Learning Perl

第五版
涵盖 Perl 5.10

Perl 语言入门

Randal L. Schwartz,
Tom Phoenix & brian d foy 著
盛春 蒋永清 王晖 译

O'REILLY®
東南大學出版社

Perl 语言入门

Perl 语言入门

第五版

Randal L. Schwartz,
Tom Phoenix & brian d foy 著
盛春　蒋永清　王晖译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

Perl语言入门:第5版 /（美）施瓦茨（Schwartz, R. L.），（美）菲尼克斯（Phoenix, T.），（美）福瓦（Foy, B. D.）著；盛春，蒋永清，王晖译．－南京：东南大学出版社，2009.8

书名原文：Learning Perl, Fifth Edition

ISBN 978-7-5641-1763-4

Ⅰ.P… Ⅱ.①施… ②菲… ③福… ④盛… ⑤蒋… ⑥王…

Ⅲ.PERL语言－程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2009）第125922号

江苏省版权局著作权合同登记

图字：10-2008-353号

Perl语言入门（第五版）

出版发行：东南大学出版社
地　　址：南京四牌楼2号　　邮编：210096
出 版 人：江　汉
网　　址：http://press.seu.edu.cn
电子邮件：press@seu.edu.cn
印　　刷：扬中市印刷有限公司
开　　本：787毫米×980毫米　16开本
印　　张：21.5印张
字　　数：361千字
版　　次：2009年8月第1版
印　　次：2009年8月第1次印刷
书　　号：ISBN 978-7-5641-1763-4
印　　数：1~4000册
定　　价：48.00元（册）

本社图书若有印装质量问题，请直接与读者服务部联系。电话（传真）：025-83792328

O'Reilly Media, Inc.介绍

O'Reilly Media, Inc.是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

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

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

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

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

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目录

前言 1

第一章 简介 9

问题与答案 9

“Perl”这个词是什么意思？ 12

如何取得 Perl？ 17

我该怎么编写 Perl 程序？ 20

走马观花 25

习题 26

第二章 标量数据 28

数字 28

字符串 31

Perl 内建警告信息 35

标量变量 37

用 print 输出结果 39

if 控制结构 43

获取用户输入 44
chomp 操作符 45
while 控制结构 46
习题 48

第三章 列表与数组 49

访问数组中的元素 50
特殊的数组索引值 51
列表直接量 52
列表的赋值 54
字符串中的数组内插 57
foreach 控制结构 58
标量上下文与列表上下文 60
列表上下文中的 <STDIN> 63
习题 65

第四章 子程序 66

定义子程序 66
调用子程序 67
返回值 68
参数 69
子程序中的私有变量 71
长度可变的参数列表 72
关于词法（my）变量 74
use strict 编译命令 75
return 操作符 77
非标量返回值 79
持久性私有变量 80
习题 81

第五章 输入与输出 83
读取标准输入 83
钻石操作符输入 85
调用参数 87
输出到标准输出 88
使用 printf 格式化输出 91
文件句柄 93
打开文件句柄 95
用 die 处理严重错误 98
使用文件句柄 101
复用标准文件句柄 102
使用 say 来输出 103
习题 104

第六章 哈希 105
什么是哈希？ 105
访问哈希元素 108
哈希函数 113
哈希的典型应用 115
%ENV 哈希 117
习题 118

第七章 漫游正则表达式王国 119
什么是正则表达式？ 119
使用简易模式 120
字符集 126
习题 128

第八章 以正则表达式进行匹配 129

以 m// 进行匹配 129
可选修饰符 130
锚位 132
绑定操作符 =~ 133
模式串中的内插 134
捕获变量 135
通用量词 141
优先级 142
模式测试程序 144
习题 145

第九章 用正则表达式处理文本 146

用 s/// 替换 146
可选修饰符 148
split 操作符 149
join 函数 150
列表上下文中的 m// 151
更强大的正则表达式 152
习题 159

第十章 其他控制结构 160

unless 控制结构 160
Until 控制结构 161
条件修饰词 162
裸块控制结构 163
elsif 子句 164
自增和自减 165
for 控制结构 167
循环控制 170

三目操作符 ?: 174
逻辑操作符 175
习题 180

第十一章 Perl 模块 181
寻找模块 181
安装模块 182
使用简单模块 183
习题 190

第十二章 文件测试 191
文件测试操作符 191
localtime 函数 200
按位运算操作符 201
习题 202

第十三章 目标操作 203
在目录树中移动 203
文件名通配 204
文件名通配的另一种语法 205
目录句柄 206
递归的目录列表 208
操作文件与目录 208
删除文件 208
重命名文件 210
链接与文件 211
建立及移除目录 216
修改权限 217
更改隶属关系 218

修改时间戳 218
习题 219

第十四章 字符串与排序 221
在字符串内用 index 搜索 221
用 substr 处理子串 222
高级排序 226
习题 232

第十五章 智能匹配与 given-when 结构 233
智能匹配操作符 233
智能匹配操作的优先级 236
given 语句 237
多个项目的 when 匹配 242
习题 243

第十六章 进程管理 245
system 函数 245
exec 函数 249
环境变量 250
用反引号捕获输出结果 250
将进程视为文件句柄 254
用 fork 开展地下工作 256
发送及接收信号 256
习题 259

第十七章 高级 Perl 技巧 260
用 eval 捕获错误 260
用 grep 来筛选列表 263

用 map 对列表进行转换 264
不带引号的哈希键 265
切片 265
习题 270

附录 A 习题解答 271

附录 B 超越小骆驼 305

前言

欢迎阅读《Perl 语言入门》第五版，此版本顺应 Perl 5.10 及其后续版本的新特性而更新。当然，如果你还在用 Perl 5.6（这个版本已经发布很久了，你还没想过升级？），这本书同样适用于你。

假如你正在寻找用 30 到 45 小时就能掌握 Perl 语言编程的最佳方式，那么你已经找到了！在后面的 300 多页里，我们会提供精心安排的入门指引，介绍这个在互联网中担负重任的程序语言。它也是最受全世界系统管理员、网络黑客（web hacker）及业余程序员青睐的程序语言。

我们不可能只花几小时就把 Perl 的全部知识传授给你，会这么保证的书大概都撒了一点谎。相对地，我们慎选了 Perl 中完整又实用的部分供你学习。这些材料足以编写 128 行以内的小程序，大约 90% 的 Perl 程序都不需要很多篇幅。当你准备继续深入时，建议您阅读《Intermediate Perl》这本书，该书涵盖了许多本书舍去不讲的部分。此外，我们还纳入了许多知识点以便后续的扩展和研习。

每章的内容并不多，可以在一两个小时之内读完。各章后面都有一系列的习题，帮助你巩固刚学到的知识，在附录 A 中还附有习题解答，供你比对思考。因此，本书可以说是相当适合作为“Perl 入门”的课堂教材。我们对此有第一手的经验，因为本书的内容几乎是逐字逐句从“Learning Perl”教学中萃取出来的，而这正是我们教过上千名学生的招牌课程。话虽如此，我们也将本书设计成适合自学的形式。

虽然 Perl 是活生生的“Unix 工具箱”，但你并不需要是 Unix 大师，甚至也不必懂 Unix 就可以使用本书。除非特别注明，否则我们所提到的一切都可以同样应用到 Windows 版本的 ActivePerl（ActiveState 出品），以及许许多多其他新潮的 Perl 版本。

阅读本书之前，虽然不需事先具备任何 Perl 的基础，但我们还是衷心希望你先熟悉一

些写程序的概念，像变量（variable）、循环（loop）、子程序（subroutine）和数组（array）以及最重要的“用你最熟悉的文本编辑器来编辑源代码”。我们不会花时间来尝试说明这些概念。有些人平生所学的第一个程序语言就是 Perl，并因学习本书而获得成功，我们相当高兴有这样的例子，但是我们并不敢保证每个人都能取得一样的成果。

排版约定

本书使用以下的字体惯例：

等宽字（`Constant width`）

用于方法名称（method name）、函数名称（function name）、变量（variable）、属性（attribute）以及程序代码范例。

等宽黑体字（**`Constant width bold`**）

用于用户所输入的内容。

等宽斜体字（*`Constant width italic`*）

用于程序代码中可被替换的项目（例如：*`filename`*，你可以将它代换成实际的文件名）。

斜体字（*Italic*）

用于正文所提到的文件名称、URL、主机名称、第一次提及的重要词汇以及命令。

脚注

一般附加在括号之内，初次（也许是第二次、第三次）阅读本书时应该略过。有一些不完全正确的用语是为了简化说明，而脚注会说明事实。通常脚注中的资料是高级主题，不会在本书其他部分讨论到。

如何与我们联系

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然在所难免。如果你发现有什么错误或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

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

中国：

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

你也可以发送电子邮件给我们。若是想要加入我们的邮件列表，或是索取书籍类目手册，可以发邮件到：

info@oreilly.com

询问技术问题或对本书进行评论，请发送邮件到：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

我们还为本书建立了一个专属网页，你可以在此找到关于本书的相关信息，包括范例程序、勘误表，以及将来再版的计划等。该网页还提供一系列可下载的文本文件（以及一些Perl程序），在做本书练习的时候可以直接使用。此页面地址为：

http://www.oreilly.com/catalog/9780596520106（原版书）
http://www.oreilly.com.cn/book.php?bn=978-7-5641-1763-4（中文版）

其他关于本书或其他书籍的信息，请访问O'Reilly的网站：

http://www.oreilly.com
http://www.oreilly.com.cn

使用程序代码范例

本书的目的是让你能够在实践中解决问题。基本上，你不用事先联络我们就可以使用本书所提供的程序代码及文件，除非是大量复制。举例来说，在你的程序中，若用到几段本书中的程序代码，不需要经过我们的同意；但是做成光盘发布、销售O'Reilly书中的例子，则必须经过授权。回答别人的问题时，引用本书的文字和程序代码，也不需要经过我们的同意；但在你的产品文件中，若大量加入本书的文字与程序代码，则必须经过授权。虽非必要，但我们会十分感谢你在引用本书的内容和范例时提到出处。完整的信息通常包括书名、作者、出版商及ISBN编号。例如："Learning Perl, Fifth Edition, by Randal L. Schwartz, Tom Phoenix and brian d foy. copyright 2008 O'Reilly Media, Inc., 978-0-596-52010-6。"。如果你的情况有别于上述情形，并心存疑问的话，请给我们来信permissions@oreilly.com。

本书的历史

为了满足读者的好奇心，Randal 在这里告诉你关于这本书的来历：

1991 年我跟 Larry Wall 写完第一本《Perl 语言编程》之后，硅谷的 Taos Mountain Software 公司跟我联络，要我准备一些培训课程，内容包含 12 次左右的课程，并训练他们的教师继续开课。我就按约写了这个课程给他们【注 1】。

在课程进行了三四次之后（1991 年底），有个人走到面前跟我说："不瞒你说，我真地很喜欢《Perl 语言编程》这本书，但是这堂课的教材更容易吸收，你真地应该写一本像这个课程的书。"这听起来像是个好机会，所以我开始认真地考虑这个点子。

我写信给 Tim O'Reilly，附上了一份企划书。这是以 Taos Mountain 课程纲要为基础，再根据课堂上的观察调整并修改了一些章节。这可能是有史以来我的企划书最快被接受的记录 —— 我在 15 分钟后收到了 Tim 的回信："我们一直在等待你的第二本书。《Perl 语言编程》太热销了。"接下来的一年半时间里，我就努力完成了第一版的《Perl 语言入门》。

在那段时间里，我找到硅谷以外教授 Perl 的机会【注 2】，所以我就以正在编写阶段的《Perl 语言入门》为蓝本制作了一套课程。我为许多不同的客户教课（包括我的主要签约人 Intel Oregon），并利用上课所得到的响应进一步微调本书的草稿。

第一版在 1993 年的 11 月 1 日【注 3】问世，销售空前成功，甚至很快就追上了《Perl 语言编程》的销量。

在第一版的封底上这么写着："由卓越的 Perl 讲师所著"，事后证明这是正确的预言。随后的几个月里，我收到来自美国各地的电子邮件，邀请我到他们那里教 Perl。接下来的 7 年中，我的公司成了全球领先的 Perl 现场培训公司，我个人的飞行里程数也飙升到了百万英里。之后互联网的兴起更是锦上添花，许多站长都采用 Perl 作为内容管理、交互式 CGI 及网站维护的语言。

注 1：在合约中，我保留了习题的所有权，我希望有一天能以不同的方式来使用它们，比如说我以前曾经写过的杂志专栏。习题是 Taos 公司的课程里唯一还能在本书中出现的东西。

注 2：我与 Taos 公司的合约有条独特的条款，因此不能在硅谷教授类似的课程，我也遵守了此条款很多年。

注 3：这个日期我记得很清楚，因为那也是我由于一些跟计算机有关的行为在家被逮捕的日子。关于我与 Intel 公司的合约，我被判有罪。

我跟 Stonehenge 的首席培训师兼内容经理 Tom Phoenix 密切合作了两年。我请他对 Llama 课程做实验，把某些东西移来移去，再打散一些内容。当他带着我们认为是最好的修订本出现时，我就联络 O'Reilly，说："是该有本新书的时候了！"于是第三版就这么诞生了。

在小骆驼书第三版问世的两年后，我和 Tom 决定把一些"高级"的课程移出来成为一本独立的、专门给需要写"100 到 10000 行代码"的人看的书，那就是在 2003 年完成的羊驼书。

不过，在我的同事 brian d foy 从海湾战争回来之后，同样是讲师的他注意到教材必须进一步适应普通学生的需求，因此这两本书都应该适当地改写。于是他对 O'Reilly 推销这个想法，希望在 Perl 6 完成之前进行小骆驼书与羊驼书的最后一次改版（但愿如此）。而此版本的确反映了那些变动的需求。我很少需要给 brian 什么建议，他一向都是顶尖的作者，在写作团队里面他给人的感觉就像是尽责的英国管家。

2007 年 12 月 18 日，"Perl 5 掌门人"（perl 5 porters）发布了 Perl 5.10，一个标志性的版本，融入了众多新特性。之前的 5.8 版本专注于 Perl 的基础架构改良和 Unicode 支持。而最新的版本，以稳固的 5.8 为基础，增加了一系列崭新的特性，特别是那些取自正在开发中（尚未发布）的 Perl 6 的一些理念。其中某些特性，诸如正则表达式里的命名捕捉，比起传统做法来要好很多，对 Perl 初学者来说也更容易掌握。我们未曾想过本书会有第五版，但 Perl 5.10 实在是太有趣了，我们无法故步不前。

读者可能注意到了这一版跟前一版本的某些差异：

- 某些内容为 Perl 5.10 而更新，某些代码仅在此新版本中可用。在讨论 Perl 5.10 的特性时，我们会在行文中加以提示说明。对于那些代码片段，我们也一律使用特殊的 `use` 语句加以区别，提示你使用正确的版本：

  ```
  use 5.010; # 当前脚本需要 Perl 5.10 或更高版本
  ```

  如果在代码示例中没有看到 `use 5.010` 这句话，就说明它也可以在 Perl 5.6 以上版本中工作。要看当前使用何种版本，可在命令行使用 `-v` 参数查看：

  ```
  prompt% perl -v
  ```

  这里列举一些我们将要讨论到的 Perl 5.10 新特性，及其相应章节。介绍这些新特性的同时，我们还会展示如何用老的方法实现相同的功能：

- 有很多 Perl 5.10 的新特性是有关于正则表达式的，包括相对反向引用（第七章），新的字符集简写（第七章）和命名捕捉（第八章）。

- Perl 5.10 包含一个新的条件切换语句，称作 given-when。我们会在第十五章中同智能匹配操作符一同讲解。
- 子程序现在可以像 C 一样拥有静态变量，不过在 Perl 里称为 state 变量。这类变量能在子程序的多次调用之间保留其中的值，且作用域限于子程序内。我们会在第四章中阐述此特性。

致谢

来自 Randal

我想要感谢 Stonehenge 过去与现在的讲师们（Joseph Hall、Tom Phoenix、Chip Salzenberg、brian d foy 与 Tad McClellan），谢谢他们愿意每周到教室授课并且带回自己的笔记，注明哪部分有用（及哪部分没用），如此我们才能调整这本书的内容。我要特别点名 Tom Phoenix，我的共同作者与事业伙伴，他花了大量时间改进 Stonehenge 的 Llama 课程，也为本书注入了主要的原始内容。还有 brain d foy，在第四版中担任了主要的写作任务，从而完成了我从收件箱转发的无数待办事项。

此外，我还要感谢 O'Reilly 的每一位人员，尤其是我们非常有耐心与眼光的编辑 Allison Randal（不是我的亲戚，但她的姓氏拼法真好）。还有 Tim O'Reilly 本人，是他让我在一开始就有写作大骆驼与小骆驼这两本书的机会。

我由衷感谢过去购买本书的读者，这些钱让我免于流浪街头与夜宿囚牢；感谢我班上的学生，他们把我训练成为一名更好的讲师；还有“财富一千”上大排长龙、在过去选购我们的课程、未来也会继续捧场的客户们。

和以前一样，我得特别感谢 Lyle 与 Jack，他们教会了我几乎所有关于写作的知识，我永远不会忘记你们。

来自 Tom

我必须附和 Randal 对 O'Reilly 的每个人致上谢意。在第三版的时候，我们的编辑是 Linda Mui，她细心地指出书中太过火的玩笑与脚注，当然留下来的那些也不是她的错。她与 Randal 在整个写作过程中不断指导我，我非常感激。在第四版中，Allison Randal 成为了我们的新编辑，我也由衷感谢她。

另一些跟 Randal 一样要感谢的是 Stonehenge 其他的讲师们，当我临时更新课程教材以尝试新的教学技巧时，他们几乎不曾抱怨过。在教学方法上，你们提出了许多我未曾想过的主意。

多年来，我在俄勒冈科学与工业博物馆（Oregon Museum of Science and Industry，OMSI）工作，而我要感谢那里的人们，他们迫使我磨练自己的教学技巧，让我学着在每个聚会、展示与讲解中插入一两个笑话。

谢谢新闻组上的伙伴们，你们对我的每个贡献都给予赞赏与鼓励。如同以往，希望这些对各位有所帮助。

谢谢我的众多学生，在我尝试变换角度，以解释某个概念的时候，他们能提出疑问（附赠一脸迷惑）。希望本书的新版可以解除剩下的难题。

当然，最诚挚的感谢特别留给与我共同创作的作者，Randal。他给予我高度自由，让我可以在课堂上（以及书中）尝试各种讲述方法，而且时刻敦促我将这些阐述写入书中。还有一点务必要和 Randal 说的：我被你深深感动，你热心劝勉他人，免于为了像你一样的官司而耗费大量的时间与精力，你是良好的典范。

谢谢我的妻子 Jenna，谢谢你如此温柔体贴，为生活中大大小小的事感谢你。

来自 brian

我必须先谢谢 Randal，因为我就是从本书的第一版开始学习 Perl 的。而在 1998 年他要我进入 Stonehenge 开始讲课时，我又得再仔细读一遍！要学一件事的最好方法通常就是先去教。在那之后，只要他认为我该学的，Randal 都经常指点我，不管是 Perl 还是其他方面的事。例如在一次网络会议上，他决定我们应该用 Smalltalk 来展示，不要用 Perl。我总是很惊讶于他渊博的知识。一开始就是他建议我写与 Perl 有关的东西，现在我才能够回应他，帮助完成了本书的编写。Randal，我感到非常荣幸。

在任职于 Stonehenge 的期间，跟 Tom Phoneix 见面的时间恐怕还不到两星期，但我多年以来都是用他的教材来上我们的 Perl 课程。他的版本后来成为本书的第三版。在使用他的教材时，我也学到了解释任何概念的新方式，也深入了 Perl 的更多领域。

说服 Randal 让我参与小骆驼书的改版之后，我负责写出企划书、维护全书大纲以及版本控制。我们第四版的编辑 Allison Randal 不但在这些事情上给予了很多帮助，也收到了我的许多电子邮件而没有怨言。

特别感谢跟 Perl 没有关系的 Stacey、Buster、Mimi、Roscoe、Amelia、Lila 以及许多人，不但在我很忙碌的时候带我散心，还在我不能出去玩的时候一直跟我闲聊。

来自我们大家

感谢我们的校稿人员 David H. Adler、Andy Armstrong、Dave Cross、Chris Devers、Paul Fenwick、Stephen B. Jenkins、Matthew Musgrove、Steve Peters 与 Wil Wheaton，谢谢他们对本书草稿所提的建议。

感谢我们的众多学生，这些年来让我们知道这个课程的内容在哪些部分需要改善。因为你们，我们今天才得以对本书如此自豪。

感谢诸多 Perl 推广组（Perl Mongers）在我们访问各位的城市时给我们宾至如归的招待。什么时候我们会再来的。

最后，向我们的朋友 Larry Wall 致上最诚挚的谢意，他慷慨与大家分享这个新颖又强大的工具（也是玩具），让我们能够更快、更简单以及更有趣地完成我们的工作。

第一章

简介

欢迎阅读这本小骆驼书（Llama book）！

从 1993 年迄今（第五版），本书已拥有大约 50 万名的读者。我们最大的期盼是读者们喜欢本书的内容。可以确定的是，在写作当时我们是乐在其中的【注 1】。

问题与答案

你或许有些关于 Perl 的问题，如果你已经快速翻阅过本书，大概知道书中的内容，也可能有些关于本书的问题。所以，我们打算利用这一章予以回答。

这本书适合你吗？

如果你的个性和我们差不多，现在你可能正站在书店里【注 2】，考虑是否应该买下这本小骆驼书来学习 Perl，或是选择身旁讲别的语言的书，像以蛇、某种饮料或是某个字母

注 1：明确地说，本书第一版的作者是 Randal L. Schwartz，第二版是 Randal L. Schwartz 与 Tom Christiansen 合著，第三版是 Randal 与 Tom Phoenix 合著，而现在是 Randal、Tom Phoenix 与 brain d foy 合著。因此，本书提及的“我们”指的是最后那三位。现在，如果你怀疑我们如何能在卷首就说“写作当时我们也乐在其中”，答案其实很简单：我们写书的方式，是从本书的最后开始，从后往前。我们知道这听起来很奇怪，但老实说，当我们写完索引之后，剩下来的部分就完全不成问题了。

注 2：事实上，如果你真的跟我们差不多，那么现在你会是在图书馆，而不是书店。我们都是蓝领阶层。

命名的书【注3】。还有两分钟，这家店的经理就会过来告诉你：这里不是图书馆【注4】，你要么就买点东西，要么就给我滚蛋。也许你想利用这两分钟马上看个简单的 Perl 程序，这样你才有办法知道 Perl 究竟有多强大，它到底能做些什么。如果你是这么想的话，请参考本章后面的“走马观花”一节。

为何有这么多的脚注？

感谢你注意到了，这本书里真的有很多的脚注。忽略它们吧。之所以需要它们是因为 Perl 到处充斥着特例。这算得上是好事，因为现实生活中也是到处都有特例。

就是因为有这些特例，我们不能昧着良心说“fizzbin 操作符可用来对 hoozistaitc 变量进行 frobnicate 处理”，却不加上脚注来说明例外情况【注 5】。我们很有良心，所以加上了这些脚注。但即使略过不读，你也可以问心无愧（这段话能成立，实在很有意思）。

许多的例外与可移植性有关。Perl 来自 Unix 系统，而且目前仍扎根于 Unix 中，与 Unix 息息相关。但无论是因为在 Unix 以外的系统上运行（或是别的什么原因）造成的，我们总是尽力呈现可能出现的意外状况。我们希望不懂 Unix 的读者们也能认为这是一本相当好的 Perl 入门书（而且你也可以因此而免费学到一点 Unix 的知识）。

其他的例外状况则与所谓的“80-20”定律有关。这是说 Perl 里 80% 的功能可以用文档中 20% 的部分加以描述，而另外 20% 的功能却需要占据其他 80% 的篇幅。所以，为了保持本书的篇幅短小，我们在正文中介绍那些简单明了的东西，把意味深长的部分留在脚注中介绍（脚注将用小一号的字体，这样就可以用更少的位置写更多的东西）【注6】。当你在不看脚注的情况下将本书读完一遍后，也许是为了查资料，你会想要翻回到之前的某些章节。此时，如果你已经好奇得忍不住，那么就去读读脚注吧。它们中有很多只不过是计算机笑话而已。

注 3：在你来信指出 Python 源自一出喜剧而非蛇之前，我们得先予以解释，其实我们所说的是 CORBA 而非 Python，我们当时误以为 CORBA 就是 cobra（眼镜蛇）。

注 4：除非它刚好就是。

注 5：除了当你趁星期二停电的机会玩 fizzbin 的时候（美剧《星际迷航》中，fizzbin 是一个扑克游戏，有很多奇怪的例外规则，比如周二不能翻开第二张牌）；或者是在 5.6 版之前的 Perl 中，在某个定义了原型的子程序中的循环块内使用 use integer 编译命令。

注 6：我们甚至讨论过把整本书做成脚注以节省页数，但是“脚注的脚注”听起来有点怪。

关于习题和解答？

每章的结尾都有练习题，因为我们都曾以相同的教材教过上千名学生【注7】。所以我们精心设计了这些习题，让你有机会尝试一般人常犯的错误。

不是说我们希望你犯错，而是你需要这样的机会。因为大部分的错误在你的 Perl 编程生涯中都会出现，还不如现在先体验。有了前车之鉴（完成练习时你犯过的那些错误），在你赶进度交付 Perl 程序时就不会重蹈覆辙了。另外，在这个过程中我们会一直伴随着你：附录 A 里有我们对习题的解答以及一些相关的说明 —— 说明你犯的（以及没犯的）错误。当你做完这些习题之后，可以核对一下答案。

在你努力尝试克服问题之前，尽量先不要偷看答案。在自己找出答案的情况下完成习题，学习的效果会比直接看答案来得好。就算一直想不出答案，也不要老是用头撞墙，不用那么在意，就先翻到下一章吧。

即使你没有犯任何错误，在你做完习题之后也应该看一下解答，其中的解释会说明一些（乍看之下或许）并不明显的细节。

想要额外再练习的话，可以翻翻《Learning Perl Student Workbook》（O'Reilly） 这本书，每个章节都增补了许多习题。

习题前标的数字代表什么意思？

每个习题之前都会有个数字，是以方括号框起来的，看起来像这样：

1. [2] 当方括号里的数字 2 出现在习题之前时，表示什么意思？

这个数字是我们（非常粗略地）估计你完成这部分的练习需要花费的时间。那只是非常粗略的估算，所以如果你已经全部完成（包括编写、测试和调试）而只用了一半的时间，或是花了两倍的时间还没完成，都请不要太惊讶。另一方面，如果你真的被难倒了，我们也不会告诉别人，你的答案是偷看附录 A 得来的。

如果我是教 Perl 课程的讲师？

如果你想在自己的课程里以本书作为教材（多年来有许多人都这样做），请留意我们对各章习题的设计，都尽量让大部分的学生能在 45 分钟到 1 小时之内完成，再留下一些休息的时间。某些章节的习题需要的时间会少一些，某些章节则要多一些。会出现这种

注7： 不是一次教那么多。

情况，是因为在填完方括号里的数字后，我们发现自己竟然不会算加法（好在我们还知道要怎么让计算机帮我们做这件事）。

之前提到过，我们还有一本辅助用书，《Learning Perl Student Workbook》，其中为每个章节都额外增加了若干习题。如果你已经有第四版的这本工具书的话，请注意调整一下章节的顺序，这次我们新增了一章，并作了些次序上的调整。

“Perl”这个词是什么意思？

Perl 是实用摘录与报表语言（Practical Extraction and Report Language）的缩写，不过有时候也被称作病态折中式垃圾列表器（Pathologically Eclectic Rubbish Lister），此外还有其他不同的全名展开方式。Perl 是个溯写字（backronym），而不是缩写词；是 Perl 这个词本身先被使用，后来才给出展开的词作诠释。这也就是为什么“Perl”不会全部用大写来表示的原因。我们无需争辩这两种全名哪个是对的，它们都受到 Larry 的认可。

你可能会在某些文章里看到以小写 p 来表示“perl”。一般说来，大写 P 表示的“Perl”指的是程序语言，而小写 p 表示的“perl”指的则是实际编译并运行程序的解释器。

Larry 为什么要创造 Perl？

在 20 世纪 80 年代中期，他想要实现某个 bug 汇报系统，从而为类似新闻组的文档系统产生报表，这超出了 *awk* 的能力范围。身为懒惰的程序员【注 8】，Larry 决定写个多用途的工具，让它不仅能解决这个问题，还能在别的地方使用。Perl 第零版就这样诞生了。

Larry 为什么不用其他的语言？

世界上不缺乏程序语言，不是吗？但是在当时，Larry 却找不到任何真正符合他需要的语言。如果时下的某种语言在当年就能够出现的话，Larry 或许就会使用它了。他当时需要的是像 shell 或 *awk* 一样能够快速编程，又具有类似 *grep*、*cut*、*sort* 和 *sed* 的高级功能【注 9】，而不必回头使用像 C 这种类型的语言。

注 8：我们说 Larry 懒惰，并不是说他的坏话，懒惰其实是一种美德。手推车是由懒得扛东西的人发明的；书写是由懒得记忆的人发明的；Perl 的创造者也是懒人，若不能发明一个新语言就懒得做事。

注 9：要是你不知道这些是什么东西，请别担心。重点只在于它们是 Larry 当时手上能够用到的 Unix 工具程序，但是功能不够强大。

Perl 试图填补低级语言（如 C、C++ 或汇编语言）和高级语言（如 shell 编程）之间的空隙。低级语言通常既难写又丑陋，但是执行速度很快而且不受限制。在任何机器上，都很难赢过写得好的低级程序的执行速度。它们几乎可以做所有事。高级语言是另一个极端，它们通常速度缓慢、难写又丑陋，并且限制重重。如果系统上不提供执行必要功能的某个命令，那么你的 shell 程序会有很多事情都不能做。而 Perl 则相当容易，几乎不受限制，速度通常又很快，就是看起来有点丑陋。

我们来看看上面所说的 Perl 的四个特性：

首先，Perl 很容易。不过接下来你会发现这指的是容易使用。学习 Perl 并不会特别容易。如果你会开车，你一定是花了好几个星期或几个月的时间来学习，最后开起来才会那么容易。当你花在写 Perl 程序上的时间和学开车的时间一样长时，Perl 对你而言就会很容易了。

Perl 几乎不受限制，几乎没什么事是 Perl 办不到的。你大概不会想用 Perl 来编写中断－微内核层次（interrupt-microkernel-level）的设备驱动程序（尽管已经有人这么做了），但是一般人用来处理日常琐事的程序，从写完即丢的小程序到企业级的大型应用程序，都很适合用 Perl 来写。

Perl 的速度通常很快。这是因为所有 Perl 的开发者同时也都是用户，所以我们都希望它快。假设有人为 Perl 加上某个确实非常酷的功能，可同时会让其他程序变慢，那么 Larry 一定会拒绝加入这个新功能，直到我们找出让它变快的方法为止。

Perl 有点丑陋，这是事实。Perl 的标志是骆驼，这来自于值得尊敬的大骆驼书（即《Perl 语言编程》，由 Larry Wall、Tom Christiansen 和 Jon Orwant 合著，O'Reilly 出版）的封面，这本小骆驼书（以及姐妹作羊驼书 Alpaca）算是该书的表亲。骆驼长得也有点丑陋，但是它们努力工作，哪怕是在严酷的环境下也一样。骆驼能在种种不利的条件下帮你把事情搞定，哪怕它们长相丑陋，而且气味更糟，有时候还会对你吐口水。Perl 就有一点像这样。

Perl 用起来究竟是容易还是困难？

它简单好用，但有时候不太好学。当然，这只是一般而言。在 Larry 设计 Perl 的时候，他必须做出许多取舍。每当他有机会让程序员感到方便、但让学 Perl 的人觉得不便时，他几乎每次都站在程序员这边。这是因为你只需要学一次 Perl，但却可以常常使用【注 10】。

注 10：如果你每周或每个月只花几分钟的时间在程序设计上，容易学习的语言会比较合适，因为在你下次使用时可能就忘光了。Perl 是为每天至少花 20 分钟写程序（而且以 Perl 程序为主）的程序员而设计的。

Perl 有不少简便操作方式可以让程序员节省时间。举例来说，大部分的函数都具有默认行为，它通常就是你使用该函数时想采取的方式。因此，你会看到如下的 Perl 程序代码：【注 11】

```
while (<>) {
  chomp;
  print join("\t", (split /:/)[0, 2, 1, 5] ), "\n";
}
```

要是不使用 Perl 的默认行为与简写，那么上面这段程序的长度大概会增长十几倍，这样一来，阅读与编写的时间也会大幅增加。它会用到更多的变量，让维护和调试也更加困难。如果你已经能看懂一点 Perl，你将会发现上面的程序里没有变量，这是问题的重点。这里用到的变量都是以默认行为来工作的。但是，让程序员容易使用的这些功能是要在学习时付出代价的，你得先学会这些默认行为与简写才行。

例如，在英语里大家常会使用缩写，并不会觉得不妥。没错，“will not”跟“won't”这两种写法的意义一模一样。但是大家都会说“won't”，而不太会说“will not”。一来因为比较省时间，二来因为大家都熟悉这样的模式。同样，Perl 也会缩写常用的“字句”，把许多难写的程序浓缩成简洁有力的“成语”，好让维护人员能够快速地“听说”Perl。

一旦熟悉 Perl 之后，你就可以花更少的时间去摆弄 shell 的引用（或 C 语言的声明），有更多的时间来浏览网站。这是因为 Perl 能让你事半功倍。Perl 简明的语法让你能够（毫不费力地）建立很酷而且更流畅的解决方案，或是用途广泛的工具程序。由于 Perl 既跨平台又随处可用，所以这次实现的工具可以在下次的任务里沿用，让你有更多的时间浏览网站、充实自己。

Perl 是非常高端的语言。这表示程序代码的密度相当高，Perl 程序代码的长度大约是等效的 C 程序代码的 30% 到 70% 左右。这使得编写、阅读、调试和维护 Perl 程序的效率非常高。哪怕只写过一点程序的人都明白，当子程序小到能够放进一个屏幕时，开发时就能免于上下滚动代码。此外，既然程序里 bug 的数量大致与源代码的长度成正比【注 12】（而不是与程序的功能成正比），较短的 Perl 程序代码平均起来会含有较少的 bug。

像任何语言一样，Perl 也可以变成“只写”（write-only），它可以写出让人完全看不懂的程序。但只要你稍微用心，就可以避免这项常见的恶名。没错，Perl 程序对门外汉来

注 11：在这里我们无法详细解说，不过这个例子程序将会从文件读入一些数据，并把数据从原来的格式转成另一种格式。程序里用到的所有功能，本书都将会介绍到。

注 12：要是程序里有任何一段的篇幅超过屏幕的显示范围，bug 的数量还会显著上升。

说，看起来可能像线路噪声；但是对经验丰富的 Perl 程序员来说，它看来就像大交响乐团的总谱。你只要遵照书里的指引，就能写出易于阅读且易于维护的程序，它们大概赢不了模糊 Perl 代码大赛（Obfuscated Perl Contest）。

Perl 怎么会这么流行？

Larry 稍加测试 Perl，并在各处加点功能之后，把它发布给 Usenet 的读者社群，也就是一般所谓的“网络”。这群散居世界各处的（上万名）用户给了 Larry 反馈，要求 Perl 做这个、做那个，其中有许多事情是 Larry 从来没有想到要让他的小 Perl 去处理的。

结果是，Perl 不断成长。它的功能变多了，能执行它的平台也增加了。当年这个只能在少数几种 Unix 系统上执行的小语言，而今长成了具有上千页的在线自由文档、成打的书籍、数个主流的 Usenet 新闻组（以及成堆的新闻组与邮件列表）、新闻组里无数的读者，还有时下几乎所有系统皆可使用的版本。当然，也还有这本小骆驼书。

Perl 目前的发展如何？

Larry Wall 已经不再亲自编写所有的 Perl 核心代码，但他仍然指引开发的方向并作出关键性的决策。目前维护 Perl 的是一个热心的开发者团队，我们称之为“Perl 5 掌门人”。要跟进他们目前的任务与话题，可以加入他们的邮件列表 *perl5-porters@perl.org*。

在我们写下这段文字的同时（2008 年 3 月），有许多事情正围绕着 Perl 发生。在过去几年里，有许许多多的人投入到下个重要版本的开发中：Perl 6。

但别这样就把 Perl 5 给扔了，因为那是时下最稳定的版本。目前来说，稳定的 Perl 6 还不会那么快出现，Perl 5 也不会这么快消失，人们应该会同时使用这两个版本的 Perl 好几年。Perl 5 掌门人也会一如继往地维护 Perl 5，并且从 Perl 6 中吸收更多好用的点子。而我们继续修订此书，恰恰是因为从刚刚面世的 Perl 5.10 里可以看出 Perl 5 掌门人在后续的 Perl 5.12 上已经展开了相关工作，Perl 5 活力不减。

在 2000 年的时候，Perl 社群正在进行 Perl 的重写，Larry Wall 率先提出了下个主要版本的规划。次年，一个名为 Parrot 的解释器诞生了，但大部分的用户并没有感受到太多变动。而在 2005 年，唐宗汉开始牵头启动 Pugs（Perl User Golfing System）项目，这是以 Haskell 语言实现的轻量级 Perl 6。全世界（不论是 Perl 方面还是 Haskell 方面）的开发人员都参与进来并提供了协助。因为目前还在发展中，所以我们不敢下定论，但你已经可以用 Pugs 来写简单的 Perl 6 程序了。关于 Perl 6 的详情，请看 *http://perlsix.org* 和 *http://www.pugscode.org*。目前就此书而言，我们暂时还不用担心 Perl 6 的事情。

哪些事情最适合用 Perl 来做？

Perl 很适合在三分钟内写出“急功近利”的程序，Perl 也很适合用来编写用处广泛、需要十几个程序员花三年时间完成的大型程序。当然，你会发现大部分的程序从构思到完成测试，只需要花一小时以内的时间。

Perl 擅长处理整体来说“约有 90% 与文字处理有关，10% 与其他事务有关”的问题。这似乎占了当前编程任务需求的绝大部分。在理想的世界中，所有的程序员都会每种语言；进行每个项目时，他们都能选择最适合的语言。通常，他们会选择 Perl【注 13】。虽然在 Larry 创造 Perl 时，那时 Web 甚至还没在 Tim Berners-Lee 的脑海中闪现，可是两者却在网络上结合了。有些人宣称 Perl 在 20 世纪 90 年代初期的扩张，让许多信息的内容能迅速地转换成 HTML 格式，使得互联网得以延续。当然，Perl 也是小型 CGI 脚本（由 Web 服务器运行的程序）的最佳搭档语言。直到现在，许多搞不清楚状况的人，还会提出像“CGI 不就是 Perl 吗？”或“除了 CGI 之外，Perl 还有什么用呢？”这样的问题。我们觉得这些似是而非的问题挺好笑的。

哪些事情不适合用 Perl 来做？

所以，既然 Perl 能做的事情这么多，哪些事*不适合*用它做呢？嗯，如果你想做出封闭式的二进制可执行文件（*opaque binary*），请不要使用 Perl。所谓的“封闭式”，指的是取得或购得你程序的人无法从程序里看到你的秘密算法，因此也无法协助你进行维护或调试。当你把 Perl 程序给某人时，通常给的是源代码，而非封闭式的二进制可执行文件。

不过，如果你需要封闭式的二进制可执行文件，我们必须告诉你其实没有这种东西。只要有人能安装并运行你的程序，他就能将它还原成各种程序语言的源代码。的确，该程序代码可能和原本的不一样，但是它总是某种源代码。不幸的是，要保护你的秘密算法，真正的办法只有一种：聘用足够多的律师。他们能写出一份授权条款，上面说“你可以用这个程序做这件事，但是不能做那件事。要是你违反了我们的规定，我们有足够多的律师能让你后悔莫及”。

注 13：不过，请不要只听我们的一面之词。如果想知道 Perl 和 X 语言哪个比较好，最好的方法就是把两者都学会，看看后来最常用的是哪个，那就是最适合你的语言。反正，你会因为学了 X 语言而增进对 Perl 的了解（反之亦然），所以并没有浪费时间。

如何取得 Perl?

你的机器上可能已经有 Perl 了。至少，我们不管到哪里都可以找到 Perl。许多系统内置 Perl，而系统管理员经常会在单位里的每台机器上安装 Perl。不过就算在系统中找不到 Perl，你总还是可以免费取得它。

Perl 有两种不同的授权条款。对只是使用 Perl 的大部分人来说，这两种条款并没有什么差别。不过如果你想修改 Perl，请详细阅读这两份授权条款，上面对发布变动过的 Perl 有些限制。对于不想修改 Perl 的人而言，授权条款基本上是说："这是自由软件——你爱怎么用都行。"

事实上，它不仅是自由软件，还能在几乎所有自称为 Unix、具有 C 编译器的系统上顺利运行。你只要下载它，键入一两条命令，Perl 就能自行设定与安装。还有更好的办法，就是找到系统管理员，让他键入这一两条命令来帮你安装【注 14】。除了 Unix 和类似的系统之外，对 Perl 上瘾的人们还将它移植到了别的平台上，如 Mac OS X、VMS、OS/2，甚至还包括 MS/DOS 以及所有 Windows 的现代版本。在你看到这句话时，可能又增加了更多平台【注 15】。这些 Perl 的移植版本（*ports*）经常会附上安装程序，甚至比 Unix 的安装程序更容易使用。请参考 CPAN 的"ports"部分的链接。

CPAN 是什么?

CPAN 就是 Perl 综合典藏网（Comprehensive Perl Archive Network），可以说是非常方便的 Perl 卖场。里面有 Perl 本身的源代码、各种非 Unix 系统的安装程序【注 16】、范例程序、说明文档、扩展模块以及跟 Perl 相关的历史消息。简而言之，CPAN 是包罗万象的。

CPAN 有数百个镜像站点分布在世界各地。请从 *http://search.cpan.org/* 或是 *http://kobesearch.cpan.org/* 开始，慢慢浏览整个典藏网吧。如果你无法连上网络，也可以到附近卖技术书籍的书店找找，可能会有 CPAN 部分实用内容构成的 CD 或 DVD 出售。

注 14：如果系统管理员不能安装软件，要他们做什么？如果难以说服管理员安装 Perl，可以用买披萨请客作为交换条件。我们还没碰过能拒绝免费披萨（或其他容易弄到的食物）的系统管理员。

注 15：并非完全如此，在我们写此书时，在 Blackberry 上还不能这么做——这个系统实在是太过繁复交织，尽管已经是简化下来的系统。但已有传闻说它可以在 WinCE 上使用。

注 16：在 Unix 系统上，从源代码重新编译 Perl 几乎总是较好的选择。其他的系统也许没有 C 编译器，也缺少用于编译的工具，所以 CPAN 为它们准备了预先编译过的可执行文件。

不过，请确定它是最新的版本。既然 CPAN 每天都会更新，那么两年前的库存就已经算是老古董了。更好的办法就是找个可以访问网络的朋友，帮你用当天的 CPAN 烧录光盘。

如何取得 Perl 的支持服务？

嗯，你手上有完整的源代码，因此可以自己动手修正 bug！

听起来不怎么样，不是吗？但是这确实是件好事。因为 Perl 没有“源代码保护条款”，所以任何人都可以修正里面的 bug。事实上，在你找到并确认某个 bug 的时候，大概已经有人将它修改好了。Perl 是由全世界的数千人共同维护的。

当然，我们并不是指 Perl 有一大堆 bug，但是它充其量也只是一个程序，所有的程序至少都会有一个 bug。要了解拥有 Perl 源代码有多么实用，请想象你现在用的不是 Perl，而是某种叫 Forehead 的语言，授权出售它的公司很大，老板则是某位发型难看的亿万富翁（以上纯属虚构，大家都知道没有 Forehead 这种语言）。如果发现了 Forehead 里的 bug，你能做什么呢？首先，你可以汇报问题。然后，你可以怀抱希望 —— 希望他们会修改这个问题、希望它能赶快被修好、希望下个版本不会太贵。你还可以希望下个版本不会加上新的功能和 bug，并且希望该大公司不会因为犯了反托拉斯法而被拆成两半。

但是，Perl 的源代码则是掌握在你手上。在极罕见的情况下，就算某个缺陷没有其他办法可以修改，你也可以雇用某个程序员或开发团队来帮你解决。如果你买了一台新款的机器，Perl 还不能在上面运行，你也可以自己进行移植。如果想要某个新功能，你也知道该怎么办。

还有别的支持方式吗？

当然有！我们最喜欢的方式之一就是 Perl 推广组（Perl Mongers），它是全世界的 Perl 用户集会的组织。相关信息请参考 *http://www.pm.org/*。你的附近应该就有个分部，可以找到专家或认识专家的人。如果附近没有分部，你很容易就可以自己成立一个。

当然，请不要忽略了 Perl 的说明文档，它们提供了及时的支持。除了在线手册（manpage）【注 17】之外，CPAN 的网站 *http://www.cpan.org* 也有文档可看，*http://perldoc.perl.org* 上则有 HTML 与 PDF 格式的文档，在 *http://faq.perl.org/* 上则有最新版本的 Perl 常见问题集（perlfaq）。

注 17：术语 *manpages* 是 Unix 文化里对“说明文档”的讲法。如果你使用的系统不是 Unix，那么 Perl 的在线手册应该可以在系统原生的文档系统里找到。

另一个权威性的数据来源则是《Perl 语言编程》这本书，因为该书封面动物的关系，它通常被称为大骆驼书（就像本书被叫成小骆驼书一样）。大骆驼书里包含了完整的参考数据、一些教学范例以及许多与 Perl 相关的杂项信息。另外还有一本口袋大小的 Perl 5 Pocket Reference，作者是 Johan Vromans（O'Reilly 出版），很适合拿在手上阅读（或是放进口袋里）。

如果想找人问问题，Usenet 上有许多新闻组，也有为数众多的邮件列表【注 18】。无论何时，在某个时区都会有专家回答 Usenet 上 Perl 新闻组里的问题，Perl 是个日不落帝国，这表示在你提出问题的几分钟后通常就会有人提供解答。但如果你没有先查过说明文档和常见问题集，几分钟之内就会挨骂。

Perl 官方的 Usenet 新闻组位于 *comp.lang.perl.** 这一级。在编写此书时，一共有五个组，但也会随时间的改变而有所增减。你（或是在你的系统上负责管理 Perl 的人）通常要订阅 *comp.lang.perl.announce* 组，它是个低流量、仅仅只有重要公告信息的组，也包括了 Perl 在信息安全方面的公告信息。如果需要人指点 Usenet 的使用方式，请就近询问专家。

此外，也有一些从事 Perl 讨论的社群网站。其中很受欢迎的 Perl Monastery（*http://www.perlmonks.org*）上面有许多 Perl 书籍和专栏的作者，至少包括了本书其中两位作者。你也可以看看 *http://learn.perl.org/* 与相关的邮件列表 *beginners@perl.org*。如果要了解 Perl 的新闻，可以去 *http://use.perl.org/*。众多著名的 Perl 程序员还会建立自己的博客，发表一些有关 Perl 的文章，大部分都可以在这里看到 *http://planet.perl.org*。

如果你需要签一份 Perl 的支持服务合约，有好几家公司会愿意收你的钱。不过通常情况下，使用那些免费的支持资源就已足够。

如果发现 Perl 有 bug，我该怎么办？

当你发现 Perl 有 bug，请再次查看说明文档【注 19】。【注 20】Perl 里有许多特殊功能和异常，所以你遇到的或许是个功能而不是 bug。另外，请确定你当前安装的是最新版的 Perl，也许你碰到的问题在新的版本中已经修改好了。

注 18：在 *http://lists.perl.org* 可以找到邮件列表的清单。

注 19：即使是 Larry，也承认自己时常查阅说明文档。

注 20：有时还得反复查两三次。我们在文档查某个异常状况的线索时，经常会发现新的细节。它们最后都成了讲座或是专栏文章里面的花絮。

如果你 99% 地确定自己真正找到了 bug，请问问周围的人。你可以问同事、在当地的 Perl 推广组集会时发问或是在 Perl 会议上提出。它很可能其实是个功能，而不是 bug。

一旦你 100% 地确定它是真正的 bug，请写一个试验程序（如果还没做的话）。理想的试验程序是一个自给自足的小程序，任何 Perl 用户在执行它时都可以看到你发现的错误状况。在你作出能够清楚显示问题的测试程序后，请用 *perlbug* 这个工具程序（内置于 Perl 的发行版本）来报告这个 bug。它会自动寄出一份电子邮件给 Perl 的开发团队，所以请在完成测试程序的开发之后，再使用 *perlbug* 提交。

不出意外的话，通常送出 bug 报告后几分钟之内，你就会收到反馈信息。你也可以在回头着手做正事之前，先打上自己写的简单补丁（patch）。不过，在最坏的情况下也许不会有人响应，Perl 的开发团队对你并没有义务，甚至不一定非读你的 bug 报告不可。但因为我们全都是 Perl 的爱好者，因此不会容忍这样的 bug 逍遥法外。

我该怎么编写 Perl 程序？

差不多是问这个问题的时候了（即使你还没问）。Perl 程序是一个纯文本文件，你可以用自己喜欢的文本编辑器来创建与编辑它。（Perl 不需要特殊的开发环境，虽然也有厂商提供这样的商业软件。我们使用这类软件的经验不足，因此无法推荐。）

一般来说，你应该使用程序员专用的编辑器，而不是普通的编辑软件。两者有什么不同呢？程序员专用的编辑器可以快速方便地执行写程序时常用的那些操作，例如调整一段代码的缩排或是查找定位成对的左右花括号。在 Unix 系统上最受欢迎的两套程序员专用编辑器是 *emacs* 和 *vi*（以及它们的衍生版本），BBEdit 和 TextMate 则是 Mac OS X 系统上不错的编辑器，在 Windows 上则有很多人推荐使用 UltraEdit 与 PFE（Programmer's Favorite Editor）。在 *perlfaq2* 的在线手册（manpage）中也列出了其他一些编辑器。如果你不知道自己的系统上有哪些文本编辑器可用，请就近找专家帮忙。

本书习题需要编写简单的程序，其长度都不超过二三十行，所以你用哪种文本编辑器都没问题。

有些初学者会尝试使用文字处理器来代替文本编辑器。我们不建议这种做法，因为这样最起码是不方便，有可能干脆行不通。不过我们不会阻止你。请让文字处理器以“纯文本”格式来保存文件，以免它的默认格式导致运行有问题。几乎所有的文字处理器都会提醒你，正在编辑的 Perl 程序里面有一大堆拼写错误并且过度使用分号。

在某些情况下，你可能需要在一台机器上写程序，再传送到另一台机器上运行。这个时候，请使用“文本”模式（text mode）或者“ASCII”模式（ASCII mode）来传送程

序，而不是“二进制”模式（binary mode）。这是因为在不同的机器上，文本的格式也不一样。如果不用文本模式传送，则运行的结果可能会有所不同——某些版本的 Perl 在检测到换行符不一致时甚至还会中断运行。

一个简单的程序

按历史悠久的惯例，任何根植于 Unix 文化的程序语言入门书都会以“Hello, world”这个程序作为开头。所以，下面就是它在 Perl 里头的写法：

```
#!/usr/bin/perl
print "Hello, world!\n";
```

假设你已经将上面两行输入文本编辑器里了（暂且别管程序各部分是什么意思，我们很快就会讨论到）。一般来说，程序可以用任何文件名保存。Perl 程序并不需要命名为任何特殊的文件名或扩展名，最好不要用扩展名比较好【注 21】。不过，某些 Unix 以外的系统上也许必须使用 *.plx*（代表 PerL eXecutable）之类的扩展名。进一步的信息，请参考你的系统中 Perl 发行版本的相关说明。

接下来，你可能还需要告诉系统，该文件是一个可执行的程序（也就是一个命令）。在不同的系统需要的操作方式也会有所不同，也许只需要将程序文件存储到某个地方就行了（多数时候在你当前的工作目录也行）。在 Unix 系统上，你可以使用 *chmod* 命令将程序文件的属性修改为可执行，如下所示：

```
$ chmod a+x my_program
```

其中，最前面的美元符号（以及空白）代表命令行提示符（shell prompt），在你的系统上多半会不太一样。如果你习惯用 `755` 之类的数值而不是 `a+x` 这样的符号来代表权限，那当然也没有问题。不管用哪种写法，它都能告诉系统，这个文件现在已经是一个程序了。

现在，你已经可以运行它了：

```
$ ./my_program
```

命令开头的点号与斜线，表示要在当前工作目录（current working directory）里查找这

注 21：为什么不带扩展名比较好？假设你写了一个计算保龄球分数的程序，然后告诉所有朋友它的文件名是 *bowling.plx*。之后某一天，你决定以 C 语言来改写它。要保留原来的文件名，继续让人以为它是用 Perl 写的吗？还是要跟大家说它换了个新名字呢？（拜托，可执行文件的名称请不要叫 *bowling.c*！）正确答案是：程序是用什么语言写的，跟用户一点关系也没有。所以，程序最初的文件名就应该叫 *bowling*。

个程序。你不一定每次都得使用它们，但是在完全了解它们的用处之前，每次执行命令的时候还是都加上去比较好【注 22】。如果一切正常，那可真是太棒了。通常，你会发现程序里有 bug。这时只要修改程序，再运行一次就好了。但是不必每次都使用 *chmod*，因为第一次设定之后文件的可执行属性不会发生变化。（当然，如果问题其实是 *chmod* 没设对，那么命令行里多半会出现像“无此权限”（permission denied）这样的错误信息。）

另外在 Perl 5.10 里还有种不同的写法，我们这就来看看。这次我们不用 `print` 而改用 `say` 命令，它的效果基本相同，但却不需要输入换行符，并且减少了键入次数。由于这是个新特性，而你可能还没安装 Perl 5.10，所以在这里我们使用了 `use 5.010` 语句，告诉 Perl 我们想要这个新特性，把它加载进来：

```
#!/usr/bin/perl

use 5.010;

say "Hello World!";
```

这个程序只能在 Perl 5.10 中运行，在本书后续章节中介绍到 Perl 5.10 的新特性时，我们都会明确告诉你这一点，并且使用 `use 5.010` 语句作为提醒。因为 Perl 总是将版本号看作是三位数表示的，所以前导零千万不要省略，记住要写成 `use 5.010` 而不是 `use 5.10`（Perl 会把它看作是 `5.100`，一个目前我们还没发布的版本！）。

程序里写的是什么？

就像其他“形式自由”（free-form）的语言一样，Perl 通常可以随意加上空白（像空格、制表符与换行符等）使程序代码更易阅读。不过多数 Perl 程序的格式都比较一致，就和本书使用的格式相似。我们强烈建议并鼓励使用适当的缩排，因为缩排能有效地让程序变得好读；而好的文本编辑器也会帮你处理此事。此外，好的注释也能够帮助我们轻松阅读，并理解程序要做的事情。Perl 里的注释是从井号（#）开始，到行尾结束的部分。（Perl 里没有“注释块”【注 23】）。我们不会在本书展示的程序中使用很多注释，因为代码前后已经有了详细的解释。但是在你的程序里，请尽量在需要时加上注释。

注 22：简而言之，这是为了防止 shell 执行同名的其他程序（或 shell 的内置命令）。初学者常犯的一个错误，就是把第一个程序取名为 `test`。很多系统已经有 `test` 这个程序（或 shell 的内置命令）了，如此一来，运行的就不是他们的程序，而是系统自带的程序或命令。

注 23：不过，有许多模拟的方法，请参考常见问题集（FAQ）的说明。在大部分的系统里，键入 *perldoc perlfaq* 就可以了。

于是，格式混乱的程序，还是拿“Hello, world”来作例子，也可以写成这样（我们不得不说，这是相当奇怪的写法）：

```
#!/usr/bin/perl
    print    # This is a comment
"Hello, world!\n"
  ;    # Don't write your Perl code like this!
```

第一行其实是个具有特殊意义的注释。在 Unix 系统里【注 24】，如果文本文件开头的最前两个字符是 #!，那么后面跟着的就是用来执行这个文件的程序路径。在上例中，该程序就是 */usr/bin/perl*。

事实上，Perl 程序里最缺乏可移植性的就是 #! 那行了，因为你必须确定在每台机器上 perl 是放在什么路径下的。幸好大多数情况都不外乎 */usr/bin/perl* 和 */usr/local/bin/perl* 这两种。如果不是，你就得找出系统上的 perl 解释器程序究竟是藏在哪个路径下，然后改用此路径。在 Unix 系统上，还可以写成这种样子，让 env 命令自动帮你定位 perl 的路径：

```
#!/usr/bin/env perl
```

如果在所有可查找的目录下都找不到 perl 解释器，就近请教一下系统管理员或是使用同样系统的朋友吧。

在非 Unix 的系统中，传统上第一行会写成 `#!perl`（其实这也是有用的）。至少，它能让维护人员在着手修正程序时，马上知道这是个 Perl 程序。

要是 #! 行写错了，shell 通常会给出错误信息。它的内容可能会出乎意料，像是“File not found”（找不到文件）。这并不是说 shell 找不到你要运行的程序，而是说找不到 */usr/bin/perl*。我们也很想把这个报错信息改得更清楚明确些，但它不是 Perl 发出的，发出抱怨的是 shell。（顺便提一下，指定路径时不要将 *usr* 错拼成 *user* —— Unix 的发明者也真是懒惰到家了，他们总是习惯于省略许多字母。）

另一个你可能会碰上的问题，就是你的系统完全不支持 #! 行。这样一来，你的 shell（或者其他不管你的系统用的是什么类似 shell 的东西）也许会直接尝试运行你的程序，出来的结果大概会令人失望或是出人意料。要是你搞不清某些奇怪的错误信息在讲什么，可以用 *perldiag* 在线手册查找。

注 24：无论如何，比较近代的系统都有此功能。使用井号加惊叹号的“sh-bang”机制【译注 1】，是在 20 世纪 80 年代中期出现的。对 Unix 这样历史悠久的系统而言，这已经算是相当早期的功能了。

译注 1：发音为“shebang”，和在“the whole shebang”中一样。

所谓的“主”（main）程序完全由普通的 Perl 语句（statement）组成（子程序里的除外，这个我们稍后再谈）。这和 C 或 Java 语言不同，Perl 里头没有以“main”命名的例程（routine）。事实上，好多 Perl 程序可能根本不用子程序。

另外，Perl 程序并不需要变量声明的部分，这点和其他语言不同。如果过去你都一直习惯于声明所有变量，那现在可能会有点不安。不过，这种特性使得我们可以编写“急功近利”的 Perl 程序。如果程序的长度只有两行，却将其中一行耗费在声明变量上，似乎不太值得。如果真的想声明变量的话，那好极了，我们会在第四章里讨论怎么做。

大部分的语句都是表达式后面紧接着一个分号。下面的语句，我们已经看到过好几次了：

```
print "Hello, world!\n";
```

你大概已经猜到了，上面这行会输出 `Hello, world!`。接在这两个词后面的是转义写法的 `\n`，其含义对于熟悉 C、C++ 和 Java 的用户来说应该不陌生：它就是换行符（newline character）。当它跟在某些字符串后面打印出来时，光标位置会从行末移到下一行的开头，这样结束运行后，shell 的提示符就可以在新的一行开始，而不必跟在程序输出的信息后面，所以在写这样的程序时，最好在每一行输出的信息后面都加上换行符作为结尾。下一章，我们将会看到更多关于换行符的转义写法以及其他所谓“反斜线转义”（backslash escape）的介绍。

但是，应该如何编译 Perl 程序呢？

只需要直接运行它就可以了。只此一步，Perl 解释器就能完成编译和运行这两个动作：

```
$ perl my_program
```

运行程序时，Perl 内部的编译器会先载入整个源程序，将之转换成内部使用的 *bytecode*，这是一种 Perl 在内部用来表示程序语法树的数据结构。然后交给 Perl 的 bytecode 引擎执行。所以，如果在第 200 行有个语法错误，那么在开始运行第二行【注 25】程序之前，Perl 就会报告这个错误。如果你要在程序中做一个运行 5000 次的循环，它只会被编译一次，然后每次循环都以最快的速度运行。除此之外，为了提高易读性，不论你用多少注释和空白，它们都不会影响运行时（runtime）的速度。你甚至可以使用完全由常量组成的算式，它的值只会在程序开始时计算一次，哪怕它是在某个循环中也只计算一次，然后在后续的执行中重用计算结果。

注 25：除非第二行恰好是编译时（compile-time）的操作，例如 `BEGIN` 块或 `use` 调用时执行的代码。

不用说，编译是要花时间的 —— 所以如果只是为了要迅捷地完成某个简短任务，而去运行一个冗长的包含其他各种任务代码的 Perl 程序，会显得效率低下，因为花在编译上的时间可能会比运行时间还要长。不过 Perl 编译器的运行速度非常快，通常编译时间只占运行时间的极少部分而已。

不过，有个例外的状况，如果你编写的是 CGI 脚本，它有可能每分钟会被 Web 服务器调用运行成百上千次（这样的使用频率非常高。如果像其他的 Web 程序一样，只是每天运行成百上千次的话，就没什么好担心的了）。大多数这类程序的运行时间都很短，所以重复编译造成的时间消耗问题可能会比较明显。如果你碰上这种情况，就需要想办法将程序代码编译后让它驻留到内存中，好让后续的调用跳过编译，直接运行。Apache Web 服务器（*http://perl.apache.org*）的 `mod_perl` 扩展模块或是像 `CGI::Fast` 之类的 Perl 模块都可以帮你解决这类问题。

要是把编译后的 bytecode 存储起来，能否节省编译时间？或者更进一步地说，能否将 bytecode 转换成另一种语言，比如 C 语言，然后再进行编译？好吧，其实以上两种做法在某些情况下都是可行的，不过老实说这么做并没什么好处，程序不会因此变得更易使用、维护、调试或安装，甚至（由于某些技术性因素）还会让程序运行得更慢。Perl 6 在这方面应该会好很多，不过在眼下（编写本书时）就下结论似乎有点言之过早。

走马观花

读到这里，你一定想看看可以派实际用场的 Perl 程序到底是什么样子的（要是你其实不想，麻烦暂时合作一下）。请看：

```
#!/usr/bin/perl
@lines = `perldoc -u -f atan2`;
foreach (@lines) {
  s/\w<([^>]+)>/\U$1/g;
  print;
}
```

现在，如果这是你第一次看到这样的 Perl 代码的话，可能会觉得有点怪异（事实上，每次读到这样的 Perl 代码，都会让你觉着挺奇怪的），不过，我们会逐行讲解这个例子程序，看看它到底做了些什么（下面的介绍会非常简略，毕竟这一节只是“走马观花”。程序里用到的所有功能在后面的章节中都会详加说明，你不需要现在就把程序彻底搞懂）。

第一行是我们介绍过的 `#!` 行。在你自己的系统上，可能需要修改一下，确认使用的是正确的路径，具体请参考前文所述。

第二行运行了一个外部命令，使用一对反引号（`` ` ``）来调用（反引号的按键在全尺寸的

美式键盘上，通常会放在数字键 1 的左边。请小心，别把反引号和单引号“'”搞混了）。我们要运行的外部命令是 *perldoc -u -f atan2*，。请在命令行上键入这条命令，看看它会输出什么信息。*perldoc* 命令在大部分系统上都有，可以用它来阅读 Perl 及其相关扩展和工具程序的说明文档【注 26】。执行这条命令，会显示在 Perl 中如何使用三角函数 `atan2` 的一些信息。不过这里我们只是用它来示范如何处理外部命令的输出结果。

当反引号里的命令执行完毕后，输出结果会一行行依次存储在 `@lines` 这个数组变量里。接下来一行程序代码会启动一个循环，依次对每行数据进行处理。循环里的代码是缩排过的，虽然 Perl 并不强迫你这么做，但是好的程序员都会如此要求自己。

循环里的第一行代码看起来最恐怖：`s/\w<([^>]+)>/\U$1/g;`。此处不会深入讨论细节，它的大概意思就是：对每个包含一对尖括号（< >）的行，进行相应的数据替换操作。而在 *perldoc* 命令的输出结果里，应该至少有一行符合此操作条件。

至于循环内的第二行代码，来了个大变样，一下简洁不少，它直接输出每行的内容（有可能被上面的替换操作修改过）。最后的输出结果看起来应该和 *perldoc -u -f atan2* 的执行结果差不多，只是其中出现尖括号的地方有所不同。

就这样，在短短数行代码中，我们调用了别的程序，并将它的输出结果存到内存，然后更新内存里的数据，最后输出。很多时候，我们都是用 Perl 来做这种数据转换的工作的。

习题

一般来说，每章结束时都会有几道习题，解答在附录 A 中可以找到。不过在本章的习题中就不用写程序了 —— 答案其实已经在前面出现过了。

如果在你的机器上不能做这些习题，请先自己详细检查几遍，然后再就近询问专家。别忘了有时候我们需要修改一下程序，弱化一部分功能再运行，就像我们在文中提到的那样。

1. [7] 键入前面的“Hello, world”程序，想办法让它运行起来！（程序文件可以取任何名称，比如说 `ex1-1` 这样简洁的名称就不错，可以表示第一章的第 1 道习题。）

注 26：如果没有 *perldoc* 命令可用，很可能意味着你的系统没有命令行界面，并且你的 Perl 也无法通过反引号运行外部命令（像是 *perldoc*）或是打开一个管道传送数据给外部程序，有关管道方面的知识可参考第十六章。碰上这种情况，现在只需要简单的跳过调用 *perldoc* 的这行程序。

2. [5] 在命令行提示符后键入 *perldoc -u -f atan2* 这条命令，并观察它的输出结果。如果无法执行，请就近询问系统管理员或参考 Perl 发行版本的说明，找出如何调用 *perldoc* 或是相近的其他命令（下道题目也会用到这个命令）。

3. [6] 键入第二个例子程序（参阅上一节），看看它会输出什么。（提醒一下：请仔细按照书上的标点符号键入，不要打错了！）你能看出来它是如何改变 *perldoc* 命令的输出结果的吗？

第二章

标量数据

英语跟许多其他语言一样区别单数（singular）和复数（plural）。作为一个由人类语言学家设计的计算机语言，Perl 也有类似的区别。一般来说，Perl 用标量（*scalar*）【注 1】来称呼单件事物。标量是 Perl 里面最简单的一种数据类型。对大部分的标量来说，要么是数字（比如 255 或是 3.25e20），要么是由字符组成的串（比如 `hello` 【注 2】或林肯总统的 Gettysburg演讲词）。虽然你可能会认为，数字和字符串是两码事，但是在 Perl 里，这两者大多数情况下是可以在内部转换的。

你可以使用操作符（比如加法或字符串连接）对标量进行操作，产生的结果通常也是一个标量。标量可以存储在标量变量里，也可以从文件和设备读取（或写入这些位置）。

数字

虽然标量可以是数字或字符串，但此刻我们先把这两者分开来说。我们会先讨论数字类型的标量，然后再来讨论字符串类型的标量。

注 1：这和数学或物理学上同名的“标量”没什么关系，Perl 语言里是没有“向量”（vector）的。

注 2：如果用过其他的程序语言，你可能会认为 hello 是五个字符的集合，而不是单一事物。但是在 Perl 里，字符串就是单一的标量值。当然，有必要时，我们还是可以访问单独的字符，做法会在后面的章节里提到。

所有数字的内部格式都相同

接下来几段里，我们会看到整数（比如 255 或 2001）和浮点数（比如 3.14159 或 1.35×10^{25} 等）。但在 Perl 内部，则总是按“双精度浮点数”的要求来保存并进行运算【注 3】。也就是说，Perl 内部并不存在整数值 —— 程序中用到的整型常量会被转换成等效的浮点数值【注4】。当这类转换发生时，你大概无法注意到（也许根本不想注意），不过如果你想找找看有没有专门的整型运算（相对于浮点 运算），请马上停手，因为根本没有这种东西【注 5】。

浮点数直接量

直接量（literal）是某个数值在 Perl 代码中的书写方式。直接量并非运算的结果，也不是 I/O（输入／输出）操作的结果，而是直接写在程序源代码里的数据。

看到 Perl 浮点数的直接量的写法时，你应该不会感到陌生。小数点与前置的正负号都是可选的，数字后面也可以加上用“e”表示的 10 的次方标识符（指数表示法）。举例来说：

```
1.25
255.000
255.0
7.25e45  # 7.25 乘以 10 的 45 次方（一个很大的数）
-6.5e24  # 负 6.5 乘以 10 的 24 次方
         # （一个非常大的负数）
-12e-24  # 负 12 乘以 10 的 -24 次方
         # （一个非常小的负数）
-1.2E-23 # 另一种表示法：字母 e 也可以是大写的
```

注 3：此处所谓的双精度浮点数（double-precision floating-point value），就是当初用来编译 Perl 的 C 编译器的 `double` 类型。虽然它的大小可能因机器而异，但是大部分现代的系统都会使用 IEEE-745 的格式，它能够表示 15 位的精确度，有效值的范围从 `1e-100` 到 `1e100`。

注 4：其实 Perl 内部有些时候也会使用整数，只是不在程序员的控制范围中。这也就是说，唯一能够觉察的时候，通常就是程序运行更快的时候。又有谁会抱怨这种好事呢？

注 5：好吧，是有 `integer` 这个编译命令（pragma）没错，但是它已经超出了本书的讨论范围。当然，某些运算会将浮点数强制转换成整数，我们稍后会谈到。不过这跟我们前面讲的不是一回事。

整数直接量

整数直接量也相当直观易懂，例如：

```
0
2001
-40
255
61298040283768
```

最后一个数字读起来很费力。Perl 允许你在整数直接量里插入下划线，将若干位数分开，写成这样看起来就很清楚了：

```
61_298_040_283_768
```

这两种写法都表示同一个数字，只是看起来有些不一样而已。你也许会觉得应该用逗号分隔才对，但是逗号在 Perl 里已经有更重要的用途了（我们会在下一章看到），所以为了避免产生歧义，这里只能用下划线。

非十进制的整数直接量

像许多其他程序语言一样，Perl 也允许使用十进制（decimal）以外的其他进制来表示数字。八进制（octal）直接量以 0 开头，十六进制（hexadecimal）直接量以 0x 开头，而二进制（binary）直接量则以 0b 开头【注 6】。十六进制数的 A 到 F（或是小写的 a 到 f 也行），代表十进制数的 10 到 15 的数字。举例来说：

```
0377        # 八进制的 377，等于十进制的 255
0xFF        # 十六进制的 FF，也等于十进制的 255
0b11111111 # 也等于十进制的 255
```

虽然这三个数字看起来并不相同，但对 Perl 而言它们都是同一个数字。既然 0xFF 和 255.000 在 Perl 里没有分别，那么采用哪种写法最能说明问题；我们就该用哪种，这样对你自己和维护程序员都有好处（其实看不懂代码逻辑的可怜虫很多，但这个可怜虫往往就是你自己：在三个月之后，你可能完全记不起当时为什么要这么写）。

当非十进制直接量的长度超过四个字符时，读起来可能会有些困难。因此 Perl 同样容许在这些直接量中使用下划线，分开后读起来更容易些：

注 6：“前置零”（leading zero）的表示法，只对直接量有效，但不能在字符串自动转换成数字时使用，我们稍后会看到。你可以使用 oct() 和 hex()，将看来像八进制或十六进制的直接量转换成数字。虽然没有用来转换二进制直接量的 bin 函数，但是 oct() 可以接受以 0b 开头的字符串。

```
0x1377_0B77
0x50_65_72_7C
```

数字操作符

Perl 提供了一般的加法（+）、减法（-）、乘法（*）及除法（/）等操作符（operator）。举例来说：

```
2 + 3       # 2 加 3，得 5
5.1 - 2.4   # 5.1 减 2.4，得 2.7
3 * 12      # 3 乘 12，得 36
14 / 2      # 14 除以 2，得 7
10.2 / 0.3 # 10.2 除以 0.3，得 34
10 / 3      # 总是按照浮点型数据进行除法运算，所以得 3.3333333...
```

Perl 还支持取模（*modulus*）操作符（%）。表达式 `10 % 3` 的结果是 1，也就是 10 除以 3 的余数。取模操作符先取整然后再求余，所以 `10.5 % 3.2` 和 `10 % 3` 的计算结果是相同的。【注 7】另外，Perl 也提供类似 FORTRAN 语言的乘幂（*exponentiation*）操作符，满足了许多 Pascal 和 C 用户的心愿。乘幂操作符以双星号表示，比如 `2**3` 代表 2 的 3 次方，得 8【注 8】。此外还有其他一些数字操作符，我们会在以后用到的时候逐一介绍。

字符串

字符串就是一连串的字符（如 `hello`）。字符串可以由各种字符任意组合而成【注 9】。最短的字符串是不含任何字符的空字符串。最长的字符串的长度没有限制，它甚至可以填满所有内存（与此同时你也无法再对它进行任何操作）。这符合 Perl 尽可能遵循的“无内置限制”（no built-in limits）的原则。字符串通常是由可输出的字母、数字及标点符号组成，其范围介于 ASCII 编码的 32 到 126 之间。不过，因为字符串能够包含任何字符，所以可用它来对二进制数据进行创建、扫描与操作，这是许多其他工具语言望尘莫

注 7：要注意的是，进行模运算时，如果其中一边或两边都是负数，则不同的 Perl 版本可能会得出不同的结果。

注 8：一般来说，你不能计算负数的负数次方。数学怪人们（math geeks）都知道，这样算出来的结果将会是复数（`complex number`）。如果要使用复数的话，必须借助 `Math::Complex` 模块。

注 9：和 C 或 C++ 语言不同，空字符（NUL）在 Perl 里并没有特殊意义。Perl 会另行记住字符串的长度，而不是用空字符来表示字符串的结尾。

及的。举例来说，你可以将一个图形文件或编译过的可执行文件读进 Perl 的字符串变量里，改变它的内容后，再写回去。

和数字一样，字符串也有直接量，也就是 Perl 程序中的字符串书写方式。字符串直接量有两种不同的形式：单引号内的字符串与双引号内的字符串。

单引号内的字符串直接量

单引号内的字符串直接量（*single-quoted string literal*）指的是一对单引号圈引的一串字符。前后两个单引号并不属于字符串的一部分，它们只是用来让 Perl 识别字符串的开头与结尾。除了单引号和反斜线字符外，单引号内所有字符都代表它们自己（包括换行字符，如果该字符串表示多行的数据的话）。要表示反斜线字符本身，需要在这个反斜线字符前再加一个反斜线字符；要表示单引号本身时，同样在单引号前加一个反斜线字符。来看看这些情况：

```
'fred'     # 总共 4 个字符：f、r、e 和 d
'barney'   # 总共 6 个字符
''         # 空字符串（没有字符）
'Don\'t let an apostrophe end this string prematurely!'
'the last character of this string is a backslash: \\'
'hello\n' # hello 后面接着反斜线和 n
'hello
there'     # hello、换行符、there（总共 11 个字符）
'\'\\'     # 单引号，紧接着反斜线
```

要注意的是，单引号内的 \n 并不代表换行字符，而是表示反斜线和 n 这两个字符。只有在后面是另一个反斜线或单引号时，前面的反斜线才有特殊的意义。

双引号内的字符串直接量

双引号内的字符串直接量（*double-quoted string literal*）跟其他语言里的字符串差不多。和单引号内的字符串一样，它也是一连串的字符，不过是被一对双引号圈引的。但在这里，反斜线字符拥有更强大、更完整的能力，可用来表示控制字符，或是用八进制或十六进制写法来表示任何字符。这里是一些双引号内的字符串：

```
"barney"         # 和 'barney' 的写法一样的效果
"hello world\n" # hello world，后面接着换行符
"The last character of this string is a quote mark: \""
"coke\tsprite"  # coke，制表符（tab）和 sprite
```

请注意，对 Perl 来说双引号内的字符串直接量 "barney" 和单引号内的字符串直接量 'barney' 是相同的，都是代表那六个字符组成的串。这和前面提到的数字直接量的情

况相类似，0377 只不过是 255.0 的另一种写法。Perl 让你用自己觉得最合理的方式来书写直接量。当然，如果你想使用反斜线转义字符（backslash escape），像用 \n 来表示换行字符那样，就得用双引号。

可以通过反斜线加上各种不同的字符，以代表各种不同格式的数据，这种方式通常被称为反斜线转义。这里有张较为完整【注 10】的清单：表 2-1 列出各种双引号内字符串转义字符的用法和代表的意义。

表 2-1：双引号内字符串的反斜线转义

组合	意义
\n	换行
\r	回车
\t	水平制表符
\f	换页符
\b	退格
\a	系统响铃
\e	Esc（ASCII 编码的转义字符）
\007	任何八进制的 ASCII 值（此例中 007 表示系统响铃）
\x7f	任何十六进制的 ASCII 值（此例中 7f 表示删除键的控制代码）
\cC	控制符，也就是 Control 键的代码（此例表示同时按下 Ctrl 键和 C 键的返回码）
\\	反斜线
\"	双引号
\l	将下个字符转为小写
\L	将到 \E 为止的所有字符转为小写
\u	将下个字符转为大写
\U	将到 \E 为止的所有字符转为大写
\Q	将到 \E 为止的非单词（non-word）字符加上反斜线
\E	结束 \L，\U 或 \Q

双引号内字符串的另一种特性称为变量内插（*variable interpolated*），这是指在使用字符串时，将字符串内的变量名称替换成该变量当前的值。因为我们还没有正式介绍什么是变量，所以这部分稍后再谈。

注 10： Perl 最近的版本增加了“Unicode 转义字符”，但是这里不会讨论。

字符串操作符

字符串可以用 . 操作符（没错，就是句点符号）拼接起来。两边的字符串都不会因此而被改写，就像 2+3 的运算不会改变 2 或 3 一样。此运算产生的字符串（自然比之前连接符两边的字符串都要长）可以继续用来运算或存储到变量里。举例来说：

```
"hello" . "world"        # 等同于 "helloworld"
"hello" . ' ' . "world" # 等同于 'hello world'
'hello world' . "\n"     # 等同于 "hello world\n"
```

要注意的是，连接运算必须显式地使用 . 操作符，不像其他某些语言，只需把两个字符串放在一起就行。

还有个比较特殊的字符串重复（*string repetition*）操作符，它是一个小写字母 x。此操作符会将其左边的操作数（也就要重复的字符串）与它本身重复连接，重复次数则由右边的操作数（某个数字）指定。举例来说：

```
"fred" x 3           # 得 "fredfredfred"
"barney" x (4+1)     # 得 "barney" x 5，亦即 "barneybarneybarneybarneybarney"
5 x 4                # 本质上就是 "5" x 4，所以得 "5555"
```

最后一个例子有必要详加说明。因为重复操作符的左操作数必须是字符串类型，所以数字 5 在进行重复操作前，先被转换成单字符的串 "5"（具体转换规则稍后会提到），然后这个新字符串被复制了四次，生成含有四个字符的新字符串 5555。注意，如果我们将操作数对调，亦即写成这样 4 x 5，结果会得到重复五次的字符 4，也就是 44444。这说明字符串重复操作并不满足交换（commutative）律。

复制次数（右操作数）在使用前会先取整（4.8 变成 4）。复制次数小于 1 时，会生成长度为零的空字符串。

数字与字符串之间的自动转换

通常 Perl 会根据需要，自动在数字和字符串之间进行类型转换。那么它究竟是如何知道需要数字还是字符串呢？这完全取决于操作符需要什么类型的操作数。如果操作符（像是 +）需要的是数字，Perl 就会将操作数视为数字；在操作符（像是 .）需要字符串时，Perl 便会将操作数视为字符串。因此，你不必担心数字和字符串间的差异，只管合理使用操作符，Perl 会自动完成剩下的工作。

对数字进行运算的操作符（比如乘法）如果遇到字符串类型的操作数，Perl 会自动将字

符串转换成等效的十进制浮点数进行运算【注 11】。因此 "12" * "3" 的结果会是 36。字符串中非数字的部分（以及前置的空白符号）会被略过，所以 "12fred34" * "3" 也会得出 36，而不会出现任何警告信息【注 12】。在最极端的情形下，完全不含数字的字符串会被转换成零。比如将 "fred" 当成数字来用，就属于这种情况。

同样的，需要字符串的操作符（比如字符串连接）意外得到数字时，该数字就会被转换为（输出效果）相同的字符串。举例来说，如果想将字符串 Z 与“5 乘以 7 的结果”相连接【注 13】，可以直接写成这样：

```
"Z" . 5 * 7 # 等同于 "Z" . 35，得 "Z35"
```

也就是说，大多时候你根本不必关心数字和字符串的区别，Perl 会完成转换数据的工作【注 14】。

Perl 内建警告信息

在发现程序有些不对劲的时候，你可以让 Perl 发出警告。要启用警告功能，请在命令行运行程序时使用 -w 选项：

```
$ perl -w my_program
```

或者，如果想要每次运行都启用警告功能，可使用 #! 行来指明：

```
#!/usr/bin/perl -w
```

即使在非 Unix 系统上也可以这么写，传统上我们一般都这么写，不过也许此时 Perl 的路径并不会起作用，所以也可以这么写：

```
#!perl -w
```

注 11：“前置零”（leading zero）的技巧只对直接量有效，不能用在字符串的自动转换上。请使用 hex() 或 oct() 来转换字符串。

注 12：除非你要求 Perl 显示警告信息。这件事我们马上就会提到。

注 13：我们很快就会提到优先级和括号。

注 14：如果担心效率的话，大可放心，Perl 通常会记得转换的结果，因此转换只需要进行一次就行了。

在 Perl 5.6 或者更高的版本中，你还可以通过启用 warnings 这个编译命令来打开警告功能（请小心使用，有些人还在用旧版的 Perl）【注 15】。

```
#!/usr/bin/perl
use warnings;
```

现在，当你把 '12fred34' 当数字使用时，Perl 就会发出警告：

```
Argument "12fred34" isn't numeric
```

当然，警告信息通常是给程序员看的，而不是最终用户。要是连程序员都不看的话，警告信息大概也没什么用。另外，警告信息并不会改变程序的行为，只会让它偶尔抱怨几声。如果看不懂某个警告信息，可以利用 diagnostics 这个编译命令看更详细的问题描述。*perldiag* 在线手册里列出了所有的简要警告信息和详细诊断说明。

```
#!/usr/bin/perl
use diagnostics;
```

在把 use diagnostics 这个编译命令加进程序之后，你可能会觉得程序在启动的时候或多或少有点慢。那是因为，你的程序正在忙着加载（到内存）警告和详细说明，那么稍后万一有错误或是警告发生，你就可以马上看到这些相关的错误信息。换个角度看，这能说明一种优化程序的方法：如果你已了解各种警告信息的意义，不必再阅读运行时警告的解释文档，那就去掉 use diagnostics 这个编译命令，这样程序的启动就会变快，内存的使用也随之变少。（如果你能够修改好程序，让它不再产生那些警告信息，那就太棒了。不过关闭错误信息输出则更方便。）

更进一步，这种优化也可以通过 Perl 的命令行选项 -M 来实现。与其每次修改程序代码，不如仅在需要时再加载 diagnostics 编译命令：

```
$ perl -Mdiagnostics ./my_program
Argument "12fred34" isn't numeric in addition (+) at ./my_program line 17 (#1)
    (W numeric) The indicated string was fed as an argument to
    an operator that expected a numeric value instead.  If you're
    fortunate the message will identify which operator was so unfortunate.
```

在后面的讲解中，我们会介绍那些可能触发 Perl 警告的危险代码。但请记住，在未来的 Perl 版本中，警告信息的内容或是触发方式都有可能变动。

注 15：warnings 编译命令还可以产生语意的警告。具体如何使用，请查阅 *perllexwarn* 在线手册。

标量变量

变量（*variable*）就是某个容器的名称，里面可以存储一个或多个值【注16】。变量的名称在程序运行期间保持不变，但是变量的值通常会不断改变。

你大概猜到了，标量变量存储的是标量值。标量变量的名称是以美元符号（$）开头，后面接着所谓的 Perl 标识符：一个字母或下划线为首，后面可以跟上多个字母、数字或下划线。你也可以把它想成由一个以上的字母、数字或下划线构成，但是开头不能是数字。另外，大写和小写的字母会被区分：$Fred 和 $fred 是两个不同的变量。另外，变量名称中的每一个字母、数字和下划线都有区分作用，所以：

```
$a_very_long_variable_that_ends_in_1
```

