

Intermediate Perl

第2版
涵盖Perl 5.14

Perl进阶



Randal L. Schwartz,
[美] brian d foy, Tom Phoenix 著
韩雷 译

O'REILLY®



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

Perl进阶

本书是*Learning Perl*一书的进阶。学完本书之后，您可以使用Perl语言的特性编写从简单脚本到大型程序在内的所有程序，正是Perl语言的这些特性使其成为通用的编程语言。本书为读者深入介绍了模块、复杂的数据结构以及面向对象编程等知识。

本书每章的篇幅都短小精悍，读者可以在一到两个小时内读完。每章末尾的练习有助于您巩固在本章所学的知识。如果您已掌握了*Learning Perl*中的内容并渴望能更进一步，本书将向您讲授Perl语言的绝大多数核心概念，以便在任何平台上编写出健壮的程序。

本书主题包括：

- 包和命名空间；
- 引用和作用域，包括正则表达式的引用；
- 操作复杂的数据结构；
- 面向对象编程；
- 编写和使用模块；
- 测试Perl代码；
- 为CPAN做出贡献。

与*Learning Perl*一样，本书中的内容紧紧围绕作者自1991年以来所讲授的颇受欢迎的Perl入门课程展开。本书涵盖了Perl语言直至5.14版本以来的最新修订。

Randal L. Schwartz精通软件设计、系统管理、安全、技术写作及培训。除了本书之外，他还参与合著了多本书籍，包括*Learning Perl*、*Programming Perl*以及*Mastering Perl*（均为O'Reilly公司出版）。

brain d foy是一位多产的Perl教师和作家，运营着The Perl Review网站以帮助人们使用并理解Perl。他是*Learning Perl*、*Mastering Perl*以及*Effective Perl Programming*的合著者。

Tom Phoenix在Stonehenge咨询服务公司教授Perl课程，并在Usenet的comp.lang.perl.misc和comp.lang.perl.moderated新闻组中答疑解惑。他还为Perl做出了巨大贡献。



O'REILLY
oreilly.com.cn

封面设计：Karen Montgomery，张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale only in the territory of People's Republic of China

(excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/程序设计/Perl

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-40206-6



9 787115 402066 >

ISBN 978-7-115-40206-6

定价：69.00 元

O'REILLY®

Perl 进阶 (第2版)

[美] Randal L. Schwartz
brian d foy 著
Tom Phoenix

韩雷译

人民邮电出版社

北京

图书在版编目（C I P）数据

Perl进阶：第2版 / (美) 施瓦茨 (Schwartz, R. L.),
(美) 福瓦 (Foy, B. D.) , (美) 菲尼克斯 (Phoenix, T.)
著；韩雷译。— 北京：人民邮电出版社，2015.10
ISBN 978-7-115-40206-6

I. ①P… II. ①施… ②福… ③菲… ④韩… III. ①
Perl语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2015)第210563号

版权声明

Copyright © 2012 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2015. Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

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

本书中文简体字版由 O'Reilly Media, Inc. 授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

◆ 著 [美] Randal L. Schwartz brian d foy Tom Phoenix
译 韩雷
责任编辑 傅道坤
责任印制 张佳莹 焦志炜
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
三河市海波印务有限公司印刷
◆ 开本：787×1000 1/16
印张：22.25
字数：452 千字 2015 年 10 月第 1 版
印数：1—2 500 册 2015 年 10 月河北第 1 次印刷
著作权合同登记号 图字：01-2011-7482 号

定价：69.00 元

读者服务热线：(010) 81055410 印装质量热线：(010) 81055316

反盗版热线：(010) 81055315

內容提要

Perl 是一种功能强大的通用编程语言，享有“一种拥有各种语言功能的梦幻脚本语言”、“UNIX 中的王牌工具”等美誉，受到了国内程序员和系统管理员的青睐。

本书作为 *Learning Perl* 一书的进阶，主要讲解了如何更加有效地利用 Perl 进行开发。本书总共分为 21 章，每章内容篇幅不大，主要内容包括 Perl 简介、使用模块、中级操作基础、引用简介、引用和作用域、操作复杂的数据结构、对子例程的引用、文件句柄引用、正则表达式引用、构建更大型的程序、创建自己的 Perl 发行版、对象简介、测试简介、带数据的对象、Exporter 模块、对象析构、Moose 简介、高级测试、为 CPAN 贡献代码等知识。

本书适合具有一定 Perl 基础的程序员和系统管理员阅读。对于高级 Perl 程序员来讲，本书也是必备的技术参考读物。

O'Reilly Media, Inc.介绍

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

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

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

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

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

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

——Linux Journal

作者序

I've had the pleasure of talking to much of the Perl community through my books, the talks I give, and the other things I write. However, China is a huge part of the world that I don't interact with. I've tried to speak a little Chinese, but only enough so that everyone smiles and laughs when I get the tones wrong and make a nonsense statement.

The translation of Learning Perl for the Chinese programmers greatly pleases me, although I worry that the ancient American television show we use in the examples won't make any sense. Not even the American students in my classes in the United States know about Gilligan or the Skipper from Gilligan's Island, so maybe it's not that important. Just know that Gilligan is the fool who constantly and consistently disappoints his best friend, the Skipper.

I'm always happy to hear from readers and what they've done with something I've been a part of. If you'd like to send me a message at brian.d.foy@gmail.com, we can both see how well Google Translate works. Or maybe you can send it as a Perl program, which I hope we can both understand once you finish this book.

Since the Chinese market is new to me, I'm sure there's all sorts of programming wisdom that's missing from the Perl community I travel in. Please tell me about those too.

Good luck.

brian d foy

作者序（中文版）

我非常高兴能够有机会和诸多 Perl 社区讨论我的这本书、演讲和其他我所编写的内容。中国是一个世界大国，不过我还没有亲身接触过。我曾经尝试说一点中文，但是每当我发错音或者说了错误的句子后，大家都只是对我报以“嘲弄般”的微笑。

Learning Perl 一书的翻译对于中国读者的帮助已经很让我感到欣慰，尽管我还有些担心我们在本书示例中使用的这个古老的美国电视节目会让人感到困惑，甚至很多美国本土的学生都不知道 Gilligan's Island 的 Gilligan 或者 Skipper。也许这并不重要，只需要知道 Gilligan 是那个一直并且不断地让他最好的朋友 Skipper 失望的“蠢货”就够了。

我一直很高兴收到来自读者的消息，以及有幸能够参与读者所完成的项目。如果你想要向 brian.d.foy@gmail.com 发送任何消息，我们都将会看到 Google 翻译的效果，或者你能够以一个 Perl 程序的方式发送给我，我希望在你读完本书之后，我们都能理解程序中的内容。

中国市场于我而言是全新的市场，我确信在访问 Perl 社区时，有各种类型的编程智慧被我错过了，如有机会也请向我分享这些编程智慧。

好运！

brian d foy

译者序

Perl 是一门强大的通用编程语言，凭借其简单、灵活、可移植性强等特点，受到了国内程序员和系统管理员的青睐。因为 Perl 是解释执行，这样就极大地节省了编译的时间。Perl 也内置了很多数据结构，如散列和列表，使用起来极其方便，而且配合 Data::Dumper 这个标准库，我们可以很方便地调试复杂的数据结构。此外，Perl 可以在几乎所有的操作系统上运行，甚至有很多是你没有听过的系统。

我们假定您已经阅读过了 *Learning Perl* 这本 Perl 入门书，如果你不满足于对 Perl 的粗浅理解，想要更为深入、详细、系统地学习 Perl，本书就是为您准备的！本书是 Perl 语言学习的进阶图书（从其英文书名 *Intermediate Perl* 上也可以看得出来），当前为第 2 版。需要说明的是，本书第 1 版的书名与第 2 版不同，它的名字是 *Learning Perl Objects, References, and Modules*，当时国内并没有出版社引进。

前面提到，本书是 Perl 的进阶图书，学习完本书之后，读者也可以朝着 *Mastering Perl* 一书迈进，从而彻底掌握 Perl 语言。总之，本书的目标就是让读者能够熟练地掌握 Perl 语言中的引用、数据结构、面向对象、编写测试、构建模块及通过 CPAN 安装模块等内容，以及流行的面向对象框架 Moose 和一些 Perl 的常见习语等。

感谢人民邮电出版社引进本书，这对国内广大 Perl 爱好者来讲是一大福音，可以帮助越来越多的人掌握这门“古老”而又不断发展的现代编程语言。感谢我的妻子 Vivi，因为承接了本书的翻译工作，牺牲了很多在一起的时光，谢谢你的理解与支持。本书中文版的顺利付梓有你一份功劳。

最后，尽管本书篇幅不大，但是限于本人专业水平、翻译能力有限，对于书中内容的把握难免有不准确之处，还请读者批评指教。

韩雷

2015 年 2 月 22 日夜

序

Perl 的面向对象机制是经典的“戏法”。它使用 Perl 已有的一系列非面向对象特性，例如包、引用、散列、数组、子例程和模块，然后，虽然没有暗自使用其他特性，但还是设法构造出功能完整的对象、类和方法。看起来就像是突如其来生成的一样。

这是一个非常奇妙的技巧。这意味着你能够基于你已有的 Perl 知识构建，并且按照你自己的方法很轻易地进入面向对象的 Perl 开发，而不必先去征服新语法的“大山”或者探索新技巧的“海洋”。这也意味着你能够通过从已有的 Perl 面向对象结构中逐步选取合适的结构来优化调整面向对象的 Perl，以满足你自己的需要。

但是还有一个问题：因为 Perl 运用包、引用、散列、数组、子例程和模块作为它的面向对象机制的基础，也就是说，如果想要使用面向对象的 Perl，你就需要理解包、引用、散列、数组、子例程和模块这些内容。

然后还有一个小问题：学习曲线不可能消除，它只是倒退几步。

也就是说，我们到底需要掌握多少关于非面向对象的 Perl 知识，才能够开始学习关于 Perl 的所有面向对象知识呢？

本书给出了以上所有问题的答案。在后续内容中，Randal 通过近 20 年来使用 Perl 的经历，以及近 40 年观看电视剧 *Gilligan's Island* 和 *Mr. Ed* 中的情节，来解释 Perl 语言中每一个面向对象特性的基础。而且更好的是，我们将会确切地展示如何组合这些组件，以创建更有用的类和对象。

因此，如果你在使用 Perl 的对象、引用和模块时感觉自己像 Gilligan 一样，那么本书就是剧中教授会推荐给你的。

以上消息来源可靠。

——Damian Conway，2003 年 5 月

前言

大约在 20 年前（几乎与互联网的历史一样长），Randal Schwartz 编写了 *Learning Perl* 第 1 版。在接下来的这些年里，Perl 本身基本上从一种很酷并且主要由 UNIX 系统管理员使用的脚本语言，成长为一门健壮的面向对象编程语言，而且能在所有人已知的操作系统上运行，甚至包括一些并不广为人知的系统。

在 *Learning Perl* 的前后 6 个版本中，它保持着基本一致的片幅，大概 300 页，而且一直包含几乎相同的一些内容，以保持结构紧凑，很适合初级程序员学习。但是，还有很多关于 Perl 的内容需要学习。

Randal 将英文原书第一版命名为 *Learning Perl Object, References, and Modules*，后来我们将英文原书重命名为 *Intermediate Perl*，但是我们认为也可以叫做 *Learning More Perl*。这是一本介绍 *Learning Perl* 中所没有涵盖的内容的书。我们将向你展示使用 Perl 语言编写更大型程序的方法。

在 *Learning Perl* 中，我们将每一章都设计得非常短，以至于你只需要花费大概 1 小时的时间来阅读。每一章都以一系列练习结束，这些练习可以帮助读者巩固所学的知识，而且这些练习的答案可以参考书末的附录。与 *Learning Perl* 一样，我们也为本书开发了很多素材，可以将这些素材用于教学环境。

除非我们特别进行了标记，否则本书介绍的全部内容能够适用于任何平台，无论是 UNIX、Linux 还是 Windows，无论是来自 ActiveState 的 ActivePerl，还是 Strawberry Perl，或者是 Perl 的其他现代实现。要想更好地使用本书，你只需要熟悉 *Learning Perl* 中的内容，而且有志于进一步地学习 Perl。

