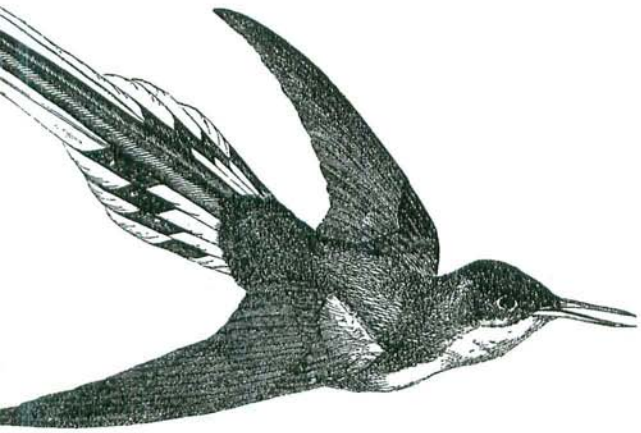
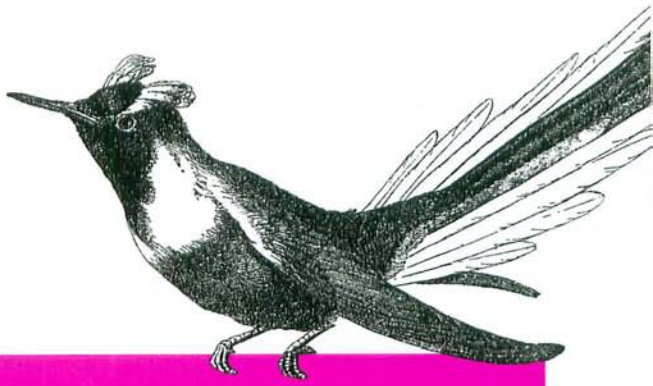
The Ruby Programming Language

Everything You Need to Know

涵盖 Ruby 1.8和1.9





Ruby编程语言

David Flanagan & Yukihiro Matsumoto 著

why the lucky stiff 配图

廖志刚 张禾 译

O'REILLY®

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

O'REILLY®

Ruby 编程语言

The Ruby Programming Language

David Flanagan
Yukihiro Matsumoto 著

廖志刚 张 禾 译

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

内容简介

本书详细介绍了 Ruby 1.8 和 1.9 版本各方面的内容。在对 Ruby 进行了简要的综述之后，本书详细介绍了以下内容：Ruby 的句法和语法结构，数据结构和对象，表达式和操作符，语句和控制结构，方法、proc、lambda 和闭包，反射和元编程，Ruby 平台。

本书还包含对 Ruby 平台上丰富的 API 的详尽介绍，并用带有详尽注释的代码演示了 Ruby 进行文本处理、数字运算、集合、输入/输出、网络开发和并发编程的功能。

版权贸易合同登记号　图字：01-2007-3440

图书在版编目（CIP）数据

Ruby 编程语言 /（美）弗拉纳根（Flanagan,D.），（美）松本行弘（Matsumoto,Y.）著；廖志刚，张禾译.
—北京：电子工业出版社，2009.1
ISBN 978-7-121-07701-2

I. R… II.①弗…②松…③廖…④张… III.计算机网络－程序设计 IV.TP393.09

中国版本图书馆 CIP 数据核字（2008）第 173456 号

责任编辑：陈元玉
项目管理：梁　晶
封面设计：Karen Montgomery　张　健
印　　刷：北京市天竺颖华印刷厂
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：28.25　字数：500 千字
印　　次：2009 年 1 月第 1 次印刷
定　　价：68.00 元

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

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

服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet 门户和商业网站），再到WebSite(第一个桌面 PC 的Web服务器软件)，O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

译者序

Ruby 在 Rails 的大红大紫之后，已经逐渐成为主流脚本语言中重要的一员。今天，Ruby 已经在各种平台上开花结果，除了老牌的 CRuby，JRuby、IronRuby 和 XRuby 也蓬勃发展起来，各种 IDE 对 Ruby 的支持也日渐成熟。如今的 Ruby，已不再是默默无闻的青涩少年，而俨然成为编程语言中的新贵了。

Ruby 方面的图书也日渐大卖，各种 Ruby 及 Rails 方面的图书如雨后春笋般涌现。稍微唤醒一下不太尘封的记忆，就能想起那张著名的图片，两本 Ruby 及 Rails 的镐头书跟一大摞 Java 图书对照性地摆放在一起。图片的目的在于说明 Ruby 的易学易用，根本不用像学习 Java 一样要精通十八般武艺。今天，这个结论依然正确。不过，随着 Ruby 的日渐流行，人们越来越希望从各个角度对 Ruby 进行了解。

本书的风格跟 C 语言的经典《The C Programming Language》相似，书名的相似性自不必说，作者搭配同样也是著名技术作家和语言缔造者的组合。这样的搭配既保证了本书的可读性，也保证了本书的广度和深度。本书与一般的 Ruby 语言图书相比，除了介绍如何使用 Ruby 语言外，还在很多地方讲述了 Ruby 的实现原理及内部运作机制，这当然跟作者的背景分不开。读完本书，您会有知其然亦知其所以然的感觉，同样也更加体会到 Ruby 语言之美。

本书的前面五章由张禾翻译，后面五章由廖志刚翻译，最后由廖志刚统稿。要特别感谢博文的陈元玉和晓菲编辑，她们在全程中一直给我们鞭策和鼓励，也给我们的翻译提出了很多宝贵的意见，是她们的辛勤努力才使得本书呈现出您所看到的最终面貌。

好了，就说这些。请仔细欣赏这颗灿烂夺目的红宝石吧。

廖志刚

2008 年 9 月于西安

目录
Table of Contents

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

前言
Preface

本书是 *Ruby in a Nutshell*（O'Reilly 公司出版）一书的更新和扩展版本，那本书的作者是松本行弘（Yukihiro Matsumoto），他更为人熟知的名字是 Matz。本书的风格有点像那本由 Brian Kernighan 与 Dennis Ritchie 合著的经典书籍《C 编程语言》（*C Programming Language*，Prentice Hall 出版）。本书的目标是在不使用语言规范格式的情况下，详尽讲解 Ruby 的方方面面。本书的读者是那些有其他语言经验的 Ruby 初学者，以及那些希望提升水平的 Ruby 编程人员。

在第 1 章中，你能找到本书的组织架构。

致谢
Acknowledgments

David Flanagan

首先，我必须感谢 Matz 创造了这样一个优美的编程语言，感谢他帮助我理解了这门语言，也感谢他所写的 *Ruby in a Nutshell* 一书，它是本书写作的基础。

我还要感谢：

- *why the lucky stiff*（译注 1），他为本书设计了很多欢快的图片（你将在章首页中见到它们）。当然，还有他那本经典的 Ruby 书籍 *Why's (poignant) guide to Ruby*，该书可以在 *http://poignantguide.net/ruby/* 上找到。
- 我的技术审阅者们：David A. Black、Ruby Power and Light 网站（LLC，*http://rubypal.com/*）的主管、Sun 公司 JRuby 团队的 Charles Oliver Nutter、Ruby 1.8.6 分支的维护者，以及 Ken Cooper。他们的意见提高了本书的质量，并使本书更加清晰。当然，如果还有错误，都是我自己的过错。
- 我的编辑 Mike Loukides。他不断鼓励我创作本书，并在我写作的过程中保持了足够的耐心。

译注 1：这是 Ruby 领域一个著名的名字，当然，这是别名。

最后，我当然要把谢意和爱献给我的家庭。

——David Flanagan
http://www.davidflanagan.com
2008 年 1 月

松本行弘
Yukihiro Matsumoto

除了感谢 David 列出的那些人（不包括我自己），我还要感谢来自全球社区成员的帮助，尤其是那些日本社区的成员，这里仅列出少数几个：Koichi Sasada、Nobuyoshi Nakada、Akira Tanaka、Shugo Maeda、Usaku Nakamura 和 Shyouhei Urabe（排名不分先后）。

最后，我要感谢我的家庭，他们原谅了作为丈夫和父亲的我投身于 Ruby 的开发中。

——Yukihiro Matsumoto
2008 年 1 月

关于字体的约定
Conventions Used in This Book

本书使用如下排版约定：

斜体（*Italic*）或汉仪中黑简体
: 用于目录名、电子邮件地址、URL 和新术语。

等宽字体（`Courier New`）
: 用于代码列表、关键字、变量、函数、命令选项、数据库名、参数、类名和文本中出现的 HTML 标签。

等宽加粗体（**`Courier New`**）
: 用于用户键入的代码和命令行。

等宽加斜体（*`Courier New`*）
: 用作占位符，你应该用自己程序中真实的值来取代它。

代码示例
Using Code Examples

本书是用来帮助你完成工作的。一般而言，你可以在你的代码和文档中使用本书的代码。除非重写了代码的关键部分，否则你无须征求我们的同意。例如，在你自己的代码中使用本书的几块代码是可以的，无须告知我们；但销售或分发 O'Reilly 书籍中代码的 CD 须得到我们的许可。引用本书的示例代码来回答问题无须征求许可；但是在你的产品文档中大量使用本书的示例代码则须征得许可。

我们希望你标明引用，但并不强求。一个引用通常包括标题、作者、出版社和 ISBN 号。例如："*The Ruby Programming Language* by David Flanagan and Yukihiro Matsumoto. Copyright 2008 David Flanagan and Yukihiro Matsumoto, 978-0-596-51617-8."

如果你觉得你对代码的使用方式超出了上述所说明的范围，请通过 *permissions@oreilly.com* 联系我们。

怎样联系我们

How to Contact Us

如果你想就本书发表评论或有任何疑问，敬请联系出版社：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

奥莱理软件（北京）有限公司
北京市 西城区 西直门南大街 2 号 成铭大厦 C 座 807 室
邮政编码：100055
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

北京博文视点资讯有限公司（武汉分部）
湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室
邮政编码：430074
电话：（027）87690813　　传真：（027）87690595
网页：*http://bv.csdn.net*

读者服务信箱：
reader@broadview.com.cn（读者信箱）
bvtougao@gmail.com　　（投稿信箱）

出版商为本书建立了网页，来提供勘误表、示例或其他信息，网址如下：

http://www.oreilly.com/catalog/9780596516178
http://www.oreilly.com.cn/book.php?bn=978-7-121-07701-2（中文版）

如有关于本书的建议或技术问题，请发送邮件到：

bookquestions@oreilly.com

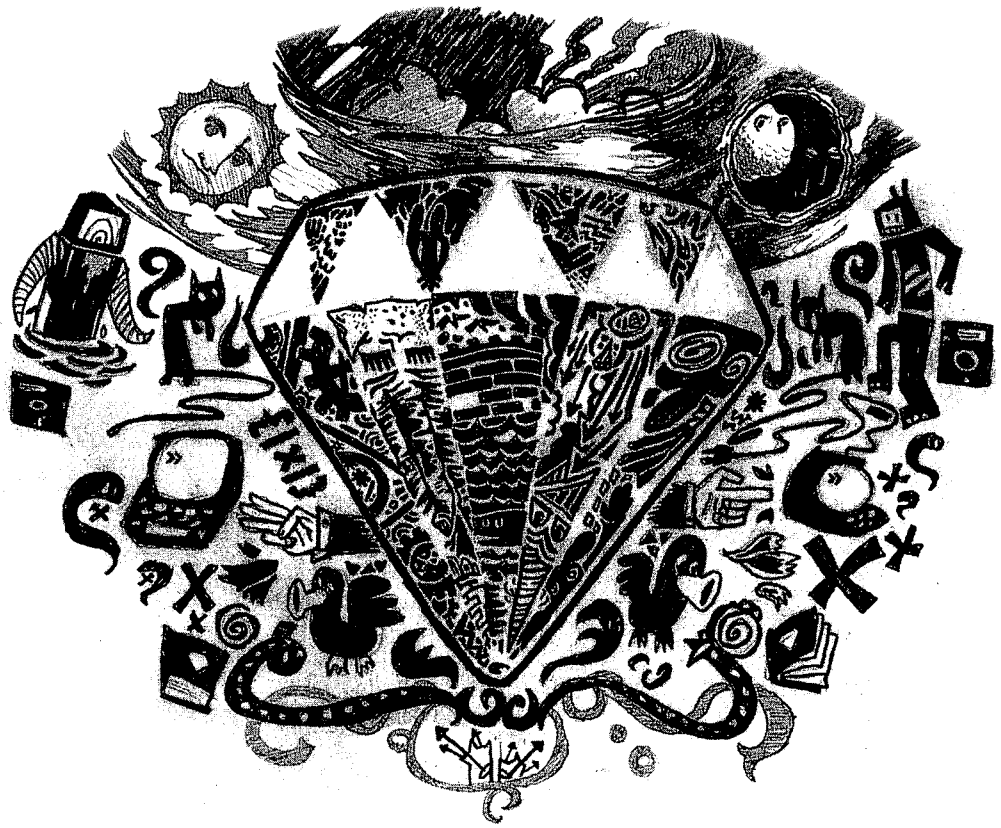
想知道 O'Reilly 更多有关书籍、会议、资源中心和 O'Reilly Network 的信息，请登录 O'Reilly 网站：

http://www.oreilly.com

第 1 章

导言

Introduction



Ruby 是一门动态编程语言。一方面它具有复杂但富于表现力的语法，另一方面它还拥有一套核心类库，该类库包含了丰富而强大的 API。Ruby 的灵感源自 Lisp、Smalltalk 及 Perl 语言，但采用了一套便于 C 和 Java™程序员学习的语法。Ruby 是一门纯粹的面向对象语言，但它同样适用于过程式和函数式编程风格。它拥有强大的元编程（metaprogramming）能力，可用于创建领域特定语言（domain-specific language），即 DSL。

Matz 谈 Ruby

在英语语系的 Ruby 社区里，人们称呼 Yukihiro Matsumoto 为 Matz。他是 Ruby 的创造者，同时也是 *Ruby in a Nutshell*（O'Reilly）（本书正是该书的更新和拓展）一书的作者。他说：

> 在我创造 Ruby 以前，我了解许多语言，但是它们从未令我完全满意过。它们总是和我的期望有差距，要么有些丑陋，要么有些糟糕，要么过于复杂，要么过于简单，因此我想创建属于自己的语言，来满足我作为程序员的需要。我很了解这门语言的目标用户：我自己。令我惊讶的是，世界上的许多程序员和我的感觉一样。他们探索 Ruby 并利用 Ruby 编程时，能够感受到快乐。
>
> 在 Ruby 语言的整个发展过程中，我始终致力于使编程更加快速和简单。Ruby 的所有特性（feature），包括那些面向对象的特性，都被设计成像那些普通的程序员（比如说我）所期望的那样。大多数程序员都觉得 Ruby 优雅且易于使用，用它来编程令人惬意。

Matz 在设计 Ruby 时的指导思想可以用他的那句常被引用的话来概况：

***Ruby* 是作为一门令程序员快乐的语言而设计的。**

1.1 漫游 Ruby
A Tour of Ruby

本节将会带你漫游 Ruby 中一些最为有趣的特性。虽然本书的后续部分会对本节所涉及的内容进行详细的阐述，但是通过此次漫游，你可以先了解这门语言的特质（flavor）。

1.1.1 Ruby 是面向对象的
Ruby Is Object-Oriented

Ruby 是一门**完全**的面向对象语言，我们的旅途就从这里开始。Ruby 里的每个值都是对象，即便是简单的数值字面量，以及 `true`、`false` 和 `nil` 也是如此（`nil` 是一个特殊的值，用于表示“没有值”；它是 Ruby 版的 `null`）。在下面的代码里，我们对一些值调用了名为 `class` 的方法。Ruby 的注释以#开头，而=>箭头后面的内容则表示被注释代码的返回值（这将作为贯穿本书的一个惯例）。

```
1.class      # => Fixnum: the number 1 is a Fixnum
0.0.class    # => Float: floating-point numbers have class Float
```

```
true.class   # => TrueClass: true is a the singleton instance of TrueClass
false.class  # => FalseClass
nil.class    # => NilClass
```

许多语言里，在函数和方法调用后都要接圆括号，但是上述代码里却没有出现圆括号。在Ruby 中，圆括号通常都是可选的而且一般都被省略掉，尤其是在调用那些不带参数的方法时更是如此。由于省略了圆括号，这些方法调用看起来更像是对对象的命名字段（field）或命名变量的引用。虽然这种形式是有意而为的，但是事实上 Ruby 对其对象有着严格的封装（译注 1），在对象之外根本就没有办法直接访问对象的内部状态。这样的访问必须要通过访问器（accessor）方法，比如上述代码中的 `class` 方法。

1.1.2 代码块和迭代器
Blocks and Iterators

Ruby 中，在整数上调用方法并不是什么晦涩难懂的东西，其实 Ruby 程序员经常这么干：

```
3.times { print "Ruby! " }    # Prints "Ruby! Ruby! Ruby! "
1.upto(9) {|x| print x }      # Prints "123456789"
```

`times` 和 `upto` 是由整数对象实现的方法。它们是一种被称为**迭代器**（*iterator*）的特殊方法，其行为类似于循环。包含在花括号中的代码被称为**代码块**（*block*），它们与方法调用相关联并且充当了循环体的角色。对于迭代器和代码块的使用是 Ruby 的另一个显著的特性。虽然 Ruby 的确支持普通的 `while` 循环，但是更为常见的做法是将循环与一些结构搭配使用，而这些结构实际上是方法调用。

除了整数以外，还有其他一些对象也具有迭代器方法。数组（及类似的“可枚举的”对象）定义了一个名为 `each` 的迭代器，它为数组里的每一个元素调用一次相关联的代码块，每次对代码块的调用都以数组里的一个元素作为参数：

```
a = [3, 2, 1]          # This is an array literal
a[3] = a[2] - 1        # Use square brackets to query and set array elements
a.each do |elt|        # each is an iterator. The block has a parameter elt
  print elt+1          # Prints "4321"
end                    # This block was delimited with do/end instead of {}
```

其他各种有用的迭代器都是在 `each` 的基础之上定义的：

```
a = [1,2,3,4]                  # Start with an array
b = a.map {|x| x*x }           # Square elements: b is [1,4,9,16]
c = a.select {|x| x%2==0 }     # Select even elements: c is [2,4]
a.inject do |sum,x|            # Compute the sum of the elements => 10
  sum + x
end
```

与数组类似，哈希（hash）是 Ruby 里的基础数据结构。正如其名字所暗示的那样，它们基于哈希表数据结构，用于将任意的键对象映射到值对象。（换言之，我们可以说哈希将

译注 1：即不要因为看起来像是直接引用了对象的字段，就以为破坏了封装，其实不然。

任意的值对象与键对象相关联。）与数组类似，哈希使用方括号来查询和设置其中的值。不同于数组使用整数值作为索引，哈希使用键对象作为索引。类似于 Array 类，Hash 类定义了一个 each 迭代器方法。这个方法为哈希里的每一个键/值对调用一次相关联的代码块，并且将键和值都作为参数传递给代码块（这也是 Hash 类和 Array 类的不同之处）。

```
h = {                              # A hash that maps number names to digits
  :one => 1,                       # The "arrows" show mappings: key=>value
  :two => 2                        # The colons indicate Symbol literals
}
h[:one]                            # => 1.  Access a value by key
h[:three] = 3                      # Add a new key/value pair to the hash
h.each do |key,value|              # Iterate through the key/value pairs
  print "#{value}:#{key}; "        # Note variables substituted into string
end                                # Prints "1:one; 2:two; 3:three; "
```

Ruby 的哈希可以使用任何对象作为键，但是 Symbol 对象是最为常用的。符号（symbol）是不可变的、功能受限的字符串，可以通过其身份来对符号进行比较，而非通过其文本内容（因为两个不同的 Symbol 对象是绝对不会包含同样的内容的）。

将一个代码块和一个方法调用相关联的能力是 Ruby 的一个基本且非常强大的特性，尽管其最常见的用途是用在那些类似循环的构造里，但是对于那些仅调用代码块一次的方法来说，这一特性同样有帮助。

```
File.open("data.txt") do |f| # Open named file and pass stream to block
  line = f.readline          # Use the stream to read from the file
end                          # Stream automatically closed at block end

t = Thread.new do          # Run this block in a new thread
  File.read("data.txt")    # Read a file in the background
end                        # File contents available as thread value
```

顺便提一下，请注意前面关于 Hash.each 的例子里包含这一行有趣的代码：

```
print "#{value}:#{key}; "     # Note variables substituted into string
```

由双引号引起来的字符串里可以包含任何 Ruby 表达式，前提是这些表达式包含在分界符 #{和}里。分界符之间的表达式的值将被转换成一个字符串（通过调用其 to_s 方法，所有的对象都支持该方法），结果字符串将替换原字符串字面量里的表达式文本及分界符。这种将表达式的值替换进字符串的方式通常被称为**字符串内插**（*string interpolation*）。

1.1.3 Ruby 中的表达式和操作符
Expressions and Operators in Ruby

Ruby 的语法是面向表达式的，诸如 if 之类的控制结构在其他语言里被称为语句，但是在 Ruby 里，它们其实是表达式，它们和其他更为简单的表达式一样具有值。我们可以像下面这样编写代码：

```
minimum = if x < y then x else y end
```

虽然在 Ruby 中所有“语句”实际上都是表达式，但是它们并不都会返回有意义的值。比方说，虽然 while 循环和方法定义都是表达式，但是它们在正常情况下都会返回 nil。

和大多数语言一样，Ruby 中的表达式通常由值和操作符构成。那些了解 C、Java、JavaScript 或任何类似语言的人，会觉得 Ruby 中的大部分操作符都很熟悉。下面是一些 Ruby 操作符的例子，包含了一些常用的和一些不太常见的操作符。

```
1 + 2                    # => 3: addition
1 * 2                    # => 2: multiplication
1 + 2 == 3               # => true: == tests equality
2 ** 1024                # 2 to the power 1024: Ruby has arbitrary size ints
"Ruby" + " rocks!"       # => "Ruby rocks!": string concatenation
"Ruby! " * 3             # => "Ruby! Ruby! Ruby! ": string repetition
"%d %s" % [3, "rubies"]  # => "3 rubies": Python-style, printf formatting
max = x > y ? x : y      # The conditional operator
```

许多 Ruby 操作符都是作为方法来实现的，而且类可以按照它们的需要来定义（或重定义）这些方法。（但是它们不能定义全新的操作符，因为可识别的操作符是一个固定的集合。）比方说，对于整数和字符串来说，+ 和 * 的行为是不同的，而且你还可以在自己的类里随意地定义这些方法。另一个不错的例子是<<操作符。依照 C 语言的做法，Ruby 中的整型类 Fixnum 和 Bignum 都使用<<操作符来进行按位左移操作。与此同时（依照 C++的做法），在其他类里——比如字符串、数组及流——使用这个操作符来做添加操作。如果你创建了一个新类，可以用某种方式将值追加给它，那么在这个类里定义<<操作符是一个好主意。

[]操作符是可被重写的最为强大的操作符之一。通过这个操作符，Array 类可以用索引来访问数组元素，而 Hash 类则可以用键来访问哈希值。在你的类里面，可以根据任何目的重定义[]操作符，你甚至还可以将其定义成接受多个以逗号分隔的参数的方法。（Array 类的[]操作符接受一个索引和一个长度作为参数，来表明原数组的一个子数组或“片段（slice）”。）此外，如果你希望[]操作符能够在一个赋值表达式的左侧使用，那么你可以定义对应的[]=操作符，赋值操作右侧的值将作为最后一个参数被传递给那个实现了[]=操作符的方法。

1.1.4 方法
Methods

我们可以用 def 关键字来定义方法，方法的返回值是方法体中最后一个被执行的表达式的值，示例如下：

```
def square(x)     # Define a method named square with one parameter x
  x*x             # Return x squared
end               # End of the method
```

如果将上述方法定义在类或模块（module）之外，那么它就是一个全局函数，而不是一个方法（须要通过某个对象来调用方法）。（但是从技术上来讲，一个这样的方法会成为Object类的私有方法。）还可以专门为某个对象定义方法，只要以对象名为方法名的前缀即可。这样的方法被称为**单键（*singleton*）方法**，而且它们也是Ruby定义类方法的方式：

```
def Math.square(x)  # Define a class method of the Math module
  x*x
end
```

Math模块是Ruby核心类库的一部分，上述代码给它添加了一个新方法。这是Ruby的关键特性之一——类和模块都是“开放的”，而且可以在运行时修改和扩展。

可以为方法参数指定默认值，而且方法可以接受任意数量的参数。

1.1.5 赋值
Assignment

Ruby当中的=操作符（它是不可重写的）负责将一个值赋给一个变量：

```
x = 1
```

赋值操作符可以和其他操作符进行组合，比如+和-：

```
x += 1          # Increment x: note Ruby does not have ++.
y -= 1          # Decrement y: no -- operator, either.
```

Ruby支持并行赋值，即允许在赋值表达式中出现多于一个的值和多于一个的变量：

```
x, y = 1, 2       # Same as x = 1; y = 2
a, b = b, a       # Swap the value of two variables
x,y,z = [1,2,3]   # Array elements automatically assigned to variables
```

Ruby允许其方法返回多个值，当与这样的方法联合使用时，并行赋值非常有用。比如：

```
# Define a method to convert Cartesian (x,y) coordinates to Polar
def polar(x,y)
  theta = Math.atan2(y,x)     # Compute the angle
  r = Math.hypot(x,y)         # Compute the distance
  [r, theta]                  # The last expression is the return value
end

# Here's how we use this method with parallel assignment
distance, angle = polar(2,2)
```

以等号（=）结尾的方法比较特殊，因为Ruby允许以赋值操作的语法来调用它们。假如一个对象o拥有一个叫做x=的方法，那么下面两行代码所做的事情是一样的：

```
o.x=(1)        # Normal method invocation syntax
o.x = 1        # Method invocation through assignment
```

1.1.6 作为前缀和后缀的标点符号
Punctuation Suffixes and Prefixes

在前面我们已经见到，可以在赋值表达式中调用以=结尾的方法。Ruby 中的方法还可以以问号或感叹号结尾，问号被用于标示谓词，即返回 Boolean 值的方法。比方说，Array 和 Hash 类都定义了一个 empty?方法，它会测试该数据结构是否包含元素。出现在方法名尾部的感叹号表明使用该方法时需要多加小心。许多 Ruby 核心类都定义了成对的方法，它们具有同样的名字，彼此的差别在于其中一个以感叹号结尾，而另一个则没有。通常情况下，不带感叹号的方法返回调用该方法的对象的一个修改过的拷贝，而带感叹号的方法则是一个可变方法（mutator（译注 2）），该方法会修改原对象。比如，Array 类就定义了 sort 和 sort!方法。

除了这些出现在方法名后面的标点符号之外，你还将见到出现在 Ruby 变量名头部的标点符号：全局变量以$为前缀，实例变量以@为前缀，类变量以@@为前缀。习惯这些前缀须要付出一点点代价，但是过一会儿你就会意识到这些前缀可以告诉你变量的作用域。为了消除 Ruby 那非常灵活的语法所带来的一些歧义，这些前缀是必须的。你可以这样看待这些前缀：它们是为了省略方法调用时的圆括号而付出的代价。

1.1.7 Regexp 和 Range
Regexp and Range

我们先前提到数组和哈希是 Ruby 的基本数据结构，同时也演示了数字和字符串的使用。还有两种数据类型值得一提：一个是 Regexp（正则表达式）对象，它描述了一个文本模式并且含有方法来判断一个给定的字符串是否匹配该模式；而另一个是 Range，它代表了位于两个端点之间的值（通常是整数）。在 Ruby 当中，正则表达式和范围（range）都有字面量语法：

```
/[Rr]uby/          # Matches "Ruby" or "ruby"
/\d{5}/            # Matches 5 consecutive digits
1..3               # All x where 1 <= x <= 3
1...3              # All x where 1 <= x < 3
```

Regexp 和 Range 对象为相等性测试定义了标准的==操作符，此外，它们还定义了===操作符来测试匹配性（matching）和从属关系（membership）。Ruby 的 case 语句（类似于 C 或 Java 中的 switch 语句）使用===来将它的表达式和每个可能的条件分支进行比较，所以===操作符也常常被称为**条件相等性操作符**（*equality operator*）。它进行像下面这样的条件测试：

译注 2：可变方法是指方法会修改对象的状态，而非方法本身可变。

```
# Determine US generation name based on birth year
# Case expression tests ranges with ===
generation = case birthyear
           when 1946..1963: "Baby Boomer"
           when 1964..1976: "Generation X"
           when 1978..2000: "Generation Y"
           else nil
           end

# A method to ask the user to confirm something
def are_you_sure?                  # Define a method. Note question mark!
  while true                       # Loop until we explicitly return
    print "Are you sure? [y/n]: "  # Ask the user a question
    response = gets                # Get her answer
    case response                  # Begin case conditional
    when /^[yY]/                   # If response begins with y or Y
      return true                  # Return true from the method
    when /^[nN]/, /^$/             # If response begins with n,N or is empty
      return false                 # Return false
    end
  end
end
```

1.1.8 类和模块
Classes and Modules

一个类就是一些相关方法的集合，这些方法将操作一个对象的状态。一个对象的状态被保存在它的实例变量中：即那些以@开头的变量，其值是特定于该对象的。下面的代码定义了一个叫做 Sequence 的示例类，并且演示了如何编写迭代器方法和操作符。

```
#
# This class represents a sequence of numbers characterized by the three
# parameters from, to, and by. The numbers x in the sequence obey the
# following two constraints:
#
#    from <= x <= to
#    x = from + n*by, where n is an integer
#
class Sequence
  # This is an enumerable class; it defines an each iterator below.
  include Enumerable   # Include the methods of this module in this class

  # The initialize method is special; it is automatically invoked to
  # initialize newly created instances of the class
  def initialize(from, to, by)
    # Just save our parameters into instance variables for later use
    @from, @to, @by = from, to, by  # Note parallel assignment and @ prefix
  end

  # This is the iterator required by the Enumerable module
  def each
    x = @from          # Start at the starting point
    while x <= @to     # While we haven't reached the end
```

```
    yield x          # Pass x to the block associated with the iterator
    x += @by         # Increment x
  end
end

# Define the length method (following arrays) to return the number of
# values in the sequence
def length
  return 0 if @from > @to        # Note if used as a statement modifier
  Integer((@to-@from)/@by) + 1   # Compute and return length of sequence
end

# Define another name for the same method.
# It is common for methods to have multiple names in Ruby
alias size length  # size is now a synonym for length

# Override the array-access operator to give random access to the sequence
def[](index)
  return nil if index < 0    # Return nil for negative indexes
  v = @from + index*@by      # Compute the value
  if v <= @to                # If it is part of the sequence
    v                        # Return it
  else                       # Otherwise...
    nil                      # Return nil
  end
end

# Override arithmetic operators to return new Sequence objects
def *(factor)
  Sequence.new(@from*factor, @to*factor, @by*factor)
end

def +(offset)
  Sequence.new(@from+offset, @to+offset, @by)
end
end
```

下面是使用 Sequence 类的代码：

```
s = Sequence.new(1, 10, 2)    # From 1 to 10 by 2's
s.each {|x| print x }         # Prints "13579"
print s[s.size-1]             # Prints 9
t = (s+1)*2                   # From 4 to 22 by 4's
```

Sequence 类的关键特性就是其 each 迭代器。如果我们仅对迭代器方法感兴趣，那么就没有必要定义整个类。我们只须简单地编写一个接受 from、to 及 by 参数的迭代器方法即可。为了不让这个迭代器方法成为一个全局函数，我们将其定义在如下属于它自己的模块里：

```
module Sequences                        # Begin a new module
  def self.fromtoby(from, to, by)       # A singleton method of the module
    x = from
    while x <= to
      yield x
      x += by
    end
```

```
  end
end
```

有了这样的迭代器，我们就可以像下面这样编写代码：

```
Sequences.fromtoby(1, 10, 2) {|x| print x }  # Prints "13579"
```

一个这样的迭代器使得没有必要为了迭代一个数字序列而创建一个 Sequence 对象，但是这个方法的名字很长，而且其调用语法也难以让人满意，我们真正想要的是以非 1 的步长来对数值的 Range 对象进行迭代。Ruby 令人惊异的特性之一就是它的类都是**开放**的，即便是那些内建的核心类也是如此，这意味着任何程序都可以为它们添加方法。所以我们可以为 Range 类定义一个新的迭代器：

```
class Range                     # Open an existing class for additions
  def by(step)                  # Define an iterator named by
    x = self.begin              # Start at one endpoint of the range
    if exclude_end?             # For ... ranges that exclude the end
      while x < self.end        # Test with the < operator
        yield x
        x += step
      end
    else                        # Otherwise, for .. ranges that include the end
      while x <= self.end       # Test with <= operator
        yield x
        x += step
      end
    end
  end                           # End of method definition
end                             # End of class modification

# Examples
(0..10).by(2) {|x| print x}   # Prints "0246810"
(0...10).by(2) {|x| print x}  # Prints "02468"
```

虽然这个 by 方法使用起来很方便，但它却是多余的；因为 Range 类已经定义了一个名为 step 的迭代器，用于实现同样的目的。Ruby 的核心 API 是很丰富的，这个平台是值得花时间去学习的（请参见第 9 章），这样你就不必浪费时间去编写那些已经实现的方法了。

1.1.9 Ruby 的意外之处
Ruby Surprises

每一门语言都会有那样一些特性，它们使那些刚接触该语言的人大伤脑筋，Ruby 也不例外。下面我们就来描述两个令人吃惊的特性。

Ruby 的字符串是可变的，这尤其会使 Java 程序员大伤脑筋。[]=操作符允许你改变一个字符串中的字符，或者插入、删除及替换子字符串。<<操作符允许你在一个字符串的后面追加一些东西，而且 String 类还定义了各种其他的方法对字符串进行就地的更改。因为字符串是可变的，所以字符串字面量在程序中并不是独一无二的对象。如果你在一个循环

内包含了一个字符串字面量，那么每次循环都会为它创建一个新的对象。通过在一个字符串（或任何对象）上调用 `freeze` 方法，你可以防止将来对该对象的任何改变。

Ruby 的条件式和循环（比如 `if` 和 `while`）通过评估条件表达式来决定执行哪一个分支或是否继续循环。条件表达式通常求值为 `true` 或 `false`，但是这并不是必须的。`nil` 值被当成 `false` 来处理，而**任何其他值都和 `true` 一样处理**。这很可能会让 C 程序员和 JavaScript 程序员大吃一惊，因为前者期望 `0` 能够像 `false` 那样工作，而后者则期望空字符串`""`能够像 `false` 那样工作。

1.2 体验 Ruby

Try Ruby

我们希望这趟 Ruby 关键特性之旅已经激起了你的兴趣，而且你正渴望体验一下 Ruby。为了达到目的，你需要一个 Ruby 解释器，并且你还要知道如何使用三个工具，即 *irb*、*ri* 和 *gem*，它们是和 Ruby 解释器捆绑在一起的。本节将解释如何获得和使用它们。

1.2.1 Ruby 解释器

The Ruby Interpreter

Ruby 的官方站点是 *http://www.ruby-lang.org*。如果你的电脑上还没有安装 Ruby，那么你可以参照 *ruby-lang.org* 主页上的下载链接来获得一些说明，这些说明是关于下载和安装基于标准 C 的 Ruby 参考实现的。

一旦安装了 Ruby，你就可以使用 `ruby` 命令来调用 Ruby 解释器：

```
% ruby -e 'puts "hello world!"'
hello world!
```

`-e` 命令行选项使解释器执行一行指定的 Ruby 代码。更为常见的做法是，你将 Ruby 程序输入到一个文件中，然后告诉解释器去调用这个文件：

```
% ruby hello.rb
hello world!
```

其他的 Ruby 实现

在缺少一份正式的 Ruby 语言规范的情况下，ruby-lang.org 上的 Ruby 解释器就是定义该语言的参考实现。有时它被称为 MRI 或“Matz's Ruby Implementation”。对于 Ruby 1.9，原先的 MRI 解释器与 YARV（“Yet Another Ruby Virtual machine”）合并之后产生了一个新的参考实现。该实现将 Ruby 代码编译成字节码（bytecode），然后在一个虚拟机上执行该字节码。

然而，这个参考实现并不是唯一的 Ruby 实现。在本书写作之时，有一个可供选择的实现（JRuby）已经发布了，而且还有其他几个实现也在开发当中。

JRuby

JRuby 是一个基于 Java 的 Ruby 实现，可以从 *http://jruby.org* 获得。在本书写作之时，当前的发行版是 JRuby 1.1，它与 Ruby 1.8 兼容。当你读到本书时，一个与 Ruby 1.9 兼容的 JRuby 也许已经出现了。JRuby 是开放源代码软件，主要由 Sun 公司进行开发。

IronRuby

IronRuby 是微软为其.NET 框架和动态语言运行时（Dynamic Language Runtime，DLR）而实现的 Ruby 版本。IronRuby 的源代码在微软的许可证下也可以获得。在本书写作之时，IronRuby 还没有达到 1.0 版本的水平。项目的主页是 *http://www.ironruby.net*。

Rubinius

Rubinius 是一个开放源代码项目，该项目对自己的描述是“一个主要由 Ruby 编写的可选的 Ruby 实现。Rubinius 的虚拟机名为 shotgun，它松散地基于 Smalltalk-80 虚拟机架构。”在本书写作之时，Rubinius 并没有达到 1.0 版本。Rubinius 项目的主页位于 *http://rubini.us*。

Cardinal

Cardinal 是一个打算在 Parrot 虚拟机（其目标是支持 Perl 6 和许多其他动态语言）上运行的 Ruby 实现。在本书写作之时，Parrot 和 Cardinal 都没能达到 1.0 版本。Cardinal 没有自己的主页，而是作为 Parrot 开放源代码项目的一部分存在的，位于 *http://www.parrotcode.org*。

1.2.2 显示输出
Displaying Output

为了体验 Ruby 的特性，你需要一个显示输出的方法，使你的测试程序可以打印它们的执行结果。在之前的“hello world”代码里用到的 `puts` 函数就是一种显示输出的方式。不太严谨的解释是，`puts` 将一个文本字符串打印到控制台并在其后添加一个换行符（除非该字符串已经以一个换行符结尾了）。如果传递给 `puts` 的对象不是一个字符串，那么 `puts` 就调用该对象的 `to_s` 方法，并且打印该方法所返回的字符串。`print` 方法和 `puts` 方法做的基本上是同一件事情，但是它并不在末尾添加一个换行符。举例来讲，将下面的两行代码输入一个文本编辑器并且保存到一个名为 *count.rb* 的文件里。

```
9.downto(1) {|n| print n }          # No newline between numbers
puts " blastoff!"                   # End with a newline
```

现在用 Ruby 解释器来运行该程序：

```
% ruby count.rb
```

它会产生如下的输出：

```
987654321 blastoff!
```

你也许发现函数 `p` 是 `puts` 的一个有用的替代者，不仅仅是因为 `p` 比 `puts` 更简短，而且

还因为它是通过 inspect 方法将对象转换成字符串的，有时候此方法能比 to_s 方法返回对程序员更加友好的对象表示。比如，当要打印一个数组时，p 的输出采用的是数组字面量的表示法（译注 3），而 puts 则只是简单地将每个元素打印到单独的一行。

1.2.3 使用 irb 与 Ruby 进行交互
Interactive Ruby with irb

irb（"interactive Ruby" 的缩写）是一个 Ruby shell。在 irb 的提示符下输入任意的 Ruby 表达式，它会执行该表达式并将其值显示出来，这通常是你体验那些从本书阅读到的语言特性的最容易的方式。下面是一段带注解的 *irb* 会话的例子：

```
$ irb --simple-prompt          # Start irb from the terminal
>> 2**3                        # Try exponentiation
=> 8                           # This is the result
>> "Ruby! " * 3                # Try string repetition
=> "Ruby! Ruby! Ruby! "        # The result
>> 1.upto(3){|x| puts x }      # Try an iterator
1                              # Three lines of output
2                              # Because we called puts 3 times
3
=> 1                           # The return value of 1.upto(3)
>> quit                        # Exit irb
$                              # Back to the terminal prompt
```

这个会话例子向你展示了在探索 Ruby 时，为了有效地利用 *irb* 而所须要了解的一切。此外确实还有许多其他重要的特性，包括子 shell（subshell，在 irb 的提示符下输入 "irb" 即可开启一个子 shell）和可配置性。

1.2.4 使用 ri 查看 Ruby 文档
Viewing Ruby Documentation with ri

ri（注 1）文档查看器是另一个重要的 Ruby 工具。在命令行调用 *ri*，后接一个 Ruby 类、模块或方法的名字，*ri* 就会显示对应的文档。你也可以只指定一个方法名而不加类名或模块名作为限定，但是这样会显示一个包含了所有叫这个名字的方法的列表（除非这个方法是独一无二的）。通常情况下，你可以用句点将类或模块名与方法名进行分隔。如果一个类定义了同名的类方法和实例方法，那么你就要用::引用类方法，而用#引用实例方法。下面是一些调用 *ri* 的例子：

```
ri Array
ri Array.sort
ri Hash#each
ri Math::sqrt
```

译注 3：比如[1, 2, 3]。

注 1：对于 "ri" 代表了什么存在不同的意见。它的称谓包括："Ruby Index"，"Ruby Information" 及 "Ruby Interactive"。

这些被 *ri* 显示的文档是从 Ruby 源代码中的一些注释里提取出来的，这些注释是经过特殊格式化了的。详情请参见第 2.1.1.2 节。

1.2.5 使用 gem 进行 Ruby 包管理
Ruby Package Management with gem

Ruby 的包管理系统被称为 RubyGems。使用 RubyGems 来发布的包或模块被称为“gems”。RubyGems 使安装 Ruby 软件变得易如反掌，而且能够自动管理包之间复杂的依赖关系。

RubyGems 的前端脚本是 *gem*。在 Ruby 1.9 里，它和 *irb*、*ri* 一样，是随 Ruby 一起发行的。在 Ruby 1.8 里，你必须单独安装它——参见 *http://rubygems.org*。一旦安装了 *gem*，你可以像下面这样使用它：

```
# gem install rails
Successfully installed activesupport-1.4.4
Successfully installed activerecord-1.15.5
Successfully installed actionpack-1.13.5
Successfully installed actionmailer-1.3.5
Successfully installed actionwebservice-1.2.5
Successfully installed rails-1.2.5
6 gems installed
Installing ri documentation for activesupport-1.4.4...
Installing ri documentation for activerecord-1.15.5...
...etc...
```

正如你所见到的，`gem install` 命令将安装你所需要的 gem 的最新版本，如果在安装这些 gem 的时候，要用到任何其他的 gem，它也会一并安装。*gem* 还有其他一些有用的子命令。举例如下：

```
gem list                  # List installed gems
gem enviroment            # Display RubyGems configuration information
gem update rails          # Update a named gem
gem update                # Update all installed gems
gem update --system       # Update RubyGems itself
gem uninstall rails       # Remove an installed gem
```

在 Ruby 1.8 里，你安装的 gem 无法被 Ruby 的 `require` 方法自动加载（请参见第 7.6 节获得更多关于使用 `require` 方法加载 Ruby 代码模块的信息）。如果你编写的程序须要使用一些以 gem 形式安装的模块，那么在 `require` 这些模块之前，必须首先 require `rubygems` 这个模块。有一些 Ruby 1.8 的发行版预先配置了 `RubyGems` 库，但是你须要手工下载和安装。`rubygems` 模块的载入将改变 `require` 方法本身，使它在搜索标准库之前先搜索由安装过的 gems 所组成的集合。你也可以通过在运行 Ruby 时指定`-rubygems` 命令行选项的方式来自动开启 `RubyGems` 的支持。此外，如果你将`-rubygems` 添加到 `RUBYOPT` 环境变量里，那么当每次调用 Ruby 的时候，都会载入 `RubyGems` 库。

虽然 `rubygems` 模块成了 Ruby 1.9 标准库的一部分，但由于 Ruby 1.9 本身知道如何找到那些安装过的 gems，所以对于加载 gems 这件事情来说，`rubygems` 模块已经不再是必需的了，你无须再把 `require 'rubygems'`这行代码放到那些须要使用 gem 的程序中了。

当你使用 require 来载入一个 gem 时（1.8 和 1.9 版皆是如此），require 会载入你所指定的 gem 的最新版本。如果有更加具体的版本需求，那么可以在调用 require 之前使用 gem 方法来指定，这将找到一个和你所指定的版本约束相匹配的 gem 版本并且“激活”它，如此一来，接下来的 require 就会载入该版本的 gem：

```
require 'rubygems'                          # Not necessary in Ruby 1.9
gem 'RedCloth', '> 2.0', '< 4.0'            # Activate RedCloth version 2.x or 3.x
require 'RedCloth'                          # And now load it
```

你可以在第 7.6.1 节找到更多关于 require 和 gem 的信息。对于 RubyGems、*gem* 程序及 rubygems 模块的完整描述超出了本书的范围。gem 是一个自我文档化（self-documenting）的命令，可以从运行 gem help 开始。如果需要 gem 方法的详细信息，请尝试 ri gem。此外，可以参见 *http://rubygems.org* 的文档获得完整的细节信息。

1.2.6 更多 Ruby 教程
More Ruby Tutorials

本章以一个 Ruby 语言的指导性介绍开篇。你可以使用 *irb* 来试验一下该教程中的代码片段。如果你希望在正式深入研究 Ruby 语言之前能够获得更多的教程，那么跟随 *http://www.ruby-lang.org* 上的链接可以得到两个不错的教程：其中之一是被称为“Ruby in Twenty Minutes”的一个基于 *irb* 的教程（注 2）；另一个教程被称为“Try Ruby!”，它的有趣之处在于它运行在你的 web 浏览器里，而你的系统上无须安装 Ruby 或 *irb*（注 3）。

1.2.7 Ruby 资源
Ruby Resources

Ruby 的官方站点（*http://www.ruby-lang.org*）是寻找指向其他 Ruby 资源的链接的地方，这些资源包括在线文档、库、邮件列表、博客、IRC 频道、用户组及会议。请尝试主页上题为“Documentation”、“Libraries”及“Community”的链接。

1.3 关于本书
About This Book

正如其标题所暗示的那样，本书涵盖了 Ruby 语言的方方面面，并且希望能够讲得全面和易于理解。本书所讨论的这个版本涵盖了 Ruby 语言的 1.8 和 1.9 版。Ruby 使语言和平台之间的界限模糊起来，所以我们对 Ruby 语言的讲解里也包含了对 Ruby 核心 API 的具体概述。但是，由于本书不是一个 API 参考手册，所以并没有完全地涵盖 Ruby 的核心类。

注 2：当本书写作之时，本教程的 URL 是 *http://www.ruby-lang.org/en/documentation/quickstart/*。
注 3：如果你在 Ruby 的主页上找不到“Try Ruby!”的链接，请试一下这个：*http://tryruby.hobix.com*。

此外，本书既不是一本关于 Ruby 框架（像 Rails）的书，也不是一本关于 Ruby 工具（像 *rake* 和 *gem*）的书。

本章以一个包含了大量注释的例子结束，该例子展示了一个较为复杂的 Ruby 程序。后续章节以自下而上的方式对 Ruby 进行了详尽的阐述。

- 第 2 章涵盖了 Ruby 的词法和句法结构，包含一些基本的问题，比如字符集、大小写敏感性及保留字。
- 第 3 章解释了数据类型——数字、字符串、range、数组等——这些都是 Ruby 程序所能操作的，此外还涵盖了所有 Ruby 对象都具有的一些基本特性。
- 第 4 章涵盖了基本的 Ruby 表达式——字面量（literal）、变量引用、**方法**调用及赋值——此外还解释了用于将基本的 Ruby 表达式组合成复合表达式的操作符。
- 第 5 章解释了那些在其他语言里被称为语句或控制结构的 Ruby 表达式，包括条件式（conditionals）、循环（loops）（包含代码块和迭代器方法）、异常，以及其他 Ruby 表达式。
- 第 6 章正式地阐述了 Ruby 的方法定义和调用语法。而且还涵盖了名为 procs 和 lambdas 的可调用对象。这一章还包含对于闭包（closure）的解释及对于 Ruby 中函数式编程的探索。
- 第 7 章解释了如何在 Ruby 中定义类和模块。类是面向对象编程的基础。此外本章还涵盖了诸如继承、方法可见性、mixin 模块及方法名解析算法等主题。
- 第 8 章涵盖了一些使程序可以对自身进行检查和操作的 Ruby API，此外还展示了元编程技术，该技术借助上述 API 使编程更加容易。本章还包含了一个领域特定语言的例子。
- 第 9 章用一些简单的代码片段展示了 Ruby 的核心平台里最为重要的类和方法。这并不是一个核心类的参考手册，而是一个具体的概述，主题包括文本处理、数值计算、集合（比如数组和哈希）、输入/输出、网络及线程。阅读完该章之后，你将理解 Ruby 平台的广度，而且你可以使用 *ri* 或一份在线文档来深入探索 Ruby 平台。
- 第 10 章涵盖了最顶层的 Ruby 语言环境，包括全局变量和函数、Ruby 解释器所支持的命令行参数，以及 Ruby 的安全机制。

1.3.1 如何阅读本书
How to Read This Book

使用 Ruby 编程很轻松，但是 Ruby 却不是一门简单的语言。由于本书全面地阐述了 Ruby，所以它也不是一本简单的书（尽管我们希望你能够很容易地阅读和理解本书）。本书是为那些有经验的程序员而准备的，他们希望掌握 Ruby 而且愿意通过认真地阅读和思考来达到该目的。

像其他的编程书籍一样，本书通篇都包含了向前和向后的引用。编程语言并不是线性系统，也不可能对它们进行线性的描述。正如你在章节大纲里所见的那样，本书采用了自下而上的方式来描述 Ruby，从描述 Ruby 语法中最简单的元素开始，过渡到对后续的更高级的语法结构的讲解——从标记到值再到表达式，从控制结构到方法再到类。这是描述编程语言的一种经典的方式，但是这也不能避免向前引用的问题。

本书的初衷是可以让你按照写作顺序来进行阅读，但是对于一些高级主题来说，在第一遍阅读时最好采取略读或跳过的方式，因为在阅读完后续章节后再回过头来审视这些主题时，将会有更好的理解。从另一个角度来讲，也不要被那些向前的引用给吓跑了，其实许多这样的向前引用都只是简单地提供一些信息，为了告诉你后面将会提供更多的细节，这些向前引用并不意味着为了要理解当前的内容，就必须要先了解那些将来才涉及的细节。

1.4 一个 Ruby 版的 Sudoku 解答
A Sudoku Solver in Ruby

本章以一个较为复杂的 Ruby 程序结束，其目的在于让你对 Ruby 程序的外观有一个更好的概念。我们选择以一个 Sudoku（注 4）解答来作为样例程序，它是一段中等长度的程序，其中展示了一些 Ruby 的特性。不要期望能够理解示例 1-1 的每个细节，但是仔细阅读还是必要的。这段程序的注释很完整，所以阅读起来应该问题不大。

示例 1-1：一个 Ruby 版的 Sudoku 解答

```
#
# This module defines a Sudoku::Puzzle class to represent a 9x9
# Sudoku puzzle and also defines exception classes raised for
# invalid input and over-constrained puzzles. This module also defines
# the method Sudoku.solve to solve a puzzle. The solve method uses
# the Sudoku.scan method, which is also defined here.
#
# Use this module to solve Sudoku puzzles with code like this:
```

注 4：Sudoku 是一种逻辑谜题。它的形式是一种 9×9 的网格，每个方格里都可以放入数字。这个谜题的任务是用 1~9 的数字填满所有方格，并使任意行、列或 3×3 的子网格里都不会两次出现同一个数字。Sudoku 在日本流行了一段时间，而且在 2004 年和 2005 年的时候，突然在英语语系的国家里流行起来。如果你还不了解 Sudoku，请阅读维基百科里的对应条目（*http://en.wikipedia.org/wiki/Sudoku*），并且尝试一下谜题的在线版（*http://websudoku.com/*）。

```
#
#  require 'sudoku'
#  puts Sudoku.solve(Sudoku::Puzzle.new(ARGF.readlines))
#
module Sudoku

  #
  # The Sudoku::Puzzle class represents the state of a 9x9 Sudoku puzzle.
  #
  # Some definitions and terminology used in this implementation:
  #
  # - Each element of a puzzle is called a "cell".
  # - Rows and columns are numbered from 0 to 8, and the coordinates [0,0]
  #   refer to the cell in the upper-left corner of the puzzle.
  # - The nine 3x3 subgrids are known as "boxes" and are also numbered from
  #   0 to 8, ordered from left to right and top to bottom. The box in
  #   the upper-left is box 0. The box in the upper-right is box 2. The
  #   box in the middle is box 4. The box in the lower-right is box 8.
  #
  # Create a new puzzle with Sudoku::Puzzle.new, specifying the initial
  # state as a string or as an array of strings. The string(s) should use
  # the characters 1 through 9 for the given values, and '.' for cells
  # whose value is unspecified. Whitespace in the input is ignored.
  #
  # Read and write access to individual cells of the puzzle is through the
  # [] and []= operators, which expect two-dimensional [row,column] indexing.
  # These methods use numbers (not characters) 0 to 9 for cell contents.
  # 0 represents an unknown value.
  #
  # The has_duplicates? predicate returns true if the puzzle is invalid
  # because any row, column, or box includes the same digit twice.
  #
  # The each_unknown method is an iterator that loops through the cells of
  # the puzzle and invokes the associated block once for each cell whose
  # value is unknown.
  #
  # The possible method returns an array of integers in the range 1..9.
  # The elements of the array are the only values allowed in the specified
  # cell. If this array is empty, then the puzzle is over-specified and
  # cannot be solved. If the array has only one element, then that element
  # must be the value for that cell of the puzzle.
  #
  class Puzzle

    # These constants are used for translating between the external
    # string representation of a puzzle and the internal representation.
    ASCII = ".123456789"
    BIN = "\000\001\002\003\004\005\006\007\010\011"

    # This is the initialization method for the class. It is automatically
    # invoked on new Puzzle instances created with Puzzle.new. Pass the input
    # puzzle as an array of lines or as a single string. Use ASCII digits 1
    # to 9 and use the '.' character for unknown cells. Whitespace,
    # including newlines, will be stripped.
    def initialize(lines)
```

```
  if (lines.respond_to? :join) # If argument looks like an array of lines
    s = lines.join             # Then join them into a single string
  else                         # Otherwise, assume we have a string
    s = lines.dup              # And make a private copy of it
  end

  # Remove whitespace (including newlines) from the data
  # The '!' in gsub! indicates that this is a mutator method that
  # alters the string directly rather than making a copy.
  s.gsub!(/\s/, "")  # /\s/ is a Regexp that matches any whitespace

  # Raise an exception if the input is the wrong size.
  # Note that we use unless instead of if, and use it in modifier form.
  raise Invalid, "Grid is the wrong size" unless s.size == 81

  # Check for invalid characters, and save the location of the first.
  # Note that we assign and test the value assigned at the same time.
  if i = s.index(/[^123456789\.]/)
    # Include the invalid character in the error message.
    # Note the Ruby expression inside #{} in string literal.
    raise Invalid, "Illegal character #{s[i,1]} in puzzle"
  end

  # The following two lines convert our string of ASCII characters
  # to an array of integers, using two powerful String methods.
  # The resulting array is stored in the instance variable @grid
  # The number 0 is used to represent an unknown value.
  s.tr!(ASCII, BIN)        # Translate ASCII characters into bytes
  @grid = s.unpack('c*')   # Now unpack the bytes into an array of numbers

  # Make sure that the rows, columns, and boxes have no duplicates.
  raise Invalid, "Initial puzzle has duplicates" if has_duplicates?
end

# Return the state of the puzzle as a string of 9 lines with 9
# characters (plus newline) each.
def to_s
  # This method is implemented with a single line of Ruby magic that
  # reverses the steps in the initialize() method. Writing dense code
  # like this is probably not good coding style, but it demonstrates
  # the power and expressiveness of the language.
  #
  # Broken down, the line below works like this:
  # (0..8).collect invokes the code in curly braces 9 times--once
  # for each row--and collects the return value of that code into an
  # array. The code in curly braces takes a subarray of the grid
  # representing a single row and packs its numbers into a string.
  # The join() method joins the elements of the array into a single
  # string with newlines between them. Finally, the tr() method
  # translates the binary string representation into ASCII digits.
  (0..8).collect{|r| @grid[r*9,9].pack('c9')}.join("\n").tr(BIN,ASCII)
end

# Return a duplicate of this Puzzle object.
# This method overrides Object.dup to copy the @grid array.
```

```
def dup
  copy = super          # Make a shallow copy by calling Object.dup
  @grid = @grid.dup     # Make a new copy of the internal data
  copy                  # Return the copied object
end

# We override the array access operator to allow access to the
# individual cells of a puzzle. Puzzles are two-dimensional,
# and must be indexed with row and column coordinates.
def [](row, col)
  # Convert two-dimensional (row,col) coordinates into a one-dimensional
  # array index and get and return the cell value at that index
  @grid[row*9 + col]
end

# This method allows the array access operator to be used on the
# lefthand side of an assignment operation. It sets the value of
# the cell at (row, col) to newvalue.
def []=(row, col, newvalue)
  # Raise an exception unless the new value is in the range 0 to 9.
  unless (0..9).include? newvalue
    raise Invalid, "illegal cell value"
  end
  # Set the appropriate element of the internal array to the value.
  @grid[row*9 + col] = newvalue
end

# This array maps from one-dimensional grid index to box number.
# It is used in the method below. The name BoxOfIndex begins with a
# capital letter, so this is a constant. Also, the array has been
# frozen, so it cannot be modified.
BoxOfIndex = [
  0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,
  3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,
  6,6,6,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8
].freeze

# This method defines a custom looping construct (an "iterator") for
# Sudoku puzzles.  For each cell whose value is unknown, this method
# passes ("yields") the row number, column number, and box number to the
# block associated with this iterator.
def each_unknown
  0.upto 8 do |row|             # For each row
    0.upto 8 do |col|           # For each column
      index = row*9+col         # Cell index for (row,col)
      next if @grid[index] != 0 # Move on if we know the cell's value
      box = BoxOfIndex[index]   # Figure out the box for this cell
      yield row, col, box       # Invoke the associated block
    end
  end
end

# Returns true if any row, column, or box has duplicates.
# Otherwise returns false. Duplicates in rows, columns, or boxes are not
# allowed in Sudoku, so a return value of true means an invalid puzzle.
```

```
  def has_duplicates?
    # uniq! returns nil if all the elements in an array are unique.
    # So if uniq! returns something then the board has duplicates.
    0.upto(8) {|row| return true if rowdigits(row).uniq! }
    0.upto(8) {|col| return true if coldigits(col).uniq! }
    0.upto(8) {|box| return true if boxdigits(box).uniq! }

    false  # If all the tests have passed, then the board has no duplicates
  end

  # This array holds a set of all Sudoku digits. Used below.
  AllDigits = [1, 2, 3, 4, 5, 6, 7, 8, 9].freeze

  # Return an array of all values that could be placed in the cell
  # at (row,col) without creating a duplicate in the row, column, or box.
  # Note that the + operator on arrays does concatenation but that the -
  # operator performs a set difference operation.
  def possible(row, col, box)
    AllDigits - (rowdigits(row) + coldigits(col) + boxdigits(box))
  end

  private  # All methods after this line are private to the class

  # Return an array of all known values in the specified row.
  def rowdigits(row)
    # Extract the subarray that represents the row and remove all zeros.
    # Array subtraction is set difference, with duplicate removal.
    @grid[row*9,9] - [0]
  end

  # Return an array of all known values in the specified column.
  def coldigits(col)
    result = []                  # Start with an empty array
    col.step(80, 9) {|i|         # Loop from col by nines up to 80
      v = @grid[i]               # Get value of cell at that index
      result << v if (v != 0)    # Add it to the array if non-zero
    }
    result                       # Return the array
  end

  # Map box number to the index of the upper-left corner of the box.
  BoxToIndex = [0, 3, 6, 27, 30, 33, 54, 57, 60].freeze

  # Return an array of all the known values in the specified box.
  def boxdigits(b)
    # Convert box number to index of upper-left corner of the box.
    i = BoxToIndex[b]
    # Return an array of values, with 0 elements removed.
    [
      @grid[i],    @grid[i+1],  @grid[i+2],
      @grid[i+9],  @grid[i+10], @grid[i+11],
      @grid[i+18], @grid[i+19], @grid[i+20]
    ] - [0]
  end
end  # This is the end of the Puzzle class
```

```
# An exception of this class indicates invalid input,
class Invalid < StandardError
end

# An exception of this class indicates that a puzzle is over-constrained
# and that no solution is possible.
class Impossible < StandardError
end

#
# This method scans a Puzzle, looking for unknown cells that have only
# a single possible value. If it finds any, it sets their value. Since
# setting a cell alters the possible values for other cells, it
# continues scanning until it has scanned the entire puzzle without
# finding any cells whose value it can set.
#
# This method returns three values. If it solves the puzzle, all three
# values are nil. Otherwise, the first two values returned are the row and
# column of a cell whose value is still unknown. The third value is the
# set of values possible at that row and column. This is a minimal set of
# possible values: there is no unknown cell in the puzzle that has fewer
# possible values. This complex return value enables a useful heuristic
# in the solve() method: that method can guess at values for cells where
# the guess is most likely to be correct.
#
# This method raises Impossible if it finds a cell for which there are
# no possible values. This can happen if the puzzle is over-constrained,
# or if the solve() method below has made an incorrect guess.
#
# This method mutates the specified Puzzle object in place.
# If has_duplicates? is false on entry, then it will be false on exit.
#
def Sudoku.scan(puzzle)
  unchanged = false  # This is our loop variable

  # Loop until we've scanned the whole board without making a change.
  until unchanged
    unchanged = true          # Assume no cells will be changed this time
    rmin,cmin,pmin = nil      # Track cell with minimal possible set
    min = 10                  # More than the maximal number of possibilities

    # Loop through cells whose value is unknown.
    puzzle.each_unknown do |row, col, box|
      # Find the set of values that could go in this cell
      p = puzzle.possible(row, col, box)

      # Branch based on the size of the set p.
      # We care about 3 cases: p.size==0, p.size==1, and p.size > 1.
      case p.size
      when 0  # No possible values means the puzzle is over-constrained
        raise Impossible
      when 1  # We've found a unique value, so set it in the grid
        puzzle[row,col] = p[0] # Set that position on the grid to the value
        unchanged = false      # Note that we've made a change
```

```
      else    # For any other number of possibilities
        # Keep track of the smallest set of possibilities.
        # But don't bother if we're going to repeat this loop.
        if unchanged && p.size < min
          min = p.size                    # Current smallest size
          rmin, cmin, pmin = row, col, p  # Note parallel assignment
        end
      end
    end
  end

  # Return the cell with the minimal set of possibilities.
  # Note multiple return values.
  return rmin, cmin, pmin
end

# Solve a Sudoku puzzle using simple logic, if possible, but fall back
# on brute-force when necessary. This is a recursive method. It either
# returns a solution or raises an exception. The solution is returned
# as a new Puzzle object with no unknown cells. This method does not
# modify the Puzzle it is passed. Note that this method cannot detect
# an under-constrained puzzle.
def Sudoku.solve(puzzle)
  # Make a private copy of the puzzle that we can modify.
  puzzle = puzzle.dup

  # Use logic to fill in as much of the puzzle as we can.
  # This method mutates the puzzle we give it, but always leaves it valid.
  # It returns a row, a column, and set of possible values at that cell.
  # Note parallel assignment of these return values to three variables.
  r,c,p = scan(puzzle)

  # If we solved it with logic, return the solved puzzle.
  return puzzle if r == nil

  # Otherwise, try each of the values in p for cell [r,c].
  # Since we're picking from a set of possible values, the guess leaves
  # the puzzle in a valid state. The guess will either lead to a solution
  # or to an impossible puzzle. We'll know we have an impossible
  # puzzle if a recursive call to scan throws an exception. If this happens
  # we need to try another guess, or re-raise an exception if we've tried
  # all the options we've got.
  p.each do |guess|          # For each value in the set of possible values
    puzzle[r,c] = guess      # Guess the value

    begin
      # Now try (recursively) to solve the modified puzzle.
      # This recursive invocation will call scan() again to apply logic
      # to the modified board, and will then guess another cell if needed.
      # Remember that solve() will either return a valid solution or
      # raise an exception.
      return solve(puzzle)   # If it returns, we just return the solution
    rescue Impossible
      next                   # If it raises an exception, try the next guess
    end
```

```
      end

      # If we get here, then none of our guesses worked out
      # so we must have guessed wrong sometime earlier.
      raise Impossible
    end
  end
```


示例 1-1 的长度为 345 行。因为这个例子是为了这个介绍性的章节而编写的，所以它包含了特别详尽的注释。除去注释和空行之后的代码长度仅为 129 行，是一个非常好的面向对象的 Suduko 解答，而且该解答不依赖于简单的暴力破解算法。我们希望这个例子能展现 Ruby 的强大和表现力。

Ruby 程序的结构和运行

The Structure and Execution of Ruby Programs

本章解释了 Ruby 程序的结构。首先讲解词法结构、涵盖标记（token）及组成它们的字符；接下来，讲解 Ruby 程序的句法结构，阐明一串标记是如何构成表达式、控制结构、类及方法之类的结构的；最后，本章描述 Ruby 的代码文件，解释 Ruby 程序是如何在多个代码文件之间进行分布的，以及 Ruby 解释器是如何执行一个 Ruby 代码文件的。

2.1 词法结构
Lexical Structure

Ruby 解释器会把一个程序解析为一个**标记**序列，标记包括注释、字面量、标点符号、标识符及关键字。本节将介绍这些标记，并且还包含两方面的重要信息，即组成这些标记的字符和分隔这些标记的空白符。

2.1.1 注释
Comments

Ruby 中的注释以#字符开头并持续到该行结束，Ruby 解释器将忽略#字符及其后的任何文本（但是并不忽略换行符。因为换行符是一个有意义的空白符，也许会作为语句的终结符）。如果#字符出现在一个字符串或正则表达式字面量里（请参见第 3 章），那么它将作为此字符串或正则表达式的一部分，而非引入一段注释：

```
# This entire line is a comment
x = "#This is a string"                # And this is a comment
y = /#This is a regular expression/   # Here's another comment
```

通常情况下，通过在每一行前添加一个#字符来表示多行注释：

```
#
# This class represents a Complex number
# Despite its name, it is not complex at all.
#
```

请注意 Ruby 中没有类似于/*...*/的 C 风格注释，你无法在一行代码中嵌入注释。

2.1.1.1 嵌入式文档

Ruby 支持另外一种称为**嵌入式文档**（*embedded document*）的多行注释风格。这种注释以一个“=begin”开头，并以一个“=end”结尾（“=end”所在的那一行也包括在内）。任何出现在“begin”之后的文本都会被作为注释而忽略掉，前提是这些文本和“begin”之间至少有一个空格作为分界符。位于“end”之后的文本也遵循同样的规则。

利用嵌入式文档，可以轻松地注释掉大段代码，而无需在每行代码前添加#字符：

```
=begin Someone needs to fix the broken code below!
   Any code here is commented out
=end
```


请注意，“=”必须作为该行的第一个字符，这样才能使嵌入式文档生效：

```
# =begin This used to begin a comment. Now it is itself commented out!
   The code that goes here is no longer commented out
# =end
```

正如其名字所暗示的那样，嵌入式文档既可用于在程序中插入大段的文档，又可以在 Ruby 程序中嵌入其他语言编写的源代码（比如 HTML 或 SQL）。嵌入式文档通常会被一些后期处理工具使用，在“=begin”之后，通常跟着一个标识符来表明希望用哪个工具来处理该注释。

2.1.1.2 文档化的注释

在 Ruby 程序里的方法、类及模块定义之前，可以包含嵌入式的 API 文档，这些文档是一些遵循特殊格式的注释。可以利用之前在第 1.2.4 节里介绍的 *ri* 工具来浏览这些文档。*rdoc* 工具能够从 Ruby 代码中提取这些文档化的注释，然后要么将其格式化为 HTML，要么格式化成 *ri* 工具能显示的格式。对 *rdoc* 工具的详细描述超出了本书的范围，详情请参见 Ruby 源代码里的 *lib/rdoc/README* 文件。

文档化的注释必须出现在那些模块、类及方法之前，从而文档化它们的 API。这些注释的写法有两种，通常情况下，你可以采用以#字符开头的多行注释的方式来编写，此外，你还可以按照以“=begin rdoc”开头的嵌入式文档的方式来编写。（如果你遗漏了“rdoc”，那么 *rdoc* 工具将不会处理这些注释。）

下面的例子展示了 Ruby 文档化的注释所采用的标记语法里，最为重要的一些格式化元素。在之前提到的那个 *README* 文件里，你可以找到对这些语法的详细描述。

```
#
# Rdoc comments use a simple markup grammar like those used in wikis.
#
# Separate paragraphs with a blank line.
#
# = Headings
#
# Headings begin with an equals sign
#
# == Sub-Headings
# The line above produces a subheading.
# === Sub-Sub-Heading
# And so on.
#
# = Examples
```

```
#
#   Indented lines are displayed verbatim in code font.
#     Be careful not to indent your headings and lists, though.
#
# = Lists and Fonts
#
# List items begin with * or -. Indicate fonts with punctuation or HTML:
# * _italic_ or <i>multi-word italic</i>
# * *bold* or <b>multi-word bold</b>
# * +code+ or <tt>multi-word code</tt>
#
# 1. Numbered lists begin with numbers.
# 99. Any number will do; they don't have to be sequential.
# 1. There is no way to do nested lists.
#
# The terms of a description list are bracketed:
# [item 1]  This is a description of item 1
# [item 2]  This is a description of item 2
#
```

2.1.2 字面量
Literals

字面量就是那些直接出现在Ruby源代码里的值。字面量包括数字、文本字符串及正则表达式（其他字面量，比如数组和哈希值，是一些更为复杂的表达式，而非单个标记）。Ruby中关于数字和字符串字面量的语法其实相当复杂，所以直到第3章才会展现其细节。此时只要用一个例子对Ruby字面量进行简单说明即可：

```
1                   # An integer literal
1.0                 # A floating-point literal
'one'               # A string literal
"two"               # Another string literal
/three/             # A regular expression literal
```

2.1.3 标点符号
Punctuation

标点符号字符在Ruby中应用得很广泛，大多数Ruby操作符都是用标点符号字符来表示的，比如用“+”字符表示相加操作、用“*”字符表示相乘操作，以及用“||”字符表示“布尔或”操作。完整的Ruby操作符清单可以参见第4.6节。标点符号字符还有一些用途，可以用于字符串、正则表达式、数组及哈希字面量的分界符；还可以用于分组并分隔多个表达式、方法参数及数组索引。在Ruby的语法里随处可见对标点符号字符的其他各式各样的用法。

2.1.4 标识符
Identifiers

一个**标识符**就是一个名字。Ruby利用标识符对变量、类及方法之类的东西进行命名。Ruby

标识符由字母、数字和下划线字符组成，但是不能以数字开头。标识符不包含空白符或非打印字符，此外，除了下面即将提到的情况之外，标识符也不包含标点符号。

以从 A 到 Z 这 26 个大写字母开头的标识符是常量。如果你试图修改一个这样的标识符，Ruby 解释器将发出警告（但不是错误）。类和模块名则必须以大写字母开头，否则会报错。下面列出一些标识符：

```
i
x2
old_value
_internal     # Identifiers may begin with underscores
PI            # Constant
```

按照惯例，非常量的多字节标识符以下划线分隔多个字节，比如“like_this”；而多字节的常量则以“LikeThis”或“LIKE_THIS”的形式表示。

2.1.4.1 大小写敏感性

Ruby 是一种大小写敏感的语言，小写字母和大写字母是截然不同的。比如，关键字“end”和关键字“END”完全是两码事。

2.1.4.2 标识符中的 Unicode 字符

对于 ASCII 字符集，Ruby 标识符的构成规则定义了哪些字符是有效的，哪些是无效的。此外，总的来讲，ASCII 字符集之外的所有字符都是构成 Ruby 标识符的有效字符，包括那些看起来是标点符号的字符。比方说，在一个采用 UTF-8 编码的文件里，下面的 Ruby 代码是有效的：

```
def ×(x,y)   # The name of this method is the Unicode multiplication sign
  x*y        # The body of this method multiplies its arguments
end
```

与此类似，一个日本程序员可以在他的标识符里包含日本汉字字符，只要他的程序编码是 SJIS 或 EUC 即可。请参见第 2.4.1 节获取更多在编写 Ruby 程序时，采用非 ASCII 编码的信息。

关于标识符构成的规则是基于 ASCII 字符来制定的，但是这些规则并不适用于那些 ASCII 字符集之外的字符。比如，一个标识符是不能以 ASCII 数字开头的，但是它却可以以一个非拉丁语系里的数字开头。与此类似，一个标识符如果以 ASCII 字符集里的大写字母开头，那么它将被认为是一个常量。但是，在那些非 ASCII 字符集里，即便是以大写字符开头的也不一定是常量，比如 Å 这个标识符就不是一个常量。

仅当两个标识符均由相同的字节序列表示时，才能认为它们是相等的。在以 Unicode 为代表的一些字符集里，对于同一个字符可能存在多个码点（codepoint）（译注 1）。在 Ruby 语言里，并没有对 Unicode 进行规范化，即使两个字符有同样的含义，或者有同样的字体字形，但是如果它们的码点不一样，就会被认为是两个不同的字符。

译注 1：即不同的编码可以表示同一个字符。

2.1.4.3 标识符里的标点符号

标点符号可以出现在 Ruby 标识符的开始或结尾，它们具有如下含义：

$ 全局变量以美元符号开头。沿袭 Perl 语言的做法，Ruby 也定义了一些包含其他标点符号的全局变量，比如$_ and $-K。请参见第 10 章获得这些特殊全局变量的列表。

@ 实例变量以一个@符号开头，而类变量则以两个@符号开头。本书将在第 7 章对实例变量和类变量进行讲解。

? 作为一个有用的惯例，那些返回布尔值的方法通常都有一个以问号结尾的名字。

! 有些方法的名字以感叹号结尾，这是在提醒你使用这些方法时要小心。这个命名惯例通常是为了对两种方法进行区分：以感叹号结尾的方法通常会改变调用它们的对象，不以感叹号结尾的方法则不会修改调用它们的对象，而是修改原始对象的一个拷贝并返回。

= 如果方法名是以等号结尾的，那么在调用此方法时可以省略此等号。这种方法通常被置于赋值操作符的左侧（关于这个主题，你可以在第 4.5.3 节和第 7.1.5 节找到更多内容）。

下面是一些在开头或结尾包含标点符号的标识符：

```
$files          # A global variable
@data           # An instance variable
@@counter       # A class variable
empty?          # A Boolean-valued method or predicate
sort!           # An in-place alternative to the regular sort method
timeout=        # A method invoked by assignment
```

许多 Ruby 操作符都被实现为方法，因此类可以按照它们的需要来重定义这些操作符，这样一来，我们就可以将特定的操作符作为方法名来使用。在这种情况下，标点符号字符或操作符字符将被当成标识符来处理，而不是操作符。请参见第 4.6 节获取更多有关 Ruby 操作符的信息。

2.1.5 关键字
Keywords

下面的关键字对于 Ruby 来说具有特殊的意义，它们被 Ruby 的解析器（parser）进行特殊处理：

```
__LINE__        case        ensure      not         then
__ENCODING__    class       false       or          true
__FILE__        def         for         redo        undef
BEGIN           defined?    if          rescue      unless
END             do          in          retry       until
alias           else        module      return      when
and             elsif       next        self        while
begin           end         nil         super       yield
break
```

除了上述关键字外，下面还列有三个类似关键字的标记。当它们出现在一行代码的开头时，Ruby 解析器会对其另眼相看。

```
=begin    =end      __END__
```

如我们之前所见的，出现在一行开头的=begin 和=end 将区隔出一块多行注释。如果__END__单独出现在一行（没有任何前置或后置的空白），那么它将标示程序的结束（同时也是数据区的开始）。

在大多数语言里，这些关键字被称为“保留字”，不能被作为标识符使用。但是 Ruby 解释器非常灵活，如果你在这些关键字前面分别加上@、@@或$，就可以分别把它们作为实例变量名、类变量名或全局变量名来使用。此外，你还可以将这些关键字用作方法名，但是这样的方法名不能单独出现在程序中，必须通过一个对象来显式地调用它们。请注意，将这些关键字作为标识符将会产生令人迷惑的代码，把它们作为保留字来看待是非常好的实践。

Ruby 语言的许多重要特性实际上是用 Kernel、Module、Class 及 Object 这几个类的方法来实现的，因此，将下面的标识符也作为保留字来处理，是不错的实践。

```
# These are methods that appear to be statements or keywords
at_exit          catch           private          require
attr             include         proc             throw
attr_accessor    lambda          protected
attr_reader      load            public
attr_writer      loop            raise

# These are commonly used global functions
Array            chomp!          gsub!            select
Float            chop            iterator?        sleep
Integer          chop!           load             split
String           eval            open             sprintf
URI              exec            p                srand
abort            exit            print            sub
autoload         exit!           printf           sub!
autoload?        fail            putc             syscall
binding          fork            puts             system
block_given?     format          rand             test
callcc           getc            readline         trap
caller           gets            readlines        warn
chomp            gsub            scan

# These are commonly used object methods
allocate         freeze          kind_of?         superclass
clone            frozen?         method           taint
display          hash            methods          tainted?
dup              id              new              to_a
enum_for         inherited       nil?             to_enum
eql?             inspect         object_id        to_s
equal?           instance_of?    respond_to?      untaint
extend           is_a?           send
```

2.1.6 空白符

Whitespace

空白符包括空格符、制表符和换行符，它们本身并不是标记，而是用于分隔不同的标记，以免这些标记合并成一个标记。除了分隔标记这一基本功能之外，大多数的空白符仅被用于将代码格式化成易于阅读和理解的形式，而被 Ruby 解释器忽略掉了。然而，并非所有的空白符都被忽略了。有一些空白符是必须的，还有一些实际上是被禁止的。Ruby 的语法极富表现力但也很复杂，有时候插入或删除空白符将导致程序的语义发生变化。虽然这样的情形并不多见，但还是很有必要了解它们。

2.1.6.1 作为语句终结符的换行符

将换行符作为语句终结符是空白符依赖性（whitespace dependency）最常见的形式。在类似 C 和 Java 的语言里，每个语句都必须以分号结尾。在 Ruby 中，你也可以用分号来终结语句，但这并不是必须的。只有当你试图在一行代码里面放置多条语句时，才须要使用分号分隔它们。除此之外，依照惯例分号都是被省略掉的。

在没有显式的分号帮忙的情况下，Ruby 解释器就必须靠它自己来找出语句是在何处结尾的。如果一行 Ruby 代码是一个句法完整的语句，那么 Ruby 就将该行结尾的换行符看成是此语句的终结符，否则 Ruby 将继续解析下一行的语句，直到得到一个完整的语句。（本节稍后将说明，在 Ruby 1.9 里存在一个例外的情况。）

如果编写得当，即便所有的语句都位于同一行也不会带来什么问题。但是如果语句被分布在多行，那么你就要小心谨慎地分行，以免 Ruby 解释器误解了一行代码的本意。这又是所谓的空白符依赖性在作怪：你的程序将依据你插入换行符位置的不同而表现出不同的行为。比如，下面的代码将 x 和 y 的和赋给 total：

```
total = x +     # Incomplete expression, parsing continues
  y
```

但是下面的代码将 x 赋给 total，然后再对 y 求值：

```
total = x     # This is a complete expression
  + y         # A useless but complete expression
```

作为另一个例子，让我们来看看 return 和 break 语句。这些语句可以后接一个可选的表达式，该表达式将提供返回值。如果在这些关键字和表达式之间插入一个换行符，那么这些语句将在关键字之后立即结束，后续的表达式不会被作为这些语句的一部分来求值（译注 2）。

下列情形下，你可以放心地插入一个换行符而不必担心你的语句被提前结束。在一个操作符之后插入换行符（译注 3）。在方法调用的句点之后插入换行符（译注 4）。在数组或哈希表字面量里，用于分隔各元素的逗号之后（译注 5）。

你可以用一个“\”来对换行符进行转义，这样就可以避免 Ruby 自动终结该语句了：

译注 2：这很可能和你想表达的意思不一致。

译注 3：比如语句“puts x + y”的“+”之后。

译注 4：比如语句“Foo.new.say_hello”中的任意一个句点之后。

译注 5：比如语句“[1,2,3]”或“{ x=>1,y=>1 }”中的任意一个逗号之后。

```
var total = first_long_variable_name + second_long_variable_name \
  + third_long_variable_name # Note no statement terminator above
```

在 Ruby 1.9 里，关于语句终结符的规则发生了一点小变化：如果一行代码的第一个非空白的字符是一个句点，那么这一行将被当成是上一行的延续，而且在该行语句之前的那个换行符将不被当作语句终结符。这种以句点开头的行对于那些有时和“流式 API (fluent APIs)”配合使用的方法调用链非常有用，其中的每个方法调用都会返回一个对象，在此对象的基础上后续的方法调用得以延续。比如：

```
animals = Array.new
  .push("dog")   # Does not work in Ruby 1.8
  .push("cow")
  .push("cat")
  .sort
```

2.1.6.2 空格符与方法调用

Ruby 的语法允许在特定环境下与方法调用相关的圆括号可以被省略，这使 Ruby 的方法使用起来就好像它们是语句一样。这也是 Ruby 优雅性的重要体现。然而，不幸的是，它也带来了另一个有危害的空白符依赖性。比较下面的两行代码，它们之间的差异仅仅在于一个空格符：

```
f(3+2)+1
f (3+2)+1
```

第一行代码将 5 传递给方法 f，然后在结果上加 1。由于第二行代码在方法名之后出现了一个空格符，所以 Ruby 就会假设方法调用的圆括号被省略了。在空格符之后出现的圆括号将括起子表达式 3+2，而整个表达式 (3+2)+1 将被用作方法的参数。如果在执行代码时使用 -w 参数开启了警告，Ruby 将在它碰到具有二义性的代码时发出警告。

对于这种空白符依赖性问题的解决办法很直接：

- 永远不要在方法名和其后的左圆括号之间留白；
- 如果一个方法的第一个参数以圆括号开头，那么在此方法的调用中，请一直使用圆括号，比如 f((3+2)+1)；
- 请一直使用 Ruby 解释器的 -w 选项，这样它就会在你忘记了上述规则时发出警告。

2.2 句法结构
Syntactic Structure

到目前为止，我们讨论了 Ruby 程序里的标记及构成它们的字符，现在我们将简要地描述一下这些词法标记是如何组成更大的句法结构的。本节将从最简单的表达式到最大型的模块来描述 Ruby 程序的句法。实际上，本节为后续章节提供了一份路线图。

Ruby 中最基本的句法单元就是**表达式**。Ruby 解释器将**评估**表达式并且产生值。最简单的表达式就是**初级表达式**（*primary expressions*）。它们直接代表值。本章之前描述的数字和字符串字面量就是初级表达式。其他的初级表达式包括特定的关键字，比如 `true`、`false`、`nil` 及 `self`。对变量的引用也是初级表达式，对它们进行求值的结果就是其指向的变量的值。

可以编写复合表达式来代表更复杂的值。

```
[1,2,3]                    # An Array literal
{1=>"one", 2=>"two"}       # A Hash literal
1..3                       # A Range literal
```

操作符用于对多个值进行计算，所以我们可以通过操作符来组合更简单的子表达式构成复合表达式（译注 6）：

```
1            # A primary expression
x            # Another primary expression
x = 1        # An assignment expression
x = x + 1    # An expression with two operators
```

第 4 章涵盖了有关操作符和表达式的内容，包括变量和赋值表达式。

表达式还可以和 Ruby 的关键字联合起来构成**语句**。比如用于执行条件代码的 `if` 语句，以及用于执行循环代码的 `while` 语句：

```
if x < 10 then        # If this expression is true
  x = x + 1           # Then execute this statement
end                   # Marks the end of the conditional

while x < 10 do       # While this expression is true...
  print x             # Execute this statement
  x = x + 1           # Then execute this statement
end                   # Marks the end of the loop
```

在 Ruby 里，这些语句从技术上来看都是表达式，但是在那些影响程序控制流的表达式和那些不影响控制流的表达式之间做出区分还是有用的。第 5 章解释了 Ruby 的控制结构。

几乎在最简单的程序里，我们也要将表达式和语句组织成参数化的单元，这样它们就能被重复执行而且能够操作不同的输入。你也许见过将这些参数化的单元称为函数、过程或子例程。鉴于 Ruby 是一门面向对象的语言，我们称 Ruby 中的这种参数化的单元为**方法**。在第 6 章将介绍方法，以及与其密切相关的 *proc* 和 *lambda* 结构。

最后，我们可以把设计为互操作的方法组合成类，把相互关联的类及独立于那些类的一些方法组织成**模块**。类和模块将作为第 7 章的主题。

译注 6：子表达式的求值结果就是值，可以作为操作符的操作数。

2.2.1 Ruby 当中的块结构

Block Structure in Ruby

Ruby 程序具有块状结构。模块、类、方法定义及绝大多数 Ruby 语句都包含由嵌套代码所构成的块。这些块由关键字或标点符号进行分隔，而且按照惯例，使用两个空格符的缩进。Ruby 程序包含两种块结构，其中之一被正式称为“块（block）”。这种块就是一段与迭代器（iterator）方法相关联的代码，也可以将其看成是传递给迭代器方法的代码段。

```
3.times { print "Ruby! " }
```

在上述代码里，花括号及其中的代码就是与迭代器方法调用 `3.times` 相关联的块。这种块既可以用花括号来分隔，也可以用 `do` 和 `end` 关键字来分隔。

```
1.upto(10) do |x|
  print x
end
```

通常情况下，如果块中的代码多于一行，那么就采用 `do` 和 `end` 分界符。请注意块中的代码具有两个空格符的缩进。第 5.4 节将详细介绍块。

先前我们提到 Ruby 当中有两种块结构。为了避免与上述真正意义上的块发生混淆，我们称另一种块为**体**（*body*）（然而在实际中，术语“块”常常被用于指代它们两者）。所谓的体就是一个语句列表，该列表包括类定义体、方法定义体、`while` 循环体等诸如此类的结构。Ruby 的体永远不会用花括号作为分界符，而是采用关键字。关于语句体、方法定义体、类定义体及模块定义体的各种特定语法将在第 5 章、第 6 章和第 7 章中被涵盖到。

体和块可以互相嵌套，而且典型的 Ruby 程序都包含几层嵌套代码，它们之间通过相对缩进来保持可读性。如下例所示：

```
module Stats                              # A module
  class Dataset                           # A class in the module
    def initialize(filename)              # A method in the class
      IO.foreach(filename) do |line|      # A block in the method
        if line[0,1] == "#"               # An if statement in the block
          next                            # A simple statement in the if
        end                               # End the if body
      end                                 # End the block
    end                                   # End the method body
  end                                     # End the class body
end                                       # End the module body
```

2.3 文件结构

File Structure

关于 Ruby 代码文件该如何组织只有为数不多的几条规则，这些规则并不是与 Ruby 语言自身直接相关的，而是与 Ruby 程序的部署相关的。

首先，如果一个 Ruby 程序包含有“shebang”注释，那么该注释必须是第一行。这种注释是为了指示（Unix 类）操作系统如何执行该文件的。

其次，如果一个 Ruby 程序包含一个“coding”注释（如第 2.4.1 节所述），而且不包含“shebang”注释，那么该“coding”注释就应该出现在第一行；否则，由于“shebang”注释必须出现在第一行，所以在有“shebang”注释的情况下，“coding”注释就只能出现在第二行。

再次，如果一个文件包含一行代码，该行代码仅包含一个__END__标记，而且在此标记前后均无空白符，那么 Ruby 解释器将在此停止对该文件的处理。在该文件的余下部分，可以包含任何数据，而且程序可以通过 IO 流对象 DATA 对其进行读取。（请参见第 10 章和第 9.7 节获得更多有关全局常量的信息。）

Ruby 程序不需要全部位于一个文件中，比如许多程序都会从外部代码库中加载额外的 Ruby 代码。程序使用 require 从其他文件中载入代码，require 根据搜索路径来查找特定的代码模块，并且保证不会重复载入给定的模块。详情请参见第 7.6 节。

下面的代码阐明了上述各条关于 Ruby 文件结构的规则：

```
#!/usr/bin/ruby -w            shebang comment
# -*- coding: utf-8 -*-       coding comment
require 'socket'              load networking library

  ...                         program code goes here

__END__                       mark end of code
  ...                         program data goes here
```

2.4 程序的编码
Program Encoding

在最底层，一段 Ruby 程序就是一串字符。Ruby 采用 ASCII 字符集定义它的词法规则。比方说，注释要以#字符（ASCII 码为 35）开头，空白符包括水平制表符（ASCII 9）、换行符（ASCII 10）、垂直制表符（ASCII 11）、换页符（ASCII 12）、回车符（ASCII 13）及空格符（ASCII 32）。所有的 Ruby 关键字都是用 ASCII 字符编写的，所有的操作符和其他标点符号也都取自 ASCII 字符集。

默认情况下，Ruby 解释器假定 Ruby 源代码是采用 ASCII 进行编码的。但是 Ruby 程序并不是必须要采用 ASCII 编码方式。Ruby 解释器也可以处理那些采用其他编码方式的文件，只要它们能够代表完整的 ASCII 字符集即可。Ruby 解释器必须要知道源文件所采用的编码方式，这样它才可以将这些文件里的字节解释成字符。你既可以在 Ruby 源文件里标示编码方式，也可以在调用 Ruby 解释器的时候指定编码方式。很快我们就会讲解这些做法。

事实上，Ruby 解释器对于 Ruby 程序里的字符的处理方式非常灵活。除了一些有特殊含义的字符及那些不允许出现在标识符中的字符以外，Ruby 程序可以包含其编码所允许的任何字符。我们在之前已经解释过，标识符可以包含 ASCII 字符集以外的字符。对于注释、字符串及正则表达式字面量来说也是如此，它们可以包含任何不同于分界符的字符。在采用

ASCII 编码的文件里，字符串可以包含任意的字节，包括那些不可打印的控制字符。（然而，像这样直接采用底层字节的方式是不值得推荐的；Ruby 字符串字面量支持转义序列，因此任意的字符都可以用其对应的数字编码来替代。）如果文件的编码是 UTF-8，那么注释、字符串和正则表达式就可以包含任意的 Unicode 字符。如果文件的编码是日文的 SJIS 或 EUC，那么字符串就可以包含日文字符。

2.4.1 指定程序所使用的编码
Specifying Program Encoding

默认情况下，Ruby 解释器假定程序的编码是 ASCII。在 Ruby 1.8 里，你可以使用 `-K` 命令行选项来指定一个不同的编码，采用 `-Ku` 选项来调用 Ruby 解释器，运行一个采用 UTF-8 编码的包含了 Unicode 字符的 Ruby 程序。你还可以分别采用 `-Ke` 和 `-Ks` 选项来运行采用 EUC-JP 或 SJIS 编码的包含了日文字符的程序。

Ruby 1.9 同样支持 `-K` 选项，但是已经不再是指定程序文件编码方式的优先选择了。与其让用户在运行程序的时候指定其编码，不如让程序的作者在文件开头放入一个特殊的“编码注释（coding comment）”来指定编码（注 1）。比如：

```
# coding: utf-8
```

该注释必须完全使用 ASCII 字符来编写，字符串 `coding` 后必须接一个冒号或等号，然后再接期望的编码名称（该名称不能包含空格，也不能包含除了连字符和下划线以外的标点符号）。在冒号或等号的两边允许有空白符存在，而且字符串 `coding` 可以包含任意的前缀，比如以 `en` 为前缀则拼写为 `encoding`。整个注释，包含 `coding` 字符串和编码名称，都不区分大小写，可以随意采用大写或小写字符。

文本编辑器可以通过编码注释来获取文件的编码方式。Emacs 的用户可以像下面这样编写：

```
# -*- coding: utf-8 -*-
```

vi 的用户可以像下面这样编写：

```
# vi: set fileencoding=utf-8 :
```

通常情况下，只有出现在文件第一行的编码注释才是有效的。但是当文件第一行是一个 shebang 注释时（在 Unix 类操作系统上，该注释使一个脚本成为可执行的），编码注释就可以出现在第二行。

```
#!/usr/bin/ruby -w
# coding: utf-8
```

注 1：在这一点上，Ruby 遵循了 Python 的惯例。参见 *http://www.python.org/dev/peps/pep-0263/*。

编码名称不区分大小写，可以用大写、小写或大小写混合的方式来编写。Ruby 1.9 至少支持下列编码方式：ASCII-8BIT（又被称为 BINARY），US-ASCII（7 位的 ASCII），从 ISO-8859-1 到 ISO-8859-15 的欧洲编码，Unicode 编码 UTF-8，以及日文编码 SHIFT_JIS（也被称为 SJIS）和 EUC-JP。你所使用的 Ruby 发行版也许可支持更多的编码方式。

作为一个特例，如果一个文件的头三个字节是 OxEF OxBB OxBF，那么该文件的编码方式就是 UTF-8。这三个字节被称为 BOM（Byte Order Mark 的缩写），在 UTF-8 编码的文件里是可选的。（一些 Windows 程序在保存 Unicode 文件时会将这三个字节添加到文件开头。）

在 Ruby 1.9 中，关键字__ENCODING__（在开头和结尾各有两个下划线）含有当前正在执行的代码的源编码（source encoding）。其结果值是一个 Encoding 对象。（参见第 3.2.6.2 节可以获得更多关于 Encoding 类的信息。）

2.4.2 源编码和默认外部编码
Source Encoding and Default External Encoding

在 Ruby 1.9 里，理解一个 Ruby 文件的**源编码**和一个 Ruby 进程的**默认外部编码**之间的差别是非常重要的。如我们之前所描述的：源编码会告诉 Ruby 解释器如何解读一个脚本中的字符。通常采用编码注释来设定一个文件的源编码。一个 Ruby 程序可以包含多个文件，不同的文件可以采用不同的源编码。一个文件的源编码会影响到该文件中的字符串字面量的编码方式。更多关于字符串编码的内容，请参见第 3.2.6 节。

而默认外部编码（the default external encoding）是不同的：它是当 Ruby 从文件或流中读取内容时采用的默认编码。默认外部编码对于 Ruby 进程来说是全局性的，而且不会随着文件的不同而改变。通常情况下，默认外部编码是基于你的电脑的区域设置来进行设置的。如我们稍后将要讲述的，你也可以使用命令行选项显式地指定默认外部编码。默认外部编码并不改变字符串字面量的编码方式，但是如我们将在第 9.7.2 节所见到的那样，它对于 I/O 来讲是非常重要的。

我们之前将-K 解释器选项描述成一种设定源编码的方式，事实上，这个选项真正做的事情是设置进程的默认外部编码，然后用这个编码作为默认的源编码。

在 Ruby 1.9 里，-K 选项的存在是为了与 Ruby 1.8 保持兼容，但并不是首选的设置默认外部编码的方式。两个新选项，-E 和–encoding，允许你通过其全名而不是单字符缩写的方式来指定编码。比如：

```
ruby -E utf-8             # Encoding name follows -E
ruby -Eutf-8              # The space is optional
ruby --encoding utf-8     # Encoding following --encoding with a space
ruby --encoding=utf-8     # Or use an equals sign with --encoding
```

完整的细节请参见第 10.1 节。

你可以使用 Encoding.default_external 查询默认外部编码，它返回一个 Encoding 对象。你也可以使用 Encoding.locale_charmap 获得从区域设置得到的字符编码的名称（作为字符串）。这个方法总是基于区域设置的，并且会忽略那些用于覆盖默认外部编码的命令行选项。

2.5 Ruby 程序的运行
Program Execution

Ruby 是一门脚本语言，这意味着 Ruby 程序就是待执行的语句的列表或脚本。默认情况下，语句按照其出现顺序依次执行。Ruby 的控制结构（将在第 5 章描述）可以更改这种默认的执行顺序，比如，允许语句按条件执行或重复执行。

那些习惯了传统的静态编译型语言（诸如 C 或 Java）的程序员或许会有点困惑，在 Ruby 里不存在一个作为程序运行入口点的 main 方法。Ruby 接受一个待执行语句的脚本，然后从第一行代码开始一直执行到最后一行。

（实际上，上一句的表述并不准确。Ruby 解释器首先在文件中扫描 BEGIN 语句，然后执行该语句体所包含的代码。执行完 BEGIN 语句后，Ruby 会回到第一行代码，开始顺序地执行代码。参见第 5.7 节可以获得更多关于 BEGIN 的信息。）

Ruby 和编译型语言之间的另一个差异与模块、类和方法定义有关。在编译型语言里，上述语法结构是由编译器进行处理的；但是在 Ruby 中，它们也是语句。当 Ruby 解释器遇到一个类定义时，它就执行该语句，产生一个新类。类似地，当 Ruby 解释器遇到一个方法定义时，它执行该定义，产生一个新方法。在程序的后续部分里，解释器可能会碰到并执行一个对该方法的调用，此调用将执行该方法体内的语句。

可以从命令行调用 Ruby 解释器并指定一个待执行的脚本。虽然有时候会将一些非常简单的单行脚本直接写在命令行上，但是更为常见的做法还是指定一个包含脚本的文件名，Ruby 解释器会读取文件并执行其中的脚本，它首先执行任何 BEGIN 块，然后再从文件的第一行开始执行，直到发生下面所描述的各种情况。

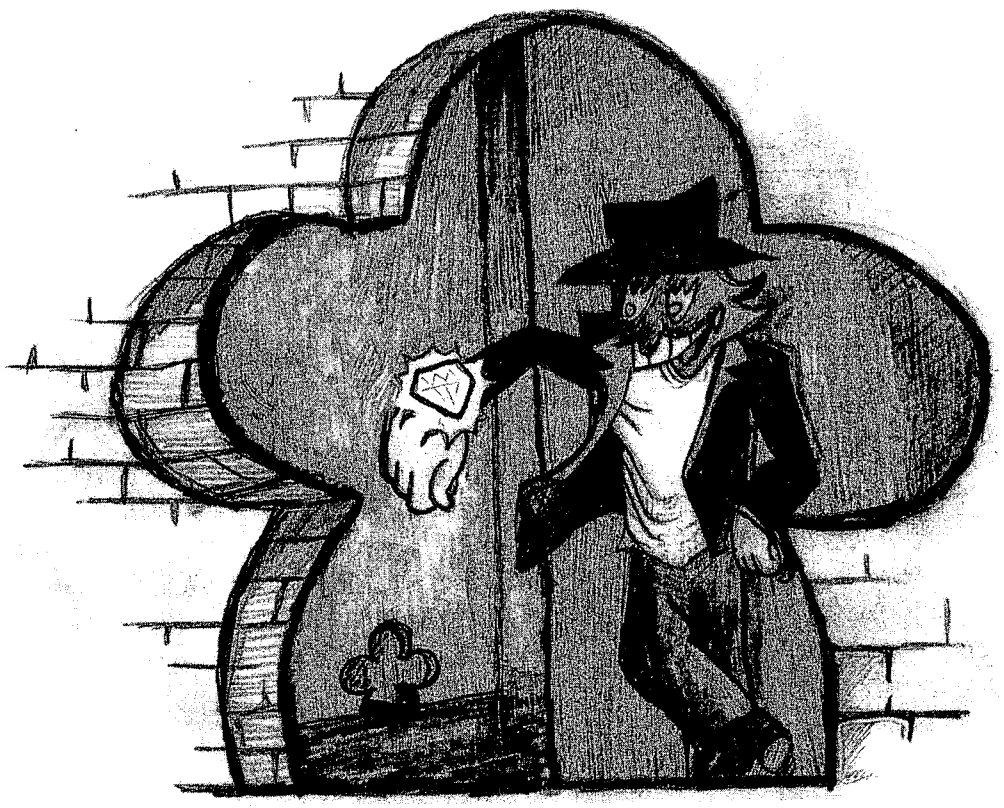
- 它执行了一个导致 Ruby 程序终结的语句。
- 它到达了文件的结尾。
- 它读入一行代码，此代码用标记__END__标示了文件的逻辑结尾。

通常情况下（除非调用了 exit! 方法），Ruby 解释器在退出之前会执行任何 END 语句，以及任何通过 at_exit 函数注册过的“关闭钩子（shutdown hook）”代码。

第 3 章

数据类型和对象

Datatypes and Objects



为了理解一门编程语言，你必须了解它可以操作什么类型的数据，以及针对这些数据都有哪些操作。本章讲述了那些被 Ruby 程序所操作的值。首先对数值和文本值进行全面的描述，然后解释数组和哈希——两种重要的数据结构，同时也是 Ruby 的基本组成部分，接下来本章将解释范围（ranges）、符号（symbols），以及 `true`、`false` 和 `nil` 这几个特殊的值。Ruby 中所有的值都是对象，本章结尾将对那些所有对象的共同特性进行详细的描述。

本章所介绍的类都是 Ruby 语言的基础数据类型。本章讲述了这些类型的基本行为，比如：程序中的字面量是如何撰写的，整数和浮点数的算术运算是如何工作的，文本数据是如何编码的，值对象是如何用于哈希键值的等。虽然本章涵盖了数字、字符串、数组及哈希这些类型，但是并不打算解释它们所包含的 API。在第 9 章我们将以举例子的方式展示这些 API，并且还会涵盖许多其他重要的（但不是基础的）类。

3.1 数字
Numbers

Ruby 包含了 5 个用来表示数字的内建类，此外标准库还包含了 3 个数值类，它们有时也会被用到。图 3-1 表示了类之间的继承关系。

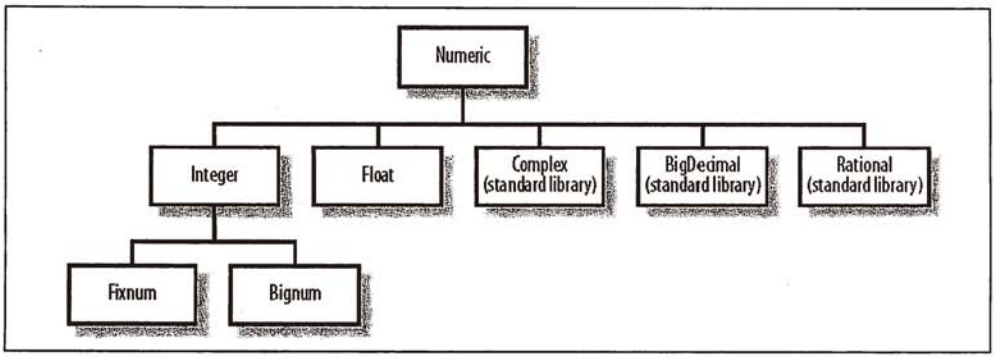


图 3-1：数值类继承关系

Ruby 中的所有数字对象都是 `Numeric` 类的实例。所有的整数都是 `Integer` 类的实例。如果一个整数值能容纳在 31 个二进制位里（在大多数实现上是如此），那么它是 `Fixnum` 类的实例，否则它就是 `Bignum` 类的实例。`Bignum` 对象用于表示任意大小的整数，如果一个操作的操作数是 `Fixnum` 对象，但是其结果超出了 `Fixnum` 的范围，那么该结果将被透明地转换成一个 `Bignum` 对象。类似地，如果一个操作的操作数是 `Bignum` 对象，但是其结果能够存放在 `Fixnum` 对象里，那么该结果将被透明地转换成一个 `Fixnum` 对象。Ruby 使用 `Float` 类来近似地表示实数，该类会利用本地平台的浮点数表示形式。

Complex、BigDecimal 及 Rational 类并非 Ruby 的内建类，但是它们作为标准库的一部分与 Ruby 一同发布。Complex 类代表了复数。BigDecimal 类表示具有任意精度的实数，使用的是十进制表示法而不是二进制表示法。Rational 类表示有理数，即两个整数相除后得到的数。

所有的数值对象都是**不可变**的，没有什么方法可以让你改变数值对象所拥有的值。如果你将一个数值对象的引用传递给一个方法，你无须担心该方法会改变该对象。Fixnum 对象最为常用，因此 Ruby 实现通常将它们作为立即值（immediate values）来处理，而不是引用。因为数字对象是不可变的，所以这两种做法之间也没什么差别。

3.1.1 整数字面量
Integer Literals

一个整数字面量就是一串数字：

```
0
123
12345678901234567890
```

如果整数值能够容纳于 Fixnum 类，那么该值是一个 Fixnum，否则它就是一个 Bignum，该类型支持任意大小的整数。可以在整数字面量里插入下划线（既不在开头也不在结尾），此特性有时被称为千分符（thousands separator）：

```
1_000_000_000     # One billion (or 1,000 million in the UK)
```

如果一个整数字面量以零开头，而且数字个数大于 1，那么它不会被视为十进制数。以 0x 或 0X 开头的整数是十六进制数，使用字母 a 到 f（或者 A 到 F）来表示数字 10 到 15。以 0b 或 0B 开头的是二进制数，只包含数字 0 和 1。以 0 开头而且后面没有其他字母的整数是八进制数，由 0 到 7 的数字构成。例如：

```
0377              # Octal representation of 255
0b1111_1111       # Binary representation of 255
0xFF              # Hexadecimal representation of 255
```

在整数字面量的前面加上一个减号就可以表示负数。字面量也可以以加号开头，但这并不会改变该字面量的含义。

3.1.2 浮点数字面量
Floating-Point Literals

一个浮点数字面量就是在一个可选的正号或负号后依次加上：一个或多个十进制数字、一个小数点（字符）及其后的一个或多个十进制数字，以及一个可选的指数部分。指数部分由字母 e 或 E 开始，后接一个可选的正号或负号，然后接一个或多个十进制数字。和整数

字面量的情况一样，在浮点数字面量里也可以使用下划线。和整数字面量的不同之处在于，浮点值的基数只能为 10。下面是一些浮点数字面量的例子：

```
0.0
-3.14
6.02e23          # This means 6.02 x 1023
1_000_000.01     # One million and a little bit more
```

Ruby 要求在小数点的前后都要有数字出现。比如说你不能写.1，而必须显示地写 0.1。为了避免 Ruby 复杂的语法所带来的二义性，这一点是非常必要的。在这方面 Ruby 和许多其他的语言都不相同。

3.1.3 Ruby 中的算术操作
Arithmetic in Ruby

所有的 Ruby 数值类型都定义了标准的+、-、*及/操作符，依次对应于加、减、乘、除 4 种操作。当一个整数结果太大而无法容纳于一个 Fixnum 时，Ruby 会自动地将其转化成一个 Bignum，这样做的结果就是，Ruby 中的整数算术操作不会像其他语言中那样产生溢出。浮点数（至少在那些采用标准 IEEE-754 浮点数表示法的平台上）会上溢至两个特殊的值，即正无穷大或负无穷大，此外还会下溢至 0。

除法操作符的行为是与操作数的类型有关的。如果两个操作数都是整数，那么此操作符进行的是截断型的整数除法（truncating-integer division）。如果任意一个操作数是 Float，那么执行的将是浮点数除法。

```
x = 5/2      # result is 2
y = 5.0/2    # result is 2.5
z = 5/2.0    # result is 2.5
```

除数为 0 的整数除法将导致一个 ZeroDivisionError 被抛出。除数为 0 的浮点数除法不会导致错误，只会返回一个名为 Infinity 的值。0.0/0.0 是一个特例。在大多数新式硬件和操作系统上，它都等于另一个特殊的浮点值，即 NaN，表示 Not-a-Number。

取模操作符（%）计算整数除法的余数：

```
x = 5%2     # result is 1
```

虽然没那么常用，但是%操作符也可以用于 Float 操作数：

```
x = 1.5%0.4   # result is 0.3
```

除法操作、取模操作和负数

当有一个操作数（但并非两个）是负数时，Ruby 的整数除法和取模操作将不同于 C、C++ 及 Java 语言的做法（但是和 Python 及 Tcl 的操作一样）。比如，-7/3 的商，可以用浮点数表示为-2.33。但是，由于整数除法的结果必须为整数，所以必须对结果进行圆整。Ruby 采取的是向负无穷大圆整，得到的结果为-3，但 C 及其相关语言的做法是向 0 圆整而得到-2。（当然，这只是一种表示结果的方式，实际上并没有进行浮点数除法。）

按照 Ruby 对整数除法的定义，可以得到一个很重要的推论，即-a/b 等于 a/-b，但可能不等于-(a/b)。

Ruby 对于取模操作符的定义也与 C 和 Java 语言的不同。在 Ruby 中，-7%3 的结果是 2，但在 C 和 Java 中的结果却是-1。由于两种操作的商不同，所以其结果的量当然也不同。此外，两个结果的符号也不同。在 Ruby 中，结果的符号始终和第二个操作数的符号保持一致。在 C 和 Java 里，结果的符号始终和第一个操作数的符号保持一致。(Ruby 还定义了一个 remainder 方法，在结果的量和符号方面，它的行为都类似于 C 的取模操作。)

Ruby 从 Fortran 语言处借鉴了**操作符来表示指数操作。指数不必是整数：

```
x**4              # This is the same thing as x*x*x*x
x**-1             # The same thing as 1/x
x**(1/3.0)        # The cube root of x
x**(1/4)          # Integer division means this is x**0, which is always 1
x**(1.0/4.0)      # This is the fourth-root of x
```

当一个表达式含有多个指数操作时，它们按照从右到左的顺序被执行，因此，4**3**3 的值与 4**9 相同，而不同于 64**2。

指数操作能导致非常大的值。请记住，整数可以变得任意大，但是 Float 对象却不能表示大于 Float::MAX 的数，因此，表达式 10**1 000 可以得到精确的整数结果，但是表达式 9.9**1 000 却会溢出到 Infinity 这个 Float 值。

Fixnum 和 Bignum 值支持标准的位操作符——~、&、|、^、>>及<<。这些操作符在 C、Java 及很多其他语言中都很常见。(细节请参见第 4.6 节。) 此外，可以像索引数组那样对整数值的单个位进行索引，索引 0 将返回最低有效位:

```
even = (x[0] == 0)  # A number is even if the least-significant bit is 0
```

3.1.4 浮点数的二进制表示和圆整错误
Binary Floating-Point and Rounding Errors

大多数计算机硬件和计算机语言（包括 Ruby)，都采用一种浮点数表现形式对实数进行近似的处理，比如 Ruby 的 Float 类就是如此。为了高效地使用硬件，大多数浮点数表现形式都是二进制的，可以精确地表达类似 1/2、1/4 及 1/1 024 的分数。不幸的是，我们更为常用的分数是 1/10、1/100、1/1 000 等（尤其是进行财务计算时)，即便像 0.1 这样简单的数，也无法用二进制的浮点数表现形式来精确地表达。

Float 对象的精度很高，可以很好地近似表示 0.1，但是无法做到完全精确，这会带来一些问题。请考虑下面这个简单的 Ruby 表达式：

```
0.4 - 0.3 == 0.1    # Evaluates to false in most implementations
```

由于圆整错误，0.4 和 0.3 的近似值之间的差值与 0.1 的近似值并不相同。这个问题并不仅限于 Ruby，C、Java、JavaScript 及所有采用 IEEE-754 浮点数的语言都存在这个问题。

对于此问题的一个解决方案就是采用十进制表示实数，而不是二进制。来自 Ruby 标准库的 BigDecimal 类就是一个这样的例子。针对 BigDecimal 对象的算术操作要比针对 Float 值的慢上许多倍，对于典型的财务计算来说，BigDecimal 的速度是足够快的，但是对于科学计算来说，它就显得力不从心了。在第 9.3.3 节有一个使用 BigDecimal 库的简短的例子。

3.2 文本
Text

Ruby 使用 String 类的对象表示文本。字符串是可变的对象，而且 String 类定义了一个强大的方法集合，用于完成提取子字符串、插入和删除文本、搜索及替换等操作。Ruby 提供了许多在程序中表示字符串字面量的方法，其中有一些可以支持一种字符串内插语法(string interpolation syntax)。这种语法非常强大，通过它可以将任意的 Ruby 表达式的值替换到字符串字面量中。接下来的一些章节将解释字符串、字符字面量及字符串操作符，第 9.1 节将涵盖完整的字符串 API。

Ruby 使用 Regexp 对象表示文本模式，并且定义了在程序中包含正则表达式字面量的语法。比如，代码/[a-z]\d+/表示在一个小写字母的后面加上一个或多个数字。虽然正则表达式是一个常用的 Ruby 特性，但是 regexps 并不像数字、字符串及数组那样属于基础的数据类型。请参见第 9.2 节获得关于正则表达式语法及 Regexp API 的文档。

Ruby 1.8 和 Ruby 1.9 中的文本

Ruby 1.8 和 Ruby 1.9 之间最大的变化在于 1.9 对于 Unicode 及其他多字节文本表示法提供了全面的内建支持。这种变化的影响是非常广泛的，对它的描述将贯穿本节，尤其是在第 3.2.6 节。

3.2.1 字符串字面量
String Literals

Ruby 提供多种将字符串直接嵌入程序中的方法。

3.2.1.1 由单引号引用的字符串字面量

被单引号（撇号字符）所引用的字符串是形式最简单的字符串字面量，单引号之间的文本就是字符串的值。

```
'This is a simple Ruby string literal'
```