有别于：

```
$a_very_long_variable_that_ends_in_2
```

在 Perl 里，标量变量名之前总有 $ 符号【注17】。在 shell 脚本中，变量命名也要用 $，但赋值时则不能用 $。在 *awk* 或 C 里，则完全用不到 $。如果你需要使用不同的程序语言，那么难免会搅和在一起，这并不奇怪。（大部分 Perl 程序员会建议你不要再写 shell、*awk* 和 C 程序了，但那往往不切实际。）

慎选变量名称

一般来说，变量名称应该和变量的用途相关。举例来说，$r 不太能说明其用途，但是 $line_length 就很明确。如果某个变量在程序里只有两三行代码会用到，像 $n 这样的简短名称还没什么关系；但若是整个程序里都用到的变量，该取个比较有意义的名称。

另外，恰如其分地使用下划线可以让名称更为清晰易读，尤其是在维护程序员的母语和你不同的情况下。举例来说，$super_bowl 这个名称就比 $superbowl（超级杯）要好，因为后者也可以看成 $superb_owl（雄伟的猫头鹰）。像 $stopid 的意思到底是 $s_to_pid（转换什么东西为进程号？）、$stop_id（某种“停止”对象的代号？），或只是愚笨（stupid）的笔误？

注16：我们接下来会看到，一个标量变量只能存储一个值。但是像数组（array）和哈希（hash）之类的变量类型就可以存储多个值。

注17：在 Perl 的术语里面，这种符号叫做“印记”（sigil）。

在我们写 Perl 程序时，大部分变量名称都使用小写字母，本书的程序也一样。大写字母仅在某些特殊情况下使用。全大写（例如 $ARGV）通常用来表示某种特殊变量。如果变量名称有好几个字母，有的人走的是 $underscores_are_cool 的“下划线派”，有的人走的是 $giveMeInitialCaps 的“首字母大写派”，这都没什么，关键是要保持一致的风格。

当然，名称的好坏其实对 Perl 来说并无差别。你可以把程序中最重要的三个变量取名为 $OOOOOOOOO、$OOOOOOOO 和 $OOOOOOOOO，Perl 也不会觉得怎样——不过那样请别找我们维护你的程序！

标量的赋值

标量变量的操作中，最常见的就是赋值（*assignment*）运算，也就是将某个值存进变量里。Perl 的赋值操作符为等号（这和其他程序语言差不多），等号的左边是变量名称，右边为某个表达式，对表达式求值的结果作为赋予变量的值。举例来说：

```
$fred   = 17;          # 将 $fred 的值设为 17
$barney = 'hello';     # 将 $barney 的值设为 5 个字符组成的字符串 'hello'
$barney = $fred + 3;   # 将 $barney 设为 $fred 当前值加上 3 后的结果，即 20
$barney = $barney * 2; # $barney 现在被设成 $barney 当前值乘以 2 后的结果，即 40
```

请注意，最后一行中 $barney 变量出现了两次：第一次是取值（在等号右方的表达式中），第二次是赋值（在等号左方），表示要将右方表达式的运算结果存到该变量中。这样写不但合法、安全，而且还十分常见。事实上，正因为这种用法太常见了，我们还有种更简便的书写方式，请看下节。

双目赋值操作符

我们经常会用到类似 $fred = $fred + 5 这种形式（同样的变量出现在赋值操作符的两边）的表达式，于是 Perl（同 C 或 Java 语言一样）提供了更新变量内容的简写方式——也就是双目赋值操作符（*binary assignment operator*）。几乎所有用来求值的双目操作符都可以接上等号，成为相应的双目赋值操作符。举例来说，下面这两行是等效的：

```
$fred  = $fred + 5; # 不使用双目赋值操作符
$fred += 5;         # 使用双目赋值操作符
```

以下这两行也是等效的：

```
$barney  = $barney * 3;
$barney *= 3;
```

以上的例子里，双目赋值操作符都是以某种方式直接修改变量的值，而非对表达式求值后覆盖原变量值。

另一个常见的双目赋值操作符是由字符串连接操作符（.）改造而成的追加操作符（.=）：

```
$str  = $str . " "; # 在 $str 末尾追加一个空格字符
$str .= " ";        # 用追加操作符做同样的事
```

几乎所有的双目操作符都可以这么用。举例来说，乘幂操作符也能改成 **=，所以 $fred **= 3 的意思就是“将 $fred 里的值自乘 3 次（也就是取三次方），再存回 $fred”。

用 print 输出结果

一般我们都想要程序输出一些信息，否则，也许会有人以为它什么事都没做。print( ) 操作符就是用来做这件事的：它可以接受标量值作为参数，然后不经修饰地将它传送到标准输出（standard output）。除非你做了什么特别的设定，否则“标准输出”指的就是终端屏幕。举例来说：

```
print "hello world\n"; # 输出 hello world，后面接着换行符

print "The answer is ";
print 6 * 7;
print ".\n";
```

你也可以用 print 输出一系列用逗号隔开的值：

```
print "The answer is ", 6 * 7, ".\n";
```

这实际上就是一个列表（*list*），但我们现在还没提到列表，稍后再说明。

字符串中的标量变量内插

双引号圈引的字符串直接量能进行变量内插（*variable interpolation*）【注 18】。这就是说，字符串内的所有标量变量【注 19】名，都会被替换为该变量当前的值。举例来说：

```
$meal   = "brontosaurus steak";
$barney = "fred ate a $meal";     # $barney 现在是 "fred ate a brontosaurus steak"
$barney = 'fred ate a ' . $meal;  # 另一种等效的写法
```

注 18：这和数学或统计学上的“替换”（interpolation）无关。

注 19：以及某些其他的变量类型，我们稍后会再说明。

如上例最后一行所示，不用双引号也可以达成相同的效果，但用双引号写起来更简便，也更清晰。

如果标量变量从来没有被赋值过【注 20】，就会用空字符串来代替：

```
$barney = "fred ate a $meat"; # $barney 现在是 "fred ate a "
```

如果要打印的就是这个变量，则不必使用变量内插的方式：

```
print "$fred"; # 双引号是多余的
print $fred;   # 这样写比较好
```

在单个变量的两边加上引号也不算出错，不过这么写的话，别的程序员可能会在背后嘲笑你哦【注 21】。变量内插也称为双引号内插，因为它只会在双引号（而非单引号）里起作用。在 Perl 里还有其他一些字符串中内插的情况，稍后遇到时我们再详加说明。

如果要将美元符号本身放进双引号内的字符串，可以在它前面用反斜线转义，以避开它的特殊意义：

```
$fred = 'hello';
print "The name is \$fred.\n";    # 会输出 $ 符号
print 'The name is $fred' . "\n"; # 这样也行
```

进行内插时，Perl 会尽可能使用最长且合法的变量名称。要是你想在内插的值后面，紧接着输出字母、数字或下划线【注 22】，可能会碰上麻烦：当 Perl 检查变量名称时，它会违背你的本意，将后面的字符当作变量名称的一部分。解决办法很简单，和 shell 脚本一样，Perl 里面我们可以用一对花括号将变量名围起来，以避免歧义。要不然，可以先把字符串拆成两半，再利用连接操作符接起来：

```
$what = "brontosaurus steak";
$n = 3;
print "fred ate $n $whats.\n";          # 不是 steaks，而是 $whats 的值
print "fred ate $n ${what}s.\n";        # 现在是用 $what 了
print "fred ate $n $what" . "s.\n";     # 另一种写法，比较麻烦
print 'fred ate ' . $n . ' ' . $what . "s.\n"; # 写起来特别麻烦
```

注 20：事实上，这将会是特殊的未定义值 undef，我们稍后会详细说明。启用警告信息时，Perl 会抱怨在内插时遇到了未定义的值。

注 21：嗯，这种写法会让该值被当成字符串处理，而不是当成数字处理。在某些罕见的状况下，也许需要这种写法，但是在平时，这几乎是在浪费打字时间。

注 22：另外还有一些可能会出问题的字符。在标量变量名称后面，如果需要接上左方括号或左方括号，请在括号前面加上反斜线。要是变量名称后面接的是单引号或两个冒号，也需要用同样的方式处理。要不然，你也可以使用正文里提到的花括号表示法来代替。

操作符的优先级与结合性

在复杂的表达式里，哪个操作会先发生，取决于操作符的优先级。举例来说，在表达式 `2+3*4` 中，要先算加法还是乘法？如果先算加法，我们会得到 `5*4`，也就是 `20`；如果先算乘法（就像数学课里教的），就会得到 `2+12`，也就是 `14`。还好，Perl 的选择和一般的数学算法相同，是先算乘法。也就是说，乘法的优先级高于加法。

你可以用括号（即圆括号）来改变默认的优先级。任何放在括号里的运算都比括号外的优先级高（和数学课里学到的一样）。因此，如果真的想让加法比乘法先计算，可以写成 `(2+3)*4`，得 `20`；也可以加上多余的括号，像 `2+(3*4)`，以强调乘法比加法先计算的事实。

在加法和乘法的例子里，优先级相当清楚。但是当我们碰到像字符串连接与乘幂的情形时，问题就出现了。正确的解决之道不外乎参考标准的 Perl 操作符优先级表，如表 2-2 所示【注 23】。（虽然其中有些操作符我们还没谈到，也未必涵盖在本书的范围里，不过你还是可以到 *perlop* 的在线手册中查阅它们的具体定义。）

表 2-2：操作符的结合性与优先级（从高至低排序）

结合性	操作符
左	括号；给定参数的列表操作符
左	`->`
	`++ --`（自增；自减）
右	`**`
右	`\ ! ~ + -`（单目操作符）
左	`=~ !~`
左	`* / % x`
左	`+ - .`（双目操作符）
左	`<< >>`
	具名的单目操作符（`-X` 文件测试；`rand`）
	`< <= > >= lt le gt ge`（“不相等”操作符）
	`== != <=> eq ne cmp`（“相等”操作符）
左	`&`
左	`| ^`

注 23：给 C 程序员一个好消息：所有同时在 Perl 和 C 里出现的操作符，它们的优先级和结合性都是相同的。

表 2-2：操作符的结合性与优先级（从高至低排序）（续）

结合性	操作符
左	&&
左	||
	.. ...
右	?:（三目操作符）
右	= += -= .=（以及类似的赋值操作符）
左	, =>
	列表操作符（向右结合）
右	not
左	and
左	or xor

在这个表格里，任何操作符的优先级都高于列在它下方的所有操作符，并低于列在它上方的所有操作符。如果操作符间的优先级相同，则按照结合性的规则来判断。

当两个优先级相同的操作符抢着使用三个操作数时，优先级便交由结合性解决：

```
4 ** 3 ** 2 # 4 ** (3 ** 2)，得 4 ** 9（向右结合）
72 / 12 / 3 # (72 / 12) / 3，得 6/3，得 2（向左结合）
36 / 6 * 3  # (36 / 6) * 3，得 18
```

在第一个例子里，因为 ** 操作符是向右结合的，所以隐含的括号便放在右边；而 * 和 / 是向左结合的，因此隐含的括号便放在左边。

所以说，你应该把优先级表背下来吗？那才奇怪呢！事实上，在记不起顺序又懒得查表时，用括号就是了。毕竟，要是你在没有括号的情况下忘记顺序，维护程序员也会遇到相同的麻烦。所以，还是对他好一点吧：或许将来某一天，这个人就是你。

比较操作符

对数值进行比较时，Perl 的逻辑比较操作符类似于代数系统的：<、<=、==、>=、>、!=。这些操作符的返回值一律是真（*true*）或假（*false*），我们将会在下一节详细讨论这些返回值的含义。这些操作符在 Perl 中的写法和在其他语言中的写法可能有所不同。例如，“相等”用的是 == 而不是单个的 =，因为 = 在 Perl 里有别的用途。另外，“不相等”用的是 !=，因为 <> 在 Perl 里也有别的用途。然后，“大于或等于”用的是 >= 而非 =>，因为后者在 Perl 里也有别的用途。事实上，几乎每一种标点符号的组合，在 Perl 里都有其他的用处。所以呢，如果哪天你的灵感突然告竭，就让猫猫在键盘上走个几圈，再进行调试吧。

想要比较字符串时，Perl 有一系列的字符串比较操作符，看起来像是些奇怪的短语：lt、le、eq、ge、gt、ne。它们会逐一比对两个字符串里的字符，判定它们是否彼此相等或是哪一个排在前面。(在 ASCII 编码顺序里，大写字母排在小写字母的前面，请留意。)

所有的比较操作符（包含对数值以及对字符串的）都列在表 2-3 里。

表 2-3:数值与字符串的比较操作符

比较	数值	字符串
相等	==	eq
不等	!=	ne
小于	<	lt
大于	>	gt
小于或等于	<=	le
大于或等于	>=	ge

下面是一些用到比较操作符的表达式：

```
35 != 30 + 5          # 假
35 == 35.0            # 真
'35' eq '35.0'        # 假  (当成字符串来比较)
'fred' lt 'barney'    # 假
'fred' lt 'free'      # 真
'fred' eq "fred"      # 真
'fred' eq 'Fred'      # 假
' ' gt ''             # 真
```

if 控制结构

学会如何对两个值进行比较后，你可能会需要根据比较的结果决定下一步的流程。Perl 和所有同类型的程序语言一样，也具备 if 条件语句控制结构：

```
if ($name gt 'fred') {
  print "'$name' comes after 'fred' in sorted order.\n";
}
```

如果想在条件不符合时走另一条路，可以使用 else 关键字：

```
if ($name gt 'fred') {
  print "'$name' comes after 'fred' in sorted order.\n";
} else {
  print "'$name' does not come after 'fred'.\n";
  print "Maybe it's the same string, in fact.\n";
}
```

可选（执行）的程序块周围一定要加上表示块界限的花括号（这点和 C 语言不同，不管你学过 C 没有）。你最好和上面一样，将块里的程序代码向内缩排，这样程序读起来会方便许多。如果你采用的是程序员专用的编辑器（见第一章），它通常会自动帮你解决此事。

布尔值

其实，任何标量值都可以成为 if 控制结构里的判断条件。如果你将真或假值放进某个变量里，使用起来会很方便：

```
$is_bigger = $name gt 'fred';
if ($is_bigger) { ... }
```

Perl 是怎么决定某个值的真假呢？和其他语言不同，Perl 并没有专用的“布尔”(Boolean)数据类型，它是靠一些简单的规则来判断的：【注 24】

- 如果值为数字，0 为假；所有其他数字都为真。
- 如果值为字符串，空字符串（''）为假；所有其他字符串都为真。
- 如果不是数字也不是字符串，就先转换成数字或字符串再行判断。【注 25】

以上规则中隐藏了另一条规则。字符串 '0' 跟数字 0 是同一个标量值，所以 Perl 会将他们一视同仁。也就是说，字符串 '0' 是唯一被当成假的非空字符串。

要取得任何布尔值的相反值，可以使用！这个单目取反操作符。若它后面的操作数为真，就返回假；若后面的操作数为假，则返回真：

```
if (! $is_bigger) {
  # 如果 $is_bigger 不为真，则执行这一段代码
}
```

获取用户输入

直到现在，你大概已经相当好奇，该如何让 Perl 程序读取从键盘输入的值？最简单的方式，就是使用“行输入”操作符 <STDIN>【注 26】。

注 24：这并不是 Perl 内部的完整规则，但足以让你用来进行判断。

注 25：也就是说，undef（我们马上就会看到）表示假，而且所有的引用（在羊驼书里才有说明）都是真。

注 26：这其实是作用在 STDIN 文件句柄上的“行输入”操作符，但是在我们讨论到文件句柄之前（见第五章）无法多做解释。

只要在程序代码中能使用标量值的位置写上 <STDIN>，Perl 就会从标准输入（standard input）读进一行文字（直到换行符为止）。标准输入可以有多种意义，但除非做过什么特别的设定，否则它指的就是调用程序的用户（多半就是你）手边的键盘。如果 <STDIN> 里没有可读取的字符（通常都是这样的，除非你预先打了一整行的字符），Perl 程序就会停下来，等待你输入某些字符，直到换行符（即按下回车键）出现为止。【注 27】

<STDIN> 返回的字符串，一般在最后都会跟有一个换行符【注 28】。所以，你的程序代码可以这么写：

```
$line = <STDIN>;
if ($line eq "\n") {
  print "That was just a blank line!\n";
} else {
  print "That line of input was: $line";
}
```

但在实践中，很少需要保留最后面的换行符，所以这时候 chomp 操作符就派上用场了。

chomp 操作符

乍看之下，chomp 操作符的用途过于单一了：它只能用在一个变量上，而且该变量的内容必须为字符串。如果此字符串的结尾是换行符，chomp 能将它移除。这几乎就是它的全部功能。举例来说：

```
$text = "a line of text\n"; # 假设是从 <STDIN> 读进来的
chomp($text);               # 去除行末的换行符
```

其实它还是非常有用的，你以后写的每个程序几乎都少不了它。如刚才所见，处理字符串变量时，它是去除行末换行符的最佳方式。事实上，chomp 还有更容易的用法，仅需遵从这条简单规则：在 Perl 程序里，任何需要变量的地方都可以用赋值运算来代替。首先，Perl 执行赋值操作，然后按你想要的任意一种方式使用变量。所以，chomp 最常见的用法就像是这样：

注 27：精确地说，其实是你的系统在等待输入，而 Perl 在等待系统。虽然因系统及配置不同而带来差异，但你通常都可以在按下 Enter 键之前用退格（backspace）键来修改错字，因为这是系统在处理，而不必麻烦 Perl。如果你需要额外的输入控制功能，请从 CPAN 下载 Term::ReadLine 模块。

注 28：这里有个例外，那就是标准输入流在读入行期间突然中止了。可是，不以换行符结尾的文件，当然不能算是合适的文本文件！

```
chomp($text = <STDIN>); # 读入文字，略过最后的换行符

$text = <STDIN>;        # 做一样的事情……
chomp($text);           # ……但分成两步来做
```

乍看之下，采用合并方式的 chomp 写起来并不容易，也许还更复杂些！如果你把它想成两次操作（读取一行文字，再对它做 chomp），那么分开来写确实比较自然。但是，如果你将它看成是一次操作（只读取文字，不含换行符），那么合并的写法就更自然了。许多 Perl 程序员都倾向于使用这种写法，所以现在你也该开始习惯起来。

其实，chomp 本质上是函数（function）。作为一个函数，它就有自己的返回值。chomp 函数的返回值是实际移除的字符数。这个数字几乎没有用处：

```
$food = <STDIN>;
$betty = chomp $food; # 会得到返回值 1，不过我们早就知道了！
```

正如你所见，使用 chomp 时，可以加上括号，也可以不加。这是 Perl 的另一项惯例：除非去掉括号会改变表达式的意义，否则括号用不用都可以。

如果字符串后面有两个以上的换行符【注 29】，chomp 也仅仅删除一个。如果结尾处没有换行符，它什么也不做，直接返回零。

while 控制结构

Perl 和大部分用来实现算法的语言一样，也有好几种循环结构【注 30】。在 while 循环中，只要条件持续为真，就会不断执行块里的程序代码：

```
$count = 0;
while ($count < 10) {
  $count += 2;
  print "count is now $count\n"; # 依次打印值 2 4 6 8 10
}
```

这里的真假值与之前提到的 if 条件测试里的真假值定义相同。块外围的花括号也和 if 控制结构的一样必不可少。条件表达式在第一次执行块之前就会被求值，所以，如果它一开始就为假，里面的循环就会被直接略过。

注 29：如果我们逐行读取输入，这种情况并不会发生。不过，假如我们把输入分隔符（$/）设成换行符以外的值、使用 read 函数或是手动连接字符串，那就大有可能。

注 30：任何程序员总有一天会不小心写出无穷循环。要是你的程序运行个不停，通常可以用能够停掉系统上其他程序的方法来停掉它。通常按下 Control-C 就可以停止程序的运行。请查询系统的说明文档，以确定实际的做法。

undef 值

如果还没赋值就用到了某个标量变量，会有什么结果呢？答案是，不会发生什么大不了的事，也绝对不会让程序中止运行。在首次被赋值之前，变量的初始值就是特殊的 undef（未定义），它在 Perl 里的意思只不过是：这里空无一物——走开、走开。如果你想把这个“空无一物”当成数字使用，它就会假设这是 0；如果当成字符串使用，它就会假设这是空字符串。但是 undef 既不是数字也不是字符串，它完全是另一种类型的标量值。

既然 undef 作为数字时会被视为零，我们可以很容易地构造一个数值累加器，它在开始时是空的：

```
# 累加一些奇数
$n = 1;
while ($n < 10) {
  $sum += $n;
  $n += 2; # 准备下一个奇数
}
print "The total was $sum.\n";
```

在循环开始之前，$sum 的初始值是 undef，但这不妨碍程序的运行。当第一次执行循环时，$n 的值是 1，所以循环里的第一行会将 $sum 的值加上 1。这么做，如同将现值为 0 的变量加 1（因为我们把 undef 当成数字用），所以累加的值会变成 1。此后它已经被初始化了，所以就能按常规方式累加。

同样的道理，也可以做出一个字符串累加器，它在一开始时是空的：

```
$string .= "more text\n";
```

如果 $string 的初始值是 undef，它在这里就会被当成空字符串处理，变量的值被设为 "more text\n"。如果它里面已经有字符串，就会把新的文字追加在后面。

Perl 程序员常常根据需要，把新变量当作零或空字符串来用。

许多操作符在参数越界或不合理时，会返回 undef。除非有特别处理，否则就会返回零或空字符串。实际上这并不代表有什么大问题，而且有很多程序员就是靠这种特性来写程序的。但你该知道的是，在警告信息开启时，Perl 通常会对未定义值的危险用法发出警告，因为那可能就是程序里的缺陷。举例来说，复制某个 undef 的变量到另一个变量没有什么问题，但若要用 print 将它输出，就会引发警告信息。

defined 函数

“行输入”操作符 <STDIN> 有时候会返回 undef。在一般状况下，它会返回一行文字。但若没有更多输入，比如读到文件结尾（end-of-file），它就会返回 undef 来表示这个状况【注 31】。要判断某个字符串是 undef 而不是空字符串，可以使用 defined 函数。如果是 undef，该函数返回假，否则返回真：

```
$madonna = <STDIN>;
if ( defined($madonna) ) {
  print "The input was $madonna";
} else {
  print "No input available!\n";
}
```

如果想自己创建 undef 值，可以直接使用同名的 undef 操作符：

```
$madonna = undef; # 犹如完璧之身
```

习题

下列习题的解答，请参阅附录 A：

1. [5] 写一个程序，计算在半径为 12.5 时，圆的周长应该是多少。圆周长是半径的长度乘上 2π（大约是 2 乘以 3.141592654）。计算结果大约是 78.5。
2. [4] 修改上题的程序，让它提示用户键入半径的长度。当用户键入 12.5 时，出来的计算结果应该和上题相同。
3. [4] 修改上题的程序，当用户键入小于 0 的半径时，输出 0 而非负数。
4. [8] 写一个程序，提示用户键入两个数字（分两行键入），然后输出两者的乘积。
5. [8] 写一个程序，提示用户键入一个字符串及一个数字（分两行键入），然后以数字为重复次数，连续输出字符串（提示：使用 x 操作符）。在用户键入“fred”和“3”时，应该会输出 3 行“fred”；如果用户键入的是“fred”与“299792”，输出结果应该是一大堆。

注 31： 一般来说，键盘输入没有“文件结尾”可言；不过，此输入也可能是由文件重定向而来的。另外，用户也有可能按下系统转义的“文件结尾”按键序列。

第三章

列表与数组

如果说 Perl 的标量代表的是单数（singular），那么正如第二章开头所讲的，在 Perl 里代表复数（plural）的就是列表和数组。

列表（*list*）指的是标量的有序集合，而数组（*array*）则是存储列表的变量。在 Perl 里，这两个术语常常混用。不过更精确地说，列表指的是数据，而数组指的是变量。列表的值不一定要放在数组里，但每个数组变量都一定包含一个列表（即使该列表可能是空的）。图 3-1 所示的就是一个列表，无论它是否存储在某个数组中。

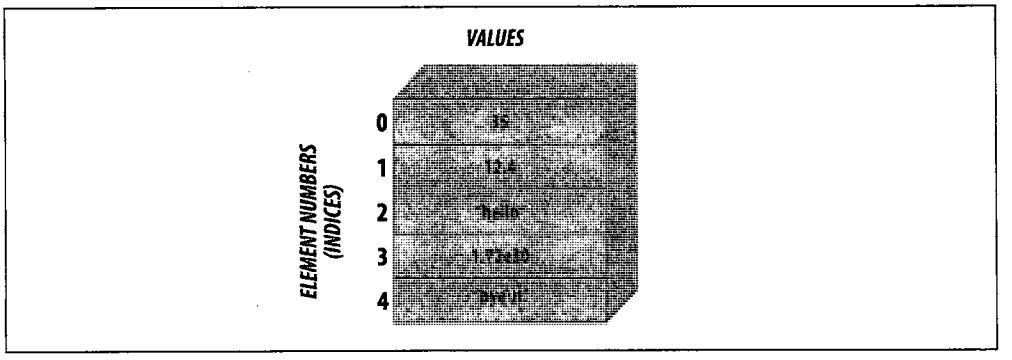

图 3-1：包含 5 个元素的列表

数组或列表中的每个元素（*element*）都是单独的标量变量，拥有独立的标量值。这些值是有序的，也就是说，从起始元素到终止元素的先后次序是固定的。数组或列表中的每

个元素都有相应的整数作为索引，此数字从0开始递增【注1】，每次加1。所以数组或列表的头一个元素总是第0个元素。

因为每个元素都是独立不相关的标量值，所以列表或数组可能包含数字、字符串、undef值或是不同类型标量值的混合。不过最常见的，还是让所有元素都具有相同的类型。像是由书本标题组成的列表（全都是字符串），或是由余弦函数值组成的列表（全都是数字）。

数组和列表可以包含任意多个元素。最少的情况是没有任何元素，最多的情况则是把可用的内存全部塞满。同样地，这遵循Perl“去除不必要限制”的原则。

访问数组中的元素

如果你曾在其他语言中使用过数组，那么当你看到Perl使用下标数字（subscript）来引用数组元素时，应该会觉得习以为常。

数组元素是以连续的整数来编号，从0开始，之后的每一个元素依次加1，如下所示：

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";
```

数组（此例中为"fred"）的名字空间（namespace）和标量的名字空间，是完全分开的。你可以在同一个程序里再取一个名为$fred的标量变量，Perl会将两者当成不同的东西，而不会搞混【注2】。（可是维护程序员也许会搞混，所以请不要随便将你的变量取相同的名称！）

凡是能够使用$fred这类标量变量的地方【注3】，也都可以使用像$fred[2]这样的数组元素。举例来说，你可以取出数组元素的值，或是用上章介绍的各种表达式改变它的值：

注1：跟其他的程序语言不同，数组和列表的索引在Perl里总是由0开始。早期的Perl版本允许你改变数组和列表索引值的起始编号（不是针对个别的数组或列表，而是一次全部生效！）。后来Larry了解到这是个错误的功能，所以目前非常不鼓励（滥）用它。不过，假如你的好奇心无药可救，倒是可以看看*perlvar*在线手册中对$[变量的说明。

注2：这种做法确实在语法上没有二义性，只是有炫耀技术的嫌疑。

注3：几乎是如此。最明显的例外是foreach循环的控制变量，它必须是简单的标量，我们会在本章稍后提到。还有其他的例外，像print、printf使用的“间接对象槽”（indirect object slot）及“间接文件句柄槽”（indirect filehandle slot）。

```
print $fred[0];
$fred[2]  = "diddley";
$fred[1] .= "whatsis";
```

当然，任何求值能得到数字的表达式都可以用作下标。假如它不是整数，则会自动舍去小数，无论正负：

```
$number = 2.71828;
print $fred[$number - 1]; # 结果和 print $fred[1] 相同
```

假如下标超出数组的尾端，则对应的值将会是 undef。这点和一般的标量相同，如果从来没有对标量变量进行过赋值，它的值就是 undef。

```
$blank = $fred[ 142_857 ];  # 未使用的数组元素，会得到 undef 的结果
$blanc = $mel;              # 未使用的标量 $mel，也会得到 undef 的结果
```

特殊的数组索引值

假如你对索引值超过数组尾端的元素进行赋值，数组将会根据需要自动扩大——只要有可用的内存分配给 Perl，数组的长度是没有上限的。如果在扩展过程中需要创建增补元素，那么它们的默认取值为 undef。

```
$rocks[0]  = 'bedrock';       # 一个元素……
$rocks[1]  = 'slate';         # 又一个……
$rocks[2]  = 'lava';          # 再来一个……
$rocks[3]  = 'crushed rock'; # 再来一个……
$rocks[99] = 'schist';        # 现在有 95 个 undef 元素
```

有时候，你会想要找出数组里最后一个元素的索引值。对我们正在使用的数组 rocks 而言，最后一个元素的索引值是 $#rocks【注 4】。但这个数字比数组元素的个数少 1，因为还有一个编号为 0 的元素：

```
$end = $#rocks;                    # 99，也就是最后一个元素的索引值
$number_of_rocks = $end + 1;       # 正确，但后面会看到更好的做法
$rocks[ $#rocks ] = 'hard rock'; # 最后一块石头
```

最后一个例子里，把 $#name 当成索引值的做法十分常见，所以 Larry 为我们提供了简写：从数组尾端往回计数的“负数数组索引值”。不过，超出数组大小的负数索引值是不会绕回来的。假如你在数组中有 3 个元素，则有效的负数索引值为 -1（最后一个元素）、-2（中间的元素）以及 -3（第一个元素）。实践中，似乎没有人会使用 -1 以外的负数索引值。

注 4：　这种丑陋的语法来自 C shell。还好，我们在实际应用中很少会看到这种写法。

```
$rocks[ -1 ]    = 'hard rock';    # 和上面最后一个例子相同，但更简单
$dead_rock      = $rocks[-100];   # 得到 'bedrock'
$rocks[ -200 ] = 'crystal';      # 严重错误!
```

列表直接量

程序中需要引入列表的时候可以写成数组的样子，也就是在圆括号中用逗号隔开的一系列值。这些值构成了列表中的元素。举例来说：

```
(1, 2, 3)       # 包含 1、2、3 这三个数字的列表
(1, 2, 3,)      # 相同的三个数字（末尾的逗号会被忽略）
("fred", 4.5)   # 两个元素，"fred" 和 4.5
( )             # 空列表：零个元素
(1..100)        # 100 个整数构成的列表
```

上例最后一行用到了 .. 范围操作符（*range operator*），这是我们第一次看到。该操作符会从左边的数字计数到右边，每次加 1，以产生一连串的数字。举例来说：

```
(1..5)             # 与 (1, 2, 3, 4, 5) 相同
(1.7..5.7)         # 同上：这两个数字都会被去掉小数部分
(5..1)             # 空列表：.. 仅向上计数
(0, 2..6, 10, 12) # 与 (0, 2, 3, 4, 5, 6, 10, 12) 相等
($m..$n)           # 范围由 $m 和 $n 当前的值来决定
(0..$#rocks)       # 上节的 rocks 数组里的所有索引数字
```

正如最后两行所示，数组中的元素不必都是常数——它们可以是表达式，在每次用到这些直接量时都会重新计算。比如：

```
($m, 17)        # 两个值：$m 的当前值，以及 17
($m+$o, $p+$q) # 两个值
```

当然，列表可以包含任何标量值，像下面这个典型的字符串列表：

```
("fred", "barney", "betty", "wilma", "dino")
```

qw 简写

在 Perl 程序里，经常需要建立简单的单词列表（如同前面的例子）。这时只需使用 qw 简写，就不必键入许多索然无味的引号：

```
qw( fred barney betty wilma dino ) # 同上，但更简洁，也更少键入
```

qw 表示 “quoted word”（加上引号的单词）或“quoted by whitespace”（用空白圈引），讲法因人而异。不管怎么说，Perl 都会将其当成单引号内的字符串来处理（所以，在 qw 构建的列表中，不能像双引号内的字符串一样使用 \n 或 $fred）。其中的空白

字符（如空格、制表符以及换行符）会被抛弃，然后剩下的就是列表的元素。因为空白字符会被抛弃，所以上面的列表也可以写成这样（虽不常见）：

```
qw(fred
  barney     betty
wilma dino)  # 同上，但空白字符的用法比较奇怪
```

因为 qw 算是一种引号，所以不能将注释放在 qw 列表中。

前面两个例子是以一对圆括号作为定界符（delimiter），其实 Perl 还允许你用任何标点符号作为定界符。常用的写法有：

```
qw! fred barney betty wilma dino !
qw/ fred barney betty wilma dino /
qw# fred barney betty wilma dino #    # 看起来像是注释!
qw( fred barney betty wilma dino )
qw{ fred barney betty wilma dino }
qw[ fred barney betty wilma dino ]
qw< fred barney betty wilma dino >
```

如同最后4行所示，前后两个定界符可能是不同的。如果起始定界符是某种“左”字符，则结尾的定界符就得是相应的“右”字符。如果用其他符号，起始和结尾的定界符则必须相同。

如果你需要在被圈引的字符串内使用定界符，那说明你可能是挑错了定界符。不过，在你无法或不希望更换定界符的情况下，还是可以通过反斜线转义来引入这个字符的：

```
qw! yahoo\! google ask msn ! # 将 yahoo! 作为一个元素包含进来
```

和单引号内的字符串一样，两个连续的反斜线表示一个实际的反斜线。

虽然 Perl 的座右铭是办法不止一种（There's More Than One Way To Do It），但你可能会纳闷，有谁会需要那么多不同的定界符呢？我们之后会看到，Perl 另外还有许多其他类似的圈引写法，用起来都非常称手。不过就现在来看，如果需要一连串的 Unix 文件名的列表时，这些定界符就很有用了：

```
qw{
  /usr/dict/words
  /home/rootbeer/.ispell_english
}
```

要是只能以斜线为定界符，这个列表将会变得相当臃肿难读，而且代码的编写和维护都很麻烦。

列表的赋值

就像标量值可被赋值给变量一样，列表值也可以赋值到变量：

```
($fred, $barney, $dino) = ("flintstone", "rubble", undef);
```

左侧列表中的三个变量会依次被赋予右侧列表中对应的值，相当于我们分别做了三次独立的赋值操作。因为列表是在赋值运算开始之前建立的，所以在 Perl 里互换两个变量的值相当容易【注 5】：

```
($fred, $barney) = ($barney, $fred); # 互换两者的值
($betty[0], $betty[1]) = ($betty[1], $betty[0]);
```

可是如果（在等号左边的）变量的个数不等于给定的列表值（来自等号右边）的个数时，会发生什么情况呢？对列表进行赋值时，多出来的值会被悄悄忽略掉——Perl 认为：如果你真的想要将这些值存放起来的话，你必然会先告知存储位置。另一种情况，如果变量的个数多过给定的列表值，那么那些多出来的变量将会被设成 undef【注 6】。

```
($fred, $barney) = qw< flintstone rubble slate granite >; # 忽略掉末尾两个元素
($wilma, $dino)  = qw[flintstone];                        # $dino 被设为 undef
```

明白了列表赋值，你便可以用如下代码来构建一个字符串数组【注 7】：

```
($rocks[0], $rocks[1], $rocks[2], $rocks[3]) = qw/talc mica feldspar quartz/;
```

不过，当你希望引用整个数组时，Perl 提供了一个比较简单的记法。只要在数组名之前加上 @ （at）字符（后面没有检索用的方括号）就可以了。你可以将它读作“all of the”（全部的，所有的），所以 @rocks 可以读作“所有的 rocks”【注 8】。这种写法在赋值操作符的两边都可以使用：

```
@rocks  = qw/ bedrock slate lava /;
@tiny   = ( );                      # 空列表
@giant  = 1..1e5;                   # 包含 100,000 个元素的列表
```

注 5：这和 C 之类的程序语言相反，它们通常没有简单的办法来做这种事。C 语言的程序员往往要靠某些宏来达成相同的结果，或者干脆用个变量来暂时存储这个值。

注 6：嗯，对标量变量来说是这样。数组变量则会变成空列表，稍后我们就会看到。

注 7：我们非常自信地假设 rocks 数组在此语句之前是空的。假如 $rocks[7] 有值，那么此次的赋值运算并不会影响到该元素。

注 8：Larry 宣称他之所以选择 $ 与 @ 这两个符号，是因为 $calar 看起来像 scalar（标量），而 @rray 则像 array（数组）。如果你看不惯，或是不想用这种方式帮助记忆，那也无所谓。

```
@stuff  = (@giant, undef, @giant);   # 包含 200,001 个元素的列表
$dino   = "granite";
@quarry = (@rocks, "crushed rock", @tiny, $dino);
```

最后一行进行的赋值运算会将 @quarry 设成拥有5个元素的列表(bedrock、slate、lava、crushed rock 和 granite)，因为 @tiny 贡献了零个元素给这个列表（请注意，由于空列表里没有任何元素，也就不会有 undef 被赋值到列表中——但是（如果需要 undef）我们也可以显式写明，就像之前对 @stuff 的操作那样)。此外，值得注意的是，数组名会被展开成（它所拥有的）元素列表。因为数组只能包含标量，不能包含其他数组，所以数组无法成为列表里的元素【注9】。被赋值之前，数组变量的值是空列表，即（ ）。就像标量变量的初始值是 undef 一样，新的或是空的数组的初始值是空列表。

要留意的是，将某个数组复制到另一个数组时，仍然算是列表的赋值运算，只不过这些列表是存储在数组里而已。举例来说：

```
@copy = @quarry; # 将一个数组中的列表复制到另一个数组
```

pop 和 push 操作符

要新增元素到数组尾端时，只要将它存放到更高的索引值对应的新位置就行了。不过，真正的 Perl 程序员是不太用索引的【注 10】。在随后几个小节中，我们将会介绍一些（不用索引）对数组进行操作的方法。

我们常把数组当成堆栈来用，这时的新增和删除操作都在列表的右手边（就是数组里位于“最后面”的那一端，也就是索引值最高的那一端）发生。因为这类操作经常发生，所以有自己专用的函数。

注 9： 不过在羊驼书里你会学到一种称为“引用”(reference) 的特殊标量。它能让我们做出“列表的列表”以及其他更有趣或有用的结构。但在那种情况下，也不是真的将列表存进另一个列表中，而是将引用存放到数组里。

注 10： 当然，我们是开玩笑的。但是这个玩笑里隐含着一个事实：Perl 并不擅长使用索引值来访问数组。如果使用 pop、push 或别的不需要索引值的操作方式，通常会比用了很多索引操作的程序更快，同时也能避免“边界差一”(off-by-one) 错误，这通常也称为“栅栏柱错误”(fencepost error)。时常会有 Perl 初学者想要比较 Perl 和 C 在速度上的差异，于是直接拿一个针对 C 语言优化过的排序算法（里面用到了很多索引操作）改写成 Perl（一样也用到了很多索引操作）来进行竞速比较后，很纳闷为何 Perl 会这么慢。答案就是，拿 Stradivari（斯特拉迪瓦里家族）制造的小提琴去敲铁钉，实在不应该被当成一种发音技法。

pop 操作符可用来取出数组中最后一个元素，同时返回该元素值：

```
@array  = 5..9;
$fred   = pop(@array);  # $fred 变成 9, @array 现在是 (5, 6, 7, 8)
$barney = pop @array;   # $barney 变成 8, @array 现在是 (5, 6, 7)
pop @array;             # @array 现在是 (5, 6), 7 被抛弃了
```

最后一行是在空上下文（void context）中使用 pop 操作符。所谓的“空上下文”只不过是好听的说法，表示返回值无处可去。这样使用 pop 操作符不是错误，因为这可能恰好是你需要的。

如果数组是空的，pop 什么也不做（因为没有任何元素可供移出），直接返回 undef。

你也许已经注意到了，pop 后面加不加括号都可以。这是 Perl 的惯例之一：只要不会因为拿掉括号而改变原意，括号就是可省略的【注 11】。与此对应的是 push 操作符，用于添加一个元素（或是一串元素）到数组的尾端：

```
push(@array, 0);       # @array 现在是 (5, 6, 0)
push @array, 8;        # @array 现在是 (5, 6, 0, 8)
push @array, 1..10;    # @array 得到了 10 个新元素
@others = qw/ 9 0 2 1 0 /;
push @array, @others; # @array 又得到了 5 个新元素 (共 19 个)
```

注意，push 的第一个参数，或是 pop 的那个唯一的参数，都必须是要操作的数组变量——对列表直接量进行压入（push）或弹出（pop）操作是不可能的。

shift 和 unshift 操作符

push 和 pop 操作符处理的是数组的尾端（或者说是数组的“右”边，最高下标值的部分，怎么理解都行）；相似地，unshift 和 shift 操作符则是对数组的“开头”（或者说是数组的“左”边，最低下标值的部分）进行相应的处理。来看几个例子：

```
@array = qw# dino fred barney #;
$m = shift(@array);       # $m 变成 "dino", @array 现在是 ("fred", "barney")
$n = shift @array;        # $n 变成 "fred", @array 现在是 ("barney")
shift @array;             # 现在 @array 变空了
$o = shift @array;        # $o 变成 undef, @array 还是空的
unshift(@array, 5);       # @array 现在仅包含一个元素的列表 (5)
unshift @array, 4;        # @array 现在是 (4, 5)
@others = 1..3;
unshift @array, @others; # @array 又变成了 (1, 2, 3, 4, 5)
```

与 pop 类似，对于一个空的数组变量，shift 会返回 undef。

注 11： 教育程度较高的读者会发现这句话其实是同义反复（tautology）。

字符串中的数组内插

和标量一样，数组的内容同样可以被内插到双引号串中。内插时，会在数组的各个元素之间自动添加分隔用的空格【注 12】：

```
@rocks = qw{ flintstone slate rubble };
print "quartz @rocks limestone\n";  # 输出 5 种以空白隔开的石头
```

数组被内插后，首尾都不会增添额外空格；若你真的需要，自己加吧：

```
print "Three rocks are: @rocks.\n";
print "There's nothing in the parens (@empty) here.\n";
```

假如你忘记了数组内插是这样写的，当你将电子邮件地址放进双引号内时，可能会大吃一惊：

```
$email = "fred@bedrock.edu";  # 错！这样会内插 @bedrock 这个数组
```

尽管我们只是想要显示电子邮件地址，Perl 却看到了一个叫做 @bedrock 的数组，并继而尝试将之内插。对某些版本的 Perl，我们会看到这样的警告信息【注 13】：

```
Possible unintended interpolation of @bedrock (可能意外的 @bedrock 内插)
```

要避免这种问题，要么将 @ 转义，要么直接用单引号来定义字符串：

```
$email = "fred\@bedrock.edu"; # 正确
$email = 'fred@bedrock.edu';  # 另一种写法，殊途同归
```

内插数组中的某个元素时，会被替换成该元素的值，如你所愿：

```
@fred = qw(hello dolly);
$y = 2;
$x = "This is $fred[1]'s place";     # 得 "This is dolly's place"
$x = "This is $fred[$y-1]'s place"; # 同上
```

请注意，索引表达式（index expression）会被当成普通字符串表达式处理。该表达式中的变量不会内插。也就是说，假如 $y 包含字符串 "2*4"，索引结果仍然为 1，而非 7。因为 "2*4" 被看作数字时（$y 在对数字求值的表达式中）相当于数字 2【注 14】。

注 12：事实上，此分隔符就是特殊变量 $" 的值，默认为空格。

注 13：某些 5.6 以前的 Perl 版本确实会将此视为致命错误，但后来改为警告信息，否则就太吵人了。

注 14：当然，如果你启用了警告信息，Perl 可能会提醒你 "2*4" 这个数字看起来怪怪的。

如果要在某个标量变量后面接着写左方括号，你需要先将这个方括号隔开，它才不至于被识别为数组检索的一部分。做法如下：

```
@fred = qw(eating rocks is wrong);
$fred = "right";                # 我们想要说 "this is right[3]"
print "this is $fred[3]\n";     # 用到了 $fred[3]，打印 "wrong"
print "this is ${fred}[3]\n";   # 打印 "right" （用花括号避开误解）
print "this is $fred"."[3]\n"; # 还是打印 right（用分开的字符串）
print "this is $fred\[3]\n";    # 还是打印 right （用反斜线避开误解）
```

foreach 控制结构

如果能对整个数组或列表进行处理，将是非常方便的，为此 Perl 提供了另一种控制结构。foreach（循环）能逐项遍历列表中的值，依次迭代（循环过程）：

```
foreach $rock (qw/ bedrock slate lava /) {
  print "One rock is $rock.\n";  # 依次打印所有三种石头的名称
}
```

每次循环迭代时，控制变量（control variable）（即此例中的 $rock）都会从列表中取得新的值。第一次执行时，控制变量的值是 "bedrock"；而第三次时，控制变量的值是 "lava"。

控制变量并不是列表元素的复制品——实际上，它就是列表元素本身。也就是说，假如在循环中修改了控制变量的值，也就同时修改了这个列表元素，如同下面的代码片段所示。这种设计很有效，也被广泛认可；但是如果没有预备，它却可以让你大吃一惊。

```
@rocks = qw/ bedrock slate lava /;
foreach $rock (@rocks) {
  $rock = "\t$rock";        # 在 @rocks 的每个元素前加上跳格符
  $rock .= "\n";            # 同时在末尾加上换行符
}
print "The rocks are:\n", @rocks; # 各自占一行，并使用缩排
```

当循环结束后，控制变量的值会变成什么？它仍然是循环执行之前的值。Perl 会自动存储 foreach 循环的控制变量，并在循环结束之后还原。在循环执行期间，我们无法访问或改变已存储的值，所以当循环结束时，变量仍然保持循环前的值；如果它之前从未被赋值，那就仍然是 undef。也就是说，假如你想将循环的控制变量取名为 $rock 的话，不必担心之前是否用过同名的变量。

Perl 的“老地方”：$_

假如你在 foreach 循环的开头省略了控制变量，Perl 就会使用它的“老地方”变量 $_。这个变量除了名称比较特别之外，和其他标量变量（几乎）没什么差别。举例来说：

```
foreach (1..10) {  # 默认会用 $_ 迭代
  print "I can count to $_!\n";
}
```

虽然它并非 Perl 中唯一的默认变量，却是最常见的一个。我们以后还会看到，在许多种情况下，当未告知 Perl 使用哪个变量或数值时，Perl 都会自动使用 $_，从而使程序员免于命名（和键入）新变量的痛苦。没错，这里的 print 就是一个例子，在没有参数时，它会打印 $_ 的值：

```
$_ = "Yabba dabba doo\n";
print;  # 默认打印 $_
```

reverse 操作符

reverse 操作符会读取列表的值（可能来自数组），并按相反的次序返回该列表。因此，假如你对范围操作符（即 ..）只能递增计数感到失望，可以这样来弥补：

```
@fred   = 6..10;
@barney = reverse(@fred);  # 得 10, 9, 8, 7, 6
@wilma  = reverse 6..10;   # 同上，但不需要额外的数组
@fred   = reverse @fred;   # 将逆序后的结果放回原来那个数组
```

值得注意的是，最后一行用了两次 @fred。Perl 总是会先计算（等号的右边）要赋的值，再实际进行赋值的操作。

请记住，reverse 会返回次序相反的列表，但它并不会修改传进来的参数。假如返回值无处可去，那这种操作也就变得毫无意义：

```
reverse @fred;          # 错误：这不会修改 @fred
@fred = reverse @fred; # 这才好
```

sort 操作符

sort 操作符会读取列表的值（可能来自数组），而且会根据内部的字符编码的顺序，对它们进行排序。对 ACSII 编码的字符串而言，排序的规则就是 ASCII 码序。当然 ASCII 的码序相当奇怪：大写字符排在小写字符前面，数字排在字母之前，而标点符号则散落各处。但是以 ASCII 码序进行排序只是默认的做法。在第十四章里，我们将会看到如何按自定的规则来进行排序。来看几个例子：

```
@rocks   = qw/ bedrock slate rubble granite /;
@sorted  = sort(@rocks);        # 得 bedrock, granite, rubble, slate
@back    = reverse sort @rocks; # 逆序后从 slate 到 bedrock 排列
@rocks   = sort @rocks;         # 将排序后的结果存回至 @rocks
@numbers = sort 97..102;        # 得 100, 101, 102, 97, 98, 99
```

从最后一个例子可以看出，将数字当成字符串来排序，这样的结果会不太对。根据默认的排序规则，任何以 1 开头的字符串会被排在以 9 开头的字符串之前。另外，它和 reverse 一样不会修改参数，所以要对数组排序时，你必须将排序后的结果存回数组：

```
sort @rocks;           # 错误，这么做不会修改 @rocks
@rocks = sort @rocks; # 现在收集来的石头排得井井有条
```

标量上下文与列表上下文

这是本章最重要的一节，事实上，这甚至也是本书最重要的一节。哪怕说你的 Perl 水平完全取决于对本节的了解程度，也一点都不夸张。所以，假如你到目前为止一直都是随意翻阅本书的话，现在应该是全神贯注的时候了。

这并不是说本节有多么难懂。这里的概念其实非常简单：同一个表达式，出现在不同的地方会有不同的意义。你应该不会陌生，因为在自然语言里，这种情况随处可见。以英语为例【注 15】，假如有人问你单词“read”【注 16】代表什么意思，你一定答不上来，因为在不一样的地方，它表达的意思可能会不同。除非你知道上下文（*context*），否则一定没办法确认它的含义。

所谓上下文，指的是表达式所在的位置。在 Perl 解析表达式的时候，要么希望得到一个标量，要么希望得到一个列表【注 17】。表达式所在的位置，Perl 期望得到什么，那就是该表达式的上下文【注 18】。

注 15：假如英语不是你的母语，这样的类比对你可能并不明显。但是每种语言都会有上下文感知（context sensitivity），因此你可以想出你自己语言上的例子。

注 16：如果是在讲话而不是写书，那他们也许是在问单词“red”代表什么意思（英文里，read 的过去式与 red 同音）。所以这么说无论如何都会造成混淆。就像 Douglas Hofstadter 所说的，没有任何语言能够毫无歧义地表达思想，尤其是英语。

注 17：当然，Perl 也可能期望得到其他的东西。另有一些上下文在这里无法介绍。事实上，没有人知道 Perl 到底有多少种上下文，Perl 长老团还没有对这个问题达成共识。

注 18：这和我们人类使用的语言也有相似之处。如果我犯了一个文法上的错误，而你能马上注意到错误，那是因为你会在某些位置上希望某些相关的用字。到后来，你一定也能以这种方法来读 Perl，不过在开始时得要先想一想。

```
42 + something # 这里的 something 必须是标量
sort something # 这里的 something 必须是列表
```

就算 *something* 这个单词的拼写保持不变，它仍然会在某种情况下得出单一的标量值，而在另外的情况下产生一个列表【注 19】。在 Perl 中，表达式总是根据所需要的上下文返回对应的值。以数组的“名称”【注 20】为例：在列表上下文中，它会产生元素的列表；但是在标量上下文中，则会返回数组中元素的个数：

```
@people = qw( fred barney betty );
@sorted = sort @people;   # 列表上下文：barney, betty, fred
$number = 42 + @people;   # 标量上下文：42 + 3 ，得 45
```

即使是普通的赋值运算（对标量或列表赋值），都可以有不同的上下文：

```
@list = @people; # 得到三个人的名单列表
$n = @people;    # 得到人数 3
```

但请不要立刻得出结论，认为在标量上下文中，一定会得到（列表上下文中返回的列表的）元素个数。大部分能产生列表的表达式【注 21】，在标量上下文中的返回值，可能要比元素个数*更有*意义。

在标量上下文中使用产生列表的表达式

有些表达式通常是用来产生列表的，假如你在标量上下文中使用它们，结果会怎样？那就要看该表达式的作者怎么说了。通常这位作者是 Larry，他会在说明文档里详细解释。事实上，学习 Perl 的大部分时间，都是在学习 Larry 的思维方式【注 22】。因此，只要你能以 Larry 的方式思考，就可以预测 Perl 接下来会怎么做。但在你学习的过程中，可能还是需要看看 Perl 的说明文档。

某些表达式不会在标量上下文中返回任何值。例如，`sort` 在标量上下文中会返回什么？其实你大可不必对列表排序以得知其元素个数。所以，除非有人修改了实现方式，否则 `sort` 在标量上下文中总是返回 `undef`。

注 19： 当然，列表可能恰好只有一个元素；它也可能是空的，或是具有任何数目的元素。

注 20： 对了，数组 `@people` 的名称就只是 `people`。`@` 符号只是一个限定词 (qualifier)。

注 21： 暂且不管这一小节的论点为何，其实“产生列表”的表达式和“产生标量”的表达式之间并无不同，任何表达式都可以产生列表或标量，根据上下文而定。因此当我们说“产生列表的表达式”时，我们指的是该表达式通常被用在列表上下文。所以当它们不小心被用在标量上下文中时（像 `reverse` 或是 `@fred`），你可能会因此感到惊讶。

注 22： 这很自然，因为在写 Perl 时，Larry 已经试着揣测你的想法，并预测你的需求了！

另一个例子是 reverse。在列表上下文中，它很自然地返回逆序后的列表；在标量上下文中，则返回逆序后的字符串（先将列表中所有字符串全部连接在一起，再对结果进行反序处理）：

```
@backwards = reverse qw/ yabba dabba doo /;
   # 会变成 doo, dabba, yabba
$backwards = reverse qw/ yabba dabba doo /;
   # 会变成 oodabbadabbay
```

刚开始时，往往很难一眼就看出，某个表达式到底是在标量上下文还是在列表上下文中。但是，请相信我们，它终将成为你的第二天性。

为了能够让你渐入佳境，下面列出一些常见的上下文：

```
$fred = something;              # 标量上下文
@pebbles = something;           # 列表上下文
($wilma, $betty) = something; # 列表上下文
($dino) = something;            # 还是列表上下文
```

不要被只有一个元素的列表给蒙骗了，最后一个例子是列表上下文，而不是标量上下文。这里的括号非常重要，使得第四行和第一行完全不同。如果是给列表赋值（不管其中元素的个数），那就是列表上下文。如果是给数组赋值，那还是列表上下文。

下面是其他我们曾见过的表达式，以及各表达式的上下文。先来看看标量上下文中使用 *something* 的例子：

```
$fred = something;
$fred[3] = something;
123 + something
something + 654
if (something) { ... }
while (something) { ... }
$fred[something] = something;
```

接下来再看看列表上下文的情况：

```
@fred = something;
($fred, $barney) = something;
($fred) = something;
push @fred, something;
foreach $fred (something) { ... }
sort something
reverse something
print something
```

在列表上下文中使用产生标量的表达式

这种情况十分简单：如果表达式求值结果为标量值，则自动产生一个仅含此标量值的列表：

```
@fred = 6 * 7; # 得到仅有单个元素的列表 (42)
@barney = "hello" . ' ' . "world";
```

好吧，其实这里有个小小的陷阱：

```
@wilma = undef;  # 糟糕！结果是得到一个列表，而且仅有的一个元素为未定义 (undef)
  # 这和下面的做法效果完全不同
@betty = ( );    # 这才是正确的清空数组的方法
```

因为 undef 是标量值，所以将 undef 赋值给数组并不会清空该数组。要清空的话，直接赋值一个空列表就可以了【注 23】。

强制指定标量上下文

偶尔在列表上下文的地方，你想要强制引入标量上下文，这可以使用伪函数 scalar。它可不是真正的函数，只不过告诉 Perl 这里要切换到标量上下文：

```
@rocks = qw( talc quartz jade obsidian );
print "How many rocks do you have?\n";
print "I have ", @rocks, " rocks!\n";          # 错误，这会输出各种石头的名称
print "I have ", scalar @rocks, " rocks!\n"; # 对了，打印出来的是石头种数
```

说来也奇怪，没有相应的函数可用来强行切入列表上下文。原因很简单，因为你根本用不到，相信我们。

列表上下文中的 <STDIN>

我们之前看过的 <STDIN> 操作符，在列表上下文中会返回不同的值。就像前面提过的，<STDIN> 在标量上下文中会返回输入数据的下一行；在列表上下文中，则会返回*所有*剩下的行，直到文件结尾为止。返回的每一行都会成为列表中的某个元素。举例来说：

```
@lines = <STDIN>; # 在列表上下文中读取标准输入
```

当输入数据来自某个文件时，它会读取文件的剩余部分。但是如果输入数据的来源是键盘，应该如何表达文件结尾呢？对 Unix 或类似的系统（包括 Linux 和 Mac OS X）来

注 23：实际上，如果变量是在某个特定的词法域内声明的，就不用显式清空，因为程序运行到此范围外，就会自动恢复初始的清空状态。所以这种写法在实际的程序中并不多见。下一章就会介绍有关词法域的内容。

说，通常可以键入 Control-D 【注 24】来告知系统，不会再有任何输入了。即使这个特殊字符会被打印在屏幕上，Perl 也不会看到它【注25】。对 DOS/Windows 系统来说，就要用 Control-Z 了【注 26】。假如你使用的是其他操作系统，请查看系统的说明文档或就近找专家咨询。

假如运行程序的人在键入了三行数据之后，又用适当的按键来表示文件结尾，最后得到的数组就会包含三个元素。每个元素都是以换行符结尾的字符串，因为这些换行符也是输入的内容。

如果在读取这些数据行时，能一次性 chomp 所有的换行符岂不更好？可以直接把数组交给 chomp，它会自动去掉每个元素的换行符。比如：

```
@lines = <STDIN>; # 读入所有行
chomp(@lines);    # 去掉所有的换行符
```

不过更常见的做法，是按之前介绍的风格写成：

```
chomp(@lines = <STDIN>); # 读入所有行，换行符除外
```

虽然你可以按个人喜好来决定怎么做，不过大多 Perl 程序员都会倾向于第二种更紧凑的写法。

输入数据一经读取，就无法回头再读一次。这也许对你（但不见得对所有人）而言是理所当然的事【注 27】，一旦读到文件结尾，就没有更多输入数据可供读取了。

如果要读入的是某个 400MB 大小的日志文件，会有什么结果？“行输入”操作符会读取所有的数据行，同时占用大量内存【注 28】。Perl 不会限制你做这种事，但是系统中的其他用户（当然还有系统管理员）很可能会抗议。在读取大量数据时，通常应该考虑替代方案，避免一次将全部的数据读进内存。

注 24：这只是默认按键，你可用 stty 命令加以变更。但是这个默认配置很普遍——我们还没见过哪种 Unix 系统会使用其他字符来代表文件结尾。

注 25：实际上是操作系统“看到了”这个控制按键，并据此向应用程序报告“文件结尾”的信号。

注 26：在 DOS/Windows 上的某些 Perl 版本有个缺陷：按下 Control-Z 之后，在屏幕上输出的第一行会被盖掉。你可以在读进输入后，输出一个空白行（即 "\n"）来解决这个问题。

注 27：没错，如果你的输入来源可以使用 seek 函数，那么你也可以返回起点，从头再读入一遍。但这种做法并不适合在此处讨论。

注 28：通常，需要的内存远超过文件的大小。换句话说，一个 400MB 大小的文件在读进数组时，通常会占据至少 1GB 的内存。这是因为 Perl 通常会浪费内存来节省时间。这是个不错的取舍：假如内存不够，还可以再买一些；但如果时间不够，那就完了。

习题

以下习题答案参见附录 A：

1. [6] 写一个程序，读入一些字符串（每行一个），直到文件结尾为止。然后，再以相反顺序输出这个列表。假如输入来自键盘，你需要在 Unix 系统上键入 Control-D 或在 Windows 系统上键入 Control-Z 来表示输入的结束。

2. [12] 写一个程序，读入一些数字（每行一个），直到文件结尾为止。然后，根据每一个数字，输出如下名单中相应的人名（请将这份名单写到程序里；也就是说，你的程序代码里应该出现这份名单）。比方说，如果输入的数字是 1、2、4 和 2，那么输出的人名将会是 fred、betty、dino 和 betty。

```
fred betty barney dino wilma pebbles bamm-bamm
```

3. [8] 写一个程序，读入一些字符串（每行一个），直到文件结尾为止。然后，请按照 ASCII 码序输出所有字符串。换句话说，假如你键入的是 fred、barney、wilma、betty，输出应该显示 barney betty fred wilma。所有的字符串可以成一行输出吗？或是分开在不同行输出？你能分别让程序以这两种方式输出吗？

第四章

子程序

我们已见过并用过一些内置的系统函数，像 chomp、reverse 和 print 等。但是，就如同其他的语言一样，Perl 也可以让你创建子程序（*subroutines*），也就是用户定义的函数【注 1】。它让我们可以重复利用已有的代码【注 2】。子程序名也是 Perl 标识符（由字母、数字和下划线组成，但是不能以数字开头），有时候视情况会以“与号”（&）开头。关于何时可以省略及何时需要这个“与号”，我们将在本章末尾进行介绍。没有其他声明的情况下，我们都将使用该符号，通常来说这是一种保险的做法。

子程序名属于独立的名字空间，这样 Perl 并不会将同一段代码中的子程序 &fred 和标量 $fred 搞混淆，然而这样做往往不合常理。

定义子程序

要定义你自己的子程序，可使用关键字 sub，子程序名（不包含“与号”）以及经过缩进的代码块（花括号中的内容）【注 3】，它们组成了子程序的主体，举例来说：

注 1：在 Perl 中，我们一般不会像 Pascal 程序员那样区分有返回值的函数和无返回值的子过程（procedure）。但是请注意子程序总是由用户定义的，而函数则不一定。所以，有时候函数这个词可能被当成子程序的同义词，但它也可能指某个 Perl 的内置函数。大多数时候，可以自己定义的是子程序，而不是内置函数，所以我们这章的主题是子程序。

注 2：本书中至少 40% 的代码示例都取自于已有的其他程序，如果你合理利用，那它们之中至少 75% 的代码也是可以被你重用的。

注 3：好吧，爱较真的家伙们，我们承认：恰当地说，花括号是代码块的组成部分。另外 Perl 并不要求缩进 —— 但维护程序员会需要。所以还是遵循良好的编码风格吧。

```
sub marine {
  $n += 1;  # 全局变量 $n
  print "Hello, sailor number $n!\n";
}
```

子程序可以在程序的任意位置定义，有 C 语言或 Pascal 背景的程序员喜欢将子程序的定义放在文件的开头。其他人可能更喜欢将它们放在文件的结尾，从而使程序主体出现在开头部分。在 Perl 中这完全取决于你。任何情况下你都不需要对子程序进行事先声明【注 4】。子程序的定义是全局的；除非你使用一些强有力的技巧，否则不存在所谓的私有子程序【注 5】。假如你定义了两个重名的子程序，那么后面的那个子程序会覆盖掉前面的【注 6】。一般来说，重名这种做法很不好，它会让维护程序员感到困惑。

正如你在之前的例子中看到的，你可以在子程序中使用任何全局变量。事实上，目前你见过的所有变量都是全局的。这意味着你可以在程序的任何位置访问这些变量。很多语言学的好事者又该为此而感到“震惊”了，但 Perl 开发组很多年前就组织了一队手持“火把”的“愤青团”，把这些好事者“赶出城”。我们稍后会在“子程序中的私有变量”一节中学到如何建立私有变量。

调用子程序

你可以在任意表达式中使用子程序名（前面加上“与号”）来调用它：【注 7】

```
&marine;  # 打印 Hello, sailor number 1!
&marine;  # 打印 Hello, sailor number 2!
&marine;  # 打印 Hello, sailor number 3!
&marine;  # 打印 Hello, sailor number 4!
```

通常，我们把对子程序的调用称为呼叫（*calling*）子程序。

注 4：除非你的子程序比较特别且定义了“原型”，“原型”告诉编译器如何解析和解释传进来的参数。这种情况比较少见 —— 详细信息请参考 *perlsub* 在线手册。

注 5：如果你想了解这些强大的技巧，请阅读 Perl 的文档中关于保存在私有变量（lexical 变量）中的代码引用（coderef）的相关章节。

注 6：这种情况值得你注意。

注 7：通常还会在后面加上一对括号，哪怕是空括号。因为子程序会继承调用者的 @_ 值，这个我们马上将会讨论，所以请继续阅读，不然你写的代码可能会和你预期的效果不同！

返回值

子程序被调用时一定是作为表达式的某个部分，即使该表达式的求值结果不会被用到。之前我们在调用 &marine 的时候，先对包含调用动作的表达式求值，但随即就把结果丢弃了。

很多时候，我们需要调用某个子程序并对它的返回值做进一步的处理。所以我们需要注意子程序的返回值。在 Perl 中，所有的子程序都有一个返回值 —— 子程序并没有“有返回值”和“没有返回值”之分。但并不是所有的 Perl 子程序都包含有用的返回值。

既然任何 Perl 子程序都有返回值，那么规定每次必写“return 某值”就显得有点费事，所以 Larry 将它简化了。在子程序的执行过程中，它会不断进行运算。而最后一次运算的结果（不管是什么），都会被自动当成子程序的返回值。

举例来说，我们定义如下的子程序：

```
sub sum_of_fred_and_barney {
  print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
  $fred + $barney;  # 这就是返回值
}
```

这个子程序里最后执行的表达式，就是计算 $fred 与 $barney 的总和。因此，$fred 与 $barney 的总和就是返回值。以下是实际运行的情况：

```
$fred = 3;
$barney = 4;
$wilma = &sum_of_fred_and_barney;      # $wilma 为 7
print "\$wilma is $wilma.\n";
$betty = 3 * &sum_of_fred_and_barney;  # $betty 为 21
print "\$betty is $betty.\n";
```

这段代码会产生如下的输出：

```
Hey, you called the sum_of_fred_and_barney subroutine!
$wilma is 7.
Hey, you called the sum_of_fred_and_barney subroutine!
$betty is 21.
```

此处的 print 语句只用于协助调试，让我们得以确定该子程序被调用了，程序完工后便可将它删除。不过，假设你在这段程序代码的结尾新增了另外一行，像这样：

```
sub sum_of_fred_and_barney {
  print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
  $fred + $barney;  # 这不是返回值!
  print "Hey, I'm returning a value now!\n";      # 糟糕!
}
```

本例中，最后执行的表达式并非加法运算，而是 print 语句。它的返回值通常是 1，代表“输出成功”【注 8】，但它不是我们真正想要返回的值。所以在子程序里增加额外的程序代码时，请小心检查最后执行的表达式是哪一个，确定它是你要的返回值。

那么，那个第二个（不完善的）子程序中 $fred 与 $barney 相加的结果怎样了？我们并没有将总和存储起来，所以 Perl 会丢弃它。启用警告信息时，Perl 会注意到将两数相加的结果丢弃是毫无用处的，并显示“a useless use of addition in a void context”之类的信息来警告你。*void context*（空上下文）这个术语只是表示运算结果没有存储到变量里，也未被任何函数使用。

“最后执行的表达式”的准确含义是最后执行的表达式，而非程序代码的最后一行。比如说，下面这个子程序将会返回 $fred 和 $barney 两者中值较大者：

```
sub larger_of_fred_or_barney {
  if ($fred > $barney) {
    $fred;
  } else {
    $barney;
  }
}
```

上面的例子中，最后执行的表达式是自成一行的 $fred 或是 $barney，他们将充当子程序的返回值。必须等到执行阶段，得知这些变量的内容后，我们才会知道返回值到底是 $fred 还是 $barney。

目前为止都还只是些范例而已。接下来会介绍如何在每次调用子程序时传入不同的值，而不用再依靠全局变量，这就会越来越有意义了。

参数

假如子程序 larger_of_fred_or_barney 不强迫我们一定用全局变量 $fred 和 $barney，那么使用上会更灵活。在目前的状况下，假如我们想从 $wilma 和 $betty 中取出较大值，必须先将它们复制到 $fred 和 $barney，才可以使用子程序 larger_of_fred_or_barney。此外，假如别处需要用到 $fred 和 $barney 原本的值，我们还得先将它们复制到其他变量，比方说 $save_fred 和 $save_barney。然后当子程序执行完毕后，我们又必须将这些值复制回 $fred 和 $barney。

注 8：当 print 成功时返回真值，失败时返回假值。你将在下一章了解到如何检测各种出错。

还好，Perl 子程序可以有参数（argument）。要传递参数列表（argument list）到子程序里，只要在子程序调用的后面加上被括号圈引的列表表达式（list expression）就行了。做法如下所示：

```
$n = &max(10, 15);  # 包含两个参数的子程序调用
```

参数列表将会被传入子程序，让子程序随意使用。当然，得先将这个列表存在某处，Perl 会自动将参数列表化名为特殊的数组变量 @_，该变量在子程序执行期间有效。子程序可以访问这个数组，以判断参数的个数以及参数的值。

这表示子程序的第一个参数存储于 $_[0]，第二个参数存储于 $_[1]，依次类推。但是，请特别注意，这些变量和 $_ 变量毫无关联，就像 $dino[3]（数组 @dino 中的元素之一）与 $dino（一个独立的标量变量）毫无关联一样。参数列表总得存进某个数组变量里，好让子程序使用，而 Perl 将这个数组叫做 @_，仅此而已。

现在，你可以写一个类似于 &larger_of_fred_or_barney 的子程序 &max，但是可以使用子程序的第一个参数（$_[0]）而不用 $fred，也可以使用子程序的第二个参数（$_[1]）而不用 $barney。因此，最后你可以将 &max 写成这样：

```
sub max {
  # 请比较它和子程序 &larger_of_fred_or_barney 的差异
  if ($_[0] > $_[1]) {
    $_[0];
  } else {
    $_[1];
  }
}
```

好吧，就像上面所说的，你可以那么写。但是，这里的一堆下标让程序变得不雅观，而且难以阅读、编写、检查和调试。我们马上就会看到更好的办法。

这个子程序还存在另外一个问题。&max 这个名字虽然既好听又简洁，但是却没有说明这个子程序只接受两个参数：

```
$n = &max(10, 15, 27);  # 糟糕!
```

多余的参数会被忽略——反正子程序不会用到 $_[2]，所以 Perl 并不在乎里面是否有值。参数如果不足也会被忽略——如果用到超出 @_ 数组边界的参数，只会得到 undef。本章稍后，我们将会看到如何写出更好的 &max 子程序，并且让它可以配合任意数目的参数。

@_ 变量是子程序的私有变量【注 9】。假如已经有了全局变量 @_，则该变量在子程序调用前会先被存起来，并在子程序返回时恢复原本的值【注 10】。这也表示，子程序可以将参数传给其他的程序，而不用担心遗失自己的 @_ 变量。就算是嵌套的子程序（nested subroutine），在访问私有的 @_ 变量时也一样。即使子程序递归调用自己，但每次调用时仍然会取得一个新的 @_。所以在当前的子程序调用中，@_ 总是包含了它的参数列表。

子程序中的私有变量

既然每次调用子程序时 Perl 都会给我们新的 @_，难道不能让它产生私有变量吗？这当然可以。

默认情况下，Perl 中所有的变量都是全局变量。也就是说，在程序里的任何地方都可以访问它们。但是你可以随时运用一个操作符来创建私有的词法变量（*lexical variables*）。这个操作符就是 my：

```
sub max {
  my($m, $n);          # 该语句块中的新私有变量
  ($m, $n) = @_;       # 将参数赋值给变量
  if ($m > $n) { $m } else { $n }
}
```

这些变量的作用范围（*scope*）被圈定在语句块中（它们是该语句块内的私有变量），其他地方的 $m 或 $n 完全不受这两个私有变量的干扰。反之，别的程序代码也都无法访问或修改这些私有变量，不管是无心或是有意【注 11】。所以，我们可以把这个子程序放进世界上任何一个 Perl 程序里，且不用担心与那个程序中（可能存在）的 $m 和 $n 变量冲突【注 12】。另外还值得一提的是，在前一个例子中的 if 语句块中，作为返回值的表达式后面没有分号。虽然 Perl 允许你省略语句块中最后一个分号，但实际上通常只有像前面的例子那样，在程序代码简单到整个语句块只有一行时，才有必要省略分号。

注 9：除非调用子程序的时候前面加了“与号”，且后面没有跟括号（或者参数）。那种特殊情况下 @_ 数组会从调用者上下文中继承下来。一般来说这不是个好主意，但有时也有它的用处。

注 10：你也许认识到了这和上一章里 foreach 循环保存控制变量的机制相同，变量的值都是被 Perl 自动地保存和恢复的。

注 11：一些高级程序员应该知道，词法变量可以通过引用来从其作用域之外访问到，而不是通过变量名。

注 12：当然，如果那个程序中恰巧也有个 &max 子程序，就肯定会搞混淆。

前一个例子中的子程序还可以进一步简化。你是否注意到列表（$m, $n）被写了两次？my 操作符也可以应用到括号内的变量列表，所以习惯上会将这个子程序中的前两行语句合并起来：

```
my($m, $n) = @_;  # 将保存在 @_ 中的参数赋值给具体的变量
```

这一行语句会创建私有变量并为它们赋值，让第一个参数的名称变成较好记的 $m，而第二个参数则为 $n。几乎所有的子程序都会以类似的程序代码作为开头。当你看到这一行时就会知道，这个子程序具有两个标量参数，而在子程序中它们分别被称为 $m 和 $n。

长度可变的参数列表

在真实的 Perl 代码中，常常把更长的（任意长度的）列表作为参数传给子程序。这延续了 Perl 的理念“去除不必要的限制”。当然，这点和许多传统程序语言并不一样。那些语言需要“强类型”（strictly typed）的子程序；也就是说，只允许一定个数的参数并且预先限制它们的类型。Perl 的灵活是件好事，不过当子程序以超乎作者预期个数的参数被调用时（例如我们之前看到的子程序 &max），也许会造成问题。

当然，子程序可以很容易地通过检查 @_ 数组的长度来确定参数的个数是否正确。比方说，我们可以将 &max 写成如下的形式以检查它的参数列表【注 13】：

```
sub max {
  if (@_ != 2) {
    print "WARNING! &max should get exactly two arguments!\n";
  }
  # 代码和前面一样，这里省略
  .
  .
  .
}
```

上面的 if 判断，是在标量上下文中直接使用数组的“名称”来取得数组元素的个数，你应该在第三章中见过这个用法。

但是在追求实用的 Perl 程序中，这种检查方式很少见，比较好的做法是让子程序能够适应任意数目的参数。

注 13：当你从下一章学了 warn 函数后，便可以换用它来生成合适的警告。如果出错情况比较严重，可以用 die，这个函数同样会在下一章学到。

更好的 &max 子程序

那么，让我们改写 &max，让它能接受任意数目的参数：

```
$maximum = &max(3, 5, 10, 4, 6);

sub max {
  my($max_so_far) = shift @_;  # 数组中的第一个值，暂时把它当成最大值
  foreach (@_) {               # 遍历数组 @_ 中的其他元素
    if ($_ > $max_so_far) {    # 当前元素比 $max_so_far 更大吗?
      $max_so_far = $_;
    }
  }
  $max_so_far;
}
```

上面的程序代码使用了一般称为“高水线（high-water mark）”的算法：大水过后，在最后一波浪消退时，高水线会标示出最高的水位。本例中，$max_so_far 记录了高水线，也就是我们见过的最大数字。

第一行程序代码会对参数数组 @_ 进行 shift 操作，并将所得到的 3（范例程序里的第一个参数）存入 $max_so_far 变量。所以 @_ 现在的内容为 (5, 10, 4, 6)，因为 3 已被移走。现在最大的数字是 3，也就是第一个参数。

然后 foreach 循环会遍历参数列表 @_ 里其他的值。循环的控制变量默认为 $_（别忘了 @_ 和 $_ 没有任何关系，它们的名称相似纯属巧合）。循环第一次执行时，$_ 是 5，而 if 进行比较时看到 $_ 比 $max_so_far 还大，所以 $max_so_far 会被设成 5 —— 新的高水线。

下一次执行循环时，$_ 是 10。这是新的最大值，所以它会被存入 $max_so_far。

下一次，$_ 是 4。这时 if 比较的结果为假，因为 $_ 不比 $max_so_far（即 10）大，所以会跳过 if 里的程序代码。

下一次，$_ 是 6，因此 if 里的程序代码又被跳过一次。这是最后一次执行循环，所以整段循环就执行完了。

此时，$max_so_far 就变成了返回值。既然它是目前最大值，并且我们已经遍历过所有数字，那它一定是列表中最大的值：10。

空参数列表

即使超过两个参数，改进后的 &max 算法现在仍然适用。但是假如没有任何参数，又会怎样呢？

乍听之下，这似乎有点杞人忧天。毕竟，怎么可能会有人调用 &max 却不传入任何参数？但是，也许会有人写出如下的程序代码：

```
$maximum = &max(@numbers);
```

某些时候，数组 @numbers 或许是一个空的列表。举例来说，也许数组中的内容是程序从文件里读入的，但文件却是空的。所以，这种情况下 &max 会怎样呢？

子程序的第一行会对参数数组 @_（现在是空的）进行 shift 操作，以此作为 $max_so_far 的值。这并不会出错，因为数组是空的，所以 shift 会返回 undef 给 $max_so_far。

现在 foreach 循环要遍历 @_ 数组，但是由于 @_ 是空的，所以循环本身不会被执行。

接下来，Perl 会将 $max_so_far 的值 undef 作为子程序的返回值。从某种角度来看，那是正确的结果，因为在空列表中没有最大的值。

当然，调用这个子程序的人得留意，返回值可能是 undef，除非他能确保参数列表不为空。

关于词法（my）变量

事实上，词法变量（lexical variable）可使用在任何块（block）内，而不仅限于子程序的语句块。举例来说，它可以在 if、while 或 foreach 的语句块里使用：

```
foreach (1..10) {
  my($square) = $_ * $_;  # 该循环中的私有变量
  print "$_ squared is $square.\n";
}
```

上面的例子中，变量 $square 对其所属语句块（也就是 foreach 循环的语句块）来说是私有的。如果变量的定义并未出现在任何语句块里，则该变量对于整个程序文件来说就是私有的。到目前为止，你的程序还用不到两个以上的程序文件，所以这还不成问题。这里的重点在于，词法变量的作用域受限于（引入它的）最内层语句块（或文件）。只有语句块上下文范围（textual scope）内的程序代码才能以 $square 这个名称使用该变量。这在程序维护上非常有利。如果 $square 的值出错了，那么就可以在有限的程序代码范围内找到罪魁祸首。有经验的程序员都知道（这往往是付出惨痛代价换来的），将变量范围圈定在一页或少数几行程序代码内，的确能加快开发及测试周期。

还需要注意的是，my 操作符并不会更改变量赋值时的上下文：

```
my($num) = @_;  # 列表上下文，和 ($num) = @_; 相同
my $num  = @_;  # 标量上下文，和 $num = @_; 相同
```

在第一行里，按照列表上下文，$num 会被设为第一个参数；在第二行里，按照标量上下文，它会被设为参数的个数。这两行都有可能是程序员的本意，我们并不能单从一行程序代码断定你要的是哪个，因此在你搞错时，Perl 无法提出警告（当然，你不会真的把这两行放在同一个子程序里，因为相同的作用范围内不能定义两个同名的词法变量，以上只是举例而已）。所以，看到这样的程序代码时，你可以忽略 my 这个词，直接判断变量赋值时的上下文。

每当我们谈到 my() 操作符时，请记住，在 my 不使用括号时，只用来声明单个词法变量【注 14】：

```
my $fred, $barney;          # 错！没声明 $barney
my($fred, $barney);         # 两个都声明了
```

当然，你也可以使用 my 来创建新的私有数组【注 15】：

```
my @phone_number;
```

所有新变量的值一开始都是空的：标量被设为 undef，数组被设为空列表。

use strict 编译命令

Perl 通常是个相当宽容的语言【注 16】。但是，也许你会想要 Perl 更严格一些，这时只要使用 use strict 编译命令就行了。

所谓编译命令（*pragma*），就是对编译器的指示，告诉它关于程序代码的一些信息。本例使用了 use strict 编译命令，这会让 Perl 语法编译器（对当前语句块或源文件剩下的部分）强制执行一些严格的、确保良好程序设计的规则。

这么做的重要性何在？嗯，设想你正在写程序，并键入如下的一行程序：

```
$bamm_bamm = 3;  # Perl 会自动创建这个变量
```

注 14：像往常一样，打开 warning 开关可以在 my 被滥用的时候产生警告信息，否则你就得自己打电话给 1-800-LEXICAL-ABUSE 报告这个了（搞笑）。使用 strict 编译命令（稍后会提及）可以避免这类错误的发生。

注 15：或者哈希也行，这个你将在第六章看到。

注 16：而你大概还没有注意到。

现在，你继续写了一些代码。当上一行程序被新写的代码挤出屏幕顶端后，你又写了下面这行程序，想增加那个变量的值：

```
$bammbamm += 1;  # 糟了!
```

因为 Perl 看到了一个新的变量名（变量名中的下划线是有意义的），它会创建一个新的变量，然后增加它的值。如果你有先见之明，已经启用了 warning 的警告功能，那么 Perl 就会警告你这两个全局变量（或其中之一）在程序中只出现过一次。可是如果你不够谨慎，那么这两个变量都会在程序里并存，Perl 也并不发出警告。

要告诉 Perl 你愿意接受更严格的限制，请将 use strict 这个编译命令放在程序开头（或在任何想要强制使用这些规则的语句块和文件里）：

```
use strict;  # 强制使用一些严格的、良好的程序语言规则
```

就目前看来约束在于【注17】：Perl 会要求你一定要用 my 来声明每个新的变量【注18】。

```
my $bamm_bamm = 3;  # 新的词法变量
```

现在，如果你再像之前那样拼错变量名，Perl 就会抗议，说你从未声明 $bammbamm 这个变量，因此错误在编译阶段就会被找出来。

```
$bammbamm += 1;  # 没有这个变量：这会是编译时的严重错误（fatal error）
```

当然，此限制只适用于新创建的变量；Perl 的内置变量（如 $_ 和 @_）则完全不用事先声明【注19】。如果你在程序已经写完之后再加 use strict，通常会得到一大堆警告信息，所以如果有需要，最好是在一开始写程序时就使用它。

根据大部分人的建议，比整个屏幕长的程序都应该加上 use strict。这一点我们也相当认同。

注 17：要学习其他的限制规则，请阅读 strict 的文档。知道任何编译命令的名字也就知道了相应文档名，所以你可以使用 *perldoc strict*（或者你系统的查看文档命令）来查看 strict 的文档。简单来说，大多数情况下其他的限制规则会要求对字符串使用引号，并且所有的引用必须是硬引用。放心，这些限制规则不会对 Perl 的初学者造成困扰。

注 18：当然还有其他的方法声明变量。

注 19：最好别声明 $a 和 $b 这两个变量，因为在某些情况下，他们默认被用作 sort 函数的内置变量，换用其他的变量名吧。事实上 use strict 不能限制这两个变量，而这也常被当作 bug 提交给 Perl 维护者。

从这里开始，即使没有清楚写出 use strict，我们所提供的大部分（但并非全部）范例程序也都会在它的限制下执行。也就是说，在合适的位置尽可能用 my 来声明变量。不过，即使我们没有从头到尾都这样做，仍然建议你尽可能在程序里加进 use strict。

return 操作符

return 操作符会从子程序中立即返回某个值：