在你读完本书之后，你会了解你所需的 Perl 语言的大多数核心概念。这个系列的下一本是 *Mastering Perl*，该书将专注于应用你所了解的内容，来编写更高效、更健壮的 Perl 应用，以及管理 Perl 软件开发的生命周期。

在你 Perl 职业生涯的任何时候，你都应当拥有 *Programming Perl* 这本书，该书是（几乎是）Perl 语言最权威的“圣经”。

本书结构

本书分为 3 个部分。第一部分介绍如何处理引用，这也是复杂数据结构以及 Perl 面向对象编程的关键所在；第二部分介绍对象以及 Perl 如何实现面向对象编程；第三部分

将介绍如何处理 Perl 的模块结构、测试，以及用于发布代码的社区基础架构。

你最好从头开始阅读本书，适时停下完成每一章的练习。每一章都基于之前的章节，而且我们也将假定在介绍新主题时，你已经了解了本书之前章节中的内容。

第 1 章是关于全部内容的简单介绍。

第 2 章介绍如何使用 Perl 的核心模块和其他人编写的第三方模块。后续章节将展示如何创建我们自己的模块，但在此之前，你仍然可以使用已有的模块。

第 3 章选择性地介绍一些中级 Perl 技巧，你会在本书后续部分用到这些技巧。

第 4 章介绍一定程度的重定向，允许相同的代码操作不同的数据集。

第 5 章讲述 Perl 如何跟踪数据指针，并且引入匿名数据结构和自动带入。

第 6 章讨论创建、访问以及输出任意深度和嵌套的数据结构，包括数组的数组和散列的散列。

第 7 章介绍如何以动态创建并且后续执行的匿名子例程的方式捕获行为。

第 8 章讲述如何以标量形式存储文件句柄，可以很容易地在程序间传递或者在数据结构中存储。

第 9 章介绍如何编译正则表达式而不必立即使用它们，随后将它们作为构建块用于更大的模式之中。

第 10 章介绍关于排序的复杂操作、施瓦茨变换和处理递归定义的数据。

第 11 章讨论如何通过将代码分散在不同的文件和命名空间中来构建更大的程序。

第 12 章讲解如何创建一个 Perl 发行版作为你面向对象编程的第一步。

第 13 章介绍类、方法调用、继承和重载。

第 14 章讲述如何开始测试你的模块，这样就会发现你所创建的发行版中的代码问题。

第 15 章讨论如何为每个实例添加数据，包括构造函数、getter 和 setter。

第 16 章使用多重继承、自动方法以及文件句柄的引用。

第 17 章讲述 use 的工作方式，我们如何决定从模块中导出的内容，以及如何创建自己的导入例程。

第 18 章介绍如何向已销毁的对象中添加行为，包括对象持久性。

第 19 章介绍 Moose，Moose 是 CPAN 上的一个对象框架。

第 20 章介绍高级测试，测试代码以及元代码复杂的一面，诸如文档以及测试覆盖率。

第 21 章介绍如何通过将你的代码上传到 CPAN 上来与全世界分享你的代码。

附录提供所有练习的答案。

本书的惯例



提示

这个图标用来强调一个提示、建议或一般说明。



警告

这个图标用来说明一个警告或注意事项。

代码示例的使用

本书的目的是为了帮助读者完成工作。一般而言，你可以在你的程序和文档中使用本书中的代码，而且也没有必要取得我们的许可。但是，如果你要复制的是核心代码，则需要和我们取得联系。例如，你可以在无须获取我们许可的情况下，在程序中使用本书中的多个代码块。但是，销售或分发 O'Reilly 图书中的代码光盘则需要取得我们的许可。通过引用本书中的示例代码来回答问题时，不需要事先获得我们的许可。但是，如果你的产品文档中融合了本书中的大量示例代码，则需要取得我们的许可。

在引用本书中的代码示例时，如果能列出本书的出处信息最好不过。出处信息通常包括书名、作者、出版社和 ISBN。例如：“*Intermediate Perl* by Randal L. Schwartz, brian d foy, and Tom Phoenix. Copyright 2012 Randal L. Schwartz, brain d foy, and Tom Phoenix, 978-1-449-39309-0.”

在使用书中的代码时，如果不确定是否属于正常使用，或是否超出了我们的许可，请通过 permissions@oreilly.com 与我们联系。

联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社。

美国：

O'Reilly Media Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们还为本书建立了一个网页，其中包含了勘误表、示例和其他额外的信息。你可以通过如下地址访问该网页：

<http://www.oreilly.com/catalog/9781449393090>

关于本书的技术性问题或建议，请发邮件到：

bookquestions@oreilly.com

欢迎登录我们的网站 (<http://www.oreilly.com>)，查看我们的更多书籍、课程、会议和最新动态等信息。

Facebook: <http://facebook.com/oreilly>

Twitter: <http://twitter.com/oreillymedia>

YouTube: <http://www.youtube.com/oreillymedia>

Safari® 在线图书

Safari 在线图书 (www.safaribooksonline.com) 是一个按需订阅的数字图书馆。它通过图书与视频形式分发来自技术领域和商务领域的世界级作者的专家级内容。

技术专家、软件开发人员、Web 设计师以及商业和创意人士都使用 Safari 在线图书作为研究、解决问题、学习和认证培训的首选资源。

Safari 在线图书还为组织、政府机构和个人提供了一系列产品组合和计费项目。用户可以通过完全可搜索的数据库来访问数千本图书、培训视频以及出版前的手稿，这些内容都来自诸如 O'Reilly、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等出版社。

致谢

来自 Randal

在 *Learning Perl* 第 1 版的前言中，我感谢了 Beaverton McMenamin 的 Cedar Hills 酒吧^{注1}，

注 1: <http://www.mcmenamins.com/>.

在这个距离我家不远的“免费办公室隔间”里面，我在我的 Powerbook 140 上编写了该书的大部分初稿。就像我最喜欢的球队会在季后赛里面每天都穿上“幸运袜”一样，我在同一个酒馆里面编写了本书的全部内容（包括这段文字），希望第一本书的幸运之光将照耀我两次（当我在更新本书第 2 版的前言时，我发现我的幸运袜确实有效果）。

这家 McM 酒吧也有同样棒的精酿啤酒和油腻的三明治，但是没有我喜欢的披萨和面包，取而代之的是黑莓脆皮馅饼（当地一种特产）和辛辣的什锦饭（酒吧加了两个摊位，放了一些合伙经营的餐桌）。此时我没有再使用 Powerbook 140，而是使用 Titanium Powerbook，相对前者来说，后者有 1000 倍的磁盘容量、500 倍的内存容量和快 200 倍的 CPU，运行在真正基于 UNIX 的操作系统（OS X）上而不是限制版的 MacOS 上。我也通过我的 144kbit/s 的手机调制解调器更新了全部章节的初稿（包括这部分），然后直接发送电子邮件给审校人员，而不必等到回家用我自己的 9600 波特的外置调制解调器和电话线。时代真是变了啊。

然后，再次感谢 McMenamin 的 Cedar Hills 酒吧的工作人员提供的摊位和热情服务。

与 *Learning Perl* 之前的版本一样，我也感激在这里所说的一切，以及向 Stonehenge 咨询服务公司的同学表示感谢，当我过度使用“huh?” 时，他们给我及时并且准确的反馈（通过眨眼睛和尴尬地组织问题的方式）。通过数十场演讲的反馈，我能够持续精炼和重构本书中的内容。

说到这里，本书中的内容来自于 Silicon Graphics 公司 Margie Levine 的一个为时半天的“What’s new in Perl 5?” 的演讲摘要，此外，作为我经常当场讲授的一个 4 天的“大羊驼”课程（当时的主要目标是第 4 版的 Perl）的补充。最终，我有一个想法就是增加这些记录，使其变为一门完整的课程，并招募 Stonehenge 公司的讲师 Joseph Hall 完成这个工作（他从绘制的示例中选择完整形式）。Joseph 为 Stonehenge 公司开发了一个为期两天的课程，与他自己卓越的 *Effective Perl Programming* 图书（Addison-Wesley Professional）配套，我们也使用这本书作为我们的课程教材（直到今天）。

Stonehenge 公司的其他讲师在过去一些年中也涉猎了“包、引用、对象和模块”的课程，包括 Chip Salzenberg 和 Tad McClellan。但是，最近的大量修改是由我的高级讲师 Tom Phoenix 完成的，他经常荣获“Stonehenge 公司月度最佳员工”，可能我要为此放弃我最喜欢的停车位。

在我编写本书的过程中，Tom Phoenix 贡献了本书中的大部分练习和一套及时的复习笔记，包括为我提供完整的段落，以替换我已经写好的“糊涂话”。不论在课堂上，还是在编写过程中，我们合作得都非常好，因为他的这些贡献，我们都认为 Tom 是作为一个合著者参与的，但是，本书任何部分的错误和疏漏之处都归因于我，因为那些都不是 Tom 的错。

最后一点也很重要，要特别感谢 brian d foy，他带领本书进入了第 2 版，而且编写了上

一版和这一版之间大部分的修改内容。

一本书如果没有主题和发行渠道，就什么都不是，为此我必须感谢与 Larry Wall 和 Tim O'Reilly 的长期合作。谢谢你们，创造了一个行业，在近 20 年的时间里面，我可以从中收益支付我所有的生活必需品、所有想要购买的东西和梦想。

一如既往，特别感谢 Lyle 和 Jack 教会我所需的写作知识，说服我不仅要做一个程序员，还应该学习如何写作；我也是个作家。只是碰巧知道如何编程而已。谢谢！

对于你，本书的读者，为此我辛苦了无数个小时准备，啜饮着冰镇啤酒，狼吞虎咽地吃着美味的芝士蛋糕，小心翼翼地不让蛋糕碎屑掉落在笔记本电脑的键盘上：感谢你们阅读本书。衷心希望本书能够对于你精通 Perl 有所帮助（哪怕只是少许帮助）。如果你在街上遇见我，请告诉我：Hi，我喜欢你的这本书^{#2}。我很喜欢这样。非常感谢！

来自 brian

首先，我必须感谢 Randal，因为我从 *Learning Perl* 第 1 版开始学习 Perl，然后通过为 Stonehenge 咨询服务公司讲授“美洲驼”和“羊驼”课程，学习 Perl 的其余部分内容。讲课通常就是最好的学习方式。

最需要感谢 Perl 社区，人们组织了丰富多彩和多种多样的小组，使我们引以快乐地使用这门语言来编写工具、网站和使 Perl 更有用的模块。很多人通过我在其他工作过程中与他们的讨论，也为本书提供了间接的贡献。有太多人了，但是如果你也曾经与我一起做过一些与 Perl 相关的东西，可能你也为本书做过一些贡献。

来自 Tom

首先感谢 O'Reilly 的整个团队帮助我们完成这本书。

感谢我在 Stonehenge 咨询服务公司共事多年的同事和同学，以及在 Usenet 上帮助过我的人。你们的想法和建议极大地改进了本书中的内容。

真诚地感谢合著者 Randal，给我很大的自由，让我以不同的方式开发教学素材。

感谢我的妻子 Jenna Padbury，感谢你为我处理了很多善后工作。

来自我们 3 位作者

感谢审校人员为本书的初稿提供建议。Tom Christansen 做了很多令人惊叹的工作，不但修正了他所发现的每一个技术问题，而且润色了本书的内容。本书在他的帮助之下变得更完善。David Golden，PAUSE 管理团队的一位成员和 CPAN 工具链的黑客，在模块发布流程的部分帮助我们校正了很多细节问题。Moose 的一些开发者，包括 Stevan

注 2：同时可以询问一个 Perl 问题，我不介意。

Little、Curtis “Ovid” Poe 和 Jesse Luehrs，也对于相关章节提供了帮助。Sawyer X，Module::Starter 模块当前的维护人员，在我们编写本书的相关章节时，也极大地帮助了我们。

