，然而鲜有语言能跟从 Smalltalk 推动这种构造原则去转化为逻辑结论。举例来说，许多语言容忍值不是对象，如 Java 里面的原始值。或者它们允许静态字段和方法不隶属于任何对象。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，但它们有一个讨厌的趋势，把事情复杂化并限制了可伸缩性。

相反，Scala 是纯粹格式的面向对象语言：每个值都是对象，每个操作都是方法调用。例如，如果你用 Scala 描述 `1 + 2`，你实际上调用了定义在 `Int` 类里面一个名为 `+` 的方法。你可以用一个像操作符一样的名字定义方法，这样你的 API 的使用者就能按照操作符的标记使用了。这就是前例里面显示的 Scala 的行动类 API 定义者如何让你能够使用类似 `requester!sum` 这样的表达式：“`!`”是行动类的方法。

如果说到对象组合，Scala 比多数别的语言更胜一筹。Scala 的**特质**：***trait*** 就是其中一例。所谓特质就像 Java 的接口，但它们同样可以有方法实现乃至字段。对象是由**混入组成**：***mixin composition*** 构造的，这种方式使用类的定义并加入一定数量的特质定义构成。用这种方式，不同方面的类可以被包装入不同的特质。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，特质可以可以把一些新的功能加入到还未定义的超类中。这使得特质比类更具有“可加性”。尤其特别的是，它避免了多重继承里面，当同样的类被通过若干不同渠道继承时发生的，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“全须全尾儿”的函数式语言。函数式语言的思想早于（电子）计算机。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，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，最近几年对函数式语言和技术的热情持续高涨。

函数式编程有两种理念做指导，第一种理念是函数是第一类值。在函数式语言中，函数也是值，与，比如说，整数或字串，在同一个地位。你可以把函数当作参数传递给其他函数，当作结果从函数中返回或保存在变量里。你也可以在函数里定义其他函数，就好像在函数

[7] Kay，《Smalltalk 的早期历史》【Kay96】

里定义整数一样。还可以定义匿名函数，就好像你或许会写像 42 这样的整数文本那样方便地用函数文本抛洒在代码中。

把函数作为第一类值为操作符上的抽象和创建新控制结构提供了便利的方法。这种函数的泛化提供了很强的表现力，常能产生非常易读和清晰的程序。而且常在伸展性上扮演重要的角色。例如，之前在行动类例子里演示的 `receive` 构造就是一个把函数当作参数调用的方法。`receive` 构造里面的代码是个未被执行的传入 `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为二类地位。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，它们不允许你定义指向环境中什么值的第一类嵌套函数，也不能定义匿名函数文本。

函数式编程的第二个主要理念是程序的操作符应该把输入值映射到输出值而不是就地修改数据。要看到其中的差别，可以考虑一下 Ruby 和 Java 对字串的实现。在 Ruby 里，字串是一个字符数组。字串中的字符可以被独立的改变。举例来说你可以在同一个字串对象里把分号改成句号。而另一方面，在 Java 和 Scala 里，字串是一种数学意义上的字符序列。使用表达式如 `s.replace(';', '.')`在字串里替换字符会产生一个新的，不同于原字串 `s` 的对象。用另一种表达方式来说就是在 Java 里字串是不可变的（immutable）而在 Ruby 里是可变的。因此单看字串来说，Java 是函数式语言，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。

另一种说明函数式编程第二种理念的方式是方法不应有任何**副作用**：***side effect***。它们唯一的与所在环境交流的方式应该是获得参数和返回结果。举例来说，Java 的 `String` 类的 `replace` 方法符合这个描述。它带一个字串和两个字符并产生一个所有一个字符都被另一个替代掉的新字串。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***，就是说方法调用对任何给定的输入可以用它的结果替代而不会影响程序的语义。

函数式语言鼓励不可变数据结构和指称透明的方法。有些函数式语言甚至需要它们。Scala 给你选择。如果你需要，你也可以写成**命令**：***imperative*** 形式，用可变数据和有副作用的方法调用编程。但是 Scala 通常可以在你需要的时候轻松避免它们，因为有好的函数式编程方式做替代。

1.3　为什么选择 Scala？

Scala 是为你准备的吗？你必须自己看明白并做决定。除了伸展性之外，我们发现喜欢用 Scala 编程实际上还有很多理由。最重要的四个将在本节讨论的方面该是：兼容性，简短，高层级抽象和高级的静态类别。

Scala 是兼容的

Scala不需要你从Java平台后退两步然后跳到Java语言前面去。它允许你在现存代码中加点儿东西——在你已有的东西上建设——因为它被设计成无缝地与Java实施互操

作。[8]Scala程序会被编译为JVM的字节码。它们的执行期性能通常与Java程序一致。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，显式接口描述，或粘接代码。实际上，几乎所有Scala代码都极度依赖于Java库，而经常无须在程序员意识到这点。

交互式操作的另一个方面是 Scala 极度重用了 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 的数组被映射到 Java 数组。Scala 同样重用了许多标准 Java 库类型。例如，Scala 里的字串文本`"abc"`是 `java.lang.String`，而抛出的异常必须是 `java.lang.Throwable` 的子类。

Scala 不仅重用了 Java 的类型，还把它们“打扮”得更漂亮。例如，Scala 的字串支持类似于 `toInt` 和 `toFloat` 的方法，可以把字串转换成整数或者浮点数。因此你可以写 `str.toInt` 替代 `Integer.parseInt(str)`。如何在不打破互操作性的基础上做到这点呢？Java 的 `String` 类当然不会有 `toInt` 方法。实际上，Scala 有一个解决这种高级库设计和互操作性不相和谐的通用方案。Scala 可以让你定义**隐式转换**：***implicit conversion***，这常常用在类型失配，或者选用不存在的方法时。在上面的例子里，当在字串中寻找 `toInt` 方法时，Scala 编译器会发现 `String` 类里没有这种方法，但它会发现一个把 Java 的 `String` 转换为 Scala 的 `RichString` 类的一个实例的隐式转换，里面定义了这么个方法。于是在执行 toInt 操作之前，转换被隐式应用。

Scala 代码同样可以由 Java 代码调用。有时这种情况要更加微妙，因为 Scala 是一种比 Java 更丰富的语言，有些 Scala 更先进的特性在它们能映射到 Java 前需要先被编码一下。第 29 章说明了其中的细节。

Scala 是简洁的

Scala 程序一般都很短。Scala 程序员曾报告说与 Java 比起来代码行数可以减少到 1/10。这有可能是个极限的例子。较保守的估计大概标准的 Scala 程序应该有 Java 写的同样的程序一半行数左右。更少的行数不仅意味着更少的打字工作，同样意味着更少的话在阅读和理解程序上的努力及更少的出错可能。许多因素在减少代码行上起了作用。

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 语法里还有很多其他的地方省略了东西。比方说，比较一下你在 Java 和 Scala 里是如何写类及构造函数的。在 Java 里，带有构造函数的类经常看上去是这个样子：

```
// 在 Java 里
class MyClass {
  private int index;
  private String name;
  public MyClass(int index, String name) {
    this.index = index;
    this.name = name;
  }
}
```

[8] 也有跑在.NET 平台下的 Scala 变体，但是在 JVM 下的变体目前支持的最好。

在 Scala 里，你会写成这样：

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

根据这段代码，Scala编译器将制造有两个私有字段的类，一个名为index的Int类型和一个叫做name的String类型，还有一个用这些变量作为参数获得初始值的构造函数。这个构造函数还将用作为参数传入的值初始化这两个字段。一句话，你实际拿到了与罗嗦得多的Java版本同样的功能。[9]Scala类写起来更快，读起来更容易，最重要的是，比Java类更不容易犯错。

有助于 Scala 的简洁易懂的另一个因素是它的类型推断。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。

但或许减少代码最关键的是因为已经存在于你的库里而不需要写的代码。Scala 给了你许多工具来定义强有力的库让你抓住并提炼出通用的行为。例如，库类的不同方面可以被分成若干特质，而这些有可以被灵活地混合在一起。或者，库方法可以用操作符参数化，从而让你有效地定义那些你自己控制的构造。这些构造组合在一起，就能够让库的定义既是高层级的又能灵活运用。

Scala 是高层级的

程序员总是在和复杂性死磕。为了高产出的编程，你必须明白你工作的代码。过度复杂的代码成了很多软件工程崩溃的原因。不幸的是，重要的软件往往有复杂的需求。这种复杂性不可避免；必须（由不受控）转为受控。

Scala 可以通过让你提升你设计和使用的接口的抽象级别来帮助你管理复杂性。例如，假设你有一个 `String` 变量 `name`，你想弄清楚是否 `String` 包含一个大写字符。在 Java 里，你或许这么写：

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

在 Scala 里，你可以写成：

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

Java 代码把字串看作循环中逐字符步进的低层级实体。Scala 代码把同样的字串当作能用**论断**：***predicate*** 查询的字符高层级序列。明显 Scala 代码更短并且——对训练有素的眼睛来说——比 Java 代码更容易懂。因此 Scala 代码在通盘复杂度预算上能极度地变轻。它也更少给你机会犯错。

---

[9] 唯一实际的差别在于 Scala 例子里制造的字段是不变的（final）。你将在 10.6 节里学习如何把它们变成可变的（non-final），

论断，_.isUpperCase，是一个Scala里面函数式文本的例子。[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。为此需要定义一个包含抽象功能的方法的接口。例如，如果你想支持对字串的查询，就应引入一个只有一个方法 hasProperty 的接口 CharacterProperty：

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

然后你可以在 Java 里用这个接口格式一个方法 exists：它带一个字串和一个 CharacterProperty 并返回真如果字串中有某个字符符合属性。然后你可以这样调用 exists：

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

然而，所有这些真的感觉很重。重到实际上多数 Java 程序员都不会惹这个麻烦。他们会宁愿写个循环并漠视他们代码里复杂性的累加。另一方面，Scala 里的函数式文本真地很轻量，于是就频繁被使用。随着对 Scala 的逐步了解，你会发现越来越多定义和使用你自己的控制抽象的机会。你将发现这能帮助避免代码重复并因此保持你的程序简短和清晰。

Scala 是静态类型的

静态类型系统认定变量和表达式与它们持有和计算的值的种类有关。Scala坚持作为一种具有非常先进的静态类型系统的语言。从Java那样的内嵌类型系统起步，能够让你使用**泛型：*generics***参数化类型，用**交集：*intersection***联合类型和用**抽象类型：*abstract type***隐藏类型的细节。[12]这些为建造和组织你自己的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，你或许发现Scala把它的静态类型系统列为其优点之一有些奇怪。毕竟，没有静态类型系统已被引为动态语言的某些主要长处。绝大多数普遍的针对静态类型的论断都认为它们使得程序过度冗长，阻止程序员用他们希望的方式表达自己，并使软件系统动态改变的某些模式成为不可能。然而，这些论断经常针对的不是静态类型的思想，而是指责特定的那些被意识到太冗长或太不灵活的类型系统。例如，Alan Kay，Smalltalk语言的发明者，有一次评论：“我不是针对类型，而是不知道有哪个没有完痛的类型系统，所以我还是喜欢动态类型。”[13]

---

[10] 返回类型为 Boolean 的函数式文本被称作论断。
[11] 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] 泛型将在第十九章，交集将在第十二章，抽象类型将在第二十章讨论。
[13] Kay，在面向对象编程的含义的 email 中。【Kay03】

我们希望能在书里说服你，Scala 的类型系统是远谈不上会变成“完痛”。实际上，它漂亮地说明了两个关于静态类型通常考虑的事情（的解决方案）：通过 类型推断避免了赘言和通过模式匹配及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将更被赏识。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，可以保证这样的属性：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；函数带了正确个数的参数；只有字串可以被加到字串集之中。

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，或除零错误。同样也查不到你的程序不符合式样书（假设有这么一份式样书）。静态类型系统因此被认为不很有用而被忽视。舆论认为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，又为何自寻烦恼呢？我们认为这种论调不对头。尽管静态类型系统确实不能**替代**单元测试，但是却能减少用来照顾那些确需测试的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，但它们是无论多少测试都不能给的真正的保证。

安全的重构。静态类型系统提供了让你具有高度信心改动代码基础的安全网。试想一个对方法加入额外的参数的重构实例。在静态类型语言中，你可以完成修改，重编译你的系统并容易修改所有引起类型错误的代码行。一旦你完成了这些，你确信已经发现了所有需要修改的地方。对其他的简单重构，如改变方法名或把方法从一个类移到另一个，这种确信都有效。所有例子中静态类型检查会提供足够的确认，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，当它们弄乱程序时，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不可能很容易地从程序中自己想出来的。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；没必要同样的句子重复两遍。

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。在上例中，以下两个不太讨厌的替代品也能一样工作：

[14] Dijkstra，“Notes on Structured Programming”，7。【Dij70】

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

Scala 里的类型推断可以走的很远。实际上，就算用户代码丝毫没有显式类型也不稀奇。因此，Scala 编程经常看上去有点像是动态类型脚本语言写出来的程序。尤其显著表现在作为粘接已写完的库控件的客户应用代码上。而对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。这很自然。综上，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

Scala 的设计受许多编程语言和研究思想的影响。事实上，仅很少的 Scala 的特点是全新的；大多数都已经被以另外的形式用在其他语言中了。Scala 的革新主要来源于它是如何构造并放在一起的。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，而它们大部分借自于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，诸如它的基本类型，类库和它的执行模式。

Scala也欠了其他语言的很多情。它的统一对象模型是由Smalltalk发起的，之后又被Ruby发扬光大。他的通用嵌套的思想（几乎所有的 Scala 里的构造都能被嵌套进其他构造）也出现在 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 的基于行动类的并发库几乎全是 Erlang 的思想。

Scala不是第一种强调伸展性和扩展性的语言。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“之后的 700 种编程语言”[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）。把前缀的操作符视为函数的特别的思想可以被回溯到Iswim和Smalltalk。另一个重要的思想是允许函数式文本（或代码块）作为参数，从而能让库定义控制结构。同样可以追回到Iswim和Smalltalk。Smalltalk和Lisp两者都具有灵活的语法，广泛应用在建造内嵌的领域特化的语言。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伸展语言；与Scala相较，它是建在低层级，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 里集成了函数式编程的一些元素的包括 Ruby，Smalltalk 和 Python。在 Java

---

[15] 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。这种背离的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`”这种语法这样做很简单——只要省略冒号和类型即可。但是在 C 风格“`Type variable`”语法里你没办法简单地去掉类型——没办法有标记开始定义了。一些替代的关键字将作为占位符替掉缺少的类型（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] Landin，“The Next 700 Programming Languages”【Lan66】

平台上，Pizza，Nice 和 Multi-Java 都用函数式思想扩展了类 Java 内核。还有一些接受了对象系统的以函数式为主的语言；OCaml，F#和 PLT-Scheme 是其中的例子。

Scala同样也对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，还有他的拆分器提供了独立于表达的方式去做模式匹配。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　结语

本章里，我们走马观花的了解了什么是 Scala 和它怎样在编程中帮助你。说句实话，Scala 不是一颗能魔法般让你更多产的银弹。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。本书的目的是引领你变缓 Scala 的学习曲线，一次一小步。我们认为你将发现这是有回报的智力体验，扩展你的知识面并让你有一些不同的想法。希望你能同样获得快乐和使用 Scala 编程的动力。

下一章，我们将带你开始写一些 Scala 的代码。

[17] 更多信息请在参考书目中查阅【Ode03】，【Ode05】，和【Emi07】。

第2章

Scala 的第一步

是时候写点儿 Scala 代码了。在我们开始深度 Scala 教程之前，我们将用两章来给你画一张 Scala 大致的图纸，更重要的是，带你写一些代码。我们鼓励你实际尝试所有出现在本章以及后续章节中的代码例子。开始学习 Scala 最好的方法就是用它编程。

要执行本章的例子，你应该有一份标准的Scala安装。想要的话，可以到http://www.scala-lang.org/downloads并依照你的平台的向导。你也可以使用Eclipse，IntelliJ，或NetBeans的Scala插件，但是对于本章的这几步来说，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 的编程老手，接下来的两张将给你足够的知识让你能用 Scala 写些有用的程序。如果你缺乏经验，那么其中的一些材料对你来说或许会显得有些神秘。不过别急。为了加快你上手速度，我们不得不抛下一些细节。所有的东西将在后续章节中以更不“**流水**：***firehose***”的风格解释。另外，我们在接下来的两章里插入了相当多的脚注来提供更多的信息并指引你到本书后续段落中去发现更详细的解释。

第一步：学习使用 Scala 解释器

开始Scala最简单的方法是使用Scala解释器，它是一个编写Scala表达式和程序的交互式“shell”。简单地在解释器里输入一个表达式，它将计算这个表达式并打印结果值。Scala的交互式shell就叫做`scala`。你可以在命令提示符里输入`scala`使用它：[2]

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

在你输入表达式，如 1 + 2，并敲了回车之后：

```
scala> 1 + 2
```

解释器会打印：

```
res0: Int = 3
```

这行包括：

- 一个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），
- 一个冒号（`:`），跟着表达式的类型（Int），

[1] 我们用 Scala 版本 2.7.2 测试了本书的例子。
[2] 如果你在使用 Windows，你将需要在“命令提示符”这个 DOS 窗口中输入 `Scala` 命令。

- 一个等号（=），
- 计算表达式所得到的结果（3）。

Int类型指代了scala包的类Int。Scala里的包与Java里的包很相似：它们把全局命名空间分区并提供了信息隐藏的机制。[3]类Int的值对应着Java的int值。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。当你把你的Scala代码编译成Java字节码，Scala编译器将使用Java的原始类型以便获得其带来的性能益处。

resX 识别符还将用在后续的代码行中。例如，既然 res0 已在之前设为 3，res0 * 3 就是 9：

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

打印必要的，却不仅此而已的，Hello, world! 贺词，输入：

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

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

第二步：定义一些变量

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 就不能再赋值了。与之对应的，var 如同 Java 里面的非 final 变量。var 可以在它生命周期中被多次赋值。下面是一个 val 的定义：

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

这个语句引入了 msg 当作字串"Hello, world!"的名字。类型是 java.lang.String，因为 Scala 的字串是由 Java 的 String 类实现的。

如果你之前曾定义过 Java 变量，你会发现一个很醒目的差别：无论 java.lang.String 还是 String 都没有出现在 val 的定义中。本例演示了**类型推断**：*type inference*，这种 Scala 能自动理解你省略的类型的能力。在这个例子里，因为你用一个字串文本初始化了 msg，Scala 推断 msg 的类型是 String。如果 Scala 解释器（或编译器）可以推断类型，那么让它这么做而不是写一些没必要的显式类型标注常常是最好的选择。不过，如果你愿意，也可以显式地定义类型，也许有些时候你也应该这么做。显式的类型标注不但可以确保 Scala 编译器推断你倾向的类型，还可以作为将来代码读者有用的文档。Java 中变量的类型指定在其名称之前，与之不同的是，Scala 里变量的类型在其名称之后，用冒号分隔。如：

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

[3] 如果你不熟悉 Java 的包，你可以把它们认为是提供类的全名。因为 Int 是 scala 包的成员，“Int”是这个类的简化名，“scala.Int”是它的全名。包的细节在第十三章中描述。

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

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

回到原来的那个 msg，现在它定义好了，你可以按你的想法使用它，如：

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

你对msg不能做的，因为是val而不是var，就是再给它赋值。[5]例如，看看你做如下尝试的时候编译器怎么报错的：

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

如果可重赋值是你需要的，你应使用 var，如下：

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

由于 greeting 是 var 而不是 val，你可以在之后对它重新赋值。比如说，如果你之后心态不平了，你可以修改你的 greeting 为：

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

要输入一些能跨越多行的东西，只要一行行输进去就行。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.
```

如果你意识到你输入了一些错误的东西，而解释器仍在等着你更多的输入，你可以通过按两次回车取消掉：

```
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）。

---

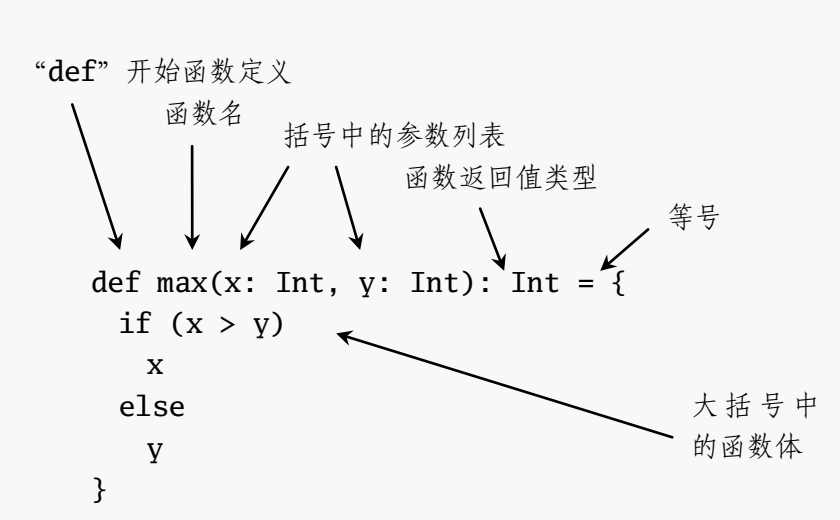
[4] java.lang.String 的简化名是 String。
[5] 然而在解释器中，你可以用一个之前已经使用了的名字**定义**新的 val。这种机制将在 7.7 节中解释。

第三步：定义一些函数

现在你已经用过了 Scala 的变量，或许想写点儿函数。下面是在 Scala 里面的做法：

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

函数的定义用def开始。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。每个函数参数后面必须带前缀冒号的类型标注，因为Scala编译器（还有解释器，但之后我们将只说编译器）没办法推断函数参数类型。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。这个东西定义了max函数的**结果类型**：***result type***。[6]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大括号。本例中，函数体里仅有一个if表达式，选择x或者y，哪个较大，就当作max函数的结果。就像这里演示的，Scala的if表达式可以像Java的三元操作符那样产生一个值。举例来说，Scala表达式“if (x > y) x else y”与Java里的“(x > y) ? x : y”表现得很像。在函数体前的等号提示我们函数式编程的世界观里，函数定义一个能产生值的表达式。函数的基本结构在 图 2.1 里面演示。



图释 2.1 Scala 函数的基本构成

有时候Scala编译器会需要你定义函数的结果类型。比方说，如果函数是递归的，[7]你就必须显式地定义函数结果类型。然而在max的例子里，你可以不用写结果类型，编译器也能够推断它。[8]同样，如果函数仅由一个句子组成，你可以可选地不写大括号。这样，你就可以把max函数写成这样：

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

[6] 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] 如果一个方法调用自身，就称为递归。
[8] 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。

```
max2: (Int,Int)Int
```

一旦你定义了函数，你就可以用它的名字调用它，如：

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

还有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
```

当你定义了 greet()函数，解释器会回应一个 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。Unit 的结果类型指的是函数没有返回有用的值。Scala 的 Unit 类型比较接近 Java 的 void 类型，而且实际上 Java 里每一个返回 void 的方法都被映射为 Scala 里返回 Unit 的方法。因此结果类型为 Unit 的方法，仅仅是为了它们的副作用而运行。在 greet()的例子里，副作用是在标准输出上打印一句客气的助词。

下一步，你将把 Scala 代码放在一个文件中并作为脚本执行它。如果你想离开解释器，输入:quit 或者:q。

```
scala> :quit
$
```

第四步：编写一些 Scala 脚本

尽管 Scala 的设计目的是帮助程序员建造非常大规模的系统，但它也能很好地缩小到做脚本的规模。脚本就是一种经常会被执行的放在文件中的句子序列。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
```

于是你又会得到另外的祝词：

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

通过 Scala 的名为 args 的数组可以获得传递给 Scala 脚本的命令行参数。Scala 里，数组以零开始，通过在括号里指定索引访问一个元素。所以 Scala 里数组 steps 的第一个元素是 steps(0)，不是像 Java 里的 steps[0]。作为测试，输入以下内容到新文件 helloarg.scala：

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

[9] 你可以使用“**制式：*pound-bang***”语法在 Unix 和 Windows 里不输入“scala”就运行脚本，附录 A 中对此作了说明。

然后运行：

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

这条命令里，"planet"被作为命令行参数传递，并在脚本里作为 args(0)被访问。因此，你会看到：

```
Hello, planet!
```

注意这个脚本包括了一条注释。Scala编译器将忽略从//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。这与你的预期一样。表达式"Hello, "+"world!"将产生字符串"Hello, world!"。[10]

第五步：用 while 循环；用 if 判断

要尝试 while，在 printargs.scala 文件里输入以下代码：

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

注意

虽然本节的例子有助于解释 while 循环，但它们并未演示最好的 Scala 风格。在下一段中，你会看到避免用索引枚举数组的更好的手段。

这个脚本开始于变量定义，var i = 0。类型推断认定 i 的类型是 scala.Int，因为这是它的初始值的类型，0。下一行里的 while 结构使得代码块（大括号之间的代码）重复执行直到布尔表达式 i < args.length 为假。args.length 给出了 args 数组的长度。代码块包含两句话，每个都缩进两个空格，这是 Scala 的推荐缩进风格。第一句话，println(args(i))，输出第 i 个命令行参数。第二句话，i += 1，让 i 自增一。注意 Java 的++i 和 i++在 Scala 里不起作用，要在 Scala 里自增，必须写成要么 i = i + 1，或者 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
```

想要更好玩儿一些，就把下列代码输入到新文件 echoargs.scala：