```
my @names = qw/ fred barney betty dino wilma pebbles bamm-bamm /;
my $result = &which_element_is("dino", @names);

sub which_element_is {
  my($what, @array) = @_;
  foreach (0..$#array) {  # 数组 @array 中所有元素的下标
    if ($what eq $array[$_]) {
      return $_;          # 一发现就提前返回
    }
  }
  -1;                     # 没找到符合条件的元素 (return 在这里可有可无)
}
```

上面的程序里，&which_element_is 子程序被用来找出 "dino" 在数组 @names 中的下标值。首先，我们会用 my 声明参数名：我们要找的 $what，以及供程序搜索的数组 @array。本例中，@array 将会是数组 @names 的拷贝。foreach 循环将会利用下标值逐项处理 @array（第一个下标值是 0，最后一个下标值是 $#array，和第三章的一样）。

每一次执行 foreach 循环时，我们会检查 $what 中的字符串是否等于【注 20】数组 @array 的当前元素。如果两者相等，我们就立刻返回其下标值。这是写 Perl 程序时关键字 return 最常见的用法：立即返回某个值，而不要执行子程序的其余部分。

但是，假如我们找不到符合条件的那个元素呢？本例中，子程序的作者选择返回 -1 以作为“查无此值”的代码。虽然在这个例子里，返回 undef 可能会更符合 Perl 的风格（Perlish），但是这个程序员决定使用 -1。最后一行写成 return -1 也行，但实际上并不需要用到 return 这个单词。

有些程序员在每次返回值的时候都会使用 return，以表明它是返回值。这一章前面介绍子程序 &larger_of_fred_or_barney 的时候说过，只有返回发生在子程序的最后一行代码之前的时候，才需要使用 return。其实虽然不必写，而且写了也不犯罪，然而很多 Perl 程序员认为它是多余的七次按键。

注 20：注意这里使用的是字符串相等比较符 eq，而不是数字相等比较符 ==。

省略“与号”

遵照先前的承诺，我们现在要告诉你，在调用子程序时何时可以省略“与号”。如果编译器在调用子程序之前看到了子程序的定义，或者 Perl 可以从语法识别它是子程序调用，那么该子程序就可以像内置函数那样，在调用时省略“与号”（不过我们马上就会看到，这个规则潜藏了一个陷阱）。

换句话说，假如 Perl 单从语法上便能看出它是省略了“与号”的子程序调用，通常就不会有什么问题。也就是说，你只要将参数列表放进括号里，它就一定是函数【注 21】调用：

```
my @cards = shuffle(@deck_of_cards);  # shuffle 前没必要用 &
```

或者，如果 Perl 的内部编译器已经见过子程序的定义，那么“与号”通常也可以省略。这种情况下，你甚至可以去掉参数列表两边的括号：

```
sub division {
  $_[0] / $_[1];                    # 用第一个参数除以第二个参数
}

my $quotient = division 355, 113;  # 使用 &division
```

上面的程序可以运行，是因为它符合“在不影响程序代码意义的前提下，括号可以省略”这条规则。

但是不要将子程序的声明放在调用子程序之后，不然编译器就无法知道 division 的意义何在。编译器需要在子程序调用之前先看到子程序的定义，才有办法像内置函数般调用该子程序。

不过这还不算陷阱。真正的陷阱是：假如这个子程序和 Perl 内置函数同名，你就必须使用“与号”来调用。配合“与号”，你才有办法正确调用子程序；少了它就只能在没有同名内置函数的状况下，正确调用子程序：

```
sub chomp {
  print "Munch, munch!\n";
}

&chomp;  # 这里必须使用 &，绝不能省略！
```

如果少了“与号”，就算我们已经定义过子程序 &chomp，仍然会调用到内置函数 chomp。所以，真正的省略规则如下：除非你知道 Perl 所有的内置函数名，否则请务

注 21：在本例中，这个函数就是子程序 &shuffle。但是正如你一会儿将要看到的一样，它也可能是个内置函数。

必在调用函数时使用“与号”。也就是说，你写的前一百个程序里都可以加上“与号”。但是当你看到其他人的程序中省略“与号”，那不一定是个错误；也许他们已经知道 Perl 没有同名的内置函数【注 22】。当程序员打算以调用 Perl 内置函数的方式来调用自己的子程序时，通常是编写模块的时候，他们经常使用原型（*prototype*）来告诉 Perl 该如何对待子程序的参数。构造模块是高级课题，但是如果你已经预备好了，可以参阅 Perl 的说明文档（特别是在线手册 *perlmod* 和 *perlsub*），以便了解更多关于子程序原型和构造模块的信息。

非标量返回值

子程序不仅可以返回标量值，如果你在列表上下文中调用它【注 23】，它就能返回列表值。

假设你想取出某段范围的数字（如同范围操作符 .. 的递增序列），只是你不但想递增，还想要递减序列。虽然范围操作符只能产生递增序列，不过要反过来取也不是什么难事：

```
sub list_from_fred_to_barney {
  if ($fred < $barney) {
    # 从 $fred 数到 $barney
    $fred..$barney;
  } else {
    # 从 $fred 倒数回 $barney
    reverse $barney..$fred;
  }
}
$fred = 11;
$barney = 6;
@c = &list_from_fred_to_barney; # @c 的值为 (11, 10, 9, 8, 7, 6)
```

此例中，我们会先用范围操作符取得从 6 到 11 的列表，再用 `reverse` 操作符把它反转过来。最后的结果就是我们想要的：`$fred`（11）倒数到 `$barney`（6）的列表。

其实你还可以什么都不返回。单写一个 `return` 不给任何参数时，在标量上下文中返回值就成了 `undef`，在列表上下文中则成了空列表。这可以表示子程序执行有误，能够告诉调用者，没办法得到有意义的返回值。

注 22：当然，也许这确实是个错误。你可以搜索 *perlfunc* 和 *perlop* 的在线手册来查看它是否是某个内置函数的名字。如果你打开了 warning 开关，那么一般 Perl 会给你相关的警告信息。

注 23：你可以用 `wantarray` 函数来判断子程序是在标量上下文还是列表上下文环境中被执行的，它可以让你在写子程序的时候更容易地判断出什么时候用列表，什么时候用标量。

持久性私有变量

在子程序中可以使用 my 操作符来创建私有变量，但每次调用这个子程序的时候，这个私有变量都会被重新定义。使用 state 操作符来声明变量，我们便可以在子程序的多次调用间保留变量的值，并将变量的作用域局限于子程序中。

让我们返回去看一看这章的第一个例子，我们有个子程序名为 marine，它会使变量的值增加：

```
sub marine {
  $n += 1;  # 全局变量 $n
  print "Hello, sailor number $n!\n";
}
```

现在我们才见识到 strict 的厉害，只要把它加到我们的程序中，你就会看到之前这段程序里的全局变量 $n 现在是不允许使用的。另外我们也不能用 my 来将 $n 声明为词法变量，因为词法变量不能保留变量的值（语句块结束了，词法变量的值也就不存在了）。

我们可以用 state 来声明变量，它会告诉 Perl 该变量是某个子程序的私有变量，并且在子程序的多次调用间保留该变量的值：

```
use 5.010;

sub marine {
  state $n = 0;  # 持久性私有变量 $n
  $n += 1;
  print "Hello, sailor number $n!\n";
}
```

现在我们可以在用了 strict 编译命令，并且不用全局变量的前提下，得到和之前相同的输出了。当我们第一次调用该子程序的时候，Perl 声明并初始化变量 $n，而在接下来的调用中，这个表达式将被 Perl 忽略。每次子程序返回后，Perl 都会将变量 $n 的当前值保留下来，以备下次调用中使用。

类似标量变量，其他任意类型的变量都可以被声明为 state 变量。下面的子程序可以通过声明 state 数组来保留它的参数及计算总和：

```
use 5.010;

running_sum( 5, 6 );
running_sum( 1..3 );
running_sum( 4 );

sub running_sum {
```

```
state $sum = 0;
state @numbers;

foreach my $number ( @_ ) {
  push @numbers, $number;
  $sum += $number;
  }

say "The sum of (@numbers) is $sum";
}
```

每次我们调用这个子程序的时候，它都会将新的参数与之前的参数相加，因此每次都会输出一个新的总和：

```
The sum of (5 6) is 11
The sum of (5 6 1 2 3) is 17
The sum of (5 6 1 2 3 4) is 21
```

但是在使用数组和哈希类型的 state 变量时，还是有一些轻微限制的。在 Perl 5.10 中我们不能在列表上下文中初始化这两种类型的 state 变量：

```
state @array = qw(a b c); # 错误!
```

这样声明会出错，也许在未来的 Perl 新版本中我们可以这样做：

目前，在列表上下文环境中初始化 state 变量是被禁止的。

习题

以下习题答案参见附录 A：

1. [12] 写一个名为 total 的子程序，它可以返回给定列表中数字相加的总和。(提示：该子程序不需要执行任何 I/O，它只需要按要求处理它的参数并给调用者返回一个值就行了。) 用下面这个程序来检验一下你写完的子程序，第一次调用时返回的列表中数字的总和应该是 25。

```
my @fred = qw{ 1 3 5 7 9 };
my $fred_total = total(@fred);
print "The total of \@fred is $fred_total.\n";
print "Enter some numbers on separate lines: ";
my $user_total = total(<STDIN>);
print "The total of those numbers is $user_total.\n";
```

2. [5] 使用前面这个程序中你写的子程序，计算出从 1 加到 1000 的总和。

3. [18] 额外附加题：写一个名为 &above_average 的子程序，当给定一个包含多个数字的列表时，返回其中大于这些数的平均值的数。(提示：另外写一个子程序，

通过用这些数的总和除以列表中数字的个数来计算它们的平均值。）当你写完后，用下面的程序来检验一下：

```
my @fred = above_average(1..10);
print "\@fred is @fred\n";
print "(Should be 6 7 8 9 10)\n";
my @barney = above_average(100, 1..10);
print "\@barney is @barney\n";
print "(Should be just 100)\n";
```

4. [10] 写一个名为 greet 的子程序，当给定一个人名作为参数时，打印出欢迎他的信息，并告诉他前一个来宾的名字：

```
greet( "Fred" );
greet( "Barney" );
```

按照表达式的顺序，它应该打印出：

```
Hi Fred! You are the first one here!
Hi Barney! Fred is also here!
```

5. [10] 修改前面这个程序，告诉所有新来的人，之前已经迎接了哪些人：

```
greet( "Fred" );
greet( "Barney" );
greet( "Wilma" );
greet( "Betty" );
```

按照表达式的顺序，它应该打印出：

```
Hi Fred! You are the first one here!
Hi Barney! I've seen: Fred
Hi Wilma! I've seen: Fred Barney
Hi Betty! I've seen: Fred Barney Wilma
```

第五章

输入与输出

先前为了习题的需要我们已经介绍过输入/输出（input/output，简称 I/O）的一些用法。现在则要涉及大多数情况下会遇到的大多数（约 80%）I/O 问题。对于标准输入流、输出流及错误流这几个概念，如果你都已经很熟悉了，那你目前算是处于超前状态。不熟的话，阅读完本章也就可以迎头赶上了。现在，就让我们先将“标准输入”当成“键盘”、将“标准输出”当成“屏幕”吧。

读取标准输入

读取标准输入流相当容易，我们在前面已经用过“行输入”<STDIN> 操作符【注 1】。在标量上下文中执行该操作时，将会返回标准输入中的下一行：

```
$line = <STDIN>;                # 读取下一行
chomp($line);                   # 截掉最后的换行符

chomp($line = <STDIN>);         # 习惯用法，效果同上
```

如果读到文件结尾（end-of-file），“行输入”操作符就会返回 undef。这样的设计是为了配合循环使用，可以自然地跳出循环：

```
while (defined($line = <STDIN>)) {
  print "I saw $line";
}
```

第一行程序代码做了许多事：读取标准输入、将它存入某个变量、检查变量的值是否被定义，以及我们是否该执行 while 循环的主体（也就是还没遇到输入的结尾）。因此

注 1：我们称 <STDIN> 为“行输入”操作，实际上它是针对文件句柄的“行输入”操作（用尖括号表示）。关于文件句柄，我们将在这章稍后的部分讨论。

在循环主体内，我们会在 $line 变量里看到各行输入的内容【注2】。这是十分常见的操作，所以 Perl 顺理成章为它定义了一个简写，如下所示：

```
while (<STDIN>) {
  print "I saw $_";
}
```

Larry 从一堆冷僻的符号中找到一对尖括号，创造出了这个简写。换句话说，从字面上来讲，它的意思是："读取一行标准输入，看它是不是为真（通常是）。若是真，就进入 while 循环，并在下次循环时忘记刚刚读入的那一行，进入下一行！" Larry 将这个无用的语法，赋予了有用的意义。

它实际上和前面的第一个循环相同：它告诉 Perl 将标准输入读进某个变量，（只要能读到内容，表示还没到达文件结尾）然后进入 while 循环。然而这次 Perl 并不是把它读入 $line 里，而是放进它的"老地方"变量 $_ 中，仿佛执行了下面的程序：

```
while (defined($_ = <STDIN>)) {
  print "I saw $_";
}
```

继续看下去之前，我们必须讲清楚一件事：这个简写只有在最早的写法中才能正常运行。如果你将"行输入"操作符放在其他任何地方（特别是自成一行的语句时），它并不会读一行输入，并自动存入默认变量 $_。只有当 while 循环的条件表达式里只有"行输入"操作符的前提下，这个简写才起作用【注3】。假如条件表达式里放了其他的东西，它就无法按你的预期运行了。

"行输入"操作符（<STDIN>）和 Perl 的"老地方"变量（$_）之间并没有什么关联，只是在这个简写里，输入的内容会恰好存储在 $_ 变量中而已。

然而，如果在列表上下文中调用"行输入"操作符，它会返回一个列表，其中包含（其余）所有的输入内容，每个列表的元素代表一行输入内容：

```
foreach (<STDIN>) {
  print "I saw $_";
}
```

再次强调，"行输入"操作符和 Perl 的"老地方"变量（$_）之间并没有任何关联，只

注2： 可能你注意到了在这里我们没有使用 chomp 函数。一般我们把该函数写在循环体内的第一行，而不是放在循环的条件表达式中。

注3： for 循环和 while 循环其实是一样的，所以这里的简写语法对 for 也可用。

是在这个例子里，foreach 的默认控制变量是 $_ 而已。在这个循环里，我们会在 $_ 中看到各行输入的内容。

这好像在哪里听过。没错，它和 while 循环的行为完全一样，不是吗？

两者不同之处在于它们背后的运作方式。在 while 循环里，Perl 会读取一行输入，把它存入某个变量并且执行循环的主体。接下来，它会回头去寻找其他的输入行。但是在 foreach 循环里，“行输入”操作符会在列表上下文中执行（因为 foreach 需要逐项处理列表的内容）。为此，在循环能够开始执行之前，它必须先将输入全部读进来。假如输入来自 400MB 大小的 Web 服务器日志文件，它们的差异会十分明显！因此最好的做法，通常是尽量使用 while 循环的简写，让它每次处理一行。

钻石操作符输入

还有另外一种读取输入的方法，就是使用钻石【注 4】操作符 <>。它能让程序在处理调用参数（稍后我们会看到）的时候，提供类似于标准 Unix 工具程序的功能【注 5】。如果想用 Perl 编写类似 *cat*、*sed*、*awk*、*sort*、*grep*、*lpr* 的工具，钻石操作符将会是你的好帮手。但若你想让它处理更复杂的参数格式，钻石操作符可能就帮不上忙了。

程序的调用参数（*invocation arguments*）通常是命令行上跟在程序名后面的几个“单词”【注 6】。在下面的例子里，命令行参数就是要依次处理的几个文件的名字：

```
$ ./my_program fred barney betty
```

这行命令的意思是：执行 *my_program* 命令（它可以在当前的目录中找到），然后它应该处理文件 *fred*，接着是文件 *barney*，最后是文件 *betty*。

若不提供任何调用参数，程序会处理标准输入流。但有个特例：如果你把连字符（-）当

注 4：钻石操作符是 Larry 的女儿 Heidi 命名的，某天当 Randal 拿着他新写的 Perl 培训材料去 Larry 家给他看的时候，这个操作符还没有一个叫得出的名字。Larry 也想不出来，八岁的 Heidi 灵机一动，说“它像钻石”，于是便有了这个名字，谢谢 Heidi!

注 5：不仅限于 Unix 系统。也许其他系统也采用了相同的方式来处理调用参数。

注 6：当程序开始运行的时候，它会有长度为零个或者多个的参数列表，这取决于运行它的程序。通常是由 shell 来启动这个程序，这时参数列表就取决于你在命令行上的输入。稍后我们会看到你能使用任意字符串来充当调用参数。因为他们一般出现在命令行上，所以我们称之为“命令行参数”。

作参数，它也代表标准输入【注 7】。所以，假如调用参数是 `fred - betty`，那么程序应该处理文件 *fred*，接着是标准输入流，最后才是文件 *betty*。

让程序以这种方式运行的好处，就是你可以在运行时指定程序的输入源。举例来说，你不需要重写程序，就可以在管道（pipeline）里使用它（稍后会有更深入的讨论）。Larry 为 Perl 加上这个功能，是想帮你轻松写出（使用起来）类似标准 Unix 工具的程序，甚至在非 Unix 的系统也行。事实上，Larry 是为了让他自己的程序使用起来类似标准的 Unix 工具，才设计出这个功能的。因为各厂商的工具程序在运行方式上不尽相同，所以 Larry 自己写出这些工具程序并将它安装在各种机器上，这样它们的运行方式就都会完全相同了。这当然也意味着他得把 Perl 移植到他能找到的每台机器上。

钻石操作符是“行输入”操作符的特例。不过它并不是从键盘取得输入，而是从用户指定的位置读取【注 8】：

```
while (defined($line = <>)) {
  chomp($line);
  print "It was $line that I saw!\n";
}
```

所以，假设这个程序运行时的调用参数是 `fred`、`barney` 和 `betty`，输出结果就会是“It was [从文件 *fred* 取得的一行内容]！”、“It was [从文件 *fred* 取得的另一行内容]！”等，直到遇到文件 *fred* 的结尾为止。接下来，它会自动切换到文件 `barney`，逐个输出它的内容，然后再换到文件 *betty*。请注意，在切换到另一个文件时，中间并没有间断，因为使用钻石操作符时，就好像这些文件已经合并成一个很大的文件。【注 9】钻石操作符只有在碰到所有输入的结尾时，才会返回 `undef`(然后我们就会跳出 `while` 循环)。

既然这只是“行输入”操作符的一种特例，因此我们可以使用先前看到的简写，将输入读取到默认的 `$_` 里：

```
while (<>) {
  chomp;
  print "It was $_ that I saw!\n";
}
```

注 7：可能你对 Unix 的一些东西还不熟悉：大多数标准的工具，像 *cat* 和 *sed*，它们都把连字符（-）当作标准输入流。

注 8：输入可能来自键盘，也可能来自其他设备。

注 9：无论对你是否有用，当前在处理的文件名都会保存在 Perl 的特殊变量 `$ARGV` 中。如果当前是从标准输入获取的数据，那么当前文件名会是 "-"（连字符）。

这和前面的循环效果相同，但是可以少打些字。你可能也注意到，我们使用了 chomp 的默认用法：不加参数时，chomp 会直接作用在 $_ 上。节约按键，从小地方做起!

由于钻石操作符通常会处理所有的输入，所以当它在程序里出现好几次时，通常是错误的。如果程序里同时出现两个钻石操作符，尤其是在使用第一个钻石操作符进行读取的 while 循环中接着使用第二个的话，它几乎是不可能正常工作的【注 10】。在我们的经验里，当初学者在程序中放入第二个钻石操作符时，往往是想读取下一行输入，其实 $_ 才是他们想要的东西。请切记，钻石操作符是用来读取输入的，而输入的内容可以在 $_ 中找到（起码是在一般的默认情况下）。

假如钻石操作符无法打开某个文件并读入内容，便会显示相关的出错诊断信息，就像：

```
不能打开 wilma：无此文件或目录 (can't open wimla: No such file or directory)
```

然后钻石操作符会自动跳到下一个文件，就像 *cat* 或其他标准工具程序的做法一样。

调用参数

技术上来说，钻石操作符其实不会去检查调用参数，它的参数其实是来自 @ARGV 数组。这个数组是由 Perl 解释器事先建立的特殊数组，其内容就是由调用参数组成的列表。换句话说，它和别的数组没有不同（除了奇怪的全大写名称之外），只不过在程序开始运行时，@ARGV 里就已经塞满了调用参数【注 11】。

你可以像使用其他数组一样来使用 @ARGV：你可以把元素从它里面 shift 出去，或是用 foreach 逐项加以处理。你也可以检查是否有参数是以连字符（-）开头的，然后将它们当成调用选项处理（就像 Perl 对待它自己的 -w 选项一样）【注 12】。

钻石操作符会查看数组 @ARGV 以决定该用哪些文件名，如果它找到的是空列表，就会改用标准输入流；否则，就会使用 @ARGV 里的文件列表。这表示程序开始运行之后，

注 10：如果你在使用第二个钻石操作符前，重新初始化了特殊变量 @ARGV，那么这样就是可行的。我们将在下一小节中了解变量 @ARGV。

注 11：C 程序员可能会以此联想到 argc（Perl 中没有这个），以及该程序的名字在哪（它在 Perl 的特殊变量 $0 里，不包含在 @ARGV 中）。另外，调用程序的方式不同，这里的情况也可能不同。具体请参看 *perlrun* 手册。

注 12：如果你需要的命令选项多于一两个，最好以标准的方式使用相关的模块。请参看 Getopt::Long 和 Getopt::Std 模块的文档，它们都是随 Perl 发行的标准模块。

只要尚未使用钻石操作符，你就可以对 @ARGV 动点手脚。这样我们就可以处理三个特定的文件，不管用户在命令行参数中指定了什么：

```
@ARGV = qw# larry moe curly #;  # 强制让钻石操作符读取这三个文件
while (<>) {
  chomp;
  print "It was $_ that I saw in some stooge-like file!\n";
}
```

输出到标准输出

print 操作符会读取一个列表里的所有值，并把每一项（当然是一个字符串）依次送到标准输出。它在每一项之前、之后与之间都不会再加上额外的字符【注 13】。要是想在每个元素之间加上空白，并在结尾加上换行符，你得这么做：

```
$name = "Larry Wall";
print "Hello there, $name, did you know that 3+4 is ", 3+4, "?\n";
```

当然，直接使用数组和使用数组内插，在打印效果上是不同的：

```
print @array;       # 把数组的元素打印出来
print "@array";     # 打印出一个字符串（数组元素内插的结果）
```

第一个 print 表达式会一个接一个打印出数组中所有的元素，元素之间不会有空格。而第二个则不同，它只打印一个字符串，也就是 @array 在双引号中内插形成的字符串，也就是数组所有元素的（空格分隔）字符串【注 14】。所以，如果 @array 的内容是 qw/ fred barney betty /【注15】，那么第一个表达式会输出 fredbarneybetty，而第二个会输出以空格隔开的 fred barney betty。

不过，在你打算总是使用第二种写法之前，请先想象 @array 包含了一串未截尾的输入行。也就是说，请想象里面的每个字符串都是以换行符结尾。这时，第一个 print 语句会分三行输出 fred、barney 和 betty。但是第二个 print 语句则会输出这样的结果：

```
fred
 barney
 betty
```

注 13：默认情况下它不会额外添加什么，但是这种默认情况（和 Perl 里很多其他的默认情况一样）是可以被修改的。对默认情况的修改会增加维护人员的困难，所以，除非是在短小精悍，需要快速完成功能的程序中，或者是在某一小段代码里，一般请不要修改这些默认情况。需要详细了解，请参看 *perlvar* 手册。

注 14：空格也是默认情况而已，请再次参看 *perlvar* 在线手册。

注 15：这是 Perl 的一种写法，它代表包含三个元素的列表，你知道的，对吧？

你看得出其中的空格是从哪里来的吗？因为 Perl 把数组内插到字符串中时，会在每个元素之间加上空格。因此我们得到的字符串将会是数组的第一个元素（fred 与换行符），接着是一个空格，然后是数组的下一个元素（barney 与换行符），接着是一个空格，然后是数组的最后一个元素（betty 与换行符）。结果就是，除了第一行，其他几行看起来好像经过缩排一样。每隔一两周，在新闻组或论坛中就会出现如下主题的新帖子：

Perl 会在第一行之后缩进剩下的行

我们甚至不必看正文，马上就知道他的程序用了双引号里的数组，而数组里全是没有截尾过的字符串。我们会问“你是不是在双引号里放了数组，而数组里面是没 chomp 过的字符串？”，而答案常常都是肯定的。

一般来说，如果数组里的字符串包含换行符号，那么只要直接将它们输出来就好了：

```
print @array;
```

要是它们不包含换行符号，你通常会想在结尾补上一个：

```
print "@array\n";
```

为了帮你分清楚哪个是哪个，通常在使用引号的场合，字符串后面最好都加上 \n。

一般情况下，程序的输出结果会先送到缓冲区。也就是说，不会每当有一点点输出就直接送出去，而是先积攒起来，直到数量够多时才造访外部设备。为什么要这样做呢？举例来说，当输出结果要保存到磁盘时，只为了添加一两个字符到文件结尾就去访问磁盘，（相对来讲）这样既慢又没效率。因此，输出的结果通常会先送到缓冲区，等到缓冲区满了或是在输出结束时（例如程序运行完毕），才会将它刷新到磁盘（也就是实际写到磁盘，或是写到其他地方）。通常这就是你想要的效果。

但是假如你（或程序）立刻就要输出，你也许愿意牺牲一些效率，在每次 print 时立刻刷新缓冲区。在这种情况下，请参阅 Perl 的在线手册，以进一步了解如何控制缓冲。

由于 print 处理的是待打印的字符串列表，因此它的参数会在列表上下文中执行。而钻石操作符（“行输入”操作符的特殊形式）在列表上下文中会返回由许多输入行组成的列表，所以它们彼此可以配合工作：

```
print <>;            # 和 Unix 下的 'cat' 命令功能差不多

print sort <>;       # 和 Unix 下的 'sort' 命令功能差不多
```

老实说，*cat* 和 *sort* 这两个标准的 Unix 命令还有许多额外的功能是上面两行程序代码做不到的。但它们绝对是物超所值！现在你可以用 Perl 重写所有的 Unix 工具程序，然

后轻松地将它们移植到任何有 Perl 的机器上，不管那台机器的操作系统是否为 Unix。在各种不同的机器上，你都可以保证程序会用相同的方式运行【注 16】。

有个比较不明显的问题，那就是 print 后面可有可无的括号，这可能会让人糊涂。别忘了有这条规则：除非这样做会改变表达式的意义，否则 Perl 里的括号可以省略。所以有两种方法可以输出一样的东西：

```
print("Hello, world!\n");
print "Hello, world!\n";
```

到目前为止还不错。不过 Perl 还有另一条规则：假如 print 的调用看起来像函数调用，它就是一个函数调用。这个规则很简单，但是“看起来像函数调用”是什么意思呢？

在函数调用里，函数名后面必须紧接着【注 17】一对括号，里面包含了函数的参数，如下所示：

```
print (2+3);
```

这看起来像函数调用，所以它的确是个函数调用。它会输出 5，然后和其他函数一样返回某个值。print 的返回值不是真就是假，代表 print 是否成功执行。除非发生了 I/O 错误，否则它一定会成功。所以在下面的语句里，$result 通常会是 1：

```
$result = print("hello world!\n");
```

可是，如果你用别的方式来处理这个结果，会发生什么事呢？假设你决定将返回值乘以 4：

```
print (2+3)*4;  # 糟糕!
```

当 Perl 看到这行程序代码，它会遵照你的要求输出 5。接着，Perl 会从 print 取得返回值 1，再将它乘以 4。然后它会丢掉这项乘积，因为你没告诉它接下来要做什么。这时，你旁边就会有人看到，然后说：“嘿，Perl 连数学都不会！应该输出 20，而不是 5！”

注 16：实际上 Perl Power Tools 项目（简称 PPT）的目标就是用 Perl 来重写所有经典 Unix 基础程序（甚至包括游戏！），但是在重写 shell 的时候陷入了难题。PPT 项目一度非常有用，因为它使得所有标准的基础程序能运行在很多非 Unix 机器上。

注 17：这里我们说“紧接着”是因为在调用函数的时候，Perl 不允许函数名和括号之间有换行符。如果有换行符，Perl 会把括号当成是创建列表的操作符，而不是函数调用的一部分。我们只是想让你知道这些琐碎的细节，如果你实在好奇，可以在手册里面寻找更详细的内容。

问题在于 Perl 可以省略括号，而大多数人又容易忘记括号的归属。没有括号的时候，print 是列表操作符，会把其后列表里的所有东西全都输出来。一般来说，这就是我们想要的。但是假如 print 后面紧跟着左括号，它就是一个函数调用，只会将括号内的东西输出来。因为该行程序代码有括号，所以对 Perl 来说，就和下面的写法一样：

```
( print(2+3) ) * 4;  # 糟糕!
```

好在只要你启用了警告信息，Perl 几乎都能帮你找出这类问题。所以请使用 -w 或加上 use warnings，至少在你开发程序与调试时打开它。

实际上，这条规则——“假如它看起来像函数调用，它就是函数调用”，不仅对 print 适用，也适用于 Perl 所有的列表函数【注 18】，只不过 print 可能最容易让人注意到这条规则。如果 print （或者其他的函数名）后面接着一个左括号，请务必确定在函数的所有参数之后也有相应的右括号。

使用 printf 格式化输出

处理输出结果时，你也许希望使用控制能力比 print 更强的操作符。事实上，你可能已经习惯使用 C 的 printf 函数来产生格式化过的输出结果。别担心，Perl 里同名的函数也能提供类似的功能。

printf 操作符的参数包括“格式字符串”及“要输出的数据列表”。格式化【注 19】字符串好像用来填空的模板，代表你想要的输出格式：

```
printf "Hello, %s; your password expires in %d days!\n",
  $user, $days_to_die;
```

格式字符串里可以有多个所谓的转换（*conversion*）。每种转换都会以百分比符号（%）开头，然后以某个字母结尾（稍后就会看到，在这两个符号之间可以存在额外的有效字符）。而后面的列表里元素的个数应该和转换的数目一样多，如果数目不对，就无法正确运行。上面的例子里有两个元素和两个转换格式，所以输出的结果看起来会像这样：

```
Hello, merlyn; your password expires in 3 days!
```

注 18：不需要参数或者只需要一个参数的函数不包括在内。

注 19：我们这里说的“格式”只是一般意义下的。Perl 本身还有个报表格式化的功能，目前还没有涉及到，它在附录 B，在这里我们将不会讨论它，免得你把它和这里的“格式”搞混淆。

printf 可用的转换格式很多，所以我们在这里只会说明最常用的部分。当然，你可以在 *perlfunc* 在线手册里找到详细的说明。

要输出恰当的数值形式，可以使用 %g【注 20】，它会按需要自动选择浮点数、整数甚至是指数形式：

```
printf "%g %g %g\n", 5/2, 51/17, 51 ** 17;  # 2.5 3 1.0683e+29
```

%d 格式代表十进制【注 21】的整数，它会舍去小数点之后的数字：

```
printf "in %d days!\n", 17.85;  # in 17 days!
```

请注意，它会无条件舍去，而非四舍五入。等一下我们就会学到如何四舍五入。

在 Perl 里，printf 最常用在字段式的数据输出上，因为大多数的转换格式都可以让你指定宽度。如果数据太长，字段会按需要自动扩展：

```
printf "%6d\n", 42;  # 输出结果看起来像 ____42 (_ 符号代表空格)
printf "%2d\n", 2e3 + 1.95;  # 2001
```

%s 代表字符串格式，所以它的功能其实就是字符串内插，只是它还能设定字段宽度：

```
printf "%10s\n", "wilma";  # 看起来像 _____wilma
```

如果宽度字段是负值，则会向左对齐（适用于上述各种转换）：

```
printf "%-15s\n", "flintstone";  # 看起来像 flintstone_____
```

%f 转换格式（浮点数）会按需要四舍五入，甚至还可以指定小点数之后的输出位数：

```
printf "%12f\n", 6 * 7 + 2/3;     # 看起来像 ___42.666667
printf "%12.3f\n", 6 * 7 + 2/3;   # 看起来像 ______42.667
printf "%12.0f\n", 6 * 7 + 2/3;   # 看起来像 ___________43
```

要输出真正的百分号，请使用 %%，它的特殊之处在于，不会输出（参数）列表中的任何元素【注 22】：

注 20：你可以把 g 当成“General”数字转换，或“Good conversion for this number”，或“Guess what I want the output to look like”。

注 21：如果你需要，还有 %x 代表十六进制，%o 代表八进制。你可以把 %d 当作是“decimal”，以帮助记忆。

注 22：也许你认为在百分号前加个反斜线就行了。不错的尝试，但是不对。这是因为这些格式符号是表达式，而表达式 "\%" 就代表字符串 '%'。所以就算我们加了反斜线，printf 还是不知道该怎么做。一般 C 程序员比较习惯这么做。

```
printf "Monthly interest rate: %.2f%%\n",
  5.25/12;  # 运算后的值看起来像 "0.44%"
```

数组和 printf

一般来说，你不会将数组当成 printf 的参数。这是因为数组可以包含任意数目的元素，而格式字符串却只会用到固定数目的元素：假如格式字符串里有三个转换格式，那就必须刚好有三个元素可供使用。

不过，没有人规定你不能在程序运行时动态产生格式字符串，它们可以是任意的表达式。但是要想做得正确需要一些技巧，把格式字符串存进变量可能会带来方便（尤其有助于调试）：

```
my @items = qw( wilma dino pebbles );
my $format = "The items are:\n" . ("%10s\n" x @items);
## print "the format is >>$format<<\n"; # 用于调试
printf $format, @items;
```

这段程序用了x 操作符(参见第二章)来复制你所指定的字符串，复制的次数与 @items（在标量上下文中使用）的个数相同。在上面的例子里，因为列表有三个元素，所以复制次数是 3 。这样一来，产生的格式字符串就和直接写“The items are:\n%10s\n%10s\n%10s\n”一样。程序会先输出标题，接着将每个元素显示成独立的一行，每行都靠右对齐，字段一律10个字符宽。很酷吧？这还不算酷，因为最酷的是我们可以把它们全都组合在一起：

```
printf "The items are:\n".("%10s\n" x @items), @items;
```

请注意，我们在标量上下文中用了一次 @items 以取得它的长度，然后又在列表上下文中用了它一次以取得它的内容。上下文的重要性可见一斑。

文件句柄

文件句柄（filehandle）就是程序里代表 Perl 进程（process）与外界之间的 I/O 联系的名字。也就是说，它是“这种联系”的名称，不一定是文件名。

文件句柄的命名如同 Perl 其他的标识符（以字母、数字及下划线组成，但不得以数字开头）。但是因为没有任何前置字符，所以当我们阅读它的时候，有可能会与现在或将来的保留字互相混淆，或是与第十章介绍到的标签(label)互相混淆。所以同样的，Larry 也建议你使用全大写字母来命名文件句柄。这样不仅看起来更加明显，也能避免与将要引入的（小写）保留字冲突，以至程序出错。

但是，有6个特殊文件句柄是 Perl 保留的，它们的名字是STDIN、STDOUT、STDERR、DATA、ARGV 和 ARGVOUT【注 23】。虽然你可以选择任何喜欢的文件句柄名，但不应使用保留的名字，除非确实需要以特殊方式使用上述六个句柄【注 24】。

上述的默认文件句柄，也许你早就认得了。当程序启动时，文件句柄 STDIN 就是 Perl 进程和它的输入源之间的联系，也就是俗称的标准输入流。它通常是用户的键盘输入，除非用户要求别的输入来源，像从文件读取输入或是经由管道（pipe）读取另一个程序的输出【注 25】。当然还有标准输出流 STDOUT。默认情况下它会输出到用户的屏幕，但用户也可以把它送到文件或是另一个程序，我们稍后会看到这种范例。这些标准的流乃是源自于 Unix 的“标准 I/O”函数库，不过大部分现代的操作系统也都支持【注 26】。一般来说，程序应该盲目地从 STDIN 读取数据、盲目地将数据写到 STDOUT，相信用户（或者广义地说，是启动你程序的那个程序）已经将它们都设定好了。比如说，用户可以在 shell 里运行以下命令：

```
$ ./your_program <dino >wilma
```

这条命令告诉 shell，程序的输入应该来自文件 *dino*，输出应该送到文件 wilma。这个程序只要盲目地从 STDIN 读进它的输入、（按照我们的意思）处理这些输入数据，之后再盲目地把输出送到 STDOUT，那么这一切就 OK 了。

这样不需要额外的工作，这个程序就可以正确地在管道（*pipeline*）中运行。这又是另一个来自 Unix 的概念，它可以让我们将命令写成这样：

```
$ cat fred barney | sort | ./your_program | grep something | lpr
```

假如你不熟悉这些 Unix 命令，那也没有关系。上面这条命令的意思是由 *cat* 命令将文件 *fred* 的每一行输出，再加上文件 *barney* 里的每一行。之后，将以上输出作为 *sort* 命

注 23：也许有些人讨厌用大写字母，哪怕一会儿，所以他们可能喜欢用小写字母来写这些名字，比如 stdin。Perl 也许能让你这么做，但是不保证你总这么做是不出错的。至于什么时候这么做可以什么时候不行，已经超出了本书的范围。但重要的是，如果你老用小写，说不定什么时候就出错了，所以这些句柄还是别用小写吧。

注 24：在某些情况下，你可以重用这些名称。但是维护你程序的人可能会错误地认为你在使用这些内置的保留文件句柄，他会很容易混淆的。

注 25：我们这里所说的那三个默认 I/O 流是针对 Unix shell，而不是指启动某个程序的 shell。我们会在第十六章了解用 Perl 启动其他程序的时候会发生什么。

注 26：如果你对非 Unix 系统的标准输入和输出不熟悉，请参看 *perlport* 和你的系统手册中讲解 Unix shell（或类似的程序，它们能根据你键入的命令调用其他程序）的章节。

令的输入，*sort* 会将所有数据行排序，再传给 *your_program*。当它完成处理之后，*your_program* 会将输出的数据送到 *grep*，这个命令会过滤掉数据中的某些行，并将剩下的数据输出到 *lpr* 这个命令，而 *lpr* 则应该把所有输入数据用打印机打出来。真酷！

在 Unix 以及许多其他现代的操作系统里，上述的管道用法相当常见。这样一来，你只要用好几个简单、标准的组件，就能构造出功能强大且复杂的命令。每个组件都能把一件事情做的很好，至于如何巧妙地组合起来就是你的任务了。

除了 STDIN 和 STDOUT，还有另一个标准 I/O 流。如果（在上一个例子中）*your_program* 发出任何警告或是其他诊断信息，这些信息就不应该继续在管道中往下传递。我们已经用 *grep* 命令来筛除特定字符串以外的任何数据，因此它很可能会丢弃警告信息。而且即使它留下了警告信息，我们可能也不想让它传递到管道下游的其他程序。这就是为什么我们还有一个标准错误流：STDERR。即使输出会被重定向到下一个程序或文件，错误信息仍能流向用户需要之处。错误信息在默认情况下通常是输出到用户的屏幕【注27】，但用户还是可以用如下的 shell 命令将错误信息转向某个文件：

```
$ netstat | ./your_program 2>/tmp/my_errors
```

打开文件句柄

Perl 提供的三种文件句柄 STDIN、STDOUT 和 STDERR，都是由产生 Perl 进程的父进程（可能是 shell）自动打开的文件或设备。当你需要其他的文件句柄时，请使用 open 操作符告诉 Perl，要求操作系统为你的程序和外界架起一道桥梁。这里有一些例子：

```
open CONFIG, "dino";
open CONFIG, "<dino";
open BEDROCK, ">fred";
open LOG, ">>logfile";
```

第一行会打开名为 CONFIG 的文件句柄，让它指向文件 *dino*。换句话说，它会打开（已存在的）文件 *dino*，文件中的任何内容，都能从文件句柄 CONFIG 读到我们的程序中来。这和利用 shell 的重定向（例如 <dino 这样的写法）将文件内容经 STDIN 读入程

注 27：另外，错误信息在一般情况下是不会被缓冲的。标准错误流和标准输出流会指向同一个地方（比如显示器），而错误信息可能在正常输出之前出现。举例来说，如果你的程序打印一行常规的文本，然后试图除以零，这个时候除以零相关的错误信息就可能比常规文本先输出到屏幕。

序里的做法相似。事实上，第二行正好用到了这样的技巧，它和第一行所做的事完全相同，只不过用了小于号来声明“此文件只是用来输入的”，而这正是默认的操作【注28】。

尽管你无须使用小于号来打开一个文件以供输入，但是我们还是提到了它，这是因为在例子的第三行有个与之相对的大于号，它可以用来创建一个新的文件：它会打开文件句柄 BEDROCK 并输出到新文件 fred。大于号的用途跟 shell 的重定向一样，它会将输出送到一个名为 fred 的新文件。如果已经存在一个名为 fred 的文件，那么就清除原有的内容并且以新内容取代之。

如例子的第四行所示，你可以使用两个大于号指明以追加的方式来打开文件（这仍然和 shell 的重定向方式相同）。换句话说，如果文件原本就存在，那么新的数据将会添加在原有文件内容的后面；如果它不存在，就会创建一个新文件，和只有一个大于符号的情形类似。此特性对于日志文件（log file）来说非常方便，你的程序可以在每次运行时只加几行数据到日志文件里。这就是为什么第四个例子的文件句柄称为 LOG，而文件名叫做 *logfile*。

你可以使用任何的标量表达式来代替文件名说明符。不过，你通常会想要明确指定输入或输出的方向：

```
my $selected_output = "my_output";
open LOG, "> $selected_output";
```

注意大于号后的空格。Perl 会忽略它【注 29】，但这个空格能防止意外发生。如果 $selected_output 的值是 ">passwd" 而之前又没有空格的话，就会变成以替换方式写入，而非以追加方式写入文件。

在相对新的 Perl（5.6 版之后）中，open 另有一种使用三个参数的写法：

```
open CONFIG, "<", "dino";
open BEDROCK, ">", $file_name;
open LOG, ">>", &logfile_name();
```

注 28：为了安全这么做很重要。之后（更深入的介绍来自第十四章）我们将看到在文件名中可以被用到的几个魔法字符。假如 $name 中是用户选择的文件，直接使用 $name 打开文件会使那几个魔法字符的功能都可作用于该文件。这对用户来说可能是一种方便，但是也可能是个安全漏洞。反之，用 "<$name" 来打开这个文件就安全多了，因为它明确指示了按只读方式打开文件。当然，这也无法避免其他可能的错误。如果你想了解更多打开文件相关的细节，尤其是在特别关注安全的时候，请参看 *perlopentut* 手册。

注 29：是的，这意味着如果你的文件名以空格开头，Perl 同样会忽略掉这些空格。如果你关心这个，请参看 *perlfunc* 和 *perl*open*tut*。

其优点在于语法上可以很容易的区分模式串（第二个参数）与文件名本身（第三个参数），这种写法在安全方面有些好处【注30】。不过，如果你的程序必须与旧版的 Perl 兼容（比如计划发布此程序到 CPAN），就不能使用此语法，或者在程序代码里注明“只与新版 Perl 兼容”【注31】。

我们将会在本章稍后看到这些文件句柄的用法。

不正确的文件句柄

Perl 和其他的程序语言一样，它自身无法打开文件，只能要求操作系统帮它打开文件。当然，操作系统可能会以权限不足、文件名错误等理由拒绝打开。

如果你尝试从不正确的文件句柄（即没有正确打开的文件句柄）读取数据，将会立刻读到文件结尾（对于你将会在本章看到的各种 I/O 方法而言，“文件结尾”在标量上下文中是 undef，在列表上下文中则是空列表）。如果试图将数据写进不正确的文件句柄，这些数据将会无声无息地被丢弃。

幸好，这种可怕的情况能轻易避免。首先，如果我们一开始就用 -w 选项或者 warning 编译命令来启用警告信息，那么在用到不正确的文件句柄时，Perl 通常会发出警告。但即使没有打开警告，open 的返回值也能告诉我们它的执行成功与否：返回真表示成功，返回假则表示失败。所以，程序可以写成这样：

```
my $success = open LOG, ">>logfile";  # 捕获返回值
if ( ! $success) {
  # open 操作失败
  ...
}
```

虽然你可以这么做，但是之后还会看到其他的方法。

关闭文件句柄

当你不再需要某个文件句柄时，可以用 close 操作符来关闭它：

```
close BEDROCK;
```

注30：它在安全上也有缺点。这么做能让某个恶意用户把可能的恶意字符注入无辜的善良程序中。一旦我们学了正则表达式（第七章），你就可以用它来检查过滤用户的输入。假如你的程序可能被恶意用户使用，请阅读 Perl 的羊驼书中关于 Perl 的安全特性的章节或者 *perlsec* 手册。

注31：举例来说，你可以使用 use 5.6 来标示。

所谓“关闭文件句柄”，也就是让 Perl 去通知操作系统，我们对该数据流的处理已经全都完成了。所以，请系统将尚未写入的输出数据写到磁盘，以免有人等着使用它【注 32】。当你重新打开某个文件句柄时（也就是说，在新的 open 命令中重用之前的文件句柄名），Perl 会自动关闭原先的文件句柄。在程序结束时，Perl 也会自动关闭文件句柄【注 33】。

正因为 Perl 这么贴心，所以很多简单的小程序都不必操心关闭的问题。但是若你想要写得工整些，请为每个 open 搭配一个 close。通常，最好是在每个文件句柄用完之后就立刻关闭它，哪怕程序马上就结束了【注 34】。

用 die 处理严重错误

让我们先稍稍岔开话题。目前需要一种不直接相关于（或者说不只用于）I/O 的机制，用于在异常发生时提前退出程序。

当 Perl 遇到严重错误（fatal error）时（例如：除以零、使用不合法的正则表达式，或调用未定义的子程序），你的程序应该中止运行，并用错误信息告知原因【注 35】。这样的功能可以用 die 函数来实现，它让我们能够自定义“严重错误”信息。

注 32：假如你熟悉 I/O 系统，你应该知道这里还有很多内容。一般来说，当文件句柄关闭的时候情况会是这样的：如果仍有文件输入，它会被忽略；如果仍有管道输入，那么对这个管道进行写操作的程序会收到管道被关闭的信号；如果有输出到文件或者管道，那么缓冲区会被刷新（也就是缓冲区的内容会马上发送出去）；如果文件句柄被加了锁，那么锁会被释放。更多的细节，请参看你的系统文档中关于 I/O 的章节。

注 33：退出程序会关闭所有的文件句柄，但是如果 Perl 解释器本身出错了，那么缓冲区的内容不会被刷新。也就是说，如果你的程序由于除以零而意外崩溃了，Perl 解释器仍会在运行，它仍能保证你的数据及时地输出，但是如果 Perl 解释器本身无法运行了（比如内存不足或者收到了意外信号），那么最后的一点数据可能就无法及时得到处理（输出或写到磁盘）。通常这不是个大问题。

注 34：关闭文件句柄会刷新输出缓冲，并释放该文件上的任何锁。因为可能有人等着用这些文件，所以需要长时间运行的程序最好尽可能快地关闭它打开的文件句柄。但是我们很多的程序也许只需要一两秒就运行完了，所以这种情况下你不关闭也没问题。关闭文件句柄同样也会释放可能的有限资源，所以它不仅仅只是为了程序看起来工整而已。

注 35：这是默认情况，但是错误可能被 eval 块捕获，关于这个，我们会在第十七章进一步了解。

die 函数会输出你指定的信息（到为这类信息预留的标准错误流中），并且让你的程序立刻终止，并返回不为零的退出码。

你可能还不知道，每个在 Unix（以及许多其他现代操作系统）上运行的程序都会有一个退出状态（exit status），用来告诉我们程序的运行是否成功。那些以调用其他程序为工作的程序（像工具程序 *make*），会查看那些程序的结束状态来判断是否一切顺利。所谓的“结束状态”其实只用一个字节来表示，所以它能传递的信息不多。传统上，零代表成功，非零代表失败。也许“1”代表命令参数中的语法错误，“2”代表处理某程序时发生了错误，而“3”则可能代表找不到某个配置文件，各个程序的细节不尽相同。但是，“0”一定代表程序顺利完成。像 *make* 这样的程序，在看到失败的结束状态时，就不会再执行下一步了。

所以，我们可以将前面的例子改写成这样：

```
if ( ! open LOG, ">>logfile") {
  die "Cannot create logfile: $!";
}
```

如果 open 失败，die 会终止程序的运行，并且告诉我们无法创建日志文件。可是冒号后面的 $! 代表什么呢？那就是可读的系统出错信息。一般来说，当系统拒绝我们所请求的服务（像打开某个文件）时，$! 会给我们一个理由。在这个例子里，也许是“权限不足”（permission denied）或“文件找不到”（file not found）之类的，也就是你在 C 或是其他类似的语言里调用 perror 取得的字符串。这个解释性的系统错误信息就保存在 Perl 的特殊变量 $! 中【注 36】。因此，将 $! 放到错误提示中，帮助用户了解自己遇到了什么问题，这是个不错的主意。不过，倘若你用 die 函数来显示程序中不属于系统服务请求的错误，这种时候请不要在信息里加上 $!，因为这时它保存的可能只是 Perl 底层操作导致的无关信息。只有在系统服务请求失败后的瞬间，$! 的值才会有用。如果操作成功了，就不会在 $! 里留下任何有用的信息。

die 还会帮你做一件事。它会自动将 Perl 程序名和行号【注 37】附加在错误信息的后面，因此你就可以轻易判断出程序里的哪个 die 函数才是造成过早结束的原因。在上一个例子里，可能会看到如下的错误信息(假设 $! 的内容是 permission denied)：

注 36：在一些非 Unix 的操作系统中，$! 可能包含类似 error number 7 的信息，让用户去相关的文档中查看到底是哪里出错了。在 Windows 和 VMS 系统中，特殊变量 $^E 可能会包含一些附加的诊断信息。

注 37：如果在读取文件块的时候发生了错误，那么出错信息还会包含“块编号”（一般是行号）和文件句柄名，这些信息对于跟踪 bug 很有用。

```
Cannot create logfile: permission denied at your_program line 1234.
```

这非常有用！事实上，每当我们想要了解错误信息的内幕，总会需要程序行号。如果你不想显示行号和文件名，请在 die 函数的信息尾端加上换行符。也就是说，die 的另一种用法就是加上结尾的换行符，形式如下所示：

```
if (@ARGV < 2) {
  die "Not enough arguments\n";
}
```

如果命令行参数不足两个，范例程序会显示这行信息并中止运行。因为行号在此对用户并没有用处（毕竟这是用户导致的错误），所以这里并不会显示行号和程序的文件名。一个建议就是，用来指示用法错误的信息里可以加上结尾的换行符，但是若想在调试过程中追踪相关的错误，就不要加上结尾的换行符【注 38】。

请一定记得检查 open 的返回值，因为之后的程序代码必须在文件打开成功时才能顺利运行。

使用 warn 输出警告信息

die 函数可以用来模拟 Perl 的内置错误（比如除以零）。相似的，warn 函数的功能就是模拟 Perl 的内置警告（比如启用警告信息时，使用某个 undef 变量，却把它当成定义值来用，就会导致内置警告）。

warn 函数的功能就和 die 函数差不多，不同之处在于最后一步，它不会终止程序的运行。如有需要，它也可以加入程序名与行号，而且它会将信息送到标准错误流，这点是和 die 函数一样的【注 39】。

在讨论过 die 以及 warn 之后，现在要回到有关 I/O 的事情上来。请继续阅读。

注 38：程序名保存在 Perl 的特殊变量 $0 中，你可以把它加在这行中：$0:Not enough arguments\n，当你的程序在管道或者 shell 脚本中被调用的时候，这样做会非常有用。但是，在程序执行的时候，$0 还是可以被改变的。另外你可能还想了解一下 __FILE__ 和 __LINE__ 这两个特殊记号（或者 caller 函数），这样你就可以自己写一行语句来输出符合你格式要求的信息。

注 39：和严重错误信息不同，警告信息是无法被 eval 块捕获的。但是如果你想捕获警告信息的话，请参看 *perlvar* 手册中关于伪信号 __WARN__ 的章节。

使用文件句柄

当文件句柄以读取模式打开后，你可以轻易地从它读取一行数据，就像从 STDIN 读取标准输入流一样。因此，以“从 Unix 系统读取密码文件”为例：

```
if ( ! open PASSWD, "/etc/passwd") {
  die "How did you get logged in? ($!)";
}

while (<PASSWD>) {
  chomp;
  ...
}
```

在这个例子里，die 的消息中用了一对括号围住 $!。它们只不过是括住输出信息的括号而已（有时候标点符号就只是标点符号）。正如你看到的，所谓的“行输入”操作符是由两部分组成的：一对尖括号（真正的“行输入”操作符）以及里面用来输入的文件句柄。

以写入或是添加模式打开的文件句柄可以在 print 或 printf 函数中使用。使用时，请直接将它放在关键字之后、参数列表之前：

```
print LOG "Captain's log, stardate 3.14159\n";  # 输出到文件句柄 LOG
printf STDERR "%d percent complete.\n", $done/$total * 100;
```

注意到文件句柄和要输出的内容之间没有逗号了吗【注 40】？有括号时，它看起来会更奇怪。下面两种写法也都没错：

```
printf (STDERR "%d percent complete.\n", $done/$total * 100);
printf STDERR ("%d percent complete.\n", $done/$total * 100);
```

改变默认的文件输出句柄

默认情况下，假如你不为 print（或是 printf，我们下面的说明对两者同样有效）指定文件句柄，它的输出就会送到 STDOUT。不过，你可以使用 select 操作符来改变默认的文件句柄。在下面的例子里，我们会将输出送到 BEDROCK 这个文件句柄：

```
select BEDROCK;
print "I hope Mr. Slate doesn't find out about this.\n";
print "Wilma!\n";
```

注 40：如果你对英语或者语言学很在行，当我们说这种情况叫做“间接对象语法”的时候，你可能会说：“哦！当然拉！句柄名后面没有跟逗号——所以它是间接对象！”但是其实我们并不懂为什么这里就不用逗号，之所以省略逗号是因为 Larry 说这个逗号是可以省略的。

一旦切换（select）了默认输出的文件句柄，程序就会一直往那里输出。但是，这么做很容易使余下的程序变得混淆，所以这并不是一个好办法。因此，当你所指定的默认文件句柄使用完毕之后，最好把它设回原先的默认值 STDOUT 来【注 41】。将数据输出到文件句柄时，默认情况下都会经过缓冲的处理。不过，只要将特殊变量 $| 设定为 1，就会使当前的（也就是修改变量时所指定的）默认文件句柄在每次进行输出操作后，立刻刷新缓冲区。所以，如果要让输出的内容（比如某个后台程序的实时日志）立即显示，你就可以这么做：

```
select LOG;
$| = 1;  # 不要将 LOG 的内容保留在缓冲区
select STDOUT;
# …… 时间流逝，婴儿都学会走路了，斗转星移之后 ……
print LOG "This gets written to the LOG at once!\n";
```

复用标准文件句柄

我们在前面提到过，如果你要复用某个文件句柄（换句话说，如果已经打开了某个名为 FRED 的文件句柄，而现在又要打开同样名为 FRED 的文件句柄），Perl 会自动帮你关闭原有的文件句柄。我们也提到过，你不应该复用 Perl 的 6 个标准文件句柄，除非你想使用该文件句柄实现特殊功能。我们还说过，来自 die 和 warn 的信息以及 Perl 内部的出错信息都会自动送到 STDERR。如果以上三个知识能融会贯通，你就会意识到，错误信息不一定都要送到程序的标准错误输出流，也可以送到文件里：【注 42】

```
# 将错误信息写到我自定的错误日志中
if ( ! open STDERR, ">>/home/barney/.error_log") {
  die "Can't open error log for append: $!";
}
```

在复用了 STDERR 之后，任何从 Perl 产生的错误信息都会送到新的文件里。但如果程

注 41：在某些情况下可能想切换的句柄不是 STDOUT，你可以在 *perlfunc* 手册中查看关于 select 的内容来了解如何保存和恢复当前文件句柄。另外，实际上在 *perlfunc* 文档中你能找到两个 Perl 内置的 select 函数，他们虽然名字一样，但是另外一个 select 在调用的时候有 4 个参数。

注 42：如果你没有理由的话不要这样做。最好是让用户自己在运行你程序的时候来决定什么信息该重定向到哪里，而不是由你把它硬性规定在程序里。但是当你的程序是会被其他程序自动运行的（比如说被 web 服务器或者 cron 或 at 这些定时程序）时候，又或者你的程序要启动其他的进程（比如用我们将在第十六章学到的 system 或者 exec 函数）并且你需要这个进程有不同的 I/O 时，这样做会比较灵活。

序执行到 die 这部分代码，那会怎样呢？也就是说，如果无法成功打开文件来接收错误信息，那么错误信息会流到哪里去？

答案是：在复用这三个系统文件句柄 STDIN、STDOUT 或 STDERR 失败时，Perl 会热心地帮你找回原先的文件句柄【注 43】。也就是说，Perl 只有在成功打开新的句柄连接时，才会关闭默认的系统文件句柄。所以程序中可以用这个技巧，对三个系统文件句柄中任何一个（或全部）进行重定向【注 44】，这跟“程序从命令行运行时由 shell 进行 I/O 重定向”的功能是一样的。

使用 say 来输出

Perl 5.10 从正在开发的 Perl 6 中借来了 say 这个函数(Perl 6 中的 say 函数可能是借鉴了 Pascal 的 println 函数)。它的功能和 print 函数差不多，但是会在每行输出的结尾自动加上换行符，下面的方式输出的结果一样：

```
use 5.010;

print "Hello!\n";
print "Hello!", "\n";
say "Hello!";
```

如果你想输出某个变量的值，并在它的末尾带上换行，其实不必额外构造一个字符串，也不必用 print 函数打印列表。直接 say 这个变量就可以了。在你想输出某些内容并换行的时候，这个函数非常好用：

```
use 5.010;

my $name = 'fred';
print "$name\n";
print $name, "\n";
say $name;
```

在内插数组的时候，你仍然需要将它用引号括起来，这样它能把数组的元素用空格分开：

注 43：至少如果你没修改特殊变量 $^F 的情况下，Perl 会这么做。这个特殊变量告诉 Perl 在复用这三个句柄失败的时候恢复它们的默认值，最好不要修改该变量。

注 44：别把 STDIN 修改成用来输出的文件句柄，或者把另外两个标准输出句柄改成用来输入的句柄。这种把它们搞混淆的事情，想想都头痛。

```
use 5.010;

my @array = qw( a b c d );

say @array;    # 输出 "abcd\n"

say "@array"; # 输出 "a b c d\n";
```

和 print 函数一样，你可以为 say 指定一个文件句柄：

```
use 5.010;

say BEDROCK "Hello!";
```

因为这个是 Perl 5.10 的新特性，所以我们只能在打开 Perl 5.10 新特性的情况下使用它。虽然老的 print 仍然和以前一样工作得挺好，但是也许有的 Perl 程序员会马上换用这个函数，因为这样他们可以省掉4次按键（print 比 say 多两个字符，再加上 \n 两个字符）

习题

以下习题答案参见附录 A：

1. [7] 写一个功能跟 *cat* 相似的程序，但是将各行的内容反序后输出（某些系统会有名为 *tac* 的类似工具）。假如用 ./tac fred barney betty 来运行你的程序，它的输出结果应该是 *betty* 文件的最后一行到第一行，接着是文件 *barney* 与 *fred*，同样是由最后一行到第一行。（如果你也将程序取名为 *tac*，请一定要在运行时加上 ./，这样才不会运行你系统中的同名程序！）

2. [8] 写一个程序，要求用户分行键入各个字符串，然后以 20 个字符宽、向右对齐的方式输出每个字符串。为了确定输出结果在适当的字段中，请一并输出由数字组成的“标尺行”（rule line）（这只是为了方便调试）。请验证自己没有误用 19 个字符的宽度！举例来说，输入 hello、good-bye 后应该会得到如下的输出结果：

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
                   hello
              good-bye
```

3. [8] 修改上一个程序，让用户自行选择字符宽度，因此在键入 30 的时候，hello、good-bye（在不同行上）应该会向右对齐到第 30 个字符（提示：关于如何控制变量的内插，请参阅第二章的“字符串里的标量变量内插”一节）。附加题：参考用户键入的宽度，自动调整标尺行的宽度。

第六章

哈希

在这一章里面，我们将会看到使 Perl 立足于杰出编程语言的著名特色：哈希【注 1】。尽管哈希是如此强劲有力的特性，但那些多年使用其他语言的人却可能从未听说过。好在一旦听说，几乎每个新的 Perl 程序中就会用到哈希，它就是如此重要。

什么是哈希？

所谓哈希就是一种数据结构，和数组的相同之处在于：可以容纳很多值（没有上限），并能随机存取。而区别在于：不像数组是以数字来检索，哈希是以名字来检索。也就是说检索用的键不是数字，而是保证唯一的字符串。参见图 6-1。

所谓键其实就是字符串，所以我们不必用数字 3 来获取数组元素，而是用 `wilma` 这个名字来存取哈希元素。

这些键可以是任何字符串——你可以用任意字符串表达式作为哈希键。它们也必须是唯一的字符串，就像数组只能有一个编号为 3 的元素一样，哈希也只能有一个名叫 `wilma` 的元素。

注 1：以前我们叫它关联数组（associative array）。但是 1995 年左右，Perl 社区认为这个名字太罗嗦、写起来也麻烦，于是我们把名字改成了哈希（hash）。

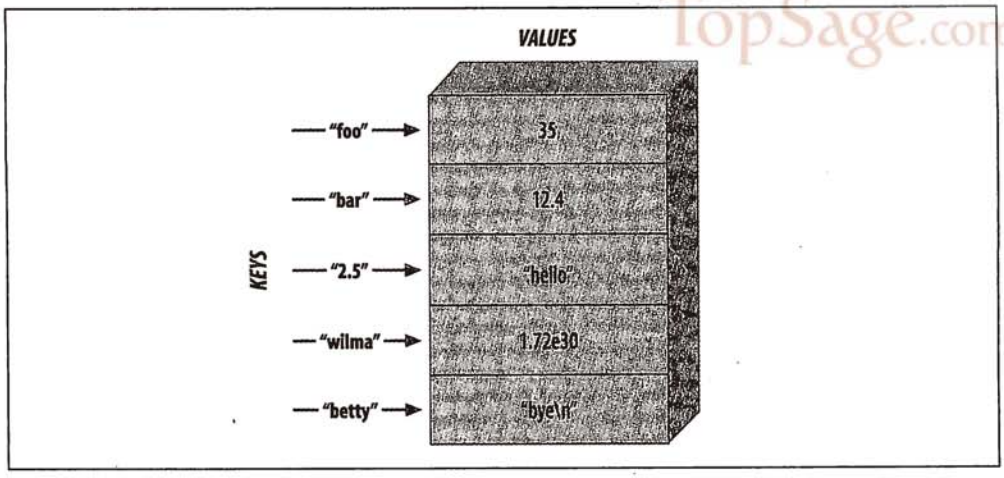

图 6-1：哈希的键与值

另一种看待哈希的方法，是将它想象成一大桶的数据，其中每个数据都有关联的标签，参看图 6-2。可以伸手到桶里任意取出一张标签，查看它附着的数据是什么。但是桶里没有所谓的第一个元素，只有一堆数据。然而在数组里，第一个元素为元素 0，然后是元素 1、2，等等。但是在哈希里没有顺序，因此也没有第一，有的只是一些键／值对。

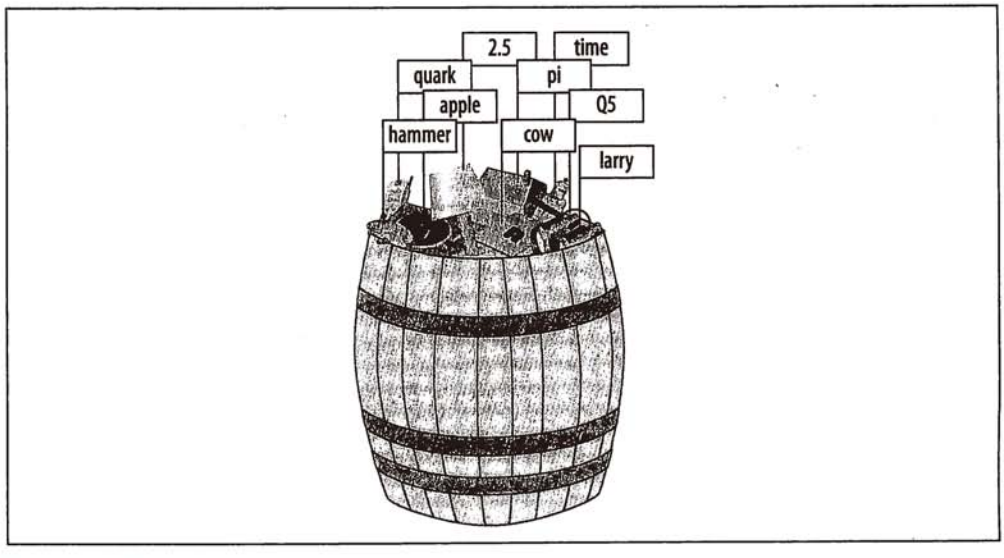

图 6-2：哈希像一桶数据

这些键和值都是任意的标量，但是键总是会被转换成字符串。假如你以数字表达式 50/20 为键【注 2】，那么它会被转换成一个含有三个字符的字符串 "2.5"，恰好就是刚才图中的一个键。

根据 Perl“去除不必要限制”的原则，哈希可能是任意大小，从没有任何键／值对的空哈希到填满内存的巨大哈希都可以。

某些语言的哈希实现在键／值对增多时会逐渐变慢，例如 *awk* 语言就是如此。Larry 正是从这种语言中引入了哈希。但是 Perl 的版本并没有这个问题，它有良好、高效、可伸缩的算法【注 3】。因此不论哈希只有三个键／值对，还是三百万个键／值对，从其中取出一个都应该一样快捷。大哈希没什么好恐惧的。

值得一提的是：虽然这些键是唯一的，但是它们对应的值可以重复。哈希的值可以是数字、字符串、undef，或是这些类型的组合【注 4】。但是哈希的键则全都必须是唯一的字符串。

为什么使用哈希？

当第一次听到哈希的时候，尤其是在多年从事其他语言的开发人员看来，没有哈希这个怪兽好像也很正常。其实这个创意的根源就是需要将一组数据对应到另一组数据。例如下面这些典型的 Perl 哈希应用：

按名字找姓

名字可以作为键，而姓可以成为值。这当然需要限定名字是唯一的，如果出现了两个叫做 randal 的人就行不通了。通过哈希可以按任何人的名字找到相应的姓。例如以 tom 为键取得值 phoenix。

用主机名找 IP 地址

你也许知道在因特网上，每台计算机同时拥有一个主机名如 *www.stonehenge.com*，以及一个 IP 地址如 123.45.67.89。这是因为机器喜欢和数字打交道，但是一般人比

注 2：这是一个数字表达式，而不是 5 个字母的字符串 "50/20"。当然如果我们使用了该字符串作为哈希键，那它就会保持不变。

注 3：技术上来说，Perl 会在需要时重建大型哈希表。其实之所以用哈希这个术语就是因为实现的数据类型就是哈希表。

注 4：事实上，任何标量值都可以，也包括目前还没提到的其他标量类型。

较记得住名字。主机名是唯一的字符串，可以用来作为哈希的键。通过哈希可以按主机名找到相应的 IP 地址。

按 IP 地址找主机名

当然，你也可以反其道而行。我们一般把 IP 地址当成数字，但它们也是唯一的字符串，因此可以成为哈希键。在这个哈希中，我们可以查询 IP 地址以决定相应的主机名。请注意，这个哈希并不等同于上面例子的哈希：哈希是从键到值的单行道，我们无法在哈希中查询值并反推出相应的键！所以这两个例子恰好用到了相反的哈希，一个存放 IP 地址，另一个存放主机名。然而有了其中一个哈希，要生成另一个哈希是相当容易的，稍后可以看到。

按单词统计出现次数

这是个极为常见的哈希应用，因为相当常见，所以可能会在末尾的习题里面出现。这个问题是计算某个文件里每一个单词出现的频率。也许你正在为一堆文件编写索引，然后当用户检索 `fred` 的时候，就能知道这个词在甲文件中出现了 5 次，在乙文件中出现了7次，而在丙文件中则从未出现。有了这个索引我们就知道哪个文件可能是用户最感兴趣的。在索引构造程序扫描每份指定的文件时，对于每个找到的 `fred`，就给 `fred` 这个键对应的值加 1。也就是说，假如 `fred` 之前在文件中已经出现过两次，它的值就是 `2`，现在它会被加到 `3`。假如我们以前未曾见过 `fred`，该值就会从默认的 `undef` 加到 `1`。

按用户名统计每个人使用（或者浪费）的磁盘块数量

系统管理员会喜欢这个例子：系统中的用户名是唯一的字符串，因此可以被当成哈希的键来检索用户相关的信息。

按驾驶执照号码找出姓名

也许有许多人都叫做 John Smith，但是起码他们应该有不同的驾照号码。因此唯一的驾照号码可以成为键，而人名可以作为值。

另一种思考方式是将哈希当成极其简单的数据库，其中每个键的“名下”都可以存储相应的数据。事实上只要问题中带有找出重复、唯一、交叉引用、查表之类的字眼，就很有可能会用到哈希。

访问哈希元素

要访问哈希元素，需要使用如下语法：

```
$hash{$some_key}
```

这和访问数组的做法类似，只是使用了花括号而非方括号来引出索引【注5】。而且现在的键表达式是字符串，而非数字：

```
$family_name{"fred"} = "flintstone";
$family_name{"barney"} = "rubble";
```

图 6-3 显示了如何对哈希键赋值。

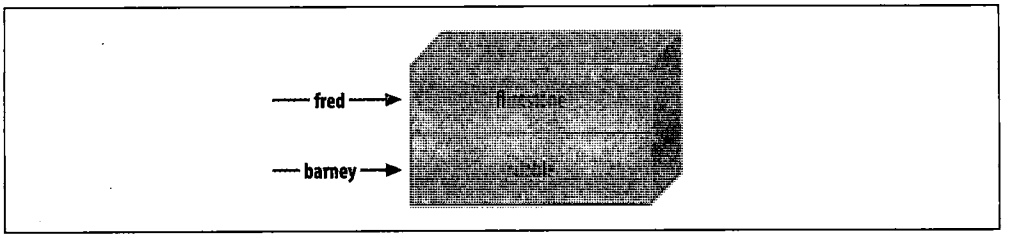

图 6-3：哈希键赋值

这让我们可以写出这样的代码：

```
foreach $person (qw< barney fred >) {
  print "I've heard of $person $family_name{$person}.\n";
}
```

哈希变量的命名和其他 Perl 的标识符相似，可以有字母、数字和下划线，但不可以用数字开头。另外哈希有自己的名字空间，也就是说哈希元素 `$family_name{"fred"}` 和子程序 `&family_name` 之间毫无关联。当然也没有必要故弄玄虚把所有东西都起同样的名字。假如同时有一个名叫 `$family_name` 的标量变量以及像 `$family_name[5]` 这样的数组元素，Perl 也毫不在意。我们人类要学习 Perl 的做法，换句话说，我们必须仔细看清楚标识符前后的标点符号来判断它的真实意义。倘若名称之前有一个美元符号而之后紧接着花括号，那么此处访问的就是一个哈希元素。

在挑选哈希名字的时候，最好使得哈希名和键之间能放进去一个 `for` 字。比如 `family_name for fred` 是 `flintstone`。因此把哈希命名为 `family_name` 能清晰的反应出键和值之间的关系。

当然，哈希键不一定是上面例子里的字符串或简单的标量变量，也可以是任意的表达式：

```
$foo = "bar";
print $family_name{ $foo . "ney" };  # 打印 "rubble"
```

注5：为什么要用花括号，而不用方括号呢？Larry 的解释是：因为哈希的访问方法要比常规的数组访问方法更酷一些，所以也自然需要使用更花哨的括号。

若对某个已存在的哈希元素赋值，就会覆盖之前的值：

```
$family_name{"fred"} = "astaire";  # 给已有的元素赋上新值
$bedrock = $family_name{"fred"};   # 得到 "astaire"，早先的值已不存在
```

这和数组与标量的情形类似，如果对 $pebbles[7] 或者 $dino 赋值，就会覆盖之前的值。如果对 $family_name{"fred"} 赋值，同样会覆盖之前的值。

哈希元素会因赋值而诞生：

```
$family_name{"wilma"} = "flintstone";              # 增加一个新的键值对
$family_name{"betty"} .= $family_name{"barney"};  # 在需要的时候动态创建该元素
```

这和数组与标量的情形如出一辙，假如以前不存在 $pebbles[17] 或 $dino，赋值之后这些变量就会出现。假如以前不存在 $family_name{"betty"}，现在就有了。

访问哈希表里不存在的值会得到 undef：

```
$granite = $family_name{"larry"};  # 没有 larry 这个键，所以值为 undef
```

此做法也跟数组与标量的情形相似：假如 $pebbles[17] 或 $dino 里还没有值，访问这些变量时会得到 undef。假如没有任何值存放在 $family_name{"larry"} 中，访问它的时候也会得到 undef。

访问整个哈希

要指代整个哈希，可以用百分号（%）作为前缀。因此前面我们使用的哈希应该称为 %family_name。

为了方便起见，哈希可以被转换成列表，反过来也行。对哈希赋值会带来列表赋值的上下文，列表的元素应该是键／值对。下面代码对应于图 6-1：【注 6】

```
%some_hash = ("foo", 35, "bar", 12.4, 2.5, "hello",
      "wilma", 1.72e30, "betty", "bye\n");
```

在列表上下文中，哈希会自动变成一些简单的键／值对：

```
@any_array = %some_hash;
```

我们把这个变换叫做哈希松绑，将它变成键／值对的列表。当然键／值对也不一定按照当初赋值的顺序松绑。

注 6：尽管任何列表都可以使用，但是必须得有偶数个成员，因为哈希必须是键／值对。奇数会导致不可靠的结果，当然你仍然可以忽略这个警告。

```
print "@any_array\n";
  # 可能会给出像这样的结果：
  # betty bye (以及一个换行) wilma 1.72e+30 foo 35 2.5 hello bar 12.4
```

之所以顺序乱掉是因为 Perl 已经为了哈希的快速检索而对键／值对的存储作了特别的排序。因此选择使用哈希的场合，要么元素存储顺序无关紧要，要么可以容易地在元素输出时进行排序。

当然，即使键／值对的顺序被打乱，列表里的每个键还是会带着相应的值。所以，即使无法知道某个键 foo 会出现在列表的哪个位置，仍然可以确信相应的值 35 会跟在后面。

哈希赋值

这是个不常见的用法，但是哈希真地可以这样复制：

```
%new_hash = %old_hash;
```

这里 Perl 做的工作要比看到的复杂得多。因为 Perl 的数据结构复杂得多，所以它和 Pascal 或 C 语言里简单的复制内存块的做法有很大差异。其实这一行代码会导致 %old_hash 松绑成为键／值对的列表，然后通过列表赋值重新构造每个键／值对，最终形成哈希 %new_hash。

更常见的用法是以某种方式转换哈希。例如建立一个反转哈希：

```
%inverse_hash = reverse %any_hash;
```

这会将 %any_hash 松绑成为键／值对的列表，看起来是这样的 (*key, value, key, value, key, value, ...*)。然后利用 reverse 的列表翻转功能，形成一个这样的新列表 (*value, key, value, key, value, key, ...*)，这就导致键值互换。当结果存回 %inverse_hash 时，我们就能以原本在 %any_hash 里的值来进行检索，它已经成为 %inverse_hash 的键，而按它找到的值则是 %any_hash 的某个键。现在我们能够按值来找键了。

当然敏锐的读者可能猜到这种技巧只能在哈希值唯一的情况下才能奏效，否则就会导致重复的键，而这对于哈希是不可能的。对于这个问题 Perl 采用后发先至的原则，用列表最后的键覆盖之前的键。当然我们说过了键／值对在哈希松绑之后的顺序是无法预知的，所以也无法预知哪个键会被覆盖。这个技巧最好是在确定原始哈希的值唯一的情况下使用【注 7】。不过这个技巧也适用于前面提到的 IP 地址和主机名的例子：

注 7：或是当你不介意哈希键重复的时候。举例来说，我们可以将 %family_name 哈希从名／姓表反转成为姓／名表，用来查看某个姓是否存在。如果反转后的哈希没有 slate 键，我们就能确定原始的哈希也没有这个姓。

```
%ip_address = reverse %host_name;
```

这样一来，我们就可以轻松地用主机名或 IP 地址来检索相应的 IP 地址或主机名。

胖箭头

在哈希赋值时常常会发现列表中的键／值对并不易区分开来。例如在下面的赋值中，任何人都要逐个扫描列表成员，同时默念着：键、值，键、值等等，然后才搞清楚 2.5 其实是一个键，而不是值：

```
%some_hash = ("foo", 35, "bar", 12.4, 2.5, "hello",
      "wilma", 1.72e30, "betty", "bye\n");
```

如果 Perl 能让我们将此类列表中的键与值成对组合，方便区别谁是谁，岂不更好？Larry 也有感于此，因此他发明了胖箭头（=>）【注 8】。对 Perl 而言，它只是逗号的另一种写法，因此我们常常称呼它为胖逗号。也就是说，在任何需要逗号的地方都可以用胖箭头代替，这对 Perl 来说没什么区别【注 9】。所以产生名／姓哈希的另一种方式是：

```
my %last_name = (  # 哈希也可以是词法变量
  "fred"   => "flintstone",
  "dino"   => undef,
  "barney" => "rubble",
  "betty"  => "rubble",
);
```

这样组合的姓和名看起来更加清晰了，哪怕把所有的姓和名都写在一行里面也可以读懂。请注意，列表结尾有一个额外的逗号，这种写法不但无伤大雅，而且便于维护。当需要加入更多人的信息的时候，只要确保每行都有一组键／值对和结尾的逗号就行了。Perl 会明白每个键／值对之间隔着逗号，整个列表以一个额外的逗号结尾。

注 8：没错还有一个瘦箭头 ->，它是用来服务于引用（reference）的，可以参考 *perlreftut* 或者 *perlref* 了解引用这个高级话题。

注 9：哦，其实还有一个细微的差别：胖箭头左边的任何裸词都能自动引用，因此胖箭头左边的裸词不需要引号。另外，哈希键所在的花括号也有类似的自动引用能力，因此裸词也可以在那里使用。裸词的定义是字母、数字和下划线的序列，可以用加号或减号开头，但不能用数字开头。

哈希函数

很自然，围绕哈希有很多有用的函数。

keys 和 values 函数

keys 函数能返回哈希的键列表，而 values 函数能返回值列表。如果哈希没有任何成员，则两个函数都返回空列表：

```
my %hash = ("a" => 1, "b" => 2, "c" => 3);
my @k = keys %hash;
my @v = values %hash;
```

这样，@k 会包含 "a"、"b" 和 "c"，而 @v 则会包含 1、2 和 3。当然顺序会有所不同，别忘了 Perl 存储哈希的顺序无法预测。但是可以确定返回的键列表和值列表的顺序是一样的：如果 "b" 是键列表的最后一个元素，那么 2 也一定是值列表的最后一个元素。如果 "c" 是键列表的第一个元素，那么 3 也一定是值列表的第一个元素。只要在取得键与取得值这两个动作之间没有改变哈希，顺序就会一致。但是如果哈希里新增了元素，Perl 就可能自作主张重新排列，从而保持高速检索的能力【注 10】。在标量上下文中，这两个函数都会返回哈希中键／值对的个数。这个计算过程不必对整个哈希进行遍历，因而非常高效：

```
my $count = keys %hash;  # 得到 3，也就是说有三对键值
```

偶尔也能看到别人的程序里把哈希当成布尔表达式来判断真假，例如：

```
if (%hash) {
  print "That was a true value!\n";
}
```

只要哈希中有一个键／值对，就会返回真【注 11】。所以，这样写的意思就是，假如哈希不是空的则如何如何。不过，这属于相当罕见的写法。

注 10：显然，如果你在 keys 和 values 调用之间又增加了新的哈希元素，那么返回的两个列表就会有不同数量的元素，从而导致它们难以匹配。因此正常的人不会这么写程序。

注 11：实际的结果将是对 Perl 维护人员很有用的一个内部调试用的字符串。字符串类似于 4/16，如果哈希不是空的，这个值就是真，若是空哈希则返回假，因此普通用户可以将它当成布尔值使用。

each 函数

如果需要罗列哈希的每个键/值对，常见的写法就是使用 each 函数，它能用两个元素的列表形式返回键/值对【注 12】。每次对同一个哈希调用此函数，它就会返回下一组键/值对，直到所有的元素都被访问过。也就是再没有任何新的键/值对，此时 each 会返回空列表。

实际使用时，唯一适合使用 each 的地方就是在 while 循环中，如下所示：

```
while ( ($key, $value) = each %hash ) {
  print "$key => $value\n";
}
```

这里有很多技巧。首先，each %hash 会从哈希中返回一组键/值对，结果是含有两个元素的列表：如果键是 "c" 而值是 3，则列表就会是 ("c", 3)。该列表会赋值给 ($key, $value)，因此 $key 会成为 "c"，而 $value 则变成 3。

但是这里的列表赋值操作是在 while 循环的条件表达式中计算的，也就是在标量上下文中赋值。说得更具体些，最终在 while 内部的是布尔上下文，目的是要求真假值，布尔上下文是一种比较特殊的标量上下文。赋值后得到的列表，在标量上下文中的求值结果为列表的成员数量，所以在这个例子中，我们得到的是 2。因为 2 是真值，所以继而运行循环块，并打印 c => 3。

下一次循环中，each %hash 会返回一组新的键/值对，假设这次是 ("a", 1)。之所以能返回下一个键/值对，是因为哈希还记着上次访问的位置，更深入的说就是每个哈希都有一个定位器【注 13】。这两个值会存进 ($key, $value)。因为列表的元素个数是真值 2，所以继续执行循环块，输出 a => 1。

再一次执行循环时，我们已经知道会发生什么事了，所以看到输出是 b => 2 时不会感觉意外。

注 12：另一个常见的罗列哈希的方法是用 foreach 遍历哈希的键列表，这一节末尾能看到这种写法。

注 13：由于每个哈希有自己的定位器，因此处理不同哈希的 each 调用可以嵌套。既然大家已经习惯了脚注的风格，不妨介绍一些少见的技巧：使用 keys 或者 values 函数可以重置哈希的定位器。另外用列表重置哈希内容时也可以重置定位器，或者 each 调用遍历了整个哈希的时候也能重置。然而在遍历哈希的过程中增加新的键/值对就不太好，因为这不会重置定位器，反而会迷惑开发人员、维护程序员，另外还能愚弄 each。

我们知道循环不可能一直运行下去。当 Perl 执行 each %hash 却已经没有任何键/值对时，each 会返回空列表【注 14】。空列表会被赋值到 ($key, $value)，因此 $key 得到 undef，$value 也会得到 undef。

但因为这是在 while 循环的条件表达式中运行，所以刚才的赋值都不重要。在标量上下文中，列表赋值运算的值是源列表中元素的个数，在此情况下是 0。因为 0 这个值为假，所以 while 循环就结束了，程序会继续运行接下来的部分。

当然，each 返回键/值对的顺序是乱的。但它与 keys 和 values 返回的顺序相同，也就是哈希的自然顺序。假如你需要依次处理哈希，只要对键排序就行了。方法如下如示：

```
foreach $key (sort keys %hash) {
  $value = $hash{$key};
  print "$key => $value\n";
  # 或者，可以略去额外的 $value 变量：
  #  print "$key => $hash{$key}\n";
}
```

我们将会在第十四章看到更多有关哈希排序的内容。

哈希的典型应用

讲到这里不妨一起看看哈希的更多应用实例。

Bedrock 图书馆的借阅信息程序以一个哈希来记录每个人借出了几本书：

```
$books{"fred"} = 3;
$books{"wilma"} = 1;
```

要判断某项哈希元素的真假很简单：

```
if ($books{$someone}) {
  print "$someone has at least one book checked out.\n";
}
```