如果你需要在一个由单引号引用的字符串字面量里放入一个撇号，那么请在其前面放置一个反斜线，这样 Ruby 解释器就不会把它当成字符串终结符了：

```
'Won\'t you read O\'Reilly\'s book?'
```

反斜线还可用于转义另一个反斜线，这样一来第二个反斜线就不再是转义字符了。下面是一些要用到两个反斜线的例子：

```
'This string literal ends with a single backslash: \\'
'This is a backslash-quote: \\\''
'Two backslashes: \\\\'
```

在一个由单引号引用的字符串里，如果一个反斜线后面的字符不是单引号，也不是反斜线，那么该反斜线就没有任何特殊作用，因此，大多数情况下，在字符串字面量里不需要成对地出现反斜线（尽管它们可以成对出现）（译注 1）。比如，下面的两个字符串字面量是相等的：

```
'a\b' == 'a\\b'
```

由单引号引用的字符串可以跨越多行，得到的字符串字面量里包含换行字符。无法用一个反斜线来转义行尾的换行符。

```
'This is a long string literal \
that includes a backslash and a newline'
```

如果你希望将一个由单引号引用的长字符串分布到多行里而且不引入换行符，那么只须简单地将其划分成彼此相邻的字符串字面量即可，Ruby 解释器会在分析代码的过程中把它们连接在一起。但是需要注意的是，你必须对各字面量之间的换行符进行转义（参见第 2 章），否则 Ruby 会将这样的换行符解释成语句的终结符。

```
message =
'These three literals are '\
'concatenated into one by the interpreter. '\
'The resulting string contains no newlines.'
```

3.2.1.2 由双引号引用的字符串字面量

由双引号引用的字符串字面量要比由单引号引用的字符串字面量灵活得多，由双引号引用的字符串字面量支持许多转义序列，比如代表换行的\n、代表 tab 的\t，以及代表双引号的\"，这种双引号不会终结字符串（译注 2）。

```
"\t\"This quote begins with a tab and ends with a newline\"\n"
"\\"  # A single backslash
```

在 Ruby 1.9 中，\u 这个转义序列能够将任意的 Unicode 字符嵌入由双引号引用的字符串里，这些字符是由其码点（codepoint）来指定的。这个转义序列很复杂，我们将在单独的一节

译注 1：这样做的初衷是为了用一个反斜线对另一个反斜线进行转义。
译注 2：不经转义的双引号是一种字符串字面量的终结符。

里讲述它（参见第 3.2.1.3 节）。许多其他的反斜线转义序列都比较晦涩，用于将二进制数据编码进字符串里。转义序列的完整列表请参见表 3-1。

更为强大的，由双引号引用的字符串字面量可以包含任意的 Ruby 表达式。当创建字符串的时候，字符串字面量将对表达式进行求值，然后将值转换成字符串并插回到原字符串中，用于替换原表达式的文本。在 Ruby 中，这种表达式的值对其自身的替换被称为“字符串内插”。这些表达式位于花括号中，且在花括号前有一个#字符，此外整个结构位于由双引号引用的字符串中：

```
"360 degrees=#{2*Math::PI} radians" # "360 degrees=6.28318530717959 radians"
```

当要插入到字符串字面量中的表达式只是一个对于全局、实例或类变量的引用时，花括号可以被省略：

```
$salutation = 'hello'          # Define a global variable
"#$salutation world"           # Use it in a double-quoted string
```

当不希望#字符被特殊处理的时候，你可以在它前面用一个反斜线来进行转义。只有当#字符后面是{、$或@字符时，才需要这样的转义：

```
"My phone #: 555-1234"                    # No escape needed
"Use \#{ to interpolate expressions"      # Escape #{ with backslash
```

字符串内插与 sprintf

C 程序员或许会为 Ruby 也支持 printf 和 sprintf（注 1）函数而感到高兴，它们用于将格式化之后的值内插到字符串中：

```
sprintf("pi is about %.4f", Math::PI) # Returns "pi is about 3.1416"
```

这种风格的内插方式的优势在于，格式字符串能够指定各种选项，比如可以指定一个 Float 对象将显示多少位小数。在真正的 Ruby 风格中，还有一个和 sprintf 方法等价的操作符：只要简单地在一个格式字符串和待插入其中的参数之间放置一个%操作符即可：

```
"pi is about %.4f" % Math::PI # Same as example above
"%s: %f" % ["pi", Math::PI]# Array on righthand side for multiple args
```

由双引号引用的字符串字面量可以跨越多行，除非使用一个反斜线进行转义，否则每一行的终结符将成为字符串字面量的一部分（译注 3）：

```
"This string literal
has two lines \
but is written on three"
```

你也许更偏爱将行终结符显式编码到字符串里，这样做是为了施行网络 CRLF（Carriage Return Line Feed，回车换行）行终结符，比如在 HTTP 协议里就是这样做的，做法就是将字符串字面量写在一行里，并且在行尾显式地加上\r 和\n 转义序列。请记住，相邻的字符

注 1：使用 *ri* 来深入学习：ri Kernel.sprintf。

译注 3：换言之，如果某一行的终结符被反斜线转义了，那么它就不会包含在字符串字面量中，就好像从来没有出现过该终结符一样。

串字面量会被自动地连接起来，但是如果它们位于**不同的**行，那就必须使用反斜线对它们之间的换行符进行转义：

```
"This string has three lines.\r\n" \
"It is written as three adjacent literals\r\n" \
"separated by escaped newlines\r\n"
```

表 3-1：由双引号引用的字符串内的反斜线转义序列

转义序列	含义
\ *x*	除非 *x* 是一个行终结符，或者是 abcefnrstuvxCM01234567 这些特殊字符中的一个，否则位于 *x* 之前的\是没有特殊作用的，即*x* 等价于 *x* 本身。这种语法也可以对\、#及"字符的特殊含义进行转义
\a	BEL 字符（ASCII 码为 7），会产生一次控制台铃声，等价于\C-g 或\007
\b	退格键字符（ASCII 码为 8），等价于\C-h 或\010
\e	ESC 字符（ASCII 码为 27），等价于\033
\f	换页符（ASII 码为 12），等价于\C-l 和\014
\n	换行符（ASCII 码为 10），等价于\C-j 和\012
\r	回车符（ASCII 码为 13），等价于\C-m 和\015
\s	空格符（ASCII 码为 32）
\t	水平制表符（ASCII 码为 9），等价于\C-i 和\011
\u*nnnn*	Unicode 码点 *nnnn*，每个 *n* 都是一个十六进制数。在这种形式的\u 转义序列中，所有的 4 个数字都是必须的，不能省略开头的零。在 Ruby 1.9 及其后的版本中提供此项支持
\u{十六进制数字}	由**十六进制**数字指定的 Unicode 码点。请参见正文中对此转义序列的描述，在 Ruby 1.9 及其后的版本中提供此项支持
\v	垂直制表符（ASCII 码为 11），等价于\C-k 和\013
\ *nnn*	字节 *nnn*，其中 *nnn* 是三个位于 000 和 377 之间的八进制数字
\ *nn*	等价于\0*nn*，其中 *nn* 是两个位于 00 和 77 之间的八进制数字
\ *n*	等价于\00*n*，其中 *n* 是位于 0 和 7 之间的八进制数字
\x*nn*	字节 *nn*，*nn* 是两个位于 00 和 FF 之间的十六进制数字（当 ABCDEF 作为十六进制数字时，大写或小写形式都是可以的）
\x*n*	等价于\x0*n*，其中 *n* 是一个位于 0 和 F（或 f）之间的十六进制数字
\c*x*	\C-*x* 的缩写形式
\C-*x*	表示这样一种字符，它的编码方式如下：将 *x* 的第 6 位和第 7 位归零，保留所有的高位及 5 个低位。*x* 可以是任意的字符，但是这样的序列通常用于表示从 Control-A 到 Control-Z 的控制字符（ASCII 码从 1 到 26）。鉴于 ASCII 码表的布局，你可以使用小写或大写形式的 *x*。\c*x* 是其缩写形式。*x* 可以是任何单个字符或是除了\C \u、\x、*nnn* 之外的转义字符
\M-*x*	表示这样一种字符，它的编码方式如下：将 *x* 的高位置 1。常用于表示“meta”字符，这样的字符从技术上讲并不属于 ASCII 字符集。*x* 可以是任何单个字符或是除了\M \u、\x、*nnn* 之外的转义字符。\M 可以和\C 组合使用，比如\M-\C-A

转义序列	含义
\ *eol*	一个位于行终止符前面的反斜线将转义该终结符。反斜线和行终结符都不会出现在字符串中

3.2.1.3 Unicode 转义序列

在 Ruby 1.9 中，通过\u 转义序列，由双引号引用的字符串可以包含任意的 Unicode 字符，最简单的形式是在\u 之后加上 4 个十六进制数字（字母可以是大写或小写），它们代表了位于 0000 和 FFFF 之间的 Unicode 码点。比如：

```
"\u00D7"       # => "x": leading zeros cannot be dropped
"\u20ac"       # => " ": lowercase letters are okay
```

\u 的第二种形式是：首先在它后面加一个{符号，然后接 1 到 6 个十六进制数字，最后再接一个}符号。位于花括号之间的数字可以表示位于 0 和 10FFFF 之间的任何 Unicode 码点，且开头的零可以被省略：

```
"\u{A5}"          # => "¥": same as "\u00A5"
"\u{3C0}"         # Greek lowercase pi: same as "\u03C0"
"\u{10ffff}"      # The largest Unicode codepoint
```

最后，\u{}这种形式的转义方式允许多个码点同时嵌入一个转义中，只需在花括号内放入多组由 1 到 6 个十六进制数字组成的序列即可，各组之间用一个空格或 tab 字符进行分隔，在开始花括号之后或结束花括号之前均不允许出现空格。

```
money = "\u{20AC A3 A5}"  # => " £¥"
```

花括号内的空格并不会被编码到字符串中，但是你可以使用 Unicode 码点 20 将 ASCII 空格字符编码进字符串内：

```
money = "\u{20AC 20 A3 20 A5}"  # => "  £ ¥"
```

使用\u 转义序列的字符串使用的是 Unicode UTF-8 编码方式。（参见第 3.2.6 节可了解更多关于字符串编码的信息。）

字符串内的\u 转义序列通常都是合法的，但并不总是这样。如果源文件采用了一种不同于 UTF-8 的编码方式，而且文件中的字符串包含了用该编码方式进行编码的多字节字符（指字符字面量，而不是用转义序列创建的字符），那么在该字符串内使用\u 就是非法的——因为无法让一个字符串采用两种不同的编码方式对其包含的字符进行编码。如果源编码方式是 UTF-8（参见第 2.4.1 节），那么你可以一直使用\u。此外在仅包含 ASCII 字符的字符串里，你可以始终使用\u。

\u 转义序列除了可以出现在由双引号引用的字符串里，还可以出现在其他形式的被引用文本里（很快就会讲到），比如正则表达式、字符字面量、由%-和%Q 定界的字符串、由%W 定界的数组、here documents，以及由反引号定界的命令字符串。Java 程序员需要注意的是，Ruby 中的\u 转义序列只能出现在被引用文本里，而不能出现在程序的标识符里。

3.2.1.4 字符串字面量的分界符

当一段文本里含有撇号和引号时，要将其作为由单引号或双引号引用的字符串字面量来处理，就会显得比较棘手。Ruby 支持一种更一般化的语法来引用字符串字面量（正如我们随后将要看到的，对于正则表达式和数组字面量也有这样的语法支持）。以%q开头的字符串字面量将遵循单引号引用字符串的规则，而以%Q（或%）开头的字符串字面量将遵循双引号引用字符串的规则。紧跟在 q 或 Q 之后的第一个字符是该字符串的分界符从该分界符之后的第一个字符开始，直到下一个相匹配的（未被转义的）分界符之间的内容就组成了该字符串。如果起始分界符为(、[、{或<，那么与之相匹配的分界符就是)、]、}或>，否则结束分界符就是和起始分界符同样的字符（请注意，反引号“`”和撇号“'”是不相匹配的）。下面是一些例子：

```
%q(Don't worry about escaping ' characters!)
%Q|"How are you?", he said|
%-This string literal ends with a newline\n-  # Q omitted in this one
```

如果你发现需要对分界符进行转义，那么既可以使用一个反斜线（即便在限制更为严格的%q形式下），也可以选择另一个分界符。

```
%q_This string literal contains \_underscores\__
%Q!Just use a _different_ delimiter\!!
```

如果你采用的是那种成对的分界符，并且你的字符串字面量的内容里也包含这样的分界符，那么只要这些分界符能适当地成对出现，就不必特地对其进行转义：

```
# XML uses paired angle brackets:
%<<book><title>Ruby in a Nutshell</title></book>> # This works
# Expressions use paired, nested parens:
%((1+(2*3)) = #{(1+(2*3))})                       # This works, too
%(A mismatched paren \( must be escaped)          # Escape needed here
```

3.2.1.5 Here document

对于很长的字符串字面量来说，其内容可能包含了各种字符，所以无论使用哪个字符作为分界符，都要担心字面量里的字符转义问题。Ruby 对于这个问题的解决方法是允许你指定一个任意的字符序列，并将其作为字符串的分界符。这种字面量乃是借鉴自 Unix shell 的语法，有史以来一般称其为 *here document*。（因为文档就位于源代码中，而不是在一个外部文件里。）

Here document 以<<或<<-开头，后面紧跟（为了避免与左移位操作符<<向混淆，不允许有空格）一个用于指定结尾分界符的标识符或字符串，从下一行开始直至该分界符单独出现在一行为止，其间的文本都被作为该字符串字面量的内容。比如：

```
document = <<HERE       # This is how we begin a here document
This is a string literal.
It has two lines and abruptly ends...
HERE
```

Ruby 解释器通过每次从其输入中读取一行的方式来获得一个字符串字面量的内容，但是这并不意味着<<必须是其所在行的最后的内容。事实上，在读取了一个 here document 的内容之后，Ruby 解释器会回到该 here document 的开头部分所在的行，并继续解析后面的内容。比如，下面的 Ruby 代码通过连接两个 here document 及一个普通的双引号（译注 4）引用的字符串的方式创建了一个字符串：

```
greeting = <<HERE + <<THERE + "World"
Hello
HERE
There
THERE
```

第一行的<<HERE 使解析器读取第二和第三行，而<<THERE 让解析器读取第四和五行。读取完毕，这三个字符串字面量就会被合并成为一个。

Here document 的结尾分界符必须单独出现在一行上，该分界符的后面甚至不能接注释。如果 here document 以<<开头，那么结尾分界符必须出现在一行的开头。如果 here document 以<<-开头，那么在结尾分界符之前可以出现空白。出现在 here document 的起始部分的换行符不作为字面量的一部分，但是出现在 here document 的结尾部分的换行符则包含在字面量之内，因此，除了空的 here document，每个 here document 都以一个行结束符（译注 5）作为结尾。空的 here document 与""一样：

```
empty = <<END
END
```

如果你采用一个没有被引号括起来的标识符作为分界符，就像前面的例子那样，那么 here document 就会表现得像那些被双引号引用的字符串一样，其中的反斜线用于转义，而#用于内插字符串。如果你希望尽量按照字面意义来解释被引用的字符串，不允许有任何转义字符，只需将分界符置于单引号中即可，这样做还可以让你在分界符中使用空格。

```
document = <<'THIS IS THE END, MY ONLY FRIEND, THE END'
    .
    . lots and lots of text goes here
    . with no escaping at all.
    .
THIS IS THE END, MY ONLY FRIEND, THE END
```

围绕着分界符的单引号暗示着这个字符串字面量就像一个被单引号引用的字符串一样。事实上，这种 here document 要更为严格一些，因为单引号本身并不是一个分界符，所以没有必要采用一个反斜线来对其进行转义，此外由于反斜线在这种情况下不再需要作为转义字符了，所以也没有必要采用反斜线来对其自身进行转义，因此，在这种 here document 里，反斜线也是字符串字面量的一部分。

你还可以使用一个由双引号引用的字符串字面量来作为 here document 的分界符。除了允许在分界符里出现空格，这与使用单个分界符的情形是一样的：

译注 4：原文此处为单引号，但是代码中却是双引号，所以此处翻译成双引号。
译注 5：即换行符。

```
document = <<-"# # #"    # This is the only place we can put a comment
<html><head><title>#{title}</title></head>
<body>
<h1>#{title}</h1>
#{body}
</body>
</html>
                # # #
```

值得注意的是，here document 里以#开头的注释只能出现在第一行的<<标记之后、字面量开始之前的位置。在上述代码里出现的所有#字符当中，有一个用于引导注释（译注 6），有三个用于将表达式插入到字面量中（译注 7），其余的均作为分界符。

3.2.1.6 反引号所引用的命令的执行

Ruby 支持另一种有关引号和字符串的语法。当使用反引号（即`字符，也称为反引号）来引用文本时，该文本被作为一个由双引号引用的字符串字面量来处理。该文本的值将被传递给一个特殊的名为 Kernel.`的方法，该方法将该文本的值作为一个操作系统 shell 命令来执行，并且将该命令的输出作为字符串返回。

考虑下面的 Ruby 代码：

```
`ls`
```

在一个 Unix 系统上，这 4 个字符将产生一个字符串，该字符串所代表的命令列出当前目录下的文件名。当然这具有很强的平台依赖性，粗略地讲，Windows 下的`dir`和它是等价的。

Ruby 支持一种泛型化的引用语法，可以用来替代反引号。类似于先前介绍的%Q 语法，这里采用%x（表示执行）来代替反引号：

```
%x[ls]
```

请注意，位于反引号（或者%x）中的文本将被作为一个由双引号引用的字面量来处理，因此任何 Ruby 表达式都可以被内插到该字符串中。比如：

```
if windows
  listcmd = 'dir'
else
  listcmd = 'ls'
end
listing = `#{listcmd}`
```

然而，在这种情况下，直接调用反引号方法会更加简单：

```
listing = Kernel.`(listcmd)
```

3.2.1.7 字符串字面量和可变性

Ruby 的字符串是可变的。因此，Ruby 解释器无法用同一个对象来表达两个相同的字符串字

译注 6：即第一行的#。

译注 7：即两个#{title}和一个#{body}。

面量。（如果你是一个 Java 程序员，也许你会觉得这很奇怪。）每当 Ruby 遇见一个字符串字面量时，它都会新建一个对象。如果你在一个循环体内包含了一个字面量，那么 Ruby 在每次迭代的时候都会创建一个新的对象。你可以运行下面这样的代码来向自己证明这一点：

```
10.times { puts "test".object_id }
```

为了获得更好的运行效率，你应该避免在循环中使用字符串字面量。

3.2.1.8 String.new 方法

除了前面所提到的各种创建字符串字面量的方法，你还可以通过 `String.new` 方法来创建新字符串。在没有参数的情况下，此方法会返回一个不包含字符的新字符串。如果有一个字符串作为其参数，那么该方法将会返回一个新的 `String` 对象，它含有和参数字符串一样的文本。

3.2.2 字符字面量
Character Literals

在 Ruby 程序中，你可以通过在字符前加一个问号的方式来表示单个字符构成的字面量，不需要使用任何形式的引号：

```
?A   # Character literal for the ASCII character A
?"   # Character literal for the double-quote character
??   # Character literal for the question mark character
```

虽然 Ruby 具有表达字符字面量的语法，但是它没有一个专门的类来表示单个字符，而且，Ruby 1.9 对字符字面量的解释方式也不同于 Ruby 1.8。在 Ruby 1.8 里，字符字面量被求值为该字符所对应的整数编码。比如，由于 ASCII 编码里大写 A 的值为 65，所以`?A` 就等于 65。在 Ruby 1.8 里，字符字面量的语法仅用于 ASCII 字符和单字节字符。

在 Ruby 1.9 及其后的版本里，字符就是长度为 1 的字符串，也就是说，字面量`?A` 和字面量`'A'`是一样的，而且在新代码里已经不再需要这种字符字面量语法了。在 Ruby 1.9 里，字符字面量语法可用于多字节字符，而且可以结合`\u` Unicode 转义序列来使用（但不能用于像`\u{a b c}`这样的多码点（multicodepoint）的形式）。

```
?\u20AC == ?     # => true: Ruby 1.9 only
?  == "\u20AC"   # => true
```

在之前的表 3-1 里列出的所有转义字符，均可以和字符字面量的语法结合起来使用：

```
?\t     # Character literal for the TAB character
?\C-x   # Character literal for Ctrl-X
?\111   # Literal for character whose encoding is 0111 (octal)
```

3.2.3 字符串操作符
String Operators

String 类定义了一些有用的操作符用于操作文本字符串。+操作符用于将两个字符串连接起来，并将结果作为一个新字符串对象返回：

```
planet = "Earth"
"Hello" + " " + planet    # Produces "Hello Earth"
```

Java 程序员请注意，Ruby 里的+操作符不会将其右侧操作数自动地转换成字符串，你必须亲自动手：

```
"Hello planet #" + planet_number.to_s  # to_s converts to a string
```

在 Ruby 中，采用字符串内插通常要比采用+操作符进行字符串连接要更简单一些，在字符串内插时，对 to_s 的调用是自动进行的：

```
"Hello planet ##{planet_number}"
```

<<操作符会将其第二个参数添加到第一个参数后面，这对于 C++程序员来说应该很熟悉。这个操作符不同于+，它会修改左侧的操作数，而不是创建并返回一个新对象。

```
greeting = "Hello"
greeting << " " << "World"
puts greeting   # Outputs "Hello World"
```

与+类似，<<操作符也不会对右侧操作数做任何的类型转换，但是，如果右侧操作数是一个整数，那么它将被作为一个字符编码来处理，对应的字符将被添加到左侧操作数上。在 Ruby 1.8 里，只有位于 0 和 255 之间的整数值是合法的。在 Ruby 1.9 里，可以使用任何整数，只要该整数在左侧的字符串所采用的编码方式里是一个有效的码点即可。

```
alphabet = "A"
alphabet << ?B   # Alphabet is now "AB"
alphabet << 67   # And now it is "ABC"
alphabet << 256  # Error in Ruby 1.8: codes must be >=0 and < 256
```

*操作符期望其右侧操作数是一个整数，它返回一个字符串，该字符串将依照右侧操作数所指定的次数来重复左侧的文本：

```
ellipsis = '.'*3    # Evaluates to '...'
```

如果左侧的操作数是一个字符串字面量，那么任何内插操作都只会在重复操作之前被执行一次，这意味着下面这段弄巧成拙的代码不会按照所期望的那样运行：

```
a = 0;
"#{a=a+1} " * 3   # Returns "1 1 1 ", not "1 2 3 "
```

String 类定义了所有标准的比较操作符。==和!=用于比较字符串之间的相等性和不等性。当且仅当两个字符串拥有相同的长度，而且所有字符都相等时它们才是相等的。<、<=、>及>=操作符通过比较构成字符串的字符的编码，对字符串进行比较。如果一个字符串是另

一个字符串的前缀，那么短字符串要小于长字符串。字符串之间的比较会严格地基于字符编码，不会对将要比较的字符串进行任何标准化（normalization），此外自然语言的排序规则（如果它不同于字符编码的数字序列）也会被忽略。

字符串比较是大小写敏感的（注 2）。在 ASCII 里，大写字符的编码值均小于小写字符，比如“Z” < “a”。对 ASCII 字符进行大小写不敏感的比较，既可以使用 casecmp 方法（参见第 9.1 节），也可以在比较之前先通过 downcase 或 upcase 方法将字符串转换成相同的大小写形式。（请记住，Ruby 关于大写或小写字符的理解仅限于 ASCII 字符集。）

3.2.4 访问字符和子字符串
Accessing Characters and Substrings

也许在 String 所支持的操作符中，最重要的就是方括号数组索引操作符[]，它用于提取或改变一个字符串的某些部分。该操作符非常灵活，可以用于多种类型的操作数，还可以用在赋值操作的左侧，作为改变字符串内容的一种方式。

在 Ruby 1.8 里，一个字符串就好比一个由字节或 8 位的字符编码构成的数组，该数组的长度可以通过 length 或 size 方法获得。此外你可以通过在方括号里指定字符索引的方式获取或设置该数组的元素。

```
s = 'hello';      # Ruby 1.8
s[0]              # 104: the ASCII character code for the first character 'h'
s[s.length-1]     # 111: the character code of the last character 'o'
s[-1]             # 111: another way of accessing the last character
s[-2]             # 108: the second-to-last character
s[-s.length]      # 104: another way of accessing the first character
s[s.length]       # nil: there is no character at that index
```

请注意，如果从字符串的末尾开始计算索引，那么数组索引就是负值，而且从 −1 开始，自右向左依次减小。还有一点值得注意的是，如果你试图访问一个超出了字符串末尾的字符，Ruby 并不会抛出异常，只是简单地返回 nil。

在 Ruby 1.9 中，当你索引单个字符时，会返回一个由单个字符组成的字符串，而不是该字符的编码值（译注 8）。

```
s = 'hello';      # Ruby 1.9
s[0]              # 'h': the first character of the string, as a string
s[s.length-1]     # 'o': the last character 'o'
s[-1]             # 'o': another way of accessing the last character
s[-2]             # 'l': the second-to-last character
```

注 2：在 Ruby 1.8 里，通过设置全局变量$= （该变量已是“不推荐的 (deprecated)”）为 true，可以使==、<及相关联的比较操作符进行大小写不敏感的比较。然而你不应该这样做。如果你设置了这个变量，那么即使在调用 Ruby 解释器时不使用-w，也会引发一个警告信息。此外 Ruby 1.9 已经不再支持$=/。

译注 8：Ruby 1.8 里返回的就是编码值，可对照前面的代码来看。

```
s[-s.length]      # 'h': another way of accessing the first character
s[s.length]       # nil: there is no character at that index
```

为了改变一个字符串里的单个字符，你只须简单地将方括号置于赋值表达式的左侧即可。在 Ruby 1.8 里，等号右侧可以是一个 ASCII 字符编码值，也可以是一个字符串。但是在 Ruby 1.9 里，等号右侧必须是一个字符串。无论是 Ruby 1.8 还是 Ruby 1.9，都可以在等号右侧使用字符字面量：

```
s[0] = ?H             # Replace first character with a capital H
s[-1] = ?O            # Replace last character with a capital O
s[s.length] = ?!      # ERROR! Can't assign beyond the end of the string
```

在 Ruby 1.8 和 Ruby 1.9 中，赋值语句的右侧可以是一个任意的字符串，既可以是包含多字符的字符串，也可以是空字符串：

```
s = "hello"       # Begin with a greeting
s[-1] = ""        # Delete the last character; s is now "hell"
s[-1] = "p!"      # Change new last character and add one; s is now "help!"
```

大多数情况下，你都会希望从一个字符串里获取子字符串，而不是单一字符的编码值。为了达到这个目的，你只须要在方括号里使用两个由逗号分隔的操作数即可。第一个操作数指定索引值（可能是负的），第二个操作数指定长度值（必须是非负值）。其结果是一个从指定的索引开始，包含指定数量的字符的子字符串：

```
s = "hello"
s[0,2]            # "he"
s[-1,1]           # "o": returns a string, not the character code ?o
s[0,0]            # "": a zero-length substring is always empty
s[0,10]           # "hello": returns all the characters that are available
s[s.length,1]     # "": there is an empty string immediately beyond the end
s[s.length+1,1]   # nil: it is an error to read past that
s[0,-1]           # nil: negative lengths don't make any sense
```

如果你将一个字符串赋值给按上述方式索引的子字符串，那么就相当于用该字符串替换了该子字符串。如果等号右侧是一个空字符串，那么就相当于一个删除操作，相应地，如果左侧的长度值为 0，那么就相当于一个插入操作：

```
s = "hello"
s[0,1] = "H"                  # Replace first letter with a capital letter
s[s.length,0] = " world"      # Append by assigning beyond the end of the string
s[5,0] = ","                  # Insert a comma, without deleting anything
s[5,6] = ""                   # Delete with no insertion; s == "Hellod"
```

另一种提取、插入、删除或替换子字符串的方法是通过一个 Range 对象索引一个字符串。我们将在第 3.5 节详细解释 range，目前可以先将 Range 理解为两个由若干个点分隔的整数。当使用一个 Range 对象来索引一个字符串时，其返回值就是由位于此 Range 内的字符所构成的子字符串：

```
s = "hello"
s[2..3]           # "ll": characters 2 and 3
s[-3..-1]         # "llo": negative indexes work, too
s[0..0]           # "h": this Range includes one character index
```

```
s[0...0]              # "": this Range is empty
s[2..1]               # "": this Range is also empty
s[7..10]              # nil: this Range is outside the string bounds
s[-2..-1] = "p!"          # Replacement: s becomes "help!"
s[0...0] = "Please "      # Insertion: s becomes "Please help!"
s[6..10] = ""             # Deletion: s becomes "Please!"
```

不要把这两种索引字符串的方式搞混了，它们一个是由逗号分隔的两个整数，另一个是Range 对象。虽然两者都包含了两个整数，但是存在着重要的差别：由逗号分隔的是一个索引值和一个长度值；而 Range 对象则指定了两个索引。

还可以用一个字符串来索引另一个字符串。当采用这种方式时，如果目标字符串里含有和索引字符串相匹配的子字符串，那么第一个匹配的子字符串将会被返回，否则就会返回 nil。事实上这种形式的字符串索引只有出现在一个赋值语句的左侧时才有用武之地，这种情况下，你希望将匹配到的子字符串替换成等号右侧的字符串：

```
s = "hello"           # Start with the word "hello"
while(s["l"])         # While the string contains the substring "l"
  s["l"] = "L";       # Replace first occurrence of "l" with "L"
end                   # Now we have "heLLo"
```

最后，你可以使用一个正则表达式来索引一个字符串。(本书将在第 9.2 节讲述正则表达式对象)，其结果是第一个匹配正则表达式所指定的模式的子字符串。同样地，这种形式的字符串索引出现在一个赋值操作的左侧时最为有用。

```
s[/[aeiou]/] = '*'      # Replace first vowel with an asterisk
```

3.2.5 对字符串进行迭代
Iterating Strings

在 Ruby 1.8 里，String 类定义了一个 each 方法用于按行迭代一个字符串。String 类包含了一些来自 Enumerable 模块的方法，它们可以用于处理一个字符串里的行。在 Ruby 1.8 里，你可以使用 each_byte 迭代器按字节的方式来迭代一个字符串。但是和[]操作符相比，each_byte 没什么优势，因为在 Ruby 1.8 里，对字节的随机访问和顺序访问是一样快的。

在 Ruby 1.9 里，形势大不一样了。String 类不但取消了 each 方法，而且不再包含 Enumerable 模块。Ruby 1.9 定义了三个命名清晰的字符串迭代器来取代 each 方法：each_byte 按字节对一个字符串进行顺序的迭代，与之类似，each_char 按字符进行迭代，而 each_line 则按行迭代。如果你希望以字符的方式处理一个字符串，那么使用 each_char 比使用[]操作符和字符索引的方式更加高效：

```
s = "¥1000"
s.each_char {|x| print "#{x} " }          # Prints "¥ 1 0 0 0". Ruby 1.9
0.upto(s.size-1) {|i| print "#{s[i]} "}  # Inefficient with multibyte chars
```

3.2.6 字符串编码和多字节字符
String Encodings and Multibyte Characters

Ruby 1.8 的字符串和 Ruby 1.9 的字符串有本质上的不同：

- 在 Ruby 1.8 里，字符串是一个字节序列。当字符串用于表示文本时（而不是二进制数据），该字符串里的每个字节都被假定为代表了一个 ASCII 字符。在 Ruby 1.8 里，字符串里的单个元素不是字符，而是数字，即实际的字节值或字符编码值。
- 另一方面，在 Ruby 1.9 中，字符串是实实在在的字符序列，而且那些字符不必局限于 ASCII 字符集。在 Ruby 1.9 里，字符串的单个元素是字符——表示成长度为 1 的字符串，而不是字符编码的整数值。每个字符串都有一个编码方式，它指定了字符串的字节和那些字节所代表的字符之间的对应关系。一些编码方式，比如对 Unicode 字符进行编码的 UTF-8 编码方式，采用可变数目的字节来表示每个字符，因此字节和字符之间，不再保持 1 对 1（甚至不是 2 对 1）的对应关系。

接下来的子章节解释了和 Ruby 1.9 字符串的编码相关联的特性，而且还展示了 Ruby 1.8 里如何通过 `jcode` 库对多字节字符进行初步支持。

3.2.6.1 Ruby 1.9 里的多字节字符

Ruby 1.9 重写了 `String` 类，从而可以知悉并且恰当地处理多字节字符。虽然多字节支持是 Ruby 1.9 最大的变化，但不是一个显而易见的变化：那些使用了多字节字符串的代码只是安安静静地正常运行而已。然而了解为什么这些代码能正常运行是很有意义的，本节将对此进行详细的解释。

如果一个字符串包含了多字节字符，那么其字节数量就不等于字符数量。在 Ruby 1.9 中，`length` 和 `size` 方法都返回一个字符串的字符数，而新的 `bytesize` 方法则返回字节数：

```
# -*- coding: utf-8 -*-  # Specify Unicode UTF-8 characters

# This is a string literal containing a multibyte multiplication character
s = "2×2=4"

# The string contains 6 bytes which encode 5 characters
s.length                # => 5: Characters:  '2'   '×'   '2'  '='  '4'
s.bytesize              # => 6: Bytes (hex): 32   c3 97  32   3d   34
```

请注意，上述代码的第一行是一个编码注释，它将源编码（参见第 2.4.1 节）设置成 UTF-8。如果没有这个编码注释，Ruby 解释器将不知道如何把字符串字面量里的字节序列解码成字符序列。

如果一个字符串包含了由可变数目的字节进行编码的字符，那么就不可能从字符索引直接映射到字节偏移。比如，在上述字符串里，第二个字符起始于第二个字节，但是第三个字符起始于第四个字节，这意味着你不能再假定随机访问一个字符串的字符是一个快速操

作了。当你使用[]操作符访问一个多字节字符串里的一个字符或子字符串时，Ruby 必须在内部对字符串进行顺序的迭代，从而找到所期望的字符索引。因此，总的来说，你在进行字符串处理的时候，应该尽量使用顺序算法。也就是说，尽可能使用 each_char 迭代器，而不是反复调用[]操作符。从另一方面来讲，通常也无需对此太过担心，Ruby 的实现会对那些可以优化的情况进行优化，而且如果一个字符串完全由单字节的字符组成，那么对这些字符的随机访问就会保持高效。如果你想自己进行优化，那么可以使用 ascii_only?这个实例方法来测定一个字符串是否完全由 7 位的 ASCII 字符构成。

Ruby 1.9 的 String 类定义了一个 encoding 方法，用于返回一个字符串的编码方式（返回值是一个 Encoding 对象，描述如下）：

```
# -*- coding: utf-8 -*-
s = "2×2=4"       # Note multibyte multiplication character
s.encoding        # => <Encoding: UTF-8>
t = "2+2=4"       # All characters are in the ASCII subset of UTF-8
t.encoding        # => <Encoding: ASCII-8BIT>
```

一个字符串字面量的编码方式基于其所在的文件的源编码，但是该字符串的编码也不总是和源编码保持一致的。比如，如果一个字符串字面量仅仅包含 7 位的 ASCII 字符，那么即使其源编码是 UTF-8（ASCII 的一个超集），其 encoding 方法也会返回 ASCII。这种优化使字符串方法可以知道该字符串的所有字符的长度都是一个字节。此外，如果一个字符串字面量包含了\u 转义序列，那么它的 encoding 方法将返回 UTF-8，即使其源编码是其他的编码方式。

ASCII 和 BINARY 编码

先前出现的“ASCII-8BIT”编码是 Ruby 1.9 对于 Ruby 1.8 里的遗留编码的命名，它是 ASCII 字符集，对不可打印字符和控制字符的使用没有限制。在这种编码里，一个字节总是和一个字符相等，而且字符串可以包含二进制数据或字符数据。

某些 Ruby 1.9 方法要求你指定一个编码名称（或 Encoding 对象，请参见后面），你可以用“ASCII-8BIT”来指定 ASCII 编码，或者使用它的别名“BINARY”。这也许显得有些奇怪，但却是事实：就 Ruby 而言，一串没有编码的（“BINARY”）字节和一串 8 位的 ASCII 字符是相等的，因为“BINARY”编码实际上意味着“未编码的字节”，所以你可以通过传递 nil，而不是一个编码名称或 Encoding 对象的方式来指定该编码。

Ruby 1.9 还支持一种名为“US-ASCII”的编码，它是真正的 7 位 ASCII。与 ASCII-8BIT 的区别在于，它不允许任何字节的第 8 位被设置。

某些字符串操作，比如连接和模式匹配，要求两个字符串（或一个字符串和一个正则表达式）具有相互兼容的编码。比如，如果将一个 ASCII 字符串和一个 UTF-8 字符串进行连接，那么你将得到一个 UTF-8 字符串。然而，将一个 UTF-8 字符串和一个 SJIS 字符串进行连接是不可能的：编码方式不兼容，而且会抛出一个异常。你可以使用类方法 Encoding.compatible?来测试两个字符串（或一个字符串和一个正则表达式）的编码是否兼容。如果两个实参的编码方式是兼容的，该方法就会返回两种编码方式的超集，否则返回 nil。

你可以使用 force_encoding 方法显式地设置一个字符串的编码方式。如果有一个由字节构成的字符串（或许是从一个 I/O 流里读取出来的），而且你希望告诉 Ruby 如何将它们解释为字符，那么 force_encoding 方法就可派上用场。或者，假如有一个由多字节字符构成的字符串，但是你希望通过[]来索引单个字节，这时 force_encoding 方法也能帮上忙：

```
text = stream.readline.force_encoding("utf-8")
bytes = text.dup.force_encoding(nil)   # nil encoding means binary
```

force_encoding 并没有生成其调用者的一份拷贝，而是在修改了该字符串的编码之后再将其返回。该方法没有进行任何的字符转换——字符串的底层字节没有变化，只是 Ruby 对它的解释方式发生了变化。如上述代码所示，force_encoding 的参数可以是一个编码名称或 nil，后者会指定 binary 编码。你还可以传递一个 Encoding 对象来指定编码。

force_encoding 不做任何验证工作。它不会检查字符串的底层字节是否代表了所指定的编码方式的一个有效的字符序列。你可以使用 valid_encoding?方法来进行验证。这个实例方法不接受实参，它检查是否可以使用该字符串的编码方式来将其字节序列解释成一个有效的字符序列。

```
s = "\xa4".force_encoding("utf-8")    # This is not a valid UTF-8 string
s.valid_encoding?                     # => false
```

字符串的 encode 方法（及会修改方法调用者的 encode!方法）与 force_encoding 方法截然不同。它返回一个字符串，代表了与其调用者一样的字符序列，但编码方式却不相同。为了像这样改变一个字符串的编码，encode 方法必须改变构成该字符串的底层字节。下面是一个例子：

```
# -*- coding: utf-8 -*-
euro1 = "\u20AC"                      # Start with the Unicode Euro character
puts euro1                            # Prints " "
euro1.encoding                        # => <Encoding:UTF-8>
euro1.bytesize                        # => 3

euro2 = euro1.encode("iso-8859-15")   # Transcode to Latin-15
puts euro2.inspect                    # Prints "\xA4"
euro2.encoding                        # => <Encoding:iso-8859-15>
euro2.bytesize                        # => 1
```

```
euro3 = euro2.encode("utf-8")          # Transcode back to UTF-8
euro1 == euro3                         # => true
```

你并不需要经常使用 encode 方法。最常见的改变字符串编码的情形，是在将它们写入文件或通过网络连接进行发送的时候。此外，如我们将在第 9.7.2 节所见的，Ruby 的 I/O 流类库在写出文本时，可以对它们进行自动的编码转换。

如果要在一个由一些未经编码的字节组成的字符串上调用 encode 方法，那么你须要传递两个参数给 encode，第一个参数指定你所期望的编码，而第二个参数则指定字符串的当前编码。在将该字符串的编码转换成由第一个参数所指定的编码之前，Ruby 会首先使用第二个参数所指定的编码来解释该字符串。比如：

```
# Interpret a byte as an iso-8859-15 codepoint, and transcode to UTF-8
byte = "\xA4"
char = byte.encode("utf-8", "iso-8859-15")
```

也就是说，下面两行代码具有相同的效果：

```
text = bytes.encode(to, from)
text = bytes.dup.force_encoding(from).encode(to)
```

字符编码之间的差别不仅体现在从字节到字符的映射方式上，而且体现在它们所能表示的字符集上。Unicode（也被称为 UCS，即 Unicode Character Set 的缩写）试图表示所有的字符，而那些不是基于 Unicode 的字符编码方式只能表示一个字符子集，因此，作为一个例子，不可能将所有采用 UTF-8 编码的字符串转换成采用 EUC-JP 编码，那些既不是拉丁字符又不是日文字符的 Unicode 字符将无法被转换。

如果 encode 或 encode!方法遇到了一个无法进行编码转换的字符，就会抛出异常：

```
# The iso-8859-1 encoding doesn't have a Euro sign, so this raises an exception
"\u20AC".encode("iso-8859-1")
```

3.2.6.2 Encoding 类

Ruby 1.9 中的 Encoding 类代表了一种字符编码。Encoding 对象充当了编码方式的标识符，而且只有为数不多的几个方法。name 方法返回一个编码方式的名字。to_s 方法是 name 方法的同义词。inspect 方法用比 name 方法更详尽的方式将一个 Encoding 对象转换成一个字符串。

Ruby 为其所支持的每种内建编码方式都定义了一个常量，这是为你的程序指定硬编码方式的最简单的方法。这些预定义的常量至少包含如下几个：

```
Encoding::ASCII_8BIT      # Also ::BINARY
Encoding::UTF_8           # UTF-8-encoded Unicode characters
Encoding::EUC_JP          # EUC-encoded Japanese
Encoding::SHIFT_JIS       # Japanese: also ::SJIS, ::WINDOWS_31J, ::CP932
```

请注意，由于这些都是常量，所以它们必须采取大写形式，而且编码名称里的连字符必须转换成下划线。Ruby 1.9 还支持如下编码：US-ASCII 编码、欧洲字符编码 ISO-8859-1 至 ISO-8859-15、采用大尾格式（big-endian）和小尾格式（little-endian）的 Unicode 编码 UTF-16 和 UTF-32。但是这些编码都是在需要的时候才被动态载入的，而不是内建的，而且只有在它们被使用的时候才会有对应的常量存在。

如果你有一个字符串形式的编码名称，而且希望获得对应的 Encoding 对象，那么可以使用 Encoding.find 工厂方法：

```
encoding = Encoding.find("utf-8")
```

如果需要，使用 Encoding.find 方法可以动态地载入指定的编码。Encoding.find 接受大写或小写形式的编码名称，调用一个 Encoding 对象的 name 方法可以得到该编码名称的字符串形式。

如果需要一个可用编码方式列表，你可以调用 Encoding.list 方法，它返回一个由 Encoding 对象构成的数组。Encoding.list 方法只列出内建的编码及任何已经动态载入的编码。通过调用 Encoding.find 方法可以载入新的编码，新载入的编码将在后续的 Encoding.list 调用中列出。

使用 Encoding.default_external 方法可以得到一个 Encoding 对象，它代表了默认的外部编码（请参见第 2.4.2 节）。为了获得当前区域设置（locale）的编码方式，你可以调用 Encoding.locale_ charmap 方法，并且将结果字符串传递给 Encoding.find。

大多数期望接受一个 Encoding 对象的方法可接受 nil 作为 Encoding::BINARY（比如，未编码的字节）的同义词，同时它们也可接受一个编码名称（比如 ascii、binary、utf-8、euc-jp、siis）。

如前面所见，encode 方法接受一个编码作为实参，并采用该编码对调用者字符串进行编码。该编码方式可以是 Encoding 类所支持的编码方式的一个超集，也就是说（取决于你的 Ruby 发行版所支持的编码方式），可以使用 encode 方法将一个由字符构成的字符串转换成一个由字节构成的字符串，而 Ruby 无法将该字符串里的字节解释成字符。在某些情况下，这种编码能力也许是不可或缺的。比如，当要和一个采用了某种罕见的编码方式的遗留服务器进行通信时，你就需要这种编码能力。你可以传递 Encoding 对象给 encode 方法，但是当须要使用 Encoding 类不支持的编码方式时，就必须用字符串形式的编码名称来指定编码。

3.2.6.3 Ruby 1.8 里的多字节字符

通常情况下，Ruby 1.8 将所有的字符串解释成由 8 位字节构成的序列。在标准库的 jcode 模块里，包含了对多字节字符的初步支持（使用 UTF-8，EUC 或 SJIS 编码）。

为了使用这个库，需要 require jcode 模块，然后将全局的$KCODE 变量设置成你的多字节字符所使用的编码方式（或在启动 Ruby 解释器的时候使用-K 命令行选项）。jcode 库为 String 对象定义了一个新的 jlength 方法，它将返回按字符计的字符串长度，而不是按字节计。

1.8 版里已经存在的 length 和 size 方法保持不变，它们都返回按字节计的字符串长度。

jcode 库并没有修改字符串的数组索引操作符（即[]），并且不允许随机访问构成多字节字符串的字符。但是 jcode 库定义了一个新的 each_char 迭代器，它的工作方式类似于标准的 each_byte 方法，将每个字符作为字符串（而不是字符的码点）传递给你所提供的代码块：

```
$KCODE = "u"          # Specify Unicode UTF-8, or start Ruby with -Ku option
require "jcode"       # Load multibyte character support

mb = "2\303\2272=4"   # This is "2x2=4" with a Unicode multiplication sign
mb.length             # => 6: there are 6 bytes in this string
mb.jlength            # => 5: but only 5 characters
mb.mbchar?            # => 1: position of the first multibyte char, or nil
mb.each_byte do |c|   # Iterate through the bytes of the string.
  print c, " "        # c is Fixnum
end                   # Outputs "50 195 151 50 61 52 "
mb.each_char do |c|   # Iterate through the characters of the string
  print c, " "        # c is a String with jlength 1 and variable length
end                   # Outputs "2 x 2 = 4 "
```

为了处理多字节字符串，jcode 库还修改了几个已有的 String 方法，比如 chop、delete 及 tr。

3.3 数组
Arrays

一个数组就是一系列的值，可以通过这些值在该序列中的位置或索引来访问它们。在 Ruby 中，数组的第一个值的索引为 0，size 和 length 方法返回一个数组中的元素个数，数组中最后一个元素的索引是 size-1。负索引值将从数组的末尾开始计数，因此还可以通过索引 -1 来访问数组的最后一个元素。倒数第二个元素的索引为-2，依次类推。如果你试图读取一个超出数组末尾（以一个大于或等于 size 的索引值）或数组开头（以一个小于-size 的索引值）的元素，Ruby 会返回 nil 而不会抛出异常。

Ruby 的数组是无类型且可变的，数组里的元素不必都属于同一个类型，而且它们可以随时改变。此外，数组的大小也是可以动态改变的，你可以向数组添加元素，数组会按需增长。如果你向一个超出数组末尾的元素进行赋值，数组将会动态增长，而且用 nil 来填充那些多出来的位置。（但是，向超出数组开头的元素进行赋值是错误的。）

一个数组字面量就是一个由逗号分隔的值所构成的列表，位于一对方括号内：

```
[1, 2, 3]             # An array that holds three Fixnum objects
[-10...0, 0..10,]     # An array of two ranges; trailing commas are allowed
[[1,2],[3,4],[5]]     # An array of nested arrays
```

```
[x+y, x-y, x*y]       # Array elements can be arbitrary expressions
[]                    # The empty array has size 0
```

Ruby 支持一种表达数组字面量的特殊语法。这种字面量里的元素是一些不包含空格的短字符串：

```
words = %w[this is a test]    # Same as: ['this', 'is', 'a', 'test']
open = %w| ( [ { < |          # Same as: ['(', '[', '{', '<']
white = %W(\s \t \r \n)       # Same as: ["\s", "\t", "\r", "\n"]
```

如同%q和%Q会引入一个 String 字面量一样，%w和%W会引入一个数组字面量。针对%w和%W的分界符规则与%q和%Q的相同。在分界符所包含的内容里，无须使用引号来引用数组元素字符串，而且元素之间不须要使用逗号来分隔，数组元素之间采用空白来分隔。

你还可以通过 Array.new 构造函数来创建数组，这为以编程的方式来初始化数组元素提供了机会：

```
empty = Array.new          # []: returns a new empty array
nils = Array.new(3)        # [nil, nil, nil]: new array with 3 nil elements
zeros = Array.new(4, 0)    # [0, 0, 0, 0]: new array with 4 0 elements
copy = Array.new(nils)     # Make a new copy of an existing array
count = Array.new(3) {|i| i+1}  # [1,2,3]: 3 elements computed from index
```

为了获取一个数组元素的值，你只需将整数类型的索引值放入方括号中即可：

```
a = [0, 1, 4, 9, 16]       # Array holds the squares of the indexes
a[0]                       # First element is 0
a[-1]                      # Last element is 16
a[-2]                      # Second to last element is 9
a[a.size-1]                # Another way to query the last element
a[-a.size]                 # Another way to query the first element
a[8]                       # Querying beyond the end returns nil
a[-8]                      # Querying before the start returns nil, too
```

以上所有表达式，除了最后一个，都可以用在一个赋值操作的左侧：

```
a[0] = "zero"     # a is ["zero", 1, 4, 9, 16]
a[-1] = 1..16     # a is ["zero", 1, 4, 9, 1..16]
a[8] = 64         # a is ["zero", 1, 4, 9, 1..16, nil, nil, nil, 64]
a[-9] = 81        # Error: can't assign before the start of an array
```

与字符串类似，也可以采用如下两种方式来索引数组：第一种是利用两个整数，第一个表示起始索引，第二个表示被索引的元素个数；第二种是采用一个 Range 对象。无论采用那种方式，索引表达式都会返回指定的子数组：

```
a = ('a'..'e').to_a        # Range converted to ['a', 'b', 'c', 'd', 'e']
a[0,0]                     # []: this subarray has zero elements
a[1,1]                     # ['b']: a one-element array
a[-2,2]                    # ['d','e']: the last two elements of the array
a[0..2]                    # ['a', 'b', 'c']: the first three elements
a[-2..-1]                  # ['d','e']: the last two elements of the array
a[0...-1]                  # ['a', 'b', 'c', 'd']: all but the last element
```

当上述索引方式被用于一个赋值操作的左侧时，索引所指定的子数组将被替换成等号右侧的数组元素。利用这种基本操作还可以实现插入和删除：

```
a[0,2] = ['A', 'B']       # a becomes ['A', 'B', 'c', 'd', 'e']
a[2...5]=['C', 'D', 'E']  # a becomes ['A', 'B', 'C', 'D', 'E']
a[0,0] = [1,2,3]          # Insert elements at the beginning of a
a[0..2] = []              # Delete those elements
a[-1,1] = ['Z']           # Replace last element with another
a[-1,1] = 'Z'             # For single elements, the array is optional
a[-2,2] = nil             # Delete last 2 elements in 1.8; replace with nil in 1.9
```

除了用于索引一个数组的[]操作符之外，Array 类还定义了许多其他有用的操作符。使用+操作符来连接两个数组：

```
a = [1, 2, 3] + [4, 5]        # [1, 2, 3, 4, 5]
a = a + [[6, 7, 8]]           # [1, 2, 3, 4, 5, [6, 7, 8]]
a = a + 9                     # Error: righthand side must be an array
```

-操作符从一个数组中减掉另一个数组。它首先生成左侧数组的一份拷贝，然后从该拷贝中删除所有出现在右侧数组中的元素：

```
['a', 'b', 'c', 'b', 'a'] - ['b', 'c', 'd']    # ['a', 'a']
```

+操作符将创建一个新数组，它包含了左右两个数组的所有元素。使用<<操作符可在一个数组后面添加元素：

```
a = []            # Start with an empty array
a << 1            # a is [1]
a << 2 << 3       # a is [1, 2, 3]
a << [4,5,6]      # a is [1, 2, 3, [4, 5, 6]]
```

类似于 String 类，Array 类也采用乘法操作符来表示重复：

```
a = [0] * 8       # [0, 0, 0, 0, 0, 0, 0, 0]
```

Array 类还借鉴了布尔操作符|和&，它们分别用于两个数组之间的并集和交集运算。|操作符会合并那两个作为其实参的数组，然后从结果中去除所有重复的元素。&操作符将返回一个仅包含了那些同时出现在两个数组中的元素的数组，返回的数组中也不包含任何重复的元素：

```
a = [1, 1, 2, 2, 3, 3, 4]
b = [5, 5, 4, 4, 3, 3, 2]
a | b    # [1, 2, 3, 4, 5]: duplicates are removed
b | a    # [5, 4, 3, 2, 1]: elements are the same, but order is different
a & b    # [2, 3, 4]
b & a    # [4, 3, 2]
```

请注意这些操作符不具传递性：比如 a|b 不同于 b|a。但是，如果你忽略元素之间的顺序，将数组作为无序结合来看待，那么这些操作符将会更有意义。此外还需要注意的是，Ruby 并没有规定那些并集和交集算法的行为，而且不保证返回数组中的元素顺序。

Array 类定义了许多有用的方法，我们仅在此讨论 each 迭代器，它用于循环遍历一个数组的元素：

```
a = ('A'..'Z').to_a       # Begin with an array of letters
a.each {|x| print x }     # Print the alphabet, one letter at a time
```

你可能想一探究竟的其他 Array 方法包括：clear、compact!、delete_if、each_index、empty?、fill、flatten!、include?、index、join、pop、push、reverse、reverse_each、rindex、shift、sort、sort!、uniq!及 unshift。

我们将在第 4.5.5 节讨论并行赋值，在第 6 章讨论方法调用时将再次谈到数组。此外，我们将在第 9.5.2 节详细探究 Array 的 API。

3.4 哈希
Hashes

哈希是一种数据结构，它维护了一个**键**对象集合，而且将每个键都与一个值关联起来。哈希也被称为**映射**（*map*），因为它们将键映射到值上。哈希有时也被称为**关联数组**（*associative array*），因为它们将每个键都与值关联起来，而且可以将其理解成数组，只不过数组的索引可以是任意的对象而不仅限于整数。下面的例子能给出更清晰的解释：

```
# This hash will map the names of digits to the digits themselves
numbers = Hash.new     # Create a new, empty, hash object
numbers["one"] = 1     # Map the String "one" to the Fixnum 1
numbers["two"] = 2     # Note that we are using array notation here
numbers["three"] = 3

sum = numbers["one"] + numbers["two"]  # Retrieve values like this
```

接下来将描述 Ruby 中哈希字面量的语法，而且会解释一个对象想成为哈希键所要满足的条件。本书将在第 9.5.3 节提供更多有关 Hash 类 API 的信息。

3.4.1 哈希字面量
Hash Literals

一个哈希字面量就是一列由逗号分隔的键/值对，被包含在花括号中。键和值之间由双字符的“箭头”操作符=>来分隔。前面例子里的 Hash 对象可以像下面这样来创建：

```
numbers = { "one" => 1, "two" => 2, "three" => 3 }
```

一般来说，作为哈希的键，Symbol 对象比字符串更高效：

```
numbers = { :one => 1, :two => 2, :three => 3 }
```

Symbol 是不可改变的、功能受限的字符串，编写成以冒号为前缀的标识符；在后面的第 3.6 节将会更详细地介绍它们。

Ruby 1.8 允许用逗号来取代=>，但是 Ruby 1.9 已经不再支持这种不被推荐的语法：

```
numbers = { :one, 1, :two, 2, :three, 3 } # Same, but harder to read
```

Ruby 1.8 和 Ruby 1.9 都允许在键/值列表的最后跟上一个逗号：

```
numbers = { :one => 1, :two => 2, } # Extra comma ignored
```

当使用符号作为键时，Ruby 1.9 支持一种非常有用且简洁的哈希字面量语法，在这种语法中，分号被置于哈希键的后面并取代了=>（注 3）：

```
numbers = { one: 1, two: 2, three 计。3 }
```

请注意，在哈希键标识符和分号之间，不允许有任何空格。

3.4.2 哈希码、相等性及可变键
Hash Codes, Equality, and Mutable Keys

不出意外，Ruby 里的哈希是采用一种称为**哈希表**的数据结构来实现的。那些作为哈希键的对象必须有一个 hash 方法，该方法为该键返回一个 Fixnum 的**哈希码**。如果两个键相等，那么它们必须具有相同的哈希码。不相等的键也可能会拥有同样的哈希码，但是仅当一个哈希表只有极少的重复哈希码时，其效率才是最高的。

Hash 类采用 eql?方法比较键之间的相等性。对于大多数 Ruby 类来讲，eql?方法与==操作符一样（请参见第 3.8.5 节）。如果在所定义的新类里面重写了 eql?方法，那么你必须同时重写 hash 方法，否则你的类实例将无法作为一个哈希键。

如果你定义了一个类却没有重写 eql?方法，那么在该类的实例对象作为哈希键时，将对它们的对象同一性进行比较。你的类的两个不同实例将作为不同的哈希键，即使它们表示了相同的内容。在这种情况下，默认的 hash 方法是适宜的：它返回该对象的独一无二的 object_id。

请注意，使用可变对象作为哈希键会带来一些问题：改变一个对象的内容通常会改变其哈希码。如果你使用一个对象作为键，然后又改变了该对象，那么内部的哈希表将被破坏，而且该哈希的行为也将不再正确。

由于字符串是可变的，但是常常用作哈希键，所以 Ruby 将它们作为特例进行处理。对所有那些作为键的字符串，Ruby 会为其生成私有拷贝。然而，这是唯一的特例。你在使用其他任何可变对象作为哈希键时，必须非常小心谨慎。请考虑为这些可变对象生成私有拷贝，或者调用 freeze 方法。如果你必须使用可变的哈希键，那么请在每次更改键之后，调用 Hash 类的 rehash 方法。

3.5 范围
Ranges

一个 Range 对象表示位于一个开始值和一个结束值之间的一些值。范围（range）字面量的表现形式是在开始和结束值之间放置二三个点。如果使用两个点，那么该范围就是**包含性**

注 3：这是一种与 JavaScript 对象所使用的语法非常类似的语法。

的（*inclusive*），结束值将是该范围的一部分。如果使用三个点，那么该范围就是排他性的，结束值将不是范围的一部分。

```
1..10        # The integers 1 through 10, including 10
1.0...10.0   # The numbers between 1.0 and 10.0, excluding 10.0 itself
```

可以利用include?方法来测试一个值是否被包含在一个范围内（参见后面关于其他可选方法的讨论）：

```
cold_war = 1945..1989
cold_war.include? birthdate.year
```

Range的定义里还有一层隐含的意思，那就是顺序性（ordering）。如果一个范围对象是一些位于两个端点之间的值，那么显然必须有一些方法将这些值和端点进行比较。在Ruby中，这是通过比较操作符<=>来完成的。该操作符对它的两个操作数进行比较，并且根据它们的相对顺序（或相等）而返回-1、0或1。诸如数值和字符串之类的具有顺序性的类，都定义了<=>操作符。如果一个值想要成为范围的端点，那么它必须支持该操作符。通常情况下一个范围包含的值和它的端点都属于同一个类，然而，从技术的角度来讲，如果能通过范围端点的<=>操作符对一个值进行比较，并且结果表明这个值位于两个端点之间，那么就可以认为这个值是该范围的一个成员。

范围的主要用途是比较，该比较能够决定一个值是否位于该范围之内。它还有一个重要的用途是迭代：如果一个范围的端点所隶属的类定义了succ方法（为了得到后续成员），那么该范围的成员将组成一个离散的集合，而且能够使用each、step及Enumerable类的方法对这些成员进行迭代。作为例子，请考虑'a'..'c'这个range：

```
r = 'a'..'c'
r.each {|l| print "[#{l}]"}       # Prints "[a][b][c]"
r.step(2) { |l| print "[#{l}]"}   # Prints "[a][c]"
r.to_a                            # => ['a','b','c']: Enumerable defines to_a
```

上述代码可以工作的原因在于，String类定义了succ方法，而且'a'.succ得到'b'，'b'.succ的结果是'c'。可以像这样被迭代的范围对象被称为离散范围。由于无法对那些端点没有定义succ方法的范围进行迭代，所以这些范围被称为连续范围。具有整型端点的范围是离散的，而具有浮点型端点的范围是连续的。

在典型的Ruby程序里，具有整型端点的范围对象是最常用的。因为整型范围是离散的，所以它们可以被用于索引字符串和数组，它们也适合用来表示一个由递增的值构成的可枚举集合。

请注意，上述代码首先将一个范围字面量赋给一个变量，然后再通过该变量调用该范围的方法。如果你希望直接在一个范围字面量上调用方法，那么你必须将该字面量用圆括号括起来，否则该方法调用将会作用于该范围的末尾端点上，而不是作用于范围对象本身：

```
1..3.to_a    # Tries to call to_a on the number 3
(1..3).to_a  # => [1,2,3]
```

3.5.1 测试一个 Range 的成员关系
Testing Membership in a Range

Range 类定义了一些方法，用于决定一个任意值是否是一个范围的成员（也就是说，是否属于该范围）。在探讨细节之前，有必要先解释一下范围成员关系的两种不同定义。这与连续范围和离散范围之间的差异有关。第一种定义是，如果值 x 和范围对象 begin..end 的关系满足：

```
begin <= x <= end
```

那么值 x 就是该范围的成员。

此外，如果值 x 和对象 begin...end（有三个点）的关系满足：

```
begin <= x < end
```

那么值 x 也是该范围的成员。

由于所有范围的端点值都必须实现<=>操作符，所以这种成员关系的定义适用于任何 Range 对象，而且不需要端点实现 succ 方法，我们可以称之为连续的成员关系测试（continuous membership test）。

第二种成员关系的定义，即离散的成员关系（discrete membership），却依赖于 succ 方法。它将一个 begin..end 这样的 Range 对象看做一个集合，该集合包含 begin、begin.succ、begin.succ.succ 等。依据这种定义，范围的成员关系是一种集合成员关系（set membership），而且仅当一个值是某次 succ 方法调用的返回值时，这个值才被包含在该范围之内。请注意，对离散的成员关系的测试开销可能要远远大于对连续的成员关系的测试开销。

有了这些背景知识，我们下面就来描述那些用于测试成员关系的 Range 方法。Ruby 1.8 支持两个方法，即 include?和 member?。它们是同义词，而且都使用连续的成员关系测试：

```
r = 0...100      # The range of integers 0 through 99
r.member? 50     # => true: 50 is a member of the range
r.include? 100   # => false: 100 is excluded from the range
r.include? 99.9  # => true: 99.9 is less than 100
```

在 Ruby 1.9 里，情况发生了一些变化。一个新的 cover?方法被引入，其工作机理与 Ruby 1.8 里的 include?和 member?方法相同：它总是使用连续的成员关系测试。在 Ruby 1.9 里 include?和 member? 仍然是同义词。如果范围的端点是数字，那么这两个方法就使用连续的成员关系测试，就像它们在 Ruby 1.8 里所做的那样。然而，假如范围的端点是非数值对象，那么它们就会使用离散的成员关系测试。我们可以用一个离散的字符串范围来阐明这些变化（你或许须要使用 *ri* 来搞懂 String.succ 的工作机理）：

```
triples = "AAA".."ZZZ"
triples.include? "ABC"        # true; fast in 1.8 and slow in 1.9
triples.include? "ABCD"       # true in 1.8, false in 1.9
triples.cover?   "ABCD"       # true and fast in 1.9
triples.to_a.include? "ABCD"  # false and slow in 1.8 and 1.9
```

实际情况下，大多数范围都具有数值端点，Range API 在 Ruby 1.8 和 1.9 之间的差异没有太大影响。

3.6 符号
Symbols

一个 Ruby 解释器的典型实现会维护一个符号表（symbol table），在这个表中，它存储了其所知晓的所有类、方法及变量的名称，使一个这样的解释器可以避免大多数的字符串比较：比如，它可通过一个方法名在符号表中的位置来对其进行引用。这样一来，就将一个相对开销较大的字符串操作转化成了一个开销较小的整数操作。

这些符号并不是完全局限于解释器内部的，它们也可以被 Ruby 程序所使用。一个 Symbol 对象引用一个符号，通过在一个标识符或字符串的前面加上冒号的方式来表示一个符号字面量：

```
:symbol                    # A Symbol literal
:"symbol"                  # The same literal
:'another long symbol'     # Quotes are useful for symbols with spaces
s = "string"
sym = :"#{s}"              # The Symbol :string
```

和作用于字符串字面量的%q 和%Q 一样，Symbol 也具有一个%s 字面量语法，它允许在符号字面量里使用任意的分界符：

```
%s["]     # Same as :'"'
```

在反射代码（reflective code）里，符号常常被用于引用方法名。比如，假设我们希望知道某个对象是否包含 each 方法：

```
o.respond_to? :each
```

下面是另一个例子，它测试一个给定的对象是否能响应一个指定的方法，如果可以，则调用那个方法：

```
name = :size
if o.respond_to? name
  o.send(name)
end
```

你可以使用 intern 或 to_sym 方法将一个 String 转换成一个 Symbol，而且你可以使用 to_s 方法或其别名 id2name 将一个 Symbol 转换回一个 String。

```
str = "string"        # Begin with a string
sym = str.intern      # Convert to a symbol
sym = str.to_sym      # Another way to do the same thing
str = sym.to_s        # Convert back to a string
str = sym.id2name     # Another way to do it
```

两个字符串可能包含相同的内容，但是它们是两个完全不同的对象，对于符号来讲，这是不可能的。两个拥有相同内容的的字符串都会转换成同一个 Symbol 对象，两个不同的 Symbol 对象将总是含有不同的内容。

当你使用字符串的目的在于将其作为一种独一无二的标识符，而不在于它们的文本内容时，请改用符号。比如，与其编写一个期望以字符串“AM”或“PM”为实参的方法，倒不如

将该方法改写成期望符号:AM或:PM，比较两个 Symbol 对象的相等性要远远快于比较两个字符串的相等性。出于这个原因，我们通常都会优先选择以符号对象为哈希键，而不是字符串。

在 Ruby 1.9 中，Symbol 类定义了一些 String 方法，比如 length、size、比较操作符，甚至[]和~=操作符，这使在某种程度上，符号可以和字符串交换使用，而且使符号可以被作为一种不可变的（也是不可被垃圾收集的（garbage-collected））字符串来使用。

3.7 True、False 和 Nil

True, False, and Nil

我们在第 2.1.5 节已经知道了 true、false 和 nil 是 Ruby 的关键字。true 和 false 是两个布尔值，它们代表了真和假、是和否、开和关，nil 是一个特殊的保留值，用于表示没有值。

这三个关键字分别对应一个特殊的对象，true 是 TrueClass 类的一个单键实例（singleton instance），同样地，false 和 nil 分别是 FalseClass 和 NilClass 类的单键实例。请注意，Ruby 没有 Boolean 类。TrueClass 和 FalseClass 的超类都是 Object。

如果你想检查一个值是否为 nil，可以简单地将其与 nil 进行比较，或者使用 nil?方法：

```
o == nil      # Is o nil?
o.nil?        # Another way to test
```

请注意，true、false 和 nil 所引用的都是对象而不是数值，false 和 nil 不同于 0，true 和 1 也不是一回事。当 Ruby 需要一个布尔值时，nil 表现为 false，而 nil 或 false 之外的所有值都表现为 true。

3.8 对象

Objects

Ruby 是一门非常纯粹的面向对象语言：所有的值都是对象，而且没有基本类型（primitive type）和对象类型的区别，这一点不同于许多其他语言。在 Ruby 中，所有对象都继承一个 Object 类，而且共享那些定义于此类中的方法。本节讲述了 Ruby 中所有对象的一些公共特性，虽然某些方面的细节有点多，但这些都是必读的基础知识。

3.8.1 对象引用

Object References

当我们在 Ruby 中使用对象时，其实是在使用对象**引用**。我们操作的是对象的一个引用，而非对象本身。（注 4）当我们将一个值赋给一个变量时，我们并没有将一个对象拷贝到该变

注 4：如果熟悉 C 或 C++，那么你可以将一个引用理解为一个指针，即对象在内存中的地址值。然而，Ruby 并没有真正使用指针。Ruby 中的引用是不可见的，只存在于 Ruby 实现的内部。在 Ruby 中没有办法进行下面这样的操作：获取一个值的地址，解引用（dereference）一个值，或者进行指针算术运算。

量中，而是在此变量中存储了一个指向那个对象的引用。下面代码可以说明这一点：

```
s = "Ruby"    # Create a String object. Store a reference to it in s.
t = s         # Copy the reference to t. s and t both refer to the same object.
t[-1] = ""    # Modify the object through the reference in t.
print s       # Access the modified object through s. Prints "Rub".
t = "Java"    # t now refers to a different object.
print s,t     # Prints "RubJava".
```

在 Ruby 中，当把一个对象传递给一个方法时，其实传递的是该对象的引用，被传递的既不是对象本身，也不是一个指向该对象的引用的引用。换言之，方法实参是通过**值**而不是**引用**来传递的，只不过被传递的值正好是对象的引用。

因为对象引用被传递给方法，所以方法可以利用这些引用来修改底层的对象。在方法调用返回之后，还可以见到这些修改的结果。

3.8.1.1 立即值

我们之前说过在 Ruby 中所有的值都是对象，而且所有的对象都通过引用来操作，然而，在引用的实现中，Fixnum 和 Symbol 对象实际上是“立即值（immediate values）”而非引用。Fixnum 和 Symbol 都没有可变方法，所以它们都是不可变的，这就意味着，与其说它们是被按值操作的，还不如说它们是被按引用操作的。

应该将立即值的存在作为 Ruby 的一个实现细节来看待。立即值和引用值的唯一差别在于，不能为立即值定义单键方法（singleton methods）。（本书将在第 6.1.4 节讲解单键方法。）

3.8.2 对象生命周期
Object Lifetime

本章所讲述的那些 Ruby 内建类都具有字面量语法，这使创建这些类的实例变得非常简单，只须直接将这些对象的值写入代码中即可。但是其他类的对象须要显式地创建，而且通常都是通过一个称为 new 的方法来完成的：

```
myObject = myClass.new
```

new 是 Class 类的一个方法，它首先为新对象分配内存，然后通过调用该对象的 initialize 方法初始化这个新建的“空”对象的状态，传递给 new 方法的参数直接被传递给 initialize 方法。大多数类都定义了一个 initialize 方法，用于完成任何对于实例来说必须的初始化工作。

new 和 initialize 方法是创建对象的默认技术，但是类也可以定义其他一些被称为“工厂方法”的方法来返回其实例。我们将在第 7.4 节学到更多关于 new、initialize 和工厂方法的内容。

与 C 和 C++语言不同，Ruby 永远不须要显式释放（deallocate）对象，Ruby 采用了一种被称为**垃圾收集**（*garbage collection*）的技术来自动销毁那些不再需要的对象。当不再有引用指向某个对象时，我们称这个对象是**不可企及的**（*unreachable*）。（准确地说，可能还有其他不可企及的对象具有指向该对象的引用，但是由于这些对象本身是不可企及的，所以这些引用也没有意义。）不可企及的对象将成为垃圾收集的目标。

Ruby 采用了垃圾回收机制，这意味着相对于那些使用须要显式释放对象的语言所编写的程序来说，Ruby 程序将更少地遭受内存泄漏的困扰。但是垃圾回收并不能完全避免内存泄漏：对于那些本应短期存在的对象，如果某些代码创建了指向它们的引用，而这些引用是长期存在的，那么这些代码就成了内存泄漏之源。比如，有一个作为缓存的哈希对象（译注 9）。如果不使用某种最近最少使用的算法（least-recently-used algorithm）对缓存进行调整，那么只要这个哈希是可企及的，那些被缓存的对象就是可企及的。如果通过一个全局变量来引用该哈希对象，那么只要 Ruby 解释器还在运行，它就会一直处于可企及的状态（译注 10）。

3.8.3 对象标识
Object Identity

每个对象都有一个对象标识符，它是一个 `Fixnum`，而且你可以通过 `object_id` 方法来获得。该方法的返回值在该对象的生命周期中是独一无二且恒久不变的，只要该对象是可访问的，它就会一直持有一个同样的 ID，而且其他对象也不会共享这个 ID。

`id` 方法是 `object_id` 的同义词，但是已经不推荐使用了。在 Ruby 1.8 里，如果你使用它，会得到一个警告，而 Ruby 1.9 已经将其删除了。

`__id__`是 `object_id` 方法的一个有效的同义词，它是作为一种后备措施而存在的，所以即使 `object_id` 方法被取消定义（undefined）或重写了，你依然可以得到一个对象的 ID。

`Object` 类实现了 `hash` 方法，简单地返回一个对象的 ID。

3.8.4 对象的类和对象的类型
Object Class and Object Type

在 Ruby 中有几种方法可以用来确定一个对象所属的类，最简单的方法就是询问它自己：

```
o = "test"    # This is a value
o.class       # Returns an object representing the String class
```

如果你对一个对象的类继承结构感兴趣，可以询问任何一个类的超类是什么：

```
o.class                          # String: o is a String object
o.class.superclass               # Object: superclass of String is Object
o.class.superclass.superclass# nil: Object has no superclass
```

在 Ruby 1.9 中，`Object` 不再是类继承结构的真正根元素了：

译注 9：通过该对象来引用一些被缓存对象。

译注 10：这样一来，它所引用的那些被缓存对象也会一直处于可企及的状态，它们所占据的内存就一直无法被垃圾回收，于是就可能产生内存泄漏。

```
# Ruby 1.9 only
Object.superclass          # BasicObject: Object has a superclass in 1.9
BasicObject.superclass     # nil: BasicObject has no superclass
```

请参见第 7.3 节获得更多关于 BasicObject 的信息。

检查一个对象所属的类的一种非常直接的方法就是通过直接比较：

```
o.class == String       # true if is o a String
```

instance_of?方法完成同样的事情，但是更为优雅一些：

```
o.instance_of? String   # true if o is a String
```

通常情况下，当测试一个对象所属的类时，我们也想知道该对象是否是这个类的某个子类的实例。为了达到这个目的，我们使用 is_a?方法，或者它的同义方法 kind_of?：

```
x = 1                       # This is the value we're working with
x.instance_of? Fixnum       # true: is an instance of Fixnum
x.instance_of? Numeric      # false: instance_of? doesn't check inheritance
x.is_a? Fixnum              # true: x is a Fixnum
x.is_a? Integer             # true: x is an Integer
x.is_a? Numeric             # true: x is a Numeric
x.is_a? Comparable          # true: works with mixin modules, too
x.is_a? Object              # true for any value of x
```

Class 类定义了===操作符，可以替换 is_a?方法：

```
Numeric === x           # true: x is_a Numeric
```

这一点是 Ruby 所特有的，但是比起较为传统的 is_a?方法，这样做的可读性可能差一些。

在 Ruby 中，每个对象都属于一个定义良好的类（class），而且该类在此对象的生命周期中永远不变。从另一方面来看，一个对象的**类型**（*type*）则更多变一些，一个对象的类型与其所属的类有关，但是类只是该对象的类型的一部分。当谈及一个对象的类型时，我们指的是一个刻画了该对象的行为集合。换言之，一个对象的类型就是它可以响应的方法集合。(这个定义是递归的，因为重要的不仅是方法的名称，而且还有那些方法能接受的实参的类型。)

在使用 Ruby 编程的时候，我们常常不太在意一个对象所属的类是什么，只想知道是否可以在它身上调用某些方法。让我们来看看<<操作符这个例子。数组、字符串、文件及其他 IO 相关的类都将其定义为追加操作符（append operator）。如果我们正在编写一个产生文本输出的方法，那么也可以在该方法中使用这个操作符，然后在调用这个方法的时候，使用任何实现了<<操作符的参数。我们并不在意参数所属的类型，只要能够通过它调用<<操作符即可。我们可以使用 respond_to?方法来测试这一点：

```
o.respond_to? :"<<"  # true if o has an << operator
```

这种做法的缺点在于它仅仅检查了一个方法的名称，而没有检查该方法的参数。比如，Fixnum 和 Bignum 将<<实现为左移位操作符，而且希望其参数为一个数字而不是字符串。当我们使用 respond_to?方法来测试时，整数对象看起来是“可追加的”，但是当我们试图在一个字符串后面添加一个整数时，就会产生错误。对于这个问题没有普遍适用的解决方法，但是对于本例中的情况，一个权宜之计是采用 is_a?方法显式地将 Numeric 对象排除在外。

```
o.respond_to? :"<<" and not o.is_a? Numeric
```

关于类型和类之间的差别的另一个例子是 StringIO 类（来自 Ruby 标准库）。StringIO 使我们可以对字符串对象进行读写，就像它们是 IO 对象一样。StringIO 模仿了 IO 的 API——StringIO 对象定义了和 IO 对象一样的方法，但是 StringIO 并不是 IO 的一个子类。如果你编写了一个接受流（stream）实参的方法，而且使用 is_a? IO 来测试该实参，那么就无法使用 StringIO 类的对象作为该方法的实参。

关注类型而不是类，形成了 Ruby 中一种被称为“duck typing”的编程风格。我们将在第 7 章见到有关 duck typing 的例子（译注 11）。

3.8.5 对象的相等性

Object Equality

Ruby 拥有多得出奇的方法来比较对象的相等性。了解它们的工作机理是非常重要的，这样你才能在适当的时候选择正确的方法。

3.8.5.1 equal?方法

equal?方法由 Object 定义，用于测试两个值是否引用了同一个对象。对于两个不同的对象，该方法始终返回 false：

```
a = "Ruby"              # One reference to one String object
b = c = "Ruby"          # Two references to another String object
a.equal?(b)             # false: a and b are different objects
b.equal?(c)             # true: b and c refer to the same object
```

按照惯例，子类永远不要重写 equal?方法。

另一种检查两个对象是否是同一个对象的方法是比较它们的 object_id：

```
a.object_id == b.object_id   # Works like a.equal?(b)
```

3.8.5.2 ==操作符

==操作符是最为常用的进行相等性比较的方法。在 Object 类里，它是 equal?方法的同义词，用于比较两个对象引用是否是同样的。大多数类都重定义了这个操作符，使不同的实例之间也可以进行相等性测试：

```
a = "Ruby"          # One String object
b = "Ruby"          # A different String object with the same content
```

译注 11：Duck Typing 是一种动态类型机制，在这种机制下对象的有效语义是由它当前所拥有的方法和属性集来决定的，而不是由它所继承的类来决定的。这与主流语言（比如 Java、C#等）所采用的静态类型机制是不一样的。详情请参见维基百科。在 Ruby 的世界里，duck typing 是基本价值观之一。

```
a.equal?(b)  # false: a and b do not refer to the same object
a == b       # true: but these two distinct objects have equal values
```

请注意，上述代码表明单个等号（=）是赋值操作符，两个等号（==）才是 Ruby 用于比较相等性的操作符（这是 Ruby 和许多其他编程语言所共有的惯例）。

大多数的 Ruby 标准类都定义了==操作符，目的在于实现一个合理的相等性定义。Array 和 Hash 类都定义了==。如果两个数组拥有相同数量的元素，而且对应位置上的元素通过==比较的结果也是相等的，那么我们就说这两个数组通过==比较的结果是相等的。如果两个哈希拥有相同数量的键/值对，而且对应的键和值也是相等的，那么我们就说这两个哈希通过==比较是相等的。（值之间的相等性是通过==操作符来测试的，而键之间是通过 eql?方法来比较的，这将在本章后面进行描述。）

针对 Java 程序员的相等性说明

如果你是一位 Java 程序员，那么你已经习惯于采用==操作符测试两个对象是否是同一个对象，以及使用 equals 方法测试两个不同的对象是否拥有同样的值。在这些方面，Ruby 的惯例与 Java 的恰好相反。

Numeric 类在它们的==操作符里将进行简单的类型转换，因此（比如）Fixnum 1 和 Float 1.0 通过==比较的结果是相等的。诸如 String 和 Array 等类的==操作符，通常要求它的两个操作符属于同一个类。如果右侧操作数定义了一个 to_str 或 to_ary 转换方法（请参见第 3.8.7 节），那么原先的==操作符就会调用右侧操作数所定义的操作符==，以此来决定右侧操作数是否等于左侧的字符串或数组。因此，虽然并不常见，但我们确实可以定义一些类，使它们的比较行为类似于字符串或数组的比较行为。

!=（“不等于”）在 Ruby 中用于测试不等性。当 Ruby 见到!=时，它会简单地使用==操作符并且反转其结果，这意味着一个类只须定义==操作符来实现其相等性的含义。Ruby 会为你免费提供!=操作符。但是在 Ruby 1.9 中，类也可以显式地定义它们自己的!=操作符。

3.8.5.3 eql?方法

Object 将 eql?方法定义成 equal?方法的同义词，那些重定义了 eql?方法的类通常将其作为一个严格版的==操作符，即不允许进行类型转换。比如：

```
1 == 1.0     # true: Fixnum and Float objects can be ==
1.eql?(1.0)  # false: but they are never eql!
```

Hash 类采用 eql?检查两个哈希键是否相等。如果两个对象通过 eql?比较的结果是真，那么它们的 hash 方法也必须返回相同的值。典型情况下，如果你创建了一个类并且定义了==操作符，那么可以简单地编写一个 hash 方法并且定义 eql?方法，使其使用==。

3.8.5.4 ===操作符

===操作符通常被称为“条件相等性（case equality）”操作符，用于测试一个 case 语句的目标值是否和某个 when 从句相匹配。(case语句是一种多路分支结构，将在第 5 章被讲述。)

Object 类定义了一个默认的===操作符，它会调用==操作符，因此，对于许多类来说，条件相等性和==相等性是一样的。但是某些关键的类重新定义了===，使其不仅仅是一个成员关系或匹配操作符：Range 类定义了===，用于测试一个值是否位于某个范围内；Regexp 类定义了===，用于测试一个字符串是否与某个正则表达式相匹配；Class 类定义了===，用于测试一个对象是否是该类的一个实例；在 Ruby 1.9 中，Symbol 类定义了===，当其右侧操作数和左侧操作数是同一个符号对象时，或者当其右侧操作数是一个持有和左侧符号对象相同文本的字符串时，该操作符返回 true。举例如下：

```
(1..10) === 5     # true: 5 is in the range 1..10
/\d+/ === "123"   # true: the string matches the regular expression
String === "s"    # true: "s" is an instance of the class String
:s === "s"        # true in Ruby 1.9
```

像这样显式地使用===操作符并不多见，更为常见的是在一个 case 语句里隐式地使用它。

3.8.5.5 =~操作符

String 和 Regexp（及 Ruby 1.9 里的 Symbol 类）定义了=~操作符用于进行模式匹配。=~操作符其实根本不是一个相等性操作符，但是它的确含有一个等号，所以处于完整性考虑，将其也列在这里。Object 类定义了一个什么也不做（no-op）的=~，它总是返回 false。比如，如果你的类定义了某些模式匹配操作或具有近似相等性（approximate equality）的含义，那么你可以在该类中定义=~操作符。!~被定义成=~的反义，而且它只有在 Ruby 1.9 中才是可定义的，在 Ruby 1.8 中则不可定义。

3.8.6 对象的顺序
Object Order

实践中，每个类都可以定义一个==方法用于测试其实例之间的相等性，有些类还可以定义其实例之间的顺序性，即对于这样一个类的任意两个实例，它们必须要么相等，要么一个“小于”另一个。在定义了这样的顺序性的类里，数字类是最显而易见的例子。根据组成它们的字符编码的数值顺序，字符串也是具有顺序性的。(对于 ASCII 文本来说，这种顺序性可以被粗略地看成是大小写敏感的字母顺序。）如果一个类定义了顺序性，那么可以对该类的实例进行比较和排序。

在 Ruby 中，类通过实现<=>操作符来定义其顺序性。当左侧操作数小于右侧操作数时，该操作符返回-1，反之则返回 1，此外如果两个操作数相等，那么就返回 0。如果两个操作数

之间的比较操作没有意义（比如，假设右侧操作数属于另一个类），那么该操作符返回 nil。

```
1 <=> 5       # -1
5 <=> 5       # 0
9 <=> 5       # 1
"1" <=> 5     # nil: integers and strings are not comparable
```

只要实现了<=>操作符就可以对值进行比较，但是这并不特别直观，所以典型情况下，那些定义了<=>操作符的类还会将 Comparable 模块作为一个 mixin 包含进来。（模块和 mixin 将在第 7.5.2 节被讲述。）依据<=>操作符，Comparable 这个 mixin 定义了以下一些操作符：

<	小于
<=	小于或等于
==	等于
>=	大于或等于
>	大于

Comparable 模块没有定义!=操作符，Ruby 会自动地将该操作符定义成==操作符的反义。除了这些比较操作符之外，Comparable 还定义了一个有用的比较方法 between?：

```
1.between?(0,10)  # true: 0 <= 1 <= 10
```

如果<=>操作符返回 nil，那么所有基于它的操作符都将返回 false。NaN 这个特殊的 Float 值可以作为它的一个例子：

```
nan = 0.0/0.0;        # zero divided by zero is not-a-number
nan < 0               # false: it is not less than zero
nan > 0               # false: it is not greater than zero
nan == 0              # false: it is not equal to zero
nan == nan            # false: it is not even equal to itself!
nan.equal?(nan)       # this is true, of course
```

请注意，定义<=>及包含 Comparable 模块，这两个行为将为你的类定义一个==操作符，但是有些类会定义它们自己的==操作符，这通常发生在它们能比基于<=>的相等性测试更高效地实现==操作符时。类可以在它们的==和<=>操作符里，实现不同的相等性概念。举个例子，一个类可以定义其==操作符来实现大小写敏感的字符串比较操作，而定义<=>操作符来实现大小写不敏感的操作，这样一来它的实例们就可以更加自然地排序。尽管如此，总的来说最好的做法是让<=>返回 0，让==返回 true。

3.8.7 对象的转换

Object Conversion

许多 Ruby 类都定义了一些方法，它们可以返回对象的一种表示形式（representation），而这种表示形式本身却是另一个类的值。在这类方法当中，用于获取一个对象的 String 表示形

式的 to_s 方法，也许是最常被实现和最有名的了。在接下来的小节里，本书描述了各种转换类别。

3.8.7.1 显式转换

某些类定义了一些显式转换方法，供那些须要把一个值转换成另一种表现形式的应用代码使用。这类方法中，最为常见的是 to_s、to_i、to_f 及 to_a 方法，它们分别将调用对象转换成 String、Integer、Float 及 Array。

通常情况下，那些内建的方法并不会自动地调用上述转换方法。如果你调用一个参数为 String 的方法，却传递一个其他类型的对象给它，该方法是不会主动调用参数的 to_s 方法来进行转换的。(然而，对于那些将内插到由双引号引用的字符串中的值，Ruby 会利用其 to_s 方法进行自动转换。)

to_s 很可能是最重要的转换方法，因为对象的字符串表现方式在用户界面方面得到了非常广泛的应用。to_s 方法的一个重要的替代性选择是 inspect 方法。通常来说，to_s 用于返回一个人类可读的对象表现形式，适用于最终用户；另一方面，inspect 方法常用于调试，它返回一个能够给 Ruby 开发者带来帮助的表现形式。从 Object 继承来的默认的 inspect 方法，只是简单地调用 to_s。

3.8.7.2 隐式转换

有时候，一个类具有一些属于其他类的特征。Ruby 的 Exception 类表示程序中的一个错误或异常条件，而且封装了一条错误信息。在 Ruby 1.8 里，Exception 对象不仅能被转换成字符串，而且它们是“类似字符串 (string-like)”的对象。在许多情形下，它们都能被当成字符串来处理，就好像它们是字符串一样 (注 5)。比如：

```
# Ruby 1.8 only
e = Exception.new("not really an exception")
msg = "Error: " + e  # String concatenation with an Exception
```

由于 Exception 对象是类似字符串的，所以它们可被用于字符串连接操作符。大多数其他 Ruby 类都不支持这样的操作。Exception 对象之所以能表现得像 String 对象一样，是因为在 Ruby 1.8 里它实现了隐式转换方法 to_str，而且 String 类的+操作符调用其右侧操作数的 to_str 方法。

其他隐式转换方法包括 to_int、to_ary 及 to_hash，它们分别适用于那些想成为类似于整数的、数组的及 hash 的对象。不幸的是，对于在何种情形下这些隐式转换方法会被调用，并没有很好的文档化。此外，在内建类里，这些隐式转换方法也没有被普遍地实现。

注 5：然而，并不鼓励这样做。而且 Ruby 1.9 已经不再允许从 Exception 到 String 的隐式转换了。

在早些时候，我们提到==操作符在测试相等性的时候，可以进行一种弱（weak）类型转换。在 String、Array 和 Hash 类中定义的==操作符会检查其右侧操作数是否和左侧操作数属于同一个类。如果属于同一个类，就对它们进行比较，否则，==操作符会检查右侧操作数是否定义了一个 to_str、to_ary 或 to_hash 方法。==操作符并不调用这些方法，但是如果真的存在一个这样的方法，它们就会调用右侧操作数的==方法，以这样的方式来决定是否与左侧操作数相等。

在 Ruby 1.9 里，内建的 String、Array、Hash、Regexp 及 IO 类都定义了一个名为 try_convert 的类方法。如果这些方法的实参定义了一个合适的隐式转换方法，那么它们就对其进行转换，否则返回 nil。如果对象 o 定义了 to_ary 方法，那么 Array.try_convert(o)将返回 o.to_ary；否则，它会返回 nil。如果你想编写一些允许其实参进行隐式转换的方法，那么这些 try_convert 方法将提供便利。

3.8.7.3 转换函数

Kernel 模块定义了四个转换方法，作为全局转换函数。这些函数——Array、Float、Integer 及 String——具有和它们将要转换到的类相同的名字，而且它们因为以大写字母开头而显得与众不同。

Array 函数试图通过调用 to_ary 方法来将其实参转换成一个数组。如果没有定义 to_ary 方法，或者该方法返回 nil，那么 Array 函数就会试着调用 to_a 方法。如果没有定义 to_a 方法，或者该方法返回 nil，那么 Array 函数就会简单地返回一个新数组，并且将其实参作为该数组的一个元素。

Float 函数将其类型为 Numeric 的实参直接转换成 Float 对象，对于任何非 Numeric 的值，它会调用其 to_f 方法。

Integer 函数将其实参转换成一个 Fixnum 或 Bignum。如果其实参是一个 Numeric 值，它直接进行转换，浮点值会被截断而不是圆整；如果其实参是一个字符串，它寻找一个基数指示符（以 0 开头表示八进制，以 0x 开头表示十六进制，或者以 0b 开头表示二进制），并且据此对字符串进行转换。与 String.to_i 不同的是，Integer 函数不允许尾端出现非数值的字符。对于任何其他类型的实参，Integer 函数首先试图采用 to_int 进行转换，然后再使用 to_i 进行转换。

最后，String 函数通过简单地调用其实参的 to_s 方法将其转换成一个字符串。

3.8.7.4 算术操作符的类型强制转换

数值类型定义了一个名为 coerce 的转换方法，这个方法的意图是将其实参的类型转换成与其调用者相同的类型，或者将这两个对象都转换成更一般的兼容类型。coerce 方法总是返

回一个数组，它包含了两个类型相同的数值。该数组的第一个元素由 coerce 方法的实参转换而来，第二个元素来自 coerce 方法的调用对象（如果需要，就进行转换）：

```
1.1.coerce(1)          # [1.0, 1.1]: coerce Fixnum to Float
require "rational"     # Use Rational numbers
r = Rational(1,3)      # One third as a Rational number
r.coerce(2)            # [Rational(2,1), Rational(1,3)]: Fixnum to Rational
```

算术操作符使用 coerce 方法。比如，Fixnum 的+操作符并不知道如何操作 Rational 数字，如果其右侧操作数是一个 Rational 值，它不知道该如何把该值加到左侧操作数上，coerce 方法提供了解决之道。算术操作符被编写成如下方式：当不知道右侧操作数的类型时，它们将调用右侧操作数的 coerce 方法，并且将左侧操作数作为实参传递过去。回到刚才的那个将一个 Fixnum 和一个 Rational 相加的例子，Rational 的 coerce 方法返回一个包含了两个 Rational 值的数组，现在 Fixnum 的+操作符只须简单地在数组所包含的值上调用+即可。

3.8.7.5 布尔类型转换

在讨论类型转换时，我们要特别注意布尔值。Ruby 对于其布尔值有着非常严格的限制：true 和 false 具有 to_s 方法，分别返回字符串“true”和“false”，此外没有定义任何其他的转换方法，不存在将其他值转换成布尔值的 to_b 方法。

在有些语言里，false 和 0 是一回事，或者可以和 0 相互转换。在 Ruby 里，true 和 false 是与其他对象截然不同的两个对象，而且不存在任何隐式的转换能够将其他值转换成 true 或 false。但是，这并不是故事的全部。Ruby 中的布尔操作符及条件和循环结构可使用布尔表达式，它们也可以接受 true 和 false 之外的值。规则很简单：在布尔表达式里，任何不同于 false 或 nil 的值都表现得像 true 一样（但不是转换成 true），nil 则表现得像 false 一样。

假设要测试变量 x 是否是 nil，在一些语言里，你必须显式地编写一个值为 true 或 false 的比较表达式：

```
if x != nil       # Expression "x != nil" returns true or false to the if
  puts x          # Print x if it is defined
end
```

这段代码在 Ruby 里也能运行，但是更常见的做法是简单地利用这个事实：所有不同于 nil 和 false 的值都表现得像 true 一样：

```
if x          # If x is non-nil
  puts x      # Then print it
end
```

请记住在 Ruby 中，类似于 0、0.0 及空字符串“”的值都表现得像 true 一样。如果你已经习惯了那些类似于 C 或 JavaScrpit 的语言，这一点会让你大吃一惊。

3.8.8 拷贝对象
Copying Objects

Object 类定义了两个密切相关的方法，用于拷贝对象。clone 和 dup 方法都返回一个调用它们的对象的浅拷贝，如果被拷贝的对象含有指向其他对象的引用，那么只有这些引用被拷贝，而那些被引用的对象本身却不会被拷贝。

如果被拷贝的对象定义了一个 initialize_copy 方法，那么 clone 和 dup 方法就会简单地分配一个新的空实例（该实例所属的类与被拷贝对象的相同），然后在其上调用 initialize_copy 方法。被拷贝的对象将作为实参被传递，而且这个“拷贝构造函数”可以随意的初始化该副本。比如，initialize_copy 方法可以递归地拷贝一个对象的内部数据，这样得到的结果对象就不仅仅是原始对象的一个浅拷贝了。

类还可以直接重写 clone 和 dup 方法，从而得到它们期望的任何副本。

Object 类定义的 clone 和 dup 方法存在两处重要的差别。首先，clone 方法会拷贝一个对象的被冻结（frozen）和受污染（tainted）状态（稍后会讲解），而 dup 方法仅仅拷贝受污染状态，在一个被冻结的对象上调用 dup 方法将返回一个未被冻结的副本。其次，clone 方法会拷贝对象的所有单键方法，而 dup 则不会。

3.8.9 编组对象
Marshaling Objects

你可以通过调用类方法 Marshal.dump 来保存一个对象的状态（注 6）。如果你将一个 I/O 流对象作为第二个实参传递给 Marshal.dump 方法，它就把被保存对象的状态（而且会递归地包括它所引用的所有对象）写到这个流当中。否则，Marshal.dump 方法会简单地将编组后的状态作为一个二进制字符串返回。

为了恢复一个编组后的对象，你可以将一个包含了该对象的字符串或 I/O 流传递给 Marshal.load。

编组一个对象是一种保存其状态以供后续使用的非常简单的方法，而且这些方法可以为 Ruby 程序提供一种自动的文件格式。然而，请注意，Marshal.dump 和 Marshal.load 所使用的二进制格式是存在版本依赖性的，新版本的 Ruby 不保证能够读取那些由较旧版本的 Ruby 写入的对象。

另一种对 Marshal.dump 和 Marshal.load 方法的用法是创建对象的深拷贝（deep copy）：

```
def deepcopy(o)
  Marshal.load(Marshal.dump(o))
end
```

注 6：“marshal”这个单词及其变形词有时候被拼写成包含两个 l，如：marshall、marshalled 等，如果采用这种方式拼写这个单词，那么你要记住 Ruby 的这个类的名称只包含一个 l。

请注意，文件流、I/O 流，以及 Method 和 Binding 对象，由于过于动态而无法被编组，没有可靠的方法能恢复它们的状态。

YAML（“YAML Ain't Markup Language”）是 Marshal 模块的一种被广泛采用的替代方案，它用于将对象保存成可读的文本格式，以及从这些文本中装载对象。它位于标准库中，你必须在使用它之前先 require 'yaml'。

3.8.10 冻结对象
Freezing Objects

可以调用其 freeze 方法将任何对象**冻结**起来，一个被冻结的对象将变得不可改变——它所有的内部状态都不能被改变，而且对其可变方法的调用也会失败：

```
s = "ice"          # Strings are mutable objects
s.freeze           # Make this string immutable
s.frozen?          # true: it has been frozen
s.upcase!          # TypeError: can't modify frozen string
s[0] = "ni"        # TypeError: can't modify frozen string
```

如果一个类被冻结，那么将无法再向该类添加任何方法。

你可以通过 frozen?方法来检查一个对象是否被冻结了。一旦被冻结，就没有办法“解冻”该对象了。如果你使用 clone 方法拷贝一个被冻结的对象，那么得到的副本也将是被冻结的，但是如果你采用 dup 方法拷贝一个被冻结的对象，那么得到的副本将不是被冻结的。

3.8.11 污染对象
Tainting Objects

为了避免 SQL 注入攻击及类似的安全风险，Web 应用必须常常追踪一些数据，它们来源于不可信的用户输入。Ruby 为这类问题提供了一个简单的解决方法：可以通过调用其 taint 方法将任何对象标记成受污染的（tainted）。一旦一个对象成了受污染的，那么任何源自它的对象都将成为受污染的。可以通过 tainted?方法测试一个对象的受污染性：

```
s = "untrusted"       # Objects are normally untainted
s.taint               # Mark this untrusted object as tainted
s.tainted?            # true: it is tainted
s.upcase.tainted?     # true: derived objects are tainted
s[3,4].tainted?       # true: substrings are tainted
```

用户输入——比如命令行参数、环境变量及用 gets 方法读入的字符串——都会自动地成为受污染的。

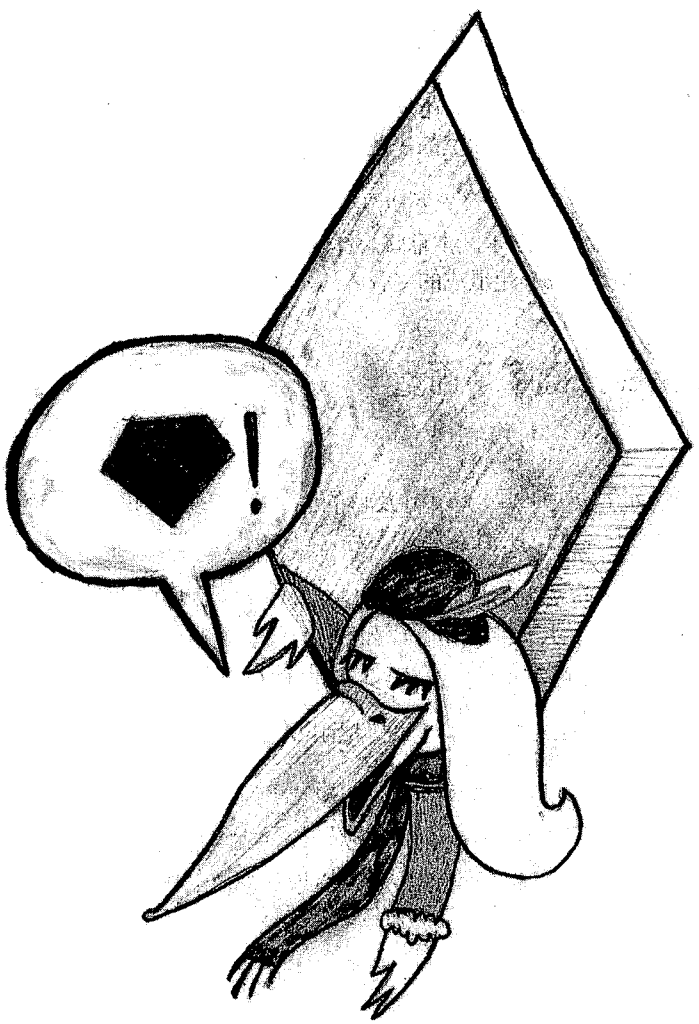
由 clone 和 dup 方法得到的受污染对象的副本也是受污染的。一个受污染的对象可以通过 untaint 方法成为未受污染的（untainted），当然，你只有在检查了该对象的内容并且确信没有安全风险的前提下才应该这么做。

Ruby 的对象污染（tainting）机制在与全局变量$SAFE 配合使用时威力最大。当该变量被设置成大于 0 的值时，Ruby 将对各种内建的方法进行限制，使它们无法使用受污染的数据。参见第 10 章，你可获得$SAFE 变量的更进一步的细节信息。

第 4 章

表达式和操作符

Expressions and Operators



一个**表达式**就是一块 Ruby 代码。Ruby 解释器可以对其求值并得到一个结果。下面是一些例子：

```
2                   # A numeric literal
x                   # A local variable reference
Math.sqrt(2)        # A method invocation
x = Math.sqrt(2)    # Assignment
x*x                 # Multiplication with the * operator
```

如你所见，基本的表达式——比如字面量、变量引用及方法调用——可以和**操作符**一起组合成更大的表达式，比如赋值操作符和乘法操作符。

许多编程语言都会区分低层的表达式和高层的**语句**，比如条件和循环。在这些语言里，语句用于控制一个程序的流程，但是它们没有值。它们是被执行，而不是被求值。在 Ruby 中，表达式和语句之间并没有清晰的界限。Ruby 中的所有东西，包括类和方法定义在内，都可以作为一个表达式来求值，并且返回一个值。但是，在那些典型地用于表达式的语法和典型地用于语句的语法之间进行区分，还是有用处的。那些会影响控制流程的 Ruby 表达式将在第 5 章进行描述，而定义方法和类的表达式将在第 6 章和第 7 章讲述。

本章的内容涵盖了那些更为简单的、更加传统意义上的表达式。字面量值是最简单的表达式，我们已经在第 3 章讲述过了。本章将解释变量引用、常量引用、方法调用、赋值，以及由较小的表达式和操作符一起组成的复合表达式。

4.1 字面量和关键字字面量
Literals and Keyword Literals

字面量就是一些诸如 `1.0`、`hello world` 及 `[]` 的值，它们是被直接嵌入你的程序文本中的。我们曾在第 2 章介绍过它们，并且在第 3 章对它们进行了详细的描述。

值得注意的是，许多字面量，比如数字，都是基本表达式，即最简单的表达式（它们不是由更简单的表达式组成）。其他字面量，比如数组字面量、哈希字面量及那些使用了内插的由双引号引用的字符串，由于包含子表达式，因此都不是基本表达式。

某些 Ruby 关键字是基本表达式，可以把它们当作**关键字字面量**或特殊形式的变量引用：

`nil`	求值为 `nil`，属于 `NilClass` 类
`true`	求值为 `TrueClass` 类的单键实例，是一个代表了布尔值 `true` 的对象
`false`	求值为 `FalseClass` 类的单键实例，是一个代表了布尔值 `false` 的对象
`self`	求值为当前对象（请参见第 7 章获得更多关于 `self` 的信息）

__FILE__	求值为一个字符串，该字符串包含了 Ruby 解释器正在执行的文件名称。这在错误消息里非常有用
__LINE__	求值为一个整数，表示当前代码的行数，该代码属于__FILE__所代表的文件
__ENCODING__	求值为一个 Encoding 对象，该对象指定了当前文件的编码（仅限于 Ruby 1.9）

4.2 变量引用
Variable References

一个**变量**就是一个值的名字。通过赋值表达式可以创建变量并对其赋值。本章后面将讲述赋值表达式。在程序中，当一个变量名不是出现在赋值操作的左侧时，它就是一个变量引用表达式，而且会被求值成该变量的值。

```
one = 1.0    # This is an assignment expression
one          # This variable reference expression evaluates to 1.0
```

正如在第 2 章所描述的，Ruby 中有四种类型的变量，而且由词法规则来规范它们的名字。以$开头的变量是全局变量，在整个 Ruby 程序里都是可见的。以@和@@开头的变量分别是实例变量和类变量，用于面向对象编程（本书将在第 7 章解释它们）。以一个下划线或小写字符开头的变量是局部变量，它们仅在当前方法或代码块内有定义。（请参见第 5.4.3 节获取更多关于局部变量作用域的信息。）

变量通常都具有简单的、不加限定修饰的名字。如果在一个表达式中出现了一个.或::，那么该表达式要么是一个常量引用，要么是一个方法调用。比方说，Math::PI 是一个常量引用，而表达式 item.price 是对一个称为 price 的方法的调用，该方法属于一个名为 item 的变量。

Ruby 解释器在它启动的时候定义了一些全局变量，请参见第 10 章获得这些变量的列表。

4.2.1 未初始化的变量
Uninitialized Variables

通常来说，你应该总是在使用变量之前，对其进行赋值或**初始化**。然而在某些情况下，Ruby 允许你使用那些还没有被初始化的变量。针对不同类型的变量具有不同的规则。

类变量

在使用类变量之前，必须对其进行赋值。如果你引用一个未被赋值的类变量，Ruby 将抛出一个 NameError。

实例变量

如果你引用了一个未被初始化的实例变量，Ruby 将返回 nil。然而，对这种行为的依赖是不好的编程风格。如果你在调用 Ruby 解释器时使用-w 选项，那么 Ruby 将针对未初始化的变量发出一个警告。

全局变量

未初始化的全局变量类似于未初始化的实例变量：它们的值为 nil，但是在使用-w 选项运行 Ruby 时，会产生警告。

局部变量

这种情况要比其他的更复杂一些，因为局部变量没有一个标点符号作为前缀，这就意味着局部变量引用看起来就像方法调用表达式一样。如果 Ruby 解释器遇见了一个对局部变量的赋值操作，它就会知道等号左侧的是一个变量而不是一个方法调用，而且返回这个局部变量的值。如果不是赋值操作，那么 Ruby 就会将这样的表达式当成方法调用。如果没有方法能匹配那个名字，Ruby 将抛出一个 NameError。

因此，总的来说，试图在一个局部变量被初始化之前就使用它会导致一个错误。但是还存在一个略显古怪的行为——当 Ruby 解释器看到针对一个变量的赋值表达式时，它会认为这个变量存在了，即使这个赋值操作并没有被实际执行，该变量也已经存在了。一个已经存在但是还没有被赋值的变量会被给予默认值 nil。比如：

```
a = 0.0 if false  # This assignment is never executed
print a           # Prints nil: the variable exists but is not assigned
print b           # NameError: no variable or method named b exists
```

4.3 常量引用
Constant References

除了其值应该在程序运行期间保持不变之外，Ruby 中的常量类似于变量。Ruby 解释器并不会保证常量的常量性，但是如果程序修改了一个常量的值，它确实会发出一个警告。从词法的角度讲，除了以大写字母开头之外，常量的名字看起来类似于变量的名字。按照惯例，大多数常量名都是全部大写的，并且使用下划线来分隔单词，比如 LIKE_THIS。Ruby 的类和模块名也是常量，但依照惯例，它们采用驼峰格式（camel case），比如 LikeThis（译注 1）。

尽管常量看起来就好像首字母大写的局部变量一样，它们却具有全局的可见性：可以在一个 Ruby 程序的任何地方使用它们而不必考虑作用域。但是与全局变量不同的是，常量可以在类和模块中被定义，因此可以拥有限定修饰的名字。

一个常量引用是一个表达式，它的值就是该常量的值。最简单的常量引用是基本表达式——它们只包含常量的名字：

译注 1：驼峰格式为英文单词之间不加空格，每个单词的首字母都大写。

```
CM_PER_INCH = 2.54    # Define a constant.
CM_PER_INCH           # Refer to the constant. Evaluates to 2.54.
```

除了这样的简单引用之外，常量引用还可以是复合表达式。在这种情况下，将采用::将常量名和定义它的类或模块名分隔开来。::的左侧可以是任意的表达式，只要它的值是一个类或模块对象。(然而通常情况下，这个表达式都只是一个指向类或模块名的常量引用。)::的右侧是定义在该类或模块中的一个常量的名字。比如：

```
Conversions::CM_PER_INCH # Constant defined in the Conversions module
modules[0]::NAME         # Constant defined by an element of an array
```

模块可以嵌套，这意味着常量可以被定义在嵌套的名字空间里，就像下面这样：

```
Conversions::Area::HECTARES_PER_ACRE
```

::左侧的操作数可以省略，这种情况下将在全局域中查找右侧的常量：

```
::ARGV      # The global constant ARGV
```

请注意，并不存在一个针对常量的“全局作用域”。类似于全局函数，全局常量被定义（和被查找）在 Object 类里，因此，表达式::ARGV 只是 Object::ARGV 的简写形式。

当一个常量引用表达式被::限定修饰的时候，Ruby 很清楚该到何处去查找这个指定的常量。但是当没有::时，Ruby 解释器必须查找到该常量的一个恰当的定义。它将在语法上的外围作用域里查找，而且还会在外围类或模块的继承层次结构中进行查找。本书将在第 7.9 节介绍其全部细节。

当 Ruby 对一个常量引用表达式求值的时候，它返回该常量的值，或者当没有常量能匹配该名字时，它抛出一个 NameError 异常。请注意，只有在真正被赋值之后，常量才存在。这和变量是不同的。当 Ruby 解释器看到对变量的赋值时，它们就已经存在了，即使并没有执行该赋值操作。

Ruby 解释器在其启动的时候预定义了一些常量。请参见第 10 章获得它们的列表。

4.4 方法调用
Method Invocations

一个方法调用表达式由四部分组成：

- 一个任意的表达式，它的值就是将要在其上调用该方法的对象。这个表达式后接分隔符.或::，然后再接方法名。表达式和分隔符都是可选的，如果它们被省略了，就会在 self 上调用该方法。

- 被调用的方法名称。这是一个方法调用表达式里唯一必须存在的部分。
- 将要传递给该方法的参数值。参数列表可以置于一对圆括号中，但通常圆括号都是可选的。（本书将在第 6.3 节详细讨论可选的和必须的圆括号。）如果有多个参数，它们将被逗号分隔。所需参数的个数和类型依赖于方法的定义，有些方法不需要参数。
- 一个可选的代码块，它位于一对花括号或一个 do/end 中，方法可以使用 yield 关键字对该代码块进行调用。这种将任意的代码与任何方法调用关联在一起的能力，构成了 Ruby 强大的迭代器方法的基石。我们将在第 5.3 节和第 5.4 节学习更多有关代码块关联方法调用的内容。

通常使用.将一个方法名和它的调用对象分隔开来。也可以使用::，但是由于它使方法调用看起来像常量引用表达式，所以用的比较少。

当 Ruby 解释器获得了一个方法名及它的调用对象之后，它就会通过一个名为“方法查找”或“方法名解析”的过程，为该方法寻找一个合适的方法定义。目前，细节并不重要，本书将在第 7.8 节对它们进行全面的解释。

一个方法调用表达式的值就是方法体中最后一个被求值的表达式的值。在第 6 章，我们会讲述更多关于方法定义、方法调用及方法返回值的内容，但是在这里，我们先给出一些方法调用的例子：

```
puts "hello world"    # "puts" invoked on self, with one string arg
Math.sqrt(2)          # "sqrt" invoked on object Math with one arg
message.length        # "length" invoked on object message; no args
a.each {|x| p x }     # "each" invoked on object a, with an associated block
```

调用全局函数

再看一下先前展示的方法调用：

```
puts "hello world"
```

这是一个对 Kernel 模块的 puts 方法的调用。在 Kernel 中定义的方法是全局函数，就好比是在所有类之外的顶层空间中定义的方法。全局函数是被作为 Object 类的私有方法来定义的。我们将在第 7 章学习更多关于私有方法的内容。目前你只须要知道，私有方法是不允许显式地通过一个调用对象来进行调用的——它们总是通过 self 来隐式地调用的。self 总是有定义的，而且不管它的值是什么，该值都是一个 Object。由于全局函数是 Object 的方法，所以不管 self 的值是什么，在任何上下文里都是可以调用这些方法（隐式地）的。

先前我们使用了 message.length 作为方法调用的例子。你可能会认为它是一个变量引用表达式，其值是对象 message 的 length 变量的值。然而，事实并非如此。Ruby 拥有一个非常纯粹的面向对象编程模型：Ruby 对象将它所有的内部实例变量封装起来，只对外暴露一些方法。由于 length 方法不需要实参，而且调用它时省略了圆括号，所以它看起来像一个变量引用。事实上，这是有意而为的。我们称这样的方法为属性访问器方法，称 message 对象拥有一个 length 属性（注 1）。如我们即将见到的，可以为 message 对象定义一个名为 length=的方法。如果这个方法接受一个实参，那么它就是一个属性可变方法，而且 Ruby 会在处理一个赋值操作的时候调用它。如果定义了一个这样的方法，那么下面的两行代码调用的都是同一个方法：

```
message.length=(3)    # Traditional method invocation
message.length = 3    # Method invocation masquerading as assignment
```

