

# Ruby基础教程（第4版）

书号	978-7-115-36646-7
出版日期	2014-09
页数	352
定价	79.00 元
印刷方式	黑白
类别	<a href="#">rubyruby2.0</a>

★ Ruby入门第一书，原版重印27次！

★ 松本行弘亲自审校并作推荐序

★ 日本Ruby协会创始人兼会长倾情力作

“这是一本绝对不会让初学者失望的Ruby入门书。”

——Ruby之父 松本行弘

本书为日本公认的最好的Ruby入门教程。松本行弘亲自审校并作序推荐。本书支持最新的Ruby 2.0，也附带讲解了可运行于1.9版本的代码，事无巨细且通俗易懂地讲解了编写程序时所需要的变量、常量、方法、类、流程控制等的语法，以及主要类的使用方法和简单的应用，让没有编程经验的读者也能轻松掌握Ruby，找到属于自己的快乐编程方式，做到融会贯通并灵活运用到实际工作中。

本书适合Ruby初学者学习参考，有一定Ruby编程基础的读者若想再回顾一下Ruby的各知识点，本书也能提供不少帮助。

<作译者介绍>

高桥征义 (Masayoshi Takahashi)

日本Ruby协会创始人兼会长。日本著名的IT书籍电子书平台达人出版会董事长。著有《Ruby基础教程》《Rails3绝技190招》等。喜欢的作家是新井素子。

后藤裕藏 (Yuuzou Gotou)

日本网络应用通信研究所董事。喜欢平克·弗洛伊德乐队。

松本行弘 (Yukihiro Matsumoto)

Ruby语言发明者，亦是亚洲首屈一指的编程语言发明者。现兼任网络应用通信研究所 (NaCI) 研究员、乐天技术研究所研究员、Heroku首席架构师等。昵称“Matz”。讨厌东京，喜欢温泉。

何文斯

上海交通大学电子工程系研究生毕业，现就职于某国际独立软件开发商，从事软件售后支持工作。对面向对象的程序设计，脚本语言及其在语音、图像等信号处理中的应用有着浓厚的兴趣。

原书书名	たのしいRuby 第4版
原书书号	978-4797372274
原书国家	日本
原书出版社	SoftBank Creative
原书页数	520

## 推荐序

Ruby 常常被称为“国产语言”。作为 Ruby 的设计者，我的确是个如假包换的日本人，Ruby 最具代表性的实现——CRuby<sup>1</sup> 中的许多核心成员也都是日本人。但是，对 Ruby 的开发和发展做出过大大小小贡献的人里有很多都不是日本人。在 JVM 上使用的 JRuby、用 C++ 实现的 Rubinius 等，其主要开发者是美国人；MacRuby、RubyMotion 的主要开发者是比利时人。还有使 Ruby 发扬光大的 Ruby 社区，其大部分活动都在日本以外的国家或地区进行。Ruby 社区里最具代表性，也是最早的 Ruby 技术大会——RubyConf 每年都会在美国举行。除此以外，在美洲、欧洲、亚洲等世界各地也都会举办其他的 Ruby 技术大会。Ruby 是无数人努力的结晶，是一个社区，是一种文化。所以，我多少有点反感因 Ruby 诞生于日本就将其冠以“国产”的说法。

<sup>1</sup>也称 MRI。——译者注

但是，日本也有引以自豪之处。一是日本拥有世界上最早建立的 Ruby 社区。我访问过许多国家的社区，可以说日本的社区是世界上水平最高的。另外，日本拥有一批经验丰富的 Ruby 社区成员。他们是 Ruby 最早的一批使用者，并通过各种活动和实际开发，孕育出了属于 Ruby 自身的多元文化。这样的人才是 Ruby 社区中可贵的瑰宝。

本书是最早使用 Ruby 的先驱者们为了欢迎下一批社区成员而写的一本入门书。本书前 3 个版本帮助过无数新人融入到 Ruby 社区。这次，对应 Ruby 最新版的第 4 版比以往更详细，更通俗易懂，对大家学习 Ruby 会有很大的帮助。通过学习本书，衷心希望大家体会到 Ruby 编程带来的乐趣。

2013 年 4 月

松本行弘

## 译者序

---

曾经有同事问我，为什么这么喜欢 Ruby？我的回答是，因为 Ruby 非常有趣，用 Ruby 写程序是一件快乐的事情。对方满脸困惑，似乎在质疑——写程序也能让人感到快乐？的确，现在不少人认为编程是一件又苦又累的差事。代码搬运工、码农等大家的自嘲语也很难让人把编程与快乐联系在一起。回想当初刚学习编程的时候，我们曾因为实现了某个算法、某个功能而感到兴奋，而工作后却被项目进度、加班等压得喘不过气来，似乎已经忘记了编程原本是一件令人快乐的事情。

“快乐编程”是本书的主旨，也是 Ruby 令人着迷的原因之一。本书继承了日语技术类书籍的优良传统，采用了大量图、表、例子，讲解通俗易懂。从编程基础的数据类型、控制语句，到面向对象编程、鸭子类型、正则表达式等高级编程技巧，带领着读者逐步进入 Ruby 的程序世界，使大家沉浸在编程的乐趣之中。而对于久经沙场的“老鸟”们，Ruby 那如诗篇一样优雅的语法、各种魔术般的语法糖，以及能把我们从枯燥无味的重复劳动中解放出来的丰富强大的类库，都一定都能唤起大家的“集体回忆”，重拾已经失去的编程乐趣。

2007 年接触 Ruby 后，我就喜欢上了这个“小家伙”。偶然一次机会，我从 Ruby China 社区得知图灵公司正在寻找这本书的译者。非常幸运，我得到了这个宝贵的机会。

在此非常感谢图灵公司以及 Ruby China 社区，也非常感谢翻译过程中图灵公司各位编辑给予的帮助。

这是我第一次译书，其间所耗费的时间与精力远远超出了当初的预期。翻译期间，我牺牲了很多与家人共处的时间，在此深深感谢家人们的谅解、关心与支持，同时也非常感谢朋友们、同事们在这段日子里给我的鼓励与支持。

参与本书的翻译，是我人生中一次奇妙的经历。记得以前我曾经对计算机硬件非常着迷，经常阅读硬件杂志。记得当时有一本计算机硬件入门杂志，整本都是采用彩色铜版纸印刷，图文并茂，手把手地教读者装配、使用计算机。不过慢慢地，有读者抱怨内容太浅显，希望作者能写点高深的内容。当时杂志编辑的一段回复，到现在我还记忆犹新，大意是“我们的任务就是迎接更多的新朋友，同时让更多的老朋友抛弃我们，当你觉得我们已经无法满足你的求知欲时，那么恭喜你，你已经毕业了，我们的任务也完成了”。这也是我此刻的心情。

最后，预祝大家通过本书都能找到属于自己的 Ruby 快乐编程之道。

何文斯

2014 年 5 月 4 日，写于广州

# 前言

## 乐在其中的编程语言

与计算机程序“交流”的方式有两种。第一种方式是使用程序，另外一种是编写程序。

然而，编写程序的人相对要少，大部分人都是使用程序而已。这个有点接近“读文章的人”与“写文章的人”的比例。读小说、散文、纪实文学等的人很多，但写小说、纪实文学的人数量上就远比读者少。

这里说的“文章”不仅仅是指商业出版物，还包括个人网站。有很多人几乎每天都更新博客，有的是与身边的人分享有趣的事情，有的是提供某些有用的信息。虽然可能只是一些微不足道的信息，但还是会有读者乐于阅读，这类读者就是“用户”，因此博客也可以说是一种“供读者阅读的文章”。

大家基于各种目的创建了这类网站，其中不少人是因为很享受自己编写内容的过程。以个人网站为例，单纯追求创作乐趣的人可能会更多。

编程不也是如此吗？也就是说，并不仅仅是为了某种目的而编程，而是因为编程时乐在其中。

编程的乐趣并非单指程序内容，使用的编程语言不同，所获得的乐趣也不一样。像这样，让编程本身变得有趣的编程语言真的存在吗？

——存在。Ruby 就是其中一种。

\*\*\*

Ruby 是一种旨在使大家编程时能乐在其中的编程语言。完全面向对象，有丰富的类库，直观、人性化的语法等都是 Ruby 的特征，但这些并不是 Ruby 的目的，只是快乐编程的手段。

在程序世界里，有着种类繁多的语言。这些语言诞生的缘由多种多样，有的是为了编写运行速度快的程序，有的是为了可以在短时间内编写程序，有的是为了让程序只需编写一次就可以在任何环境中运行，有的是为了使小孩也能进行简单编程，等等。但是，似乎并没有哪个语言积极地宣称其目的是为了快乐编程。这可能是由于各个语言的设计者，并没有认真考虑过让任何人都可以编程。

当然，使大家编程时乐在其中的语言，肯定是一种简单易掌握的语言，复杂的语言不可能让人体会到快乐。同时，这门语言又必须是一个功能强大的语言，若非如此，实际编写程序时会非常费劲。毋庸置言，Ruby 就是这样一种简单易掌握，并且功能强大的编程语言。

\*\*\*

为了让零编程经验的读者轻松掌握 Ruby，本书会巨细无遗地介绍 Ruby。从编写程序时所需要的变量、常量、方法、类、流程控制等的语法说明，到主要类的使用方法和简单的应用，都会尽量用通俗易懂的方式来说明。对于从未接触过计算机的读者来说，也许这有点难，但是那些稍微懂点 HTML 的读者很容易就能做到融会贯通。另外，对于那些并非初学者的读者来说，若想再回顾一下 Ruby 的各知识点，本书也能提供不少帮助。

希望各位读者能通过本书，熟练掌握 Ruby，找到属于自己的快乐而有趣的编程方式，并灵活运用到实际中，笔者将不胜荣幸。

欢迎来到 Ruby 的世界！

高桥征义 | 后藤裕藏

# 本书的读者对象

## 0.1 关于 Ruby

在开始编程之前，让我们先了解一下什么是 Ruby。

- **Ruby 是脚本语言**

用 C 或者 Java 语言编写的程序，在运行前需要执行编译这一步骤，把源码翻译成计算机可以理解的机器码。而用脚本语言编写的源码并不需要编译，直接运行程序便可。

也就是说，在使用脚本语言时，开发流程会从

源码编写 → 源码编译 → 程序运行

变为

源码编写 → 程序运行

因此，与需要编译的语言相比，Ruby 更能让大家轻松享受到编程之趣。

- **Ruby 是面向对象的语言**

Ruby 是一群热爱面向对象编程的程序员，为了实现最优秀的面向对象语言而设计、开发的一门语言。它是完全面向对象的，所思考的东西都可以直接通过代码表达出来。<sup>1</sup> 同时，Ruby 也具有继承、Mix-in 等面向对象语言的必备特性。

另外，Ruby 不仅提供了丰富的标准类库，还具有对应各种异常的错误处理机制、自动释放内存的垃圾回收机制等提高编程效率的特性。

- **Ruby 是跨平台的语言**

Ruby 能在 Mac OS X、Linux、FreeBSD、Solaris 等类 Unix 操作系统以及 Windows 操作系统等平台上运行。它的大部分脚本无需修改即可在各个不同的平台环境下运行。

- **Ruby 是开源软件**

Ruby 诞生时，松本行弘先生就公开了源码，使之成为开源软件（自由软件，Free Software）。任何人都可以随意获取 Ruby，并自由使用。自 1995 年松本行弘先生在互联网上发布 Ruby 以来，Ruby 得到了来自各方的广泛支持，并一直活跃至今。

1与面向过程的编程方法相比，我们一般认为面向对象的编程方法比较符合人的思维习惯。——译者注

## 0.2 本书的读者对象

本书是一本入门级图书，面向具备一定计算机知识但没有编程经验的读者，旨在帮助他们掌握 Ruby 编程知识。本书尽量以无需具备专业知识也能读懂的方式向大家介绍 Ruby，但省略了“启动 / 关闭计算机”“Shift 键的使用方法”等基础知识的说明。本书面向这样的读者：

- 具备操作文件和执行命令等基础的计算机知识
- 可使用编辑器创建文本文件
- 计划学习编程

## 0.3 本书的构成

本书采用“循序渐进，逐步深入”的写作方式，对于有 Ruby 基础的读者来说，前半部分或许有些沉闷。建议已经掌握 Ruby 语法等基础知识的读者，快速浏览前两部分，从第 3 部分开始仔细阅读。

- **第 1 部分 Ruby 初体验**

利用简单的 Ruby 小程序，介绍计算机程序的基本构成。

- **第 2 部分 Ruby 的基础**

介绍 Ruby 语法、规则等 Ruby 编程基础知识，以及类、模块等面向对象编程的思考方法和术语。

- **第 3 部分 Ruby 的类**

要编写程序，只懂语法还远远不够。Ruby 之所以能使大家快乐编程，主要缘于 Ruby 精心设计的标准类库。

在本部分，我们会列举多个 Ruby 的基础类，介绍其功能和使用方法。

- **第 4 部分 动手制作工具**

在本部分，我们将进行一次总复习，介绍一些稍微复杂点的 Ruby 程序，让大家尝试一下如何用 Ruby 编写实际的程序。

- **附录 A Ruby 运行环境的构建**

介绍各个平台的 Ruby 安装方法。

- **附录 B 参考**

介绍使用 Ruby 时所需的知识以及各相关信息。

## 0.4 Ruby 的运行环境

本书内容适用于 Ruby 2.0.0 版本，并兼容 Ruby 1.9 系列版本。适用的操作系统为 Windows7/8 和 Mac OS X、Linux 等常见类 Unix 操作系统。

在继续阅读本书前，请读者按照附录 A.1 节的说明，安装 Ruby 运行环境。

## 第 1 部分 Ruby 初体验

---

让我们先从简单的程序入手，从整体上了解 Ruby 的基本概念，并对如何使用 Ruby 编写程序有个初步印象。

# 第 1 章 Ruby 初探

那么，我们赶快来看一下 Ruby 能做些什么。

本章会介绍以下内容。

- 使用 Ruby

了解如何使用 Ruby 编写并执行程序。

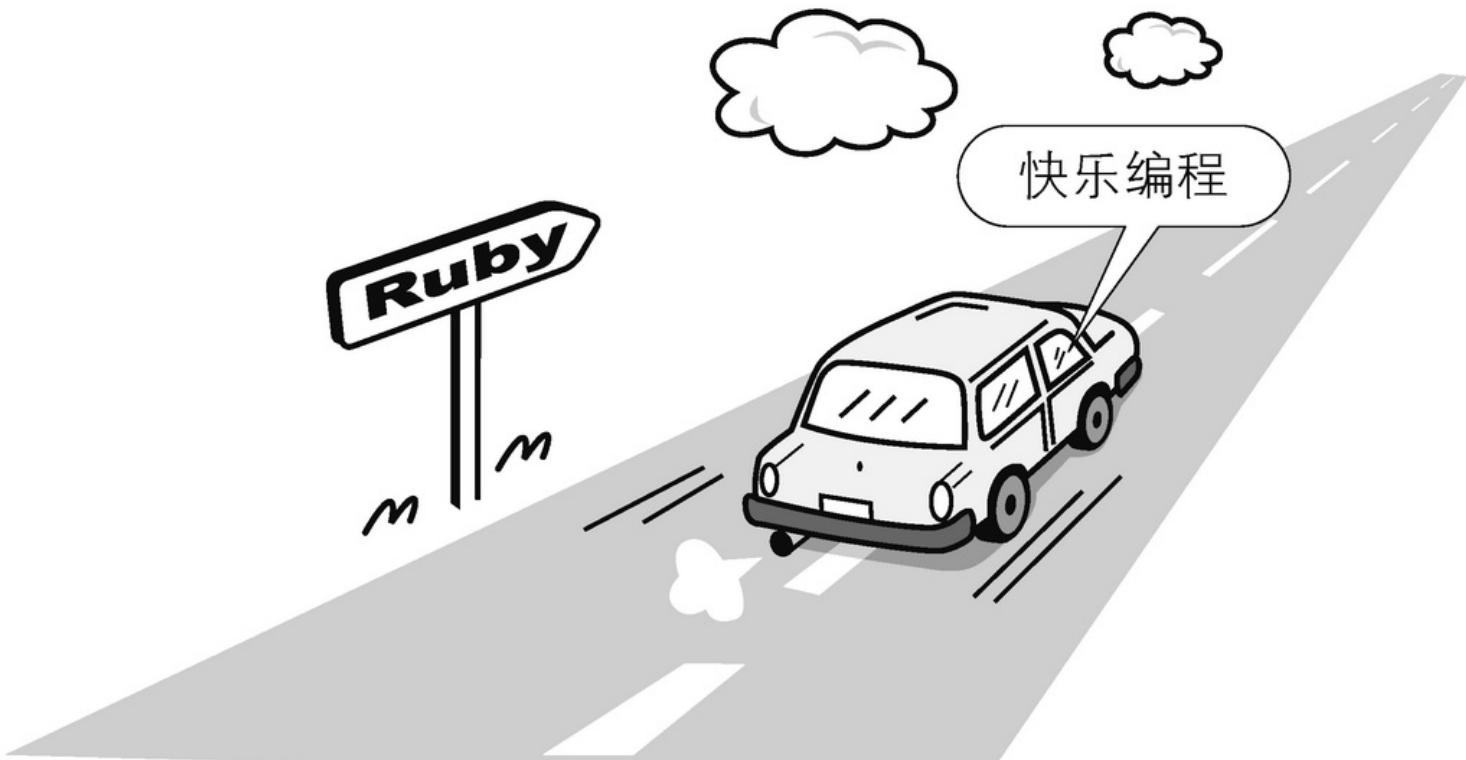
- 使用字符和数值

了解字符和数值是如何输出、计算以及给变量赋值等内容。

- 使用条件判断和循环处理

了解如何通过字符串或数值比较进行条件判断处理，以及如何进行循环处理。

读完本章，大家就能大概掌握用 Ruby 编写程序的方法了。



## 1.1 Ruby 的运行方法

首先，让我们用 Ruby 编写一个在屏幕上输出字符的小程序。

Ruby 程序有多种执行方法，其中最常见的方法是使用 ruby 命令来执行，其次就是使用 irb 命令，以交互式命令行方式来执行。若只是想执行小程序，使用 irb 命令会相对简单一点。

接下来，我们先介绍 ruby 命令以及 irb 命令的使用方法。

另外，读者如果还没安装 Ruby，请参考附录 A 预先安装好 Ruby 运行环境。

**注** 本书适用于 Ruby 2.0 或者 Ruby 1.9。由于 Mac OS X 和 Linux 系统默认安装的 Ruby 版本比较旧，因此请读者安装新版本的 Ruby。

### 1.1.1 ruby 命令的执行方法

首先，让我们看看代码清单 1.1 的程序。

#### 代码清单 1.1 helloruby.rb

```
print("Hello, Ruby.\n")
```

**注** 日文 Windows 系统中的 \ 会显示为 ¥。原则上，本书统一书写为 \。

各位是否有些沮丧呢？一般听到“程序”，我们可能会联想到一长串密码似的东西。但这个程序的代码只有一行，总共才 20 来个字符。可这的确是一个真真切切的程序，执行后就可以达到我们预想的目的。

请大家打开编辑器，写入上述程序，将文件名修改为 helloruby.rb，保存文件。.rb 是 Ruby 程序的后缀，表示这个文件内容是 Ruby 程序。

备注 写代码时会用到编辑器或者 IDE，它们的相关内容请参考 A.5 节。

接下来让我们启动控制台，执行程序。

备注 关于控制台的启动方法，请参考附录 A 里各操作系统的说明。

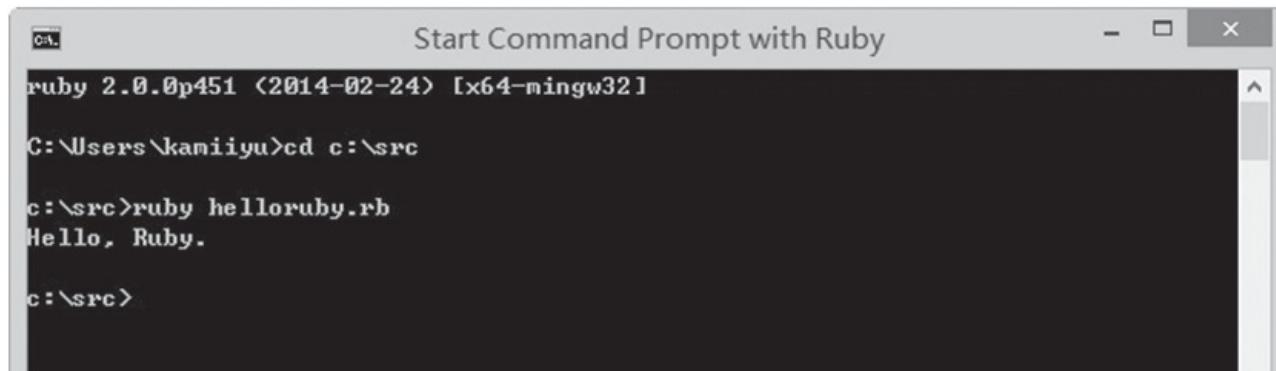
启动控制台后，使用 cd 命令，移动到存放 helloruby.rb 的文件夹中。例如，使用 Windows，文件放在 C 盘的 src 文件夹（c:\src），然后执行以下命令，

```
> cd c:\src
```

接着再执行，

```
> ruby helloruby.rb
```

执行后，如图 1.1 所示，会显示“Hello, Ruby.”。



```
Start Command Prompt with Ruby
ruby 2.0.0p451 (2014-02-24) [x64-mingw32]
C:\Users\kamiiyu>cd c:\src
c:\src>ruby helloruby.rb
Hello, Ruby.
c:\src>
```

图 1.1 执行 Ruby

备注 如果执行时出错，请参考附录 A 以及 B.5 节的内容。

### 1.1.2 irb 命令的执行方法

接下来，我们介绍 irb 命令的执行方法。

与 ruby 命令一样，irb 命令在控制台执行，不过不需要指定程序文件。

执行 irb 命令后，就会出现以下这样的命令提示符。1

1 使用 Ruby Installer for Windows 安装包安装的 Ruby，在执行 irb 命令的时候有可能会显示“DL is deprecated, please use Fiddle”这样的警告。这是由于 Ruby Installer for Windows 安装包附带的 readline 库引用了名为 DL 的旧版本的库，但这个警告不会对使用有任何影响。——译者注

#### 执行示例

```
> irb
irb(main):001:0>
```

在这里，只需把刚才代码清单 1.1 的代码原封不动地在控制台写一次，然后直接按下回车键，即可执行程序。

#### 执行示例

```
irb(main):001:0> print("Hello, Ruby.\n")
Hello, Ruby. ← print 方法输入的结果
=> nil
irb(main):002:0>
```

备注 第三行的 nil 是 print 方法的返回值。关于方法的返回值我们将在 7.3.1 节详细说明。

像这样，在控制台写的程序可以马上在控制台里执行，这对进行简单的小测试非常方便。但是，这个方法并不适合大程序，这时我们应该考虑使用 ruby 命令。

注 在使用 Mac OS X 时，irb 命令会有无法正确输入日语的情况。这时可在 irb 命令后加上 --noreadline 选项，执行 irb --noreadline 命令。这样一来，关闭 readline 功能后，就可以正常输入日语了。但请注意，由于关闭了 readline 功能，在控制台编辑已经输入的字符、查看历史输入记录等功能都将无法使用。

在控制台输入 exit 后，按回车键，或者同时按下 Ctrl + d，都可以终止 irb 命令。

## 1.2 程序解说

接下来，让我们详细解说一下代码清单 1.1 的程序，虽然代码只有孤零零的一行。

### 1.2.1 对象

首先，请留意 "Hello, Ruby.\n" 这部分。

```
print("Hello, RUBY.\n")  
    └─字符串对象
```

这部分被称为 String 对象或者字符串对象，也可以直接称这部分为字符串。也就是说，这部分是一个代表字符串 Hello, Ruby. 的对象（图 1.2）。



图 1.2 数据与对象

字符串、数值、时间等各种数据，在 Ruby 中都是对象。

备注 字符串末尾的 \n 是换行符。

### 1.2.2 方法

这一次，让我们留意一下 `print` 这部分。

```
print("Hello, RUBY.\n")  
    └─字符串对象
```

`print` 是一个方法。所谓方法，就是对象的行为。数值的加法或乘法运算、字符串的合并、某时刻一小时后或者一天后的计算等操作，都是通过执行方法来实现。

`print` 方法的作用就是输出 () 里的内容。因此，执行 `helloworld.rb` 后，在控制台显示了字符串对象——Hello, Ruby.。

我们把执行方法时必需的条件称为参数。例如，我们在说明 `print` 方法时，会说“`print` 方法会把作为参数传递过来的字符串，输出到控制台中”。

我们更换一下 `print` 方法的参数，试一下把它换成其他字符串。

```
print("Hello, RUBY!\\n")
```

这一次，我们希望输出大写字母的 Hello, RUBY!。是不是感觉会更加精神点呢？

## 1.3 字符串

我们再详细看看有关字符串的内容。

### 1.3.1 换行符与 \

上文我们提到过字符 \n 是换行符。利用换行符，我们可以用普通的字符达到换行的效果，例如，为达到以下效果，

```
Hello,  
Ruby  
!
```

程序可以这么写：

```
print("Hello,\nRuby\n!\n")  
    └─换行符
```

原本也可以像下面一样这么写：

```
print("Hello,  
Ruby")
```

```
!
")
```

输出结果虽然与第一种写法是一样的，但是，这样的写法会降低程序的可读性，因此并不是一个好的写法。既然 Ruby 已经帮我们准备了换行符，我们就直接用第一种方法吧。

除了 `\n` 以外，当我们想在字符串里嵌入特殊符号时，也会用到 `\`。双引号是表示字符串开始和结束的符号，假设我们希望字符串里包含双引号，程序要写成 `\\"`。

```
print("Hello, \"Ruby\"\.\n")
```

的输出结果为：

```
Hello, "Ruby".
```

像这样，程序会对字符串里 `\` 后的字符做特殊处理。<sup>2</sup> 因此，如果字符串里需要包含 `\`，程序要写成 `\\"`。例如，

<sup>2</sup>这个过程称为转义，`\` 称为转义字符。——译者注

```
print("Hello \\ Ruby!")
```

的输出结果为：

```
Hello \ Ruby!
```

请注意，两个 `\` 的输出结果是一个 `\`。

### 1.3.2 '' 与 '''

创建字符串对象除了可以使用 `" "`（双引号）外，也可以使用 `' '`（单引号）。我们试试把之前程序的双引号换成单引号，看一下有什么样的效果。

```
print('Hello, \nRuby\n!\\n')
```

这次的输出结果为：

```
Hello, \nRuby\n!\\n
```

程序会原封不动地输出单引号里的内容。

也就是说，在单引号里，像 `\n` 这样的特殊字符不经过转义，程序会原封不动地直接输出。但也有例外，例如在字符串里想嵌入 `\` 与单引号时，还是需要在之前加上 `\`。也就是像这样，

```
print('Hello, \\\\'Ruby\\''.')
```

的输出结果为：

```
Hello, \\'Ruby'.
```

## 1.4 方法的调用

关于方法，我们再详细说明一下。Ruby 在调用方法时可以省略 `()`。因此，代码清单 1.1 的 `print` 方法可以这样写：

```
print "Hello, Ruby.\n"
```

另外，如果想连续输出多个字符串，可以用逗号 `(,)` 分隔各字符串，程序会按顺序输出字符串。因此，如下写法也是可以的：

```
print "Hello, ", "Ruby", ".", "\n"
```

虽然这种写法可以方便地输出多个字符串，但是如果遇到比较复杂的参数，使用 `()` 会更加便于理解。因此，建议在习惯 Ruby 的语法之前，不要使用省略 `()` 的写法。在一些较为简单的情况下，本书会使用省略 `()` 的写法。

一般来说，Ruby 是以从上到下的顺序执行方法的。例如，执行下面的程序会得到相同的结果，也就是 `Hello, Ruby..`。

```
print "Hello, "
print "Ruby"
print "."
print "\n"
```

## 1.5 puts 方法

`puts` 方法与 `print` 方法稍有区别，`puts` 方法在输出结果的末尾一定会输出换行符。用 `puts` 方法时，代码清单 1.1 的程序可改写为这样：

```
puts "Hello, Ruby."
```

不过，当参数为两个字符串时，

```
puts "Hello, ", "Ruby!"
```

各字符串末尾都会加上换行符，因此会输出下面的结果：

```
Hello,  
Ruby!
```

某些情况下，使用 `print` 方法可能会顺手些；而某些情况下，使用 `print` 方法并不方便。请大家按照实际情况选择该使用哪个方法。

## 1.6 p 方法

接下来，我们再介绍一个与输出有关的方法。Ruby 提供了一个更简单的输出对象内容的方法——`p` 方法。

无论使用 `print` 方法还是 `puts` 方法，输出数值 1 和字符串 "1" 时，结果都只是单纯的 1。这样一来，我们就无法判断输出的结果到底是属于哪个对象。这种情况下，`p` 方法可以很好地解决这个问题。使用 `p` 方法时，数值结果和字符串结果会以不同的形式输出。让我们赶快来尝试一下。

```
puts "100"  #=> 100  
puts 100    #=> 100  
p "100"    #=> "100"  
p 100      #=> 100
```

**备注** 本书在表示程序输出内容时，会在方法的旁边添加`#=>`字符，其右侧即为方法的输出结果。在这个例子中，`puts "100"`、`puts 100`、`p 100` 的输出结果为字符串 `100`，`p "100"` 的输出结果为字符串 `"100"`。

像这样，输出结果为字符串时，输出结果会被双引号括起来，一目了然。另外，使用 `p` 方法时，换行符 (`\n`)、制表符 (`\t`) 等特殊字符不会转义，会像下面那样直接输出（代码清单 1.2）。

### 代码清单 1.2 puts\_and\_p.rb

```
puts "Hello, \n\tRuby."  
p "Hello, \n\tRuby."
```

#### 执行示例

```
> ruby puts_and_p.rb  
Hello,  
      Ruby.  
"Hello, \n\tRuby."
```

如果只是需要输出程序执行的结果、信息等，可以选择 `print` 方法；如果想确认程序的执行情况，则可选择 `p` 方法。原则上，`p` 方法是提供给编程者使用的。

## 1.7 中文的输出

到目前为止，我们使用的字符串都只包含字母。

接下来，我们看看如何输出中文字符。其实，输出中文字符也不是多难的事，只要把双引号内的字母换成中文字符即可。像下面这样：

### 代码清单 1.3 kirisubo.rb

```
print "话说某个朝代，后宫妃嫔甚多，\n"  
print "其中有一宫女，出身并不十分高贵，却蒙圣恩宠爱。\\n"
```

#### 执行示例

```
> ruby kirisubo.rb  
话说某个朝代，后宫妃嫔甚多，  
其中有一宫女，出身并不十分高贵，却蒙圣恩宠爱。
```

不过，编码设定不当也会导致输出错误、乱码等情况。遇到这样的情况时，请参考专栏“中文使用注意事项”。

#### 专栏

#### 中文使用注意事项

在某些 Ruby 运行环境里，执行包含中文的脚本时，有可能出现以下错误。

### 执行示例

```
> ruby kiritsubo.rb
kiritsubo.rb:1: invalid multibyte char (US-ASCII)
kiritsubo.rb:1: invalid multibyte char (US-ASCII)
```

这是由于编写程序时没有指定程序的编码方式造成的。Ruby 程序的编码方式，是通过在程序的首行代码添加注释“`# encoding: 编码方式`”来指定的（编码的规则称为 `encoding`）。我们称这个注释为魔法注释（magic comment）。

例如，使用简体中文版 Windows 常用编码 GBK 编写代码时，可像下面这样写魔法注释：

```
# encoding: GBK
print "话说某个朝代，后宫妃嫔甚多，\n"
print "其中有一宫女，出身并不十分高贵，却蒙圣恩宠爱。\n"
```

这样指定程序的编码方式后，Ruby 在执行程序时就可以正确识别程序中的中文。下表是各平台常用的编码方式。平台有多个常用编码方式时，请按照实际情况选择适合的编码方式。

平台	编码方式
Windows	GBK（或者GB 2312）
Mac OS X	UTF-8
Unix	UTF-8

另外，从 Ruby 2.0 开始，若没指定魔法注释，Ruby 会默认使用 UTF-8 编码方式。因此，在 Ruby 2.0 中如果希望代码采用 UTF-8 的编码方式时，可省略魔法注释。

除此以外，使用上述 `p` 方法输出中文时，有时也会出现乱码的情况。这时，可使用“`-E 编码方式`”这个选项来指定输出结果的编码方式。例如，希望以 UTF-8 编码方式在控制台输出结果，可像下面这样执行命令：

### 执行示例

```
> ruby -E UTF-8 脚本文件名 < 脚本执行
> irb -E UTF-8 < irb 启动
```

## 1.8 数值表示与计算

讲解了字符串之后，让我们再来看看 Ruby 是怎么处理数值的。在 Ruby 程序里，整数和小数（浮点数）的处理方式都很自然。

### 1.8.1 数值

首先，让我们先了解一下 Ruby 是如何表示数值的。1.2 节提到，Ruby 中的字符串是以字符串对象的形式存在的。同样地，数值也是以“数值对象”的形式存在的。也就是说，在程序里操作的都是数值对象。

Ruby 的整数的表示方法很简单。直接输入数字就可以了，例如，

```
1
```

表示 1 的整数（`Fixnum`）对象。同样地，

```
100
```

表示 100 的整数对象。

再如，

```
3.1415
```

这表示的是 3.1415 的浮点数（`Float`）对象。

**备注** `Fixnum` 和 `Float` 是对象所属类（class）的名称。类的详细内容我们将会在第 4 章和第 8 章说明。

数值的输出与字符串输出一样，也是用 `print` 方法和 `puts` 方法。

```
puts(10)
```

执行以上代码后，

执行示例

```
10
```

会输出到屏幕中。

### 1.8.2 四则运算

Ruby 还可以对数值进行运算，并输出其结果。我们来看看 Ruby 是怎么进行四则运算的。首先，我们用一下 `irb` 命令。

执行示例

```
> irb --simple-prompt
>> 1 + 1
=> 2 <- 1 + 1 的执行结果
>> 2 - 3
=> -1 <- 2 - 3 的执行结果
>> 5 * 10
=> 50 <- 5 * 10 的执行结果
>> 100 / 4
=> 25 <- 100 / 4 的执行结果
```

备注 `irb` 命令后的选项 `--simple-prompt` 会简化 `irb` 的输出结果。

在一般的编程语言里，乘法运算符用 `*`，除法运算符用 `/`。Ruby 也延续了这个习惯。

让我们再试一下复杂点的四则运算。四则运算的基本原则是“先乘除后加减”，Ruby 也遵循这个原则。也就是说，

```
20 + 8 / 2
```

的结果是 24。如果  $20 + 8$  后再想除 2，可以使用括号，

```
(20 + 8) / 2
```

这时答案为 14。

### 1.8.3 数学相关的函数

除四则运算外，Ruby 中还可以使用数学函数，如平方根、`sin` 和 `cos` 等三角函数、指数函数等。使用数学函数时，必须在函数前加上 `Math.` 标识。

备注 不想在函数前加 `Math.` 时，则需要 `include Math` 这段代码。关于这些用法，我们会在 8.6.1 节进行说明。

求正弦时使用 `sin` 方法，求平方根时使用 `sqrt` 方法。执行相对应的函数方法，即可得到该函数的计算结果。一般我们称这一过程为“执行方法后得到结果”，所得到的结果则称为返回值。

执行示例

```
> irb --simple-prompt
>> Math.sin(3.1415)
=> 9.26535896604902e-05 <- sin 方法的返回值
>> Math.sqrt(10000)
=> 100.0 <- sqrt 方法的返回值
```

注 不同的 Ruby 版本，或者在不同平台下执行时，返回值的位数可能会不同。

第一个返回值的结果是 `9.26535896604902e-05`，这是一种用来表示极大数或极小数的方法。“`(小数) e (整数)`”表示“(小数)  $\times 10^{\text{整数}}$  次幂”。这个例子中，其结果值为“ $9.26535896604902 \times 10^{-5}$  次幂”，也就是 `0.0000926535896604902`。

## 1.9 变量

变量是程序里不可缺少的元素之一。可以将其理解为给对象贴上的标签。

我们可以像下面这样给对象贴上标签（图 1.3）：

变量名 = 对象

我们称这个过程为“将对象赋值给变量”。

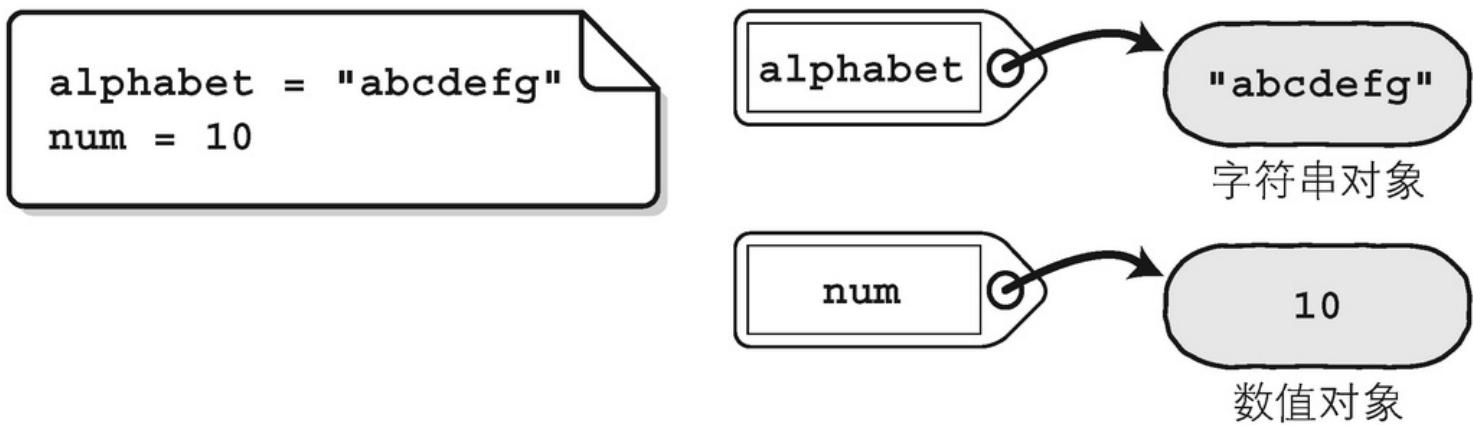


图 1.3 变量和对象

```
alphabet = "abcdefg"
num = 10
age = 18
name = 'TAKAHASHI'
```

为了说明如何使用变量，让我们看看以下这个求长方体表面积和体积的例子（代码清单 1.4）。

代码清单 1.4 area\_volume.rb

```
x = 10
y = 20
z = 30
area = (x*y + y*z + z*x) * 2
volume = x * y * z
print "表面积=", area, "\n"
print "体积=", volume, "\n"
```

若不使用变量，则程序会像下面这样：

```
print "表面积=", (10*20 + 20*30 + 30*10) * 2, "\n"
print "体积=", 10*20*30, "\n"
```

这样一来，一旦要修改一个值，那么好几个地方也必须一起修改。上例只有两行代码，修改起来并不太麻烦，但遇到比较复杂的程序时，修改起来就会非常费劲。

另外，变量还可以清晰地表示某个值所代表的含义。因此，为变量起一个容易理解的名称是非常重要的。例如，

```
hoge = (foo*bar + bar*baz + baz*foo) * 2
funi = foo * bar * baz
```

像这样的代码，使人完全搞不清楚这个程序的目的。所以，建议大家平时多加注意，最好使用 `area`、`volume` 等意义明确的单词作为变量名。

## print方法和变量

让我们再回顾一下 `print` 方法。

```
print "表面积 =", area, "\n"
```

这个 `print` 方法有三个参数：“表面积 =”、`area`、`"\n"`。`print` 方法会按照顺序，输出这三个参数。

“表面积 =” 表示值为“表面积 =”的字符串，因此 `print` 方法会直接输出。`area` 表示 `area` 这个变量引用的对象，在这个例子里就是 `2200` 这个整数，因此 `print` 方法输出的是该整数值。

最后那个 `"\n"` 是换行符，因此 `print` 方法也会直接输出。

这三个值经过 `print` 方法处理后，会在屏幕输出“表面积 = 2200”加换行的结果。

传递给 `print` 方法的字符串参数也可以像下面这样写：

```
print "表面积 = #{area}\n"
```

`"表面积 = #{area}\n"` 为一个整体的字符串。`#{area}` 这样的写法，表示把 `area` 的值嵌入到字符串中。在字符串里使用 `#{...}` 这样的写法，可以把通过计算后得到的值嵌入到字符串中。输出结果里除了可以嵌入变量名，也可以嵌入 `"表面积 = #{{(x*y + y*z + z*x) * 2}}\n"` 这样的计算公式，得到的输出结果

是一样的。

一般向屏幕输出结果时，我们都希望同时输出换行符，因此使用 `puts` 方法时，连 `\n` 也都不需要了，这样程序会变得更加简洁。

```
puts "表面积 = #{area}"
```

## 1.10 注释

我们在程序里可以写注释。注释虽然写在程序里面，但程序并不会执行注释的内容，也就是说，注释的内容对程序的执行结果不会产生任何影响。

大家也许会问：“为什么会在程序里写这种与程序运行无关的东西呢？”的确，对于只执行一次就可以的简单程序，并不需要特别的注释。但是，一般来说，我们都会多次使用写好的程序。那么，若希望记录

- 程序的名称、作者、发布条件等信息
- 程序说明

等内容时，就需要用到注释。

Ruby 用`#`表示注释的开始 3。某行是以`#`开头时，则整行都是注释。某行中间出现`#`时，则`#`以后部分就都是注释。另外，行的开头用`=begin`和`=end`括起来的部分也是注释 4。这样的注释方法，在程序开头或结尾写长说明时会经常用到。

3单行注释。——译者注

4多行注释。——译者注

代码清单 1.5 `comment_sample.rb`

```
=begin
《Ruby 基础教程（第 4 版）》示例
注释的使用示例
2006/06/16 创建
2006/07/01 追加一部分注释
2013/04/01 第 4 版更新
=end
x = 10 # 宽
y = 20 # 长
z = 30 # 高
# 计算表面积和体积
area = (x*y + y*z + z*x) * 2
volume = x * y * z
# 输出
print "表面积=", area, "\n"
print "体积=", volume, "\n"
```

除此以外，注释还有使某行代码“暂时不执行”的作用。

Ruby 没有 C 中使行的某部分成为注释的写法，只要是以`#`开始的部分，到行末为止一定都是注释。

## 1.11 控制语句

编程语言中都有控制语句。

控制语句能让程序在某种条件下，改变执行顺序，或者只执行某一部分。

### 控制语句的分类

控制语句大致可以分成以下几类。

- **顺序控制：**按照程序的编写顺序，从头到尾执行。
- **条件控制：**若某条件成立，则执行`○○`，否则执行`××`。
- **循环控制：**在某条件成立之前，反复执行`○○`。
- **异常控制：**发生某种异常时，执行`○○`。

顺序控制是程序最常见的处理方式。若不做特殊处理，程序会按照代码的编写顺序执行。

条件控制，是指根据条件执行分支处理。如果没有满足条件，程序会跳过某部分处理，继续执行其他处理。在 Ruby 中，可使用的条件判断语句有`if`、`unless`、`case` 等。

循环控制，是指根据条件反复执行某个处理。在这种情况下，该处理的执行顺序会与程序编写的顺序不同，执行过一次的程序，会从头再执行一次。

异常控制有点特殊。程序执行时，意料之外的错误发生后，就会跳出正在执行的那部分程序，然后执行其他地方的程序，使程序能继续执行下去。根据实际

情况，有时也会让程序马上结束。

接下来，我们进一步介绍条件控制和循环控制。

## 1.12 条件判断：if～then～end

`if` 语句用于根据条件变化，改变程序的行为。`if` 语句的语法如下所示：

`if 条件 then`

    条件成立时执行的处理

`end`

条件一般是指能返回 `true` 或者 `false` 的表达式。例如比较两个值，相同则返回 `true`，不同则返回 `false`，这样的表达式可作为条件。

我们在比较数值的大小时，会用到等号、不等号等运算符。在 Ruby 中，`=` 已经被用作赋值运算了，因此判断是否相等的运算符要用两个并列等号`==` 来代替`=`。另外，`<` 和 `>` 运算符在 Ruby 中分别用 `<=` 和 `>=` 来替代。

条件表达返回的结果为 `true` 或者 `false`，`true` 表示条件成立，`false` 表示条件不成立。

```
p (2 == 2)    #=> true
p (1 == 2)    #=> false
p (3 > 1)    #=> true
p (3 > 3)    #=> false
p (3 >= 3)   #=> true
p (3 < 1)    #=> false
p (3 < 3)    #=> false
p (3 <= 3)   #=> true
```

我们也可以使用`==` 运算符比较字符串。字符串内容相同则返回 `true`，内容不同则返回 `false`。

```
p ("Ruby" == "Ruby")    #=> true
p ("Ruby" == "Rubens")  #=> false
```

判断值不相等时要使用`!=` 运算符，它与`#`意思相同。

```
p ("Ruby != "Rubens")  #=> true
p (1 != 1)              #=> false
```

那么，接下来让我们来看一下如何使用这些运算符编写条件语句。代码清单 1.6 是一个简单的程序：变量 `a` 大于等于 10 时输出 `bigger`，小于 9 时输出 `smaller`。

代码清单 1.6 bigger\_smaller.rb

```
a = 20
if a >= 10 then
  print "bigger\n"
end
if a <= 9 then
  print "smaller\n"
end
```

在这里可以省略 `then` 关键字。

```
if a >= 10
  print "bigger\n"
end
|
```

如果希望对条件成立和条件不成立时采取不同的处理，我们可以使用 `else` 关键字。

`if 条件 then`

    条件成立时执行的处理

`else`

    条件不成立时执行的处理

`end`

使用 `else` 关键字改写刚才的程序后，会变成下面这样，

```
if a >= 10
  print "bigger\n"
else
  print "smaller\n"
```

```
end
```

## 1.13 循环

有时，我们会遇到希望多次循环执行同样的处理的情况。下面，我们来介绍两种执行循环处理的方法。

### 1.13.1 while 语句

`while` 语句是执行循序时用到的一种基本语句。同样地，`do` 关键字可以省略。

`while` 循环条件 `do`

希望循环的处理

```
end
```

- 例：按顺序输出从 1 到 10 十个数字

```
i = 1
while i <= 10
  print i, "\n"
  i = i + 1
end
```

### 1.13.2 times 方法

循环处理的循环次数如果已确定，使用 `times` 方法会更加简单。

循环次数 `.times do`

希望循环的处理

```
end
```

- 输出 100 行“All work and no play makes Jack a dull boy.”

```
100.times do
  print "All work and no play makes Jack a dull boy.\n"
end
```

`times` 方法被称为迭代器（iterator）。迭代器是 Ruby 的一个特色功能。从迭代器的英语拼写方法我们可以知道，迭代器表示的是循环（iterate）的容器（-or）。类似地，运算符（operator）也就是运算（operate）的容器（-or），等等。总之，迭代器就是指用于执行循环处理的方法。

Ruby 除了 `times` 方法外，还提供了很多迭代器，典型的有 `each` 方法。`each` 方法的相关内容，我们会在第 2 章中与数组、散列一起介绍。

## 第 2 章 便利的对象

在第 1 章里，我们介绍了 Ruby 的基本数据对象“字符串”以及“数值”。当然，Ruby 能使用的数据对象不可能只有这两种。一般 Ruby 程序里数据对象的结构都比它们复杂得多。

假设现在我们用 Ruby 做一个通讯录，通讯录一般有以下项目：

- 名字
- 拼音
- 邮政编码
- 都道府县<sup>1</sup>
- 地址
- 电话号码
- 邮箱地址
- SNS 的 URL
- 登记日期

<sup>1</sup>日本的行政区域单位——译者注

在这些项目里，邮政编码用 7 位的数字表示，除此以外的项目都是用字符串表示（一般来说，登记日期这个项目应该用 Date 对象来表示。关于 Date 对象我们将会在第 20 章介绍）。

这样一来，一组的项目集合起来后就可以表示一个人的基本信息（图 2.1），再把亲朋好友的基本信息都收集后就成为一本通讯录。

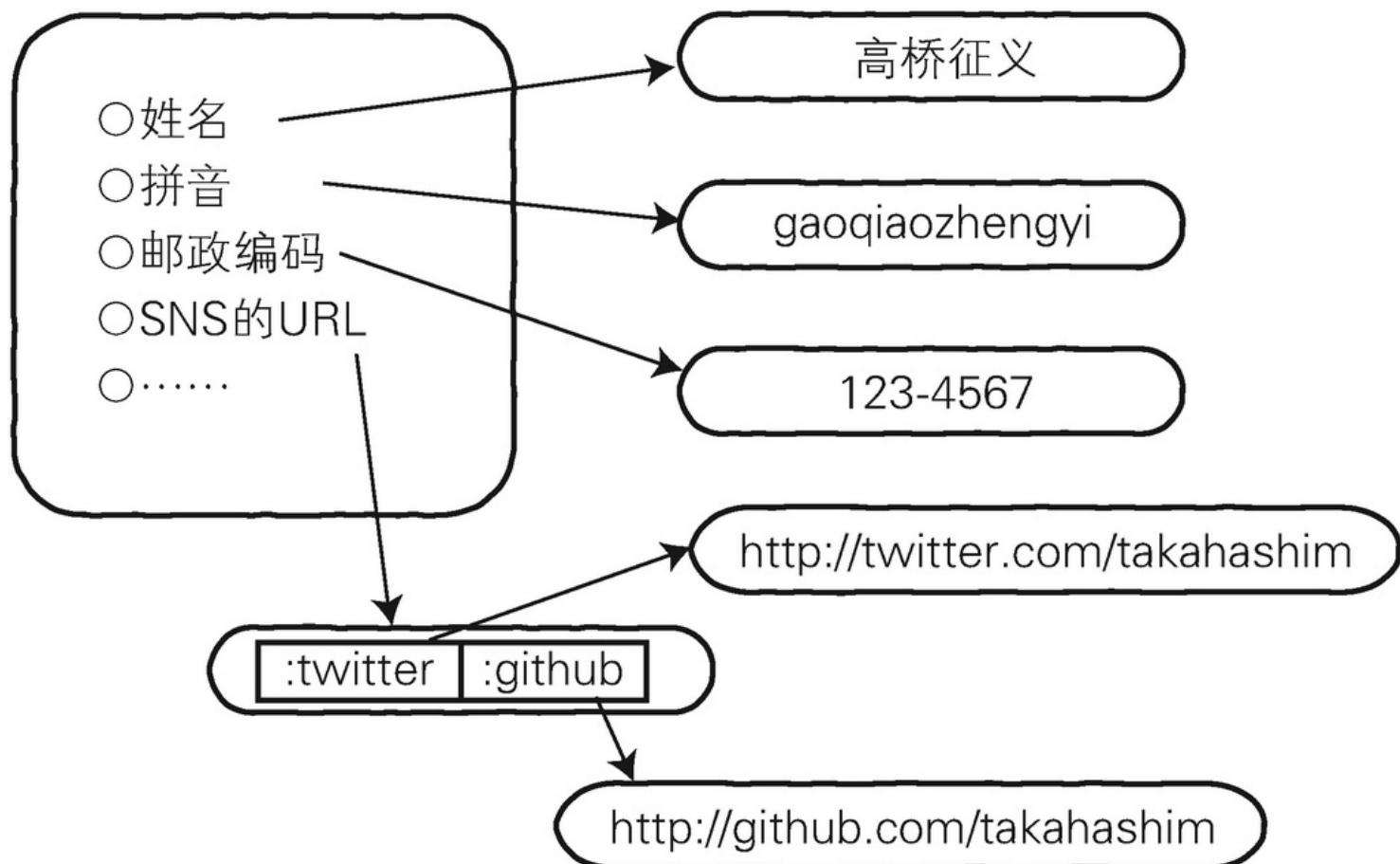


图 2.1 收集、汇总各项目

不同数据间的组合无法用字符串或数值这样简单的对象来表示，因此我们需要一个可以用以表示数据集合的数据结构。

本章我们将介绍数组和散列这两种新的数据结构。另外，我们还会介绍处理字符串时常用的工具——正则表达式。

**备注** 像数组、散列这样保存对象的对象，我们称为容器（container）。

数组、散列、正则表达式的应用十分广泛，关于它们的详细用法我们会在后面的章节介绍，在这里我们只会简略地说明一下，让大家对它们有个初步印象。

## 2.1 数组

数组（array）是一个按顺序保存多个对象的对象，它是基本的容器之一。我们一般称为数组对象或者 Array 对象。

### 2.1.1 数组的创建

要创建数组，我们需要把各数组的元素用逗号隔开，然后再用 [] 把它们括起来即可。首先，让我们创建一个简单的数组。

```
names = ["小林", "林", "高野", "森冈"]
```

在这个例子里，我们创建了一个叫 names 的数组对象，分别保存了 4 个数组元素：小林、林、高野、森冈。如图 2.2 所示。

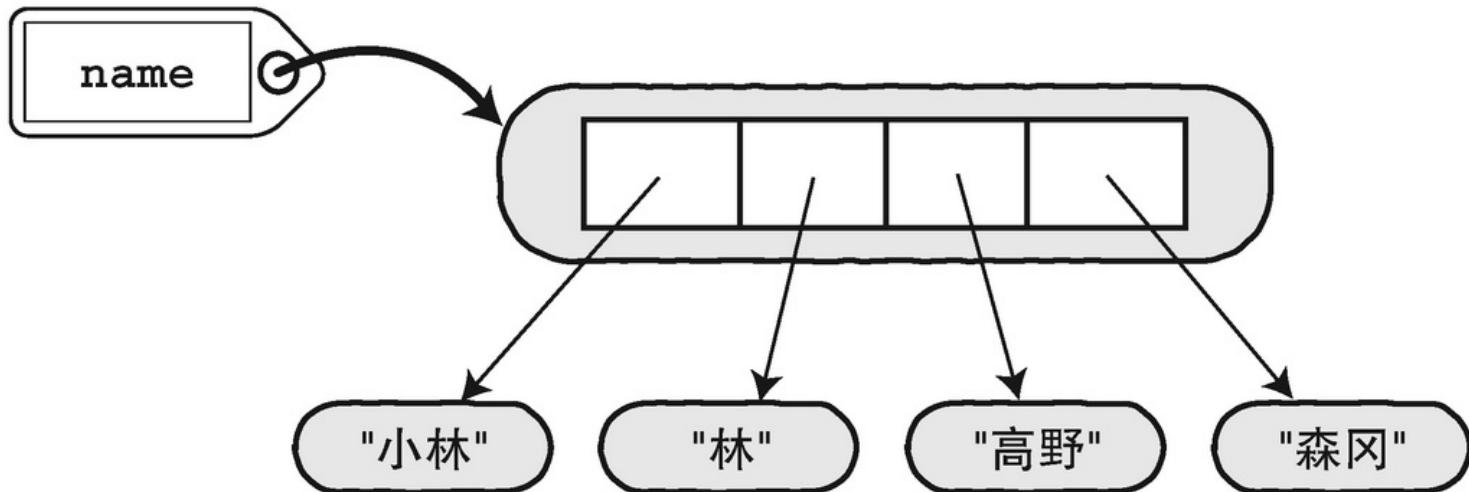


图 2.2 数组对象

### 2.1.2 数组对象

在数组元素对象还未确定的情况下，我们可以用 [] 表示一个空数组对象。

```
names = []
```

除此以外，还有其他方法创建数组，我们将会在第 13 章再详细说明。

### 2.1.3 从数组中抽取对象

保存在数组里的每个对象，都各自有一个表示其位置的编号，我们称之为索引（index）。利用索引，我们可以进行把对象保存到数组、从数组中抽取对象等操作。

要从数组中抽取元素（对象），我们可以使用以下方法。

#### 数组名 [ 索引 ]

如下所示，有一个名为 names 的数组对象。

```
names = ["小林", "林", "高野", "森冈"]
```

把 names 数组里第一个元素拿出来，我们可以这么做

```
names[0]
```

因此，若执行以下代码，

```
print "第一个名字为: ", names[0], ".\n"
```

得到的结果为，

```
第一个名字为小林。
```

同样地，names[1] 表示林，names[2] 表示高野。

#### 执行示例

```
> irb --simple-prompt  
=> ["小林", "林", "高野", "森冈"]
```

```
>> names[0]
=> "小林"
>> names[1]
=> "林"
>> names[2]
=> "高野"
>> names[3]
=> "森冈"
```

备注 数组的索引值是从 0 开始，并非 1。因此，`a[1]` 返回的并不是数组第一个元素，而是第二个元素。刚接触编程时，大家比较容易弄错（即便是熟悉以后也有可能会犯这样的错误）。大家在使用数组时，务必注意索引值这个特点。

注 在 Windows 命令行中，用 Alt + Shift 键切换中英文输入法模式。

#### 2.1.4 将对象保存到数组中

我们可以将新的对象保存到已经创建的数组中。

将数组里的某个元素置换为其他对象，我们可以这样做：

**数组名 [ 索引 ] = 希望保存的对象**

我们试着置换一下刚才的 `names` 数组的内容，将 "野尻" 放在数组首位。

```
names[0] = "野尻"
```

执行下面的程序，我们可以知道 "野尻" 的确已经被置换为首位的数组元素。

执行示例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森冈"]
=> ["小林", "林", "高野", "森冈"]
>> names[0] = "野尻"
=> "野尻"
>> names
=> ["野尻", "林", "高野", "森冈"]
```

在保存对象时，如果指定了数组中不存在的索引值时，则数组的大小会随之而改变。Ruby 数组的大小是按实际情况自动调整的 2。<sup>2</sup>

2即动态数组。——译者注

执行示例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森冈"]
=> ["小林", "林", "高野", "森冈"]
>> names[4] = "野尻"
=> "野尻"
>> names
=> ["小林", "林", "高野", "森冈", "野尻"]
```

#### 2.1.5 数组的元素

任何对象都可以作为数组元素保存到数组中。例如，我们除了可以创建字符数组，还可以创建数值数组。

```
num = [3, 1, 4, 1, 5, 9, 2, 6, 5]
```

Ruby 数组还支持多种不同对象的混合保存。

```
mixed = [1, "歌", 2, "风", 3]
```

这里，我们不再举其他例子了，像时间、文件等对象也都可以作为数组元素。

#### 2.1.6 数组的大小

我们可以用 `size` 方法获知数组的大小。例如，若想获知数组 `array` 的大小，程序可以这么写：

```
array.size
```

我们现在就用 `size` 方法，查看一下刚才的 `names` 数组的大小。

## 执行示例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森冈"]
=> ["小林", "林", "高野", "森冈"]
>> names.size
=> 4
```

size 方法的返回值就是数组的大小。

### 2.1.7 数组的循环

有时，我们希望输出所有数组元素，或者对在数组中符合某条件的元素执行 xx 方法，不符合条件的执行 yy 方法。为实现这些目的，我们需要一种方法遍历所有数组元素。

为此，Ruby 提供了 `each` 方法。我们在第 1 章介绍迭代器时，已经稍微接触了一下 `each` 方法。

`each` 方法的语法如下：

数组 `.each do |变量|`

希望循环的处理

`end`

`each` 后面在 `do ~ end` 之间的部分称为块（block）<sup>3</sup>。因此，`each` 这样的方法也可以称为带块的方法。我们可以把多个需要处理的内容合并后写到块里面。

3也称为代码块。——译者注

块的开始部分为 `| 变量 |`。`each` 方法会把数组元素逐个拿出来，赋值给指定的 `| 变量 |`，那么块里面的方法就可以通过访问该变量，实现循环遍历数组的操作。

接下来，我们实际操作一下，按顺序输出数组 `names` 的元素。

## 执行示例

```
> irb --simple-prompt
>> names = ["小林", "林", "高野", "森冈"]
=> ["小林", "林", "高野", "森冈"]
>> names.each do |n|
?>   puts n
>> end
小林
林
高野
森冈
=> ["小林", "林", "高野", "森冈"] ]
```

像 “`do~end`” 这样跨多行的代码，输入 `end` 之前的代码是不会执行的。

puts 方法的执行结果

each 方法的返回值

每循环一次，就会把当前的数组元素赋值给变量 `|n|`（图 2.3）。



图 2.3 循环时 `n` 的变化

除了 `each` 方法外，数组还提供了许多带块的方法，我们在实际的数组操作中会经常使用到，详细的内容会在 13.6 节介绍。

## 2.2 散列

散列（hash）也是一个程序里常用到的容器。散列是键值对（key-value pair）的一种数据结构。在 Ruby 中，一般是以字符串或者符号（Symbol）作为键，来保存对应的对象（图 2.4）。

```
address = {name: "高桥", pinyin: "gaoqiao",
           :postal => "1234567"}
```

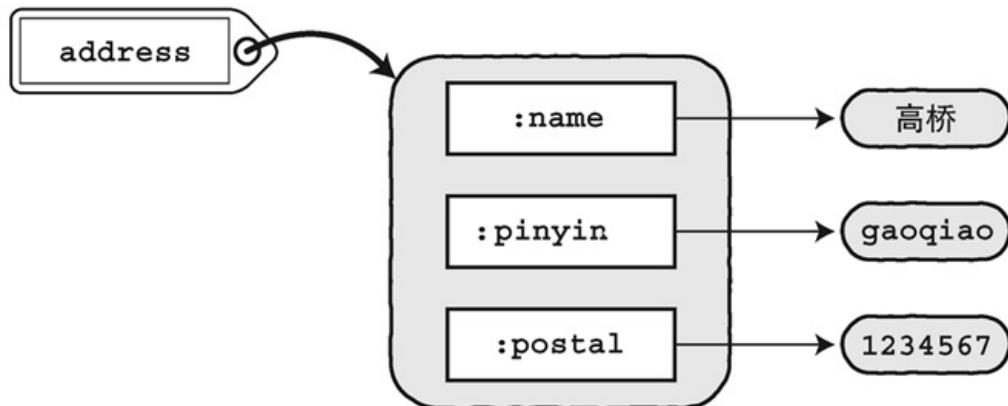


图 2.4 散列

### 2.2.1 什么是符号

在 Ruby 中，符号（symbol）与字符串对象很相似<sup>4</sup>，符号也是对象，一般作为名称标签来使用，用来表示方法等的对象的名称。

<sup>4</sup>可以将符号简单理解为轻量级的字符串。——译者注

创建符号，只需在标识符的开头加上冒号就可以了。

```
sym = :foo      # 表示符号":foo"
sym2 = :"foo"    # 意思同上
```

符号能实现的功能，大部分字符串也能实现。但像散列键这样只是单纯判断“是否相等”的处理中，使用符号会比字符串比较更加有效率，因此在实际编程中我们也会时常用到符号。

另外，符号与字符串可以互相任意转换。对符号使用 `to_s` 方法，则可以得到对应的字符串。反之，对字符串使用 `to_sym` 方法，则可以得到对应的符号。

#### 执行示例

```
> irb --simple-prompt
>> sym = :foo
=> :foo
>> sym.to_s      # 将符号转换为字符串
=> "foo"
>> "foo".to_sym    # 将字符串转换为符号
=> :foo
```

### 2.2.2 散列的创建

创建散列的方法与创建数组的差不多，不同的是，不使用 `[]`，而是使用 `{}` 把创建的内容括起来。散列用 `=>` 来定义获取对象时所需的键（key），以及键相对应的对象（value）。

```
address = {name: "高桥", pinyin: "gaoqiao", postal: "1234567"}
```

将符号当作键来使用时，程序还可以像下面这么写：

```
address = {name: "高桥", pinyin: "gaoqiao", postal: "1234567"}
```

### 2.2.3 散列的使用

从散列取出对象、将对象保存到散列的使用方法也都和数组非常相似。我们使用以下方法从散列里取出对象。

#### 散列名 [ 键 ]

保存对象时使用以下方法。

散列名[键] = 希望保存的对象

#### 执行示例

```
> irb --simple-prompt
>> address = {name: "高桥", pinyin: "gaoqiao"}
```

```
=> {:name=>"高桥", :pinyin=>"gaoqiao"}  
>> address[:name]  
=> "高桥"  
>> address[:pinyin]  
=> "gaoqiao"  
>> address[:tel] = "000-1234-5678"  
=> "000-1234-5678"  
>> address  
=> {:name=>"高桥", :pinyin=>"gaoqiao", :tel=>"000-1234-5678"}
```

## 2.2.4 散列的循环

使用 `each` 方法，我们可以遍历散列里的所有元素，逐个取出其元素的键和对应的值。循环数组时是按索引顺序遍历元素，循环散列时按照键值组合遍历元素。

散列的 `each` 语法如下。

```
散列 .each do |键变量, 值变量|  
  希望循环的处理  
end
```

事不宜迟，我们马上来看看怎么用。

### 执行示例

```
> irb --simple-prompt  
>> address = {name: "高桥", pinyin: "gaoqiao"}  
=> {:name=>"高桥", :pinyin=>"gaoqiao"}  
>> address.each do |key, value|  
?>   puts "#{key}: #{value}"  
>> end  
name: 高桥  
pinyin: gaoqiao  
=> {:name=>"高桥", :pinyin=>"gaoqiao"}
```

显而易见，程序循环执行了输出散列 `address` 的键和值的 `puts` 方法 5。<sup>5</sup>

5原文是 `print` 方法。——译者注

## 2.3 正则表达式

在 Ruby 中处理字符串时，我们常常会用到正则表达式（regular expression）。使用正则表达式，可以非常简单地实现以下功能：

- 将字符串与模式（pattern）相匹配
- 使用模式分割字符串

Ruby 的前辈——Perl、Python 等脚本语言至今还在使用正则表达式。Ruby 继承了这一点，把正则表达式的使用嵌入到语法中，大大简化了正则表达式的调用方式。正是在正则表达式的帮助下，字符串处理变成了一个 Ruby 非常擅长的领域。

### 模式与匹配

我们有时会有按照特定模式进行字符串处理的需求，比如“找出包含○○字符串的行”或者“抽取○○和 ×× 之间的字符串”。判断字符串是否适用于某模式的过程称为匹配，如果字符串适用于该模式则称为匹配成功（图 2.5）。



图 2.5 匹配的例子

像这样的字符串模式就是所谓的正则表达式。

乍一看，“正则表达式”这个词可能会给人一种深奥、难理解的印象。的确，正则表达式非常复杂，但如果只是使用单纯的匹配功能，也并不怎么难。所以大家也无需感到如临大敌，我们暂时只需要知道有个工具叫“正则表达式”就足够了。

创建正则表达式对象的语法如下所示。

/ 模式 /

例如，我们希望匹配 Ruby 字符串的正则表达式为：

```
/Ruby/
```

把希望匹配的内容直接写出来，就这么简单。匹配字母、数字时，模式按字符串原样写就可以了。6

6汉字也可以通过同样的方法做匹配。——译者注

我们用运算符 =~ 来匹配正则表达式和字符串。它与判断是否为同一个对象时用到的 == 有点像。

匹配正则表达式与字符串的方法是：

/ 模式 / =~ 希望匹配的字符串

若匹配成功则返回匹配部分的位置。字符的位置和数组的索引一样，是从 0 开始计数的。也就是说，字符串的首个字符位置为 0。反之，若匹配失败，则返回 nil。

#### 执行示例

```
> irb --simple-prompt
>> /Ruby/ =~ "Ruby"
=> 0
>> /Ruby/ =~ "Diamond"
=> nil
```

之前曾提到过，使用单纯的字母、数字、汉字模式时，如果字符串里存在该模式则匹配成功，否则匹配失败。

#### 执行示例

```
> irb --simple-prompt
>> /Ruby/ =~ "Yet Another Ruby Hacker,"
=> 12
>> /Yet Another Ruby Hacker,/ =~ "Ruby"
=> nil
```

正则表达式右边的 / 后面加上 i 表示不区分大小写匹配。

#### 执行示例

```
> irb --simple-prompt
>> /Ruby/ =~ "ruby"
=> nil
>> /Ruby/ =~ "RUBY"
=> nil
>> /Ruby/i =~ "ruby"
=> 0
>> /Ruby/i =~ "RUBY"
=> 0
>> /Ruby/i =~ "rUbY"
=> 0
```

除此以外，正则表达式还有很多写法和用法，更详细内容将会在第 16 章介绍。

#### 专栏

##### nil 是什么

nil 是一个特殊的值，表示对象不存在。像在正则表达式中表示无法匹配成功一样，方法不能返回有意义的值时就会返回 nil。另外，从数组或者散列里获取对象时，若指定不存在的索引或者键，则得到的返回值也是 nil。

#### 执行示例

```
> irb --simple-prompt
>> hash = {"a"=>1, "b"=>2}
=> {"a"=>1, "b"=>2}
>> hash["c"]
=> nil
```

if 语句和 while 语句在判断条件时，如果碰到 false 和 nil，则会认为是“假”，除此以外的都认为是“真”。因此，除了可以使用返回 true 或者 false 的方法，也可以使用“返回某个值”或者返回“nil”的方法作为判断条件表达式。

```
names = ["小林", "林", "高野", "森冈"]
["小林", "林", "高野", "森冈"]
names.each do |name|
  if /林/ =~ name
    puts name
  end
end
```

### 执行示例

```
> ruby print_hayasi.rb
小林
林
```

# 第3章 创建命令

本章介绍 Ruby 从命令行读取并处理数据的方法。另外，作为第1部分的总结，让我们来实际体验一下如何用 Ruby 实现 Unix 的 grep 命令，以便大家了解用 Ruby 编写程序的大概流程。

## 3.1 命令行的输入数据

到目前为止，我们写的程序都是向屏幕输出数据。现在我们考虑一下怎么输入数据。在创建命令前，我们首先得知道怎么使用命令。那么，让我们先来看看怎么把数据传递给程序。

向程序传递数据，最简单的方法就是使用命令行。Ruby 程序中，使用 `ARGV` 这个 Ruby 预定义好的数组来获取从命令行传递过来的数据。数组 `ARGV` 中的元素，就是在命令行中指定的脚本字符串参数。

在命令行指定多个脚本参数时，各参数之间用空格间隔。

代码清单 3.1 `print_argv.rb`

```
puts "首个参数: #{ARGV[0]}"
puts "第2个参数: #{ARGV[1]}"
puts "第3个参数: #{ARGV[2]}
```

执行示例

```
> ruby print_argv.rb 1st 2nd 3rd
首个参数: 1st
第2个参数: 2nd
第3个参数: 3rd
```

使用数组 `ARGV` 后，程序需要用到的数据就不必都写在代码中。同时，抽取数据、保存数据等普通的数组操作对于 `ARGV` 都是适用的。

代码清单 3.2 `happy_birth.rb`

```
name = ARGV[0]
print "Happy Birthday, ", name, "!\n"
```

执行示例

```
> ruby happy_birth.rb Ruby
Happy Birthday, Ruby!
```

从参数里得到的数据都是字符串，因此如果希望进行运算时，需要对获得的数据进行类型转换。把字符串转换为整数，我们可以使用 `to_i` 方法。

代码清单 3.3 `arg_arith.rb`

```
num0 = ARGV[0].to_i
num1 = ARGV[1].to_i
puts "#{num0} + #{num1} = #{num0 + num1}"
puts "#{num0} - #{num1} = #{num0 - num1}"
puts "#{num0} * #{num1} = #{num0 * num1}"
puts "#{num0} / #{num1} = #{num0 / num1}"
```

执行示例

```
> ruby arg_arith.rb 5 3
5 + 3 = 8
5 - 3 = 2
5 * 3 = 15
5 / 3 = 1
```

## 3.2 文件的读取

Ruby 脚本除了读取命令行传递过来的字符串参数外，还可以读取预先写在文件里的数据。

Ruby 的源代码中，有一个名为 `ChangeLog` 的文本文件。文件里面记录了 Ruby 相关的修改日志。

文件内容如下所示：

```
Mon Feb 27 23:46:09 2012 Yukihiro Matsumoto <matz@ruby-lang.org>
```

备注 Ruby 的源代码可以从 Ruby 官网下载。ChangeLog 文件可以在 Ruby 的 github 库里找到。

- Ruby 源码下载地址: [https://www.ruby-lang.org/zh\\_cn/downloads/](https://www.ruby-lang.org/zh_cn/downloads/)
- github 上的 ChangeLog 文件地址: [https://raw.github.com/ruby/ruby/v2\\_0\\_0\\_0/ChangeLog](https://raw.github.com/ruby/ruby/v2_0_0_0/ChangeLog)

我们就利用这个文件，练习一下用 Ruby 如何进行文件操作。

### 3.2.1 从文件中读取内容并输出

首先，我们先做一个简单文件内容读取程序。读取文件内容的流程，如下所示：

① 打开文件。

② 读取文件的文本数据。

③ 输出文件的文本数据。

④ 关闭文件。

程序代码如下：

代码清单 3.4 `read_text.rb`

```
1: filename = ARGV[0]
2: file = File.open(filename) # ①
3: text = file.read          # ②
4: print text              # ③
5: file.close               # ④
```

与之前的例子相比，这个例子的代码终于有点程序的模样了，接下来我们逐行分析。

第 1 行，将命令行参数 `ARGV[0]` 赋值给变量 `filename`。也就是说，`filename` 表示我们希望读取的文件名。第 2 行，`File.open(filename)` 表示打开名为 `filename` 的文件，并返回读取该文件所需的对象。可能会有读者不太明白什么是“读取该文件所需的对象”，不过不要紧，目前我们暂时只需要知道有这么一个对象就可以了。我们会在第 17 章中详细说明这个对象。

“读取该文件所需的对象”实际在第 3 行使用。在这里，`read` 方法读取文本数据，并将读取到的数据赋值给 `text` 变量。接下来，第 4 行的代码会输出 `text` 的文本数据。到目前为止，我们使用过好多次 `print` 方法了，大家应该不会陌生了吧。然后，程序执行最后一段代码的 `close` 方法。这样，就可以关闭之前打开了的文件了。

像下面那样执行这个程序后，指定的文件内容会一下子全部输出到屏幕中。

> ruby `read_text.rb` 文件名

其实，如果只是读取文件内容，直接使用 `read` 方法会使程序更简单。

代码清单 3.5 `read_text_simple.rb`

```
1: filename = ARGV[0]
2: text = File.read(filename)
3: print text
```

关于 `File.read` 方法的详细用法，我们会在第 17 章进行说明。

更进一步，如果不使用变量，一行代码就可以搞定了。

代码清单 3.6 `read_text_oneline.rb`

```
1: print File.read(ARGV[0])
```

### 3.2.2 从文件中逐行读取内容并输出

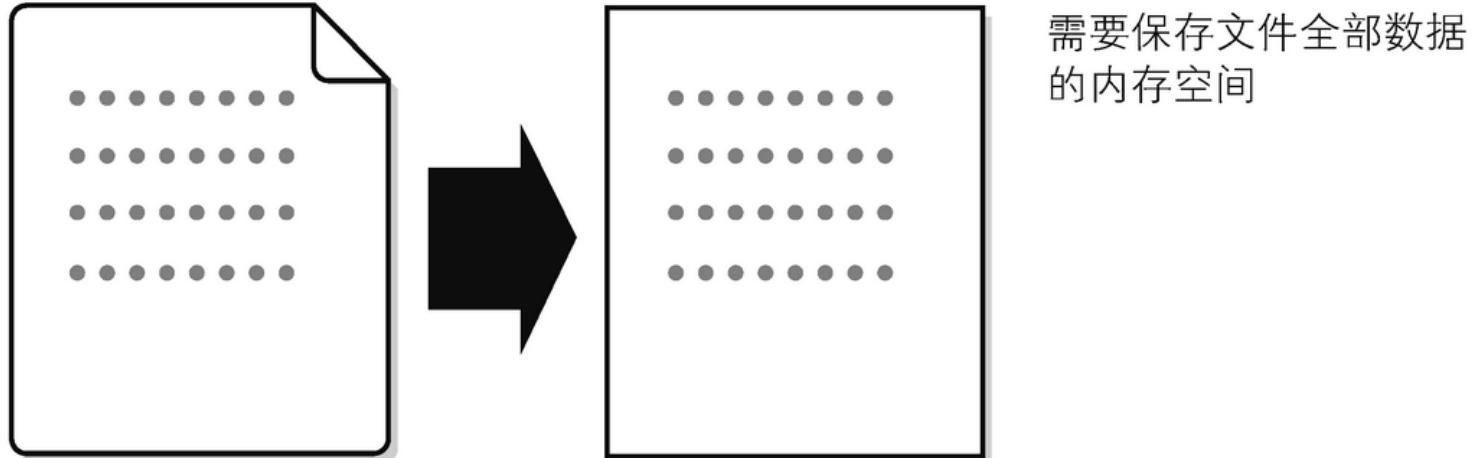
现在，我们了解了如何使用 Ruby 读取并输出文件里的所有内容。但是，刚才的程序有如下的问题：

- 一下子读取全部文件内容会很耗时；
- 读取文件的内容会暂时保存在内存中，遇到大文件时，程序有可能因此而崩溃。

例如一个文件有 100 万行数据，我们只希望读取其最初的几行。这种情况下，如果程序不管三七二十一读取文件的全部内容，无论从时间还是内存角度来讲，都是严重的浪费。

因此，我们只能放弃“读取文件全部内容”的做法（图 3.1），将代码清单 3.4 的程序改为逐行读取并输出（代码清单 3.7）。这样，只需要具备当前行数据大小的内存就足够了。

## ● file.read



## ● file.each\_line

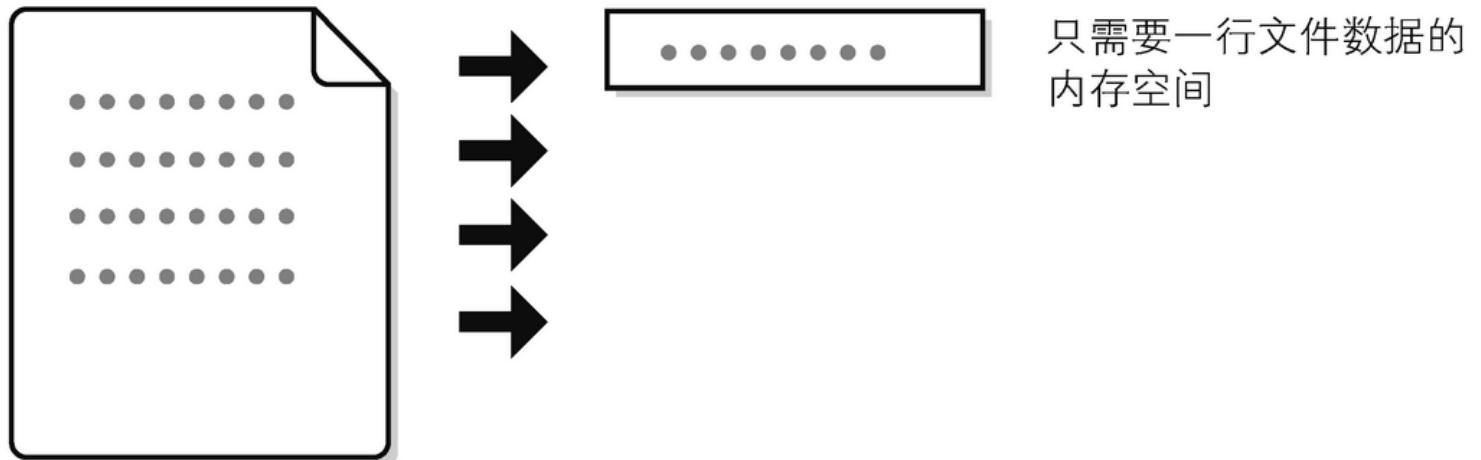


图 3.1 read 方法和 each\_line 方法的区别

代码清单 3.7 read\_line.rb

```
1: filename = ARGV[0]
2: file = File.open(filename)
3: file.each_line do |line|
4:   print line
5: end
6: file.close
```

程序的第 1 行和第 2 行与代码清单 3.4 是一样的，从第 3 行开始有了变化。程序的第 3 行到第 5 行使用了 each\_line 方法。

each\_line 方法很像第 2 章介绍的 each 方法。each 方法是用于逐个处理数组元素，顾名思义，each\_line 方法就是对文件进行逐行处理。因此在这里，程序会逐行读取文件的内容，使用 print 方法输出该行的文件内容 line，直到所有行的内容输出完为止。

### 3.2.3 从文件中读取指定模式的内容并输出

Unix 中有一个叫 grep 的命令。grep 命令利用正则表达式搜索文本数据，输出按照指定模式匹配到的行。我们试试用 Ruby 实现 grep 命令（代码清单 3.8）。

代码清单 3.8 simple\_grep.rb

```
1: pattern = Regexp.new(ARGV[0])
2: filename = ARGV[1]
3:
4: file = File.open(filename)
5: file.each_line do |line|
6:   if pattern =~ line
```

```
7:     print line
8:   end
9: end
10: file.close
```

命令行输入以下命令，执行代码清单 3.8。

```
> ruby simple_grep.rb 模式 文件名
```

程序有点长，我们逐行分析一下。

Ruby 执行该脚本时，需要有两个命令行参数——`ARGV[0]` 和 `ARGV[1]`。第 1 行，程序根据第 1 个参数创建了正则表达式对象，并赋值给变量 `pattern`。`Regexp.new(str)` 表示把字符串 `str` 转换为正则表达式对象。接着第 2 行，把第 2 个参数赋值给作为文件名的变量 `filename`。

第 4 行，打开文件，创建文件对象，并将其赋值给变量 `file`。

第 5 行，与代码清单 3.7 一样，读取一行数据，并将其赋值给变量 `line`。

第 6 行，使用 `if` 语句，判断变量 `line` 的字符串是否匹配变量 `pattern` 的正则表达式。如果匹配，则在程序第 7 行输出该字符串。这个 `if` 语句没有 `else` 部分，因此，若不匹配程序什么都不会做。程序循环读取文件，重复以上操作。

假设我们希望输出 Changelog 文件中含有 `matz` 的行，可以执行以下命令：

```
> ruby simple_grep.rb matz Changelog
```

`matz` 是松本行弘先生的昵称，这样我们就可以轻松找到他的修改之处了。

### 3.3 方法的定义

到目前为止，我们用过很多 Ruby 的方法了，其实我们也能定义方法。定义方法的语法如下所示：

```
def 方法名
  希望执行的处理
end
```

假设我们需要定义一个输出“Hello, Ruby.”的方法。

```
def hello
  print "Hello, Ruby.\n"
end
```

执行这 3 行代码的程序，实际并不会输出任何结果。这是由于在调用 `hello` 方法前，程序就已经结束了。因此方法定义好后，我们还要通过“调用”告诉 Ruby，我们要执行这个方法。

#### 代码清单 3.9 hello\_ruby2.rb

```
1: def hello
2:   puts "Hello, Ruby"
3: end
4:
5: hello()
```

#### 执行示例

```
> ruby hello_ruby2.rb
Hello, Ruby
```

执行 `hello()` 调用了 `hello` 方法后，程序就会执行第 1 ~ 3 行定义的内容。

### 3.4 其他文件的引用

有时，我们希望在其他的程序里也能重复使用程序的某部分。例如，在某个程序里写好某个方法，在其他程序里也可以调用。

大部分的编程语言都提供了把多个不同程序组合为一个程序的功能。像这样，被其他程序引用的程序，我们称为库（library）。

Ruby 使用 `require` 方法来引用库。

```
require 希望使用的库名
```

库名可以省略后缀 `.rb`。

调用 `require` 方法后，Ruby 会搜索参数指定的库，并读取库的所有内容（图 3.2）。库内容读取完毕后，程序才会执行 `require` 方法后面的处理。

## use\_hello.rb

### 处理流程

```
require "hello"  
hello()
```

## hello.rb

```
def hello  
  print("Hello, ruby.\n")  
end
```

图 3.2 库与引用库的程序

我们来实际操作一下，将刚才已经完成的 simple\_grep.rb 作为库提供给其他程序引用。作为库的文件不用做特别的修改，我们只需把定义了 simple\_grep 方法的文件（代码清单 3.10），和引用该文件的程序文件（代码清单 3.11）放在同一个文件夹即可。

### 代码清单 3.10 grep.rb

```
def simple_grep(pattern, filename)  
  file = File.open(filename)  
  file.each_line do |line|  
    if pattern =~ line  
      print line  
    end  
  end  
  file.close  
end
```

### 代码清单 3.11 use\_grep.rb

```
require "./grep"          # 读取grep.rb (省略".rb")  
  
pattern = Regexp.new(ARGV[0])  
filename = ARGV[1]  
simple_grep(pattern, filename) # 调用simple_grep 方法
```

这里，程序把执行 simple\_grep 方法时所需要的检索模式以及文件名两个参数，分别赋值给 pattern 变量以及 filename 变量。请注意，在这个例子里，use\_grep.rb 引用了在 grep.rb 定义的 simple\_grep 方法。与代码清单 3.8 一样，执行以下命令输出 Changelog 文件里包含 matz 字符串的行。

```
> ruby use_grep.rb matz Changelog
```

注 库与脚本放在同一文件夹时，需要用 ./ 来明示文件存放在当前目录。

Ruby 提供了很多便利的标准库，在我们的程序需要用到时，都可以使用 require 方法加以引用。

例如，我们的程序可以引用 date 库，这样程序就可以使用返回当前日期的 Date.today 的方法，或者返回指定日期对象的 Date.new 方法。下面是一个求从 Ruby 的生日——1993 年 2 月 24 日到今天为止的天数的小程序。关于 date 库，我们将会在第 20 章详细说明。

```
require "date"  
  
days = Date.today - Date.new(1993, 2, 24)  
puts(days.to_i)    #=> 7396
```

### 专栏

### pp 方法

Ruby 除了提供 p 方法外，还提供了一个有类似作用的方法——pp。pp 是英语 pretty print 的缩写。使用 pp 方法，我们需要使用 require 方法引用 pp 库。

### 代码清单 p\_and\_pp.rb

```
require "pp"  
  
v = [{  
  key00: "《Ruby 基础教程 第4 版》",  
  key01: "《Ruby 秘笈》",  
  key02: "《Rails3 秘笈》"  
}]  
p v
```

## 执行示例