```
var i = 0
while (i < args.length) {
  if (i != 0)
```

[10] 你也可以把空格放在加号的旁边，如"Hello, " + "world!"。然而本书中，我们将把‘+’和字串文本间的空格去掉。

```
    print(" ")
  print(args(i))
  i += 1
}
println()
```

在这个版本里，用 print 调用替代了 println 调用，这样所有参数将被输出在同一行里。为了更好的可阅读性，你应该用 if(i != 0)检查，除了第一个之外的每个参数前插入一个空格。由于第一次做 while 循环时 i != 0 会失败，因此在头一个参数之前不会输出空格。最后，你应该在末尾多加一个 println，这样在输出所有参数之后会有一个换行。这样你的输出就非常漂亮了。如果用下面的命令运行脚本：

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

就能得到：

```
Scala is even more fun
```

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，就是不能像在 Ruby 里面那样在 Scala 里这么写：if i < 10。在 Scala 里必须写成 if (i < 10)。）另外一点与 Java 类似的，是如果代码块仅有一个句子，大括号就是可选的，就像 echoargs.scala 里面 if 句子演示的。并且尽管你没有看到，Scala 也和 Java 一样使用分号分隔句子的，只是 Scala 里的分号经常是可选的，从而可以释放你的右小手指。如果你有点儿罗嗦的脾气，那么就把 echoargs.scala 脚本写成下面的样子好了：

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

第六步：用 foreach 和 for 枚举

尽管或许你没意识到，在前一步里写 while 循环的时候，你正在用**指令式**：*imperative* 风格编程。指令式风格，是你常常使用像 Java，C++和 C 这些语言里用的风格，一次性发出一个指令式的命令，用循环去枚举，并经常改变共享在不同函数之间的状态。Scala 允许你指令式地编程，但随着你对 Scala 的深入了解，你可能常会发现你自己在用一种更**函数式**：*functional* 的风格编程。实际上，本书的一个主要目的就是帮助你变得对函数式风格感觉像和指令式风格一样舒适。

函数式语言的一个主要特征是，函数是第一类结构，这在 Scala 里千真万确。举例来说，另一种（简洁得多）打印每一个命令行参数的方法是：

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

这行代码中，你在 args 上调用 foreach 方法，并把它传入函数。此例中，你传入了带有一个叫做 arg 参数的**函数文本**：***function literal***。函数体是 println(arg)。如果你把上述代码输入到新文件 pa.scala，并使用命令执行：

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

你会看到：

```
Concise
is
nice
```

前例中，Scala 解释器推断 arg 的类型是 String，因为 String 是你调用 foreach 的那个数组的元素类型。如果你喜欢更显式的，你可以加上类型名，不过如此的话你要把参数部分包裹在括号里（总之这是语法的普通形式）：

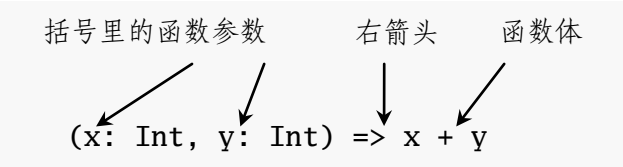
```
args.foreach((arg: String) => println(arg))
```

运行这个脚本的结果与前一个相同。

如果你更喜欢简洁的而不是显式的风格，就可以充分体会到Scala特别简洁的优越性。如果函数文本由带一个参数的一句话组成，你都不需要显式命名和指定参数。[11]这样，下面的代码同样有效：

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

总而言之，函数文本的语法就是，括号里的命名参数列表，右箭头，然后是函数体。语法演示在图 2.2 中。



图释 2.2 Scala 函数文本的语法

现在，到这里你或许想知道那些你在指令式语言如 Java 或 C 里那么信任的 for 循环到哪里去了呢。为了努力引导你向函数式的方向，Scala 里只有一个指令式 for（称为 **for 表达式**：***expression***）的函数式近似。目前你还看不到他们全部的力量和表达方式，直到你读到了（或者先瞄一眼）第 7.3 节，我们仅仅带您在这里领略一下。创建一个新文件 forargs.scala，输入以下代码：

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

这个表达式里“for”之后的括号包含arg<-args。[12]<-右侧的是熟悉的args数组。<-左侧的是“arg”，val的名称（不是var）。（因为总归是val，你只要写arg就可，不要写成val arg。）尽管arg可能感觉像var，因为他在每次枚举都会得到新的值，但它的确是val：arg不能在for表达式的函数体中重新赋值。取而代之，对每个args数组的元素，一个新的arg val

[11] 这种简写被称为**偏应用函数**：***partially applied function***，将在 8.6 节里描述。
[12] 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”。

将被创建并初始化为元素值，然后for的函数体将被执行。

如果执行 forargs.scala 脚本：

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

可以看到：

```
for
arg
in
args
```

Scala 的 for 表达式可以比这个做得更多，但是这个例子足以让你起步了。我们将在 7.3 节和第二十三章中展示给你更多关于 for 的东西。

结语

本章，你学习了一些 Scala 的基础并，但愿，利用此机会去写了一些 Scala 代码。下一章，我们将继续这个概况介绍并深入一些更先进的话题中去。

第3章

Scala 的下一步

本章继续前一章对 Scala 的介绍。本章里，我们会介绍一些更先进的特征。在你完成本章之后，应该已经有足够的知识去让你开始用 Scala 写一些有用的脚本。和前一章一样，我们建议你一边阅读一边试一下这些例子。感受 Scala 的最好办法是开始写它。

第七步：带类型的参数化数组

Scala 里可以使用 new 实例化对象或类实例。当你在 Scala 里实例化对象，可以使用值和类型把它**参数化**：***parameterize***。参数化的意思是在你创建实例的时候“设置”它。通过把加在括号里的对象传递给实例的构造器的方式来用值参数化实例。例如，下面的 Scala 代码实例化一个新的 java.math.BigInteger 并使用值"12345"参数化：

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

通过在方括号里设定一个或更多类型来参数化实例。代码 3.1 里展示了一个例子。在这个例子中，greetStrings 是类型 Array[String]（字串数组）的值，并被第一行代码里的值 3 参数化，使它的初始长度为 3。如果把代码 3.1 里的代码作为脚本执行，你会看到另一个 Hello, world!的祝词。请注意当你同时用类型和值去参数化实例的时候，类型首先在方括号中出现，然后跟着值在圆括号中。

```
val greetStrings = new Array[String](3)
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
for (i <- 0 to 2)
  print(greetStrings(i))
```

代码 3.1 用类型参数化数组

注意

尽管代码 3.1 里的代码演示了一些重要的概念，但它没有展示 Scala 里创建和初始化数组的推荐方式。你会在第 48 页的代码 3.2 中看到更好的方式。

如果想用一种更显式的方式，你可以显式定义 greetStrings 的类型：

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

由于 Scala 有类型推断，这行代码与代码 3.1 里的第一行代码语义一致。不过这种形式说明了类型参数化部分（方括号里的类型名）形成了实例类型的部分，而值参数化部分（圆括号里的值）不是。greetStrings 的类型是 Array[String]，不是 Array[String](3)。

代码 3.1 的下三行代码初始化了 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 里的数组是通过把索引放在圆括号里面访问的，而不是像 Java 那样放在方括号里。所以数组的第零个元素是 `greetStrings(0)`，不是 `greetStrings[0]`。

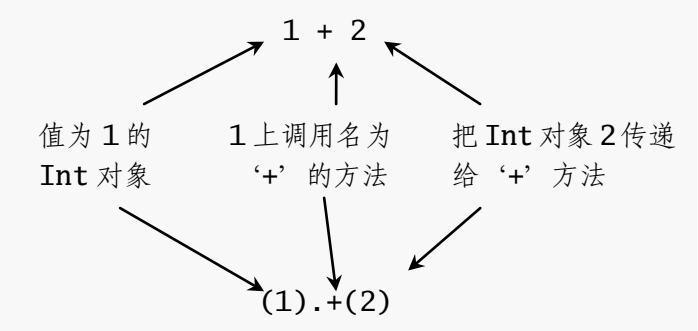
这三行代码演示了搞明白 Scala 如何看待 `val` 的意义的重要概念。当你用 `val` 定义一个变量，那么这个变量就不能重新赋值，但它指向的对象却仍可以暗自改变。所以在本例中，你不能把 `greetStrings` 重新赋值成不同的数组；`greetStrings` 将永远指向那个它被初始化时候指向的同一个 `Array[String]`实例。但是你**能**一遍遍修改那个 `Array[String]`的元素，因此数组本身是可变的。

代码 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演示了Scala的另一个通用规则：如果方法仅带一个参数，你可以不带点或括号的调用它。本例中的`to`实际上是带一个`Int`参数的方法。代码 `0 to 2` 被转换成方法调用`(0).to(2)`。[1]请注意这个语法仅在你显示指定方法调用的接受者时才起作用。不可以写 `pringln 10`，但是可以写成“`Console println 10`”。

从技术上讲，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，当第一步里你在 Scala 解释器里输入 `1 + 2`，你实际上正在 `Int` 对象 `1` 上调用一个名为`+`的方法，并把 `2` 当作参数传给它。如图 3.1 所示，你也可以使用传统的方法调用语法把 `1 + 2` 替代写成`(1).+(2)`。



图释 3.1 Scala 里所有的操作符都是方法调用

这里演示的另一重要思想可以让你看到为什么数组在 Scala 里是用括号访问的。与 Java 比 Scala 很少有特例。数组和 Scala 里其他的类一样只是类的实现。当你在一个或多个值或变量外使用括号时，Scala 会把它转换成对名为 `apply` 的方法调用。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它的一样的方法调用。这个原则不仅仅局限于数组：任何对某些在括号中的参数的对象的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。所以这不是一个特例，而是一个通则。

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，编译器将把它转化为对

[1] 这个 to 方法实际上返回的不是一个数组而是一个不同种类的序列，包含值 0，1 和 2，可以让 for 表达式遍历。序列和其他集合将在第十七章描述。

带有括号里参数和等号右边的对象的update方法的调用。例如，

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

将被转化为

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

因此，下列 Scala 代码与你在代码 3.1 里的代码语义一致：

```
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 在对待任何事上追求概念的简洁性，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 里原始类型和相应的包装类间的，或者数组和正常的对象间的差别。而且这种统一并未损害重要的性能代价。Scala 编译器使用 Java 数组，原始类型，及可存在于编译完成代码里的原生数学类型。

尽管目前为止在这一步里你看到的例子编译运行良好，Scala 提供了通常可以用在你真实代码里的更简洁的方法创造和初始化数组。它看起来就像展示在代码 3.2 中的样子。这行代码创建了长度为 3 的新数组，用传入的字串"zero"，"one"和"two"初始化。编译器推断数组的类型是 Array[String] ，因为你把字串传给它。

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

代码 3.2 创造和初始化数组

你在代码 3.2 里实际做的就是调用了一个叫做apply的工厂方法，从而创造并返回了新的数组。apply方法带可变数量个参数[2]，被定义在Array的**伴生对象**：***companion object***上。你会在 4.3 节里学到更多关于伴生对象的东西。如果你是一个Java程序员，你可以认为这个就像在Array类上调用一个叫做apply的静态方法。更罗嗦的调用同样的apply方法的办法是：

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

第八步：使用 List

方法不应该有副作用是函数风格编程的一个很重要的理念。方法唯一的效果应该是计算并返回值。用这种方式工作的好处就是方法之间很少纠缠在一起，因此就更加可靠和可重用。另一个好处（静态类型语言里）是传入传出方法的所有东西都被类型检查器检查，因此逻辑错误会更有可能把自己表现为类型错误。把这个函数式编程的哲学应用到对象世界里意味着使对象不可变。

如你所见，Scala 数组是一个所有对象都共享相同类型的可变序列。比方说 Array[String]

[2] 可变长度参数列表，或者说**重复参数**：***repeated parameters***，在 8.8 节中描述。

仅包含 String。尽管实例化之后你无法改变 Array 的长度，它的元素值却是可变的。因此，Array 是可变的对象。

说到共享相同类型的不可变对象序列，Scala 的 List 类才是。和数组一样，List[String] 包含的仅仅是 String。Scala 的 List，scala.List，不同于 Java 的 java.util.List，总是不可变的（而 Java 的 List 可变）。更通常的说法，Scala 的 List 是设计给函数式风格的编程用的。创建一个 List 很简单。代码 3.3 做了展示：

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

代码 3.3 创造和初始化列表

代码 3.3 中的代码完成了一个新的叫做oneTwoThree的val，并已经用带有整数元素值 1，2 和 3 的新List[Int]初始化。[3]因为List是不可变的，他们表现得有些像Java的String：当你在一个List上调用方法时，似乎这个名字指代的List看上去被改变了，而实际上它只是用新的值创建了一个List并返回。比方说，List有个叫“:::”的方法实现叠加功能。你可以这么用：

```
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.")
```

如果你执行这个脚本，你会看到：

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

或许 List 最常用的操作符是发音为“cons”的“::”。Cons 把一个新元素组合到已有 List 的最前端，然后返回结果 List。例如，若执行这个脚本：

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

你会看到：

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

注意

表达式“1 :: twoThree”中，::是它右操作数，列表 twoThree，的方法。你或许会疑惑::方法的关联性上有什么东西搞错了，不过这只是一个简单的需记住的规则：如果一个方法被用作操作符标注，如 a * b，那么方法被左操作数调用，就像 a.*(b)——除非方法名以冒号结尾。这种情况下，方法被**右**操作数调用。因此，1 :: twoThree 里，::方法被 twoThree 调用，传入 1，像这样：twoThree.::(1)。

5.8 节中将描述更多操作符关联性的细节。

由于定义空类的捷径是Nil，所以一种初始化新List的方法是把所有元素用cons操作符串

[3] 不用写成 new List 因为“List.apply()”是被定义在 scala.List **伴生对象**上的工厂方法。你将在第十一步里了解更多关于伴生对象的事情。

起来，Nil作为最后一个元素。[4]比方说，下面的脚本将产生与之前那个同样的输出，“List(1, 2, 3)”：

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

Scala 的 List 包装了很多有用的方法，表格 3.1 罗列了其中的一些。列表的全部实力将在第十六章释放。

为什么列表不支持append？

类 List 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::做前缀则仅花费常量时间。如果你想通过添加元素来构造列表，你的选择是把它们前缀进去，当你完成之后再调用 reverse；或使用 ListBuffer，一种提供 append 操作的可变列表，当你完成之后调用 toList。ListBuffer 将在 22.2 节中描述。

表格 3.1 类型 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"和"d"的新 List[String]）
thrill(2)	返回在 thrill 列表上索引为 2（基于 0）的元素（返回"until"）
thrill.count(s => s.length == 4)	计算长度为 4 的 String 元素个数（返回 2）
thrill.drop(2)	返回去掉前 2 个元素的 thrill 列表（返回 List("until")）
thrill.dropRight(2)	返回去掉后 2 个元素的 thrill 列表（返回 List("Will")）
thrill.exists(s => s == "until")	判断是否有值为"until"的字串元素在 thrill 里（返回 true）
thrill.filter(s => s.length == 4)	依次返回所有长度为 4 的元素组成的列表（返回 List("Will", "fill")）
thrill.forall(s => s.endsWith("l"))	辨别是否 thrill 列表里所有元素都以"l"结尾（返回 true）
thrill.foreach(s => print(s))	对 thrill 列表每个字串执行 print 语句（"Willfilluntil"）
thrill.foreach(print)	与前相同，不过更简洁（同上）
thrill.head	返回 thrill 列表的第一个元素（返回"Will"）
thrill.init	返回 thrill 列表除最后一个以外其他元素组成的列表（返回 List("Will", "fill")）
thrill.isEmpty	说明 thrill 列表是否为空（返回 false）
thrill.last	返回 thrill 列表的最后一个元素（返回"until"）
thrill.length	返回 thrill 列表的元素数量（返回 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（返回 List("Willy", "filly", "until")）
thrill.mkString(", ")	用列表的元素创建字串（返回"will, fill, until"）
thrill.remove(s => s.length == 4)	返回去除了 thrill 列表中长度为 4 的元素后依次排列的元素列表（返回 List("until")）

[4] 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。

`thrill.reverse`	返回含有 `thrill` 列表的逆序元素的列表（返回 `List("until", "fill", "Will")`）
`thrill.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)`	返回包括 `thrill` 列表所有元素，并且第一个字符小写按照字母顺序排列的列表（返回 `List("fill", "until", "Will")`）
`thrill.tail`	返回除掉第一个元素的 `thrill` 列表（返回 `List("fill", "until")`）

第九步：使用 Tuple

另一种有用的容器对象是**元组**：*tuple*。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。而列表应该是 `List[Int]`或 `List[String]`的样子，元组可以同时拥有 `Int` 和 `String`。元组很有用，比方说，如果你需要在方法里返回多个对象。Java 里你将经常创建一个 JavaBean 样子的类去装多个返回值，Scala 里你可以简单地返回一个元组。而且这么做的确简单：实例化一个装有一些对象的新元组，只要把这些对象放在括号里，并用逗号分隔即可。一旦你已经实例化了一个元组，你可以用点号，下划线和一个基于 1 的元素索引访问它。代码 3.4 展示了一个例子：

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

代码 3.4 创造和使用元组

代码 3.4 的第一行，你创建了元组，它的第一个元素是以 `99` 为值的 `Int`，第二个是 `"luftballons"`为值的 `String`。Scala 推断元组类型为 `Tuple2[Int, String]`，并把它赋给变量 `pair`。第二行，你访问 `_1` 字段，从而输出第一个元素，`99`。第二行的这个“`.`”与你用来访问字段或调用方法的点没有区别。本例中你正用来访问名叫 `_1` 的字段。如果执行这个脚本，你能看到：

```
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]

访问元组的元素

你或许想知道为什么你不能像访问 `List` 里的元素那样访问元组的，就像 `pair(0)`。那是因为 `List` 的 `apply` 方法始终返回同样的类型，但是元组里的或许类型不同。`_1` 可以有一个结果类型，`_2` 是另外一个，诸如此类。这些 `_N` 数字是基于 1 的，而不是基于 0 的，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。

[5] 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`。

第十步：使用 Set 和 Map

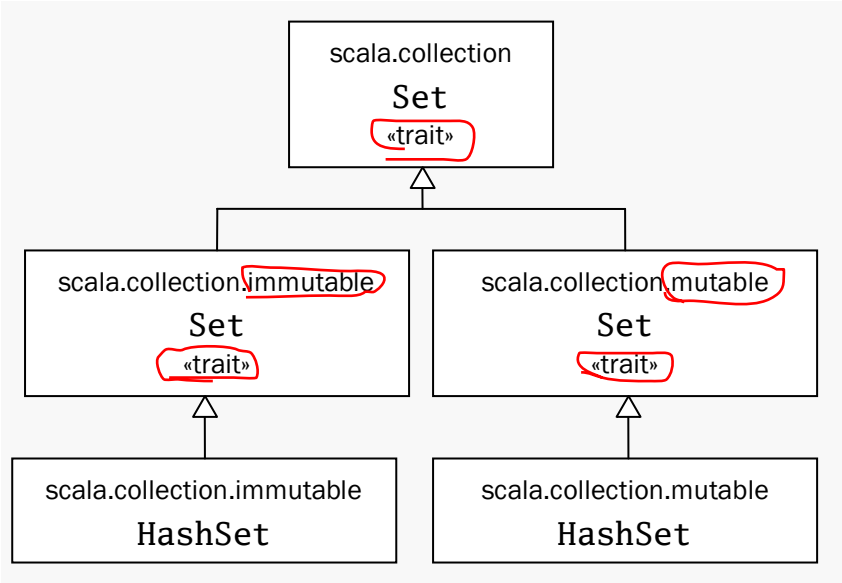
因为 Scala 致力于帮助你充分利用函数式和指令式风格两方面的好处，它的集合类型库于是就区分了集合类的可变和不可变。例如，数组始终是可变的，而列表始终不可变。当问题讨论到集和映射，Scala 同样提供了可变和不可变的替代品，不过用了不同的办法。对于集和映射，Scala 把可变性建模在类继承中。

例如，Scala 的 API 包含了集的一个基本**特质**：***trait***，特质这个概念接近于 Java 的接口。（你将在第 12 章找到更多关于特质的说明。）Scala 于是提供了两个子特质，一个是可变的集，另一个是不可变的集。就如你在图 3.2 里会看到的，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，因为每个都放在不同的包里。Scala 的 API 里具体的 `Set` 类，如图 3.2 的 `HashSet` 类，扩展了要么是可变的，要么不可变的 `Set` 特质。（尽管 Java 里面称为“实现”了接口，在 Scala 里面称为“扩展”或“混入”了特质。）因此，如果你想要使用 `HashSet`，你可以根据你的需要选择可变的或不可变的变体。创造集的缺省方法展示在代码 3.5 中：

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

代码 3.5 创造，初始化，和使用不可变集

代码 3.5 的第一行代码里，定义了名为 `jetSet` 的新 `var`，并使用了包含两个字串，`"Boeing"` 和`"Airbus"`的不可变集完成了初始化。就像例子中展示的，Scala 中创建集的方法与创建列表和数组的类似：通过调用 `Set` 伴生对象的名为 `apply` 的工厂方法。代码 3.5 中，对 `scala.collection.immutable.Set` 的伴生对象调用了 `apply` 方法，返回了一个缺省的，不可变 `Set` 的实例。Scala 编译器推断 `jetSet` 的类型为不可变 `Set[String]`。



图释 3.2 Scala 的 Set 类继承关系

要向集加入新的变量，可以在集上调用+，传入新的元素。可变的和不可变的集都提供了+方法，但它们的行为不同。可变集将把元素加入自身，不可变集将创建并返回一个包含了

添加元素的新集。代码 3.5 中，你使用的是不可变集，因此+调用将产生一个全新集。因此尽管可变集提供的实际上是+=方法，不可变集却不是。本例中，代码的第二行，“jetSet += "Lear"”，实质上是下面写法的简写：

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

因此在代码 3.5 的第二行，你用一个包含了"Boeing"，"Airbus"和"Lear"的新集重新赋值了 jetSet 这个 var。最终，代码 3.5 的最后一行打印输出了集是否包含字串"Cessna"。（正如你所料到的，输出 false。）

如果你需要不可变集，就需要使用一个**引用：*import***，如代码 3.6 所示：

```
import scala.collection.mutable.Set
val movieSet = Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

代码 3.6 创建，初始化，和使用可变集

代码 3.6 的第一行里引用了可变Set。就像 Java那样，引用语句允许你使用简单名，如Set，以替代更长的，全标识名。结果，当你在第三行写Set的时候，编译器就知道你是指scala.collection.mutable.Set。在那行里，你使用包含字串"Hitch"和"Poltergeist"的新可变集初始化了movieSet。下一行通过在集上调用+=方法向集添加了"Shrek"。正如前面提到的，+=是实际定义在可变集上的方法。如果你想的话，你可以替换掉movieSet += "Shrek"的写法，写成movieSet.+=("Shrek")。[6]

尽管目前为止看到的通过可变和不可变的 Set 工厂方法制造的缺省的集实现很可能能够满足极大多数的情况，但偶尔你也或许想要个显式的集类。幸运的是，语法是相同的。只要引用你需要的类，并使用它伴生对象的工厂方法即可。例如，如果你需要一个不可变的 HashSet，你可以这么做：

```
import scala.collection.immutable.HashSet
val hashSet = HashSet("Tomatoes", "Chilies")
println(hashSet + "Coriander")
```

Map 是 Scala 里另一种有用的集合类。和集一样，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，你能在图 3.3 里看到，Map 的类继承机制看上去和 Set 的很像。scala.collection 包里面有一个基础 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。

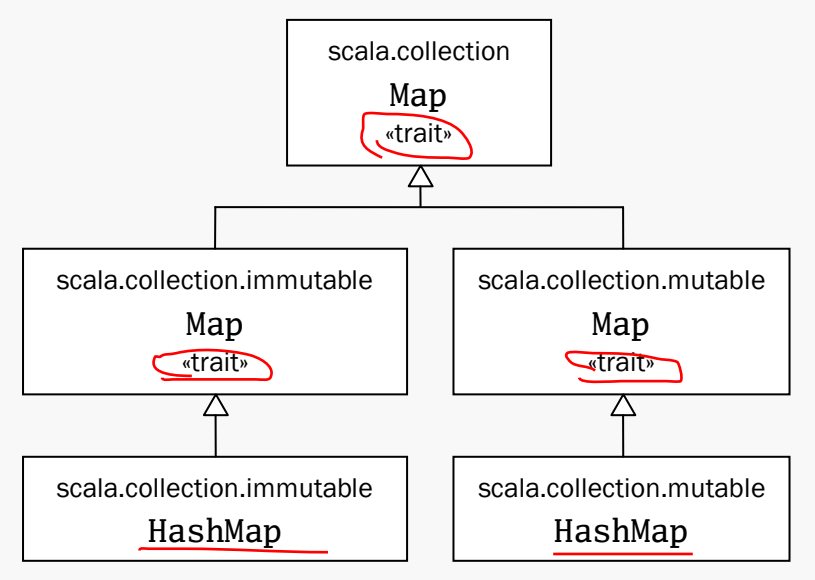
Map 的实现，如显示在类继承图 3.3 里的 HashMap，扩展了要么可变，要么不可变特质。你可以使用与那些用在数组，列表和集中的一样的工厂方法去创造和初始化映射。例如，代码 3.7 展示了可变映射的创造过程：

```
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.")
```