也感谢我们的学生，是你们多年来的反馈使我们知道哪些部分的课程内容需要改进。也正是因为你们的帮助，我们所有人才能为本书感到自豪。

感谢诸多的 Perl Monger，当我们访问你们所在的城市时，让我们感到很轻松自在。希望有机会我们再次一起相聚。

最后，我们最由衷地感谢我们的朋友 Larry Wall，正是因为你的智慧，向全世界分享你的这个真正酷并且强大的工具，使我们的工作变得更快、更简单而且更有趣。

目录

第 1 章 简介	1
1.1 背景知识	2
1.2 strict 和 warnings	2
1.3 Perl v5.14	3
1.4 关于这些脚注	4
1.5 关于后续的练习	4
1.6 获取帮助的方式	5
1.7 如果是一个 Perl 课程讲师	5
1.8 练习	6
第 2 章 使用模块	7
2.1 标准发行版	7
2.2 探讨 CPAN	8
2.3 使用模块	9
2.4 功能接口	10
2.5 面向对象的接口	11
2.5.1 一个更典型的面向对象模块: Math::BigInt	12
2.5.2 更佳的模块输出	13
2.6 核心模块的内容	14
2.7 Perl 综合典藏网	15
2.8 通过 CPAN 安装模块	16
2.8.1 CPANminus	16
2.8.2 手动安装模块	17
2.9 适时设定路径	18
2.10 在程序外部设定路径	20
2.10.1 使用 PERL5LIB 扩展@INC	20
2.10.2 在命令行扩展@INC 目录	21
2.11 local::lib	21
2.12 练习	23
第 3 章 中级基础	24
3.1 列表操作符	24
3.1.1 使用 grep 表达式过滤列表	25
3.1.2 使用 map 转换列表	27

3.2 使用 eval 捕获错误	28
3.3 用 eval 语句块动态编译代码	30
3.4 使用 do 语句块	31
3.5 练习	32
第 4 章 引用简介	33
4.1 在多个数组上完成相同任务	33
4.2 PeGS: Perl 图形结构	35
4.3 对数组取引用	36
4.4 对数组引用进行解引用操作	38
4.5 去除大括号	40
4.6 修改数组	40
4.7 嵌套的数据结构	41
4.8 用箭头简化嵌套元素的引用	43
4.9 散列的引用	44
4.10 检查引用类型	47
4.11 练习	49
第 5 章 引用和作用域	51
5.1 关于数据引用的更多信息	51
5.2 如果它曾是变量名将会怎样	52
5.3 引用计数和嵌套数据结构	53
5.4 当引用计数出现问题时	55
5.5 直接创建匿名数组	57
5.6 创建匿名散列	59
5.7 自动带入	61
5.8 自动带入和散列	63
5.9 练习	65
第 6 章 操作复杂的数据结构	67
6.1 使用调试器查看复杂的数据	67
6.2 使用 Data::Dumper 模块查看复杂的数据	71
6.3 数据编组	74
6.3.1 使用 Storable 模块对复杂数据排序	75
6.3.2 YAML 模块	80
6.3.3 JSON 模块	81
6.4 使用 map 和 grep 操作符	81
6.5 应用一点间接方法	81
6.6 选择和改变复杂数据	83
6.7 练习	84

第 7 章 对子例程的引用	86
7.1 对命名子例程的引用	86
7.2 匿名子例程	90
7.3 回调	92
7.4 闭包	93
7.5 从一个子例程返回另一个子例程	94
7.6 作为输入参数的闭包变量	97
7.7 闭包变量作为静态局部变量	98
7.8 查询我们自己的身份	101
7.8.1 令人着迷的子例程	102
7.8.2 转储闭包	105
7.9 练习	105
第 8 章 文件句柄引用	107
8.1 旧方法	107
8.2 改进的方法	108
8.3 指向字符串的文件句柄	110
8.4 文件句柄集合	111
8.5 IO::Handle 模块和其他相应的模块	112
8.5.1 IO::File 模块	113
8.5.2 IO::Scalar 模块	114
8.5.3 IO::Tee 模块	115
8.5.4 IO::Pipe 模块	116
8.5.5 IO::Null 模块和 IO::Interactive 模块	117
8.6 目录句柄	117
8.7 练习	118
第 9 章 正则表达式引用	120
9.1 正则表达式引用之前	120
9.2 预编译模式	122
9.2.1 正则表达式选项	123
9.2.2 应用正则表达式引用	123
9.3 作为标量的正则表达式	124
9.4 建立正则表达式	126
9.5 创建正则表达式的模块	128
9.5.1 使用常见的模式	128
9.5.2 组装正则表达式	129

9.6 练习	130
第 10 章 实用的引用技巧	132
10.1 更佳的输出	132
10.2 用索引排序	134
10.3 更为高效的排序	135
10.4 施瓦茨变换	136
10.5 使用施瓦茨变换实现多级排序	137
10.6 递归定义的数据	138
10.7 构建递归定义的数据	139
10.8 显示递归定义的数据	142
10.9 避免递归	143
10.10 练习	146
第 11 章 构建更大型的程序	148
11.1 修改通用代码	148
11.2 使用 eval 插入代码	149
11.3 使用 do 语句	150
11.4 使用 require 语句	151
11.5 命名空间冲突的问题	153
11.6 使用包作为命名空间分隔符	154
11.7 Package 指令的作用域	156
11.8 包和专门词汇	157
11.9 练习	159
第 12 章 创建你自己的发行版	160
12.1 Perl 模块的两个构建系统	160
12.1.1 在 Makefile.PL 内部	161
12.1.2 在 Build.PL 文件内部	162
12.2 我们的第一个发行版	163
12.2.1 h2xs 工具	163
12.2.2 Module::Starter 模块	164
12.2.3 定制模版	165
12.3 在你的发行版内部	165
12.3.1 META 文件	167
12.3.2 添加额外的模块	168
12.4 模块内部	169
12.5 老式文档	171

12.5.1	段落的 Pod 命令.....	172
12.5.2	Pod 段落	172
12.5.3	Pod 格式标记	173
12.5.4	检查 Pod 格式	174
12.6	模块中的代码	174
12.7	模块构建的总结	175
12.7.1	创建基于 Module::Build 模块的发行版	175
12.7.2	创建 ExtUtils::Makemaker 发行版	176
12.8	练习	176
第 13 章	对象简介.....	177
13.1	如果我们可以和动物对话	177
13.2	介绍方法的调用箭头	179
13.3	方法调用的额外参数	180
13.4	调用第二个方法进一步简化	181
13.5	关于@ISA 的几个注意事项	183
13.6	方法重写	184
13.7	开始从不同的地方查找	186
13.8	使用 SUPER 的实现方法	186
13.9	要对 @_ 做些什么	187
13.10	我们在哪里	187
13.11	牧场总结	187
13.12	练习	189
第 14 章	测试简介.....	190
14.1	为什么需要测试	190
14.2	Perl 的测试流程.....	191
14.3	测试的艺术	193
14.4	测试用具	195
14.5	标准测试	195
14.5.1	模块编译的检查.....	196
14.5.2	模板测试.....	198
14.5.3	测试 Pod	200
14.6	添加第一个测试	201
14.7	测量测试覆盖率	204
14.7.1	子例程覆盖率	205
14.7.2	语句覆盖率	205
14.7.3	分支覆盖率	205
14.7.4	条件覆盖率	206

14.8	练习	206
第 15 章	带数据的对象	208
15.1	马属于马类，各从其类是吗	208
15.2	调用实例方法	210
15.3	访问实例数据	211
15.4	如何构建 Horse 的实例	211
15.5	继承构造函数	212
15.6	编写能够使用类或实例作为参数的方法	213
15.7	为方法添加参数	213
15.8	更有趣的实例	214
15.9	一匹不同颜色的马	215
15.10	收回存款	216
15.11	不要查看“盒子”里面的内容	217
15.12	更快的 setter 和 getter	218
15.13	getter 作为双倍的 setter	219
15.14	仅仅限制一个类方法或者实例方法	219
15.15	练习	220
第 16 章	一些高级对象主题	221
16.1	通用方法	221
16.2	为了更好的行为而测试对象	222
16.3	最后的手段	224
16.4	使用 AUTOLOAD 创建访问器	225
16.5	更容易地创建 getter 和 setter	226
16.6	多重继承	228
16.7	练习	229
第 17 章	Exporter	230
17.1	use 语句在做什么	230
17.2	使用 Exporter 模块导入子例程	231
17.3	@EXPORT 和 @EXPORT_OK	232
17.4	使用 %EXPORT_TAGS 分组	233
17.5	定制导入例程	234
17.6	练习	236
第 18 章	对象析构	237
18.1	清理	237
18.2	嵌套对象析构	239
18.3	终结一个“死去”的 Horse 类	242
18.4	间接对象表示法	243

18.5	子类中的额外实例	245
18.6	使用类变量	246
18.7	削弱参数	248
18.8	练习	250
第 19 章	Moose 简介	251
19.1	用 Moose 模块创建之前的 Animal 模块	251
19.1.1	使用“角色”替换“继承”	254
19.1.2	默认值	254
19.1.3	约束值	256
19.1.4	封装方法	257
19.1.5	只读属性	258
19.2	改进的赛马类	259
19.3	进一步学习	260
19.4	练习	260
第 20 章	高级测试	261
20.1	跳过测试	261
20.2	测试面向对象特性	262
20.3	分组测试	263
20.4	测试大型字符串	264
20.5	测试文件	265
20.6	测试 STDOUT 和 STDERR	266
20.7	使用模拟对象	268
20.8	编写我们自己的 Test::*: 测试模块	270
20.9	练习	273
第 21 章	贡献到 CPAN	274
21.1	Perl 综合典藏网	274
21.2	准备阶段	274
21.3	PAUSE 的工作方式	275
21.3.1	索引器	276
21.3.2	模块维护人员	277
21.4	在我们开始工作之前	278
21.5	准备发行版	278
21.5.1	创建或更新 README	279
21.5.2	检查构建文件	279
21.5.3	更新清单	279
21.5.4	添加版本字符串	281
21.5.5	测试发行版	281

21.6	上传发行版.....	282
21.7	在多个平台上测试.....	282
21.8	发布模块.....	283
21.9	练习	283
附录	练习答案.....	285

第1章

简介

欢迎你进一步了解 Perl。你打开这本书的原因可能是你想编写超过 100 行的 Perl 程序，或者你的老板要求你来学。

Learning Perl 是一本经典著作，因为该书将介绍运用 Perl 编写小型和中型程序的方法（据观察，这也是 Perl 编程所完成的主要工作）。但是，为了避免“本书”的篇幅太长以至于吓退初学者，我们特意慎重地对知识进行梳理和选择。

后面的篇幅会以同样的风格介绍余下的故事。本书将会覆盖编写 100 行甚至 1 万行（甚至更长）的程序所需要的知识。

例如，你将会学到与多名程序员在同一项目中共同工作的方式，通过编写可复用的 Perl 模块，你可以将这些模块包含进发行版中由普通的 Perl 工具使用。这很重要，因为除非你每天工作 35 小时，否则你很难在没有帮助的情况下完成更庞大的任务。同样，在协作环境下，你要确保你开发的所有代码在最终的应用程序中能够很好地同他人的代码配合。

本书也展示如何处理更大和更复杂的数据结构。例如我们可能会提到的一个所谓的“散列的散列”或“数组散列的数组的数组”。只要你掌握了一些关于引用的知识，你就在通往理解任意复杂的数据结构的道路上，这将使你的工作更轻松。

其次，你还必须理解面向对象编程这一时髦概念，就是允许你的部分代码（由其他程序员编写的代码）能在同一个项目中或多或少地被重用。本书也很好地介绍该部分内容，尽管你可能从来没有面向对象的概念。

在团队中工作的一个重要的方面是：有一个发布周期以及一个进行单元测试和集成测试的流程。在本书中你会学到如何把你的代码打包成一个发行版并且对该发行版提供单元测试，在目标环境中对你的代码进行开发和验证。

另外，就像我们在已经出版的 *Learning Perl* 中所保证和传递的那样，我们将会在整个

学习过程中用一些有趣的例子和双关语让你的学习过程充满乐趣。尽管我们把 Fred、Barney、Betty 和 Wilma 送回家了，但我们让一些新的明星来担当角色。

1.1 背景知识

我们假定你已经读过 *Learning Perl*，至少是读过第 5 版，或者至少你假装已经读过，并且已经使用 Perl 有一段时间，已经完全理解它的基础知识。例如，我们将不会在本书中解释访问数组元素或者从子例程返回值的方法。

确保你已经理解下面的内容，所有这些都在 *Learning Perl* 中介绍过。

- 在你的系统上运行 Perl 脚本的方法。
- Perl 的 3 个基本变量类型：标量、数组和散列。
- 控制结构，如 while、if、for 和 foreach。
- 子例程。
- 基本的正则表达式。
- 列表操作符，如 grep、map、sort 和 print。
- 文件操作，如 open、文件读取和 -x（文件测试）。

本书对于这些主题进行深入探讨，不过假定你已经掌握这些基础知识。

本书最后的部分讲述发行版和对 CPAN 做贡献。要做到这些，你现在需要申请一个 PAUSE 账户，当你读到对应部分的时候，就可以使用它。可以通过以下链接申请该账户：https://pause.perl.org/pause/authenquery?ACTION=request_id.