```
> ruby p_and_pp.rb
[{:key00=>"《Ruby 基础教程 第4 版》", :key01=>"《Ruby 秘笈》", :key02=>"《Rails3 秘笈》"}]
[{:key00=>"《Ruby 基础教程 第4 版》",
:key01=>"《Ruby 秘笈》",
:key02=>"《Rails3 秘笈》"}]
```

与 `p` 方法有点不同，`pp` 方法在输出对象的结果时，为了更容易看懂，会适当地换行以调整输出结果。建议像本例的散列那样，在需要确认嵌套的内容时使用 `pp` 方法。

## 第 2 部分 Ruby 的基础

---

我们在写程序时都需要遵守一些编程规则。接下来，我们就来看看用 Ruby 编写程序时需要遵守哪些规则。

# 第4章 对象、变量和常量

本章会介绍使用 Ruby 操作数据时需要掌握的基础知识，主要有以下四部分内容。

- 对象
- 类
- 变量
- 常量

## 4.1 对象

在 Ruby 中，表现数据的基本单位称为对象（object）。

对象的类型非常多，我们这里只介绍一些常用的对象。

- 数值对象

1、`-10`、`3.1415` 等是表示数字的对象，另外还有表示矩阵、复数、素数、公式的对象。

- 字符串对象

`"你好"`、`"hello"` 等表示文字的对象。

- 数组对象、散列对象

表示多个数据的集合的对象。

- 正则表达式对象

表示匹配模式的对象。

- 时间对象

比如“2013 年 5 月 30 日早上 9 点”等表示时间的对象。

- 文件对象

一般我们可以理解为表示文件本身的对象，但确切来说，它是对文件进行读写操作的对象。

- 符号对象

表示用于识别方法等名称的标签的对象。

除此以外，Ruby 还有范围对象（Range）、异常对象（Exception）等。

## 4.2 类

Ruby 的类（class）表示的就是对象的种类。

对象拥有什么特性等，这些都是由类来决定的。到目前为止，我们介绍过的对象与其所属类的对应关系如表 4.1 所示。

表 4.1 对象与类的对象表

对象	类
数值	<code>Numeric</code>
字符串	<code>String</code>
数组	<code>Array</code>
散列	<code>Hash</code>

正则表达式	Regexp
文件	File
符号	Symbol

## 备注

“`xx` 类的对象”，我们一般也会说成“`xx` 类的实例（Instance）”。所有 Ruby 对象其实都是某个类的实例，因此在 Ruby 中的对象和实例的意义几乎是一样的。

另外，我们在强调某个对象是属于某个类时，经常会使用“实例”来代替“对象”。例如，我们会说“字符串对象 `"foo"` 是 `String` 类的实例”。

表 4.1 的类都是 Ruby 默认提供的，我们也可以按照实际需要自定义新的类。

类的相关内容，我们将会在第 8 章详细说明。

## 4.3 变量

在 1.9 节我们提到过，变量就像是对象的名片。

Ruby 中有四种类型的变量。

- 局部变量（local variable）
- 全局变量（global variable）
- 实例变量（instance variable）
- 类变量（class variable）

变量的命名方式决定了变量的种类。

- 局部变量

以英文字母或者 `_` 开头。

- 全局变量

以 `$` 开头。

- 实例变量

以 `@` 开头。

- 类变量

以 `@@` 开头。

除了以上四种类型以外，还有一种名为伪变量（pseudo variable）的特殊变量。<sup>1</sup> 伪变量是 Ruby 预先定义好的代表某特定值的特殊变量，因此即使我们在程序里给伪变量赋值，它的值也不会改变。Ruby 中，`nil`、`true`、`false`、`self` 等都是伪变量。它们表面上虽然看着像变量，但实际的行为又与变量有差别，因此称为伪变量。

<sup>1</sup>实际上还有一种叫预定义变量（Pre-defined Variable）的特殊变量。——译者注

### 局部变量与全局变量

首先让我了解一下什么是局部变量。

所谓局部，即变量在程序中的有效范围（也称为变量的作用域）是局部的。也就是说，在程序某个地方声明的变量名，在其他地方也可以使用，程序会也会认为这两个变量是没有关系的。<sup>2</sup>

<sup>2</sup>局部变量也可称为本地变量。——译者注

与局部变量相对的是全局变量。只要全局变量的名称相同，不管变量在程序的哪个部分使用，程序就认为是它们是同一个变量。

举个例子，假设有个程序引用了其他程序作为自己的程序一部分。这时，如果原程序与被引用程序中，都有一个相同名称的变量 `x`，由于 `x` 是局部变量，因此程序不会认为这两个变量 `x` 是同一个变量。但是，如果是拥有相同名称的全局变量 `$x`，则程序会认为这两个变量 `$x` 是相同的变量。

代码清单 4.1 和代码清单 4.2 是调查变量作用域的两个小程序。在 `scopetest.rb` 中，我们预先将变量 `$x` 和 `x` 都定义为 0 后，读取 `sub.rb` 的内容。在 `sub.rb` 中，我们再把刚才两个变量的值都设为 1。然后，回到 `scopetest.rb` 程序的第 6 行和第 7 行，我们输出这两个变量的值后会发现，`x` 的值没有变化，

但 \$x 的值已经是 1 了。这是由于在 scopetest.rb 以及 sub.rb 中，程序会把 \$x 当作同一个变量来处理，而把 x 当作不同的变量来处理。

#### 代码清单 4.1 scopetest.rb

```
1: $x = 0
2: x = 0
3:
4: require "./sub"
5:
6: p $x #=> 1
7: p x #=> 0
```

#### 代码清单 4.2 sub.rb

```
1: $x = 1 ## 对全局变量赋值
2: x = 1 ## 对局部变量赋值
```

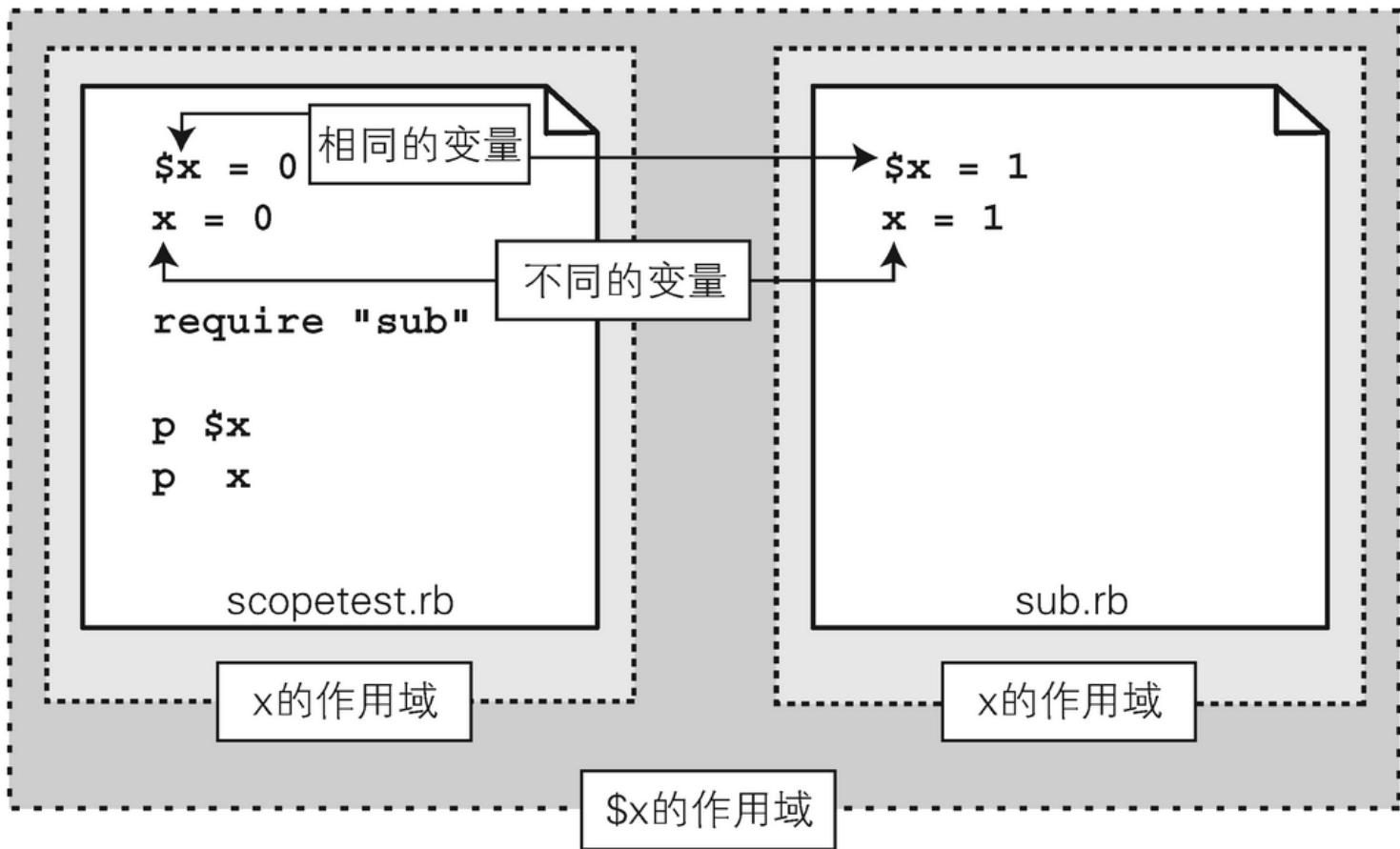


图 4.1 局部变量与全局变量

一般我们并不推荐使用全局变量。全局变量的值在程序的任何地方都可以修改，因此在规模较大的程序中使用时，会增加程序不必要的复杂度，给阅读程序、修改程序造成意想不到的麻烦。本书也很少对全局变量进行说明，示例中也没使用过。

程序首次给局部变量赋值的同时，该局部变量就被初始化了。如果引用了未初始化的局部变量，程序会抛出异常。

#### 执行示例

```
> irb --simple-prompt
>> x + 1
NameError: undefined local variable or method `1' for main:Object
from (irb):1
from /usr/local/bin/irb:16:in `<main>'
```

实例变量与类变量，是在定义类的时候用到的变量，因此我们留到第 8 章再详细说明。

## 4.4 常量

与变量类似的有常量（constant）。常量的作用和变量一样，是某个对象的“名片”。不过与变量不同的是，对已经赋值的常量再进行赋值时，Ruby 会做出警告。

#### 执行示例

```
> irb --simple-prompt
```

```

>> TEST = 1
=> 1
>> TEST = 2
(irb):4: warning: already initialized constant TEST
(irb):3: warning: previous definition of TEST was here
=> 2

```

常量以大写英文字母开头。例如，Ruby 的运行版本（`RUBY_VERSION`）、运行平台（`RUBY_PLATFORM`）、命令行参数数组（`ARGV`）等，都是 Ruby 预定义的好常量。关于预定义常量，我们将会在 B.4.2 节介绍。

## 4.5 保留字

表 4.2 中的单词，在程序中作为名称使用时会受到限制。这些受到限制的单词，我们称为保留字。在程序里，如果不小心使用了 `end`、`next` 等作为变量名，Ruby 会提示我们语法错误。

### 执行示例

```

> irb --simple-prompt
>> end = 1
SyntaxError: (irb):8: syntax error, unexpected keyword_end
end = 1
^
from /usr/local/bin/irb:16:in `<main>'

```

表 4.2 Ruby 的关键字一览

<code>\_\_LINE\_\_</code>	<code>\_\_ENCODING\_\_</code>	<code>\_\_FILE\_\_</code>	<code>BEGIN</code>	<code>END</code>
<code>alias</code>	<code>and</code>	<code>begin</code>	<code>break</code>	<code>case</code>
<code>class</code>	<code>def</code>	<code>defined?</code>	<code>do</code>	<code>else</code>
<code>elsif</code>	<code>end</code>	<code>ensure</code>	<code>false</code>	<code>for</code>
<code>if</code>	<code>in</code>	<code>module</code>	<code>next</code>	<code>nil</code>
<code>not</code>	<code>or</code>	<code>redo</code>	<code>rescue</code>	<code>retry</code>
<code>return</code>	<code>self</code>	<code>super</code>	<code>then</code>	<code>true</code>
<code>undef</code>	<code>unless</code>	<code>until</code>	<code>when</code>	<code>while</code>
<code>yield</code>				

## 4.6 多重赋值

我们已经介绍过“`变量 = 值`”这样的变量赋值方法，Ruby 还提供了一个只用一个表达式就能给多个变量赋值的简便方法——多重赋值。很多情况下我们都会用到多重赋值，在这里举几个比较典型的例子供大家参考。

### 4.6.1 合并执行多个赋值操作

有时我们希望把一组的变量同时赋值。

```

a = 1
b = 2
c = 3

```

像这样的赋值语句，程序可以简化为只有一行。

```
a, b, c = 1, 2, 3
```

这样就能轻松地将 1、2、3 分别赋值给变量 `a`、`b`、`c`。如果对一组不相关的变量进行多重赋值，程序会变得很难懂，因此建议对彼此相关变量进行多重赋值。

即使 `=` 左右两边列表的数量不相等，Ruby 也不会报错。左边被赋值的变量的个数比较多时，Ruby 会自动将 `nil` 赋值给未分配值的变量。

```

a, b, c, d = 1, 2
p [a, b, c]    #=> [1, 2, nil]

```

变量部分比较少时，Ruby 会忽略掉该值，不会分配多余的值。

```

a, b, c = 1, 2, 3, 4
p [a, b, c]    #=> [1, 2, 3]

```

变量前加上`*`, 表示 Ruby 会将未分配的值封装为数组赋值给该变量。

```
a, b, *c = 1, 2, 3, 4, 5
p [a, b, c]      #=> [1, 2, [3, 4, 5]]
a, *b, c = 1, 2, 3, 4, 5
p [a, b, c]      #=> [1, [2, 3, 4], 5]
```

## 4.6.2 置换变量的值

现在我们来考虑一下如何置换变量`a`、`b` 的值。通常, 我们需要一个临时变量`tmp` 暂时地保存变量的值。

```
a, b = 0, 1
tmp = a      # 暂时保存变量a 的值
a = b      # 将变量b 的值赋值给a
b = tmp      # 将原本变量a 的值赋值给变量b
p [a, b]      #=> [1, 0]
```

使用多重赋值, 只需一行程序就搞定了。

```
a, b = 0, 1
a, b = b, a      # 置换变量a、b 的值
p [a, b]      #=> [1, 0]
```

## 4.6.3 获取数组的元素

用数组赋值, 左边有多个变量时, Ruby 会自动获取数组的元素进行多重赋值。

```
ary = [1, 2]
a, b = ary
p a      #=> 1
p b      #=> 2
```

只是希望获取数组开头的元素时, 可以按照以下示例那样做。左边的变量列表以, 结束, 给人一种“是不是还没写完?”的感觉, 建议尽量少用这样的写法。

```
ary = [1, 2]
a, = ary
p a      #=> 1
```

## 4.6.4 获取嵌套数组的元素

我们来看看数组`[1, [2, 3], 4]`, 用之前介绍的方法, 我们可以分别取出`1`、`[2, 3]`、`4` 的值。

```
ary = [1, [2, 3], 4]
a, b, c = ary
p a      #=> 1
p b      #=> [2, 3]
p c      #=> 4
```

像下面那样把左边的变量括起来后, 就可以再进一步将内部数组的元素值取出来。

```
ary = [1, [2, 3], 4]
a, (b1, b2), c = ary      # 对与数组结构相对应的变量赋值
p a      #=> 1
p b1     #=> 2
p b2     #=> 3
p c      #=> 4
```

只要等号左边的变量的结构与数组的结构一致, 即使再复杂的结构, 多重赋值都可以轻松对应。

### 专栏

#### 变量的命名方法

以变量名开头来决定变量的种类, 这是 Ruby 中对变量命名时唯一要坚决遵守的规则。虽然如此, 但是根据以往的编程经验, 也有一些非强制性的、约定俗成的变量命名规则。在大多数情况下, 遵循这些规则能使程序变得易于阅读, 对我们来说有百利而无一害。

- 不要过多使用省略的名称

有些编程语言会限制变量名的长度, 但 Ruby 不需要在意变量名的长度。当然, 过长的名称是不便于阅读的, 但是与其起个不知所云的短的名称, 老老实实地为变量取个长点的好理解的名称, 对以后阅读、理解程序是非常有帮助的。

但是，我们还是有一些约定俗成的短名称变量。进行数学、物理等计算时，根据计算对象的不同，很多情况下会使用短名称的变量名，像坐标使用 `x`、`y`、`z`，速度使用 `v`、`w`，循环次数使用 `m`、`n` 等。另外，我们编写程序时，也经常使用 `i`、`j`、`k` 等作为循环时需要用到的变量名。

- 对于多个单词组合的变量名，使用 `_` 隔开各个单词，或者单词以大写字母开头

也就是说，要么这样叫做 `sort_list_by_name`，要么叫做 `sortListByName`。一般来讲，Ruby 中的变量名和方法名使用前者，类名和模块名的使用后者。

# 第 5 章 条件判断

本章我们将详细讨论一下控制结构之一的条件判断，主要包括以下内容。

- 什么是条件判断。
- 条件判断中不可或缺的比较运算符、真假值 1、逻辑运算符。
- 条件判断的种类及其写法和使用方法。

1也称布尔值。——译者注

## 5.1 什么是条件判断

接下来，我们来考虑一下如何将公历转换为平成纪年 2。首先，我们将输入的字符串转换为数值后减去 1988，最后输出运算结果，结束程序。程序如代码清单 5.1 所示。

2日本的纪年方法。1989 年为平成元年，2014 年是平成 26 年。——译者注

代码清单 5.1 ad2heisei.rb

```
# 将公历转换为平成纪年

ad = ARGV[0].to_i
heisei = ad - 1988
puts heisei
```

执行结果如下：

### 执行示例

```
> ruby ad2heisei.rb 2013
25
```

但是，这个程序有点小问题。如果我们输入 1989 年以前的年份，返回值会变成 0 或者负数。

### 执行示例

```
-8
```

按道理，1989 年以前的年份是不能转换为平成 XX 年的，因此程序本不应允许输入示例中那样的年份。我们将程序稍微改进一下，若输入 1989 年以前的年份，程序则返回“无法转换”的提示。

在这样的情况下，为了实现程序在“某个条件时执行○○处理，否则执行 ×× 处理”，Ruby 为我们准备了条件判断语句。

条件判断语句主要有以下三种。

- `if` 语句
- `unless` 语句
- `case` 语句

接下来，我们将会介绍这些条件判断语句及其写法。

## 5.2 Ruby 中的条件

在说明条件语句之前，我们首先来看看在 Ruby 中是如何写条件的。

### 条件与真假值

我们在之前的章节已经介绍过了在条件判断中常用到的比较运算符。等号`==`，不等号`<`、`>`等都是比较运算符。

比较的结果分为 `true` 和 `false` 两种。顾名思义，比较结果正确时为 `true`，错误时为 `false`。

除了比较运算符外，Ruby 中还有很多可以作为条件判断的方法。例如，字符串的 `empty?` 方法，该字符串的长度为 0 时返回 `true`，否则返回 `false`。

```
p "".empty?    #=> true
p "AAA".empty? #=> false
```

另外，除了 `true` 和 `false` 外，还有其他值可作为条件判断的值。例如，用正则表达式进行匹配时，匹配成功返回该字符串的位置，匹配失败返回 `nil`。

```
p /Ruby/ =~ "Ruby"    #=> 0  
p /Ruby/ =~ "Diamond" #=> nil
```

关于 Ruby 中的真假值的定义，可参考表 5.1。

表 5.1 Ruby 的真假值

真	false、nil 以外的所有对象
假	false、nil

也就是说，Ruby 会认为 false 与 nil 代表假，除此以外的所有值都代表真。因此，Ruby 中的真 / 假并非绝对等同于 true/false。true 代表真，false 代表假，同时，不返回 true 或 false 的方法只要能返回 nil，也可作为条件判断的表达式来使用。另外，在 Ruby 中还有个约定俗成的规则，为了使程序更容易理解，返回真假值的方法都要以 ? 结尾。建议大家在写程序时也遵守这个规则。

### 5.3 逻辑运算符

在判断多个条件表达式时，我们会用到逻辑运算符 && 和 ||。

条件 1 && 条件 2

表示条件 1 为真，并且条件 2 也为真时，则整体的表达式返回真。两者中只要一个返回假时，则整体的表达式返回假。

相对地，

条件 1 || 条件 2

表示条件 1 为真，或者条件 2 为真时，整体的表达式返回真。两者同时为假时，则整体的表达式返回假。

还有表示否定的逻辑运算符：

! 条件

表示相反的条件。也就是，条件为假时，表达式返回真；条件为真时，表达式返回假。例如，我们想判断整数 x 是否在 1 到 10 之间，if 语句可以这么写：

```
if x >= 1 && x <= 10  
|  
end
```

与上面的条件相反，表示“1 到 10 以外”时使用 !，表达式可以写成 !(x >= 1 && x <= 10)。不过，像下面写成“小于 1，或者大于 10”可能更加直接，更便于理解。

```
if x < 1 || x > 10  
|  
end
```

条件判断对于控制程序的行为非常重要。过于复杂、难以理解的条件，会使程序的目的也会变得难以琢磨。建议大家在写程序时，注意尽量写便于理解的条件。

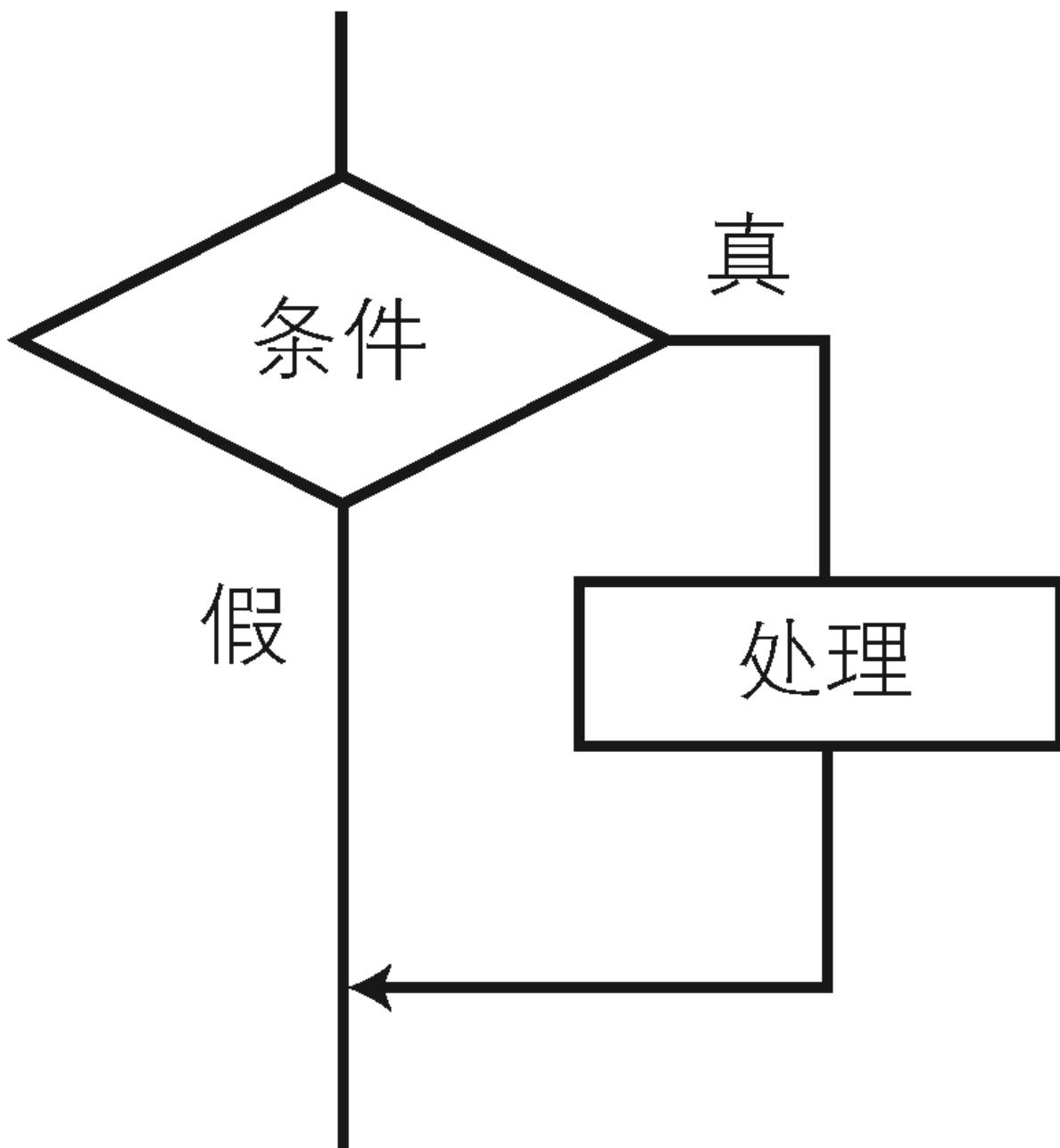
在 Ruby 中，还有与 &&、||、! 意思相同，但优先级略低的逻辑运算符 and、or、not。关于运算符的优先级，我们将在第 9 章 9.5 节讨论。

### 5.4 if 语句

接下来，我们就来看看条件判断语句到底如何使用。if 语句是最基本的条件判断语句，用法如下：

```
if 条件 then  
  处理  
end
```

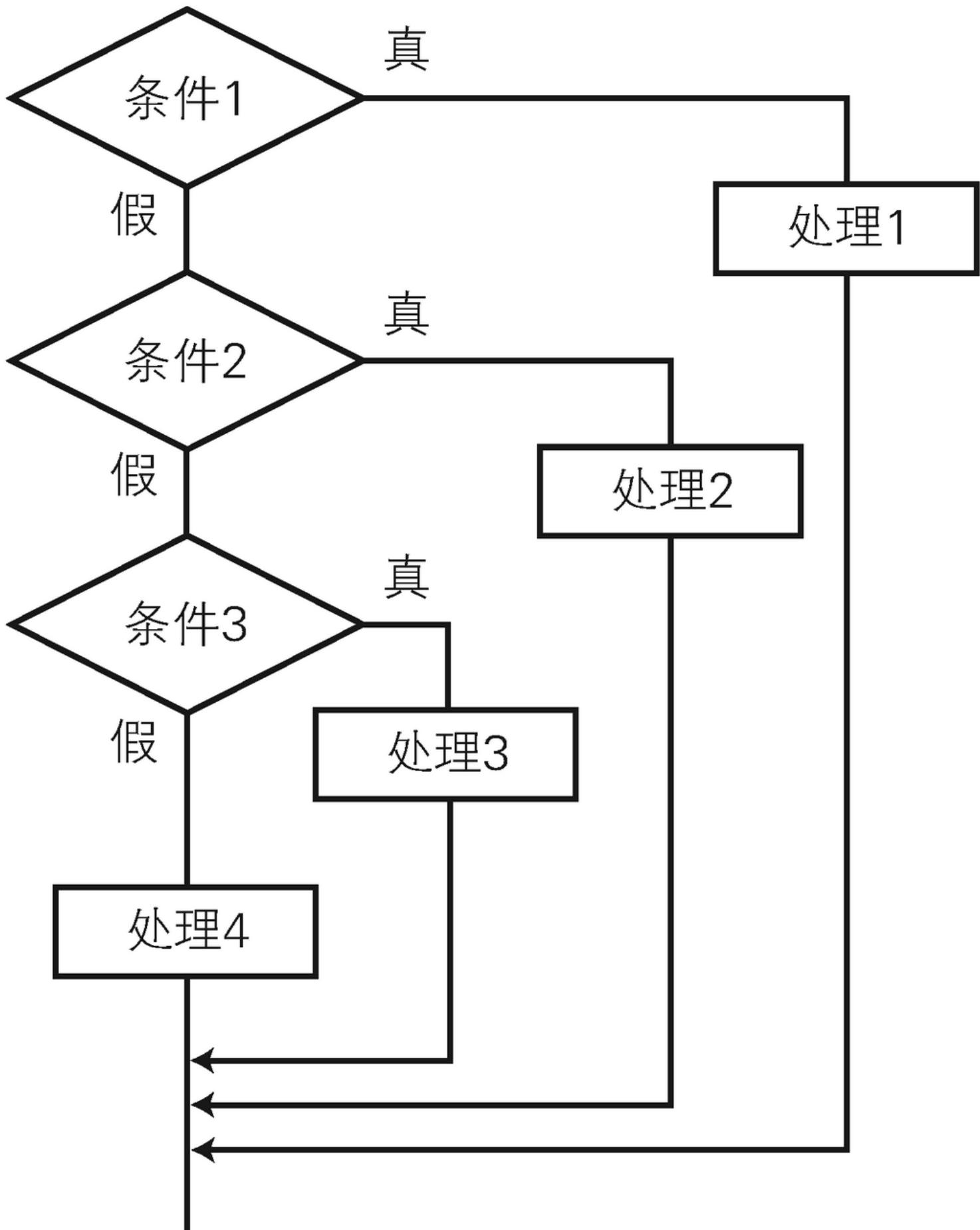
※ 可以省略 then



在这基础上可再加上 `elsif`、`else` :

```
if 条件 1 then
    处理 1
elsif 条件 2 then
    处理 2
elsif 条件 3 then
    处理 3
else
    处理 4
end
```

※ 可以省略 `then`



Ruby 会按照从上到下的顺序进行判断。首先，条件 1 为真时程序执行处理 1。条件 1 为假时，程序再判断条件 2，若为真时执行处理 2。同样地，条件 2 为假时，程序再判断条件 3.....本例中虽然只有 4 个条件分支，但根据实际需要可以添加无限个的分支。最后，如果前面所有条件都为假时则执行处理 4。

我们来看看使用 `elsif` 的例子（代码清单 5.2）。

代码清单 5.2 `if_elsif.rb`

```

a = 10
b = 20
if a > b

```

```
puts "a 比b 大"  
elsif a < b  
  puts "a 比b 小"  
else  
  puts "a 与b 相等"  
end
```

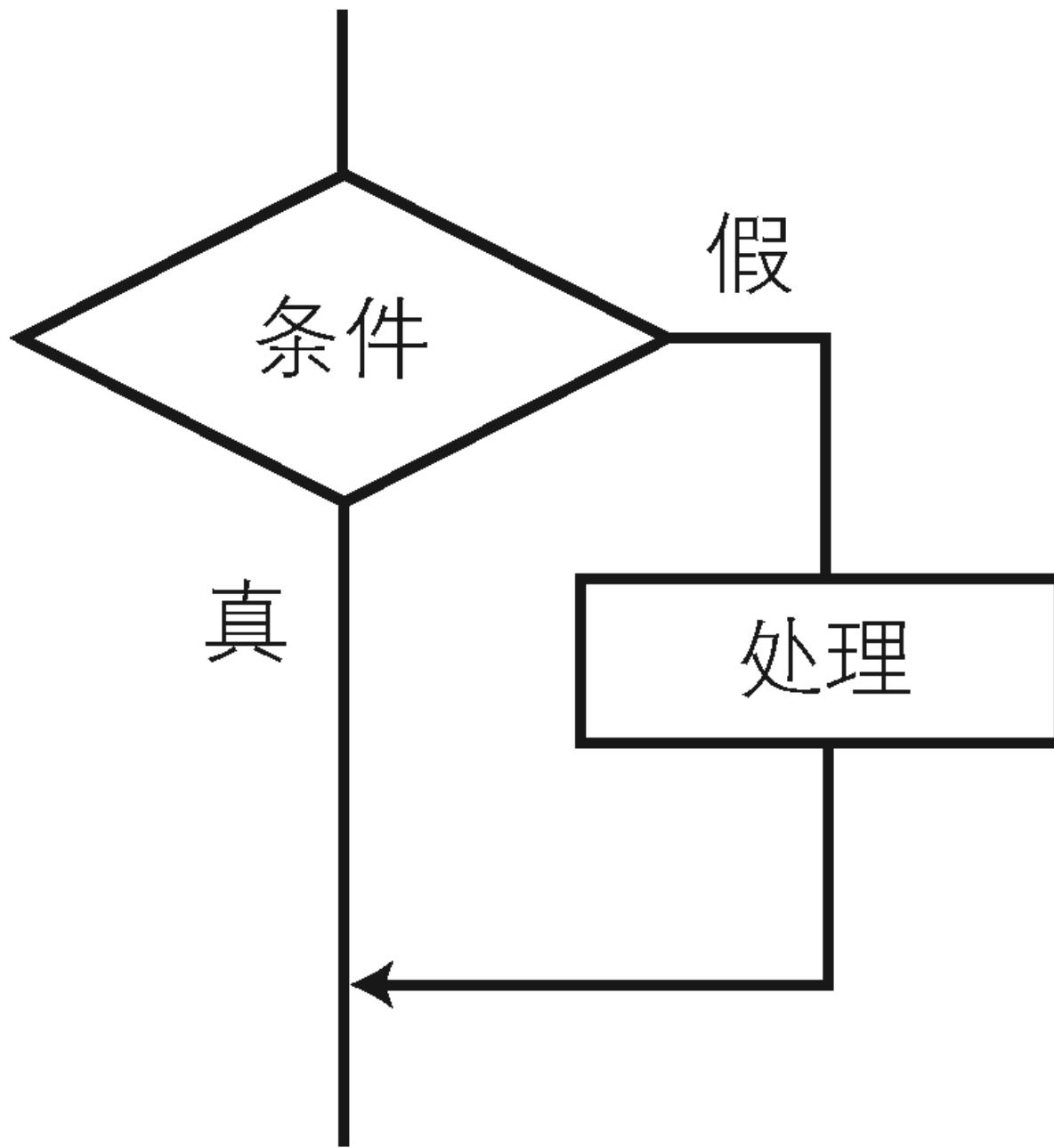
这是一个比较 a、b 大小的程序。比较结果分为 a 比 b 大、a 比 b 小或者 a 与 b 相等三种情况。这种情况下，我们可以使用 `if ~ elsif ~ else` 结构。

## 5.5 unless 语句

`unless` 语句的用法刚好与 `if` 语句相反。`unless` 语句的用法如下：

```
unless 条件 then  
  处理  
end
```

※ 可以省略 `then`



`unless` 语句的形式和 `if` 语句一样。但 `if` 语句是条件为真时执行处理，`unless` 语句则刚好相反，条件为假时执行处理。

下面是使用 `unless` 的例子（代码清单 5.3）。

代码清单 5.3 unless.rb

```
a = 10  
b = 20  
unless a > b  
  puts "a 不比b 大"
```

这个程序执行后输出“`a` 不比 `b` 大”。`unless` 语句的条件 `a > b` 为假，所以程序执行了 `puts` 方法。

`unless` 语句也可以使用 `else`。

`unless` 条件

    处理 1

`else`

        处理 2

`end`

这个与下面的 `if` 语句是等价的。

`if` 条件

    处理 2

`else`

        处理 1

`end`

对比以上两种写法，我们可以知道处理 1 和处理 2 的位置互换了，`if` 语句通过这样的互换，能达到与使用 `unless` 语句时同样的效果。

## 5.6 case 语句

条件有多个时，使用 `if` 与 `elsif` 的组合虽然也能达到判断多个条件的效果，但是如果需要比较的对象只有一个，根据这个对象值的不同，执行不同的处理时，使用 `case` 语句会使程序更简单，更便于理解。

`case` 语句的用法如下：

`case` 比较对象

`when` 值 1 `then`

        处理 1

`when` 值 2 `then`

        处理 2

`when` 值 3 `then`

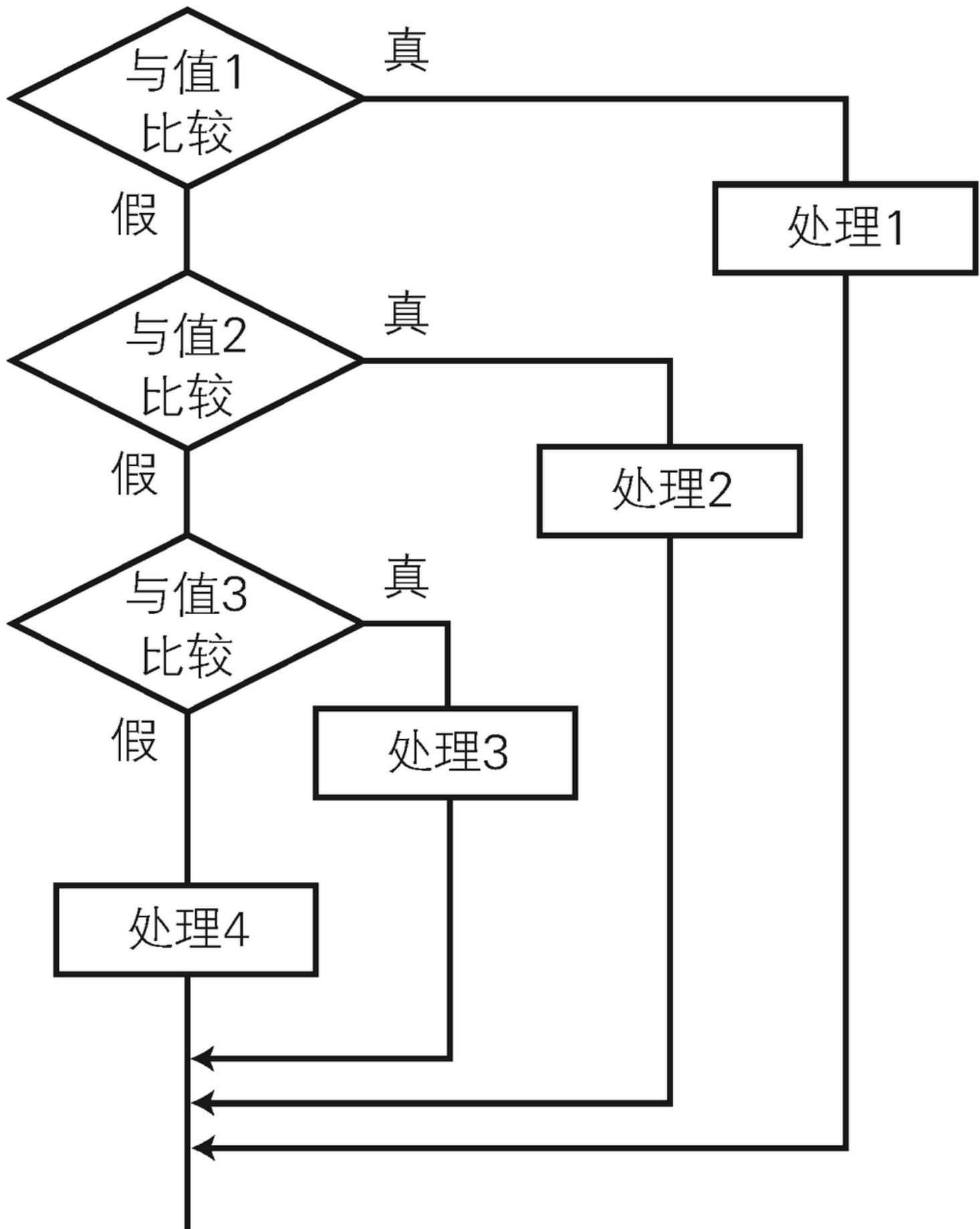
        处理 3

`else`

        处理 4

`end`

※ 可以省略 `then`



本例的比较对象的值有 3 个，但根据实际情况可以无限增加下去。

还有，`when` 可以一次指定多个值。下面的示例（代码清单 5.4）从数组 `tags` 的开头依次取出元素，判断元素值，输出相应的结果。

#### 代码清单 5.4 case.rb

```

tags = [ "A", "IMG", "PRE" ]
tags.each do |tagname|
  case tagname
  when "P", "A", "I", "B", "BLOCKQUOTE"
    puts "#{tagname} has child."
  end
end

```

```
when "IMG", "BR"
  puts "#{tagname} has no child."
else
  puts "#{tagname} cannot be used."
end
end
```

## 执行示例

```
> ruby case.rb
A has child.
IMG has no child.
PRE cannot be used.
```

我们再来看看其他例子。

### 代码清单 5.5 case\_class.rb

```
array = [ "a", 1, nil ]
array.each do |item|
  case item
  when String
    puts "item is a String."
  when Numeric
    puts "item is a Numeric."
  else
    puts "item is something."
  end
end
```

## 执行示例

```
> ruby case_class.rb
item is a String.
item is a Numeric.
item is something.
```

在本例中，程序判断传过来的对象类型是字符串（`String` 类）还是数值（`Numeric` 类），或者均不是以上两者，然后再输出相应结果。

在这里，我们同样是使用 `case` 语句，不过判断的主体与之前的例子有点区别。本例中的 `when` 实际并不是直接判断传过来的字符串，而是先查找该对象属于哪个类，然后再根据这个类的信息来进行条件判断。

我们还可以根据正则表达式的匹配结果进行不同处理。下面是使用正则表达式做判断的 `case` 语句的例子。

```
text.each_line do |line|
  case line
  when /^From:/i
    puts "发现寄信人信息"
  when /^To:/i
    puts "发现收信人信息"
  when /^Subject:/i
    puts "发现主题信息"
  when /^\$/
    puts "头部解析完毕"
    exit
  else
    ## 跳出处理
  end
end
```

这是一个解析电子邮件头部的程序。为了简化程序，我们并没有考虑有多个头部的情况，而且电子邮件里的内容我们也没取出来。在这里，大家掌握程序的大概的处理流程就可以了。

`each_line` 方法逐行读取电子邮件正文数据 `text`，并将每行的内容赋值给变量 `line`。这个是处理文件、文本数据时的典型的写法。

接着 `case` 语句判断得到的字符串的内容，执行不同的处理。以 `From:` 开头时输出“发现寄信人信息”，以 `To:` 开头时输出“发现收信人信息”，以 `Subject:` 开头时输出“发现主题信息”。

最后的 `when` 判断的 `^\$/`，表示行的开头后马上就接着是行尾的意思<sup>3</sup>，也就是说，这是表示空行的正则表达式。电子邮件的头部和正文间一定会以空行作间隔，因此根据这个规则我们就可以把空行作为头部结束的标志。当 `when` 遇到空行，输出“头部解析完毕”的信息后调用 `exit` 方法，结束程序。

<sup>3</sup> 在正则表达式中，`^` 表示匹配字符串的开始，`$` 表示匹配字符串的结束。——译者注

## ==== 与 case 语句

在 `case` 语句中，`when` 判断值是否相等时，实际是使用 `==` 运算符来判断的。左边是数值或者字符串时，`==` 与 `=` 的意义是一样的，除此以外，`==` 还可以与 `=~` 一样用来判断正则表达式是否匹配，或者判断右边的对象是否属于左边的类，等等。对比单纯的判断两边的值是否相等，`==` 能表达更加广义的“相等”。

```
p (/zz/ === "xyzzy")    #=> true
p (String === "xyzzy")  #=> true
p ((1..3) === 2)        #=> true
```

用 `if` 语句改写 `case` 语句的程序如下所示。请注意 `when` 指定的对象在 `==` 的左边。

```
case value
when A
  处理 1
when B
  处理 2
else
  处理 3
end
```



```
if A == value
  处理 1
elsif B == value
  处理 2
else
  处理 3
end
```

## 5.7 if 修饰符与 unless 修饰符

`if` 与 `unless` 可以写在希望执行的代码的后面。像下面这样：

```
puts "a 比 b 大" if a > b
```

这与下面的写法是等价的。

```
if a > b
  puts "a 比 b 大"
end
```

使用修饰符的写法会使程序更加紧凑。通常，我们在希望强调代码执行的内容时会使用修饰符写法。同样地，在使用修饰符写法时，请大家注意程序的易读性。

## 5.8 总结

本章介绍了以下内容。

- 真假值

真假值是条件表达式的返回值。

- `nil` 或者 `false` 时为假
- 除此以外的值为真

- 条件判断语句

条件判断语句有：

- `if` 语句
- `unless` 语句
- `case` 语句

- 比较

用 `if` 语句、`unless` 语句做比较时，会用到比较运算符（`==`, `!=`, `<`, `>` 等）、以 `?` 结尾的方法、逻辑运算符等。

- **if 语句、unless 语句**

两者皆为条件判断的基本语句。

- **case 语句**

在遇到像“根据对象的不同状态，采取不同的处理”那样的分情况处理时，我们会用到 `case` 语句。

分情况处理时，不同的对象所采取的判断方法也不一样。具体来说是根据 `==` 运算符的比较特性，实现分情况处理。更详细的说明请参考专栏“`==` 与 `case` 语句”。

大部分的编程语言都有条件判断。本书也在很多地方使用了条件判断。大家可以参考这些内容，逐渐熟练掌握什么时候该用哪种条件判断语句。

## 专栏

### 对象的同一性

所有的对象都有标识和值。

标识（ID）用来表示对象同一性。Ruby 中所有对象都是唯一的，对象的 ID 可以通过 `object_id`（或者 `_id_`）方法取得。

```
ary1 = []
ary2 = []
p ary1.object_id    #=> 67653636
p ary2.object_id    #=> 67650432
```

我们用 `equal?` 方法判断两个对象是否同一个对象（ID 是否相同）。

```
str1 = "foo"
str2 = str1
str3 = "f" + "o" + "o"
p str1.equal?(str2)    #=> true
p str1.equal?(str3)    #=> false
```

对象的“值”就是对象拥有的信息。例如，只要对象的字符串内容相等，Ruby 就会认为对象的值相等。Ruby 使用 `==` 来判断对象的值是否相等。

```
str1 = "foo"
str2 = "f" + "o" + "o"
p str1 == str2    #=> true
```

除了 `==` 以外，Ruby 还提供 `eq?` 方法用来判断对象的值是否相等。`==` 与 `eq?` 都是 `Object` 类定义的方法，大部分情况下它们的执行结果都是一样的。但也有例外，数值类会重定义 `eq?` 方法，因此执行后有不一样结果。

```
p 1.0 == 1      #=> true
p 1.0.eq?(1)   #=> false
```

凭直觉来讲，把 `1.0` 与 `1` 判断为相同的值会更加方便。在一般情况进行值的比较时使用 `==`，但是在一些需要进行更严谨的比较的程序中，就需要用到 `eq?` 方法。例如，`0` 与 `0.0` 作为散列的键时，会判断为不同的键，这是由于散列对象内部的键比较使用了 `eq?` 方法来判断。

```
hash = { 0 => "0" }
p hash[0.0]    #=> nil
p hash[0]      #=> "0"
```

# 第6章 循环

与条件判断一样，循环也遍布于程序的各个角落，是程序中不可缺少的重要组成部分。本章我们将会探讨：

- 程序中的循环是什么
- 写循环时需要注意的事项
- 循环的种类及其写法

## 6.1 循环的基础

我们在编写程序时，常常遇到“希望这个处理重复执行多次”的情况。例如：

- 希望同样的处理执行 X 次

更复杂点的例子有：

- 用其他对象置换数组里的所有元素；
- 在达成某条件之前，一直重复执行处理。

这时，我们都需要用到循环。

接下来，我们将会介绍 Ruby 中基本的循环结构。其中比较特别的是，除了用传统的循环语句实现循环外，我们还能用方法来实现循环，也就是说我们可以根据自己的需要定制循环方法。关于如何定制循环，我们会在第 11 章再详细说明，这里我们先介绍一些预定义的循环语法结构。

## 6.2 循环时的注意事项

下面两点是循环时必须注意的。

- 循环的主体是什么
- 停止循环的条件是什么

大家也许会认为，我们自己写的循环处理，“循环的主体是什么”我们自己总会知道吧。但是，实际编写程序时，稍不注意就会发生把不应该循环的处理加入到循环中这样的错误。而且，如果是循环里再嵌套循环的结构，在哪里做怎么样的循环、循环的结果怎么处理等都可能会使程序变得难以读懂。

另外，如果把“停止循环的条件”弄错了，有可能会发生处理无法终止，或者处理还没完成但已经跳出循环等这样的情况。大家写循环结构时务必注意上述两点，避免发生错误。

## 6.3 实现循环的方法

Ruby 中有两种实现循环的方法。

- 使用循环语句

利用 Ruby 提供现有的循环语句，可以满足大部分循环处理的需求。

- 使用方法实现循环

将块传给方法，然后在块里面写上需要循环的处理。一般我们在为了某种特定目的而需要定制循环结构时，才使用方法来实现循环。

下面是我们接下来要介绍的六种循环语句或方法。

- `times` 方法
- `while` 语句
- `each` 方法
- `for` 语句
- `until` 语句
- `loop` 方法

Ruby 的常用循环结构就介绍到这里，接下来就让我们来具体看看如何使用这些语句或方法实现循环。

## 6.4 `times` 方法

如果只是单纯执行一定次数的处理，用 `times` 方法可以很轻松实现。

假设我们希望把“满地油菜花”这个字符串连续输出 7 次。

## 代码清单 6.1 times.rb

```
7.times do
  puts "满地油菜花"
end
```

### 执行示例

```
> ruby times.rb
满地油菜花
满地油菜花
满地油菜花
满地油菜花
满地油菜花
满地油菜花
满地油菜花
```

使用 `times` 方法实现循环时，需要用到块 `do ~ end`。

```
循环次数.times do
  希望循环的处理
end
```

块的 `do ~ end` 部分可以用 `{~}` 替换，像下面这样：

```
循环次数.times {
  希望循环的处理
}
```

在 `times` 方法的块里，也是可以获知当前的循环次数的。

```
10.times do |i|
  |
end
```

这样，就可以把当前的循环次数赋值给变量 `i`。我们来看看实际的例子（代码清单 6.2）。

## 代码清单 6.2 times2.rb

```
5.times do |i|
  puts "第#{i} 次的循环。"
end
```

### 执行示例

```
> ruby times2.rb
第 0 次的循环。
第 1 次的循环。
第 2 次的循环。
第 3 次的循环。
第 4 次的循环。
```

请注意循环的次数是从 0 开始计算的。把循环次数的初始值设为 1 不失为一个好方法，但可惜我们不能这么做，因此我们只能在块里面对循环次数做调整（代码清单 6.3）。

## 代码清单 6.3 times3.rb

```
5.times do |i|
  puts "第#{i+1} 次的循环。"
end
```

### 执行示例

```
> ruby times3.rb
第 0 次的循环。
第 1 次的循环。
第 2 次的循环。
第 3 次的循环。
第 4 次的循环。
```

但是，这样的写法会使变量 `i` 的值与实际输出的值产生差异。从降低程序复杂度来看，这并不是一个好的编程习惯。若是对循环次数比较在意时，我们不

必勉强使用 `times` 方法，可使用下面即将介绍的 `for` 语句和 `while` 语句。

## 6.5 for 语句

`for` 语句同样是用于实现循环的。需要注意的是，与刚才介绍的 `times` 方法不同，`for` 并不是方法，而是 Ruby 提供的循环控制语句。

以下是使用 `for` 语句的典型示例（代码清单 6.4）。

代码清单 6.4 `for.rb`

```
1: sum = 0
2: for i in 1..5
3:   sum = sum + i
4: end
5: puts sum
```

执行示例

```
> ruby for.rb
15
```

这是一个求从 1 到 5 累加的程序。`for` 语句的结构如下所示：

`for 变量 in 开始时的数值..结束时的数值 do`

希望循环的处理

`end`

※ 可以省略 `do`

我们回顾一下程序代码清单 6.4。程序第 1 行将 0 赋值给变量 `sum`，程序第 5 行输出变量 `sum` 的值并换行。

第 2 行到第 4 行的 `for` 语句指定变量 `i` 的范围是从 1 到 5。也就是说，程序一边从 1 到 5 改变变量 `i` 的值，一边执行 `sum = sum + i`。如果不使用循环语句，这个程序可以改写为：

```
sum = 0
sum = sum + 1
sum = sum + 2
sum = sum + 3
sum = sum + 4
sum = sum + 5
puts sum
```

`for` 语句与 `times` 方法不一样，循环的开始值和结束值可以任意指定。例如，我们想计算从变量 `from` 到变量 `to` 累加的总数，使用 `times` 方法的程序为：

```
from = 10
to = 20
sum = 0
(to - from + 1).times do |i|
  sum = sum + (i + from)
end
puts sum
```

使用 `for` 语句的程序为：

```
from = 10
to = 20
sum = 0
for i in from..to
  sum = sum + i
end
puts sum
```

使用 `for` 语句后的程序变得更加简单了。

另外，`sum = sum + i` 这个式子也有更简单的写法：

```
sum += i
```

本例是加法的简写，做减法、乘法时也同样可以做这样的省略。

```
a -= b
a *= b
```

第 9 章我们会再详细讨论这方面的内容，现在暂时先记着有这么一个省略写法就可以了。

## 6.6 普通的 for 语句

其实上一节介绍的是 `for` 语句的特殊用法，普通的 `for` 语句如下所示：

```
for 变量 in 对象 do  
    希望循环的处理  
end
```

※ 可以省略 `do`

可以看出，`in` 后面的部分和之前介绍的有点不同。

但和之前的 `for` 语句相比也并非完全不一样。实际上，`..` 或者 `...` 都是创建范围对象时所需的符号。

当然，并非任何对象都可以指定给 `for` 语句使用。下面是使用数组对象的例子。

### 代码清单 6.5 for\_names.rb

```
names = ["awk", "Perl", "Python", "Ruby"]  
for name in names  
    puts name  
end
```

#### 执行示例

```
> ruby for_names.rb  
awk  
Perl  
Python  
Ruby
```

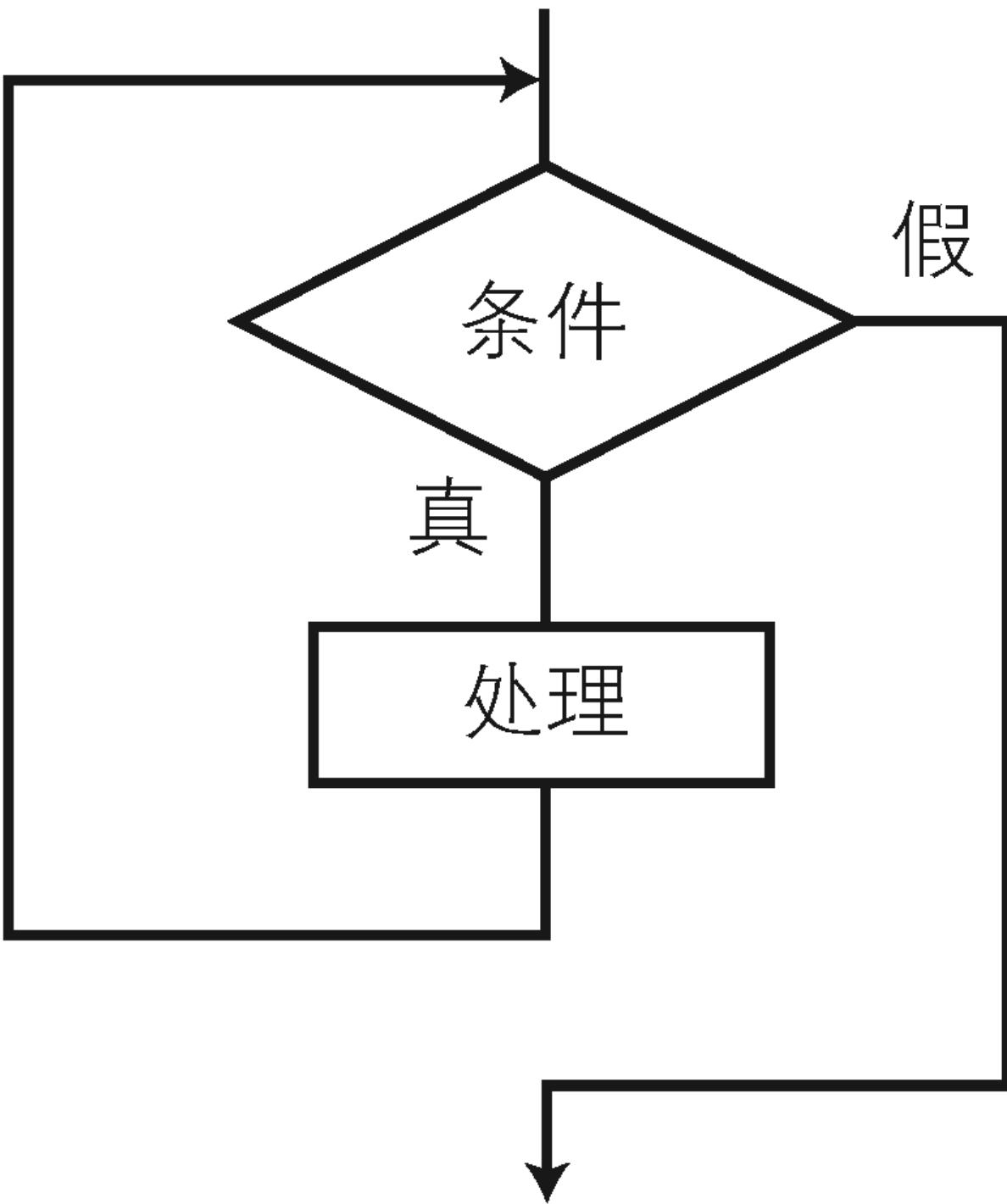
本例中，循环遍历各数组的元素，并各自将其输出。

## 6.7 while 语句

不管哪种类型的循环，`while` 语句都可以胜任，`while` 语句的结构如下：

```
while 条件 do  
    希望循环的处理  
end
```

※ 可以省略 `do`



这几行程序的意思就是，只要条件成立，就会不断地重复循环处理。我们来看看下面的示例（代码清单 6.6）。

#### 代码清单 6.6 while.rb

```
i = 1
while i < 3
  puts i
  i += 1
end
```

#### 执行示例

```
> ruby while.rb
1
2
```

本例为什么会得出这样的结果呢。首先，程序将 1 赋值给变量 `i`，这时 `i` 的值为 1。接下来 `while` 语句循环处理以下内容：

1. 执行 `i < 3` 的比较。
2. 比较结果为真（也就是 `i` 比 3 小）时，程序执行 `puts i` 和 `i += 1`。比较结果为假（也就是 `i` 大于等于 3）时，程序跳出 `while` 循环，不执行任何内容。
3. 返回 1 处理。

首次循环，由于 `i` 的初始值为 1，因此程序执行 `puts 1`。第 2 次循环，`i` 的值为 2，比 3 小，因此程序执行 `puts 2`。当程序执行到第 3 次循环，`i` 的值为 3，比 3 小的条件不成立，也就是说比较结果为假，因此，程序跳出 `while` 循环，并终止所有处理。

我们再来写一个使用 `while` 语句的程序。

把之前使用 `for` 语句写的程序（代码清单 6.4），改写为使用 `while` 语句程序（代码清单 6.7）。

#### 代码清单 6.7 while2.rb

```
sum = 0
i = 1
while i <= 5
  sum += i
  i += 1
end
puts sum
```

这时与使用 `for` 语句的程序有细微的区别。首先，变量 `i` 的条件指定方式不一样。`for` 语句的例子通过 `1..5` 指定条件的范围。`while` 语句使用比较运算符 `<=`，指定“`i` 小于等于 5 时（循环）”的条件。

另外，`i` 的累加方式也不一样。`while` 语句的例子在程序里直接写出了 `i` 是如何进行加 1 处理的——`i += 1`。而 `for` 语句的例子并不需要在程序里直接写如何对 `i` 进行操作，自动在每次循环后对 `i` 进行加 1 处理。

就像这个例子一样，只要 `for` 语句能实现的循环，我们没必要特意将它改写为 `while` 语句。程序代码清单 6.8 是使用 `while` 语句反而会更便于理解的一个例子。

#### 代码清单 6.8 while3.rb

```
sum = 0
i = 1
while sum < 50
  sum += i
  i += 1
end
puts sum
```

在这个例子里，作为循环条件的不是变量 `i`，而是变量 `sum`。循环条件现在变为“`sum` 小于 50 时执行循环处理”。一般来说，不通过计算，我们并不知道 `i` 的值为多少时 `sum` 的值才会超过 50，因此这种情况下使用 `for` 语句的循环反而会让程序变得难懂。

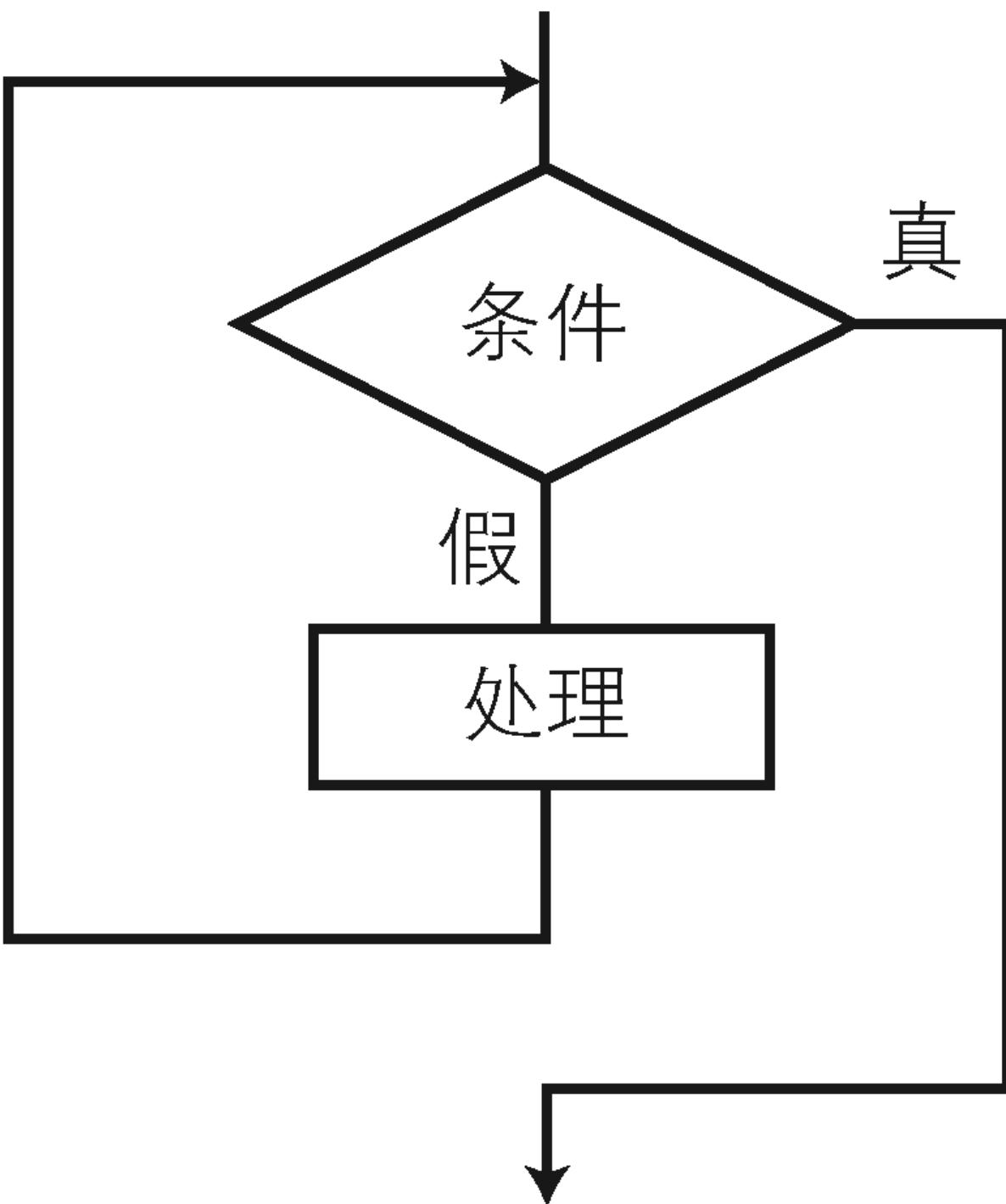
有时 `for` 语句写的程序易懂，有时候 `while` 语句写的程序易懂。我们会在本章的最后说明，如何区别使用 `for` 语句和 `while` 语句。

## 6.8 until 语句

与 `if` 语句相对的有 `unless` 语句，同样地，与 `while` 语句相对的有 `until` 语句。`until` 语句的结构与 `while` 语句完全一样，只是条件判断刚好相反，不满足条件时才执行循环处理。换句话说，`while` 语句是一直执行循环处理，直到条件不成立为止；`until` 语句是一直执行循环处理，直到条件成立为止。

```
until 条件 do
  希望循环的处理
end
```

※ 可以省略 `do`



代码清单 6.9 是使用了 `until` 语句的程序。

代码清单 6.9 until.rb

```

sum = 0
i = 1
until sum >= 50
  sum += i
  i+= 1
end
puts sum

```

本例是将使用 `while` 语句的程序（代码清单 6.8）用 `until` 语句改写了，与 `while` 语句所使用的条件刚好相反。

其实，在 `while` 语句的条件上使用表示否定的运算符 `!`，也能达到和 `until` 语句相同的效果。

代码清单 6.10 while\_not.rb

```

sum = 0
i = 1
while !(sum >= 50)
  sum += i
  i += 1
end
puts sum

```

虽然可以使用 `while` 语句的否定形式代替 `until` 语句。但是，有时对一些比较复杂的条件表达式使用否定，反而会不直观，影响程序理解，在这种情况下，我们应该考虑使用 `until` 语句。

## 6.9 each 方法

`each` 方法将对象集合里的对象逐个取出，这与 `for` 语句循环取出数组元素非常相似。实际上，我们可以非常简单地将使用 `for` 语句的程序（代码清单 6.5）改写为使用 `each` 方法的程序（代码清单 6.11）。

代码清单 6.11 `each_names.rb`

```
names = ["awk", "Perl", "Python", "Ruby"]
names.each do |name|
  puts name
end
```

`each` 方法的结构如下：

```
对象.each do |变量|
  希望循环的处理
end
```

在说明 `times` 方法我们曾提到过，块的 `do ~ end` 部分可换成 `{ ~ }`。

```
对象.each {|变量|
  希望循环的处理
}
```

这与下面的程序的效果是几乎一样。

```
for 变量 in 对象
  希望循环的处理
end
```

在 Ruby 内部，`for` 语句是用 `each` 方法来实现的。因此，可以使用 `each` 方法的对象，同样也可以指定为 `for` 语句的循环对象。

在介绍 `for` 语句时我们举过使用范围对象的例子（代码清单 6.4），我们试着用 `each` 方法改写一下。

代码清单 6.12 `each.rb`

```
sum = 0
(1..5).each do |i|
  sum = sum + i
end
puts sum
```

像本例这样，我们可以轻松互相改写两种用法。那到底什么时候该用 `for` 语句，什么时候该用 `each` 方法呢，我们将在本章的最后讨论这个问题。

## 6.10 loop 方法

还有一种循环的方法，没有终止循环条件，只是不断执行循环处理。Ruby 中的 `loop` 就是这样的循环方法。

```
loop do
  print "Ruby"
end
```

执行上面的程序后，整个屏幕会不停的输出文字 `Ruby`。为了避免这样的情况发生，在实际使用 `loop` 方法时，我们需要用到接下来将要介绍的 `break`，使程序可以中途跳出循环。

备注 程序不小心执行了死循环时，我们可以使用 `CTRL + c` 来强行终止程序。

## 6.11 循环控制

在进行循环处理的途中，我们可以控制程序马上终止循环，或者跳到下一个循环等。Ruby 提供了如下表（表 6.1）所示的三种控制循环的命令。

表 6.1 循环控制命令

命令	用途
<code>break</code>	终止程序，跳出循环

next	跳到下一次循环
redo	在相同的条件下重复刚才的处理

在控制循环的命令中，可能不太容易区分使用 `next` 和 `redo`。借着下面的例子（代码清单 6.13），我们来介绍一下如何区分这三种循环命令。

#### 代码清单 6.13 `break_next_redo.rb`

```

1: puts "break 的例子:"
2: i = 0
3: ["Perl", "Python", "Ruby", "Scheme"].each do |lang|
4:   i += 1
5:   if i == 3
6:     break
7:   end
8:   p [i, lang]
9: end
10:
11: puts "next 的例子:"
12: i = 0
13: ["Perl", "Python", "Ruby", "Scheme"].each do |lang|
14:   i += 1
15:   if i == 3
16:     next
17:   end
18:   p [i, lang]
19: end
20:
21: puts "redo 的例子:"
22: i = 0
23: ["Perl", "Python", "Ruby", "Scheme"].each do |lang|
24:   i += 1
25:   if i == 3
26:     redo
27:   end
28:   p [i, lang]
29: end

```

我们来看看本例中的 `break`、`next`、`redo` 有什么不同。程序由三部分组成，除了 `break`、`next`、`redo` 这三部分的代码外，其他地方都是相同的。下面是执行后的结果。

#### 执行示例

```

> ruby break_next_redo.rb
break 的例子:
[1, "Perl"]
[2, "Python"]
next 的例子:
[1, "Perl"]
[2, "Python"]
[4, "Scheme"]
redo 的例子:
[1, "Perl"]
[2, "Python"]
[4, "Ruby"]
[5, "Scheme"]

```

#### 6.11.1 `break`

`break` 会终止全体程序。在代码清单 6.13 中，`i` 为 3 时，程序会执行第 6 行的 `break`（图 6.1）。执行 `break` 后，程序跳出 `each` 方法循环，前进至程序的第 10 行。因此，程序没有输出 `Ruby` 和 `Scheme`。

```

i = 0
["Perl", "Python", "Ruby", "Scheme"].each do |lang|
  i += 1
  if i == 3
    break
  end
  p [i, lang]
end

```

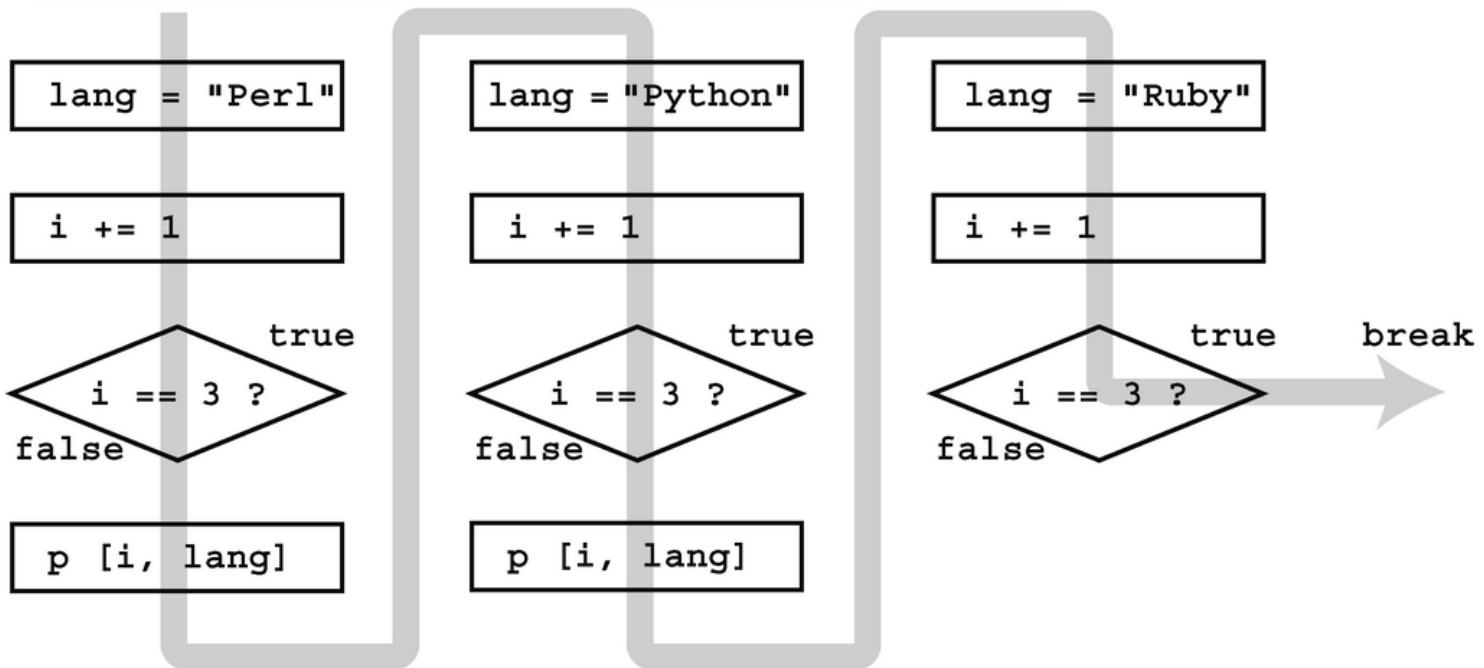


图 6.1 `break`

我们再来介绍一个关于 `break` 的例子。程序代码清单 6.14 对第 3 章的 `simple_grep.rb` 做了点小修改，使程序最多只能输出 10 行匹配到的内容。匹配的时候，累加变量 `matches`，当达到 `max_matches` 时，程序就会终止 `each_line` 方法的循环。

代码清单 6.14 `ten_lines_grep.rb`

```

pattern = Regexp.new(ARGV[0])
filename = ARGV[1]
max_matches = 10      # 输出的最大行数
matches = 0          # 已匹配的行数
file = File.open(filename)
file.each_line do |line|
  if matches >= max_matches
    break
  end
  if pattern =~ line
    matches += 1
    puts line
  end
end
file.close

```

### 6.11.2 `next`

使用 `next` 后，程序会忽略 `next` 后面的部分，跳到下一个循环开始的部分。在代码清单 6.13 中，`i` 为 3 时在执行第 16 行的 `next` 后，程序前进到 `each` 方法的下个循环（图 6.2）。也就是说，将 `Scheme` 赋值给 `lang`，并执行 `i += 1`。因此，程序并没有输出 `Ruby`，而是输出了 `Scheme`。

```

i = 0
["Perl", "Python", "Ruby", "Scheme"].each do |lang|
  i += 1
  if i == 3
    redo
  end
  p [i, lang]
end

```

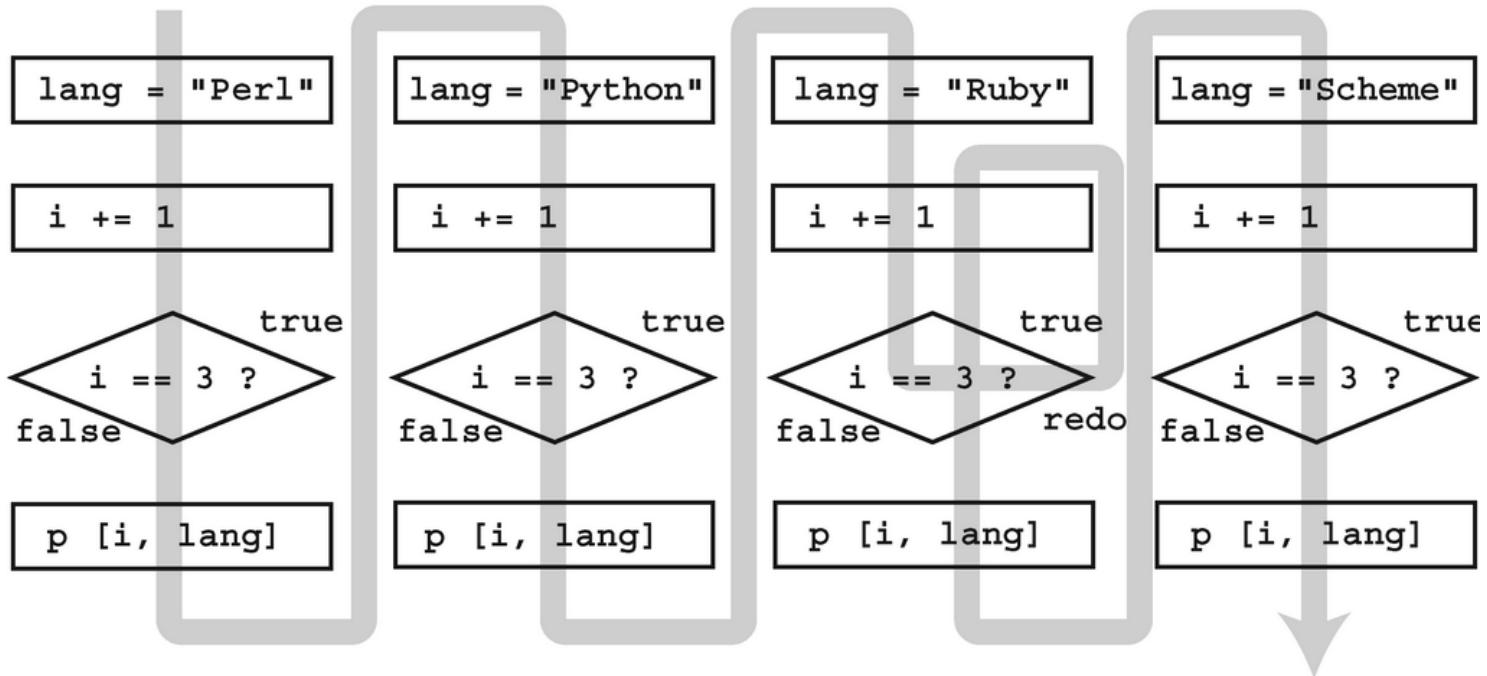


图 6.2 next

我们再来看看另外一个 `next` 的例子（代码清单 6.15）。程序逐行读取输入的内容，忽略空行或者以 # 开头的行，原封不动地输出除此以外所有行的内容。

执行以下命令后，我们会得到去掉 `fact.rb`（代码清单 6.16）的注释和空行后的 `stripped_fact.rb`（代码清单 6.17）。

```
> ruby strip.rb fact.rb > stripped_fact.rb
```

代码清单 6.15 strip.rb

```

file = File.open(ARGV[0])
file.each_line do |line|
  next if /^\s*$/ =~ line # 空行
  next if /^#/ =~ line      # 以“#”开头的行
  puts line
end
file.close

```

代码清单 6.16 fact.rb

```

# 求10 的阶乘
ans = 1
for i in 1..10
  ans *= i
end

# 输出
puts "10! = #{ans}"

```

代码清单 6.17 stripped\_fact.rb

```

ans = 1
for i in 1..10
  ans *= i
end

```

```
puts "10! = #{ans}"
```

### 6.11.3 redo

`redo` 与 `next` 非常像，与 `next` 的不同之处是，`redo` 会再执行一次相同的循环。

在代码清单 6.13 中，与 `next` 时的情况不同，`redo` 会输出 Ruby。这是由于，`i` 为 3 时就执行了第 26 行的 `redo`，程序只是返回循环的开头，也就是从程序的第 24 行的 `i += 1` 部分开始重新再执行处理，所以 `lang` 的值并没有从 Ruby 变为 Scheme。由于重复执行了 `i += 1`，`i` 的值变为 4，这样 `if` 语句的条件 `i == 3` 就不成立了，`redo` 也不会再执行了，程序顺利成章地输出了 [4, "Ruby"] 以及 [5, "Scheme"]（图 6.3）。

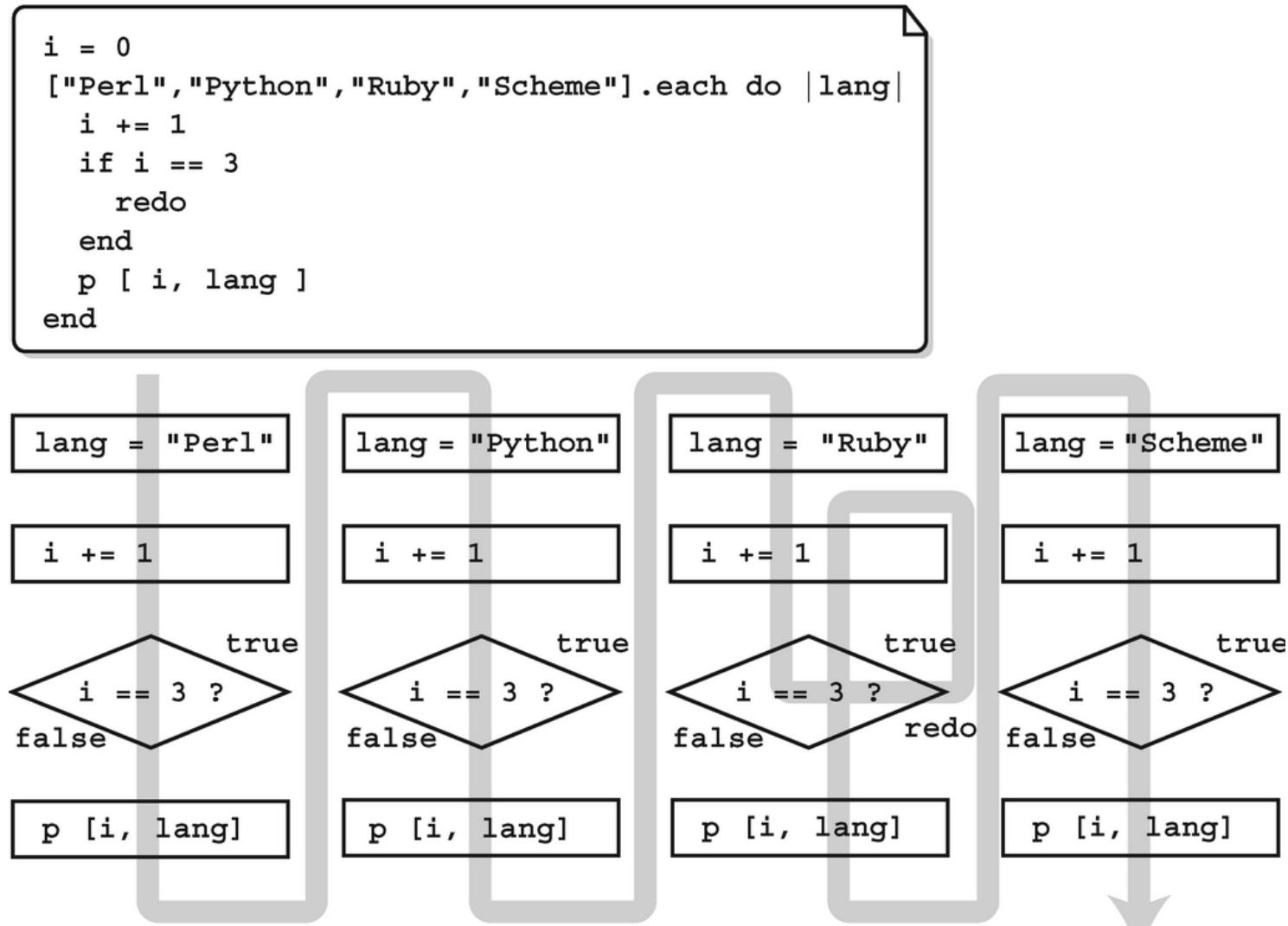


图 6.3 redo

另外，大家要注意 `redo` 的使用方法，稍不留神就会在同样的条件下，不断地重复处理，陷入死循环中。

`break`、`next` 和 `redo` 中，一般比较常用是 `break` 和 `next`。大家应该熟练掌握这两个命令的用法。即使是 Ruby 默认提供的库里面，实际上也很难找到 `redo` 的踪影，所以当我们在希望使用 `redo` 时，应该好好考虑是否真的有必要使用 `redo`。

## 6.12 总结

本章我们主要介绍了循环时所用的语句以及方法。

如果纯粹只考虑实现循环功能，任何种类的循环都可以用 `while` 语句实现。讲得极端一点，其他循环语句和方法根本也没存在的必要。即便如此，那为什么 Ruby 还提供了那么丰富的循环语句和方法呢。这是因为，程序并不只是单纯为了实现功能，同时还应该使代码更便于读写，使人更容易理解。

为了使大家更快地熟悉循环的用法，我们在最初介绍循环语句和方法的一览表中加上“主要用途”一栏（表 6.2），以供大家参考。

表 6.2 循环语句、方法及其主要用途

	主要用途
<code>times</code> 方法	确定循环次数时使用
<code>for</code> 语句	从对象取出元素时使用（ <code>each</code> 的语法糖）

<code>while</code> 语句	希望自由指定循环条件时使用
<code>until</code> 语句	使用 <code>while</code> 语句使循环条件变得难懂时使用
<code>each</code> 方法	从对象取出元素时使用
<code>loop</code> 方法	不限制循环次数时使用

## 备注

语法糖 (syntax sugar)，是指一种为了照顾一般人习惯而产生的特殊语法<sup>1</sup>。例如，使用常规的语法调用方法，那么加法运算应该写成 `3.add(2)`。但是对于一般人来说，写成 `3 + 2` 会更直观，更简明易懂。

语法糖并不会加强编程语言的任何功能，但是对于提高程序的易读性来讲是不可或缺的。

<sup>1</sup>即一种符合人的思维模式、工作习惯等，便于理解的“甜蜜”的语法。——译者注

以上都是笔者个人意见，仅供大家参考。

确定循环次数时循环处理使用 `times` 方法，除此以外的大部分循环处理，根据情况选择 `while` 语句或者 `each` 方法，一般也能写出直观易懂的程序。请大家先熟练掌握这三种循环方法。

## 专栏

### `do~end` 与 `{~}`

在 `times` 方法的示例中，我们介绍了块的两种写法，`do ~ end` 与 `{ ~ }`。从执行效果来看，两种方法虽然没有太大区别，但一般我们会遵守以下这个约定俗成的编码规则：

- 程序是跨行写的时候使用 `do ~ end`
- 程序写在 1 行的时候用 `{ ~ }`

以 `times` 方法来举例，会有以下两种写法。

```
10.times do |i|
  puts i
end
```

或者，

```
10.times{|i| puts i}
```

刚开始大家可能会有点不习惯。我们可以这样理解，`do ~ end` 表示程序要执行内容是多个处理的集合，而 `{ ~ }` 则表示程序需要执行的处理只有一个，即把整个带块的方法看作一个值。

如果用把 `do ~ end` 代码合并在一起，程序会变成下面这样：

```
10.times do |i| puts i end
```

以上写法，怎么看都给人一种很难断句的感觉。虽然实际上使用哪种写法都不会影响程序的运行，但在刚开始编写程序时，还是建议大家先遵守这个编码规则。

# 第7章 方法

方法是由对象定义的与该对象相关的操作。在 Ruby 中，对象的所有操作都被封装成方法。

## 7.1 方法的调用

首先，让我们温习一下如何调用方法。

### 7.1.1 简单的方法调用

调用方法的语法如下所示：

```
对象.方法名(参数 1, 参数 2, ..., 参数 n)
```

以对象开头，中间隔着句点，后面接着是方法名，方法名后面是一排并列的用 () 括起来的参数。不同的方法定义的参数个数和顺序也都不一样，调用方法时必须按照定义来指定参数。另外，调用方法时 () 是可以省略的。

上面的对象被称为接收者（receiver）。在面向对象的世界中，调用方法被称为“向对象发送消息（message）”，调用的结果就是“对象接收（receive）了消息”（图 7.1）。也就是说，方法的调用就是把几个参数连同消息一起发送给对象的过程。

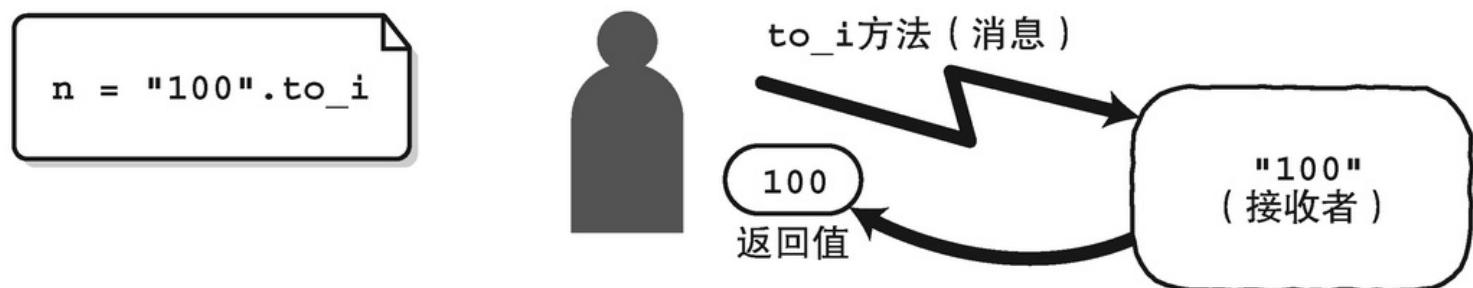


图 7.1 将消息发送给对象

### 7.1.2 带块的方法调用

正如第 6 章提到的 `each` 方法、`loop` 方法，方法本身可以与伴随的块一起被调用。这种与块一起被调用的方法，我们称之为带块的方法。

带块的方法的语法如下：

```
对象.方法名(参数, ...) do |变量 1, 变量 2, ...|
  块内容
end
```

`do ~ end` 这部分就是所谓的块。除 `do ~ end` 这一形式外，我们也可以用 `{~}` 将块改写为其他形式：

```
对象.方法名(参数, ...){|变量 1, 变量 2, ...|
  块内容
}
```

**备注** 使用 `do ~ end` 时，可以省略把参数列表括起来的 `()`。使用 `{~}` 时，只有在没有参数的时候才可以省略 `()`，有一个以上的参数时就不能省略。

在块开头的 `| ~ |` 部分中指定的变量称为块变量。在执行块的时候，块变量由方法传到块内部。不同的方法对应的块变量的个数、值也都不一样。之前介绍过的 `times` 方法有一个块变量，执行块时，方法会从 0 开始依次把循环次数赋值给块变量（代码清单 7.1）。

#### 代码清单 7.1 times\_with\_param.rb

```
5.times do |i|
  puts "第#{i} 次循环。"
end
```

#### 执行示例

```
> ruby times_with_param.rb
第0 次循环。
第1 次循环。
第2 次循环。
第3 次循环。
第4 次循环。
```

### 7.1.3 运算符形式的方法调用

Ruby 中有些方法看起来很像运算符。四则运算等的二元运算符、`-`（负号）等的一元运算符、指定数组、散列的元素下标的`[]`等，实际上都是方法。

- `obj + arg1`
- `obj =~ arg1`
- `-obj`
- `!obj`
- `obj[arg1]`
- `obj[arg1] = arg2`

这些虽然与一般的方法调用的语法有点不一样，但我们可以将`obj`理解为接收者，将`arg1`、`arg2`理解为参数，这样一来，它们就又是我们所熟知的方法了。我们可以自由定义这种运算符形式的方法。

备注 在用方法实现的运算符中，有的可以自由地重新定义，有的则不能改变默认的定义。关于运算符，我们会在第9章中再详细说明。

## 7.2 方法的分类

根据接收者种类的不同，Ruby 的方法可分为3类：

- 实例方法
- 类方法
- 函数式方法

下面，我们就来说明一下这3种方法。

### 7.2.1 实例方法

实例方法是最常用的方法。假设有一个对象（实例），那么以这个对象为接收者的方法就被称为实例方法。

下面是实例方法的一些例子：