现在考虑下面这样的代码，假定变量 a 持有一个数组：

```
a[0]
```

你可能会再次认为这是一种特殊的变量引用表达式，而这次的变量是一个数组元素。但是，这是一个方法调用。Ruby 解释器将对数组的访问转换成下面这种形式：

```
a.[](0)
```

对数组的访问变成了一个对数组的[]方法的调用，并且将数组索引作为实参。这种数组访问语法并不仅限于数组，任何对象都可以定义一个[]方法。当使用方括号来“索引”该对象时，任何位于方括号中的值都会被作为实参传递给[]方法。如果[]方法被实现成期望三个实参，那么你就需要在调用该方法时，在方括号中放置三个由逗号分隔的表达式。

对数组的赋值操作也是通过方法调用来完成的。如果一个对象 o 定义了一个名为[]=的方法，那么表达式 o[x]=y 将变成 o.[]=(x,y)，而且表达式 o[x,y]=z 将变成 o.[]=(x,y,z)。

本章后面我们将看到许多 Ruby 操作符都被定义成方法，像 x+y 这样的表达式被转换成 x.+(y)，+就是方法名。Ruby 的许多操作符都是方法，这意味着你可以在自己的类中重定义这些操作符。

现在让我们考虑下面这个非常简单的表达式：

```
x
```

注 1：这并不意味着每个无参的方法都是属性访问器方法。比如一个数组对象的 sort 方法就没有参数，但是它却并不返回一个属性值。

如果存在一个名为 x 的变量（即如果 Ruby 解释器见到了一个对 x 的赋值操作），那么这就是一个变量引用表达式。如果不存在这样的变量，那么它就会被当成是在 self 对象上的，对方法 x 的一次不带实参的调用。

Ruby 的关键字 super 是一个特殊的方法调用表达式，在创建一个子类时会用到这个关键字。如果调用它时不带实参，那么它调用超类中的一个与当前方法同名的方法，而且将当前方法的所有实参都传递给那个同名方法。此外，我们还可以在调用它时传递任意数量的实参，就好像它是一个真正的方法名一样。本书将在第 7.3.3 节详细介绍 super 关键字。

4.5 赋值
Assignments

一个赋值表达式可以为一个或多个左值（lvalue）指定一个或多个值。**左值**是一个术语，表示那些可以出现在一个赋值操作符左侧的东西。（相应地，称那些位于一个赋值操作符右侧的值为**右值**。）在 Ruby 中变量、常量、属性及数组元素是左值。对于不同类型的左值，对应的赋值表达式的规则和含义都不同。本节会详细描述每种情况。

在 Ruby 中有三种不同形式的赋值表达式。简单的赋值包括一个左值、=操作符及一个右值。比如：

```
x = 1     # Set the lvalue x to the value 1
```

缩写形式的赋值操作是一个简写的表达式，它通过在当前变量的值上应用某些操作（比如相加）来更新该对象的值。缩写形式的赋值操作使用类似于+=和*=的赋值操作符，它们将一个二元操作和一个等号结合起来：

```
x += 1    # Set the lvalue x to the value x + 1
```

最后，任何具有多于一个左值或多于一个右值的赋值表达式称为并行赋值。下面是一个简单的例子：

```
x,y,z = 1,2,3   # Set x to 1, y to 2 and z to 3
```

当一个并行赋值的左值数量和右值数量不等，或者其右值是一个数组时，情况会更复杂一些。下面是全部细节。

一个赋值表达式的值就是那个被赋给左值的值（或是一个数组）。此外，赋值操作符是右结合的——如果在单个表达式里出现了多个赋值操作，它们将按从右到左的顺序依次求值，这意味着可以通过串联多个赋值操作的形式将同一个值赋给多个变量：

```
x = y = 0   # Set x and y to 0
```

请注意，这并不是并行赋值——它只是将两个简单赋值串联在一起：首先 y 被赋值为 0，然后 x 被赋值为第一个赋值操作的值（也是 0）。

赋值及其副作用

比一个赋值表达式的值更为重要的是，赋值操作设置了一个变量（或其他左值）的值，因此会影响系统状态。这种对系统行为的影响被称为赋值操作的副作用。

许多表达式都没有副作用，不会影响系统状态，它们是幂等的，这意味着这样的表达式可以被一遍又一遍地求值，而且每次都会返回相同的结果；同时对这些表达式进行求值时，不会影响到其他表达式的值。下面是一些没有副作用的表达式：

```
x + y
Math.sqrt(2)
```

赋值操作不是幂等的，理解这一点非常重要：

```
x = 1       # Affects the value of other expressions that use x
x += 1      # Returns a different value each time it is evaluated
```

一些方法，诸如 Math.sqrt，是幂等的：对它们的调用没有副作用。其他方法则不是幂等的，这主要依赖于这些方法是否对非局部变量进行了赋值操作。

4.5.1 对变量进行赋值
Assigning to Variables

当我们想到赋值操作时，我们经常想到变量，事实上，它们是赋值表达式最常见的左值。回忆一下，Ruby 拥有四种类型的变量：局部变量、全局变量、实例变量及类变量，它们之间通过变量名的首字母进行区分。但是对这四种变量的赋值操作的行为是一样的，因此我们不需要在此对它们的类型进行区分。

请记住，在对象之外，Ruby 的实例变量是永远不可见的，而且实例变量名永远不会被对象名来限定修饰。考虑这个赋值操作：

```
point.x, point.y = 1, 2
```

这个表达式的左值不是变量，而是属性。本书稍后就会解释。

对一个变量的赋值操作的行为和你的期望一样：变量被设置成指定的值。唯一的问题与变量声明，以及局部变量名和方法名之间的二义性有关。Ruby 没有显式声明一个变量的语法：当对一个变量赋值时，它就存在了。此外，局部变量名和方法名看起来一样——没有类似于$之类的前缀来区分它们。因此，一个诸如 x 的简单表达式既可以引用一个名为 x 的局部变量，也可以引用 self 对象的一个名为 x 的方法。为了消除这种二义性，Ruby 将一个标识符当作局部变量来处理，只要它之前见到了对这个变量的赋值操作即可，即便这个赋值操作并没有被执行也是如此。下面的代码说明了这一点：

```
class Ambiguous
  def x; 1; end   # A method named "x". Always returns 1

  def test
    puts x        # No variable has been seen; refers to method above: prints 1

    # The line below is never evaluated, because of the "if false" clause. But
    # the parser sees it and treats x as a variable for the rest of the method.
    x = 0 if false

    puts x    # x is a variable, but has never been assigned to: prints nil

    x = 2     # This assignment does get evaluated
    puts x    # So now this line prints 2
  end
end
```

4.5.2 对常量赋值
Assigning to Constants

常量和变量之间有一个很明显的区别：在程序执行期间，它们的值应该保持不变。因此，有一些针对常量赋值的特殊规则：

- 对一个已经存在的常量进行赋值会导致 Ruby 发出一个警告，然而 Ruby 仍然会执行这个赋值，这意味着常量并不是真正恒久不变的。
- 在方法体内部是不允许对常量进行赋值的。Ruby 认为方法应该可以被多次调用；如果你可以在一个方法内部对常量进行赋值，那么当每次调用该方法时，都会发出警告，因此，Ruby 干脆就不允许这样做了。

与变量不同，只有在 Ruby 解释器真正地执行了对常量的赋值表达式之后，常量才会存在。像下面这样的一个没有被求值的表达式是不会产生一个常量的：

```
N = 100 if false
```

这就意味着常量绝对不会处于未初始化的状态。如果一个常量已经存在，那么它必然会有一个被赋给它的值。只有将 `nil` 赋给一个常量时，它才会具有 `nil` 值。

4.5.3 对属性和数组元素的赋值
Assigning to Attributes and Array Elements

Ruby 中对属性或数组元素的赋值其实是对方法调用的简写。假定一个对象具有一个名为 `m=` 的方法，方法名的最后一个字符是等号，那么 `o.m` 就可以作为一个赋值表达式的右值。更近一步，假定将值 `v` 赋给它：

```
o.m = v
```

Ruby 解释器将这个赋值操作转换成如下的方法调用：

```
o.m=(v)  # If we omit the parens and add a space, this looks like assignment!
```

也就是说，解释器将值 v 传递给方法 m=，该方法可以对 v 做任何操作。典型地，它会检查该值是否属于期望的类型，并且在期望的范围内，然后它将该值存入对象的一个实例变量里。像 m=这样的方法通常都会有一个对应的 m 方法，它简单地返回那个最近一次传递给 m= 的值。我们称 m=是一个**可变**方法，而 m 是一个**读取器**方法（*getter* method）。当一个对象具有这样一对方法时，我们就说它具有一个属性 m。在其他语言里，属性有时会被称为“properties”。我们将在第 7.1.5 节学习更多关于属性的内容。

对数组元素的赋值也是通过方法调用来完成的。如果一个对象 o 定义了一个接受两个参数的、名为[]=的方法（就是这三个标点字符构成了方法名），那么表达式 o[x]=y 实际上会被像下面这样执行：

```
o.[]=(x,y)
```

如果一个对象具有一个接受三个参数的、名为[]=的方法，那么位于方括号内的两个值将会被作为索引。下面的两个表达式是等价的：

```
o[x,y] = z
o.[]=(x,y,z)
```

4.5.4 缩写形式的赋值
Abbreviated Assignment

缩写形式的赋值是一种将赋值和其他操作结合在一起的赋值操作。

```
x += 1
```

+=并不是一个真正的 Ruby 操作符，上述表达式只是下述表达式的缩写：

```
x = x + 1
```

缩写形式的赋值不能和并行赋值相结合，它仅在只有一个左值和一个右值的情况下才是有效的。如果左值是一个常量，那就不应该使用它，因为这会对常量重新赋值并导致一个警告。但是，当左值是一个属性时，是可以采用缩写形式的赋值的。下面的两个表达式是等价的：

```
o.m += 1
o.m=(o.m()+1)
```

当左值是一个数组元素时，缩写形式的赋值也是有效的。下面两个表达式是等价的：

```
o[x] -= 2
o.[]=(x, o.[](x) - 2)
```

这段代码使用的是-=而不是+=，正如你可能希望的那样，-=这个伪操作符会在其左值上减去右值。

除了+=和-=，还有 11 个伪操作符可用于缩写形式的赋值。它们在表 4-1 中被列出。请注意，它们本身并不是操作符，而只是一些使用别的操作符的简写表达式。本章的后续部分将详细描述这些操作符的意义。此外，如我们即将见到的，那些被调用的操作符本身也被定义为方法。比如，如果一个类定义了一个名为+的方法，那么+=的意义也会随之改变，而且这改变将影响到该类的所有实例。

表 4-1：缩写形式的赋值伪操作符

赋值操作	展开形式
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`
`x **= y`	`x = x ** y`
`x &&= y`	`x = x && y`
`x ||= y`	`x = x || y`
`x &= y`	`x = x & y`
`x |= y`	`x = x	y`
`x ^= y`	`x = x ^ y`
`x <<= y`	`x = x << y`
`x >>= y`	`x = x >> y`

||=习惯用法

正如本节开头所介绍的，最常用的缩写形式的赋值操作是+=，它会增加一个变量的值。此外，-=也常常被用于减少变量的值。其他的伪操作符则用得比较少。然而还有一个习惯用法值得一提。假定你要编写一个方法完成如下事情：先计算一些值，然后将它们添加到一个数组里，最后返回该数组。你希望那个用于存放结果值的数组可以由用户来指定。但是如果用户没有指定一个数组，那么你希望可以创建一个新的空数组。你可以使用下面这行代码：

```
results ||= []
```

稍微考虑一下。它可以被展开为：

```
results = results || []
```

如果你了解其他语言里的||操作符，或者你通过先前的阅读已经了解了 Ruby 中的||操作符，那么你就会知道这个赋值操作的值等于 results 的值，除非 results 为 nil 或 false。如果那样，该赋值操作的值就会是一个新的空数组。这意味着，这个缩写形式的赋值操作不会修改 results，除非它是 nil 或 false，如果那样，results 会被赋值为一个新的空数组。

但是事实上，缩写形式的||=的行为和上面展开式的行为稍有不同。如果||=的左值不是 nil 或 false，那么就不会有任何的赋值操作发生。如果左值是一个属性或数组元素，那么本来用于进行赋值操作的可变方法也不会被调用（译注 2）。

4.5.5 并行赋值

Parallel Assignment

并行赋值是一种赋值表达式，它可能具有多个左值，也可能具有多个右值，或者同时具有多个左值和右值。多个左值之间采用逗号进行分隔，多个右值之间也是如此。左值和右值都可以以*为前缀。虽然*不是一个真正的操作符，但是有时我们也称其为**展开操作符**（*splat operator*）。本节后面将介绍*的含义。

大多数并行赋值表达式都简单易懂，其含义也显而易见，但是也存在一些复杂的情况。接下来本书将解释所有可能的情况。

4.5.5.1 左值和右值的数量相等

左值和右值的数量相等的并行赋值是最简单的：

```
x, y, z = 1, 2, 3   # x=1; y=2; z=3
```

在这种情况下，第一个右值将赋给第一个左值，第二个右值将赋给第二个左值，以此类推。

这些赋值操作是以并行的方式有效进行的，而不是串行的。比如，下面两行代码是不同的：

```
x,y = y,x         # Parallel: swap the value of two variables
x = y; y = x      # Sequential: both variables have same value
```

4.5.5.2 一个左值，多个右值

当只有一个左值但是有多个右值时，Ruby 会创建一个数组来容纳所有右值，并且将这个数组赋给左值：

```
x = 1, 2, 3       # x = [1,2,3]
```

你可以在左值前放一个*，这不会改变这个赋值操作的意义，也不会改变返回值。

译注 2：对于 results = results || []这个展开式，无论 results 的值如何，都会发生赋值操作。

如果你不希望多个右值被组合进一个数组，那么就在左值后接一个逗号。即使在逗号之后没有左值了，这样做也会使 Ruby 表现得好像有多个左值一样。

```
x, = 1, 2, 3     # x = 1; other values are discarded
```

4.5.5.3 多个左值，一个数组右值

当有多个左值，但是只有一个右值时，Ruby 试图将右值扩展成一个值列表，然后再进行赋值。如果右值是一个数组，Ruby 扩展该数组，这样一来每个数组元素都会成为一个右值。如果右值不是一个数组，但是它实现了 to_ary 方法，那么 Ruby 就调用此方法并且扩展其返回的数组：

```
x, y, z = [1, 2, 3]  # Same as x,y,z = 1,2,3
```

并行赋值在经过了这样的转换之后，就具有多个左值和零个（如果被展开的数组是空的）或多个右值。如果左值和右值的数量相等，那么赋值操作会像 4.5.5.1 节描述的那样进行；如果数量不同，那么赋值操作将会像 4.5.5.4 节描述的那样进行。

我们可以使用前面介绍的后接逗号的小技巧把一个普通的非并行赋值操作转换成一个并行赋值操作，而且自动地拆开右侧的数组。

```
x = [1,2]    # x becomes [1,2]: this is not parallel assignment
x, = [1,2]   # x becomes 1: the trailing comma makes it parallel
```

4.5.5.4 左值和右值的数量不同

如果左值比右值多，而且没有使用展开操作符，那么第一个右值将会被赋给第一个左值，第二个右值被赋给第二个左值，以此类推，直到所有的右值都已经被赋给左值了。接下来，每个剩下的左值都将被赋值为 nil，其原有的值被覆盖。

```
x, y, z = 1, 2  # x=1; y=2; z=nil
```

如果右值比左值多，而且没有使用展开操作符，那么右值将被依次赋给左值，而且剩余的右值被丢弃：

```
x, y = 1, 2, 3 # x=1; y=2; 3 is not assigned anywhere
```

4.5.5.5 展开操作符

如果一个右值以*开头，那么就意味着它是一个数组（或一个类似于数组的对象），而且它的每个元素都应该是一个右值。首先数组的元素会替换该数组在原右值列表中的位置，然后进行如上节所述的赋值操作：

```
x, y, z = 1, *[2,3]  # Same as x,y,z = 1,2,3
```

Ruby 1.8 里，在一个赋值操作中，*只能出现在最后一个右值的前面，但是在 Ruby 1.9 里，一个并行赋值的右值列表里可以出现任意多个*，而且它们可以出现在列表的任何位置。但

无论是 Ruby 1.8 还是 1.9，试图在一个嵌套数组前放置两个*都是非法的：

```
x,y = **[[1,2]]   # SyntaxError!
```

在 Ruby 1.8 里，数组、范围及哈希右值可以被展开。在 Ruby 1.9 里，数组、范围及枚举器（enumerator，请参见第 5.3.4 节）右值可以被展开。除了这些类之外，如果你在一个属于其他类的值前面放置一个*，那么展开的结果就是这个值本身。你也可以定义自己的可展开类型。在 Ruby 1.8 里，可以通过定义 to_ary 方法来实现，该方法将返回一个值数组。在 Ruby 1.9 里，将方法名由 to_ary 改为 to_splat 即可。

当在一个左值前放置一个*时，意味着所有多出来的右值都会被放入一个数组并且赋给该左值。赋给该左值的值总是一个数组，它可以包含零个、一个或多个元素：

```
x,*y = 1, 2, 3    # x=1; y=[2,3]
x,*y = 1, 2       # x=1; y=[2]
x,*y = 1          # x=1; y=[]
```

在 Ruby 1.8 里，*只能出现在最后一个左值的前面。在 Ruby 1.9 里，虽然并行赋值的左侧也只能出现一个*，但是它可以出现在左值列表的任何位置：

```
# Ruby 1.9 only
*x,y = 1, 2, 3    # x=[1,2]; y=3
*x,y = 1, 2       # x=[1]; y=2
*x,y = 1          # x=[]; y=1
```

请注意，在一个并行赋值表达式的两侧可能都会出现 splat：

```
x, y, *z = 1, *[2,3,4]  # x=1; y=2; z=[3,4].
```

最后，回忆一下之前我们描述的，两个简单的并行赋值，它们要么只有一个左值，要么就只有一个右值，它们表现出来的行为就好像在左值或右值之前有一个*一样。在那两个例子里，显式地在左值或右值前加上一个*也不会有额外的效果。

4.5.5.6 并行赋值中的圆括号

可以在并行赋值的左侧使用圆括号来进行“子赋值”，这应该是它最不为人知的特性之一。如果将两个或多个左值放在一对圆括号里构成一个组，那么它将被作为一个左值来处理。一旦确定了和它对应的右值，它就会递归地应用并行赋值的规则——那些右值被赋给那些位于圆括号中的左值。考虑下面的赋值操作：

```
x,(y,z) = a, b
```

这实际上是同时进行两个赋值操作：

```
x = a
y,z = b
```

但需要注意的是，第二个赋值本身就是一个并行赋值。由于我们在左侧使用了圆括号，所以会进行一个递归的并行赋值。为了让这个操作能够进行，b必须是一个诸如数组或枚举器的可展开对象。

下面这些具体的例子能更清晰地展示这个特性。请注意，左侧的圆括号会“拆开（unpack）”右侧的嵌套数组中与其对应的那一层：

```
x,y,z = 1,[2,3]                 # No parens: x=1;y=[2,3];z=nil
x,(y,z) = 1,[2,3]               # Parens: x=1;y=2;z=3

a,b,c,d = [1,[2,[3,4]]]         # No parens: a=1;b=[2,[3,4]];c=d=nil
a,(b,(c,d)) = [1,[2,[3,4]]]     # Parens: a=1;b=2;c=3;d=4
```

4.5.5.7 并行赋值的值

并行赋值表达式的返回值是所有右值构成的数组（在右值中所有的展开操作符展开之后）。

并行赋值和方法调用

请注意，如果一个并行赋值紧跟在一个方法名之后，那么Ruby解释器不会将并行赋值中的逗号解释成左值和右值之间的分隔符，而解释成该方法调用的参数分隔符。如果你想要测试并行赋值的返回值，你可能会像下面这样编写代码，将其打印出来：

```
puts x,y=1,2
```

但是，这并不会产生你所期望的结果：Ruby认为你调用了puts方法，并传给它三个参数：x，y=1，以及2。接下来，你或许会尝试着将并行赋值放到一对圆括号中：

```
puts (x,y=1,2)
```

这样也不行，因为括号本身也会被解释成方法调用的一部分（虽然Ruby会抱怨在方法名称和起始圆括号之间存在空格）。为了达成目的，你必须采用嵌套的圆括号：

```
puts((x,y=1,2))
```

作为Ruby语法的表达能力的一部分，这是一种极端特殊的情形。幸运的是，须要使用这种语法的场合非常少。

4.6 操作符

Operators

在Ruby语言中，一个**操作符**是一个标记，它代表了一种在一个或多个操作数上进行的操作（比如相加或比较）。操作数本身是表达式，操作符使我们能够将这些操作数表达式组合成更大的表达式。比如，数字字面量2和操作符+能够组合成表达式2+2。在下面的例子里，

通过乘法操作符和小于操作符将一个数值字面量、一个方法调用及一个变量引用表达式组合成了一个新的表达式：

```
2 * Math.sqrt(2) < limit
```

本节后面的表 4-2 总结了 Ruby 表达式，而且接下来的小节会对它们进行详细的描述。为了完全理解操作符，你须要先理解操作符的元数（arity）、优先级及结合性。

操作符的**元数**指的是它的操作数的个数。一元操作符期望一个操作数，二元操作符期望两个操作数，三元操作符（这是唯一的一个）期望三个操作数。每个操作符的元数被列于表 4-2 中标记为“N”的那一栏。请注意操作符+和-都具有一元和二元两种形式。

操作符的**优先级**指定了一个操作符和它的操作数绑定得有多“紧密”，而且会影响到一个表达式的求值顺序。比如，考虑以下表达式：

```
1 + 2 * 3     # => 7
```

乘法操作符的优先级要高于加法操作符，因此乘法操作被先执行，整个表达式的值为 7。表 4-2 也是按优先级从高到低的顺序排列的。请注意，布尔操作 AND、OR 及 NOT 都对应于一个较高优先级和较低优先级的操作符。

操作符的优先级只是指定了一个表达式的默认求值顺序，你可以使用圆括号来分组子表达式并且指定你的求值顺序。比如：

```
(1 + 2) * 3   # => 9
```

当一个表达式中出现多个同样的操作符（或多个拥有同样优先级的操作符）时，它们之间的求值顺序将由操作符的**结合性**来决定。表 4-2 的第 A 列指定了每个操作符的结合性：值“L”表明对应的表达式将从左到右进行求值；值“R”意味着对应的表达式将从右到左来求值；而值“N”则意味着对应的操作符**不具有结合性**，不能在一个表达式里多次使用，除非有圆括号来指定求值顺序。

大多数算术操作符都是左结合的，这意味着 `10-5-2` 将以 `(10-5)-2` 的方式进行求值，而不是 `10-(5-2)`。另一方面，求幂操作是右结合的，因此 `2**3**4` 将以 `2**(3**4)` 的方式求值。赋值是另一个右结合的操作符。在表达式 `a=b=0` 中，`0` 首先会被赋给变量 `b`，然后表达式 `b=0` 的值（也是 `0`）会被赋给变量 `a`。

Ruby 将许多操作符都作为方法来实现，这使类可以为它们的操作符定义新的含义。表 4-2 的第 M 列表明了哪些操作符是方法：被标记为“Y”的操作符被实现为方法，可以被重定

义；而标记为“N”的方法则不能被重定义。一般来说，类会定义它们自己的算术、排序及相等性操作符，但是它们也许不会重定义各种布尔操作符。我们根据这些操作符在Ruby标准类中最常见的用途对其进行分类，其他类或许会为这些操作符定义不同的含义。比如，+操作符进行算术加运算，并被归类为算术操作符，但是它也被用于连接字符串和数组。一个基于方法的操作符将作为其左侧操作数的一个方法被调用（在一元操作符的情况下，左侧操作数将是其唯一的操作数），右侧操作数将作为实参被传递给该方法。你可以查找一个类的任何基于方法的操作符定义，和你查找该类的其他方法的定义一样。比如，你可以使用*ri*来查找字符串类的*操作符的定义：

```
ri 'String.*'
```

为了定义一元+和一元-操作符，你可以使用+@和-@作为方法名来避免和二元+及二元-操作符之间的二义性。操作符!=和!~被定义成==及=~操作符的反义词。在Ruby 1.9中，你可以重定义!=和!~，但在之前版本的Ruby中你不可以这样做。Ruby 1.9还允许重定义一元!操作符。

表4-2：按优先级（从高到低）排列的Ruby操作符，以及其元数（N）、结合性（A）及可定义性（M）

操作符	N	A	M	对应的操作
! ~ +	1	R	Y	布尔值NOT，按位补，一元加[a]
**	2	R	Y	求幂操作
-	1	R	Y	一元减（使用-@来定义）
* / %	2	L	Y	乘法、除法、取模（取余）
+ -	2	L	Y	加法（或者连接）、减法
<< >>	2	L	Y	按位左移（或添加）、按位右移
&	2	L	Y	按位与
| ^	2	L	Y	按位或、按位异或
< <= >= >	2	L	Y	排序
== === != =~ !~ <=>	2	N	Y	相等性、模式匹配、比较[b]
&&	2	L	N	布尔与
||	2	L	N	布尔或
.. ...	2	N	N	范围的创建，以及名为flip-flop的布尔表达式
?:	3	R	N	条件操作符
rescue	2	L	N	异常处理操作符
= **= *= /= %= += -= <<= >>= &&= &= ||= |= ^=	2	R	N	赋值操作符

续表

操作符	N	A	M	对应的操作
defined?	1	N	N	测试变量的定义和类型
not	1	R	N	布尔值非（低优先级）
and or	2	L	N	布尔值与，布尔值或（低优先级）
if unless while until	2	N	N	条件和循环修饰符

[a] 在 Ruby 1.9 之前，不能重定义!。使用+@来定义一元加。
[b] 在 Ruby 1.9 之前，不能重定义!=和!~。

4.6.1 一元+和-
Unary + and -

一元减操作符将改变其数值实参的符号，也可以在数值操作数前加上一元加操作符，但是它不会对该操作数带来任何影响——它直接将其操作数的值返回。提供一元加操作符是为了和一元减操作符相对称，当然，它也可以被重定义。请注意，一元减操作符的优先级比一元加操作符的略低一点，这将在下一节讲解**操作符时被提到。

这些一元操作符对应的方法名是-@和+@。在如下场合将会用到这些名字：重定义这些操作符、将这些操作符作为方法进行调用、查找关于这些操作符的文档。这些特殊的名称对于消除一元加、减操作符与二元加、减操作符之间的歧义是非常重要的。

4.6.2 求幂操作符：**
Exponentiation: **

将按第二个参数所指定的幂，对第一个参数执行求幂操作，你还可以通过将第二个参数设为一个分数的方式来求第一个参数的根。比如，x 的立方根为 x(1.0/3.0)。类似地，x**-y 和 1/(x**y)是一样的。**操作符是右结合的，所以 x**y**z 和 x**(y**z)是一样的。最后，由于**的优先级高于一元减操作符，所以-1**0.5 和-(1**0.5)是一样的。如果你想求-1 的平方根，那么你必须使用圆括号：(-1)**0.5。(结果并不是一个数字，而且表达式被求值为 NaN。)

4.6.3 算术操作符：+、-、*、/及%
Arithmetic: +, -, *, /, and %

操作符+、-、*及/可以对所有数值类执行加法、减法、乘法及除法操作，可以通过取模操作符%来得到余数。除数为 0 的整数除法操作将抛出 ZeroDivisionError。除数为 0 的浮点数除法操作将返回正或负 Infinity。被除数和除数均为 0 的浮点数除法操作将返回 NaN。请参见第 3.1.3 节获得更多关于 Ruby 整数和浮点数算术操作的信息。

String 类采用+操作符进行字符串连接操作，采用*操作符进行字符串重复操作，采用%操作符将 sprintf 的实参插入到一个字符串中，从而替换该字符串中对应的占位符。

Array 类使用+操作符进行数组之间的连接操作，使用-操作符来进行数组间的减操作。取决于第二个实参的类型，Array 将按不同的方式使用*操作符。如果一个数组与一个数字 N“相乘”，那么结果将是一个新数组，其内容由原数组的内容重复 N 次构成。但是当一个数组与一个字符串相乘时，其结果等价于将该字符串作为实参来调用该数组的 join 方法。

4.6.4 移位和追加操作符：<<和>>

Shift and Append: << and >>

类 Fixnum 和 Bignum 定义了<<和>>操作符，用于对其左侧操作数的底层位模式，进行向左或向右的移位操作。它们的右侧操作数表明了将要移动的位数，如果是负值将会进行反向移动：左移-2 和右移 2 是等价的。当对一个 Fixnum 进行左移操作时，其高端二进制位是永远不会被“移除”的。如果一个移位操作的结果超出了 Fixnum 的取值范围，那么将返回一个 Bignum 值。右移操作则总是丢弃其实参的低端二进制位。

将一个数字左移 1 位相当于将其乘以 2，将一个数字右移一位则相当于将其除以 2。在下面的例子中，先以二进制记数法来表示一些数字，然后再把操作结果转换回二进制形式：

```
(0b1011 << 1).to_s(2)     # => "10110"   11 << 1 => 22
(0b10110 >> 2).to_s(2)    # => "101"     22 >> 2 => 5
```

<<操作符还可以作为追加操作符，而且这种形式更常见。String、Array 及 IO 类都定义了一个这样的<<操作符，此外还有其他一些来自标准库的“可追加的”类也定义了该操作符，比如 Queue 和 Logger 类：

```
message = "hello"          # A string
messages = []              # An empty array
message << " world"        # Append to the string
messages << message        # Append message to the array
STDOUT << message          # Print the message to standard output stream
```

4.6.5 补、并、交操作符：~、&、|及^

Complement, Union, Intersection: ~, &, |, and ^

Fixnum 和 Bignum 定义了这些操作符，用于指定按位非、与、或，以及异或操作。~是一个高优先级的一元操作符，其他的几个都是中等优先级的二元操作符。

~将其整数操作数的每个 0 位设置成 1，而将每个 1 位设置成 0，由此产生一个数的二进制 1 的补。对于任意整数 x 来说，~x 和-x-1 是一样的。

&是针对两个整数的二进制与操作符。只有在两个操作数对应的位均为 1 时，结果中相应的位才会被设置成 1。比如：

```
(0b1010 & 0b1100).to_s(2)  # => "1000"
```

|是针对两个整数的二进制或操作符。只要两个操作符对应的位中有一个为 1，结果中相应的位就会被设置成 1。比如：

```
(0b1010 | 0b1100).to_s(2)  # => "1110"
```

^是针对两个整数的二进制异或（排他的或）操作符。只有在两个操作数对应的位中有且仅有一个为1时，结果中相应的位才会被设置成1。比如：

```
(0b1010 ^ 0b1100).to_s(2)  # => "110"
```

其他类也可使用这些操作符，通常情况下，它们的含义都会兼容于那些类的逻辑与、或、非操作。数组类使用&和|进行集合交和并操作。当&作用于两个数组时，它返回一个新数组，该数组只包含了那些同时出现在两个原数组中的元素。当|作用于两个数组时，它返回一个新数组，该数组包含了那些出现在任意一个原数组中的元素。请参见第9.5.2.7节的细节和例子。

TrueClass、FalseClass及NilClass定义了&、|及^（但是没有~），所以这些操作符也可以作为布尔操作符。但是请注意这并不是正确的方式。布尔操作符&&和||（稍后将在第4.6.8节描述）能够更高效地作用于布尔操作数，因为它们不会对右侧操作数求值，除非该操作数的值影响到整个布尔操作的值。

4.6.6 比较操作符：<、<=、>、>=和<=>

Comparison: <, <=, >, >=, and <=>

有些类为它们的值定义了一种自然顺序。比如：数字按大小排序；字符串按字母表顺序来排序；日期则按年代顺序来排序。小于（<），小于等于（<=），大于等于（>=）及大于（>）操作符用于判断两个值的相对顺序。如果判断对了，那么它们的值为true，否则就为false。（此外，当参数类型不兼容时，它们通常会抛出一个异常。）

类可以逐一地定义各个比较操作符。但是更为简单和常见的做法是为一个类定义一个<=>操作符。这是一个多用途的比较操作符，其返回值将表明两个操作数的相对顺序。如果左操作数小于右操作数，那么<=>将返回-1；如果左操作数更大，<=>返回+1；如果两个操作数相等，<=>返回0；如果两个操作数不可比较，那么它返回nil。（注2）一旦定义了<=>操作符，一个类就可以简单地包括Comparable模块，它在<=>的基础上定义了其他的比较操作符（包括==操作）。

Module类值得特别提一下：它实现了那些用于表明子类关系的比较操作符（Module是Class的超类）。对于类A和类B，如果A是B的子类或更远一些的后代类，那么A < B将

注2：<=>操作符的有些实现会返回一个小于0的值，而不是-1；返回一个大于0的值，而不是+1。如果你要实现<=>，你应该让其返回-1、0或+1。但是如果你要使用别人实现的<=>，那么你应该测试其返回值是否小于或大于0，而不是假定结果总会是-1、0或+1。

为 true。在这种情况下，“小于”意味着“更特化”或“类型更窄”。如我们将在第 7 章学到的，当声明一个子类时，也会用到<字符。

```
# 声明类 A 为类 B 的子类
class A < B
end
```

Module 还定义了>操作符，它的行为与<类似，只是在操作数之间进行了对调。Module 还定义了<=和>=，所以，如果两个操作数属于同一个类，这些操作符就会返回 true。关于这些 Module 比较操作符的最有趣的事情是，Module 只为它的值定义了部分排序（partial ordering）。请考虑一下 String 和 Numeric 类。它们都是 Object 的子类，而且都不是对方的子类。既然如此，当两个操作数彼此无关时，比较操作符就会返回 nil，而不是 true 或 false。

```
String < Object      # true: String is more specialized than Object
Object > Numeric     # true: Object is more general than Numeric
Numeric < Integer    # false: Numeric is not more specialized than Integer
String < Numeric     # nil: String and Numeric are not related
```

如果一个类为它的值定义了全排序（total ordering），而且 a < b 不为 true，那么你可以肯定 a >= b 为 true。但是当一个类只定义了部分排序（partial ordering）时，比如 Module，那么你就不应该做这样的假设。

4.6.7 相等性操作符：==、!=、=~、!~和===

Equality: ==, !=, =~, !~, and ===

==是相等性操作符，它根据左侧操作数定义的“equal”方法测试两个操作数是否相等。!=操作符和==恰好相反：它调用==，然后返回一个与==返回值相反的值。在 Ruby 1.9 中，你可以重定义!=，但是在 Ruby 1.8 里不可以。请参见第 3.8.5 节获得 Ruby 中对象相等性的更多细节。

=~是模式匹配操作符。Object 类定义了这个操作符，且始终返回 false。String 类重定义了这个操作符，且期望一个 Regexp 作为右操作数。Regexp 类也重定义了这个操作符，且期望一个 String 作为右操作数。如果字符串和模式之间不匹配，那么这些操作符都会返回 nil。如果字符串和模式相匹配，那么这些操作符将返回开始匹配处的整数索引值。（请注意，在布尔表达式中，nil 表现得像 false，而任何整数都表现得像 true。）

!~操作符和=~恰好相反：它调用=~，而且在=~返回 nil 时返回 true，在=~返回一个整数时返回 false。在 Ruby 1.9 中，你可以重定义!~，但是在 Ruby 1.8 里不可以。

===操作符是条件相等性操作符，它被 case 语句隐式地使用（请参见第 5 章）。对它的显式使用要远远少于==。Range、Class 和 Regexp 类定义了这个操作符，作为一种成员关系或模式匹配操作符。其他的类继承了 Object 类的定义，只是简单地调用==操作符而已。请参见第 3.8.5 节。请注意，不存在!==操作符，如果想对===求反，那么你必须自己完成。

4.6.8 布尔操作符：&&、||、!、and、or、not

Boolean Operators: &&, ||, !, and, or, not

Ruby 将布尔操作符内建于语言当中，而不是基于方法，比如，类不能定义它们自己的&&方法。这样做的原因是，布尔操作符可以用于任何值，必须对任何类型的操作数都表现出一致的行为。Ruby 定义了特殊的 true 和 false 值，但是没有一个 Boolean 类型。就所有布尔操作符而言，值 false 和 nil 被认为是 false，此外所有其他的值，包括 true、0、NaN、""、[]及{}，都被认为是 true。请注意，!是一个例外，你可以在 Ruby 1.9 中重定义这个操作符（但是在 Ruby 1.8 中不行）。还有一点值得注意的是，你可以定义名为 and、or，以及 not 的方法，但是它们只是普通方法而已，不会改变那些与它们同名操作符的行为。

将 Ruby 的布尔操作符定义为语言的一个核心部分，而不是可重定义的方法的另一个原因是二元操作符是“短路式的”。如果一个操作的值可以完全由其左侧操作数决定，那么右侧操作数将被忽略，根本不会对其进行求值。如果右侧操作数是一个具有副作用的表达式（比如一个赋值操作，或者对一个带副作用的方法的调用），那么基于左侧操作数的值，该副作用可能出现也可能不会出现。

&&是一个布尔与操作符。如果它的左右操作数的值都是真值，那么它返回一个真值，否则它就返回一个假值。请注意，这里说的是“一个真值”和“一个假值”，而不是“true 值”和“false 值”。在表达式中，&&常被用于连接诸如==和<之类的比较操作符，比如：

```
x == 0 && y > 1
```

比较和相等性操作符的结果将为值 true 和 false，在这种情况下，&&操作符将作用于真正的布尔值上。但是情况并非始终如此，我们还可以像下面这样使用&&操作符：

```
x && y
```

在这种情况下，x 和 y 可以是任何东西。该表达式的值要么是 x 的值，要么是 y 的值。如果 x 和 y 都是真值，那么 y 的值就是该表达式的值。如果 x 是一个假值，那么 x 就是表达式的值。否则，y 必定是一个假值，那么 y 就是该表达式的值。

下面说明&&是如何工作的。首先，它对其左操作数进行求值。假如结果是 nil 或 false，那么它将直接返回该值而不再对右操作数求值；否则，左操作数将是一个真值，整个&&操作符的值取决于右侧操作数的值，在这种情况下，&&操作符对其右操作数进行求值并返回该值。

你也可以在代码中有效地利用&&也许会忽略其右侧操作符这一事实。考虑下面这个表达式：

```
x && print(x.to_s)
```

此代码运行的结果是将 x 的值以字符串的形式打印出来，但只有在 x 不为 nil 或 false 的情况下才会这样（注 3）。

||操作符返回其操作数之间的布尔或操作的结果。只要任意一个操作数为真值，它就会返回一个真值。如果两个操作数都是假值，那么它返回一个假值。类似于&&，如果||的右操作数对于该操作的结果没有影响，那么||会忽略它。||的工作原理如下：首先，它会对其左操作数求值。如果结果不是 nil 或 false，那么它返回该值。否则，它对其右操作数进行求值并返回该值。

||可用于连接多个比较或相等性表达式：

```
x < 0 || y < 0 || z < 0   # Are any of the coordinates negative?
```

在这种情况下，||的操作数将是真正的值 true 或 false，而且结果也将是值 true 或 false。但是||也可以作用于 true 和 false 之外的值。一个对||的习惯用法是返回一系列候选值中的第一个非 nil 值：

```
# If the argument x is nil, then get its value from a hash of user preferences
# or from a constant default value.
x = x || preferences[:x] || Defaults::X
```

请注意，&&的优先级高于||。考虑下面这个表达式：

```
1 || 2 && nil     # => 1
```

上面这行代码首先执行&&操作，并且该表达式的结果是 1。如果先执行||操作，那么结果将为 nil：

```
(1 || 2) && nil   # => nil
```

!操作符进行一个一元布尔非操作。如果操作数是 nil 或 false，那么!操作符将返回 true。否则!返回 false。

!操作符具有最高的优先级，这意味着如果你想计算一个表达式的逻辑反，而该表达式本身也包含一些操作符，那么你必须使用圆括号：

```
!(a && b)
```

顺便提一下，布尔逻辑的法则之一允许上述表达式写成下面这样：

```
!a || !b
```

and、or 及 not 操作符分别是&&、||及!的低优先级版本，使用这些操作符的理由之一是，它们的名字是英文单词，这将使你的代码易于阅读。举个例子，试着读一下下面这行代码：

```
if x > 0 and y > 0 and not defined? d then d = Math.sqrt(x*x + y*y) end
```

注 3：可以像这样写这个表达式并不意味着应该这样写。在第 5 章，我们将见到这个表达式可以更好地写成：

```
print(x.to_s) if x
```

另一个理由就是它们的优先级比赋值操作符更低，这意味着你可以编写下面这样的布尔表达式，不断地对变量赋值，直到遇见一个假值。

```
if a = f(x) and b = f(y) and c = f(z) then d = g(a,b,c) end
```

如果使用&&而不是 and，就无法编写这样的表达式。

你需要注意的是，and 和 or 的优先级相同（not 的优先级稍高一点）。因为 and 和 or 具有同样的优先级，而&&和||具有不同的优先级，所以下面两个表达式的结果不同：

```
x || y && nil          # && is performed first   => x
x or y and nil         # evaluated left-to-right => nil
```

4.6.9 范围和 Flip-Flops：..和...

Ranges and Flip-Flops: .. and ...

之前我们在第 3.5 节介绍 Range 的字面量语法时，曾见过..和...。当一个范围的开始和结束端点是整数字面量时，比如 1..10，Ruby 解释器将创建一个 Range 字面量对象。但是如果开始和结束端点的表达式比整数字面量更复杂时，比如 x..2*x，那么再称其为 Range 字面量就不准确了，我们称其为范围创建表达式。因此，..和...不仅是 range 字面量语法，还是操作符。

..和...操作符不是基于方法的，因此无法重定义。它们的优先级相对较低，这就意味着在使用它们时，通常不需要给左操作数和右操作数加上圆括号：

```
x+1..x*x
```

因为这些操作符的值都是一个 Range 对象，所以 x..y 和下面这行代码是一样的：

```
Range.new(x,y)
```

此外，x...y 和下面这行代码是一样的：

```
Range.new(x,y,true)
```

4.6.9.1 称为 flip-flop 的布尔表达式

当..和...操作符被用在一个条件式（比如 if 语句），或者一个循环（比如 while 循环）中时（第 5 章将介绍更多有关条件式和循环的内容），它们不会创建 Range 对象。相反地，它们将创建一种特殊的布尔表达式，名为 *flip-flop*。和比较及相等性表达式一样，一个 flip-flop 表达式的值也为 true 或 false。但是一个 flip-flop 表达式的特殊之处在于，它的值依赖于此前的求值结果，这就意味着一个 flip-flop 表达式具有与其关联的状态，它必须保存此前求值的信息。由于它具有状态，所以你可能会认为一个 flip-flop 是一个某种类型的对象，但事实上 flip-flop 并不是对象，而是 Ruby 表达式。Ruby 解释器在处理完一个 flip-flop 表达式之后，将为它在内部存储一个解析后的表现形式，其中就保存了该表达式的状态（只是作为一个布尔值）。

有了这些背景知识之后，请考虑下面代码中的 flip-flop。第一个..创建一个 Range 对象，第二个..则创建一个 flip-flop 表达式：

```
(1..10).each {|x| print x if x==3..x==5 }
```

在一个由条件式或循环所构成的上下文中，一个 flip-flop 由两个通过..操作符相连的布尔表达式构成。除非其左侧表达式为 true，否则一个 flip-flop 表达式就是 false，而且在左侧表达式为 true 之前，它的值都会是 false。一旦该表达式为 true，那么它就会“flips”到一个持久的 true 状态。它会保持该状态，而且对其后续的求值也返回 true，直到其右侧表达式成为 true 为止。如果其右侧表达式为 true 了，那么该 flip-flop 就会“flops”回一个持久的 false 状态，对其后续的求值也返回 false，直到其左侧表达式再次成为 true 为止。

在上面的代码例子中，该 flip-flop 被反复求值，相应的 x 的值也从 1 一直增加到 10。起初，它的状态为 false，而且在 x 等于 1 和 2 的时候一直是 false。当 x 等于 3 的时候，该 flip-flop 的状态就成为 true 而且返回 true。在 x 等于 4 和 5 的时候，它一直返回 true。但是当 x 等于 5 的时候，该 flip-flop 的状态又回到了 false，而且对于后续的 x，它总是返回 false。上述代码的执行结果是打印出 345。

可以使用..或...来编写 flip-flop。其差别在于：当一个.. flip-flop 为 true 时，它会返回 true，并且测试它的右侧表达式以决定是否需要将其内部状态设置回 false；而对于... flip-flop 来说，它等到下一次求值的时候才测试其右侧表达式。考虑下面的两行代码：

```
# Prints "3". Flips and flops back when x==3
(1..10).each {|x| print x if x==3..x>=3 }
# Prints "34". Flips when x == 3 and flops when x==4
(1..10).each {|x| print x if x==3...x>=3 } # Prints "34"
```

Flip-flop 是一个非常晦涩的 Ruby 特性，因此最好不要在你的代码中使用它。但是它们并不是 Ruby 所独有的，Ruby 从 Perl 那里继承了这个特性，而 Perl 则从 Unix 的文本处理工具 *sed* 和 *awk* 那里继承了这个特性（注 4）。Flip-flop 的初衷是在一个开始模式和一个结束模式之间匹配一个文本文件的行，而且这仍然是使用它们的有效方式。下面的这个简单的 Ruby 程序展示了一个 flip-flop，它逐行地从一个文本文件中读取内容，打印出含有“TODO”的行。它会不断地打印文本行，直到读入一个空行为止：

```
ARGF.each do |line|   # For each line of standard in or of named files
  print line if line=~/TODO/..line=~/^$/ # Print lines when flip-flop is true
end
```

很难正式地描述一个 flip-flop 的精确行为，通过研究那些行为与 flip-flop 等价的代码，能够更容易地理解 flip-flop。下面的函数，其行为与 x==3..x==5 这个 flip-flop 相同，它将左侧和右侧条件式硬编码到函数当中，而且使用一个全局变量保存 flip-flop 的状态：

注 4：..会创建一个 *awk* 风格的 flip-flop，而...则会创建一个 *sed* 风格的 flip-flop。

```
$state = false            # Global storage for flip-flop state
def flipflop(x)           # Test value of x against flip-flop
  if !$state              # If saved state is false
    result = (x == 3)     # Result is value of lefthand operand
    if result             # If that result is true
      $state = !(x == 5)  # Then saved state is not of the righthand operand
    end
    result                # Return result
  else                    # Otherwise, if saved state is true
    $state = !(x == 5)    # Then save the inverse of the righthand operand
    true                  # And return true without testing lefthand
  end
end
```

有了这个 flip-flop 函数，我们就可以编写下面的代码，它能像我们先前的例子那样打印出 345：

```
(1..10).each {|x| print x if flipflop(x) }
```

下面的函数模拟了 3 点 flip-flop x==3...x>=3 的行为：

```
$state2 = false
def flipflop2(x)
  if !$state2
    $state2 = (x == 3)
  else
    $state2 = !(x >= 3)
    true
  end
end

# Now try it out
(1..10).each {|x| print x if x==3...x>=3 }    # Prints "34"
(1..10).each {|x| print x if flipflop2(x) }   # Prints "34"
```

4.6.10 条件操作符: ?:

Conditional: ?:

?:操作符被称为条件操作符，它是 Ruby 当中唯一的一个三元（有三个操作数）操作符，第一个操作数位于问号前面，第二个操作数位于问号和分号之间，第三个操作数则位于分号之后。

?:操作符对其第一个操作数求值。如果第一个操作数的值不同于 false 或 nil，那么整个表达式的值就是第二个操作数的值；否则，如果第一个操作数是 false 或 nil，那么整个表达式的值就是第三个操作数的值。无论属于哪种情况，都会有一个操作数不被求值（如果它含有一些具有副作用的操作，比如赋值，那么就可能会有影响）。下面是一个使用该操作符的例子：

```
"You have #{n} #{n==1 ? 'message' : 'messages'}"
```

如你所见，?:操作符的行为就像压缩版的 if/then/else 语句。（Ruby 的 if 条件式将在第 5 章被描述。）第一个操作数就是那个将被测试的条件，类似于 if 后面的表达式；第二个操

作数类似于 then 后面的代码；第三个操作数则类似于 else 之后的代码。?:操作符和 if 语句之间的差别在于，if 语句允许其 then 和 else 从句包含任意数量的代码，但是?:操作符仅允许一个表达式。

?:操作符的优先级相当低，这意味着通常情况下都不必在操作数的周围放上圆括号。如果第一个操作数使用了 defind?操作符，或者第二个和第三个操作数进行了赋值操作，那么就需要将它们用圆括号括起来。由于 Ruby 允许方法名以问号结尾，所以如果?:操作符的第一个操作数以一个标识符结尾，那么你必须要么用圆括号将第一个操作数括起来，要么在第一个操作数和问号之间加上一个消除歧义的空格。如果你不这么做，那么 Ruby 解释器就会将?:操作符的问号理解成之前的标识符的一部分。比如：

```
x==3?y:z          # This is legal
3==x?y:z          # Syntax error: x? is interpreted as a method name
(3==x)?y:z        # Okay: parentheses fix the problem
3==x ?y:z         # Spaces also resolve the problem
```

?:操作符里的问号必须和第一个实参处于同一行。在 Ruby 1.8 中，分号必须和第二个实参处于同一行。但是在 Ruby 1.9 中，可以在分号前出现一个换行符。在这种情况下，你必须在分号后面接一个空格，这样一来它就不会引入一个符号字面量了。

表 4-2（出现在本章前面）指出?:操作符是右结合的，如果在一个表达式里使用了两次?:操作符，那么最右边的那个操作符会被当成一组：

```
a ? b : c ? d : e         # This expression...
a ? b : (c ? d : e)       # is evaluated like this..
(a ? b : c) ? d : e       # NOT like this
```

对于?:操作符来说，这种模棱两可的情况其实很少发生。在下面的例子中，使用了三个条件操作符来计算三个变量中的最大值。因为只有一种可能的解析方式，所以不须要使用圆括号（但是问号前面的空格还是需要的）。

```
max = x>y ? x>z ? x : z : y>z ? y : z
max = x>y ? (x>z ? x : z) : (y>z ? y : z)  # With explicit parentheses
```

4.6.11 赋值操作符
Assignment Operators

你已经在第 4.5 节阅读了有关赋值表达式的内容。对于那些表达式中的赋值操作符，在此还有一些值得一提的：首先，一个赋值表达式的值就是那个位于赋值操作符右侧的值（或一个值数组）。其次，赋值操作符是右结合的。这两点结合在一起才使得像下面这样的表达式能正常运行：

```
x = y = z = 0          # Assign zero to variables x, y, and z
x = (y = (z = 0))      # This equivalent expression shows order of evaluation
```

再次，赋值操作符的优先级非常低，这意味着几乎所有跟在赋值操作符后面的东西，都在赋值操作发生之前被求值。and、or 及 not 操作符是几个主要的例外情况。

最后，虽然不能像定义方法那样来定义赋值操作符，但是那些复合赋值操作符，比如+=，还是可以使用像+这样的可重定义操作符。重定义+操作符并不影响+=操作符所进行的赋值操作，但是会影响该操作符所进行的相加操作。

4.6.12 defined?操作符

The defined? Operator

define?是一个一元操作符，它测试其操作数是否已经被定义过了。通常情况下，如果使用一个未定义的变量或方法，那么就会产生异常。当位于 define?操作符右侧的表达式使用了一个未定义的变量或方法时（包含那些定义成方法的操作符），define?操作符只返回 nil 而不抛出异常。类似地，如果作为其操作数的表达式在一个不合适的上下文中（也就是说，当没有代码块可供 yield，或者没有超类方法可供调用时）使用了 yield 或 super，那么 define?操作符也返回 nil。理解下面这一点很重要：define?操作符并不真正地对其操作数表达式进行求值，而只测试一下是否**可以**对其求值且不产生错误。下面是 define?操作符的一种典型的使用方式：

```
# Compute f(x), but only if f and x are both defined
y = f(x) if defined? f(x)
```

如果操作数是已经定义的，那么 define?操作符就返回一个字符串。这个返回字符串的值通常不重要，真正重要的是它是一个真值——既不是 nil 也不是 false。但是，我们也可以通过检查这个操作符的返回值来了解一些右侧表达式的类型信息。表 4-3 列出了这个操作符的返回值列表：

表 4-3：define?操作符的返回值

操作数表达式的类型	返回值
对已经定义的局部变量的引用	"local-variable"
对已经定义的代码块局部变量的引用（仅限于 Ruby 1.8）	"local-variable (in-block)"
对已经定义的全局变量的引用	"global-variable"
在发生一个成功的匹配之后所定义的一些特殊的正则表达式全局变量，包括$&、$+、$`、$'，以及$1 到$9 的所有变量（仅限于 Ruby 1.8）	Name of variable, as a string
对已经定义的常量的引用	"constant"
对已经定义的实例变量的引用	"instance-variable"
对已经定义的类变量的引用	"class variable" (note no hyphen)
nil	"nil" (note this is a string)
true、false	"true", "false"

续表

操作数表达式的类型	返回值
self	"self"
当有一个相关联的代码块时，就是 yield（还可以参见 Kernel 模块的 block_given?方法）	"yield"
当所处的上下文允许时，就是 super	"super"
赋值（实际上并没有发生赋值）	"assignment"
方法调用，包括那些被定义成方法的操作符（实际上并没有发生方法调用，也无须传递正确数目的实参；还可以参见 Object.respond_to?）	"method"
任何其他有效的表达式，包括字面量和内建的操作符	"expression"
任何使用了未定义的编码或方法名的表达式，或者使用了不允许的 yield 或 super	nil

define?操作符的优先级非常低，如果你希望测试两个变量是否被定义了，请使用 and 而不是&&：

```
defined? a and defined? b     # This works
defined? a && defined? b      # Evaluated as: defined?((a && defined? b))
```

4.6.13 语句修饰符
Statement Modifiers

rescue、if、unless、while 及 until 分别是异常处理、条件式及循环语句，它们影响 Ruby 程序的控制流。它们还可以用作语句修饰符，就像下面这样：

```
print x if x
```

当采用这种修饰符形式的时候，我们可以把它们理解成这样一种操作符，其右侧表达式的值会影响左侧表达式的执行。（或者，在 rescue 修饰符中，左侧表达式的异常状态会影响右侧操作数的执行）

将上面这些关键字描述成操作符并没有太大用处。第 5 章将仔细描述它们的语句形式和修饰符形式。在表 4-2 中列出这些关键字只是为了展示它们相对于其他操作符的优先级。请注意，它们的优先级都很低，但是 rescue 语句修饰符的优先级要高于赋值操作符的优先级。

4.6.14 非操作符
Nonoperators

大多数 Ruby 操作符都是标点符号字符，Ruby 的语法也使用了一些不是操作符的标点符号字符。虽然我们已经在本书的其他地方见过（或将要见到）许多非操作符的标点符号，但还是在此回顾一下：

()

圆括号是方法定义和调用语法中的可选部分。将方法调用理解成一种特殊的表达式，比将()理解成方法调用操作符更好一些。圆括号还可用于分组，以此影响子表达式的求值顺序。

[]

方括号用于数组字面量，以及查询和设置数组和哈希值。在那种上下文中，它们是方法调用的语法糖，其行为在某种程度上就像可重定义的操作符(可以具有任意的元数)。请参见第 4.4 节和第 4.5.3 节。

{}

在代码块中，花括号是 do/end 的替代性选择，此外它还能用于哈希字面量中。无论在那种情况下，它们都不会表现得像一个操作符。

.和::

.和::用于限定修饰名字，将方法名与其调用对象分开，或者将常量名与其所在的模块分开。因为其右侧不是一个值而是一个标识符，所以它们不是操作符。

;、, 和 =>

这些标点符号都是分隔符而不是操作符。分号（;）用于分隔同一行中的多个语句；逗号（,）用于分隔方法参数、数组和哈希字面量中的元素；箭头（=>）则用于在哈希字面量中分隔哈希键和哈希值。

:

一个冒号用做符号字面量的前缀，此外在 Ruby 1.9 中它还用于哈希语法。

*、&和 <

这些标点符号在某些上下文中是操作符，在其他情况下则不是。在一个赋值或方法调用表达式中的数组前放一个*，将把该数组展开或拆开成单个元素。虽然有时我们称它为展开操作符，但它并不是一个真正的操作符，*a 并不能作为一个独立的表达式而存在。

在方法定义中，可以将&放在最后一个参数的前面，这会将任何传递给该方法的代码块赋给该参数。(请参见第 6 章。) 在方法调用时，我们还可以借助&将一个 proc 对象传递给一个方法，就好像该 proc 对象是一个代码块一样。

在类定义中，<用于指定当前类的超类。

第5章

语句和控制结构

Statements and Control Structures

请思考下面的 Ruby 程序，它将两个传递给它的命令行参数相加并输出结果：

```
x = ARGV[0].to_f      # Convert first argument to a number
y = ARGV[1].to_f      # Convert second argument to a number
sum = x + y           # Add the arguments
puts sum              # Print the sum
```

这个程序主要由变量赋值和方法调用构成。它纯粹的顺序执行方式令其格外简单，4 行代码逐条地执行，没有任何分支和重复。但是很少能有如此简单的程序。本章介绍了 Ruby 的控制结构，它们也被称为**控制流**，它们改变了程序的顺序执行方式。我们将涵盖如下内容：

- 条件式
- 循环
- 迭代器和代码块
- 控制流变更语句，比如 return 和 break
- 异常
- 特殊的 BEGIN 和 END 语句
- 被称为纤程（fiber）和连续体（continuation）的高深的控制结构

5.1 条件式
Conditionals

在任何程序设计语言中，最常见的控制结构就是条件式，这是一种让计算机有条件地执行代码的方式：只有在满足某些条件的情况下才执行相关代码。条件就是一个表达式——只要它的值不为 `false` 或 `nil`，就能满足条件。

Ruby 具有丰富的语法来表达条件式。下面将会描述各种语法选择。在编写 Ruby 代码时，你可以选择对于手头的工作来说最为优雅的一种。

5.1.1 if

`if` 是最简单的条件式。它具有多种形式，其中最简单的如下：

```
if expression
  code
end
```

如果（而且只有在这种情况下）*expression* 的值不是 `false` 或 `nil`，那么位于 `if` 和 `and` 之间的 *code* 将被执行。*code* 和 *expression* 之间必须有分界符，它可以是一个换行符，也

可以是一个分号，还可以是关键字 then（注 1）。下面是编写一个简单条件式的两种方式：

```
# If x is less than 10, increment it
if x < 10                    # newline separator
  x += 1
end
if x < 10 then x += 1 end    # then separator
```

你还可以使用 then 作为分隔符标记，并且在它后面接一个换行符，这样可以使代码更健壮，即使后来删除了换行符，这些代码仍然可以正常运行：

```
if x < 10 then
  x += 1
end
```

如果你熟悉 C 语言，或者熟悉那些语法源自 C 的语言，那么关于 Ruby 的 if 语句，你应该注意两个重要方面：

- 围绕着条件表达式的圆括号是可选的（而且通常都不用），Ruby 使用换行符、分号，或者关键字 then 对条件表达式和后续内容进行分隔。
- 即使按条件执行的代码体只包含一行代码，end 关键字也是必须的。下面将介绍 if 语句的修饰符形式（modifier form），它是一种简单的条件式，可以不包含 end。

5.1.1.1 else

if 语句可以包含一个 else 从句，用于指定在条件为假的情况下所执行的代码：

```
if expression
  code
else
  code
end
```

如果 *expression* 的值不是 false 或 nil，那么位于 if 和 else 之间的 *code* 将被执行；否则（如果 *expression* 的值是 false 或 nil），位于 else 和 end 之间的 *code* 将被执行。和简单形式的 if 一样，你需要在 *expression* 后面加上一个换行符，或者一个分号，或者关键字 then，和 *code* 相隔离。关键字 else 和 end 可以完全界定第二块 *code*，所以不再需要换行符或其他额外的分界符了。

下面是一个包含了 else 从句的条件式：

```
if data           # If the array exists
  data << x       #   then append a value to it.
else              # Otherwise...
```

注 1：Ruby 1.8 还允许使用冒号进行分隔，但是 Ruby 1.9 已经不允许了。

```
  data = [x]      #   create a new array that holds the value.
end               # This is the end of the conditional.
```


5.1.1.2 elsif

如果你想在一个条件式内测试多个条件分支，那么可以通过在 if 和 else 之间加入一个或多个 elsif 从句的方式来实现。elsif 是“else if”的缩写形式。请注意，elsif 里只有一个 e。一个使用了 elsif 的条件式看起来就像下面这样：

```
if expression1
  code1
elsif expression2
  code2
      .
      .
      .
elsif expressionN
  codeN
else
  code
end
```

如果 *expression1* 的值不等于 false 或 nil，那么 *code1* 就被执行，否则，将求 *expression2* 的值。如果它的值不等于 false 或 nil，那么 *code2* 被执行。这个过程会不断重复执行，直到遇见某个不等于 false 或 nil 的 expression，或者测试完了所有的 elsif 从句。如果最后一个 elsif 从句对应的 expression 的值也是 false 或 nil，而且在 elsif 从句后面有一个 else 从句，那么那些位于 else 和 end 之间的代码将被执行。如果没有 else 从句，那么就不执行任何代码。

elsif 类似于 if：在 expression 和 code 之间必须有一个换行符、一个分号或一个关键字 then 来作为分界符。下面是一个使用 elsif 的多路条件式的例子：

```
if x == 1
  name = "one"
elsif x == 2
  name = "two"
elsif x == 3 then name = "three"
elsif x == 4; name = "four"
else
  name = "many"
end
```

5.1.1.3 返回值

在大多数语言里，if 条件式是一个语句。但在 Ruby 中，一切都是表达式，包括那些通常被称为语句的控制结构。一个 if“语句”的返回值（也就是对一个 if 表达式进行求值而得到的值）就是被执行的代码中最后一个表达式的值；如果没有执行任何代码，那么返回值就是 nil。

作为 if 语句能返回一个值的例子，我们可以优雅地将前面提到的多路条件式重写成下面这样：

```
name = if    x == 1 then "one"
       elsif x == 2 then "two"
       elsif x == 3 then "three"
       elsif x == 4 then "four"
       else              "many"
       end
```

5.1.2 作为修饰符的 if
if As a Modifier

采用 if 的普通语句形式时，Ruby 的语法要求其结尾必须有一个 end 关键字。对于那些简单的、只有一行代码的条件式来说，这显得有些笨拙。这只是一个语法分析问题，解决之道就是将 if 语句本身作为分界符，将代码和条件式隔离开。这种情况下，我们通常不将代码写成：

```
if expression then code end
```

而是像下面这样写：

```
code if expression
```

此时，if 被称为语句（或表达式）**修饰符**。假如你是一位 Perl 程序员，也许已经习惯了这种语法。否则，请注意要先编写执行代码，然后再编写条件表达式。比如：

```
puts message if message    # Output message, if it is defined
```

这种语法更加强调那些将要执行的代码，而不是对应的条件式。当条件式本身不太重要，或者它几乎始终为真时，采用这样的语法能使得你的代码更具可读性。

虽然条件式是后编写的，但它是先被求值的。如果它的值不等于 false 或 nil，那么对应的代码将被执行，而且所得的结果将作为这个修饰表达式的返回值。否则，代码将不会执行，而且这个修饰表达式的返回值将为 nil。显而易见，这种语法不允许出现任何 else 从句。

当 if 作为修饰符时，它必须紧跟在被修饰的语句或表达式后面，其间不能有换行符。如果我们在先前的那个例子中插入一个换行符，就会把它变成一个未经修饰的方法调用和一个不完整的 if 语句：

```
puts message          # Unconditional
if message            # Incomplete!
```

if 修饰符的优先级非常低，它与操作数绑定的紧密程度还不如赋值操作符，所以当使用 if 修饰符时，请确认你确实修饰了想要修饰的表达式。比如，下面的两行代码是不一样的：

```
y = x.invert if x.respond_to? :invert
y = (x.invert if x.respond_to? :invert)
```

在第一行代码里，if 修饰符作用于整个赋值表达式。如果 x 没有名为 invert 的方法，那么不会有任何事情发生。在第二行代码里，if 修饰符仅仅作用于方法调用。如果 if 没有一个 invert 方法，那么该修饰表达式的值将为 nil，然后这个值被赋给 y。

if 修饰符只和最近的表达式绑定在一起。如果你想要修饰多个表达式，那么你可以使用圆括号或一个 begin 语句进行组合。这种方法的问题在于，读者阅读完整个 if 表达式之前，他们并不知道那些位于圆括号或 begin 语句中的代码是条件式的一部分。此外，以这样一种方式使用 if 修饰符也失去了这种语法的主要优势，即简洁性。所以当要包含多行代码时，你应该使用传统的 if 语句而不是 if 修饰符。请比较下面三个并列的备选方案：

```
if expression    begin              (
  line1            line1                line1
  line2            line2                line2
end              end if expression    )   end if expression
```

请注意，一个被 if 从句修饰的表达式本身还可以被再次修饰，因此可以在一个表达式上附加多个 if 修饰符：

```
# Output message if message exists and the output method is defined
puts message if message if defined? puts
```

但是，像这样重复一个 if 修饰符会使得代码难以阅读，而且将两个条件式合并成一个表达式会更有意义：

```
puts message if message and defined? puts
```

5.1.3 unless

unless，作为一个语句或一个修饰符，和 if 恰好相反：当且仅当与其关联的表达式的值为 false 或 nil 时，它才会执行相应的代码。除了不允许出现 elsif 从句，它的语法和 if 的类似：

```
# single-way unless statement
unless condition
  code
end

# two-way unless statement
unless condition
  code
else
  code
end

# unless modifier
code unless condition
```

类似于 if 语句，unless 语句也要求其条件式和代码之间有一个换行符、一个分号，或者一个 then 关键字进行分隔。还有一点类似于 if 的是，unless 语句也是表达式，而且返回所执行的代码的值，或者当没有执行任何代码时返回 nil。

```
# Call the to_s method on object o, unless o is nil
s = unless o.nil?                    # newline separator
  o.to_s
end
s = unless o.nil? then o.to_s end    # then separator
```

对于下面这样的单行条件式，unless 的修饰符形式通常会更清晰一些：

```
s = o.to_s unless o.nil?
```

Ruby 没有为 unless 条件式提供对应于 if 中的 elsif 从句的等价物，但是，如果愿意编写一些稍显啰嗦的代码，你还是可以实现多路 unless 语句的：

```
unless x == 0
  puts "x is not 0"
else
  unless y == 0
    puts "y is not 0"
  else
    unless z == 0
      puts "z is not 0"
    else
      puts "all are 0"
    end
  end
end
```

5.1.4 case

case 语句是一个多路条件式，它具有两种形式，其中简单的那种形式（很少使用）只是 if/elsif/else 的一个替代性选择。下面两个并列的表达式是等价的：

```
name =  case                          name =  if   x == 1 then "one"
        when x == 1 then "one"                elsif x == 2 then "two"
        when x == 2 then "two"                elsif x == 3 then "three"
        when x == 3 then "three"              elsif x == 4 then "four"
        when x == 4 then "four"               else "many"
        else "many"                           end
        end
```

正如你在上述代码中所见的，和 if 语句一样，case 语句也会返回一个值。而且，你还可以用一个换行符或分号替代那个跟在 when 从句后面的关键字 then（注 2）。

注 2：在 Ruby 1.8 中，和 if 语句一样，允许用冒号来替代关键字 then。但是在 Ruby 1.9 里已经不能这样做了。

```
case
when x == 1
  "one"
when x == 2
  "two"
when x == 3
  "three"
end
```

case 语句按照 when 表达式的编写顺序对它们逐一进行测试，直到找到一个值为 true 的为止。如果找到了这样一个 when 表达式，那么它执行位于这个 when 和后续的 when、else 或 end 之间的语句。最后执行的那个表达式的值将被作为 case 语句的返回值。一旦找到了一个值为 true 的 when 从句，它就不再考虑其他的 when 从句。

case 语句中的 else 从句是可选的，但是如果确实需要它，它必须出现在所有其他 when 从句之后，位于整个 case 语句的末尾。如果没有为 true 的 when 从句，而且存在一个 else 从句，那么位于 else 和 end 之间的代码将被执行。这块代码中最后执行的表达式的值将被作为整个 case 语句的值。如果没有为 true 的 when 从句，而且也没有 else 从句，那么就不会执行任何代码，而且 case 语句的返回值为 nil。

case 语句中的一个 when 从句可以包含多个表达式（用逗号分隔），只要这些表达式中有一个为 true，那么与该 when 从句相关联的代码就被执行。在简单形式的 case 语句里，逗号的作用不大，就好像一个||操作符一样。

```
case
when x == 1, y == 0 then  "x is one or y is zero"  # Obscure syntax
when x == 2 || y == 1 then "x is two or y is one"  # Easier to understand
end
```

目前为止，我们所展示的关于 case 的所有例子都是较简单且较少使用的形式，实际上 case 的功能比这强大得多。在大部分例子中，每个 when 从句的左侧都是一样的。在 case 的常用形式里，我们将这些 when 从句的重复的左侧表达式提取出来，与 case 本身关联在一起：

```
name =   case x
         when 1                    # Just the value to compare to x
            "one"
         when 2 then "two"         # Then keyword instead of newline
         when 3; "three"           # Semicolon instead of newline
         else "many"               # Optional else clause at end
         end
```

在这种形式的 case 语句里，与 case 相关联的那个表达式被求值一次，然后 case 语句将其与那些从 when 表达式中得到的值进行比较。整个比较过程将按照 when 从句的编写顺序进行，而且与第一个匹配成功的 when 从句相关联的代码将被执行。如果没有成功匹配，那么与 else 从句（如果确实存在一个）相关联的代码将被执行。这种形式的 case 语句的返回

值与简单形式的 case 语句的返回值是一样的：要么是最后一个被执行的表达式的值，要么是 nil（没有 when 或 else 匹配成功就会如此）。

理解 case 语句的重点在于，搞清楚 when 从句里的值是如何与 case 关键字后的表达式的值相比较的。这个比较使用的是===操作符。会在 when 从句里的值上调用===操作符，并且以 case 表达式的值为实参，因此，上面的那个 case 语句和下面的是等价的（除了上面的 case 语句中的 x 只求值一次）：

```
name =   case
         when 1 === x then "one"
         when 2 === x then "two"
         when 3 === x then "three"
         else "many"
         end
```

===是**条件相等性操作符**（*case equality*）。对于许多类来说，比如先前提到的 Fixnum 类，===操作符的行为和==是一样的。但某些特定的类也重定义了该方法，用于实现一些有趣的行为：Class 类将===定义为测试其右侧操作数是否为左侧操作数所命名的类的一个实例；Range 类将===定义为测试其右侧操作数是否位于左侧操作数的范围之内；Regexp 类将===定义为测试其右侧操作数是否匹配左侧操作数所指定的模式；在 Ruby 1.9 里，Symbol 类将===定义为测试符号或字符串的相等性。基于这些对条件相等性操作符的定义，我们可以编写一些像下面这样有趣的 case 语句：

```
# Take different actions depending on the class of x
puts case x
     when String then "string"
     when Numeric then "number"
     when TrueClass, FalseClass then "boolean"
     else "other"
     end

# Compute 2006 U.S. income tax using case and Range objects
tax = case income
      when 0..7550
        income * 0.1
      when 7550..30650
        755 + (income-7550)*0.15
      when 30650..74200
        4220 + (income-30655)*0.25
      when 74200..154800
        15107.5 + (income-74201)*0.28
      when 154800..336550
        37675.5 + (income-154800)*0.33
      else
        97653 + (income-336550)*0.35
      end

# Get user's input and process it, ignoring comments and exiting
```

```
# when the user enters the word "quit"
while line=gets.chomp do      # Loop, asking the user for input each time.
  case line
  when /^\s*#/                # If input looks like a comment...
    next                      # skip to the next line.
  when /^quit$/i              # If input is "quit" (case insensitive)...
    break                     # exit the loop.
  else                        # Otherwise...
    puts line.reverse         # reverse the user's input and print it.
  end
end
```

一个 when 从句可以关联多个表达式，它们之间由逗号分隔，而且依次调用它们的===操作符，也就是说可能会出现多个不同的值都可以引发同一段代码的执行的情况。

```
def hasValue?(x)              # Define a method named hasValue?
  case x                      # Multiway conditional based on value of x
  when nil, [], "", 0         # if nil===x || []===x || ""===x || 0===x then
    false                     # method return value is false
  else                        # Otherwise
    true                      # method return value is true
  end
end
```

case 与 switch 的比较

对于 Java 程序员和那些熟悉 C 风格语法语言的程序员来说，switch 这个多路条件语句应该毫不陌生，它类似于 Ruby 中的 case 语句，但是，两者还是有一些重要的差异：

- 在 Java 及相关语言中，这个多路条件语句的名字是 switch，从句则标记为 case 和 default，而 Ruby 则用 case 作为语句的名字，使用 when 和 else 来标记从句。
- 在其他语言中，switch 语句只是简单地将程序的控制流转移到合适的 case 从句。从那里开始，控制流可以“穿越”多个后续的 case 从句不断执行，直到到达 switch 语句的末尾，或者遇到一个 break 或 return 语句才会结束。这种行为使多个 case 从句可以引用同一块代码。在 Ruby 中，同样的目的，是通过允许每个 when 从句关联多个逗号分隔的表达式来实现的。Ruby 的 case 语句是不允许这种“穿越”行为的。
- 在 Java 和大多数具有 C 风格语法的编译型语言中，与每个 case 标签相关联的表达式必须是编译时常量，而不能是任意的运行时表达式，这常常使编译器能够用一个非常快速的查询表 (lookup table) 来实现 switch 语句。在 Ruby 中没有这样的限制，case 语句的性能相当于一个具有多个 elsif 从句的 if 语句的性能。

5.1.5 ?:操作符
The ?: Operator

在之前的第 4.6.10 节介绍过条件操作符?:，它的行为与 if 语句非常相似。它用?替换了 then，用:替换了 else，提供了一种简洁的表达条件式的方式。

```
def how_many_messages(n) # Handle singular/plural
  "You have " + n.to_s + (n==1 ? " message." : " messages.")
end
```

5.2 循环
Loops

本节将讲述 Ruby 的简单循环语句：while、until 和 for。Ruby 还具有定义那种被称为**迭代器**的自定义循环结构的能力。迭代器（请参见第 5.3 节）或许比 Ruby 的内建循环语句更为常用一些，本章稍后部分将介绍它们。

5.2.1 while 和 until
while and until

while 和 until 是 Ruby 的基本循环语句，**当**某个特定条件为真，或者**直到**某个条件为真时，它们就会执行一段代码。

```
x = 10                  # Initialize a loop counter variable
while x >= 0 do         # Loop while x is greater than or equal to 0
  puts x                #   Print out the value of x
  x = x - 1             #   Subtract 1 from x
end                     # The loop ends here

# Count back up to 10 using an until loop
x = 0                   # Start at 0 (instead of -1)
until x > 10 do         # Loop until x is greater than 10
  puts x
  x = x + 1
end                     # Loop ends here
```

循环条件是一个位于关键字 while 和 do，或者 until 和 do 之间的布尔表达式，循环体是位于关键字 do 和 end 之间的 Ruby 代码。while 循环首先测试其循环条件，如果得到的值不是 false 或 nil，它就会执行一遍循环体，然后再次测试其循环条件。按照这种方式，循环体可能被执行零次，也可能被反复执行多次，只要条件一直为真即可（或者更严格的说，是非 false 和非 nil）。

until 循环的行为正好相反。首先还是先测试循环条件，当结果为 false 或 nil 时，就会执行循环体。这意味着循环体可能被执行零次，也可能在条件为 false 或 nil 的前提下被执行多次。通过简单地对循环条件求反，我们可以将任何一个 until 循环转换成 while 循环。大多数程序员都熟悉 while 循环，却不太习惯 until 循环。因此，除非 until 真的能使代码更清晰，否则你可能还是更愿意使用 while 循环。

while 或 until 循环中的关键字 do 类似于 if 语句的关键字 then，只要在循环条件和循环体之间存在一个换行符（或分号），就可以将 do 省略掉（注 3）。

5.2.2 作为修饰符的 while 和 until

while and until As Modifiers

如果一个循环体仅包含一个 Ruby 表达式，那么你可以将 while 和 until 作为修饰符置于该表达式之后，以一种非常紧凑的形式来表达该循环。比如：

```
x = 0                          # Initialize loop variable
puts x = x + 1 while x < 10    # Output and increment in a single expression
```

在这种修饰符语法中，while 关键字亲自分隔了循环体和循环条件，从而省却了 do（或换行符）和 end 关键字。请将上述代码和下面的代码进行比较，它刻意将传统的 while 循环写在了一行代码中：

```
x = 0
while x < 10 do puts x = x + 1 end
```

与 while 一样，until 也可以作为修饰符：

```
a = [1,2,3]                    # Initialize an array
puts a.pop until a.empty?      # Pop elements from array until empty
```

请注意，当 while 和 until 作为修饰符时，它们必须和那些被它们所修饰的循环体处于同一行中。如果在循环体和 while 或 until 关键字之间存在一个换行符，那么 Ruby 解释器会将循环体当成一个未经修饰的表达式来处理，而且将 while 或 until 当作一个普通循环的开头部分。

当 while 和 until 作为单个 Ruby 表达式的修饰符时，虽然循环条件出现在循环体之后，但还是会先测试循环条件，而循环体会被执行零次或多次，就好像它位于一个普通的 while 或 until 循环中一样。

上述规则有一个例外情况：当循环体是一个位于关键字 begin 和 end 之间的复合表达式时，它会在测试循环条件之前先执行一遍。

```
x = 10                 # Initialize loop variable
begin                  # Start a compound expression: executed at least once
  puts x               # output x
  x = x - 1            # decrement x
end until x == 0       # End compound expression and modify it with a loop
```

上述结构非常类似于 C，C++及 Java 中的 do/while 循环。虽然它和其他语言的 do/while 循环比较相似，但是由于这种结合了 begin 语句的循环修饰符的特殊行为比较违反直觉，所以不鼓励使用它。Ruby 的未来版本可能会禁止这种将 while、until 修饰符与 begin/end 相结合的方式。

注 3：在 Ruby 1.8 里还可以使用一个冒号来代替 do 关键字，但是在 Ruby 1.9 中已经不允许这样做了。

请注意，如果你用圆括号将多个语句组合起来，并且后接一个 until 修饰符，那么就不会发生上述的特殊行为：

```
x = 0              # Initialize loop variable
(                  # Start a compound expression: may be executed 0 times
  puts x           # output x
  x = x - 1        # decrement x
) until x == 0     # End compound expression and modify it with a loop
```

5.2.3 for/in 循环

The for/in Loop

for 或 for/in 循环可对一个可枚举对象（比如一个数组）的元素进行迭代。在每次迭代过程中，它将一个元素赋给一个特定的循环变量并且执行循环体。一个 for 循环看起来像下面这样：

```
for var in collection do
  body
end
```

var 是一个变量或一个由逗号分隔的变量列表，*collection* 则是一个具有 each 迭代器方法的对象。数组、哈希及许多其他的 Ruby 对象都定义了 each 方法。for/in 循环调用指定对象的 each 迭代器方法返回相应的值，随后它将每个值（或一组值）赋给指定的变量（或一组变量），然后执行**循环体**内的代码。和 while 和 until 循环一样，for 循环里的关键字 do 是可选的，可以被替换成一个换行符或分号。

下面是一些 for 循环的例子：

```
# Print the elements in an array
array = [1,2,3,4,5]
for element in array
  puts element
end

# Print the keys and values in a hash
hash = {:a=>1, :b=>2, :c=>3}
for key,value in hash
  puts "#{key} => #{value}"
end
```

一个 for 循环的循环变量（或变量组）并不是该循环的局部变量，在对应的循环退出后，它们仍然具有定义。类似地，在循环体中定义的新变量也能在循环结束后继续存在。

for 循环对 each 迭代器方法的依赖意味着它非常像迭代器。比如，上面有一个枚举哈希的键和值的 for/in 循环，我们还可以通过显式地采用 each 迭代器的方式来完成同样的功能。

```
hash = {:a=>1, :b=>2, :c=>3}
hash.each do |key,value|
  puts "#{key} => #{value}"
end
```

循环的 for 版本和 each 版本之间的惟一差别在于，跟在一个迭代器之后的代码块会定义一个新的变量作用域。我们将在本章后面讨论迭代器的时候提供相关细节。

5.3 迭代器和可枚举对象
Iterators and Enumerable Objects

虽然 while、until 和 for 循环是 Ruby 语言的核心部分之一，但是通常情况下我们更倾向于使用一个特殊的方法来编写循环，那就是**迭代器**。迭代器是 Ruby 最为重要的特性之一，下面是一些经常会出现在入门级的 Ruby 手册中的例子：

```
3.times { puts "thank you!" }      # Express gratitude three times
data.each {|x| puts x }            # Print each element x of data
[1,2,3].map {|x| x*x }             # Compute squares of array elements
factorial = 1                      # Compute the factorial of n
2.upto(n) {|x| factorial *= x }
```

times、each、map 及 upto 方法都是迭代器，它们和紧随其后的代码进行交互。yield 语句就是这些迭代器背后复杂的控制结构。yield 语句从迭代器中临时地将程序控制权返回给那个调用迭代器的方法，具体来说，程序的控制流会从迭代器转移到那个与迭代器调用相关联的代码块中，当程序执行完代码块之后，迭代器方法重新获得控制权并且从位于 yield 语句之后的第一个语句开始继续执行。

正如你在前面的例子里所见到的，代码块可以被参数化。一个代码块起始处的那对垂直条就相当于一个方法定义中的圆括号——它们包含了一个形参名列表。yield 语句就像一个方法调用，它的后面可以接零个或多个表达式，而这些表达式的值将被赋给对应的代码块形参。

迭代器并非一定要迭代

在这本书里，我们用迭代器这个术语指代任何使用了 yield 语句的方法。也就是说，只要它们使用了 yield 就可以被称为迭代器，而不一定要实现迭代或循环的功能。（注 4）Object 类所定义的 tap 方法（在 Ruby 1.9 中）就是一个这样的例子。它调用相关联的代码块一次，并且将其调用者作为惟一的实参传递给该代码块，然后返回其调用者。下面的代码展示了一个使用 tap 输出调试信息的例子，可以看到，当我们要“嵌入”一个方法调用链时，它显得很方便：

注 4：在日本的 Ruby 社区里，“迭代器”这个术语已经不被使用了，因为它包含了一种迭代的意思，但是这种迭代其实并不是必须的。一个像“期待一个关联代码块的方法”这样的短语虽然显得很啰嗦，但是却能提供更准确的表述。

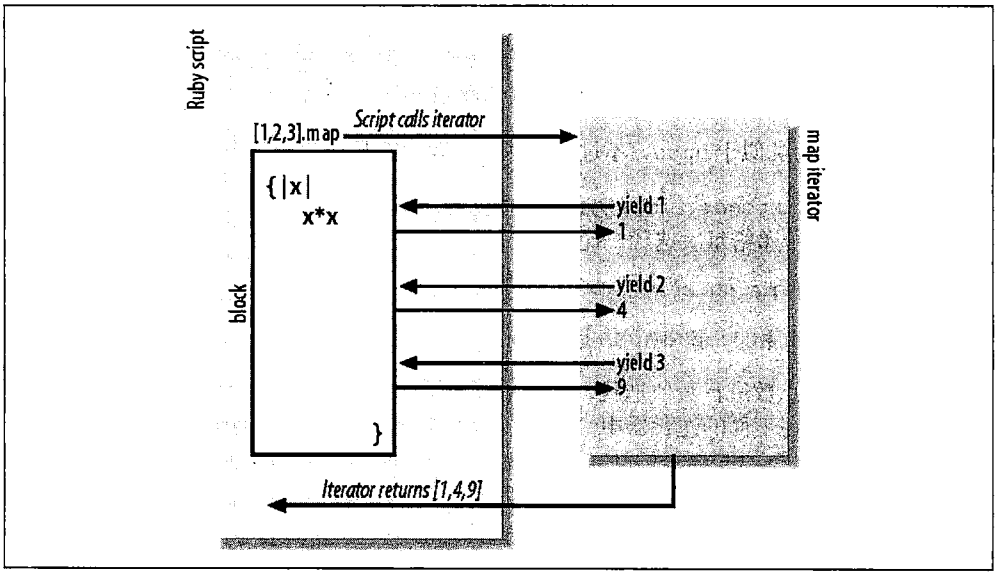


图 5-1：一个迭代器和其调用方法之间的控制流转换

```
chars = "hello world"
  .each_char
  .to_a
  .map {|c| c.succ }
  .sort
```

```
.tap {|x| puts "original object: #{x.inspect}"}
.tap {|x| puts "each_char returns: #{x.inspect}"}
.tap {|x| puts "to_a returns: #{x.inspect}"}
.tap {|x| puts "map returns: #{x.inspect}" }
.tap {|x| puts "sort returns: #{x.inspect}"}
```

迭代器的另一个常见功能就是自动的资源回收。比如，File.open 方法可被用作一个迭代器，它打开指定的文件并创建一个 File 对象来表示它。如果没有代码块与其调用相关联，那么它就简单地返回一个 File 对象，并且将关闭文件的责任留给调用代码。但是，假如有一个代码块与其调用相关联，它就将新创建的那个 File 对象传递给该代码块，并且在代码块返回之后自动关闭相应的文件。这就确保了文件始终被关闭，而且可以将程序员从这些清理工作的细节中解放出来。

5.3.1 数值迭代器
Numeric Iterators

Ruby 的核心 API 提供了许多标准的迭代器。Kernel 类的 loop 方法表现得像一个无限循环，它不断地执行与其关联的代码块，直到该代码块执行一个 return、break 或其他能够退出循环的语句为止。

Integer 类定义了三个常用的迭代器。upto 方法为一个区间内的每个整数调用其关联的代码块，该区间的左端点就是调用 upto 方法的那个整数，其右端点则是被传递给 upto 的那个实参。比如：

```
4.upto(6) {|x| print x}   # => prints "456"
```

如你所见，upto 把每一个整数都传递给与其相关联的代码块，而且包含该区间的起始和结束端点。总的来说，n.upto(m)会调用其相关联的代码块 m-n+1 次。

downto 方法的行为类似于 upto，不过它是从一个较大的数迭代到一个较小的数。

当在整数 n 上调用 Integer.times 方法时，该方法将调用相关联的代码块 n 次，而且把从 0 到 n-1 的值依次传递给每次迭代。比如：

```
3.times {|x| print x }    # => prints "012"
```

总的来讲，n.times 和 0.upto(n-1)是等价的。

如果你希望使用浮点数进行数值迭代，那么可以使用 Numeric 类定义的更为复杂的 step 方法。作为例子，在下面的迭代器中，Step 方法将从 0 开始以 0.1 为步长进行迭代，直到 Math::PI 为止。

```
0.step(Math::PI, 0.1) {|x| puts Math.sin(x) }
```

5.3.2 可枚举的对象
Enumerable Objects

Array、Hash、Range 及许多其他的类都定义了一个 each 迭代器，它将集合的每个元素传递给相关联的代码块。这或许是 Ruby 中最为常用的迭代器。正如我们先前所见到的，for 循环只能作用于那些具有 each 方法的对象。下面是一些 each 迭代器的例子：

```
[1,2,3].each {|x| print x }   # => prints "123"
(1..3).each  {|x| print x }   # => prints "123" Same as 1.upto(3)
```

each 迭代器并不是那些传统的“数据结构”类的专利（译注 1），Ruby 的 IO 类也定义了一个 each 迭代器，它可以把那些从输入/输出对象中读取出来的文本行传递出来。因此在 Ruby 中你可以像下面这样处理一个文件的行：

```
File.open(filename) do |f|          # Open named file, pass as f
  f.each {|line| print line }       # Print each line in f
end                                 # End block and close file
```

大多数定义了 each 方法的类都包含 Enumerable 模块，它定义了许多更特殊的迭代器，而它们都是基于 each 方法来实现的。each_with_index 就是这些有用的迭代器的其中之一。如果把它运用到先前的例子上，我们就可以给每行增加一个行号：

```
File.open(filename) do |f|
  f.each_with_index do |line,number|
    print "#{number}: #{line}"
  end
end
```

一些最为常用的 Enumerable 迭代器包括 collect、select、reject 及 inject。collect

译注 1：数组、哈希等都是经典的数据结构。

方法（也被称为 map）为那个调用它的可枚举对象的每个元素执行相关联的代码块，并且将所有返回值组合到一个数组中，然后再返回：

```
squares = [1,2,3].collect {|x| x*x}   # => [1,4,9]
```

select 方法为那个调用它的可枚举对象的每个元素执行相关联的代码块，如果代码块的返回值不是 false 或 nil，那么该元素就会被选中，最后 select 将所有被选中的元素组合成一个数组并返回。

```
evens = (1..10).select {|x| x%2 == 0} # => [2,4,6,8,10]
```

reject 方法和 select 方法恰好相反，它返回那些使代码块返回 nil 或 false 的元素。比如：

```
odds = (1..10).reject {|x| x%2 == 0}  # => [1,3,5,7,9]
```

inject 方法比其他的方法稍微复杂一些。它用两个参数调用相关联的代码块，第一个参数是一个来自此前的迭代的累计值，第二个参数则是那个调用它的可枚举对象的下一个元素。如果当前迭代不是最后一次迭代，那么其代码块的返回值就成为下一次迭代的第一个参数（即累计值），否则，就成为 inject 迭代器的返回值。如果在调用 inject 的时候传给它一个参数，那么该参数成为累计值变量的初始值，否则，就使用那个可枚举对象的第一个元素作为累计值变量的初始值。（在这种情况下，对于该对象的前两个元素，只会调用一次代码块。）下面的例子将进一步阐明 inject 的用法：

```
data = [2, 5, 3, 4]
sum = data.inject {|sum, x| sum + x }     # => 14    (2+5+3+4)
floatprod = data.inject(1.0) {|p,x| p*x } # => 120.0 (1.0*2*5*3*4)
max = data.inject {|m,x| m>x ? m : x }    # => 5     (largest element)
```

请参见第 9.5.1 节获得 Enumerable 模块及其迭代器的进一步信息。

5.3.3 编写自定义的迭代器

Writing Custom Iterators

一个迭代器方法的定义性特征（defining feature）在于它可调用一个与方法调用相关联的代码块。你是通过 yield 语句来做到这一点的。下面是一个很简单的迭代器，它只是调用了两次相关联的代码块而已：

```
def twice
  yield
  yield
end
```

如果想给代码块传递实参，那么你就要在 yield 语句后面放上一个由逗号分隔的表达式列表。和方法调用一样，可以将实参值放入圆括号内，但这是可选的。下面用一个简单迭代器展示了 yield 的用法：

```
# This method expects a block. It generates n values of the form
# m*i + c, for i from 0..n-1, and yields them, one at a time,
# to the associated block.
def sequence(n, m, c)
```

```
  i = 0
  while(i < n)          # Loop n times
    yield m*i + c       # Invoke the block, and pass a value to it
    i += 1              # Increment i each time
  end
end

# Here is an invocation of that method, with a block.
# It prints the values 1, 6, and 11
sequence(3, 5, 1) {|y| puts y }
```

术语：yield 和迭代器（iterator）

基于你的编程背景，你或许觉得术语“yield”和“迭代器”有些令人困惑。前面介绍的 sequence 方法非常清晰地展示了为什么会有 yield 这样一个名字。每次在计算完序列中的一个数之后，该方法将控制权让给代码块（同时传递出那个计算得来的值），这样代码块才能和它相互配合。但是情况并不总是这样清晰，在有些代码里，看起来就好像是代码块向它的调用方法传回了一个结果。

一个像 sequence 这样的方法会希望有一个相关联的代码块，而且多次调用它。因为一个这样的方法无论是外形还是行为都像一个循环，所以它被称为**迭代器**。如果你习惯了那些类似于 Java 的语言，那么这或许令你感到困惑，因为在那样的语言中，迭代器是对象。在 Java 中，那些使用迭代器的代码具有控制权，而且它们在需要的时候从迭代器中“取出”值。在 Ruby 中，迭代器方法具有控制权，而且它们将值“推送”给那些需要它们的代码块。

这个术语问题与“内部迭代器”和“外部迭代器”之间的差异有关，这将在本节后续进行讨论。

下面是 Ruby 迭代器的另一个例子，它传递两个实参给相关联的代码块。值得注意的是，这个迭代器实现在内部使用了另外一个迭代器：

```
# Generate n points evenly spaced around the circumference of a
# circle of radius r centered at (0,0). Yield the x and y coordinates
# of each point to the associated block.
def circle(r,n)
  n.times do |i|    # Notice that this method is implemented with a block
    angle = Math::PI * 2 * i / n
    yield r*Math.cos(angle), r*Math.sin(angle)
  end
end

# This invocation of the iterator prints:
# (1.00, 0.00) (0.00, 1.00) (-1.00, 0.00) (-0.00, -1.00)
circle(1,4) {|x,y| printf "(%.2f, %.2f) ", x, y }
```

使用 yield 关键字的确很像调用一个方法（有关方法调用的全部细节请参见第 6 章）。围绕参数的圆括号是可选的。你可以使用*将一个数组展开成一些单个的参数，yield 甚至还允许你将一个没有用花括号括起来的哈希字面量作为其参数。但是，它和方法调用的不同之处在于不能在 yield 表达式后面接代码块，你不能将一个代码块传递给另一个代码块。

如果一个方法在调用时没有相关联的代码块，那么在定义该方法时使用 yield 就是错误的，因为在这种情况下，将没有什么可以作为 yield 的目标。有时候你希望编写这样一个方法：当有相关联的代码块时就 yield，否则将采取一些默认行为（而不是抛出一个错误）。为了做到这一点，你可以使用 block_given?方法判断是否在调用该方法时带有一个代码块。block_given?及它的同义词 iterator?都是 Kernel 的方法，因此它们表现得像全局函数一样。下面是一些例子：

```
#
# Return an array with n elements of the form m*i+c
# If a block is given, also yield each element to the block
def sequence(n, m, c)
  i, s = 0, []                  # Initialize variables
  while(i < n)                  # Loop n times
    y = m*i + c                 # Compute value
    yield y if block_given?     # Yield, if block
    s << y                      # Store the value
    i += 1
  end
  s                             # Return the array of values
end
```

5.3.4 枚举器
Enumerators

一个**枚举器**是一个 Enumerable 对象，其目的在于枚举其他的对象。在 Ruby 1.8 中，为了使用枚举器，你必须 require 'enumerator'，而 Ruby 1.9 已经内建了枚举器，所以你无需再 require 了。（后面我们会见到，Ruby 1.9 的内建枚举器的功能比 Ruby 1.8 的枚举器类库的功能丰富得多。）

枚举器是类 Enumerable::Enumerator 的实例。虽然可以通过 new 直接实例化这个类，但是通常情况下我们并不会通过这种方式来创建枚举器，而是使用 Object 类的 to_enum 或其同义词 enum_for 方法。如果在调用的时候没有提供参数，那么 to_enum 会返回一个枚举器，这个枚举器的 each 方法只是简单地调用目标对象的 each 方法。假设你有一个数组和一个方法，该方法期望一个可枚举对象。因为数组是可变的，而且你并不确定该方法不会修改该数组，所以你不想直接将数组传递给该方法。为了达到这个目的，与其创建一个该数组的防御性深拷贝，不如调用它的 to_enum 方法，并且将生成的枚举器传递给那个方法。事实上，你为你的数组创建了一个可枚举但不可变的代理对象。

```
# Call this method with an Enumerator instead of a mutable array.
# This is a useful defensive strategy to avoid bugs.
process(data.to_enum)  # Instead of just process(data)
```

你也可以给 to_enum 方法传递实参，尽管在这种情况下，enum_for 这个方法名显得更自然一些。enum_for 方法的第一个实参应该是一个符号，它标识了一个迭代器方法（来自原先

的对象)。enum_for 方法返回一个枚举器，它的 each 方法调用那个迭代器方法，同时把 enum_for 方法的其余实参传递给那个迭代器方法。在 Ruby 1.9 中，String 类不是 Enumerable 的，但是它具有三个迭代器方法：each_char、each_byte 和 each_line。假如我们想使用一个 Enumerable 方法，比如 map，而且我们希望基于 each_char 迭代器，那么我们可以像下面这样来创建一个枚举器：

```
s = "hello"
s.enum_for(:each_char).map {|c| c.succ }  # => ["i", "f", "m", "m", "p"]
```

在 Ruby 1.9 中，通常情况下都不需要像先前的例子那样显式地使用 to_enum 或 enum_for 方法，因为当以不带代码块的方式调用 Ruby 1.9 所内建的迭代器方法时（包括数值迭代器 times、upto、downto、step、each 及 Enumerable 相关的方法），它们会自动地返回一个枚举器，因此，为了给一个方法传递一个数组枚举器而非数组本身，你只须简单地调用 each 方法即可：

```
process(data.each)  # Instead of just process(data) (译注 2)
```

如果我们使用 chars 别名（注 5）代替 each_char 就会显得更自然一些。比如，为了将一个字符串的字符映射到一个字符数组上，只须使用 .chars.map 即可：

```
"hello".chars.map {|c| c.succ }  # => ["i", "f", "m", "m", "p"]
```

下面是一些其他的例子，它们都依赖于迭代器方法所返回的枚举器对象。请注意，并不是只有 Enumerable 模块定义的迭代器方法才能返回枚举器对象，像 times 和 upto 这样的数值迭代器也可以：

```
enumerator = 3.times                    # An enumerator object
enumerator.each {|x| print x }          # Prints "012"

# downto returns an enumerator with a select method
10.downto(1).select {|x| x%2==0} # => [10,8,6,4,2]

# each_byte iterator returns an enumerator with a to_a method
"hello".each_byte.to_a              # => [104, 101, 108, 108, 111]
```

当以不带代码块的方式调用你自己的迭代器方法时，你可以通过返回 self.to_enum 的方式来实现上述行为。下面的例子修改了先前提到的 twice 迭代器，使你在不带代码块调用它时，能够返回一个枚举器：

```
def twice
  if block_given?
    yield
    yield
  else
    self.to_enum(:twice)
  end
end
```

在 Ruby 1.9 中，迭代器对象还定义了一个 with_index 方法，该方法在 Ruby 1.8 的迭代器

译注 2：原文此处的代码是 process.(data.each_char) 和内容不符，疑为笔误。
注 5：Ruby 1.9.0 版遗漏了 chars，但是在初始版本发布后不久就纠正了这个疏忽。

模块里没有被提供。with_index 只是返回一个新的枚举器，它为迭代添加一个索引形参。比如，下面的例子将把一个字符串的字符抽取出来，并且包含这些字符在字符串里的索引：

```
enumerator = s.each_char.with_index
```

最后，请记住无论是 Ruby 1.8 还是 1.9，迭代器都是一些可被用于 for 循环的 Enumerable 对象。例如：

```
for line, number in text.each_line.with_index
  print "#{number+1}: #{line}"
end
```

5.3.5 外部迭代器
External Iterators

到目前为止，我们关于枚举器的讨论都集中在它们作为 Enumerable 代理对象的用法上。但是在 Ruby 1.9 中，枚举器还有另外一种非常重要的用途：它们是**外部迭代器**。你可以通过反复调用一个枚举器的 next 方法来遍历一个集合的元素。当所有元素都被遍历之后，next 方法抛出一个 StopIteration 异常：

```
iterator = 9.downto(1)              # An enumerator as external iterator
begin                               # So we can use rescue below
  print iterator.next while true    # Call the next method repeatedly
rescue StopIteration                # When there are no more values
  puts "...blastoff!"               # An expected, nonexceptional condition
end
```

内部迭代器和外部迭代器的比较

"gang of four" 在他们那关于设计模式的经典书籍里非常清楚地定义和比较了内部和外部迭代器（注 6）：

> 一个基本的问题是决定到底由哪一方来控制迭代，是迭代器还是使用迭代器的客户代码？当客户代码控制迭代时，该迭代器被称为**外部迭代器**，反之当迭代器控制迭代时，该迭代器就是一个**内部迭代器**。那些使用外部迭代器的客户代码必须负责推进整个遍历过程并且显式地从迭代器中获得下一个元素。相比之下，在使用内部迭代器时，客户代码将一个操作传递给一个内部迭代器，该迭代器依次在每个元素上应用该操作……
>
> 外部迭代器比内部迭代器更灵活。比如，使用外部迭代器可以很容易地比较两个集合的相等性，如果用内部迭代器则不太可能。但是从另一个角度来看，内部迭代器更容易使用，因为它们已经为你定义好了迭代逻辑。

注 6：*Gamma, Helm, Johnson, Vlissides 著. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.*

在 Ruby 中，像 each 这样的迭代器方法是内部迭代器，它们控制着迭代并且将值“推送”给那个与方法调用相关联的代码块。枚举器具有一个 each 方法，可用于内部迭代，但是在 Ruby 1.9 及其后的版本里，它们还能充当外部迭代器——客户代码可以使用 next 方法顺序地从一个枚举器中“取出”值。

外部迭代器的使用非常简单，每次需要另一个元素的时候调用 next 方法即可，遍历完所有元素之后，next 抛出一个 StopIteration 异常。这或许会显得有些反常——产生异常是为了一个期望的结束条件，而不是为了一个异常事件。（StopIteration 是 StandardError 和 IndexError 的后代类；请注意，它是那些在其名字内不包含“error”字样的异常类之一。）在外部迭代器技术上，Ruby 参照了 Python 的处理方式。没有必要去检查 next 的返回值是否是一个特殊的迭代终结值，也没有必要在调用 next 之前先调用一些类似于 next?的断言方法。

为了简化使用外部迭代器的循环操作，Kernel.loop 方法（Ruby 1.9）包含了一个隐式的 rescue 从句，而且在 StopIteration 抛出时干净利落地退出循环，因此，你可以更简单地编写先前所展示的倒计时代码：

```
iterator = 9.downto(1)
loop do                    # Loop until StopIteration is raised
  print iterator.next      # Print next item
end
puts "...blastoff!"
```

使用 rewind 方法可以使许多外部迭代器重新开始迭代，但是请注意，rewind 并不是对所有的迭代器都有效。如果一个枚举器基于一个像 File 这样从文件中顺序读入行的对象，那么调用 rewind 方法并不能使迭代从头开始。总的来说，如果重新调用底层 Enumerable 对象的 each 方法并不能使其重新开始一个迭代，那么调用 rewind 方法也不会奏效。

一个外部迭代器一旦启动（也就是在第一次调用 next 方法之后），就不能再克隆或复制该枚举器。可以克隆一个枚举器的典型时机包括：在 next 被调用之前、在 StopIteration 被抛出之后，或者在 rewind 被调用之后。

通常来说，具有 next 方法的枚举器是从具有 each 方法的 Enumerable 对象创建而来的。如果因为某些原因，你所定义的类为外部迭代提供了一个 next 方法，而不是为内部迭代提供一个 each 方法，那么你可以很容易地根据 next 来实现 each。事实上，将一个实现了 next 方法的外部可迭代类转换成一个 Enumerable 类非常简单，只需要像下面这样混入一个模块即可（用 include，请参见第 7.5 节）：

```
module Iterable
  include Enumerable          # Define iterators on top of each
  def each                    # And define each on top of next
    loop { yield self.next }
  end
end
```

另一种使用外部迭代器的方式是将其传递给一个内部迭代器方法，就像下面这样：

```
def iterate(iterator)
  loop { yield iterator.next }
end

iterate(9.downto(1)) {|x| print x }
```

先前从《设计模式》那本书里所引用的那段话暗示了外部迭代器的关键特性之一：它们解决了并行迭代的问题。假定你有两个 Enumerable 集合，而且需要成对地迭代它们的元素：首先是两个集合的第一个元素，然后是第二个元素，依次类推。在没有外部迭代器的情况下，你必须将其中的一个集合转换成一个数组（通过 Enumerable 模块的 to_a 方法），这样才能在用 each 迭代一个集合的时候访问另一个元素。

示例 5-1 展示了三个迭代器方法的实现。这三个迭代器都可以接受任意数目的 Enumerable 对象，并且以不同的方式对它们进行迭代：一个是简单的顺序迭代，只采用了内部迭代器；另外两个则是并行迭代，而且只有使用枚举器的外部迭代特性才能达到目的。

示例 5-1：采用外部迭代器的并行迭代

```
# Call the each method of each collection in turn.
# This is not a parallel iteration and does not require enumerators.
def sequence(*enumerables, &block)
  enumerables.each do |enumerable|
    enumerable.each(&block)
  end
end

# Iterate the specified collections, interleaving their elements.
# This can't be done efficiently without external iterators.
# Note the use of the uncommon else clause in begin/rescue.
def interleave(*enumerables)
  # Convert enumerable collections to an array of enumerators.
  enumerators = enumerables.map {|e| e.to_enum }
  # Loop until we don't have any more enumerators.
  until enumerators.empty?
    begin
      e = enumerators.shift     # Take the first enumerator
      yield e.next              # Get its next and pass to the block
    rescue StopIteration        # If no more elements, do nothing
    else                        # If no exception occurred
      enumerators << e          # Put the enumerator back
    end
  end
end

# Iterate the specified collections, yielding tuples of values,
# one value from each of the collections. See also Enumerable.zip.
def bundle(*enumerables)
  enumerators = enumerables.map {|e| e.to_enum }
```

```
  loop { yield enumerators.map {|e| e.next} }
end

# Examples of how these iterator methods work
a,b,c = [1,2,3], 4..6, 'a'..'e'
sequence(a,b,c) {|x| print x}     # prints "123456abcde"
interleave(a,b,c) {|x| print x}   # prints "14a25b36cde"
bundle(a,b,c) {|x| print x}       # '[1, 4, "a"][2, 5, "b"][3, 6, "c"]'
```

示例 5-1 中的 bundle 方法类似于 Enumerable.zip 方法。在 Ruby 1.8 里，zip 必须先将它的那些 Enumerable 实参转换成数组，以便在对它的调用对象（一个 Enumerable 对象）进行迭代的时候能够使用。然而在 Ruby 1.9 里，zip 方法可以使用外部迭代器。典型情况下，这使 zip 方法更具空间和时间效率，而且还使它可以作用于一些无边界集合，这些集合无法转换成一个具有固定大小的数组。

5.3.6 迭代和并发修改
Iteration and Concurrent Modification

总的来说，Ruby 的核心集合类都是直接对当前的对象集合进行迭代，而不是先为自己建立这些对象的私有拷贝或“快照”，然后再在这些拷贝或“快照”上进行迭代，而且在被迭代的时候，这些集合类不会试图去检测或防止对集合的并发修改。举个例子，如果你调用了一个数组的 each 方法，而在后续的代码块里又调用了同一个数组的 shift 方法，那么迭代的结果可能就会出乎你的意料：

```
a = [1,2,3,4,5]
a.each {|x| puts "#{x},#{a.shift}" }  # prints "1,1\n3,2\n5,3"
```

如果一个线程在另一个线程正在迭代某个集合的时候修改了该集合，那么你可能就会看到类似的令人惊讶的行为。避免出现这种情况的一种方法就是在迭代之前先创建一份防御性的集合拷贝。作为例子，下面的代码为 Enumerable 模块添加了一个 each_in_snapshot 方法来实现这种行为：

```
module Enumerable
  def each_in_snapshot &block
    snapshot = self.dup   # Make a private copy of the Enumerable object
    snapshot.each &block  # And iterate on the copy
  end
end
```

5.4 代码块
Blocks

使用代码块是使用迭代器的基础。在此前的小节里，我们的注意力主要集中在将迭代器当成一种循环结构之上。虽然代码块也隐含在我们的讨论当中，但它并不是主题。现在让我们将目光转向代码块本身。接下来的各个小节将解释如下内容：

- 将一个代码块和一个方法调用相关联的语法；
- 一个代码块的“返回值”；

- 代码块中的变量的作用域；
- 代码块形参和方法形参之间的差别。

5.4.1 代码块的语法
Block Syntax

代码块不会单独存在；它们只有出现在一个方法调用之后才是合法的。但是，你可以将一个代码块放在任何方法后面；如果该方法不是迭代器，也没有用 yield 来调用其后的代码块，那么该代码块就会被安静地忽略掉。你可以使用一对花括号或一对 do/end 关键字来作为代码块的分界符，起始的花括号或 do 关键字必须和方法调用处于同一行，否则 Ruby 解释器会将该行的行终结符当成该语句的终结符，并且以不带代码块的方式来调用该方法：

```
# Print the numbers 1 to 10
1.upto(10) {|x| puts x } # Invocation and block on one line with braces
1.upto(10) do |x|        # Block delimited with do/end
  puts x
end
1.upto(10)               # No block specified
 {|x| puts x }           # Syntax error: block not after an invocation
```

一个常见的惯例是：当代码块只有一行时采用花括号，有多行时就采用 do 和 end。但这也不仅仅是个惯例问题，Ruby 的分析器把{和它前面的标记紧紧地绑定在一起。如果你省略了方法调用实参周围的圆括号，而且采用花括号作为代码块的分界符，那么该代码块将和最后一个方法调用实参绑定在一起，而不是方法调用本身，这也许并不是你所期望的。为了避免发生这种情况，你应该要么在方法调用实参周围加上圆括号，要么就采用 do 和 end 作为分界符。

```
1.upto(3) {|x| puts x }       # Parens and curly braces work
1.upto 3 do |x| puts x end    # No parens, block delimited with do/end
1.upto 3 {|x| puts x }        # Syntax Error: trying to pass a block to 3!
```

和方法一样，你可以将代码块参数化。所有形参都位于一对竖线（|）之间，形参之间则由逗号分隔，除此之外，它们和方法的形参非常相似：

```
# The Hash.each iterator passes two arguments to its block
hash.each do |key, value|     # For each (key,value) pair in the hash
  puts "#{key}: #{value}"     # Print the key and the value
end                           # End of the block
```

一个常见的惯例是将方法调用、代码块形参及{或 do 关键字写在同一行里，但是语法上并不要求如此。

5.4.2 代码块的值
The Value of a Block

目前为止，本章所介绍的迭代器例子都把值传递给与其相关联的代码块，但是忽略掉

代码块的返回值。然而并非始终如此。下面将以 Array.sort 方法为例。如果你将一个代码块与这个方法调用相关联，它将成对的元素传递给代码块，然后由代码块来完成排序。代码块的返回值（-1，0 或 1）表明了两个参数的相对顺序。该代码块的“返回值”对于迭代器来说是可见的，而且作为其 yield 语句的值。

一个代码块的“返回值”就是它里面最后执行的那个表达式的值，因此，为了按从长到短的顺序对一个单词数组进行排序，我们可以像下面这样编写代码：

```
# The block takes two words and "returns" their relative order
words.sort! {|x,y| y <=> x }
```

我们将短语“返回值”放到引号当中，是有一个非常重要的理由的：一般来说，你不应该使用 return 关键字来从一个代码块中返回。一个位于代码块中的 return 语句将导致包含该代码块的那个方法返回（不是与代码块相关联的那个方法，而是包含代码块的那个方法）。当然，不可否认有些时候这正是你所期望的行为。但是，如果你希望从代码块返回到那个与它相关联的方法中，请不要使用 return。如果你需要代码块在执行完最后一个表达式之前就返回到调用方法中，或者你希望代码块能返回多个值，那么你应该使用 next，而不是 return。（第 5.5 节将解释 return、next 及相关的 break 语句的细节。）下面是一个使用 next 从代码块中返回的例子：

```
array.collect do |x|
  next 0 if x == nil      # Return prematurely if x is nil
  next x, x*x             # Return two values
end
```

请注意，以这种方式来使用 next 并不多见，你可以很容易地将上述代码重写成不带 next 的形式：

```
array.collect do |x|
  if x == nil
    0
  else
    [x, x*x]
  end
end
```

5.4.3 代码块和变量作用域
Blocks and Variable Scope

代码块定义了一个新的变量作用域：在一个代码块内定义的变量仅存在于该代码块内，在其之外就没有定义了。但是请小心，在一个方法内定义的局部变量在该方法的所有代码块中都可见，所以，如果一个代码块对一个已经在它外部定义过的变量进行赋值，那么就不会创建一个新的块级局部变量，而是将新值赋给那个已经存在的局部变量。有时候，这正是我们希望的行为：

```
total = 0
data.each {|x| total += x }       # Sum the elements of the data array
puts total                        # Print out that sum
```

不过有些时候，虽然我们并不希望在代码块内修改外围作用域的变量，却在不经意间这么做了。在 Ruby 1.8 中，这是一个代码块形参所特有的问题。在 Ruby 1.8 中，如果一个代码块形参和一个已经存在的变量同名，那么在调用该代码块时就会对那个已经存在的变量赋值，而不是创建一个新的块级局部变量。举个例子，下面的代码就有问题，因为它将同样的标识符 i 作为两个嵌套代码块的形参：

```
1.upto(10) do |i|          # 10 rows
  1.upto(10) do |i|        # Each has 10 columns
    print "#{i} "          # Print column number
  end
  print " ==> Row #{i}\n"  # Try to print row number, but get column number
end
```