[6] 因为代码 3.6 里的集是可变的，所以不需要对 movieSet 重新赋值，所以它是 val。相对的，在代码 3.5 中对可变集使用+=需要对 jetSet 重新赋值，因此它是 var。

```
println(treasureMap(2))
```

代码 3.7 创造，初始化，和使用可变映射



图释 3.3 Scala 的 Map 类继承关系

代码 3.7 的第一行里，你引用了可变形式的Map。然后就定义了一个叫做treasureMap的val并使用空的包含整数键和字串值的可变Map初始化它。映射为空是因为你没有向工厂方法传递任何值（“Map[Int, String]()”的括号里面是空的）。[7]下面的三行里你使用->和+=方法把键/值对添加到Map里。像前面例子里演示的那样，Scala编译器把如 1 -> "Go to island"这样的二元操作符表达式转换为(1).->("Go to island.")。因此，当你输入 1 -> "Go to island."，你实际上是在值为 1 的Int上调用->方法，并传入值为"Go to island."的String。这个->方法可以调用Scala程序里的任何对象，并返回一个包含键和值的二元元组。[8]然后你在把这个元组传递给treasureMap指向的Map的+=方法。最终，最后一行输出打印了treasureMap中的与键 2 有关的值。如果你执行这段代码，将会打印：

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

如果你更喜欢不可变映射，就不用引用任何类了，因为不可变映射是缺省的，代码 3.8 展示了这个例子：

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

代码 3.8 创造，初始化，和使用不可变映射

由于没有引用，当你在代码 3.8 的第一行里提及 Map 时，你会得到缺省的映射：scala.collection.immutable.Map。传给工厂方法入五个键/值元组，返回包含这些传入的键/值对的不可变 Map。如果你执行代码 3.8 中的代码，将会打印输出 IV。

[7] 显式类型参数化，“[Int, String]”，对代码 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。相对的，如展示在代码 3.8 之中的，编译器可以从传递给映射的工厂方法的值推断参数类型，因此就不需要显式类型参数了。

[8] Scala 里允许你对任何对象调用->的机制被称为**隐式转换**，将在第 21 章里涉及。

第十一步：学习识别函数式风格

第 1 章里提到过，Scala 允许你用**指令式**风格编程，但是鼓励你采用一种更**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 是你有可能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把你向这方面引导。不过这也需要你这方面的一些工作，我们鼓励你付出努力。如果你来自于指令式的背景，我们相信学习用函数式风格编程将不仅让你变成更好的 Scala 程序员，而且还能拓展你的视野并使你变成通常意义上好的程序员。

通向更函数式风格路上的第一步是识别这两种风格在代码上的差异。其中的一点蛛丝马迹就是，如果代码包含了任何 var 变量，那它大概就是指令式的风格。如果代码根本就没有 var——就是说**仅仅**包含 val——那它大概是函数式的风格。因此向函数式风格推进的一个方式，就是尝试不用任何 var 编程。

如果你来自于指令式的背景，如 Java，C++，或者 C#，你或许认为 var 是很正统的变量而 val 是一种特殊类型的变量。相反，如果你来自于函数式背景，如 Haskell，OCamel，或 Erlang，你或许认为 val 是一种正统的变量而 var 有亵渎神灵的血统。然而在 Scala 看来，val 和 var 只不过是你工具箱里两种不同的工具。它们都很有用，没有一个天生是魔鬼。Scala 鼓励你学习 val，但也不会责怪你对给定的工作选择最有效的工具。尽管或许你同意这种平衡的哲学，你或许仍然发现第一次理解如何从你的代码中去掉 var 是很挑战的事情。

考虑下面这个改自于第 2 章的 while 循环例子，它使用了 var 并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
  }
}
```

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)
}
```

这个例子演示了减少使用 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，更不容易犯错的代码。

当然，你可以走得更远。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，其副作用是打印到标准输出流。函数有副作用的马脚就是结果类型为 Unit。如果某个函数不返回任何有用的值，就是说其结果类型为 Unit，那么那个函数唯一能让世界有点儿变化的办法就是通过某种副作用。更函数式的方式应该是定义对需打印的 arg 进行格式化的方法，但是仅返回格式化之后的字串，如代码 3.9 所示：

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

代码 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：满眼看不到副作用或者 var。能在任何可枚举的集合类型（包括数组，列表，集和映射）上调用的 mkString 方法，返回由每个数组元素调用 toString 产生结果组成的字串，以传入字串间隔。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 方法那样实际打印输出，但可以简单地把它的结果传递给 println 来实现：

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

每个有用的程序都可能有某种形式的副作用，因为否则就不可能对外部世界提供什么值。偏好于无副作用的方法可以鼓励你设计副作用代码最少化了的程序。这种方式的好处之一是可以有助于使你的程序更容易测试。举例来说，要测试本节之前给出三段 printArgs 方法的任一个，你将需要重定义 println，捕获传递给它的输出，并确信这是你希望的。相反，你可以通过检查结果来测试 formatArgs：

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

Scala 的 assert 方法检查传入的 Boolean 并且如果是假，抛出 AssertionError。如果传入的 Boolean 是真，assert 只是静静地返回。你将在第十四章学习更多关于断言和测试的东西。

虽如此说，不过请牢记在心：不管是 var 还是副作用都不是天生邪恶的。Scala 不是强迫你用函数式风格编任何东西的纯函数式语言。它是一种指令式/函数式混合的语言。你或许发现在某些情况下指令式风格更符合你手中的问题，在这时候你不应该对使用它犹豫不决。然而，为了帮助你学习如何不使用 var 编程，在第 7 章中我们会给你看许多有 var 的特殊代码例子和如何把这些 var 转换为 val。

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。只有在特定需要和判断之后才选择 var，可变对象和有副作用的方法。

第十二步：从文件里读取信息行

处理琐碎的，每日工作的脚本经常需要处理文件。本节中，你将建立一个从文件中读行记录，并把行中字符个数前置到每一行，打印输出的脚本。第一版展示在代码 3.10 中：

```
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")
```

代码 3.10　从文件中读入行

此脚本开始于从包 scala.io 引用名为 Source 的类。然后检查是否命令行里定义了至少一个参数。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一个 Source 对象，你在其上调用 getLines。函数返回 Iterator[String]，在每个枚举里提供一行包括行结束符的信息。for 表达式枚举这些行并打印每行的长度，空格和这行记录。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打印一条信息。如果你把这些代码放在文件 contchars1.scala，并运行它调用自己：

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

你会看到：

```
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")
```

尽管当前形式的脚本打印出了所需的信息，你或许希望能让数字右序排列，并加上管道符号，这样输出看上去就替换成：

```
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")
```

想要达到这一点，你可以对所有行枚举两次。第一次决定每行字符计数的最大宽度。第二次打印输出之前计算的最大宽度。因为要枚举两次，你最好把它们赋给变量：

```
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。一旦你使用它完成遍历，枚举器就失效了。而通过调用 toList 把它转换为 List，你就可以枚举任意次数，代价就是把文件中的所有行一次性贮存在内存里。lines 变量因此就指向着包含了命令行指定的文件文本字串的数组。

下一步，因为要对每行字符数计算两次，每个枚举计算一次，你或许会考虑把表达式拉出来变成一个小函数，专门用来计算传入字串的字符长度：

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

有了这个函数，你就可以计算最大长度了：

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

这里你用一个 for 表达式枚举了每一行，计算这些行的宽度，并且，如果比当前最大宽度还大，就把它赋值给 maxWidth，一个初始化为 0 的 var。（max 方法是你可以在任何 Int 上调用的，可以返回被调用者和被传入者中的较大的值。）如果你希望不用 var 发现最大值，替代的方法是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
)


val widths = lines.map(widthOfLength)
```

reduceLeft 方法把传入的方法应用于 lines 的前两个元素，然后再应用于第一次应用的结果和 lines 接下去的一个元素，等等，直至整个列表。每次这样的应用，结果将是碰到的最长一行，因为传入的函数，(a, b) => if (a.length > b.length) a else b，返回两个传入字串的最长那个。reduceLeft 将传回最后一次应用的结果，也就是本例 lines 中包含的最长字串。

得到这个结果之后，你可以通过把最长一行传给 widthOfLength 计算最大的宽度：

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

最后剩下的就是用一个合适的格式把这些行打印出来。你可以这么做：

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

在这个 for 表达式里，你再一次枚举了全部行记录。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 个空格的字串。最终，你打印出你想要格式的信息。全部的脚本展示在代码 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")
```

代码 3.11　对文件的每行记录打印格式化的字符数量

结语

有了本章获得的知识，你应该已经可以开始在小任务，尤其是脚本里使用 Scala。下一章里，我们将在这些话题上深入更多细节，并介绍其它这里没有提到过的话题。

第4章

类和对象

前面两章中，你已经看到 Scala 里的类和对象的基础。本章中，我们将带你更深入一些。你会学到更多关于类，字段和方法的东西，并浏览分号推断。你会学到更多关于单例对象的东西，包括如何使用他们编写和运行 Scala 程序。如果你熟悉 Java，你会发现 Scala 里的概念相似，但不完全相同。所以即使你是一位 Java 大师，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，你就可以用关键字 new 从类的蓝图里创建对象。比方说，如果给出了类的定义：

```
class ChecksumAccumulator {
  // class definition goes here
}
```

你就能创建 ChecksumAccumulator 对象：

```
new CheckSumAccumulator
```

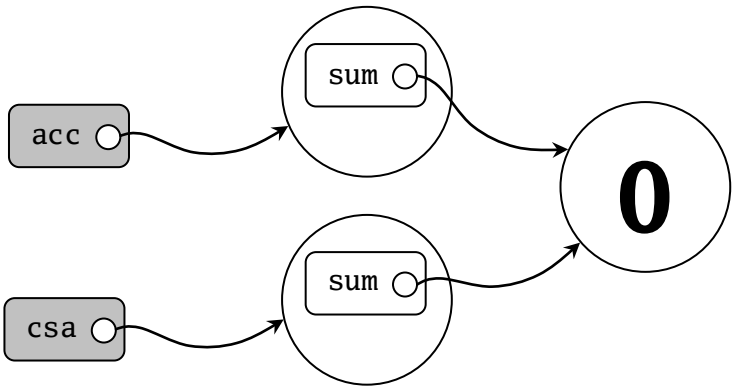
类定义里，可以放置字段和方法，这些被笼统地称为**成员**：*member*。字段，不管是用 val 或是用 var 定义的，都是指向对象的变量。方法，用 def 定义，包含了可执行的代码。字段保留了对象的状态或者数据，而方法使用这些数据对对象做运算工作。当你实例化类的时候，执行期环境会设定一些内存来保留对象状态的镜像——也就是说，变量的内容。举例来说，如果你定义了 ChecksumAccumulator 类并给它一个叫做 sum 的 var 字段：

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

并实例化两次：

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

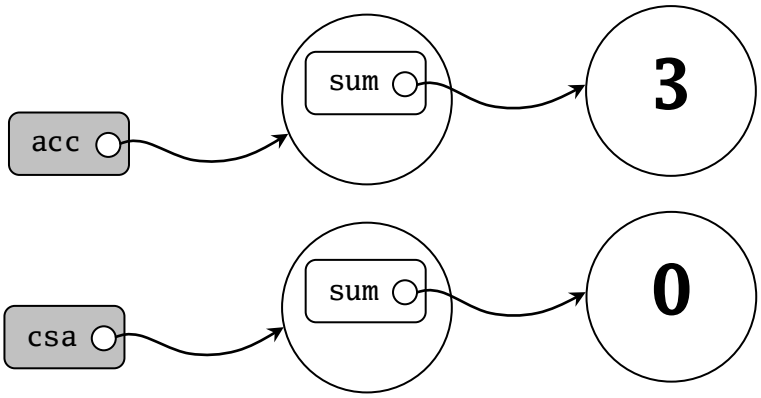
对象在内存里的镜像看上去大概是这样的：



由于在类 ChecksumAccumulator 里面定义的字段 sum 是 var，而不是 val，你之后可以重新赋值给它不同的 Int 值，如：

```
acc.sum = 3
```

现在，图像看上去会变成：



这张图里第一件要注意的事情是这里有两个 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镜像。你不仅可以因为看到两个 sum 变量来体会关于这个的演示，同样可以通过改变其中一个时，另一个不变来发现这点。

本例中另外一件需要注意的事情是，尽管 acc 是 val，你仍可以改变 acc 指向的对象。你对 acc（或 csa）不能做的事情是由于它们是 val，而不是 var，你不可以把它们再次赋值为不同的对象。例如，下面的尝试将会失败：

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

于是你可以总结出来，acc 将永远指向初始化时指向的同一个 ChecksumAccumulator 对象，但是包含于对象中的字段可以随时改动。

想让对象具有鲁棒性的一个重要的方法就是保证对象的状态——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* 去阻止外界直接对它的访问，因为私有字段只能被定义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，你在 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 的缺省访问级别。

现在 sum 是私有的，所以唯一能访问 sum 的代码是定义在类自己里面的。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，两个都以基本的的函数定义方式展示，参见第 40 页的图 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
}
```

尽管在这个 ChecksumAccumulator 版本里的 add 和 checksum 方法正确地实现了预期的功能，你还是可以用更简洁的风格表达它们。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个计算得到的值。

对于方法来说推荐的风格实际是避免显式的尤其是多个返回语句。代之以把每个方法当作是创建返回值的表达式。这种哲学将鼓励你制造很小的方法，把较大的方法分解为多个更小的方法。另一方面，设计选择取决于设计内容，Scala 使得编写具有多个，显式的 return 的方法变得容易，如果那的确是你期望的。

[1] 参数是 val 的理由是 val 更容易讲清楚。你不需要多看代码以确定是否 val 被重新赋值，而 var 则不然。

因为 checksum 要做的只有计算值，不需要 return。所以这个方法的另一种简写方式是，假如某个方法仅计算单个结果表达式，则可以去掉大括号。如果结果表达式很短，甚至可以把它放在 def 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
}
```

像 ChecksumAccumulator 的 add 方法那样的结果类型为 Unit 的方法，执行的目的就是它的副作用。通常我们定义副作用为在方法外部某处改变状态或者执行 I/O 活动。比方说，在 add 这个例子里，副作用就是 sum 被重新赋值了。表达这个方法的另一种方式是去掉结果类型和等号，把方法体放在大括号里。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。代码 4.1 的 add 方法里演示了这种风格：

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

代码 4.1 类 ChecksumAccumulator 的最终版

应该注意到令人困惑的地方是当你去掉方法体前面的等号时，它的结果类型将注定是 Unit。不论方法体里面包含什么都不例外，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，但方法的结果类型被声明为 Unit，那么 String 将被转变为 Unit 并失去它的值。下面是这个例子：

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

例子里，String 被转变为 Unit 因为 Unit 是函数 f 声明的结果类型。Scala 编译器会把一个以过程风格定义的方法，就是说，带有大括号但没有等号的，在本质上当作是显式定义结果类型为 Unit 的方法。例如：

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

因此，如果你本想返回一个非 Unit 的值，却忘记了等号时，那么困惑就出现了。所以为了得到你想要的结果，你需要插入等号：

```
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。如果你愿意可以输入一个，但若一行里仅有一个语句也可不写。另一方面，如果一行里写多个语句那么分号是需要的：

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

如果你想输入一个跨越多行的语句，多数时候你只需输入，Scala 将在正确的位置分隔语句。例如，下面的代码被认为是一个跨四行的语句：

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

然而，偶尔 Scala 也许没有按照你的愿望把句子分割成两部分：

```
x
+ y
```

这会被分成两个语句 x 和+ y。如果你希望把它作为一个语句 x + y，你可以把它包裹在括号里：

```
(x
+ y)
```

或者，你也可以把+放在行末。正是由于这个原因，当你在串接类似于+的中缀操作符，把操作符放在行尾而不是行头是普遍的 Scala 风格：

```
x +
y +
z
```

分号推断的规则

分割语句的精确规则非常有效却出人意料的简单。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字。
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

如第 1 章所提到的，Scala 比 Java 更面向对象的一个方面是 Scala 没有静态成员。替代品

是，Scala 有**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就像是类定义。代码 4.2 展示了一个例子：

```
// 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
    }
}
```

代码 4.2 类 ChecksumAccumulator 的伴生对象

表中的单例对象被叫做 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，他被称作是这个类的**伴生对象**：***companion object***。你必须在同一个源文件里定义类和它的伴生对象。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单例对象有一个方法，calculate，用来计算所带的String参数中字符的校验和。它还有一个私有字段，cache，一个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，看看是否传递进来的字串已经作为键存在于映射当中。如果是，就仅仅返回映射的值，“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并用新建的ChecksumAccumulator实例初始化它。[3]下一行是个for表达式，对传入字串的每个字符循环一次，并在其上调用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，下一行的方法在acc上调用checksum，获得传入字串的校验和，并存入叫做cs的val。下一行，“cache += (s -> cs)”，传入的字串键映射到整数的校验和值，并把这个键-值对加入cache映射。方法的最后一个表达式，“cs”，保证了校验和为此方法的结果。

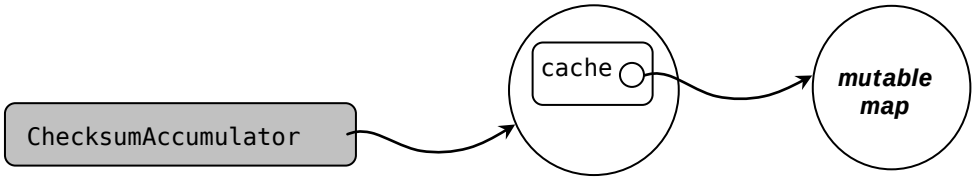
如果你是 Java 程序员，考虑单例对象的一种方式是把它当作是或许你在 Java 中写过的任何静态方法之家。可以在单例对象上用类似的语法调用方法：单例对象名，点，方法名。例如，可以如下方式调用 ChecksumAccumulator 单例对象的 calculate 方法：

[2] 这里我们使用了缓存例子来说明带有域的单例对象。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] 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。

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

然而单例对象不只是静态方法的收容站。它同样是个第一类的对象。因此你可以把单例对象的名字看作是贴在对象上的“名签”：



定义单例对象不是定义类型（在 Scala 的抽象层次上说）。如果只是 ChecksumAccumulator 对象的定义，你就建不了 ChecksumAccumulator 类型的变量。宁愿这么说，ChecksumAccumulator 类型是由单例对象的伴生类定义的。然而，单例对象扩展了超类并可以混入特质。由于每个单例对象都是超类的实例并混入了特质，你可以通过这些类型调用它的方法，用这些类型的变量指代它，并把它传递给需要这些类型的方法。我们将在第十二章展示一些继承自类和特质的单例对象的例子。

类和单例对象间的一个差别是，单例对象不带参数，而类可以。因为你不能用new关键字实例化一个单例对象，你没机会传递给它参数。每个单例对象都被作为由一个静态变量指向的**虚构类**：*synthetic class*的一个实例来实现，因此它们与Java静态类有着相同的初始化语法。[4]特别要指出的是，单例对象会在第一次被访问的时候初始化。

不与伴生类共享名称的单例对象被称为**孤立对象**：*standalone object*。由于很多种原因你会用到它，包括把相关的功能方法收集在一起，或定义一个 Scala 应用的入口点。下一段会说明这个用例。

4.4　Scala 程序

要执行 Scala 程序，你一定要提供一个有 main 方法（仅带一个参数，Array[String]，且结果类型为 Unit）的孤立单例对象名。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。代码 4.3 展示了一个例子：

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

代码 4.3 程序 Summer

代码 4.3 单例对象的名字是Summer。它的main方法具有合适的签名，所以你可以把它用作程序。文件中的第一个语句是引用定义在前例中ChecksumAccumulator对象中的calculate

[4] 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$。

方法。这个引用语句允许你在文件之后的部分里使用方法的简化名。[5]main方法体简单地打印输出每个参数和参数的校验和，用冒号分隔。

注意

Scala 隐式引用了包 java.lang 和 scala 的成员，和名为 Predef 的单例对象的成员。到每个 Scala 源文件中。Predef，被放置在包 scala 中，包含了许多有用的方法。例如，当在 Scala 源文件中写 println 的时候，你实际调用了 Predef 的 println。（Predef.println 运转并调用 Console.println，做实际的工作。）当你写 assert，你是在调用 Predef.assert。

要执行 Summer 应用程序，把代码 4.3 的代码放在文件 Summer.scala 中。因为 Summer 使用了 ChecksumAccumulator，把 ChecksumAccumulator 的代码，包括代码 4.1 的类和代码 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 需要你在跟着类命名的文件里放上一个公共类——如文件 SpeedRacer.java 里要放上类 SpeedRacer——Scala 里，你可以任意命名.scala 文件，而不用考虑里面放了什么 Scala 类或代码。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，这样程序员就可以通过查看文件名的方式更容易地找到类。这就是我们在本例中文件 ChecksumAccumulator.scala 和 Summer.scala 上使用的方式。

无论 ChecksumAccumulator.scala 还是 Summer.scala 都不是脚本，因为他们是以定义结束的。反过来说，脚本必然以一个结果表达式结束。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。其中一种方式是使用 scalac，Scala 的基本编译器。输入：

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

这将编译你的源文件，不过在编译完成之前或许会有一个可感知的停顿。原因是每次编译器启动时，都要花一些时间扫描 jar 文件内容，并在即使你提交的是新的源文件也在查看之前完成其他初始化工作。因此，Scala 的发布包里还包括了一个叫做 fsc（快速 Scala 编译器）的 Scala 编译器**后台服务**：***daemon***。你可以这样使用：

```
$ 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去编译，后台进程完成编译。下一次你执行 fsc 时，后台进程就已经在运行了，于是 fsc 将只是把文件列表发给后台进程，它会立刻开始编译文件。使用 fsc，你只需要在第一次等待 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 来关闭。

不论执行scalac还是fsc命令，都将创建Java类文件，然后你可以用scala命令，就像之前的例子里调用解释器那样运行它。不过，不是像前面每个例子里那样把包含了Scala代码的带有.scala扩展名的文件交给它解释执行，[6]在这里你要给它包含了正确签名的main方法

[5] 如果你是 Java 程序员，你可以认为这种引用类似于 Java 5 引入的精通引用特性。然而 Scala 里的一个不同是，你可以从任何对象引用成员，而不只是单例对象。

[6] 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。

的孤立对象名。因此，你可以这样运行Summer应用程序：

```
$ scala Summer of love
```

你会看到两个命令行参数的校验和被打印出来：

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

4.5　Application 特质

Scala 提供了一个特质，scala.Application，可以节省你一些手指的输入工作。尽管我们还没有完全提供给你去搞明白它如何工作的所有需要知道的东西，不过我们还是认为你可能想要知道它。代码 4.4 展示了一个例子：