```
p "10, 20, 30, 40".split(",")      #=> ["10", "20", "30", "40"]
p [1, 2, 3, 4].index(2)           #=> 1
p 1000.to_s                      #=> "1000"
```

在本例中，从上到下，分别以字符串、数组、数值对象为接收者。

对象能使用的实例方法是由对象所属的类决定的。调用对象的实例方法后，程序就会执行对象所属类预先定义好的处理。

虽然相同名称的方法执行的处理大多都是一样的，但具体执行的内容则会根据对象类型的不同而存在差异。例如，几乎所有的对象都有`to_s`方法，这是一个把对象内容以字符串形式输出的方法。然而，虽然都是字符串形式，但在数值对象与时间对象的情况下，字符串形式以及字符串的创建方法却都不一样。

```
p 10.to_s            #=> "10"
p Time.now.to_s     #=> "2013-03-18 03:17:02 +900"
```

### 7.2.2 类方法

接收者不是对象而是类本身时的方法，我们称之为类方法。例如，我们在创建对象的时候就使用到了类方法。

```
Array.new          # 创建新的数组
File.open("some_file")  # 创建新的文件对象
Time.now          # 创建新的 Time 对象
```

此外，不直接对实例进行操作，只是对该实例所属的类进行相关操作时，我们也会用到类方法。例如，修改文件名的时候，我们就会使用文件类的类方法。

```
File.rename(oldname, newname)  # 修改文件名
```

类方法也有运算符的形式。

```
File.rename(oldname, newname)  # 修改文件名
```

调用类方法时，可以使用`::`代替`.`。在Ruby语法中，两者所代表的意思是一样的。

关于类方法，我们会在第8章中再进行详细的说明。

### 7.2.3 函数式方法

没有接收者的方法，我们称之为函数式方法。

虽说是没有接收者，但并不表示该方法就真的没有可作为接收者的对象，只是在函数式方法这个特殊情况下，可以省略接收者而已。

```
print "hello!"      # 在命令行输出字符串  
sleep(10)         # 在指定的时间内睡眠，终止程序
```

函数式方法的执行结果，不会根据接收者的状态而发生变化。程序在执行 `print` 方法以及 `sleep` 方法的时候，并不需要知道接收者是谁。反过来说，不需要接收者的方法就是函数式方法。

## 专栏

### 方法的标记法

接下来，我们来介绍一下 Ruby 帮助文档中方法名的标记方法。标记某个类的实例方法时，就像 `Array#each`、`Array#inject` 这样，可以采用以下标记方法：

`类名 # 方法名`

请注意，这只是写帮助文档或者说明时使用的标记方法，程序里面这么写是会出错的。

另外，类方法的标记方法，就像 `Array.new` 或者 `Array::new` 这样，可以采用下面两种写法：

`类名 . 方法名`

`类名 :: 方法名`

这和实际的程序语法是一致的。

## 7.3 方法的定义

定义方法的一般语法如下：

```
def 方法名(参数 1, 参数 2, ...)  
希望执行的处理  
end
```

方法名可由英文字母、数字、下划线组成，但不能以数字开头。

### 代码清单 7.2 hello\_with\_name.rb

```
def hello(name)  
  puts "Hello, #{name}."  
end  
  
hello("Ruby")
```

**备注** 虽然在说明如何定义实例方法或类方法之前，应该先说明如何定义类，但关于类的定义我们还未说明。因此，现在我们只需掌握一点，即定义了方法后，就能像省略接收者的函数式方法那样调用方法了。

通过 `hello` 方法中的 `name` 变量，我们就可以引用执行时传进来的参数。该程序的参数为字符串 `Ruby`，执行结果如下：

#### 执行示例

```
> ruby hello_with_name.rb  
Hello, Ruby.
```

也可以指定默认值给参数（代码清单 7.3）。参数的默认值，是指在没有指定参数的情况下调用方法时，程序默认使用的值。定义方法时，用参数名 = 值这样的写法定义默认值。

### 代码清单 7.3 hello\_with\_default.rb

```
def hello(name="Ruby")  
  puts "Hello, #{name}."  
end  
  
hello()          # 省略参数的调用  
hello("Newbie") # 指定参数的调用
```

#### 执行示例

```
> ruby hello_with_default.rb  
Hello, Ruby.  
Hello, Newbie.
```

方法有多个参数时，从参数列表的右边开始依次指定默认值。例如，希望省略三个参数中的两个时，就可以指定右侧两个参数为默认值。

```
def func(a, b=1, c=3)
|
end
```

请注意只省略左边的参数或者中间的某个参数是不行的。

### 7.3.1 方法的返回值

我们可以用 `return` 指定方法的返回值。

`return` 值

下面我们来看看求立方体体积的例子。参数 `x`、`y`、`z` 分别代表长、宽、高。`x * y * z` 的结果就是方法的返回值。

```
def volume(x, y, z)
  return x * y * z
end

p volume(2, 3, 4)    #=> 24
p volume(10, 20, 30) #=> 6000
```

可以省略 `return` 语句，这时方法的最后一个表达式的结果就会成为方法的返回值。下面我们再通过求立方体的表面积这个例子，来看看如何省略 `return`。这里，`area` 方法的最后一行的 `(xy + yz + zx) * 2` 的结果就是方法的返回值。

```
def area(x, y, z)
  xy = x * y
  yz = y * z
  zx = z * x
  (xy + yz + zx) * 2
end

p area(2, 3, 4)    #=> 52
p area(10, 20, 30) #=> 2200
```

方法的返回值，也不一定是程序最后一行的执行结果。例如，在下面的程序中，比较两个值的大小，并返回较大的值。`if` 语句的结果就是方法的返回值，即 `a > b` 的结果为真时，`a` 为返回值；结果为假时，则 `b` 为返回值。

```
def max(a, b)
  if a > b
    a
  else
    b
  end
end

p max(10, 5)    #=> 10
```

因为可以省略，所以有些人就会感觉好像没什么机会用到 `return`，但是有些情况下我们会希望马上终止程序，这时 `return` 就派上用场了。用 `return` 语句改写的 `max` 方法如下所示，大家可以对比一下和之前有什么异同。

```
def max(a, b)
  if a > b
    return a
  end
  return b    # 这里的return 可以省略
end

p max(10, 5)    #=> 10
```

如果省略 `return` 的参数，程序则返回 `nil`。方法的目的是程序处理，所以 Ruby 允许没有返回值的方法。Ruby 中有很多返回值为 `nil` 的方法，第 1 章中介绍的 `print` 方法就是其中一。

`print` 方法只输出参数的内容，返回值为 `nil`。

```
p print("1:")    #=> 1:nil
#  (显示print 方法的输出结果1: 与p 方法的输出结果nil)
```

### 7.3.2 定义带块的方法

之前我们已经介绍过带块的方法，下面就来介绍一下如何定义带块的方法。

首先我们来实现 `myloop` 方法，它与利用块实现循环的 `loop` 方法的功能是一样的。

#### 代码清单 7.4 myloop.rb

```
def myloop
  while true
    yield          # 执行块
  end

  num = 1          # 初始化num
  myloop do
    puts "num is #{num}" # 输出num
    break if num > 100   # num 超过 100 时跳出循环
    num *= 2           # num 乘2
  end
```

这里第一次出现了 `yield`，`yield` 是定义带块的方法时最重要的关键字。调用方法时通过块传进来的处理会在 `yield` 定义的地方执行。

执行该程序后，`num` 的值就会像 1、2、4、8、16……这样 2 倍地增长下去，直到超过 100 时程序跳出 `myloop` 方法。

#### 执行示例

```
> ruby myloop.rb
num is 1
num is 2
num is 4
num is 8
num is 16
num is 32
num is 64
num is 128
```

本例的程序中没有参数，如果 `yield` 部分有参数，程序就会将其当作块变量传到块里。块里面最后的表达式的值既是块的执行结果，同时也可作为 `yield` 的返回值在块的外部使用。

关于带块的方法的使用方法，我们会在第 11 章中再详细说明。

### 7.3.3 参数个数不确定的方法

像下面的例子那样，通过用“`*` 变量名”的形式来定义参数个数不确定的方法，Ruby 就可以把所有参数封装为数组，供方法内部使用。

```
def foo(*args)
  args
end

p foo(1, 2, 3)    #=> [1, 2, 3]
```

至少需要指定一个参数的方法可以像下面这样定义：

```
def meth(arg, *args)
  [arg, args]
end

p meth(1)      #=> [1, []]
p meth(1, 2, 3) #=> [1, [2, 3]]
```

所有不确定的参数都被作为数组赋值给变量 `args`。“`*` 变量名”这种形式的参数，只能在方法定义的参数列表中出现一次。只确定首个和最后一个参数名，并省略中间的参数时，可以像下面这样定义：

```
def a(a, *b, c)
  [a, b, c]
end

p a(1, 2, 3, 4, 5)    #=> [1, [2, 3, 4], 5]
p a(1, 2)              #=> [1, [], 2]
```

### 7.3.4 关键字参数

关键字参数是 Ruby 2.0 中的新特性。

在目前为止介绍过的方法定义中，我们都需要定义调用方法时的参数个数以及调用顺序。而使用关键字参数，就可以将参数名与参数值成对地传给方法内部

使用。

使用关键字参数定义方法的语法如下所示：

```
def 方法名(参数 1: 参数 1 的值, 参数 2: 参数 2 的值, ...)  
    希望执行的处理  
end
```

除了参数名外，使用“参数名：值”这样的形式还可以指定参数的默认值。用关键字参数改写计算立方体表面积的 area 方法的程序如下所示：

```
def area2(x: 0, y: 0, z: 0)  
    xy = x * y  
    yz = y * z  
    zx = z * x  
    (xy + yz + zx) * 2  
end  
  
p area2(x: 2, y: 3, z: 4)      #=> 52  
p area2(z: 4, y: 3, x: 2)      #=> 52 (改变参数的顺序)  
p area2(x: 2, z: 3)            #=> 12 (省略y)
```

这个方法有参数 x、y、z，各自的默认值都为 0。调用该方法时，可以像 x: 2 这样，指定一对实际的参数名和值。在用关键字参数定义的方法中，每个参数都指定了默认值，因此可以省略任何一个。而且，由于调用方法时也会把参数名传给方法，所以参数顺序可以自由地更改。

不过，如果把未定义的参数名传给方法，程序就会报错，如下所示：

```
area2(foo: 0)      #=> 错误: unknown keyword: foo (ArgumentError)
```

为了避免调用方法时因指定了未定义的参数而报错，我们可以使用“`**` 变量名”的形式来接收未定义的参数。下面这个例子的方法中，除了关键字参数 x、y、z 外，还定义了 `**arg` 参数。参数 arg 会把参数列表以外的关键字参数以散列对象的形式保存。

```
def meth(x: 0, y: 0, z: 0, **args)  
    [x, y, z, args]  
end  
  
p meth(z: 4, y: 3, x: 2)      #=> [2, 3, 4, {}]  
p meth(x: 2, z: 3, v: 4, w: 5) #=> [2, 0, 3, {:v=>4, :w=>5}]
```

#### • 关键字参数与普通参数的搭配使用

关键字参数可以与普通参数搭配使用。

```
def func(a, b: 1, c:2)  
    |  
end
```

上述这样定义时，`a` 为必须指定的普通参数，`b`、`c` 为关键字参数。调用该方法时，可以像下面这样，首先指定普通参数，然后是关键字参数。

```
func(1, b: 2, c: 3)
```

#### • 用散列传递参数

调用用关键字参数定义的方法时，可以使用以符号作为键的散列来传递参数。这样一来，程序就会检查散列的键与定义的参数名是否一致，并将与散列的键一致的参数名传递给方法。

```
def area2(x: 0, y: 0, z: 0)  
    xy = x * y  
    yz = y * z  
    zx = z * x  
    (xy + yz + zx) * 2  
end  
  
args1 = {x: 2, y: 3, z: 4}  
p area2(args1)              #=> 52  
  
args2 = {x: 2, z: 3}        #=> 省略y  
p area2(args2)              #=> 12
```

### 7.3.5 关于方法调用的一些补充

下面补充说明一下调用方法时传递参数的方法。

## • 把数组分解为参数

将参数传递给方法时，我们也可以先分解数组，然后再将分解后的数组元素作为参数传递给方法。在调用方法时，如果以“`* 数组`”这样的形式指定参数，这时传递给方法的就不是数组本身，而是数组的各元素被按照顺序传递给方法。但需要注意的是，数组的元素个数必须要和方法定义的参数个数一样。

```
def foo(a, b, c)
  a + b + c
end

p foo(1, 2, 3)      #=> 6

args1 = [2, 3]
p foo(1, *args1)   #=> 6

args2 = [1, 2, 3]
p foo(*args2)      #=> 6
```

## • 把散列作为参数传递

我们用 `{ ~ }` 这样的形式来表示散列的字面量（literal）。将散列的字面量作为参数传递给方法时可以省略 `{}`。

```
def foo(arg)
  arg
end

p foo({ "a"=>1, "b"=>2})    #=> {"a"=>1, "b"=>2}
p foo("a"=>1, "b"=>2)        #=> {"a"=>1, "b"=>2}
p foo(a: 1, b:2)              #=> {:a=>1, :b=>2}
```

当虽然有多个参数，但只将散列作为最后一个参数传递给方法时，可以使用下面的写法：

```
def bar(arg1, arg2)
  [arg1, arg2]
end

p bar(100, {"a"=>1, "b"=>2})    #=> [100, {"a"=>1, "b"=>2}]
p bar(100, "a"=>1, "b"=>2)        #=> [100, {"a"=>1, "b"=>2}]
p bar(100, a: 1, b: 2)            #=> [100, {:a=>1, :b=>2}]
```

第3种形式是把符号作为键的散列传递给方法，与使用关键字参数调用方法的形式一模一样。其实，关键字参数就是模仿这种将散列作为参数传递的写法而设计出来的。使用关键字参数定义方法，既可以对键进行限制，又可以定义参数的默认值。因此建议大家在实际编写程序的时候多尝试使用关键字参数。

## 专栏

### 如何书写简明易懂的程序

程序不只是为了让计算机理解、执行而存在的，还要能便于人们读写。即使是实现相同功能的程序，有的可能通俗易懂，有的却晦涩难懂。程序是否易懂，除了与程序的设计和架构有关外，程序的外观也起着很重要的作用。而通过注意下面列举的3点，就可以使程序变得更漂亮。

- 换行和；
- 缩进（`indent`）
- 空白

下面，我们就依次来看看它们到底有什么魔法可以使我们的程序变漂亮。

- 换行和；

Ruby语法的特征之一就是换行可作为语句结束的标志使用。

除了可以使用换行表示语句结束外，我们还可以使用`;`。这样一来，一行程序里就可以写多条语句，例如，

```
str = "hello"; print str
```

这样的写法与下面的写法具有一样的效果。

```
str = "hello"
print str
```

使用该语法时，把换行看作是一种自然的语句间隔，会更加便于我们读写程序。比起把多个操作都写在一行里，适当的换行是书写简明易懂的程序的第一步。

然而，过多地使用 ;，往往会使程序变得难以读懂。因此，在使用 ; 之前，应问问自己是不是非使用不可。经过仔细的考量，如果觉得使用后会使程序变得易懂的话再使用也不迟。顺便说一下，笔者平时也很少会用到 ;。

- 缩进

缩进，也就是使文字“后退”。这是在程序行的开头输入适当的空白字符来强调程序整体感的一种书写方法。在本书中，我们用两个空白表示一个缩进1。

在下面的例子中，为了表示 `print` 方法的那两行程序是 `if ~ end` 的内部处理，程序进行了缩进。

```
if a == 1
```

```
    print message1  
    print message2
```

```
end
```

插入循环等的时候，会使用更深的缩进。这样一来，语句和循环的对应关系就会变得一目了然。

```
while a < 100
```

```
    while b < 20  
        b = b + 1  
        print b  
    end  
    a = a + 1  
    print a
```

```
end
```

下面列举了一些需要使用缩进的情况。

- 条件分支

```
if a > 0
```

```
    some_method()
```

```
else
```

```
    other_method()
```

```
end
```

- 循环

```
while i < 10
```

```
    method()  
    i = i - 1
```

```
end
```

- 块

```
some_value.each do |i|
```

```
    i.method()
```

```
end
```

- 方法、类等的定义

```
def foo
```

```
    print "hello"
```

```
end
```

使用缩进时，需要遵守以下事项。

- 不要突然使用缩进

x=10 y=20

```
z=30      # 不好的例子
```

- 确保缩进的整齐

```
if (foo)
```

```
  if (bar)
    if (buz)      # 缩进得太多了
    end
end
```

```
end
```

- 空白

空白存在于程序的各个角落。使用空白时，我们需要注意以下事项。

- 确保空白长度整齐，保持良好的平衡感

特别是，如果在运算符前后使用的空白长度不一样，程序就很可能出现莫名其妙的错误。例如，计算 `a` 加 `b` 时，不同的空白写法，得到的可能是完全不一样的结果。

`a+b` ◦ 好的写法

`a + b` ◦ 好的写法

`a +b` ✗ 不好的写法

`a+ b` ✗ 不好的写法

`a +b` 表示调用参数为 `+b` 的方法 `a`，整个表达式容易被误认为 `a(+b)`，因此不是好的写法。可见，在 `+` 前后书写空白时，要确保平衡。

- 良好的编码风格

养成良好的编码风格，首先应该从参考其他人的 Ruby 程序，并模仿他们的写法开始。编写程序的技巧也好，良好的编码风格也好，想要熟练掌握，都要大量阅读其他人写的程序。<sup>2</sup>

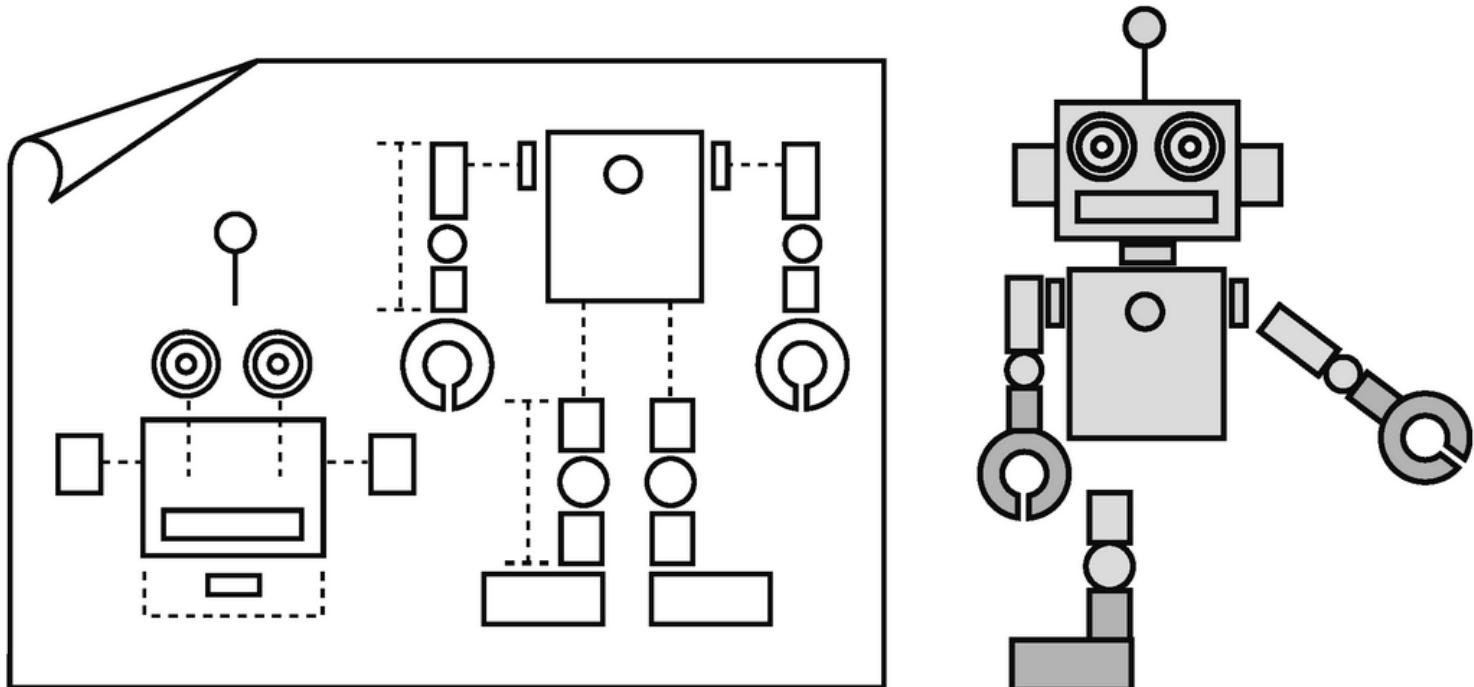
<sup>1</sup>这也是 Ruby 社区里面一个约定俗成的规矩。——译者注

<sup>2</sup>现在在 Ruby 社区最流行的 Web 框架 Rails 就是不错的参考教材之一。——译者注

## 第8章 类和模块

到目前为止，我们已经向大家介绍了基本数据类型（数值、字符串、数组、散列）、方法（操作数据的工具）、控制结构（描述程序如何运行）等程序运行所必需的要素。这些也都是大部分编程语言共通的元素，从某种意义上可以说是程序运行的基础。

作为面向对象的脚本语言的 Ruby，还为我们提供了面向对象程序设计的支持。接下来，我们首先会了解面向对象程序设计的共通概念——类，以及 Ruby 独有的功能——模块，然后再讨论面向对象程序设计的基本设计方法。



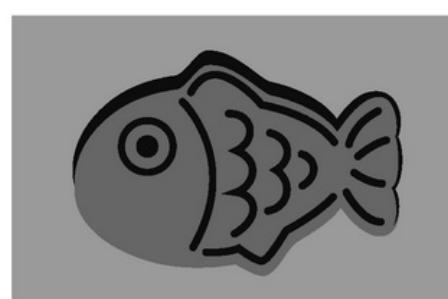
### 8.1 类是什么

类（class）是面向对象中一个重要的术语。关于类，我们在第 4 章中已经做过简单的说明，现在就进一步讨论一下在面向对象语言中如何使用类。

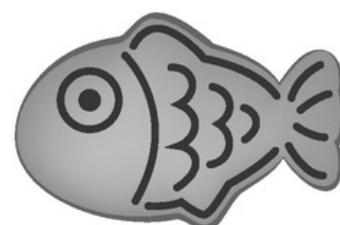
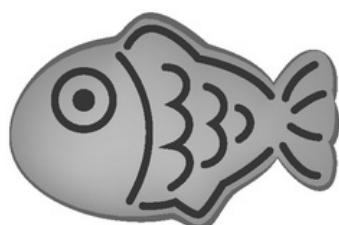
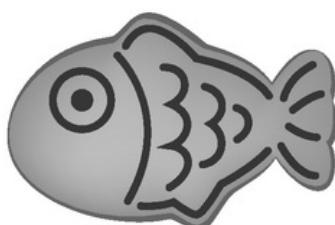
#### 8.1.1 类和实例

类表示对象的种类。Ruby 中的对象都一定属于某个类。例如，我们常说的“数组对象”“数组”，实际上都是 `Array` 类的对象（实例）。还有字符串对象，实际上是 `String` 类的对象（实例）。

相同类的对象所使用的方法也相同。类就像是对象的雏形或设计图（图 8.1），决定了对象的行为。



模型（类）



鲷鱼烧<sup>1</sup>（实例）

图 8.1 类和实例的关系

<sup>1</sup>一种传统的日本食物。——译者注

我们在生成新的对象时，一般会用到各个类的 `new` 方法。例如，使用 `Array.new` 方法可以生成新的数组对象。

```
ary = Array.new  
p ary #=> []
```

备注 像数组、字符串这样的类，也可以使用字面量（像 `[1, 2, 3]`、`"abc"` 这样的写法）来生成对象。

当想知道某个对象属于哪个类时，我们可以使用 `class` 方法。

```
ary = []  
str = "Hello world."  
p ary.class #=> Array  
p str.class #=> String
```

当判断某个对象是否属于某个类时，我们可以使用 `instance_of?` 方法。

```
ary = []  
str = "Hello world."  
p ary.instance_of?(Array) #=> true  
p str.instance_of?(String) #=> true  
p ary.instance_of?(String) #=> false  
p str.instance_of?(Array) #=> false
```

### 8.1.2 继承

我们把通过扩展已定义的类来创建新类称为继承。

假设我们需要编写一个在屏幕上显示时间的小程序。根据用户的喜好，这个小程序能以模拟时钟或者电子时钟的方式显示。

模拟时钟与电子时钟，两者只是在时间的表现形式上不一样，获取当前时间的方法以及闹钟等基本功能都是相同的。因此，我们可以首先定义一个拥有基本功能的时钟类，然后再通过继承来分别创建模拟时钟类和电子时钟类（图 8.2）。

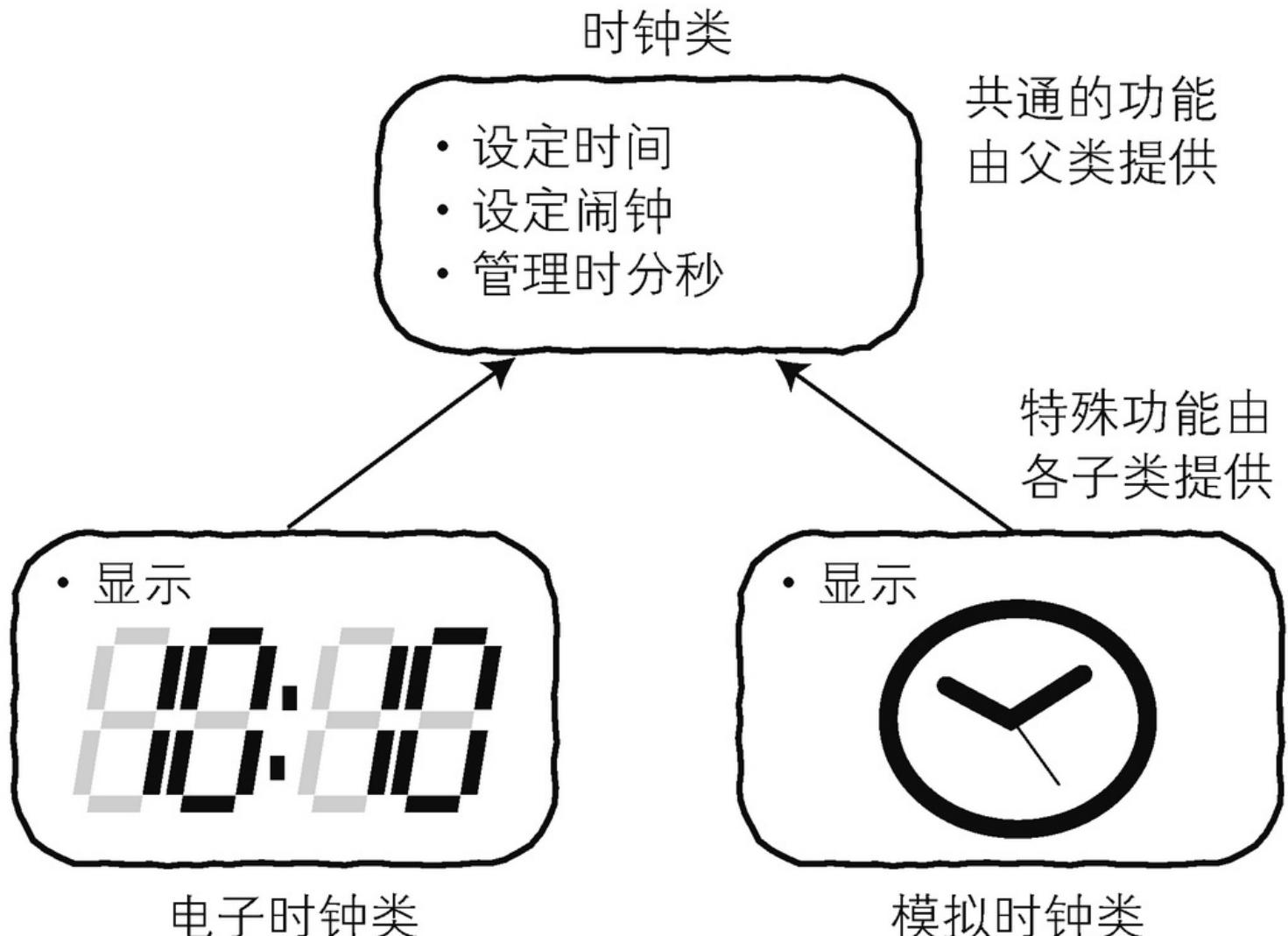


图 8.2 继承时钟类的模拟时钟类与电子时钟类

继承后创建的新类称为子类（subclass），被继承的类被称为父类（superclass）。通过继承我们可以实现以下事情：

2也称超类。——译者注

- 在不影响原有功能的前提下追加新功能。
- 重定义原有功能，使名称相同的方法产生不同的效果。
- 在已有功能的基础上追加处理，扩展已有功能。

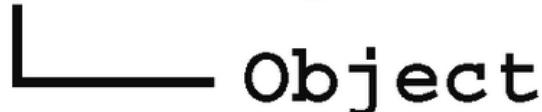
此外，我们还可以利用继承来轻松地创建多个具有相似功能的类。

`BasicObject` 类是 Ruby 中所有类的父类，它定义了作为 Ruby 对象的最基本功能。

**备注** `BasicObject` 类是最基础的类，甚至连一般对象需要的功能都没有定义。因此普通对象所需要的类一般都被定义为 `Object` 类。字符串、数组等都是 `Object` 类的子类。关于 `BasicObject` 和 `Object`，我们会在 8.3.2 节中再详细说明。

图 8.3 是本书中涉及的类继承关系图。另外，`Exception` 类下还有众多子类，这里不再详细罗列。

# BasicObject



`Array`

`String`

`Hash`

`Regexp`

`IO`

  └ `File`

`Dir`

`Numeric`

  └ `Integer`

    └ `Fixnum`

    └ `Bignum`

  └ `Float`

  └ `Complex`

  └ `Rational`

`Exception`

`Time`

图 8.3 类的继承关系

子类与父类的关系称为“is-a 关系”<sup>3</sup>。例如，`String` 类与它的父类 `Object` 就是 is-a 关系。

3这里的“a”指的是英语中的不定冠词“a”。——译者注

之前我们提到过查找对象所属的类时使用 `instance_of?` 方法，而根据类的继承关系反向追查对象是否属于某个类时，则可以使用 `is_a?` 方法。

```
str = "This is a String."
p str.is_a?(String)      #=> true
p str.is_a?(Object)      #=> true
```

顺便提一下，由于 `instance_of?` 方法与 `is_a?` 方法都已经在 `Object` 类中定义过了，因此普通的对象都可以使用这两个方法。

在本章的最后，我们会介绍类与模块的创建方法。在自己创建类之前，如果想对 Ruby 预先提供的类的使用方法一睹为快，请跳过第 2 部分余下的内容，直接进入到第 3 部分。

## 8.2 类的创建

事不宜迟，让我们来看看如何创建类。定义类时有很多约束，我们先从最基础的开始。

代码清单 8.1 就是一个创建类的例子：

代码清单 8.1 `hello_class.rb`

```
class HelloWorld          # class 关键字
  def initialize(myname = "Ruby")  # initialize 方法
    @name = myname # 初始化实例变量
  end

  def hello            # 实例方法
    puts "Hello, world. I am      #{@name}."
  end
end

bob = HelloWorld.new("Bob")
alice = HelloWorld.new("Alice")
ruby = HelloWorld.new

bob.hello
```

### 8.2.1 class 关键字

`class` 关键字在定义类时使用。以下是 `class` 关键字的一般用法：

```
class 类名
  类的定义
end
```

类名的首字母必须大写。

### 8.2.2 initialize 方法

在 `class` 关键字中定义的方法为该类的实例方法。代码清单 8.1 的 `hello` 方法就是实例方法。

其中，名为 `initialize` 的方法比较特别。使用 `new` 方法生成新的对象时，`initialize` 方法会被调用，同时 `new` 方法的参数也会被原封不动地传给 `initialize` 方法。因此初始化对象时需要的处理一般都写在这个方法中。

```
def initialize(myname = "Ruby")  # initialize 方法
  @name = myname                 # 初始化实例变量
end
```

在这个例子中，`initialize` 方法接受了参数 `myname`。因此，

```
bob = HelloWorld.new("Bob")
```

像这样，就可以把 `"Bob"` 传给 `initialize` 方法生成对象。由于 `initialize` 方法的参数指定了默认值 `"Ruby"`，因此，像下面这样没有指定参数时，

```
ruby = HelloWorld.new
```

会自动把 `"Ruby"` 传给 `initialize` 方法。

### 8.2.3 实例变量与实例方法

我们再回头看看代码清单 8.1 的 `initialize` 方法。

```
def initialize(myname = "Ruby") # initialize 方法
  @name = myname
  # 初始化实例变量
end
```

通过 `@name = myname` 这行程序，作为参数传进来的对象会被赋值给变量 `@name`。我们把以 `@` 开头的变量称为实例变量。在不同的方法中，程序会把局部变量看作是不同的变量来对待。而只要在同一个实例中，程序就可以超越方法定义，任意引用、修改实例变量的值。另外，引用未初始化的实例变量时的返回值为 `nil`。

不同实例的实例变量值可以不同。只要实例存在，实例变量的值就不会消失，并且可以被任意使用。而局部变量则是在调用方法时被创建，而且只能在该方法内使用。

我们来看看下面的例子：

```
alice = HelloWorld.new("Alice")
bob = HelloWorld.new("Bob")
ruby = helloWorld.new
```

`alice`、`bob`、`ruby` 各自拥有不同的 `@name`（图 8.4）。

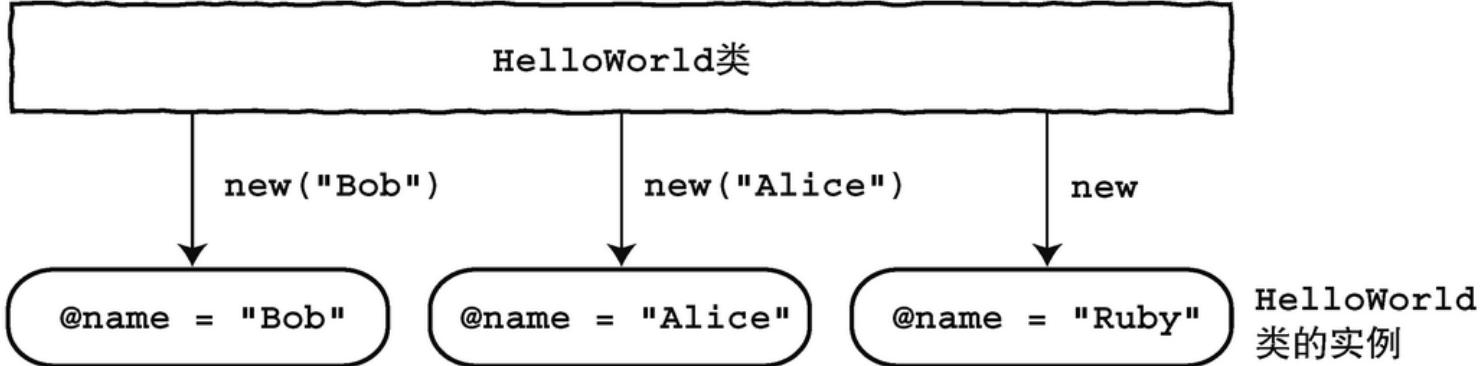


图 8.4 `HelloWorld` 类与实例

可以在实例方法中引用实例变量，下面是 `HelloWorld` 类定义的 `hello` 方法引用 `@name` 的例子：

```
class HelloWorld
  |
  def hello                      # 实例方法
    puts "Hello, world. I am #{@name}."
  end
end
```

通过以下方式调用 `HelloWorld` 类定义的 `hello` 方法：

```
bob.hello
```

输出结果如下所示：

```
Hello, world. I am Bob.
```

#### 8.2.4 存取器

在 Ruby 中，从对象外部不能直接访问实例变量或对实例变量赋值，需要通过方法来访问对象的内部。

为了访问代码清单 8.1 中 `HelloWorld` 类的 `@name` 实例变量，我们需要定义以下方法：

##### 代码清单 8.2 `hello_class.rb` (部分)

```
class HelloWorld
  |
  def name          # 获取@name
    @name
  end

  def name=(value) # 修改@name
    @name = value
  end
  |
end
```

第一个方法 `name` 只是简单地返回 `@name` 的值，我们可以像访问属性一样使用该方法。

```
p bob.name      #=> "Bob"
```

第二个方法的方法名为 `name=`，使用方法如下：

```
bob.name = "Robert"
```

乍一看，该语法很像是在给对象的属性赋值，但实际上却是在调用 `name("Robert")` 这个方法。利用这样的方法，我们就可以突破 Ruby 原有的限制，从外部来自由地访问对象内部的实例变量了。

当对象的实例变量有多个时，如果逐个定义存取器，就会使程序变得难懂，而且也容易写错。为此，Ruby 为我们提供了更简便的定义方法 `attr_reader`、`attr_writer`、`attr_accessor`（表 8.1）。只要指定实例变量名的符号（symbol），Ruby 就会自动帮我们定义相应的存取器。

表 8.1 存取器的定义

定义	意义
<code>attr_reader :name</code>	只读（定义 <code>name</code> 方法）
<code>attr_writer :name</code>	只写（定义 <code>name=</code> 方法）
<code>attr_accessor :name</code>	读写（定义以上两个方法）

也可以像下面这样只写一行代码，其效果与刚才的 `name` 方法以及 `name=` 方法的效果是一样的。

```
class HelloWorld
  attr_accessor :name
end
```

备注 Ruby 中一般把设定实例变量的方法称为 writer，读取实例变量的方法称为 reader，这两个方法合称为 accessor。另外，有时也把 reader 称为 getter，writer 称为 setter，合称为 accessor method<sup>4</sup>。

4一般把 accessor (method) 翻译为存取器或者访问器，本书统一翻译为存取器。——译者注

## 8.2.5 特殊变量 `self`

在实例方法中，可以用 `self` 这个特殊的变量来引用方法的接收者。接下来就让我们来看看其他的实例方法如何调用 `name` 方法。

代码清单 8.3 `hello_class.rb`（部分）

```
class HelloWorld
  attr_accessor :name
  |
  def greet
    puts "Hi, I am #{self.name}."
  end
end
|
```

`greet` 方法里的 `self.name` 引用了调用 `greet` 方法时的接收者。

调用方法时，如果省略了接收者，Ruby 就会默认把 `self` 作为该方法的接收者。因此，即使省略了 `self`，也还是可以调用 `name` 方法，如下所示：

```
def greet
  print "Hi, I am #{name}"
end
```

另外，在调用像 `name=` 方法这样的以 `=` 结束的方法时，有一点需要特别注意。即使实例方法中已经有了 `name = "Ruby"` 这样的定义，但如果仅在方法内部定义名为 `name` 的局部变量，也不能以缺省接收者的方式调用 `name=` 方法。这种情况下，我们需要用 `self.name = "Ruby"` 的形式来显式调用 `name` 方法。

```
def test_name
  name = "Ruby"          # 为局部变量赋值
  self.name = "Ruby"      # 调用name= 方法
end
```

**备注** 虽然 `self` 本身与局部变量形式相同，但由于它是引用对象本身时的保留字，因此我们即使对它进行赋值，也不会对其本身的值有任何影响。像这样，已经被系统使用且不能被我们自定义的变量名还有 `nil`、`true`、`false`、`_FILE_`、`_LINE_`、`_ENCODING_` 等。

## 8.2.6 类方法

方法的接收者就是类本身（类对象）的方法称为类方法。正如我们在 7.2.2 节中提到的那样，类方法的操作对象不是实例，而是类本身。

下面，让我们在 `class << 类名 ~ end` 这个特殊的类定义中，以定义实例方法的形式来定义类方法。

```
class << HelloWorld
  def hello(name)
    puts "#{name} said hello."
  end
end

HelloWorld.hello("John")  #=> John said hello.
```

在 `class` 上下文中使用 `self` 时，引用的对象是该类本身，因此，我们可以使用 `class << self ~ end` 这样的形式，在 `class` 上下文中定义类方法。

```
class HelloWorld
  class << self
    def hello(name)
      puts "#{name} said hello."
    end
  end
end
```

除此以外，我们还可以使用 `def 类名 . 方法名 ~ end` 这样的形式来定义类方法。

```
def HelloWorld.hello(name)
  puts "#{name} said hello."
end

HelloWorld.hello("John")  #=> John said hello.
```

同样，只要是在 `class` 上下文中，这种形式下也可以像下面的例子那样使用 `self`。

```
class HelloWorld
  def self.hello(name)
    puts "#{name} said hello."
  end
end
```

**备注** `class << 类名 ~ end` 这种写法的类定义称为单例类定义，单例类定义中定义的方法称为单例方法。

## 8.2.7 常量

在 `class` 上下文中可以定义常量。

```
class HelloWorld
  Version = "1.0"
  |
end
```

对于在类中定义的常量，我们可以像下面那样使用 `::`，通过类名来实现外部访问。

```
p HelloWorld::Version  #=> "1.0"
```

## 8.2.8 类变量

以 `@@` 开头的变量称为类变量。类变量是该类所有实例的共享变量，这一点与常量类似，不同的是我们可以多次修改类变量的值。另外，与实例变量一样，从类的外部访问类变量时也需要存取器。不过，由于 `attr_accessor` 等存取器都不能使用，因此需要直接定义。代码清单 8.4 的程序在代码清单 8.1 的 `HelloWorld` 类的基础上，添加了统计 `hello` 方法被调用次数的功能。

### 代码清单 8.4 hello\_count.rb

```
class HelloCount
  @@count = 0          # 调用hello 方法的次数

  def HelloCount.count  # 读取调用次数的类方法
    @@count
  end
```

```

def initialize(myname="Ruby")
  @name = myname
end

def hello
  @@count += 1      # 累加调用次数
  puts "Hello, world. I am #{@name}.\n"
end
end

bob = HelloCount.new("Bob")
alice = HelloCount.new("Alice")
ruby = HelloCount.new

p HelloCount.count      #=> 0
bob.hello
alice.hello
ruby.hello
p HelloCount.count      #=> 3

```

## 8.2.9 限制方法的调用

到目前为止，我们定义的方法，都能作为实例方法被任意调用，但是有时候我们可能并不希望这样。例如，只是为了汇总多个方法的共同处理而定义的方法，一般不会公开给外部使用。

Ruby 提供了 3 种方法的访问级别，我们可以按照需要来灵活调整。

- `public`.....以实例方法的形式向外部公开该方法
- `private`.....在指定接收者的情况下不能调用该方法（只能使用缺省接收者的方式调用该方法，因此无法从实例的外部访问）
- `protected`.....在同一个类中时可将该方法作为实例方法调用

在修改方法的访问级别时，我们会为这 3 个关键字指定表示方法名的符号。

首先来看看使用 `public` 和 `private` 的例子（代码清单 8.5）。

代码清单 8.5 acc\_test.rb

```

class AccTest
  def pub
    puts "pub is a public method."
  end

  public :pub    # 把pub 方法设定为public (可省略)

  def priv
    puts "priv is a private method."
  end

  private :priv # 把priv 方法设定为private
end

acc = AccTest.new
acc.pub
acc.priv

```

`AccTest` 类的两个方法中，`pub` 方法可以正常调用，但是在调用 `priv` 方法时程序会发生异常，并出现以下错误信息：

### 执行示例

```

> ruby acc_test.rb
pub is a public method.
acc_test.rb:17:in `<main>': private method `priv' called for
#<AccTest:0x007fb4089293e8> (NoMethodError)

```

希望统一定义多个方法的访问级别时，可以使用下面的语法：

```

class AccTest
  public # 不指定参数时,
        # 以下的方法都被定义为public

  def pub
    puts "pub is a public method."

```

```
end
```

```
private # 以下的方法都被定义为private
```

```
def priv
  puts "priv is a private method."
end
end
```

备注 没有指定访问级别的方法默认为 `public`, 但 `initialize` 方法是个例外, 它通常会被定义为 `private`。

定义为 `protected` 的方法, 在同一个类 (及其子类) 中可作为实例方法使用, 而在除此以外的地方则无法使用。

代码清单 8.6 定义了拥有 X、Y 坐标的 `Point` 类。在这个类中, 实例中的坐标可以被外部读取, 但不能被修改。为此, 我们可以利用 `protected` 来实现交换两个坐标值的方法 `swap`。

#### 代码清单 8.6 point.rb

```
class Point
  attr_accessor :x, :y      # 定义存取器
  protected :x=, :y=        # 把x= 与y= 设定为protected

  def initialize(x=0.0, y=0.0)
    @x, @y = x, y
  end

  def swap(other)          # 交换x、y 值的方法
    tmp_x, tmp_y = @x, @y
    @x, @y = other.x, other.y
    other.x, other.y = tmp_x, tmp_y    # 在同一个类中
                                         # 可以被调用
    return self
  end
end

p0 = Point.new
p1 = Point.new(1.0, 2.0)
p [ p0.x, p0.y ]           #=> [0.0, 0.0]
p [ p1.x, p1.y ]           #=> [1.0, 2.0]

p0.swap(p1)
p [ p0.x, p0.y ]           #=> [1.0, 2.0]
p [ p1.x, p1.y ]           #=> [0.0, 0.0]

p0.x = 10.0                 #=> 错误 (NoMethodError)
```

## 8.3 扩展类

### 8.3.1 在原有类的基础上添加方法

Ruby 允许我们在已经定义好的类中添加方法。下面, 我们来试试给 `String` 类添加一个计算字符串单词数的实例方法 `count_word` (代码清单 8.7)。

#### 代码清单 8.7 ext\_string.rb

```
class String
  def count_word
    ary = self.split(/\s+/) # 用空格分割接收者
    return ary.size         # 返回分割后的数组的元素总数
  end
end

str = "Just Another Ruby Newbie"
p str.count_word          #=> 4
```

### 8.3.2 继承

正如在 8.1.2 节中介绍的那样, 利用继承, 我们可以在不对已有的类进行修改的前提下, 通过增加新功能或重定义已有功能等手段来创建新的类。

定义继承时, 在使用 `class` 关键字指定类名的同时指定父类名。

```
class 类名 < 父类名
  类定义
end
```

在程序清单 8.8 中，创建一个继承了 `Array` 类的 `RingArray` 类。`RingArray` 类只是重定义了读取数组内容时使用的 `[]` 运算符。该程序通过 `super` 关键字调用父类中同名的方法（在本例中也就是 `Array#[]`）。

#### 代码清单 8.8 ring\_array.rb

```
class RingArray < Array # 指定父类
  def [](i)           # 重定义运算符[]
    idx = i % size    # 计算新索引值
    super(idx)         # 调用父类中同名的方法
  end
end

wday = RingArray["日", "月", "火", "水", "木", "金", "土"]
p wday[6]    #=> "土"
p wday[11]   #=> "木"
p wday[15]   #=> "月"
p wday[-1]   #=> "土"
```

对 `RingArray` 类指定了超过数组长度的索引时，结果就会从溢出部分的开头开始重新计算索引（图 8.5）。

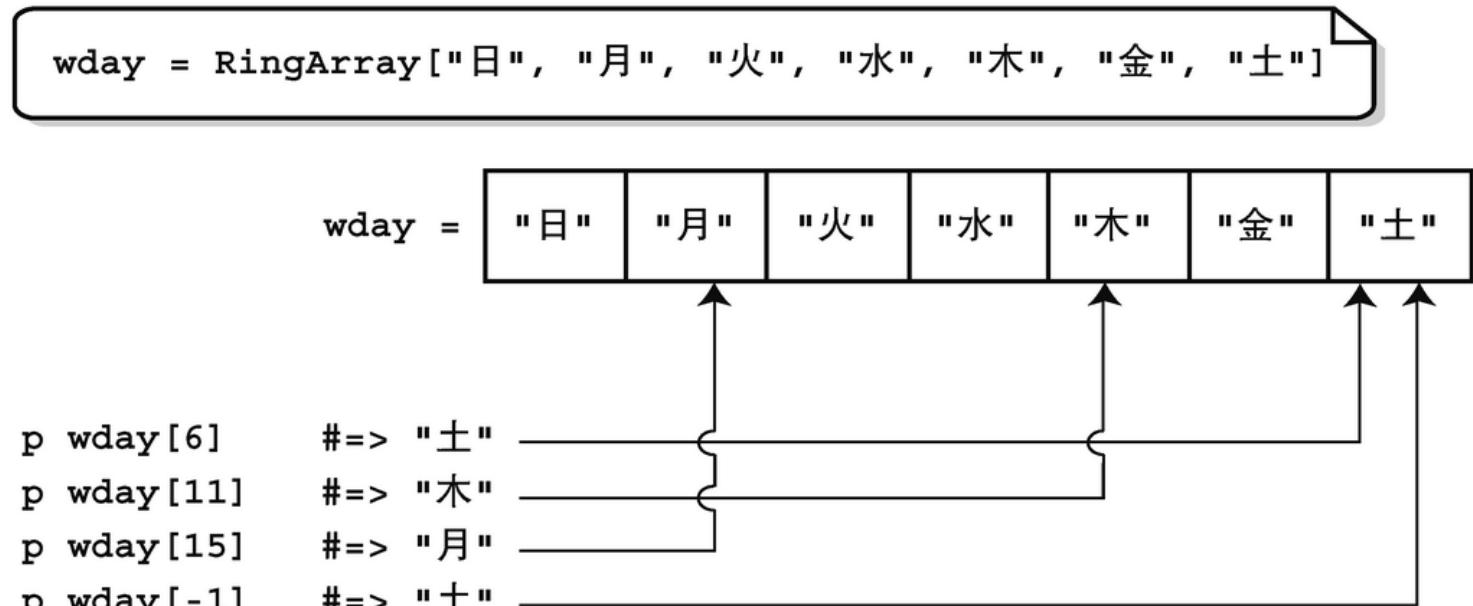


图 8.5 `RingArray` 类

利用继承，我们可以把共同的功能定义在父类，把各自独有的功能定义在子类。

定义类时没有指定父类的情况下，Ruby 会默认该类为 `Object` 类的子类。

`Object` 类提供了许多便于实际编程的方法。但在某些情况下，我们也有可能会希望使用更轻量级的类，而这时就可以使用 `BasicObject` 类。

`BasicObject` 类只提供了组成 Ruby 对象所需的最低限度的方法。类对象调用 `instance_methods` 方法后，就会以符号的形式返回该类的实例方法列表。下面我们就用这个方法来对比一下 `Object` 类和 `BasicObject` 类的实例方法。

#### 执行示例

```
> irb --simple-prompt
>> Object.instance_methods
=> [:nil?, :==~, :!~, :eql?, :hash, :<=>, :class, :singleton_class, :clone,
:dup, :taint, :tainted?, :untaint, :untrust, :untrusted?, :trust, :freeze,
:frozen?, :to_s, ..... 等众多方法名.....]
>> BasicObject.instance_methods
=> [::=, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__, :__id__]
```

虽然大部分方法我们都还没有接触到，但据此也可以看出，相对于 `Object` 类持有多种方法，`BasicObject` 类所拥有的功能都是最基本的。定义 `BasicObject` 的子类时，与 `Object` 类不同，需要明确指定 `BasicObject` 类为父类，如下所示：

```
class MySimpleClass < BasicObject
|
end
```

## 8.4 alias 与 undef

### 8.4.1 alias

有时我们会希望给已经存在的方法设置别名。这种情况下就需要使用 `alias` 方法。`alias` 方法的参数为方法名或者符号名。

```
alias 别名 原名      # 直接使用方法名
alias :别名 :原名    # 使用符号名
```

像 `Array#size` 与 `Array#length` 这样，为同一种功能设置多个名称时，我们会使用到 `alias`。

另外，除了为方法设置别名外，在重定义已经存在的方法时，为了能用别名调用原来的方法，我们也要用到 `alias`。

下面的例子中定义了类 `C1` 及其子类 `C2`。在类 `C2` 中，对 `hello` 方法设置别名 `old_hello` 后，重定义了 `hello` 方法。

```
class C1                  # 定义C1
  def hello                # 定义hello
    "Hello"
  end
end

class C2 < C1            # 定义继承了C1 的子类C2
  alias old_hello hello   # 设定别名old_hello
  def hello                # 重定义hello
    "#{old_hello}, again"
  end
end

obj = C2.new
p obj.old_hello          #=> "Hello"
p obj.hello              #=> "Hello, again"
```

## 8.4.2 undef

`undef` 用于删除已有方法的定义。与 `alias` 一样，参数可以指定方法名或者符号名。

```
undef 方法名      # 直接使用方法名
undef :方法名    # 使用符号名
```

例如，在子类中希望删除父类定义的方法时可以使用 `undef`。

专栏

单例类

在 8.2.6 节中介绍定义类方法的方法时，我们提到了单例类定义，而通过利用单例类定义，就可以给对象添加方法（单例方法）。单例类定义被用于定义对象的专属实例方法。在下面的例子中，我们分别将 `"Ruby"` 赋值给 `str1` 对象和 `str2` 对象，然后只对 `str1` 对象添加 `hello` 方法。这样一来，两个对象分别调用 `hello` 方法时，`str1` 对象可以正常调用，但 `str2` 对象调用时程序就会发生错误。

```
str1 = "Ruby"
str2 = "Ruby"

class << str1
  def hello
    "Hello, #{self}!"
  end
end

p str1.hello      #=> "Hello, Ruby!"
p str2.hello      #=> 错误 (NoMethodError)
```

Ruby 中所有的类都是 `Class` 类的实例，对 `Class` 类添加实例方法，就等于给所有的类都添加了该类方法。因此，只希望对某个实例添加方法时，就需要利用单例方法。

单例类的英语为 `singleton class` 或者 `eigenclass`。

## 8.5 模块是什么

模块是 Ruby 的特色功能之一。如果说类表现的是事物的实体（数据）及其行为（处理），那么模块表现的就只是事物的行为部分。模块与类有以下两点不同：

- 模块不能拥有实例
- 模块不能被继承

## 8.6 模块的使用方法

接下来，我们就来介绍一下模块的典型用法。

### 8.6.1 提供命名空间

所谓命名空间（namespace），就是对方法、常量、类等名称进行区分及管理的单位。由于模块提供各自独立的命名空间，因此 A 模块中的 `foo` 方法与 B 模块中的 `foo` 方法，就会被程序认为是两个不同的方法。同样，A 模块中的 `FOO` 常量与 B 模块的 `FOO` 常量，也是两个不同的常量。

无论是方法名还是类名，当然都是越简洁越好，但是像 `size`、`start` 等这种普通的名称，可能在很多地方都会使用到。因此，通过在模块内定义名称，就可以解决命名冲突的问题。

例如，在 `FileTest` 模块中存在与获取文件信息相关的方法。我们使用“模块名 · 方法名”的形式来调用在模块中定义的方法，这样的方法称为模块函数。

```
# 检查文件是否存在
p FileTest.exist?("/usr/bin/ruby")  #=> true
# 文件大小
p FileTest.size("/usr/bin/ruby")      #=> 1374684
```

如果没有定义与模块内的方法、常量等同名的名称，那么引用时就可以省略模块名。通过 `include` 可以把模块内的方法名、常量名合并到当前的命名空间。下面是与数学运算有关的 `Math` 模块的例子。

```
# 圆周率（常量）
p Math::PI          #=> 3.141592653589793
# 2 的平方根
p Math.sqrt(2)      #=> 1.4142135623730951

include Math        # 包含Math 模块
p PI               #=> 3.141592653589793
p sqrt(2)          #=> 1.4142135623730951
```

像这样，通过把一系列相关的功能汇总在一个模块中，就可以集中管理相关的命名。

### 8.6.2 利用 Mix-in 扩展功能

Mix-in 就是将模块混合到类中。在定义类时使用 `include`，模块里的方法、常量就都能被类使用。

像代码清单 8.9 那样，我们可以把 `MyClass1` 和 `MyClass2` 中两者共通的功能定义在 `MyModule` 中。虽然有点类似于类的继承，但 Mix-in 可以更加灵活地解决下面的问题。

- 虽然两个类拥有相似的功能，但是不希望把它们作为相同的种类（`class`）来考虑的时候
- Ruby 不支持父类的多重继承，因此无法对已经继承的类添加共通的功能的时候

关于继承和 Mix-in 之间的关系，我们会在 8.8 节中进行说明。

代码清单 8.9 `mixin_sample.rb`

```
module MyModule
  # 共通的方法等
end

class MyClass1
  include MyModule
  # MyClass1 中独有的方法
end

class MyClass2
  include MyModule
  # MyClass2 中独有的方法
end
```

## 8.7 创建模块

我们使用 `module` 关键字来创建模块。

语法与创建类时几乎相同。模块名的首字母必须大写。

```
module 模块名
  模块定义
end
```

下面，让我们参考着代码清单 8.1 的 `HelloWorld` 类，来看看如何创建模块（代码清单 8.10）。

代码清单 8.10 `hello_module.rb`

```

module HelloModule          # module 关键字
Version = "1.0"            # 定义常量

def hello(name)           # 定义方法
  puts "Hello, #{name}."
end

module_function :hello    # 指定hello 方法为模块函数
end

p HelloModule::Version    #=> "1.0"
HelloModule.hello("Alice") #=> Hello, Alice.

include HelloModule        # 包含模块
p Version                 #=> "1.0"
hello("Alice")             #=> Hello, Alice.

```

### 8.7.1 常量

和类一样，在模块中定义的常量可以通过模块名访问。

```
p HelloModule::Version    #=> "1.0"
```

### 8.7.2 方法的定义

和类一样，我们也可以在 `module` 上下文中定义方法。

然而，如果仅仅定义了方法，虽然在模块内部与包含此模块的上下文中都可以直接调用，但却不能以“模块名.方法名”的形式调用。如果希望把方法作为模块函数公开给外部使用，就需要用到 `module_function` 方法。`module_function` 的参数是表示方法名的符号。

```

def hello(name)
  puts "Hello, #{name}."
end

module_function :hello

```

以“模块名.方法名”的形式调用时，如果在方法中调用 `self`（接收者），就会获得该模块的对象。

```

module FooMoudle
  def foo
    p self
  end
  module_function :foo
end

FooMoudle.foo    #=> FooMoudle

```

此外，如果类 Mix-in 了模块，就相当于为该类添加了实例方法。在这种情况下，`self` 代表的就是被 Mix-in 的类的对象。

即使是相同的方法，在不同的上下文调用时，其含义也会不一样，因此对于 Mix-in 的模块，我们要注意根据实际情况判断是否使用模块函数功能。一般不建议在定义为模块函数的方法中使用 `self`。

## 8.8 Mix-in

现在，我们来详细讨论一下本章开头就提到的 Mix-in。这里使用 `include` 使类包含模块（代码清单 8.11）。

代码清单 8.11 `mixin_test.rb`

```

module M
  def meth
    "meth"
  end
end

class C
  include M  # 包含M 模块
end

c = C.new
p c.meth    #=> meth

```

类 `c` 包含模块 `M` 后，模块 `M` 中定义的方法就可以作为类 `C` 的实例方法供程序调用。

另外，如果想知道类是否包含某个模块，可以使用 `include?` 方法。

```
C.include?(M) #=> true
```

类 `C` 的实例在调用方法时，Ruby 会按类 `C`、模块 `M`、类 `C` 的父类 `Object` 这个顺序查找该方法，并执行第一个找到的方法。被包含的模块的作用就类似于虚拟的父类。

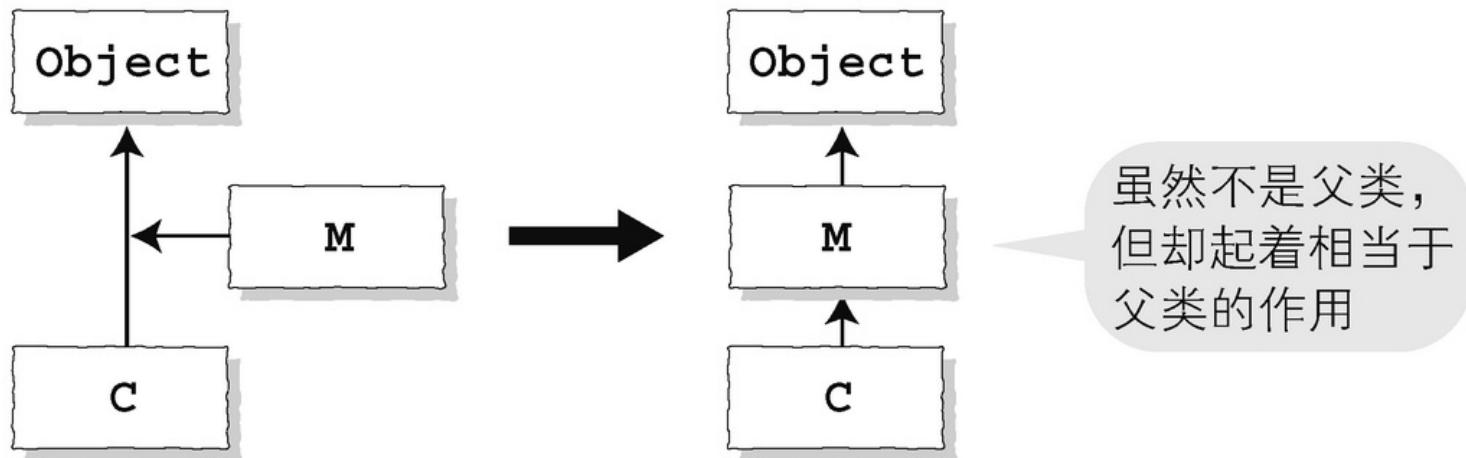


图 8.6 类的继承关系

我们用 `ancestors` 方法和 `superclass` 方法调查类的继承关系。在代码清单 8.11 中追加以下代码并执行，我们就可以通过 `ancestors` 取得继承关系的列表。进而也就可以看出，被包含的模块 `M` 也被认为是类 `C` 的一个“祖先”。而 `superclass` 方法则直接返回类 `C` 的父类。

```
p C.ancestors      #=> [C, M, Object, Kernel, BasicObject]
p C.superclass     #=> Object
```

**备注** `ancestors` 方法的返回值中的 `Kernel` 是 Ruby 内部的一个核心模块，Ruby 程序运行时所需的共通函数都封装在此模块中。例如 `p` 方法、`raise` 方法等都是由 `Kernel` 模块提供的模块函数。

虽然 Ruby 采用的是不允许多个父类的单一继承模型，但是通过利用 Mix-in，我们就既可以保持单一继承的关系，又可以同时让多个类共享其他功能。

在 Ruby 标准类库中，`Enumerable` 模块就是利用 Mix-in 扩展功能的一个典型例子。使用 `each` 方法的类中包含 `Enumerable` 模块后，就可以使用 `each_with_index` 方法、`collect` 方法等对元素进行排序处理的方法。`Array`、`Hash`、`IO` 等类都包含了 `Enumerable` 模块（图 8.7）。这些类虽然没有继承这样的血缘关系，但是从“可以使用 `each` 方法遍历元素”这一点来看，可以说它们都拥有了某种相似甚至相同的属性。

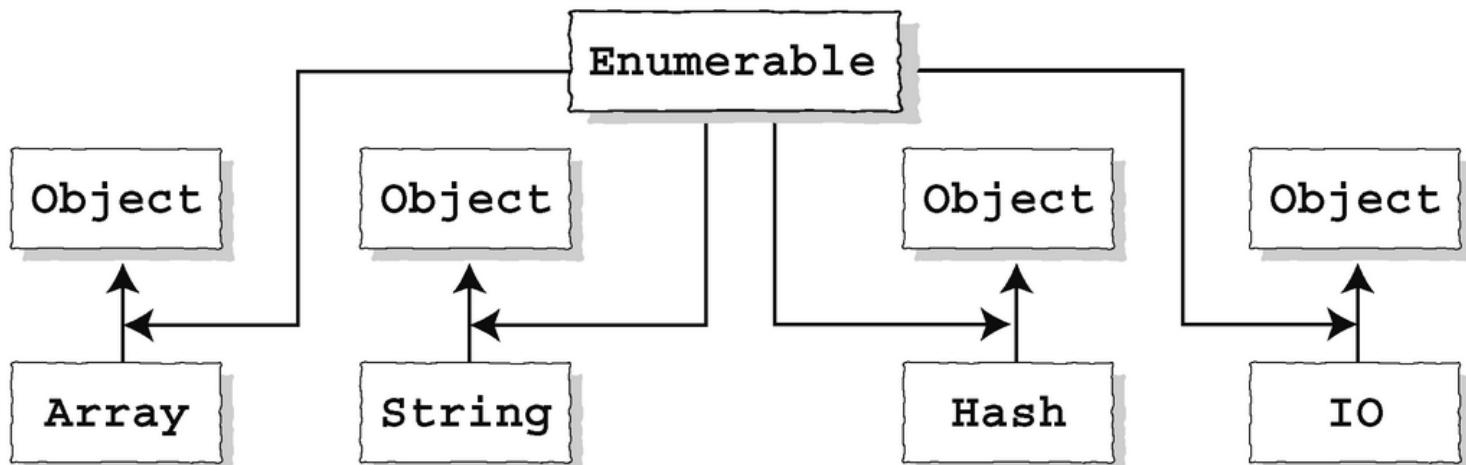


图 8.7 Enumerable 模块和各类的关系

单一继承的优点就是简单，不会因为过多的继承而导致类之间的关系变得复杂。但是另一方面，有时我们又会希望更加积极地重用已有的类，或者把多个类的特性合并为更高级的类，在那样的情况下，灵活使用单一继承和 Mix-in，既能使类结构简单易懂，又能灵活地应对各种需求。

### 8.8.1 查找方法的规则

首先，我们来了解一下使用 Mix-in 时方法的查找顺序。

① 同继承关系一样，原类中已经定义了同名的方法时，优先使用该方法。

```
module M
  def meth
    "M#meth"
  end
```

```
end

class C
  include M      # 包含M
  def meth
    "C#meth"
  end
end

c = C.new
p c.meth      #=> C#meth
```

② 在同一个类中包含多个模块时，优先使用最后一个包含的模块。

```
module M1
  |
end

module M2
  |
end

class C
  include M1      #=> 包含M1
  include M2      #=> 包含M2
end

p C.ancestors  #=> [C, M2, M1, Object, Kernel]
```

③ 嵌套 `include` 时，查找顺序也是线性的，此时的关系如图 8.8 所示。

```
module M1
  |
end

module M2
  |
end

module M3
  include M2      #=> 包含M2
end

class C
  include M1      #=> 包含M1
  include M3      #=> 包含M3
end

p C.ancestors  #=> [C, M3, M2, M1, Object, Kernel]
```

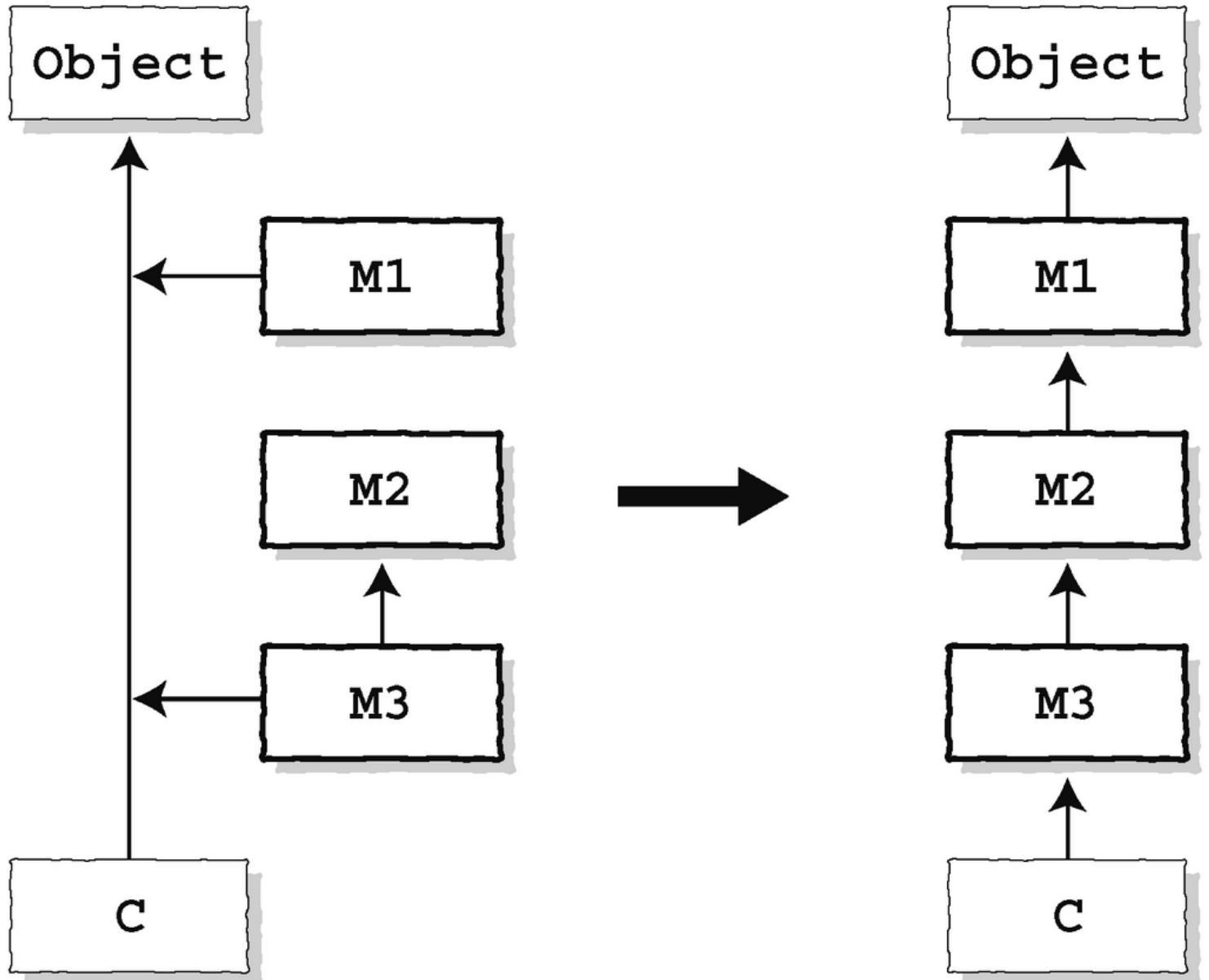


图 8.8 嵌套 include 时的关系

④ 相同的模块被包含两次以上时，第 2 次以后的会被省略。

```
module M1
|
end

module M2
|
end

class C
  include M1      #=> 包含M1
  include M2      #=> 包含M2
  include M1      #=> 包含M1
end

p C.ancestors    #=> [C, M2, M1, Object, Kernel, BasicObject]
```

## 8.8.2 extend 方法

在之前的专栏中，我们已经介绍了如何逐个定义单例方法，而利用 `Object#extend` 方法，我们还可以实现批量定义单例方法。`extend` 方法可以使单例类包含模块，并把模块的功能扩展到对象中。

```
module Edition
def edition(n)
  "#{self} 第#{n} 版"
end
end

str = "Ruby 基础教程"
```

```
str.extend(Edition)      #=> 将模块Mix-in进对象
p str.edition(4)         #=> "Ruby 基础教程第4 版"
```

`include`可以帮助我们突破继承的限制，通过模块扩展类的功能；而`extend`则可以帮助我们跨过类，直接通过模块扩展对象的功能。

### 8.8.3 类与 Mix-in

在 Ruby 中，所有类本身都是`Class`类的对象。我们之前也介绍过接收者为类本身的方法就是类方法。也就是说，类方法就是类对象的实例方法。我们可以把类方法理解为：

- `Class`类的实例方法
- 类对象的单例方法

继承类后，这些方法就会作为类方法被子类继承。对子类定义单例方法，实际上也就是定义新的类方法。

除了之前介绍的定义类方法的语法外，使用`extend`方法也同样能为类对象追加类方法。下面是使用`extend`方法追加类方法，并使用`include`方法追加实例方法的一个例子。

```
module ClassMethods      # 定义类方法的模块
  def cmethod
    "class method"
  end
end

module InstanceMethods # 定义实例方法的模块
  def imethod
    "instance method"
  end
end

class MyClass
  # 使用extend 方法定义类方法
  extend ClassMethods
  # 使用include 定义实例方法
  include InstanceMethods
end

p MyClass.cmethod      #=> "class method"
p MyClass.new.imethod #=> "instance method"
```

**备注** 在 Ruby 中，所有方法的执行，都需要通过作为接收者的某个对象的调用。换句话说，Ruby 的方法（包括单例方法）都一定属于某个类，并且作为接收者对象的实例方法被程序调用。从这个角度来说，人们只是为了便于识别接收者的类型，才分别使用了“实例方法”和“类方法”这样的说法。

## 8.9 面向对象程序设计

“面向对象”这个概念，被广泛地应用在问题分析、系统设计或程序设计等系统和程序开发领域中。虽然这个概念目前被用在了各种各样的领域中，但首先使用这个概念的是与程序设计相关的领域。

由于本书是程序设计语言的入门书，因此这里并不会向其他领域过多延伸，而只会介绍与程序设计语言（当然就是 Ruby 了）相关的对象和面向对象方面的基础知识。

下面，我们暂且不讨论具体如何编写程序，而是先来了解一下编写程序时的一些思考方法。因为话题比较抽象，所以可能会有点难懂，不过请大家放轻松，耐心地读下去你就会抓住窍门的。

### 8.9.1 对象是什么

包括 Ruby 在内，世界上有多种面向对象的程序设计语言。不同的语言，不仅语法不一样，功能也千差万别，但它们几乎都有一个共通点，就是将程序处理的主体作为“对象”来考虑。

一般情况下，程序语言的处理主体是数据。之前提到的数值、字符串、数组等都是简单的数据。

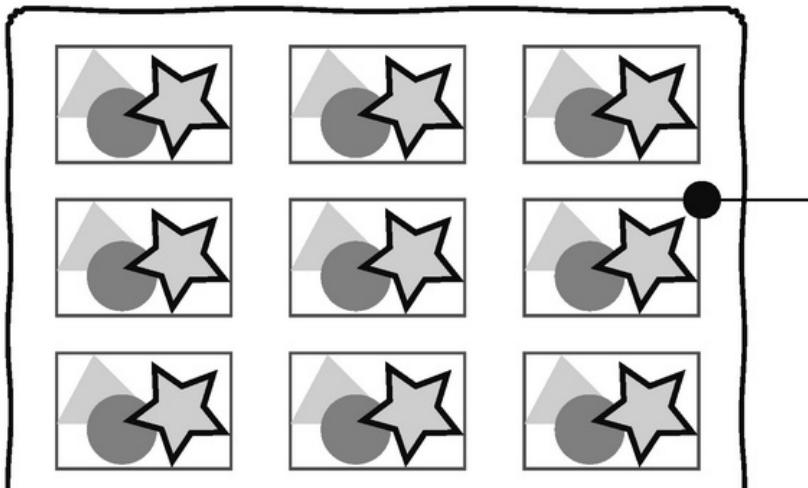
而面向对象的语言中的“对象”就是指数据（或者说数据的集合）及操作该数据的方法的组合。之前我们提到过 Ruby 里的数值`3.14`是`Float`类的实例。这个`3.14`不仅是表示`3.14`这个数值的数据，还包括与数值相关的操作方法。

```
f = 3.14
p f.round    #=> 3 (四舍五入)
p f.ceil     #=> 4 (进位)
p f.to_i     #=> 3 (整数变换)
```

像这样，把数据以及处理数据的操作方法作为对象合并在一起贯穿整体，在面向对象程序设计中是很常见的。例如，将浮点数做四舍五入处理的`round`方法是可以被作为`Float`类的一部分来提供的，这样一来，数据以及处理数据的方法的组合也不会出现错误。

如果只是简单的数值处理，并不会有太大的问题，但大部分程序都需要更复杂的数据构成。例如，在一个处理图片的程序中，图片的长宽、颜色、包括图片本身的内容都需要转换为二进制数据。如果能把一个图片作为一个零部件来处理，那么像图册这样复杂的应用程序也就变得容易编写了（图 8.9）。

图册



图片

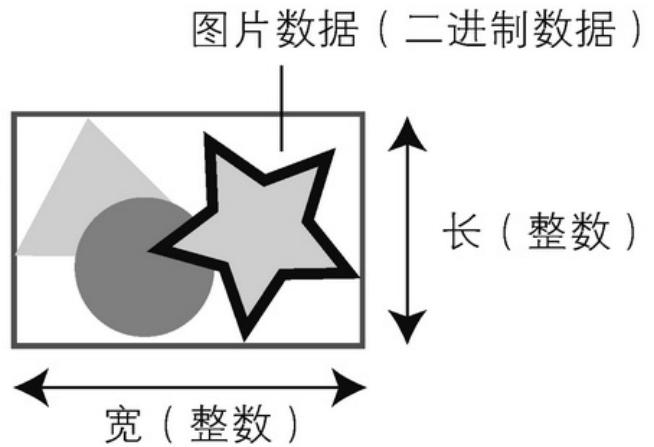


图 8.9 结构化后的数据

在开发大型程序的时候，若不将大量的数据整合到一起并根据一定的规则进行整理，程序处理本身的统一性会荡然无存。面向对象程序设计会把这种归类统一的数据作为各种各样的对象来看待。在对象中，数据以及处理数据的方法也是成套存在的，而且还负有处理数据的责任。

另外，就像网络上的服务器管理的文件一样，远程数据也可以作为程序的处理对象来考虑。在网络程序设计里，Web、邮件等不同的应用程序，都需要遵守各自不同的规范（也称为协议）。用程序来实现协议的情况下，一般会把管理消息格式、规范等的程序抽象成库（library）。Ruby 的库里就有现成的 `Net::HTTP`、`Net::POP` 等类，可以非常轻松地编写网络程序。

### 8.9.2 面向对象的特征

上文中我们简单地介绍了面向对象程序设计的思考方法，下面就让我们来整理一下面向对象的特征。

- 封装

所谓封装（encapsulation），就是指使对象管理的数据不能直接从外部进行操作，数据的更新、查询等操作都必须通过调用对象的方法来完成。通过封装，可以防止因把非法数据设置给对象而使程序产生异常的情况发生。

为此，就需要编写不会让对象内部产生异常的方法。最理想的做法是，在定义方法时就考虑如何避免错误的发生，而不是在使用方法编写程序时才开始注意。

Ruby 对象在默认情况下是强制被封装的，因此无法从外部直接访问 Ruby 对象的实例变量。虽然有像 `attr_accessor`（8.2.4 节）这样简单定义访问级别的方法，但也不要过度使用，建议只在需要公开时才使用。

封装的另外一个好处就是，可以隐藏对象内部数据处理的具体细节，把内部逻辑抽象地表现出来。例如，通过使用 `Time` 类，就可以进行从系统获取当前时间、从时间里提取年月日等操作。

```
t = Time.now      # 从系统获取当前时间
p t.year          #=> 2013 (从时间里提取年)
```

从系统获取当前时间时是如何处理的、`Time` 对象内部是以什么形式管理时间的、以及从时间中提取年时要进行何种运算等等，以上这些事情都由 `Time` 类的方法来实现。就算对象内部的数据结构改变了，只要公开给外部的方法名、功能等没有改变，类的使用者就完全不需要理会内部逻辑作出了怎样的修改，照常使用即可。相反，类的编写者只要提供的方法恰当，就可以直接修改类的内部逻辑，而不需要在意类的使用者。可见，封装对类的编写者和使用者来说都非常有好处。

- 多态

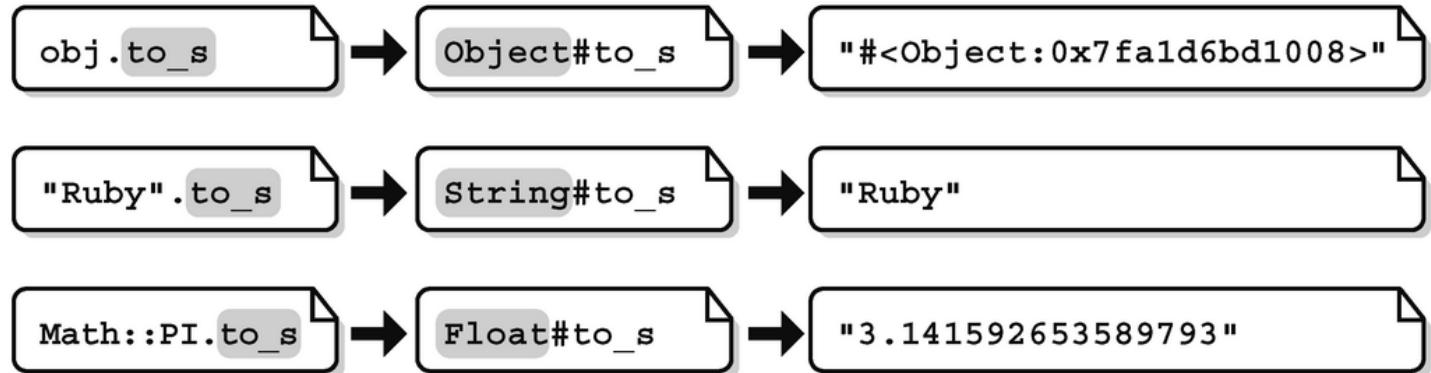
对象是数据及其处理的组合。对象知道数据是怎样被处理的。换句话说，各个对象都有自己独有的消息解释权。一个方法名属于多个对象（不同对象的处理结果也不一样）这种现象，用面向对象的术语来说，就是多态（polymorphism）。

例如，我们可以观察一下对 `Object` 类、`String` 类和 `Float` 类的各对象调用 `to_s` 方法的运行结果，可以看出，不同的类得到的结果是不一样的。

```
obj = Object.new      # 对象 (Object)
str = "Ruby"          # 字符串 (String)
num = Math::PI         # 数值 (Float)

p obj.to_s            #=> "#<Object:0x07fa1d6bd1008>"
p str.to_s            #=> "Ruby"
p num.to_s            #=> "3.141592653589793"
```

三者的 `to_s` 方法名一样，含义也都是“以可以显示的形式把数据转换为字符串”，但实际的字符串转换方式却因对象而异（图 8.10）。`String` 类和 `Float` 类都是继承自 `Object` 类，也都重新定义了从 `Object` 类继承的 `to_s` 方法，并提供了更适合自己语义的 `to_s` 方法。



※ 虽然方法名一样，但调用的却是各个类专用的版本。

※ 虽然方法名一样，但调用的却是各个类专用的版本。

图 8.10 多态

### 8.9.3 鸭子类型

下面，我们来看一种结合对象特征，灵活运用多态的思考方法——鸭子类型（duck typing）。鸭子类型的说法来自于“能像鸭子那样走路，能像鸭子一样啼叫的，那一定就是鸭子”这句话。这句话的意思是，对象的特征并不是由其种类（类及其继承关系）决定的，而是由对象本身具有什么样的行为（拥有什么方法）决定的。例如，假设我们希望从字符串数组中取出元素，并将字母转换成小写后返回结果（代码清单 8.12）。

代码清单 8.12 `fetch_and.downcase.rb`

```
def fetch_and.downcase(ary, index)
  if str = ary[index]
    return str.downcase
  end
end

ary = ["Boo", "Foo", "Woo"]
p fetch_and.downcase(ary, 1) #=> "foo"
```

实际上，除了数组外，我们也可以像下面那样，把散列传给该方法处理。

```
hash = {0 => "Boo", 1 => "Foo", 2 => "Woo"}
p fetch_and.downcase(hash, 1) #=> "foo"
```

`fetch_and.downcase` 方法对传进来的参数只有两个要求：

- 能以 `ary[index]` 形式获取元素
- 获取的元素可以执行 `downcase` 方法

只要参数符合这两个要求，`fetch_and.downcase` 方法并不关心传进来的到底是数组还是散列。

Ruby 中的变量没有限制类型，所以不会出现不是某个特定的类的对象，就不能给变量赋值的情况。因此，在程序开始运行之前，我们都无法知道变量指定的对象的方法调用是否正确。

这样的做法有个缺点，就是增加了程序运行前检查错误的难度。但是，从另外一个角度来看，则可以非常简单地使没有明确继承关系的对象之间的处理变得通用。只要能执行相同的操作，我们并不介意执行者是否一样；相反，虽然实际上是不同的执行者，但通过定义相同名称的方法，也可以实现处理通用化。这就是鸭子类型思考问题的方法。

利用鸭子类型实现处理通用化，并不要求对象之间有明确的继承关系，因此，要想灵活运用，可能还需要花不少功夫。例如刚才介绍的 `obj[index]` 的形式，就被众多的类作为访问内部元素的手段而使用。刚开始时，我们可以先有意识地留意这种简单易懂的方法，然后再由浅入深，慢慢地就可以抓住窍门了。

### 8.9.4 面向对象的例子

接下来让我们通过一个实际的例子，来看看对象是如何被构造的。代码清单 8.13 是一个利用 `Net::HTTP` 类取得 Ruby 官网首页的 HTML，并将其输出到控制台的脚本。

代码清单 8.13 `http_get.rb`

```
1: require "net/http"
2: require "uri"
```

```
3: url = URI.parse("http://www.ruby-lang.org/ja/")
4: http = Net::HTTP.start(url.host, url.port)
5: doc = http.get(url.path)
6: puts doc
```

程序的第 1、2 行中引用了 `net/http` 库以及 `uri` 库。这样，我们就可以使用 `Net::HTTP` 类和 `URI` 模块了。第 3 行使用了 `URI` 模块的 `parse` 方法来解析 URL 的字符串，返回的结果是字符串解析后的 `URI::HTTP` 类的对象。根据 URL 的编写规则，URL 会被分解成多个属性。