然而哈希里有一些元素并不为真：

```
$books{"barney"} = 0;        # 现在没有借阅图书
$books{"pebbles"} = undef;   # 从未借阅过图书，这是张新办的借书卡
```

因为 Pebbles 不曾借过任何书，所以她的键值是 undef，而不是 0。

注 14：因为是在列表上下文中运行，所以它不能返回 undef 表示失败；那会被转化为一个元素的列表（undef）而不是空列表（ ）。

每个有借书证的人在哈希里都有相应的键。对于每个键（也就是图书馆的借阅者）来说，都有相应的值，这个值若不是借出图书的数量，就是 undef，意味着一个从未使用过借书证的读者。

exists 函数

若要检查哈希中是否有某个键（也就是某人是否有借书证），可以使用 exists 函数，它能返回真或假，分别表示键存在与否，和键对应的值无关：

```
if (exists $books{"dino"}) {
  print "Hey, there's a library card for dino!\n";
}
```

也就是说，exists $books{"dino"} 返回真，意味着 dino 存在于 keys %books 返回的键列表中。

delete 函数

delete 函数能从哈希中删除指定的键及其相对应的值。假如没有这样的键，它就会直接结束，而不会出现任何警告或错误信息。

```
my $person = "betty";
delete $books{$person};  # 撤回 $person 的借书卡
```

请注意，这并不是将 undef 存入哈希值。在这两种情况下，exists($books{"betty"}) 会得出相反的结果。在 delete 之后，键便不会出现在哈希之中，但是存入 undef 之后键却一定会存在。

在这个例子中，delete 与存入 undef 的差异，就如同取消 Betty 的借书证与给她一张没用过的借书证一样完全不同。

哈希值内插

可以 将单一哈希元素内插到双引号引起的字符串中：

```
foreach $person (sort keys %books) {                # 按次序访问每位借阅者
  if ($books{$person}) {
    print "$person has $books{$person} items\n";    # fred 借了 3 本书
  }
}
```

但是整个哈希的内插是不支持的，"%books" 的结果只是6个字符的串 %books【注15】。到这里为止，我们已经看到了所有在双引号中需要反斜线的魔力字符：$ 和 @ 符号，因为它们后面的变量能进行内插。还有 "，若不用反斜线转义，这个符号就会结束双引号开头的字符串。另外还有 \ 自己也需要转义。双引号中其他的任何字符都代表自己，不必转义【注16】。

%ENV 哈希

Perl 程序既然运行在某个环境中，就需要对周围的影响有所感知。Perl 获取这些信息的方法是存取 %ENV 哈希。例如常常需要从 %ENV 中存取 PATH 键的值：

```
print "PATH is $ENV{PATH}\n";
```

根据你的操作系统和设定，可能看到如下的信息：

```
PATH is /usr/local/bin:/usr/bin:/sbin:/usr/sbin
```

多数时候这些环境变量都已经自动设置好了，但你也可以添加自己的环境变量。不同的操作系统和 shell 有不同的设定方法：

```
Bourne shell

  $ CHARACTER=Fred; export CHARACTER
  $ export CHARACTER=Fred

csh

  % setenv CHARACTER=fred

DOS or Windows command

  C:> set CHARACTER=fred
```

只要在程序外设定了任何的环境变量，就可以在 Perl 中如此获取：

```
print "CHARACTER is $ENV{CHARACTER}\n";
```

注15：它实在不可能代表任何别的东西，如果想用它来输出整个哈希里所有的键/值对，这么做几乎没有任何用处。此外在上一章中，百分比号经常会出现在 printf 的格式字符串里，如果赋予它别的意义，事情将会变得很不方便。

注16：但是要小心变量名之前的缩写符号（'）、左方括号（[）、左花括号（{）、瘦箭头（->）和双冒号（::），因为它们可能会带来意外的麻烦。

习题

以下习题答案参见附录 A：

1. [7] 编程读入用户指定的名字并且汇报相应的姓。拿熟人的姓和名测试，如果你太专注电脑以至于一个人也不认识的话，也可以使用如下列表：

   表 6-1：数据样本

输入	输出
fred	flintstone
barney	rubble
wilma	flintstone

2. [15] 编程读取一系列单词，每行一个【注 17】直到文件结束，然后打印每个单词出现次数的列表。（提示：别忘了，把未定义值当成数字使用时，Perl 会自动将它转换成 0。回头去看看前面计算总和的习题，可能会有所帮助。）这样如果输入单词为 fred、barney、fred、dino、wilma、fred，每个词一行。输出应该告诉我们 fred 出现了 3 次。附加题，如何根据 ASCII 编码排序输出报表。

3. [15] 编程输出 %ENV 哈希所有的键／值对，输出按照 ASCII 编码排序，分两列打印。附加题，设法让打印结果纵向对齐。注意 length 函数可以帮助确定第一列的宽度。测试完毕后加入更多新环境变量再次验证程序的输出无错。

注 17： 必须每个单词分行输入，因为我们暂时还没有介绍如何对一行输入进行分词处理。

第七章

漫游正则表达式王国

Perl 有众多区别于其他语言的特色。在这些特色中最重要的就是对正则表达式的强力支持。这些支持提供了快速、灵活、可靠的字符串处理能力。

不过，这个能力是有代价的。正则表达式其实是 Perl 内嵌的、自成一体的微型编程语言。没错，你正在学习另一门语言【注 1】！好在这个语言并不难。在这一章，你将会进入正则表达式的王国，暂时忘掉 Perl 世界也没关系。在下一章我们会告诉你，这个王国是如何融入 Perl 世界的。

正则表达式并不只是 Perl 的一部分，它们也出现在 *sed*、*awk*、*procmail* 与 *grep* 中，以及在大多数程序员专用的编辑器（如 *vi* 与 *emacs*）中，甚至是更奇怪的地方。好消息是，如果你用过上述的某些工具，那么已经赢在起点了。再仔细看看，你会发现更多使用或支持正则表达式的工具，像网上的搜索引擎（通常就是用 Perl 写成的）、电子邮件客户端等等。坏消息是各家正则表达式的语法不尽相同，在学习的过程中，可能得习惯时不时出现的反斜线。

什么是正则表达式？

正则表达式，在 Perl 中常常叫做模式，是一个匹配（或不匹配）某字符串的模板【注 2】。

注 1：可能会有人说正则表达式不是完整的编程语言。我们懒得辨解。

注 2：爱较真的人会要求更严谨的定义。可是即使给出定义，他们也会说 Perl 的模式不算真正的正则表达式。你如果想认真学习正则表达式，我们推荐 Jeffrey Friedl 写的《Mastering Regular Expressions》（O'Reilly 出版）。

也就是说，虽然有无限多可能的文本字符串存在，但只要用一个模式就可以将它们干净利落地分成两组：匹配的串与不匹配的串。模式绝对没有仁慈、写意之类的性格，它要么匹配，要么就不匹配。

模式可能只匹配一个字符串，或是两个、三个、十个、上百个，或多得无法计数。然而也可能匹配所有的字符串，除了一个、多个或无限多个【注 3】。前面说了，正则表达式是一种小程序，它们说简单的方言。其实这个程序的任务很简单：查看一个字符串，然后决定匹配或不匹配【注 4】。这就是它生活的全部。

另外一个用到正则表达式的地方就是 Unix 的 *grep* 命令，它会检查哪几行文本匹配指定的模式，然后输出那几行。举例来说，如果你想知道某个文件在哪一行提到 `flint`，并且同一行内还跟着 `stone`，你可以利用如下的 *grep* 命令：

```
$ grep 'flint.*stone' chapter*.txt
chapter3.txt:a piece of flint, a stone which may be used to start a fire by striking
chapter3.txt:found obsidian, flint, granite, and small stones of basaltic rock, which
chapter9.txt:a flintlock rifle in poor condition. The sandstone mantle held several
```

不要把正则表达式和 shell 的文件名通配（*glob*）混为一谈。在 Unix shell 中键入 `*.pm` 来匹配所有以 `.pm` 结尾的文件就是典型的文件名通配。上面的例子使用了 `chapter*.txt` 这样的文件名通配（你也许已经注意到了，必须用单引号将正则表达式括起来，不然会被 shell 当成文件名通配）。文件名通配使用了许多与正则表达式相同的字符，但是这些字符在使用方式上完全不同【注 5】。我们会在第十三章进一步介绍文件名通配，但现在暂且放一下。

使用简易模式

若模式匹配的对象是 `$_` 的内容，只要把模式写在一对正斜线（/）中就可以了，如下所示：

```
$_ = "yabba dabba doo";
if (/abba/) {
```

注 3：也可能会有总是匹配或永远不匹配的模式。在极少数情况下，这种模式可能也会有用，但一般来说属于使用不当。

注 4：程序也可以返回信息供 Perl 参考。其中一种信息是正则表达式捕获，稍后我们会提到。

注 5：文件名通配有时也被叫做模式。不过更糟的是，某些差劲的 Unix 入门书（可能恰好也是门外汉写的）也称它为正则表达式，其实它们绝对不是。这种说法只会让许多 Unix 初学者感到迷惑。

```
    print "It matched!\n";
}
```

表达式 /abba/ 会在 $_ 中寻找这 4 个字符组成的串，如果找到就返回真。这里会找到不止一个字符串，但这并不是关键。只要曾经找到过，匹配结果就是真，否则为假。

由于模式匹配通常用来返回真或假值，所以往往会在 if 或 while 的条件表达式里看到它。

所有在双引号圈引的字符串中能使用的技巧（尤其是反斜线转义），都可以在模式串里使用。因此，/cake\tsprite/ 这个模式会匹配 cake、一个制表符和 sprite 这 11 个字符。

关于元字符

当然，如果模式只能用来匹配正文串，那就没什么特色了。因此我们引入了不少特殊字符，叫做元字符，在正则表达式中表达特殊的含义。

例如，点号 (.) 是任何单字符的通配符，当然换行（也就是 "\n"）要除外。因此，betty 将会被 /bet.y/ 这个模式匹配，而 betsy、bet=y、bet.y，或是任何前三个字符为 bet、中间接任何一个字符（换行符除外）、后面为 y 的字符串，也都会被 /bet.y/ 匹配。但是，像 bety 或 betsey 就不匹配了，因为在 t 与 y 之间没有（或超过）一个字符。点号只能用来匹配一个字符。

因此如果你想要匹配字符串中的句点号，虽然也可以用点号，但这样会额外匹配不相干的字符（换行符除外）。要是希望点号只能匹配句点号本身，只需在前面加上反斜线就好了。此规则也适用于任何 Perl 正则表达式里用到的元字符：在任何元字符前面加上反斜线，就会使它失去元字符的特殊作用。例如，/3\.14159/ 这个模式里面就没有通配符。

因此，反斜线是我们的第二个元字符。如果要得出真正的反斜线，请用两个反斜线表示，这个规则也适用于 Perl 的其他地方。

简易的量词

常常需要在某个模式中重复某些东西。而星号（*）正是用来匹配前面的内容零次或多次的。因此 /fred\t*barney/ 能匹配 fred 和 barney 之间有任意多个制表符的串。也就是说这个模式包含了以下所有的情形：用 "fred\tbarney" 来匹配一个制表符；用 "fred\t\tbarney" 来匹配两个制表符；用 "fred\t\t\tbarney" 来

匹配三个制表符；甚至包括用 "fredbarney" 来匹配两者之间为空的串。因为星号表示匹配零次或更多，所以两个字符之间可以有数百个制表符，但是除了制表符之外，不能出现其他字符。把星号想成前面的东西可以重复任意多次，也可以是零次，这可能会比较容易理解。因为星号是乘法操作符，而乘法也就是重复的意思。

如果除了制表符外还想匹配其他字符，该怎么做呢？点号会匹配任意字符【注6】，因此 .* 会匹配任意字符无限多次。也就是说，不管 fred 与 barney 之间夹着什么东西，都会匹配模式 /freb.*barney/。任何一行中只要提到了 fred，并且在其后提到 barney，就会匹配此模式。我们经常戏称 .* 为捡破烂模式，因为它能通吃所有的字符串。

星号应该说是一种量词，它指定了前一个条目的数量。但是，它不是唯一的量词，加号(+)是另外一个。加号会匹配前一个条目一次以上：/fred +barney/ 会匹配在 fred 与 barney 之间用空格隔开而且只用空格隔开的字符串（空格不是元字符）。它不会匹配 fredbarney，因为加号表示在两个名称之间必须有一个以上的空格。所以，这里起码少了一个空格。把加号想成“算上刚才所说的，再加上任意次重复”，或许会比较容易理解。

第三个量词与星号及加号类似，但是限制更严格。也就是问号（?），表示前一个条目是可有可无的。就是说，它的前一个条目可以出现一次或是不出现。和另外两个量词相同的是，问号也指定了前一个条目出现的次数。不过在使用问号的情况下，它只会出现一次（如果有匹配的条目）或不出现（如果没有匹配的条目），除此之外没有任何其他的可能性。这很容易理解为：“刚才所说的，有还是没有都行”。

因为这三个量词指定了*前*一个条目重复出现的次数，所以它们都必须接在某个东西之后。

模式分组

在数学中，圆括号（()，或称小括号）用来分组。因此，圆括号也是元字符。举例来说，模式 /fred+/ 会匹配像 freddddddddd 这样的字符串，可是这种字符串实际上不常出现。不过，模式 /(fred)+/ 会匹配像 fredfredfred 这种字符串，这可能才是

注 6：换行符除外。接下来不会一直提醒这件事，因为你已经知道了。反正，因为字符串里不太会出现换行符，所以它通常没什么影响。但是别忘了这个细节，因为总有一天，某个换行符会溜进你的字符串里，你必须记住点号不会匹配换行符。

你想要的。那么，模式 /(fred)*/ 又如何呢？它会匹配像 Hello, world 这样的字符串【注 7】。

圆括号同时也使得部分字符串重新引用成为可能。我们可以用反向引用来引用圆括号中（的模式所）匹配的文字。\1、\2 这样的写法就是在使用反向引用。而反斜线后面的数字和括号的组号匹配。

还记得么？使用圆括号包围的点号可以匹配任何非回车字符。我们可以用如下程序反向引用刚刚匹配的字符 \1：

```
$_ = "abba";
if (/(.)\1/) {  # 同 'bb' 相匹配
  print "It matched same character next to itself!\n";
}
```

(.)\1 表明需要匹配连续出现的两个同样的字符。(.)会首先匹配 a，但是在匹配反向引用的时候就会发现下一个字符不是 a，导致匹配失败。Perl 会跳过这个字符，用(.)来匹配下一个字符 b，在匹配反向引用的时候会发现下一个字符也是 b，这样就构成了成功的匹配。

反向引用不必总是附在相应的括号后面。下面的模式会匹配 y 后面的 4 个连续的非回车字符，并且用 \1 在 d 字符之后重复这 4 个字符：

```
$_ = "yabba dabba doo";
if (/y(....) d\1/) {
  print "It matched the same after y and d!\n";
}
```

也可以用多个括号来分成多组，每组都可以有自己的反向引用。我们可以用括号定义一个非换行字符的组，然后再跟上一个非换行字符的组。然后用反向引用 \2 和 \1 来构成有趣的回文模式，比如 abba：

```
$_ = "yabba dabba doo";
if (/y(.)(.)\2\1/) { # 同 'abba' 相匹配
  print "It matched the same after y and d!\n";
}
```

讲到这里经常会有人问，该如何区分哪个括号是第几组？幸运的是 Larry 是用合乎常理的方式来给它们编号的，只要数左括号（包括嵌套括号）的序号就可以了：

注 7：星号代表要匹配重复出现零次以上的 fred。在能接受零次的情况下，要使匹配失败是很难的！任何字符串都匹配该模式，即使是空字符串也一样。

```
$_ = "yabba dabba doo";
if (/y((.)(.)\3\2) d\1/) {
  print "It matched!\n";
}
```

有时可能得拆开来写，才能看清楚这个模式中各个部分的结构（当然，这种写法的格式不正确【注 8】）：

```
(         # 第一个左括号
    (.)   # 第二个左括号
    (.)   # 第三个左括号
    \3
    \2
)
```

Perl 5.10 有一种新的反向引用写法。不再只是简单地用反斜线和组号，而是用了 \g{N} 这种写法。其中 N 是想要反向引用的组号。在刚才的例子中使用这种新写法会更加清晰。

以往一想到要在数字模式中使用反向引用，就会觉得非常麻烦。在下面的例子中就能看到这种情况，我们要用 \1 来重复刚刚用括号匹配的字符，然后是紧接着的原始串 11：

```
$_ = "aa11bb";
if (/(.)\111/) {
  print "It matched!\n";
}
```

Perl 必须猜测我们的意思：这里到底是 \1、\11 还是 \111 呢？在这个问题上 Perl 的逻辑很简单，尽可能多的尝试反向引用，从而导致 \111 是最终的答案。然而因为没有第 111 或 11 组括号存在，Perl 会在程序编译时失败。

通过使用 \g{1}，就能排除前面模式中的二义性：【注 9】

```
use 5.010;

$_ = "aa11bb";
if (/(.)\g{1}11/) {
  print "It matched!\n";
}
```

用 \g{N} 写法的额外好处是，我们甚至可以用负数。相比绝对位置的反向引用，相对反向引用会更加有趣。这样我们可以用 -1 来表达刚才的模式：

注 8：当然其实可以用 /x 修饰词来展开复杂的正则表达式，下一章我们再说明具体该怎么做。

注 9：通常我们会用 \g{1} 更精简的形式 \g1 来表达，但还是建议写上花括号。为避免困惑，我们总是用更长的格式。

```
use 5.010;

$_ = "aa11bb";
if (/(.)\g{-1}11/) {
  print "It matched!\n";
}
```

这样若要在模式中加入更多的内容，就不必总是修改反向引用了。因为要加入另外一组括号，就会导致大多数绝对反向引用失效，而相对反向引用则不会，因为它使用的是相对于自己的位置，而不是绝对编号，所以维护起来很轻松：

```
use 5.010;

$_ = "aa11bb";
if (/(.)(.)\g{-1}11/) {
  print "It matched!\n";
}
```

择一匹配

竖线(|)通常可以读成“或”，意思是左边匹配或者右边匹配都行。也就是说如果左边的模式匹配失败了，还可以用右边的再试试。因此 /fred|barney|betty/ 能匹配任何含有 fred 或者 barney 或者 betty 的字符串。

现在可以使用 /fred(|\t)+barney/ 这样的模式来匹配 fred 和 barney 之间空格、制表符或两者组合出现一次以上的字符串。加号表示重复一次或更多。每次只要有重复，(|\t) 就可能匹配空格或制表符【注 10】。在这两个名字之间至少要有一个空格或制表符。

若要求 fred 与 barney 之间的字符必须都一样，你可以把上述模式改成 /fred(+|\t+)barney/。如此一来，中间的分隔符就一定得全是空格或全是制表符。

模式 /fred (and|or) barney/ 可用来匹配任何含有 fred and barney 或 fred or barney 的字符串【注 11】。我们还可以用 /fred and barney|fred or barney/ 模式来匹配这两个字符串，但是这样太长了，执行效率也可能会降低，这取决于正则表达式引擎内建的优化策略。

注 10：这种匹配通常用字符集来做会更有效率，本章稍后会有说明。

注 11：请注意，在正则表达式里 and 与 or 这两个词并不是操作符！因为它们是字符串的一部分，所以在这里用等宽字来表示。

字符集

字符集是指一串可能出现的字符集合，通过写在方括号（[]）内来表示。它只匹配单个字符，但可以是字符集里列出的任何一个。

例如字符集 [abcwxyz] 会匹配这 7 个字符中的任何一个。为了方便起见，你可以使用连字符 (-)，这样之前的字符集也可以写成 [a-cw-z]。这不会省掉几次击键，但是来建立 [a-zA-Z] 这样的经典字符集就非常方便，这个字符集可以匹配 52 个字母中的任何一个【注 12】。定义字符集时可以使用字符简写，类似双引号内的转义序列，因此字符集 [\000-\177] 将会匹配任何 7 位的 ASCII 字符【注 13】。当然，字符集只是完整模式的一部分，在 Perl 中它从来不会单独出现。例如，你可能会见到如下的程序代码：

```
$_ = "The HAL-9000 requires authorization to continue.";
if (/HAL-[0-9]+/) {
  print "The string mentions some model of HAL computer.\n";
}
```

有时候，指定字符集范围以外的字符会比指定字符集内的字符更容易。可以在字符集内部开头的地方加上脱字符 (^)，表示这些字符除外。也就是说，[^def] 会匹配这三个字符以外的任何字符，而 [^n\-z] 则会匹配 n、连字符与 z 以外的任何字符（请注意，这里的连字符要加上反斜线，因为它在字符集里具有特殊意义。但在 /HAL-[0-9]+/ 里的第一个连字符则不需要反斜线，因为字符集括号以外的连字符没有特殊意义）。

字符集简写

有些字符集十分常用，因此具有自己的简写。比方说，代表任意数字的字符集 [0-9] 可以被简写成 \d。因此，上面关于 HAL 的模式可以改写成 /HAL-\d+/。

\w 这个简写表示单词字符：[A-Za-z0-9_]。如果你觉得“单词”里面应该只含有字母、数字和下划线，那你会很高兴看到 Perl 也是这么认为的。不过多数情况下，现实生活中我们所说的单词是由字母、连字符和缩写符号组成的【注 14】，所以我们很想改

注 12：注意 Å、É、Î、Ø 或 Ü 这样的字符并不包含在其中，但是 Unicode 处理功能完成之后，范围就会自动升级以支持这些特殊字符。

注 13：除非你不用 ASCII 而是使用 EBCDIC 字符集。

注 14：起码英语是这样的。可能在其他语言中，组合单词的元素会有所不同。在 ASCII 编码的英文中，单引号和缩写符号是一样的，这造成了些麻烦，无法简单说明 cat's 是一个带缩写的单词还是单引号末尾的词。没准这就是计算机仍然无法（代替人类）管理世界的原因吧。

变“单词”的定义。直至本书完稿之时，Perl 开发者还在为此而努力，但仍未完成【注 15】。因此，只有在你想要字母、数字与下划线时才应该使用 \w。

当然，\w 并不会匹配一个单词，它只会匹配单词字符集里的一个字符。虽然如此，要匹配完整的单词时，只要使用加号修饰符就行了。像 /fred \w+ barney/ 这样的模式，会匹配 fred、一个空格、一个单词再接一个空格与 barney。也就是说，如果在 fred 与 barney 之间有一个两边用空格隔开的单词【注 16】，就会被这个模式匹配。

在前一个例子中，你可能感觉到有必要更灵活地匹配空白。\s 简写擅长处理空白，它相当于 [\f\t\n\r]。也就是说，它等于是包含 5 种空白的字符集：换页、制表、换行、回车以及空格。这些都是专门用来移动打印位置的字符，它们不消耗打印机的墨水。不过，和之前其他的简写一样，\s 只会匹配字符集里的某个字符。因此，比较常见的做法是使用 \s* 来匹配任意数目的空白（也包括了零个空白），或用 \s+ 来匹配一个以上的空白字符（事实上，不带量词使用 \s 是很少见的）。因为这些空白字符对肉眼来说都是一样的，所以我们可以用这个简写来统一处理它们。

Perl 5.10 增加了更多字符集来描述空白。\h 简写能匹配横向空白，其实就是一个包含制表符和空格的字符集 [\t]。\v 这个简写用来匹配纵向的空格，其实也就是写 [\f\n\r] 的快捷方式。\R 简写能匹配任何类型的断行，这给跨操作系统的断行匹配带来了便利，因为 \R 知道各种风格的断行方式。

反义简写

有些时候只是为了获取以上几种简写的反义，而必须写出类似 [^\d]、[^\w] 或是 [^\s] 这样的模式，来表示一个非数字、非词或者非空白字符。其实它们的大写版本就是用来完成这种任务的：也就是 \D、\W 或者 \S。这些大写版本能匹配相应小写版本范围以外的字符。

这些简写既可以作为模式里独立的字符集，也可以作为方括号里字符集的一部分。也就是说，/[\dA-Fa-f]+/ 可以用来匹配十六进制数字，十六进制表示法使用了字母 ABCDEF（或 abcdef）作为额外的数字。

注 15：不过还是在和本地化相关的领域，得到了一些有用但有限的进展，请参考 *perllocale* 在线手册。

注 16：从这里开始，我们不会再将“单词”放在引号里了。至此，你应该知道我们指的是由字母、数字、下划线组成的单词。

另外一个复合字符集是 [\d\D]，表示任何数字或非数字。也就是说，它会匹配任何字符！这是匹配任意字符（包括换行符）的常见做法。（而点号 . 则匹配换行符以外的所有字符）。此外，还有完全无用的 [^\d\D] 字符集，它匹配既不是数字也不是非数字的字符。没错，那就是什么都不匹配！

习题

下列习题的解答请参阅附录 A，对正则表达式的功能感到惊讶是正常的，正因为这样，本章的习题比其他章节的习题更重要。请做好会遇到意外的心理准备：

1. [10] 写个程序，从输入中读取数据，遇到包含 fred 字符串的行就打印出该行。如果输入某一行包含字符串 fred、frederick 或者 Alfred，请问是否会匹配并打印？另外写个文本文件，在里面随意编个 fred flintstone 跟他朋友的故事。然后用这个文件作为输入来测试这一题以及后面几道题。

2. [6] 修改上一题的程序，让它也能接受 fred。那么，当你的输入字符串是 fred、frederick 或者 Alfred 时，是否也匹配？（请将包含这些名称的各行加到刚才的文本文件。）

3. [6] 写个程序，只输出其输入中含有点号的每一行，如果没有就予以忽略。同样以刚才的文本文件进行测试。如果其中有行文字是 Mr. Slate，看看会输出吗？

4. [8] 写个程序，输出含有要求单词的行。要求的单词用大写字母开头，但并非全大写。此程序是否会输出 Fred，而不输出含有 Fred 或 FRED 的行？

5. [8] 写个程序，打印那些有两个相连且相同的非空格字符的行。应该能匹配的词如 Mississippi、Bamm-Bamm 或者 llama。

6. [8] 附加题：写个程序，输出在输入数据中同时出现 wilma 以及 fred 的每一行。

第八章

以正则表达式进行匹配

前一章我们已经漫游了正则表达式的王国。接下来你将会知道，那个世界的一切如何与 Perl 接轨。

以 m// 进行匹配

我们已经用过双斜线的写法来编写模式，好像 `/fred/`。但事实上，这是 `m//`（模式匹配）操作符的简写。就像我们在说明 `qw//` 操作符时提到的，可以选择使用任何成对的定界符。所以，我们可以把同样的表达式写成 `m(fred)`、`m<fred>`、`m{fred}` 或 `m[fred]`。我们也可以使用非成对定界符【注 1】。

这里有个简写：如果你选择双斜线作为定界符，那么你可以省略开头的 `m`。因为 Perl 程序员们喜欢省略多余的字符，所以大部分的模式匹配都会以双斜线来编写，就像 `/fred/` 这样。

显然你应该选择模式中不会出现的符号作为定界符【注2】。在匹配常规网址开头的模式

注 1：非成对定界符是没有左右之分的符号，也就是在两端使用同样的标点符号。

注 2：在使用成对的定界符时，通常不用担心在模式里出现的定界符，因为该定界符在模式里通常是成对出现的。也就是说，`m(fred(.*)barney)`、`m{\w{2,}}` 与 `m[wilma[\n\t]+betty]` 都没问题，即使模式中含有引号也行，因为每一个左定界符号都会有一个相应的右定界符号。但是尖括号（< 与 >）并非正则表达式的元字符，所以它们可能不会成对出现。如果模式是 `m{(\d+)\s*>=?\s*(\d+)}`，那么在以尖括号作为定界符的状况下，大于号前面就需要加上反斜线，才不会过早结束模式。

时，可能会用 /^http:\/\// 匹配起始的 http://。其实可以选择更好的定界符，从而提高代码可读性，也能降低维护成本。例如这么写：m%^http://%【注3】。常见的定界符是花括号。在程序员专用的编辑器里，可能会具有从左花括号跳到相应右花括号的功能，这在维护方面有很大的帮助。

可选修饰符

这是一些可有可无的修饰字符，有时候称为开关。它们可以成组附加在某个正则表达式结尾的定界符的右边，并改变正则表达式的默认行为。

用 /i 来进行大小写无关的匹配

使用 /i 修饰符，可让你在进行模式匹配时不区分大小写，使得你能够轻易匹配 FRED、fred 或 Fred：

```
print "Would you like to play a game? ";
chomp($_ = <STDIN>);
if (/yes/i) {  # 大小写无关的匹配
  print "In that case, I recommend that you go bowling.\n";
}
```

用 /s 来匹配任意字符

默认情况下，点号（.）无法匹配换行符，这对大多数单行匹配的情况是合适的。但是如果字符串中含有换行符，而你希望点号能用来匹配它们，那么 /s 修饰符可以完成这个任务。它会将模式中的每个点号【注4】按字符集 [\d\D] 的效果来处理，就是说会匹配任意字符（包括换行符）。当然，你需要有一个包含换行符的字符串，才能看出它的差异：

```
$_ = "I saw Barney\ndown at the bowling alley\nwith fred\nlast night.\n";
if (/Barney.*fred/s) {
  print "That string mentions fred after Barney!\n";
}
```

因为这两个名称并不在同一行，所以如果省略 /s 修饰符，上述匹配就会失败。

注3：请记住，正斜线并不是元字符，所以使用其他定界符时，不需要再加上反斜线转义。

注4：若你希望其中部分点号能匹配换行符，只需要将那些部分替换成 [\d\D] 就行了。

用 /x 加入空白

第三个修饰符能够在模式里面随意加上空白，目的是使它更容易阅读、理解。

```
/-?\d+\.?\d*/           # 都挤在一起，很难看清是什么意思
/ -? \d+ \.? \d* /x     # 加入空白后，稍微清楚些
```

由于加上 /x 之后模式里面可以任意插入空白，所以原始的空白与制表符就失去意义了，它们会被忽略掉。如果还要匹配空白与制表符的话，就得在前面补上一个反斜线字符（也可以采用其他方式），不过 \s（或 \s*，或 \s+）还是比较常见的匹配空白的写法。

由于在 Perl 里，注释也算是一种空白，所以我们甚至可以借机在模式里写上注释以指明其用途：

```
/
  -?        # 零个或一个减号
  \d+       # 一个或多个数字
  \.?       # 零个或一个小数点
  \d*       # 零个或多个数字
/x          # 字符串结尾
```

因为井号是注释的标记，所以如果要表示真的井号时，就得写成 \# 或 [#]。另外，在注释里不要把定界符也写进去，不然就会被视为模式的终点。

组合选项修饰符

如果在一个模式中使用多个修饰符，可将它们连在一起使用。它们之间的先后顺序并不会影响匹配的结果：

```
if (/barney.*fred/is) {  # 同时使用 /i 和 /s
  print "That string mentions Fred after Barney!\n";
}
```

将同样的模式展开并加上注释后的模样：

```
if (m{
  barney # 小伙子 barney
  .*     # 之间的任何东西
  fred   # 大嗓门的 fred
}six) {  # 同时使用 /s、/i 和 /x
  print "That string mentions Fred after Barney!\n";
}
```

请注意，此处以花括号作为定界符，这让程序员专用的编辑器可以方便地（在正则表达式的两头间）跳转。

其他选项

当然还有许多其他的修饰符选项，我们会在用到的时候加以介绍。当然，你也可以参阅 *perlop* 在线手册中 m// 和正则表达式操作符的相关介绍。另外本章稍后也会继续介绍修饰符。

锚位

默认情况下，模式匹配的过程开始于待匹配字符串的开头，如果不相符就一直往字符串后面浮动，看其他位置能否匹配。但是加入一些锚位，可以让模式直接匹配字符串的某处。

脱字符【注 5】(^) 是一个锚位，用来标示字符串的开头，而美元符号 ($) 也是一个锚位，用来标示字符串的结尾【注 6】。因此，/^fred/ 只匹配位于字符串最前端的 fred，如果是 manfred mann 这个字符串，则不能匹配。而 /rock$/ 也只匹配位于字符串最后面的 rock，如果是 knute rockne，也同样失败。

某些时候，这两个锚位会一起使用，以确保模式可以匹配整个字符串。有个常见的实例是 /^\s*$/，这个模式可以匹配空白行。但是空白行可能包含了若干看不到的空白字符，像空格与制表符。用打印机输出的话，匹配这个模式的每一行都能保持白纸的本色，所以也可以说这个模式对所有的空白行来说等效。如果此处不在前后加上两个锚位，则会把非空白行也一起算进去。

单词锚位

锚位并不局限于字符串的首尾。比如 \b 是单词边界锚位，它匹配任何单词的首尾【注7】。因此，/\bFred\b/ 可匹配 fred，但无法匹配 frederick、alfred 或 manfred mann。这在文字处理器的搜索命令里，通常称为整词搜索模式。

注 5：没错，你的确见过脱字符以别的方式出现在模式里。当它出现在字符集的第一个位置时，代表取字符集的补集。但是在字符集以外时，它就是一个意义完全不同的元字符，代表字符串的开头。毕竟，键盘上能用的字符就这么多，总是会有几个需要重复使用。

注 6：事实上，除了匹配字符串的结尾之外，它同时也匹配字符串结尾的换行符。这样一来，无论字符串后面有没有换行符号，都可以用它来标示字符串结尾的位置。大部分人其实不必区分这些细节，但总有一天会用到的。请记住，模式 /^fred$/ 会匹配 fred 与 fred\n 这两个字符串。

注 7：有些正则表达式引擎会以一个锚位来标示单词开头，并以另一个锚位来标示单词结尾，不过 Perl 只用一个 \b 来通配这两者。

不过，这里所说的单词并不是一般的英文单词，而是由一组 \w 字符构成的字符集，也就是由普通英文字母、数字与下划线组成的单词。\b 锚位匹配的是一组 \w 字符的开头或结尾。

在图 8-1 中，每个词下方会出现灰色下划线，\b 会匹配的位置则以箭头标识。因为每个单词都会有开头与结尾，所以字符串中的单词边界一定是偶数个。

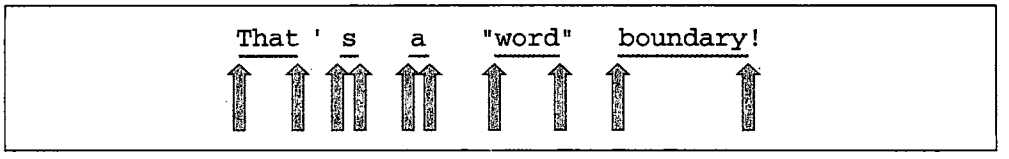

图 8-1：用 \b 匹配单词边界

此处所谓的单词是指一连串的字母、数字与下划线的组合，也就是匹配 /\w+/ 模式的字符串。该句子共有 5 个单词：That、s、a、word 以及 boundary【注 8】。要注意的是，word 两边的引号并不会改变单词边界。这些单词是由一组 \w 字符构成的。

因为单词边界锚位 \b 只匹配每组 \w 字符的开头或结尾，所以每个箭头会指向灰色下划线的开头或结尾。

单词边界锚位非常有用，它保证我们不会意外地在 delicatessen 中找到 cat、在 boondoggle 中找到 dog 或在 selfishness 中找到 fish。有时候，你只会用到一个单词边界锚位，像用 /\bhunt/ 来匹配 hunt、hunting 或 hunter，而排除了 shunt；或是用 /stone\b/ 来匹配 standstone 或 flintstone，但不包括 capstones。

非单词边界锚位是 \B，它能匹配所有 \b 不能匹配的位置。因此，模式 /\bsearch\B/ 会匹配 searches、searching 与 searched，但不匹配 search 或 researching。

绑定操作符 =~

默认的情况下模式匹配的对象是 $_，绑定操作符 =~ 则能让 Perl 拿右边的模式来匹配左边的字符串，而非匹配 $_【注 9】。例如：

注 8：从这里可以看出来，为什么我们想改变单词的定义了，因为 That's 应该算成一个单词，而不该拆成两个单词与所有格符号。就算是在最严肃的英文资料中，也不难看到这种风格的缩写。

注 9：除了模式匹配，某些别的运算也会用到绑定操作符，我们稍后会提到它们。

```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /\brub/) {
  print "Aye, there's the rub.\n";
}
```

绑定操作符虽然看起来像某种赋值操作符，其实并非如此！它只表示本来这个模式会匹配 $_ 变量，但请针对左边的字符串匹配吧。若没有绑定操作符，表达式就会使用默认的 $_。

在下面这个（不寻常的）例子里，$likes_perl 会被赋予一个布尔值，这个结果取决于用户键入的内容。这个程序属于“急功近利”型的，因为判断之后就丢弃了用户的输入。这行代码大致上的功能是读取输入行，匹配字符串与模式，然后舍弃输入行的内容【注 10】。没有进一步使用 $_，也没有改变它。

```
print "Do you like Perl? ";
my $likes_perl = (<STDIN> =~ /\byes\b/i);
...  # 耗时的其他操作 ...
if ($likes_perl) {
  print "You said earlier that you like Perl, so...\n";
  ...
}
```

因为绑定操作符的优先级相当高，也就没必要用圆括号来括住模式测试表达式。所以下面这一行如同上面的表达式一样，会将匹配结果（而非该行输入的内容）存进变量：

```
my $likes_perl = <STDIN> =~ /\byes\b/i;
```

模式串中的内插

正则表达式里可以进行双引号形式的内插。这让我们可以很快写出如下类似 *grep* 的程序：

```
#!/usr/bin/perl -w
my $what = "larry";

while (<>) {
  if (/^($what)/) {  # 模式的锚位被定在字符串的开头
    print "We saw $what in beginning of $_";
  }
}
```

不管 $what 的内容是什么，当我们进行模式匹配的时候，该模式都会成为 $what 的值。在这里它和 /^(larry)/ 是相同的意思，也就是在每行的开头寻找 larry。

注 10：请记住，除非 while 循环的条件表达式中只有整行输入操作符（<STDIN>），否则输入行不会自动存入 $_。

但是，$what 不一定来自字符串直接量，我们可以从 @ARGV 里的命令行参数来取得：

```
my $what = shift @ARGV;
```

如果第一个命令行参数是 fred|barney，则模式会变成 /^(fred|barney)/，也就是在每一行开头寻找 fred 或 barney【注 11】。寻找 larry 时多余的圆括号现在变得很重要，如果没有它们，模式就会在字符串开头匹配 fred 或者在字符串的任何地方匹配 barney。

将程序代码改成从 @ARGV 读取模式后，这个程序就和 Unix 的 *grep* 命令很像。但是，我们必须注意字符串里的元字符。如果 $what 的值为 fred(barney)，则模式就会变成 /^(fred(barney)/，你也知道这样是不会正确工作的，它会以 invalid regular expression 错误来中断你的程序。有一些高级技巧【注 12】，可捕获这类错误（或者从一开始就解除元字符的魔法），让它不再中断你的程序。不过目前只需要记住，一旦赋予用户使用正则表达式的权力，他们就该负起正确使用的责任。

捕获变量

到目前为止，每当我们在模式里使用圆括号的时候，都只是用来表示不同的模式组。但圆括号同时也启动了正则表达式处理引擎的捕获功能。捕获功能指的是，把（圆括号中模式所匹配的）部分字符串暂时记下来的能力。如果有一对以上的圆括号，就会有一次以上的捕获。每个被捕获的对象是原本的字符串，而不是模式。

因为捕捉变量存储的都是字符串，所以它们都是标量变量。在 Perl 里，它们的名字类似 $1 或者 $2。模式里的括号有多少对，匹配变量就有多少个。你大概也已经猜到了，$4 的意思就是第四对括号捕获的字符串【注 13】。

注 11：明眼的读者将会知道，你通常无法在命令行上键入 fred|barney 这个参数，因为竖线符号是 shell 的元字符。请参考手头的 shell 说明文档，学习如何为命令行参数加上引号。

注 12：在此状况下，你可以使用 eval 块来捕捉这个错误。你也可以用 quotemeta（或是它的等效形式 \Q）为内插的文字转义，这样一来，它就不会被当成正则表达式了。

注 13：在模式匹配的过程中，这个变量与反向引用 \4 都是指同样的字符串。但是它们两个并非同一件事的两个名称：\4 引用的是在模式匹配还没完成时的字符串，而 $4 引用的则是模式已经全部匹配完成之后的结果。关于反向引用，参见 *perlre* 在线手册。

这些变量能够取出字符串里的某些部分，因此是正则表达式威力强大的重要原因之一：

```
$_ = "Hello there, neighbor";
if (/\s(\w+),/) {              # 捕获空白符和逗号之间的单词
  print "the word was $1\n";   # 打印 the word was there
}
```

你也可以一次捕获多个串：

```
$_ = "Hello there, neighbor";
if (/(\S+) (\S+), (\S+)/) {
  print "words were $1 $2 $3\n";
}
```

上面的程序代码让我们知道这些单词是 Hello there neighbor。请注意，在输出结果里没有逗号。因为模式里的逗号放在圆括号外面，所以第二次捕获中不会有逗号。使用这个技巧，我们可以精确筛选捕获的（和跳过的）数据。

有时甚至可能产生空匹配变量【注 14】，因为那部分的模式允许为空。也就是说要考虑捕获变量为空的情况：

```
my $dino = "I fear that I'll be extinct after 1000 years.";
if ($dino =~ /(\d*) years/) {
  print "That said '$1' years.\n";  # $1 为 1000
}

$dino = "I fear that I'll be extinct after a few million years.";
if ($dino =~ /(\d*) years/) {
  print "That said '$1' years.\n";  # $1 为空字符串
}
```

捕获变量的生命期

这些捕获变量通常能存活到下次成功的模式匹配为止【注 15】。也就是说，失败的匹配不会改动上次成功匹配时捕获的内容，而成功的匹配会将它们重置。换句话说，捕获变量只应该在匹配成功时使用；否则就会得到之前一次模式匹配的捕获内容。下面的（失败）案例本来应该输出从 $_ 捕获的某个单词。但是，如果比对失败，它会输出可能遗留在 $1 里的任何字符串：

```
$wilma =~ /(\w+)/;  # 不对！这里的结果不一定正确
print "Wilma's word was $1... or was it?\n";
```

注 14：而不是尚未定义。若模式中有三个以下的圆括号，$4 才会是 undef。

注 15：有效范围的规则实际上更为复杂（若有需要请参考说明文档），但除非你希望捕获变量在模式匹配执行后好几行还能保持不变，否则应该不会遇到问题。

这就是为什么模式匹配总是出现在 if 或 while 条件表达式里：

```
if ($wilma =~ /(\w+)/) {
  print "Wilma's word was $1.\n";
} else {
  print "Wilma doesn't have a word.\n";
}
```

既然这些捕获内容不会永远留存，那么 $1 之类的匹配变量只应该在模式匹配后的数行内使用。如果维护程序员在原先的正则表达式和 $1 的使用之间加入了一个新的正则表达式，$1 将会是第二次捕获的值，而非第一次。因此，如果需要在数行之外使用捕获变量，通常最好的做法是将它复制到某个一般的变量里。这么做也使得程序代码变得更容易阅读：

```
if ($wilma =~ /(\w+)/) {
  my $wilma_word = $1;
  ...
}
```

稍后，在第九章我们会学到如何在模式匹配发生的同时，直接将捕获的内容存到变量，免于 $1 这种粗糙的写法。

不捕获模式

目前所见的圆括号都会捕捉部分的匹配串到捕获变量中，但有时候却需要关闭这个功能，而仅仅只是用它来进行分组。比如某个模式中有些部分是可选的，之后的部分却是要捕捉的。好比在某些时候巨无霸（bronto）是可选的，而之后的部分，比如牛排（steak）或者汉堡（burger）却正是我们感兴趣的一样。

```
if (/(bronto)?saurus (steak|burger)/) {
    print "Fred wants a $2\n";
}
```

尽管有时 bronto 是不存在的，还是得把 $1 变量留给它。Perl 只是按左圆括号的序号来决定捕获变量名，这导致我们真正想捕获的内容只能进入 $2。在更加复杂的模式中，这种情况非常让人困惑。

还好 Perl 的正则表达式允许使用括号但不作捕捉。我们把这叫做不捕捉括号，书写的时候也有些差别。需要在左括号的后面加上问号和冒号(?:)【注 16】，以告知 Perl 这一对括号完全是为了分组而存在的。

注 16：这是问号在正则表达式中的四种用法：原文问号（需要转义）、有无量词、非贪婪修饰符（下一章介绍）和这里的不捕捉前缀。

可以在这里使用不捕捉括号来跳过 bronto，这样就可以用 $1 来捕获需要的内容：

```
if (/(?:bronto)?saurus (steak|burger)/) {
    print "Fred wants a $1\n";
}
```

之后若我们修改正则表达式，以支持巨无霸汉堡的熏肉特色版本，就可以在这个模式中加入不捕捉的选项。这时候感兴趣的串仍然会进入捕获变量 $1。若是没有这个功能，就得在每次加入分组括号之后修改捕获变量的名字。

```
if (/(?:bronto)?saurus (?:BBQ )?(steak|burger)/) {
    print "Fred wants a $1\n";
}
```

Perl 的正则表达式有很多其他的圆括号修饰符能完成更复杂的功能，比如前瞻、后顾、内嵌注释，甚至可以在模式中嵌入执行程序。可以参考 *perlre* 在线手册了解细节。

命名捕捉

虽然可以用括号的捕捉能力并且在 $1、$2 这样的变量中存取捕获的串。但是即使对于较为简单的模式来说，管理这样的数字变量也是比较困难的。例如下面这个匹配两个名字的正则表达式：

```
use 5.010;

my $names = 'Fred or Barney';
if( $names =~ m/(\w+) and (\w+)/ ) { # 不会被匹配
    say "I saw $1 and $2";
    }
```

实际上我们看不到 say 的输出，这是因为模式串中的期望是 and，而实际变量 $names 中提供的是 or。我们考虑之后认为应该允许两者并存，所以就在正则中加入“择一”来匹配 and 或者 or。这当然需要在模式中加入一对括号：

```
use 5.010;

my $names = 'Fred or Barney';
if( $names =~ m/(\w+) (and|or) (\w+)/ ) { # 现在可以匹配了
    say "I saw $1 and $2";
    }
```

现在我们看到了第二个输出，但是它却不是我们期望的名字，因为第二对括号的引入导致了问题。引入普通的括号导致“择一”模式匹配的串进入了 $2，而期望的串则意外进入了 $3：

```
I saw Fred and or
```

当然我们可以用不捕捉括号的写法来解决，可这换汤不换药，只不过是把原先的问题换成数算括号对应的数字问题罢了。假如模式中有更多捕获该怎么办呢？

现在不必记忆 $1 这些数字的含义了，Perl 5.10 引入了正则表达式命名捕捉的概念。现在捕捉的结果会进入一个特殊的哈希 %+，其中的键就是在捕捉时候使用的特殊标签，其中的值则是被捕获的串。为捕获串加标签的方法是使用 (?<LABEL>PATTERN) 这样的写法，而 LABEL 可以自行命名【注 17】。这里将第一个捕捉标签定为 name1，而第二个标签则是 name2。使用捕获串时需要访问的位置也变成了 $+{name1} 和 $+{name2}。

```
use 5.010;

my $names = 'Fred or Barney';
if( $names =~ m/(?<name1>\w+) (?:and|or) (?<name2>\w+)/ ) {
    say "I saw $+{name1} and $+{name2}";
    }
```

现在就能看到正确的结果：

```
I saw Fred and Barney
```

一旦使用了捕捉标签，就可以随意移动位置并加入更多的捕获括号，不会因为括号的次序变化导致麻烦：

```
use 5.010;

my $names = 'Fred or Barney';
if( $names =~ m/((?<name2>\w+) (and|or) (?<name1>\w+))/ ) {
    say "I saw $+{name1} and $+{name2}";
    }
```

在使用捕捉标签之后，也给反向引用带来了更新的必要。之前我们使用 \1 或者 \g{1} 这样的写法，现在我们可以使用 \g{label} 这样的写法。

```
use 5.010;

my $names = 'Fred Flinstone and Wilma Flinstone';

if( $names =~ m/(?<last_name>\w+) and \w+ \g{last_name}/ ) {
    say "I saw $+{last_name}";
    }
```

注 17： Perl 也允许用 Python 的语法 (?P<LABEL>...) 完成同样的功能。

我们也可以用另一种语法来表示反向引用。\k<label> 等效于 \g{label}：【注 18】

```
use 5.010;

my $names = 'Fred Flinstone and Wilma Flinstone';

if( $names =~ m/(?<last_name>\w+) and \w+ \k<last_name>/ ) {
    say "I saw $+{last_name}";
    }
```

自动匹配变量

有三个不请自来的捕获变量【注 19】，不必使用捕捉圆括号就能引入。这看来真是不错，但是还有个坏消息，这些变量的名称很奇怪。

虽然 Larry 可能不反对给它们取比较正常的名字，像 $gazoo 或 $ozmidiar。但是，这些都是你在自己的程序里可能会用到的名称。为了让普通的 Perl 程序员在为第一个程序的第一个变量命名时，不必绕开 Perl 所有的特殊变量名【注 20】，Larry 给内置变量起了一些稀奇古怪的名字，也可以说是“惊世骇俗”的名字。在这里用的是标点符号名：$&、$` 和 $'。这些名称奇怪、丑陋且诡异，但它们总得有个名字【注 21】。字符串里实际匹配模式的部分会被自动存进 $& 里：

```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
  print "That actually matched '$&'.\n";
}
```

当上面的程序运行时，会告诉我们字符串里匹配的部分是 " there," （一个空格、一个单词以及一个逗号）。第一个捕获串存储在 $1，是具有 5 个字母的单词 there，但是 $& 里有整个的匹配段落。

注 18： \k<label> 与 \g{label} 有细微的差别。在两个以上的组有同样的标签的时候，\k<label> 和 \g{label} 会引用最早的那组，但是 \g{N} 就可以实现相对反向引用。另外如果是 Python 的爱好者，也可以使用 (?P=label) 的语法。

注 19： 其实没有什么不请自来的好事，自来的意思是它们不需要引号也会生效。别着急，我们会在稍后提及它们的真正代价。

注 20： 你仍然需要避开少数几个传统的变量名（如 $ARGV），不过它们为数不多，也都是全大写的。所有 Perl 的内置变量在 *perlvar* 在线手册里都有介绍。

注 21： 若你真的无法忍受这些奇怪的名称，可以试试 English 模块，它会为 Perl 所有的奇怪变量赋予接近正常的名称。但是，使用这个模块的人一直不多；相反的是，Perl 程序员已经逐渐喜爱这些标点符号变量名，无论它们有多么奇怪。

匹配起始位置之前的字符串会存到 $` 里，而匹配结束位置之后的字符串则存到 $' 里。另外一个解释方法是，$` 保存了正则表达式引擎在找到匹配段落之前略过的部分，而 $' 则保存了字符串中剩下的、从来没有匹配到的部分。如果将这三个字符串依次连接起来，就一定会得到原来的字符串：

```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
  print "That was ($`)($&)($').\n";
}
```

程序运行时会把字符串显示为 (Hello)(there,)(neighbor)，展现出这三个自动匹配变量的实际用途。这个例子看来应该有点眼熟，这是因为在第七章的模式测试程序中有类似的代码，用来显示模式的各个部分匹配的串：

```
print "Matched: |$`<$&>$'|\n";  # 三个自动匹配变量
```

如同前面提到的编号匹配变量，这三个自动匹配变量都有可能是空字符串，它们的有效范围也与编号的匹配变量相同。一般情况下，它们的值会一直持续到下一次模式匹配成功之前。

我们之前提过这三个变量可以自由使用。不过，自由是有代价的。在这里的代价是：一旦你在程序的任何部分使用了某个自动匹配变量，其他正则表达式的运行速度也会变慢【注 22】。这虽不会严重拖慢速度，但足以成为隐患，许多 Perl 程序员干脆永远不碰这些自动匹配变量【注 23】。他们会找出别的办法。例如你可以将整个模式加上括号，然后以 $1 来代替 $&（当然，你可能需要调整模式的捕获编号）。

匹配变量（包括自动匹配变量以及带编号的匹配变量）最常用在替换运算中，这会在下一章介绍。

通用量词

模式中的量词代表前置条目的重复次数。到目前为止，我们已经见过三个量词：*、+ 和 ?。如果这三个量词都不符合需要，你还可以在花括号（{}）里指定重复次数的范围。

注 22：在每个块的进入点与离开点之间，也差不多就是每个地方了。

注 23：虽然大部分的人并没有真的测过他们的程序，以确定这些别的办法是否真的会节省时间，对他们而言，这些变量像是有毒似的。但是我们不能责怪他们不测试速度，许多利用这三个变量的程序一周只会多花费几分钟的 CPU 时间，因此测试速度与进行优化反而会浪费更多的时间。既然如此，为什么要害怕可能额外多出的几毫秒呢？附带一提，Perl 的开发者正在努力解决这个问题，但是在 Perl 6 之前可能不会有解决方案。

因此，模式 /a{5,15}/ 可匹配重复出现 5 到 15 次的字母 a。如果 a 连续出现 3 次，则不匹配，因为次数太少了；若 a 出现 5 次，就会匹配成功；出现 10 次时，也会匹配成功。因为 15 是上限，所以如果 a 出现 20 次，就只有前 15 个会匹配。

如果省略第二个数字但保留逗号，则表示匹配次数没有上限。所以，/(fred){3,}/ 这个模式会匹配重复出现 3 次以上的 fred（在每个 fred 之间不能有空格等额外字符）。因为没有上限，所以即使字符串里有 88 个 fred，也会匹配成功。

如果同时省略逗号与上限次数，那么花括号里的数字就表示一个固定的次数：/\w{8}/ 会匹配正好 8 个字符的单词串（或许是大型字符串里的一部分）。而 /,{5}chameleon/ 所匹配的就是 ,,,,,chameleon。对歌手 George 来说，这是一句很好的歌词：comma comma comma comma comma chameleon。

事实上，我们之前提到的三个量词字符都只是常用的简写而已。星号和量词 {0,} 相同，表示零次或多次。加号相当于 {1,}，表示一次以上。然后，问号也可以写成 {0,1}。因为这三个简写几乎能满足全部的需求，所以平时不太会需要花括号量词。

优先级

看完正则表达式中这一大堆的元字符，你可能会觉得需要一张卡片帮助整理思路。确实有一张优先级表，告诉我们模式中哪些部分的紧密度最高。不像操作符的优先级表，正则表达式的优先级表相当简单，只有 4 个级别。我们顺便再回顾所有 Perl 模式中使用的元字符。表 8-1 可以用来参考。

表 8-1：正则表达式优先级

正则表达式特性	例子
圆括号（分组或者捕获）	(...), (?:...), (?<LABEL>...)
量词	a* a+ a? a{n,m}
锚位和序列	abc ^a a$
择一	a|b|c
元素	a [abc] \d \1

1. 在优先级顶端的是圆括号(?)，用来分组和捕获。括号总是比其他的东西更有黏性。
2. 第二级是量词，也就是重复操作符：星号（*）、加号（+）、问号（?）以及花括号量词，像{5,15}、{3,}与{5}。它们都会与其前面的条目紧密连接。
3. 第三优先级是锚位和序列。锚位包括：脱字符（^）定位字符串开头、美元符号（$）定位字符串结尾、词边界符\b、非词边界\B。虽然没有使用元字符，序列（彼此相邻的条目）事实上也是操作符。也就是说，单词里的字母之间和锚位与字母之间的紧密程度是相同的。
4. 最低的优先级是“择一”竖线（|）。因为是最低一级，它实际上会将模式拆成数个部分。竖线之所以为最低一级，是因为我们希望模式/fred|barney/中，单词字母间的紧密程度高于“择一”竖线。假设“择一”的优先级高于序列，那么这个模式的解释方式就成了“fred后面是d或b，之后再接着arney”。所以，“择一”竖线在优先级表的最底层，这样单词里的字母才会紧密连接在一起。
5. 完全没有优先级的就是那些组成模式的基本元素。包括每个独立的字符、字符集和反向引用。

优先级实例

当你需要解读相当复杂的正则表达式时，你就得照Perl的方式，使用优先级表按部就班地进行分析。

举例来说，/^fred|barney$/大概不会是程序员想要的模式。因为“择一”竖线的优先级比较低，这样整个模式就会被拆成两半。这个模式要不就是匹配字符串开头的fred，要不就是匹配字符串结尾的barney。程序员实际想要的多半是/^(fred|barnet)$/，也就是匹配只包含fred或是只包含barney的每一行【注24】。而/(wilma|pebbles?)/要怎么解释呢？那个问号量词会紧接着前面的字符【注25】，所以这个模式可以匹配到willma、pebbles以及pebble这三个字符串，或是长字符串中的一小部分（因为模式中没有锚位）。

模式/^(\w+)\s+(\w+)$/可用来匹配开始是一个单词、再来是一些空白、然后又是一个单词（前面或后面没有其他东西）的行。举例来说，它大概就是用来匹配fred

注24：也许还包括字符串结尾的换行符，就像之前介绍$锚位时提到的。

注25：因为在pebbles里，问号量词与字母s间的连接要比s与其他字母的连接来得紧密。

flintstone 之类的字符串。这里的圆括号并不是为了分组而存在，可能只是为了把匹配的字符串捕获下来。

在尝试理解一个很复杂的模式时，试着加上一些括号会对弄清优先级有好处。但请记住，圆括号同时也会有捕获的效果。因此建议尽可能用非捕捉的圆括号来分组。

还有更多

尽管我们已经介绍了日常编程中最常见的正则表达式，但是还有些特性仍未提到。请进一步参阅在线手册 *perlre*、*perlquick* 与 *perlretut*，其中有 Perl 模式的延伸介绍【注 26】。

模式测试程序

编写 Perl 程序的时候，每个程序员都免不了要使用正则表达式，但有时候很难轻易看出一个模式能做什么事。而且常常会发现，模式匹配的范围总是比预期的大些或小些。要不就是开始匹配的位置早些或晚些，要不就是根本无法匹配。

下面是一个有用的程序，可用来检测某些字符串是否能被指定的模式匹配：

```
#!/usr/bin/perl
while (<>) {                          # 每次读一行输入
  chomp;
  if (/YOUR_PATTERN_GOES_HERE/) {
    print "Matched: |$`<$&>$'|\n";    # 特殊匹配变量
  } else {
    print "No match: |$_|\n";
  }
}
```

这个模式测试工具是给程序员（而不是给一般用户）使用的，你一看就明白，因为并没有提示符，也没有用法说明。它会把所有输入一行行读进来，然后以你在 YOUR_PATTERN_GOES_HERE 指定的模式进行匹配。如果该行匹配模式，就会利用三个特殊的匹配变量（$`、$& 和 $'）来展示实际的匹配。假设你使用的模式是 /match/，而输入的字符串是 beforematchafter，那么你会看到的程序输出就是 |before<match>after|，尖括号里面的内容是字符串匹配模式的部分。若结果跟你预测的不一样，马上就可以知道。

注 26：请参考 CPAN 上的 YAPE::Regexp::Explain 模块，它会将正则表达式转换成英文。

习题

下列习题中有些会要求你使用本章的模式测试程序。你可以手动输入该程序，小心不要打错任何一个标点符号【注 27】。而我们在序言里就已经提到了，直接从 O'Reily 网站下载程序或其他好东东是多快好省的方法。这个程序的文件名是 pattern_test【注28】。

以下习题答案参见附录 A：

1. [8] 利用模式测试程序，写个模式，使其能够匹配到 match 这个字符串。你可以把 beforematchafter 输入至程序里测试，看是否会正确显示匹配到的部分以及前后的部分？

2. [7] 利用模式测试程序，写个模式，使其能够匹配任何以字母 a 结尾的单词（以 \w 组成的单词）。此模式是否能够匹配到 wilma？是否无法匹配到 barney？此模式是否能够匹配到 Mrs._Wilma_Flintsone？还有 wilma&fred 呢？把上一章习题里的样例文本文件拿到这里测测看（并把这些测试字符串加到该文本文件里）。

3. [5] 修改上题的程序，使其在匹配到以 a 结尾的单词的同时也将其存储在 $1 里。接着修改程序的输出，让变量的内容出现在单引号中，例如：$1 contains 'Wilma'。

4. [5] 修改上题的程序，使用命名捕获而不是 $1 这样的老办法。接着修改程序的输出，让标签名字出现在结果中，例如：'word' contains 'Wilma'。

5. [5] 附加题：修改上一题的程序，使其在定位到以 a 结尾的单词后，再将之后的 5 个字符（如果有那么多的话）捕获至一个独立的内存变量。修改程序输出，把所用到的这两个内存变量都输出来。假设你输入的字符串是 I saw Wilma yesterday，那么后面取到的 5 个字符就是 ␣yest。如果你输入的是 I, Wilma!，那么第二个内存变量的内容只会有一个字符。看看你的模式是否还可以成功匹配 wilma 这个简单的字符串？

6. [5] 写个新程序（不是测试程序），输出其输入中以空白结尾的行（换行符不算）。输出的时候，在行尾多加一些记号，这样比较容易看出空白字符。

注 27：如果你真的要自己键入此程序代码，请记着，反引号（`）与缩写（'）是两个完全不同的字符。全尺寸的键盘上（至少在美式键盘上），反引号位于 1 键的左边。

注 28：如果下载来的程序代码与书中的程序代码有些不同，也不用太惊讶。

第九章

用正则表达式处理文本

正则表达式也可以修改文本。之前我们只是告诉你如何进行模式匹配，但现在要用模式来定位部分字符串，并进行修改。

用 s/// 替换

如果把 m// 模式匹配想象成文字处理器的搜索功能，那么替换功能就是 Perl 的 s/// 替换。此操作符能将指定变量合乎模式的那个部分【注 1】替换为另一个串：

```
$_ = "He's out bowling with Barney tonight.";
s/Barney/Fred/;  # 把 Barney 替换为 Fred
print "$_\n";
```

如果匹配失败，则什么事都不会发生，变量也不受影响：

```
# 接着之前的代码。$_ 现在为 "He's out bowling with Fred tonight."
s/Wilma/Betty/;  # 试图用 Betty 替换 Wilma (将会失败)
```

当然，模式串与替换串还可以更加复杂。下面的替换字符串用到了第一个捕获变量，也就是 $1，模式匹配时会对它赋值：

```
s/with (\w+)/against $1's team/;
print "$_\n";  # 打印 "He's out bowling against Fred's team tonight."
```

这里还有其他可能的替换操作（此处只是举例，在实际场合，不会接连进行这么多互不相关的替换操作）。

注 1：不像 m// 可以匹配任何字符串与表达式，s/// 修改的数据必须是左值。虽然左值从理论上说是任何可以放在赋值操作符左边的东西，不过几乎都是变量。

```
$_ = "green scaly dinosaur";
s/(\w+) (\w+)/$2, $1/;  # 替换后为 "scaly, green dinosaur"
s/^/huge, /;            # 替换后为 "huge, scaly, green dinosaur"
s/,.*een//;             # 空替换，此时为 "huge dinosaur"
s/green/red/;           # 匹配失败：仍为 "huge dinosaur"
s/\w+$/($`!)$&/;        # 替换后为 "huge (huge !)dinosaur"
s/\s+(!\W+)/$1 /;       # 替换后为 "huge (huge!) dinosaur"
s/huge/gigantic/;       # 替换后为 "gigantic (huge!) dinosaur"
```

s/// 返回的是布尔值，替换成功时为真，否则为假：

```
$_ = "fred flintstone";
if (s/fred/wilma/) {
  print "Successfully replaced fred with wilma!\n";
}
```

用 /g 进行全局替换

在前面的例子中你可能注意到了，即使有其他可以替换的部分，s/// 也只会进行一次替换。当然，这只不过是默认的行为而已。/g 修饰符可让 s/// 进行所有可能的、不重复的【注 2】替换：

```
$_ = "home, sweet home!";
s/home/cave/g;
print "$_\n";  # 打印 "cave, sweet cave!"
```

一个相当常见的全局替换是缩减空白，也就是将任何连续的空白转换成单一空格：

```
$_ = "Input  data\t may have    extra whitespace.";
s/\s+/ /g;  # 现在它变成了 "Input data may have extra whitespace."
```

每次只要我们提到如何缩减空白，所有的学生都想知道，如何删除开头和结尾的空白。其实很简单，只要两步：

```
s/^\s+//;  # 删除开头的空白字符
s/\s+$//;  # 删除结尾的空白字符
```

精简到一步的写法则是使用“择一”匹配的竖线符号并配合 /g 修饰符，但这么写其实会运行得稍微慢一点点，至少在编写本书时还是如此。正则表达式的引擎会不断改进，如果要进一步了解如何写出更快（或更慢）的模式，可参阅 Jeffery Friedl 写的《Mastering Regular Expression》（O'Reilly）一书。

```
s/^\s+|\s+$//g;  # 去除开头和结尾的空白字符
```

注 2：不重复是因为每次都会从最近一次替换结束的地方开始新的匹配。

不同的定界符

就像 m// 与 qw// 一样，我们也可以改变 s/// 的定界符。但是，替换运算会用到三个定界符，所以情况有点不同。

对于一般没有左右之分的（非成对）字符，用法便跟使用斜线一样，只要重复三次即可。下面，我们以井号【注 3】作为定界符：

```
s#^https://#http://#;
```

但是，如果使用有左右之分的成对字符，就必须使用两对：一对圈引模式，一对圈引替换字符串。而且在这种情况下，圈引字符串的定界符和圈引模式的定界符不必相同。事实上，字符串甚至可以用非成对的定界符。以下三行的意思相同：

```
s{fred}{barney};
s[fred](barney);
s<fred>#barney#;
```

可选修饰符

不仅是 /g 修饰符【注 4】，替换运算也可以使用我们常在模式匹配中使用的 /i、/x 与 /s 修饰符。（修饰符的顺序对结果没有任何影响）：

```
s#wilma#Wilma#gi;   # 将所有的 WiLmA 或者 WILMA 等一律替换为 Wilma
s{__END__.*}{}s;    # 将 __END__ 标记和其后所有的内容都截掉
```

绑定操作符

就像在说明 m// 时提到的，我们可以用绑定操作符为 s/// 选择不同的目标：

```
$file_name =~ s#^.*/##s;  # 将 $file_name 中所有的 Unix 风格的路径全部去除
```

大小写转换

在替换运算中，常常需要把单词全部改成大写（或是小写）。用 Perl 很容易就能做到，只要使用某些反斜线转义字符就行了。\U 转义字符会将其后的所有字符转换成大写：

注 3：对我们的英国朋友致歉，因为井号是 pound，而 pound 也就是英镑！虽然在 Perl 里，井号一般用于注释的开头，但当解析器知道要等待某个定界符时，井号就不会被视为注释的开头。紧跟在代表开始替换的 s 字符之后的井号，就是其中一个例子。

注 4：即使现在的定界符不是斜线，我们也会将修饰符写成 /i。

```
$_ = "I saw Barney with Fred.";
s/(fred|barney)/\U$1/gi;  # $_ 现在成了 "I saw BARNEY with FRED."
```

同理，\L 转义字符会将其后的字符转换成小写。承前例：

```
s/(fred|barney)/\L$1/gi;  # $_ 现在成了 "I saw barney with fred."
```

默认情况下，它们会影响之后全部的替换字符串。你也可以用 \E 结束大小写转换的影响：

```
s/(\w+) with (\w+)/\U$2\E with $1/i;  # $_ 替换后为"I saw FRED with barney."
```

使用小写形式（\l 与 \u）时，它们只会影响之后的第一个字符：

```
s/(fred|barney)/\u$1/ig;  # $_ 替换后为 "I saw FRED with Barney."
```

你甚至可将它们并用。你可以同时使用 \u 与 \L 来表示全部转小写，但首字母大写【注 5】：

```
s/(fred|barney)/\u\L$1/ig;  # $_ 现在成了 "I saw Fred with Barney."
```

附带一提，虽然这里介绍的是替换时的大小写转换，但它们也适用于任何双引号内的字符串：

```
print "Hello, \L\u$name\E, would you like to play a game?\n";
```

split 操作符

另外一个使用正则表达式的操作符是 split，它会根据分隔符拆开一个字符串。这对处理被制表符、冒号、空白或任意符号分隔的数据相当有用【注 6】。只要你能将分隔符写成模式（通常是很简单的正则表达式），就可以使用 split 提取数据。它的用法如下：

```
@fields = split /separator/, $string;
```

split 操作符【注 7】用拆分模式串“扫过”指定的字符串，并返回字段（也就是子串）列表。期间只要模式在某处匹配成功，该处就是一个字段的结尾、下一个字段的开头。

注 5：\L 与 \u 哪一个放前面都可以。Larry 了解到有人可能会把它们颠倒，所以他让 Perl 能够理解你想要首字母大写然后其余小写。他真是个好心人。

注 6：以逗号分隔的字段（一般称为 CSV 文件）除外。用 split 处理 CSV 文件是很痛苦的事，最好还是使用 CPAN 的 Text::CSV 模块吧。

注 7：它是一个操作符，虽然它的行为看来像一个函数，而且大家通常都会称它为函数。不过，技术细节上的差异已超出了本书的范围。

所以，任何匹配模式的内容都不会出现在返回字段中。下面就是以冒号作为分隔符的典型 split 模式：

```
@fields = split /:/, "abc:def:g:h";  # 得到 ("abc", "def", "g", "h")
```

如果两个分隔符连在一起，就会产生空字段：

```
@fields = split /:/, "abc:def::g:h";  # 得到 ("abc", "def", "", "g", "h")
```

这里有个规则，它乍看之下很古怪，但很少造成问题：split 会保留开头处的空字段，并省略结尾处的空字段。例如【注 8】：

```
@fields = split /:/, ":::a:b:c:::";  # 得到 ("", "", "", "a", "b", "c")
```

利用 /\s+/ 模式进行空白分隔也是常见的做法。在此模式下，所有的空白会被当成一个空格来处理：

```
my $some_input = "This  is a \t        test.\n";
my @args = split /\s+/, $some_input;  # ("This", "is", "a", "test.")
```

split 默认会以空白字符分割 $_：

```
my @fields = split;  # 等效于 split /\s+/, $_;
```

这几乎就等于以 /\s+/ 为模式，只是它会省略开头的空字段。所以，即使该行以空白开头，也不会在返回列表的开头处看到空字段。（若你想以这种方式来分解用空格分隔的字符串，则可以用一个空格来作为模式：split ' ',$other_string。用一个空格来作为模式是 split 的特殊用法。）

一般来说，用来 split 的模式就像之前看到的这样简单。但是如果你用到更复杂的模式，请避免在模式里用到捕获圆括号，因为这会启动所谓的“分隔保留模式”（详情请参考 *perlfunc* 在线手册）。在 split 里使用非捕捉圆括号 (?:) 的写法，就可以安全地进行分组。

join 函数

join 函数不会使用模式，它的功能与 split 恰好相反：split 会将字符串分解为数个片段（子字符串），而 join 则会把这些片段联合成一个字符串，它的用法如下所示：

```
my $result = join $glue, @pieces;
```

注 8：这只是默认的行为。这么做是为了提高效率。若你担心失去结尾处的空字段，只要以 -1 作为 split 的第三个参数就能保留它们了。相关细节请参阅 *perlfunc* 在线手册。

你可以把 join 的第一个参数理解为胶水，它可以是任何字符串。其余的参数则是一串片段。join 会把胶水涂进每个片段之间，并返回结果字符串：

```
my $x = join ":", 4, 6, 8, 10, 12;  # $x 为 "4:6:8:10:12"
```

上例中，我们有5个条目，所以只有4个冒号。也就是说，有4层胶水。胶水只在两个片段中间出现，不在之前也不在其后。所以，胶水的层数会比列表中的条目少一个。

换句话说，列表至少要有两个元素，否则胶水无法涂进去：

```
my $y = join "foo", "bar";        # 只有一个 "bar"，这里不会起作用
my @empty;                        # 空数组
my $empty = join "baz", @empty;   # 没有元素，所以得到一个空的字符串
```

使用上面的 $x，我们可以先分解字符串，再用不同的定界符将它接起来：

```
my @values = split /:/, $x;  # @values 为 (4, 6, 8, 10, 12)
my $z = join "-", @values;   # $z 为 "4-6-8-10-12"
```

虽然 split 与 join 合作无间，但是请别忘了，join 的第一个参数是字符串，而不是模式。

列表上下文中的 m//

使用 split 的时候，模式指定的分隔符并非真正有用的数据字段。但有时候，指定想要留下的部分反而更简单。

在列表上下文中使用模式匹配操作符（m//）时，如果模式匹配成功，那么返回的是所有捕获变量的列表；如果匹配失败，则会返回空列表：

```
$_ = "Hello there, neighbor!";
my($first, $second, $third) = /(\S+) (\S+), (\S+)/;
print "$second is my $third\n";
```

如此就能给那些（可在下一次模式匹配后访问的）匹配变量起既好记又动听的名字。另外因为程序代码中并未用到 =~ 绑定操作符，所以该模式匹配是针对 $_ 进行的。

你之前在 s/// 的例子中看到的 /g 修饰符，同样也可以用在 m// 操作符上，其效果就是让模式能够匹配到字符串中的许多地方。下面的例子中具有一对圆括号的模式，它会在每次匹配成功时返回一个捕获串：

```
my $text = "Fred dropped a 5 ton granite block on Mr. Slate";
my @words = ($text =~ /([a-z]+)/ig);
print "Result: @words\n";
# 打印：Result: Fred dropped a ton granite block on Mr Slate
```

这就好像自己动手实现了 split 的功能。不过并非指定想要去除的部分，反而是指定想要留下的部分。

如果模式中有多对圆括号，那么每次匹配就能捕获多个串。假设，我们想把一个字符串变成哈希，就可以这样做：

```
my $data = "Barney Rubble Fred Flintstone Wilma Flintstone";
my %last_name = ($data =~ /(\w+)\s+(\w+)/g);
```

每次模式匹配成功，就会返回一对被捕获出来的值。这一对值正好成为新哈希的键／值对。

更强大的正则表达式

读过三章关于正则表达式的内容之后，你现在已经知道，正则表达式已经成为深入 Perl 内核的强大功能。而 Perl 开发人员对它加入了更多的功能，你会在接下来的一节看到其中最重要的那些功能。同时你也会看到正则表达式引擎更有趣的内部运作机制。

非贪婪量词

目前为止，我们（在第七章和第八章）看到的 4 个量词全都是贪婪量词。也就是说在保证整体匹配的前提下，它们会尽量匹配长字符串，实在不行才会吐出一点。举例来说：假设你以 /fred.+barney/ 来匹配 fred and barney went bowling last night 这个字符串。我们可以一眼就看出来会匹配成功，但是现在我们要深入了解一下匹配的过程中到底发生了什么事情【注 9】。首先，模式中的 fred 部分将会逐字匹配与其相同的字符串。模式的下个部分是 .+，它会匹配换行符之外的所有字符（至少一次）。但加号量词（+）是个贪婪的量词，它会尽量匹配最多的字符串。所以，进行到此，它会一口气吞掉字符串剩下的部分，一路到最后的 night。（看到这里你大概觉得要出什么意外了，不过故事还没讲完呢。）

现在轮到模式中的 barney 部分，但是它已经没办法进行匹配，因为刚才已经进行到字符串的最后面。此时，.+ 模式就会很不情愿的吐出一个字符，反正就算少了一个字符，这部分的模式还算是匹配成功。（虽然它很贪婪，不过为了顾全大局、让整体匹配都成功，就算自己没有匹配到全部的字符串也可以忍受。）

注 9：正则表达式引擎会进行一定程度的优化，使得事实与这里描述的细节并不完全一样，而优化的方式也随着 Perl 的版本而不断改进。但就功能而言，你不会感觉到事实与描述的差异。如果你想知道事情的真相，那么你得读最新版的源代码。如果找到缺陷，请提交补丁。

现在又轮到 barney 部分进行匹配，但还是无法成功。因此，.+ 再次吐出一个字符试试看。就这样一个字符一个字符地，.+ 匹配的部分一路减少到了 barney 之前。此刻，模式中的 barney 部分终于能够匹配成功，于是整个模式也就匹配成功了。

正则表达式引擎会一直进行上述的回溯动作，不断地调整模式匹配的内容来适应字符串，直到终于找到一个整体匹配成功的方案为止，要是直到最后都没找到就宣告失败【注10】。从这个例子我们可以知道回溯的动作可能非常繁琐，因为量词囫囵吞下太长的字符串，于是正则表达式引擎不得不使它吐出许多来。

然而，对于每一个贪婪的量词，都有一个非贪婪的量词。以加号（+）为例，我们可以改用非贪婪的量词 +?，这表示一次或更多匹配（加号的本意）。但是匹配的字符串却是越短越好，而不再是越长越好。现在我们把刚才的模式改成 /fred.+?barney/，再来看看这个新学到的量词是如何运作的。

模式中的 fred 部分还是一样。但这次模式中的下一个部分换成了 .+?，它会匹配到一个以上的字符，但是越短越好，也就是最好一个字符，所以它匹配的部分是 fred 后面的空白字符。接下来出现的模式是 barney，但就目前的位置而言，匹配会失败（因为目前要匹配的部分是 and barney 的 and）。于是 .+? 模式又很不情愿地多匹配了一个字符 a，然后把控制权交给之后的模式重试。但 barney 还是匹配失败，所以 .+? 只好再吞下一个字符 n，就这样一直进行下去。直到 .+? 匹配了5个字符之后，barney 模式终于能够匹配成功，于是整个模式也就匹配成功了。

其实还是有一些回溯的动作，但是这次正则表达式引擎的回溯次数比较少了，在速度上应该是大有改善才对。不过呢，这是在 barney 跟 fred 都离得很近的情况下才会成立。如果要处理的数据都是 fred 在字符串开始处，barney 在字符串尾，那么选用贪婪的量词反而会比较快。所以最终的速度其实取决于正则表达式处理的数据。

非贪婪的量词并不只跟效率有关。尽管只要贪婪版本可以成功匹配的字符串，它们也同样可以匹配成功（匹配失败的情况亦然），但是它们匹配的字符串长度是不同的。举例来说，假设你手边有一些类似 HTML 的【注11】文本，而你需要去除 <BOLD> 跟 </BOLD> 这样的标记，并保留剩余的内容。如果要处理的字符串是这样：

注10：有些正则表达式引擎会尝试每种可能，就算已经确定匹配成功，也会继续找下去。不过 Perl 的正则表达式引擎只专注于模式是否能够匹配成功，所以只要找到一个成功匹配的就算成功。这些问题也请参考《Mastering Regular Expressions》。

注11：再次说明，并不是真的 HTML，因为你无法用简单的正则表达式来解析 HTML。如果你真的需要处理 HTML 或类似的标记语言，请使用 CPAN 上的相关模块，让它们来处理所有复杂的事项。

```
I' m talking about the cartoon with Fred and <BOLD>Wilma</BOLD>!
```

那么可以用下面这个替换表达式把标记去掉。但这会有什么问题呢？

```
s#<BOLD>(.*)</BOLD>#$1#g;
```

问题就出在星号量词太贪心了【注 12】。如果是下面这个字符串该怎么办？

```
I thought you said Fred and <BOLD>Velma</BOLD>, not <BOLD>Wilma</BOLD>
```

在此情况下，该模式就会从第一个 <BOLD> 匹配直到最后一个 </BOLD>，把这中间的部分全部取出来。这就错了，我们这里要用的其实是非贪婪的量词。非贪婪版本的星号是 *?，所以新的替换表达式应该改写成这样：

```
s#<BOLD>(.*?)</BOLD>#$1#g;
```

这么一来就正确无误了。

看过非贪婪的加号 +? 与非贪婪的星号 *? 之后，你大概可以举一反三，猜出另外两个非贪婪的量词要怎么写。非贪婪的花括号量词看来也一样，但问号是加在右括号后面，写成 {5,10}? 或者 {8,}? 就行【注 13】。就连问号量词也有非贪婪的版本：??。虽然这还是一样会匹配到一次或零次，但会优先考虑零次的情况。

跨行的模式匹配

传统的正则表达式都是用来匹配单行文本。由于 Perl 可以处理任意长度的字符串，其模式匹配也可以处理多行文本，与处理单行文本并无差异。当然了，先得有表达式可以表示多行文本才行。下面的写法可以表示 4 行的文本：

```
$_ = "I'm much better\nthan Barney is\nat bowling,\nWilma.\n";
```

^ 和 $ 通常是用来匹配整个字符串的开始和结束的（参考第八章）。但是当模式加上 /m 修饰符之后，就可以让它们也匹配串内的换行符（把 m 当成 multiple lines 会比较好记）。这样一来，它们所代表的位置就不再是整个字符串的头尾，而是每行的开头跟结尾。因此，下面的模式就成立了：

```
print "Found 'wilma' at start of line\n" if /^wilma\b/im;
```

注 12：还有一个可能会发生的问题：此处应该加上 /s 修饰符，因为结束标记可能直到后面好几行才会出现。好在此处只是个例子。如果我们正在设计产品，那我们就会遵照我们自己的建议，找个质量较高的模块来用。

注 13：理论上，即使次数是个明确的数字，也会存在非贪婪的量词，例如 {3}?。但由于已经直接讲明了次数，所以就没有了贪婪与非贪婪的差别了。

同样地，你也可以对多行文本逐个进行替换。接下来的程序会先把整个文件读进一个变量【注 14】，然后把文件名前置于每一行的开头：

```
open FILE, $filename
  or die "Can't open '$filename': $!";
my $lines = join '', <FILE>;
$lines =~ s/^/$filename: /gm;
```

一次更新多个文件

程序化地更新文件内容时，最常见的做法就是先打开一个新文件，然后把跟旧文件相同的内容写进去，并且在需要的位置进行改写。后面会看到，这样做和直接更新文件的做法效果大致相同，只是有些附带的好处。

举例来说，我们现在有几百个格式类似的文件。其中一个叫做 `fred03.dat`，里面都是如下几行的内容：

```
Program name: granite
Author: Gilbert Bates
Company: RockSoft
Department: R&D
Phone: +1 503 555-0095
Date: Tues March 9, 2004
Version: 2.1
Size: 21k
Status: Final beta
```

我们必须修改这个文件，让它有一些新的信息。下面就是应该改成的样子：

```
Program name: granite
Author: Randal L. Schwartz
Company: RockSoft
Department: R&D
Date: June 12, 2008 6:38 pm
Version: 2.1
Size: 21k
Status: Final beta
```

简单地说，我们要做三项改动：`Author` 字段的姓名要改，`Date` 要改成今天的日期，而 `Phone` 则要删除。另外还有几百个文件也都要进行这些改变。

要在 Perl 中直接修改文件内容可以使用钻石操作符（<>）。虽然一眼可能看不出来，但下面的程序确实可以完成刚才的要求。此程序的新意在于特殊变量 `$^I`。现在我们先跳过它，等一下再说明：

注 14：当然希望这是一个小文件。虽然变量可以很大。

```
#!/usr/bin/perl -w

use strict;

chomp(my $date = `date`);
$^I = ".bak";

while (<>) {
  s/^Author:.*/Author: Randal L. Schwartz/;
  s/^Phone:.*\n//;
  s/^Date:.*/Date: $date/;
  print;
}
```

因为我们需要今天的日期，所以这个程序一开始就使用了系统的 *date* 命令。另外一个比较好的做法就是在标量上下文中使用 Perl 自己的 `localtime` 函数（但两者的日期格式稍有差异）：

```
my $date = localtime;
```

下一行则是对 `$^I` 变量赋值，不过我们现在先跳过不看。

钻石操作符会读取命令行参数指定的那些文件。程序的主循环一次会读取、更新及输出一行（以之前学到的知识来推断，你没准觉得经过修改的内容会飞快地在终端上滚过，而本来的文件却没修改。不过请耐心地看下去）。注意第二个正则替换是把电话号码那一行换成空字符串，连换行符也一起去掉。所以到了要输出的时候，其实什么都不会输出，就好像从来就没有出现过 `Phone` 这个字段一样。由于大部分的输入行都不会匹配这三个模式，所以它们在输出的时候都不会有任何变动。