Ruby 1.9 的处理方式有所不同：代码块形参的作用域范围始终在代码块内，而且对代码块的调用也不会对已经存在的值进行赋值。如果使用了 -w 选项来调用 Ruby 1.9，那么当一个代码块形参和一个已经存在的变量同名的时候，它就会发出警告。这可以帮助你编写出在 Ruby 1.8 和 1.9 中行为一致的代码。

在另一个重要方面，Ruby 1.9 也有所不同。代码块语法经过了扩展，使你可以声明块级局部变量，而且即使在外围作用域里已经有同名变量存在了，也能保证那些局部变量的块级局部性。为了做到这一点，你需要在代码块形参列表的后面接一个分号，然后再接一个以逗号分隔的块级局部变量列表。下面是一个例子：

```
x = y = 0                  # local variables
1.upto(4) do |x;y|         # x and y are local to block
                           # x and y "shadow" the outer variables
  y = x + 1                # Use y as a scratch variable
  puts y*y                 # Prints 4, 9, 16, 25
end
[x,y]                      # => [0,0]: block does not alter these
```

在这段代码里，x 是一个代码块形参：当以 yield 来调用代码块时，它就获得了一个值。y 是一个块级局部变量，它没有从 yield 调用那里接受任何值。在代码块真正对其赋值之前，它的值一直都是 nil。定义这些块级局部变量的目的在于保证你不会在不经意间修改那些已经存在的变量的值。（举个例子，当把一个代码块从一个方法拷贝并粘贴到另一个方法时，就可能发生这种情况。）如果你在调用 Ruby 1.9 时使用了 -w 选项，那么它就会发出警告，告诉你有一个块级局部变量隐藏了一个已经存在的外围变量。

当然，代码块可以具有多个参数及多个块级局部变量。下面展示了一个具有两个参数和三个块级局部变量的代码块：

```
hash.each {|key,value; i,j,k| ... }
```

5.4.4 向一个代码块传递实参
Passing Arguments to a Block

我们此前说过，代码块形参非常类似于方法形参，然而它们并不是严格地相等。与方法调

用的规则相比，将 yield 关键字后面的实参值传递给代码块形参的规则更类似于变量赋值的规则。因此，当一个迭代器执行 yield k,v 来调用一个声明了 |key,value| 形参的代码块时，赋值过程等价于下面的赋值语句：

```
key,value = k,v
```

迭代器 Hash.each_pair 像下面这样传递一个键/值对（注 7）：

```
{:one=>1}.each_pair {|key,value| ... } # key=:one, value=1
```

在 Ruby 1.8 里，以变量赋值的方式来理解代码块调用时的形参赋值过程会更清晰一些。回忆一下在 Ruby 1.8 里，只有当外围方法中不存在和代码块形参同名的局部变量时，这些形参才是局部于代码块的。如果已经存在了同名的局部变量，那么代码块形参的赋值过程就会对那些局部变量进行赋值。事实上，在 Ruby 1.8 里，任何形式的变量都可以被作为代码块形参，包括全局变量和实例变量：

```
s{:one=>1}.each_pair {|$key, @value| ... } # No longer works in Ruby 1.9
```

这个迭代器将全局变量$key 设置成:one，将实例变量@value 设置成 1。正如已经提到的那样，Ruby 1.9 使代码块形参的作用域范围严格地局部于代码块本身。这也就意味着，全局或实例变量不再是合法的代码块形参了。

迭代器 Hash.each 将一个键/值对以一个包含两个元素的数组的形式传递给代码块。尽管如此，下面这样的代码也很常见：

```
hash.each {|k,v| ... }  # key and value assigned to params k and v
```

这种代码之所以能运行，是因为并行赋值的缘故。被传递的那个两元素数组会像下面这样赋值给变量 k 和 v：

```
k,v = [key, value]
```

按照并行赋值的规则（请参见第 4.5.5 节），等号右侧的一个数组被拆成单个的元素，并且被赋值给左侧的多个变量。

但是代码块形参的赋值过程和并行赋值也不是完全一样的。设想有一个迭代器将两个值传递给它的代码块。按照并行赋值的规则，我们或许希望出现这样的情况：如果代码块只接受一个参数，那么 Ruby 就会自动地将那两个参数组成一个数组，然后再赋给那个单一的参数。但是事实并非如此：

```
def two; yield 1,2; end  # An iterator that yields two values
two {|x| p x }           # Ruby 1.8: warns and prints [1,2],
two {|x| p x }           # Ruby 1.9: prints 1, no warning
two {|*x| p x }          # Either version: prints [1,2]; no warning
two {|x,| p x }          # Either version: prints 1; no warning
```

在 Ruby 1.8 中，当代码块只有一个形参时，多个实参将先被组成一个数组，然后才被赋值。但是这种做法已经不再被推荐了，而且它会产生一个警告信息。在 Ruby 1.9 中，第一个实

注 7：Ruby 1.8 里的 each_pair 将两个独立的值传递给代码块。Ruby 1.9 里的 each_pair 迭代器是 each 迭代器的同义词，如我们即将提到的，它传递一个单独的数组实参给代码块。这里所展示的代码在两个版本下都能正常运行：

参被赋给代码块形参，第二个实参则被安静地忽略掉。如果我们希望多个实参能够被自动地组成一个数组，然后再赋值给单一的代码块形参，那么我们就必须像方法调用那样，在形参前面加一个*作为前缀。（有关方法形参和方法声明的详细讨论，请参见第 6 章。）此外我们还可以显式地丢弃第二个实参，只需在一个代码块形参列表后面加一个逗号即可。这就好比在说："确实存在第二个形参，但是它并没有被使用，而且我也懒得去给它起名字了。"

在这种情况下，代码块调用的行为既不像并行赋值，也不像方法调用。如果我们声明了一个具有单一形参的方法，却试图传递两个实参给它，那么 Ruby 的反应不会只是打印一个警告那么简单，而是抛出一个错误。

在 Ruby 1.8 里，只有最后一个代码块形参才能具有一个*前缀。Ruby 1.9 则取消了这个限制，无论代码块形参在参数列表中的位置如何，都可以具有一个*前缀：

```
def five; yield 1,2,3,4,5; end    # Yield 5 values
five do |head, *body, tail|       # Extra values go into body array
  print head, body, tail          # Prints "1[2,3,4]5"
end
```

和方法调用一样，yield 语句也允许不带花括号的哈希作为其最后一个实参值（请参见第 6.4.4 节）。也就是说，如果 yield 的最后一个实参是一个哈希字面量，那么你可以省略花括号。由于 yield 哈希字面量的迭代器并不多见，所以我们只能编造一个例子来说明这一点：

```
def hashiter; yield :a=>1, :b=>2; end     # Note no curly braces
hashiter {|hash| puts hash[:a] }          # Prints 1
```

在 Ruby 1.9 中，最后一个代码块形参可以具有一个&前缀，表明它将接受与该代码块调用相关的任何代码块。但是，回忆一下，一个 yield 调用也许并没有相关的代码块。在第 6 章里，我们将了解到可以将一个代码块转换成一个 Proc 对象，而且**可以**将 Proc 调用与代码块关联起来。一旦你阅读了第 6 章的内容，将很容易理解下面的代码例子：

```
# This Proc expects a block
printer = lambda {|&b| puts b.call }  # Print value returned by b
printer.call { "hi" }                 # Pass a block to the block!
```

代码块形参和方法形参之间的一个重要差别就是，代码块形参不允许有默认值，但是方法形参可以。也就是说，下面的写法是**非法的**：

```
[1,2,3].each {|x,y=10| print x*y }     # SyntaxError!
```

Ruby 1.9 定义了一种创建 Proc 对象的新语法，这种新语法允许参数具有默认值。相关细节要等到第 6 章才讲述，但是上述代码可以被重写成下面这样：

```
[1,2,3].each &->(x,y=10) { print x*y }  # Prints "102030"
```

5.5 改变控制流
Altering Control Flow

除了条件式、循环及迭代器之外，Ruby 还支持一些可以改变 Ruby 程序控制流的语句。它们是：

`return`

使一个方法退出并且向其调用者返回一个值。

`break`

使一个循环（或迭代器）退出。

`next`

使一个循环（或迭代）跳过当前迭代的剩余部分，进入下一个迭代。

`redo`

从头开始一个循环或迭代器的当前迭代。

`retry`

重新开始一个迭代器，重新对整个表达式进行求值。如我们在本章后面将要见到的，`retry` 关键字还能用于异常处理。

`throw/catch`

一种多用途的控制结构，其命名和行为都类似于一种异常传播和处理机制。`throw` 和 `catch` 并不是 Ruby 中首要的异常处理机制（该机制是 `raise` 和 `rescue`，本章后面将描述）。相反地，它们被用作一种多级或多标签的 `break`。

接下来的子章节将详细地讲述上述各个语句。

5.5.1 return

`return` 语句使其外围方法返回到其调用者。如果你熟悉 C、Java 或相关的语言，那么你或许已经对 `return` 语句有了一种直观的理解。但是请不要跳过本节，因为对你来说，代码块中的 `return` 语句的行为也许并不那么直观。

作为一种选择，`return` 语句可以后接一个表达式，或者一个逗号分隔的表达式列表。如果没有表达式，那么该方法的返回值是 `nil`。如果有一个表达式，那么该方法的返回值是该表达式的值。如果在 `return` 关键字后面有多个表达式，那么该方法的返回值就是一个数组，它包含了所有这些表达式的值。

大多数方法都不要求有一个 `return` 语句。当控制流程到达方法尾部时，该方法会自动地返回到它的调用者。在这种情况下，该方法的返回值就是最后的那个表达式的值。大多数程序员都会省略那些不必要的 `return` 语句。他们不将一个方法的最后一行写成 `return x`，而只简单地写成 `x`。

如果你希望从一个方法中提前返回，或者希望返回多个值，那么 return 语句就有用武之地了。比如：

```
# Return two copies of x, if x is not nil
def double(x)
  return nil if x == nil      # Return prematurely
  return x, x.dup             # Return multiple values
end
```

当初次学习 Ruby 代码块时，你可能很自然地将它们理解成某种嵌套函数或迷你函数（mini-method）。而且如果真的以这种方式来理解它们，你也许会认为代码块中的 return 只是简单地使代码块返回到那个调用它的迭代器。但是代码块并不是方法，它里面的 return 关键字的行为也并非如此。事实上，return 的行为非常一致，无论它位于嵌套得多深的代码块中，它总会使得外围方法返回（注 8）。

请注意，外围方法并不等于调用方法。当一个 return 语句位于一个代码块中时，它不仅会使得该代码块返回，还会使得那个调用代码块的迭代器返回，而且它还会使得外围方法返回。外围方法，也称为**词法上的外围**方法，是指当你查看源代码时，那个包围着该代码块的方法。图 5-2 阐明了一个代码块中的 return 语句的行为。

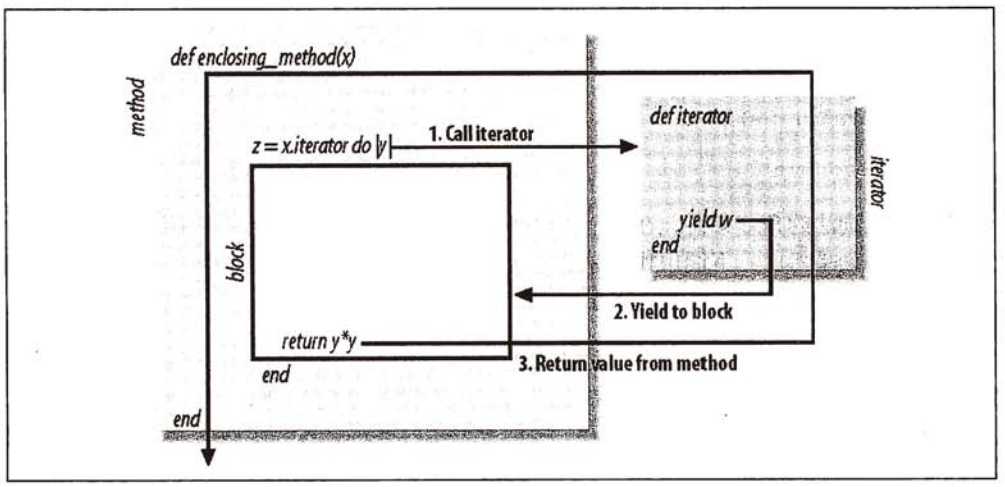


图 5-2：一个代码块中的 return 语句

下面的代码定义了一个方法，它使用 return 从一个代码块中返回：

```
# Return the index of the first occurrence of target within array or nil
# Note that this code just duplicates the Array.index method
def find(array, target)
  array.each_with_index do |element,index|
```

注 8：当我们在第 6.5.5.1 节介绍 lambda 表达式时，会见到一种例外情况。一个 lambda 是一种从代码块创建而来的函数，一个 lambda 中的 return 的行为不同于一个普通的代码块中的 return 的行为。

```
    return index if (element == target)  # return from find
  end
  nil  # If we didn't find the element, return nil
end
```

上述代码中的 return 语句不仅使代码块返回到 each_with_index 迭代器那里，而且返回 each_with_index 迭代器，同时它还使 find 方法返回一个值到其调用者。

5.5.2 break

当被用在一个循环中时，break 语句将控制权传递到循环外紧接着循环的第一个表达式。那些了解 C、Java 或一门类似语言的读者应该已经熟悉了一个循环中 break 的用法：

```
while(line = gets.chop)       # A loop starts here
  break if line == "quit"     # If this break statement is executed...
  puts eval(line)
end
puts "Good bye"               # ...then control is transferred here
```

当被用在一个代码块中时，break 不但将控制权传递出代码块，而且传递到那个调用代码块的迭代器之外，到达该迭代器调用后的第一个表达式。比如：

```
f.each do |line|              # Iterate over the lines in file f
  break if line == "quit\n"   # If this break statement is executed...
  puts eval(line)
end
puts "Good bye"               # ...then control is transferred here
```

如你所见，从词法的角度来讲，在一个代码块中使用 break 和在一个循环中使用 break 是一样的。但是如果考虑一下调用栈，你就会发现代码块中的 break 更复杂一些，因为它使与代码块相关联的迭代器方法也返回了。图 5-3 阐明了这一点。

请注意，和 return 不一样，break 不会使词法上的外围方法返回。break 只能出现在一个词法上的外围循环或代码块里，在任何其他上下文里使用 break 都会导致一个 LocalJumpError。

5.5.2.1 带一个值的 break

回忆一下，Ruby 中所有的句法结构都是表达式，都可以具有一个值。break 语句可以为它所跳出的循环或迭代器指定一个值，也可以在 break 关键字后面加上一个表达式，或者一个逗号分隔的表达式列表。如果 break 后面没有表达式，那么循环表达式的值，或者迭代器方法的返回值就是 nil。如果 break 后面有一个表达式，那么该表达式的值就成为循环

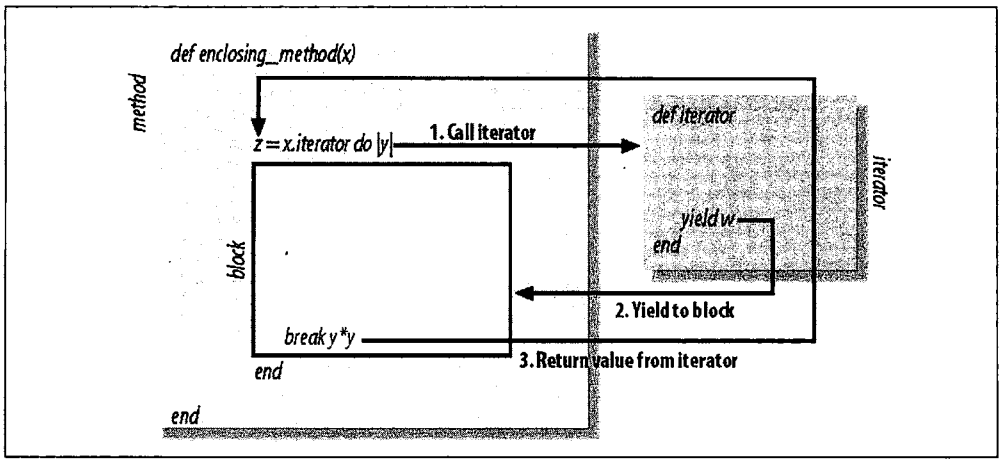


图 5-3：一个代码块中的 break 语句

表达式的值，或者成为迭代器的返回值。如果 break 后面有多个表达式，那么这些表达式的值会被放到一个数组里，该数组成为循环表达式的值，或者成为迭代器的返回值。

与此相对，一个正常结束的、没有 break 语句的 while 循环的值总是 nil。一个正常结束的迭代器的返回值取决于该迭代器的定义。许多迭代器，比如 times 和 each，只是简单地返回那个调用它们的对象。

5.5.3 next

next 语句使一个循环或迭代器结束当前迭代并开始下一轮迭代。对于 C 和 Java 程序员来说，它的等价的控制结构为 continue。下面是循环中的 next：

```
while(line = gets.chop)       # A loop starts here
  next if line[0,1] == "#"    # If this line is a comment, go on to the next
  puts eval(line)
  # Control goes here when the next statement is executed
end
```

当用在一个代码块里的时候，next 使代码块立即结束，将控制权返回给迭代器方法，然后该迭代器可能会开始新一轮的迭代并再次调用该代码块：

```
f.each do |line|              # Iterate over the lines in file f
  next if line[0,1] == "#"    # If this line is a comment, go to the next
  puts eval(line)
  # Control goes here when the next statement is executed
end
```

从词法上来讲，在一个代码块中使用 next 和在一个 while、until 或 for/in 循环中使用 next 是一样的，但是当考虑调用序列的时候，你就会发现在代码块中使用 next 更复杂。图 5-4 阐明了这一点。

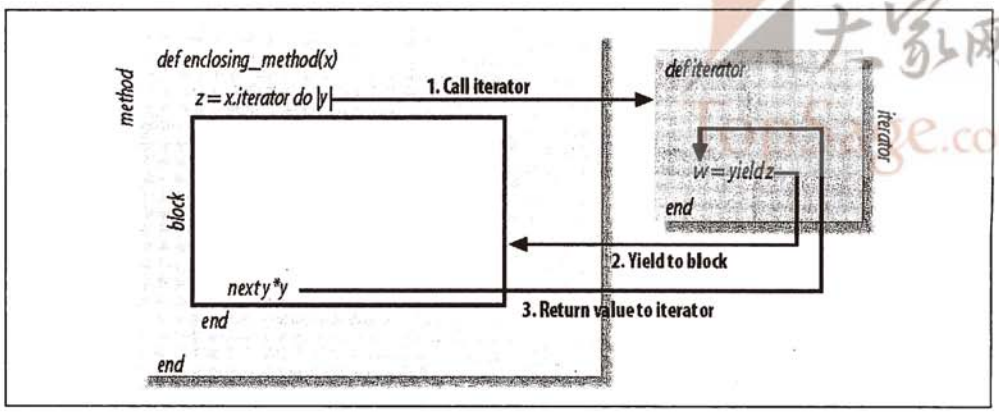


图 5-4：一个代码块中的 next 语句

next、break 和 return

将图 5-4 和图 5-2 及图 5-3 进行比较是很有意义的。next 语句使一个代码块返回到调用它的迭代器方法那里。break 语句使代码块返回到调用它的迭代器，同时使迭代器返回到外围方法。return 语句使代码块返回到调用它的迭代器，同时使迭代器返回到外围方法，而且返回外围方法到它的调用者。

next 只能被用在一个循环或代码块里，如果把它用在其他上下文里，就会抛出一个 LocalJumpError。

5.5.3.1 next 和代码块的值

类似于 return 和 break 关键字，next 可以单独使用，也可以后接一个表达式，还可以后接一个逗号分隔的表达式列表。当用在一个循环中时，next 之后的任何值都会被忽略掉。但是当用在一个代码块中时，next 之后的单个或多个表达式将成为那个调用该代码块的 yield 语句的“返回值”。如果 next 之后没有表达式，那么 yield 语句的值为 nil。如果 next 之后有一个表达式，那么该表达式的值成为 yield 的值。如果 next 后面有一个表达式列表，那么 yield 的值就是一个由那些表达式的值所构成的数组。

在我们先前关于 return 语句的讨论里，我们仔细解释了代码块和函数的差别，以及 return 语句不会使一个代码块返回到调用它的迭代器。如你所见，next 语句却正好可使一个代码块返回到调用它的迭代器。下面的代码展示了 next 的一种可能的用法：

```
squareroots = data.collect do |x|
  next 0 if x < 0  # Return 0 for negative values
```

```
  Math.sqrt(x)
end
```

一般情况下，一个 yield 表达式的值就是代码块中最后一个表达式的值。类似于 return 语句，通常情况下，并不需要显式使用 next 来指定一个值。比如，你可以像下面这样来编写上述代码：

```
squareroots = data.collect do |x|
  if (x < 0) then 0 else Math.sqrt(x) end
end
```

5.5.4 redo

redo 语句重新开始一个循环或迭代器的当前迭代，这一点和 next 不同。next 将控制权传递到一个循环或代码块的末尾，从而可以开始下一轮迭代，而 redo 则将控制权传递到循环或代码块的开头，从而可以重新开始当前迭代。如果在学习 Ruby 以前，你已经了解 C 一类的语言，那么对你来说，redo 可能是一种新的控制结构。

redo 将控制权传递给循环体或代码块的第一个表达式。它不会重新测试循环条件，也不会获取迭代器的下一个元素。下面的这个 while 循环，本来可以在三次循环后正常地结束，但是一个 redo 语句使它总共迭代了四次：

```
i = 0
while(i < 3)   # Prints "0123" instead of "012"
  # Control returns here when redo is executed
  print i
  i += 1
  redo if i == 3
end
```

redo 并不是一个常用的语句，许多像上面这样的例子都是杜撰出来的。它有一种用法是在提示用户输入的时候，从输入错误中恢复过来。下面的代码使用一个位于代码块中的 redo 来达到这样的目的：

```
puts "Please enter the first word you think of"
words = %w(apple banana cherry)   # shorthand for ["apple", "banana", "cherry"]
response = words.collect do |word|
  # Control returns here when redo is executed
  print word + "> "                # Prompt the user
  response = gets.chop             # Get a response
  if response.size == 0            # If user entered nothing
    word.upcase!                   # Emphasize the prompt with uppercase
    redo                           # And skip to the top of the block
  end
  response                         # Return the response
end
```

5.5.5 retry

retry 语句通常被用在一个 rescue 从句里，用来重新执行一段抛出了异常的代码。这将在第 5.6.3.5 节描述。但是在 Ruby 1.8 里，retry 还有另外一种用法：它可从头开始一个基于迭代器的迭代（或者任何方法调用）。retry 语句的这种用法非常少见，而且 Ruby 1.9 已经将它从语言中移除了，因此，这应该被当成一种不被推荐的语言特性，不应该再在新代码中使用。

在一个代码块中，retry 语句并不仅仅 redo 当前的代码块调用，它会使代码块及其迭代器方法退出，然后重新对迭代器表达式求值，重新开始迭代。考虑下面的代码：

```
n = 10
n.times do |x|          # Iterate n times from 0 to n-1
  print x               # Print iteration number
  if x == 9             # If we've reached 9
    n -= 1              # Decrement n (we won't reach 9 the next time!)
    retry               # Restart the iteration
  end
end
```

上述代码用 retry 来重启迭代，但是它小心地避免形成无限循环。在第一次调用时，它打印出数字 0123456789 之后重启；在第二次调用时，它打印出 0123345678，而且不再重启。

retry 语句的神奇之处在于，每次它重启迭代器的时候所做的事情并非完全一样。它完全重新计算迭代器表达式，这就意味着每次重启迭代器的时候，该迭代器的实参（甚至迭代器的调用对象）都可能不同。如果你还没有习惯像 Ruby 这样高度动态的语言，那么这种重新求值的过程可能显得有些违反直觉。

retry 语句的应用并不仅限于代码块，它通常只对最近的那个包含它的方法调用重新求值，这意味着它可以用于编写像下面这样的迭代器（在 Ruby 1.9 之前），这种迭代器的行为类似于 while 循环：

```
# This method behaves like a while loop: if x is non-nil and non-false,
# invoke the block and then retry to restart the loop and test the
# condition again. This method is slightly different than a true while loop:
# you can use C-style curly braces to delimit the loop body. And
# variables used only within the body of the loop remain local to the block.
def repeat_while(x)
  if x        # If the condition was not nil or false
    yield     # Run the body of the loop
    retry     # Retry and re-evaluate loop condition
  end
end
```

5.5.6 throw 和 catch
throw and catch

throw 和 catch 是 Kernel 模块的方法，它们定义了一种可以被理解成多级 break 的控制结构。throw 不但可以跳出当前的循环或代码块，而且可以向外跳出任意级数，使与 catch 一同定义的那个代码块退出。catch 不需要和 throw 处于同一个方法中，它既可以位于调用方法中，也可以位于调用栈中某个更深的地方。

像 Java 和 JavaScript 这样的语言允许使用任意的前缀来给循环命名或打标签。当给循环打完标签之后，一个称为“带标签的 break”的控制结构就能使该命名循环退出。Ruby 的 catch 方法定义了一个带标签的代码块，而 Ruby 的 throw 方法则可以使该代码块退出。但是 throw 和 catch 的用途要比一个带标签的 break 广泛得多：首先，它们可以和任意的语句一起使用，而不仅限于循环；更进一步，一个 throw 可以沿着调用栈向上传播，使一个位于调用方法中的代码块退出。

如果你熟悉 Java 和 JavaScript 这样的语言，那么你可能会将 throw 及 catch 和那些语言中用于抛出和处理异常的关键字等同起来。然而 Ruby 处理异常的方式和那些语言不同，它使用了 raise 和 rescue，这些我们在本章后面就会学到。但是这种和异常之间的相似性是有意而为的。调用 throw 非常类似于抛出一个异常。一个 throw 传播出当前词法范围并且沿着调用栈向上传递的行为也非常类似于一个异常的产生和向上传递的过程。（在本章后面，我们将学习更多有关异常传播的内容。）虽然它们和异常具有相似性，但你最好还是将 throw 和 catch 当成一种多用途的（虽然可能会很少被用到）控制结构，而不是一种异常机制。如果你希望发出一个错误或异常条件，那么请使用 raise 而不是 throw。

下面的代码展示了如何运用 throw 和 catch “跳出”嵌套的循环：

```
for matrix in data do          # Process a deeply nested data structure.
  catch :missing_data do       # Label this statement so we can break out.
    for row in matrix do
      for value in row do
        throw :missing_data unless value # Break out of two loops at once.
        # Otherwise, do some actual data processing here.
      end
    end
  end
  # We end up here after the nested loops finish processing each matrix.
  # We also get here if :missing_data is thrown.
end
```

请注意，catch 方法接受一个符号作为实参，并且还关联着一个代码块，它执行该代码块并且在其退出的时候返回，或者在指定的符号被抛出的时候返回。throw 也期望一个符号作为其实参，并且使对应的 catch 调用返回。如果没有和那个传递给 throw 的符号相匹

配的 catch 调用，那么它就抛出一个 NameError 异常。catch 和 throw 不仅可以接受符号实参，还可以接受字符串实参，字符串实参在内部被转换成符号实参。

throw 和 catch 的一个特性是，即使它们位于不同的方法中，也能正常工作。我们可以重构上述代码，将最深处的循环放到一个单独的方法中，即便如此控制流也能正常工作。

如果一个 throw 没有被调用，那么对应的 catch 调用就会返回其代码块中最后一个表达式的值。如果一个 throw 被调用了，那么默认情况下，对应的 catch 的返回值为 nil。但是，你可以向 throw 传递第二个参数，以此为 catch 指定任意的返回值。catch 的返回值不仅可以帮助你将代码块的正常结束和由 throw 引起的异常结束区分开来，而且作为对 throw 的响应，你还可以编写代码来进行任何必要的特殊处理。

在实际编码当中，throw 和 catch 并不常用。如果你发现自己正在同一个方法中使用 catch 和 throw，那么请考虑重构代码，将 catch 放到一个单独的方法中，并用 return 来替换 throw。

5.6 异常和异常处理

Exceptions and Exception Handling

一个**异常**是一个代表了某些异常条件的对象，它表明某些事情出了问题。这可能是一个编程错误：试图除以 0；试图在一个未定义某方法的对象上调用该方法；或者向一个方法传递一个无效的实参。这也可能是一些外部条件导致的结果：在断网的情况下发起一个网络请求，或者在系统已经没有可用内存的情况下试图创建一个对象。

当这些错误或条件中的一个发生时，程序就会**产生**（或**抛出**）一个异常。默认情况下，在异常发生时，Ruby 程序会终止；但是也可以定义异常处理器。一个异常处理器就是一段代码，如果在其他代码运行的过程中发生了异常，那么异常处理器代码就会被运行。从这种意义上去理解，异常是一种控制语句，抛出一个异常会把控制流传递给异常处理代码。这就好比使用 break 语句来退出循环一样。但是，如我们将要见到的，异常和 break 语句是大不相同的：为了达到异常处理器，它们可以将控制权从多层嵌套代码块中传递出来，甚至可以沿着调用栈不断上升。

Ruby 使用 Kernel 模块的 raise 方法来抛出异常，使用一个 rescue 从句来处理异常。由 raise 抛出的异常是 Exception 类或其某个子类的实例。本章先前描述的 throw 和 catch 方法并不是用来抛出和处理异常的，但是一个由 throw 抛出并传播的符号对象的行为和一个由 raise 抛出的异常对象的行为是一样的。在接下来的子章节里，本书将详细描述异常对象、异常传播、raise 方法及 rescue 从句。

5.6.1 异常类和异常对象
Exception Classes and Exception Objects

异常对象是 Exception 类或它的某个子类的对象。Exception 类有很多子类，这些子类通常并不定义新的方法或行为，但是使用它们，你可以按照类型来对异常进行分类。

```
Object
 +--Exception
     +--NoMemoryError
     +--ScriptError
     |   +--LoadError
     |   +--NotImplementedError
     |   +--SyntaxError
     +--SecurityError          # Was a StandardError in 1.8
     +--SignalException
     |   +--Interrupt
     +--SystemExit
     +--SystemStackError       # Was a StandardError in 1.8
     +--StandardError
         +--ArgumentError
         +--FiberError         # New in 1.9
         +--IOError
         |   +--EOFError
         +--IndexError
         |   +--KeyError       # New in 1.9
         |   +--StopIteration  # New in 1.9
         +--LocalJumpError
         +--NameError
         |   +--NoMethodError
         +--RangeError
         |   +--FloatDomainError
         +--RegexpError
         +--RuntimeError
         +--SystemCallError
         +--ThreadError
         +--TypeError
         +--ZeroDivisionError
```

图 5-5：Ruby 异常类的继承结构

你不必逐一地熟悉这些异常子类，它们的名字就会告诉你它们的用途。值得注意的是，这些子类中的大多数都继承自一个称为 StandardError 的类，它们是一些“普通的”异常，典型的 Ruby 程序会尝试着处理它们。其他异常则代表了那些更低层的、更严重的，或者可恢复性更小的情况，一般的 Ruby 程序都不会试图去处理它们。

如果试图使用 *ri* 来寻找有关这些异常类的文档，你将发现它们中的大多数都没有被文档化，

其部分原因是这些异常都没有添加新方法，而是沿用了 Exception 基类所定义的方法。了解一个给定异常类的最重要事情是它何时能被唤起，唤起的方法比异常类自己更能证明这一说法。

5.6.1.1 异常对象的方法

Exception 类定义了两个返回异常的细节信息的方法。message 方法会返回一个字符串，它提供了适合人类阅读的异常信息。如果一个 Ruby 程序带着一个未处理的异常退出了，那么通常这条信息就会被显示给最终用户，但是这条信息的主要目的并非如此，而是给程序员诊断问题时提供帮助。

异常对象的另一个重要方法是 backtrace。这个方法返回一个字符串数组，它代表了异常抛出点的调用栈。该数组的每个元素都是一个具有如下形式的字符串：

```
filename : linenumber in methodname
```

该数组的第一个元素指明了异常发生的位置；第二个元素指定了发生异常的那个方法的调用位置；第三个元素则指定了调用该方法的那个方法的调用位置；依此类推。（Kernel 方法 caller 也会以这种格式返回一个调用栈，你可以在 *irb* 里试验一下。）典型地，异常对象都是由 raise 方法来创建的。当创建完毕后，raise 方法会适当地设置该异常的栈轨迹。如果你创建了自己的异常对象，那么你可以通过 set_backtrace 方法来按照你的想法设置其栈轨迹。

5.6.1.2 创建异常对象

如我们将要见到的，异常对象典型地都是由 raise 方法创建的，但是你可以通过 new 方法，或者另一个名为 exception 的类方法来创建自己的异常对象。这两个方法都接受一个可选的字符串实参。如果指定了该参数，那么这个字符串就成为 message 方法的值。

5.6.1.3 定义新的异常类

如果你正在定义一个 Ruby 代码模块，那么定义一个 StandardError 类的子类，将其作为你的模块所特有的异常类，通常都是合宜的。它可能只是一个微不足道的，由如下一行代码来定义的子类：

```
class MyError < StandardError; end
```

5.6.2 使用 raise 抛出异常
Raising Exceptions with raise

Kernel 模块的 raise 方法能抛出一个异常。fail 是 raise 的同义词，当要抛出一个可使程序退出的异常时，你可用到它。这里有几种调用 raise 的方法：

- 如果调用 raise 的时候没有实参，那么它创建一个新的 RuntimeError 对象（没有消息）并将其抛出；或者，如果在一个 rescue 从句里不带实参地调用 raise，那么它简单地将正在处理的对象重新抛出。

- 如果以一个 Exception 对象作为实参来调用 raise，那么它抛出该异常。虽然这很简单，但这并不是一种常见的使用 raise 的方法。

- 如果以一个字符串实参来调用 raise，那么它创建一个新的 RuntimeError 异常对象，将指定的字符串作为该异常对象的消息并且将其抛出。这是一种常见的 raise 使用方法。

- 如果传递给 raise 的第一个实参是一个具有 exception 方法的对象，那么 raise 调用 exception 方法并将其返回的异常对象抛出。因为 Exception 类定义了一个 exception 方法，所以你可以将任何异常的类对象作为 raise 方法的第一个参数。

  raise 接受一个字符串作为其可选的第二个参数。如果指定了一个字符串，那么它就会被传递给第一个实参的 exception 方法。这个字符串将用作异常消息。

  raise 还可接受一个可选的第三个参数。你可以指定一个字符串数组，它们会作为该异常对象的回溯轨迹（backtrace）。如果没有指定第三个参数，那么 raise 将自行设置该异常对象的回溯轨迹（使用 Kernel 模块的 caller 方法）。

下面的代码定义了一个简单的方法，如果调用该方法时，其形参接受的是一个无效的值，那么它就会引发一个异常：

```
def factorial(n)                    # Define a factorial method with argument n
  raise "bad argument" if n < 1     # Raise an exception for bad n
  return 1 if n == 1                # factorial(1) is 1
  n * factorial(n-1)                # Compute other factorials recursively
end
```

这个方法用一个字符串实参来调用 raise。下面是一些抛出同样异常的等价方法：

```
raise RuntimeError, "bad argument" if n < 1
raise RuntimeError.new("bad argument") if n < 1
raise RuntimeError.exception("bad argument") if n < 1
```

在这个例子中，也许一个类型为 ArgumentError 的异常比 RuntimeError 异常更合适一些：

```
raise ArgumentError if n < 1
```

此外一个提供了更多细节信息的错误消息会更有帮助：

```
raise ArgumentError, "Expected argument >= 1. Got #{n}" if n < 1
```

我们在这里抛出异常的目的是指出调用 factorial 方法时出现的问题，而不是说该方法内部的代码有问题。由这里的代码所抛出的异常具有一个回溯轨迹，它的第一个元素标识了异常的抛出地点，第二个元素则标识了那个以错误实参调用 factorial 的方法。如果我们

希望直接指向问题代码，那么我们可以将 Kernel 模块的 caller 方法作为 raise 的第三个实参，以此来提供一个自定义的栈轨迹。

```
if n < 1
  raise ArgumentError, "Expected argument >= 1. Got #{n}", caller
end
```

请注意，factorial 方法只是检查它的实参是否处于正确的范围内，并没有检查它的类型是否正确。我们可以将下面的代码作为该方法的第一行代码，以此来进行更为仔细的错误检查。

```
raise TypeError, "Integer argument expected" if not n.is_a? Integer
```

在另一方面，请注意，如果我们传递一个字符串实参给先前编写的 factorial 方法，那么将发生什么呢？Ruby 使用<操作符来比较实参 n 和整数 1，如果实参是一个字符串，那么比较就没有意义，而且会抛出一个 TypeError。如果实参是某些没有定义<操作符的类的一个实例，那么就抛出一个 NoMethodError。

这里的关键点在于，即使我们在自己的代码里没有调用 raise 方法，程序也可能会抛出异常。因此，即使我们不会自行抛出那些异常，但是了解如何处理它们也是非常重要的。下一节就将讲述异常处理。

5.6.3 使用 rescue 来处理异常

Handling Exceptions with rescue

raise 是 Kernel 模块的一个方法。相反地，rescue 从句则是 Ruby 语言的一个基础部分。rescue 本身并不是一个语句，而是一个可以附加到其他 Ruby 语句的从句。最常见的情况是，将一个 rescue 从句附加到一个 begin 语句上。begin 语句的作用非常简单，就是为了区隔出一块代码，在这块代码中抛出的异常会被处理。一个带 rescue 从句的 begin 语句看起来像下面这样：

```
begin
  # Any number of Ruby statements go here.
  # Usually, they are executed without exceptions and
  # execution continues after the end statement.
rescue
  # This is the rescue clause; exception-handling code goes here.
  # If an exception is raised by the code above, or propagates up
  # from one of the methods called above, then execution jumps here.
end
```

5.6.3.1 命名异常对象

在一个 rescue 从句里。全局变量$!引用了当前正在处理的 Exception 对象。感叹号是一个助记符，一个异常就像一种令人惊叹的事情。如果你的程序包含了下面这一行：

```
require 'English'
```

那么你就可以使用全局变量$ERROR_INFO来代替$!。

在rescue从句里为异常对象指定一个变量名，比使用$!或$ERROR_INFO更好一些：

```
rescue => ex
```

如此一来，这个rescue从句里的语句就可以使用变量ex来引用那个描述异常的Exception对象。比如：

```
begin                                  # Handle exceptions in this block
  x = factorial(-1)                    # Note illegal argument
rescue => ex                           # Store exception in variable ex
  puts "#{ex.class}: #{ex.message}"    # Handle exception by printing message
end                                    # End the begin/rescue block
```

请注意，rescue从句并没有定义一个新的变量作用域。一个在rescue从句中命名的变量，在rescue从句之后仍然是可见的。如果你在rescue从句里用了一个变量来引用异常对象，那么该异常对象在rescue结束之后仍然可见，即便此时$!已经不再引用异常变量了。

5.6.3.2 按类型处理异常

目前所展示的rescue从句可处理任何属于StandardError（或其子类）的异常，但是会忽略其他的异常。如果你希望处理一个StandardError继承体系之外的非标准的异常，或者仅仅希望处理特定类型的异常，那么你必须在rescue从句里包含一个或多个异常类。下面代码中的rescue从句能够处理任何异常：

```
rescue Exception
```

下面代码中的rescue从句能够处理一个ArgumentError，并将异常对象赋值给变量e：

```
rescue ArgumentError => e
```

回忆一下，之前我们定义的factorial方法能够抛出ArgumentError或TypeError。下面的代码能够处理这些异常并且将异常对象赋值给变量error：

```
rescue ArgumentError, TypeError => error
```

最终我们在这里见到了rescue从句最通用的形式：rescue关键字后接一个由零个或多个逗号分隔的表达式，每个表达式都必须是一个代表了Expression类或其子类的类对象。这些表达式后面可以可选地接一个=>和一个变量名。

现在，假定我们想处理ArgumentError和TypeError，但是希望以不同的方式来处理它们。我们可以使用一个case语句，根据异常对象所属的类来运行不同的代码。但是更为优雅的做法是使用多个rescue从句。一个begin语句可以具有零个或多个rescue从句：

```
begin
  x = factorial(1)
rescue ArgumentError => ex
  puts "Try again with a value >= 1"
rescue TypeError => ex
  puts "Try again with an integer"
end
```

请注意，Ruby 解释器将按照 rescue 从句编写的顺序来匹配异常，所以，你应该把最特殊的异常子类放到第一位，然后依次接上那些更加一般的异常类型。举个例子，如果你想以不同于 IOError 的方式来处理 EOFError，那么就要将处理 EOFError 的 rescue 从句放在第一位，否则处理 IOError 的代码就会处理 EOFError。如果你想要一个“捕获全部”的 rescue 从句，用来处理所有没被之前的 rescue 从句处理的异常，那么你可以将 rescue Exception 作为最后一个 rescue 从句。

5.6.3.3 异常的传播

既然我们已经介绍了 rescue 从句，那么现在就可以解释更多异常传播的细节。当一个异常发生时，程序的控制权会立刻向外和向上传递，直到找到一个合适的 rescue 从句来处理该异常。当执行 raise 方法时，Ruby 解释器会查看那个包含它的代码块是否有相关联的 rescue 从句。如果没有（或者现有的 rescue 从句不是用来处理那种异常的），那么解释器就在那个包含了当前代码块的代码块里查找。如果在调用 raise 的方法里没有一个合适的 rescue 从句，那么该方法就会退出。

一个方法异常退出，和正常的返回是不一样的。该方法没有返回值，而且异常对象会从调用该方法的地方开始继续传播。异常向外传播，试图在外围代码块里查找一个能处理它的 rescue 从句。如果找不到这样的 rescue 从句，那么该方法就返回到它的调用者。在整个调用栈上，这样的过程不断重复并向上传播。如果最终也没有找到一个异常处理器，那么 Ruby 解释器就会打印出异常消息和回溯轨迹，然后退出。下面的代码提供了一个具体的例子：

```
def explode             # This method raises a RuntimeError 10% of the time
  raise "bam!" if rand(10) == 0
end

def risky
  begin                 # This block
    10.times do         # contains another block
      explode           # that might raise an exception.
    end                 # No rescue clause here, so propagate out.
  rescue TypeError      # This rescue clause cannot handle a RuntimeError..
    puts $!             # so skip it and propagate out.
  end
  "hello"               # This is the normal return value, if no exception occurs.
end                     # No rescue clause here, so propagate up to caller.
```

```
def defuse
  begin                     # The following code may fail with an exception.
    puts risky              # Try to invoke print the return value.
  rescue RuntimeError => e  # If we get an exception
    puts e.message          # print the error message instead.
  end
end

defuse
```

在 explode 方法里将抛出一个异常，那个方法没有 rescue 从句，所以该异常向外传播到该方法的一个名为 risky 的调用者。risky 具有一个 rescue 从句，但是只能处理 TypeError 异常，不能处理 RuntimeError 异常。该异常从 risky 的词法代码块里被传播出来，然后向上传播到了 risky 的调用者——一个名为 defuse 的方法。defuse 具有一个能处理 RuntimeError 异常的 rescue 从句，所以控制权就被传递到这个 rescue 从句，异常传播过程也就结束了。

请注意，这段代码使用了一个带代码块的迭代器（Integer.times 方法）。为简单起见，我们说该异常从这个词法代码块中被传播出来。事实上，对于异常传播来说，代码块的行为更像方法调用。从代码块那里产生的异常会向上传播到调用它的迭代器。像 Integer.times 这样的预定义循环迭代器，本身并不做任何异常处理，所以该异常从 times 迭代器开始，沿着调用栈向上传播到 risky 方法。

5.6.3.4 在异常处理过程中发生的异常

如果在执行一个 rescue 从句的过程中发生了一个新的异常，那么原先被处理的那个异常被丢弃，新的异常从它产生的地方开始传播。请注意，这个新异常不能在它发生的那个 rescue 从句的后续从句里被处理。

5.6.3.5 rescue 从句中的 retry

当在一个 rescue 从句中使用 retry 语句时，它会重新运行该 rescue 从句所依附的那段代码。当一个异常是由一个短暂的故障所引起的，比如一个超负荷的服务器，那么用再试一次的方式来处理该异常也许就是有意义的。但是许多其他的异常，都反映了编程错误（TypeError、ZeroDivisionError）或者非短暂的故障（EOFError 或 NoMemoryError）。对于这些异常来说，retry 并不是合适的处理技术。

在下面这个简单的例子里，使用了 retry 来等待网络故障恢复。它试图读取一个 URL 的内容，在发生故障时就会重试。它最多尝试四次，并且使用“指数后退算法（exponential backoff）”来增加每次尝试之前的等待时间：

```
require 'open-uri'

tries = 0          # How many times have we tried to read the URL
begin              # This is where a retry begins
  tries += 1       # Try to print out the contents of a URL
  open('http://www.example.com/') {|f| puts f.readlines }
rescue OpenURI::HTTPError => e    # If we get an HTTP error
  puts e.message                  # Print the error message
  if (tries < 4)                  # If we haven't tried 4 times yet...
    sleep(2**tries)               # Wait for 2, 4, or 8 seconds
    retry                         # And then try again!
  end
end
```

5.6.4 else 从句

The else Clause

一个 begin 语句可以在所有 rescue 从句之后接一个 else 从句。你也许会猜测 else 从句是一个捕获所有的（catch-all）rescue 从句：它处理任何没有被之前的 rescue 从句处理的异常。但这并不是 else 的目的。else 从句是 rescue 从句的一个替代性选择，如果所有的 rescue 从句都不需要，那么就可以使用它。也就是说，如果 begin 语句中的代码执行完毕而且没有任何异常，那么 else 从句中的代码就被执行。

将代码放到一个 else 从句中和简单地将这些代码附加到 begin 语句的后面非常相似。惟一的差别在于当你使用一个 else 从句时，该从句中产生的任何异常都不归 rescue 从句处理。

Ruby 中 else 从句并不常用，但是如果想强调一块代码的正常结束和异常结束之间的差别，那么 else 从句就能够从编码风格方面提供帮助。

请注意，在没有任何 rescue 从句的情况下使用 else 从句是没什么意义的，Ruby 解释器允许这种情况出现但是会发出警告。在 else 从句的后面不可以出现 rescue 从句。

最后，只有在 begin 从句中的代码被执行完毕，而且控制流自然而然地被转移到 else 从句中时，程序才会执行 else 从句。如果在执行 begin 从句的过程中发生了异常，那么显然 else 从句就不会再被执行。此外，begin 从句中的 break、return、next，以及类似的语句都可能阻止 else 从句的执行（译注 3）。

5.6.5 ensure 从句

The ensure Clause

begin 语句还可以拥有一个位于最后的从句。ensure 从句是可选的，如果和其他从句同时出现，那么它必须出现在所有的 rescue 从句和 else 从句之后。此外，它还可以在没有任何 rescue 或 else 从句的情况下单独使用。

无论 begin 从句中的代码发生了什么事，ensure 从句中的代码都会被执行：

译注 3：因为这些语句都可能导致控制流提早地离开 begin 从句而转移到程序的其他地方，这样一来控制流就无法自然地转移到 else 从句，也就不可能执行 else 从句了。

- 如果代码正常地执行完毕，那么控制流转移到 else 从句——如果存在的话——然后转移到 ensure 从句。
- 如果代码要执行一个 return 语句，那么在返回之前，控制流会跳过 else 从句，直接转移到 ensure 从句并执行其中的代码。
- 如果 begin 从句的代码发生了异常，那么控制流转移到合适的 rescue 从句，然后会转移到 eusure 从句。
- 如果没有 rescue 从句，或者没有能够处理该异常的 rescue 从句，那么控制流直接转移到 ensure 从句，在异常传播到外围代码块或沿着调用栈向上传递之前，先执行 ensure 从句中的代码。

ensure 从句的目的在于确保完成那些收尾工作，比如关闭文件、断开数据库连接、提交或取消事务等。它是一个强大的控制结构。如果你在任何时候分配了一些资源（比如一个文件句柄或数据库连接），那么都应该使用它来确保进行适当的释放或清理工作。

请注意，ensure 从句使异常的传播更加复杂。在我们先前的讲解里，忽略了有关 ensure 从句的讨论。从异常的发生点到异常的处理点并非一蹴而就，而是经历了一个复杂的传播过程。Ruby 解释器向外查找外围代码块，向上查找调用栈。在每个 begin 语句中，它都会查找一个可能处理该异常的 rescue 从句。此外它还会查找相关的 ensure 从句，并执行所有遇到的 ensure 从句。

一个 ensure 从句可以取消一个异常的传播过程，只要它发起一些能够导致控制流转移的动作即可。如果一个 ensure 从句抛出了一个异常，那么新的异常传播过程将取代原有的。如果一个 ensure 从句包含一个 return 语句，那么异常传播将被停止，而且包含它的方法将被返回。诸如 break 和 next 之类的控制结构也具有类似的效果：停止当前的异常传播，转而进行指定的控制流转移。

一个 ensure 从句还使方法返回值的概念更加复杂。虽然 ensure 从句通常都用来确保即使发生异常，也能执行某些代码，但是它们也能确保在方法返回之前执行某些代码。如果 begin 语句的代码体中包含一个 return 语句，那么 ensure 从句中的代码会在方法返回其调用者之前执行。更进一步，如果一个 ensure 从句自身包含一个 return 语句，那么它将改变该方法的返回值。作为例子，下面的代码会返回 2：

```
begin
  return 1     # Skip to the ensure clause before returning to caller
ensure
  return 2     # Replace the return value with this new value
end
```

请注意，除非显式地使用一个 return 语句，否则 ensure 从句不会改变一个方法的返回值。比如，下面的方法会返回 1，而不是 2：

```
def test
  begin return 1 ensure 2 end
end
```

如果一个 begin 语句没有传播一个异常，那么该语句的值就是 begin、rescue 或 else 从句中最后执行的那个表达式的值，ensure 从句中的代码保证被执行，但是不会影响到 begin 语句的值。

5.6.6 Method、Class 和 Module 定义中的 rescue
rescue with Method, Class, and Module Definitions

在这个有关异常处理的讨论中，我们至始至终都将 rescue、else 及 ensure 关键字描述成 begin 语句的从句，事实上，它们还可以作为 def 语句（定义一个方法）、class 语句（定义一个类）及 module 语句（定义一个模块）的从句。本书将在第 6 章讲述方法定义，在第 7 章讲述模块定义。

下面的代码展示了一个带有 rescue、else 及 ensure 从句的方法定义的大纲：

```
def method_name(x)
  # The body of the method goes here.
  # Usually, the method body runs to completion without exceptions
  # and returns to its caller normally.
rescue
  # Exception-handling code goes here.
  # If an exception is raised within the body of the method, or if
  # one of the methods it calls raises an exception, then control
  # jumps to this block.
else
  # If no exceptions occur in the body of the method
  # then the code in this clause is executed.
ensure
  # The code in this clause is executed no matter what happens in the
  # body of the method. It is run if the method runs to completion, if
  # it throws an exception, or if it executes a return statement.
end
```

5.6.7 作为语句修饰符的 rescue
rescue As a Statement Modifier

除了可以作为一个从句，rescue 还能被用作一个语句修饰符。你可以在任何语句的后面接上 rescue 关键字和另一个语句，如果第一个语句发生了异常，那么就会执行第二个语句。比如：

```
# Compute factorial of x, or use 0 if the method raises an exception
y = factorial(x) rescue 0
```

相当于：

```
y = begin
      factorial(x)
    rescue
```

```
  0
end
```

语句修饰符这种语法的好处在于，不需要 begin 和 end 关键字。当以这种方式来使用时，rescue 必须被单独使用，不能带有异常类名和变量名。一个 rescue 修饰符能够处理任何 StandardError 异常，但是不能处理其他类型的异常。与 if 和 while 修饰符不同，rescue 修饰符的优先级高于赋值操作符的优先级（请参见第 4 章的表 4-2），这就意味着，rescue 修饰符只会应用到一个赋值操作的右侧（像上面的例子那样），而不会应用到整个赋值操作。

5.7 BEGIN 和 END
BEGIN and END

BEGIN 和 END 是 Ruby 的保留字，分别用于声明一些在 Ruby 程序的开始和结束时要执行的代码。（请注意，大写形式的 BEGIN 和 END 与小写形式的 begin 和 end 完全是两码事。）如果在一个程序中有多个 BEGIN 语句，那么它们将按照 Ruby 解释器遇见它们的先后顺序被执行。如果有多个 END 语句，那么它们将按照与 Ruby 解释器遇见它们的顺序相反的顺序被执行——也就是说，第一个遇见的 END 语句将被最后执行。在 Ruby 中，这些语句并不常用。它们来源于 Perl 语言，而 Perl 语言又从 awk 文本处理语言中继承了它们。

在 BEGIN 和 END 后面必须接上以下这些东西：一个左花括号、任意数量的 Ruby 代码、一个右花括号。花括号是必须的，而且不允许使用 do 和 end。比如：

```
BEGIN {
  # Global initialization code goes here
}

END {
  # Global shutdown code goes here
}
```

BEGIN 和 END 语句之间有一些微妙的差别。BEGIN 语句在任何其他代码之前执行，包括其外围代码，这意味着 BEGIN 语句定义了一个与外围代码完全分开的局部变量作用域。只有将 BEGIN 语句放在顶层代码中才是有意义的，因为即便一个 BEGIN 语句位于条件式或循环当中，它也会被执行，而且外围条件并不能影响到它的执行。请考虑如下代码：

```
if (false)
  BEGIN {
    puts "if";                    # This will be printed
    a = 4;                        # This variable only defined here
  }
else
  BEGIN { puts "else" }           # Also printed
end
```

```
10.times {BEGIN { puts "loop" }} # Only printed once
```

在上述代码中，不管 BEGIN 语句的上下文如何，与这三个 BEGIN 语句相关的代码都会被执行，而且仅执行一次。在 BEGIN 代码块内定义的变量在该代码块之外是不可见的，而且该代码块之外的所有变量都还没有被定义。

END 语句则有所不同。它们在正常的程序执行过程被执行，因此它们和外围代码共享局部变量。如果一个 END 语句位于一个未被执行的条件式中，那么与其相关的代码将不会被注册成在程序结束的时候被执行。即使一个 END 语句位于一个循环中且被执行了多次，与其相关的代码也只会被注册一次：

```
a = 4;
if (true)
  END {                              # This END is executed
    puts "if";                       # This code is registered
    puts a                           # The variable is visible; prints "4"
  }
else
  END { puts "else" }                # This is not executed
end
10.times {END { puts "loop" }}       # Only executed once
```

Kernel 方法 at_exit 为 END 语句提供了一个替代性选择，它可以注册一个代码块，该代码块刚好在 Ruby 解释器退出之前被执行。和 END 语句一样，与第一个 at_exit 调用相关的代码块将被最后执行。如果 at_exit 方法在一个循环中被调用了多次，那么与其相关的代码块也会在 Ruby 解释器退出时被执行多次。

5.8 线程、纤程和连续体

Threads, Fibers, and Continuations

本节将介绍线程，它是 Ruby 中用于并发执行的控制结构，此外本节还将介绍两个更加高深的控制结构，即纤程和连续体。

5.8.1 用于并发的线程

Threads for Concurrency

一个**执行线程**就是一个 Ruby 语句序列，其执行过程和 Ruby 解释器所执行的主语句序列的执行过程是并行的（或者看起来如此）。Ruby 用 Thread 对象表示线程，但是它们也可以被看成是处理并发的控制结构。第 9.9 节将提供 Ruby 并发编程的细节。本节只是一个简单的概览，展示了如何创建线程。

Ruby 编程中代码块的使用非常普遍，借助这种方式来创建新线程也非常容易，只需调用 Thread.new 并给它一个代码块即可，这会创建一个新的执行线程并且开始执行代码块中的

代码。与此同时，原先的线程从 Thread.new 的调用中返回并且继续执行后续的语句。当代码块中的语句被执行完毕，新线程就会结束。通过 Thread 对象的 value 方法可以获得代码块的返回值。(如果你在线程结束前调用 value 方法，那么调用者将被阻塞，直到该线程返回一个值为止。)

下面的代码展示了一种使用线程并行地读取多个文件内容的方法：

```
# This method expects an array of filenames.
# It returns an array of strings holding the content of the named files.
# The method creates one thread for each named file.
def readfiles(filenames)
  # Create an array of threads from the array of filenames.
  # Each thread starts reading a file.
  threads = filenames.map do |f|
    Thread.new { File.read(f) }
  end

  # Now create an array of file contents by calling the value
  # method of each thread. This method blocks, if necessary,
  # until the thread exits with a value.
  threads.map {|t| t.value }
end
```

请参见第 9.9 节获取有关 Ruby 线程和并发的更多信息。

5.8.2 用于协同例程的纤程

Fibers for Coroutines

Ruby 1.9 引入了一种名为**纤程**的控制结构，并且用类 Fiber 的对象来表示它。在其他场合，名称“纤程”通常表示一种轻量级线程，但是在 Ruby 中，将纤程描述成**协同例程** (*coroutine*) 更好一些，或者再准确一点，称其为**半协同例程** (*semicoroutine*)。协同例程最常见的用法是实现**生成器** (*generator*)。生成器是一种对象，它们可以先计算出一个部分结果，然后将该结果返回给调用者并保存计算的状态，使将来调用者可以恢复计算并获取下一个结果。在 Ruby 中，Fiber 类用于实现从内部迭代器 (比如 each 方法) 到枚举器或外部迭代器的自动转换。

请注意，纤程是一种高级控制结构，而且相对来说比较晦涩。大多数 Ruby 程序员永远都不需要直接使用 Fiber 类。如果你以前没有协同例程或生成器方面的编程经验，那么刚开始你或许会觉得它们不太好理解。如果是这样，请仔细研究本书的例子并且多做一些练习。

和线程一样，一个纤程也具有一个代码体。你可以用 Fiber.new 来创建一个纤程，并且给它一个代码块来指定需要执行的代码。与线程不同的是，一个纤程的代码体不会立刻被执行，你须要调用那个代表该纤程的 Fiber 对象的 resume 方法来开始该纤程的运行。在第一次调用一个纤程的 resume 方法时，程序控制权被传递到纤程体的起始部分，然后该纤程开始运行，直到运行完整个纤程体或遇到 Fiber.yield 这个类方法为止。Fiber.yield 方法

把程序控制权传回给调用者，使得对 resume 的调用返回。它还将保存纤程的状态，这样一来当下次调用 resume 方法时，该纤程就可以从上次离开的地方开始运行。下面是一个简单的例子：

```
f = Fiber.new {                # Line  1: Create a new fiber
  puts "Fiber says Hello"      # Line  2:
  Fiber.yield                  # Line  3: goto line 9
  puts "Fiber says Goodbye"    # Line  4:
}                              # Line  5: goto line 11
                               # Line  6:
puts "Caller says Hello"       # Line  7:
f.resume                       # Line  8: goto line 2
puts "Caller says Goodbye"     # Line  9:
f.resume                       # Line 10: goto line 4
                               # Line 11:
```

由于创建一个纤程的时候并不会立即执行其纤程体，所以上述代码一开始只是创建了一个纤程，直到第七行的时候，它才产生一些输出。对于 resume 和 Fiber.yield 的调用会来回地传递程序控制权，使来自纤程和调用者的输出彼此交织在一起。上述代码的输出结果如下：

```
Caller says Hello
Fiber says Hello
Caller says Goodbye
Fiber says Goodbye
```

值得注意的是，Fiber.yield 的“yielding”行为和 yield 语句的是不同的，Fiber.yield 将控制权从当前的纤程传回其调用者，而 yield 语句则将控制权从一个迭代器方法传递给与其相关的代码块。

5.8.2.1 纤程的实参和返回值

通过 resume 和 yield 方法的实参和返回值，纤程可以和其调用者交换数据。第一次调用 resume 时的实参将被传递给与该纤程相关的代码块，它们成为代码块形参的值。在后续调用中，resume 的实参值将成为 Fiber.yield 的返回值。相反地，Fiber.yield 的实参值成为 resume 的返回值。当纤程的代码块结束时，最后一个被执行的表达式的值将成为 resume 的返回值。下面的代码展示了该过程：

```
f = Fiber.new do |message|
  puts "Caller said: #{message}"
  message2 = Fiber.yield("Hello") # "Hello" returned by first resume
  puts "Caller said: #{message2}"
  "Fine"                          # "Fine" returned by second resume
end

response = f.resume("Hello")      # "Hello" passed to block
puts "Fiber said: #{response}"
```

```
response2 = f.resume("How are you?") # "How are you?" returned by Fiber.yield
puts "Fiber said: #{response2}"
```

调用者向纤程传递了两条消息，而纤程也向调用者返回了两条消息。代码的输出为：

```
Caller said: Hello
Fiber said: Hello
Caller said: How are you?
Fiber said: Fine
```

在调用者代码中，将传递给纤程的消息始终作为 resume 方法的实参，而来自纤程的反馈则是该方法的返回值。在纤程体中，纤程所接受的消息，除第一条以外，都来自 Fiber.yield 的返回值。此外，在所有反馈中，除了最后一条，都是 Fiber.yield 的实参。纤程所接受的第一条消息是通过代码块的形参得到的，最后一条反馈则来自代码块的返回值。

5.8.2.2 使用纤程实现生成器

到目前为止，我们所展示的那些有关纤程的例子都不够实际，下面我们将展示一些更加典型的用法。首先，我们编写一个斐波那契数（Fibonacci）生成器——一个纤程对象，在每次调用其 resume 方法时返回斐波那契数列中的下一个数，其代码如下：

```
# Return a Fiber to compute Fibonacci numbers
def fibonacci_generator(x0,y0)    # Base the sequence on x0,y0
  Fiber.new do
    x,y = x0, y0                  # Initialize x and y
    loop do                       # This fiber runs forever
      Fiber.yield y               # Yield the next number in the sequence
      x,y = y,x+y                 # Update x and y
    end
  end
end

g = fibonacci_generator(0,1)      # Create a generator
10.times { print g.resume, " " }  # And use it
```

上面的代码会打印斐波那契数列的头 10 个数字：

```
1 1 2 3 5 8 13 21 34 55
```

由于 Fiber 是一个令人困惑的控制结构，所以我们在编写生成器的时候也许会倾向于隐藏其 API。下面是另一个版本的斐波那契数生成器，它定义了自己的类并且实现了和枚举器一样的 next 和 rewind API：

```
class FibonacciGenerator
  def initialize
    @x,@y = 0,1
    @fiber = Fiber.new do
      loop do
        @x,@y = @y, @x+@y
        Fiber.yield @x
      end
    end
  end
```

```
  def next            # Return the next Fibonacci number
    @fiber.resume
  end

  def rewind          # Restart the sequence
    @x,@y = 0,1
  end
end

g = FibonacciGenerator.new        # Create a generator
10.times { print g.next, " " }    # Print first 10 numbers
g.rewind; puts                    # Start over, on a new line
10.times { print g.next, " " }    # Print the first 10 again
```

通过包含 Enumerable 模块及定义下面的 each 方法（我们首先在第 5.3.5 节使用过），我们可以让 FibonacciGenerator 类成为可枚举的：

```
def each
  loop { yield self.next }
end
```

相反地，假如我们有一个 Enumerable 对象，希望在它的基础上编写一个枚举器风格的生成器，那么我们可以使用这个类：

```
class Generator
  def initialize(enumerable)
    @enumerable = enumerable  # Remember the enumerable object
    create_fiber              # Create a fiber to enumerate it
  end

  def next                    # Return the next element
    @fiber.resume             # by resuming the fiber
  end

  def rewind                  # Start the enumeration over
    create_fiber              # by creating a new fiber
  end

  private
  def create_fiber            # Create the fiber that does the enumeration
    @fiber = Fiber.new do     # Create a new fiber
      @enumerable.each do |x| # Use the each method
        Fiber.yield(x)        # But pause during enumeration to return values
      end
      raise StopIteration     # Raise this when we're out of values
    end
  end
end

g = Generator.new(1..10) # Create a generator from an Enumerable like this
loop { print g.next }    # And use it like an enumerator like this
g.rewind                 # Start over like this
g = (1..10).to_enum      # The to_enum method does the same thing
loop { print g.next }
```

虽然学习 Generator 类的实现对我们很有帮助，但是这个类本身并没有比 to_enum 方法提供更多的功能。

5.8.2.3 高级的纤程特性

标准库的 fiber 模块提供了额外的、更加强大的纤程特性。要使用这些特性，你必须：

```
require 'fiber'
```

但是基于如下理由，你应该尽可能地避免使用这些额外的特性：

- 并不是所有的 Ruby 实现都支持这些特性，比如 JRuby 在当前的 Java 虚拟机上就不支持它们。
- 它们是如此地强大，以至于对它们的误用甚至能让 Ruby 虚拟机崩溃。

Fiber 类的核心特性实现了半协同例程。它们并不是真正的协同例程，因为在调用者和纤程之间存在着本质上的不对称性：调用者使用 resume 而纤程使用 yield。但是如果你使用了 fiber 库，那么 Fiber 类就具有了一个 transfer 方法，该方法使任何纤程都能把控制权传递给其他的纤程。在下面的例子中，两个纤程使用 transfer 方法来回地传递控制权(和值)：

```
require 'fiber'

f = g = nil

f = Fiber.new {|x|          # 1:
  puts "f1: #{x}"           # 2: print "f1: 1"
  x = g.transfer(x+1)       # 3: pass 2 to line 8
  puts "f2: #{x}"           # 4: print "f2: 3"
  x = g.transfer(x+1)       # 5: return 4 to line 10
  puts "f3: #{x}"           # 6: print "f3: 5"
  x + 1                         # 7: return 6 to line 13
}
g = Fiber.new {|x|          # 8:
  puts "g1: #{x}"           # 9: print "g1: 2"
  x = f.transfer(x+1)       #10: return 3 to line 3
  puts "g2: #{x}"           #11: print "g2: 4"
  x = f.transfer(x+1)       #12: return 5 to line 5
}
puts f.transfer(1)          #13: pass 1 to line 1
```

这段代码会产生如下的输出：

```
f1: 1
g1: 2
f2: 3
g2: 4
f3: 5
6
```

你或许永远都不须要使用 transfer 方法，但是它的存在有助于解释“纤程”这个名称。你

可以将纤程理解为一个执行线程内的独立执行路径，然而与线程不同的是，没有在纤程之间传递控制权的调度器，纤程必须通过 transfer 方法显式地调度它们自己。

除了 transfer 方法，fiber 库还定义了一个名为 alive?的实例方法来检测一个纤程是否还在运行。此外它还定义了一个名为 current 的类方法，用于返回当前掌握控制权的 Fiber 对象。

5.8.3 连续体
Continuations

连续体是另一种复杂晦涩的控制结构，大多数程序员永远都不须要使用它。一个连续体的表现形式包括 Kernel 模块的 callcc 方法及 Continuation 对象。在 Ruby 1.8 中，连续体是核心平台的一部分，但是在 Ruby 1.9 中，它们被纤程所替代并且被转移到了标准库中。要在 Ruby 1.9 中使用它们，你必须显式地 require 它们：

```
require 'continuation'
```

实现上的困难阻碍了其他 Ruby 实现（比如 JRuby，一个基于 Java 的实现）对连续体的支持。由于它们不再被很好地支持了，所以新的 Ruby 代码就不应该再使用它们。如果你有一些采用 Ruby 1.8 的代码依赖于连续体，那么你也许可以采用 Ruby 1.9 的纤程来替换它们。

Kernel 模块的 callcc 方法会执行与其相关的代码块，并且将一个新建的 Continuation 对象作为唯一的实参传递给该代码块。Continuation 对象有一个 call 方法，它使 callcc 调用返回到它的调用者，传递给 call 的值将成为 callcc 调用的返回值。从这个角度来看，callcc 类似于 catch，而 Continuation 对象的 call 方法则类似于 throw。

但是连续体还是非常独特的，因为你可以将 Continuation 对象保存到一个位于 callcc 代码块之外的变量中。你可以反复地调用该对象的 call 方法，使程序的控制流跳转到 callcc 调用后的第一条语句处。

下面的代码展示了如果使用连续体来定义一个方法，该方法的行为类似于 BASIC 编程语言中的 goto 语句：

```
# Global hash for mapping line numbers (or symbols) to continuations
$lines = {}

# Create a continuation and map it to the specified line number
def line(symbol)
  callcc {|c| $lines[symbol] = c }
end

# Look up the continuation associated with the number, and jump there
def goto(symbol)
  $lines[symbol].call
end

# Now we can pretend we're programming in BASIC
```

```
i = 0
line 10              # Declare this spot to be line 10
puts i += 1
goto 10 if i < 5     # Jump back to line 10 if the condition is met

line 20              # Declare this spot to be line 20
puts i -= 1
goto 20 if i > 0
```

方法、Proc、Lambda 和闭包

Methods,Procs,Lambdas, and Closures

方法是一个有名代码块，是与一个或多个对象相关联的参数化代码。**调用**方法时须要给出方法名、所在对象（有时称为**接收者**），以及零个或多个与有名参数对应的参数值。方法中最后一个表达式的值将作为方法调用的返回值。

许多语言会区分函数（没有关联对象）和方法（需要有一个接收者对象）。因为 Ruby 是一种纯粹的面向对象语言，所以所有方法都是“真正”的方法，至少会与一个对象相关联。目前，我们还没有讲到如何定义一个 Ruby 类，因此本章示例的方法定义看起来更像是没有关联对象的全局方法，但事实上，Ruby 会把它们隐式地定义为 `Object` 对象的私有方法。

方法是 Ruby 句法的基础部件之一，但它不是 Ruby 程序可以操作的值。也就是说，Ruby 方法不是像字符串、数字或数组那样的对象。不过，可以用一个 `Method` 对象表示一个特定的方法，然后通过 `Method` 对象间接调用该方法。

在 Ruby 中，方法并非参数化代码的唯一形式。在第 5.4 节定义的代码块（Block）也是一种可以带有参数的可执行代码段。跟方法不同，代码块没有名字，而且它们只能通过迭代器（iterator）方式被间接调用。

跟方法相似，代码块也不是 Ruby 可以操作的对象。不过可以创建一个对象来代表一个代码块，这种方法经常也使用在实际的 Ruby 程序中，用一个 `Proc` 对象代表一个代码块。与 `Method` 对象相似，我们可以通过 `Proc` 来执行它代表的代码块。有两种方式的 `Proc` 对象，一种被称为 *proc*，另一种被称为 *lambda*，它们有一些细微的差别。proc 和 lambda 的行为更像是函数而非方法，它们的一个重要特性是它们都是**闭包**（*closure*）：它们会保持定义时所在范围内的局部变量，即使在另外的范围内被调用时，它们也可以对这些变量进行访问。

方法的句法形式相当丰富和复杂，本章第一部分将对它进行详细说明。首先解释怎样定义简单的方法，然后后面三节将介绍比较高级的主题，包括方法名、方法括号和方法参数。请注意，方法调用是一种表达式，这已在第 4.4 节中讨论过，在本章前面四个小节中将介绍方法调用的更多细节。

在方法之后，将着重介绍 proc 和 lambda。本章将解释怎样创建和调用它们，以及详细介绍它们之间细微的差别。还有一个单独的章节说明如何把 proc 和 lambda 作为闭包使用。接下来的小节是关于 `Method` 对象的，它的行为与 lambda 很相似。在本章的最后，将探索用 Ruby 进行函数式编程这个高级课题。

6.1 定义简单方法
Defining Simple Methods

在本书中，你已经看到了很多方法调用的例子，在第 4.4 节也详细介绍了方法调用的句法细节。本节将解释方法定义的基础知识。在后面的三个小节，会涉及方法名、方法括号和方法参数的细节，这些小节解释一些关于方法定义和方法调用的高级课题。

关键字 def 用于方法定义，在其后是方法名和可选的参数名列表，参数名列表会用一对圆括号括住。构成方法主体的代码放在参数列表之后，end 关键字用于表明方法定义的结束。参数名在方法代码体中可以像变量一样使用，它们的值来自方法调用时的参数。下面是一个方法示例：

```
# Define a method named 'factorial' with a single parameter 'n'
def factorial(n)
  if n < 1                # Test the argument value for validity
    raise "argument must be > 0"
  elsif n == 1            # If the argument is 1
    1                     # then the value of the method invocation is 1
  else                    # Otherwise, the factorial of n is n times
    n * factorial(n-1)    # the factorial of n-1
  end
end
```

这段代码定义了一个名为 factorial 的方法，它有一个名为 n 的参数。n 标识符在方法代码体中可以被看作一个变量。这个方法是一个递归方法，所以在方法代码体中会调用方法自身。对这个方法的调用很简单，只要给出方法名和用圆括号括住的参数值即可。

6.1.1 方法返回值
Method Return Value

方法可能正常结束，也可能非正常结束。当方法抛出异常（exception）时，就会出现非正常结束的情况。在前面给出的 factorial 方法中，如果给定小于 1 的参数值，就会让方法非正常结束。如果方法正常结束，方法代码的最后一个表达式的值会作为方法调用表达式的值（译注 1）。在 factorial 方法中，方法代码的最后一个表达式要么是 1，要么是 n*factorial(n-1)。

return 关键字将在到达方法最后一个表达式前强行返回。如果一个表达式跟在 return 关键字之后，它的值会作为方法返回值。如果 return 后面没有任何表达式，那么返回值为 nil。在下面这个 factorial 方法的变体中，须要使用 return 关键字：

```
def factorial(n)
  raise "bad argument" if n < 1
```

译注 1：即方法返回值。

```
  return 1 if n == 1
  n * factorial(n-1)
end
```

对方法的最后一行也可以使用 return 关键字，这可以强调这个表达式用作这个方法的返回值。不过在一般的编程实践中，如非必须，会省略 return。

Ruby 方法可以返回多个值，这须要显式使用 return 语句，并把要返回的值用逗号分隔开：

```
# Convert the Cartesian point (x,y) to polar (magnitude, angle) coordinates
def polar(x,y)
  return Math.hypot(y,x), Math.atan2(y,x)
end
```

当有多个返回值时，这些值会被收集到一个数组中，然后这个数组将作为方法的唯一返回值，因此，也可以不用多个返回值，而直接用一个数组作为返回值：

```
# Convert polar coordinates to Cartesian coordinates
def cartesian(magnitude, angle)
  [magnitude*Math.cos(angle), magnitude*Math.sin(angle)]
end
```

通常我们想使用并行赋值时（参见第 4.5.5 节），会使用这种形式的方法定义。通过并行赋值，每个返回值会被赋给一个单独的变量。

```
distance, theta = polar(x,y)
x,y = cartesian(distance,theta)
```

6.1.2 方法和异常处理
Methods and Exception Handling

用 def 语句定义方法时，也可以包含异常处理代码。异常处理代码会使用 rescue、else 和 ensure 从句，这与 begin 语句的方式一致。在短小的方法中，用 rescue 子句和 def 语句配合让代码非常干净整洁，这意味着你不用使用一个多余的 begin 语句，也省去了一级额外的格式缩进。参见第 5.6.6 节可获得更多细节。

6.1.3 在对象上调用方法
Invoking a Method on an Object

方法永远是在一个对象上被调用的（这个对象有时被称为接收者，这是因为在面向对象的术语里，方法被称为“消息”，它们被“发送”给接收者对象）。在方法代码中，self 这个关键字代表方法被调用的对象。如果调用方法时不指定对象，这表示该方法在 self 上被调用。

在第 7 章将看到怎样为类定义方法。不过，你已经在本书中看到过如何调用一个对象的方法，比如：

```
first = text.index(pattern)
```

像绝大多数面向对象的编程语言一样，Ruby 使用“.”来分隔方法和调用的对象。这段代码把 pattern 变量传给名为 index 的方法，index 方法属于一个存储在 text 变量中的对象，方法的返回值将被存储到 first 变量中。

6.1.4 定义单键方法（译注 2）
Defining Singleton Methods

目前为止，我们定义的方法都是全局方法。如果把前面的那些 def 语句放在一个 class 语句中，这些方法就成为该类的实例方法，它们被定义在这个类的所有实例对象上。（类和实例方法将在第 7 章中解释。）

不过，也可以使用 def 语句为一个特定的对象定义方法，只须简单地在 def 关键字后加上一个求值结果为对象的表达式，在这个表达式之后需要一个句点符号和要定义的方法名。这样定义的方法被称为**单键方法**，因为它只在单个对象上可用：

```
o = "message"     # A string is an object
def o.printme     # Define a singleton method for this object
  puts self
end
o.printme         # Invoke the singleton
```

像 Math.sin 和 File.delete 这样的类方法（类方法在第 7 章中介绍）实际上是单键方法。Math 是一个常量，它引用一个 Module 对象；而 File 常量则引用一个 Class 对象。这两个对象分别有名为 sin 和 delete 的单键方法。

在实现中，Ruby 一般把 Fixnum 和 Symbol 的值看作立即值（immediate value）而非真正的对象引用（参见第 3.8.1.1 节），因此，Fixnum 和 Symbol 对象无法定义单键方法。为了保持一致性，其他的 Numeric 对象也不能定义单键方法。

6.1.5 取消方法定义
Undefining Methods

用 def 语句定义的方法可以用 undef 语句取消定义：

```
def sum(x,y); x+y; end        # Define a method
puts sum(1,2)                 # Use it
undef sum                     # And undefine it
```

在这段代码中，def 语句定义了一个全局方法，undef 语句又取消了这个定义。undef 也可以在类中使用（这是第 7 章的主题），用来取消实例方法的定义。有趣的是，undef 也可以用于取消继承方法的定义，而不影响所继承的类中该方法的定义。假设类 A 定义了一个方

译注 2：这里的单键方法和设计模式中的单键模式没有关系。

法 m，而类 B 是类 A 的子类，于是类 B 也继承了 m 方法（子类和继承也在第 7 章中解释）。如果不想让类 B 调用方法 m，可以在子类的代码中用 undef 取消 m 的定义。

undef 语句并不常用，在实践中，更常见的方式是使用 def 语句重新定义一个方法，来达到**取消**或删除这个方法的目的。

注意，undef 语句必须紧跟一个表示方法名的标识符。对于用 def 语句定义的单键方法，不能用它来取消定义。

在一个方法或模块中，还可以使用 undef_method 方法（它是 Module 的私有方法）来取消方法定义，采用此方式，需要把代表要取消方法名字的符号（symbol）传给 undef_method 方法。

6.2 方法名
Method Names

习惯上，方法名以小写字母打头。（方法名也可以用大写字母开头，但是那看着像常量名。）如果方法名超过一个单词，一般用下划线分隔不同单词，而不是使用像 likeThis 那样的大小写混排方式。

> **方法名解析**
>
> *本节讲述了在定义方法时给出的方法名，一个相关的主题是名字解析：Ruby 解释器如何找到方法调用表达式中方法名的定义？给出这个问题的答案必须要等到讨论了 Ruby 类之后。后者将在第 7.8 节中讲述。*

方法名可以（但并非必须）以等号、问号或感叹号结尾。等号后缀表示这个方法是一个赋值**方法**（*setter*），可以使用赋值句法进行调用。赋值方法已在第 4.5.3 节中描述，在第 7.1.5 节中可以找到更多的例子。以问号和感叹号结尾的方法对 Ruby 解释器没有什么特殊意义，允许使用它们是因为这样可以引入两个非常有用的命名习惯。

第一个命名习惯是：名称以问号结尾的方法应返回一个可以回答方法调用者给出问题的值，比如，数组的 empty? 方法在列表为空时返回 true。这样的方法被称为**断言**（*predicate*）。断言通常返回一个 true 或 false 的 Boolean 值，但这并非必须，因为对于 Boolean 值来说，除了 false 和 nil 之外的任何值都会被解释为 true。（比如，Numeric 类的 nonzero?方法，在给定的值为 0 时返回 nil，否则只是简单地返回给定的值。）

第二个命名习惯是：当一个方法名以感叹号结束时，我们需要小心调用该方法。比如 Array 对象有一个 sort 方法，这个方法产生一个该数组的拷贝，然后在这个拷贝上进行排序；该

对象还有一个 sort!方法，它对这个数组本身进行排序，感叹号表明在使用这个版本的方法时需要多加小心。

以感叹号结尾的方法通常是**可变**（*mutator*）方法，它会改变对象的内部状态。不过也并非总是如此，也有许多可变方法不以感叹号结尾，而一些不变方法以感叹号结尾，那些没有不变形式的可变方法（比如 Array.fill）通常不以感叹号结尾。

考查一下 exit 这个全局函数，它 Ruby 程序以一种可控的方式结束运行。这个方法有一个名为 exit!的变体，它让程序立刻结束，而不运行任何 END 块和任何用 at_exit 注册的钩子（hook）方法。exit!方法并非一个修改器方法，而是 exit 方法的一个“危险”变体，用“!”可以提醒程序员谨慎地使用这个方法。

6.2.1 操作符方法
Operator Methods

许多 Ruby 操作符，比如+、*和（甚至）数组索引符[]，是用定义在类中的方法实现的。通过定义一个与操作符同“名”的方法，可以定义一个操作符。（仅有的例外是一元加和一元减操作符，它们分别使用+@和-@来定义。）即使整个方法名都是标点符号也是可以的，可以像下面这样定义一个方法：

```
def +(other)              # Define binary plus operator: x+y is x.+(y)
  self.concatenate(other)
end
```

第 4 章的表 4-2 指出了哪些 Ruby 操作符可以通过方法来定义的。这些操作符是可以使用的全部操作符，除了这些操作符，不能定义使用其他标点符号的新操作符。在第 7.1.6 节可以看到更多定义基于方法的操作符的例子。

定义一元操作符的方法没有参数；定义二元操作符的方法有一个参数，这个方法应该对 self 和这个参数进行操作。数组访问操作符[]和[]=是特例，因为它们可以使用任意数目的参数，对于[]=，最后一个参数永远是被赋值的对象。

6.2.2 方法别名
Method Aliases

在 Ruby 中，方法有多个名字并非罕见。Ruby 提供了一个名为 alias 的关键字用于为已有方法定义一个新的名字。示例如下：

```
alias aka also_known_as   # alias new_name existing_name
```

执行完这个语句后，aka 标识符将与 also_known_as 标识符引用同一个方法。

方法别名让 Ruby 更富表现力，同时更像一门自然语言。当一个方法有多个名字时，可以选择一个看起来在代码中最自然的名字。比如，Range 类有个方法用于测试给定值是否处于范围之内，可以称呼这个方法为 include?或 member?。如果把范围看作一种集合，那么使用 member?可能是最自然的选择。

使用别名的一个更加实际的理由是可以为一个方法增加新的功能。下面是为已有方法增加功能的常用方式：

```
def hello                          # A nice simple method
  puts "Hello World"               # Suppose we want to augment it...
end

alias original_hello hello         # Give the method a backup name

def hello                          # Now we define a new method with the old name
  puts "Your attention please"     # That does some stuff
  original_hello                   # Then calls the original method
  puts "This has been a test"      # Then does some more stuff
end
```

在这段代码中用到了全局方法，而别名更常见于类的实例方法。（这将在第 7 章中讲述。）这时，alias 必须在要定义别名的方法所在的 class 定义中使用。Ruby 中的类可以被“重新打开”（这也将在第 7 章中讨论）——这意味着你的代码可以处理已经存在的类。用 class 语句“打开”它，然后像示例那样使用 alias 来修改或增加那个类的已有方法。这种方式被称为“别名链”，我们将在第 8.11 节对它进行详细讨论。

别名不是重载（overload）

一个 Ruby 方法可以有两个名字，但是两个方法不能共享一个名字。在静态语言中，可以通过参数数目和类型区分方法，因此只要两个方法有不同数目或类型的参数，它们就可以共享同一个方法名，不过这种方式的重载在 Ruby 中是不可能的。

不过，方法重载对 Ruby 并非必需。方法的参数可以接受各种类型的值；另外（我们会很快看到），Ruby 方法的参数可以被声明为默认值，这些参数在方法调用时可以被省略。这使得可以用不同数目的参数调用同一个方法。

6.3 方法和圆括号
Methods and Parentheses

在绝大多数方法调用中，Ruby 允许省略圆括号。在简单的情形下，这让代码看起来很整洁；不过在复杂的情形下，会导致歧义和混乱。在接下来的几个小节中，我们将讨论这种情况。

6.3.1 可选的圆括号
Optional Parentheses

在许多常用的 Ruby 惯例中，可以在方法调用时省略圆括号。比如，下面的两行代码是等价的：

```
puts "Hello World"
puts("Hello World")
```

在第一行中，puts 看起来像是一个 Ruby 内置的关键字、语句或命令，而等价的第二行则表明它不过是省略了圆括号的全局方法调用。尽管第二种方式更加明确，但第一种方式更简明、更常用，并且更自然。

接着，看看这段代码：

```
greeting = "Hello"
size = greeting.length
```

如果熟悉其他的面向对象语言，你可能会认为 length 是一个属性、字段或字符串类型的变量。然而，Ruby 是一种强面向对象编程语言，它的对象被完全封装，与它们进行交互的唯一方式就是调用它们的方法。在本例中，greeting.length 是一个方法调用，length 方法没有参数，而且在调用时也没有使用圆括号。下面的代码与本例等价：

```
size = greeting.length()
```

使用可选的圆括号可以强调这里发生了一个方法调用；而对没有参数的方法调用时省略圆括号，让它看起来是一个属性访问，这在实践中很常见。

如果调用的方法没有参数或仅有一个参数时，圆括号通常被省略。当有多个参数时，圆括号也可以省略（尽管不是很常见），如下所示：

```
x = 3                 # x is a number
x.between? 1,5        # same as x.between?(1,5)
```

在方法定义时，参数列表两端的圆括号也可以被省略，尽管很难说它让代码更清晰可读。比如，下面的代码定义了一个对参数求和的方法：

```
def sum x, y
  x+y
end
```

6.3.2 必需的圆括号
Required Parentheses

在某些代码中，如果省略圆括号就会产生歧义，因此 Ruby 要求加上它们。最常见的例子是方法的嵌套调用，如 `f g x, y`，在 Ruby 中，这种形式会被解释为 `f(g(x,y))`。Ruby 1.8 会对此发出一个警告，因为这段代码也可以被解释为 `f(g(x),y)`，不过这个警告在 Ruby 1.9 中被删除了。下面的代码使用了上面定义的 `sum` 方法，它会输出 `4`，不过在 Ruby 1.8 中还会得到一个警告：

```
puts sum 2, 2
```

要消除这条警告，可以用圆括号括住参数：

```
puts sum(2,2)
```

注意用圆括号括住外层方法的参数并不能消除二义性：

```
puts(sum 2,2)   # Does this mean puts(sum(2,2)) or puts(sum(2), 2)?
```

在嵌套方法调用时，只有在有多个调用参数时才会出现二义性。在下面的代码中，Ruby 只能用一种方式对之进行解释：

```
puts factorial x   # This can only mean puts(factorial(x))
```

尽管这里没有二义性，如果在 `x` 两端省略圆括号，Ruby 1.8 还是会发出一个警告。

有时，省略圆括号会产生句法错误。比如，缺少了圆括号，下面的表达式会变得模棱两可，Ruby 根本不想去猜你究竟想干什么：

```
puts 4, sum 2,2  # Error: does the second comma go with the 1st or 2nd method?
[sum 2,2]        # Error: two array elements or one?
```

可选的圆括号还会引发另外一个问题：在使用圆括号进行方法调用时，左括号**必须**紧跟方法名，不能带有留白。这是因为圆括号具有双重责任：它既被用于括住参数列表，也被用于进行表达式分组。考虑如下的两个表达式，它们的区别仅在于一个空格：

```
square(2+2)*2    # square(4)*2 = 16*2 = 32
square (2+2)*2   # square(4*2) = square(8) = 64
```

在第一个表达式中，圆括号表示方法调用，而在第二个表达式中则代表表达式分组。为了消除潜在的二义性，在参数本身带有圆括号时，应该总是使用圆括号来进行方法调用。第二个表达式应该使用下面这种更清晰的方式：

```
square((2+2)*2)
```

让我们用最后一个例子（twist）完成对圆括号的讨论。你应该记得下面的表达式是有歧义的，它会引发一个警告：

```
puts(sum 2,2)   # Does this mean puts(sum(2,2)) or puts(sum(2), 2)?
```

消除这种歧义的最好方法是对 sum 方法的参数使用圆括号，另外一种方法是在 puts 和左括号之间放入一个空格：

```
puts (sum 2,2)
```

增加这个空格会把用于方法调用的圆括号转换为表达式分组圆括号。这样一来圆括号用于分组子表达式，逗号就不再被解释为 puts 调用的参数分隔符了。

6.4 方法参数
Method Arguments

简单的方法声明会在方法名后面包含一个参数名列表（在可选的圆括号内），但是关于方法参数还有更多的内容，本节的子小节会覆盖如下内容。

- 怎样声明一个具有默认值的参数，这样可以在方法调用时省略该参数。
- 怎样声明一个可以接受任意数目参数的方法。
- 怎样用一种特殊的语法来传递一个哈希对象给一个方法，使这个方法像使用了有名参数一样。
- 怎样声明一个方法，使关联块在方法调用时被当作方法的参数。

6.4.1 参数默认值
Parameter Defaults

在定义方法时，可以为一些或全部参数指定默认值，这样，在方法调用时就可能使用比较少的参数。如果有参数被省略，就使用该参数的默认值。在参数名后加入等号和一个值，就可以为该参数指定一个默认值：

```
def prefix(s, len=1)
  s[0,len]
end
```

这个方法声明了两个参数，但是第二个参数有默认值，这意味着可以使用一个或两个参数来调用这个方法：

```
prefix("Ruby", 3)     # => "Rub"
prefix("Ruby")        # => "R"
```

参数的默认值不一定必须是常量：它们可以是任意表达式，也可以是实例变量的引用，还可以是前面定义参数的引用。比如：

```
# Return the last character of s or the substring from index to the end
def suffix(s, index=s.size-1)
  s[index, s.size-index]
end
```

参数默认值在运行时被求值，而不是在解析时被求值。在下面的方法中，默认值[]在每次方法调用时产生一个新的空数组，而并非在数组定义产生一个数组，然后不断重用它：

```
# Append the value x to the array a, return a.
# If no array is specified, start with an empty one.
def append(x, a=[])
  a << x
end
```

在Ruby 1.8中，带有默认值的参数必须出现在普通参数的后面，Ruby 1.9则放宽了这一限制，不过它仍要求所有带默认值的参数必须连续出现——不能在它们之间放入一个普通的参数。如果一个方法有多个带默认值的参数，并且在方法调用时仅指定了部分参数，这些参数会按照从左到右的顺序填充。假设一个方法有两个参数，而且它们都有默认值，你可以用零个、一个或两个参数进行方法调用。如果你指定了一个参数，它会被赋给第一个参数，第二个参数则使用默认值。然而，无法为第二个参数指定一个值而让第一个参数使用默认值。

6.4.2 可变长度参数列表和数组
Variable-Length Argument Lists and Arrays

有时我们想编写可以接受任意数目参数的方法。要做到这点，需要在某个参数前放上一个*符。在方法的代码中，这个参数表示一个数组，它包含传给这个位置上的零个或多个参数。比如：

```
# Return the largest of the one or more arguments passed
def max(first, *rest)
  # Assume that the required first argument is the largest
  max = first
  # Now loop through each of the optional arguments looking for bigger ones
  rest.each {|x| max = x if x > max }
  # Return the largest one we found
  max
end
```

max 方法需要至少一个参数，不过它可以接受任意数目的附加参数，第一个参数被赋给 first 参数，其余的附加参数被存放在 rest 数组中。可以通过如下方式调用 max 方法：

```
max(1)        # first=1, rest=[]
max(1,2)      # first=1, rest=[2]
max(1,2,3)    # first=1, rest=[2,3]
```

注意在 Ruby 中，所有 Enumerable 对象自动具有一个 max 方法，所以这里定义的方法并非十分有用。

用*打头的参数不能超过一个。在 Ruby 1.8 中，这个参数必须出现在所有普通参数和带默认值参数的后面。如果没有参数带有&前缀（参见后面），它应该是参数列表的最后一个。在 Ruby 1.9 中，用*打头的参数仍然要放在带有默认值参数的后面，其后可以再指定普通参数，但是该普通参数仍然须要放在&打头的参数之前。

6.4.2.1 给方法传递数组参数

我们已经看到在方法定义时，如何用*使多个参数被放入到一个数组中。我们也可以用它把一个数组（或范围、枚举）参数中的每个元素分散、扩展为独立的方法参数。*符有时被称为 splat 操作符，尽管它实际上并非一个操作符。在第 4.5.5 节讨论并行赋值时我们已经看到过它的用法。

设想我们想找到数组中的最大值（而且我们不知道 Ruby 数组已经内置了 max 方法!），可以用如下方式把数组中的值传给 max 方法（先前定义的那个）：

```
data = [3, 2, 1]
m = max(*data)   # first = 3, rest=[2,1] => 3
```

如果不小心忘了用*符：

```
m = max(data)   # first = [3,2,1], rest=[] => [3,2,1]
```

在这种情况下，只把数组作为第一个变量传给 max 方法，而 max 方法不会做任何比较工作，直接返回第一个参数。

*符也可以用于那些返回数组的方法，把返回的数组展开传给另外一个方法调用。考虑前面定义的 polar 和 cartesian 方法：

```
# Convert the point (x,y) to Polar coordinates, then back to Cartesian
x,y = cartesian(*polar(x, y))
```

在 Ruby 1.9 中，枚举是可以 splat 的对象。要找到字符串中最大的字母，可以像下面这样写：

```
max(*"hello world".each_char)  # => 'w'
```

6.4.3 实参到形参的映射（译注 3）
Mapping Arguments to Parameters

当一个方法具有默认参数或以*打头的参数时，该方法被调用时的参数赋值会变得有点复杂。

在 Ruby 1.8 中，这些特殊参数的位置被限定，所以参数可以以从左到右的顺序被赋值，前面的实参被赋给普通参数；如果有剩下的实参，则被赋给那些有默认值的参数；如果还有剩下的实参，则被赋给数组参数。

在 Ruby 1.9 中需要更聪明一些，因为参数的顺序不再限制得那么严格。设想一个方法有 o 个普通参数，d 个带默认值的参数和一个*打头的数组参数，假定我们用 a 个实参对它进行调用。

如果 a 小于 o，将抛出一个 ArgumentError 异常，表明我们给出的实参个数少于最低要求。

如果 a 大于等于 o 并小于等于 o+d，那么最左边的 a - o 个带有默认值的参数会得到参数值，剩下的（右边的）o+d-a 个带默认值参数则不会得到参数值，它们将使用默认值。

如果 a 大于 o+d，那么*打头的数组参数会得到 a-o-d 个参数值；否则，它将为空。

一旦这些计算完成，实参将从左到右与参数对应，并给每个参数赋予适合的值。

6.4.4 哈希表作为有名参数
Hashes for Named Arguments

当一个方法有多于两三个的参数时，程序员很难记清楚参数的顺序。一些语言允许为参数值指定对应的参数名，Ruby 不支持这种句法，不过如果方法使用哈希对象作为参数（或参数之一），可以得到近似的功能：

```
# This method returns an array a of n numbers. For any index i, 0 <= i < n,
# the value of element a[i] is m*i+c. Arguments n, m, and c are passed
# as keys in a hash, so that it is not necessary to remember their order.
def sequence(args)
  # Extract the arguments from the hash.
  # Note the use of the || operator to specify defaults used
  # if the hash does not define a key that we are interested in.
  n = args[:n] || 0
  m = args[:m] || 1
  c = args[:c] || 0
```

译注 3：在可能产生混淆的地方，本节将区分形参（一般为 parameter）和实参（一般为 argument），在不至于混淆的地方，有时会统一翻译为参数。

```
  a = []                          # Start with an empty array
  n.times {|i| a << m*i+c }       # Calculate the value of each array element
  a                               # Return the array
end
```


可以用下面的哈希字面量方式来调用这个方法：

```
sequence({:n=>3, :m=>5})      # => [0, 5, 10]
```

为了更好地支持这种编程风格，如果哈希对象是最后一个参数（或在后面只有一个用&打头的代码块参数），Ruby 允许省略哈希字面量的大括号。没有大括号的哈希有时被称为**裸哈希**（*bare hash*），如果我们使用这种参数，看起来就像是使用有名参数，可以按照我们喜欢的顺序给定各个参数：

```
sequence(:m=>3, :n=>5)        # => [0, 3, 6, 9, 12]
```

像其他 Ruby 方法一样，可以省略圆括号：

```
# Ruby 1.9 hash syntax
sequence c:1, m:3, n:5        # => [1, 4, 7, 10, 13]
```

如果省略了圆括号，就必须也省略大括号，否则 Ruby 认为你传递了一个代码块给这个方法：

```
sequence {:m=>3, :n=>5}       # Syntax error!
```

6.4.5 代码块参数
Block Arguments

在第 5.3 节中，我们指出代码块是一段与方法调用相关联的代码，而迭代器是一个以代码块做输入的方法。每个方法调用都可以紧跟一个代码块，每个关联代码块的方法可以通过 `yield` 语句调用代码块中的代码。为了唤醒我们的记忆，下面是先前章节中使用代码块方式的 `sequence` 方法变体：

```
# Generate a sequence of n numbers m*i + c and pass them to the block
def sequence2(n, m, c)
  i = 0
  while(i < n)          # loop n times
    yield i*m + c       # pass next element of the sequence to the block
    i += 1
  end
end

# Here is how you might use this version of the method
sequence2(5, 2, 2) {|x| puts x }  # Print numbers 2, 4, 6, 8, 10
```

代码块的特性之一是匿名性，它们并非被以传统的方式传给方法：它们没有名字，通过一个关键字而非方法被调用。如果希望用更明确的方式来控制一个代码块（比如你想把这个

代码块传给其他某个方法），可以在方法最后面加上一个参数，并用&做这个参数名的前缀（注 1）。这样，这个参数就会指向传给方法的代码块（如果有的话）。这个参数的值是一个 Proc 对象，它不是通过 yield 语句被调用，而是直接通过 Proc 的 call 方法来调用：

```
def sequence3(n, m, c, &b)    # Explicit argument to get block as a Proc
  i = 0
  while(i < n)
    b.call(i*m + c)           # Invoke the Proc with its call method
    i += 1
  end
end

# Note that the block is still passed outside of the parentheses
sequence3(5, 2, 2) {|x| puts x }
```

注意，这种使用&符的方式仅仅是改变了方法的定义，方法调用的方式保持不变，只是把代码块参数放在了方法定义的圆括号之内，而代码块本身还是在方法定义的括号之外定义的。

显式传递 Proc 对象

如果创建了一个 Proc 对象（在本章稍后将看到如何做），并且希望把它显式传递给一个方法，你可以像传递其他值一样，将其传递给这个方法——Proc 对象与其他对象没什么不同，也是一个对象。在这种情形下，方法定义中不应该使用&符号：

```
# This version expects an explicitly-created Proc object, not a block
def sequence4(n, m, c, b)    # No ampersand used for argument b
  i = 0
  while(i < n)
    b.call(i*m + c)          # Proc is called explicitly
    i += 1
  end
end

p = Proc.new {|x| puts x }   # Explicitly create a Proc object
sequence4(5, 2, 2, p)        # And pass it as an ordinary argument
```

在本章的前面部分，我们两次提到特殊参数必须放在参数列表的最后，用&打头的代码块参数必须是最后一个，因为在调用方法时，代码块并非以普通的方式传递给方法，有名代码块参数与省略了括号的数组或哈希参数不同，彼此也没有什么联系。比如，下面的两个方法是合法的：

注 1：我们对以&打头的参数使用术语“block argument”而非“block parameter”，这是因为术语“block parameter”用于表示代码块本身的参数列表（比如 |x|）。

```
def sequence5(args, &b) # Pass arguments as a hash and follow with a block
  n, m, c = args[:n], args[:m], args[:c]
  i = 0
  while(i < n)
    b.call(i*m + c)
    i += 1
  end
end

# Expects one or more arguments, followed by a block
def max(first, *rest, &block)
  max = first
  rest.each {|x| max = x if x > max }
  block.call(max)
  max
end
```

这些方法可以正常工作，不过值得注意的是，可以轻松避开这种复杂性，只需要让代码块保持匿名，并用 yield 语句进行调用即可。

同样值得注意的是 yield 语句在定义了一个&参数时也可以正常工作，即使把代码块转换为一个 Proc 对象作为参数传递，它还是可以被当作一个匿名类被调用，就像代码块参数不存在一样。

6.4.5.1 在方法调用时使用&

早先我们已经看到如何在方法定义中使用*把多个参数打包到一个数组中，而且可以在方法调用时使用*来表明一个数组应该被解开，使每个元素成为一个独立的参数。&也同样可以在方法定义和方法调用时被使用。先前我们只看到如何在方法定义中使用&符号，它把一个普通的代码块定义为一个在方法中可以使用的有名 Proc 对象。在方法调用时，如果在一个 Proc 对象前使用&，在这个调用中，就会像对一个普通代码块一样处理该 Proc 对象。

考虑如下对两个数组进行求和的代码：

```
a, b = [1,2,3], [4,5]                      # Start with some data.
sum = a.inject(0) {|total,x| total+x }     # => 6. Sum elements of a.
sum = b.inject(sum) {|total,x| total+x }   # => 15. Add the elements of b in.
```

在第 5.3.2 节，我们已经介绍过 inject 迭代器。如果你忘记了，可以使用 ri Enumerable.inject 命令来查看其文档。要注意的是这段代码中的两个代码块是一样的，与其让 Ruby 解释器将同样的代码块解析两次，不如为这个代码块创建一个 Proc 对象，然后把它使用两次：

```
a, b = [1,2,3], [4,5]                      # Start with some data.
summation = Proc.new {|total,x| total+x } # A Proc object for summations.
sum = a.inject(0, &summation)              # => 6
sum = b.inject(sum, &summation)            # => 15
```

如果在方法调用中使用&，它必须被用于修饰方法的最后一个参数。代码块可以与任何方法调用关联起来，即使这个方法并不需要一个代码块，而且也没有 yield 语句。任何方法调用都可以用一个&参数作为最后一个参数。

在方法调用中，&通常出现在一个 Proc 对象之前，其实它能出现在所有支持 to_proc 方法的对象之前。Method 类（本章后面将介绍）就有这个方法，所以 Method 对象可以像 Proc 对象一样被传给迭代器。

在 Ruby 1.9 中，Symbol 类也定义了 to_proc 方法，使符号也可以用&修饰并传给迭代器。当一个符号用这种方式传递时，它被假定为一个方法名。to_proc 方法返回的 Proc 对象调用作为其第一个参数的那个有名方法，并把其余参数传给这个有名方法。一个典型的例子是：给定一个字符串数组，在一个新创建的数组中把它们都转换成大写方式。Symbol.to_proc 方法允许我们用如下优雅的方式实现它：

```
words = ['and', 'but', 'car']     # An array of words
uppercase = words.map &:upcase    # Convert to uppercase with String.upcase
upper = words.map {|w| w.upcase } # This is the equivalent code with a block
```

6.5 Proc 和 Lambda
Procs and Lambdas

代码块是 Ruby 的一种句法结构，它们不是对象，也不能像对象一样操作，不过，可以创建对象来表示一个代码块。根据对象的创建方式，它被称作一个 *proc* 或一个 *lambda*。proc 的行为与代码块类似，而 lambda 的行为则与方法类似，不过，它们都是 Proc 类的实例。

后面的小节将解释：

- 怎样以 proc 和 lambda 的形式创建 Proc 对象；
- 怎样触发 Proc 对象；
- 如何确定一个 Proc 对象需要多少参数；
- 如何判定两个 Proc 对象是否相同；
- proc 和 lambda 的区别是什么。

6.5.1 创建 proc
Creating Procs

我们已经见过创建 Proc 对象的一种方法：用&做前缀的代码块参数把代码块和方法关联起来。还可以把这个 Proc 对象作为返回值供其他方法使用：

```
# This method creates a proc from a block
def makeproc(&p)  # Convert associated block to a Proc and store in p
```

```
  p              # Return the Proc object
end
```

在定义的 makeproc 方法中，我们可以创建一个 Proc 对象：

```
adder = makeproc {|x,y| x+y }
```

现在 adder 变量指向一个 Proc 对象。用这种方式创建的 Proc 对象是 proc，而不是 lambda。所有的 Proc 对象都有一个 call 方法，当这个方法调用时，会运行创建这个 proc 时定义的代码块代码，比如：

```
sum = adder.call(2,2)  # => 4
```

除了被直接调用，Proc 对象也能被传给方法、存入数据结构或像其他任何 Ruby 对象一样操作。

除了通过方法调用创建 proc，在 Ruby 中还有三种方法可以用于创建 Proc 对象（可以是 proc 或 lambda），这些方法更常用，而且无需像先前那样定义 makeproc 方法。除了这些创建 Proc 的方法，Ruby 1.9 还支持一种新的字面量句法来定义 lambda。接下来的子小节中将讨论 Proc.new、lambda 和 proc 方法，同时解释 Ruby 1.9 中的 lambda 字面量句法。

6.5.1.1 Proc.new

在本章早先的例子中，我们已经看到 Proc.new 的使用。这是大多数类都支持的一个普通 new 方法，它是创建一个 Proc 类实例的最明显的方式。Proc.new 不带参数，它返回一个 proc（而非 lambda）方式的 Proc 对象；如果在调用 Proc.new 时关联一个代码块，它返回的对象将代表那个代码块。例如：

```
p = Proc.new {|x,y| x+y }
```

如果调用 Proc.new 时没有关联一个代码块，它返回的 proc 会指向所在方法关联的代码块。这种方式可以作为&做前缀的代码块参数的一种替代方式。比如，下面的两个方法是等价的：

```
def invoke(&b)     def invoke
  b.call             Proc.new.call
end              end
```

6.5.1.2 Kernel.lambda

另外一种创建 Proc 对象的方式是使用 lambda 方法，它是 Kernel 模块的一个方法，所以看起来像是一个全局函数。就像名字所暗示的，lambda 方法返回的 Proc 对象是一个 lambda 而非 proc。lambda 方法不带参数，但是在调用时必须关联一个代码块：

```
is_positive = lambda {|x| x > 0 }
```

Lambda 的历史

Lambda 和 lambda 方法的名字来自 *lambda* 演算，它是数理逻辑的一个分支，被应用到函数式编程语言中。Lisp 也使用术语“lambda”来表示可以被当作对象操作的函数。

6.5.1.3 Kernel.proc

在 Ruby 1.8 中，全局的 `proc` 方法是 `lambda` 的同义词。尽管它叫 proc，但其实返回的是一个 lambda 而非 proc。Ruby 1.9 修正了这个问题，在这个版本中，`proc` 是 `Proc.new` 的同义词。

因为这种二义性，我们不应该在 Ruby 1.8 的代码中使用 `proc`，否则代码行为可能在升级解释器时发生变化。如果使用的是 Ruby 1.9 而且确信它们不会在 Ruby 1.8 解释器下执行，可以把 `proc` 作为 `Proc.new` 的快捷方式使用。

6.5.1.4 Lambda 字面量

Ruby 1.9 支持一种全新的句法，把 lambda 定义为字面量，我们先看看 Ruby 1.8 是如何用 `lambda` 方法创建 lambda 的：

```
succ = lambda {|x| x+1}
```

在 Ruby 1.9 中，我们可以用如下方式把它转换成字面量：

- 把 `lambda` 方法名替换为->符号。
- 把大括号内的参数列表移到大括号之前。
- 把参数列表的分隔符从`||`变为`()`。

做了这些改变后，我们就得到一个 Ruby 1.9 的 lambda 字面量：

```
succ = ->(x){ x+1 }
```

succ 现在容纳了一个 `Proc` 对象，我们可以像其他方式一样使用它：

```
succ.call(2)    # => 3
```

把这个句法引入 Ruby 也是充满争议的，人们需要一些时间去适应它。注意这个箭头符号与在哈希字面量中的箭头符号是不同的，lambda 字面量使用它作为一个连接符，而在哈希字面量中，它被作为一个等号来使用。

跟 Ruby 1.9 中的代码块一样，lambda 字面量的参数列表可以包括一个代码块内局部变量的声明，这样可以防止覆盖包含代码块的代码中的同名变量。在参数列表后，放在一个分号后面的就是局部变量列表：

```
# This lambda takes 2 args and declares 3 local vars
f = ->(x,y; i,j,k) { ... }
```

从这个新的lambda句法中，我们可以得到的好处之一是现在lambda可以为参数设定默认值，就像方法能做的那样：

```
zoom = ->(x,y,factor=2) { [x*factor, y*factor] }
```


就像方法声明一样，lambda 字面量定义的圆括号是可选的，因为参数列表、局部变量列表完全用->，；和{分隔开，我们可以用如下方式重写上面定义的三个 lambda：

```
succ = ->x { x+1 }
f = -> x,y; i,j,k { ... }
zoom = ->x,y,factor=2 { [x*factor, y*factor] }
```

当然，lambda 的参数和局部变量都是可选的，lambda 字面量对它们可以完全省略。最小的 lambda 如下，它不带任何参数并返回 `nil`：

```
->{}
```

这种新句法的好处之一是简洁性。在需要把一个lambda作为方法或另一个lambda的参数时，它对你会很有帮助：

```
def compose(f,g)          # Compose 2 lambdas
  ->(x) { f.call(g.call(x)) }
end
succOfSquare = compose(->x{x+1}, ->x{x*x})
succOfSquare.call(4)      # => 17: Computes (4*4)+1
```

Lambda 字面量产生的 `Proc` 对象跟代码块不同，如果想把 lambda 传给一个期望获得代码块的方法，可以像对待其他 `Proc` 对象一样，用&修饰这个字面量。下面演示如何用代码块和 lambda 字面量将一个数字数组以降序排列：

```
data.sort {|a,b| b-a }   # The block version
data.sort &->(a,b){ b-a } # The lambda literal version
```

在这种情况下，如你所见，普通的代码块语法更简单。

6.5.2 调用 Proc 和 Lambda
Invoking Procs and Lambdas

Proc 和 lambda 是对象而非方法，它们不能像方法一样被调用。如果 `p` 表示一个 `Proc` 对象，不能像方法一样调用 `p`。但是因为 `p` 是一个对象，因此可以调用 `p` 的方法。我们已经提到过 `Proc` 类定义了一个 `call` 方法，调用这个方法会执行原始代码块的代码。传给 `call` 方法的参数成为代码块的参数，而代码块的返回值将作为 `call` 方法的返回值：

```
f = Proc.new {|x,y| 1.0/(1.0/x + 1.0/y) }
z = f.call(x,y)
```

Proc 类还定义了数组访问操作符，它的工作方式与 call 一样，这意味着可以用与方法调用类似的方式调用 proc 和 lambda，只是圆括号被换成了方括号。比如，上面的 proc 调用可以替换成如下方式：

```
z = f[x,y]
```

Ruby 1.9 还提供了一种方式来调用 Proc 对象，可以用圆括号前面加上一个句点符号来替代方括号：

```
z = f.(x,y)
```

.()看起来像一个缺少方法名的方法调用，这并非一个可以定义的操作符，它只是一个调用 call 方法的语法糖，可以用于任何定义了 call 方法的对象，而不限于 Proc 对象。

6.5.3 Proc 对象的元数
The Arity of a Proc

Proc 或 lambda 的元数（*arity*）是指它希望的参数个数。(arity 这个词源自一元（unary）、二元（binary）和三元（ternary）等单词的 ary 词根。) Proc 对象有一个 arity 方法，用于返回期望的参数个数。例如：

```
lambda{||}.arity          # => 0. No arguments expected
lambda{|x| x}.arity       # => 1. One argument expected
lambda{|x,y| x+y}.arity   # => 2. Two arguments expected
```

当 Proc 对象拥有一个*打头的参数时，它可以接受任意数目的参数，这时 arity 方法就会变得有点让人费解。当 Proc 对象带有可选参数时，arity 方法返回一个值为-n-1 的负数，其中 n 表示该对象必须有 n 个参数，负数值表明它也可以接受额外的可选参数。-n-1 是 n 的互补数，可以通过~操作符互相转换。因此如果 arity 返回一个负数 m，那么~m(或者-m-1)就是必需参数的个数：

```
lambda {|*args|}.arity        # => -1.  ~-1 = -(-1)-1 = 0 arguments required
lambda {|first, *rest|}.arity # => -2.  ~-2 = -(-2)-1 = 1 argument required
```

对于 arity 方法，还有最后一点需要提及。在 Ruby 1.8 中，不带任何参数（也就是说，不带|符号）定义的 Proc 对象可以接受任意个数的参数（这些参数会被忽略），arity 方法返回-1，表明没有任何必需参数。而在 Ruby 1.9 中，对这种行为进行了修改，这种方式定义的 Proc 对象的元数为 0。该对象是一个 lambda，在调用时给定任何参数都会报错：

```
puts lambda {}.arity  # -1 in Ruby 1.8; 0 in Ruby 1.9
```

6.5.4 Proc 相等性
Proc Equality

Proc 类定义了==方法来判定两个 Proc 对象是否相等。不过，要记住两个 proc 或 lambda 拥有同样的代码并不意味着它们是相等的：

```
lambda {|x| x*x } == lambda {|x| x*x }  # => false
```