```
require "uri"
url = URI.parse("http://www.ruby-lang.org/ja/")
p url.scheme      #=> "http"           (体系: URL 的种类)
p url.host        #=> "www.ruby-lang.org" (主机名)
p url.port        #=> "80"             (端口号)
p url.path        #=> "/ja/"            (路径)
p url.to_s         #=> "http://www.ruby-lang.org/ja/"
```

体系（scheme）是指使用哪种通信协议。连接网络上的服务器时，需要知道服务器的主机名以及端口号。路径用于定位服务器上管理的文件。`URI::HTTP` 类的作用就是，把 URL 字符串解析后分解出来的信息，以对象的形式再次整合在一起。

需要注意的是，模块名是 `URI` 而不是 `URL`。`URL` 指的是 `URI` 标识符中某种特定种类的东西。关于两者的关系，这里不再详细介绍，现阶段我们只需要知道 `URL` 是 `URI` 的一种就可以了。

让我们再次回到代码清单 8.13，在程序第 4 行，把主机名和端口号传给 `Net::HTTP` 类的 `start` 方法，并创建 `Net::HTTP` 对象。在程序第 5 行，对 `Net::HTTP` 的 `get` 方法指定路径，获取文档内容。最后，在程序第 6 行，把得到的文档内容输出到控制台。由于得到的文档内容是 `String` 对象，因此后续处理与 `Net::HTTP` 类没有关系。

调用 `Net::HTTP#get` 方法的时候，对象内部会做以下处理：

- ① 使用主机名和端口号，与服务器建立通信（叫做 `socket`，套接字）
- ② 使用路径，创建代表请求信息的 `Net::HTTPRequest` 对象
- ③ 对套接字写入请求信息
- ④ 从套接字中读取数据，并将其保存到代表响应信息的 `Net::HTTPResponse` 对象中
- ⑤ 利用 `Net::HTTPResponse` 本身提供的功能，解析响应信息，提取文档部分并返回。

流程图如下所示。

## http\_get.rb

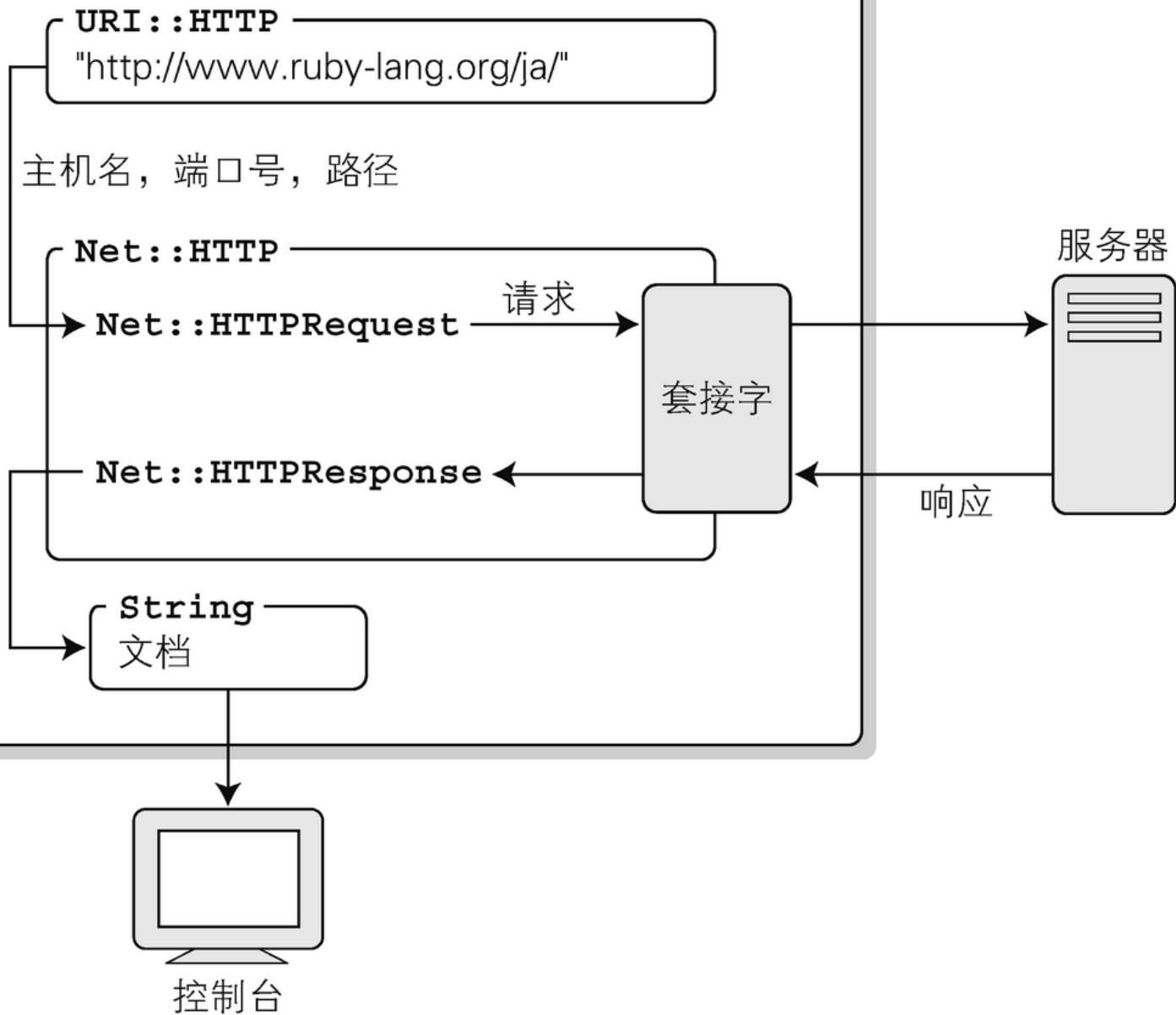


图 8.11 `http_get.rb` 的流程图

在这个例子中，URL 解析由 `URI::HTTP` 负责，网络连接由套接字负责，与信息交换相关的操作由 `Net::HTTPRequest` 和 `Net::HTTPResponse` 负责，通信中必要的套接字、请求、响应等相关操作由 `Net::HTTP` 负责。像这样，不同的对象各司其职，决定应该如何配置参数、该执行什么样的处理等事项。

这些事项不仅在新建程序时有用，在扩展、修改已有程序时也非常有用。对象之间通过方法交换信息，而至于这些信息在彼此内部是如何被处理的，则并不需要关心。在生成类时，我们只要牢记把适当的信息交给适当的方法处理，就可以设计出易于读写的程序。

重要的是如何自然地写出程序<sup>5</sup> 这除了需要丰富的程序设计经验外，还需要拥有设计模式等类结构相关的知识。“自然”这样的说法可能有点夸张，但通过指把事物的外部特征作为参考依据，我们就可以使用与实际事物相近的模型去组织、构建程序。

<sup>5</sup>这里指符合人脑思维的程序。一般认为，人脑的思维方式是面向对象的。——译者注。

# 第 9 章 运算符

接下来，我们将详细讨论一下 Ruby 的运算符。

在本章的前半部分，我们会补充介绍之前没介绍的运算符语法、以及逻辑运算符在 Ruby 中的一些习惯用法。

Ruby 的运算符能通过定义方法的方式来改变其原有的含义。在本章的后半部分，我们会讨论一下如何定义运算符。

## 9.1 赋值运算符

正如我们之前所介绍的那样，Ruby 的变量是在首次赋值的时候创建的。之后，程序可能会对变量引用的对象做各种各样的处理，甚至再次给变量赋值。例如，对 `a` 变量加 1，对 `b` 变量乘 2，如下所示：

```
a = a + 1  
b = b * 2
```

上面的表达式可被改写为以下形式：

```
a += 1  
b *= 2
```

大部分的二元运算符 `op` 都可以做如下转换。

```
var op= val  
↓  
var = var op val
```

将二元运算符与赋值组合起来的运算符称为赋值运算符。表 9.1 为常用的赋值运算符。

表 9.1 赋值运算符

<code>&amp;&amp;=</code>	<code>  =</code>	<code>^=</code>	<code>&amp;=</code>	<code> =</code>	<code>&lt;&gt;</code>	<code>&gt;&gt;=</code>
<code>+=</code>	<code>-=</code>	<code>\*=</code>	<code>/=</code>	<code>%=</code>	<code>\*\*=</code>	

除了变量之外，赋值运算符也同样适用于经由方法的对象操作。下面两个表达式是等效的。

```
$stdin.lineno += 1  
$stdin.lineno = $stdin.lineno + 1
```

请读者注意，上面的式子调用的是 `$stdin.lineno` 和 `$stdin.lineno=` 这两个方法。也就是说，使用赋值运算符的对象必须同时实现 `reader` 以及 `writer` 存取方法。

## 9.2 逻辑运算符的应用

在介绍逻辑运算符的应用例子之前，我们需要先来了解一下逻辑运算符的以下一些特征。

- 表达式的执行顺序是从左到右
- 如果逻辑表达式的真假已经可以确定，则不会再判断剩余的表达式
- 最后一个表达式的值为整体逻辑表达式的值

下面我们来详细讨论一下。首先，请看下面的与 `||` 相关的表达式。

条件 1 `||` 条件 2

上面的表达式一定会按照条件 1、条件 2 的顺序来判断表达式的值。条件 1 的判断结果为真时，不需要判断条件 2 的结果也可以知道整体表达式的结果为真。反过来说，只有当条件 1 的判断结果为假时，才需要判断条件 2。也就是说，Ruby 的逻辑运算符会避免做无谓的判断。下面我们来进一步扩展该逻辑表达式：

条件 1 `||` 条件 2 `||` 条件 3

这种情况下，只有当条件 1 和条件 2 两者都为假的时候，才会进行条件 3 的判断。这里的条件表达式指的是 Ruby 中所有的表达式。

```
var || "Ruby"
```

在上面的表达式中，首先会判断 `var` 的真假值，只有当 `var` 为 `nil` 或者 `false` 时，才会判断后面的字符串 `"Ruby"` 的真假值。之前我们也提到过，逻辑表达式的返回值为最后一个表达式的返回值，因此这个表达式的返回值为：

- `var` 引用对象时，`var` 的值

- `var` 为 `nil` 或者 `false` 时，字符串 "Ruby"

接下来，我们再来讨论一下 `&&`。基本规则与 `||` 是一样的。

#### 条件 1 && 条件 2

与 `||` 刚好相反，只有当条件 1 的判断结果为真时，才会判断条件 2。

下面是逻辑运算符的应用例子。假设希望给变量 `name` 赋值，一般我们会这么做：

```
name = "Ruby"      # 设定 name 的默认值
if var            # var 不是 nil 或者 false 时
  name = var      # 将 var 赋值给 name
end
```

使用 `||` 可以将这 4 行代码浓缩为一行代码。

```
name = var || "Ruby"
```

下面我们稍微修改一下程序，假设要将数组的首元素赋值给变量。

```
item = nil        # 设定 item 的初始值
if ary          # ary 不是 nil 或者 false 时
  item = ary[0]  # 将 ary[0] 赋值给 item
end
```

如果 `ary` 为 `nil`，则读取 `ary[0]` 时就会产生程序错误。在这个例子中，预先将 `item` 的值设定为了 `nil`，然后在确认 `ary` 不是 `nil` 后将 `ary[0]` 的值赋值给了 `item`。像这样的程序，通过使用 `&&`，只要像下面那样一行代码就可以搞定了：

```
item = ary && ary[0]
```

在确定对象存在后再调用方法的时候，使用 `&&` 会使程序的编写更有效率。从数学的角度上来看，下面的逻辑表达式表达的是一样的意思，但是从编程语言的角度来看却并不是一样的。

```
item = ary[0] && ary    # 错误的写法
```

最后，我们来看看 `||=` 的赋值运算符。

```
var ||= 1
```

和

```
var = var || 1
```

的运行结果是一样的。只有在 `var` 为 `nil` 或者 `false` 的时候，才把 1 赋值给它。这是给变量定义默认值的常用写法。

## 9.3 条件运算符

条件运算符 `?:` 的用法如下：

条件 ? 表达式 1 : 表达式 2

上面的表达式与下面使用 `if` 语句的表达式是等价的：

```
if 条件
  表达式 1
else
  表达式 2
end
```

例如，对比 `a` 与 `b` 的值，希望将比较大的值赋值给 `v` 时，程序可以像下面这样写：

```
a = 1
b = 2
v = (a > b) ? a : b
p v    #=> 2
```

虽然笔者比较喜欢这样简洁的写法，但如果表达式过于复杂就会使程序变得难懂，因此建议不要滥用此写法。条件运算符也称为三元运算符。

## 9.4 范围运算符

在 Ruby 中有表示数值范围的范围（range）对象。例如，我们可以像下面那样生成表示 1 到 10 的范围对象。

```
Range.new(1, 10)
```

用范围运算符可以简化范围对象的定义。以下写法与上面例子的定义是等价的：

```
1..10
```

我们在第 6 章 `for` 循环的例子中也使用过这个运算符。

```
sum = 0
for i in 1..5
  sum += i
end
puts sum
```

范围运算符有 `..` 和 `...` 两种。`x..y` 和 `x...y` 的区别在于，前者的范围是从 `x` 到 `y`；后者的范围则是从 `x` 到 `y` 的前一个元素。

对 `Range` 对象使用 `to_a` 方法，就会返回范围中从开始到结束的值。下面就让我们使用这个方法来确认一下 `..` 和 `...` 有什么不同。

```
p (5..10).to_a    #=> [5, 6, 7, 8, 9, 10]
p (5...10).to_a  #=> [5, 6, 7, 8, 9]
```

如果数值以外的对象也实现了根据当前值生成下一个值的方法，那么通过指定范围的起点与终点就可以生成 `Range` 对象。例如，我们可以用字符串对象生成 `Range` 对象。

```
p ("a".."f").to_a      #=> ["a", "b", "c", "d", "e", "f"]
p ("a"..."f").to_a    #=> ["a", "b", "c", "d", "e"]
```

在 `Range` 对象内部，可以使用 `succ` 方法根据起点值逐个生成接下来的值。具体来说就是，对 `succ` 方法的返回值调用 `succ` 方法，然后对该返回值再调用 `succ` 方法……直到得到的值比终点值大时才结束处理。

### 执行示例

```
> irb --simple-prompt
>> val = "a"
=> "a"
>> val = val.succ
=> "b"
>> val = val.succ
=> "c"
>> val = val.succ
=> "d"
```

## 9.5 运算符的优先级

运算符是有优先级的，表达式中有多个运算符时，优先级高的会被优先执行。例如四则运算中的“先乘除后加减”。表 9.2 是关于运算符的优先级的一些例子。把 Ruby 的运算符按照优先级由高到低的顺序进行排列，如图 9.1 所示。

表 9.2 运算符的优先级示例

表达式	含义	结果
<code>1 + 2 * 3</code>	<code>1 + (2 * 3)</code>	7
<code>"a" + "b" * 2 + "c"</code>	<code>"a" + ("b" * 2) + "c"</code>	"abbc"
<code>3 * 2 ** 3</code>	<code>3 * (2 ** 3)</code>	24
<code>2 + 3 &lt; 5 + 4</code>	<code>(2 + 3) &lt; (5 + 4)</code>	true

```
2 < 3 && 5 > 3
```

```
(2 < 3) && (5 > 3)
```

```
true
```

高



::

[]

+ (一元运算符) ! ~

\*\*

- (一元运算符)

\* / %

+ -

<< >>

&

| ^

> >= < <=

<=> == === != =~ !~

&&

||

? : (条件运算符)

... ...

= (包含+= -= \*= /=等)

not

and or

低

图 9.1 运算符的优先级

如果不按照优先级的顺序进行计算，可以用 `()` 将希望优先计算的部分括起来，当有多个 `()` 时，则从最内侧的 `()` 开始算起。因此，如果还未能熟练掌握运算符的优先顺序，建议多使用 `()`。

## 9.6 定义运算符

Ruby 的运算符大多都是作为实例方法提供给我们使用的，因此我们可以很方便地定义或者重定义运算符，改变其原有的含义。但是，表 9.3 中列举的运算符是不允许修改的。

表 9.3 不能重定义的运算符

<code>::</code>	<code>&amp;&amp;</code>	<code>  </code>	<code>..</code>	<code>...</code>	<code>?:</code>	<code>not</code>	<code>=</code>	<code>and</code>	<code>or</code>
-----------------	-------------------------	-----------------	-----------------	------------------	-----------------	------------------	----------------	------------------	-----------------

### 9.6.1 二元运算符

定义四则运算符等二元运算符时，会将运算符名作为方法名，按照定义方法的做法重定义运算符。运算符的左侧为接收者，右侧被作为方法的参数传递。在代码清单 9.1 的程序中，我们将为表示二元坐标的 `Point` 类定义运算符 `+` 以及 `-`。

代码清单 9.1 point.rb

```
class Point
  attr_reader :x, :y

  def initialize(x=0, y=0)
    @x, @y = x, y
  end

  def inspect # 用于显示
    "(#{x}, #{y})"
  end

  def +(other) # x、y 分别进行加法运算
    self.class.new(x + other.x, y + other.y)
  end

  def -(other) # x、y 分别进行减法运算
    self.class.new(x - other.x, y - other.y)
  end

  point0 = Point.new(3, 6)
  point1 = Point.new(1, 8)

  p point0      #=> (3, 6)
  p point1      #=> (1, 8)
  p point0 + point1 #=> (4, 14)
  p point0 - point1 #=> (2, -2)
```

如上所示，定义二元运算符时，我们常把参数名定义为 `other`。

在定义运算符 `+` 和 `-` 的程序中，创建新的 `Point` 对象时，我们使用了 `self.class.new`。而像下面这样，直接使用 `Point.new` 方法也能达到同样的效果。

```
def +(other)
  Point.new(x + other.x, y + other.y)
end
```

使用上面的写法时，返回值一定是 `Point` 对象。如果 `Point` 类的子类使用了 `+` 和 `-`，则返回的对象应该属于 `Point` 类的子类，但是这样的写法却只能返回 `Point` 类的对象。在方法内创建当前类的对象时，不直接写类名，而是使用 `self.class`，那么创建的类就是实际调用 `new` 方法时的类，这样就可以灵活地处理继承与 Mix-in 了。

### 专栏

#### puts 方法与 p 方法的不同点

代码清单 9.1 中定义了用于显示的 `inspect` 方法，在 `p` 方法中把对象转换为字符串时会用到该方法。另外，使用 `to_s` 方法也可以把对象转换为字符串，在 `puts`、`print` 方法中都有使用 `to_s` 方法。下面我们来看看两者的区别。

```
> irb --simple-prompt
>> str = "Ruby 基础教程"
=> "Ruby 基础教程"
>> str.to_s
=> "Ruby 基础教程"
>> str.inspect
=> "\"Ruby 基础教程\""
```

`String#to_s` 的返回结果与原字符串相同，但 `String#inspect` 的返回结果中却包含了 `\"`。这是因为 `p` 方法在输出字符串时，为了让我们更明确地知道输出的结果就是字符串而进行了相应的处理。这两个方法的区别在于，作为程序运行结果输出时用 `to_s` 方法；给程序员确认程序状态、调查对象内部信息等时用 `inspect` 方法。

除了 `puts` 方法、`print` 方法外，`to_s` 方法还被广泛应用在 `Array#join` 方法等内部需要做字符串处理的方法中。

`inspect` 方法可以说是主要使用 `p` 方法进行输出的方法。例如，`irb` 命令的各行结果的显示就用到了 `inspect` 方法。我们在写程序的时候，如果能根据实际情况选择适当的方法，就会达到事半功倍的效果。

## 9.6.2 一元运算符

可定义的一元运算符有 `+`、`-`、`~`、`!` 4 个。它们分别以 `+@`、`-@`、`~@`、`!@` 为方法名进行方法的定义。下面就让我们试试在 `Point` 类中定义这几个运算符（代码清单 9.2）。这里需要注意的是，一元运算符都是没有参数的。

代码清单 9.2 `point.rb` (部分)

```
class Point
|
def +@
  dup                      # 返回自己的副本
end

def -@
  self.class.new(-x, -y)   # 颠倒x、y各自的正负
end

def ~@
  self.class.new(-y, x)    # 使坐标翻转90度
end

point = Point.new(3, 6)
p +point  #=> (3, 6)
p -point  #=> (-3, -6)
p ~point  #=> (-6, 3)
```

## 9.6.3 下标方法

数组、散列中的 `obj[i]` 以及 `obj[i]=x` 这样的方法，称为下标方法。定义下标方法时的方法名分别为 `[]` 和 `[]=`。在代码清单 9.3 中，我们将会定义 `Point` 类实例 `pt` 的下标方法，实现以 `v[0]` 的形式访问 `pt.x`，以 `v[1]` 的形式访问 `pt.y`。

代码清单 9.3 `point.rb` (部分)

```
class Point
|
def [](index)
  case index
  when 0
    x
  when 1
    y
  else
    raise ArgumentError, "out of range `#{index}'"
  end
end

def []=(index, val)
  case index
  when 0
    self.x = val
  when 1
    self.y = val
  else
    raise ArgumentError, "out of range `#{index}'"
  end
end

point = Point.new(3, 6)
p point[0]          #=> 3
p point[1] = 2     #=> 2
p point[1]          #=> 2
p point[2]          #=> 错误 (ArgumentError)
```

参数 `index` 代表的是数组的下标。由于本例中的类只有两个元素，因此当索引值指定 2 以上的数值时，程序就会认为是参数错误并抛出异常。

# 第 10 章 错误处理与异常

程序在运行时会伴随着各种各样的错误发生。如果我们在编写程序时不犯任何错误，并且所有处理都能正常执行的话，那么就不会产生程序错误，但实际上并不可能有这么完美的程序。在本章中，我们将围绕着程序错误及其应对方法，向大家介绍一下 Ruby 异常处理的相关内容。

## 10.1 关于错误处理

在介绍实际的程序例子前，我们先来了解一下程序错误相关的基础知识。在程序执行的过程中，通常会有以下错误发生：

- **数据错误**

在计算家庭收支的时候，若在应该写金额的一栏上填上了商品名，那么就无法计算。此外，HTML 这种格式的数据的情况下，如果存在没有关闭标签等语法错误，也会导致无法处理数据。

- **系统错误**

硬盘故障等明显的故障，或者没把 CD 插入到驱动器等程序无法恢复的问题。

- **程序错误**

因调用了不存在的方法、弄错参数值或算法错误而导致错误结果等，像这样，程序本身的缺陷也可能会导致错误。

程序在运行时可能会遇到各种各样的错误。如果对这些错误放任不管，大部分程序都无法正常运行，因此我们需要对这些错误做相应的处理。

- **排除错误的原因**

在文件夹中创建文件时，如果文件夹不存在，则由程序本身创建文件夹。如果程序无法创建文件夹，则需要再考虑其他解决方法。

- **忽略错误**

程序有时候也会有一些无伤大雅的错误。例如，假设运行程序时需要读取某个配置文件，如果我们事前已经在程序中准备好了相应配置的默认值，那么即使无法读取该设定文件，程序也可以忽略这个错误。

- **恢复错误发生前的状态**

向用户提示程序发生错误，指导用户该如何进行下一步处理。

- **重试一次**

曾经执行失败的程序，过一段时间后再重新执行可能就会成功。

- **终止程序**

只是自己一个人用的小程序，也许本来就没必要做错误处理。

而至于实际应该采取何种处理，则要根据程序代码的规模、应用程序的性质来决定，不能一概而论。但是，对于可预期的错误，我们需要留意以下两点：

- **是否破坏了输入的数据，特别是人工制作的数据。**

- **是否可以对错误的内容及其原因做出相应的提示。**

覆盖了原有文件、删除了花费大量时间输入的数据等，像这样的重要数据的丢失、破坏可以说是灾难性的错误。另外，如果错误是由用户造成的，或者程序自身不能修复的话，给用户简明易懂的错误提示，会大大提升程序的用户体验。

Ruby 为我们提供了异常处理机制，可以使我们非常方便地应对各种错误。

## 10.2 异常处理

在程序执行的过程中，如果程序出现了错误就会发生异常。异常发生后，程序会暂时停止运行，并寻找是否有对应的异常处理程序。如果有则执行，如果没有，程序就会显示类似以下信息并终止运行。

### 执行示例

```
> ruby test.rb
test.rb:2:in `initialize': No such file or directory - /no/file(Errno::ENOENT)
    from test.rb:2:in `open'
    from test.rb:2:in `foo'
    from test.rb:2:in `bar'
    from test.rb:9:in `main'
```

该信息的格式如下：

文件名: 行号 :in 方法名: 错误信息 (异常类名)

```
from 文件名:行号:in 方法名
```

以 `from` 开头的行表示发生错误的位置。

没有异常处理的编程语言的情况下，编程时就需要逐个确认每个处理是否已经处理完毕（图 10.1）。在这类编程语言中，大部分程序代码都被花费在错误处理上，因此往往会使程序变得繁杂。

## ●没有异常处理的语言

```
if a() == false  
    错误处理  
end  
if b() == false  
    错误处理  
end  
if c() == false  
    错误处理  
end
```

`a()`, `b()`, `c()`发生  
错误时返回`false`

## ●Ruby中的异常处理

```
begin  
    a()  
    b()  
    c()  
rescue  
    错误处理  
end
```

`a()`, `b()`, `c()`中  
发生异常

图 10.1 异常处理

异常处理有以下优点：

- 程序不需要逐个确认处理结果，也能自动检查出程序错误
- 会同时报告发生错误的位置，便于排查错误
- 正常处理与错误处理的程序可以分开书写，使程序便于阅读

### 10.3 异常处理的写法

Ruby 中使用 `begin ~ rescue ~ end` 语句描述异常处理。

```
begin  
    可能会发生异常的处理  
rescue  
    发生异常时的处理  
end
```

在 Ruby 中，异常及其相关信息都是被作为对象来处理的。在 `rescue` 后指定变量名，可以获得异常对象。

```
begin  
    可能会发生异常的处理  
rescue => 引用异常对象的变量  
    发生异常时的处理  
end
```

即使不指定变量名，Ruby 也会像表 10.1 那样把异常对象赋值给变量 `$!`。不过，把变量名明确地写出来会使程序更加易懂。

表 10.1 异常发生时被自动赋值的变量

变量	意义
----	----

\$!	最后发生的异常（异常对象）
\$@	最后发生的异常的位置信息

此外，通过调用表 10.2 中的异常对象的方法，就可以得到相关的异常信息。

表 10.2 异常对象的方法

方法名	意义
class	异常的种类
message	异常信息
backtrace	异常发生的位置信息（\$@ 与 \$!.backtrace 是等价的）

代码清单 10.1 是 Unix 的 wc 命令的简易版。结果会输出参数中指定的各文件的行数、单词数、字数（字节数），最后输出全部文件的统计结果。

#### 执行示例

```
> ruby wc.rb intro.rb sec01.rb sec02.rb
      50      67      1655 intro.rb
      81      92      3455 sec01.rb
     123     162      3420 sec02.rb
     254     321      8520 total
```

代码清单 10.1 wc.rb

```
ltotal=0                      # 行数合计
wtotal=0                      # 单词数合计
ctotal=0                      # 字数合计
ARGV.each do |file|
  begin
    input = File.open(file)      # 打开文件 (A)
    l=0                          # file 内的行数
    w=0                          # file 内的单词数
    c=0                          # file 内的字数
    input.each_line do |line|
      l += 1
      c += line.size
      line.gsub!(/^\s+/, "")      # 删除行首的空白符
      ary = line.split(/\s+/)      # 用空白符分解
      w += ary.size
    end
    input.close                  # 关闭文件
    printf("%8d %8d %8d %s\n", l, w, c, file)  # 整理输出格式
    ltotal += l
    wtotal += w
    ctotal += c
  rescue => ex
    print ex.message, "\n"        # 输出异常信息 (B)
  end
end

printf("%8d %8d %8d %s\n", ltotal, wtotal, ctotal, "total")
```

在 (A) 处无法打开文件时，程序会跳到 rescue 部分。这时，异常对象被赋值给变量 ex，(B) 部分的处理被执行。

如果程序中指定了不存在的文件，则会提示发生错误，如下所示。提示发生错误后，并不会马上终止程序，而是继续处理下一个文件。

#### 执行示例

```
> ruby wc.rb intro.rb sec01.rb sec02.rb sec03.rb
```

```

50      67      1655 intro.rb
81      92      3455 sec01.rb
123     188     3729 sec02.rb
No such file or directory - sec03.rb
254     321     8520 total

```

如果发生异常的方法中没有 `rescue` 处理，程序就会逆向查找调用者中是否定义了异常处理。下面来看看图 10.2 这个例子。调用 `foo` 方法，尝试打开一个不存在的文件。若 `File.open` 方法发生异常，那么该异常就会跳过 `foo` 方法以及 `bar` 方法，被更上一层的 `rescue` 捕捉。

```

def foo
  File.open("/no/file")
end

def bar
  foo()
end

begin
  bar()
rescue => ex
  print ex.message, "\n"
end

```

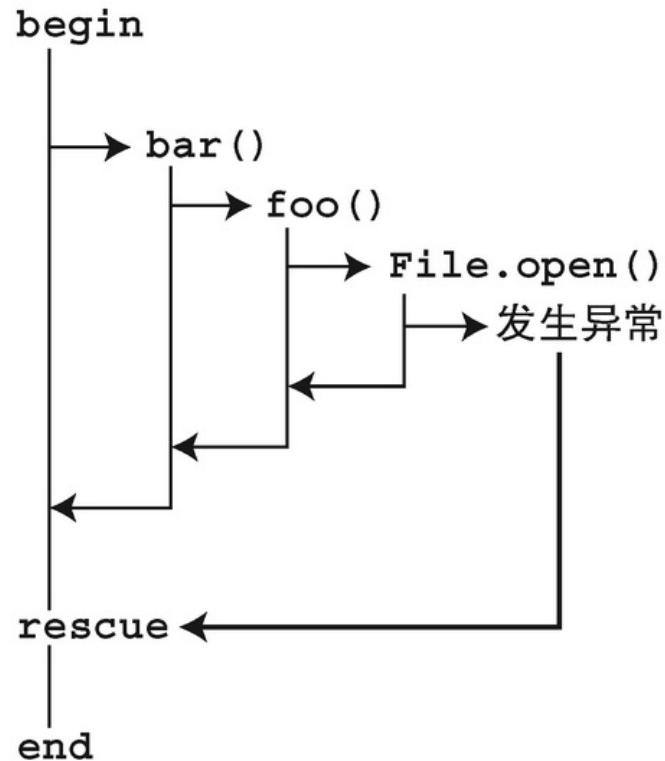


图 10.2 异常处理的流程

然而，并不是说每个方法都需要做异常处理，只需根据实际情况在需要留意的地方做就可以了。在并不特别需要解决错误的情况下，也可以不捕捉异常。当然，不捕捉异常就意味着如果有问题发生程序就会马上终止。

## 10.4 后处理

不管是否发生异常都希望执行的处理，在 Ruby 中可以用 `ensure` 关键字来定义。

```

begin
  有可能发生异常的处理
rescue => 变量
  发生异常后的处理
ensure
  不管是否发生异常都希望执行的处理
end

```

现在，假设我们要实现一个拷贝文件的方法，如下所示。下面的 `copy` 方法是把文件从 `from` 拷贝到 `to`。

```

def copy(from, to)
  src = File.open(from)          # 打开原文件from (A)
begin
  dst = File.open(to, "w")      # 打开目标文件to (B)
  data = src.read
  dst.write(data)
  dst.close
ensure
  src.close                     # (C)
end
end

```

在 (A) 部分，如果程序不能打开原文件，那么就会发生异常并把异常返回给调用者。这时，不管接下来的处理是否能正常执行，`src` 都必须得关闭。关闭 `src` 的处理在 (C) 部分执行。`ensure` 中的处理，在程序跳出 `begin ~ end` 部分时一定会被执行。即使 (B) 中的目标文件无法打开，(C) 部分的处理也同样会被执行。

## 10.5 重试

在 `rescue` 中使用 `retry` 后，`begin` 以下的处理会再重做一遍。

在下面的例子中，程序每隔 10 秒执行一次 `File.open`，直到能成功打开文件为止，打开文件后再读取其内容。

```
file = ARGV[0]
begin
  io = File.open(file)
rescue
  sleep 10
  retry
end

data = io.read
io.close
```

不过需要注意的是，如果指定了无论如何都不能打开的文件，程序就会陷入死循环中。

## 10.6 rescue 修饰符

与 `if` 修饰符、`unless` 修饰符一样，`rescue` 也有对应的修饰符。

`表达式 1 rescue 表达式 2`

如果表达式 1 中发生异常，表达式 2 的值就会成为整体表达式的值。也就是说，上面的式子与下面的写法是等价的：

```
begin
  表达式 1
rescue
  表达式 2
end
```

我们再来看看下面的例子：

```
n = Integer(val) rescue 0
```

`Integer` 方法当接收到 "123" 这种数值形式的字符串参数时，会返回该字符串表示的整数值，而当接收到 "abc" 这种非数值形式的字符串参数时，则会抛出异常（在判断字符串是否为数值形式时经常用到此方法）。在本例中，如果 `val` 是不正确的数值格式，就会抛出异常，而 0 则作为 = 右侧整体表达式的返回值。像这样，这个小技巧经常被用在不需要过于复杂的处理，只是希望简单地对变量赋予默认值的时候。

## 10.7 异常处理语法的补充

如果异常处理的范围是整个方法体，也就是说整个方法内的程序都用 `begin ~ end` 包含的话，我们就可以省略 `begin` 以及 `end`，直接书写 `rescue` 与 `ensure` 部分的程序。

```
def foo
  方法体
rescue => ex
  异常处理
ensure
  后处理
end
```

同样，我们在类定义中也可以使用 `rescue` 以及 `ensure`。但是，如果类定义途中发生异常，那么异常发生部分后的办法定义就不会再执行了，因此一般我们不会在类定义中使用它们。

```
class Foo
  类定义
rescue => ex
  异常处理
ensure
  后处理
end
```

## 10.8 指定需要捕捉的异常

当存在多个种类的异常，且需要按异常的种类分别进行处理时，我们可以用多个 `rescue` 来分开处理。

```
begin
  可能发生异常的处理
```

```
rescue Exception1, Exception2 => 变量  
    对 Exception1 或者 Exception2 的处理  
rescue Exception3 => 变量  
    对 Exception3 的处理  
rescue  
    对上述异常以外的异常的处理  
end
```

通过直接指定异常类，可以只捕捉我们希望处理的异常。

```
file1 = ARGV[0]  
file2 = ARGV[1]  
begin  
    io = File.open(file1)  
rescue Errno::ENOENT, Errno::EACCES  
    io = File.open(file2)  
end
```

在本例中，程序如果无法打开 `file1` 就会打开 `file2`。程序中捕捉的 `Errno::ENOENT` 以及 `Errno::EACCES`，分别是文件不存在以及没权限打开文件时发生的异常。

## 10.9 异常类

之前我们提到过异常也是对象。Ruby 中所有的异常都是 `Exception` 类的子类，并根据程序错误的种类来定义相应的异常。图 10.3 为 Ruby 标准库中的异常类的继承关系。

# Exception

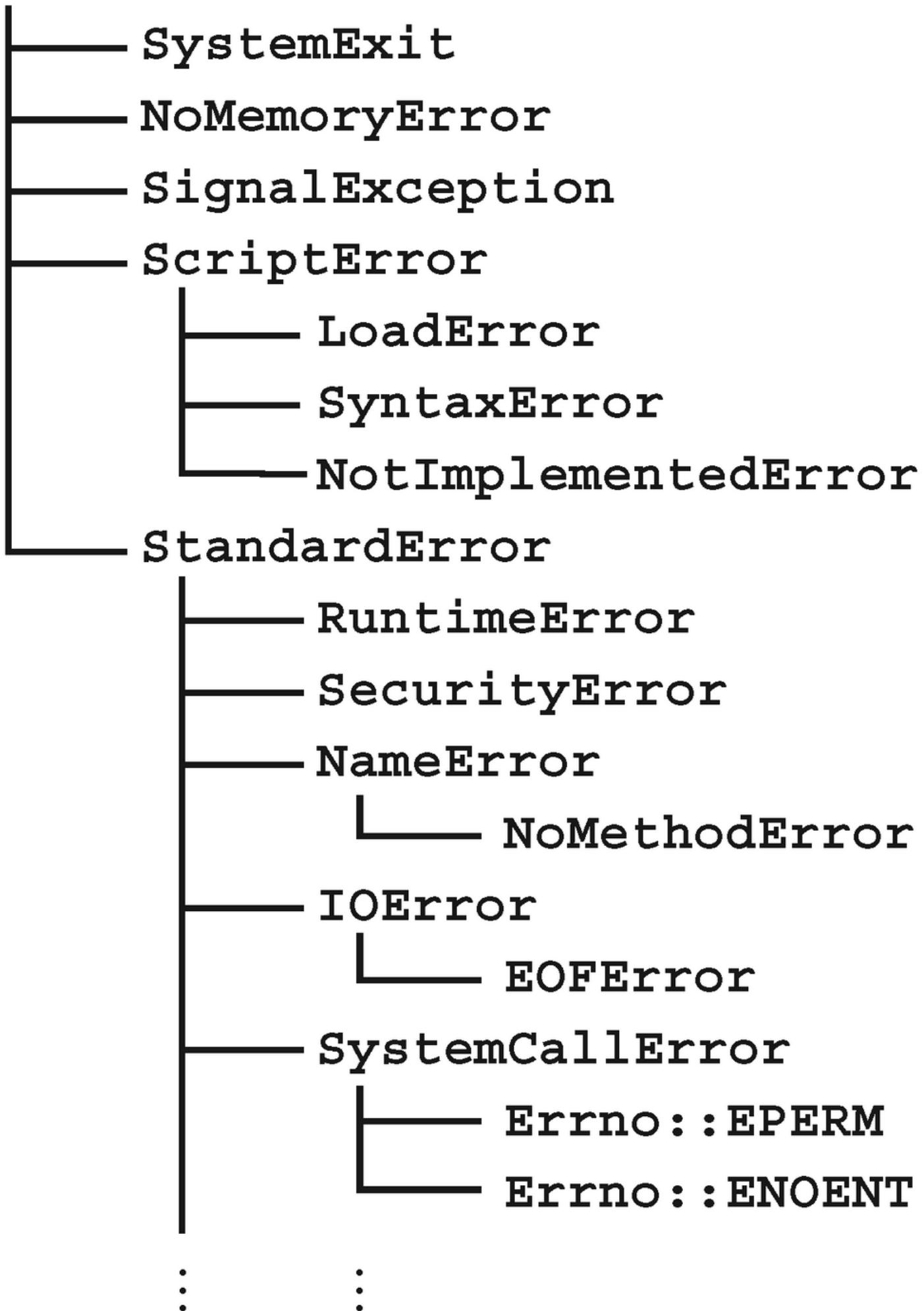


图 10.3 异常类的继承关系

在 `rescue` 中指定的异常的种类实际上就是异常类的类名。`rescue` 中不指定异常类时，程序会默认捕捉 `StandardError` 类及其子类的异常。

`rescue` 不只会捕捉指定的异常类，同时还会捕捉其子类。因此，我们在自己定义异常时，一般会先定义继承 `StandardError` 类的新类，然后再继承这个新类。

```
MyError = Class.new(StandardError)    # 新的异常类
MyError1 = Class.new(MyError)
MyError2 = Class.new(MyError)
MyError3 = Class.new(MyError)
```

这样定义后，通过以下方式捕捉异常的话，同时就会捕捉 `MyError` 类的子类 `MyError1`、`MyError2`、`MyError3` 等。

```
begin
|
rescue MyError
|
end
```

在本例中，

```
MyError = Class.new(StandardError)
```

上述写法的作用是定义一个继承 `StandardError` 类的新类，并将其赋值给 `MyError` 常量。这与使用在第 8 章中介绍过的 `class` 语句定义类的效果是一样的。

```
class MyError < StandardError
end
```

使用 `class` 语句，我们可以进行定义方法等操作，但在本例中，由于我们只需要生成继承 `StandardError` 类的新类就可以了，所以就向大家介绍了这个只需 1 行代码就能实现类的定义的简洁写法。

## 10.10 主动抛出异常

使用 `raise` 方法，可以使程序主动抛出异常。在基于自己判定的条件抛出异常，或者把刚捕捉到的异常再次抛出并通知异常的调用者等情况下，我们会使用 `raise` 方法。

`raise` 方法有以下 4 种调用方式：

- **`raise message`**

抛出 `RuntimeError` 异常，并把字符串作为 `message` 设置给新生成的异常对象。

- **`raise 异常类`**

抛出指定的异常。

- **`raise 异常类, message`**

抛出指定的异常，并把字符串作为 `message` 设置给新生成的异常对象。

- **`raise`**

在 `rescue` 外抛出 `RuntimeError`。在 `rescue` 中调用时，会再次抛出最后一次发生的异常 (`$!`)。

# 第 11 章 块

Ruby 中大量使用了块（block）。块原本只是为了循环而产生的语法结构，但现在程序中许多地方也都使用了块。因此，如何灵活地使用块，也是 Ruby 的重点之一。

下面就让我们来讨论一下块的作用及其用途。

## 11.1 块是什么

块就是在调用方法时，能与参数一起传递的多个处理的集合。之前在介绍 `each` 方法、`time` 方法等与循环有关的部分时，我们就已经接触过块。接收块的方法会执行必要次数的块。块的执行次数由方法本身决定，因此不需事前指定，甚至有可能一次都不执行。

1有时也称代码块。——译者注

在下面的例子中，我们使用 `each` 方法，把保存在 `Array` 对象中的各个整数依次取 2 次幂后输出。`do` 和 `end` 之间的部分就是所谓的块。在本例中，块总共被执行了 5 次。

```
[1, 2, 3, 4, 5].each do |i|
  puts i ** 2
end
```

正如在第 7 章中所介绍的那样，我们把这样的方法调用称为“调用带块的方法”或者“调用块”。块的调用方法一般采用以下形式。

对象.方法名(参数列表) do |块变量|  
希望循环的处理

`end`

或者

对象.方法名(参数列表){|块变量|  
希望循环的处理  
}

块的开头是块变量。块变量就是在执行块的时候，从方法传进来的参数。不同方法的块变量个数也不相同。例如，在 `Array#each` 方法中，数组的元素会作为块变量被逐个传递到块中。而在 `Array#each_with_index` 方法中，则是 [元素，索引] 两个值被传递到块中。

### 执行示例

```
> irb --simple-prompt
>> ary = ["a", "b", "c"] ← 对 ary 变量赋值
=> ["a", "b", "c"]
>> ary.each{|obj| p obj } ← Array#each 的例子
  "a"
  "b"
  "c"
=> ["a", "b", "c"]
>> ary.each_with_index{|obj, idx| p [obj, idx]} ← Array#each_with_index 的例子
?>   p [obj, idx]
>> []
  ["a", 0]
  ["b", 1]
  ["c", 2]
=> ["a", "b", "c"]
```

而在第 6 章中介绍的 `loop` 方法则不需要传递块变量。

## 11.2 块的使用方法

### 11.2.1 循环

在 Ruby 中，我们常常使用块来实现循环。在接收块的方法中，实现了循环处理的方法称为迭代器（iterator）。`each` 方法就是一个典型的迭代器。

在下面的例子中，我们把数组的各个元素转换为大写后输出。

```
alphabet = ["a", "b", "c", "d", "e"]
alphabet.each do |i|
```

```
puts i.upcase  
end
```

和数组一样，散列也能将元素一个个拿出来，但与数组不同的是，散列会将 [key, value] 的组合作为数组来提取元素。如代码清单 11.1 所示，可以成对地提取散列的全部键、值。本例中使用 `pair[1]` 提取并合计了散列的值，提取散列的键时则可以使用 `pair[0]`。

#### 代码清单 11.1 hash\_each.rb

```
sum = 0  
outcome = {"参加费"=>1000, "挂件费用"=>1000, "联欢会费用"=>4000}  
outcome.each do |pair|  
  sum += pair[1] # 指定值  
end  
puts "合计: #{sum}"
```

在接收块变量时，多重赋值规则也是同样适用的。我们稍微把代码清单 11.1 的程序修改一下，使之变成代码清单 11.2 那样，这样一来，键、值就可以被分别赋值给不同的变量了。

#### 代码清单 11.2 hash\_each2.rb

```
sum = 0  
outcome = {"参加费"=>1000, "挂件费用"=>1000, "联欢会费用"=>4000}  
outcome.each do |item, price|  
  sum += price  
end  
puts "合计: #{sum}"
```

`File` 对象被用于读写文件的内容。使用 `File` 对象可将文件数据从头到尾读取出来。

根据文件内容的不同，我们需要考虑是以字符为单位，还是以行为单位来做读取处理。代码清单 11.3 是使用了 `File` 类的 `each` 方法的一个程序示例，它会把 `sample.txt` 文件的内容按顺序逐行读取出来并输出。

#### 代码清单 11.3 file\_each.rb

```
file = File.open("sample.txt")  
file.each_line do |line|  
  print line  
end  
file.close
```

除了 `each_line` 方法外，`File` 对象中还有以字符为单位来循环读取数据的 `each_char` 方法、以及以字节为单位进行循环读取的 `each_byte` 方法等等。而其他对象也有很多以 `each_XX` 命名的循环读取数据的方法。

#### 11.2.2 隐藏常规处理

上文中我们介绍了将块用于循环的迭代器的例子。但正如本章开头所介绍的那样，除了迭代器以外，块还被广泛使用在其他地方。其中一个用法就是确保后处理被执行。下面我们来看一个典型的例子——`File.open` 方法。`File.open` 方法在接收块后，会将 `File` 对象作为块变量，并执行一次块。这里，我们可以使用块把代码清单 11.3 改写为代码清单 11.4 那样。

#### 代码清单 11.4 file\_open.rb

```
File.open("sample.txt") do |file|  
  file.each_line do |line|  
    print line  
  end  
end
```

与改写之前的程序相比，`File` 对象读取数据的部分一样，不同点在于没有了最后的 `close` 方法的调用。如果使用完打开的文件后没有将文件关闭的话，有可能会产生其他程序无法打开该文件，或者到达一次性可打开的文件数的上限时无法再打开新文件等问题。而在代码清单 11.4 的程序中，即使遇到无法打开文件等错误也可以正常关闭文件，因为块内部进行了程序清单 11.5 那样的处理。

#### 代码清单 11.5 file\_open\_no\_block.rb

```
file = File.open("sample.txt")  
begin  
  file.each_line do |line|  
    print line  
  end  
ensure  
  file.close  
end
```

`File.open` 方法使用块时，块内部的处理完毕并跳出方法前，文件会被自动关闭，因此就不需要像代码清单 11.3 那样使用 `File#close` 方法。

文件使用完毕后，由方法执行关闭操作，而我们只需将必要的处理记述在块中即可。这样一来可以减少程序的代码量，二来可以防止忘记关闭文件等错误的发生。

### 11.2.3 替换部分算法

下面我们再来介绍一个块的常见用法。这一次我们以数组排序为例，来了解一下指定处理顺序时块的使用方法。

- **自定义排列顺序**

`Array` 类的 `sort` 方法是对数组内元素进行排序的方法。对数组元素进行排序，可以采取多种方法。

- 按数字的大小顺序
- 按字母顺序
- 按字符串的长度顺序
- 按数组元素的合计值的大小顺序

如果按照这样的条件分别定义相应的排序方法，就会使方法的数量过多，不便于记忆。因此，在 `Array#sort` 方法中，元素的排序步骤由方法决定，用户只能指定元素间关系的比较逻辑。

`Array#sort` 方法没有指定块时，会使用 `<=>` 运算符对各个元素进行比较，并根据比较后的结果进行排序。`<=>` 运算符的返回值为 `-1`、`0`、`1` 中的一个。

表 11.1 `a <=> b` 的结果

<code>a &lt;=&gt;</code> 时	<code>-1</code> (比 0 小)
<code>a == b</code> 时	<code>0</code>
<code>a &gt; b</code> 时	<code>1</code> (比 0 大)

使用 `<=>` 运算符比较字符串时，会按照字符编码的顺序进行比较。比较字母时，会按先大写字母后小写字母的顺序排列。

```
array = ["ruby", "Perl", "PHP", "Python"]
sorted = array.sort
p sorted    #=> ["PHP", "Perl", "Python", "ruby"]
```

我们可以通过调用块来指定排列顺序。下面的例子与不使用块时的执行结果是一样的。

```
array = ["ruby", "Perl", "PHP", "Python"]
sorted = array.sort{ |a, b| a <=> b }
p sorted    #=> ["PHP", "Perl", "Python", "ruby"]
```

在 `sort` 方法的末尾添加了块 `{ |a, b| a <=> b }`，`sort` 方法会根据块的执行结果判断元素的大小关系。当需要比较元素的大小关系时，块中需要比较的两个对象就会被作为块变量调用。对块变量 `a` 和 `b` 进行比较后，数组整体就会按该顺序排列。

在这里，我们需要注意块中最后一个表达式的值就是块的执行结果，因此 `<=>` 运算符必须在最后一行使用。

**备注** 块的最后一个表达式不是指块的最后一行表达式，而是指在块中最后执行的表达式。

按字符串的长度排序时，可以采用如下方法。

```
array = ["ruby", "Perl", "PHP", "Python"]
sorted = array.sort{ |a, b| a.length <=> b.length }
p sorted    #=> ["PHP", "ruby", "Perl", "Python"]
```

在之前的例子中，我们只是单纯地比较了字符串 `a`、`b`，这里我们使用 `String#length` 方法，来比较字符串的长度。用 `<=>` 运算符比较数值时，得到的是由小到大的排列顺序，因此，比较字符串长度时，结果就是按照由短到长的顺序进行排列。

像这样，块经常被用来在 `sort` 方法中实现自定义排列顺序。

- **预先取出排序所需的信息**

我们再来详细看看 `sort` 方法的块。每次比较元素时，`sort` 方法都会调用一次将两个元素作为块变量的块。这里，我们仍以刚才介绍的按字符串长度排序的程序为例，来看看程序调用了 `length` 方法多少次。

#### 代码清单 11.6 `sort_comp_count.rb`

```
ary = %w(
  Ruby is a open source programming language with a focus
```

```

on simplicity and productivity. It has an elegant syntax
that is natural to read and easy to write
)

call_num = 0      # 块的调用次数
sorted = ary.sort do |a, b|
  call_num += 1 # 累加块的调用次数
  a.length <= b.length
end

puts "排序结果 #{sorted}"
puts "数组的元素数量 #{ary.length}"
puts "调用块的次数 #{call_num}"

```

### 执行示例

```

> ruby sort_comp_count.rb
排序结果 ["a", "a", "on", "to", "It", "to", "is", "an", ....]
数组的元素数量 28
调用块的次数 91

```

可以看出，在这个例子中，我们对 28 个元素进行了排序，块总共被调用了 91 次。由于每调用 1 次块，`length` 方法就会被调用 2 次，因此最终就会被调用 182 次。而实际上，我们只需对所有的字符串都调用 1 次 `length` 方法，然后再用得出的结果进行排序就可以了。像这样，在能够通过 `<=` 运算符对转换后的结果进行比较的情况下，使用 `sort_by` 方法会使排序更加有效率。

```

ary = %w(
  Ruby is a open source programming language with a focus
  on simplicity and productivity. It has an elegant syntax
  that is natural to read and easy to write
)
sorted = ary.sort_by{ |item| item.length }
p sorted

```

`sort_by` 方法会将每个元素在块中各调用一次，然后再根据这些结果做排序处理。这种情况下，虽然比较的次数不变，但获取排序所需要的信息的次数（本例中为 28 次）只需与元素个数一样就可以了。

总结一下，元素排序算法中公共的部分由方法本身提供，我们则可以用块来替换方法中元素排列的顺序（或者取得用于比较的信息），或者根据不同的目的来替换需要更改的部分。

## 11.3 定义带块的方法

在第 7 章中我们简单地介绍了如何定义带块的方法，接下来我们就来详细地讨论一下。

### 11.3.1 执行块

首先让我们重温一下第 7 章中的 `myloop` 方法（代码清单 11.7）。

#### 代码清单 11.7 myloop.rb

```

def myloop
  while true
    yield          # 执行块
  end
end

num = 1            # 初始化num
myloop do
  puts "num is #{num}" # 输出num
  break if num > 100 # num 超过100 后跳出循环
  num *= 2          # num 乘2
end

```

`myloop` 方法在执行 `while` 循环的同时执行了 `yield` 关键字，`yield` 关键字的作用就是执行方法的块。因为这个 `while` 循环的条件固定为 `true`，所以会无限循环地执行下去，但只要在块里调用 `break`，就可以随时中断 `myloop` 方法，来执行后面的处理。

### 11.3.2 传递块参数，获取块的值

在刚才的例子中，块参数以及块的执行结果都没有被使用。接下来，我们会定义一个方法，该方法接收两个整数参数，并对这两个整数之间的整数做某种处理后进行合计处理，而“某种处理”则由块指定（代码清单 11.8）。

#### 代码清单 11.8 total.rb

```
1: def total(from, to)
```

```

2:   result = 0          # 合计值
3:   from.upto(to) do |num| # 处理从from 到to 的值
4:     if block_given?      # 如果有块的话
5:       result += yield(num) # 累加经过块处理的值
6:     else                  # 如果没有块的话
7:       result += num      # 直接累加
8:     end
9:   end
10:  return result        # 返回方法的结果
11: end
12:
13: p total(1, 10)         # 从1 到10 的和 => 55
14: p total(1, 10){|num| num ** 2 } # 从1 到10 的2 次幂的和 => 385

```

`total` 方法会先使用 `Integer#upto` 方法把 `from` 到 `to` 之间的整数值按照从小到大的顺序取出来，然后交给块处理，最后再将块处理后的值累加到变量 `result`。程序第 5 行中，对 `yield` 传递参数后，参数值就会作为块变量传递到块中。同时，块的运行结果也会作为 `yield` 的结果返回。

程序第 4 行的 `block_given?` 方法被用来判断调用该方法时是否有块被传递给方法，如果有则返回 `true`，反之返回 `false`。如果方法没有块，则在程序第 7 行中直接把 `num` 相加。

在本例中，对 `yield` 传递 1 个参数，就有 1 个块变量接收。下面我们来看看对 `yield` 传递 0 个、1 个、3 个等多个参数时，对应的块变量是如何进行接收的（代码清单 11.9）。

#### 代码清单 11.9 block\_args\_test.rb

```

def block_args_test
  yield()          # 0 个块变量
  yield(1)         # 1 个块变量
  yield(1, 2, 3)   # 3 个块变量
end

puts "通过|a| 接收块变量"
block_args_test do |a|
  p [a]
end
puts

puts "通过|a, b, c| 接收块变量"
block_args_test do |a, b, c|
  p [a, b, c]
end
puts

puts "通过|*a| 接收块变量"
block_args_test do |*a|
  p [a]
end
puts

```

#### 执行示例

```

> ruby block_args_test.rb
通过|a| 接收块变量
[nil]
[1]
[1]

通过|a, b, c| 接收块变量
[[nil, nil, nil]]
[[1, nil, nil]]
[[1, 2, 3]]

通过|*a| 接收块变量
[[[]]]
[[[1]]]
[[[1, 2, 3]]]

```

首先我们注意到，`yield` 参数的个数与块变量的个数是不一样的。从 `|a|` 和 `|a, b, c|` 的例子中可以看出，块变量比较多时，多出来的块变量值为 `nil`，而块变量不足时，则不能接收参数值。

最后的通过 `|*a|` 接收的情况是将所有块变量整合为一个数组来接收。这与定义方法时接收可变参数的情况非常相似。

另外，在第 4 章中介绍的抽取嵌套数组的元素的规则，同样也适用于块变量。例如，`Hash#each_with_index` 方法的块变量有 2 个，并以 `yield([键, 值], 索引)` 的形式传递。像代码清单 11.10 那样，在接收块变量后，我们就可以把 `[键, 值]` 部分分别赋值给不同的变量。

## 代码清单 11.10 param\_grouping.rb

```
hash = {a: 100, b: 200, c: 300}
hash.each_with_index do |(key, value), index|
  p [key, value, index]
end
```

### 执行示例

```
> ruby param_grouping.rb
[:a, 100, 0]
[:b, 200, 1]
[:c, 300, 2]
```

## 11.3.3 控制块的执行

在调用代码清单 11.8 的 `total` 方法时，如果像下面那样在中途使用 `break`，`total` 方法的结果会变成什么样子呢？

```
n = total(1, 10) do |num|
  if num == 5
    break
  end
  num
end
p n      #=> ??
```

答案是 `nil`。在块中使用 `break`，程序会马上返回到调用块的地方，因此 `total` 方法中返回计算结果的处理等都会被忽略掉。但作为方法的结果，当我们希望返回某个值的时候，就可以像 `break 0` 这样指定 `break` 方法的参数，这样该值就会成为方法的返回值。

此外，如果在块中使用 `next`，程序就会中断当前处理，并继续执行下面的处理。使用 `next` 后，执行块的 `yield` 会返回，如果 `next` 没有指定任何参数则返回 `nil`，而如果像 `next 0` 这样指定了参数，那么该参数值就是返回值。

```
n = total(1, 10) do |num|
  if num % 2 != 0
    next 0
  end
  num
end
p n      #=> 30
```

最后，如果在块中使用 `redo`，程序就会返回到块的开头，并按照相同的块变量再次执行处理。这种情况下，块的处理结果不会返回给外部，因此需要十分小心 `redo` 的用法，注意不要使程序陷入死循环。

## 11.3.4 将块封装为对象

如前所述，在接收块的方法中执行块时，可以使用 `yield` 关键字。

而 Ruby 还能把块当作对象处理。把块当作对象处理后，就可以在接收块的方法之外的其他地方执行块，或者把块交给其他方法执行。

这种情况下需要用到 `Proc` 对象。`Proc` 对象是能让块作为对象在程序中使用的类。定义 `Proc` 对象的典型的方法是，调用 `Proc.new` 方法这个带块的方法。在调用 `Proc` 对象的 `call` 方法之前，块中定义的程序不会被执行。

在代码清单 11.11 的例子中，定义一个输出信息的 `Proc` 对象，并调用两次。这时，程序就会把 `call` 方法的参数作为块参数来执行块。

## 代码清单 11.11 proc1.rb

```
hello = Proc.new do |name|
  puts "Hello, #{name}."
end

hello.call("World")
hello.call("Ruby")
```

### 执行示例

```
> ruby proc1.rb
Hello, World.
Hello, Ruby.
```

把块从一个方法传给另一个方法时，首先会通过变量将块作为 `Proc` 对象接收，然后再传给另一个方法。在方法定义时，如果末尾的参数使用“& 参数名”的形式，Ruby 就会自动把调用方法时传进来的块封装为 `Proc` 对象。

下面，我们将代码清单 11.8 中块的接收方法加以改写，如下所示：

#### 代码清单 11.12 total2.rb

```
1: def total2(from, to, &block)
2:   result = 0          # 合计值
3:   from.upto(to) do |num| # 处理从from 到to 的值
4:     if block          # 如果有块的话
5:       result +=      # 累加经过块处理的值
6:       block.call(num)
7:     else              # 如果没有块的话
8:       result += num  # 直接累加
9:     end
10:    end
11:   return result      # 返回方法的结果
12: end
13:
14: p total2(1, 10)           # 从1 到10 的和 => 55
15: p total2(1, 10){|num| num ** 2 } # 从1 到10 的2 次幂的和 => 385
```

我们在首行的方法定义中定义了 `&block` 参数。像这样，在变量名前添加 `&` 的参数被称为 Proc 参数。如果在调用方法时没有传递块，`Proc` 参数的值就为 `nil`，因此通过这个值就可以判断出是否有块被传入方法中。另外，执行块的语句不是 `yield`，而是 `block.call(num)`，这一点与之前的例子也不一样。

在第 7 章中我们提到过方法可以有多个参数，而且定义参数的默认值等时都需要按照一定的顺序。而 `Proc` 参数则一定要在所有参数之后，也就是方法中最后一个参数。

将块封装为 `Proc` 对象后，我们就可以根据需要随时调用块。甚至还可以将其赋值给实例变量，让别的实例方法去任意调用。

此外，我们也能将 `Proc` 对象作为块传给其他方法处理。这时，只需在调用方法时，用“`&Proc` 对象”的形式定义参数就可以了。例如，向 `Array#each` 方法传递块时，可以像代码清单 11.13 那样定义。

#### 代码清单 11.13 call\_each.rb

```
def call_each(ary, &block)
  ary.each(&block)
end

call_each [1, 2, 3] do |item|
  p item
end
```

这样一来，我们就可以非常方便地把调用 `call_each` 方法时接收到的块，原封不动地传给 `ary.each` 方法。

#### 执行示例

```
> ruby call_each.rb
1
2
3
```

## 11.4 局部变量与块变量

块内部的命名空间与块外部是共享的。在块外部定义的局部变量，在块中也可以继续使用。而被作为块变量使用的变量，即使与块外部的变量同名，Ruby 也会认为它们是两个不同的变量。请看代码清单 11.14。

#### 代码清单 11.14 local\_and\_block.rb

```
x = 1          # 初始化x
y = 1          # 初始化y
ary = [1, 2, 3]

ary.each do |x| # 将x 作为块变量使用
  y = x        # 将x 赋值给y
end

p [x, y]      # 确认x 与y 的值
```

#### 执行示例

```
> ruby local_and_block.rb
[1, 3]
```

在 `ary.each` 方法的块中，`x` 的值被赋值给了局部变量 `y`。因此，`y` 保留了最后一次调用块时块变量 `x` 的值 3。而变量 `x` 的值在调用 `ary.each` 前后并没有发

生改变。

相反，在块内部定义的变量不能被外部访问。在刚才的例子中，如果把第 2 行的代码删掉，程序就会出错。

```
x = 1          # 初始化 x
#y = 1          # 初始化 y
ary = [1, 2, 3]

ary.each do |x| # 将 x 作为块变量使用
  y = x          # 将 x 赋值给 y
end

p [x, y]      # 引用 y 时会出错误 (NameError)
```

块中变量的作用域之所以这么设计，是为了通过与块外部共享局部变量，从而扩展变量的有效范围。在块内部给局部变量赋值的时候，要时刻注意它与块外部的同名变量的关系。大家一定要小心 Ruby 中的这个小陷阱。

块变量是只能在块内部使用的变量（块局部变量），它不能覆盖外部的局部变量，但 Ruby 为我们提供了定义块变量以外的块局部变量的语法。使用在块变量后使用 ; 加以区分的方式，来定义块局部变量。这里我们再稍微修改一下刚才的例子，如下所示。可以看出，块执行后 x 和 y 的值并没有变化。

#### 代码清单 11.15 local\_and\_block2.rb

```
x = y = z = 0      # 初始化x、y、z
ary = [1, 2, 3]
ary.each do |x; y| # 使用块变量x，块局部变量y
  y = x            # 代入块局部变量y
  z = x            # 代入不是块局部变量的变量z
  p [x, y, z]     # 确认块内的 x、y、z 的值
end
puts
p [x, y, z]      # 确认x、y、z 的值
```

#### 执行示例

```
> ruby local_and_block2.rb
[1, 1, 1]
[2, 2, 2]
[3, 3, 3]

[0, 0, 3]
```

## 第 3 部分 Ruby 的类

---

Ruby 中有各种各样的类。掌握类的使用方法后，我们就能更深地体会到 Ruby 有趣的地方。

# 第 12 章 数值类

到现在为止，数值（Numeric）类已经出现过好多次了。接下来我们就来详细讨论一下数值类的加减运算等基本操作、以及一些常用的功能。

- 数值类的构成

介绍包含 Fixnum、Float 等在内的数值类的构成。

- 数值的字面量（literal）

介绍在程序中描述数值的各种方法。

- 算术运算

介绍四则运算等基本的算术运算、以及数值运算时使用的模块 Math 的用法。

- 类型转换

介绍转换数值类型的方法，例如 Integer 与 Float 的互相转换。

- 位运算

介绍进行位运算的运算符。

- 随机数

介绍获取随机数时使用的一些相关功能。

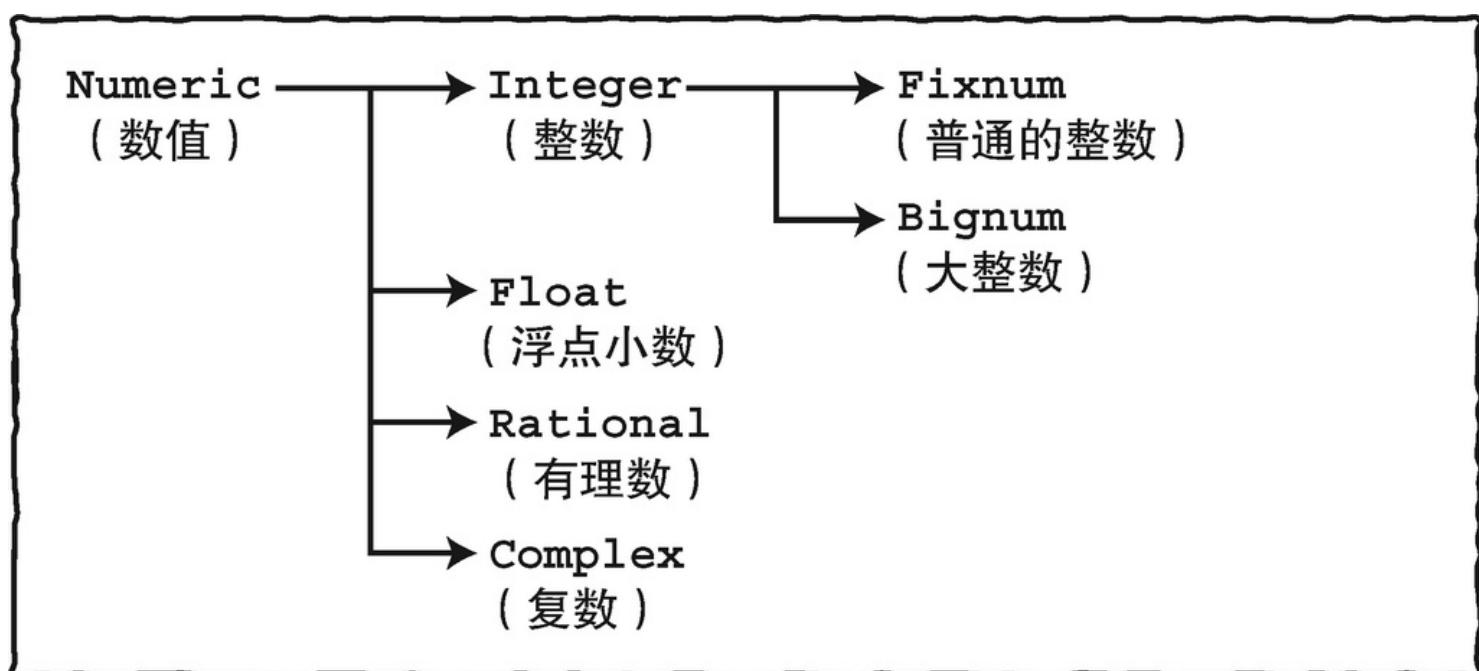
- 计数

介绍通过 Integer 指定循环次数的方法。

## 12.1 数值类的构成

在数值类中，有像 -1、0、1、10 这样的表示整数的 Integer 类，也有像 0.1、3.141592 这样的具有精度的、表示浮点小数的 Float 类。

这些数值类都被定义为了 Numeric 类的子类。另外，Integer 类又可以分为两种，一种是表示计算机硬件可以处理的数值的 Fixnum 类，另外一种是表示比 Fixnum 更大的数值的 Bignum 类。



程序中用到的整数一般都是 Fixnum 类范围内的整数。如果使用的整数超过了 Fixnum 的范围，Ruby 就会自动将其转换为 Bignum 类。因此，在写程序的时候，我们几乎可以忽略上述整数类的区别。下面是计算 2 的 10 次幂以及 2 的 1000 次幂的例子，`**` 是表示乘方的运算符。

### 执行示例

```
> irb --simple-prompt
>> n = 2 ** 10
=> 1024
>> n.class
=> Fixnum
>> m = 2 ** 1000
```

```
=> 1071508607186267320948425049060001810561404811705533607443750388370351051124936
1224931983788156958581275946729175531468251871452856923140435984577574698574803934
5677748242309854210746050623711418779541821530464749835819412673987675591655439460
77062914571196477686542167660429831652624386837205668069376
>> m.class
=> Bignum
```

Ruby 也可以处理有理数和复数。表示有理数用 `Rational` 类，表示复数用 `Complex` 类。

`Rational` 对象用“`Rational( 分子 , 分母 )`”的形式定义，例如，

```
[1] >>> Rational(2, 5)
```

上述这样的分数计算，可以用 `Rational` 对象改写成下面那样。我们还可以使用 `Rational#to_f` 方法将其转换为 `Float` 对象。

```
a = Rational(2, 5)
b = Rational(1, 3)
p a          #=> (2/5)
p b          #=> (1/3)
c = a + b
p c          #=> (11/15)
p c.to_f     #=> 0.7333333333333333
```

`Complex` 对象用“`Complex( 实数 , 虚数 )`”的形式定义。以下是计算复数  $2i$  的 2 次幂的例子：

```
c = Complex(0, 2) ** 2
p c          #=> (-4+0i)
```

## 12.2 数值的字面量

表 12.1 是表示数值对象的字面量的例子。

表 12.1 数值对象的字面量

字面量	作用（括号内为 10 进制的值）
123	表示10进制整数
0123	表示8进制整数（83）
0o123	表示8进制整数（83）
0d123	表示10进制整数（123）
0x123	表示16进制整数（291）
0b1111011	表示2进制整数（123）
123.45	浮点小数
1.23e4	浮点小数的指数表示法（ $1.23 \times 10^4$ 的 4 次幂=12300.0）
1.23e-4	浮点小数的指数表示法（ $1.23 \times 10^{-4}$ 的 -4 次幂=0.000123）

单纯的数字罗列表示 10 进制整数。以 `0b` 开头的数值表示 2 进制数，以 `0` 或者 `0o` 开头的数值表示 8 进制数，以 `0d` 开头的数值表示 10 进制数，以 `0x` 开头的数值表示 16 进制数。字面量中的 `_` 会被自动忽略。因此，在使用每 3 位数字间隔一下这样的数值表示方法时会十分方便。

```
p 1234567      #=> 1234567
p 1_234_567    #=> 1234567
p 0b11111111   #=> 255
p 01234567    #=> 342391
p 0x12345678  #=> 305419896
```

包含小数点的数值为浮点小数。我们还可以采用有效数字与指数配合的科学计数法来表示浮点小数。格式为“有效数字”+“英文字母 e (或者 E)”+“表示指数的整数”。

```
p 1.234        #=> 1.234
p 1.234e4      #=> 12340.0
p 1234e-4      #=> 0.0001234
```

## 12.3 算数运算

表 12.2 列出的是数值对象间进行基本的算术运算时用到的运算符。

表 12.2 算数运算的运算符

运算符	运算
+	加法运算
-	减法运算
*	乘法运算
/	除法运算
%	取余运算
**	乘方运算

`Integer` 对象与 `Float` 对象的运算结果为 `Float` 对象。`Integer` 对象之间、`Float` 对象之间的运算结果分别为 `Integer` 对象、`Float` 对象。

```
p 1 + 1      #=> 2
p 1 + 1.0    #=> 2.0
p 2 - 1      #=> 1
p 2 - 1.0    #=> 1.0
p 3 * 2      #=> 6
p 3 * 2.0    #=> 6.0
p 3 * -2.0   #=> -6,0
p 5 / 2      #=> 2
p 5 / 2.0    #=> 2.5
p 5 % 2      #=> 1
p 5 % 2.0    #=> 1.0
p 5 ** 2     #=> 25
p 5 ** 0.5   #=> 2.23606797749979
p 5 ** -2.0  #=> 0.04
p 5 ** -2    #=> 0.04
```

这里需要注意的是，指数为负整数的乘方返回的结果是表示有理数的 `Rational` 对象。

```
p 5 ** -2.0  #=> 0.04
p 5 ** -2    #=> (1/25)
```

## 除法

除了 / 和 % 以外，数值对象中还有一些与除法相关的方法。

- `x.div(y)`

返回  $x$  除以  $y$  后的商的整数。

```
p 5.div(2)      #=> 2
p 5.div(2.2)    #=> 2
p -5.div(2)     #=> -3
p -5.div(2.2)   #=> -3
```

- **`x.quo(y)`**

返回  $x$  除以  $y$  后的商，如果  $x$ 、 $y$  都是整数则返回 `Rational` 对象。

```
p 5.quo(2)      #=> (5/2)
p 5.quo(2.2)    #=> 2.2727272727272725
p -5.quo(2)     #=> (-5/2)
p -5.quo(2.2)   #=> -2.2727272727272725
```

- **`x.modulo(y)`**

与 `x % y` 等价。

- **`x.divmod(y)`**

将  $x$  除以  $y$  后的商和余数作为数组返回。商是将  $x/y$  的结果去掉小数点后的部分而得到的值。余数的符号与  $y$  的符号一致，余数的值为  $x \% y$  的结果。假设有运算式如下，

```
ans = X.divmod(y)
```

这时，下面的等式是成立的。

```
x == ans[0] * y + ans[1]
```

```
p 10.divmod(3.5)      #=> [2, 3.0]
p 10.divmod(-3.5)     #=> [-3, -0.5]
p -10.divmod(3.5)     #=> [-3, 0.5]
p -10.divmod(-3.5)    #=> [2, -3.0]
```

- **`x.remainder(y)`**

返回  $x$  除以  $y$  的余数，结果的符号与  $x$  的符号一致。

```
p 10.remainder(3.5)      #=> 3.0
p 10.remainder(-3.5)     #=> 3.0
p -10.remainder(3.5)     #=> -3.0
p -10.remainder(-3.5)    #=> -3.0
```

另外，除数为 0 时，`Integer` 类会返回错误，而 `Float` 类则会返回 `Infinity`（无限大）或者 `NaN`（Not a Number）。如果再用这两个值进行运算，那么结果只会返回 `Infinity` 或者 `NaN`。程序把输入的数据直接用于运算的时候，除数有可能会为 0，我们应当注意避免这样的情况发生。

```
p 1 / 0      #=> 错误 (ZeroDivisionError)
p 1 / 0.0    #=> Infinity
p 0 / 0.0    #=> NaN
p 1.divmod(0)  #=> 错误 (ZeroDivisionError)
p 1.divmod(0.0) #=> 错误 (FloatDomainError)
```

## 12.4 Math 模块

`Math` 模块提供了三角函数、对数函数等常用的函数运算的方法。该模块中定义了模块函数和常量，例如，求平方根时，可以采用下述方法。

```
p Math.sqrt(2)    #=> 1.4142135623730951
```

表 12.3 为 `Math` 模块定义的方法。

表 12.3 `Math` 模块定义的方法

方法名	作用
<code>acos(x)</code>	反余弦函数

<code>acosh(x)</code>	反双曲余弦函数
<code>asin(x)</code>	反正弦函数
<code>asinh(x)</code>	反双曲正弦函数
<code>atan(x)</code>	反正切函数
<code>atan2(x, y)</code>	表示 4 个象限的反正切函数
<code>atanh(x)</code>	反双曲正切函数
<code>cbrt(x)</code>	立方根
<code>cos(x)</code>	余弦函数
<code>cosh(x)</code>	双曲余弦函数
<code>erf(x)</code>	误差函数
<code>erfc(x)</code>	余补误差函数
<code>exp(x)</code>	指数函数
<code>frexp(x)</code>	把一个浮点数分解为尾数和指数
<code>gamma(x)</code>	伽玛函数
<code>hypot(x, y)</code>	计算三角形的斜边长度
<code>ldexp(x, y)</code>	返回 $x$ 乘以 $2$ 的 $y$ 次幂的值
<code>lgamma(x)</code>	伽马函数的自然对数
<code>log(x)</code>	底数为 $e$ 的对数（自然对数）
<code>log10(x)</code>	底数为 $10$ 的对数（常用对数）
<code>log2(x)</code>	底数为 $2$ 的对数
<code>sin(x)</code>	正弦函数

<code>sinh(x)</code>	双曲正弦函数
<code>sqrt(x)</code>	平方根
<code>tan(x)</code>	正切函数
<code>tanh(x)</code>	双曲正切函数

另外，`Math` 模块还定义了表 12.4 的常量。

表 12.4 `Math` 模块定义的常量

常量名	作用
<code>PI</code>	圆周率 (3.141592653589793)
<code>E</code>	自然对数的底数 (2.718281828459045)

## 12.5 数值类型转换

将 `Integer` 对象转换为 `Float` 对象时，可以使用 `to_f` 方法。相反，使用 `to_i` 方法则可以将 `Float` 对象转换为 `Integer` 对象（`Integer#to_i` 方法和 `Float#to_f` 方法返回与接收者一样的值）。另外，也可以把字符串转换为数值。

```
p 10.to_f      #=> 10.0
p 10.8.to_i    #=> 10
p -10.8.to_i   #=> -10
p "123".to_i   #=> 123
p "12.3".to_f  #=> 12.3
```

`Float#to_i` 方法返回的结果会把小数点以后的值去掉。我们用 `round` 方法对小数进行四舍五入的处理。

```
p 1.2.round    #=> 1
p 1.8.round    #=> 2
p -1.2.round   #=> -1
p -1.8.round   #=> -2
```

返回比接收者大的最小整数用 `ceil` 方法，返回比接收者小的最大整数用 `floor` 方法。

```
p 1.5.ceil     #=> 2
p -1.5.ceil    #=> -1
p 1.5.floor    #=> 1
p -1.5.floor   #=> -2
```

我们还可以将数值转换为 `Rational` 对象和 `Complex` 对象，分别使用 `to_r` 和 `to_c` 方法，如下所示。

```
p 1.5.to_r      #=> (3/2)
p 1.5.to_c      #=> (1.5+0i)
```

## 12.6 位运算

`Integer` 类可以进行表 12.5 所示的位运算。

表 12.5 `Integer` 类的位运算符

运算符	运算
<code>~</code>	按位取反（一元运算符）

&	按位与
	按位或
^	按位异或 ( $(a \& \sim b)   (\sim a \& b)$ )
>>	位右移
<<	位左移

```

def pb(i)
    # 使用printf 的%b 格式
    # 将整数的末尾8 位用2 进制表示
    printf("%08b\n", i & 0b11111111)
end

b = 0b11110000
pb(b)           #=> 11110000
pb(~b)          #=> 00001111
pb(b & 0b00010001) #=> 00010000
pb(b | 0b00010001) #=> 11110001
pb(b ^ 0b00010001) #=> 11100001
pb(b >> 3)      #=> 00011110
pb(b << 3)      #=> 10000000

```

## 专栏

### 位与字节

在计算机的世界中，我们经常会接触到“位”与“字节”，接下来我们就来介绍一下它们所代表的意义。

- 位 (bit)

位是计算机中最小的数据单位，表示 ON 或 OFF，或者 0 或 1。据说原本是“Binary digit”的简称。

- 位与 2 进制

虽然位中只包含两种信息，但将位的信息两两组合的话，就可以表示 00、01、10、11 这四种信息。同样，3 位一组的话可以表示八种信息，4 位一组的话可以表示十六种信息。随着位的数量的增加，可以表示的信息的数量也会成倍的增加。

像这样，只用 0 和 1 的计数方式称为 2 进制。我们一般使用的计数方式是 10 进制，这种计数方式总共可表示从 0 到 9 十个数。下面是 2 进制与 10 进制的对照表。

表 10 进制、2 进制、8 进制、16 进制对照表

10 进制	2 进制	8 进制	16 进制
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9

10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11

- **8 进制与 16 进制**

计算机处理的信息是用 2 进制表示的。但是如果全部都只用 0、1 来表示的话，位数就会变得非常大，不便于人们理解。因此就可以使用 8 进制和 16 进制来解决这个问题。8 进制使用 0 到 7 共 8 个数，用 1 位数表示 3 个位（bit）。16 进制使用 0 到 15 共 16 个数，用 1 位数表示 4 个位（bit）。由于一般我们使用的数字只有 10 个，因此在 16 进制中，10 到 15 之间的数字就用英文字母 A 到 F 表示。

- **字节（byte）**

计算机在表示数的时候，会把一定数量的位（bit）的组合——字节作为计数单位。一个字节有多少位并没有共通的标准。在以前，根据制造商和机种的不同，1 个字节包含的位的数量也不一样，但现在 1 个字节有 8 位已经是业界的常识了。1 个字节可以表示的 10 进制数是从 0 到 255，8 进制数是从 000 到 377，16 进制数是从 00 到 FF。由于 2 个 16 进制位刚好等于 8 位（1 个字节），因此用 16 进制来表示以字节为单位的数据会非常方便。

## 12.7 随机数

有时候随机性可能会帮助我们解决一些问题。随机性一般有以下特质。

- 没有规则和法则依据
- 一定范围内的数会均等地出现

拿掷骰子为例，我们不能预测下一个投出的是哪一面，但骰子各个面投出的几率都是一样的。我们把这样的情况称为随机，随机得到的数值称为随机数。在掷骰子或者洗扑克牌那样需要偶然性的情况下，或者像加密后的密码那样希望得到一些难以被预测的数据时，一般都会用到随机数。

我们可以用 `Random.rand` 方法得到随机数。不指定参数时，`Random.rand` 方法返回比 1 小的浮点小数。参数为正整数时，返回 0 到该正整数之间的数值。

```
p Random.rand      => 0.13520495197709
p Random.rand(100) => 31
p Random.rand(100) => 84
```

程序不能生成真正的随机数，只能通过某种算法生成看起来像随机数的值，这样的随机数称为模拟随机数。生成模拟随机数需要以某个值为基础，这个值称为随机数的种子。模拟随机数终究只是通过计算得到的数值，只要随机数的种子一样，那么得到值就有可能重复出现。使用 `Random.new` 方法初始化随机数生成器，然后再使用 `Random#rand` 方法，就可以对 `Random` 对象指定随机数种子，从而生成随机数。

```
r1 = Random.new(1)    # 初始化随机数组
p [r1.rand, r1.rand]
=> [0.417022004702574, 0.7203244934421581]

r2 = Random.new(1)    # 再次初始化随机数组
p [r2.rand, r2.rand]
=> [0.417022003702574, 0.7203244934421581]
```

`Random.new` 方法不指定参数的情况下，则会用随机生成的随机数种子初始化 `Random` 对象，因此每次得到的随机数组也会不一样。

```
r1 = Random.new
p [r1.rand, r1.rand]
=> [0.49452535392946817, 0.34141702823098863]

r2 = Random.new
p [r2.rand, r2.rand]
=> [0.9464262066747281, 0.01911968591048996]
```

在信息安全领域中，“优质的随机”是一个重要的课题。生成用于加密 key 的随机数时，不能重复出现是非常重要的，因此就需要我们慎重地选择难以被预测的随机种子。在一些特殊的情况下可能会需要初始化 `Random` 对象，而一般情况下直接用最开始介绍的 `Random.rand` 方法就足够了。

## 12.8 计数

除了数值计算外，`Integer` 类还能计算处理的次数、数组的元素个数等。接下来介绍的方法就是按照数值指定的次数执行循环处理的迭代器。

- `n.times{|i| ...}`

循环 `n` 次，从 0 到 `n-1` 的值会被依次赋值给块变量。

```
ary = []
10.times do |i|
  ary << i
end
p ary    #=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- `from.upto(to){|i| ...}`

从 `from` 开始循环对 `i` 进行加 1 处理，直到 `i` 等于 `to`。`from` 比 `to` 大时不会执行循环处理。

```
ary = []
2.upto(10) do |i|
  ary << i
end
p ary    #=> [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- `from.downto(to){...}`

从 `from` 开始循环对 `i` 进行减 1 处理，直到 `i` 等于 `to`。`from` 比 `to` 小时不会执行循环处理。

```
ary = []
10.downto(2) do |i|
  ary << i
end
p ary    #=> [10, 9, 8, 7, 6, 5, 4, 3, 2]
```

- `from.step(to, step){...}`

从 `from` 开始循环对 `i` 进行加 `step` 处理，直到 `i` 等于 `to`。`step` 为正数时，`from` 比 `to` 大时不会执行循环处理。`step` 为负数时，`from` 比 `to` 小时不会执行循环处理。

```
ary = []
2.step(10, 3) do |i|
  ary << i
end
p ary    #=> [2, 5, 8]

ary = []
10.step(2, -3) do |i|
  ary << i
end
p ary    #=> [10, 7, 4]
```

如果不对 `times`、`upto`、`downto`、`step` 的各方法指定块，则会返回 `Enumerator` 对象。这样，之前通过 `step` 方法的块获取的一连串数值，就同样也可以通过 `Enumerator#collect` 方法获取。关于 `Enumerator` 对象，我们会在第 14 章中介绍。

```
ary = 2.step(10).collect{|i| i * 2}
p ary    #=> [4, 6, 8, 10, 12, 14, 16, 18, 20]
```

## 12.9 近似值误差

处理浮点小时很容易因误差产生问题。这里我们来看看具体的例子，执行下面的程序后会产生意想不到的结果。

```
a = 0.1 + 0.2
b = 0.3
p [a, b]    #=> [0.3, 0.3]
p a == b    #=> false
```

虽然我们期待  $0.1 + 0.2$  与  $0.3$  的比较结果为 `true`，但实际结果却是 `false`。为什么会这样呢？

在 10 进制中，就像  $1/10$ 、 $1/100$ 、 $1/1000$ ……这样，我们会用 10 取幂后的倒数来表示数值。而另一方面，`Float` 类的浮点小数则是用 2 取幂后的倒数来表示，如  $1/2$ 、 $1/4$ 、 $1/8$ ……。因此，在处理  $1/5$ 、 $1/3$  这种用 2 进制无法正确表示的数值时，结果就会产生误差。而如果要用 2 进制的和来表示这类数值的话，计算机就必须在适当的位置截断计算结果，这样就产生了近似值误差。

如果可以把小数转换为两个整数相除的形式，那么通过使用 `Rational` 类进行运算，就可以避免近似值误差。

```
a = Rational(1, 10) + Rational(2, 10)
b = Rational(3, 10)
p [a, b]      #=> [(3/10), (3/10)]
p a == b
```

另外，Ruby 还提供了 `bigdecimal` 库，可以有效处理拥有更多小数位的 10 进制数。

## 专栏

### Comparable 模块

Ruby 的比较运算符（`==`、`<=` 等）实际上都是方法。`Comparable` 模块里封装了比较运算符，将其 Mix-in 到类后，就可以实现实例间互相比较的方法（下表）。Comparable 在英语中就是“可以比较”的意思。

表 Comparable 模块封装的方法

	<code>&lt;&gt;</code>	<code>==</code>	<code>&gt;=</code>	<code>&gt;</code>	<code>between?</code>
--	-----------------------	-----------------	--------------------	-------------------	-----------------------

`Comparable` 模块中的各运算符都会使用 `<=` 运算符的结果。`<=` 运算符如果能像下表那样定义的话，上表中的各个方法就都可以使用。

表 表 `a <= b` 的结果

<code>a &lt;&gt; 时</code>	-1(比0小)
<code>a == b 时</code>	0
<code>a &gt; b 时</code>	1(比0大)

下面的 `Vector` 类表示拥有 `x` 和 `y` 两个坐标的向量。为了比较向量间的坐标，这里定义了 `<=` 运算符。然后，通过包含（include） `Comparable` 模块，就可以实现上表中的比较方法。

```
class Vector
  include Comparable
  attr_accessor :x, :y

  def initialize(x, y)
    @x, @y = x, y
  end

  def scalar
    Math.sqrt(x ** 2 + y ** 2)
  end

  def <= (other)
    scalar <= other.scalar
  end
end

v1 = Vector.new(2, 6)
v2 = Vector.new(4, -4)
p v1 <= v2      #=> 1
p v1 < v2       #=> false
p v1 > v2       #=> true
```

在本书介绍过的类中，`Numeric`、`String`、`Time` 都包含了 `Comparable` 模块。

## 练习题

1. 表示温度的单位有摄氏、华氏两种。请定义将摄氏转换为华氏的方法 `cels2fahr`。摄氏与华氏的转换公式如下：

$$\text{华氏} = \text{摄氏} \times \frac{9}{5} + 32$$

2. 与 1 相反，请定义将华氏转换为摄氏的方法 `fahr2cels`。然后从 1 摄氏度到 100 摄氏度，请按照每隔 1 摄氏度输出一次的方式，输出对应的华氏温度。

3. 请定义返回掷骰子（1 到 6 随机整数）的结果的 `dice` 方法。

4. 请定义合计掷 10 次骰子的结果的 `dice10` 方法。

5. 请定义调查整数 `num` 是否为素数的 `prime?(num)` 方法。素数的定义为除了 1 和自己外不能被整除的数。个位整数中，2、3、5、7 为素数。

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

# 第 13 章 数组类

本章将详细介绍数组（Array）类。

- 数组的创建方法

介绍如何新创建数组、以及如何通过其他对象创建数组。

- 索引的使用方法

数组的基本用法是利用索引（下标）访问数组内各元素。在这一部分中我们会介绍索引的相关用法。

- 作为集合的数组与作为列的数组

Ruby 中可以把数组作为集合操作，也可以把数组作为列操作。在这一部分中我们会介绍这两种数组的用法。

- 主要的数组方法

通过数组方法，不仅可以将执行结果以新对象的形式返回，还可以对元素进行互换、删除等操作，进而变更已存在的对象。在这一部分中我们会详细介绍如何使用这些数组方法。

- 数组与迭代器

迭代器经常被用来逐个处理数组中的元素。在这一部分中我们会介绍迭代器的基本用法、以及数组的迭代方法。

- 处理数组中的元素

除了迭代器之外，使用其他方法也可以处理数组内的各元素。在这一部分中我们会介绍几种具有代表性的方法。

- 同时访问多个数组

任何对象都可以作为数组的元素。在这一部分中我们会介绍一些数组使用过程中需要注意的事项。

## 13.1 复习数组

在第 2 章中我们已经介绍过数组了，现在我们先来复习一下。

数组是带索引的对象的集合。

数组有以下特征：

- 可以从数组中获取某个索引的元素（对象）

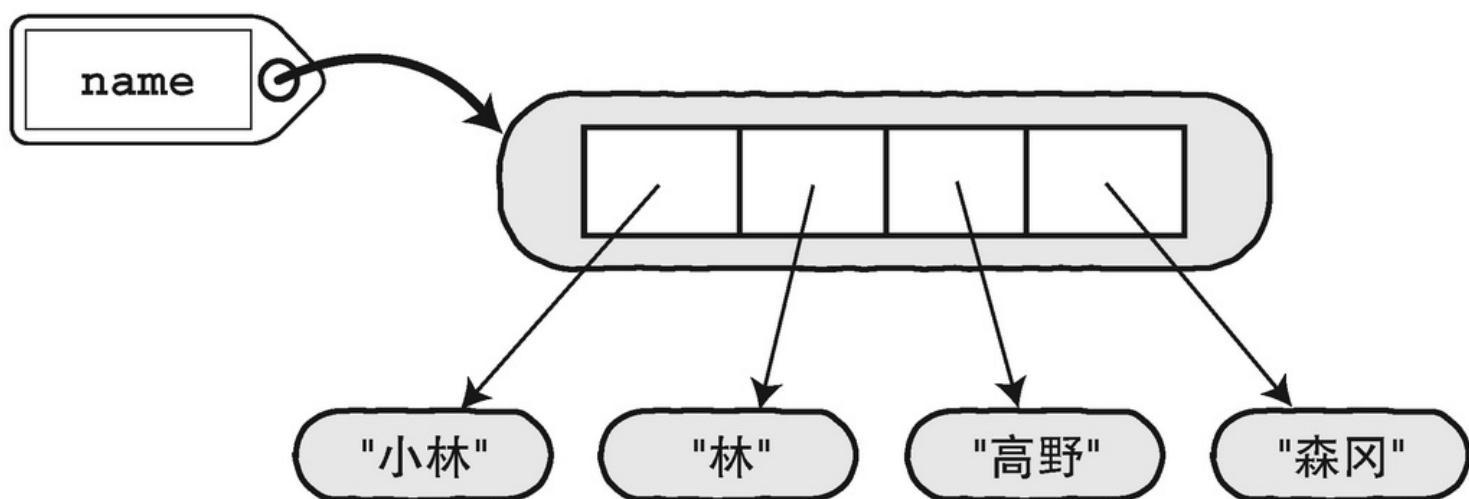
例：`print name[2]`

- 可以将任意的值（对象）保存到数组的某个索引的元素中

例：`name[0] = "野尻"`

- 使用迭代器可以逐个取出数组中的元素

例：`names.each{|name| puts name}`



## 13.2 数组的创建方法

在第 2 章中，我们介绍了使用 `[]` 来创建数组的方法。