```
import ChecksumAccumulator.calculate
object FallWinterSpringSummer extends Application {
  for (season <- List("fall", "winter", "spring"))
    println(season +": "+ calculate(season))
}
```

代码 4.4 使用 Application 特质

使用这个特质的方法是，首先在你的单例对象名后面写上“extends Application” 。然后代之以 main 方法，你可以把想要放在 main 方法里的代码直接放在单例对象的大括号之间。就这么简单。之后可以像对其它程序那样编译和运行。

这种方式之所以能奏效是因为特质 Application 声明了带有合适的签名的 main 方法，并由你的单例对象继承，使它可以像个 Scala 程序那样用。大括号之间的代码被收集进了单例对象的**主构造器**：*primary constructor*，并在类被初始化时被执行。如果你不明白所有这些指的是什么也不用着急。之后的章节会解释这些，目前可以暂时不求甚解。

继承自 Application 比写个显式的 main 方法要短，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，所以它必须带有显式的 main 方法，如代码 4.3 所示。第二，因为某些 JVM 线程模型里的局限，如果你的程序是多线程的就需要显式的 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。因此只有当你的程序相对简单和单线程情况下你才可以继承 Application 特质。

4.6　结语

本章向你介绍了 Scala 里的类和对象的基础，并向你展示了如何编译和运行程序。下一章，你会学到 Scala 的基本类型及如何使用它们。

第5章

基本类型和操作

现在你已经看到了真正的类和对象，正是在更深层次看看 Scala 的基本类型和操作的好时候。如果你熟悉 Java，你会很开心地发现 Java 基本类型和操作符在 Scala 里有同样的意思。然而即使你是一位资深 Java 开发者，这里也仍然有一些有趣的差别使得本章值得一读。因为本章提到的一些 Scala 的方面实质上与 Java 相同，我们插入了一些注释，Java 开发者可以安全跳过，以加快你的进程。

本章里，你会获得 Scala 基本类型的概观，包括 `String` 和值类型 `Int`，`Long`，`Short`，`Byte`，`Float`，`Double`，`Char` 还有 `Boolean`。你会学到可以在这些类型上执行的操作，包括 Scala 表达式里的操作符优先级是如何工作的。你还会学到隐式转换是如何“丰富”这些基本类型的变体，并带给你那些由 Java 提供支持之外的附加操作。

5.1 一些基本类型

表格 5.1 显示了 Scala 的许多基本的类型和其实例值域范围。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***。

表格 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`	字符序列
`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，你可以在任何地方只用简化名（就是说，像`Boolean`，或`Char`，或`String`这样的名字 。

注意

目前实际上你可以使用与 Java 的原始类型相一致的 Scala 值类型的小写化名。比如，Scala 程序里你可以用 `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

[1] 包的概念在第 2 章的第二步作了简要说明，将在第十三章做更深入介绍。

再支持乃至移除小写变体，因此跟随社区的大流，在你的 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。这让 Scala 编译器能直接把 Scala 的**值类型**：***value type*** 实例，如 Int 或 Double，在它产生的字节码里转译成 Java 原始类型。

5.2　文本

所有在表 5.1 里列出的基本类型都可以写成**文本**：***literal***。文本是直接在代码里写常量值的一种方式。

Java 程序员的快速通道

本节里多数文本的语法和在 Java 里完全一致，因此如果你是一位 Java 大师，你可以安心地跳过本节的多数内容。你应该看得两个差异分别是 Scala 的原字串和符号文本，将在第 73 页描述。

整数文本

类型 Int，Long，Short 和 Byte 的整数文本有三种格式：十进制，十六进制和八进制。整数文本的开头方式说明了数字的基。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，不论你用什么形式的整数文本初始化，Scala 的 shell 始终打印输出基于 10 的整数值。因此解释器会把你用文本 0x00FF 初始化的 hex2 变量的值显示为十进制的 255。当然，你也可以不采信我们的话。开始感受语言的好方法是你一边读本章的时候一边在解释器里试试这些语句。）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
```

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
```

如果整数文本结束于 L 或者 l，就是 Long 类型，否则就是 Int 类型。一些 Long 类型的整数文本有：

```
scala> val prog = 0XCAFEBABEL
prog: Long = 3405691582
scala> val tower = 35L
tower: Long = 35
scala> val of = 31l
of: Long = 31
```

如果 Int 类型的文本被赋值给 Short 或者 Byte 类型的变量，文本就会被看作是能让文本值在那个类型有效范围内那么长的 Short 或者 Byte 类型。如：

```
scala> val little: Short = 367
little: Short = 367
scala> val littler: Byte = 38
littler: Byte = 38
```

浮点数文本

浮点数文本是由十进制数字，可选的小数点和可选的 E 或 e 及指数部分组成的。下面是一些浮点数文本的例子：

```
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 为底的幂次数。因此，1.2345e1 就是 1.2345 乘以 10^1，等于 12.345。如果浮点数文本以F或f结束，就是Float类型的，否则就是Double类型的。可选的，Double浮点数文本也可以D或d结尾。Float文本举例如下：

```
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
```

字符文本

字符文本可以是在单引号之间的任何 Unicode 字符，如：

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

除了在单引号之间显式地提供字符之外，你还可以提供一个表示字符代码点的前缀反斜杠的八进制或者十六进制数字。八进制数必须在'\0'和'\377'之间。例如字母 A 的 Unicode 字符代码点是八进制 101。因此：

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

字符文本同样可以以前缀\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，上面代码里的两个 unicode 字符扩展之后的结果。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非 ASCII 的 Unicode 字符的 Scala 源文件用 ASCII 来代表。

表格 5.2 特殊字符文本转义序列

文本	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）

最终，还有一些字符文本被表示成特殊的转义序列，参见表格 5.2。例如：

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

字串文本

字串文本由双引号（"）环绕的字符组成：

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

引号内的字符语法与字符文本相同，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串很笨拙。因此 Scala 为**原始字串**：*raw String* 引入了一种特殊的语法。以同一行里的三个引号（"""）开始和结束一条原始字串。内部的原始字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，下面的程序使用了原始字串打印输出一条消息：

```
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串里。为了解决这个常见情况，字串类引入了 `stripMargin` 方法。使用的方式是，把管道符号（|）放在每行前面，然后在整个字串上调用 `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.
```

符号文本

符号文本被写成'*<标识符>*，这里*<标识符>*可以是任何字母或数字的标识符。这种文本被映射成预定义类 `scala.Symbol` 的实例。特别是，文本`'cymbal` 将被编译器扩展为工厂方法调用：`Symbol("cymbal")`。符号文本典型的应用场景是你在动态类型语言中使用一个标识符。比方说，或许想要定义个更新数据库记录的方法：

```
scala>  def updateRecordByName(r: Symbol, value: Any) {
          // code goes here
        }
updateRecordByName: (Symbol,Any)Unit
```

方法带了一个符号参数指明记录的字段名和一个字段应该更新进记录的值。在动态类型语

言中，你可以通过传入一个未声明的字段标识符给方法调用这个操作，但 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")
```

基本同样简洁的替代方案是，你可以传递一个符号文本：

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

除了发现它的名字之外，没有太多能对符号做的事情：

```
scala> val s = 'aSymbol
s: Symbol = 'aSymbol
scala> s.name
res20: String = aSymbol
```

另一件值得注意的事情是符号是被**拘禁**：***interned*** 的。如果你把同一个符号文本写两次，那么两个表达式将指向同一个 Symbol 对象。

布尔型文本

布尔类型有两个文本，true 和 false：

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

就这些东西了。现在你简直（literally）[2]可以称为Scala的专家了。

5.3　操作符和方法

Scala 为它的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作用在普通方法调用上华丽的语法。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，它带一个 Int 参数并返回一个 Int 结果。这个+方法在两个 Int 相加时被调用：

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

想要证实这点，可以把表达式显式地写成方法调用：

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

而真正的事实是，Int包含了许多带不同的参数类型的**重载**：***overload***的+方法。[3]例如，Int

[2] 象征意义的说法。（似乎是双关语）

还有另一个也叫+的方法参数和返回类型为Long。如果你把Long加到Int上，这个替换的+方法就将被调用：

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

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像操作符一样的东西。你可以把**任何**方法都当作操作符来标注。例如，类 String 有一个方法 indexOf 带一个 Char 参数。indexOf 方法搜索 String 里第一次出现的指定字符，并返回它的索引或-1 如果没有找到。你可以把 indexOf 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 提供一个重载的 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。不过当你用操作符标注方式调用带多个参数的方法时，这些参数必须放在括号内。例如，以下是如何把另一种形式的 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言语法：任何方法都可以是操作符。**使用**方法的方式使它成为操作符。如果写成 s.indexOf('o')，indexOf 就不是操作符。不过如果写成，s indexOf 'o'，那么 indexOf 就是操作符了，因为你以操作符标注方式使用它。

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：前缀和后缀。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——操作符带后两个操作数，一个在左一个在右——相反，前缀和后缀操作符都是**一元**：*unary* 的：它们仅带一个操作数。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。与中缀操作符一致，这些前缀操作符是在值类型对象上调用方法的简写方式。然而这种情况下，方法名在操作符字符上前缀了“unary_”。例如，Scala 会把表达式-2.0 转换成方法调用“(2.0).unary_-”。你可以输入通过操作符和显式方法名两种方式对方法的调用来演示这一点：

```
scala> -2.0           // Scala调用了(2.0).unary_-
res2: Double = -2.0
scala> (2.0).unary_-
res3: Double = -2.0
```

[3] 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。

可以当作前缀操作符用的标识符只有+，-，!和~。因此，如果你定义了名为unary_!的方法，就可以像!p这样在合适的类型值或变量上用前缀操作符方式调用这个方法。但是如果你定义了名为unary_*的方法，就没办法用成前缀操作符了，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，这或许就不会是你想当然的了！[4]

后缀操作符是不用点或括号调用的不带任何参数的方法。Scala 里，你可以舍弃方法调用的空括号。例外就是如果方法带有副作用就加上括号，如 println()，不过如果方法没有副作用就可以去掉括号，如 String 上调用的 toLowerCase：

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

后面的这个例子里，方法没带参数，或者还可以去掉点，采用后缀操作符标注方式：

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

例子里，toLowerCase 被当作操作数 s 上的后缀操作符。

因此要想知道 Scala 的值类型里你可以用哪些操作符，所有需要做的就是在 Scala 的 API 文档里查询定义在值类型上的方法。不过由于本书是 Scala 的教程，我们会在后续几段里带您快速浏览这些方法中的大部分。

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。如果你是一个匆忙的 Java 牛人，你可以安心地跳到第 79 页的 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

你可以通过中缀操作符，加号（+），减号（-），乘号（*），除号（/）和余数（%），在任何数类型上调用数学方法。以下是一些例子：

```
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
```

[4] 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。

```
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。如果你的确想要 IEEE754 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前缀+和-操作符（方法 unary_+和 unary_-），允许你指示文本数是正的还是负的，如-3 或+4.0。如果你没有指定一元的+或-，文本数被解释为正的。一元符号+也存在只是为了与一元符号-相协调，不过没有任何效果。一元符号-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

你可以用关系方法：大于（>），小于（<），大于等于（>=）和小于等于（<=）比较数类型，像等号操作符那样，产生一个 Boolean 结果。另外，你可以使用一元操作符!（unary_!方法）改变 Boolean 值。以下是一些例子：

```
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（||），以中缀方式带 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最少能决定结果的部分。换句话说，逻辑与和逻辑或表达式的右手侧部分在左手侧部分能决定结果时就不再被评估了。举个例子，如果逻辑与表达式的左手侧计算结果为 false，那么表达式的结果将注定是 false，因此右手侧部分不再做评估。与之类似，如果逻辑或表达式的左手侧部分计算结果为 true，那么表达式的结果将必然是 true，于是右手侧部分不再被计算。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，进入方法之前所有的参数都会被评估，因此方法怎么可能选择不评估他的第二个参数呢？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设置。这个设置被称为**叫名参数**：*by-name parameter*，将在 9.5 节中讨论。

5.6　位操作符

Scala让你能够使用若干位方法对整数类型的单个位执行操作。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：

[5] 按位异或方法对它的操作数执行**互斥或**：*exclusive or* 操作。一致的位产生 0。差异的位产生 1。因此 0011 ^ 0101 产生 0110。

```
scala> 1 & 2
res24: Int = 0
scala> 1 | 2
res25: Int = 3
scala> 1 ^ 3
res26: Int = 2
scala> ~1
res27: Int = -2
```

第一个表达式，1&2，与运算了 1（0001）和 2（0010）的每一个位，并产生了 0（0000）。第二个表达式，1 | 2，对同样的操作数的每一个位执行或运算，并产生 3（0011）。第三个表达式，1 ^ 3，异或 1（0001）和 3（0011）的每一个位，产生 2（0010）。最后的表达式，~1，转换 了 1（0001）的每一个位，产生了-2，二进制看起来是 1111 1111 1111 1111 1111 1111 1111 1110。

Scala 整数类型还提供了三个位移方法：左移（<<），右移（>>）和无符号右移（>>>）。使用在中缀操作符方式时，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
res38: Int = -1
scala> -1 >>> 31
res39: Int = 1
scala> 1 << 2
res40: 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

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

```
scala> 1 == 2
res24: Boolean = false
scala> 1 != 2
res25: Boolean = true
```

[6] 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> 2 == 2
res26: 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，你可以用他比较列表：

```
scala> List(1, 2, 3) == List(1, 2, 3)
res27: Boolean = true
scala> List(1, 2, 3) == List(4, 5, 6)
res28: Boolean = false
```

进一步，你还可以比较不同类型的两个对象：

```
scala> 1 == 1.0
res29: Boolean = true
scala> List(1, 2, 3) == "hello"
res30: Boolean = false
```

你甚至可以比较 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
res31: Boolean = false
scala> null == List(1, 2, 3)
res32: Boolean = false
```

如你所见，==已经被仔细地加工过，因此在许多情况下你都可以得到你想要的相等性的比较。这只是用了一个非常简单的规则：首先检查左侧是否为null，如果不是，调用equals方法。由于equals是一个方法，因此比较的精度取决于左手边的参数。又由于已经有一个自动的null检查，因此你不需要手动再检查一次了。[7]

这种类型的比较对于不同的对象也会产生 true，只要他们的内容是相同的并且它们的 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串的比较：

```
scala> ("he" + "llo") == "hello"
res33: Boolean = true
```

Scala 的==与 Java 的有何差别

Java 里的既可以比较原始类型也可以比较参考类型。对于原始类型，Java 的==比较值的相等性，如 Scala。然而对于参考类型，Java 的==比较了**参考相等性**：*reference equality*，也就是说这两个变量是否都指向于 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 的全部细节将在 11.1 节和 11.2 节给出。还有，可以看一下第二十八章，了解如何编写好的 equals 方法。

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

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。当然你还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，你可以这么写表达式：

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

由于 Scala 没有操作符，实际上，是以操作符的格式使用方法的一个途径，你或许想知道操作符优先级是怎么做到的。Scala 基于操作符格式里方法的第一个字符决定优先级（这个规则有一个例外，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。

表格 5.3 操作符优先级

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

表格 5.3 以降序方式展示了根据方法第一个字符指定的优先级，同一行的字符具有同样的优先级。表格中字符的位置越高，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<，在表格 5.3 里的位置比+（+方法的第一个也是唯一的一个字符）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。

上面提到的优先级规则的一个例外，有关于以等号结束的**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肩并肩地出现在表达式里，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 49 页提到的，任何以 ‘:’ 字符结尾的方法由它的右手侧操作数调用，并传入左操作数。以其他字符结尾的方法有其他的说法。它们都是被左操作数调用，并传入右操作数。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 是一个表达式而不仅仅是一个不可变值的指针的话，那么更精确的意义上说，a:::b 将会当作是：

```
{ 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减少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!”[8]

5.9　富包装器

你可以在 Scala 基本类型上调用的方法远多于前几段里面讲到过的。表格 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***，一种将在第二十一章描述其细节的技术。现在所有要知道的就是本章介绍过的每个基本类型，都有一个“富包装器”可以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，你还应该查看一下每个基本类型的富包装器的 API 文档。这些类可参见表格 5.5。

表格 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

[8] 到目前为止，你应该能指出给出的这段代码，Scala 编译器会调用成(bills.!*&^%~(code)).!()。

```
(1.0 / 0) isInfinity        true
4 to 6                      Range(4, 5, 6)
"bob" capitalize            "Bob"
"robert" drop 2             "bert"
```

表格 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　结语

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，我们会告诉你用能带给那些看过的操作符新的实现的函数式风格来设计对象到底是什么意思。

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，没有任何可变状态的对象的类。作为运行的例子，我们将创造若干把分数作为不可变对象建模的类的变体。在这过程中，我们会展示给你 Scala 面向对象编程的更多方面：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 的式样书

分数：*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 被称作是**分子**：*numerator*，d 被称作是**分母**：*denominator*。分数的例子有：$\frac{1}{2}$，$\frac{2}{3}$，$\frac{112}{239}$ 和 $\frac{2}{1}$。与浮点数相比较，分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模型化分数的行为，包括允许它们执行加，减，乘还有除运算。要加两个分数，首先要获得公分母，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，$\frac{7}{6}$。要乘两个分数，可以简单的两个分子相乘，然后两个分母相乘。因此，$\frac{1}{2}\times\frac{2}{5}$ 得到了 $\frac{2}{10}$，还可以简化表示成它的“通常”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 $\frac{5}{6}$。

一个，或许不怎么重要的，发现是数学上，分数不具有可变的状态。一个分数加到另外一个分数上，产生的结果是一个新的分数。而原来的数不会被“改变”。我们将在本章设计的不可变的 `Rational` 类将秉承这一属性。每个分数将都被表示成一个 `Rational` 对象。当两个 `Rational` 对象相加时，一个新的带着累加结果的 `Rational` 对象将被创建出来。

本章还将捎带提一些 Scala 让你写出感觉像原生语言支持的库的方法。例如，在本章结尾你将能用 `Rational` 类这样做：

```
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

开始设计 Rational 类的着手点是考虑客户程序员将如何创建一个新的 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大括号（当然你如果想的话也可以）。在类名，Rational，之后括号里的 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一个带同样的两个参数的**主构造器**：***primary constructor***。

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具逻辑性，因为它们没有随着时间而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，而或许需要在把可变对象传递给其它代码之前，需要先建造个以防万一的副本。第三，没有机会能让两个同时访问不可变对象的线程破坏它合理构造的状态，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，如果可变对象在被放进了 HashSet 之后被改变，那么你下一次查找这个 HashSet 就找不到这个对象了。

不可变对象唯一的缺点就是它们有时需要复制很大的对象图而可变对象的更新可以在原地发生。有些情况下这会变得难以快速完成而可能产生性能瓶颈。结果，要求库提供可变替代以使其更容易在大数据结构的中间改变一些元素也并非是一件稀奇的事情。例如，类 StringBuilder 是不可变的 String 的可变替代。第十八章中我们会给出更多 Scala 里设计可变对象的细节。

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。这可以潜在地节省很多固定写法，尤其是对小类来说。

Scala 编译器将把你放在类内部的任何不是字段的部分或者方法定义的代码，编译进主构造器。例如，你可以像这样打印输出一条除错消息：

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

根据这个代码，Scala 编译器将把 println 调用放在 Rational 的主构造器。因此，println 调用将在每次创建一个新的 Rational 实例时打印这条除错信息：

```
scala> new Rational(1, 2)
```

```
Created 1/2
res0: Rational = Rational@a0b0f5
```

6.3　重新实现 toString 方法

前例中当 Rational 实例被创建之后，解释器打印输出“Rational@a0b0f5”。解释器是通过调用 Rational 对象的 toString 方法获得的这个看上去有些好玩儿的字串。缺省情况下，Rational 类继承了定义在 java.lang.Object 类上的 toString 实现，只是打印类名，一个@符号和一个十六进制数。toString 的结果主要是想通过提供可以用在除错时的语句打印，日志消息，测试错误报告和解释器，除错器输出的信息来尝试对程序员提供帮助。目前 toString 提供的结果不会特别有用，因为它没有给出任何它被调用的 Rational 数值的任何线索。更有用的 toString 实现应该打印出 Rational 的分子和分母。你可以通过在 Rational 类里增加 toString 方法的方式**重载**：*override* 缺省的实现，如：

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

方法定义前的 override 修饰符标示了之前的方法定义被重载；第 10 章会更进一步说明。现在分数显示得很漂亮了，所以我们去掉了前一个版本的 Rational 类里面的 println 除错语句。你可以在解释器里测试 Rational 的新行为：

```
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。如本章早些时候提到的，分数的分母不能为零。然而目前主构造器会接受把零传递给 d：

```
scala> new Rational(5, 0)
res6: Rational = 5/0
```

面向对象编程的一个优点就是它允许你把数据封装在对象之内以便于你确保数据在整个生命周期中是有效的。像 Rational 这样的不可变对象，这就意味着你必须确保在对象创建的时候数据是有效的（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。

解决这个问题的最好办法是为主构造器定义一个**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：

[1] require 方法定义在 scala 包里的孤立对象 Predef 上。4.4 节中提到过，Predef 对象的成员都已被自动

```
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 将通过抛出 IllegalArgumentException 来阻止对象被构造。

6.5　添加字段

现在主构造器可以正确地执行先决条件，我们将把注意力集中到支持加法。想做到这点，我们将在类 Rational 上定义一个公开的 add 方法，它带另一个 Rational 做参数。为了保持 Rational 不可变，add 方法必须不能把传入的分数加到自己身上。而是必须创建并返回一个全新的带有累加值的 Rational。你或许想你可以这么写 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)
}
```

很不幸，上面的代码会让编译器提示说：

```
<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都在你的add代码可引用的范围内，但是在调用add的对象中仅能访问它们的值。因此，当你在add的实现里讲n或d的时候，编译器将很高兴地提供给你这些类参数的值。但绝对不会让你使用that.n或that.d，因为that并不指向add被调用的Rational对象。[2]要想访问that的n和d，需要把它们放在字段中。代码 6.1 展示了如何把这些字段加入类Rational。[3]

在代码 6.1 展示的Rational版本里，我们增加了两个字段，分别是numer和denom，并用类参数n和d初始化它们。[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。类Rational的这个版本能够编译通过，可以通过分数的加法测试它：

引入到每个 Scala 源文件中。

[2] 实际上，在 that 指的是调用 add 的对象时， Rational 可以加到自己身上。但是因为你可以传递任何 Rational 对象给 add，所以编译器仍然不会让你说 that.n。

[3] 10.6 节中你会发现**参数化域**：*parametric field*，提供了编写同样代码的捷径。

[4] 尽管 n 和 d 是用在类的函数体内，因为他们只是用在构造器之内，Scala 编译器将不会为它们自动构造域。所以就这些代码来说，Scala 编译器将产生一个有两个 Int 域的类，一个是 numer，另一个是 denom。

```
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
    )
}
```

代码 6.1 带字段的 Rational

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2
scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3
scala> oneHalf add twoThirds
res0: Rational = 7/6
```

另一件之前不能而现在可以做的事是在对象外面访问分子和分母。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
res7: Int = 1
scala> r.denom
res8: 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，我们考虑添加一个方法，lessThan，来测试给定的分数是否小于传入的参数：

```
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 前缀而只是写 numer；着两种写法是相同的。

举一个不能缺少 this 的例子，考虑在 Rational 类里添加 max 方法返回指定分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 是冗余的，你写成(lessThan(that))也是一样的。但第二个 this 表示了当测试为假的时候的方法的结果；如果你省略它，就什么都返回不了了。

6.7　从构造器

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**从构造器**：***auxiliary constructor***。比方说，分母为 1 的分数只写分子的话就更为简洁。如，对于 $\frac{5}{1}$ 来说，可以只是写成 5。因此，如果不是写成 Rational(5, 1)，客户程序员简单地写成 Rational(5)或许会更好看一些。这就需要给 Rational 添加一个只带一个参数，分子，的从构造器并预先设定分母为 1。代码 6.2 展示了应该有的样子。

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1)
  override def toString = numer+"/"+denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

代码 6.2 带有从构造器的 Rational

Scala 的从构造器开始于 def this(...)。Rational 的从构造器主体几乎完全是调用主构造器，直接传递了它的唯一的参数，n，作为分子和 1 作为分母。输入下列代码到解释器里可以实际看到从构造器的效果：

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

Scala 里的每一个从构造器的第一个动作都是调用同一个类里面其他的构造器。换句话说就是，每个 Scala 类里的每个从构造器都是以“this(...)”形式开头的。被调用的构造器既可以是主构造器（好像 Rational 这个例子），也可以是从文本上来看早于调用构造器的其它从构造器。这个规则的根本结果就是每一个 Scala 的构造器调用终将结束于对类的主构造器的调用。因此主构造器是类的唯一入口点。

注意

若你熟悉 Java，你或许会奇怪为什么 Scala 构造器的规矩比 Java 的还要大。Java 里，构造器的第一个动作必须要么调用同类里的另一个构造器，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 里更严格的限制实际上是权衡了更高的简洁度和与 Java 构造器相比的简易性所付出的代价之后作出的设计。超类，构造器调用和继承交互的细节将在第 10 章里解释。

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它所需要的要大。例如分数 $\frac{66}{42}$，可以更约简化为相同的最简形式，$\frac{11}{7}$，但 Rational 的主构造器当前并不做这个工作：

```
scala> new Rational(66, 42)
res15: Rational = 66/42
```

要想对分数进行约简化，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 是能够除尽 66 和 42 的最大的整数。）$\frac{66}{42}$ 的分子和分母都除以 6 就产生它的最简形式，$\frac{11}{7}$。代码 6.3 展示了如何做到这点：

```
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)
}
```

代码 6.3 带私有字段和方法的 Rational

这个版本的 Rational 里，我们添加了私有字段，g，并修改了 numer 和 denom 的初始化器（初始化器：initializer 是初始化变量，例如初始化 numer 的“n / g”，的代码）。因为 g 是私有的，它只能在类的主体之内，而不能在外部被访问。我们还添加了一个私有方法，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。正如你在 4.1 节中看到的，想让一个字段或方法私有化你只要把 private 关键字放在定义的前面。私有的“助手方法”gcd 的目的是把类的其它部分，这里是主构造器，需要的代码分离出来。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代码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 都将被构造成它的最简形式：

```
scala> new Rational(66, 42)
res24: Rational = 11/7
```

6.9　定义操作符

Rational 加法的当前实现仅就完成功能来讲是没问题的，但它可以做得更好用。你或许会问你自己为什么对于整数或浮点数你可以写成：

```
x + y
```

但是如果是分数就必须写成：

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

或至少是：

```
x add y
```

没有合理的解释为什么就必须是这样的。分数和别的数应该是一样的。数学的角度上看他们甚至比，唔，浮点数，更自然。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。既然已经到这儿了，你可以同样实现一个*方法以实现乘法，结果展示在代码 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)
}
```

代码 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
res32: Rational = 7/6
```

与以往一样，在最后输入的那行里的语法格式相等于一个方法调用。你也能这么写：

```
scala> x.+(y)
res33: Rational = 7/6
```