只有当一个 Proc 对象是另一个 Proc 对象的克隆（clone）或复制品（duplicate）时，==方法才会返回 true：

```
p = lambda {|x| x*x }
q = p.dup
p == q                          # => true: the two procs are equal
p.object_id == q.object_id      # => false: they are not the same object
```

6.5.5 Lambda 和 Proc 的区别在哪儿
How Lambdas Differ from Procs

proc 是代码块的对象形式，它的行为就像一个代码块。Lambda 的行为略有不同，它的行为更像方法而非代码块。调用一个 proc 则像对代码块进行 yield，而调用一个 lambda 则像调用一个方法。在 Ruby 1.9 中，可以通过 Proc 对象的实例方法 lambda?来判定该实例是一个 proc 还是 lambda，如果返回值为真，那么它是一个 lambda，否则为一个 proc。接下来的几个小节将详细解释 proc 和 lambda 的区别。

6.5.5.1 代码块、procs 和 lambda 中的 return 语句

在第 5 章中我们讲过，return 语句使代码从所包含的方法中返回，即使它被包含在代码块代码中也是如此。在一个代码块中的 return 语句不仅仅会从调用代码块的迭代器返回，它还会从调用迭代器的方法返回。例如：

```
def test
  puts "entering method"
  1.times { puts "entering block"; return }  # Makes test method return
  puts "exiting method"  # This line is never executed
end
test
```

proc 与代码块类似，因此如果调用的 proc 执行一个 return 语句，它会试图从代码块（这个代码块被转换为一个 proc）所在的方法中返回。比如：

```
def test
  puts "entering method"
  p = Proc.new { puts "entering proc"; return }
  p.call                  # Invoking the proc makes method return
  puts "exiting method"   # This line is never executed
end
test
```

不过，因为 proc 经常在不同方法间传递，在 proc 中使用 return 语句会十分诡异。在 proc 被调用时，在句法上包含该 proc 的方法已经返回：

```
def procBuilder(message)              # Create and return a proc
  Proc.new { puts message; return }   # return returns from procBuilder
  # but procBuilder has already returned here!
end

def test
  puts "entering method"
  p = procBuilder("entering proc")
  p.call                    # Prints "entering proc" and raises LocalJumpError!
  puts "exiting method"     # This line is never executed
end
test
```

在把代码块转换成一个对象后，可以四处传递该对象，并且在“上下文”之外使用它。如果这样做，则要承担从一个已经返回的方法中返回的风险，就像本例所示的那样。当这种情况发生时，Ruby 会抛出一个 LocalJumpError 异常。

当然，在这个臆造的例子中，可以通过去掉多余的 return 语句来修复这个问题。不过 return 语句并非总是多余的，这时可以通过使用 lambda 而非 proc 来修复这个问题。如前所述，lambda 更像方法而非代码块。这样，在 lambda 中的 return 语句仅仅从 lambda 自身返回，而不会从产生 lambda 的方法中返回：

```
def test
  puts "entering method"
  p = lambda { puts "entering lambda"; return }
  p.call                # Invoking the lambda does not make the method return
  puts "exiting method"  # This line *is* executed now
end
test
```

Lambda 中的 return 仅仅从 lambda 自身返回，这个事实意味着我们根本无须考虑 LocalJumpError：

```
def lambdaBuilder(message)        # Create and return a lambda
  lambda { puts message; return } # return returns from the lambda
end

def test
  puts "entering method"
  l = lambdaBuilder("entering lambda")
  l.call                    # Prints "entering lambda"
  puts "exiting method"     # This line is executed
end
test
```

6.5.5.2 代码块、procs 和 lambda 中的 break 语句

图 5-3 显示了在代码块中 break 语句的行为：它使该代码块返回到它的迭代器，然后该迭代器再返回到调用它的方法。因为 proc 与代码块相似，我们可以设想 proc 中的 break 语句有同样的行为。不过这并不容易被验证。当我们用 Proc.new 创建一个 proc 时，这个 Proc.new 就是 break 语句应该返回的地方，当我们调用 proc 对象时，这个迭代器已经返回了，因此，用 Proc.new 创建的 proc 有一个顶级 break 语句（译注 4）是说不通的：

```
def test
  puts "entering test method"
  proc = Proc.new { puts "entering proc"; break }
  proc.call              # LocalJumpError: iterator has already returned
  puts "exiting test method"
end
test
```

如果通过迭代器方法的&参数方式来创建一个 proc，我们可以调用它让该迭代器方法返回：

```
def iterator(&proc)
  puts "entering iterator"
  proc.call  # invoke the proc
  puts "exiting iterator"   # Never executed if the proc breaks
end

def test
  iterator { puts "entering proc"; break }
end
test
```

Lambda 类似于方法，因此如果把一个 break 语句孤零零地放在那里，而不是出现在循环或迭代方法中，是说不通的！下面的语句，没有任何东西可以被 break，你可能认为它会失败。不过，在这种情况下，break 的行为与 return 一样：

```
def test
  puts "entering test method"
  lambda = lambda { puts "entering lambda"; break; puts "exiting lambda" }
  lambda.call
  puts "exiting test method"
end
test
```

6.5.5.3 其他的控制流语句

顶级的 next 语句在代码块、proc 和 lambda 中有相同的行为：它使调用代码块、proc 或 lambda 的 yield 语句或 call 方法返回。如果 next 语句后面跟一个表达式，那么这个表达式会成为该代码块、proc 或 lambda 的**返回值**。

译注 4：即不出现在任何循环或迭代器中的 break 语句。

redo 语句在 proc 和 lambda 中也有相同的行为：它让控制流转向该 proc 或 lambda 的开始处。

在 proc 或 lambda 中，retry 是被禁止使用的：使用它永远会导致一个 LocalJumpError 异常。

raise 语句在代码块、proc 和 lambda 中有相同的行为：异常总是向调用堆栈的上层传播。如果 raise 异常的代码块、proc 或 lambda 中没有 rescue 语句，该异常会首先传播到 yield 该代码块的方法，或者用 call 调用该 proc 或 lambda 的方法。

6.5.5.4 传给 proc 和 lambda 的参数

用 yield 调用一个代码块与调用一个方法类似，但并不相同。在使用参数的方式上，两者的处理并不相同。yield 和 invocation 的语义不同，yield 语义更类似于 5.4.4 节描述的并行赋值。你可能已经猜到了，调用 proc 使用的是 yield 语义，而调用 lambda 使用的是 invocation 语义：

```
p = Proc.new {|x,y| print x,y }
p.call(1)       # x,y=1:    nil used for missing rvalue: Prints 1nil
p.call(1,2)     # x,y=1,2:   2 lvalues, 2 rvalues:       Prints 12
p.call(1,2,3)   # x,y=1,2,3: extra rvalue discarded:     Prints 12
p.call([1,2])   # x,y=[1,2]: array automatically unpacked: Prints 12
```

上面的代码显示了 proc 对参数处理的灵活性：可以安静地抛弃多余参数，将 nil 赋给遗漏的参数，甚至可以拆开数组。（如果 proc 需要的是单个参数，而给出的是多个参数时，它可以把这些参数打包成一个数组传给这个 proc。这个特性在这里没有演示。）

lambda 在这方面就没有那么灵活了，跟方法一样，我们必须用与声明时同样多的参数对它进行调用：

```
l = lambda {|x,y| print x,y }
l.call(1,2)       # This works
l.call(1)         # Wrong number of arguments
l.call(1,2,3)     # Wrong number of arguments
l.call([1,2])     # Wrong number of arguments
l.call(*[1,2])    # Works: explicit splat to unpack the array
```

6.6 闭包
Closures

在 Ruby 中，proc 和 lambda 都是**闭包**（*closure*）。“闭包”这个术语源自初期的计算机科学，它表示一个对象既是一个可调用的函数，同时也是绑定在这个函数上的一个变量。当创建一个 proc 或 lambda 时，得到的 Proc 对象不仅包含了可执行的代码块，也绑定了代码块中所使用的全部变量。

我们已经知道代码块可以使用在其外定义的局部变量和方法参数。比如，在下面的代码中，collect 迭代器关联的代码块使用了方法参数 n：


```
# multiply each element of the data array by n
def multiply(data, n)
  data.collect {|x| x*n }
end

puts multiply([1,2,3], 2)   # Prints 2,4,6
```

更有意思的是（或可能更让人惊奇），如果代码块变成一个 proc 或 lambda，它还可以在方法已经返回后继续访问 n。下面的代码演示了这一点：

```
# Return a lambda that retains or "closes over" the argument n
def multiplier(n)
  lambda {|data| data.collect{|x| x*n } }
end
doubler = multiplier(2)       # Get a lambda that knows how to double
puts doubler.call([1,2,3])    # Prints 2,4,6
```

multiplier 方法返回一个 lambda，这个 lambda 在定义它的范围外被使用，我们称之为一个闭包，这个闭包封装（或保持）了所绑定的方法参数 n。

6.6.1 闭包和共享变量
Closures and Shared Variables

需要注意的是闭包并不持有它引用的变量，而确实持有所使用的变量，并延长了其生命周期，或者说 lambda 或 proc 在创建时并不静态绑定使用的变量，相反，其绑定是动态的，在 lambda 或 proc 运行时才去查找变量的值。

作为例子，下面的代码定义了一个返回两个 lambda 的方法，因为它们在同样的范围内被定义，因此它们都可以访问这个范围内的变量。当一个 lambda 改变了某个变量值时，这个改变对另外一个 lambda 也是可见的：

```
# Return a pair of lambdas that share access to a local variable.
def accessor_pair(initialValue=nil)
  value = initialValue  # A local variable shared by the returned lambdas.
  getter = lambda { value }           # Return value of local variable.
  setter = lambda {|x| value = x }    # Change value of local variable.
  return getter,setter                # Return pair of lambdas to caller.
end

getX, setX = accessor_pair(0) # Create accessor lambdas for initial value 0.
puts getX[]       # Prints 0. Note square brackets instead of call.
setX[10]          # Change the value through one closure.
puts getX[]       # Prints 10. The change is visible through the other.
```

在同样范围内创建的 lambda 会共享变量，这可能成为一个有用的特性，也可能成为 bug 之源。在任何时候，如果一个方法创建了多个闭包，我们就需要特别小心变量的使用。考虑如下代码：

```
# Return an array of lambdas that multiply by the arguments
def multipliers(*args)
  x = nil
  args.map {|x| lambda {|y| x*y }}
end

double,triple = multipliers(2,3)
puts double.call(2)    # Prints 6 in Ruby 1.8
```

这个 `multipliers` 方法使用 `map` 迭代器和一个代码块返回一组 lambda（在代码块中创建）。在 Ruby 1.8 中，代码块参数并不总是限定在代码块内访问（参见 5.4.3 节），因此创建的所有 lambda 都能访问 `multipliers` 方法的局部变量 `x`。如上所述，闭包并不保持变量的当前值，而是保持变量本身，这样每个 lambda 都可共享变量 `x`。这个变量只有一个值，而每个创建的 lambda 都使用同样的值，这就是为什么我们把这个 lambda 命名为 `double`（加倍），而返回的值却是三倍而非两倍的原因。

对于这段特殊的代码，Ruby 1.9 不会存在这个问题。因为在 Ruby 1.9 中，代码块参数仅在代码块内部可用。不过，如果在循环中创建 lambda，并在 lambda 中使用了一个循环变量（比如数组索引），还是会遇到这样的问题。

6.6.2 闭包和绑定
Closures and Bindings

`Proc` 定义了一个名为 `binding` 的方法，调用这个方法会返回一个 `Binding` 对象，它表示该闭包所使用的绑定。

关于绑定的更多内容

目前为止，我们讨论的闭包绑定就好像是变量名和变量值的映射。其实，绑定并不限于变量，它包含所有与执行方法相关的信息，比如 self 的值及将被 yield 调用的代码块（如果有的话）。

`Binding` 对象自身并没有什么值得注意的方法，不过它可以作为全局函数 `eval` 的第二个参数（参见第 8.2 节），它能为 eval 函数执行给定代码时提供一个上下文环境。在 Ruby 1.9 中，`Binding` 也自带了 `eval` 方法，你可能更愿意直接使用它。（使用 `ri` 可以获得更多关于 `Kernel.eval` 和 `Binding.eval` 的信息。）

使用 `Binding` 对象和 `eval` 方法可以让我们获得一个操控闭包行为的后门，让我们再看看先前的例子：

```
# Return a lambda that retains or "closes over" the argument n
def multiplier(n)
  lambda {|data| data.collect{|x| x*n } }
end
doubler = multiplier(2)       # Get a lambda that knows how to double
puts doubler.call([1,2,3])    # Prints 2,4,6
```

假定我们要改变 doubler 的行为：

```
eval("n=3", doubler.binding)  # Or doubler.binding.eval("n=3") in Ruby 1.9
puts doubler.call([1,2,3])    # Now this prints 3,6,9!
```

eval 方法还提供了一种快捷方式，可以直接把 Proc 对象传给它，而无须传递 Proc 的 Binding 对象，因此可以用如下方式来调用 eval 方法：

```
eval("n=3", doubler)
```

绑定并不是闭包的专有特性。Kernel.binding 方法也可返回一个 Binding 对象，它表示调用此方法时的那些有效绑定。

6.7 Method 对象
Method Objects

Ruby 的方法和代码块都是可执行的语言构件，但它们不是对象。proc 和 lambda 是代码块的对象版本，它们可以被执行，也可以像数据一样操作。Ruby 元编程（或者说**反射**）能力相当强大，方法也可以用 Method 类的实例来表示（元编程在第 8 章中介绍，但是 Method 对象在这里介绍）。需要注意的是，通过 Method 对象调用方法不如直接调用高效。通常，Method 对象并不像 lambda 或 proc 那样频繁使用。

Object 类定义了一个名为 method 的方法，它接受一个用字符串或符号表示的方法名，返回的 Method 对象表示在接收者对象中相应的方法（如果没有这个方法，会抛出一个 NameError 异常）。例如：

```
m = 0.method(:succ)  # A Method representing the succ method of Fixnum 0
```

在 Ruby 1.9 中，还可以通过 public_method 方法获得一个 Method 对象，它与 method 方法类似，但是忽略被保护（protected）方法和私有（private）方法。

Method 类并非 Proc 类的子类，但是行为上与后者很相似。跟 Proc 对象一样，Method 对象也用 call 方法（或[]操作符）调用；而且，Method 类也定义了 arity 方法，它与 Proc 类的 arity 方法相似。要调用 Method 对象 m，可以用如下方式：

```
puts m.call    # Same as puts 0.succ. Or use puts m[].
```

通过 Method 对象调用方法并不改变方法调用的语义，也不改变控制流语句（比如 return

和 break）的语义。Method 对象的 call 方法使用的是方法调用（method-invocation）语义，而非 yield 语义，因此，Method 对象更像 lambda 而非 proc。

Method 对象与 Proc 对象的工作方式非常相似，因此常常被用来代替 Proc 对象。在须要使用 Proc 的地方，可以用 Method.to_proc 方法把 Method 对象转换为 Proc 对象。这就是为什么 Method 对象可以用一个&做前缀，然后用这个对象取代代码块作为参数传递的原因。例如：

```
def square(x); x*x; end
puts (1..10).map(&method(:square))
```

用 Proc 定义 Method

除了可以把一个 Method 对象转换为一个 Proc 对象，还可以反过来把一个 Proc 对象转换为一个 Method 对象。（Module 的）define_method 方法要求一个 Symbol 对象作为参数，它把这个符号作为方法名，以关联的代码块为方法的主体来创建一个方法。如果不用代码块，也可以以一个 Proc 对象或 Method 对象为方法的第二个参数。

Method 对象和 Proc 对象的一个重要区别是 Method 对象不是闭包。Ruby 的方法是完全自包含的，它们从不会试图访问其范围外的任何局部变量，因此，唯一一个 Method 对象绑定的值是 self——该方法的宿主对象。

在 Ruby 1.9 中，Method 类定义了三个新方法（它们不能在 Ruby 1.8 中使用）：name 方法返回方法的名字（用字符串方式）；owner 方法返回这个方法被定义的类；receiver 方法返回该方法绑定的对象。对于任何 Method 对象 m，m.receiver.class 必须是 m.owner 或其子类。

6.7.1 无绑定的 Method 对象
Unbound Method Objects

除了 Method 类，Ruby 还定义了一个 UnboundMethod 类。就像其名字所暗示的，UnboundMethod 对象代表一个没有绑定对象的方法。因为 UnboundMethod 对象没有绑定对象，它不能被调用，因此也没有定义 call 或[]方法。

要得到一个 UnboundMethod 对象，我们可以对任意类或模代码块使用 instance_method 方法：

```
unbound_plus = Fixnum.instance_method("+")
```

在 Ruby 1.9 中，我们也可以通过 public_instance_method 方法来获得一个 UnboundMethod 对象。它的工作方式与 instance_method 方法类似，但是忽略被保护（protected）方法和私有（private）方法。

为了调用一个无绑定的方法，我们必须首先使用 bind 方法绑定一个对象：

```
plus_2 = unbound_plus.bind(2)   # Bind the method to the object 2
```

bind 方法返回一个 Method 对象，它可以用 call 方法调用：

```
sum = plus_2.call(2)    # => 4
```

获得 UnboundMethod 对象的另外一个方法是用 Method 类的 unbind 方法：

```
plus_3 = plus_2.unbind.bind(3)
```

在 Ruby 1.9 中，跟 Method 类一样，UnboundMethod 也有类似的 name 和 owner 方法。

6.8 函数式编程
Functional Programming

与 Lisp 和 Haskell 不同，Ruby 并非一种函数式编程语言，不过 Ruby 提供了代码块、proc 和 lambda，它们可以很好地用于函数式编程风格。当使用代码块和 Enumerable 的迭代器（比如 map 或 inject）时，你使用的是函数式编程风格。下面是使用 map 和 inject 迭代器的例子：

```
# Compute the average and standard deviation of an array of numbers
mean = a.inject {|x,y| x+y } / a.size
sumOfSquares = a.map{|x| (x-mean)**2 }.inject{|x,y| x+y }
standardDeviation = Math.sqrt(sumOfSquares/(a.size-1))
```

如果你对函数式编程着迷，可以很容易地为 Ruby 的内置类增加特性来得到函数式编程。本章的其余部分将探讨用函数编程的各种可能性。这部分的代码比较多，它们主要作为扩展思路的探索，而非说明它们比普通的编程方式更优越。尤其值得一提的是，这些代码会频繁重定义操作符，这使得代码难以理解和维护。

这部分的内容比较高级，我们假定你对第 7 章的内容有所了解。因此，如果是第一次阅读本书，你可能会希望先跳过这部分内容。

6.8.1 对一个 Enumerable 对象应用一个函数
Applying a Function to an Enumerable

map 和 inject 是 Enumerable 类定义的两个最重要的迭代器，它们都需要一个代码块。如果用以函数为中心的方式编写程序，我们会喜欢那些可以让函数应用到 Enumerable 对象上的方法：

```
# This module defines methods and operators for functional programming.
module Functional

  # Apply this function to each element of the specified Enumerable,
  # returning an array of results. This is the reverse of Enumerable.map.
```

```
  # Use | as an operator alias. Read "|" as "over" or "applied over".
  #
  # Example:
  #   a = [[1,2],[3,4]]
  #   sum = lambda {|x,y| x+y}
  #   sums = sum|a   # => [3,7]
  def apply(enum)
    enum.map &self
  end
  alias | apply

  # Use this function to "reduce" an enumerable to a single quantity.
  # This is the inverse of Enumerable.inject.
  # Use <= as an operator alias.
  # Mnemonic: <= looks like a needle for injections
  # Example:
  #   data = [1,2,3,4]
  #   sum = lambda {|x,y| x+y}
  #   total = sum<=data   # => 10
  def reduce(enum)
    enum.inject &self
  end
  alias <= reduce
end

# Add these functional programming methods to Proc and Method classes.
class Proc; include Functional; end
class Method; include Functional; end
```

注意，我们是在 Functional 模块中定义方法，然后把这个模块包含（include）在 Proc 和 Method 类中，这样，apply 和 reduce 方法就可以用在 proc 和 lambda 对象上。后面的绝大多数方法仍然定义在 Functional 模块中，因此它们也可以被 Proc 和 Method 对象使用。

有了上面定义的 apply 和 reduce 方法，我们可以重构下面的统计方法：

```
sum = lambda {|x,y| x+y }          # A function to add two numbers
mean = (sum<=a)/a.size             # Or sum.reduce(a) or a.inject(&sum)
deviation = lambda {|x| x-mean }   # Function to compute difference from mean
square = lambda {|x| x*x }         # Function to square a number
standardDeviation = Math.sqrt((sum<=square|(deviation|a))/(a.size-1))
```

值得注意的是，最后一行尽管很简洁，但是那些非标准的操作符使得它难以阅读。还要注意|符是我们自己定义的，它是左连接的，因此，上面的代码要在一个 Enumerable 对象上应用多个函数，则须要使用圆括号，也就是说，必须使用 square|(deviation|a)，而不能使用 square|deviation|a。

6.8.2 复合函数
Composing Functions

如果有两个函数 f、g，有时我们会希望定义一个新函数 h= f(g())，它也可被称为 f 由 g 复合。我们可以用一个方法自动进行函数复合，其代码如下：

```
module Functional
  # Return a new lambda that computes self[f[args]].
  # Use * as an operator alias for compose.
  # Examples, using the * alias for this method.
  #
  # f = lambda {|x| x*x }
  # g = lambda {|x| x+1 }
  # (f*g)[2]   # => 9
  # (g*f)[2]   # => 5
  #
  # def polar(x,y)
  #   [Math.hypot(y,x), Math.atan2(y,x)]
  # end
  # def cartesian(magnitude, angle)
  #   [magnitude*Math.cos(angle), magnitude*Math.sin(angle)]
  # end
  # p,c = method :polar, method :cartesian
  # (c*p)[3,4]  # => [3,4]
  #
  def compose(f)
    if self.respond_to?(:arity) && self.arity == 1
      lambda {|*args| self[f[*args]] }
    else
      lambda {|*args| self[*f[*args]] }
    end
  end

  # * is the natural operator for function composition.
  alias * compose
end
```

示例中的注释演示了如何对 Method 对象和 lambda 使用 compose 方法。我们可以用这个新的*函数复合操作符来简化标准差的计算。仍然使用上面定义的 sum、square 和 deviation，标准差的计算变为：

```
standardDeviation = Math.sqrt((sum<=square*deviation|a)/(a.size-1))
```

区别在于在 square 和 deviation 应用到数组 a 之前，我们先将它们复合成单个函数。

6.8.3 局部应用函数
Partially Applying Functions

在函数式编程中，**局部应用**是指用一个函数和部分参数值产生一个新的函数，这个函数等价于用某些固定参数调用原有的函数。例如：

```
product = lambda {|x, y| x*y }        # A function of two arguments
double = lambda {|x| product(2,x) }   # Apply one argument
```

局部应用可以用 Functional 模块中的方法（和操作符）进行简化：

```
module Functional
  #
  # Return a lambda equivalent to this one with one or more initial
  # arguments applied. When only a single argument
  # is being specified, the >> alias may be simpler to use.
  # Example:
  #   product = lambda {|x,y| x*y}
  #   doubler = lambda >> 2
  #
  def apply_head(*first)
    lambda {|*rest| self[*first.concat(rest)]}
  end

  #
  # Return a lambda equivalent to this one with one or more final arguments
  # applied. When only a single argument is being specified,
  # the << alias may be simpler.
  # Example:
  #  difference = lambda {|x,y| x-y }
  #  decrement = difference << 1
  #
  def apply_tail(*last)
    lambda {|*rest| self[*rest.concat(last)]}
  end

  # Here are operator alternatives for these methods. The angle brackets
  # point to the side on which the argument is shifted in.
  alias >> apply_head    # g = f >> 2 -- set first arg to 2
  alias << apply_tail    # g = f << 2 -- set last arg to 2
end
```

使用这些方法和操作符，可以简单地用 product>>2 来定义我们的 double 函数。使用局部应用，我们使标准差计算变得更加抽象，这可以通过一个更通用的 difference 函数来实现：

```
difference = lambda {|x,y| x-y }  # Compute difference of two numbers
deviation = difference<<mean      # Apply second argument
```

6.8.4 缓存（Memoizing）函数
Memoizing Functions

Memoization 是函数式编程的一个术语，表示缓存函数调用的结果。如果一个函数对同样的参数输入总是返回相同的结果，另外出于某种需要我们认为这些参数会不断使用，而且执行这个函数比较消耗资源，那么 memoization 可能是一个有用的优化。我们可以通过下面的方法使 Proc 和 Method 对象的 memoization 自动化：

```
module Functional
  #
  # Return a new lambda that caches the results of this function and
  # only calls the function when new arguments are supplied.
  #
  def memoize
    cache = {}  # An empty cache. The lambda captures this in its closure.
    lambda {|*args|
      # notice that the hash key is the entire array of arguments!
      unless cache.has_key?(args)  # If no cached result for these args
        cache[args] = self[*args]  # Compute and cache the result
      end
      cache[args]                  # Return result from cache
    }
  end
  # A (probably unnecessary) unary + operator for memoization
  # Mnemonic: the + operator means "improved"
  alias +@ memoize       # cached_f = +f
end
```

下面是如何使用 memoize 方法或一元+操作符的例子：

```
# A memoized recursive factorial function
factorial = lambda {|x| return 1 if x==0; x*factorial[x-1]; }.memoize
# Or, using the unary operator syntax
factorial = +lambda {|x| return 1 if x==0; x*factorial[x-1]; }
```

注意这里的 factorial 方法是一个递归函数，它自己也会调用缓存版本的自身函数，得到最佳的缓存效果。否则，如果定义一个非缓存版本的递归函数，然后根据它定义一个缓存版本方法，优化效果就没有那么突出了：

```
factorial = lambda {|x| return 1 if x==0; x*factorial[x-1]; }
cached_factorial = +factorial # Recursive calls aren't cached!
```

6.8.5 符号、方法和 Proc
Symbols, Methods, and Procs

Symbol、Method 和 Proc 类有亲密的关系。我们已经看到 method 方法可以用一个 Symbol 对象作为参数，然后返回一个 Method 方法。

Ruby 1.9 为 Symbol 类增加了一个有用的 to_proc 方法，这个方法允许用&打头的符号作为代码块被传入到一个迭代器中。在这里，这个符号被假定为一个方法名。在用 to_proc 方法创建的 Proc 对象被调用时，它会调用第一个参数所表示的名字的方法，剩下的参数则作为参数传递给这个方法。示例如下：

```
# Increment an array of integers with the Fixnum.succ method
[1,2,3].map(&:succ)  # => [2,3,4]
```

如果不用 Symbol.to_proc 方法，我们会不得不更啰嗦一些：

```
[1,2,3].map {|n| n.succ }
```

Symbol.to_proc 最初是作为 Ruby 1.8 的扩展被引入的，它通常使用如下方式实现：

```
class Symbol
  def to_proc
    lambda {|receiver, *args| receiver.send(self, *args)}
  end
end
```

这个实现使用 send 方法（参见第 8.4.3 节）来调用一个符号命名的方法。不过，也可以像下面这样做：

```
class Symbol
  def to_proc
    lambda {|receiver, *args| receiver.method(self)[*args]}
  end
end
```

除了 to_proc，还能定义一些相关而且有用的工具方法，首先从 Module 类开始：

```
class Module
  # Access instance methods with array notation. Returns UnboundMethod,
  alias [] instance_method
end
```

这里我们为 Module 类的 instance_method 定义了一个快捷方式。注意，这个方法会返回一个 UnboundMethod 对象，它在绑定到一个对象前不能被调用，下面是一个使用这种新记号的例子（注意，我们可以用名字像索引一样获得一个方法！）：

```
String[:reverse].bind("hello").call   # => "olleh"
```

用同样的语法糖，绑定一个无绑定的方法也可以变得简单：

```
class UnboundMethod
  # Allow [] as an alternative to bind.
  alias [] bind
end
```

使用这里定义的别名，以及已有的用来调用方法的[]别名，上面的代码会变成：

```
String[:reverse]["hello"][]   # => "olleh"
```

第一个方括号索引一个方法，第二个方括号将它绑定，而第三个方括号则调用它。

接下来，既然我们用[]操作符来索引一个类的方法，那么就用[]=来定义一个实例方法：

```
class Module
  # Define a instance method with name sym and body f.
  # Example: String[:backwards] = lambda { reverse }
  def []=(sym, f)
```

```
    self.instance_eval { define_method(sym, f) }
  end
end
```

[]=操作符的定义有点让人费解——这是 Ruby 的高级内容。define_method 是 Module 的私有方法，我们用 instance_eval 方法（Object 的一个公开方法）运行一个代码块（包括一个私有方法的调用），就像它位于方法定义的模块中一样。在第 8 章中我们将再次看到 instance_eval 和 define_method 方法。

让我们用新的[]=操作符定义一个新的 Enumerable.average 方法：

```
Enumerable[:average] = lambda do
  sum, n = 0.0, 0
  self.each {|x| sum += x; n += 1 }
  if n == 0
    nil
  else
    sum/n
  end
end
```

现在我们有了[]和[]=操作符，它们可以用于获得和设置一个类或模块的成员方法。我们也可以对类的单键方法做相似的事情（也包括类或模块的类方法）。任何对象都可以有单键方法，不过给 Object 类定义一个[]操作符有点说不通，因为已经有太多的子类已经定义了这个操作符，因此，我们可以用另外一种方式，给 Symbol 类定义这样的操作符：

```
#
# Add [] and []= operators to the Symbol class for accessing and setting
# singleton methods of objects. Read : as "method" and [] as "of".
# So :m[o] reads "method m of o".
#
class Symbol
  # Return the Method of obj named by this symbol. This may be a singleton
  # method of obj (such as a class method) or an instance method defined
  # by obj.class or inherited from a superclass.
  # Examples:
  #   creator = :new[Object]   # Class method Object.new
  #   doubler = :*[2]          # * method of Fixnum 2
  #
  def [](obj)
    obj.method(self)
  end

  # Define a singleton method on object o, using Proc or Method f as its body.
  # This symbol is used as the name of the method.
  # Examples:
  #
  #  :singleton[o] = lambda { puts "this is a singleton method of o" }
  #  :class_method[String] = lambda { puts "this is a class method" }
  #
  # Note that you can't create instance methods this way. See Module.[]=
```

```
  #
  def []=(o,f)
    # We can't use self in the block below, as it is evaluated in the
    # context of a different object. So we have to assign self to a variable.
    sym = self
    # This is the object we define singleton methods on.
    eigenclass = (class << o; self end)
    # define_method is private, so we have to use instance_eval to execute it.
    eigenclass.instance_eval { define_method(sym, f) }
  end
end
```

通过这里定义的 Symbol.[]方法，以及前面描述的 Functional 模块，我们可以写出如下精巧的（也是难读的）代码：

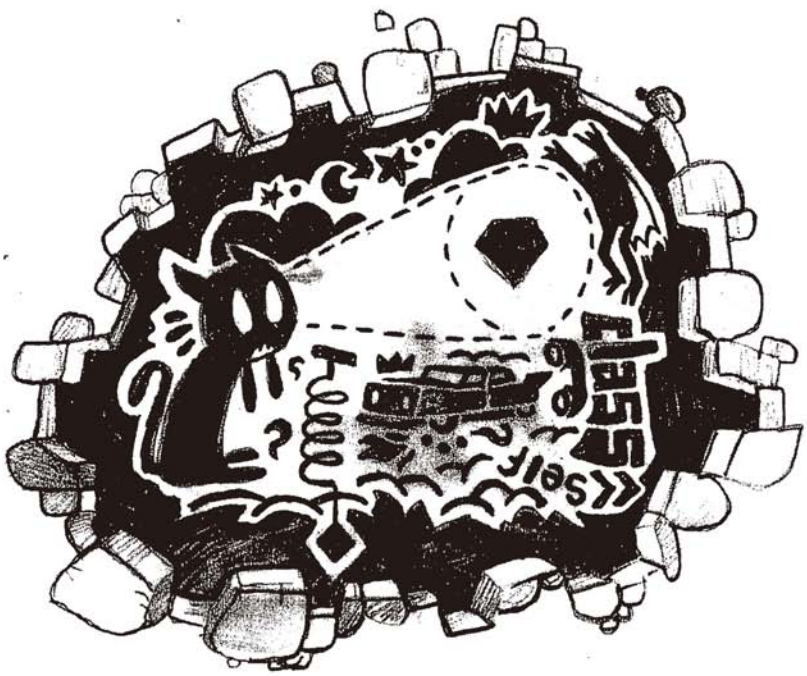
```
dashes = :*['-']          # Method * of '-'
puts dashes[10]           # Prints "----------"

y = (:+[1]*:*[2])[x]      # Another way to write y = 2*x + 1
```

Symbol 类定义的[]=与 Module 类的[]=相似，都使用 instance_eval 调用 define_method 方法，不同点在于单键方法并不像实例方法一样在类中定义，而是在对象的 *eigenclass* 中定义，在第 7 章中，我们还将见到 eigenclass。

第7章

类和模块

Classes and Modules



Ruby 是纯粹意义上的面向对象编程语言：Ruby 中的每个值都是（或至少行为上像）一个对象。每个对象都是某个类的一个实例，类定义了一个对象需要响应的一组方法。类可以扩展或子类化其他类，从而继承或重载父类中的方法。类还可以包含模块，或者从中继承方法。

Ruby 对象是严格封装的：只能通过定义的方法访问其内部状态。方法使用的成员变量在对象外部不能直接访问，不过可以通过定义读者方法（getter）、写者方法（setter）等访问器方法（accessor），使它们看起来好像是直接访问的。这些访问器方法对也被称为**属性**，它们与成员变量是完全不同的。类中定义的方法可以有“公开（public）”、“被保护（protected）”和“私有（private）”等可见性，它们会影响这些方法可以从哪里及怎样被调用。

与对象状态的封装性相反，Ruby 中的类非常开放。每个 Ruby 程序都可以为现有类添加方法，而且也可以为单个对象添加“单键方法（singleton method）”。

许多 Ruby 的面向对象架构都是 Ruby 内核的组成部分。其他一些部分（比如属性的创建和方法可见性的声明）则通过方法实现，而非真正的关键字。本章以一个扩展的教程开头，它展示了怎样定义一个类并添加方法。在这个教程后会有一些高级课题，包括：

- 方法可见性；
- 子类化和继承；
- 方法创建和初始化；
- 既作为命名空间也作为可包含“混入（mixin）”的模块；
- 单键方法和 eigenclass；
- 方法名解析算法；
- 常量名解析算法。

7.1 定义一个简单类
Defining a Simple Class

首先用一个可扩展的教程来探讨类相关的内容，这个教程开发了一个名为 Point 的类，它用来表示一个几何中的点，具有 X 和 Y 坐标，后面的子小节将演示：

- 定义一个新类；
- 为这个类创建实例；
- 为这个类添加一个初始化方法；
- 为这个类增加属性访问器方法；
- 为这个类定义操作符；

- 为这个类定义一个迭代器方法使之实现 Enumerable 接口；
- 重载一些重要的 Object 方法，比如 to_s、==、hash 和<=>；
- 定义类方法、类变量、类实例变量和常量。

7.1.1 创建类
Creating the Class

Ruby 使用 class 关键字创建类：

```
class Point
end
```

跟绝大多数 Ruby 结构（construct）一样，类定义以 end 结束。除了定义一个新类外，class 关键字还创建了一个常量用于引用这个类。这个常量名跟类名相同，因此所有类名必须以大写字母开头（译注 1）。

在类的代码体中（但是在任何实例方法定义体外），self 关键字表示定义的类。

像绝大多数 Ruby 语句一样，class 是一个表达式。class 表达式的值等于类定义体中最后一个表达式的值。一般而言，类的最后一个表达式都是一个用于定义方法的 def 语句，def 语句的值总是等于 nil。

7.1.2 实例化一个 Point 对象
Instantiating a Point

即便还没有为 Point 类添加任何内容，我们也可以对它进行实例化：

```
p = Point.new
```

常量 Point 指向一个类对象，代表我们定义的新类。所有类对象都有一个名为 new 的方法，用于创建一个新的实例。

我们把这个新创建的 Point 实例存放在一个局部变量 p 中，但是不能用它做什么有意思的工作，因为还没有为这个类定义任何方法。不过，我们可以询问这个对象是什么类型：

```
p.class          # => Point
p.is_a? Point    # => true
```

7.1.3 初始化一个 Point 对象
Initializing a Point

当创建 Point 对象时，我们希望用代表 X 和 Y 坐标的两个数对它进行初始化。在许多面向对象编程语言中，这是通过一个“构造函数（constructor）”来实现的。在 Ruby 中则是通过一个 initialize 方法实现的：

```
class Point
  def initialize(x,y)
    @x, @y = x, y
```

译注 1：因为常量以大写字母开头。

```
  end
end
```

这里只有三行新代码，不过发生了不少重要的事情。在第 6 章中我们详细解释了 def 关键字，不过在那里主要讲述的是如何定义一个全局可以访问的方法。当 def 语句在一个 class 定义中以非限定（unqualified）的方法名出现时（像本例中那样），它为这个类定义一个**实例方法**。实例方法是在一个类的实例上进行调用的方法。当一个实例方法被调用时，self 的值就是该方法定义所在类的一个实例。

另一个需要理解之处是 initialize 方法在 Ruby 中有特殊用处。类对象的 new 方法在创建一个实例后，自动调用该实例的 initialize 方法，传给 new 方法的所有参数被传递给 initialize 方法。因为我们的 initialize 方法需要两个参数，所以在调用 Point.new 方法时，须要提供两个值：

```
p = Point.new(0,0)
```

除了被 Point.new 自动调用外，initialize 方法会自动成为类的私有方法。对象自身可以调用 initialize 方法，不过不能显式对 p 调用 initialize 来重新初始化其状态。

现在让我们看看 initialize 方法的代码体，它接受两个参数，分别存在局部变量 x 和 y 中，然后把它们赋值给实例变量@x 和@y。实例变量总是用@打头，它们 “属于” 那个 self 指向的对象。每个 Point 类的实例都拥有这两个变量的拷贝，用于存储自己的 X 和 Y 坐标。

实例变量封装

对象的实例变量只能被该对象的实例方法访问。实例方法外的代码不能对实例变量进行读写操作（除非使用在第 8 章描述的反射机制）。

最后，要提醒那些熟悉 Java 或类似语言的程序员，在静态类型语言中，需要声明变量，这也包括实例变量。尽管知道 Ruby 变量是不用声明的，但是你可能还会觉得须要编写类似下面的代码：

```
# Incorrect code!
class Point
  @x = 0   # Create instance variable @x and assign a default. WRONG!
  @y = 0   # Create instance variable @y and assign a default. WRONG!

  def initialize(x,y)
    @x, @y = x, y   # Now initialize previously created @x and @y.
  end
end
```

这段代码并不会像 Java 程序员所期望的那样工作。实例变量只能在 self 的上下文中被解析，当 initialize 方法调用时，self 会指向一个 Point 实例。但是这段代码中它们却不在该方法中，而是属于 Point 类定义的一部分，这样，这两个赋值语句执行时，self 指向 Point 类本身而不是一个 Point 实例。在 initialize 方法内的@x 和@y 变量与在其外的该变量完全不同。

7.1.4 定义 to_s 方法
Defining a to_s Method

任何自定义的类都应该定义一个 to_s 实例方法，用来返回代表该对象的字符串。这在调试的时候会特别有用，下面显示如何为 Point 类定义这个方法：

```
class Point
  def initialize(x,y)
    @x, @y = x, y
  end

  def to_s        # Return a String that represents this point
    "(#@x,#@y)"   # Just interpolate the instance variables into a string
  end
end
```

有了这个新定义的方法，我们可以像下面这样创建一个点并把它打印出来：

```
p = new Point(1,2)    # Create a new Point object
puts p                # Displays "(1,2)"
```

7.1.5 访问器方法和属性
Accessors and Attributes

我们的 Point 类使用了两个实例变量。不过，正如我们所见到的，这两个变量只能在实例方法中被访问。如果想让 Point 类的用户可以使用一个点的 X 和 Y 坐标，我们须要提供访问器方法，让它们返回这些变量的值。

```
class Point
  def initialize(x,y)
    @x, @y = x, y
  end

  def x           # The accessor (or getter) method for @x
    @x
  end

  def y           # The accessor method for @y
    @y
  end
end
```

定义了这些方法，便可以像下面这样编写代码：

```
p = Point.new(1,2)
q = Point.new(p.x*2, p.y*3)
```

p.x和p.y这两个表达式看起来像变量引用，其实它们不过是省略了圆括号的方法调用。

如果想让 Point 类成为一个可变类（mutable）（这可能不是一个好主意），可以给实例变量增加写者（setter）方法：

```
class MutablePoint
  def initialize(x,y); @x, @y = x, y; end

  def x; @x; end       # The getter method for @x
  def y; @y; end       # The getter method for @y

  def x=(value)        # The setter method for @x
    @x = value
  end

  def y=(value)        # The setter method for @y
    @y = value
  end
end
```

前面讲过可以用赋值语句调用这样的写者方法，所以有了这些方法定义，我们可以编写如下代码：

```
p = Point.new(1,1)
p.x = 0
p.y = 0
```

在类中使用写者方法

一旦定义了像 x=这样的写者方法，你可能会试图在实例方法中使用它们。也就是说，你想用 x=2 来隐式调用 x=(2)，而不再用@x=2。不过这是不行的，x=2 只会创建一个新的局部变量。

这对学习使用 Ruby 写者方法的新手来说并非一个罕见的错误。只有当对一个对象使用赋值表达式时，才会调用它的写者方法。如果你希望在定义写者方法的类中使用这个写者方法，则要通过 self 显式地调用它，比如：self.x=2。

给实例变量增加这种简单读写方法的需求俯拾皆是，因此 Ruby 提供了一种方法使之自动化。Module 类定义了 attr_reader 和 attr_accessor 方法，因为所有的类都是模块（Class 类是 Module 的子类），所以可以在任何类的定义中调用这些方法。每个方法都接受任意数目的符号（用作属性名）作为参数，attr_reader 为给定名字的实例变量创建同名的读者方法，因此，如果我们想要创建一个可变的（mutable）Point 类，可以像下面这样：

```
class Point
  attr_accessor :x, :y # Define accessor methods for our instance variables
end
```


如果想定义一个不变（immutable）版本的类，可以像下面这样：

```
class Point
  attr_reader :x, :y  # Define reader methods for our instance variables
end
```

每个方法的属性名除了用符号表示，也可以用字符串表示。尽管提倡使用符号，下面的写法也是允许的：

```
attr_reader "x", "y"
```

attr是一个类似的方法，它的名字更短一些。不过，它在Ruby 1.8和1.9中的行为并不相同。在Ruby 1.8中，attr一次只能定义一个属性。如果只给出一个符号对象做参数，它会定义一个读者方法；如果在符号后面在带上一个true，它会同时定义一个写者方法：

```
attr :x          # Define a trivial getter method x for @x
attr :y, true    # Define getter and setter methods for @y
```

在Ruby 1.9中，可以像Ruby 1.8一样使用attr，不过也可以把它用作attr_reader的同义词。

attr、attr_reader和attr_accessor方法可以为我们创建实例方法。这是**元编程**（*metaprogramming*）的一个例子，它展示了Ruby的一个强大特性。在第8章中会有更多关于元编程的例子。注意attr这些方法在类的定义内（并在所有方法定义外）被调用，这使得它们仅在类定义时执行一次。在这里没有效率方面的问题：这样创建的读者和写者方法和硬编码出来的方法是一样快的。记住，这样创建的读者和写者方法只能做简单的工作：它们仅仅简单地映射同名的实例变量。如果想得到更复杂的访问器，比如写者方法给另外一个变量赋值，或者读者方法返回从两个变量计算得到的值，你只能自己动手定义。

7.1.6 定义操作符
Defining Operators

我们想用+操作符来为两个Point对象实现向量加的功能，用*操作符实现Point的标量乘法，以及用一元减操作符实现乘以-1的等价功能。像+这样基于方法的操作符就是以标点符号命名的方法。因为减号操作符有一元和二元两种形式，Ruby用-@来作为一元减的方法名。下面是带有数学操作符定义的Point类：

```
class Point
  attr_reader :x, :y   # Define accessor methods for our instance variables

  def initialize(x,y)
```

```
    @x,@y = x, y
  end

  def +(other)          # Define + to do vector addition
    Point.new(@x + other.x, @y + other.y)
  end

  def -@                # Define unary minus to negate both coordinates
    Point.new(-@x, -@y)
  end

  def *(scalar)         # Define * to perform scalar multiplication
    Point.new(@x*scalar, @y*scalar)
  end
end
```

让我们看看+方法的实现。它可以使用 self（方法调用时所在对象）中的实例变量@x，但是不能访问其他 Point 对象的@x 实例变量。Ruby 没有这样的语法，所有的实例变量都隐式使用 self，因此，我们的+方法须要使用 x 和 y 这两个读者方法。（稍后我们可以看到如何让某些方法只允许被同类型的对象访问，而禁止被其他类型的对象访问）

类型检查和Duck Typing

我们的+方法没有做类型检查，它只是简单地假定会得到一个合适的对象。在 Ruby 中，对于“合适”的定义是相当宽松的。在上面的例子中，任何具有 x 和 y 方法（只要它们不需要参数并返回一个数值）的对象都是合适的。我们不关心这个对象是不是真正的 Point 对象，只要它看起来像是一个 Point 对象就行了。这种方式有时也称为“duck typing”，它源自一句格言“如果它走路像鸭子并且也像鸭子一样嘎嘎叫，那它就是一只鸭子。”

如果我们传给+方法的对象不适合，Ruby 会抛出一个异常。比如，我们试图给一个点加上 3，会产生如下的错误信息：

```
NoMethodError: undefined method `x' for 3:Fixnum
        from ./point.rb:37:in `+'
```

这表明 3 这个 Fixnum 对象没有名为 x 的方法，因此 Point 类的+方法会产生错误。这可以帮助我们找到问题的源头，不过这看起来有点抽象。如果检查参数的类型，会让代码在使用这个方法时更容易调试。下面是带有类检验的版本：

```
def +(other)
  raise TypeError, "Point argument expected" unless other.is_a? Point
  Point.new(@x + other.x, @y + other.y)
end
```

下面的例子中对类型检验比较宽松，它给出更细致的错误信息，而且还支持 duck typing：

```
  def +(other)
    raise TypeError, "Point-like argument expected" unless
      other.respond_to? :x and other.respond_to? :y
    Point.new(@x + other.x, @y + other.y)
  end
```

注意这个版本还是假定 x 和 y 方法会返回数值，如果其中一个方法返回的是如字符串这样的对象，我们得到的错误消息还是相当抽象。

另外一种类型检验的方式是放在事情发生之后，在执行完这些代码后来处理任何发生的异常，并且再抛出一个更加适合的异常对象：

```
  def +(other)        # Assume that other looks like a Point
    Point.new(@x + other.x, @y + other.y)
  rescue              # If anything goes wrong above
    raise TypeError,  # Then raise our own exception
      "Point addition with an argument that does not quack like a Point!"
  end
```

注意，我们定义的*方法需要一个数值参数，而非 Point 对象。如果 p 是一个点，p*2 是允许的；不过由于我们的实现方式，2*p 是不能工作的。第二个表达式是对 Integer 类调用*方法，这个类根本不知道如何与 Point 类一块工作，因为 Integer 类不知道怎样和一个点相乘，它调用 Point 的 coerce 方法来寻求帮助（参见第 3.8.7.4 节来获得更多细节）。如果我们想让 2*p 与 p*2 返回同样的结果，可以定义一个 coerce 方法：

```
# If we try passing a Point to the * method of an Integer, it will call
# this method on the Point and then will try to multiply the elements of
# the array. Instead of doing type conversion, we switch the order of
# the operands, so that we invoke the * method defined above.
def coerce(other)
  [self, other]
end
```

7.1.7 用[ ]访问数组和哈希表

Array and Hash Access with [ ]

Ruby 用方括号访问数组和哈希表，并且也允许任何类定义[]方法来使用方括号操作符。我们可以为这个 Point 类定义一个[]方法，使 Point 对象就像一个长度为 2 的只读数组，或者像一个具有主键:x 和:y 的只读哈希表：

```
# Define [] method to allow a Point to look like an array or
# a hash with keys :x and :y
def [](index)
  case index
  when 0, -2: @x      # Index 0 (or -2) is the X coordinate
  when 1, -1: @y      # Index 1 (or -1) is the Y coordinate
  when :x, "x": @x    # Hash keys as symbol or string for X
  when :y, "y": @y    # Hash keys as symbol or string for Y
```

```
    else nil              # Arrays and hashes just return nil on bad indexes
    end
  end
```


7.1.8 枚举坐标值
Enumerating Coordinates

如果一个 Point 对象可以像一个两元素数组一样被访问，我们可能会想让它可以像一个真正数组一样迭代出所有元素。下面我们为 Point 类定义了 each 迭代器，因为 Point 对象永远只会有两个元素，这个迭代器无须构造一个循环，只须要调用 yield 两次：

```
# This iterator passes the X coordinate to the associated block, and then
# passes the Y coordinate, and then returns. It allows us to enumerate
# a point as if it were an array with two elements. This each method is
# required by the Enumerable module.
def each
  yield @x
  yield @y
end
```

定义了这个迭代器，我们可以编写如下的代码：

```
p = Point.new(1,2)
p.each {|x| print x }   # Prints "12"
```

更重要的是，一旦定义了 each 迭代器，我们就可以混入 Enumerable 模块的一些方法，这些方法都是基于 each 定义的。这样，通过加入下面这一行代码，它就会获得超过 20 个迭代器：

```
include Enumerable
```

如果加入了这行代码，我们可以写出如下这样有趣的代码：

```
# Is the point P at the origin?
p.all? {|x| x == 0 } # True if the block is true for all elements
```

7.1.9 Point 的相等性
Point Equality

到目前为止，这个 Point 类的任意两个实例永远无法相等，即使它们的 X 和 Y 坐标都相同。为了补救，我们必须实现==操作符。（也许要重读第 3 章的第 3.8.5 节来回顾一下 Ruby 中各种关于相等性的标记。）

下面是 Point 类的一个==方法实现：

```
def ==(o)                  # Is self == o?
  if o.is_a? Point         # If o is a Point object
    @x==o.x && @y==o.y     # then compare the fields.
  elsif                    # If o is not a Point
    false                  # then, by definition, self != o.
  end
end
```

Duck Typing 和相等性

我们先前实现的+操作符根本没有做类型检验：就好像每个传进来的参数对象都会具有返回数值的 x、y 方法一样。这个==方法则不同：它不允许 duck typing，而是要求输入参数必须是一个 Point 对象。这是一种实现上的选择，上面代码所定义的相等性要求某个 Point 的对象必须自身就是一个 Point 对象。

不同的实现可以更加严格，也可以更加宽松。上面的实现使用了 is_a?方法来判定参数的类别，这允许一个 Point 子类的实例与一个 Point 实例相等。更严格的实现可以使用 instance_of?方法来禁止子类实例。相似地，上面的实现使用了==来比较 X 和 Y 坐标，==操作符允许类型转换，这意味着点 (1, 1) 与 (1.0, 1.0) 相等。这可能正是它应该做的，不过我们也可以用更严格的 eql?方法来定义相等性。

一个更宽松的等价性实现可以支持 duck typing。不过还需要多加小心，在参数对象没有 x 和 y 方法时，不能抛出一个 NoMethodError 异常，相反，它应该返回 false：

```
def ==(o)                       # Is self == o?
  @x == o.x && @y == o.y        # Assume o has proper x and y methods
rescue                          # If that assumption fails
  false                         # Then self != o
end
```

前面在第 3.8.5 节中我们提到 Ruby 对象还要定义一个 eql?方法来测试相等性。默认情况下，eql?方法与==操作符一样，检测的是对象标志相等性而非对象内容相等性。我们往往希望 eql?方法与==操作符有同样的工作方式，这可以通过如下的别名来实现：

```
class Point
  alias eql? ==
end
```

另一方面，有两个原因使得我们有时不想让 eql?与==等价。首先，有些类定义 eql?方法进行更严格的相等性测试。比如，在 Numeric 类及其子类中，==允许类型转换，而 eql?不允许。如果我们相信 Point 类的用户希望用两种不同的方式比较实例，可以用这样的方式定义相等性。因为点就是两个数值，所以用类似数值的相等性方式也是说得通的。这样定义的 eql?方法跟==很相似，不过在比较点坐标时使用的是 eql?而不是==：

```
def eql?(o)
  if o.instance_of? Point
    @x.eql?(o.x) && @y.eql?(o.y)
  elsif
    false
```

```
    end
  end
```

值得注意的是，对于任何实现任意对象集合的类（集合、列表和树），这都是正确的方式，==操作符应该用成员对象的==操作符进行比较，而 eql?应该使用成员对象的 eql?方法进行比较。

用不同于==的方式实现 eql?方法的第二个原因是，你可能想在该对象作为哈希主键时有特殊行为。Hash 类的 eql?方法使用主键（而非值）进行比较，如果不定义 eql?方法，哈希表会用对象标志对对象实例进行比较，这意味着如果有一个元素的主键是 p，那么你就只能使用 p 来获取这个元素，即使 p==q，也不能用 q 来获取该元素。可变对象（mutable object）不适合做哈希表的主键，让 eql?方法保持未定义可以干净地绕过这个问题（参见第 3.4.2 节，查看更多关于哈希和可变对象做主键的内容）。

因为 eql?方法用于哈希对象，所以永远不能只单独定义它，还必须定义一个方法来计算对象的哈希码。如果两个对象用 eql?方法表明相等，那么它们的 hash 方法必须返回同样的值。（两个不相等的对象有可能有同样的哈希码，但是应该尽力避免这样的情形。）

实现最优的 hash 方法十分困难，幸运的是，有一个简单的方法可以完美地适用于计算任意对象的哈希码：简单地把该类中所有使用对象的哈希码组合起来（更精确的描述是：把所有用 eql? 进行比较的对象的哈希码组合起来）。其技巧在于用适合的方式来组合哈希码，下面的方式不值得推荐：

```
def hash
  @x.hash + @y.hash
end
```

这种方法的问题在于它对点（1，0）和点（0，1）返回同样的哈希码，这是合法的，不过在用 Point 对象作为主键时，哈希表的效率会降低。使用如下的方法会好一点：

```
def hash
  code = 17
  code = 37*code + @x.hash
  code = 37*code + @y.hash
  # Add lines like this for each significant instance variable
  code  # Return the resulting code
end
```

这个通用的生成哈希码的方式应该适用于绝大多数 Ruby 类，这个例子（及 17、37 这两个常量）是从 Joshua Bloch 的 *Effective Java*（Prentice Hall 出版社出版）一书中改造而来的。

7.1.10 对点进行排序
Ordering Points

设想我们想为 Point 对象定义一个顺序，这样可以对它们进行比较和排序。对点设定顺序有多种方式，我们选择按照它们到原点距离的方式。这个距离用毕达哥拉斯定理计算：X 和 Y 坐标的平方之和的平方根。

要为 Point 对象定义这样的顺序，只须定义<=>操作符（参见第4.6.6节），并包含 Comparable 模块，这样会把我们实现的<=>通用操作符混入相等性及关系运算符中。<=>操作符比较 self 和作为参数传入的对象，如果 self 小于那个对象，返回-1；如果两个对象相等，则返回 0；如果 self 大于那个对象，返回 1。（当两个对象不是可比类型时，返回 nil。）下面是我们对<=>的实现，有两件事值得注意：首先，它没有使用 Math.sqrt 方法，而是简单比较坐标平方之和；其次，在计算了平方和之后，它只是简单地把自身代理给 Float 类的<=>操作符。

```
include Comparable  # Mix in methods from the Comparable module.

# Define an ordering for points based on their distance from the origin.
# This method is required by the Comparable module.
def <=>(other)
  return nil unless other.instance_of? Point
  @x**2 + @y**2 <=> other.x**2 + other.y**2
end
```

注意 Comparable 接口使用我们定义的<=>来定义==方法，我们定义的这个基于距离的比较符会使得==方法认为点(1,0)与(0,1)是相等的。不过，因为我们对 Point 类显式定义了自己的==方法，Comparable 模块的==方法没有机会被调用。理想情况下，==和<=>操作符对于相等性的定义是一致的，不过这在我们的 Point 类中是不可能的。我们的操作符最终会得到如下结果：

```
p,q = Point.new(1,0), Point.new(0,1)
p == q      # => false: p is not equal to q
p < q       # => false: p is not less than q
p > q       # => false: p is not greater than q
```

最后，值得注意的是 Enumerable 模块定义了若干方法，比如 sort、min 和 max，它们只有在被枚举的对象定义了<=>操作符后才能运作。

7.1.11 可变 Point 类
A Mutable Point

我们现在开发的 Point 类是**不变**（*immutable*）的：在一个点对象被创建后，没有公开的 API 来修改该点的 X 和 Y 坐标。这应该是正确的方式，不过让我们来看看如果希望点对象是可变的，我们可以做些什么。

首先，我们需要 x=和 y=这两个写者方法，使 X 和 Y 坐标可以直接修改。我们可以显式定义这两个方法，或者只是简单地把 attr_reader 替换为 attr_accessor：

```
attr_accessor :x, :y
```

接下来，我们修改+操作符的实现。当我们希望对点 p 的坐标加上点 q 的坐标时，我们可以直接修改 p 的坐标值，而不是返回一个新创建的 Point 对象。我们把这个方法命名为 add!，用一个感叹号表示这个方法会修改调用对象的内部状态：

```
def add!(p)          # Add p to self, return modified self
  @x += p.x
  @y += p.y
  self
end
```

在增加一个可变方法时，如果该方法还有一个不变方式的版本，我们通常在方法名后加上一个感叹号。在这个例子中，如果我们还有一个名为 add 的方法返回一个新创建的对象时，add!这个方法名会变得更有意义。可变方法的不变版本通常只在复制的对象上使用可变方法来实现：

```
def add(p)             # A nonmutating version of add!
  q = self.dup         # Make a copy of self
  q.add!(p)            # Invoke the mutating method on the copy
end
```

在这个简单的例子中，我们的 add 方法与已经定义的+操作符基本相同，所以它不是很必要的。如果无须定义一个不变版本的 add，我们可以考虑去掉 add!方法名中的感叹号，让 add 这个名字本身（注意用的是 add 而非 plus）来表示它是一个可变方法。

7.1.12 轻松快速地创建可变类
Quick and Easy Mutable Classes

如果想创建一个可变 Point 类，有一种方法，即用 Struct 类进行创建。Struct 是一个 Ruby 内核类，可以用于生成其他类。在这些生成的类中，你指定的字段名自动具有访问器方法。用 Struct.new 来创建一个新类可以有两种方式：

```
Struct.new("Point", :x, :y)   # Creates new class Struct::Point
Point = Struct.new(:x, :y)    # Creates new class, assigns to Point
```

命名匿名类

上面例子中的第二行代码使用了一个 Ruby 类的有趣特性：如果把一个未命名的类对象赋值给一个常量，这个常量名就成为该类的名字。在使用 Class.new 这个构造方法时，你能看到同样的行为：

```
C = Class.new    # A new class with no body, assigned to a constant
c = C.new        # Create an instance of the class
c.class.to_s     # => "C": constant name becomes class name
```

一旦使用 Struct.new 创建了一个类，它可以像其他任何类一样使用，它的 new 方法使用每一个指定的有名字段为参数，同时还为这些字段创建读写方式的访问器方法：

```
p = Point.new(1,2)    # => #<struct Point x=1, y=2>
p.x                   # => 1
p.y                   # => 2
p.x = 3               # => 3
p.x                   # => 3
```

Struct 类还定义了[]和[]=操作符，可以实现数组和哈希表方式的索引方式，它甚至还提供了 each 和 each_pair 迭代器，可以迭代一个实例中的每个字段值：

```
p[:x] = 4             # => 4: same as p.x =
p[:x]                 # => 4: same as p.x
p[1]                  # => 2: same as p.y
p.each {|c| print c} # prints "42"
p.each_pair {|n,c| print n,c }   # prints "x4y2"
```

基于 Struct 的类自动具有可用的==操作符，在类实例作为哈希表的主键会用到它（不过因为它们是可变的，使用时需要注意）；另外，还默认拥有一个有用的 to_s 方法：

```
q = Point.new(4,2)
q == p        # => true
h = {q => 1} # Create a hash using q as a key
h[p]          # => 1: extract value using p as key
q.to_s        # => "#<struct Point x=4, y=2>"
```

像这样定义的 Point 类不会自动具有与点相关的方法，比如我们前面定义的 add!方法和<=>操作符。不过，可以自己添加这些方法。Ruby 的类定义并非是静态的，任何类（包括用 Struct.new 定义的类）都是“开放的”，可以向其中增加方法。下面的 Point 类最初用 Struct 定义，然后添加了与点相关的方法：

```
Point = Struct.new(:x, :y)    # Create new class, assign to Point
class Point                   # Open Point class for new methods
  def add!(other)             # Define an add! method
    self.x += other.x
    self.y += other.y
    self
  end
```

```
  include Comparable          # Include a module for the class
  def <=>(other)              # Define the <=> operator
    return nil unless other.instance_of? Point
    self.x**2 + self.y**2 <=> other.x**2 + other.y**2
  end
end
```

正如本节开始时所提到的，Struct 类用于创建可变类。不过，只要做一点工作，我们就可以把基于 Struct 的类变为不变类：

```
Point = Struct.new(:x, :y)    # Define mutable class
class Point                   # Open the class
  undef x=,y=,[]=             # Undefine mutator methods
end
```

7.1.13 一个类方法
A Class Method

让我们用另外一种方式对 Point 对象进行加操作。它不是通过一个 Point 对象的实例方法实现加一个点，而是用一个名为 sum 的新方法，可以接受任意数量的 Point 对象为参数，把它们加在一起，然后返回一个新的 Point 对象。这个方法不是某个 Point 对象的实例方法，而是一个**类方法**，可以通过 Point 类本身进行调用。我们可以用如下方式调用 sum 方法：

```
total = Point.sum(p1, p2, p3)  # p1, p2 and p3 are Point objects
```

记住，表达式 Point 指向一个 Class 对象，它表示用于描述点的类。要给 Point 类增加一个类方法，其实是要给 Point 对象定义一个单键方法（在第 6.1.4 节我们介绍了单键方法）。要定义一个单键方法，我们可以像平常一样使用 def 语句，不过需要在给定方法名的同时指明方法所属的对象。类方法 sum 定义如下：

```
class Point
  attr_reader :x, :y      # Define accessor methods for our instance variables

  def Point.sum(*points)  # Return the sum of an arbitrary number of points
    x = y = 0
    points.each {|p| x += p.x; y += p.y }
    Point.new(x,y)
  end

  # ...the rest of class omitted here...
end
```

这个定义显式使用了类的名字，这和调用时的句法是一样的。类方法还可以用 self 而非类名进行定义，这样该方法就成为如下方式：

```
def self.sum(*points)  # Return the sum of an arbitrary number of points
  x = y = 0
  points.each {|p| x += p.x; y += p.y }
```

```
    Point.new(x,y)
  end
```

使用 self 而非 Point 让这段代码看起来没那么清晰，但是它是 DRY（Don't Repeat Yourself，不要重复自己）原则的一个应用。使用 self 而不是类名，在改变类名时，无须修改类方法的定义。

还有一种方式可以定义类对象，尽管它比上面两种方式更加晦涩，不过在定义多个类方法时，它更加方便。在现有代码中，我们会经常看到下面这种方式：

```
# Open up the Point object so we can add methods to it
class << Point      # Syntax for adding methods to a single object
  def sum(*points)  # This is the class method Point.sum
    x = y = 0
    points.each {|p| x += p.x; y += p.y }
    Point.new(x,y)
  end

  # Other class methods can be defined here
end
```

这种技术也可以在类定义体内使用，在这里可以用 self 来代替类名：

```
class Point
  # Instance methods go here

  class << self
    # Class methods go here
  end
end
```

在第 7.7 节我们将学到更多关于这种句法的知识。

7.1.14 常量
Constants

很多类都应该定义一些常量，下面是对 Point 类有用的一些常量：

```
class Point
  def initialize(x,y)  # Initialize method
    @x,@y = x, y
  end

  ORIGIN = Point.new(0,0)
  UNIT_X = Point.new(1,0)
  UNIT_Y = Point.new(0,1)

  # Rest of class definition goes here
end
```

在类定义体内，这些常量可以用非限定名进行引用。当然，在定义体外，则须要使用类名做前缀进行访问：

```
Point::UNIT_X + Point::UNIT_Y   # => (1,1)
```

注意，在本例中的常量使用了该类的实例，因此在没有定义该类的 initialize 方法前，这些常量无法定义。另外，在类外定义这些 Point 的常量也是完全合法的：

```
Point::NEGATIVE_UNIT_X = Point.new(-1,0)
```

7.1.15 类变量
Class Variables

类变量对所有该类实例的方法都可见，也对类定义本身可见。与实例变量相似，类变量也被封装在类中；它们可以被类的实现代码所使用，不过对该类的用户是不可见的。类变量名以@@打头。

对于 Point 类来说，其实并不真正需要类变量。不过作为教程，假设我们想统计创建了多少个 Point 对象并计算它们的平均坐标值，下面是一种实现方式：

```
class Point
  # Initialize our class variables in the class definition itself
  @@n = 0             # How many points have been created
  @@totalX = 0        # The sum of all X coordinates
  @@totalY = 0        # The sum of all Y coordinates

  def initialize(x,y) # Initialize method
    @x,@y = x, y      # Sets initial values for instance variables

    # Use the class variables in this instance method to collect data
    @@n += 1          # Keep track of how many Points have been created
    @@totalX += x     # Add these coordinates to the totals
    @@totalY += y
  end

  # A class method to report the data we collected
  def self.report
    # Here we use the class variables in a class method
    puts "Number of points created: #@@n"
    puts "Average X coordinate: #{@@totalX.to_f/@@n}"
    puts "Average Y coordinate: #{@@totalY.to_f/@@n}"
  end
end
```

在上面的代码中，值得注意的是实例方法、类方法和类定义本身（不在任何方法中）都使用了类变量。类变量与实例变量有本质区别，我们已经看到实例变量总是在 self 中被引用

求值，这也是实例变量在实例方法内和实例方法外被引用时完全不同的原因；另一方面，类变量总是在创建它的类对象中被引用求值的。

7.1.16 类实例变量
Class Instance Variables

类也是对象，它们可以像其他对象一样拥有实例变量。类的实例变量——通常被称为类实例变量——与类变量不同，不过它们十分相似，以至于类实例变量经常被用来代替类变量。

在类定义体内而在实例方法定义体外使用的实例变量被称为类实例变量。跟类变量相似，类实例变量与类本身相关联而非与类的某个特定实例关联。类实例变量的一个不足在于它们不能像类变量那样在实例方法中被使用，另一个不足之处则在于它们容易与普通的实例变量混淆，由于没有使用独特的前缀符号，有时候很容易混淆一个实例变量是与类关联还是与类的实例关联。

类实例变量优于类变量的一个最重要的特性是当继承现有类时，类实例变量的行为不像类变量那样让人混淆。在本章的后面部分将对此进行仔细介绍。

让我们把有统计功能版本的 `Point` 类改写为使用类实例变量而不是类变量。由于类实例变量不能被实例方法使用，我们不得不把这些统计代码从 `initialize` 方法（这是一个实例方法）中提取到 `new` 这个类方法中，它用于创建 Point 对象：

```
class Point
  # Initialize our class instance variables in the class definition itself
  @n = 0                # How many points have been created
  @totalX = 0           # The sum of all X coordinates
  @totalY = 0           # The sum of all Y coordinates

  def initialize(x,y) # Initialize method
    @x,@y = x, y        # Sets initial values for instance variables
  end

  def self.new(x,y)   # Class method to create new Point objects
    # Use the class instance variables in this class method to collect data
    @n += 1             # Keep track of how many Points have been created
    @totalX += x        # Add these coordinates to the totals
    @totalY += y

    super               # Invoke the real definition of new to create a Point
                        # More about super later in the chapter
  end
```

```
  # A class method to report the data we collected
  def self.report
    # Here we use the class instance variables in a class method
    puts "Number of points created: #@n"
    puts "Average X coordinate: #{@totalX.to_f/@n}"
    puts "Average Y coordinate: #{@totalY.to_f/@n}"
  end
end
```

因为类实例变量不过是类对象的实例变量，我们可以用 `attr`、`attr_reader` 和 `attr_accessor` 为它们创建访问器方法。不过，要注意在合适的上下文中使用这些元编程的方法。还记得可以使用 `class << self` 语法来创建类方法吗？同样的句法也可以用来为类实例变量定义访问器方法：

```
class << self
  attr_accessor :n, :totalX, :totalY
end
```

在定义了这些访问器后，就可以用 `Point.n`、`Point.totalX` 和 `Point.totalY` 来引用原始数据了。

7.2 方法可见性：Public、Protected、Private
Method Visibility: Public, Protected, Private

实例方法的可见性可以是**公开**（*public*）、**私有**（*private*）或**被保护**（*protected*）的。如果使用过其他的面向对象编程语言，你可能对这些术语已经很熟悉了。不过要小心，Ruby 语言的这些术语的含义与其他语言多少有一些区别。

如果没有明确声明为 private 或 protected，方法默认的可见性是 public 的。一个例外是 `initialize` 方法，它总是默认为 private；另外一个例外是所有在类外定义的“全局”方法——这些方法被定义为 `Object` 对象的私有实例方法，一个公开（public）方法可以在任何地方被调用——对它们的使用没有任何限制。

私有（private）方法是实现一个类时使用的内部方法，它只能被这个类（或它的子类，我们在后面会讲到）的实例方法所调用。私有方法只能隐式地被 `self` 对象调用，并且不能通过一个对象进行显式调用。如果 `m` 是一个私有方法，那么只能用 `m` **这种方式**来调用它，而不能用 `o.m` 或 `self.m` 来调用它。

被保护（protected）方法与私有方法的相似之处在于它也只能在该类或子类的内部被调用；它与私有方法的不同之处在于它可以被该类的实例显式调用，而不仅仅只能被 `self` 所隐式调用。例如，一个被保护方法可以定义一个访问器，让该类的不同实例可以共享它们的内部状态，但该类的用户则不能访问这个内部状态。

被保护方法最不常用，而且也最难理解。对于什么时候可以调用一个被保护方法，其规则可以用如下方式进行正式描述：一个被类 C 所定义的被保护方法，只有当对象 o 和 p 的类都是 C 的子类（或 C 本身）时，p 中的方法才可以调用对象 o 上的该方法。

方法可见性是通过三个分别名为 public、private 和 protected 的方法来定义的。这三个方法都是 Module 类的实例方法。所有的类都是模块，在类的定义体内（但在方法定义体外），self 指向被定义的类。因此，public、private 和 protected 方法就可以像语言的关键字一样被使用，但实际上，它们不过是在 self 上被调用的方法而已。有两种方式可以调用这些方法。如果不使用参数，它们把后面所有紧跟的方法用给定的可见性进行定义。类可以像下面这样使用这些方法：

```
class Point
  # public methods go here

  # The following methods are protected
  protected

  # protected methods go here

  # The following methods are private
  private

  # private methods go here
end
```

这些方法也可以用一个或多个方法名（字符串或者符号）做参数进行调用。如果使用这种方式，它们会更改给定方法的可见性。这种用法要求可见性的声明必须在该方法的定义之后，一种做法是在类的末尾处一次性声明所有的私有方法和被保护方法。下面是一个例子，其中定义了一个类，它具有一个名为 utility_method 的私有方法和一个被保护的访问器方法：

```
class Widget
  def x                          # Accessor method for @x
    @x
  end
  protected :x                   # Make it protected

  def utility_method             # Define a method
    nil
  end
  private :utility_method        # And make it private
end
```

在 Ruby 中，public、private 和 protected 仅仅应用于方法。被封装的实例变量和类变

量在效果上是私有的，而常量则在效果上是公开的。没法让一个实例变量可以在类外被访问（当然，除了定义一个访问器方法）；同样，也无法定义一个类外无法使用的常量。

偶尔，把一个类方法声明为私有也是有用的。例如，如果某个类要定义工厂方法，你可能希望 new 方法变为私有的。要做到这点，你可以使用 private_class_method 方法，它的参数是一个或多个用符号表示的方法名：

```
private_class_method :new
```

可以用 public_class_method 方法把一个私有类方法再次变为公开方法。这两个方法都不能像 public、protected 和 private 方法那样使用无参数的方式。

Ruby 在设计上就是一个十分开放的语言。把一些方法声明为私有或被保护的是良好的编程风格，可以避免因粗心大意而使用那些不属于公开 API 的方法。不过，理解这点非常重要：Ruby 的元编程能力可以很容易实现调用私有或被保护方法，甚至是访问封装的实例对象。要调用前面定义的 utility_method 私有方法，我们可以使用 send 方法，也可以使用 instance_eval 方法在该对象的上下文中执行一个代码块：

```
w = Widget.new                          # Create a Widget
w.send :utility_method                  # Invoke private method!
w.instance_eval { utility_method }      # Another way to invoke it
w.instance_eval { @x }                  # Read instance variable of w
```

如果想通过方法名调用方法，但是不想不小心调用了一个私有方法，可以（在 Ruby 1.9 中）使用 public_send 而非 send。前者与 send 类似，但是不会调用接收者的私有方法。在第 8 章中将详细介绍 public_send、send 及 instance_eval。

7.3 子类化和继承
Subclassing and Inheritance

绝大多数的面向对象编程语言——包括 Ruby——都支持**子类化**机制，它允许基于一个已经存在的类的行为进行修改，并创建一个新的类。我们对于子类化的讨论会从一些基本的术语开始，如果用过 Java、C++或类似语言，你很可能已经熟知这些术语了。

当定义一个类时，我们可能会指定它从某个被称为**超类**（*superclass*）的类进行**扩展**（*extend*）或**继承**（*inherit from*）。如果我们定义类 Ruby 扩展(extend)了类 Gem，我们称类 Ruby 是 Gem 的一个**子类**（*subclass*），而 Gem 是 Ruby 的**超类**（*superclass*）。如果定义一个类时不指定超类，那么它隐式地继承自 Object 类。一个类可以有任意多的子类，而每个类只能有一个超类。Object 是个例外，它没有超类。

类可以有多个子类，但是只能有一个超类，这个事实意味着类可以被组织成一个树型结构，这被称为 Ruby **类等级**（*hierarchy*），Object 类是这个等级的根（root），所有其他类都直接或间接继承自它。一个类的**后代**（*descendant*）指它的子类及子类的子类（这样一直递归下去）；一个类的**祖先**（*ancestor*）指它的父类及父类的父类（一直向上到 Object）。在第 5 章中的图 5-5 显示了 Ruby 类等级的一个部分，包括 Exception 类及它所有的后代。在那幅图中，我们可以看到 EOFError 类的祖先有 IOError、StandardError、Exception 和 Object。

Ruby 1.9 中的 BasicObject

在 Ruby 1.9 中，Object 不再是类等级中的根，一个名为 BasicObject 的新类取代了这个位置，而 Object 被称为 BasicObject 的子类。BasicObject 类非常简单，几乎没有自己的方法，它在作为代理包装类的超类时会很有用（就像第 8 章中示例 8-5 所示的那样）。

在 Ruby 1.9 中创建类时，除非指定超类，否则仍然从 Object 继承。绝大多数的程序员永远不会使用或扩展 BasicObject 类。

扩展一个类的句法很简单，仅在 class 语句后添加一个<字符及超类的名字即可。例如：

```
class Point3D < Point    # Define class Point3D as a subclass of Point
end
```

在后面的小节中，我们将充实这个三维的 Point 类，显示如何从超类中继承方法，以及如何覆盖或增强继承来的方法，使之在子类中具有新的行为。

子类化一个 Struct

在本章的前面部分，我们看到了如何用 Struct.new 自动创建一个简单类。我们也可以子类化一个基于结构的类，这样可以添加那些不能自动产生的方法：

```
class Point3D < Struct.new("Point3D", :x, :y, :z)
  # Superclass struct gives us accessor methods, ==, to_s, etc.
  # Add point-specific methods here
end
```

7.3.1 继承方法
Inheriting Methods

我们刚刚定义的 Point3D 类是一个无关痛痒的 Point 子类，它声明自己是扩展自 Point 类的，但 class 定义的部分没有内容，所以它什么也没有添加。在效果上，一个 Point3D 对象就是一个 Point 对象，唯一能看到的一个区别是 class 方法的返回值：

```
p2 = Point.new(1,2)
p3 = Point3D.new(1,2)
print p2.to_s, p2.class   # prints "(1,2)Point"
print p3.to_s, p3.class   # prints "(1,2)Point3D"
```

class 方法的返回值是不同的，不过更引人注目的是那些相同的部分。这个 Point3D 对象继承了 Point 类定义的 to_s 方法，同时还继承了 initialize 方法——这使得我们可以以与创建 Point 对象一样的方式用 new 方法来创建 Point3D 对象（注 1）。在上面这段代码中还有另外一个方法继承的例子：Point 和 Point3D 都继承了 Object 中定义的 class 方法。

7.3.2 覆盖方法
Overriding Methods

当定义一个新类时，我们可以通过定义新的方法来增加行为。不过，作为一个同样重要的方式，通过重定义继承的方法，我们可以定制继承的行为。

例如，Object 类定义的 to_s 方法用一种非常泛化的方式把一个对象转化为字符串：

```
o = Object.new
puts o.to_s      # Prints something like "#<Object:0xb7f7fce4>"
```

当我们在 Point 类中定义 to_s 方法时，就是在覆盖从 Object 中继承的 to_s 方法。

对于面向对象编程和子类化，一个重要特性是当方法被调用时，它们会被动态查找，从而找到适合的方法或重定义的方法，也就是说，方法调用并非在解析时进行绑定，而是在运行时进行绑定的。下面的代码演示了这一重要特性：

```
# Greet the World
class WorldGreeter
  def greet                    # Display a greeting
    puts "#{greeting} #{who}"
  end

  def greeting                 # What greeting to use
    "Hello"
  end

  def who                      # Who to greet
    "World"
  end
```

注 1：如果你是一个 Java 程序员，这也许会让你感到吃惊。Java 类定义特殊的构造方法进行初始化，这些方法不会被继承。在 Ruby 中，initialize 方法只是一个普通方法，可以像其他方法一样被继承。

```
end

# Greet the world in Spanish
class SpanishWorldGreeter < WorldGreeter
  def greeting                # Override the greeting
    "Hola"
  end
end

# We call a method defined in WorldGreeter, which calls the overridden
# version of greeting in SpanishWorldGreeter, and prints "Hola World"
SpanishWorldGreeter.new.greet
```

如果原先尝试过面向对象编程，程序的这种特性很可能对你是不言而喻而且司空见惯的，不过如果是第一次接触这个概念，它可能显得很深奥。我们调用从 WorldGreeter 中继承来的 greet 方法时，这个 greet 方法调用 greeting 方法。在 greet 方法定义时，greeting 方法返回“Hello”。但是我们对 WorldGreeter 进行了子类化，而且调用 greet 的对象具有一个重新定义的 greeting 方法。当我们调用 greeting 时，Ruby 在被调用的对象上查找该方法的恰当定义，最后找到这个西班牙语版本的 greeting 方法，而不是英语版本的 greeting 方法。这种在运行时查找一个方法正确定义的方式称为**方法名解析**（*method name resolution*），在本章最后的第 7.8 节中将有详细描述。

我们还有很好的理由把一个类定义为**抽象**（*abstract*）类，这种类会调用未定义的“抽象”方法，这些方法会留待子类进行实现。抽象的反面是**具体**（*concrete*），如果一个继承自抽象类的类实现了祖先类中所有的抽象方法，它就是一个具体类。例如：

```
# This class is abstract; it doesn't define greeting or who
# No special syntax is required: any class that invokes methods that are
# intended for a subclass to implement is abstract.
class AbstractGreeter
  def greet
    puts "#{greeting} #{who}"
  end
end

# A concrete subclass
class WorldGreeter < AbstractGreeter
  def greeting; "Hello"; end
  def who; "World"; end
end

WorldGreeter.new.greet  # Displays "Hello World"
```

7.3.2.1 覆盖私有方法

私有方法不能在所定义的类外被调用，但是它们会被子类继承，这意味着子类可以调用它们，也可以覆盖它们。

在子类化一个别人写的类时需要特别小心：私有方法一般作为内部的辅助方法被使用，它们不是公开 API 的一部分，不被外界可见。如果没有读过那个类的源代码，你可能覆盖了它的私有方法而浑然不觉。如果碰巧定义了一个跟超类私有方法同名的方法（不管是以哪种可见性），你就无意中覆盖了超类的内部工具方法，这几乎总会导致不可预见的行为。

其结果是，在 Ruby 中，应该只对你了解的超类进行子类化。如果希望使用某个类的公开 API 而非它的具体实现，应该通过封装（其实例）和代理来扩展它的功能，而不是继承它。

7.3.3 通过链式调用来扩展功能
Augmenting Behavior by Chaining

有时，覆盖一个方法时，我们并不想完全取代它，而只想增加一些新代码对它的行为进行扩展。要做到这点，我们要在覆盖的方法中调用被覆盖的方法，这被称为**“链式调用（*chaining*）”**，它通过 `super` 关键字实现。

`super` 就像是一个特殊的方法调用：它调用超类中与当前方法同名的方法（注意超类并不一定要定义这个方法，它可以从它的祖先类中继承），可以像普通方法调用一样为 `super` 指定参数。方法链式调用一个常用且重要的场所是 `initialize` 方法，下面是我们为 `Point3D` 方法编写的 `initialize` 方法：

```
class Point3D < Point
  def initialize(x,y,z)
    # Pass our first two arguments along to the superclass initialize method
    super(x,y)
    # And deal with the third argument ourself
    @z = z;
  end
end
```

如果单独使用关键字 `super`——不带参数和圆括号——会使用所有当前方法的参数对超类的方法进行调用。不过要注意的是，传送给超类方法的参数是当前方法的参数，如果当前方法修改了参数变量的值，那么传给超类方法的是修改过的值。

就像普通方法调用一样，`super` 参数两端的圆括号是可选的。不过，因为仅使用 `super` 具有特殊含义，在不使用参数并且该方法在形式上具有一个或多个参数时，必须显式使用空的圆括号来表示不传递任何参数。

7.3.4 类方法的继承
Inheritance of Class Methods

类方法可以像实例方法一样被继承和覆盖。如果我们的 Point 类定义了一个类方法 sum，那么 Point3D 子类会继承这个方法，也就是说，如果 Point3D 不自己定义一个名为 sum 的类方法，那么表达式 Point3D.sum 与 Point.sum 将调用同一个方法。

出于编程风格的考虑，最好还是通过定义时所在的类来调用类方法。一个代码维护者在看到表达式 Point3D.sum 时会在 Point3D 类中寻找 sum 方法的定义，他可能会花不少时间才能发现原来这个方法是在 Point 类中定义的。在明确指明接收者的情况下调用类方法时，要尽量避免依赖继承——应该总是通过定义时所在的类来调用类方法（注 2）。

在类方法的定义中，可以不明确指明接收者就调用其他类方法。在这种情况下，它的接收者隐式地指向 self，在类方法中，self 的值就是它所在的类。在这里——在类方法的定义中——类方法的继承就很有用了：它可以隐式调用一个类方法，即使这个类方法定义在它的超类中。

最后，注意类方法也可以像实例方法一样使用 super，用来调用超类中的同名方法。

7.3.5 继承和实例变量
Inheritance and Instance Variables

在 Ruby 中，实例变量常常看起来仿佛可以被继承。例如，考虑如下代码：

```
class Point3D < Point
  def initialize(x,y,z)
    super(x,y)
    @z = z;
  end

  def to_s
    "(#@x, #@y, #@z)"  # Variables @x and @y inherited?
  end
end
```

Point3D 的 to_s 方法引用了超类 Point 中的@x 和@y 变量，这段代码的结果很可能正如你所料：

```
Point3D.new(1,2,3).to_s  # => "(1, 2, 3)"
```

正因如此，你可能想说这些变量被继承了，不过 Ruby 不是这样工作的。所有 Ruby 对象都包含一组实例变量，这些变量不是在相应的类中被定义的，它们只是在被赋值时才被创建

注 2：Class.new 方法是一个例外——它应该在新创建的类上被调用。

出来。因为实例变量并非在类中被定义，所以它们与继承和子类化机制无关。

在上面的代码中，Point3D 定义的 initialize 方法链式调用了超类中的 initialize 方法。这个被链入的方法对@x 和@y 变量进行了赋值，这使得这些变量出现在这个特定的 Point3D 实例中。

从 Java 语言（或其他强类型的语言，在类的定义中会为其实例定义一组字段）转过来的程序员可能要花费一些时间来习惯它。不过，其实这非常简单：Ruby 的实例变量不会被继承，它们与继承机制毫无关系。有时它们看起来好像被继承的原因是，方法在第一次赋值时创建它们，而这些**方法**经常是继承而来的并进行了链式调用。

这里有一个重要的推论。因为实例变量与继承无关，所以子类中使用的实例变量不会“遮蔽（shadow）”超类中的实例变量。如果子类使用的某个实例变量与祖先类中的某个实例变量同名，它会覆盖祖先类中该变量的值，这既可能是有意为之的（改变父类的行为），也有可能是在无意中实现的。如果是后者，那么肯定会引入 bug。在讨论私有方法时我们已经提到应该仅仅扩展你熟悉的（可以控制的）的超类，这里又为这个论点增加了一个理由。

最后，前面说过类实例变量不过是 Class 对象的实例变量，因此，它们也不会被继承。进一步说，Point 和 Point3D 对象（在此它们作为 Class 的对象，而非它们代表的类）仅仅是 Class 类的实例，它们没有什么联系，因此不可能从对方中继承任何变量。

7.3.6 继承和类变量

Inheritance and Class Variables

类变量可以被该类及其子类所共享。如果一个类 A 定义了一个@@a 变量，那么子类 B 可以使用这个变量，表面看起来像继承了这个变量，其实还是有区别的。

设想为类变量赋值的情形会让这区别变得清楚些。如果子类为一个已在超类中使用的类变量赋值，它不会创建这个类变量的私有拷贝，而是直接修改父类中看到的那个变量，同时所有其他子类也会看到这个被修改的值。如果运行时使用-w 选项，Ruby 1.8 会对这种做法发出一个警告，不过 Ruby 1.9 不会。

如果一个类使用了类变量，它的所有子类和后代类都可以通过修改这个变量来改变这个类的行为。这也是使用类实例变量而非类变量的一个有力论据。

下面的代码演示了类变量的共享性，它的运行结果是输出 123：

```
class A
  @@value = 1                  # A class variable
  def A.value; @@value; end    # An accessor method for it
end
print A.value                  # Display value of A's class variable
class B < A; @@value = 2; end  # Subclass alters shared class variable
print A.value                  # Superclass sees altered value
class C < A; @@value = 3; end  # Another alters shared variable again
print B.value                  # 1st subclass sees value from 2nd subclass
```

7.3.7 常量的继承
Inheritance of Constants

常量与实例方法相似，可以被继承和覆盖。不过，常量在继承方面与方法有一个重要的差别。

例如，我们的 Point3D 类可以使用 Point 超类中定义的 ORIGIN 常量，尽管最好使用常量定义的类作为限定名来使用这个这个常量，Point3D 也可以用 ORIGIN 这样的非限定方式来使用它，甚至还可以使用 Point3D::ORIGIN 这样的方式。

当子类（比如 Point3D）重定义了一个常量时，常量的继承会变得很有趣。一个三维点的类很可能会希望定义名为 ORIGIN 的常量来代表一个三维点，因此 Point3D 可能会有如下代码：

```
ORIGIN = Point3D.new(0,0,0)
```

如你所知，Ruby 将在常量被重定义时发出警告，不过在本例中，这是一个新创建的常量。现在我们有了两个常量：Point::ORIGIN 和 Point3D::ORIGIN。

常量和方法的重要区别在于，常量首先在其句法空间中进行查找，然后才在继承体系中查找（第 7.9 节中有其细节），这意味着如果 Point3D 中继承的方法使用了 ORIGIN 常量，那么即使 Point3D 定义了自己的 ORIGIN 常量，这些方法的行为也不会改变。

7.4 对象创建和初始化
Object Creation and Initialization

在 Ruby 中，对象通常通过类的 new 方法来创建。本节将详细解释这是如何发生的，也会解释其他创建对象的机制（比如克隆（cloning）和反序列化（unmarshaling））。在每个子小节中，将说明如何定制这些新创建对象的初始化。

7.4.1 new、allocate 和 initialize

new, allocate, and initialize

每个类都继承了 new 这个类方法。这个方法要进行两项工作：它必须分配一个新对象，即让这个对象存在，并初始化这个对象。这两项工作，它分别交给 allocate 和 initialize 方法来实现。如果 Ruby 的 new 方法使用 Ruby 来实现，它应该看起来如下：

```
def new(*args)
  o = self.allocate   # Create a new object of this class
  o.initialize(*args) # Call the object's initialize method with our args
  o                   # Return new object; ignore return value of initialize
end
```

allocate 是 Class 类的一个实例方法，被所有类对象所继承，它的作用在于创建类的一个实例。你可以用这个方法创建一个未初始化的实例。不过不要试图覆盖它，Ruby 总是忽略那些覆盖的版本，而直接调用原始版本。

initialize 是一个实例方法，绝大多数的类都需要这个方法。每个不是从 Object 继承而来的类都应该使用 super 来链式调用超类的 initialize 方法，initialize 方法的一般性工作就是创建该对象的实例变量并为它们赋初始值。通常，客户端代码为 new 方法传递参数，这些参数将再次被传给 initialize 方法，然后赋值给这些实例变量。initialize 方法无须返回一个初始化后的对象，实际上，initialize 方法的返回值被忽略。Ruby 让 initialize 方法隐式成为私有方法，这意味着你不能显式对它进行调用。

Class::new 和 Class#new

Class 类定义了两个名为 new 的方法。一个是 Class#new，它是一个实例方法；另一个是 Class::new，它是一个类方法（我们用 *ri* 中的命名习惯来消除歧义）。我们在前面已经介绍过第一个方法，它被所有类对象继承而变成一个类方法，用于创建和初始化新的实例。

Class::new 是 Class 类自身的一个方法，用于创建新的类。

7.4.2 工厂方法

Factory Methods

让类实例以多种方式进行初始化通常是有用的，一般可以通过对 initialize 方法使用默认参数来实现。比如，通过下面定义的 initialize 方法，可以用 2 个或 3 个参数来调用 new 方法：

```
class Point
  # Initialize a Point with two or three coordinates
  def initialize(x, y, z=nil)
```

```
    @x,@y,@z = x, y, z
  end
end
```

不过，有时使用默认参数是不够的，我们须要使用工厂方法而不是 new 来创建类的实例。假设我们希望能用笛卡尔坐标系或极坐标系来初始化 Point 对象：

```
class Point
  # Define an initialize method as usual...
  def initialize(x,y)  # Expects Cartesian coordinates
    @x,@y = x,y
  end

  # But make the factory method new private
  private_class_method :new

  def Point.cartesian(x,y)    # Factory method for Cartesian coordinates
    new(x,y)  # We can still call new from other class methods
  end

  def Point.polar(r, theta)   # Factory method for polar coordinates
    new(r*Math.cos(theta), r*Math.sin(theta))
  end
end
```

上面的代码还是依赖于 new 和 initialize 方法，但是它把 new 变成了私有方法，因此 Point 的用户不能直接调用它，必须使用自定义的工厂方法来创建 Point 对象。

7.4.3 dup、clone 和 initialize_copy

dup, clone, and initialize_copy

另外一种产生新对象的方法是使用 dup 和 clone 方法（参见第 3.8.8 节）。这些方法分配一个调用者所属类的实例，然后把调用者的所有实例变量及修改（taintedness）都拷贝到新创建的对象中。clone 比 dup 的拷贝动作更彻底，它还拷贝调用者的单键方法，如果调用者处在冻结状态（frozen），那么这个状态也被拷贝到新创建的对象中。

如果一个类定义了一个名为 initialize_copy 的方法，那么 clone 和 dup 方法在拷贝完实例变量后，在新创建的对象上执行这个方法。（clone 在冻结拷贝之前调用 initialize_copy 方法，所以这个方法还可以修改拷贝对象。）initialize_copy 接受原始对象为参数，可以对拷贝对象做一些修订，不过，它不能用于创建自身的拷贝，initialize_copy 方法的返回值被忽略。像 initialize 方法一样，Ruby 总是保证 initialize_copy 方法是私有的。

当 clone 和 dup 方法把实例变量从原始对象拷贝到拷贝对象中时，它们拷贝的是引用而非

实际值。换句话说，它们用的是浅拷贝（shallow copy），这也是为什么很多类须要定制这些方法的一个原因。下面代码定义的 initialize_copy 方法则执行一个深拷贝：

```
class Point                    # A point in n-space
  def initialize(*coords)      # Accept an arbitrary # of coordinates
    @coords = coords           # Store the coordinates in an array
  end

  def initialize_copy(orig)    # If someone copies this Point object
    @coords = @coords.dup      # Make a copy of the coordinates array, too
  end
end
```

这里显示的类把自己的状态存放在一个数组中，如果没有这个 initialize_copy 方法，在使用 clone 或 dup 拷贝对象时，拷贝的对象与原始对象共享同一个状态数组。对拷贝对象所做的修改也会影响原始对象的状态，这不是我们所希望的结果，因此我们必须定义一个 initialize_copy 方法来为这个数组也创建一个拷贝。

一些类，比如定义枚举类型的类，可能会希望严格限制实例的个数。这样的类除了要把 new 方法设为私有外，还须要阻止拷贝动作。下面的代码演示了一种方法：

```
class Season
  NAMES = %w{ Spring Summer Autumn Winter }    # Array of season names
  INSTANCES = []                               # Array of Season objects

  def initialize(n)   # The state of a season is just its
    @n = n            # index in the NAMES and INSTANCES arrays
  end

  def to_s            # Return the name of a season
    NAMES[@n]
  end

  # This code creates instances of this class to represent the seasons
  # and defines constants to refer to those instances.
  # Note that we must do this after initialize is defined.
  NAMES.each_with_index do |name,index|
    instance = new(index)          # Create a new instance
    INSTANCES[index] = instance    # Save it in an array of instances
    const_set name, instance       # Define a constant to refer to it
  end

  # Now that we have created all the instances we'll ever need, we must
  # prevent any other instances from being created
  private_class_method :new,:allocate # Make the factory methods private
  private :dup, :clone                # Make copying methods private
end
```

这里的代码使用了一些元编程的技术，你可能在读完第 8 章后更容易理解它们。这段代码的要点是在最后一行把 dup 和 clone 方法设为私有。

另一种阻止拷贝对象的方法是使用 undef 语句简单删除 clone 和 dup 方法。不过更好的方式是重定义 clone 和 dup 方法，让它们抛出异常，在异常的消息中明确说明拷贝动作是不允许的。这个错误消息对使用该类的程序员很有帮助。

7.4.4 marshal_dump 和 marshal_load

marshal_dump and marshal_load

创建对象的第三种方式是调用Marshal.load方法来重新生成早先用Marshal.dump序列化的对象。Marshal.dump 方法保存一个对象的类，并递归序列化其中每个实例变量的值。这种方式挺有效，绝大多数对象都可以用这两个方法进行存储和恢复。

有些类须要修改实现序列化（和反序列化）的方式。这样做的一个理由是为对象状态提供更加紧凑的表示方式；另一个理由是避免保存一些易变（volatile）数据（比如缓存中的内容），它们应该在反序列化时被清除。可以通过定义实例方法 marshal_dump 来定制序列化方式，它返回一个被序列化的对象（比如字符串或一些选定的实例变量值的数组）给调用者。

如果自定义了一个 marshal_dump 方法，必须同时定义一个对应的 marshal_load 方法。marshal_load 被一个新分配的（用 allocate 方法）但是尚未初始化的对象所调用，它需要一个由 marshal_dump 返回的可再生的对象拷贝作为参数，然后根据这个参数对象的状态初始化接收者对象。

作为例子，让我们返回一个前面定义的多维 Point 类。如果我们限制所有的坐标必须为整数，那么就可以通过把一组整数坐标打包成一个字符串来节约一些存储空间（也许需要用 *ri* 来读一下 Array.pack 的文档以帮助理解下面的代码）：

```
class Point                    # A point in n-space
  def initialize(*coords)      # Accept an arbitrary # of coordinates
    @coords = coords           # Store the coordinates in an array
  end

  def marshal_dump             # Pack coords into a string and marshal that
    @coords.pack("w*")
  end

  def marshal_load(s)          # Unpack coords from unmarshaled string
    @coords = s.unpack("w*")   # and use them to initialize the object
  end
end
```

如果对一个类禁用了 clone 和 dup 方法——比如前面定义的 *Season* 类——你可能须要定制序列化方法，因为通过序列化和反序列化一个对象可以很容易实现对象的拷贝，可以对 marshal_dump 和 marshal_load 方法进行重定义，让它们抛出异常，从而完全阻止序列化。不过这种方式太强硬了，一个更优雅的方式是定制反序列化方法，让 Marshal.load 返回一个已存在的对象而不是创建一个拷贝。

要做到这点，因为 marshal_load 的返回值会被忽略，我们必须定义另外一对序列化方法。_dump 是一个实例方法，它必须返回一个字符串表示对象的状态。相应的_load 方法是一个类方法，它接受_dump 返回的字符串作为参数，并返回一个对象。_load 方法可以创建一个新对象，也可以仅仅返回一个现有对象的引用。

为了允许序列化但禁止拷贝，对于 Season 对象，我们可以增加如下方法：

```
class Season
  # We want to allow Season objects to be marshaled, but we don't
  # want new instances to be created when they are unmarshaled.
  def _dump(limit)            # Custom marshaling method
    @n.to_s                   # Return index as a string
  end

  def self._load(s)           # Custom unmarshaling method
    INSTANCES[Integer(s)]     # Return an existing instance
  end
end
```

7.4.5 单键模式
The Singleton Pattern

单键是指仅仅拥有一个实例的类。在面向对象框架中，单键经常用于存储全局程序状态，它可以用来替代类方法和类变量。

> **术语“单键”**
>
> 本节讨论的是在面向对象编程领域中一个著名的设计模式——单键模式。在 Ruby 中，我们须要注意单键（singleton）这个术语，因为它被重载了。给某个对象添加一个方法，但并非给这个类增加一个方法，这种方式被称为**单键方法**（参见 6.1.4 节）。具有这种方法的类对象有时被称为**单键类**（尽管在本书中使用的是 *eigenclass* 这个术语，参见 7.7 节）。

要恰当实现单键模式，须要使用前面演示的若干技巧，new 和 allocate 方法必须被设为私有，必须阻止 dup 和 clone 产生新的拷贝，等等。幸运的是，在标准库中有一个 Singleton 模块为我们做了这一切，我们只需要 require 'singleton'并在自己定义的类中使用

include Singleton 即可。这会定义一个名为 instance 的类方法，它不带参数并返回该类的单个实例。我们应该定义一个 initialize 方法来执行这个单实例的初始化，不过要注意的是，这个方法不能带任何参数。

作为例子，让我们重新审视本章开始时统计所创建点对象个数的问题。现在我们不把这些数据存储在 Point 类的类变量中，而使用一个 PointStats 的单键实例：

```
require 'singleton'           # Singleton module is not built-in

class PointStats              # Define a class
  include Singleton           # Make it a singleton

  def initialize              # A normal initialization method
    @n, @totalX, @totalY = 0, 0.0, 0.0
  end

  def record(point)           # Record a new point
    @n += 1
    @totalX += point.x
    @totalY += point.y
  end

  def report                  # Report point statistics
    puts "Number of points created: #@n"
    puts "Average X coordinate: #{@totalX/@n}"
    puts "Average Y coordinate: #{@totalY/@n}"
  end
end
```

有了这样一个类，我们可以改写 Point 类的 initialize 方法如下：

```
def initialize(x,y)
  @x,@y = x,y
  PointStats.instance.record(self)
end
```

Singleton 模块自动创建 instance 类方法，我们可以在这个单键实例上调用 record 这个实例方法。相似地，如果我们想查询统计数据，可以用：

```
PointStats.instance.report
```

7.5 模块
Modules

与类相似，模块也是方法、常量和类变量的命名组，模块的定义跟类非常相似，只是用

module 关键字取代 class 关键字即可。不过与类不同，模块不能被实例化，也不能被子类化。模块是独立的，在继承体系中没有任何所谓的“模块等级”。

模块可以作为命名空间和混入（mixin）使用，下面的子小节会解释这两种用法。

就像类对象是 Class 类的实例一样，一个模块对象也是 Module 类的一个实例，这意味着所有的类都是模块，不过并非所有的模块都是类。类可以像模块一样作为命名空间使用，但是不能用于混入。

7.5.1 模块用于命名空间
Modules as Namespaces

在不须要使用面向对象编程时，模块是组织相关方法一个好方法。例如，假设你在编写时使用 Base64 编码方式的编码（encode）和解码（decode）方法，用于把文本编码为二进制数据及从二进制反转回文本，在这里无须定义一个编码器（encoder）和解码器（decoder）对象，因此没有理由为它们定义一个类。我们所需的只是两个方法：一个用于编码，一个用于解码。我们可以像下面这样定义两个全局方法：

```
def base64_encode
end

def base64_decode
end
```

为了防止和其他编码、解码方法命名冲突，我们使用 base64 作为方法名的前缀。这种方案是可行的，不过绝大多数程序员都选择避免把方法加入全局命名空间中，因此，一个更好的方案是把这两个方式放入一个 Base64 模块中：

```
module Base64
  def self.encode
  end

  def self.decode
  end
end
```

注意我们用 self.作为方法名的前缀，因此这些方法是这个模块的“类方法”。我们也可以显式使用模块名来定义这些方法：

```
module Base64
  def Base64.encode
  end

  def Base64.decode
  end
end
```

这样定义方法有点重复，不过它更好地表现出使用这些方法的句法：

```
# This is how we invoke the methods of the Base64 module
text = Base64.encode(data)
data = Base64.decode(text)
```


注意跟类名一样，模块名也用大写字母开头。定义一个模块会创建一个跟模块同名的常量，这个常量的值是一个 Module 对象，用于代表那个模块。

模块中可以包含常量。我们的 Base64 模块实现定义了一个常量，用来存放在 Base64 编码中使用的 64 个字符：

```
module Base64
  DIGITS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' \
           'abcdefghijklmnopqrstuvwxyz' \
           '0123456789+/'
end
```

在 Base64 模块外，须要使用 Base64::DIGITS 对这个常量进行引用。而在模块内，encode 和 decode 方法则只使用 DIGITS 就可以了。如果这两个方法想共享一些非常量的数据，可以像在类中一样使用类变量（用@@做前缀）。

7.5.1.1 嵌套命名空间

模块（也包括类）是可以嵌套的，这除了会产生嵌套命名空间外，没有其他副作用：一个嵌套在其他模块（或类）中的模块（或类）对所嵌套的模块（或类）没有任何特殊的访问方式。仍然使用 Base64 这个例子，假设我们想定义特殊的类分别用于编码和解码，因为 Encoder 和 Decoder 类是相关的，我们想把它们定义在一个模块内：

```
module Base64
  DIGITS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'

  class Encoder
    def encode
    end
  end

  class Decoder
    def decode
    end
  end

  # A utility function for use by both classes
  def Base64.helper
  end
end
```

用这样的结构组织代码，我们得到两个新类：Base64::Encoder 和 Base64::Decoder。在

Base64 模块内部，这两个类可以互相使用非限定名进行访问，无须加 Base64 前缀，而且也可以不加前缀，直接使用 DIGITS 常量。

另一方面，让我们看看 Base64.helper 这个工具方法。嵌套在模块中的 Encoder 和 Decoder 类对于该模块中的方法没有特别的访问方式，它们只能用全限定名 Base64.helper 来访问 helper 方法。

因为类是模块，所以它们都可以被嵌套。在一个类中嵌套另外一个类，只影响内部类的命名空间，而不会给内部类对外部类的方法和变量的访问带来任何特殊之处。如果代码需要一个辅助类（helper class）、代理类（proxy class）或任何不属于公开 API 的一部分的类，你可能想用嵌套类的方式把它用内部类实现。这可以让命名空间十分整洁，不过不能让被嵌套的类成为私有类。

参见第 7.9 节，可以看到在嵌套模块中，常量名是如何被解析的。

7.5.2 模块用于混入
Modules As Mixins

模块的第二种用途比第一种更为强大。如果一个模块定义了实例方法而非类方法，这些实例方法可以混入其他类中。Enumerable 和 Comparable 是两个著名的混入模块。Enumerable 模块定义了若干有用的迭代器，它们都基于 each 迭代器。Enumerable 模块自身并没有定义 each 迭代器，但是如果混入 Enumerable 模块的类定义了 each 迭代器，它就立刻具有这些强大的迭代器。Comparable 也与此类似，它定义了一组比较操作符，后者都是基于通用比较符<=>的，如果你的类定义了<=>，那么混入 Comparable 模块后，它就能自动拥有<、<=、==、>、 >= 和 between?这些操作符和方法。

想把一个模块混入一个类中，我们须要使用 include。include 通常像一个关键字一样被使用：

```
class Point
  include Comparable
end
```

但实际上，它不过是 Module 类的一个私有实例方法，它隐式地被 self 调用——在这里就是包含模块的类。如果使用方法的形式，上面的代码变为：

```
class Point
  include(Comparable)
end
```

include 是私有方法，它必须以函数形式被调用，而不能写成 self.include(Comparable)。include 方法可以接受任意多的 Module 对象进行混入，因此一个定义了 each 和<=>的类可以加入下面的代码：

```
include Enumerable, Comparable
```

包含一个模块会影响 is_a? 这个类型检查方法及条件比较符===。例如，在 Ruby 1.8 中，String 类混入了 Comparable 模块和 Enumerable 模块：

```
"text".is_a? Comparable       # => true
Enumerable === "text"         # => true in Ruby 1.8, false in 1.9
```

注意 instanceof?只对比接收者的类，而不管其超类或模块，所以下面的代码会返回 false：

```
"text".instance_of? Comparable  # => false
```

尽管每个类都是模块，但是 include 方法不允许把一个类包含在另外一个类中，它的参数必须是用 module 进行声明的模块，而不能是类。

不过，把一个模块包含在另外一个模块中则是合法的，这样做让被包含模块的实例方法成为该模块中的实例方法。作为例子，考虑第 5 章的这段代码：

```
module Iterable      # Classes that define next can include this module
  include Enumerable          # Define iterators on top of each
  def each                    # And define each on top of next
    loop { yield self.next }
  end
end
```

混入一个模块的普通方式是使用 Module.include 方法，另外一种方法则是使用 Object.extend 方法，这个方法使指定模块的实例方法成为接收对象的单键方法。(如果接收者对象是一个 Class 实例，那么这些方法就成为那个类的类方法。) 下面是一个例子：

```
countdown = Object.new        # A plain old object
def countdown.each            # The each iterator as a singleton method
  yield 3
  yield 2
  yield 1
end
countdown.extend(Enumerable)  # Now the object has all Enumerable methods
print countdown.sort          # Prints "[1, 2, 3]"
```

7.5.3 可包含的命名空间模块
Includable Namespace Modules

我们也可以这样定义模块：让它们不仅定义命名空间，而且它们的方法也能被混入。Math 模块就是这样一个例子：

```
Math.sin(0)       # => 0.0: Math is a namespace
include 'Math'    # The Math namespace can be included
sin(0)            # => 0.0: Now we have easy access to the functions
```

Kernel 模块也是这样的例子：我们可以通过 Kernel 命名空间来调用它的方法，或者把这些方法当作 Object 的私有方法进行调用，因为 Kernel 模块已混入 Object 类中。

如果想创建像 Math 或 Kernel 这样的模块，需要把模块的方法定义为实例方法，然后用 module_function 把这些方法转换为“模块函数（module functions.）”。module_function 方法与 public、private 和 protected 方法相似，都是 Module 类的私有实例方法，它接受任意数量的方法名（用符号或字符串形式）作为参数。调用它的主要效果是对给定方法创建类方法的拷贝，另一个效果是把那些实例方法变为私有的（关于这点，马上还将讲到更多内容）。

像 public、private 和 protected 方法一样，module_function 方法也可以用无参方式调用。用这种方式调用时，后面定义的模块的实例方法都会成为模块函数：它们成为公开的类方法及私有的实例方法。一旦用无参的方式调用 module_function，它对其余所有的模块定义部分生效，因此如果想把某些方法定义为非模块函数的方法时，应该把它们定义在前面。

起初，看到 module_function 把实例方法变为模块私有方法时，可能会让人有点意外。这样做的原因并非出于访问控制的考虑，因为显然这些方法还是可以通过模块的命名空间进行访问的；相反，这样做的原因是想让这些方法必须用无接收者的函数风格的调用方式。（称呼它们为模块函数而非模块方法的原因就是它们必须以函数风格调用。）强制被包含模块的方法以无接收者的方式调用，这是为了减少与真正的实例方法混淆的可能性。假设我们定义了一个类，它有很多用于处理三角问题的方法。出于方便，我们包含了 Math 模块，这样我们可以直接使用 sin 方法（而不用 Math.sin）。sin 方法被 self 隐式调用，但是我们不想让它对 self 对象做任何操作。

在定义一个模块函数时，应该避免使用 self，因为它的值依赖于你如何调用它。当然可以定义一个模块函数，让它的行为依赖于调用方式。不过如果你想这么做，把这个方法定义为一个类方法和一个实例方法会更合理。

7.6 加载和请求模块
Loading and Requiring Modules

Ruby 程序可以被分散在多个文件中，分割程序最自然的方式是对每一个非平凡类或模块都单独使用一个文件。这些分离的文件可以使用 require 或 load 重组为一个单个程序（并且，如果设计得当，该程序也可以被其他程序复用）。这两个方法是定义在 Kernel 中的全局函数，但是可以像关键字一样被使用。require 方法还可以用于从标准库中加载文件。

load 和 require 的作用相似，不过 require 要比 load 常用得多，两个函数都可以加载并执行指定的 Ruby 源代码文件。如果指定加载的文件使用一个绝对路径或相对于~（用户的

主目录）的路径，那么指定的文件被加载。不过，通常情况下会指定一个相对路径，load和 require 在 Ruby 加载路径（后面会详述加载路径）的目录中做相对路径查找。

尽管从整体上来说很类似，它们还是有重要的区别。

- 除了加载源代码，require 还可以加载 Ruby 的二进制扩展。当然，二进制扩展是与实现相关的，不过在基于 C 的实现中，这些共享库的文件名一般都以*.so* 或*.dll* 做后缀名。
- load 方法的参数要求是包括扩展名的完整文件名，而 require 则通常只要求传入一个库的名字，不需要像文件名那样的后缀；此时，require 命令根据库名寻找一个合适的源文件或本地扩展文件。如果一个目录下对同样的库名既有*.rb* 结尾的源文件，又有二进制扩展的文件，require 命令会加载那个源文件而非二进制文件。
- load 把一个文件加载多次，而 require 则试图避免对同一文件的多次加载。（不过，如果使用两个不同但是等价的路径来 require 同一个库， require 方法会被愚弄。在 Ruby 1.9 中，require 把相对路径展开为绝对路径，这使得它更难被愚弄。）require 把加载过的文件名放入全局数组$"（也被称为$LOADED_FEATURES）中，从而可以追踪加载的库。load 则不这么做。
- load 在当前$SAFE 级别上加载指定文件，而 require 则使用$SAFE=0 对指定库进行加载，即使调用 require 的代码对此变量有大于 0 的值。参见 10.5 节可以获得$SAFE 及 Ruby 安全性的更多知识。（注意如果$SAFE 的值大于 0，require 拒绝加载任何文件名被修改的或任何来自公开可写目录的文件，因此，在理论上，用降低的$SAFE 级别来调用 require 是安全的。）

下面的子小节会进一步探讨 load 和 require 的行为。

7.6.1 加载路径
The Load Path

Ruby 的加载路径是一个数组，可以通过全局变量$LOAD_PATH or $:（可以这样来记住这个全局变量名：冒号在类 Unix 操作系统中，用做路径的分隔符）进行访问。数组的每个元素都是一个目录名，Ruby 在这些目录中查找加载文件，前面的目录比后面的目录优先被查找。在 Ruby 1.8 中，数组中的元素必须是字符串；而在 Ruby 1.9 中，它们既可以是字符串，也可以是任意实现了 to_path 方法（该方法返回字符串）的对象。

$LOAD_PATH 的默认值取决于 Ruby 的实现、运行的操作系统及 Ruby 的安装路径。下面是它在 Ruby 1.8 中的典型值，使用 ruby -e 'puts $:'获得：

```
/usr/lib/site_ruby/1.8
/usr/lib/site_ruby/1.8/i386-linux
/usr/lib/site_ruby
/usr/lib/ruby/1.8
/usr/lib/ruby/1.8/i386-linux
.
```