```
nums = [1, 2, 3, 4, 5]
strs = ["a", "b", "c", "d"]
```

除此以外还有其他的创建方法，接下来就简要地介绍一下。

### 13.2.1 使用 `Array.new`

创建类的实例时使用的 `new` 方法，创建数组时也同样可以使用。

```
a = Array.new
p a          #=> []
a = Array.new(5)
p a          #=> [nil, nil, nil, nil, nil]
a = Array.new(5, 0)
p a          #=> [0, 0, 0, 0, 0]
```

`Array` 类的情况下，若 `new` 方法没有参数，则会创建元素个数为 0 的数组；若 `new` 方法只有 1 个参数，则会创建元素个数为该参数个数，且各元素初始值都为 `nil` 的数组；若 `new` 方法有两个参数，则第 1 个参数代表元素的个数，第 2 个参数代表元素值的初始值。

当希望创建元素值相同的数组时，建议使用这个方法。

### 13.2.2 使用 `%w` 与 `%d`

创建不包含空白的字符串数组时，可以使用 `%w`。

```
lang = %w(Ruby Perl Python Scheme Pike REBOL)
p lang #=> ["Ruby", "Perl", "Python", "Scheme", "Pike",
#           "REBOL"]
```

虽然给人的感觉只是节省了书写 `" "` 和 `,` 的时间，但是如果能掌握这种字符串数组的创建方法，就会使程序更加简洁。此外，Ruby2.0 还提供了创建符号（Symbol）数组的 `%i`。

```
lang = %i(Ruby Perl Python Scheme Pike REBOL)
p lang  #=> [:Ruby, :Perl, :Python, :Scheme, :Pike, :REBOL]
```

在本例中，创建数组时使用了 `()` 将数组元素括了起来，但实际上还可以使用如 `<>`、`|||`、`!!`、`@@`、`AA` 这样的任意字符。

虽然 Ruby 允许我们使用任意字符，但若用一些不常用的字符来创建数组的话，可能就会使程序不便于阅读。在选择表示字符串数组元素的字符时，还要注意该字符不能在要创建的字符串中出现，因此建议使用 `()`、`<>`、`|||`。

### 13.2.3 使用 `to_a` 方法

到现在为止，我们已经介绍了三种传统的创建数组的方法，下面我们就来看看如何将其他对象转换为数组。

很多类都定义了 `to_a` 方法，该方法能把该类的对象转换为数组。

```
color_table = {black: "#000000", white: "#FFFFFF"}
p color_table.to_a  #=> [[:black, "#000000"],
#                         [:white, "#FFFFFF"]]
```

对散列对象使用 `to_a` 方法，结果就会得到相应的数组的数组。具体来说就是，将散列中的各键、值作为一个数组，然后再把这样的数组放到一个大数组中。

### 13.2.4 使用字符串的 `split` 方法

我们再来介绍一个将对象转换为数组的方法。对用逗号或者空白间隔的字符串使用 `split` 方法，也可以创建数组。

```
column = "2013/05/30 22:33 foo.html proxy.example.jp".split()
p column
#=> ["2013/05/30", "22:33", "foo.html", "proxy.example.jp"]
```

关于 `split` 方法我们会在第 14.6 节中详细说明。

## 13.3 索引的使用方法

在了解了如何创建数组之后，下面我们就来看看如何操作数组。

首先，我们将介绍如何用索引操作数组。用索引操作数组的方法是操作数组的基础方法。这部分内容也许会和第 2 章的内容有些重复，不过这里我们会重点介绍一些新的内容，以使大家能够对索引有一个系统的了解。

### 13.3.1 获取元素

对数组指定索引值，就可以获取相应的元素。我们可以逐个获取元素，也可以一次获取多个元素。

通过 `[]` 指定索引，获取元素。`[]` 有以下 3 种用法：

(a) `a[n]`

(b) `a[n..m]` 或者 `a[n...m]`

(c) `a[n, len]`

用法 (a) 是我们在第 2 章中使用过的获取索引值为  $n$  的元素的方法。例如，通过 `alpha[0]` 获取数组 `alpha` 的首个元素。这里要注意，数组的索引值是从 0 开始的（图 13.1）。

```
alpha = ["a", "b", "c", "d", "e"]
p alpha[1]      #=> "b"
```

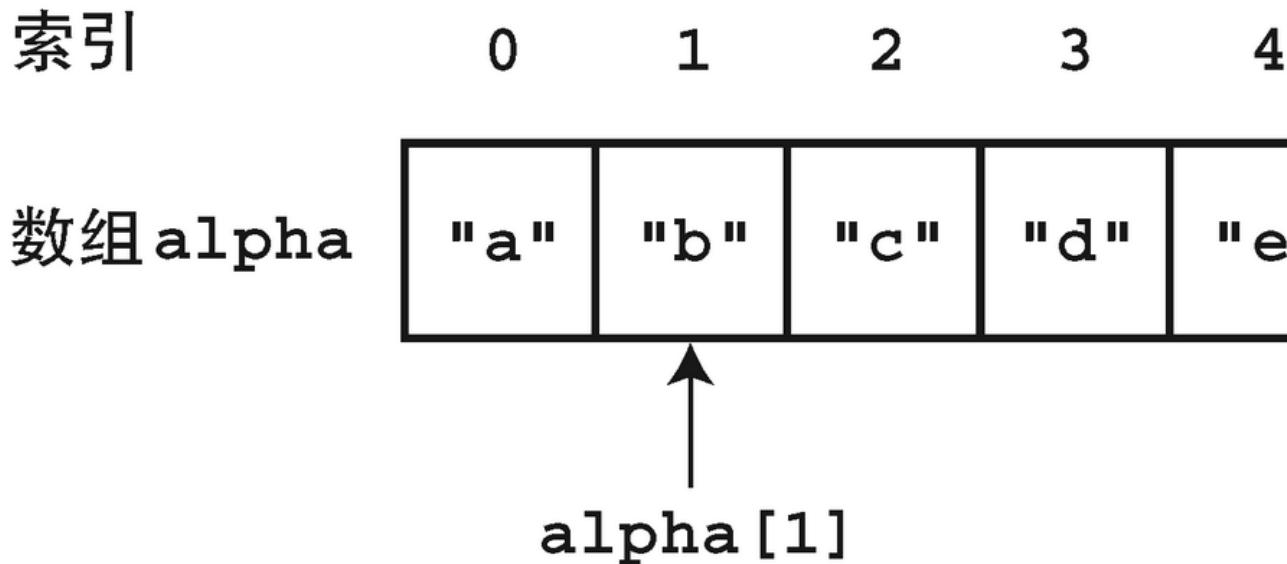


图 13.1 数组与索引的关系

索引值为负数时，不是从数组的开头，而是从数组的末尾开始获取元素（图 13.2）。如果指定的索引值大于元素个数，则返回 `nil`。

```
alpha = ["a", "b", "c", "d", "e"]
p alpha[-1]      #=> "e"
p alpha[-2]      #=> "d"
```

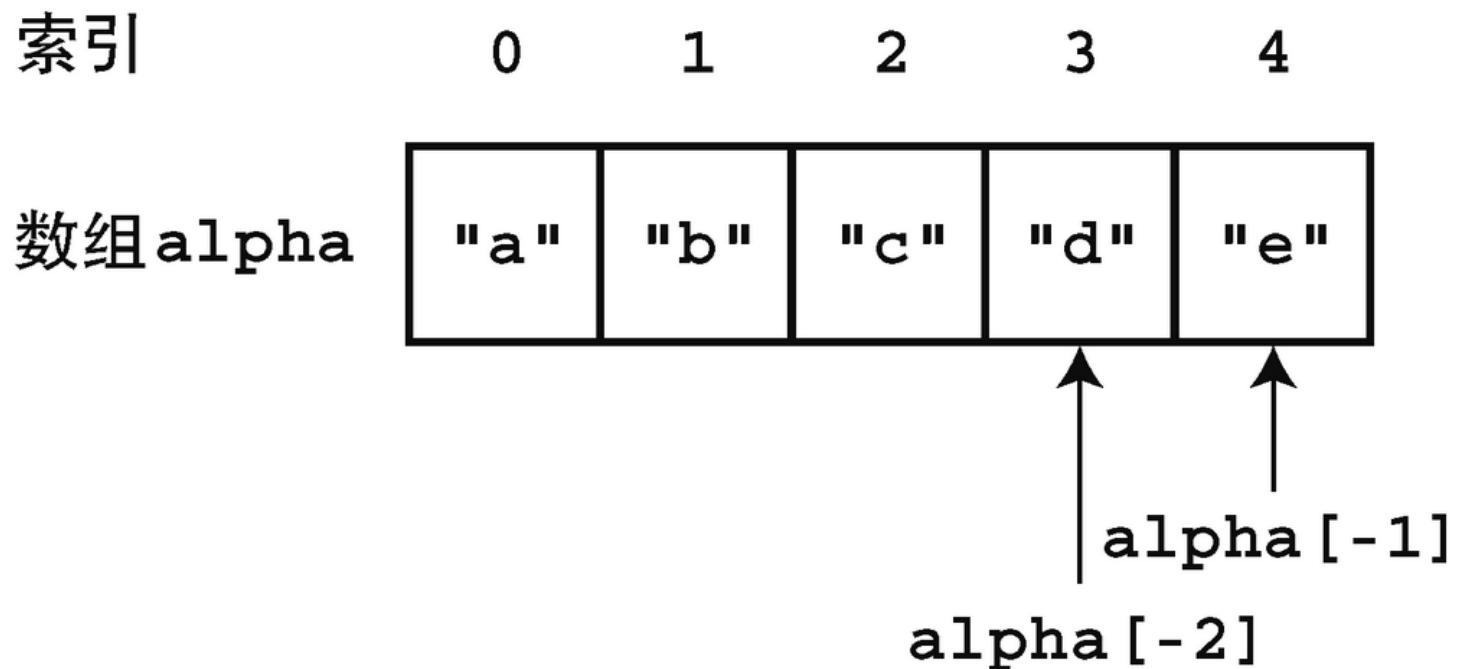


图 13.2 索引值为负数

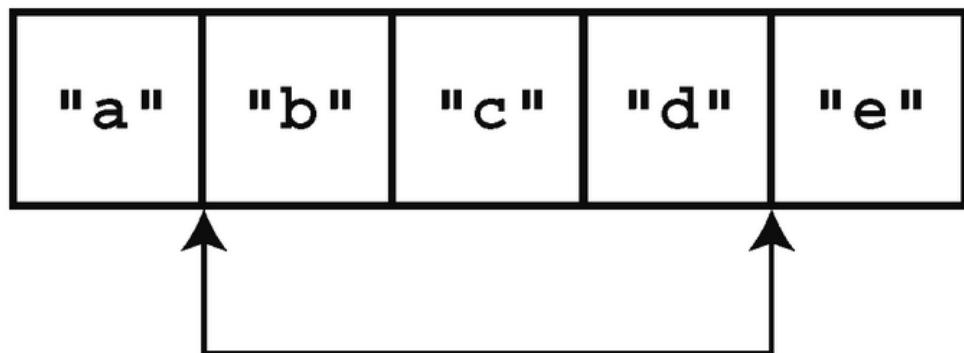
用法 (b) 的  $a[n..m]$  表示获取从  $a[n]$  到  $a[m]$  的元素，然后用它们创建新数组并返回（图 13.3）。 $a[n..m]$  表示获取从  $a[n]$  到  $a[m-1]$  的元素，并用它们创建新数组返回。虽然下面的例子中只讨论  $[n..m]$  的形式，但能用  $[n..m]$  的地方同样能用  $[n...m]$ 。

```
alpha = ["a", "b", "c", "d", "e"]
p alpha[1..3]      #=> ["b", "c", "d"]
```

索引

0      1      2      3      4

数组 alpha



`alpha[1..3]`

图 13.3 指定索引范围

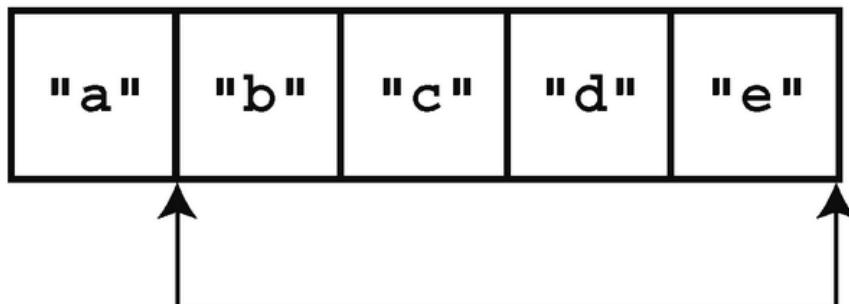
如果 `m` 的值比数组长度大，则返回的结果与指定数组最后一个元素时是一样的（图 13.4）。

```
alpha = ["a", "b", "c", "d", "e"]
p alpha[1..7]      #=> ["b", "c", "d", "e"]
```

索引

0      1      2      3      4

数组 alpha



`alpha[1..7]`

图 13.4 索引值比数组长度大时

用法 `(c) [n, len]` 表示从 `a[n]` 开始，获取之后的 `len` 个元素，用它们创建新数组并返回（图 13.5）。

```

alpha = ["a", "b", "c", "d", "e"]
p alpha[2, 2]      #=> ["c", "d"]
p alpha[2, 3]      #=> ["c", "d", "e"]

```

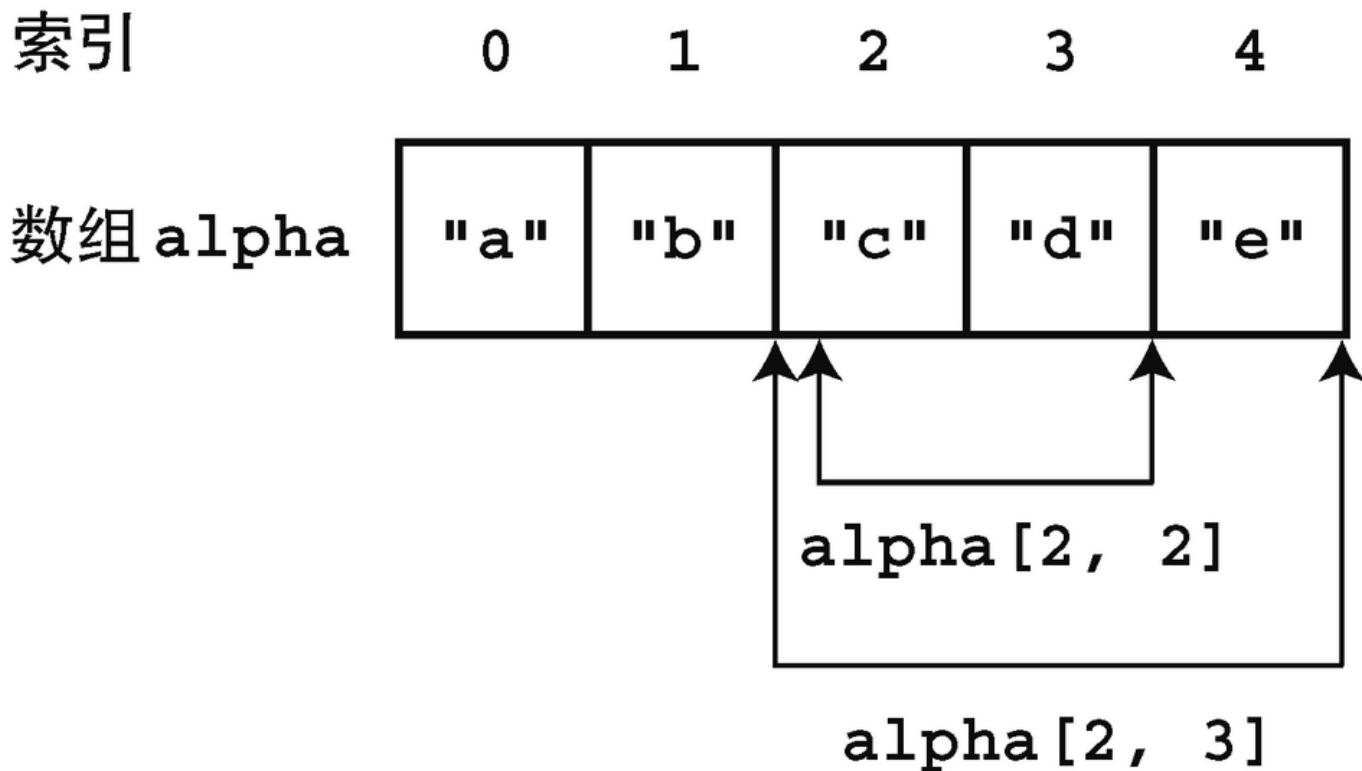


图 13.5 从某个元素开始，获取多个元素

另外，我们还可以用普通的方法代替`[]`。

- `a.at(n)` .....与 `a[n]` 等价
- `a.slice(n)` .....与 `a[n]` 等价
- `a.slice(n..m)` .....与 `a[n..m]` 等价
- `a.slice(n, len)` .....与 `a[n, len]` 等价

不过一般情况下我们很少会使用上述方法。

### 13.3.2 元素赋值

使用`[]`、`at`、`slice`方法除了可以获取元素外，还可以对元素赋值。

- `a[n] = item`

这是将`a[n]`的元素值变更为`item`。在下面的例子中，我们来尝试把`B`赋值给第2个元素，把`E`赋值给第5个元素（图13.6）。

```

alpha = ["a", "b", "c", "d", "e", "f"]
alpha[1] = "B"
alpha[4] = "E"
p alpha    #=> ["a", "B", "c", "d", "E", "f"]

```

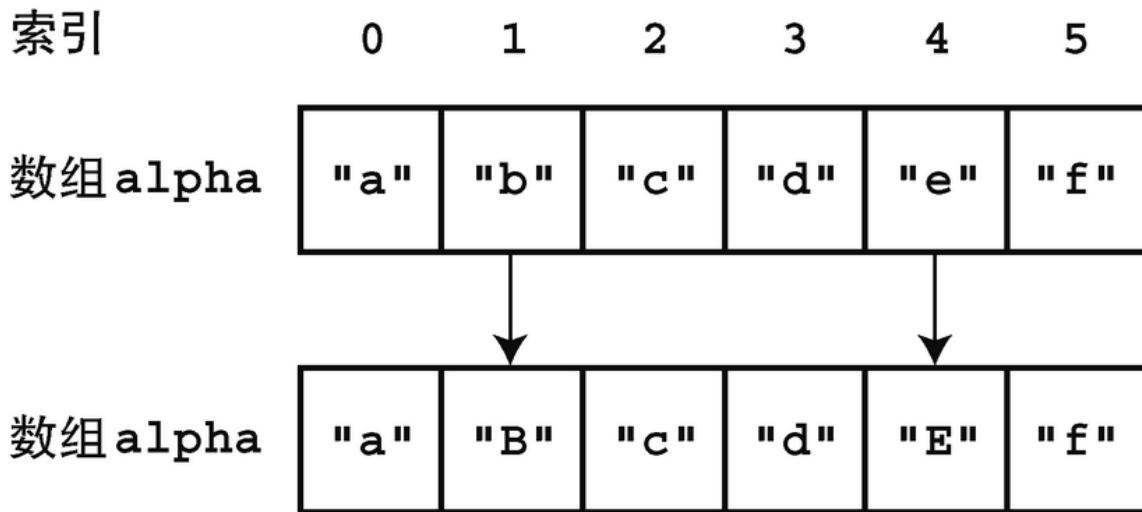


图 13.6 元素赋值

上面的例子介绍的是对一个元素赋值，实际上 Ruby 还可以一次对多个元素赋值。指定多个元素的方法与在 13.3.1 节中介绍的获取多个元素的方法是一样的。

在下面的例子中，我们来尝试对数组的第 3 个元素到第 5 个元素赋值。图 13.7 表示的是使用  $[n..m]$  的形式进行赋值时的情况。下面是使用  $[n, len]$  的形式进行赋值的例子。

```

alpha = ["a", "b", "c", "d", "e", "f"]
alpha[2, 3] = ["C", "D", "E"]
p alpha    #=> ["a", "b", "C", "D", "E", "f"]

```

```

alpha = ["a", "b", "c", "d", "e", "f"]
alpha[2..4] = ["C", "D", "E"]
p alpha    #=> ["a", "b", "C", "D", "E", "f"]

```

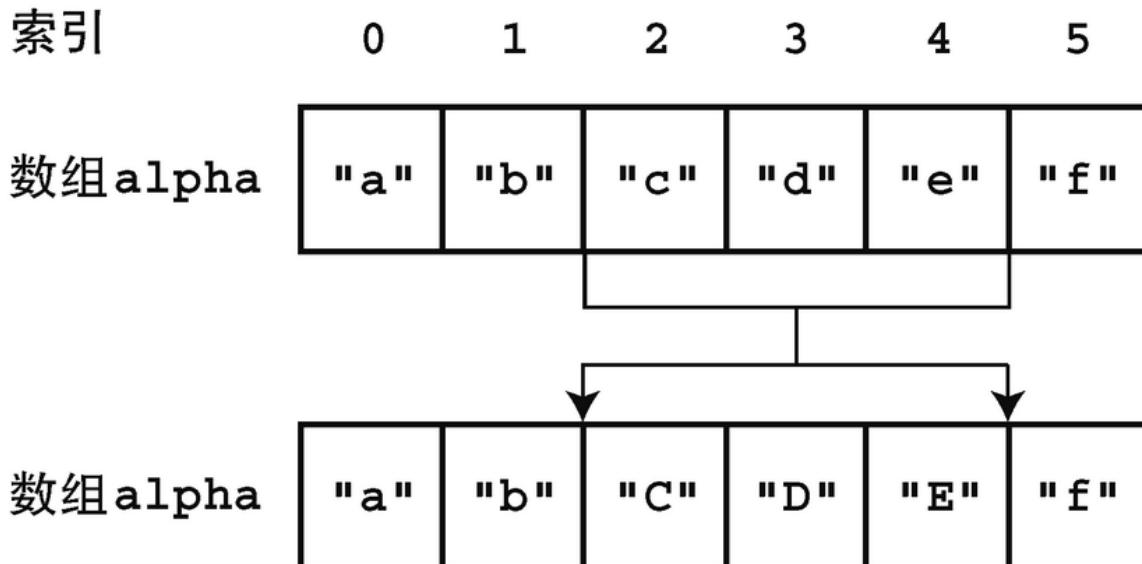


图 13.7 对多个元素赋值

### 13.3.3 插入元素

我们还可以在保持当前元素不变的情况下，对数组插入新的元素。

插入元素其实也可以被认为是对 0 个元素进行赋值。因此，指定  $[n, 0]$  后，就会在索引值为  $n$  的元素前插入新元素（图 13.8）。

```
alpha = ["a", "b", "c", "d", "e", "f"]
alpha[2, 0] = ["X", "Y"]
p alpha #=> ["a", "b", "X", "Y", "c", "d", "e", "f"]
```

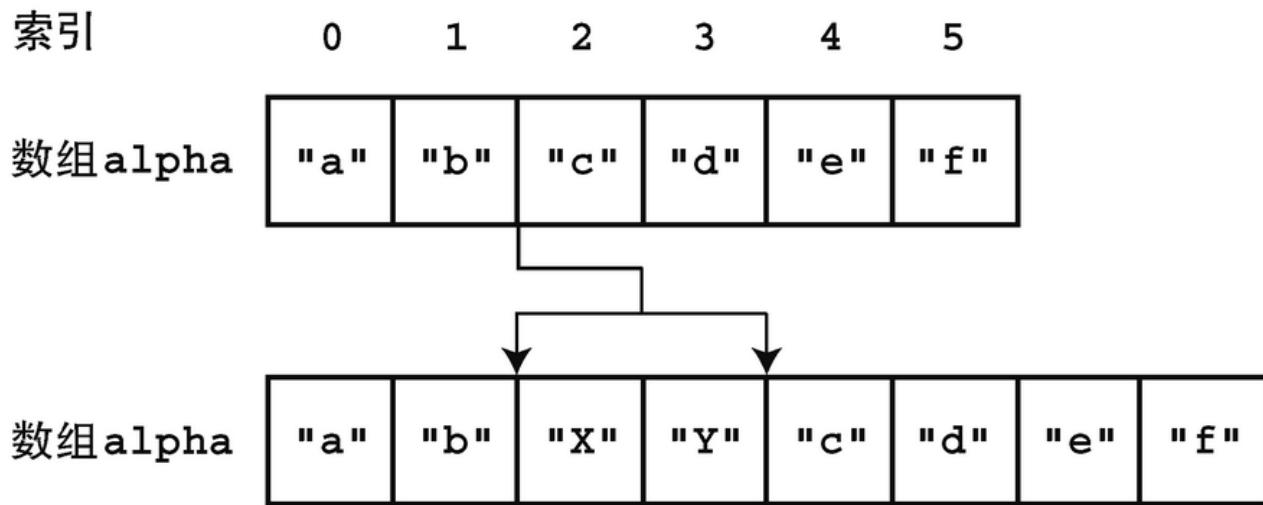


图 13.8 插入元素

### 13.3.4 通过多个索引创建数组

通过使用我们在 13.3.1 节中介绍的方法，虽然可以获取多个连续的元素，但是却不能获取分散的元素。而使用 `values_at` 方法，就可以利用多个索引来分散获取多个元素，并用它们创建新数组。

- `a.values_at(n1, n2, ...)`

用这个方法，我们就可以每隔一个元素获取一次（图 13.9）。

```

alpha = %w(a b c d e f)
p alpha.values_at(1,3,5) #=> ["b", "d", "f"]

```

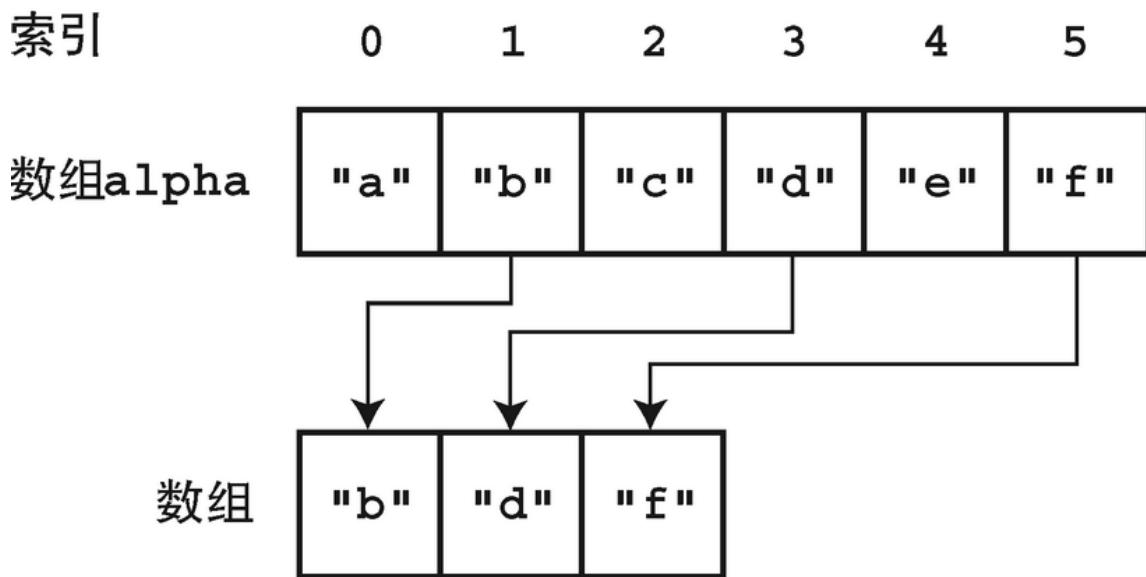


图 13.9 获取分散的元素并创建数组

#### 13.4 作为集合的数组

到目前为止的数组操作都是通过索引完成的。也就是说，从哪里获取元素、给哪个元素赋值、在哪里插入元素这些操作都是直接指定数组索引后进行的。

的确，数组、`Array` 类本来就是带有索引的对象，使用索引也是理所当然。不过有些时候我们会希望不通过索引而直接操作数组元素。

例如，我们可以把数组当成集合，这样一来，`Array` 类中的各元素就变了集合里的元素。

然而，由于集合没有顺序的概念，因此 `["a", "b", "c"]`、`["b", "c", "a"]`、`["c", "b", "a"]` 就都可以被认为是同一个集合。

这样操作数组时，如果我们还关心“这个对象是数组的第几个元素”之类的问题，就可能会造成混乱。这是因为，索引操作实际上只是数组封装的一个功能而已。

接下来，我们就来看看如何把数组当作集合使用。而在下一节中，我们会再讨论把数组当作列使用时的方法。

集合的基本运算分为交集和并集。

- 取出同时属于两个集合的元素，并创建新的集合
- 取出两个集合中的所有元素，并创建新的集合

我们把第 1 种集合称为交集，第 2 种集合成为并集。

- 交集.....`ary = ary1 & ary2`
- 并集.....`ary = ary1 | ary2`

图 13.10 描述的是 Ruby 数组中的交集和并集。

```

ary1 = ["a", "b", "c"]
ary2 = ["b", "c", "d"]
p (ary1 & ary2)  #=> ["b", "c"]
p (ary1 | ary2)  #=> ["a", "b", "c", "d"]

```

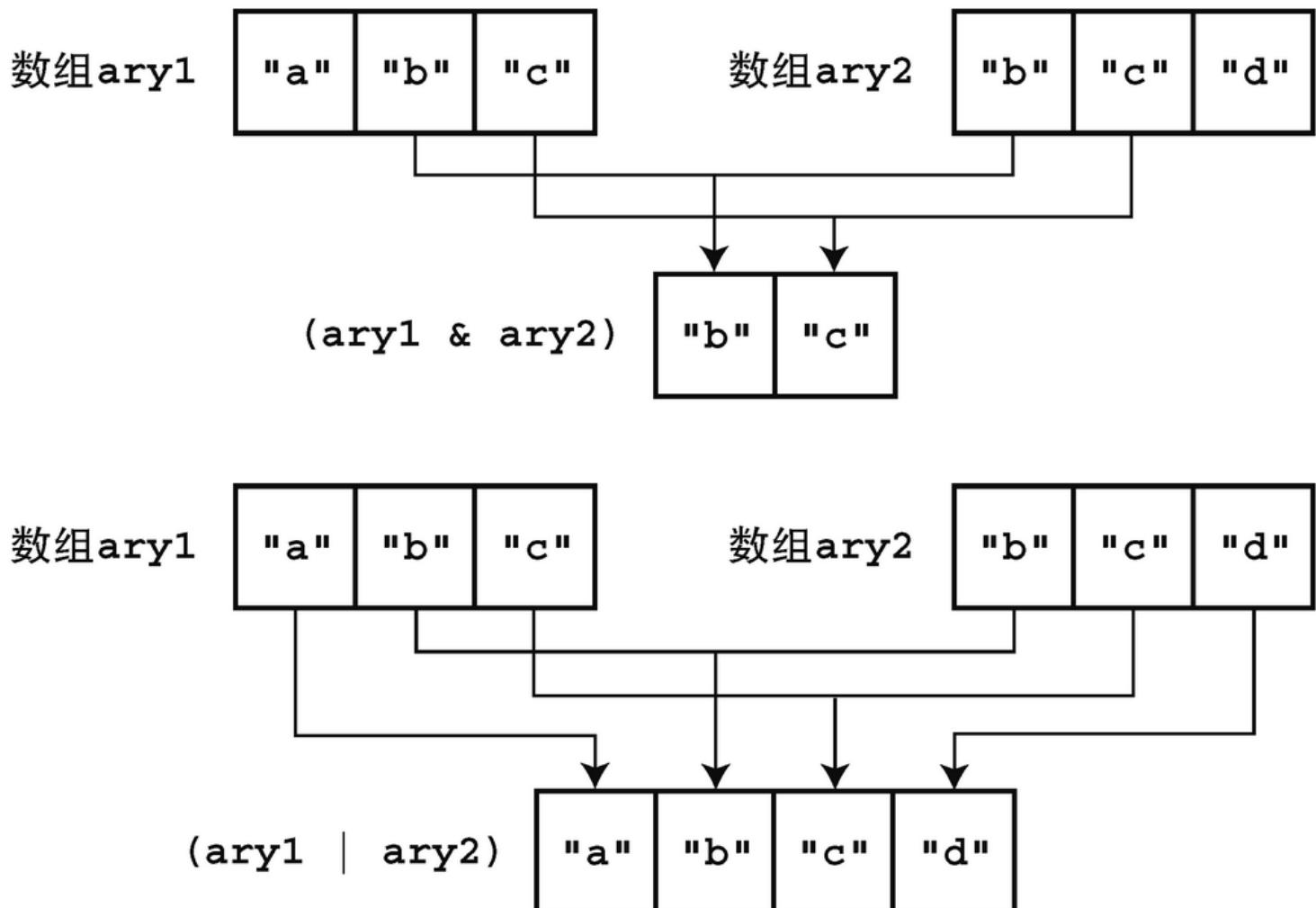


图 13.10 交集与并集

集合还有另外一种运算——补集，即获取某个集合中不属于另外一个集合的元素。但是 `Array` 类的情况下，由于没有全集的概念，因此也就没有补集。不过 `Array` 类有把某个集合中属于另外一个集合的元素删除的差运算（图 13.11）。

- 集合的差 .....`ary = ary1 - ary2`

```

ary1 = ["a", "b", "c"]
ary2 = ["b", "c", "d"]
p (ary1 - ary2)    #=> ["a"]

```

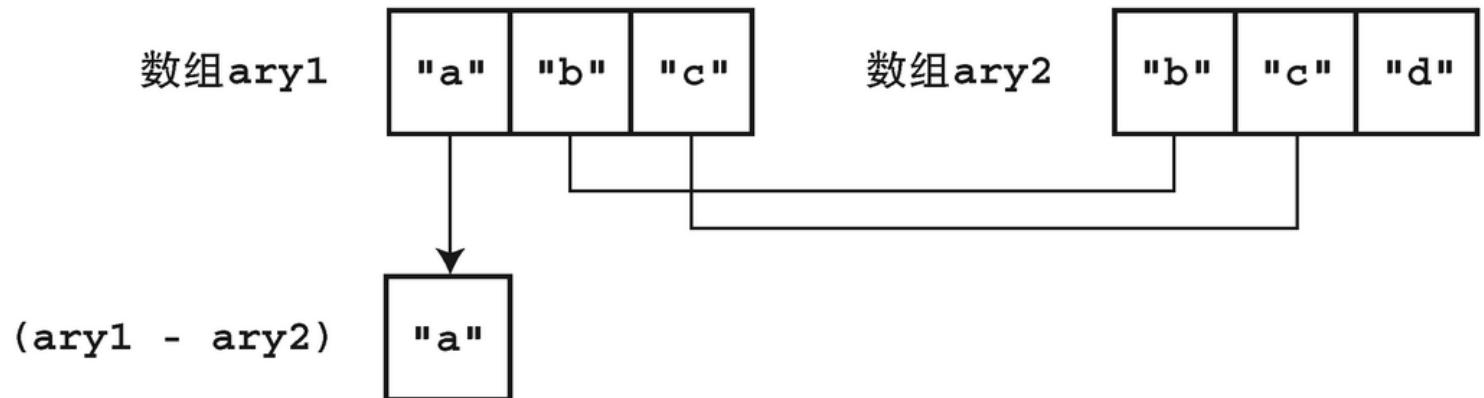


图 13.11 集合的差

由于图 13.11 的数组 ary2 中包含的字符串 "d" 在数组 ary1 中并没有，因此不会被保留在执行结果中。

#### “|”与“+”的不同点

连接数组除了可以使用 `|` 外还可以使用 `+`。这两种方法看起来比较相似，但是有相同元素时它们的效果就不一样了。

```

num = [1, 2, 3]
even = [2, 4, 6]
p (num + even)    #=> [1, 2, 3, 2, 4, 6]
p (num | even)    #=> [1, 2, 3, 4, 6]

```

数组 `num` 与数组 `even` 都有元素 2。使用 `+` 时元素 2 会有两个，使用 `|` 时相同的元素只会有一个。

## 13.5 作为列的数组

下面，我们来看看把数组对象当作列来看待时的情况。

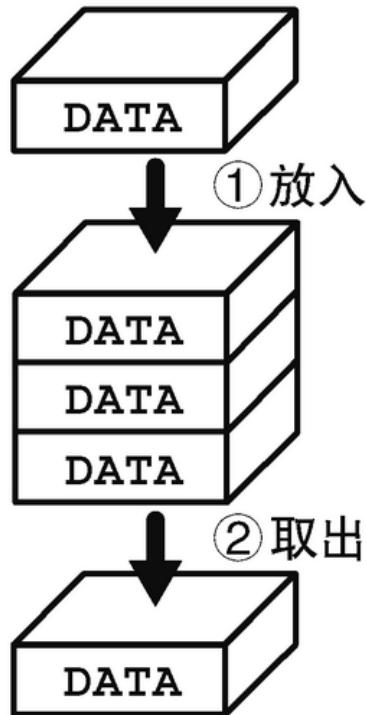
数据结构的队列（queue）和栈（stack）都是典型的列结构。这两个相对的数据结构都有以下两种操作数据的方式。

- 追加元素
- 获取元素

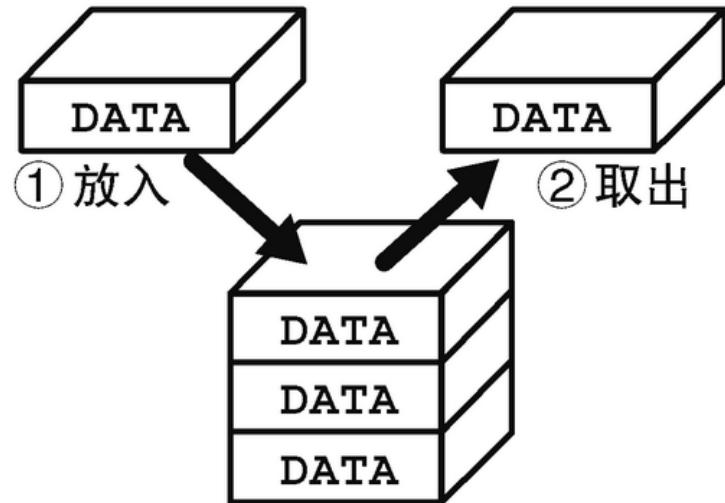
队列是一种按元素被追加时的顺序来获取元素的数据结构（图 13.12 (a)）。这样的做法称为 FIFO（First-in First-out），也就是“先进先出”的意思。这与人们为等待某件事而排成一列时的情况一样，因此有时候也称为等待队列。

而栈则是一种按照与元素被追加时的顺序相反的顺序来获取元素的数据结构。这样的做法称为 LIFO（Last-in First-out），是一种“先进后出”的数据结构（图 13.12 (b)）。也就是说，在末尾追加元素，并从末尾开始获取元素。

( a ) 队列



( b ) 栈



后放进去的东西先拿出来

## 先放进去的东西先拿出来

图 13.12 队列与栈

简单地说就是，按 A、B、C 的顺序保存数据时，按照 A、B、C 的顺序取得数据的数据结构就是队列，按照 C、B、A 的顺序取得数据的数据结构就是栈。

队列与栈都是比较复杂的数据结构，同时也是提高程序运行效率所不可缺少的工具。

在数组的开头或末尾插入元素，或者从数组的开头或末尾获取元素等操作，是实现队列、栈等数据结构所必须的前提条件。Ruby 的数组封装了如表 13.1 所示的方法，因此可以很轻松地实现这些前提条件。

表 13.1 操作数组开头与末尾的元素的方法

	对数组开头的元素的操作	对数组末尾的元素的操作
追加元素	<code>unshift</code>	<code>push</code>
删除元素	<code>shift</code>	<code>pop</code>
引用元素	<code>first</code>	<code>last</code>

利用图 13.13 所示的 `push` 方法和 `shift` 方法可以实现队列，利用图 13.14 所示的 `push` 方法和 `pop` 方法可以实现栈。

```

alpha = ["a", "b", "c", "d", "e"]
p alpha.push("f")    #=> ["a", "b", "c", "d", "e", "f"]
p alpha.shift         #=> "a"
p alpha              #=> ["b", "c", "d", "e", "f"]

```

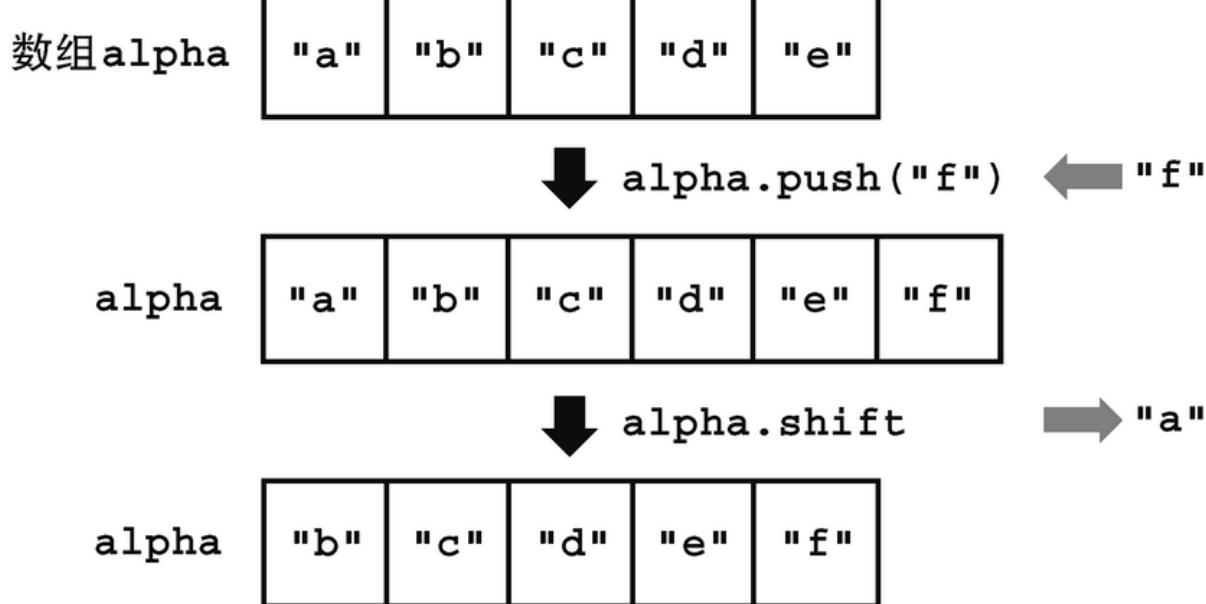


图 13.13 队列

```

alpha = ["a", "b", "c", "d", "e"]
p alpha.push("f")    #=> ["a", "b", "c", "d", "e", "f"]
p alpha.pop          #=> "f"
p alpha              #=> ["a", "b", "c", "d", "e"]

```

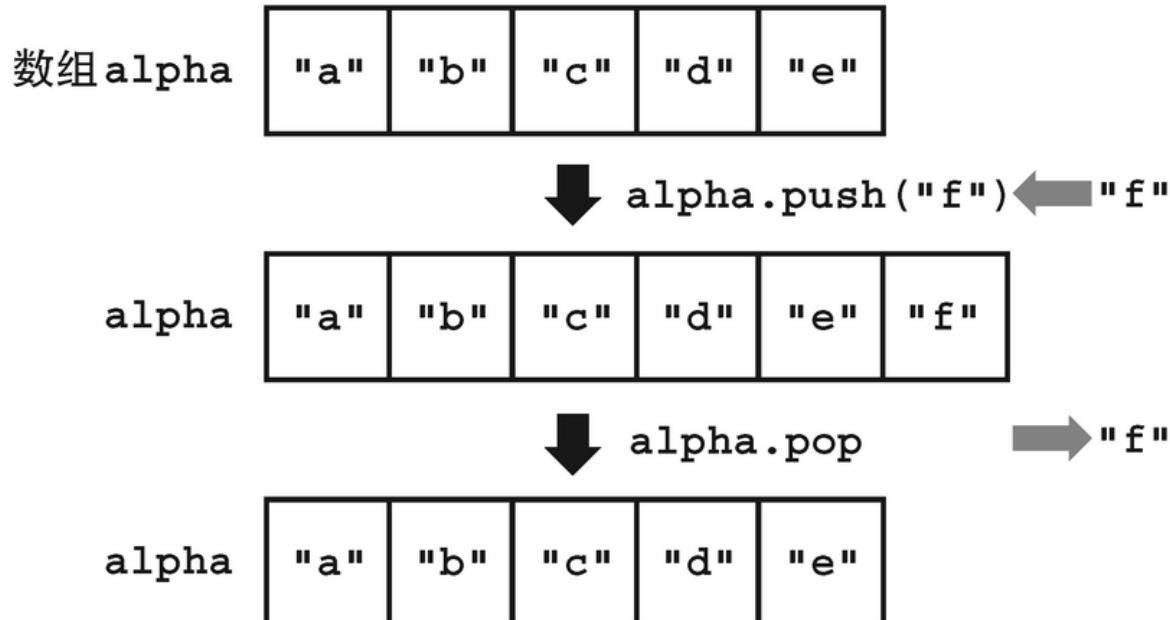


图 13.14 栈

要注意的是，`shift`方法和`pop`方法不只是获取数组元素，而且还会把该元素从数组中删除。如果只是单纯地希望引用元素，则应该使用`first`方法和

`last` 方法。

```
a = [1, 2, 3, 4, 5]
p a.first    #=> 1
p a.last     #=> 5
p a          #=> [1, 2, 3, 4, 5]
```

## 13.6 主要的数组方法

数组方法有很多，下面我们将选取最常用的几种方法，并把具有相同功能的方法归纳在一起分别加以介绍。

### 13.6.1 为数组添加元素

- `a.unshift (item)`

将 `item` 元素添加到数组的开头。

```
a = [1, 2, 3, 4, 5]
a.unshift(0)
p a    #=> [0, 1, 2, 3, 4, 5]
```

- `a << item`

`a.push (item)`

`<<` 与 `push` 是等价的方法，在数组 `a` 的末尾添加新元素 `item`。

```
a = [1, 2, 3, 4, 5]
a << 6
p a    #=> [1, 2, 3, 4, 5, 6]
```

- `a.concat (b)`

`a + b`

连接数组 `a` 和数组 `b`。`concat` 是具有破坏性的方法，而 `+` 则会根据原来的数组元素创建新的数组。

```
a = [1, 2, 3, 4, 5]
a.concat([8, 9])
p a    #=> [1, 2, 3, 4, 5, 8, 9]
```

- `a[n] = item`

`a[n..m] = item`

`a[n, len] = item`

把数组 `a` 指定的部分的元素替换为 `item`。

```
a = [1, 2, 3, 4, 5, 6, 7, 8]
a[2..4] = 0
p a    #=> [1, 2, 0, 6, 7, 8]
a[1, 3] = 9
p a    #=> [1, 9, 7, 8]
```

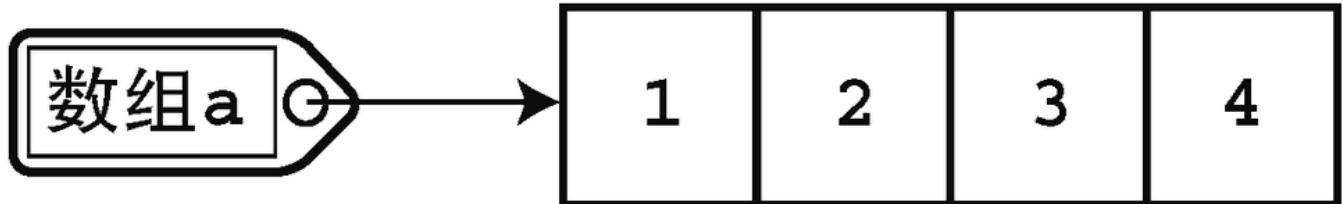
### 专栏

#### 具有破坏性的方法

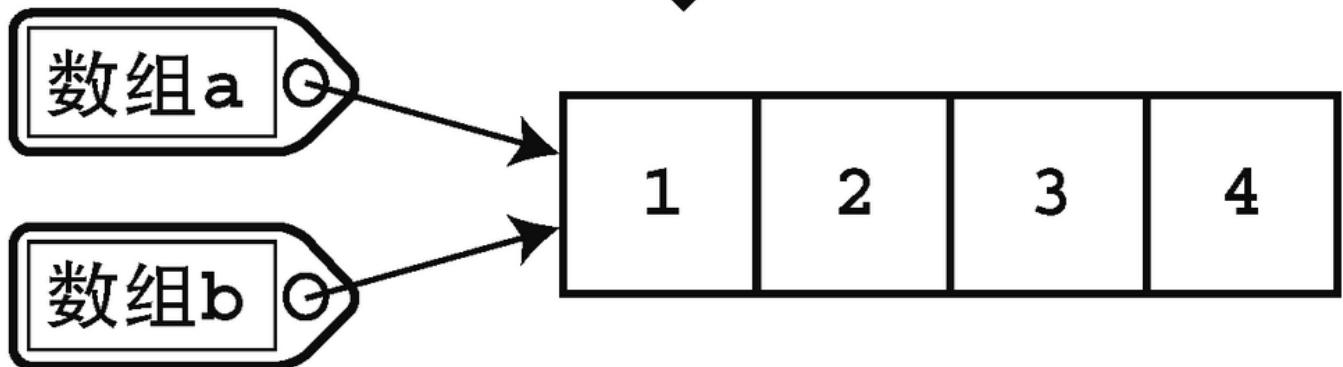
像 `pop` 方法、`shift` 方法那样，会改变接收者对象值的方法称为具有破坏性的方法。在使用具有破坏性的方法时需要特别小心，因为当有变量也引用了接收者对象时，如果接受者对象值发生了改变，变量值也会随之发生变化。我们来看看下面的例子。

```
a = [1, 2, 3, 4]
b = a
p b.pop    #=> 4
p b        #=> [1, 2, 3]
p a        #=> [1, 2, 3]
```

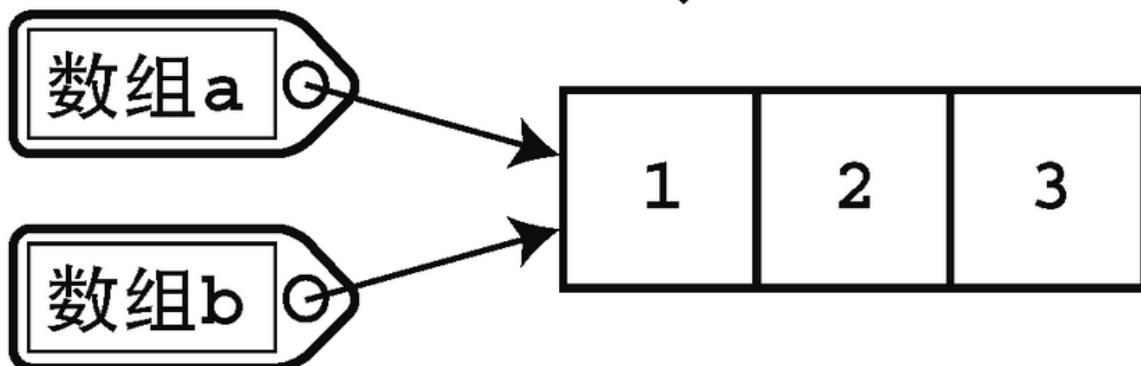
执行 `pop` 方法删除元素后，变量 `a` 引用的数组的元素也被删除，从 `[1, 2, 3, 4]` 变为了 `[1, 2, 3]`，同时变量 `b` 引用的数组元素也被删除了。这是由于执行 `b = a` 后，并不是将变量 `a` 的内容复制给了变量 `b`，而是让变量 `b` 和变量 `a` 同时引用了一个对象。



↓ 执行 `b = a`



↓ 执行 `b.pop` → 4



在 Ruby 的方法中，有像 `sort` 和 `sort!` 这样，在相同方法名后加上 `!` 的方法。为了区分方法是否具有破坏性，在具有破坏性的方法末尾添加 `!`。这一做法目前已经成为了通用的规则。

### 13.6.2 从数组中删除元素

根据某些条件从数组中删除元素。

- `a.compact`  
`a.compact!`

从数组 `a` 中删除所有 `nil` 元素。`compact` 方法会返回新的数组，`compact!` 则直接替换原来的数组。`compact!` 方法返回的是删除 `nil` 元素后的 `a`，但是如果什么都没有删除的话就会返回 `nil`。

```
a = [1, nil, 3, nil, nil]
a.compact!
p a    #=> [1, 3]
```

- `a.delete(x)`

从数组 `a` 中删除 `x` 元素。

```
a = [1, 2, 3, 2, 1]
a.delete(2)
p a #=> [1, 3, 1]
```

- `a.delete_at(n)`

从数组中删除 `a[n]` 元素。

```
a = [1, 2, 3, 4, 5]
a.delete_at(2)
p a    #=> [1, 2, 4, 5]
```

- `a.delete_if{|item| ... }`  
`a.reject{|item| ... }`  
`a.reject!{|item| ... }`

判断数组 `a` 中的各元素 `item`, 如果块的执行结果为真, 则从数组 `a` 中删除 `item`。`delete_if` 和 `reject!` 方法都是具有破坏性的方法。

```
a = [1, 2, 3, 4, 5]
a.delete_if{|i| i > 3}
p a    #=> [1, 2, 3]
```

- `a.slice!(n)`  
`a.slice!(n..m)`  
`a.slice!(n, len)`

删除数组 `a` 中指定的部分, 并返回删除部分的值。`slice!` 是具有破坏性的方法。

```
a = [1, 2, 3, 4, 5]
p a.slice!(1, 2)    #=> [2, 3]
p a                #=> [1, 4, 5]
```

- `a.uniq`  
`a.uniq!`

删除数组 `a` 中重复的元素。`uniq!` 是具有破坏性的方法。

```
a = [1, 2, 3, 4, 3, 2, 1]
a.uniq!
p a    #=> [1, 2, 3, 4]
```

- `a.shift`

删除数组 `a` 开头的元素, 并返回删除的值。

```
a = [1, 2, 3, 4, 5]
a.shift    #=> 1
p a        #=> [2, 3, 4, 5]
```

- `a.pop`

删除数组 `a` 末尾的元素, 并返回删除的值。

```
a = [1, 2, 3, 4, 5]
a.pop    #=> 5
p a        #=> [1, 2, 3, 4]
```

### 13.6.3 替换数组元素

将数组元素替换为别的元素的方法中, 也分为带 `!` 的和不带 `!` 的方法, 前者是具有破坏性的会改变接收者对象值的方法, 后者则是直接返回新数组的方法。

- `a.collect{|item| ... }`  
`a.collect!{|item| ... }`  
`a.map{|item| ... }`  
`a.map!{|item| ... }`

将数组 `a` 的各元素 `item` 传给块, 并用块处理过的结果创建新的数组。从结果来看, 数组的元素个数虽然不变, 但由于经过了块处理, 因此数组的元素和之前会不一样。

```
a = [1, 2, 3, 4, 5]
a.collect!{|item| item * 2}
p a    #=> [2, 4, 6, 8, 10]
```

- `a.fill(value)`  
`a.fill(value, begin)`  
`a.fill(value, begin, len)`  
`a.fill(value, n..m)`

将数组 `a` 的元素替换为 `value`。参数为一个时, 数组 `a` 的所有元素值都会变为 `value`。参数为两个时, 从 `begin` 到数组末尾的元素值都会变为 `value`。参

数为三个时，从 `begin` 开始 `len` 个元素的值会变为 `value`。另外，当第 2 个参数指定为 `[n..m]` 时，则指定范围内的元素值都会变为 `value`。

```
p [1, 2, 3, 4, 5].fill(0)      #=> [0, 0, 0, 0, 0]
p [1, 2, 3, 4, 5].fill(0, 2)    #=> [1, 2, 0, 0, 0]
p [1, 2, 3, 4, 5].fill(0, 2, 2) #=> [1, 2, 0, 0, 5]
p [1, 2, 3, 4, 5].fill(0, 2..3) #=> [1, 2, 0, 0, 5]
```

- `a.flatten`

```
a.flatten!
```

平坦化数组 `a`。所谓平坦化是指展开嵌套数组，使嵌套数组变为一个大数组。

```
a = [1, [2, [3]], [4], 5]
a.flatten!
p a    #=> [1, 2, 3, 4, 5]
```

- `a.reverse`

```
a.reverse!
```

反转数组 `a` 的元素顺序。

```
a = [1, 2, 3, 4, 5]
a.reverse!
p a #=> [5, 4, 3, 2, 1]
```

- `a.sort`

```
a.sort!
```

```
a.sort{|i, j| ...}
```

```
a.sort!{|i, j| ...}
```

对数组 `a` 进行排序。排序方法可以由块指定。没有块时，使用 `<=>` 运算符比较。

```
a = [2, 4, 3, 5, 1]
a.sort!
p a    #=> [1, 2, 3, 4, 5]
```

关于如何使用块指定排序方法，在 11.2.3 节中我们已经介绍过了。

- `a.sort_by{|i| ...}`

对数组 `a` 进行排序。根据块的运行结果对数组的所有元素进行排序。

```
a = [2, 4, 3, 5, 1]
p a.sort_by{|i| -i}    #=> [5, 4, 3, 2, 1]
```

详情请参考 11.2.3 节。

## 13.7 数组与迭代器

迭代器是实现循环处理的方法，而数组则是多个对象的集合。在对这些对象进行某种处理，或者取出某几个对象时，都需要大量用到迭代器。

例如，对数组的各元素进行相同的操作时使用的 `each` 方法就是典型的迭代器。该方法会遍历数组的所有元素，并对其进行特定的处理。

此外，接收者不是数组的情况下，为了让迭代器的执行结果能作为某个对象返回，也会用到数组。其中 `collect` 方法就是一个具有代表性的方法。`collect` 方法会收集某种处理的结果，并将其合并为一个数组后返回。

```
a = 1..5
b = a.collect{|i| i += 2}
p b    #=> [3, 4, 5, 6, 7]
```

在上面的例子中，接收者为范围对象，而结果则是数组对象。像这样，迭代器和数组就被紧密地结合在一起了。

## 13.8 处理数组中的元素

对数组中的元素进行处理时可以采取多种方法。

### 13.8.1 使用循环与索引

传统的方法是使用循环，也就是在遍历数组的同时，利用索引逐个访问数组元素。

例如，在代码清单 13.1 中，我们把数组的元素逐个取出来并输出。

### 代码清单 13.1 list.rb

```
list = ["a", "b", "c", "d"]
for i in 0..3
  print "第", i+1,"个元素是",list[i],"\n"
end
```

代码清单 13.2 的程序是对数值数组的元素进行合计的例子。

### 代码清单 13.2 sum\_list.rb

```
list = [1, 3, 5, 7, 9]
sum = 0
for i in 0..4
  sum += list[i]
end
print "合计: ",sum,"\n"
```

#### 13.8.2 使用 each 方法逐个获取元素

在数组中，通过 `each` 方法可以实现循环操作。下面，我们尝试使用 `each` 方法来改写代码清单 13.2（代码清单 13.3）。

### 代码清单 13.3 sum\_list2.rb

```
list = [1, 3, 5, 7, 9]
sum = 0
list.each do |elem|
  sum += elem
end
print "合计: ",sum,"\n"
```

但是，使用 `each` 方法时，我们并不知道元素的索引值。因此，当需要指定元素的索引值时，可以使用 `each_with_index` 方法。

### 代码清单 13.4 list2.rb

```
list = ["a", "b", "c", "d"]
list.each_with_index do |elem, i|
  print "第", i+1,"个元素是",elem,".\n"
end
```

#### 13.8.3 使用具有破坏性的方法实现循环

如果数组内各元素全部处理完毕后该数组就不需要了，那么我们就可以通过逐个删除数组元素使数组变空这样的手段来实现循环。

```
while item = a.pop
  ## 对 item 进行处理
end
```

假设在循环开始前已经有元素在数组 `a` 中。如果逐个删除数组 `a` 中的元素，就会对删除的元素进行处理。最后，当数组为空时，循环结束。

#### 13.8.4 使用其他迭代器

Ruby 中实现了不少像 `collect`、`map` 方法这样一眼就能看出其作用的基本操作。当希望创建某种迭代器时，请翻阅 Ruby 参考手册，一般情况下在里面都能找到我们需要的迭代器。这样就不会因为花精力创建了一个 Ruby 本来就有的迭代器而感到失望了。

#### 13.8.5 创建专用的迭代器

不过有时也可能找不到自己想要的迭代器，这时就只能根据需要创建属于自己的迭代器了。

关于迭代器的创建，请参考 11.3 节。

## 13.9 数组的元素

数组中可以存放各种各样的对象。除了数值、字符串外，我们还可以在数组对象中存放别的数组对象或散列对象等等。

#### 13.9.1 使用简单的矩阵

下面我们来看看用数组来表示矩阵的例子。

数组的各个元素也可以是数组，也就是所谓的数组的数组，这样的形式经常被用于表示矩阵。

例如，我们试试用数组的数组这种形式来表示图 13.15 那样的矩阵。

1,	2,	3
4,	5,	6
7,	8,	9

图 13.15 3 行 3 列的矩阵

第 1 行为 [1, 2, 3]，第 2 行为 [4, 5, 6]，第 3 行为 [7, ,8 , 9]，把它们归纳为数组，如下所示。

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

如果想取出元素 6，我们可以像下面这样做。

```
a[1][2]
```

首先用 a[1] 表示 [4, 5, 6] 这个数组，然后再指定第 3 位的元素，这样就能达到我们的目的了。

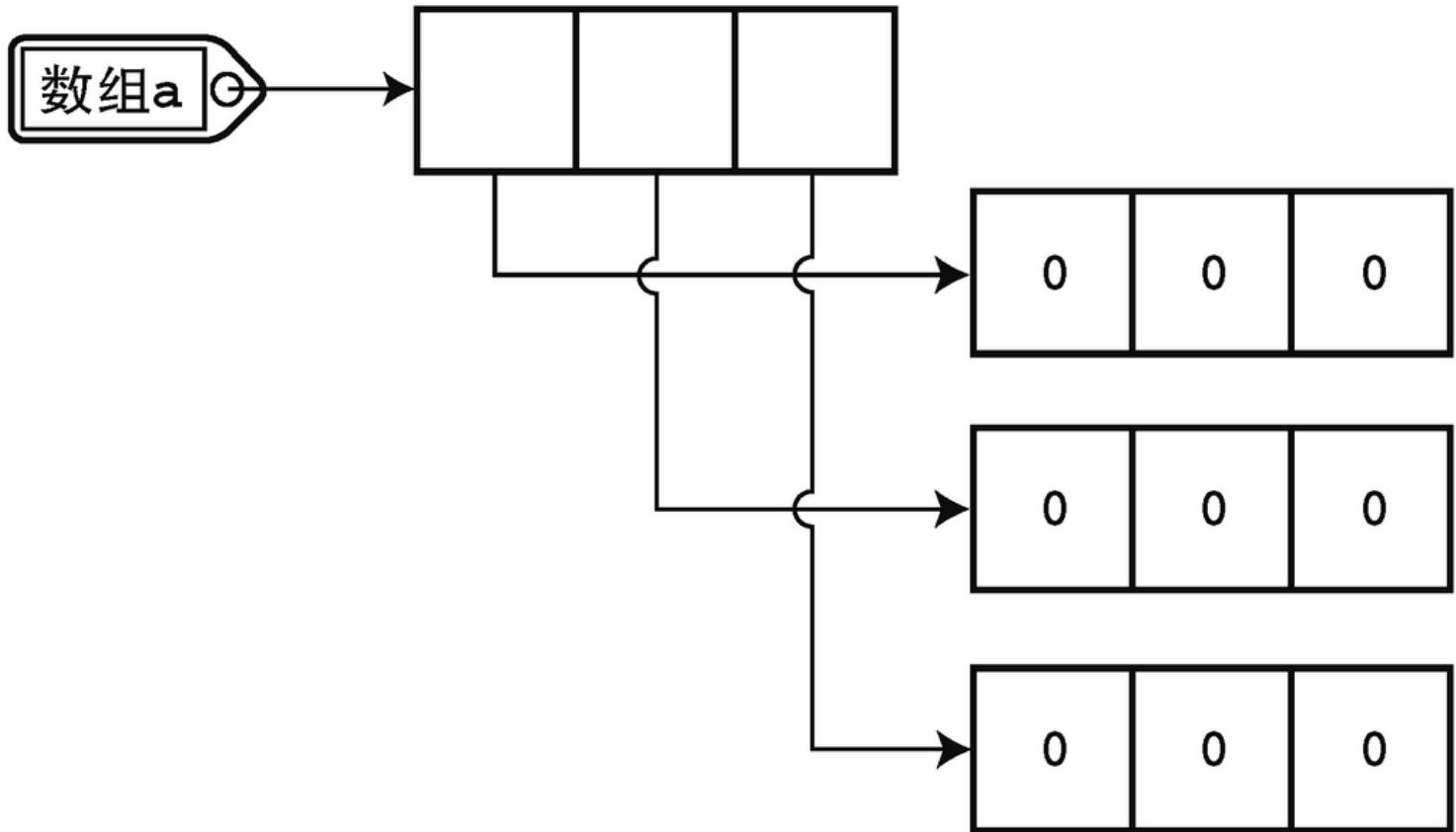
### 13.9.2 初始化时的注意事项

把数组对象或者散列对象作为数组元素时，需要注意该对象初始化时的问题。

```
a = Array.new(3, [0, 0, 0])
```

在上面的例子中，我们可能会以为 a 为 [[0, 0, 0], [0, 0, 0], [0, 0, 0]]，但实际却是另外一个结果（图 13.16）。

## ( a ) 期待的结果



## ( b ) `Array.new(3, [0, 0, 0])` 的结果

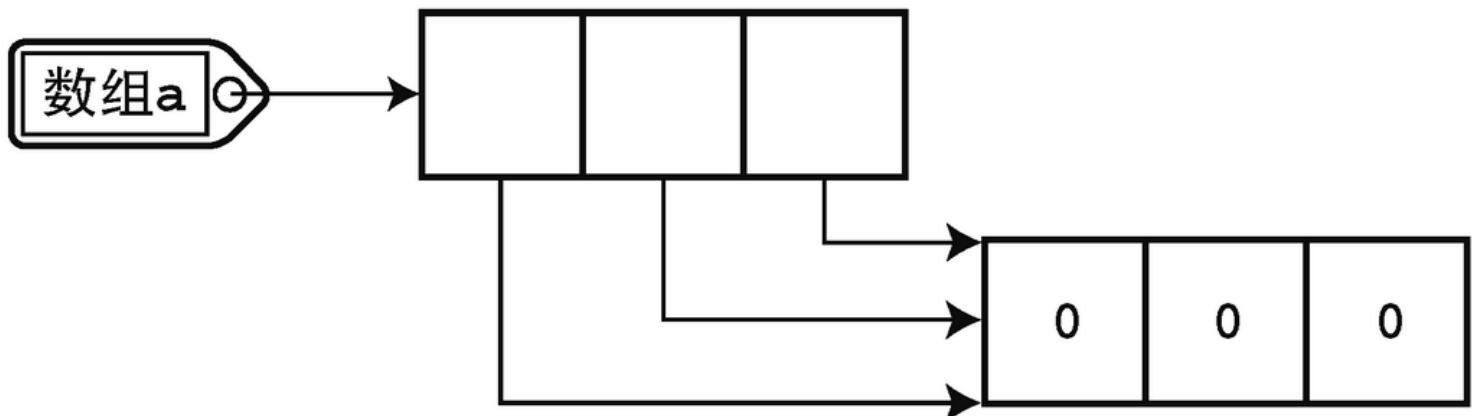


图 13.16 数组的初始化

像下面那样，原本只是打算变更第 1 行的第 2 个元素，结果所有行的第 2 个元素都发生了改变。

```
a = Array.new(3, [0, 0, 0])
a[0][1] = 2
p a    #=> [[0, 2, 0], [0, 2, 0], [0, 2, 0]]
```

为了解决这个问题，我们可以指定 `new` 方法的块和元素个数。程序调用与元素个数一样次数的块，然后再将块的返回值赋值给元素。每次调用块都会生成新的对象，这样一来，各个元素引用同一个对象的问题就不会发生了。

```
a = Array.new(3) do
  [0, 0, 0]
end
p a    #=> [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

a[0][1] = 2
p a    #=> [[0, 2, 0], [0, 0, 0], [0, 0, 0]]
```

进行下述操作后，对应的元素的索引值就会被赋值给 `i`，这样就可以根据索引值初始化出不同的值了。

```
a = Array.new(5){|i| i + 1}
p a    #=> [1, 2, 3, 4, 5]
```

## 13.10 同时访问多个数组

接下来我们来看看用相同的索引值同时访问对多个数组时的情况。在代码清单 13.5 中，合计三个数组中索引值相同的元素，并将结果保存在新数组（`result`）中。

代码清单 13.5 `sum_with_each.rb`

```
ary1 = [1, 2, 3, 4, 5]
ary2 = [10, 20, 30, 40, 50]
ary3 = [100, 200, 300, 400, 500]

i = 0
result = []
while i < ary1.length
  result << ary1[i] + ary2[i] + ary3[i]
  i += 1
end
p result  #=> [111, 222, 333, 444, 555]
```

使用 `zip` 方法可以程序变得更简单（代码清单 13.6）。

代码清单 13.6 `sum_with_zip.rb`

```
ary1 = [1, 2, 3, 4, 5]
ary2 = [10, 20, 30, 40, 50]
ary3 = [100, 200, 300, 400, 500]

result = []
ary1.zip(ary2, ary3) do |a, b, c|
  result << a + b + c
end
p result  #=> [111, 222, 333, 444, 555]
```

`zip` 方法会将接收器和参数传来的数组元素逐一取出，而且每次都会启动块。参数可以是一个也可以是多个。

### 专栏

#### Enumerable 模块

介绍完 `Comparable` 模块后，我们再来介绍一下常被用于 Mix-in 的 `Enumerable` 模块。`Enumerable` 的意思是“可以被计数的”、“可以被列举的”。在本书介绍过的类中，`Array`、`Dir`、`File`、`Hash`、`IO`、`Range`、`Enumerator` 等类中都包含了 `Enumerable` 模块。

表 `Enumerable` 模块定义的方法

<code>all?</code>	<code>any?</code>	<code>chunk</code>	<code>collect</code>
<code>collect_concat</code>	<code>count</code>	<code>cycle</code>	<code>detect</code>
<code>drop</code>	<code>drop_while</code>	<code>each_cons</code>	<code>each_entry</code>
<code>each_slice</code>	<code>each_with_index</code>	<code>each_with_object</code>	<code>entries</code>
<code>find</code>	<code>find_all</code>	<code>find_index</code>	<code>first</code>
<code>flat_map</code>	<code>grep</code>	<code>group_by</code>	<code>include?</code>
<code>inject</code>	<code>lazy</code>	<code>map</code>	<code>max</code>
<code>max_by</code>	<code>member?</code>	<code>min</code>	<code>min_by</code>
<code>minmax</code>	<code>minmax_by</code>	<code>none?</code>	<code>one?</code>
<code>partition</code>	<code>reduce</code>	<code>reject</code>	<code>reverse_each</code>
<code>select</code>	<code>slice_before</code>	<code>sort</code>	<code>sort_by</code>
<code>take</code>	<code>take_while</code>	<code>to_a</code>	<code>zip</code>

本章中介绍的 `Array` 类的方法中，实际上有些是在 `Enumerable` 模块中定义的。关于上文中没有介绍的方法的用法，请参考 Ruby 参考手册。

就像 Comparable 模块需要 `<=>` 运算符那样，Enumerable 模块则需要 `each` 方法。例如，如果用 Ruby 来实现 `each_with_index` 方法的话，大概就是下面这样（实际的程序会更加复杂，例如会有没有块时返回 `Enumerator` 对象等处理）。

```
module Enumerable
  def each_with_index
    index = 0          # 初始化索引
    each do |item|
      yield(item, index) # 将元素与index 作为参数
      # 执行块
    index += 1          # 累加索引值
  end
end
```

我们在创建提供循环处理的类的时候，可以首先创建迭代器 `each` 方法，然后再包含 `Enumerable` 模块，这样一来，上表的方法就都可以使用了。

## 练习题

1. 创建一个数组 `a`，使 1 到 100 的整数按升序排列（即 `a[0]` 为 1，`a[99]` 为 100）。
2. 将 `a` 数组中的各元素扩大 100 倍，创建新数组 `a2`（即 `a2[0]` 为 100）。另外，不创建新数组，将原数组中的各元素都扩大 100 倍。
3. 获取 `a` 数组中值是 3 的倍数的元素，创建新数组 `a3`（即 `a3[0]` 为 3，`a3[2]` 为 9）。另外，不创建新数组，把 3 的倍数以外的元素全部删除。
4. 将 `a` 数组按倒序排列。
5. 求 `a` 数组中的整数的和。
6. 从 1 到 100 的整数数组中，取出 10 个分别包含 10 个元素的数组，如  $1 \sim 10$ ,  $11 \sim 20$ ,  $21 \sim 30$ 。再将取出的全部数组按顺序保存到数组 `result` 中，请考虑以下代码中 `???` 的部分应该怎么写。

```
ary = [ 包含1~100 的整数数组 ]
result = Array.new
10.times do |i|
  result << ary[???
end
```

7. 定义方法 `sum_array`，合计数组 `nums1` 和 `nums2` 中相对应的各个元素的值，并将合计结果作为数组返回。

```
p sum_array([1, 2, 3], [4, 6, 8])  #=> [5, 8, 11]
```

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

# 第 14 章 字符串类

正如在第 4 章中说明的那样，Ruby 的字符串都是 `String` 类的对象。本章将详细介绍 `String` 类。

- 字符串的创建

列举各种创建字符串的方法。

- 获取字符串的长度

介绍获取字符串长度的方法，说明“字符数”与“字节数”的不同。

- 字符串的索引

与数组一样，也是通过索引进行字符串操作，在这一部分中我们会介绍字符串中索引的用法。

- 字符串的连接与分割

介绍连接字符串的方法 `+`、`<<`、`concat`，分割字符串的方法 `split`。

- 字符串的比较

介绍判断字符串是否相等的方法、以及在进行字符串排序时如何比较字符串的“大小”。

- 换行符的使用方法

介绍字符串中换行符的使用方法。

- 字符串的检索与置换

介绍检索与置换字符串的方法。

- 字符串与数组的共通方法

字符串与数组中有不少作用相似的同名方法，在这一部分中我们会集中介绍这些方法。

- 日语字符编码的转换

介绍如何对日语字符串进行必要的字符编码转换。

## 14.1 字符串的创建

最简单的字符创建方法就是把字符的集合用 `" "` 或者 `' '` 括起来并直接写到程序中。

```
str1 = "这也是字符串"
str2 = '那也是字符串'
```

**注** 在脚本中使用日语时需要注意编码（encoding）问题，相关内容请参考 19.2 节 1。

1在脚本中使用中文时也同样需要注意编码问题。——译者注

在第 1 章中，我们已经简单地介绍了用 `" "` 与用 `' '` 创建字符串的不同点。而除此以外，使用 `" "` 时还可以执行用 `#{}`  括起来的 Ruby 式子，并将执行结果嵌入到字符串中。这个 `#{}`  就称为内嵌表达式（embedded expressions）。

```
moji = "字符串"
str1 = "那也是#{moji}"
p str1    #=> "那也是字符串"
str2 = ' 那也是#{moji}'
p str2    #=> "那也是\#{moji}"
```

使用 `" "` 时，可以显示使用 \ 转义的特殊字符（表 14.1）。

表 14.1 使用 \ 转义的特殊字符

特殊字符	意义
\t	水平制表符（0x09）
\n	换行符（0x0a）

\r	回车 (0x0d)
\f	换页 (0x0c)
\b	退格 (0x08)
\a	响铃 (0x07)
\e	溢出 (0x1b)
\s	空格 (0x20)
\v	垂直制表符 (0x0b)
\nnn	8 进制表示方式 ( <i>n</i> 为 0~7)
\xnn	16 进制表示方式 ( <i>n</i> 为 0~9、a~f、A~F)
\C-X、\C-x	Control + X
\M-X	Meta(Alt) + X
\M-\C-x	Meta(Alt) + Control + X
\x	表示 x 字符本身 (x 为除以上字符外的字符)
\Unnnn	Unicode 字符的 16 进制表示方式 ( <i>n</i> 为 0~9、a~f、A~F)

下面我们来看看不用 “”、“” 时应如何创建字符串。

#### 14.1.1 使用 %Q 与 %q

当创建包含 “” 或者 ‘’ 的字符串时，比起使用 \"、\' 进行转义，使用 %Q 或者 %q 会更简单。

```
desc = %Q{Ruby 的字符串中也可以使用' 和"。}
str = %q|Ruby said, 'Hello world!|
```

使用 %Q 相当于用 “” 创建字符串，使用 %q 则相当于用 ‘’ 创建字符串。

#### 14.1.2 使用 Here Document

Here Document 是源自于 Unix 的 shell 的一种程序写法，使用 << 来创建字符串。创建包含换行的长字符串时用这个方法是最简单的。

<<"结束标识符"

字符串内容

结束标识符

<< 后面的结束标识符可以用 “” 括着的字符串或者用 ‘’ 括着的字符串来定义。用 “” 括住的字符串中可以使用转义字符和内嵌表达式，而用 ‘’ 括住的字符串则不会做任何特殊处理，只会原封不动地显示。另外，使用既没有 “” 也没有 ‘’ 的字符串时，则会被认为是用 “” 创建的字符串。

由于 Here Document 整体就是字符串的字面量，因此可以被赋值给变量，也可以作为方法的参数。

我们一般将 `EOF` 或者 `EOB` 作为结束标识符使用。`EOF` 是“End of File”的简写，`EOB` 是“End of Block”的简写。

Here Document 的结束标识符一定要在行首。因此，在程序缩进比较深的地方使用 Here Document 的话，有时就会像下面的例子那样，出现整个缩进乱掉的情况。

```
10.times do |i|
  10.times do |j|
    print(<<"EOF")
i: #{i}
j: #{j}
i*j = #{i*j}
EOB
end
end
```

若希望缩进整齐，可以像下面那样用 `<<-` 代替 `<<`。这样程序就会忽略结束标识符前的空格和制表符，结束标识符也就没有必要一定要写在行首了。

```
10.times do |i|
  10.times do |j|
    print(<<-"EOF")
i: #{i}
j: #{j}
i*j = #{i*j}
  EOB
end
end
```

这样操作以后，`print` 和 `EOB` 的缩进就会变得整齐，便于阅读。下面是将 Here Document 赋值给变量时的做法。

```
str = <<-EOB
Hello!
Hello!
Hello!
  EOB
```

#### 14.1.3 使用 `sprintf` 方法

就像用 8 进制或 16 进制的字符串来表示数值那样，我们也可以用 `sprintf` 方法输出某种格式的字符串。关于 `sprintf` 的具体使用方法，请参考专栏《printf 方法与 sprintf 方法》。

#### 14.1.4 使用 ``

通过用 `命令` 的形式，我们可以得到命令的标准输出并将其转换为字符串对象。下面是获取 Linux 的 `ls` 命令与 `cat` 命令的输出内容的例子：

##### 执行示例

```
> irb --simple-prompt
>> `ls -l /etc/hosts`
=> "-rw-r--r-- 1 root root 158 Jan 12 2010 /etc/hosts\n"
>> puts `cat /etc/hosts`
# Host Database
#
127.0.0.1 localhost
255.255.255.255 broadcasthost
::1 localhost
fe80::1%lo0 localhost
=> nil
```

##### 专栏

##### `printf` 方法与 `sprintf` 方法

虽然 `printf` 方法与 `sprintf` 方法都不是 `String` 类的方法，但在处理字符串时会经常用到它们。下面我们就来看看它们的用法。

- 关于 `printf` 方法

`printf` 方法可以按照某种格式输出字符串。例如在输出数值时，有时我们会需要在数值前补零，或者限定小数点显示的位数等，在这些情况下，用 `printf` 方法都能非常轻松地实现。

```
1: n = 123
2: printf("%d\n", n)
3: printf("%4d\n", n)
4: printf("%04d\n", n)
```

```
5: printf("%d\n", n)
```

执行结果如下。

```
123  
123  
0123  
+123
```

`printf` 方法的第 1 个参数表示字符串的输出格式。而从第 2 个参数开始，往后的参数都会被依次嵌入到格式中 `%` 所对应的位置。

在本例中，第 2 行的 `printf` 方法被指定为了 `%d`，这表示输出的字符是整数。

`%` 与 `d` 之间还能插入字符。第 3 行的 `printf` 方法里插入了 `4`，这表示按照 4 位整数的格式输出。我们发现，执行结果中出现了 `123` 这种开头有 1 个空格的情况，这是因为要把 3 位整数以 4 位的格式输出，因此就多输出了 1 个空格。

在第 4 行中，`%` 与 `d` 中间插入了 `04`，这表示若输出的整数位数不足，整数的开头就会做补零处理。

第 5 行中指定了 `+`，这表示输出的结果一定会包含 `+` 或者 `-`。

上面是关于数值格式的指定方法，同样，我们也可以指定字符串格式。

```
1: n = "Ruby"  
2: printf("Hello,%s!\n", n)  
3: printf("Hello,%8s!\n", n)  
4: printf("Hello,%-8s!\n", n)
```

执行结果如下：

```
Hello,Ruby!  
Hello,      Ruby!  
Hello,Ruby    !
```

本例的第 2 行程序中指定了 `%s`，这表示将参数解析为字符串。参数 `n` 的值为 `Ruby`，因此输出的字符串为 `Hello,Ruby!`。

在第 3 行中，`%` 与 `s` 之间插入了数字 8，这表示将 `Ruby` 输出为 8 位字符串。

在第 4 行中，插入的内容为 `-8`，这表示按靠左对齐的方式输出 8 位字符串。

#### • 关于 `sprintf` 方法

`printf` 方法会把内容输出到控制台，而 `sprintf` 方法则是把同样的输出内容转换为字符串对象。开头的 `s` 指的就是 `String`。

```
p sprintf("%d", 123)          #=> "123"  
p sprintf("%04d", 123)         #=> "0123"  
p sprintf("%+d", 123)          #=> "+123"  
p sprintf("Hello,%s!\n", n)     #=> "Hello,Ruby!"  
p sprintf("Hello,%8s!\n", n)     #=> "Hello,      Ruby!"  
p sprintf("Hello,%-8s!\n", n)    #=> "Hello,Ruby    !"
```

## 14.2 获取字符串的长度

我们用 `length` 方法和 `size` 方法获取字符串的长度。两者都返回相同的结果，大家根据自己的习惯选用即可。

```
p "just another ruby hacker.".length  #=> 25  
p "just another ruby hacker.".size      #=> 25
```

若是中文字符串，则返回字符数。

```
p '面向对象编程语言'.length #=> 8
```

如果想获取的不是字符数，而是字节数，可以用 `bytesize` 方法。

```
p '面向对象编程语言'.bytesize #=> 24
```

想知道字符串的长度是否为 0 时，可以使用 `empty?` 方法，该方法常被用于在循环等处理中判断字符串是否为空。

```
p "".empty?    #=> true  
p "foo".empty? #=> false
```

## 14.3 字符串的索引

获取字符串中指定位置的字符，例如获取“开头第 3 位的字符”时，与数组一样，我们也需要用到索引。

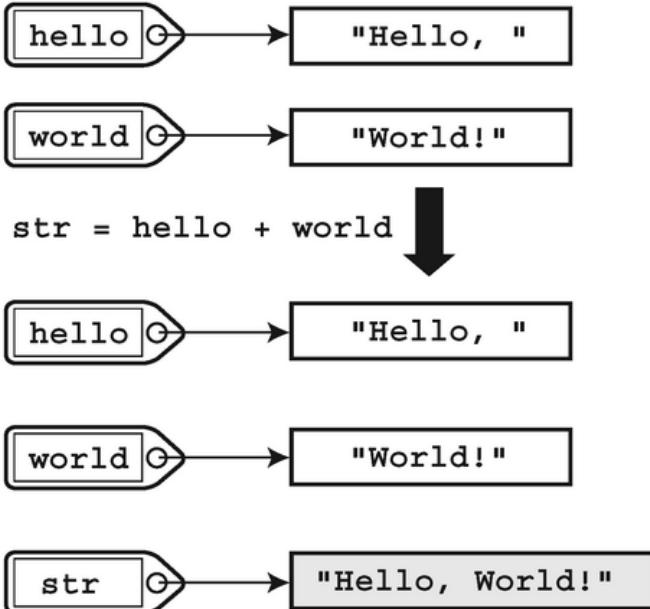
```
str = "全新的String 类对象"
p str[0]    #=> "全"
p str[3]    #=> "S"
p str[9]    #=> "类"
p str[2, 8] #=> "的String 类"
p str[4]    #=> "t"
```

## 14.4 字符串的连接

连接字符串有以下两种方法（图 14.1）：

- 将两个字符串合并为新的字符串
- 扩展原有的字符串

( a ) 将两个字符串合并为新的字符串



( b ) 扩展原有的字符串

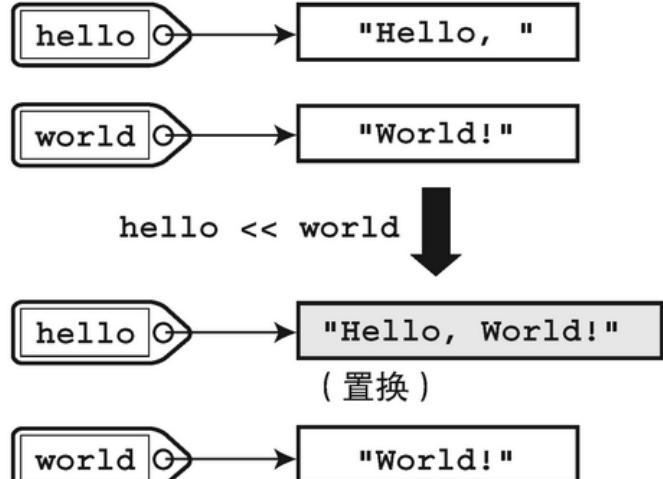


图 14.1 连接字符串

用 `+` 创建新的字符串。

```
hello = "Hello, "
world = "World!"

str = hello + world
p str    #=> "Hello, World!"
```

为原有字符串连接其他字符串时，可以使用 `<<` 或者 `concat` 方法。