这样的结果跟我们想要的已经相差不远了，但是我们还没有告诉你更新过的内容是如何写回文件的。这个问题的答案就在 `$^I` 这个变量中。这个变量的默认值是 `undef`，这不会对程序造成任何影响。但如果将其赋值成某个字符串，钻石操作符就会具有比平常更多的魔力。

对于钻石操作符的魔力我们已经知道不少：它会自动帮你打开许多文件，而且如果没有指定文件，它就会从标准输入读进数据。但如果 `$^I` 中是个字符串，该字符串就会变成备份文件的扩展名。现在我们来看看这是如何运作的。

先假设钻石操作符正好打开了文件 `fred03.dat`。除了像以前一样打开文件之外，它还会把文件名改成 `fred03.dat.bak`【注 15】。虽然打开的是同一个文件，但是它在

注 15：在非 Unix 系统上实现细节可能有些不同，但结果应该都是一样的。详见你的 Perl 版本的注意事项。

磁盘上的文件名已经不同了。接着，钻石操作符会打开一个新文件并将它取名为 `fred03.dat`。这么做并不会有任何问题，因为我们已经没有同名文件了。现在钻石操作符会把默认的输出设定为这个新打开的文件，所以输出来的所有内容都会被写进这个文件【注 16】。这样 `while` 循环会从旧文件读进一行输入，做了一些改动之后把新的内容写进新文件。在普通的机器上，这样的程序可以在几秒内修改上百个文件。够厉害吧？

所以程序结束后，用户会发现什么呢？他们会说“喔，我懂了。Perl 根据我的需要编辑了 `fred03.dat` 文件的内容，而且好心地把原始拷贝备份到 `fred03.dat.bak` 文件”。不过我们知道真相，Perl 并没有编辑任何文件，它只是创建了一个修改过的新拷贝。趁我们还在盯着他那冒烟的魔术师手杖看的时候，把文件偷偷调了包。真是高明呀！

有些人会把 `$^I` 的值设为 ~ 这个字符，因为 *emacs* 在处理备份文件的文件名时也是这么做。而如果把 `$^I` 设为空字符串，就会直接修改文件的内容，但不会留下任何备份。只要模式中不小心打错一个字，就可能会把整份数据全都清空，所以如果你真的想看看备份磁带质量如何，就尽情的用空字符串吧！全部确认无误之后再把备份文件删除是轻而易举的事。如果做错了，则需要把已备份的文件还原回来，大概你已经想到可以用 Perl 来做这件事了（第十四章中有文件批量改名的例子）。

从命令行进行在线编辑

前一节那样编程修改文件的程序已经很简单了。不过 Larry 认为那还不够简单。

假设你需要更新上百个文件，把里面拼错成 `Randall` 的名字改成 `Randal`。你可以写个像前一节那样的程序来完成此事。或者，你也可以在命令行上使用下面的单行程序做到：

```
$ perl -p -i.bak -w -e 's/Randall/Randal/g' fred*.dat
```

Perl 的命令行选项设计非常巧妙，让你只用极少的按键就能建立一个完整的程序【注17】。我们先来看看它们的用处。

注 16：钻石操作符也会尽可能复制原文件的使用者权限以及拥有者设定。例如，如果本来的文件是所有用户皆可读取，那么新的文件也应该如此。

注 17：参考 *perlrun* 在线手册了解全部的选项。

以 perl 开头的命令的作用如同在文件的开头写上 #!/usr/bin/perl:表示以 perl 程序来处理随后的脚本。

-p 选项则可以让 Perl 自动生成一小段程序，看起来类似如下的片段【注 18】：

```
while (<>) {
  print;
}
```

如果不需要这么多功能，还可以用 -n 选项，这样就可以把自动执行的 print 去掉，所以你可以自行决定什么内容需要 print（*awk* 迷们应该认得 -p 与 -n）。这点细微差别对大程序来说无关轻重，但是对于保护键盘和手指来说非常重要。

下一个出现的选项是 -i.bak，其作用就是在程序开始运行之前把 $^I 设为 .bak。如果你不想做备份，请直接写出 -i，不要加扩展名。如果你不带备用的降落伞，那就带上备用飞机吧。

之前已经介绍过 -w 选项了，它能打开警告。

选项 -e 用来告诉 Perl 后面跟着的是程序代码。也就是说，s/Randall/Randal/g 这个字符串会被直接当成 Perl 程序代码。因为目前我们已经有个 while 循环了（来自 -p 选项），所以这段程序代码会被放到循环中 print 前面的位置。基于一些技术上的原因，用 -e 选项指定的程序中可以省略末尾的分号。如果你指定了多个 -e 选项，就会有多段程序代码，此时只有最后一段程序末尾的分号可以省略。

最后一个命令行参数是 fred*.dat，表示 @ARGV 的值应该是匹配此文件名模式的所有文件名。把以上所有片段全都组合在一起，就好像写了下面这个程序，并且用 fred*.dat 这个参数调用它一样：

```
#!/usr/bin/perl -w

$^I = ".bak";

while (<>) {
  s/Randall/Randal/g;
  print;
}
```

把这个程序与我们上一节使用的程序相比，就会发现这两者十分相似。这些命令行选项还挺管用的，不是吗？

注 18：其实 print 出现在 continue 块中。进一步的信息请参阅 *perlsyn* 与 *perlrun* 在线手册。

习题

以下习题答案参见附录 A：

1. [7] 建立一个模式，无论 $what 的值是什么，它都可以匹配三个 $what 的内容连在一起的字符串。也就是说，如果 $what 的值是 fred，那么你的模式应该匹配 fredfredfred；若 $what 的值为 fred|barney，那么你的模式应该匹配 fredfredbarney、barneyfredfred、barneybarneybarney 或许多其他组合。（提示：你应该在模式测试程序的开头，放上类似 my $what = 'fred|barney'; 这样的语句。）

2. [12] 写个程序来复制并修改指定的文本文件。在副本里，此程序会把出现字符串 Fred（大小写不计）的每一处都换成 Larry（也就是 Manfred Mann 换成 ManLarry Mann）。输入文件名应该在命令行上指定（不询问用户），输出文件名则是本来的文件名加上 .out。

3. [8] 修改前一题的程序，以便把所有的 Fred 换成 Wilma 并把所有的 Wilma 换成 Fred。如果输入的是 fred&wilma，那么正确的输出应是 Wilma&Fred。

4. [10] 附加题：写个程序，把你目前写过的所有程序都加上版权声明，也就是加上一行这样的文字：

   ```
   ## Copyright (C) 20XX by Yours Truly
   ```

   把它放在第一行的 #! 之后。你应该直接修改文件内容并且做备份。假设你将在命令行指定待修改的文件名。

5. [15] 额外附加题：修改前一题程序里的模式，如果文件里已经有版权声明，就不再进行修改。提示：你可能需要知道钻石操作符当前正在读取的文件名称，可以在 $ARGV 里找到。

第十章

其他控制结构

在这一章中，我们将会看到另外一些写 Perl 程序的方法。这些技巧通常不会提升 Perl 语言本身的威力，但是能更加轻松地解决问题。你不一定要在自己的程序中使用这些技巧，不过可别就此小看这一章，迟早在别人的程序代码中你会看到这些控制结构。事实上，在这本书中你就一定能看到这些技巧的应用实例。

unless 控制结构

在 `if` 控制结构中，只有当条件表达式为真的时候才执行某块代码。如果你想让程序块在条件为假的时候才执行，请把 `if` 改成 `unless`。

```
unless ($fred =~ /^[A-Z_]\w*$/i) {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

使用 `unless` 意味着，要么条件为真，要么执行某块代码。这就好像使用 `if` 控制结构来判断相反的条件。另一种说法是它类似于独立的 `else` 子句。也就是说，当看不懂某个 `unless` 语句时，你可用如下的 `if` 代替：

```
if ($fred =~ /^[A-Z_]\w*$/i) {
  # 什么都不做
} else {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

这么做与运行效率高低无关，两种写法应该会被编译成相同的内部字节码。另外一个改写的方法，就是以取反操作符 ! 来否定条件：

```
if ( ! ($fred =~ /^[A-Z_]\w*$/i) ) {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

通常应该选择最容易理解的方法来写代码，因为这通常对于维护程序员来说也是最容易理解的。如果用 if 来表达最合适，那么就这么写也行。但是更多的情况下使用 unless 能使你的表达更加自然。

unless 附带的 else 子句

其实哪怕是在 unless 结构中也可以使用 else 语句，虽然支持这样的语法，但是可能会导致困惑：

```
unless ($mon =~ /^Feb/) {
  print "This month has at least thirty days.\n";
} else {
  print "Do you see what's going on here?\n";
}
```

或许有些人会这么写，特别是当第一个子句相当短（也许只有一行）而第二个子句却有多行的时候。不过要是让我们来写，就会将它换成取反的 if 语句，或者干脆对调两个代码块成为一个普通 if 控制结构。

```
if ($mon =~ /^Feb/) {
  print "Do you see what's going on here?\n";
} else {
  print "This month has at least thirty days.\n";
}
```

请时刻记住，代码的读者起码分为两类：机器会执行代码；而人会维护代码。如果人都无法理解你写的程序，那么迟早机器也会做错事情。

Until 控制结构

有时也许会想要颠倒 while 循环的条件。那么，请使用 until：

```
until ($j > $i) {
  $j *= 2;
}
```

这个循环会一直执行，直到条件为真。它只不过是个改装过的 while 循环罢了，两者之间唯一的差别在于 until 会在条件为假（而非真）时重复执行。因为条件判断发生在循环第一次迭代之前，所以它仍旧是一个执行零次以上的循环，和 while 循环

一样【注1】。类似 if 和 unless 转化的例子，你可以用否定条件的方法将任何 until 循环改写成 while 循环。不过通常还是建议在更简单和自然的情况下使用 until。

条件修饰词

为了进一步简化表达，表达式后面可以接着一个用来控制它的修饰词。例如用 if 修饰词来模拟一个 if 块：

```
print "$n is a negative number.\n" if $n < 0;
```

这能达到和下面代码完全相同的效果，只是省去了键入圆括号和花括号的必要【注 2】：

```
if ($n < 0) {
  print "$n is a negative number.\n";
}
```

之前曾经提到，那些使用 Perl 的家伙都懒于打字。其实这个更短的版本也更容易用英语读出来：print this message if $n is less than zero。

注意即使条件写在后面，它仍然会先执行。这与通常由左至右的顺序相反。阅读 Perl 代码的方法就是学习 Perl 解释器的内部工作原理，先把语句全部读完再判断其含义。

还有其他的修饰词：

```
&error("Invalid input") unless &valid($input);
$i *= 2 until $i > $j;
print " ", ($n += 2) while $n < 10;
&greet($_) foreach @person;
```

以上的写法大都能如你所愿地工作，起码我们希望如此。换句话说，上面每一行都可以效仿 if 修饰词进行改写。例如：

```
while ($n < 10) {
  print " ", ($n += 2);
}
```

值得注意的是，在 print 参数列表中，圆括号里的表达式会将 $n 加 2 并且将结果存回 $n，而最新的值被返回并打印出来。

注 1：Pascal 程序员请注意：你们的 repeat until 循环至少会执行一次，但是 Perl 的 until 循环可能完全不执行，因为条件判断是在每次循环执行之前。

注 2：当然其实还省去了换行符。但是要注意花括号其实还有引入新变量的功能。在某些时候这很有用，要进一步了解请参考说明文档。

这些简写形式读起来像自然语言：调用 &greet 子程序问候 @person 的每个成员。倍增 $i 直到它大于 $j【注3】。这些修饰词常见用法之一，就是像这样的语句：

```
print "fred is '$fred', barney is '$barney'\n"          if $I_am_curious;
```

用这种倒装句编写程序，可以把语句中重要的部分放在前面。上面那个语句的重点是查看一些变量的值，而不是检查是否好奇【注4】。有些人喜欢将整个语句写成一行，也可能在 if 之前加上些制表符使它向右边缩进一些，上面那个例子就是如此。也有人喜欢将 if 修饰词放在下一行并缩进一些：

```
print "fred is '$fred', barney is '$barney'\n"
    if $I_am_curious;
```

虽然这些修饰词表达式都可以用块的形式重写，也就是用最传统的方法写。但是另一个方向的重写却不一定成功。修饰词的两边都只能写单个表达式，因此不能写某事 if 某事 while 某事 until 某事 unless 某事 foreach 某事，因为那样太让人困惑了。另外修饰词的左边也不能放多条语句，如果确实需要，还是建议你回到传统做法，仍然写那些圆括号和花括号。

如同我们之前在 if 修饰词那里谈到的，右边的控制表达式总是先求值，和老式写法执行顺序是一样的。

在使用 foreach 修饰词的时候无法自选控制变量，必须得使用 $_。这通常不是问题，不过若真需要自选控制变量，可以用老式的 foreach 循环重写。

裸块控制结构

所谓的裸块就是没有关键字或条件的代码块。比如现在有一个 while 循环，如下所示：

```
while (condition) {
  body;
  body;
  body;
}
```

注3： 起码我们是非常喜爱它们的。

注4： 当然，$I_am_curious 这个变量是我们杜撰出来的，并非内建的 Perl 变量。通常使用这个技巧的时候，变量会命名为 $TRACING，或者用 constant 编译命令来声明。

然后拿走关键字 while 和条件，就会得到一个裸块：

```
{
  body;
  body;
  body;
}
```

裸块就像一个 while 或 foreach 循环，可是它从不循环，只执行一次。也就是一个伪循环！

稍后就能看到裸块的一些其他应用，这里先看它是如何为临时词法变量圈定有效范围的：

```
{
  print "Please enter a number: ";
  chomp(my $n = <STDIN>);
  my $root = sqrt $n;  # 计算平方根
  print "The square root of $n is $root.\n";
}
```

这个块中的 $n 和 $root 都是限于局部访问的临时变量。一个关于局部变量的准则是：最好把变量声明在最小的范围之内。如果某个变量只会在几行代码里使用，你可以把这几行放到一个裸块里，并就近声明变量。当然，如果我们稍后还需要用到 $n 或者 $root，便需要在更大范围中声明这些变量。

你可能已经注意到这里的 sqrt 函数很陌生。没错，那是一个我们不曾见过的函数。Perl 有许多内置函数无法在本书介绍，请查阅 *perlfunc* 在线手册自行学习。

elsif 子句

许多情况下，你需要逐项检查一系列的条件表达式，看看其中哪个为真。这可以通过 if 控制结构的 elsif 子句来写成如下代码：

```
if ( ! defined $dino) {
  print "The value is undef.\n";
} elsif ($dino =~ /^-?\d+\.?$/) {
  print "The value is an integer.\n";
} elsif ($dino =~ /^-?\d*\.\d+$/) {
  print "The value is a _simple_ floating-point number.\n";
} elsif ($dino eq '') {
  print "The value is the empty string.\n";
} else {
  print "The value is the string '$dino'.\n";
}
```

Perl 会一个接一个地测试这些条件。当其中某项符合时，相应的程序代码块就会执行，然后整段代码结束【注5】，并执行接下来的程序代码。如果没有任何一项符合，则执行最末端的 else 块。else 子句无疑可以省略，但是在这个例子里面最好保留。

elsif 子句的数量并没有限制，但别忘了 Perl 必须执行前面的 99 个失败的测试，才会到达第一百个。如果要写十几个 elsif，不如考虑使用更加有效的方式来编写。看看 *perlfaq* 列出的关于如何模拟 case 或 switch 的常见问题，Perl 5.10 或者更高版本的用户还可以选择使用第十五章中介绍的 given-when。

你可能已经注意到了，这个关键字的拼写居然是 elsif，好像缺少了一个 e。但是如果你写成具有两个 e 的 elseif，Perl 会告诉你拼写错误。为什么呢？因为 Larry 说了算【注6】。

自增和自减

编程中常常需要对标量进行自增或自减。因为它们经常出现，所以就像其他常用的表达式一样，有相应的简写。

使用自增操作符 ++ 能对标量加 1，好像 C 一类的语言：

```
my $bedrock = 42;
$bedrock++;  # $bedrock 加1，变成 43
```

跟其他将变量加 1 的方法一样，标量若未定义将会被创建：

```
my @people = qw{ Fred barney Fred wilma dino barney Fred pebbles };
my %count;                          # 新的空哈希
$count{$_}++ foreach @people;  # 按需要创建新的键／值对
```

第一次处理 foreach 循环时，$count{$_} 会自增。先是 $count{"fred"}，因此它会从 undef 变成 1，因为之前这个哈希值不存在。下一次执行循环时，$count{"barney"} 会变成 1；在这之后，$count{"fred"} 会变成 2。每次处理循环时，%count 中的某个元素就会自动递增，当然也有可能被创建。在整个循环完成

注5：不像 C 语言的 switch 结构那样会接着执行下一个块。

注6：事实上，他坚持这种拼法，甚至拒绝融入另一种拼法的建议。如果你坚持要拼写另一个 e 也很简单：第一步，发明你自己的语言；第二步，让语言非常流行。如果程序语言是你自己发明的，关键字要怎么拼是你的自由。我们希望你的语言不要拒绝 elseunless 这样的建议。

后 $count{"fred"} 的值应该是 3。这是个快速而简易的方法，可用来检查列表中有哪些元素，并计算每个元素出现了几次。

类似的，自减操作符（--）从标量中减去 1。

```
$bedrock--;  # $bedrock 减1，又变回 42 了
```

自增的值

可以获得变量的值并且同时进行修改。把 ++ 操作符写在变量之前就能先增加变量的值，然后获取变量的值。这是前置自增：

```
my $m = 5;
my $n = ++$m;  # $m 增加到 6，并把该值赋给 $n
```

或者把 -- 操作符放在变量之前获取其值。这是前置自减：

```
my $c = --$m;  # $m 减少到 5，并将该值赋给 $c
```

现在是技巧性的部分。将变量名称放在前面会先取得值，之后再自增或是自减。这种做法称为后置自增或后置自减：

```
my $d = $m++;  # $d 得到的是 $m 之前的值 (5)，然后 $m 增加到 6
my $e = $m--;  # $e 得到的是 $m 之前的值 (6)，然后 $m 减少到 5
```

之所以有趣，是因为这里同时做了两件事。我们在同一个表达式中取值并且修改它的值。如果操作符在前，就会先自增（或是自减），然后使用新值；如果变量在前，就会先返回其原来的值，然后再自增或自减。另外一种说法是，这些操作符会返回某个值，但是它们具有修改变量值的副作用。

如果你写的表达式中只有自增或自减【注 7】，不使用值而只是利用副作用，前置后置将没有任何区别【注 8】：

```
$bedrock++;  # $bedrock 加1
++$bedrock;  # 同样，$bedrock 加1
```

这类操作符的一个常见用法就是（配合哈希计数）判断之前已见过的元素：

```
my @people = qw{ Fred barney bamm-bamm wilma dino barney betty pebbles };
my %seen;
```

注 7：也就是空上下文。

注 8：写过编译器的人可能会猜测，后置自增及后置自减是否会比它们的前置版本要慢，但是 Perl 并不会如此机械。在空上下文中，Perl 会自动对后置形式进行优化。

```
foreach (@people) {
  print "I've seen you somewhere before, $_!\n"
    if $seen{$_}++;
}
```

当 barney 第一次出现的时候，$seen{$_}++ 的值为假，因为 $seen{$_} 的值也就是 $seen{"barney"} 的值是 undef。不过由于这个表达式具有将 $seen{"barney"} 递增的副作用，所以当 barney 再次出现的时候，$seen{"barney"} 的值就是真，可以被打印了。

for 控制结构

Perl 的 for 控制结构类似其他语言（如 C 语言）当中的常见 for 循环。看起来像这样：

```
for (initialization; test; increment) {
  body;
  body;
}
```

虽然对 Perl 而言，这种类型的循环事实上只是一种变相的 while 循环，如下所示【注 9】：

```
initialization;
while (test) {
  body;
  body;
  increment;
}
```

for 循环目前最常见的用途，就是控制重复的运算过程：

```
for ($i = 1; $i <= 10; $i++) {  # 从 1 数到 10
  print "I can count to $i!\n";
}
```

如果之前曾经见过这种做法，那么你不必看注释就知道第一行在说什么。在循环开始之前，控制变量 $i 被设置为 1。然后，它就像 while 循环一样，当 $i 的值小于或等于 10 时，循环会不断迭代执行。每次迭代之后与下一次迭代之前会进行递增运算，也就是将控制变量 $i 加 1。

因此，在循环第一次执行时，$i 是 1。因为它小于或等于 10，所以程序会输出信息。虽然递增操作符被写在循环顶端，但在逻辑上它却是位于循环底部，等输出信息之后才

注 9：　其实递增是在 continue 块发生的，本书不会涉及。请查看 *perlsyn* 在线手册了解真相。

会执行。于是，$i 递增到 2，依然小于或等于 10，因此程序会再次输出信息。接着 $i 递增到 3，还是小于或等于 10，依此类推。

最后，程序会输出数到 9 的信息。然后 $i 递增到 10，依然小于或等于 10，所以程序会执行最后一次循环，并且输出信息表明数到了 10。$i 在最后一次递增时变成了 11，这次不再小于或等于 10。所以程序会跳出循环继续执行接下来的代码。

因为这三个部分被一起放在循环的开头，所以老练的程序员看到第一行就明白："这是一个将 $i 从 1 数到 10 的循环。"

注意，当循环结束之后，它的控制变量取值会在范围之外。在这个例子里面，控制变量的值已经涨到了 11【注 10】。这种循环非常灵活，可以用来进行各式各样的计数。比如从 10 数到 1：

```
for ($i = 10; $i >= 1; $i--) {
  print "I can count down to $i\n";
}
```

下面这个循环从 -150 开始累加 3，一直到 1000：【注 11】

```
for ($i = -150; $i <= 1000; $i += 3) {
  print "$i\n";
}
```

事实上这三个循环控制部分（初始化、测试和递增）都可以为空，但是即使不需要它们也还得保留分号。在下面这个极不寻常的例子里面，测试条件是一个替换，而递增则是空：

```
for ($_ = "bedrock"; s/(.)//; ) {  # 当 s/// 这个替换成功时，循环继续
  print "One character is: $1\n";
}
```

在隐式 while 循环中的测试表达式是一个替换，成功替换时会返回真。在这个例子里，当循环第一次执行时，替换会拿走 bedrock 中的 b 字母。每次循环会拿走一个字母，直到字符串为空。这时替换会失败，导致循环结束。

在测试表达式为空的时候，两个连续的分号会被强行解释为真，从而导致死循环。但是在你学到本章的后面，能自如地中断这种循环之前，请不要尝试它：

注 10：请务必看看《This is Spinal Tap》这部风靡一时的电影，了解什么是 up to eleven。

注 11：注意其实是不可能真的数到 1000 的，最后一个数字应该是 999，因为 $i 所有的值都应该是 3 的整数倍。

```
for (;;) {
  print "It's an infinite loop!\n";
}
```

如果真的需要，更具 Perl 风格的死循环是【注 12】是 while 版本的：

```
while (1) {
  print "It's another infinite loop!\n";
}
```

虽然 C 程序员比较熟悉第一种方式，但即使是初学 Perl 的人也知道 1 总是真，从而导致死循环，因此第二种写法通常更好一些。Perl 很聪明地意识到这种常量表达式是可以优化的，因此不会导致性能问题。

foreach 和 for 之间的秘密关系

事实上，在Perl的解析器里，foreach 和 for 这两个关键字是等价的。也就是说，当 Perl 看到其中之一时，也就好像看到了另一个。Perl 可以从圆括号里的内容判断出你的意图。如果里面有两个分号，它就是刚才见到的 for 循环。若是没有分号就是一个纯正的 foreach 循环：

```
for (1..10) {  # 和一个从 1 到 10 的 foreach 循环一样
  print "I can count to $_!\n";
}
```

这是一个纯正的 foreach 循环，但却用了 for 关键字。除此以外，本书其他的例子都会写成 foreach 的形式。不过在真实的世界中，你认为大多数 Perl 牛仔会多打那四个字母吗【注 13】? 除了初学者的程序代码，它通常都会被写成 for，所以你必须像 Perl 一样检查分号来判断它是哪一种循环。

在 Perl 的世界里，纯正的 foreach 循环几乎总是更好的选择。在上面的 foreach 循环例子（表面上写成了 for 循环）里面，可以一眼看出它是从 1 到 10 的循环。可是，下面这个 for 循环也在试图做一样的事情，你看得出它有什么问题么？请自己找答案，

注 12：如果不小心进入死循环，试试看按 Control-C 能否终止。当然在按下之后，程序可能还是会在屏幕上滚过一些信息才停止，这是系统 I/O 和其他的因素而导致的。现在我们已经尽力指出危险所在了。

注 13：如果你觉得还是谨慎些比较好的话，那么你就是还缺少历练。在程序员（尤其是 Perl 程序员）看来，懒惰是传统美德。如果不信，可以参加下次 Perl Monger 聚会调查一下。

不要偷看脚注【注 14】：

```
for ($i = 1; $i < 10; $i++) {   # 糟糕！这里某个地方有错误
  print "I can count to $_!\n";
}
```

循环控制

目前为止你大概已经感觉到，Perl 是一种所谓的结构化编程语言。特别是 Perl 程序的任何块都只有一个入口，也就是块的顶端。不过相比前面介绍过的结构，有时候需要更多样化的控制方式。例如有时候可能需要一个起码执行一次的 while 循环，或者需要提早退出某块代码。Perl 有三个循环控制操作符，你可以在循环里使用它们，让循环非常灵活。

last 操作符

操作符 last 能立即中止循环。就像在 C 一类语言中的 break 操作符一样。它是循环的紧急出口。当你看到 last，循环就会结束。例如：

```
# 打印出所有提到 fred 的行，直到碰到 __END__ 记号为止
while (<STDIN>) {
  if (/__END__/) {
    # 碰到这个记号说明再也没有其他的输入了
    last;
  } elsif (/fred/) {
    print;
  }
}
## last 指令会跳到这里 #
```

只要输入行中有 __END__ 记号，这个循环就会结束。当然，结尾的那行注释只是提醒而已，完全可以省略。我们只是将它放在那里，好让流程更加清晰。

在 Perl 中有 5 种循环块。也就是 for、foreach、while、until 以及裸块【注 15】。

注 14：这里有两个半错误。第一，条件判断部分使用小于号，因此实际的循环只执行 9 次而非 10 次。这种循环很容易陷入所谓的栅栏问题，比如有位农夫想建一条 30 米的栅栏，每隔 3 米立一个栅栏，那么农夫需要多根栅栏才够？答案可不是 10 个哦。第二，控制变量是 $i，但循环体中使用的却是 $_。还有半个错误是，对于这种写法，需要读、写、维护、调试的代码更多，这也就是为什么我们说纯正的 foreach 是一个更好的选择。

注 15：没错，你可以用 last 来跳出裸块。但这并不意味着可以裸奔回家。

而 if 块或者子程序带的花括号【注 16】不是循环的块。如同前面的例子，last 操作符对整个循环块起作用。

last 操作符只会对运行中最内层的循环块发挥作用。要跳出外层块，请继续看下去，我们很快就会提到。

next 操作符

有时候并不需要立刻退出循环，但是需要立刻结束当前这次迭代。这就是 next 操作符的用处，它会跳到内层循环块的底端【注 17】。在 next 之后，程序将会继续执行循环的下次迭代，这和 C 一类语言中 continue 操作符的功能相似：

```
# 分析输入文件中的单词
while (<>) {
  foreach (split) {  # 将 $_ 分解成单词，然后每次将一个单词赋值给 $_
    $total++;
    next if /\W/;    # 如果碰到不是单词的字符，跳过之后的表达式
    $valid++;
    $count{$_}++;    # 分别统计每个单词出现的次数
    ## 上面的 next 语句如果运行，会跳到这里 ##
  }
}

print "total things = $total, valid words = $valid\n";
foreach $word (sort keys %count) {
  print "$word was seen $count{$word} times.\n";
}
```

这个例子比前面的要复杂些，所以我们逐步进行解说。while 循环逐行读取来自钻石操作符的输入并放进 $_，这我们先前已经看过了。循环每次执行时，$_ 就得到输入数据的下一行。

在循环中，foreach 能遍历 split 返回的列表。还记得 split 不带参数的默认行为吗【注 18】? 它会用空白来切分 $_，也就是说把 $_ 分解成由单词组成的列表。既然

注 16：这可能是个坏主意，但确实可以在子程序里用循环控制操作符来影响子程序外面的循环。也就是说，如果在循环块内调用一个子程序，子程序内执行 last 操作，同时子程序没有循环块，那么程序的流程会跳到主程序循环块的后面。在将来的 Perl 中，这种在子程序内的循环控制能力会被去掉，没有人会怀念它的。

注 17：这里我们又撒了一个小谎。事实上 next 会跳到循环中常常省略的 continue 块的开头处。参见 *perlsyn* 了解详细的信息。

注 18：不记得了，也别太过担心。不必在那些 *perldoc* 能查到的东西上花费太多精力。

foreach 循环没有提到其他控制变量，控制变量就应该是 $_。因此我们会在 $_ 中依次看到所有的单词。

可是，我们不是才说过 $_ 是用来存储每一行的输入吗？在外层循环就是这样。但是在 foreach 循环里，它却能循环存储每一个单词。Perl 能正确处理 $_ 的多版本重用，这种事并不奇怪。

对 foreach 循环来说，每当我们在 $_ 中看到一个单词时，$total 就会递增，所以它会是全部单词的总数。下一行是这个例子的关键，它会检查单词里是否包含任何非单词字符（字母、数字和下划线以外的字符）。因此如果其中出现了像 Tom's 或者 full-sized 这样的连接符，或者后面紧接着逗号、引号或任何其他奇怪的字符，那它就会匹配这个模式，导致循环直接跳到下一个单词。

不过如果找到了一个普通的单词，比如 fred。在此情况下，我们会将 $valid 加 1，连带 $count{$_} 也累加，记录此单词出现的次数。所以，在这两个循环执行完毕后，我们就对用户指定的所有文件中的每一行进行单词统计。

我们不会再解释最后几行的意思。到了这里，我们希望你已经有能力对付这样的程序代码。

跟 last 一样，next 也可以用在 5 种循环块中：for、foreach、while、until 或是裸块。同样的，如果有多层的嵌套循环块，next 只会对最内层起作用。这一节的最后，我们将会看到如何突破这种限制。

redo 操作符

循环控制族的第三个成员是 redo。它能将控制返回到本次循环的顶端，不经过任何条件测试，也不会进入下一次循环迭代。而那些用过 C 一类语言的人却会对这个操作符感觉陌生。因为那些语言里没有这个概念。例子如下：

```
# 打字测试
my @words = qw{ Fred barney pebbles dino wilma betty };
my $errors = 0;

foreach (@words) {
  ## redo 指令会跳到这里 ##
  print "Type the word '$_': ";
  chomp(my $try = <STDIN>);
  if ($try ne $_) {
    print "Sorry - That's not right.\n\n";
    $errors++;
    redo;  # 跳回循环的顶端
```

```
    }
}
print "You've completed the test, with $errors errors.\n";
```

和另外两个操作符一样，redo 在这5种循环块里都可以使用，并且在循环块嵌套的情况下只对最内层的循环起作用。

next 和 redo 两者之间最大的区别在于 next 会正常继续下一次迭代，而 redo 则会重新执行这次的迭代。下面的程序可以让你体验这三种操作符工作方式的区别【注19】：

```
foreach (1..10) {
  print "Iteration number $_.\n\n";
  print "Please choose: last, next, redo, or none of the above? ";
  chomp(my $choice = <STDIN>);
  print "\n";
  last if $choice =~ /last/i;
  next if $choice =~ /next/i;
  redo if $choice =~ /redo/i;
  print "That wasn't any of the choices... onward!\n\n";
}
print "That's all, folks!\n";
```

如果你不键入任何字，只是按下回车键，则循环会逐次增加计数。如果你在数字4显示的时候选择 last，那么循环就会因此而结束，你将看不到数字5。如果你在数字4显示的时候选择 next，就会直接进入数字5而不提示 onward 信息。如果你在数字4显示的时候选择 redo，那么会在4这个数字上重来。

带标签的块

有时候需要用标签从内层对外层的循环块进行控制。在 Perl 里，标签和其他标识一样由字母、数字和下划线组成，但是不能以数字开头。然而由于标签没有前置符号，就可能和内置函数或自定义子程序名混淆。所以将标签命名为 print 或 if 是很糟糕的选择。因此 Larry 建议用全大写字母命名标签，这样不仅能防止跟其他标识符相互冲突，也使得它们在程序中能突显出来。无论大写还是小写，标签总是罕见的，只会在很少的 Perl 程序中出现。

要对某个循环块加上标签，通常只要将标签及一个冒号放在循环前面就行了。之后在循环里，若有需要就可以在 last、next 或 redo 的后面填上这个标签。

注19：如果你已经从 O'Reilly 网站（在序介绍过）下载了这些文件的话，这个程序的名字叫做 *lnr-example*。

```
LINE: while (<>) {
  foreach (split) {
    last LINE if /__END__/;  # 跳出标签为 LINE 的循环
    ...
  }
}
```

为了增进可读性，通常的建议是把标签靠左写，哪怕当前的代码的层次缩进很深。注意标签应该用来命名整块代码，而不是用来标明程序中的某个具体位置【注 20】。在上面的例子程序里，特殊的 __END__ 记号代表了输入的结束。只要看到这个记号，程序就会忽略所有接下来的输入行，即使还有其他未读的文件。

通常应该以名词来为循环命名【注 21】。也就是说，因为外层循环是每次处理一行，所以可称之为 LINE。如果也要为内层循环取个名字，我们可能会叫它 WORD，因为它每次处理一个单词。如此一来，写出 next WORD 或者 redo LINE 之类的代码也很自然。

三目操作符 ?:

当 Larry 考虑 Perl 要提供哪些操作符时，他不想让老的 C 程序员有机会怀念那些 C 有而 Perl 没有的东西。所以他把 C 所有的操作符都搬过来了【注 22】。这个决定导致 Perl 拥有了 C 语言最让人困惑的操作符，也就是三目 ?: 操作符。虽然它可能令人困惑，不过有时也相当有用。

三目操作符就像一个 if-then-else 控制表达式。由于使用时需要三个操作数，所以称为三目操作符。它看起来像这样：

```
条件表达式 ? 真表达式 : 假表达式
```

首先，执行条件表达式，看看究竟是真还是假。如果为真，则执行真表达式；否则，就执行假表达式。每次使用时都会执行问号右边两个表达式中的一个，另一个则会被跳过。换句话说，若条件表达式为真，则第二个表达式会被求值并返回，忽略第三个表达式；倘若条件表达式为假，则忽略第二个表达式，而对第三个表达式求值并返回。

注 20：无论如何，标签不是为了 goto 而设定的。

注 21：起码，这么做要比随意命名更有意义。你就算把循环的标签设为 XYZZY 或是 PLUGH，Perl 也不会介意。可要是不熟悉 20 世纪 70 年代的 Colossal Cave 游戏，就没人会知道你这么写是什么意思。

注 22：严格说来，他其实舍弃了在 Perl 中无用的操作符，例如将数字转换成内存地址的操作符。当然他还加上了几个让 C 程序员嫉妒的操作符，比如字符串连接。

在下面的例子里，子程序 &is_weekend 的执行结果决定了哪个字符串表达式会被赋值给变量：

```
my $location = &is_weekend($day) ? "home" : "work";
```

下面的例子中，我们会计算并输出一个平均值，或是在无法计算时用一行连字符来代替。

```
my $average = $n ? ($total/$n) : "-----";
print "Average: $average\n";
```

任何使用 ?: 操作符的表达式都可以改写成 if 结构，但是常常会更拖沓冗长：

```
my $average;
if ($n) {
  $average = $total / $n;
} else {
  $average = "-----";
}
print "Average: $average\n";
```

这是你可能喜欢的技巧，用来写出干净利落的多路分支：

```
my $size =
  ($width < 10) ? "small"  :
  ($width < 20) ? "medium" :
  ($width < 50) ? "large"  :
                  "extra-large"; # default
```

实际上，这是由三层嵌套的 ?: 操作符组成的。而且一旦明白其诀窍所在，就会觉得十分简洁。

当然，这个操作符并不是非用不可，初学者可能看了头疼。不过，迟早你会在其他人的程序里看到它，而我们也希望有一天你会在自己的程序里使用它。

逻辑操作符

如你所愿，Perl 拥有全套的逻辑操作符，可以用来对付布尔（真假）值。例如常用来进行联合逻辑测试的“逻辑与”操作符 && 和“逻辑或”操作符 ||：

```
if ($dessert{'cake'} && $dessert{'ice cream'}) {
  # 两个条件都为真
  print "Hooray! Cake and ice cream!\n";
} elsif ($dessert{'cake'} || $dessert{'ice cream'}) {
  # 至少一个条件为真
  print "That's still good...\n";
} else {
  # 两个条件都为假 —— 什么也不干
}
```

Perl 在这里可能会走捷径。如果“逻辑与”的左边表达式为假，整个表达式就不可能为真，因为必须两边都为真才会得到真。因此这时不必再检查右边的表达式，从而避免对其求值。针对下面的例子，请考虑 $hour 是 3 的时候会怎样：

```
if ( (9 <= $hour) && ($hour < 17) ) {
  print "Aren't you supposed to be at work...?\n";
}
```

相似的地方还有，“逻辑或”操作符的左边表达式为真，那么右边也会免于求值。请考虑下面的例子中 $name 是 fred 会如何：

```
if ( ($name eq 'fred') || ($name eq 'barney') ) {
  print "You're my kind of guy!\n";
}
```

因为这个特性，这种操作符被称为短路逻辑操作符。它们只要有可能就走捷径获得结果。事实上，依赖这种短路行为的代码很常见，比如求平均值的程序：

```
if ( ($n != 0) && ($total/$n < 5) ) {
  print "The average is below five.\n";
}
```

这个例子里，右边的表达式只有在左边为真的时候才运算，因此程序不会因为意外的“除以零”而崩溃。

短路操作符的值

与 C 一类的语言不同的地方在于，Perl 的短路操作符求得的值不只是简单的布尔值，而是最后运算的那部分表达式的值。这恰好和布尔值兼容，因为最后部分的值若是真，整个就是真，反之亦然。

但是这个返回值会很有用。“逻辑或”操作提供的一个便利就是附加默认值：

```
my $last_name = $last_name{$someone} || '(No last name)';
```

如果 $someone 在哈希中并不存在，左边的计算结果就是 undef，也就是假。所以“逻辑或”操作符必须对右边的表达式求值，使其成为默认值。这个习惯用法中，默认值不只是为 undef 设置的，也是为所有逻辑假的值设置的。要进一步限制的话得使用三目操作符：

```
my $last_name = defined $last_name{$someone} ?
    $last_name{$someone} : '(No last name)';
```

这太麻烦了，而且必须反复写 $last_name{$someone}。好在 Perl 5.10 提供了更简洁的写法，请看下一节。

“定义否”操作符

在前一节我们谈到了 || 操作符能提供默认值。但是没有考虑到特殊情况：就是已定义的假值，也可能被意外的替换为默认值。因此后来又有了更丑陋的三目操作符版本。

Perl 5.10 能避免这样的 bug，是因为引入了“定义否”操作符 //，在发现左边已定义的值时进行短路，不论左边的值是真是假。现在哪怕有人的名字是 0，代码也能正常工作：

```
use 5.010;

my $last_name = $last_name{$someone} // '(No last name)';
```

有时候需要给一个未定义变量赋值，若已定义则保留原值。比如程序常常会参考 VERBOSE 环境变量来决定是否打印信息。现在可以检查 %ENV 哈希中 VERBOSE 键的值是否定义，若未定义则对它赋值：

```
use 5.010;

my $Verbose = $ENV{VERBOSE} // 1;
print "I can talk to you!\n" if $Verbose;
```

可以多弄几个值来测试一下 //，看看它会在那种情况下才会返回 default：

```
use 5.010;

foreach $try ( 0, undef, '0', 1, 25 ) {
    print "Trying [$try] ---> ";
    my $value = $try // 'default';
    say "\tgot [$value]";
    }
```

输出显示只有在 $try 是 undef 的时候才会收到 default 字符串。

```
Trying [0] --->     got [0]
Trying [] --->      got [default]
Trying [0] --->     got [0]
Trying [1] --->     got [1]
Trying [25] --->    got [25]
```

有时候需要对一个未定义的变量赋值。例如开启了警告状态，并打印一个未定义的值，就会收到烦人的错误：

```
use warnings;
```

```
my $name;  # 没有值，未定义
printf "%s", $name; # 在 printf 使用了未初始化的变量
```

当然这种错误常常无害，可以忽略。但是如果确实要打印未定义的值，就可以用一个空串来代替：

```
use 5.010;
use warnings;

my $name;  # 没有值，未定义
printf "%s", $name // '';
```

使用部分求值操作符的控制结构

之前看到的4个操作符 &&、||、// 和 ?: 都有一个共性：根据左边的求值决定是否计算右边的表达式。有些情况会执行的表达式，在另外的情况下并不执行。因此被统称为部分求值操作符，部分求值操作符是天然的控制结构，因为不会执行所有的表达式【注23】。并非 Larry 热衷于在 Perl 里加入更多的控制结构，而是因为在决定将部分求值操作符加进 Perl 的那一刻，它们就成了天然的控制结构。毕竟任何能激发或跳过某段程序代码的东西都会成为控制结构。

只有带副作用（例如进行变量修改或是产生输出）的受控表达式才会幸运地引起注意。例如运行下面的代码：

```
($m < $n) && ($m = $n);
```

你很快会注意到，“逻辑与”的运算结果并没有被赋值到任何地方【注24】。为什么呢？

如果 $m 真的小于 $n，左边就为真，所以会执行右边的赋值。不过若是 $m 不小于 $n，则左边为假并导致右边被跳过。所以，上面的程序代码基本上和下面这一行做的是同样的事情，但后者显然比较容易理解：

```
if ($m < $n) { $m = $n }
```

你或许会在维护某个程序时看到如下这一行：

```
($m > 10) || print "why is it not greater?\n";
```

注23：看到这里，没准已经有人纳闷，为什么本章要介绍这些逻辑操作符呢？

注24：然而别忘了如果它是子程序的最后一项表达式的话，就会成为返回值。

如果 $m 真的大于10，那么左边为真，因此“逻辑或”运算就算完成了。若非如此，则左边为假，于是会接着输出信息。跟上面一样，这其实可以（而且也建议）使用传统的写法，比如用 if 或 unless 来写。

而那些特别擅长逻辑的人甚至学会了用读英语的方式来读这几行程序代码。例如：检查 $m 是否小于 $n，若是如此，则进行赋值；检查 $m 是否大于 10，若非如此，则输出信息。

会以这种方式来写控制结构的人，通常是 C 程序员或是老的 Perl 程序员。他们之所以这样写是为了提高所谓的性能。也有人认为这些技巧能让他们的程序比较酷，还有一些人只是模仿别人的编程风格而已。

三目操作符同样可以成为控制结构。在下面的例子里，我们想将 $x 赋值给两个变量中较小的那个：

```
($m < $n) ? ($m = $x) : ($n = $x);
```

如果 $m 比较小，它会得到 $x；否则的话，就归 $n 所有了。

“逻辑与”、“逻辑或”操作符还有另一种写法。你可以将它写成单词：and 和 or【注25】。这种单词操作符和标点符号形式的效果相同，但是前者在运算优先级上要低得多。既然单词操作符不会紧紧地粘住附近的表达式，它们需要的括号可能就会少一些：

```
$m < $n and $m = $n;  # 写成相应的 if 语句版本会更好
```

当然，可能你还是要用到更多的圆括号，因为优先级是一个怪兽。如果对优先级不是非常有把握，请回来使用圆括号。然而单词操作符的优先级很低，所以你通常可以想象它们会把表达式拆成两片，先做左边所有的事情，如果需要再做右边所有的事情。

尽管以逻辑操作符作为控制结构可能令人困惑，但有时候却是众人都认可的程序写法。如下就是打开文件的习惯写法：

```
open CHAPTER, $filename
  or die "Can't open '$filename': $!";
```

通过使用低优先级的短路 or 操作符，我们表达了 open this file or die 的意思。如果文件打开成功，就会返回真值，此时 or 就不必执行了；但如果文件打开失败，or 就还得去执行右边的部分，也就是丢出信息并终止程序。

注25：其实还有低优先级的 not 操作符，与此对应的逻辑反操作符是 !，另外也有很罕见的 xor 操作符。

所以，用这些操作符作为控制结构是 Perl 习惯用语的一部分，也就是 Perl 约定俗成的表达方法。适当使用的话，程序会更具威力；否则，你的程序将会难以维护。请别滥用它们【注 26】。

习题

参考附录 A 核对以下练习的答案：

1. [25] 编个程序，让用户不断猜测范围从 1 到 100 的秘密数字，直到猜中为止。程序应该以秘密公式 `int(1 + rand 100)` 【注 27】来随机产生秘密数字。当用户猜错时，程序应该响应 Too hight 或 Too low。如果用户键入 `quit` 或 `exit` 等字样，或是键入一个空白行，程序就应该中止。当然，如果用户猜到了，程序也该中止！

2. [10] 修改前一个练习的程序，打印额外的调试信息，例如程序选择的秘密数字。确保修改的部分可以用开关控制，而且调试开关即使关上也不会产生警告。如果使用 Perl 5.10 版本可以用 `//` 操作符，否则请使用三目操作符。

3. [10] 修改第六章的练习3的程序，打印出那些未定义的环境变量，显示为 `(undefined value)`。可以修改环境变量的值，来测试程序正确地打印了那些假值。如果使用 Perl 5.10 版本，可以用 `//` 操作符，否则请使用三目操作符。

注 26： 以上的怪异代码，一个月写一次以上就可以算是滥用，当然 `die` 除外。

注 27： 若是好奇，可请参考 *perlfunc* 在线手册关于 `int` 和 `rand` 函数的介绍。

第十一章

Perl 模块

关于 Perl 的故事远不止我们在这本书里提到的，有许多人还用 Perl 做各种有趣的事。如果碰上什么棘手的问题难以解决，多半可以在 Perl 综合典藏网（CPAN）上找到别人现成的解决方案。CPAN 在世界各地都有服务器与镜像站点，并包含了上千个可用的开源 Perl 模块。

在这里我们不打算教你如何编写自己的模块，要学的话可以看羊驼书。本章我们会教你如何使用现有的模块。

寻找模块

Perl 模块有两种来源：一种是随 Perl 发行版本一同打包的，所以安装了 Perl 就可以用这些模块；另一种则需要从 CPAN 下载，需要自己安装。除非特别说明，一般我们讨论的都是随 Perl 一同发布的模块。

要找寻那些没有随 Perl 发布的模块，可以到 CPAN Search 网站（*http://search.cpan.org*），或是到 Kobes's Search 网站（*http://kobesearch.cpan.org/*）【注 1】看看。可以按类别查找，也可以直接搜索。

这两个网站都很不错，你可以在下载整个模块软件包之前，先看看相关的文档和使用范例。同时也可以先看看一起打包发布的还有哪些文件，而不必等到下载、安装之后再了解。

注 1：这个域名里该有两个“s”，但是到现在为止还是没人去申请一个新的正确的域名。谁会在意呢？

在寻找模块之前，请先检查一下系统上是否已经安装过。可以用 *perldoc* 试着打开该模块的文档。比方说，Perl 自带有 CGI.pm 模块（我们稍后会讨论此模块），所以应该可以直接阅读它的文档：

```
$ perldoc CGI
```

改用没装过的模块名称再试试看，你会看到一条错误信息。

```
$ perldoc Llamas
$ No documentation found for "Llamas".
```

你的系统上可能还安装有其他格式的文档，比如 HTML 格式。当然前提是模块带有基础文档【注 2】。

安装模块

如果想要安装系统上没有的模块，只需下载打包发布的文件，解压缩后在 shell 中运行一连串命令就可以了。详细的信息可以查阅 README 或是 INSTALL 这两个文件。如果此模块使用 MakeMaker【注 3】，那多半可以用如下方式安装：

```
$ perl Makefile.PL
$ make install
```

如果你没有权限安装模块到系统全局目录（安装后其他用户也可使用的目录），则可以在 Makefile.PL 后面加上 PREFIX 参数，来指定以你的用户身份可写的安装目录：

```
$ perl Makefile.PL PREFIX=/Users/fred/lib
```

有些 Perl 模块的开发者会要求使用另一个辅助模块 Module::Build 来编译并安装他们的作品。此时安装的方式大致如下：

```
$ perl Build.PL
$ ./Build install
```

注 2：《Intermediate Perl》（O'Reilly 出版）这本书中涵盖了与 Perl 文档相关的章节。但现在请相信我们，绝大多数模块的文档都已经自包含在该模块的代码文件里了。

注 3：该模块的全称是 ExtUtils::MakeMaker，它也是随 Perl 一起发布的。其中的一系列工具能帮你生成安装文件，并保证其中的安装指令适应 Perl 和系统两方面的要求。

有些模块的工作还依赖于其他模块，所以必须先安装好其他模块，才能继续安装。与其自己手动一个个安装依赖模块，不如用 Perl 自带的 CPAN.pm 来完成这项繁琐的工作【注 4】。你可以在命令行启动 CPAN.pm，在其中运行相关的模块维护命令：

```
$ perl -MCPAN -e shell
```

这样写起来还是稍嫌复杂，所以一段时间之前本书作者之一（brian d foy）写了个小小的脚本程序，叫作“cpan”。它内置于 Perl 发行包中，一般会与 *perl* 命令以及附带的工具程序一并安装（在某个目录中）。现在，只要把想安装的模块名称列表当成参数传给它就行了：

```
$ cpan Module::CoreList LWP CGI::Prototype
```

“我没有命令行！”或许你会这样抱怨。如果你使用的是 ActiveState 公司发行的 Perl 版本（有针对 Windows、Linux 和 Solaris 的版本），还可以使用 Perl 包管理器（Perl Package Manager，简称 PPM）【注 5】来安装模块。你甚至还可以从 ActiveState 出品的 CD 或 DVD【注 6】获取他们的 Perl 发行版本以及相应的模块仓库。到目前为止，不管你的操作系统是什么，都该有办法安装 Perl 模块了。

使用简单模块

假设在程序里有个包含路径的长文件名，像是 /usr/local/bin/perl，而你想要取文件的 *basename*。这不难，最后一个斜线之后的部分就是 basename（此例中为 perl）：

```
my $name = "/usr/local/bin/perl";
(my $basename = $name) =~ s#.*/##;  # 唉呀!
```

如你所见，Perl 会先在圆括号中进行赋值，然后再进行替换操作。这里的替换操作将结尾为斜线的字符串（也就是文件路径的部分）替换成空字符串，这样就只剩下 basename。

注 4：“.pm”扩展文件名代表“Perl Module”，为了与其他概念相区分，通常谈到一些流行的模块时都会带上“.pm”。在这里，CPAN（Perl 综合典藏网）和模块 CPAN 不是同一个东西，所以我们说“CPAN.pm。”

注 5：*http://aspn.activestate.com/ASPN/docs/ActivePerl/faq/ActivePerl-faq2.html*。

注 6：你可以建立一个本地的 CPAN 仓库用以制作你自己的 CD 或者 DVD 光盘。尽管现在 CPAN 有将近 4GB 大，但是你能用 minicpan（创意来自本书的一个作者）下载所有模块的最新版本，而这只要 800MB 就够了。请参看 CPAN::Mini 模块。

如果这么写，看起来应该可以工作。好吧，这只是看起来，实际上这么写有三个问题。

首先，Unix 上的文件或目录名称可能会包含换行符（这虽然不是常常发生，但确实可能发生）。由于正则表达式的点号（.）无法匹配换行符，像 "/home/fred/flintstone\n/brontosaurus" 这样的文件全名，便无法正常运作——此段代码会认为文件的 basename 是 "flintstone\n/brontosaurus"。你可以用模式的 /s 选项加以修正（如果你考虑到这种微妙而又罕见的情况的话），写作：s#.*/##s。第二个问题是，这段代码仅仅考虑了 Unix 下的情况。它假设目录分隔符总是 Unix 风格的斜线，而没有考虑其他系统（使用反斜线或冒号）的情况。

第三个（也是最大的）问题是，我们正在试图解决别人已经解决的问题。Perl 自带了相当数量的标准模块，它们作为 Perl 的扩展，增加了许多新的特性和功能。若这还不够，CPAN 上还有更多好用的模块，每周都有许多新模块加入，每天都有许多老模块更新。如果需要某些模块提供的功能，你（或最好是系统管理员）可以去那里下载安装，然后直接用现成的。

我们会在本章后续的章节中，选取一些随 Perl 一同捆绑发布的简单模块，为你演示它们的部分特性和用法（事实上，这些模块还可以做更多的事，目前只是管中窥豹，借机展示模块使用的常例）。

本书无法介绍关于模块使用的全部知识，因为你得先了解引用和对象之类的高级主题才有办法使用某些模块【注 7】。这些话题（以及该如何创建模块等等）都在羊驼书里有详细精妙的阐释。而本节则可以帮你从简单模块的使用开始起步。想要进一步了解某些有趣且实用的模块，可以参考附录 B。

File::Basename 模块

在前面的例子中，我们用了不可移植的方式取得 basename。这种解决问题的方式看起来简洁，却可能因为微妙的差异而失败（比如这里就假设文件名或目录名中不会出现换行符）。并且我们还“重新发明了轮子”，解决别人早就解决过（也调试过）好几次的问题。

要从文件全名里取出 basename，这里有个更好的做法，就是使用 Perl 自带的 File::Basename 模块。只要执行 *perldoc File::Basename* 这个命令或是通过特定平台的文档系统，你就能看到它的说明。这是使用新模块的第一步（没准也会是第三步与第五步）。

注 7：如同我们在后面几页中将要看到的一样，你可以调用这些带有对象和引用的模块，但并不需要先明白这些高级话题。

很快，你准备好要使用它，在程序开头的地方，用 use 指令来声明【注 8】：

```
use File::Basename
```

在程序的编译阶段，Perl 看到这行代码后，会尝试寻找此模块的源代码并加载进来。接着就好像 Perl 突然多出了一些新函数，在程序接下来的部分都可以使用这些函数了【注 9】。此前的例子中我们需要的正是 basename 函数：

```
my $name = "/usr/local/bin/perl";
my $basename = basename $name;  # 返回 'perl'
```

嗯，这样虽然在 Unix 上行得通，但如果我们的程序在 MacPerl、Windows 或是 VMS 等系统上运行呢？不必担心——这个模块会判断你用的是哪种系统，并且使用该系统默认的文件命名规则（当然，这时 $name 里存放的，就是属于该系统的文件名称字符串了）。

此模块还提供了一些其他的相关函数。其中的 dirname 函数可以从文件全名里取得目录名称。这个模块也能让你将文件名和扩展名分开，或是改变默认的文件名规则【注 10】。

仅选用模块中的部分函数

假设你想在以前写好的程序里加上 File::Basename 模块，却发现程序里已经有个叫作 &dirname 的子程序——这就是说，程序里的子程序和模块里的某个函数有着相同的名称【注 11】。现在有个麻烦，通过使用模块而引入的 dirname，恰好和自己写的子程序同名。该如何化解这样的冲突呢？

只需在 File::Basename 的 use 声明中，加上*导入列表*（import list）来指明要它提供哪些函数，它就不会给你别的函数了。在此，我们只需要 basename 函数：

```
use File::Basename qw/ basename /;
```

注 8：一般我们会在文件的头部声明这些模块，这样维护程序员比较容易弄清楚到底用了哪些模块。这样的话，如果你要在新机器上安装你的程序，问题会简单很多。

注 9：你猜到了：事情不是这么简单，你得知道包和全称。当你的主程序超过几百行（不算你使用的模块的代码行），成为较大的 Perl 程序后，你就得学习使用这些高级特性了，请阅读 *perlmod* 在线手册。

注 10：当你试图处理从 Windows 机器获得的 Unix 上的文件名时（比如通过 FTP 连接发送命令的时候），你可能需要修改文件名规则。

注 11：好吧，你可能没有和 &dirname 同名用途却不同的子程序，只是举个例子。因为一些模块提供成百个新函数（真的！），所以还是可能发生这种冲突的。

如果像下面这么写的话，就表示我们完全不要引入任何函数：

```
use File::Basename qw/ /;
```

通常也会写成：

```
use File::Basename ();
```

为什么要这么做呢？嗯，这条指令告诉 Perl，加载 File::Basename 模块，这和前面一样，但不要导入任何函数名称。导入函数的目的，是要使用简短的函数名称，像 basename 和 dirname。然而，哪怕不导入这些名称，我们还是可以通过全名的方式来调用相应的函数：

```
use File::Basename qw/ /;                          # 不导入函数名称

my $betty = &dirname($wilma);                      # 使用我们自己的子程序 &dirname
                                                   # (略去该子程序的具体内容)

my $name = "/usr/local/bin/perl";
my $dirname = File::Basename::dirname $name;       # 使用模块中的 dirname 函数
```

如你所见，模块里的 dirname 函数的全名是 File::Basename::dirname。加载模块后，无论是否导入 dirname 这种简短名称，我们都可以随时使用函数全名的方式来调用。

大多情况下，使用模块默认的导入列表就行了。不过，你随时可以用自己定义的列表。一来可以略去你不需要的默认导出函数；二来还可以按需要导入那些不会自动导出的函数，因为大多数模块的默认导入列表里都会省略某些（不常使用的）函数。

没错，有些模块默认的导入列表就是特别长。所有模块的说明文档，都应该列出可供导入的符号（如果有的话），但你随时都可以用自己的列表覆盖掉默认的导入列表，就像上面 File::Basename 例子里的做法一样。而空列表意味着不导入任何符号。

File::Spec 模块

现在你可以取文件的 basename 了，这很有用。此外你还常会需要将 basename 和目录名结合起来，以构造文件的全名。比如对 /home/rootbeer/ice-2.1.txt 这样的文件全名，想要为其 basename 加上前缀：

```
use File::Basename;

print "Please enter a filename: ";
chomp(my $old_name = <STDIN>);
```

```
my $dirname = dirname $old_name;
my $basename = basename $old_name;

$basename =~ s/^/not/;  # 给 basename 加上前缀
my $new_name = "$dirname/$basename";

rename($old_name, $new_name)
  or warn "Can't rename '$old_name' to '$new_name': $!";
```

看出问题在哪了吗？和前面一样，我们假设文件名遵循 Unix 的惯例，即目录名接着正斜线，再接着文件的 basename。幸好，Perl 也提供了能够解决这个问题的模块。

你可以利用 `File::Spec` 模块来操作文件标识符（file specification），也就是文件名、目录名以及文件系统里的其他名称。它和 `File::Basename` 一样会在运行时判断系统的类型，并且总是采用正确的规则。但是 `File::Spec` 是面向对象的（Object Oriented，简称 OO）模块，这点和 `File::Basename` 不同。

要是你从未因 OO 的热潮而激动，请不必烦心。如果你了解对象是什么，那非常好，你可以使用这个 OO 模块；要是你不知道何谓对象，那也没关系，只要照样键入我们展示给你的代码，它就会正常工作，时间长了也就渐渐明白了。

在这个例子里，`File::Spec` 的说明文档告诉我们应该使用 `catfile` 这个方法（method）。“方法”是什么意思呢？就目前我们所关心的来说，它只不过是另一种函数而已。不同的是，你必须通过 `File::Spec` 的全名方式来调用方法，比如：

```
use File::Spec;

.
.  # 取得 $dirname 和 $basename 的值，方法同上
.

my $new_name = File::Spec->catfile($dirname, $basename);

rename($old_name, $new_name)
  or warn "Can't rename '$old_name' to '$new_name': $!";
```

如你所见，调用方法时的全名，应该由模块的名称（此处称为类或 class）、一个瘦箭头（->）以及方法的简短名称构成。请注意，这里用的是瘦箭头，而不是 `File::Basename` 里的双冒号，看到瘦箭头，就说明是面向对象的写法，后面是要调用的方法名。

不过，既然我们通过全名来调用方法，那么模块会导入哪些符号呢？答案是什么都没有。对 OO 模块来说，这是正常的做法。这样一来，你就不必担心自己的子程序会跟 `File::Spec` 里众多方法的名称重复了。

你一定要使用这些模块吗？其实你总是可以权衡的。比方说，假设你确定自己的程序不会在 Unix 之外的系统上运行，你又完全了解 Unix 文件名的规则【注 12】，那么你也可以将这些限制写定在程序中。但用上这些模块能让你花更少的时间，轻松构造更健壮的程序，而且更容易移植。

CGI.pm

若你需要创建 CGI 程序（本书不作此方面介绍），请用 CGI.pm 模块【注 13】。就算你明白甚至精通 CGI 交互过程中的所有细节，也不必在自己的脚本中，亲自实现那些着实让许多人困扰的接口操作和输入信息的解析。CGI.pm 的作者 Lincoln Stein 已经花了相当多的时间，确保此模块在大部分服务器和操作系统上都能正常运作。直接使用此模块，然后你就可以腾出精力，专注于脚本中真正有趣的部分。

CGI 模块有两种风格：古朴的函数接口和面向对象接口。我们将使用前一种。在此之前，你可以照着 CGI.pm 文档中的例子依样画葫芦。下面的简单 CGI 脚本会解析 CGI 输入，并以纯文本的方式来显示输入字段的名称和值。我们在“导入列表”中使用了 :all，这是一种标签写法，用来指定一组要导出的函数，而非一个，这和之前看到的例子有所不同【注 14】：

```
#!/usr/bin/perl

use CGI qw(:all);

print header("text/plain");

foreach my $param ( param() )
    {
    print "$param: " . param($param) . "\n";
    }
```

我们还可以输出 HTML 格式的结果，看起来会更花俏些。CGI.pm 有许许多多的实用函数可以派上用场。它可以负责输出 CGI 头，然后再用 start_html() 来生成 HTML

注 12：如果你不知道文件名和目录名可以包含换行符的话（我们之前提到过的），那就说明你不完全了解这些规则，对吧？

注 13：和 CPAN.pm 模块类似，谈到它的时候我们会把 CGI.pm 中的“.pm”读出来，以区别于 CGI 这个协议。

注 14：该模块还提供了其他可供导出的函数分组标签。比如说，如果只想处理 CGI 就可以只导入 :cgi；若要用输出 HTML 的函数，只导入 :html4 就可以了。进一步信息请参考 CGI.pm 的文档。

文档开头的部分，另外还有为数众多的，和 HTML 标签同名的函数可供使用，比如用 h1() 输出 H1 标记内的标题：

```
#!/usr/bin/perl

use CGI qw(:all);

print header(),
    start_html("This is the page title"),
    h1( "Input parameters" );

my $list_items;
foreach my $param ( param() )
    {
    $list_items .= li( "$param: " . param($param) );
    }

print ul( $list_items );

print end_html();
```

不难吧？你不必知道 CGI.pm 是怎么办到这一切的，你只需相信它做得到就行了。一旦学会放手让 CGI.pm 去做所有困难的事情，你便能专注在程序里有趣的部分了。

CGI.pm 模块还能做更多的事，像处理 cookie 信息、页面重定向以及多重页面表单等等。限于篇幅，我们不再详细讨论，要继续学习的话可以参考该模块文档中的例子。

数据库与 DBI

DBI（数据库接口，Database Interface）模块并未内置于 Perl 标准发行版本中，但这却是最热门的模块之一，许多人或多或少地需要连接数据库。DBI 的美妙之处在于，不管哪种常见的数据库，都可以用相同的接口进行操作，从 CSV 文件，到 Oracle 之类的大型数据库服务器，无不尽然。它还支持对 ODBC 的驱动操作，而且有些驱动程序是数据库厂商自行提供的。想通盘了解完整细节，请参阅《Programming the Perl DBI》一书，由 Alligator Descartes 和 Tim Bunce 合著（O'Reilly）。你也可以访问 DBI 模块的官方网站 *http://dbi.perl.org/*。

安装完 DBI 之后，你还必须安装 DBD（数据库驱动程序，Database Driver）。在 CPAN 上搜索一下就能得到一长串 DBD 模块列表。请安装与你的数据库服务器对应的驱动，并确定其版本与你的服务器一致。

DBI 是面向对象模块，但你不必为了要用它而去了解 OO 编程的全部细节，学习文档中的例子就好了。想要连接数据库，你得 use DBI 模块，并调用它的 connect 方法：

```
use DBI;

$dbh = DBI->connect($data_source, $username, $password);
```

变量 $data_source 指定了要连接的数据库信息，以及使用哪一种 DBD 作底层交互。对 PostgreSQL 数据库来说，驱动器是 DBD::Pg 模块，所以 $data_source 就像这样：

```
my $data_source = "dbi:Pg:dbname=name_of_database";
```

连上数据库之后，就可以开始准备查询、执行查询以及读取查询结果等一系列操作。

```
$sth = $dbh->prepare("SELECT * FROM foo WHERE bla");
$sth->execute();
@row_ary  = $sth->fetchrow_array;
$sth->finish;
```

完成工作后，断开与数据库的连接：

```
$dbh->disconnect();
```

当然，还有许多其他 DBI 能做的事，具体请参阅它的文档。

习题

以下习题解答见附录 A：

1. [15] 从 CPAN 安装 Module::CoreList 模块。输出 Perl 5.008 自带的所有模块的清单。要建立一个哈希，其键为指定 Perl 版本自带模块的名称，可以使用下面这行代码：

```
my %modules = %{ $Module::CoreList::version{5.008} };
```

第十二章

文件测试

早先我们已经演示过如何打开文件以输出其内容。通常，打开文件的操作会直接创建一个新文件，如果存在同名文件的话，还会清空该文件的内容。有些时候，你可能需要先检查一下同名文件是否存在；有些时候，你可能需要看看给定的文件究竟存在多久了；或者有些时候，你可能需要比较一些文件的大小，看是否都在某个级别以上、并且是否已经有一段时间无人使用。对于这类信息的取得，Perl 有一套完整的文件测试操作符可供你使用。

文件测试操作符

在运行那些会创建新文件的程序之前，让我们先检查指定的文件是否已经存在，以免无意之中覆盖了重要的电子表格或是宝贵的生日档案。要达到此目的，我们可以用 `-e` 文件测试操作符来测试文件存在与否：

```
die "Oops! A file called '$filename' already exists.\n"
  if -e $filename;
```

请注意，这个例子中 `die` 抛出的消息中并没有包含 `$!`，因为我们现在并不关心系统为何拒绝请求。接下来的例子，是要测试某个文件是否能保证时常更新。我们测试的是一个已经存在的文件句柄，而非表示文件名的字符串。假设我们需要某个程序的配置文件保持每周或每两周更新一次（也许这是一个病毒资料库）。如果文件在过去 28 天里都没变动过，显然是出了问题：

```
warn "Config file is looking pretty old!\n"
  if -M CONFIG > 28;
```

第三个例子就复杂多了。假设我们的硬盘空间已满，但是不想花钱再买硬盘，于是我们决定找出最大而且很久没用到的文件，将它们移到备份盘上。我们需要遍历文件列表【注 1】，看看哪些是大于 100KB 的文件。在确定某个文件够大之后，我们还得确定它已经超过 90 天没被访问过（这样我们才能确信它不太常用），才可以将它归档到磁带【注 2】：

```
my @original_files = qw/ Fred barney betty wilma pebbles dino bamm-bamm /;
my @big_old_files;  # 将要移到备份盘上的、既大且旧的文件列表
foreach my $filename (@original_files) {
  push @big_old_files, $filename
    if -s $filename > 100_000 and -A $filename > 90;
}
```

这是我们第一次看到的写法，你也许注意到了，`foreach` 循环里的控制变量是 `my` 变量。这等于声明了变量的有效范围位于循环内，所以使用 `use strict` 时这段代码还是可以工作。若没有 `my` 这个关键字，就会使用全局变量 `$filename`。

所有的文件测试操作符，看起来总是由一个连字符和某个字母组成，字母代表要进行何种测试，后面跟着要测试的文件名或文件句柄。大部分的测试都会返回或真或假的布尔值，但也有些会返回比较有趣的结果。请参考表 12-1 的完整列表，然后继续阅读后面的章节，了解有关特例的详细说明。

表 12-1：文件测试操作符及其意义

文件测试操作符	意义
-r	文件或目录，对目前（有效的）用户或组来说是可读的
-w	文件或目录，对目前（有效的）用户或组来说是可写的
-x	文件或目录，对目前（有效的）用户或组来说是可执行的
-o	文件或目录，由目前（有效的）用户拥有
-R	文件或目录，对实际的用户或组来说是可读的
-W	文件或目录，对实际的用户或组来说是可写的
-X	文件或目录，对实际的用户或组来说是可执行的
-O	文件或目录，由实际的用户拥有
-e	文件或目录，是存在的

注 1：实际上并不会像此例那样，在数组中放置一连串的文件。一般来说，会使用“文件名通配”（glob）或是“目录句柄”（directory handle）来从文件系统中读取。因为这里还没提到目录句柄（参见第十三章），我们只好以列表来示意了。

注 2：还有种比这个例子更高效的实现方法，你会在本章篇末看到。

表 12-1：文件测试操作符及其意义（续）

文件测试操作符	意义
-z	文件存在而且没有内容（对目录来说永远为假）
-s	文件或目录存在而且有内容（返回值是以字节为单位的文件大小）
-f	是普通文件
-d	是目录
-l	是符号链接
-S	是 socket 类型的文件
-p	是命名管道，也就是先入先出（fifo）队列
-b	是块设备文件（比如某个可挂载的磁盘）
-c	是字符设备文件（比如某个 I/O 设备）
-u	文件或目录设置了 setuid 位
-g	文件或目录设置了 setgid 位
-k	文件或目录设置了 sticky 位
-t	文件句柄是 TTY 设备（类似系统函数 isatty() 的测试；不能对文件名进行此测试）
-T	看起来像是文本文件
-B	看起来像是二进制文件
-M	最后一次修改后至今的天数
-A	最后一次访问后至今的天数
-C	最后一次文件节点编号（inode）变更后至今的天数

-r，-w，-x 和 -o 这几个操作符测试的是：有效用户或组【注 3】是否有相应的文件权限。所谓有效用户，指的是“负责”运行这个程序的人【注 4】。这些测试会查看文件的“权限位”（permission bit），以此判断哪些操作是允许的。如果系统使用访问控制列表（Access Control List，简称 ACL），那么测试将根据该列表进行判断。上述测试只能返回系统对操作的看法，但受实际情况限制，允许的事未必真的可行。比如说，对某个放在 CD-ROM 中的文件做 -w 测试可能为真，可实际上你却无法修改此文件；而对某个空文件进行 -x 测试时也可能会返回真，但实际上空文件又怎么能被执行呢？

注 3：测试操作符 -o 及 -O 只管用户 ID，而不理会组 ID。

注 4：好学的人请注意：相应的 -R、-W、-X 及 -O 测试，使用的是实际用户或组的 ID。这在你的程序以 set-ID 方式运行时相当重要。在这种情况下，它是调用程序的用户 ID。请参阅任何切实介绍高级 Unix 程序设计的书，进一步了解 set-ID 程序。

如果文件内容不为空，-s 会返回表示文件大小的数字，单位是字节。若仅用于条件判断，非零数字亦代表真值。

Unix 文件系统上【注5】有且仅有7种文件类型，分别可由以下七种文件测试操作符代表：-f、-d、-l、-S、-p、-b 和 -c。任何一个文件类型都应该符合其中一种。但如果你有指向某个文件的符号链接，那么 -f 和 -l 都会为真。所以，如果你想要知道某个文件是否为符号链接，最好先测试 -l（我们将会在第十三章学到更多关于符号链接的知识）。

文件时间测试操作符，诸如 -M、-A 和 -C（是的，都是大写），分别会返回从该文件最后一次被修改、被访问或者它的 inode 被更改后，到现在的天数【注6】（inode 是文件系统的索引条目，其中记录了某个文件的所有属性信息，但文件内容除外——有关细节请参阅系统函数 stat 的在线手册，或找本切实介绍 Unix 内部细节的书看看）。天数的值用浮点数表示，如果两天零一秒之前被修改的文件，可能会返回像 2.00001 这样的值（这里所说的天数并不是平常所说的自然天。举例来说，假如你在凌晨 1:30 的时候检查某个曾在午夜前一小时修改过的文件，则 -M 的返回值大约是 0.1，也就是过去的小时数换算到天的数字，这是一个相对时间）。

在检查文件的时间记录时，可能会得到像 -1.2 这样的负数，这表示文件最后一次被访问的时间戳是在未来 30 小时后！实际上，程序开始运行的那一刻，会被记录下来作为当前时间，而在做关于时间的文件测试时，计算两者之间的差距【注7】。所以负值可能表示已经运行很久的程序，找到某个刚刚才被访问过的文件。或者是谁不小心（也可能是故意的）将文件属性设成未来时间。

至于 -T 和 -B，则会测试某个文件是文本文件还是二进制文件。但是对文件系统有一点经验的人都知道，（至少在与 Unix 类似的操作系统下）没有任何位会告诉你它是二进制文件还是文本文件——那么 Perl 是如何办到的？答案是 Perl 作弊：它会打开文件，检查开头的几千个字节，然后作出一个合理的猜测。如果它看到很多空字节、不寻常的

注5：在许多非 Unix 文件系统下也行，但不是每种文件测试在任何地方都有意义。例如，非 Unix 系统上大概就找不到块设备文件。

注6：在非 Unix 的系统上可能会有所不同，因为有些系统记录的时间与 Unix 不同。例如在某些系统上，ctime 字段（即 -C 测试操作符返回的信息）是文件的创建时间（Unix 并不跟踪这个时间），而不是 inode 改变的时间。请参阅 *perlport* 在线手册。

注7：这取决于 $^T 变量的值。如果想求出相对于特定时刻的天数，可以改变它的值（可用 $^T = time; 这样的语句）。

控制字符而且还设定了高位（即第八位是 1）的字节，那么它看起来就是二进制文件；如果文件里没有许多奇怪的东西，而且它看起来像文本文件，那就猜测为文本文件。如你所料，这样总会有猜错的时候。如果文本文件内有很多瑞典文或是法文（法文和瑞典文都有许多设定了高位的字符，就像某些 ISO-8859- 开头的字符集，甚至是 Unicode 也一样），那么 Perl 可能就会被蒙骗，误以为它是一个二进制文件。因此，这种猜测并不完美。不过，如果你只想把编译过的文件和源文件分开，或是将 HTML 文件和 PNG 文件分开，那么这两种测试操作符还算够用。

你可能会以为 `-T` 和 `-B` 出现的结果必定相反，因为文件若不是文本文件，就该是二进制文件。但是，有两种特殊情况会让测试结果相同：如果文件不存在，两者都会返回假，因为它既不是文本文件也不是二进制文件；在空文件的情况下，两者都会返回真，因为它既是空的文本文件也是空的二进制文件。

关于 `-t` 的测试，如果被测试的文件句柄是一个 TTY 设备，测试的返回值就为真——简单来说，如果该文件可以交互，就判断为 TTY 设备，所以普通文件或是管道（pipe）都可以排除在外。当 `-t STDIN` 返回真的时候，通常意味着可以用交互方式向用户提出一些问题；若返回值为假，那就表示输入来源是个普通文件或是管道，而不是键盘。

至于其他的文件测试操作符，如果你不知道它们的意义也不用担心——要是你没听说过，也就不会用到。但如果你感兴趣的话，找一本切实介绍 Unix 程序设计的书吧。（在非 Unix 系统中，这些测试都会尽力模仿 Unix 系统的实现，要是碰上某个无法实现的功能，就会返回 `undef`。通常你都可以猜到实际测试的结果。）

如果文件测试操作符后面没写文件名或文件句柄（也就是只写了 `-r` 或 `-s`），那么默认的操作数就是 `$_` 里的文件名【注 8】。所以，若要测试一连串的文件名，找出哪些是可读的话，可以这么做：

```
foreach (@lots_of_filenames) {
  print "$_ is readable\n" if -r;  # 亦即 -r $_
}
```

但如果省略参数，请特别小心，任何接在测试操作符之后的，就算看起来不像，也会被当作要测试的目标。比方说，你想知道以 KB 为单位（而不是以字节为单位）的文件大小，那么你可能会直接把 `-s` 操作的结果除以 `1000`（或是 `1024`），像这样：

```
# 文件名保存在 $_ 中
my $size_in_K = -s / 1000;  # 糟糕!
```

注 8：测试操作符 `-t` 是个例外，因为此项测试对文件名（不可能是 TTY）而言无用武之地，所以默认情况下它测试 `STDIN`。

当 Perl 的语法解析器看到斜线时，不会认为那是一个除法操作符。因为它在看到 -s 操作符后，就要尝试寻找它的可选操作数，也就是要测试的文件，现在它看到的是一个斜线开头的，像是某个正则表达式，但找不到结尾，于是报错。要避免这种问题，可在文件测试操作符的两边加上括号：

```
my $size_in_k = (-s) / 1024;  # 默认使用 $_ 作测试
```

当然，写明要测试的文件，总是安全的做法。

同一文件的多项属性测试

如果要一次测试某个文件的若干属性，可以将各个文件测试组成一个逻辑表达式。例如我们只想操作那些既可读、又可写的文件，可以依此检查这两个属性并用 and 合并起来：

```
if( -r $file and -w $file ) {
    ... }
```

这可是个非常昂贵的操作，每次执行文件测试时，Perl 都得从文件系统取出所有相关信息（实际上，每次 Perl 都在内部做了一次 stat 操作，下节我们就会介绍）。虽然在做 -r 测试的时候，我们已经拿到了所有的相关信息，可到了做 -w 测试的时候，Perl 又去要相同的信息。多浪费啊！如果要对海量文件测试各种属性，这时性能问题就会非常明显。

Perl 有个特别的简写可以避免这种重复劳动。它就是虚拟文件句柄 _（就是那个下划线字符），它会告诉 Perl 用上次查询过的文件信息来做当前的测试。现在，Perl 就只需要查询一次文件信息即可：

```
if( -r $file and -w _ ) {
    ... }
```

我们并非只能在一条语句中连续使用 _。以下就是在两条 if 语句中使用的例子：

```
if( -r $file ) {
    print "The file is readable!\n";
    }

if( -w _ ) {
    print "The file is writable!\n";
    }
```

这么用的时候，必须要清楚代码最后一次查询的是否为同一个文件。若是在两个文件测试之间，又调用了某个子程序，那么最后一个查询的文件可能会变化。比如说，下面的

例子调用 lookup 子程序，在其内部对另一个文件做了一次测试。当子程序返回后，再执行文件测试时，文件句柄 _ 代表的就不是预想的 $file，而是 $other_file：

```
if( -r $file ) {
    print "The file is readable!\n";
    }

lookup( $other_file );

if( -w _ ) {
    print "The file is writable!\n";
    }

sub lookup {
    return -w $_[0];
    }
```

栈式文件测试操作

在 Perl 5.10 之前，如果要一次测试多个文件属性，只能分开为若干独立的操作，哪怕是为了节省点力气用虚拟句柄 _ 也不例外。比如说，我们想要测试某个文件是否可读写，就必须要分别作可读测试和可写测试：

```
if( -r $file and -w _ ) {
    print "The file is both readable and writable!\n";
    }
```

若能一次完成就更容易。Perl 5.10 允许我们使用“栈式（stack）”写法将文件测试操作符排成一列，放在要测试的文件名前：

```
use 5.010;

if( -w -r $file ) {
    print "The file is both readable and writable!\n";
    }
```

这个使用栈式写法的例子和上一个例子做相同的事，仅是语法上有所改变。注意，使用栈式写法时，靠近文件名的测试会先执行，次序为从右往左。不过通常测试次序不是很重要。

对于复杂情况来说，这种栈式文件测试特别好用。比如我们想要列出所有可读、可写、可执行，并隶属于当前用户的所有目录，只需要按恰当的顺序放置这些测试操作符：

```
use 5.010;

if( -r -w -x -o -d $file ) {
    print "My directory is readable, writable, and executable!\n";
    }
```

对于返回真假值以外的测试来说，栈式写法并不出色。像下面的例子，我们原本想要确认某个小于 512 字节的目录，可实际上没做到：

```
use 5.010;

if( -s -d $file < 512) { # 错啦! 千万不要这么做
    print "The directory is less than 512 bytes!\n";
    }
```

按其内部的实现方式展开，我们可以看到上面的例子实际上相当于如下的写法。整个合并起来的文件测试表达式成了比较运算的一个操作数：

```
if( ( -d $file and -s _ ) < 512 ) {
    print "The directory is less than 512 bytes!\n";
    }
```

当 -d 返回假时，Perl 将假值同数字 512 做比较。比较的结果就变为真，因为假等效为数字 0，而 0 永远小于 512。为了避免这种令人困惑的写法，还是用分开的方式写比较好，这对将来的维护程序员来说也更友善：

```
if( -d $file and -s _ < 512 ) {
    print "The directory is less than 512 bytes!\n";
    }
```

stat 和 lstat 函数

用前面介绍的文件测试操作符，已经可以获取某个文件或文件句柄的各种常用属性，但这只是一部分，还有许多其他属性信息没有对应的测试符。比如说，没有任何文件测试操作符会返回文件链接数，或是该文件拥有者的 ID (uid)。想知道文件所有其他的相关信息，请使用 stat 函数。此函数能返回和同名 Unix 系统调用差不多一样完整的文件信息（总之比你想知道的还多）【注 9】。函数 stat 的参数可以是文件句柄（包括虚拟文件句柄 _），或是某个会返回文件名的表达式。如果 stat 函数执行失败（通常是因为无效的文件名或是文件不存在），它会返回空列表；要不然就返回一个含 13 个数字元素的列表，具体意义见下面标量变量构成的列表：

注 9：在非 Unix 系统上，stat、lstat 以及文件测试操作符都应该返回“可能范围内最接近的值”。例如，没有用户 ID 的系统（也就是以 Unix 观点来看，该系统刚好只有一个“用户”）可能会返回零，以表示用户及组 ID，将唯一的用户当成系统管理员。如果 stat 或 lstat 执行失败，便会返回空列表。如果文件测试操作符的底层系统调用失败（或者在该系统上无法取得），则该测试操作符通常会返回 undef。请参阅 *perlport* 在线手册里各种系统的实现清单。

```
my($dev, $ino, $mode, $nlink, $uid, $gid, $rdev,
  $size, $atime, $mtime, $ctime, $blksize, $blocks)
    = stat($filename);
```

这些变量名代表 stat 函数返回的列表中对应的数据，你应该抽出点时间看看 *stat(2)* 在线手册里的详细说明。不过在这里，我们会简单地列出比较重要的几个：

$dev 与 $ino

即文件所在设备的编号与文件的 inode 编号。这两个编号决定了这个文件的唯一性，就好比是身份证一样。即使它具有多个不同的文件名（使用硬链接创建），设备编号和 inode 编号的组合依然是独一无二的。

$mode

文件的权限位集合，还包含其他信息位。如果你曾用 Unix 命令 *ls -l* 查看过详细（冗长）的文件列表，你会看到其中每一行都是由类似 -rwxr-xr-x 这样的字符串开始的。长度共 9 个字符的连字符和字母，刚好对应 $mode 最低的 9 个权限位【注 10】。以这个字串而言，其实就是八进制数值 0755。而开头的这一位，则表示文件的其他信息。所以如果需要用到权限模式，那你就得学会如何使用位运算操作符（本章稍后就会介绍）。

$nlink

文件或目录的（硬）链接数，也就是这个条目有多少个真实名称。这个数值对目录来说总会是 2 或更大的数字，对文件来说（通常）是 1。我们会在第十三章中介绍如何为文件创建链接，那时可以了解更多有关链接的信息。在 *ls -l* 的结果中，权限位之后的数字就是文件链接数。

$uid 与 $gid

文件拥有者的用户编号及组 ID。

$size

以字节为单位的文件大小，和 -s 文件测试操作符的返回值相同。

*$atime*，*$m*time 与 *$c*time

三种时间戳，但这里以系统时间格式来表示：一个 32 位的整数，表示从纪元（Epoch）算起的秒数。在 Unix 与某些其他的系统中，纪元是从公元 1970 年世界标准时间的午夜算起，但也有的系统上会不同。本章稍后将会进一步阐述如何将时间戳的值转换成有用数据。

注 10：字符串里的第一个字符并不代表权限位，而是表示此条目的类型：其中连字符代表一般文件，d 代表目录，l 代表符号链接。*ls* 命令是根据最低 9 位之上的其他位来判断文件类型。

对符号链接名调用 stat 函数，将会返回符号链接指向对象的信息，而非符号链接本身的信息（除非该链接指向的对象目前无法访问）。若你需要符号链接本身的信息（多半没用），你可以用 lstat（它会返回同样顺序、同样意义的内容）来代替 stat。如果 lstat 的参数不是符号链接，它会和 stat 一样返回空列表。

就像文件测试操作符一样，stat 和 lstat 的默认操作数是 $_。也就是说，底层的 stat 系统调用会对标量变量 $_ 里的文件名进行操作。

localtime 函数

你能获得的时间戳值（例如，使用 stat 函数）看起来通常会像 1180630098 这样。这对大多数人来说不太方便，除非你想通过减法来比较两个时间戳大小。所以，你可能需要将它转换成让人比较容易阅读的形式，好比 Thu May 31 09:48:18 2007 这样的字符串。Perl 可以在标量上下文中使用 localtime 函数完成这种转换：