1.2 strict 和 warnings

Learning Perl 中介绍了 strict 和 warnings 杂项，并且希望你在所有代码中使用它们。然而对于你将在本书中见到的大多数代码，假定我们已经打开 strict 和 warnings，因此我们不必因为重复的样本代码而分散精力，正如我们不使用 shebang 行和常见的文本块。当我们呈现完整的示例时，我们也将包含这些杂项。

你可能希望做我们做的事情，而不是从零开始编写程序，我们打开一个模板，里面有我们常用的代码。直到你开发出自己的模板，完成标准的文档并且用你喜欢的做事方式，你可以从一个简单的示例开始，假定周围都是如下所示的代码示例：

```
#!/usr/local/bin/perl  
use strict;  
use warnings;  
  
__END__
```

1.3 Perl v5.14

本书使用当前最新的 Perl v5.14 版本，该版本于 2011 年发布。通常，该语言的详细信息在此版本下是稳定的。特别是自从一些双重生命的模块在 CPAN 上分别独立出现时，我们使用的一些模块可能已经更新。因为我们通常提出 Perl 的基本理念并且通常扼要概述模块，所以对于每一个模块的更新，你总是需要查看该模块的文档。



注意

当我们于 2012 年年中完成本书的编写时，Perl v5.16 版已经准备在本书递交出版商一周后发布，我们可能很巧合地在本书中会描述到它的某些特性。

一些更新的特性需要我们明确地声明，我们希望使用它们而不必影响针对之前 Perl 版本的程序。启用这些特性最简单的办法是告诉 Perl 我们需要的版本。数字 5.014 必须有 3 个数字放置于小数点后（假使有一天出现 Perl 5.140）：

```
use 5.014;
```

```
say "Hello World!";
```

可以加上符号 v 和它的多个部分：

```
use v5.14.2;
```

使用两个小数点的形式，可以省略字母 v：

```
use 5.14.2;
```

但是，这将诱惑我们在所有情况下省略字母 v。

每当我们编写一些需要 Perl 特定版本中特性的代码时，我们将会插入 use v5.14 行（或者任何合适的版本信息），使用能够使特性生效的第一个版本。如果可以，我们也会展示一些能够在 Perl 之前版本上工作的代码。我们考虑使用 Perl v5.8 版，该版本于 2002 年第一次发布，是任何人都能使用的最早版本，因此当没有指定一个版本时，该代码示例就假定为 Perl v5.8。通常情况下，我们致力于使编写的代码能够用于尽可能多的人和尽可能多的 Perl 版本上，但是我们也希望你使用尽可能新的 Perl 版本。

要掌握关于 Perl v5.14 的更多基础知识，你可能需要查阅 *Learning Perl* 第 6 版。

关于版本的一个注解

在本书中，我们通过前面的字母，以 v5.M.N 的形式表示 Perl 的版本。目前为止，我们也在版本前加一个前缀“Perl”，但是当我们谈及版本间的差异时，将会变得很沉闷。相反，我们将不再谈论 Perl 的这一点。当我们说“v5.14.2”时，我们就是在谈论 Perl 5.14.2 版。这是在我们编写本书时的维护版本，尽管 v5.16 版的发布指日可待。

在 v5 之后的数字要么是奇数要么是偶数，并且这就是实验版本和维护版本的差别。维护版本（如 v5.14）用于普通用户和生产场景。实验版本（如 v5.15）是用于 Perl 5 核心维护小组（Porter）添加新特性、重新实现或者优化代码和改变不稳定的代码的地方。当他们准备完毕时，将实验版本升级到维护版本，是通过将第二个数字加一，变为偶数来表示的。

对于第三个数字（如 v5.14.2 中的 2）是一个修正发布。当我们说 v5.14 时，我们所指的是该版本下的全部修正版本。有时候，我们需要表示一个特定版本，在 *Learning Perl* 中，你可能记得在 v5.10.0 版和 v5.10.1 版之间，智能匹配修正了一个严重的设计 bug，并且改变了它的行为。

本书仅限于 v5，还有另外一件事，有时候叫做 Perl v6，但是这与 v5 毫无关系。Perl v6 被设计成一个全新的语言规范，并且也是由 Larry wall 设计的，但它不是 v5 的升级（即使在 2000 年，我们认为可能是这样）。我们知道这很让人迷惑，并且对于 v6 的用户而言也是一样，这就是 v6 规范的实现使用了不同名字的原因，如 Rakudo 和 Niecza。

1.4 关于这些脚注

类似于 *Learning Perl*，本书加入了一些更深奥的主题，并且对于这些主题添加了脚注。第一次阅读本书的读者可以选择先不读，再到下一次阅读时再阅读该部分主题。脚注中的内容与本书后续内容关系不大。

1.5 关于后续的练习

完成每一章后面的练习很重要。亲自动手训练效果会更好。提供该训练的最佳方式是在每看书半个小时至一个小时后，提供一系列练习。如果你的阅读速度很快，阅读完一章花费的时间可能小于半小时。慢一点，深呼吸，然后做练习！

每个练习有一个“完成分钟数”的评级。我们打算使该评级能够达到钟形曲线的中点，但如果你花费了更多或者更少的时间，也别灰心。有时候这仅仅取决于你在学习或工

作中，面对相同编程任务的次数。这个数字只是一个参考。

附录中列出了每个练习的答案。再说一次，别偷看答案；否则你将毁掉练习的价值。

1.6 获取帮助的方式

作为本书的作者，我们总是很高兴尽我们所能地帮助别人，但是我们已经被爆满的 E-mail 所淹没。有一些在线资源，你可以从那里获取帮助，你可以直接从我们这边获取这些资源，也可以从其他很多在 Perl 社区中有过贡献的人那里获得。

Stack Overflow (<http://www.stackoverflow.com/>)

Stack Overflow 是一个对于所有类型的编程问题进行免费问答的网站，并且有很多知识丰富的 Perler 经常回答你的问题。你很可能在一个小时内得到免费的完美解答。你甚至可能从本书作者那里得到答案。

Perlmonks (<http://www.perlmonks.org/>)

Perlmonks 是一个在线 Perl 社区，你可以在这里提问，发表你关于 Perl 的想法，并且和其他 Perler 互动。如果你有一个关于 Perl 的问题，人们可能愿意在 Perlmonks 上讨论。你可以搜索归档文件或者开创一个新话题。

`learn@perl.org` 和 <http://learn.perl.org/>

`learn@perl.org` 邮件列表是一个专门为 Perl 新手提问题而设计的安全地方，不必担心你会打扰任何人。该列表在等待你的问题，无论你问的是多基础的问题。

`module-authors@perl.org`

如果你的问题是专门关于编写和发布模块的内容的，有一个特别的邮件列表：
`module-authors@perl.org`。

`comp.lang.perl.misc`

如果你更多的时间在使用 Usenet，你可以在 `comp.lang.perl.misc` 上提问题。一些长期的 Perl 用户关注这个组，并且有时候他们也很有帮助。

1.7 如果是一个 Perl 课程讲师

如果你是一个 Perl 讲师，决定使用本书作为你的教材，你需要知道，对于大多学员而言，本书中的练习都足够短小，只需 45 分钟到一个小时就可以完成，留下的一点时间可以休息一下。一些章节的练习可能会更快一些，而另一些章节可能需要稍长的时间。这是因为一旦在方括号中写入所有的小数字，我们就发现我们不知道添加的方式。

现在，让我们翻开下一页，正式进入学习阶段。

1.8 练习

在每一章结尾，我们按照如下方式列出了练习。在每个练习之前，我们加入了我们认为大多数人完成该练习所需花费的时间。如果你需要更长时间，这也没什么。

可以在附录 A 中找到本练习的答案。

1. [5 分钟] 在 <http://pause.perl.org/> 创建一个 PAUSE 账号。你将在本书的最后一章用到该账号，并且我们希望你能提前创建好。
2. [5 分钟] 查阅本书的网站：<http://www.intermediateperl.com/>。你可能对于下载的部分感兴趣，该部分有一些对于练习有用的文件。下载归档文件，即使你后续没有办法访问互联网，你也能离线使用这些文件。

使用模块

Perl 的一个杀手级特性是 Perl 综合典藏网 (Comprehensive Perl Archive Network)，我们称其为 CPAN。Perl 安装时已经自带许多模块，但是 CPAN 拥有更多的第三方模块。如果我们需要用 Perl 解决某些问题，或者完成某个任务，可能在 CPAN 上就有现成的模块能够帮助我们完成。高效的 Perl 程序员是能够聪明地使用 CPAN 的人。*Learning Perl* 中对 CPAN 有一个简单的介绍，因为它太重要了，我们将在此再次叙述。



注意

我们可以浏览 CPAN 的主页 (<http://www.cpan.org/>)，或者它的搜索界面：CPAN Search (<http://search.cpan.org/>) 和 MetaCPAN (<https://www.metacpan.org/>)。

模块是用于程序的构建块，模块能够提供可复用的子例程、变量和甚至是面向对象的类。在通往构建模块的道路上，我们将展示你可能感兴趣的一些东西。我们也将介绍一些使用模块的基础知识，这些知识已经由其他人编写完成。

正如我们在 *Learning Perl* 中所注明的，我们不必理解模块的全部内容，以及当使用模块时它们内部的运作方式（学习本书后你将会有更深入的理解）。通过模块文档中的示例，我们能够做很多事情。为了启动 Perl，我们将立即开始使用 Perl 模块，等以后再解释它们的结构和特殊语法。

2.1 标准发行版

Perl 已经自带很多流行的模块，在 v5.14 发行版中，超过 66MB 的内容是模块。在 1996 年 10 月，v5.3.7 发行版拥有 98 个模块。在 2012 年年初，v5.14.2 发行版拥有 652 个模块。这确实是 Perl 的一个优势：它已经自带了我们所需要的很多东西，让你不需要做

太多额外的工作，就可以完成有用并且复杂的程序。



注意

使用 `Module::CoreList` 模块查看不同 Perl 版本中自带模块的信息，这就是我们最终得到这些数字的方法。

贯穿本书，我们将试图确定 Perl 安装时自带的模块是哪些（并且通常情况下，是 Perl 第一次收录这些模块时的版本）。我们称它们为“核心模块”，或者将它们标注为“标准发行版”。如果我们安装了 Perl，我们将拥有这些模块。因为我们使用 v5.14 编写本书，当考虑 Perl 的核心内容是哪些时，我们将假定为 Perl 当前版本的模块内容。

当开发我们自己的代码时，我们可能希望考虑是否仅使用核心模块，以便我们可以确保每一个安装相同 Perl 版本的人，都将拥有这些模块。我们将避免在此争论，主要是因为我们太热爱 CPAN 了，以至于不能没有它。同时我们也将立刻展示如何确定哪个模块来自于哪个 Perl 版本的方法。

2.2 探讨 CPAN

CPAN 无疑是 Perl 最吸引人的特性，它由一群努力工作的志愿者提供工具和服务，使人们发布高质量的软件并且评估和安装模块变得轻而易举。尽管这不是有用的 CPAN 工具综合列表，但是它包括我们最常使用的服务。从这个列表开始，我们也将很快地找到有用的服务。

CPAN Search (<http://search.cpan.org/>)

最久负盛名且最为大众所知的 CPAN 搜索服务是 Graham Barr 的 CPAN 搜索。我们能够浏览或者搜索模块，并且每个发行版的页面拥有关于该发行版重要事件和信息的链接，包括来源于第三方的信息，如测试结果、bug 报告等。

MetaCPAN (<https://www.metacpan.org/>)