```
hello = "Hello, "
world = "World!"

hello << world
p hello    #=> "Hello, World!"
hello.concat(world)
p hello    #=> "Hello, World!World!"
```

使用 `+` 也能连接原有字符串。

```
hello = hello + world
```

用 `+` 连接原有字符串的结果会被再次赋值给变量 `hello`，这与使用 `<<` 的结果是一样的。但用 `+` 连接后的字符串对象是新创建的，并没有改变原有对象，因此即使有其他变量与 `hello` 同时指向原来的对象，那些变量的值也不会改变。而另一方面，由于使用 `<<` 与 `concat` 方法时会改变原有的对象，因此就会对指向同一对象的其他变量产生影响。虽然一般情况下使用 `<<` 与 `concat` 方法会比较有效率，但是我们也应该根据实际情况来选择适当的字符串连接方法。

## 14.5 字符串的比较

我们可以使用 `==` 或者 `!=` 来判断字符串是否相同。例如在表达式 `str1 == str2` 中，`str1` 与 `str2` 为相同字符串时则返回 `true`，为不同字符串时则返回

false。!= 与 == 表示正相反的意思。

```
p "aaa" == "baa" #=> false
p "aaa" == "aa" #=> false
p "aaa" == "aaa" #=> true
p "aaa" != "baa" #=> true
p "aaa" != "aaa" #=> false
```

虽然判断字符串是否相同时使用 == 或者 != 会很方便，但判断是否为相似的字符串时，使用正则表达式则会简单得多。

#### 14.5.1 字符串的大小比较

字符串也有大小关系，但字符串的大小关系并不是由字符串的长度决定的。

```
p ("aaaaa" < "b") #=> true
```

字符串的大小由字符编码的顺序决定。英文字母按照“ABC”的顺序排列，日语的平假名与片假名按照“あいうえお”的顺序排列<sup>2</sup>，在排列英语或日语的字符串时，就可以使用该顺序。不过，Ruby 中日语的排序规则与字典中的顺序是不同的。例如对“かけ”、“かこ”、“がけ”这 3 个单词排序时，字典中的顺序是“かけ”、“がけ”、“かこ”，而在 Ruby 中，从小到大依次是“かけ”、“かこ”、“がけ”。

<sup>2</sup>即按日语 50 音的顺序排列。——译者注

另外，由于不能从汉字字符串得到汉字读音<sup>3</sup>，因此，如果想根据读音排列汉字，就需要事先准备好读音数据。

<sup>3</sup>这里特指日语中使用的汉字及其日语读音，与中文中使用的汉字以及拼音是不同的。——译者注

中文字符也同样是由字符编码的顺序决定的。例如，在 UTF-8 字符编码表中，“一”的编码为 U+4E00，“丁”的编码为 U+4E01，两个字符的比较结果如下。

```
p ("一" < "丁") #=> true
```

## 专栏

### 字符编码

计算机中的字符都是用数值来管理的，这样的数值也称为编码。

字符与数值的对应关系如下表所示。

字符	数值
A	65
B	66
C	67

我们把上表那样字符与数值一一对应的关系称为字符编码。但字符编码并不是一个正确的专业术语，因此我们需要小心使用。

ASCII 编码是计算机的基础。ASCII 编码是指把英文字母、数值、其他符号、以及换行、制表符这样的特殊字符集合起来，为它们分配 1 到 127 之间的数值，并使之占用 1 个字节空间（1 个字节可以表示 0 到 255）。另外，在欧美，一种名为 ISO-8859-1 的编码曾经被广泛使用，该编码包含了欧洲常用的基本字符（拼写符号、原音变音等），并为它们分配 128 到 255 之间的数值。也就是说，大部分的字符都是占用 1 个字节的空间。

使用日语时不可能不使用平假名、片假名或者汉字，但只用一个字节来表示这些字符是不可能的。因此，为了表示这些字符，用两个字节表示一个字符的技术就诞生了。

不过，非常可惜的是日语的字符编码并不只有 1 种，而是大概可以分为以下 4 种编码方式，并且相同编码方式得到的字符也不一定相等。

编码方式	主要使用的地方
Shift_JIS	Windows 文本
UTF-8	Unix 文本、HTML 等

EUC-JP	Unix 文本
ISO-2022-JP	电子邮件等

为字符分配与之对应的数值，这样的分配方式就称为字符编码方式（character encoding scheme）。在日本，常用的编码方式有 Shift\_JIS、EUC-JP、ISO-2022-JP 这 3 种。它们是日语标准字符编码 JIS X0208 的基础。字符编码的名称一般都直接使用编码方式的名称。例如我们会说“这个文本的字符编码是 EUC，在 Windows 中打开的时候要小心”。

编码方式不同的情况下，即使是相同的字符，对其分配的数值也会不一样。编码方式的不同，就是导致俗称为“乱码”的问题的原因之一<sup>4</sup>。

字符	Shift_JIS	EUC-JP	ISO-2022-JP	UTF-8
あ	82A0	A4A2	2422	E38182

在上表中，あ字符被分配的数值是用 16 进制表示的。据此可以看出，不同的编码方式所对应的数值是不一样的。像这样表示字符的值，我们称之为码位（code point）。在 Ruby 中可以用 `String#ord` 方法获取字符的码位。

```
#encoding: EUC-JP
p "あ".ord      #=> 42146(16 进制为A4A2)
```

另外，虽然 ISO-2022-JP 中使用了与 ASCII 相同的编码，但实际上也使用了一些小技巧以与 ASCII 有所区别（这些小技巧比较复杂，本专栏就不深入说明了）。

最近，一种名为 Unicode 的新的国际化字符编码慢慢地普及了起来。其中，UTF-8 就是 Unicode 的一种编码方式。

但 Unicode 并不能解决所有编码问题，它的产生也只是增加了一个不能互相转换的字符编码而已。而且，Unicode 中还有多种编码方式，很有可能导致在意想不到的地方出现问题。现在市面上有很多详细介绍字符编码的专业著作，也比较容易找到，有兴趣的读者可以自行查找阅读。不过，关于字符编码的理论一来比较深奥，二来存在各种各样的立场，其中也不乏偏见（笔者愚见）。因此，要想深入了解字符编码，建议大家不要只看某一本著作，而应该集各家之言，互相比较着阅读。

4 在计算机中用 GB 系列编码显示中文字符，最早的 GB 编码是 GB2312，接着有 GBK，现在最新的是 GB18030，每次扩展都完全保留之前版本的编码，因此每个新版本都可向下兼容。Windows 平台的中文字符使用是 GBK 编码，但在非 Windows 平台下的中文字符一般都使用 UTF-8 编码，因此在跨平台使用时要小心字符编码问题。——译者注

## 14.6 字符串的分割

用特定字符分割字符串时可以使用 `split` 方法。例如，用冒号（:）分割字符串的程序就可以像下面那样写：

```
column = str.split(/:/)
```

这样一来，分割后的各项字符串就会以数组的形式赋值给 `column`。

```
str = "高桥:gaoqiao:1234567:000-123-4567"
column = str.split(/:/)
p column
#=> ["高桥", "gaoqiao", "1234567", "000-123-4567"]
```

## 14.7 换行符的使用方法

用 `each_line` 等方法从标准输入读取字符串时，末尾肯定有换行符。然而，在实际处理字符串时，换行符有时候会很碍事。这种情况下，我们就需要删除多余的换行符（表 14.2）。

表 14.2 删除换行符的方法

	删除最后一个字符	删除换行符
非破坏性的	<code>chop</code>	<code>chomp</code>
破坏性的	<code>chop!</code>	<code>chomp!</code>

`chop` 方法与 `chop!` 方法会删除字符串行末的任何字符，`chomp` 方法与 `chomp!` 方法则只在行末为换行符时才将其删除。

```
str = "abcde"      # 没有换行符的情况
newstr = str.chop
p newstr      #=> "abcd"
newstr = str.chomp
p newstr      #=> "abcde"

str2 = "abcd\n"    # 有换行符的情况
newstr = str2.chop
p newstr      #=> "abcd"
newstr = str2.chomp
p newstr      #=> "abcd"
```

用 `each_line` 方法循环读取新的行时，一般会使用具有破坏性的 `chomp!` 方法直接删除换行符。

```
f.each_line do |line|
  line.chomp!
  处理 line
end
```

上面是 `chomp!` 的典型用法。另外，不同的运行环境下，换行符也不同，关于这方面的内容，请参考专栏《关于换行》。

## 14.8 字符串的检索与置换

字符串处理一般都离不开检索与置换。Ruby 可以很轻松地处理字符串。

### 14.8.1 字符串的检索

我们可以用 `index` 方法或者 `rindex` 方法，来检查指定的字符串是否存在在某字符串中。

`index` 方法会从左到右检查字符串中是否存在参数指定的字符串，而 `rindex` 方法则是按照从右到左的顺序来检查（`rindex` 的“r”表示的就是 right（右）的意思）。

```
str = "BBBBBBB"
p str.index("BB")      #=> 1
p str.rindex("BB")     #=> 5
```

找到字符串时，`index` 方法和 `rindex` 方法会返回字符串首个字符的索引值，没找到时则返回 `nil`。

另外，如果只是想知道字符串中是否有参数指定的字符串，用 `include?` 方法会更好。

```
str = "BBBBBBB"
p str.include?("BB")    #=> true
```

除了直接检索字符串外，Ruby 还可以使用正则表达式来检索。关于正则表达式，我们将在第 16 章中详细介绍。

### 专栏

#### 关于换行符

所谓换行符就是指进行换行的符号。就像在专栏《字符编码》中介绍的那样，计算机里使用的字符都被分配了 1 个与之对应的数值，同理，换行符也有 1 个与之对应的数值。不过麻烦的是，不同的 OS 对换行符的处理也不同。

下面是常用的 OS 的换行符。这里，LF（LineFeed）的字符为 `\n`，CR（Carriage Return）的字符为 `\r`。

OS 种类	换行符
Unix 系列	LF
Windows 系列	CR + LF
Mac OS 9 以前	CR

Ruby 中的标准换行符为 LF，一般在 `IO#each_line` 等方法中使用。也就是说，Ruby 在处理 Max OS 9 以前版本的文本时，可能会出现不能正确换行的情况。

我们可以通过参数指定 `each_line` 方法所使用的换行符，默认为 `each_line("\n")`。

## 14.8.2 字符串的置换

有时我们可能会需要用其他字符串来替换目标字符串中的某一部分。我们把这样的替换过程称为置换。用 `sub` 方法与 `gsub` 方法即可实现字符串的置换。

关于 `sub` 方法与 `gsub` 方法，我们将会在 16.6.1 节中详细介绍。

## 14.9 字符串与数组的共同方法

字符串中有很多与数组共同的方法。

当然，继承了 `Object` 类的实例的方法，在字符串（`String` 类的实例）以及数组（`Array` 类的实例）中也都能使用。除此以外，下面的方法也都能使用。

(a) 与索引操作相关的方法

(b) 与 `Enumerable` 模块相关的方法

(c) 与连接、反转（`reverse`）相关的方法

### 14.9.1 与索引操作相关的方法

在 14.3 节中我们提到过字符串也能像数组那样操作索引。数组适用的索引操作方法，也同样适用于字符串。

- `s[n] = str`
- `s[n..m] = str`
- `s[n, len] = str`

用 `str` 置换字符串 `s` 的一部分。这里 `n`、`m`、`len` 都是以字符为单位的。

```
str = "abcde"
str[2, 1] = "C"
p str      #=> "abCde"
```

- `s.slice!(n)`
- `s.slice!(n..m)`
- `s.slice!(n, len)`

删除字符串 `s` 的一部分，并返回删除的部分。

```
str = "Hello, Ruby."
p str.slice!(-1)    #=> "."
p str.slice!(5..6)  #=> ","
p str.slice!(0, 5)  #=> "Hello"
p str              #=> "Ruby"
```

### 14.9.2 返回 `Enumerator` 对象的方法

在处理字符串的方法中，有以行为单位进行循环处理的 `each_line` 方法、以字节为单位进行循环处理的 `each_byte` 方法、以及以字符为单位进行循环处理的 `each_char` 方法。调用这些方法时若不带块，则会直接返回 `Enumerator` 对象，因此，通过使用这些方法，我们就可以像下面的例子那样使用 `Enumerable` 模块的方法了。

```
# 用 collect 方法处理用 each_line 方法获取的行
str = "壹\n贰\n叁\n"
tmp = str.each_line.collect do |line|
  line.chomp 3
end
p tmp      #=> ["壹壹壹", "贰贰贰", "叁叁叁"]

# 用 collect 方法处理用 each_byte 方法获取的数值
str = "abcde"
tmp = str.each_byte.collect do |byte|
  -byte
end
p tmp      #=> [-97, -98, -99, -100, -101]
```

专栏

`Enumerator` 类

虽然 `Enumerable` 模块定义了很多方便的方法，但是作为模块中其他方法的基础，将遍历元素的方法限定为 `each` 方法，这一点有些不太灵活。

`String` 对象有 `each_byte`、`each_line`、`each_char` 等用于循环的方法，如果这些方法都能使用 `each_with_index`、`collect` 等 `Enumerable` 模块的方法的话，那就方便多了。而 `Enumerator` 类就是为了解决这个问题而诞生的。

`Enumerator` 类能以 `each` 方法以外的方法为基础，执行 `Enumerable` 模块定义的方法。使用 `Enumerator` 类后，我们就可以用 `String#each_line` 方法替代 `each` 方法，从而来执行 `Enumerable` 模块的方法了。

另外，不带块的情况下，大部分 Ruby 原生的迭代器在调用时都会返回 `Enumerator` 对象。因此，我们就可以对 `each_line`、`each_byte` 等方法的返回结果继续使用 `map` 等方法。

```
str = "AA\nBB\nCC\n"
p str.each_line.class      #=> Enumerator
p str.each_line.map{|line| line.chomp }
#=> ["AA", "BB", "CC"]
p str.each_byte.reject{|c| c == 0x0a }
#=> [65, 65, 66, 66, 67, 67]
```

### 14.9.3 与连接、反转（reverse）相关的方法

除了与 `Enumerable` 模块、索引等相关的办法外，字符串中还有一些与数组共同的方法。

- `s.concat(s2)`  
`s+s2`

与数组一样，字符串也能使用 `concat` 方法和 `+` 连接字符串。

```
s = "欢迎"
s.concat("光临")
p s      #=> "欢迎光临"
```

- `s.delete(str)`  
`s.delete!(str)`

从字符串 `s` 中删除字符串 `str`。

```
s = "防/止/检/索"
p s.delete("/")      #=> "防止检索"
```

- `s.reverse`  
`s.reverse!`

反转字符串 `s`。

```
s = "晚上好"
p s.reverse      #=> "好上晚"
```

## 14.10 其他方法

- `s.strip`  
`s.strip!`

这是删除字符串 `s` 开头和末尾的空白字符的方法。在不需要字符串开头和末尾的空白时，用这个方法非常方便。

```
p " Thank you. ".strip      #=> "Thank you."
```

- `s.upcase`  
`s.upcase!`  
`s.downcase`  
`s.downcase!`  
`s.swapcase`  
`s.swapcase!`  
`s.capitalize`  
`s.capitalize!`

所谓 `case` 在这里就是指英文字母的大、小写字母的意思。`~case` 方法就是转换字母大小写的方法。

`upcase` 方法会将小写字母转换为大写，大写字母保持不变。

```
p "Object-Oriented Language".upcase
#=> "OBJECT-ORIENTED LANGUAGE"
```

`downcase` 方法则刚好相反，将大写字母转换小写。

```
p "Object-Oriented Language".downcase  
#=> "object-oriented language"
```

`swapcase` 方法会将大写字母转换为小写，将小写字母转换为大写。

```
p "Object-Oriented Language".swapcase  
#=> "oBJECT-oRIENTED lANGUAGE"
```

`capitalize` 方法会将首字母转换为大写，将其余的字母转换为小写。

```
p "Object-Oriented Language".capitalize  
#=> "Object-oriented language"
```

## • `s.tr`

`s.tr!`

源自于 Unix 的 `tr` 命令的方法，用于置换字符。

该方法与 `gsub` 方法有点相似，不同点在于 `tr` 方法可以像 `s.tr("a-z", "A-Z")` 这样一次置换多个字符。

```
p "ABCDE".tr("B", "b")      #=> "AbCDE"  
p "ABCDE".tr("BD", "bd")    #=> "AbCdE"  
p "ABCDE".tr("A-E", "a-e")  #=> "abcde"
```

相反，`tr` 方法不能使用正则表达式，也不能指定两个字符以上的字符串。

## 14.11 日语字符编码的转换

字符编码转换有两种方法，分别是使用 `encode` 方法和使用 `nkf` 库的方法。

### 14.11.1 `encode` 方法

`encode` 方法是 Ruby 中基本的字符编码转换的方法。将字符编码由 EUC-JP 转换为 UTF-8，程序可以像下面这样写 5：

5简体中文的 GBK、GB2312 等编码也可以用同样的方法来转换。——译者注

```
# encoding: EUC-JP  
  
euc_str = "日语EUC 编码的字符串"  
utf8_str = euc_str.encode("utf-8")
```

另外，Ruby 中还定义了具有破坏性的 `encode!` 方法。

```
# encoding: EUC-JP  
  
str = "日语EUC 编码的字符串"  
str.encode!("utf-8")  # 将str 转换为UTF-8
```

`encode` 方法支持的字符编码，可通过 `Encoding.name_list` 方法获得。

### 14.11.2 `nkf` 库

使用 `encode` 方法可以进行字符编码的转换，但却不能进行半角假名与全角假名之间的转换。全半角假名的转换我们需要使用 `nkf` 库。

`nkf` 库由 `NKF` 模块提供。`NKF` 模块是 Unix 的 `nkf` (Network Kanji code conversion Filter) 过滤命令在 Ruby 中的实现。

`NKF.nkf` 用类似于命令行选项的字符串指定字符编码等。

`NKF.nkf( 选项字符串, 转换的字符串 )`

`nkf` 方法的主要参数如表 14.3 所示。

表 14.3 `nkf` 的主要参数

选项	意义
<code>-d</code>	从换行符中删除 CR

-c	往换行符中添加 CR
-x	不把半角假名转换为全角假名
-m0	抑制 MIME 处理
-h1	把片假名转换为平假名
-h2	把平假名转换为片假名
-h3	互换平假名与片假名
-z0	把 JIS X 0208 的数字转换为 ASCII
-z1	加上 -z0，把全角空格转换为半角空格
-z2	加上 -z0，把全角空格转换为两个半角空格
-e	输出的字符编码为 EUC-JP
-s	输出的字符编码为 Shift-JIS
-j	输出的字符编码为 ISO-2022-JP
-w	输出的字符编码为 UTF-8 (无 BOM)
-w8	输出的字符编码为 UTF-8 (有 BOM)
-w80	输出的字符编码为 UTF-8 (无 BOM)
-w16	输出的字符编码为 UTF-16 (Big Endian/ 无 BOM)
-w16B	输出的字符编码为 UTF-16 (Big Endian/ 有 BOM)
-w16B0	输出的字符编码为 UTF-16 (Big Endian/ 无 BOM)
-w16L	输出的字符编码为 UTF-16 (Little Endian/ 有 BOM)
-w16L0	输出的字符编码为 UTF-16 (Little Endian/ 无 BOM)
-E	输入字符编码为 EUC-JP

-S	输入字符编码为 Shift-JIS
-J	输入字符编码为 ISO-2022-JP
-W	输入的字符编码为 UTF-8 (无 BOM)
-W8	输入的字符编码为 UTF-8 (有 BOM)
-W80	输入的字符编码为 UTF-8 (无 BOM)
-W16	输入的字符编码为 UTF-16 (Big Endian/ 无 BOM)
-W16B	输入的字符编码为 UTF-16 (Big Endian/ 有 BOM)
-W16B0	输入的字符编码为 UTF-16 (Big Endian/ 无 BOM)
-W16L	输入的字符编码为 UTF-16 (Little Endian/ 有 BOM)
-W16L0	输入的字符编码为 UTF-16 (Little Endian/ 无 BOM)

为了避免半角假名转换为全角假名，或者因电子邮件的特殊字符处理而产生问题，如果只是单纯对字符进行编码转换的话，一般使用选项 `-x` 和 `-m0`（可以合并书写 `-xm0`）就足够了。

下面是把 EUC-JP 字符串转换为 UTF-8 的例子：

```
# encoding: EUC-JP
require "nkf"

euc_str = "日语EUC 编码的字符串"
utf8_str = NKF.nkf("-E -w -xm0", euc_str)
```

不指定输入字符编码时，`nkf` 库会自动判断其编码，基本上都可以按如下方式书写：

```
# encoding: EUC-JP
require "nkf"

euc_str = "日语EUC 编码的字符串"
utf8_str = NKF.nkf("-w -xm0", euc_str)
```

`NKF` 模块在 Ruby 字符串还未支持 `encoding` 功能以前就已经开始被使用了。大家可能会感觉选项的指定方法等与现在 Ruby 的风格有点格格不入，这是因为 `nkf` 库把其他命令的功能硬搬了过来的缘故。如果不涉及一些太特殊的处理，一般使用 `encode` 就足够了。

## 练习题

1. 有字符串 "Ruby is an object oriented programming language"，请创建一个数组，数组元素为这个字符串的各个单词。
2. 有将 1 的数组按照英文字母顺序进行排序。
3. 有不区分大小写，将 2 的数组按照英文字母顺序进行排序。
4. 有把 1 的字符串的全部单词的首字母转换为大写，转换后的结果为 "Ruby Is A n Object Oriented Programming Language"。
5. 有统计 1 的字符串中的字符及其数量，并按下面的形式输出结果（空格字符 6 个，'R' 1 个，'a' 4 个等等）。

```
' ': *****
'R': *
'a': ****
'b': **
```

```
'c': *
```

6. 定义方法 `han2num`，将汉字数字转换为 1 ~ 9999 的阿拉伯数字形式，例如将“七千一百二十三”转换为“7123”。

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

# 第 15 章 散列类

在本章将详细介绍散列（Hash）类。

- 复习散列

简略地介绍散列的相关用法。

- 散列的创建方法

介绍如何创建散列。

- 获取、设定键值

介绍批量获取键值的方法。

- 条件判断

介绍判断键值是否在散列中存在的方法。

- 查看大小

介绍查看散列大小的方法。

- 初始化

比较散列的初始化与新创建有何异同。

- 使用例子

以计算单词数量的程序为例，介绍散列的用法。

## 15.1 复习散列

在复习散列前，我们再次回顾一下数组的用法。

通过索引可以获取数组元素或对其赋值。

```
person = Array.new
person[0] = "田中一郎"
person[1] = "佐藤次郎"
person[2] = "木村三郎"
p person[1] #=> "佐藤次郎"
```

散列与数组一样，都是表示对象集合的对象。数组通过索引访问对象内的元素，而散列则是利用键。索引只能是数值，而键则可以是任意对象。通过使用键，散列就可以实现对元素的访问与赋值。

```
person = Hash.new
person["tanaka"] = "田中一郎"
person["satou"] = "佐藤次郎"
person["kimura"] = "木村三郎"
p person["satou"] #=> "佐藤次郎"
```

在本例中，`tanaka`、`satou` 等字符串就是键，对应的值为 `田中一郎`、`佐藤次郎`。散列中 `[]` 的用法也与数组非常相似。

## 15.2 散列的创建

与数组一样，创建散列的方法也有很多。其中下面两种是最常用到的。

### 15.2.1 使用 `{}`

使用字面量直接创建散列。

#### `{ 键 => 值 }`

像下面那样指定键值对，键值对之间用逗号（`,`）隔开。

```
h1 = {"a"=>"b", "c"=>"d"}
p h1["a"] #=> "b"
```

另外，用符号作为键时，

#### `{ 键: 值 }`

也可以采用上述定义方法。

```
h2 = {a: "b", c: "d"}  
p h2      #=> {:a=>"b", :c=>"d"}
```

### 15.2.2 使用 Hash.new

Hash.new是用来创建新的散列的方法。若指定参数，则该参数值为散列的默认值，也就是指定不存在的键时所返回的值。没指定参数时，散列的默认值为nil。

```
h1 = Hash.new  
h2 = Hash.new("")  
p h1["not_key"]    #=> nil  
p h2["not_key"]    #=> ""
```

散列的键可以使用各种对象，不过一般建议使用下面的对象作为散列的键。

- 字符串（String）
- 数值（Numeric）
- 符号（Symbol）
- 日期（Date）

更多详细内容请参考专栏《关于散列的键》。

## 15.3 值的获取与设定

与数组一样，散列也是用[]来实现与键相对应的元素值的获取与设定的。

```
h = Hash.new  
h["R"] = "Ruby"  
p h["R"]    #=> "Ruby"
```

另外，我们还可以用store方法设定值，用fetch方法获取值。下面的例子的执行结果与上面的例子是一样的。

```
h = Hash.new  
h.store("R", "Ruby")  
p h.fetch("R")    #=> "Ruby"
```

使用fetch方法时，有一点与[]不一样，就是如果散列中不存在指定的键，程序就会发生异常。

```
h = Hash.new  
p h.fetch("N")    #=> 错误 (IndexError)
```

如果对fetch方法指定第2个参数，那么该参数值就会作为键不存在时散列的默认值。

```
h = Hash.new  
h.store("R", "Ruby")  
p h.fetch("R", "(undef)")    #=> "Ruby"  
p h.fetch("N", "(undef)")    #=> "(undef)"
```

此外，fetch方法还可以使用块，此时块的执行结果为散列的默认值。

```
h = Hash.new  
p h.fetch("N"){ String.new }    #=> ""
```

### 15.3.1 一次性获取所有的键、值

我们可以一次性获取散列的键、值。由于散列是键值对形式的数据类型，因此获取键、值的方法是分开的。此外，我们还可以选择是逐个获取，还是以数组的形式一次性获取散列的所有键、值，不过这两种情况下使用的方法是不同的（表15.1）。

表15.1 获取散列的键与值的方法

	数组形式	迭代器形式
获取键	keys	each_key{  键   .....}

获取值	<code>values</code>	<code>each_value{  值   .....</code>
获取键值对[键, 值]	<code>to_a</code>	<code>each{  键 , 值   .....</code> <code>each{  数组   .....</code>

`keys` 与 `values` 方法各返回封装为数组后的散列的键与值。`to_a` 方法则会先按下面的形式把键值对封装为数组,

## [键, 值]

然后再将所有这些键值对数组封装为一个大数组返回。

```
h = {"a"=>"b", "c"=>"d"}
p h.keys    #=> ["a", "c"]
p h.values  #=> ["b", "d"]
p h.to_a    #=> [["a", "b"], ["c", "d"]]
```

除了返回数组外，我们还可以使用迭代器获取散列值。

使用 `each_key` 方法与 `each_value` 方法可以逐个获取并处理键、值。使用 `each` 方法还可以得到 [键, 值] 这样的键值对数组。

关于迭代器的例子请参考 15.8 节。

无论是使用 `each` 方法按顺序访问散列元素，还是使用 `to_a` 方法来获取全部的散列元素，这两种情况下都是可以按照散列键的设定顺序来获取元素的。

### 15.3.2 散列的默认值

下面我们来讨论一下散列的默认值（即指定散列中不存在的键时的返回值）。在获取散列值时，即使指定了不存在的键，程序也会返回某个值，而且不会因此而出错。我们有 3 种方法来指定这种情况下的返回值。

- 1. 创建散列时指定默认值

`Hash.new` 的参数值即为散列的默认值（什么都不指定时默认值为 `nil`）。

```
h = Hash.new(1)
h["a"] = 10
p h["a"]    #=> 10
p h["x"]    #=> 1
p h["y"]    #=> 1
```

这个方法与初始化数组一样，所有的键都共享这个默认值。

- 2. 通过块指定默认值

当希望不同的键采用不同的默认值时，或者不希望所有的键共享一个默认值时，我们可以使用 `Hash.new` 方法的块指定散列的默认值。

```
h = Hash.new do |hash, key|
  hash[key] = key.upcase
end
h["a"] = "b"
p h["a"]    #=> "b"
p h["x"]    #=> "X"
p h["y"]    #=> "Y"
```

块变量 `hash` 与 `key`，分别表示将要创建的散列以及散列当前的键。用这样的方法创建散列后，就只能在需要散列默认值的时候才会执行块。此外，如果不对散列进行赋值，通过指定相同的键也可以执行块。

- 3. 用 `fetch` 方法指定默认值

最后就是刚才已经介绍过的 `fetch` 方法。当 `Hash.new` 方法指定了默认值或块时，`fetch` 方法的第 2 个参数指定的默认值的优先级是最高的。

```
h = Hash.new do |hash, key|
  hash[key] = key.upcase
end
p h.fetch("x", "(undef)")    #=> "(undef)"
```

## 15.4 查看指定对象是否为散列的键或值

- `h.key?(key)`
- `h.has_key?(key)`

```
h.include?(key)  
h.member?(key)
```

上面 4 个方法都是查看指定对象是否为散列的键的方法，它们的用法和效果都是一样的。大家可以统一使用某一个，也可以根据不同的情况选择使用。

散列的键中包含指定对象时返回 `true`，否则则返回 `false`。

```
h = {"a" => "b", "c" => "d"}  
p h.key?("a")      #=> true  
p h.has_key?("a")  #=> true  
p h.include?("z")   #=> false  
p h.member?("z")    #=> false
```

- `h.value?(value)`

```
h.has_value?(value)
```

查看散列的值中是否存在指定对象的方法。这两个方法只是把 `key?`、`has_key?` 方法中代表键的 `key` 部分换成了值 `value`，用法是完全一样的。

散列的值中有指定对象时返回 `true`，否则则返回 `false`。

```
h = {"a"=>"b", "c"=>"d"}  
p h.value?("b")      #=> true  
p h.has_value?("z")  #=> false
```

## 15.5 查看散列的大小

- `h.size`

```
h.length
```

我们可以用 `length` 方法或者 `size` 方法来查看散列的大小，也就是散列键的数量。

```
h = {"a"=>"b", "c"=>"d"}  
p h.length      #=> 2  
p h.size        #=> 2
```

- `h.empty?`

我们可以用 `empty?` 方法来查看散列的大小是否为 0，也就是散列中是否存在任何键。

```
h = {"a"=>"b", "c"=>"d"}  
p h.empty?      #=> false  
h2 = Hash.new  
p h2.empty?    #=> true
```

## 15.6 删除键值

像数组一样，我们也可以成对地删除散列中的键值。

- `h.delete(key)`

通过键删除用 `delete` 方法。

```
h = {"R"=>"Ruby"}  
p h["R"]      #=> "Ruby"  
h.delete("R")  
p h["R"]      #=> nil
```

`delete` 方法也能使用块。指定块后，如果不存在键，则返回块的执行结果。

```
h = {"R"=>"Ruby"}  
p h.delete("P"){|key| "no #{key}."}    #=> "no P."
```

- `h.delete_if{|key, val| ... }`

```
h.reject!{|key, val| ... }
```

希望只删除符合某种条件的键值的时候，我们可以使用 `delete_if` 方法。

```
h = {"R"=>"Ruby", "P"=>"Perl"}  
p h.delete_if{|key, value| key == "P"}    #=> {"R"=>"Ruby"}
```

另外，虽然 `reject!` 方法的用法与 `delete_if` 方法相同，但当不符合删除条件时，两者的返回值却各异。

`delete_if` 方法会返回的是原来的散列，而 `reject!` 方法则返回的是 `nil`。

```
h = {"R"=>"Ruby", "P"=>"Perl"}  
p h.delete_if{|key, value| key == "L"}  
#=> {"R"=>"Ruby", "P"=>"Perl"}  
p h.reject!{|key, value| key == "L"} #=> nil
```

## 15.7 初始化散列

- `h.clear`

用 `clear` 方法清空使用过的散列。

```
h = {"a"=>"b", "c"=>"d"}  
h.clear  
p h.size #=> 0
```

这有点类似于使用下面的方法创建新的散列：

```
h = Hash.new
```

实际上，如果程序中只有一个地方引用 `h` 的话，两者的效果是一样的。不过如果还有其他地方引用 `h` 的话，那效果就不一样了。我们来对比一下下面两个例子（图 15.1）。

### 【例 1】

```
h = {"k1"=>"v1"}  
g = h  
h.clear  
p g #=> {}
```

### 【例 2】

```
h = {"k1"=>"v1"}  
g = h  
h = Hash.new  
p g #=> {"k1"=>"v1"}
```

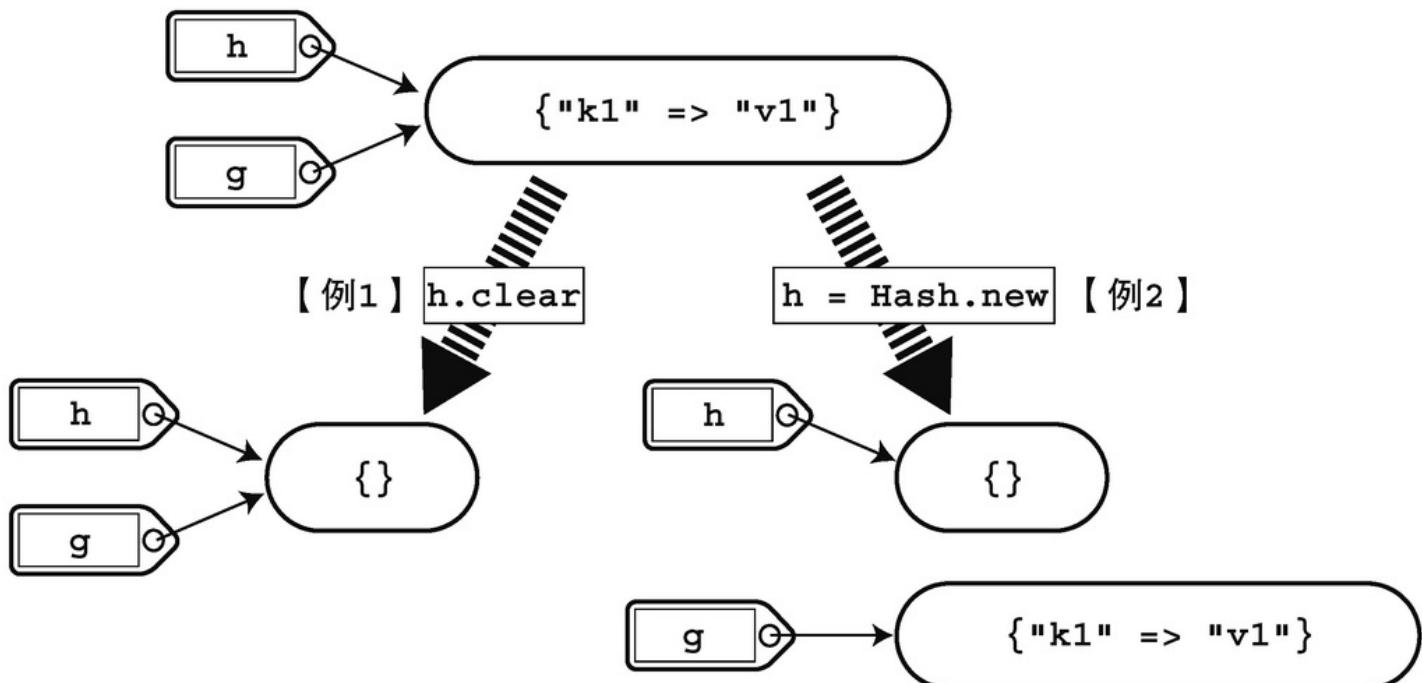


图 15.1 例 1 与例 2 的不同点

在例 1 中，`h.clear` 清空了 `h` 引用的散列，因此 `g` 引用的散列也被清空，`g` 与 `h` 还是引用同一个散列对象。

而在例 2 中，程序给 `h` 赋值了新的对象，但 `g` 还是引用原来的散列，也就是说，`g` 与 `h` 分别引用不同的散列对象。

需要注意的是，这里方法操作的不是变量，而是变量引用的对象。

散列的值也可以是散列，也就是所谓的“散列的散列”，这与数组中的“数组的数组”的用法是一样的。

```
table = {"A"=>{"a"=>"x", "b"=>"y"},  
        "B"=>{"a"=>"v", "b"=>"w"} }  
p table["A"]["a"] #=> "x"  
p table["B"]["a"] #=> "v"
```

在本例中，名为 `table` 的散列的值也是散列。因此，这里使用了 `["A"]["a"]` 这种两个键并列的形式来获取值。

## 15.8 应用示例：计算单词数量

下面我们用散列写个简单的小程序。在代码清单 15.1 中，程序会统计指定文件中的单词数量，并按出现次数由多到少的顺序将其显示出来。

代码清单 15.1 `word_count.rb`

```
1: # 计算单词数量  
2: count = Hash.new(0)  
3:  
4: ## 统计单词  
5: File.open(ARGV[0]) do |f|  
6:   f.each_line do |line|  
7:     words = line.split  
8:     words.each do |word|  
9:       count[word] += 1  
10:    end  
11:  end  
12: end  
13:  
14: ## 输出结果  
15: count.sort{|a, b|  
16:   a[1] <= b[1]  
17: }.each do |key, value|  
18:   print "#{key}: #{value}\n"  
19: end
```

首先，在程序第 2 行创建记录单词出现次数的散列 `count`。`count` 的键表示单词，值表示该单词出现的次数。如果键不存在，那么值应该为 0，因此将 `count` 的默认值设为 0。

在程序第 6 行到第 11 行的循环处理中，读取指定的文件，并以单词为单位分割文件，然后再统计各单词的数量。

在程序第 6 行，使用 `each_line` 方法读取每行数据，并赋值给变量 `line`。接下来，在程序第 7 行，使用 `split` 方法分割变量 `line`，将其转换为以单词为单位的数组，然后赋值给变量 `words`。

在程序第 8 行的循环处理中，对 `words` 使用 `each` 方法，逐个取出数组中的单词，然后将各单词作为键，从 `count` 中获取对应的出现次数，并做 +1 处理。

在程序第 15 行的循环处理中，输出统计完毕的出现次数。然后，在程序第 15 行到第 17 行，使用 `sort` 方法的块将单词按出现次数进行排序。

这里有两个关键点。一是使用了 `<=` 运算符进行排序，另外一点是比较对象使用了数组的第 2 个元素，如 `a[1]`、`b[1]`。

`<=` 运算符会比较左右两边的对象，检查判断它们的关系是 `<`、`=`、还是 `>`。`<` 时结果为负数，`=` 时结果为 0，`>` 时结果为正数。另外，之所以使用 `a[1]` 这样的数组，是因为用 `sort` 方法获取 `count` 的对象时，各个值会被作为数组提取出来，如下所示：

### [ 单词, 出现次数 ]

这样一来，`a[0]` 就表示单词本身，`a[1]` 才表示出现次数。因此，通过比较 `a[1]` 与 `b[1]`，就能实现按出现次数排序。

在程序第 17 行，`each` 方法会将排序后的散列元素逐个取出，然后再在程序第 18 行输出该单词与出现次数。

以上就是整个程序的执行流程，下面就让我们来实际执行一下这个程序，统计 Ruby 的 `README` 文件中各单词出现的次数。

### 执行示例

```
> ruby word_count.rb README  
=: 1  
What's: 1  
end:: 1  
rdoc: 1  
|  
you: 10  
of: 11  
Ruby: 1  
and: 13  
to: 22  
the: 23
```

根据这个结果我们可以看出，除符号之外，出现最多的单词是“the”，总共出现了 23 次。

## 专栏

### 关于散列的键

下面我们来讨论一下用数值或者自己定义的类等对象作为散列的键时需要注意的地方。在下面的例子中，我们首先尝试创建一个以数值为键的散列。

```
h = Hash.new
n1 = 1
n2 = 1.0
p n1==n2      #=> true
h[n1] = "exists."
p h[n1]      #=> "exists."
p h[n2]      #=> nil
```

用 `n1` 可以获取以 `n1` 为键保存的值，但是用与 `n1` 有相同的值的 `n2` 却无法获取。这是由于使用 `n2` 时，无法在散列中找到与之对应的值，因此就返回了默认值 `nil`。

在散列内部，程序会将散列获取值时指定的键，与将值保存到散列时指定的键做比较，判断两者是否一致。这时，判断键是否一致与键本身有着莫大的关系。具体来说，对于两个键 `key1`、`key2`，当 `key1.hash` 与 `key2.hash` 得到的整数值相同，且 `key1.eql?(key2)` 为 `true` 的时候，就会认为这两个键是一致的。

像本例那样，虽然使用 `==` 比较时得到的结果是一致的，但是，当两个键分别属于 `Fixnum` 类和 `Float` 类的对象时，由于不同类的对象不能判断为相同的键，因此就会产生与期待不同的结果。

## 练习题

1. 定义散列 `wday`，内容为星期的英文表达与中文表达的对应关系。

```
p wday[:sunday]    #=> "星期天"
p wday[:monday]   #=> "星期一"
p wday[:saturday] #=> "星期六"
```

2. 使用散列的方法，统计 1 的散列 `wday` 中键值对的数量。

3. 使用 `each` 方法和 1 的散列 `wday`，输出下面的字符串：

```
“sunday”是星期天。
“monday”是星期一。
|
```

4. 散列没有像数组的 `%w` 这样的语法。因此，请定义方法 `str2hash`，将被空格、制表符、换行符隔开的字符串转换为散列。

```
p str2hash("blue 蓝色 white 白色\nred 红色")
#=> {"blue"=>"蓝色", "white"=>"白色", "red"=>"红色"}
```

参考答案：请到图灵社区本书的“随书下载”处下载（<http://www.ituring.com.cn/book/1237>）。

# 第 16 章 正则表达式类

Ruby 的特点是“万物皆对象”，正则表达式也不例外。正则表达式对象所属的类就是接下来我们将要介绍的 `Regexp` 类。

- 正则表达式的写法与用法

介绍正则表达式的基础。

- 正则表达式的模式与匹配

介绍什么是正则表达式的元字符（meta character）、以及如何通过元字符进行匹配。

- 通过 `quote` 方法创建正则表达式

介绍一些不常用的正则表达式的创建方法。

- 正则表达式的选项

介绍几个能够用正则表达式设定的选项

- 捕获（capture）

介绍通过正则表达式匹配时的一个重要功能——捕获。

- 使用正则表达式的方法

介绍以正则表达式为参数的方法。

- 正则表达式的例子

介绍如何通过正则表达式匹配 URL。

## 16.1 关于正则表达式

下面我们开始介绍有关正则表达式的一些概念及用法。

### 16.1.1 正则表达式的写法与用法

正如我们在 2.3 节中介绍的那样，正则表达式描述的是一种“模式”，该模式被用于匹配字符串。一般情况下，我们把正则表达式模式的对象（`Regexp` 类对象）称为“正则表达式对象”，或直接称为“正则表达式”。

到目前为止，我们都是使用纯文本文字作为模式，而实际上还有更复杂的模式。例如，通过模式可以很简单地匹配“首字符为 A 到 D 中的某个字母，从第 2 个字符开始为数字”这样的字符串（这个模式可写为 `/[A-D]\d+/-/`）。

但模式也并非是万能的，例如像“与 Ruby 类似的字符串”这种含糊的模式就无法书写。模式说明的东西应该更具体一些，例如“以 R 开头，以 y 结尾，由 4 个字母组成”（这个模式可写为 `/R..y/-/`）。

为了能够熟练掌握正则表达式，首先就需要理解正则表达式模式的写法。因此，在学习正则表达式的具体用法前，本章将首先介绍一下正则表达式的写法，然后再介绍如何使用正则表达式。

### 16.1.2 正则表达式对象的创建方法

在程序中，通过用 `//` 将表示正则表达式模式的字符串括起来，就可以非常简单地创建出正则表达式。

另外，我们也可以使用类方法 `Regexp.new(str)` 来创建对象。当程序中已经定义了字符串对象 `str`，且希望根据这个字符串来创建正则表达式时，用这个方法会比较好。

```
re = Regexp.new("Ruby")
```

除上述两种方法外，与数组、字符串一样，我们也可以通过使用 `%` 的特殊语法来创建。正则表达式的情况下使用的是 `%r`，如果正则模式中包含 `/`，用这种方法会比较方便。语法如下所示：

```
%r (模式)
%r <模式>
%r |模式|
%r! 模式!
```

## 16.2 正则表达式的模式与匹配

了解正则表达式的创建方法后，接下来讨论一下模式。`=~` 方法是正则表达式中常用的方法，可以用来判断正则表达式与指定字符串是否匹配。

正则表达式 `=~` 字符串

无法匹配时返回 `nil`，匹配成功则返回该字符串起始字符的位置。

正如我们在第 5 章中介绍的那样，Ruby 会将 `nil` 与 `false` 解析为“假”，将除此以外的值解析为“真”，因此，如果要根据匹配结果执行不同的处理，则可以像下面这样写。

```
if 正则表达式 =~ 字符串
```

```
    匹配时的处理
```

```
else
```

```
    不匹配时的处理
```

```
end
```

我们还可以使用 `!~` 来颠倒“真”与“假”。

### 16.2.1 匹配普通字符

我们首先来看看如何通过模式进行简单的匹配（表 16.1）。当模式中只写有英文、数字时，正则表达式会单纯地根据目标字符串中是否包含该模式中的字符来判断是否匹配（在本章的所有表中，匹配部分都用▶匹配部分◀来表示）。

表 16.1 匹配普通字符的例子

模式	字符串	匹配部分
<code>/ABC/</code>	"ABC"	"▶ABC◀"
<code>/ABC/</code>	"ABCDEF"	"▶ABC◀DEF"
<code>/ABC/</code>	"123ABC"	"123▶ABC◀"
<code>/ABC/</code>	"A1B2C3"	(不匹配)
<code>/ABC/</code>	"AB"	(不匹配)
<code>/AB/</code>	"abc"	(不匹配)

### 16.2.2 匹配行首与行尾

在上一节的例子中，`/ABC/` 模式的情况下，只要是包含 `ABC` 的字符串就都可以匹配。但如果我们只想匹配 `ABC` 这一字符串，也就是说只匹配 `"ABC"`，而不匹配 `"012ABC"`、`"ABCDEF"` 等，这时的模式应该怎么写呢？这种情况下，我们可以使用模式 `/^ABC$/`。

`^`、`$` 是有特殊意义的字符。但它们并不用于匹配 `^` 与 `$` 字符。像这样的特殊字符，我们称之为元字符（meta character）。在稍后的章节中，我们会介绍 `^`、`$` 以外的其他元字符。

`^` 表示匹配行首，`$` 表示匹配行尾（表 16.2）。也就是说，模式 `/^ABC/` 匹配行首为 `ABC` 的字符串，模式 `/ABC$/` 匹配行尾为 `ABC` 的字符串。

表 16.2 `^` 与 `&` 的使用例子

模式	字符串	匹配部分
<code>/^ABC\$/</code>	"ABC"	"▶ABC◀"
<code>/^ABC\$/</code>	"ABCDEF"	(不匹配)
<code>/^ABC\$/</code>	"123ABC"	(不匹配)
<code>/^ABC/</code>	"ABC"	"▶ABC◀"

/^ABC/	"ABCDEF"	"▶ABC◀DEF"
/^ABC/	"123ABC"	不匹配
/ABC\$/	"ABC"	"▶ABC◀"
/ABC\$/	"ABCDEF"	(不匹配)
/ABC\$/	"123ABC"	"123▶ABC◀"

可能有人会觉得行首、行尾不是字符，“匹配行首”这样的说法比较别扭，不过用多了就会习惯了。

## 专栏

### 行首与行尾

^\\$ 分别匹配“行首”、“行尾”，而不是“字符串的开头”、“字符串末尾”。匹配字符串的开头用元字符 \A，匹配字符串的末尾用元字符 \Z。

这两种情况有什么不同呢？Ruby 的字符串，也就是 `String` 对象中，所谓的“行”就是用换行符 (\n) 间隔的字符串。因此模式 `/^ABC/` 也可以匹配字符串 "012\nABC"。也就是说，

```
012
ABC
```

像上面这种跨两行的字符串的情况下，由于第 2 行是以 ABC 开头的，因此也可以匹配。

那么为什么要将行首 / 行尾与字符串的开头 / 结尾分开定义呢？这是有历史原因的。

具体来说，原本正则表达式只能逐行匹配字符串，不能匹配多行字符串。因此就可以认为一个“字符串”就是一“行”。

但是，随着正则表达式的广泛使用，人们开始希望可以匹配多行字符串。而如果仍用 ^\$ 来匹配字符串的开头、结尾的话就很容易造成混乱，因此就另外定义了匹配字符串开头、结尾的元字符。

另外，还有一个与 \Z 类似的表现，就是 \z，不过两者的作用有点不一样。`\Z` 虽然也是匹配字符串末尾的元字符，但它有一个特点，就是如果字符串末尾是换行符，则匹配换行符前一个字符。

```
p "abc\n".gsub(/\z/, "!")
=> "abc\n!"
```

我们一般常用 \z，而很少使用 \Z。

### 16.2.3 指定匹配字符的范围

有时候我们会希望匹配“ABC 中的 1 个字符”。像这样，选择多个字符中的 1 个时，我们可以使用 []。

- [AB] .....A或B
- [ABC] .....A或B或C
- [CBA] .....同上（与[]中的顺序无关）
- [012ABC] .....0、1、2、A、B、C中的1个字符

不过，如果按照这样的写法，那么匹配“从 A 到 Z 的全部英文字母”时就麻烦了。这种情况下，我们可以在 [] 中使用 -，来表示一定范围内的字符串。

- [A-Z] .....从A到Z的全部大写英文字母
- [a-z] .....从a到z的全部小写英文字母
- [0-9] .....从0到9的全部数字
- [A-Za-z] .....从A到Z与从a到z的全部英文字母
- [A-Za-z\_] .....全部英文字母与 \_

备注 字符的范围也称为“字符类”。请注意这里的“类”与面向对象中的“类”的意义是不一样的。

如果 `[]` 是 `[]` 中首个或者最后 1 个字符，那么就只是单纯地表示 `[]` 字符。反过来说，如果 `[]` 表示的不是字符类，而是单纯的字符 `[]`，那么就必须写在模式的开头或者末尾。

- `[A-Za-z0-9_-]` .....全部英文字母、全部数字、`_`、`-`

在 `[]` 的开头使用 `^` 时，`^` 表示指定字符以外的字符。

- `[^ABC]` .....A、B、C 以外的字符
- `[^a-zA-Z]` .....a 到 z, A 到 Z (英文字母) 以外的字符

表 16.3 为一些实际进行匹配的例子。另外，在 1 个模式中还可以使用多个 `[]` (表 16.4)。

表 16.3 使用 `[]` 的例子

模式	字符串	匹配部分
<code>/[ABC]/</code>	<code>"B"</code>	<code>"► B ◀"</code>
<code>/[ABC]/</code>	<code>"BCD"</code>	<code>"► B ◀CD"</code>
<code>/[ABC]/</code>	<code>"123"</code>	(不匹配)
<code>/a[ABC]c/</code>	<code>"aBc"</code>	<code>"► aBc ◀"</code>
<code>/a[ABC]c/</code>	<code>"1aBcDe"</code>	<code>"1► aBc ◀De"</code>
<code>/a[ABC]c/</code>	<code>"abc"</code>	(不匹配)
<code>/[^ABC]/</code>	<code>"1"</code>	<code>"► 1 ◀"</code>
<code>/[^ABC]/</code>	<code>"A"</code>	(不匹配)
<code>/a[^ABC]c/</code>	<code>"aBcabc"</code>	<code>"aBc►abc◀"</code>

表 16.4 使用多个 `[]` 的例子

模式	字符串	匹配部分
<code>/[ABC][AB]/</code>	<code>"AB"</code>	<code>"► AB ◀"</code>
<code>/[ABC][AB]/</code>	<code>"AA"</code>	<code>"► AA ◀"</code>
<code>/[ABC][AB]/</code>	<code>"CA"</code>	<code>"► CA ◀"</code>
<code>/[ABC][AB]/</code>	<code>"CCCCA"</code>	<code>"CCC►CA ◀"</code>
<code>/[ABC][AB]/</code>	<code>"xCBx"</code>	<code>"x►CB◀x"</code>

/[ABC][AB]/	"CC"	(不匹配)
/[ABC][AB]/	"CxAx"	(不匹配)
/[ABC][AB]/	"C"	(不匹配)
/[0-9][A-Z]/	"0A"	"►0A◀"
/[0-9][A-Z]/	"000AAA"	"00►0A◀AA"
/[^A-Z][A-Z]/	"1A2B3C"	"►1A◀2B3C"
/[^0-9][^A-Z]/	"1A2B3C"	"1►A2◀B3C"

#### 16.2.4 匹配任意字符

有时候我们会希望定义这样的模式，即“不管是什么字符，只要匹配 1 个字符就行”。这种情况下，我们可以使用元字符 `.`。

- .....匹配任意字符

表 16.5 为实际进行匹配的例子。

模式	字符串	匹配部分
/A.C/	"ABC"	"►ABC◀"
/A.C/	"AxC"	"►AxC◀"
/A.C/	"012A3C456"	"012►A3C◀456"
/A.C/	"AC"	(不匹配)
/A.C/	"ABBC"	(不匹配)
/A.C/	"abc"	(不匹配)
/aaa.../	"00aaabcde"	"00►aaabcd◀e"
/aaa.../	"aaabb"	(不匹配)

然而，可能有读者会问：“程序在什么时候会需要能够匹配任意字符的字符呢”。的确，任意字符都能匹配的话，也就没有必要特意指定了。

在下面两种情况下，一般会使用这个元字符。

- 在希望指定字符数时使用

`/^...$/` 这样的模式可以匹配字符数为 3 的行。

- 与稍后介绍的元字符 `*` 配合使用

关于这部分的详细内容请参考 16.2.6 节。

#### 16.2.5 使用反斜杠的模式

与字符串一样，我们也可以使用 `\+1` 个英文字母这样的形式来表示换行、空白等特殊字符。

- `\s`

表示空白符，匹配空格（`0x20`）、制表符（Tab）、换行符、换页符（表 16.6）。

表 16.6 使用 `\s` 的例子

模式	字符串	匹配部分
----	-----	------

/ABC\sDEF/	"ABC DEF"	"►ABC DEF◀"
/ABC\sDEF/	"ABC\tDEF"	"►ABC\tDEF◀"
/ABC\sDEF/	"ABCDEF"	(不匹配)

- \d

匹配 0 到 9 的数字（表 16.7）。

表 16.7 使用 \d 的例子

模式	字符串	匹配部分
/\d\d\d-\d\d\d\d/	"012-3456"	"►012-3456◀"
/\d\d\d-\d\d\d\d/	"01234-012345"	"01►234-0123◀45"
/\d\d\d-\d\d\d\d/	"ABC-DEFG"	(不匹配)
/\d\d\d-\d\d\d\d/	"012-21"	(不匹配)

- \w

匹配英文字母与数字（表 16.8）。

表 16.8 使用 \w 的例子

模式	字符串	匹配部分
/\w\w\w/	"ABC"	"►ABC◀"
/\w\w\w/	"abc"	"►abc◀"
/\w\w\w/	"012"	"►012◀"
/\w\w\w/	"AB C"	(不匹配)
/\w\w\w/	"AB\nC"	(不匹配)

- \A

匹配字符串的开头（表 16.9）。

表 16.9 使用 \A 的例子

模式	字符串	匹配部分
/\AABC/	"ABC"	"►ABC◀"
/\AABC/	"ABCDEF"	"►ABC◀DEF"
/\AABC/	"012ABC"	(不匹配)
/\AABC/	"012\nABC"	(不匹配)

- \z

匹配字符串的末尾（表 16.10）。

表 16.10 使用 \z 的例子

模式	字符串	匹配部分
/ABC\z/	"ABC"	"►ABC◀"
/ABC\z/	"012ABC"	"012►ABC◀"
/ABC\z/	"ABCDEF"	(不匹配)
/ABC\z/	"012\nABC"	"012\n►ABC◀"

/ABC\z/

"ABC\nDEF"

(不匹配)

## • 元字符转义

我们还可以用 \ 对元字符进行转义。在 \ 后添加 ^、\$、[ 等非字母数字的元字符后，该元字符就不再发挥元字符的功能，而是直接被作为元字符本身来匹配（表 16.11）。

表 16.11 使用 \ 的例子

模式	字符串	匹配部分
/ABC\[ /	"ABC[ "	"► ABC[ ◀"
/\^ABC/	"ABC"	(不匹配)
/\^ABC/	"012^ABC"	"012►^ABC◀"

## 16.2.6 重复

有时候，我们会需要重复匹配多次相同的字符。例如，匹配 "Subject:" 字符串后多个空白符，空白符后又有字符串这样的行”（这是匹配电子邮件的主题时使用的模式）。

正则表达式中用以下元字符来表示重复匹配的模式。

- \* .....重复 0 次以上
- + .....重复 1 次以上
- ? .....重复 0 次或 1 次

首先我们来看看表示匹配重复 0 次以上的 \*（表 16.12）。

表 16.12 使用 \* 的例子

模式	字符串	匹配部分
/A*/	"A"	"► A ◀"
/A*/	"AAAAAA"	"► AAAAAA ◀"
/A*/	" "	"► ◀"
/A*/	"BBB"	"► ◀" BBB
/A*C/	"AAAC"	"► AAAC ◀"
/A*C/	"BC"	"B ► C ◀"
/A*C/	"AAAB"	(不匹配)
/AAA*C/	"AAC"	"► AAC ◀"
/AAA*C/	"AC"	(不匹配)
/A.*C/	"AB012C"	"► AB012C ◀"

/A.*C/	"AB CD"	"►AB C◀D"
/A.*C/	"ACDE"	"►AC◀DE"

使用 \* 匹配电子邮件的主题的模式如表 16.13 所示。

表 16.13 使用 \* 的例子（之二）

模式	字符串	匹配部分
/^Subject:\s*.*\$/	"Subject: foo"	"►Subject: foo◀"
/^Subject:\s*.*\$/	"Subject: Re: foo"	"►Subject: Re: foo◀"
/^Subject:\s*.*\$/	"Subject:Re^2 foo"	"►Subject:Re^2 foo◀"
/^Subject:\s*.*\$/	"in Subject:Re foo"	(不匹配)

+ 表示匹配重复 1 次以上（表 16.14）。

表 16.14 使用 + 的例子

模式	字符串	匹配部分
/A+/"	"A"	"►A◀"
/A+/"	"AAAAAA"	"►AAAAAA◀"
/A+/"	" "	(不匹配)
/A+/"	"BBB"	(不匹配)
/A+C/	"AAAC"	"►AAAC◀"
/A+C/	"BC"	(不匹配)
/A+C/	"AAAB"	(不匹配)
/AAA+C/	"AAC"	(不匹配)
/AAA+C/	"AC"	(不匹配)
/A.+C/	"AB012C"	"►AB012C◀"
/A.+C/	"AB CD"	"►AB C◀D"

/A.+C/	"ACDE"	(不匹配)
--------	--------	-------

? 表示匹配重复 0 次或 1 次（表 16.15）。

表 16.15 使用 ? 的例子

模式	字符串	匹配部分
/^A?\$/	"A"	"► A ◀"
/^A?\$/	" "	"► ◀"
/^A?\$/	"AAAAAA"	(不匹配)
/^A?C/	"AC"	"► AC ◀"
/^A?C/	"AAAC"	(不匹配)
/^A?C/	"BC"	(不匹配)
/^A?C/	"C"	"► C ◀"
/AAA?C/	"AAAC"	"► AAAC ◀"
/AAA?C/	"AAC"	"► AAC ◀"
/AAA?C/	"AC"	(不匹配)
/A.?C/	"ACDE"	"► AC ◀ DE"
/A.?C/	"ABCDE"	"► ABC ◀ DE"
/A.?C/	"AB012C"	(不匹配)
/A.?C/	"AB_CD"	(不匹配)

### 16.2.7 最短匹配

匹配 0 次以上的 \* 以及匹配 1 次以上的 + 会匹配尽可能多的字符 1。相反，匹配尽可能少的字符 2 时（重复后的模式首次出现的位置之前的部分），我们可以用以下元字符：

1也称贪婪匹配。——译者注

2也称懒惰匹配。——译者注

- \*? .....0 次以上的重复中最短的部分

- +? .....1 次以上的重复中最短的部分

下表（16.16）是与不带 ? 时的 \* 与 + 的对比。

表 16.16 使用 \*、+、\*?、+? 的例子

模式	字符串	匹配部分
/A.*B/	"ABCDABCDA <sub>B</sub> C"	"► ABCDABCDA <sub>B</sub> C ◀ CD"
/A.*C/	"ABCDABCDA <sub>B</sub> C"	"► ABCDABCDA <sub>B</sub> C ◀ D"
/A.*?B/	"ABCDABCDA <sub>B</sub> C"	"► AB ◀ CDABCDA <sub>B</sub> C"
/A.*?C/	"ABCDABCDA <sub>B</sub> C"	"► ABC ◀ DABCDA <sub>B</sub> C"
/A.+B/	"ABCDABCDA <sub>B</sub> C"	"► ABCDABCDA <sub>B</sub> C ◀ CD"
/A.+C/	"ABCDABCDA <sub>B</sub> C"	"► ABCDABCDA <sub>B</sub> C ◀ D"
/A.+?B/	"ABCDABCDA <sub>B</sub> C"	"► ABCDAB ◀ CDABCDA <sub>B</sub> C"
/A.+?C/	"ABCDABCDA <sub>B</sub> C"	"► ABC ◀ DABCDA <sub>B</sub> C"

### 16.2.8 () 与重复

在刚才的例子中，我们只是重复匹配了 1 个字符，而通过使用 ()，我们还可以重复匹配多个字符（表 16.17）。

表 16.17 使用 () 的例子

模式	字符串	匹配部分
/^(ABC)*\$/	"ABC"	"► ABC ◀ "
/^(ABC)*\$/	" "	"► ◀ "
/^(ABC)*\$/	"ABCABC"	"► ABCABC ◀ "
/^(ABC)*\$/	"ABCABCAB"	(不匹配)
/^(ABC)+\$/	"ABC"	"► ABC ◀ "
/^(ABC)+\$/	" "	(不匹配)
/^(ABC)+\$/	"ABCABC"	"► ABCABC ◀ "
/^(ABC)+\$/	"ABCABCAB"	(不匹配)

/^(ABC)?\$/	"ABC"	"► ABC ◀"
/^(ABC)?\$/	" "	"► ◀"
/^(ABC)?\$/	"ABCABC"	(不匹配)
/^(ABC)?\$/	"ABCABCAB"	(不匹配)

### 16.2.9 选择

我们可以用 `|` 在几个候补模式中匹配任意一个（表 16.18）。

表 16.18 使用 `|` 的例子

模式	字符串	匹配部分
/^(ABC DEF)\$/	"ABC"	"► ABC ◀"
/^(ABC DEF)\$/	"DEF"	"► DEF ◀"
/^(ABC DEF)\$/	"AB"	(不匹配)
/^(ABC DEF)\$/	"ABCDEF"	(不匹配)
/^(AB CD)+\$/	"ABCD"	"► ABCD ◀"
/^(AB CD)+\$/	" "	(不匹配)
/^(AB CD)+\$/	"ABCABC"	(不匹配)
/^(AB CD)+\$/	"ABCABCAB"	(不匹配)

### 16.3 使用 `quote` 方法的正则表达式

有时候我们可能会希望转义（escape）正则表达式中的所有元字符。而 `quote` 方法就可以帮我们实现这个想法。`quote` 方法会返回转义了元字符后的正则表达式字符串，然后再结合 `new` 方法，就可以生成新的正则表达式对象了。

```
re1 = Regexp.new("abc*def")
re2 = Regexp.new(Regexp.quote("abc*def"))
p (re1 =~ "abc*def") #=> nil
p (re2 =~ "abc*def") #=> 0
```

`quote` 方法的问题在于不能以元字符的格式写元字符。因此，在写一些复杂的正则表达式时，建议不要使用 `quote` 方法，而是乖乖地对元字符进行转义。

### 16.4 正则表达式的选项

正则表达式中还有选项，使用选项可以改变正则表达式的一些默认效果。

设定正则表达式的选项时，只需在 `/.../` 的后面指定即可，如 `/.../im`，这里的 `i` 以及 `m` 就是正则表达式的选项。

忽略英文字母大小写的选项。指定这个选项后，无论字符串中的字母是大写还是小写都会被匹配。

• **x**

忽略正则表达式中的空白字符以及 # 后面的字符的选项。指定这个选项后，我们就可以使用 # 在正则表达式中写注释了。

• **m**

指定这个选项后，就可以使用 \n 匹配换行符了。

```
str = "ABC\nDEF\nGHI"  
p /DEF.GHI/ =~ str      #=> nil  
p /DEF.GHI/m =~ str    #=> 4
```

表 16.19 中总结了几种常用的选项。

表 16.19 正则表达式的选项

选项	选项常量	意义
i	Regexp::IGNORECASE	不区分大小写
x	Regexp::EXTENDED	忽略模式中的空白字符
m	Regexp::MULTILINE	匹配多行
o	(无)	只使用一次内嵌表达式

Regexp.new 方法中的第 2 个参数可用于指定选项常量。只需要 1 个参数时，可不指定第 2 个参数或者直接指定 nil。

例如，/Ruby 脚本 /i 这一正则表达式，可以像下面那样写：

```
Regexp.new("Ruby 脚本", Regexp::IGNORECASE)
```

另外，我们还可以用 | 指定多个选项。这时，/Ruby 脚本 /im 这一正则表达式就变成了下面这样：

```
Regexp.new("Ruby 脚本",  
          Regexp::IGNORECASE | Regexp::MULTILINE)
```

## 16.5 捕获

除了检查字符是否匹配外，正则表达式还有另外一个常用功能，甚至可以说是比匹配更加重要的功能——捕获（后向引用）。

所谓捕获，就是从正则表达式的匹配部分中提取其中的某部分。通过“\$ 数字”这种形式的变量，就可以获取匹配了正则表达式中的用 () 括住的部分的字符串。

```
/(.)(.)(.)/ =~ "abc"  
first = $1  
second = $2  
third = $3  
p first    #=> "a"  
p second   #=> "b"  
p third    #=> "c"
```

在进行匹配的时候，我们只知道是否匹配、匹配第几个字符之类的信息。而使用捕获后，我们就可以知道哪部分被匹配了。因此，通过这个功能，我们就可以非常方便地对字符串进行分析。

在 16.2.8 节中我们提到了 () 也被用于将多个模式整理为一个。在修改程序中的正则表达式时，如果改变了 () 的数量，那么将要引用的部分的索引也会随之改变，有时就会带来不便。这种情况下，我们可以使用 (?:) 过滤不需要捕获的模式。

```
/(.)(\d\d)+(.)/ =~ "123456"  
p $1    #=> "1"  
p $2    #=> "45"  
p $3    #=> "6"
```

```
/(..)(?:\d\d)+(.)/ =~ "123456"
p $1      #=> "1"
p $2      #=> "6"
```

除了“\$ 数字”这种形式以外，保存匹配结果的变量还有 \$、\$&、\$，分表代表匹配部分前的字符串、匹配部分的字符串、匹配部分后的字符串。为了方便大家快速理解这 3 个变量的含义，我们来看看下面这个例子：

```
/C./ =~ "ABCDEF"
p $`      #=> "AB"
p $&      #=> "CD"
p $'      #=> "EF"
```

这样一来，我们就可以将字符串整体分为匹配部分与非匹配部分，并将其分别保存在 3 个不同的变量中。

## 16.6 使用正则表达式的方法

字符串相关的方法中有一些使用了正则表达式，接下来我们就来介绍一下其中的 `sub` 方法、`gsub` 方法、`scan` 方法。

### 16.6.1 sub 方法与 gsub 方法

`sub` 方法与 `gsub` 方法的作用是用指定的字符置换字符串中的某部分字符。

`sub` 方法与 `gsub` 方法都有两个参数。第 1 个参数用于指定希望匹配的正则表达式的模式，第 2 个参数用于指定与匹配部分置换的字符。`sub` 方法只置换首次匹配的部分，而 `gsub` 方法则会置换所有匹配的部分。

```
str = "abc  def  g  hi"
p str.sub(/\s+/, ' ')
p str.gsub(/\s+/, ' ')
```

`\s+/` 是用于匹配 1 个以上的空白字符的模式。因此在本例中，`sub` 方法与 `gsub` 方法会将匹配的空白部分置换为 1 个空白。`sub` 方法只会置换 `abc` 与 `def` 间的空白，而 `gsub` 方法则会将字符串后面匹配的空白部分全部置换。

`sub` 方法与 `gsub` 方法还可以使用块。这时，程序会将字符串中匹配的部分传递给块，并在块中使用该字符串进行处理。这样一来，块中返回的字符串就会置换字符串中匹配的部分。

```
str = "abracatabra"
nstr = str.sub(/.a/) do |matched|
  '<' + matched.upcase + '>'
end
p nstr    #=> "ab<RA>catabra"

nstr = str.gsub(/.a/) do |matched|
  '<' + matched.upcase + '>'
end
p nstr    #=> "ab<RA><CA><TA>b<RA>"
```

在本例中，程序会将字符串 `a` 以及 `a` 之前的字母转换为大写，并用 `<>` 将其括起来。

`sub` 方法与 `gsub` 方法也有带 `!` 的方法。`sub!` 方法与 `gsub!` 方法会直接将作为接受者的对象变换为置换后的字符串。

### 16.6.2 scan 方法

`scan` 方法能像 `gsub` 方法那样获取匹配部分的字符，但不能做置换操作。因此，当需要对匹配部分做某种处理时，可以使用该方法（代码清单 16.1）。

代码清单 16.1 scan1.rb

```
"abracatabra".scan(/.a/) do |matched|
  p matched
end
```

#### 执行示例

```
> ruby scan1.rb
"ra"
"ca"
"ta"
"ra"
```

在正则表达式中使用 `()` 时，匹配部分会以数组的形式返回（代码清单 16.2）。

代码清单 16.2 scan2.rb

```
"abracatabra".scan(/(.)\a/) do |matched|
  p matched
end
```

#### 执行示例

```
> ruby scan2.rb
["r", "a"]
["c", "a"]
["t", "a"]
["r", "a"]
```

另外，如果指定与 `()` 相等数量的块参数，则返回的结果就不是数组，而是各个元素（代码清单 16.3）。

#### 代码清单 16.3 scan3.rb

```
"abracatabra".scan(/(.)\a/) do |a, b|
  p a+"-"+b
end
```

#### 执行示例

```
> ruby scan3.rb
"r-a"
"c-a"
"t-a"
"r-a"
```

如果没有指定块，则直接返回匹配的字符串数组。

```
p "abracatabra".scan(/\.\a/)    #=> ["ra", "ca", "ta", "ra"]
```

## 16.7 正则表达式例子

接下来我们来看看用正则表达式匹配 URL 的例子。

首先我们需要“找出包含 URL 的行”。创建表示完整的 URL 的正则表达式会非常复杂，不过我们可以稍微变通一下，把目标改为“找出类似于 URL 的字符串”，这时，就可以用如下模式来进行匹配。

```
/http:\//\//
```

这个匹配模式的好处在于便于操作，而且也的确可以匹配 URL。

在此基础上，我们还可以进一步写出“获取类似于 URL 的字符串中的某部分”的正则表达式。例如，获取 HTTP 的 URL 中的服务器地址的模式时，可以像下面这样书写。

```
/http:\//([^\/*])\//
```

`[^\/*]*` 表示匹配不含 / 的连续字符串。

上述例子中使用了较多 /，不便于阅读，这种情况下我们可以使用 `%r` 将其改写成像下面那样：

```
%r|http://([^\/*])/|
```

现在我们就来看看这样写是否可以匹配（代码清单 16.4）。

#### 代码清单 16.4 url\_match.rb

```
str = "http://www.ruby-lang.org/ja/"
%r|http://([^\/*])/| =~ str
print "server address: ", $1, "\n"
```

#### 执行示例

```
> ruby url_match.rb
server address: www.ruby-lang.org
```

可以发现，的确可以获取服务器地址。

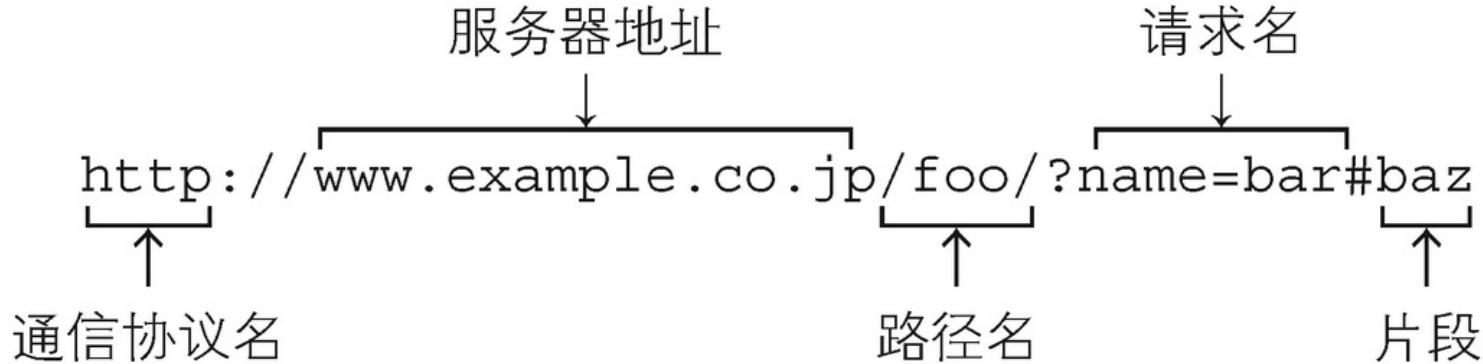
然后，我们再看看获取服务器地址以外部分的正则表达式。

```
%r|^(([^:/?#]+):)?(//([^\?#]*))?([^\?#]*)(\?( [^\#]*))?(#.*)?)|
```

这是在 RFC2396“Uniform Resource Identifiers(URI)”这个定义 URI 语法的文件中使用的正则表达式。

这个正则表达式可以被原封不动地用在 Ruby 中。如果用这个正则表达式进行匹配，则 HTTP 等协议名会被保存在 \$2 中，服务器地址等会被保存在 \$4 中，路径名会被保存在 \$5 中，请求部分会被保存在 \$7 中，片段 (fragment) 会被保存在 \$9 中。

例如，<http://www.example.co.jp/foo/?name=bar#bar> 这个 URI 的情况下，http 为通信协议名，www.example.co.jp 为服务器地址，/foo/ 为路径名，name=bar 为请求名，baz 为片段。



然而，写到这种程度，正则表达式已经变得非常复杂了。如果把正则表达式写成在任何情况下都能匹配的万能模式，就会使得正则表达式变得难以读懂，增加程序的维护成本。相比之下，只满足当前需求的正确易懂的正则表达式则往往更有效率。

例如，匹配邮政编号的正则表达式，可以写成下面这样：

```
/\d\d\d-\d\d\d\d/
```

这样，就不会匹配只有 3 位数字，或者没有 - 的邮政编码了。

在不需要太过严格的输入检查时，直接用 `/\d+-?\d*/` 匹配就可以了。

**备注** 想进一步了解正则表达式的读者可以参考 Jeffrey E.F.Friedl 著的 *Mastering Regular Expressions*。该书系统地介绍了正则表达式，而且评价也非常高。

## 练习题

- 电子邮件的地址格式为账号名 @ 域名。请写出一个正则表达式，将账号名保存在 \$1 中，域名保存在 \$2 中。
- 利用 gsub 方法，将字符串“正则表达式真难啊，怎么这么难懂！”置换为“正则表达式真简单啊，怎么这么易懂！”
- 定义方法 `word_capitalize`，当被指定的参数为用连字符 (hyphen) 连接的英文字母字符串时，对被连字符分割的部分做 Capitalize 处理（即单词的首字母大写，其余小写）。

```
p word_capitalize("in-reply-to")    #=> "In-Reply-To"  
p word_capitalize("X-MAILER")      #=> "X-Mailer"
```

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

# 第 17 章 IO 类

到目前为止出现的程序中有一些是关于处理文件中的数据的。IO 类的主要作用就是让程序与外部进行数据的输入（input）/ 输出（output）操作。在本章中，我们将会讨论以下内容。

- 输入/输出的种类

介绍 IO 类支持的输入 / 输出对象。

- 基本的输入/输出操作

介绍 IO 类的基本操作，以行为单位或者以大小为单位进行读写操作。

- 文件指针、二进制模式 / 文本模式、缓冲处理

深入介绍一些与输入 / 输出有关的细节。二进制模式与文本模式是 Windows 特有的规范，若不注意常常会出现问题。

- 与命令进行交互

介绍与外部命令进行数据交换的方法

## 17.1 输入 / 输出的种类

我们首先来了解一下输入 / 输出的对象到底是什么。

### 17.1.1 标准输入 / 输出

程序在启动后会预先分配 3 个 IO 对象。

- 标准输入

标准输入可以获取从键盘输入的内容。通过预定义常量 `STDIN` 可调用操作标准输入的 IO 对象。另外，用全局变量 `$stdin` 也可以引用标准输入的 IO 对象。不指定接收者的 `gets` 方法等都会默认从标准输入中获取数据。

- 标准输出

向标准输出写入的数据会显示在屏幕上。通过预定义常量 `STDOUT` 可调用操作标准输出的 IO 对象。另外，用全局变量 `$stdout` 也可以引用标准输出的 IO 对象。不指定接收者的 `puts`、`print`、`printf` 等方法会默认将数据写入到标准输出。

- 标准错误输出

向标准错误输出写入的数据会显示在屏幕上。通过预定义常量 `STDERR` 可调用操作标准错误输出的 IO 对象。另外，用全局变量 `$stderr` 也可以引用标准错误输出的 IO 对象。

标准错误输出原本是用于输出错误信息的，但实际上，除输出警告或者错误之外，在希望与程序正常输出的信息做出区别时也可以使用它（代码清单 17.1）。

### 代码清单 17.1 out.rb

```
$stdout.print "Output to $stdout.\n" # 标准输出
$stderr.print "Output to $stderr.\n" # 标准错误输出
```

#### 执行示例

```
> ruby out.rb
Output to $stdout.
Output to $stderr.
```

将输出结果重定向到文件时，标准输出的内容会被写入到文件，只有标准错误输出的内容被输出到屏幕中。

#### 执行示例

```
> ruby out.rb > log.txt
Output to $stderr.
```

**备注** 在执行程序时，在命令后加上 `>` 文件名，就可以将程序执行时的输出结果保存到文件中。我们把这个控制台的功能称为“重定向”。通过这个功能，不仅 ruby 命令，程序的输出内容也都可以保存在文件中。

根据需要灵活使用标准输出与标准错误输出，可以使我们很方便地分开获取正常信息与错误信息。

**备注** ruby 命令的错误信息也会被输出到标准错误输出。

通常标准输入、标准输出、标准错误输出都是与控制台关联的。但是将命令的输出重定向到文件，或者使用管道（pipe）将结果传递给其他程序时则与控制台没有关系。根据实际的使用情况，程序的输入 / 输出状态也各异。`IO` 对象是否与控制台关联，我们可以通过 `tty?` 方法判断。

代码清单 17.2 是一个检查标准输入是否为屏幕的例子。

#### 代码清单 17.2 `tty.rb`

```
if $stdin.tty?  
  print "Stdin is a TTY.\n"  
else  
  print "Stdin is not a TTY.\n"  
end
```

下面我们用不同的方式调用这个程序，看看有何不同。首先是普通调用。

#### 执行示例

```
> ruby tty.rb  
Stdin is a TTY.
```

将命令的输出结果传给管道，或者通过文件输入内容时，程序的结果会不一样。

#### 执行示例

```
> echo | ruby tty.rb  
Stdin is not a TTY.  
> ruby tty.rb < data.txt  
Stdin is not a TTY.
```

备注 TTY 是 TeleTYpe 的缩写。

### 17.1.2 文件输入 / 输出

通过 `IO` 类的子类 `File` 类可以进行文件的输入 / 输出处理。`File` 类中封装了文件删除、文件属性变更等文件专用的功能，而一些基本的输入 / 输出处理则使用继承自 `IO` 类的方法。

- `io = File.open(file, mode)`  
`io = open(file, mode)`

通过 `File.open` 方法或 `open` 方法打开文件并获取新的 `IO` 对象。

模式（mode）会指定以何种目的打开文件（表 17.1）。缺省模式为只读模式（"r"）。在 Windows 环境下，在各模式后加上 `b`、通过 "`rb`"、"`rb+`" 等这样的形式即可表示二进制模式（后述）。

表 17.1 模式

模式	意义
r	用只读模式打开文件。
r+	用读写模式打开文件。
w	用只写模式打开文件。文件不存在则创建新的文件；文件已存在则清空文件，即将文件大小设置为0。
w+	读写模式，其余同 "w"。
a	用追加模式打开文件。文件不存在则创建新的文件。
a+	用读取/追加模式打开文件。文件不存在则创建新的文件。

- `io.close`

使用 `close` 方法关闭已打开的文件。

1 个程序中同时打开文件的数量是有限制的，因此使用完的文件应该尽快关闭。如果打开多个文件而不进行关闭操作，程序就很可能会在使用 `open` 方法时突然产生异常。

`File.open` 方法如果使用块，则文件会在使用完毕后自动关闭。这种情况下，`IO` 对象会被作为块变量传递给块。块执行完毕后，块变量引用的 `IO` 对象也会自动关闭。这种写法会使输入 / 输出的操作范围更加清晰。

```
File.open("foo.txt") do |io|  
  while line = io.gets  
    |
```

```
end  
end
```

- **io.close?**

用 `close?` 方法可以检查 `io` 对象是否关闭了。

```
io = File.open("foo.txt")  
io.close  
p io.closed?    #=> true
```

- **File.read(file)**

使用类方法 `read` 可以一次性读取文件 `file` 的内容。

```
data = File.read("foo.txt")
```

**备注** 在 Windows 中不能使用 `File.read` 方法读取像图像数据等二进制数据。`File.read` 方法使用文本模式打开文件时，会对换行符等进行转换，因此无法得到正确的结果。详细内容请参考 17.4 节。

## 17.2 基本的输入 / 输出操作

输入 / 输出操作的数据为字符串，也就是所谓的 `String` 对象。执行输入操作后，会从头到尾按顺序读取数据，执行输出操作后，则会按写入顺序不断追加数据。

### 17.2.1 输入操作

- **io.gets(rs)**  
**io.each(rs)**  
**io.each\_line(rs)**  
**io.readlines(rs)**

从 `io` 类的对象 `io` 中读取一行数据。用参数 `rs` 的字符串分行。省略 `rs` 时则用预定义变量 `$/`（默认值为 `"\n"`）。

这些方法返回的字符串中包含行末尾的换行符。用 `chomp!` 方法可以很方便地删除字符串末尾的换行符。

输入完毕后再尝试获取数据时，`gets` 方法会返回 `nil`。另外，我们还可以使用 `eof?` 方法检查输入是否已经完毕。

```
while line = io.gets  
  line.chomp!  
  |      # 对line 进行的操作  
end  
p io.eof?    #=> true
```

`while` 条件表达式中同时进行了变量赋值与条件判断的操作。将 `gets` 方法的返回值复制给 `line`，并将该值作为 `while` 语句的条件来判断。上面是 `gets` 方法的经典用法，大家应该尽快掌握这种写法。

用 `each_line` 方法也可以实现同样的效果。

```
io.each_line do |line|  
  line.chomp!  
  |      # 对line 进行的操作  
end
```

另外，用 `readlines` 方法可以一次性地读取所有数据，并返回将每行数据作为元素封装的数组。

```
ary = io.readlines  
ary.each_line do |line|  
  line.chomp!  
  |      # 对line 进行的操作  
end
```

**备注** `gets` 方法与 `puts` 方法，分别是“get string”、“put string”的意思。

- **io.lineno**

```
io.lineno=(number)
```

使用 `gets` 方法、`each_line` 方法逐行读取数据时，会自动记录读取的行数。这个行数可以通过 `lineno` 方法取得。此外，通过 `lineno=` 方法也可以改变这个值，但值的改变并不会对文件指针（后述）有影响。

在下面的例子中，逐行读取标准输入的数据，并在行首添加行编号。

```
$stdin.each_line do |line|
  printf("%3d %s", $stdin.lineno, line)
end
```

- **io.each\_char**

逐个字符地读取 `io` 中的数据并执行块。将得到的字符（`String` 对象）作为块变量传递。

```
io.each_char do |ch|
  |      # 对line 进行的操作
end
```

- **io.each\_byte**

逐个字节地读取 `io` 中的数据并启动块。将得到的字节所对应的 ASCII 码以整数值的形式传递给块变量。

- **io.getc**

只读取 `io` 中的一个字符。根据文件编码的不同，有时一个字符会由多个字节组成，但这个方法只会读取一个字符，然后返回其字符串对象。数据全部读取完后再读取时会返回 `nil`。

```
while ch = io.getc
  |      # 对line 进行的操作
end
```

- **io.ungetc(ch)**

将参数 `ch` 指定的字符退回到 `io` 的输入缓冲中。

```
# hello.txt 中的内容为“Hello, Ruby.\n”
File.open("hello.txt") do |io|
  p io.getc  #=> "H"
  io.ungetc(72)
  p io.gets  #=> "Hello, Ruby.\n"
end
```

指定一个字符大小的字符串对象。对可退回的字符数没有限制。

- **io.getbyte**

只读取 `io` 中的一个字节，返回得到的字节转换为 ASCII 码后的整数对象。数据全部读取完后再读取时会返回 `nil`。

- **io.ungetbyte(byte)**

将参数 `byte` 指定的一个字节退回到输入缓冲中。参数为整数时，将该整数除以 256 后的余数作为 ASCII 码字符返回一个字节；参数为字符串时，只返回字符串的第一个字节。

- **io.read(size)**

读取参数 `size` 指定的大小的数据。不指定大小时，则一次性读取全部数据并返回。

```
# hello.txt 中的内容为“Hello, Ruby.\n”
File.open("hello.txt") do |io|
  p io.read(5)  #=> "Hello"
  p io.read     #=> ",Ruby.\n"
end
```

## 17.2.2 输出操作

- **io.puts(str0, str1, ...)**

对字符串末尾添加换行符后输出。指定多个参数时，会分别添加换行符。如果参数为 `String` 类以外的对象，则会调用 `to_s` 方法，将其转换为字符串后再输出。

### 代码清单 17.3 stdout\_put.rb

```
$stdout.puts "foo", "bar", "baz"
```

#### 执行示例

```
> ruby stdout_put.rb
foo
bar
```

- ***io.putc(ch)***

输出参数 *ch* 指定的字符编码所对应的字符。参数为字符串时输出首字符。

#### 代码清单 17.4 stdout\_putc.rb

```
$stdoutputc(82) # 82 是R 的ASCII 码
$stdoutputc("Ruby")
$stdoutputc("\n")
```

#### 执行示例

```
> ruby stdout_putc.rb
RR
```

- ***io.print(str0, str1, ...)***

输出参数指定的字符串。参数可指定多个字符串。参数为 *String* 以外的对象时会自动将其转换为字符串。

- ***io.printf(fmt, arg0, arg1, ...)***

按照指定的格式输出字符串。格式 *fmt* 的用法与 *printf* 方法一样，请参考第 192 页专栏《printf 方法与 sprintf 方法》。

- ***io.write(str)***

输出参数 *str* 指定的字符串。参数为 *String* 以外的对象时会自动将其转换为字符串。方法返回值为输出的字节数。

```
size = $stdout.write("Hello.\n")      #=> Hello.
p size                           #=> 6
```

- ***io<<str***

输出参数 *str* 指定的字符串。`<<` 会返回接收者本身，因此可以像下面这样写：

```
io << "foo" << "bar" << "baz"
```

## 17.3 文件指针

一般情况下，我们会以行为单位处理文本数据。由于只有当读取到换行符时才能知道行的长度，因此，如要读取第 100 行的数据，就意味着要将这 100 行的数据全部读取。另外，如果我们修改了数据，行的长度也会随之变更，这样一来，文件中后面的数据就都要做出修改。

为了提高读取效率，可以将文件分成固定长度的文件块，来直接访问某个位置的数据（虽然据此可以访问任意位置的数据，但却不能处理超过指定长度的数据）。

我们用文件指针（file pointer）或者当前文件偏移量（current file offset）来表示 *IO* 对象指向的文件的位置。每当读写文件时，文件指针都会自动移动，而我们也可以使文件指针指向任意位置来读写数据。

- ***io.pos***

*io.pos=(position)*

通过 *pos* 方法可以获得文件指针现在的位置。改变文件指针的位置用 *pos=* 方法。