```
my $timestamp = 1180630098;
my $date = localtime $timestamp;
```

在列表上下文中，localtime 会返回一个数字元素组成的列表，其中个别元素的取值可能会出乎你的意料：

```
my($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst)
  = localtime $timestamp;
```

$mon 是范围从 0 到 11 的月份值，很适合用来索引月份名。$year 是一个自 1900 年起算的年数，所以将这个值加上 1900 就是实际的年份。$wday 的范围从 0（星期天）到 6（星期六），$yday 则表示目前是今年的第几天，范围从 0（1 月 1 日）到 364 或 365（12 月 31 日）。

另外与此相关的，还有两个有用的函数。gmtime 函数和 localtime 函数一样，只不过它返回的是世界标准时间（俗称格林威治标准时间）。如果需要从系统时钟取得当前的时间戳，可以使用 time 函数。不提供参数的情况下，不论 localtime 或 gmtime 函数，默认情况下都使用当前 time 返回的时间值：

```
my $now = gmtime;  # 取得当前的世界标准时间的时间戳字符串
```

关于操作时间及日期的进一步信息，请参考附录 B 提到的诸多有用模块。

按位运算操作符

如果你需要逐位进行运算，比如对 stat 函数返回的权限位进行处理，就必须用到按位运算操作符（bitwise operator）。该操作符在二进制数据上进行数学运算。“按位与”操作符（bitwise-and，记作 &）会给出两边参数对应的位置中，哪些位*同时*都为 1。举个例子，表达式 10 & 12 得到的值是 8。“按位与”操作符只有在两边相应的位均为 1 的状况下才会产生 1。因此，10（写成二进制是 1010）和 12（二进制为 1100）的“按位与”运算结果是 8（二进制 1000，也就是 10 和 12 以二进制数字表示时，同时为 1 的位所组成的数字）。请参考图 12.1。

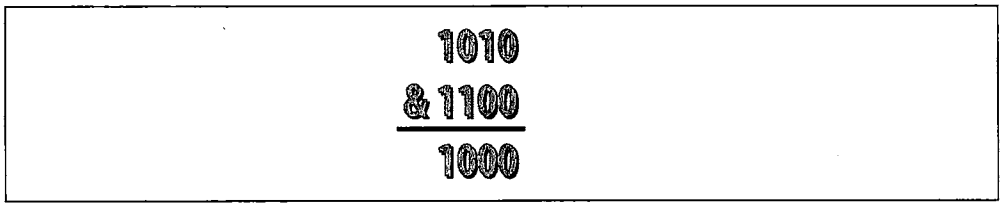

图 12-1：“按位与”操作

表 12-2 列出了各种位运算操作符及其意义：

表 12-2：按位运算操作符

表达式	意义
10 & 12	“按位与”——哪些位在两边都为真（此例得 8）
10 | 12	“按位或”——哪些位在任一边为真（此例得 14）
10 ^ 12	“按位异或”——哪些位在任何一边为真，但另一边为假（此例得 6）
6 << 2	“按位左移”——将左边操作数向左移动数位，移动位数由右边操作数指定，并以 0 来填补最低位（此例得 24）
25 >> 2	“按位右移”——将左边操作数向右移动数位，移动位数由右边操作数指定，并丢弃移出的最低位（此例得 6）
~ 10	“按位取反”，也称为取反码——返回操作数逐位反相之后的数值（此例得 0xFFFFFFF5，但请参考后面的说明）

好吧，下面来看几个例子，看看我们可以用这些操作符对 stat 函数返回的 $mode 信息如何进行按位操作。按位操作的结果可以给 chmod 使用（我们会在第十三章中进一步介绍此函数）：

```
# $mode 是从配置文件 CONFIG 的 stat 信息中取出的状态值
```

```
warn "Hey, the configuration file is world-writable!\n"
  if $mode & 0002;                                    # 配置文件有安全隐患
my $classical_mode = 0777 & $mode;                    # 遮蔽额外的高位
my $u_plus_x = $classical_mode | 0100;                # 将一个位设为 1
my $go_minus_r = $classical_mode & (~ 0044);          # 将两个位都设为 0
```

使用位字符串

按位运算操作符既可以操作位字符串（bitstring），也可以对整数进行操作。如果操作数都是整数，结果也会是整数（整数最少会是一个 32 位的整数，但是如果你的硬件支持更多位的整数，它也可能会更大。例如在 64 位的机器上，`~10` 的结果会是 `0xFFFFFFFFFFFFFFF5`，而不是 32 位机器上的 `0xFFFFFFF5`。）

不过，假如按位运算操作符的任一操作数是字符串，则 Perl 会把它当成位字符串来处理。换句话说，`"\xAA" | "\x55"` 的结果会是 `"\xFF"`。注意，这个例子里的值都是单字节（single-byte）的字符串，而结果是八位都为 1 的字节。Perl 对位字符串的长度没有限制。

这是少数 Perl 区分字符串和数字的地方。如果想了解利用按位运算操作符处理位字符串的细节，请参阅 *perlop* 在线手册。

习题

以下习题答案参见附录 A：

1. [15] 写一个程序，从命令行取得一串文件名，并汇报这些文件是否可读、可写、可执行以及是否的确存在。（提示：如果你可以写一个函数，一次做完这些测试会很方便。）如果先对文件做 *chmod* 为 0，你的程序会汇报什么？（也就是说，如果你使用 Unix 系统，*chmod 0 some_file* 这样的命令就会把文件标示成不可读、不可写也不可执行。）在大部分的 shell 下，用星号作为参数，代表当前目录下的所有文件。也就是说，你可以用 `./ex12-2 *` 这样的命令来向程序一次传送多个要测试属性的文件。

2. [10] 写一个程序，从命令行参数指定的文件中找出最旧的文件并且以天数汇报它已存在了多久。若列表是空的（也就是命令行中没有提及任何文件），那么它该做什么？

3. [10] 写一个程序，使用栈式文件测试操作符，列出命令行参数指定的所有文件，是否拥有者是你自己，以及是否可读、可写。

第十三章

目标操作

上一章我们运行程序时，创建的临时文件都保存在和程序文件相同的目录中，这么做有点乱。现代的操作系统都使用目录来组织和管理文件，比如将披头士（Beatles）的 MP3 文件和我们的小骆驼书各章的重要源文件分开存放，以免不小心把 MP3 文件寄给出版社。Perl 可以直接对目录进行操作，即使在不同的操作系统下，做法也还是差不多的。

在目录树中移动

程序运行时会以自己的工作目录（working directory）作为相对路径的起点。也就是说，当我们说起 `fred` 这个文件时，其实指的是当前工作目录下的 `fred`。

你可以用 `chdir` 操作符来改变当前的工作目录。它和 Unix shell 的 *cd* 命令差不多：

```
chdir "/etc" or die "cannot chdir to /etc: $!";
```

因为这是一个对操作系统的调用，所以发生错误时便会设定标量变量 `$!` 的值。如果 `chdir` 的返回值为假，则表示有些事情没有顺利完成。这时，你通常应该检查一下 `$!`。

由 Perl 程序启动的所有进程，都会继承 Perl 程序的工作目录（我们会在第十六章谈到）。但对于启动 Perl 程序的进程（像 shell【注 1】）而言，它的工作目录就不会随 Perl 程序的工作目录的改变而改变。这意味着，没办法写出任何 Perl 程序来代替 shell 里的 *cd* 命令，因为一旦退出 Perl 程序，又会回到开始的工作目录。

注 1：这并不是 Perl 的限制。实际上它是 Unix、Windows 和其他系统的一个特性。如果你真的想改变 shell 的工作目录，请参看 shell 的相关文档。

如果省略参数，Perl 会猜想你要回到自己的用户主目录（home directory），于是将当前工作目录设为此目录，这和在 shell 下使用不加参数的 *cd* 命令效果差不多。这是少数的，不以 `$_` 作为默认参数的情形之一。

有些 shell 能使用波浪号前缀来定位另一个用户的主目录（像 `cd ~merlyn`）。这个功能来自 shell，而非操作系统。因为 Perl 的 `chdir` 是通过直接调用操作系统实现的，所以这里无法使用这种波浪号开头的写法。

文件名通配

一般来说，shell 会将命令行里的文件名模式展开成所有匹配的文件名。这称为文件名通配（*globbing*）。比如说，假设你将 `*.pm` 这个文件名模式交给 *echo* 命令，shell 会将它展开成名称相匹配的文件列表：

```
$ echo *.pm
barney.pm dino.pm Fred.pm wilma.pm
$
```

这里的 *echo* 命令不必知道该如何展开 `*.pm`，因为 shell 会先将它展开，再交给 *echo* 处理。这也同样适合于 Perl 程序：

```
$ cat >show-args
foreach $arg (@ARGV) {
  print "one arg is $arg\n";
}
^D
$ perl show-args *.pm
one arg is barney.pm
one arg is dino.pm
one arg is Fred.pm
one arg is wilma.pm
$
```

请注意，*show-args* 完全不必了解如何进行文件名通配处理——放在 `@ARGV` 里的已经是展开好了的名称。

不过有时候在程序内部，也可能会想要用 `*.pm` 之类的模式。我们可以不花太多力气就把它展开成相匹配的文件名吗？当然！只要用 `glob` 操作符就行了：

```
my @all_files = glob "*";
my @pm_files = glob "*.pm";
```

其中，`@all_files` 会取得当前目录中的所有文件，并按字母顺序排序，但不包括以点号开头的文件，这和 shell 中的做法完全相同。`@pm_files` 得到的列表与之前在令

行使用 *.pm 时的相同。

其实，任何能够在命令行上键入的模式，都可以作为（唯一的）参数交给 glob 处理，如果要一次匹配多种模式，可以在参数中用空格隔开各个模式：

```
my @all_files_including_dot = glob ".* *";
```

其中，我们加上了“.*”参数以取得所有以点号开头的文件名。请注意，在引号括住的字符串里，两个条目之间的空格是有意义的：它分隔了两个要进行文件名通配处理的条目【注 2】。glob 操作符的效果之所以和 shell 完全相同，是因为在 Perl 5.6 版之前，它只不过是在后台调用 */bin/csh*【注 3】来执行展开的功能而已。因此文件名通配非常耗时，而且还可能在目录太大时（或别的情况下）崩溃。负责的 Perl 黑客（hacker）会避开文件名通配处理，而改用目录句柄（directory handle，本章稍后会详细讨论）。不过，如果你用的是新版的 Perl，就不必再担心这件事了。

文件名通配的另一种语法

虽然我们一直在介绍文件名通配，也介绍了 glob 操作符的用法，可在许多进行文件名通配处理的程序里你可能完全看不到 glob 这个词。为什么呢？嗯，那是因为过去有大部分的程序都是在 glob 操作符出现之前写的。它们使用尖括号语法（angle-bracket syntax）来调用此功能，看起来就跟读取自文件句柄差不多：

```
my @all_files = <*>; ## 效果和这样的写法完全一致：my @all_files = glob "*";
```

和双引号字符串内插的情形类似，尖括号内的变量也会被替换为变量当前的值，然后按照文件名通配展开成对应的文件名称列表：

```
my $dir = "/etc";
my @dir_files = <$dir/* $dir/.*>;
```

此处，因为 $dir 会被展开成它当前的值，所以最终会取得指定目录下的所有文件，而无论是否以点号开头。

这样说来，假如尖括号既表示从文件句柄读取，又代表文件名通配操作，那 Perl 如何决定用哪一种呢？嗯，因为合理的文件句柄必须是严格意义上的 Perl 标识符，所以如

注 2：Windows 用户可能习惯使用 *.* 这个文件名通配来代表“所有文件”。但其实它表示的是“所有中间包含点的文件名”，甚至对 Windows 平台的 Perl 来说也是这个意思。

注 3：如果 C shell 不存在，它会调用其他合法的替代程序。

果尖括号内是满足 Perl 标识符条件的，就作为文件句柄来读取；否则，它代表的就是文件名通配操作。举例来说：

```
my @files = <FRED/*>;  ## glob
my @lines = <FRED>;    ## 从文件句柄读取
my $name = "FRED";
my @files = <$name/*>; ## glob
```

上述规则的唯一例外，就是当尖括号内仅是一个简单的标量变量（不是哈希或数组元素）时，那么它就是间接文件句柄读取（*indirect filehandle read*【注 4】），其中变量的值就是待读取的文件句柄名称：

```
my $name = "FRED";
my @lines = <$name>; ## 对句柄 FRED 进行间接文件句柄读取
```

Perl 会在编译阶段决定它是文件名通配还是从文件句柄读取，因此和变量的内容无关。

假如你喜欢，也可以使用 `readline` 操作符来执行间接文件句柄读取【注 5】，让程序读起来更清楚些：

```
my $name = "FRED";
my @lines = readline FRED;  ## 从 FRED 读取
my @lines = readline $name; ## 从 FRED 读取
```

不过，因为间接文件句柄读取并不常见，并且通常也只用在简单的标量变量上，所以很少有用到 `readline` 操作符的机会。

目录句柄

若想从目录里取得文件名列表，还可以使用目录句柄（*directory handle*）。目录句柄看起来像文件句柄，使用起来也没多大的差别。你可以打开它（以 `opendir` 代替 `open`）、读取它的内容（以 `readdir` 代替 `readline`），然后将它关闭（以 `closedir` 代替 `close`）。只不过读到的是目录里的文件名（或其他东西的名称），而不是文件的内容。举例来说：

注 4：如果间接句柄是一个文本字符串，那么在 `use strict` 的情况下将受到符号引用（symbolic reference）的限制。此外，间接句柄还可能是符号表通配（typeglob）或某个 I/O 对象的引用，这种情况即使在使用了 `use strict` 的时候，也是可以正常工作的。

注 5：如果你使用 Perl 5.005 或以上的版本。

```
my $dir_to_process = "/etc";
opendir DH, $dir_to_process or die "Cannot open $dir_to_process: $!";
foreach $file (readdir DH) {
  print "one file in $dir_to_process is $file\n";
}
closedir DH;
```

和文件句柄一样，目录句柄会在程序结束时自动关闭，也会在用这个句柄再打开另一个目录前自动关闭。

目录句柄和旧版 Perl 里的文件名通配不同，它绝对不会启动另一个进程。所以，对于需要榨取更多计算能力的程序来说，前者可以提供更佳的性能。不过，目录句柄也是个低级操作符，也就是说我们必须自己多做点事。

例如，返回的名称列表并未按照任何特定的顺序排列【注6】。此外，列表里将会包含所有的文件，而不只是匹配某些模式的部分（比如刚才的 *.pm 文件名通配）。另外列表里包含了以点号开头的文件，特别注意 . 和 .. 也在其中【注7】。因此，如果我们只想处理名字以 *pm* 结尾的文件，则可以在循环内跳过部分处理：

```
while ($name = readdir DIR) {
  next unless $name =~ /\.pm$/;
  # ... 进一步的处理 ...
}
```

请注意，这是正则表达式的语法，而不是文件名通配。若想取得所有不以点号开头的文件，我们可以这么写：

```
next if $name =~ /^\./;
```

如果要排除 .（当前目录）和 ..（上层目录）两个条目，则可以直接写明：

```
next if $name eq "." or $name eq "..";
```

接下来要说明的是最让人迷惑的地方，所以请特别注意。readdir 操作符返回的文件名并不包含路径，它们只是目录下的文件名而已。所以，我们不会看到 */etc/passwd*，而只会见到 *passwd*(因为这是另一个与文件名通配操作的区别，所以很容易把人搞糊涂)。

所以得加上路径名称才是文件的全名：

注6：这其实是目录列表里（未经排序）的条目顺序，就像执行 *ls -f* 或 *find* 时得到的顺序一样。

注7：许多古老的 Unix 程序都犯了这个错误，以为 . 和 .. 一定是前两个返回的条目（无论是否排序）。如果你根本没这么想过，请忘记这段脚注，因为这是错误的猜想。事实上，我们现在已经后悔提到它了。

```
opendir SOMEDIR, $dirname or die "Cannot open $dirname: $!";
while (my $name = readdir SOMEDIR) {
  next if $name =~ /^\./;              # 跳过点号开头的文件
  $name = "$dirname/$name";            # 拼合为完整的路径
  next unless -f $name and -r $name;  # 只需要可读的文件
  ...
}
```

若是没有接上路径，文件测试操作符会在当前目录下查找文件，而不是在 `$dirname` 指定的目录下。这是使用目录句柄时最常犯的错误。

递归的目录列表

在你的 Perl 编程生涯刚开始的数十个小时内大都不需要进行递归目录访问。因此，我们不想现在就深入介绍如何用 Perl 脚本来实现 *find* 这样的功能。我们只会提醒你 Perl 内置的模块 `File::Find` 用起来是很方便的，可以用来实现绝妙的递归目录处理。另外也希望你不要编写自己的搜索例程。似乎新手在入门的数十个小时后都会想这么做，然后在“局部目录句柄”和“返回原始目录”之类的问题上受挫。

操作文件与目录

常常有人使用 Perl 来打理文件和目录，因为 Perl 是在 Unix 环境下成长起来，而且仍然主要为 Unix 服务，这一章可能看起来会比较偏向 Unix。值得庆幸的是在非 Unix 系统上，Perl 也能以同样的方式工作。

删除文件

我们之所以会创建文件，通常是为了让数据有个地方落脚。但是一旦数据过时，就该让文件消失。在 Unix shell 下，我们可以键入 *rm* 命令来删除文件：

```
$ rm slate bedrock lava
```

在 Perl 中则使用 `unlink` 操作符：

```
unlink "slate", "bedrock", "lava";
```

这将会把这三个文件放进碎纸机，从此消失在系统中。

既然 `unlink` 的参数是列表，`glob` 函数又恰好返回列表，我们只要融合两者，就可以一次删除多个文件：

```
unlink glob "*.o";
```

这跟我们在 shell 中 `rm *.o` 很像，但是不必启动外部的 *rm* 进程。所以可以更快地让这些重要的文件消失！

unlink 的返回值代表成功地删除了多少个文件。所以在第一个例子里，改用如下的写法可以检查它是否执行成功：

```
my $successful = unlink "slate", "bedrock", "lava";
print "I deleted $successful file(s) just now\n";
```

如果返回值是 3，我们当然知道文件全都被删除了；如果是 0，则表示没有删除任何文件。那如果是 1 或 2 呢？嗯，我们没办法知道被删除的是哪个。假如你一定要知道，请使用循环每次删除一个：

```
foreach my $file (qw(slate bedrock lava)) {
  unlink $file or warn "failed on $file: $!\n";
}
```

因为每次只删除一个文件，所以返回值不是 0（失败）就是 1（成功），这刚好可以当成布尔值来控制 warn 的执行。在这里 or warn 和之前第五章中介绍的 or die 是很相似的，只是后果没那么严重。我们在 *warn* 的信息后面加上了换行符，这样就不会显示是哪行程序发出的警告，因为这里出错的根源并非我们的程序。

当 unlink 执行失败时，内置的 $! 变量会被设成操作系统错误的相关信息。此变量只有在循环处理每一个文件的过程中才可用，因为每次系统调用失败时都会重设 $! 变量的内容。unlink 不能用来删除目录（这同不带参数的 *rm* 命令不能删除目录一样），你得使用稍后提到的 rmdir 函数。

在 Unix 上有个鲜为人知的事实：某个文件可能你无法读取、写入、执行。其实它根本就是别人的文件，但你还是可以将它删除。这是因为删除文件的权限跟文件本身的权限位无关，它取决于文件所在目录的权限位。

之所以提到这个，是因为 Perl 的初学者在测试 unlink 时，常会在建立一个文件后将它 *chmod* 成 0（这样就无法对它进行读写），看看这是否能让 unlink 执行失败。可是结果恰好相反，文件却像肥皂泡一样消失了【注 8】。不过，如果你真的想看到 unlink

注 8：有些人知道 *rm* 在删除这种文件时通常会提醒用户确认。不过 *rm* 是个命令，而 unlink 则是系统调用。系统调用不需要确认，也从来不说抱歉。

执行失败，只要试着删除 */etc/passwd* 或类似的系统文件就行了。因为这个文件是由系统管理员控制的，因此你无法将它删除【注 9】。

重命名文件

想为现有的文件取个新名字时，使用 rename 函数是很方便的做法：

```
rename "old", "new";
```

跟 Unix 的 *mv* 命令一样，这会将名为 old 的文件改为同一个目录下名为 new 的文件。你甚至可以将文件移到其他目录中：

```
rename "over_there/some/place/some_file", "some_file";
```

只要运行程序的用户拥有足够的权限，这会将其他目录中名为 some_file 的文件移动到当前目录里【注 10】。和大部分调用操作系统功能的函数一样，rename 执行失败时返回假，并且会将操作系统返回的错误信息存到 $! 里，从而可以（通常也应该）用 or die（或是 or warn）来向用户汇报问题。

新闻组里最常见【注 11】的 Unix shell 问题就是：如何批量把 .old 结尾的文件改名为 .new 结尾的。下面就是 Perl 擅长的做法：

```
foreach my $file (glob "*.old") {
  my $newfile = $file;
  $newfile =~ s/\.old$/.new/;
  if (-e $newfile) {
    warn "can't rename $file to $newfile: $newfile exists\n";
  } elsif (rename $file, $newfile) {
    ## 改名成功，什么都不需要做
  } else {
    warn "rename $file to $newfile failed: $!\n";
  }
}
```

此程序会先检查 $newfile 是否存在，因为只要用户具有删除目的文件的权限，rename 就会高高兴兴地覆盖掉现有的文件。加上这项检查，就可以降低损失数据的几

注 9：当然，如果你在尝试这种操作时太粗心，忘记了现在正以系统管理员的身份登录，那是罪有应得。

注 10：此外，它们还必须在同一个文件系统里。这条规则存在的理由稍后介绍。

注 11：批量改名不仅是历史常见问题，也是这些新闻组目前最常见的问题，同时还是常见问题集里名列榜首，并最早被解答的问题。世界真小。

率。当然，如果你打算覆盖已经存在的文件，比如 wilma.new，就不必在程序里先用 -e 来测试了。

循环里的前两行程序代码可以（通常也会）合并成这样：

```
(my $newfile = $file) =~ s/\.old$/.new/;
```

这种做法会先声明 $newline 并从 $file 里取得它的初始值，然后对 $newfile 进行替换。你可以把它读成：用右边的模式将 $file 变换成 $newfile。而考虑到优先级的因素，括号是必需的。

没准有些程序员会注意到：为什么替换运算中，左边的点号之前有反斜线，而右边却没有？其实这是因为两边的意义不同，左边的部分是正则表达式，右边的则可视为双引号内的字符串。所以我们需要用模式 /\.old$/ 来表示字符串结尾的 .old（因为不想替换掉 betty.old.old 这个文件名里第一次出现的 .old，所以得将锚位定在字符串结尾），但是在右边可以直接写成 .new 以作为替换字符串。

链接与文件

要进一步了解文件和目录的运作，先搞清楚 Unix 的文件及目录模型会有所帮助，即使你的系统跟 Unix 的运作方式稍有差异。限于篇幅，想深入了解文件系统的读者请参考任何清晰讲解 Unix 内部细节的资料。

“挂载的卷”指的是硬盘或相似的设备，例如磁盘分区、软盘、CD-ROM 或 DVD-ROM。其中可能含有任意数量的文件和目录。每个文件都存储在文件索引号（*inode*）对应的位置中，我们可以把它想象成磁盘上的门牌号码。某个文件或许会存在 inode 613 下，而另一个则可能存在 inode 7033 下。

不过，寻找某个特定的文件时，我们得从它的目录找起。目录是一种由系统管理的特殊文件。基本上目录是一份文件名和 inode 号对照表【注 12】。目录里除了其他的内容外，一定会有两个特殊条目：第一个是 .（点），代表目录本身；另一个则是 ..（点点），指的是目录结构中的上层目录【注 13】。图 13-1 展示了两个 inode。一个是名为 chicken

注 12：在 Unix 系统上（其他系统通常没有 inode、硬链接等概念），可以利用 *ls* 命令的 -i 选项来查看文件的 inode 号。不妨试着键入像 *ls -ail* 这样的命令。假如文件系统里有两个以上的条目具有相同的 inode 号，那么它们实际上是一个文件，只占用一份磁盘空间。

注 13：Unix 的根目录（*root*）没有上层目录。在根目录下 .. 和 . 都指向目录本身。

的文件，另一个是 Barney 的诗歌目录 /home/barney/poems，其中包含了 chicken 文件。文件的 inode 编号是 613，而目录的 inode 编号是 919（该目录的名称 poems，并没有在这张图中，因为当前目录本身的信息是被存放在上层目录中的）。目录中有三个文件（包括 chicken）和两个子目录（其中一个 inode 919 指向当前目录本身）的 inode 条目。

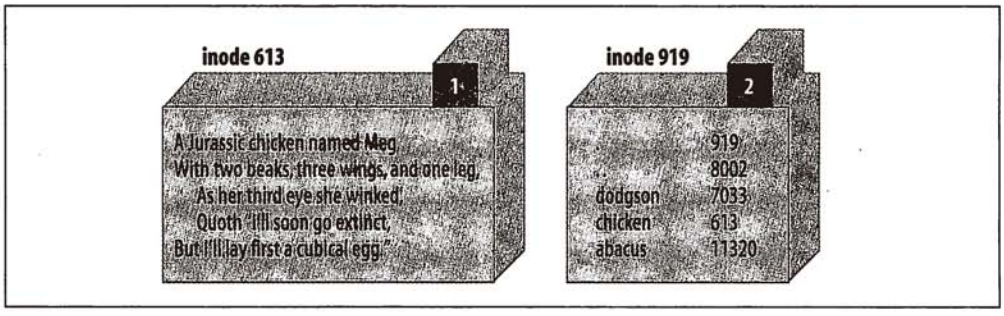

图 13-1：鸡在蛋前

要在目录里创建一个新文件时，系统会新增一个条目来记录文件名与新的 inode 编号。系统怎么知道哪个 inode 可以用呢？答案是每个 inode 都有自己的*链接数*。如果 inode 并未在任何目录里出现，它的链接数就一定是零。因此，所有链接数为零的 inode 都可以用来存放新的文件。每当 inode 被列入目录中，链接数就会递增；当它在目录的列表里被删除时，链接数就会递减。对上图里的文件 chicken 来说，inode 的链接数是 1，我们把链接数显示在 inode 数据右上方的小框里。

不过，有些 inode 会出现在多个目录的列表里。举例来说，前面提过每个目录都会有 . 这个条目，它会指回目录本身的 inode。所以任何目录的链接数都至少有两个：一个位于它的上层目录的列表里，另一个位于它本身的列表里。除此之外，如果里面有子目录，则每个子目录还会通过 .. 条目再增加一个链接【注 14】。在图 13-1 中，目录的 inode 链接数为 2（右上角的小框）。链接数代表的是该 inode 的真实名称的数量【注 15】。那么一般文件的 inode 也可以在目录的列表里重复出现吗？当然可以。假设 Barney 在上述的目录里用 Perl 的 link 函数建立了一个新的链接：

注 14：这表示目录的链接数一定等于子目录的数量加上 2。有些系统确实如此，而有些系统则可能有不同的实现。

注 15：在传统的 *ls -l* 输出中，硬链接的数量会显示在权限标记（如 -rwxr-xr-x）的右边。这个数值对目录来说总是大于 1，而对普通文件来说则几乎全是 1，其中的道理你现在应该知道了。

```
link "chicken", "egg"
  or warn "can't link chicken to egg: $!";
```

这就好像在 Unix shell 命令行上键入 `ln chicken egg` 的效果。`link` 在成功时会返回真，失败时则会返回假，并且设定 `$!` 的值，Barney 可以在错误信息里引用这个值。这个程序运行后，`egg` 这个名称就会指向文件 `chicken`，反之亦然。`egg` 和 `chicken` 这两个名称现在没有主从先后之别，而且（你大概猜到了）要仔细调查才能知道是先有鸡还是先有蛋。图 13-2 展示了新的情况，图中有两个指向 inode 613 的链接。

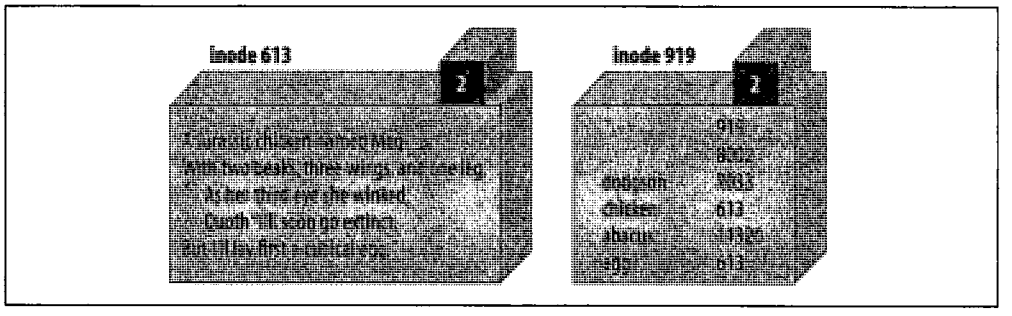

图 13-2：蛋链接到鸡

所以，这两个文件名都会指向磁盘上的同一处。如果文件 *chicken* 里面有两百个字节的数据，那么 *egg* 里也会有相同的两百个字节，而且总共还是两百个字节（因为这只不过是同一份文件的两个名称）。如果 Barney 为 *egg* 文件新增了一行文字，则 *chicken* 文件的结尾处也会出现相同的一行文字【注 16】。现在如果 Barney 意外（或者蓄意）删除了 *chicken* 文件，数据并不会丢失，因为还可以用 *egg* 这个文件名来访问。反过来说如果他删除的是 *egg* 文件，那么还可以访问 *chicken*。当然如果他把两个文件都删了，数据就丢失了【注 17】。关于目录列表里的链接还有一条规定：在目录列表中所有 inode 指向的文件都必须在本挂载卷中【注 18】。这样一来，即使将物理介质（可能是磁盘）移到另一台机器上，其中的目录和文件链接仍然有效。正因为这个，`rename` 虽然可以将

注 16：当你尝试建立链接、更改文件时，请注意大部分的文本编辑器都不会“就地”(in place)编辑文件，而是将改动过的内容存入副本。如果 Barney 使用文本编辑器来修改 *egg*，那么最后很可能会有一个名为 egg 的新文件以及一个名为 *chicken* 的旧文件。它们是两个不同的文件，而不是一个文件的两个链接。

注 17：虽然系统不一定会立刻覆盖掉这个 inode，但在链接数减到零时通常很难救回数据。最近做过备份了吗？

注 18：有个例外，就是存储设备根目录里的 .. 条目，它会指向被挂载设备的目录。

文件移到别的目录里，但是来源和目的地必须位于同一个文件系统（挂载卷）上。如果要跨磁盘移动文件，就必须另外分配新的 inode 条目。对于简单的系统调用来说，这种操作实在太复杂了。

链接的另一个限制，就是不能为目录建立额外的名称。这是因为目录必须按照层次排列，如果没有这条规则，*find* 和 *pwd* 之类的工具程序很快就会在文件丛林中迷失了。

因此，不能增加目录的链接数，也不能跨挂载卷链接。幸好，链接的这些固有限制是可以绕过的，只要使用另一种链接方式：*符号链接*【注 19】。符号链接（也叫做*软链接*，以便和前面所说的真正的硬链接区分开来）是目录里的一种特殊条目，用来告诉系统到别的地方找找看。假设 Barney（在那个诗歌的目录下）使用 Perl 的 `symlink` 函数建立了一个软链接，如下所示：

```
symlink "dodgson", "carroll"
  or warn "can't symlink dodgson to carroll: $!";
```

这和 Barney 在 shell 下执行 *ln -s dodgson carroll* 命令的效果类似。图13-3显示了执行的结果，包括 inode 7033 的那首诗在内。

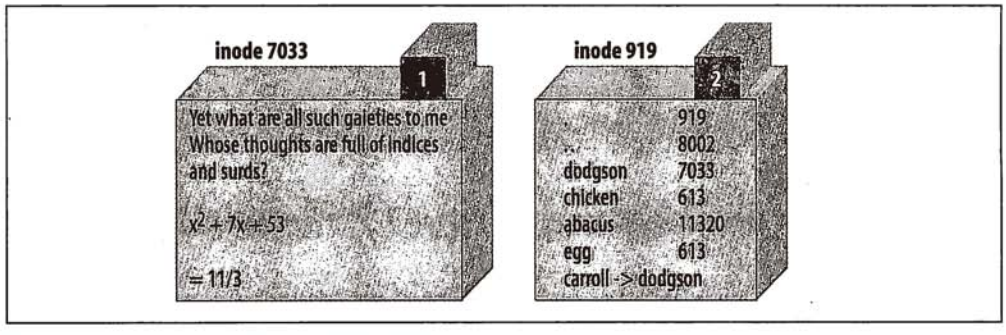

图 13-3：指向 inode 7033 的符号链接

现在如果 Barney 想读取 */home/barney/poems/carroll*，由于系统会自动跟随符号链接，所以结果和直接打开 */home/barney/poems/dodgson* 相同。但是这个新名称并不是文件真正的名称，因为（如图 13-3 所示）inode 7033 的链接数仍然是 1 而已。符号链接只是告诉系统：你如果是来这里找 *carroll* 的话，请到 *dodgson* 那里去。

符号链接和硬链接不同，它可以跨文件系统为目录建立软链接（也就是一个新的目录名）。事实上，符号链接能指向任何文件名，而不管它放在哪个目录里，甚至还可以指

注 19：有些非常古老的 Unix 系统不支持符号链接，但是近年来已经很罕见了。

向不存在的文件！不过，软链接不像硬链接那样可以防止数据被删除，因为它并不会增加 inode 的链接数。如果 Barney 删掉了 *dodgson*，系统就不能跟随 *carroll* 这个软链接了【注20】。虽然 *carroll* 这个条目还在，但是尝试读取它会得到像 `file not found` 这样的错误。此时，以 `-l 'carroll'` 进行文件测试会返回真，而 `-e 'carroll'` 则会返回假：它是个符号链接，可实际上并不存在。

由于软链接可以指向目前还不存在的文件，所以在创建文件时也很有用。Barney 将大部分的文件放在自己的主目录 */home/barney* 下，不过他也经常需要访问某个名称很长、难以键入的目录 */usr/local/opt/system/httpd/root-dev/users/staging/barney/cgi-bin*。因此他建立了 */home/barney/my_stuff* 这个符号链接来指向那个名称很长的目录，这样他要进去就很容易了。假如他（从自己的主目录）创建了 *my_stuff/bowling* 这个文件，该文件的真实名称将会是 */usr/local/opt/system/httpd/root-dev/users/staging/barney/cgi-bin/bowling*。下个星期，要是系统管理员将 Barney 的文件移到 */usr/local/opt/internal/httpd/www-dev/users/staging/barney/cgi-bin* 目录，那么 Barney 只要更改符号链接指向的目录，他和他的程序就又可以轻松地找到文件了。

在你的系统上，*/usr/bin/perl* 或 */usr/local/bin/perl* 常会是符号链接，可能两者都指向真正的 Perl 二进制文件。假设你是系统管理员，刚编译了一份新版的 Perl。旧版的 Perl 当然还在运行，而你也不想因为升级导致停顿。当你准备要更新 Perl 时，只要更改一两个符号链接就行了。这样一来，任何以 `#!/usr/bin/perl` 开头的程序都会自动改用新版的 Perl。要是出了什么问题（虽然不大可能），只要改回原来的符号链接，就又能切换到旧版的 Perl。（不过因为你是个好管理员，所以会在改版之前先通知用户测试新的 */usr/bin/perl-7.2*。改版之后，你也会保留旧版的 Perl 一个月，并让所有需要这段过渡时间的用户将程序的第一行改成 `#!/usr/bin/perl-6.1`。）

软硬链接都很有用，这个事实可能让人惊讶。很多非 Unix 系统则没有任何链接机制，因此造成使用困难。在这些系统中，符号链接被实现为“快捷方式”或“别名”，参考 *perlport* 在线手册了解这些平台最新的进展。

要取得符号链接指向的位置，请使用 `readlink` 函数。它会返回符号链接指向的位置，或在参数不是符号链接时返回 `undef`：

```
my $where = readlink "carroll";              # 得到 "dodgson"
my $perl = readlink "/usr/local/bin/perl";   # 告诉你实际的 perl 程序究竟躲在何处
```

注 20：　当然，删除 *carroll* 只会删除符号链接而已。

这两种链接都可以用 unlink 移除，你现在该了解这个名字的含义了吧。unlink 只是从目录里移除该文件名的链接条目，并将它的链接数递减，必要时释放 inode。

建立及移除目录

在现有的目录下建立新的目录是件很容易的事，只要调用 mkdir 函数即可：

```
mkdir "fred", 0755 or warn "Cannot make fred directory: $!";
```

没错，返回值为真表示成功，失败时则会设定 $! 的值。

可是第二个参数 0755 是什么意思呢？它代表目录的初始权限【注 21】(将来随时可以再更改)。写成八进制数值，是因为它会被解释成三位一组的 Unix 权限值，适合用八进制来表达。没错，就算在 Windows 或 MacPerl 中，你也需要略懂 Unix 的权限值，才有办法使用 mkdir 函数。0755 是个不错的设定，因为它赋予你所有的权限，而其他人只能读取却不能更改任何内容。

注意 mkdir 函数并不要求你用八进制写这个值，它只是需要某个数字（直接量或运算结果都行)。但除非你能快速心算出八进制的 0755 等于十进制的 493，否则还是让 Perl 来算比较方便。此外，如果你不小心遗漏了数字开头的零，就会得到十进制的 755，这等于八进制的 1363，那是一个相当奇怪的权限组合。

正如第二章提过的，想当成数字来用的字符串，即使以 0 开头，也不会被解释成八进制数字。所以下面这么写是行不通的：

```
my $name = "fred";
my $permissions = "0755";  # 危险，不能这么用
mkdir $name, $permissions;
```

糟糕，因为 0755 会被当成十进制处理，所以相当于我们用奇怪的 01363 权限值建立了一个目录。要正确处理字符串，请使用 oct 函数，它能强行把字符串当成八进制数字处理，无论它是否以零开头：

```
mkdir $name, oct($permissions);
```

当然，在程序中指定权限时，不必使用字符串，直接用数字就行了。通常是在使用用户键入权限值时才会需要额外的 oct 函数。举例来说，假设我们从命令行取得参数：

注 21： 权限值通常会被 umask 的值修改。更详细的说明，请参阅 umask(2) 在线手册。

```
my ($name, $perm) = @ARGV;  # 从命令行最先传入的两个参数，分别是目录名称和权限
mkdir $name, oct($perm) or die "cannot create $name: $!";
```

变量 $perm 的值一开始就被当成字符串处理，所以 oct 函数会将它正确地解释成通用的八进制表示法。

想移除空目录时，可以使用 rmdir 函数，它的用法和 unlink 函数很像，只是每次调用只能删除一个目录：

```
foreach my $dir (qw(fred barney betty)) {
  rmdir $dir or warn "cannot rmdir $dir: $!\n";
}
```

对非空的目录调用，rmdir 操作符会执行失败。你可以先用 unlink 来删除目录的内容，再试着移除应该已经清空的目录。举例来说，假设我们需要一个目录来存放程序运行时产生的许多临时文件：

```
my $temp_dir = "/tmp/scratch_$$";        # 在临时文件的名称中，使用了当前进程号
mkdir $temp_dir, 0700 or die "cannot create $temp_dir: $!";
...
# 将临时目录 $temp_dir 作为所有临时文件存放的场所
...
unlink glob "$temp_dir/* $temp_dir/.*"; # 删除临时目录 $temp_dir 中所有的文件
rmdir $temp_dir;                        # 现在是空目录，可以删除了
```

新建的临时目录名将会包含当前的进程标识符，每个当前运行着的进程都有这么一个独一无二的数字代号，在 Perl 里会把这个代号存储在变量 $$ 中（类似于 shell）。这么做是为了避免和别的进程冲突，只要它们也在路径名称里包含进程标识符。（事实上，通常在进程标识符之外还会加上程序名称。所以，如果这个程序名是 quarry，那么目录名称多半应该像 /tmp/quarry_$$ 这样。）

在程序结尾处，最后的 unlink 应该会移除临时目录里所有的文件，然后 rmdir 函数才有办法将清空后的目录删除。不过，如果我们在临时目录里创建了子目录，那么 unlink 操作符在处理它们时将会失败，rmdir 也会跟着失败。请参考 Perl 自带的 File::Path 模块，里面的 rmtree 函数提供了比较完整的解决方案。

修改权限

Unix 的 *chmod* 命令可用来修改文件或目录的权限。Perl 里对应的 chmod 函数也能进行一样的操作：

```
chmod 0755, "fred", "barney";
```

和许多其他的操作系统接口函数一样，chmod 会返回成功更改的条目数量，哪怕只有一个参数，它也会在失败时将 $! 设成合理的错误信息。第一个参数代表 Unix 的权限值（即使在非 Unix 的 Perl 版本也一样）。这个值通常会写成八进制，理由和前面介绍的 mkdir 相同。

Unix 的 *chmod* 命令能接受用符号表示的权限（例如 +x 或 go=u-w），但是 chmod 函数并不接受【注 22】。

更改隶属关系

只要操作系统许可，你可以用 chown 函数来更改一系列文件的拥有者以及它们所属的组。chown 会同时更改拥有者和所属组，它们必须以数字形式的用户标识符及组标识符来指定。例如：

```
my $user = 1004;
my $group = 100;
chown $user, $group, glob "*.o";
```

如果要处理的不是数字，而是像 merlyn 这样的字符串呢？答案很简单，只要用 getpwnam 函数将用户名翻译成数字，再用相应的 getgrnam【注 23】函数来把组名翻译成数字：

```
defined(my $user = getpwnam "merlyn") or die "bad user";
defined(my $group = getgrnam "users") or die "bad group";
chown $user, $group, glob "/home/merlyn/*";
```

这里我们用 defined 函数来确认返回值不是 undef，也就是说验证指定的用户或组是存在的。

正确调用时 chown 函数会返回受影响的文件数量，在错误发生时则会设定 $! 的值。

修改时间戳

在某些罕见的情况下，可能需要修改某个文件最近的更改或访问时间以欺骗其他程序，

注 22：除非你从 CPAN 安装了 File::chmod 模块，这个模块能使 chmod 升级，从而支持符号表示的权限值。

注 23：这两个函数的名称可说是有史以来最丑陋的。但是请不要骂 Larry，他只是沿用伯克利那群人取的名字而已。

这时可以使用 utime 函数来造假。它的前两个参数是新的访问时间和更改时间，其余的参数是要修改时间戳的文件名列表。时间的格式采用的是内部时间戳的格式（也就是第十二章中介绍的 stat 函数的返回值列表格式）。

“right now” 是个很方便的时间戳，time 函数能以正确的格式返回此值。所以，如果想修改当前目录下所有的文件，让它们看起来是在一天前被更改，却是在此刻被访问的，只要这么写就行了：

```
my $now = time;
my $ago = $now ? 24 * 60 * 60;  # 一天的秒数
utime $now, $ago, glob "*";     # 将最后访问时间改为当前时间，最后修改时间改为一天前
```

当然，你可以随意修改文件的时间戳，将它设成未来或过去的任何时间（直到我们能用 64 位的时间戳之前，在 Unix 上只能设成 1970 年到 2038 年之间，其他系统则可能各有不同）。也许你可以利用这个技巧来创建某个目录，用来存放你写的时光隧道小说的手稿。

在文件有任何更动时，第三个时间戳（ctime 值）一定会被设成 “now”，所以没有函数可以修改它（就算用 utime 修改成功，它也会立刻被设回 “now”）。这是因为 ctime 主要是给增量备份的程序用的：如果某个文件的 ctime 比备份盘上的新，那它就该再次备份了。

习题

下面的程序可能会造成危险！请在没什么重要文件的目录下测试它们，以免不小心删除了重要的数据。

以下习题答案参见附录 A：

1. [12] 写一个程序，让用户键入一个目录名称并且从当前目录切换过去。如果用户键入一行空白字符，则会以主目录作为默认目录，所以应当会切换到他本人的主目录中。切换过去后，输出该目录的内容（不含以点号开头的文件）并按照英文字母顺序排列。（提示：用目录句柄还是用文件名通配更容易呢？）如果切换目录失败则应显示警告信息，但不必输出目录的内容。
2. [4] 修改前一题的程序，让它输出所有文件，包括以点号开头的文件。
3. [5] 如果你在前一题使用的是目录句柄，那么请以文件名通配重写一次；如果使用的是文件名通配，那么请以目录句柄重写一次。

4. [6] 编写功能和 *rm* 类似的程序，删除命令行指定的任何文件（不用支持 *rm* 的所有参数）。

5. [10] 编写功能和 *mv* 类似的程序，将命令行的第一个参数重命名为第二个参数（不必处理 *mv* 的各种选项或任何额外的参数）。别忘了第二个参数可以是目录。假如它是目录，请在新目录中使用原来的文件名。

6. [7] 如果你的系统支持，写一个功能和 *ln* 类似的程序，建立从第一个参数到第二个参数的硬链接（不必处理 *ln* 的各种选项或额外的参数）。如果系统不支持硬链接，那只要输出关于它本来会进行的操作的信息就行了。注意：这个程序和前一题有点像，希望这个提醒可以节省写程序的时间。

7. [7] 如果操作系统支持，请修改上一题的程序，让它接受可能出现在其他参数之前的 *-s* 选项。此选项表示要建立的是软链接，而非硬链接（即使系统无法使用硬链接，也请用这个程序试试看是否至少能建立软链接）。

8. [7] 如果操作系统支持，写一个程序，让它在当前目录下查找所有符号链接并输出它们的值（和 *ls -l* 的格式一样：`name -> value`）。

第十四章

字符串与排序

在 Perl 擅长处理的问题中，约有 90% 与文本处理有关，其余 10% 则覆盖了其他领域。所以毋庸置疑，Perl 的文本处理能力很强，之前我们用正则表达式解决的那些问题就是证明。不过，有时正则表达式引擎对你而言可能太过理想化，你要的是更简单的字符串处理功能，这也就是本章的主题所在。

在字符串内用 index 搜索

搜索子串其实也就是要找出它在主串中的相对位置。可以借助 index 函数解决这个问题。例如下面的用法：

```
$where = index($big, $small);
```

Perl 会在 $big 字符串里寻找 $small 字符串首次出现的地方，并且返回一个整数代表第一个匹配字符的位置，返回的字符位置是从零算起的。如果子串是在字符串最开始的位置找到的，那么 index 会返回 0；如果是在第二个字符，则返回值为 1；如果无法找到子串，那么会返回 -1【注 1】。在这个例子里面 $where 会得到 6：

```
my $stuff = "Howdy world!";
my $where = index($stuff, "wor");
```

另一种理解位置的方法，就是把它当成要走到子串之前需要跳过的字符数。因为 $where 是 6，所以我们知道，必须跳过 $stuff 的前 6 个字符，才会走到 wor。

注 1：之前使用 C 的程序员肯定会注意到，这类似于 C 语言的 index 函数。现在的 C 程序员也可能知道，但要想真正理解目前的话题，你最好能成为以前的 C 程序员。

index 函数每次都会返回首次出现子串的位置。不过，你可以再加上可选的第三个参数，来指定开始搜索的地方，这样 index 就不会从字符串的最开头寻找，而是从该参数指定的位置开始寻找子串：

```
my $stuff  = "Howdy world!";
my $where1 = index($stuff, "w");                # $where1 为 2
my $where2 = index($stuff, "w", $where1 + 1);   # $where2 为 6
my $where3 = index($stuff, "w", $where2 + 1);   # $where3 为 -1 (没找到)
```

当然，要重复搜索某个子串时通常会使用循环。第三个参数相当于可能的最小返回值，如果子串无法在该位置或其后被找到，那么返回值就是 -1。

偶尔会需要搜索子串最后出现的位置【注 2】。这个结果可以用 rindex 函数计算。在下面的例子中，我们可以找到最后一个斜线，它在串中的位置是 4：

```
my $last_slash = rindex("/etc/passwd", "/");  # 值为 4
```

rindex 函数也有可选的第三个参数，但是这里是用来限定返回的最大位置：

```
my $fred = "Yabba dabba doo!";
my $where1 = rindex($fred, "abba");               # $where1 为 7
my $where2 = rindex($fred, "abba", $where1 - 1); # $where2 为 1
my $where3 = rindex($fred, "abba", $where2 - 1); # $where3 为 -1
```

用 substr 处理子串

substr 操作符只处理较长字符串中的一部分。它的用法如下：

```
$part = substr($string, $initial_position, $length);
```

它需要三个参数：一个字符串、一个从零起算的起始位置（类似 index 的返回值）以及子串的长度。找到的子串会被返回：

```
my $mineral = substr("Fred J. Flintstone", 8, 5);  # 值为 "Flint"
my $rock = substr "Fred J. Flintstone", 13, 1000;  # 值为 "stone"
```

在上面的例子里你可能已经注意到了，假如子串的长度（此例为 1000 个字符）超出字符串的结尾，Perl 不会抱怨，只不过你会得到比预期长度（1000）短的字符串。如果你想要一直取到字符串结尾，那么不论字符串长短，只要省略第三个参数（子串长度）就行了。做法如下：

注 2：当然，其实并非最后一个。Perl 会从字符串的尾端开始找，一直找到开头，但只返回第一个找到的位置，就结果而言是相同的。返回的数值也同样是从零开始起算，与计算子串的位置是同样的概念。

```
my $pebble = substr "Fred J. Flintstone", 13;  # 为 "stone"
```

一个较大字符串中子串的起始位置可以为负值，表示从字符串结尾开始倒数（比如，位置 -1 就是最后一个字符【注 3】）。在下面的例子中，位置 -3 是从字符串结尾算起的第三个字符，也就是字母 i 的位置：

```
my $out = substr("some very long string", -3, 2);  # $out 为 "in"
```

如你所愿，index 与 substr 可以紧密合作。在下面的例子里，我们会取出以字符 l 的位置开头的子串：

```
my $long = "some very very long string";
my $right = substr($long, index($long, "l") );
```

接下来就很有意思了：假如字符串是个变量，你就可以修改该字符串被选取的部分【注 4】：

```
my $string = "Hello, world!";
substr($string, 0, 5) = "Goodbye";  # $string 现在的值为 "Goodbye, world!"
```

如你所见，用来取代的（子）字符串的长度并不一定要与被取代的子串长度相同，字符串会自行调整长度。如果这样还不能让你眼前一亮，你还可以用绑定操作符（=~）只对字符串的某部分进行操作。下面的例子只会处理字符串的最后 20 个字符，将所有的 fred 替换成 barney：

```
substr($string, -20) =~ s/fred/barney/g;
```

说实话，我们在自己写的程序中从来都没用到这个功能，而你大概也不会用到。不过知道 Perl 能够比你需要的更有用，这样的感觉蛮不错的，不是吗？

substr 与 index 能办到的事多半也能用正则表达式做到。所以，请选择最适合解决问题的方法。但是 substr 与 index 通常会快一点，因为它们没有正则表达式引擎的额外负担：它们总是区分大小写，它们不必担心元字符，而且也不会设定任何的内存变量。

注 3：这与第三章介绍的数组检索方式类似。数组应该由 0 开始（表示第一个元素）往右数，或是从 -1（表示最后一个元素）开始往左数。子串的位置也是从 0 开始（表示第一个字符）往右数，或是从 -1（表示最后一个字符）往左数。

注 4：从技术上来讲，只要是左值都行。但此名词的精确定义超出了本书的范围，你可以简单地想成所有能放在赋值号（=）左边的东西。通常该处放的都是变量，但是如同这里所示，也可以放 substr 操作符。

如果不想给 substr 函数赋值（因为乍看之下会觉得有点奇怪），你也可以通过传统的4个参数的方法来使用它【注5】，其中第四个参数是替换子串：

```
my $previous_value = substr($string, 0, 5, "Goodbye");
```

返回值是替换前的子串。当然，你总是能在空上下文中使用这个函数，这样可以轻易丢掉返回值。

用 sprintf 格式化数据

sprintf 函数与 printf 有相同的参数（可选的文件句柄参数除外），但它返回处理过的字符串，而不会把它打印出来。此函数方便之处在于，你可以将格式化后的字符串存放在变量里以便稍后使用。此外，你也可以对结果进行额外的处理，单靠 printf 做不到这些：

```
my $date_tag = sprintf
  "%4d/%02d/%02d %2d:%02d:%02d",
  $yr, $mo, $da, $h, $m, $s;
```

在上面的例子里，$data_tag 会得到类似 2038/01/19 3:00:08 的结果。格式字符串作为 sprintf 的第一个参数，会在某些格式数值前置零，这种用法并未在第五章中提及。格式串中数值字段的前置零，表示必要的时候会在数值前面补零，以符合要求的宽度。如果格式数值未被前置零，那么日期与时间字符串里的数值就没有前置零，只有前置空格，比如 2038/ 1/19 3: 0: 8。

sprintf 格式化的财务数据

sprintf 的一种常见用法就是用来格式化小数点后具有特定精度的数值，比如若要将 2.49997 这个金额显示成 2.50，而不是 2.5，我们可以使用 "%.2f" 这个字符串轻松完成格式化：

```
my $money = sprintf "%.2f", 2.49997;
```

四舍五入能够使得数字精简并且易读，但是绝大多数情况下应该保留数据的精度，只在输出时做四舍五入运算。

注5：这里指的是“函数调用”的传统风格，而不是指 Perl 的传统风格。因为这个写法是不久之前才引进 Perl 的。

如果手头有个财务数据非常大，需要使用逗号来分隔才能有效阅读，那么下面这个子程序能提供很多便利【注 6】：

```
sub big_money {
  my $number = sprintf "%.2f", shift @_;
  # 下面的循环中，每次在匹配到的合适位置加一个逗号
  1 while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
  # 在正确的位置补上美元符号
  $number =~ s/^(-?)/$1\$/;
  $number;
}
```

这个子程序用了一些前面没见过的技巧，但不难推测其含义。子程序的第一行是在对第一个（也是唯一的）参数进行格式化，让它在小数点之后刚好有两位数字。也就是说，假如参数是 12345678.9，那么 $number 就会是 12345678.90。

程序代码的下一行用到了 while 修饰词。就像之前第十章中介绍修饰词时提到的，我们可以将它改写成一个传统 while 循环：

```
while ($number =~ s/^(-?\d+)(\d\d\d)/$1,$2/) {
  1;
}
```

这到底是在做什么？其实这里的意思是，只要这个替换运算返回真（表示成功），就去执行循环主体。但是循环里的程序没有任何作用！对 Perl 来说，这没关系，这不过是要让我们知道，该语句的主要目的是在执行条件表达式（替换运算），而不是这个无用的循环主体。这么做时习惯上会使用数值 1 来占个位子，其实任何数值都可以使用【注 7】。下面这行程序代码与上面的循环有相同的效果：

```
'keep looping' while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
```

所以，现在我们知道替换运算是该循环的真正目的。但是此处的替换运算又做了什么事？别忘了，这里的 $number 是个像 12345678.90 这样的字符串。上面的模式会匹配字符串的第一个部分，但是它不会越过小数点（你知道为什么吗？）。内存变量 $1 会是 12345，而 $2 会是 678，所以这个替换运算会让 $number 变成 12345,678.90（别忘了，它不会匹配小数点，所以字符串后面的部分并不会改变）。

注 6：是的，我们也知道并非全世界都是三个数字一组、用逗号分隔，并且用美元作货币符号。不过这是一个不错的例子，不必计较细节。

注 7：也就是说，随便写什么都一样。其实，Perl 会对常量表达式进行优化，因此不会导致任何运行时等待。

你知道模式开头的减号有什么用处吗？（提示：这个减号只能在字符串中的一个地方出现），在本节结束时，万一你还没找到答案，可以看看我们的解答。

整个替换过程并非到此为止。既然替换运算成功了，这个无事可做的循环就会再来一次。因为这次模式只能替换逗号之前的部分，所以 `$number` 会变成 `12,345,678.90`。于是在每次循环执行后，替换运算就会在数字中加一个逗号。

循环的工作还没完呢。因为上一次替换成功了，所以又会重新开始循环。但是由于模式必须匹配从字符串开头的至少4个数字，所以这一次它无法匹配任何东西，于是就到了循环的结尾。

为什么我们不直接使用 `/g` 修饰符来进行全局查找与替换，以省下 `1 while` 循环造成的麻烦与困惑呢？这是因为必须从小数点倒回处理，而不是从字符串开头依次处理，所以不能这么做。像这样将逗号插入一个数值里，只用 `s///g` 是没办法做到的【注8】。你想到减号的作用了吗？它让程序也可以处理负号开始的数值字符串。程序代码的下一行也有一样的效果：它会把美元符号放在正确的地方，所以 `$number` 会变成 `$12,345,678.90`；如果是负数，则会变成 `-$12,345,678.90`。请注意，美元符号不一定会是字符串的首字母，不然这行代码会简单许多。最后，结尾的程序代码会返回我们已经格式化得漂漂亮亮的财务数据，可以准备送到年度报表里打印了。

高级排序

之前我们曾经在第三章中介绍如何利用内置的 `sort` 操作符，以 ASCII 码序对列表排序。但是如果你想要按数字大小进行排序，或是以不区分大小写的方式进行排序呢？你也许还想按照存储在哈希内的信息（对列表）进行排序。Perl 能以任何需要的顺序来为列表排序。从这里到本章结束为止，我们会看到所有上述这些例子。

Perl 允许你建立自己的排序规则子程序，或简称排序子程序，来实现你想要的排序方式。乍听到排序子程序这个术语时，如果你上过计算机科学的课，脑海中可能会浮现出冒泡排序、希尔排序和快速排序。然后你会说："拜托，别再谈这些了！"请放心，事情没那么复杂，其实还相当简单。Perl 其实知道怎么对列表排序，它只是不知道要用什么样的规则，所以排序子程序只是用来说明具体的规则。

注8：至少还得使上几招我们从没教过你的正则技巧，才能完成任务。Perl 核心开发小组总是在尝试超越极限，所以在任何 Perl 的书中写"不能"都要小心。

为什么需要这样？仔细想想，排序其实就是比较一堆东西，然后将它们依序排好。由于不可能一次比较所有东西，所以最终一定都是两两相比，并根据两者间的顺序进行定位，从而让全体成员就位。Perl 已经知道这些步骤了，只是不知道你要如何确定两者的顺序，而这就是需要由你来写的程序。

这就是说排序子程序并不需要比较许多元素，只要能比较两个元素就行。只要它能确定两个元素的顺序，Perl 就有办法（反复向排序子程序提问）返回排好序的数据。

排序子程序的定义和普通的子程序几乎相同。它会被反复调用，每次都会检查要排序列表中的两个元素。

假如子程序要比较两个待排序的参数，你可能会先写出如下代码：

```
sub any_sort_sub {   # 实际上这么写不能正确工作，这里只是为了方便说明问题
  my($a, $b) = @_;   # 声明两个变量并给它们赋值
  # 在这里开始比较 $a 和 $b
  ...
}
```

但是请注意，排序子程序会一次次地被调用，往往会运行上百或上千次。在子程序开始的地方声明变量 $a 与 $b 并给它们赋值，看起来只会花一丁点时间。但是，把这些时间乘以排序子程序可能被调用的次数，就可以看出，这对整体性能会造成不小的影响。

我们并不会这么做，事实上这么做是行不通的。其实在子程序开始之前，Perl 已经帮我们办好了这些事。实际写出的排序子程序并不会有前面例子的第一行，$a 与 $b 都已经被自动赋值好了。当排序子程序开始运行时，$a 与 $b 会是两个来自原始列表的元素。

子程序会返回一个数值，用来描述两个元素之间的顺序，就像 C 语言中 qsort(3) 的调用机制，但调用它的是 Perl 内建的排序机制。假如在结果列表中 $a 应该在 $b 之前，排序子程序就会返回 -1；如果 $b 应该在 $a 之前，它就会返回 1。

如果 $a 与 $b 的先后不必区分，则该子程序会返回 0。为什么会有这样的情况呢？也许你正在进行一个不区分大小写的排序，而被比较的两个串是 fred 和 Fred；或者，你正在进行一个按数值大小的排序，而这两个数字相等。

现在我们可以写出如下的排序子程序：

```
sub by_number {
  # 排序子程序，使用 $a 和 $b 这两个变量进行比较
  if ($a < $b) { -1 } elsif ($a > $b) { 1 } else { 0 }
}
```

要使用这个排序子程序，请把它的名字（去掉 & 符号）写在 sort 关键字与待排序的列表之间。下面的例子会把按数字大小排好序的结果列表放进 @result 里：

```
my @result = sort by_number @some_numbers;
```

我们将这个排序子程序命名为 by_number，因为这个名称描述了它的排序方式。不过更重要的是，你可以将这一行用来排序的程序代码读成“sort by number”，就像在说英语一样。许多排序子程序的名称都是以 by_ 开头的，用以描述它们的排序方式。类似的，我们也可以把排序子程序命名为 numerically，不过这样要输入更多的字符，因此也比较容易写错。

请注意，在排序子程序中，我们并不需要花力气声明并设定 $a 与 $b，如果我们这么做，该子程序反而无法正常运作。请让 Perl 帮忙设定 $a 与 $b，我们只要编写它们的比较方式。

事实上，我们可以让它更简单高效。因为常常需要用到这样的三路比较，所以 Perl 提供了一个方便的简写。在这种场合应该使用飞碟操作符（<=>）【注 9】。这个操作符会比较两个数字并且返回 -1、0 或 1，好让它们依数字排序。所以，我们可以把那个排序子程序写得更好看，如下所示：

```
sub by_number { $a <=> $b }
```

因为飞碟操作符只能用来比较数值，所以很容易猜到，会有另一个相应的三路字符串比较操作符：cmp。这两个操作符非常易记，而且一目了然。飞碟操作符与数值比较类的操作符 >= 一脉相承，而 cmp 操作符与字符串比较类的操作符 ge 一脉相承。当然 cmp 是三个字母而 ge 是两个字母，但是别忘了 cmp 也有三个返回值而 ge 是两个【注 10】。当然 cmp 的顺序与 sort 默认的排序规则相同。所以你不必自己编写下列子程序，因为它和 sort 默认的排序方法相同【注 11】：

```
sub ASCIIbetically { $a cmp $b }

my @strings = sort ASCIIbetically @any_strings;
```

不过，cmp 可以用来建立更复杂的排序顺序，例如不区分大小写的排序：

注 9：这么称呼是因为《星际大战》里面的某种钛战机就是这样。起码我们觉得很像。

注 10：这并非偶然。Larry 这么做是为了让 Perl 更好学，也更好记忆。他骨子里就是个语言学家，所以他深知学习语言的痛苦。

注 11：如果你也在写一本 Perl 入门书并且需要举例说明，就可以这么写。

```
sub case_insensitive { "\L$a" cmp "\L$b" }
```

这个例子里，我们比较了 $a 里的字符串（强制转换成小写）和 $b 里的字符串（强制转换成小写），以产生不区分大小写的排序规则。

要注意的是，我们并没有修改被比较元素，我们只是使用它们的值而已。这一点很重要：因为性能的需要，$a 和 $b 并不是数据项的拷贝，它们实际上是原始列表元素的临时化名。所以，如果我们改变它们的话，就会弄乱原始的数据。千万别这么做，Perl 不支持也不建议这种行为。

如果排序子程序像我们的例子一样简单（大部分的情况下都很简单），你就可以让程序代码更为简单，只要把子程序内嵌到函数名的位置就行了。做法如下：

```
my @numbers = sort { $a <=> $b } @some_numbers;
```

事实上，在新潮的 Perl 开发界，几乎不会有人写额外的排序子程序，你常常会看到的就是这种内嵌的排序子程序。

假设要以递减的顺序进行排序，reverse 函数可以帮助我们轻松解决：

```
my @descending = reverse sort { $a <=> $b } @some_numbers;
```

这里有个小窍门。比较操作符（<=> 与 cmp）是短视的，它们并不知道哪个操作数是 $a，哪个操作数是 $b，只知道哪一个值在左边，哪一个在右边。所以，如果我们把 $a 与 $b 对调，比较操作符每次就会得到相反的结果。换句话说，下面这么做也能得到反向排序的结果：

```
my @descending = sort { $b <=> $a } @some_numbers;
```

稍加练习之后，你就可以一眼看出这一行在做什么。它是递减比较（因为 $b 在 $a 之前，也就是递减的顺序），而且是数值比较（因为它使用飞碟操作符，而不是 cmp）。所以，它代表以反向顺序进行的数值排序。在较高的 Perl 版本中，这两种方法并没有太明显的差别，因为 reverse 已经被当成是 sort 的一个修饰词了，在处理时会做特殊的优化，避免把反向排序理解成排序后再做翻转。

哈希按值排序

一旦轻松掌握列表排序，就会碰到按哈希的值进行排序的问题。好比故事的三个主角昨晚跑去打保龄球，他们的积分存储在下面的哈希里。我们希望能用适当的顺序将名字打印出来，也就是积分最高的人在最上面。所以我们需要按积分来对此哈希排序：

```
my %score = ("barney" => 195, "fred" => 205, "dino" => 30);
```

```
my @winners = sort by_score keys %score;
```

当然，我们实际上无法按积分来对哈希排序，这只是口头上的说法而已。无法对哈希排序！虽然我们之前曾对哈希排序，但其实是对哈希键排序（以 ASCII 码序）。现在我们仍要对哈希键进行排序，只是现在的规则是比较它们在哈希中的值。在这个例子里，我们需要的结果是：三个主角名的列表，按照他们的保龄球积分进行排序。

这个排序子程序相当容易实现。我们要的是积分的数值比较，而不是名字。换句话说，我们不该比较 $a 与 $b 这两个球员名字，而是比较 $score{$a} 与 $score{$b} 这两个积分。如果你也想到这点了，那么答案就呼之欲出了。如下所示：

```
sub by_score { $score{$b} <=> $score{$a} }
```

让我们来仔细分析其运作原理。假如它第一次被调用时，Perl 把 $a 设定为 barney，而 $b 则是 fred。所以这次的比较是 $score{"fred"} <=> $score{"barney"}，哈希查表之后也就是 205 <=> 195。别忘了飞碟操作符是短视的，所以当它看到 205 在左边，而 195 在右边时，它就机械的说："不，原始顺序不对，$b 应该在 $a 之前"。所以它会告诉 Perl，fred 应该在 barney 之前。

它第二次被调用时，$a 仍然是 barney，而 $b 则变成了 dino。短视的飞碟操作符看到 30 <=> 195，30 在左边，195 在右边，它就会认为原始顺序是对的。$a 应该在 $b 之前，也就是 barney 应该在 dino 之前。最终 Perl 得到了排序的结果：fred 是冠军、barney 第二名，然后是 dino。

为什么这里的比较将 $score{$b} 放在 $score{$a} 之前，而不是反过来呢？因为我们想要将按积分降序排列，由分数最高者依次往下排列。所以只要稍加练习，就可以一眼看出：$score{$b} <=> $score{$a} 表示按积分降序排列。

按多值排序

忘记登记昨晚第四个球员的分数了，其实应该是如下的哈希：

```
my %score = (
  "barney" => 195, "fred" => 205,
  "dino" => 30, "bamm-bamm" => 195,
);
```

现在显然 bamm-bamm 和 barney 积分相同。所以在排序完成之后，哪一个应该排在前面呢？我们不能预测，因为比较操作符在比较这两个数值时，发现两边都看到相同的分数，就会返回零。

也许这无关紧要，不过通常我们更喜欢可预测的排序。如果有多个选手同分的话，他们当然都应该排在一起，但是这些名字应该按 ASCII 码序排列。这样的排序子程序该怎么写呢？其实也很简单：

```
my @winners = sort by_score_and_name keys %score;

sub by_score_and_name {
  $score{$b} <=> $score{$a}   # 根据分数降序排列
    or
  $a cmp $b                   # 分数相同的再按名字的 ASCII 码序排列
}
```

这里是怎么运作的？嗯，如果飞碟操作符看到两个不同的数值，那这就是我们想要的比较运算。它会返回 -1 或 1，而这两个都为真，所以在这里低优先级的短路操作符 or 会将表达式的其他部分跳过，并返回我们所要的比较结果（别忘了，短路操作符 or 会返回最后执行的表达式的结果）。但是，如果飞碟操作符看到两个相同的分数，它会返回 0。因为这个值为假，所以让 cmp 操作符获得执行的机会，于是就返回对哈希键进行字符串比较的结果。也就是说，如果同分的话，就会按字符串的顺序进行最终裁决。

我们知道，使用 by_score_and_name 这个排序子程序时，它绝不会返回 0（知道为什么吗？答案在脚注里【注 12】）。所以，我们知道排列的顺序都是可预测的。也就是说，如果今天的数据和明天的数据都一样，那么今天和明天的答案也会一样。

当然，没有什么理由限制排序子程序只能做两级排序。下面列出的程序代码能对借书证编号列表进行五级排序输出【注 13】。这个例子里的排序根据的是每个读者的未缴罚金（用 &fines 子程序计算，在此未列出）、目前他们借阅的本数（取自 %items）、他们的姓名（先按姓排，后按名字排，两者都取自哈希），最后是借书证的编号，以防前面的信息都相同：

```
@patron_IDs = sort {
  &fines{$b} <=> &fines{$a} or
  $items{$b} <=> $items{$a} or
  $family_name{$a} cmp $family_name{$b} or
  $personal_name{$a} cmp $family_name{$b} or
  $a <=> $b
```

注 12：让其返回零的唯一情况，是当两个字符串完全相同时。而现在处理的是哈希的键，我们早就知道它们绝不会相同。如果你在调用 sort 时传入了一个列表，内有重复（完全相同）的字符串，那么比较这些字符串时就会返回零，但此处传入的是哈希的键。

注 13：虽然在史前时代很不寻常，但在现代化的社会里，确实是有可能需要进行五级排序。

```
} @patron_IDs;
```

习题

以下习题答案参见附录 A：

1. [10] 编程读入一连串的数字并将它们按值排序，将结果以靠右对齐的格式输出。请以下列数据来测试你的程序，或使用从 O'Reilly 网站（见本书的序）取得的 numbers 文件：

```
17 1000 04 1.50 3.14159 -10 1.5 4 2001 90210 666
```

2. [15] 编程以不区分大小写的字符顺序把下列的哈希数据按姓氏排序后输出。当姓一样时，就以名字排序（还是一样，不区分大小写）。也就是说，输出结果中的第一个名字应该是 Fred，最后一个应该是 Betty。所有姓相同的人应该要排在一起。千万别更改原始数据。这些名字应该以它原来的大小写形式被显示出来。（你可以在下载的文件中找到 *sortable_hash* 这个文件，里面有产生下面这个哈希的程序代码。）

```
my %last_name = qw{
  fred flintstone Wilma Flintstone Barney Rubble
  betty rubble Bamm-Bamm Rubble PEBBLES FLINTSTONE
};
```

3. [15] 编程在输入字符串中找出特定子串出现的位置并将其输出。例如：输入字符串为 "This is a test."，而子串是 "is"，则程序应该会汇报位置 2 和 5；如果子串是 "a"，程序应该会汇报 8；如果子串是 "t"，程序将汇报什么？

第十五章

智能匹配与 given-when 结构

要是计算机自己能弄明白我们想要做的事，并做完它，那该有多好啊！Perl 已经尽力去这么做了，在你想要数字的时候给你数字，想要字符串的时候给你字符串，想要单个值的时候给你标量，想要列表的时候给你列表。加上现在 Perl 5.10 提供的智能匹配操作符和 given-when 控制结构，就更完美了。

智能匹配操作符

新的智能匹配操作符（写作 ~~）会根据需要自己决定该用何种方式比较两端的操作数。如果操作数看起来像是数字，那就按数字来比较大小。如果看起来像是字符串，就按照字符串来比较。如果某一端操作数是正则表达式，那就当作模式匹配来执行。它还能完成许多复杂的任务，如果换成功能相当的实现，会是一大堆代码。所以有了它，能省不少力气。

这个 ~~ 看起来和第八章介绍过的绑定操作符（=~）很像，不过 ~~ 更能干些。有时候，它甚至就能代替绑定操作符。之前，你已经学会使用绑定操作符来指定正则表达式匹配字符串 $name：

```
print "I found Fred in the name!\n" if $name =~ /Fred/;
```

现在，用智能匹配操作符代替绑定操作符，也能完成同样的任务：

```
use 5.010;

say "I found Fred in the name!" if $name ~~ /Fred/;
```

智能匹配操作符看到左侧有个标量，右侧有个正则表达式，于是它自己推断出应该执行模式匹配操作。不算惊天动地，但已经开始了微妙的改进。

在处理更复杂的情况时，智能匹配操作符才会大显身手。比方说，你想在哈希 %names 中查找任何匹配 Fred 的键，如果找到就打印一条消息出来。你无法用 exists 判定，因为它需要给定确切的键。当然，你可以用 foreach 来遍历每个键，尝试用正则表达式匹配，跳过那些不匹配的，直到发现要找的键，保存到标记变量 $flag 中，然后用 last 跳出循环：

```
my $flag = 0;
foreach my $key ( keys %names ) {
    next unless $key =~ /Fred/;
    $flag = $key;
    last;
    }

print "I found a key matching 'Fred'. It was $flag\n" if $flag;
```

唷！这么麻烦，连解释起来都费力，不过这么写也有好处，那就是各种 Perl 5 版本都支持。可有了智能匹配操作符，只要把哈希写在左侧，把正则表达式写在右侧，就搞定了：

```
use 5.010;

say "I found a key matching 'Fred'" if %names ~~ /Fred/;
```

之所以智能匹配操作符知道该怎么做，是因为它看到了一个哈希和一个正则表达式。遇到这两种操作数时，智能匹配操作符就知道该遍历 %names 的所有键，用给定的正则表达式逐个测试。如果找到匹配的那个键，它就知道该停下来了，并立即返回真。这里的匹配和对标量的匹配不太一样。它很聪明，能因地制宜，知道怎么做才是正确的。并且它集数种不同的操作于一身，仅仅一个操作符就能解决各式各样的问题。

如果想要比较两个数组（为简单起见，暂时只考虑相同长度数组之间的比较），可以按数组索引依次遍历，取出相同位置的两个元素来比较。如果比较下来两者相等，则令计数器 $equal 自增 1。循环结束后，如果 $equal 的数字和数组 @names1 的长度一致，就说明这两个数组是完全相同的：

```
my $equal = 0;
foreach my $index ( 0 .. $#names1 ) {
    last unless $names1[$index] eq $names2[$index];
    $equal++;
    }

print "The arrays have the same elements!\n"
    if $equal == @names1;
```

还是要说，这么做太麻烦了。就不能有更轻松一点的实现方法吗？等等！智能匹配操作符如何？直接把两个数组放在 ~~ 两端。就这么点代码做了和上面一样的事情，而且几乎算不上是程序：

```
use 5.010;

say "The arrays have the same elements!"
    if @names1 ~~ @names2;
```

好吧，再来看一个例子。在调用了某个函数之后，如果要检查它的返回结果是否在某个集合中。回想第四章介绍的 max() 函数，你已经知道 max() 将会返回给定列表中最大的那个数字。你可以将 max 的返回结果和传给它的参数列表比较，还是用之前的那种老式笨拙的写法：

```
my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );

my $flag = 0;
foreach my $num ( @nums ) {
    next unless $result == $num;
    $flag = 1;
    last;
    }

print "The result is one of the input values\n" if $flag;
```

你已经知道我们会怎么说：这么做太麻烦啦！把中间那些代码都扔掉吧，改用 ~~ 好了。这就容易多了：

```
use 5.010;

my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );

say "The result [$result] is one of the input values (@nums)"
    if @nums ~~ $result;
```

使用智能匹配时，对两边操作数的顺序没有要求，倒过来写也行。智能匹配操作符才不关心谁在哪边呢：

```
use 5.010;

my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );

say "The result [$result] is one of the input values (@nums)"
    if $result ~~ @nums;
```

智能匹配操作符符合“交换律”（和高中代数中的术语一样），这意味着操作数的顺序无关。智能匹配操作符和加法、乘法算符一样，总是能得到相同的结果。并且无论实际执行的是哪种匹配，操作数的顺序都无关，所以下面两行是等效的：

```
use 5.010;

say "I found a name matching 'Fred'" if $name ~~ /Fred/;
say "I found a name matching 'Fred'" if /Fred/ ~~ $name;
```

智能匹配操作的优先级

现在你已经看到了，智能匹配操作符是这么简洁灵巧。如果还需要进一步了解各种情况下它会如何工作，请仔细查看 perlsyn 在线手册中“Smart matching in detail”部分的表格。下面的表 15-1 列出了一部分。

表 15-1：智能匹配操作符对不同操作数的处理方式

例子	匹配方式
`%a ~~ %b`	哈希的键是否一致
`%a ~~ @b`	至少 %a 中的一个键在列表 @b 之中
`%a ~~ /Fred/`	至少一个键匹配给定的模式
`%a ~~ 'Fred'`	哈希中某一指定键 $a{Fred} 是否存在
`@a ~~ @b`	数组是否相同
`@a ~~ /Fred/`	有一个元素匹配给定的模式
`@a ~~ 123`	至少有一个元素转化为数字后是 123
`@a ~~ 'Fred'`	至少有一个元素转化为字符串后是 'Fred'
`$name ~~ undef`	$name 确实尚未定义
`$name ~~ /Fred/`	模式匹配
`123 ~~ '123.0'`	数字和（看起来 numish 的）字符串是否大小相等
`'Fred' ~~ 'Fred'`	字符串是否完全相同
`123 ~~ 456`	是否大小相等

当使用智能匹配操作符时，Perl 会按此图表自上而下查看适用的操作数对。先找到哪一种搭配就选择对应的操作。操作数的顺序并不重要。假如你有一个数组和哈希要进行智能匹配：

```
use 5.010;

if( @array ~~ %hash ) { ... }
```

Perl 先去找对应于一个哈希和一个数组的操作数类型，结果要求 @array 中至少有一个元素是哈希 %hash 的键。这很容易，因为对于这两种操作数类型，只有一种处理情况。那要是两个都是标量呢？

```
use 5.010;

if( $fred ~~ $barney ) { ... }
```

就这里的代码而言，你无法判断究竟要做哪种具体的匹配操作。因为 Perl 需要查看 $scalar1 和 $scalar2 里边存的究竟是什么类型的数据。在拿到具体的数据之前，Perl 无法决断该如何做。是要按数字比较？还是要按字符串比较？

要弄明白 Perl 将如何比较 $fred 和 $barney，就要让 Perl 先看下它们各自的值，再遵从我们上面介绍的规则办事。它会按优先级自上而下看过来，若发现合适的情况就进行对应的比较。这儿有个小窍门：Perl 认得出那些看起来像数字的字符串，我们将此类字符串称作 *numish* 字符串，比如 '123'，'3.14149' 等等。这里我们都用了引号，说明它们本质上还是字符串，也就是一串字符序列。而 Perl 却可以悄然转换这种字符串为数字，也不发出任何警告。如果 Perl 看到一个数字和一个 numish 字符串在智能匹配操作符的两侧，它就会按照数字来比较大小。否则，就按照字符串来进行比较。

given 语句

given-when 控制结构能够根据 given 的参数，执行某个条件对应的语句块。这是 Perl 用来应对 C 语言的 switch 语句的等效物，只不过更具 Perl 色彩，所以更时尚，名字也更新潮些。

这儿的几行代码，从命令行取第一个参数，$ARGV[0]，然后依次走一遍 when 条件判断，看看是否可以找到 Fred。每个 when 语句块都对应于不同的处理方式，下面的例子中，我们将从最宽松的条件开始测试：

```
use 5.010;

given( $ARGV[0] ) {
    when( /fred/i ) { say 'Name has fred in it' }
    when( /^Fred/ ) { say 'Name starts with Fred' }
    when( 'Fred'  ) { say 'Name is Fred' }
    default         { say "I don't see a Fred" }
    }
```

given 会将参数化名为 $_【注 1】，每个 when 条件都尝试用智能匹配对 $_ 进行测试。当然为了概念清晰，你可以用显式智能匹配的方式改写上面的例子：

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it' }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred' }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred' }
    default               { say "I don't see a Fred" }
    }
```

如果 $_ 不能满足任何的 when 条件，Perl 就会执行 default 语句块。下面是几次运行后的输出结果：

```
$ perl5.10.0 switch.pl Fred
Name has fred in it
$ perl5.10.0 switch.pl frederick
Name has fred in it
$ perl5.10.0 switch.pl Barney
I don't see a Fred
$ perl5.10.0 switch.pl Alfred
Name has fred in it
```

你可能会说，“不稀奇，我可以用 if-elsif-else 来写这个例子”。接下来的例子正是用这种方式写的，而且还用 my 来声明当前语句块的私有 $_ 变量。这可是 Perl 5.10 里面 my 的新用法。

```
use 5.010;

{
my $_ = $ARGV[0]; # lexical $_ as of 5.10!

   if( $_ ~~ /fred/i ) { say 'Name has fred in it' }
elsif( $_ ~~ /^Fred/ ) { say 'Name starts with Fred' }
elsif( $_ ~~ 'Fred'  ) { say 'Name is Fred' }
else                   { say "I don't see a Fred" }
}
```

如果 given 能做的和 if-elsif-else 完全一样的话，它就没必要存在了。和 if-elsif-else 不同，given-when 可以在满足某个条件的基础上，继续测试其他条件。而 if-elsif-else 一旦满足了某个条件，就只能执行对应的那个语句块。

注 1：按 Perl 的术语来称呼，given 是一个主题符，因为它能使后面的参数成为主题，而主题也正是 Perl 6 当中给 $_ 起的（时尚）新名字。

在继续深入之前，还是来看看显式写法的例子，搞清楚究竟发生了些什么。除非明确指定，否则在 when 的语句块后面，好像都有一句 break，它告诉 Perl 现在就跳出 given-when 结构，继而执行后面的程序。之前的例子真好像带有 break 似的，尽管不需要你自己输入：

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it'; break }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred'; break }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred'; break }
    default               { say "I don't see a Fred"; break }
    }
```

这种写法用在以上的例子中并不太合适。因为我们是按从宽泛到特殊的顺序来测试的。如果传来的参数匹配 /fred/i，Perl 就不会测试其余的 when 条件了。同样的道理，我们不能一开始就测试是否包含 Fred，因为这样就总没有机会做后面的测试，执行到第一个 when 语句块后，Perl 就会跳出该控制结构。

如果在 when 语句块的末尾使用 continue，Perl 就会尝试执行后续的 when 语句了。这就是 if-elsif-else 力不能及的地方。当另一个 when 的条件满足时，会执行对应语句块（同样，默认退出整个控制结构，除非你写清楚）。在每个 when 语句块的末尾写上 continue，就意味着所有的条件判断都会执行：

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it'; continue }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred'; continue } # 注意!
    default               { say "I don't see a Fred" }
    }
```

