

O'REILLY®

Scala

学习手册

Learning Scala



Jason Swartz 著
苏金国 杨健康 等译

中国电力出版社

Scala学习手册

Jason Swartz 著
苏金国 杨健康 等译

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

O'Reilly Media, Inc.授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

Scala学习手册/（美）斯瓦茨（Swartz, J.）著；苏金国等译. —北京：中国电力出版社，2016.2

书名原文：Learning Scala

ISBN 978-7-5123-8774-4

I. ①S… II. ①斯… ②苏… III. ①JAVA语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2016）第006593号

北京市版权局著作权合同登记

图字：01-2015-7601号

Copyright © 2015 Jason Swartz, All right reserved.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2016. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2015。

简体中文版由中国电力出版社出版2016。英文原版的翻译得到O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc.的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

封面设计/ Ellie Volckhausen, 张健

出版发行/ 中国电力出版社 (<http://www.cepp.sgcc.com.cn>)

地 址/ 北京市东城区北京站西街19号（邮政编码100005）

经 销/ 全国新华书店

印 刷/ 北京丰源印刷厂

开 本/ 787毫米×980毫米 16开本 14.5印张 273千字

版 次/ 2016年2月第一版 2016年2月第一次印刷

印 数/ 0001—3000册

定 价/ 48.00元（册）

敬 告 读 者

本书封底贴有防伪标签，刮开涂层可查询真伪

本书如有印装质量问题，我社发行部负责退换

版 权 专 有 翻 印 必 究

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

本书献给我挚爱的妻子，感谢她预见了美好的未来。

还要献给我亲爱的女儿，感谢她预见了这本书的问世。

目录

前言	1
----------	---

第一部分 Scala核心

第1章 可扩展语言概述	11
-------------------	----

安装Scala.....	11
使用Scala REPL	12
小结	14
练习	14

第2章 处理数据：字面量、值、变量和类型	16
----------------------------	----

值	18
变量	19
命名	20
类型	22
小结	33
练习	33

第3章 表达式和条件式	34
-------------------	----

表达式	34
if...else表达式块	36
匹配表达式	38
循环	44

小结	48
练习	49
第4章 函数	51
过程	54
用空括号定义函数	54
使用表达式块调用函数	55
递归函数	56
嵌套函数	58
用命名参数调用函数	58
有默认值的参数	59
Vararg参数	60
参数组	60
类型参数	61
方法和操作符	63
编写可读的函数	66
小结	68
练习	68
第5章 首类函数	70
函数类型和值	71
高阶函数	73
函数字面量	74
占位符语法	77
部分应用函数和柯里化	78
传名参数	80
偏函数	81
用函数字面量块调用高阶函数	82
小结	85
练习	85
第6章 常用集合	87
列表、集和映射	87

List里有什么？	90
列表算术运算	94
映射列表	96
归约列表	97
转换集合	102
使用集合的模式匹配	103
小结	105
练习	106
第7章 更多集合	109
可变集合	109
数组	113
Seq和序列	115
Stream	116
一元集合	118
小结	131
练习	132
第二部分 面向对象Scala	
第8章 类	139
定义类	143
更多类类型	148
更多字段和方法类型	150
包装	153
私密性控制	159
私密性访问修饰符	161
最终类和密封类	162
小结	163
练习	163

第9章 对象、Case类和Trait	167
对象	167
Trait	175
导入实例成员	183
小结	184
中场休息——配置你的第一个Scala项目	185
练习	190
第10章 高级类型	197
元组和函数值类	199
隐含参数	201
隐含类	202
类型	204
小结	213
思考题	214
附录A 保留字	217

前言

欢迎学习Scala，也欢迎学习这本书。在这本书中，我将用浅显易懂的方式全面介绍Scala编程语言。

本书的读者对象

这本书主要面向使用过面向对象语言（如Java、Ruby或Python），而且希望通过学习Scala提高编程水平的开发人员。Java开发人员会发现Scala中同样有面向对象、静态类型和泛型集合等概念。不过，他们需要转换到Scala更具有表述性，而且更为灵活的语法，另外要使用不可变数据和函数式面量来解决问题。Ruby和Python开发人员很熟悉使用函数式面量（也就是闭包或块）来处理集合，不过可能需要适应Scala的支持泛型的静态类型系统。

这本书将沿着一个循序渐进的学习曲线，为这些开发人员以及其他希望用Scala编程语言完成开发的人提供一个基于实例的全面指南。

为什么写这本书？

2012年我选择Scala时，发现学习这个语言的过程漫长又艰难，但实际上本不该如此。确实有一些关于Scala的书已经涵盖这种语言的核心特性。不过，我发现很难从Java转换到我们不熟悉的Scala语法、不可变数据结构以及Scala的高度可扩展性。我用了好几个星期才逐渐熟悉如何编写新代码，又用了好几个月来充分理解其他开发人员的代码，最后花了接近一年的时间才搞清楚这种语言的一些更高级的特性。

我下决心写这本书，是为了将来的开发人员学习这种语言能更容易一些，而现在即使利用这本书，学习Scala的过程仍不轻松。学习新技能总是要面临一些挑战，而学习一种新语言，尤其是语法很特别，而且采用了一些新方法的语言，则更需要全心投入，并且要做大量工作。不过，有了这本书，至少可以让这个过程更容易一些。希望它能让更多的开发人员选择Scala，能更好地掌握Scala并把它作为主要开发语言。

为什么学习Scala（或者为什么要看这本书）？

我个人非常喜欢使用Scala开发程序，而且如果有人要编写服务器应用或者适用Java类语言开发的其他类型程序，都会强烈推荐这种语言。如果你的工作领域适合运行Java虚拟机，如Web应用、服务、作业或数据处理，我就会推荐Scala。

下面来解释为什么要采纳这个建议，以及为什么要用Scala开发程序。

原因1——你的代码会更棒

学习Scala，就能使用函数式编程技术使你的应用更稳定，减少由于未预见的副作用所产生的问题。从可变数据结构转换到不可变数据结构，另外从常规方法转换到对环境没有影响的纯函数，你的代码将更安全、更稳定，而且更容易理解。

另外，代码还将更简单，更具有表达性。如果你使用过某种动态语言，如Python、Ruby或JavaScript，应该已经了解使用简短的表达式语法的好处，这可以避免不必要的标点符号，而且可以把映射、过滤和归约操作压缩为简单的单行代码。如果你比较熟悉Java、C#或C++等静态类型语言，使用Scala能避免那些静态语言中的显式类型、标点符号和样板代码。你还能选择其他编译型语言中很少见的表达性语法。

最后一点，你的代码将是强类型的（即使没有指定显式类型），并且支持多重继承以及mixin功能。另外，所有类型不兼容的问题都会在代码运行前捕获。使用静态类型语言的开发人员应该对Scala的类型安全性和高性能不会陌生。使用动态语言的开发人员则可以在使用表达性语言的同时大大提高安全性和性能。

原因2——你会成为一个更棒的工程师

作为一个工程师，如果既能写简短并具有表达性的代码（如Ruby或Python专家），又能提交类型安全的高性能应用（如Java或C++专家），肯定非常出色。我相信，如果读完这本书后开始使用Scala编程，就能写出兼具上述优点的程序。你能充分利用Scala的函数式编程特性，提供类型安全的表达性代码，还可以实现更高的生产力。

学习任何新的编程语言都需要下一番功夫，因为你要选择不同的新技术来解决问题，以及完成算法和数据结构设计，另外还要用全新的语法表述这些新技术。基于这个前提，选择类似Scala的函数式编程语言可以帮助你明确数据可变性、高阶函数和副作用等概念，而且不只是把它们作为书面的一些新想法，而是要了解如何将这些概念应用到当前的编程工作和设计中。你可能会发现，你目前的工作中也许没有必要使用内联函数和静态类型，但是通过使用这些特性，你就能体会到它们的优点和缺点。另外，如果你当前使用的语言可以部分应用这些特性，如Java 8中新增的lambda表达式支持，你就能更好地做好准备。

原因3——你会成为一个更快乐的工程师

不可否认，作为一个你从未见过的人，我不可能知道Scala开发会对你产生什么影响，给出这样的结论可能有些武断。我的意思是，如果你的编程水平提高到一定程度，能轻松地编写代码，这些代码能更好地工作、更易于阅读、更容易调试，比以前运行得更快，而且这样来编写代码花费的时间更短，那么你肯定会更快乐地去做这件事。

当然，编写代码并不是生活的全部。软件工程师的工作计划中可能只有不到一半的时间用来编写代码。

不过，写代码的这段时间会更有意思，你会对你的工作更自豪。这应该足以作为学习新知识的理由。

为什么本书可能不适合你

要知道，Scala素来就有难学的坏名声。这个语言结合了两种显然冲突的软件工程范式：面向对象编程和函数式编程。这种结合会让初学者很迷茫，需要经过一定的练习才能掌握它的语法。另外，Scala还有一个复杂的类型系统，这个类型系统支持在一个特殊的层次上建立定制的类型声明，这在其他语言中很少见。想要弄明白这个类型系统的语法和作用，这很有难度，特别是如果对抽象代数或类型理论没有太多经验，将尤其困难。

如果你没有足够的时间来读这本书并完成书中的练习，或者如果你希望通过更有挑战性或更理论化的方式来学习这个语言，那么这本书可能不适合你。

本书的语法记法

下面给出一个例子，你会在这本书中看到类似下面的语法记法：

```
val <identifier>[: <type>] = <data>
```

这个例子是一个值（value）的定义，这是Scala中的一个变量类型，这种类型不允许重新赋值。这里使用了我自己的非正式记法来定义Scala语言的语法，它比定义语言常用的传统记法更易懂，不过这也有缺点：这种记法不够正式和准确。

这种记法如下：

- 关键字和标点符号像在源代码中一样原样显示。
- 可变项（如值、类型和字面量）用尖括号包围（"<"和">"）。
- 可选部分用中括号包围（"["和"]"）。

例如，在前面的例子中，“val”是一个关键字，“identifier”和“data”是可变项，会随上下文改变，“type”是一个可选项，（如果指定）必须用一个冒号（“：“）与标识符（identifier）分开。

除了这本书之外，强烈建议还应当阅读正式的Scala语言规范。尽管它使用了一种可能很难学的传统语法记法，不过在确定给定特性的具体语法需求时，这种记法仍很有意义。这个规范的官方名字是“The Scala Language Specification”（Odersky, 2011），可以在Scala官方网站找到，也可以在网上搜索。

本书使用约定

以下是本书中使用的排版约定：

斜体 (*Italic*)

指示新术语、URL、email地址、文件名和文件扩展名。

定宽字体 (**Constant width**)

用于程序代码清单，以及在段落中用来指示程序元素，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

定宽粗体 (**Constant width bold**)

显示要由用户输入的命令或其他文字。

定宽斜体 (*Constant width italic*)

用定宽斜体显示的文字要替换为用户提供过的值或由上下文确定的值。

提示： 这表示一个提示或建议。

说明：这表示一个一般说明。

警告：这表示一个警告或警示。

使用代码示例

可以从<http://bit.ly/Learning-Scala-materials>下载补充材料（包括代码示例、练习等）。

这本书的目的就是要帮助你完成工作。一般来讲，如果本书提供了示例代码，你完全可以在你的程序和文档中使用这些代码，不需要联系我们来得到许可，除非你直接复制了大部分的代码。例如，如果你在编写一个程序，使用了本书中的多段代码，这并不需要得到许可。但是出售或发行O'Reilly示例代码光盘则需要得到许可。回答问题时如果引用了这本书的文字和示例代码，这不需要得到许可。但是如果你的产品的文档借用了本书中的大量示例代码，则需要得到许可。

我们希望但不严格要求标明引用出处。引用信息通常包括书名、作者、出版商和ISBN。例如“Learning Scala by Jason Swartz (O'Reilly). Copyright 2015 Jason Swartz, 978-1-449-36793-0”。

如果你认为你在使用代码示例时超出了合理使用范围或者上述许可范围，可以随时联系我们：permissions@oreilly.com。

Safari®图书在线

Safari®图书在线 (www.safaribooksonline.com) 是一个应需而变的数字图书馆，通过图书和视频方式提供世界顶尖作者在技术和商业领域积累的专家经验。

技术专家、软件开发人员、Web设计人员和企业以及有创意的专业人员都使用Safari图书在线作为其主要资源来完成研究、解决问题、深入学习和资质培训。

Safari图书在线为机构、政府部门和个人提供了多种产品组合和定价程序。

订阅者可以在一个可以快捷搜索的数据库中访问多家出版社提供的成千上万种图书、培训视频和正式出版前手稿，如O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones

& Bartlett、Course Technology以及其他数十家出版公司。关于Safari图书在线的更多信息，请访问我们的在线网站。

联系我们

请将关于本书的意见和问题通过以下地址提供给出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

针对这本书，我们还建有一个网页，列出了有关勘误、示例和其他信息。可以通过以下地址访问这个页面：<http://bit.ly/learning-scala>。

如果对这本书有什么意见，或者要询问技术上的问题，请将电子邮件发至：bookquestions@oreilly.com。

要想了解O'Reilly 图书、课程、会议和新闻的更多信息，请访问我们的网站：<http://www.oreilly.com>。

致谢

感谢我的编辑Meghan Blanchette，感谢她为提高这本书的质量并使它能顺利出版做出的不懈努力。还要感谢Simon St. Laurent对这本书的提案以及整个过程的启动提供的帮助和鼓励。

多位卓越的审校人花了大量时间阅读和审阅了这本书的多个版本，没有他们，这本书不可能问世。非常非常感谢大家：Edward Yue Shung Wong、Shannon “JJ” Behrens、Manish Pandit、Devendra Jaisinghani、Art Peel、Ryan Delucchi、Semmy Purewal、Luc Perkins、Robert Geist和Alexander Trauzzi！我从你们身上学到了很多，非常感谢你们所做的一切。

还要感谢Martin Odersky教授以及EPFL和Typesafe的好人们，另外要感谢Scala社区成员创建并改进了这样一个绝妙的语言。

还要感谢我的妻子Jeanne和女儿Oona，她们做出了牺牲并为我提供了精神上的支持，使我能安心写这本书。

最后要感谢我的兄弟Joshua，是他建议我写一本书。Josh，当初我并不知道你想要的是什么，不过现在终于知道了，希望这本书能让你满意。

第一部分

Scala核心

可扩展语言概述

Scala编程语言的名字很容易让人产生遐想，因为这种语言出自瑞士洛桑的École polytechnique fédérale de Lausanne (瑞士洛桑联邦理工学院，EPFL)。Scala的logo是一个环形楼梯，这可能会让你认为它源于La Scala一词，这是意大利语的楼梯，或者可能来自于意大利歌剧院 Teatro alla Scala。实际上，Scala这个名字只是“可扩展语言”(SCALable Language)的缩写，这个词很贴切地表达了它的意图。Martin Odersky教授和他的EPFL团队于2003年创建了这个语言，用来为Java虚拟机(JVM)平台上的函数式编程以及面向对象编程提供一个高性能的开发环境。

既然已经了解了背景故事，下面就来安装Scala，并尝试使用这种语言。

安装Scala

作为一种JVM语言，Scala要求使用Java运行时库。Scala 2.11（你将要使用的版本）至少需要Java 6。不过，为了得到最优的性能，我建议安装Java 8 JDK（也就是Java SE标准环境）。对于大多数平台，都可以从Oracle网站直接下载Java 8 JDK（或者更新的版本）。网站已经提供了安装工具，所以你不需要手动配置PATH变量来完成安装。

安装完成后，从命令行运行`java -version`来验证你的Java版本。下面给出的例子是对Java 8运行这个命令：

```
$ java -version
java version "1.8.0_05"
Java(TM) SE Runtime Environment (build 1.8.0_05-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.5-b02, mixed mode)
```

既然已经安装了Java，下面再来安装Scala。安装Scala有两种方法（安装任何其他编程工具也是一样）：一种是手动安装，这适合那些喜欢修改系统环境变量的命令行高手；另一种就是适合所有其他人的自动安装。

要手动安装Scala，可以从<http://www.scala-lang.org>下载Scala 2.11，把它的"bin"目录增加到你的系统路径。这个发布版本包括Scala运行时库、工具、已编译库和源代码，不过我们需要的最重要的是scala命令。这个命令提供了很多特性，特别是还提供了一个REPL (Read-Eval-Print-Loop) shell，我们将使用这个shell来学习和尝试Scala语言。

可以使用一个包管理器来自动安装Scala，如面向OS X的Homebrew (<http://brew.sh/>)、面向Windows的 Chocolatey (<https://chocolatey.org/>)，或者面向Linux系统的apt-get/Yum (<http://bit.ly/ls-aptget>)。这些包管理器都可以免费获得，它们能查找包、下载和解压缩包，还可以安装包，安装包之后就可以从命令行访问这个包了。在所有这些包管理器中，Scala包可以作为"scala"得到，所以可以用(brew/choco/apt-get-yum) install scala来安装。

安装时，从命令行执行Scala命令。应该可以看到如下的一个欢迎消息（不过你的Scala和Java版本信息可能不同）：

```
$ scala
Welcome to Scala version 2.11.0 (Java HotSpot(TM) 64-Bit Server VM,
Java 1.8.0_05).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

如果看到*Welcome to Scala*消息和scala>提示符，就说明现在你已经在Scala REPL环境中，可以开始编写代码了。

如果找到了这个命令，但是启动时出现了问题，需要首先确保已经正确安装Java命令，而且你的系统路径指向正确的Java版本。

使用Scala REPL

如果你使用过其他REPL shell，如Python的python、Ruby的irb或Groovy的groovysh，你会发现Scala REPL并不陌生。与Python、Ruby和Groovy运行时环境提供的REPL类似，Scala的REPL支持一次计算和执行一行代码，并提供很有帮助的反馈。

如果你没有用过REPL，或者不习惯在IDE或编辑器之外编写代码，可能需要一些实

践练习来了解如何在Scala REPL中开发应用。不过，Scala REPL提供了一种非凡的方法，可以帮助你快速地学习和尝试Scala语言和库，并及时提供响应。你可以输入单行代码来进行计算和编译，你创建的变量将在会话的生命周期一直可用。多行粘贴模式支持输入多行代码一同编译（而不是单独编译），而且可以在任何时间加载外部源代码和库。另外Scala REPL还提供了一个帮助系统，只需要输入:help命令就可以启动这个帮助系统。

下面开始使用REPL，首先来实现所有编程书中经典的第一个练习：“Hello World”应用。启动REPL，确保可以在屏幕上看到scala>提示符：

```
scala>
```

在提示符后面输入`println("Hello, World!")`，再按回车键。REPL会运行你的`println()`命令，并在命令的下一行上输出结果。在这个输出之后的下一行会显示另一个scala>提示符，等待运行下一个新命令。这就是Read（读）、Evaluate（计算）、Print（打印）、Loop（循环）行为，REPL也因此而得名。

下面是REPL中显示的输入和响应：

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

```
scala>
```

祝贺你，你已经成功地编写并执行了Scala代码。

`println()`函数在所有Scala代码中都可用，它向JVM的`stdout`流输出一个消息。在应用服务器中使用时，`stdout`流通常写至一个文件（例如，Tomcat的`'catalina.out'`），但在Scala REPL中，`println()`函数的消息会直接显示在REPL中。

可以使用标准的`readline`式上下箭头在输入行之间前后切换。例如，按向上箭头并按回车键，这会重新运行前一个命令，或者可以按向上箭头并输入一个新消息来显示。REPL历史可以跨会话存储，所以可以退出，再次运行`scala`命令，然后再按向上箭头访问之前运行过的命令。

下面来完成一个简单的算术运算。在一个新提示符下键入`5 * 7`并按回车键，会得到类似下面的输出：

```
scala> 5 * 7
res0: Int = 35
scala>
```

这一次Scala命令并没有显示输出，而是返回了一个值，也就是5和7的乘积。命令返回值时（或者其本身就是值），REPL会把它赋给一个新的常量变量，从而可以在以后的运算中引用这个值。这些“res”变量会顺序编号（“res”可能是“result”的简写），因此总会有一个唯一的容器来存储命令的结果。

现在`res0`包含这个乘法命令的输出，下面来使用这个结果。在一个新提示符下输入`2 * res0`，并按回车键。应该能看到类似下面的结果：

```
scala> 2 * res0
res1: Int = 70
```

```
scala>
```

正如所料，REPL在这个算术表达式中识别出前面创建的`res0`变量，并生成一个新变量`res1`来存储这个新表达式的结果。

小结

通过以上的介绍，希望你了解到，使用Scala REPL可以计算和运行代码，这为学习这个编程语言提供了一个丰富的学习环境。接下来学习这本书时，可以一直打开REPL，用它来验证你学到的所有知识。本书中的所有代码示例都按REPL会话的形式原样显示，一方面可以验证它们能正常工作，并显示输出的结果，另一方面也便于你在自己的REPL会话中复制这些代码。

更棒的是，你可以修改和调整这些代码示例，直到代码无法运行。Scala是一个编译型的静态类型语言，所以REPL（它会在你按下回车键之后编译一行代码）会让你立即知道是否输入了不正确的Scala代码。这会帮助你更快地学习这个语言，而且可以更好地理解决它的语法和特性的限制。

练习

1. `println()`是一种很好的打印字符串的方法，不过你能找到另外一种不使用`println`打印字符串的方法吗？另外，REPL支持哪些类型的数字、字符串和其他数据？
2. 在Scala REPL中，将温度值22.5从摄氏度转换为华氏度。转换公式是 $cToF(x) = (x * 9/5) + 32$ 。
3. 将练习2的结果除以2，再将它转换为摄氏度。不用自己复制粘贴这个值，可以使之前生成的常量变量（例如，“`res0`”）。

4. REPL可以用:`:load <file>`命令从一个外部文件加载和解释Scala代码。创建一个新文件，名为*Hello.scala*，并增加一个命令来打印一条欢迎辞，然后从REPL执行这个命令。
5. 还可以用另一种方法加载外部Scala代码：用“`raw`”（原始）模式把代码粘贴到REPL，代码将在REPL中编译，就好像它们来自一个源文件一样。为此，输入:`:paste -raw`，按回车键，然后粘贴练习4中源文件的内容。退出“粘贴”模式后，应该能看到欢迎辞。

处理数据：字面量、值、变量和类型

这一章我们将介绍Scala的核心数据和变量类型。首先来看字面量、值、变量和类型的定义：

- 字面量（literal）或字面数据是直接出现在源代码中的数据，如数字5、字母'A'和文本“Hello, World”。
- 值（value）是一个不可变的、有类型的存储单元。可以在定义值时为它指定数据，不过不允许重新赋值。
- 变量（variable）是一个可变的、有类型的存储单元。可以在定义变量时为它指定数据，而且可以在任何时间重新赋值。
- 类型（type）是所处理数据的种类（kind），这是对数据的一个定义或分类。Scala中的所有数据都对应一个特定的类型，所有Scala类型都定义为包含方法的类，这些方法将用来处理这些数据。

在Scala中，如果某些值和变量不再使用，将由Java虚拟机的垃圾回收系统自动撤销其中存储的数据。不能手动地撤销值和变量，也不需要这么做。

下面通过在Scala REPL中具体处理数据来熟悉这些概念。Scala值采用语法`val <name>: <type> = <literal>`定义，我们将创建一个名为x、类型为Int的值（Int是"integer"的简写），并将它赋值为字面数字5：

```
scala> val x: Int = 5
x: Int = 5
```

发生了什么？REPL会读取这个值的定义，完成计算，然后重新打印这个值作为确认（重申一次，这是一个Read-Evaluate-Print-Loop shell）。现在这个新值（名为x）已经定义，可以在后面的代码中使用。下面就来使用这个值：

```
scala> x  
res0: Int = 5  
  
scala> x * 2  
res1: Int = 10  
  
scala> x / 5  
res2: Int = 1
```

这3个输入行的语法在Scala中都是合法的，它们都会返回一个整数值。对于每一个输入行，由于将返回一个值，REPL会重复这个值和它的类型，并赋给一个唯一命名的值（scala中的“value”），这些值名从res0开始顺序编号（res是“result”的简写）。可以像显式定义的值一样使用这些“结果”值：

```
scala> res0 * res1  
res3: Int = 50
```

在这里，值res0和res1相乘，返回值50，并存储在名为res3的新值中。

现在来使用变量。与值不同，变量是可变的，可以为它们赋新值，变量用语法var <name>: <type> = <literal>定义。

下面给出使用变量的一个例子：

```
scala> var a: Double = 2.72  
a: Double = 2.72  
  
scala> a = 355.0 / 113.0  
a: Double = 3.1415929203539825  
  
scala> a = 5  
a: Double = 5.0
```

在这个例子中，我们定义变量a的类型为Double，这是一个双精度浮点数。由于这是一个变量，所以接下来可以为它重新赋一个不同的值。

以上只是关于使用Scala值、变量、类型和字面量的一个简单的介绍。在这一章后面的部分我们将更深入地了解这些内容。

值

值是不可变的、有类型的存储单元，按惯例这也是存储数据的默认方法。可以使用val关键字定义一个新值。

语法：定义值

```
val <identifier>[: <type>] = <data>
```

值需要有一个名和所赋的数据，不过不要求有显式的类型。如果未指定类型（也就是说，不包括": <type>"语法），Scala编译器会根据所赋的数据来推导它的类型。

下面是在Scala REPL中定义值及其类型的一些例子：

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

scala> val greeting: String = "Hello, World"
greeting: String = Hello, World

scala> val atSymbol: Char = '@'
atSymbol: Char = @
```

从语法图可以注意到，在值定义中，指定类型是可选的。如果可以根据赋值来推导值的类型（例如，在第一个例子中，字面量20显然是一个整数），在这种情况下，值定义中就可以省略类型。Scala编译器将从赋值判断这个值的类型，这个过程称为类型推导（type inference）。定义值时如果没有指定类型，并不表示这个值无类型；实际上就像在定义中包含了类型一样，已经为它们指定了适当的类型。

下面再来看几个没有指定类型的例子：

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

scala> val greeting = "Hello, World"
greeting: String = Hello, World

scala> val atSymbol = '@'
atSymbol: Char = @
```

在这些例子中，与之前显式指定类型一样，这些值最后会有同样的类型（Int、String和Char）。Scala编译器（通过REPL）可以推导出字面量20对应类型Int、字面量"Hello, World"对应类型String，字面量'@'对应类型Char。

编写代码时可以使用Scala的类型推导，这是一种很有帮助的快捷方法，因为这样就无

需显式地写出值的类型。一般原则是，只有在不会削弱代码可读性的情况下才使用类型推导。如果阅读代码的人无法确定值的类型是什么，在这种情况下，最好还是在值定义中包含显式的类型。

尽管类型推导可以推断出存储数据所用的正确类型，但是不会覆盖你设置的显式类型。如果定义一个值时指定了类型，而这个类型与初始值不兼容，就会得到一个编译错误：

```
scala> val x: Int = "Hello"
<console>:7: error: type mismatch;
  found   : String("Hello")
  required: Int
         val x: Int = "Hello"
```

这里的错误表明不能用Int类型存储String。

变量

在计算机科学里，变量（variable）一词通常表示一个唯一的标识符，对应一个已分配或保留的内存空间，可以在这个空间里存储值，还可以从这个空间获取值。只要保留了内存空间，就可以反复地赋新值。因此，这个内存空间的内容是动态的，或可变的（variable）。

大多数语言中，如C、Java、PHP、Python和Ruby，这正是处理可赋值的命名内存空间的一般模式。变量是动态、可变而且可赋值的（除了一些例外情况，例如用Java的final关键字定义的变量会有一些特殊限制）。

在Scala中，按惯例，值优先于变量，这是因为值可以为源代码带来稳定性和可预测性。定义一个值时，可以确信不论什么代码访问它，它都会保持相同的值。如果在代码段开始时指定一个值，而且直至代码段结束这个值都不会改变，阅读和调试这个代码段就会更为容易。最后一点，有些数据可能在应用的整个生命周期都可用，或者可能要由并发或多线程代码访问，处理这些数据时，不可变的值会更稳定，而且与可能会被意外修改的可变数据相比，也不那么容易出错。

在这本书中，我们的示例代码和练习更倾向于使用值而不是变量。不过，有时变量可能更适用，如存储临时数据或者在循环中累加值的局部变量，这些情况下可能就要使用变量。

前面已经详细地解释了值相对于变量的优点，暂不考虑这个方面，下面来介绍Scala中如何使用变量。

要用var关键字利用给定的名字、类型和赋值来定义一个变量。

语法：定义变量

```
var <identifier>[: <type>] = <data>
```

与值类似，定义变量时可以指定显式的类型，也可以不指定类型。如果没有指定类型，Scala编译器会使用类型推导来确定变量的正确类型。但与值不同的是，任何时间都可以为变量重新赋新数据。

下面给出一个例子，这里定义了一个变量，然后为它重新赋值，也就是把它重新赋值为它本身与另一个数的乘积：

```
scala> var x = 5
x: Int = 5
```

```
scala> x = x * 4
x: Int = 20
```

尽管变量可以重新赋值，但是不能改变为它指定的类型，所以不能将一个变量重新赋值为类型不兼容的数据。例如，如果定义一个类型为Int的变量，然后为它赋一个String值，这就会导致一个编译器错误：

```
scala> var x = 5
x: Int = 5

scala> x = "what's up?"
<console>:8: error: type mismatch;
 found   : String("what\`s up?")
 required: Int
      x = "what's up?"
           ^
```

不过，如果定义一个类型为Double的变量，再赋值为一个Int值，这是可以的，因为Int数可以自动转换为Double数：

```
scala> var y = 1.5
y: Double = 1.5
```

```
scala> y = 42
y: Double = 42.0
```

命名

Scala中的名字可以使用字母、数字和一些特殊的操作符（operator）字符。因此可以

使用标准算术运算符（例如* 和:+）和常量（例如π和φ）取代比较长的名字，从而使代码更有表述性。

Scala语言规范 (<http://bit.ly/ls-scalaref>) 将这些操作符字符定义为 “\u0020-007F和 Unicode Sm [Symbol/Math] (符号/数学) 类别中的所有其他字符……但不包括括号(())和点号。” 中括号（在这段文字中称为“括号”）保留为在类型参数化中使用，点号则保留为用于访问对象（实例化类型）的字段和方法。

下面是结合字母、数字和字符构成Scala合法标识符的规则：

1. 一个字母后跟有0个或多个字母和数字。
2. 一个字母后跟有0个或多个字母和数字，然后是一个下划线(_)，后面是一个或多个字母和数字，或者一个或多个操作符字符。
3. 一个或多个操作符字符。
4. 一个或多个除反引号外的任意字符，所有这些字符要包围在一对反引号中。

说明：与其他名字不同，用反引号包围的名字可以是Scala中的保留字，如true、while、=和var。

下面在REPL中尝试这些命名规则：

```
scala> val π = 3.14159 ①
π : Double = 3.14159

scala> val $ = "USD currency symbol"
$: String = USD currency symbol

scala> val o_0 = "Hmm".
o_0: String = Hmm

scala> val 50cent = "$0.50" ②
<console>:1: error: Invalid literal number
      val 50cent = "$0.50"
                  ^

scala> val a.b = 25 ③
<console>:7: error: not found: value a
      val a.b = 25

scala> val `a.b` = 4 ④
a.b: Int = 4
```

- ① 特殊字符“π”是一个合法的Scala标识符。

- ② 值名“50cent”是不合法的，因为名字不能以数字开头。在这里，编译器最初会把这个名字解析为一个字面量数字，不过解析到字母“c”时会出错。
- ③ 值名“a.b”是不合法的，因为点号不是操作符字符。
- ④ 加上反引号重写这个值就可以解决问题，不过从美学角度讲，使用反引号不太美观。

按惯例，值和变量名应当用小写字母开头，然后其余单词的首字母大写。这通常称为 camel case 记法，尽管并不严格要求这样做，但建议所有 Scala 开发人员都采用这种记法。这有助于与类型和类区分，类型和类也采用 camel case 记法，但会以一个大写字母开头（同样地，这也只是一个惯例，而非严格的规定）。

类型

Scala 包括数值类型（例如 Int 和 Double）和非数值类型（例如 String），可以用来定义值和变量。这些核心类型是所有其他类型的基石，这包括对象和集合，集合本身也是对象，也包括可以处理其数据的方法和操作符。

与 Java 和 C 不同，Scala 中没有基本类型的概念。尽管 Java 虚拟机支持基本整数类型 int 和整数类 Integer，而 Scala 只支持它自己的整数类 Int。

数值数据类型

表 2-1 展示了 Scala 的数值数据类型。

表 2-1：核心数值类型

类型名	描述	大小	最小值	最大值
Byte	有符号整数	1字节	-127	128
Short	有符号整数	2字节	-32768	32767
Int	有符号整数	4字节	-2^{31}	$2^{31}-1$
Long	有符号整数	8字节	-2^{63}	$2^{63}-1$
Float	有符号浮点数	4字节	n/a	n/a
Double	有符号浮点数	8字节	n/a	n/a

说明：参见 `java.lang.Float` 和 `java.lang.Double` 的 API 文档，其中描述了这些浮点数的可计算最大值和最小值。

Scala支持根据类型等级自动将数字从一种类型转换到另一种类型。表2-1中的数值类型按其自动转换等级排序，Byte类型等级最低，可以转换为任何其他类型。

下面来尝试创建不同类型的值，并自动转换为更高等级的类型：

```
scala> val b: Byte = 10
b: Byte = 10

scala> val s: Short = b
s: Short = 10

scala> val d: Double = s
d: Double = 10.0
```

这里的b和s值被赋值为一个有更高等级的新值，所以它们会自动转换（或“向上转换”）为更高等级。

说明：Java开发人员会发现这些类型名很熟悉，它们正是同名的核心JVM类型的包装器（只不过JVM的Integer在Scala中名为Int）。包装JVM类型可以确保Scala和Java互操作，而且Scala可以使用所有Java库。

Scala不允许从较高等级类型自动转换到较低等级类型。这是有道理的，因为如若不然，转换为一个存储空间较小的类型时就会丢失数据。下面来看一个例子，这里试图把一个较高等级类型自动转换为一个较低等级类型，最后会导致一个错误：

```
scala> val l: Long = 20
l: Long = 20

scala> val i: Int = l
<console>:8: error: type mismatch;
 found   : Long
 required: Int
      val i: Int = 1
```

可以选择使用`toType`方法手动完成类型间的转换，所有数值类型都有这样一个方法。尽管转换到一个较低等级类型有可能导致丢失数据，但是如果你很清楚数据与较低等级类型兼容，这个类型转换就会很有用。

例如，下面有一个`Long`值，可以使用`toInt`方法安全地转换为类型`Int`，因为它的数据在`Int`的存储范围内：

```
scala> val l: Long = 20
l: Long = 20

scala> val i: Int = l.toInt
```

```
i: Int = 20
```

除了使用显式类型，还有一种方法是使用Scala的字面量类型记法，直接指定字面量数据的类型。表2-2是指定字面量类型的记法的完整列表。

表2-2：数值字面量

字面量	类型	描述
5	Int	默认认为无修饰（前缀或后缀）的整数字面量是Int类型
0x0f	Int	"0x" 前缀指示十六进制记法
5l	Long	"l" 后缀指示Long类型
5.0	Double	默认认为无修饰（前缀或后缀）的小数字面量是Double类型
5f	Float	"f" 后缀指示Float类型
5d	Double	"d" 后缀指示Double类型

说明：字面量字符不区分大小写

Scala的字面量类型中可以使用小写或大写字符。字面量数字5L与字面量数字5l是相同的。

下面来尝试使用这些字面量，这里用它们赋新值但是不指示类型。Scala REPL将利用类型推导得出各个值的适当类型：

```
scala> val anInt = 5
anInt: Int = 5

scala> val yellowRgb = 0xfffffoo
yellowRgb: Int = 16776960

scala> val id = 100l
id: Long = 100

scala> val pi = 3.1416
pi: Double = 3.1416
```

字符串

String类型表示文本“串”，这是所有编程语言中最常用的核心类型之一。Scala的String建立在Java的String基础上，另外增加了多行字面量和字符串内插等特有的特性。

String字面量要使用双引号，特殊字符要用反斜线转义：

```
scala> val hello = "Hello There"
hello: String = Hello There
```

```
scala> val signature = "With Regards, \nYour friend"
signature: String =
With Regards,
Your friend
```

与数值类型类似，`String`类型也支持使用数学运算符。例如，可以使用相等运算符(`==`)比较两个`String`值。与Java不同，相等操作符(`==`)会检查真正的相等性，而不是对象引用相等性：

```
scala> val greeting = "Hello, " + "World"
greeting: String = Hello, World

scala> val matched = (greeting == "Hello, World")
matched: Boolean = true

scala> val theme = "Na " * 16 + "Batman!" // what do you expect this to print?
```

可以用三重引号创建多行`String`。多行字符串是字面量，所以如果其中使用了反斜线，不要认为它是一个特殊字符的开头：

```
scala> val greeting = """She suggested reformatting the file
| by replacing tabs (\t) with newlines (\n);
| "Why do that?", he asked. """
greeting: String =
She suggested reformatting the file
by replacing tabs (\t) with newlines (\n);
"Why do that?", he asked.
```

字符串内插

如果要根据其他值建立一个`String`，利用字符串相加就很容易做到。下面是在`Float`值前面和后面增加文本来构建的一个`String`：

```
scala> val approx = 355/113f
approx: Float = 3.141593

scala> println("Pi, using 355/113, is about " + approx + ".")
Pi, using 355/113, is about 3.141593.
```

要在`String`中加入值或变量，更直接的一种方法是利用字符串内插（string interpolation），这是一种特殊的模式，采用这种模式可以识别和解析外部值和变量名。Scala的字符串内插记法是在字符串的第一个双引号前面增加一个“`s`”前缀，然后可以使用美元符（`$`）指示外部数据的引用（可以有可选的大括号）。

下面是使用字符串内插的一个例子：

```
scala> println(s"Pi, using 355/113, is about $approx." )
```

```
Pi, using 355/113, is about 3.141593.
```

如果引用中有非字（nonword）字符（如算式），或者如果引用与周围文本无法区分，就需要使用可选的大括号：

```
scala> val item = "apple"
item: String = apple

scala> s"How do you like them ${item}s?"
res0: String = How do you like them apples?

scala> s"Fish n chips n vinegar, ${"pepper "*3}salt"
res1: String = Fish n chips n vinegar, pepper pepper pepper salt
```

字符串内插的替代格式是使用printf记法，想要控制数据格式化时，如字符个数或小数值的显示，这种记法非常有用。要使用printf记法，需要把前缀改为“f”，然后将printf记法紧跟在引用后面：

说明：如果你不熟悉printf，关于printf的格式有很多在线资源，包括java.util.Formatter的官方Javadoc (<http://bit.ly/ls-javadoc>)，它是Scala格式化字符串使用的底层引擎。

```
scala> val item = "apple"
item: String = apple

scala> f"I wrote a new $item%.3s today"
res2: String = I wrote a new app today

scala> f"Enjoying this $item ${355/113.0}%.5f times today"
res3: String = Enjoying this apple 3.14159 times today
```

与前面的例子相比，采用这些printf记法时，引用会比较难读，不过可以对输出提供基本控制。

既然我们已经了解了如何利用字符串控制数据输出，下面来看如何用正则表达式做相反的工作。

正则表达式

正则表达式（regular expression）是字符和标点符号组成的一个字符串，表示一个搜索模式。正则表达式因Perl和诸如Grep等命令行工具得以普及，这是大多数编程语言（包括Scala）库的一个标准特性。

Scala正则表达式的格式基于Java类java.util.regex.Pattern。如果你不熟悉这个类型，建议读一读java.util.regex.Pattern的Javadoc（Java API文档），因为Java（以及Scala）的正则表达式可能与你在其他语言和工具中使用的正则表达式格式有所不同。

String类型提供了很多支持正则表达式的内置操作。表2-3显示了其中的一些操作。

表2-3：正则表达式操作

操作名	示例	描述
matches	"Froggy went a' courting" matches ".* courting"	如果正则表达式与整个字符串匹配，返回true
replaceAll	"milk, tea, muck" replaceAll ("m[^]+k", "coffee")	用替换文本取代所有匹配文本
replaceFirst	"milk, tea, muck" replaceFirst ("m[^]+k", "coffee")	用替换文本取代第一个匹配文本

要对正则表达式完成更高级的操作，需要调用它的r操作符将字符串转换为正则表达式类型。这会返回一个Regex实例，除了提供捕获组支持，还可以处理其他搜索和替换操作。利用捕获组（capture group），可以选择一个给定字符串中的项，根据正则表达式模式将它们转换为本地值。这个模式必须至少包括一个捕获组，由小括号定义，输入则至少包括一个被捕获模式来返回相应的值。

语法：用正则表达式捕获值

```
val <Regex value>(<identifier>) = <input string>
```

下面来尝试捕获值，这里要从前例的输出（见“字符串内插”一节）捕获数字值。我们将使用多行字符串来存储正则表达式模式，因为它们是字面量，所以我们可以直接写反斜线，而无需再增加第二个用来转义的反斜线：

```
scala> val input = "Enjoying this apple 3.14159 times today"
input: String = Enjoying this apple 3.14159 times today

scala> val pattern = """.* apple ([\d.]+) times .*""".r ①
pattern: scala.util.matching.Regex = .* apple ([\d.]+) times .* ②

scala> val pattern(amountText) = input ③
amountText: String = 3.14159

scala> val amount = amountText.toDouble ④
amount: Double = 3.14159
```

- ① 捕获组是单词apple和times之间的数字序列和一个点号。
- ② 完整的正则表达式类型是scala.util.matching.Regex，或者直接写为util.matching.Regex。
- ③ 必须承认，这个格式有点奇怪。包含捕获组匹配结果的新值名为amountText，不过这个名字并不是直接跟在val标识符后面。

- ④ 将文本形式的数量转换为Double，这就得到了数字值。

正则表达式是一种处理文本的简洁而高效的方法，提供了匹配、替换和捕获等操作。如果你还不熟悉正则表达式，很有必要花些时间好好研究正则表达式，因为它们在现代软件开发中的应用非常广泛。

Scala类型概述

这一节将从数字和字符串转向一个更宽泛的领域，我们将介绍核心类型的范围。Scala的所有类型，从数字到字符串以至集合，都属于一个类型层次体系。你在Scala中定义的每一个类也会自动归入这个层次体系。

图2-1 显示了Scala核心（数值和非数值）类型的层次体系。

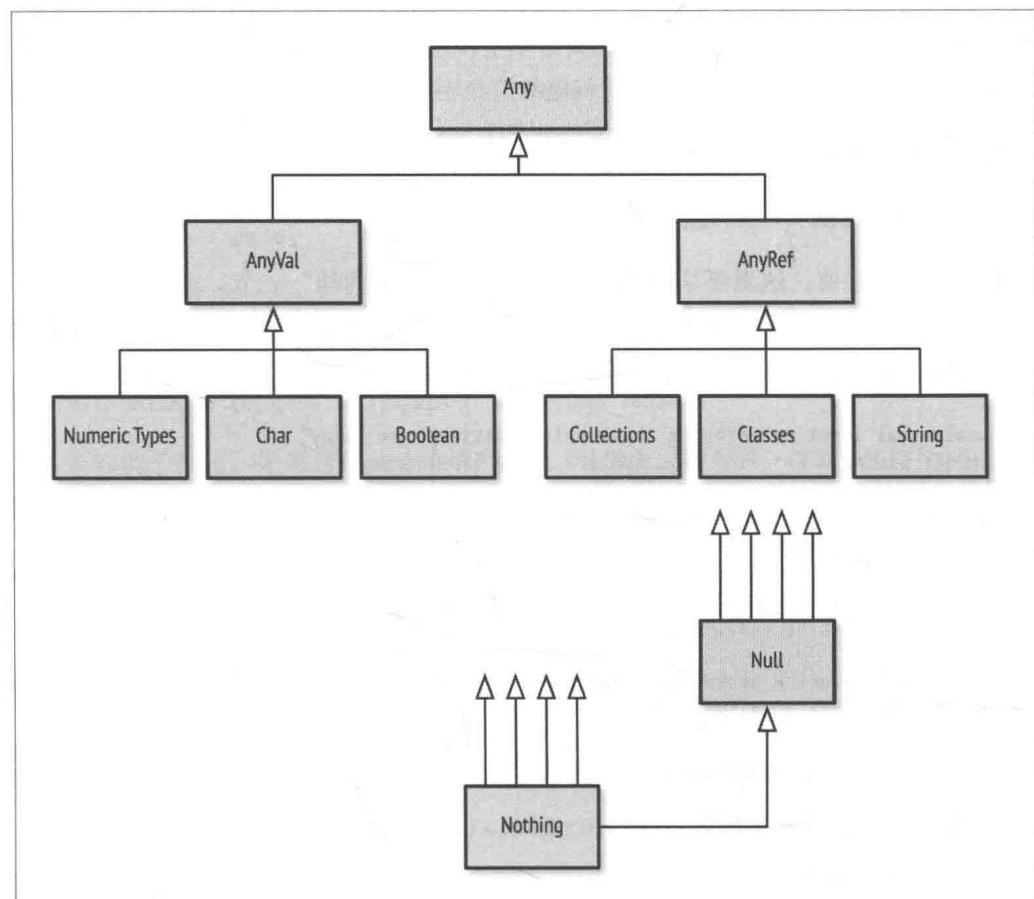


图2-1：Scala类型层次体系

图中的空心三角箭头指示超类型，这是面向对象图的一种常用记法。图下方带多个箭头的类型指示它们是这个系统中所有类型（包括你自己定义的类）的子类型。

表2-4完整地列出了图中提到的特定类型，后面会给出更全面的介绍。

表2-4：核心非数值类型

类型名	描述	可否实例化
Any	Scala中所有类型的根	否
AnyVal	所有值类型的根	否
AnyRef	所有引用（非值）类型的根	否
Nothing	所有类型的子类	否
Null	所有指示null值的AnyRef类型的子类	否
Char	Unicode字符	是
Boolean	true或false	是
String	字符串（即文本）	是
Unit	指示没有值	否

Any、AnyVal和AnyRef类型是Scala类层次体系的根。Any是绝对的根，所有其他类型都由它的两个子类型（AnyVal和AnyRef）派生。扩展AnyVal的类型称为值类型（value types），因为它们是用来表示数据的核心值。这包括我们介绍过的所有数值类型，以及Char、Boolean和Unit。访问AnyVal类型与访问其他类型类似，不过可以在运行时作为对象在堆中分配内存，或者也可以作为JVM基本类型值在栈中分配内存。所有其他类型都以AnyRef作为根，只能作为对象在堆中分配内存。“AnyRef”中的“Ref”指示它们是引用类型，可以通过一个内存引用访问。

Scala类型层次体系最下面是Nothing和Null类型。Nothing是所有其他类型的一个子类型，它的存在只是要为那些影响程序流的操作提供一个兼容的返回类型。例如，return关键字的返回类型就是Nothing（它会带一个返回值提前退出函数），所以可以在初始化一个值的过程中使用，而不会影响这个值的类型。Nothing只能用作为类型，因为它不能实例化。

类型层次体系最下面的另一个类型是Null，这是所有AnyRef类型的一个子类型，它的存在是为了给关键字null提供一个类型。例如，一个String变量可能会在某个时间赋值为null，因此这个变量不会指向内存中的任何字符串实例。将一个声明类型为String的变量赋值为null是可以接受的，因为null是String的一个兼容类型。一般来讲，Scala的语法更优先使用实际类型和实例而不是保留字，例如，专门为null定义一个类型就可以反映出这一点。

`Char`是唯一一个也可以出现在表2-1中的类型。作为`String`类型的基础，`Char`包含单个字符，所以有时被认为是一个文本单位。实际上，这是一个标量类型，可以与其他数字来回转换。

`Char`字面量要使用单引号，与使用双引号的`String`字面量相区别。如果你熟悉ASCII字符编码系统，下面这个例子应该不难理解：

```
scala> val c = 'A'  
c: Char = A  
  
scala> val i: Int = c  
i: Int = 65  
  
scala> val t: Char = 116  
t: Char = t
```

`Boolean`类型只有两个值：`true`和`false`。除了使用`true`和`false`，还可以通过比较和布尔逻辑操作符得到`Boolean`值：

```
scala> val isTrue = !true  
isTrue: Boolean = false  
  
scala> val isFalse = !true  
isFalse: Boolean = false  
  
scala> val unequal = (5 != 6)  
unequal: Boolean = true  
  
scala> val isLess = (5 < 6)  
isLess: Boolean = true  
  
scala> val unequalAndLess = unequal & isLess  
unequalAndLess: Boolean = true  
  
scala> val definitelyFalse = false && unequal  
definitelyFalse: Boolean = false
```

注意：`&`和`&&`有什么区别？

布尔比较操作符`&&`和`||`都很“懒”，如果第一个参数足以得出结论，它们就不会再去计算第二个参数。操作符`&`和`||`则会在返回结果之前对两个参数都进行检查。

与很多动态语言不同，Scala不支持其他类型到`Boolean`的自动转换。非`null`字符串不会计算为`true`，数字`0`也不等于`false`。如果你需要将一个值的状态计算为`Boolean`，就要使用显式的比较：

```
scala> val zero = 0
```

```
zero: Int = 0  
scala> val isValid= zero > 0  
isValid: Boolean = false
```

`Unit`类型与这里的其他核心类型（数值和非数值）不同，它并不是一个指示数据的类型，而是指示没有数据。在某个方面，它类似Java和C中使用的`void`关键字，这个关键字用来定义一个函数不返回数据。`Unit`类型在Scala中作为不返回任何结果的函数或表达式的返回类型。例如，可以认为常用的`println`函数就返回一个`Unit`类型，因为它不返回任何结果。

`Unit`字面量是一对空的小括号`()`，想想看，这确实可以很好地表示没有任何值。如果你愿意，也可以用`Unit`类型定义值或变量，不过，重申一次，`Unit`类型通常都用来定义函数和表达式：