不过这样写可读性不佳。

另外一件要提的是基于 5.8 节中提到的 Scala 的操作符优先级规则，Rational 里面的*方法要比+方法绑定得更结实。或者说，Rational 涉及到+和*操作的表达式会按照预期的方式那样表现。例如，x + x * y 会当作 x + (x * y)而不是(x + x) * y：

```
scala> x + x * y
res34: Rational = 5/6
scala> (x + x) * y
res35: Rational = 2/3
scala> x + (x * y)
res36: Rational = 5/6
```

6.10　Scala 的标识符

现在你已经看到了 Scala 里两种构成标识符的方式：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它的两种。本节将说明所有的这四种标识符构成方式。

字母数字标识符： ***alphanumeric identifier*** 起始于一个字母或下划线，之后可以跟字母，数字，或下划线。‘$’字符也被当作是字母，但是被保留作为 Scala 编译器产生的标识符之用。用户程序里的标识符不应该包含‘$’字符，尽管能够编译通过；但是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]标识符习俗，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代码里有许多其它非标识符用法。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

标识符结尾使用下划线的一个结果就是，比如说，如果你尝试写一个这样的定义，“val name_: Int = 1”，你会收到一个编译器错误。编译器会认为你正常是定义一个叫做“name_:”的变量。要让它编译通过，你将需要在冒号之前插入一个额外的空格，如：“val name_ : Int = 1”。

[5] 这种风格被称为**驼峰式：*camel case***，因为标识符由一个个首字母大写的内嵌单词组成。

[6] 在 16.5 节里，你将看到有些时候你需要给一种被称为**用例类：*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。

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，但它还是个变量。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，表示圆周和它的直径的比值。这个值不太可能改变，因此 `Pi` 显然是个常量。你还可以用常数去给一些你代码里作为**幻数**：***magic number*** 要用到的值一个名字：文本值不具备解释能力，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，用例将在 15.2 节中说明。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，就应使用这个内部表达。

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`。如果你想要得到第一种解释，你要在‘`<`’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 里写 `x<-y` 的时候不注意加空格或括号的。

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。例如，`unary_+`被用做定义一元的‘`+`’操作符的方法名。或者，`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；第十八章进一步说明。

文本标识符：***literal identifier*** 是用反引号`` `...` ``包括的任意字串。如：

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

它的思路是你可以把任何运行时认可的字串放在反引号之间当作标识符。结果总是 Scala 标识符。即使包含在反引号间的名称是 Scala 保留字，这个规则也是有效的。在 Java 的 `Thread` 类中访问静态的 `yield` 方法是其典型的用例。你不能写 `Thread.yield()`因为 `yield` 是 Scala 的保留字。然而，你仍可以在反引号里引用方法的名称，例如 `Thread.`yield`()`。

6.11　方法重载

回到类 `Rational` 上来。在最近一次改变之后，你可以在分数上用自然的风格做加法和乘法。但别忘了还有混合运算。例如，你不能把一个分数和一个整数乘在一起，因为‘`*`’的操作数只能是分数。所以对于分数 `r` 你不能写 `r * 2`。而必须写成 `r * new Rational(2)`，

[7] 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，或不是字母，数字，括号，方括号，大括号，单或双引号，或下划线，句号，分号，冒号，回退字符的 7 位 ASCII 字符。

看上去不漂亮。为了让 Rational 用起来更方便，可以在类上增加能够执行分数和整数之间的加法和乘法的新方法。既然已经到这里了，还可以再加上减法和除法。结果展示在代码 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)
}
```

代码 6.5 含有重载方法的 Rational

现在每种数学方法都有两个版本了：一个带分数做参数，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，+这个名字被一个带 Rational 的和另一个带 Int 的方法使用。方法调用里，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。如果你尝试输入：

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

```
x: Rational = 2/3
scala> x * x
res37: Rational = 4/9
scala> x * 2
res38: Rational = 4/3
```

你会看到*方法的调用取决于每个例子里面右侧操作数的类型。

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。有时如果不止一个最符合的版本；这种情况下编译器会给你一个“参考模糊”的错误。

6.12　隐式转换

现在你能写 `r * 2` 了，或许你想交换操作数，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)
  val res2 = 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。

然而，Scala 里有另外一种方法解决这个问题：你可以创建一个在需要的时候能自动把整数转换为分数的隐式转换。试着把这行代码加入到解释器：

```
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` 修饰符告诉编译器若干情况下自动调用它。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
res0: Rational = 4/3
```

请注意隐式转换要起作用，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。现在，你要在解释器内直接定义它。

正如你在这个例子中能领略到的，隐式转换是把库变得更灵活和更方便的非常强大的技术。因为他们如此强大，所以也很容易被误用。第二十一章里你将发现隐式转换的更多细节，包括在需要的时候把它们带入作用范围的方式。

译者注：

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。

6.13　一句警告

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，不过请牢记能力带来的责任。

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，对于客户程序员来说哪个隐式转换被应用了并非显而易见。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，但不能简洁的过了头。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　结语

本节中，你看到了 Scala 类中更多的元素。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。

尽管本章中显示的 `Rational` 最终版本满足了开始时候的需求集，它仍有改善的空间。实际上当你学了一定的知识能把 `Rational` 变得更好的时候，我们将在后续的书里回到这个例子上。例如，在第二十八章，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。在第二十一章，你会学到如何把隐式方法定义放在 `Rational` 的伴生对象中，这样当客户程序员在使用 `Rational` 的时候就可以更容易地把它们放在作用范围中。

第7章

内建控制结构

Scala 里没有多少内建控制结构。仅有的包括 `if`，`while`，`for`，`try`，`match` 和函数调用。如此之少的理由是，从一开始 Scala 就包括了函数文本。代之以在基本语法之上一个接一个添加高层级控制结构，Scala 把它们汇集在库里。第 9 章将更细致地展现如何做到这点。本章将展现仅有的几个内建控制结构。

有件你会注意到的事情是，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，因此程序的控件也应当这么做。你也可以把这种方式看做早已存在于指令式语言中的一种趋势（函数调用返回值，被调用函数更新被当作参数传入的输出变量也归于此类）的逻辑推演。另外，指令式语言经常具有三元操作符（如 C，C++和 Java 的`?:`操作符），表现得就像 `if`，却产生值。Scala 采用了这种三元操作符模型，但是把它称为 `if`。换句话说，Scala 的 `if` 可以产生值。于是 Scala 持续了这种趋势让 `for`，`try` 和 `match` 也产生值。

程序员能够利用这些结果值简化他们的代码，就好象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计算的结果。去掉这些临时变量能让代码更简洁并避免许多你在一个分支里设置了变量却在另外一个分支里忘了设置的 bug。

总而言之，Scala 的基础控制结构虽然少但也足够提供指令式语言里重要的东西了。进一步说，由于都具有结果值，它们能帮助你缩短代码。为了让你看到所有这些都是怎么工作的，本章将近距离展现 Scala 的基础控制结构。

7.1 if 表达式

Scala 的 if 如同许多其它语言中的一样工作。它测试一个状态并据其是否为真，执行两个分支中的一个。下面是一个常见的例子，以指令式风格编写：

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

这段代码声明了一个变量，`filename`，并初始化为缺省值。然后使用 `if` 表达式检查是否提供给程序了任何参数。如果是，就把变量改成定义在参数列表中的值。如果没有参数，就任由变量设定为缺省值。

这段代码可以写得更好一点，因为就像第 2 章第三步提到过的，Scala 的 `if` 是能返回值的表达式。代码 7.1 展示了如何不使用任何 `var` 而实现前面一个例子同样的效果：

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

代码 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，那么初始化元素，args(0)，被选中。否则，缺省值被选中。这个 if 表达式产生了被选中的值，然后 filename 变量被初始化为这个值。这段代码更短一点儿，不过它的实际优点在于使用了 val 而不是 var。使用 val 是函数式的风格，并能以差不多与 Java 的 final 变量同样的方式帮到你。它让代码的读者确信这个变量将永不改变，节省了他们扫描变量字段的所有代码以检查它是否改变的工作。

使用 val 而不是 var 的第二点好处是他能更好地支持**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)，你可以这么写：

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

选择权在你。怎么写都行。使用 val 可以帮你安全地执行这类重构以不断革新你的代码。

> 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 循环表现的和在其它语言中一样。包括一个状态和循环体，只要状态为真，循环体就一遍遍被执行。代码 7.2 展示了一个例子：

```
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
}
```

代码 7.2 用 while 循环计算最大公约数

Scala 也有 do-while 循环。除了把状态测试从前面移到后面之外，与 while 循环没有区别。代码 7.3 展示了使用 do-while 反馈从标准输入读入的行记录直到读入空行为止的 Scala 脚本：

```
var line = ""
do {
  line = readLine()
  println("Read: " + line)
} while (line != null)
```

代码 7.3 用 do-while 从标准输入读取信息

while 和 do-while 结构被称为“循环”，不是表达式，因为它们不产生有意义的结果，结果的类型是 Unit。说明产生的值（并且实际上是唯一的值）的类型为 Unit。被称为 ***unit value***，写做 ()。()的存在是 Scala 的 Unit 不同于 Java 的 void 的地方。请在解释器里尝试下列代码：

```
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，产生 true。

另一个产生 unit 值的与此相关的架构，是对 var 的再赋值。比如，假设尝试用下面的从 Java（或者 C 或 C++）里的 while 循环成例在 Scala 里读取一行记录，你就遇到麻烦了：

```
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 的值将永远产生 true。而在 Java 里，赋值语句可以返回被赋予的那个值，同样情况下标准输入返回的一条记录在 Scala 的赋值语句中永远产生 unit 值，()。因此，赋值语句“line = readLine()”的值将永远是()而不是""。结果，这个 while 循环的状态将永远不会是假，于是循环将因此永远不会结束。

由于 while 循环不产生值，它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。虽然如此，Scala 仍然包含了 while 循环，因为有些时候指令式的解决方案更可读，尤其是对那些以指令式背景为主导的程序员来说。例如，如果你想做一段重复某进程直到某些状态改变的算法代码，while 循环可以直接地表达而函数式的替代者，大概要用递归实现，或许对某些代码的读者来说就不是那么显而易见的了。

如，代码 7.4 展示了计算两个数的最大公约数的替代方式。[1]给定同样的值x和y，代码 7.4 展示的gcd函数将返回与代码 7.2 中gcdLoop函数同样的结果。这两种方式的不同在于gcdLoop写成了指令式风格，使用了var和while循环，而gcd更函数式风格，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)
```

代码 7.4 使用递归计算最大公约数

通常意义上，我们建议你如质疑 var 那样质疑你代码中的 while 循环。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，为了让你的程序有任何改变，while 循环通常不是更新 var 就是执行 I/O。可以在之前的 gcdLoop 例子里看到。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对 while 循环抱有更怀疑的态度。如果没有对特定的 while 或 do 循环较好的决断，请尝试找到不用它们也能做同样事情的方式。

7.3　for 表达式

Scala 的 for 表达式是为枚举准备的“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来表达各种各样的枚举。简单的用法完成如把整数序列枚举一遍那样通常的任务。更高级的表

[1] 代码 7.4 中展示的 gcd 函数使用了首先在代码 6.3 中展示的同名函数，为类 Rational 计算最大公约数，所使用的同样的方法，主要的差别在于代码 7.4 的 gcd 的参数使用 Long 而不是 Int。

达式可以列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

你能用 for 做的最简单的事情就是把一个集合类的所有元素都枚举一遍。如，代码 7.5 展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，".", 的文件，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)
```

代码 7.5 用 for 循环列表目录中的文件

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。编译器推断 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 方法产生文件或目录的名称，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]第 82 页的表格 5-4 中看到的Range类型是其中一个方便的特例，你可以使用类似于“1 to 5”这样的语法创建一个Range，然后用for枚举。以下是一个简单的例子：

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

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

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

像这样枚举整数在 Scala 里是很平常的，但在其他语言中就不是这么回事。其它语言中，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

[2] 更精确地说，在<-符号右侧的表达式必须支持名为 foreach 的方法。

的每个设置执行一次 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。

这种类型的枚举在 Scala 里不常见的原因是直接枚举集合类也做得同样好。这样做，你的代码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***。该从 0 开始还是从 1 开始？应该加-1，+1，还是什么都不用直到最后一个索引？这些问题很容易回答，但也很容易答错。还是避免碰到为佳。

过滤

有些时候你不想枚举一个集合类的全部元素。而是想过滤出一个子集。你可以通过把**过滤器：*filter***：一个 if 子句加到 for 的括号里做到。如代码 7.6 的代码仅对当前目录中以“.scala”结尾的文件名做列表：

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere if file.getName.endsWith(".scala"))
  println(file)
```

代码 7.6 用带过滤器的 for 发现.scala 文件

或者你也可以这么写：

```
for (file <- filesHere)
  if (file.getName.endsWith(".scala"))
    println(file)
```

这段代码可以产生与前一段代码同样的输出，而且对于指令式背景的程序员来说看上去更熟悉一些。然而指令式格式只是一个可选项，因为这个 for 表达式的运用执行的目的是为了它的打印这个副作用并产生 unit 值()。正如在本节后面将展示的，for 表达式之所以被称为“表达式”是因为它能产生令人感兴趣的值，一个其类型取决于 for 表达式<-子句的集合。

如果愿意的话，你可以包含更多的过滤器。只要不断加到子句里即可。例如，为了加强防卫，代码 7.7 中的代码仅仅打印文件而不是目录。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)
```

代码 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。这是代码 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，代码 7.8 展示的 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.*")
```

代码 7.8 在 for 表达式中使用多个发生器

如果愿意的话，你可以使用大括号代替小括号环绕发生器和过滤器。使用大括号的一个好处是你可以省略一些使用小括号必须加的分号。

mid-stream（流间）变量绑定

请注意前面的代码段中重复出现的表达式 `line.trim`。这不是个可忽略的计算，因此你或许想每次只算一遍。通过用等号（=）把结果绑定到新变量可以做到这点。绑定的变量被当作 `val` 引入和使用，不过不用带关键字 `val`。代码 7.9 展示了一个例子。

```
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.*")
```

代码 7.9 在 for 表达式里的流间赋值

代码中，名为 `trimmed` 的变量被从半当中引入 `for` 表达式，并被初始化为 `line.trim` 的结果值。之后的 `for` 表达式就可以在两个地方使用这个新变量，一次在 `if` 中，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放过，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制造一个值，本例中是 file。当 for 表达式完成的时候，结果将是一个包含了所有产生的值的集合。结果集合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。即使循环体是一个被大括号包围的代码块，也一定把 yield 放在左括号之前，而不是代码块的最后一个表达式之前。请抵挡住写成如下方式的诱惑：

```
for (file <-filesHere if file.getName.endsWith(".scala")) {
  yield file // 语法错误！
}
```

例如，代码 7.10 展示的 for 表达式首先把包含了所有当前目录的文件的名为 filesHere 的 Array[File]，转换成一个仅包含 .scala 文件的数组。对于每一个对象，产生一个 Iterator[String]（fileLines 方法的结果，定义展示在代码 7.8 中），提供方法 next 和 hasNext 让你枚举集合的每个元素。这个原始的枚举器又被转换为另一个 Iterator[String]仅包含含有子字串"for"的修剪过的行。最终，对每一行产生整数长度。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
```

代码 7.10　用 for 把 Array[File]转换为 Array[Int]

目前，你已经看过了 Scala 的 for 表达式所有主要的特征。然而这个段落过得实在是快了一些。for 表达式更透彻的介绍在第二十三章。

7.4　使用 try 表达式处理异常

Scala 的异常和许多其它语言的一样。代之用普通方式那样返回一个值，方法可以通过抛出一个异常中止。方法的调用者要么可以捕获并处理这个异常，或者也可以简单地中止掉，并把异常升级到调用者的调用者。异常可以就这么升级，一层层释放调用堆栈，直到某个方法处理了它或没有剩下其它的方法。

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，Scala 里， throw 也是有结果类型的表达式。下面举一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")
```

这里发生的事情是，如果 n 是偶数，half 将被初始化为 n 的一半。如果 n 不是偶数，那么在 half 能被初始化为任何值之前异常将被抛出。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。任何使用从 throw 返回值的尝试都不会起作用，因此这样做无害。

从技术角度上来说，抛出异常的类型是 Nothing。尽管 throw 不实际得出任何值，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 类型将在以后的 11.3 节中讨论。

捕获异常

用来捕获异常的语法展示在代码 7.11 中。选择 catch 子句这样的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十五章详述。

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
try {
  val f = new FileReader("input.txt")
  // Use and close file
} catch {
  case ex: FileNotFoundException => // Handle missing file
  case ex: IOException => // Handle other I/O error
}
```

代码 7.11　Scala 的 try-catch 子句

这个 try-catch 表达式的行为与其它语言中的异常处理一致。程序体被执行，如果抛出异常，每个 catch 子句依次被尝试。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 的一个差别是 Scala 里不需要你捕获**检查异常**：*checked exception*，或把它们声明在 throws 子句中。如果你愿意，可以用 AThrows 标注声明一个 throws 子句，但这不是必需的。

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。如，你或许想让打开的文件即使是方法抛出异常退出也要确保被关闭。代码 7.12 展示了这个例子。

```
import java.io.FileReader
val file = openFile()
try {
  // 使用文件
} finally {
  file.close() // 确保关闭文件
}
```

代码 7.12　Scala 的 try-finally 子句

注意

代码 7.12 展示了确保非内存资源，如文件，套接字，或数据库链接被关闭的惯例方式。首先你获得了资源。然后你开始一个 try 代码块使用资源。最后，你在 finally 代码块中关闭资源。这种 Scala 里的惯例与在 Java 里的一样，然而，Scala 里你还使用另一种被称为贷出模式：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

和其它大多数 Scala 控制结构一样，try-catch-finally 也产生值。如，代码 7.13 展示了如何尝试拆分 URL，但如果 URL 格式错误就使用缺省值。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，如果有的话，被抛弃。通常 finally 子句做一些清理类型的工作如关闭文件；他们不应该改变在主函数体或 try 的 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.scalalang.org")
  }
```

代码 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 的差别仅源于 Java 的 try-finally 不产生值。Java 里，如果 finally 子句包含一个显式返回语句，或抛出一个异常，这个返回值或异常将“凌驾”于任何之前源于 try 代码块或某个它的 catch 子句产生的值或异常之上。如：

```
def f(): Int = try { return 1 } finally { return 2 }
```

调用 f()产生结果值 2。相反：

```
def g(): Int = try { 1 } finally { 2 }
```

调用 g()产生 1。这两个例子展示了有可能另大多数程序员感到惊奇的行为，因此通常最好还是避免从 finally 子句中返回值。最好是把 finally 子句当作确保某些副作用，如关闭打开的文件，发生的途径。

7.5　match 表达式

Scala 的匹配表达式允许你在许多**可选项：*alternative*** 中做选择，就好象其它语言中的 switch 语句。通常说来 match 表达式可以让你使用任意的**模式：*pattern*** 做选择，第十五章会介绍。通用的模式可以稍等再说。目前，只要考虑使用 match 在若干可选项中做选择。

作为例子，代码 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。如果是字串"salt"，就打印"pepper"，如果是"chips"，就打印"salsa"，如此递推。缺省情况用下划线（_）说明，这是常用在 Scala 里作为占位符表示完全不清楚的值的通配符。

```
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?")
}
```

代码 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。其中之一是任何种类的常量，或其他什么东西，都能用作 Scala 里的 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，可选项是字串。另一个区别是在每个可选项的最后并没有 break。取而代之，break 是隐含的，不会有从一个可选项转到另一个里面去的情况。这通常把代码变短了，并且避免了一些错误的根源，因为程序员不再因为疏忽在选项里转来转去。

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。在前一个例子里，match 表达式的每个可选项打印输出一个值。只生成值而不是打印也可以一样做到，展示在代码 7.15 中。match 表达式产生的值储存在 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），还解开了两个不相干的关注点：首先选择食物名，其次打印它。

```
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)
```

代码 7.15　生成值的 match 表达式

7.6　离开 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他们与函数式文本，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，但是在函数式文本中表示什么呢？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从而换得在语言上的简洁性。尽管如此，请不要着急。有许多不用 break 和 continue 的编程方式，如果你能有效利用函数式文本，就能比原来的代码写得更短。

最简单的方式是用 if 替换每个 every 和用布尔变量替换每个 break。布尔变量指代是否包含它的 while 循环应该继续。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串。Java 里你可以——如果你很喜欢 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;
}
```

如果要字面直译成 Scala 的代码，代之以执行一个 if 然后 continue，你可以写一个 if 环绕 while 余下的全部内容。要去掉 break，你可以增加一个布尔变量提示是否继续做下去，不过在这里你可以复用 foundIt。使用这两个技巧，代码就可以像代码 7.16 这样完成了：

```
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
}
```

代码 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。循环内有个 i 是否小于 args.length 的测试，然后检查"-"，然后检查".scala"。

如果要去掉代码 7.16 里面的 var，你可以尝试的一种方式是用递归函数重写循环。比方说，你可以定义带一个整数值做输入的 searchFrom 函数，向前搜索，并返回想要的参数的索引。采用这种技巧的代码看上去会像展示在代码 7.17 中这样的：

```
def searchFrom(i: Int): Int =
  if (i >= args.length) -1// 不要越过最后一个参数
  else if (args(i).startsWith("-")) searchFrom(i + 1)// 跳过选项
  else if (args(i).endsWith(".scala")) i // 找到!
  else searchFrom(i + 1) // 继续找
val i = searchFrom(0)
```

代码 7.17　不用 var 做循环的递归替代方法

代码 7.17 的版本提供了一个能够看得懂的名字说明这个函数在做什么，它用递归替代了循环。每个 continue 都被带有 i + 1 做参数的递归调用替换掉，有效地跳转到下一个整数。许多人都发现当他们开始使用递归后，这种编程风格更易于理解。

注意

Scala 编译器不会实际对代码 7.17 展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

现在你已经看过了 Scala 的内建控制结构，我们将在本节中使用它们来解释 Scala 里的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 允许你在嵌套范围内定义同名变量。因此如果你是 Java 程序员，或许至少还是快速浏览一下。

Scala程序里的变量定义有一个能够使用的**范围**：***scope***。范围设定的最普通不过的例子就是，大括号通常引入了一个新的范围，所以任何定义在打括号里的东西在括号之后就脱离了范围。[3]作为演示，请看一下代码 7.18 里展示的函数：

```
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
```

[3] 这条规则有几个例外，因为在 Scala 里有时候你可以用大括号代替小括号。表达式语法的替代品是这种使用大括号例子的其中之一，将在 7.3 节描述。

```
      // 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
}
```

代码 7.18　打印乘法表时的变量范围

printMultiTable函数打印了乘法表。[4]函数的第一个语句引入了变量i并初始化为整数 1。然后你可以在函数余下的部分里使用名称i。

printMultiTable 接下去的语句是一个 while 循环：

```
while (i <= 10) {
  var j = 1
  ...
}
```

你可以在这使用 i 因为它仍在范围内。在 while 循环的第一个语句里，你引入了另一个变量，叫做 j，并再次初始化为 1。因为变量 j 定义在 while 循环的大括号内，所以只能用在 while 循环里。如果你想尝试在 while 循环的大括号之后，在那个说 j，prod 和 k 已经出了范围的注释后面，再用 j 做点儿什么事，你的程序就编译不过了。

本例中定义的所有变量——i，j，prod 和 k——都是**本地变量**：***local variable***。对于它们被定义的函数来说是“本地”的。每次函数被调用的时候，一整套全新的本地变量将被使用。

一旦变量被定义了，你就不可以在同一个范围内定义同样的名字。比如，下面的脚本不会被编译通过：

```
val a = 1
val a = 2 // 编译不过
println(a)
```

然而，你可以在一个内部范围内定义与外部范围里名称相同的变量。下列脚本将编译通过并可以运行：

```
val a = 1;
{
  val a = 2 // 编译通过
```

[4] 代码 7.18 展示的 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。

```
  println(a)
}
println(a)
```

执行时，这个脚本会先打印 2，然后打印 1，因为定义在内部打括号里的a是不同的变量，将仅在大括号内部有效。[5]Scala和Java间要注意的一个不同是，与Scala不同，Java不允许你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，内部变量被说成是**遮蔽**：***shadow***了同名的外部变量，因为在内部范围内外部变量变得不可见了。

或许你已经注意到了一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，你可以对你的核心内容重用变量名。撇开别的不说，这样能允许你当发现你在解释器里第一次定义变量时犯了错误的时候改变主意。你能这么做的理由是因为，在理论上，解释器在每次你输入新的语句时都创建了一个新的嵌套范围。因此，你可以把之前解释的代码虚拟化认为是：