这里还有一个小问题。我们注意到 default 总是会运行：

```
$ perl5.10.0 switch.pl Alfred
Name has fred in it
I don't see a Fred
```

default 块相当于一个测试条件永远为真的 when 语句。如果在 default 之前的 when 语句使用了 continue，Perl 就会继续执行 default 语句。所以本质上，default 就是另一个 when：

```
use 5.010;

given( $ARGV[0] ) {
```

```
    when( $_ ~~ /fred/i ) { say 'Name has fred in it'; continue }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred'; continue } # 注意!
    when( 1 == 1        ) { say "I don't see a Fred" } # 相当于 default 语句块
    }
```

要解决这个问题，只要拿掉最后一个 when 的 continue 就可以了：

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it'; continue }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred'; break } # 现在对了!
    when( 1 == 1        ) { say "I don't see a Fred" }
    }
```

现在我们已经都交代清楚了，下面再用常规写法改写，以后你也可以将这种形式用到实际的程序中：

```
use 5.010;

given( $ARGV[0] ) {
    when( /fred/i ) { say 'Name has fred in it'; continue }
    when( /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( 'Fred'  ) { say 'Name is Fred'; }
    default         { say "I don't see a Fred" }
    }
```

笨拙匹配

除了在 given-when 中使用智能匹配外，还可以像前面那样使用所谓笨拙（dumb）匹配。当然，这不是说真的笨拙，我们指的就是你已经掌握的正则匹配方式。Perl 只要看到明确书写的（任何类型的）比较操作符或是（正则表达式的）绑定操作符，它就会按这些操作符的要求去做：

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ =~ /fred/i ) { say 'Name has fred in it'; continue }
    when( $_ =~ /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( $_ eq 'Fred'  ) { say 'Name is Fred' }
    default               { say "I don't see a Fred" }
    }
```

你甚至可以混用笨拙匹配和智能匹配，每个 when 表达式都会各自处理不同类型的匹配方式：

```
use 5.010;

given( $ARGV[0] ) {
    when( /fred/i )        { # 智能匹配
        say 'Name has fred in it'; continue }
    when( $_ =~ /^Fred/ ) { # 笨拙匹配
        say 'Name starts with Fred'; continue }
    when( 'Fred'  )        { # 智能匹配
        say 'Name is Fred' }
    default                { say "I don't see a Fred" }
    }
```

注意笨拙和智能这两种匹配方式并非泾渭分明，因为正则表达式的匹配操作符默认也是用 $_ 来测试的。

智能匹配操作符用于判断是否相同（或者差不多是相同），所以在需要比较大小时，就不能用智能匹配了。还是直接用传统的比较操作符好了：

```
use 5.010;

given( $ARGV[0] ) {
    when( /^-?\d+\.\d+$/ ) { # 智能匹配
        say 'Not a number!' }
    when( $_ > 10 )        { # 笨拙匹配
        say 'Number is greater than 10'  }
    when( $_ < 10 )        { # 笨拙匹配
        say 'Number is less than 10' }
    default                { say 'Number is 10' }
    }
```

某些特定情况下，Perl 会自动使用笨拙匹配模式。若在 when 里调用某个函数【注 2】，Perl 会根据返回值的真假进行判断：

```
use 5.010;

given( $ARGV[0] ) {
    when( name_has_fred( $_ ) ) { # 笨拙匹配
        say 'Name has fred in it'; continue }
    }
```

同样的规则还适用于 Perl 的内建函数，比如 defined、exists 和 eof，因为这些函数本来就是要返回真假值的。

取反的表达式，包括取反的正则表达式，都不会使用智能匹配模式。下面的例子就和你在上一章看到的类似：

注 2：Perl 对方法调用也不会使用智能匹配，不过面向对象的内容应该在《Intermediate Perl》(O'Reilly) 中再介绍。

```
use 5.010;

given( $ARGV[0] ) {
    when( ! $boolean )  { # 笨拙匹配
        say 'Name has fred in it' }
    when( ! /fred/i   ) { # 笨拙匹配
        say 'Does not match Fred' }
    }
```

多个项目的 when 匹配

有时候需要遍历许多元素，可 given 只能一次接受一个参数。当然你可以将 given 放到 foreach 里面循环测试。比如，要遍历 @names，依次将各元素赋值到 $name，然后再用 given：

```
use 5.010;

foreach my $name ( @names ) {
    given( $name ) {
        ...
        }
    }
```

猜我想说啥？没错，这么写太罗嗦了。难道你还没厌倦这种额外多出来的工作吗？这回，给 @names 的当前元素起个化名，让 given 直接用这个化名好了。Perl 就该聪明得多！别担心，事实就是如此。

要遍历多个元素，就别用 given。使用 foreach 的简写方式，让它给当前正在遍历的元素起个化名 $_。此外，若要用智能匹配，当前元素就只能是 $_。

```
use 5.010;

foreach ( @names ) { # 不要使用具名变量!
    when( /fred/i ) { say 'Name has fred in it'; continue }
    when( /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( 'Fred'  ) { say 'Name is Fred'; }
    default         { say "I don't see a Fred" }
    }
```

一般遍历的时候，总希望可以看到当前的工作状态。可以在 foreach 语句块中写上其他语句，比如 say：

```
use 5.010;

foreach ( @names ) { # 不要使用具名变量!
    say "\nProcessing $_";
```

```
    when( /fred/i ) { say 'Name has fred in it'; continue }
    when( /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( 'Fred'  ) { say 'Name is Fred'; }
    default         { say "I don't see a Fred" }
    }
```

你甚至还可以在若干 when 语句之间写上其他语句，比方说，在 default 之前加上一条输出调试信息的语句（在 given 结构里也可以这么用哦）。

```
use 5.010;

foreach ( @names ) { # 不要使用具名变量!
    say "\nProcessing $_";

    when( /fred/i ) { say 'Name has fred in it'; continue }
    when( /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( 'Fred'  ) { say 'Name is Fred'; }
    say "Moving on to default...";
    default         { say "I don't see a Fred" }
    }
```

习题

以下习题答案参见附录 A：

1. [15] 重写第十章中的第一道习题，也就是猜数字的那题，使用 given 结构来实现。想想看该如何处理非数字的输入？这里不需要用智能匹配。

2. [15] 用 given-when 结构写一个程序，根据输入的数字，如果能被 3 整除，就打印“Fizz”，如果能被 5 整除，就打印“Bin”，如果能被 7 整除就打印“Sausage”。比如输入数字 15，程序该打印“Fizz”和“Bin”，因为 15 可以同时被 3 和 5 整除。思考一下，可以让程序输出“Fizz Bin Sausage”的最小数字该是多少？

3. [15] 用 for-when 写个程序，要求从命令行遍历某个目录下的文件列表，并报告每个文件的可读、可写和可执行属性状态。不需要使用智能匹配。

4. [20] 使用 given 和智能匹配写个程序，从命令行得到一个数字，打印出这个数字除了 1 和它本身以外的因数。例如输入 99，你写的程序该报告 3，9，11 和 33 这 4 个数。如果输入的数字就是一个质数，程序要报告说明这是质数。如果输入的不是数字，也应该报告说明输入有误，不会继续计算。虽然可以用 if 结构和笨拙匹配来实现，不过作为练习，这里请只用智能匹配来实现。

   作为开始，下面给出一段返回质数的函数供你使用。它会尝试从 2 到输入数字 $number 的一半范围内的数字，看能被哪些整除：

```
sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. ($number/2 ) ) {
        push @divisors, $divisor unless $_ % $divisor;
        }

    return @divisors;
    }
```

5. [20] 修改上题程序，报告输入数字的奇偶情况、是否是质数（除 1 和本身外不能被整除）、是否可以被某个你最喜欢的数字整除。还是只能用智能匹配实现。

第十六章

进程管理

身为程序员最高兴的就是能运行别人的程序，省去自己编写的麻烦。现在，我们来看看如何用 Perl 照管新生的孩子【注 1】。

在 Perl 里有句话是“办法不止一种”，这里也是如此。这些办法可能有许多重叠或差异之处，并且各有特色。如果你不喜欢头一种方法，大可往下读个一两页，找到比较合胃口的方式。

Perl 的可移植性非常高。本书的其他章节大都不需要用脚注来说明某个程序在 Unix 上是这样、在 Windows 上是那样，而在 VMS 又是另外一种情况。但是当你想在自己的计算机上运行别人的程序时，请注意：在 Macintosh 上能找到的程序，与 Cray 上的多半大不相同。本章的例子将以 Unix 环境为主；如果你使用的不是 Unix 系统，难免会碰到一些差异。

system 函数

在 Perl 中，启动子进程最简单的方法是用 `system` 函数。例如要从 Perl 调用 Unix 的 *date* 命令，需要如下代码：

```
system "date";
```

这会创建一个子进程来运行 *date* 命令，并且它继承了 Perl 的标准输入、标准输出和标准出错。这意味着 *date* 通常输出的日期时间字符串会送到目前 Perl 的 `STDOUT` 指向的地方。

注 1：　实际上是新生的子进程。

提供给 system 函数的参数就是那些在 shell 中常常键入的命令。所以当你想用 *ls -l $HOME* 之类比较复杂的命令时，只要把它全部放进参数里就行了：

```
system 'ls -l $HOME';
```

请注意，因为这里的 $HOME 是 shell 的变量，所以用的不是双引号，而是单引号。否则，因为美元符号也是 Perl 进行变量内插的符号，所以 shell 不可能看到它。当然，我们也可以这么写：

```
system "ls -l \$HOME";
```

但是这样写很快就会乱成一团。

目前 *date* 命令只是输出结果而已。假设它成了比较健谈的命令，每次都会问“你想知道哪个时区的时间？”【注2】这个信息会先出现在标准输出上，接着程序从标准输入（继承自 Perl 的 STDIN）等待回应。你会看到这个问题，然后键入答案（例如 Zimbabwe time），然后 *date* 才能完成它的工作。

子进程正在运行时，Perl 会很有耐心地等它结束。如果 *date* 命令耗时 37 秒，Perl 就会暂停 37 秒。不过，你还是可以利用 shell 的功能来启动后台进程【注 3】：

```
system "long_running_command with parameters &";
```

现在会启动 shell，而它会注意到命令行结尾的“与号 &”。于是 shell 让 *long_running_command* 成为后台进程并立即退出。接着，Perl 会注意到 shell 已经返回了，现在可以做别的事情了。在这个例子中，*long_running_command* 其实是 Perl 的孙进程，只是 Perl 无法直接控制（或访问）它。

当命令“很简单”的时候，不会用到 shell。在之前运行 *date* 与 *ls* 命令时，Perl 会寻找待执行的命令，必要时会使用继承而来的 PATH 变量【注 4】。找到之后，就直接启动它。但如果字符串里出现奇怪的字符，例如 shell 元字符：美元符号、分号、竖线等，

注 2：据我所知，目前还没有人设计出这种工作方式的 date 命令。

注 3：我们这里介绍的依赖于你的系统。Unix shell（*/bin/sh*）允许你在运行这类命令时使用“与号&”来启动一个后台进程。当然如果你在非 Unix 系统上工作，它们往往不能这样启动后台进程，那样自然也无法这么做。

注 4：PATH 可以随时通过 $ENV{'PATH'} 改变。初始时，这是从父进程（通常是 shell）继承的环境变量。改变这个值会影响新的子进程，但不能影响之前的父进程。PATH 是搜索可执行程序（命令）的目录列表，甚至在非 Unix 系统之中也是如此。

那么会调用标准 Bourne Shell（*/bin/sh*【注 5】）来处理。这样一来，shell 是子进程，要执行的命令则是孙进程（或是更下一代）。例如，你可以把如下简短的 shell 脚本放进参数里：

```
system 'for i in *; do echo == $i ==; cat $i; done';
```

这里我们再次使用了单引号，因为其中的美元符号是给 shell 用的，而不应该由 Perl 解释。双引号会让 Perl 内插 `$i` 的值，而不是让 shell 处理【注 6】。附带一提，这个简短的 shell 脚本会将当前目录中每个文件的文件名及其内容显示出来。如果你不相信我们的话，不妨自己试试。

避免使用 Shell

system 函数也可以用一个以上的参数调用【注 7】，如此一来，不管你给的文本有多复杂，都不会用到 shell。做法如下：

```
my $tarfile = "something*wicked.tar";
my @dirs = qw(fred|flintstone <barney&rubble> betty );
system "tar", "cvf", $tarfile, @dirs;
```

在这个例子里，第一个参数 `"tar"` 是命令名称，在 shell 里可以通过 `PATH` 环境变量辅助定位。接下来，后面的参数会被逐项传递给前面的命令。即使参数里出现对 shell 有意义的字符，例如 `$tarfile` 存储的文件名中的星号，或者 `@dirs` 存储的路径中的管道符号，大小于号以及与号，都不会被 shell 误解为特殊含义。所以 `tar` 命令会刚好得到五个参数，一个选项，一个打包后的文件名称，三个要打包的目录。若是使用下面的写法：

```
system "tar cvf $tarfile @dirs";  # 乱了!
```

就会让 shell 理解为，把 `fred` 目录打包为一个文件，打包过程输出的信息再通过管道送至 *flintstone* 命令处理，但该命令应该会从 `barney` 文件读取标准输入，所以会忽略

注 5：或者在你编译 Perl 的时候检查到的 shell。实际情况下，Unix 类的系统中几乎总是 */bin/sh*。

注 6：当然如果你设置 `$i = '$i'`，就能确保正常工作。但没准哪天会有一个维护程序员“好心”帮你删掉那行！

注 7：或者按间接对象写法，第一个参数后面不用逗号，例如 `system {'fred'} 'barney';`。它实际上会运行程序 `fred`，但却让 `fred` 误以为自己是 `barney`。查看 *perlfunc* 在线手册了解更多信息。

管道过来的数据，整个过程全部转到后台执行。最后运行 rubble 命令，将输出保存到 betty 文件。这显然不是程序作者的原意，因为 shell 扩展了这些目录名称中的特殊字符，会造成很多意想不到的结果。

这真是有点吓人【注 8】，尤其是当数据来自 Web 表单这样的用户输入的时候。所以也许你真的应该早些下决心，开始用 system 的多参数版本来启动子进程。当然也会同时失去 shell 提供的 I/O 重定向、后台进程等等功能，天下没有免费的午餐。

另外请注意，system 的单参数调用也可以和多参数版本非常接近：

```
system $command_line;
system "/bin/sh", "-c", $command_line;
```

不过没人会用后面的写法，除非你想使用不同的 shell 来处理事情，例如 C shell：

```
system "/bin/csh", "-fc", $command_line;
```

即使如此，这种情形还是很少见。因为即使“通用” shell【注 9】也有足够的灵活性，尤其是在脚本处理方面。

操作符的返回值是根据子进程的结束状态来决定的【注 10】。在 Unix 里，退出值 0 代表正常，非 0 值则代表有问题：

```
unless (system "date") {
  # 返回 0 的话就代表执行成功
  print "We gave you a date, OK!\n";
}
```

注意绝大多数操作符遵从“真为正常，假为异常”的准则，而这里恰恰相反，所以若要套用 do this or die 的写法，我们就得先颠倒真假。最简单的做法就是在 system 操作符之前加上逻辑反，也就是感叹号：

```
!system "rm -rf files_to_delete" or die "something went wrong";
```

注 8：除非你已经开始 taint 检查，也就是对用户那里来的数据进行筛查，否则难免被某个用户戏弄。

注 9：也就是 */bin/sh* 或者 Unix 系统中已经安装的最接近 Bourne shell 的那个。如果你没有安装“通用” shell 的话，Perl 会尝试随便调用一种命令解释器，因此你得后果自负。请参考有关 Perl 移植的文档了解详细信息。

注 10：其实是所谓等待状态，就是用子进程退出码乘以 256，若内核转储加上 128，再加上可能导致退出的信号。但是我们发现没有必要进行解码检查，因为大多数时候真假检查就够了。

注意在这个例子里面，若要显示错误信息不能参考 $! 变量。因为错误多半发生在 *rm* 命令的运行时刻，不是 $! 能捕捉的系统调用相关错误。

exec 函数

到目前为止，我们提到的 system 函数的所有语法也都适用于 exec 函数。当然有一个重要的例外，system 函数会创建子进程，子进程会在 Perl 睡眠期间执行任务。而 exec 函数却导致 Perl 进程自己去执行任务。这类似于子程序调用与 goto 的差别。

例如，若要运行 */tmp* 目录下的 *bedrock* 程序，带上 *-o args1* 以及原程序的参数。就可以这样写：

```
chdir "/tmp" or die "Cannot chdir /tmp: $!";
exec "bedrock", "-o", "args1", @ARGV;
```

当运行到 exec 操作符的时候，Perl 找到 *bedrock* 程序并且“跳进去”执行。这样，就不再有 Perl 进程了【注 11】，只有那个 *bedrock* 进程。这样在 *bedrock* 执行结束时候，没有 Perl 进程在等待。只能返回到启动这个程序的命令行提示符。

为何要这样做呢？其实这个 Perl 程序的主要功能是为另一个程序执行设定运行的环境，因此当其他程序执行时，它已经完成了自己的使命。如果我们使用 system 而不是 exec，那就有些浪费，导致 Perl 痴痴地等着另一个程序运行完毕，然后才能跟着结束。

虽说如此，但实际上我们很少会用到 exec，一般都是同 fork 一起使用，这稍后会介绍。因此如果吃不准到底要用 system 还是 exec，就总是用 system 好了，大多数情况下都是稳妥的。

一旦启动要执行的程序，Perl 便放手退出，因此在 exec 调用之后写的任何代码都无法运行，除非是编程接管启动过程中的错误：

```
exec "date";
die "date couldn't run: $!";
```

事实上，如果打开自动警告，并且在 exec 后面写了除 die 以外的代码【注 12】，就会收到警告。

注 11： 其实还是那个老的进程，通过执行 Unix 的 exec(2) 或类似的系统调用而切换。这样进程号是不变的。

注 12： 或者 exit。除非在块结尾调用 exec。在新版本的 Perl 中这可能会改变。

环境变量

在使用这里讨论的任何方法启动其他进程的时候，可能会需要设置程序的环境。前面谈到我们可以在一个特定的工作目录下启动进程，然后它会从父进程继承这个目录。设置好的环境变量也是如此继承的。

最广为人知的环境变量是 PATH。若没有听说过，多半你在使用不支持环境变量的系统。在 Unix 类的系统上，PATH 是以冒号分隔的目录列表，其元素是可执行文件的搜索路径。当你键入 *rm fred* 这样的命令时，系统会在目录列表中依次寻找 *rm* 命令。Perl（或系统）会用 PATH 来搜索可执行程序，启动之后子程序若需要调用其他程序，也会使用 PATH 来搜索。当然如果指定了命令的全路径名（像 */bin/echo*），就没必要在 PATH 里搜索了。但这样写对大多数人来说太不方便了。

在 Perl 中，环境变量对应于特殊的 %ENV 哈希，其中每个键都代表一个环境变量。在程序开始运行时，%ENV 会保留从父进程（通常为 shell）继承而来的设定值。修改此哈希就能改变环境变量，也能被 Perl 调用的子进程继承。假如现在需要调用系统的 *make* 程序（进而调用其他程序），并且想以私有目录作为寻找命令（包括 *make* 自己）的首选位置，此外还要禁用（*make* 和其他程序敏感的）IFS 环境变量，就可以这么写：

```
$ENV{'PATH'} = "/home/rootbeer/bin:$ENV{'PATH'}";
delete $ENV{'IFS'};
my $make_result = system "make";
```

新创建的进程会继承父进程的环境变量，包括当前工作目录、标准输入输出、标准错误和另外一些“小秘密”。可以参考系统与程序设计相关的文档了解更多细节。但是请记住：修改从父进程继承的变量并不能影响 shell 或者其他父进程。

用反引号捕获输出结果

无论用 system 还是 exec 调用，被调用程序的输出都会定向到 Perl 的标准输出。有时候我们的兴趣就在于捕获输出的字符串，并进一步处理。这其实只要以魔力反引号来代替单引号或双引号：

```
my $now = `date`;               # 捕获 date 的输出
print "The time is now $now"; # 这里不需要换行符，因为 date 的输出里已经包含
```

一般来说，*date* 命令能输出长度约为 30 个字符的行，其中含有今天的日期与时间，以及一个换行符。当我们把 *date* 放在反引号里时，Perl 会执行这个命令并将其标准输出结果以字符串形式捕获。在这个例子中，字符串会赋给 $now 变量。

这就像 Unix shell 的反引号一样，但是 shell 还会做额外的处理：它会将最后一个换行符移除，这样便于连接其他有用的东西。Perl 总是很诚实，它会直接使用接收到的真实输出。要在 Perl 中取得相同的结果，我们可以对取得的字符串进行一次 chomp 运算：

```
chomp(my $no_newline_now = `date`);
print "A moment ago, it was $no_newline_now, I think.\n";
```

Perl 解释反引号里面的值的方式类似于 system 的单参数形式【注 13】，并且在解释器中会以双引号字符串形式展开，这意味着反斜线转义与变量内插都会正常处理【注14】。例如要取得一系列 Perl 函数的说明文档，可以重复执行 *perldoc* 命令，每次使用不同的参数：

```
my @functions = qw{ int rand sleep length hex eof not exit sqrt umask };
my %about;

foreach (@functions) {
  $about{$_} = `perldoc -t -f $_`;
}
```

请注意，每次循环执行时 $_ 的值都会不同，这让我们可以每次执行不同的命令并取得它的输出。另外，如果你还不熟悉这些函数，不妨趁此机会看一下说明文档，了解其使用的细节。

反引号写法要模拟单引号很麻烦【注 15】，因为变量内插和反斜线转义总是会起效。另外也没有简单的方法可以模拟 system 的多参数版本，也就是避免 shell 扩展。如果反引号内的命令很复杂，Unix 的 Bourne Shell（或是系统设定的其他 shell）就会自动被用来解释该命令。

下面的反面例子用来说明什么是滥用。换句话说，不需要捕获输出的时候，最好不要使用反引号【注 16】。例如：

注 13：就是说，也是以“通用” shell (*/bin/sh*) 或相近的脚本解释器来解释的，就像 system 那样。

注 14：所以若要向 shell 发送反斜线符号，就得连写两个反斜线。因为在 Windows 环境中常常需要两个连续的反斜线，就要连写 4 个。

注 15：有些高难度方法，如将字符串写在 qx'...' 里面，或是用单引号将所有的字符串放进某个变量中，然后把那个变量内插在反引用串之中，因为 Perl 只支持单次内插。

注 16：这种情况可以称为空上下文。

```
print "Starting the frobnitzigator:\n";
`frobnitz -enable`; # 请别这么做!
print "Done!\n";
```

这里的问题是 Perl 会费力捕获命令的输出，然后被直接丢弃。而且这样一来，你也失去了以 system 的多参数形式来精确控制传入参数的能力。所以，在安全与效率的双重考虑之下，请改用 system 吧。

以反引号执行的命令，会继承 Perl 当前的标准错误流。如果该命令将错误信息送到标准错误，就可能会显示在终端上，从而导致用户困惑，因为他并未运行 *frobnitz* 程序。如果你想要一并捕获标准输出和标准错误，就可以使用 shell 规范将标准错误合并至标准输出。也就是 Unix shell 的俗语 2>&1:

```
my $output_with_errors = `frobnitz -enable 2>&1`;
```

注意这会让标准错误与标准输出的信息交织在一起，就像在终端上看到的那样，当然可能因为缓冲的原因有顺序上的细微差别。如果你需要分别捕获标准输出和标准错误，就得考虑使用更加麻烦的解决方案【注 17】。类似的，被执行的命令也会继承 Perl 当前使用的标准输入。我们日常执行的命令大都不会使用标准输入，所以没有这方面的问题。但是，如果 *date* 命令询问你要使用的时区（正如我们之前假设的），这样就会有问题，因为提示文字 which time zone? 会被送至标准输出，成为捕获内容的一部分，然后 *date* 会试着从标准输入读进数据。由于用户根本看不到提示文字，所以他不知道该输入数据！没多久，用户就会打电话给你，说你的程序卡住了。

因此，请勿使用会读取标准输入的命令。如果你不太确定它是否会从标准输入读取数据，请将标准输入重定向为从 */dev/null* 读取数据，如下所示：

```
my $result = `some_questionable_command arg arg argh </dev/null`;
```

这样一来，被调用的 shell 就会将输入重定向到 */dev/null*，接着再执行那个交互式的命令。这样就算它要求输入，也只会读到文件结束符。

在列表上下文中使用反引号

如果命令会输出很多行，在标量上下文中使用反引号会得到一个很长的文本串，其中包含换行符。然而在列表上下文中同样的反引号调用则返回输出的文本行列表。

注 17：比如使用标准 Perl 发行的模块 IPC::Open3，或者自己编程处理 fork 相关的事宜，稍后会有展示。

例如 Unix 下的 *who* 命令通常会用多行列出当前登录系统的每个用户：

```
merlyn     tty/42      Dec 7  19:41
rootbeer   console     Dec 2  14:15
rootbeer   tty/12      Dec 6  23:00
```

最左边的列是用户名称，中间的列是 tty 名（也就是登录位置），其余的列则是登录的日期与时间（也许还有远程的登录信息，但是本例没有）。在标量上下文中，我们在一个变量中得到所有输出，所以必须自行拆开：

```
my $who_text = `who`;
```

但是在列表上下文中，则会自动取得拆成多行的数据：

```
my @who_lines = `who`;
```

现在 @who_lines 里会有多个拆分好的以换行结尾的字符串。对这个结果调用 chomp 就可以删除所有元素末尾的换行符。不过不如换个思路，只要用 foreach 就可以逐行处理，循环中默认使用 $_ 作为控制变量：

```
foreach (`who`) {
  my($user, $tty, $date) = /(\S+)\s+(\S+)\s+(.*)/;
  $ttys{$user} .= "$tty at $date\n";
}
```

这里用三次循环覆盖以上的数据，你的系统可能有更多人登录，因此要循环更多次。注意这里还用了正则表达式进行匹配，但并没有明确使用绑定操作符（=~），而是直接针对 $_ 进行匹配。这样写很好，因为数据就在那里。

另外请注意，这个正则表达式会寻找一个非空白的单词、数个空白、一个非空白的单词、数个空白，接着是剩余的所有单词，但是不包括换行符号（因为点号通常不匹配换行符）【注 18】。这个模式经过精心设计，能匹配 $_ 代表的数据。例如处理第一行的时候，$1 会是 merlyn，$2 会是 tty/42，$3 则是 Dec 7 19:41，这是成功的捕获。

因为这个正则表达式是在列表上下文中进行运算的，所以如第八章进行匹配所述，它不会像标量上下文中那样返回真假值，而是将被捕获的变量放进列表中。因此，$user 最后会得到 merlyn 这个值，其他变量则依此类推。

循环中的第二条语句只是用来存储 tty 与日期信息，之所以对（可能是 undef 的）哈希值进行追加是考虑到同一个用户可能多次登录（例如 rootbeer）。

注 18：点号默认不匹配换行符的原因你现在看到了。它可以让我们轻松编写以上的模式，不必担心最后的换行。

将进程视为文件句柄

到目前为止，我们看到的方法都是由 Perl 同步控制子进程：启动一个命令，然后等着它结束，或许还会获取其输出。但是 Perl 其实也可以启动一个异步运行的子进程，并和它保持通信【注 19】，直到子进程结束执行为止。

要启动并发运行的子进程，请将命令放在 open 调用的文件名部分，并且在它前面或后面加上竖线，也就是管道符号。也有人将这种调用方式叫做*打开管道*。

```
open DATE, "date|" or die "cannot pipe from date: $!";
open MAIL, "|mail merlyn" or die "cannot pipe to mail: $!";
```

第一个例子里的竖线在命令的右边，表示该命令执行时它的标准输出会转向只读的文件句柄 DATE，就像在 shell 里执行 *date | your_program* 这个命令一样。在第二个例子里，竖线在左边，所以该命令的标准输入会连接到只写的文件句柄 MAIL，就像在 shell 里运行 *your_mail | mail merlyn* 这个命令一样。不论竖线在左在右，都会启动一个独立于 Perl 的进程【注 20】。如果无法创建子进程，open 就会失败。如果命令不存在或没有发生错误而正常结束，这在打开时通常不会有错误发生，但是在关闭时却会报错。稍后我们就会遇到这样的状况。

无论是从哪种角度来说，打开之后的程序无法分辨，也不用关心，到底文件句柄后面是一个进程还是真的文件。对于以读取模式打开的文件句柄，你只要这样读就好了：

```
my $now = <DATE>;
```

想传递数据给 mail 进程。让它从标准输入等待读取邮件正文，以便发送给 *merlyn*。只要打印到文件句柄就行了：

```
print MAIL "The time is now $now"; # 假设 $now 以换行符结尾
```

总之可以假设这些文件句柄都连接了魔力文件，一个包含了 *date* 命令的输出，另一个可以自动用 *mail* 命令发送文件。

如果进程连接到某个以读取模式打开的文件句柄，然后它结束运行了，则文件句柄会返回文件结尾，就像读完了正常的文件一样。当你关闭用来写入数据到某进程的文件句柄时，该进程会读到文件结尾。所以，要提交邮件并发送，只要关闭这个文件句柄即可：

注 19：使用管道或者其他操作系统提供的进程间通信机制。

注 20：如果 Perl 进程比命令早结束，等待中的读调用会得到文件结尾，而下次写调用会收到 broken pipe 错误。

```
close MAIL;
die "mail: non-zero exit of $?" if $?;
```

关闭连接至进程的文件句柄，会让 Perl 等待该进程结束以取得它的结束状态。结束状态会存入 `$?` 变量（联想到 Bourne Shell 里的同名变量了吗？），它的值就与 `system` 函数返回的数值一样：零表示成功，非零值代表失败。每个结束的进程都会覆盖掉前一个返回值，所以，如果你需要这个值，请尽快保存。（如果你好奇的话，`$?` 变量也会存储前一次 `system` 或用反引号圈引的命令的返回值。）

这些进程间的协同方式，就像 shell 中被管道连接的命令一样。如果你试着想要读取数据，但是没有任何数据送达，程序就会暂停，但不会额外消耗 CPU 时间，直到送出数据的进程有数据发送为止。同样，如果生产数据的程序超出读取程序的速度，它就会减速运行，直到读取数据的程序赶上为止。进程之间会有缓冲（一般是 8KB 大小），这样可以避免相互锁定。

为什么要用文件句柄的方式来和进程打交道呢？假如要根据计算的结果来决定写到其他进程的数据，这是唯一简单的做法。可是如果只想读取，反引号通常更易于使用。然而如果子进程不时有数据要送给父进程，就必须用管道了。

例如 Unix 的 *find* 命令可以依照文件的属性来寻找文件的位置。然而，如果文件很多，这通常会耗费不少时间，尤其是可能要从根目录开始找时。虽然可以将 *find* 命令放在反引号内，但你也可以在每找到一个文件时就立即取得它的名称。这通常是比较好的做法：

```
open F, "find / -atime +90 -size +1000 -print|" or die "fork: $!";
while (<F>) {
  chomp;
  printf "%s size %dK last accessed on %s\n",
    $_, (1023 + -s $_)/1024, -A $_;
}
```

find 命令这次运行是要查找那些 90 天内未被存取过的 1000 块以上的大文件，它们非常适合被归档到永久性介质中。在 *find* 工作的时候，Perl 会等待。每找到一个文件，Perl 会立刻收到文件名并进一步显示文件的相关信息供分析。如果我们用反引号调用它的话，就得等到 *find* 彻底搜完才能看到第一行输出。从任务监控角度来说，往往看到执行的最新进展才能让人放心。

用 fork 开展地下工作

除了上述的高级接口，针对 Unix 以及其他系统的进程管理，Perl 几乎都能直接执行系统调用。如果你从来没有这样做过【注 21】，不妨略过本节。虽然这章的篇幅不足以详述全部的细节，我们还是可以大致展示这个技术：

```
system "date";
```

让我们看看，同样的事情用底层系统调用该怎么做：

```
defined(my $pid = fork) or die "Cannot fork: $!";
unless ($pid) {
  # 能运行到这里的是子进程
  exec "date";
  die "cannot exec date: $!";
}
# 能运行到这里的是父进程
waitpid($pid, 0);
```

这里检查了 fork 的返回值，它在失败时会返回 undef。如果成功了，则下一行开始就会有两个不同的进程在运行。因为只有父进程的 $pid 不是零，所以只有子进程会执行 exec 函数。父进程会略过该部分，直接执行 waitpid 函数，也就是在那里等待特定的子进程结束（在这期间，若是其他的子进程结束执行，则会被忽略掉）。如果这些听起来像天书的话，那也没关系，继续使用 system 函数。你不会被朋友取笑的。

在付出额外的代价之后，程序员也获得了最大的控制权，可以创建任意管道、对文件句柄进行处理，也可以进一步了解子进程号和父进程号。但是对于这一章来说还是太复杂了，想要走的更远的话，请直接参阅 *perlipc* 在线手册，以及任何一本切实谈论系统编程的书。

发送及接收信号

Unix 信号是发往一个进程的脉冲。信号不会说得很详细，它就像汽车的喇叭声一样：汽笛对你而言，可能代表“小心！桥断了”、“绿灯啦！快走”、“快停下！车顶上有个小孩”，或是“你好，小鸟”。好在，Unix 信号比这些稍微强一些，因为针对不同的情况会有不同的信号【注 22】。信号会用不同的名称相互区分，像 SIGINT 就代表中断信号，另

注 21：或者你在运行一个不支持 fork 的系统。但是 Perl 开发人员正尝试实现进程 fork 的功能，即使这种系统和 Unix 的进程模型差距非常大。

注 22：当然与以上这些状况并非完全相似，不过确实有相近的 Unix 信号。比如信号 SIGHUP、SIGCONT、SIGINT 以及虚拟的零信号 SIGZERO。

外还有一个相关的整数可用来识别。它的取值范围从 1 到 16、1 到 32 或是 1 到 63，依系统而定。通常某个重大事件发生时就会发出信号，例如在终端上按下 Control-C 这样的中断组合键，就会给与此终端相连的所有进程发送 SIGINT 信号【注 23】。信号可能是由系统自动发送的，但也可能来自别的进程。

可以从 Perl 进程发送信号给别的进程，但是得先知道目标进程的编号。要说明如何取得进程编号可能有点复杂【注24】，不过如果已经知道要发送 SIGINT 信号给进程 4201，那么做法就简单明了：

```
kill 2, 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

发送信号的命令取名为 kill，因为发明信号的主要目的之一就是中止运行了太久的进程。因为 2 号信号就是 SIGINT，所以这里也可以改成字符串 INT。如果该进程早已退出【注 25】，就会收到返回的错误，因此可以用这个技巧来判断进程是否仍然存活。编号为零的特殊信号用来测试能否发送信号，而不至于真的发送。因此，探测进程存活与否的程序可以写成这样：

```
unless (kill 0, $pid) {
  warn "$pid has gone away!";
}
```

接收信号好像比发送信号要有趣些。你为什么会想要这么做呢？假设你有一个程序会在 */tmp* 目录里创建文件。正常情况下，程序结束前就会删除这些文件。如果有人在程序运行时按下 Control-C，结果就会在 */tmp* 目录里留下垃圾，而这是很不礼貌的事情。要解决这个问题，就得创建一个负责清理的信号处理程序：

```
my $temp_directory = "/tmp/myprog.$$"; # 在这个目录下创建文件
mkdir $temp_directory, 0700 or die "Cannot create $temp_directory: $!";

sub clean_up {
  unlink glob "$temp_directory/*";
  rmdir $temp_directory;
}
```

注 23：你可能以为按下 Control-C 就会中止程序，其实这样做只是送出一个 SIGINT 信号，而这个信号恰巧能中止程序而已。稍后就能看到如何在收到 SIGINT 信号时做些其他的事，而不是单单中止程序。

注 24：进程标识符要么是自己 fork 时产生的，要么是从某个文件中或是从外部程序中取得。从外部程序中取得进程号更困难，而且更容易出错，因此那些要运行很久的程序往往把自己的进程编号写在一个文件里面，并且在文档中对此文件明确说明。

注 25：如果你不是超级用户的话，想给别人的进程发信号就会失败。另外给别人的进程发 SIGINT 也总是很失礼的举动。

```
sub my_int_handler {
  &clean_up;
  die "interrupted, exiting...\n";
}

$SIG{'INT'} = 'my_int_handler';
.
.    # 时间流逝，程序在运行着，在临时的目录中创建一些
.    # 临时的文件，然后可能有人按了 Control-C
.
# 这里是正常运行的结尾部分
&clean_up;
```

对特殊哈希 %SIG 赋值就能设置信号处理程序。哈希键是信号名称，注意不用写固定前缀 SIG。哈希值是【注 26】子程序名，注意不需要“与号”。现在只要收到 SIGINT 信号，Perl 就会暂停手上的事务并立刻执行信号处理子程序。这里子程序会清理临时文件并退出。当然即使没有按 Control-C，也还是会在程序末尾调用 &clean_up。

假如信号处理子程序没有结束程序而是直接返回，那么程序会从先前暂停的地方继续执行。如果该信号只是要暂停并处理某些事情，而不是停止整个程序的话，这种时候就该用返回而非退出。举例来说，假设处理文件里的每行都慢到要花几秒钟的时间，而你想要在收到信号时停止处理，却不想让等待中的这一行中断，这时，只要在信号处理子程序中设定一个标记，然后在每行处理结束时检查它即可：

```
my $int_count;
sub my_int_handler { $int_count++ }
$SIG{'INT'} = 'my_int_handler';
...
$int_count = 0;
while (<SOMEFILE>) {
  ... 一些需要花几秒时间来执行的操作 ...
  if ($int_count) {
    # 中断发生!
    print "[processing interrupted...]\n";
    last;
  }
}
```

这样在处理完每行之后，如果没有按下 Control-C，$int_count 的值将会是 0，循环会继续下一次的处理。否则中断处理程序会将 $int_count 加 1，之后的检查会让循环中止。

注 26： 哈希值也可以是子程序引用，但是这里暂不使用。

所以，你可以设定标记或是直接跳离程序，而这两种方法足以涵盖绝大部分捕捉信号的需求。但要记住目前的信号处理机制并非毫无缺陷【注 27】，所以在信号处理中请尽量少做事情，否则程序可能会在你想不到的时候崩溃。

习题

参考附录 A 解答下面练习：

1. [6] 写一个程序，它会进入某个特定的目录，比如系统的*根目录*。然后执行 *ls -l* 命令获得该目录内容的详细报告。如果你使用非 Unix 的系统，请使用该系统上相应的命令来取得详细的目录列表。

2. [10] 修改前面的程序，让它将命令的输出送到当前目录下的 *ls.out* 文件，错误输出则送到 *ls.err* 文件。请不必担忧结果文件为空的情况。

3. [8] 写一个程序，它会解析 *date* 命令的输出以判断今天是星期几。如果是工作日，则输出 `get to work`，否则输出 `go play`。*date* 命令的输出中，星期一是用 `Mon` 来表示的【注 28】。如果你使用非 Unix 系统因而没有 *date* 命令，那就做一个假的小程序，只要输出像 *date* 命令的输出结果即可。如果你保证不问下面两行小程序的原理，我们就无偿奉上：

```
#!/usr/bin/perl
print localtime( ) . "\n";
```

注 27：这正是 Perl 开发人员最迫切修正的问题之一，所以我们可以期待安全的信号处理会是 Perl 6 首要的新特性。目前主要的问题在于信号会在任何时间出现，甚至在 Perl 还没有准备好的时候。如果在 Perl 正在分配一块内存时某个信号突然进来了，而且没准信号处理程序恰好也要分配一块内存，这种情况下你的程序就报销了。你无法控制 Perl 程序什么时候分配内存空间，通常用 C 写成的 XSUB 程序代码倒是很适合进行信号处理。关于这个高级主题的详细信息，请参阅 Perl 的说明文档。

注 28：起码在英语环境中是这样的。你可以根据系统设定灵活调整。

第十七章

高级 Perl 技巧

到目前为止，本书已经展现了语言核心，也就是每个 Perl 用户都需要明白的东西。然而还有些可选的技术装备，它们价值不菲但并非必知必会。我们把它们当中最重要的那些都收编在这一章里。

当然别被这一章的标题吓到了，这些技术并非特别难懂。我们所谓的高级，其实只是从初学者的角度来说的。所以第一次阅读本书的时候，为了尽快使用 Perl，你可能想要跳过这个章节，在一两个月之后，为了更多了解 Perl，你也可以回头再学。就把这一章想像为超级脚注好了【注 1】。

用 eval 捕获错误

有时看上去平淡无奇的代码却能导致严重错误。以下的这些典型语句都能让程序崩溃：

```
$barney = $fred / $dino;          # 零作除数?

print "match\n" if /^($wilma)/;   # 不合法的正则表达式?

open CAVEMAN, $fred               # 用户输入所产生的错误?
  or die "Can't open file '$fred' for input: $!";
```

你只要尝试捕捉这些潜在问题，就会知道捕获所有的潜在异常非常困难。比如在刚才的例子中，你要如何检查字符串 `$wilma` 才能确保它引入的正则表达式合法呢？好在 Perl 提供了简洁的 `eval` 块来捕获代码运行时的严重错误：

```
eval { $barney = $fred / $dino } ;
```

注 1：　我们真的试图把这种超级脚注加入草稿，但被 O'Reilly 的编辑断然拒绝。

现在即使 $dino 是零，这一行也不致于让程序崩溃。这里的 eval 其实是一个表达式（而不是一种控制结构，如 while 或者 foreach），因此在块的结束必须要有分号。

如果在运行时 eval 这块代码真的诱发了常见错误，就会停止运行，但不至于使程序崩溃。这就意味着在 eval 结束时，我们需要知道程序到底是正常结束，还是触发异常而导致结束。答案是查看专用的 $@ 变量。如果 eval 曾经触发异常，$@ 就会含有（几乎崩溃的）程序写下的遗言，没准是“Illegal division by zero at my_program line 12”。如果没有错误的话，$@ 的内容就是空。这意味着 $@ 的内容应该可以被当做逻辑真假值：真反而说明有错误。这就是为什么你会常常看到 eval 块之后有如下的代码：

```
print "An error occurred: $@" if $@;
```

eval 同时也真的是一个块结构，这意味着可以引入新的 my（词法）变量。这段程序显示了 eval 块如何完成许多的工作：

```
foreach my $person (qw/ fred wilma betty barney dino pebbles /) {
  eval {
    open FILE, "<$person"
      or die "Can't open file '$person': $!";

    my($total, $count);

    while (<FILE>) {
      $total += $_;
      $count++;
    }

    my $average = $total/$count;
    print "Average for file $person was $average\n";

    &do_something($person, $average);
  };

  if ($@) {
    print "An error occurred ($@), continuing\n";
  }
}
```

在这段程序中可能捕获多少异常？如果在文件打开过程中有错误，就会被捕捉；如果在平均值计算时候有“除以零”的错误，也会被捕捉；甚至神秘的 &do_something 子程序调用中的错误也会连带着被捕获。因为 eval 块激活期间会捕获所有发生的错误。（调用他人书写的子程序的时候，这个特性非常棒。因为编程中常常无法充分估计到异常的发生。）

如果错误在处理某个文件的时候发生，我们会获得错误信息，不过程序会跳过剩下的步骤，接着处理下一个文件。

你可以试试把一个 eval 块嵌在另一个 eval 块的里面。最里层的那个会优先捕获运行时错误，不让它泄露到外层。当然在内层的 eval 结束的时候，你可能希望用 die 来重新产生异常，从而给外层的 eval 留下捕获机会。一个 eval 会捕获运行期间的所有异常，包括那些因为子程序调用引发的异常，就像刚才看到的那样。

我们曾提到 eval 是一个表达式，因此在花括号结束之后必须跟一个分号。但是既然这是一个表达式，就得有个返回值。如果没有发生异常，则会返回最后一个被计算的表达式的值，或是可有可无的 return 操作符返回的值。以下就是免于“除以零”错误的算法：

```
my $barney = eval { $fred / $dino };
```

如果 eval 捕获了一个异常，返回值就是 undef 或者一个空列表，具体是什么要看上下文。在前面一个例子中，$barney 要么是一个除法运算的正确结果，要么就是 undef。我们不必去检查 $@ 的值，当然在进一步使用 $barney 的值之前做一个 defined($barney) 检查倒是必要的。

有 4 种类型的问题是 eval 无法捕获的。首先是那些能让 Perl 解释器崩溃的严重错误，比如内存不足或者收到 Perl 不可接管的信号。如果连 Perl 都无法正常运行下去，显然无法捕获这些错误【注 2】。其次，eval 中的语法错误会在编译时就被抓住，用不着通过 $@ 返回。

再次，exit 操作符将导致程序立即退出，哪怕是在 eval 块中调用的。在写子程序的时候应该养成的习惯，就是不要使用 exit，而应该用 die 来汇报问题。

最后一类 eval 无法捕捉的问题是警告，无论是用户通过 warn 产生的警告，还是 Perl 内部 -w 命令行参数或者 use warnings 编译命令产生的警告。要让 eval 捕获警告有一套专门的机制，可以参考 Perl 文档中 __WARN__ 这个伪信号。

需要指出 eval 也有另外一种不易掌握的形式，实际上这种形式常常带来危险。你会常常听到有人说，考虑到代码安全，不应使用 eval。他们的正确之处在于指出了使用 eval 时，要加倍小心安全漏洞，但是他们说的其实是 eval 的另一种形式，也就是 eval 字符串。如果 eval 的对象是一对花括号中的代码块，就完全可以放心，这是安全的。

注 2：　到底是哪些异常如此特殊？它们已经在 *perldiag* 文档中用 (X) 记号标明了。

用 grep 来筛选列表

有时候你可能希望选出列表中的部分成员。例如选出奇数，或者筛选文件中提到了 Fred 的行。在这一节你会看到，其实列表中的那些元素只需要用 grep 操作符即可筛选出来。

让我们先试试解决第一个问题，从一大堆数字中挑出奇数。我们可以不加思索的写出：

```
my @odd_numbers;

foreach (1..1000) {
  push @odd_numbers, $_ if $_ % 2;
}
```

这段代码用了取余操作符（%），这曾在第二章介绍过。如果收到一个偶数，它对 2 取余会得到零，也就是假。而奇数会得到 1，也就是真，因此只有奇数会被塞入数组中。

其实写这段代码并不是犯罪，只是达不到真正的简练和高效。因为 Perl 已经有了 grep 操作符。

```
my @odd_numbers = grep { $_ % 2 } 1..1000;
```

这一行简短的代码可以得到 500 个奇数的列表。它是如何做到的呢？grep 的第一个参数是代码块，其中的“代换”变量是 $_，代码块对列表的每个元素计算，并返回真或假。而代码块之后的参数则是将要被筛选的元素列表。grep 操作符对列表的每个元素算出代码块的值，好像之前版本的 foreach 循环做的那样。代码块计算结果为逻辑真的那些元素，将会出现在 grep 操作符返回的列表中。

在 grep 运行过程中，$_ 会轮流成为列表中每个元素的化名。这和之前看到的 foreach 循环版本是一样的。因此如果在 grep 表达式中修改 $_ 的内容就会破坏原始列表，这通常会导致很糟糕的结果。

grep 操作符和 Unix 经典正则文本过滤器 grep 同名，当然我们的 Perl 版本更加强劲。现在我们从一个文件中滤出包含了 fred 的行。

```
my @matching_lines = grep { /\bfred\b/i } <FILE>;
```

grep 还有一个更加简单的版本。如果你需要的只是用一个简单的表达式进行选取（用不着一个块），只要用逗号结束那个表达式就行。以下就是刚才例子的简化版本：

```
my @matching_lines = grep /\bfred\b/i, <FILE>;
```

用 map 对列表进行转换

另一个常见问题是如何对整个列表进行演算。例比要将一组数字按财务数据格式输出，像第十四章中的 &big_money 子程序做到的那样。只是现在的要求是读取原始列表，并产生格式化的新列表（用于输出）。下面是一种可行的方案：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
my @formatted_data;

foreach (@data) {
  push @formatted_data, &big_money($_);
}
```

这段代码和 grep 那一节开头的代码看来很像，不是吗？这意味着用来替代的代码看来同样很像 grep 版本的：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);

my @formatted_data = map { &big_money($_) } @data;
```

map 操作符和 grep 非常相似，因为它们有同样的参数：一个关注 $_ 的代码块和一个待处理的列表。而且它们的工作模式也非常相似，为每个成员执行一次代码块，块中用 $_ 这个化名迭代原始列表每个成员。但是差别在于：块最后返回的结果不是真假，而是新产生的列表元素【注 3】。任何 grep 或 map 都可以用 foreach 循环来重写，当然还得把结果存放在一个临时数组中才行。但是简短些的版本通常也更便捷和高效。另外 grep 或 map 的结果仍然是一个列表，所以可以直接传给另一个函数去处理。比如要用右对齐的财务数据格式打印列表：

```
print "The money numbers are:\n",
  map { sprintf("%25s\n", $_) } @formatted_data;
```

当然我们还可以一次性完成这个处理，不需要使用临时数组 @formatted_data：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
print "The money numbers are:\n",
  map { sprintf("%25s\n", &big_money($_) ) } @data;
```

如同 grep 一样，我们也可以用更简单的语法来调用 map。如果你只需要一个简单的表达式进行映射，那么就用一个逗号跟在表达式后面，不需要再写一个块了。

```
print "Some powers of two are:\n",
  map "\t" . ( 2 ** $_ ) . "\n", 0..15;
```

注 3：另一个重要区别在于 map 表达式是在列表上下文中计算的，因此可以返回多个元素，可能导致不常见的（一对多）映射。

不带引号的哈希键

Perl 提供了很多简写来释放程序员的创造力。其中一个非常方便的地方就是，某些哈希键的引号可以省略。

当然不是说每个键值都可以省略引号，因为哈希键可能是任何字符串。但是键值常常很简单，如果其中仅仅包含字母、数字与下划线，并且没有用数字打头，你就可以省略引号。这种不加引号的简单字符串被称为裸词，因为它不需要引号。

使用这种简写的场合是哈希键最常见的地方：在哈希元素引用的花括号里。举例来说，你可以将 `$score{"fred"}` 简写成 `$score{fred}`。因为很多哈希键都是这么简单，所以能省略引号真的很方便。但请注意，如果有裸词以外的东西出现在花括号里，Perl 就会将它解读成表达式。

另一个哈希键会出现的地方就是用键 / 值对列表来初始化哈希的时候。胖箭头 (=>) 特别适合放在键与值之间，因为当哈希键是裸词时，胖箭头会为它加上引号。

```
# 关于保龄球比赛得分的哈希
my %score = (
  barney   => 195,
  fred     => 205,
  dino     => 30,
);
```

胖箭头和逗号之间有一处重要的差异：在胖箭头左边的裸词，会被视为已加上引号（不过它右边的内容就没有这个优待了）。胖箭头的这项功能还可以在其他地方使用，然而最常见的地方就是哈希。

切片

我们常常只需要关注列表当中的某些单元。例如 Bedrock 图书馆就用一个大文件来存放读者信息【注 4】。文件中的每一行都描述了一个读者，用 6 个字段（冒号作为分隔符）分别描述姓名、卡号、住址、宅电、工作电话和手头借阅数量。文件的内容看来如此：

```
fred flintstone:2168:301 Cobblestone Way:555-1212:555-2121:3
barney rubble:709918:3128 Granite Blvd:555-3333:555-3438:0
```

注 4：虽然其实他们也知道应该用大型数据库而非平文本，但是他们计划的系统升级时间是下个冰河纪以后。

图书馆的某个应用程序只需要卡号和借出数量，不关心其他数据。所以可以这样来读出需要的两个字段：

```
while (<FILE>) {
  chomp;
  my @items = split /:/;
  my($card_num, $count) = ($items[1], $items[5]);
  ...   # 现在可以用这两个变量来继续工作
}
```

但是 @items 数组不会有其他的用处，看来是一种浪费【注 5】。也许用一组标量来容纳 split 的结果会更好些：

```
my($name, $card_num, $addr, $home, $work, $count) = split /:/;
```

好的，这确实避免了引入导致浪费的数组 @items，但是我们现在有了 4 个不需要的标量。有人图方便，将这种占位变量命名为 $dummy_1，表示 split 出来的此位置上的元素是无用的。但 Larry 觉得这么做太麻烦，因此他引入了一种特殊的 undef 写法。如果被赋值的列表中含有 undef 的话，就干脆忽略源列表的相应元素：

```
my(undef, $card_num, undef, undef, undef, $count) = split /:/;
```

这不是更好吗？应该说这样确实避免了引入垃圾变量，但是问题是要弄清 undef 的数量，才能正确获取 $count。而且如果列表元素数量稍微多些就更麻烦了。若要获取 stat 结果中的 mtime 数值则必须写出如下的代码：

```
my(undef, undef, undef, undef, undef, undef, undef,
  undef, undef, $mtime) = stat $some_file;
```

如果弄错了 undef 的数量，就可能获得 atime 或者 ctime 的值，这会是一个很难发现的 bug。更好的办法是使用数组下标来检索列表内容，这就是所谓的列表切片。因为 mtime 是 stat 产生列表的第 9 个元素【注 6】，我们可以这样获取数据：

```
my $mtime = (stat $some_file)[9];
```

这里 stat 周围的括号是必须的，因为需要用它们产生列表上下文。如果你像下面这样写，就不会正常工作：

```
my $mtime = stat($some_file)[9];  # 语法错误!
```

注 5：其实也并非很大的浪费，只是对于那些不明白切片的人来说，对比差异才能改变他们习惯的写法。

注 6：其实是第 10 个元素，但是下标是 9，因为第一个元素的下标是 0。和数组下标从零开始是一样的。

列表切片必须有一对圆括号引出作为列表，后面跟上方括号括住的下标范围。但是函数为引入参数而使用的圆括号不算。

回到 Bedrock 图书馆的例子，切片的对象是 split 返回的列表。我们用下标 1 和 5 检索相应字段：

```
my $card_num = (split /:/)[1];
my $count = (split /:/)[5];
```

像这样在标量上下文中检索（每次取一个列表元素）也不错，但是如果能避免两次调用 split，就会更加简单和高效。因此让我们用列表上下文一次成型：

```
my($card_num, $count) = (split /:/)[1, 5];
```

这里用下标 1 和 5 来检索列表的内容，返回两个元素的列表。然后我们把结果赋值到两个 my 变量组成的列表中去，这恰好是我们期望的：一次切片成型，并轻松对两个变量赋值。

切片常常是从列表中读取少量值的最简单方法。这里我们从列表中拉出第一个和最后一个值，借助下标 -1 代表最后一个元素的便利【注 7】：

```
my($first, $last) = (sort @names)[0, -1];
```

切片的下标可以是任意顺序的，也可以重复。这个例子从10个元素的列表中找出5个元素：

```
my @names = qw{ zero one two three four five six seven eight nine };
my @numbers = ( @names )[ 9, 0, 2, 1, 0 ];
print "Bedrock @numbers\n";  # 打印：Bedrock nine zero two one zero
```

数组切片

前面的例子还可以进一步简化。在进行数组切片（而不是列表切片）的时候，圆括号并非必须。所以我们可以这样进行切片：

```
my @numbers = @names[ 9, 0, 2, 1, 0 ];
```

省略圆括号并非简写而已，其实是一种存取数组元素的不同方法：数组切片。我们曾经在第三章中提到 @names 前面的 @ 符号意味着所有的元素。其实从语言学角度来看，它更加像是一种复数的标志，非常类似英文 cats 和 dogs 后面的 s 字母。在 Perl 中美元符号意味着一个东西，而 @ 符号意味着一组东西。

注 7：用排序方法找最大／最小值并非最有效，但是 Perl 的排序还是非常高效的，对元素数量为三位数的列表来说是可以接受的。

切片总是一个列表，所以数组切片总是使用一个 @ 符号来标识。当你看见类似 @names[...] 的时候，需要以 Perl 的习惯来看开头的符号和后面的方括号。方括号意味着你要检索数组成员，@ 符号则意味着获取的是列表【注 8】，不像美元符号意味着获取单个元素。请参考图 17-1。

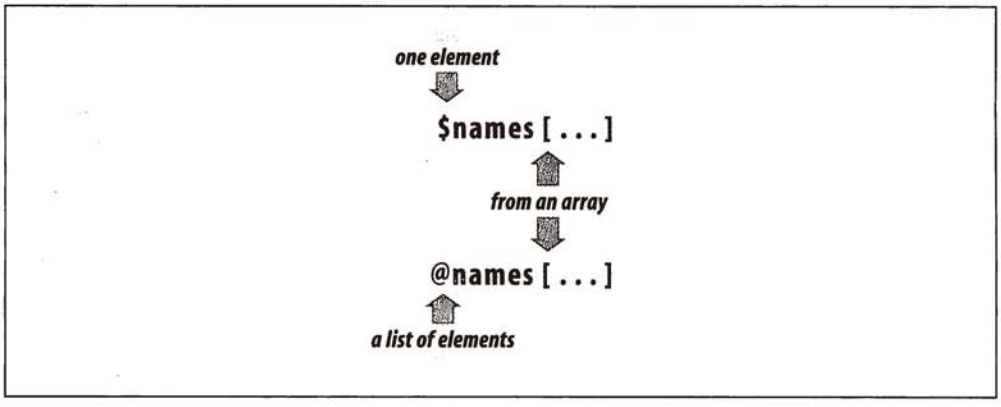

图 17-1：数组切片和单个元素的区别

变量之前的标点符号（美元 $ 或 @ 符号）决定了下标表达式的上下文。如果前面有个美元符号，下标表达式就会在标量上下文中计算。但是如果之前有个 @ 符号的话，下标表达式就会在列表上下文中计算，从而得到切片。

因此这里看到的 @names[2, 5] 和 ($names[2], $names[5]) 有同样的含义。如果希望得到一组值，就可以用数组切片的写法。任何需要列表的地方都可以替换成更简单的数组切片。

但是有个切片可以工作、列表却不能的场合。那就是切片可以被直接内插到字符串中去：

```
my @names = qw{ zero one two three four five six seven eight nine };
print "Bedrock @names[ 9, 0, 2, 1, 0 ]\n";
```

如果我们想要内插 @names，这就会产生数组所有成员的串，中间用空格填充。如果我们要内插的是 @names[9, 0, 2, 1, 0]，就会得到那些元素用空格填充的字符串【注 9】。让我们回到 Bedrock 图书馆，假设我们的程序需要修改读者 Slate 先生的

注 8：当然说到整个列表的时候，并不是意味着必须要多于一个元素，毕竟列表也可以为空。

注 9：更准确的说，列表的这些元素会被 Perl 内建的 $" 变量填充，而它的默认值就是空格。这一般不应该改变。在内插时候 Perl 会实际上用 join $", @list 运算，而 @list 也就是那个列表表达式。

地址和电话号码，因为他刚刚搬到了 Hollyrock 山庄的一个新家。如果我们得到了一个关于他的信息列表 @items，那么就可以简单的修改数组中的那两个元素。

```
my $new_home_phone = "555-6099";
my $new_address = "99380 Red Rock West";
@items[2, 3] = ($new_address, $new_home_phone);
```

和前面一样，数组切片可以用更简洁的方式来表示一系列的元素。在这个例子里，最后一行程序代码其实就是赋值给 ($items[2], $items[3])，但更简洁高效。

哈希切片

和数组切片相似，哈希成员也可以用哈希切片方式检索。还记得三个选手的保龄球积分吗？它们存放在哈希 %score 中。我们可以用哈希元素所构成的列表来取出这些积分，或是使用切片。这两个技巧实际的效果相当，但第二种做法更加简洁高效：

```
my @three_scores = ($score{"barney"}, $score{"fred"}, $score{"dino"});

my @three_scores = @score{ qw/ barney fred dino/ };
```

切片一定是列表，因此哈希切片也是用 @ 符号来表示【注 10】。如果看到类似 @names{ ...} 的代码，你需要以 Perl 的习惯来看开头的符号和后面的花括号。花括号意味着你要检索哈希，@ 符号意味着获取的是列表，而不是美元符号代表的单个值。请参考图 17-2。

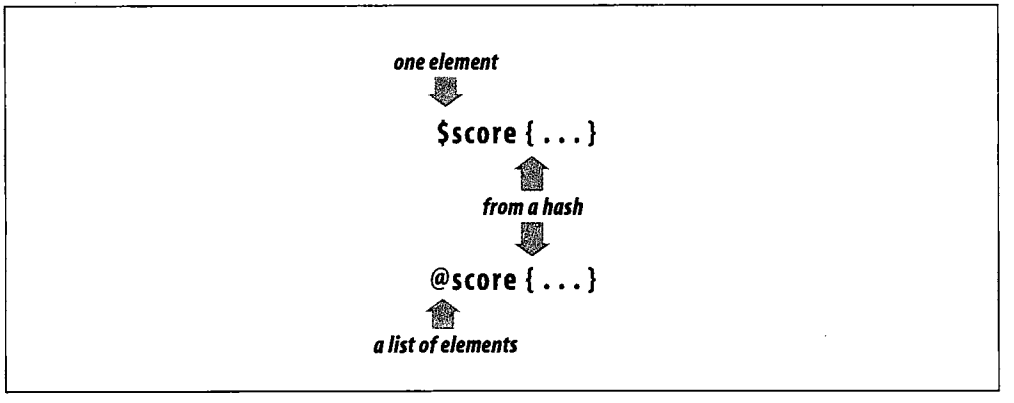

图 17-2：哈希切片和单个元素的区别

如同我们在数组切片中所见的，变量前置的标点符号（可能是美元符号或 @ 符号）决定了下标表达式的上下文。如果前置美元符号，下标表达式就会在标量上下文中运算以

注 10： 不管你是否认为我们啰嗦，我们就是想强调哈希切片和数组切片是类似的。

得出单一键值【注 11】。但是如果前置 @ 符号，下标表达式就会在列表上下文中计算以返回一组键值。

这里自然会有人问，为什么提到哈希时并没有用百分比符号（%）？因为百分比符号表示整个哈希，哈希切片（就像其他的切片）一定是列表而不是哈希【注 12】。在 Perl 中，美元符号代表一个东西，而 @ 符号则代表一组东西，百分比符号意味着一个哈希。

如同我们在数组切片中所见的，在 Perl 的任何地方，你都可以使用哈希切片来代表哈希里相应的元素。下面的程序可以将这些选手的保龄球分数存入哈希，不必担心意外修改哈希中的其他元素。

```
my @players = qw/ barney fred dino /;
my @bowling_scores = (195, 205, 30);
@score{ @players } = @bowling_scores;
```

最后一行所做的事相当于对（`$score{"barney"}, $score{"fred"}, $score{"dino"}`）这个具有三个元素的列表进行赋值。

哈希切片也可以内插进字符串。下面的例子会输出我们关注的保龄球选手的积分：

```
print "Tonight's players were: @players\n";
print "Their scores were: @score{@players}\n";
```

习题

下列习题的解答请参阅附录 A：

1. [30] 写个程序，从文件中读取一连串的字符串（每行一个），然后让用户键入模式以便进行字符串匹配。对每一个模式，程序应该说明文件里共有多少字符串匹配成功、各是哪些字符串。对于所键入的每个新模式，不应重新去读取文件，应该把这些字符串存放在内存里。文件名可以直接写在程序里。假如某个模式不合法（例如：括号不对称），那么程序应该汇报这些错误，并且让用户继续尝试其他模式。假如用户键入的不是模式而是空白行，那么程序就该停止运行。（如果你需要一个充满有趣字符串的文件来进行匹配，那么试试 *sample_text* 这个文件吧。你应该已经从 O'Reilly 的网站下载了这个文件了。下载方式请在序言中查找。）

注 11：有一个例外，但很难碰到，因为如今的 Perl 程序中很少使用它。参见 *perlvar* 在线手册中关于 $; 的部分。

注 12：哈希切片是一个切片而非哈希，如同炉火是指火而非炉，而火炉是指炉而非火。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

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

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

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

附录 A

习题解答

前面各章所附的习题可在此处找到解答。

第二章习题解答

1. 下面是其中一种做法：

```
#!/usr/bin/perl -w
$pi = 3.141592654;
$circ = 2 * $pi * 12.5;
print "The circumference of a circle of radius 12.5 is $circ.\n";
```

正如你所看到的，此程序以常见的 #! 行开头，你机器上的 Perl 安装路径可能会有所不同。另外，我们也启用了警告信息。

程序代码的正文里的第一行会将 $pi 的值设成我们需要的 π (3.141592654)。这种使用常量【注 1】的做法有许多好处：在 3.141592654 重复出现时，可以节省键入的时间；避免在某处使用 3.141592654，却在另一处使用 3.14159 所造成的意外错误；你只需要检查一行程序代码，就可以避免因为不小心键入 3.141952654 而让宇宙飞船飞到别的星球去；键入 $pi 要比键入 π 容易得多，特别是在你没有 Unicode 环境的时候。并且，这样做在 π 的值改变的时候也利于程序的维护【注 2】。接下来，我们会计算出圆周长并将它保存到 $circ 中，然

注 1：如果你喜欢用更正式的方式来使用常量，那么你可以使用 constant 编译命令。

注 2：它（π）的取值的最近一次改变发生在一个多世纪之前，印地安纳州州议会通过了该决议。请参看 1897 年印第安纳州州议会的第 246 号文件 *http://www.cs.uwaterloo.ca/~alopez-o/math-faq/node45.html*。

后漂亮地将其输出。信息最后面是换行符，因为只要是合格的程序，每行的输出都该以换行符作为结尾。如果没有换行符，视 shell 的提示符而定，则输出结果可能会变成这样：

```
The circumference of a circle of radius 12.5 is
78.53981635.bash-2.01$[]
```

行尾的方框代表闪烁的光标，也就是 shell 在信息结尾的提示符【注 3】。既然圆周长不应该是 78.53981635.bash-2.01$，那么这应该算是程序的 bug。因此，请务必在每一行输出的结尾加上 \n。

2. 下面是其中一种做法：

```
#!/usr/bin/perl -w
$pi = 3.141592654;
print "What is the radius? ";
chomp($radius = <STDIN>);
$circ = 2 * $pi * $radius;
print "The circumference of a circle of radius $radius is $circ.\n";
```

此程序和前一题类似，不过这次我们提示用户键入半径长度，然后用 $radius（半径）代替前一题里写定的 12.5。事实上，如果在写第一题的程序时能够考虑得更周全，当时我们也应该使用 $radius 这个变量。要注意的是，我们对输入值使用了 chomp，就算不这么做，上面的算式依然有效，因为 "12.5\n" 之类的字符串会自动转换成数字 12.5 。但是当我们要输出信息时，它看起来就会像这样：

```
The circumference of a circle of radius 12.5
 is 78.53981635.
```

我们发现，即使之前已经将 $radius 当成数字使用，但换行符还是会留在里面。因为在 $radius 和 is 中间有空格，所以输出的第二行开头也有空格。这个例子告诉我们，除非有特殊原因，否则请一律对输入值进行 chomp 处理。

3. 下面是其中一种做法：

```
#!/usr/bin/perl -w
$pi = 3.141592654;
print "What is the radius? ";
chomp($radius = <STDIN>);
$circ = 2 * $pi * $radius;
if ($radius < 0) {
$circ = 0;
}
print "The circumference of a circle of radius $radius is $circ.\n";
```

注 3：我们问了 O'Reilly，请他们帮我们用会闪烁的墨水来印这个输入光标，我们愿意额外付钱，但他们做不了。

在这里我们进一步地检查了有问题的半径值。即使所输入的半径值不合理，程序至少也不会返回负的圆周长。你也可以先将半径设成 0，再计算它的圆周长。办法不止一种。事实上，“办法不止一种（There is more than one way to do it!）”是 Perl 的座右铭。每道习题的解答都以“下面是其中一种做法”开头，道理就在此。

4. 下面是其中一种做法：

```
print "Enter first number: ";
chomp($one = <STDIN>);
print "Enter second number: ";
chomp($two = <STDIN>);
$result = $one * $two;
print "The result is $result.\n";
```

要注意的是，这个解答里省略了 #! 那行。事实上，接下来我们一律假设你已经知道它的存在，所以不用每次都重复提它。

上面的变量名称也许取得不太好。在长一点的程序里，维护程序员可能会认为 $two 的值应该是 2。在这么短的程序里没什么关系，但是如果程序很长，就该取比较具有描述性的名称，像 $first_response。

在这个程序里，无论我们有没有对 $one 和 $two 这两个变量进行 chomp 都无关紧要，因为它们在赋值之后不会被当成字符串来用。可是，如果维护程序员在下星期修改程序，让它输出像 The result of multiplying $one by $two is $result.\n 这样的信息，那么讨厌的换行符又会回来作祟。再次强调，除非有特殊原因（像下一题的情况），否则请一律对输入值进行 chomp 处理【注4】。

5. 下面是其中一种做法：

```
print "Enter a string: ";
$str = <STDIN>;
print "Enter a number of times: ";
chomp($num = <STDIN>);
$result = $str x $num;
print "The result is:\n$result";
```

从某个角度来看，这个程序和前一题几乎完全相同。在这里，我们也是计算字符串的重复次数，因此保留了和前一题相同的程序结构。不过，这次我们不想对第一行输入字符串进行 chomp，因为题目上要求将重复的每一行分开显示。这样一来，假设用户所输入的字符串是 fred 与换行符，而重复次数为 3 时，每行的 fred 后面就都会正确地加上换行符。

注 4：使劲地嚼与通常的咀嚼是一样的（chomping is like chewing）——虽然不是必须的，但大部分时候没有坏处。（chomp 刚好是使劲地嚼的意思，原文在这里打了个比方。）

程序尾端的 print 语句里，我们将换行符放在 $result 的前面，这样第一行的 fred 才会以自成一行的方式被显示出来。换句话说，我们不想让所输出的三行 fred 中只有两行对齐，如下所示：

```
The result is: fred
fred
fred
```

这次我们不必在 print 输出的结尾加上换行符，因为 $result 应该已经以换行符结尾了。

程序里的空格在大部分的情况下对 Perl 没有影响，要不要加空格是你的自由。但请小心，别拼错字了！如果程序里的 x 和它前面的变量名称 $str 间没有空格，Perl 所看到的将会是 $strx，从而导致运行失败。

第三章习题解答

1. 下面是其中一种做法：

```
print "Enter some lines, then press Ctrl-D:\n"; # 或者 Ctrl-Z
@lines = <STDIN>;
@reverse_lines = reverse @lines;
print @reverse_lines;
```

还有更简单的做法：

```
print "Enter some lines, then press Ctrl-D:\n";
print reverse <STDIN>;
```

除非所输入的列表在程序后面还会用到，否则大部分的 Perl 程序员都会选择第二种做法。

2. 下面是其中一种做法：

```
@names = qw/ fred betty barney dino wilma pebbles bamm-bamm /;
print "Enter some numbers from 1 to 7, one per line, then press Ctrl-D:\n";
chomp(@numbers = <STDIN>);
foreach (@numbers) {
print "$names[ $_ ? 1 ]\n";
}
```

因为数组索引是从 0 数到 6，所以这里必须将索引值减 1，好让用户能够从 1 数到 7。另外一种做法是在 @names 数组前面加上一个值来充数，比如下面的 dummy_item：

```
@names = qw/ dummy_item fred betty barney dino wilma pebbles bamm-bamm /;
```

如果你还额外检查了用户的输入是否在 1 到 7 的范围内，请给自己加分。

3. 如果想让所有的输出结果均显示在同一行，下面是其中一种做法：

```
chomp(@lines = <STDIN>);
@sorted = sort @lines;
print "@sorted\n";
```

或者让每行分开显示的做法：

```
print sort <STDIN>;
```

第四章习题解答

1. 下面是其中一种做法：

```
sub total {
  my $sum;  # 私有变量
  foreach (@_) {
    $sum += $_;
  }
  $sum;
}
```

这个子程序使用了 $sum 来存储到目前为止的总和。每次子程序开始执行时，$sum 都会是新创建的变量，因此其值为 undef。之后，foreach 循环会以 $_ 作为控制变量来逐项处理 @_ 里的参数列表。（请注意：我们再次强调参数数组 @_ 跟 foreach 循环的默认变量 $_ 之间并没有任何自动产生的联系。）

foreach 循环第一次执行时，会将第一项参数（存储在 $_ 中）与 $sum 相加。此时 $sum 的值还是 undef，因为它还没有被存入任何东西。但是，由于 Perl 能从数值操作符 += 判断它是被当成数字使用，所以会假设它的值是 0，然后再将总和存回 $sum 里。

当循环再次执行时，下一项参数也会与 $sum 相加，而这时 $sum 的值已经不是 undef 了。两者的总和又会存回 $sum 里，然后再以相同的方式处理接下来的参数。全部处理完毕之后，程序的最后一行会将 $sum 返回给调用者。

对某些人来说，这个子程序可能有缺陷。假设子程序被调用时有一个空的参数列表，像在例子中重写的子程序 &max。这样，$sum 的值将会是 undef，也就是此子程序所返回的值。但是在这个子程序里，“比较正确”的做法是将空列表的总和设成 0 而非 undef。（当然，如果你认为空列表的总和应该与 (3, -5, 2) 的总和有所区别，那么返回 undef 是正确的做法。）

如果不想看到未定义的返回值，办法很简单：请直接将 $sum 的初始值设为0，而不是默认的 undef：

```
my $sum = 0;
```

如此一来，即使参数列表是空的，这个子程序也一定会返回定义过的数值。

2. 下面是其中一种做法：

```
# 记得加上前一题里 &total 子程序的代码!
print "The numbers from 1 to 1000 add up to ", total(1..1000), ".\n";
```

要注意的是，我们不能在双引号括住的字符串里直接调用子程序【注5】，所以子程序调用是 print 的另一个独立参数。总和应该是 500500，一个很好看的整数。程序的运行应该花不了多少时间，传递 1000 个参数对 Perl 而言是常见的小事。

3. 下面是其中一种做法：

```
sub average {
  if (@_ == 0) { return }
  my $count = @_;
  my $sum = total(@_);                    # &total 来自前面的习题
  $sum/$count;
}

sub above_average {
  my $average = average(@_);
  my @list;
  foreach my $element (@_) {
    if ($element > $average) {
      push @list, $element;
    }
  }
  @list;
}
```

在 average 里，如果参数列表是空的，子程序就会结束，但并没有明确写上返回值。因此调用者将会取得 undef【注6】这个返回值，这表示空列表没有平均值。如果参数列表不是空的，那么 &total 就能帮忙计算其平均值。此处并无必要使用 $sum 与 $count 这两个临时变量，但它们能让程序变得容易阅读些。

第二个子程序 above_average 会建立并返回由期望的元素构成的列表（为何循环的控制变量是 $element，而不是 Perl 的“老地方”变量 $_？）。请注意，这个子程序对于空参数列表有不同的处理方式。

注5： 如果你用些高级技巧就可以做到。一般来说在 Perl 里没有什么是你完全无法做的。

注6： 如果 &average 在列表上下文中的话会返回一个空列表。

4. 要记住 greet 上一次对话的人，可以使用一个 state 变量。一开始它会是 undef，这样我们就知道 Fred 是它第一个问候的人。在这个子程序的结尾，我们把当前的 $name 保存在 $last_name 中，这样下一次我们才能记得它是什么：

```
use 5.010;

greet( 'Fred' );
greet( 'barney' );

sub greet {
    state $last_person;

    my $name = shift;

    print "Hi $name! ";

    if( defined $last_person ) {
        print "$last_person is also here!\n";
        }
    else {
        print "You are the first one here!\n";
        }

    $last_person = $name;
    }
```

5. 下面这种方法和前面的差不多，但是这次我们把所有出现过的名字都保存下来。我们不使用标量变量，而是用数组 @names 这个状态变量来保存所有的名字：

```
use 5.010;

greet( 'Fred' );
greet( 'barney' );
greet( 'Wilma' );
greet( 'Betty' );

sub greet {
    state @names;

    my $name = shift;

    print "Hi $name! ";

    if( @names ) {
        print "I've seen: @names\n";
        }
    else {
        print "You are the first one here!\n";
        }

    push @names, $name;
    }
```

第五章习题解答

1. 下面是其中一种做法：

```
print reverse <>;
```

嗯，蛮简单的！能够这样写，是因为 print 的参数是所要输出的字符串列表，也就是在列表上下文中调用 reverse 的结果。reverse 的参数是要被倒置的字符串列表，也就是在列表上下文中调用钻石操作符（diamond operator）的结果。钻石操作符所返回的列表是由用户选择的所有文件里的每一行所组成的。这个列表与 *cat* 命令所输出的结果相同。于是 reverse 会将此列表倒置，再交由 print 输出。

2. 下面是其中一种做法：

```
print "Enter some lines, then press Ctrl-D:\n";  # 或者 Ctrl-Z
chomp(my @lines = <STDIN>);

print "1234567890" x 7, "12345\n";  # 标尺行，到第 75 个字符的地方

foreach (@lines) {
  printf "%20s\n", $_;
}
```

此处，我们会先读取所有的文本行，再对它们进行 chomp 处理。接下来，我们会输出标尺行（ruler line）。由于它是帮忙调试的工具，所以在程序写完之后我们通常会把它变成注释。我们可以重复键入 "1234567890"，甚至使用复制与粘贴来制造出各种长度的标尺行，但是我们选择了上面这种做法，因为这么做比较酷。

接下来，foreach 循环会逐项处理列表里的每行文本，将它们交由 %20s 转换后输出。还有另一种做法可以一次输出全部的列表，而不必使用循环：

```
my $format = "%20s\n" x @lines;
printf $format, @lines;
```

这里有个常见的错误会让每行输出只有 19 个字符。假设你对自己【注 7】说："嘿，既然最后会将换行符加回去，一开始又何必对输入做 chomp 呢？"于是就省略 chomp，而把格式改成 "%20s"（不含换行符）【注 8】。程序运行时奇怪的事就发生了：输出结果少了一个空格。问题到底出在哪里？

注 7：或对 Larry 说，假如他站在你旁边的话。

注 8：除非 Larry 告诉过你别这么做。

这个做法会在 Perl 计算“需要多少个空格才有办法补齐所需字段的时候”发生问题。假设用户键入的是 **hello** 和换行符，则 Perl 看到的是 6 个字符而不是 5 个，因为换行符也算一个字符。所以，Perl 会输出 14 个空格以及含有 6 个字符的字符串，凑起来刚好就是你在 "%20s" 里所需要的 20 个字符。糟糕！

Perl 判断字符串长度时当然不会去看它的内容，Perl 只会检查其字符数量。多余的换行符（或是其他特殊符号，像制表符或空字符）会造成意料之外的计算结果【注 9】。

3. 下面是其中一种做法：

```
print "What column width would you like? ";
chomp(my $width = <STDIN>);

print "Enter some lines, then press Ctrl-D:\n";  # 或者 Ctrl-Z
chomp(my @lines = <STDIN>);

print "1234567890" x (($width+9)/10), "\n";       # 长度按需的标尺行

foreach (@lines) {
  printf "%${width}s\n", $_;
}
```

这段程序和前一题相似，只不过这次会先问字段宽度。在程序一开始就询问，是因为在键入文件结尾指示符（end-of-file indicator）之后就不能再取得输入了（至少在某些系统上是如此）。当然，在实际读取用户输入时通常会用更好的输入结尾指示符（end-of-input indicator）。在后面的习题解答中会看到实例。

与前一题的另一个差异是在标尺行的处理上。按照附加题里的条件，我们使用了一些数学技巧，让标尺行至少和需要的长度相同。一个额外的挑战：你能证明这里的算式是正确的吗？（提示：考虑 50 和 51 两种宽度，然后别忘了 x 对右边的操作数是取整数，而不是四舍五入。）

我们会用表达式 "%${width}s\n" 来产生这次的格式，其中使用 $width 进行内插。花括号是必要的隔离符号，可将变量名称与后面的 s 隔开；如果没有花括号，替换的就会是错误的变量 $widths。如果你忘了怎么使用花括号，也可以使用 '%' . $width . "s\n" 这样的表达式来产生相同的格式化字符串。

$width 的值是另一个需要 chomp 的例子。如果没有对字段宽度进行 chomp，最后的格式字符串看起来就会像 "%30\ns\n"。完全无效。

注 9： 现在 Larry 不欠你什么了。

以前知道 printf 的人也许还会想到另一种解法。既然 printf 是从 C 语言借来的，而且 C 语言里并没有字符串变量内插，因此我们也可以使用 C 程序员的技巧。在转换字符串里，如果在应该放数值的地方出现了星号（*），则会使用参数列表里的值来替代，如下所示：

```
printf "%*s\n", $width, $_;
```

第六章习题解答

1. 下面是其中一种做法：

```
my %last_name = qw{
  fred flintstone
  barney rubble
  wilma flintstone
};
print "Please enter a first name: ";
chomp(my $name = <STDIN>);
print "That's $name $last_name{$name}.\n";
```

在这个程序里，我们使用 qw// 列表（以花括号为界定符号）来初始化哈希。对于这个简单的数据设定而言并没有什么问题，因为数据的值是简单的名字与姓氏的配对，因此也很容易维护。但是，如果你的数据里含有空格，例如，如果 Robert De Niro（罗伯特·德·尼罗）或 Mary Kay Place（玛丽·凯·普莱斯）的话，这种简单的方法就不一定管用了。

你也可以将每一个键／值对分开设定，如下所示：

```
my %last_name;
$last_name{"fred"} = "flintstone";
$last_name{"barney"} = "rubble";
$last_name{"wilma"} = "flintstone";
```

请注意，如果你打算使用 my 来声明哈希（可能因为采用了 use strict），则必须在声明之后才可对元素进行赋值。你不能只对变量里的某部分使用 my，如下所示：

```
my $last_name{"fred"} = "flintstone";  # 哎呀，错啦!
```

换句话说，my 操作符只能声明独立的变量，不能用来声明数组或哈希里的元素。另外，请注意词法变量 $name 是在 chomp 函数调用的括号内声明的。这种“需要时再声明”的做法在 Perl 程序里十分常见。

在这段程序里，chomp 也是不可或缺的。如果有人键入了 "fred\n" 这 5 个字符，而我们又没有对它进行 chomp，程序就会去找键值为 "fred\n" 的哈希元

素，但是却找不到。当然，chomp 并不是万能的，如果所键入的是 "fred \n"（后面多了一个空格），我们就没办法以目前为止所学到的技巧来判断用户想要的其实是 fred。

如果你还检查了哈希的键是否存在（使用 exists 函数），以便在用户打错字时显示说明信息，请给自己加分。

2. 下面是其中一种做法：

```
my(@words, %count, $word);      # 声明变量（可以省略）
chomp(@words = <STDIN>);

foreach $word (@words) {
  $count{$word} += 1;           # 或者 $count{$word} = $count{$word} + 1;
}

foreach $word (keys %count) {   # 或者 sort keys %count
  print "$word was seen $count{$word} times.\n";
}
```

在这里，我们一开始就声明了所有的变量。这对用过 Pascal 等语言的人来说，可能会比“需要时再声明”要熟悉得多（在 Pascal 里，变量一定要在最前面声明）。当然，我们是假设 use strict 正在起作用所以才会声明这些变量。Perl 在默认的情形下并不需要这种声明。

接下来，我们会在列表上下文中使用“行输入”操作符 <STDIN> 将所有的输入行读进 @words 里，然后一次对全部的输入行进行 chomp。这样一来，@words 的内容就会是由所有输入的单词所组成的列表了（假设一切顺利，则每行只有一个单词）。

现在，第一个 foreach 循环会逐项处理各个单词。该循环中包含了整个程序里最重要的一行，它会将 $count{$word} 的值加上 1，然后再存回 $count{$word}。你也可以不使用 += 操作符，而改用比较长的写法。不过，较短的写法会稍微有效率一点，因为 Perl 只需要在哈希里查询一次 $word 就行了【注 10】。在第一个 foreach 循环里，每次出现的单词都会让 $count{$word} 的值加 1。假设第一个单词是 fred，则 $count{"fred"} 的值就会加1。既然这是第一次用到 $count{"fred"}，它的值自然是 undef。不过，因为我们将它当成数字来用（利用数值操作符 += 或是较长写法的 +），所以 Perl 会自动把 undef 转换为 0。相加的总和为 1，所以会将 1 存回 $count{"fred"}。