```
scala> val nada = ()  
nada: Unit = ()
```

以上我们已经介绍了核心类型，下面来看这些类型共有的操作。

类型操作

表2-5显示了Scala中所有类型都可用的操作。`toString`和`hashCode`方法是所有JVM实例都有的操作。

表2-5：常用类型操作

操作名	示例	描述
<code>asInstanceOf[<type>]</code>	<code>5.asInstanceOf[Long]</code>	将一个值转换为指定类型的值。如果这个值与新类型不兼容，会导致一个错误
<code>getClass</code>	<code>(7.0 / 5).getClass</code>	返回一个值的类型（即类）
<code>isInstanceOf</code>	<code>(5.0).isInstanceOf[Float]</code>	如果这个值有给定的类型，则返回 <code>true</code>
<code>hashCode</code>	<code>"A".hashCode</code>	返回这个值的散列码，这对于基于散列的集合很有用
<code>to<type></code>	<code>20.toByte; 47.toFloat</code>	转换函数，可以将一个值转换为一个兼容的值
<code>toString</code>	<code>(3.0 / 4.0).toString</code>	将值显示为一个字符串

警告：避免使用ASINSTANCEOF

如果值无法转换为所请求的类型，`asInstanceOf`操作会导致一个错误。为了避免这个操作带来的运行时错误，应当尽可能使用`to<type>`完成类型转换操作。

这一章目前为止介绍的所有类型都是标量值（可能String是个例外），表示单个元素（或者，对于Unit，它表示没有任何元素）。作为对这些标量值的补充，这一章最后会介绍Tuple类型，它可以把两个或多个值收集为一个新的有序的元素。

元组

元组（tuple）是一个包含两个或多个值的有序容器，所有这些值可以有不同的类型。如果你使用过关系型数据库，可能已经熟悉这个概念。在关系型数据库中，表中的一行可以认为是一个元组。如果需要对值逻辑分组，将它们表示为一个统一的单位，元组就很有用。不过，不同于列表和数组，没有办法迭代处理一个元组中的元素。元组的作用只是作为多个值的容器。

创建元组时，值之间要用逗号分隔，并用一对小括号包围。

语法：创建元组

```
( <value 1>, <value 2>[, <value 3>...] )
```

例如，下面是一个包含Int、String和Boolean值的元组：

```
scala> val info = (5, "Korben", true)
info: (Int, String, Boolean) = (5,Korben,true)
```

可以根据元素的索引（从1开始）访问元组中的单个元素（例如，第一个元素的索引为1，第二个元素的索引为2，依此类推）：

```
scala> val name = info._2
name: String = Korben
```

要创建一个大小为2的元组，另一种候选形式是利用关系操作符(->)。这是一种表示元组中键-值对的流行的快捷方式：

```
scala> val red = "red" -> "0xff0000"
red: (String, String) = (red,0xff0000)

scala> val reversed = red._2 -> red._1
reversed: (String, String) = (0xff0000,red)
```

元组提供了一种建立数据结构的通用方法，如果要对离散的元素进行分组以便处理，元组会很有用。

小结

这一章要从头读到尾可能有一些难度，因为你必须在没有学习如何用Scala具体编程的情况下了解所有这些类型和数据。很高兴你做到了！

这一章最奇怪、最让人意想不到的部分是什么？使用关键字声明值和变量定义？使用与先前截然相反的方法定义变量（如果你有Java背景）：先是变量名，然后才是类型？很多代码都可以使用固定的、不可重新赋值的值而不是（可变的）变量？

如果很难理解这些想法，有一个好消息：随着拥有更多的Scala开发经验，你就会发现这些想法其实很自然。最后看来，对于一个设计良好的函数式编程语言，显然这些将成为不二的选择。

现在你应该已经知道如何定义你自己的值和变量，不过我们还没有学习从哪里得到这些有用的数据并保存在值和变量中。下一章中，你将学习这样一些方法，可以使用表达式（expressions）逻辑结构推导和计算这些数据。

练习

- 编写一个新的摄氏度到华氏度转换的程序（使用公式 $(x * 9/5) + 32$ ），把每一步转换的结果保存在单独的值中。你认为这些值的类型分别是什么？
- 修改摄氏度到华氏度转换的公式，返回一个整数而不是浮点数。
- 使用输入值2.7255，生成字符串“`You owe $2.73`”。可以使用字符串内插来实现吗？
- 有没有更简单的方法重写以下代码？

```
val flag: Boolean = false  
val result: Boolean = (flag == false)
```

- 将数字128转换为一个Char、String和Double，然后再转换回Int。你认为还能得到原来的值吗？需要特殊的转换函数吗？
- 使用输入字符串“Frank,123 Main,925-555-1943,95122”和正则表达式匹配来获取电话号码。可以把电话号码的每一部分转换为单独的整数值吗？如何把它存储在一个元组中？

表达式和条件式

这一章重点介绍Scala的表达式、语句和条件式。这本书中使用的表达式（expression）表示在执行后会返回一个值的代码单元。如果使用大括号（{和}）将多行代码收集在一起，也可以认为是一个表达式。这称为一个表达式块（expression block）。

表达式为函数式编程提供了基础，因为利用表达式可以返回数据而不会修改现有的数据（如变量）。这就允许使用不可变数据，这是一个关键的函数式编程概念，这说明新数据会存储在新的值中而不是存储在现有的变量中。当然，函数可以用来返回新数据，不过在某种意义上讲它们只是另一种类型的表达式。

如果所有代码可以组织（或概念化）为一组有层次的表达式，包括一个或多个返回值的表达式，使用不可变数据就很自然。表达式的返回值会传递到其他表达式，或者存储到值中。通过减少变量的使用，就可以减少函数和表达式的副作用。换句话说，它们只作用于你提供的输入，除了返回值之外不会影响任何其他数据。这正是函数式编程的主要目标和优点之一。

表达式

前面已经提到，表达式（expression）是返回一个值的代码单元。

下面先来看Scala中一个简单表达式的例子，这只是一个String字面量：

```
scala> "hello"
res0: String = hello
```

没错，这并不是一个让人印象深刻的表达式。下面来看一个更复杂的表达式：

```
scala> "hel" + 'l' + "o"
res1: String = hello
```

这个例子和前面的例子都是合法的表达式，尽管实现方式不同，但会生成相同的结果。表达式最重要的是它返回的值。表达式的关键就是要返回捕获和使用的值。

用表达式定义值和变量

我们已经见过包含字面量值("hello")和计算值的表达式。之前我们定义值和变量时都赋为字面量值。不过，更准确的说法是将它们赋为表达式的返回值，而不论表达式是字面量（例如5）、计算式还是函数调用。

在此基础上，下面重新给出根据表达式定义值和变量的语法。

语法：使用表达式定义值和变量

```
val <identifier>[: <type>] = <expression>
var <identifier>[: <type>] = <expression>
```

由于字面量值也是一种表达式，这些定义更全面，而且更准确。实际上，表达式也是定义Scala中大多数语法的坚实基础。可以在后面出现的语法注解中查找“<expression>”，这会指示哪里会使用表达式。

表达式块

可以使用大括号（{和}）结合多个表达式创建一个表达式块（expression block）。表达式有自己的作用域，可以包含表达式块中的局部值和变量。块中的最后一个表达式将作为整个表达式块的返回值。

来看一个例子，下面这行代码中有两个表达式，这最好作为一个表达式块：

```
scala> val x = 5 * 20; val amount = x + 10
x: Int = 100
amount: Int = 110
```

我们真正关心的是"amount"，所以下面把包含"x"值的表达式结合到一个块中。我们将用这个返回值来定义"amount"值：

```
scala> val amount = { val x = 5 * 20; x + 10 }
amount: Int = 110
```

块中的最后一个表达式"x + 10"确定了块的返回值。"x"值之前与"amount"在同一级定

义，现在则在块局部定义。这样一来，这个代码就更为清晰，因为使用"x"的目的就是为了定义"amount"，现在可以很清楚地看出这一点。

表达式块可以根据需要跨多行。可以重写前面的例子，不再包含分号，如下所示：

```
scala> val amount = {  
|   val x = 5 * 20  
|   x + 10  
| }  
amount: Int = 110
```

表达式块也可以嵌套，每一级表达式块都有自己的值和变量。

下面是一个简短的例子，这里展示了一个三层嵌套的表达式块：

```
scala> { val a = 1; { val b = a * 2; { val c = b + 4; c } } }  
res5: Int = 6
```

这些示例可能还不能充分说明为什么要使用表达式块。不过，要理解表达式块的语法和组合性，这很重要，因为这一章后面介绍控制结构时还会谈到表达式块。

语句

语句（statement）就是不返回值的表达式。语句的返回类型为Unit，这种类型指示没有值。Scala编程中常用的语句包括println()调用以及值和变量定义。

例如，下面的值定义就是一个语句，因为它不返回任何结果：

```
scala> val x = 1  
x: Int = 1
```

REPL重复了x的定义，但是不会返回可以用来创建新值的具体数据。

与表达式块不同，语句块不返回值。由于语句块没有输出，通常用来修改现有的数据，或者完成应用之外的修改（例如，写至控制台、更新数据库或连接到一个外部服务器）。

if...else表达式块

if...else条件表达式是一个经典的编程构造，可以根据一个表达式的解析结果（true或false）选择一个代码分支。在很多语言中，它的形式为一个"if...else if...else"块，即首先是一个"if"，然后是0个或多个"else if"，最后是一个包揽所有其他情况的"else"语句。

作为练习，你可以在Scala中用同样的方式写这些"if...else if...else"块，它们的工作与Java和其他语言中是一样的。不过，在形式化语法中，Scala只支持一个"if"和可选的"else"块，而不能识别"else if"块。

那么，为什么"else if"块在Scala中还能正常工作呢？这是因为"if...else"块是基于表达式块的，而表达式块很容易嵌套，"if...else if...else"表达式就等价于一个嵌套的"if...else { if...else }"表达式。逻辑上，这完全等同于一个"if...else if...else"块，在语法上，Scala会把第二个"if else"识别为外部"if...else"块的一个嵌套表达式。

下面来分析具体的"if"和"if...else"块，首先来看简单的"if"块的语法。

if表达式

语法：使用if表达式

```
if (<Boolean expression>) <expression>
```

这里的Boolean expression（布尔表达式）指示一个将返回true或false的表达式。

下面是一个简单的if块，如果布尔表达式为true，则会打印一个通知：

```
scala> if ( 47 % 3 > 0 ) println("Not a multiple of 3")
Not a multiple of 3
```

当然，47不是3的倍数，所以这个布尔表达式为true，因此会调用println。

尽管if块可以作为一个表达式，不过更适合类似这样的语句。使用if块作为表达式的问题在于：它们只能有条件地返回一个值。如果布尔表达式返回false，你认为if块会返回什么？

```
scala> val result = if ( false ) "what does this return?"
result: Any = ()
```

这个例子中结果值的类型未指定，所以编译器会使用类型推导来确定最合适的类型。可能会返回一个String或Unit，所以编译器会选择根类Any。这是String（扩展了AnyRef）和Unit（扩展了AnyVal）共同的根类。

与单个的"if"块不同，"if...else"块很适合处理表达式。

if...else表达式

语法：if...else表达式

```
if (<Boolean expression>) <expression>
else <expression>
```

下面来看一个例子：

```
scala> val x = 10; val y = 20
x: Int = 10
y: Int = 20

scala> val max = if (x > y) x else y
max: Int = 20
```

可以看到，整个if和else表达式只包含x和y值。所得到的值将赋给max，我们和Scala编译器都知道这个值是一个Int，因为这两个表达式的返回值类型都是Int。

有人可能会奇怪，为什么Scala没有三元表达式（三元表达式在C和Java中很常见，其中标点符号字符?和:相当于一个单行if和else表达式）。从这个例子可以清楚地看出，实际上Scala并不需要三元表达式，因为它的if和else块可以很紧凑地写在一行上（而且，与C和Java不同，它们已经是表达式了）。

如果在if...else表达式中使用一个表达式而不是表达式块，只要所有代码都能放在同一行上，这也能正常工作。不过，如果你的if...else表达式不能很容易地放在一行上，就应该考虑使用表达式块，这样你的代码会更可读。如果一个if表达式没有相应的else表达式，则必须使用大括号，因为这些语句有可能会带来副作用。

if...else块是编写条件逻辑的一种简单而常用的方法。不过，Scala还有更精巧的方法来实现条件逻辑，即使用匹配表达式（match expressions）。

匹配表达式

匹配表达式（Match expressions）类似C和Java的"switch"语句，首先会计算一个输入项，并执行第一个“匹配”模式，然后返回它的值。与C和Java的"switch"语句类似，Scala的匹配表达式支持一个默认或通配的“全包型”模式。但与它们不同的是，只能有0个或1个模式可以匹配。而不会从一个模式“贯穿”到下一个模式，也不需要"break"语句来避免这种贯穿行为。

传统的"switch"语句仅限于按值匹配，不过Scala的匹配表达式极其灵活，还允许匹配各种各样的其他项，如类型、正则表达式、数值范围和数据结构内容。尽管很多匹配表达式可以用简单的"if...else if...else"块替代，不过这样一来就会失去匹配表达式所提供的简洁语法。

实际上，大多数Scala开发人员更愿意使用匹配表达式而不是"if...else"块，因为匹配表达式更具有表达性，而且语法更简洁。

在这一节中，我们将介绍匹配表达式的基本语法和用法。学习这本书的过程中，你还会了解到可能适用于匹配表达式的一些新特性。可以尝试使用匹配表达式，找出利用匹配表达式表述关系或相等性的新方法。

语法：使用匹配表达式

```
<expression> match {  
    case <pattern match> => <expression>  
    [case...]  
}
```

注意：允许有多个表达式，但不推荐这种做法

Scala官方支持箭头(=>)后面跟有多个表达式，不过并不推荐这种做法，因为这会降低可读性。如果一个case块中有多个表达式，可以通过大括号把它们转换为一个表达式块。

下面来尝试使用匹配表达式，将上一节中的"if...else"示例改为使用匹配表达式。在这个版本中，首先会处理布尔表达式，再将结果与true或false匹配：

```
scala> val x = 10; val y = 20  
x: Int = 10  
y: Int = 20  
  
scala> val max = x > y match {  
|   case true => x  
|   case false => y  
| }  
max: Int = 20
```

这个逻辑实际上与"if...else"块中的逻辑完全相同，只是实现方式不同。

下面再来看一个匹配表达式的例子，将取一个整数状态码，尝试为它返回最合适的消息。取决于表达式的输入，这里除了返回一个值，还可以有其他动作：

```
scala> val status = 500  
status: Int = 500  
  
scala> val message = status match {  
|   case 200 =>  
|     "ok"  
|   case 400 => {  
|     println("ERROR - we called the service incorrectly")  
|   }
```

```
    "error"
}
case 500 => {
    println("ERROR - the service encountered an error")
    "error"
}
}
ERROR - the service encountered an error
message: String = error
```

如果状态码为400或500，除了返回消息"error"外，这个匹配表达式还会打印错误消息。`println`语句很好地展示了如何在一个`case`块中包含多个表达式。一个`case`块中的语句和表达式个数没有任何限制，不过只是最后一个表达式会用作为匹配表达式的返回值。

可以用一个模式替换式（pattern alternative）结合多个模式，其中任何一个模式匹配都会触发`case`块。

语法：模式替换式

```
case <pattern 1> | <pattern 2> .. => <one or more expressions>
```

利用模式替换式（pattern alternative），通过对多个模式重用相同的`case`块，可以避免重复的代码。下面的例子展示了如何使用这些管道符号(|)将7个模式匹配表达式合并为两个模式：

```
scala> val day = "MON"
day: String = MON

scala> val kind = day match {
|   case "MON" | "TUE" | "WED" | "THU" | "FRI" =>
|     "weekday"
|   case "SAT" | "SUN" =>
|     "weekend"
|
| }
kind: String = weekday
```

到目前为止，这些例子都存在一种可能性：很有可能找不到与输入表达式匹配的模式。如果发生这种情况，例如，如果前例的输入是"MONDAY"，你认为会发生什么？

动手尝试比泛泛空谈更有意思，所以下面来看一个例子，这里的匹配表达式没有为输入表达式提供一个匹配的模式：

```
scala> "match me" match { case "nope" => "sorry" }
scala.MatchError: match me (of class java.lang.String)
... 32 elided
```

输入"match me"无法与给定的唯一模式"nope"匹配，所以Scala编译器把它处理为一个运行时错误。错误类型`scala.MatchError`指示了这个错误是由于匹配表达式无法处理其输入。

注意：前例中的消息“… 32 elided”表示这里为了便于阅读压缩了这个错误的栈轨迹（栈轨迹是一个函数调用列表，包含由此向下直至导致这个错误的所有嵌套函数调用）。

为了避免错误破坏匹配表达式，可以使用一个通配的“全匹配”模式（match-all），或者增加足够的模式来涵盖所有可能的输入。将通配模式作为最后一个模式放在匹配表达式中，这会匹配所有可能的输入模式，从而避免`scala.MatchError`发生。

用通配模式匹配

在匹配表达式中可以使用两种通配模式：值绑定和通配符（即“下划线”）。

利用值绑定（value binding）或变量绑定（variable binding），将把匹配表达式的输入绑定到一个局部值，然后可以在case块的体中使用。由于模式包含所绑定的值名，并没有要匹配的具体模式，因此值绑定是一个通配模式，它能匹配任何输入值。

语法：值绑定模式

```
case <identifier> => <one or more expressions>
```

下面是一个例子，这里尝试匹配一个特定的字面量，如果不匹配则使用值绑定确保能匹配所有其他可能的值：

```
scala> val message = "Ok"
message: String = Ok

scala> val status = message match {
    | case "Ok" => 200
    | case other => {
    |   println(s"Couldn't parse $other")
    |   -1
    | }
    | }
status: Int = 200
```

值`other`是为这个case块定义的，并赋为`message`的值（`message`是匹配表达式的输入）。

另一种通配模式是使用通配符。通配符是一个下划线（_）字符，相当于一个匿名占位

符，最后将在运行时替换为一个表达式的值。与值绑定类似，下划线通配符没有提供可以匹配的具体模式，因此这是一个通配模式，可以匹配任何输入值。

语法：通配符模式

```
case _ => <one or more expressions>
```

不能在箭头右侧访问通配符，这与值绑定不同。如果需要在case块中访问通配符的值，可以考虑使用值绑定，或者直接访问匹配表达式的输入（如果可以）。

注意：为什么使用下划线作为通配符？

使用下划线表示未知值的做法来自于数学领域，特别是算术领域，问题中缺少的数通常用一个或多个下划线表示。例如，等式 $5 * \underline{\quad} = 15$ 就是要得出所缺少的值 $\underline{\quad}$ 。

下面的例子与前面类似，不过这里使用了一个通配符而不是绑定值：

```
scala> val message = "Unauthorized"
message: String = Unauthorized

scala> val status = message match {
    |   case "Ok" => 200
    |   case _ => {
    |     println(s"Couldn't parse $message")
    |     -1
    |   }
    | }
Couldn't parse Unauthorized
status: Int = -1
```

在这里，下划线通配符将与运行时匹配表达式的输入值匹配。不过，不能像访问绑定值那样在case块中访问通配符，因此，要用匹配表达式的输入创建包含相关信息的println语句。

用模式哨卫匹配

模式哨卫（pattern guard）向值绑定模式增加一个if表达式，从而可以为匹配表达式增加条件逻辑。使用模式哨卫时，只有当表达式返回true时模式才匹配。

语法：模式哨卫

```
case <pattern> if <Boolean expression> => <one or more expressions>
```

与常规的if表达式不同，这里的if表达式不需要在其布尔表达式两边加小括号

((和))。常规的if表达式需要小括号来简化整个命令的解析，并从条件表达式中区分出布尔表达式。在这里，箭头(=>)就可以完成这个任务来简化解析。不过，如果愿意，也可以在布尔表达式两边加小括号。

下面使用一个模式哨卫区分非null和null响应，并报告正确的消息：

```
scala> val response: String = null
response: String = null

scala> response match {
|   case s if s != null => println(s"Received '$s'")
|   case s => println("Error! Received a null response")
|
}
Error! Received a null response
```

用模式变量匹配类型

利用匹配表达式完成模式匹配还有一种方法，即匹配输入表达式的类型。如果匹配，模式变量（Pattern variables）可以把输入值转换为一个不同类型的值，然后可以在case块中使用这个新值和类型。

语法：指定模式变量

```
case <identifier>: <type> => <one or more expressions>
```

对于模式变量的命名，除了值和变量的有关命名要求外，唯一的限制是它们必须以一个小写字母开头。

既然所有值都有类型，而且它们通常都有很好的描述性，所以你可能考虑使用匹配表达式来确定一个值的类型。Scala中支持多态（polymorphic）类型，这就是一个线索，由此可以看出匹配表达式的作用。类型为Int的值可以赋给另一个类型为Any的值，或者可以从一个Java或Scala库调用作为Any值返回。尽管这个数据实际上是一个Int，但是这个值可以有更高层的类型Any。

下面就来实现这种情况，首先创建一个Int，将它赋给一个Any，再使用一个匹配表达式解析它的真正的类型：

```
scala> val x: Int = 12180
x: Int = 12180

scala> val y: Any = x
y: Any = 12180

scala> y match {
|   case x: String => s"'x'"
```

```
|   case x: Double => f"$x%.2f"
|   case x: Float  => f"$x%.2f"
|   case x: Long   => s"${x}l"
|   case x: Int    => s"${x}i"
|
| }
res9: String = 12180i
```

尽管为匹配表达式指定的值类型为Any，但是它存储的数据是作为Int创建的。匹配表达式能够根据这个值的实际类型来匹配，而不是根据它给定的类型完成匹配。因此，仍能把整数12180（尽管指定为类型Any）正确地识别为一个整数，并相应地格式化。

循环

循环是这一章介绍的最后一个基于表达式的控制结构。循环（loop）一词是指反复地执行一个任务，可能包括迭代处理一个数据范围或者一直重复直至一个布尔表达式返回false。

Scala中最重要的循环结构是for循环（for-loop），也称为for-comprehension。For循环可以迭代处理一个数据范围，每次迭代会执行一个表达式，并返回所有表达式返回值的一个集合（可选）。for循环可以灵活定制，支持嵌套迭代、过滤和值绑定。

首先，我们来介绍一个名为Range（范围）的新的数据结构，可以用来迭代处理一个数字序列。需要使用to或until操作符并指定开始和结束整数来创建范围，to操作符会创建一个包含列表（inclusive list），until操作符则创建一个不包含列表（exclusive list）。

语法：定义数值范围

```
<starting integer> [to|until] <ending integer> [by increment]
```

下面是for循环的基本定义。

语法：用基本for循环迭代处理

```
for (<identifier> <- <iterator>) [yield] [<expression>]
```

yield关键字是可选的。如果表达式中指定了这个关键字，调用的所有表达式的返回值将作为一个集合返回。如果没有指定这个关键字，但是指定了表达式，将会调用这个表达式，但是不能访问它的返回值。

可以用小括号或大括号定义for循环。这两种方式的区别体现在使用多个迭代器时（每行一个迭代器）。或者后面会看到，使用其他合法的for循环项时二者也有区别。如果

for循环使用小括号，最后一个迭代器之前的各个迭代器行必须以一个分号结尾。对于使用大括号的for循环，迭代器行末尾的分号则是可选的。

下面迭代处理一周中的7天（从1到7，包括1和7），打印一个简单的周计划表，并为每一天打印一个表头：

```
scala> for (x <- 1 to 7) { println(s"Day $x:") }
Day 1:
Day 2:
Day 3:
Day 4:
Day 5:
Day 6:
Day 7:
```

这个循环表达式（实际上是一个语句，因为这里没有yield关键字）中的大括号是可选的，因为这里只有一个命令，不过我还是增加了大括号，这样看起来更像传统的Java/C "for" 循环。

不过，如果我实际上需要这些"Day X:"消息的一个集合呢？这样就可以采用其他方式重用这个集合，或者可以根据需要打印很多次。yield关键字可以解决这个问题。可以把迭代语句转换为一个表达式，它会返回各个消息而不是把它们打印出来，再增加yield关键字把整个循环转换为一个返回集合的表达式：

```
scala> for (x <- 1 to 7) yield { s"Day $x:" }
res10: scala.collection.immutable.IndexedSeq[String] = Vector(Day 1:,
 Day 2:, Day 3:, Day 4:, Day 5:, Day 6:, Day 7:)
```

Scalarep的输出比我们预想的要复杂。这里报告了res10的类型是IndexedSeq[String]，这是String的一个索引序列，并赋值为一个Vector，这是IndexedSeq的一个实现。由于Scala支持面向对象多态，所以可以把一个Vector（IndexedSeq的一个子类型）赋给一个IndexedSeq类型的值。

在某种意义上，可以认为这个for循环是一个映射（map），因为它使用了将日期呈现为String的表达式，并把这个表达式应用到输入范围中的每一个成员。我们使用这个循环将从1到7的数字范围映射为相同大小的一个消息集合。类似于其他序列，这个集合现在可以在其他for循环中用作为一个迭代器。

下面创建一个for循环迭代处理我们构建的序列，并打印各个消息，这一次所有消息都显示在同一行上，而不是分行显示。同样地，迭代表达式中只有一个命令，所以这一次我们省略了大括号，因为在这里并不需要大括号：

```
scala> for (day <- res0) print(day + ", ")
```

```
Day 1:, Day 2:, Day 3:, Day 4:, Day 5:, Day 6:, Day 7:,
```

迭代器哨卫

类似匹配表达式中的模式哨卫，迭代器哨卫（iterator guard）也称为过滤器（filter），可以为迭代器增加一个if表达式。使用迭代器哨卫时，可以跳过一次迭代，除非if表达式返回true。

语法：迭代器哨卫

```
for (<identifier> <- <iterator> if <Boolean expression>) ...
```

下面是一个使用迭代器哨卫的例子，这里使用迭代器哨卫来创建一个3的倍数的集合：

```
scala> val threes = for (i <- 1 to 20 if i % 3 == 0) yield i
threes: scala.collection.immutable.IndexedSeq[Int] = Vector(3, 6, 9, 12, 15, 18)
```

迭代器哨卫也可以与迭代器分开，出现在单独的一行上。下面再给出一个for循环的例子，这里迭代器和迭代器哨卫是分开的：

```
scala> val quote = "Faith,Hope,,Charity"
quote: String = Faith,Hope,,Charity

scala> for {
    |   t <- quote.split(",")
    |   if t != null
    |   if t.size > 0
    | }
    | { println(t) }
Faith
Hope
Charity
```

嵌套迭代器

嵌套迭代器（Nested iterators）是增加到一个for循环的额外的迭代器，迭代总数随迭代器个数倍增。之所以称为嵌套迭代器，是因为把它们增加到同一个循环中与写为单独的嵌套循环有相同的效果。由于迭代总数是所有迭代器的乘积，增加一个只迭代一次的嵌套循环不会改变总的迭代次数，但是如果一个嵌套循环根本不迭代（迭代次数为0），则会取消所有迭代，即总的迭代次数也是0。

下面的for循环包含两个迭代器：

```
scala> for { x <- 1 to 2
    |     y <- 1 to 3 }
    | { print(s"($x,$y) ") }
```

```
(1,1) (1,2) (1,3) (2,1) (2,2) (2,3)
scala>
```

由于两个迭代器的乘积是6次迭代，所以print语句会调用6次。

值绑定

for循环中的一个常见做法是基于当前迭代在表达式块中定义临时值或变量。在Scala中，另一种替代方法是在for循环的定义中使用值绑定（value binding），它们的工作是一样的，不过使用值绑定有助于尽可能降低表达式块的规模和复杂性。绑定值可以用于嵌套迭代器、迭代器哨卫和其他绑定值。

语法：for循环中的值绑定

```
for (<identifier> <- <iterator>; <identifier> = <expression>) ...
```

在这个例子中，我们对一个Int使用了“左移”二进制操作符(<<)，来计算2的0次幂到2的8次幂。这个操作符的参数是左移1位的次数，实际上左移1位就是乘以2。每个操作的结果绑定到当前迭代的值"pow"：

```
scala> val powersOf2 = for (i <- 0 to 8; pow = 1 << i) yield pow
powersOf2: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 4, 8,
16, 32, 64, 128, 256)
```

循环的每次迭代中都会定义"pow"值并为它赋值。由于这个值由for循环通过yield返回，结果将是每次迭代得到的"pow"值的一个集合。

通过在for循环定义中使用值绑定，可以把循环的大部分逻辑都集中在定义中。结果将是一个更简洁的for循环，还可以得到一个更为简洁的yield表达式（如果使用了yield）。

While 和Do/While循环

除了for循环，Scala还支持"while"和"do/while"循环，这会重复一个语句直到一个布尔表达式返回false。不过，在Scala中这些循环没有for循环那么常用，因为它们不是表达式，不能用来获得值。

语法：While循环

```
while (<Boolean expression>) statement
```

作为一个非常简单的例子，这里给出一个while循环，将一个数反复递减，直到它不再大于0：

```
scala> var x = 10; while (x > 0) x -= 1
x: Int = 0
```

下面这个"do/while"循环也类似，不过会在第一次计算布尔表达式之前执行语句。在这个例子中，我设置了一个返回false的布尔表达式，但是在语句运行之后才会检查这个布尔表达式：

```
scala> val x = 0
x: Int = 0

scala> do println(s"Here I am, x = $x") while (x > 0)
Here I am, x = 0
```

while和do/while循环也有自己的用途，例如，如果你在读取一个套接字，需要继续迭代处理，直到再没有可读的内容为止。不过，Scala提供了很多比while和do/while循环更有表述性而且功能更强的方法来处理循环。这包括我们已经介绍过的for循环，以及第6章将要学习的新方法。

小结

这一章详细介绍了if/else条件、模式匹配和循环。这些结构为编写Scala中的核心逻辑奠定了坚实的基础。

不过，这3个（或2个）结构不仅对了解表达式的基础很重要，对于学习Scala开发也同样重要。正如这一章的名字所示，表达式及其返回值是所有应用的核心组成构件。表达式本身看起来是一个很简单明了的概念，用整个一章来介绍这个内容似乎有些“小题大做”。之所以我会用一章专门介绍表达式，这是因为学习使用表达式是一个很有用也很有意义的技能。编写代码时要考虑使用表达式，甚至要用表达式来建立应用的结构。编写表达式时要记住一些重要的原则：①如何将代码组织为表达式；②表达式如何得到返回值；③如何处理这个返回值。

除了作为代码组织的基础，表达式还是Scala语法的基础。在这一章中，我们介绍了如何基于表达式定义if/else条件、模式匹配和循环。下一章中，我们还会继续采用这种做法介绍函数，函数就是可重用的命名表达式，我们将介绍如何基于表达式定义函数。后面的各章也会继续沿用这种方式，利用表达式来定义有关概念和结构。因此，理解表达式和表达式块的基本性质和语法对于理解这个语言的语法至关重要。

练习

尽管Scala REPL提供了一个很好的途径来尝试这个语言的特性，不过在REPL中不只是编写一两行代码，而需要编写多行代码，就可能很困难。由于接下来需要使用多行代码，所以现在开始使用单独的Scala源文件。

scala命令可以启动Scala REPL，还可以用来计算和执行Scala源文件：

```
$ scala <source file>
```

可以试试看，创建一个名为*Hello.scala*的新文件，它包含以下内容：

```
println("Hello, World")
```

然后用scala命令执行这个文件：

```
$ scala Hello.scala  
Hello, World  
  
$
```

应该可以看到下一行会打印结果 ("Hello, World")。

要执行外部Scala文件，还有一种方法：可以在Scala REPL中使用:load命令。如果你希望使用Scala REPL的同时还能使用文本编辑器或IDE编辑代码，这会很有用。

下面来试试看，在创建*Hello.scala*文件所在的同一个目录中，启动Scala REPL并运行:load Hello.scala：

```
scala> :load Hello.scala  
Loading Hello.scala...  
Hello, World  
  
scala>
```

注意：:load命令是一个Scala REPL特性，实际上这不是Scala语言的一部分。Scala REPL命令与常规Scala语法的区别在于它们有一个“:”前缀。

现在你可以在Scala REPL中开发，或者也可以在一个单独的文本编辑器或IDE中开发，下面就可以开始完成这一章的练习了。

1. 给定一个字符串名，写一个匹配表达式，如果非空则返回相同的字符串，如果为空就返回字符串"n/a"。

- 给定一个双精度数，写一个表达式，如果这个数大于0则返回"greater"，如果等于0返回"same"，如果小于0则返回"less"。你能用if...else块来写这个表达式吗？使用匹配表达式呢？
- 写一个表达式，将输入值cyan, magenta或yellow转换为相应的6字符十六进制字符串表示。如何处理错误条件？
- 打印数字1到100，每行包含一组5个数。例如：：

```
1, 2, 3, 4, 5,  
6, 7, 8, 9, 10  
...
```

- 写一个表达式，打印数字1到100，不过3的倍数除外，3的倍数要打印为"type"，另外要把5的倍数打印为"safe"，对于3以及5的倍数（如15），则打印"typesafe"。
- 能不能重写练习5，把代码写在一行上？这样做可能可读性不好，不过把代码缩减为最简形式是一门艺术，这也是学习语言的一个很好的练习。

函数

函数是可重用逻辑的核心构件。当然，你可能已经知道这一点，因为几乎所有其他语言都包含函数（或方法，这是面向对象版本的函数）。用整个一章来介绍所有语言都有的一个概念看起来有些奇怪，不过，对于Scala和其他函数式编程语言来说，函数尤其重要。

函数式编程语言特别强调支持创建高可重用、可组合的函数，可以帮助开发人员利用函数组织代码。就像UNIX高手会把多个单用途工具组合在一个复杂的管道命令里一样，使用函数式语言的程序员也会把单用途的函数调用结合在一起构成操作链（例如，可以想想Map/Reduce）。可以选择一个简单函数（如将一个数加倍）应用到一个包含50000个节点的列表，或者指定给一个作用对象在本地执行或者在一个远程服务器上执行。

在Scala中，函数(functions)是可重用的命名表达式。函数可以参数化，可以返回一个值，但这些特性都不是必要的。不过，这些特性对于确保最大程度的可重用性和可组合性很有用，还可以帮助你编写更简短、更可读，而且更稳定的应用。通过使用参数化函数，可以规范化重复的代码，简化逻辑，使你的逻辑更容易理解。测试代码也会更为容易，因为与代码中反复出现的非规范化逻辑相比，规范化和参数化的逻辑测试时会更为容易。

如果遵循标准函数式编程方法论，尽可能构建纯(pure)函数，还会得到更大的好处。在函数式编程中，纯函数是指：

- 有一个或多个输入参数。

- 只使用输入参数完成计算。
- 返回一个值。
- 对于相同的输入总返回相同的值。
- 不使用或影响函数之外的任何数据。
- 不受函数之外的任何数据的影响。

纯函数基本上等价于数学中的函数，定义为仅由输入参数得出的一个计算式，这是函数式编程中的程序基本构件。与不满足这些需求的函数相比，纯函数更稳定，因为它们无状态，而且与外部数据是正交的（如文件、数据库、套接字、全局变量或其他共享数据）。实际上，它们是不可破坏而且非破坏性的纯逻辑表达式。

另一方面，要编写一个有用的应用，而且不希望影响文件、数据库或套接字，这确实可能很困难，所以编写只包含纯函数的应用是很少见的。Scala开发人员可能不是想方设法在应用中完全使用纯函数，而是通常会做出让步，努力寻求减少非纯函数的方法。通常要保证以适当的方式清晰地命名和组织非纯函数，以便与纯函数区分，这是模块化和组织Scala应用的一个常见目标。

记住这一点，下面来学习如何在Scala中编写函数。由于Scala的函数定义很灵活，有很多可选的组件，我们可以首先从最基本的类型开始。

语法：定义无输入的函数

```
def <identifier> = <expression>
```

最基本的Scala函数是表达式的一个命名包装器。如果需要一个函数来格式化当前数据，检查一个远程服务是否提供了新数据，或者只是要返回一个固定的值，就可以采用这个格式。下面是定义和调用无输入函数的一个例子：

```
scala> def hi = "hi"
hi: String
scala> hi
res0: String = hi
```

类似值和变量，即使没有显式地定义返回类型，函数也总会有一个返回类型。另外，同样与值和变量类似，如果确实指定了显式的类型，函数将更易读。

语法：定义函数时指定返回类型

```
def <identifier>: <type> = <expression>
```

这个函数定义也是无输入的，不过从这个定义可以看到：值和变量定义中的“冒号—类型”格式也可以用于函数定义。下面还是来定义"hi"函数，不过这里指定了一个显式的类型，从而有更好的可读性：

```
scala> def hi: String = "hi"
hi: String
```

现在来看一个完整的函数定义。

语法：定义函数

```
def <identifier>(<identifier>: <type>[, ... ]): <type> = <expression>
```

下面试着创建一个函数，来完成一个基本的数学运算：

```
scala> def multiplier(x: Int, y: Int): Int = { x * y }
multiplier: (x: Int, y: Int)Int

scala> multiplier(6, 7)
res0: Int = 42
```

这些函数的函数体基本上由表达式或表达式块组成，在这里最后一行将成为表达式的返回值，相应地也是函数的返回值。我建议函数始终采用这种做法，不过，有些情况下，可能需要在函数的表达式块结束前退出并返回一个值。可以使用`return`关键字来显式指定函数的返回值，然后退出函数。

有时需要提前退出函数，一个常见的用法是在出现不合法或异常的输入值时停止执行。例如，下面的"trim"函数在调用JVM String的"trim"方法之前首先验证输入值是否`null`：

```
scala> def safeTrim(s: String): String = {
    |   if (s == null) return null
    |   s.trim()
    |
safeTrim: (s: String)String
```

现在应该对Scala中如何定义和调用函数有基本的认识了。

警告：要想更多地了解Scala的函数，可以重写第2章和第3章中的一些代码示例，把它们改写为函数。如果可能，可以从示例表达式中去除固定值，在新函数中改写为输入参数。

过程

过程（procedure）是没有返回值的函数。以一个语句（如`println()`调用）结尾的函数也是一个过程。如果有一个简单的函数，没有显式的返回类型，而且最后是一个语句，Scala编译器就会推导出这个函数的返回类型为`Unit`，这表示没有值。对于超过一行的过程，可以显式地指定类型`Unit`，这会让读者清楚地知道这个函数（过程）没有返回值。

下面是一个简单的日志记录过程，首先定义为有一个隐式的返回类型，然后定义为有一个显式的返回类型：

```
scala> def log(d: Double) = println(f"Got value $d%.2f")
log: (d: Double)Unit

scala> def log(d: Double): Unit = println(f"Got value $d%.2f")
log: (d: Double)Unit

scala> log(2.23535)
Got value 2.24
```

关于过程，你可能还会看到另一种语法，即定义时不指定`Unit`返回类型，而且过程体前面没有等号，不过非正式地，这种语法现在已经废弃。如果采用这种语法，示例的`log()`方法可以写为：

```
scala> def log(d: Double) { println(f"Got value $d%.2f") }
```

前面已经指出，维护Scala语言的开发人员已经非正式地废弃了这个语法。这个语法的问题在于，大量开发人员可能会无意地编写有返回值的过程，并希望能把这个返回值返回给调用者。如果采用这个过程语法，所有返回值（或最后的表达式）都会被丢弃。为了解决这个问题，建议开发人员坚持采用常规的函数定义，也就是要使用一个等号，这样就可以尽可能避免合法的返回值被忽略。

用空括号定义函数

要定义和调用一个无输入的函数（即没有输入参数的函数），还有一种方法：可以使用空括号。你会发现这种方式很适用，因为这样可以清楚地区分函数和值。

语法：用空括号定义函数

```
def <identifier>():<type> = <expression>
```

还可以使用空括号调用这样一个函数，也可以不加空括号：

```
scala> def hi(): String = "hi"
hi: ()String

scala> hi()
res1: String = hi

scala> hi
res2: String = hi
```

不过，反过来是不行的。如果定义一个函数时没有加小括号，Scala就不允许在调用这个函数时增加小括号。这个规则可以避免将调用无括号的函数与调用函数返回值相混淆。

注意：有副作用的函数应当使用小括号

对于无输入的函数，Scala有一个约定：如果函数有副作用（也就是说，会修改其范围之外的数据），定义时就应当加空括号。例如，如果一个无输入的函数要向控制台写一个消息，定义时就应当加空括号。

使用表达式块调用函数

使用一个参数调用函数时，可以利用一个用大括号包围的表达式块发送参数，而不是用小括号包围值。通过使用表达式块调用函数，可以完成一些计算和其他动作，然后利用这个块的返回值调用函数。

语法：用表达式块调用函数

```
<function identifier> <expression block>
```

有些情况下，可能更适合使用表达式块来调用函数，例如，必须向函数发送一个计算值时。不必先计算一个量然后把它保存在局部值中再传递给函数，完全可以在表达式块中完成计算。表达式块会在调用函数之前计算，而且表达式块的返回值将用作这个函数的参数。

下面的例子展示了如何在一个表达式块中计算值来调用一个函数：

```
scala> def formatEuro(amt: Double) = f" $amt%.2f"
formatEuro: (amt: Double)String

scala> formatEuro(3.4645)
res4: String = €3.46
```

```
scala> formatEuro { val rate = 1.32; 0.235 + 0.7123 + rate * 5.32 }
res5: String = €7.97
```

如果已经计算出希望传入函数的值，很自然地，可以使用小括号指定函数参数。不过，如果计算式只是用于这个函数，而且可以保证代码的可读性，用表达式块调用函数将是一个很好的选择。

递归函数

递归（recursive）函数就是调用自身的函数，可能要检查某类参数或外部条件来避免函数调用陷入无限循环。递归函数在函数式编程中相当普遍，因为它们为迭代处理数据结构或计算提供了一种很好的方法，而且不必使用可变的数据，因为每个函数调用自己的栈来存储函数参数。

下面是一个递归函数的例子，这个函数用来得到一个整数的给定正数次幂：

```
scala> def power(x: Int, n: Int): Long = {
    |   if (n >= 1) x * power(x, n-1)
    |   else 1
    |
power: (x: Int, n: Int)Long

scala> power(2, 8)
res6: Long = 256

scala> power(2, 1)
res7: Long = 2

scala> power(2, 0)
res8: Long = 1
```

使用递归函数的一个问题是可能会遇到致命的“栈溢出”错误，这表示调用一个递归函数的次数太多，耗尽了所有已分配的栈空间。

为了避免这种情况，Scala编译器可以用尾递归（tail-recursion）优化一些递归函数，使得递归调用不使用额外的栈空间。对于利用尾递归优化的函数，递归调用不会创建新的栈空间，而是使用当前函数的栈空间。只有最后一个语句是递归调用的函数才能由Scala编译器完成尾递归优化。如果调用函数本身的结果不作为直接返回值，而是有其他用途，这个函数就不能优化。

幸运的是，可以利用函数注解（function annotation）来标志一个函数将完成尾递归优化。函数注解是从Java编程语言沿用而来的一种特殊的语法，在函数定义前加一个“at”

符号（@）和一个注解类型来标志它有特殊用途。如果用尾递归函数注解标志一个函数，而它并不能完成尾递归优化，就会在编译时导致一个错误。

要标志一个函数将完成尾递归优化，需要在函数定义前或者在前一行上增加文本@annotation.tailrec。

下面仍给出同一个例子，不过这里增加了"tailrec"注解，这样Scala编译器就会知道我们希望它完成尾递归优化，如果无法优化，编译器要将它处理为一个错误：

```
scala> @annotation.tailrec
| def power(x: Int, n: Int): Long = {
|   if (n >= 1) x * power(x, n-1)
|   else 1
| }
<console>:9: error: could not optimize @tailrec annotated method power:
it contains a recursive call not in tail position
    if (n >= 1) x * power(x, n-1)
```

哎呀，这个函数不能优化，因为递归调用不是这个函数的最后一个语句。这是可以理解的。下面把"if"和"else"条件换一下，再来试试看：

```
scala> @annotation.tailrec
| def power(x: Int, n: Int): Long = {
|   if (n < 1) 1
|   else x * power(x, n-1)
| }
<console>:11: error: could not optimize @tailrec annotated method power:
it contains a recursive call not in tail position
    else x * power(x, n-1)
           ^
```

嗯，现在递归调用确实是函数的最后一项。不过，可以看到，我们得到递归调用的结果后，将它乘以一个值，所以实际上这个乘法才是函数的最后一个语句，而不是递归调用。

要修正这个问题有一个好办法，可以把乘法移到被调用函数的最前面，而不要乘以函数调用的结果。现在函数的最后就是一个未做任何修改的递归调用的简单结果：

```
scala> @annotation.tailrec
| def power(x: Int, n: Int, t: Int = 1): Int = {
|   if (n < 1) t
|   else power(x, n-1, x*t)
| }
power: (x: Int, n: Int, t: Int)Int

scala> power(2,8)
res9: Int = 256
```

成功了！"tailrec"注解和成功的编译可以保证这个函数将完成尾递归优化，所以以后每次调用时不会增加更多的栈帧。

尽管这个例子看起来有些难理解，不过应当知道，递归和尾递归方法对于不使用可变数据完成迭代处理很有意义。你会看到本书后面介绍的很多数据结构都包含大量用尾递归实现的函数。

嵌套函数

函数是命名的参数化表达式块，而表达式块是可以嵌套的，所以函数本身也是可以嵌套的，这应该不奇怪。

有些情况下，需要在一个方法中重复某个逻辑，但是把它作为一个外部方法又没有太大意义。对于这些情况，就可以在函数中定义另一个内部函数，这个内部函数只能在该函数中使用。

下面来看一个方法，它取3个整数，返回其中值最大的整数：

```
scala> def max(a: Int, b: Int, c: Int) = {  
|   def max(x: Int, y: Int) = if (x > y) x else y  
|   max(a, max(b, c))  
| }  
max: (a: Int, b: Int, c: Int)Int  
  
scala> max(42, 181, 19)  
res10: Int = 181
```

`max(Int, Int)`嵌套函数中的逻辑只定义了一次，不过在外部函数中使用了两次，这样可以减少重复的逻辑，简化整个函数。

这里的嵌套函数与外部函数同名，不过，由于它们的参数不同（嵌套函数只有两个整数参数），所以它们之间不会发生冲突。Scala函数按函数名以及其参数类型列表来区分。不过，即使函数名和参数类型相同，它们也不会冲突，因为局部（嵌套）函数优先于外部函数。

用命名参数调用函数

调用函数的惯例是按原先定义时的顺序指定参数。不过，在Scala中，还可以按名调用参数，这样就允许不按顺序指定参数。

语法：按名指定参数

```
<function name>(<parameter> = <value>)
```

在下面的例子中，定义了一个简单的两参数函数并且调用了两次，先照惯例按顺序指定参数，然后按参数名指定参数值：

```
scala> def greet(prefix: String, name: String) = s"$prefix $name"
greet: (prefix: String, name: String)String
```

```
scala> val greeting1 = greet("Ms", "Brown")
greeting1: String = Ms Brown
```

```
scala> val greeting2 = greet(name = "Brown", prefix = "Mr")
greeting2: String = Mr Brown
```

阅读下一节关于默认值的介绍，可以了解按名调用参数非常有用。

有默认值的参数

定义函数时有一个常见的问题：需要确定函数要有哪些输入参数，从而可以最大限度地重用。在Scala、Java和其他语言中，一种常见的解决方案是为一个函数提供多个版本，这些函数同名，但是输入参数表不同。这种做法称为函数重载（function overloading），因为函数名可以用于不同的输入。通常的做法是把一个有 x 个参数的函数复制到一个有 $x - 1$ 个参数的新函数，后者调用前一个有 x 个参数的函数时，缺少的参数将使用一个默认值。

Scala为这个问题提供了一个更简洁的解决方案：可以为任意参数指定默认值，使得调用者可以忽略这个参数。

语法：为函数参数指定默认值

```
def <identifier>(<identifier>: <type> = <value>): <type>
```

下面再来看上一节中的欢迎示例，这里对“prefix”参数使用一个默认值。由于“name”参数仍是必要的，所以我们在调用时只指定这个参数，之所以要按名调用参数，原因是无法按参数顺序来调用（因为“prefix”参数在前！）：

```
scala> def greet(prefix: String = "", name: String) = s"$prefix$name"
greet: (prefix: String, name: String)String
```

```
scala> val greeting1 = greet(name = "Paul")
greeting1: String = Paul
```

这很有用，不过如果调用函数时不必指定参数名就更好了。可以重新组织这个函数，让必要的参数在前，这样就可以直接调用这个函数而不再需要使用参数名：

```
scala> def greet(name: String, prefix: String = "") = s"$prefix$name"
greet: (name: String, prefix: String)String

scala> val greeting2 = greet("Ola")
greeting2: String = Ola
```

从编程风格来讲，最好适当地组织函数参数，让必要参数在前，而有默认值的参数在后。这也强调了必要参数的重要性，另外调用函数时可以不指定默认参数，而且不需要使用参数名。

Vararg参数

Java和C开发人员应该认识vararg，这是一个函数参数，可以匹配调用者的0个或多个实参。最流行的用法是字符串内插函数，如C的printf()和Java的String.format()。

Scala也支持vararg参数，所以可以定义输入参数个数可变的函数。vararg参数后面不能跟非vararg，因为无法加以区分。在函数中，vararg参数（实现为一个集合，这个内容将在第6章介绍）可以用作为for循环中的迭代器。

要标志一个参数匹配一个或多个输入实参，在函数定义中需要该参数类型后面增加一个星号 (*)。

下面来看使用vararg参数创建一个求和函数的例子，这个函数会返回其输入整数的总和：

```
scala> def sum(items: Int*): Int = {
|   var total = 0
|   for (i <- items) total += i
|   total
| }
sum: (items: Int*)Int

scala> sum(10, 20, 30)
res11: Int = 60

scala> sum()
res12: Int = 0
```

参数组

到目前为止，我们已经了解可以对函数定义参数化，并用小括号包围参数表。Scala还

提供了另外一种选择，可以把参数表分解为参数组（parameter groups），每个参数组分别用小括号分隔。

下面来看一个"max"函数的例子，这里的两个输入参数被分解到各自的参数组中：

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