MetaCPAN 是 CPAN 的下一代搜索界面。它几乎完成 CPAN Search 的所有事情，但是加入了一个 API，因此我们能够基于他们的数据编写我们自己的应用。

CPAN Testers (<http://cpantesters.org/>)

由作者上传至 CPAN 的每一个模块都被自动测试。一大群测试人员下载当前发行版，并且在他们各自的平台上测试。他们将结果发送至 CPAN Testers 的中央数据库，该数据库整理所有的测试报告。作为模块的作者，我们拥有免费的测试服务。作为模块的用户，我们能够检查测试报告，判断发行版的质量或者查看它是否能够在我们的计算机配置上工作。

CPANdeps (<http://deps.cpantesters.org/>)

David Cantrell 做了比 CPAN Tester 更进一步的工作：合并了测试报告中所有关于模块依赖性的信息。不是单独依赖于模块本身的测试。通过记录整个依赖性链中的测试结果，我们能够查阅安装问题的可能性。对于任何软件的安装，最让人感到失落的就是在安装过程中出现错误，但是 CPANdeps 能够帮助我们杜绝这类问题。作为该服务的一部分，David 也在维护 C5.6PAN 和 C5.8PAN，这些是 CPAN 的专业化版本，分别拥有仅能够在 v5.6 和 v5.8 这两个发行版上运行的每个模块的最终版本。

CPAN RT (<http://rt.cpan.org/>)

RT 是来源于 Best Practical^{译注1}的问题跟踪系统，并且为 CPAN 的模块作者提供服务。CPAN 上的每个模块在 RT 上自动获取问题队列，并且对于很多模块，RT 是主要的问题队列。一些模块的作者可能偏好使用其他的 bug 跟踪系统，但使用 RT 可以作为良好的开端。

2.3 使用模块

几乎每个 Perl 模块都具备文档，即使我们可能不知道所有幕后的工作方式，但如果知道接口的使用方式，我们就真的不需要担心这些事情。毕竟，这就是有接口的原因：隐藏细节。



注意

我们也可以使用 <http://perldoc.perl.org/> 网站，以 HTML 格式或者 PDF 格式读取 Perl 一些版本的文档。

在本地机器上，可以使用 perldoc 命令读取模块文档^{注1}。输入感兴趣的模块名称，然后得到它的输出文档：

```
% perldoc File::Basename

NAME

    fileparse - split a pathname into pieces
    basename - extract just the filename from a path
    dirname - extract just the directory from a path

SYNOPSIS

    use File::Basename;
```

译注 1：更多信息可以查询 <http://www.bestpractical.com/>

注 1：在 UNIX 系统上，man 命令也可以。

```
($name,$path,$suffix) = fileparse($fullname,@suffixlist)
fileparse_set_fstype($os_string);
$basename = basename($fullname,@suffixlist);
$dirname = dirname($fullname);
```

上面的文档只显示包含在文档的顶部最重要的部分（至少当我们开始时是最重要的）。模块文档通常遵循传统 UNIX 帮助手册的格式，以命名（NAME）部分和概要（SYNOPSIS）部分作为开始。

概要提供模块的使用示例，如果我们能够暂时抛开模块并且遵循示例中的方法，我们就能够使用该模块。这就是说，我们可能尚未熟悉一些 Perl 技巧和概要中的语法，但我们能够仅遵循示例中的方法进行并且完成期望的工作。

现在，因为 Perl 是一个过程、函数、面向对象和其他一些语言类型的混合体，Perl 模块引入大量不同的接口。我们将以略微不同的方式使用这些模块，但只要我们检查文档，就不会犯错误。

2.4 功能接口

要加载一个模块，可以使用 Perl 内置的 `use` 函数。我们不会在此深入讨论所有细节，稍后会在第 11~17 章展开。此时此刻，我们只是想要使用模块。我们从 `File::Basename` 模块开始，该模块来自于核心模块。将该模块加载到脚本中，可以按照如下所示方法：

```
use File::Basename;
```

当按照上面的方法编写程序时，`File::Basename` 模块把三个子例程（`fileparse`、`basename` 和 `dirname`）引入脚本（第 17 章将展示使用它们的方法）。从此以后，我们能够使用这些子例程，如同已经直接在同一文件中定义了它们一样。

```
my $basename = basename( $some_full_path );
my $dirname = dirname( $some_full_path );
```

这些例程从一个完整的路径名中提取文件名和目录部分。例如，如果我们在 Windows 上运行，并且`$some_full_path` 为 D:\Projects\Island Rescue\plan7.rtf，然后`$basename` 将得到 plan7.trf，并且`$dirname` 将得到 D:\Projects\Island Rescue。如果我们运行在类 UNIX 系统上，并且`$some_full_path` 为 /home/Gilligan/Projects/Island Rescue/plan7.rtf，然后`$basename` 将得到 plan7.trf，并且`$dirname` 将得到 /home/Gilligan/Projects/Island Rescue。

`File::Basename` 模块能够识别当前环境下的操作系统类型，因此它的函数能够理解对于我们可能遇到的不同种类的分隔符，正确解析字符串的方法。

然而，假如我们已经拥有一个 `dirname` 子例程。我们现在使用 `File::Basename` 模块提供的函数定义覆盖它！如果我们打开警告，就将看到一条由此产生的消息，但除此以外，Perl 真的不在乎。

选择需要导入的内容

很幸运，我们能够通过在模块名之后指定一个子例程列表，使 `use` 操作限定它的操作，该列表叫做导入列表：

```
use File::Basename ('fileparse', 'basename');
```

现在，该模块仅仅提供这两个导入的子例程，并且不干涉我们自己的 `dirname` 例程。但这样输入很笨拙，所以更常见的是按照如下方式使用 `qw` 操作符编写：

```
use File::Basename qw( fileparse basename );
```

即使导入列表中仅有一项，我们仍然倾向于将该项写入 `qw()` 列表，使程序一致性更好并且易于维护；通常情况下，我们将返回到要求“给我该模块的另一种编写方法”，如果使用 `qw()` 操作符编写导入列表，事情将变得很简单。

我们保护了自己拥有的 `dirname` 例程，但如果我们要使用 `File::Basename` 的 `dirname` 例程提供的功能该怎么做？没问题，我们只需要把它以完整包的规范拼写出就可以使用。

```
my $dirname = File::Basename::dirname($some_path);
```

`use` 后的列表名称不会改变在模块的包中定义的子例程（在该示例中，`File::Basename`）。我们能够始终使用方法完整的路径名，不必考虑导入列表，如下所示：

```
my $basename = File::Basename::basename($some_path);
```



注意

在调用子例程时，我们不需要在前面添加“`&`”符号，这是因为编译器已经通过 `use` 语句知道子例程的名称。

举一个极端的（但极端有用）例子，我们能够将导入列表指定为一个空列表，如下所示：

```
use File::Basename (); # no import  
my $base = File::Basename::basename($some_path);
```

空列表和没有列表的概念是不一样的。空列表的意思是说：“请不要导入任何子例程”，而没有列表的意思是说：“请导入默认的子例程”。如果模块的作者工作做得比较好，默认值可能就是我们想要的子例程。

2.5 面向对象的接口

与 `File::Basename` 模块导入的子例程相比，另一个核心模块 `File::Spec` 也提供类似的功能。`File::Spec` 模块用于支持文件规范的一般操作。（文件规范通常是文件或者路径的名称，但它可能是一个不存在的文件名称，也就是说，它不是一个真的文件名，是这样吗？）



注意

如果我们想要一个函数接口，我们就可以使用 `File::Spec::Functions` 模块。

与 `File::Basename` 模块不同，`File::Spec` 模块有一个主要面向对象的接口。与之前一样，我们使用如下 `use` 语句加载该模块：

```
use File::Spec;
```

然而，因为该模块拥有一个面向对象的接口，所以它不导入任何子例程。相反，接口告诉我们使用它的类方法访问该模块的功能。`catfile` 方法使用合适的目录分隔符连接一个字符串列表：

```
my $filespec = File::Spec->catfile( $homedir{gilligan},  
    'web_docs', 'photos', 'USS_Minnow.gif' );
```

以上语句调用 `File::Spec` 模块的 `catfile` 类方法，该方法为本地操作系统创建一个合适的路径，并且返回单个字符串^{注2}。这与 `File::Spec` 模块提供的其他大概二十多个操作在语法上都是相似的。

`File::Spec` 模块提供一些以恰当的方式处理文件路径的其他方法。我们能够在 `perlport` 文档中获得更多关于可移植性问题的信息。

2.5.1 一个更典型的面向对象模块：`Math::BigInt`

不要因为 `File::Spec` 模块没有创建对象，而且看上去比较像是“非面向对象”模块而感到失望。让我们查看另一个核心模块：`Math::BigInt`，该模块能够处理超出 Perl 本身范围的整数。



注意

Perl 被当前运行的硬件架构所限制，这是少数几个硬件限制之一。

不必像使用字面量一样使用数字，`Math::BigInt` 模块将其变为数字：

```
use Math::BigInt;  
  
my $value = Math::BigInt->new(2); # start with 2  
  
$value->bpow(1000); # take 2**1000  
  
print $value->bstr, "\n"; # print it out
```

如前所述，该模块没有任何导入内容。它的整个接口使用类方法，如 `new` 关键字，放置于类名之后用于创建实例，然后调用实例方法，如 `bpow` 方法和 `bstr` 方法，这些方法放置于实例名称后。

注2：在 UNIX 系统中，该字符串可能类似于`/home/gilligan/web_docs/photos/USS_Minnow.gif`。在 Windows 系统中，通常使用反斜线作为目录分隔符。该模块使我们很容易编写可移植的代码，至少考虑到文件规范。

2.5.2 更佳的模块输出

Perl 的强项之一是它的报表功能。我们可能认为它仅限于文本，但是通过使用合适的模块，我们能够创建任何格式。例如，使用 Spreadsheet::WriteExcel 模块，当我们能够创建不仅有用而且格式漂亮的 Excel 文档时，我们就成为办公室的明星。

正如我们直接使用 Excel 应用程序所知道的，我们打开一个工作表并且在工作表中放入数据。如下所示，直接使用文档中的代码，我们很容易创建第一个工作表：

```
use Spreadsheet::WriteExcel;

# Create a new Excel workbook
my $workbook = Spreadsheet::WriteExcel->new('perl.xls');

# Add a worksheet
my $worksheet = $workbook->add_worksheet();
```

此时可以插入数据。与 Excel 类似，该模块对于以字母命名的行号和以数字命名的列号能够跟踪行和列。将内容放入第一个单元格，我们遵循文档中的示例代码，按照如下方式，使用 write 方法写入数据。

```
$worksheet->write( 'A1', 'Hello Excel!' );
```

然而，在程序内部，可以很容易使用数字跟踪行和列，因此 Spreadsheet::WriteExcel 模块也按照该方法实现。写方法已经足够智能，能够识别我们正在使用的单元格描述，如下所示，由于我们不得不记住模块从零开始计数，因此第一行和第一列都是零。

```
$worksheet->write( 0, 0, 'Hello Excel' ); # in Excel's A1 cell
```

这已经使我们可以做很多事情，但我们还能够做更多，使工作表看起来更漂亮。首先，必须创建一个表单格式：

```
my $red_background = $workbook->add_format(
    color    => 'white',
    bg_color => 'red',
    bold      => 1,
);

my $bold = $workbook->add_format(
    bold      => 1,
);
```

如下所示，一旦我们拥有了一个格式，我们就能够调用该格式作为最后一个参数写入数据：

```
$worksheet->write( 0, 0, 'Colored cell', $red_background );
$worksheet->write( 0, 1, 'bold cell', $bold );
```

除了 write 之外，还有一些其他方法用于处理特定类型的数据。如果我们想要插入如同“01234”的字符串，我们不希望 Excel 忽略前导的数字“0”。然而，如果没有对于 Excel 进行特别指定，Excel 就尽可能猜测输入的数据内容。如下所示，我们将向 Excel 说明，可以使用 write_string 方法写入字符串：

```
my $product_code = '01234';
$worksheet->write_string( 0, 2, $product_code );
```