```
val a = 1;
{
  var a = 2;
  {
    println(a)
  }
}
```

这段代码可以像 Scala 脚本那样编译和执行，而且像输入到解释器里的代码那样，打印输出 2。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。通常更好的办法是选择一个新的有意义的变量名而不是遮蔽外部变量。

7.8　重构指令式风格的代码

为了帮助你在函数式风格上获得更多的领悟，本节我们将重构代码 7.18 中以指令式风格打印乘法表的方式。我们的函数式替代品展示在代码 7.19 中。

代码 7.18 中的代码在两个方面显示出了指令式风格。首先，调用 `printMultiTable` 有副作用：在标准输出上打印乘法表。代码 7.19 中，我们重构了函数，让它把乘法表作为字串返回。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一个优点是它们很容易进行单元测试。要测试 `printMultiTable`，你需要重定义 `print` 和 `println` 从而能够检查输出的正确性。测试 `multiTable` 就简单多了，只要检查结果即可。

[5] 另外，本例中在 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。

```
// 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
  }
// 以字串形式返回一行乘法表
def makeRow(row: Int) = makeRowSeq(row).mkString
// 以字串形式返回乘法表，每行记录占一行字串
def multiTable() = {
  val tableSeq = // 行记录字串的序列
    for (row <- 1 to 10)
      yield makeRow(row)
  tableSeq.mkString("\n")
}
```

代码 7.19　创建乘法表的函数式方法

printMultiTable 里另一个揭露其指令式风格的信号来自于它的 while 循环和 var。与之相对，multiTable 函数使用了 val，for 表达式，**帮助函数**：*helper function*，并调用了 mkString。

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 for 表达式从 1 到 10 枚举列数。这个 for 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，乘积字串叠加成的结果。for 表达式的结果是一个包含了这些生成字串作为元素的序列（scala.Seq 的某个子类）。另一个帮助函数，makeRow，仅仅调用了 makeRowSeq 返回结果的 mkString 函数。叠加序列中的字串把它们作为一个字串返回。

multiTable 方法首先使用一个 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 获得该行的字串。因为字串前缀 yield 关键字，所以表达式的结果就是行字串的序列。现在仅剩下的工作就是把字串序列转变为单一字串。mkString 的调用完成这个工作，并且由于我们传递进去"\n"，因此每个字串结尾插入了换行符。如果把 multiTable 返回的字串传递给 println，你将看到与调用 printMultiTable 所生成的同样的输出结果：（略）

7.9　结语

Scala 的内建控制结构少到极点，但足够需要。它们表现得如同它们的指令式的对应，但因为它们倾向于产生值，所以它们也支持函数式风格。同样重要的是，它们很仔细地舍去一些东西，因此保留了空间给 Scala 最强大的特征之一，函数式文本，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，你需要一些方法把它们分割成更小的，更易管理的片段。为了分割控制流，Scala 提供了所有有经验的程序员都熟悉的方式：把代码分割成函数。实际上，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，函数文本和函数值。本章带你体会所有 Scala 中的这些函数的风味。

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，代码 8.1 展示了两个可以合作根据一个给定的文件名读文件并打印输出所有长度超过给定宽度的行的方法。每个打印输出的行前缀它出现的文件名：

```
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)
  }
}
```

代码 8.1 带私有的 processLine 方法的 LongLines 对象

processFile 方法带了 filename 和 width 做参数。它用文件名创建了一个 Source 对象并，在 for 表达式的发生器中，对 source 调用 getLines。第 3 章的第十二步曾经提到，getLines 返回一个枚举器，能在每一次枚举中从文件里取出一行，包括换行符。for 表达式通过调用帮助方法，processLine，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，跟着一个冒号，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]

```
object FindLongLines {
  def main(args: Array[String]) {
```

[1] 本书中，我们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，也是为了减少会模糊例子重点部分的套路代码。作为交换就是，当输入错误时，代之以产生有助的错误信息，我们的例子程序将抛出异常。

```
    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) {
```

到此为止，这些与你能用面向对象语言做的很像。然而 Scala 里的函数概念比方法更宽泛。Scala 描述函数的另外的方法将在后续节中描述。

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一个重要设计原则：程序应该被解构成若干小的函数，每个完成一个定义良好的任务。单个函数经常很小。这种风格的好处是它给了程序员许多可以灵活组装成更复杂事物的建造模块。每个小块应该充分简化到足以单独理解。

这种方式的一个问题是所有这些帮助函数的名称会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，就最好对类的客户隐藏帮助函数。它们经常不能独立表达什么意思，并且如果之后用其它方式重写类的话，也常会想保持能删掉帮助方法的足够的灵活度。

Java 里，达成这个目的的主要工具是 private 方法。这种私有方法的方式在 Scala 里同样有效，如代码 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另一个函数中。就好象本地变量那样，这种本地函数仅在包含它的代码块中可见。以下是一个例子：

```
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)
  }
}
```

在这个例子中，我们通过把私有方法，processLine，转换为本地方法，processFile，重构了展示在代码 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，另一个改善变为可能了。请注意 filename 和 width 是怎样不改变地传入到帮助函数中。这不是必须的，因为本地函数可以访问包含它们的函数的参数。你可以直接使用外部 processLine 函数的参数，如代码 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)
  }
}
```

代码 8.2 带本地 processLine 方法的 LongLines

更简单了，不是吗？这种对外层函数的参数的使用是 Scala 提供的通用嵌套的很平常也很有用的例子。7.7 节描述的嵌套和作用域应用于所有的 Scala 架构，包括函数。这是一个简单的原则，不过非常强大，尤其在拥有函数作为第一类值的语言中。

8.3　函数是第一类值

Scala 拥有**第一类函数**：*first-class function*。你不仅可以定义函数和调用它们，还可以把函数写成没有名字的**文本**：*literal* 并把它们像**值**：*value* 那样传递。我们在第 2 章介绍了函数文本并在第 44 页的图 2.2 里展示了基本语法。

函数文本被编译进一个类，类在运行期实例化的时候是一个**函数值**：*function value*。[2]因此函数文本和值的区别在于函数文本存在于源代码，而函数值存在于运行期对象。这个区别很像类（源代码）和对象（运行期）的那样。

以下是对数执行递增操作的函数文本的简单例子：

```
(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，这是一个把任何整数 x 映射为 x + 1 的函数。

函数值是对象，所以如果你愿意可以把它们存入变量。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。以下是这两种动作的例子：

```
scala> var increase = (x: Int) => x + 1
increase: (Int) => Int = <function>
scala> increase(10)
res0: Int = 11
```

本例中，因为 increase 是 var，你可以在之后重新赋给它不同的函数值。

```
scala> increase = (x: Int) => x + 9999
```

[2] 任何函数值都是某个扩展了若干 scala 包的 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。

```
increase: (Int) => Int = <function>
scala> increase(10)
res2: Int = 10009
```

如果你想在函数文本中包括超过一个语句，用大括号包住函数体，一行放一个语句，就组成了一个代码块。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产生的那个表达式。

```
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文本和函数值。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]它带一个函数做参数，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
```

另一个例子是，集合类型还有 `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。测试是通过函数的使用来提供的。例如，函数`(x: Int) => x > 0` 可以被用作过滤。这个函数映射正整数为真，其它为假。下面说明如何把它用作 `filter`：

```
scala> someNumbers.filter((x: Int) => x > 0)
res6: List[Int] = List(5, 10)
```

像 `foreach` 和 `filter` 这样的方法将在本书后面描述。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] `foreach` 方法被定义在特质 `Iterable` 中，它是 `List`，`Set`，`Array`，还有 `Map` 的共有超特质。参见第 17 章相关细节。

8.4　函数文本的短格式

Scala 提供了许多方法去除冗余信息并把函数文本写得更简短。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。

一种让函数文本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文本开始，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编译器能或不能解开谜题有感觉了。

第二种去除无用字符的方式是省略类型是被推断的参数之外的括号。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

如果想让函数文本更简洁，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文本内仅出现一次。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 115 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，函数文本_ > 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时，编译器有可能没有足够的信息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
```

请注意_ + _将扩展成带两个参数的函数文本。这也是仅当每个参数在函数文本中最多出现一次的情况下你才能使用这种短格式的原因。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　偏应用函数

尽管前面的例子里下划线替代的只是单个参数，你还可以使用一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**偏应用函数**：*partially applied function*。Scala 里，当你调用函数，传入任何需要的参数，你就是在把函数**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
```

偏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，要创建不提供任何三个所需参数的调用 sum 的偏应用表达式，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 编译器以偏应用函数表达式，sum _，实例化一个带三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三个参数之上时，它就转回头调用 sum，并传入这三个参数：

```
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编译器依照偏应用函数表达式sum _，自动产生的类的一个实例。编译器产生的类有一个apply方法带三个参数。[4]之所以带三个参数是因为sum _表达式缺少的参数数量为三。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，传入三个参数 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，简单地把这三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”在 apply 方法具有同样的参数列表和结果类型的函数值中。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它方法，但是如果你把方法或嵌套函数通过在名称后面加一个下划线的方式包装在函数值中，就可以做到了。

现在，尽管 sum _确实是一个偏应用函数，或许对你来说为什么这么称呼并不是很明显。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应用函数。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应用程序表达式，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 115 页）的这种写法：

```
someNumbers.foreach(println _)
```

你可以只是写成：

```
someNumbers.foreach(println)
```

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

[4] 产生的类扩展了特质 Function3，定义了三个参数的 apply 方法。

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 的偏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，偏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偏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)方法如 tail()，那么你会忘记你需要传递给 drop 一个数字。你或许会写，“println(drop)”。如果 Scala 采用偏应用函数在哪儿都 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‘_’。Scala 允许你仅在需要函数类型的地方才能省略这个仅用的_。

8.7　闭包

到本章这里，所有函数文本的例子仅参考了传入的参数。例如，(x: Int) => x > 0 里，函数体用到的唯一变量，x > 0，是 x，被定义为函数参数。然而也可以参考定义在其它地方的变量：

```
(x: Int) => x + more // more 是多少？
```

函数把“more”加入参考，但什么是 more 呢？从这个函数的视点来看，more 是个**自由变量**：*free variable*，因为函数文本自身没有给出其含义。相对的，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，一个 Int。如果你尝试独立使用这个函数文本，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 的什么东西同样的函数文本将工作正常：

```
scala> var more = 1
more: Int = 1
scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function>
scala> addMore(10)
res19: Int = 11
```

依照这个函数文本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”自由变量的绑定对函数文本执行的“关闭”行动。不带自由变量的函数文本，如(x: Int) => x + 1，被称为**封闭术语**：***closed term***，这里**术语**：***term*** 指的是一小部分源代码。因此依照这个函数文本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文本，如(x: Int) => x + more，都是**开放术语**：***open term***。因此，任何依照(x: Int) => x + more 在运行期创建的函数值将必须捕获它的自由变量，more，的绑定。由于函数值是关闭这个开放术语(x: Int) => x + more 的行动的最终产物，得到的函数值将包含一个指向捕获的 more 变量的参考，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，答案是闭包看到了这个变化。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
```

例子用了一个循环的方式计算 List 的累加和。变量 sum 处于函数文本 sum += _的外围，函数文本把数累加到 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会怎样？例如，如果闭包使用了某个函数的本地变量，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？

仅有一个答案与语言余下的部分共存：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：

[5] 相对的，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。

```
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 是一个已经返回的方法调用的参数也没有区别。Scala 编译器在这种情况下重新安排了它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。这种重新安排的工作都是自动关照的，因此你不需要操心。请任意捕获你想要的变量：val，var，或参数。

8.8　重复参数

Scala 允许你指明函数的最后一个参数可以是重复的。这可以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，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

在 7.2 节中，我们提到过想要把更新 var 的 while 循环转换成仅使用 val 的更函数式风格的话，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。但是否指令式的方式或许会更有效率呢？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，因为递归调用看上去比简单的从循环结尾跳到开头要更花时间。

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。

道义上你不应羞于使用递归算法去解决你的问题。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。

跟踪尾递归函数

尾递归函数将不会为每个调用制造新的堆栈框架；所有的调用将在一个框架内执行。这可能会让

检查程序的堆栈跟踪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一个异常：

```
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)
```

你会得到：

```
scala> bang(5)
java.lang.Exception: bang!
       at .bang(<console>:5)
       at .<init>(<console>:6)
...
```

这回，你仅看到了 bang 的一个堆栈框架。或许你会认为 bang 在调用自己之前就崩溃了，但这不是事实。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了：

```
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事三个 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: ifeq 12
    10: dload_1
    11: dreturn
    12: aload_0
    13: dload_1
    14: invokevirtual #27; //Method improve:(D)D
    17: dstore_1
    18: goto 2
```

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，尾调用优化受限于方法或嵌套函数在最后一个操

作调用本身，而没有转到某个函数值或什么其它的中间函数的情况。（如果你还不能完全明白尾递归，参见 8.9 节）。

8.10　结语

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，函数文本，及函数值。除了普通的函数调用之外，Scala 还提供了偏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的如手动优化版本的 while 循环一样快。下一章将建立在这些函数的基础上并展示 Scala 对函数丰富的支持能够如何帮助你在控制上实现抽象。

第9章

控制抽象

第 7 章里，我们指出 Scala 没有太多的内建控制抽象，因为它提供给你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。本章中，我们会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 curry 化和叫名参数。

9.1 减少代码重复

所有的函数都被分割成通用部分，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它算法。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***——带其它函数做参数的函数——给了你额外的机会去组织和简化代码。

高阶函数的一个好处是它们能让你创造控制抽象从而使你减少代码重复。例如，假设你正在写一个文件浏览器，并且你想要提供一个 API，能够允许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 方法是定义在你提供给你用户的 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-pub-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
```

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。那么你该做什么呢？

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(_))
```

这个例子中展示的函数文本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 方法里的函数文本_.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，不过你不需要指定参数类

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，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展示在代码 9.1 中那样。

```
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))
}
```

代码 9.1 使用闭包减少代码重复

这个例子演示了函数作为第一类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文本，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。相对的，最近的例子里面用到的函数文本_.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的时候帮助减少代码重复。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]很多已经在第三章的表格 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。

[1] 这些特定用途循环方法被定义在特质 Iterable 中，被 List，Set，Array，还有 Map 扩展。参见第 17 章的讨论。

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，你就可以这样调用：

```
scala> containsNeg(List(1, 2, 3, 4))
res0: Boolean = false
scala> containsNeg(List(1, 2, 3, -4))
res1: Boolean = true
```

不过更简洁的定义这个方法的方式是通过在传入的 List 上调用高阶函数 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：

```
def containsOdd(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num % 2 == 1)
      exists = true
  exists
}
```

若你比较了 containsNeg 和 containsOdd 的函数体，你会发现除了 if 表达式之外，其它东西都是重复的。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　Curry 化

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，不过任何人都不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 ***curry 化***的函数式编程技巧。

curry 化的函数被应用了多个参数列表，而不是仅仅一个。代码 9.2 展示了一个规整的，未被 curry 化的函数，它实现两个 Int 型参数，x 和 y 的加法。

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (Int,Int)Int
scala> plainOldSum(1, 2)
res4: Int = 3
```

代码 9.2 定义和调用“陈旧的”函数

相对的，代码 9.3 展示了 curry 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
```

代码 9.3 定义和调用 curry 化的函数

这里发生的事情是当你调用 curriedSum，你实际上背靠背地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：

```
scala> second(2)
res6: Int = 3
```

first 和 second 函数只是 curry 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。你可以用偏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参数加一并返回结果：

```
scala> onePlus(2)
res7: Int = 3
```

然后以下是你如何获得对唯一的 Int 参数加二函数的方式：

```
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

拥有第一类函数的语言中，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
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。

任何时候你发现你的代码中多个地方有重复的控制模式，你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()
  }
}
```

[2] 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。

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 而不是用户的代码，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**贷出模式**：***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大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，就能可选地使用大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，你使用了大括号替代小括号包围 println 的参数。然而，这个大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，当你尝试用大括号保卫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大括号替代小括号的机制的目的是让客户程序员能写出包围在大括号内的函数文本。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，因此你不能使用大括号。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 curry 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。代码 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()
  }
}
```

代码 9.4 使用贷出模式写文件

新的版本不同于旧版本的地方仅在于现在它有两个参数列表每个里面有一个参数替代了原来的一个参数列表里面有两个参数。仅比较这两个参数的差异。展示在第 131 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)
}
```

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，被包围在大括号中。

9.5　叫名参数：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，在于大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，根本没有值要传入大括号之间的代码，那该怎么做呢？为了解决这种情况，Scala 提供了叫名参数。

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函数将带一个函数值做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true。如果标志位被关闭了，myAssert将安静地什么都不做。

如果没有叫名参数，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：

[3] 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。

```
myAssert(() => 5 > 3)
```

你或许很想省略函数文本里的空参数列表和=>符号，写成如下形式：

```
myAssert(5 > 3) // 不会有效，因为缺少() =>
```

叫名函数恰好为了实现你的愿望而出现。要实现一个叫名函数，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 参数改为叫名参数。代码 9.5 展示了它的样子：

```
def byNameAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

代码 9.5 使用叫名参数

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)
```

叫名类型中，空的参数列表，()，被省略，它仅在参数中被允许。没有什么叫名变量或叫名字段这样的东西。

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>)...
```

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　结语

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 curry 化和叫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。

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 Scala 一些基本面向对象的方面。本章将拾起第 6 章省略的部分并更加深入到 Scala 的面向对象编程的细节中去。我们将比较类的两个基础关系：组合与继承。组合意味着一个类持有另一个的参考，使用参考类帮助实现任务。继承是超类/子类的关系。除此之外，我们还将讨论抽象类，无参数方法，扩展类，重载方法和字段，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

作为本章运行的例子，我们将创造一个制造和渲染二维布局元素的库。每个元素将代表一个填充字符的长方形。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串的元素：

```
elem(s: String): Element
```

正如你所见，元素将以名为 Element 的类型为模型。你将能在元素上调用 above 或 beside，传入第二个元素，从而得到合并了这两个的新元素。例如，下面的表达式将构建一个包含两列，每列高度为二，的更大的元素：

```
val column1 = elem("hello") above elem("***")
val column2 = elem("***") above elem("world")
column1 beside column2
```

打印这个表达式的结果将是：

```
hello ***
```

```
*** world
```

在对象能通过简单的部件及组合操作符的帮助被构建的系统中，布局元素是个好的例子。本章里，我们将定义类使得元素对象能被构建自数组，行记录，以及长方形——简单部件。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**组合子**：*combinator*，因为它们把某些区域的元素组合成新的元素。

以组合子的方式思考问题通常是实现库的设计的好方法：它能回报以考虑在应用域构建对象的基础方法。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？组合子是怎么挂在一起的？什么是最通用的组合？它们满足任何有趣的规则吗？如果你对这些问题都有好的答案，你的库设计就在正轨上了。

10.2　抽象类

我们的第一个任务就是定义代表布局元素的类型 Element。由于元素是二维的字符长方形，包括成员，指向布局元素内容的 contents，是合情合理的。内容可以被表达成字串数组，这里每个字串代表一行。因此，contents 返回的结果类型就是 Array[String]。代码 10.1 展示了它看起来的样子。

这个类里，contents 被声明为没有实现的方法。换句话说，方法是类 Element 的**抽象**：*abstract* 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 {
def contents: Array[String]
}
```

代码 10.1　定义抽象方法和类

```
abstract class Element ...
```

abstract 修饰符说明类或许有没实现的抽象成员。结果，你不能实例化抽象类。如果你尝试这么做，你会得到编译器错误：

```
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类 Element 的子类，你将能实例化它们因为它们补上了缺失的 contents 定义。

请注意类 Element 的 contents 方法并没带有 abstract 修饰符。如果方法没有实现（也就是说，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（或允许）抽象修饰符。拥有实现的方法被称为**具体的**：*concrete*。

另一个术语用法需要分辨**声明**：*declaration* 和**定义**：*definition*。类 Element **声明**了抽象方法 contents，但当前没有**定义**具体方法。然而下一节，我们要定义一些具体方法来加强 Element。

10.3　定义无参数方法

作为下一步，我们将向 Element 添加显示宽度和高度的方法，展示在代码 10.2 中。height 方法返回 contents 里的行数。width 方法返回第一行的长度，或如果元素没有行记录，返回零。（也就是说你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
}
```

代码 10.2　定义无参数方法 width 和 height

请注意 Element 的三个方法没一个有参数列表，甚至连个空列表都没有。例如，代之以：

```
def width(): Int
```

方法定义了不加括号的：

```
def width: Int
```

这种无参数方法在Scala里是非常普通的。相对的，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***。推荐的惯例是在没有参数**并且**方法仅通过读含有对象的方式访问可变状态（专指其不改变可变状态）时，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受通过字段还是方法实现属性的决定的影响。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。唯一的差别是与的访问或许稍微比方法调用要快，因为字段值在类被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。换句话说，字段在每个 Element 对象上需要更多的内存空间。因此类的使用概况，属性表达成字段还是方法更好，决定了其实现，并且这个概况还可以随时改变。重点是 Element 类的客户不应在其内部实现改变的时候受影响。

特别是如果类的字段变成了访问函数，且访问函数是**纯**的，就是说它没有副作用并且不依赖于可变状态，那么类 Element 的客户不需要被重写。客户都不应该需要关心这些。

目前为止一切良好。但仍然有些琐碎的复杂的东西要去做以协同 Java 处理事情的方式。问题在于 Java 没有实现统一访问原则。因此 Java 里是 string.length()，不是 string.length（尽管是 array.length，不是 array.length()）。不用说，这让人很困惑。

[1] Meyer，面向对象软件构造【Mey00】

为了在这道缺口上架一座桥梁，Scala 在遇到混合了无参数和空括号方法的情况时很大度。特别是，你可以用空括号方法重载无参数方法，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，在调用的方法表达的超过其接收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），或读出不是接受调用者的字段的 var，无论是直接的还是非直接的通过使用可变对象，那么空括号是合适的。这种方式是让参数列表扮演一个可见的线索说明某些有趣的计算正通过调用被触发。例如：

```
"hello".length // 没有副作用，所以无须()
println() // 最好别省略()
```

总结起来，Scala 里定义不带参数也没有副作用的方法为无参数方法，也就是说，省略空的括号，是鼓励的风格。另一方面，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选择一个字段。这样你的客户看到了副作用会很奇怪。相同地，当你调用带副作用的函数，请确信写这个调用的时候包括了空的括号。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，使用括号，但如果仅提供了对某个属性的访问，省略括号。

10.4　扩展类

我们仍然需要能够创建新的元素对象。你已经看到了因为类 Element 是抽象的，所以“new Element”不能被用来做这件事。因此，为了实例化一个元素，我们需要创建扩展了 Element 并实现抽象的 contents 方法的子类。代码 10.3 展示了一种可能的方式：

```
class ArrayElement(conts: Array[String]) extends Element {
  def contents: Array[String] = conts
}
```

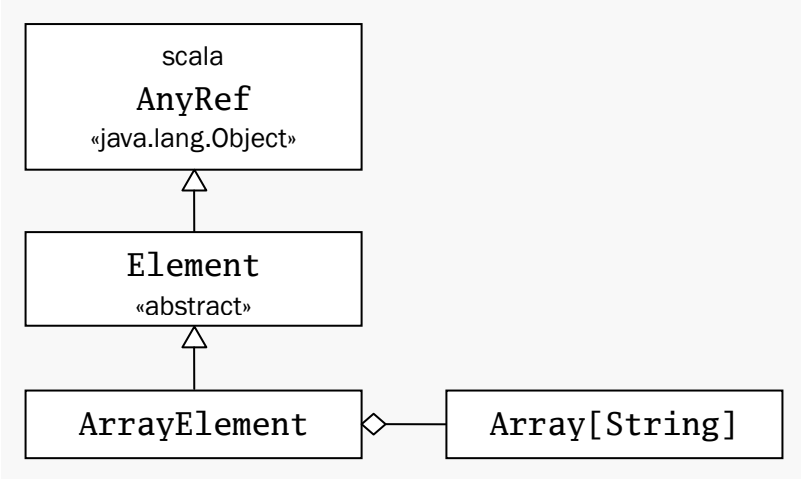
代码 10.3　定义 ArrayElement 为 Element 的子类

类 ArrayElement 定义为**扩展**了类 Element。就好象 Java 里，你在类名之后使用 extends 子句那样：

```
... extends Element ...
```

这种 extends 子句有两个效果：使类 ArrayElement 从类 Element **继承**所有非私有的成员，并且使 ArrayElement 成为 Element 的**子类型**。由于 ArrayElement 扩展了 Element，类 ArrayElement 被称为类 Element 的**子类**。反过来，Element 是 ArrayElement 的**超类**。

如果你省略 extends 子句，Scala 编译器隐式地假设你的类扩展自 scala.AnyRef，在 Java 平台上与 java.lang.Object 一致。因此，类 Element 隐式地扩展了类 AnyRef。你可以在图释 10.1 上看到这些继承关系。



图释 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，除了以下两点。首先，超类的私有成员不被子类继承。其次，在子类中实现的与超类中的成员具有相同名称和参数的将不被继承到子类中。这种情况我们说子类的成员**重载：***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，我们还可以说具体的成员**实现：***implement* 了抽象的。

例如，ArrayElement的contents方法重载（或者可说成：实现）了类Element的抽象方法contents。[2] 相对的，类ArrayElement从类Element继承了width和height方法。例如，给定ArrayElement的一个对象ae，你可以使用ae.width查询其长度，就好象width是定义在类ArrayElement中一样：

```
scala> val ae = new ArrayElement(Array("hello", "world"))
ae: ArrayElement = ArrayElement@d94e60
scala> ae.width
res1: Int = 5
```