```
scala> val larger = max(20)(39)  
larger: Int = 39
```

在这个例子中，参数组看起来没有太大意义。为什么不把所有参数都放在一个组中呢？对函数字面量使用参数组时才会真正体现出参数组的好处，有关内容在第5章“用函数字面量块调用高阶函数”一节中介绍。

类型参数

到目前为止，我们讨论过的函数参数都是“值”参数，即传入函数的输入数据。在Scala中，作为值参数的补充，还可以传递类型参数，类型参数指示了值参数或返回值使用的类型。通过使用类型参数，可以提高函数的灵活性和可重用性，这样一来，函数参数或返回值的类型不再固定，而是可以由函数调用者设置。

下面给出定义包含类型参数的函数的语法。为了更简单地表示这个语法，我去了掉了返回类型后面的所有内容，另外不再写为通用的“identifier”，这里修改了相应标识来指示每个标识符的具体用途（因为如果不加区分，“def”后的每一项都是标识符）。

语法：定义函数的类型参数

```
def <function-name>[type-name](parameter-name: <type-name>): <type-name>...
```

介绍新特性时，通常我会先指出语法，然后再给出一个例子，不过，由于类型参数是一个很难学的内容，所以这里稍做调整，我们会展示这个特性如何解决一个给定的问题。

警告：有时我会展示一些错误的做法，譬如这里就是如此。这是一个很有用的例子，可以很好地介绍类型参数的作用，不过要提前警告一句：有些示例代码练习并不正确。

假设希望一个函数只返回它的输入（这称为同一性函数（identity）），在这里输入定义为一个String：

```
def identity(s: String): String = s
```

这是可以的，不过只能用于String。没有办法对其他类型（如Int）调用这个函数，除非再定义另外一个函数：

```
def identity(i: Int): Int = i
```

现在我为Int定义了这个函数，不过如果对想要使用的每一个类型都重新定义这个函数，这实在太费劲了。能不能只使用根类型Any？这个Any类型适用所有类型。我们可以试试看，为这个函数传入一个新的String，然后保存返回值：

```
scala> def identity(a: Any): Any = a
identity: (a: Any)Any

scala> val s: String = identity("Hello")
<console>:8: error: type mismatch;
 found   : Any
 required: String
      val s: String = identity("Hello")
                  ^
```

这个例子并不能正常工作。我原来想把结果赋给一个String，但是由于函数的返回类型是Any，所以不能这么做，这导致一个Scala编译错误。

怎么解决这个问题？不必为使用特定的类型（如String或Int）来定义函数，也不要使用一个通用的“根”类型（如Any），可以将类型参数化，使它适用于调用者想要使用的任何类型。

下面是使用类型参数定义的同一性函数，这样它就可以用于你提供的任何类型了：

```
scala> def identity[A](a: A): A = a
identity: [A](a: A)A

scala> val s: String = identity[String]("Hello")
s: String = Hello

scala> val d: Double = identity[Double](2.717)
d: Double = 2.717
```

这个同一性函数的类型参数是A，与值参数a类似，这只是一个唯一的标识符。这里用它来定义值参数a的类型和函数的返回类型。

既然已经用类型参数定义了同一性函数，下面可以用[String]调用这个函数，在函数调用作用域中将值参数类型和返回类型转换为String。然后可以再用[Double]来调用这个函数，在函数调用作用域中将参数类型和返回类型转换为适用于Double值。

当然，我们知道Scala还提供了另一个绝妙的特性：类型推导。在前面的例子中，实

际上没有必要将[String]类型参数传递到这个"identity"方法，因为编译器从我们传入的String字面量可以推导出这一点，或者由于我们将这个函数的返回值赋给一个String值，所以也可以由此推导出这个类型。

下面把前例中两个函数调用中的类型参数删除，从而说明编译器可以推导出类型参数：

```
scala> val s: String = identity("Hello")
s: String = Hello

scala> val d: Double = identity(2.717)
d: Double = 2.717
```

看起来很棒。这里只有一个可以删除的显式类型，即值的类型。对于String和Double类型的输入值，Scala编译器可以推导出类型参数的类型，以及将赋为返回值的值的类型：

```
scala> val s = identity("Hello")
s: String = Hello

scala> val d = identity(2.717)
d: Double = 2.717
```

这里可以看到类型参数和类型推导的意义。向一个函数传入的字面量就足以改变它的值参数类型、返回值类型，以及将赋为返回值的值的类型。

一般来讲，这并不是定义值的最可读的做法，因为读代码的人可能需要仔细检查函数定义才能确定函数返回值将赋给什么类型的值。不过，这确实很好地展示了Scala类型系统的灵活性和强大功能，从中也可以看出它对高可重用函数的支持。

方法和操作符

到目前为止，我们讨论了如何使用函数，但没有指出它们具体在哪里使用。在REPL中单独定义函数时，它们可以帮助学习核心概念。不过，在实际中，函数常存在于对象中，用来处理对象的数据，所以对函数更适合的说法通常是“方法”。

方法（method）是类中定义的一个函数，这个类的所有实例都会有这个方法。Scala（类似Java和Ruby）中调用方法的标准做法是使用中缀点记法（infix dot notation），方法名前面有实例名和一个点（.）分隔符作为前缀。

语法：用中缀点记法调用方法

```
<class instance>.<method>[(<parameters>)]
```

下面来试试看，`String`类型提供了很多很有用的方法，下面来调用其中一个方法：

```
scala> val s = "vacation.jpg"
s: String = vacation.jpg

scala> val isJPEG = s.endsWith(".jpg")
isJPEG: Boolean = true
```

应该可以看到，这个值是类型`String`的一个实例，而`String`类有一个名为`endsWith()`的方法。将来我们会用完整的类名来引用方法，如`String.endsWith()`，不过通常都会用实例名而不是类型名来调用方法。

你会发现，Scala中的大部分类型都提供了丰富的方法可以使用。要成为一个熟练的Scalar开发人员，需要充分学习Scala库，熟悉众多类型及其方法，这是必经之路。官方 Scala API文档 (<http://bit.ly/ls-scalaapi>) 提供了可用类型及其方法的一个完整的列表。强烈建议花些时间来学习你要使用的类型，并尝试使用它们的新方法。

注意：查找`String`类型的文档

`String`类型的文档分为两部分，分别包含在Scala文档的`StringOps`页面和`java.lang.String` Javadocs (<http://bit.ly/ls-string>) 中，这是因为Scala包装了Java的`String`，来提供一些补充的功能。

下面继续研究新方法，这里将尝试使用`Double`类型的一些方法：

```
scala> val d = 65.642
d: Double = 65.642

scala> d.round
res13: Long = 66

scala> d.floor
res14: Double = 65.0

scala> d.compare(18.0)
res15: Int = 1

scala> d.+(2.721)
res16: Double = 68.363
```

`round`和`floor`方法相当简单。它们没有参数，只是对对象中的值返回一个修改后的版本

（当前对象是一个Double，它的值为65.642）。compare方法有一个参数，根据给定的参数小于、等于还是大于d的值，将返回1、0或-1。

最后一个方法名只有一个字符，即一个加号（+），不过这仍是一个合法的函数，它有一个参数，将返回d与这个参数的和。专门提供一个方法来处理加法看起来有些奇怪，因为我们完全可以直接使用加法操作符，不过这个方法实际上正是加法操作符的具体实现。

下面用更浅显的方式来解释。Scala中实际上没有加法操作符，也没有任何其他算术运算符。我们在Scala中使用的所有算术运算符其实都是方法，写为简单的函数，它们使用相应的操作符符号作为函数名，并绑定到一个特定的类型。

之所以可以这么做，是因为还可以采用另一种形式调用对象的方法，这称为操作符记法（operator notation），这里不使用传统的点记法，而是使用空格来分隔对象、操作符方法和方法的参数（只有一个参数）。每次写`2 + 3`时，Scala编译器会把它识别为操作符记法，并相应地处理，就好像写为`2.+3`一样，这里调用了值为2的一个Int的加法方法，并提供参数3，最后会返回值5。

要采用操作符记法调用对象的方法，要求这个方法只有一个参数，另外对象、方法和这个参数之间要用空格分隔。不需要其他的任何标点符号。

语法：用操作符记法调用方法

```
<object> <method> <parameter>
```

对于这种记法，更准确的说法应当是中缀操作符记法（infix operator notation），因为操作符（对象的方法）位于两个操作数中间。

下面再来看前例中的两个方法调用，不过这里使用操作符记法重写这两个调用。前例中的前两个方法不能采用中缀操作符记法，因为它们没有参数：

```
scala> d compare 18.0
res17: Int = 1
```

```
scala> d + 2.721
res18: Double = 68.363
```

结果等价于前例中的结果，这在我们意料之中，因为它们只是用相同的输入值调用了相同的函数。

注意：如果方法有多个参数呢？

类似简单数学运算，操作符记法只适用于单参数方法，不过也可以用于包含多个参数的方法。要做到这一点，需要把参数表包围在小括号里，把它处理为单个（但包装的）参数。例如，可以采用“staring” substring(1,4)的形式调用String.substring(start,end)。

下面是三个数相加。从操作符来看，这里发生了两次相加。如何把它转换为方法调用呢？

```
scala> 1 + 2 + 3
res19: Int = 6
```

答案是，第一个操作符是一个方法调用1 + 2。第二个操作是应用到第一个调用结果的另一个方法调用，或3 + 3。可以使用相同的技术把常规的方法调用串起来，前提是各个方法的结果是一个对象，而且可以在这个对象上调用下一个操作符方法。

Scala支持中缀操作符记法来调用对象方法，这有很多好处。操作符不再作为语法的一部分并以隐含的方式实现，而是作为对象中实现的方法，可以直接查看或调用。因此语法可以简化。开发人员完全可以实现自己的操作符，因为每个单参数方法都可以用作为操作符，而且可以尽量简化方法，使方法只取一个参数，从而允许采用操作符记法。最后一点，代码可读性可以改善，可以去除不必要的标点符号，而把重点放在简单对象、方法和参数组件上。

使用操作符记法只有一个缺点，有时它可能会降低而不是提高代码的可读性。例如，如果串链10个方法调用，它们之间只用空格分隔，与常规的点记法相比，这个代码读起来可能就更为困难，因为操作符和操作数可能很难区分。或者冒失的开发人员定义自己的类型时可能会让加法操作符完成一个完全不同的操作，而调用者对此可能完全不知情。

要确保在清晰可读的前提下才使用操作符记法，你会发现经常要用到这种记法。

编写可读的函数

在这一章的最后，我们对如何编写函数做一个更一般性的讨论。

编写函数的目的就是为了重用（否则，完全可以写为一次性的表达式）。而确保函数可重用性的最好的办法就是保证函数对其他开发人员可读。可读的函数清晰明了、便于理解，而且简单。

确保函数可读有两种方法。首先，尽量保证函数简短、命名适当而且含义明确。把复杂的函数分解为更简单的函数，应当不超过标准的一页纸高度（例如，40行），这样读者就不用上下滚动才能查看整个函数。另外要使用能充分体现函数作用的名字，能清楚地看出这个函数要完成什么工作。如果做到了这两点，对开发人员来说，就能很清楚地了解你的函数的目的和实现。

保证函数可读的另一种方法是在适当的地方增加注释。Scala支持的注释语法与Java和C++相同。双斜线 (//) 开始单行注释 (line comment)，注释会一直到这一行末尾结束。斜线加星号 /*）会开始一个范围注释 (range comment)，这个注释会一直到结束星号加斜线 */ 结束。在函数中可以使用这些注释来指出读者可能遗漏的细节和上下文，另外可以指出可能存在的问题或者将来要完成的工作。

另一类注释是向函数增加Scaladoc首部。Scaladoc工具（包含在Scala包中）可以根据这些函数首部生成API文档。Scaladoc首部遵循与Javadoc首部相同的格式，首先是一个范围注释，不过要用两个星号（例如，/**），然后每一行会缩进并加一个前缀，最后是一个常规的结束范围注释 */。参数可以用一个@param关键字指示，后面是参数名及其描述。

Scaladoc（或Javadoc）首部是函数注释的一个标准格式。为函数增加Scaladoc（或Javadoc）首部是一个很好的做法，即使不打算生成API文档也应当这么做。开发人员读你的函数时，在读具体的函数代码之前可能会从Scaladoc首部开始，所以要确保它准确而简洁。

下面是一个函数的示例Scaladoc首部：

```
scala> /**
| * Returns the input string without leading or trailing
| * whitespace, or null if the input string is null.
| * @param s the input string to trim, or null.
| */
| def safeTrim(s: String): String = {
|   if (s == null) return null
|   s.trim()
| }
safeTrim: (s: String)String
```

函数使用Scaladoc首部还有一个额外的好处：一些IDE对Scaladoc提供了支持（如Eclipse和IntelliJ IDEA）。它们允许开发人员阅读函数的文档，甚至不用通读源代码。开发人员调用你的函数（或方法）时，把鼠标停在函数调用上可以查看Scaladoc首部的内容，或者可以浏览函数列表来查看。

小结

由于第3章已经介绍了这个语言的大多数逻辑结构，在前一章介绍表达式之后，很有必要在这一章强调如何把这些逻辑结构组织为函数以便重用。函数的名、输入参数和返回值类型是函数定义中的重要部分，不过实际上函数的具体内容正是一个很大的表达式。

对于一本专门介绍函数式编程语言的书来说，用一整章介绍函数应当并不奇怪。不过，尽管你已经读完关于函数的这一章，但还没有完全了解关于函数需要知道的所有内容。具体来讲，在Scala中，可以把函数看作是数据，还可以把它们传入其他函数来调用。

函数作为数据，而且有自己的字面量和类型，这个概念使得函数与数据有相同的形式。下一章中你将了解到，可以像处理其他数据类型一样来处理函数，这使得函数也成为这个语言中的“首类”成员。

练习

1. 写一个函数，给定一个圆的半径，计算这个圆的面积。
2. 对练习1中的函数提供一个替代形式，将半径作为一个String。如果调用这个函数时提供了一个空String会发生什么？
3. 写一个递归函数，可以5、10、15……地打印从5到50的值，但不要使用for或while循环。可以实现尾递归吗？
4. 写一个函数，取一个毫秒值，返回一个字符串，按天、小时、分和秒描述这个值。输入值的最佳类型是什么？
5. 写一个函数，计算第一个值以第二个值为指数的幂。首先试着用math.pow来写这个函数，然后用你自己的算式来实现。你使用变量了吗？有没有方法只使用不可变的数据？你选择一个足够大的数值类型了吗？
6. 写一个函数，计算一对2D点(x和y)之差，并把结果返回为一个点。提示：这里很适合使用元组（见第2章“元组”一节）。
7. 写一个函数，取一个大小为2的元组，返回第一个位置上的Int值（如果有）。提示，这里很适合使用类型参数和isInstanceOf类型操作。
8. 写一个函数，取一个大小为3的元组，返回一个大小为6的元组，原来的各个参数后面跟着相应的String表示。例如，用(true, 22.25, "yes")调用这个函数会返

回 (true, "true", 22.5, "22.5", "yes", "yes")。能不能保证所有可能类型的元组都与你的函数兼容？调用这个函数时，能不能使用显式类型做到这一点（除了对函数结果使用显式类型，用来存储结果的值也使用显式类型）？

第5章

首类函数

函数式编程的一个关键是函数应当是首类的（first-class）。 “首类” 表示函数不仅能得到声明和调用，还可以作为一个数据类型用在这个语言的任何地方。首类函数与其他数据类型一样，可以采用字面量形式创建，而不必指定标识符；或者可以存储在一个容器中，如值、变量或数据结构；还可以用作为另一个函数的参数或者另一个函数的返回值。

如果一个函数接受其他函数作为参数，或者使用函数作为返回值，这就称为高阶函数（higher-order functions）。你可能听说过两个最著名的高阶函数：`map()`和`reduce()`。`map()`高阶函数取一个函数参数，用它将一个或多个项转换为一个新值和/或类型。`reduce()`高阶函数取一个函数参数，用它将一个包含多项的集合归约为一项。流行的Map/Reduce计算范式就使用这个概念来解决大型的计算难题，先将计算映射到大量分布式节点上，再归约其结果，达到一个适当的规模。

使用高阶函数处理数据的一个好处是：具体如何处理数据将作为实现细节，留给包含这个高阶函数的框架来完成。调用者可以指定要做什么，让高阶函数处理具体的逻辑流。这种方法实际上有一个名字：声明式编程（declarative programming），通常这与使用函数式编程有关，要求使用高阶函数或其他机制声明要做的工作，而不手动实现。与这种方法相反的是比较强制性的命令式编程（imperative programming），采用这种方式时，总是要明确指定操作的逻辑流。

那么所有这些方法在Scala中如何应用？

Scala对首类函数、高阶函数和声明式编程的使用提供了充分的支持。与其他数据类型一样，如`String`或`Int`，函数也有类型，函数的类型基于其输入参数和返回值的类型。

函数可以存储在值或变量中，可以传入其他函数，可以从另一个函数返回，另外还可以与数据结构结合来支持`map()`、`reduce()`、`fold()`和`filter()`等其他高阶函数。

在这一章中，我们将研究Scala如何使用首类函数、高阶函数以及函数字面量，以便在可以使用常规函数的地方创建和传递逻辑表达式。

函数类型和值

函数的类型（type）是其输入类型和返回值类型的一个简单组合，由一个箭头从输入类型指向输出类型。

语法：函数类型

`([<type>, ...]) => <type>`

到目前为止，我们使用的所有类型都只有一个简单的词，如`String`和`Int`，所以类型中包含标点符号和空白符看起来可能有些奇怪。不过，再仔细想想，如果不使用特定的名字，这实际上是描述一个函数的唯一的方法。由于函数的签名包括函数名、输入和输出，所以函数的类型就是输入和输出。

例如，函数`def double(x: Int): Int = x * 2`的函数类型为`Int=>Int`，这表示它有一个`Int`参数，并返回一个`Int`。函数名"double"是一个标识符，不是类型的一部分。函数体是输入与2的一个简单乘法，这不会影响函数的类型。其余信息就是输入类型和返回类型，所以这构成了函数类型本身。

下面试着在REPL中使用函数类型，这里会创建一个函数，再把它赋给一个函数值：

```
scala> def double(x: Int): Int = x * 2
double: (x: Int)Int

scala> double(5)
res0: Int = 10

scala> val myDouble: (Int) => Int = double      ①
myDouble: Int => Int = <function1>

scala> myDouble(5)                                ②
res1: Int = 10

scala> val myDoubleCopy = myDouble
myDoubleCopy: Int => Int = <function1>

scala> myDoubleCopy(5)                            ③
res2: Int = 10
```

- ① myDouble就是一个值，只不过与其他值不同，可以调用myDouble。
- ② 作为函数调用myDouble与调用double可以得到同样的结果。
- ③ 将一个函数值赋给一个新值，这与其他赋值是一样的。

"myDouble"值必须有显式的类型，以区分出它是一个函数值，而不是一个函数调用。定义函数值以及用函数赋值的另一种做法是使用通配符_。

注意：有单个参数的函数类型可以省略小括号。例如，如果一个函数有一个整数参数，并返回一个整数，类型可以写为Int => Int。

语法：用通配符为函数赋值

```
val <identifier> = <function name> _
```

下面用"myDouble"函数值做些尝试：

```
scala> def double(x: Int): Int = x * 2
double: (x: Int)Int

scala> val myDouble = double _
myDouble: Int => Int = <function1>

scala> val amount = myDouble(20)
amount: Int = 40
```

这里不需要myDouble的显式函数类型来区分函数调用。下划线(_)相当于一个占位符，表示将来的一个函数调用，这会返回一个函数值，我们可以把它保存在myDouble中。

下面再来看显式函数类型，考虑有多个输入的函数。如果函数类型中包含多个输入，则需要在输入类型上显式地加上小括号，这看起来就像是没有参数名的函数定义。

下面给出一个函数值的例子，它有一个显式函数类型，这里使用了多个参数，并用小括号包围：

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

scala> val maximize: (Int, Int) => Int = max
maximize: (Int, Int) => Int = <function2>

scala> maximize(50, 30)
res3: Int = 50
```

这里也可以使用一个通配符来取代显式类型，不过这个例子主要用来展示如何在类型中指定多个参数。

最后，下面给出一个没有输入的函数类型。空的小括号是不是让你想起某个Scala类型？这也是Unit类型的字面量表示（见表2-4），它指示没有值：

```
scala> def logStart() = "=" * 50 + "\nStarting NOW\n" + "=" * 50
logStart: ()String

scala> val start: () => String = logStart
start: () => String = <function0>

scala> println( start() )
=====
Starting NOW
=====
```

这里只是简单地介绍了如何将函数处理为数据，可以把它们存储在值中，也可以赋为静态类型。你可以自己试试这些例子，来熟悉如何指定函数类型以及把函数保存在值中，因为下面几节将介绍高阶函数和函数字面量，这些内容将建立在以上知识基础上，而且会涉及一些很有难度的新语法。接下来会越来越有意思。

高阶函数

我们已经定义了函数类型的值。高阶函数（higher-order function）也是函数，它包含一个函数类型的值作为输入参数或返回值。

下面是高阶函数的一个常见用法：在一个String上调用其他函数，不过前提是输入String非null。增加这个检查后，可以避免在null上调用方法，从而避免JVM中的NullPointerException异常：

```
scala> def safeStringOp(s: String, f: String => String) = {
    |   if (s != null) f(s) else s
    |
safeStringOp: (s: String, f: String => String)String

scala> def reverser(s: String) = s.reverse
reverser: (s: String)String

scala> safeStringOp(null, reverser)
res4: String = null

scala> safeStringOp("Ready", reverser)
res5: String = ydaeR
```

如果输入是"null"，这个调用会安全地返回这个值（"null"），而对于一个合法的String，这个调用会返回输入值的逆序串。

这个例子展示了如何传递一个现有的函数作为高阶函数的参数。用函数作为参数还有另一种做法：可以使用函数字面量内联定义，见下一节的介绍。

函数字面量

下面我们通过一个简单的例子来讨论一个很难的概念，这个概念有很多名字。在这个例子中，我们将创建一个函数字面量（function literal），这是一个能正常工作的函数，但没有名字，然后把这个函数字面量赋给一个新的函数值：

```
scala> val doubler = (x: Int) => x * 2
doubler: Int => Int = <function1>

scala> val doubled = doubler(22)
doubled: Int = 44
```

这个例子中的函数字面量语法为`(x: Int) => x * 2`，它定义了一个有类型的输入参数(x)和函数体($x * 2$)。函数字面量可以存储在函数值和变量中，或者也可以定义为一个高阶函数调用的一部分。任何接受函数类型的地方都可以使用函数字面量。

尽管函数字面量是没有名字的函数，但关于这些概念以及箭头语法的使用有很多名字。你可能知道下面这些名字：

匿名函数 (Anonymous functions)

没错，因为函数字面量确实没有函数名。这是Scala语言中函数字面量的正式名字。

Lambda表达式 (Lambda expressions)

C#和Java 8都采用这种说法，这是从原先数学中的lambda演算（lambda calculus）语法学得来的（例如， $x \rightarrow x^2$ ）。

Lambdas

lambda表达式（lambda expression）的缩写。

`function0, function1, function2, ..`

Scala编译器对函数字面量的叫法，根据输入参数的个数而定。在前例中可以看到，单参数函数字面量就名为`<function1>`。

注意：为什么不直接称它们为匿名函数？

尽管“Scala语言规范”(Odersky, 2011) 使用了匿名函数(anonymous function)一词，但是这个词更多地只是关注函数没有名字，而没有强调这种很有意思的语法，即基于箭头来定义逻辑。所以我使用了“函数字面量”，这种说法更清楚，指示了函数体的整个逻辑都内联指定！可以把函数字面量看作是一个函数值，就像字符串字面量（例如，“Hello, World”）与字符串值的关系一样：函数字面量是所赋数据的一个字面量表达式。

语法：编写函数字面量

```
([<identifier>: <type>, ... ]) => <expression>
```

下面来定义一个函数值，并赋一个新的函数字面量：

```
scala> val greeter = (name: String) => s"Hello, $name"  
greeter: String => String = <function1>  
  
scala> val hi = greeter("World")  
hi: String = Hello, World
```

仔细想想看，函数字面量实际上就是参数化表达式。我们已经了解表达式可以返回一个值，现在又有了一种方法可以参数化表达式的输入。

下面来看一个更长的例子，这里将对函数赋值和函数字面量做一个比较。首先来看这一章引言里提到的max()函数，把它赋给一个函数值，然后再重新将这个max()函数实现为函数字面量：

```
scala> def max(a: Int, b: Int) = if (a > b) a else b ①  
max: (a: Int, b: Int)Int  
  
scala> val maximize: (Int, Int) => Int = max ②  
maximize: (Int, Int) => Int = <function2>  
  
scala> val maximize = (a: Int, b: Int) => if (a > b) a else b ③  
maximize: (Int, Int) => Int = <function2>  
  
scala> maximize(84, 96)  
res6: Int = 96
```

- ① 原来的max()函数。
- ② .. 赋至一个函数值。
- ③ .. 用函数字面量重新定义。

函数字面量不一定需要输入参数。下面来定义一个不需要参数的函数字面量。我们将把另一个函数重写为函数字面量，这一次将实现这一章引言里提到的logStart()函数：

```
scala> def logStart() = "=" * 50 + "\nStarting NOW\n" + "=" * 50
logStart: ()String

scala> val start = () => "=" * 50 + "\nStarting NOW\n" + "=" * 50
start: () => String = <function0>

scala> println( start() )
=====
Starting NOW
=====
```

注意到了吗？REPL将函数字面量称为“function0”，这是无输入的函数的名字。不过，它不是值类型，实际上类型推导为`() => String`，这是一个返回字符串的无输入函数。

前面已经指出，可以在更高阶函数调用中定义函数字面量。来看一个例子，我们将用一个函数字面量调用“safeStringOp”示例（见本章前面“高阶函数”一节）：

```
scala> def safeStringOp(s: String, f: String => String) = {
    |   if (s != null) f(s) else s
    | }
safeStringOp: (s: String, f: String => String)String

scala> safeStringOp(null, (s: String) => s.reverse)
res7: String = null

scala> safeStringOp("Ready", (s: String) => s.reverse)
res8: String = ydaeR
```

函数“safeStringOp”接收一个函数值参数，名为“f”，并有条件地调用这个函数。调用它的常规函数值与我们的函数字面量没有区别。

在这个例子中，函数参数“f”的类型为`String => String`。由于已经定义了这个类型，所以可以从函数字面量中删除显式类型，因为编译器能很容易地推导出它的类型。如果删除显式类型，这意味着我们可以从函数字面量中去除小括号，因为对于单个无类型的输入没有必要加小括号。

下面采用这种更简单的语法用函数字面量调用“safeStringOp”函数：

```
scala> safeStringOp(null, s => s.reverse)
res9: String = null

scala> safeStringOp("Ready", s => s.reverse)
res10: String = ydaeR
```

这里的函数字面量去掉了显式类型和小括号，这个简化体现了它的函数本质。这些函数字面量取一个输入参数，并根据对这个参数的一个操作来返回一个值。

这些函数字面量是很简单的函数表达式，不过通过占位符语法（placeholder syntax），Scala还支持更简单的表达式。

占位符语法

占位符语法（Placeholder syntax）是函数字面量的一种缩写形式，将命名参数替换为通配符（_）。可以在以下情况使用：①函数的显式类型在字面量之外指定；②参数最多只使用一次。

下面的例子将一个函数字面量加倍，这里使用通配符取代命名参数：

```
scala> val doubler: Int => Int = _ * 2
doubler: Int => Int = <function1>
```

在这里，占位符语法是合法的，因为输入参数只使用一次，而且字面量类型有一个外部的显式定义（在值中定义）。

来看另一个例子，下面用占位符语法调用"safeStringOp"示例：

```
scala> def safeStringOp(s: String, f: String => String) = {
    |   if (s != null) f(s) else s
    |
    | }
safeStringOp: (s: String, f: String => String)String

scala> safeStringOp(null, _.reverse)
res11: String = null

scala> safeStringOp("Ready", _.reverse)
res12: String = ydaeR
```

函数字面量的体在操作上等同于`s=>s.reverse`，不过通过占位符语法得到简化。输入参数`s`的引用被替换为一个通配符（_），表示函数的第一个输入参数。实际上，这个通配符就是一个`String`输入参数。

下面通过一个例子来说明占位符的顺序有什么影响，这个例子使用了两个占位符：

```
scala> def combination(x: Int, y: Int, f: (Int,Int) => Int) = f(x,y)
combination: (x: Int, y: Int, f: (Int, Int) => Int)Int

scala> combination(23, 12, _ * _)
res13: Int = 276
```

这里使用了两个占位符，这确实会使语法更为抽象。要记住，它们会按位置替换输入参数（分别是`x`和`y`）。

如果在这里使用一个额外的占位符，将会导致一个错误，因为占位符数必须与输入参数个数一致。如果用1个或3个占位符来调用reduce方法，就会导致一个错误。

最后来改变占位符的个数，从2个改为3个。这个例子还可读吗？

```
scala> def tripleOp(a: Int, b: Int, c: Int, f: (Int, Int, Int) => Int) = f(a,b,c)
tripleOp: (a: Int, b: Int, c: Int, f: (Int, Int, Int) => Int)Int

scala> tripleOp(23, 92, 14, _ * _ + _)
res14: Int = 2130
```

tripleOp函数有4个参数：3个Int值和一个函数，这个函数可以把这3个Int值归约为1个Int。具体的函数体比参数表还要短得多，将把这个函数应用到输入值。

这个示例函数tripleOp仅适用于整数值。如果它是通用的并支持类型参数，可能会更有用。

下面使用两个类型参数重新定义tripleOp函数，一个表示通用的输入类型，另一个表示返回值类型。这可以提供灵活性，这样一来，我们可以使用任何类型的输入或者我们选择的匿名函数（只要这个匿名函数有3个输入）来调用tripleOp函数：

```
scala> def tripleOp[A,B](a: A, b: A, c: A, f: (A, A, A) => B) = f(a,b,c)
tripleOp: [A, B](a: A, b: A, c: A, f: (A, A, A) => B)B

scala> tripleOp[Int,Int](23, 92, 14, _ * _ + _)
res15: Int = 2130

scala> tripleOp[Int,Double](23, 92, 14, 1.0 * _ / _ / _)
res16: Double = 0.017857142857142856

scala> tripleOp[Int,Boolean](93, 92, 14, _ > _ + _)
res17: Boolean = false
```

如果你没有使用过Scala，这个语法确实有点难以理解。不过，读完这一章的内容并完成本章后面的练习之后，你就会习惯使用占位符了。

占位符语法在处理数据结构和集合时尤其有帮助。很多核心的排序、过滤和其他数据结构方法都会使用首类函数和占位符语法来减少调用这些方法所需的额外代码。

部分应用函数和柯里化

调用函数时（包括常规函数和高阶函数），通常要在调用中指定函数的所有参数（包含默认参数值的函数例外）。如果你想重用一个函数调用，而且希望保留一些参数不想再次输入，该怎么做呢？

要给出这个问题的答案，下面将使用一个两参数的函数作为例子，这个函数会检查给定的数是否是另一个数的因数：

```
scala> def factorOf(x: Int, y: Int) = y % x == 0
factorOf: (x: Int, y: Int)Boolean
```

如果需要这个函数的一个快捷方式，所有参数都不打算保留，可以使用这一章引言里介绍的通配符`(_)`赋值：

```
scala> val f = factorOf _
f: (Int, Int) => Boolean = <function2>

scala> val x = f(7, 20)
x: Boolean = false
```

如果想保留一些参数，可以部分应用（partially apply）这个函数，使用通配符替代其中一个参数。在这里通配符需要一个显式类型，因为它要用于生成一个函数值（包含所声明的输入类型）：

```
scala> val multipleOf3 = factorOf(3, _: Int)
multipleOf3: Int => Boolean = <function1>

scala> val y = multipleOf3(78)
y: Boolean = true
```

这个新的函数值`multipleOf3`是一个部分应用函数，因为它包含部分参数而不是`factorOf()`函数的全部参数。

要部分应用函数，还有一种更简洁的方法：可以使用有多个参数表的函数。不是将一个参数表分解为应用参数和非应用参数，而是应用一个参数表中的参数，另一个参数表不应用。这种技术称为函数柯里化（currying）：

```
scala> def factorOf(x: Int)(y: Int) = y % x == 0
factorOf: (x: Int)(y: Int)Boolean

scala> val isEven = factorOf(2) _
isEven: Int => Boolean = <function1>

scala> val z = isEven(32)
z: Boolean = true
```

从函数类型来讲，有多个参数表的函数可以认为是多个函数的一个链。单个参数表则认为是一个单独的函数调用。

示例函数`def factorOf(x: Int, y: Int)` 的函数类型为`(Int, Int) => Boolean`，而更新的示例函数`"def factorOf(x: Int)(y: Int)"` 的函数类型为`Int => Int =>`

`Boolean`。通过柯里化，函数类型变成第二个串链函数`Int => Boolean`。在前面的例子中，函数值"even"将串链函数的第一部分柯里化为整数值2。

如果想有些创意，可以写你自己的函数字面量来处理部分应用函数和柯里函数完成的工作。可以在函数字面量中保留一个可重用的参数，用它和新参数调用一个新函数，这个工作并不复杂。部分应用函数和柯里函数的好处是可以采用一种有表述性的语法。

传名参数

我们已经讨论了使用函数值作为参数的高阶函数。函数类型参数还有一种形式：传名（by-name）参数。传名参数可以取一个值，也可以取最终返回一个值的函数。由于同时支持使用值来调用以及使用函数来调用，所以如果函数有一个传名参数，将由调用者来决定究竟选择使用值还是使用函数调用。

语法：指定传名参数

```
<identifier>: => <type>
```

每次在函数中使用一个传名参数时，它会计算为一个值。如果向这个函数传入一个值，不会有任何作用，不过，如果向它传入一个函数，那么每次使用时都会调用这个函数。

将一个函数传入传名参数时，一定要清楚反复调用这个函数可能带来的开销。例如，如果一个表达式要搜索数据库并返回值，倘若只使用一次，只是向函数传入一个固定的值，性能可能还可以接受。但是，如果这个表达式要用于一个传名参数，它就会成为一个函数值，每次在方法中访问这个参数时都会调用这个函数。

与值或函数参数不同，使用传名参数的主要好处在于它们提供的灵活性。有传名参数的函数在使用值时可以使用，另外需要使用函数时也可以使用。尽管多个参数访问意味着多次调用函数参数，但也可以反过来。如果一个函数传入了一个传名参数，倘若不访问这个参数，就不会调用这个函数，这样就可以在必要时避免开销很大的函数调用。

下面试着调用有一个传名参数的函数。我们先用常规的值调用这个函数，然后再用一个函数来调用，从而验证每次访问参数时都会调用这个函数：

```
scala> def doubles(x: => Int) = { ❶
|   println("Now doubling" + x)
|   x * 2
```

```
| }  
doubles: (x: => Int)Int  
  
scala> doubles(5) ②  
Now doubling 5  
res18: Int = 10  
  
scala> def f(i: Int) = { println(s"Hello from f($i)"); i }  
f: (i: Int)Int  
  
scala> doubles( f(8) ) ③  
Hello from f(8)  
Now doubling 8  
Hello from f(8) ④  
res19: Int = 16
```

- ① 这里可以访问x传名参数，就像访问常规的传值参数一样。
- ② 用一个常规值调用doubles方法，它将正常操作。
- ③ …不过用一个函数值调用这个方法时，会在doubles方法中调用这个函数值。
- ④ 由于double方法引用了x参数两次，所以“Hello”消息会调用两次。

偏函数

目前为止我们研究的所有函数都称为全函数（total functions），因为它们能正确地支持满足输入参数类型的所有可能的值。类似`def double(x: Int) = x*2` 的简单函数就可以认为是一个全函数；没有`double()`函数不能处理的参数`x`。

不过，有些函数并不能支持满足输入类型的所有可能的值。例如，如果一个函数返回输入数的平方根，如果这个输入数为负数，它就不能工作。类似的，如果一个函数要除以一个给定的数，倘若这个数为0，函数也无法工作。这种函数称为偏函数（partial functions），因为它们只能部分应用于输入数据。

Scala的偏函数是可以对输入应用一系列case模式的函数字面量，要求输入至少与给定的模式之一匹配。调用一个偏函数时，如果所使用的数据不能满足其中至少一个case模式，就会导致一个Scala错误。

注意：偏函数和部分应用函数有什么区别？

这两个术语看起来、听上去几乎是一样的（偏函数为partial function，部分应用函数是partially applied function），所以很多开发人员会把它们混淆。偏函数与全函数不同，只接受所有可能输入值的一部分。而部分应用函数是一个部分调用的常规函数，而且将来有可能完全调用。

下面来看一个匹配表达式例子（选自第3章“匹配表达式”一节），并作为一个新的偏函数重用：

```
scala> val statusHandler: Int => String = {  
|   case 200 => "Okay"  
|   case 400 => "Your Error"  
|   case 500 => "Our error"  
| }  
statusHandler: Int => String = <function1>
```

现在我们有了一个函数字面量，它只能应用于值为200、400和500的整数。先用合法的输入值来测试：

```
scala> statusHandler(200)  
res20: String = Okay  
  
scala> statusHandler(400)  
res21: String = Your Error
```

如果用与以上任何一个case模式都不匹配的整数来调用这个函数字面量，你觉得会发生什么？

```
scala> statusHandler(401)  
scala.MatchError: 401 (of class java.lang.Integer)  
  at $anonfun$1.apply(<console>:7)  
  at $anonfun$1.apply(<console>:7)  
  ... 32 elided
```

这会导致一个MatchError，因为尽管输入值有正确的Int类型，但是无法匹配这个偏函数的任何一个case模式。

偏函数看起来像是一个奇怪的特性，因为如果偏函数无法应用，就会导致类似这样的错误。要避免这些错误，一种方法是在末尾使用一个通配模式来捕获所有其他错误，不过这样一来，“偏函数”这个叫法就不合适了。你会发现偏函数在处理集合和模式匹配时更为有用。例如，可以“收集”一个集合中由给定偏函数接受的所有项。

用函数字面量块调用高阶函数

我们介绍过不使用小括号或空格而是用表达式块调用函数（见第4章“使用表达式块调用函数”一节）。高阶函数可以重用这种记法，除了小括号，还可以使用函数字面量块来调用高阶函数，或者用函数字面量块取代小括号。用函数名和一个大的表达式块调用的函数将这个块作为一个函数字面量，它会调用0次或多次。这种语法的一种常见用法是用一个表达式块调用工具函数。例如，一个高阶函数可以把给定的表达式块放在一个数据库会话或事务中。

我们将使用"safeStringOps"函数来展示什么时候需要使用这个语法，以及如何使用。首先，下面先给出使用常规函数字面量的"safeStringOps"函数，后面再把它转换为我们要用的语法：

```
scala> def safeStringOp(s: String, f: String => String) = {  
|   if (s != null) f(s) else s  
| }  
safeStringOp: (s: String, f: String => String)String  
  
scala> val uuid = java.util.UUID.randomUUID.toString ①  
uuid: String = bfe1ddda-92f6-4c7a-8bfc-f946bdac7bc9  
  
scala> val timedUUID = safeStringOp(uuid, { s =>  
|   val now = System.currentTimeMillis ②  
|   val timed = s.take(24) + now ③  
|   timed.toUpperCase  
| })  
timedUUID: String = BFE1DDDA-92F6-4C7A-8BFC-1394546043987
```

- ① (与所有JDK类一样) 可以从Scala访问Java的java.util包中的UUID工具。
- ② System.currentTimeMillis提供了纪元时间 (自1970年1月1日GMT时间以来的经过时间)，单位为毫秒，这对创建时间戳很有用。
- ③ take(x)方法从String返回前x项，在这里就是UUID的前4部分。

在这个例子中，除了传入一个值参数，还向函数传入了一个多行函数字面量。这是允许的，不过把它们包在同一个括号块里有些怪异。

可以做些改进，将"safeStringOp"中的参数分为两个单独的组（见第4章“参数组”一节）。第二个参数组（包含函数类型）可以用表达式块语法来调用：

```
scala> def safeStringOp(s: String)(f: String => String) = {  
|   if (s != null) f(s) else s  
| }  
safeStringOp: (s: String)(f: String => String)String  
  
scala> val timedUUID = safeStringOp(uuid) { s =>  
|   val now = System.currentTimeMillis  
|   val timed = s.take(24) + now  
|   timed.toUpperCase  
| }  
timedUUID: String = BFE1DDDA-92F6-4C7A-8BFC-1394546915011
```

现在有了一个更清晰的safeStringOp调用，值参数放在小括号里传入，而函数参数作为一个独立的函数字面量块传入。

下面再来看另一个例子，这个函数有一个传名参数。我们要让这个函数更通用，使用一个类型参数作为这个传名参数返回类型以及主函数的返回类型：

```

scala> def timer[A](f: => A): A = {
|   def now = System.currentTimeMillis  ②
|   val start = now; val a = f; val end = now
|   println(s"Executed in ${end - start} ms")
|   a
| }
timer: [A](f: => A)A

scala> val veryRandomAmount = timer { ③
|   util.Random.setSeed(System.currentTimeMillis)
|   for (i <- 1 to 100000) util.Random.nextDouble ④
|   util.Random.nextDouble
| }
Executed in 13 ms
veryRandomAmount: Double = 0.5070558765221892

```

- ① 类型参数“A”使得“f”传名参数的返回类型成为“timer”函数的返回类型，从而减少用“timer”函数包围代码的影响。
- ② 这个内部嵌套函数只是为了美观，这样我们可以采用简洁的方式获取当前毫秒数。
- ③ 最后，将高阶函数的表达式块语法归约为最简单的形式：函数名和块。可以看到两个大括号之间的代码是一个表达式块，或者是一个函数字面量块，或者是由“timer”函数包围的常规代码。
- ④ 这行代码会生成并删除100000个随机的浮点数。这对展示计时问题很有用，不过我不建议在生产代码中使用这种方法。

这里使用“timer”函数包围了一个单独的代码单元，不过也可以集成到一个现有的代码基中。可以用它来包围任何函数的最后一部分，测量其性能，同时确保从代码块传递的函数返回值经过“timer”并由函数返回。

函数可以采用这种方式将单独的代码块包围在工具函数中，这也是使用“表达式块”型高阶函数调用的主要好处。使用这种调用还有另外一些好处，包括：

- 管理数据库事务，即高阶函数打开会话、调用函数参数，然后用一个commit或rollback结束事务。
- 重新尝试处理可能的错误，将函数参数调用指定次数，直到不再产生错误。
- 根据局部、全局或外部值（例如，一个数据库设置或环境变量）有条件地调用函数参数。

与Scala中的很多其他特性一样，可以有多种方法来调用高阶函数。我发现使用这种语

法与使用传统的小括号完全不同，不过对于何时使用以及在哪里使用，最重要的原则是要看对你是否合适。

小结

Scala将函数作为首类数据类型，本章就介绍了这个内容，并包括高阶函数、函数字面量和函数类型等概念。简单地讲，在真正使用首类函数之前，你可能会发现这个概念很难理解。如果你还没有用过首类函数，强烈建议先完成代码示例，试着写一些基于首类函数的代码。通过这些练习，你会对函数保存为数据以及使用它们调用高阶函数更为熟悉，在此之后，你会发现下面的练习可以帮助你更深入地了解这个很有难度的内容。

不过，用我们目前学到的数据类型还不能充分展示高阶函数的妙处和作用。要真正展示这个内容，还需要了解编写所有数据驱动代码都不可缺少的一个重要部分。接下来我会谈到集合，这是0到多个元素组成的一个数据结构，可以收集给定类型的多个值。从列表到映射，Scala不仅支持你熟悉的数据结构，还提供了丰富的高阶函数可以最大程度地提高你的生产力。我们不仅会介绍创建和迭代处理集合，还会说明如何使用`map()`、`reduce()`和`filter()`用表达性很强的代码来管理集合。

从现在开始，你会看到首类函数和高阶函数开始在代码示例和练习中扮演主导角色。不论是用我们目前学过的数据类型来展示，还是用基于高阶函数的集合库来展示，它们都是Scala语言中的耀眼的明星。

练习

1. 写一个函数字面量，取两个整数，并返回较大的一个数。然后写一个高阶函数，取一个大小为3的整数元组并结合这个函数字面量，用它返回这个元组中的最大值。
2. 库函数`util.Random.nextInt`会返回一个随机整数。用它调用“`max`”函数，提供两个随机整数和一个函数（这个函数返回两个给定整数中较大的一个）。利用另一个函数（返回两个给定整数中较小的一个）做同样的处理，然后再提供另一个函数每次都返回第二个整数。
3. 写一个高阶函数，它取一个整数并返回一个函数。返回的函数有一个整数参数（如“`x`”），并返回`x`与传入这个高阶函数的整数的乘积。
4. 假设阅读另一个开发人员的代码时看到下面这个函数：

```
def fzero[A](x: A)(f: A => Unit): A = { f(x); x }
```

这个函数会完成什么工作？能不能给出一个例子来展示如何调用这个函数？

5. 有一个名为“square”的函数，希望把它存储在一个函数值中。这样做合适吗？还有什么办法可以把一个函数存储在一个值中？

```
def square(m: Double) = m * m
val sq = square
```

6. 编写一个名为“conditional”的函数，它取一个值x和两个函数p和f，返回与x类型相同的一个值。p函数是一个谓词，取值x，并返回一个布尔值b。f函数也取值x，并返回相同类型的一个新值。这个“conditional”函数只在p(x)为true时才调用函数f(x)，否则就返回x。“conditional”函数需要多少个类型参数？
7. 还记得第3章中的“typesafe”练习吗？面试开发人员时有一个常见的问题，我把它称为“typesafe”，也就是要打印数字1到100，每行一个数。关键是3的倍数必须替换为“type”，5的倍数必须替换为“safe”。当然，如果是15的倍数，则必须打印“typesafe”。

使用练习6中的“conditional”函数来完成这个挑战。

如果“conditional”的返回类型与参数x的类型不匹配，你的答案能不能更简短？

修改“conditional”函数，使它更适合解决这个问题。

常用集合

集合（collections）框架提供了一些数据结构来收集给定类型的一个或多个值，如数组、列表、映射、集和树。大部分流行的编程语言都有自己的集合框架（或者，至少有列表和映射），因为这些数据结构是当前软件项目的基本构件。

“集合”一词因Java集合库得到普及，这是一个高性能的面向对象的类型参数化框架。由于Scala是一个JVM语言，所以可以从Scala代码访问和使用整个Java集合库。当然，如果这么做，可能就无法得到Scala本身的集合高阶操作所带来的种种好处。

与Java一样，Scala有一个高性能的面向对象的类型参数化集合框架。不过，Scala的集合还提供了一些高阶操作，如map、filter和reduce，可以用简洁而且富有表述性的表达式来管理和处理数据。它还有单独的可变和不可变的集合类型层次体系，从而可以很方便地在不可变数据（为保持稳定性）和可变数据（必要时）之间转换。

所有可迭代的集合的根是Iterable，它提供了一组公共方法，（可以想见）用来迭代处理和管理集合数据。现在就来讨论其中最流行的不可变集合。

列表、集和映射

下面先来看List类型，这是一个不可变的单链表。可以作为一个函数调用List来创建一个列表，并以逗号分隔参数的形式传入列表的内容：

```
scala> val numbers = List(32, 95, 24, 21, 17)
numbers: List[Int] = List(32, 95, 24, 21, 17)

scala> val colors = List("red", "green", "blue")
```

```
colors: List[String] = List(red, green, blue)
scala> println(s"I have ${colors.size} colors: $colors")
I have 3 colors: List(red, green, blue)
```

所有集合和String实例上都有size方法，这个方法会返回集合中的项数。如果定义时没有加小括号（见第4章“用空括号定义函数”一节），只需要直接调用size方法（只有方法名，而不加小括号）。

在第4章中，你已经了解了函数如何使用类型参数来参数化其输入值和返回值的类型。集合的类型也可以参数化，从而确保集合能记住并遵循初始化时指定的类型。在前例中就可以看出这一点，其中REPL将类型参数化的集合显示为List[Int]和List[String]。

可以使用Lisp风格的head()和tail()方法分别访问一个列表的首元素和其余元素。要直接访问单个元素，可以作为一个函数来调用这个列表，并传入该元素的索引（从0开始）：

```
scala> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)

scala> colors.head
res0: String = red

scala> colors.tail
res1: List[String] = List(green, blue)

scala> colors(1)
res2: String = green

scala> colors(2)
res3: String = blue
```

在第3章“循环”一节中，你已经了解了Range集合，这是一个连续的数字范围，而且已经了解如何用一个for循环迭代处理这样一个范围。实际上，for循环也很适合迭代处理列表（实际上，同样适合处理任何其他集合）。

下面尝试使用for循环来迭代处理"numbers"和"colors"列表：

```
scala> val numbers = List(32, 95, 24, 21, 17)
numbers: List[Int] = List(32, 95, 24, 21, 17)

scala> var total = 0; for (i <- numbers) { total += i }
total: Int = 189

scala> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)
```

```
scala> for (c <- colors) { println(c) }
red
green
blue
```

在第5章中，你了解了如何将函数作为数据并传入高阶函数。Scala的集合大量使用了高阶函数来完成迭代、映射（将一个列表逐项地转换为一个不同的列表）、归约（将一个列表“折叠”为单个元素）以及其他很有用的操作。

下面的例子展示了List和其他集合中提供的foreach()、map()和reduce()高阶函数。这些函数分别用来迭代处理列表、转换列表以及将列表归约为一项。这些方法分别需要传入一个函数字面量，包括用小括号包围的输入参数和函数体：

```
scala> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)

scala> colors.foreach( c: String => println(c) ) ①
red
green
blue

scala> val sizes = colors.map( c: String => c.size ) ②
sizes: List[Int] = List(3, 5, 4)

scala> val numbers = List(32, 95, 24, 21, 17)
numbers: List[Int] = List(32, 95, 24, 21, 17)

scala> val total = numbers.reduce( a: Int, b: Int => a + b ) ③
total: Int = 189
```

- ① `foreach()`取一个函数（更准确地讲是一个过程），对列表中的每一项分别调用这个函数。
- ② `map()`取一个函数，将一个列表元素转换为另一个值和/或类型。
- ③ `reduce()`取一个函数，将两个列表元素结合为一个元素。