还有一些其他关于 `write` 方法的特殊用法，因此检查模块文档，以查看可以写入单元格的其他内容。

除数据以外，还能够创建公式。如下所示，可以使用 `write_formula` 方法写入公式，但公式字符串以“=”符号开始（与 GUI 中的操作方式一致）：

```
$worksheet->write( 'A2', 37 );
$worksheet->write( 'B2', 42 );
$worksheet->write( 'C2', '= A2 + B2' );
```

关于该模块还有很多其他内容，并且我们应当通过检查模块文档，快速理解该模块的其他特性。后续关于引用的章节将展示更多示例。

2.6 核心模块的内容

核心模块、标准库、发行版或者版本，是标准发行版的一系列模块和插件（我们通过 CPAN 下载的版本）。当人们谈论“核心模块”时，通常是指一系列模块，我们能够期望任何特定的 Perl 版本都拥有这些模块，因此通常可以确保任何人都使用我们的程序而不必安装额外的模块。

可是，这样的描述使定义变得不明确。某些发行版，如 Strawberry Perl (<http://strawberryperl.com/>) 和 ActivePerl (<http://www.activestate.com/activeperl>)，它们在其各自的发行版中添加了额外的模块。一些供应商的版本，如 OS X，向通过他们的操作系统发布的 Perl 包中添加了额外的模块，或者甚至改变了一些标准模块。这些情况并不是那么令人厌烦。真正令人厌倦的是供应商从标准发行版删除了部分模块，或者把标准发行版分成不同供应商的安装包，因此我们不得不安装原本应该拥有的这些模块安装包^{注3}。

`Module::CoreList` 模块只是一个数据结构和接口，该模块把很多关于 Perl 5 版本的模块历史信息聚合在一起，并且提供一个可编程的方式以访问它们。该模块是一个关于变量和“类”对象接口的混合体。

如下所示，我们可以查阅一个特定的 Perl 版本中自带模块的版本号，只需要在小数位之后指定 5 位数字（3 位数字为最小版本号，两位数字为补丁版本号）：

```
use Module::CoreList;

print $Module::CoreList::version{5.01400}{CPAN}; # 1.9600
```

有时候我们希望知道相关的其他信息：第一个将该模块放入标准库的 Perl 版本是哪个？

`Module::Build` 模块是 Perl 的编译系统，第 12 章将描述该模块。`Module::CoreList` 模块

注 3：根据 Perl 的许可证，这些供应商不允许使用这种修改过的 Perl 版本，但他们这样做了。

自 v5.9.4 版本起成为 Perl 标准库的一部分。

```
use Module::CoreList;

Module::CoreList->first_release('Module::Build'); # 5.009004
```

如果只需要检查模块在第一次发布时的 Perl 版本号，就不必为加载 Module::CoreList 模块编写一个程序。只须运行 corelist 程序：

```
% corelist Module::Build

Module::Build was first released with perl 5.009004
```

如果我们拥有 Perl 的最新版本，就同样也包含 Module::CoreList 模块，我们可以使用该模块查找自身的信息：

```
% corelist

Module::CoreList was first released with perl 5.009002
```

2.7 Perl 综合典藏网

CPAN 是众多志愿者共同工作的产物，在开始流行使用 Web 之前，很多志愿者使用它们自己的小型（或者大型）FTP 站点。直到 1993 年年底，他们还在 perl-packrats 邮件列表上协调各自的工作量，之后，因为磁盘空间变得越来越便宜，所以他们决定将同样的信息复制到所有站点，而不是放在各自专门的站点上。该想法酝酿了一年左右，以 Jarkko Hietaniemi 建立的芬兰 FTP 站点作为 CPAN 的母站，所有其他镜像站点每天或者每小时从母站获取更新。

该站点的一部分工作是重新整理和组织分散在各处的 Perl 归档文件。为非 UNIX 架构操作系统的 Perl 二进制文件、脚本和 Perl 的源代码本身建立存放空间。可是，模块部分成为 CPAN 中最大并且最令人关心的部分。

在层次化功能目录中，用符号链接树组织 CPAN 中的模块，指向作者目录中实际文件的位置。模块部分还包含格式一般易于被 Perl 解析的索引，例如，Data::Dumper 模块的输出用于模块索引的具体细节。这些索引都由主服务器使用其他的 Perl 程序自动派生。通常情况下，CPAN 从一个服务器同步到另一个服务器的镜像工作由一个非常古老的 Perl 程序完成，该程序叫作 mirror.pl。

从它屈指可数的几台镜像服务器开始，CPAN 如今已经成为超过 200 台遍布于互联网各个角落的公共服务器，至少每天，甚至频繁到每小时，都在不停地更新。无论在世界的哪个角落，我们总能找到用于下载最新模块的临近 CPAN 镜像站点。

众多 CPAN 搜索和聚合站点中的一个，如 <https://www.metacpan.org/> 或者 <http://search.cpan.org/>，将可能成为我们与模块仓库交互最喜欢的站点。通过这些站点，我们能够搜索模块、查看它们的文档、浏览它们的不同发行版、查询它们的 CPAN 测试者报告以

及完成其他许多事情。

2.8 通过 CPAN 安装模块

通过 CPAN 可以直接安装一个简单的模块。可以使用 Perl 自带的 cpan 程序，只要告诉 cpan 需要安装的模块名称。如果我们想要安装 Perl::Critic 模块（可以自动审阅代码），按照如下所示的方法向 cpan 传递该模块名称：

```
% cpan Perl::Critic
```

初次运行 CPAN 时，我们可能必须仔细检查初始化 CPAN.pm 模块的每一步配置，但在此之后，该模块将直接工作。CPAN.pm 程序下载模块并且开始编译它。如果需要安装的模块依赖于其他模块，cpan 将自动获取依赖的模块并且编译它们。

如果不带参数运行 cpan，就将启动 CPAN.pm 中的交互 shell 模式。通过 shell 提示符，能够执行各种命令。可以按照如下方式安装 Perl::Tidy 模块，该模块能够用于清除 Perl 代码的格式。

```
% cpan  
cpan> install Perl::Tidy
```

要查阅关于 cpan 的其他特性，可以按照如下方式，通过 perldoc 命令查阅它的文档：

```
% perldoc cpan
```

CPANPLUS 从 v5.10 版开始成为 Perl 的核心模块，它提供连接至 CPAN 的另一个可编程接口。CPANPLUS 的工作方式与 CPAN.pm 类似，但也有我们没有在此展示的一些其他特性。如下所示，CPANPLUS 拥有 cpanp 命令，并且我们使用-i 开关安装模块。

```
% cpanp -i Perl::Tidy
```

与 cpan 类似，可以打开一个交互式的 shell，然后安装所需要的模块。如下所示，安装该模块将允许我们以编程的方式创建一个 Excel 电子表格。

```
% cpanp  
CPAN Terminal> i Spreadsheet::WriteExcel
```

要查阅关于 cpanp 的其他特性，可以通过 perldoc 查阅它的文档：

```
% perldoc cpanp
```

2.8.1 CPANminus

还有另一个便捷的工具，cpanm（cpanminus 的简写），尽管目前它还不是 Perl 的核心模块。cpanm 被设计为零配置、轻量级的 CPAN 客户端，用来处理大多数人想要做的事。我们能够从 <http://xrl.us/cpanm> 站点下载单独的文件，并且遵循它简单的说明启动。

一旦拥有 cpanm，我们可以按照如下方式安装需要的模块：

```
% cpanm DBI WWW::Mechanize
```

2.8.2 手动安装模块

我们也可以手动完成这些 cpan 为我们完成的工作，如果我们之前从没有尝试过，这至少还有教育意义。如果我们理解工具所做的事情，就更易于追踪所遭遇的问题。

我们下载模块发行版的归档文件、解压缩，然后进入它所在目录。如下所示，我们在此使用 wget，但并不在乎使用何种下载工具。我们必须找到需要使用的确切 URL，该 URL 可以从 CPAN 站点获得。

```
% wget http://www.cpan.org/.../HTTP-Cookies-Safari-1.10.tar.gz  
% tar -xzf HTTP-Cookies-Safari-1.10.tar.gz  
% cd HTTP-Cookies-Safari-1.10
```

在此处，我们使用两种方法中的一种（该方法将在第 12 章中详细解释）。如果我们找到一个名为 Makefile.PL 的文件，就运行如下这一系列编译、测试和最终安装源代码的命令。

```
% perl Makefile.PL  
% make  
% make test  
% make install
```

如果我们没有在系统目录中安装模块的权限，就可以通过配置 INSTALL_BASE 参数将该模块安装到其他路径。



注意

Perl 的默认库目录由任何配置和安装 Perl 的人设定（即使这意味着我们接受默认设定）。这可以通过 perl -V 查看。

```
% perl Makefile.PL INSTALL_BASE=/Users/home/Ginger
```

为了使 Perl 能够在以上目录中查找所安装的模块，可以设置 PERL5LIB 环境变量。Perl 把这些目录添加到它的模块目录搜索列表中。下面是在 Bourne shell 中的设置方法：

```
% export PERL5LIB=/Users/home/Ginger
```

也可以使用 lib 编译指令将模块的安装目录添加到模块的搜索路径中（尽管这样使用看起来并不友好），因为我们不但要修改代码，而且当我们想要在其他机器上运行代码时，可能在使用不同的目录。

```
#!/usr/bin/perl  
use lib qw(/Users/home/Ginger);
```

倒退一分钟，如果我们在模块的安装目录中找到的是 Build.PL 文件而不是 Makefile.PL 文件，如下所示，安装模块的操作流程是一致的，只是这些发行版使用 Module::Build 模块编译和安装代码。

```
% perl Build.PL  
% perl Build  
% perl Build test  
% perl Build install
```

如果使用 Module::Build 模块在私有目录下安装模块，我们只需要添加--install_base 参

数，告诉 Perl 以之前同样的方式查找模块：

```
% perl Build.PL --install_base /Users/home/Ginger
```

有时候我们会发现在安装包中有 Makefile.PL 和 Build.PL，那么该使用哪一个呢？答案是两个都可以。

2.9 适时设定路径

Perl 会浏览一个特殊的 Perl 数组 @INC 中的目录元素，以查找需要调用的模块。当编译 perl 时，选择一个默认的目录列表作为模块的搜索路径。可以通过运行 perl 并且使用 -V 命令行开关，在输出中查阅这些信息。

```
% perl -V
```

也可以通过编写单行 Perl 命令输出它们：

```
% perl -le "print for @INC"
```

use 语句在编译时执行，因此它在编译时通过 @INC 数组查询模块的搜索路径。除非将 @INC 数组列入考虑范围，否则这使程序变得难以理解。我们需要在加载模块之前修改 @INC 数组。

例如，假定我们拥有自己的目录（位于 /home/gilligan/lib），并且安装我们自己的 Navigation::SeatOfPants 模块（位于 /home/gilligan/lib/Navigation/SeatOfPants.pm）。当我们加载该模块时，Perl 却找不到它：

```
use Navigation::SeatOfPants;      # where is it?
```

Perl 将报错，指出它不能在 @INC 数组中找到所需加载的模块，并且显示它在该数组中的全部目录。

```
Can't locate Navigation/SeatofPants.pm in @INC (@INC contains: ...)
```

我们可能想要在调用 use 语句之前，将需要的模块目录添加到 @INC 数组中。然而，如果按照如下方式添加：

```
unshift @INC, '/Users/gilligan/lib';  # broken
use Navigation::SeatOfPants;
```

程序还不能运行。为什么？因为 unshift 语句在运行时执行，远远落后于编译时执行的 use 语句。两条语句在词法上接近，但不是执行时间上接近。仅仅因为我们在编写代码时，使两条语句相邻，并不意味着语句执行顺序与编写顺序一致。我们希望在 use 执行前修改 @INC 数组的内容。其中一个方法是在 unshift 两侧添加 BEGIN 块：

```
BEGIN { unshift @INC, '/Users/gilligan/lib'; }
use Navigation::SeatOfPants;
```

现在，BEGIN 块在编译时编译和执行，为后续 use 语句的调用设定合适的路径。

然而，这样编写很繁琐，并且需要为此做很多令人讨厌的解释，特别是向后期维护和

修改这些代码的程序员。可以按照如下方式用之前使用过的一条简洁的编译指令替换这堆乱七八糟的东西：

```
use lib '/Users/gilligan/lib';
use Navigation::SeatOfPants;
```