注 10：在 Perl 的有些版本中，这种短写法还可以避免输出一个变量未定义的警告信息。同样，你也可以对变量使用 ++ 操作符来避免这种警告信息。

在下一次 foreach 循环执行时，假设这次的单词是 barney。我们会将 $count{"barney"} 的值加1，让它也从 undef 变成 1。

现在，假设下一次的单词又是 fred。我们会将 $count{"fred"} 的值（也就是 1）再加上1而得到 2。$count{"fred"} 的值于是成为 2，表示到目前为止 fred 出现过两次。

处理完第一个 foreach 循环之后，我们已经计算出了每个单词的出现次数。哈希的键就是来自输入的单词，而相应的哈希值则是单词的出现次数。

最后，第二个 foreach 循环会逐项处理各个哈希键，也就是所有互不重复的单词。在这个循环里，每个不同的单词各会出现一次，而且每次会输出像“fred was seen 3 times”这样的信息。

附加题的解答：你可以在 keys 前面加上 sort 以便按照顺序输出哈希键。在输出结果超过十几行时将结果排序通常是件好事，它可以让调试的人迅速找到想要的条目。

3. 下面是其中一种做法：

```
my $longest = 0;
foreach my $key ( keys %ENV ) {
    my $key_length = length( $key );
    $longest = $key_length if $key_length > $longest;
    }

foreach my $key ( sort keys %ENV ) {
    printf "%-${longest}s  %s\n", $key, $ENV{$key};
    }
```

在第一个 foreach 循环中，我们会遍历所有的哈希键并使用 length 函数来得到它们的长度。如果当前的哈希键长度比保存在 $longest 变量中的长度还长，那么我们就把长一些的那个值保存在变量 $longest 中。

一旦我们遍历完所有的哈希键，我们就用 printf 函数把键和值分两列打印出来。这里使用在第五章的第三个练习中用过的同样的技巧，就是使用内插的方式将 $longest 替换进模板字符串中。

第七章习题解答

1. 下面是其中一种做法：

```
while (<>) {
  if (/fred/) {
    print;
  }
}
```

十分简单。本习题的重点其实是让你亲手试试例子里的各个字符串。它不会匹配 Fred，这表示正则表达式会区分大小写（我们稍后会提到如何不区分大小写）。它会匹配 frederick 和 Alfred，因为这两个字符串里都含有 fred 这 4 个字符（我们稍后会提到，如何比对独立的单词，让它不匹配 frederick 和 Alfred）。

2. 下面是其中一种做法：

把第一题的答案里的模式改成 /[fF]red/。除此之外，也可以试试 /(f|F)red/ 或 /fred|Fred/，不过使用字符集会比较高效。

3. 下面是其中一种做法：

把第一题的答案里的模式改成 /\./。必须加上反斜线（因为点号是元字符），或是写成字符集 /[.]/。

4. 下面是其中一种做法：

把第一题的答案里的模式改成 /[A-Z][a-z]+/。

5. 下面是其中一种做法：

把第一题的答案里的模式改成 /(\S)\1/。\S 字符集会匹配所有的非空白字符，而括号可以让你使用反向引用 \1 来匹配紧跟着的同样的字符。

6. 下面是其中一种做法：

```
while (<>) {
  if (/wilma/) {
    if (/fred/) {
      print;
    }
  }
}
```

此程序只有在 /wilma/ 匹配成功时才会测试 /fred/ 是否匹配。不过，fred 可以在 wilma 之前出现，也可以在它之后出现。这两项测试是互相独立的。

如果你想要省略掉第二层的 if 测试，也可以像下面这么写【注 11】：

```
while (<>) {
  if (/wilma.*fred|fred.*wilma/) {
    print;
  }
}
```

注 11：知道“逻辑与”操作符（我们将在第十章看到）的人可以在 if 判断的条件中同时对 /fred/ 和 /wilma/ 进行匹配测试。这样会更有效率，更容易扩展，比之前给出的方法好多了。但我们还没学过“逻辑与”操作符。

之所以能这样写，是因为若不是 wilma 在 fred 之前出现，就是 fred 在 wilma 之前出现。如果我们只写了 /wilma.*fred/，那么即使 fred and wilma flintstone 这一行中提到过 wilma 和 fred，还是不会与此模式相匹配。

我们把这一题当作附加题，是因为许多人在这里有理解上的障碍。我们提到了正则表达式里的“or”运算（也就是竖线符号“|”），但是却从来没有提到“and”运算。这是因为正则表达式里并没有“and”运算【注 12】。如果你想知道两个表达式是否都匹配成功，那就对它们都进行测试。

第八章习题解答

1. 有一种很简单的做法，我们已经直接写在章节的正文中了。其输出应为 before<match>after，如果你的输出不是这样，那就是你绕远路了。

2. 下面是其中一种做法：

```
/a\b/
```

（当然可以参考模式测试程序里面的模式！）如果不幸匹配了 barney，说明你可能需要使用单词边界锚位。

3. 下面是其中一种做法：

```
#!/usr/bin/perl
while (<STDIN>) {
  chomp;
  if (/(\b\w*a\b)/) {
    print "Matched: |$`<$&>$'|\n";
    print "\$1 contains '$1'\n";        # 多输出一行
  } else {
    print "No match: |$_|\n";
  }
}
```

这是稍微修改过的模式测试程序，除了模式不同外，也额外加了一行打印 $1 的程序代码。

此处的模式在括号内使用了一对 \b 单词边界锚位【注 13】，但即使写在括号外面，

注 12：但实际上还是有一些需要技巧的高级方式能实现类似的“and”操作。通常这些方式会比 Perl 的“逻辑与”效率要低，但也不一定，这得看 Perl 和它的正则表达式引擎会怎么优化。

注 13：老实说，其实并没有必要写上第一个锚位，详细的原因与星号的贪婪特性有关，但此处不谈。没有第一个锚位可能会比较有效率，有第一个锚位则比较清晰，所以最后我们还是选择了清晰。

也完全没有任何差别。那是因为锚位只会对应到字符串中的某个位置，而不会对应到某个字符：它的宽度为零（不会被括号捕捉到）。

4. 下面这道习题的解答和前面的效果差不多，但是用了不同的正则表达式：

```
#!/usr/bin/perl

use 5.010;

while (<STDIN>) {
  chomp;
  if (/(?<word>\b\w*a\b)/) {
    print "Matched: |$`<$&>$'|\n";
    print "'word' contains '$+{word}'\n";        # 新的输出行
  } else {
    print "No match: |$_|\n";
  }
}
```

5. 下面是其中一种做法：

```
m!
  (\b\w*a\b)        # $1：某个以字母 a 结尾的英文单词
  (.{0,5})          # $2：后面接上的字符不超过 5 个
!xs                 # /x 和 /s 修饰符
```

（因为现在使用了两个内存变量，所以别忘了补上显示 $2 的程序代码。如果你自己又将模式修改成只使用一个内存变量，请把多余的那一整行标为注释。）如果不再成功匹配 wilma，也许在模式中需要把“零个以上字符”改成“一个以上的字符”。/s 修饰符可以暂时忽略，因为数据里面应该没有换行符（如果有的话，/s 修饰符可能会产生不同的输出）。

6. 下面是其中一种做法：

```
while (<>) {
  chomp;
  if (/\s+$/) {
   print "$_#\n";
  }
}
```

井号（#）在这里用作标示，表示行尾的位置。

第九章习题解答

1. 下面是其中一种做法：

```
/($what){3}/
```

在 $what 替换完成后，会产生类似 /(fred|barney){3}/ 的模式。如果省略圆括号，模式会变成 /fred|barney{3}/，这也就等于是 /fred|barneyyy/。因此，圆括号是不可或缺的。

2. 下面是其中一种做法：

```
my $in = $ARGV[0];
unless (defined $in) {
  die "Usage: $0 filename";
}

my $out = $in;
$out =~ s/(\.\w+)?$/.out/;

unless (open IN, "<$in") {
  die "Can't open '$in': $!";
}

unless (open OUT, ">$out") {
  die "Can't write '$out': $!";
}

while (<IN>) {
  s/fred/Larry/gi;
  print OUT $_;
}
```

此程序一开始会先清点它的命令行参数，预期应该要有一个。如果没有取得，就抱怨一下；如果取得，则把参数复制到 $out 并把扩展名换成 .out（其实直接把文件名附加上 .out 就行了）。

IN 和 OUT 这两个文件句柄都被打开之后的部分，才是程序最主要的部分。如果你没有使用 /g 和 /i 这两个修饰符，请自行扣半分，因为这样一来就没办法换掉所有的 fred 和 所有的 Fred 了。

3. 下面是其中一种做法：

```
while (<IN>) {
  chomp;
  s/fred/\n/gi;          # 将所有的 FRED 替换为临时的占位符
  s/Wilma/fred/gi;       # 将所有的 WILMA 替换为 Fred
  s/\n/Wilma/g;          # 再将所有占位符换回为 Wilma
  print OUT "$_\n";
}
```

请把上题程序的循环换成这段循环。要进行这种互换，我们必须先找到一个“占位符”，而且必须是不会出现在数据中的。因为使用了 chomp（最后输出的时候会补上一个换行符），所以我们知道换行符（\n）是绝对不会出现在字符串中的，所以换行符就可以充当占位符。NUL 字符（\0）也是另一个不错的选择。

4. 下面是其中一种做法：

```
$^I = ".bak";           # 准备备份
while (<>) {
  if (/^#!/) {          # 是 #! 开头的那行吗?
    $_ .= "## Copyright (C) 20XX by Yours Truly\n";
  }
  print;
}
```

运行此程序时，应该在命令行参数中指定需要更新的文件。假设你的习题文件名都以 ex 开头，比如 *ex01_1*、*ex01_2*，那么你可以这样下命令：

```
./fix_my_copyright ex*
```

5. 为了避免重复加上版权声明，我们得分两回处理所有文件。第一回，我们会先建立一个哈希，它的键是文件名称，而它的值是什么并不重要。为了简单起见，此处将值设为 1：

```
my %do_these;
foreach (@ARGV) {
  $do_these{$_} = 1;
}
```

第二回，我们会把这个哈希当成待办列表（to-do list）逐个处理，并把已经包含版权声明行的文件移除。目前正在读取的文件名称可用 $ARGV 取得，所以可以直接把它拿来当哈希键：

```
while (<>) {
  if (/^## Copyright/) {
    delete $do_these{$ARGV};
  }
}
```

最后的部分就跟之前所写的程序一样，但我们会事先把 @ARGV 的内容改掉：

```
@ARGV = sort keys %do_these;
$^I = ".bak";           # 准备备份
while (<>) {
  if (/^#!/) {          # 是 #! 开头的那行吗?
    $_ .= "## Copyright (c) 20XX by Yours Truly\n";
  }
  print;
}
```

第十章习题解答

1. 下面是其中一种做法：

```
my $secret = int(1 + rand 100);
# 在调试时，去掉下面这行注释
# print "Don't tell anyone, but the secret number is $secret.\n";

while (1) {
  print "Please enter a guess from 1 to 100: ";
  chomp(my $guess = <STDIN>);
  if ($guess =~ /quit|exit|^\s*$/i) {
    print "Sorry you gave up. The number was $secret.\n";
    last;
  } elsif ($guess < $secret) {
    print "Too small. Try again!\n";
  } elsif ($guess == $secret) {
    print "That was it!\n";
    last;
  } else {
    print "Too large. Try again!\n";
  }
}
```

此程序的第一行会从范围 1 到 100 挑出一个秘密数字，运作细节如下。首先，rand 是 Perl 的随机数函数，所以 rand 100 会产生 0 以上 100 以下的随机数。也就是说，该表达式的最大值差不多是 99.999【注 14】。加 1 之后，数字的范围将会是 1 到 100.999，然后使用 int 函数取出整数的部分，这就是我们所需要的范围 1 到 100 的数字。

放在注释后面的程序代码可协助程序的开发与调试，也可以帮你作弊。程序的主要部分是无穷的 while 循环。执行到 last 之前，它会让我们不断猜下去。

测试数字之前先测试字符串，这一点很重要。如果我们不这样做，你猜得出用户键入 quit 时会怎么样吗？它会被解释成数字(如果启动警告，就会显示警告信息)，因为它作为数字使用时是 0，可怜的用户会收到“数值太小”的信息。这样的话，我们可能根本执行不到字符串测试的部分。

这里的无穷循环还有另外一种写法，就是使用裸块及 redo。这么写既不会执行得比较慢，也不会比较快，只是写法不同而已。一般来说，如果大部分的时候会继续循环，应该使用 while，因为它默认会继续循环。如果只有在例外状况下才会继续循环，那么裸块也许是比较好的选择。

注 14：真正的上限视你的系统而定。如果你真想知道，参见 *http://www.cpan.org/doc/FMTEYEWTK/random*。

2. 这个程序是在之前的解答基础上做了少量的修改。我们需要在开发过程中打印秘密数字，所以在 $Debug 变量为真的时候调用 print。而 $Debug 的值要么来自于环境变量，要么是默认值 1。通过使用 // 操作符，我们在 $ENV{DEBUG} 未定义的时候设置它为 1：

```
use 5.010;

my $Debug = $ENV{DEBUG} // 1;

my $secret = int(1 + rand 100);

print "Don't tell anyone, but the secret number is $secret.\n"
    if $Debug;
```

如果不使用 Perl 5.10 的特性，就必须做些额外的工作：

```
my $Debug = defined $ENV{DEBUG} ? $ENV{DEBUG} : 1;
```

3. 这种方法参考了第六章的练习三答案：

在程序开始的时候，我们设置了环境变量的值。键 ZERO 和 EMPTY 对应假值（而非未定义），而键 UNDEFINED 没有值。

之后在 printf 的参数列表中，使用 // 操作符来针对未定义的值打印 (undefined)：

```
use 5.010;

$ENV{ZERO}      = 0;
$ENV{EMPTY}     = '';
$ENV{UNDEFINED} = undef;

my $longest = 0;
foreach my $key ( keys %ENV )
    {
    my $key_length = length( $key );
    $longest = $key_length if $key_length > $longest;
    }

foreach my $key ( sort keys %ENV )
    {
    printf "%-${longest}s  %s\n", $key, $ENV{$key} // "(undefined)";
    }
```

通过使用 //，可以确保对 ZERO 和 EMPTY 键对应的假值不做处理。

若不使用 Perl 5.10 的功能，也可以使用三目操作符：

```
    printf "%-${longest}s  %s\n", $key,
        $ENV{$key} ? $ENV{$key} : "(undefined)";
```

第十一章习题解答

1. 这个答案里用了哈希引用（关于哈希引用就得请你去看羊驼书了，但是我们在此已经展示了它的用法。只要你知道怎么用它，就暂时别担心其运作细节。先把事情办完，再慢慢来学习和了解）。

   下面是其中一种做法：

```
#!/usr/bin/perl

use Module::CoreList;

my %modules = %{ $Module::CoreList::version{5.006} };

print join "\n", keys %modules;
```

第十二章习题解答

1. 其中一种做法：

```
foreach my $file (@ARGV) {
  my $attribs = &attributes($file);
  print "'$file' $attribs.\n";
}

sub attributes {
  # 报告某个文件的属性
  my $file = shift @_;
  return "does not exist" unless -e $file;

  my @attrib;
  push @attrib, "readable" if -r $file;
  push @attrib, "writable" if -w $file;
  push @attrib, "executable" if -x $file;
  return "exists" unless @attrib;
  'is ' . join " and ", @attrib;  # 返回值
}
```

   在这个例子里，使用子程序仍然是比较方便的做法。对于每个文件，主循环会用一行来输出它的属性，也许它会告诉我们 `'cereal-killer'` 是可执行的(executable)，而 `'sasquatch'` 则是不存在的。

   上面的子程序会告诉我们某个文件的属性。当然，如果文件根本不存在，那就不需要进行其他测试了。因此我们会先测试它是否存在，如果不存在，就提早返回。

   如果文件确实存在，我们将会建立一个列表用来存储文件的属性（如果你使用特殊的 _ 文件句柄而非 `$file`，以避免重复调用系统测试每项属性的话，请给自己加

分)。要(效仿上面三个测试)加入新的测试是很简单的。但是,如果所有测试都不成功呢?嗯,即使不能说什么别的,起码可以说它存在,所以我们就这样做了。如果 @attrib 里有任何元素的话,它的值就会是真(这是在“布尔上下文”里一种特殊的标量上下文)。这里的 unless 子句正是利用了这个上下文。

不过,要是取得了某些属性,我们就可以用 "and" 把它们连接起来(使用 join)并且在前面加上 "is",以造出 is readable and writable 这样的语句。这样的处理并不完美,如果有三个属性的话,它会说该文件 is readable and writable and executable。虽然句子里有太多 and,但还算可以接受。如果想要再加上更多其他属性的测试,而你又在意这种事情的话,也许该将它的输出改成像 is readable, writable, exeutable, and nonempty 这样。

要注意的是,如果你碰巧没有在命令行键入任何文件名,则程序将不会有任何输出。这很合理:如果你查询零个文件的信息,本来就该得到零行的结果。但是,这种做法可以跟下一题的程序作个比较。

2. 下面是其中一种做法:

```
die "No file names supplied!\n" unless @ARGV;
my $oldest_name = shift @ARGV;
my $oldest_age = -M $oldest_name;

foreach (@ARGV) {
  my $age = -M;
  ($oldest_name, $oldest_age) = ($_, $age)
    if $age > $oldest_age;
}

printf "The oldest file was %s, and it was %.1f days old.\n",
  $oldest_name, $oldest_age;
```

程序一开始会检查文件名,如果没有取得任何文件名,就会显示错误信息。这是因为程序的用途是找出最旧的文件,如果没有取得文件名,自然就不会有最旧的文件。

我们再一次用到了“高水线”(high-water-Mark)算法。第一个文件当然是目前唯一看过的文件中最旧的。我们必须记下它的年龄,并存储在 $oldest_age 变量里。

对于每个文件,我们都会像上面那样利用 -M 文件测试来取得它们的年龄(不过,这里用的是 $_ 的默认参数)。一般所谓的文件“年龄”,通常指的是上次修改的时间,虽然你也可以做不同的解释。如果目前的文件年龄大于 $oldest_age,我们就会以列表赋值的方式同时更新文件名和年龄变量的值。尽管不一定要采取列表赋值的方式,但它确实是一次更新数个变量的好方法。

此程序中，我们将 -M 返回的年龄存进临时变量 $age 里。如果不使用临时变量，每次都直接使用 -M，又会怎样呢？首先，除非使用特殊的 _ 文件句柄，否则我们每次都得向操作系统询问文件的年龄，这可能会多花一些时间（你大概不会注意到，除非有成千上百个文件；就算真的有这么多文件，也可能不会有什么影响）。不过更重要的是我们应该考虑到如果有人在我们进行检查的同时更新文件的话，该怎么办？也就是说，在我们第一次使用 -M 取得某个文件的年龄时，发现它是目前为止所看到最旧的。但是，在我们第二次使用 -M 之前，有人修改了那个文件，将它的时间戳设成了当前的时间。这样一来，实际存进 $oldest_age 里的可能是系统上年龄最轻的文件。程序运行的结果，就会是自该文件之后最旧的文件，而不是全部文件中最旧的。这种问题，调试起来会非常困难！

程序结尾处我们以 printf 输出文件名和年龄，并将天数取到小数点后一位。假如你将年龄转换成天数、小时及分钟来显示，请给自己加分。

3. 下面是其中一种做法：

```
use 5.010;

say "Looking for my files that are readable and writable";

die "No files specified!\n" unless @ARGV;

foreach my $file ( @ARGV ) {
    say "$file is readable and writable" if -o -r -w $file;
    }
```

使用栈式文件测试操作的前提是用 Perl 5.10 以上版本，因此我们使用 use 语句开头来确保版本正确。我们检查 @ARGV 数组，确保其中有数据供 foreach 处理，否则呼叫 die。

我们使用三个文件测试操作符：-o 用来检查是否拥有文件，-r 用来检查文件是否可读，而 -w 用来检查是否可写。把它们堆叠在一起成为 -o -r -w，可以一并检查是否通过，也就是我们需要的效果。

如果要用 Perl 5.10 之前的版本完成以上功能，也只是稍微多些代码而已。需要用 print 带上回车来模拟 say，并且用短路操作符 && 来连接文件测试：

```
print "Looking for my files that are readable and writable\n";

die "No files specified!\n" unless @ARGV;

foreach my $file ( @ARGV ) {
    print "$file is readable and writable\n"
        if( -w $file && -r _ && -o _ );
    }
```

第十三章习题解答

1. 下面是其中一种做法，使用文件名通配：

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {          # 若是空行
  chdir or die "Can't chdir to your home directory: $!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

my @files = <*>;
foreach (@files) {
  print "$_\n";
}
```

首先，我们会显示一个简单的提示，然后取得用户想要的目录，对它进行必要的chomp（如果没有 chomp，则会尝试转到一个（名字结尾有换行符的）目录。这在Unix 上是合法的，所以 chdir 函数导致意外产生）。

接下来，如果目录名称不是空的，我们会切换到该目录下，遇到错误就中断执行。如果名称是空的，就以 home 目录来代替。

最后，对“星号”的通配会返回（新）工作目录中所有的文件名，并自动按字母排序，然后逐一输出。

2. 下面是其中一种做法：

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {          # 若是空行
  chdir or die "Can't chdir to your home directory:
$!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

my @files = <.* *>;        ## 现在加上了 .*
foreach (sort @files) {    ## 现在排序
  print "$_\n";
}
```

和前一题有两点差别：第一，这次的通配操作包含了“点号星号”，它会匹配所有以点号开头的文件名；第二，我们必须对所得到的列表进行排序，因为取出的列表中，以点号开头的文件名会和不以点号开头的文件名交错排列，看起来比较凌乱，排序之后会清楚很多。

3. 下面是其中一种做法：

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {          # 若是空行
  chdir or die "Can't chdir to your home directory:
$!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

opendir DOT, "." or die "Can't opendir dot: $!";
foreach (sort readdir DOT) {
  # next if /^\./;              ## 如果我们要跳过点开头的文件
  print "$_\n";
}
```

这个程序的结构同前两题，但是现在我们改用打开目录句柄的方式。变更工作目录（working directory）之后，我们会打开当前目录（current directory），也就是 DOT 目录句柄（directory handle）。

为什么要用 DOT 呢？如果用户键入了像 /etc 这样的绝对目录名称，那么打开它并没有什么问题。但是，如果用户键入了像 fred 这样的相对目录名称呢？让我们来看看会发生什么事。首先，让我们 chdir 到 fred 目录，然后再用 opendir 来打开 fred。可是，这样会打开新目录里的 fred，而不是原来目录里的 fred。只有 . 总是表示“当前目录”（起码在 Unix 和类似的系统上是这样）。

readdir 函数会取得目录里所有的文件名，然后再由程序将它们排序输出。如果以这种方式来做第一题，那么我们就应该略过以点号开头的文件。要这么做，只需把 foreach 循环里的注释去掉就行了。

你也许会怀疑：“为什么要先 chdir 呢？readdir 类型的函数对当前目录并不敏感，它其实可以作用在任何目录上”。最主要的动机是想让用户只按一个键，就可以转移到其 home 目录。但是，这个程序也可以作为“通用文件管理器”的雏形。也许接下来我们可以设计一个功能，让用户选择要备份目录里的哪些文件等。

4. 下面是其中一种做法：

```
unlink @ARGV;
```

如果想在程序遇到问题时对用户提出警告，也可以这样写：

```
foreach (@ARGV) {
  unlink $_ or warn "Can't unlink '$_': $!, continuing...\n";
}
```

这里每行的内容都会化名为 $_，然后成为 unlink 的参数。如果其中出现问题，警告信息能提供线索。

5. 下面是其中一种做法：

```
use File::Basename;
use File::Spec;

my($source, $dest) = @ARGV;

if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

rename $source, $dest
  or die "Can't rename '$source' to '$dest': $!\n";
```

程序里实际做事的只有最后一行，其他的程序是为了“把文件移动到目录中”而存在的。先在一开始声明所用到的模块，再为命令行参数取有意义的名称。如果 $dest 是目录，我们需要从 $source 名称中取出文件的 basename，并将它附加到 $dest 后面。最后，一旦 $dest 经过必要的处理，rename 函数会执行实际改名的动作。

6. 下面是其中一种做法：

```
use File::Basename;
use File::Spec;

my($source, $dest) = @ARGV;

if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

link $source, $dest
  or die "Can't link '$source' to '$dest': $!\n";
```

正如题目中所提示的，此程序和前一题十分相似，唯一的差别在于这次执行的是 link 而非 rename。如果你的系统不支持硬链接，则最后的语句可以改成这样：

```
print "Would link '$source' to '$dest'.\n";
```

7. 下面是其中一种做法：

```
use File::Basename;
use File::Spec;

my $symlink = $ARGV[0] eq '-s';
shift @ARGV if $symlink;

my($source, $dest) = @ARGV;
if (-d $dest) {
```

```
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

if ($symlink) {
  symlink $source, $dest
    or die "Can't make soft link from '$source' to '$dest': $!\n";
} else {
  link $source, $dest
    or die "Can't make hard link from '$source' to '$dest': $!\n";
}
```

开头几行程序代码（在两个 use 声明之后）会先检查第一个命令行参数，如果它是 -s，就表示所要建立的是软链接，所以我们将此判断的真假值存储在 $symlink 变量中。如果检查到了 -s，我们还得将它去掉，也就是下一行程序代码所做的事。之后的数行程序代码是从上一题的解答复制过来的。最后，依照 $symlink 的值是真是假，程序会选择建立硬链接或软链接。最后，我们还更改了 die 后面的信息，让它清楚显示出我们试图建立的是哪一种链接。

8. 下面是其中一种做法：

```
foreach (<.* *>) {
  my $dest = readlink $_;
  print "$_ -> $dest\n" if defined $dest;
}
```

通配操作所返回的每个条目都会依次作为 $_ 的值。如果该条目是软链接，那么就会由 readlink 返回一个已定义的值并且输出链接位置；如果不是，测试条件就会失败，从而使得程序略过该条目。

第十四章习题解答

1. 其中一种做法：

```
my @numbers;
push @numbers, split while <>;
foreach (sort { $a <=> $b } @numbers) {
  printf "%20g\n", $_;
}
```

程序代码的第二行实在令人困惑，不是吗？嗯，这是故意的。虽然我们建议你编写清楚易懂的程序代码，但是也有人以写出复杂难解的程序为乐【注 15】，所以你最

注 15：应该说，我们不建议你在日常写程序的时候使用。不过把编写令人困惑的程序当作游戏来玩还是挺有趣的，而花一两个周末来搞懂别人写的迷津程序（obfuscated program）也会受益匪浅。如果想看看这些程序或找人帮忙解码，请在下次 Perl Monger 大会上问问。你也可以在 Web 上搜索 YAPH，或是看看自己能否解开这一章结尾处的迷津程序。

好能预先做好准备。总有一天，你也会需要维护这种难懂的程序代码。

因为那一行用到了 while 修饰符，所以与下面的循环等效：

```
while (<>) {
  push @numbers, split;
}
```

这样好多了，但是也许还是有点不清楚（不过，这种写法我们可以接受。它还没有复杂到“一眼难以看懂”）。while 循环每次会读入一行（从用户所要求的输入来源，也就是钻石操作符），接着（在默认的状况下）split 会以空白来分割该行，于是会产生一个单词列表，也就是数字列表，毕竟这里的输入只不过是一系列以空白分隔的数字而已。这样一来，无论输入怎么排列，while 循环都会将其中所有的数字存进 @numbers 里。

接下来，foreach 循环会逐行输出排序过的列表，使用 %20g 数值格式来让它们靠右对齐。如果你使用 %20s，又会怎样呢？嗯，因为后者是字符串格式，所以它不会更改输出中的字符串。你是否注意到样本数据里同时包含了 1.50 和 1.5，以及 04 和 4 呢？如果你将它们当成字符串输出，那么多余的零字符还会留在输出结果里；但是 %20g 是数值格式，所以相等数值的呈现方式也会相同。这两种格式都有可能是对的，具体使用哪个要根据情况决定。

2. 下面是其中一种做法：

```
# 别忘了在练习用的文件和下载来的文件中改用哈希 %last_name 的另一种写法

my @keys = sort {
  "\L$last_name{$a}" cmp "\L$last_name{$b}"  # 按姓氏排序
   or
  "\L$a" cmp "\L$b"                          # 按名字排序
} keys %last_name;

foreach (@keys) {
  print "$last_name{$_}, $_\n";              # 打印：Rubble,Bamm-Bamm
}
```

对于这个程序没什么好解释的。它会依题目的要求对哈希键进行排序，然后输出。我们之所以会先输姓再输名，纯粹只为了好玩而已，题目里并没有指定要用哪种显示方式。所以此题的答案就留给你自己去解释了。

3. 下面是其中一种做法：

```
print "Please enter a string: ";
chomp(my $string = <STDIN>);
print "Please enter a substring: ";
chomp(my $sub = <STDIN>);
```

```
my @places;

for (my $pos = ?1; ; ) {                    # 三块结构的技巧性用法
  $pos = index($string, $sub, $pos + 1);    # 找出下个位置
  last if $pos == ?1;
  push @places, $pos;
}

print "Locations of '$sub' in '$string' were: @places\n";
```

这个程序的开头十分简单。它要求用户键入字符串，然后声明一个数组来存储子串出现的位置。但是接下来的 for 循环似乎就又是个“精巧至上”的程序代码。做这种事好玩可以，但绝不应该在实际应用的程序里出现。不过，这里展示的技巧可能以后用得到，所以让我们来看看它是如何运作的。

用 my 来声明的 $pos 变量是 for 循环有效范围内的私有变量，初始值为 -1。这里我们就不再卖关子了，直接告诉你它的功能是存储在较长的字符串里子串的出现位置。for 循环的“测试”和“递增”部分都是空的，所以这是个无穷循环（当然，我们终究会脱离循环的，这次是用 last）。

循环主体的第一行语句会从位置 $pos + 1 开始寻找子字符串的出现位置。也就是说，在循环第一次执行 $pos 还是 -1 的时侯，会从位置 0 （字符串开头）开始寻找。接着把子串的出现位置存入 $pos。如果它是 -1，就不必再执行 for 循环了，所以我们会用 last 来脱离循环。如果 $pos 不是 -1，我们就会将位置存进 @places，然后再进行下一次循环。这时，$pos + 1 会让程序在继续寻找子串时，从上次出现的位置的后面一格开始。如此一来，我们得到了想要的解答，一切又都恢复了原来的平静。

如果你不想使用这种奇妙的 for 循环，也可以用下面的写法来得到相同的结果：

```
{
  my $pos = ?1;
  while (1) {
    ... # 和上面代码中的循环部分相同
  }
}
```

外层的裸块限制住了 $pos 的有效范围。你不一定得这么做，但是在尽可能小的有效范围内声明变量通常是比较好的做法。这么做能减少程序里同时“活着”的变量，让我们得以降低之后不小心将 $pos 这个名称用到别处的可能性。基于同样的道理，如果不将变量声明在较小的有效范围里，通常就应该给它取较长的名称，以避免之后不小心重复用到。在这个程序里，$substring_position 就是一个不错的名字。

另一方面，如果你想让程序代码成为迷津（你真坏！），同一个程序也可以写成如

下所示的大怪物（我们真坏！）：

```
for (my $pos = ?1; ?1 !=
  ($pos = index
    +$string,
    +$sub,
    +$pos
    +1
  );
push @places, ((((+$pos)))) {
    'for ($pos != 1; # ;$pos++) {
      print "position $pos\n";#;';#' } pop @places;
}
```

这份更刁钻古怪的程序代码可以代替原本程序里奇妙的 for 循环。到了这里，你的知识应该已经足以解读出它的意义了。现在，你也可以写出自己的迷津程序，让朋友吃惊、使敌人困惑。请将这份力量用以为善，不要作恶。

对了，假设你在 This is a test. 里寻找 t 的话，结果会是什么呢？它出现在 10 和 13 这两个位置；它并不会出现在位置 0，因为它会区分大小写。

第十五章习题解答

1. 下面是其中一种做法，用来重写第十章中的猜数程序。我们不必使用智能匹配，但可以使用 given：

```
use 5.010;

my $Verbose = $ENV{VERBOSE} // 1;

my $secret = int(1 + rand 100);

print "Don't tell anyone, but the secret number is $secret.\n"
    if $Verbose;

LOOP: {

    print "Please enter a guess from 1 to 100: ";
    chomp(my $guess = <STDIN>);

    my $found_it = 0;

    given( $guess ) {
        when( ! /^\d+$/ )     { say "Not a number!" }
        when( $_ > $secret ) { say "Too high!"    }
        when( $_ < $secret ) { say "Too low!"     }
        default              { say "Just right!"; $found_it++ }
        }

    last LOOP if $found_it;
```

```
    redo LOOP;
}
```

在第一个 when 中，我们先检查是否为数字。如果没有数字字符，或整个串为空，就不用继续后面的数值比较了。

注意我们没有把 last 放在 default 块中。原先我们就是这么做的，但 Perl 5.10.0 版本会因此发出警告，所以现在去掉了（或许以后的新版本不会有这个问题）。

2. 下面是其中一种做法：

```
use 5.010;

for (1 .. 105) {
    my $what = '';
    given ($_) {
        when (not $_ % 3) { $what .= ' fizz'; continue }
        when (not $_ % 5) { $what .= ' buzz'; continue }
        when (not $_ % 7) { $what .= ' sausage' }
    }
    say "$_$what";
}
```

3. 下面是其中一种做法：

```
use 5.010;

for( @ARGV )
    {
    say "Processing $_";

    when( ! -e ) { say "\tFile does not exist!" }
    when( -r _ ) { say "\tReadable!"; continue }
    when( -w _ ) { say "\tWritable!"; continue }
    when( -x _ ) { say "\tExecutable!"; continue }
    }
```

如果在 for 块中使用 when，就不必再写 given 了。接下来的程序先判断文件是否存在（其实是反过来判断的）。如果执行到第一个 when 块当中，就会报告文件不存在，并靠隐含的 break 跳出。避免执行之后的 when 块。

在第二个 when 当中，我们测试文件可读与否，这次用的是 -r 测试操作符。另外还用到了特殊的虚文件句柄 _ 来访问文件缓冲（也就是上次文件测试打开的 stat 信息）。如果不写 _，其实程序也能工作，只是测试更繁琐些。块的最后使用了 continue 来跳入下一个 when 测试。

4. 以下是使用 given 和智能匹配的一种做法：

```
use 5.010;
```

```
say "Checking the number <$ARGV[0]>";

given( $ARGV[0] ) {
    when( ! /^\d+$/ ) { say "Not a number!" }

    my @divisors = divisors( $_ );

    my @empty;
    when( @divisors ~~ @empty ) { say "Number is prime" }

    default { say "$_ is divisible by @divisors" }
    }

sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. $number/2 ) {
        push @divisors, $divisor unless $number % $divisor;
        }

    return @divisors;
    }
```

首先汇报正在处理的参数。这个习惯很好，可以告诉大家程序还在运行。我们用尖括号来区分 $ARGV[0] 内容和其他的内容。

在 given 中，我们写了两个 when 块，用来组织一些其他语句。前面的 when 用来判断参数确实是数值。如果那个模式匹配失败，我们就用相应块内的代码输出“Not a number!”。这个 when 块有隐含的 break 功能，可以退出整个 given 结构。如果测试通过，就能调用 divisors()。这个调用其实可以写在 given 之外，但那样可能有风险。万一传入的不是数字，比如是 'Fred' 怎么办？为避免 Perl 警告信息，我们还是把 when 当成一种预警机制。

一旦除法结束，我们就希望知道 @divisors 中是否有什么数据。这时候当然可以在标量上下文中获取元素的数量，但是也可以使用智能匹配。我们早就知道在比较两个数组的时候，需要有同样的元素和同样的顺序。这里我们创建一个空数组 @empty，自然没有任何元素。如果拿它和 @divisors 比较，智能匹配就只会在没有因数的时候才能成功。如果成功的话，就可以执行相应的 when 块，这一块也是靠隐式的 break 退出的。

最终通过测试的数一定不是质数，所以用 default 块来打印整除数字列表。

这里还有些更加精彩的内容，虽然我们承诺在《Learning Perl》中不提引用，而是留到《Intermediate Perl》再说。但还是忍不住要告诉你，声明一个空数组用于比较是多余的，其实可以用匿名数组一次成形：

```
when( @divisors ~~ [] ) { ... }
```

5. 这里是一种解决方法，在前一个练习基础上写成：

```
use 5.010;

say "Checking the number <$ARGV[0]>";

my $favorite = 42;

given( $ARGV[0] ) {
    when( ! /^\d+$/ ) { say "Not a number!" }

    my @divisors = divisors( $ARGV[0] );

    when( @divisors ~~ 2 ) {      # 如果 2 在 @divisors 里面
        say "$_ is even";
        continue;
        }

    when( !( @divisors ~~ 2 ) ) { # 如果 2 不在 @divisors 里面
        say "$_ is odd";
        continue;
        }

    when( @divisors ~~ $favorite ) {
        say "$_ is divisible by my favorite number";
        continue;
        }

    when( $favorite ) { # $_ ~~ $favorite
        say "$_ is my favorite number";
        continue;
        }

    my @empty;
    when( @divisors ~~ @empty ) { say "Number is prime" }

    default { say "$_ is divisible by @divisors" }
    }

sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. ($ARGV[0]/2 + 1) ) {
        push @divisors, $divisor unless $number % $divisor;
        }

    return @divisors;
    }
```

这个扩展练习增加了更多的 `when` 块来完成更多的情况判断。一旦获得

@divisors 数组，就可以用智能匹配来检查其中内容。如果 2 在整除数字之中，就可以断定这是偶数。我们在报告之后用显式的 continue 来驱动 given 结构执行之后的 when 判断。对于奇数的情况，同样使用智能匹配，只是对结果取反。同样的技巧还可以判断是否我们喜欢的数字在 @divisors 中，或者输入的恰巧就是我们喜欢的数字。

第十六章习题解答

1. 其中一种做法：

```
chdir "/" or die "Can't chdir to root directory: $!";
exec "ls", "-l" or die "Can't exec ls: $!";
```

第一行程序代码会将当前工作目录切换到根目录，它的名称总是固定的。第二行使用了多参数的 exec 函数来将结果传送到标准输出。我们也可以使用单参数的形式，但上面的做法并没什么不好。

2. 下面是其中一种做法：

```
open STDOUT, ">ls.out" or die "Can't write to ls.out: $!";
open STDERR, ">ls.err" or die "Can't write to ls.err: $!";
chdir "/" or die "Can't chdir to root directory: $!";
exec "ls", "-l" or die "Can't exec ls: $!";
```

程序的前两行会重新打开 STDOUT 与 STDERR 并将它们重定向到当前工作目录下的两个文件里（在切换工作目录之前）。接下来，在工作目录切换之后，目录列表命令（*ls*）将会执行，并把数据传送到之前打开的两个文件里。

最后一个 die 所显示的信息会出现在哪里呢？当然，它会跑到 *ls.err* 里，因为在那时 STDERR 已经被定向到该文件里了。至于 chdir 后面的 die 也会将信息传送到 *ls.err* 里。可是，如果在第二行上无法重新打开 STDERR，错误信息又会在哪里出现呢？它会在原本的 STDERR 上出现。这是因为当 STDIN、STDOUT、STDERR 这三个标准文件句柄重新打开失败时，原先的文件句柄仍然会继续打开着。

3. 下面是其中一种做法：

```
if (`date` =~ /^S/) {
  print "go play!\n";
} else {
  print "get to work!\n";
}
```

因为 Saturday（周六）和 Sunday（周日）都是以 S 开头，而且 *date* 命令的输出

结果又是以“今天是星期几”作为开始，所以这个程序十分简单，只要检查 *date* 命令的输出，看看它是否以 S 开头就行了。还有许多更复杂的做法可以获得相同的结果，其中大部分我们都介绍过了。

不过，如果要实际应用这个程序，我们大概会将模式换成 `/^(Sat|Sun)/`。它会稍微慢一点点，但是几乎没什么影响；再说，这对维护程序员而言要好懂多了。

第十七章习题解答

1. 下面是其中一种做法：

```
my $filename = 'path/to/sample_text';
open FILE, $filename
  or die "Can't open '$filename': $!";
chomp(my @strings = <FILE>);
while (1) {
  print "Please enter a pattern: ";
  chomp(my $pattern = <STDIN>);
  last if $pattern =~ /^\s*$/;
  my @matches = eval {
    grep /$pattern/, @strings;
  };
  if ($@) {
    print "Error: $@";
  } else {
    my $count = @matches;
    print "There were $count matching strings:\n",
      map "$_\n", @matches;
  }
  print "\n";
}
```

此程序使用 `eval` 块来捕捉使用正则表达式时可能发生的错误。在 `eval` 块里，`grep` 会筛选出字符串列表中匹配模式的字符串。

一旦 `eval` 执行完毕，程序会汇报错误信息或显示匹配的字符串。请注意，为了在每个字符串后面加上换行符，我们使用 `map` 来对输出进行“unchomp”操作。

附录B

超越小骆驼

本书已涵盖甚广，但难免还有内容我们没有提及。在附录的这个部分，我们将会多谈谈Perl能做什么，并提供能让你更深入了解Perl的参考资料。此处提及的东西有些是崭新的，因此在你阅读本书的时候可能已经有了更新。我们常常让你去看文档以了解全貌。正是因为这些变化，我们不奢望每位读者都逐字阅读本附录，但我们希望你至少会略读标题，这样当某人对你说“你根本不能在甲项目用Perl，因为Perl不能实现乙功能”时，你才能好整以暇地回应。

最重要的是（为了免于在每一段后面反复说：）我们未能介绍的其他重要部分已由《Intermediate Perl》（也就是O'Reilly的羊驼书）所涵盖。请读羊驼书，尤其在你（独自或与他人合作）写出上百行程序代码时。没准你已经对Fred与Barney的故事感觉厌烦了，想要知道另外世界中的7个人【注1】在海难中漂流到荒岛上之后的求生故事。

在羊驼书之后，还可以去看《Mastering Perl》（也就是O'Reilly的驼羊书）。这本书涉及了日常的Perl编程中遇到的问题，例如：性能检测和调试、配置文件、日志等等。还介绍了如何分析他人代码，并最终将它们与自己的应用程序集成。

更多文档

Perl自带的文档乍看似乎浩如烟海，不过你可以用关键字在文档中搜索。搜索特定主题时，从*perltoc*（总目录）和*perlfaq*（常见问答集）这两节开始会比较好。在大部分的系统上，*perldoc*命令应能查到Perl核心包、已安装的模块以及相关程序的使用说明

注1：　可以把他们称为漂流者。

(包括 *perldoc* 本身)。也可以在线阅读 *http://perldoc.perl.org*，虽然那里永远是最新版本的 Perl 手册。

正则表达式

没错，正则表达式的功能比我们所提到的还多。Jeffrey Friedl 的著作《Mastering Regular Expressions》(O'Reilly)是我们读过的在这方面最出色的技术书籍之一【注2】。该书前半部讨论一般的正则表达式，而后半部则说明各主要语言(当然包含 Perl)的正则表达式。此书深入介绍了正则表达式引擎内部的运作方式，并解释为什么某些模式的写法会比其他写法更好、更有效率。所有想要认真学习 Perl 的人都应该看看这本书。此外，也请参阅 *perlre* 在线手册(以及高版本 Perl 中加入的 *perlretut* 和 *perlrequick* 在线手册)。当然，羊驼书也提到了更多关于正则表达式的内容。

包

包【注3】让你可以划分多个名字空间(namespace)。想象一下，有10个程序员正在合力开发某个大型项目。当你在开发这个项目时，使用了 `$fred`、`@barney`、`%betty`、`&wilma` 等全局变量，如果我不小心也用了这些变量名称，会有什么后果呢？包会让我们分别保存这些变量名称，我可以访问你的 `$fred`，当然你同样也可以访问我所定义的这些变量而不会有意外发生。如果你想让 Perl 更灵活，使用包是必要的，它也可以让我们管理更大的程序。羊驼书对包也有详细的探讨。

扩展 Perl 的功能

在 Perl 相关的论坛中常见的忠告之一就是："不要重新发明轮子"。你可以拿其他人已经写过的程序代码去使用。最常见的方式就是利用某个函数库或模块来扩展 Perl 的功能。有许多模块会跟着 Perl 一起被安装起来，至于其他的模块则可以在 CPAN 找到。当然，你也可以编写一些属于自己的函数库或模块。

注2：这么说并不是因为出版商是 O'Reilly，而是因为书确实很棒。

注3：包这个名字可能是个不幸的选择，它没有让人正确地联想到名字空间，而是让人想到打包的代码集(而这在 Perl 里其实是模块或库)。包其实是全局符号名的集合，把 `$fred` 或 `&wilma` 之类的东西收集在一起。而名字空间和代码包是两个概念。

函数库

许多程序语言都提供了和 Perl 一样的函数库支持。函数库也就是为某个共同目标编写的子程序（subroutine）的集合。不过近来在 Perl 的使用上，使用模块的场合比函数库多。

模块

所谓模块也就是“聪明的函数库”。模块通常会提供一套用法如同内置函数的子程序。模块聪明的地方在于它将自己的实现细节放在独立的包中，只导出你需要使用的部分，如此可以避免模块覆盖掉你的程序中所定义的符号。

虽然大部分的模块都是由 Perl 写成的，但是也有一部分是由像 C 这样的语言来完成的。就像 MD5 算法，它是一组功能强大的校验和检查（checksum）函数【注 4】。使用了大量的低级位运算。这部分虽然 Perl 也可以做到，但相对而言，速度却可能慢上几百倍【注 5】。而使用 C 来实现这样的算法，能提高不少效率。`Digest::MD5` 模块就是使用了经过编译的 C 程序代码。当你使用这个模块时，可以让你像使用 Perl 模块一样计算出 MD5 的结果。

寻找与安装模块

也许你的系统里已经安装了一些需要的模块，但是要如何知道自己的系统里到底装了哪些模块呢？你可以从 CPAN 的 *http://www.cpan.org/authors/id/P/PH/PHOENIX/* 目录下载一个名为 *inside* 的程序来完成这项工作。

如果要用的模块还没安装，可以到 CPAN（*http://search.cpan.org/*）寻找。若想了解如何安装模块到系统中，请参考 *perlmodinstall* 在线手册的说明。

使用一个模块时，一般会把 `use` 指令放在程序最开始的地方。这样，可以让负责在新系统中安装程序的人一目了然，知道程序依赖哪些模块。

注 4：并非简单的校验和而已，但可以先这么理解。

注 5：要知道 `Digest::Perl::MD5` 是一个纯 Perl 的 MD5 算法实现。我们发现在同样的数据集上测试的时候，它比 `Digest::MD5` 要慢 280 倍。请注意 C 语言中位运算能够编译成单条指令，因此一行代码能在几个时钟周期内完成。Perl 是很快的，但是别太理想化。

编写自己的模块

在一些少见的情况下，你也可能找不到需要的模块。此时，资深的程序员可以用 Perl 或其他的程序语言（常常是 C 语言）写出一个新的模块。这部分的说明可以在 *perlmod* 和 *perlmodlib* 在线手册里找到。

几个重要的模块

这一节我们会介绍几个最重要模块【注 6】的最主要的功能【注 7】。除非另外有说明，否则这里所讨论的模块应该都能在安装了 Perl 的机器上找到。当然，你也可以在 CPAN 上找到它们的最新版本。

CGI 模块

许多人会用 Perl 编写可在 Web 服务器上运行的程序（通常称为 CGI 程序）。Perl 自带了 CGI 模块，在第十一章中已经有了一个例子，稍后你可以读到更多说明。

Cwd 模块

有时候，你需要知道当前工作目录的名称（通常也可以使用 .，但是也许你想把目录名称存起来以便稍后再切换回当前目录）。Perl 自带的 Cwd 模块提供了 cwd 函数，让你得以取回当前工作目录的名称。

```
use Cwd;

my $directory = cwd;
```

Fatal 模块

如果你觉得每次都必须在 open 或 chdir 之后写上 or die 很烦人，Fatal 模块也许就是专门为你而设计的。只要跟它说你要与哪个函数一起工作，这些函数就会自动检查是否执行失败。这样就有如你在它们后面写上了 or die 命令以及适当的错误信息。这么做并不会影响你所调用的其他包（例如，你所使用的模块里的程序代码），所以别用这种方式来修正没写好的程序。它只是让你可以节省一些时间，通常它会用在一些没

注 6：这里介绍的只是每个模块的最主要的功能，要进一步了解的话请查看模块自带的文档。

注 7：肯定还有更加重要的模块，但是往往对于本书的读者来说太复杂了。说复杂是因为它们都涉及到了 Perl 的引用和对象。

有必要直接控制错误信息的简单程序中。例如：

```
use Fatal qw/ open chdir /;

chdir '/home/merlyn';  # 现在 "or die" 机制已经被自动实现了
```

File::Basename 模块

我们在第十一章中曾说明过这个模块。它的主要用途是从完整的文件名中取出 basename 或目录名：

```
use File::Basename;

for (@ARGV) {
  my $basename = basename $_;
  my $dirname = dirname $_;
  print "That's file $basename in directory $dirname.\n";
}
```

File::Copy 模块

如果你需要复制或移动文件，File::Copy 模块就是专门为你而设计的（通常人们会想要直接调用系统程序来做这些事，但这么做将无法在不同的平台上执行）。本模块提供了 move 与 copy 函数，凡是相应的系统程序可以使用的地方都可以使用它们：

```
use File::Copy;

copy("source", "destination")
  or die "Can't copy 'source' to 'destination': $!";
```

File::Spec 模块

当你需要对文件名进行操作时（比较正确的说法是文件标识符），一般能跨平台且较可靠的方法就是使用 File::Spec，而不是直接使用 Perl。假使你想让一个目录名与一个文件名组合成完整的路径文件名（就像我们在第十一章中看到的），则可以使用 catfile 来达成，而不必去管你的程序所在的系统是用斜线还是其他字符来分隔它们。你也可以用 curdir 函数来取得当前目录的名称（任何 Unix 系统上都是 .）。

File::Spec 模块是用面向对象的方式写成的，但是你不必了解对象就能使用它。只要键入 File::Spec 并在后面接着一个小箭头以及你要使用的函数（其实是方法）名称就可以了。像这样：

```
use File::Spec;

my $current_directory = File::Spec->curdir;
opendir DOT, $current_directory
  or die "Can't open current directory '$current_directory': $!";
```

Image::Size 模块

当你有图像文件时，你也许想知道它的高度与宽度（编写产生 HTML 的程序时，如果你想要为 IMG 标记指定图像的大小时，这就很有用）。你可以从 CPAN 上取得 Image::Size 模块，它认得常见的 GIF、JFIF（JPEG）、PNG 以及其他的图像格式。例如：

```
use Image::Size;

# 获取图片 fred.png 的高度和宽度
my($fred_height, $fred_width) = imgsize("fred.png");
die "Couldn't get the size of the image"
  unless defined $fred_height;
```

Net::SMTP 模块

如果你希望程序能够用 SMTP 服务器发送电子邮件（这是我们现在最常用的方式），可以使用 Net::SMTP 模块来执行这项工作【注 8】。这个模块可以从 CPAN 取得。虽然它采用的是面向对象的设计方式，但是照着下面所列出的语法来使用就行了。你必须更改程序里 SMTP 主机名等设定才有办法在自己的系统上执行。你的系统管理员或附近的专家可以告诉你该用哪些设定。举例来说：

```
use Net::SMTP;

my $from = 'YOUR_ADDRESS_GOES_HERE';          # 例如 fred@bedrock.edu
my $site = 'YOUR_SITE_NAME_GOES_HERE';        # 例如 bedrock.edu
my $smtp_host = 'YOUR_SMTP_HOST_GOES_HERE';   # 例如 mail 或者 mailhost
my $to = 'president@whitehouse.gov';

my $smtp = Net::SMTP->new($smtp_host, Hello => $site);

$smtp->mail($from);
$smtp->to($to);
$smtp->data( );

$smtp->datasend("To: $to\n");
$smtp->datasend("Subject: A message from my Perl program.\n");
$smtp->datasend("\n");
```

注 8：　没错，你现在知道怎么用 Perl 发送垃圾邮件了。但是别那么干！

```
$smtp->datasend("This is just to let you know,\n");
$smtp->datasend("I don't care what those other people say about you,\n");
$smtp->datasend("I still think you're doing a great job.\n");
$smtp->datasend("\n");
$smtp->datasend("Have you considered enacting a law naming Perl \n");
$smtp->datasend("the national programming language?\n");

$smtp->dataend( );                              # 注意，不要写成 datasend!
$smtp->quit;
```

POSIX 模块

如果你需要使用 POSIX（IEEE 1003.1）功能，那么 POSIX 模块正适合你。它提供了许多 C 语言里的常用函数，例如三角函数（asin、cosh）、一般的数学函数（floor、frexp）、字符识别函数（isupper、isalpha）、低级的 I/O 函数（creat、open）以及一些其他的函数（asctime、clock）。你也许会想使用这些函数的“全名”来调用它们；这也就是说，在函数名称前面加上 POSIX 以及两个冒号，如下所示：

```
use POSIX;

print "Please enter a number: ";
chomp(my $str = <STDIN>);

$! = 0;  # 清空错误信息变量
my($num, $leftover) = POSIX::strtod($str);

if ($str eq '') {
  print "That string was empty!\n";
} elsif ($leftover) {
  my $remainder = substr $str, -$leftover;
  print "The string '$remainder' was left after the number $num.\n";
} elsif ($!) {
  print "The conversion function complained: $!\n";
} else {
  print "The seemingly-valid number was $num.\n";
}
```

Sys::Hostname 模块

Sys::Hostname 模块提供了 hostname 函数，它会返回主机名，只要它能判断得出来。如果它不能判断，也许是因为你的机器没有连接 Internet 或是没有设好。此函数在失败时会自动以 die 结束执行，因此在后面加上 or die 是没有意义的。举例来说：

```
use Sys::Hostname;
my $host = hostname;
print "This machine is known as '$host'.\n";
```

Text::Wrap 模块

Text::Wrap 模块提供了断行的功能，可用来进行简单的断行。最前面两个参数分别用来指定第一行与其他行的缩排，而剩余的参数则是文章的各个段落：

```
use Text::Wrap;

my $message = "This is some sample text which may be longer " .
  "than the width of your output device, so it needs to " .
  "be wrapped to fit properly as a paragraph. ";
$message x= 5;

print wrap("\t", "", "$message\n");
```

Time::Local 模块

如果想将某个时间值（例如像 time 函数的返回值）转换成年、月、日、时、分、秒等一系列数值，可以在列表上下文中使用 Perl 内置的 localtime【注 9】。（而在标量上下文中，它会返回格式化过的时间字符串，通常就是你想要的格式。）要是你想反过来，用年、月、日等数据来算出时间值，可以用此模块中的 timelocal 函数。需要注意的是，代表 2008 年 3 月的 $mon 和 $year 并不是 3 和 2008，因此在使用前请先阅读说明文档。例如：

```
use Time::Local;

my $time = timelocal($sec, $min, $hr, $day, $mon, $year);
```

编译命令

编译命令（pragma）是一组特殊的模块。每个版本的 Perl 里都有，用来影响内部编译过程。我们已经提到过 strict 这个编译命令。Perl 版本对应的 *perlmodlib* 在线手册列出了所有可用的编译命令。

编译命令和一般的模块相同，都是以 use 指令来下达。其中有些作用于词法作用域（lexically scoped）的编译命令和词法变量（my 变量）一样，只会作用在所属的最内层块或文件之内。其他的编译命令可能会对整个程序起作用，或是只作用在当前的包上。如果没有使用任何包，后者也会作用在整个程序上。一般来说，编译命令应该放在程序代码的开头。编译命令的作用范围应该都可以参考各自的说明文档。

注 9：　实际上 localtime 在列表上下文中的返回值会与期望值稍有不同，查看文档了解详情。

constant 编译命令

如果你曾用过其他程序语言，也许已经看过某些声明常量的方式。常量很适合那些需要在程序的开头处统一设定，同时又需要不时调整的值。Perl 可以利用 constant 编译命令来声明常量，它的作用范围是整个包。它会让编译器视某个标识符为常量值，在每次出现时进行优化。举例来说：

```
use constant DEBUGGING => 0;
use constant ONE_YEAR => 365.2425 * 24 * 60 * 60;

if (DEBUGGING) {
  # DEBUGGING 为真时输出调试信息；交付使用时可设为假，以跳过此段代码，优化性能
  ...
}
```

diagnostics 编译命令

Perl 的诊断信息经常令人难以理解，至少乍看之下是如此。不过，你总是可以在 *perldiag* 在线手册里找到解释，它往往还会附上问题的可能原因，并教你如何解决。diagnostics 编译命令可让你省掉搜索在线手册的麻烦，它会让 Perl 在任何信息出现时找到相关的信息并输出。跟大多数的编译命令不同，它并不是为了日常使用而设计的，因为它会让程序一开始就读入整份 *perldiag* 在线手册。这对速度和内存而言都是很沉重的负担。只有在调试的过程中，而且估计会遇到不熟悉的错误信息时，才需要使用这个编译命令。它的影响范围是整个程序。使用方式如下：

```
use diagnostics;
```

lib 编译命令

一般来说，模块总是应该安装在标准目录中，供所有人使用，但是只有系统管理员才有办法做这件事。如果你想要安装私用的模块，一般是安装在自己的目录里。那么，Perl 该怎样找到它们呢？这就是 lib 编译命令的作用了。它会告诉 Perl 要先到哪里寻找模块（因此，它也很适合用来试验某个给定模块的新版本）。它会影响之后加载的所有模块。语法是：

```
use lib '/home/rootbeer/experimental';
```

请务必以绝对路径为参数，因为程序运行时的当前工作目录可能会改变。这对于 CGI 程序（通过 Web 服务器运行的程序）而言特别重要。

strict 编译命令

你已经使用 use strict 好一阵子了，但往往还不知道它是编译命令。它只会作用于词法作用域（lexically scoped），能促使你遵从良好的程序设计规则。请参考它的说明文档，以了解它在你的 Perl 里加入了哪些限制。羊驼书则介绍了 strict 完成的其他事情。

vars 编译命令

在少数情况下，你可能需要在 use strict 作用时使用全局变量。这时，你可以用 vars 编译命令进行声明【注 10】。它的作用范围是整个包，可让 Perl 知道你是有意使用某些全局变量的：

```
use strict;
use vars qw/ $fred $barney /;

$fred = "This is a global variable, but that's all right.\n";
```

羊驼书包含了这方面的细节。

warnings 编译命令

从 5.6 版开始，Perl 可以使用 warnings 编译命令来控制块内的警告信息【注 11】。也就是说，与其使用 -w 选项来全面启用或停用警告信息，不如在某段程序中关闭未定义值相关的警告，但是仍然发出其他警告。这也可以作为留给维护程序员的信息："我了解这段程序代码会触发警告，但是我知道自己在做什么。"请参考这个编译命令的说明文档，其中会列出该 Perl 版本中的各类警告。

数据库

若你有数据库，Perl 也能为它服务。这一节会提到常见的几种数据库类型。我们已经在第十一章中的 DBI 模块中大致介绍过了。

注 10：如果你的程序永远不会在 5.6 版之前的 Perl 中运行，那么最好使用 our 关键字来代替 vars 编译命令。

注 11：如果你的程序的解释器是 5.6 版之前的 Perl，请不要使用 warnings 编译命令。

直接访问系统数据库

Perl 可以直接访问某些系统数据库，但有时候会需要模块的帮助。像 Windows 的注册表（Registry）数据库（记载了机器级的设定）、Unix 的密码数据库（列出了用户名称与相关信息）以及域名数据库（将 IP 地址转换成机器名，或是做反向转换）。

访问平面文件数据库

如果想直接访问自己的平面文件数据库，可以利用模块来完成（似乎每一两个月就会出现新的模块，所以这里所提供的任何列表都会过时）。

其他的操作符与函数

由于篇幅有限，我们无法介绍所有的操作符及函数：从 `..` 标量操作符到 `,` 标量操作符、从 `wantarray` 到 `goto` (!) 以及从 `caller` 到 `chr`。请参阅在线手册 *perlop* 及 *perlfunc*。

使用 tr/// 进行转译

虽然 `tr///` 操作符看起来像正则表达式，但它事实上会将某组字符转译成另一组字符。它也可以迅速算出特定字符的出现次数。请参阅在线手册 *perlop*。

Here 文档

Here 文档是一次引用多行字符串的好方法。请参阅在线手册 *perldata*。

数学

Perl 几乎能做你所能想象得到的任何数学计算。

高级数学函数

Perl 内置所有基本的数学函数（平方根、余弦、对数、绝对值等），细节请参考在线手册 *perlfunc*。虽然省略了某些函数（比如正切或以 10 为底的对数），但它们可以用现有的函数轻松组合而成，或是利用简单的模块完成（请参考 `POSIX` 模块，里面有许多常用的数学函数）。

虚数与复数

虽然虚数与复数不属于 Perl 的核心功能，不过可以利用模块来处理。这些模块能重载（overload）一般的操作符与函数，让你在处理复数时还能继续使用 * 做乘法，或使用 sqrt 计算平方根。请参考 Math::Complex 模块。

大数与高精确度数值

如果需要，Perl 可以处理任意大的数字，而且很精确。举例来说，你可以计算 2000 的阶乘，或是计算 π 到小数点之后 10000 位。请参考 Math::BigInt 及 Math::BigFloat 模块。

列表与数组

Perl 里有些功能可以让人们更容易整批处理列表或数组。

map 与 grep

第十七章中我们曾讨论过列表处理操作符 map 和 grep。限于篇幅，我们无法提到它们全部的功能。请参阅 *perlfunc* 在线手册，里面有进一步的信息及范例。也请参考羊驼书以得知更多使用 map 与 grep 的方式。

splice 操作符

利用 splice 操作符，你可以在数组中间插入或移除数据项，以让数组随需要扩大或缩小（这和 substr 对字符串所做的事差不多）。因为 Perl 有此功能，所以基本上不需要链表（linked list）。请参阅在线手册 *perlfunc*。

位与块

vec 操作符可用来处理由位所组成的数组，即位串（*bitstring*）。你可以设定第 123 位的值、清除第 456 位的值或检查第 789 位的值。位串的大小没有上限。vec 操作符的块长度可以设成 2 的乘幂，这在你需要把字符串视为由半字节（nybble）所组成的数组时很有用。请参阅在线手册 *perlfunc*。

格式

Perl 的格式化功能可让你轻易制作出具有自动页首、格式固定的报表。事实上，这是 Larry 当初开发 Perl 的主要原因：作为实用摘录及报告语言。不幸的是，它的功能十分有限。使用格式化最让人心碎的，莫过于发现格式化的功能无法满足日新月异的需求。通常这种情况发生时，程序的输出部分就得从头写起，换成格式化以外的方式。话又说回来了，如果你确定格式化能满足目前及未来的所有需求，它还是个很酷的功能。请参阅在线手册 *perlform*。

网络与进程间通讯

对于系统上所有让程序间彼此通讯的功能，Perl 通常都会支持。本节展示了几种常用的方式。

System V IPC

Perl 支持所有 System V IPC（进程间通讯）的标准函数，包括：消息队列（message queue）、信号量（semaphore）、共享内存（shared memory）等。当然，Perl 里的数组和 C 语言不同，并不是存储在连续的内存块中【注 12】，因此，共享内存无法直接共享 Perl 的数据。不过，有些模块可以帮忙转译数据，让 Perl 的数据看起来像存储在共享内存里。请参阅在线手册 *perlfunc* 与 *perlipc*。

Sockets

Perl 对 TCP/IP socket 提供了完整的支持，所以只使用 Perl 就可以写出 Web 服务器、Web 浏览器、Usenet 新闻服务器或客户端、finger 服务器或客户端、FTP 服务器或客户端、SMTP / POP / SOAP 的服务器或客户端以及 Internet 上所用的任何协议的客户机或服务器。当然，你不需要了解底层技术细节，因为所有常见的协议都已经有模块可用了。举例来说，你只要使用 LWP 模块再加上一两行程序代码【注 13】，就能做出 Web

注 12：事实上，说 Perl 的数组存储在“一块内存”里面通常是不对的。它们几乎一定会分散在许多小块的内存中。

注 13：利用 LWP 虽然轻易就能做出可下载页面或图像的“Web 浏览器”，但是如何实际显示给用户看又是另一个问题了。你可以利用 Tk 或 Gtk 的插件来驱动 X11 进行显示，或使用 Curses 模块在字符终端上绘图。其实最终问题就是选择 CPAN 上哪些合适的模块。

服务器或客户端。如果想引用高质量的程序代码，LWP 模块（事实上，它是一系列互相支持的模块，它已经实现出了 Web 上绝大部分的功能）也是绝佳的对象。至于其他的协议，请用协议名称来搜索相关的模块。

安全性

Perl 提供了不少与安全性有关的强大功能，这让 Perl 程序比相应的 C 程序更加安全。其中最重要的可能就是一般称为污染检查（*taint checking*）的数据流分析。当它被启用时，Perl 会记住哪些数据是来自用户或环境的（因此不能盲目相信）。一般来说，当这些所谓“受污染”（tainted）的数据对其他的进程、文件或目录产生影响时，Perl 会禁止该项操作并中断程序。这并不完美，但却能有效避免安全相关的问题。还有在这里讲不完的细节，请参考在线手册 *perlsec*。

调试

Perl 自带了一个很棒的调试器，它支持断点（breakpoint）、观察点（watchpoint）、单步执行（single-stepping）以及所有命令行调试器该有的功能。它其实是用 Perl 写成的，如果它本身有缺陷，我们也不知道该如何对它进行调试。除了基本的调试器命令之外，你还可以在程序运行到一半的时候从调试器来运行 Perl 程序代码 —— 调用子程序、改变变量甚至是重新定义子程序。最新的信息，请参考在线手册 *perldebug*。羊驼书详细介绍了调试器的使用。

另外一个调试的技巧就是使用 `B::Lint` 模块。这个模块能对可能的情况报警，而这些情况 `-w` 开关不能捕获。

通用网关接口（CGI）

Perl 在 Web 上最普遍的用途就是编写 CGI 程序。这种程序在 Web 服务器上可用来处理表单、进行搜索、产生动态网页或是计算网页的访问次数。

只要使用 Perl 自带的 CGI 模块，就能轻易访问表单参数，也能产生 HTML 作为响应。你也许想要略过该模块，直接复制并粘贴某一段号称可以访问表单参数的程序代码，但是它们几乎都有不少的缺陷【注 14】。不过，在编写 CGI 程序时有许多重要的事情必

注 14：有些接口上的细节是这些小程序不支持的。请相信我们的建议：还是使用模块比较好。

须予以考虑。本书限于篇幅无法详细讨论这些主题【注 15】。

安全第一、安全第一、安全第一

我们再怎么强调安全性都不过分。世界上对计算机的成功攻击里，大概有一半是靠 CGI 程序里的安全漏洞完成的。

并行运算的相关问题

许多进程同时访问单一文件或资源，这在 CGI 程序里是很常见的情况。

符合标准

无论你花多少工夫测试程序，最多大概也只能覆盖 1% 或 2% 的 Web 浏览器和服务器【注 16】。这是因为已经有数以千计的同类软件，而且每几天都会出现新的。解决之道是遵循标准，这样就能在所有的软件上运作【注 17】。

疑难排解与调试

CGI 程序实际的运行环境常与开发时不同，没有办法用一般的方法直接访问，所以你还得学习新的问题排查和调试技巧。

安全第一、安全第一、安全第一!

我们必须再说一次：别忘了安全！当你的程序对全世界的入侵者公开时，任何时候都不能忽略安全性。

上面的列表还不包括 URI 编码、HTML 实体 (entity)、HTTP 与响应码 (respone code)、SSL (Secure Socket Layer)、服务器端包括 (Server-side Includes，简称 SSI)、here 文档、动态产生图像、程序化地产生 HTML 表格、表单及窗口小插件 (widget)、隐藏式表单元素、Cookie 的取得与设定、路径信息 (path info)、捕捉错误、重定向、污染检查、国际化 (internationalization，简称 i18n) 与本地化 (localization)、在 HTML 里内嵌 Perl 程序代码 (或是反过来)、如何调节 Apache 与 `mod_perl`，以及如何使用 `LWP` 模块【注 18】。任何讲 Web 程序设计的 Perl 书都应该包括上述大部分 (或全部)

注 15：不少校阅者在看过本书草稿之后，都希望能再多讲一点 CGI 编程方面的知识。我们也想这样，但如果多讲一点点而省略众多安全细节的话，会给读者带来风险，这总归不太好。而如果要详细展开讨论 CGI 程序设计的相关议题，恐怕本书的篇幅以及成本都要再增加 50%。

注 16：请注意，每种品牌的浏览器在各操作系统上的各个版本都算是一种浏览器，而你无法测试那么多。每次听到别人说网站已经用“两种浏览器”测试过了，或是说“不知道在另一种浏览器上是否能用”，我们都会忍不住笑出声来。

注 17：最起码自己遵循标准的话，就可以同那些不循规蹈矩的程序员划清界限。

注 18：现在你应该了解，为何我们不想将所有内容都塞进这本书里面了吧?

的主题。Scott Guelich 等人的《CGI Programming with Perl》（由 O'Reilly 出版）相当不错；Lincoln Stein 的《Network Programming with Perl》（Addison-Wesley 出版）也值得一读。

命令行选项

Perl 有许多不同的命令行选项，其中有不少选项能让你直接从命令行写出有用的程序。请参阅在线手册 *perlrun*。

内置变量

Perl 有一大堆内置的变量（例如 `@ARGV` 与 `$0`），有些能提供有用的信息或是让人控制 Perl 的运作方式。请参阅 *perlvar* 在线手册。

语法扩展

Perl 的语法里还有更多的技巧，包括 `continue` 块与 `BEGIN` 块。请参阅在线手册 *perlsyn* 及 *perlmod*。

引用

Perl 的引用（reference）跟 C 语言的指针（pointer）差不多，不过工作原理则比较类似 Pascal 或 Ada 里的相应功能。引用会“指向”某个内存位置，但是因为没有指针运算和内存的直接分配、释放功能，所以你可以确定任何引用都是有效的。引用可以用来实现面向对象程序设计、复杂的数据结构以及其他有用的技巧。请参阅在线手册 *perlreftut* 及 *perlref*。羊驼书涵盖了许多引用的重要细节。

复杂的数据结构

引用让我们能够在 Perl 里建立复杂的数据结构。举例来说，Perl 可以做出二维数组【注 19】，或是更有趣的结构，像哈希组成的数组、哈希组成的哈希、哈希的数组组成的哈希【注 20】。请参阅 perldsc（数据结构范例）及 perllol（由列表组成的列表）这

注 19：其实不完全是这样；不过你可以装得非常像，直到自己都不太记得有何差别。

注 20：事实上，这些东西 Perl 都做不到；它们只不过是简称而已，实际的状况并非如此。我们所谓的“数组的数组”，在 Perl 里面其实是“数组引用的数组”。

两个在线手册。羊驼书详细介绍了这部分内容，包括复杂数据的操作技巧，像排序与统计。

面向对象程序设计

没错，Perl 也有对象，并且和其他语言是术语兼容（buzzword-compatible）的。面象对象（object-oriented，简称 OO）程序设计让你能够通过继承（inheritance）、覆盖（overriding）以及动态方法判定（dynamic method lookup）来自行定义功能相关的数据类型【注 21】。Perl 并不强迫你使用对象，这跟某些面向对象语言不同（在 Perl 中，许多面向对象的模块在使用时，并不需要先了解对象是什么）。不过，如果程序代码长度大于 N 行，那么以面向对象的方式设计对程序员而言可能比较有效率（虽然运行时可能会慢一点点）。没有人知道 N 的值确切是多少，不过我们估计它在几千左右。请阅读在线手册 *perlobj* 及 *perlboot* 以作为入门。进一步的权威信息请参考由 Damian Conway 编写的《Object-Oriented Perl》（由 Manning Press 出版）。羊驼书也同样详细介绍了有关对象的内容。

匿名子程序与闭包

乍听之下这也许很奇怪，不过没有名称的子程序也有它的用处。这种子程序可以作为其他子程序的参数，也可以放进数组或哈希里作为 jump table 来使用。闭包则是从 Lisp 世界引入的概念，它的意思（差不多）是具有私有数据的匿名子程序。照例，羊驼书里也提到了它。

捆绑变量

捆绑变量（tied variable）可以用一般的方式来访问，但是它背后使用的是你自己的程序代码。所以，你可以建立存储在远程机器上的标量或某个总能保持排序的数组。请参阅在线手册 *perltie*。

操作符重载

你可以利用 overload 模块来重新定义操作符，包括加法、连接、比较甚至是从字符串到数字的自动转换。例如，实现复数的模块就可以用这种方式来让复数乘以 8 得出正确的复数。

注 21： 面向对象有专门的术语。事实上，任何两种面向对象的术语常常不太通用。

动态加载

动态加载就是让程序在运行时加载某些额外的功能，然后继续运行下去。动态加载 Perl 程序代码从来不是问题，而动态加载二进制扩展模块就更有意思了【注 22】。非 Perl 语言的模块就是这样做出来的。

嵌入

（从某种角度来说）与动态加载相对的技术就是“嵌入”了。

如果你想写个非常酷的文字处理器，假设用 C++ 语言实现【注 23】。现在，你希望用户能够使用 Perl 的正则表达式来执行强大的“查找并替换”功能，于是将一段 Perl 程序嵌入你的程序里。接下来，你也可以将 Perl 的某些功能开放给用户。高级用户可以用 Perl 编写子程序，使之成为菜单里的功能。他们也可以写一小段 Perl 程序来自定义文字处理器的操作。之后，你在网站上留了一个位置，让用户分享并交流这些 Perl 程序片段。这样一来，就突然有上千名程序员在扩展此程序的功能，而你的公司不必为此有额外支出。为了这些好处，你要付钱给 Larry 吗？完全免费，请参考 Perl 所附的授权条款。Larry 实在是个好人，你至少该寄给他一封感谢函吧。

虽然我们还没听说过这种文字处理器，但是有些人已经使用同样的技巧做出功能强大的其他程序了。其中一个例子是 Apache 的 `mod_perl`，它将 Perl 嵌入到功能已经十分强大的 Apache 里面。如果你想要嵌入 Perl，请参考 `mod_perl` 的做法，因为它是完全开源的，所以你可以看到它的实现细节。

把其他语言写就的程序转换成 Perl 程序

如果你有以 *sed* 或 *awk* 语言写成的程序，却希望能用 Perl 做一样的事，那你大可放心。Perl 不但具备这两个语言的所有功能，还附有转换程序，它们大概已经安装在你的系统

注 22：只要系统支持，通常都能以动态加载的方式进行二进制扩展。如果系统不支持，也可以对扩展模块进行静态编译，也就是说 Perl 在编译时即包含了该扩展模块，可以直接使用。

注 23：没准儿用这个语言来写字处理器是很合适的。没错我们喜爱 Perl，但是并没有发毒誓不再用别的语言。如果语言 X 是最佳选择的话，就该使用它。不过 X 常常是 Perl。

上了。请参考 *s2p*（从 *sed* 转换到 Perl）及 *a2p*（从 *awk* 转换到 Perl）的说明文档【注 24】。因为机器写的程序比不上人写的，所以其产生的程序代码并不怎么样，但是它至少是个开始，要进一步修改也很容易。转换后的程序的运行速度也可能有所不同。不过，在修复好程序代码里明显的瓶颈之后，应该和原来的程序差不多。

想在 Perl 里使用 C 语言的算法吗？也请放心，将 C 程序代码编译成 Perl 能使用的模块并不困难。事实上，任何能编译成目标码的程序语言通常也都能被转换成模块。请参考 *perlxs* 在线手册、`Inline` 模块以及 SWIG 系统。

你想将现有的 shell 脚本转换成 Perl 吗？真不幸，并没有从 shell 自动转换到 Perl 的方法。这是因为 shell 本身几乎什么事都不做，它大部分的时间都在运行别的程序。当然，我们可以写个程序来将 shell 里的每一行都转成 `system` 调用，可是这样会比 shell 直接调用慢得多。要将 shell 里所用到的 *cut*、*rm*、*sed*、*awk*、*grep* 转换成有效率的 Perl 程序代码，实在需要人的智力才办得到。将 shell 脚本从头改写是比较好的做法。

把单行 find 命令转换成 Perl 程序

系统管理员的常见任务之一就是递归地搜索目录树来寻找某些项目。在 Unix 上，这通常要靠 *find* 命令来完成。这件事，我们也可以直接用 Perl 来做。

Perl 自带的 *find2perl* 命令所接受的参数和 *find* 命令的相同。不过，它并不会实际进行搜索，而是输出用来进行搜索的 Perl 程序。既然是程序代码，你就可以根据需要自行修改（程序的风格会有点奇怪）。

find2perl 有个很好用的参数是 *find* 里没有的，那就是 `-eval` 选项。它后面的参数是实际的 Perl 程序代码，此程序代码会在每次文件被找到时运行。当它运行时，当前工作目录将会是该文件所在的目录，`$_` 的值则是找到的目录或文件名。

下面是 *find2perl* 的一种用法。假设你是 Unix 系统上的管理员，想要将 */tmp* 目录下陈旧的文件全部删除【注 25】。下面就是产生该程序的方法：

```
$ find2perl /tmp -atime +14 -eval unlink >Perl-program
```

注 24：如果你在使用 *gawk*、*nawk* 或者其他的衍生版本，*a2p* 也许就无法进行转换了。这里所提到的两个转换程序都是在很久之前写成的，除了能保证在新版的 Perl 中运行之外，没人感觉有必要进一步更新。

注 25：这通常是每天清晨 *cron* 定时完成的任务。

该命令会在 */tmp*（以及所有的子目录）里寻找 atime（上次访问时间）离现在至少 14 天的所有项目。它会对每个项目运行 Perl 程序代码 `unlink`，而且会将默认变量 `$_` 的值当成所要删除的文件名。输出结果（重定向到 *Perl-program* 这个文件里）将会是执行上述任务的程序代码。接下来，只需要安排它在适当的时间运行就行了。

提供给程序的命令行选项

如果想让程序处理命令行选项（像 Perl 本身的 `-w` 警告选项），有现成的模块可以提供标准的做法。请参考 `Getopt::Long` 及 `Getopt::Std` 模块的说明文档。

嵌入式说明文档

Perl 本身的说明文档是以 *pod*（也就是 plain-old documentation）写成的。你可以将这种说明文档直接放进程序里，之后可以把它转成纯文本、HTML 或许多其他的格式。请参考在线手册 *perlpod*。羊驼书也介绍了这部分内容。

打开文件句柄的其他方式

打开文件句柄（`filehandle`）时还有许多别的模式可供使用。请参阅在线手册 *perlopentut*。

语系设定（locale）与 Unicode

世界毕竟不大。为了让程序能够在使用不同“字母系统”的地方运作，Perl 支持了语系设定（locale）与 Unicode。

语系设定用来告诉 Perl 要如何以本地的方式操作。举例来说，字符 æ 是否在字母表的最后？或者在在 S 和 Œ 之间呢？三月在当地的说法又是什么呢？请参考在线手册 *perllocale* 的说明（不要和在线手册 *perllocal* 搞混了）。

请参考在线手册 *perlunicode* 以便了解目前的 Perl 版本如何支持 Unicode。在编写本书时，Perl 的每一个新版本都包含了许多与 Unicode 相关的改变，但是希望不久之后就会稳定下来。

线程（thread）与 fork

Perl 现在支持线程了。虽然（在本书编写时）线程还是实验性的功能，但是在某些应用中已经可行了。Perl 对 fork 的支持（在可用的系统上）比较完善。请参阅在线手册 *perlfork* 及 *perlthrtut*。

图形用户界面（GUI）

Tk 是既庞大又具威力的一个模块，能够写出跨平台的图形界面。请参考 Nancy Walsh 与 Steve Lidie 合著的《Mastering Perl/Tk》（O'Reilly 出版）。

更多……

你只要看看 CPAN 上的模块列表，就可以找到更多不同用途的模块：从产生图表及图像，到下载电子邮件；从计算贷款分期付款，到预测日落时间。新的模块不断出现，所以你现在看到的 Perl 又比我们写作本书时要强大多了。我们不可能一直跟上新的模块，所以就到此为止。

由于 Perl 的世界不断扩张，所以连 Larry 也自认无法跟上 Perl 所有的发展。他并不会对 Perl 感到无聊，他总是能够在这个扩张的世界里找到新的角落。本书的作者大概也有同感。Larry，谢谢你！

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

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

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

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

作者简介

Randal L. Schwartz 一直以来都是Perl畅销书的作者，他自认为很幸运，能成为两本学习Perl的基础书籍的作者之一。除了《Perl语言编程》与《Perl语言入门》以外，Randal也是《Unix Review》、《Web Techniques》、《Sys Admin》与《Linux Magazine》等杂志的Perl专栏作家。

Tom Phoenix 自1982年起就投身教育领域。13年来他在科学博物馆工作时多半与解剖、爆炸为伍，工作范围从可爱动物到高压电都有。在此之后，他展开了在Stonehenge Consulting Service这家顾问公司里的Perl教学生涯，从1996年开始至今。他也不断地走访各地，所以说不定你马上就会在Perl推广组的聚会中见到他。只要一有时间他就会回答*comp.lang.perl.misc*与*comp.lang.perl.moderated*等新闻组上的问题，并投身于Perl的发展，使其更加有用。不但工作方面与Perl有关，Tom这位黑客除了关于Perl的工作之外还是个业余的密码学家，而且还会西班牙文。他目前定居在美国俄勒冈州波特兰市。

brian d foy 从1998年起成为Stonehenge Consulting Service的讲师。自他在大学研究所里念物理时就是Perl用户，并且从他拥有第一台计算机开始就是忠诚的Mac计算机用户。他创办了第一个Perl社群，也就是纽约Perl推广组，而“Perl推广组”这个推广Perl的非营利组织在其协助之下在全球设立了200多个Perl社群。他还负责维护*perlfaq*这份Perl核心文档、若干CPAN模块以及一些独立的小程序。他是The Perl Review这份致力于Perl的杂志的发行人，同时也是大小Perl集会的常客，像Perl Conference、Perl University、MarcusEvans BioInformatics'02以及YAPC。在The O'Reilly Network、*The Perl Journal*、*Dr. Dobbs*、*The Perl Review*、*use.perl.org*以及许多Perl新闻组上都可以看到他的文章。

封面介绍

《Perl语言入门》第五版的封面动物是骆马（Lama glama）。它是骆驼（camel）的同类，原生于安地诺（Andean）山脉附近。骆马类族群里还包括可养驯的羊驼（alpaca），以及它野生的祖先原驼（guanaco）和小羊驼（vicuna）。在远古人类栖息

地找到的骨骸显示羊驼和骆马早在4500年前就被驯化了。1531年，当西班牙征服者超过了位于安第斯高地（high Andes）的印加帝国时，发现了大群的这两种动物。骆马适合高山生活，它们的血色素可以携带比其他哺乳动物更多的氧气。

驼马最高重达300磅（约合136千克），通常作为驮兽使用。驮运货物的队伍可能由数百只动物组成，每天最多可以前进20英里（约合32千米）。骆马可以驮背50磅（约合23千克）以内的重物，但是脾气通常不好，而且会以吐口水和咬人来表达不满。对安第斯的居民来说，骆马也是食用肉、织毛、兽皮及燃油的来源。它们的毛能编成绳子和毛毯，干燥后的粪便则可以作为燃料使用。

封面图片使用的是19世纪Dover Pictorial Archive的雕版。