`Set`是一个不可变的无序集合，只包含不重复的唯一元素，不过其工作与`List`类似。下面的例子中，首先用重复的项创建一个`Set`（可以看到，所创建的`Set`中并不包含重复的元素）。作为`Iterable`的一个子类型，与`List`实例类似，`Set`实例也支持同样的操作：

```
scala> val unique = Set(10, 20, 30, 20, 20, 10)
unique: scala.collection.immutable.Set[Int] = Set(10, 20, 30)

scala> val sum = unique.reduce( a: Int, b: Int => a + b )
sum: Int = 60
```

`Map`是一个不可变的键/值库，在其他语言中也称为散列映射（`hashmap`）、字典（`dictionary`）或关联数组（`associative array`）。在`Map`中，值按一个给定的唯一键存储，可以使用这个键来获取相应的值。键和值都可以类型参数化，像创建从整数到字符串的映射一样，也可以很容易地创建从字符串到整数的映射。

创建`Map`时，指定键-值对为元组（见第2章“元组”一节）。可以使用关系操作符（`->`）来指定键和值元组。

下面的例子是一个颜色名到颜色值的`Map`，使用关系操作符对来构建。与`Set`类似，`Map`类型也是`Iterable`的一个子类型，所以支持与`List`同样的操作：

```
scala> val colorMap = Map("red" -> 0xFF0000, "green" -> 0xFF00,
  "blue" -> 0xFF)
colorMap: scala.collection.immutable.Map[String,Int] =
Map(red -> 16711680, green -> 65280, blue -> 255)

scala> val redRGB = colorMap("red")
redRGB: Int = 16711680

scala> val cyanRGB = colorMap("green") | colorMap("blue")
cyanRGB: Int = 65535

scala> val hasWhite = colorMap.contains("white")
hasWhite: Boolean = false

scala> for (pairs <- colorMap) { println(pairs) }
(red,16711680)
(green,65280)
(blue,255)
```

这一节中，我们认识了常用的集合`List`、`Map`和`Set`，它们都是根类型`Iterable`的子类型。我们还了解了`Iterable`及其子类型中提供的一些方法：`foreach()`、`map()`和`reduce()`。不过，目前还只是接触了这些集合的一点皮毛。

在这一章后面，我们将分析这些常用集合的结构和操作，为保持一致性，下面将特别强调`List`类型。

List里有什么？

创建`List`或其他类型的集合时，标准的做法是作为一个函数来调用这个集合，并提供必要的内容：

```
scala> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)
```

可以在集合中存储任何类型的值，而不只是目前为止我们所用的数字和字符串。例如，可以创建一个集合的集合：

```
scala> val oddsAndEvents = List(List(1, 3, 5), List(2, 4, 6))
oddsAndEvents: List[List[Int]] = List(List(1, 3, 5), List(2, 4, 6))
```

或者可能有一个元组（大小为2）的集合，创建一个看上去类似Map的List：

```
scala> val keyValues = List(('A', 65), ('B', 66), ('C', 67))
keyValues: List[(Char, Int)] = List((A,65), (B,66), (C,67))
```

要访问列表中的单个元素，可以作为一个函数调用这个列表，并提供一个索引号（从0开始）。下面的例子会按索引访问一个List中的第1个和第4个元素：

```
scala> val primes = List(2, 3, 5, 7, 11, 13)
primes: List[Int] = List(2, 3, 5, 7, 11, 13)
```

```
scala> val first = primes(0)
first: Int = 2
```

```
scala> val fourth = primes(3)
fourth: Int = 7
```

可以把一个列表分解为表头（head）和表尾（tail），表头即列表的第一项，表尾是列表中的其余项：

```
scala> val first = primes.head
first: Int = 2
```

```
scala> val remaining = primes.tail
remaining: List[Int] = List(3, 5, 7, 11, 13)
```

List是一个不可变的递归数据结构，所以列表中的每一项都有自己的表头和越来越短的表尾。可以利用这一点来创建你自己的List迭代器，从表头开始，延续到后面的表尾。

创建这样一个迭代器时，比较困难的是需要确定何时到达列表末尾。我们可以试着检查if `list.size > 0`，不过，由于这是一个链表，所以size方法每次都必须遍历到列表末尾。幸运的是，列表还提供了一个`isEmpty`方法，可以使用这个方法确定列表是否已经为空，而无需遍历列表。

下面是用一个while循环构建的迭代器，它会遍历列表，直到`isEmpty`返回true：

```
scala> val primes = List(2, 3, 5, 7, 11, 13)
primes: List[Int] = List(2, 3, 5, 7, 11, 13)
```

```
scala> var i = primes
i: List[Int] = List(2, 3, 5, 7, 11, 13)

scala> while(! i.isEmpty) { print(i.head + ", "); i = i.tail }
2, 3, 5, 7, 11, 13,
```

或者，也可以采用递归形式，下面的函数会遍历列表，不过这里没有使用可变的变量：

```
scala> val primes = List(2, 3, 5, 7, 11, 13)
primes: List[Int] = List(2, 3, 5, 7, 11, 13)

scala> def visit(i: List[Int]) { if (i.size > 0) { print(i.head + ", ");
    visit(i.tail) } }
visit: (i: List[Int])Unit

scala> visit(primes)
2, 3, 5, 7, 11, 13,
```

这个递归函数可以算是一个代表，`List`（以及一般的`Iterable`集合）中很多方法都采用类似的方式实现。除此以外，大多数情况下，它们都写为函数，可以把返回值收集到一个结果或一个新的列表中。

调用`isEmpty`检查是否到达列表末尾是可以的，不过还有一种更高效的方法来完成这个检查。所有列表都有一个`Nil`实例作为终结点，所以迭代器可以通过比较当前元素和`Nil`来检查是否到达列表末尾：

```
scala> val primes = List(2, 3, 5, 7, 11, 13)
primes: List[Int] = List(2, 3, 5, 7, 11, 13)

scala> var i = primes
i: List[Int] = List(2, 3, 5, 7, 11, 13)

scala> while(i != Nil) { print(i.head + ", "); i = i.tail }
2, 3, 5, 7, 11, 13,
```

`Nil`实际上是`List[Nothing]`的一个单例实例。应该记得（见表2-4），`Nothing`类型是所有其他Scala类型的一个不可实例化的子类型。因此，`Nothing`类型的列表与所有其他类型的列表都兼容，可以安全地用作为它们的终结点。

创建一个新的空列表时，实际上会返回`Nil`而不是一个新实例。因为`Nil`是不可变的，所以它和一个全新的空列表实例实际上并没有区别。类似地，如果创建一个只包含一项的新列表，实际上就是在创建一个列表元素，而这个元素指向`Nil`作为其表尾。

下面通过一些例子来加以说明：

```
scala> val l: List[Int] = List()
l: List[Int] = List()

scala> l == Nil
res0: Boolean = true

scala> val m: List[String] = List("a")
m: List[String] = List(a)

scala> m.head
res1: String = a

scala> m.tail == Nil
res2: Boolean = true
```

这里使用了两个显式类型的列表，分别是Int和String，以此说明不论数据的类型是什么，List总是以Nil结尾。

Cons操作符

利用与Nil的这种关系，可以采用另外一种方法构造列表。同样类似于Lisp，Scala支持使用cons（construct的简写）操作符来构建列表。使用Nil作为基础，并使用右结合的cons操作符::绑定元素，就可以构建一个列表，而不必使用传统的List(...)格式。

注意：右结合记法

目前为止，我们在空格分隔操作符记法中使用的所有操作符都是左结合的（left-associative），也就是说，要在左边的实体上调用（例如，`10 / 2`）。采用右结合记法时，操作符以一个冒号（`:`）结尾时会触发操作，要在右边的实体上调用操作符。

下面的例子展示了如何使用cons操作符构建一个列表：

```
scala> val numbers = 1 :: 2 :: 3 :: Nil
numbers: List[Int] = List(1, 2, 3)
```

这看起来可能有些奇怪，不过要记住，`::`只是List的一个方法。它取一个值，这会成为新列表的表头，它的表尾指向调用`::`的列表。可以结合cons操作符使用传统的点记法，不过这看起来有些奇怪。

好了，下面来试试看：

```
scala> val first = Nil.::(1)
first: List[Int] = List(1)

scala> first.tail == Nil
res3: Boolean = true
```

如前所述，这确实看起来有些奇怪，所以还是把cons操作符(::)用作为右结合操作符。这一次，我们要用它在一个现有的列表前面追加一个值（插在表的前面），这就会创建一个全新的列表。实际上我们以前已经这样做过，因为这与将一个值追加到Nil（空列表）前面的操作是一样的：

```
scala> val second = 2 :: first
second: List[Int] = List(2, 1)

scala> second.tail == first
res4: Boolean = true
```

尽管"second"列表包含"first"列表，不过这两个都是合法的列表，可以独立使用。这个例子将一个值增加到另一个值来构建列表，展示了Scala不可变列表的递归性和可重用性，可以把它作为这一节的一个很好的总结。

列表算术运算

现在我们已经探讨了List类型的内部原理，而且编写了自己的迭代器。有了这些理解，下面先不考虑这个方面，来换个角度，研究丰富的List方法。这一节我们主要研究列表的基本算术运算。这里使用的“算术”一词并不严格，我是指这些操作可以增加、删除、分解、合并以及修改列表的组织，而不改变列表元素（即其内容）本身。当然，这里所说的“修改”是指“返回一个新列表，其中包含所请求的修改”，因为List是一个不可变的集合。

表6-1显示了List的部分算术方法。要得到所有方法的完整清单，参见List类型的官方Scaladoc页面 (<http://bit.ly/ls-scalalist>)。

表6-1：列表的算术运算

方法名	示例	描述
::	1 :: 2 :: Nil	为列表追加单个元素。这是一个右结合操作符
:::	List(1, 2) :::: List(2, 3)	在列表前面追加另一个列表。这是一个右结合操作符
++	List(1, 2) ++ Set(3, 4, 3)	为列表追加另一个集合
==	List(1, 2) == List(1, 2)	如果集合类型和内容都相同，返回true
distinct	List(3, 5, 4, 3, 4).distinct	返回不包含重复元素的列表版本
drop	List('a', 'b', 'c', 'd').drop 2	从列表中去除前n个元素
filter	List(23, 8, 14, 21).filter (_ > 18)	从列表返回经过一个true/false函数验证的元素
flatten	List(List(1, 2), List(3, 4)).flatten	将一个列表的列表转换为元素列表

表6-1：列表的算术运算（续）

方法名	示例	描述
partition	List(1, 2, 3, 4, 5) partition (_ < 3)	根据一个true/false函数的结果，将元素分组为由两个列表构成的一个元组
reverse	List(1, 2, 3).reverse	逆置列表
slice	List(2, 3, 5, 7) slice (1, 3)	返回列表的一部分，从第一个索引直到第二个索引（但不包括第二个索引）
sortBy	List("apple", "to") sortBy (_.size)	按给定函数返回的值对列表排序
sorted	List("apple", "to").sorted	按自然值对核心Scala类型的列表排序
splitAt	List(2, 3, 5, 7) splitAt 2	根据位于一个给定索引前面还是后面将元素分组为由两个列表构成的一个元组
take	List(2, 3, 5, 7, 11, 13) take 3	从列表抽取前n个元素
zip	List(1, 2) zip List("a", "b")	将两个列表合并为一个元组列表，每个元组包含两个列表中各个索引的相应元素

注意： 使用操作符和点记法

在表6-1中，有些例子使用了操作符记法（如list drop 2），而另外一些使用了点记法（如list.flatten）。选择哪一种记法取决于个人喜好，但是没有操作参数时除外，这种情况下必须采用点记法（如list.flatten中）。

发现这个表中的高阶函数了吗？下面是3个高阶操作的例子：filter、partition和sortBy（分别在Scala REPL中执行）：

```
scala> val f = List(23, 8, 14, 21) filter (_ > 18)
f: List[Int] = List(23, 21)

scala> val p = List(1, 2, 3, 4, 5) partition (_ < 3)
p: (List[Int], List[Int]) = (List(1, 2),List(3, 4, 5))

scala> val s = List("apple", "to") sortBy (_.size)
s: List[String] = List(to, apple)
```

为sortBy方法指定一个函数时，它会返回一个值，用来对列表中的元素排序，而filter和partition方法分别取一个谓词函数。谓词函数（predicate function）得到一个输入值后会相应地返回true或false。对于partition函数，这个谓词使用了占位符语法（见第5章“占位符语法”一节），当输入值小于3时返回true，否则返回false。

如果集合方法同时也是高阶函数（如filter、map和partition），就非常适合使用占位符语法。作为输入的函数参数将作用于列表中的各个元素。因此，发送到这个方法的匿名函数中的下划线（_）就表示列表中的各项。

例如，传入partition方法的匿名函数_< 3就指示要检查列表中的每一个元素，查看它是否小于3。值1和2确实小于3，因此会划分到一个单独的列表中。

对于这些算术方法，有一点很重要，`::`、`drop`和`take`在列表前面完成操作，因此不存在性能损失。应该记得，`List`是一个链表，所以从前面增加项或删除项不需要遍历整个列表。对于短小的列表，列表遍历是一个很简单的操作，不过，如果要处理包含数千个或甚至数百万项的列表，倘若一个操作需要遍历列表，性能就会很成问题。

也就是说，这些操作的反向操作将作用于列表末尾，因此确实需要一个完整的列表遍历。另外，由于在列表末尾增加项会改变这个列表，这就需要复制整个列表，并返回复制得到的列表。同样地，如果不是处理很庞大的列表，这不会带来严重的内存问题，不过一般情况下还是最好在列表前面操作，而不是在末尾操作。

`::`、`drop`和`take`的反向操作分别是`:+`（一个左结合操作符）、`dropRight`和`takeRight`。这些操作符的参数与其反向操作的参数完全相同。

下面是这些列表追加操作的例子：

```
scala> val appended = List(1, 2, 3, 4) :+ 5
appended: List[Int] = List(1, 2, 3, 4, 5)

scala> val suffix = appended takeRight 3
suffix: List[Int] = List(3, 4, 5)

scala> val middle = suffix dropRight 2
middle: List[Int] = List(3)
```

这些例子中使用的列表很小，因此遍历列表并将其内容复制到一个新数组中时，所需的时间和内存都很小，可以忽略不计。不过，还是应当考虑采用作用于列表开头的操作，而不是作用于列表末尾。

映射列表

映射（Map）方法是指取一个函数，将它应用于列表的每一个成员，再把结果收集到一个新列表。在集合论中（以及更一般的数学领域中），映射（map）就是在一个集合的各个元素与另一个集合中的各个元素之间创建一个关联。在某种意义上，这两个定义描述了List中的`map`方法会做什么：将一个列表中的各项映射到另一个列表，使得另一个列表与第一个列表有相同的大小，但是有不同的数据或元素类型。6-2展示了Scala列表的部分映射方法。

表6-2：列表映射操作

操作名	示例	描述
collect	List(0, 1, 0) collect {case 1 => "ok"}	使用一个偏函数转换各个元素，保留可应用的元素
flatMap	List("milk,tea") flatMap (_.split(','))	使用一个给定函数转换各个元素，将结果列表“扁平化”到这个列表中
map	List("milk","tea") map (_.toUpperCase)	使用给定函数转换各个元素

下面来看REPL中这些列表映射操作符如何工作：

```
scala> List(0, 1, 0) collect {case 1 => "ok"}  
res0: List[String] = List(ok)  
  
scala> List("milk,tea") flatMap (_.split(','))  
res1: List[String] = List(milk, tea)  
  
scala> List("milk","tea") map (_.toUpperCase)  
res2: List[String] = List(MILK, TEA)
```

flatMap示例使用String.split()方法将用竖线分隔的文本转换为一个字符串列表。特别需要指出，这就是java.lang.String.split()方法，它返回一个Java数组，而不是一个列表。幸运的是，Scala可以把Java数组转换为它自己的类型Array，这个类型扩展了Iterable。由于List也是Iterable的子类型，而且flatMap方法在Iterable上定义，所以字符串数组列表可以安全地扁平化为一个字符串列表。

归约列表

我们研究了如何改变列表的大小和结构，还讨论了如何把列表转换为完全不同的值和类型。下面来看如何把列表收缩为单个值，这称为归约（reducing）列表。

列表归约是处理集合的一个常见操作。是不是需要对列表求和？或者需要计算多个测试基准的平均持续时间？如果想要检查一个集合是否包含某个特定的元素，或者想查看一个谓词函数对于列表中的每一个元素是否返回“true”，要怎么做？这些都属于列表归约，因为它们的逻辑都是将一个列表缩减为单个值。

Scala的集合支持数学归约操作（例如，查找一个列表的总和）和逻辑归约操作（例如，确定一个列表是否包含某个给定的元素）。它们还支持通用的高阶操作，称为折叠（folds），可以用来创建任何其他类型的列表归约算法。

首先来看内置的数学归约操作。表6-3列出了一些归约方法。

表6-3：数学归约操作

操作名	示例	描述
max	List(41, 59, 26).max	查找列表中的最大值
min	List(10.9, 32.5, 4.23, 5.67).min	查找列表中的最小值
product	List(5, 6, 7).product	将列表中的数相乘
sum	List(11.3, 23.5, 7.2).sum	对列表中的数求和

下面的一组操作可以将列表归约为一个布尔值。表6-4列出了这些方法。

表6-4：布尔归约操作

操作名	示例	描述
contains	List(34, 29, 18) contains 29	检查列表中是否包含这个元素
endsWith	List(0, 4, 3) endsWith List(4, 3)	检查列表是否以一个给定列表结尾
exists	List(24, 17, 32) exists (_ < 18)	检查一个谓词是否至少对列表中的一个元素返回true
forall	List(24, 17, 32) forall (_ < 18)	检查一个谓词是否对列表中的每一个元素都返回true
startsWith	List(0, 4, 3) startsWith List(0)	测试列表是否以一个给定的列表开头

这些布尔列表归约操作特别适合采用中缀操作符记法，不仅是因为它们有单个参数，还因为它们的名字是动词。“list contains x”（列表包含x）读起来就像是一个操作的描述，尽管这确实是一个合法的静态类型函数调用。

除了可读性非常好，这些操作还很类似。要在其中选择正确的操作来完成一个任务，可能更强调可读性，而不只是看适合性。为了说明它们很类似，来看一个例子，下面使用3个不同的操作在一个验证结果列表中搜索“false”项：

```
scala> val validations = List(true, true, false, true, true, true)
validations: List[Boolean] = List(true, true, false, true, true, true)

scala> val valid1 = !(validations contains false)
valid1: Boolean = false

scala> val valid2 = validations forall (_ == true)
valid2: Boolean = false

scala> val valid3 = validations.exists(_ == false) == false
valid3: Boolean = false
```

逻辑上，检查验证列表是否不包含“false”等同于确保这个列表只包含“true”。

这些操作非常有用，所以Scala集合加入了这些操作，不过它们没有那么复杂，我们也

可以自己来实现。下面来创建我们自己的列表归约操作，来说明这是如何做到的。实际上，实现归约操作只需要迭代处理一个累加器（accumulator）变量，这个变量包含目前为止的当前结果，要基于当前元素更新累加器：

```
scala> def contains(x: Int, l: List[Int]): Boolean = {  
|   var a: Boolean = false  
|   for (i <- l) { if (!a) a = (i == x) }  
|   a  
}  
contains: (x: Int, l: List[Int])Boolean  
  
scala> val included = contains(19, List(46, 19, 92))  
included: Boolean = true
```

这样是可以的，不过可能还可以进一步改善。如果把“contains”逻辑从维护累加器值和迭代处理列表的工作中分离出来怎么样？通过把“contains”逻辑移到一个函数参数中，我们可以创建一个可重用的函数，从而支持其他列表归约操作。

下面与前例中的逻辑相同，不过核心“contains”逻辑移至一个函数参数。我们把这个通用函数命名为boolReduce，来表示这是一个布尔列表归约操作：

```
scala> def boolReduce(l: List[Int], start: Boolean)(f: (Boolean, Int) =>  
|   Boolean): Boolean = {  
|  
|   var a = start  
|   for (i <- l) a = f(a, i)  
|   a  
}  
boolReduce: (l: List[Int], start: Boolean)(f: (Boolean, Int) => Boolean)Boolean  
  
scala> val included = boolReduce(List(46, 19, 92), false) { (a, i) =>  
|   if (a) a else (i == 19)  
| }  
included: Boolean = true
```

我们的通用boolReduce函数不再仅限于确定一个列表是否包含某个元素，现在可以在任何其他布尔列表归约操作中重用。理论上讲，我们完全可以实现exists、forall、startsWith和其他布尔操作。

下面进一步拓宽这个例子的适用面。boolReduce函数很适合对整数列表完成布尔操作，不过还可以进一步推广，让它适用于任何类型的列表和归约操作。如果这个函数的列表元素、累加器值以及结果（需要结果来完成匹配）取类型参数，就可以用它来实现max、sum和其他数学运算。

下面将boolReduce操作重写为reduceOp，之所以重新命名是因为这个操作不再特定于布尔类型，Int和Boolean类型已经分别替换为类型参数A和B。最棒的是，我们的示例调

用不需要做任何改变，与处理boolReduce时完全相同，这要归功于Scala的类型参数推导。为了验证这个新操作不限于整数列表和布尔结果，下面再增加一个sum示例实现：

```
scala> def reduceOp[A,B](l: List[A], start: B)(f: (B, A) => B): B = { ①
|   var a = start
|   for (i <- l) a = f(a, i)
|   a
| }
reduceOp: [A, B](l: List[A], start: B)(f: (B, A) => B)B

scala> val included = reduceOp(List(46, 19, 92), false) { (a, i) => ②
|   if (a) a else (i == 19)
| }
included: Boolean = true

scala> val answer = reduceOp(List(11.3, 23.5, 7.2), 0.0)(_ + _) ③
answer: Double = 42.0
```

- ① 把实际类型替换为类型参数可能会使代码可读性下降。如果不清楚A和B参数表示什么，可以查看boolReduce函数定义，比较这些函数中的参数。
- ② 我选择“a”作为累加器值的名字，“i”作为列表中的当前元素名。通过写函数字面量，这就允许为输入参数定义你自己的名字！
- ③ 在这里，我选择采用占位符语法，因为参数在函数体中只访问一次。

现在的reduceOp方法是一个通用的从左到右（或从前到后）的列表归约操作。它可以用来实现一个数学归约操作，如max，或者也可以用来实现一个布尔归约操作，如contains。实际上，这个方法可以用来创建任何其他列表归约操作，至少是那些支持从左向右（或从第一个元素到最后一个元素）扫描列表的操作。

幸运的是，要想获得reduceOp函数的功能，你并不需要写下或记住这个函数。Scala的集合提供了一些内置的操作，类似于reduceOp，这些操作相当灵活，可以提供从左到右、从右到左以及与顺序无关的不同版本，还提供了不同的方法来处理累加器和累加值。这些根据输入函数归约列表的高阶函数通常称为列表折叠（list-folding）操作，因为归约列表的函数也称为折叠（fold）。

表6-5展示了Scala集合的部分列表归约操作。为了简化函数的比较，每个操作的示例都重用了前例中实现的"sum"函数。

表6-5：通用列表归约操作

操作名	示例	描述
fold	List(4, 5, 6).fold(0)(_ + _)	给定一个起始值和一个归约函数来归约列表
foldLeft	List(4, 5, 6).foldLeft(0)(_ + _)	给定一个起始值和一个归约函数从左到右归约列表
foldRight	List(4, 5, 6).foldRight(0)(_ + _)	给定一个起始值和一个归约函数从右到左归约列表
reduce	List(4, 5, 6).reduce(_ + _)	给定一个归约函数，从列表中第一个元素开始归约列表
reduceLeft	List(4, 5, 6).reduceLeft(_ + _)	给定一个归约函数，从列表中第一个元素开始从左到右归约列表
reduceRight	List(4, 5, 6).reduceRight(_ + _)	给定一个归约函数，从列表中第一个元素开始从右到左归约列表
scan	List(4, 5, 6).scan(0)(_ + _)	取一个起始值和一个归约函数，返回各个累加值的一个列表
scanLeft	List(4, 5, 6).scanLeft(0)(_ + _)	取一个起始值和一个归约函数，从左到右返回各个累加值的一个列表
scanRight	List(4, 5, 6).scanRight(0)(_ + _)	取一个起始值和一个归约函数，从右到左返回各个累加值的一个列表

这3个归约操作`fold`、`reduce`和`scan`实际上并没有太大差别。你能看出如何把`reduce`实现为一个特殊情况的`fold`吗？或者如果给定`scan`函数，如何实现`fold`？

有意思的是，各个操作左/右方向的版本（如`foldLeft`）与无方向的版本（如`fold`）之间的区别可能比这3个归约操作之间的区别更重要。一方面，`fold`、`reduce`和`scan`都限于返回与列表元素类型相同的一个值。因此，可以在一个整数列表上用`foldLeft`实现`forall`布尔操作，但是无法用`fold`做到这一点。

另一个主要区别是顺序。举例来说，尽管`foldLeft`和`foldRight`指定了迭代处理列表时的方向，但无方向的操作没有指定迭代的顺序。这通常会让开发人员很困惑，因为不清楚会采用哪个方向。

例如，如果你的集合不是顺序存储，而是分布在数十个不同的计算机上会怎么样？或者如果它们都在同一个计算机上，但是归约操作开销过大，你希望能够并行运行，又会怎么样？在这些情况下，有必要对顺序地迭代处理列表的归约操作与可能按所需的任何顺序实现归约的操作（可能基于实现它的集合）有所区别，这是有意义的。

除非特别要使用分布式或并行集合，或者你在开发一个将由这种集合重用的库，否则完全可以只选择左/右方向版本的归约操作。还有一个建议，除非需要从右到左迭代，

否则最好选择“左”操作，因为左操作的实现需要更少的列表遍历。

在研究这3个列表折叠操作之前，我们曾经用很复杂的方法实现了contains和sum操作。现在我们使用前面介绍的新的折叠操作重新实现这两个操作：

```
scala> val included = List(46, 19, 92).foldLeft(false) { (a, i) => ①
|   if (a) a else (i == 19)
| }
included: Boolean = true

scala> val answer = List(11.3, 23.5, 7.2).reduceLeft(_ + _) ②
answer: Double = 42.0
```

- ① 除了调用foldLeft（一个列表操作），这里没有太大变化。可以在这里使用reduceLeft吗？
- ② 归功于reduceLeft，现在这个操作更为简短，它使用了列表的第一个元素作为起始值，而没有把它作为一个参数。

在这一节中，我们介绍了列表归约/折叠操作，包括特定操作和通用操作。数值和布尔列表归约操作有广泛的使用，不过如果你还需要其他的操作，现在应该已经知道如何自己来创建。

转换集合

列表无处不在，特别是这一章的例子中大量使用了列表，不过其他集合肯定也很重要，它们分别有各自的用途。我发现需要集合时我会默认地使用列表，不过有时确实需要映射、集或其他类型的集合。幸运的是，这些类型之间很容易相互转换，所以可以先创建一个类型的集合，然后经过转换得到另一个类型的集合。

表6-6包含了这样一些方法。由于List.toList()操作很“傻”（不过这样做是可以的），这些例子展示了从一个类型转换为一个完全不同的类型。

表6-6：转换集合的操作

操作名	示例	描述
mkString	List(24, 99, 104).mkString(", ")	使用给定分隔符将一个集合呈现为Set
toBuffer	List('f', 't').toBuffer	将一个不可变的集合转换为一个可变的集合
toList	Map("a" → 1, "b" → 2).toList	将一个集合转换为一个List
toMap	Set(1 → true, 3 → true).toMap	将一个2元元组（长度为2）的集合转换为一个Map
toSet	List(2, 5, 5, 3, 2).toSet	将一个集合转换为一个Set
toString	List(2, 5, 5, 3, 2).toString	将一个集合呈现为一个String，包括集合的类型

如果你有一个映射，不过只想得到映射键的一个列表，或者给定一个列表，希望用它生成一个查找映射，这种情况下就可以考虑这些操作。作为不可变的集合，List、Map和Set不能由空集合构建，所以更适合从现有的集合映射。利用这些操作，可以将一种类型的数据映射到另一种类型，甚至可以从一个序列转换到一个键-值库（或者反之）。

Java和Scala集合兼容性

对于转换集合，还有一个重要的方面需要说明。由于Scala在JVM上编译和运行，经常需要与JDK以及其他Java库进行交互，这是一个很常见的需求。在这个交互中，一部分工作就是要在Java和Scala集合之间转换，因为默认情况下这两个集合类型是不兼容的。

可以增加以下命令来支持手动完成Java和Scala集合之间的转换。尽管这个命令现在看起来有点奇怪，不过本书后面还会在面向对象Scala的上下文中做进一步研究，有了更多的了解之后，就会发现这个命令是有道理的：

```
scala> import collection.JavaConverters._  
import collection.JavaConverters._
```

这个import命令将JavaConverters及其方法增加到当前命名空间。在REPL中这表示当前会话，而在源文件中当前命名空间表示文件的其余部分或局部作用域（也就是增加了这个import命令的地方）。表6-7展示了导入JavaConverters时为Java和Scala集合增加的操作。

表6-7：Java和Scala集合转换

操作名	示例	描述
asJava	List(12, 29).asJava	将这个Scala集合转换为一个相应的Java集合
asScala	new java.util.ArrayList(5).asScala	将这个Java集合转换为一个相应的Scala集合

通过导入JavaConverters，将有更多的Java库和JVM函数可用，而不必改变使用Scala集合的方式。

使用集合的模式匹配

这一章最后要介绍的操作不是一个命名集合方法，而是对集合使用匹配表达式（参见第3章“匹配表达式”一节）。如果还记得，我们使用过匹配表达式来匹配单个值模式：

```
scala> val statuses = List(500, 404)
statuses: List[Int] = List(500, 404)

scala> val msg = statuses.head match {
|   case x if x < 500 => "okay"
|   case _ => "whoah, an error"
|
| }
msg: String = whoah, an error
```

利用模式哨卫（见第3章“模式哨卫”一节），还可以匹配一个集合中的单个值：

```
scala> val msg = statuses match {
|   case x if x contains(500) => "has error"
|   case _ => "okay"
|
| }
msg: String = has error
```

由于集合支持等号(==)，它们也支持模式匹配，这应该不奇怪。要匹配整个集合，可以使用一个新集合作为模式：

```
scala> val msg = statuses match {
|   case List(404, 500) => "not found & error"
|   case List(500, 404) => "error & not found"
|   case List(200, 200) => "okay"
|   case _ => "not sure what happened"
|
| }
msg: String = error & not found
```

可以使用值绑定（见第3章“值绑定”一节）在模式哨卫中将值绑定到集合中的一些或所有元素：

```
scala> val msg = statuses match {
|   case List(500, x) => s"Error followed by $x"
|   case List(e, x) => s"$e was followed by $x"
|
| }
msg: String = Error followed by 404
```

列表可以分解为表头元素和表尾。采用同样的方法，作为模式，它们可以匹配表头和表尾元素：

```
scala> val head = List('r','g','b') match {
|   case x :: xs => x
|   case Nil => ''
|
| }
head: Char = r
```

元组尽管并不是正式的集合，不过也支持模式匹配和值绑定。由于一个元组可以支持不同类型的值，与集合的模式匹配相比，有时元组的模式匹配功能更有用：

```
scala> val code = ('h', 204, true) match {
```

```
| case (_ , _ , false) => 501
| case ('c' , _ , true) => 302
| case ('h' , x , true) => x
| case (c , x , true) => {
|   println(s"Did not expect code $c")
|   x
| }
|
code: Int = 204
```

模式匹配是Scala语言的一个核心特性，而不只是标准集合库中的一个操作。这个特性可以广泛用于Scala的数据结构，使用这个特性可以缩短和简化逻辑，而在其他语言中需要大量的工作才能实现这些逻辑。

小结

不论是创建、映射、过滤还是完成其他操作，处理集合都是软件开发的一个重要部分。列表、映射和集是实现可扩展数据结构的主要构件，它们已经包含在Java、Ruby、Python、PHP和C++中，作为默认库的一部分。Scala的集合库与其他语言中的集合有一个区别：Scala的集合支持不可变数据结构和高阶操作。

Scala中的核心数据结构List、Map和Set都是不可变的。它们不能调整大小，也不能改变其内容。作为一种比可变集合优先的方式，默认地会自动将不可变集合包collection.immutable导入到Scala命名空间。之所以不可变集合更为优先，就是为了促使开发人员转向不可变的集合（更一般的，转向不可变的数据），这也是函数式编程中的“最佳实践”。不过，并不是说可变集合没有不可变集合强大，或相比之下能力逊色。Scala的可变集合拥有与不可变集合相同的特性，此外还支持一组修改操作。下一章我们将了解可变集合，以及如何将可变集合转换为不可变集合（或者将不可变集合转换为可变集合）。

很多语言都可以取一个集合并用一个匿名函数迭代处理或映射这个集合，包括Ruby和Python。不过，这么做的同时，还能够确保集合以及匿名函数输入和返回类型的类型需求，这一点则相当少见。提供类型安全高阶函数的集合支持声明式编程，能够创建具有表达性的代码，而且几乎没有运行时类型转换错误。这些特性结合起来，使得Scala集合从其他语言和框架中可用的那些集合库中脱颖而出，可以使用户大大提高生产力。另外，Scala集合是一元的（monadic），支持以一种高级、类型安全的方式将操作串链在一起。下一章还会学习一元集合。

练习

还记得我前面给出的建议（见第3章“练习”）吗？我建议把开发环境从REPL转换为一个外部Scala源文件。如果你还没有完成这个转换，你会发现，由于规模和复杂性的原因，在REPL中做这些练习几乎是不可能的。

我还建议使用一个专业的IDE（如IntelliJ IDEA CE或基于Eclipse的Scala IDE）完成这些练习。你会立即得到反馈，知道代码是否可以成功编译，而且可以得到Scala库函数的代码完成提示和文档。另外有一些面向简单编辑环境（如Sublime Text、VIM和Emacs）的插件，它们也支持这个功能，不过如果开始使用Scala，最好使用一个完备成熟的IDE，这可能更容易、更快捷。

这一节中的练习将帮助你进一步熟悉这一章介绍的集合和操作。建议你除了写最基本的解决方案外，再花些时间为各个实现寻找一些替代方案。这会帮助你更加了解类似函数之间的细微差别，如fold和reduce，或head和slice，另外还可以提供一些工具，使你能绕过这些函数，开发你自己的解决方案。

1. 创建一个列表，包含前20个Long奇数。你能用一个for循环、filter操作以及map操作创建这个列表吗？编写这个代码的最高效、表达性最好的方法是什么？
2. 写一个名为"factors"的函数，它取一个数，返回这个数的因数构成的一个列表（除了1和它本身）。例如，factors(15)应当返回List(3, 5)。

然后写一个新函数，将"factors"应用到一个数字列表。试着使用练习1生成的Long数列表。例如，用List(9, 11, 13, 15)执行这个函数会返回(3, 3, 5)，因为9的因数是3，而15的因数是3（又是3）和5。这里用map和flatten合适吗？或者for循环是不是更合适？

3. 写一个函数first[A](items: List[A], count: Int): List[A]，这会返回一个给定列表中的前x个项。例如，first(List('a', 't', 'o'), 2) 应当返回List('a', 't')。可以把它写为一个单行函数，直接调用一个完成这个任务的内置列表操作，或者更好的做法是实现你自己的解决方案。你能用一个for循环来实现吗？foldLeft呢？能不能使用一个只访问head和tail的递归函数？
4. 写一个函数，取一个字符串列表，并返回这个列表中最长的字符串。可以不使用可变的变量吗？这里非常适合使用我们介绍过的列表折叠操作（见表6-5）。你能用fold和reduce实现这个函数吗？如果取一个函数参数（它将比较两个字符串，并返回优先的那个字符串），你的函数是不是更有用？如果这个函数可以应用到通用列表，也就是任何类型的列表，该怎么做呢？

- 写一个函数将一个列表逆置。可以把它写为一个递归函数吗？这里很适合使用匹配表达式。
- 写一个函数，它取一个List[String]，并返回一个(List[String], List[String])，这是字符串列表的一个元组。第一个列表应当是原列表中的回文项（前向和后向写法都相同，如"racecar"）。元组中的第二个列表应当是原列表中所有其余的项。可以使用partition很容易地实现这个函数，不过能不能用其他操作来实现？
- 这一章中最后一个练习是一个多部分问题。我们将从强大的OpenWeatherMap API（而且免费）读取和处理天气预报。

要从URL读取内容，我们要使用Scala库操作io.Source.fromURL(url: String)，它会返回一个io.Source实例。然后把源文本使用getLines.toList操作归约为单个行的集合。下面是一个例子，这里使用io.Source从一个URL读取内容，将它分解为单行文本，然后将结果返回为一个字符串列表：

```
scala> val l: List[String] = io.Source.fromURL(url).getLines.toList
```

我们将从以下URL获取天气预报，它采用XML格式：

```
scala> val url =
  "http://api.openweathermap.org/data/2.5/forecast?mode=xml&lat=55&lon=0"
```

将这个URL读入一个字符串列表。完成后，打印第一行，验证确实得到了一个XML文件。结果应当与下面类似：

```
scala> println( l(0) )
<?xml version="1.0" encoding="utf-8"?>
```

如果没有看到XML首部，要确保你的URL是正确的，而且已经连入互联网。

下面来处理这个包含XML文档的List[String]。

- 检查确认有正确的内容，打印这个文件的前10行，这应当只需要一行代码。
- 天气预报中的城市名出现在前10行。从相应的行中抽取，并打印其XML元素。然后从XML元素中抽取城市名和国家代码，把它们打印在一起（例如"Paris, FR"）。这里很适合使用正则表达式从XML标记中抽取文本（见第2章“正则表达式”一节）。
- 如果你不想使用正则表达式捕获组，可以在字符串上使用replaceAll()操作，删除城市名和国家名两边的文本。
- 这里有多少个预报段？写出最短的表达式来统计段数。
- 每个天气预报段中的"symbol" XML元素包括一个天气预报的描述。与抽取城

市名和国家代码类似，用同样的方式抽取这个元素。试着迭代处理天气预报，打印这个描述。

然后打印接下来12个小时的天气描述（不包括XML元素），创建一个非正式的天气预报。

- f. 下面找出这个预报中使用了哪些描述。打印一个有序列表，其中包括天气预报中的所有这些描述，并去除重复的项。
- g. 这些描述可能很有用。“symbol” XML元素中包含一个属性，其中包含符号编号。创建一个从符号编号到描述的Map。从预报访问符号值，并检查描述是否与XML文档匹配，从而验证这个Map是正确的。
- h. 接下来24小时的最高温度和最低温度是什么？
- i. 这个天气预报中的平均温度是什么？可以使用温度元素中的“value”属性来计算这个值。

完成这些练习后，除了你选择的方法，有没有更简单或者更简短的方法？你喜欢中缀点记法还是中缀操作符记法？使用`for..yield`是不是比`map`和`filter`等高阶操作更容易？

可以在这里调整你的答案，找出你喜欢的编码风格，通常需要结合易写、易读和表述性。

更多集合

在第6章中，我们认识了Iterable根类型和它的3个不可变的子类型：有序集合List和无序集合Set和Map。这些集合都称为常用集合（common），这是因为它们在现代编程语言中无处不在，而不只是因为它们很基础、很普通。在这一章中，我们将介绍不太普遍但也很重要的Scala集合。

首先来看可变集合，一般会认为可变集合很普遍，因为与不可变集合相比，会有更多的语言支持可变集合。然后我们再转而介绍数组、流和其他集合。

可变集合

我们熟悉的不可变集合List、Set和Map在创建之后不能改变（见“不可变”的定义）。不过，确实可以把它们变换到新集合。例如，可以创建一个不可变的映射，然后变换这个映射，删除其中一个映射并增加另一个映射：

```
scala> val m = Map("AAPL" -> 597, "MSFT" -> 40) ①
m: scala.collection.immutable.Map[String,Int] =
Map(AAPL -> 597, MSFT -> 40)

scala> val n = m - "AAPL" + ("GOOG" -> 521) ②
n: scala.collection.immutable.Map[String,Int] =
Map(MSFT -> 40, GOOG -> 521)

scala> println(m) ③
Map(AAPL -> 597, MSFT -> 40)
```

- ① 包含“AAPL”和“MSFT”键的新映射。
- ② 删除“APPL”并增加“GOOG”，这会得到一个不同的集合……

③ ……而“m”中原来的集合仍保持不变。

你得到的是“n”中存储的一个全新的集合。原来的集合（存储在“m”值中）仍保持不变。这正是不可变数据（immutable data）的含义，它表示数据和数据结构不可变或不能改变状态，从而可以提高代码的稳定性，并避免bug。举例来说，与有可能在任何时候改变而且很容易破坏的数据结构相比（例如，读取一个正在改变状态的数据结构），如果数据结构很严格，从不改变状态，在并发代码中使用时会更为安全。

不过，有时确实需要使用可变的数据，另外某些情况下使用可变数据也是完全安全的。例如，创建只在一个函数中使用的可变数据结构，或者在返回之前转换为不可变数据结构，这些都是安全的。你可能希望在一系列“if”条件块中为列表增加元素，或者在迭代处理另一个单独的数据结构时增加列表元素，而不需要存储一系列局部值的每一个变换。

这一节中，我们将研究3种构建可变集合的方法。

创建新的可变集合

要修改集合，最直接的方法是利用一个可变的集合类型。见表7-1，其中列出了对应标准的不可变List、Map和Set类型的可变集合类型。

表7-1：可变的集合类型

不可变类型	可变类型
collection.immutable.List	collection.mutable.Buffer
collection.immutable.Set	collection.mutable.Set
collection.immutable.Map	collection.mutable.Map

尽管collection.immutable package会自动增加到Scala的当前命名空间，但不会增加collection.mutable package。创建可变的集合时，要确保包含类型的完整包名。

collection.mutable.Buffer类型是一个通用的可变序列，支持在开头、中间或末尾增加元素。

下面的例子使用这个类型构建一个整数列表，先从一个元素开始创建：

```
scala> val nums = collection.mutable.Buffer(1)
nums: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1)

scala> for (i <- 2 to 10) nums += i

scala> println(nums)
```

```
Buffer(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

下面的例子使用了同样的缓冲区，不过这里从一个空集合开始。由于没有默认值，我们必须用类型参数（在这里就是Int）指定集合的类型：

```
scala> val nums = collection.mutable.Buffer[Int]()
nums: scala.collection.mutable.Buffer[Int] = ArrayBuffer()

scala> for (i <- 1 to 10) nums += i

scala> println(nums)
Buffer(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

构建映射和集的过程也类似。创建一个空集合时，只需要为新集指定类型参数，或者为一个新映射指定键和值类型参数。

利用toList方法，可以随时把可变的缓冲区转换回不可变的列表：

```
scala> println(nums)
Buffer(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val l = nums.toList
l: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

集和映射也类似，可以使用toSet和toMap方法把这些可变的集合转换回相应的不可变类型。

从不可变集合创建可变集合

除了直接创建可变的集合，还可以从不可变的集合转换为可变的集合。如果开始有一个不可变集合，希望修改这个集合，或者希望写"List()"而不是"collection.mutable.Buffer()", 就可以采用这种方法。

不可变集合List、Map和Set都可以用toBuffer方法转换为可变的collection.mutable.Buffer类型。对于列表，显然这很自然，因为缓冲区和列表类型都是序列。Maps作为Iterable的一个子类型，也可以认为是序列，可以作为键-值元组序列转换为缓冲区。不过，将集转换为缓冲区要困难一些，因为缓冲区不能实现集的唯一性约束。幸运的是，可以在转换Set时删除缓冲区数据中的所有重复项。

下面给出一个例子，这里将一个不可变的映射转换为可变的映射，然后再转换回不可变的映射：

```
scala> val m = Map("AAPL" -> 597, "MSFT" -> 40)
m: scala.collection.immutable.Map[String,Int] =
```

```
Map(AAPL -> 597, MSFT -> 40)

scala> val b = m.toBuffer ①
b: scala.collection.mutable.Buffer[(String, Int)] =
  ArrayBuffer((AAPL,597), (MSFT,40))

scala> b trimStart 1 ②

scala> b += ("GOOG" -> 521) ③
res1: b.type = ArrayBuffer((MSFT,40), (GOOG,521))

scala> val n = b.toMap ④
n: scala.collection.immutable.Map[String,Int] =
  Map(MSFT -> 40, GOOG -> 521)
```

- ① 现在包含键-值对的映射是一个元组序列。
- ② trimStart从缓冲区开头删除一项或多项。
- ③ 删除“AAPL”项后，要增加一个“GOOG”项。
- ④ 现在这个元组缓冲区又变成不可变的映射。

除了`toMap`，还可以使用缓冲区方法`toList`和`toSet`将缓冲区转换为一个不可变的集合。

下面将这个大小为2的元组缓冲区分别转换为一个`List`和一个`Set`。毕竟，既然这个大小为2的元组集合是从映射创建的，没有必要再返回为其原来的映射。

为了验证`Set`施加的唯一性约束，首先向缓冲区中增加一个重复的项。下面来看会有什么结果：

```
scala> b += ("GOOG" -> 521)
res2: b.type = ArrayBuffer((MSFT,40), (GOOG,521), (GOOG,521))

scala> val l = b.toList
l: List[(String, Int)] = List((MSFT,40), (GOOG,521), (GOOG,521))

scala> val s = b.toSet
s: scala.collection.immutable.Set[(String, Int)] = Set((MSFT,40), (GOOG,521))
```

这里成功地创建了列表“l”和集“s”，列表中包含重复的项，而集中只包含唯一的项。

`Buffer`类型是一个很好的通用可变集合，与`List`类似，不过还可以增加、删除和替换内容。由于它支持转换方法，而且相应的不可变类型提供了`toBuffer`方法，使得`Buffer`类型成为处理可变数据的一个很有用的机制。

缓冲区唯一的缺点是过于宽泛。如果你只是需要迭代地建立一个集合（如利用一个循环来创建），那么完全可以使用集合构建器。

使用集合构建器

Builder是Buffer的一个简化形式，仅限于生成指定的集合类型，而且只支持追加操作。

要为一个特定的集合类型创建构建器，可以调用这个类型的newBuilder方法，并指定集合元素的类型。调用构建器的result方法可以把它转换到最终的Set。下面的例子将用构建器创建一个Set：

```
scala> val b = Set.newBuilder[Char]
b: scala.collection.mutable.Builder[Char, scala.collection.immutable.
  Set[Char]] = scala.collection.mutable.SetBuilder@726dcf2c

scala> b += 'h' ❶
res3: b.type = scala.collection.mutable.SetBuilder@d13d812

scala> b ++= List('e', 'l', 'l', 'o') ❷
res4: b.type = scala.collection.mutable.SetBuilder@d13d812

scala> val helloSet = b.result ❸
helloSet: scala.collection.immutable.Set[Char] = Set(h, e, l, o)
```

- ❶ 增加一项，两个追加操作之一。
- ❷ 增加多项，另一个追加操作。
- ❸ 与缓冲区不同，构建器知道相应的不可变类型。

那么，为什么使用Builder而不是Buffer或某个可变的集合类型呢？如果你只是迭代地构建一个可变的集合，要把它转换为不可变的集合，Builder类型就是一个很好的选择。如果在构建可变集合时需要Iterable操作，或者不打算把它转换为不可变的集合，那么更好的选择是使用Buffer或其他可变的集合类型。

在这一节中，我们研究了不可变集合和可变集合之间的转换方法，它们要么不可改变，要么可以完全修改。下一节中，我们将介绍一个打破这些规则的“集合”，这种集合的大小是不可变的，但是内容可变。

数组

Array是一个大小固定的可变索引集合。这不是正式意义上的集合，因为它不在"scala.

collections"包里，而且没有扩展根类型Iterable（尽管它包含所有Iterable操作，如map和filter）。Array类型实际上只是Java数组类型的一个包装器，另外还提供了一个高级特性，称为隐含类（implicit class），使它可以像序列一样使用。Scala提供Array类型来保证与JVM库和Java代码的兼容性，另外用来作为索引集合的后备库，这使得数组很有用。

下面给出一些使用数组的例子，从中可以展示数组的单元可变性，以及对Iterable操作的支持：

```
scala> val colors = Array("red", "green", "blue")
colors: Array[String] = Array(red, green, blue)

scala> colors(0) = "purple" ①

scala> colors ②
res0: Array[String] = Array(purple, green, blue)

scala> println("every purple: " + colors) ③
every purple: [Ljava.lang.String;@70cf32e3

scala> val files = new java.io.File(".").listFiles ④
files: Array[java.io.File] = Array(./.Build.scala, ./Dependencies.scala,
  ./build.properties, ./JUnitXmlSupport.scala, ./Repositories.scala,
  ./plugins.sbt, ./project, ./SBTInitialization.scala, ./target)

scala> val scala = files map (_.getName) filter(_ endsWith "scala")
scala: Array[String] = Array(Build.scala, Dependencies.scala,
  JUnitXmlSupport.scala, Repositories.scala, SBTInitialization.scala)
```

- ① 使用一个基于0的索引来替换Array中的任何项。
- ② Scala REPL知道如何打印Array……
- ③ ……不过println()不知道，它只能调用某个类型的toString()方法。
- ④ JDK类java.io.File中的listFiles方法会返回一个数组，可以很容易地映射和过滤这个数组。

Java数组没有覆盖所有Java和Scala对象都有的toString()方法，因此会使用默认实现，将打印类型参数和引用。所以，如果在一个Array上调用toString()，得到的输出可读性可能不好，如上例所示。幸运的是，对于Scala集合不会看到这个输出，因为Scala集合都覆盖了toString()，打印时可以对内容和结构提供更可读的输出。

了解并理解Array类型非常重要，不过，我不建议经常使用这个类型，除非JVM代码中需要用到。还有很多其他很好的序列可以使用，见下一节的介绍。

Seq和序列

Seq是所有序列的根类型，包括类似List的链表和类似Vector的索引（直接存取）列表。如果Array类型是一个集合，可以认为它是一个索引序列，因为可以直接访问元素而无需遍历。作为一个根类型，Seq本身不能实例化，但是可以调用Seq创建List，这是创建List的一个快捷方式：