```
# hello.txt 中的内容为"Hello, Ruby.\n"

File.open("hello.txt") do |io|
  p io.read(5)    #=> "Hello"
  p io.pos        #=> 5
  io.pos = 0
  p io.gets       #=> "Hello, Ruby.\n"
end
```

- ***io.seek(offset, whence)***

移动文件指针的方法。参数 *offset* 为用于指定位置的整数，参数 *whence* 用于指定 *offset* 如何移动（表 17.2）。

表 17.2 *whence* 中指定的值

<i>whence</i>	意义
<i>IO::SEEK_SET</i>	将文件指针移动到 *offset* 指定的位置

IO::SEEK_CUR	将 *offset* 视为相对于当前位置的偏移位置来移动文件指针
IO::SEEK_END	将 *offset* 指定为相对于文件末尾的偏移位置

- `io.rewind`

将文件指针返回到文件的开头。`lineno` 方法返回的行编号为 0。

```
# hello.txt 中的内容为"Hello, Ruby.\n"
File.open("hello.txt") do |io|
  p io.gets      #=> "Hello, Ruby.\n"
  io.rewind
  p io.gets      #=> "Hello, Ruby.\n"
end
```

- `io.truncate(size)`

按照参数 `size` 指定的大小截断文件。

```
io.truncate(0)          # 将文件大小置为0
io.truncate(io.pos)    # 删除当前文件指针以后的数据
```

## 17.4 二进制模式与文本模式

正如我们在第 14 章的专栏《关于换行符》中介绍的那样，不同平台下的换行符也不同。

虽然各个平台的换行符不一样，但为了保证程序的兼容性，会将字符串中的 `\n` 转换为当前 OS 的换行符并输出。此外，在读取的时候也会将实际的换行符转换为 `\n`。

图 17.1 为在 Windows 中转换换行符的情形。

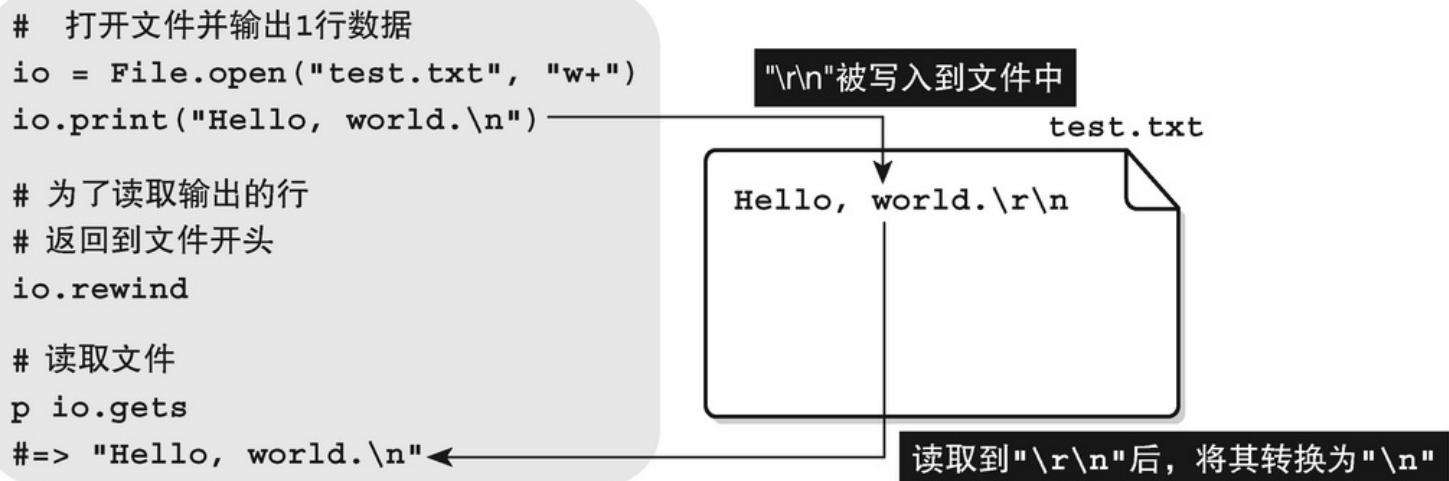


图 17.1 Windows 环境中字符 `\n` 的转换

当需要确定文件大小进行输入 / 输出处理时，或者直接使用从其他平台拷贝的文件时，如果进行换行符转换，就有可能会引发问题。

为了解决上述那样的问题，Ruby 中还提供了不进行换行符转换的方法。换行符处理的前提是以行为单位做输入 / 输出处理，需要转换时称为文本模式，反之不需要转换时则称为二进制模式。

- `io.binmode`

新的 `IO` 对象默认是文本模式，使用 `binmode` 方法可将其变更为二进制模式。

```
File.open("foo.txt", "w") do |io|
  io.binmode
  io.write "Hello, world.\n"
end
```

这样就可以既不用转换换行符，又能得到与文件中一模一样的数据。

备注 转换为二进制模式的 `IO` 对象无法再次转换为文本模式。

## 17.5 缓冲

即使对 `IO` 对象输出数据，结果也并不一定马上就会反映在控制台或者文件中。在使用 `write`、`print` 等方法操作 `IO` 对象时，程序内部会开辟出一定的空间来保存临时生成的数据副本（图 17.2）。这部分空间就称为缓冲（buffer）。缓冲里累积一定量的数据后，就会做实际的输出处理，然后清空缓冲。

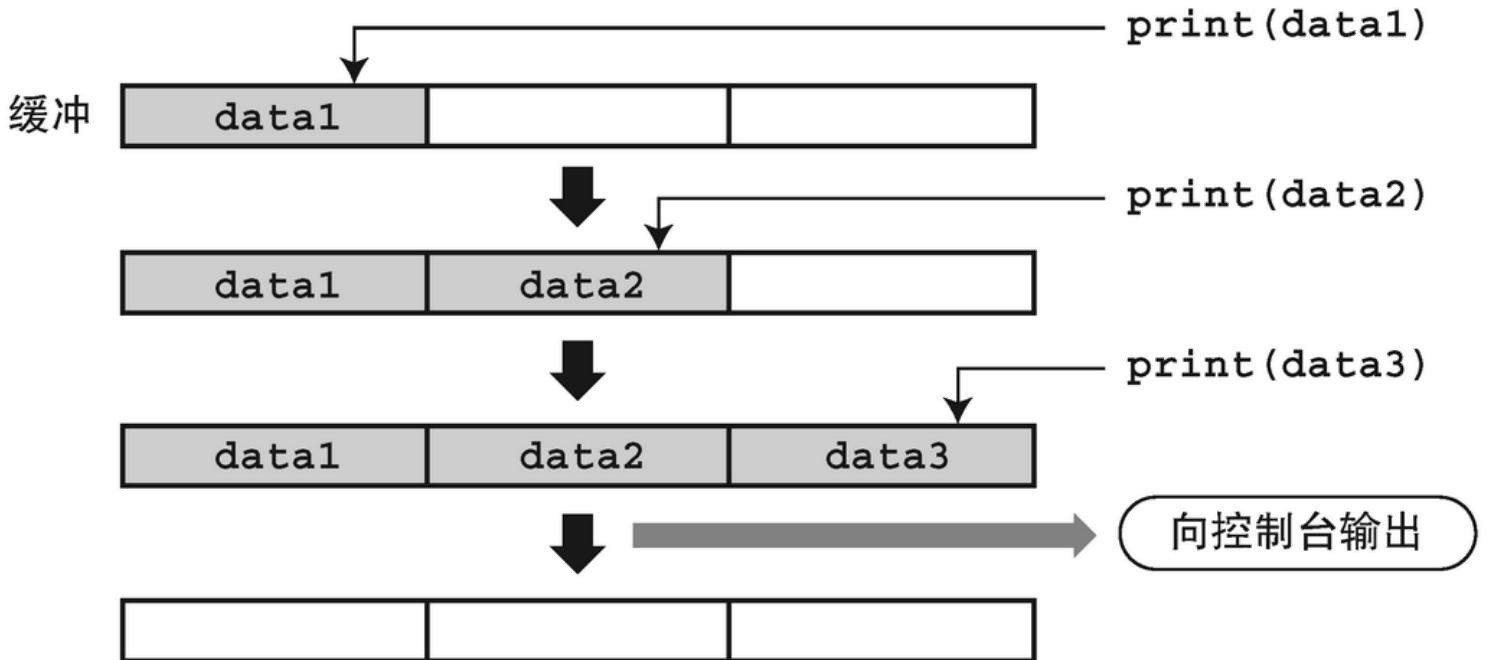


图 17.2 缓冲的状态

像这样，使用临时缓冲进行数据处理称为缓冲处理（buffering）。

在向控制台输出的两种方式（标准输出与标准错误输出）中，标准错误输出完全不采用缓冲处理。因此，当两种方式混合使用时，程序实际输出的顺序可能会与程序代码中记录的顺序不一样。

我们来看看代码清单 17.5 的例子（不同平台下的运行结果可能会不一样）。

代码清单 17.5 test\_buffering1.rb

```
$stdout.print "out1 "
$stderr.print "err1 "
$stdout.print "out2 "
$stdout.print "out3 "
$stderr.print "err2\n"
$stdout.print "out4\n"
```

#### 执行示例

```
> ruby test_buffering1.rb
err1 err2
out1 out2 out3 out4
```

标准错误输出的主要目的是输出如警告、错误等信息，因此执行结果必须马上反映出来。再次强调，建议在显示程序中正常信息以外的信息时使用标准错误输出。

虽然缓冲处理可以提高输出效率，但有时候我们会希望执行结果可以马上反映出来，这时我们就可以用下面的方法来同步数据的操作与输出。

- `io.flush`

强制输出缓冲中的数据。在代码清单 17.5 的基础上追加 `$stdout.flush` 的调用（代码清单 17.6）。

代码清单 17.6 test\_buffering2.rb

```
$stdout.print "out1 "; $stdout.flush
$stderr.print "err1 "
$stdout.print "out2 "; $stdout.flush
$stdout.print "out3 "; $stdout.flush
$stderr.print "err2\n"
$stdout.print "out4\n"
```

#### 执行示例

```
> ruby test_buffering2.rb
out1 err1 out2 out3 err2
out4
```

- `io.sync`

```
io.sync=(state)
```

通过 `io.sync = true`, 程序写入缓冲时 `flush` 方法就会被自动调用。

#### 代码清单 17.7 test\_buffering3.rb

```
$stdout.sync = true # 同步输出处理
$stdout.print "out1"
$stderr.print "err1"
$stdout.print "out2"
$stdout.print "out3"
$stderr.print "err2\n"
$stdout.print "out4\n"
```

即使不逐次调用 `flush` 方法, 也可以像下面那样按顺序输出:

##### 执行示例

```
> ruby test_buffering3.rb
out1 err1 out2 out3 err2
out4
```

## 17.6 与命令进行交互

虽然 Ruby 是几乎什么都能实现的强大的语言, 但是也会有与其他命令进行数据交换的时候。例如, 读取使用 GUN zip 压缩的数据的时候, 使用 `gunzip` 命令会很方便。在 Ruby 中, 使用 `IO.open` 方法可以与其他命令进行数据处理。

- `IO.open(command, mode)`

参数 `mode` 的使用方法与 `File.open` 方法是一样的, 参数缺省时默认为 "`r`" 模式。

用 `IO.open` 方法生成的 `IO` 对象的输入 / 输出, 会关联启动后的命令 `command` 的标准输入 / 输出。也就是说, `IO` 对象的输出会作为命令的输入, 命令的输出则会作为 `IO` 对象的输入。

我们来改造一下第 3 章的 `simple_grep.rb`, 利用 `gunzip` 命令解压处理扩展名为 `.gz` 的文件 (`-c` 为将解压后的结果写入到标准输出时的选项)。

#### 代码清单 17.8 simple\_grep\_gz.rb

```
pattern = Regexp.new(ARGV[0])
filename = ARGV[1]
if /.gz$/ =~ filename
  file = IO.open("gunzip -c #{filename}")
else
  file = File.open(filename)
end
file.each_line do |text|
  if pattern =~ text
    print text
  end
end
```

注 执行代码清单 17.8 时需要 `gunzip` 命令。

- `open("|command", mode)`

将带有管道符号的命令传给 `open` 方法的效果与使用 `IO.open` 方法是一样的。

```
filename = ARGV[0]
open("|gunzip -c #{filename}") do |io|
  io.each_line do |line|
    print line
  end
end
```

## 17.7 open-uri 库

除了控制台、文件以外, 进程间通信时使用的管道 (pipe)、网络间通信时使用的套接字 (socket) 也都可以作为 `IO` 对象使用。管道、套接字的使用方法超出了本书的范围, 因此不做详细说明, 这里我们简单地介绍一下利用 HTTP、FTP 从网络获取数据的方法。

通过 `require` 引用 `open-uri` 库后, 我们就可以像打开普通的文件一样打开 HTTP、FTP 的 URL (代码清单 17.9)。使用 `open-uri` 库的功能时, 不要使用 `File.open` 方法, 只使用 `open` 方法即可。

#### 代码清单 17.9 read\_uri.rb

```

require "open-uri"

# 通过HTTP 读取数据
open("http://www.ruby-lang.org") do |io|
  puts io.read # 将Ruby 的官方网页输出到控制台
end

# 通过FTP 读取数据
url = "ftp://www.ruby-lang.org/pub/ruby/2.0/ruby-2.0.0-p0.tar.gz"
open(url) do |io|
  open("ruby-2.0.0-p0.tar.gz", "w") do |f| # 打开本地文件
    f.write(io.read)
  end
end

```

通过 HTTP 协议时，服务器会根据客户端的状态改变应答的内容，比如返回中文或英语的网页等。为了实现这个功能，请求时就需要向服务器发送元信息（meta information）。例如，代码清单 17.10 中的 HTTP 头部信息 Accept-Language 就表示优先接收中文网页。指定 HTTP 头部信息时，会将其以散列的形式传递给 `open` 方法的第 2 个参数。

**代码清单 17.10 `read_uri_cn.rb`**

```

require "open-uri"

options = {
  "Accept-Language" => "zh-cn, en;q=0.5",
}
open("http://www.ruby-lang.org", options){|io|
  puts io.read
}

```

## 17.8 `stringio` 库

在测试程序时，虽然我们会希望知道向文件或控制台输出了什么，但程序实际执行的结果却往往很难知道。为此，我们可以通过向模拟 `IO` 对象的对象进行输出来确认执行结果。

`StringIO` 就是用于模拟 `IO` 对象的对象。通过 `require` 引用 `stringio` 库后，就可以使用 `StringIO` 对象了（代码清单 17.11）。

**代码清单 17.11 `stringio_puts.rb`**

```

require "stringio"

io = StringIO.new
io.puts("A")
io.puts("B")
io.puts("C")
io.rewind
p io.read #=> "A\nB\nC\n"

```

实际上，向 `StringIO` 对象进行的输出并不会被输出到任何地方，而是会被保存在对象中，之后就可以使用 `read` 方法等来读取该输出。

`StringIO` 对象还有另外一种用法，那就是将字符串数据当作 `IO` 数据处理。将大数据保存在文件中，并将小数据直接传输给别的处理时，通过使用 `StringIO` 对象，程序就可以不区分对待 `IO` 对象和字符串了。实际上，之前介绍的用 `open-uri` 库打开 URI 时，也是有时候返回 `IO` 对象，有时候返回 `StringIO` 对象。不过一般情况下，我们不需要在意这两者的区别。通过将数据字符串传递给 `StringIO.new` 方法的参数，就可以由字符串创建 `StringIO` 对象（代码清单 17.12）。

**代码清单 17.12 `stringio_gets.rb`**

```

require "stringio"

io = StringIO.new("A\nB\nC\n")
p io.gets #=> "A\n"
p io.gets #=> "B\n"
p io.gets #=> "C\n"

```

`StringIO` 对象可以模拟本章中介绍的大部分输入 / 输出操作。

## 练习题

1. 创建脚本，读取文本文件中的内容，按以下条件进行处理。这里将空白和换行以外的连续字符串称为“单词”。

(a) 统计文本的行数

(b) 统计文本的单词数

(c) 统计文本的字符数

2. 创建脚本，按照以下条件覆盖文本文件中原有的数据。

(a) 将文件中的行逆序排列

(b) 保留文件中第 1 行数据，其余全部删除

(c) 保留文件中最后 1 行数据，其余全部删除

3. 定义一个功能类似于 Unix 中的 tail 命令的 `tail` 方法。

`tail` 方法有两个参数。

`tail( 行数, 文件名 )`

从文件的末尾开始数，输出参数指定的行数的内容。也就是说，假设有一个有 100 行数据的文件 `some_file.txt`，执行 `tail(10, "some_file.txt")` 后，程序就会跳过前 90 行数据，只在标准输出中输出最后 10 行数据。

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

# 第 18 章 File 类与 Dir 类

在第 17 章中，我们介绍了如何读写文件中数据。由于 `IO` 类是 `File` 类的父类，因此在 `File` 类中就可以使用 `IO` 类中操作文件的方法。

在本章中，我们将会学习文件名及属性等表面操作，同时也会介绍如何操作文件（数据容器）以及目录（文件容器）等。

与文件以及目录相关的操作一般有以下几种：

- **文件的操作**

名称的变更、拷贝、删除等基本操作。

- **目录的操作**

目录的引用、创建、删除等操作。

- **属性的操作**

“只读”等文件属性的操作。

我们日常在使用计算机的时候，在操作大量文件时，可能都会感到麻烦，甚至还会造成操作失误等。而如果将对大量数据进行的单调操作通过程序来处理的话，就不仅可以提高处理速度，而且还可以避免错误。例如，像只是单纯地变更文件名这种程度的操作，只需几行代码就可以轻松搞定，而且这次写好的程序在以后还可以多次使用。因此我们应该积极地使这种简单的操作变得自动化。不过，在 Windows、Mac OS 平台下，目录被称为文件夹，这一点还请读者们注意。

## 18.1 File 类

`File` 类中实现了操作文件系统的方法。

### 18.1.1 变更文件名

- `File.rename(before, after)`

我们可以用 `File.rename` 方法变更文件名。

```
File.rename("before.txt", "after.txt")
```

还可以将文件移动到已存在的目录下，目录不存在则程序会产生错误。

```
File.rename("data.txt", "backup/data.txt")
```

注 `File.rename` 方法无法跨文件系统或者驱动器移动文件。

如果文件不存在，或者没有适当的文件操作权限等，则文件操作失败，程序抛出异常。

#### 执行示例

```
> irb --simple-prompt
>> File.open("/no/such/file")
Errno::ENOENT: No such file or directory - /no/such/file
from (irb):1:in `initialize'
from (irb):1:in `open'
from (irb):1
from /usr/bin/irb:12:in `<main>'
```

### 18.1.2 复制文件

只用一个 Ruby 预定义的方法是无法复制文件的。这时，我们可以利用 `File.open` 方法与 `write` 方法的组合来实现文件复制。

```
def copy(from, to)
  File.open(from) do |input|
    File.open(to, "w") do |output|
      output.write(input.read)
    end
  end
end
```

不过，由于文件复制是常用的操作，如果每次使用时都需要自己重新定义一次的话就非常麻烦。因此，我们可以通过引用 `fileutils` 库，使用其中的  `FileUtils.cp`（文件拷贝）、 `FileUtils.mv`（文件移动）等方法来操作文件。

```
require "fileutils"
```

```
 FileUtils.cp("data.txt", "backup/data.txt")
 FileUtils.mv("data.txt", "backup/data.txt")
```

`File.rename` 不能实现的跨文件系统、驱动器的文件移动，用 `FileUtils.mv` 方法则可以轻松实现。关于 `fileutils` 库，我们在后续章节中会详细介绍。

### 18.1.3 删除文件

- `File.delete(file)`  
`File.unlink(file)`

我们可以使用 `File.delete` 方法或 `File.unlink` 方法删除文件。

```
File.delete("foo")
```

## 18.2 目录的操作

`Dir` 类中实现了目录相关的操作方法。在详细说明前，我们先来复习一下有关目录的一些基础知识。

目录中可以存放多个文件。除了文件之外，目录中还可以存放其他目录，其他目录中又可以再存放目录……如此无限循环。通过将多个目录进行排列、嵌套，就可以轻松地管理大量的文件。

Windows 的资源管理器（图 18.1）的左侧就是可视化的目录层次结构（树形结构）。用目录名连接 / 的方法即可指定目录中的文件。由于我们可以通过目录名指定文件的位置，因此表示文件位置的目录名就称为路径（path）或路径名。另外，我们把目录树的起点目录称为根目录（root directory），根目录只用 / 表示。

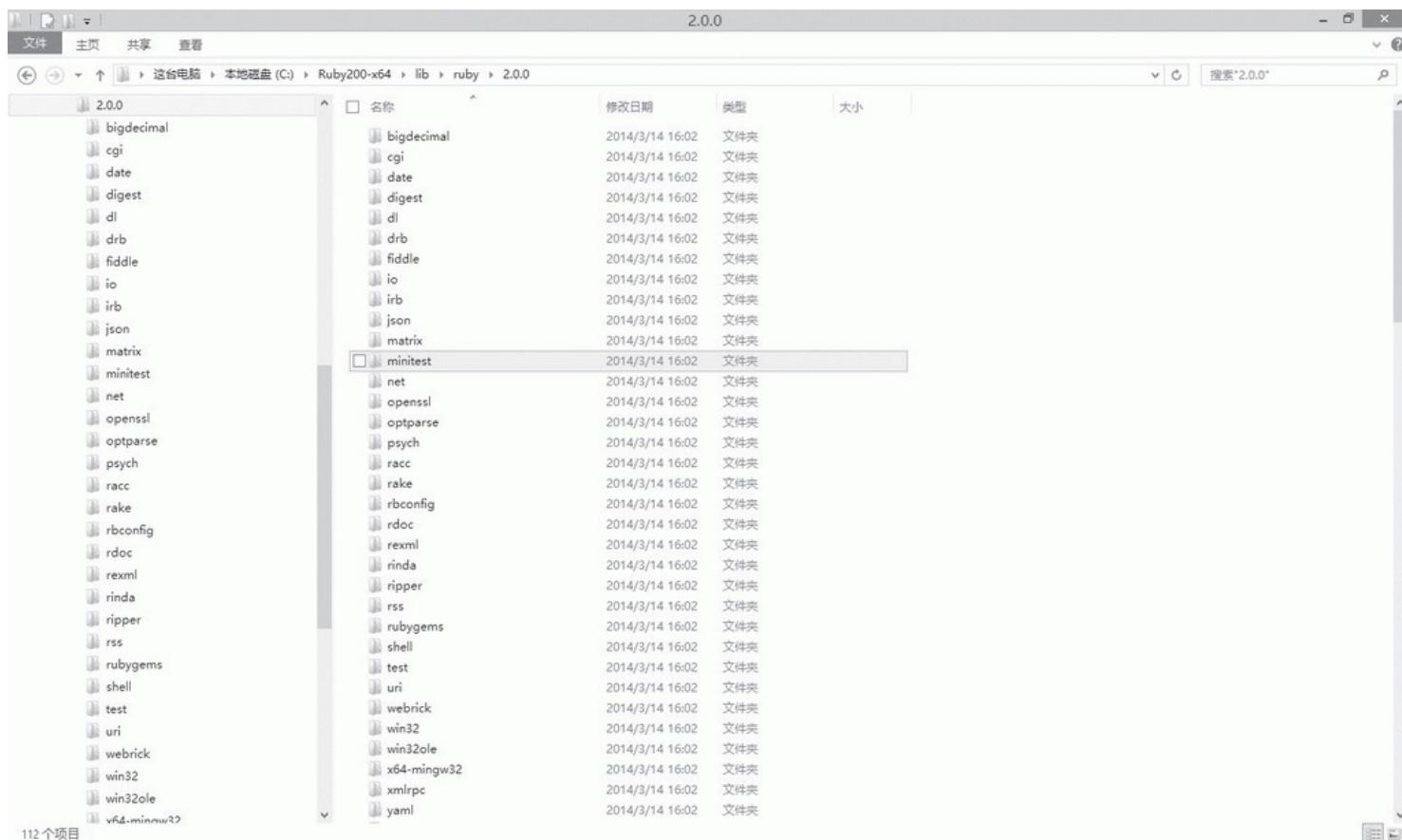


图 18.1 Windows 的资源管理器

专栏

关于 Windows 的路径名

在 Windows 的命令行中，目录分隔符用的是 \。由于使用 \ 后不仅会使字符串难以读懂，而且也不能直接在 Unix 中执行同一个程序，因此还是建议大家尽量使用 /。但有一点请大家注意，像 WIN32OLE 这样使用 Windows 特有的功能时，使用 / 后可能会使程序变得无法执行。

在 Windows 中，驱动器是目录的上层文件管理单位。一般用 1 个英文字母（盘符）表示与之相对应的驱动器，如 A: 表示软盘，C:、D:……表示硬盘。这种情况下，请读者把各驱动器当成独立的根目录来看待。例如，很明显地 C:/ 与 D:/ 表示的是不同的驱动器，但如果只写 / 的话，程序执行位置的不同，其表示的驱动器也不同，继而所表示的目录也不同。

- `Dir.pwd`  
`Dir.chdir(dir)`

程序可以获取运行时所在的目录信息，即当前目录（current directory）。使用 `Dir.pwd` 方法获取当前目录，变更当前目录使用 `Dir.chdir` 方法。我们

可以对 `Dir.chdir` 方法的参数 `dir` 指定相对与当前目录的相对路径，也可以指定相对于根目录的绝对路径。

```
p Dir.pwd          #=> "/usr/local/lib"
Dir.chdir("ruby/2.0.0") #=> 根据相对路径移动
p Dir.pwd          #=> "/usr/local/lib/ruby/2.0.0"
Dir.chdir("/etc")    #=> 根据绝对路径移动
p Dir.pwd          #=> "/etc"
```

当前目录下的文件，我们可以通过指定文件名直接打开，但如果变更了当前目录，则还需要指定目录名。

```
p Dir.pwd          #=> "/usr/local/lib/ruby/2.0.0"
io = File.open("find.rb")
#=> 打开"/usr/local/lib/ruby/2.0.0/find.rb"
io.close
Dir.chdir("../..")   # 移动到上两层的目录中
p Dir.pwd          #=> "/usr/local/lib"
io = File.open("ruby/2.0.0/find.rb")
#=> 打开"/usr/local/lib/ruby/2.0.0/find.rb"
io.close
```

### 18.2.1 目录内容的读取

像介绍文件的时候一样，我们先来了解一下如何读取已存在的目录。读取目录内容的方法与读取文件的方法基本上是一样的。

① 打开目录

② 读取内容

③ 关闭

- `Dir.open(path)`

```
Dir.close
```

与 `File` 类一样，`Dir` 类也有 `open` 方法与 `close` 方法。

我们先试试读取 `/usr/bin` 目录。

```
dir = Dir.open("/usr/bin")
while name = dir.read
  p name
end
dir.close
```

我们也可以像下面那样用 `Dir#each` 方法替换 `while` 语句部分。

```
dir = Dir.open("/usr/bin")
dir.each do |name|
  p name
end
dir.close
```

和 `File.open` 同样，对 `Dir.open` 使用块后也可以省略 `close` 方法的调用。这时程序会将生成的 `Dir` 对象传给块变量。

```
Dir.open("/usr/bin") do |dir|
  dir.each do |name|
    p name
  end
end
```

程序会输出以下内容。

```
".
..
"gnomevfs-copy"
"updmap"
"signver"
"bluetooth-sendto"
|
```

- `dir.read`

与 `File` 类一样，`Dir` 类也有 `read` 方法。

执行 `Dir#read` 后，程序会遍历读取最先打开的目录下的内容。这里读取的内容可分为以下 4 类：

- 表示当前目录的 .
- 表示上级目录的 ..
- 其他目录名
- 文件名

请注意 `/usr/bin` 与 `/usr/bin/` 表示同一个目录。

代码清单 18.1 中的程序会操作指定目录下的所有路径。命令行参数 `ARGV[0]` 的路径为目录时，会对该目录下的文件进行递归处理，除此以外（文件）的情况下则调用 `process_file` 方法。`traverse` 方法会输出指定目录下的所有文件名，执行结果只显示在控制台中。

注释中带 \* 的代码表示忽略当前目录和上级目录，不这么做的话，就会陷入无限循环中，不断地重复处理同一个目录。

#### 代码清单 18.1 `traverse.rb`

```
def traverse(path)
  if File.directory?(path) # 如果是目录
    dir = Dir.open(path)
    while name = dir.read
      next if name == "." # *
      next if name == ".." # *
      traverse(path + "/" + name)
    end
    dir.close
  else
    process_file(path) # 处理文件
  end
end
def process_file(path)
  puts path # 输出结果
end

traverse(ARGV[0])
```

#### • `Dir.glob`

使用 `Dir.glob` 方法后，就可以像 shell 那样使用 \* 或者 ? 等通配符（wildcard character）来取得文件名。`Dir.glob` 方法会将匹配到的文件名（目录名）以数组的形式返回。

下面我们列举一些常用的匹配例子。

- 获取当前目录中所有的文件名。（无法获取 Unix 中以 “.” 开始的隐藏文件名）

```
Dir.glob("*")
```

- 获取当前目录中所有的隐藏文件名

```
Dir.glob(".*")
```

- 获取当前目录中扩展名为 `.html` 或者 `.htm` 的文件名。可通过数组指定多个模式。

```
Dir.glob(["*.html", "*.htm"])
```

- 模式中若没有空白，则用 `%w(...)` 生成字符串数组会使程序更加易懂。

```
Dir.glob(%w(*.html *.htm))
```

- 获取子目录下扩展名为 `.html` 或者 `.htm` 的文件名。

```
Dir.glob(["*/*.html", "*/*.htm"])
```

- 获取文件名为 `foo.c`、`foo.h`、`foo.o` 的文件。

```
Dir.glob("foo.[cho]")
```

- 获取当前目录及其子目录中所有的文件名，递归查找目录。

```
Dir.glob("**/*")
```

- 获取目录 `foo` 及其子目录中所有扩展名为 `.html` 的文件名，递归查找目录。

```
Dir.glob("foo/**/*.html")
```

可以像下面那样用 `Dir.glob` 方法改写代码清单 18.1 的 `traverse` 方法。

#### 代码清单 18.2 `traverse_by_glob.rb`

```
def traverse(path)
  Dir.glob(["#{path}/**/*", "#{path}/**/.*"]).each do |name|
    unless File.directory?(name)
      process_file(name)
    end
  end
end
```

### 18.2.2 目录的创建与删除

- `Dir.mkdir(path)`

创建新目录用 `Dir.mkdir` 方法。

```
Dir.mkdir("temp")
```

- `Dir.rmdir(path)`

删除目录用 `Dir.rmdir` 方法。要删除的目录必须为空目录。

```
Dir.rmdir("temp")
```

### 18.3 文件与目录的属性

文件与目录都有所有者、最后更新时间等属性。接下来我们就来看看如何引用和更改这些属性。

- `File.stat(path)`

通过 `File.stat` 方法，我们可以获取文件、目录的属性。`File.stat` 方法返回的是 `File::Stat` 类的实例。`File::Stat` 类的实例方法如表 18.1 所示。

表 18.1 `File::Stat` 类的实例方法

方法	返回值的含义
<code>dev</code>	文件系统的编号
<code>ino</code>	i-node 编号
<code>mode</code>	文件的属性
<code>nlink</code>	链接数
<code>uid</code>	文件所有者的用户 ID
<code>gid</code>	文件所属组的组 ID
<code>rdev</code>	文件系统的驱动器种类
<code>size</code>	文件大小
<code>blksize</code>	文件系统的块大小
<code>blocks</code>	文件占用的块数量
<code>atime</code>	文件的最后访问时间
<code>mtime</code>	文件的最后修改时间
<code>ctime</code>	文件状态的最后更改时间

其中，除 `atime` 方法、`mtime` 方法、`ctime` 方法返回 `Time` 对象外，其他方法都返回整数值。

通过 `uid` 方法与 `gid` 方法获取对应的用户 ID 与组 ID 时，需要用到 `Etc` 模块。我们可以通过 `require 'etc'` 来引用 `Etc` 模块。

备注 mswin32 版的 Ruby 不能使用 `Etc` 模块。

下面是显示文件 `/usr/local/bin/ruby` 的用户名与组名的程序：

```
require 'etc'

st = File.stat("/usr/local/bin/ruby")
pw = Etc.getpwid(st.uid)
p pw.name      #=> "root"
gr = Etc.getgrgid(st.gid)
p gr.name      #=> "wheel"
```

- **`File.ctime(path)`**

`File.mtime(path)`

`File.atime(path)`

这三个方法的执行结果与实例方法 `File::Stat#ctime`、`File::Stat#mtime`、`File::Stat#atime` 是一样的。需要同时使用其中的两个以上的方法时，选择实例方法会更加有效率。

- **`File.utime(atime, mtime, path)`**

改变文件属性中的最后访问时间 `atime`、最后修改时间 `mtime`。时间可以用整数值或者 `Time` 对象指定。另外，同时还可以指定多个路径。下面是修改文件 `foo` 的最后访问时间以及最后修改时间的程序。通过 `Time.now` 方法创建表示当前时间的 `Time` 对象，然后将时间设为当前时间减去 100 秒。

```
filename = "foo"
File.open(filename, "w").close      # 创建文件后关闭

st = File.stat(filename)
p st.ctime      #=> 2013-03-30 04:20:01 +0900
p st.mtime      #=> 2013-03-30 04:20:01 +0900
p st.atime      #=> 2013-03-30 04:20:01 +0900

File.utime(Time.now-100, Time.now-100, filename)
st = File.stat(filename)
p st.ctime      #=> 2013-03-30 04:20:01 +0900
p st.mtime      #=> 2013-03-30 04:18:21 +0900
p st.atime      #=> 2013-03-30 04:18:21 +0900
```

- **`File.chmod(mode, path)`**

修改文件 `path` 的访问权限（permission）。`mode` 的值为整数值，表示新的访问权限值。同时还能指定多个路径。

备注 在 Windows 中只有文件的所有者才有写入的权限。执行权限则是由扩展名（.bat、.exe 等）决定的。

访问权限是用于表示是否可以进行执行、读取、写入操作的 3 位数（图 18.2）。访问权限又细分为所有者、所属组、其他三种权限，因此完整的访问权限用 9 位数表示。



图 18.2 访问权限值的含义

例如，设定“所有者为读写权限，其他用户为只读权限”时的权限位（permission bit）为 110100100。由于 8 进制刚好是用 1 个数字表示 3 位，因此一般会将权限位用 8 进制表示。刚才的 110100100 用 8 进制表示则为 0644。

以下程序是将文件 test.txt 的访问权限设为 0755（所有者拥有所有权限，其他用户为只读权限）：

```
File.chmod(0755, "test.txt")
```

像对现在的访问权限追加执行权限这样追加特定运算时，需要对从 `File.stat` 方法得到的访问权限位进行追加位的位运算，然后再用计算后的新值重新设定。使用按位或运算追加权限位。

```
rb_file = "test.rb"
st = File.stat(rb_file)
File.chmod(st.mode | 0111, rb_file) # 追加执行权限
```

- `File.chown(owner, group, path)`

改变文件 `path` 的所有者。`owner` 表示新的所有者的用户 ID，`group` 表示新的所属组 ID。同时也可以指定多个路径。执行这个命令需要有管理员的权限。

**备注** Windows 中虽然也提供了这个方法，但调用时什么都不会发生。

## FileTest 模块

`FileTest` 模块中的方法用于检查文件的属性（表 18.2）。该模块可以在 `include` 后使用，也可以直接作为模块函数使用。其中的方法也可以作为 `File` 类的类方法使用。

表 18.2 FileTest 模块的方法

方法	返回值
<code>exist?(path)</code>	<code>path</code> 若存在则返回 <code>true</code>
<code>file?(path)</code>	<code>path</code> 若是文件则返回 <code>true</code>
<code>directory?(path)</code>	<code>path</code> 若是目录则返回 <code>true</code>

<code>owned?(path)</code>	<code>path</code> 的所有者与执行用户一样则返回 <code>true</code>
<code>grpowned?(path)</code>	<code>path</code> 的所属组与执行用户的所属组一样则返回 <code>true</code>
<code>readable?(path)</code>	<code>path</code> 可读取则返回 <code>true</code>
<code>writable?(path)</code>	<code>path</code> 可写则返回 <code>true</code>
<code>executable?(path)</code>	<code>path</code> 可执行则返回 <code>true</code>
<code>size(path)</code>	返回 <code>path</code> 的大小
<code>size?(path)</code>	<code>path</code> 的大小比 0 大则返回 <code>true</code> ， 大小为 0 或者文件不存在则返回 <code>nil</code>
<code>zero?(path)</code>	<code>path</code> 的大小为 0 则返回 <code>true</code>

## 18.4 文件名的操作

操作文件时，我们常常需要操作文件名。Ruby 为我们提供了从路径名中获取目录名、文件名的方法、以及相反的由目录名和文件名生成路径名的方法。

- `File.basename(path[, suffix])`

返回路径 `path` 中最后一个 `"/"` 以后的部分。如果指定了扩展名 `suffix`，则会去除返回值中扩展名的部分。在从路径中获取文件名的时候使用本方法。

```
p File.basename("/usr/local/bin/ruby")      #=> "ruby"
p File.basename("src/ruby/file.c", ".c")    #=> "file"
p File.basename("file.c")                   #=> "file"
```

- `File.dirname(path)`

返回路径 `path` 中最后一个 `"/"` 之前的内容。路径不包含 `"/"` 时则返回 `..`。在从路径中获取目录名的时候使用本方法。

```
p File.dirname("/usr/local/bin/ruby")      #=> "/usr/local/bin"
p File.dirname("ruby")                     #=> ".."
p File.dirname("/")                       #=> "/"
```

- `File.extname(path)`

返回路径 `path` 中 `basename` 方法返回结果中的扩展名。没有扩展名或者以 `..` 开头的文件名时则返回空字符串。

```
p File.extname("helloruby.rb")           #=> ".rb"
p File.extname("ruby-2.0.0-p0.tar.gz")   #=> ".gz"
p File.extname("img/foo.png")            #=> ".png"
p File.extname("/usr/local/bin/ruby")    #=> ""
p File.extname("~/.zshrc")              #=> ""
p File.extname("/etc/init.d/ssh")        #=> ""
```

- `File.split(path)`

将路径 `path` 分割为目录名与文件名两部分，并以数组形式返回。在知道返回值的数量时，使用多重赋值会方便得多。

```
p File.split("/usr/local/bin/ruby")
      #=> ["/usr/local/bin", "ruby"]
p File.split("ruby")
      #=> [".", "ruby"]
p File.split("/")
      #=> ["/", ""]

dir, base = File.split("/usr/local/bin/ruby")
p dir
      #=> "/usr/local/bin"
p base
      #=> "ruby"
```

- `File.join(name1[, name2, ...])`

用 `File::SEPARATOR` 连接参数指定的字符串。`File::SEPARATOR` 的默认设值为 `"/"`。

```
p File.join("/usr/bin", "ruby")      #=> "/usr/bin/ruby"
p File.join(".", "ruby")            #=> "./ruby"
```

- `File.expand_path(path[, default_dir])`

根据目录名 `default_dir`, 将相对路径 `path` 转换为绝对路径。不指定 `default_dir` 时, 则根据当前目录转换。

```
p Dir.pwd                      #=> "/usr/local"
p File.expand_path("bin")        #=> "/usr/local/bin"
p File.expand_path("../bin")     #=> "/usr/bin"
p File.expand_path("bin", "/usr") #=> "/usr/bin"
p File.expand_path("../etc", "/usr") #=> "/etc"
```

在 Unix 中, 可以用 `~` 用户名的形式获取用户的主目录 (home directory)。`~/` 表示当前用户的主目录。

```
p File.expand_path("~/gotoyozo/bin")    #=> "/home/gotoyozo/bin"
p File.expand_path("~/takahashim/bin")   #=> "/home/takahashim/bin"
p File.expand_path("~/bin")             #=> "/home/gotoyozo/bin"
```

## 18.5 与操作文件相关的库

接下来我们来介绍一下 Ruby 默认提供的与操作文件相关的库。预定义的 `File` 类与 `Dir` 类只提供了 OS 中 Ruby 可以操作的最基本的功能。为了提高写程序的效率, 有必要掌握本节中所介绍的库。

### 18.5.1 find 库

`find` 库中的 `find` 模块被用于对指定的目录下的目录或文件做递归处理。

- `Find.find(dir){|path| ...}`

```
Find.prune
```

`Find.find` 方法会将目录 `dir` 下的所有文件路径逐个传给路径 `path`。

使用 `Find.find` 方法时, 调用 `Find.prune` 方法后, 程序会跳过当前查找目录下的所有路径 (只是使用 `next` 时, 则只会跳过当前目录, 子目录的内容还是会继续查找)。

代码清单 18.3 是一个显示命令行参数指定目录下内容的脚本。`listdir` 方法会显示参数 `top` 路径下所有的目录名。将不需要查找的目录设定到 `IGNORES` 后, 就可以通过 `Find.prune` 方法忽略那些目录下的内容的处理。

代码清单 18.3 `listdir.rb`

```
require 'find'

IGNORES = [ /^\./, /^CVS$/, /^RCS$/ ]

def listdir(top)
  Find.find(top) do |path|
    if FileTest.directory?(path) # 如果path 是目录
      dir, base = File.split(path)
      IGNORES.each do |re|
        if re =~ base           # 需要忽略的目录
          Find.prune            # 忽略该目录下的内容的查找
        end
      end
      puts path                # 输出结果
    end
  end
end

listdir(ARGV[0])
```

### 18.5.2 tempfile 库

`tempfile` 库用于管理临时文件。

在处理大量数据的程序中, 有时候会将一部分正在处理的数据写入到临时文件。这些文件一般在程序执行完毕后就不再需要, 因此必须删除, 但为了能够确实删除文件, 就必须记住每个临时文件的名称。此外, 有时候程序还会同时处理多个文件, 或者同时执行多个程序, 考虑到这些情况, 临时文件还不能使用相同的名称, 而这就形成了一个非常麻烦的问题。

`tempfile` 库中的 `Tempfile` 类就是为了解决上述问题而诞生的。

- `Tempfile.new(basename[, tempdir])`

创建临时文件。实际生成的文件名的格式为“`basename+ 进程 ID+ 流水号`”。因此，即使使用同样的 `basename`，每次调用 `new` 方法生成的临时文件也都是不一样的。如果不指定目录名 `tempdir`，则会按照顺序查找 `ENV["TMPDIR"]`、`ENV["TMP"]`、`ENV["TEMP"]`、`/tmp`，并把最先找到的目录作为临时目录使用。

- `tempfile.close(real)`

关闭临时文件。`real` 为 `true` 时则马上删除临时文件。即使没有明确指定删除，`Tempfile` 对象也会在 GC（详情请参考本章最后的专栏）的时候并一并删除。`real` 的默认值为 `false`。

- `tempfile.open`

再次打开 `close` 方法关闭的临时文件。

- `tempfile.path`

返回临时文件的路径。

### 18.5.3 fileutils 库

在之前的内容中，我们已经接触过了 `fileutils` 库的 `FileUtils.cp`、`FileUtils.mv` 方法。通过 `require` 引用 `fileutils` 库后，程序就可以使用 `FileUtils` 模块中提供的各种方便的方法来操作文件。

- `FileUtils.cp(from, to)`

把文件从 `from` 拷贝到 `to`。`to` 为目录时，则在 `to` 下面生成与 `from` 同名的文件。此外，也可以将 `from` 作为数组来一次性拷贝多个文件，这时 `to` 必须指定为目录。

- `FileUtils.cp_r(from, to)`

功能与 `FileUtils.cp` 几乎一模一样，不同点在于 `from` 为目录时，则会进行递归拷贝。

- `FileUtils.mv(from, to)`

把文件（或者目录）`from` 移动到 `to`。`to` 为目录时，则将文件作为与 `from` 同名的文件移动到 `to` 目录下。也可以将 `from` 作为数组来一次性移动多个文件，这时 `to` 必须指定为目录。

- `FileUtils.rm(path)`

- `FileUtils.rm_f(path)`

删除 `path`。`path` 只能为文件。也可以将 `path` 作为数组来一次性删除多个文件。`FileUtils.rm` 方法在执行删除处理的过程中，若发生异常则中断处理，而 `FileUtils.rm_f` 方法则会忽略错误，继续执行。

- `FileUtils.rm_r(path)`

- `FileUtils.rm_rf(path)`

删除 `path`。`path` 为目录时，则进行递归删除。此外，也可以将 `path` 作为数组来一次性删除多个文件（或者目录）。`FileUtils.rm_r` 方法在执行处理的过程中，若发生异常则中断处理，而 `FileUtils.rm_rf` 方法则会忽略错误，继续执行。

- `FileUtils.compare(from, to)`

比较 `from` 与 `to` 的内容，相同则返回 `true`，否则则返回 `false`。

- `FileUtils.install(from, to[, option])`

把文件从 `from` 拷贝到 `to`。如果 `to` 已经存在，且与 `from` 内容一致，则不会拷贝。`option` 参数用于指定目标文件的访问权限，如下所示。

```
FileUtils.install(from, to, :mode => 0755)
```

- `FileUtils.mkdir_p(path)`

使用 `Dir.mkdir` 方法创建 `"foo/bar/baz"` 这样的目录时，需要像下面那样按顺序逐个创建上层目录。

```
Dir.mkdir("foo")
Dir.mkdir("foo/bar")
Dir.mkdir("foo/bar/baz")
```

而如果使用 `FileUtils.mkdir_p` 方法，则只需调用一次就可以自动创建各层的目录。此外，也可以将 `path` 作为数组来一次性创建多个目录。

```
FileUtils.mkdir_p("foo/bar/baz")
```

## 关于 GC

在第 3 部中我们介绍了各种类型的对象，而在程序中生成这些对象（一部分除外）时都会消耗内存空间。例如数组、字符串等，如果长度变大了，那么它们需要的内存空间也会随之变大。程序为了能正常运行不可避免地要创建对象，但是计算机的内存空间却不是可以无限使用的，因此就必须释放已经不需要的对象所占用的内存空间。

下面的写法在本书中已经出现过多次，在这种情况下，变量 `line` 引用的字符串，在下一次读取时就不能再被引用了。

```
io.each_line do |line|
  print(line)
end
```

还有，在方法执行完毕后，在方法中临时生成的对象也不再需要了。

```
def hello(name)
  msg = "Hello, #{name}"      #=> 创建新的字符串对象
  puts(msg)
end
```

但是，内存空间释放并不是大部分程序主要关心的功能，而且忘记释放内存，或者错把正在用的对象释放等，都很可能引起难缠的程序漏洞（bug）。因此，在 Ruby（Java、Perl、Lisp 等语言也都具备这样的功能）中，解析器（interpreter）会在适当的时机，释放已经没有被任何地方引用的对象所占的资源。这样的功能，我们称之为 Garbage Collection（垃圾回收的意思），简称 GC。

有了 GC 后，我们就无需再为内存管理而烦恼了。GC 是支撑 Ruby 的宗旨——快乐编程的重要功能之一。

## 练习题

1. 变量 `$:` 以数组的形式保存着 Ruby 中可用的库所在的目录。定义 `print_libraries` 方法，利用变量 `$:`，按名称顺序输出 Ruby 中可用的库。
2. 定义 `du` 方法，功能类似于 Unix 命令 du，递归输出文件及目录中的数据大小。

本方法只有一个参数。

`du( 目录名 )`

输出指定目录下的文件大小（字节数）以及目录大小。目录大小为该目录下所有文件大小的总和。

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

# 第 19 章 Encoding 类

在本章中，我们将介绍 `Encoding` 类、以及 Ruby 中编码的相关用法。

- **Ruby 的编码与字符串**

介绍 Ruby 中字符串与编码的使用方法。

- **脚本编码与魔法注释**

再次介绍脚本编码与魔法注释

- **Encoding 类**

介绍编码的基础——`Encoding` 类的相关用法。

- **正则表达式与编码**

说明正则表达式与编码的关系。

- **IO 类与编码**

说明 `IO` 类与编码的关系。

## 19.1 Ruby 的编码与字符串

字符编码是计算机进行字符操作的基础，这一点我们已经在第 14 章的专栏中做过了介绍。就像在专栏中介绍的那样，字符编码有多种，而且即使是在同一个程序中，有时候输入 / 输出的字符编码也有可能不一样。例如程序输入是 UTF-8 字符编码，而输出却是 Shift\_JIS 字符编码等情况。虽然“あ”的 UTF-8 的字符编码与 Shift\_JIS 的字符编码实际上是不同的，但经过适当的转换，也是可以编写这样的程序的。

至于程序如何处理字符编码，不同的编程语言有不同的解决方案。Ruby 的每个字符串对象都包含“字符串数据本身”以及“该数据的字符编码”两个信息。其中，关于字符编码的信息即我们一般所讲的编码。

创建字符串对象一般有两种方法，一种是在脚本中直接以字面量的形式定义，另外一种是从程序的外部（文件、控制台、网络等）获取字符串数据。数据的获取方式决定了它的编码方式。截取字符串的某部分，或者连接多个字符串生成新字符串等的时候，编码会继承原有的字符串的编码。

程序向外部输出字符串时，必须指定适当的编码。

Ruby 会按照以下信息决定字符串对象的编码，或者在输入 / 输出处理时转换编码。

- **脚本编码**

决定字面量字符串对象编码的信息，与脚本的字符编码一致。详细内容请参考 19.2 节。

- **内部编码与外部编码**

内部编码是指从外部获取的数据在程序中如何处理的信息。与之相反，外部编码是指程序向外部输出时与编码相关的信息。两者都与 `IO` 对象有关联。详细内容请参考 19.5 节。

## 19.2 脚本编码与魔法注释

我们在第 1 章中简单介绍过魔法注释。Ruby 脚本的编码就是通过在脚本的开头书写魔法注释来指定的。

脚本自身的编码称为脚本编码（script encoding）。脚本中的字符串、正则表达式的字面量会依据脚本编码进行解释。脚本编码为 EUC-JP 时，字符串、正则表达式的字面量也都为 EUC-JP。同样，如果脚本编码为 Shift\_JIS，那么字符串、正则表达式的字面量也为 Shift\_JIS。

我们把指定脚本编码的注释称为魔法注释（magic comment）。Ruby 在解释脚本前，会先读取魔法注释来决定脚本编码。

魔法注释必须写在脚本的首行（第 1 行以 `#!` ~ 开头时，则写在第 2 行）。下面是将脚本编码指定为 UTF-8 的例子。

```
# encoding: utf-8
```

**备注** 在 Unix 中，赋予脚本执行权限后，就可以直接执行脚本。这时，可以在文件开头以 `#!` 命令的路径的形式来指定执行脚本的命令。在本书的例子中，我们经常使用 `>ruby` 脚本名 这样的形式来表示在命令行执行脚本的命令为 `ruby`，但若像“`#!/usr/bin/ruby`”这样，在文件开头写上 `ruby` 命令的路径的话，那么就能直接以 `>` 脚本名的形式执行脚本了。

此外，为了可以兼容 Emacs、VIM 等编辑器的编码指定方式，我们也可以像下面这样写。

```
# -*- coding: utf-8 -*-      # 编辑器为Emacs 的时候
# vim:set fileencoding=utf-8: # 编辑器为VIM 的时候
```

程序代码的编码会严格检查是否与脚本编码一致。因此，有时候直接写上日语的字符串后就会产生错误 1。

1中文也需要注意同样的问题。——译者注

```
# encoding: US-ASCII
a = 'こんにちは'      #=> invalid multibyte char (US-ASCII)
```

由于 US-ASCII 不能表示日语的字符串，因此会产生错误。在 Ruby1.9 中，没有魔法注释时默认脚本编码也为 US-ASCII，因此也会产生这个错误。

为了使日语字符能正常显示，必须指定适当的编码 2。而在 Ruby2.0 中，由于没有魔法注释时的默认脚本编码为 UTF-8，因此如果代码是以 UTF-8 编码编写的话，那么就无须使用魔法注释了。

2中文字符也一样。——译者注

但有时仅使用魔法注释是不够的。例如，使用特殊字符 \u 创建字符串后，即使脚本编码不是 UTF-8，其生成的字符串也一定是 UTF-8。

```
# encoding: EUC-JP
a = "\u3042\u3044"
puts a          #=> "あい"
p a.encoding   #=> #<Encoding:UTF-8>
```

因此，必须使用 `encode!` 方法明确进行编码转换。

```
# encoding: EUC-JP
a = "\u3042\u3044"
a.encode!("EUC-JP")
p a.encoding   #=> #<Encoding:EUC-JP>
```

这样，变量 `a` 的字符串的编码也就变为 EUC-JP 了。

### 19.3 Encoding 类

我们可以用 `String#encoding` 方法来调查字符串的编码。`String#encoding` 方法返回 `Encoding` 对象。

```
p "こんにちは".encoding #=> #<Encoding:UTF-8>
```

本例中的“こんにちは”字符串对象的编码为 UTF-8。

**备注** 日语 Windows 环境中的字符编码一般为 Windows-31J。这是 Windows 专用的扩展自 Shift\_JIS 的编码，例如，Shift\_JIS 中原本并没有①。Windows-31J 还有一个别名叫 CP932（Microsoft code page932 的意思），在互联网上就字符编码讨论时，有时候会用到这个名称。3

3简体中文 Windows 环境中使用的字符编码为 GBK（CP936），向下兼容 GB2312 编码。——译者注

在脚本中使用不同的编码时，需要进行必要的转换。我们可以用 `String#encode` 方法转换字符串对象的编码。

```
str = "こんにちは"
p str.encoding      #=> #<Encoding:UTF-8>
str2 = str.encode("EUC-JP")
p str2.encoding    #=> #<Encoding:EUC-JP>
```

在本例中，我们尝试把 UTF-8 字符串对象转换为新的 EUC-JP 字符串对象。

在操作字符串时，Ruby 会自动进行检查。例如，如果要连接不同编码的字符串则会产生错误。

```
# encoding: utf-8

str1 = "こんにちは"
p str1.encoding      #=> #<Encoding:UTF-8>
str2 = "あいうえお".encode("EUC-JP")
p str2.encoding      #=> #<Encoding:EUC-JP>
str3 = str1 + str2 #=> incompatible character encodings: UTF-8
                  #=> and EUC-JP(Encoding::CompatibilityError)
```

为了防止错误，在连接字符串前，必须使用 `encode` 方法等把两者转换为相同的编码。

还有，在进行字符串比较时，如果编码不一样，即使表面的值相同，程序也会将其判断为不同的字符串。

```
# encoding: utf-8
p "あ" == "あ".encode("Shift_JIS")      #=> false
```

另外，在本例中，用 `String#encode` 指定编码时，除了可以使用编码名的字符串外，还可以直接使用 `Encoding` 对象来指定。

#### Encoding 类的方法

接下来，我们将会介绍 `Encoding` 类的方法。

- `Encoding.compatible?(str1, str2)`

检查两个字符串的兼容性。这里所说的兼容性是指两个字符串是否可以连接。可兼容则返回字符串连接后的编码，不可兼容则返回 `nil`。

```
p Encoding.compatible?("AB".encode("EUC-JP"),
                      "あ".encode("UTF-8"))    #=> #<Encoding:UTF-8>
p Encoding.compatible?("あ".encode("EUC-JP"),
                      "あ".encode("UTF-8"))    #=> nil
```

`AB` 这个字符串的编码无论是 EUC-JP 还是 UTF-8 都是一样的，因此，将其转换为 EUC-JP 后也可以与 UTF-8 字符串连接；而 `あ` 这个字符串则无法连接，因此返回 `nil`。

- `Encoding.default_external`

返回默认的外部编码，这个值会影响 `IO` 类的外部编码，详细内容请参考 19.5 节。

- `Encoding.default_internal`

返回默认的内部编码，这个值会影响 `IO` 类的内部编码，详细内容请参考 19.5 节。

- `Encoding.find(name)`

返回编码名 `name` 对应的 `Encoding` 对象。预定义的编码名由不含空格的英文字母、数字与符号构成。查找编码的时候不区分 `name` 的大小写。

```
p Encoding.find("Shift_JIS")  # => #<Encoding:Shift_JIS>
p Encoding.find("shift_JIS")  # => #<Encoding:Shift_JIS>
```

表 19.1 为预定义的特殊的编码名。

表 19.1 特殊的编码名

名称	意义
<code>locale</code>	根据本地信息决定的编码
<code>external</code>	默认的外部编码
<code>internal</code>	默认的内部编码
<code>filesystem</code>	文件系统的编码

- `Encoding.list`

`Encoding.name_list`

返回 Ruby 支持的编码一览表。`list` 方法返回的是 `Encoding` 对象一览表，`Encoding.name_list` 返回的是表示编码名的字符串一览表，两者的结果都以数组形式返回。

```
p Encoding.list
#=> [#<Encoding:ASCII-8BIT>, #<Encoding:UTF-8>, ...]
p Encoding.name_list
#=> ["ASCII-8BIT", "UTF-8", "US-ASCII", "Big5", ...]
```

- `enc.name`

返回 `Encoding` 对象 `enc` 的编码名。

```
p Encoding.find("shift_jis").name  #=> "Shift_JIS"
```

- `enc.names`

像 EUC-JP、eucJP 这样，有些编码有多个名称。这个方法会返回包含 `Encoding` 对象的名称一览表的数组。只要是这个方法中的编码名称，都可以在通过 `Encoding.find` 方法检索时使用。

```
enc = Encoding.find("Shift_JIS")
p enc.names  #=> ["Shift_JIS", "SJIS"]
```

专栏

**ASCII-8BIT 与字节串**

ASCII-8BIT 是一个特殊的编码，被用于表示二进制数据以及字节串。因此有时候我们也称这个编码为 BINARY。

此外，把字符串对象用字节串形式保存的时候也会用到这个编码。例如，使用 `Array#pack` 方法将二进制数据生成为字符串时，或者使用 `Marshal.dump` 方法将对象序列化后的数据生成为字符串时，都会使用该编码。

下面是用 `Array#pack` 方法，把 IP 地址的 4 个数值转换为 4 个字节的字节串。

```
str = [127, 0, 0, 1].pack("C4")
p str          #=> "\x7F\x00\x00\x01"
p str.encoding # => #<Encoding:ASCII-8BIT>
```

`pack` 方法的参数为字节串化时使用的模式，C4 表示 4 个 8 位的不带符号的整数。执行结果为 4 个字节的字节串，编码为 ASCII-8BIT。

此外，在使用 `open-uri` 库等工具通过网络获取文件时，有时候并不知道字符编码是什么。这时候的编码也默认使用 ASCII-8BIT。

```
# encoding: utf-8
require 'open-uri'
str = open("http://www.example.jp/").read
p str.encoding #=> #<Encoding:ASCII-8BIT>
```

即使是编码为 ASCII-8BIT 的字符串，实际上也还是正常的字符串，只要知道字符编码，就可以使用 `force_encoding` 方法。这个方法并不会改变字符串的值（二进制数据），而只是改变编码信息。

```
# encoding: utf-8
require 'open-uri'
str = open("http://www.example.jp/").read
str.force_encoding("Windows-31J")
p str.encoding #=> #<Encoding:Windows-31J>
```

这样以来，我们就可以把 ASCII-8BIT 的字符串当作 Windows-31J 字符串来处理了。

使用 `force_encoding` 方法时，即使指定了不正确的编码，也不会马上产生错误，而是在对该字符串进行操作的时候才会产生错误。检查编码是否正确，可以用 `valid_encoding?` 方法，不正确时则返回 `false`。

```
str = "こんにちは"
str.force_encoding("US-ASCII") #=> 不会产生错误
str.valid_encoding?           #=> false
str + "みなさん"             #=> Encoding::CompatibilityError
```

## 19.4 正则表达式与编码

与字符串同样，正则表达式也有编码信息。

正则表达式的编码即其匹配字符串的编码。例如，用 EUC-JP 的正则表达式对象去匹配 UTF-8 字符串时就会产生错误，反之亦然。

```
# encoding: EUC-JP
a = "\u3042\u3044"
p /あ/ =~ a    #=> incompatible encoding regexp match
               #=> (EUC-JP regexp with UTF-8 string)
               #=> (Encoding::CompatibilityError)
```

通常情况下，正则表达式字面量的编码与代码的编码是一样的。指定其他编码的时候，可使用 `Regexp` 类的 `new` 方法。在这个方法中，表示模式第 1 个参数的字符串编码，就是该正则表达式的编码。

```
str = "模式".encode("EUC-JP")
re = Regexp.new(str)
p re.encoding # => #<Encoding:EUC-JP>
```

## 19.5 IO 类与编码

使用 `IO` 类进行输入 / 输出操作时编码也非常重要。接下来，我们就向大家介绍一下 `IO` 与编码的相关内容。

### 19.5.1 外部编码与内部编码

每个 `IO` 对象都包含有外部编码与内部编码两种编码信息。外部编码指的是作为输入 / 输出对象的文件、控制台等的编码，内部编码指的是 Ruby 脚本中的编码。`IO` 对象的编码的相关方法如表 19.2 所示。

表 19.2 与 `IO` 类编码相关的方法

方法名	意义
-----	----

<code>IO#external_encoding</code>	返回 <code>IO</code> 的外部编码
<code>IO#internal_encoding</code>	返回 <code>IO</code> 的内部编码
<code>IO#set_encoding</code>	设定 <code>IO</code> 的编码

没有明确指定编码时，`IO` 对象的外部编码与内部编码各自使用其默认值 `Encoding.default_external`、`Encoding.default_internal`。默认情况下，外部编码会基于各个系统的本地信息设定，内部编码不设定。Windows 环境下的编码信息如下所示。

```
p Encoding.default_external    #=> #<Encoding:Windows-31J>
p Encoding.default_internal   #=> nil
File.open("foo.txt") do |f|
  p f.external_encoding       #=> #<Encoding:Windows-31J>
  p f.internal_encoding      #=> nil
end
```

### 19.5.2 编码的设定

在刚才的例子中我们打开了文本文件（`foo.txt`），但 `IO` 对象（`File` 对象）的编码与文件的实际内容其实是没关系的。因为编码原本就只是用来说明如何处理字符的信息，因此对文本文件以外的文件并没有多大作用。

在 17.3 节中说明如何按字节操作文件时，我们介绍了 `IO#seek` 方法与 `IO#read(size)` 方法，这些方法都不受编码影响，对任何数据都可以进行读写操作。`IO#read(size)` 方法读取的字符串的编码为表示二进制数据的 ASCII-8BIT。

设定 `IO` 对象的编码信息，可以通过使用 `IO#set_encoding` 方法，或者在 `File.open` 方法的参数中指定编码来进行。

- `io.set_encoding(encoding)`

`IO#set_encoding` 方法以 "外部编码名:内部编码名" 的形式指定字符串 `encoding`。把外部编码设置为 Shift\_JIS，内部编码设置为 UTF-8 的时候，可以像下面那样设定。

```
$stdin.set_encoding("Shift_JIS=UTF-8")
p $stdin.external_encoding  #=> #<Encoding:Shift_JIS>
p $stdin.internal_encoding #=> #<Encoding=UTF-8>
```

- `File.open(file, "mode:encoding")`

为了在打开文件 `file` 时通过 `File.open` 方法指定编码 `encoding`，可以在第二个参数中指定 `mode` 的后面用冒号（`:`）分割，并按顺序指定外部编码以及内部编码（内部编码可省略）。

```
# 指定外部编码为UTF-8
File.open("foo.txt", "w:UTF-8")

# 指定外部编码为Shift_JIS
# 指定内部编码为UTF-8
File.open("foo.txt", "r:Shift_JIS=UTF-8")
```

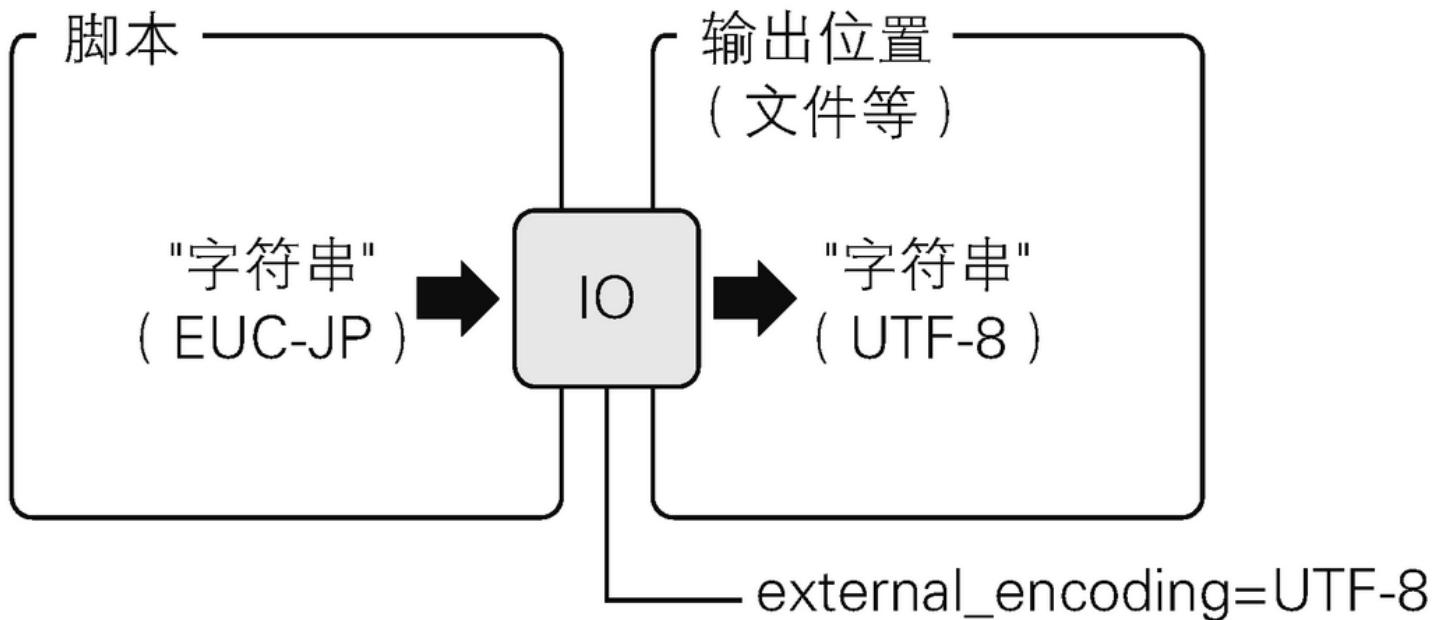
### 19.5.3 编码的作用

接下来，我们来看看 `IO` 对象中设定的编码信息是如何工作的。

- `输出时编码的作用`

外部编码影响 `IO` 的写入（输出）。在输出的时候，会基于每个字符串的原有编码和 `IO` 对象的外部编码进行编码的转换（因此输出用的 `IO` 对象不需要指定内部编码）。

如果没有设置外部编码，或者字符串的编码与外部编码一致，则不会进行编码的转换。在需要进行转换的时候，如果输出的字符串的编码不正确（比如实际上是日语字符串，但编码却是中文），或者是无法互相转换的编码组合（例如用于日语与中文的编码），这时程序就会抛出异常。



- 外部编码没有设定时不会进行转换
- 把字符串的编码转换为外部编码
- 无法转换时抛出异常

图 19.1 输出时与编码相关的行为

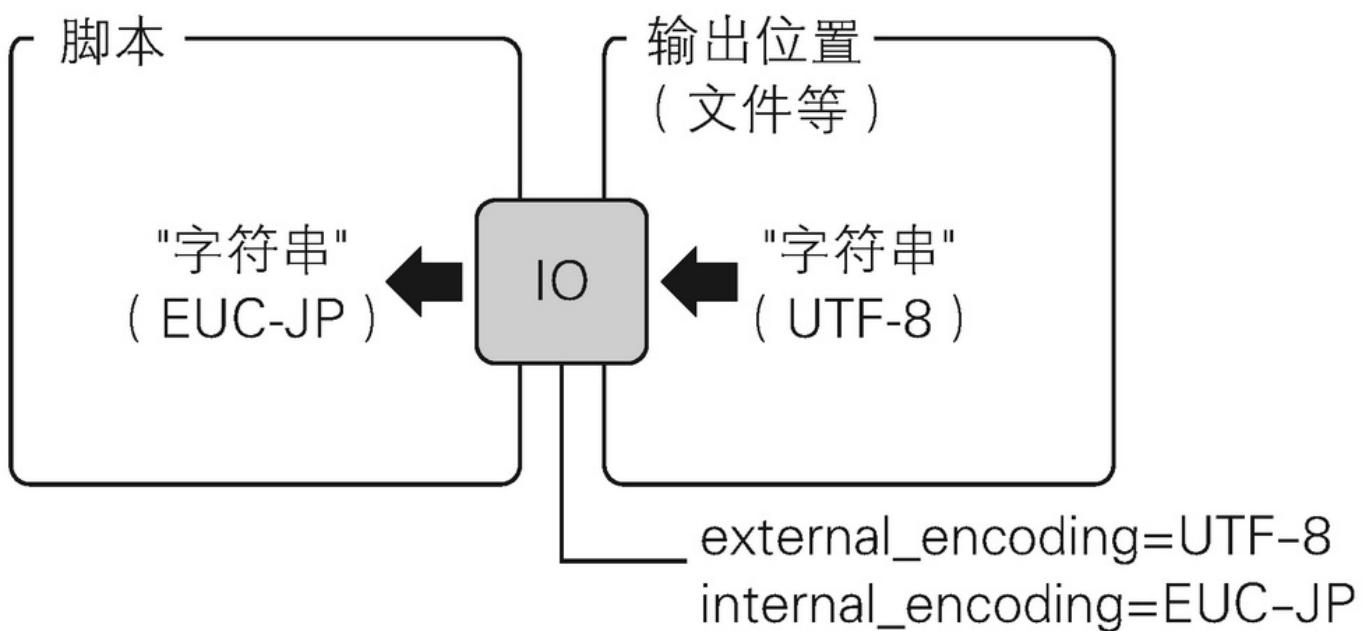
• 输入时编码的作用

`Io` 的读取（输入）会稍微复杂一点。首先，如果外部编码没有设置，则会使用 `Encoding.default_external` 的值作为外部编码。

设定了外部编码，但内部编码没设定的时候，则会将读取的字符串的编码设置为 `Io` 对象的外部编码。这种情况下并不会进行编码的转换，而是将文件、控制台输入的数据原封不动地保存为 `String` 对象。

最后，外部编码和内部编码都设定的时候，则会执行由外部编码转换为内部编码的处理。输入与输出的情况一样，在编码转换的过程中如果数据格式或者编码组合不正确，程序都会抛出异常。

大家或许会感觉有点复杂，其实只要使用的环境与实际使用的数据的编码一致，我们就不需要考虑编码的转换。另外一方面，如果执行环境与数据的编码不一致，那么我们就需要在程序里有意识地处理编码问题。



- 内部编码没设定时把外部编码设定给字符串
- 外部编码没设定时使用`Encoding.default_external`
- 设定了内部编码时则从外部编码进行转换
- 无法转换时抛出异常

图 19.2 输入时与编码相关的行为

专栏

#### UTF8-MAC 编码

在 Mac OS X 中，文件名中如果使用了浊点或者半浊点字符<sup>4</sup>，有时候就会产生一些奇怪的现象。

例如，创建文件 `ルビー.txt` 并执行下面的程序，可以发现，预计执行结果应该为 `found.`，但实际结果却是 `not found.`。

#### 代码清单 utf8mac.rb

```
# encoding: utf-8
Dir.glob("*.txt") do |filename|
  if filename == "ルビー.txt"
    puts "found."; exit
  end
end
puts "not found."
```

#### 执行示例

```
> touch ルビー.txt
> ruby utf8mac.rb
not found.
```

另一方面，执行以下脚本，这次会输出 `found.`。

#### 代码清单 utf8mac\_fix.rb

```
# encoding: utf-8
Dir.glob("*.txt") do |filename|
  if filename.encode("UTF8-MAC") == "ルビー.txt".encode("UTF8-MAC")
    puts "found."; exit
  end
end
puts "not found."
```

#### 执行示例

```
> touch ルビー.txt
```

```
> ruby utf8mac_fix.rb  
found.
```

这是由于 Mac OS X 中的文件系统使用的编码不是 UTF-8，而是一种名为 UTF8-MAC（或者叫 UTF-8-MAC）的编码的缘故。

那么，UTF8-MAC 是什么样的编码呢。我们通过下面的例子来看一下。

#### 代码清单 utf8mac\_str.rb

```
# encoding: utf-8  
str = "𠂇"  
puts "size: #{str.size}"  
p str.each_byte.map{|b| b.to_s(16)}  
puts "size: #{str.encode("UTF8-MAC").size}"  
p str.encode("UTF8-MAC").each_byte.map{|b| b.to_s(16)}
```

#### 执行示例

```
> ruby utf8mac_str.rb  
size: 1  
["e3", "83", "93"]  
size: 2  
["e3", "83", "92", "e3", "82", "99"]
```

本例表示的是在 UTF-8 和 UTF8-MAC 这两种编码方式的情况下，分别以 16 进制的形式输出字符串 “𠂇” 的长度以及各个字节的值。从结果中我们可以看出，UTF-8 时的值为“ec,83,93”，UTF8-MAC 时则是“e3,83,92,e3,82,99”。而转换为 UTF8-MAC 后，字符串的长度也变为了两个字符。

在 UTF8-MAC 中，字符𠂇（Unicode 中为 U+30D3）会分解为字符𠂇（U+30D2）与浊点字符（U+3099）两个字符。用 UTF-8 表示则为 8392E3 与 8299E3 两个字节串，因此就得到了之前的结果。

像这样，如果把 Mac OS X 的文件系统当作是普通的 UTF-8 看待，往往就会有意料之外的事情发生。在操作日语文件、目录时务必注意这个问题 5。

4即日语字母右上角的两个小点和小圆圈。——译者注

5除了日语字母外，一些“含附加符号”的字母也会做同样的处理。例如，字符“ü”会拆分为字符“u”与字符“”的组合。虽然中文字符中一般很少有“附加符号”的情况，但在处理中文字符以外的字符时需要小心这个“陷阱”。——译者注

## 练习题

1. 定义 `to_utf8(str_gbk, str_gb2312)` 方法，连接 GBK 字符串 `str_gbk` 以及 GB2312 字符串 `str_gb2312`，并将连接后的字符串的编码转换为 UTF-8 后返回。
2. 创建编码为 GBK 的文本文件，文件内容为“你好”，定义一个脚本，读取该文件并将其按 UTF-8 编码方式输出。
3. 请找出 UTF-8 字符串 `str`，要求其 `str.encode("GB18030")` 与 `str.encode("GBK")` 的结果不一样。
4. 在刚才的专栏的第二个脚本中，比较时双方都转换为了 UTF8-MAC。请修改 `if` 语句的条件，使 `ルヒー.txt` 在比较时可以保持 UTF-8 编码的形式。

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

# 第 20 章 Time 类与 Date 类

在本章中，我们将会介绍操作时间的 `Time` 类、以及操作日期的 `Date` 类。

- **Time 类与 Date 类**

对 `Time` 类以及 `Date` 类的概要进行说明

- **时间、日期的获取**

介绍获取当前时间、日期的方法。

- **时间、日期的计算**

介绍时间、日期的比较及运算方法。

- **转换为字符串**

介绍把时间、日期转换为字符串的方法。

- **解析字符串**

介绍把表示时间、日期的字符串分别转换为 `Time` 对象、`Date` 对象的方法。

## 20.1 Time 类与 Date 类

`Time` 类用于表示时间。时间除了表示年月日时分秒的信息以外，还包含了表示地域时差的时区（time zone）信息。例如我们可以计算中国当前时间是国际协调时间的几点。

`Date` 类只用于表示年月日。因此，相对于 `Time` 类以秒为单位计算时间，`Date` 类则是以天为单位进行计算的。`Date` 类还可以求下个月的同一天、本月末等日期。

`Time` 类与 `Date` 类表示时间、日期时并没有什么特别限制（前提是现在的日历能一直用下去，甚至“西历 100 亿年”这样的时间、日期也都是可以表示的）。但实际文件的时间戳、程序的运行时间等系统内的时间、以及数据库中的时间类型数据等的情况下，有时候就会受到执行环境的限制。

## 20.2 时间的获取

- `Time.new`

`Time.now`

通过 `Time.new` 方法或者 `Time.now` 方法获取表示当前时间的 `Time` 对象。

```
p Time.new      #=> 2013-03-30 03:06:00 +0900
sleep 1         #=> 等待1 秒
p Time.now      #=> 2013-03-30 03:06:01 +0900
```

- `t.year`

`t.month`

`t.day`

也可以获取时间对象中的年、月、日。

```
t = Time.now
p t          #=> 2013-03-30 03:07:13 +0900
p t.year    #=> 2013
p t.month   #=> 3
p t.day     #=> 30
```

表 20.1 列举了时间的相关方法。

表 20.1 时间的相关方法

方法名	意义
<code>year</code>	年
<code>month</code>	月
<code>day</code>	日
<code>hour</code>	时

<code>min</code>	分
<code>sec</code>	秒
<code>usec</code>	秒以下的位数（以毫秒为单位）
<code>to_i</code>	从 1970 年 1 月 1 日到当前时间的秒数
<code>wday</code>	一周中的第几天（0 表示星期天）
<code>mday</code>	一个月中的第几天（与 <code>day</code> 方法一样）
<code>yday</code>	一年中的第几天（1 表示 1 月 1 日）
<code>zone</code>	时区（ <code>JST</code> 等）

- `Time.mktime(year[, month[, day[, hour [, min[, sec[, usec]]]]]])`

通过 `Time.mktime` 方法可以根据指定时间获取 `Time` 对象。

```
t = Time.mktime(2013, 5, 30, 3, 11, 12)
p t      #=> 2013-05-30 03:11:12 +900
```

文件的创建时间、更新时间等也都能以 `Time` 对象的形式获取。详情请参考 18.3 节。

## 20.3 时间的计算

`Time` 对象之间可以互相比较、运算。

```
t1 = Time.now
sleep(10)    # 等10 秒
t2 = Time.now
p t1 < t2    #=> true
p t2 - t1    #=> 10.005073
```

还可以增加或减少 `Time` 对象的秒数。

```
t = Time.now
p t                  #=> 2013-03-30 03:11:44 +0900
t2 = t + 60 * 60 * 24 #=> 增加24 小时的秒数
p t2                #=> 2013-03-31 03:11:44 +0900
```

## 20.4 时间的格式

- `t.strftime(format)`  
`t.to_s`

通过 `Time#strftime` 方法可以把时间转换为遵循某种格式的字符串。表 20.2 为格式 (`format`) 中可以使用的字符串。

表 20.2 `Time#strftime` 中的格式字符串

格式	意义与范围
<code>%A</code>	星期的名称（ <code>Sunday</code> 、 <code>Monday</code> .....）
<code>%a</code>	星期的缩写名称（ <code>Sun</code> 、 <code>Mon</code> .....）
<code>%B</code>	月份的名称（ <code>January</code> 、 <code>February</code> .....）
<code>%b</code>	月份的缩写（ <code>Jan</code> 、 <code>Feb</code> .....）
<code>%c</code>	日期与时间
<code>%d</code>	日（01 ~ 31）
<code>%H</code>	24 小时制（00 ~ 23）
<code>%I</code>	12 小时制（01 ~ 12）
<code>%j</code>	一年中的天（001 ~ 366）

%M	分 (00 ~ 59)
%m	表示月的数字 (01 ~ 12)
%p	上午或下午 (AM、PM)
%S	秒 (00 ~ 60)
%U	表示周的数字。以星期天为一周的开始 (00 ~ 53)
%W	表示周的数字。以星期一为一周的开始 (00 ~ 53)
%w	表示星期的数字。0 表示星期天 (0 ~ 6)
%X	时间
%x	日期
%Y	表示西历的数字
%y	西历的后两位 (00 ~ 99)
%z	时区 (JST 等)
%Z	时区 (+0900 等)
%%	原封不动地输出 %

例如，`Time#to_s` 方法得到的字符串格式与 `"%Y-%m-%d %H:%M:%S %z"` 是等价的。

```
t = Time.now
p t.to_s                      #=> 2013-03-30 03:13:14 +0900
p t.strftime("%Y-%m-%d %H:%M:%S %z") #=> 2013-03-30 03:13:14 +0900
```

**备注** `Time#strftime` 方法的格式是与平台相关的，不同平台下的执行结果可能不一样。例如，在 Windows 中，`"%z"` 的执行结果会显示“中国标准时间”。

- `t.rfc2822`

通过 `Time#rfc2822` 方法可以生成符合电子邮件头部信息中的 Date : 字段格式的字符串。在互联网的相关文档 RFC (Request For Comments) 中，有一个关于电子邮件形式定义的 RFC 2822 文档，`rfc2822` 这个方法名就来自于此。使用这个方法前，需要预先通过 `require "time"` 引用 `time` 库。

```
require "time"

t = Time.now
p t.rfc2822      #=> "Sat, 30 Mar 2013 03:13:34 +0900"
```

- `t.iso8601`

通过 `Time#iso8601` 方法生成符合 ISO 8601 国际标准的时间格式的字符串。使用这个方法时也需要引用 `time` 库。

```
require "time"

t = Time.now
p t.iso8601    #=> "2013-03-30T03:13:34+09:00"
```

## 20.5 本地时间

世界各地都有时差。大家的计算机中也设有时区，一般计算机中的时间都是根据时区来设定的。

- `t.utc`  
`t.localtime`

我们可以用 `Time#utc` 方法把 `Time` 对象的时区变更为国际协调时间 (UTC)。反之，用 `Time#localtime` 方法则可以把 UTC 变更为本地时间。

```
t = Time.now
p t      #=> 2013-03-30 03:15:19 +0900
t.utc
p t      #=> 2013-03-29 18:15:19 UTC
t.localtime
p t      #=> 2013-03-30 03:15:19 +0900
```

## 20.6 从字符串中获取时间

可以将以字符串形式表示的时间转换为 `Time` 对象。

- `Time.parse(str)`

通过使用 `require "time"`，我们就可以使用 `Time.parse` 方法，来操作以字符串形式表现的时间。`Time.parse` 方法会解析参数字符串 `str`，返回对应的 `Time` 对象。

`Time.parse` 方法除了可以返回与 `Time#to_s` 方法相同的格式，还可以返回 "yyyy/mm/dd" 等多种格式。

```
require "time"

p Time.parse("Sat Mar 30 03:54:15 UTC 2013")
#=> 2013-03-30 03:54:15 UTC
p Time.parse("Sat, 30 Mar 2013 03:54:15 +0900")
#=> 2013-03-30 03:54:15 +0900
p Time.parse("2013/03/30")
#=> 2013-03-30 00:00:00 +0900
p Time.parse("2013/03/30 03:54:15")
#=> 2013-03-30 03:54:15 +0900
p Time.parse("H25.03.31")
#=> 2013-03-31 00:00:00 +0900
p Time.parse("S48.9.28")
#=> 1973-09-28 00:00:00 +0900
```

## 20.7 日期的获取

`Date` 类用于处理不包含时间的日期。使用 `Date.today` 方法可以得到表示当前日期的 `Date` 对象。使用 `Date` 类需要引用 `date` 库。

```
require "date"

d = Date.today
puts d      #=> 2013-03-30
```

与 `Time` 类一样，日期也有其相关的方法。

```
require "date"

d = Date.today
p d.year      # 年 => 2013
p d.month     # 月 => 3
p d.day       # 日 => 30
p d.wday      # 一周中的第几天 (0 表示星期天)      => 6
p d.mday      # 一个月中的第几天 (与 day 方法一样) => 30
p d.yday      # 一年中的第几天 (1 表示 1 月 1 日)   => 89
```

还可以用指定日期生成 `Date` 对象。

```
require "date"

d = Date.new(2013, 3, 30)
puts d      #=> 2013-03-30
```

`Date` 类有一个特点是，可以对月末的日期做-1 处理 (-2 表示月末的前一天)。当然也可以应对闰年。

```
require "date"

d = Date.new(2013, 2, -1)
puts d      #=> 2013-02-28

d = Date.new(2016, 2, -1)
puts d      #=> 2016-02-29
```

## 20.8 日期的运算

`Date` 对象之间的运算以天为单位。因此，`Date` 对象之间进行减法运算时，返回的是两者之间的天数。日期减法运算的结果不是整数，而是 `Rational` 对象。此外，还可以将 `Date` 对象与整数进行加法、减法等运算，这时会返回该对象前后的日期。

```
require "date"

d1 = Date.new(2013, 1, 1)
```

```
d2 = Date.new(2013, 1, 4)
puts d2 - d1      #=> 3/1 (3 天的意思)

d = Date.today
puts d           #=> 2013-03-30
puts d + 1       #=> 2013-03-31
puts d + 100     #=> 2013-07-08
puts d - 1       #=> 2013-03-29
puts d -100      #=> 2012-12-20
```

通过使用 `>>` 运算符，我们就可以获取后一个月相同日期的 `Date` 对象。同理，使用 `<<` 运算符得到的是表示前一个月相同日期的 `Date` 对象。如果该月中没有相同的日期（例如 2 月 30 日），则会返回月末的日期。

```
require "date"

d = Date.today
puts d           #=> 2013-03-30
puts d >> 1     #=> 2013-04-30
puts d >> 100    #=> 2021-07-30
puts d << 1     #=> 2013-02-28
puts d << 100    #=> 2004-11-30
```

## 20.9 日期的格式

与 `Time` 类一样，通过 `strftime` 方法也可以将日期按指定的格式转换为字符串。但结果中时间的部分会全部变为 0。

```
require "date"

t = Date.today
p t.strftime("%Y/%m/%d %H:%M:%S")
#=> "2013/03/30 00:00:00"
p t.strftime("%a %b %d %H:%M:%S %Z %Y")
#=> "Sat Mar 30 00:00:00 +00:00 2013"
p t.to_s      #=> "2013-03-30"
```

## 20.10 从字符串中获取日期

使用 `Date.parse` 方法可以将字符串转换为日期。这个方法可以应对多种日期格式。

```
require "date"

puts Date.parse("Sat Mar 30 03:50:12 JST 2013")  #=> 2013-03-30
puts Date.parse("H25.05.30")                      #=> 2013-05-30
puts Date.parse("S48.9.28")                        #=> 1973-09-28
```

## 练习题

1. 定义 `cparsedate` 方法，把“2013 年 5 月 30 日下午 8 点 17 分 50 秒”这种使用了“年月日时分秒”的字符串转换为 `Time` 对象。
2. 定义 `ls_t` 方法，把指定目录下的文件按时间顺序排列，就像 Unix 的 `ls -t` 命令那样。这个方法只有一个参数。

### `ls_t( 目录名 )`

将指定目录下的文件名按照时间从小到大的顺序排列并输出。

3. 使用 `Date` 类获取本月中每天所对应的星期，并按以下日历格式输出结果。

May 2013
Su Mo Tu We Th Fr Sa
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

# 第 21 章 Proc 类

在本章中，我们将介绍 `Proc` 类相关的内容。

- **Proc** 类是什么

介绍 `Proc` 类是什么、以及创建 `Proc` 对象的几种方法。

- **Proc** 对象的特征

`Proc` 对象具有部分程序的特质，并不是普通的数据，这里将会介绍 `Proc` 类的相关特质。

- **Proc** 类的实例方法

介绍 `Proc` 类的实例方法。

## 21.1 Proc 类是什么

所谓 `Proc`，就是使块对象化的类。`Proc` 与块的关系非常密切，在第 11 章中我们也介绍过 `Proc` 类。请大家结合第 11 章的内容，一起学习本章。

下面，我们来看看如何创建与执行 `Proc` 对象。

- `Proc.new(...)`