子类型化：*subtyping* 是指子类的值可以被用在需要其超类的值的任何地方。例如：

```
val e: Element = new ArrayElement(Array("hello"))
```

变量e被定义为类型Element，所以其初始化的值也应当是Element。实际上，初始化值的类型是ArrayElement。这也没问题，因为类ArrayElement扩展了类Element，并且因此，类型ArrayElement适用于类型Element。[3]

图释 10.1 还展示了存在于 ArrayElement 和 Array[String]之间的**组合：***composition* 关系。这种关系被称为组合的原因是由于类 ArrayElement 是被 Array[String]“组合”出来的。因此 Scala 编译器将在它为 ArrayElement 产生的二进制类中安置一个字段用来保留传入的 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。

[2] 这个设计的一个漏洞是因为返回数组是可变的，所以客户端能改变它。本书中我们希望事情尽量简化，但当 ArrayElement 是真实项目中的部分时，你应当考虑代之以返回一个数组的防御性拷贝。另一个问题是我们现在并不确信 contents 数组所有的 String 元素具有同样的长度。这可以通过在主构造器中检查前提条件，并且一旦违反则抛出异常的方式来解决。

[3] 想了解更多子类和子类型之间的差异，参见词汇表中的 subtype。

10.5　重载方法和字段

统一访问原则只是 Scala 在对待字段和方法方面比 Java 更统一的一个方面。另一个差异是 Scala 里，字段和方法属于相同的命名空间。这使得字段重载无参数方法成为可能。比如说，你可以改变类 `ArrayElement` 中 `contents` 的实现，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，如展示在代码 10.4 中的那样：

```
class ArrayElement(conts: Array[String]) extends Element {
  val contents: Array[String] = conts
}
```

代码 10.4　用字段重载无参数方法

这个 `ArrayElement` 的版本里，字段 `contents`（用 `val` 定义）完美地实现了类 `Element` 里的无参数方法 `contents`（用 `def` 定义）。

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，而在 Java 里这样做被允许。例如，下面的 Java 类能够很好地编译：

```
// 在Java里的代码
class CompilesFine {
  private int f = 0;
  public int f() {
    return 1;
  }
}
```

但是相应的 Scala 类将不能编译：

```
class WontCompile {
  private var f = 0   // 编译不过，因为字段和方法重名
  def f = 1
}
```

通常情况下，Scala 仅为定义准备了两个命名空间，而 Java 有四个。Java 的四个命名空间是字段，方法，类型和包。与之相对，Scala 的两个命名空间是：

- 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清楚，因为这样你就可以使用`val`重载无参数的方法，这种你在Java里做不到的事情。[4]

[4] Scala 里包共享了与字段和方法相同的命名空间的原因是为了让你能除了仅仅引用类型名以及单例对象的字段和方法之外，还能直接引用包。这同样是你在 Java 中无法做到的。将在 13.2 节中描述。

10.6　定义参数化字段

再次考虑上一节中展示的 ArrayElement 类的定义。它有一个参数 conts，其唯一目的是被复制到 contents 字段。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”，一个表明或许某些不必须的荣誉和重复在你代码中出现的信号。

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避免这种代码味道，展示在代码 10.5 中：

```
class ArrayElement(   // 请注意，小括号
  val contents: Array[String]
) extends Element
```

代码 10.5　定义 contents 为参数化字段

注意现在 contents 参数前缀了 val。这是在同一时间使用相同的名称定义参数和字段的一个简写方式。尤其特别的是，类 ArrayElement 现在拥有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。就好象类被写成如下的方式，其中 x123 是参数的任意未曾用过的名字：

```
class ArrayElement(x123: Array[String]) extends Element {
  val contents: Array[String] = x123
}
```

同样也可以使用var前缀类参数，这种情况下相应的字段将能重新被赋值。最终，还有可能添加如private，protected，[5]或override这类的修饰符到这些参数化字段上，就好象你可以在其他类成员上做的事情。比方说，考察下列类定义：

```
class Cat {
  val dangerous = false
}
class Tiger(
  override val dangerous: Boolean,
  private var age: Int
) extends Cat
```

Tiger 的定义是以下包括重载成员 dangerous 和 private 成员 age 的类定义替代写法的简写：

```
class Tiger(param1: Boolean, param2: Int) extends Cat {
  override val dangerous = param1
  private var age = param2
}
```

两个成员都初始化自相应的参数。我们任意选择了这些参数名，param1 和 param2。重要的是它们不会与范围内的任何其它名称冲突。

[5] protected 修饰符，可以授权给子类访问，将在第 13 章详细描述。

10.7　调用超类构造器

现在你有了两个类组成的完整系统：抽象类 Element，和扩展它的具体类 ArrayElement。或许你还在设想其它表达元素的方式。比方说，客户或许想要创造由给定单行字串构成的布局元素。面向对象编程让使用新数据变体扩展系统变得容易。只要加入子类即可。例如，代码 10.6 展示了扩展 ArrayElement 的 LineElement 类：

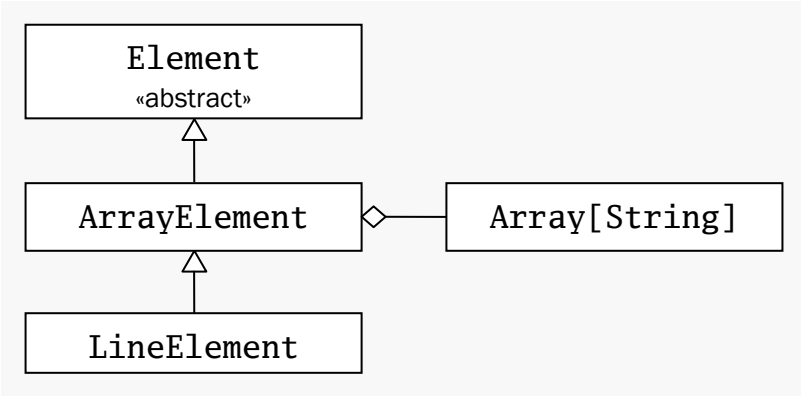
```
class LineElement(s: String) extends ArrayElement(Array(s)) {
  override def width = s.length
  override def height = 1
}
```

代码 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]），LineElement 需要传递一个参数到它的超类的主构造器。要调用超类构造器，只要把你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，类 LineElement 传递了 Array(s)到 ArrayElement 的主构造器，把它放在超类 ArrayElement 的名称后面的括号里：

```
... extends ArrayElement(Array(s)) ...
```

有了新的子类，布局元素的继承级别现在看起来就像展示在图释 10.2 中的那样了。



图释 10.2　LineElement 的类关系图

10.8　使用 override 修饰符

请注意 LineElement 里 width 和 height 的定义带着 override 修饰符。你在 6.3 节中的 toString 方法中看到过。Scala 里所有重载了父类具体成员的成员都需要这样的修饰符。如果成员实现的是同名的抽象成员则这个修饰符是可选的。而如果成员并未重载或实现什么其它基类里的成员则禁用这个修饰符。由于类 LineElement 的 height 和 width 重载了类 Element 的具体成员定义，override 是需要的。

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，编译器会回应错误信息：

```
$ scalac LineElement.scala
```

```
.../LineElement.scala:50:
error: method hight overrides nothing
  override def hight = 1
               ^
```

系统改进的时候，override 公约显得更重要。假设你定义了一个 2D 画图方法库。你把它公开，并广泛使用。库的下一个版本里你想在你的基类 Shape 里增加一个使用以下签名的新方法：

```
def hidden(): Boolean
```

你的新方法将被用在许多画图方法中去决定是否需要把形状画出来。这或许会产生显著的提速，但你不可以冒着破坏客户代码的风险做这件事。毕竟客户说不定已经使用不同的 hidden 实现定义了 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是否对象是隐藏的。因为这两个版本的 hidden 互相重载，你的画图方法将停止对象的消失，这可真不是你想要的！这些“意外的重载”就是被称为“脆基类”问题的最通常的表现。这个问题是指如果你添加了新的成员到类层级的基类中（通常我们称为超类），你会有破坏客户代码的风险。

Scala不能完全解决脆基类问题，不过它与Java相比有所改善。[6]如果画图库和它的客户是用Scala写的，那么客户的hidden原始实现就不会有override修饰符，因为这时候还没有另外一个使用那个名字的方法。一旦你添加了hidden方法到你Shape类的第二个版本，客户的重编译将给出像下列这样的错误：

```
.../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，代之以错误的执行，你的客户将得到一个编译期错误，这常常是更可取的。

10.9　多态和动态绑定

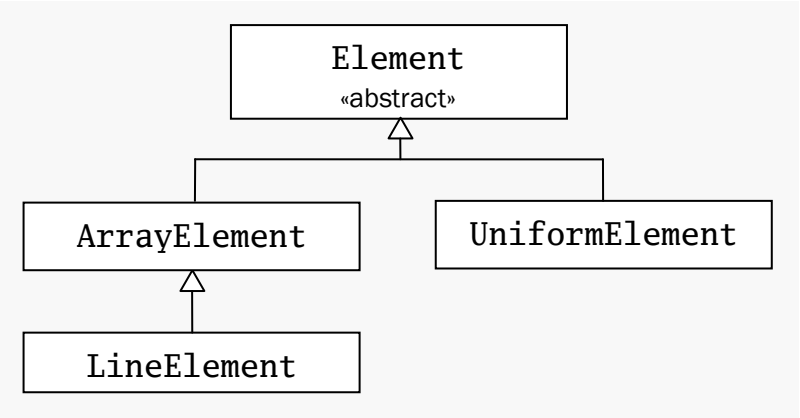
在 10.4 节中你看到了类型Element的变量可以指向类型ArrayElement的对象。这种现象的名字叫**多态**：*polymorphism*，是指“许多形状”或“许多形式”的意思。这种情况下，Element对象可以有许多形式。[7]目前为止，你已经看到了两种形式：ArrayElement和LineElement。你可以通过定义新的Element子类创造Element的更多形式。例如，下面定义了拥有给定长度和高度并被指定字符充满的新的Element形式：

```
class UniformElement(
  ch: Char,
  override val width: Int,
  override val height: Int
) extends Element {
```

[6] Java1.5 中，@Override 标注被引入并与 Scala 的 override 修饰符起相同的工作，不过不像 Scala 的 override，它不是必需的。

[7] 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*，将在第 19 章讨论。

```
  private val line = ch.toString * width
  def contents = Array.make(height, line)
}
```



图释 10.3　布局元素的类层级

类 Element 的继承层级现在看上去如图释 10.3 展示的样子。结果，Scala 将接受所有的下列赋值，因为赋值表达式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)
```

若你检查继承层级，你会发现这四个 val 定义的每一个里，等号右侧表达式的类型都在将被初始化的等号左侧的 val 类型之下。

然而，另一半的故事是，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。为了演示这种行为，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 中重载 demo，但 UniformElement 除外：

```
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")
  }
```

```
}
// UniformElement inherits Element's demo
class UniformElement extends Element
```

如果你把这些代码输入到了解释器中，那么你就能定义这个带了一个 Element 并调用 demo 的方法：

```
def invokeDemo(e: Element) {
  e.demo()
}
```

如果你传给 invokeDemo 一个 ArrayElement，你会看到一条消息指明 ArrayElement 的 demo 实现被调用，尽管被调用 demo 的变量 e 的类型是 Element：

```
scala> invokeDemo(new ArrayElement)
ArrayElement's implementation invoked
```

相同的，如果你传递 LineElement 给 invokeDemo，你会看到一条指明 LineElement 的 demo 实现被调用的消息：

```
scala> invokeDemo(new LineElement)
LineElement's implementation invoked
```

传递 UniformElement 时的行为一眼看上去会有些可以，但是正确：

```
scala> invokeDemo(new UniformElement)
Element's implementation invoked
```

因为 UniformElement 没有重载 demo，它从它的超类 Element 继承了 demo 的实现。因此，当对象的类是 UniformElement 时，Element 的实现就是要调用的 demo 的正确实现。

10.10　定义 final 成员

设计一个继承层级的某些时候，你想要确保成员不被子类重载。Scala 里和 Java 里一样，通过添加 final 修饰符给成员来做到。例如，你可以在 ArrayElement 的 demo 方法前放一个 final 修饰符，如代码 10.7 中展示的那样。

```
class ArrayElement extends Element {
  final override def demo() {
    println("ArrayElement's implementation invoked")
  }
}
```

代码 10.7　声明 final 方法

有了这个版本的 ArrayElement，尝试在它的子类，LineElement，重载 demo 方法，将编译不过：

```
elem.scala:18: error: error overriding method demo
  in class ArrayElement of type ()Unit;
method demo cannot override final member
    override def demo() {
```

```
                ^
```

你或许还多次想确保整个类都没有子类。要做到这点只要简单地通过在类的声明上添加 final 修饰符把整个类声明为 final 即可。如，代码 10.8 展示了如何声明 ArrayElement 为 final：

```
final class ArrayElement extends Element {
  override def demo() {
    println("ArrayElement's implementation invoked")
  }
}
```

代码 10.8　声明 final 类

有了这个版本的 ArrayElement，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 {
                          ^
```

我们现在将去掉 final 修饰符和 demo 方法，并回到早先实现的 Element 家族。我们将把我们本章剩余部分的注意力集中在完成布局库的工作版本上。

10.11　使用组合与继承

组合与继承是利用其它现存类定义新类的两个方法。如果你接下来的工作主要是代码重用，通常你应采用组合而不是继承。只有继承受脆基类问题之苦，这种情况你可能会无意中通过改变超类而破坏了子类。

关于继承关系你可以问自己一个问题，是否它建模了一个*is-a*关系。[8]例如，说ArrayElement**是**Element是合理的。你能问的另一个问题是，是否客户想要把子类类型当作超类类型来用。[9]在ArrayElement的例子里，我们的确期待客户会想要把ArrayElement当作Element使用。

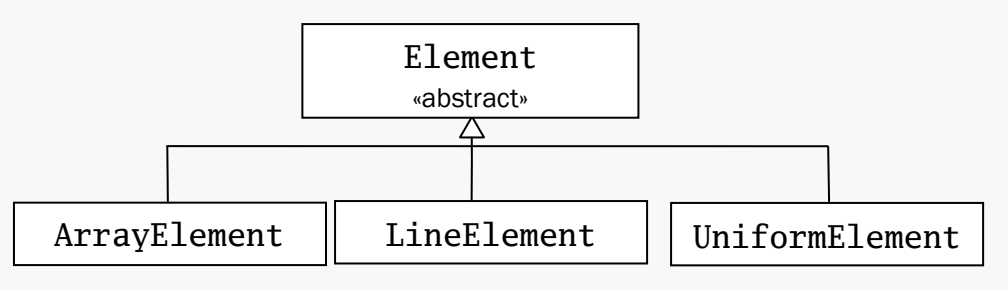
如果你对展示在图释 10.3 的继承关系问了这些的问题，那么是否感觉其中的任何关系有可疑吗？尤其是，对你来说 LineElement **是** ArrayElement 是否显而易见呢？你是否认为客户会需要把 LineElement 当作 ArrayElement 使用？实际上，我们把 LineElement 定义为 ArrayElement 主要是想重用 ArrayElement 的 contents 定义。因此或许把 LineElement 定义为 Element 的直接子类会更好一些，就像这样：

```
class LineElement(s: String) extends Element {
  val contents = Array(s)
  override def width = s.length
  override def height = 1
}
```

前一个版本中，LineElement与ArrayElement有一个继承关系，从那里继承了contents。现在它

[8] Meyers，《Effective C++》 【Mey91】
[9] Eckel，《Thinking in Java》 【Eck98】

与Array有一个组合关系：在它自己的contents字段中持有一个字串数组的引用。[10] 有了LineElement的这个实现，Element的继承层级现在看上去如展示在图释 10.4 中那样。



图释 10.4　修改了 LineElement 后的类层级

10.12 实现 above，beside 和 toString

接下来一步，我们将在类 Element 中实现方法 above。把一个元素放在另一个上面是指串连这两个元素的 contents 值。因此方法 above 的第一个草案看上去可能是这样的：

```
def above(that: Element): Element =
  new ArrayElement(this.contents ++ that.contents)
```

++操作符串连两个数组。Scala 里的数组被表示为 Java 数组，但是支持更多的方法。特别是，Scala 里的数组继承自类 scala.Seq，能够表现象序列这样的结构并包含许多访问和转换序列的方法。本章会解释某些数组方法，更全面的讨论将在第 17 章。

实际上，前面展示的代码并不完全足够，因为它不允许你把不同长度的元素堆叠在一起。然而本节为了让事情保持简单，我们会任由其状态并仅仅把相同长度的元素传递给 above。10.14 节里，我们会给 above 做个改良，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，我们将创造一个新的元素，其中的每一行都来自于两个元素的相应行的串连。如前所述，为了保持事情简单我们会一开始假设两个元素高度相同。这产生了方法 beside 的下列设计：

```
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，并串连 this.contents 和 that.contents 中相应的数组元素来填充。最终产生了新的 ArrayElement 包含了新的 contents。

尽管 beside 的这个实现可以工作，但它是指令式风格，马脚露在我们索引数组的循环。这个方

[10] 类 ArrayElement 也与 Array 有组合关系，因为它的参数化 contents 字段持有字串数组的引用。ArrayElement 的代码展示在第 xx 页的代码 10.5 中。其组合关系用一个菱形表现在类图中，正如展示在第 xx 页的图释 10.1 中那样。

法可以替代缩减成一个表达式：

```
new ArrayElement(
  for (
    (line1, line2) <- this.contents zip that.contents
) yield line1 + line2
)
```

这里，this.contents 和 that.contents 两个数组被使用 zip 操作符转换为一个对子的数组（可以称为 Tupele2）。zip 方法从它的两个参数中拣出相应的元素并组织成对子数组。

例如，表达式：Array(1, 2, 3) zip Array("a", "b")

将生成：Array((1, "a"), (2, "b"))

如果两个操作数组的其中一个比另一个长，zip 将舍弃余下的元素。在上面的表达式中，左操作数的第三个元素，3，没有组成结果的部分，因为它在右操作数中没有相对的元素。

结果数组然后通过 for 表达式被枚举遍历。这里，表达式“for ((line1, line2) <- ...” 允许你在一个模式：pattern 中命名对子的两个元素，也就是说，line1 现在代表对子的第一个元素，line2 代表第二个。Scala 的模式匹配系统将在第 15 章描述。现在，你可以就把这当作在每次枚举中定义两个 val，line1 和 line2 的方式。

for 表达式有个 yield 部分能产生结果。结果与枚举遍历的表达式类型一致，也就是说，是数组。数组的每个元素都是相应行，line1 和 line2 串连的结果。因此这段代码的最终结果与前一个版本的 beside 一样，不过因为它避免了显示的数组索引，结果用一种更少犯错的方式实现了。

你还需要一个显示元素的方式。通常，可以通过定义 toString 方法返回元素格式化成的字串做到。下面是它的定义：

```
override def toString = contents mkString "\n"
```

toString 的实现使用了 mkString，它被定义在所有序列中，包括数组。正如你在 7.8 节中看到的 像“arr mkString sep”这样的表达式能返回数组 arr 所有元素组成的字串。通过调用 toString 方法每个元素被映射为字串。分隔符字串 seq 被插入到连续的元素字串当中。因此表达式“contents mkstring "\n"”格式化 contents 数组为字串，其中每个数组元素占据一行。

请注意 toString 没有带空参数列表。这个遵循了统一访问原则的建议，因为 toString 是一个纯的不带任何参数的方法。

附加了这三个方法，类 Element 现在看上去如代码 10.9 所展示的。

```
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"
}
```

代码 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“依原件”展现给你的客户。但是你或许还是选择把层级隐藏在工厂对象之后。工厂对象包含了构建其它对象的方法。客户与实惠使用这些工厂方法实现对象的构造而不是直接使用 new 构造对象。这种方式的一个好处是对象的创建可以被集中化并且对象实际代表类的细节可以被隐藏。这种隐藏一方面简化客户理解你的库，因为更少的细节被暴露出来，另一方面提供给你更多机会在之后改变库的实现而不会破坏客户代码。

为布局元素构建工厂的第一任务是选择工厂方法应该放在哪儿。它们应该是单例对象成员还是类成员？包含它们的对象或类应该怎么调用？这里有许多可能性。最直接的方案是创建类 Element 的伴生对象并把它做成布局元素的工厂方法。对于这种方式，你唯一要暴露给客户的就是 Element 的类/对象组合，隐藏它的三个实现类 ArrayElement，LineElement 和 UniformElement。

代码 10.10 是遵循了这个方案的设计。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)
}
```

代码 10.10 带有工厂方法的工厂对象

这些工厂方法使得改变类 Element 的实现通过使用 elem 工厂方法实现而不用显式地创建新的 ArrayElement 实例成为可能。为了不使用单例对象的名称，Element，认证而调用工厂方法，我们将在源文件顶上引用 Element.elem。换句话说，代之以在 Element 类内部使用 Element.elem 调用工厂方法，我们将引用 Element.elem，这样我们只要使用它们的简化名，elem，就可以调用工厂方法。代码 10.11 展示了类 Element 在这些改变之后的样子。

```
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"
}
```

代码 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。Scala 里，你可以在类和单例对象中定义其它的类和单例对象。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。代码 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)
}
```

代码 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良。展示在代码 10.11 中的 Element 的版本并不完全，因为他不允许客户把不同宽度的元素堆叠在一起，或者不同高度的元素靠在一起。比方说，下面的表达式将不能正常工作，因为组合元素的第二行比第一行要长：

```
new ArrayElement(Array("hello")) above
new ArrayElement(Array("world!"))
```

与之相似的，下面的表达式也不能正常工作，因为第一个 ArrayElement 高度为二，而第二个的高度只是一：

```
new ArrayElement(Array("one", "two")) beside
new ArrayElement(Array("one"))
```

代码 10.13 展示了一个私有帮助方法，widen，能够带个宽度做参数并返回那个宽度的 Element。结果包含了这个 Element 的内容，居中，左侧和右侧留需带的空格以获得需要的宽度。代码 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，布局库可以待用了。

```
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"
}
```

代码 10.13 有了 widen 和 heighten 方法的 Element

10.15 把代码都放在一起

操练布局库所有这些元素的好玩儿的方法就是写一个画给定数量边界的螺旋的程序。这个 Spiral 程序，展示在代码 10.14 中，是这么做的：

```
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))
  }
}
```

代码 10.14 Spiral 程序

因为 Spiral 是个带有合适签名的 main 方法的独立的对象，所以它是个 Scala 程序。Spiral 带一个命令行参数，一个整数，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边界的螺旋，或者右边的那样更大的螺旋：（略）

10.16 结语

本章中，你看到了 Scala 里与面向对象编程有关的更多的概念。其中，你遇到了抽象类，继承和子类化，类层级，参数化字段，及方法重载。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 章重新回到布局库的工作中。

第11章

Scala 的层级

现在你已经在前一章里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，那么定义在 Any 中的方法就是“普遍”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，Null 和 Nothing，主要都扮演通用的**子**类。例如，就像说 Any 是所有其它类的超类，Nothing 是所有其它类的子类。本章中，我们将带你周游 Scala 的类层级。