```
scala> val inks = Seq('C', 'M', 'Y', 'K')
inks: Seq[Char] = List(C, M, Y, K)
```

序列集合的Seq层次体系如图7-1所示，表7-2提供了这些类型的描述。

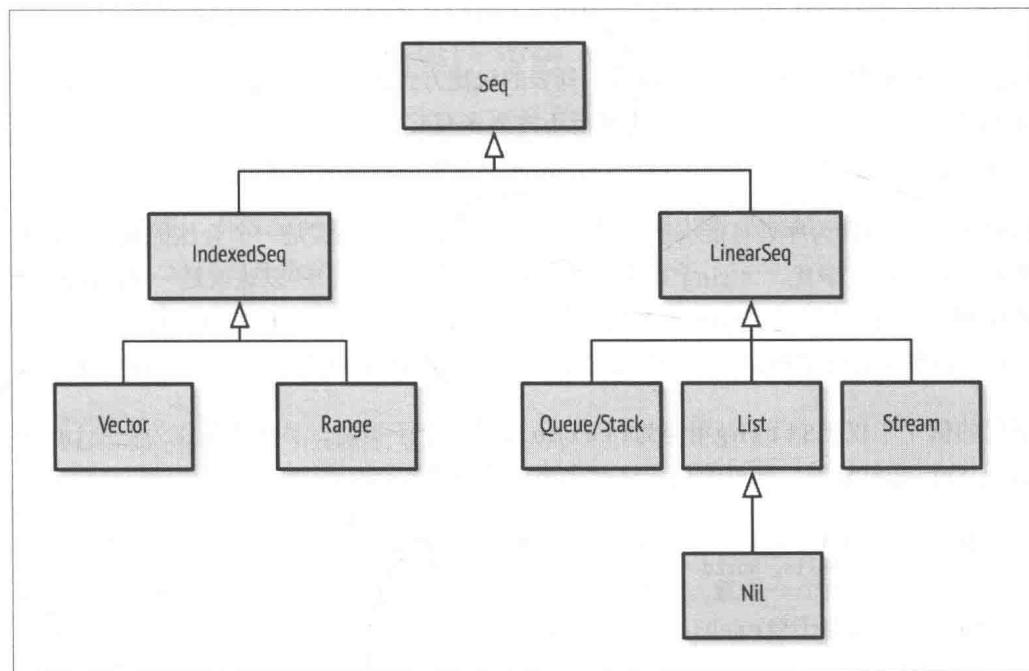


图7-1：序列集合层次体系

表7-2：序列类型

类型名	描述
Seq	所有序列的根类型，List()的快捷方式
IndexedSeq	索引序列的根类型，Vector()的快捷方式
Vector	这一类列表有一个后备Array实例，可以按索引访问
Range	整数范围。动态生成数据
LinearSeq	线性（链表）序列的根类型
List	元素的单链表

表7-2：序列类型（续）

类型名	描述
Queue	先进先出（first-in-first-out, FIFO）列表
Stack	后进先出（last-in-first-out, LIFO）列表
Stream	懒列表。访问元素时才增加相应元素
String	字符集合

`Vector`类型要以一个`Array`提供后备存储。作为一个索引序列（因为数组是有索引的），可以根据索引直接访问`Vector`中的项。与之不同，要访问一个`List`（链表）中的第n个元素，则需要从列表表头开始访问n-1步。Java开发人员会发现`Vector`与Java的“`ArrayList`”有些类似，而C++开发人员很容易发现这就类似“`Vector`”模板。

`List`链表的快捷方式`Seq`和`Vector`索引列表的快捷方式`IndexedSeq`意义不大，因为前一个只是少写了一个字符，而后一个甚至还需要多写4个字符。除非你非常喜欢高级类型（如`Seq`）而不是具体实现（如`List`），通常并不需要使用这些快捷方式。

`String`类型被列为序列可能有些奇怪，不过在Scala中这确实是一个合法的集合，就像其他集合一样。毕竟，“string”的名字源于它是一个字符串，所以这就是一个`Char`元素的序列。`String`类型是一个不可变的集合，它扩展了`Iterable`，支持`Iterable`操作，同时还可以作为Java字符串的一个包装器，支持`split`和`trim`等`java.lang.String`操作。

下面的例子不仅把`String`用作为`Iterable`的一个子类型，另外还用作为[一个`java.lang.String`包装器](#)，这里使用了这两个类型的方法：

```
scala> val hi = "Hello, " ++ "worldly" take 12 replaceAll ("w", 'W')
hi: String = Hello, World
```

`++`和`take`操作来自于`Iterable`，可以处理字符序列，而`replaceAll`是一个`java.lang.String`操作，作为Scala操作符来调用。

这一章讨论的最后一个序列是`Stream`类型，这种集合在访问元素时完成构建。这是函数式编程语言中的一个很常用的集合，不过需要多花些时间来学习，所以我们专门用一节来介绍。花些时间试着完成这些例子，熟悉一下`Stream`，因为这是一个很需要了解而且很有帮助的集合。

Stream

`Stream`类型是一个懒（lazy）集合，由一个或多个起始元素和一个递归函数生成。第一次访问元素时才会把这个元素增加到集合中，这与其他不可变集合正相反，不可变

集合要在实例化时接收全部内容。流生成的元素会缓存，以备以后获取，确保每个元素只生成一次。流可能是无界的，理论上是无限的集合，只是在访问元素时才会生成这个元素。流也可以以Stream.Empty结束，这对应于List.Nil。

与列表类似，流是递归数据结构，包括一个表头（当前元素）和一个表尾（集合的其余部分）。可以利用一个函数以及该函数的递归调用来构建，这个函数返回一个新的流（其中包含表头元素），该函数的递归调用可以构建表尾。可以使用Stream.cons用表头和表尾构建一个新的流。

下面是一个示例函数，它会构建并递归生成一个新的流。通过递增起始的整数值，最后会创建一个连续递增的整数集合：

```
scala> def inc(i: Int): Stream[Int] = Stream.cons(i, inc(i+1))
inc: (i: Int)Stream[Int]

scala> val s = inc(1)
s: Stream[Int] = Stream(1, ?)
```

这里得到了流，不过它只包含起始值(1)，并承诺将来会有其他值(?)。下面通过用take“获取”这些值来建立接下来的4个元素，并获取内容（作为一个列表）：

```
scala> val l = s.take(5).toList
l: List[Int] = List(1, 2, 3, 4, 5)

scala> s
res1: Stream[Int] = Stream(1, 2, 3, 4, 5, ?)
```

我们得到前5个元素，并获取这些元素，作为一个普通列表。通过打印原来的流实例，可以看到它现在包含5个元素，而且准备生成更多元素。可以继续利用take生成20、200或者2000个元素。流包含一个递归函数调用（具体的，就是一个函数值），可以用来无限地生成新元素。

除了Stream.cons操作符，还有一种替代语法，可以使用有些神秘的#::操作符，我们把它称为流的cons操作符。它会完成与Stream.cons相同的工作，只不过采用右结合记法，作为对列表cons操作符::的补充（见第6章“Cons操作符”一节）。

下面再给出“inc”函数，这里使用了cons操作符#::。我还把参数重命名为“head”，从而更好地展示它将作为新Stream实例的表头元素：

```
scala> def inc(head: Int): Stream[Int] = head #:: inc(head+1)
inc: (head: Int)Stream[Int]

scala> inc(10).take(10).toList
res0: List[Int] = List(10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
```

注意：流从哪里构建？

很多开发人员发现流cons操作符语法#::很让人困惑，因为并没有显式地创建底层集合。实际上不需要困惑，因为这里使用了一个称为隐含转换（implicit conversion）的高级特性，可以从递归函数的类型=> Stream[A]生成新的Stream实例。如果你能接受递归函数调用（前例中的inc(i+1)）可以“魔法般地”生成一个Stream作为表尾，就不难理解再在这个魔法表尾前加上表头元素来创建一个新Stream。

下面来创建一个有界的流。我们对递归函数使用两个参数，一个指定新的表头元素，另一个指定要增加的最后一个元素：

```
scala> def to(head: Char, end: Char): Stream[Char] = (head > end) match {  
|   case true => Stream.empty  
|   case false => head #:: to((head+1).toChar, end)  
| }  
to: (head: Char, end: Char)Stream[Char]  
  
scala> val hexChars = to('A', 'F').take(20).toList  
hexChars: List[Char] = List(A, B, C, D, E, F)
```

通过使用新的"to"函数，可以创建一个有界的流，其中只包含写十六进制数时使用的字母。这个流的take操作只返回可用的元素，会在加上Stream.empty终止集合时结束。

用来生成新Stream的递归函数每次会生成新的表头元素（在目前为止的例子中）。在之前的例子中我们往往使用了一个或两个参数函数，不过，也很容易建立一个包含0个或数十个参数的函数。重要的是要为新流定义表头值。

我们介绍的所有集合都能包含0、1个、2个或更多项，只要机器的内存允许或者JVM环境允许，这些集合就可以扩展。接下来，我们将学习一组不能超过一个元素的集合，你可能会惊讶地发现，在你认为绝对不适用集合的一些情况下居然可以使用这种集合。

一元集合

这一章最后讨论的一组集合是一元集合（monadic），它们支持类似Iterable中的变换操作，但是包含的元素不能多于1个。“一元”（monadic）这个词来自于希腊语，表示一个单元，在范畴论中就是一串操作中的一个环节。

首先来讨论Option类型，这是扩展了Iterable的一个一元集合。

Option集合

作为一个大小不会超过1的集合，Option类型表示一个值的存在或不存在。这个值有可能没有（例如，从未初始化，或者不能计算），因此可以包装在一个Option集合中，从而清楚地指示它有可能不存在。

有些开发人员认为Option可以安全地替代null值，告诉用户这个值可能不存在，从而减少触发NullPointerException异常的可能性。另外一些开发人员则把它看作是构建操作链的一种更安全的方法，确保操作链中只包含有效的值。

Option类型本身没有实现，而是依赖两个子类型提供具体实现：Some和None，Some是一个类型参数化的单元素集合，None是一个空集合。None类型没有类型参数，因为它永远不包含任何内容。可以直接使用这些类型，或者调用Option()来检测null值，从而选择适当的子类型。

下面用非null和null值来创建一个Option：

```
scala> var x: String = "Indeed"
x: String = Indeed
```

```
scala> var a = Option(x)
a: Option[String] = Some(Indeed)
```

```
scala> x = null
x: String = null
```

```
scala> var b = Option(x)
b: Option[String] = None
```

可以使用isDefined和isEmpty分别检查一个给定的Option是Some还是None：

```
scala> println(s"a is defined? ${a.isDefined}")
a is defined? true
```

```
scala> println(s"b is not defined? ${b.isEmpty}")
b is not defined? true
```

下面来看一个更实际的例子，我们将定义一个函数返回Option值。在这里，除法操作符 (/) 外加了一个检查，以避免除0操作。对于合法的输入会返回一个Some包装器，对于非法输入则会返回None：

```
scala> def divide(amt: Double, divisor: Double): Option[Double] = {
    |   if (divisor == 0) None
    |   else Option(amt / divisor) ②
    | }
```

```
divide: (amt: Double, divisor: Double)Option[Double]
```

```
scala> val legit = divide(5, 2)
legit: Option[Double] = Some(2.5) ③

scala> val illegit = divide(3, 0)
illegit: Option[Double] = None ④
```

- ① 返回类型是Option并有一个Double类型参数，确保合法的结果会得到正确的类型。
- ② 这会返回一个Some包装器，因为被除数是一个非null值。
- ③ 对于一个合法的除数，被除数会包装在一个Some中。
- ④ 对于不合法的除数，我们会得到None，表示没有值。

如果一个函数会返回包装在Option集合中的值，这说明它可能不能应用于输入数据，因此无法返回一个合法的结果。这会清楚地警告调用者：这个值只是有可能存在，需要仔细地处理其结果。通过这种方式，Option可以提供一种类型安全的方法来处理函数结果，这比Java的标准做法（即返回null值指示缺少数据）要安全得多。

采用这种方式，Scala的集合使用Option类型提供安全的操作来处理空集合的情况。例如，尽管head操作对非空列表很适用，但是对于空列表则会抛出一个错误。更安全的方法是headOption，（可以想见）这个方法返回的表头元素会包装在一个Option中，确保即使在空列表上也能正常工作。

下面的例子调用了headOption来安全地处理空集合：

```
scala> val odds = List(1, 3, 5)
odds: List[Int] = List(1, 3, 5)

scala> val firstOdd = odds.headOption
firstOdd: Option[Int] = Some(1)

scala> val evens = odds filter (_ % 2 == 0)
evens: List[Int] = List()

scala> val firstEven = evens.headOption
firstEven: Option[Int] = None
```

find操作提供了集合选项的另一种用法，这个操作结合了filter和headOption，将返回与一个谓词函数匹配的第一个元素。下面的例子展示了使用find的成功与不成功的集合搜索：

```
scala> val words = List("risible", "scavenger", "gist")
words: List[String] = List(risible, scavenger, gist)

scala> val uppercase = words find (w => w == w.toUpperCase)
uppercase: Option[String] = None
```

```
scala> val lowercase = words find (w => w == w.toLowerCase)
lowercase: Option[String] = Some(risible)
```

实际上，从某种意义上讲，我们已经用过列表归约操作将一个集合归约为单个Option。不过，因为Option本身也是一个集合，所以还可以继续对它进行变换。

例如，可以使用filter和map以某种方式变换“小写”结果，可以保留值，或者采用其他方式变换（可能丢失值）。这些操作都是类型安全的，不会导致null指针异常：

```
scala> val filtered = lowercase filter (_ endsWith "ible") map (_.toUpperCase)
filtered: Option[String] = Some(RISIBLE)

scala> val exactSize = filtered filter (_.size > 15) map (_.size)
exactSize: Option[Int] = None
```

在第二个例子中，filter不能应用于“RISIBLE”值，所以返回None。紧接着的map操作作用于None，没有任何作用，只是再次返回None。

这是Option作为一元集合的一个很好的例子，它可以提供一个单元，在操作链中安全地执行（而且保证类型安全）。操作可以应用到当前值(Some)，但不会应用到没有值(None)，不过结果类型仍与最后的操作类型一致（前例中的Option[Int]）。

我们已经介绍了如何创建和变换Option集合。不过，你可能想知道把Option变换为所需的值、类型或存在性之后可以用来做什么。

从Option抽取值

Option集合提供了一种安全的机制，而且提供了一些操作来存储和变换可能存在也可能不存在的值。另外，毫不奇怪，它还提供了一些安全操作可以抽取可能存在的值。

让人奇怪的是，Option集合还提供了一个不安全的抽取操作：get()方法。如果对一个实际上是Some实例的Option调用这个方法，可以成功地接收到其中包含的值。不过，如果对一个None实例调用get()，就会触发一个“没有这样的元素”错误。

警告：避免Option.get()

Option.get()不安全，应当避免使用，因为它破坏了类型安全操作的初衷，可能导致运行时错误。要尽可能地使用诸如fold或getOrElse等操作，从而可以定义一个安全的默认值。

我们将重点介绍抽取Option值的安全操作。这些操作的核心策略是提供一个框架来管理可能不存在的值，如使用一个替代（即“默认”）值而不是没有值，或者使用一个函数，它可能生成一个替代值或者产生一个错误条件。

表7-3列出了这样一些操作。这个表中的示例也可以用类似Some(10)或None等Option字面量值来写，不过这对于展示如何处理可能存在的数据没有帮助。实际上，下面的例子会调用函数nextOption，每次它会随机地返回一个合法的或者不存在的Option值。在REPL中尝试使用这个函数以及下面的例子，看看Some和None如何改变这些操作的结果：

```
scala> def nextOption = if (util.Random.nextInt > 0) Some(1) else None  
nextOption: Option[Int]  
  
scala> val a = nextOption  
a: Option[Int] = Some(1)  
  
scala> val b = nextOption  
b: Option[Int] = None
```

表7-3：安全的Option抽取操作

操作名	示例	描述
fold	nextOption.fold(-1)(x x)	对于Some，从给定函数返回值（根据嵌入的值），或者返回起始值。也可以用foldLeft、foldRight和reduceXXX方法将一个Option归约为它的内嵌值或者一个计算值
getOrElse	nextOption.getOrElse 5 or nextOption.getOrElse { println("error!"); -1 }	为Some返回值，或者为None返回传名参数的结果（见第5章“传名参数”一节）
orElse	nextOption.orElse nextOption	并不真正抽取值，而是试图为None填入一个值。如果非空，则返回这个Option，否则从给定的传名参数返回一个Option
匹配表达式	nextOption match { case Some(x) => x; case None => -1 }	使用一个匹配表达式处理值（如果存在）。Some(x)表达式将其数据抽取到指定的值“x”，这可能会用作为匹配表达式的返回值，或者在将来的变换中重用

Option类型是一元集合作为单个可串链单元的一个很好的例子。整个Scala集合库都用到了Option，例如，序列的find和headOption操作中就使用了这个类型。也可以在你自己的函数中使用，如果需要表示可能存在的值，就可以用它来表示输入参数和返回值。很多人认为这比使用null（也就是未初始化的值）更安全，因为可以很清楚地知道它有可能没有值，而使用null则无法避免null指针错误。

Option是表示可能存在值的一个通用的一元集合，可以包含其类型参数中指定的任意类型的值。现在我们来看两个有特定用途的一元集合：Try对应成功的值，Future对应最终的值。

Try集合

util.Try集合将错误处理转变为集合管理。它提供了一种机制来捕获给定函数参数中发生的错误，并返回这个错误，或者如果函数成功则返回结果。

Scala允许通过抛出异常（exceptions）来产生错误，异常是可能包含一个消息或其他信息的错误类型。在Scala代码中抛出一个异常会中断程序流，并把控制交回给最近的处理器来处理这个特定的异常。未处理的异常会终止应用，不过大多数Scala应用框架和Web容器会负责避免这种情况发生。

异常可能由你自己的代码抛出，也可能由你调用的库方法抛出，或者由Java虚拟机（JVM）抛出。如果调用了None.get或Nil.head（一个空列表的表头），JVM会抛出一个java.util.NoSuchElementException异常，如果访问一个null值的字段或方法，则会抛出java.lang.NullPointerException异常。

可以使用throw关键字并提供一个新的Exception实例来抛出异常。为Exception提供的文本消息是可选的：

```
scala> throw new Exception("No DB connection, exiting...")
java.lang.Exception: No DB connection, exiting...
... 32 elided
```

为了具体测试我们的异常，下面创建一个函数，它会根据输入条件抛出一个异常。然后我们用它触发异常来进行测试：

```
scala> def loopAndFail(end: Int, failAt: Int): Int = {
    |   for (i <- 1 to end) {
    |     println(s"$i ")
    |     if (i == failAt) throw new Exception("Too many iterations")
    |   }
    | }
loopAndFail: (end: Int, failAt: Int)Int
```

下面使用这个loopAndFail，不过提供一个比检查更大的迭代数，这样就能得到一个异常。这里展示了异常如何中断for循环和整个函数：

```
scala> loopAndFail(10, 3)
1)
2)
3)
java.lang.Exception: Too many iterations
  at $anonfun$loopAndFail$1.apply$mcVI$sp(<console>:10)
  at $anonfun$loopAndFail$1.apply(<console>:8)
  at $anonfun$loopAndFail$1.apply(<console>:8)
  at scala.collection.immutable.Range.foreach(Range.scala:160)
```

```
at .loopAndFail(<console>:8)
... 32 elided
```

抛出异常之后，接下来要捕获和处理异常。要“捕获”一个异常，需要把可能有问题的代码包围在util.Try monadic集合中。

注意：不使用try/catch块？

Scala确实支持try {} .. catch {}块，catch块包含一系列case语句，尝试与所抛出的错误匹配。我建议完全采用util.Try()，因为它提供了一种更安全、更有表述性，而且纯粹一元的方法来处理错误。

util.Try类型与Option类似，没有具体实现，而是有两个已实现的子类型Success和Failure。Success类型包含相应表达式的返回值（如果没有抛出任何异常），Failure类型包含所抛出的Exception。

下面用util.Try包围一些loopAndFail函数调用，看看会得到什么：

```
scala> val t1 = util.Try( loopAndFail(2, 3) ) ①
1)
2)
t1: scala.util.Try[Int] = Success(2) ②

scala> val t2 = util.Try{ loopAndFail(4, 2) } ③
1)
2)
t2: scala.util.Try[Int] = Failure(
  java.lang.Exception: Too many iterations) ④
```

- ① util.Try()取一个函数参数，所以这个loopAndFail调用会自动转换为一个函数字面量。
- ② 函数字面量（安全的loopAndFail调用）安全地退出，所以这里有一个包含返回值的Success。
- ③ 调用util.Try并提供表达式块（见第4章“表达式块”一节）也是允许的。
- ④ 这个函数字面量中抛出一个异常，所以有一个包含指定异常的Failure。

现在来看如何处理可能的错误。由于util.Try和它的子类型都是一元集合，你会找到很多熟悉的方法来处理这些情况。不过，你可能会发现，要为应用选择适当的错误处理方法（以及究竟是否处理错误），这取决于具体的需求和上下文。从我的经验来看，几乎没有通用的错误处理方法。

表7-4提供了处理错误的部分策略。为了更好地描述成功和失败状态，下面定义一个随机的错误函数，示例中将使用这个函数：

```
scala> def nextError = util.Try{ 1 / util.Random.nextInt(2) }
nextError: scala.util.Try[Int]

scala> val x = nextError
x: scala.util.Try[Int] = Failure(java.lang.ArithmetricException:
/ by zero)

scala> val y = nextError
y: scala.util.Try[Int] = Success(1)
```

尝试下面的例子时，就能用成功和失败来进行测试。

表7-4：使用Try的错误处理方法

方法名	示例	描述
flatMap	nextError flatMap { _ =>nextError }	对于Success，调用一个同样返回util.Try的函数，从而将当前返回值映射到一个新的内嵌返回值（或一个异常）。由于我们的“nextError”演示函数没有输入，所以使用下划线来表示当前Success未用的输入值
foreach	nextError foreach(x => println("success!" + x))	一旦Success，则执行给定的函数，或者对于Failure，则不执行
getOrElse	nextError getOrElse 0	返回Success中的内嵌值，或者对于Failure则返回传名参数的值
orElse	nextError orElse nextError	与flatMap相反。对于Failure，调用一个同样返回util.Try的函数。利用orElse，可以把一个Failure变成Success
toOption	nextError.toOption	将util.Try转换为Option，这里Success会变成Some，Failure会变成None。如果你更习惯使用Option，这会很有用，但是缺点是可能会丢失内嵌的Exception
map	nextError map (_ * 2)	对于Success，调用一个函数，将内嵌值映射到一个新值
匹配表达式	nextError match { case util.Success(x) =>x; case util.Failure(error) => -1 }	使用一个匹配表达式来处理包含返回值的Success（存储在“x”中），或者包含异常的Failure（存储在“error”中）。不提供显示：可以用一个好的日志框架记录错误，确保关注并跟踪错误

表7-4：使用Try的错误处理方法（续）

方法名	示例	描述
什么都不做	nextError	这是所有方法中最简单的错误处理方法，也是我个人最喜欢的方法。要使用这种方法，只需要允许异常在调用栈中向上传播，直到被捕获或者导致当前应用退出。这个方法可能对某些敏感情况破坏性过大，但是可以确保所抛出的异常不会被忽略

很多开发人员都需要处理一个常见的异常：验证字符串中存储的数字。下面给出一个例子，这里使用orElse操作来尝试从一个字符串解析一个数，如果成功则使用foreach操作打印结果：

```
scala> val input = " 123 "
input: String = " 123 "

scala> val result = util.Try(input.toInt) orElse util.Try(input.trim.toInt)
result: scala.util.Try[Int] = Success(123)

scala> result foreach { r => println(s"Parsed '$input' to $r!") }
Parsed ' 123 ' to 123!

scala> val x = result match {
|   case util.Success(x) => Some(x)
|   case util.Failure(ex) => {
|     println(s"Couldn't parse input '$input'")
|     None
|   }
| }
x: Option[Int] = Some(123)
```

重申一次：最好的错误处理策略取决于你的当前需求和上下文。不要遇到一个异常后将它忽略，也就是把它替换为一个默认值，一定要避免这种错误处理方法。如果抛出了一个异常，至少应该报告并处理。

Future集合

我们要讨论的最后一个一元集合是concurrent.Future，这会发起一个后台任务。类似Option和Try，future表示一个可能的值，并提供了安全操作来串链其他操作或者抽取值。与Option和Try不同，future的值并不是立即可用，因为创建future时后台任务可能仍在工作。

你已经知道Scala代码在Java虚拟机上执行，也就是JVM，但你可能不知道，它还在Java的“线程”中操作，这是JVM中的轻量级并发进程。默认地，Scala代码在JVM的

“主”线程中运行，不过可以支持在并发线程中运行后台任务。调用future并提供一个函数会在一个单独的线程中执行该函数，而当前线程仍继续操作。因此，future除了作为线程最终返回值的一个一元容器，还是后台Java线程的一个监视器。

幸运的是，创建future相当容易，只需要调用future，并指定希望在后台运行的函数。

下面来创建一个future，指定一个打印消息的函数。在创建这个futue之前，必须指定当前会话或应用的“上下文”来并发运行函数。我们将使用默认的"global" 上下文，这样就会使用Java的线程库：

```
scala> import concurrent.ExecutionContext.Implicits.global
import concurrent.ExecutionContext.Implicits.global

scala> val f = concurrent.Future { println("hi") }
hi
f: scala.concurrent.Future[Unit] =
scala.concurrent.impl.Promise$DefaultPromise@29852487
```

后台任务打印了"hi"之后才会返回future。下面再来看一个例子，它让后台线程睡眠 ("sleeps")，这里使用了Java的Thread.sleep，确保在后台任务运行的同时我们也能得到future！

```
scala> val f = concurrent.Future { Thread.sleep(5000); println("hi") }
f: scala.concurrent.Future[Unit] =
scala.concurrent.impl.Promise$DefaultPromise@4aa3d36

scala> println("waiting")
waiting

scala> hi
```

在睡眠5秒（即5000毫秒）之后，后台任务打印了"hi"消息。与此同时，"main"线程中的代码则有时间在后台任务完成之前打印一个"waiting"消息。

还可以设置回调函数或另外的future，当future的任务完成时执行这个回调或future。作为一个例子，可能有一个API调用需要在后台启动一个持续时间很长的重要操作，它会把控制交给调用者。也可以选择等待，阻塞 "main"线程，直到后台任务完成。可以在future中开始一个异步事件，如网络文件传输，而让"main"线程睡眠，直到这个任务完成或者达到某个“超时”时间。

future可以异步处理（"main"线程继续操作）或者同步处理（"main" 线程等待任务完成）。由于异步操作更高效，使后台线程和当前线程都能继续执行，所以我们先来看异步处理。

异步处理future

除了生成后台任务，future可以看作是一元集合。可以为一个future串链一个函数或另一个future，从而在当前future完成之后执行，并把第一个future的成功结果传递到这个新函数或future。

以这种方式处理的future最后会返回一个util.Try，其中包含函数的返回值或一个异常。如果成功（有返回值），将把这个返回值传入串链的函数或future，并转换为一个future来返回它自己的成功或失败结果。如果是失败（即抛出一个异常），则不会执行另外的函数或future。采用这种方式，作为一元集合，future只是操作序列中的一环，其中包含一个内嵌的值。这与Try类似，如果遇到一个失败，就会中断这个链，另外与Option也类似，它会在不再有值时中断链。

要接收一个future的最终结果，或者一个future链的最终结果，可以指定一个回调函数。这个回调函数接收最终的成功值或异常，而原来创建future的代码可以转去执行其他任务。

表7-5列出了串链future和设置回调函数的一些操作。与前面的操作表类似，我们首先提供一个随机函数，从而提供一个实际的测试用例。这个函数nextFtr会睡眠一段时间，然后返回一个值，或者抛出一个异常。利用它的内部函数"rand"，可以很容易地设置一个睡眠时间（最大5秒即5000毫秒），并确定成功还是失败：

```
scala> import concurrent.ExecutionContext.Implicits.global
import concurrent.ExecutionContext.Implicits.global

scala> import concurrent.Future
import concurrent.Future

scala> def nextFtr(i: Int = 0) = Future {
    |   def rand(x: Int) = util.Random.nextInt(x)
    |
    |   Thread.sleep(rand(5000))
    |   if (rand(3) > 0) (i + 1) else throw new Exception
    |
  }
nextFtr: (i: Int)scala.concurrent.Future[Int]
```

注意：Thread.sleep()使用安全吗？

这一节关于future的例子会使用Java库方法Thread.sleep，来帮助展示运行后台任务的并发性和可能延迟的相关特性。不过，实际上应当避免在你自己的future中使用Thread.sleep，因为这样效率太低。如果确实需要让一个future睡眠，要考虑使用回调函数。

表7-5：异步future操作

操作名	示例	描述
fallbackTo	nextFtr(1) fallbackTo	将第二个future串链到第一个future，并返回一个新的future。如果第一个future不成功，则调用第二个future
	nextFtr(2)	
flatMap	nextFtr(1) flatMap	将第二个future串链到第一个future，并返回一个新的future。如果第一个future成功，则使用它的返回值调用第二个future
	nextFtr()	
map	nextFtr(1) map (_ * 2)	将给定的函数串链到future，返回一个新的future。如果这个future成功，其返回值将用来调用这个函数
onComplete	nextFtr() onComplete {	future的任务完成后，将用一个util.Try调用指定函数，
	_getOrElse 0 }	其中包含一个值（如果成功）或者一个异常（如果失败）
onFailure	nextFtr() onFailure	如果future的任务抛出一个异常，将使用这个异常来调
	{ case _ => "Error!" }	用给定函数
onSuccess	nextFtr() onSuccess	如果future的任务成功完成，将使用返回值来调用给定
	{ case x=>s"Got \$x" }	函数
Future.sequence	concurrent.Future	并发地运行给定序列中的future，返回一个新的future。
	sequence List(如果序列中的所有future都成功，则返回其返回值的一
	nextFtr(1), nextFtr(5))	个列表。否则返回future中出现的第一个异常

关于future，我们给出的代码示例可以很好地展示如何创建和管理future。不过，需要创建、管理及抽取future才算真正有用。下面来看一个更实际的future例子，展示如何从头到尾完整地使用future。

在这个例子中，我们将使用OpenWeatherMap API（还记得第6章“练习”中介绍的这个API吗？），检查两个城市的当前温度（在Kelvin!），并报告哪里更暖和。由于调用远程API可能很耗费时间，所以我们将在并发future中建立API调用，与主线程并发地运行这些任务：

```
scala> import concurrent.Future ①
import concurrent.Future

scala> def cityTemp(name: String): Double = {
|   val url = "http://api.openweathermap.org/data/2.5/weather"
|   val cityUrl = s"$url?q=$name"
|   val json = io.Source.fromURL(cityUrl).mkString.trim ②
|   val pattern = """.*temp":([\d.]+).*""".r ③
|   val pattern(temp) = json ④
|   temp.toDouble
| }
cityTemp: (name: String)Double

scala> val cityTemps = Future sequence Seq( ⑤
```

```

|   Future(cityTemp("Fresno")), Future(cityTemp("Tempe"))
| )
cityTemps: scala.concurrent.Future[Seq[Double]] =
scala.concurrent.impl.Promise$DefaultPromise@51e0301d

scala> cityTemps onSuccess {
|   case Seq(x,y) if x > y => println(s"Fresno is warmer: $x K") ⑥
|   case Seq(x,y) if y > x => println(s"Tempe is warmer: $y K")
| }
Tempe is warmer: 306.1 K

```

- ① 没错，有时输入太多次“concurrent.Future”确实很麻烦。`import`命令可以将一个包的类型导入到当前会话的命名空间。
- ② `io.Source`有很多有用的I/O操作来帮助实现Scala应用。
- ③ 捕获JSON响应中的“temp”字段。
- ④ 使用正则表达式从一个捕获组生成一个值（有关这个主题的内容可以复习第2章“正则表达式”一节）。
- ⑤ 通过调用`Future.sequence`，并发地调用`future`序列，并返回其结果的一个列表。
- ⑥ 这是使用一个模式哨卫在序列上完成的模式匹配（见第6章“使用集合的模式匹配”一节，了解如何对集合使用模式匹配）。

这个例子中，我们可以建立一个远程API的多个并发调用，而不用阻塞主线程，也就是Scala REPL会话。要调用一个远程API并使用正则表达式解析其JSON结果，只需要几行代码就可以实现（这里所说的几行是指只有十行左右），而并发地执行也需要差不多同样多的代码。

现在你应该已经很了解如何创建`future`以及异步地处理`future`。在下一节中，我们将介绍如果必须等待一个`future`完成该怎么做。

同步处理`future`

如果等待一个后台线程完成时要阻塞一个线程，这往往是一个非常耗费资源的操作。对于大流量或高性能应用，要避免这么做，而应当使用回调函数如`onComplete`或`onSuccess`。不过，有些情况下，可能确实需要阻塞当前线程，并等待后台线程完成，而不论后台线程是否成功。

要阻塞当前线程并等待另一个线程完成，需要使用`concurrent.Await.result()`，它取后台线程和一个最大等待时间作为参数。如果`future`在给定时间以内完成，则返回其结果，不过不能按时完成的`future`会导致一个`java.util.concurrent.TimeoutException`

异常。抛出的这个异常可能需要使用util.Try来安全地管理超时条件，所以要确保选择可以接受的时间，尽量减少这种情况发生。

为了展示如何使用concurrent.Await.result，下面使用之前用来测试异步操作所创建的"nextFtr"演示函数（见本章前面“异步处理future”一节）。首先导入"duration"包的内容来访问Duration类型，以便指定时间段以及其单位的类型：

```
scala> import concurrent.duration._ ①
import concurrent.duration._

scala> val maxTime = Duration(10, SECONDS) ②
maxTime: scala.concurrent.duration.FiniteDuration = 10 seconds

scala> val amount = concurrent.Await.result(nextFtr(5), maxTime)
amount: Int = 6 ③

scala> val amount = concurrent.Await.result(nextFtr(5), maxTime)
java.lang.Exception ④
  at $anonfun$nextFtr$1.apply$mcI$sp(<console>:18)
  at $anonfun$nextFtr$1.apply(<console>:15)
  at $anonfun$nextFtr$1.apply(<console>:15)
...

```

- ① 末尾的下划线 (_) 将给定包的所有成员导入到当前命名空间。
- ② SECONDS是concurrent.duration包的一个成员，指示给定时间段（如这里的10）单位为秒。
- ③ "nextFtr()" 返回一个成功值时，concurrent.Await会返回这个值……
- ④ ……但是当"nextFtr()"抛出一个异常时，当前线程会中断。

尽管第一个concurrent.Await.result调用是一个成功的调用，但是第二个调用导致一个异常，并中断了Scala REPL。处理同步操作时，你可能希望增加自己的util.Try包装器，来确保future中抛出的异常不会破坏当前流。并不严格要求这么做，因为完全允许异常传播，这是一个合法的设计选择。

小结

可变集合在大多数编程语言中都很流行也很有用，而在Scala中则兼具两方面的优点。它们可以用作为递增的缓冲区来扩展集合，可以使用缓冲区、构建器或其他方法一次扩展一项，另外还支持不可变集合的大量操作。

集合（特别是Scala定义的集合）不只是应用数据的简单容器。一元集合提供了类型安

全的可串链操作和管理，可以处理一些敏感而复杂的情况，如缺少数据、错误条件和并发处理。

在Scala中，要开发安全而且表述性强的软件，不可变、可变和一元集合是不可缺少的构件和基础。它们在Scala代码中广泛使用，用途很广。

通过学习并熟悉Iterable的核心操作，并结合一元集合的安全操作串链，你可以更好地利用它们作为你的Scala应用的核心基础。

这一章就是关于Scala的第一部分的最后一章。在第二部分中，我们将介绍面向对象Scala，这是这个编程语言的一个核心特性，同时还会继续使用我们目前所学到的知识。

练习

1. 斐波纳契数列以数字"1, 1" 开始，后面的各个元素分别是前面两个元素之和。我们将使用这个数列来熟悉这一章介绍的集合。
 - a. 编写一个函数，返回斐波纳契数列中前x个元素的一个列表。你能用一个Buffer来实现吗？这里适合使用Builder吗？
 - b. 编写一个新的斐波纳契函数，将新的斐波纳契数增加到一个已有的数字列表。它取一个数字列表(List[Int]) 和要增加的新元素个数，并返回一个新的列表(List[Int])。尽管输入列表和返回的列表是不可变的，不过应该可以在函数内部使用一个可变的列表。你能只用不可变列表写这个函数吗？哪一个版本（使用可变集合和不可变集合）更合适，更可读？
 - c. Stream集合是创建一个斐波纳契数列的很好的解决方案。创建一个流，生成一个斐波纳契数列。用它打印这个数列中的前100个元素，生成一个格式化报告，每行10个元素，并用逗号分隔。
 - d. 编写一个函数，取斐波纳契数列中的一个元素，返回这个数列中的下一个元素。例如，fibNext(8)应当返回13。如何处理不合法的输入，如fibNext(9)？要如何向调用者传达缺少一个返回值？
2. 在Array集合的例子中（见本章前面“数组”一节），我们使用了java.io.File(<path>).listFiles操作来返回当前目录中的文件数组。写一个函数对一个目录完成同样的工作，并把各项转换为相应的String表示（使用toString方法）。过滤掉所有点文件（以'.'字符开头的文件），打印其余的文件时以分号分隔

(;)。如果你的计算机上有一个包含大量文件的目录，针对这个目录测试这个函数。

3. 取练习2得到的文件列表，打印一个报告，显示字母表中的各个字母以及以该字母开头的文件数。
4. 写一个函数返回两个数之积，这两个数分别指定为String，而不是一个数值类型。你能同时支持整数和浮点数吗？如何表达其中一个或两个输入可能不合法？你能使用一个匹配表达式处理转换的数吗？能不能使用for循环来做到？
5. 编写一个函数安全地包装JVM库方法System.getProperty(<String>)的调用，避免产生异常或null结果。给定属性名，System.getProperty(<String>)返回一个JVM环境属性值。例如，System.getProperty("java.home")会返回当前运行的Java实例的路径，而System.getProperty("user.timezone") 返回操作系统的时区属性。不过，这个方法使用时可能很危险，因为它可能抛出异常，或者对不合法的输入返回null。在Scala REPL中试着调用System.getProperty("")或System.getProperty("blah")，来看它如何响应。

有经验的Scala开发人员会构建他们自己的函数库，用Scala一元集合包装不安全的代码。你的函数要把输入传入这个方法，确保能安全地处理和过滤异常和null值。对这里使用的示例属性名调用你的函数，包括合法和不合法的输入，来验证它不会产生异常或返回null结果。

6. 写一个函数，报告一个项目的最近GitHub提交(commit)情况。GitHub提供了一个给定用户的最近commit的RSS提要、存储库和分支(branch)，其中包含XML，可以用正则表达式解析这个XML。这个函数应当取用户、存储库和分支，读取并解析RSS提要，然后输出提交信息。这应当包含每个提交的日期、题目和作者。

可以使用以下RSS URL来获取一个给定存储库和分支的当前commit：

```
https://github.com/<user name>/<repo name>/commits/<branch name>.atom
```

可以如下将RSS提要获取为一个字符串：

```
scala> val u = "https://github.com/scala/scala/commits/2.11.x.atom"
u: String = https://github.com/scala/scala/commits/2.11.x.atom

scala> val s = io.Source.fromURL(u)
s: scala.io.BufferedSource = non-empty iterator

scala> val text = s.getLines.map(_.trim).mkString("")
text: String = <?xml version="1.0" encoding="UTF-8"?><feed xmlns=...
```

处理XML可能很点困难。可以使用`text.split(<token>)` 将文本分解为单独的`<entry>`，然后用正则表达式捕获组（见第2章“正则表达式”一节）解析出`<title>`和其他元素。还可以迭代处理XML文件的所有行，发现元素时把它们增加到一个缓冲区，然后把它们转换为一个新列表。

完成这个练习后（这里有很多工作要做），还有几个特性很值得好好研究：

- a. 将`user`、`repo`和`branch`参数移到一个元组参数中。
- b. 在练习a基础上，让函数取一个GitHub项目列表，按指定项目的顺序打印各个项目的提交情况报告。
- c. 在练习b基础上，获取所有项目，使用`future`并发地提交数据，等待结果（不超过5秒），然后按指定项目的顺序打印各个项目的提交情况报告。
- d. 在练习c基础上，把提交情况混杂在一起，按提交日期排序，然后打印报告，增加一个额外的 "repo"列。

实现这些额外的特性可能需要花一些时间，不过这对学习Scala以及提高你的Scala开发能力很有意义。

完成这些特性后，针对以下项目测试你的提交报告：

```
https://github.com/akka/akka/tree/master  
https://github.com/scala/scala/tree/2.11.x  
https://github.com/sbt/sbt/tree/0.13  
https://github.com/scalaz/scalaz/tree/series/7.2.x
```

这些项目都还在进行中或已经完成，所以应该能在你的报告里看到一些很有意思的提交活动数据。有必要好好浏览这些核心开源Scala项目的存储库，或者至少应该看看它们的文档，来了解这些卓越的项目。

7. 写一个命令行脚本来调用练习6得到的GitHub提交报告函数，并打印结果。这需要使用一个UNIX shell；如果你使用Windows系统，可能需要一个兼容的UNIX环境，如Cygwin或Virtualbox（运行一个UNIX虚拟机）。另外还需要安装SBT (Simple Build Tool)，这个构建工具支持依赖文件管理和插件，在Scala项目中很常用。可以从<http://www.scala-sbt.org/>下载面向各个环境的SBT，包括一个MSI Windows Installer版本。SBT也可以从流行的包管理器得到。如果你在使用OS X的Homebrew，可以用`brew install sbt`来安装。

注意： SBT难学吗？

可能有一点。在这个练习中，我们只用它作为一个shell脚本启动工具，以便在Scala中编写和执行shell脚本。后面几章中，我们将介绍如何编写SBT构建的脚本来管理你自己的项目。

8. 下面是一个基于SBT的示例Scala脚本，它将命令行参数读取为一个List，并打印一个欢迎辞。用三个星号开始的注释块专门用作为SBT设置的注释。在这个脚本中，这个注释指定我们希望使用Scala语言的2.11.1版本：

```
#!/usr/bin/env sbt -Dsbt.main.class=sbt.ScriptMain

/***
scalaVersion := "2.11.1"
*/

def greet(name: String): String = s"Hello, $name!"

// Entry point for our script
args.toList match {
  case List(name) => {
    val greeting = greet(name)
    println(greeting)
  }
  case _ =>
    println("usage: HelloScript.scala <name>")
}
```

把这个脚本复制到文件*HelloScript.scala*中，将权限改为可执行（UNIX环境下执行`chmod a+x HelloScript.scala`）。然后可以直接运行这个脚本：

```
$ ./HelloScript.scala Jason
[info] Set current project to root-4926629s8acd7bce0b (in
      build file:/Users/jason/.sbt/boot/4926629s8acd7bce0b/)
Hello, Jason!
```

你的提交报告脚本需要取多个GitHub项目作为参数。为了保证参数简洁，可能希望结合各个项目的输入构成一个要解析的字符串，如*scala/scala/2.11.x*。

输出应当简洁、格式良好，而且可读。使用固定的列宽可能会有帮助，为此可以在字符串内插时使用printf风格的格式化代码（见第2章“字符串内插”一节）。

第二部分

面向对象Scala

类

在本书第一部分，你已经了解了Scala的核心类型以及如何将它们分组为集合。现在用类来建立你自己的类型。

类（Classes）是面向对象语言的核心构件，这是数据结构与函数（“方法”）的组合。用值和变量定义的类可以根据需要实例化多次，每一个实例都可以用自己的输入数据初始化。利用继承（inheritance），类可以扩展其他类，创建一个超类和子类的层次体系。多态（Polymorphism）使得这些子类可以代表其父类，而封装（encapsulation）提供了私密控制，可以管理类的外在表现。如果这些词对你来说都很陌生，建议你首先去了解一般的面向对象编程方法。尽管我们会介绍基于这些概念的Scala面向对象特性，但我们不会花时间详细介绍这些概念本身。理解这些概念可以帮助你充分利用Scala的面向对象特性，并设计表达性好而且可重用的类型。

首先定义最简单的类并完成实例化：

```
scala> class User
defined class User

scala> val u = new User
u: User = User@7a8c8dcf

scala> val isAnyRef = u.asInstanceOf[AnyRef]
isAnyRef: Boolean = true
```

现在有了我们的第一个类。REPL输出时，你会看到类名和一个十六进制串。这是这个实例的JVM内部引用。如果创建一个新实例，应该会看到输出一个不同的值，因为第二个实例应该有一个不同的内存位置，因此会有与第一个实例不同的引用。

类名"User"后面输出的十六进制数看起来可能有点奇怪。这个结果是由JVM的java.lang.Object.toString方法输出的。java.lang.Object类是JVM中所有实例的根，包括Scala，这实际上等价于Scala根类型Any。打印一个实例时，REPL就会调用这个实例的toString方法，这是它从根类型继承的方法。这个User类的具体父类型是AnyRef（见表2-4），这是所有可实例化的类型的根。因此，在User类上调用toString实际上会调用其父类AnyRef，然后再继续调用更上一级父类Any，这与java.lang.Object相同，就是在这里提供了toString方法。

下面重新设计User类，让它更加有用。我们将增加一个值以及处理这个值的一些方法。我们还会覆盖默认的toString方法，提供一个更有信息含量的版本：

```
scala> class User {
|   val name: String = "Yubaba"
|   def greet: String = s"Hello from $name"
|   override def toString = s"User($name)"
| }
defined class User

scala> val u = new User
u: User = User(Yubaba)

scala> println( u.greet )
Hello from Yubaba
```

这里增加了值和方法，还有一个全新的toString方法，它会显示这个实例的内容。

下面让这个类更有用一些，将"name"字段从一个固定值转换为参数化的值。毕竟，不会有人都需要多个有相同名字的User类实例。在Scala中，类参数（如果有）在类名后指定，这就像函数定义中函数参数跟在函数名后面一样：

```
scala> class User(n: String) {
|   val name: String = n
|   def greet: String = s"Hello from $name"
|   override def toString = s"User($name)"
| }
defined class User

scala> val u = new User("Zeniba")
u: User = User(Zeniba)

scala> println(u.greet)
Hello from Zeniba
```

类参数"n"在这里用来初始化"name"值。不过，它不能用在任何一个方法中。类参数可以用来初始化字段（类中的值和变量），或者用于传入函数，但是一旦类已经创建，这些参数就不再可用。

除了使用类参数来完成初始化，我们还可以把某个字段声明为类参数。通过在类参数前增加关键字`val`或`var`，类参数就成为类中的一个字段。下面来试试看，把“name”字段移至类参数：

```
scala> class User(val name: String) {  
|   def greet: String = s"Hello from $name"  
|   override def toString = s"User($name)"  
| }  
defined class User
```

既然有了一个简单而有用的类，下面就来使用这个类。以下是对列表使用这个新类的一个例子：

```
scala> val users = List(new User("Shoto"), new User("Art3mis"),  
|   new User("Aesch"))  
users: List[User] = List(User(Shoto), User(Art3mis), User(Aesch)) ①  
  
scala> val sizes = users map (_.name.size) ②  
sizes: List[Int] = List(8, 7, 5)  
  
scala> val sorted = users sortBy (_.name)  
sorted: List[User] = List(User(Aesch), User(Art3mis), User(Shoto))  
  
scala> val third = users find (_.name contains "3") ③  
third: Option[User] = Some(User(Art3mis))  
  
scala> val greet = third map (_.greet) getOrElse "hi" ④  
greet: String = Hello from Art3mis
```

- ① 注意到了吗？这个新类是`List`的类型参数。我们覆盖的`toString`方法会清楚地显示`List`的内容。
 - ② 还记得操作符记法（见第4章“方法和操作符”一节）和占位符语法（见第5章“占位符语法”一节）如何工作吗？通过二者结合，可以得到这行简短而且有表述性的代码。
 - ③ 在这一行上，采用操作记法，我们使用了一个Scala操作（`find`，如果存在匹配，它会根据谓词函数返回第一个匹配）和一个Java操作（`java.lang.String`中的`contains`）。
 - ④ 你能看出为什么对`Option[String]`结合使用`map`和`getOrElse`的正确结果是`String`？
- 列表和列表操作不是这一章的重点，因为我们已经在第6章和第7章相当全面地介绍过。不过，Scala开发人员开发他们自己的类时，很有可能会在集合中使用自己的类。这个例子展示了Scala集合不仅适用于核心Scala类型，还适用于你自己定义的任何其他类。

最后再给出几个继承和多态的例子。在Scala中，一个类可以使用**extends**关键字扩展最多一个其他类，另外可以用**override**关键字覆盖（即取代）所继承方法的行为。类中的字段和方法可以用**this**关键字访问（如果确实必要），父类中的字段和方法可以用**super**关键字访问。如果一个方法需要访问其父类中的相应方法（它覆盖了该方法），**super**关键字尤其有用。

我会用一个父类"A"和子类"C"以及二者之间的一个类"B"来展示这些内容：

```
scala> class A {  
|   def hi = "Hello from A"  
|   override def toString = getClass.getName  
| }  
defined class A  
  
scala> class B extends A  
defined class B  
  
scala> class C extends B { override def hi = "hi C -> " + super.hi }  
defined class C  
  
scala> val hiA = new A().hi  
hiA: String = Hello from A  
  
scala> val hiB = new B().hi  
hiB: String = Hello from A  
  
scala> val hiC = new C().hi  
hiC: String = hi C -> Hello from A
```

A和B有相同的"hi"方法，因为B 继承了它的父类方法。C定义了它自己的"hi"，覆盖了A中的版本，并调用A中的"hi"方法，把它包含在消息中。

这些"hi"方法的结果与你期望的一样吗？看到一个B实例输出"Hello from A"可能会有些误导，不过这个硬编码的消息（包含A的类名）确实是B通过扩展得到的方法的真实行为。要让"hi"方法提供更准确的信息，应当包含当前类的类名，如我们的"toString"方法，而不是硬编码写入类名"A"。如果这样做，你认为B和C的"hi"方法会输出什么结果？

下面再来看Scala的多态，多态是指类能够采用其他兼容类的形式。这里所说的“兼容”是指一个子类的实例可以用于替代其父类的实例，但是反过来不行。子类扩展其父类，所以支持父类的所有字段和方法，但是反过来并不成立。

我们将使用前面定义的A、B和C类来测试这一点：