/usr/lib/ruby/1.8/ 目录是 Ruby 标准库安装的路径，*/usr/lib/ruby/1.8/i386-linux/*目录中包含 Linux 下对标准库的二进制扩展，*site_ruby* 目录中存放安装的跟本机相关的库。注意 site_ruby 目录被首先查找，这意味着可以用这里的库覆盖 Ruby 的标准库。当前工作路径"." 在搜索路径的最后，它是用户运行 Ruby 程序时所在的目录，与 Ruby 程序所放置的路径是不同的。

在 Ruby 1.9 中，默认的加载路径更加复杂。下面是一个典型的例子：

```
/usr/local/lib/ruby/gems/1.9/gems/rake-0.7.3/lib
/usr/local/lib/ruby/gems/1.9/gems/rake-0.7.3/bin
/usr/local/lib/ruby/site_ruby/1.9
/usr/local/lib/ruby/site_ruby/1.9/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/vendor_ruby/1.9
/usr/local/lib/ruby/vendor_ruby/1.9/i686-linux
/usr/local/lib/ruby/vendor_ruby
/usr/local/lib/ruby/1.9
/usr/local/lib/ruby/1.9/i686-linux
.
```

一个小变化是 Ruby 1.9 在加载路径中加入了 *vendor_ruby* 目录，它在 *site_ruby* 之后而在标准库之前被搜索。这个目录可以供操作系统供应商进行定制。

在 Ruby 1.9 中，更显著的变化是加入了 RubyGems 的安装路径。在上面显示的路径中，搜索路径的前两个目录是用 *gem* 命令（RubyGems 包管理系统的命令）安装的 *rake* 包的路径。在上面的例子中，仅仅安装了一个 gem 包，不过如果你的系统中安装了很多 gem 包，默认的安装路径列表会变得很长。（在运行不需要 gem 的程序时，可以使用`--disable-gems`这个命令行参数，它让程序的启动速度有稍许提高。）如果一个 gem 的多个版本同时被安装在系统中，最高版本的 gem 会出现在默认加载路径中，可以使用 `Kernel.gem` 方法来修改这个默认行为。

在 Ruby 1.9 中内置了 RubyGems，*gem* 命令随 Ruby 分发，可以用来安装新的包，而且安装包的路径自动加到默认加载路径中。在 Ruby 1.8 中，RubyGems 必须被单独安装（尽管一些 Ruby 1.8 的分发包会自动捆绑它），而且 gem 安装的路径不会添加到加载路径中。相反，

Ruby 1.8 程序请求(require)rubygems 模块，这样会替换默认的 require 方法，新的 require 方法知道怎样查找安装的 gem 包。参见第 1.2.5 节，可以获取更多关于 RubyGems 的知识。

可以用-I 这个命令行参数告诉 Ruby 解释器把后面的目录加到搜索路径的前部。多次使用-I 参数可以指定多个目录，也可以在一个参数后指定多个目录，每个目录用冒号分隔（在 Windows 下是分号）。

Ruby 程序也可以通过修改$LOAD_PATH 数组的内容来修改加载路径，下面是一个例子：

```
# Remove the current directory from the load path
$:.pop if $:.last == '.'

# Add the installation directory for the current program to
# the beginning of the load path
$LOAD_PATH.unshift File.expand_path($PROGRAM_NAME)

# Add the value of an environment variable to the end of the path
$LOAD_PATH << ENV['MY_LIBRARY_DIRECTORY']
```

最后，要记住你可以对 load 或 require 使用绝对路径（用/或~打头），这样可以完全绕过加载路径。

7.6.2 执行加载的代码
Executing Loaded Code

load 和 require 会立刻执行指定文件中的代码，不过，这种执行方式与直接调用文件中的代码并不等价（注 3）。

用 load 或 require 加载的文件在顶级范围中被执行，而不是在 load 或 require 被调用的等级中被执行。被加载的文件可以访问那些在加载时已定义的所有全局变量和常量，但不能访问其初始化时所在的局部范围。这意味着：

- 调用 load 和 require 的代码所定义的局部变量不能被加载的文件访问；
- 被加载文件所定义的局部变量在加载完成后会被废弃，在该文件外这些变量不能被访问；
- 在被加载文件的开始处，self 的值永远是主对象（main object），这与 Ruby 解释器在开始运行时是一样的，也就是说，调用 load 或 require 的方法不会把接收者对象传送给所加载的文件；

注 3：可以用另一种方式为 C 程序员解释：load 和 require 与 C 语言的#include 指令不同。#include 一个文件的作用与在 Ruby 中用 eval 方法执行一个文件中所有解析出的代码相似：eval(File.read(filename))。但是即使如此，它们也还是有差别的，eval 方法不会设置局部变量。

- 在加载的文件中，当前的模块嵌套被忽略，例如，不能打开某个类然后加载一组方法定义，加载的文件在顶级范围内被处理，而不是在某个类或模块中。

7.6.2.1 包裹式加载

load 方法有一个不大常用的特性，因此在前面我们没有对之进行介绍。如果在调用时给出第二个参数并且其值不是 nil 或 false，它会“包裹（wrap）”给定的文件并加载到一个匿名模块中，这意味着加载的文件不会影响全局命名空间，它命名的所有常量（包括类和模块）被放入这个匿名模块中。我们可以把这种包裹的加载方式作为一种预防性的安全措施（或者作为减少命名空间冲突的 bug 的一种方式）。在第 10.5 节中，当 Ruby 在“沙箱”中运行非信任代码时，这些代码只能运行这种方式的 load，而不能使用 require。

当一个文件被加载到一个匿名模块中，它仍然可以设置全局变量，而且这些变量也可以被加载的代码所使用。假设你写了一个 *util.rb* 文件，它定义了一个有很多工具方法的 Util 模块，如果希望这些方法即使在用包裹方式加载时仍可使用，你可以在文件尾部加入如下代码：

```
$Util = Util   # Store a reference to this module in a global variable
```

现在，把 *util.rb* 加载到匿名空间中的代码也可以访问这些工具方法了，不过不是使用常量 Util，而是使用全局变量$Util。

在 Ruby 1.8 中，甚至可以把匿名模块自身传回给加载的代码：

```
if Module.nesting.size > 0        # If we're loaded into a wrapper module
  $wrapper = Module.nesting[0]    # Pass the module back to the loading code
end
```

参见第 8.1.1 节，可以查看更多 Module.nesting 的细节。

7.6.3 自动加载模块
Autoloading Modules

Kernel 和 Module 的 autoload 方法支持一种按需惰性加载机制。全局 autoload 函数允许你注册一个未定义的常量（通常是类或模块名）及一个定义了该常量的包名，当这个常量第一次被引用时，那个注册的包就使用 require 进行加载。例如：

```
# Require 'socket' if and when the TCPSocket is first used
autoload :TCPSocket, "socket"
```

Module 类定义了自己的 autoload 方法，它可以注册嵌套模块中定义的常量。

使用 autoload?或 Module.autoload?方法可以测试一个常量是否加载一个文件，它们带有一个符号参数。如果这个符号被引用时加载了一个文件，autoload?方法返回这个文件名；否则（没有使用自动加载，或者这个文件已经被加载），autoload?方法返回 nil。

7.7 单键方法和 Eigenclass

Singleton Methods and the Eigenclass

在第 6 章中，我们看到可以定义单键方法——只对某个对象而非一类对象有效的方法。为了给一个对象 Point 定义 sum 方法，可以用如下方式：

```
def Point.sum
  # Method body goes here
end
```

就像本章前面所提到的，类方法无非就是代表该类的 Class 实例上的单键方法。

一个对象的单键方法并没有被它的类所定义，但是它们也是方法，必须与某个类产生某种联系。一个对象的单键方法是它关联的匿名 *eigenclass* 的实例方法。“Eigen”是一个德语单词，大致相当于“自我（self）”、“自身的（own）”、“特定的（particular to）”或“特有的（characteristic of）”。eigenclass 也被称为**单键类**或**元类**（*metaclass*，它被使用得比较少）。“eigenclass”并没有被 Ruby 社区普遍接受，不过在本书中使用这个术语。

Ruby 有一个句法用来打开一个对象的 eigenclass 并向其中加入方法，这样可以避免逐个定义单键方法，而是一次为 eigenclass 定义多个单键方法。为了打开对象 o 的 eigenclass，我们可以使用 class << o。例如，可以用如下方法为 Point 定义类方法：

```
class << Point
  def class_method1       # This is an instance method of the eigenclass.
  end                     # It is also a class method of Point.

  def class_method2
  end
end
```

如果在 class 定义内部打开这个类对象的 eigenclass，我们可以用 self 来避免重复类名。下面重复本章早先的一个例子：

```
class Point
  # instance methods go here

  class << self
    # class methods go here as instance methods of the eigenclass
  end
end
```

要小心使用这些句法。注意，下面这 3 行代码有显著差别：

```
class Point                # Create or open the class Point
class Point3D < Point      # Create a subclass of Point
class << Point             # Open the eigenclass of the object Point
```

一般而言，用独立的单键方法定义比打开 eigenclass 定义单键方法更清晰。

在打开一个对象的 eigenclass 时，self 指向 eigenclass 对象，因此一般用如下方式获得对象 o 的 eigenclass：

```
eigenclass = class << o; self; end
```

我们可以把上面的代码形式化为 `Object` 的一个方法，这样就可以请求任意对象的 eigenclass：

```
class Object
  def eigenclass
    class << self; self; end
  end
end
```

除非你在进行某些高级的元编程，否则不需要像上面那样定义的 `eigenclass` 工具方法。不过，了解 eigenclass 还是值得的，不仅因为在现有代码中可以偶尔看到它们，也因为它们在 Ruby 的方法名解析算法中有重要作用，我们马上就会讲到这点。

7.8 方法查找 Method Lookup

当 Ruby 为一个方法调用表达式求值时，它首先必须找出哪个方法被调用，这个过程被称为**方法查找**（*method lookup*）或**方法名解析**（*method name resolution*）。对于方法调用表达式 `o.m`，Ruby 使用如下步骤进行名字解析。

1. 首先，检查 o 的 eigenclass 是否有名为 m 的单键方法。
2. 如果在 eigenclass 中没有方法 m，在 o 的类中寻找名为 m 的实例方法。
3. 如果在类中没有找到 m 方法，Ruby 在类所包含的所有模块中查找名为 m 的实例方法。如果类包含的模块不止一个，它们以包含顺序的逆序被查找。也就是说，最后包含的模块将第一个被查找。
4. 如果在类中和包含的模块中都找不到实例方法 m，Ruly 会向上到超类中去查找，这样将对每个祖先类和它们包含的模块重复进行第二步和第三步的操作。

5. 如果进行了这样的查找还是没有找到一个名为 m 的方法，就会调用一个名为 method_missing 的方法。为了找到该方法的适合定义，Ruby 将从第一步开始执行这个名字解析的算法。Kernel 模块为 method_missing 方法提供了一个默认实现，所以这个名字解析算法一定可以被解析成功。method_missing 方法将在第 8.4.5 节中被详细介绍。

让我们来看一个具体的例子，假设有如下代码：

```
message = "hello"
message.world
```

我们想在 String 的实例“hello”上执行一个名为 world 的方法，名字解析算法将按如下方式执行。

1. 检查它的 eigenclass 中的单键方法，在本例中没有任何单键方法。
2. 检查 String 类，该类没有名为 world 的实例方法。
3. 检查 String 类包含的 Comparable 和 Enumerable 模块，它们也都没有名为 world 的实例方法。
4. 检查 String 的超类，在这里是 Object。Object 类也没有定义名为 world 的方法。
5. 检查 Object 包含的 Kernel 模块，还是找不到名为 world 的方法。我们只能去查找名为 method_missing 的方法。
6. 在上述的每个对象（String 对象的 eigenclass、String 类、Comparable 和 Enumerable 模块、Object 类和 Kernel 模块）中查找 method_missing 方法。第一个找到的 method_missing 方法是在 Kernel 模块中定义的，因此这个方法被调用，它抛出一个异常：

```
NoMethodError: undefined method `world' for "hello":String
```

看起来 Ruby 在每次方法调用时都要做一次穷举搜索，不过，在典型的（解释器）实现中，在方法找到后，其结果被缓存起来，这样后面再查找同名方法时（其间如果没有方法定义）会快很多。

7.8.1 类方法查找
Class Method Lookup

类方法名的解析算法跟实例方法名的一样，不过有点小小的变化。让我们从一个没有变化的简单的例子开始。下面是一个没有进行任何方法定义的类 C：

```
class C
end
```

记住，如果我们用这样的方式定义了一个类，那么常量 C 就指向一个 Class 的实例。任何类方法不过是 C 对象的单键方法而已。

一旦定义了类 C，我们就可以用下面的方法调用表达式来使用类方法 new：

```
c = C.new
```

要解析 new 方法，Ruby 首先查找 C 对象的 eigenclass 中定义的单键方法。我们的类并没有定义任何方法，所以什么也找不到。在查找完 eigenclass 后，名字解析算法查找 C 的类对象。C 的类对象是 Class 的实例，因此 Ruby 将在 Class 中查找这个方法，并且找到一个名为 new 的实例方法。

你没有看错。算法要找到一个类方法 C.new，最后却找到一个实例方法 Class.new。在面向对象的风格中，类方法和实例方法是截然不同的；不过在 Ruby 中，类也用对象表示，这个区别就仅仅是表面性的了。不论是实例方法还是类方法，每个方法调用都有一个接收者对象和一个方法名，名字解析算法会为那个对象找到合适的方法定义。我们的 C 对象是一个 Class 类的实例，所以我们当然可以通过 C 来调用 Class 的实例方法。更进一步，Class 继承了 Module、Object 和 Kernel 的实例方法，这些方法也可以作为 C 的方法访问。把这些方法称为“类方法”的唯一原因就是我们的 C 对象正好是一个类。

我们的类方法 C.new 最后被发现是 Class 类的实例方法。不过，如果在 Class 类中找不到这个方法，名字解析算法会像在实例方法名解析中那样继续查找下去。如果在 Class 类中找不到，就到包含的模块中（Class 类没有包含任何模块）查找；然后去超类 Module 中查找；接着，查找 Module 的模块（Module 实际上没有包含模块）；最后查找 Module 的超类 Object 及它的模块 Kernel。

在本节开始处提到的变化与类方法的一个特性相关：类方法也可以像实例方法一样被继承。例如，可以如下定义一个 Integer.parse 方法：

```
def Integer.parse(text)
  text.to_i
end
```

Fixnum 是 Integer 的子类，可以通过如下表达式调用这个方法：

```
n = Fixnum.parse("1")
```

在前面算法的描述中可以看到，Ruby 首先查找 Fixnum 的 eigenclass 来查找单键方法。接下来，将查找 Class、Module、Object 和 Kernel 的实例方法。这样到哪里查找 parse 方法

呢？Integer 的类方法是 Integer 对象的单键方法，这就是说它在 Integer 的 eigenclass 中被定义。Integer 的 eigenclass 如何被名字解析算法找到呢？

类对象有些特殊：它们有超类。类对象的 eigenclass 也同样特别：它们也有超类。普通对象的 eigenclass 是独立的，没有超类。让我们用 Fixnum'和 Integer'分别表示 Fixnum 和 Integer 的 eigenclass，Fixnum'的超类就是 Integer'。

明白了这个变化，现在可以详细解释这个名字解析算法了。当 Ruby 查找一个对象的 eigenclass 中定义的单键方法时，也对这个 eigenclass 的超类（以及所有祖先类）进行查找。这样当查找 Fixnum 的一个类方法时，Ruby 首先查找 Fixnum、Integer、Numeric 和 Object 的单键方法，然后再查找 Class、Module、Object 和 Kernel 的实例方法。

7.9 常量查找
Constant Lookup

如果引用一个常量时没有限定任何命名空间，Ruby 解释器必须查找该常量的定义。与查找方法定义一样，这也是通过一个名字解析算法实现的。不过，常量与方法的解析方式截然不同。

Ruby 首先试图在常量引用所在的语句范围内进行解析，这意味着首先在常量引用所在的类或模块中查找，看看这个常量是否在该类或模块中被定义。如果没有，则继续查找包含此类或模块的范围，这个过程会一直到没有包含的类或模块为止。注意顶层（或“全局”）常量不在这个查找范围内。Module.nesting 方法返回一个类和模块的列表，它正好就是所要查找的类和模块，并且其顺序就是查找的顺序。

如果完成了上面的步骤后没有找到常量定义，Ruby 将试图在继承体系中进行解析，在常量引用类或模块的祖先类中查找。常量引用类或模块的 ancestors 方法返回在本步骤中查找的类和模块列表。

如果还没找到，Ruby 将检查顶层的常量定义。

如果这样还没有找到所需常量，而引用常量的类或模块定义了 const_missing 方法，那么该方法被调用，这样就有机会为那个常量提供一个值。在第 8 章中将介绍 const_missing 钩子方法，示例 8-3 将演示它的用法。

下面是一些在常量查找算法中值得注意的要点。

- 在所容纳（enclosing）的模块中定义的常量优先于所包含（included）模块中定义的常量。
- 在查找时，在类包含模块中定义的常量优先于超类中的常量。
- Object 类是所有类继承体系中的一员。不在任何类或模块中定义的顶级常量与顶级方法类似：它们都隐式定义在 Object 上。如果在一个类中引用一个顶级常量，那么，它在继承体系中查找时进行解析。不过，如果是在模块中被引用，在搜寻完该模块的所有祖先类后，要明确检查一下 Object 类。
- Kernel 模块是 Object 的一个祖先，这意味着在 Kernel 中定义的常量看起来就像顶层常量一样，不过它们可以被在 Object 中定义的真正顶层常量所覆盖。

示例7-1在6个不同的范围内对常量进行了定义和解析，演示了前面描述的常量名查找算法。

示例 7-1：常量名解析

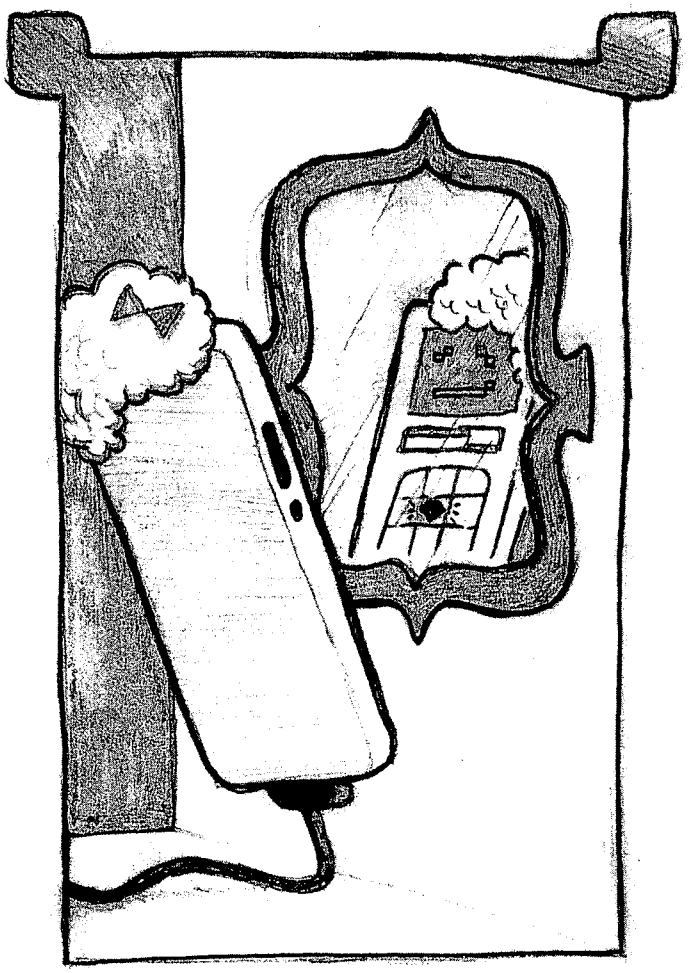
```
module Kernel
  # Constants defined in Kernel
  A = B = C = D = E = F = "defined in kernel"
end

# Top-level or "global" constants defined in Object
A = B = C = D = E = "defined at toplevel"

class Super
  # Constants defined in a superclass
  A = B = C = D = "defined in superclass"
end

module Included
  # Constants defined in an included module
  A = B = C = "defined in included module"
end

module Enclosing
  # Constants defined in an enclosing module
  A = B = "defined in enclosing module"

  class Local < Super
    include Included

    # Locally defined constant
    A = "defined locally"

    # The list of modules searched, in the order searched
    # [Enclosing::Local, Enclosing, Included, Super, Object, Kernel]
```

```
    search = (Module.nesting + self.ancestors + Object.ancestors).uniq

    puts A  # Prints "defined locally"
    puts B  # Prints "defined in enclosing module"
    puts C  # Prints "defined in included module"
    puts D  # Prints "defined in superclass"
    puts E  # Prints "defined at toplevel"
    puts F  # Prints "defined in kernel"
  end
end
```

第 8 章

反射和元编程

Reflection and Metaprogramming



我们已经看到 Ruby 是一个非常动态的语言：可以在运行时为类加入新的方法；为现有方法增加别名；甚至可以为单个对象定义方法。另外，它还为**反射**（*reflection*）提供了大量的 API。反射，也被称为**内省**（*introspection*），表示一个程序可以审视自身的状态和结构。例如，一个 Ruby 程序可以获得 Hash 类定义的方法列表；可以查询一个给定对象某个实例变量的值；也可以迭代全部在当前解释器中定义的 Regexp 对象。反射 API 比这些例子走得更远，它允许一个程序修改自身的状态和结构。一个 Ruby 程序可以动态设置有名变量、调用有名方法，甚至定义新类和新方法。

Ruby 的反射 API，以及其动态特性、代码块、迭代器控制结构、可省略圆括号的句法，使它非常适合进行**元编程**。元编程可以被粗略定义为用程序（或框架）帮助你写程序。换句话说，元编程是一系列用于扩展 Ruby 句法的技术，该技术使编程更加轻松。元编程与**领域特定语言**（*domain-specific languages*, DSLs）的关系很密切。在 Ruby 中，DSL 通常使用方法调用和代码块，它们像语言在某个特殊领域上的扩展，可以像语言的关键字一样使用。

本章开始的几个章节将介绍 Ruby 的反射 API。这些 API 相当丰富，有非常多的方法。这些方法绝大多数被定义在 Kernel、Object 和 Module 中。

当阅读这些章节时，你须要牢记的一点是，反射的本质并不是元编程。元编程一般在某种程度上扩展 Ruby 的句法或行为，通常涉及若干反射机制。在介绍完 Ruby 的核心反射 API 后，本书将用例子演示使用这些 API 的常用元编程技术。

注意，本章将覆盖一些高级主题，不读本章也不会妨碍你成为一个富有生产力的程序员。你可以先读本书的其余部分，最后再读本章。你也可以把本章看做某种期终测试：如果理解了本章中的例子（尤其是最后一个比较长的），你就精通了 Ruby!

8.1 类型、类和模块
Types, Classes, and Modules

最常用的反射方法是那些用于确定一个对象类型的方法——这个实例是属于哪个类的？它响应哪些方法？在第 3.8.4 节中，已经介绍了绝大多数的这类重要方法，下面进行一个回顾：

`o.class`

返回对象 o 的类。

`c.superclass`

返回类 c 的超类。

o.instance_of? c

判断是否 o.class == c。

o.is_a? c

判断 o 是否是 c 或它某个子类的实例。如果 c 是模块，则这个方法判断 o.class（或它的任意祖先类）是否包含了该模块。

o.kind_of? c

kind_of? 是 is_a?的同义词。

c === o

对于任意类或模块 c，判断是否 o.is_a?(c)。

o.respond_to? name

判断对象 o 是否具有一个给定名称的公开或被保护方法。如果给出第二个参数并设为 true，该方法将会检查私有方法。

8.1.1 祖先类和模块
Ancestry and Modules

除了上面的那些方法，还有一些相关的反射方法，它们可以判断一个类的祖先和一个类或模块包含的模块。这些方法很容易从下面的演示中理解：

```
module A; end                  # Empty module
module B; include A; end;      # Module B includes A
class C; include B; end;       # Class C includes module B

C < B                 # => true: C includes B
B < A                 # => true: B includes A
C < A                 # => true
Fixnum < Integer      # => true: all fixnums are integers
Integer <Comparable   # => true: integers are comparable
Integer < Fixnum      # => false: not all integers are fixnums
String < Numeric      # => nil: strings are not numbers

A.ancestors           # => [A]
B.ancestors           # => [B, A]
C.ancestors           # => [C, B, A, Object, Kernel]
String.ancestors      # => [String, Enumerable, Comparable, Object, Kernel]
                      # Note: in Ruby 1.9 String is no longer Enumerable

C.include?(B)         # => true
C.include?(A)         # => true
B.include?(A)         # => true
 A.include?(A)        # => false
A.include?(B)         # => false

A.included_modules    # => []
B.included_modules    # => [A]
C.included_modules    # => [B, A, Kernel]
```

上面的代码演示了 include?方法，它是 Module 模块定义的一个公开实例方法。它还对 include 方法（没有问号）进行了两次调用，include 方法是 Module 的私有实例方法。作为私有方法，它只能在 self 上被调用，因此它只能在 class 或 module 的定义体中被使用。这种使用 include 方法的方式就像使用关键字一样，这正是 Ruby 核心句法使用元编程的一个例子。

一个与私有 include 方法相关的方法是公开方法 Object.extend，这个方法把指定模块的实例方法作为调用对象的单键方法，这样就扩展了调用对象：

```
module Greeter; def hi; "hello"; end; end # A silly module
s = "string object"
s.extend(Greeter)         # Add hi as a singleton method to s
s.hi                      # => "hello"
String.extend(Greeter)    # Add hi as a class method of String
String.hi                 # => "hello"
```

类方法 Module.nesting 与模块包含及祖先类无关，它返回一个数组，用于指明当前代码所处的模块嵌套情况。Module.nesting[0]表示当前类或模块，Module.nesting[1]表示包含它的类或模块，依此类推：

```
module M
  class C
    Module.nesting   # => [M::C, M]
  end
end
```

8.1.2 定义类和模块
Defining Classes and Modules

类和模块分别是类 Class 和 Module 的实例，因此，你可以像下面这样动态创建它们：

```
M = Module.new          # Define a new module M
C = Class.new           # Define a new class C
D = Class.new(C) {      # Define a subclass of C
  include M             # that includes module M
}
D.to_s                  # => "D": class gets constant name by magic
```

Ruby 有一个很好的特性：如果把动态创建的匿名模块或类赋给一个常量，常量的名字就成为这个模块或类的名字（可以通过 name 或 to_s 方法返回这个名字）。

8.2 对字符串和块进行求值
Evaluating Strings and Blocks

Ruby 中一个最强大和最直接的反射特性之一就是 eval 方法。如果程序可以生成一个有效 Ruby 代码的字符串，就可以用 Kernel.eval 方法对它进行求值：

```
x = 1
eval "x + 1"  # => 2
```

eval 非常强大，不过，除非是编写一个执行用户输入代码的外壳程序（比如 *irb*），一般用户并不需要它。（并且在网络环境下，直接调用 eval 执行接收来的代码几乎肯定是不安全的，因为可能会接收到恶意代码。）不大老练的程序员有时会使用 eval 作为救命稻草，如果发现自己的代码中有这种情况，你应该看看是否有其他方式来避免它。话虽如此，eval 和一些类似方法还是有它有用之处。

8.2.1 Binding 对象和 eval 方法
Bindings and eval

一个 Binding 对象代表某一时刻 Ruby 变量的绑定状态，Kernel.binding 返回调用时实际发生的绑定。可以把一个 Binding 对象作为第二个参数传给 eval 方法，那么指定的字符串将在这些绑定的上下文中被求值。例如，如果定义了一个实例方法，它返回一个代表了对象内部变量绑定情况的 Binding 对象，然后我们就可以用这些绑定对对象的实例变量进行读写操作。该示例可以像下面这样实现：

```
class Object      # Open Object to add a new method
  def bindings    # Note plural on this method
    binding       # This is the predefined Kernel method
  end
end

class Test        # A simple class with an instance variable
  def initialize(x); @x = x; end
end

t = Test.new(10)          # Create a test object
eval("@x", t.bindings)    # => 10: We've peeked inside t
```

注意，其实没有必要像本例中定义 Object.bindings 方法这样来获得对象的实例变量，本书很快将介绍几种简单的方式来读（和写）对象实例变量的值。

正如在第 6.6.2 节中所说的，Proc 定义了一个公开的 binding 方法，它返回一个 Binding 对象，用于代表该 Proc 对象中的变量绑定。

Ruby 1.9 对 Binding 对象定义了一个 eval 方法，所以你无需再把 Binding 对象作为全局 eval 方法的第二个参数，直接调用 Binding 对象的 eval 方法即可。这两种方式不过在风格上有所区别罢了，在本质上是一样的。

8.2.2 instance_eval 和 class_eval 方法
instance_eval and class_eval

Object 类定义了一个名为 instance_eval 的方法，Module 类定义了一个名为 class_eval 的方法（module_eval 是 class_eval 的同义词），这些方法都能像 eval 一样对 Ruby 代码进行求值，但是有两个重要区别。第一个区别是这些方法会在指定对象或模块的上下文中对代码进行求值，该对象或模块成为代码求值时 self 的值。示例如下：

```
o.instance_eval("@x")  # Return the value of o's instance variable @x

# Define an instance method len of String to return string length
String.class_eval("def len; size; end")

# Here's another way to do that
# The quoted code behaves just as if it was inside "class String" and "end"
String.class_eval("alias len size")

# Use instance_eval to define class method String.empty
# Note that quotes within quotes get a little tricky...
String.instance_eval("def empty; ''; end")
```

注意当求值的代码中定义了方法时，instance_eval 和 class_eval 方法有微妙但本质的区别。instance_eval 为这个对象创建一个单键方法（当对象是一个类对象时，该方法成为类方法），class_eval 则定义一个普通的实例方法。

这两个方法与 eval 的第二个重要区别是它们可以对代码块进行求值。当参数是代码块而非字符串时，代码块中的代码会在适当的上下文中执行。这样，上面的调用可以用如下方式进行改写：

```
o.instance_eval { @x }
String.class_eval {
  def len
    size
  end
}
String.class_eval { alias len size }
String.instance_eval { def empty; ""; end }
```

8.2.3 instance_exec 和 class_exec 方法
instance_exec and class_exec

Ruby 1.9 还定义了两个求值方法：instance_exec 和 class_exec（和它的别名方法：module_exec），这些方法在接收者对象的上下文中对给定代码块（而非字符串）进行求值，这与 instance_eval 和 class_eval 一样。不同之处在于 exec 的这两个方法可以接受参数，并把参数传给代码块，这样，给定的代码块会在给定对象的上下文中被执行，并可以获得该对象以外的参数。

8.3 变量和常量

Variables and Constants

Kernel、Object 和 Module 类定义了很多反射方法来列出各种名字（以字符串方式），包括所有定义的全局变量、当前使用的局部变量、一个对象的全部实例变量、一个类或模块的全部类变量和常量：

```
global_variables      # => ["$DEBUG", "$SAFE", ...]
x = 1                 # Define a local variable
local_variables       # => ["x"]

# Define a simple class
class Point
  def initialize(x,y); @x,@y = x,y; end   # Define instance variables
  @@classvar = 1                          # Define a class variable
  ORIGIN = Point.new(0,0)                 # Define a constant
end

Point::ORIGIN.instance_variables  # => ["@y", "@x"]
Point.class_variables             # => ["@@classvar"]
Point.constants                   # => ["ORIGIN"]
```

global_variables、instance_variables、class_variables 和 constants 方法在 Ruby 1.8 中返回字符串数组，在 Ruby 1.9 中则返回符号数组，而 local_variables 方法在两个版本中都返回字符串数组。

8.3.1 查询、设置和检测变量

Querying, Setting, and Testing Variables

除了列出各种定义的变量和常量，Ruby 的 Object 和 Module 类还定义了很多反射方法用于查询、设置和删除实例变量、类变量和常量。没有专门的方法用于读写局部和全局变量，不过这可以通过 eval 方法实现：

```
x = 1
varname = "x"
eval(varname)             # => 1
eval("varname = '$g'")    # Set varname to "$g"
eval("#{varname} = x")    # Set $g to 1
eval(varname)             # => 1
```

注意 eval 在一个临时范围内对代码进行求值，它可以改变一个现有实例变量的值。不过 eval 定义的变量只能在 eval 调用局部可见，在 eval 返回时它们就不可访问了。（这看起来就像代码在一个代码块中被求值一样，代码块的局部变量在代码块外不可见。）

你可以对任何对象的实例变量、任何类或模块的类变量和常量进行查询、设置和存在性检测操作：

```
o = Object.new
o.instance_variable_set(:@x, 0)   # Note required @ prefix
o.instance_variable_get(:@x)      # => 0
```

```
o.instance_variable_defined?(:@x) # => true

Object.class_variable_set(:@@x, 1)    # Private in Ruby 1.8
Object.class_variable_get(:@@x)       # Private in Ruby 1.8
Object.class_variable_defined?(:@@x)  # => true; Ruby 1.9 and later

Math.const_set(:EPI, Math::E*Math::PI)
Math.const_get(:EPI)              # => 8.53973422267357
Math.const_defined? :EPI          # => true
```

在 Ruby 1.9 中，如果把 `false` 作为 `const_get` 和 `const_defined?`的第二个参数，则表明只在当前类或模块中进行查找，而无须考虑继承的常量。

在 Ruby 1.8 中，查询和查找类变量的方法是私有的，这时，可以通过 `class_eval` 方法调用它们：

```
String.class_eval { class_variable_set(:@@x, 1) } # Set @@x in String
String.class_eval { class_variable_get(:@@x) }    # => 1
```

`Object` 和 `Module` 对象中的一些私有方法被用来取消实例变量、类变量和常量的定义，它们都返回被删除变量或常量的值。因为这些方法是私有的，不能直接在对象、类或模块上进行调用，所以只能通过 `eval` 或 `send` 方法（在本章后面将讲到）完成：

```
o.instance_eval { remove_instance_variable :@x }
String.class_eval { remove_class_variable(:@@x) }
Math.send :remove_const, :EPI  # Use send to invoke private method
```

如果一个模块定义了 `const_missing` 方法，当找不到指定常量时，该方法将被调用。你可以让这个方法返回给定名字的常量。（例如，这个特性可以用来实现一种自动加载机制，让类或模块可以被按需加载。）示例如下：

```
def Symbol.const_missing(name)
  name # Return the constant name as a symbol
end
Symbol::Test   # => :Test: undefined constant evaluates to a Symbol
```

8.4 方法
Methods

`Object` 和 `Module` 类定义了一组用于列举、查询、调用和定义方法的方法。我们将依次考察这些方法。

8.4.1 方法的列举和测试
Listing and Testing For Methods

`Object` 定义了一组用于列举对象所定义方法名的方法，这些方法会返回方法名数组。方法名在 Ruby 1.8 中用字符串表示，而在 Ruby 1.9 中则使用符号。

```
o = "a string"
o.methods                # => [ names of all public methods ]
o.public_methods         # => the same thing
o.public_methods(false)  # Exclude inherited methods
o.protected_methods      # => []: there aren't any
o.private_methods        # => array of all private methods
o.private_methods(false) # Exclude inherited private methods
def o.single; 1; end     # Define a singleton method
o.singleton_methods      # => ["single"] (or [:single] in 1.9)
```

你也可以向类查询其定义的方法，而不用非要向其实例进行查询。下面的方法由 Module 定义，像 Object 中的方法一样，在 Ruby 1.8 中返回字符串数组，而在 Ruby 1.9 中则返回符号数组：

```
String.instance_methods == "s".public_methods             # => true
String.instance_methods(false) == "s".public_methods(false) # => true
String.public_instance_methods == String.instance_methods # => true
String.protected_instance_methods         # => []
String.private_instance_methods(false)    # => ["initialize_copy",
                                          #     "initialize"]
```

前面说过类或模块的类方法其实就是 Class 或 Module 对象的单键方法，因此你也可以用 Object.singleton_methods 来列举类方法：

```
Math.singleton_methods  # => ["acos", "log10", "atan2", ... ]
```

除了上面这些列举方法，Module 类还定义了一些断言方法用于判断某个特定类或模块是否定义了一个给定名称的实例方法：

```
String.public_method_defined? :reverse        # => true
String.protected_method_defined? :reverse     # => false
String.private_method_defined? :initialize    # => true
String.method_defined? :upcase!               # => true
```

Module.method_defined?方法用于检测给定的名称是否用于定义了一个公开或保护方法，它与 Object.respond_to?方法的功能基本类似。在 Ruby 1.9 中，你可以指定该方法的第二个参数为 false 来表示不考虑继承的方法。

8.4.2 获得 Method 对象
Obtaining Method Objects

要查询一个特定名称的方法，你可以在任意对象上调用 method 方法，或者在任意模块上调用 instance_method 方法。前者返回一个绑定于接收者的可调用 Method 对象，而后者返回一个 UnboundMethod 对象。在 Ruby 1.9 中，可以用 public_method 和 public_instance_method 限定只查找公开方法，在第 6.7 节已经介绍过这些方法及其返回的对象：

```
"s".method(:reverse)              # => Method object
String.instance_method(:reverse)  # => UnboundMethod object
```

8.4.3 调用方法
Invoking Methods

如前所述（在第 6.7 节中也有介绍），你可以在任意对象上用 method 方法获得一个 Method 对象，它可以代表该对象的一个有名字的方法。Method 对象与 Proc 对象一样，都有一个 call 方法，可以用它调用该方法。

通常，调用一个有名字的方法更简单的方式是使用 send 方法：

```
"hello".send :upcase          # => "HELLO": invoke an instance method
Math.send(:sin, Math::PI/2)   # => 1.0: invoke a class method
```

send 在接收者对象上调用名为第一个参数的方法，后面的其他参数会作为调用方法的参数。方法名“send”来自于面向对象编程的术语，在那里调用一个方法被称为向一个对象“发送一个消息”。

send 可以调用一个对象的任意有名方法，也包括私有或被保护方法。在前面我们就用 send 调用过 Module 的私有方法 remove_const。因为全局函数实际上是 Object 的私有方法，所以你也可以在任意对象上用 send 方法对它们进行调用（尽管我们不大可能这么做）：

```
"hello".send :puts, "world"        # prints "world"
```

Ruby 1.9 中，定义了一个名为 public_send 的替代方法，它与 send 的功能相似，不过它只调用公开方法，不会调用私有或被保护方法：

```
"hello".public_send :puts, "world"  # raises NoMethodError
```

send 是 Object 类中非常基础的一个方法，不过这个名字太过普通，有可能被子类中的方法所覆写。因此，Ruby 定义了__send__作为其同义词方法，如果试图删除或重定义这个方法，你会得到一个警告。

8.4.4 定义方法、取消方法定义及别名方法
Defining, Undefining, and Aliasing Methods

如果想为类或模块定义一个新的实例方法，你可以用 define_method 方法。它是 Module 的一个实例方法，第一个参数是新方法的名字（用符号表示），方法体要么用 Method 对象表示作为第二个参数，要么用代码块表示。要记住 define_method 是一个私有方法，必须在须要使用它的类或模块内来调用它：

```
# Add an instance method named m to class c with body b
def add_method(c, m, &b)
  c.class_eval {
    define_method(m, &b)
  }
end

add_method(String, :greet) { "Hello, " + self }

"world".greet   # => "Hello, world"
```

定义属性访问器方法

attr_reader 和 attr_accessor 方法（参见第 7.1.5 节）也可以用来定义新方法。与 define_method 类似，它们也是 Module 的私有方法，而且可以用 define_method 轻松实现。这些创建方法的方法是表明 define_method 用途的完美例子。值得注意的是，因为这些方法是用来在类定义内部使用的，所以 define_method 是私有方法，不会造成任何影响。

如果要用 define_method 创建一个类方法（或者任何单键方法），你可以在相应的 eigenclass 上调用它：

```
def add_class_method(c, m, &b)
  eigenclass = class << c; self; end
  eigenclass.class_eval {
    define_method(m, &b)
  }
end

add_class_method(String, :greet) {|name| "Hello, " + name }

String.greet("world")  # => "Hello, world"
```

在 Ruby 1.9 中，你可以简单地使用 define_singleton_method 方法，它是 Object 的一个方法：

```
String.define_singleton_method(:greet) {|name| "Hello, " + name }
```

define_method 方法的一个缺点是不能定义方法体使用代码块的方法。如果想动态创建一个接受代码块的方法，须要联合使用 def 和 class_eval 方法。如果要创建的方法有足够的动态性，你可能无法把一个代码块传给 class_eval 方法，这时则要用一个字符串表示要定义的方法，然后对它进行求值。在本章后面部分我们将看到一些这样的例子。

如果要为已有方法创建一个同义词或别名方法，你可以使用 alias 语句：

```
alias plus +        # Make "plus" a synonym for the + operator
```

不过在进行动态编程时，有时须要使用 alias_method 方法。与 define_method 一样，alias_method 也是 Module 类的私有方法。因为它是一个方法，所以它可以接受两个任意的表达式作为它的参数，而不用在源代码中加入两个硬编码的标识符。（因为是方法，所以在参数间须要加入一个逗号。）alias_method 经常用于创建别名链，下面是一个简单示例，在本章后面你还会看到更多的例子。

```
# Create an alias for the method m in the class (or module) c
def backup(c, m, prefix="original")
  n = :"#{prefix}_#{m}"       # Compute the alias
  c.class_eval {              # Because alias_method is private
```

```
    alias_method n, m      # Make n an alias for m
  }
end

backup(String, :reverse)
"test".original_reverse # => "tset"
```


正如在第 6.1.5 节中所见到的，你可以用 undef 语句取消一个方法的定义，不过这只用于可以用硬编码标识符表示名字的方法。如果须要动态删除由程序计算得到名字的方法，有两个选择：remove_method 或 undef_method。它们都是 Module 的私有方法，remove_method 删除当前类中的方法定义。如果该方法在超类中有定义，该定义在执行该方法后被继承下来。undef_method 则更严厉一些，它阻止给定方法在任何实例上的调用，即使存在这样一个在超类中继承而来的方法。

如果在定义类时，希望阻止对它进行动态修改，你可以简单地调用该类的 freeze（中文意为冻结）方法。一旦该类被冻结，则不能被改变。

8.4.5 处理未定义的方法
Handling Undefined Methods

当方法名解析算法（参见第 7.8 节）无法找到一个方法时，它会转而去寻找一个名为 method_missing 的方法。当此方法被调用时，它的第一个参数是一个符号，表示无法找到的方法的名字，后面的参数都是本来要被传给那个没有找到的方法的，如果在进行方法调用时带了一个代码块，这个代码块也会被传给 method_missing 方法。

在 Kernel 模块中，method_missing 的默认实现不过是简单地抛出一个 NoMethodError 异常。如果不对这个异常进行捕获，程序将在给出一个错误消息后退出。这也是通常在调用一个不存在的方法后所期望的结果。

为一个类定制自己的 method_missing 方法使你有机会处理该类实例上的任何方法调用。method_missing 钩子是 Ruby 动态机制中最强大的工具之一，它常常被用于元编程技术。在本章后面部分，我们将看到一些例子。现在，下面的示例代码为 Hash 类增加了一个 method_missing 方法，它使我们可以像方法一样对某个主键的值进行查询或设置：

```
class Hash
  # Allow hash values to be queried and set as if they were attributes.
  # We simulate attribute getters and setters for any key.
  def method_missing(key, *args)
    text = key.to_s

    if text[-1,1] == "="              # If key ends with = set a value
```

```
      self[text.chop.to_sym] = args[0]      # Strip = from key
    else                                    # Otherwise...
      self[key]                             # ...just return the key value
    end
  end
end

h = {}            # Create an empty hash object
h.one = 1         # Same as h[:one] = 1
puts h.one        # Prints 1. Same as puts h[:one]
```

8.4.6 设置方法的可见性
Setting Method Visibility

在第 7.2 节中，本书引入了 public、protected 和 private 方法，它们看起来很像语言的关键字，不过实际上只是 Module 定义的私有方法而已。这些方法一般只在类定义中被静态使用，不过，通过 class_eval 方法，它们也可以被动态使用：

```
String.class_eval { private :reverse }
"hello".reverse  # NoMethodError: private method 'reverse'
```

private_class_method 和 public_class_method 与它相似，不过它们是对类的公开方法进行操作：

```
# Make all Math methods private
# Now we have to include Math in order to invoke its methods
Math.private_class_method *Math.singleton_methods
```

8.5 钩子方法
Hooks

Module、Class 和 Object 类实现了若干回调方法，或者被称为**钩子**方法。这些方法并非默认定义的，不过如果为一个模块、类或对象定义了这些方法，它们会在特定事件发生时被调用，这使得在定义子类时、包含模块时或定义方法时，我们可以扩展 Ruby 的行为。这些钩子方法（除了那些已经过期的方法，本书对它们不做介绍）名以“ed”结尾。

当定义一个新类时，Ruby 在其超类上调用类方法 inherited，它以新类作为参数，这可以让类为它们的子类增加一些行为或进行一些限制。前面讲过类方法是继承来的，因此，如果一个新类的任何超类定义了 inherited 方法，该方法都会被调用。基于此，我们可以定义 Object.inherited 方法来捕获所有定义新类的事件：

```
def Object.inherited(c)
  puts "class #{c} < #{self}"
end
```

当一个模块被另一个模块或类包含时，被包含模块的类方法 included 将被调用，其参数是

包含它的模块或类。这让被包含模块有机会增强或修改该类——它可以用于为一个模块定义特殊意义的包含方式。除了给被包含类增加新的方法，具有 included 方法的模块还可以修改其现有的方法，比如：

```
module Final                # A class that includes Final can't be subclassed
  def self.included(c)      # When included in class c
    c.instance_eval do      # Define a class method of c
      def inherited(sub)    # To detect subclasses
        raise Exception,    # And abort with an exception
              "Attempt to create subclass #{sub} of Final class #{self}"
      end
    end
  end
end
```

相似地，如果一个模块定义了一个名为 extended 的类方法，它将在一个对象扩展该模块（用 Object.extend 方法）时被调用。extended 方法的参数当然是要扩展的对象，它可以对要扩展的对象做任何想做的操作。

除了有跟踪包含类和模块的钩子方法，还有用于跟踪类和模块的方法的钩子方法，以及跟踪任意对象单键方法的钩子方法。如果为任意类或模块定义一个名为 method_added 的方法，它将在为该类或模块定义一个实例方法时被调用：

```
def String.method_added(name)
  puts "New instance method #{name} added to String"
end
```

值得注意的是，method_added 类方法会被该类的子类所继承。但是因为这个钩子方法没有任何类信息的参数，所以在添加某个方法时，无法知道是定义 method_added 方法所在的类添加的，还是它的某个子类所添加的。解决这个问题的一个方法是为定义 method_added 的类同时定义 inherited 钩子方法，然后这个 inherited 方法为每个子类定义一个 method_added 方法。

在为任意对象添加单键方法时，将调用该对象的 singleton_method_added 方法，新方法的名字被传给该方法。记得对于类来说，单键方法就是类方法：

```
def String.singleton_method_added(name)
  puts "New class method #{name} added to String"
end
```

有趣的是，在首次定义 singleton_method_added 这个钩子方法时，它也会对自身进行调用。这个钩子方法还有另外一种用途，这时，singleton_method_added 作为包含模块的类的实例方法。在向该类的实例中添加任何单键方法时，它都会得到通知：

```
# Including this module in a class prevents instances of that class
# from having singleton methods added to them. Any singleton methods added
# are immediately removed again.
module Strict
  def singleton_method_added(name)
    STDERR.puts "Warning: singleton #{name} added to a Strict object"
    eigenclass = class << self; self; end
    eigenclass.class_eval { remove_method name }
  end
end
```

除了 method_added 和 singleton_method_added 钩子方法，有的钩子方法可追踪实例方法和单键方法的删除或取消。当一个实例方法被删除或取消时，method_removed 和 method_undefined 类方法将在该模块上被调用。当一个单键方法被删除或取消时，该对象的实例方法 singleton_method_removed 和 singleton_method_undefined 被调用。

最后要说明的是，method_missing 和 const_missing 的行为也像钩子方法，不过它们在本章其他部分被加以说明。

8.6 跟踪
Tracing

Ruby 定义了很多特性用于跟踪程序的执行，对于调试代码和打印有意义的错误消息，它们很有用处。其中最简单的两个特性是两个关键字：__FILE__和__LINE__。这些关键字表示代码文件名及代码所出现的行号，这样你可以用它们来获得错误发生的确切位置：

```
STDERR.puts "#{__FILE__}:#{__LINE__}: invalid data"
```

另外，值得注意的是，Kernel.eval、Object.instance_eval 和 Module.class_eval 方法都接受一个文件名（或其他字符串）和一个行号作为它们最后两个参数。如果要对从文件中抽取的代码求值，你可以用这些参数来指定求值计算中的__FILE__和__LINE__值。

无疑，你已经注意到当一个异常抛出时，打印到控制台的错误消息包含了文件名和行号信息。这些信息当然是基于__FILE__和__LINE__而得来的。每个 Exception 对象关联一个**追踪信息**，可以准确得知它是在哪里抛出的，是在方法中的什么地方抛出的，以及是在调用哪个方法时抛出的，等等。Exception.backtrace 方法返回一个包含这些信息的字符串数组。数组的第一个元素是异常发生的位置，后面的每个元素都是一个更高一级的调用堆栈窗口。

不过，无须抛出异常就可以获得当前调用栈的信息。Kernel.caller 方法可返回当前调用堆栈的状态，返回值的形式与 Exception.backtrace 的相同。如果不带参数，caller 方

法返回的调用堆栈的第一个元素是一个方法，该方法调用了那个调用 caller 的方法，也就是说，caller[0]是调用当前方法的位置。caller 方法也可以带有一个参数，指明忽略前面多少个堆栈窗口，它的默认值是 1，caller(0)[0]指明 caller 方法调用的位置，这意味着 caller[0]等价于 caller(0)[1]，而 caller(2)等价于 caller[1..-1]。

Exception.backtrace 和 Kernel.caller 返回的调用堆栈也包含方法名。在 Ruby 1.9 之前，你必须解析这些字符串来提取方法名。不过在 Ruby 1.9 中，当前执行的方法名（用符号表示）可以用 Kernel.__method__或同义词方法 Kernel.__callee__获得，__method__在与__FILE__和__LINE__联合使用时很有用：

```
raise "Assertion failed in #{__method__} at #{__FILE__}:#{__LINE__}"
```

注意，即使我们通过别名方式调用一个方法，__method__方法还是返回方法的原始定义名。

除了简单地打印发生错误的文件名和行号，你还可以进一步打印出错行的代码。如果程序定义了一个名为 SCRIPT_LINES__的全局常量，并且使它等同于一个哈希表对象，那么 require 和 load 方法在每次加载文件时会向它增加一个条目。哈希表的主键是文件名，它们关联的对象是对应文件所包含的代码行数组。如果希望其中也包含主文件（而不只是它加载的文件），你可以用如下方式对它进行初始化：

```
SCRIPT_LINES__ = {__FILE__ => File.readlines(__FILE__)}
```

这样，你就可以在任何地方用如下表达式获得当前的代码行：

```
SCRIPT_LINES__[__FILE__][__LINE__-1]
```

Ruby 可以通过 Kernel.trace_var 方法跟踪对全局变量的赋值。该方法的参数包括一个代表全局变量名的符号和一个字符串（或一个代码块），当给定名字的变量发生变化时，对传入的字符串进行求值（或执行代码块）。如果传入的是代码块，新的变量值会作为参数被传给块。如果要停止对变量的跟踪，可调用 Kernel.untrace_var 方法。在下面的例子中，要注意 caller[1]的用法，它用于确定跟踪变量块的调用位置：

```
# Print a message every time $SAFE changes
trace_var(:$SAFE) {|v|
  puts "$SAFE set to #{v} at #{caller[1]}"
}
```

最后要介绍的跟踪方法是 Kernel.set_trace_func，它注册一个 Proc，在每行代码执行

后运行。如果想写一个调试模块，允许一行行地单步运行整个程序，set_trace_func 方法会很有用。不过我们在此并不对其进行详细介绍。

8.7 ObjectSpace 和 GC
ObjectSpace and GC

ObjectSpace 模块定义了一组方便的低级方法，它们有时对调试和元编程有用。最值得注意的方法是 each_object，它是一个迭代器方法，可以迭代出解释器知道的每个对象（或者某个指定类的所有实例）：

```
# Print out a list of all known classes
ObjectSpace.each_object(Class) {|c| puts c }
```

ObjectSpace._id2ref 是 Object.object_id 的反向方法，它的参数是一个对象 ID，它返回相对应的对象；如果没有对应的对象，则抛出一个 RangeError 异常。

ObjectSpace.define_finalizer 可以注册一个 Proc 对象或一个代码块，它们在给定对象被垃圾收集时调用。不过，在注册这样一个方法时必须注意不能使用被垃圾收集的对象。任何用于终结（finalize）对象的值必须能被 finalizer 块获得，这样它们无须通过被终结对象而使用。你可以用 ObjectSpace.undefine_finalizer 方法为一个对象删除所有注册的 finalizer 块。

最后一个要介绍的 ObjectSpace 方法是 ObjectSpace.garbage_colect，它强制让 Ruby 运行垃圾收集器。垃圾收集功能也可以通过 GC 模块获得。GC.start 是 ObjectSpace.garbage_collect 的同义词方法，可以通过 GC.disable 方法临时关闭垃圾收集，用 GC.enable 使之再次生效。

混合使用_id2ref 和 define_finalizer 方法可以实现对象的“弱引用（weak reference）”定义，这使得该对象保有值的引用，然而，在它们不可用时并不阻止对它们进行垃圾收集。标准库的 WeakRef 类（在 *lib/weakref.rb* 中）有其示例，你可以参考。

8.8 定制控制结构
Custom Control Structures

Ruby 代码块及可省略括号的句法，使定义外观和行为都很像控制结构的迭代器方法一样非常容易。Kernel 的 loop 方法就是一个简单的例子。在本节我们将进一步定义三个例子。这里的例子用到了 Ruby 的线程 API，你可能要参照第 9.9 节来弄懂全部细节。

8.8.1 延迟和重复执行：after 和 every 方法

Delaying and Repeating Execution: after and every

示例 8-1 定义了两个名为 after 和 every 的全局方法。这两个方法都接受一个表示秒数的数值参数，而且都可以关联一个代码块。after 方法创建一个线程并立即返回这个创建的 Thread 对象。新创建的线程在等待给定的秒数后调用（不带参数）所关联的代码块。every 方法与之相似，但是它重复调用关联代码块，每次调用之间会给定睡眠的秒数。every 方法的第二个参数在关联代码块第一次调用时被传给代码块，而每次调用的结果则被传给下一次调用。every 方法关联的代码块可以用 break 来取消后面的调用。

下面是一些使用 after 和 every 的例子：

```
require 'afterevery'

1.upto(5) {|i| after i { puts i} }    # Slowly print the numbers 1 to 5
sleep(5)                              # Wait five seconds
every 1, 6 do |count|                 # Now slowly print 6 to 10
  puts count
  break if count == 10
  count + 1                           # The next value of count
end
sleep(6)                              # Give the above time to run
```

最后的 sleep 调用用于防止程序在 every 创建的线程完成计数之前退出。在学习了如何使用 after 和 every 后，我们将看看如何实现它们。如果你不大明白 Thread.new 的含义，记得参考第 9.9 节。

示例 8-1：after 和 every 方法

```
#
# Define Kernel methods after and every for deferring blocks of code.
# Examples:
#
#   after 1 { puts "done" }
#   every 60 { redraw_clock }
#
# Both methods return Thread objects. Call kill on the returned objects
# to cancel the execution of the code.
#
# Note that this is a very naive implementation. A more robust
# implementation would use a single global timer thread for all tasks,
# would allow a way to retrieve the value of a deferred block, and would
# provide a way to wait for all pending tasks to complete.
#

# Execute block after sleeping the specified number of seconds.
def after(seconds, &block)
  Thread.new do       # In a new thread...
    sleep(seconds)    # First sleep
    block.call        # Then call the block
```

```
  end              # Return the Thread object right away
end

# Repeatedly sleep and then execute the block.
# Pass value to the block on the first invocation.
# On subsequent invocations, pass the value of the previous invocation.
def every(seconds, value=nil, &block)
  Thread.new do                    # In a new thread...
    loop do                        # Loop forever (or until break in block)
      sleep(seconds)               # Sleep
      value = block.call(value)    # And invoke block
    end                            # Then repeat..
  end                              # every returns the Thread
end
```

8.8.2 用同步代码块实现线程安全
Thread Safety with Synchronized Blocks

当编写多线程程序时，重要的一点是两个线程不能试图同时修改同一个对象。实现方法之一是把需要线程安全的代码放在调用一个 Mutex 对象的 synchronize 方法的代码块中。再说一次，这些内容将在第 9.9 节中被详细讨论。在示例 8-2 中，我们将更进一步，实现一个名为 synchronized 的全局方法来模拟 Java 的 synchronized 关键字。这个 synchronized 方法期望一个对象参数和一个代码块，它从对象参数中获取关联的 Mutex 对象，然后用 Mutex.synchronize 调用这个代码块。由于 Ruby 对象与 Java 对象不同，它没有关联的 Mutex 对象，因此我们在示例 8-2 中使用了一个小技巧，它为 Object 定义了一个名为 mutex 的实例方法。有趣的是，mutex 方法的实现用关键字形式使用了新定义的 synchronized 方法。

示例 8-2：简单的同步代码块

```
require 'thread'  # Ruby 1.8 keeps Mutex in this library

# Obtain the Mutex associated with the object o, and then evaluate
# the block under the protection of that Mutex.
# This works like the synchronized keyword of Java.
def synchronized(o)
  o.mutex.synchronize { yield }
end

# Object.mutex does not actually exist. We've got to define it.
# This method returns a unique Mutex for every object, and
# always returns the same Mutex for any particular object.
# It creates Mutexes lazily, which requires synchronization for
# thread safety.
class Object
  # Return the Mutex for this object, creating it if necessary.
  # The tricky part is making sure that two threads don't call
  # this at the same time and end up creating two different mutexes.
  def mutex
    # If this object already has a mutex, just return it
    return @__mutex if @__mutex
```

```
    # Otherwise, we've got to create a mutex for the object.
    # To do this safely we've got to synchronize on our class object.
    synchronized(self.class) {
      # Check again: by the time we enter this synchronized block,
      # some other thread might have already created the mutex.
      @__mutex = @__mutex || Mutex.new
    }
    # The return value is @__mutex
  end
end

# The Object.mutex method defined above needs to lock the class
# if the object doesn't have a Mutex yet. If the class doesn't have
# its own Mutex yet, then the class of the class (the Class object)
# will be locked. In order to prevent infinite recursion, we must
# ensure that the Class object has a mutex.
Class.instance_eval { @__mutex = Mutex.new }
```

8.9 缺失的方法和常量

Missing Methods and Missing Constants

method_missing 方法是 Ruby 方法查找算法的一个重要部分（参见第 7.8 节），并且当希望调用对象的任意方法时（译注 1），它能发挥强大的作用。Module 的 const_missing 方法对于常量查找算法有类似的作用，而且也对运行时计算或惰性初始化常量有用。下面的例子将演示这些方法。

8.9.1 用 const_missing 实现 Unicode 方式的代码点常量

Unicode Codepoint Constants with const_missing

示例 8-3 定义了一个 Unicode 模块，它似乎为从 U+0000 到 U+10FFFF 的每个 Unicode 代码点（codepoint）都定义了一个常量（一个 UTF-8 编码的字符串）。定义这么多常量的唯一可行方式就是使用 const_missing 方法。下面的代码假设一个常量被引用一次后，很可能会被继续引用，因此 const_missing 方法调用了 Module.const_set 方法，在计算出一个值后为之定义一个真正的常量。

示例 8-3：用 const_missing 实现的 Unicode 代码点常量

```
# This module provides constants that define the UTF-8 strings for
# all Unicode codepoints. It uses const_missing to define them lazily.
# Examples:
#   copyright = Unicode::U00A9
#   euro = Unicode::U20AC
#   infinity = Unicode::U221E
module Unicode
  # This method allows us to define Unicode codepoint constants lazily.
  def self.const_missing(name)  # Undefined constant passed as a symbol
    # Check that the constant name is of the right form.
    # Capital U followed by a hex number between 0000 and 10FFFF.
    if name.to_s =~ /^U([0-9a-fA-F]{4,5}|10[0-9a-fA-F]{4})$/
      # $1 is the matched hexadecimal number. Convert to an integer.
```

译注 1：即调用的方法名是任意的，不管它是否真正在类中进行过定义。

```
      codepoint = $1.to_i(16)
      # Convert the number to a UTF-8 string with the magic of Array.pack.
      utf8 = [codepoint].pack("U")
      # Make the UTF-8 string immutable.
      utf8.freeze
      # Define a real constant for faster lookup next time, and return
      # the UTF-8 text for this time.
      const_set(name, utf8)
    else
      # Raise an error for constants of the wrong form.
      raise NameError, "Uninitialized constant: Unicode::#{name}"
    end
  end
end
```

8.9.2 用 method_missing 方法跟踪方法调用

Tracing Method Invocations with method_missing

在第 8.4.5 节，本书演示了用 method_missing 扩展 Hash 类。现在，在示例 8-4 中，我们将演示怎样用 method_missing 方法把对一个方法的调用代理到另一个对象上。在这个例子中，这样做的目的是输出该对象的跟踪信息。

示例 8-4 定义了一个 Object.trace 实例方法和一个 TracedObject 类。trace 方法返回一个 TracedObject 对象，这个对象使用 method_missing 方法来捕获方法调用，并跟踪它们，然后把这些调用代理到要跟踪的对象上。你可以用如下方式使用它：

```
a = [1,2,3].trace("a")
a.reverse
puts a[2]
puts a.fetch(3)
```

这会产生如下的跟踪信息：

```
Invoking: a.reverse() at trace1.rb:66
Returning: [3, 2, 1] from a.reverse to trace1.rb:66
Invoking: a.fetch(3) at trace1.rb:67
Raising: IndexError:index 3 out of array from a.fetch
```

注意，除了演示了 method_missing 方法，示例 8-4 还演示了 Module.instance_methods、Module.undef_method 和 Kernel.caller 方法。

示例 8-4：用 method_missing 方法跟踪方法调用

```
# Call the trace method of any object to obtain a new object that
# behaves just like the original, but which traces all method calls
# on that object. If tracing more than one object, specify a name to
# appear in the output. By default, messages will be sent to STDERR,
# but you can specify any stream (or any object that accepts strings
# as arguments to <<).
class Object
  def trace(name="", stream=STDERR)
    # Return a TracedObject that traces and delegates everything else to us.
    TracedObject.new(self, name, stream)
```

```
  end
end

# This class uses method_missing to trace method invocations and
# then delegate them to some other object. It deletes most of its own
# instance methods so that they don't get in the way of method_missing.
# Note that only methods invoked through the TracedObject will be traced.
# If the delegate object calls methods on itself, those invocations
# will not be traced.
class TracedObject
  # Undefine all of our noncritical public instance methods.
  # Note the use of Module.instance_methods and Module.undef_method.
  instance_methods.each do |m|
    m = m.to_sym  # Ruby 1.8 returns strings, instead of symbols
    next if m == :object_id || m == :__id__ || m == :__send__
    undef_method m
  end

  # Initialize this TracedObject instance.
  def initialize(o, name, stream)
    @o = o           # The object we delegate to
    @n = name        # The object name to appear in tracing messages
    @trace = stream  # Where those tracing messages are sent
  end

  # This is the key method of TracedObject. It is invoked for just
  # about any method invocation on a TracedObject.
  def method_missing(*args, &block)
    m = args.shift       # First arg is the name of the method
    begin
      # Trace the invocation of the method.
      arglist = args.map {|a| a.inspect}.join(', ')
      @trace << "Invoking: #{@n}.#{m}(#{arglist}) at #{caller[0]}\n"
      # Invoke the method on our delegate object and get the return value.
      r = @o.send m, *args, &block
      # Trace a normal return of the method.
      @trace << "Returning: #{r.inspect} from #{@n}.#{m} to #{caller[0]}\n"
      # Return whatever value the delegate object returned.
      r
    rescue Exception => e
      # Trace an abnormal return from the method.
      @trace << "Raising: #{e.class}:#{e} from #{@n}.#{m}\n"
      # And re-raise whatever exception the delegate object raised.
      raise
    end
  end

  # Return the object we delegate to.
  def __delegate
    @o
  end
end
```

8.9.3 通过代理实现对象同步
Synchronized Objects by Delegation

在示例 8-2 中，我们看到了一个全局方法 synchronized，它接受一个对象，并在该对象所关联的 Mutex 的保护下执行一个代码块。在这个例子中，主要的工作在于实现 Object.mutex 方法，而 synchronized 方法则没多少内容：

```
def synchronized(o)
  o.mutex.synchronize { yield }
end
```

在示例 8-5 中，我们修改了这个方法，如果不带代码块调用这个方法，它返回一个包装给定对象的 SynchronizedObject 对象。SynchronizedObject 是基于 method_missing 实现的代理包装类，它与示例 8-4 中的 TraceObject 很相似，不过它是 Ruby 1.9 中 BasicObject 的子类，因此无须显式删除 Object 的实例方法。注意下面的代码不能单独运行，它须要使用前面定义的 Object.mutex 方法。

示例 8-5：用 method_missing 实现同步方法

```
def synchronized(o)
  if block_given?
    o.mutex.synchronize { yield }
  else
    SynchronizedObject.new(o)
  end
end

# A delegating wrapper class using method_missing for thread safety
# Instead of extending Object and deleting our methods we just extend
# BasicObject, which is defined in Ruby 1.9. BasicObject does not
# inherit from Object or Kernel, so the methods of a BasicObject cannot
# invoke any top-level methods: they are just not there.
class SynchronizedObject  < BasicObject
  def initialize(o); @delegate = o;  end
  def __delegate; @delegate; end

  def method_missing(*args, &block)
    @delegate.mutex.synchronize {
      @delegate.send *args, &block
    }
  end
end
```

8.10 动态创建方法
Dynamically Creating Methods

元编程的一个重要技术是用方法创建方法，attr_reader 和 attr_accessor 方法（参见第 7.1.5 节）就是这样的例子，它们是 Module 的私有方法，但是可以像关键字一样被用在类

定义中。它们接受属性名称作为参数，并且通过这些名字动态创建方法。下面的例子是这些创建属性访问方法的变种，用来演示两种不同的动态创建方法的技术。

8.10.1 用 class_eval 定义方法

Defining Methods with class_eval

示例 8-6 为 Module 定义了两个名为 readonly 和 readwrite 的私有方法。这两个方法的工作与 attr_reader、attr_accessor 完成的工作相似，定义它们是为了演示其实现方式。它们的实现实际上相当简单：readonly 和 readwrite 首先构造一个表示一段 Ruby 代码的字符串，它包含定义相应访问器方法的 def 语句；接着，用 class_eval（本章前面介绍过）对这段代码进行求值。这样使用 class_eval 会带来少许解析代码的开销，不过，这样做的好处是我们定义的这些方法自身无须使用任何反射 API，它们可以直接查询和设置一个实例变量。

示例 8-6：用 class_eval 实现属性方法

```
class Module
  private    # The methods that follow are both private

  # This method works like attr_reader, but has a shorter name
  def readonly(*syms)
    return if syms.size == 0  # If no arguments, do nothing
    code = ""                 # Start with an empty string of code
    # Generate a string of Ruby code to define attribute reader methods.
    # Notice how the symbol is interpolated into the string of code.
    syms.each do |s|                        # For each symbol
      code << "def #{s}; @#{s}; end\n"      # The method definition
    end
    # Finally, class_eval the generated code to create instance methods.
    class_eval code
  end

  # This method works like attr_accessor, but has a shorter name.
  def readwrite(*syms)
    return if syms.size == 0
    code = ""
    syms.each do |s|
      code << "def #{s}; @#{s} end\n"
      code << "def #{s}=(value); @#{s} = value; end\n"
    end
    class_eval code
  end
end
```

8.10.2 用 define_method 定义方法

Defining Methods with define_method

示例 8-7 使用另一种技术实现属性访问器，它实现的 attributes 方法与示例 8-6 中的 readwrite 方法类似，不过它不是使用任意个数的属性名作为参数，而是以一个哈希对象

为参数。这个哈希表使用属性名为主键，以相应属性的默认值为对应值。class_attrs 方法与 attributes 类似，不过它定义的不是实例属性，而是类属性。

前面讲过当作为最后一个参数进行方法调用时，哈希表字面量的花括号可以省略，因此你可以用下面的方式调用 attributes 方法：

```
class Point
  attributes :x => 0, :y => 0
end
```

在 Ruby 1.9 中，你还可以使用更简洁的句法：

```
class Point
  attributes x:0, y:0
end
```

这是 Ruby 的灵活句法用于创建像关键字方法的另一例子。

示例 8-7 中 attributes 方法的实现与示例 8-6 中 readwrite 方法的实现颇有不同。attributes 方法没有使用 class_eval 对 Ruby 代码字符串进行求值，而是用一个代码块定义了该属性访问器的方法体，然后用 define_method 来定义这个方法。因为这种方法定义不允许直接在方法体中插入标识符，因此必须使用像 instance_variable_get 那样的反射技术。基于此，一般用 attributes 定义的属性不如用 readwrite 定义的有效率。

关于 attributes 方法，有趣的一点是它没有用任何显式的方式给出属性的默认值，实际上，每个属性的默认值可以在定义方法的代码块范围内被访问（参见第 6.6 节可以获得更多关于闭包的知识）。

定义类属性的 class_attrs 相当简单：它在该类的 eigenclass 上调用 attributes 方法，这意味着生成的方法会使用类实例变量（参见第 7.1.16 节）而非普通的类变量。

示例 8-7：用 define_method 定义属性方法

```
class Module
  # This method defines attribute reader and writer methods for named
  # attributes, but expects a hash argument mapping attribute names to
  # default values. The generated attribute reader methods return the
  # default value if the instance variable has not yet been defined.
  def attributes(hash)
    hash.each_pair do |symbol, default|     # For each attribute/default pair
      getter = symbol                       # Name of the getter method
      setter = :"#{symbol}="                # Name of the setter method
      variable = :"@#{symbol}"              # Name of the instance variable
      define_method getter do               # Define the getter method
        if instance_variable_defined? variable
```

```
        instance_variable_get variable    # Return variable, if defined
      else
        default                           # Otherwise return default
      end
    end

    define_method setter do |value|       # Define setter method
      instance_variable_set variable,     # Set the instance variable
                            value         # To the argument value
    end
  end
end

# This method works like attributes, but defines class methods instead
# by invoking attributes on the eigenclass instead of on self.
# Note that the defined methods use class instance variables
# instead of regular class variables.
def class_attrs(hash)
  eigenclass = class << self; self; end
  eigenclass.class_eval { attributes(hash) }
end

# Both methods are private
private :attributes, :class_attrs
end
```

8.11 别名链
Alias Chaining

正如我们已经看到的，Ruby 中的元编程经常涉及方法的动态定义，而对方法的动态**修改**也一样常见。方法定义通过我们称为**别名链**（*alias chaining*）（注 1）的技术实现，它的工作方式如下：

- 首先，创建一个要修改方法的别名。这个别名用作该方法未修改版本的名称。
- 接着，定义该方法的新版本。这个新方法应该通过别名调用该方法未修改的版本，不过在调用它之前或之后可以加入新的功能。

注意，这些步骤可以被重复应用（只要每次使用一个新的别名即可），这样就创建了一个方法和别名的链。

本节介绍了三个别名链的例子。第一个例子静态地使用了别名链，就是说使用了普通的 `alias` 和 `def` 语句，第二个和第三个例子要更加动态一些，通过使用 `alias_method`、`define_ method` 和 `class_eval` 方法，可以用别名链接任意的有名字的方法。

注 1：这也曾被称作**猴打补丁**（*monkey patching*），不过因为这个词最初是带着嘲笑的口吻说出的，我们在本书中不予使用。它有时也被幽默地称为**鸭式冲击**（*duck punching*）。

8.11.1 跟踪文件加载和类定义
Tracing Files Loaded and Classes Defined

示例 8-8 用于追踪程序加载的所有文件和定义的所有类。当程序退出时，它打印一个报表。你可以用这段代码来探视已有的程序，以更好地理解它做了些什么。使用这段代码的一种方式是在程序前部插入如下语句：

```
require 'classtrace'
```

不过，更简单的方式是对 Ruby 解释器使用-r 选项：

```
ruby -rclasstrace my_program.rb  --traceout /tmp/trace
```

-r 选项将在执行程序前加载指定的库。在第 10.1 节，你可以获得更多 Ruby 解释器命令行参数的知识。

示例 8-8 中，使用了静态别名链对所有 Kernel.require 和 Kernel.load 方法的调用进行跟踪，并定义了一个 Object.inherited 钩子方法跟踪所有新类的定义。当程序结束时，它利用 Kernel.at_exit 方法执行一个代码块。(在第 5.7 节中介绍的 END 语句也可以完成这项工作。) 除了对 require 和 load 使用别名链及定义了 Object.inherited 外，它在全局命名空间中做的唯一修改是定义了 ClassTrace 模块，跟踪所需的所有状态都在这个模块中被保存，这样可以不使用全局变量，从而不会污染全局命名空间。

示例 8-8：跟踪文件加载和类定义

```
# We define this module to hold the global state we require, so that
# we don't alter the global namespace any more than necessary.
module ClassTrace
  # This array holds our list of files loaded and classes defined.
  # Each element is a subarray holding the class defined or the
  # file loaded and the stack frame where it was defined or loaded.
  T = []  # Array to hold the files loaded

  # Now define the constant OUT to specify where tracing output goes.
  # This defaults to STDERR, but can also come from command-line arguments
  if x = ARGV.index("--traceout")      # If argument exists
    OUT = File.open(ARGV[x+1], "w")    # Open the specified file
    ARGV[x,2] = nil                    # And remove the arguments
  else
    OUT = STDERR                       # Otherwise default to STDERR
  end
end

# Alias chaining step 1: define aliases for the original methods
alias original_require require
alias original_load load

# Alias chaining step 2: define new versions of the methods
def require(file)
  ClassTrace::T << [file,caller[0]]   # Remember what was loaded where
  original_require(file)              # Invoke the original method
end
```

```
def load(*args)
  ClassTrace::T << [args[0],caller[0]]     # Remember what was loaded where
  original_load(*args)                     # Invoke the original method
end

# This hook method is invoked each time a new class is defined
def Object.inherited(c)
  ClassTrace::T << [c,caller[0]]           # Remember what was defined where
end

# Kernel.at_exit registers a block to be run when the program exits
# We use it to report the file and class data we collected
at_exit {
  o = ClassTrace::OUT
  o.puts "="*60
  o.puts "Files Loaded and Classes Defined:"
  o.puts "="*60
  ClassTrace::T.each do |what,where|
    if what.is_a? Class   # Report class (with hierarchy) defined
      o.puts "Defined:    #{what.ancestors.join('<-')} at #{where}"
    else                  # Report file loaded
      o.puts "Loaded:     #{what} at #{where}"
    end
  end
}
```

8.11.2 为线程安全链接方法

Chaining Methods for Thread Safety

本章前面的两个例子涉及了线程安全。示例 8-2 定义了一个 synchronized 方法（基于 Object.mutex 方法），它在一个 Mutex 对象的保护下执行一个代码块。接着，示例 8-5 重定义了这个 synchronized 方法，在调用时如果没有关联代码块，它会返回一个 SynchronizedObject 对象包装一个对象，通过这个包装对象，你可以保护对任意方法的访问。现在，在示例 8-9 中，我们要再一次扩展 synchronized 方法，当它在类或模块定义中被调用时，它会用别名链方式为给定方法加入同步机制。

别名链通过 Module.synchronize_method 方法实现，这个方法使用一个辅助方法 Module.create_alias 为任意给定方法（也包括像+操作符）定义一个恰当的别名。

在定义了这些新的 Module 方法后，示例 8-9 再次重定义了 synchronized 方法。当这个方法在一个类或模块中被调用时，它对每个传入的符号调用 synchronize_method 方法。不过，有趣的是它也可以用无参方式调用。这时，它为后面定义的所有方法加入同步机制。（它使用 method_added 钩子方法，在新方法加入时会得到通知。）注意这个例子中的代码使用了示例 8-2 中定义的 Object.mutex 方法及示例 8-5 中定义的 SynchronizedObject 类。

示例 8-9：用于线程安全的别名链

```
# Define a Module.synchronize_method that alias chains instance methods
# so they synchronize on the instance before running.
class Module
  # This is a helper function for alias chaining.
  # Given a method name (as a string or symbol) and a prefix, create
  # a unique alias for the method, and return the name of the alias
  # as a symbol. Any punctuation characters in the original method name
  # will be converted to numbers so that operators can be aliased.
  def create_alias(original, prefix="alias")
    # Stick the prefix on the original name and convert punctuation
    aka = "#{prefix}_#{original}"
    aka.gsub!(/([\=\|\&\+\-\*\/\^\!\?\~\%\<\>\[\]])/) {
      num = $1[0]                          # Ruby 1.8 character -> ordinal
      num = num.ord if num.is_a? String    # Ruby 1.9 character -> ordinal
      '_' + num.to_s
    }

    # Keep appending underscores until we get a name that is not in use
    aka += "_" while method_defined? aka or private_method_defined? aka

    aka = aka.to_sym                 # Convert the alias name to a symbol
    alias_method aka, original       # Actually create the alias
    aka                              # Return the alias name
  end

  # Alias chain the named method to add synchronization
  def synchronize_method(m)
    # First, make an alias for the unsynchronized version of the method.
    aka = create_alias(m, "unsync")
    # Now redefine the original to invoke the alias in a synchronized block.
    # We want the defined  method to be able to accept blocks, so we
    # can't use define_method, and must instead evaluate a string with
    # class_eval. Note that everything between %Q{ and the matching }
    # is a double-quoted string, not a block.
    class_eval %Q{
      def #{m}(*args, &block)
        synchronized(self) { #{aka}(*args, &block) }
      end
    }
  end
end

# This global synchronized method can now be used in three different ways.
def synchronized(*args)
  # Case 1: with one argument and a block, synchronize on the object
  # and execute the block
  if args.size == 1 && block_given?
    args[0].mutex.synchronize { yield }

  # Case two: with one argument that is not a symbol and no block
  # return a SynchronizedObject wrapper
  elsif args.size == 1 and not args[0].is_a? Symbol and not block_given?
    SynchronizedObject.new(args[0])
```

```
  # Case three: when invoked on a module with no block, alias chain the
  # named methods to add synchronization. Or, if there are no arguments,
  # then alias chain the next method defined.
  elsif self.is_a? Module and not block_given?
    if (args.size > 0) # Synchronize the named methods
      args.each {|m| self.synchronize_method(m) }
    else
      # If no methods are specified synchronize the next method defined
      eigenclass = class<<self; self; end
      eigenclass.class_eval do # Use eigenclass to define class methods
        # Define method_added for notification when next method is defined
        define_method :method_added do |name|
          # First remove this hook method
          eigenclass.class_eval { remove_method :method_added }
          # Next, synchronize the method that was just added
          self.synchronize_method name
        end
      end
    end

  # Case 4: any other invocation is an error
  else
    raise ArgumentError, "Invalid arguments to synchronize()"
  end
end
```

8.11.3 用方法链实现跟踪
Chaining Methods for Tracing

示例 8-10 是示例 8-4 的变体，它们都支持跟踪一个对象有名方法的调用。示例 8-4 中，使用了方法代理和 method_missing 方法定义一个 Object.trace 方法，它能返回一个可以跟踪的包装对象。而示例 8-10 则使用了方法链修改对象的方法，它定义了 trace! 和 untrace! 方法链接有名字的方法及取消链接。

本例的有趣之处在于它使用了与示例 8-9 不同的一种链接方式，它只是简单地为指定对象定义单键方法，然后在单键方法中使用 super 调用实例方法的原始定义，而没有创建任何别名方法。

示例 8-10：用单键方法实现链式调用进行跟踪

```
# Define trace! and untrace! instance methods for all objects.
# trace! "chains" the named methods by defining singleton methods
# that add tracing functionality and then use super to call the original.
# untrace! deletes the singleton methods to remove tracing.
class Object
  # Trace the specified methods, sending output to STDERR.
  def trace!(*methods)
    @_traced = @_traced || []    # Remember the set of traced methods

    # If no methods were specified, use all public methods defined
    # directly (not inherited) by the class of this object
    methods = public_methods(false) if methods.size == 0
```

```
  methods.map! {|m| m.to_sym }  # Convert any strings to symbols
  methods -= @_traced           # Remove methods that are already traced
  return if methods.empty?      # Return early if there is nothing to do
  @_traced |= methods           # Add methods to set of traced methods

  # Trace the fact that we're starting to trace these methods
  STDERR << "Tracing #{methods.join(', ')} on #{object_id}\n"

  # Singleton methods are defined in the eigenclass
  eigenclass = class << self; self; end

  methods.each do |m|         # For each method m
    # Define a traced singleton version of the method m.
    # Output tracing information and use super to invoke the
    # instance method that it is tracing.
    # We want the defined  methods to be able to accept blocks, so we
    # can't use define_method, and must instead evaluate a string.
    # Note that everything between %Q{ and the matching } is a
    # double-quoted string, not a block. Also note that there are
    # two levels of string interpolations here. #{} is interpolated
    # when the singleton method is defined. And \#{} is interpolated
    # when the singleton method is invoked.
    eigenclass.class_eval %Q{
      def #{m}(*args, &block)
        begin
          STDERR << "Entering: #{m}(\#{args.join(', ')})\n"
          result = super
          STDERR << "Exiting: #{m} with \#{result}\n"
          result
        rescue
          STDERR << "Aborting: #{m}: \#{$!.class}: \#{$!.message}"
          raise
        end
      end
    }
  end
end

# Untrace the specified methods or all traced methods
def untrace!(*methods)
  if methods.size == 0        # If no methods specified untrace
    methods = @_traced        # all currently traced methods
    STDERR << "Untracing all methods on #{object_id}\n"
  else                        # Otherwise, untrace
    methods.map! {|m| m.to_sym }  # Convert string to symbols
    methods &= @_traced   # all specified methods that are traced
    STDERR << "Untracing #{methods.join(', ')} on #{object_id}\n"
  end

  @_traced -= methods     # Remove them from our set of traced methods

  # Remove the traced singleton methods from the eigenclass
  # Note that we class_eval a block here, not a string
  (class << self; self; end).class_eval do
```

```
      methods.each do |m|
        remove_method m     # undef_method would not work correctly
      end
    end

    # If no methods are traced anymore, remove our instance var
    if @_traced.empty?
      remove_instance_variable :@_traced
    end
  end
end
```

8.12 领域特定语言
Domain-Specific Languages

Ruby 元编程的一个常见目的是创建**领域特定语言**（*domain-specific language*, DSL）。一个 DSL 不过是对 Ruby 句法（通过看起来像关键字的方法）或 API 的扩展，它们可以让你用更自然的方式处理问题或表现数据。在下面的例子中，问题领域是输出 XML 格式的数据，我们将定义两种 DSL 来解决这一问题，其中一种很简单，而另外一种则更聪明（注 2）。

8.12.1 用 method_missing 实现简单的 XML 输出
Simple XML Output with method_missing

首先我们创建一个简单的、名为 XML 的类，用于创建 XML 输出。下面演示了这个 XML 类将如何被使用：

```
pagetitle = "Test Page for XML.generate"
XML.generate(STDOUT) do
  html do
    head do
      title { pagetitle }
      comment "This is a test"
    end
    body do
      h1(:style => "font-family:sans-serif") { pagetitle }
      ul :type=>"square" do
        li { Time.now }
        li { RUBY_VERSION }
      end
    end
  end
end
```

这段代码看起来不像 XML，而更像 Ruby。下面是它的输出（为清晰起见，加入了一些换行符）：

```
<html><head>
<title>Test Page for XML.generate</title>
<!-- This is a test -->
```

注 2：Jim Weirich 的 Builder API 提供了一种完整的解决方案。参见 *http://builder.rubyforge.org*。

```
</head><body>
<h1 style='font-family:sans-serif'>Test Page for XML.generate</h1>
<ul type='square'>
<li>2007-08-19 16:19:58 -0700</li>
<li>1.9.0</li>
</ul></body></html>
```


要实现这个类和它支持的 XML 生成句法，我们须要依赖：

- Ruby 的代码块结构；
- Ruby 在方法调用时可选的花括号；
- Ruby 在为方法传入哈希表字面量时可以省略花括号。
- `method_missing` 方法。

示例 8-11 显示了如何实现这样一个简单的 DSL。

示例 8-11：用于产生 XML 输出的简单 DSL

```
class XML
  # Create an instance of this class, specifying a stream or object to
  # hold the output. This can be any object that responds to <<(String).
  def initialize(out)
    @out = out  # Remember where to send our output
  end

  # Output the specified object as CDATA, return nil.
  def content(text)
    @out << text.to_s
    nil
  end

  # Output the specified object as a comment, return nil.
  def comment(text)
    @out << "<!-- #{text} -->"
    nil
  end

  # Output a tag with the specified name and attributes.
  # If there is a block invoke it to output or return content
  # Return nil.
  def tag(tagname, attributes={})
    # Output the tag name
    @out << "<#{tagname}"

    # Output the attributes
    attributes.each {|attr,value| @out << " #{attr}='#{value}'" }

    if block_given?
      # This block has content
      @out << '>'              # End the opening tag
      content = yield          # Invoke the block to output or return content
      if content               # If any content returned
        @out << content.to_s   # Output it as a string
      end
      @out << "</#{tagname}>" # Close the tag
```

```
    else
      # Otherwise, this is an empty tag, so just close it.
      @out << '/>'
    end
    nil # Tags output themselves, so they don't return any content
  end

  # The code below is what changes this from an ordinary class into a DSL.
  # First: any unknown method is treated as the name of a tag.
  alias method_missing tag

  # Second: run a block in a new instance of the class.
  def self.generate(out, &block)
    XML.new(out).instance_eval(&block)
  end
end
```

8.12.2 用方法生成方式验证 XML 输出

Validated XML Output with Method Generation

示例 8-11 中的 XML 类对于生成格式良好的 XML 文档很有用，但是它没有任何错误检查机制来保证生成的 XML 符合任意给定的 XML 语法。在下一个例子（示例 8-12）中，会加入简单的错误检查机制（尽管它不能保证完全的有效性——那需要更多的代码）。该示例实际上是把两个 DSL 融入到一个 DSL 中，第一个 DSL 用于定义 XML 语法：它定义一组标签，以及每个标签可用的属性。你可以以如下方式使用它：

```
class HTMLForm < XMLGrammar
  element :form, :action => REQ,
                 :method => "GET",
                 :enctype => "application/x-www-form-urlencoded",
                 :name => OPT
  element :input, :type => "text", :name => OPT, :value => OPT,
                  :maxlength => OPT, :size => OPT, :src => OPT,
                  :checked => BOOL, :disabled => BOOL, :readonly => BOOL
  element :textarea, :rows => REQ, :cols => REQ, :name => OPT,
                     :disabled => BOOL, :readonly => BOOL
  element :button, :name => OPT, :value => OPT,
                   :type => "submit", :disabled => OPT
end
```

第一个 DSL 通过类方法 XMLGrammar.element 来定义，要使用它，需要子类化 XMLGrammar 来创建一个新类。element 方法的第一个参数是标签名，第二个参数是合法的属性，用一个哈希表对象表示。这个哈希表对象的主键是属性名，它们可以映射属性的默认值，也可用映射常量 REQ 表示必需属性，或者用常量 OPT 表示可选属性。调用 element 方法会为你定义的子类生成一个给定名字的方法。

新定义的 XMLGrammar 子类是第二个 DSL，通过它，你可以创建符合给定规则的 XML 输

出。XMLGrammar 类没有定义 method_missing 方法，因此不允许创建一个不在语法范围内的标签。用于输出标签的 tag 方法会对属性进行错误检查。对语法生成子类的使用方式与示例 8-11 中的 XML 类相似：

```
HTMLForm.generate(STDOUT) do
  comment "This is a simple HTML form"
  form :name => "registration",
      :action => "http://www.example.com/register.cgi" do
    content "Name:"
    input :name => "name"
    content "Address:"
    textarea :name => "address", :rows=>6, :cols=>40 do
      "Please enter your mailing address here"
    end
    button { "Submit" }
  end
end
```

示例 8-12 显示了 XMLGrammar 类的实现。

示例 8-12：用于验证 XML 输出的 DSL

```
class XMLGrammar
  # Create an instance of this class, specifying a stream or object to
  # hold the output. This can be any object that responds to <<(String).
  def initialize(out)
    @out = out  # Remember where to send our output
  end

  # Invoke the block in an instance that outputs to the specified stream.
  def self.generate(out, &block)
    new(out).instance_eval(&block)
  end

  # Define an allowed element (or tag) in the grammar.
  # This class method is the grammar-specification DSL
  # and defines the methods that constitute the XML-output DSL.
  def self.element(tagname, attributes={})
    @allowed_attributes ||= {}
    @allowed_attributes[tagname] = attributes

    class_eval %Q{
      def #{tagname}(attributes={}, &block)
        tag(:#{tagname},attributes,&block)
      end
    }
  end

  # These are constants used when defining attribute values.
  OPT = :opt      # for optional attributes
  REQ = :req      # for required attributes
  BOOL = :bool    # for attributes whose value is their own name

  def self.allowed_attributes
    @allowed_attributes
  end
```

```
# Output the specified object as CDATA, return nil.
def content(text)
  @out << text.to_s
  nil
end

# Output the specified object as a comment, return nil.
def comment(text)
  @out << "<!-- #{text} -->"
  nil
end

# Output a tag with the specified name and attribute.
# If there is a block, invoke it to output or return content.
# Return nil.
def tag(tagname, attributes={})
  # Output the tag name
  @out << "<#{tagname}"

  # Get the allowed attributes for this tag.
  allowed = self.class.allowed_attributes[tagname]

  # First, make sure that each of the attributes is allowed.
  # Assuming they are allowed, output all of the specified ones.
  attributes.each_pair do |key,value|
    raise "unknown attribute: #{key}" unless allowed.include?(key)
    @out << " #{key}='#{value}'"
  end

  # Now look through the allowed attributes, checking for
  # required attributes that were omitted and for attributes with
  # default values that we can output.
  allowed.each_pair do |key,value|
    # If this attribute was already output, do nothing.
    next if attributes.has_key? key
    if (value == REQ)
      raise "required attribute '#{key}' missing in <#{tagname}>"
    elsif value.is_a? String
      @out << " #{key}='#{value}'"
    end
  end

  if block_given?
    # This block has content
    @out << '>'              # End the opening tag
    content = yield          # Invoke the block to output or return content
    if content               # If any content returned
      @out << content.to_s   # Output it as a string
    end
    @out << "</#{tagname}>" # Close the tag
  else
    # Otherwise, this is an empty tag, so just close it.
    @out << '/>'
  end
```

```
    nil # Tags output themselves, so they don't return any content.
  end
end
```

第9章 Ruby平台

The Ruby Platform

Ruby 内核库定义了很多强大的 API，可以作为编程平台，它们值得你花费一些时间来学习和掌握，尤其是那些关键的类，比如 String、Array、Hash、Enumerable 和 IO。如果不熟悉这些类中定义的方法，你可能会重新发明轮子。

本章对这些方法进行了说明，我们不指望它能成为一个 API 的详尽参考，只是希望能用一些代码片段演示如何使用这些重要类和模块的重要方法。这些类和模块大多来自核心库，也有少量来自标准库的常用类。本章的目的是让你能对已存在的大量方法有所了解，须要使用时你可记得它们的位置，而且可以用 *ri* 获得文档。

本章内容很多，我们按照如下内容划分了小节：

- 字符串和文本处理；
- 正则表达式；
- 数字和数学运算；
- 日期和时间；
- Enumerable 模块及 Array、Hash 和 Set 集合；
- 输入/输出和文件；
- 网络；
- 线程和并发。

在本章开始的时候，那些代码片段经常只用一行演示一个方法使用，但是在后面讲述网络和线程时，示例代码就变得很长。这些示例代码将展示如何完成像创建一个网络客户端这样的任务，或者如何用线程来并发处理一个集合中的元素。

9.1 字符串
Strings

第 3 章中已经解释了 Ruby 字符串的句法及字符串的连接（+）、追加（<<）、重复（*）和索引（[]）操作符，本节将介绍 String 类的有名字的方法，其后的相关章节详尽介绍某些特殊主题。

首先来看看在第 3 章中操作符的替代方法：

```
s = "hello"
s.concat(" world")    # Synonym for <<. Mutating append to s. Returns new s.
s.insert(5, " there") # Same as s[5] = " there". Alters s. Returns new s.
s.slice(0,5)          # Same as s[0,5]. Returns a substring.
s.slice!(5,6)         # Deletion. Same as s[5,6]="". Returns deleted substring.
s.eql?("hello world") # True. Same as ==.
```

有好几个方法可以用于获得字符串的长度：

```
s.length          # => 5: counts characters in 1.9, bytes in 1.8
s.size            # => 5: size is a synonym
s.bytesize        # => 5: length in bytes; Ruby 1.9 only
s.empty?          # => false
"".empty?         # => true
```

下面的 String 方法用于查找和替换文本。在后面介绍完正则表达式后，你应该重新看看它们：

```
s = "hello"
# Finding the position of a substring or pattern match
s.index('l')      # => 2: index of first l in string
s.index(?l)       # => 2: works with character codes as well
s.index(/l+/)     # => 2: works with regular expressions, too
s.index('l',3)    # => 3: index of first l in string at or after position 3
s.index('Ruby')   # => nil: search string not found
s.rindex('l')     # => 3: index of rightmost l in string
s.rindex('l',2)   # => 2: index of rightmost l in string at or before 2

# Checking for prefixes and suffixes: Ruby 1.9 and later
s.start_with? "hell" # => true.  Note singular "start" not "starts"
s.end_with? "bells"  # => false

# Testing for presence of substring
s.include?("ll")      # => true: "hello" includes "ll"
s.include?(?H)        # => false: "hello" does not include character H

# Pattern matching with regular expressions
s =~ /[aeiou]{2}/   # => nil: no double vowels in "hello"
s.match(/[aeiou]/) {|m| m.to_s} # => "e": return first vowel

# Splitting a string into substrings based on a delimiter string or pattern
"this is it".split        # => ["this", "is", "it"]: split on spaces by default
"hello".split('l')        # => ["he", "", "o"]
"1, 2,3".split(/,\s*/)    # => ["1","2","3"]: comma and optional space delimiter

# Split a string into two parts plus a delimiter. Ruby 1.9 only.
# These methods always return arrays of 3 strings:
"banana".partition("an") # => ["b", "an", "ana"]
"banana".rpartition("an")# => ["ban", "an", "a"]: start from right
"a123b".partition(/\d+/) # => ["a", "123", "b"]: works with Regexps, too

# Search and replace the first (sub, sub!) or all (gsub, gsub!)
# occurrences of the specified string or pattern.
# More about sub and gsub when we cover regular expressions later.
s.sub("l", "L")               # => "heLlo": Just replace first occurrence
s.gsub("l", "L")              # => "heLLo": Replace all occurrences
s.sub!(/(.)(.)/, '\2\1')      # => "ehllo": Match and swap first 2 letters
s.sub!(/(.)(.)/, "\\2\\1")    # => "hello": Double backslashes for double quotes

# sub and gsub can also compute a replacement string with a block
# Match the first letter of each word and capitalize it
"hello world".gsub(/\b./) {|match| match.upcase } # => "Hello World"
```

本例的最后一行使用了 upcase 方法，把一个字符串转换为大写方式。String 类定义了一组用于处理大小写的方法（但是没有提供用于测试一个字符大小写和种类的方法）：

```
# Case modification methods
s = "world"       # These methods work with ASCII characters only
s.upcase          # => "WORLD"
s.upcase!         # => "WORLD"; alter s in place
s.downcase        # => "world"
s.capitalize      # => "World": first letter upper, rest lower
s.capitalize!     # => "World": alter s in place
s.swapcase        # => "wORLD": alter case of each letter

# Case insensitive comparison. (ASCII text only)
# casecmp works like <=> and returns -1 for less, 0 for equal, +1 for greater
"world".casecmp("WORLD")      # => 0
"a".casecmp("B")              # => -1 (<=> returns 1 in this case)
```

String 定义了一组增加和删除空白符的方法，绝大多数方法都有可变（用!结尾）和不变两个版本：

```
s = "hello\r\n"       # A string with a line terminator
s.chomp!              # => "hello": remove one line terminator from end
s.chomp               # => "hello": no line terminator so no change
s.chomp!              # => nil: return of nil indicates no change made
s.chomp("o")          # => "hell": remove "o" from end
$/ = ";"              # Set global record separator $/ to semicolon
"hello;".chomp        # => "hello": now chomp removes semicolons and end

# chop removes trailing character or line terminator (\n, \r, or \r\n)
s = "hello\n"
s.chop!               # => "hello": line terminator removed. s modified.
s.chop                # => "hell": last character removed. s not modified.
"".chop               # => "": no characters to remove
"".chop!              # => nil: nothing changed

# Strip all whitespace (including \t, \r, \n) from left, right, or both
# strip!, lstrip! and rstrip! modify the string in place.
s = "\t hello \n"     # Whitespace at beginning and end
s.strip               # => "hello"
s.lstrip              # => "hello \n"
s.rstrip              # => "\t hello"

# Left-justify, right-justify, or center a string in a field n-characters wide.
# There are no mutator versions of these methods. See also printf method.
s = "x"
s.ljust(3)            # => "x  "
s.rjust(3)            # => "  x"
s.center(3)           # => " x "
s.center(5, '-')      # => "--x--": padding other than space are allowed
s.center(7, '-=')     # => "-=-x-=-": multicharacter padding allowed
```

字符串可以用字节和行两种方式进行枚举，分别用 each_byte 和 each_line 两个迭代器实现。在 Ruby 1.8 中，each 方法是 each_line 的同义词，而且 Ruby 类包含了 Enumerable

模块。不过要避免使用 each 迭代器，因为在 Ruby 1.9 中删除了 each 方法，并且不再包含 Enumerable 模块。Ruby 1.9（及 Ruby 1.8 的 jcode 库）定义了一个 each_char 迭代器，用于进行字符方式的枚举：

```
s = "A\nB"                          # Three ASCII characters on two lines
s.each_byte {|b| print b, " " }     # Prints "65 10 66 "
s.each_line {|l| print l.chomp}     # Prints "AB"

# Sequentially iterate characters as 1-character strings
# Works in Ruby 1.9, or in 1.8 with the jcode library:
s.each_char { |c| print c, " " } # Prints "A \n B "

# Enumerate each character as a 1-character string
# This does not work for multibyte strings in 1.8
# It works (inefficiently) for multibyte strings in 1.9:
0.upto(s.length-1) {|n| print s[n,1], " "}

# In Ruby 1.9, bytes, lines, and chars are aliases
s.bytes.to_a                    # => [65,10,66]: alias for each_byte
s.lines.to_a                    # => ["A\n","B"]: alias for each_line
s.chars.to_a                    # => ["A", "\n", "B"] alias for each_char
```

String 定义了一组用于从字符串解析数字的方法，以及用于把字符串转换为符号的方法：

```
"10".to_i               # => 10: convert string to integer
"10".to_i(2)            # => 2: argument is radix: between base-2 and base-36
"10x".to_i              # => 10: nonnumeric suffix is ignored. Same for oct, hex
" 10".to_i              # => 10: leading whitespace ignored
"ten".to_i              # => 0: does not raise exception on bad input
"10".oct                # => 8: parse string as base-8 integer
"10".hex                # => 16: parse string as hexadecimal integer
"0xff".hex              # => 255: hex numbers may begin with 0x prefix
" 1.1 dozen".to_f       # => 1.1: parse leading floating-point number
"6.02e23".to_f          # => 6.02e+23: exponential notation supported

"one".to_sym            # => :one -- string to symbol conversion
"two".intern            # => :two -- intern is a synonym for to_sym
```

最后，让我们看看 String 的一些其他方法：

```
# Increment a string:
"a".succ                        # => "b": the successor of "a". Also, succ!
"aaz".next                      # => "aba": next is a synonym. Also, next!
"a".upto("e") {|c| print c } # Prints "abcde. upto iterator based on succ.

# Reverse a string:
"hello".reverse       # => "olleh". Also reverse!

# Debugging
"hello\n".dump        # => "\"hello\\n\"": Escape special characters
"hello\n".inspect     # Works much like dump

# Translation from one set of characters to another
"hello".tr("aeiou", "AEIOU") # => "hEllO": capitalize vowels. Also tr!
"hello".tr("aeiou", " ")     # => "h ll ": convert vowels to spaces
```

```
"bead".tr_s("aeiou", " ")     # => "b d": convert and remove duplicates

# Checksums
"hello".sum           # => 532: weak 16-bit checksum
"hello".sum(8)        # => 20: 8 bit checksum instead of 16 bit
"hello".crypt("ab")   # => "ablOJrMf6tlhw": one way cryptographic checksum
                      # Pass two alphanumeric characters as "salt"
                      # The result may be platform-dependent

# Counting letters, deleting letters, and removing duplicates
"hello".count('aeiou')    # => 2: count lowercase vowels
"hello".delete('aeiou')   # => "hll": delete lowercase vowels. Also delete!
"hello".squeeze('a-z')    # => "helo": remove runs of letters. Also squeeze!
# When there is more than one argument, take the intersection.
# Arguments that begin with ^ are negated.
"hello".count('a-z', '^aeiou')    # => 3: count lowercase consonants
"hello".delete('a-z', '^aeiou')   # => "eo: delete lowercase consonants
```

9.1.1 格式化文本
Formatting Text

如你所知，用双引号括住的字符串字面量中可以使用各种 Ruby 表达式，它们的值会被插入字符串中，例如：

```
n, animal = 2, "mice"
"#{n+1} blind #{animal}"  # => '3 blind mice'
```

在第 3 章中描述了这种对字符串字面量插入表达式的句法。Ruby 也支持其他一些对字符串插入值的方式：`String` 类定义了一个格式操作符`%`，`Kernel` 模块定义了全局 `printf` 和 `sprintf` 方法。这些方法和操作符与 C 语言的 `printf` 方法很相似。这种方式优于普通字面量插入方式之处在于可以精确控制字段的宽度，这对生成 ASCII 方式的报表很有用处。它的另一个优点是可以显示浮点数，这对科学计算应用（及金融应用）很有用处。最后，`printf` 方式的格式化把字符串和要插入的值解耦开，这对应用程序的国际化和本地化有好处。

下面是`%`操作符的例子。在 `Kernel.sprintf` 的文档中可以找到这些方法格式化指示符的详尽文档：

```
# Alternatives to the interpolation above
printf('%d blind %s', n+1, animal)    # Prints '3 blind mice', returns nil
sprintf('%d blind %s', n+1, animal)   # => '3 blind mice'
'%d blind %s' % [n+1, animal]  # Use array on right if more than one argument

# Formatting numbers
'%d' % 10         # => '10': %d for decimal integers
'%x' % 10         # => 'a': hexadecimal integers
'%X' % 10         # => 'A': uppercase hexadecimal integers
'%o' % 10         # => '12': octal integers
```

```
'%f' % 1234.567       # => '1234.567000': full-length floating-point numbers
'%e' % 1234.567       # => '1.234567e+03': force exponential notation
'%E' % 1234.567       # => '1.234567e+03': exponential with uppercase E
'%g' % 1234.567       # => '1234.57': six significant digits
'%g' % 1.23456E12     # => '1.23456e+12': Use %f or %e depending on magnitude

# Field width
'%5s' % '<<<'         # '  <<<': right-justify in field five characters wide
'%-5s' % '>>>'        # '>>>  ': left-justify in field five characters wide
'%5d' % 123           # '  123': field is five characters wide
'%05d' % 123          # '00123': pad with zeros in field five characters wide

# Precision
'%.2f' % 123.456      # '123.46': two digits after decimal place
'%.2e' % 123.456      # '1.23e+02': two digits after decimal = three significant digits
'%.6e' % 123.456      # '1.234560e+02': note added zero
'%.4g' % 123.456      # '123.5': four significant digits

# Field and precision combined
'%6.4g' % 123.456     # ' 123.5': four significant digits in field six chars wide
'%3s' % 'ruby'        # 'ruby': string argument exceeds field width
'%3.3s' % 'ruby'      # 'rub': precision forces truncation of string

# Multiple arguments to be formatted
args = ['Syntax Error', 'test.rb', 20]  # An array of arguments
"%s: in '%s' line %d" % args # => "Syntax Error: in 'test.rb' line 20"
# Same args, interpolated in different order!  Good for internationalization.
"%2$s:%3$d: %1$s" % args      # => "test.rb:20: Syntax Error"
```

9.1.2 打包和解包二进制字符串
Packing and Unpacking Binary Strings

Ruby 字符串不仅可以存放文本数据，也可以存放二进制数据。在使用二进制格式文件或二进制网络协议时，你可能会用到 `Array.pack` 和 `String.unpack` 这一对方法。使用 `Array.pack` 方法可以把数组中的元素编码为一个二进制数组，而使用 `String.unpack` 可以解码一个二进制字符串，提取其中的数值并把这些数值作为一个数组返回。这些编码和解码操作都在一个格式化字符串的控制之下，这个字符串中的字母指明编码的数据类型，而其中的数字指明重复的次数。这个格式化的创建方式相当神秘，你可以在 `Array.pack` 和 `String.unpack` 的文档中找到这些字母的详尽列表。下面是一些简单的例子：

```
a = [1,2,3,4,5,6,7,8,9,10]   # An array of 10 integers
b = a.pack('i10')             # Pack 10 4-byte integers (i) into binary string b
c = b.unpack('i*')            # Decode all (*) the 4-byte integers from b
c == a                        # => true

m = 'hello world'             # A message to encode
data = [m.size, m]            # Length first, then the bytes
template = 'Sa*'              # Unsigned short, any number of ASCII chars
b = data.pack(template)       # => "\v\000hello world"
b.unpack(template)            # => [11, "hello world"]
```

9.1.3 字符串和编码
Strings and Encodings

在第 3.2.6 节中描述了 String 的 encoding、encode、encode!和 force_encoding 方法及 Encoding 类。如果想在程序中使用 Unicode 或其他方式的多字节编码，你可能要重读那一小节。

9.2 正则表达式
Regular Expressions

正则表达式（也叫 regexp 或 regex）描述了一个文本模式。Ruby 用 Regexp 类（注 1）来实现正则表达式，Regexp 和 String 类都定义了一些模式匹配方法和操作符。就像绝大多数支持正则表达式的语言一样，Ruby 的 Regexp 语法基本遵循 Perl 5 的语法（但是不完全相同）。

9.2.1 Regexp 字面量
Regexp Literals

正则表达式字面量用斜线符分隔：

```
/Ruby?/  # Matches the text "Rub" followed by an optional "y"
```

后面的斜线符在严格意义上并非真正的分隔符，因为一个正则表达式字面量后面可以跟一个或多个可选的标识字符，用于指定如何进行模式匹配。例如：

```
/ruby?/i  # Case-insensitive: matches "ruby" or "RUB", etc.
/./mu     # Matches Unicode characters in Multiline mode
```

表 9-1 列出了可用的修饰符。

表 9-1：正则表达式修饰符

修饰符	描述
i	匹配文本时忽略大小写
m	跨行进行模式匹配。这样换行符被当作普通字符对待：.可以匹配换行符
x	扩展句法：允许在正则表达式中放入空白符和注释
o	对#{}这样的插入仅仅在对正则表达式字面量第一次求值时被执行一次
u,e,s,n	把正则表达式解释为 Unicode (UTF-8)、EUC、SJIS 或 ASCII。如果没有这样的修饰符，则对正则表达式使用源文件的编码方式

正如字符串字面量可以用%Q 分隔一样，Ruby 的正则表达式也可以用%r 后跟一个分隔符来定义。当正则表达式中有许多斜杠符时，如果不想使用转义符，这会很有用处：

注 1：JavaScript 程序员需要注意，Ruby 的这个类用的是小写的 e，这个跟 JavaScript 的 RegExp 类不同。

```
%r|/|            # Matches a single slash character, no escape required
%r[</(.*)>]i     # Flag characters are allowed with this syntax, too
```

在正则表达式中，()、[]、{}、.、?、+、*、|、^和$这些字符有特殊含义。如果某个待匹配的模式中包含这些字符的字面量，要用反斜杠进行转义。如果某个模式中包含反斜杠，需要用两个反斜杠表示：

```
/\(\)/        # Matches open and close parentheses
/\\/          # Matches a single backslash
```

正则表达式字面量的行为与双引号括住的字符串字面量类似，可以包含如\n、\t和（在Ruby 1.9中）\u这样的转义字符（在第3章的表3-1可以查到完整的转义序列列表）：

```
money = /[$\u20AC\u{a3}\u{a5}]/ # match dollar,euro,pound, or yen sign
```

另外一点与双引号的字符串字面量相似的，是正则表达式字面量也可以用#{}句法插入任意的Ruby表达式：

```
prefix = ","
/#{prefix}\t/   # Matches a comma followed by an ASCII TAB character
```

注意插入行为是在正则表达式解析之前就完成了的，这意味着插入表达式中的特殊字符会成为正则表达式句法的一部分。通常，插入表达式在每次对正则表达式字面量求值前被执行一次，不过，如果使用了 o 修饰符，这个插入表达式仅仅在第一次解析时被执行。下面的例子很好地演示了o修饰符的行为：

```
[1,2].map{|x| /#{x}/}     # => [/1/, /2/]
[1,2].map{|x| /#{x}/o}    # => [/1/, /1/]
```

9.2.2 Regexp的工厂方法
Regexp Factory Methods

你也可以不使用正则表达式的字面量，而直接使用 Regexp.new（或它的同义词：Regexp.compile）来创建一个正则表达式：

```
Regexp.new("Ruby?")                              # /Ruby?/
Regexp.new("ruby?", Regexp::IGNORECASE)          # /ruby?/i
Regexp.compile(".", Regexp::MULTILINE, "u")      # /./mu
```

在把一个字符串传给 Regexp 的构造方法前，你可以用 Regexp.escape 对字符串中的特殊字符进行转义：

```
pattern = "[a-z]+"                # One or more letters
suffix = Regexp.escape("()")      # Treat these characters literally
r = Regexp.new(pattern + suffix)  # /[a-z]+\(\)/
```

在Ruby 1.9中，工厂方法Regexp.union可以创建一个“联合”了任意多字符串或Regexp对象的模式。（也就是说，如果一个字符串可以匹配其中任意一个，生成的模式就可以匹配它。）它可以接受任意多个字符串或模式作为参数，也可以把它们放在一个数组中作为一个

参数传入。这个工厂方法对于创建一个匹配一组单词中任意一个单词的模式非常适用。与new和compile方法不同，传入Regexp.union的字符串可自动转义：

```
# Match any one of five language names.
pattern = Regexp.union("Ruby", "Perl", "Python", /Java(Script)?/)
# Match empty parens, brackets, or braces. Escaping is automatic:
Regexp.union("()", "[]", "{}")   # => /\(\)|\[\]|\{\}/
```