现在，lib 编译指令获取一个或者多个参数，并且将它们添加到@INC 数组的起始部分，这与之前的 unshift 命令实现同样的功能。该方法可行的原因是它在编译时执行，而不是运行时。因此，及时为 use 后面紧跟的语句做准备。



注意

use lib 语句也能够从所需要的库中 unshift 一个独立架构的库，这显得比之前相应的显式声明更有价值。

因为 use lib 编译指令几乎总是有一个独立位置的路径名，所以这是传统的方式并且我们建议你将它放置于文件的顶部。当我们需要把该文件移植到一个新的操作系统中或者当 lib 路径名变更时，这将易于查明和更新。（如果能够将模块安装于一个标准@INC 数组的路径下，就能够完全消除 use lib 语句，但这样并不总是实用。）

可以认为 use lib 语句不是指“使用这个库”，而是“使用这个路径查找库（和模块）”。很多情况下，我们看到代码按如下方式编写：

```
use lib '/Users/gilligan/lib/Navigation/SeatOfPants.pm'; # WRONG
```

然后，程序员会对它在定义中没有获取后续内容的原因感到疑惑。use lib 语句确实在编译时运行，因此以如下方式编写的代码同样不能运行：

```
my $LIB_DIR = '/Users/gilligan/lib';
...
use lib $LIB_DIR;      # BROKEN
use Navigation::SeatOfPants;
```

Perl 在编译时创建\$LIB_DIR 变量的声明（因此我们使用 use strict 语句也不能够得到错误，尽管实际使用 use lib 语句将会报错），但是直到运行时才会执行对于/Users/gilligan/lib/实际的赋值操作。又一次晚了！

此时此刻，我们需要将该声明语句放入 BEGIN 块中，或者依赖于另一个编译时操作：使用 use constant 语句设定一个常量：

```
use constant LIB_DIR => '/Users/gilligan/lib';
...
use lib LIB_DIR;
use Navigation::SeatOfPants;
```

这样，问题再次解决。也就是说，我们需要的库最终取决于运算结果。这将处理我们需要解决的 99% 的问题。

我们不必始终理解路径是在运行时还是编译时执行。在之前的示例中，我们对路径进行硬编码。如果我们不知道确切的路径将是什么，因为我们需要在很多机器之间复制

这些代码，FindBin 模块，Perl 自带的核心模块将帮助我们完成该任务。该模块查找脚本目录的完整路径，我们能够使用它创建需要的新路径：

```
use FindBin qw($Bin);
```

现在，\$Bin 变量是对应脚本所在目录的路径。如果在同一目录中有库文件，下一行可以按照如下方式编写：

```
use lib $Bin;
```

如果库文件放置于与脚本目录相邻的目录中，我们就按照如下方式，将正确的路径部分组合起来，使代码能够正确运行：

```
use lib "$Bin/lib"; # in a subdirectory
```

```
use lib "$Bin/../lib"; # up one, then down into lib
```

因此，如果通过脚本所在目录知道了相对路径，我们就不必对整个路径进行硬编码，这将使脚本更具可移植性。

当我们在第 12 章开始编写自己的模块时，将介绍更多相关的技巧。

2.10 在程序外部设定路径

use lib 语句有一个很大的弊端：我们不得不在源代码中放入库文件的路径。我们可能在某个位置拥有我们自己安装的本地模块，但我们的同事在另一个位置拥有该模块。我们不希望在从团队其他成员中获取源代码时，每次都修改源代码，并且我们不希望在源代码中列出所有模块的位置。Perl 提供一系列方法，使我们能够扩展模块的搜索路径而不必修改源代码。

2.10.1 使用 PERL5LIB 扩展@INC

Skipper 必须编辑使用他的私有库文件的每个程序，包括之前部分的代码行。如果有太多代码需要修改，就能够设定 PERL5LIB 环境变量为目录名。例如：在 C shell 中，可以使用如下方式：

```
setenv PERL5LIB /home/skipper/perl-lib
```

在 Bourne 风格的 shell 中，可以使用如下方式：

```
export PERL5LIB=/home/skipper/perl-lib
```

Skipper 能够设定一次 PERL5LIB，然后忽略它。然而，除非 Gilligan 能够拥有同样的 PERL5LIB 环境变量，否则它的程序将会运行失败！当 PERL5LIB 环境变量适用于个人使用时，我们不能依赖它与其他人共享程序。（并且我们不能为整个团队的程序员添加一个通用的 PERL5LIB 环境变量。相信我们，我们尝试过。）

PERL5LIB 变量能够在类 UNIX 系统中包含由冒号分割的多个目录，并且在 Windows 系统

中包含由分号分割的多个目录（除了这些之外，我们靠自己）。Perl 在@INC 的起始部分部插入所有指定的目录。在 UNIX 系统中使用类 bash 的 shell，将按照如下方式设定：

```
% export PERL5LIB=/home/skipper/perl-lib:/usr/local/lib/perl5
```

在 Windows 系统中，按照以下所示设定：

```
C:\.. set PERL5LIB="C:/lib/skipper;C:/lib/perl5"
```

当系统管理员可能需要在系统级启动脚本上添加一个 PERL5LIB 设定时，大多数人对此感到不悦。设定 PERL5LIB 环境变量的目的是使非管理员用户能够扩展 Perl，以识别额外的模块安装目录。如果系统管理员希望添加额外的模块安装目录，他只需要重新编译并且重新安装 Perl。

2.10.2 在命令行扩展@INC 目录

如果 Gilligan 意识到 Skipper 的一个程序丢失合适的指令，Gilligan 要么添加合适的 PERL5LIB 变量，要么使用一个或者多个-I 选项直接调用 perl。例如，调用 Skipper 的 get_us_home 程序，可能需要按照如下方式在命令行执行：

```
% perl -I/home/skipper/perl-lib /home/skipper/bin/get_us_home
```

显然，如果程序本身定义额外的库文件，对于 Gilligan 就会很容易。但有时需要一直等到添加一个-I 开关才解决。即使 Gilligan 不能编辑 Skipper 的程序也能工作。他仍然有权限阅读代码，但是比如，Gilligan 能够和 Skipper 的程序一样使用该技术尝试新版本库文件。



注意

使用 PERL5LIB 或者-I 选项扩展@INC，同样自动添加指定目录下的特定于版本和架构的子目录。自动添加这些目录，将简化包含架构或者版本敏感组件的 Perl 模块的安装任务，例如，编译的 C 代码。

2.11 local::lib

默认情况下，CPAN 工具将新模块安装到与 perl 相同的目录，但是我们可能没有在该目录创建文件的权限，或者我们可能不得不调用某种管理员权限来实现。这是 Perl 新手的一个常见问题，因为他们不能理解在他们喜欢的任何地方安装 Perl 模块的难度。一旦我们知道如何做，我们就能够安装和使用任何模块，而不必请求系统管理员为我们安装。

local::lib 模块，我们将不得不从 CPAN 获取，因为该模块目前还不是 Perl 的核心模块^{译注2}。该模块设定多种环境变量，它们将影响 CPAN 客户端安装模块的位置和 Perl 程序查找所安装模块的位置。我们能够看到它们在命令行中使用-M 开关设定加载模块的方法，但是没有任何其他参数。假若那样，local::lib 模块使用 Bourne shell 命令输出它的设定，

译注 2：从 v5.14 开始成为核心模块。

可以放置如下所示的任意一个登录文件：

```
% perl -Mlocal::lib  
export PERL_LOCAL_LIB_ROOT="/Users/Ginger/perl5";  
export PERL_MB_OPT="--install_base /Users/Ginger/perl5";  
export PERL_MM_OPT="INSTALL_BASE=/Users/Ginger/perl5";  
export PERL5LIB="...";  
export PATH="/Users/Ginger/perl5/bin:$PATH";
```



注意

即使使用不同的 shell 环境变量，local::lib 也能输出 Bourne shell 命令。我们不得不将这些命令转换成我们自己的。

诀窍在于安装 local::lib 模块，这样我们就可以使用该模块，可以按照如下方式手动下载和安装 local::lib 模块：

```
% perl Makefile.PL --bootstrap  
% make install
```



注意

我们需要一个最新版本的 CPAN.pm 模块或者 APP::Cpan 模块，以使用 cpan 的-I 开关，我们已经在 Perl v5.14 中添加 local::lib 模块特性。

一旦使用 local::lib 模块，我们就能够通过 CPAN 工具使用它。如果我们使用-I 开关安装模块，cpan 客户端就将支持 local::lib 模块。

```
% cpan -I Set::Crossproduct
```

cpanm 工具是智能化的。如果我们已经设定的环境变量与 local::lib 模块为我们设定的环境变量一致，就直接使用它。如果不一致，就检查默认模块目录的写入权限。如果我们没有写权限，它就自动为我们启用 local::lib 模块。如果我们想要明确地指定使用 local::lib 模块，就需要按照如下方式：

```
% cpanm --local-lib HTML::Parser
```

如果我们使用 local::lib 模块，当在程序中加载模块时，程序就知道在何处查找我们安装的模块：

```
# inside our Perl program  
use local::lib;
```

我们展示了 local::lib 模块使用的默认设定，但是有一种方法可以处理我们想要使用的任何路径。在命令行条件下，可以通过-M 命令向加载的模块提供一个导入列表：

```
% perl -Mlocal::lib='~/perlstuff'  
export PERL_LOCAL_LIB_ROOT="/Users/Ginger/perlstuff";  
export PERL_MB_OPT="--install_base /Users/Ginger/perlstuff";  
export PERL_MM_OPT="INSTALL_BASE=/Users/Ginger/perlstuff";  
export PERL5LIB="/Users/Ginger/foo/lib/perl5/darwin-2level:  
    /Users/Ginger/perlstuff/lib/perl5";  
export PATH="/Users/Ginger/perlstuff/bin:$PATH";
```

2.12 练习

可以在附录中的“第 2 章答案”部分找到这些练习的答案。

1. [25 分钟] 读取当前目录中的文件列表，并且将名字变为完整的路径规范。不要使用 shell 或者外部程序来获取当前路径。对于 File::Spec 模块和 Cwd 模块，两者都是 Perl 自带的模块，能够帮助你完成任务。在每条路径的输出之前加上 4 个空格，之后加上换行符。
2. [20 分钟] 安装 local::lib 模块，并且当你安装 Module::CoreList 模块（或者你喜欢的其他模块）时，使用它。写一个程序，该程序报告 Perl v5.14.2 版本下所有模块的名称和第一次发布的时间。查阅 local::lib 的文档，查看是否有特别的安装说明。
3. [35 分钟] 解析在原书封底的国际标准图书编号（International Standard Book Number, ISBN）(9781449393090)。从 CPAN 安装 Business::ISBN 模块，并使用该模块从数字中提取组编码和出版商编码。

第3章

中级基础

在开始学习本书的实质性内容之前，我们希望介绍一些中级水平的 Perl 习语，并且将在本书后续的内容中使用它们。对于这些习语的理解和掌握程度可以用于区分初级和中级 Perl 程序员。我们也将介绍第一个字符集，并且将在贯穿本书的示例中使用它们。

3.1 列表操作符

列表是一个有序的标量集合，表的值是它们本身，有时我们在数组中存储列表，该数组容器用于存储有序列表。列表操作符用于处理多个元素，并且不介意它们使用字面量列表、来自子例程的返回值或者数组变量。

我们已经介绍了 Perl 中的一些列表操作符，但可能没有考虑到它们与列表一起使用。最常见的列表操作符是 print，我们向 print 操作符提供一个或者多个参数，print 操作符就自动将这些列表元素组合起来。

```
print 'Two castaways are ', 'Gilligan', ', and ', 'Skipper', "\n";
```

Learning Perl 这本书已经介绍过一些其他的列表操作符，如 sort 操作符将输入的列表进行排序。在他们的主题曲中，遇难者并没有按照字母顺序出场，sort 操作符可以修正这个错误：

```
my @castaways = sort qw(Gilligan Skipper Ginger Professor Mary Ann);  
reverse 操作符返回一个逆序列表。
```

```
my @castaways = reverse qw(Gilligan Skipper Ginger Professor Mary Ann);
```

如下所示，我们甚至可以在合适的位置使用这些操作符，将同样的数组名称放置于赋值语句的左右两边。Perl 先计算右边的值，得到结果，然后赋值回初始变量名：

```
my @castaways = qw(Gilligan Skipper Ginger Professor);
push @castaways, 'Mary Ann';

@castaways = reverse @castaways;
```