```
scala> val a: A = new A  
a: A = A
```

```
scala> val a: A = new B
a: A = B

scala> val b: B = new A
<console>:9: error: type mismatch;
  found   : A
  required: B
      val b: B = new A
          ^
scala> val b: B = new B
b: B = B
```

每次都可以存储一个相同类型的实例作为值，就像将一个子类的实例存入其父类类型的值中。不过，将父类实例存储到其子类类型的值中则不行。Scala编译器会正确地指出A实例与期望的B类型不兼容。对于这种情况，另一个说法是A实例不符合期望的B类型。B类是A的一个扩展，所以A的字段和方法是B的子集，反过来却并不成立。事实上，B并没有增加任何字段和方法，但也无法改变这一点。

下面利用这个知识，创建A、B和C实例的一个列表。应当如何声明这个列表的类型呢？A、B还是C？

为了确保列表可以包含所有这些类的实例，应当把列表定义为List[A]，这与所有这些类都兼容：

```
scala> val misc = List(new C, new A, new B)
misc: List[A] = List(C, A, B)

scala> val messages = misc.map(_.hi).distinct.sorted
messages: List[String] = List(Hello from A, hi C -> Hello from A)
```

哎呀！尽管我打算把列表定义为List[A]，不过忘记增加显式类型了。幸运的是，Scala编译器能推导出这3个实例的公共类型为A（父类），并正确地设置列表的类型参数。想想看，这正是编译器类型推导特性的重要用途：可以查找一个或多个实例的（最特定的）基本公共类型。

以上对Scala的类做了基本的介绍。在这一章的后面，我们会介绍定义类（包含字段和方法）的完整语法、其他类型的类及指定类型参数的有关细节。

定义类

类是对一个类型的定义，其中包含有核心类型和/或其他类的字段（值和变量）。类还

包含方法，也就是处理所包含字段的一些函数，另外还包含嵌套类定义。这一节首先来看一个基本的类定义，接下来会介绍更多参数化的类。

语法：定义一个简单的类

```
class <identifier> [extends <identifier>] [{ fields, methods, and classes }]
```

这一章引言里定义的类A、B和C已经展示了这个类定义（除了嵌套类）。标识符（*identifier*）是类/类型名，后面是要扩展的类（如果有），然后是可选的一组大括号，其中定义这个类的字段和方法。字段（*fields*）是值或变量，方法（*methods*）是作为类的一部分所定义的函数。

类似表达式和函数，类可以相互嵌套。嵌套类除了可以访问自己的字段和方法，还可以访问其父类的字段和方法。实际上，表达式、函数和类都可以相互嵌套，不过如果看到一个“if..else”表达式块定义并使用自己的私有类，这可能有些奇怪。

可以在类的实例上调用类的方法，或者访问它的字段，实例就是一个内存分配，提供了类字段的存储空间。这个动作（即预留空间来分配一个类的内容）称为实例化（*instantiation*）。可以使用new关键字按类名实例化一个类，可以加小括号，也可以没有。

更有用的是类可以有类参数（*class parameters*），这是初始化类所用的字段和方法的输入值，或者甚至可以作为类的字段。类参数是一个用逗号分隔的名字和类型列表，形式与函数（即方法）输入参数相同。

语法：定义有输入参数的类

```
class <identifier> ([val|var] <identifier>: <type>[, ... ])
    [extends <identifier>(<input parameters>)]
    [{ fields and methods }]
```

正是因为类可以有输入参数，所以程序员才有理由创建多个实例，因为每个实例可以有自己的不同内容。下面来创建一个类，值和变量字段都作为参数：

```
scala> class Car(val make: String, var reserved: Boolean) {
    |   def reserve(r: Boolean): Unit = { reserved = r }
    |
defined class Car

scala> val t = new Car("Toyota", false)
t: Car = Car@4eb48298

scala> t.reserve(true)
```

```
scala> println(s"My ${t.make} is now reserved? ${t.reserved}")
My Toyota is now reserved? true
```

可以用标准中缀点记法访问一个类的字段和方法，实例与字段或方法之间用一个点号（.）分隔。调用一个实例的单参数方法时，也可以使用中缀操作符记法。

与函数类似，可以用命名参数调用类参数（见第4章“用命名参数调用函数”一节），按位置调用的参数（从第一个开始）必须出现在第一个命名参数前面，不过后面的命名参数可以按任意的顺序使用。

举例来说，下面创建"Car"的一个新实例，这里会按与位置相反的顺序使用命名参数：

```
scala> val t2 = new Car(reserved = false, make = "Tesla")
t2: Car = Car@2ff4f00f

scala> println(t2.make)
Tesla
```

如果一个类扩展了其他需要参数的类，则要确保类定义中包含这些参数。`extends`关键字后的类应当根据需要有自己的一组输入参数。

在这个例子中，我们定义了Car的一个新的子类，名为Lotus，它在定义中指定了其父类的输入参数。为便于参考，这里再次给出Car类的定义：

```
scala> class Car(val make: String, var reserved: Boolean) {
    |   def reserve(r: Boolean): Unit = { reserved = r }
    |
defined class Car

scala> class Lotus(val color: String, reserved: Boolean) extends
    Car("Lotus", reserved)
defined class Lotus

scala> val l = new Lotus("Silver", false)
l: Lotus = Lotus@52c46334

scala> println(s"Requested a ${l.color} ${l.make}")
Requested a Silver Lotus
```

这个新子类Lotus有自己的新字段color，另外还有一个非字段输入参数来初始化其父类Car。

除了输入参数，类参数还从函数借用了另一个特性，可以为参数定义默认值（见第4章“有默认值的参数”一节）。这样调用者实例化类时就无需指定任何类参数。

语法：定义有输入参数和默认值的类

```
class <identifier> ([val|var] <identifier>: <type> = <expression>[, ... ])
  [extends <identifier>(<input parameters>)]
  [{ fields and methods }]
```

下面重新定义Car类，为"reserved"字段使用一个默认值，这样一来，实例化类时可以只指定"make"字段。我们将增加第3个字段以及一个默认值，所以实际上可以同时混合使用默认和必要参数：

```
scala> class Car(val make: String, var reserved: Boolean = true,
|           val year: Int = 2015) {
|   override def toString = s"$year $make, reserved = $reserved"
| }
defined class Car

scala> val a = new Car("Acura") ①
a: Car = 2015 Acura, reserved = true

scala> val l = new Car("Lexus", year = 2010) ②
l: Car = 2010 Lexus, reserved = true

scala> val p = new Car(reserved = false, make = "Porsche") ③
p: Car = 2015 Porsche, reserved = false
```

- ① 只需要第1个参数，可以按位置调用。
- ② 在这里，指定了第1个和第3个参数。因为第3个参数的顺序不对，所以必须按名来指定这个参数。
- ③ 这一次所有参数都不是按位置指定，而且忽略最后一个参数。

不过，对于类定义来说，从函数借用的特性不只是命名参数和默认值。类型参数（见第4章“类型参数”一节）在类定义中也可用，也就是函数中输入或返回类型的非数据指示符。

再来想想看，你已经使用过有类型参数的类，所以这并不奇怪。我们见过最常用的是List[A]，它使用一个类型参数来确定其元素及操作的类型。例如，List[String]可能包含String实例，并支持处理和返回一个String的操作。

下面继续修改类定义语法（它还会不断扩展），在类中包含一个或多个类型参数。

语法：定义一个有类型参数的类

```
class <identifier> [type-parameters]
  ([val|var] <identifier>: <type> = <expression>[, ... ])
  [extends <identifier>[type-parameters](<input parameters>)]
```

```
[{ fields and methods }]
```

集合就是可以使用类型参数的类，你已经见过这样的例子。整数列表List[Int](1, 2, 3)就有Int类型参数。

下面来创建我们自己的集合，并使用一个类型参数来确保类型安全性。新集合将扩展Traversable[A]，这是Iterable的父类（见第6章）。

当然，由于只有一个元素，这里并没有太多遍历工作要做。不过，通过扩展这个基本集合，可以得到我们已经习惯使用的各个有用的集合操作：

```
scala> class Singular[A](element: A) extends Traversable[A] { ❶
|   def foreach[B](f: A => B) = f(element) ❷
| }
defined class Singular

scala> val p = new Singular("Planes")
p: Singular[String] = (Planes) ❸

scala> p foreach println ❹
Planes

scala> val name: String = p.head ❺
name: String = Planes
```

- ❶ 这是一个很好的例子，可以在类定义中将类型参数传入父类。
- ❷ 通过定义一个foreach()操作，Traversable可以确保这个类是一个真正的集合，可以使用这个类来支持几乎所有集合操作。
- ❸ 这里对类型参数化的类完成验证，REPL输出了类名以及用来实例化这个类的参数化类型名（String）。
- ❹ 这里使用了我们定义的foreach方法，化简为最基本的调用。
- ❺ 再次使用Foreach，这一次是间接调用，访问Traversable.head时，它就会调用foreach。通过扩展Traversable，我们可以访问head以及大量其他标准的集合操作。

目前我们已经介绍了命名类、继承、实例化、输入参数和类型参数。不管你是否相信，Scala中还有很多其他方法可以定制类定义，很有必要有所了解。例如，控制封装级别（即私密性）和建立抽象层次，或者以某种方式定义方法，从而无需方法名也可以访问这些方法！可以继续了解面向对象Scala的一些有趣的特性。

更多类类型

除了我们目前介绍过的基本类定义，Scala还提供了很多其他类定义。这一节中，我们将查看定义和创建类的其他方法。

抽象类

抽象（abstract class）是将由其他类扩展的一个类，而自己不能实例化。抽象类由 abstract关键字指定，定义类时这个关键字要放在class关键字前面。

可以用抽象类定义其子类必须有的核心字段和方法，但不提供具体实现。基于多态，如果一个值的类型为抽象类，它可以具体指向某个非抽象子类的实例，调用方法时实际上最后会在子类上调用。

由于抽象类声明（declaring）但不定义字段和方法，因此包含未实现的字段和方法。已声明字段或方法包含名和参数，但是字段没有起始值，而方法没有具体实现。如果一个类扩展了一个已声明字段和方法的抽象类，但这个类本身未标志为抽象类，它就必须提供这些字段和方法的实现。抽象类也可以包含已实现的字段和方法，子类中不要求提供这些字段和方法的实现。

下面创建我们自己的抽象类，其中包含已声明的字段和方法，然后尝试实现：

```
scala> abstract class Car {
|   val year: Int
|   val automatic: Boolean = true
|   def color: String
| }
defined class Car

scala> new Car()
<console>:9: error: class Car is abstract; cannot be instantiated
          new Car()

scala> class RedMini(val year: Int) extends Car {
|   def color = "Red"
| }
defined class RedMini

scala> val m: Car = new RedMini(2005)
m: Car = RedMini@5f5a33ed
```

如果实例化抽象类Car本身，这个尝试会失败，原因很明显，这个类是抽象的，不可实例化。不过，可以看看Scala编译器给出的一个很有帮助的错误消息。

创建一个扩展Car的子类，不过要增加一个值参数，另外提供color方法的一个具体实现，这就可以解决这个问题。RedMini类是其父抽象类的一个成功的实现，可以提供年份作为参数来完成实例化。

另一方面，如果一款汽车只有一种颜色，这可能不太好。还可以定义一个更好的子类：取颜色作为输入参数。下面来完成这个修改，定义一个新的子类：

```
scala> class Mini(val year: Int, val color: String) extends Car
defined class Mini

scala> val redMini: Car = new Mini(2005, "Red")
redMini: Car = Mini@1f4dd016

scala> println(s"Got a ${redMini.color} Mini")
Got a Red Mini
```

这个新类"Mini"会取color作为一个输入参数。

注意：等一下，用一个值来实现抽象类？

如果在一个实例上调用无小括号的无参数方法，看上去与访问它的一个值是一样的，所以并不奇怪，确实可以使用一个值来实现一个必要的方法。对于调用者来说，语法是一样的，因为无括号的方法不会有副作用（见第4章“用空括号定义函数”一节），它们的行为应当是一样的。

抽象类在面向对象设计中是一个很有用的工具，利用抽象类，可以创建一个有用的基本类型，而将实现委托给子类。

匿名类

在上一节中，我们已经看到一个类定义Mini，它实现了其父类的声明方法。要为一个父类方法提供实现，还有一种不太正式的做法：可以利用匿名类（anonymous class）来实现，这是一个不可重用的、没有名字的类定义。

要定义一个“一次性”的匿名类，可以实例化父类（可能是抽象类），并在类名和参数后面加上大括号，其中包含你的具体实现。这会得到一个扩展了给定父类的实例，而且有一个一次性的实现，可以像传统类定义创建的实例一样使用。

下面利用一个“listener”类来做些尝试，这是一种设计模式，用于发送通知，这在Java应用中使用很广泛：

```
scala> abstract class Listener { def trigger }
defined class Listener
```

```
scala> val myListener = new Listener {
|   def trigger { println(s"Trigger at ${new java.util.Date}") }
| }
myListener: Listener = $anon$1@59831016

scala> myListener.trigger
Trigger at Fri Jan 24 13:08:51 PDT 2014
```

myListener值是一个类实例，但是它的类定义包含在实例化它本身的同一个表达式中。要创建一个新的myListener，必须重新定义这个匿名类。

下面是一个更有展示性的例子，你会发现在这种情况下使用匿名类很有用。我们有一个类Listening，它可以注册一个Listener，以后根据需要触发这个监听器。可以先用一行代码实例化匿名类，然后再用一行代码把它传递到另一个类的注册函数，不过我们没有这么做，而是把这两个步骤合并为一步：定义匿名类作为方法调用的一部分。对于有JavaScript经验的人来说，这看上去很熟悉，特别是如果你已经用过jQuery风格的事件处理器：

```
scala> abstract class Listener { def trigger }
defined class Listener

scala> class Listening {
|   var listener: Listener = null
|   def register(l: Listener) { listener = l }
|   def sendNotification() { listener.trigger }
| }
defined class Listening

scala> val notification = new Listening()
notification: Listening = Listening@66596c4c

scala> notification.register(new Listener {
|   def trigger { println(s"Trigger at ${new java.util.Date}") }
| })
```

```
scala> notification.sendNotification
Trigger at Fri Jan 24 13:15:32 PDT 2014
```

利用匿名类，类定义不必是稳定的或可重用的。如果一个子类只需要使用一次，就可以采用匿名类语法，这有助于简化你的代码基。

更多字段和方法类型

我们已经介绍了其他类类型，另外还有一些可以使用的替代字段和方法。下面来看类中的另外一些可用字段（值和变量）和方法。

重载方法

重载（overloaded）方法是为调用者提供更多选择的一种策略。一个类可能有两个或多个名字和返回值相同的方法，但是输入参数的设置不同。通过多个实现重载一个方法名，调用一个特定名字的方法时可以有多种选择。

下面是一个重载方法的例子，这些方法同名，但是有不同的参数。在这个例子中，第二个重载方法调用了第一个重载方法，不过适当地修改了它的输入参数：

```
scala> class Printer(msg: String) {
|   def print(s: String): Unit = println(s"$msg: $s")
|   def print(l: Seq[String]): Unit = print(l.mkString(", "))
| }
defined class Printer

scala> new Printer("Today's Report").print("Foggy" :: "Rainy" :: "Hot" :: Nil)
Today's Report: Foggy, Rainy, Hot
```

如果两个方法同名而且有相同的输入参数，但是有不同的返回值，这是不可能的。这样会导致Scala编译器错误，因为在编译时没有办法有针对性地选择其中某一个方法。

重载可能是一个有用的特性，不过很多Scala开发人员更愿意使用默认值参数而不是重载。可以在方法中为参数提供默认值，而不是提供两个方法（一个方法有参数，另一个方法无参数，无参数的方法会调用另一个有参数的方法并提供默认值），这样可以减少编写不必要的代码。

apply方法

名为"apply"的方法有时是指它要作为一个默认方法或一个注入方法（injector method），可以直接调用而不需要方法名。apply方法实际上是一个快捷方式，可以使用小括号触发功能而不需要方法名。

下面来看一个类，它将数字乘以一个预定义的量：

```
scala> class Multiplier(factor: Int) {
|   def apply(input: Int) = input * factor
| }
defined class Multiplier

scala> val tripleMe = new Multiplier(3)
tripleMe: Multiplier = Multiplier@339cde4b

scala> val tripled = tripleMe.apply(10)
tripled: Int = 30

scala> val tripled2 = tripleMe(10)
```

```
tripled2: Int = 30
```

可以用我们的"tripleMe"实例将一个给定的数乘以3（有或没有"apply"都可以）。你可能还记得这个语法：从一个列表按索引获取元素，这里就用到了List.apply方法：

```
scala> val l = List('a', 'b', 'c')
l: List[Char] = List(a, b, c)
```

```
scala> val character = l(1)
character: Char = b
```

在这里，List.apply(index)方法提供了按索引访问元素的方法，这个操作相当常见，所以很适合作为列表的默认方法。

将一个方法作为默认方法有一个潜在的缺点，这可能会让代码看起来有些奇怪。访问默认方法应当很自然，比如列表的存取方法。类似列表的存取方法，只有在合理的情况下才应当使用apply方法。

懒值

我们已经介绍了一些很有意思的工作，可以在Scala中用方法来完成，不过下面来看可以用字段完成的工作。目前为止，类中使用的字段（值和变量）都是在类第一次实例化时创建的。不过，懒值（Lazy values）则只在第一次实例化这些值时才创建。定义一个值时可以在val关键字前面加上关键字lazy来创建一个懒值。

从某种意义上，懒值是介于常规类值和方法之间的一种机制。初始化一个常规类值所用的表达式只是在实例化时执行一次，构成一个方法的表达式则有所不同，它在每次调用这个方法时都会执行。不过，初始化一个懒值的表达式会在调用这个值时执行，但只是第一次。在这方面，懒值就是一个缓存的函数结果。

这个概念用一个例子来解释可能更清楚。下面显示了常规值和懒值的计算：

```
scala> class RandomPoint {
    |   val x = { println("creating x"); util.Random.nextInt }
    |   lazy val y = { println("now y"); util.Random.nextInt }
    |
}
defined class RandomPoint

scala> val p = new RandomPoint()
creating x
p: RandomPoint = RandomPoint@6c225adb

scala> println(s"Location is ${p.x}, ${p.y}")
now y
Location is 2019268581, -806862774
```

```
scala> println(s"Location is ${p.x}, ${p.y}")
Location is 2019268581, -806862774
```

类RandomPoint用表达式初始化两个字段，它们会在返回随机生成的数之前打印一个消息。"x"字段是一个常规值，它在创建实例"p"时初始化。"y"是一个懒值，在我们第一次访问它时才初始化，不过只是第一次。在第二个输出结果中，两个值都已初始化，而且是稳定的。

如果要确保时间或性能敏感操作在类的生命期中只执行一次，懒值则是一种很好的方法。它们常用于存储基于文件的属性、打开的数据库连接，以及其他只有在确实必要时才初始化的不可变数据等信息。通过在一个lazy val表达式中初始化数据，可以确保只有在类实例的生命期中至少访问一次时才操作。

包装

我们已经介绍了很多方法来定义类、方法和字段。如果创建了你自己的类，一段时间之后你肯定希望适当地加以组织，从而避免命名空间冲突。

包就是Scala（和Java）中的代码组织系统。利用包，可以按目录使用点分隔路径来组织Scala代码。如果在Scala源文件最前面使用package关键字，则声明这个文件中的所有类都将包含在这个包中。

语法：为Scala文件定义包

```
package <identifier>
```

Scala采用Java标准来为包命名，包以组织或企业域的逆串开头，然后加上其他名构成路径。例如，在Netflix开发并提供工具方法的一个Scala类可能包装在"com.netflix.utilities"中。

Scala源文件要存储在与包匹配的目录中。例如，"com.netflix.utilities"包中的"DateUtilities"应当存储在`com/netflix/utilities/DateUtilities.scala`下。Scala编译器会把生成的.class文件（JVM可执行代码的标准二进制格式）存储在与包匹配的目录结构中。

下面来试试看，创建一个源文件并指定包，然后编译这个文件。我们将用scalac命令编译源文件，并生成一个类文件，存储在当前目录中：

```
$ mkdir -p src/com/oreilly
$ cat > src/com/oreilly/Config.scala
```

```
package com.oreilly  
  
class Config(val baseUrl: String = "http://localhost")  
  
$ scalac src/com/oreilly/Config.scala  
  
$ ls com/oreilly/Config.class  
com/oreilly/Config.class
```

*src*目录可以很好地将源代码与当前目录中的其他内容分开，不过编译器并不真正使用这个目录。编译器取源文件的相对路径，完成编译，然后相对于启动编译器所在的目录生成一个类文件。

访问包装类

可以用完全点分隔包路径和类名来访问包装类。在前面的"Config"示例中，名为"Config"的类可以作为"com.oreilly.Config"来访问：

下面来访问*java.util*包中的JDK Date类：

```
scala> val d = new java.util.Date  
d: java.util.Date = Wed Jan 22 16:42:04 PDT 2014
```

要访问其他包中的类，一种更高效的方法是将它们导入到当前命名空间。这样不需要包前缀也可以访问这个类。导入一个类时要使用*import*关键字，后面是完整包和类名。

语法：导入一个包装类

```
import <package>.⟨class⟩
```

下面创建一个新Date，不过首先将这个类导入到命名空间，从而能直接按名引用：

```
scala> import java.util.Date  
import java.util.Date  
  
scala> val d = new Date  
d: java.util.Date = Wed Jan 22 16:49:17 PDT 2014
```

实例化的Date类仍在*java.util*包中，不过现在也是当前命名空间的一部分。

*import*命令是一个语句，因为它不返回任何值。与Java中不同（Java中也有一个类似的*import*关键字），代码中能使用语句的任何位置都可以使用*import*。

下面尝试在可以使用其他语句的位置上放置import。在这个例子里，我在println调用的中间增加了一个import导入java的UUID类：

```
scala> println("Your new UUID is " + {import java.util.UUID; UUID.randomUUID()})
Your new UUID is 47ba6844-3df5-403e-92cc-e429e614c9e5
```

可能并不一定要在函数调用的受限作用域中增加import。不过，对于所导入的类，在使用这个类的代码附近增加import可以帮助你更清楚地了解这个import的目的，而且还可以防止从不同的包导入多个同名的类可能导致的命名冲突。如果在不同的作用域中增加冲突的import，而不是放在文件最前面，就可以正常使用这些类而不会有冲突。

除了导入整个包和类，还可以导入包的一部分，无需引用整个包，而是可以只导入一个类。Scala的import是可累积的，所以如果导入了一个包，对于这个包中的类，可以从这些类的完整路径中删除这个包名。

下面再来看Date类，这里利用它的部分包路径来访问：

注意：现在要重置你的REPL会话和命名空间

如果你在Scala REPL中尝试这些例子（为什么不呢），则需要重置你的会话，验证import确实能正常工作。这会清除之前所有的import，从而可以只测试你定义的新import。为此，将类重新导入到REPL的命名空间之前，要键入:reset来重置会话。

```
scala> import java.util
import java.util

scala> val d = new util.Date
d: java.util.Date = Wed Jan 22 2014 06:18:52 PDT 2014
```

累积import确实能正常工作，现在只使用util包就可以访问java.util包中的类。

Scala还支持用下划线（_）操作符导入一个包的全部内容。导入后，这个包中的每一个类都会增加到命名空间。你可能还记得我们用这种方法导入了多个Future辅助类（见第7章“同步处理Future”一节），而不是一次导入一个类。

下面使用这种全部导入（import-all）特性将所有可变的集合导入到我们当前的命名空间，然后尝试使用这个包中的ArrayBuffer和Queue集合：

```
scala> import collection.mutable._
import collection.mutable._

scala> val b = new ArrayBuffer[String]
b: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer()
```

```
scala> b += "Hello"
res0: b.type = ArrayBuffer(Hello)

scala> val q = new Queue[Int]
q: scala.collection.mutable.Queue[Int] = Queue()

scala> q.enqueue(3, 4, 5)

scala> val pop = q.dequeue
pop: Int = 3

scala> println(q)
Queue(4, 5)
```

现在可以按名访问**ArrayBuffer**和**Queue**集合（充分包装的类），而不用显式地从包中导入类。当然，并不只是可以使用这两个集合。因为我们导入了**collection.mutable**包的全部内容，所以所有可变集合在我们的命名空间中都可用。

关于**ArrayBuffer**，注意到了吗？我们导入了**collection.mutable**包中的所有内容，但是REPL会把它的完整类名打印为**scala.collection.mutable.ArrayBuffer**。Scala在每个Scala类中完成自己的自动导入，导入整个**scala._** 和 **java.lang._**包。这样就可以直接访问**scala**和**java.lang**中的类和包，而无需使用完整路径。因此，Scala的随机工具类位于**scala.util.Random**，不过完全可以作为**util.Random**来访问。类似地，尽管让当前线程睡眠的类正式定义为**java.lang.Thread**，不过完全可以按其类名来访问。

从一个包导入所有类和子包可能有一个缺点。如果所导入的包中有一个类与命名空间中的一个类同名，那么就无法访问命名空间中已有的那个类。举例来说，**collection.mutable**包有一个可变版本的**Map**，与不可变版本的**Map**同名。导入整个**mutable**包后，新创建的所有**Map**都将是可变的。这可能是我们想要的，不过，如果并不是我们的本意，就要检查成批导入的包的内容。

除了导入整个包，另一种做法是使用导入组（import group）。利用这个特性，可以列出一组导入的类名，而不是一个完整的包。

语法：使用导入组

```
import <package>.{<class 1>[, <class 2>...]}
```

利用导入组，可以直接导入**Queue**和**ArrayBuffer**集合而不用导入可变的**Map**：

```
scala> import collection.mutable.{Queue, ArrayBuffer}
import collection.mutable.{Queue, ArrayBuffer}

scala> val q = new Queue[Int]
```

```
q: scala.collection.mutable.Queue[Int] = Queue()
scala> val b = new ArrayBuffer[String]
b: scala.collection.mutable.ArrayBuffer[String] = ArrayBuffer()
scala> val m = Map(1 -> 2)
m: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2)
```

只导入我们想要的可变集合之后，使用Queue和ArrayBuffer可变集合的同时仍能访问Map不可变集合。在这个例子中，导入组只是一个快捷方式，可以让我们少写一行代码，不过如果是同一个包中的多个类，显然可以减少“import”段的大小。

实际上还有一种方法可以同时将可变和不可变Map集合增加到当前命名空间，而不会发生冲突。为此，可以使用一个导入别名（import alias），在局部命名空间中对某个类型重命名。只是对类的局部命名空间引用重命名，而不是类本身，所以对于命名空间之外的类（通常是你编辑的文件）并没有任何改变。

语法：使用导入别名

```
import <package>.{<original name>=><alias>}
```

下面使用导入别名将collection.mutable.Map集合导入到我们的命名空间，但它不会与标准的不可变Map发生冲突：

```
scala> import collection.mutable.{Map=>MutMap}
import collection.mutable.{Map=>MutMap}

scala> val m1 = Map(1 -> 2)
m1: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2)

scala> val m2 = MutMap(2 -> 3)
m2: scala.collection.mutable.Map[Int,Int] = Map(2 -> 3)

scala> m2.remove(2); println(m2)
Map()
```

利用这个别名集合“MutMap”（“mutable”的简写而不是“mutt”!），我们可以按名创建可变以及不可变的映射，而不用指定相应的包。

应当知道如何访问其他包中的类（不论是你自己的包还是库包），这对于Scala开发人员来说是一个必备的技能。用包组织你自己的类更应算是一门艺术，因为没有指南告诉我们什么是最佳的方法。我只能给出一个建议，要从可查找性和抽象性的角度来组织你的代码，这样可以方便开发人员找到你的代码，而且能知道应该查找什么代码。

包装语法

我们已经介绍了为一个或多个类声明包的最常见的形式，也就是在文件最前面加 `package <identifier>` 命令。文件中这个声明后面的所有代码都认为是这个包的成员。

指定包还有一种不太常用的形式：可以使用包装语法（packaging syntax），包作为一个块，用大括号包围它的类。采用这种格式时，只会将包块中的类指定为这个包的成员。这样同一个文件就可以包含不同包的成员，另外还能在类似REPL的非文件环境中清晰地区别不同的包。

语法：包装类

```
package <identifier> { <class definitions> }
```

下面重写前面的"Config"示例（见第8章“包装”一节），这里将使用包装语法。由于我们不依赖文件来界定包的末尾，所以可以在REPL中写整个包：

说明： Scala REPL要求包使用“Raw”粘贴模式

传统来讲，包用于标记文件，因此在REPL的标准编辑模式下是不支持的。解决办法是用`:paste -raw`进入“raw”粘贴模式，然后粘贴一个Scala文件的内容，这样就能完成编译，而且可以在REPL中使用。

```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package com {
  package oreilly {
    class Config(val baseUrl: String = "http://localhost")
  }
}

// Exiting paste mode, now interpreting.

scala> val url = new com.oreilly.Config().baseUrl
url: String = http://localhost
```

现在可以在`com.oreilly.Config`得到这个新类，而且已经包装。

你是不是认为包装语法可以嵌套？既然表达式、函数和类定义都可以嵌套，这应该并不奇怪，实际上包确实可以嵌套。嵌套包的一个好处是，可以从嵌套包名的累加推导出包路径。因此，可以从一个外部包"com"和内部包"oreilly"建立一个多部分包（如"com.oreilly"）。

好了，下面来试试看。我们再次进入"raw"粘贴模式，启用REPL的包支持：

```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package com {
  package oreilly {
    class Config(val baseUrl: String = "http://localhost")
  }
}

// Exiting paste mode, now interpreting.

scala> val url = new com.oreilly.Config().baseUrl
url: String = http://localhost
```

现在已经有两种方法为类定义包：按文件定义和按包装语法定义。尽管前者在Scala开发人员中最为流行（在我看来），不过这两种方法都是可以接受的，编译后会得到完全相同的结果。

私密性控制

包装代码之后，下一步就是使用私密性控制来管理对代码的访问。尽管可以将代码组织到不同的包（或子包），不过有时会发现一个包中的某个功能要被其他包隐藏。例如，底层的持久存储代码可能会被用户界面级代码隐藏，从而要求代码使用一个中间层来完成通信。或者你可能希望对谁能扩展子类加以限制，使得父类可以跟踪具体的实现。

默认地，Scala不会增加私密性控制。你写的任何类都可以实例化，任何代码都可以访问类的字段和方法。如果你有一个包含无状态方法的类，如工具函数，这就很合适。

如果出于某种理由，确实需要增加私密性控制，如只能在类中处理的可变状态，那么可以针对类中的一个字段和方法增加私密性控制。

一种私密性控制是将字段和方法标志为受保护（protected），这就限制为只有同一个类或其子类中的代码才能访问这个字段或方法。除了这个类和子类外，所有其他代码都不允许访问。在val、var或def关键字前面使用protected关键字可以标志这个实体受保护。

下面的例子会保护一个字段，不允许外部类访问。不过，这个字段仍可由子类访问：

```
scala> class User { protected val passwd = util.Random.nextString(10) }
defined class User
```

```
scala> class ValidUser extends User { def isValid = ! passwd.isEmpty }
defined class ValidUser

scala> val isValid = new ValidUser().isValid
isValid: Boolean = true
```

为了验证这个"passwd"字段只能由"User"及其子类访问，试着创建"User" 的一个新实例，直接访问受保护的字段。应该会看到编译器将产生一个错误，警告不能从类（或子类）外部访问"passwd"字段。

需要更严格的保护时，可以标志字段和方法为 `private`，仅限定义这个字段或方法的类可以访问。类之外的所有其他代码甚至子类都不允许访问这个字段或方法。

下面再来看"User"类。如果用明文将密码存储在一个类中，这样所有子类都可以访问这个字段，这意味着任何其他代码都可以派生这个类并访问这个密码。当然，可能确实需要一个子类实例来访问这个字段，但是对于有些应用来说，这就有可能成为一个问题。在我们的新版本中，将修正这个问题，使密码是私有的，使得只有"User"类能够访问这个字段。另外通过增加一个公共设置方法使它可变，并提供一个警告系统，从而可以检查密码改变的有关日志。最后，我们将增加一个验证系统，在外部读写时同样不会暴露我们的秘密密码：

```
scala> class User(private var password: String) {
    |   def update(p: String) {
    |     println("Modifying the password!")
    |     password = p
    |   }
    |   def validate(p: String) = p == password
    | }
defined class User

scala> val u = new User("1234")
u: User = User@94f6fb

scala> val isValid = u.validate("4567")
isValid: Boolean = false

scala> u.update("4567")
Modifying the password!

scala> val isValid = u.validate("4567")
isValid: Boolean = true
```

通过拒绝访问password字段，User类会（稍稍）更为安全和更为灵活。由于后备数据与方法解耦合，所以可以改变存储数据的位置（如一个安全的标识系统），而对于调

用者如何访问类则无需任何改变。封装和私密性控制可以防止可变的状态不会意外改变，而且可以与当前使用解耦合，这些还只是封装和私密性控制的一部分好处。

私密性访问修饰符

`private`和`protected`关键字提供了类层次体系限制，不过有时你可能还希望对类成员提供更细粒度的访问控制。例如，一个持久存储包中的一个类可能只想向同一个包中的其他类提供一些数据库级方法，以减少bug，并确保只有一个访问点。

除了指定`private`或`protected`，还可以通过指定访问修饰符增加这一级控制。访问修饰符指定相应限定只在某个给定点以上有效，如一个包、类或实例，而在这个点以内无效。例如，某个方法可能只对其包以外的调用者是秘密的（也就是它的包“以上”），而在包内部可以自由访问。不仅可以在包内部将一个字段标志为`private`，还可以在同一个类的其他实例中将字段标志为`private`，这样就只能由这个实例内的代码访问这个私有字段。

访问修饰符的另一个好处是支持类的访问控制。如果将一个类标志为对所有代码都是私有的，这没有任何好处（如果是这样，实例化这个类又有什么意义呢），不过可以把一个类标志为对其包之外的所有代码都是私有的，这可能很有用。

指定访问修饰符时，可以写包名或类名，或者在`private`或`protected`关键字后面的括号里使用`this`。如果使用包名或类名，则指定这个关键字仅限在这个包或类以外有效（在内部可以自由访问），不过，如果使用`this`，则只能由同一个实例访问。

下面通过一个例子试试看，这里会分别指定包级和实例级保护。我们使用包装语法来指示类的包，另外使用REPL的“raw”粘贴模式支持包装：

```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package com.oreilly {

    private[oreilly] class Config { ❶
        val url = "http://localhost"
    }

    class Authentication {
        private[this] val password = "jason" // TODO change me ❷
        def validate = password.size > 0
    }

    class Test {
```

```

    println(s"url = ${new Config().url}")
}
}

// Exiting paste mode, now interpreting.

scala> val valid = new com.oreilly.Authentication().validate ❸
valid: Boolean = true

scala> new com.oreilly.Test
url = http://localhost      ❹
reso: com.oreilly.Test = com.oreilly.Test@4c309d4d

scala> new com.oreilly.Config
<console>:8: error: class Config in package oreilly cannot be ❺
accessed in package com.oreilly
      new com.oreilly.Config
                           ^

```

- ❶ 现在仅限在“com.oreilly”包内访问“Config”类。这里只需要包路径的最后一部分。
- ❷ 现在除了这个类的同一个实例中的代码，所有其他人都不能访问这个秘密“password”字段。
- ❸ 下面验证从同一个实例访问这个“password”。
- ❹ “Test”类能成功地实例化“Config”类……
- ❺ ……不过在包之外则做不到。

除了私有成员的严格访问策略和保护成员的继承访问策略外，Scala的访问修饰符可以作为一个很有用的补充。对于包级保护，可以根据另一个类的亲密性覆盖这些策略。另一方面，实例级保护则基于类的具体实例对这些策略增加一个额外的限制。使用访问修饰符可以放松或加强访问限制，这对于提高应用的封装性和安全性可能很有帮助（如果使用正确）。

最终类和密封类

`protected`和`private`访问控制和修饰符可以限制对一个类或其成员的整体或基于位置的访问。不过，它们无法限制创建子类。除非将类标志为在其包之外是私有的，不过这样一来，它既不能派生也无法使用。

Final类成员不能在子类中被覆盖。如果用`final`关键字标志一个值、变量或方法，可以

确保所有子类都将使用这个实现。也可以把整个类标志为final，避免其他类派生这个类。

如果对于你的需要来说最终类过于限定，也可以考虑使用密封（sealed）类。密封类会限制一个类的子类必须位于父类所在的同一个文件中。通过密封类，编写代码时可以对其层次结构做出安全的假设。密封类在类定义和class关键字前有一个sealed关键字作为前缀。

Option就是一个常用的密封类，这是我们在前面讨论过的一个一元集合。Option类既是抽象类，也是一个密封类，实现为只有两个子类Some和None。通过确保不存在其他子类，Option可以显式地在代码中引用这些实现。要实现这个集合的未密封的版本可能比较困难，因为任何人都可以增加额外的子类，可能并不遵循Some和None所假设的行为。

类似Option实现，密封类是实现抽象父类的一种很有用的方法，这个父类“知道”并引用特定的子类。通过对同一个文件之外的子类加以限制，可以对类层次体系做出合理的假设，否则会有严重的后果（也可以视为bug）。

小结

类通常是学习一个编程语言的起点。不过，由于它们建立在值和函数基础上，所以先介绍值和函数更合适。既然你已经对类有了深入的了解，现在可以明确地告诉你：值和函数（即“方法”）必然在类中，而不会存在于类之外。类是Scala应用的核心构件，值和方法则构成了类。

不过，并不是只有类能包含值和方法。下一章中我们将介绍对象，这是Scala世界中的单例，可以单独使用，也可以结合类使用。我们还会了解trait在与类结合之前如何包含它们自己的值和函数。

练习

1. 我们要开发一个游戏网站，需要跟踪类似Xbox Two和Playstation 5之类流行的显示器（我在规划未来）。
 - a. 创建一个显示器类，跟踪品牌、型号、上市日期、WiFi类型、支持的物理媒体格式，以及最大视频分辨率。覆盖默认的toString方法，打印一个大小适当的实例描述（小于120字符）。
 - 上市日期（或出厂日期）应当是java.util.Date的一个实例。

- WiFi类型（b/g, b/g/n等）字段为可选，因为有些显示器没有WiFi。
 - 物理媒体格式应当是一个列表。这里String最适用吗？还是要使用一个匹配常量值的Int？
 - 最大视频分辨率应当采用一种特定的格式，从而可以按最大像素数对显示器排序。
- b. 测试这个新的显示器类，写一个新类，创建这个显示器类的4个实例。所有这些实例都应当有正确的值。
- c. 现在来实现游戏。创建一个游戏类，包含名字、开发商和所支持的一个显示器列表，以及一个"isRequired"方法，如果支持一个给定的显示器则返回true。
- d. 测试这个游戏类，生成一个游戏列表，每个游戏包含一个或多个显示器实例。能把这个列表转换为显示器和所支持游戏列表的一个查找表吗？能不能实现一个函数，打印游戏列表，首先按开发商排序，再按游戏名排序？
2. 创建一个面向对象风格的链表。
- a. 创建一个窗口类，包含它自身的一个实例，以及一个参数化类型的实例。构造函数取一些实例（例如，字符串、int或任何其他参数化类型），实例个数可变，这可以用vararg参数（见第4章“vararg参数”一节）实现。实现一个"foreach"方法，用户可以调用这个方法迭代处理列表，对每个元素调用相应的函数。
- 如何确定列表结束？
 - C风格的列表通常使用一个null值来指示列表末尾。在这里这是最好的方法吗？
 - 这里可以使用apply()方法吗？
- b. 相信你的链表肯定很很好地工作，不过试着用一个有趣的方法重构这个类。把你的容器类变成有两个子类的抽象类：一个表示有合法项的节点，另一个表示没有合法项的节点，指示列表的最后一项。
- 需要第二个子类的多个实例吗？
 - 有没有应当私有的辅助方法？
 - 需要子类实现的抽象方法呢？
 - 如果实现了apply()方法，各个子类应当有自己的实现吗？

- c. 为你的链表增加标准的head、tail、filter、size和map集合方法。你能用懒值实现其中某个方法吗？哪些方法要在父类中实现，哪些则要在子类中实现？
 - d. 使用递归而不是迭代实现head、tail、filter、size和map集合方法。你能确保所有这些方法都使用尾递归（见第4章“递归函数”一节）来避免超大集合的栈溢出错误吗？
3. 再换一个问题，下面创建一个目录清单类。构造字段应当是这个目录的完整路径，另外有一个谓词函数，它取一个String（文件名），如果要包含这个文件则返回true。方法"list"应当列出目录中的文件。
- 要实现这个类，需要创建java.io.File的一个实例，并用它的listFiles(filter:FilenameFilter)列出与给定过滤器匹配的文件。可以找到这个方法以及java.io.FilenameFilter类的Javadocs，不过需要明确如何从Scala调用。另外FilenameFilter参数应当作为一个匿名类传入。
- 这个类的哪些部分可以作为懒值？
 - 是否可以把java.io.FilenameFilter的匿名子类存储为一个懒值？
 - 过滤后的目录清单呢？
4. JVM库包含一个能真正工作的MIDI声音合成器。下面的例子会播放一组音符：

```
scala> val synth = javax.sound.midi.MidiSystem.getSynthesizer
synth: javax.sound.midi.Synthesizer = com.sun.media.sound
    .SoftSynthesizer@283a8ad6

scala> synth.open()

scala> val channel = synth.getChannels.head
channel: javax.sound.midi.MidiChannel = com.sun.media.sound
    .SoftChannelProxy@606d6d2c

scala> channel.noteOn(50, 80); Thread.sleep(250); channel.noteOff(30)

scala> synth.close()
```

为它创建一个更简单的界面，为此要编写一个类播放一系列音符。这个类的构造函数要得到音量（这个例子中设置为80），不过总使用相同的持续时间（这个例子中为250毫秒）。它的"play"方法要取一个音符列表，例如Seq(30, 35, 40, 45, 50, 55, 60, 65, 70)，并在合成器中播放。

- 假设getSynthesizer方法调用开销很大。如果永远不会调用"play"方法，如何避免不必要的调用getSynthesizer方法？
- 确保隐藏调用者不需要知道的字段。

- 可以使用一个Range作为输入吗？如play(30 to 70 by 5)？
- 能支持多个范围吗？例如升调、降调然后再升调？
- 假设只需要一个实例，音量设置为95。你能使用访问控制来确保这个类不会有多个实例吗？

对象、Case类和Trait

上一章我们介绍了类，这是面向对象Scala的一个核心组件。应该记得，类只定义一次，但是可以无限次实例化。这一章中，我们会了解一些新的组件，可以用来补充和丰富类，或者完全取代某些类，这取决于你的面向对象设计偏好。很多开发人员选择后者，会尽可能使用这些新组件取代“常规”的类。因此，强烈建议你花些时间学习这些组件，因为最后可能你也希望用它们取代常规的类，不仅如此，还因为它们会为大多数开发人员提供一些新的特性。

这3个组件（对象、case类和trait）相互之间并没有什么关联，所以为它们写一个共同的引言没有太大意义。因此，这一章我们会分别为各个组件提供相应的引言，首先来看对象。

对象

对象（object）是一个类类型，只能有不超过1个实例，在面向对象设计中称为一个单例（singleton）。对象不是用new关键字创建实例，只需要按名直接访问对象。对象会在首次访问时在当前运行的JVM中自动实例化，这也说明了在第一次访问它之前，并不会实例化。

Java和其他语言可以指定一个类的某些字段和方法为“静态”或“全局”，这表示这些字段和方法不会绑定到某个实例的数据，因此不必实例化类也可以访问。对象提供了类似的功能，不过将它们与可实例化的类解耦合。这种分离可以帮助区分全局和基于实例的字段和方法，并提供一种更安全、更可理解的设计。利用这个模型，可以尽

可能避免意外调用一个类的全局方法或者错误地将可变数据存储在可以全局访问的字段中。

对象和类没有完全解耦合。对象可以扩展另一个类，从而可以在一个全局实例中使用它的字段和方法。不过，反过来不成立，因为对象本身不能扩展。这是有道理的，因为没有理由派生一个对象。如果只能实例化某一个对象或者它的子类，为什么不把想要增加的特性直接增加到这个对象本身呢？

定义对象要使用`object`关键字而不是`class`。对象没有任何参数（它们会自动实例化），不过可以像定义常规类一样为对象定义字段、方法和内部类。

语法：定义对象

```
object <identifier> [extends <identifier>] [{ fields, methods, and classes }]
```

下面设计一个对象，来说明对象如何自动实例化：

```
scala> object Hello { println("in Hello"); def hi = "hi" }
defined object Hello

scala> println(Hello.hi)
in Hello
hi

scala> println(Hello.hi)
hi
```

实例化/初始化这个对象时会调用最上面的`println`，这在第一次访问这个对象时发生。重复调用对象的"hi"方法会重用同一个全局实例，所以不会另外初始化。

标准类方法会读写实例的字段，为数据提供辅助的访问点和业务逻辑。类似地，最适合对象的方法是纯函数和处理外部I/O（输入/输出）的函数。纯函数（Pure functions）会返回完全由其输入计算得到的结果，而没有任何副作用，而且在引用方面是透明的（如果替换为函数的结果，将无法区分）。I/O函数是处理外部数据的函数，如处理文件、数据库和外部服务。这些函数都不适合作为类方法，因为它们与类的字段关系不大。

作为一个例子，下面创建一个对象，将提供纯函数作为工具，这是我最喜欢的一种对象用法：

```
scala> object HtmlUtils {
|   def removeMarkup(input: String) = {
|     input
|       .replaceAll("""</?\\w[^>]*>""", "")
```

```
|       .replaceAll("<.*>","");
|   }
| }
defined object HtmlUtils

scala> val html = "<html><body><h1>Introduction</h1></body></html>"
html: String = <html><body><h1>Introduction</h1></body></html>

scala> val text = HtmlUtils.removeMarkup(html)
text: String = Introduction
```

我们的示例工具方法removeMarkup是一个纯函数，它只根据输入数据返回一个结果。作为对象HtmlUtils的成员，现在可以在任何其他代码中全局访问这个方法，而无需初始化一个类。

注意：引用透明性测试

作为一个引用透明性测试，我们可以把这个函数替换为一个只返回结果“Introduction”的函数，这对我们的系统应该没有任何影响。如果是一个读取某个字段的类方法，或者是写至控制台的对象方法，可能做不到这一点，因为它们会依赖于环境或者要对环境做出修改。关键是要尽可能考虑使用纯函数来减少依赖性问题，使你的代码自给自足。

我们已经介绍了对象作为全局（或者静态）类的基本用法，不过对象还有很多其他用途，而不只是存储函数。可以结合同名的类来使用对象（伴生对象），为它们授予特殊的权限，或者作为命令行应用的入口点。下面几节将介绍对象的更多用法。

apply方法和伴生对象

我们已经介绍了类的apply方法（见第8章“apply方法”一节），利用apply方法可以调用一个实例。这个特性对于对象也适用，从而可以按名调用对象。通过定义一个或多个这种方法，可以按名来调用你的对象，如List(1, 2, 3)。

实际上，Scala就采用了这种方式来实例化列表。List对象有一个apply()方法，它取一些参数，并由这些参数返回一个新集合。创建一元集合（见第7章“一元集合”一节）时也使用了这个特性。Option对象的apply()方法取一个值，并返回Some[A]，如果非null，结果就包含这个值，否则包含None。Future对象也有apply()方法，它取一个函数参数，并在一个后台线程中调用这个函数。这在面向对象编程中称为工厂（factory）模式，这也是对象apply()方法的一种流行用法。

具体来讲，工厂模式是一种很流行的方法，可以从伴生对象生成一个类的新实例。伴生对象（companion object）是与类同名的一个对象，与类在同一个文件中定义。为类

提供一个伴生对象在Scala中是一个常用的模式，另外还可以由此得到一个特性。从访问控制的角度讲，伴生对象和类可以认为是单个单元，所以它们可以相互访问私有和保护字段及方法。

下面我们通过一个例子来尝试使用apply()工厂模式和伴生对象模式。我们将使用REPL的:paste模式，来模拟在同一个文件中定义的类和对象，否则REPL会认为它们在不同的文件中定义：

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

class Multiplier(val x: Int) { def product(y: Int) = x * y }

object Multiplier { def apply(x: Int) = new Multiplier(x) }

// Exiting paste mode, now interpreting.

defined class Multiplier
defined object Multiplier

scala> val tripler = Multiplier(3)
tripler: Multiplier = Multiplier@5af28b27

scala> val result = tripler.product(13)
result: Int = 39
```

示例类Multiplier取一个数，并提供了一个方法product，它将这个数乘以另一个数。同名伴生对象有一个“apply”方法，它与实例的参数完全相同，这样用户就可以清楚地知道它要作为这个类的一个工厂方法。

不过，我们还没有看到伴生对象带来的好处，也就是与伴生类共享的特殊的访问控制。下面在一个新例子中尝试这一点，在这里，类会访问其伴生对象的私有成员：

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

object DBConnection {
    private val db_url = "jdbc://localhost"
    private val db_user = "franken"
    private val db_pass = "berry"

    def apply() = new DBConnection
}

class DBConnection {
    private val props = Map(
        "url" -> DBConnection.db_url,
        "user" -> DBConnection.db_user,
        "pass" -> DBConnection.db_pass
```

```
)  
    println(s"Created new connection for " + props("url"))  
}  
  
// Exiting paste mode, now interpreting.  
  