```
proc{...}
```

创建 `Proc` 对象的典型方法是通过 `Proc.new` 方法，或者对 `proc` 方法指定块。

```
hello1 = Proc.new do |name|
  puts "Hello, #{name}."
end

hello2 = proc do |name|
  puts "Hello, #{name}."
end

hello1.call("World")    #=> Hello, World.
hello2.call("Ruby")     #=> Hello, Ruby.
```

利用 `Proc.new` 方法，或者对 `proc` 方法指定块，都可以创建代表块的 `Proc` 对象。

通过调用 `Proc#call` 方法执行块。调用 `Proc#call` 方法时的参数会作为块变量，块中最后一个表达式的值则为 `Proc#call` 的返回值。`Proc#call` 还有一个名称叫 `Proc#[]`。

```
# 判断西历的年是否为闰年的处理
leap = Proc.new do |year|
  year % 4 == 0 && year % 100 != 0 || year % 400 == 0
end

p leap.call(2000)    #=> true
p leap[2013]         #=> false
p leap[2016]         #=> true
```

将块变量设置为 `|*args|` 的形式后，就可以像方法参数一样，以数组的形式接收可变数量的参数。

```
double = Proc.new do |*args|
  args.map{|i| i * 2 }      # 所有元素乘两倍
end

p double.call(1, 2, 3)    #=> [2, 3, 4]
p double[2, 3, 4]         #=> [4, 6, 8]
```

除此以外，定义普通方法时可使用的参数形式，如默认参数、关键字参数等，几乎都可以被用于块变量的定义，并被指定给 `Proc#call` 方法。关于方法定义的参数指定，请参考第 7 章。

### 21.1.1 lambda

`Proc.new`、`proc` 等有另外一种写法叫 `lambda`。与 `Proc.new`、`proc` 一样，`lambda` 也可以创建 `Proc` 对象，但通过 `lambda` 创建的 `Proc` 的行为会更接近方法。

第一个不同点是，`lambda` 的参数数量的检查更加严密。对用 `Proc.new` 创建的 `Proc` 对象调用 `call` 方法时，`call` 方法的参数数量与块变量的数量可以不同。但通过 `lambda` 创建 `Proc` 对象时，如果参数数量不正确，程序就会产生错误。

```

prc1 = Proc.new do |a, b, c|
  p [a, b, c]
end
prc1.call(1, 2)    #=> [1, 2, nil]

prc2 = lambda do |a, b, c|
  p [a, b, c]
end
prc2.call(1, 2)    #=> 错误 (ArgumentError)

```

第二个不同点是，`lambda` 可以使用 `return` 将值从块中返回。请看代码清单 21.1。`power_of` 方法会利用参数 `n` 返回“计算 `x` 的 `n` 次幂的 `Proc` 对象”。请注意，返回值并不是数值，而是进行运算的 `Proc` 对象。调用 `power_of(3)` 后，结果就会得到 `call` 方法参数值的 3 次幂的 `Proc` 对象。从 `lambda` 中返回值时使用了 `return`，这里的 `return` 会将 `lambda` 中的值返回。

#### 代码清单 21.1 power\_of.rb

```

def power_of(n)
  lambda do |x|
    return x ** n
  end
end

cube = power_of(3)
p cube.call(5)  #=> 125

```

接下来，我们尝试用 `Proc.new` 方法改写代码清单 21.1。使用 `Proc.new` 方法时，在块中使用 `return` 后，程序就会跳过当前执行块，直接从创建这个块的方法返回。在本例中，即虽然块内的 `return` 应该从 `power_of` 方法返回，但由于程序运行时 `power_of` 方法的上下文会消失，因此程序就会出现错误。

```

def power_of(n)
  Proc.new do |x|
    return x ** n
  end
end

cube = power_of(3)
p cube.call(5)  #=> 错误 (LocalJumpError)

```

不是 `lambda` 的普通块中的 `return`，会从正在执行循环的方法返回。代码清单 21.2 中的 `prefix` 方法会比较参数 `ary` 中的元素是否与 `obj` 相等，相等就返回在此之前的所有元素，不相等则返回空数组。第 6 行中的 `return` 并不会从块返回，而是跳过块，并作为 `prefix` 方法整体的返回值返回。

#### 代码清单 21.2 prefix.rb

```

1: def prefix(ary, obj)
2:   result = []          # 初始化结果数组
3:   ary.each do |item|   # 逐个检查元素
4:     result << item    # 将元素追加到结果数组中
5:     if item == obj    # 如果元素与条件一致
6:       return result   # 返回结果数组
7:     end
8:   end
9:   return result        # 所有元素检查完毕的时候
10: end
11:
12: prefix([1, 2, 3, 4, 5], 3)  #=> [1, 2, 3]

```

`break` 被用于控制迭代器的行为。这个命令会向接收块的方法的调用者返回结果值。如下所示，`break []` 会马上终止 `Array#collect` 方法，并将空数组作为 `collect` 方法的整体的返回值返回。

```

[:a, :b, :c].collect do |item|
  break []
end

```

注 用 `Proc.new` 方法或者 `proc` 方法创建的 `Proc` 对象的情况下，由于这些方法都接收块，在调用 `Proc#call` 方法的时候并没有适当的返回对象，因此就会发生错误。而 `lambda` 的情况下则与 `return` 一样，将值返回给 `Proc#call` 方法。另一方面，由于 `next` 方法的作用在于中断 1 次块的执行，因此无论如何创建 `Proc` 对象，都可以将值返回给 `call` 方法。

`lambda` 有另外一种写法——“`->( 块变量 ){ 处理 }`”。块变量在 `{ ~ }` 之前，看上去有点像函数。使用 `->` 的时候，我们一般会使用 `{ ~ }` 而不是 `do ~ end`。

```

square = ->(n){ return n ** 2 }
p square[5]  #=> 25

```

## 21.1.2 通过 Proc 参数接收块

在调用带块的方法时，通过 `Proc` 参数的形式指定块后，该块就会作为 `Proc` 对象被方法接收。代码清单 21.3 是我们在第 11 章中介绍过的例子。在 `total2` 方法中，调用 `total2` 方法时指定的块，可以作为 `Proc` 对象从变量 `block` 中获取。

### 代码清单 21.3 total2.rb

```
def total2(from, to, &block)
  result = 0          # 合计值
  from.upto(to) do |num|  # 处理从 from 到 to 的值
    if block          # 如果有块的话
      result +=       # 累加经过块处理的值
      block.call(num)
    else              # 如果没有块的话
      result += num  # 直接累加
    end
  end
  return result        # 返回方法的结果
end

p total2(1, 10)           # 从 1 到 10 的和 => 55
p total2(1, 10){|num| num ** 2} # 从 1 到 10 的 2 次冥的和 => 385
```

## 21.1.3 to\_proc 方法

有些对象有 `to_proc` 方法。在方法中指定块时，如果以 `&` 及对象的形式传递参数，对象 `.to_proc` 就会被自动调用，进而生成 `Proc` 对象。

其中，`Symbol#to_proc` 方法是比较典型的，并且经常被用到。例如，对符号 `:to_i` 使用 `Symbol#to_proc` 方法，就会生成下面那样的 `Proc` 对象。

```
Proc.new{|arg| arg.to_i }
```

这个对象在什么时候使用呢？例如，把数组的所有元素转换为数值类型时，一般的做法如下：

### 执行示例

```
>> %w(42 39 56).map{|i| i.to_i }
=> [42, 39, 56]
```

上述代码还可以像下面这样写：

### 执行示例

```
>> %w(42 39 56).map(&:to_i)
=> [42, 39, 56]
```

按照类名排序的程序，也可以写成：

### 执行示例

```
>> [Integer, String, Array, Hash, File, IO].sort_by(&:name)
=> [Array, File, Hash, IO, Integer, String]
```

熟悉这样的写法可能需要一定的时间，但这种写法不仅干净利索，而且意图明确。

## 21.2 Proc 的特征

虽然 `Proc` 对象可以作为匿名函数或方法使用，但它并不只是单纯的对象化。请看代码清单 21.4。

### 代码清单 21.4 counter\_proc.rb

```
1: def counter
2:   c = 0          # 初始化计数器
3:   Proc.new do    # 每调用 1 次 call 方法，计数器加1
4:     c += 1        # 返回加 1 后的 Proc 对象
5:   end
6: end
7:
8: # 创建计数器 c1 并计数
9: c1 = counter
10: p c1.call      #=> 1
11: p c1.call      #=> 2
12: p c1.call      #=> 3
13:
```

```

14: # 创建计数器 c2 并计数
15: c2 = counter      # 创建计数器c2
16: p c2.call        #=> 1
17: p c2.call        #=> 2
18:
19: # 再次用 c1 计数
20: p c1.call        #=> 4

```

第 1 行到第 6 行为 `counter` 方法的定义。该方法首先把作为计数器的本地变量 `c` 初始化为 0。然后每调用 1 次 `Proc#call` 方法，就将计数器加 1，并返回该 `Proc` 对象。在第 9 行中，调用 `counter` 方法，将 `Proc` 对象赋值给 `c1`。可以看到，`c1` 调用 `call` 方法后，`proc` 对象引用的本地变量 `c` 开始计数了。在第 15 行中，以同样的方法创建新的计数器，之后计数器被重置。在最后的第 20 行中，再次调用最初创建的 `c1` 的 `call` 方法，计数器开始接着之前的结果计数。

通过这个例子我们可以看出，变量 `c1` 与变量 `c2` 引用的 `Proc` 对象，是分别保存、处理调用 `counter` 方法时初始化的本地变量的。与此同时，`Proc` 对象也会将处理内容、本地变量的作用域等定义块时的状态一起保存。

像 `Proc` 对象这样，将处理内容、变量等环境同时进行保存的对象，在编程语言中称为闭包（closure）。使用闭包后，程序就可以将处理内容和数据作为对象来操作。这和在类中描述处理本身、在实例中保存数据本质上是一样的，只是从写程序的角度来看，使用类的话当然也就意味着可以使用更多的功能。

就像刚才的计数器的例子那样，`Proc` 对象可被用来对少量代码实现的功能做对象化处理。另外，由于 Ruby 中大量使用了块，因此在有一定规模的程序开发中，我们就难免会使用到 `Proc` 对象。特别是像调用和传递带块的方法时的方法、通过闭包保存数据等功能，我们都需要透彻理解才行。

## 21.3 Proc 类的实例方法

- `prc.call(args, ...)`
- `prc[args, ...]`
- `prc.yield(args, ...)`
- `prc.(args, ...)`
- `prc === arg`

上述方法都执行 `Proc` 对象 `prc`。

```

prc = Proc.new{|a, b| a + b}
p prc.call(1, 2)    #=> 3
p prc[3, 4]         #=> 7
p prc.yield(5, 6)   #=> 11
p prc.(7, 8)        #=> 15
p prc === [9, 10]   #=> 19

```

由于受到语法的限制，通过 `==` 指定的参数只能为 1 个。大家一定要牢记这个方法会在 `Proc` 对象作为 `case` 语句的条件时使用。因此，在创建这样的 `Proc` 对象时，比较恰当的做法是，只接收一个参数，并返回 `true` 或者 `false`。

下面的例子实现的是，从 1 到 100 的整数中，当值为 3 的倍数时输出 `Fizz`，5 的倍数时输出 `Buzz`，15 的倍数时输出 `Fizz Buzz`，除此以外的情况下则输出该值本身。

```

fizz = proc{|n| n % 3 == 0 }
buzz = proc{|n| n % 5 == 0 }
fizzbuzz = proc{|n| n % 3 == 0 && n % 5 == 0}
(1..100).each do |i|
  case i
  when fizzbuzz then puts "Fizz Buzz"
  when fizz then puts "Fizz"
  when buzz then puts "Buzz"
  else puts i
  end
end

```

- `prc.arity`

返回作为 `call` 方法的参数的块变量的个数。以 `|*args|` 的形式指定块变量时，返回 -1。

```

prc0 = Proc.new{ nil }
prc1 = Proc.new{|a| a }
prc2 = Proc.new{|a, b| a + b }
prc3 = Proc.new{|a, b, c| a + b + c }
prcn = Proc.new{|*args| args }

p prc0.arity    #=> 0
p prc1.arity    #=> 1
p prc2.arity    #=> 2
p prc3.arity    #=> 3
p prcn.arity    #=> -1

```

- **`prc.parameters`**

返回关于块变量的详细信息。返回值为 [ 种类 , 变量名 ] 形式的数组的列表。表 21.1 为表示种类的符号。

表 21.1 `Proc#parameters` 返回的变量种类

符号	意义
<code>:opt</code>	可省略的变量
<code>:req</code>	必需的变量
<code>:rest</code>	以 <code>\*_args_</code> 形式表示的变量
<code>:key</code>	关键字参数形式的变量
<code>:keyrest</code>	以 <code>\*\_args_</code> 形式表示的变量
<code>:block</code>	块

```

prc0 = proc{ nil }
prc1 = proc{|a| a }
prc2 = lambda{|a, b| [a, b] }
prc3 = lambda{|a, b=1, *c| [a, b, c] }
prc4 = lambda{|a, &block| [a, block] }
prc5 = lambda{|a: 1, **b| [a, b] }

p prc0.parameters    #=> []
p prc1.parameters    #=> [[:opt, :a]]
p prc2.parameters    #=> [[:req, :a], [:req, :b]]
p prc3.parameters    #=> [[:req, :a], [:opt, :b], [:rest, :c]]
p prc4.parameters    #=> [[:req, :a], [:block, :block]]
p prc5.parameters    #=> [[:key, :a], [:keyrest, :b]]

```

- **`prc.lambda?`**

判断 `prc` 是否为通过 `lambda` 定义的方法。

```

prc1 = Proc.new{|a, b| a + b}
p prc1.lambda?  #=> false

prc2 = lambda{|a, b| a + b}
p prc2.lambda?  #=> true

```

- **`prc.source_location`**

返回定义 `prc` 的程序代码的位置。返回值为 [ 代码文件名 , 行编号 ] 形式的数组。`prc` 由扩展库等生成，当 Ruby 脚本不存在时返回 `nil`。

代码清单 21.5 `proc_source_location.rb`

```

1: prc0 = Proc.new{ nil }
2: prc1 = Proc.new{|a| a }
3:
4: p prc0.source_location
5: p prc1.source_location

```

#### 执行示例

```

> ruby proc_source_location.rb
[ "proc_source_location.rb", 1 ]
[ "proc_source_location.rb", 2 ]

```

## 练习题

- 仿照 `Array#collect` 方法，定义 `my_collect` 方法。参数为拥有 `each` 方法的对象，并在块中对各元素进行处理。

```

def my_collect(obj, &block)
  (?)
end

ary = my_collect([1, 2, 3, 4, 5]) do |i|
  i * 2

```

```
end
```

```
p ary  #=> [2, 4, 6, 8, 10]
```

2. 确认使用了下述 `Symbol#to_proc` 方法的例子的执行结果。

```
to_class = :class.to_proc
p to_class.call("test")    #=> ??
p to_class.call(123)      #=> ??
p to_class.call(2 ** 1000) #=> ??
```

3. 修改计数器的例子，计算 `call` 方法的参数的合计值。请补充下面 (??) 部分的代码。

```
def accumulator
  total = 0
  Proc.new do
    (??)
  end

  acc = accumulator
  p acc.call(1)    #=> 1
  p acc.call(2)    #=> 3
  p acc.call(3)    #=> 6
  p acc.call(4)    #=> 10
```

参考答案：请到图灵社区本书的“随书下载”处下载 (<http://www.ituring.com.cn/book/1237>)。

## 第 4 部分 动手制作工具

---

Ruby 可以用来干什么？什么样的程序适合用 Ruby？在本部分中，我们会尝试解答这两个问题，并借此让大家体会一下何谓“快乐编程”。

# 第 22 章 文本处理

在本章中，我们会以在第 3 章中创建的 simple\_grep.rb 为基础，来学习文本的一般处理方法。

这里，我们将会创建实现以下功能的脚本。

- 获取 HTML 文件并进行简单的加工
- 查找单词并显示其出现的次数
- 强调查找结果并对输出结果进行加工

## 22.1 准备文本

首先是准备作为处理对象的文本。

### 22.1.1 下载文件

这里，我们以山形浩生翻译的 *The Cathedral and the Bazaar*<sup>1</sup> 为例进行说明，该翻译版本已在网上公开并可自由使用。这是一篇非常著名的有关开源项目（Open Source Project）开发模式的论文，有兴趣的读者不妨一读。

<sup>1</sup>中文简体版名为《大教堂与集市》，由机械工业出版社于 2014 年出版，卫剑钒译。——译者注

*The Cathedral and the Bazaar* 翻译版的 URL 如下所示。

- <http://cruel.org/freeware/cathedral.html>

虽然也可以使用浏览器访问、下载上面的 URL，不过既然已经学习了 Ruby，下面就让我们来试试用 Ruby 下载。

代码清单 22.1 get\_cathedral.rb

```
require "open-uri"
require "nkf"

url = "http://cruel.org/freeware/cathedral.html"
filename = "cathedral.html"

File.open(filename, "w") do |f|
  text = open(url).read
  f.write text          # UTF-8 环境下使用此段代码
  #f.write NKF.nkf("-s", text) # Shift_JIS 环境下（日语 Windows）使用此段代码
end
```

程序在处理日语字符的时候需要注意编码问题<sup>2</sup>。特别是从外部的输入，全部都用一样的编码也无可非议。在日语 Windows 环境中，由于是用 Shift\_JIS 编码传递字符给命令行，因此文件也采用了相同的编码。本例中的 HTML 文件的编码为 UTF-8，因此如果是日语 Windows 环境的话，就需要用 NFK 转换为 Shift\_JIS，然后再用 write 方法输出（请修改并执行代码清单 22.1 中注释了的代码部分）。

<sup>2</sup>中文也同样需要注意。——译者注

### 22.1.2 获取正文

从代码清单 22.1 得到的是用于在浏览器中显示的 HTML 文件。这个文件中有很多像头部、底部这样的我们不需要的部分，因此这里我们只把正文部分抽取出来。

首先必须定义好正文的起始位置与结束位置。而为此就必须先看看 HTML 里的内容。下面，我们来好好研究一下刚才下载的 cathedral.html。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN" "http://www.w3.org/TR/REC-html40/
strict.dtd">
<html lang="ja"><head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <meta name="Author" content="Eric Raymond, YAMAGATA Hiroo">
  |
<hr />
<h2><a name="1">1 伽藍方式とバザール方式</a></h2>
<p>
  Linux
  は破壊的存在なり。インターネットのかほそい糸だけで結ばれた、地球全体に散らばった数千人の開発者たちが片手間にハッキングするだけで、超一流の OS が魔法みたいに編
  |
</p><hr>
<h2><a name="version">バージョンと変更履歴</a></h2>
<p>
  $Id: cathedral-bazaar.sgml,v 1.40 1998/08/11 20:27:29 esr Exp $</p>
  |
```

通过上面的内容可以看出，以“`<h2><a name="1">1 伽藍方式とバザール方式</a></h2>`”开始的行就是正文的开始。

同样，从“`<h2><a name="version"> バージョンと変更履歴 </a></h2>`”这一行开始则为底部，之后的内容与正文无关。

然后，对这两行做上标记，把正文部分摘取出来。

#### 代码清单 22.2 cut\_cathedral.rb

```
1: htmlfile = "cathedral.html"
2: textfile = "cathedral.txt"
3:
4: html = File.read(htmlfile)
5:
6: File.open(textfile, "w") do |f|
7:   in_header = true
8:   html.each_line do |line|
9:     if in_header && /<a name="1"/ !~ line
10:      next
11:    else
12:      in_header = false
13:    end
14:    break if /<a name="version"/ =~ line
15:    f.write line
16:  end
17: end
```

在这个脚本中，以包含字符串 `<a name="1">` 的行作为开始，包含字符串 `<a name = "version">` 的行作为结束，把中间的内容保存到 `catedral.txt` 文件中。

首先，使用 `File.read` 方法读取 HTML 文件的全部内容。

接下来，对 HTML 文件的字符串使用 `each_line` 方法，逐行读取内容并赋值给变量 `line`，然后再将其保存到文件中。不过，在保存之前，需要先把 `in_header` 变量设为 `true`。这个变量可被用于检查正在处理的行是否为头部。第 9 行的 `if` 语句会利用这个变量值进行判断，如果是在头部内，并且正在读取的行不包含 `<a name="1">` 则跳出本次循环。而除此以外的情况下，则表示已经离开了头部部分，因此就要将 `in_header` 设置为 `false`。这样一来，从下个循环开始就不会再执行 `next` 了。

第 14 行中使用了 `if` 修饰符。`break if...` 这样的形式常被用于跳出循环。这种写法的优点在于可以紧凑地书写不太长的 `if` 条件。在这里，程序会判断是否为表示正文结束的行，成功匹配则跳出循环。

接下来是第 15 行，程序能走到这里就证明 `line` 是正文部分，因此，使用 `write` 方法将 `line` 的内容输出到文件。

#### 22.1.3 删除标签

但是输出到文件的正文部分中还残留着 HTML 标签（tag）。虽然即使有 HTML 标签也可以进行文本处理，但在本章中并不需要标签。因此接下来我们就来考虑一下如何把标签删除，以获取纯文本（plain text）格式的文件。

一般情况下，删除 HTML 标签，可以考虑使用解析 HTML 用的类库，不过在本例中，我们只是单纯地通过正则表达式来置换。

#### 代码清单 22.3 cut\_cathedral2.rb

```
1: require 'cgi/util'
2: htmlfile = "cathedral.html"
3: textfile = "cathedral.txt"
4:
5: html = File.read(htmlfile)
6:
7: File.open(textfile, "w") do |f|
8:   in_header = true
9:   html.each_line do |line|
10:     if in_header && /<a name="1"/ !~ line
11:       next
12:     else
13:       in_header = false
14:     end
15:     break if /<a name="version"/ =~ line
16:     line.gsub!(/<[^>]+>/, '')
17:     esc_line = CGI.unescapeHTML(line)
18:     f.write esc_line
19:   end
20: end
```

代码清单 22.3 在代码 22.2 的基础上添加了删除标签的功能，而实际两者只有第 16 行和第 17 行代码是不一样的。

在第 16 行中，用正则表达式 `/<[^>]+>/` 表示标签。由于 HTML 标签是以 `<` 开始，以 `>` 结束的，因此这样就可以匹配标签部分。在第 17 行中，使用

`CGI.escapeHTML` 方法，将 HTML 标签的 `&amp;`、`&lt;` 等转义字符，转换为普通字符 `&`、`<` 等。通过在第 1 行追加 `require 'cgi/util'`，我们就可以使用这个方法了。

做出这样的修改后，我们就可以得到以下文本：

### 1 伽藍方式とバザール方式

Linux

は破壊的存在なり。インターネットのかぼそい糸だけで結ばれた、地球全体に散らばった数千人の開発者たちが片手間にハッキングするだけで、超一流の OS が魔法みたいに編

## 22.2 扩展 simple\_grep.rb：显示次数

接下来，我们来看看 `simple_grep.rb`。这里对第 3 章中的例子稍微做了一点修改（代码清单 22.4）。

### 代码清单 22.4 simple\_grep.rb

```
pattern = Regexp.new(ARGV[0])
filename = ARGV[1]

File.open(filename) do |file|
  file.each_line do |line|
    if pattern =~ line
      print line
    end
  end
end
```

由于我们在 `File.open` 方法中使用了块，因此 `File#close` 就不再需要了，这样一来，程序也清爽了好多。

利用这个程序，我们来调查一下正文中的“伽藍”<sup>3</sup> 以及“バザール”<sup>4</sup> 这两个单词总共出现了多少次。

<sup>3</sup>中文是“大教堂”的意思。——译者注

<sup>4</sup>中文是“集市”的意思。——译者注

### 计算匹配行

由于通过 `simple_grep.rb` 匹配的行会原封不动地显示出来，因此 Mac OS X 或 Linux 的情况下，就可以配合 `wc` 命令<sup>5</sup> 查看文本的行数。

<sup>5</sup>wc 即 word count 的缩写。——译者注

### 执行示例

```
> ruby simple_grep.rb '伽藍' cathedral.txt | wc
   20      79     4352
> ruby simple_grep.rb 'バザール' cathedral.txt | wc
   38     122     8376
```

但是，我们不能确切地说正文中“伽藍”出现了 20 次，这是因为如果 1 行中该单词出现了多次的话，那么只通过计算行数是不能得出正确的结果的。

下面我们来改造这个程序，用 `String#scan` 方法，计算匹配的次数。

### 代码清单 22.5 simple\_scan.rb

```
pattern = Regexp.new(ARGV[0])
filename = ARGV[1]

count = 0
File.open(filename) do |file|
  file.each_line do |line|
    if pattern =~ line
      line.scan(pattern) do |s|
        count += 1
      end
      print line
    end
  end
end
puts "count: #{count}"
```

### 执行示例

```

> ruby simple_scan.rb '伽藍' cathedral.txt
1 伽藍方式とバザール方式
みたいな本当に大規模なツール)は伽藍のように組み立てられなきゃダメで、一人のウィザードか魔術師の小集団が、まったく孤立して慎重に組み立てあげるべ
のほうは、閉じた開発者グループとめったにないリリースとでもっと伽藍的な開発方式を続けたということだった。
count: 21
> ruby simple_scan.rb 'バザール' cathedral.txt
1 伽藍方式とバザール方式
コミュニティはむしろ、いろんな作業やアプローチが渦を巻く、でかい騒がしいバザールに似ているみたいだった（これをまさに象徴しているのが
が意識的に、ぼくがこれまでに記述してきたバザール戦術を用い、それに対して GCC
count: 43

```

结果显示，“伽藍”出现了 21 次，“バザール”出现了 43 次。

如果不需要逐行处理，而只是单纯计算出现次数的话，则有更简单的实现方法（代码清单 22.6）。

#### 代码清单 22.6 simple\_count.rb

```

pattern = Regexp.new(ARGV[0])
filename = ARGV[1]

count = 0
File.read(filename).scan(pattern) do |s|
  count += 1
end
puts "count: #{count}"

```

由于 `String#scan` 方法是对字符串使用的方法，因此，本例中没有使用 `File.open` 方法，而是通过 `File.read` 方法一次性读取了所有内容并将其保存为了字符串。

### 22.3 扩展 simple\_grep.rb：显示匹配的部分

下面，让我们再次回到原来的 `simple_scan.rb` 来继续进行改造。

#### 22.3.1 突出匹配到的位置

虽然显示出了匹配的行，但具体匹配到的位置却很难看清楚。因此下面我们就来试试在显示的时候强调匹配的部分（代码清单 22.7）。

#### 代码清单 22.7 simple\_match.rb

```

1: pattern = Regexp.new(ARGV[0])
2: filename = ARGV[1]
3:
4: count = 0
5: File.open(filename) do |file|
6:   file.each_line do |line|
7:     if pattern =~ line
8:       line.scan(pattern) do |s|
9:         count += 1
10:    end
11:    print line.gsub(pattern){|str| "<<#{str}>>"}
12:  end
13: end
14: puts "count: #{count}"
15:
16: puts "count: #{count}"

```

在第 11 行，使用 `gsub` 方法将原来直接输出变量 `line` 的地方进行转换后再输出。由于对 `gsub` 方法使用块后，匹配部分就可以通过块变量取得，因此这里在匹配部分的前后加上 `<<>>` 并返回。

执行结果如下所示：

#### 执行示例

```

> ruby simple_match.rb '伽藍' cathedral.txt
1 << 伽藍>> 方式とバザール方式
みたいな本当に大規模なツール)は<< 伽藍>> のように組み立てられなきゃダメで、一人のウィザードか魔術師の小集団が、まったく孤立して慎重に組み立てあけ
のほうは、閉じた開発者グループとめったにないリリースとでもっと<< 伽藍>> 的な開発方式を続けたということだった。
count: 21

```

可以看出，与之前相比，匹配部分被突出显示了。

## 22.3.2 显示前后各 10 个字符

然而，在上述 simple\_match.rb 的执行结果中，匹配部分在行中的位置比较分散，这时我们就希望显示效果能更加紧凑些。例如显示匹配部分及其前后各 10 个字符（代码清单 22.8）。

代码清单 22.8 simple\_match2.rb

```
pattern = Regexp.new("(.{10})(+ARGV[0]+)(.{10})")
filename = ARGV[1]

count = 0
File.open(filename) do |file|
  file.each_line do |line|
    line.scan(pattern) do |s|
      puts "#{s[0]}<<#{s[1]}>>#{s[2]}"
      count += 1
    end
  end
end
puts "count: #{count}"
```

正则表达式中的 `{n}` 表示重复之前的模式 n 次。因此，本例中的 `.{10}` 就表示匹配 10 个任意字符。除此以外，还可以用 `{n,m}` 匹配 n 次以上 m 次以下，用 `{n,}` 匹配 n 次以上，用 `{,m}` 匹配 m 次以下。

接下来，我们来看看效果如何。

### 执行示例

```
> ruby simple_match2.rb '伽藍' cathedral.txt
に大規模なツール)は<< 伽藍>> のように組み立てられ
された。静かで荘厳な<< 伽藍>> づくりなんかない—
、それどころかなぜ、<< 伽藍>> 建設者たちの想像を絶
F ツールみたいな、<< 伽藍>> 建築方式にくらべると
|
count: 12
```

通过 `count: 12` 可以看出，出现的次数减少了。这是因为匹配部分前后不够 10 个字符时就不会匹配。另外，即使是 10 个字符，英文数字（ASCII）的字符长度与日语字符也不一样，这就导致了输出结果排列不整齐。因此我们再来修改一下（代码清单 22.9）。

代码清单 22.9 simple\_match3.rb

```
1: pattern = Regexp.new("(.{0,10})(+ARGV[0]+)(.{0,10})")
2: filename = ARGV[1]
3:
4: count = 0
5: File.open(filename) do |file|
6:   file.each_line do |line|
7:     line.scan(pattern) do |s|
8:       prefix_len = 0
9:       s[0].each_char do |ch|
10:         if ch.ord < 128
11:           prefix_len += 1
12:         else
13:           prefix_len += 2
14:         end
15:       end
16:       space_len = 20 - prefix_len
17:       puts "#{space_len}#{s[0]}<<#{s[1]}>>#{s[2]}"
18:       count += 1
19:     end
20:   end
21: end
22: puts "count: #{count}"
```

修改程序第 1 行的正则表达式，将原来的匹配 10 个字符的 `{10}` 的地方，修改为匹配 0 个以上 10 个以下字符的 `{0,10}`。另外，在程序第 9 行以后，使用 `each_char` 方法逐个读取字符，并通过 `ord` 方法获取字符编码的码位。由于码位小于 128 时即为 ASCII 码，这时将长度加 1，除此以外的情况下则加 2，这些都是为了确定空白个数 `space_len` 以确保 20 个字符。然后再在 `s[0]` 之前留出与字符数相应的空白，这样输出结果就整齐多了。

### 执行示例

```
> ruby simple_match3.rb '伽藍' cathedral.txt
1 << 伽藍>> 方式とバザール方式
に大規模なツール)は<< 伽藍>> のように組み立てられ
された。静かで荘厳な<< 伽藍>> づくりなんかない—
```

```
、それどころかなぜ、<< 伽藍>> 建設者たちの想像を絶  
F ツールみたいな、<< 伽藍>> 建築方式にくらべると  
この信念のおかげで、<< 伽藍>> 建設式の開発への関与  
自分が、FSF 式の<< 伽藍>> 建設型開発モデルの問  
一派の開発方針は、<< 伽藍>> 建設の正反対のものだ  
ここに、<< 伽藍>> 建築方式とバザール式  
|  
count: 21
```

可以看出，这次的结果整齐了许多。

### 22.3.3 让前后的字符数可变更

现在我们默认前后的字符只能是 10 个。不过这个数量如果能修改，那就灵活多了。于是，我们再次改造了程序（代码清单 22.10）。

代码清单 22.10 simple\_match4.rb

```
1: len = ARGV[2].to_i
2: pattern = Regexp.new("(.{0,#{len}})(+ARGV[0]+)(.{0,#{len}})")
3: filename = ARGV[1]
4:
5: count = 0
6: File.open(filename) do |file|
7:   file.each_line do |line|
8:     line.scan(pattern) do |s|
9:       prefix_len = 0
10:      s[0].each_char do |ch|
11:        if ch.ord < 128
12:          prefix_len += 1
13:        else
14:          prefix_len += 2
15:        end
16:      end
17:      space_len = len * 2 - prefix_len
18:      puts "#{" " * space_len}#{s[0]}<<#{s[1]}>>#{s[2]}"
19:      count += 1
20:    end
21:  end
22: end
23: puts "count: #{count}"
```

将指定长度的位置换为变量 `len`，可以通过 `ARGV[2]` 将其作为第 3 个参数来指定。

这里，我们指定为 5 个字符，来看看执行结果如何。

#### 执行示例

```
> ruby simple_match4.rb '伽藍' cathedral.txt 5
      1 << 伽藍>> 方式とバザ
ツール) は<< 伽藍>> のように組
かで莊嚴な<< 伽藍>> づくりなん
ろかなぜ、<< 伽藍>> 建設者たち
みたいな、<< 伽藍>> 建築方式に
おかげで、<< 伽藍>> 建設式の開
SF 式の<< 伽藍>> 建設型開発
|
とでもっと<< 伽藍>> 的な開発方
count: 21
```

从结果可以看出，前后的字符串都变为了 5 个字符。

正如本章所演示的那样，制作工具的时候，有效做法是，首先由简单的开始，然后再慢慢地接近目标功能。即使是很难马上解决的问题，我们也可以将其细化，来逐个击破。

# 第 23 章 检索邮政编码

正式进行数据检索时一般都需要用到一些专门的框架或者应用程序，但如果只是对手头的数据加以整理以方便处理的话，根据实际情况做个简易的小工具也未尝不可。在本章中，作为 Ruby 的应用示例，我们来看看如何检索邮政编码。

## 23.1 获取邮政编码

日本的邮政编码可在邮局的官方网站上下载。

- 邮政编码检索：<http://www.post.japanpost.jp/zipcode>
- 邮政编码下载：<http://www.post.japanpost.jp/zipcode/dl/kogaki-zip.html>
- 邮政编码说明：<http://www.post.japanpost.jp/zipcode/dl/readme.html>

下载后的文件格式是 zip，用 zip 工具解压后就可以得到格式为 CSV、编码为 Shift\_JIS 的全日本的邮政编码文件 KEN\_ALL.CSV。

备注 所谓 CSV 格式是指，"aaa","bb","cccc" 这样的值之间用逗号 (,) 做间隔的数据格式。

关于 CSV 中各个字段的含义，可在邮政编码的说明页面查询。前 9 个字段的含义如下所示。

① 日本地方公共团体编码（JIS X0401、X0402）：半角数字

②（旧）邮政编码（5 位）：半角数字

③ 邮政编码（7 位）：半角数字

④ 都道府县名：半角片假名（按编码顺序排序）

⑤ 市区町村名：半角片假名（按编码顺序排序）

⑥ 町域名：半角片假名（按五十音顺序排序）

⑦ 都道府县名：汉字（按编码顺序排序）

⑧ 市区町村名：汉字（按编码顺序排序）

⑨ 町域名：汉字（按五十音顺序排序）

另外，第 13 个字段表示的是“1 个邮政编码表示多个町域”的情况，当这个值为 1 时，表示同一个邮政编码会出现在多条数据中。

下面是实际文件中的 1 条数据，可以看出，各项目之间用逗号 (,) 做间隔，除了开头和末尾的项目外，其他都用""括了起来。

```
01101,"060 ","0600042","ホカイトウ","サッポロシチュオウク","オホーツク(1-19 チョウメ)","北海道","札幌市中央区","大通西（1～19 丁目）",1,0,1,0,0,0
```

## 23.2 检索邮政编码

首先我们先写一个简单的检索程序，显示邮政编码所对应的数据（代码清单 23.1）。为了获知处理耗时，我们会获取程序的开始和结束时间，并输出两者的差。在笔者的计算机上，整个处理过程大概用了 2 秒。

代码清单 23.1 split\_zip.rb

```
code = ARGV[0]
start_time = Time.now      # 获取处理开始的时间
File.open("KEN_ALL.CSV", "r:shift_jis").each_line do |line|
  line.chomp!
  rows = line.split(/,/)
  zipcode = rows[2].gsub(/\//, '')
  if zipcode == code
    puts line.encode('utf-8')
  end
end
p Time.now - start_time # 输出处理结束时间与开始时间的差
```

通过代码 23.1 可以看出，按照从文件开头逐行读取、检索一致的邮政编码这种形式，遍历所有的行需要花几秒时间。这种做法很实用，接下来我们再来看看有没有更快的处理办法。

## 23.3 sqlite3 库

为了加快数据处理的速度，我们可以使用数据库。这次我们使用开源的关系型数据库 SQLite 以及操作数据库用的语言 SQL，来对数据进行检索、更新等操作。由于 SQLite 的第 3 版是最新版本，因此有时我们会直接称之为 SQLite3。

- SQLite 官方网站：<http://www.sqlite.org/>

用 Ruby 操作 SQLite3，需要使用 `SQLite3` 库。这里我们利用 `gem` 进行安装。关于 `gem` 的内容，请参考 B.1 节。

### 执行示例

```
> gem install sqlite3
```

注 `sqlite3` 的 `gem`，在 Windows 中是经过编译并已发布的二进制包，而在 Linux、Mac OS X 中，除了安装对应发行版的包以外，还需要执行 `gem` 命令。有关安装 `sqlite3` 的 `gem` 方面内容，我们已经在本书的官方网站（<http://www.notwork.org/tanoshiiruby4/>）中做了补充，读者可以参考。

数据库中的数据是以“表”为单位管理的。1 张数据库表就像 1 个 CSV 文件，表中有多行数据，每行数据中有多个项目。CSV 文件的格式恰好与数据库表存放数据时的结构是一致的。在数据库中创建这样的数据库表，我们就可以将数据保存到表中，使用 SQL 对数据进行插入、更新、删除等操作。

让我们先看看用 SQLite3 处理数据的例子。在保存数据前，首先需要准备保存数据用的表。这里，我们对名为 `mydb.db` 的数据库文件创建 `ADDRBOOK` 表，用于保存名字与住址。以下是创建表的程序。

```
SQLite3::Database.open "mydb.db" do |db|
  db.execute "CREATE TABLE ADDRBOOK (name TEXT, address TEXT)"
end
```

本书只用了 SQLite3 中的个别功能，实际只使用了两个方法：一个是 `SQLite3::Database` 类的类方法 `open` 方法，另外一个也是 `SQLite3::Database` 类的实例方法 `execute` 方法。

`SQLite3::Database.open` 方法的第一个参数为数据库的文件名。第 2 行的 `Database#execute` 方法，用于执行在 `mydb.db` 中创建新表 `ADDRBOOK` 的 `CREATE TABLE` 语句。接着在表中定义了 `name`、`email`、`address` 和 `tel` 四个字段。<sup>1</sup> 为了可以保存任何长度的字符串，各字段的类型都定义为没有长度限制的 `TEXT` 类型。创建表后，像下面那样对表插入数据。

<sup>1</sup> 在上面的例子中实际只定义了 `name`、`address` 两个字段。——译者注

```
SQLite3::Database.open "mydb.db" do |db|
  db.execute "INSERT INTO ADDRBOOK VALUES (?, ?, [col1, col2])"
end
```

第 1 行的 `SQLite3::Database.open` 方法和之前的一样。第 2 行的 `Database#execute` 方法执行的是 `INSERT` 语句，这是向数据库插入数据的 SQL。`(?, ?)` 表示表中的字段，分别用 `col1` 及 `col2` 对这两个字段进行赋值。

最后，用以下方法读取数据。

```
SQLite3::Database.open "mydb.db" do |db|
  db.execute("SELECT * FROM ADDRBOOK") do |rows|
    p rows
  end
end
```

同样是执行 `execute` 方法，不过参数的字符串要变为 SQL 的 `SELECT` 语句。另外，`execute` 方法可以使用块，块变量就是数组形式的 SQL 的执行结果。

关于 SQLite 的更详细的资料，请参考 SQLite 手册（<http://www.sqlite.org>）。

## 23.4 插入数据

接下来介绍的功能，都通过封装为 `JZipCode` 类的方法来实现。

首先需要设计邮政编码表的构成。这里，我们简单地设计为下面那样的表。

表 23.1 邮政编码检索表

	邮政编码	都道府县名	市区町村名	町域名	用于检索的住址
字段名	code	pref	city	address	alladdress
数据类型	TEXT	TEXT	TEXT	TEXT	TEXT

为了简化程序，数据类型都定义为 `TEXT` 类型，可保存任意长度的字符串数据。

开头 4 个字段的数据原封不动来自 CSV 文件。最后的“用于检索的住址”是连接都道府县名、市区町村名、町域名后的字符串。用住址进行检索时，可用“东京都港区”2。

我们使用 SQL 的 `CREATE TABLE` 语句创建表。表 23.1 的 SQL 如下所示。

```
CREATE TABLE IF NOT EXISTS zips (code TEXT, pref TEXT, city TEXT, addr TEXT, alladdr TEXT);
```

`IF NOT EXISTS` 表示没有同名表时才创建表。执行上面的 SQL 后就创建了有 5 个 `TEXT` 类型字段的 `zips` 表。

代码清单 23.2 中的 `JZipCode` 类实现了把邮政编码数据插入到 `zips` 表的功能 (`JZipcode#make_db` 方法)。

#### 代码清单 23.2 jzipcode.rb (插入处理)

```
require 'sqlite3'

class JZipCode
  COL_ZIP = 2
  COL_PREF = 6
  COL_CITY = 7
  COL_ADDR = 8

  def initialize(dbfile)
    @dbfile = dbfile
  end

  def make_db(zipfile)
    return if File.exists?(@dbfile)
    SQLite3::Database.open(@dbfile) do |db|
      db.execute <<-SQL
        CREATE TABLE IF NOT EXISTS zips
        (code TEXT, pref TEXT, city TEXT, addr TEXT, alladdr TEXT)
      SQL

      File.open(zipfile, 'r:shift_jis') do |zip|
        db.execute "BEGIN TRANSACTION"
        zip.each_line do |line|
          columns = line.split(/,/).map{|col| col.delete('')}
          code = columns[COL_ZIP]
          pref = columns[COL_PREF]
          city = columns[COL_CITY]
          addr = columns[COL_ADDR]
          all_addr = pref+city+addr
          db.execute "INSERT INTO zips VALUES (?, ?, ?, ?, ?)",
            [code, pref, city, addr, all_addr]
        end
        db.execute "COMMIT TRANSACTION"
      end
    end
  end
end
```

`COL_ZIP`、`COL_PREF` 等是表示数据是在 CSV 文件的第几个字段的常量。

`JZipCode.new` 方法的参数为数据库文件名。`initialize` 方法只是单纯将文件名保存到实例变量。`make_db` 方法、`find_by_code` 方法等在打开数据库时将会用到这个变量。

如果数据库文件已经存在，程序的初始化已经完成，`make_db` 方法将不会进行任何操作。

文件不存在时，`SQLite3::Database#open` 方法将会打开新增的数据库文件，执行 SQL 语句 `CREATE TABLE`。然后打开编码为 Shift\_JIS 的 CSV 文件，使用 `split` 方法以 , 分割。对分割后的各个元素的字符串使用 `delete` 方法删除”，然后再通过 `map` 方法的块，将结果保存到变量 `columns` 中。最后执行 `INSERT` 语句，把变量中的都道府县名、市区町村名、町域名等数据插入到数据库。

执行 `INSERT` 语句的前后分别有 `BEGIN TRANSACTION` 语句，以及 `COMMIT TRANSACTION` 语句，这是为加快 `INSERT` 语句执行速度常用的方法，在 `SQLite3` 中也经常使用。

## 23.5 检索数据

接下来，我们来检索已经保存好的邮政编码。代码清单 23.2 的 `JZipCode` 类定义了 `find_by_code` 方法以及 `find_by_address` 方法。

```
class JZipCode
  |
  def find_by_code(code)
    sql = "SELECT * FROM zips WHERE code = ?"
    str = ""
    SQLite3::Database.open(@dbfile) do |db|
```

```

db.execute(sql, code) do |row|
  str << sprintf("%s %s", row[0], row[4]) << "\n"
end
end
str
end

def find_by_address(addr)
  sql = "SELECT * FROM zips WHERE alladdr LIKE ?"
  str = ""
  SQLite3::Database.open(@dbfile) do |db|
    db.execute(sql, "%#{addr}%") do |row|
      str << sprintf("%s %s", row[0], row[4]) << "\n"
    end
  end
  str
end
end

```

`find_by_code` 方法以邮政编码为参数，返回该邮政编码对应的住址。`find_by_address` 方法恰好相反，以住址为参数，返回包含该住址的邮政编码。

执行检索时，与 `INSERT` 语句一样，使用 `Database#execute` 方法，执行 `SELECT` 语句。通过参数传递的值，会置换掉

`WHERE code = ?`、`WHERE alladdr LIKE` 中的条件部分 `?` 的值。检索结果以数组的形式赋值给变量 `row`，然后再将连接后的字符串保存在变量 `str`。

下面，我们通过 `irb` 命令，测试一下 `find_by_code` 方法以及 `find_by_address` 方法。

#### 执行示例

```

> irb --simple-prompt
>> require "./jzipcode"
=> true
>> jzipcode = JZipCode.new("zip.db")
=> #<JZipCode:0x2c2ab08 @dbfile="zip.db">
>> jzipcode.make_db("KEN_ALL.CSV")
=> #<SQLite3::Database:0x2c2eb40>
>> puts jzipcode.find_by_code('1060031')
1060031 東京都港区西麻布
=> nil
>> puts jzipcode.find_by_address("東京都渋谷区神")
1500047 東京都渋谷区神山町
1500001 東京都渋谷区神宮前
1500045 東京都渋谷区神泉町
1500041 東京都渋谷区神南
=> nil

```

首先，通过 `require "./jzipcode"` 引用 `jzipcode.rb` 的内容。接着，用 `JZipCode.new` 方法创建实例，通过 `make_db` 方法读取数据。之后，对 `find_by_code` 方法指定邮政编码，则会输出对应该邮政编码的住址。同样地，对 `find_by_address` 方法指定住址的某部分，则会输出包含该住址的邮政编码。

## 23.6 总结

本章我们介绍了如何使用 `SQLite3` 库提高检索大量数据时的速度。处理大量数据时，根据实际需要，使用数据库以及 `RubyGems` 等现成的类库，会大大提高程序编写效率。

数据库产品中，除了 `SQLite3` 外，`MySQL`、`PostgreSQL` 等也都被广泛使用。虽然不同的数据库产品的 SQL 语法等在细节上存在差异，但基本结构是大致相同的。有兴趣的读者可以继续学习其他数据库产品。

# 附录

---

Ruby 运行环境的构建

Ruby 的安装

在 Windows 下的安装

在 Mac OS X 下的安装

在 Unix 下的安装

编辑器与 IDE

Ruby 参考集

RubyGems

Ruby 参考手册

命令行参数

预定义变量、常量

错误信息

# 附录 A Ruby 运行环境的构建

## A.1 Ruby 的安装

接下来将介绍在 Windows、MAC OS X、Unix 下使用 Ruby 的方法。在 Windows 下利用 mswin32 版的安装包——RubyInstaller for Windows 安装 Ruby。在 Mac OS X 中默认已经安装了 Ruby，虽然可以直接使用，不过由于版本过于老旧，我们将介绍如何使用软件包管理工具升级 Ruby。在 Unix 下，我们将会介绍用从源代码编译的方法安装 Ruby。

关于 Ruby 的安装方法，也可以参考以下网页。

- 下载 Ruby (Ruby 官方网站)：[https://www.ruby-lang.org/zh\\_cn/downloads/](https://www.ruby-lang.org/zh_cn/downloads/)

请自行安装软件。

## A.2 在 Windows 下安装

本节介绍如何使用 RubyInstaller 安装 Ruby。首先可从以下网站下载 RubyInstaller。

- Ruby Installer for Windows：<http://rubyinstaller.org/>

点击首页的 Download 链接则会跳转到下载页面，从上面可以看到安装包的一览表，点击对应的下载链接就可以下载 Ruby 2.0.0-p451 安装包 1。

1翻译本书时最新版本为 Ruby 2.0.0-p451。——译者注

下面我们来介绍通过 RubyInstaller 安装 Ruby 的步骤。下面的截图是在 Windows 8 Pro 64 位版下安装 Ruby 2.0.0-p451 时截取的。

### A.2.1 开始安装

双击下载后的 rubyinstaller-2.0.0-p451-x64.exe 图标，启动 Installer。

在语言选择对话框中直接点击 OK 按钮（图 A.1）。

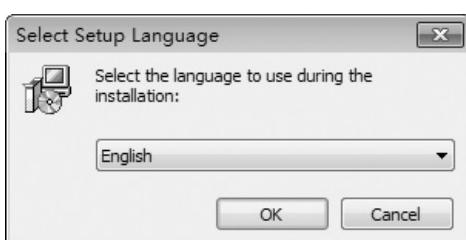


图 A.1 语言选择对话框

### A.2.2 同意软件使用许可协议

接下来显示的是 RubyInstaller 的软件使用许可协议。RubyInstaller 本身是遵循 BSD 许可证的，Ruby 以及第三方的软件的许可证需要另外再确认。如果不是用于商业用途，一般不会有什么问题。确认后，选择 I accept the License，点击 Next 按钮。

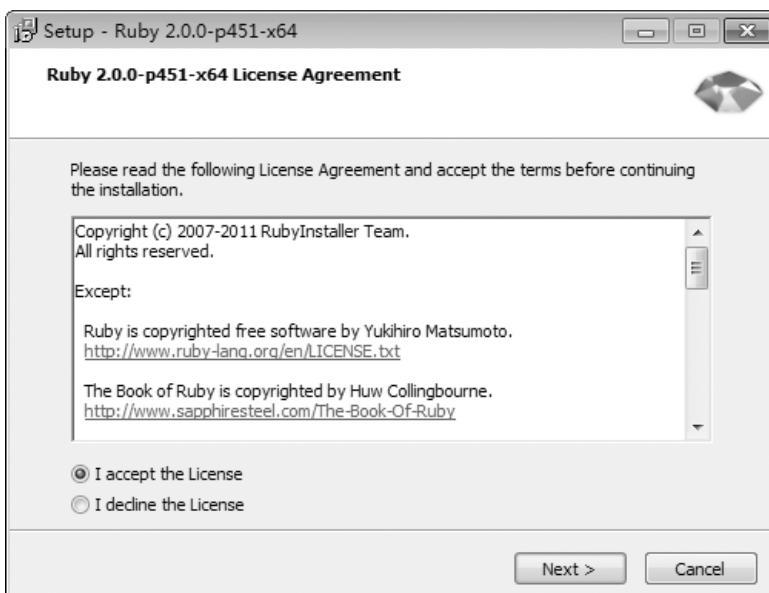


图 A.2 软件使用许可协议

## A.2.3 确认安装路径以及选项

安装时我们可以指定安装路径以及一些安装选项（图 A.3），选项有以下 3 个。

- 安装 Tcl/Tk 支持

Tcl/Tk 是用于创建视窗程序的 GUI 类库。虽然本书并没有涉及该内容，不过即使安装也不会有什么影响。

- 把 Ruby 执行文件设置到环境变量 PATH

通过设置环境变量 PATH，在普通的命令行可以直接执行 Ruby.exe。与其他程序的 DLL 读取也会有关联，在不清楚影响范围时，请不要选择。

- 把 .rb 与 .rbw 文件与 Ruby 关联

双击扩展名为 .rb 与 .rbw 的文件时，会把文件作为 Ruby 脚本来执行。

选择需要的选项后，点击 Install 按钮。

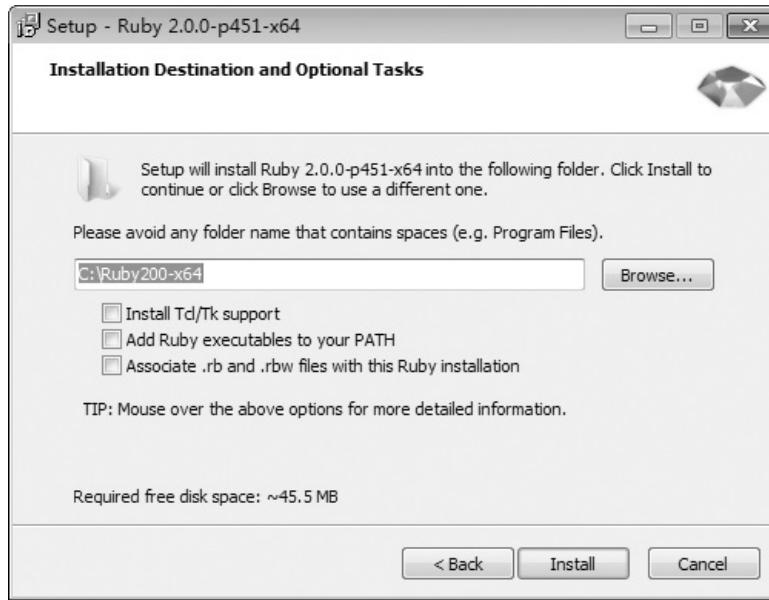


图 A.3 确认安装路径以及选项

## A.2.4 安装进度

显示安装进度（图 A.4）。

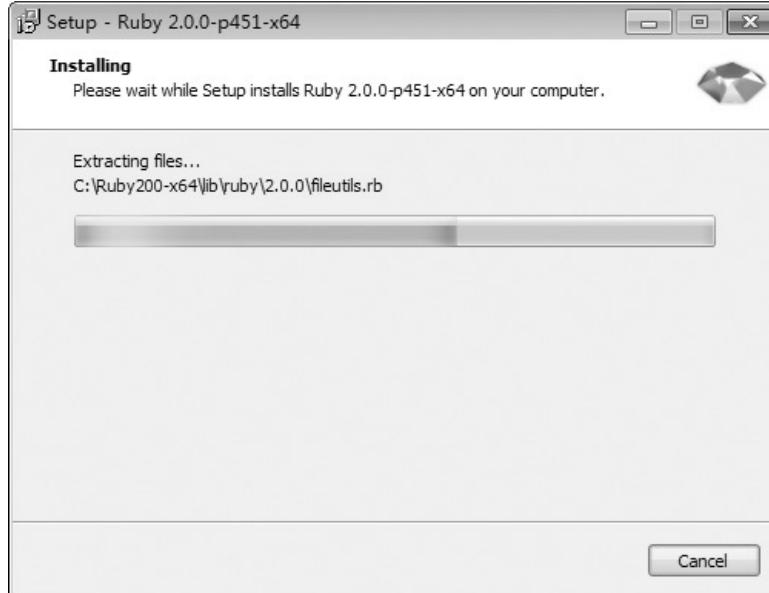


图 A.4 安装进度

## A.2.5 安装完成

安装完成后，点击 Finish 按钮结束 RubyInstaller 的安装（图 A.5）。

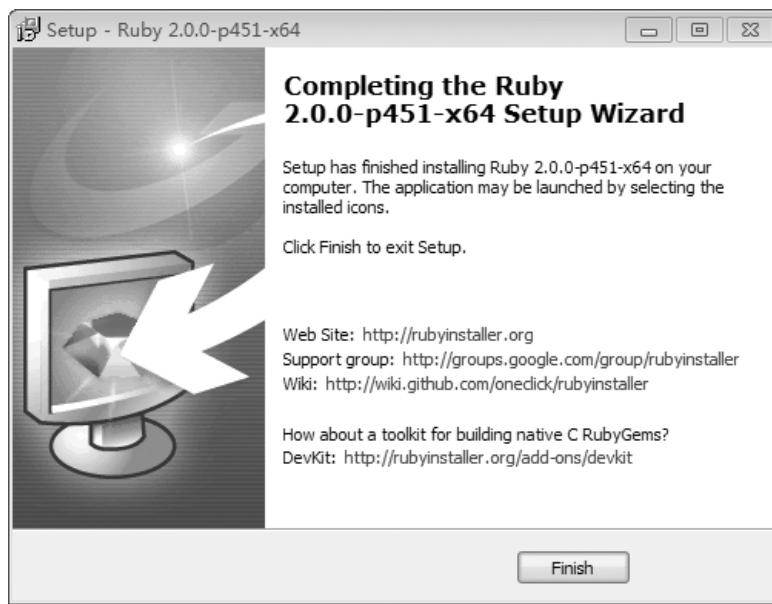


图 A.5 安装完成

#### A.2.6 启动控制台

系统为 Windows 8 时，点击开始界面上的 Start Command Prompt with Ruby，系统会自动加载执行 Ruby 需要的环境变量，然后启动控制台（图 A.6）。



图 A.6 启动 Ruby 控制台

系统为 Windows 7 时，按照以下顺序启动控制台：开始→所有程序→Ruby-2.0.0-p451-x64→Start Command Prompt with Ruby。



图 A.7 启动 Ruby 控制台（Windows 7）

启动控制台（图 4.8）后，输入 ruby -v，可以确认 Ruby 的版本。显示下图信息时表示已经安装完毕。

```
cmd Start Command Prompt with Ruby
ruby 2.0.0p451 <2014-02-24> [x64-mingw32]
C:\Users\kamiyu>ruby -v
ruby 2.0.0p451 <2014-02-24> [x64-mingw32]

C:\Users\kamiyu>
```

A screenshot of a Windows command prompt window. The title bar says "cmd Start Command Prompt with Ruby". The window contains the following text:  
ruby 2.0.0p451 <2014-02-24> [x64-mingw32]  
C:\Users\kamiyu>ruby -v  
ruby 2.0.0p451 <2014-02-24> [x64-mingw32]  
C:\Users\kamiyu>  
The window has standard Windows-style scroll bars on the right and bottom.

图 A.8 Ruby 的控制台

### A.3 在 Mac OS X 下安装

Mac OS X 虽然会默认安装 Ruby，但因为是旧版本的，所以我们需要自行安装最新版本。

为了确认当前 Ruby 的版本，首先要启动控制台。通过在 Finder 中选择应用程序→工具→终端来启动控制台。

输入 Ruby -v 就可以显示当前 Ruby 的版本。

#### 执行示例

```
> ruby -v
ruby 1.8.7 (2012-02-08 patchlevel 358) [universal.darwin12.0]
```

像这样，显示 1.8 则表示系统安装的是旧版本的 Ruby。

自行安装最新版 Ruby 时，我们可以选择使用软件包管理工具安装，或者从源代码编译安装。

从源代码安装的步骤与在 Unix 下的安装是一样的，请参考 A.4 节。

下面介绍通过安装包管理系统 MacPorts 安装 Ruby 的方法。

### 通过 MacPorts 安装

MacPorts 是 Mac OS X 平台下使用的软件包管理工具。首先要在 MacPorts 的官方网站 (<http://www.macports.org>) 下载最新的安装包。请注意，不同的 Mac OS X 的版本对应的下载 .dmg 文件是不一样的。双击下载好的安装文件后，按照画面指示即可安装 MacPorts。

安装完成后，首先执行以下命令，把 MacPorts 以及软件包列表更新到最新版本。使用 port 命令需要用到超级管理员权限，因此需要配合 sudo 命令一起使用。

```
> sudo port -v selfupdate  
> sudo port -d sync
```

系统询问密码时，输入当前用户的密码即可。

接下来就可以安装 Ruby 了。

```
> sudo port install ruby20
```

这样就可以安装 Ruby2.0.0 了。

但安装后，需要输入 ruby20 才可以执行 ruby 命令。使用以下命令，输入 ruby 时，实际执行的是 ruby20。

```
> sudo port select ruby ruby20  
Selecting 'ruby20' for 'ruby' succeeded. 'ruby20' is now active.
```

MacPorts 会将 Ruby 安装在 /opt/local/bin 目录下。不希望指定文件路径，直接使用通过 MacPorts 安装的文件时，可将以下内容追加到环境变量设定文件 ~/.bashrc 中。

```
# PATH 追加  
export PATH=/opt/local/bin:/opt/local/sbin:$PATH  
# MANPATH 追加  
export MANPATH=/opt/local/man:$MANPATH
```

可用下面的命令，确认通过 MacPorts 安装的 Ruby 版本。

```
> port select --show ruby  
The currently selected version for 'ruby' is 'ruby20'
```

## A.4 在 Unix 下安装

Unix 下有可能已经默认安装了 Ruby，可在控制台执行以下命令确认。

```
> ruby -v
```

像下面那样显示 Ruby 版本为 2.0.0 时可不重新安装。

```
ruby 2.0.0p0 (2013-02-24 revision 39474) [i386-netbsdelf]
```

如果安装的是 Ruby 1.9 等旧版本 Ruby，读者可考虑安装最新版本。

### A.4.1 从源代码编译

首先通过以下 URL 下载 Ruby 的源代码。

- <ftp://ftp.ruby-lang.org/pub/ruby/ruby-版本号-p补丁级别.tar.gz>

版本号部分表示的是 Ruby 的版本。本书翻译时最新的版本为 ruby-2.0.0-p451，因此最新的 URL 如下所示。

- <ftp://ftp.ruby-lang.org/pub/ruby/ruby-2.0.0-p451.tar.gz>

执行以下命令，解压缩包后，会自动创建 ruby-2.0.0-p451 目录。

```
> tar zxvf ruby-2.0.0-p451.tar.gz
```

进入该目录，按顺序执行以下命令，则安装完毕。

```
> cd ruby-2.0.0  
> ./configure  
> make  
> make test  
> make install
```

最后的 `make install` 命令需要配合 `sudo` 命令，使用超级管理员权限执行。没有超级管理员权限时，要么请拥有权限的管理人员安装，要么安装在当前用户可读写的目录下。例如，执行以下命令可将 Ruby 安装在当前用户的主目录下。

```
> ./configure --prefix=$HOME
```

这种情况下，`ruby` 命令、`irb` 命令等将会安装在 `$HOME/bin` 目录下。

#### A.4.2 使用二进制软件包

在 Unix 下编译安装 Ruby 虽然简单，但使用各平台的软件包管理工具安装则会便于日后管理。现在广泛使用的平台（各发行版的 Linux、BSD 等）中，都有各自对应的二进制或者从源码编译好的软件包，读者可根据各平台的使用方法安装 Ruby。

#### A.4.3 使用 Ruby 软件包管理工具

Ruby 并非某个平台独占的语言，还有一些跨平台的 Ruby 专用的软件包管理工具。其中 `rvm` 以及 `rbenv` 比较有名。在这里我们来介绍一下 `rbenv` 的使用方法。

`rbenv` 的源码托管在 github 网站，通过 `git` 命令获取。下面，我们把 `rbenv` 的源码下载到当前用户的主目录下的 `.rbenv` 目录中。

```
> git clone git://github.com/sstephenson/rbenv.git ~/.rbenv
```

环境变量 `PATH` 以及 `rbenv` 的初始化信息可在 shell 的设定文件中设定。使用 `bash` 时，执行以下命令，将必要的初始化信息设定到环境设定文件 `~/.bashrc` 中。

```
> echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc  
> echo 'eval "$(rbenv init -)"' >> ~/.bashrc
```

然后使用下面的命令更新 shell 的状态，也可以直接重启 shell。

```
> exec $SHELL -l
```

使用 `git` 命令下载 `ruby-build`。

```
> git clone git://github.com/sstephenson/ruby-build.git ~/.rbenv/plugins/ruby-build
```

这样就可以使用 `rbenv install` 命令了。如果执行 `rbenv rehash` 命令，使用 `rbenv global` 命令，则可将 `ruby` 命令设定为 2.0.0 版本。

```
> rbenv install 2.0.0-p451  
> rbenv rehash  
> rbenv global 2.0.0-p451  
> ruby -v  
ruby 2.0.0p451 (2014-02-24 revision 45167) [x86_64-darwin13.1.0]
```

## A.5 编辑器与 IDE

一般我们用文本编辑器编写 Ruby 程序。可对应 Ruby 语法的编辑器，提供了诸如在适当的地方进行代码缩进，对 `if`、`while` 等关键字、常量、字符串等加上颜色等提高编程效率的功能。

读者可参考使用本节介绍的编辑器、IDE 工具，当然也可以选择使用符合自己习惯的工具。

在 Unix、Mac OS X 系统上经常使用的编辑器有 Vim 和 Emacs。可免费使用的 Unix 中，一般都可以通过 OS 提供软件包管理工具安装。在使用英语的 Mac 用户中，Sublime Text 2 非常流行，而在日本则不太普及。

Windows 平台下的默认安装的记事本（notepad），其编辑功能非常弱，几乎不能用于编写程序。因此需要自行安装可对应 Ruby 编程的编辑器。

除了普通的编辑器外，还可以使用所谓的 IDE（集成开发环境）提供的编辑器编写 Ruby 程序。IDE 除了可以编写程序，还集成了执行、测试等功能。

以 Java IDE 为基础的 Aptana Studio（原名 RadRails）、RubyMine、NetBeans 等都是比较常用的 IDE。

- **RadRails** : <http://www.radrails.org/>
- **RubyMine** : <http://www.jetbrains.com/ruby/>
- **NetBeans** : <https://netbeans.org/>

看到这里，也许有的读者会认为若没有好用的编辑器就不能用 Ruby 编写程序。一般 Unix 系统默认都安装了 Vi 或者 Emacs，因此对使用 Unix 的读者来说问题不大，但使用 Windows 的读者却大部分只安装了记事本。

但实际情况并非如此。笔者在创建简单的程序，或者只是稍微简单修改一下程序时也经常使用记事本或通过 irb 等工具直接执行 Ruby 程序。这种情况下，虽然需要在代码缩进等方面下点功夫，但也没想象中那么花时间。

但对于初学者来说，为了便于快速入门，建议还是使用上手门槛较低的编辑器，因此我们在这里介绍了 Ruby 的各种常用的编辑器。熟练掌握编辑器的使用方法，是写出优质程序的捷径。

## 附录 B Ruby 参考集

### B.1 RubyGems

RubyGems 是一个统一安装、管理 Ruby 类库、程序的 Ruby 标准工具。在 RubyGems 中，每个单独的库称为 gem。通过 RubyGems，我们可以搜索 gem，显示 gem 相关的信息，安装 / 卸载 gem，升级旧版本的 gem，以及查看 gem 的安装进度一览表，等等。

#### gem 命令

我们一般在命令行使用 RubyGems，命令为 gem。

- **gem list**

显示 gem 的安装进度一览表。

#### 执行示例

```
gem list
```

```
> gem list
*** LOCAL GEMS ***
abstract (1.0.0)
actionmailer (4.0.0)
actionpack (4.0.0)
activemodel (4.0.0)
|
```

像本例中的 list 那样的指令称为 gem 命令。除了 list 外还有其他 gem 命令，以下列举的是其中常用的命令。

- **gem search**

用于搜索 gem，没有指定选项时，会搜索已安装的 gem 文件。

#### 执行示例

```
> gem search nokogiri
*** LOCAL GEMS ***
nokogiri (1.5.0, 1.4.1, 1.4.0)
```

指定 -r 选项后，则搜索的目标为远程仓库（remote repository）。

#### 执行示例

```
> gem search -r nokogiri
*** REMOTE GEMS ***
aaronp-nokogiri (0.0.0.20080825000844)
backupify-rsolr-nokogiri (0.12.1.1)
epp-nokogiri (1.0.0)
glebm-nokogiri (1.4.1)
nokogiri (1.5.9 ruby java x86-mingw32 x86-mswin32-60, 1.4.4.1 x86-mswin32)
|
```

- **gem install**

安装 gem，安装所需的文件会自动从互联网下载。

#### 执行示例

```
> gem install nokogiri
```

安装本地的 gem 时，不是指定 gem 名，而是指定 gem 文件名。

#### 执行示例

```
> gem install nokogiri-1.5.9.gem
```

- **gem update**

把 gem 更新为最新版本。

#### 执行示例

```
> gem update nokogiri
```

RubyGems 自身的更新也是使用这个命令，这时需要加上 `--system` 选项。

### 执行示例

```
> gem update --system
```

除此之外还有许多 `gem` 命令，表 B.1 为 `gem` 命令的一览表。

表 B.1 `gem` 命令

选项	意义
build	根据 <code>gemspec</code> 创建 <code>gem</code>
cert	管理、签署 RubyGems 的许可证时使用
check	检查 <code>gem</code>
cleanup	整理已安装的旧版本的 <code>gem</code>
contents	显示已安装 <code>gem</code> 的内容
dependency	显示已安装 <code>gem</code> 的依赖关系
environment	显示 RubyGems、Ruby 等相关的环境信息
fetch	把 <code>gem</code> 文件下载到本地目录，但不安装
generate_index	创建 <code>gem</code> 服务器所需的索引文件
help	显示 <code>gem</code> 命令的帮助说明
install	安装 <code>gem</code>
list	显示 <code>gem</code> 的一览表
lock	锁定 <code>gem</code> 版本，并输出锁定后的 <code>gem</code> 列表
mirror	创建 <code>gem</code> 仓库的镜像
outdated	显示所有需要更新的 <code>gem</code> 列表
pristine	从 <code>gem</code> 缓存中获取已安装的 <code>gem</code> ，并将其恢复为初始状态
query	搜索本地或者远程仓库的 <code>gem</code> 信息
rdoc	生成已安装的 <code>gem</code> 的 RDoc 文件
search	显示名字包含指定字符串的 <code>gem</code>
server	启动 HTTP 服务器，用于管理 <code>gem</code> 的文档及仓库
sources	管理搜索 <code>gem</code> 时所需的 RubyGems 的源以及缓存
specification	以 yaml 形式显示 <code>gem</code> 的详细信息
stale	按最后访问的时间顺序显示 <code>gem</code> 的一览表
uninstall	从本地卸载 <code>gem</code>
unpack	在本地目录解压已安装的 <code>gem</code>
update	更新指定的 <code>gem</code> (或者全部 <code>gem</code> )
which	显示读取 <code>gem</code> 时引用的类库

## B.2 Ruby 参考手册

### B.2.1 Web 上的资源

Ruby 参考手册是以源代码为基础用英语创建的。使用 `ri` 命令可以也方便地阅读手册。以下是与 Ruby 相关的在线文档资源。

- 文档（Ruby 官方网站）

[https://www.ruby-lang.org/zh\\_cn/documentation/](https://www.ruby-lang.org/zh_cn/documentation/) (简体中文)

<https://www.ruby-lang.org/en/documentation/> (英语)

## B.2.2 ri 命令

ri 命令用于在控制台阅读 Ruby 参考手册。内容是英语，不过除了功能说明外还有简单的使用示例，即使不阅读全文也可以从示例中得到启发。有些 gem 安装方式并不会在安装 gem 的同时安装相应的参考手册，因此在能连接网络的环境下，建议先参考 Web 上的资源。

下面是 ri 命令的参数，ri 命令会显示与参数指定的类、方法相关的参考手册。

**ri [选项] [类名或者方法名]**

执行 ri --help 可以显示 ri 命令的命令行参数的一览表，主要显示的是与显示形式、帮助的搜索路径等相关的选项，在这里就不详细说明了。

在控制台下执行下面的“ri 类名”或者“ri 方法名”即可阅读对应的参考手册。

### 执行示例

```
> ri Regexp
= Regexp < Object

(from ruby site)
-----
A Regexp holds a regular expression, used to match a pattern against strings.ya
Regexp objects are created using the /.../ and %r{...} literals, and by the
Regexp::new constructor.

| 

> ri Regexp.match
= Regexp.match

(from ruby site)
-----
rxp.match(str)      -> matchdata or nil
rxp.match(str,pos)  -> matchdata or nil

-----
>Returns a MatchData object describing the match, or nil if there was no match.
This is equivalent to retrieving the value of the special variable $~
following a normal match. If the second parameter is present, it specifies the
position in the string to begin the search.
```

## B.2.3 阅读参考手册的技巧

下面列举两个阅读参考手册的技巧。

- 无需一次全部记住

Ruby 原生的方法已经够多，加上其他类库的方法，我们根本不可能一次掌握全部方法。因此在熟悉 Ruby 前，抱着“在必要时阅读必要的部分”这样的态度就可以了。熟练掌握 Ruby 后，我们推荐大家试一下“重头到尾阅读一下某个类的方法”“重头到尾阅读语法说明”等阅读方法。

- 查看父类的方法

在查看某个类的方法时，有时在类的参考手册中找不到该方法的描述。这种情况下，很可能需要查看的方法并不在这个类中，而是在其父类中定义的。也就是说，该类使用的方法是继承自父类的。

## B.3 命令行选项

在执行 Ruby 时，我们可以指定命令行选项。例如指定 -v 选项执行 ruby 命令，则会显示版本号。

### 执行示例

```
> ruby -v
ruby 2.0.0p451 (2014-02-24 revision 45167) [x86_64-linux]
```

表 B.2 是 ruby 命令的命令行选项一览表。单行程序（one-liner program）这样方便的工具也都实现了，读者可以先粗略地看看。在 Linux、Unix 系统下，还可以使用 man 命令读取更详细的选项说明。

表 B.2 Ruby 的命令行选项

选项	意义
<code>-Octal</code>	用 8 进制指定 <code>IO.gets</code> 等识别的换行符
<code>-a</code>	指定为自动分割模式 (与 <code>-n</code> 或者 <code>-p</code> 选项一起使用时则将 <code>\$F</code> 设为 <code>\$_.split(\$;)</code> )
<code>-c</code>	只检查脚本的语法
<code>-Cdirectory</code>	在脚本执行前, 先移动到 <code>directory</code> 目录下
<code>-d、--debug</code>	使用 <code>debug</code> 模式 (将 <code>\$DEBUG</code> 设为 <code>true</code> )
<code>-e 'command'</code>	通过 <code>command</code> 指定一行代码的程序。本选项可指定多个
<code>-Eex[:in]、--encoding=ex[:in]</code>	指定默认的外部编码 ( <code>ex</code> ) 以及默认内部编码 ( <code>in</code> )
<code>-Fpattern</code>	指定 <code>String#split</code> 方法使用的默认分隔符 ( <code>\$;</code> )
<code>-i[extension]</code>	以替换形式编辑 <code>ARGV</code> 文件 (指定 <code>extension</code> 时则会生成备份文件)
<code>-ldirectory</code>	指定追加到 <code>\$LOAD_PATH</code> 的目录。本选项可指定多个
<code>-l</code>	删除 <code>-n</code> 或者 <code>-p</code> 选项中的 <code>\$_</code> 的换行符
<code>-n</code>	是脚本整体被 <code>'while gets(); ... end'</code> 包围 (将 <code>gets()</code> 的结果设定到 <code>\$_</code> 中)
<code>-p</code>	在 <code>-n</code> 选项的基础上, 在每个循环结束时输出 <code>\$_</code>
<code>-rlibrary</code>	在执行脚本前通过 <code>require</code> 引用 <code>library</code>
<code>-s</code>	要使脚本解析标志 (flag) 的功能有效 (' <code>ruby -s script -abc</code> ', 则 <code>\$abc</code> 为 <code>true</code> )
<code>-S</code>	从环境变量 <code>PATH</code> 开始搜索可执行的脚本
<code>-Tlevel</code>	指定不纯度检查模式
<code>-U</code>	将内部编码的默认值 ( <code>Encoding.default_internal</code> ) 设为 <code>UTF-8</code>
<code>-v、--verbose</code>	显示版本号, 冗长模式设定为有效 ( <code>\$VERBOSE</code> 设定为 <code>true</code> )
<code>-w</code>	冗长模式设定为有效

<code>-Wlevel</code>	指定冗长模式的级别 [0= 不输出警告, 1= 只输出重要警告, 2= 输出全部警告 (默认值) ]
<code>-xdirectory</code>	忽略执行脚本中 <code>#!ruby</code> 之前的内容
<code>--copyright</code>	显示版权信息
<code>--enable-feature[, ...]</code>	使 <i>feature</i> 有效
<code>--disable=feature[, ...]</code>	使 <i>feature</i> 无效
<code>--external-encoding=encoding</code>	指定默认的外部编码
<code>--internal-encoding=encoding</code>	指定默认的内部编码
<code>--version</code>	显示版本信息
<code>--help</code>	显示帮助信息

表 B.3 为 `--enable` 以及 `--disable` 选项可指定的 *feature* (功能名)。

表 B.3 `--enable`、`--disable` 选项可指定的功能名

功能名	意义
<code>gems</code>	RubyGems 是否有效 (默认有效)
<code>rubyoct</code>	是否引用环境变量RUBYOCT (默认引用)
<code>all</code>	上述功能是否全部有效

## B.4 预定义变量、常量

### B.4.1 预定义变量

预定义变量是指 Ruby 预先定义好的变量。它全部都是以 `$` 开头的变量，因此可以像全局变量那样引用。像 `$<` 相对与 `ARGF` 那样，有容易看懂的别名时，建议尽量使用该别名。

表 B.4 为预定义变量一览表。

表 B.4 预定义变量

变量名	内容
<code>\$_</code>	<code>gets</code> 方法最后读取的字符串
<code>\$&amp;</code>	最后一次模式匹配后得到的字符串
<code>\$~</code>	最后一次模式匹配相关的信息

<code>'\$`</code>	最后一次模式匹配中匹配部分之前的字符串
<code>\$'</code>	最后一次模式匹配中匹配部分之后的字符串
<code>\$+ \$1、\$2.....</code>	最后一次模式匹配中最后一个()对应的字符串 最后一次模式匹配中()匹配的字符串(第n个()对应\$ <sub>n</sub> )
<code>\$? \$!</code>	最后执行完毕的子进程的状态 最后发生的异常的相关信息
<code>\$@</code>	最后发生的异常的相关位置信息
<code>\$SAFE</code>	安全级别(默认为0)
<code>\$/</code>	输入数据的分隔符(默认为"\n")
<code>\$\</code>	输出数据的分隔符(默认为nil)
<code>\$,</code>	<code>Array#join</code> 的默认分割字符串(默认为nil)
<code>\$;</code>	<code>String#split</code> 的默认分割字符串(默认为nil)
<code>\$.</code>	最后读取的文件的行号
<code>\$&lt;</code>	<code>ARGF</code> 的别名
<code>\$&gt;</code>	<code>print、puts、p</code> 等的默认输出位置(默认为 <code>STDOUT</code> )
<code>\$0</code>	<code>\$PROGRAM_NAME</code> 的别名
<code>\$*</code>	<code>ARGV</code> 的别名
<code>\$\$</code>	当前执行中的 Ruby 的进程 ID
<code>\$:</code>	<code>\$LOAD_PATH</code> 的别名
<code>\$"</code>	<code>\$LOADED_FEATURES</code> 的别名
<code>\$DEBUG</code>	指定 debug 模式的标识(默认为nil)

<code>\$FILENAME</code>	<code>ARGF</code> 当前在读取的文件名
<code>\$LOAD_PATH</code>	执行 <code>require</code> 读取文件时搜索的目录名数组
<code>\$stdin</code>	标准输入（默认为 <code>STDIN</code> ）
<code>\$stdout</code>	标准输出（默认为 <code>STDOUT</code> ）
<code>\$stderr</code>	标准错误输出（默认为 <code>STDERR</code> ）
<code>\$VERBOSE</code>	指定冗长模式的标识（默认为 <code>nil</code> ）
<code>\$PROGRAM_NAME</code>	当前执行中的 Ruby 脚本的别名
<code>\$LOADED_FEATURES</code>	<code>require</code> 读取的类库名一览表

#### B.4.2 预定义常量

与预定义变量一样，Ruby 也有预先定义好的常量。表 B.5 为预定义常量的一览表。

表 B.5 预定义常量

常量名	内容
<code>ARGV</code>	参数，或者从标准输入得到的虚拟文件对象
<code>ARGV</code>	命令行参数数组
<code>DATA</code>	访问 <code>__END__</code> 以后数据的文件对象
<code>ENV</code>	环境变量
<code>RUBY_COPYRIGHT</code>	版权信息
<code>RUBY_DESCRIPTION</code>	<code>ruby -v</code> 显示的版本信息
<code>RUBY_ENGINE</code>	Ruby 的处理引擎
<code>RUBY_PATCHLEVEL</code>	Ruby 的补丁级别
<code>RUBY_PLATFORM</code>	运行环境的信息 (OS、CPU)
<code>RUBY_RELEASE_DATE</code>	Ruby 的发布日期

RUBY_VERSION	Ruby 的版本
STDERR	标准错误输出
STDIN	标准输入
STDOUT	标准输出

#### B.4.3 伪变量

伪变量虽然可以像变量那样引用，但是不能改变其本身的值，对其赋值会产生错误。

表 B.6 为伪变量一览表。

表 B.6 伪变量

变量名	内容
<code>self</code>	默认的接收者
<code>nil</code> 、 <code>true</code> 、 <code>false</code>	<code>nil</code> 、 <code>true</code> 、 <code>false</code>
<code>__FILE__</code>	执行中的 Ruby 脚本的文件名
<code>__LINE__</code>	执行中的 Ruby 脚本的行编号
<code>__ENCODING__</code>	脚本的编码

#### B.4.4 环境变量

表 B.7 环境变量

变量名	内容
RUBYLIB	追加到预定义变量 <code>\$LOAD_PATH</code> 中的目录名（各目录间用 : 分隔）
RUBYOPT	启动 Ruby 时的默认选项 ( <code>RUBYOPT = "-U -v"</code> 等)
RUBYPATH	<code>-S</code> 选项指定的、解析器启动时脚本的搜索路径
PATH	外部命令的搜索路径
HOME	<code>DIR.chdir</code> 方法的默认移动位置
LOGDIR	HOME 没有时的 <code>DIR.chdir</code> 方法的默认移动位置
LC_ALL、LC_CTYPE、LANG	决定默认编码的本地信息（平台依赖）

## B.5 错误信息

程序不可能一个错误（BUG）都没有。一般程序会因各种各样的错误而终止，而错误信息则为查找这些错误产生的原因而提供线索。

Ruby 的错误信息以英语等方式显示，可能会有读者觉得阅读起来会很麻烦，不过如果不仔细阅读错误信息，往往花很多时间才能解决问题。在这里，我们来介绍一些常见的错误信息及其含义。

错误信息的基本阅读方法请参考第 10 章。

### B.5.1 syntax error

```
foo.rb:2 syntax error, unexpected kEND, expecting ')'
```

程序中有语法错误。特别是括号、字符串忘记关闭时，解析器报告错误的位置很可能会比实际出错的位置靠前。遇到语法错误时，请确认以下几点。

- `if`、`while`、`begin` 等是否存在对应的 `end`
- 括号、字符串是否已关闭
- `Here Document` 是否已关闭
- 数组、哈希的元素间的分隔符是否有错或者漏写
- 运算符的使用方法是否有错

### B.5.2 NameError/NoMethodError

```
name.rb:2:in `foo': undefined local variable or method `retrun' for main:Object (NameError) from name.rb:12:in `<main>'
```

方法或变量不存在。在本例中是由于将 `return` 写成 `retrun`，因此产生了异常。

```
name.rb:2:in `foo': undefined method `inejct' for []:Array (noMethodError) from name.rb:12:in `<main>'
```

方法名有错误时，程序会抛出 `NoMethodError` 异常。这时请确认以下几点。

- 方法名、变量名的拼写是否有错
- 变量是否已赋值给对象
- 是否有将当前的类认作其他类

### B.5.3 ArgumentError

```
arg.rb:1:in `foo': wrong number of arguments (1 for 0) (ArgumentError) from arg.rb:4:in `<main>'
```

方法参数有错误。在本例中，对本不需参数的方法传递了 1 个参数，因此出现了错误。此外，像 `printf` 方法的格式字符串不正确等这样，导致传给方法的参数与期待的不一样时，也会产生这个错误。

### B.5.4 TypeError

```
type.rb:1:in `scan': wrong argument type nil (expected Regexp) (TypeError) from type.rb:1:in `<main>'
```

将方法意料之外的类对象传递给了方法。例如不小心把 `nil` 赋值给变量了，即使是熟悉编程的人也会常犯这样的错误。

### B.5.5 LoadError

```
load.rb:1:in `require': cannot load such file -- foo (LoadError) from load.rb:1:in `<main>'
```

指定给 `require` 的库无法引用。也有可能是使用中的库间接引用其他库时遇到的错误。这时应注意以下几点。

- `require` 的参数是否正确
- 需要读取的库是否已安装
- `$LOAD_PATH` 中的目录中是否有文件

## B.5.6 [BUG]

```
segv.rb:4: [BUG] Segmentation fault
ruby 2.0.0p0 (2013-02-24 revision 39474) [x86_64-linux]

-- Control frame information -----
c:0004 p:---- s:0011 e:000010 CFUNC :segv
c:0003 p:0009 s:0007 e:000006 METHOD segv.rb:4
c:0002 p:0026 s:0004 E:0021f8 EVAL    segv.rb:7 [FINISH]
c:0001 p:0000 s:0002 E:002368 TOP    [FINISH]

...DEBUG 信息...
```

Ruby 本身或者其他扩展类库引起的错误。

这些错误，在 Ruby 最新的版本中有可能已经解决了，因此建议考虑升级。如果升级以后也没解决问题，还可以通过 Ruby 的邮件列表 ruby-list ([https://www.ruby-lang.org/zh\\_cn/community/mailing-lists/](https://www.ruby-lang.org/zh_cn/community/mailing-lists/)) 反应情况。将相关问题反馈给 Ruby 开发团队，对 Ruby 往后的开发会有很大的帮助。

## 后记

---

虽然 TrueNet 享誉世界 20 载，但其服务停止后的责任该由谁承担一直悬而未决，即便如此，日本仍然还有许多旧型号的服务器在运行。

——藤井太洋《合作》

本书为 2002 年出版的《Ruby 基础教程》的修订版，第 4 版是在 2010 年出版的第 3 版的基础上修订的。为了配合 Ruby 2.0 正式版的发布，第 4 版追加了关键字参数、脚本默认编码为 UTF-8 等新特性的内容。在第 3 版中与 1.8 版相关的内容，本次修订原则上都未保留。为了兼容 1.8 版以及 1.9 版，第 3 版有时会对相同的内容分别说明，而第 4 版由于不需要考虑兼容性的问题，因此在内容说明上更为简洁。

当然，现在还有不少活跃在第一线的程序是用 1.8 版实现的，也有很多读者平时还在使用 1.8 版的环境。可以说，这就是 1.8 版的伟大之处，它为 Ruby 的发展立下了汗马功劳。话虽如此，对于初次接触 Ruby 的读者来说，还是希望大家使用最新版本的 Ruby。

第 4 版（日文版）出版于 2013 年，Ruby 当时已经 20 岁了。20 年来，它的价值不可撼动、不容置疑，但这毕竟是过往的辉煌。现在开始接触 Ruby 的初学者们，承担着继续创造新产品、新文化的任务，他们是支持 Ruby 不断发展的不可或缺的力量。希望本书有幸能为这样的 Ruby 初学者们提供帮助。

# 谢辞

---

## 两位著者的谢辞

与大多数图书一样，本书从初版到第 4 版，多蒙各方协助才得以最终出版。在此，谨向下面这些为本书提供帮助的诸多人士表示深深的谢意：本书的主编松本行弘先生，给予我们撰写本书机会的渡边哲也先生，因我们迟交原稿而造成诸多不便的 SB Creative 出版社的杉山聰先生与 TOP STUDIO 公司的武藤健志先生，协助我们提升原稿质量的麻耶（以下请允许我们一并省略敬称）、青木 miya、叶山响、加藤希、Twinspark 公司的诸位、菱沼雄太、nahi、安藤叶子、小仓正充、takkanm、松田明、高桥 yurie，为整理 Ruby 参考手册及各种文档而付出努力的 Rubyist ML 的诸位，收录在本书的 RAA 的应用程序、类库、文档的诸位作者，维护、运营 Ruby Web 站点的 webmaster 的诸位，在 Ruby 各 ML 中提供有趣且有意义的话题的诸位。最后再次对 Ruby 之父——松本行弘先生，以及众多的 Ruby 开发者们，致以最衷心的感谢。谢谢大家！

## 高桥征义的谢辞

本书是一本合著，同时也是我执笔的第一本书，可以说是我的原点。初版、修订版皆蒙受各方的大力支持，图书内容得以不断改进，我也成长了很多。非常感谢大家！

最后，对一直支持我生活、工作的妻子道一声感谢，辛苦了！

## 后藤裕藏的谢辞

连续 10 载，我一直从事着与本书相关的工作，这令我非常欣慰。本书见证了我人生中各种重要时刻。非常感谢以各种形式给予我支持的各位。在我专心工作期间，我牺牲了与家人共处的时间。对一直关心、激励我的妻子与儿子表示最衷心的谢意，谢谢你们的理解与陪伴！