9.2.3 正则表达式句法
Regular Expression Syntax

许多编程语言都支持正则表达式，而且使用的句法都与Perl中的句法相似。本书不打算对该句法做详尽讨论，只是用一些例子来快速演示一下这些句法的元素，在这些例子后面的表9-2对这些句法进行了总结。这个教程主要涉及在Ruby 1.8中使用的句法，但也演示了一些仅在Ruby 1.9中可用的特性。如果想找一本专门介绍正则表达式的书，你可以参见Jeffrey E. F. Friedl的《精通正则表达式》(*Mastering Regular Expressions*，O'Reilly)。

```
# Literal characters
/ruby/            # Match "ruby". Most characters simply match themselves.
/¥/               # Matches Yen sign. Multibyte characters are suported
                  # in Ruby 1.9 and Ruby 1.8.

# Character classes
/[Rr]uby/         # Match "Ruby" or "ruby"
/rub[ye]/         # Match "ruby" or "rube"
/[aeiou]/         # Match any one lowercase vowel
/[0-9]/           # Match any digit; same as /[0123456789]/
/[a-z]/           # Match any lowercase ASCII letter
/[A-Z]/           # Match any uppercase ASCII letter
/[a-zA-Z0-9]/     # Match any of the above
/[^aeiou]/        # Match anything other than a lowercase vowel
/[^0-9]           # Match anything other than a digit

# Special character classes
/./               # Match any character except newline
/./m              # In multiline mode . matches newline, too
/\d/              # Match a digit /[0-9]/
/\D/              # Match a nondigit: /[^0-9]/
/\s/              # Match a whitespace character: /[ \t\r\n\f]/
/\S/              # Match nonwhitespace: /[^ \t\r\n\f]/
/\w/              # Match a single word character: /[A-Za-z0-9_]/
/\W/              # Match a nonword character: /[^A-Za-z0-9_]/

# Repetition
/ruby?/           # Match "rub" or "ruby": the y is optional
/ruby*/           # Match "rub" plus 0 or more ys
/ruby+/           # Match "rub" plus 1 or more ys
/\d{3}/           # Match exactly 3 digits
/\d{3,}/          # Match 3 or more digits
/\d{3,5}/         # Match 3, 4, or 5 digits

# Nongreedy repetition: match the smallest number of repetitions
```

```
/<.*>/            # Greedy repetition: matches "<ruby>perl>"
/<.*?>/           # Nongreedy: matches "<ruby>" in "<ruby>perl>"
                  # Also nongreedy: ??, +?, and {n,m}?

# Grouping with parentheses
/\D\d+/           # No group: + repeats \d
/(\D\d)+/         # Grouped: + repeats \D\d pair
/([Rr]uby(, )?)+/ # Match "Ruby", "Ruby, ruby, ruby", etc.

# Backreferences: matching a previously matched group again
/([Rr])uby&\1ails/ # Match ruby&rails or Ruby&Rails
/(['"])[^\1]*\1/ # Single or double-quoted string
                 #   \1 matches whatever the 1st group matched
                 #   \2 matches whatever the 2nd group matched, etc.

# Named groups and backreferences in Ruby 1.9: match a 4-letter palindrome
/(?<first>\w)(?<second>\w)\k<second>\k<first>/
/(?'first'\w)(?'second'\w)\k'second'\k'first'/ # Alternate syntax

# Alternatives
/ruby|rube/       # Match "ruby" or "rube"
/rub(y|le))/      # Match "ruby" or "ruble"
/ruby(!+|\?)/     # "ruby" followed by one or more ! or one ?

# Anchors: specifying match position
/^Ruby/           # Match "Ruby" at the start of a string or internal line
/Ruby$/           # Match "Ruby" at the end of a string or line
/\ARuby/          # Match "Ruby" at the start of a string
/Ruby\Z/          # Match "Ruby" at the end of a string
/\bRuby\b/        # Match "Ruby" at a word boundary
/\brub\B/         # \B is nonword boundary:
                  #   match "rub" in "rube" and "ruby" but not alone
/Ruby(?=!)/       # Match "Ruby", if followed by an exclamation point
/Ruby(?!!)/       # Match "Ruby", if not followed by an exclamation point

# Special syntax with parentheses
/R(?#comment)/    # Matches "R". All the rest is a comment
/R(?i)uby/        # Case-insensitive while matching "uby"
/R(?i:uby)/       # Same thing
/rub(?:y|le))/    # Group only without creating \1 backreference

# The x option allows comments and ignores whitespace
/  # This is not a Ruby comment. It is a literal part
   # of the regular expression, but is ignored.
   R      # Match a single letter R
   (uby)+# Followed by one or more "uby"s
   \      # Use backslash for a nonignored space
/x            # Closing delimiter. Don't forget the x option!
```

表 9-2 总结了上面代码中演示的句法规则。

表 9-2：正则表达式句法

句法	匹配
字符类	
.	匹配除换行符外所有单个字符。如果使用 m 选项，也将匹配换行符
[...]	匹配方括号中的所有单个字符
[^...]	匹配所有不在方括号中的单个字符
\w	匹配可用作单词的字符
\W	匹配不可用作单词的字符
\s	匹配空白符，等价于[\t\n\r\f]
\S	匹配非空白符
\d	匹配数字字符，等价于[0-9]
\D	匹配非数字字符
序列、替代、分组和引用	
ab	匹配紧跟表达式 *b* 的表达式 *a*
a | *b*	匹配表达式 *a* 或 *b*
(*re*)	分组，使 re 成为一个句法单元，可以被*、+、?、|等所使用，同时“捕获”匹配的文本供以后使用
(?: *re*)	用()进行分组，但是不捕获匹配文本
(?< *name* > *re*)	像()一样对子表达式进行分组并捕获文本，并把子表达式标记为 *name* (Ruby 1.9)
(?' *name* ' *re*)	与上面一样，进行有名捕获，可以用单引号代替 *name* 两端的尖括号 (Ruby 1.9)
\1...\9	匹配与第 *n* 组子表达式所匹配文本相同的文本
\10...	如果先前有足够多的子表达式，匹配与第 *n* 组子表达式所匹配文本相同的文本；否则，用给定的八进制编码匹配字符
\k< *name* >	匹配与给定名称分组所匹配文本相同的文本
\g< *n* >	再次匹配分组 *n*，*n* 既可以是分组名也可以是分组号，而\g 只是重新匹配或冲洗执行给定的分组，持有一个普通的后向引用，用来匹配第一次匹配文本的相同文本
重复	默认情况下，重复是“贪婪的 (greedy)”——有多少匹配的文本就匹配多少次。对于后面跟着* 、+ 、?或使用带? 修饰的{}量词，这类“勉强 (reluctant)”匹配方式，尽管还是会匹配剩余的表达式部分，但是会尽量少的进行匹配。在 Ruby 1.9 中，在量词后加上一个“+”号用来表示一种“自主的 (possessive)”（不反向匹配）匹配方式
re *	匹配 *re* 零次或多次
re +	匹配 *re* 至少一次
re ?	可选，匹配 *re* 零次或一次
re { *n* }	精确匹配 *re* *n* 次
re { *n* ,}	匹配 *re* *n* 次或更多次
re { *n* , *m* }	匹配 *re* 至少 *n* 次，最多 *m* 次

续表

句法	匹配
锚	锚并不匹配任何字符，而是匹配字符间的位置关系，在某个特定位置“锚定（anchoring）”匹配
^	匹配行首
$	匹配行尾
\A	匹配字符串首部
\Z	匹配字符串尾部，如果字符串以换行符结尾，匹配前一个字符
\z	匹配字符串尾部
\G	匹配上次匹配完成时的点
\b	在括号外时，匹配单词边界；在括号内匹配回格符（backspace、0x08）
\B	匹配非单词边界
(?= *re*)	正面前向查找断言，确保后面的字符匹配 *re*，但是不包括匹配文本中的字符
(?! *re*)	负面前向查找断言，确保后面的字符不匹配 *re*
(?<= *re*)	正面反向查找断言，确保前面的字符匹配 *re*，但是不包括匹配文本中的字符（Ruby 1.9）
(?<! *re*)	负面反向查找断言，确保后面的字符不匹配 *re*（Ruby 1.9）
杂项	
(? *onflags* - *offflags*)	什么也不匹配，不过会打开 *onflags* 指定的标志，同时关闭 *offflags* 指定的标志。这两个字符串可以是 i、m 和 x 的任意组合。这种方式设置的标志在指定位置处立刻生效，在所在表达式结束时失效（或者是所在括号分组结束，或者被其他标志设置的表达式所覆盖）
(? *onflags* - *offflags* : *x*)	用给定的标志对当前子表达式匹配 *x*。这是一个非捕获性的分组
(?#...)	注释，所有括号中的文本被忽略
(?> *re*)	独立于表达式中的其余部分对 *re* 进行匹配，而不管这个匹配是否会导致其余部分匹配失败。这通常用于优化特定的复杂表达式。括号不用于捕获匹配文本

9.2.4 用正则表达式进行模式匹配
Pattern Matching with Regular Expressions

=~是 Ruby 基本的模式匹配操作符。一个操作数必须是一个正则表达式，而另一个则必须是一个字符串。（这个操作符在 Regexp 和 String 中用同样的方式实现，因此正则表达式在左边或右边均可。）=~操作符会检查字符串操作数，看它或它的子字符串是否与指定的正则表达式匹配。如果发现了一个匹配，=~操作符将返回第一个匹配在字符串中的位置。否则，返回 nil：

```
pattern = /Ruby?/i        # Match "Rub" or "Ruby", case-insensitive
pattern =~ "backrub"      # Returns 4.
```

```
"rub ruby" =~ pattern     # 0
pattern =~ "r"            # nil
```

在使用了=~操作符后，你也许会对匹配文本位置之外的一些信息感兴趣。在进行完一个成功匹配后（返回值非nil），全局变量$~指向一个MatchData对象，其中包含了所有关于该匹配的信息：

```
"hello" =~ /e\w{2}/       # 1: Match an e followed by 2 word characters
$~.string                 # "hello": the complete string
$~.to_s                   # "ell": the portion that matched
$~.pre_match              # "h": the portion before the match
$~.post_match             # "o": the portion after the match
```

$~是一个特殊的线程局部和方法局部变量，它在两个并发运行的线程中的值是不同的；并且，使用了=~操作符的方法不会修改调用者方法中的$~变量值。后面我们将深入讲解$~和相关的全局变量。这个有点神秘的变量也可以用更面向对象方式的名字Regexp.last_match来引用，用不带参数的方式调用这个方法将返回与$~相同的值。

当正则表达式中包含用圆括号括住的子表达式时，MatchData对象显得更加强大，这时，MatchData对象告诉我们匹配每个子表达式的文本（及它们的起止位置）：

```
# This is a pattern with three subpatterns
pattern = /(Ruby|Perl)(\s+)(rocks|sucks)!/
text = "Ruby\trocks!"     # Text that matches the pattern
pattern =~ text           # => 0: pattern matches at the first character
data = Regexp.last_match  # => Get match details
data.size                 # => 4: MatchData objects behave like arrays
data[0]                   # => "Ruby\trocks!": the complete matched text
data[1]                   # => "Ruby": text matching first subpattern
data[2]                   # => "\t": text matching second subpattern
data[3]                   # => "rocks": text matching third subpattern
data[1,2]                 # => ["Ruby", "\t"]
data[1..3]                # => ["Ruby", "\t", "rocks"]
data.values_at(1,3)       # => ["Ruby", "rocks"]: only selected indexes
data.captures             # => ["Ruby", "\t", "rocks"]: only subpatterns
Regexp.last_match(3)      # => "rocks": same as Regexp.last_match[3]

# Start and end positions of matches
data.begin(0)             # => 0: start index of entire match
data.begin(2)             # => 4: start index of second subpattern
data.end(2)               # => 5: end index of second subpattern
data.offset(3)            # => [5,10]: start and end of third subpattern
```

在Ruby 1.9中，如果一个模式包含有名捕获，那么该模式获得的MatchData对象可以像哈希表一样被使用，用分组的名字（用字符串或符号方式）作为主键。例如：

```
# Ruby 1.9 only
pattern = /(?<lang>Ruby|Perl) (?<ver>\d(\.\d)+) (?<review>rocks|sucks)!/
if (pattern =~ "Ruby 1.9.1 rocks!")
  $~[:lang]           # => "Ruby"
```

```
  $~[:ver]                # => "1.9.1"
  $~["review"]            # => "rocks"
  $~.offset(:ver)         # => [5,10] start and end offsets of version number
end
# Names of capturing groups and a map of group names to group numbers
pattern.names             # => ["lang", "ver", "review"]
pattern.named_captures    # => {"lang"=>[1],"ver"=>[2],"review"=>[3]}
```

有名捕获和局部变量

在 Ruby 1.9 中，如果一个正则表达式字面量包含有名捕获并且出现在=~操作符的左侧，那么捕获分组名将成为局部变量，匹配的文本被放入这些变量中。如果匹配失败，这些变量将被赋值为 nil。示例如下：

```
# Ruby 1.9 only
if /(?<lang>\w+) (?<ver>\d+\.(\d+)+) (?<review>\w+)/ =~ "Ruby 1.9 rules!"
  lang      # => "Ruby"
  ver       # => "1.9"
  review    # => "rules"
end
```

这很神奇，不过它只对正则表达式字面量有效。如果一个匹配模式被存储在变量或常量中，或是方法的返回值，或出现在操作符的右侧，那么=~操作符不会进行这种操作。如果在运行 Ruby 时打开了-w 选项，在=~操作符给一个已存在变量赋值时，Ruby 将给出一个警告。

除了=~操作符，Regexp 和 String 类还定义了 match 方法。它的功能和匹配操作符类似，不过匹配操作符返回的是匹配的位置，而这个方法则返回 MatchData 对象（如果没有匹配则返回 nil）。可以像下面这样使用它：

```
if data = pattern.match(text)  # Or: data = text.match(pattern)
  handle_match(data)
end
```

在 Ruby 1.9 中，对 match 的调用可以关联一个代码块。如果没有任何匹配，将忽略这个块，match 方法返回 nil。不过，如果可以匹配，则把匹配的 MatchData 对象传给该代码块，match 方法返回代码块的返回值。因此在 Ruby 1.9 中，上面的代码可以被简写如下：

```
pattern.match(text) {|data| handle_match(data) }
```

Ruby 1.9 还对 match 方法进行了另外一项改变，它现在能接受可选的第二个整型参数，用于指定搜索的起始位置。

9.2.4.1 用于匹配数据的全局变量

Ruby 采用了 Perl 的正则表达式句法，而且也像 Perl 一样在每次匹配后设置一些特殊的全局变量。如果你是一个 Perl 程序员，可能会觉得这些特殊变量很有用。不过如果你不是 Perl

程序员，可能会觉得这些就像天书一样！表 9-3 对这些变量进行了总结，第二列是这些变量在 English 模块中的别名。

表 9-3：特殊的正则表达式变量

变量	English	等价于
$~	$LAST_MATCH_INFO	Regexp.last_match
$&	$MATCH	Regexp.last_match[0]
$`	$PREMATCH	Regexp.last_match.pre_match
$'	$POSTMATCH	Regexp.last_match.post_match
$1	none	Regexp.last_match[1]
$2, etc.	none	Regexp.last_match[2], etc.
$+	$LAST_PAREN_MATCH	Regexp.last_match[-1]

在表 9-3 中，最重要的变量是$~，其余变量都是从它衍生而来的。如果把某个 MatchData 对象赋值给$~变量，其他的变量值也会改变。除了这个变量，其他的变量都是只读的，不能直接进行赋值。最后要强调的是，$~和从它继承而来的这些变量都是线程局部且方法局部的，这意味着两个 Ruby 线程可以同时进行模式匹配，而无须担心相互干扰；而且如果调用的方法执行模式匹配，它不会修改调用者所能看到的这些变量值。

9.2.4.2 用字符串进行模式匹配

String 类中很多方法接受 Regexp 参数。如果用一个正则表达式进行字符串索引，它将返回所匹配的那部分文本；如果正则表达式后面还有一个整数参数，它会返回 MatchData 对象所对应的元素：

```
"ruby123"[/\d+/]                    # "123"
"ruby123"[/([a-z]+)(\d+)/,1]        # "ruby"
"ruby123"[/([a-z]+)(\d+)/,2]        # "123"
```

slice 方法是字符串索引操作符[]的同义词方法。它的 slice!变体返回值与它相同，并且将从字符串中删除返回的匹配子字符串：

```
r = "ruby123"
r.slice!(/\d+/)  # Returns "123", changes r to "ruby"
```

split 方法把字符串分割成一个字符串数组，你可以把字符串或正则表达式作为分隔符：

```
s = "one, two, three"
s.split              # ["one,","two,","three"]: whitespace delimiter by default
```

```
s.split(", ")          # ["one","two","three"]: hardcoded delimiter
s.split(/\s*,\s*/)     # ["one","two","three"]: space is optional around comma
```

index方法可以搜索一个字符、子字符串或模式，并返回起始位置。如果给定一个Regexp对象作参数，则该方法的行为与=~操作符相似，不过它还可以用第二个参数指定搜索的起始位置，这样就可以在搜索完第一个之后，继续搜索下一个匹配值：

```
text = "hello world"
pattern = /l/
first = text.index(pattern)       # 2: first match starts at char 2
n = Regexp.last_match.end(0)      # 3: end position of first match
second = text.index(pattern, n)   # 3: search again from there
last = text.rindex(pattern)       # 9: rindex searches backward from end
```

9.2.4.3 搜索并替换

最重要的使用正则表达式的String方法是sub（用于替换）和gsub（用于全局替换），以及它们的变体sub!和gsub!方法。这些方法都使用正则表达式进行搜索和替换操作。sub和sub!只对第一次匹配进行替换，而gsub和gsub!会替换所有的匹配。sub和gsub不修改原有字符串，返回一个新的字符串，而sub!和gsub!则会修改调用者字符串，如果确实发生了替换，它们将返回修改的字符串，否则返回nil（这使得它们可以方便地被用于if和while语句中）：

```
phone = gets                      # Read a phone number
phone.sub!(/#.*$/, "")            # Delete Ruby-style comments
phone.gsub!(/\D/, "")             # Remove anything other than digits
```

这些查找并替换的方法并非必须跟正则表达式配合，也可以用于普通的文本替换：

```
text.gsub!("rails", "Rails")      # Change "rails" to "Rails" throughout
```

不过，使用正则表达式会非常灵活。例如，如果想把“rails”的首字母改为大写而不影响“grails”，可以：

```
text.gsub!(/\brails\b/, "Rails")  # Capitalize the word "Rails" throughout
```

在这里对搜索并替换方法进行介绍，是因为它们并非只能使用普通文本进行替换。设想你的替换依赖于匹配的细节，查找并替换方法会在真正执行替换之前处理字符串。如果字符串包含反斜杠并且后面是单个数字，这个数字将作为$~对象的索引，从MatchData对象中抽取的文本会取代反斜杠和数字所在位置的文本。例如，如果字符串包含转义序列\0，那么使用所有的匹配文本；如果替换文本包含\1，那么匹配的第一个子串用于文本替换。下

面的代码会对“ruby”进行大小写敏感的查找，把它用 HTML 的粗体标志括起来，并保留它的大小写：

```
text.gsub(/\bruby\b/i, '<b>\0</b>')
```

注意，如果用双引号括起替换字符串，须要使用两个反斜杠字符。

你可能也想用双引号插入替换方式完成同样的工作：

```
text.gsub(/\bruby\b/i, "<b>#{$&}</b>")
```

不过，这不能如你所愿，因为插入将在字符串字面量中被执行，而这时它还没有被传递给 gsub 方法。因为这个动作发生在模式匹配发生之前，所以像$&这样的变量要么还没有被定义，要么是前次匹配所遗留的值。

在 Ruby 1.9 中，可以用\k 这样的有名反向引用句法来引用有名捕获分组：

```
# Strip pairs of quotes from a string
re = /(?<quote>['"])(?<body>[^'"]*)\k<quote>/
puts "These are 'quotes'".gsub(re, '\k<body>')
```

替换字符串也可以引用不在捕获分组中的文本，可以用\&、\`、\'和\+来替换值为$&、$`、$'和$+的文本。

除了使用静态替换字符串，搜索并替换方法也可以通过代码块来调用，代码块可以被用来动态计算替换字符串。代码块的参数是要匹配的文本：

```
# Use consistent capitalization for the names of programming languages
text = "RUBY Java perl PyThOn"            # Text to modify
lang = /ruby|java|perl|python/i           # Pattern to match
text.gsub!(lang) {|l| l.capitalize }      # Fix capitalization
```

在代码块中，你可以使用在表 9-3 中列出的那些全局变量：

```
pattern = /(['"])([^\1]*)\1/   # Single- or double-quoted string
text.gsub!(pattern) do
  if ($1 == '"')       # If it was a double-quoted string
    "'#$2'"            # replace with single-quoted
  else                 # Otherwise, if single-quoted
    "\"#$2\""          # replace with double-quoted
  end
end
```

9.2.4.4 正则表达式编码

在 Ruby 1.9 中，Regexp 对象有一个与字符串类似的 encoding 方法。你可以用如下修饰词来显式指定编码：u 表示 UTF-8 编码；s 表示 SJIS 编码；e 表示 EUC-JP 编码；而 n 表示无编码。你还可以在正则表达式中用\u 转义字符显式指定 UTF-8 编码。如果不显式指定编码，

则使用源程序编码；不过如果正则表达式中所有字符都是 ASCII 编码，那么即使源程序编码是 ASCII 编码的超集，也使用 ASCII 编码。

如果试图匹配的模式与文本编码不兼容，在 Ruby 1.9 中会抛出一个异常。如果 Regexp 对象的编码不是 ASCII 编码，fixed_encoding?方法会返回 true。如果 fixed_encoding?返回 false，那么可以安全地使用这个模式来匹配编码为 ASCII 或 ASCII 超集的文本。

9.3 数字和数学运算 Numbers and Math

在第 3 章中，介绍了 Ruby 的 Numeric 类的各种子类，解释了如何写数字字面量，以及 Ruby 的整数和浮点数算术的相关知识。在这里，我们将扩展那一章的内容，讲述与数字相关的 API，以及其他和数学相关的类。

9.3.1 数字相关的方法 Numeric Methods

Numeric 和其子类定义了很多方法用于判定一个类或数值是否是一个数字。这些方法有些只能用于浮点数，有些只能用于整数：

```
# General Predicates
0.zero?           # => true (is this number zero?)
1.0.zero?         # => false
0.0.nonzero?      # => nil (works like false)
1.nonzero?        # => 1 (works like true)
1.integer?        # => true
1.0.integer?      # => false
1.scalar?         # => false: not a complex number. Ruby 1.9.
1.0.scalar?       # => false: not a complex number. Ruby 1.9.
Complex(1,2).scalar? # => true: a complex number. requires 'complex'.

# Integer predicates
0.even?           # => true (Ruby 1.9)
0.odd?            # => false

# Float predicates
ZERO, INF, NAN = 0.0, 1.0/0.0, 0.0/0.0  # Constants for testing

ZERO.finite?      # => true: is this number finite?
INF.finite?       # => false
NAN.finite?       # => false

ZERO.infinite?    # => nil: is this number infinite? Positive or negative?
INF.infinite?     # => 1
-INF.infinite?    # => -1
NAN.infinite?     # => nil

ZERO.nan?         # => false: is this number not-a-number?
INF.nan?          # => false
NAN.nan?          # => true
```

Float 类定义了一些方法用于取整。绝大多数方法在 Numeric 类中被定义，因此它们也可以被用于任意类型的数字：

```
# Rounding methods
1.1.ceil     # =>   2:  ceiling: smallest integer >= its argument
-1.1.ceil    # =>  -1:  ceiling: smallest integer >= its argument
1.9.floor    # =>   1:  floor: largest integer <= its argument
-1.9.floor   # =>  -2:  floor: largest integer <= its argument
1.1.round    # =>   1:  round to nearest integer
0.5.round    # =>   1:  round toward infinity when halfway between integers
-0.5.round   # =>  -1:  or round toward negative infinity
1.1.truncate# =>   1:  chop off fractional part: round toward zero
-1.1.to_i    # =>  -1:  synonym for truncate
```

Float 还定义了一些有用的方法和常量：

```
# Absolute value and sign
-2.0.abs      # => 2.0: absolute value
-2.0<=>0.0    # => -1: use <=> operator to compute sign of a number

# Constants
Float::MAX        # => 1.79769313486232e+308: may be platform dependent
Float::MIN        # => 2.2250738585072e-308
Float::EPSILON    # => 2.22044604925031e-16: difference between adjacent floats
```

9.3.2 Math 模块
The Math Module

Math 模块定义了常量 PI 和 E、一些三角函数和对数函数的方法及一些杂类方法。Math 的方法都是“模块方法”（参见第 7.5.3 节），这意味着它们可以通过 Math 这个命名空间来调用，或者被其他类包含后像全局函数一样被使用。下面是一些例子：

```
# Constants
Math::PI              # => 3.14159265358979
Math::E               # => 2.71828182845905

# Roots
Math.sqrt(25.0)       # => 5.0: square root
27.0**(1.0/3.0)       # => 3.0: cube root computed with ** operator

# Logarithms
Math.log10(100.0)     # => 2.0: base-10 logarithm
Math.log(Math::E**3)  # => 3.0: natural (base-e) logarithm
Math.log2(8)          # => 3.0: base-2 logarithm. Ruby 1.9 and later.
Math.log(16, 4)       # => 2.0: 2nd arg to log() is the base. Ruby 1.9.
Math.exp(2)           # => 7.38905609893065": same as Math::E**2

# Trigonometry
include Math          # Save typing: we can now omit Math prefix.
sin(PI/2)             # => 1.0: sine. Argument is in radians, not degrees.
cos(0)                # => 1.0: cosine.
tan(PI/4)             # => 1.0: tangent.
asin(1.0)/PI          # => 0.5: arcsine. See also acos and atan.
sinh(0)               # => 0.0: hyperbolic sine. Also cosh, tanh.
```

```
asinh(1.0)            # => 0.0: inverse sinh. Also acosh, atanh.

# Convert cartesian point (x,y) to polar coordinates (theta, r)
theta = atan2(y,x)    # Angle between X axis and line (0,0)-(x,y)
r = hypot(x,y)        # Hypotenuse: sqrt(x**2 + y**2)

# Decompose float x into fraction f and exponent e, such that x = f*2**e
f,e = frexp(1024.0)   # => [0.5, 11]
x = ldexp(f, e)       # => 1024: compute x = f*2**e

# Error function
erf(0.0)              # => 0.0: error function
erfc(0.0)             # => 1.0: 1-erf(x): complementary error function
```

9.3.3 数字运算
Decimal Arithmetic

标准库中的 BigDecimal 类可以用于替代 Float，当进行金融运算时，如果想避免二进制浮点运算所带来的舍入误差（参见第 3.1.4 节），它尤其有用。BigDecimal 对象的有效数字可以是任意长的，并且大小（在实践上）不受限制（可以支持超过十亿次幂的数值）。最重要的是，在进行数字运算时，它们可以提供比舍入方式更好的精度控制。示例如下：

```
require "bigdecimal"      # Load standard library
dime = BigDecimal("0.1") # Pass a string to constructor, not a Float
4*dime-3*dime == dime     # true with BigDecimal, but false if we use Float

# Compute monthly interest payments on a mortgage with BigDecimal.
# Use "Banker's Rounding" mode, and limit computations to 20 digits
BigDecimal.mode(BigDecimal::ROUND_MODE, BigDecimal::ROUND_HALF_EVEN)
BigDecimal.limit(20)
principal = BigDecimal("200000")  # Always pass strings to constructor
apr = BigDecimal("6.5")           # Annual percentage rate interest
years = 30                        # Term of mortgage in years
payments = years*12               # 12 monthly payments in a year
interest = apr/100/12             # Convert APR to monthly fraction
x = (interest+1)**payments        # Note exponentiation with BigDecimal
monthly = (principal * interest * x)/(x-1)  # Compute monthly payment
monthly = monthly.round(2)        # Round to two decimal places
monthly = monthly.to_s("f")       # Convert to human-readable string
```

可以使用 *ri* 来查看 BigDecimal 的 API，也可以在 Ruby 源代码分发包的 *ext/bigdecimal/bigdecimal_en.html* 中获得它的完整文档。

9.3.4 复数
Complex Numbers

用标准库中的 Complex 类，可以表示和操作一个复数（由一个实数和虚数构成）。像 BigDecimal 类一样，Complex 类定义了所有的普通算术操作符，还重定义了一些 Math 模块中的方法，让它们可以在复数下工作。示例如下：

```
require "complex"              # Complex is part of the standard library
c = Complex(0.5,-0.2)          # .5-.2i.
z = Complex.new(0.0, 0.0)      # Complex.new also works, but is not required
10.times { z = z*z + c }       # Iteration for computing Julia set fractals
magnitude = z.abs              # Magnitude of a complex number
x = Math.sin(z)                # Trig functions work with Complex numbers
Math.sqrt(-1.0).to_s           # => "1.0i": square root of -1
Math.sqrt(-1.0)==Complex::I    # => true
```

9.3.5 实数
Rational Numbers

标准库中的 Rational 类表示一个实数（两个整数之商），它定义了实数的算术运算操作符。最好让它与 mathn 库一起工作，这个库重定义了整数除法，用于创建实数。这个库还做了很多工作用于统一 Ruby 的算术运算，并使 Integer、Rational 和 Complex 类协同工作。

```
require "rational"             # Load the library
penny = Rational(1, 100)       # A penny is 1/100th
require "mathn"                # Makes integer division produce Rational values
nickel = 5/100
dime = 10/100
quarter = 1/4
change = 2*quarter + 3*penny # Rational result: 53/100
(1/2 * 1/3).to_s               # "1/6": mathn prints Rationals as fractions
```

9.3.6 向量和矩阵
Vectors and Matrices

matrix 库定义了 Matrix 和 Vector 类，分别代表数字矩阵和向量，以及对它们进行算术运算的操作符。对线性代数的讨论大大超出了本书的范围。下面的例子用 Vector 类代表一个二维点，并使用一个 2*2 的 Matrix 对象表示对这个点的缩放和旋转变化：

```
require "matrix"

# Represent the point (1,1) as the vector [1,1]
unit = Vector[1,1]

# The identity transformation matrix
identity = Matrix.identity(2)     # 2x2 matrix
identity*unit == unit             # true: no transformation

# This matrix scales a point by sx,sy
sx,sy = 2.0, 3.0;
scale = Matrix[[sx,0], [0, sy]]
scale*unit           # => [2.0, 3.0]: scaled point

# This matrix rotates counterclockwise around the origin
theta = Math::PI/2   # 90 degrees
rotate = Matrix[[Math.cos(theta), -Math.sin(theta)],
                [Math.sin(theta),  Math.cos(theta)]]
```

```
rotate*unit                # [-1.0, 1.0]: 90 degree rotation

# Two transformations in one
scale * (rotate*unit)      # [-2.0, 3.0]
```

9.3.7 随机数

Random Numbers

在 Ruby 中，用全局函数 Kernel.rand 产生随机数。如果不带参数，它返回一个大于等于 0.0 而小于 1.0 的伪随机浮点数。如果给定一个整型参数 max，它返回一个大于等于 0 而小于 max 的伪随机整数。示例如下：

```
rand          # => 0.964395196505186
rand          # => 0.390523655919935
rand(100)     # => 81
rand(100)     # => 32
```

如果需要一个可重复的伪随机数序列（也许是为了测试），可以用一个定值作为随机数发生器的种子：

```
srand(0)                   # Known seed
[rand(100),rand(100)]      # => [44,47]: pseudorandom sequence
srand(0)                   # Reset the seed to repeat the sequence
[rand(100),rand(100)]      # => [44,47]
```

9.4 日期和时间

Dates and Times

Time 类用于表示日期和时间。它是操作系统提供的日期和时间方法的一个简单包装，因此，在某些系统上，这个类不能用于表示 1970 年之前或 2038 年之后的日期。在 date 库中的 Date 和 DateTime 类则没有这种限制，不过下面的代码没有演示它们的用法：

```
# Creating Time objects
Time.now          # Returns a time object that represents the current time
Time.new          # A synonym for Time.now

Time.local(2007, 7, 8)             # July 8, 2007
Time.local(2007, 7, 8, 9, 10)      # July 8, 2007, 09:10am, local time
Time.utc(2007, 7, 8, 9, 10)        # July 8, 2007, 09:10 UTC
Time.gm(2007, 7, 8, 9, 10, 11)     # July 8, 2007, 09:10:11 GMT (same as UTC)

# One microsecond before the new millennium began in London
# We'll use this Time object in many examples below.
t = Time.utc(2000, 12, 31, 23, 59, 59, 999999)

# Components of a Time
t.year   # => 2000
t.month  # => 12: December
t.day    # => 31
t.wday   # => 0: day of week: 0 is Sunday
t.yday   # => 366: day of year: 2000 was a leap year
```

```
t.hour   # => 23: 24-hour clock
t.min    # => 59
t.sec    # => 59
t.usec   # => 999999: microseconds, not milliseconds
t.zone   # => "UTC": timezone name

# Get all components in an array that holds
# [sec,min,hour,day,month,year,wday,yday,isdst,zone]
# Note that we lose microseconds
values = t.to_a       # => [59, 59, 23, 31, 12, 2000, 0, 366, false, "UTC"]

# Arrays of this form can be passed to Time.local and Time.utc
values[5] += 1        # Increment the year
Time.utc(*values)     # => Mon Dec 31 23:59:59 UTC 2001

# Timezones and daylight savings time
t.zone       # => "UTC": return the timezone
t.utc?       # => true: t is in UTC time zone
t.utc_offset # => 0: UTC is 0 seconds offset from UTC
t.localtime  # Convert to local timezone. Mutates the Time object!
t.zone       # => "PST" (or whatever your timezone is)
t.utc?       # => false
t.utc_offset # => -28800: 8 hours before UTC
t.gmtime     # Convert back to UTC. Another mutator.
t.getlocal   # Return a new Time object in local zone
t.getutc     # Return a new Time object in UTC
t.isdst      # => false: UTC does not have DST. Note no ?.
t.getlocal.isdst # => false: no daylight savings time in winter.

# Weekday predicates: Ruby 1.9
t.sunday?    # => true
t.monday?    # => false
t.tuesday?   # etc.

# Formatting Times and Dates
t.to_s       # => "Sun Dec 31 23:59:59 UTC 2000": Ruby 1.8
t.to_s       # => "2000-12-31 23:59:59 UTC": Ruby 1.9 uses ISO-8601
t.ctime      # => "Sun Dec 31 23:59:59 2000": another basic format

# strftime interpolates date and time components into a template string
# Locale-independent formatting
t.strftime("%Y-%m-%d %H:%M:%S")   # => "2000-12-31 23:59:59": ISO-8601 format
t.strftime("%H:%M")               # => "23:59": 24-hour time
t.strftime("%I:%M %p")            # => "11:59 PM": 12-hour clock

# Locale-dependent formats
t.strftime("%A, %B %d")           # => "Sunday, December 31"
t.strftime("%a, %b %d %y")        # => "Sun, Dec 31 00": 2-digit year
t.strftime("%x")                  # => "12/31/00": locale-dependent format
t.strftime("%X")                  # => "23:59:59"
t.strftime("%c")                  # same as ctime

# Parsing Times and Dates
require 'parsedate'       # A versatile date/time parsing library
include ParseDate         # Include parsedate() as a global function
```

```
datestring = "2001-01-01"
values = parsedate(datestring)    # [2001, 1, 1, nil, nil, nil, nil, nil]
t = Time.local(*values)           # => Mon Jan 01 00:00:00 -0800 2001
s = t.ctime                       # => "Mon Jan  1 00:00:00 2001"
Time.local(*parsedate(s))==t      # => true

s = "2001-01-01 00:00:00-0500"    # midnight in New York
v = parsedate(s)                  # => [2001, 1, 1, 0, 0, 0, "-0500", nil]
t = Time.local(*v)                # Loses time zone information!

# Time arithmetic
now = Time.now            # Current time
past = now - 10           # 10 seconds ago. Time - number => Time
future = now + 10         # 10 seconds from now Time + number => Time
future - now              # => 10  Time - Time => number of seconds

# Time comparisons
past <=> future           # => -1
past < future             # => true
now >= future             # => false
now == now                # => true

# Helper methods for working with time units other than seconds
class Numeric
  # Convert time intervals to seconds
  def milliseconds; self/1000.0; end
  def seconds; self; end
  def minutes; self*60; end
  def hours; self*60*60; end
  def days; self*60*60*24; end
  def weeks; self*60*60*24*7; end

  # Convert seconds to other intervals
  def to_milliseconds; self*1000; end
  def to_seconds; self; end
  def to_minutes; self/60.0; end
  def to_hours; self/(60*60.0); end
  def to_days; self/(60*60*24.0); end
  def to_weeks; self/(60*60*24*7.0); end
end

expires = now + 10.days       # 10 days from now
expires - now                 # => 864000.0 seconds
(expires - now).to_hours      # => 240.0 hours

# Time represented internally as seconds since the (platform-dependent) epoch
t = Time.now.to_i     # => 1184036194 seconds since epoch
Time.at(t)            # => seconds since epoch to Time object
t = Time.now.to_f     # => 1184036322.90872: includes 908720 microseconds
Time.at(0)            # => Wed Dec 31 16:00:00 -0800 1969: epoch in local time
```

9.5 集合

本节讲述 Ruby 的集合类。集合是指任何可以包含一组值的类。Array 和 Hash 是 Ruby 中最重要的集合类，另外，在标准库中还引入了 Set 类。这些集合类都混入了 Enumerable 模块，这意味着 Emumerable 的方法在所有集合中都可用。

9.5.1 Enumerable 对象

Enumerable 模块是一种混入模块，它在 each 迭代器的基础上实现了一组有用的方法。下面要描述的 Array、Hash 和 Set 类都包含了 Enumerable 模块，因此也都具有这里所描述的方法。Range 和 IO 是另外两个值得注意的可枚举的类。在第 5.3.2 节中曾经对 Enumerable 做了简要介绍，本节将更详细地介绍这个模块。

注意，一些可枚举的类有自然的枚举顺序，它们的 each 方法遵循这个顺序。比如，数组按照索引的升序枚举各个元素；Range 对象按照升序枚举元素；IO 对象按照对应的文件和 socket 读入的文本顺序枚举每一行；在 Ruby 1.9 中，Hash 和 Set（它基于 Hash 实现）按照元素插入的顺序进行枚举；而在 Ruby 1.9 前，这些类是以随意的顺序进行枚举的。

许多 Enumerable 的方法返回一个处理后的可枚举集合或一个子集。通常情况下，如果一个 Enumerable 方法返回一个集合（而非集合中的一个值），这个集合是一个 Array 对象。不过并非始终如此，例如，Hash 类覆盖了 reject 方法，返回一个 Hash 对象而非一个数组。不管返回值究竟是什么，用 Enumerable 的方法返回的集合一定是可枚举的。

9.5.1.1 对集合进行迭代和转换

根据定义，任何 Enumerable 对象必须有一个 each 迭代器。Enumerable 还提供了一个简单的变体迭代器 each_with_index，它接受一个集合元素和一个整数。对于数组，这个整数是数组的索引；对于 IO 对象，它是行号（从 0 开始计数）；对于其他对象，它是该集合转换为数组后的数组索引值：

```
(5..7).each {|x| print x }                    # Prints "567"
(5..7).each_with_index {|x,i| print x,i }     # Prints "506172"
```

在 Ruby 1.9 中，Enumerable 定义了 cycle 迭代器，它重复迭代集合中的每个元素，无限循环一直到给定的代码块用 break、return，或者抛出异常来明确中止这个迭代。在对 Enumerable 对象进行第一次迭代时，cycle 把所有元素存储在一个数组中，以后的迭代将

在数组中进行，这意味着即使在第一遍迭代后就改变了集合，也不会影响 cycle 的行为。

each_slice 和 each_cons 这两个迭代器会迭代集合的子数组。在 Ruby 1.8 中，须要加入 require 'enumerator'语句才可以使用；而在 Ruby 1.9 中，它们已经成为内核库的一部分。each_slice(n)在 n 个元素的“切片（slice）”上对集合进行迭代：

```
(1..10).each_slice(4) {|x| print x } # Prints "[1,2,3,4][5,6,7,8][9,10]"
```

each_cons 跟 each_slice 类似，不过它以在集合上使用“滑动窗口”的方式进行迭代：

```
(1..5).each_cons(3) {|x| print x }    # Prints "[1,2,3][2,3,4][3,4,5]"
```

collect 方法把给定代码块应用到集合的每个元素上，并把所有块代码的返回值集合到一个数组中。map 是它的同义词，它把集合中的每个元素映射到一个数组上，数组的值是对相应元素执行给定块代码的结果：

```
data = [1,2,3,4]                          # An enumerable collection
roots = data.collect {|x| Math.sqrt(x)}   # Collect roots of our data
words = %w[hello world]                   # Another collection
upper = words.map {|x| x.upcase }         # Map to uppercase
```

zip 方法对一个集合用其他零个或多个集合进行插值，并把每组元素（从各个集合取出一个元素）迭代给关联代码块。如果没有关联代码块，在 Ruby 1.8 中会返回一个数组的数组，而在 Ruby 1.9 中则返回一个枚举器对象（对这个枚举器对象调用 to_a 方法则可得到在 1.8 中返回的数组）。

```
(1..3).zip([4,5,6]) {|x| print x.inspect }    # Prints "[1,4][2,5][3,6]"
(1..3).zip([4,5,6],[7,8]) {|x| print x}       # Prints "14725836"
(1..3).zip('a'..'c') {|x,y| print x,y }       # Prints "1a2b3c"
```

Enumerable 定义了一个 to_a 方法（及同义词方法 entries），用于把一个可枚举的集合转换为一个数组。在这里介绍这个方法是因为这种转换很显然需要迭代集合。在结果数组中各个元素的顺序与 each 迭代器的结果相同：

```
(1..3).to_a           # => [1,2,3]
(1..3).entries        # => [1,2,3]
```

如果加入 require 'set'语句，所有的 Enumerable 可获得一个 to_set 的转换方法。集合将在 9.5.4 节中被详细介绍：

```
require 'set'
(1..3).to_set     # => #<Set: {1, 2, 3}>
```

9.5.1.2 枚举器和外部迭代器

在第 5.3.4 节和第 5.3.5 节中，我们对枚举器及其作为外部迭代器的使用已经进行了详细介绍，本节只对第 5 章的描述进行一个简单概括，并提供一些示例。

枚举器是用 Enumerable::Enumerator 类来实现的，它只用很少的方法就完成了一个十分强大的迭代结构。枚举器主要是 Ruby 1.9 的一个特性，不过一些枚举器方法在 Ruby 1.8 中也可以通过包含 enumerator 库来得到。你可以通过 to_enum 或它的别名方法 enum_for 来创建一个 Enumerator 对象，也可以直接用不带代码块的方式调用 iterator 方法：

```
e = [1..10].to_enum                 # Uses Range.each
e = "test".enum_for(:each_byte)     # Uses String.each_byte
e = "test".each_byte                # Uses String.each_byte
```

Enumerator 对象是带有 each 方法的 Enumerable 对象，这个 each 方法建立在其他对象的某个迭代器方法上。除了作为 Enumerable 的代理对象之外，枚举器还可以作为外部迭代器被使用。为了用外部迭代器获得一个集合的元素，你可以简单地重复调用 next 方法，直到抛出 StopIteration 异常。用 Kernel.loop 迭代器可以捕获 StopIteration 异常。在 next 方法抛出 StopIteration 异常后，一般将开始一个新的迭代（这意味着这个迭代器支持重复迭代，而读取文件的迭代器就不能这样做）。如果允许重复迭代，可以在 StopIteration 抛出前调用 rewind 方法开始一个外部迭代：

```
"Ruby".each_char.max      # => "y"; Enumerable method of Enumerator
iter = "Ruby".each_char   # Create an Enumerator
loop { print iter.next }  # Prints "Ruby"; use it as external iterator
print iter.next           # Prints "R": iterator restarts automatically
iter.rewind               # Force it to restart now
print iter.next           # Prints "R" again
```

对于任意枚举器 e，你可以通过 e.with_index 获得一个新的枚举器。正如方法名所暗示的，这个新的枚举器会迭代出一个索引及原有迭代器应迭代出的值：

```
# Print "0:R\n1:u\n2:b\n3:y\n"
"Ruby".each_char.with_index.each {|c,i| puts "#{i}:#{c} }
```

Enumerable::Enumerator 类定义了一个 to_splat 方法，这意味着可以把一个用*做前缀的枚举器扩展为单个值，以供方法调用或并行赋值语句使用。

9.5.1.3 集合排序

Enumerable 最重要的方法之一是 sort，它把可枚举的集合转换为一个数组，并在该数组上对元素进行排序。默认情况下，排序依据各元素定义的<=>方法来完成。不过，如果提供了接受一对元素作参数的代码块，这个代码块会返回-1、0 或+1，用于指示元素的相对顺序：

```
w = Set['apple','Beet','carrot']  # A set of words to sort
w.sort                            # ['Beet','apple','carrot']: alphabetical
w.sort {|a,b| b<=>a }             # ['carrot','apple','Beet']: reverse
w.sort {|a,b| a.casecmp(b) }      # ['apple','Beet','carrot']: ignore case
w.sort {|a,b| b.size<=>a.size}    # ['carrot','apple','Beet']: reverse length
```

如果关联的代码块要做大量计算来进行元素比较，那么用 sort_by 会更有效率。sort_by 关联的代码块在每个元素上只被执行一次，为每个元素产生一个数值型的“排序键”，然后集合根据这些排序键进行升序排列。这样，排序键只为每个元素计算一次，而不用在每次比较中都计算两次：

```
# Case-insensitive sort
words = ['carrot', 'Beet', 'apple']
words.sort_by {|x| x.downcase}      # => ['apple', 'Beet', 'carrot']
```

9.5.1.4 搜索集合

Enumerable 定义了好几个方法来查找集合中的某个元素。include?(及同义词方法 member?)用于查找等于（用==）给定参数的元素：

```
primes = Set[2, 3, 5, 7]
primes.include? 2        # => true
primes.member? 1         # => false
```

find 方法（及同义词方法 detect）把相关代码块依次应用到集合的每个元素中。如果代码块的返回值不是 false 或 nil，find 返回相应的元素并停止迭代；如果代码块对所有元素都返回 false 或 nil，那么 find 返回 nil：

```
# Find the first subarray that includes the number 1
data = [[1,2], [0,1], [7,8]]
data.find {|x| x.include? 1}      # => [1,2]
data.detect {|x| x.include? 3}    # => nil: no such element
```

在 Ruby 1.9 中，find_index 方法的工作方式与 find 相似，不过它返回的是匹配元素的索引而非元素本身。与 find 一样，在没有匹配元素时，它返回 nil：

```
data.find_index {|x| x.include? 1}    # => 0: the first element matches
data.find_index {|x| x.include? 3}    # => nil: no such element
```

注意，对于那些不使用数字索引的集合（比如哈希表和集合），find_index 的返回值作用不大。

Enumerable 还定义了其他一些搜索方法，用于搜索一组匹配结果。在下面的小节中我们将讨论这个主题。

9.5.1.5 选择子集合

select 方法选择并返回那些代码块代码返回非空（non-nil）和非假（non-false）的元素。find_all 是 select 的同义词方法，它与 find 方法相似，不过返回的是所有匹配元素的数组：

```
(1..8).select {|x| x%2==0}    # => [2,4,6,8]: select even elements
(1..8).find_all {|x| x%2==1} # => [1,3,5,7]: find all odd elements
```

reject 方法的作用与 select 相反，返回的数组中是那些代码块代码执行结果为 nil 或 false 的元素。

```
primes = [2,3,5,7]
primes.reject {|x| x%2==0}  # => [3,5,7]: reject the even ones
```

如果希望既获得 select 的结果，也获得 reject 的结果，你可以使用 partition 方法。它返回用两个数组做元素的数组。第一个子数组中是代码块代码执行结果为真的元素；第二个子数组是代码块代码执行结果为假的元素：

```
(1..8).partition {|x| x%2==0}  # => [[2, 4, 6, 8], [1, 3, 5, 7]]
```

Ruby 1.9 的 group_by 方法是 partition 方法的推广。在这个方法中，代码块不是用来把集合分为两组，而是把代码块执行的结果作为哈希表的主键使用。所有返回值等于主键的元素构成一个数组，作为该主键的值。示例如下：

```
# Group programming languages by their first letter
langs = %w[ java perl python ruby ]
groups = langs.group_by {|lang| lang[0] }
groups # => {"j"=>["java"], "p"=>["perl", "python"], "r"=>["ruby"]}
```

grep 返回一个所有匹配给定参数的元素的数组，决定匹配与否的方法是参数的条件相等性比较符（case equality operator，===）。如果参数使用正则表达式，这个方法的效果与 Unix 的命令行工具 *grep* 相似。如果在调用时关联一个代码块，它会像 collect 或 map 一样处理匹配的元素：

```
langs = %w[ java perl python ruby ]
langs.grep(/^p/)                        # => [perl, python]: start with 'p'
langs.grep(/^p/) {|x| x.capitalize}     # => [Perl, Python]: fix caps
data = [1, 17, 3.0, 4]
ints = data.grep(Integer)               # => [1, 17, 4]: only integers
small = ints.grep(0..9)                 # [1,4]: only in range
```

在 Ruby 1.9 中，在前面提到的选择方法中又增加了 first、take、drop、take_while 和 drop_while 方法。first 方法返回 Enumerable 对象的第一个元素，如果给定一个整型参数 n，则返回一个包含前 n 个元素的数组。take 和 drop 方法都需要一个整型参数。take 方法的功能跟 first 很相似，它返回 Enumerable 对象前面 n 个元素构成的数组。drop 方法则刚好相反，它返回除了前面 n 个元素外所有元素构成的数组：

```
p (1..5).first(2)         # => [1,2]
p (1..5).take(3)          # => [1,2,3]
p (1..5).drop(3)          # => [4,5]
```

take_while 和 drop_while 方法不需要整型参数，而需要一个代码块。take_while 把 Enumerable 对象的元素依次传给代码块，直到代码块返回 false 或 nil 为止，然后前面返回值为真的元素就形成一个数组作为方法的返回值。drop_while 也把 Enumerable 对象的元素依次传给代码块，直到代码块返回 false 或 nil 为止，然而，它返回的数组包含那个返回值为假的元素及所有后继的元素：

```
[1,2,3,nil,4].take_while {|x| x }     # => [1,2,3]: take until nil
[nil, 1, 2].drop_while {|x| !x }      # => [1,2]: drop leading nils
```


9.5.1.6 集合概要信息

集合亦可以被简化为一个值，用来描述集合的某些属性。min 和 max 就是这样两个方法，它们分别返回集合的最小和最大元素（假设这些元素支持可变比较符<=>）：

```
[10, 100, 1].min      # => 1
['a','c','b'].max     # => 'c'
[10, 'a', []].min     # => ArgumentError: elements not comparable
```

像 sort 一样，min 和 max 可以接受一个代码块来比较两个元素。在 Ruby 1.9 中，你也可以使用更简单的 min_by 和 max_by：

```
langs = %w[java perl python ruby]     # Which has the longest name?
langs.max {|a,b| a.size <=> b.size }  # => "python": block compares 2
langs.max_by {|word| word.length }    # => "python": Ruby 1.9 only
```

Ruby 1.9 还定义了 minmax 和 minmax_by 方法，可以同时计算集合的最小和最大值，并把它们用一个两元素的数组返回：

```
(1..100).minmax                    # => [1,100] min, max as numbers
(1..100).minmax_by {|n| n.to_s }   # => [1,99]  min, max as strings
```

any?和 all?这两个判定方法也可以被用于简化集合，它们对集合的元素应用一个判定代码块。对于 all?方法来说，如果对集合所有元素的判定都为真（即不是 nil 也不是 false），那么它的返回值为真。对 any?来说，只要有一个元素的判定为真，返回值就为真。Ruby 1.9 中，none?在没有任何元素被判定为真的情况下返回真。在 Ruby 1.9 中，one?在有一个且只有一个元素被判定为真的情况下返回真。如果不带代码块，这些方法会直接检测集合自身：

```
c = -2..2
c.all? {|x| x>0}      # => false: not all values are > 0
c.any? {|x| x>0}      # => true: some values are > 0
c.none? {|x| x>2}     # => true: no values are > 2
c.one? {|x| x>0}      # => false: more than one value is > 0
c.one? {|x| x>2}      # => false: no values are > 2
c.one? {|x| x==2}     # => true: one value == 2
[1, 2, 3].all?        # => true: no values are nil or false
[nil, false].any?     # => false: no true values
[].none?              # => true: no non-false, non-nil values
```

在 Ruby 1.9 中，还增加了一个 count 方法：它返回在集合中等于给定值元素的个数，或者是关联代码块返回真的元素的个数：

```
a = [1,1,2,3,5,8]
a.count(1)                    # => 2: two elements equal 1
a.count {|x| x % 2 == 1}      # => 4: four elements are odd
```

最后，inject 方法是一个被用于简化集合的通用方法。Ruby 1.9 中，reduce 被定义为 inject 的别名。inject 关联的代码块需要两个参数。第一个参数是一个累积值，第二个参数是集合中的元素。在第一次迭代中，inject 方法的参数值被传给累积值。在一次迭代中，代码块返回的值会成为下次迭代的累积值。最后一个迭代的返回值成为 inject 方法的返回值。示例如下：

```
# How many negative numbers?
(-2..10).inject(0) {|num, x| x<0 ? num+1 : num }  # => 2

# Sum of word lengths
%w[pea queue are].inject(0) {|total, word| total + word.length }  # => 11
```

如果在调用 inject 时不带参数，那么在代码块第一次被调用时，集合的头两个元素被传递给代码块。(如果集合只有一个元素，那么 inject 直接返回这个元素。) 这种形式的 inject 在一些常用操作中很有用：

```
sum = (1..5).inject {|total,x| total + x}      # => 15
prod = (1..5).inject {|total,x| total * x}     # => 120
max = [1,3,2].inject {|m,x| m>x ? m : x}       # => 3
[1].inject {|total,x| total + x}               # => 1: block never called
```

在 Ruby 1.9 中，可以用一个符号表示一个方法（或操作符）传给 inject 方法，从而无须指明一个代码块。集合中的每个元素被传给那个有名方法作为累积值，它的结果成为新的累积值。在使用这种方式时，一般使用 reduce 这个同义词方法：

```
sum = (1..5).reduce(:+)                        # => 15
prod = (1..5).reduce(:*)                       # => 120
letters = ('a'..'e').reduce("-", :concat)      # => "-abcde"
```

9.5.2 数组

Arrays

在 Ruby 编程中，数组可能是最基础、最常用的数据结构了。在第 3.3 节中，我们描述了数组字面量和索引操作符，在此基础上，本节将演示 Array 类所实现的丰富 API。

9.5.2.1 创建数组

数组可以用数组字面量创建，也可以用 Array.new 类方法创建，还可以用类操作符 Array.[] 创建。示例如下：

```
[1,2,3]          # Basic array literal
[]               # An empty array
[]               # Arrays are mutable: this empty array is different
%w[a b c]        # => ['a', 'b', 'c']: array of words
Array[1,2,3]     # => [1,2,3]: just like an array literal
```

```
# Creating arrays with the new() method
empty = Array.new             # []: returns a new empty array
nils = Array.new(3)           # [nil, nil, nil]: three nil elements
copy = Array.new(nils)        # Make a new copy of an existing array
zeros = Array.new(4, 0)       # [0, 0, 0, 0]: four 0 elements
count = Array.new(3){|i| i+1}# [1,2,3]: three elements computed by block

# Be careful with repeated objects
a=Array.new(3,'a')  # => ['a','a','a']: three references to the same string
a[0].upcase!          # Capitalize the first element of the array
a                     # => ['A','A','A']: they are all the same string!
a=Array.new(3){'b'}  # => ['b','b','b']: three distinct string objects
a[0].upcase!;         # Capitalize the first one
a                     # => ['B','b','b']: the others are still lowercase
```

除了 Array 的工厂方法外，很多类都定义了 to_a 方法，用来返回一个数组。尤其值得一提的是，所有 Enumerable 对象（比如 Range 和 Hash 对象）都可以通过 to_a 方法转换为数组。另外，数组操作符（比如+）和许多数组方法（比如 slice）会创建和返回新的数组，而不是在原有数组上进行修改。

9.5.2.2 数组大小和其中的元素

下面的代码演示了如何获得一个数组的长度，以及从数组中提取元素和子数组的多种方法：

```
# Array length
[1,2,3].length        # => 3
[].size               # => 0: synonym for length
[].empty?             # => true
[nil].empty?          # => false
[1,2,nil].nitems      # => 2: number of non-nil elements
[1,2,3].nitems {|x| x>2} # => 1: # of elts matching block (Ruby 1.9)

# Indexing single elements
a = %w[a b c d]       # => ['a', 'b', 'c', 'd']
a[0]                  # => 'a': first element
a[-1]                 # => 'd': last element
a[a.size-1]           # => 'd': last element
a[-a.size-1]          # => 'a': first element
a[5]                  # => nil: no such element
a[-5]                 # => nil: no such element
a.at(2)               # => 'c': just like [] for single integer argument
a.fetch(1)            # => 'b': also like [] and at
a.fetch(-1)           # => 'd': works with negative args
a.fetch(5)            # => IndexError!: does not allow out-of-bounds
a.fetch(-5)           # => IndexError!: does not allow out-of-bounds
a.fetch(5, 0)         # => 0: return 2nd arg when out-of-bounds
a.fetch(5){|x|x*x}    # => 25: compute value when out-of-bounds
a.first               # => 'a': the first element
a.last                # => 'd': the last element
a.choice              # Ruby 1.9: return one element at random

# Indexing subarrays
```

```
a[0,2]                # => ['a','b']: two elements, starting at 0
a[0..2]               # => ['a','b','c']: elements with index in range
a[0...2]              # => ['a','b']: three dots instead of two
a[1,1]                # => ['b']: single element, as an array
a[-2,2]               # => ['c','d']: last two elements
a[4,2]                # => []: empty array right at the end
a[5,1]                # => nil: nothing beyond that
a.slice(0..1)         # => ['a','b']: slice is synonym for []
a.first(3)            # => ['a','b','c']: first three elements
a.last(1)             # => ['d']: last element as an array

# Extracting arbitrary values
a.values_at(0,2)          # => ['a','c']
a.values_at(4, 3, 2, 1)   # => [nil, 'd','c','b']
a.values_at(0, 2..3, -1)  # => ['a','c','d','d']
a.values_at(0..2,1..3)    # => ['a','b','c','b','c','d']
```

9.5.2.3 改变数组元素

下面的代码演示了如何修改数组的单个元素、向数组中插入值、从数组中删除值，以及替换现有元素的值：

```
a = [1,2,3]           # Start with this array
# Changing the value of elements
a[0] = 0              # Alter an existing element: a is [0,2,3]
a[-1] = 4             # Alter the last element: a is [0,2,4]
a[1] = nil            # Set the 2nd element to nil: a is [0,nil,4]

# Appending to an array
a = [1,2,3]           # Start over with this array
a[3] = 4              # Add a fourth element to it: a is [1,2,3,4]
a[5] = 6              # We can skip elements: a is [1,2,3,4,nil,6]
a << 7                # => [1,2,3,4,nil,6,7]
a << 8 << 9           # => [1,2,3,4,nil,6,7,8,9] operator is chainable
a = [1,2,3]           # Start over with short array
a + a                 # => [1,2,3,1,2,3]: + concatenates into new array
a.concat([4,5])       # => [1,2,3,4,5]: alter a in place: note no !

# Inserting elements with insert
a = ['a', 'b', 'c']
a.insert(1, 1, 2)     # a now holds ['a',1,2,'b','c']. Like a[1,0] = [1,2]

# Removing (and returning) individual elements by index
a = [1,2,3,4,5,6]
a.delete_at(4)        # => 5: a is now [1,2,3,4,6]
a.delete_at(-1)       # => 6: a is now [1,2,3,4]
a.delete_at(4)        # => nil: a is unchanged

# Removing elements by value
a.delete(4)           # => 4: a is [1,2,3]
a[1] = 1              # a is now [1,1,3]
a.delete(1)           # => 1: a is now [3]: both 1s removed
a = [1,2,3]
a.delete_if {|x| x%2==1} # Remove odd values: a is now [2]
```

```
a.reject! {|x| x%2==0}    # Like delete_if: a is now []

# Removing elements and subarrays with slice!
a = [1,2,3,4,5,6,7,8]
a.slice!(0)           # => 1: remove element 0: a is [2,3,4,5,6,7,8]
a.slice!(-1,1)        # => [8]: remove subarray at end: a is [2,3,4,5,6,7]
a.slice!(2..3)        # => [4,5]: works with ranges: a is [2,3,6,7]
a.slice!(4,2)         # => []: empty array just past end: a unchanged
a.slice!(5,2)         # => nil: a now holds [2,3,6,7,nil]!

# Replacing subarrays with []=
# To delete, assign an empty array
# To insert, assign to a zero-width slice
a = ('a'..'e').to_a       # => ['a','b','c','d','e']
a[0,2] = ['A','B']        # a now holds ['A', 'B', 'c', 'd', 'e']
a[2...5]=['C','D','E']    # a now holds ['A', 'B', 'C', 'D', 'E']
a[0,0] = [1,2,3]          # Insert elements at the beginning of a
a[0..2] = []              # Delete those elements
a[-1,1] = ['Z']           # Replace last element with another
a[-1,1] = 'Z'             # For single elements, the array is optional
a[1,4] = nil              # Ruby 1.9: a now holds ['A',nil]
                          # Ruby 1.8: a now holds ['A']: nil works like []

# Other methods
a = [4,5]
a.replace([1,2,3])        # a now holds [1,2,3]: a copy of its argument
a.fill(0)                 # a now holds [0,0,0]
a.fill(nil,1,3)           # a now holds [0,nil,nil,nil]
a.fill('a',2..4)          # a now holds [0,nil,'a','a','a']
a[3].upcase!              # a now holds [0,nil,'A','A','A']
a.fill(2..4) { 'b' }      # a now holds [0,nil,'b','b','b']
a[3].upcase!              # a now holds [0,nil,'b','B','b']
a.compact                 # => [0,'b','B','b']: copy with nils removed
a.compact!                # Remove nils in place: a now holds [0,'b','B','b']
a.clear                   # a now holds []
```

9.5.2.4 对数组进行迭代、搜索和排序

Array 类混入了 Enumerable 模块，因此所有 Enumerable 的迭代器都在 Array 中可用。另外，Array 类自己也定义了一些重要的迭代器，以及一些相关的搜索和排序方法：

```
a = ['a','b','c']
a.each {| elt| print elt }        # The basic each iterator prints "abc"
a.reverse_each {|e| print e}      # Array-specific: prints "cba"
a.cycle {|e| print e }            # Ruby 1.9: prints "abcabcabc..." forever
a.each_index {|i| print i}        # Array-specific: prints "012"
a.each_with_index{|e,i| print e,i} # Enumerable: prints "a0b1c2"
a.map {|x| x.upcase}              # Enumerable: returns ['A','B','C']
a.map! {|x| x.upcase}             # Array-specific: alters a in place
a.collect! {|x| x.downcase!}      # collect! is synonym for map!

# Searching methods
a = %w[h e l l o]
a.include?('e')                   # => true
```

```
a.include?('w')                    # => false
a.index('l')                       # => 2: index of first match
a.index('L')                       # => nil: no match found
a.rindex('l')                      # => 3: search backwards
a.index {|c| c =~ /[aeiou]/}       # => 1: index of first vowel. Ruby 1.9.
a.rindex {|c| c =~ /[aeiou]/}      # => 4: index of last vowel. Ruby 1.9.

# Sorting
a.sort       # => %w[e h l l o]: copy a and sort the copy
a.sort!      # Sort in place: a now holds ['e','h','l','l','o']
a = [1,2,3,4,5]              # A new array to sort into evens and odds
a.sort! {|a,b| a%2 <=> b%2}  # Compare elements modulo 2

# Shuffling arrays: the opposite of sorting; Ruby 1.9 only
a = [1,2,3]      # Start ordered
puts a.shuffle   # Shuffle randomly. E.g.: [3,1,2]. Also shuffle!
```

9.5.2.5 数组比较

当且仅当两个数组具有同样数量的元素，且这些元素具有同样的值，并以相同的顺序出现时，我们才认为这两个数组是相等的。==方法通过所包含元素的==方法来检测相等性，而eql?则通过所含元素的eql?方法来检测相等性。在绝大多数情况下，这两个相等性测试方法返回同样的结果。

Array类没有包含Comparable模块，不过它定义了<=>操作符，用来定义数组的顺序。这个顺序可以与字符串的顺序做类比，而且字符串数组的排序方式与相应的String对象相同。数组的比较是各元素逐个进行，从索引0处开始。如果任意一对元素不相等，那么数组比较方法就返回这对元素比较的结果。如果所有元素都相等，并且两个数组长度相同，那么<=>返回0。否则，一个数组等于另外一个数组的前面部分，这种情况下，较长的数组比较短的数组大。注意，空数组[]被视作所有非空数组的前面部分，因此它总是比非空数组小。另外，如果数组的某对元素不可比较（比如，一个元素是数字而另一个是字符串），那么<=>返回nil而不是-1、0或+1：

```
[1,2] <=> [4,5]       # => -1 because 1 < 4
[1,2] <=> [0,0,0]     # => +1 because 1 > 0
[1,2] <=> [1,2,3]     # => -1 because first array is shorter
[1,2] <=> [1,2]       # => 0: they are equal
[1,2] <=> []          # => +1 [] always less than a nonempty array
```

9.5.2.6 用作栈和队列的数组

push和pop方法可以在数组的尾部增加和删除元素，这样数组就可以作为一个后进先出的栈被使用：

```
a = []
a.push(1)         # => [1]: a is now [1]
a.push(2,3)       # => [1,2,3]: a is now [1,2,3]
a.pop             # => 3: a is now [1,2]
```

```
a.pop            # => 2: a is now [1]
a.pop            # => 1: a is now []
a.pop            # => nil: a is still []
```

shift 与 pop 类似，不过它删除并返回数组的是第一个元素而非最后一个。unshift 类似于 push，不过它在数组的最前端增加元素，而不是在最后增加元素。可以用 push 和 shift 实现一个先进先出的队列：

```
a = []
a.push(1)        # => [1]: a is [1]
a.push(2)        # => [1,2]: a is [1,2]
a.shift          # => 1: a is [2]
a.push(3)        # => [2,3]: a is [2,3]
a.shift          # => 2: a is [3]
a.shift          # => 3: a is []
a.shift          # => nil: a is []
```

9.5.2.7 用作集合的数组

Array 类实现了&、|和-操作符，可以执行像集合一样的交集、并集和差集操作。另外，该类还定义了 include?方法来检测一个值是否在数组中存在（或者说是否为该数组的一个成员）。它甚至还定义了 uniq 和 uniq!方法用于除去数组中的重复元素（集合不允许存在重复元素）。Array 并非一个高效率的集合实现（标准库中的 Set 类是这样一个类），不过它可以方便地被用来表示小规模的集合：

```
[1,3,5] & [1,2,3]              # => [1,3]: set intersection
[1,1,3,5] & [1,2,3]            # => [1,3]: duplicates removed
[1,3,5] | [2,4,6]              # => [1,3,5,2,4,6]: set union
[1,3,5,5] | [2,4,6,6]          # => [1,3,5,2,4,6]: duplicates removed
[1,2,3] - [2,3]                # => [1]: set difference
[1,1,2,2,3,3] - [2, 3]         # => [1,1]: not all duplicates removed

small = 0..10.to_a             # A set of small numbers
even = 0..50.map {|x| x*2}     # A set of even numbers
smalleven = small & even       # Set intersection
smalleven.include?(8)          # => true: test for set membership

[1, 1, nil, nil].uniq          # => [1, nil]: remove dups. Also uniq!
```

注意&和-操作符不会指定返回数组的元素顺序，只有在确信数组表示的是一个无序集合时才应该使用它们。

在 Ruby 1.9 中，Array 类还定义了集合的组合方法，用来计算排列、组合和笛卡尔积：

```
a = [1,2,3]

# Iterate all possible 2-element subarrays (order matters)
a.permutation(2) {|x| print x }   # Prints "[1,2][1,3][2,1][2,3][3,1][3,2]"

# Iterate all possible 2-element subsets (order does not matter)
a.combination(2) {|x| print x }   # Prints "[1, 2][1, 3][2, 3]"
```

```
# Return the Cartesian product of the two sets
a.product(['a','b'])            # => [[1,"a"],[1,"b"],[2,"a"],[2,"b"],[3,"a"],[3,"b"]]
[1,2].product([3,4],[5,6])      # => [[1,3,5],[1,3,6],[1,4,5],[1,4,6], etc... ]
```

9.5.2.8 关联数组方法

使用 assoc 和 rassoc 方法，你可以把数组作为关联数组或哈希表来使用。这时，该数组必须是数组的数组，典型的例子如下：

```
[[key1, value1], [key2, value2], [key3, value3], ...]
```

Hash 类定义了一些方法，用于把一个哈希表转换为这种形式的嵌套数组。assoc 方法查找第一个元素与给定参数匹配的嵌套数组，然后返回这个嵌套数组。rassoc 做同样的工作，不过它返回的是第二个元素匹配的嵌套数组：

```
h = { :a => 1, :b => 2}   # Start with a hash
a = h.to_a                # => [[:b,2], [:a,1]]: associative array
a.assoc(:a)               # => [:a,1]: subarray for key :a
a.assoc(:b).last          # => 2: value for key :b
a.rassoc(1)               # => [:a,1]: subarray for value 1
a.rassoc(2).first         # => :b: key for value 2
a.assoc(:c)               # => nil
a.transpose               # => [[:a, :b], [1, 2]]: swap rows and cols
```

9.5.2.9 数组杂项方法

Array 定义了一些杂项方法，它们很难被放入前面的各个类别中：

```
# Conversion to strings
[1,2,3].join              # => "123": convert elements to string and join
[1,2,3].join(", ")        # => "1, 2, 3": optional delimiter
[1,2,3].to_s              # => "[1, 2, 3]" in Ruby 1.9
[1,2,3].to_s              # => "123" in Ruby 1.8
[1,2,3].inspect           # => "[1, 2, 3]": better for debugging in 1.8

# Binary conversion with pack. See also String.unpack.
[1,2,3,4].pack("CCCC")    # => "\001\002\003\004"
[1,2].pack('s2')          # => "\001\000\002\000"
[1234].pack("i")          # => "\322\004\000\000"

# Other methods
[0,1]*3                   # => [0,1,0,1,0,1]: * operator repeats
[1, [2, [3]]].flatten     # => [1,2,3]: recursively flatten; also flatten!
[1, [2, [3]]].flatten(1)  # => [1,2,[3]]: specify # of levels; Ruby 1.9
[1,2,3].reverse           # => [3,2,1]: also reverse!
a=[1,2,3].zip([:a,:b,:c]) # => [[1,:a],[2,:b],[3,:c]]: Enumerable method
a.transpose               # => [[1,2,3],[:a,:b,:c]]: swap rows/cols
```

9.5.3 哈希表

Hashes

在第 3.4 节中我们介绍了哈希表，解释了哈希表字面量句法，以及如何用[]和[]=操作符存取哈希表中的键/值对。本节将介绍 Hash 对象 API 的更多细节。哈希表对象使用了与数组一样的方括号操作符，而且你将注意到很多 Hash 类的方法与 Array 的方法很相似。

9.5.3.1 创建哈希表对象

哈希表可以用字面量、Hash.new 方法或 Hash 类自身的[]操作符进行创建：

```
{ :one => 1, :two => 2 } # Basic hash literal syntax
{ :one, 1, :two, 2 }     # Same, with deprecated Ruby 1.8 syntax
{ one: 1, two: 2 }       # Same, Ruby 1.9 syntax. Keys are symbols.
{}                       # A new, empty, Hash object
Hash.new                 # => {}: creates empty hash
Hash[:one, 1, :two, 2]   # => {one:1, two:2}
```

在第 6.4.4 节中，我们提到，在哈希表字面量作为方法调用的最后一个参数时，可以省略字面量两端的花括号：

```
puts :a=>1, :b=>2     # Curly braces omitted in invocation
puts a:1, b:2         # Ruby 1.9 syntax works too
```

9.5.3.2 哈希表索引和成员判别

哈希表可以高效查找给定键所对应的值，也可以（但不是很高效）通过值来查找关联的键。不过，要注意的是，如果多个主键都映射了同一个值，这时会返回任意一个主键：

```
h = { :one => 1, :two => 2 }
h[:one]      # => 1: find value associated with a key
h[:three]    # => nil: the key does not exist in the hash
h.assoc :one # => [:one, 1]: find key/value pair. Ruby 1.9.

h.index 1    # => :one: search for key associated with a value
h.index 4    # => nil: no mapping to this value exists
h.rassoc 2   # => [:two, 2]: key/value pair matching value. Ruby 1.9.
```

Hash 定义了多个等价的方法用于进行成员判定：

```
h = { :a => 1, :b => 2 }
# Checking for the presence of keys in a hash: fast
h.key?(:a)       # true: :a is a key in h
h.has_key?(:b)   # true: has_key? is a synonym for key?
h.include?(:c)   # false: include? is another synonym
h.member?(:d)    # false: member? is yet another synonym

# Checking for the presence of values: slow
h.value?(1)      # true: 1 is a value in h
h.has_value?(3)  # false: has_value? is a synonym for value?
```

fetch 方法在查询哈希表对象的值时，与[]是等价的，它还可以提供选项来处理主键不存在的情况：

```
h = { :a => 1, :b => 2 }
h.fetch(:a)      # => 1: works like [] for existing keys
h.fetch(:c)      # Raises IndexError for nonexistent key
h.fetch(:c, 33)  # => 33: uses specified value if key is not found
h.fetch(:c) {|k| k.to_s } # => "c": calls block if key not found
```

如果想一次提取多个值，可以使用 values_at 方法：

```
h = { :a => 1, :b => 2, :c => 3 }
h.values_at(:c)          # => [3]: values returned in an array
h.values_at(:a, :b)      # => [1, 2]: pass any # of args
h.values_at(:d, :d, :a)  # => [nil, nil, 1]
```

可以用 select 方法以块来选择一组键和值：

```
h = { :a => 1, :b => 2, :c => 3 }
h.select {|k,v| v % 2 == 0 } # => [:b,2] Ruby 1.8
h.select {|k,v| v % 2 == 0 } # => {:b=>2} Ruby 1.9
```

这个方法覆盖了 Enumerable.select 方法，在 Ruby 1.8 中，select 方法返回一个键/值对的数组，在 Ruby 1.9 中修改了这个行为，它返回所选键/值对的哈希对象。

9.5.3.3 在哈希表对象中存储键和值

可以用[]=操作符（或同义词方法 store）把一个值与哈希对象中的一个键关联起来：

```
h = {}           # Start with an empty hash
h[:a] = 1        # Map :a=>1.  h is now {:a=>1}
h.store(:b,2)    # More verbose: h is now {:a=>1, :b=>2}
```

如果要用另一个哈希对象中的键/值对替换一个哈希对象中的全部键/值对，可以用 replace 方法：

```
# Replace all of the pairs in h with those from another hash
h.replace({1=>:a, 2=>;b}  # h is now equal to the argument hash
```

用 merge、merge!和 update 方法可以把两个哈希对象合并成一个：

```
# Merge hashes h and j into new hash k.
# If h and j share keys, use values from j
k = h.merge(j)
{:a=>1,:b=>2}.merge(:a=>3,:c=>3)  # => {:a=>3,:b=>2,:c=>3}
h.merge!(j)   # Modifies h in place.

# If there is a block, use it to decide which value to use
h.merge!(j) {|key,h,j| h }        # Use value from h
h.merge(j) {|key,h,j| (h+j)/2 }   # Use average of two values

# update is a synonym for merge!
h = {a:1,b:2}     # Using Ruby 1.9 syntax and omitting braces
```

```
h.update(b:4,c:9) {|key,old,new| old }  # h is now {a:1, b:2, c:9}
h.update(b:4,c:9) # h is now {a:1, b:4, c:9}
```

9.5.3.4 删除哈希表条目

不能简单地把对应值设为 nil 来删除一个主键，而应该使用 delete 方法：

```
h = {:a=>1, :b=>2}
h[:a] = nil       # h now holds {:a=> nil, :b=>2 }
h.include? :a     # => true
h.delete :b       # => 2: returns deleted value: h now holds {:a=>nil}
h.include? :b     # => false
h.delete :b       # => nil: key not found
# Invoke block if key not found
h.delete(:b) {|k| raise IndexError, k.to_s } # IndexError!
```

可以用 delete_if 方法和 reject! 迭代器（及 reject 迭代器，它在接收者的拷贝上进行操作）一次删除哈希对象的多个键/值对。注意，reject 方法覆盖了 Enumerable 中的同名方法，它不再返回数组，而返回一个哈希对象：

```
h = {:a=>1, :b=>2, :c=>3, :d=>"four"}
h.reject! {|k,v| v.is_a? String } # => {:a=>1, :b=>2, :c=>3 }
h.delete_if {|k,v| k.to_s < 'b' } # => {:b=>2, :c=>3 }
h.reject! {|k,v| k.to_s < 'b' }   # => nil: no change
h.delete_if {|k,v| k.to_s < 'b' } # => {:b=>2, :c=>3 }: unchanged hash
h.reject {|k,v| true }            # => {}: h is unchanged
```

最后，可以使用 clear 方法删除哈希表中的全部键/值对。这个方法没有以感叹号结尾，但是它的确会修改接收者对象本身：

```
h.clear    # h is now {}
```

9.5.3.5 从哈希表中获得数组

Hash 定义了很多方法用于从哈希表中提取数据并放入数组中：

```
h = { :a=>1, :b=>2, :c=>3 }
# Size of hash: number of key/value pairs
h.length     # => 3
h.size       # => 3: size is a synonym for length
h.empty?     # => false
{}.empty?    # => true

h.keys       # => [:b, :c, :a]: array of keys
h.values     # => [2,3,1]: array of values
h.to_a       # => [[:b,2],[:c,3],[:a,1]]: array of pairs
h.flatten    # => [:b, 2, :c, 3, :a, 1]: flattened array. Ruby 1.9
h.sort       # => [[:a,1],[:b,2],[:c,3]]: sorted array of pairs
h.sort {|a,b| a[1]<=>b[1] } # Sort pairs by value instead of key
```

9.5.3.6 哈希表迭代器

因为 Hash 类也是一个 Enumerable 对象，并定义了很多有用的迭代器，所以通常无须提取哈希表的主键、值及键/值对到数组中。在 Ruby 1.8 中，哈希对象迭代的顺序是不确定的；而在 Ruby 1.9 中，哈希对象的元素按照插入的顺序进行迭代，如下所示：

```
h = { :a=>1, :b=>2, :c=>3 }

# The each() iterator iterates [key,value] pairs
h.each {|pair| print pair }       # Prints "[:a, 1][:b, 2][:c, 3]"

# It also works with two block arguments
h.each do |key, value|
  print "#{key}:#{value} "        # Prints "a:1 b:2 c:3"
end

# Iterate over keys or values or both
h.each_key {|k| print k }         # Prints "abc"
h.each_value {|v| print v }       # Prints "123"
h.each_pair {|k,v| print k,v }    # Prints "a1b2c3". Like each
```

each 迭代器迭代出一个包含主键和值的数组，代码块调用句法使这个数组可以自动扩展到 key 和 value 两个参数上。在 Ruby 1.8 中，each_pair 迭代器把 key 和 value 作为两个独立的值来迭代（这会带来少许的性能提升），而在 Ruby 1.9 中，each_pair 只是作为 each 的同义词。

尽管 shift 不是一个迭代器，它还是可以用于迭代一个哈希对象的全部键/值对。就像数组中的同名方法那样，它删除并返回哈希对象的一个元素（在本例中是一个[key, value]的数组）：

```
h = { :a=> 1, :b=>2 }
print h.shift[1] while not h.empty?   # Prints "12"
```

9.5.3.7 默认值

一般情况下，如果某个键没有关联值，在查询时会返回 nil：

```
empty = {}
empty["one"]   # nil
```

不过，可以通过为哈希对象设定一个默认值对这种行为进行修改：

```
empty = Hash.new(-1)      # Specify a default value when creating hash
empty["one"]              # => -1
empty.default = -2        # Change the default value to something else
empty["two"]              # => -2
empty.default             # => -2: return the default value
```

也可以不设定一个特定值，而是提供一块代码来为没有关联值的主键计算值：

```
# If the key is not defined, return the successor of the key.
plus1 = Hash.new {|hash, key| key.succ }
plus1[1]          # 2
plus1["one"]      # "onf": see String.succ
plus1.default_proc    # Returns the Proc that computes defaults
plus1.default(10)     # => 11: default returned for key 10
```

如果定义了这样一个代码块，我们一般把这个计算出来的值关联给相应的主键，这样当该主键再次被查询时，无需再进行一次计算。这可以通过惰性求值来轻松实现（这也解释了为什么默认代码块会与主键一起被传给哈希对象）：

```
# This lazily initialized hash maps integers to their factorials
fact = Hash.new {|h,k| h[k] = if k > 1: k*h[k-1] else 1 end }
fact          # {}: it starts off empty
fact[4]       # 24: 4! is 24
fact          # {1=>1, 2=>2, 3=>6, 4=>24}: the hash now has entries
```

注意，为哈希对象设置默认值会覆盖使用代码块的 Hash.new 构造方法。

如果对设置哈希对象的默认值不感兴趣，或者想用自己设置的值取代默认值，你可以不使用方括号而使用 fetch 方法，fetch 方法在前面已经介绍过：

```
fact.fetch(5)   # IndexError: key not found
```

9.5.3.8 哈希码、主键相等性和可变主键

如果一个对象想作为哈希对象的主键使用，它必须定义一个 hash 方法，该方法返回一个整型的“哈希码（hashcode）”。那些没有覆盖 eql?方法的类可以直接使用由 Object 继承而来的 hash 方法，然而，如果定义了自己的 eql?方法来检测对象相等性，就必须再相应定义一个 hash 方法。如果两个不同的对象被认为是相等的，它们的 hash 方法必须返回同样的值。理想情况下，两个不相等的对象应该具有不同的哈希码。这个主题在第 3.4.2 节中已经进行了讲述，并且在第 7.1.9 节中给出了一个 hash 实现的例子。

一般情况下，哈希对象使用 eql?方法来判定主键的相等性。不过，在 Ruby 1.9 中，可以对 Hash 对象调换用 compare_by_identity 方法，强制它使用 equal?方法判定主键的相等性。在此之后，hash 方法将使用 object_id 作为哈希码。一旦如此，则无法使一个哈希对象返回到普通的相等性测试方式。使用 compare_by_identity?判定方法可以判定一个哈希对象是使用相等性（equality）还是标识符（identity）进行比较。记住，两个具有相同字符的符号字面量将被求值为相同的对象，但是具有同样字符的字符串字面量则被求值为不同对象，因此符号字面量在作为主键时，可以用标识符进行比较，而字符串则不行。

如第 3.4.2 节中所提到的，当使用一个可变对象作为哈希表主键时，必须随时保持警惕。（String 是一个特例：Hash 类为字符串主键做了一个私有的内部拷贝。）如果使用了可变主键，并

且修改了其中一个对象，你就必须调用哈希对象的 rehash 方法，以确保它能正常工作：

```
key = {:a=>1}          # This hash will be a key in another hash!
h = { key => 2 }       # This hash has a mutable key
h[key]                 # => 2: get value associated with key
key.clear              # Mutate the key
h[key]                 # => nil: no value found for mutated key
h.rehash               # Fix up the hash after mutation
h[key]                 # => 2: now the value is found again
```

9.5.3.9 哈希对象的杂项方法

invert 方法不能放入前面所述的任何类别中，它对一个哈希对象中的键和值进行转换：

```
h = {:a=>1, :b=>2}
h.invert       # => {1=>:a, 2=>:b}: swap keys and values
```

跟 Array 类相似，Hash.to_s 方法在 Ruby 1.8 中没有多大用处，一般要用 inspect 方法把一个哈希对象转换为哈希表字面量形式的字符串。在 Ruby 1.9 中，to_s 方法和 inspect 方法是一样的：

```
{:a=>1, :b=>2}.to_s       # => "a1b2" in Ruby 1.8; "{:a=>1, :b=>2}" in 1.9
{:a=>1, :b=>2}.inspect    # => "{:a=>1, :b=>2}" for both versions
```

9.5.4 集合
Sets

集合就是一组无重复值的集合。与数组不同，集合中的元素没有顺序。一个哈希表可以看作是一个键/值对的集合，反过来，集合也可以用哈希表实现：可以用键来存储集合元素，而忽略对应的值。**排序集合**（*sorted set*）是给元素加入了顺序的集合（不过不能像数组一样进行随机访问）。集合的特性之一是可以快速完成成员检测、插入和删除的操作。

Ruby 没有提供内置的集合类型，不过在标准库中包含了 Set 和 SortedSet 类，你可以通过如下代码引入这两个类：

```
require 'set'
```

Set 的 API 在很多方面都与 Array 和 Hash 的 API 相似，很多 Set 方法和操作符可以接受任意多的 Enumerable 对象作为参数。

SortedSet 类

SortedSet 继承自 Set，而且自己没有定义任何新方法，它只是保证集合的元素可以用有序的方式被迭代（或打印及转换为数组时）。SortedSet 不支持用代码块方式定制集合元素的比较方式，而要求每个集合元素必须支持用默认的<=>操作符进行比较。因为 SortedSet 的 API 与 Set 的 API 没有什么不同，所以这里不做特别介绍。

9.5.4.1 创建集合

因为 Set 不是 Ruby 内核类，因此没有字面量句法用于创建集合。set 库为 Enumerable 模块加入了一个 to_set 方法，这样你可以从任何可枚举的对象来创建集合：

```
(1..5).to_set                  # => #<Set: {5, 1, 2, 3, 4}>
[1,2,3].to_set                 # => #<Set: {1, 2, 3}>
```

或者，也可以把任何可枚举的对象传递给 Set.new 方法。如果提供了代码块，它可以（就像在 map 迭代器中那样）在元素加入集合前对之进行预处理：

```
Set.new(1..5)                  # => #<Set: {5, 1, 2, 3, 4}>
Set.new([1,2,3])               # => #<Set: {1, 2, 3}>
Set.new([1,2,3]) {|x| x+1}     # => #<Set: {2, 3, 4}>
```

如果不想先把元素放到数组或任何可枚举的对象中，你可以用 Set 类的[]操作符直接创建集合：

```
Set["cow", "pig", "hen"]   # => #<Set: {"cow", "pig", "hen"}>
```

9.5.4.2 集合的测试、比较和联合

集合最常见的操作是成员检测方法：

```
s = Set.new(1..3)      # => #<Set: {1, 2, 3}>
s.include? 1           # => true
s.member? 0            # => false: member? is a synonym
```

也可以测试一个集合是否是其他集合的子集。如果一个集合 S 的所有元素都是集合 T 的元素，那么 S 是 T 的**子集**。如果两个集合相等，那么它们互为子集和超集。如果 S 是 T 的子集但是不等于 T，那么 S 是 T 的**真子集**（*proper subset*）。在下面的例子中，T 是 S 的**真超集**（*proper superset*）：

```
s = Set[2, 3, 5]
t = Set[2, 3, 5, 7]
s.subset? t                # => true
t.subset? s                # => false
s.proper_subset? t         # => true
t.superset? s              # => true
t.proper_superset? S       # => true
s.subset? s                # => true
s.proper_subset? s         # => false
```

Set 定义了与 Array 和 Hash 一样的大小方法：

```
s = Set[2, 3, 5]
s.length                   # => 3
s.size                     # => 3: a synonym for length
```

```
s.empty?                    # => false
Set.new.empty?              # => true
```


可以通过联合两个已有集合来创建新的集合。联合方式有多种，Set 定义了&、|、-和^操作符（及相应的别名方法）来表示它们：

```
# Here are two simple sets
primes = Set[2, 3, 5, 7]
odds = Set[1, 3, 5, 7, 9]

# The intersection is the set of values that appear in both
primes & odds                  # => #<Set: {5, 7, 3}>
primes.intersection(odds)      # this is an explicitly named alias

# The union is the set of values that appear in either
primes | odds                  # => #<Set: {5, 1, 7, 2, 3, 9}>
primes.union(odds)             # an explicitly named alias

# a-b: is the elements of a except for those also in b
primes-odds                    # => #<Set: {2}>
odds-primes                    # => #<Set: {1, 9}>
primes.difference(odds)        # A named method alias

# a^b is the set of values that appear in one set but not both: (a|b)-(a&b)
primes ^ odds                  # => #<Set: {1, 2, 9}>
```

Set 类还为其中一些方法定义了可变版本，我们马上会讲到它们。

9.5.4.3 增加和删除集合元素

本节将介绍在一个集合中增加和删除元素的方法，它们是可变方法，可直接修改接收者对象，而不是创建一个修改过的拷贝并保持原有对象不变。因为这些方法没有相应的不变版本，所以并没有以感叹号结尾。

<<操作符用于向集合增加一个元素：

```
s = Set[]              # start with an empty set
s << 1                 # => #<Set: {1}>
s.add 2                # => #<Set: {1, 2}>: add is a synonym for <<
s << 3 << 4 << 5       # => #<Set: {5, 1, 2, 3, 4}>: can be chained
s.add 3                # => #<Set: {5, 1, 2, 3, 4}>: value unchanged
s.add? 6               # => #<Set: {5, 6, 1, 2, 3, 4}>
s.add? 3               # => nil: the set was not changed
```

如果想对元素增加多个元素，你可以用 merge 方法，它接受任何可枚举的对象作为参数。merge 可以被看作是 union 方法的可变版本：

```
s = (1..3).to_set      # => #<Set: {1, 2, 3}>
s.merge(2..5)          # => #<Set: {5, 1, 2, 3, 4}>
```

要删除集合中的元素，你可以用 delete 或 delete?方法，它们与 add 或 add?方法对应，只是没有等价的操作符：

```
s = (1..3).to_set     # => #<Set: {1, 2, 3}>
s.delete 1            # => #<Set: {2, 3}>
s.delete 1            # => #<Set: {2, 3}>: unchanged
s.delete? 1           # => nil: returns nil when no change
s.delete? 2           # => #<Set: {3}>: otherwise returns set
```

如果想一次删除多个集合元素，你可以用 subtract 方法。它可以接受任意可枚举的对象作为参数，可以被看作是 difference 方法的可变版本：

```
s = (1..3).to_set     # => #<Set: {1, 2, 3}>
s.subtract(2..10)     # => #<Set: {1}>
```

如果要选择性地从集合中删除元素，你可以用 delete_if 或 reject!方法。与 Array 和 Hash 类中的同名方法相似，除了在集合未改变时返回值不同，这两个方法是等价的。delete_if 方法始终返回接收者对象；而 reject!方法在集合有变化时返回接收者对象，否则返回 nil：

```
primes = Set[2, 3, 5, 7]          # set of prime numbers
primes.delete_if {|x| x%2==1}     # => #<Set: {2}>: remove odds
primes.delete_if {|x| x%2==1}     # => #<Set: {2}>: unchanged
primes.reject! {|x| x%2==1}       # => nil: unchanged

# Do an in-place intersection like this:
s = (1..5).to_set
t = (4..8).to_set
s.reject! {|x| not t.include? x}  # => #<Set: {5, 4}>
```

最后，clear 和 replace 方法与数组和哈希表中的同名方法功能相似：

```
s = Set.new(1..3)     # Initial set
s.replace(3..4)       # Replace all elements.  Argument is any enumerable
s.clear               # => #<Set: {}>
s.empty?              # => true
```

9.5.4.4 集合迭代器

集合是 Enumerable 对象，并且 Set 类定义了 each 迭代器，将枚举每个集合元素一次。在 Ruby 1.9 中，Set 与 Hash 类一样，按照元素插入的顺序进行迭代。而在 Ruby 1.9 前，迭代的顺序是任意的。对于 SortedSet，迭代的顺序是升序的。另外，map!迭代器用一个代码块对集合的每个元素进行转换，直接更新接收者对象。collect!方法是它的同义词：

```
s = Set[1, 2, 3, 4, 5]    # => #<Set: {5, 1, 2, 3, 4}>
s.each {|x| print x }     # prints "51234": arbitrary order before Ruby 1.9
s.map! {|x| x*x }         # => #<Set: {16, 1, 25, 9, 4}>
s.collect! {|x| x/2 }     # => #<Set: {0, 12, 2, 8, 4}>
```

9.5.4.5 集合杂项方法

Set 定义了一些强大的方法用于把集合分割为子集，以及把一组子集铺开成为一个大的集合。另外，它还定义了一些不那么强大的方法，我们首先介绍它们：

```
s = (1..3).to_set
s.to_a            # => [1, 2, 3]
s.to_s            # => "#<Set:0xb7e8f938>": not useful
s.inspect         # => "#<Set: {1, 2, 3}>": useful
s == Set[3,2,1]   # => true: uses eql? to compare set elements
```

classify 方法接受一个代码块，集合的每个元素被依次传给这个代码块。它的返回值是一个哈希表，这个哈希表的主键是代码块的返回值，而对应元素构成的子集成为该键对应的值。

```
# Classify set elements as even or odd
s = (0..3).to_set          # => #<Set: {0, 1, 2, 3}>
s.classify {|x| x%2}       # => {0=>#<Set: {0, 2}>, 1=>#<Set: {1, 3}>}
```

divide 方法与 classify 相似，不过它的返回值不是哈希表，而是子集的集合：

```
s.divide {|x| x%2}  # => #<Set: {#<Set: {0, 2}>, #<Set: {1, 3}>}>
```

如果关联的代码块有两个参数，divide 方法的行为会完全不同。这时，如果集合中的两个值属于同一个子集，那么代码块的返回值为真，否则返回假：

```
s = %w[ant ape cow hen hog].to_set    # A set of words
s.divide {|x,y| x[0] == y[0]}         # Divide into subsets by first letter
# => #<Set:{#<Set:{"hog", "hen"}>,    # <Set:{"cow"}>, #<Set:{"ape", "ant"}>}>
```

如果一个集合包含另一集合（所包含的集合还可以包含集合，如此递归），你可以用 flatten 方法把它展开，返回所有集合的并集。flatten!方法是它的可变版本：

```
s = %w[ant ape cow hen hog].to_set    # A set of words
t = s.divide {|x,y| x[0] == y[0]}     # Divide it into subsets
t.flatten!                            # Flatten the subsets
t == s                                # => true
```

9.6 文件和目录

Files and Directories

File 类定义了很多类方法用于处理文件系统中的文件：例如，获取给定文件的长度、检测文件的存在性及把文件名从目录名中剥离出来。这些都是类方法，与 File 对象无关，只是用字符串给定文件名即可。相似地，Dir 类也定义了一些类方法用于处理文件系统的目录。下面的子小节将演示如何完成如下工作：

- 处理和操作文件名及目录名；

- 列举目录；
- 获得文件的类型、大小、修改时间和其他属性；
- 对文件和目录执行删除、重命名和类似操作。

注意这里描述的方法只查询和操作文件本身，并不涉及读写文件的内容。读写文件的内容将在第 9.7 节中介绍。

> **在 Ruby 1.9 中指定文件名**
>
> 本章描述的许多与文件和目录相关的方法都需要一个或多个文件名参数，一般情况下，用字符串表示文件名和目录路径。不过在 Ruby 1.9 中，只要一个对象定义了一个返回字符串的 to_path 方法，你就可以用它们来表示文件和目录名。

9.6.1 文件和目录名
File and Directory Names

File 和 Dir 的类方法对指定名字的文件和目录进行操作。Ruby 使用 Unix 风格的/作为目录分隔符。即使在 Windows 平台的 Ruby 上，你也可以使用这种前斜杠形式。不过，在 Windows 上，Ruby 也可以处理那些使用反斜杠和包含盘符的路径名。File::SEPARATOR 应该在所有实现中都被定义为“/”，而 File::ALT_SEPARATOR 在 Windows 平台上为“\”，而在其他平台上为 nil。

File 类定义了一些操作文件名的方法：

```
full = '/home/matz/bin/ruby.exe'
file=File.basename(full)      # => 'ruby.exe': just the local filename
File.basename(full, '.exe')   # => 'ruby': with extension stripped
dir=File.dirname(full)        # => '/home/matz/bin': no / at end
File.dirname(file)            # => '.': current directory
File.split(full)              # => ['/home/matz/bin', 'ruby.exe']
File.extname(full)            # => '.exe'
File.extname(file)            # => '.exe'
File.extname(dir)             # => ''
File.join('home','matz')      # => 'home/matz': relative
File.join('','home','matz')   # => '/home/matz': absolute
```

File.expand_path 用于把相对路径转换为绝对路径。如果提供了第二个可选参数，它会作为目录叠加到第一个参数前，结果仍然是一个绝对路径。如果第二个参数用 Unix 风格的~打头，表明该目录是当前用户（或指定用户）的主目录；否则，这个目录被作为当前工作目录的相对路径处理（参见下面的 Dir.chdir 来查看如何修改当前工作目录）：

```
Dir.chdir("/usr/bin")      # Current working directory is "/usr/bin"
File.expand_path("ruby")          # => "/usr/bin/ruby"
File.expand_path("~/ruby")        # => "/home/david/ruby"
File.expand_path("~matz/ruby")    # => "/home/matz/ruby"
```

```
File.expand_path("ruby", "/usr/local/bin")    # => "/usr/local/bin/ruby"
File.expand_path("ruby", "../local/bin")      # => "/usr/local/bin/ruby"
File.expand_path("ruby", "~/bin")             # => "/home/david/bin/ruby"
```

File.identical?方法用于判定两个文件名是否指向同一文件。当两个文件名相同时也许没太大用处，不过当两个文件名不同时，它会很有用处。两个不同的名字可能指向同一个文件。例如，一个用绝对路径，一个用相对路径；一个文件名可能用..进入父目录然后再回来；或者，一个文件名可能是另一个文件名的符号链接或快捷方式（或某些平台上的硬链接（hard link））。不过要注意的是，只有在两个文件名指向同一个文件，而且那个文件确实存在时，File.identical?方法才返回真。还要注意 File.identical?不会像 File.expand_path 方法那样展开用户主目录字符~：

```
File.identical?("ruby", "ruby")           # => true if the file exists
File.identical?("ruby", "/usr/bin/ruby")  # => true if CWD is /usr/bin
File.identical?("ruby", "../bin/ruby")    # => true if CWD is /usr/bin
File.identical?("ruby", "ruby1.9")        # => true if there is a link
```

最后，File.fnmatch 检查一个文件名是否匹配给定的模式。这个模式不是正则表达式，而是像外壳程序中使用的文件匹配模式。“?”匹配单个字符；“*”匹配任意多的字符；“**”会匹配任意层次的目录；方括号与正则表达式中的一样，表示任意一个其中的字符。fnmatch 不允许在花括号中使用替换符（下面要介绍的 Dir.glob 方法可以）。fnmatch 方法通常应该把第三个参数设置为 File::FNM_PATHNAME，这可以阻止“*”号匹配“/”目录。如果想查找以“.”开头的“隐藏”文件或目录，应该加上 File::FNM_DOTMATCH。下面仅给出几个例子，如果想得到 fnmatch 方法的具体细节，你可以使用 ri File.fnmatch 命令。注意 File.fnmatch? 是它的同义词方法：

```
File.fnmatch("*.rb", "hello.rb")      # => true
File.fnmatch("*.[ch]", "ruby.c")      # => true
File.fnmatch("*.[ch]", "ruby.h")      # => true
File.fnmatch("?.txt", "ab.txt")       # => false
flags = File::FNM_PATHNAME | File::FNM_DOTMATCH
File.fnmatch("lib/*.rb", "lib/a.rb", flags)       # => true
File.fnmatch("lib/*.rb", "lib/a/b.rb", flags)     # => false
File.fnmatch("lib/**/*.rb", "lib/a.rb", flags)    # => true
File.fnmatch("lib/**/*.rb", "lib/a/b.rb", flags)  # => true
```

9.6.2 列举目录
Listing Directories

列举目录内容最简单的方式是用 Dir.entries 方法或 Dir.foreach 迭代器：

```
# Get the names of all files in the config/ directory
filenames = Dir.entries("config")         # Get names as an array
Dir.foreach("config") {|filename| ... }   # Iterate names
```

这些方法不会以某种特定方式对返回的文件名进行排序。在返回的文件名中，会包含“.”（当前目录）和“..”（父目录）这样的名字（在类 Unix 平台中）。如果想返回匹配某种模式的文件列表，你可以用 Dir.[]操作符：

```
Dir['*.data']         # Files with the "data" extension
Dir['ruby.*']         # Any filename beginning with "ruby."
Dir['?']              # Any single-character filename
Dir['*.[ch]']         # Any file that ends with .c or .h
Dir['*.{java,rb}']    # Any file that ends with .java or .rb
Dir['*/*.rb']         # Any Ruby program in any direct sub-directory
Dir['**/*.rb']        # Any Ruby program in any descendant directory
```

比 Dir[]更强大的方法是 Dir.glob 方法（glob 是一个 Unix 术语，是一个用于进行文件名匹配的 Shell 命令）。默认情况下，这个方法与 Dir[]类似，不过如果给它传入一个代码块，它就不再是简单传回一个数组，而是把匹配的文件名一个个传给相应代码块来处理。另外，glob 方法的第二个参数是可选的，如果为它传入 File::FNM_DOTMATCH 常量（参见前面的 File.fnmatch 方法），那么结果将包含以“.”打头的文件名（在 Unix 系统中，这些文件是隐藏文件，默认情况下，返回值不会包含它们）。

```
Dir.glob('*.rb') {|f| ... }       # Iterate all Ruby files
Dir.glob('*')                     # Does not include names beginning with '.'
Dir.glob('*',File::FNM_DOTMATCH)  # Include . files, just like Dir.entries
```

这里介绍的目录列举方法，以及 File 和 Dir 中处理相对路径的方法，其路径都是相对于“当前工作目录”的，这个目录名对 Ruby 解释器进程全局可用，你可以用 getwd 和 chdir 方法对 CWD 进行读写：

```
puts Dir.getwd            # Print current working directory
Dir.chdir("..")           # Change CWD to the parent directory
Dir.chdir("../sibling")   # Change again to a sibling directory
Dir.chdir("/home")        # Change to an absolute directory
Dir.chdir                 # Change to user's home directory
home = Dir.pwd            # pwd is an alias for getwd
```

如果为 chdir 方法传入一个代码块，在代码块结束后，当前工作目录将恢复其原始值。不过，尽管可能只是在短时间内对当前工作目录的值进行了修改，它仍然是全局性的，会对其他的线程造成影响。两个线程不能同时调用 Dir.chdir 方法。

9.6.3 测试文件
Testing Files

File 类定义了一系列方法来获得指定文件或目录的元数据，其中不少方法会返回与操作系统相关的底层信息。这里仅对最常用的一些方法进行演示，你可以对 File 和 File::Stat 类使用 *ri* 命令来获取这些方法完整列表。

下面这些简单的方法用于返回文件的基本信息，大多数方法返回一个判定值(true 或 false)：

```
f = "/usr/bin/ruby"       # A filename for the examples below

# File existence and types.
File.exist?(f)            # Does the named file exist? Also: File.exists?
File.file?(f)             # Is it an existing file?
File.directory?(f)        # Or is it an existing directory?
File.symlink?(f)          # Either way, is it a symbolic link?

# File size methods. Use File.truncate to set file size.
File.size(f)              # File size in bytes.
File.size?(f)             # Size in bytes or nil if empty file.
File.zero?(f)             # True if file is empty.

# File permissions. Use File.chmod to set permissions (system dependent).
File.readable?(f)         # Can we read the file?
File.writable?(f)         # Can we write the file? No "e" in "writable"
File.executable?(f)       # Can we execute the file?
File.world_readable?(f)   # Can everybody read it? Ruby 1.9.
File.world_writable?(f)   # Can everybody write it? Ruby 1.9.

# File times/dates. Use File.utime to set the times.
File.mtime(f)             # => Last modification time as a Time object
File.atime(f)             # => Last access time as a Time object
```

另外一种用于确定文件类型（文件、目录、符号链接等）的方法是 `ftype`，它返回字符串描述的文件类型。假定*/usr/bin/ruby* 是*/usr/bin/ruby1.9* 的符号链接（或快捷方式）：

```
File.ftype("/usr/bin/ruby")       # => "link"
File.ftype("/usr/bin/ruby1.9")    # => "file"
File.ftype("/usr/lib/ruby")       # => "directory"
File.ftype("/usr/bin/ruby3.0")    # SystemCallError: No such file or directory
```

如果关注一个文件的多种信息，使用 `stat` 或 `lstat` 方法会更高效（对符号链接，`stat` 返回所链接文件的信息，而 `lstat` 返回链接本身的信息）。这些方法返回一个 `File::Stat` 对象，这个对象中的实例方法名与 `File` 的类方法名相同（不过不带参数）。使用 `stat`，只用一次调用便可以从操作系统得到所有文件元数据，然后程序就可以从返回的 `File::Stat` 对象中获得各种需要的信息：

```
s = File.stat("/usr/bin/ruby")
s.file?               # => true
s.directory?          # => false
s.ftype               # => "file"
s.readable?           # => true
s.writable?           # => false
s.executable?         # => true
s.size                # => 5492
s.atime               # => Mon Jul 23 13:20:37 -0700 2007
```

对 `File::Stat` 使用 *ri* 命令，你可以获得所有方法的完整列表。最后，还有一个通用文件测试方法 `Kernel.test`，它是为了和 Unix 的外壳命令 *test* 保持兼容而存在的。这个方法的

绝大多数功能已经被 File 中的类方法所分解，不过在已有的脚本中，你可能还会碰见它。可以用 *ri* 获得详细信息：

```
# Testing single files
test ?e, "/usr/bin/ruby"      # File.exist?("/usr/bin/ruby")
test ?f, "/usr/bin/ruby"      # File.file?("/usr/bin/ruby")
test ?d, "/usr/bin/ruby"      # File.directory?("/usr/bin/ruby")
test ?r, "/usr/bin/ruby"      # File.readable?("/usr/bin/ruby")
test ?w, "/usr/bin/ruby"      # File.writeable?("/usr/bin/ruby")
test ?M, "/usr/bin/ruby"      # File.mtime("/usr/bin/ruby")
test ?s, "/usr/bin/ruby"      # File.size?("/usr/bin/ruby")

# Comparing two files f and g
test ?-, f, g                 # File.identical(f,g)
test ?<, f, g                 # File(f).mtime < File(g).mtime
test ?>, f, g                 # File(f).mtime > File(g).mtime
test ?=, f, g                 # File(f).mtime == File(g).mtime
```

9.6.4 创建、删除、重命名文件及目录

Creating, Deleting, and Renaming Files and Directories

File 类没有为创建文件定义特殊的方法。要创建一个文件，你可以简单用“写”的方式打开一个文件，写入零个或多个字节，然后关闭即可。如果不想丢失现有文件的信息，你可以用追加方式打开：

```
# Create (or overwrite) a file named "test"
File.open("test", "w") {}
# Create (but do not clobber) a file named "test"
File.open("test", "a") {}
```

要改变文件名，你可使用 File.rename：

```
File.rename("test", "test.old")     # Current name, then new name
```

要创建一个符号链接，你可使用 File.symlink：

```
File.symlink("test.old", "oldtest") # Link target, link name
```

在支持“硬链接（hard link）”的系统上，你可以用 File.link 方法创建它：

```
File.link("test.old", "test2")   # Link target, link name
```

最后，要删除一个文件或链接，你可使用 File.delete 方法（或者同义词方法 File.unlink）：

```
File.delete("test2")   # May also be called with multiple args
File.unlink("oldtest") # to delete multiple named files
```

在支持截断（truncate）文件的系统上，File.truncate 方法可以用给定字节数（可以是 0）截取一个文件。用 File.utime 可以设置和读取文件的修改时间，用平台相关的 File.chmod 方法可以修改文件的访问权限：

```
f = "log.messages"            # Filename
atime = mtime = Time.now      # New access and modify times
File.truncate(f, 0)           # Erase all existing content
File.utime(atime, mtime, f)   # Change times
File.chmod(0600, f)           # Unix permissions -rw-------; note octal arg
```

用 Dir.mkdir 方法可以创建一个新目录，用 Dir.rmdir（或同义词方法 Dir.delete 或 Dir.unlink）可以删除一个目录，在删除前该目录必须为空：

```
Dir.mkdir("temp")                 # Create a directory
File.open("temp/f", "w") {}       # Create a file in it
File.open("temp/g", "w") {}       # Create another one
File.delete(*Dir["temp/*"])       # Delete all files in the directory
Dir.rmdir("temp")                 # Delete the directory
```

9.7 输入/输出
Input/Output

一个 IO 对象就是一个流，它是一个可读的信息源，内容可以是字节、字符或字节（或字符）的可写池。File 是 IO 的子类，IO 对象也表示“标准输入”和“标准输出”流，用于从控制台读入或向控制台写入信息。最后，套接字对象也是 IO 对象，用于网络通信（在本章后面将介绍）。

9.7.1 打开流
Opening Streams

在进行输入输出操作之前，首先必须创建一个 IO 对象。IO 类定义了 new、open、popen 和 pipe 这些工厂方法，不过它们都是与操作系统相关的方法，在这里不做介绍。下面的子小节将介绍那些更常用的获得 IO 对象的方式。（在 9.8 节还将介绍如何创建 IO 对象进行网络通信的例子。）

9.7.1.1 打开文件

IO 中最常用的操作是读写文件。File 类定义了一些工具方法（将在下面介绍）用于读取整个文件的内容。不过，通常的操作是打开一个文件，获得一个 File 对象，然后用 IO 中的方法读写文件。

用 File.open（或 File.new）打开一个文件。它的第一个参数是文件名，一般使用字符串，不过在 Ruby 1.9 中，也可以用任何实现 to_path 方法的对象。如果没有指定一个绝对路径，则使用相对于当前工作目录的路径。用斜杠对目录进行分隔——在 Windows 环境下，Ruby 自动把它们转换为反斜杠。File.open 的第二个参数用于指定文件的打开方式：

```
f = File.open("data.txt", "r")    # Open file data.txt for reading
out = File.open("out.txt", "w")   # Open file out.txt for writing
```

File.open 的第二个参数是指定“文件方式”的字符串，它必须是下表中的某个值，在其后加上“b”可以阻止在 Windows 平台下对换行符的自动转换。对于文本文件，你可以为文

件方式字符串加上字符编码模式。对于二进制文件，则应该加上“:binary”，这些内容将在第 9.7.2 节解释。

方式	描述
"r"	读方式打开文件。这是默认的方式
"r+"	读写方式打开文件，读写直接置于文件起始处，如果文件不存在则导致失败
"w"	写方式打开文件，会创建一个新文件，或者截取一个已有文件
"w+"	跟“w”相似，不过也允许读文件
"a"	追加写方式，如果文件存在，则在尾部追加写入新的内容
"a+"	跟“a”相似，不过也允许读文件

File.open（不过 File.new 不行）也可以带一个代码块。如果提供了代码块，File.open 方法不会返回相应的 File 对象，而把它传给代码块，在代码块结束后，该 File 对象会自动关闭，代码块的返回值则成为 File.open 方法的返回值：

```
File.open("log.txt", "a") do |log|          # Open for appending
  log.puts("INFO: Logging a message")       # Output to the file
end                                         # Automatically closed
```

9.7.1.2 Kernel.open 方法

Kernel 的 open 方法与 File.open 方法相似，不过更灵活。如果文件名以|打头，它就被视作一个系统命令，返回的流被读写该命令的进程使用。当然，它是与平台相关的：

```
# How long has the server been up?
uptime = open("|uptime") {|f| f.gets }
```

如果加载了 open-uri 库，那么 open 方法也可以用于读取 http 和 ftp 的 URL，就像打开文件一样：

```
require "open-uri"                          # Required library
f = open("http://www.davidflanagan.com/") # Webpage as a file
webpage = f.read                            # Read it as one big string
f.close                                     # Don't forget to close!
```

在 Ruby 1.9 中，如果 open 的参数对象实现了 to_open 方法，那么该方法将被调用，而且返回一个打开的 IO 对象。

9.7.1.3 StringIO 类

另一种获得 IO 对象的方法是使用 stringio 库，对字符串进行读写：

```
require "stringio"
input = StringIO.open("now is the time")  # Read from this string
```

```
buffer = ""
output = StringIO.open(buffer, "w")        # Write into buffer
```

StringIO 类并非 IO 的子类，不过它定义的很多方法都与 IO 中定义的相同。这样，由于可以进行 duck typing，很多时候你可以用一个 StringIO 对象替代 IO 对象。

9.7.1.4 预定义流

Ruby 预定义了很多流，无须创建或打开就可以直接使用。全局常量 STDIN、STDOUT 和 STDERR 分别代表标准输入流、标准输出流和标准错误流。默认情况下，这些流会连接控制台或某种终端窗口。不过，根据脚本调用时的环境，这些流可能使用文件或者甚至是另外一个进程，来作为输入源或输出目标。任何 Ruby 程序都可以从标准输入读取数据，也可以向标准输出（普通的程序输出）或标准错误（即使是标准输出被重定向到一个文件，错误信息也应该能看到）输出信息。全局变量$stdin、$stdout 和$stderr 最初被设为相应流常量的值，像 puts 和 print 这样的全局函数会默认向$stdout 输出信息，如果一个脚本修改了这些全局变量的值，那么这些方法的行为将被改变。不过，真正的“标准输出”仍然可以通过 STDOUT 来获得。

另外一个预定义流是 ARGF，或者被称为$<。对于那些在命令行中指定读入文件或从控制台读入数据的脚本来说，这个流可以用来简化脚本的编写。如果某个 Ruby 脚本有命令行参数（在 ARGV 或$* array 数组中），那么 ARGF 就像是这些文件连接起来构成的单个文件，用于进行读操作。为了让这个流可以正常工作，那些并非文件名的命令行参数必须被预先处理，然后从 ARGV 数组中被移除。如果 ARGV 数组为空，那么 ARGF 与 STDIN 功能相同（参见第 10.3.1 节获取 ARGF 流的更多细节）。

最后，DATA 流用于读取 Ruby 脚本后面出现的文本。当 Ruby 脚本包含__END__标记时，它表示程序文本的结束，该标记后的任何文本将被 DATA 流所读取。

9.7.2 流和编码方式
Streams and Encodings

在 Ruby 1.9 中，最重要的变化之一是支持多字节编码。在第 3.2 节中我们已经看到了在 String 类中有多处变化，在 IO 类中也有相似的变化。

在 Ruby 1.9 中，每个流可以有两个关联的编码方式，分别是外部编码和内部编码，分别可以通过 IO 对象的 external_encoding 和 internal_encoding 方法获得。外部编码是指存储在文件中的文本的编码方式，内部编码是指在 Ruby 中用于表示文本的编码方式。如果外

部编码也需要内部编码，那么就无须指定内部编码，从流中读入的字符串会具有与之相关联的外部编码（就像用 String 的 force_encoding 方法那样）；不过，如果想在内部使用不同的编码方式，你可以指定一个内部编码方式，这样，在读入文本时，Ruby 将从外部编码方式转换为内部编码方式，而在输出文本时，则从内部编码方式转换为外部编码方式。