defined object DBConnection  
defined class DBConnection  
  
scala> val conn = DBConnection()  
Created new connection for jdbc://localhost  
conn: DBConnection = DBConnection@4d27d9d
```

这个新DBConnection对象将数据库连接数据存储在私有常量中，而同名的类可以在创建连接时读取这些数据。这些常量是全局的，因为这些设置在整个应用期间保持不变，可以安全地由系统的任何其他部分读取。

使用REPL的粘贴模式还有一个好处：对象和类会同时编译。除了对私有字段的特殊伴生访问，如果不使用粘贴模式，我们就不能在REPL中输入这些字段，因为类和对象相互引用。如果一个类引用了一个未定义的对象，编译时由于这个对象不存在，这会导致一个编译错误。

在前几章的练习中，你可能已经写过直接由scala命令执行的.scala文件。将类和对象定义在一个.scala文件中是可以的，因为它们都是同名命名空间的一部分。可以增加命令来访问.scala文件中的类和对象，用scala运行文件时会执行这些命令。

这种方法很适合用来完成测试，不过可能无法重用你的代码。scala命令会执行文件的内容，就好像是在REPL中输入的一样，不过最后得不到编译的类。要编写可重用的编译代码，需要用scalac命令编译你的类和对象，然后从你自己的应用执行。在下一节中，我们将介绍如何用Scala编写命令行应用，以便重用类和对象。

使用对象的命令行应用

大多数语言都能创建命令行应用，也就是可以从shell执行的应用。最基本地，它们会读取输入参数（可能从输入流读取），然后写至输出流。更复杂的应用可能会处理持久数据，如文件和数据库，通过网络访问其他计算机，或者启动新的应用。

Scala也支持这个特性，使用对象中的一个"main"方法作为应用的入口点。要用Scala创建一个命令行应用，需要增加一个"main"方法，它取一个字符串数组作为输入参数。编译代码后，运行scala命令并指定对象名来执行这个应用。

下面是一个简短的命令行应用的例子，它会打印当前日期。这里包含了创建文件、编

译文件以及作为应用执行文件等步骤，所有步骤都在shell中完成。入口点是对象中定义的一个"main"方法：

```
$ cat > Date.scala
object Date {
  def main(args: Array[String]) {
    println(new java.util.Date)
  }
}

$ scalac Date.scala

$ scala Date
Mon Sep 01 22:03:09 PDT 2014
```

将这个"Date"对象编译为.class文件（JVM类的二进制格式）之后，就能作为一个应用来执行这个文件。这个例子展示了创建、编译和执行一个命令行应用的基本内容，不过并没有展示输入参数的使用。

下面给出一个新例子，它模拟了Unix命令cat，将把一个文件的内容打印到控制台。这个应用取一个或多个文件名（或路径），把它们分别打印到控制台：

```
$ cat > Cat.scala
object Cat {
  def main(args: Array[String]) {
    for (arg <- args) {
      println( io.Source.fromFile(arg).mkString )
    }
  }
}

$ scalac Cat.scala

$ scala Cat Date.scala
object Date {
  def main(args: Array[String]) {
    println(new java.util.Date)
  }
}
```

这一次我们要使用输入参数了。这里使用Scala库中io.Source对象（现在终于可以为它正名了）的fromFile方法读取各个文件，另外使用集合方法mkString将各行转换为一个String来完成打印。

从某个方面来讲，最好的命令行应用就像是纯函数：它们要读取输入、完成处理，然后写输出。类似Scala集合的操作，命令行应用只完成一个任务，不过如果串链在一起，就会有很多新的机会和可能性。用Scala编写的命令行应用可能并不会取代内置的

工具和shell脚本，因为它们启动时间很长（这是JVM众所周知的一个问题），而且需要更多的内存，这使它们不能适用于所有环境。不过，基于命令行应用，编写命令行工具会更有意思，而且这也是学习语言的一种很好的方法。建议花点时间用Scala重写你喜欢的（而且比较短的）shell脚本。这样可以帮助你继续学习和尝试这个语言，你可能会发现你的Scala应用比用其他语言编写的应用更简短，而且更稳定。

作为这一节的总结，对象不仅可以作为基于实例的类的一个全局替代，而且是一种创建命令行应用的方法。结合类和伴生对象时，通过它们的协同，可以创建更简洁而且更可读的解耦合应用。

既然你已经对结合使用类和对象有了一些经验，下面就来学习如何使用case类自动完成它们的交互。

Case类

Case类（case class）是不可实例化的类，包含多个自动生成的方法。它还包括一个自动生成的伴生对象，这个对象也有其自己的自动生成的方法。类中以及伴生对象中的所有这些方法都建立在类的参数表基础上，这些参数用来构成equals实现和toString等方法，equals会迭代地比较各个字段，toString方法会简洁地打印类名和它的所有字段值。

Case类对数据传输对象很适用，根据所生成的基于数据的方法，这些类主要用于存储数据。不过，它们不适合层次类结构，因为继承的字段不能用来构建工具方法。常规的类扩展一个case类时，可能会导致生成的方法得到不合法的结果，它无法考虑子类增加的字段。不过，如果希望一个类的字段是有限的，而且这些自动生成的方法很有用，那么case类就很适用。

要创建一个case类，只需要在类定义前面增加关键字case。

语法：定义一个Case类

```
case class <identifier> ([var] <identifier>: <type>[, ... ])  
    [extends <identifier>(<input parameters>) ]  
    [{ fields and methods }]
```

注意：val关键字可以用于Case类参数吗？

默认地，case类将参数转换为值字段，所以没有必要在它们前面加上val关键字。如果需要一个变量字段，则可以使用var关键字。

表9-1 展示了为case类自动生成的类及对象方法。

表9-1：生成的case类方法

方法名	位置	描述
apply	对象	这是一个工厂方法，用于实例化case类
copy	类	返回实例的一个副本，并完成所需的修改。参数是类的字段，默认值设置为当前字段值
equals	类	如果另一个实例中的所有字段与这个实例中的所有字段匹配，则返回true。也可以用操作符==来调用
hashCode	类	返回实例字段的一个散列码，对基于散列的集合很有用
toString	类	将类名和字段呈现为一个String
unapply	对象	将实例抽取到一个字段元组，从而可以使用case类实例完成模式匹配

Scala编译器会为case类生成方法，不过除了这些方法是自动生成的，实际上它们并没有其他特殊之处。完全可以不使用case类，而是自己来增加方法和伴生对象。Case类的好处是其方便性，因为如果要为每一个基于数据的类正确地编写所有这些方法，这可能需要大量的工作和维护。另外，Case类还可以增加一些一致性，因为所有case类都有相同的特性。

现在我们已经全面地分析了case类能够做什么，下面来看它们的具体使用。在这个例子中，我们将创建一个case类，查看可以得到多少自动生成的方法：

```
scala> case class Character(name: String, isThief: Boolean)
defined class Character

scala> val h = Character("Hadrian", true) ①
h: Character = Character(Hadrian,true) ②

scala> val r = h.copy(name = "Royce") ③
r: Character = Character(Royce,true)

scala> h == r ④
res0: Boolean = false

scala> h match {
    |   case Character(x, true) => s"$x is a thief" ⑤
    |   case Character(x, false) => s"$x is not a thief"
    |
}
res1: String = Hadrian is a thief
```

- ① 这是伴生对象的工厂方法Character.apply()。
- ② 生成的toString方法（由REPL打印）是实例字段的一个简洁而简单的表示。

- ③ 对于第二个字段，第二个实例共享相同的值，所以在copy方法中只需要为第一个字段指定一个新值。
- ④ 如果两个实例都非null，==操作符会触发一个实例的equals方法，对于所生成的基于字段比较的方法，这是一个很有用的快捷方式。
- ⑤ 利用伴生对象的unapply方法，我们可以将实例分解为不同部分，绑定第一个字段（见第3章“值绑定”一节），并使用一个字面量值匹配第二个字段。

这个例子中使用的所有生成方法都依赖于case类有两个字段，name和isThief（根据case类参数）。如果你的case类扩展了另一个类，后者有其自己的字段，但是如果我们将这些字段增加为case类参数，生成的方法就不能利用这些字段。这一点很重要，使用case类之前必须了解。

如果case类不需要考虑父类的字段，你会发现它在你的代码中非常有用。利用case类，不再需要你编写太多的样板代码，case类的toString方法很有帮助，利用这个方法，调试和日志记录会更容易，总的来讲，会让面向对象编程更有意思。

就我个人来讲，我就更愿意使用case类而不是常规类来完成数据存储，另外我喜欢用对象而不是类来编写大部分函数。没错，我会用对象和trait编写函数，因为就像利用case类可以方便地管理数据一样，采用同样的方式，利用trait可以方便地重用函数。下一节我们将介绍trait，这是这一章介绍的最后一类。

Trait

trait是一种支持多重继承的类。类、case类、对象，以及（没错）trait都只能扩展不超过一个类，但是可以同时扩展多个trait。不过，与其他类型不同，trait不能实例化。

trait看起来与所有其他类型的类是一样的。不过，类似对象，trait不能有类参数。但与对象不同的是，trait可以有类型参数，这使它们有很好的可重用性。

要定义一个trait，可以在通常使用class关键字的位置上使用trait关键字。

语法：定义trait

```
trait <identifier> [extends <identifier>] [{ fields, methods, and classes }]
```

还记得前面例子中创建的HtmlUtils对象吗（见本章前面“对象”一节）？下面把这个对象实现为一个trait：

```
scala> trait HtmlUtils {
```

```
| def removeMarkup(input: String) = {
|   input
|     .replaceAll("""</?\w[^>]*>""", "")
|     .replaceAll("<.*>", "")
|   }
| }
defined trait HtmlUtils

scala> class Page(val s: String) extends HtmlUtils {
|   def asPlainText = removeMarkup(s)
| }
defined class Page

scala> new Page("<html><body><h1>Introduction</h1></body></html>").asPlainText
res2: String = Introduction
```

Page类现在直接使用removeMarkup方法，而没有指定对象名。

这可以正常工作，不过类版本的HtmlUtils也可以做同样的工作。下面会更有意思，我们要增加第二个trait。这一次我们将使用一个新关键字with，扩展第二个以及更多trait时必须使用这个关键字：

注意： trait在父类后面

如果扩展了一个类以及一个或多个trait，需要先扩展这个类，然后再使用with关键字增加trait。如果指定了父类，父类必须放在所有父trait前面。

```
scala> trait SafeStringUtils {
|   // Returns a trimmed version of the string wrapped in an Option,
|   // or None if the trimmed string is empty.
|   def trimToNone(s: String): Option[String] = {
|     Option(s) map(_.trim) filterNot(_.isEmpty)
|   }
| }
defined trait SafeStringUtils

scala> class Page(val s: String) extends SafeStringUtils with HtmlUtils {
|   def asPlainText: String = {
|     trimToNone(s) map removeMarkup getOrElse "n/a"
|   }
| }
defined class Page

scala> new Page("<html><body><h1>Introduction</h1></body></html>").asPlainText
res3: String = Introduction

scala> new Page("  ").asPlainText
res4: String = n/a

scala> new Page(null).asPlainText
```

```
res5: String = n/a
```

这个新的Page类更为健壮，它扩展了两个trait，可以处理null或空串，相应地返回消息n/a。

如果你已经熟悉JVM，可能想知道Scala如何用trait支持多重继承。毕竟，JVM类只能扩展一个父类。答案是，尽管这个语言理论上支持多重继承，但是编译器实际上会创建各个trait的副本，形成类和trait组成的一个“很高”的单列层次体系。所以，如果一个类扩展了类A以及trait B和C，编译到.class二进制文件时，实际上它会扩展一个类，这个类又扩展了另一个类，后者进一步扩展了下一个类。

这里将所扩展的类和trait的水平列表变换为一个垂直的链，各个类分别扩展另一个类，这个过程称为线性化（linearization）。这是一种复制机制，用于在只支持单重继承的执行环境中支持多重继承。JVM只支持单重继承，这一点确保了所有类层次体系是不确定的，所以不会混淆两个有竞争成员的trait。

注意：如果trait有竞争成员会发生什么？

如果一个类导入两个trait，它们有相同的字段或成员，但是没有override关键字，这个类的编译就会失败。这个编译错误就好像你要扩展一个类，并且提供了你自己的方法，但是没有增加override关键字一样。对于trait，可以增加一个公共的基类，然后用override关键字覆盖字段和方法，这样可以确保trait能够由同一个类扩展。

关于线性化，要理解最重要的一点是Scala编译器以什么顺序组织trait和可选的类来相互扩展。多重继承的顺序为从右到左（从最低的子类到最高基类）。

因此，如果一个类定义为class D extends A with B with C，其中A是一个类，B和C是trait，将由编译器重新实现为class D extends C extends B extends A。最右trait是所定义的类的直接父类，这个类或第一个trait成为最后一个父类。

需要记住的东西太多了，所以下面写一个简单的测试来验证这个顺序：

```
scala> trait Base { override def toString = "Base" }
defined trait Base

scala> class A extends Base { override def toString = "A->" + super.toString }
defined class A

scala> trait B extends Base { override def toString = "B->" + super.toString }
defined trait B

scala> trait C extends Base { override def toString = "C->" + super.toString }
defined trait C
```

```
scala> class D extends A with B with C { override def toString = "D->" +  
    super.toString }  
defined class D  
  
scala> new D()  
res50: D = D->C->B->A->Base
```

D中被覆盖的toString方法会打印类名，然后追加其父类实现的输出。幸运的是，它的所有父类也覆盖了这个方法，所以我们可以看到调用方法的具体顺序。首先调用了D中的toString，然后是trait C、trait B、类A以及最后公共基类Base的toString方法。

线性化的过程看起来可能有些奇怪，不过这是在支持多重继承的语言理论与不支持多重继承的环境实践之间的一个有效的折中。它还提供了一个确定调用的可靠方法，因为所构建的层次结构可以确保在编译时就能确定方法，而不是到运行时才确定。

线性化的另一个好处是，可以编写trait来覆盖共享父类的行为。下面给出一个例子，这里有一个基类以及一些trait，它们与子类结合时可以增加额外的功能。这个例子相当详细，所以我们将分两部分来介绍。首先，下面给出父类以及和扩展这个父类的两个trait：

```
scala> class RGBColor(val color: Int) { def hex = f"$color%06X" }  
defined class RGBColor  
  
scala> val green = new RGBColor(255 << 8).hex  
green: String = 00FF00  
  
scala> trait Opaque extends RGBColor { override def hex = s"${super.hex}FF" }  
defined trait Opaque  
  
scala> trait Sheer extends RGBColor { override def hex = s"${super.hex}33" }  
defined trait Sheer
```

这两个trait（Opaque和Sheer）扩展了RGBColor类，为父类的红绿蓝颜色增加了一个透明度。这个额外的字节在计算机图形学中通常称为一个alpha通道，所以trait把一个RGB颜色值转换为一个采用十六进制格式的RGBA（a表示alpha）颜色值。

现在来使用这些新trait。我们将扩展父类以及扩展了父类的某个trait。如果只是扩展trait，就没有办法向RGBColor传递类参数。因此，我们要扩展父类以及增加了这个功能的trait：

```
scala> class Paint(color: Int) extends RGBColor(color) with Opaque  
defined class Paint  
  
scala> class Overlay(color: Int) extends RGBColor(color) with Sheer  
defined class Overlay
```

```
scala> val red = new Paint(128 << 16).hex  
red: String = 800000FF  
  
scala> val blue = new Overlay(192).hex  
blue: String = 0000C033
```

由于trait线性化的顺序是从右到左，所以“Paint”的层次体系是“Paint” → “Opaque” → “RGBColor”。增加到Paint类的类参数用来初始化RGBColor类，而Paint和RGBColor之间的Opaque trait覆盖了hex方法来增加额外的功能。

换句话说，Paint类会输出一个不透明的颜色值，Overlay会输出一个透明的颜色值。利用trait线性化，我们可以插入额外的功能。

现在你应该知道如何定义trait以及类如何扩展trait。不过，要明确具体在哪里使用以及何时使用，还需要花一些时间，需要一些经验。Trait看起来类似于抽象类，与基于实现的Java接口很类似，不过需要清楚如何利用线性化构成类的层次体系，这一点很重要。

如果你还是不确定如何使用trait，接下来的两小节还会介绍一些特性，了解这些特性后你就会明白。我们会介绍一种方法限制trait只能用于某些类，因为有些情况下你可能想要依赖某个类的字段和方法，但并不直接扩展这个类。我们还会介绍trait不仅能在类定义中，还可以在类实例化中使用，从而提供内置的依赖注入。

自类型

自类型（self type）是一个trait注解，向一个类增加这个trait时，要求这个类必须有一个特定的类型或子类型。有自类型注解的trait不能增加到未扩展指定类型的类。在某种程度上，这可以保证trait总是扩展该类型，尽管不是直接扩展。

自类型的一种流行的用法是用trait为需要输入参数的类增加功能。trait扩展有输入参数的类不容易，因为trait本身不能有输入参数。不过，trait可以用一个自类型注解声明自己是这个父类的一个子类型，然后增加相应的功能。

要把自类型增加到trait定义中开始大括号的后面，包括一个标识符、请求的类型和一个箭头(=>)。有自类型的trait可以访问该类型的字段，就好像显式扩展了那个类型一样。

语法：定义自类型

```
trait ..... { <identifier>: <type> => .... }
```

自类型中使用的标准标识符是"self"，不过也可以使用任何其他标识符。也就是说，除了类似this的关键字，可以使用任何标识符。使用常用标识符"self"的好处是，这样会让你的代码对其他Scala开发人员更可读。

下面是一个trait的例子，这里使用了一个自类型来确保它总是指定类型的一个子类型：

```
scala> class A { def hi = "hi" }
defined class A

scala> trait B { self: A =>    ①
|   override def toString = "B: " + hi
| }
defined trait B

scala> class C extends B
<console>:9: error: illegal inheritance; ②
  self-type C does not conform to B's selftype B with A
      class C extends B
                           ^

scala> class C extends A with B    ③
defined class C

scala> new C()
res1: C = B: hi  ④
```

- ① trait B有一个自类型，这要求这个trait只能增加到指定类型（类A）的一个子类型。
- ②不过为了证明这一点，下面试着用trait B定义一个类，但是没有扩展所指定的类。尝试失败！
- ③ 这一次，trait B直接扩展了指定的类型A，所以它的自类型要求得到满足。
- ④ 实例化类C时，会调用B.toString，它再调用A.hi。实际上B trait在这里用作为A的一个子类型，所以可以调用它的方法。

这个例子展示了自类型对trait所增加的限制。不过，它并没有真正体现出自类型作为一个重要特性的好处，因为trait B也可以已经直接扩展A。

下面来看一个例子，这里展示了自类型的好处。我们将定义一个要求有参数的类，然后创建一个trait，它只能用来扩展这个类：

```
scala> class TestSuite(suiteName: String) { def start() {} } ①
defined class TestSuite

scala> trait RandomSeeded { self: TestSuite =>    ②
```

```

|   def randomStart() {
|     util.Random.setSeed(System.currentTimeMillis)
|     self.start()
|   }
|
defined trait RandomSeeded

scala> class IdSpec extends TestSuite("ID Tests") with RandomSeeded { ❸
|   def testId() { println(util.Random.nextInt != 1) }
|   override def start() { testId() }
|
|   println("Starting...")
|   randomStart()
|
defined class IdSpec

```

- ❶ 这是基类TestSuite，它有一个输入参数。
- ❷ 我们的trait需要调用TestSuite.start()同，但是不能扩展TestSuite，因为它需要硬编码的输入参数。通过使用一个自类型，就可以认为这个trait是TestSuite的一个子类型，而不需要显式声明。
- ❸ 测试类IdSpec定义我们的自类型trait为一个子类，可以调用其randomStart()方法。

利用自类型，trait可以扩展一个类而不用指定其输入参数。另外这也是一个为trait增加限制和/或需求的安全方法，可以确保它们只能在特定的上下文中使用。

以上我们研究的特性可以帮助确保得到更安全、更稳定的类型定义，下面转向另一个有意思的内容：实例化类时增加类型定义。

用Trait实例化

这一章中我们使用了trait，具体做法是在类定义中使用extends或with关键字让类扩展trait。扩展trait的类会得到这个trait的字段和方法，而不论这些字段和方法是由这个trait实现的，还是从其子类型继承的。

使用trait的另一种方法是在类实例化时为类增加trait。即使所定义的类不依赖或者甚至不知道一个给定的trait，它也可以利用这个trait的功能。关键是类实例化时增加的trait会扩展这个类，而不是由类扩展trait。Trait线性化的顺序为从左到右，这包括实例化的类，所以所有trait都会扩展这个类，而不是反过来扩展trait。

可以使用with关键字为类增加一个或多个trait。这里不能使用Extends关键字，这是有道理的。类实际上没有扩展trait，而是将被trait扩展。

下面来验证用trait实例化的类会成为该trait的一个基类，这里使用了上一节学习的自类型。下面的例子中，类由一个trait扩展，这个trait有该类的自类型，从而确保这个trait会扩展这个类：

```
scala> class A
defined class A

scala> trait B { self: A => }
defined trait B

scala> val a = new A with B
a: A with B = $anon$1@26a7b76d
```

我们的新实例为a，它的类名指定为\$anon\$1，这是结合了一个数字的"anonymous"的缩写形式。这个实例的类实际上还是匿名的，因为它包含类和trait的一个组合，但并没有正式包含在任何命名类定义中。更确切地说，我们创建了一个实例，其中trait B扩展了trait A。

用trait实例化的真正意义在于可以为现有的类增加新的功能或配置。这个特性通常称为依赖注入（dependency injection），因为父类所依赖的具体功能是在类定义之后增加的，所以会在类实例化时为类“注入”这个特性。这也说明，这个类的两个实例可以在完全不同的配置下操作，因为实例化时可能增加了不同的可配置trait。

Java开发人员可能比较熟悉 Spring (<http://spring.io/>) 或Google Guice (<http://bit.ly/lsgoogleguice>)，它们通过定制Java注解和初始化模块实现了一个类似的功能。不过，Scala的trait不要求特定的注解或特殊的包来实现依赖注入。只需要用另一个trait初始化给定的类，就能得到一个实现了依赖注入的类。

下面来使用依赖注入，这里考虑大多数应用中都很常用的一个面向数据的类：User，我们会用一些神秘的新方式修改它的输出：

```
scala> class User(val name: String) {
|   def suffix = ""
|   override def toString = s"$name$suffix"
| }
defined class User

scala> trait Attorney { self: User => override def suffix = ", esq." }
defined trait Attorney

scala> trait Wizard { self: User => override def suffix = ", Wizard" }
defined trait Wizard

scala> trait Reverser { override def toString = super.toString.reverse }
defined trait Reverser
```

```
scala> val h = new User("Harry P") with Wizard
h: User with Wizard = Harry P, Wizard

scala> val g = new User("Ginny W") with Attorney
g: User with Attorney = Ginny W, esq.

scala> val l = new User("Luna L") with Wizard with Reverser
l: User with Wizard with Reverser = draziW ,L anul
```

这里有3个新用户，他们并不是虚构的科幻人物（尽管名字与科幻人物很相似，但这纯粹是巧合），他们得到了新头衔，或者得到了可以打印名字的新方法。后缀"Wizard"和"esq"硬编码写在trait中，不过实例化时会增加到不同的用户实例。

实例化时为类增加trait是一种定义类来完成相同工作的替代快捷方式。在我们的例子中，也可以结合类和trait定义3个新的单独的类，并使用这些类。不过，利用这些实例化trait，可以更为灵活和简单，避免编写不必要的代码。通过在实例化时增加trait，可以得到各种不同的功能组合。

导入实例成员

在第8章“访问包装类”一节中，我们介绍了如何使用import关键字增加外部包中的类，这样一来，不加包前缀也可以访问这个类。这一章介绍的是其他面向对象特性（具体来说，就是对象、case类和trait），最后我们再来看一种使用命名空间导入的方法。

import关键字还可以用来将类和对象的成员导入当前命名空间，这样就可以直接访问这些成员，而不需要指定类实例或对象名。

导入类和对象成员的语法与导入包装类是一样的。可以按名导入一个类实例的单个成员，或者用下划线字符导入一组字段和方法。导入字段和方法不会覆盖私密性控制，所以只能导入那些可以正常访问的字段和方法。

下面的例子导入了一个case类成员以便访问：

```
scala> case class Receipt(id: Int, amount: Double, who: String, title: String)
defined class Receipt

scala> {
    |   val latteReceipt = Receipt(123, 4.12, "fred", "Medium Latte")
    |   import latteReceipt._
    |   println(s"Sold a $title for $amount to $who")
    |
}
Sold a Medium Latte for 4.12 to fred
```

通过从一个名字很长的值（`latteReceipt`）导入字段，我们可以在`println`语句中直接访问这些字段，这里的代码行更为简单。

不过，需要处理多个实例时，导入类和`case`类实例的成员可能很困难。从多个类导入成员会出现命名冲突，所以要让`import`语句尽量靠近使用这些成员的代码，这是一个务必遵循的实践做法。

可以用同样的方式导入对象的字段和方法。实际上，在前几章我们已经见过导入对象成员的例子。第6章“Java和Scala集合兼容性”一节中导入了`collection.JavaConverters`对象的成员来展示Java和Scala的兼容性函数。类似地，第7章“Future集合”中导入了`concurrent.ExecutionContext.Implicits`对象的全局字段来支持创建新`future`。

作为对象导入的一个例子，下面将增加`util.Random`对象的所有方法。这个对象扩展了`util.Random`类，提供了一个单例全局实例，不需要为随机数生成设置新种子时这就很有用：

```
scala> import util.Random._  
import util.Random._  
  
scala> val letters = alphanumeric.take(20).toList.mkString  
letters: String = MwDR3EyHa1cr0JqsP9Tf  
  
scala> val numbers = shuffle(1 to 20)  
numbers: scala.collection.immutable.IndexedSeq[Int] = Vector(5, 10, 18, 1,  
16, 8, 20, 14, 19, 11, 17, 3, 15, 7, 4, 9, 6, 12, 13, 2)
```

`alphanumeric()`: `Stream`和`shuffle(Traversable)`方法是`util.Random`对象（及父类）的成员，在这里可以直接访问，而不需要增加对象前缀。

导入实例成员是一种优化代码的很好的方法。不过，要注意避免命名冲突，另外还要注意不要降低代码的可读性。如果读代码的人不能清楚地知道你使用的导入成员来自哪里，就要考虑把`import`语句放在相应代码的附近。

小结

类仍是Scala应用的核心构件，不过可以由trait增强，另外对象可以作为补充或代替。利用trait，可以支持类的多重继承，从而进一步扩展代码的可用性。在类定义或实例化时，根据trait的不同顺序可以组合出大量不同的功能。对象没有trait那么灵活，不过它提供了一种内置的单例机制，与Java的单例或静态成员和类相比，则远没有那么刻板。

更准确地说，case类应当和类一同作为Scala应用的核心构件。在很多应用中，开发人员大量使用case类来取代类，因为尽管case类在派生方面存在限制，但它们能提供大量额外的特性，利远大于弊。更确切地讲，case类不只是类。它们可以生成不可见的伴生对象。可以认为case类实例等同于类实例，不过总的来讲case类要大于类。

每个类实例和字面量都对应一个特定的类型。在这一章以及前一章中，你已经了解了如何用Scala创建你自己的类型。不过类型不只是类。有一个类型参数的类是一个类型，不过每次用一个类型参数实例化时，得到的实例也是类型。可以认为List、List[Int]和List[String]都对应同一个类，尽管它们有不同的类型。对于一个给定的类，实例化时如果增加了一个trait，这也对应同一个类，不过有不同的类型。

下一章我们会区别类和类型的不同，还会学习Scala表达性语法所隐藏的新类型，另外介绍一些方法来提升你的类的灵活性和规范性。

中场休息——配置你的第一个Scala项目

以上已经介绍完这一章的主要内容。不过，开始练习之前，我们需要稍稍休息一下，下面会配置你的第一个Scala项目。目前为止我们的做法都是在REPL中编辑，并且/或者直接执行.scala文件，构建你的应用时这种做法可能并不适合。

由于引入对象作为应用的入口点，现在我们有了一种执行编译代码的机制。可以编译不同文件和包中的类，然后从应用访问这些类。

现在需要一种方法来组织这些依赖文件，管理我们的项目。我们运行的任何代码都可能有外部依赖文件，如Maven存储库中托管的Java和Scala库、本地库依赖文件、其他文件和包中的内部依赖文件，以及Scala库和运行时库中的依赖文件。通过使用一个依赖文件管理和构建工具，可以基于这些依赖文件完成编译和执行，而由这个工具处理库的下载和路径配置。还可以充分利用依赖文件管理将我们的项目导入到一个集成开发环境（Integrated Development Environment, IDE），从而可以从这个IDE或者从命令行编辑和运行代码。

如果你已经完成第7章的练习，你会知道我所指的是什么工具。这就是Simple Build Tool（SBT），这是一个基于Scala的依赖文件管理和构建工具，可以用来配置、编译和执行项目中的Scala代码。如果还没有安装这个工具，参见第7章练习中的“SBT难学吗？”，其中给出了安装SBT的有关说明。

如果已经安装了SBT，下面创建一个空的项目目录。这里不打算把项目命名为

“MyProject”或“Project1”，所以使用“HardyHeron”作为项目名，也可以把它用作为目录名。

在这个新目录中，在shell下运行以下命令，增加一个命令行应用并执行这个应用：

```
[HardyHeron] > mkdir -p src/main/scala  
[HardyHeron] > cat > src/main/scala/Hello.scala  
  
object Hello {  
    def main(args: Array[String]) {  
        println("Hello from SBT")  
    }  
}  
  
[HardyHeron] > sbt run  
[info] Set current project to hardyheron (in build file:~/HardyHeron/)  
[info] Updating {file:~/HardyHeron/}harryheron...  
[info] Resolving org.fusesource.jansi#jansi;1.4 ...  
[info] Done updating.  
[info] Compiling 1 Scala source to ~/HardyHeron/target/scala-2.10/classes...  
[info] Running Hello  
Hello from SBT  
[success] Total time: 3 s, completed June 6, 2014 10:38:08 PM  
  
[HardyHeron] >
```

注意到了吗？我们能编译和运行一个应用，甚至不需要构建脚本。SBT更支持约定而不是配置。如果没有一个特定的构建脚本，它会在*src/main/scala*下查找主体Scala代码，并在*src/test/scala*下查找只用于测试的Scala代码。命令sbt run会结合“run”命令调用SBT，这会执行它在代码基中能找到的所有命令行应用。

现在来增加一个构建脚本。尽管我们很清楚不需要构建脚本也能编译和运行应用，不过如果开始增加外部依赖文件，也就是外部Java和Scala库，那么还是需要这样一个构建脚本。下面就来增加一个构建脚本以简化以后增加依赖文件的过程。

SBT支持用其类Scala脚本语言编写构建脚本，存储在项目根一级的文件*build.sbt*中。它还支持用Scala编写构建脚本，存储在“project”目录中，其中包含一个对象扩展其*sbt.Build*父类。这两种类型的SBT构建脚本都会使用一些非标准的Scala操作符，如赋值(:=)和依赖文件分组(%)。幸运的是，如果你在上下文中看到这些操作符，通常都能很清楚地知道它们的含义。

写这本书时，目前SBT文档建议使用第一种方法，即在项目的根目录中编写一个*build.sbt*。在这个教程中，我将采用第二种方法，这可能不是推荐的做法，但是可以避免使用SBT的.sbt文件格式语言，而使用第二种方法中使用的常规Scala语法。

在命令行上运行这些命令来创建一个基于Scala的构建脚本，并执行我们的“Hello”应用。首先来看一个外部依赖库：ScalaTest测试框架。

```
[HardyHeron] > cat > project/HardyHeronBuild.scala

import sbt._ ①
import sbt.Keys._

object HardyHeronBuild extends Build ②
{
    val hardyHeronDependencies = List(
        "org.scalatest" % "scalatest_2.11" % "2.2.1" % "test" ③
    )

    val hardyHeronSettings = List( ④
        name := "HardyHeron",
        version := "1.0",
        scalaVersion := "2.11.2",
        libraryDependencies := hardyHeronDependencies
    )

    override lazy val settings = super.settings ++ hardyHeronSettings ⑤
}
```

```
[HardyHeron] >
```

```
[HardyHeron] > sbt compile
[info] Loading project definition from ~/HardyHeron/project
[info] Compiling 1 Scala source to ~/HardyHeron/project/target/scala-2.10/
  sbt-0.13/classes...
[info] Set current project to hardyheron (in build file:~/HardyHeron/)
[info] Updating {file:~/HardyHeron/}hardyheron...
[info] Resolving jline#jline;2.12 ...
[info] downloading http://repo1.maven.org/maven2/org/scalatest/   ⑥
  scalatest_2.11/2.2.1/scalatest_2.11-2.2.1.jar ...
[info]  [SUCCESSFUL ] org.scalatest#scalatest_2.11;2.2.1!scalatest_2.11.jar
  (bundle) (5232ms)
[info] Done updating.
[success] Total time: 7 s, completed June 7, 2014 12:49:44 AM
```

```
[HardyHeron] > sbt "run Hello" ⑦
[info] Loading project definition from ~/HardyHeron/project
[info] Set current project to hardyheron (in build file:~/HardyHeron/)
[info] Running Hello
Hello from SBT
[success] Total time: 0 s, completed June 7, 2014 12:58:43 AM
```

```
[HardyHeron] >
```

- ① 在基于Scala的构建文件中导入sbt包和sbt.Keys的内容。这会得到Build 基类、属性名（即“设置”，如name和version），以及特殊的SBT操作符（如:=、%和%%）。

- ② 对象名和文件名由你决定。SBT会在这里查找sbt.Build类的子类。
- ③ 这是在SBT中定义Maven/Ivy库（即“artifact”）依赖文件的标准格式。这4个组件依次是组（group）、工件（artifact）、版本（version），以及所应用的SBT组件，这里指示这个库只用于测试。还有一些公共Maven存储库搜索引擎用于查找库，允许将库格式化为SBT中的一个依赖库。最后要指出的组件是一个双百分号%，这会指示SBT在工件名后面追加_2.11（我们使用的Scala的主版本）。Scala库通常针对特定的Scala主版本完成编译，如2.10和2.11，这个格式是对标准Scala的补充，可以指示库的目标版本。
- ④ 这只是一个设置列表，使用了操作符:=，将根据sbt.Keys._中的键来定义设置。
- ⑤ 直接覆盖的唯一字段是懒值设置，这是一个常规的SBT配置List。我们首先覆盖父类的设置，然后增加我们项目的设置。
- ⑥ ScalaTest库从公共Maven存储库下载，安装在用户目录的一个缓存中，下一次执行应用时会增加到JVM“classpath”。
- ⑦ 下面运行“Hello”来验证构建是成功的。由于从命令行运行这个应用，所以，要使用双引号包围run-main Hello，这样SBT就可以把它解释为单个参数。

既然已经有了一个能正常工作的构建脚本，下面将项目导入一个IDE。这样可以得到现代IDE提供的立即编译、代码分析和可发现性等特性。如果你更熟悉文本编辑环境，如Sublime Text、Vim或Emacs，现在应当花些时间来熟悉如何使用一个IDE。尽管这些文本编辑器提供了一些扩展，可以支持Scala开发，不过你会发现，IDE可以预测和验证你写的每一行代码，在IDE中使用这个静态类型语言会让你有更高的开发效率。

对于这个教程，我们将使用IntelliJ IDEA 13（或更高版本的）IDE。可以免费下载这个非常棒的社区版本（<http://bit.ly/ls-intellij>）。确保至少安装版本13，另外要安装IntelliJ的Scala插件。不需要任何其他第三方SBT插件，因为面向Scala的IntelliJ IDEA 13可以直接打开SBT项目。

要导入这个项目，首先打开IntelliJ IDEA，选择“Open Project”，然后选择我们的“HardyHeron”项目目录。在“Import Project from SBT project”对话框中，选择所有选项，并把“Project SDK”设置为Java 1.8安装版本（即Java 8）。如果这个版本没有出现，可能需要单击“New……”配置IntelliJ IDEA来支持你的Java安装版本。图9-1展示了选择所有选项之后的对话框。



图9-1：IntelliJ IDEA 13的导入项目对话框

导入这个项目之后，IntelliJ IDEA中会出现一个项目窗口。在“Project”视图中导航（或者如果没有打开这个视图，可以从View → Tool Windows选择），直到找到Hello.scala文件，打开这个文件。应该可以看到一个类似图9-2的视图，“Project”视图中显示了项目结构，“Hello”类的源代码显示在右边，并且提供了语法突出显示。

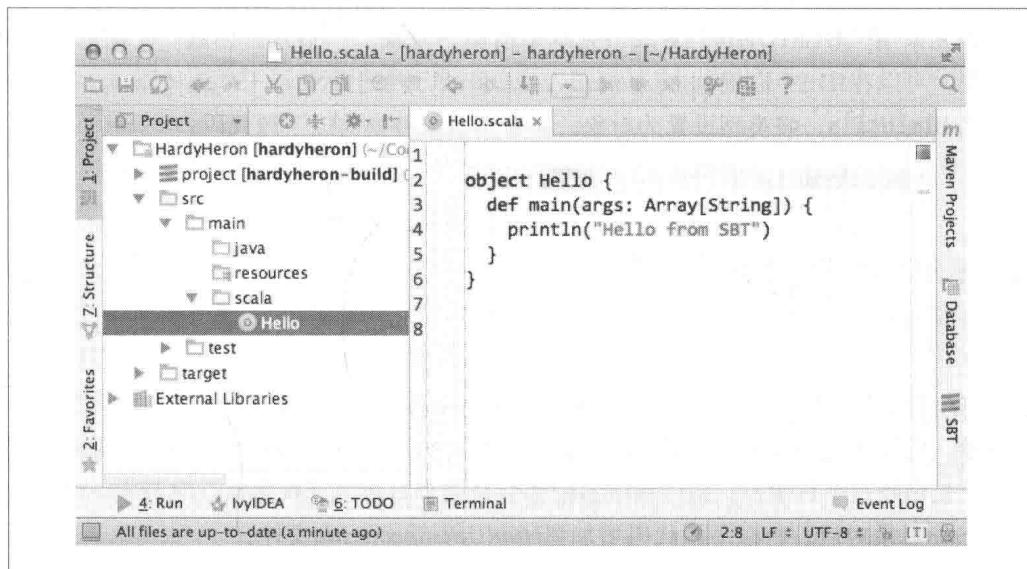


图9-2：IntelliJ IDEA中查看项目

既然已经在IDE中加载了源代码，下面从IDE运行项目。在“Hello.scala”源代码视图

中右键单击鼠标，选择Run Hello。可以看到“Hello”应用的输出消息出现在一个新视图中，即源代码视图下面的“Run”视图。

现在你已经有了一个可以在IntelliJ IDEA中运行的SBT项目，而且可以在这里增加和编辑你自己的类、对象和trait。如果你以前没有使用过这个IDE，可以访问IntelliJ IDEA产品网站 (<http://bit.ly/ls-idea>) 来了解这个IDE的更多信息。其中提供了大量截屏图、教程和指南，对这一章练习中可能遇到的很多特性做出了解释。我们只使用了这个IDE中最基本的特性，不过很有必要了解如何利用这个IDE的核心特性，这会帮助你进一步探索Scala语言。

既然已经可以在IDE中运行Scala，下面来完成一些练习。

练习

1. 下面说明如何在Scala中利用ScalaTest框架编写一个单元测试。这个练习将为IDE增加一个测试，执行这个测试，并验证是否能得到成功的结果。如果你已经熟悉如何在IDE中执行测试，这个练习可能相当简单。为了更好地理解决Test框架，建议你暂时先不考虑这个练习，可以先看看ScalaTest网站 (<http://www.scalatest.org/>) 的官方文档。

首先来看“`HtmlUtils`”对象（见本章前面“对象”一节）。创建一个新的Scala类，可以在IDE中右键单击`src/main/scala`目录，并选择New→Scala Class。键入类名`HtmlUtils`，将类型设置为对象。将单例对象替换为以下源代码：

```
object HtmlUtils {  
    def removeMarkup(input: String) = {  
        input  
            .replaceAll("""</?\w[^>]*>""", "")  
            .replaceAll("<.*>", "")  
    }  
}
```

这个新的`HtmlUtils.scala`文件应当位于`src/main/scala`，这是项目中源代码的根目录。现在再在`src/test/scala`目录下增加一个新的“`HtmlUtilsSpec`”类（必要时，需要先创建这个目录）。SBT和IntelliJ都会在这个目录下查找测试，它与主`src/main/scala`目录对应。将以下源代码增加到`HtmlUtilsSpec.scala`文件：

```
import org.scalatest._  
  
class HtmlUtilsSpec extends FlatSpec with ShouldMatchers {  
  
    "The Html Utils object" should "remove single elements" in {  
        HtmlUtils.removeMarkup("<br/>") should equal("")  
    }  
}
```

```
}

it should "remove paired elements" in {
    HtmlUtils.removeMarkup("<b>Hi</b>") should equal("Hi")
}

it should "have no effect on empty strings" in {
    val empty = true
    HtmlUtils.removeMarkup("").isEmpty should be(empty)
}

}
```

这里只使用了这个包的FlatSpec和ShouldMatchers类型，不过我们将导入所有类，以便将来可以很容易地增加其他的测试工具（例如，我最喜欢的“OptionValues”）。有很多测试类型可选，类FlatSpec是其中之一，它由Ruby的Rspec (<http://rspec.info/>) 构建。ShouldMatchers为测试增加了should和be操作符，创建了一个域特定的语言，可以使你的测试更可读。

第一个测试与其他测试稍有些不同。对于FlatSpec，文件中的第一个测试应当以一个文本描述开头，描述将在这个文件中测试什么。后面的测试则使用it关键字引用这个描述。这有助于增强测试报告的可读性。

在测试体中，equal操作符确保前面的值应当等于它的参数，在这里就是空串""。如果不相等，会导致这个测试失败，并立即退出。类似地，如果前面的值不是同一个实例，be操作符会使测试失败，这对于比较true、Nil和None等全局实例会很有用。

运行测试时，打开Preferences下的IntelliJ Plugins首选项面板，确保安装了“jnit”插件。这个插件将确保能很容易地查看和浏览你的测试结果。

一旦为项目增加了测试，在IDE中编译这个测试。如果无法编译，或者报告某个错误，如无法找到“ScalaTest”包，要确保你的构建脚本包含了ScalaTest依赖文件，而且可以在IntelliJ “Project”视图的“External Libraries”部分查看。

现在我们来运行这个测试。右键单击测试类的名HtmlUtilsSpec，选择Run HtmlUtilsSpec。执行这个测试只需要几秒的时间，如果测试和源应用输入正确，它们都会成功运行。图9-3展示了测试完成时显示的测试结果。

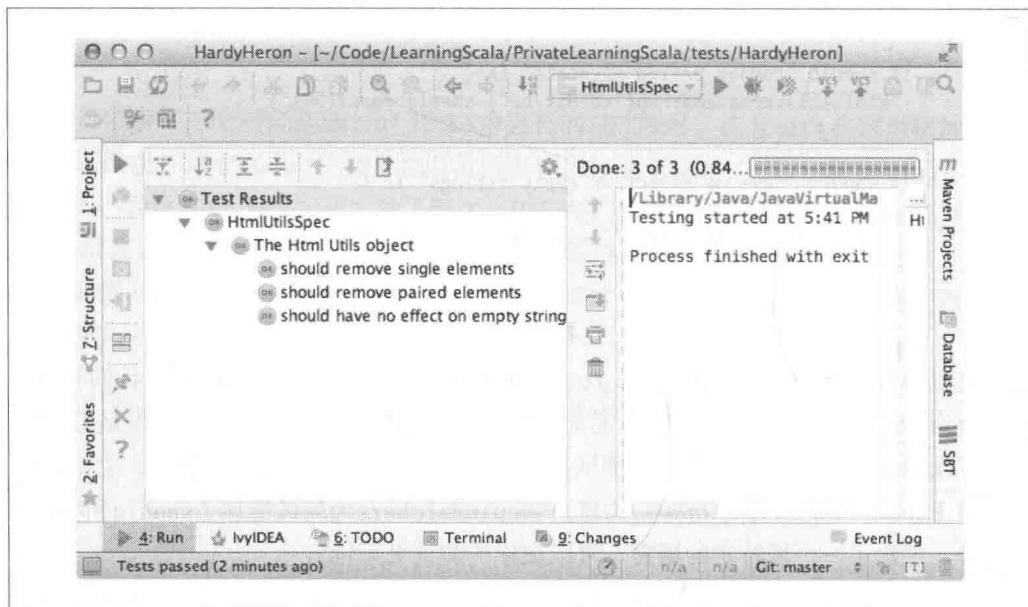


图9-3：IntelliJ IDEA中的测试结果视图

在这个练习的最后，将由你来完成一个具体的练习：为这个`HtmlUtilsSpec`测试类增加其他测试。哪些特性还没有测试？是否支持所有合法的HTML标记？

还有一个问题，要去除“script”标记中包含的JavaScript吗？还是随其余文本一同出现？可以认为这是原版本`HtmlUtils`的一个bug。增加一个测试来验证将去除JavaScript文本，然后运行这个测试。如果失败，修正`HtmlUtils`，并重新运行测试来验证已经得到修正。

祝贺你，现在你已经可以用Scala写测试了！要记住，完成这本书后面的练习时还要坚持写测试，利用这些测试来验证你的答案是否正确，并捕获其中所有（不可预见的）bug。

2. 下面来考虑这一章中的另一个例子。创建一个新的 Scala trait，名为“SafeStringUtils”，并增加以下源代码：

```
trait SafeStringUtils {  
    // Returns a trimmed version of the string wrapped in an Option,  
    // or None if the trimmed string is empty.  
    def trimToNone(s: String): Option[String] = {  
        Option(s) map(_.trim) filterNot(_.isEmpty)  
    }  
}
```

验证这个trait在IDE中能够编译。如果一切正常，完成以下步骤：

- a. 创建这个trait的一个对象版本。
 - b. 创建一个测试类SafeStringUtilsSpec，测试SafeStringUtils.trimToNone()方法。验证它会去除字符串中的空格，并安全地处理null和空串。测试类中应该有3到5个不同的测试。运行测试类，并验证它能成功地执行。
 - c. 增加一个方法，将一个字符串安全地转换为一个整数，即使字符串不可解析也不会抛出错误。编写并执行测试来测试合法和不合法的输入。在这个函数中，最适用的一元集合是什么？
 - d. 增加一个方法，将一个字符串安全地转换为一个long整数，即使字符串不可解析也不会抛出错误。编写并执行测试来测试合法和不合法的输入。在这个函数中，最适用的一元集合是什么？
 - e. 增加一个方法，返回一个给定大小的生成字符串，要求仅包括大写和小写字母。编写并执行测试来验证将返回正确的内容，而且会处理不合法的输入。在这个函数中，有没有适用的一元集合？
3. 编写一个命令行应用，搜索和替换文件中的文本。输入参数是一个搜索模式、正则表达式、替换文本以及一个或多个要搜索的文件。
- a. 首先写一个骨架命令行应用，解析输入参数：搜索模式、替换文本参数和要处理的文件作为一个字符串列表。打印这些输入来验证可以正确地捕获到输入。
 - b. 在命令行用sbt "run-main <object name> <input arguments>"执行这个骨架应用。输入参数放在"run-main"参数所在的同一个双引号内，这样SBT工具就会把它们读作为一个命令。还可以从IDE运行，选择Run→Run… 创建一个运行时配置。利用运行时配置，可以指定一次输入参数，或者也可以在每次执行时显示整个配置。验证搜索模式、替换文本和文件列表能成功地解析。
 - c. 实现这个应用的核心逻辑，读取各个输入文件，搜索并替换所指定的模式，然后把结果打印到控制台。尝试对几个输入文件运行这个应用，验证确实对模式完成了替换。
 - d. 现在把修改后的文本写回到读取这些文本的原文件。下面是使用Java库将字符串写入文件的一个例子：
- ```
import java.io._
val writer = new PrintWriter(new File("out.txt"))
writer.write("Hello, World!\nHere I am!")
writer.close()
```
- e. 要让你的应用使用时更安全，可以在修改输入文件之前先创建这些文件的一个备份。创建备份时，首先把未修改的内容写出到一个文件，在输入文件名

后加**.bak**。创建这个备份文件之前，使用新的`java.io.File(<file name>).exists()`来确保备份文件不会与已有文件重名。可以尝试增加递增的数字（如**.bak1** 和 **.bak2**）来找到唯一的备份文件名。

- f. 创建一个测试类，并编写测试来验证你的应用会按预期工作。你的应用的核心功能应当可以作为方法来调用，而不需要具体启动应用。确保将功能分解为可读而且大小合理的方法，然后为核心方法以及main方法分别编写测试。在这个练习的最后，运行你的测试，验证测试都能成功，然后从命令行针对一个测试文件运行你的应用。
4. 编写一个应用，对一个文件提供总结。它取一个文本文件作为输入，并打印一个完整的总结，包括字符数、单词数和段落数，以及使用最多的20个单词构成的一个列表。

这个应用应该足够聪明，可以过滤非单词字符。解析一个Scala文件时，应当显示（例如）单词，而不显示特殊字符（如"`{`" 或 "`//`"）。它应当还能统计包含实际内容的段落数而不会统计只有空格的段落。

编写测试，使用多行字符串来验证输出。这个应用应当模块化为单独的方法以便于测试。可以编写一个测试，给定字符串"this is is not a test"，会收到一个实例指出单词"is"为使用最多的单词。

为了真正查看你是否已经掌握这一章的内容，一定要在你的解决方案中使用对象、trait和case类。

5. 编写一个应用，对给定 GitHub 项目的最近结束的问题提供一个报告。输入参数应当包括存储库名、项目名和一个可选的要报告的问题数（默认值为10）。输出将有一个报告表头，并显示各个问题的编号、标题、用户名、评论数和标签名。输出应当是良格式的，有固定宽度的列，用竖线（|）分隔，表头用等号分隔（=）。

需要从GitHub API读入这些问题（关于读取一个URL的有关内容，更多信息见第6章“练习”）。解析JSON值，然后打印一个详细格式。下面给出的示例URL可以返回GitHub上官方Scala项目的最近10个结束的问题：

```
https://api.github.com/repos/scala/scala/issues?state=closed&per_page=10
```

我们将使用Json4s库 (<http://json4s.org/>) 将JSON响应解析为一个case类列表。首先，将这个依赖库增加到你的构建脚本，并重新构建这个项目：