11.1 Scala 的类层级

图释 11.1 展示了 Scala 的类层级的大纲。层级的顶端是类 Any，定义了包含下列的方法：

```
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，Scala 程序里的每个对象都能用==，!=或 equals 比较；用 hashCode 哈希；和用 toString 格式化。类 Any 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面重载。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载 equals 方法剪裁==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。有九个这样的值类：Byte，Short，Char，Int，Long，Float，Double，Boolean 和 Unit。其中的前八个对应到 Java 的原始类型，它们的值在运行时表示成 Java 的原始值。Scala 里这些类的实例都写成文本。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。这一点被“小伎俩”，值类都被定义为即是抽象的又是 final 的，强制贯彻。因此如果你写了：

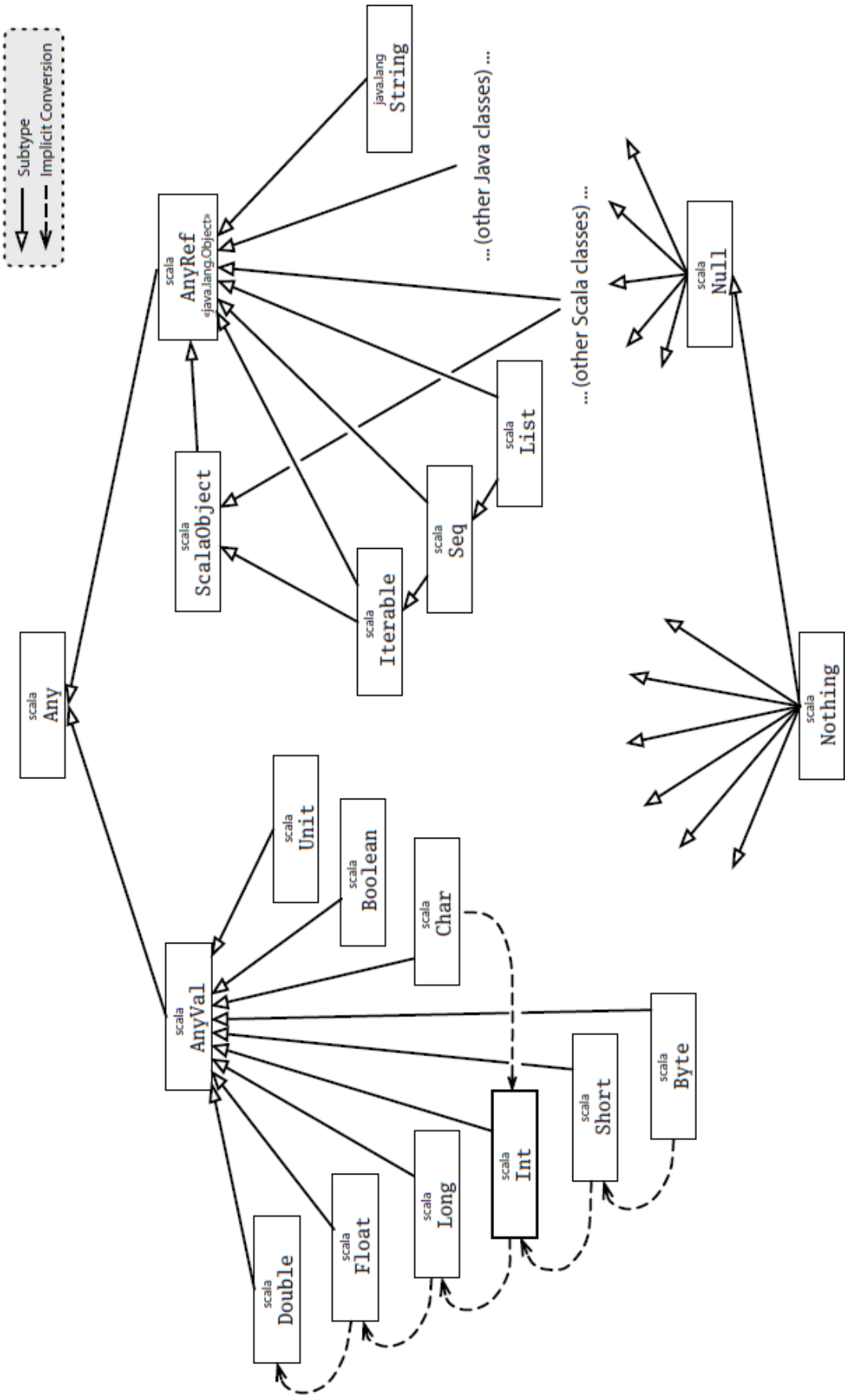
```
scala> new Int
```

你就会得到：

```
<console>:5: error: class Int is abstract; cannot be instantiated
       new Int
       ^
```

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，被写作()，在 7.2 节中讨论过。

正如第五章中解释过，值类支持作为方法的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：



图释 11.1　Scala 类层级图

```
scala> 42.toString
res1: java.lang.String = 42
scala> 42.hashCode
res2: Int = 42
scala> 42 equals 42
res3: Boolean = true
```

注意，值类的空间是扁平的；所有的值类都是 scala.AnyVal 的子类型，但是它们不是互相的子类。代之以它们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 上调用没有定义在 Int 上但定义在 RichInt 上的方法时，这个转换就被应用了。类似的“助推器类”和隐式转换存在于其它的值类。隐式转换将在第 21 章讨论细节。

类Any的另一个子类是类AnyRef。这个是Scala里所有**引用类**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]如此说来，你可以认为java.lang.Object是Java平台上实现AnyRef的方式。因此，尽管你可以在Java平台上的Scala程序里交换使用Object和AnyRef，推荐的风格是在任何地方都只使用AnyRef。

Scala 类与 Java 类不同在于它们还继承自一个名为 ScalaObject 的特别的记号特质。理念是 ScalaObject 包含了 Scala 编译器定义和实现的方法，目的是让 Scala 程序的执行更有效。到现在为止，Scala 对象包含了单个方法，名为$tag，用于内部以提速模式匹配。

[1] 存在 AnyRef 别名代替使用 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。在.NET 平台上，AnyRef 是 System.Object 的别名。

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 上的效率以及与 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原始操作。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”。

所有这些听上去都近似 Java5 里的自动装箱并且它们的确很像。不过有一个关键差异，Scala 里的装箱比 Java 里的更少看见。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, Integery) {
  return x == y;
}
System.out.println(isEqual(421,421));
```

你会发现你得到了 `false`！原因是数 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`。这是展示了 Java 不是纯面向对象语言的一个方面。我们能清楚观察到原始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
```

实际上 Scala 里的相等操作==被设计为透明的参考类型代表的东西。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，但被许多子类重载实现它们种族的相等概念。这也意味着 Scala 里你永远也不会落入 Java 知名的关于字串比较的陷阱。Scala 里，字串比较以其应有的方式工作：

```
scala>val x = "abcd".substring(2)
x:java.lang.String = cd
```

```
scala>valy="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**哈希合并**：*hash cons*然后通过引用相等比较它们的实例。[2]为这种情况，类AnyRef定义了附加的eq方法，它不能被重载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

在图释 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某些 Scala 的面向对象类型系统的“边界情况”的特殊类型。

类 Null 是 null 类型的引用；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。你不可，比方说，把 null 值赋给整数变量：

```
scala>val i: Int = null
<console>:4:error:typemismatch;
       found:Null(null)
       required:Int
```

类型 Nothing 在 Scala 的类层级的最底端；它是任何其它类型的子类型。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（代之以抛出了异常）。因为 Nothing

[2] 类实例的哈希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等于你打算创建的，你可以重用存在的实例。这样安排的结果是，任何以 equals()判断相等的两个实例同样在引用相等上判断一致。

是任何其它类型的子类，你可以非常灵活的使用像 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”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，整个状态语句的类型是 Int，正如需要的那样。

11.4　结语

本章中我们展示给你了在 Scala 类层级的顶端和底端的类。现在你已经具有了 Scala 里类继承的良好的基础，你已经做好了理解混入组合的准备。下一章，你会学到关于特质的东西。

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。不像类的继承那样，每个类都只能继承唯一的超类，类可以混入任意个特质。本章将告诉你特质是如何工作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。代码 12.1 举例如下：

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
```

代码 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，有个缺省的超类 AnyRef。它定义了一个方法，叫做 philosophize，具体的。这是个简单的特质，仅够说明特质如何工作。

一旦特质被定义了，就可以使用 extends 或 with 关键字，把它混入到类中。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它语言中的多继承有重要的差别。这部分在 12.6 节中讨论。例如，代码 12.2 展示了使用 extends 混入 Philosophical 特质的类：

```
class Frog extends Philosophical {
  override def toString = "green"
}
```

代码 12.2 使用 extends 混入特质

你可以使用 extends 关键字混入特质；这种情况下你隐式地继承了特质的超类。举例来说，在代码 12.2 中，类 Frog 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，一个特质。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。

如果想把特质混入到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。代码 12.3 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，参见代码 12.4。

```
class Animal
class Frog extends Animal with Philosophical {
  override def toString = "green"
}
```

代码 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"
}
```

代码 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 也可以重载 philosophize 方法。语法与重载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 重载了 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!
```

这时你或许推导出以下哲理：特质就像是带有具体方法的 Java 接口，不过其实它能做的更多。特质可以，比方说，声明字段和维持状态值。实际上，你可以用特质定义做任何用类定义做的事，并且语法也是一样的，除了两点。第一点，特质不能有任何“类”参数，也就是说，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)
```

但是下面定义特质的尝试将遭到失败：

```
trait NoPoint(x: Int, y: Int) // 编译不过
```

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，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，特质可以丰满一个**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。客户可以捡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
}
```

尽管类 String 成打的方法中的大多数都可以用在任何 CharSequence 上，Java 的 CharSequence 接口定义仅提供了 4 个方法。如果 CharSequence 代以包含全部 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方法一次，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。

要使用特质丰满接口，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。然后你就可以把丰满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

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，首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。

这种重复可以使用丰满了的特质消除。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。然后他就可以应用到所有其他的几何查询的具体实现中。代码 12.5 展示了代码的样子：

```
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……
```

```
}
```

代码 12.5　定义丰满了的特质

类 Component 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 {
  // 其它方法……
}
```

与之类似，Rectangle 本身也可以混入特质：

```
class Rectangle(val topLeft: Point, val bottomRight: Point)
    extends Rectangular {
  // 其它方法……
}
```

有了这些定义，你可以创建 Rectangle 对象并对它调用如 width 或 left 的几何方法：

```
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]

```
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)
```

[1] 本例基于的 Rational 类的全部代码在代码 6.5 中。

```
}
```

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。

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。并且应该返回一个整数零，如果对象相同；或者负数如果接受者小于参数；或者正数如果接受者大于参数。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
```

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，你将能够提供给类的使用者丰满的比较方法集。

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 章找到怎么做。

12.5　特质用来做可堆叠的改变

现在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。现在我们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，它们能够让你通过**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 应该以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，把它们混入到类中，并获得你所需改动的新类。

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
```

代码 12.6　抽象类 IntQueue

抽象的 IntQueue 类展示在代码 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 BasicIntQueue 实现参见代码 12.7。

```
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 }
}
```

代码 12.7　使用 ArrayBuffer 实现 BasicIntQueue

BasicIntQueue 类有一个私有字段持有数组缓存。get 方法从缓存的一段移除成员，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
```

目前为止都很好。现在看一下使用特质改变它的行为。代码 12.8 展示了在把整数放入队列的时候对它加倍。Doubling 特质有两件好玩的事情。第一个是他定义了超类，IntQueue。这个定义意味着特质只能混入到扩展了 IntQueue 的类中。因此你可以把 Doubling 混入到 BasicIntQueue，但是不能混入到 Rational。

```
trait Doubling extends IntQueue {
  abstract override def put(x: Int) { super.put(2 * x) }
}
```

代码 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 调用将直到被混入在另一个特质或类**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，而不是类，成员定义中被认可，它意味着特质必须被混入到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道道！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。如代码 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
```

代码 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代码 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)
  }
}
```

代码 12.10 可堆叠改动特质 Incrementing 和 Filtering

有了这些改动，你现在可以拣选你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
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，你可以定义十六个不同的类。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？

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

[2] 一旦特质混入了类，你也可以称其为**混入**。

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)
  }
}
```

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，被调用的方法就是链子的下一节。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。

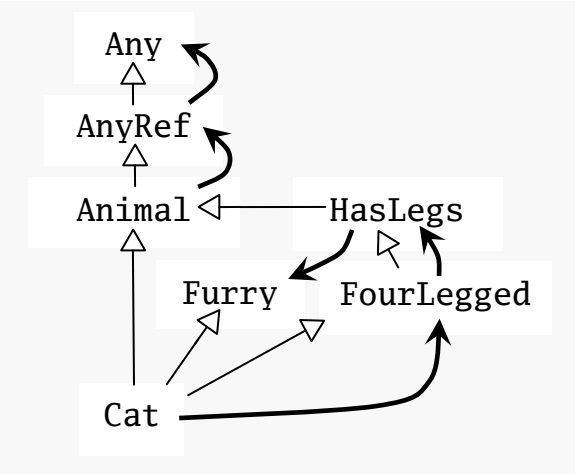
线性化的精确次序由语言的式样书描述。虽然有一点儿复杂，但你需要知道的主旨就是，在任何的线性化中，某个类总是被线性化在**所有**其超类和混入特质之前。因此，当你写了一个调用 super 的方法时，这个方法必将改变超类和混入特质的行为，没有其它路可走。

> **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，你可以安全地跳过这部分。

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
```



图释 12.1　Cat 类的继承层级和线性化次序

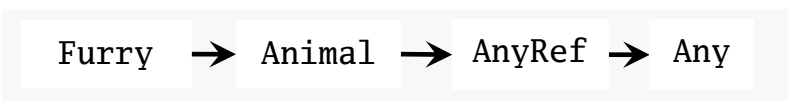
类 Cat 的继承层级和线性化次序展示在图 12.1 中。继承次序使用传统的 UML 标注指明：白色三角箭头表明继承，箭头指向超类型。黑底非三角箭头说明线性化次序，箭头指向 super 调用解决的方向。

表格 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

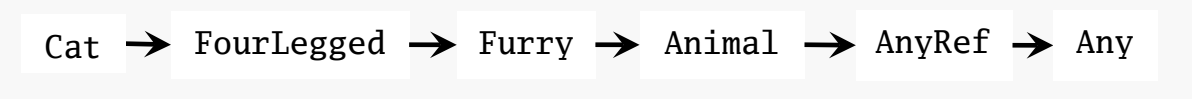
Cat 的线性化次序以下列的从后向前的顺序计算。Cat 线性化的**最后**部分是它的超类，Animal 的线性化。这个线性化被无损的复制过来。（这些类型每一个的线性化次序展示在表格 12.1 中）。因为 Animal 没有显式扩展超类或混入任何超特质，因此它缺省地扩展了 AnyRef，并随之扩展了 Any。因此，Animal 的线性化，看上去是这样的：

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：



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：

最后，Cat 线性化的第一个类是 Cat 自己：



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导则。

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，倾向于类。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，仅当你知道那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能确实解决问题，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，并且通常使用特质始终保持着选择的可能性。

12.8　结语

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的丰满的特质。

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。

第13章

包和引用

做程序的时候，尤其是很大的程序，让**耦合：*coupling***——程序的各个部分依赖于其他部分的程度——变得最小是很重要的。低耦合能让程序某一部分的很小的无所谓的改变却颠覆了另一部分的正常执行这样的风险降低。一种减少耦合性的方式是使用模块化风格写代码。把程序分解成若干比较小的模块，每个都能够区分内外部。在模块的内部——它的**实现**——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——它的接口——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，通过引入让名字可见，并且通过访问修饰符控制定义的可见度。这些构造与 Java 的构造在精神上一致，但是也有一些区别——通常能保持更好的一致性——所以，即使你已经从 Java 知道了这些，读一下本章也是有价值的。

13.1 包

Scala 的代码采用了 Java 平台的完整的包机制。本书到目前为止你已经看到的例子代码都是在***unnamed*** 包中。你可以用两种方式把代码放在命名包中。首先，可以通过把 `package` 子句放在文件顶端的方式把整个文件内容放进包里，如代码 13.1 所示。

```
package bobsrockets.navigation
class Navigator
```

代码 13.1 把文件的全部内容放进包里

代码 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吧。

注意

由于 Scala 代码是 Java 生态系统的一部分，在你发布到公开场合的时候，推荐遵从 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 里把代码放在包里的另一种方式很像 C#的命名空间。在 `package` 子句之后用大括号包起来一段要放到包里去的定义。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作为原始代码放在同一个文件，但不同的包里。如代码 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
    }
  }
}
```

代码 13.2　同一个文件嵌入不同的包

代码 13.1 中展示的类似于 Java 的语法实际上只是提供给代码 13.2 展示的更通用的嵌入语法的语法糖。实际上，如果除了签入另一个包之外对包不作任何事，你可以使用代码 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
  }
}
```

代码 13.3　较少缩进的嵌入包

正如注释所提示的，Scala 的包的确是嵌套的。也就是说，包 navigation 从语义上在包 bobsrockets 内部。Java 包，尽管是分级的，却不是嵌套的。在 Java 里，在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严谨的规则。

看看代码 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 包含在 bobsrockets 包中，它含有 navigation 成员。因此，可以不用前缀的提到 navigation，就好象类方法里的代码可以直接提到类的其它方法而不用前缀。

```
package bobsrockets {
  package navigation {
    class Navigator
  }
  package launch {
    class Booster {
      // 不用写bobsrockets.navigation.Navigator
      val nav = new navigation.Navigator
    }
  }
}
```

代码 13.4　Scala 的包确实是嵌套的

Scala 的区域规则导出的另一个结果就是内部区域的包可以隐匿被定义在外部区域的同名包。举个例子，参考代码 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）。这种重复的名字同样有效——毕竟这就是使用包的主要原因——但是这也意味着你必须为精确地访问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，看一下代码 13.5 的 MissionControl。如何引用 Booster1，Booster2，和 Booster3？访问第一个最简单。到 launch 的索引本身就能带你到包 bobsrockets.navigation.launch，因为 launch 包定义在最近的内附范围。因此，你可以简单的使用 launch.Booster1 指明第一个 booster 类。指明第二个同样也不是棘手的事。你可以写下 bobrockets.launch.Booster2 并且很清楚你所指的是哪个。然而，剩下的问题是第三个 booster 类。在内嵌的 launch 包遮盖了顶层包的情况下，怎么访问 Booster3？

```
// 文件launch.scala
package launch {
  class Booster3
}
// 文件bobsrockets.scala
package bobsrockets {
  package navigation {
    package launch {
      class Booster1
    }
    class MissionControl {
      val booster1 = new launch.Booster1
      val booster2 = new bobsrockets.launch.Booster2
      val booster3 = new _root_.launch.Booster3
    }
  }
  package launch {
    class Booster2
  }
}
```

代码 13.5　访问隐藏的包名

为了解决这种情况，Scala 提供了所有用户可创建的包之外的名为_root_的包。换句话就是，任何你写的顶层包都被当作是_root_包的成员。例如，代码 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，考虑一下代码 13.6 展示的例子：

```
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)
}
```

代码 13.6　鲍勃最爱的水果，已为引用做好准备

import 子句可以使得包或对象的成员只通过它们的名称被访问而不用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，第二个是 Java 的**按需：*on-demand*** 引用。唯一的差别是 Scala 的按需引用写作尾下划线（_）而不是星号（*）（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。

这三个引用让你尝到了引用的滋味，但是 Scala 引用实际上更为通用。举一个例子，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，代码 13.7 展示的引用是可能的：

```
def showFruit(fruit: Fruit) {
  import fruit._
  println(name +"s are "+ color)
}
```

代码 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 类型的 fruit，的所有成员。之后的 println 语句就可以直接使用 name 和 color 了。这两个索引等价于 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。

说明 Scala 的引用很灵活的另一个方面是它们可以引用包自身，而不只是非包成员。这只有你把内嵌包想象成包含在外围包之内才是自然的。例如，代码 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，如代码 13.8 所示：

```
import java.util.regex
class AStarB {
  // 访问java.util.regex.Pattern
  val pat = regex.Pattern.compile("a*b")
}
```

代码 13.8　引用包名

Scala 的引用同样可以重命名或隐藏成员。可以用跟在引用的成员对象之后的包含在括号里的**引**

用选择子句： ***import selector clause*** 做到。下面是一些例子：

```
import Fruits.{Apple, Orange}
```

这次只引用了对象 Fruits 的 Apple 和 Orange 成员。

```
import Fruits.{Apple => McIntosh, Orange}
```

这次从对象 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}
```

这次以 SDate 的名字引用了 SQL 的日期类，因此你可以在同时引用普通的 Java 日期类 Date。

```
import java.{sql => S}
```

这个以名称 S 引用了 java.sql 包，这样你就可以写成 S.Date。

```
import Fruits.{_}
```

这个引用了对象 Fruits 的所有成员。这与 import Fruits._同义。

```
import Fruits.{Apple => McIntosh, _}
```

这个从 Fruits 对象引用所有成员，不过重命名 Apple 为 McIntosh。

```
import Fruits.{Pear => _, _}
```

这个引用了**除** Pear 之外的所有 Fruits 成员。“<原始名> => _”格式的子句从被引用的名字中排除了<原始名>。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。如果你想只是得到名为 Apple 的笔记本，而不是水果，你仍然可以使用需要的两个引用如下：

```
import Notebooks._
import Fruits.{Apple => _, _}
```

这将引用所有的 Notebooks 和除了 Apple 之外所有的水果。

这些例子演示了选择性及重命名引用成员时，Scala 提供的极大的灵活性。总而言之，引用选择可以包括下列模式：

- 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 隐式地添加了一些引用到每个程序中。本质上，就好象下列的三个引用子句已经被加载了每个以“.scala”为扩展名的源文件的顶端：

```
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 包囊括了标准 Java 类。它永远被隐式包含在 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，举例来说你就可以直接用 Thread 而不是 java.lang.Thread。

现在你应该毫无疑问地意识到，scala 包含有标准的 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 而不是 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。比如，因为 Predef 是隐式引用，你可以直接写 assert 而不是 Predef.assert。

上面的这三个引用子句与其它的稍有不同，靠后的引用将遮盖靠前的。例如，StringBuilder 类被定义在 scala 包里以及，从 Java 版本 1.5 开始，还在包 java.lang 中。因为 scala 引用遮盖了 java.lang 引用，所以 StringBuilder 简单名将被看作是 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 严格遵从 Java 对访问修饰符的对待方式，但也有一些重要的差异将在本节说明。

私有成员

私有成员是按照 Java 同样对待的。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，这个规则同样应用到了内部类上。这种方式更一致，但不同于 Java。参见展示在代码 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
}
```

代码 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，类 InnerMost 里访问 f 没有问题，因为这个访问包含在 Inner 类之内。Java 会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

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。Java 中，这种访问同样可以在类的同一个包里。Scala 中，另有途径达到这种效果，将在下面介绍，因此 protected 可以就这样随它去。代码 13.10 的例子演示了 protected 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
  }
}
```

代码 13.10 Scala 和 Java 的 protected 访问差异

代码 13.10 中，类 Sub 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 的是公开的。公开成员没有显式修饰符。这样的成员可以在任何地方被访问。

保护的范围

Scala 里的访问修饰符可以通过使用修饰词增加。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 指代某些外围的包，类或单例对象。

有修饰的访问修饰符给了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 的访问理念，而这些都不是用 Scala 的简单修饰符能直接表达的。它们还能让你表达 Java 无法表达的访问规则。代码 13.11 表达了使用许多访问修饰符的例子。在这段代码中，类 Navigator 被标记为 private[bobsrockets]。这就是说这个类对包含在 bobsrockets 包的所有的类和对象可见。特别是对象 Vehicle 里对 Navigator 的访问被允许，因为 Vehicle 包含在包 launch 中 而 launch 包在 bobsrockets 中。另一方面，包 bobsrockets 包之外的所有代码都不能访问类 Navigator。

这种技巧在划分为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
    }
  }
}
```

代码 13.11 使用访问修饰词的灵活的保护范围

当然，private 修饰词同样可以直接是外围包。代码 13.11 里对象 Vehicle 的 guide 访问修饰符是这样的例子。这种访问修饰符等价于 Java 的包私有访问。

表格 13.1　LegOfJourney.distance 上的私有修饰词效果

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

所有的修饰词也可以用在 protected 上，与 private 意思相同。也就是说，类 C 的 protected[X] 修饰符允许 C 的所有子类和外围的包，类，或对象 X 访问被标记的定义。例如，代码 13.11 里的 useStarChart 方法在 Navigator 所有子类以及包含在 navigation 包里的所有代码能够被访问。这与 Java 的 protected 意思完全一致。

Private 的修饰词还能指向外围类或对象。例如代码 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。这与 Java 里的内部类的私有成员具有同样的访问能力。private[C]里的 C 如果是最外层类，那么 private 的意思和 Java 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，代码 13.11 中 类Navigator的的speed定义就是对象私有的。这就是说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，将不被允许，即使它发生在 Navigator 类之中：

```
val other = new Navigator
other.speed // this line would not compile
```

把成员标记为 private[this]是一个让它不能被同一个类中其它对象访问的保障。这在做文档时比较有用。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，表格 13.1 罗列了 private 修饰字的效果。每一行展示了一个被修饰的私有修饰符以及如果这个修饰符被附加在代码 13.11 的 LegOfJourney 类里声明的 distance 变量上意味着什么。

可见度和伴生对象

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他们之上。你已经知道在 Scala 里没有静态成员；代之以可以拥有包含成员的仅存在一个的伴生对象。例如，代码 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() {}
}
```

代码 13.12 访问伴生类和对象的私有成员

在说到私有或保护访问的时候，Scala 的访问规则给予了伴生对象和类一些特权。类把它所有的访问权限共享给半生对象，**反过来也是如此**。特别的是，对象可以访问所有它的伴生类的私有成员，就好象类也可以访问所有伴生对象的私有成员一样。

举个例子，上面的 Rocket 类可以访问方法 fuel，它在 Rocket 对象中被声明为私有。类似地，Rocket 对象也可以访问 Rocket 类里面的私有方法 canGetHome。

有一个例外，说到 protected static 成员时，Scala 和 Java 的相似性被打破了。Java 类 C 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　结语

本章里，你看到了把程序分割到包里去的基本结构。它提供给你简单并有用的模块化种类，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摩擦的影响。这个系统在精神上与 Java 的包是相同的，但是由于 Scala 选择了更一致和更通用从而有了某些差别。

往下看，第 27 章描述了比分割成包更灵活的模块系统。那种方法除了让你把代码分离到若干命名空间中以外，还允许模块被参数化以及相互继承。下一章，我们将把我们的注意力转向断言和单元测试。

附录 A

Unix 和 Windows 的 Scala 脚本

如果你比较喜欢 Unix 的某些风味，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，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
```

你就可以像 shell 脚本那样运行 Scala 脚本，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，来达到同样效果：

```
::#!
@echo off
call scala %0 %*
goto :eof
::!#
```

附录 B

翻译用词

书中原词	翻译	释义
first-class	第一类	
collection	集合	
set	集	
field	字段	