可以用 set_encoding 方法对任何 IO 对象（包括管道和网络套接字）设定编码。如果给定两个参数，则表明一个是外部编码，另一个是内部编码。也可以用一个字符串来指定两个编码方式，它们用冒号分隔。但一般情况下，一个参数常用于表明只有外部编码。参数可以是字符串或 Encoding 对象，外部编码总是被首先指定，后面可以跟一个内部编码（可选），例如：

```
f.set_encoding("iso-8859-1", "utf-8")      # Latin-1, transcoded to UTF-8
f.set_encoding("iso-8859-1:utf-8")         # Same as above
f.set_encoding(Encoding::UTF-8)            # UTF-8 text
```

set_encoding 可以对任何类型的 IO 对象使用。不过，对于文件来说，最简单的方式是在打开文件时指定编码方式。你可以在文件打开方式的字符串后加上编码名称。例如：

```
in = File.open("data.txt", "r:utf-8");          # Read UTF-8 text
out = File.open("log", "a:utf-8");              # Write UTF-8 text
in = File.open("data.txt", "r:iso8859-1:utf-8"); # Latin-1 transcoded to UTF-8
```

注意，一般不需要对输出流指定两种编码方式，这时，String 对象指定的内部编码方式被用于写入流。

如果完全不指定任何编码方式，在读入文件时，Ruby 将使用默认的外部编码；在写文件或读写管道和套接字时，则默认使用无编码方式（比如，ASCII-8BIT/BINARY 编码）。

默认情况下，默认的外部编码继承自用户的地区（locale）设定，它经常是多字节编码，因此，如果想从文件读取二进制的数据，则必须显式指明你需要未编码的字节，否则会得到默认外部编码方式的字符。要做到这点，你可以用“r:binary”方式打开文件，也可以在打开文件后用 Encoding::BINARY 参数调用 set_encoding 方法：

```
File.open("data", "r:binary")  # Open a file for reading binary data
```

在 Windows 平台下，应该用“rb:binary”方式打开二进制文件，或者对流调用 binmode 方法，这样可以防止 Windows 对换行符进行转换。这只在 Windows 平台下是必须的。

并非每个读取流的方法都使用编码，一些低级的读方法会指定读取字节的个数，这些方法在本质上使用未编码的字节，不过那些不指定读取长度的方法仍然使用编码方式。

9.7.3 读取流
Reading from a Stream

IO 类定义了大量读取流的方法，当然，只有在流可读时这些方法才能工作。你可以从 STDIN、ARGF 和 DATA 中进行读取，但是不能从 STDOUT 或 STDERR 中读取。除非显式指定为只写方式，文件和 StringIO 对象将默认以读方式打开。

9.7.3.1 读取文本行

IO 定义了多种方法用于从流中读取文本行：

```
lines = ARGF.readlines              # Read all input, return an array of lines
line = DATA.readline                # Read one line from stream
print l while l = DATA.gets         # Read until gets returns nil, at EOF
DATA.each {|line| print line }      # Iterate lines from stream until EOF
DATA.each_line                      # An alias for each
DATA.lines                          # An enumerator for each_line: Ruby 1.9
```

对于这些读取流的方法，有一些重要的注意事项。首先，readline 和 gets 方法只在处理 EOF（文件结束——end-of-file：当没有内容可以从流中读入时）的方式上有所区别。在流处于 EOF 状态时，gets 方法返回 nil，而 readline 则抛出一个 EOFError 异常。如果不知道流有多少行，用 gets；如果知道那里应该还有一行（如果它不存在则表明一个错误），用 readline。可以用 eof?方法来检测一个流是否处在 EOF 状态。

其次，gets 和 readline 方法隐式地把全局变量$_赋值为返回的文本行。很多全局方法（比如 print）在没有显式参数的情况下，使用$_作为参数。由此，上面的循环语句可以用下面这样更简洁的方式实现：

```
print while DATA.gets
```

$_最好只是用于简短的脚本中，对于较长的脚本，还是明确使用变量来存储读入的文本行。

再次，这些方法一般用于文本流（而非二进制流），并且一“行”文本包含一个字节序列，并以一个默认的换行符（在绝大多数平台上都是 newline）结尾。这些方法返回的文本行是包含换行符的（即使一个文件的最后一行可能没有换行符），你可以使用 String.chomp!去掉换行符。全局变量$/用于存放换行符，如果修改了这个变量，所有读取文本行方法的行为都将被修改。你也可以在这些方法（包括 each 迭代器）被调用时指明一个替代的换行

符。例如，在读取逗号分隔的文件时，或者在读取某些带有“记录分隔符”的二进制文件时，你可能会使用这种方式。还有两种特殊含义的换行符。如果你用 nil 做换行符，那么这些文本行读取方法读取整个流的内容，将它们作为一行返回。如果用空字符串“”作为换行符，那么这些方法会一次读取一段，以空行作为分隔符。

在 Ruby 1.9 中，get 和 readline 方法可以接受一个可选的整数作为第一个参数，或者把该整数放在分隔符字符串后作为第二个参数。如果指定了一个整数，它表明读取流的最大字节数。这个 limit 参数可以防止误读入一个过长的行。这些方法是前面所述规则的一个特例，尽管有一个参数限定了字节数，它们返回的字符串依然是有编码的。

最后，文本行读取方法 gets、readline 和 each 迭代器（及别名方法 each_line）可追踪它们读入的行号。你可以用 lineno 方法获得最近所读取行号，也可以用 lineno=方法来设置它。注意 lineno 统计的不是文件中换行符的个数，而是读取行方法调用的次数，如果使用不同的行分隔符，所返回的值可能会有所不同：

```
DATA.lineno = 0  # Start from line 0, even though data is at end of file
DATA.readline    # Read one line of data
DATA.lineno      # => 1
$.               # => 1: magic global variable, implicitly set
```

9.7.3.2 读取整个文件

IO 定义了三个类方法，可以在不打开流的情况下对文件进行读取。IO.read 方法读取整个文件（或部分文件）到一个字符串中；IO.readlines 读取文件到一个文本行数组中；IO.foreach 方法迭代文件的每一行。所有这些方法都无须初始化一个 IO 对象：

```
data = IO.read("data")           # Read and return the entire file
data = IO.read("data", 4, 2)     # Read 4 bytes starting at byte 2
data = IO.read("data", nil, 6)   # Read from byte 6 to end-of-file

# Read lines into an array
words = IO.readlines("/usr/share/dict/words")

# Read lines one at a time and initialize a hash
words = {}
IO.foreach("/usr/share/dict/words") {|w| words[w] = true}
```

尽管这些方法被定义在 IO 类中，它们操作的却是文件，而且它们也经常以 File 类方法的方式进行调用：File.read、File.readlines 和 File.foreach。

IO 类还定义了一个名为 read 的实例方法，它与同名的类方法功能相似，如果不带参数，它读取整个流的文本到一个字符串中：

```
# An alternative to text = File.read("data.txt")
f = File.open("data.txt")     # Open a file
text = f.read                 # Read its contents as text
f.close                       # Close the file
```

你也可以给 IO.read 方法指定参数，来读取特定数目的字节。这种用法将在下一节中介绍。

9.7.3.3 读取字节和字符

IO 类还定义了用于一次读取一个或多个字节（或字符）的方法，不过这些方法在 Ruby 1.8 和 Ruby 1.9 中有较大的变化，这是因为 Ruby 的字符定义发生了变化。

在 Ruby 1.8 中，字节和字符是一回事，getc 和 readchar 方法都读取一个字节并把它作为 Fixnum 对象返回。与 gets 类似，当 EOF 时，getc 返回 nil；而 readchar 与 readline 类似，在 EOF 时会抛出一个 EOFError 异常。

而在 Ruby 1.9 中，getc 和 readchar 方法被进行了修改，它们不再返回 Fixnum 对象，而返回长度为 1 的字符串。当从多字节编码的流读入数据时，如果必要，这些方法会返回多个字节（直到读完一个完整字符）。如果希望在 Ruby 1.9 中一次读取一个字节，你可以使用新的 getbyte 和 readbyte 方法。getbyte 类似 getc 和 gets，在 EOF 时返回 nil；而 readbyte 与 readchar 和 readline 类似，会抛出一个 EOFError 异常。

那些一次从流中读取一个字符的程序（比如解析器）有时需要把一个字符推回到流的缓冲区中，让它在下一次读取时返回。这可以通过 ungetc 实现。在 Ruby 1.8 中，这个方法需要一个 Fixnum 参数；而在 Ruby 1.9 中，则需要一个单字符的字符串。这个被推回的字符在下次调用 getc 或 readchar 时被返回：

```
f = File.open("data", "r:binary")     # Open data file for binary reads
c = f.getc                            # Read the first byte as an integer
f.ungetc(c)                           # Push that byte back
c = f.readchar                        # Read it back again
```

另外一种从流中读取字节的方式是使用 each_byte 迭代器。这个方法把流中的每个字节传给相关联的代码块：

```
f.each_byte {|b| ... }        # Iterate through remaining bytes
f.bytes                       # An enumerator for each_byte: Ruby 1.9
```

如果一次想读取多个字节，可以有五种方式供选择。每种方式都有些许区别：

readbytes(n)

精确读取 n 个字节，并以字符串方式返回。如果需要，在读取完 n 个字节前，它会阻塞进程。如果读取 n 个字节前就出现 EOF，那么会抛出 EOFError 异常。

`readpartial(n, buffer=nil)`

用于从流中读取 1 到 n 个字节，并返回一个新的二进制字符串，或者，如果第二个参数是 String 对象，则把得到的字符串放入这个对象中（原有内容被覆盖）。如果有一个或多个字节可供读取，该方法立刻返回这些字节（最大不超过 n 个）。只有在没有字节可读时该方法才会阻塞，如果在调用时发生 EOF，该方法会抛出 EOFError 异常。

`read(n=nil, buffer=nil)`

读取 n 个字节（如果遇到 EOF，可能比 n 个少），在 n 个字节被读入前，有可能阻塞。它返回一个新的二进制字符串，如果第二个参数是 String 对象，则把得到的字符串放入这个对象中（原有内容被覆盖），并返回这个字符串。如果流处于 EOF 状态且指定了 n，返回 nil；如果流处在 EOF 状态且省略 n 或设为 nil，则返回一个空字符串 ""。

如果省略 n 或设为 nil，该方法读取该流所有未读取的内容，并以一个编码的字符串返回，而不是返回一个未编码的字节字符串。

`read_nonblock(n, buffer=nil)`

读取当前可供读取的字节（最多 n 个），以字符串方式返回，如果 buffer 指定了一个字符串，返回值被存放到这个字符串中。该方法不阻塞，如果没有数据可供读取（比如，当流为网络套接字或 STDIN 时可能发生），该方法会抛出一个 SystemCallError 异常。如果在调用时流处在 EOF 状态，则抛出 EOFError 异常。

该方法是在 Ruby 1.9 中新加入的。（Ruby 1.9 还定义了一些其他非阻塞的 IO 方法，不过它们都是一些低级方法，在这里不做介绍。）

`sysread(n)`

这个方法与 readbytes 相似，不过它是一个低级方法，不使用缓冲区。不要把该方法与任何其他读取行或读取字节的方法混用，它们是不兼容的。

下面是读取二进制文件的一些例子：

```
f = File.open("data.bin", "rb:binary")  # No newline conversion, no encoding
magic = f.readbytes(4)          # First four bytes identify filetype
exit unless magic == "INTS"     # Magic number spells "INTS" (ASCII)
bytes = f.read                  # Read the rest of the file
                                # Encoding is binary, so this is a byte string
data = bytes.unpack("i*")       # Convert bytes to an array of integers
```

9.7.4 写入流

Writing to a Stream

写入流的方法与读取方法相对应。与那些以非“r”或“rb”方式打开的文件一样，STDOUT 和 STDERR 流也是可写的。

IO 类定义的 putc 方法用于向流中写入一个字节或字符。这个方法要么接受一个字节值，要么接受一个单字符字符串，因此在 Ruby 1.8 和 Ruby 1.9 中，它的行为没有变化：

```
o = STDOUT
# Single-character output
o.putc(65)            # Write single byte 65 (capital A)
o.putc("B")           # Write single byte 66 (capital B)
o.putc("CD")          # Write just the first byte of the string
```

IO 类定义了一些其他方法用于写入任意字符串。这些方法在参数数目和是否加入换行符方面有所区别。前面已经说过，在 Ruby 1.9 中，如果指定了编码方式，输出文本将以流的外部编码方式进行编码：

```
o = STDOUT
# String output
o << x                # Output x.to_s
o << x << y           # May be chained: output x.to_s + y.to_s
o.print               # Output $_ + $\
o.print s             # Output s.to_s + $\
o.print s,t           # Output s.to_s + t.to_s + $\
o.printf fmt,*args    # Outputs fmt%[args]
o.puts                # Output newline
o.puts x              # Output x.to_s.chomp plus newline
o.puts x,y            # Output x.to_s.chomp, newline, y.to_s.chomp, newline
o.puts [x,y]          # Same as above
o.write s             # Output s.to_s, returns s.to_s.length
o.syswrite s          # Low-level version of write
```

跟字符串和数组相似，输出流是可追加的，可以用<<操作符来写入值。puts 是最常用的输出方法，它把每个参数转换为一个字符串，然后将它们写入流中。如果写入的字符串不以换行符结束，它会增加一个。如果某个参数是一个数组，那么这个数组被递归展开，然后每个元素在一行上被输出，就好像它们直接作为 puts 的参数一样。print 方法会转换参数为字符串，然后输出这些字符串。如果全局的域分隔符（field separator）$,被置为非 nil 的值，那么它将作为每个输出字符串的分隔符。如果输出记录分隔符(output record separator) $/被置为非 nil 值，该分隔符将在所有参数输出后被打印。

printf 方法要求第一个参数是格式字符串，并把后面的参数用%操作符插入这个格式字符串中，然后把这个插入后的字符串输出，它不会加入换行符或记录分隔符。

write 方法与<<一样，只输出其单个参数，并返回写入的字节数。最后，syswrite 是一个低级的、无缓冲区的、无转码版本的 write，如果必须使用 syswrite 方法，你不能将它和其他写入方法混用。

9.7.5 随机访问方法
Random Access Methods

有些流（比如表示网络套接字的流，或者表示用户在命令行输入的流）是有序的，一旦进行了读取操作，就无法回头。另外一些流（比如从文件或字符串进行读写的流）则允许本节所要介绍的随机访问。如果试图在那些不支持随机访问的流上使用这些方法，将得到一个 SystemCallException 异常。

```
f = File.open("test.txt")
f.pos        # => 0: return the current position in bytes
f.pos = 10   # skip to position 10
f.tell       # => 10: a synonym for pos
f.rewind     # go back to position 0, reset lineno to 0, also
f.seek(10, IO::SEEK_SET)      # Skip to absolute position 10
f.seek(10, IO::SEEK_CUR)      # Skip 10 bytes from current position
f.seek(-10, IO::SEEK_END)     # Skip to 10 bytes from end
f.seek(0, IO::SEEK_END)       # Skip to very end of file
f.eof?                        # => true: we're at the end
```

如果在程序中使用了 sysread 或 syswrite，则不能使用上面介绍的 seek 方法，而要用 sysseek 方法。sysseek 与 seek 相似，不同之处在于它每次会返回文件的当前位置：

```
pos = f.sysseek(0, IO::SEEK_CUR)  # Get current position
f.sysseek(0, IO::SEEK_SET)        # Rewind stream
f.sysseek(pos, IO::SEEK_SET)      # Return to original position
```

9.7.6 关闭、清除和测试流
Closing, Flushing, and Testing Streams

在完成流的读写操作后，你必须调用 close 方法对之进行关闭。这将清空所有输入输出缓冲区，并释放所有占用的操作系统资源。一些打开流的方法可以关联一个代码块，这些方法把打开的流传给这个代码块，并在代码块结束时关闭打开的流。这样即使操作过程中发生异常，也可以保证流被正常关闭：

```
File.open("test.txt") do |f|
  # Use stream f here
  # Value of this block becomes return value of the open method
end # f is automatically closed for us here
```

另外一种方式是使用 ensure 子句：

```
begin
  f = File.open("test.txt")
  # use stream f here
ensure
  f.close if f
end
```

网络套接字的实现使用了 IO 对象，它在内部分别使用进行读写的流，你可以用 close_read

和 close_write 方法对它们进行关闭。你可以对一个打开的文件同时进行读写操作，却不能对这些 IO 对象使用 close_read 和 close_write 方法。

出于效率的考虑，Ruby 的输出方法（除了 syswrite）都会缓冲输出。输出缓冲区将在适当的时机被清空，比如在输出一个换行符时或从一个相应流中读取数据时。不过，有时需要明确清空输出缓冲区，强制立刻进行输出：

```
out.print 'wait>'     # Display a prompt
out.flush             # Manually flush output buffer to OS
sleep(1)              # Prompt appears before we go to sleep

out.sync = true       # Automatically flush buffer after every write
out.sync = false      # Don't automatically flush
out.sync              # Return current sync mode
out.fsync             # Flush output buffer and ask OS to flush its buffers
                      # Returns nil if unsupported on current platform
```

IO 定义了若干判定方法来检测一个流的状态：

```
f.eof?       # true if stream is at EOF
f.closed?    # true if stream has been closed
f.tty?       # true if stream is interactive
```

这里唯一需要解释的方法是 tty?方法，如果流连接的交互设备是终端窗口或键盘，这个方法（或不带问号的别名方法 isatty）返回 true。如果流是非交互模式的（比如文件、管道或套接字）则返回 false。tty?方法可以用于当 STDIN 被重定向到文件时，不再显示提示用户输入的消息。

9.8 网络
Networking

Ruby 的网络功能不在核心库中被实现，而由标准库实现，因此，后面的子小节并不试图列出每个可用的对象或方法。相反，我们将通过一些简单的例子来演示如何完成常用的网络任务。你可以用 *ri* 来获取完整的文档。

在最底层，网络功能是通过套接字实现的，它是一种 IO 对象。一旦打开了一个套接字，你就可以从另外一台计算机中读取数据，也可以向它写入数据，就像是对一个文件进行读写一样。套接字类的层次结构有点让人发晕，不过这些细节在下面的例子中并不重要。因特网的客户端可以使用 TCPSocket 类，而因特网服务器则可以使用 TCPServer 类（也是一个套接字）。所有的套接字类都在标准库中，因此如果要在 Ruby 程序中使用它们，你首先要加入如下代码：

```
require 'socket'
```

9.8.1 一个十分简单的客户端
A Very Simple Client

要实现一个因特网客户端应用，你须要使用 TCPSocket 类。可以通过 TCPSocket.open 类方法获得一个 TCPSocket 实例，也可以通过它的同义词方法 TCPSocket.new 来实现。它的第一个参数是连接的主机名，第二个参数是连接的端口号。(端口号是一个从 1 到 65535 的整数，可以用 Fixnum 或 String 对象表示。不同的因特网协议使用不同的端口号，比如 Web 服务的默认端口号是 80。也可以用因特网服务的名字来替代端口名作为参数，比如可以用“http”，不过这没有在文档中详细说明，而且也许与所运行的系统相关。)

一旦打开一个套接字，你就可以像任何 IO 对象一样对之进行读操作。当操作完成时，要记得像关闭文件一样关闭它。下面的代码演示了一个非常简单的客户端，它连接指定主机和端口，并从连接的套接字中读取所有可获得的数据，然后退出：

```
require 'socket'                    # Sockets are in standard library

host, port = ARGV                   # Host and port from command line

s = TCPSocket.open(host, port)      # Open a socket to host and port
while line = s.gets                 # Read lines from the socket
  puts line.chop                    # And print with platform line terminator
end
s.close                             # Close the socket when done
```

像 File.open 方法一样，TCPSocket.open 方法可以带一个代码块。用这种方式，可把打开的套接字传给该代码块，在代码块返回时自动关闭套接字，所以，可以用如下方式改写上面的代码：

```
require 'socket'
host, port = ARGV
TCPSocket.open(host, port) do |s| # Use block form of open
  while line = s.gets
    puts line.chop
  end
end                                 # Socket automatically closed
```

上面的代码可以用于访问老式的（现已不用）像 Unix 的“daytime”之类的服务。对于这样的服务，客户端无须发送请求，它只是简单地连接服务器，服务器就会返回响应数据。如果你在因特网上不能找到一个这样的服务器来测试这个客户端，也不要失望——下一节将教你如何编写一个同样简单的时间服务器。

9.8.2 一个非常简单的服务器
A Very Simple Server

要实现因特网服务器，你须要使用 TCPServer 类。在本质上，一个 TCPServer 对象是产生 TCPSocket 对象的工厂对象。用给定的端口号调用 TCPServer.open 方法可以创建一个

TCPServer 对象，接着，在返回的 TCPServer 对象上调用 accept 方法，这个方法会等待客户端连接该指定端口，在连接之后，它返回一个 TCPSocket 对象，表示与所连客户端的连接。

下面的代码演示了如何编写一个简单的时间服务器。它监听 2000 端口，当一个客户端连接这个端口时，把当前时间发送给这个客户端，接着关闭套接字以停止与客户端的连接：

```
require 'socket'                # Get sockets from stdlib

server = TCPServer.open(2000) # Socket to listen on port 2000
loop {                          # Infinite loop: servers run forever
  client = server.accept        # Wait for a client to connect
  client.puts(Time.now.ctime) # Send the time to the client
  client.close                  # Disconnect from the client
}
```

要测试上面的代码，你可以在后台运行它，也可以在另外一个终端窗口中运行它，接着，用如下命令行运行前面的客户端代码：

```
ruby client.rb localhost 2000
```

9.8.3 数据报
Datagrams

绝大多数因特网协议都是使用 TCPSocket 和 TCPServer 实现的，前面两节已经对它们做了介绍。不过，因特网协议还可以使用 UDP 的数据报来实现，它开销更低，用 UDPSocket 类实现。使用 UDP，你可以向别的计算机发送单个数据包，可以省去建立永久连接的开销。下面的代码会演示如下功能：客户端发送一个包含字符串文本的数据报给指定主机和端口，服务器（应该运行在指定的主机和端口上）接收到这个文本，把它转换为大写形式（这的确不是什么像样的服务，我知道），然后用另一个数据报把转换后的文本返回。

首先是客户端代码。注意尽管 UDPSocket 也是 IO 对象，数据报还是与其他 IO 流有很大区别，因此，应该避免使用一般的 IO 方法，而使用 UDPSocket 的底层发送和接收方法。send 方法的第二个参数指定发送的标志，即使不须要设置任何标志，这个参数也是必须的。recvfrom 方法的参数指定我们要接收数据的最大个数。在本例中，我们限定客户端和服务器最多传送 1KB：

```
require 'socket'                        # Standard library

host, port, request = ARGV              # Get args from command line

ds = UDPSocket.new                      # Create datagram socket
ds.connect(host, port)                  # Connect to the port on the host
ds.send(request, 0)                     # Send the request text
response,address = ds.recvfrom(1024)  # Wait for a response (1kb max)
puts response                           # Print the response
```

服务器代码与客户端代码一样，也使用 UDPSocket 类——对于基于数据报的服务器，没有一个特殊的 UDPServer 类。它不是使用 connect 来连接一个套接字，而是用 bind 方法表明它监听的端口。然后，服务器会与客户端一样调用 send 和 recvfrom 方法，不过与客户端调用的次序相反。在收到数据报后，它把收到的文本转换为大写形式后发送回客户端。需要特别注意的是 recvfrom 方法返回两个值：第一个值是接收的数据；第二个值是一个数组，包含数据来源的信息，我们从中提取出主机和端口信息，然后通过它们把响应数据发送回去：

```
require 'socket'                          # Standard library

port = ARGV[0]                            # The port to listen on

ds = UDPSocket.new                        # Create new socket
ds.bind(nil, port)                        # Make it listen on the port
loop do                                   # Loop forever
  request,address=ds.recvfrom(1024)       # Wait to receive something
  response = request.upcase               # Convert request text to uppercase
  clientaddr = address[3]                 # What ip address sent the request?
  clientname = address[2]                 # What is the host name?
  clientport = address[1]                 # What port was it sent from
  ds.send(response, 0,                    # Send the response back...
          clientaddr, clientport)         # ...where it came from
  # Log the client connection
  puts "Connection from: #{clientname} #{clientaddr} #{clientport}"
end
```

9.8.4 一个更复杂的客户端
A More Complex Client

下面的代码是一个更高级的因特网客户端，它与 *telnet* 的方式相似。首先连接指定的主机和端口，接着循环读入控制台的文本行，把它发送给服务器，然后接收并打印服务器的响应。这段代码演示了如何确定网络连接的本地和远端地址；使用异常处理；并使用了本章前面介绍的 read_nonblock 和 readpartial 方法。下面的代码做了很好的注释，可以很好地解释它的功能：

```
require 'socket'      # Sockets from standard library

host, port = ARGV     # Network host and port on command line

begin                 # Begin for exception handling
  # Give the user some feedback while connecting.
  STDOUT.print "Connecting..."     # Say what we're doing
  STDOUT.flush                     # Make it visible right away
  s = TCPSocket.open(host, port)   # Connect
  STDOUT.puts "done"               # And say we did it

  # Now display information about the connection.
```

```
local, peer = s.addr, s.peeraddr
STDOUT.print "Connected to #{peer[2]}:#{peer[1]}"
STDOUT.puts " using local port #{local[1]}"

# Wait just a bit, to see if the server sends any initial message.
begin
  sleep(0.5)                      # Wait half a second
  msg = s.read_nonblock(4096)     # Read whatever is ready
  STDOUT.puts msg.chop            # And display it
rescue SystemCallError
  # If nothing was ready to read, just ignore the exception.
end

# Now begin a loop of client/server interaction.
loop do
  STDOUT.print '> '       # Display prompt for local input
  STDOUT.flush            # Make sure the prompt is visible
  local = STDIN.gets      # Read line from the console
  break if !local         # Quit if no input from console

  s.puts(local)           # Send the line to the server
  s.flush                 # Force it out

  # Read the server's response and print out.
  # The server may send more than one line, so use readpartial
  # to read whatever it sends (as long as it all arrives in one chunk).
  response = s.readpartial(4096)     # Read server's response
  puts(response.chop)                # Display response to user
end
rescue            # If anything goes wrong
  puts $!         # Display the exception to the user
ensure            # And no matter what happens
  s.close if s    # Don't forget to close the socket
end
```

9.8.5 一个多路服务器

A Multiplexing Server

前面演示的简单时间服务器不维护与客户端的连接——它只是简单地给客户端传送时间信息，然后断开连接。许多高级一些的服务器则要维持一个连接，它们需要同时与多个客户端交互。一种实现方式是使用线程——每个客户端使用一个单独线程，在本章后面部分将介绍一个多线程服务器的例子；另一种实现多路服务器的方式则是使用 Kernel.select 方法。

如果一个服务器连接了多个客户端，则不能对某个客户端调用像 gets 那样的阻塞方法。如果它阻塞于等待某个客户端的输入，则不能接收其他客户端的输入，也不能接受新客户端的连接请求。select 方法则可以解决这个问题，它可以在一组 IO 对象上阻塞，当其中任意一个对象上有活动时就会返回。select 方法的返回值是一个 IO 对象数组的数组。这个数组的第一个元素是读入数据（或接受请求的连接）的流（本例中是套接字）数组。

了解了 select 方法的这些知识，就可以理解下面的服务器代码。它实现的服务并不重要——只是反转客户端输入的每一行并发送回客户端，值得关注的部分在于它处理多个连接的机制。注意用 select 方法既可监控 TCPServer 对象，也可监控每一个 TCPObject 对象。还要注意的是，服务器既处理了客户端发起的断开连接请求，也处理了非正常的客户端断开事件：

```
# This server reads a line of input from a client, reverses
# the line and sends it back. If the client sends the string "quit"
# it disconnects. It uses Kernel.select to handle multiple sessions.

require 'socket'

server = TCPServer.open(2000) # Listen on port 2000
sockets = [server]            # An array of sockets we'll monitor
log = STDOUT                  # Send log messages to standard out
while true                    # Servers loop forever
  ready = select(sockets)     # Wait for a socket to be ready
  readable = ready[0]         # These sockets are readable

  readable.each do |socket|       # Loop through readable sockets
    if socket == server           # If the server socket is ready
      client = server.accept      # Accept a new client
      sockets << client           # Add it to the set of sockets
      # Tell the client what and where it has connected.
      client.puts "Reversal service v0.01 running on #{Socket.gethostname}"
      # And log the fact that the client connected
      log.puts "Accepted connection from #{client.peeraddr[2]}"
    else                          # Otherwise, a client is ready
      input = socket.gets         # Read input from the client

      # If no input, the client has disconnected
      if !input
        log.puts "Client on #{socket.peeraddr[2]} disconnected."
        sockets.delete(socket)    # Stop monitoring this socket
        socket.close              # Close it
        next                      # And go on to the next
      end

      input.chop!                 # Trim client's input
      if (input == "quit")        # If the client asks to quit
        socket.puts("Bye!");      # Say goodbye
        log.puts "Closing connection to #{socket.peeraddr[2]}"
        sockets.delete(socket)    # Stop monitoring the socket
        socket.close              # Terminate the session
      else                        # Otherwise, client is not quitting
        socket.puts(input.reverse) # So reverse input and send it back
      end
    end
  end
end
```

9.8.6 抓取 Web 页面
Fetching Web Pages

通过 socket 库，你可以实现任意因特网协议。比如，下面的代码演示了如何抓取一个 web 页面的内容：

```
require 'socket'               # We need sockets

host = 'www.example.com'       # The web server
port = 80                      # Default HTTP port
path = "/index.html"           # The file we want

# This is the HTTP request we send to fetch a file
request = "GET #{path} HTTP/1.0\r\n\r\n"

socket = TCPSocket.open(host,port)    # Connect to server
socket.print(request)                 # Send request
response = socket.read                # Read complete response
# Split response at first blank line into headers and body
headers,body = response.split("\r\n\r\n", 2)
print body                            # And display it
```

HTTP 是一个复杂的协议，上面的代码仅仅处理了简单的情形。你可以用内置的像 Net::HTTP 这样的库来处理 HTTP 协议。下面是上面代码的另一种实现方式：

```
require 'net/http'             # The library we need
host = 'www.example.com'       # The web server
path = '/index.html'           # The file we want

http = Net::HTTP.new(host)          # Create a connection
headers, body = http.get(path)      # Request the file
if headers.code == "200"            # Check the status code
                                    # NOTE: code is not a number!
  print body                        # Print body if we got it
else                                # Otherwise
  puts "#{headers.code} #{headers.message}" # Display error message
end
```

类似的库也存在于 FTP、SMTP、POP 和 IMAP 协议上，对它们的详细介绍超出了本书的范围。

最后，本章前面讲过的 open-uri 库可以使获取页面内容的工作更加简单：

```
require 'open-uri'
open("http://www.example.com/index.html") {|f|
  puts f.read
}
```

9.9 线程和并发
Threads and Concurrency

传统的程序只有一个执行线程，程序的所有语句和指令顺序执行，直到程序结束。**多线**

程程序则有多个执行线程。在每个线程内部，顺序执行语句，但是多个线程之间则可能并发执行——比如在多核的 CPU 上。通常多个线程并非真的并发执行（比如在单核单 CPU 的机器上），但是通过对线程执行的中断，可以模拟并发执行。

像图像处理这样的程序有大量的计算，我们称之为**计算绑定**（*compute-bound*）的程序，只有真正存在多个可并发执行的 CPU，它们才可能从多线程机制中获益。不过，大多数程序并非计算绑定的，很多程序（比如 Web 浏览器）花费大量时间等待网络通信和文件 I/O，这种程序被称为 **IO 绑定**（*IO-bound*）程序。IO 绑定的程序即使在只有一个 CPU 时也有用处，Web 浏览器可以用一个线程来渲染一幅图像，而同时用另外一个线程等待从网络下载另一幅图像。

使用 `Thread` 类，Ruby 程序员可以轻松编写多线程程序。要启动一个线程，只要用一个代码块调用 `Thread.new` 方法即可。新创建的线程用来执行代码块中的代码，而原始线程在调用 `Thread.new` 方法后直接返回，接着执行下一条语句：

```
# Thread #1 is running here
Thread.new {
  # Thread #2 runs this code
}
# Thread #1 runs this code
```

对于线程的介绍首先从 Ruby 线程模型和 API 开始，这些小节将解释诸如线程生命周期、线程调度和线程状态这些知识。介绍完了这些预备性的知识后，本书将给出一些示例代码并介绍一些像线程同步这样的高级课题。

最后，值得注意的一点是，通过用 Ruby 解释器运行外部进程或创建当前进程的新拷贝，Ruby 程序还可以在操作系统层次上获得并发性。不过，这样做是依赖于操作系统的，这将在第 10 章做简单介绍。要获得进一步的信息，你可以用 *ri* 命令查看 `Kernel.system`、`Kernel.exec`、`Kernel.fork`、`IO.popen` 方法和 `Process` 模块。

线程和平台依赖性

不同操作系统实现线程的方式不同，并且操作系统之上不同 Ruby 实现的 Ruby 线程也不同。比如，Ruby 1.8 的标准 C 实现只使用了一个本地线程，所有的 Ruby 线程都在这个本地线程上运行，这意味着 Ruby 1.8 的线程是十分轻量级的，它们从不会并发运行，即使在多核 CPU 上。

Ruby 1.9 则不同：它对每个 Ruby 线程都分配一个本地线程。不过由于这个实现使用的一

些 C 代码库不是线程安全的，Ruby 1.9 采取了保守的方式，它使用的本地线程从不会同时执行。（如果那些 C 代码可以变成线程安全的，Ruby 1.9 版本的后续发布可能会放宽这个限制。）

JRuby 是 Ruby 的 Java 实现版本，它把每个 Ruby 线程映射为一个 Java 线程，不过这样也使线程的实现和行为依赖于 Java 虚拟机的实现。现在的 Java 实现一般把 Java 线程映射为本地线程，并且可以在多核 CPU 上实现真正的并发运行。

9.9.1 线程生命周期

Thread Lifecycle

如前所述，新线程可以通过 `Thread.new` 方法创建，也可以使用同义词方法 `Thread.start` 和 `Thread.fork`。在创建线程后无须启动它，它将在 CPU 资源可用时自动启动。调用 `Thead.new` 会得到一个 `Thead` 对象，`Thead` 类定义了很多方法，用于查询和操作运行的线程。

线程将运行 `Thead.new` 关联代码块中的代码，在运行完后停止。代码块中最后一条语句的值成为线程的值，它可以通过 `Thead` 对象的 `value` 方法获得。如果一个线程已经运行完成，那么这个值被立刻传给 `thread` 的 `value` 方法；否则，`value` 方法被阻塞，直到线程完成。

类方法 `Thread.current` 返回代表当前线程的 `Thread` 对象，这样线程就可以对自己进行操作。类方法 `Thread.main` 返回代表主线程的 `Thread` 对象——这是 Ruby 程序启动时的初始线程。

9.9.1.1 主线程

主线程有其特殊性：Ruby 解释器在主线程完成时会停止运行，即使在主线程创建的线程仍在运行时也是如此，因此，必须保证主线程在其他线程仍在运行时不会结束。一种方法是采用无限循环方式实现主线程，另一种方式是明确等待其他线程完成操作。前面已经提到可以对线程调用 `value` 方法来等待它结束，如果不关心线程的值，你也可以使用 join 方法等待其结束。

下面的方法将等待除主线程和当前线程（两者可能相同）外的所有线程结束：

```
# Wait for all threads (other than the current thread and
# main thread) to stop running.
# Assumes that no new threads are started while waiting.
def join_all
  main = Thread.main          # The main thread
  current = Thread.current    # The current thread
```

```
  all = Thread.list          # All threads still running
  # Now call join on each thread
  all.each {|t| t.join unless t == current or t == main }
end
```


9.9.1.2 线程和未处理的异常

如果在主线程中抛出异常并且没有被处理，Ruby 解释器将打印一条信息并退出。如果其他线程中有未处理的异常，那么只有这个线程被停止，默认情况下，解释器不会打印消息或退出。如果线程 t 因为一个未处理的异常而中止，而另外一个线程 s 调用了 t.join 或 t.vale 方法，那么 t 中产生的异常在线程 s 中被抛出。

如果希望所有线程的未处理异常都使解释器退出，你可以使用类方法 Thread.abort_on_exception=：

```
Thread.abort_on_exception = true
```

如果希望某个特定线程的未处理异常使得解释器退出，可以使用同名的实例方法：

```
t = Thread.new { ... }
t.abort_on_exception = true
```

9.9.2 线程和变量

Threads and Variables

线程的关键特性之一在于它们可以共享变量，因为线程在代码块中被定义，它们可以访问所有该代码块范围内的变量（局部变量、实例变量和全局变量等）：

```
x = 0

t1 = Thread.new do
  # This thread can query and set the variable x
end

t2 = Thread.new do
  # This thread and also query and set x
  # And it can query and set t1 and t2 as well.
end
```

当两个或多个线程同时访问一个变量时，你必须小心处理。在下面介绍线程同步时，我们将对此进行更多介绍。

9.9.2.1 线程私有变量

在线程代码块中定义的变量是线程私有的，对其他线程是不可见的。这是 Ruby 变量可见范围规则的一个简单应用。

我们经常希望线程对一个变量有自己的拷贝，这样对变量的修改不会影响线程的行为。考虑如下代码，它试图创建三个线程，分别打印 1，2 和 3：

```
n = 1
while n <= 3
  Thread.new { puts n }
  n += 1
end
```

在某些环境的某些实现下，这些代码可能会如愿打印 1，2 和 3，但是在其他一些环境或实现中，则可能未必如此。例如，这段代码完全有可能（如果这些创建的线程不是被立刻执行）打印出 4，4 和 4。每个线程读取变量 n 的一个共享拷贝，而这个变量的值在循环执行的过程中被修改。线程打印的值依赖于线程和其父线程运行的时机。

为了解决这个问题，我们把 n 的当前值传给 Thread.new 方法，并且把该值赋给一个代码块参数。代码块参数是代码块私有的（不过请参见第 5.4.3 节，查看注意事项），这个私有值不会被其他线程共享：

```
n = 1
while n <= 3
  # Get a private copy of the current value of n in x
  Thread.new(n) {|x| puts x }
  n += 1
end
```

另一个解决这个问题的方法是使用迭代器取代 while 循环。这时，n 是其所在代码块所私有的，并且在代码块执行过程中不会被修改：

```
1.upto(3) {|n| Thread.new { puts n }}
```

9.9.2.2 线程局部变量

一些特殊的 Ruby 全局变量是**线程局部**的，在不同的线程中它们可能有不同的值，比如，$SAFE（参见第 10.5 节）和$~就是这样。这意味着如果两个线程同时进行正则表达式匹配，它们看到的$~值是不同的，并且在一个线程上执行的匹配不会影响另一个线程的匹配操作。

Thread 类提供了类似哈希表的行为。它定义了[]和[]=这两个实例方法，它们可以把符号关联到任意值上。（如果使用的是字符串，它们会被转换为一个符号，与真正的哈希表不同，Thread 类只能用符号作为主键。）这些符号所关联值的行为跟线程局部变量相似，它们不像代码块局部变量那样是私有的，任何变量都可以查询其他线程的相应的值；不过它们也不是共享变量，每个线程都有自己的拷贝。

例如，假设我们创建了一些线程从一个 Web 服务器下载文件，主线程可能用于监视下载的进度。要实现这点，每个线程应该有如下代码：

```
Thread.current[:progress] = bytes_received
```

这样，主线程就可以用如下代码判断下载的字节总量：

```
total = 0
download_threads.each {|t| total += t[:progress] }
```

除了[]和[]=方法，Thread 还定义了 key?方法来判断给定主键是否存在于一个线程中。keys 方法返回一个符号数组，表示线程中定义的所有主键。上面的代码改成如下方式会更好，这样即使线程还没有被启动而且也没有定义:progress 主键，它同样可以工作：

```
total = 0
download_threads.each {|t| total += t[:progress] if t.key?(:progress)}
```

9.9.3 线程调度

Thread Scheduling

通常，Ruby 解释器运行的线程数会大于现有可用的 CPU 个数。当真正的并行处理不能实现时，你可采用线程共享 CPU 的方式进行模拟。线程共享 CPU 的过程被称为线程调度。根据 Ruby 解释器的实现及其平台，调度可能由 Ruby 解释器实现，也可能被底层操作系统所处理。

9.9.3.1 线程优先级

影响线程调度的首要因素是**线程优先级**：高优先级的线程比低优先级的线程优先调度。更准确地说，一个线程只有在没有更高优先级的线程等待时才可能被 CPU 执行。

你可以用 Thread 对象的 priority=和 priority 方法设置和查询线程的优先级。注意在线程开始运行前无法设置优先级，不过，可以在第一个动作执行前调高或降低优先级。

新创建线程的优先级与创建它的线程相同。主线程将以优先级 0 启动。

与线程的很多特性一样，线程优先级也与 Ruby 实现及底层操作系统相关。比如在 Linux 下，无特权 (nonprivileged) 线程不能调高或降低自身优先级，因此 Linux 下的 Ruby 1.9 实现 (它使用本地线程)，会忽略线程优先级的设置请求。

9.9.3.2 抢占式线程调度和 Thread.pass 方法

当多个同优先级线程须要共享 CPU 时，线程调度器决定每个线程什么时候执行及执行多久。一些调度器是抢占式的，这意味着每个同级别的线程都可以被执行固定的时间；另一种调

度器是非抢占式的：一旦一个线程开始运行，除非它睡眠、IO 阻塞或有更高优先级的线程醒来，否则它会一直运行下去。

如果一个计算绑定（compute-bound）的线程须要运行很长时间（比如，它从不出现 IO 阻塞），那么在一个非阻塞的调度器上，它会让其他线程感到“饥饿”，其他线程无法获得运行的机会。要避免这一问题，耗时的计算绑定线程应该定期调用 `Thread.pass` 方法，让别的线程有机会获得 CPU。

9.9.4 线程状态
Thread States

一个 Ruby 线程可能有五种状态。两个最值得关注的状态是关于活跃线程的：一个活跃线程可能是**可运行的**（*runnable*），也可能是处在**休眠状态的**（*sleeping*）。一个可运行的线程要么正在运行，要么已经准备好在下次 CPU 资源可用时运行；一个休眠线程要么是因为等待 I/O 操作而休眠（参见 `Kernel.sleep`），要么被自身中止（参见下面的 `Thread.stop` 方法）。线程一般在可运行状态和休眠状态中来回转换。

当一个线程不再活跃时，它可能有两种状态。它有可能是正常结束（terminated normally），也有可能是异常中止（terminated abnormally with an exception）。

最后，还有一个过渡状态。一个被杀死的线程（参见下面的 `Thread.kill` 方法）在还没有被终止前被称为处于**正在中止**（*aborting*）状态。

9.9.4.1 查询线程状态

`Thread` 类定义了若干实例方法用于检测线程的状态。当线程处于可运行状态或休眠状态时，`alive?` 方法返回 `true`；当一个线程不处于可运行状态时，`stop?` 方法返回 `true`。最后，`status` 方法返回线程的状态。下表显示了可能返回的五种状态值：

线程状态	返回值
可运行（Runnable）	`"run"`
休眠（Sleeping）	`"sleep"`
正在中止（Aborting）	`"aborting"`
正常退出（Terminated normally）	`false`
异常中止（Terminated with exception）	`nil`

9.9.4.2 状态转换：暂停、唤醒和杀死线程

在创建时，线程处在**可运行**状态，可能被立刻执行。线程可以调用 `Thread.stop` 方法暂停（pause）执行——这将使线程进入**休眠**状态。这是一个类方法，只对当前线程生效——没

有对应的实例方法，因此不能强制其他线程暂停。调用 Thead.stop 的效果与调用无参数的 Kernel.sleep 相同：这个线程将被永远停止（或直到被唤醒，如下所述）。

如果调用带参数的 Kernel.sleep 方法，那么线程暂时进入**休眠**状态。这时，线程会在给定的秒数（大约）后重新进入**可运行**状态。调用阻塞的 IO 方法也会使线程进入休眠状态，直到 IO 操作完成——事实上，就是 IO 操作的延时性使线程即使在单 CPU 系统上也是有用的。

用 Thread.stop 或 Kernel.sleep 方法暂停的线程可以被实例方法 wakeup 和 run 重新激活（即使休眠时间还没到期），这两个方法都能让线程从**休眠**状态变为**可运行**状态。run 方法还会对调度器产生请求，让当前线程放弃对 CPU 的占用，这样新唤醒的线程就可能被立刻执行。wakeup 方法只是唤醒指定线程，而不会让当前线程放弃 CPU。

在线程所在代码块正常结束或异常中止时，线程从**可运行**状态切换到一种**结束**状态。另外一种正常结束线程的方式是调用 Thread.exit 方法，在这种方式下，注意在线程结束前所有 ensure 语句将被执行。

线程也可以被另外一个线程强行中止，这可以通过在被中止线程上调用实例方法 kill 来实现。terminate 和 exit 是 kill 的同义词方法。这些方法把杀死的线程置为**正常退出**状态，被杀死的线程在真正结束运行前运行所有 ensure 语句。而 kill!方法（及同义词方法 terminate!和 exit!）在杀死线程前不允许执行任何 ensure 语句。

目前所介绍的中止线程的方法都把线程置为**正常退出**状态。你可以用实例方法 raise 从另外一个线程中抛出一个异常。如果这个线程不能处理引入的这个异常，它将进入**异常中止**状态。像正常的异常处理一样，ensure 语句被确保执行。

除非能设法知道一个线程不是正在修改系统的共享状态，否则杀死一个线程是非常危险的。用带有!的方法来中止线程则更加危险，因为这可能会遗留未关闭的文件、套接字或其他资源。如果一个线程必须接受退出命令，最好采用周期性检查标志变量状态的方式，当标志变量被置为特定值时，它们可以安全而优雅的退出。

9.9.5 列举线程和线程组
Listing Threads and Thread Groups

Thread.list 方法返回所有活跃线程（运行或休眠）的 Thread 对象数组。当一个线程结束时，它从这个数组中被删除。

除了主线程，每个线程都是由其他线程创建的，这样，线程就可以组织成一个树型结构，每个线程都有一个父线程及一组子线程。不过，Thread 类并不维护这一信息：人们通常认为线程是自治的，而不是由创建它们的线程所控制的。

如果希望给一组线程加入某种次序，你可以创建一个 ThreadGroup 对象，并向其中加入线程：

```
group = ThreadGroup.new
3.times {|n| group.add(Thread.new { do_task(n) })}
```

新创建的线程最初被放入父线程所在的线程组中。你可以用实例方法 group 来查询一个线程所在的 ThreadGroup。用 ThreadGroup 的 list 方法可以获得一个线程组中线程的数组。像 Thread.list 类方法一样，实例方法 ThreadGroup.list 也只是返回那些没有中止的线程。你可以用 list 方法定义那些操作线程组中所有线程的方法，比如，可以定义一个降低组中所有线程优先级的方法。

ThreadGroup 比一般线程数组更有用的地方在于它的 enclose 方法。一旦一个线程组被封闭（enclosed），则既不能从中删除线程，也不能加入新的线程。组中的线程可能会创建新的线程，这些新线程成为线程组的成员。如果想在低于$SAFE（参见第 10.5 节）变量所定义的安全级别中运行 Ruby 代码，并且想跟踪该代码创建的所有线程，一个封闭的 ThreadGroup 会很有用。

9.9.6 线程示例

Threading Examples

在解释了线程模型和 API 之后，我们将看一些多线程编程的实例。

9.9.6.1 并发读文件

IO 绑定程序是最经常使用线程的地方，线程可以让程序在等待用户输入、文件系统和网络时仍然继续工作。比如，下面的代码定义了一个 conread（用于并发读）方法，它接受一个文件名数组，返回一个哈希表，用于对应文件名和其对应的文件内容。它使用多线程对这些文件进行并发读，并且故意使用了 open-uri 模块，这样就可以用 Kernel.open 方法像读取文件一样打开 HTTP 和 FTP 的 URL：

```
# Read files concurrently. Use with the "open-uri" module to fetch URLs.
# Pass an array of filenames. Returns a hash mapping filenames to content.
def conread(filenames)
  h = {}                                # Empty hash of results

  # Create one thread for each file
  filenames.each do |filename|          # For each named file
```

```
    h[filename] = Thread.new do          # Create a thread, map to filename
      open(filename) {|f| f.read }       # Open and read the file
    end                                  # Thread value is file contents
  end

  # Iterate through the hash, waiting for each thread to complete.
  # Replace the thread in the hash with its value (the file contents)
  h.each_pair do |filename, thread|
    begin
      h[filename] = thread.value         # Map filename to file contents
    rescue
      h[filename] = $!                   # Or to the exception raised
    end
  end
end
```

9.9.6.2 一个多线程服务器

另一个几乎可以算教科书的线程用例是同时与多个客户端通信的服务器。前面我们介绍过可以用 Kernel.select 方法实现这样的服务器，不过更简单的方式（尽管可能不那么容易扩展）是使用线程：

```
require 'socket'

# This method expects a socket connected to a client.
# It reads lines from the client, reverses them and sends them back.
# Multiple threads may run this method at the same time.
def handle_client(c)
  while true
    input = c.gets.chop          # Read a line of input from the client
    break if !input              # Exit if no more input
    break if input=="quit"       # or if the client asks to.
    c.puts(input.reverse)        # Otherwise, respond to client.
    c.flush                      # Force our output out
  end
  c.close                        # Close the client socket
end


server = TCPServer.open(2000) # Listen on port 2000

while true                       # Servers loop forever
  client = server.accept         # Wait for a client to connect
  Thread.start(client) do |c| # Start a new thread
    handle_client(c)             # And handle the client on that thread
  end
end
```

9.9.6.3 并发迭代器

虽然 IO 绑定的任务通常可使用线程，不过并非一定要如此。下面的代码为 Enumerable 模

块加入一个 conmap 方法（用于并发 map）。它与 map 的工作方式类似，不过对于输入数组的每个元素，它都使用一个单独线程来处理：

```
module Enumerable         # Open the Enumerable module
  def conmap(&block)      # Define a new method that expects a block
    threads = []          # Start with an empty array of threads
    self.each do |item|   # For each enumerable item
      # Invoke the block in a new thread, and remember the thread
      threads << Thread.new { block.call(item) }
    end
    # Now map the array of threads to their values
    threads.map {|t| t.value } # And return the array of values
  end
end
```

下面是一个类似的并发 each 迭代器：

```
module Enumerable
  def concurrently
    map {|item| Thread.new { yield item }}.each {|t| t.join }
  end
end
```

这段代码很简洁，不过不大易读，如果你能读懂它，那就表明你正走在掌握 Ruby 语法及迭代器的康庄大道上。

前面说过在 Ruby 1.9 中，不带代码块的标准迭代器会返回一个枚举器对象，这意味着对于一个 Hash 对象 h，使用前面定义的 concurrently 方法，可以写出如下代码：

```
h.each_pair.concurrently {|*pair| process(pair)}
```

9.9.7 线程互斥和死锁

如果两个线程共享某些数据，并且至少一个线程修改了这些数据，那就必须特别小心，保证不要让某个线程看到的数据处于不一致的状态。这被称为**线程互斥**，有很多例子可表明其必要性。

首先，假设两个线程同时对文件进行处理，每个线程都对一个共享变量进行加 1 操作，从而对打开文件的总数进行跟踪。问题在于对变量的加 1 操作并非是**原子**操作，也就是说这个动作不能用一个步骤完成，Ruby 程序必须首先读出该变量的值，加上 1，然后把新的值存回这个变量中。假设目前这个计数器的值是 100，并设想两个线程对这个变量进行交叉访问。第一个线程读出该变量的值 100，不过在它进行加 1 操作前，调度器停止了这个线程的运行并让第二个线程开始运行，它读取出变量值 100，加 1，然后把 101 存储到这个计数器变量中。接着第二个线程开始读取新的文件，这会导致线程阻塞，从而使第一个线程被重新执行。第一个线程现在给 100 加 1 然后存储这个结果。这样，两个线程都对计数器进行了加 1 操作，不过计数器的值是 101 而非 102。

另外一个须要使用线程互斥的经典用例涉及电子银行应用。假设一个线程正在从一个存款户头向一个支票户头转款，另一个线程正在为客户生成月报表。如果没有适当的线程互斥，报表生成的线程有可能在钱款从存款户头被减掉之后，在支票户头被增加之前读取客户的账户信息。

对于像这样的问题，你需要用协作锁机制进行解决。每个希望访问共享数据的线程必须首先对数据进行**加锁**，锁用 Mutex（是互斥——mutual exclusion 的缩写）对象表示。要对一个 Mutex 对象加锁，你可以调用它的 lock 方法，在读取或修改完共享变量时，再调用它的 unlock 方法。如果对一个已经加锁的 Mutex 对象调用 lock 方法，在调用者成功获取一个锁之前，该方法会一直阻塞。如果每个线程都能正确地对共享数据进行加锁和解锁操作，那么不会有线程看到数据处于不一致的状态，前面所述的那些问题就不会再出现了。

在 Ruby 1.9 中，Mutex 类是内核库的一部分；而在 Ruby 1.8 中，它则是标准库的一部分。通常我们并不显式调用 lock 和 unlock 方法，而是使用关联代码块的 synchronize 方法。synchronize 方法会锁住 Mutex，运行代码块中的代码，然后在一个 ensure 子句中对 Mutex 进行解锁，这样可以保证异常被恰当处理。下面是一个银行账户的简单模型，它使用 Mutex 对象来同步线程对共享账户数据的操作：

```
require 'thread'  # For Mutex class in Ruby 1.8

# A BankAccount has a name, a checking amount, and a savings amount.
class BankAccount
  def init(name, checking, savings)
    @name,@checking,@savings = name,checking,savings
    @lock = Mutex.new        # For thread safety
  end

  # Lock account and transfer money from savings to checking
  def transfer_from_savings(x)
    @lock.synchronize {
      @savings -= x
      @checking += x
    }
  end

  # Lock account and report current balances
  def report
    @lock.synchronize {
      "#@name\nChecking: #@checking\nSavings: #@savings"
    }
  end
end
```

9.9.7.1 死锁

在使用 Mutex 对象进行线程互斥操作时必须小心避免**死锁**。在所有线程都等待获得其他线程持有的资源时，就会发生死锁，因为所有线程都被阻塞，它们不能释放所持有的锁，导致其他对象无法获得这些锁。

经典的死锁场景涉及两个线程和两个 Mutex 对象。线程 1 锁住 Mutex 1，然后试图锁住 Mutex 2；同时，线程 2 锁住 Mutex2 并且试图锁住 Mutex 1。哪个线程都无法获得它们希望的锁，而且每个线程都不会释放另一个线程所需要的锁，因此两个线程将一直阻塞下去：

```
# Classic deadlock: two threads and two locks
require 'thread'

m,n = Mutex.new, Mutex.new

t = Thread.new {
  m.lock
  puts "Thread t locked Mutex m"
  sleep 1
  puts "Thread t waiting to lock Mutex n"
  n.lock
}

s = Thread.new {
  n.lock
  puts "Thread s locked Mutex n"
  sleep 1
  puts "Thread s waiting to lock Mutex m"
  m.lock
}

t.join
s.join
```

避免这种死锁的方法是一直按照相同顺序对资源进行加锁操作，如果第二个线程在加锁 n 之前锁住 m，死锁就不会产生了。

注意，即使在不使用 Mutex 对象时，也可能发生死锁。如果某个线程对调用了 Thread.stop 的线程调用 join 方法，除非有第三个线程可以唤醒这个中止的线程，否则，这两个线程会同时死锁。

须要了解的是，某些 Ruby 实现可检测这样的简单死锁，并抛出错误退出。不过这不受保证。

9.9.8 Queue 和 SizedQueue 类

Queue and SizedQueue

标准的线程库定义了专为并发编程设计的 Queue 和 SizedQueue 数据结构，它们实现了线程安全的先进先出队列，可以用于编写生产者/消费者模型的程序。使用这种模型，一个线

程生产某种对象值并用 enq（enqueue）或同义词方法 push 方法将之放入队列中，另外一个线程则“消费”这些对象值，可以根据需要用 deq（dequeue）方法把它们从队列中移除。（pop 和 shift 方法是 deq 的同义词方法。）

Queue 适合于并发编程的关键特性在于当队列处在空状态时，deq 方法会阻塞，直到生产者线程为队列增加一个对象值。Queue 和 SizedQueue 实现相同的 API，不过 SizedQueue 有最大长度限制。如果队列达到最大长度，那么增加对象值的方法会阻塞，直到消费者线程从队列中删除一个对象值。

像 Ruby 的其他集合类一样，你可以用 size 或 length 方法确定队列元素的个数，也可以用 empty? 方法判断队列是否为空。在调用 SizedQueue.new 方法时，你可以指定 SizedQueue 对象的最大长度，在创建了 SizedQueue 对象之后，也可以用 max= 方法来修改它的最大长度。

在本章前面部分，我们看到了如何为 Enumerable 模块增加一个 conmap（并发 map）的方法。现在我们要定义一个方法，把并发 map 和并发 inject 结合在一起。它为可枚举集合的每个元素创建一个线程，并把这个线程应用于一个映射的 Proc 对象。最后一个线程作为消费者，它从队列中删除对象，在它们可用时传给 injection 的 Proc 对象。

我们把这个并发的 injection 方法称为 conject，可以把它用于像并发计算数组中数值的平方和之类的场景，不过要记得一个线性算法的速度肯定比下面这个求平方和的例子快：

```
a = [-2,-1,0,1,2]
mapper = lambda {|x| x*x }                  # Compute squares
injector = lambda {|total,x| total+x }      # Compute sum
a.conject(0, mapper, injector)              # => 10
```

定义 conject 方法的代码如下——注意它是如何使用 Queue 对象及其 enq 和 deq 方法的：

```
module Enumerable
  # Concurrent inject: expects an initial value and two Procs
  def conject(initial, mapper, injector)
    # Use a Queue to pass values from mapping threads to injector thread
    q = Queue.new
    count = 0                   # How many items?
    each do |item|              # For each item
      Thread.new do             # Create a new thread
        q.enq(mapper[item])     # Map and enqueue mapped value
      end
      count += 1                # Count items
    end

    t = Thread.new do           # Create injector thread
```

```
      x = initial              # Start with specified initial value
      while(count > 0)         # Loop once for each item
        x = injector[x,q.deq]  # Dequeue value and inject
        count -= 1             # Count down
      end
      x                        # Thread value is injected value
    end

    t.value   # Wait for injector thread and return its value
  end
end
```

9.9.9 条件变量和队列

Condition Variables and Queues

对于 Queue 对象，有一个重要事项需要注意：deq 方法会阻塞。一般情况下，我们只认为 IO 方法（或者对一个线程调用 join 方法，或者对一个 Mutex 加锁）会阻塞。不过，在多线程编程中，有时需要让某个线程等待某个条件（不在这个线程控制范围之内）成立。对于 Queue 类来说，这个条件是队列不为空：如果队列为空，一个消费者线程必须等待，直到一个生产者线程调用 enq 方法让队列不为空。

使用 ConditionVariable，你能以最清晰的方式让一个线程保持等待，直到其他线程通知它可以再次执行为止。像 Queue 一样，ConditionVariable 也是标准线程库的一部分，你可以用 ConditionVariable.new 方法创建一个 ConditionVariable 对象。用 wait 方法可以让一个线程等待这个条件；用 signal 方法可以唤醒一个等待线程；用 broadcast 可以唤醒所有等待线程。使用条件变量有一个小技巧：为了让一切正常工作，等待线程必须给 wait 方法传递一个上锁的 Mutex 对象。在这个线程等待过程中，这个 Mutex 对象会暂时解锁，而在线程唤醒后重新上锁。

作为对线程的总结，我们给出一个有时对多线程编程有用的工具类，它被称为 Exchanger，允许两个线程交换任意值。假设我们有线程 t1、t2，以及一个 Exchanger 对象 e。t1 调用 e.exchange(1)，这个方法在 t2 调用 e.exchange(2)方法之前一直阻塞（当然，使用了一个 ConditionVariable）。第二个线程不会阻塞，它简单返回 1——这个值由 t1 传入。现在，由于第二个线程调用了 exchange 方法，t1 被重新唤醒，并且对 exchange 方法返回 2。

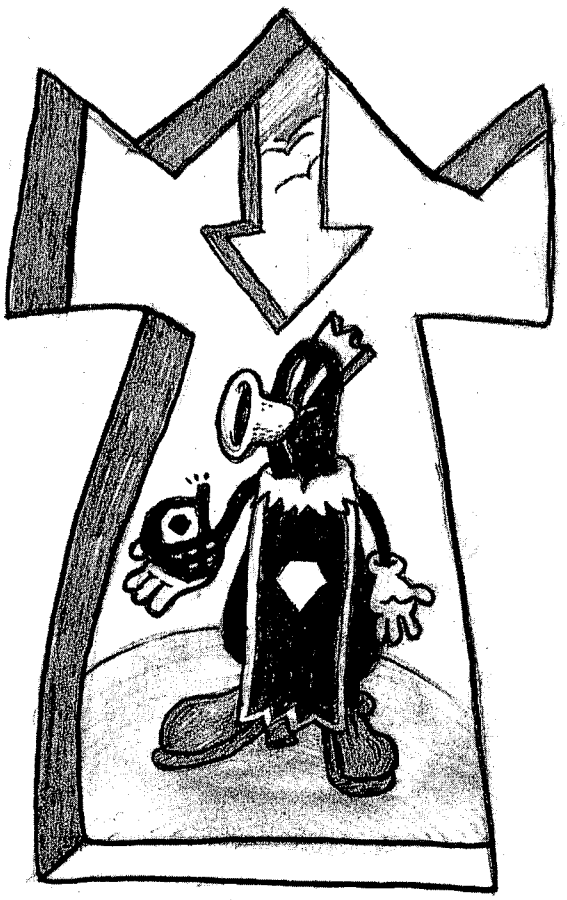
Exchanger 的实现多少有点复杂，但是它演示了 ConditionVariable 类的一个典型用法。代码中值得关注的一点是它使用了两个 Mutex 对象，一个用于同步 exchange 方法的访问，它被传给条件变量的 wait 方法；另一个 Mutex 对象用于判断调用线程是第一个还是第二个调用 exchange 的线程，它没有使用 Mutex 的 lock 方法，而使用了非阻塞的 try_lock 方法。如果@first.try_lock 返回真，那么调用的线程是第一个线程，否则是第二个线程：

```
require 'thread'

class Exchanger
  def initialize
    # These variables will hold the two values to be exchanged.
    @first_value = @second_value = nil
    # This Mutex protects access to the exchange method.
    @lock = Mutex.new
    # This Mutex allows us to determine whether we're the first or
    # second thread to call exchange.
    @first = Mutex.new
    # This ConditionVariable allows the first thread to wait for
    # the arrival of the second thread.
    @second = ConditionVariable.new
  end

  # Exchange this value for the value passed by the other thread.
  def exchange(value)
    @lock.synchronize do       # Only one thread can call this method at a time
      if @first.try_lock       # We are the first thread
        @first_value = value   # Store the first thread's argument
        # Now wait until the second thread arrives.
        # This temporarily unlocks the Mutex while we wait, so
        # that the second thread can call this method, too
        @second.wait(@lock)    # Wait for second thread
        @first.unlock          # Get ready for the next exchange
        @second_value          # Return the second thread's value
      else                     # Otherwise, we're the second thread
        @second_value = value  # Store the second value
        @second.signal         # Tell the first thread we're here
        @first_value           # Return the first thread's value
      end
    end
  end
end
```

第 10 章

Ruby 环境

The Ruby Environment



本章汇集了其他章节没有覆盖的内容，这里介绍的绝大多数特性都与Ruby运行的操作系统接口相关，因此，这里介绍的不少特性依赖于运行的操作系统。类似地，这里介绍的很多特性依赖于具体实现：不是每个Ruby解释器都用同样的方式实现该特性。本章涵盖的主题有如下几个。

- Ruby解释器的命令行参数和环境变量。
- 顶级运行环境：全局方法、变量和常量。
- 字符处理脚本的快捷方式：受Perl语言影响引入的全局方法、变量和解释器选项，这使得用Ruby编写精干而强大的文本处理程序成为可能。
- OS命令：运行Shell命令及调用底层操作系统的可执行文件，这些特性使得Ruby可以用作“胶水”语言。
- 安全性：如何通过Ruby的受污代码运行机制减少诸如SQL注入之类的风险，以及如何通过$SAFE运行级别使不可信的Ruby代码在“沙箱”中运行。

10.1 执行Ruby解释器
Invoking the Ruby Interpreter

基于C的标准Ruby实现可以通过如下命令行执行：

```
ruby [options] [--] program [arguments]
```

options 可以是零个或多个命令行参数，它们影响解释器的操作。稍后将介绍各种合法的参数。

program 是要运行的Ruby程序文件名。如果文件名以连字符开头，应使用--打头，这样Ruby解释器才不会把它当成一个命令行参数。如果使用单个连字符作为程序名，或者完全省略 *program* 和 *arguments*，解释器从标准输入读取程序文本。

最后，*arguments* 可以是任意数量的标识，这些标识会成为ARGV数组的元素。

接下来的子小节将介绍基于C的标准Ruby实现所支持的选项。注意，也可以把-W、-w、-v、-d、-I、-r和-K选项中的任意选项加入RUBYOPT环境变量中，这样每次运行Ruby解释器都会自动应用这些选项，就像在命令行中加入这些选项一样。

10.1.1 常用选项
Common Options

下面的选项可能是最常用的。绝大多数的 Ruby 实现都应该支持这些选项，或至少提供相似的替代选项：

`-w`

这个选项使 Ruby 在遇到废弃 (deprecated) 或问题代码时给出警告，同时设置`$VERBOSE`为`true`。许多 Ruby 程序员在日常编码中都使用这个选项，以确保代码的整洁性。

`-e` *`script`*

这个选项使 Ruby 运行 *`script`* 中的代码。如果指定的`-e`选项多于一个，它们关联的脚本会被看作独立的代码行。并且，如果指定一个或多个`-e`选项，那么解释器不会加载或运行任何命令行参数指定的程序 (*program*)。

为了支持简洁的单行代码脚本，用`-e`选项指定的 Ruby 代码可以使用正则表达式匹配的快捷方式，这将在本章后面部分解释。

`-I` *`path`*

这个选项把 *`path`* 中的目录加到全局`$LOAD_PATH`数组的起始处，`load`和`require`方法会在该数组包含的目录中搜索加载文件（不过不会影响命令行指定加载的 *program*）。

可以同时用多个`-I`选项，每个选项都可以指定一个或多个目录。如果一个`-I`选项指定了多个目录，这些目录在 Unix 或类 Unix 系统下应该用**冒号**(:)进行分隔，而在 Windows 系统下则用**分号**（;）进行分隔。

`-r` *`library`*

这个选项用于在运行程序前加载给定的 *`library`*。它的作用就像在程序的首部增加如下代码行：

```
require 'library'
```

在`-r`和加载的库名之间的空格是可选的，它经常被省略。

`-rubygems`

这个被经常使用的命令行参数其实并非一个真正的选项，它不过是`-r`选项的一个灵活运用。它从标准库中加载名为`ubygems`（没有`r`）的模块，而`ubygems`模块会直接加载真正的`rubygems`模块。在 Ruby 1.9 中，gem 无须通过`rubygems`模块加载，但在 Ruby 1.8 中，后者是必需的。

`--disable-gems`

这是一个 Ruby 1.9 的选项，它会阻止把额外的 gem 安装路径放到默认加载路径中。如果系统安装了很多 gem，而且正在运行的程序不使用这些 gem（或程序通过`gem`方法对所依赖的库进行显式管理），这个选项会减少程序启动的时间。

`-d, --debug`

这两个选项把全局变量$DEBUG和$VERBOSE设置为true，它们可以被程序或库代码使用，用来打印调试信息或别的操作。

`-h`

这个选项显示解释器的选项并退出。

10.1.2 警告和信息选项
Warnings and Information Options

下面的选项控制Ruby解释器显示的信息数量或类型：

`-W, -W2, --verbose`

这些都是-w选项的同义词，它们打开verbose警告并设置$VERBOSE为true。

`-W0`

这个选项禁止所有的警告。

`-v`

这个选项打印Ruby的版本号。如果没有指定程序，它不会从标准输入读取程序，而是直接退出；如果指定了程序，它运行该程序，就像给出了--verbose(或-w)选项一样。

`--version, --copyright, --help`

这些选项打印Ruby的版本号、版权信息或命令行帮助，然后退出。--help是-h的同义词。--version与-v不同之处在于它从不会执行一个给定程序。

10.1.3 编码选项
Encoding Options

下面的选项可以用来指定Ruby进程的外部编码方式，也可以为那些没有使用注释指定编码的文件指定一个默认的源代码编码方式。如果没有指定任何选项，默认的外部编码方式会从地区（locale）设置继承而来，而默认的源程序编码则为ASCII（参见第2.4节可获得更多关于源程序编码和默认外部编码的内容）：

`-K` *code*

在Ruby 1.8中，这个选项指定脚本的源程序编码方式，并且对全局变量$KCODE进行设置。在Ruby 1.9中，它设置Ruby进程的默认外部编码方式，并且设置默认源程序编码方式。

当code为a、A、n或N时，表示ASCII编码；为u或U，表示Unicode；为e或E，表示EUC-JP；而为s或S则表示SJIS。(EUC-JP和SJIS是常用的日语编码。)

`-E` *encoding*`, --encoding=` *encoding*

这两个选项与-K选项相似，不过它们使用编码名，不使用单字母的简写方式。

10.1.4 文本处理选项
Text Processing Options

下面的选项会改变 Ruby 默认的文本处理操作，或者对执行-e 选项指定的单行脚本有所帮助：

`-0 xxx`

这个选项是数字 0，而非字母 O。*xxx* 应该是不小于 0 的 3 位八进制数字。如果指定了这个选项，这个数字所代表的 ASCII 码成为输入数据记录间的分隔符，并且相应设置 $/变量值。这个分隔符为 gets 或类似方法定义“一行”文本。单独使用-0 会用编码 0 设置$/变量。-00 有特殊含义：它使 Ruby 进入一种“段落模式（paragraph mode）”，在这种模式下，两个连续的换行符用于分隔文本行。

`-a`

这个选项自动把输入的每行分割成字段，并把这些字段存储在$F 变量中。这个选项只在使用-n 或-p 循环属性时，并且在每次迭代开始处增加$F = $_.split 代码时才有效。参见-F 选项。

`-F fieldsep`

这个选项用 fieldsep 的值设定输入字段的分隔符变量$;，当使用不带参数的 split 方法时，这个选项会影响它的行为。参见-a 选项。

fieldsep 可以是单个字符，也可以是不带分隔斜杠的正则表达式。根据外壳（shell）环境的不同，在命令行的正则表达式中，有时可能需要用引号括起反斜杠，有时则需要把一个反斜杠替换为两个反斜杠。

`-i [ext]`

这个选项编辑命令行指定的文件，并取代原有文件。程序从这些文件中读取文本行，输出的文本直接替换现有文件。如果指定了 *ext*，那么为原始文件生成一个备份，文件名是原有文件名加上 *ext* 后缀。

`-l`

这个选项使输出记录分隔符$\与输入记录分隔符$/（参见-0）相同，这样在用 print 进行文本输出时，会自动加入换行符。这个选项通常与-p 或-n 选项一起使用。当与其中之一同时使用时，则自动调用 chop 方法来删除每行输入记录的分隔符。

`-n`

这个选项使程序像使用如下循环一样运行：

```
while gets            # Read a line of input into $_
  $F = split if $-a   # Split $_ into fields if -a was specified
  chop! if $-l        # Chop line ending off $_ if -l was specified
  # Program text here
end
```

即使 Ruby 1.9 已经不再支持全局方法 chop!和 split，这个选项还是可以发挥同样的作用。

这个选项经常与选项-e配合使用。参见-p选项。

-p

这个选项使程序像使用如下循环一样运行：

```
while gets              # Read a line of input into $_
  $F = split if $-a     # Split $_ into fields if -a was specified
  chop! if $-l          # Chop line ending off $_ if -l was specified
  # Program text here
  print                 # Output $_ (adding $/ if -l was specified)
end
```

即使Ruby 1.9已经不再支持全局方法chop!和split，这个选项还是可以发挥同样的作用。

这个选项经常与选项-e配合使用。参见-n选项。

10.1.5 杂项选项
Miscellaneous Options

下面的选项不能归入上述的任何类别中。

-c

这个选项解析程序并报告其中的语法错误，但是并不运行这个程序。

-C *dir*, -X *dir*

这些选项在运行程序前把当前目录切换到dir下。

-s

如果指定了这个选项，解释器会预处理程序名后用连字符开头的参数。对于那些形如-x=y的参数，解释器把$x设置为y；对于形如-x的参数，解释器设置$x为true。这些预处理过的参数将从ARGV中被移除。

-S

这个选项使解释器在RUBY_PATH环境变量目录的相对路径下寻找指定程序。如果不能找到程序，解释器将接着在PATH环境变量目录的相对路径下寻找。如果仍然没有找到，再按照普通方式查找。

-T *n*

这个选项把$SAFE设置为*n*，如果*n*没有被指定，则设置$SAFE为1。更多信息可以参见10.5节。

-x [*dir*]

这个选项从给定程序中抽取Ruby源程序，抽取的程序不包含首个以#!ruby开头的文本行之前的内容。因为它兼容大写的-x选项，所以也可以指定一个目录。

10.2 顶层环境
The Top-Level Environment

当Ruby解释器启动时，它定义一些类、模块、常量、全局变量和全局方法，它们都可以被程序使用。下面的子小节将列举这些预定义的特性。

10.2.1 预定义的模块和类
Predefined Modules and Classes

当 Ruby 1.8 解释器启动时，解释器将定义下面的模块：

```
Comparable      FileTest        Marshal         Precision
Enumerable      GC              Math            Process
Errno           Kernel          ObjectSpace     Signal
```

同时定义下面的类：

```
Array           File            Method          String
Bignum          Fixnum          Module          Struct
Binding         Float           NilClass        Symbol
Class           Hash            Numeric         Thread
Continuation    IO              Object          ThreadGroup
Data            Integer         Proc            Time
Dir             MatchData       Range           TrueClass
FalseClass      MatchingData    Regexp          UnboundMethod
```

以及定义下面的异常类：

```
ArgumentError           NameError               SignalException
EOFError                NoMemoryError           StandardError
Exception               NoMethodError           SyntaxError
FloatDomainError        NotImplementedError     SystemCallError
IOError                 RangeError              SystemExit
IndexError              RegexpError             SystemStackError
Interrupt               RuntimeError            ThreadError
LoadError               ScriptError             TypeError
LocalJumpError          SecurityError           ZeroDivisionError
```

Ruby 1.9 还会定义下面的模块、类和异常：

```
BasicObject     FiberError      Mutex           VM
Fiber           KeyError        StopIteration
```

可以通过类似下面的代码在你的环境中检测预定义的模块、类和异常：

```
# Print all modules (excluding classes)
puts Module.constants.sort.select {|x| eval(x.to_s).instance_of? Module}

# Print all classes (excluding exceptions)
puts Module.constants.sort.select {|x|
  c = eval(x.to_s)
  c.is_a? Class and not c.ancestors.include? Exception
}

# Print all exceptions
puts Module.constants.sort.select {|x|
  c = eval(x.to_s)
  c.instance_of? Class and c.ancestors.include? Exception
}
```

10.2.2 顶级常量
Top-Level Constants

当 Ruby 解释器启动时，解释器将定义下面的顶级常量（以及前面列出的模块名和类名）。如果一个模块定义了同名的常量，可以通过在常量名前加上::前缀来访问这些顶级常量。我们可以通过如下方式列出环境中的顶级常量：

```
ruby -e 'puts Module.constants.sort.reject{|x| eval(x.to_s).is_a? Module}'
```

ARGF
: 一个 IO 对象，它把 ARGV 中指定的所有文件虚拟连接起来成为一个文件。如果 ARGV 为空，那么它表示标准输入。它是$<的同义词。

ARGV
: 一个数组，包含所有命令行指定的参数。它是$*的同义词。

DATA
: 如果程序文件中包含内容为__END__的行（该行只有__END__），那么这个常量表示该行后所有文本行的流。如果程序文件没有定义这样的行，那么这个常量未定义。

ENV
: 它是一个对象，其行为像一个哈希对象，可以通过它访问解释器中可用的环境变量。

FALSE
: false 的同义词，已被废弃。

NIL
: nil 的同义词，已被废弃。

RUBY_PATCHLEVEL
: 用于指示解释器补丁号（patchlevel）的字符串。

RUBY_PLATFORM
: 用于指示 Ruby 解释器运行平台的字符串。

RUBY_RELEASE_DATE
: 用于指示 Ruby 解释器发布日期的字符串。

RUBY_VERSION
: 用于指示解释器支持的 Ruby 语言版本号的字符串。

STDERR
: 标准错误流，是$stderr 变量的默认值。

STDIN
: 标准输入流，是$stdin 变量的默认值。

STDOUT
: 标准输出流，是$stdout 的默认值。

TOPLEVEL_BINDING

一个 Binding 对象，用于表示顶级范围的绑定。

TRUE

true 的同义词，已被废弃。

10.2.3 全局变量
Global Variables

Ruby 解释器预定义了一些全局变量，它们可以被程序所访问。这些变量中，有不少变量多少有些特殊：一些变量使用标点符号作为变量名；（English.rb 模块为这些标点符号定义了英语别名，通过加入 require 'English'语句，程序可以使用这些有点啰嗦的英语别名。）一些变量是只读的，不能进行赋值；一些变量是线程局部（thread-local）的，在不同的线程中看到该变量的值可能是不同的；还有一些全局变量（包括$_、$~和继承而来的模式匹配变量）是方法局部（method-local）的，它们的值局部适用于当前方法。如果一个方法设置了这些神奇变量中的一个变量，它不会对调用方法中的同名变量产生影响。

可以通过如下方式获取 Ruby 解释器预定义的全局变量列表：

```
ruby -e 'puts global_variables.sort'
```

如果想从 English 模块中获取更有意义的名字，可以通过如下方式获取：

```
ruby -rEnglish -e 'puts global_variables.sort'
```

下面的子小节将分类介绍这些预定义的全局变量。

10.2.3.1 全局设置

下面这些全局变量的值表示配置设置并指定具体信息，比如命令行参数、Ruby 程序运行的环境等。

$*

ARGV 常量的只读同义词，在 English 模块中的同义词是$ARGV。

$$

当前 Ruby 进程的进程 ID，它是只读的，在 English 模块中的同义词是$PID 和 $PROCESS_ID。

$?

最后一个结束程序的退出状态，它是只读且线程局部的，在 English 模块中的同义词是 $CHILD_STATUS。

$DEBUG, $-d

如果在命令行中设置了-d 或--debug 属性，这两个变量值为 true。

`$KCODE, $-K`

在 Ruby 1.8 中，这个变量的值是当前文本编码的名称。它的值可以是"NONE"、"UTF8"、"SJIS"或"EUC"。这个值可以通过解释器的选项`-K`进行设置。在 Ruby 1.9 中，这个变量不再有效，如果使用它会产生一个警告。

`$LOADED_FEATURES, $"`

一个字符串数组，表示已经加载的文件名。它是只读的。

`$LOAD_PATH, $:, $-I`

一个包含一组目录名的字符串数组，在使用 `load` 和 `require` 方法来加载文件时，会在这些目录下进行查找。这个变量是只读的，不过你可以修改它引用数组的值，比如可以在路径前部或尾部增加新的目录。

`$PROGRAM_NAME, $0`

它表示当前执行的 Ruby 程序文件名。如果程序是从标准输入设备读入的，它的值则为“-”；如果命令行指定了`-e`选项，它的值则为“e”。注意这个变量与`$FILENAME`是有区别的。

`$SAFE`

当前执行程序的安全级别，参见第 10.5 节可以获得更多细节。这个变量可以在命令行中通过`-T`选项设定，它是线程局部的。

`$VERBOSE, $-v, $-w`

如果命令行设置了`-v`、`-w`或`--versbose`选项，那么它们的值为 `true`；如果设置了`-W0`选项，则值为 `nil`；否则值为 `false`。如果设置该变量值为 `nil`，则忽略所有警告。

10.2.3.2 异常处理全局变量

在异常被抛出时，下面两个全局变量在 rescue 子句中很有用：

`$!`

最后抛出的异常对象，这个异常对象也可以通过 `rescue` 子句声明中的`=>`句法获得。这个变量是线程局部的，在 English 模块中的同义词是`$ERROR_INFO`。

`$@`

最后一个异常对象的调用堆栈，等价于`$!.backtrace`。它是线程局部的，在 English 模块中的同义词是`$ERROR_POSITION`。

10.2.3.3 流和文本处理的全局变量

下面是可以影响 `Kernel` 模块中文本处理方法默认行为的 `IO` 流和变量，我们可以在第 10.3 节中找到它们的示例。

$_

Kernel 模块方法 gets 和 readline 所读取的最后一个字符串，这个值是线程局部且方法局部的。有不少 Kernel 模块的方法都对$_进行隐式使用，它在 English 模块中的同义词是$LAST_READ_LINE。

$<

ARGF 流的一个只读同义词，如果在命令行中指定了多个文件，它就像这些文件连接起来的一个虚拟 IO 对象；如果没有指定文件，它代表标准输入 IO 对象。Kernel 模块的文件读取方法（比如 gets 方法）从这个流中进行读取。需要注意的是这个流并不总与 $stdin 相同。它在 English 模块中的同义词是$DEFAULT_INPUT。

$stdin

标准输入流。它的初始值是常量 STDIN，不过很多 Ruby 程序不从它读取信息，而是使用 ARGF 或$<进行读取。

$stdout, $>

标准输出流，是 Kernel 模块中各种打印方法的目标流：比如 puts、print 和 printf 等。它在 English 模块中的同义词是$DEFAULT_OUTPUT。

$stderr

标准输出流，它的初始值是常量 STDERR。

$FILENAME

ARGF 当前所读取的文件名，等价于 ARGF.filename。它是只读的。

$.

从当前输入文件中最后读取行的行号，等价于 ARGF.lineno，在 English 模块中的同义词是$NR 和$INPUT_LINE_NUMBER。

$/, $-0

输入记录分隔符（默认是换行符），gets 和 readline 方法默认使用这个值来确定行边界，可以通过解释器选项-0 来设定这个值。它在 English 模块中的同义词是$RS 和 $INPUT_RECORD_SEPARATOR。

$\

输出记录分隔符，默认值是 nil，可以通过解释器选项-l 进行设置。如果设置了非 nil 值，在每次调用 print（不过其他输出方法不会如此，比如 puts）后，会打印这个分隔符。它在 English 模块中的同义词是$ORS 和$OUTPUT_RECORD_SEPARATOR。

$,

print 方法打印输出时各个参数间的分隔符，也是 Array.join 方法的默认分隔符，默认值是 nil。它在 English 模块中的同义词是$OFS 和$OUTPUT_FIELD_SEPARATOR

$;, $-F

split 方法使用的默认字段分隔符，默认值是 nil，可以通过解释器选项-F 进行设置。它在 English 模块中的同义词是$FS 和$FIELD_SEPARATOR。

$F

当 Ruby 解释器使用-a 选项并搭配-n 或-p 选项进行启动时，解释器将定义这个变量。它持有当前输入行的各个字段，与对当前行使用 split 方法的返回值一样。

10.2.3.4 模式匹配全局变量

下面的全局变量都是线程局部且方法局部的，它们在进行正则表达式模式匹配时被设置。

$~

最后一次模式匹配操作所生成的 MatchData 对象，它是线程局部且方法局部的。下面提到的其他模式匹配全局变量都是从它继承而来的。如果把它设置为一个新的 MatchData 对象，会同时影响其他变量的值。它在 English 模块中的同义词是 $MATCH_INFO。

$&

最近匹配的文本，等价于$~[0]。它是只读的，而且是线程局部、方法局部的，继承自$~。它在 English 模块中的同义词是$MATCH。

$`

最新匹配文本之前的字符串，等价于*$~.pre_match*。它是只读的，而且是线程局部、方法局部的，继承自$~。它在 English 模块中的同义词是$PREMATCH。

$'

最后匹配文本之后的字符串，等价于*$~.post_match*。它是只读的，而且是线程局部、方法局部的，继承自$~。它在 English 模块中的同义词是$POSTMATCH。

$+

代表最后一次模式匹配中最后一个成功匹配分组的字符串，它是只读的，而且是线程局部、方法局部的，继承自$~。它在 English 模块中的同义词是$LAST_PAREN_MATCH。

10.2.3.5 命令行选项全局变量

对应于解释器命令行选项的状态或值，Ruby 定义了一组全局变量。其中变量$-0、$-F、$-I、$-K、$-d、$-v 和$-w 已在前面小节中进行了介绍，它们都有同义词。下面是前面未介绍的其他变量。

$-a

如果指定解释器选项-a，其值为 true；否则为 false。它是只读的。

$-i

如果未指定解释器选项-i，其值为 nil；否则，其值为-i 所指定的备份文件后缀名。

$-l

如果指定-l 选项，其值为 true。它是只读的。

`$-p`

如果指定解释器选项-p，其值为 `true`；否则为 `false`。它是只读的。

`$-W`

在 Ruby 1.9 中，这个全局变量指定当前的 verbose 级别。如果使用-W0 选项，其值为 0；如果使用-w、-v 或--verbose 选项中的任意一个，其值为 2；否则为 1。它是只读的。

10.2.4 预定义的全局函数
Predefined Global Functions

Object 对象包含的 Kernel 模块中，定义了一组私有实例方法，可以用作全局函数。因为它们都是私有方法，所以必须像函数一样被调用，不能显式指定接收者对象。因为这些方法被包含在 Object 对象中，所以可以在任何地方进行调用——不管 self 值是什么，它肯定是一个对象，这些方法就在这个对象上进行隐式调用。这些 Kernel 定义的函数可以被分为若干类别，其中绝大多数方法已经在本书的其他地方做过介绍了。

10.2.4.1 关键字函数

下面这些 Kernel 函数的行为就像是语言的关键字一样，它们已经在本书其他地方做过介绍了：

```
block_given?    iterator?    loop        require
callcc          lambda       proc        throw
catch           load         raise
```

10.2.4.2 文本输入、输出和操作函数

Kernel 模块定义的这些方法中，绝大多数是 IO 方法的变体。在第 10.3 节中对它们有更加详细的说明：

```
format          print        puts        sprintf
gets            printf       readline
p               putc         readlines
```

在 Ruby 1.8 中（Ruby 1.9 则不然），Kernel 还为 String 类的一些方法定义了全局性的变体方法，它们隐式使用$_进行操作：

```
chomp    chop    gsub    scan    sub
chomp!   chop!   gsub!   split   sub!
```

10.2.4.3 OS 方法

下面的 Kernel 函数使 Ruby 程序可以使用操作系统提供的接口。它们是依赖于所在平台的，我们将在第 10.4 节中对它们进行详细说明。注意`是一个具有特殊名字的 backtick 方法，它返回任意 OS 外壳命令的输出文本：

```
`        fork     select   system   trap
exec     open     syscall  test
```

10.2.4.4 警告、失败和退出

下面的 Kernel 函数会显式警告、抛出异常、导致程序退出或注册在程序结束时运行的代码块。它们将与那些依赖于操作系统的方法一起在第 10.4 节中被介绍：

```
abort   at_exit exit    exit!   fail    warn
```

10.2.4.5 反射函数

下面的 Kernel 函数是 Ruby 反射 API 的一部分，我们已在第 8 章中对其进行了介绍：

```
binding                         set_trace_func
caller                          singleton_method_added
eval                            singleton_method_removed
global_variables                singleton_method_undefined
local_variables                 trace_var
method_missing              untrace_var
remove_instance_variable
```

10.2.4.6 转换函数

下面的 Kernel 函数试图把它们的参数转换为新类型。我们已在第 3.8.7.3 节中对它们进行了介绍：

```
Array   Float   Integer String
```

10.2.4.7 杂项 Kernel 函数

下面的 Kernel 函数不能归入上面的任何类别中：

```
autoload                rand                    srand
autoload?               sleep
```

rand 和 srand 函数用于生成随机数，已在第 9.3.7 节中被介绍；autoload 和 autoload? 已在第 7.6.3 节中被介绍；而 sleep 已在第 9.9 节被介绍，我们还将在第 10.4.4 节中对其进行介绍。

10.2.5 用户定义的全局函数
User-Defined Global Functions

当在类定义或模块定义中使用 def 来定义一个方法时，如果不指定方法的接收者对象，这个方法会成为 self 的公开实例变量，这里的 self 就是正在定义的类或模块。如果在顶级范围中（在任何类和模块之外）使用 def 语句，则有两个显著区别：首先，顶级方法是 Object 类的实例方法（即使 self 并非 Object）；其次，顶级方法总是私有的。

顶级的 self：main 对象

因为顶级方法是 Object 类的实例方法，所以你可能会设想 self 的值是 Object。但是，

顶级方法其实是一个特例：方法的确在 Object 中被定义，但是 self 是另外一个对象。这个顶级的对象被称作“main”对象，对它我们没有什么好多说的。它的类对象是 Object，并具有一个 to_s 的单键方法，返回字符串 “main”。

因为顶级方法定义在 Object 中，所以可以被任何对象所继承（包括 Module 和 Class），并且可以（如果没有被覆盖）在任何类或实例方法的定义中使用。（可以回顾一下第 7.8 节中的方法名解析算法来帮助你确信这一点。）由于顶级方法是私有的，所以它们只能像函数一样被调用，不能显式带有接收者对象。通过这种方式，Ruby 在一个严格的面向对象框架中仿真了面向过程编程的样式。

10.3 实用性信息抽取和产生报表的快捷方式
Practical Extraction and Reporting Shortcuts

Ruby 在很大程度上受脚本编程语言 Perl 的影响，Perl 这个名字是实用性信息抽取和报表语言（Practical Extraction and Reporting Language）的简写，因此，Ruby 中有很多全局函数可以用于从文件中方便地抽取信息并产生报表。在面向对象的范例中，输入和输出函数是属于 IO 的方法，而字符串操作函数是属于 String 的方法。不过，出于实用的角度，通过全局函数对预定义的输入/输出流进行读写也是很有用的。除了这些全局函数，Ruby 还效仿 Perl 对这些函数定义了特殊行为：这些函数中很多都对特殊的方法局部变量$_进行隐式操作，这个变量持有最后一次从输入流读取的行。下划线字符可以这样来记忆：它看起来像是一行文本。（绝大多数 Ruby 的全局变量都使用标点符号作为名字，这是从 Perl 中继承而来的。）除了全局输入输出函数外，还有一些全局函数可用于字符串处理，它们的功能与 String 的方法类似，不过隐式对$_进行操作。

这些全局的函数和变量不过是一些快捷方式，可以被用于短小的 Ruby 脚本中。在大型程序中，这通常被认为是一种不好的形式。

10.3.1 输入函数
Input Functions

全局函数 gets、readline 和 readlines 与 IO 中的同名方法相似（参见第 9.7.3.1 节），不过它们隐式对$<流（也可以通过常量 ARGF 进行访问）进行操作。与 IO 中的方法一样，这些全局函数隐式设置$_变量。

$<的行为类似于一个 IO 对象，不过它并非一个真正的 IO 对象。（它的 class 方法返回 Object，to_s 方法返回 “ARGF”。）这个流的实际行为是相当复杂的。如果 AGRV 为空，则$<与 STDIN 等价，作为标准输入流。如果 ARGV 非空，Ruby 会假定它是一组文件名，这时，$<就像是

这些文件的连接体一样。不过这并非对$<行为的准确描述。当第一次对$<进行读取请求时，Ruby 调用 ARGV.shift 方法移除 ARGV 中第一个文件名，然后打开这个文件进行读取；当到达这个文件的尾部时，Ruby 重复这个过程，从 ARGV 中移出下一个文件名并打开这个文件。只有在 ARGV 中没有文件名时，$<才会报告文件结束（EOF）。

这意味着你的 Ruby 脚本可以在开始从$<读取之前更改 ARGV（比如用于处理命令行参数的程序），可以让脚本在运行时对 ARGV 添加一些额外的文件名，而$<同样会打开这些额外的文件。

10.3.2 废弃的文本抽取函数
Deprecated Extraction Functions

在 Ruby 1.8 及以前的版本中，全局函数 chomp、chomp!、chop、chop!、gsub、gsub!、scan、split、sub 和 sub! 与 String 中的同名方法效果相似，不过它们是隐式对$_进行操作，而且，chomp、chop、gsub 和 sub 把结果赋值回$_中，这意味着它们实际上是那些带有感叹号结尾方法的同义词。

这些全局函数已经从 Ruby 1.9 中删除，因此不能在新代码中使用。

10.3.3 报表函数
Reporting Functions

Kernel 定义了一组全局方法用于对$stdout 进行输出。（这个全局变量在 Ruby 进程初始化时用于引用标准输出流 STDOUT，不过可以更改这个值并用下面介绍的方法改变它的行为。）

puts、print、printf 和 putc 等价于它们在 STDOUT 中的同名方法（参见第 9.7.4 节）。前面讲过 puts 会在输出文本尾部没有换行符时增加一个换行符，而 print 则不会自动增加一个换行符，但是如果设置了输出记录分隔符全局变量$\，print 会自动增加换行符。

全局函数 p 在 IO 类中没有相似方法，它是为调试所创建的，其简短的名字使之适合于快速输入。它调用每个参数的 inspect 方法并把结果传送给 puts 方法。前面讲过默认的 inspect 方法等价于 to_s 方法，不过一些类对之进行了重新定义使结果对开发者更加易读。如果加载了 pp 库，我们可以使用 pp 方法来替代 p 方法，从而获得调试结果的“优美打印（pretty print）”。（它对打印大型数组和哈希表很有用。）

前面提到 printf 方法的第一个参数是一个格式字符串，在真正进行输出前，printf 方法用其余的参数来替换格式字符串中的一些标识，我们也可以用 sprintf 方法（或其同义词方法）把格式化的结果输出到一个字符串中而非$stdout 中，它们与 String 类的%操作符相似。

10.3.4 运行单行脚本时可用的快捷方式
One-Line Script Shortcuts

在本章前面提到，我们可以用-e这个解释器选项来运行单行Ruby脚本（经常与-n和-p选项联合使用）。这里有一个从Perl继承来的特殊快捷方式，它只用于-e参数指定的单行脚本。

如果用-e指定了执行单行脚本，而这个脚本中有条件表达式（if、unless、while或until语句）包含一个正则表达式字面量，那么这个正则表达式隐式与$_进行匹配。比如，如果想打印一个文件中所有以字母A打头的行，可以使用如下代码：

```
ruby -n -e 'print if /^A/' datafile
```

如果这行代码出现在一个脚本中而非使用-e选项运行，它也能工作，不过将输出一个警告信息（即使不带-w选项）。为了避免这个警告，最好把它改成显式比较的方式：

```
print if $_ =~ /^A/
```

10.4 调用操作系统的功能
Calling the OS

Ruby有不少全局函数用于和操作系统交互，可以运行程序、创建新进程、处理信号等，Ruby最初被开发于类Unix操作系统中，因此这些函数中有很多都反映了这种操作系统的功能。由于这些函数的性质，它们不像其他函数那样易于移植，有些函数不能在Windows平台或其他非Unix平台上实现。下面的子小节将介绍那些最常用的依赖于操作系统的函数，而像syscall这样低级或平台相关的函数不会在这里被介绍。

10.4.1 调用操作系统命令
Invoking OS Commands

Kernel.`方法接受一个代表操作系统外壳命令的字符串参数，它启动一个子外壳并把给定的参数传递给后者，方法的返回值是输出到子外壳标准输出中的文本。这个方法通常用特殊的语法进行调用：它可以通过用反引号括起的字符串字面量调用，或者通过%x分隔的字符串字面量调用。示例如下：

```
os = `uname`              # String literal and method invocation in one
os = %x{uname}            # Another quoting syntax
os = Kernel.`("uname")    # Invoke the method explicitly
```

这个方法并不直接调用给定的可执行命令，而是创建一个外壳来执行命令，这意味着像文件名通配符这样的外壳特性也是可用的：

```
files = `echo *.xml`
```

启动一个进程并读取输出的另外一种方法是使用 Kernel.open 函数。这个方法是 File.open 的一个变体，通常用于打开文件。（如果使用 require 'open-uri' 语句，也可以用于打开 HTTP 和 FTP 的 URL 链接。）不过如果文件名的首字母是管道字符“|”，它打开一个管道对给定的外壳命令进行读取或写入：

```
pipe = open("|echo *.xml")
files = pipe.readline
pipe.close
```

如果想在外壳中执行一个命令但不关心它的输出，可以使用 Kernel.system 方法。如果传入单个字符串，它在外壳中执行这个字符串所表示的命令，等待命令结束，如果成功，返回 true，失败则返回 false。如果传递多个参数，第一个参数是程序名，其余的参数则成为它的命令行参数，这时，这些参数不会在外壳中被展开。

用于执行任意可执行文件的底层方式是使用 exec 函数。这个函数永远不会返回，它只是用给定的可执行文件取代当前的 Ruby 进程。如果编写一个执行其他程序的封装器脚本，那么这个函数可能会有所帮助。不过，这个函数通常与 fork 函数配合使用，其细节将在下一节描述。

10.4.2 进程和创建子进程
Forking and Processes

在第 9.9 节中讲述了用于编写多线程程序的 Ruby API。在 Ruby 中另外一种实现并发性的方式是使用多个 Ruby 进程，这可以通过 fork 函数或其同义词方法 Process.fork 来实现。使用这个函数最简单的方式是通过一个块：

```
fork {
  puts "Hello from the child process: #$$"
}
puts "Hello from the parent process: #$$"
```

通过这种方式，原始的 Ruby 进程继续执行代码块之后的代码，而新创建的 Ruby 进程则执行代码块中的代码。

如果不使用代码块，fork 的行为会有所不同。在父进程中，对 fork 函数的调用会返回一个整数值，它代表新创建的子进程的 ID；而在子进程中，同样对 fork 的调用则返回 nil。这样，前面的代码可以用如下方式实现：

```
pid = fork
if (pid)
  puts "Hello from parent process: #$$"
  puts "Created child process #{pid}"
else
  puts Hello from child process: #$$"
end
```

进程和线程的一个十分重要的区别在于进程不共享内存。当调用 fork 时，新创建的 Ruby 进程会精确复制父进程。但是任何对进程状态的修改（通过修改或创建对象实现）都在自身的地址空间中完成，子进程不能修改父进程的数据结构，而父进程也不能修改子进程中的数据结构。

如果想让父进程和子进程进行通信，可以使用 open 函数，并用“|-”作为它的第一个参数。这个函数为新创建的 Ruby 进程打开一个管道，在父进程和子进程中都会把控制权交给 open 所关联的代码块。在子进程中，代码块所接收的参数为 nil；而在父进程中，代码块接收一个 IO 对象，对这个 IO 对象进行读取可以得到子进程所写的数据；而对这个 IO 对象中写入的数据可以通过子进程的标准输入读出。示例如下：

```
open("|-", "r+") do |child|
  if child
    # This is the parent process
    child.puts("Hello child")          # Send to child
    response = child.gets              # Read from child
    puts "Child said: #{response}"
  else
    # This is the child process
    from_parent = gets                 # Read from parent
    STDERR.puts "Parent said: #{from_parent}"
    puts("Hi Mom!")                     # Send to parent
  end
end
```

Kernel.exec 函数在与 fork 或 open 配合使用时有特别的用途。前面我们看到可以使用`和 system 向操作系统发送任意的命令，不过这两个方法都是同步方式的，在命令完成前不会返回。如果想在一个单独的进程中执行系统命令，首先要使用 fork 来创建子进程，然后调用在子进程中的 exec 来运行命令。对 exec 的调用永远不会返回，它用新创建的进程替代当前进程。exec 的参数与 system 的相同。如果只有一个参数，该参数被当作外壳命令处理；如果有多个参数，第一个参数被当作可执行命令，剩下的参数成为可执行命令的“ARGV”变量：

```
open("|-", "r") do |child|
  if child
    # This is the parent process
    files = child.readlines    # Read the output of our child
    child.close
  else
    # This is the child process
    exec("/bin/ls", "-l")      # Run another executable
  end
end
```

用进程编程是一种底层编程技术，它的细节超出了本书的范围。如果想获得更多知识，可以使用 ri 来获得其他 Process 模块方法的信息。

10.4.3 捕获信号
Trapping Signals

绝大多数操作系统支持对当前运行的进程发送异步信号。当用户键入 Ctrl-C 来中止一个程序时，所发生的事情就是异步信号的一个例子。大多数的外壳程序在接收到 Ctrl-C 后，将发送一个名为“SIGINT”（用于中断运行）的信号，该信号默认的处理方式是中止程序，Ruby 允许程序“捕获”信号并定义自己的信号处理程序。这是通过 Kernel.trap 方法（或同义词方法 Signal.trap）实现的。例如，如果不想让用户使用 Ctrl-C 退出程序，你可以编写如下代码：

```
trap "SIGINT" {
  puts "Ignoring SIGINT"
}
```

除了给 trap 方法传递一个代码块，也可以给它传送一个等价的 Proc 对象。如果只是想简单忽略一个信号，可以用字符串“IGNORE”作为第二个参数。如果用“DEFAULT”作为第二个参数，操作系统将恢复该信号的默认处理方式。

在一个长时间运行的程序（比如服务器）中，有时须要定义一个信号处理方法完成诸如重新读取配置文件、输出信息到日志文件或进入调试模式的功能，在类 Unix 的操作系统上，我们通常使用 SIGUSR1 和 SIGUSR2 信号来达到这样的目的。

10.4.4 结束程序
Terminating Programs

一些相关的 Kernel 方法用于结束程序或相关的动作，exit 函数是最直接的，它抛出一个 SystemExit 异常，如果该异常没有被捕获，则导致程序退出。不过在退出前，END 块及任何用 Kernel.at_exit 注册的处理方法被执行。如果要立刻退出，可以使用 exit!方法。这两个方法都接受一个整数参数，它代表进程退出码，将被提供给操作系统。Process.exit 和 Process.exit!是这两个 Kernel 函数的同义词函数。

abort 函数向标准输出流打印指定的错误信息，然后调用 exit(1)。

fail 是 raise 的同义词方法，它一般在捕获到须要结束程序的异常时使用。像 abort 一样，fail 在程序退出时显示一个消息，例如：

```
fail "Unknown option #{switch}"
```

warn 函数与 abort 和 fail 有点关系：它向标准错误流打印一个警告消息（除非通过-W0选项显式禁止警告）。不过要注意的是，这个函数不会抛出异常或退出程序。

sleep 是另外一个相关函数，它不会让程序退出，只是让程序（或至少是程序的当前运行线程）暂停指定的秒数。

10.5 安全
Security

Ruby 的安全体系提供了一种机制，可以让程序与非受信的数据和代码共同工作。这个安全体系由两个部分构成，第一个部分是从非受信或**污染的**数据中识别安全数据的机制，第二个部分是一种**受限运行**的技术，它允许“锁定（lock down）”Ruby 环境，并防止 Ruby 解释器在污染的数据中执行有潜在危险的操作。这可以用于防止类似 SQL 注入的攻击，在这种攻击中恶意的输入会改变程序的行为。受限运行则可以更进一步，它可以让你无须担心执行的非受信代码（很可能是恶意的）会删除文件、窃取数据或做其他有害的动作。

10.5.1 可污染的数据
Tainted Data

Ruby 中的每个对象要么是可污染的，要么是不可污染的。在源程序中的字面量是不可污染的，而从外部环境中继承来的对象值是可污染的，这包括从命令行读入的字符串（ARGV）或环境变量（ENV），以及从文件、套接字或其他流读入的数据。环境变量 PATH 是一个特例：它只有在包含一个或多个处处可写的目录时才是可污染的。重要的一点是，污染性是可传染的，从可污染对象继承来的对象也是可污染的。

Object 的方法 taint、tainted?和 untaint 可以把一个不可污染的对象标记成可污染的，还可以检测一个对象的可污染性，以及把一个可污染的对象标记成不可污染的。只有在审视了代码后，知道尽管其来源不安全，但它本身是安全的时候，才可以把可污染的对象标记为不可污染的。

10.5.2 受限执行和安全级别
Restricted Execution and Safe Levels

Ruby 可以在开启**安全检测**的情况下执行程序，全局变量$SAFE 用于确定安全检查的级别。默认的安全级别一般是 0，但是运行 *setuid* 或 *setgid* 的 Ruby 程序安全级别默认为 1。（这些都是 Unix 的术语，表示一个程序运行于高于调用者的权限上。）合法的安全级别有 0、1、2、3 和 4，我们可以用 Ruby 解释器的命令行选项-T 来显式指定安全级别，也可以通过对$SAFE

赋值来设置安全级别。不过，须要注意的是，我们只能增加它的值，永远不可能降低：

```
$SAFE=1              # upgrade the safe level
$SAFE=4              # upgrade the safe level even higher
$SAFE=0              # SecurityError!  you can't do it
```

$SAFE是线程局部的。换句话说，可以在一个线程中改变$SAFE的值，而不影响其他线程中$SAFE的值。通过这一特性，线程可以变成执行非可信代码的沙箱：

```
Thread.start {       # Create a "sandbox" thread
  $SAFE = 4          # Restrict execution in this thread only
  ...                # Untrusted code can be run here
}
```

这里讨论的Ruby安全级别是与引用实现相关的，其他的实现则可能不同，尤其是JRuby，它的引用实现几乎没有尝试去模拟受限执行模式（在写作本书时是如此）。另外，记住，Ruby的安全模型不像Java的安全模型那样久经考验。下面的子小节将解释Ruby的受限执行是如何工作的，不过你可能会发现新的bug可以绕过这些限制。

10.5.2.1 安全级别0

安全级别0是默认的安全级别，不对可污染数据进行检查。

10.5.2.2 安全级别1

在这个级别上，使用可污染数据的潜在危险操作是被禁止的。不能对可污染的字符串进行求值；如果一个库名是可污染的，不能对它进行 `require`；如果一个文件名是可污染的，不能打开该文件；并且，如果主机名是可污染的，也不能通过网络连接它。允许任意输入的程序，尤其是那些联网的服务器程序，应该使用这个安全级别，这有助于发现那些通过不安全方式使用可污染数据的错误。

如果你要编写一个会执行潜在危险操作的库——比如与一个数据库服务器交互的库——那么你须要检查$SAFE的值。如果它的值为1或更高，你的库不应该对可污染的对象进行操作。例如，如果包含SQL查询的字符串是可污染的，你不应该把它发送给数据库服务器。

在安全级别为1的情况下，对程序执行的限制如下。

- 在启动时会忽略环境变量RUBYLIB和RUBYOPT。
- 在$LOAD_PATH中不包括当前目录（.）。
- 禁止使用命令行参数-e、-i、-I、-r、-s、-S和-X。
- Dir、IO、File和FileTest中有可污染参数的实例方法和类方法被禁用。

- 不能用可污染的参数调用 test、eval、require、load 和 trap。

10.5.2.3 安全级别 2

在安全级别为 2 的情况下，系统既限制级别 1 上的那些对可污染数据的操作，还对文件和进程的操作进行了限制，不管它们是不是可污染的。很少有理由把程序的安全级别设置为 2，不过系统管理员可以用这个安全级别限制程序不能创建或删除目录、改变文件权限、执行应用程序、从处处可写的目录中加载 Ruby 代码等。

在这一级别，被系统限制的方法有：

```
Dir.chdir                File.truncate            Process.egid=
Dir.chroot               File.umask               Process.fork
Dir.mkdir                IO.fctrl                 Process.kill
Dir.rmdir                IO.ioctl                 Process.setpgid
File.chmod               Kernel.exit!             Process.setpriority
File.chown               Kernel.fork              Process.setsid
File.flock               Kernel.syscall
File.lstat               Kernel.trap
```

另外，安全级别 2 禁止从处处可写的目录中加载（require 或 load）Ruby 代码或执行应用程序。

10.5.2.4 安全级别 3

安全级别 3 包含所有安全级别 2 的限制，并且所有的对象——包括源程序中的字面量（但是不包括全局环境中预定义的对象）——在创建时都是可污染的。同时，untaint 方法是禁用的。

安全级别 3 是通向安全级别 4 的一个中间级别，通常不使用。

10.5.2.5 安全级别 4

这个级别继承自安全级别 3，并阻止对任何不可污染对象的修改（包括对不可污染对象执行 taint 操作）。在这一级别上运行的程序不能修改任何全局环境，也不能修改任何在前面低安全级别上创建的不可污染对象，这实际上创建了一个沙箱，在沙箱中那些不可信的代码可以运行，但不会造成任何危害。（至少在理论上如此，未来可能会发现具体实现中的 bug，或者发现底层安全模型上的缺陷。）

在安全级别 1、2 和 3 上，都禁止对可污染的字符串执行 eval 语句。不过在安全级别 4 上，它又被允许了，这是因为安全级别 4 足够严格，对字符串的求值不会带来任何危害。下面演示了如何在第 4 级安全的沙箱中对任意代码进行求值：

```
def safe_eval(str)
  Thread.start {                    # Start sandbox thread
    $SAFE = 4                       # Upgrade safe level
```

```
    eval(str)                    # Eval in the sandbox
  }.value                        # Retrieve result
end
```

在安全级别4上，不能使用require语句加载其他Ruby文件，只能用包装的方式使用load，即把第2个参数设置为true，这使Ruby把加载的文件放入一个由匿名模块构成的沙箱中，这样它定义的类、模块和常量不会影响全局命名空间。这意味着在第4级安全下，程序可以加载外部模块定义的类和模块，但不能使用它们。

还可以更进一步限制第4级别的安全沙箱，把沙箱线程（在设置$SAFE变量前）放入一个ThreadGroup中，然后在这个线程组内部进行调用。参见9.9.5节可获得更多细节。

作为第4级安全所创建沙箱的一部分，下面的操作是被禁止的：

- require、无包装的load、autoload和include
- 修改Object类
- 修改不可污染的类或模块
- 元编程方法
- 操作非当前线程
- 访问线程局部数据
- 中止进程
- 文件输入/输出
- 修改当前环境变量
- 用srand设置随机数生成器的种子

索引
Index

Symbols

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

关于作者

David Flanagan：一位计算机程序员，绝大部分时间花在编程语言上。他还在 O'Reilly 公司出版了 *JavaScript: The Definitive Guide and Java in a Nutshell* 一书。David 拥有麻省理工学院的计算机科学和工程学位，他和妻子及孩子在美国太平洋西北地区的一些城市生活，包括西雅图、华盛顿及英属哥伦比亚的范库弗峰市。

Yukihiro Matsumoto ("Matz")：Ruby 的缔造者，一位专业程序员，为日本开源软件公司 netlab.jp 工作。Matz 也是日本开源软件的布道者，发布了若干开源产品，cmail 是其中之一，这是一个基于 Emacs 的邮件用户代理，完全用 Emacs Lisp 语言编写。Ruby 是第一款他在日本之外被广泛熟知的软件。

封面动物介绍

本书的封面动物是一种南美蜂鸟（Horned Sungem hummingbird），这些小鸟是南美洲的土著鸟类，主要生活在巴西和玻利维亚。它们喜欢像草地这样干旱开阔的栖息地，而不喜欢稠密或潮湿的森林。

蜂鸟是世界上翅膀震动最快的鸟类，这种南美蜂鸟更是可以每秒振翅 90 次。（可以跟振翅最慢的秃鹫对比，秃鹫每秒只能振翅一次。）因为蜂鸟如此轻快，它们可以依靠挥动翅膀而停在空中。为了方便吮吸花蜜，它们可以向后飞（所有鸟类中只有蜂鸟可以做到这点）。它们长而细的喙可以探伸到花的深处，因而，它们在葡萄牙语中的名字是“beija-flor”，意为亲吻鲜花的鸟类。而英文名字“hummingbird”则是来自于它们快速挥动翅膀时发出的嗡嗡声（译注 1）。

雄性 Horned Sungem 在脑袋两侧有一簇簇红色、蓝色和金色的羽毛，背部呈虹彩色泽的绿色，喉部和胸部为黑色，腹部则为白色，尾巴长而尖。雌鸟与雄鸟看起来很相似，但没有那个像皇冠一样的头部羽毛。因为这种鸟颜色艳丽，早期的西班牙探险者称之为“Joyas voladoras”，意为“飞翔的宝石”。

关于蜂鸟有很多神话传说。在巴西，黑色蜂鸟用于表示家庭成员中有人死亡。古代阿兹泰克斯人非常崇拜蜂鸟，牧师会用蜂鸟毛覆盖受诅咒的人，用于驱除诅咒。同时，蜂鸟还是死者重生的标志，阿兹泰克斯人相信死去的战士会和这些鸟一块重生。阿兹泰克斯人的上帝 Huitzilopochtli 是太阳和战争的化身，他名字的意思是"南方飞来的蜂鸟"，而南方是神灵的居住之所。

译注 1：hum 的意思为“嗡嗡声”。