```
"org.json4s" %% "json4s-native" % "3.2.10"
```

这可以放在ScalaTest依赖库前面，也可以放在它后面。IntelliJ会发现这个改变，

下载这个库，并重新构建这个项目。如果没有这么做，可以在IntelliJ中打开SBT视图，并刷新项目，或者从命令行运行sbt clean compile。

API的JSON响应相当庞大，不过并不需要解析所有字段。应当设计一个case类，包含JSON中你想要解析的那些字段，使用Option类型表示可能为null或可选的字段。解析JSON响应时，Json4s将只插入case类中定义的字段，而忽略其余的字段。

下面的例子使用Json4s从一个更大的GitHub问题文档解析“labels”数组。通过研究对应一个记录的API输出，应该能设计一系列case类，其中只包含你需要的信息。需要说明，API返回的JSON文档是一个数组，所以可能需要结合List调用extract方法（例如，extract[List[githubIssue]]）：

```
import org.json4s.DefaultFormats ①
import org.json4s.native.JsonMethods ②

val jsonText = """
{
 "labels": [
 {
 "url": "https://api.github.com/repos/scala/scala/labels/tested",
 "name": "tested",
 "color": "d7e102"
 }
]
}
"""

case class Label(url: String, name: String) ③
case class LabelDocument(labels: List[Label]) ④

implicit val formats = DefaultFormats
val labelDoc = JsonMethods.parse(jsonText).extract[LabelDocument] ⑤

val labels = labelDoc.labels
val firstLabel = labels.headOption.map(_.name)
```

- ① 除了支持数字和字符串外，DefaultFormats还支持常用的日期格式。
- ② 我们在JsonMethods中使用“native” JSON解析器来解析JSON文档，并把它们抽取到case类实例。
- ③ 我们把“labels” JSON数组中的一项称为一个“Label”。需要说明，不需要指定“color”字段。
- ④ 整个JSON文档有一个字段“labels”，所以需要一个case类表示这个文档。
- ⑤ Implicit关键字将在第10章讨论。很抱歉还没有介绍就提前使用这个概念，不过你需要这一行来确保Json4s能解析你的JSON文档。

- ⑥ `JsonMethods`会把JSON文档解析为自己的中间格式，然后再用一个给定的`case`类抽取。
- 6. 完成这个练习需要先完成前一个练习。一旦完成GitHub报告应用，下面对它进行重构来提供可重用性和可靠性。
  - a. 首先为GitHub报告编写测试，验证各个组件有正确的行为。如果你的计算机不能连入互联网（没有互联网连接），你能测试这个应用中的多少逻辑？应当能测试大部分逻辑，而不需要实际连接到GitHub网站。
  - b. 重构JSON 处理代码来使用自己的trait，例如“`JsonSupport`”。编写测试，验证它能正确地解析JSON代码，并处理`Json4s`库可能抛出的异常。有必要提供这个trait的对象版本吗？
  - c. 对Web处理代码完成同样的重构。创建你自己的“`HtmlClient`” trait和对象，取一个URL，将其内容作为一个字符串列表返回。除了内容之外，可以在类中包含服务器的状态响应吗？编写测试，验证Web处理代码能防止抛出任何异常。
  - d. 最后，重构报告生成代码中处理固定宽度列的部分，把它重构为一个可重用的trait。它能取一个任意大小的元组并打印其内容吗？有没有一种更合适的数据类型，可以支持可变数目的列，而且知道如何打印字符串而不是`double`值？确保你的报告生成代码要取最大行宽度作为参数。

# 高级类型

到目前为止，你应该已经对Scala语言有了充分的了解。如果读过前面的各章，完成了所有练习，对于如何定义类、编写函数和处理集合应该已经非常了解了。现在你已经掌握了自行构建Scala应用所需了解的所有知识。

不过，如果你还想看其他开发人员的Scala代码，查看Scala API，或者希望了解Scala如何工作，可能还需要好好读一读这一章。这一章中，我们将介绍促使Scala取得成功的众多类型特性。

一个有趣的特性是用常规的类构建高层元组和函数字面量。尽管建立在简单的基础之上，它们却有着精巧的语法。下面将元组和函数字面量创建为类实例，从中可以看到这一点：

```
scala> val t1: (Int, Char) = (1, 'a')
t1: (Int, Char) = (1,a)

scala> val t2: (Int, Char) = Tuple2[Int, Char](1, 'a')
t2: (Int, Char) = (1,a)

scala> val f1: Int=>Int = _ + 2
f1: Int => Int = <function1>

scala> val f2: Int=>Int = new Function1[Int, Int] { def apply(x: Int) = x * 2 }
f2: Int => Int = <function1>
```

另一个有意思的类型特性是隐含类。隐含类（Implicit classes）提供了一种类型安全的方法，可以采用“monkey-patch”方式为现有的类增加新方法和字段，所谓monkey-patch是指在运行时动态修改现有的代码，但不会修改原始代码。通过从原类自动转换到新类，可以在原类上直接调用隐含类中的方法和字段，而不用对类结构做任何修改：

```
scala> object ImplicitClasses {
| implicit class Hello(s: String) { def hello = s"Hello, $s" }
| def test = {
| println("World".hello)
| }
| }
defined object ImplicitClasses

scala> ImplicitClasses.test
Hello, World
```

隐含参数与隐含类的行为类似，将在局部命名空间提供参数，可以为支持隐含参数的方法增加这些参数。如果一个方法定义中部分参数为“隐含”参数，可以由包含局部隐含值的代码来调用，不过用显式参数来调用也是完全可以的：

```
scala> object ImplicitParams {
| def greet(name: String)(implicit greeting: String) = s"$greeting, $name"
| implicit val hi = "Hello"
| def test = {
| println(greet("Developers"))
| }
| }
defined object ImplicitParams

scala> ImplicitParams.test
Hello, Developers
```

最后，我们会深入到类型本身。用于类、trait和函数的类型参数确实相当灵活。可以指定必须满足某个上界（用`<:`）或某个下界（用`>:`）的类型，而不是允许所有类型都可以用作为类型参数：

```
scala> class Base { var i = 10 }; class Sub extends Base
defined class Base
defined class Sub

scala> def increment[B <: Base](b: Base) = { b.i += 1; b }
increment: [B <: Base](b: Base)Base
```

类型参数还可以变换为兼容类型，甚至仅限于一个新实例。如果指定一个类型参数作为协变类型（covariant）（用`+`），它可以变换为一个兼容的基类型。List集合就是协变的，所以子类列表可以转换为一个基类列表：

```
scala> val l: List[Base] = List[Sub]()
l: List[Base] = List()
```

通过了解这些高级类型特性，你会得到另外一些工具来编写更好的Scala代码。你会更充分地理解官方Scala库文档，因为这个库大量使用了高级类型特性。最后一点，它们还会帮助你理解很多Scala特性的基本原理。

下一节中，我们将更深入地介绍以常规类作为基础的元组和函数，以及如何利用它们的方法。

## 元组和函数值类

如果你已经读过这本书的前面各章，应该不需要再向你解释元组和函数值是什么。它们分别在第2章“元组”一节和第5章“函数类型和值”一节中做过介绍。不过，还有一点没有介绍：由一组常规的类来支持它们的特殊语法。

没错，之所以可以有类似(1, 2, true)的元组以及类似 (n: String) => s"Hello, \$n"的函数字面量，特殊之处只是……“配料”。创建这些实例的语法快捷方式很简短，而且很有表述性，但具体实现只是普通的类，你自己也可以写这些类。不过，不要因为这个发现而失望。好消息是，这说明这些高层构造是由安全的、类型参数化的类支持的。

元组实现为TupleX[Y] case类的实例，“X”是一个从1到22的数，这表示其元数（输入参数的个数）。类型参数“Y”可以是单个类型参数，对应Tuple1，或者两个类型参数（对应Tuple2），直到22个类型参数（对应Tuple22）。Tuple1[A]只有一个字段\_1，Tuple2[A,B]有字段\_1和\_2，依此类推。用小括号语法创建元组时（例如，(1, 2, true)），会用这个值实例化一个参数个数相同的元组类。换句话说，元组的表述性语法实际上就是case类的快捷方式，而且这些case类你完全可以自己来写。

TupleX[Y] case类分别用相同的数扩展一个ProductX trait。这些trait提供了一些操作，如productArity可以返回元组的元数，productElement提供了一种非类型安全的方式来访问元组的第n个元素。它们还提供了伴生对象，实现了unapply（见表9-1）来支持元组的模式匹配。

下面来看一个例子，这里不用小括号语法而通过实例化Tuple2 case类来创建元组：

```
scala> val x: (Int, Int) = Tuple2(10, 20)
x: (Int, Int) = (10,20)

scala> println("Does the arity = 2? " + (x.productArity == 2))
Does the arity = 2? true
```

元组case类只是表述性语法的一个数据为中心的实现。函数值类也类似，不过提供了一种逻辑为中心的实现。

函数值实现为FunctionX[Y] trait的实例，根据函数的元数从0到22编号。类型参数“Y”可以是单个类型参数，对应Function0（因为返回值需要一个参数），直到23个

类型参数，对应Function22。不论调用一个现有的函数还是一个新的函数字面量，函数的具体逻辑都在类的apply()方法中实现。

换句话说，写一个函数字面量时，Scala编译器会把它转换为扩展FunctionX的一个新类中的apply()方法体。这种强制机制使得Scala的函数值与JVM兼容，这就限制所有函数都实现为类方法。

下面来看FunctionX类型，首先用我们一直使用的常规语法写一个函数字面量，然后再写为FunctionX.apply()方法的体。这里将创建一个扩展Function1[A,B] trait的匿名类：

```
scala> val hello1 = (n: String) => s"Hello, $n"
hello1: String => String = <function1>

scala> val h1 = hello1("Function Literals")
h1: String = Hello, Function Literals

scala> val hello2 = new Function1[String, String] {
 | def apply(n: String) = s"Hello, $n"
 | }
hello2: String => String = <function1>

scala> val h2 = hello2("Function1 Instances")
h2: String = Hello, Function1 Instances

scala> println(s"hello1 = $hello1, hello2 = $hello2")
hello1 = <function1>, hello2 = <function1>
```

存储在hello1和hello2中的函数值实际上是相等的。Function1类以及所有其他FunctionX类会覆盖toString，名字用小写，并由尖括号包围。因此，打印hello1和hello2时，会得到相同的输出<function1>。这应该并不陌生，因为在这本书中存储函数值的所有代码示例中都应该已经见过。当然，也可能会看到<function2>，这是由Function2的值得到的，依此类推。

Function1 trait包含两个特殊方法，而Function0或任何其他FunctionX trait中没有这两个方法。可以使用这两个方法将两个或多个Function1实例结合为一个新的Function1实例，调用时会按顺序执行所有函数。唯一的限制是，第一个函数的返回类型必须与第二个函数的输入类型一致，依此类推。

方法andThen由两个函数值创建一个新的函数值，执行左边的实例后再执行右边的实例。方法compose的做法相同，只不过顺序相反。

下面用常规的函数字面量来试试看：

```
scala> val doubler = (i: Int) => i*2
doubler: Int => Int = <function1>

scala> val plus3 = (i: Int) => i+3
plus3: Int => Int = <function1>

scala> val prepend = (doubler compose plus3)(1)
prepend: Int = 8

scala> val append = (doubler andThen plus3)(1)
append: Int = 5
```

要了解首类函数如何实现为FunctionX类，这是了解Scala类型模型的重要的第一步。这个语言提供了一个简洁而且很有表述性的语法，由编译器负责支持JVM不具有表述性的运行时模型，同时提供类型安全性来支持更稳定的应用。

## 隐含参数

在第5章“部分应用函数和柯里化”一节中，我们研究了部分应用函数，可以调用一个函数而不必提供它的全部参数。其结果是一个函数值，可以再用未指定的其余参数来调用这个函数，这样就调用了原函数。

如果调用一个函数但不指定所有参数，这个函数真的能执行吗？肯定要在某个位置提供缺少的未指定的参数，以确保函数正常操作。一种方法是为函数定义默认参数，不过这要求函数必须知道缺少的这些参数的正确的值。

另一种方法是使用隐含参数（implicit parameter），调用者在其自己的命名空间提供默认值。函数可以定义一个隐含参数，通常作为与其他非隐含参数相区别的一个单独的参数组。调用者可以指示一个局部值为隐含值，作为隐含参数填入。调用函数时，如果没有为隐含参数指定值，就会使用局部隐含值，将它增加到函数调用中。

可以使用`implicit`关键字标志一个值、变量或函数参数为隐含的。可以用隐含值或变量（如果在当前命名空间中可用）填充一个函数调用中的隐含参数。

下面的例子是一个用隐含参数定义的函数。这个函数定义为一个对象方法，这样它的命名空间就与调用者的命名空间不同：

```
scala> object Doubly {
| def print(num: Double)(implicit fmt: String) = {
| println(fmt format num)
| }
| }
defined object Doubly
```

```
scala> Doubly.print(3.724)
<console>:9: error: could not find implicit value for parameter fmt: String
 Doubly.print(3.724)

scala> Doubly.print(3.724)("%.\n1f")
3.7
```

我们的新print方法有一个隐含参数，所以需要在我们的命名空间中指定一个隐含值/变量，或者显式地增加参数。幸运的是，增加显式参数就能达到目的。

下面增加一个隐含局部值来调用print方法，而不是显式地传入隐含参数：

```
scala> case class USD(amount: Double) {
 | implicit val printFmt = "%.\n2f"
 | def print = Doubly.print(amount)
 |
 }
defined class USD

scala> new USD(81.924).print
81.92
```

隐含值作为Doubly.print方法的第二个参数组，而不需要显式地传入这个参数。

Scala库中大量使用了隐含参数。它们大多用来提供调用者可能要覆盖但是也可能忽略的功能，如集合构建或默认的集合顺序。

如果使用了隐含参数，要记住一点：过度使用隐含参数可能会使你的代码很难读、很难理解。开发人员通常希望知道调用函数时传入了哪些参数。如果发现函数调用包含他们不知道的隐含参数，这可能会让他们有些失落。可以限制只有在支持函数的实现但不会改变期望逻辑或数据的情况下才使用隐含参数，这样就可以避免这种情况。

## 隐含类

Scala还有另一个隐含特性，就是与类的隐含转换，本质上这与隐含参数类似。隐含类(*implicit class*)是一种类类型，可以与另一个类自动转换。通过提供从类型A到类型B的自动转换，类型A的实例就好像是类型B的实例一样，可以有同样的字段和方法。

---

注意：隐含Def呢？

在Scala 2.10之前，隐含转换由*implicit def*方法处理，这个方法取原实例，返回所需类型的一个新实例。隐含方法被隐含类所取代，后者提供了一个更安全、更受限的作用域来转换现有实例。如果希望在你自己的代码中使用*implicit def*，可以查看Scaladocs中的*scala.language.implicitConversions()*方法，来了解如何充分利用这个特性。

---

Scala编译器发现要在一个实例上访问未知的字段或方法时，就会使用隐含转换。它会检查当前命名空间的隐含转换：①取这个实例作为一个参数；②实现缺少的字段或方法。如果它发现一个匹配，就会向隐含类增加一个自动转换，从而支持在这个隐含类型上访问这个字段或方法。当然，如果没有找到匹配，就会得到一个编译错误，在实例上调用未知的字段或方法时通常都会得到这个结果。

下面给出一个隐含类的例子，它向整数值增加一个“fishes”方法。这个隐含类取一个整数，定义了希望为整数增加的“fishes”方法：

```
scala> object IntUtils {
| implicit class Fishies(val x: Int) { ①
| def fishes = "Fish" * x ②
| }
| }
defined object IntUtils

scala> import IntUtils._ ③
import IntUtils._

scala> println(3.fishes) ④
FishFishFish
```

- ① fishes方法在对象中定义，隐含地将整数转换为自身……
- ② ……所以fishes()方法是为所有整数定义的。
- ③ 使用之前，必须将这个隐含类增加到命名空间……
- ④ ……然后就可以对任何整数调用fishes()方法。

隐含类使这种字段和方法移植成为可能，不过对于如何定义和使用隐含类还有一些限制：

1. 隐含类必须在另一个对象、类或trait中定义。幸运的是，对象中定义的隐含类可以很容易地导入到当前命名空间。
2. 它们必须取一个非隐含类参数。在前面的例子中，Int参数可以将一个Int转换为一个Fishies类，从而能访问fishes方法。
3. 隐含类的类名不能与当前命名空间中的另一个对象、类或trait冲突。因此，不能使用case类作为隐含类，因为它有自动生成的伴生对象，这会违反这个规则。

前面的示例都遵循以上的所有规则。它在一个对象中实现（"IntUtils"），取一个参数，其中包含要转换的实例，而且与其他类型不存在名字冲突。尽管可以在对象、类或trait中实现你的隐含类，不过我发现最好还是在对象中实现。对象不可派生，所以

不会从对象自动获得隐含转换。另外，可以很容易地向你的命名空间导入一个对象的一些（或所有）成员来增加这个对象的隐含类。

更准确地讲，除了`scala.Predef`对象中的成员外，通常不会自动获得其他隐含转换。作为Scala库的一部分，`scala.Predef`对象的成员会自动增加到命名空间。它提供了很多类型特性，其中也包括隐含转换，以支持Scala的一些表达性语法。例如箭头操作符`(->)`（见第2章“元组”一节），前面已经用这个操作符从两个值生成大小为2的元组。

下面是支持箭头操作符的隐含类的一个简化版本：

```
implicit class ArrowAssoc[A](x: A) {
 def ->[B](y: B) = Tuple2(x, y)
}
```

作为一个例子，取表达式`1 -> "a"`，这会生成包含一个整数和一个字符串的元组。实际上，这里完成了一个隐含转换，将整数转换为`ArrowAssoc`的一个实例，然后再调用“`->`”方法，最终会返回一个新的`Tuple2`。不过，由于隐含转换会…隐含地…增加到命名空间，所以这个表达式不大于用箭头分隔的两个值。

隐含类是向现有的类增加方法的一种很好的做法。如果精心使用，会帮助你提高代码的表达性。不过，要注意：不能影响可读性。你肯定不希望没有见过`Fishies`隐含类的开发人员困惑，“`fishes`方法到底是什么，它在哪里实现？”

## 类型

这一章已经用了好几节来介绍与类型有关的特性，如隐含转换和函数类。这一节中，我们将从与类型有关的特性转向类型本身的核心内容。

类（`class`）是一个可以包含数据和方法的实体，有一个特定的定义。类型（`type`）是一个类规范，与符合其需求的一个类或一组类匹配。例如，`Option`类是一个类型，`Option[Int]`也是一个类型。类型可以是一个关系，指定“类A或其任何子孙类”，或者“类B或其任何父类”。类型还可以更为抽象，指定“定义这个方法的任何类”。

`trait`也是如此，它们是可以包含数据和方法的实体，而且有特定的定义。类型是类规范，但也同样适用于`trait`。不过，对象并不认为是类型。对象是单例，尽管它们可以扩展类型，但是它们本身不是类型。

介绍Scala类型概念时所用的例子都与这一节研究的绝妙特性有关。这些特性可以帮助

你编写更严格、更安全、更稳定，而且文档更全面的代码，这正是建立强类型系统的关键。

首先来看如何定义你自己的类型而不用创建类。

## 类型别名

类型别名（type alias）会为一个特定的现有类型（或类）创建一个新的命名类型。编译器处理这个新类型别名时，就好像它在一个常规类中定义一样。可以从类型别名创建一个新实例，用它取代类型参数的类，也可以在值、变量和函数返回类型中指定类型别名。如果有别名的类包含类型参数，可以把这些类型参数增加到类型别名，也可以固定为特定的类型。

不过，与隐含转换类似，类型别名只能在对象、类或trait中定义。它们只适用于类型，所以对象不能用来创建类型别名。

可以使用type关键字定义一个新的类型别名。

### 语法：定义类型别名

```
type <identifier>[type parameters] = <type name>[type parameters]
```

好的，既然在介绍类型，下面就来创建一些类型！

```
scala> object TypeFun {
| type Whole = Int
| val x: Whole = 5
|
| type UserInfo = Tuple2[Int, String]
| val u: UserInfo = new UserInfo(123, "George")
|
| type T3[A,B,C] = Tuple3[A,B,C]
| val things = new T3(1, 'a', true)
| }
defined object TypeFun

scala> val x = TypeFun.x
x: TypeFun.Whole = 5

scala> val u = TypeFun.u
u: TypeFun.UserInfo = (123,George)

scala> val things = TypeFun.things
things: (Int, Char, Boolean) = (1,a,true)
```

在这个例子中，现在类型Whole是抽象类Int的一个别名。另外，类型UserInfo是一个

元组的别名，这个元组中，第一个位置上是一个整数，第二个位置上是一个字符串。由于Tuple2是一个可以实例化的case类，所以可以从类型别名UserInfo直接实例化这个类。最后的T3类型没有固定类型参数，所以可以用任何类型实例化。

类型别名是一种很好的办法，可以用局部特定的名字指示现有类型。类中可能经常使用Tuple2[Int, String]，如果将它命名为UserInfo可能更有用。不过，与其他高级类型特性一样，类型别名不能取代严谨的面向对象设计。如果一个真正的类名为UserInfo，这会更稳定，而且长期来看比使用类型别名更直观。

## 抽象类型

尽管类型别名解析为一个类，但抽象类型（abstract types）是规范，可以解析为0、1个或多个类。它们的做法与类型别名类似，不过作为规范，它们是抽象的，不能用来创建实例。抽象类型常用于类型参数，来指定一组可以接受（可以传入）的类型。抽象类型还可以用来创建抽象类中的类型声明（type declarations），即声明具体（非抽象）子类必须实现的类型。

对于后一种情况，来看一个例子：trait可以包含一个未指定类型的类型别名。这个类型声明可以在方法签名中重用，而且必须填入子类。

下面来创建这样一个trait：

```
scala> class User(val name: String)
defined class User

scala> trait Factory { type A; def create: A }
defined trait Factory

scala> trait UserFactory extends Factory {
 | type A = User
 | def create = new User("")
 |
}
defined trait UserFactory
```

Factory中的抽象类型A用作为create方法的返回类型。在具体子类中，这个类型由类型别名重定义为一个特定的类（User）。

还可以采用另一种方法写trait和类：使用类型参数。下面的例子就使用类型参数来实现前面的trait和类：

```
scala> trait Factory[A] { def create: A }
defined trait Factory

scala> trait UserFactory extends Factory[User] { def create = new User("") }
```

```
defined trait UserFactory
```

设计泛型类时，可以用抽象类型替代类型参数。如果想要有一个参数化类型，类型参数就很合适。否则，抽象类型可能更适用。UserFactory示例类可以使用参数化类型，也可以定义自己的类型别名，这两种做法都是适用的。

在这个例子中，对于Factory trait子类所允许的类型没有任何限制。不过，如果能够为类型指定界限（bounds），通常会更有用，这是一个上界或下界，确保所有类型实现都要满足某个标准。

## 定界类型

定界类型限制为只能是一个特定的类或者它的子类型或基类型。上界（upper bound）限制一个类型只能是该类型或它的某个子类型。另一种说法是，上界定义了一个类型必须是什么，而且由于支持多态，可以接受子类型。下界（lower bound）则限制一个类型只能是该类型或它扩展的某个基类型。

可以使用上界关系操作符(<:) 为类型指定一个上界。

### 语法：上界界定类型

```
<identifier> <: <upper bound type>
```

尝试使用定界类型之前，下面先定义几个类来完成测试：

```
scala> class BaseUser(val name: String)
defined class BaseUser
```

```
scala> class Admin(name: String, val level: String) extends BaseUser(name)
defined class Admin
```

```
scala> class Customer(name: String) extends BaseUser(name)
defined class Customer
```

```
scala> class PreferredCustomer(name: String) extends Customer(name)
defined class PreferredCustomer
```

下面定义一个函数，它有一个有上界的参数：

```
scala> def check[A <: BaseUser](u: A) { if (u.name.isEmpty) println("Fail!") }
check: [A <: BaseUser](u: A)Unit
```

```
scala> check(new Customer("Fred"))
```

```
scala> check(new Admin("", "strict"))
Fail!
```

类型参数A限制为等于或扩展了BaseUser类型的类型。这样一来，参数u就可以访问"name"字段。如果没有上界限制，访问未知类型的"name"字段会导致一个编译错误。u参数的具体类型是保留的，所以这个check函数的将来版本可以安全地返回正确类型的结果。

上界操作符还有一个不太严格的形式：可以使用视界（view-bound）操作符(<%）。上界要求是一个类型（与子类型兼容），视界还支持可以处理为该类型的任何类型。因此，视界接受隐含转换，尽管一个类型不是所请求的类型，但是可以转换为所请求的类型，那么也能满足视界要求。上界更限定，因为类型需求不包括隐含转换。

与上界对应的是下界，它指定了可以接受的最小的类。使用下界关系操作符(>)会指定一个类型的下界：

### 语法：下界界定类型

```
<identifier> >: <lower bound type>
```

下面来创建一个函数，返回不低于Customer类型（不过具体实现可以更低）。

```
scala> def recruit[A >: Customer](u: Customer): A = u match {
| case p: PreferredCustomer => new PreferredCustomer(u.name)
| case c: Customer => new Customer(u.name)
| }
recruit: [A >: Customer](u: Customer)A

scala> val customer = recruit(new Customer("Fred"))
customer: Customer = Customer@4746fb8c

scala> val preferred = recruit(new PreferredCustomer("George"))
preferred: Customer = PreferredCustomer@4cd8db31
```

尽管返回了一个新的PreferredCustomer实例，但preferred值的类型由返回类型设置，这可以保证不低于Customer。

界定类型还可以用来声明抽象类型。下面是一个例子，这里的抽象类声明了一个抽象类型，并在一个声明方法中使用这个类型。具体（非抽象）子类再将这个类型声明实现为一个类型别名，并在定义方法中使用这个类型别名。结果就是这个类的实现将实现这个方法，但是确保只使用兼容的类型：

```
scala> abstract class Card {
| type UserType <: BaseUser
| def verify(u: UserType): Boolean
|
| }
defined class Card
```

```
scala> class SecurityCard extends Card {
| type UserType = Admin
| def verify(u: Admin) = true
| }
defined class SecurityCard

scala> val v1 = new SecurityCard().verify(new Admin("George", "high"))
v1: Boolean = true

scala> class GiftCard extends Card {
| type UserType = Customer
| def verify(u: Customer) = true
| }
defined class GiftCard

scala> val v2 = new GiftCard().verify(new Customer("Fred"))
v2: Boolean = true
```

与非界定类型一样，究竟使用在基类中定义的抽象类型还是使用类型参数，这并没有明确的答案。很多开发人员倾向于使用类型参数来实现更有表述性的语法。不过，使用界定类型通常优于非界定类型。它们不仅可以限制不能在子类中使用不合法的类型，而且可以作为一种自描述文档。通过界定类型，可以清楚地看出一组类希望使用哪些类型。

## 类型变化

增加上界或下界可以使类型参数更为限定，增加类型变化则相反，会减少类型参数的限制。类型变化（Type variance）指定一个类型参数如何调整以满足一个基类型或子类型。

默认地，类型参数是不变的（invariant）。对于一个类型参数化的类，它的实例只与该类以及参数化类型兼容。这个实例不能存储在类型参数为基类型的值中。

这通常会让开发人员很奇怪，他们已经熟悉了Scala对多态的支持。利用多态，一个给定类型的值可以用作为它的某个基类型。例如，可以把某个类型的实例赋给其基类型的一个值。

下面是Scala多态的一个例子，允许把较低的类型存储在有更高类型的值中。这一章后面的例子中都将使用这个车辆类层次体系（包括两部分）：

```
scala> class Car { override def toString = "Car()" }
defined class Car

scala> class Volvo extends Car { override def toString = "Volvo()" }
defined class Volvo
```

```
scala> val c: Car = new Volvo()
c: Car = Volvo()
```

不过，这种多态调整对于类型参数并不适用：

```
scala> case class Item[A](a: A) { def get: A = a }
defined class Item

scala> val c: Item[Car] = new Item[Volvo](new Volvo)
<console>:12: error: type mismatch;
 found : Item[Volvo]
 required: Item[Car]
Note: Volvo <: Car, but class Item is invariant in type A.
You may wish to define A as +A instead. (SLS 4.5)
 val c: Item[Car] = new Item[Volvo](new Volvo)
```

尽管可以把一个Volvo实例赋至类型为Car的一个值，但是不能将Item[Volvo]实例赋给类型为Item[Car]的值。类型参数默认是不变的，不能替换为兼容的类型（尽管它们是兼容的）。

为了修正这个问题，需要把类型参数放在Item covariant中。协变类型参数（Covariant type parameters）可以自动在必要时调整为其基类型。通过在类型参数前面加一个加号（+），可以标志一个类型参数为协变类型参数。

下面重新用协变类型参数来定义Item类，使得Item[Volvo]类型可以变为Item[Car]：

```
scala> case class Item[+A](a: A) { def get: A = a }
defined class Item

scala> val c: Item[Car] = new Item[Volvo](new Volvo)
c: Item[Car] = Item(Volvo())

scala> val auto = c.get
auto: Car = Volvo()
```

类型参数"A"现在是协变类型参数，可以从一个子类型调整为基类型。换句话说，现在Item[Volvo]的实例可以赋至类型为Item[Car]的一个值。

Item.get()方法也采用类似的方式支持类型参数的协变。尽管实例为Item[Volvo]，实际上包含一个Volvo，但是值的类型是Item[Car]，所以c.get的返回类型为Car。

协变对于将类型参数调整为其基类型是一个很好的工具，不过并不一定可用。例如，方法的输入参数不能是协变的，这与基类型不能转换为子类型的原因相同。

如果输入参数是协变的，这意味着它会绑定到一个子类型，但是可以用一个基类型调用。这种转换是不可能的，因为基类型不能转换为子类型。

下面来看如果试图使用一个协变的类型参数作为方法的输入参数类型，Scala编译器会给出什么结果：

```
scala> class Check[+A] { def check(a: A) = {} }
<console>:7: error: covariant type A occurs in contravariant position in
 type A of value a
 class Check[+A] { def check(a: A) = {} }
```

如 Scala 编译器给出的错误所示，方法参数中使用的类型参数是逆变的（contravariant），而不是协变（covariant）。逆变（Contravariance）是指一个类型参数可以调整为一个子类型，与子类型到基类型的多态转换方向相反。

逆变类型参数要在类型参数前标志一个减号（-）。它们可以用于方法的输入参数，但是不能用作为方法的返回类型。返回类型是协变的，因为其结果可以是一个子类型（能够多态转换为一个基类型）。

下面重新用一个逆变类型参数定义这个例子，现在它应该能正常编译：

```
scala> class Check[-A] { def check(a: A) = {} }
defined class Check
```

或者，还可以保持类型参数为不可变。调用check()方法时，输入参数必须与类的类型参数有完全相同的类型：

```
scala> class Check[A] { def check(a: A) = {} }
defined class Check
```

以上展示了如何解决“逆变参数位置上有一个协变参数”的错误，不过我们还需要一个更好的例子展示逆变和协变。下面通过一个更全面的例子来了解如何定义协变和逆变类型参数。

这个例子包括两部分，第一部分将定义要使用的类和方法。我们将使用Car类、它的子类Volvo以及Volvo的一个新子类，名为VolvoWagon。利用这个3级类层次体系，我们可以选择一个中间类Volvo，试着把它替换为其子类或基类。然后使用Item测试协变，使用Check测试逆变。最后我们将定义一些方法对中间类Volvo使用Item和Check。这样我们就能用它的子类和基类完成试验，来了解具体是如何做的：

```
scala> class Car; class Volvo extends Car; class VolvoWagon extends Volvo
defined class Car
defined class Volvo
defined class VolvoWagon

scala> class Item[+A](a: A) { def get: A = a }
defined class Item
```

```
scala> class Check[-A] { def check(a: A) = {} }
defined class Check

scala> def item(v: Item[Volvo]) { val c: Car = v.get }
item: (v: Item[Volvo])Unit

scala> def check(v: Check[Volvo]) { v.check(new VolvoWagon()) }
check: (v: Check[Volvo])Unit
```

Item类显然需要一个协变类型参数。限定类型为Volvo时，其get()方法会返回一个Volvo，应该可以将这个实例存储在类型为Car的一个值中。这遵循多态的标准规则，即基类值中可以存储其子类的一个实例。

类似地，Check类显然需要一个逆变类型参数。限定类型为Volvo时，其check()方法取一个Volvo，所以应该可以传入一个VolvoWagon实例。这也遵循多态的标准规则，即可以将一个子类的实例传入需要其基类的方法中。

在第二部分中，我们将对基类、中间类和子类调用这些方法：

```
scala> item(new Item[Car](new Car())) ①
<console>:14: error: type mismatch;
 found : Item[Car]
 required: Item[Volvo]
 item(new Item[Car](new Car()))
 ^

scala> item(new Item[Volvo](new Volvo))

scala> item(new Item[VolvoWagon](new VolvoWagon())) ②

scala> check(new Check[Car]()) ③

scala> check(new Check[Volvo]())

scala> check(new Check[VolvoWagon]()) ④
<console>:14: error: type mismatch;
 found : Check[VolvoWagon]
 required: Check[Volvo]
 check(new Check[VolvoWagon]())
```

- ① Item类有一个协变类型参数，可以从一个子类调整为基类，但是反过来不行。
- ② 这里可以看到协变的具体工作，Item[VolvoWagon]变成了Item[Volvo]。
- ③ 这里完成的是逆变，Check[Car]变成了Check[Volvo]。
- ④ 不过反过来不行，因为逆变类型参数不能从一个基类变成子类。

协变和逆变可以减少类型参数的限制，不过对于如何使用协变和逆变也存在自己的限

制。如果不确定究竟如何使用，可以保持类型参数为不变的。这是类型参数的默认状态，你会发现这样做会更安全，通常情况下都应当保持所有类型参数为不可变，除非确实需要改变。

## 包对象

这一章我们讨论的大多数高级类型（如隐含参数、隐含转换和类型别名）只能在其他类型中定义。这个限制可以帮助保护这些实体，确保大多数情况下只能通过显式导入增加到命名空间。

不过`scala.Predef`对象是一个例外，这个对象的内容会自动增加到Scala的命名空间。另一个例外是包对象，这是对应各个包的唯一对象，会自动导入到该包代码的命名空间中。

包对象在自己单独的文件中定义，即`package.scala`，这个文件会放在将受其影响的包中。可以在对象定义前面增加`package`关键字来定义包对象。

下面是一个包对象的例子，这里定义了一个新的类型别名`Mappy`：

```
// located on com/oreilly/package.scala
package object oreilly {
 type Mappy[A,B] = collection.mutable.HashMap[A,B]
}
```

这个包中定义的任何类、`trait`或对象都会获得`Mappy[A,B]`类型别名，而且能直接使用这个类型别名。

Scala库中的核心“`scala`”包包括一个类似这样的包对象，可以为命名空间增加很多常用的不可变集合（不过没有像“`Mappy`”这样有趣的名字）。

包对象是定义类型别名、隐含转换和其他高级类型的一个很好的解决方案。包对象可以扩展这些特性的范围，不要求手动导入才能使用一个类、`trait`或对象。

## 小结

Scala结合了函数式编程和面向对象编程范式，支持首类函数以及类定义。现在我们可以知道，首类函数就是类定义。

Scala中的类型特性可以使类和方法更安全，更受限。通过对类型参数指定可接受的界限，代码可以声明相应的需求并确保类型安全。

在保证同样的类型安全性的同时，也可以利用类型特性减少类和方法的限制。协变和逆变类型参数可以在接受和返回兼容类型方面提供类型灵活性。另外与使用固定方法和显式参数的代码相比，隐含类和参数可以放宽限制，同时避免意料之外的类型冲突。

现在你应该能理解所有Scala代码了。接下来最好仔细查看Scala API，因为其中大量使用了变化注解（variance annotation）和隐含参数，这些内容对你来说应该不难理解了。还可以更进一步，可以阅读Scala库本身的源代码。建议从我们已经很熟悉的集合开始，如Option和Future。

除了完成这一章的练习外，你还可以了解一些很棒的Scala开源库。我们只介绍了SBT构建系统的一小部分，不过要知道，它不只是能构建基本的应用。Apache Spark (<https://spark.apache.org/>) 是一个很流行的工具，它使用Scala完成数据分析和其他计算。Typesafe (<https://typesafe.com/>) 公司管理着Scala代码基，还提供了Play (<https://www.playframework.com/>) Web框架和Akka分布式计算框架 (<http://akka.io/>)。Spray (<http://spray.io/>) 和Finagle (<http://bit.ly/ls-finagle>) 库很适合构建网络应用，不过如果你只需要一个REST API，Scalatra (<http://www.scalatra.org/>) 框架可能对你更适合。

最后，如果你确实很喜欢这一节介绍的关于Scala类型系统的内容，希望对更多类Haskell的类型安全性特性有所了解，可以查看Scalaz (<http://bit.ly/ls-scalaz>) 库。这个库读作“Scala-Zed”，与这本书中的代码相比，这个库可以帮助你编写更安全、更有表述性的代码。通过学习Scalaz库以及Typelevel小组提供的其他项目，还将帮助你成为一个更优秀的开发人员。

## 思考题

尽管这一章很重要，需要认真阅读和理解，不过这里介绍的技术层次很高，除非要用Scala编写你自己的库或高级应用，否则通常可能用不到这些技术。

在这一章中，我们不再像前面各章那样专用有一节给出标准练习。如果你已经完成了之前的所有练习，现在应该已经熟悉如何在REPL中开发简洁的类和函数，以及如何在一个IDE中开发更大规模的应用。实际上，我会问一些有关本章所介绍的高级类型特性和功能的问题。

你可能想要在REPL或IDE中做一些试验来找出这些问题的答案。可以把这些问题看作

是需要完成的一个全面的试验，通过这个试验，可以使你对这个语言的使用有更多体验。

1. 如何扩展一个函数？一个扩展Function1[A,B]的类或trait有哪些应用？如果你在写这样一个类或trait，你会扩展Function1[A,B]还是扩展A => B？
2. 如果一个函数有两个参数表，这两个参数表中分别包含一个整数，而且函数将返回一个整数，这个函数的函数类型是什么？如果把它写为一个FunctionX类，具体的类和类型参数包含什么？
3. 隐含参数的一个流行用法是用作为默认设置，在大多数情况下都可以使用这个默认设置，不过在特殊情况下可以被覆盖。假设你在写一个排序函数，它取一些文本行，这些文本行可能以一个右对齐的数字开头。如果想要根据这些数字排序（数字前面可能有空格作为前缀），如何把这个功能写在一个隐含参数中？如果允许用户覆盖这个行为，在排序中忽略这些数字，又该如何做到？
4. 假设你写了自己的Option[A]，命名为Perhaps[A]，并实现了一或两个方法来访问它的内容。需要提供哪种隐含转换才允许将它处理为一个集合？如何实例上调用flatMap和filter而不需要实现这些方法？
5. 如果要实现你自己的字符串类Characters，支持所有JVM的java.lang.String方法，另外还可以看作是一个Scala集合，要如何实现？结合类型和转换就能完成大部分工作吗？建议仔细研究scala.Predef的源代码来找出提示。
6. 要为所有元组增加一个"sum"方法，它返回元组中所有数值的总和，如果实现这个方法？例如，('a', "hi", 2.5, 1, true).sum应当返回3.5。
7. Function1类型接受类型参数，一个对应输入值，另一个对应输出值。哪一个应当是协变参数？哪一个则应当是逆变参数？



# 保留字

表A-1展示了Scala中的保留字。保留字是Scala语言定义的一部分，不能用作为标识符。为了保证这些定义简洁，我使用“类”表示“类、对象和trait”（尽管后者可能更为准确）。

表A-1：Scala的保留字

| 名        | 描述                                                                       |
|----------|--------------------------------------------------------------------------|
| -        | 通配操作符，表示一个期望的值                                                           |
| :        | 将一个值、变量或函数与其类型分开                                                         |
| @        | 定义一个类或其成员的一个注解。注解是一个JVM特性，不过在Scala中很少使用，@annotation.tailrec是一个例外，这个注解很常用 |
| #        | 这是一个类型投影，将类型与其子类型分开                                                      |
| <-       | 在for循环中将生成器与其标识符分开                                                       |
| ←        | <-的一个单字符替代(\u2190)                                                       |
| <:       | 上界操作符，限制类型为等于或扩展了给定类型的类型                                                 |
| <%       | 视界操作符，接受可以看作是给定类型的所有类型                                                   |
| =        | 赋值操作符                                                                    |
| =>       | 在匹配表达式和部分函数中用来指示一个条件表达式，在函数类型中指示一个返回类型，在函数字面量中用来定义函数体                    |
| =>       | =>的单字符替换(\u21D2)                                                         |
| >:       | 下界操作符，限制类型为等于或被给定类型扩展的类型                                                 |
| abstract | 标志一个类或trait为抽象而且不可实例化                                                    |
| case     | 在匹配表达式和部分函数中定义一个匹配模式                                                     |
| catch    | 捕获一个异常。这是util.Try一元集合之前的一种替代语法                                           |

表A-1：Scala的保留字（续）

| 名         | 描述                                                                                                                                                                                                                                                                                                                                         |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| class     | 定义一个新类                                                                                                                                                                                                                                                                                                                                     |
| def       | 定义一个新方法                                                                                                                                                                                                                                                                                                                                    |
| do        | do..while循环定义的一部分                                                                                                                                                                                                                                                                                                                          |
| else      | if...else条件表达式的第二部分                                                                                                                                                                                                                                                                                                                        |
| extends   | 为一个类定义基类型                                                                                                                                                                                                                                                                                                                                  |
| false     | 两个布尔值之一                                                                                                                                                                                                                                                                                                                                    |
| final     | 标志一个类或trait不可扩展                                                                                                                                                                                                                                                                                                                            |
| finally   | 在一个try块后执行一个表达式。这是util.Try一元集合之前的替代语法                                                                                                                                                                                                                                                                                                      |
| for       | 开始一个for循环                                                                                                                                                                                                                                                                                                                                  |
| forSome   | 定义一个存在类型。存在类型（Existential types）是用来指定类型需求的一种灵活的方法，但是在一般的Scala开发中并不鼓励这样做。参见 SIP-18 (Scala Improvement Process #18) ( <a href="http://docs.scala-lang.org/sips/completed/modularizing-language-features.html">http://docs.scala-lang.org/sips/completed/modularizing-language-features.html</a> )，其中详细介绍了为什么存在类型被认为是Scala中“选择性加入”（opt-in）的特性 |
| if        | if...else条件表达式的第一部分，或者if条件语句的主体部分                                                                                                                                                                                                                                                                                                          |
| implicit  | 定义了一个隐含转换或参数                                                                                                                                                                                                                                                                                                                               |
| import    | 将一个包、类或类的成员导入到当前命名空间                                                                                                                                                                                                                                                                                                                       |
| lazy      | 定义一个值为懒值，只在第一次访问时才定义                                                                                                                                                                                                                                                                                                                       |
| match     | 开始一个匹配表达式                                                                                                                                                                                                                                                                                                                                  |
| new       | 创建一个类的新实例                                                                                                                                                                                                                                                                                                                                  |
| null      | 这个值指示没有实例。类型为Null                                                                                                                                                                                                                                                                                                                          |
| object    | 定义一个新对象                                                                                                                                                                                                                                                                                                                                    |
| override  | 标志一个值或方法要取代基类型中同名的成员                                                                                                                                                                                                                                                                                                                       |
| package   | 定义当前包，一个递增的包名，或者一个包对象                                                                                                                                                                                                                                                                                                                      |
| private   | 标志一个类成员不能在类定义之外访问                                                                                                                                                                                                                                                                                                                          |
| protected | 标志一个类成员不能在类定义或其子类之外访问                                                                                                                                                                                                                                                                                                                      |
| return    | 显式指出一个方法的返回值。默认地，方法中的最后一个表达式会用作为返回值                                                                                                                                                                                                                                                                                                        |
| sealed    | 标志一个类只接受当前文件中的子类                                                                                                                                                                                                                                                                                                                           |
| super     | 标志要引用基类型中的成员，而不是当前类中被覆盖的那个成员                                                                                                                                                                                                                                                                                                               |
| this      | 标志要引用当前类中的成员，而不是同名的一个参数                                                                                                                                                                                                                                                                                                                    |
| throw     | 产生一个错误条件，中断当前操作流，只有当这个错误在其他位置“被捕获”时才继续操作流                                                                                                                                                                                                                                                                                                  |
| trait     | 定义一个新trait                                                                                                                                                                                                                                                                                                                                 |
| true      | 两个布尔值之一                                                                                                                                                                                                                                                                                                                                    |

表A-1：Scala的保留字（续）

| 名     | 描述                                |
|-------|-----------------------------------|
| try   | 标志一段捕获异常的代码。这是util.Try一元集合之前的替代语法 |
| type  | 定义一个新的类型别名                        |
| val   | 定义一个新的不可变的值                       |
| var   | 定义一个新的可变的变量                       |
| while | do..while循环定义的一部分                 |
| with  | 为一个类定义基trait                      |
| yield | 从一个for循环得出返回值                     |

---

注意：我喜欢的::和++操作符呢？

:: 和++ 操作符是合法的方法标识符，而不是保留字。Scala集合库用这些标识符定义方法，这说明你也可以在你自己的方法中使用这些标识符。

---



## 作者介绍

---

**Jason Swartz**是一名软件工程师，任职于美国旧金山湾区，他开发了Loyal3和Netflix的Scala应用。在转向函数式编程之前，他曾在eBay管理开发者文档以及支持小组，曾用Java编写过推广和商业化平台，另外曾在Apple构建工具和UI原型。

## 封面介绍

---

本书封面上的动物是美洲鸵鸟或美洲大鸵鸟（*Rhea americana*），这是一种个头很高、不会飞的鸟类，常见于南美东部。在当地被称作是ñandú，这是构成美洲鸵鸟种属的两种鸟类之一，另一种是比较小但更不常见的美洲小鸵鸟（*rhea pennata*）。

美洲大鸵鸟是阿根廷、玻利维亚、巴西、巴拉圭和乌拉圭特有的一种动物，主要生活在有高大植被的开放区域，如草地、树木稀少的草原和草场湿地。奇怪的是，在德国西北部乡村也生活着为数不多的美洲大鸵鸟，这些鸵鸟是2000年从一个农场逃离的。出乎人们意料的是，这些鸵鸟居然在野外环境中安定地繁衍生息下来。

尽管与非洲鸵鸟很类似，美洲大鸵鸟有3个脚趾，而不是2个，另外体型是非洲鸵鸟的一半。成年美洲大鸵鸟约5英尺高，重约44~60磅。它有灰色的蓬松翅膀，鸟冠、脖颈和背的上部分布着深色斑点。这种鸟有强有力的长腿，可以跑得很快，时速能超过每小时35英里。它的大翅膀可以帮助在奔跑时保持平衡，另外在求偶时也会炫耀它的大翅膀尽力展示自己。

美洲鸵鸟通常在水域附近筑巢。美洲大鸵鸟是一夫多妻制，所以每只雄性鸵鸟在繁殖季节（春季和夏季）可能与最多12只雌性鸵鸟交配。在这个时期，雄性鸵鸟有极强的领土占有欲，相互之间常有攻击行为。交配之后，雌性鸵鸟会在巢中产蛋，然后通常会再寻找一个新的配偶。雄性鸵鸟独自孵化它的所有配偶的蛋，通常都在同一个巢中。之后，它会很用心地保护和抚养小鸵鸟。

除了繁殖方式外，美洲大鸵鸟是群居的，会有10到100只鸵鸟构成一个鸵鸟群，甚至还会混杂其他大型动物，如鹿。美洲大鸵鸟喜欢吃植物、种子和水果，不过也会吃昆虫、小型鼠类、爬行动物和小鸟。它还会吞食小沙石来帮助消化。

O'Reilly书封面上的很多动物都是濒危动物，所有这些动物对这个世界都很重要。要了解如何帮助这些动物，可以访问[animals.oreilly.com](http://animals.oreilly.com)。

# Scala学习手册

为什么学习Scala? 你无需成为数据科学家或分布式计算专家, 也能掌握这种面向对象函数式编程语言。这是一本很实用的书, 它以通俗易懂的方式对Scala语言做了全面的介绍, 还给出了大量语法图、示例和练习来帮助你理解书中的内容。首先你将了解Scala的核心类型和语法, 然后会深入学习高阶函数和不可变的数据结构。

本书作者在书中介绍了Scala简洁而且很有表述性的语法, 解释了它的类型安全性和性能可以确保稳定性, 另外说明了Scala可以快速运行, 适用所有应用。基于这些特点, 作者展示了为什么Scala会成为想要提高水平的Ruby或Python开发人员的理想语言。

“不论你是有Python经验, 还是更熟悉Java, 你都会发现这本书对Scala的介绍非常适用。Jason的文字很实用, 而且通俗易懂。本书提供了一个简洁明了的初学者指南, 涵盖了我们熟悉的面向对象风格和这个语言的常用特性。我开始学Scala时就希望有这样一本！”

—Katherine Fellows  
Comcast公司软件工程师

- 学习核心数据类型、字面量、值和变量。
- 了解如何使用表达式（Scala语法的基础）思考以及编写代码。
- 编写接收或返回其他函数的高阶函数。
- 熟悉不可变数据结构, 利用类型安全的描述性操作轻松地完成变换。
- 创建定制的中缀操作符简化现有操作, 甚至可以针对你自己的领域编写特定的语言。
- 构建包括一个或多个trait的类来提供充分的可重用性, 或者在实例化时混合使用trait创建新功能。

---

**Jason Swartz**, 是一位软件工程师, 热衷于直观的用户界面、表述性编程语言和简洁的用户文档。他还组织了旧金山的很多Scala社区活动, 并为Netflix的客户设备程序开发过应用。

---

SCALA/JAVA/PROGRAMMING LANGUAGES

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行  
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-5123-8774-4



9 787512 387744 >

定价：48.00元

[General Information]

书名=SCALA学习手册

作者=JASON SEARTZ著

页数=222

SS号=13950109

DX号=

出版日期=2016.02

出版社=北京中国电力出版社