Perl 还有其他用于操作列表的操作符，并且一旦习惯使用它们，你就发现输入会更少，但是意图表达更清晰。

3.1.1 使用 grep 表达式过滤列表

grep 操作符使用一个“测试表达式”和一个值列表。它将列表中每一项按照顺序依次取出，并且放入\$_变量中。然后在标量上下文中对测试表达式求值。如果表达式的值为真，grep 表达式就会将\$_的值传递至输出列表：

```
my @lunch_choices = grep is_edible($_), @gilligans_possessions ;
```

在列表上下文中，grep 操作符返回一个包含所有选中项的列表。在标量上下文中，grep 表达式返回选中项的数目：

```
my @results = grep EXPR, @input_list;
my $count  = grep EXPR, @input_list;
```

如下所示，EXPR 代表任意指代\$_（显式或隐式）的标量表达式。例如，要查找所有大于 10 的数字，在如下所示的 grep 表达式中，我们将检查\$_是否大于 10：

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @bigger_than_10 = grep $_ > 10, @input_numbers;
```

结果为 16、32 和 64。以上示例使用指向\$_的显式引用。如下示例中的隐式引用指向来自模式匹配操作符的\$_变量：

```
my @end_in_4 = grep /4$/ , @input_numbers;
```

现在我们得到 4 和 64。

当 grep 表达式运行时，它隐藏任意存在于\$_中的值，也就是说 grep 表达式“借用”了该变量，但在表达式运行完成之后还原初始值。然而，\$_变量并不仅仅是数据项的副本；而是实际数据元素的别名，这与 foreach 循环中的控制变量类似。

如果测试表达式太复杂，就可以将它隐藏到一个子例程中：

```
my @odd_digit_sum = grep digit_sum_is_odd($_), @input_numbers;

sub digit_sum_is_odd {
    my $input = shift;
    my @digits = split //, $input; # Assume no nondigit characters
    my $sum;
    $sum += $_ for @digits;
    return $sum % 2;
}
```

对于上面的示例，得到的输出为列表 1、16 和 32。在子例程的最后一行中，当这些数字的所有数位之和除以 2 得到的余数为 1 时，意味着返回值为真。

但 grep 表达式的语法有两种形式：刚展示的表达式形式和如下所示的“块”形式。与

其定义一个仅用来做单次测试的显式子例程，还不如将子例程的主体逐行以块形式放置于 grep 操作符之后。在块形式的 grep 语句中，语句块和输入列表之间没有逗号：

```
my @results = grep {  
    block;  
    of;  
    code;  
} @input_list;  
  
my $count = grep {  
    block;  
    of;  
    code;  
} @input_list;
```



注意

这是有点古怪的 Perl 语法，代码块实际上是一个匿名子例程，这与 *Learning Perl* 中展示的 sort 操作符一样，这将在本书第 7 章中再次介绍。

与表达式形式类似，grep 表达式将输入列表中的每个元素临时放置于\$_ 中。下一步，它就对整个代码块求值。代码块中最后一行的求值表达式是测试表达式，它和所有测试表达式一样，在标量上下文中求值。因为它是完整的“块”，所以我们能够引入作用域限于该“块”的变量。如下所示，我们按照块形式重写了上面的示例。

```
my @odd_digit_sum = grep {  
    my $input = $_;  
    my @digits = split //, $input; # Assume no nondigit characters  
    my $sum;  
    $sum += $_ for @digits;  
    $sum % 2;  
} @input_numbers;
```

注意两个变化：在块形式中，输入值通过\$_ 变量传入，而不是参数列表，并且我们删除了 return 关键字。因为我们不再处在一个分离的子例程中：仅仅只是一个代码块中。所以保留 return 关键字就是错误的。当然，我们还可以按如下方式优化该例程，因为我们不再需要中间变量：

```
my @odd_digit_sum = grep {  
    my $sum;  
    $sum += $_ for split //;  
    $sum % 2;  
} @input_numbers;
```



注意

块形式的 grep 表达式中，return 关键字将退出包含整个代码块部分的子例程。当然，我们中的一些人在实际过程中初次使用该表达式编程时已经犯过这个错误。

我们可以在 grep 语句块中做任意想做的事情。假如我们在@links 数组中有一个 URL 列表，并且希望知道损坏的 URL 链接是哪一些。我们将链接列表发送至 grep 表达式，使用 HTTP::SimpleLinkChecker 模块（可以在 CPAN 上找到）检查它们，然后仅传递最终没有错误的链接：

```
use HTTP::SimpleLinkChecker qw(check_link);

my @good_links = grep {
    check_link( $_ );
    ! $HTTP::SimpleLinkChecker::ERROR;
} @links;
```

如果能够帮助我们和我们的同事更好地理解和维护代码，我们就启动中间变量的显式化，这才是我们主要关注的事情。

3.1.2 使用 map 转换列表

map 操作符将一个列表转换成另一个，它的语法和 grep 操作符的语法很类似，并且共享很多相同的操作步骤。例如，map 操作符把列表中的元素临时逐个放入\$_ 中，并且其语法允许表达式形式和块形式。

map 表达式用于格式转换而不是测试。map 操作符在列表上下文中对表达式求值，而不是像 grep 在标量上下文中对表达式求值。每次表达式的求值都为最终的输出列表提供一部分内容。所有各自独立的结果连接起来成为最终结果。在标量上下文中，map 表达式返回在列表上下文中的元素数目，但 map 表达式应该很少在除了列表上下文的其他场合下使用。

让我们从一个简单的示例开始：

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @result = map $_ + 100, @input_numbers;
```

如上例所示，数组中的 7 项元素依次映射到\$_，我们得到的每一项都依次添加到输出列表：每一项输出都比相应的输入数字大 100，因此@result 的值为 101, 102, 104, 108, 116, 132 和 164。

但是我们并没有限定每个输入必须对应唯一的输出。如下所示，我们可以看到当每个输入相应产生两个输出项时的情形：

```
my @result = map { $_, 3 * $_ } @input_numbers;
```

现在，对于每个输入项都有两个对应的输出项：1, 3, 2, 6, 4, 12, 8, 24, 16, 48, 32, 96, 64 和 192。如果我们需要的散列显示为一个数字与该数字的三倍组成键值对的时候，就可以将这个@result 数组保存为一个散列：

```
my %hash = @result;
```

或者，如下所示，直接使用 map 表达式生成散列而不使用中间列表：

```
my %hash = map { $_, 3 * $_ } @input_numbers;
```

这对于为散列的每个键生成一个有意义的值是适宜的，但有时我们不关注散列的值，因为我们想要使用一个更简单的方法，检查散列的一个元素是否在列表中。在这种情况下，我们向键提供任意值，仅仅只是要确保有键在散列中。如下例所示，如果存在，就使用 1 作为该键对应的有效值：

```
my %hash = map { $_, 1 } @castaways;  
  
my $person = 'Gilligan';  
  
if( $hash{$person} ) {  
    print "$person is a castaway.\n";  
}
```

map 表达式的功能是多种多样的；我们可以为每个输入项生成任意数目的输出项。并且我们并不总是需要生成同样数目的输出项。如下所示，查看一下当我们把数字打乱时所发生的事情：

```
my @result = map { split // } @input_numbers;
```

代码中的内联块将每个数字分割成单个的数位。对于 1、2、4 和 8，我们将得到单个结果。对于 16、32 和 64，我们将得到两个结果。当 map 表达式连接结果列表时，我们最终得到 1、2、4、8、1、6、3、2、6 和 4。

如果一个特定调用的结果是空列表，map 表达式就将这个空结果连接到更大的列表中，不为列表添加任何元素。我们能够利用该特性选择和删除数据项。例如，假定我们想要分割仅以数字 4 结尾的数字：

```
my @result = map {  
    my @digits = split //, $_;  
    if (@digits[-1] == 4) {  
        @digits;  
    } else {  
        ();  
    }  
} @input_numbers;
```

如果最后的数字是 4，我们就能够在列表上下文中，通过对 @digits 求值返回数字本身。如果最后的数字不是 4，我们返回一个空列表，这样就为特定的数据项高效地移除了结果。然而，我们能够使用 map 表达式替换 grep 表达式，反之则不行。

我们能够用 map 表达式和 grep 表达式做的所有事情，也能够使用显式的 foreach 循环完成。话说回来，我们也能够使用汇编程序或者二进制编码实现同样的功能。重点是正确运用 grep 表达式和 map 表达式能够降低程序的复杂度，使我们更关注高级的问题而不拘泥于具体的实现细节。

3.2 使用 eval 捕获错误

如果发生某些错误，一些平常的代码就可能使程序过早终止：

```
my $average = $total / $count; # divide by zero?  
print "okay\n" unless $match;/; # illegal pattern?  
  
open MINNOW, '>', 'ship.txt'  
or die "Can't create 'ship.txt': $!"; # user-defined die?
```

```
implement($_) foreach @rescue_scheme; # die inside sub?
```

仅仅因为代码中的某个片段出错，这并不意味着我们希望整个程序崩溃。Perl 使用 eval 操作符作为它的错误捕获机制：

```
eval { $average = $total / $count } ;
```



注意

eval 表达式是 Perl 的首选异常机制。查看 *Mastering Perl* 以获取 Perl 中处理错误的更深层次内容，包括一些拥有更佳异常框架的模块。

当运行 eval 语句块中的代码时，如果发生错误，代码块就将停止执行。但是即使块中的代码已经执行完毕，Perl 也继续运行 eval 语句块之后的代码。最常见的方法是在 eval 语句块执行完之后立即检查 \$@，该特殊变量要么为空（意味着没有错误），要么 perl 拥有来自代码运行失败的错误消息，可能类似于“除数为零”或者更长的错误消息：

```
eval { $average = $total / $count } ;  
print "Continuing after error: $" if $@;  
  
eval { rescue_scheme_42() } ;  
print "Continuing after error: $" if $@;
```

eval 语句块后的分号是必需的，因为 eval 是一个术语（不是控制结构，例如 if 或 while）。语句块是真实的语句块，并且可能包含词法变量（“my” 变量）和其他任意语句。作为函数，eval 语句块有类似于子例程的返回值（对最后一行表达式求值，或者之前通过 return 关键字返回的值）。如果块中的代码运行失败，它就什么值都不返回；如果块中的代码运行成功，它就在标量上下文中返回 undef，或者在列表上下文中返回空列表。因此，另一个计算平均数的安全方法如下所示：

```
my $average = eval { $total / $count } ;
```

现在 \$average 要么为商，要么为 undef，这取决于操作是否成功完成。

Perl 甚至支持嵌套的 eval 语句块。eval 语句块的强大之处在于，它能够在执行时持续追踪错误，所以能够捕获深度嵌套的子例程调用产生的错误。可是 eval 语句块不能捕获最严重的错误：使 perl 自己中断执行的错误。这些包含诸如未捕获的信号、内存溢出和其他灾难性的错误。eval 语句块同样不捕获语法错误，因为 perl 与其他代码一起编译 eval 代码块，它在编译时捕获语法错误，而不是运行时；它也不捕获警告（尽管 Perl 提供一个拦截警告消息的方法；在 perlvar 文档中查看 \$SIG{__WARN__}）。

对于简单的操作，直接使用 eval 语句块就足够了。我们将不会在此做深入探讨，正确地处理复杂情况是很棘手的。很幸运，v5.14 版本解决了这些问题，或者也可以使用 Try::Tiny 模块（可以在 CPAN 上找到）：

```
use Try::Tiny;
my $average = try { $total / $count } catch { "NaN" };
```

大多数这些奇怪的边际情况在 v5.14 版本中得到修正。

3.3 用 eval 语句块动态编译代码

还有使用 eval 语句块的第二种形式：它的参数是字符串表达式而不是代码块。它在运行时通过字符串的方式编译和执行代码。当这样做很有用并且支持这种方式时，一旦有任何不可信赖的数据进入字符串，就会很危险。但也有一些明显的例外，显然我们不建议对于字符串使用 eval 语句块。后续我们将使用 eval 语句块，因为这种方式可能在其他人的代码中出现，如下所示，我们将按照如下方式展示它的工作方式：

```
eval '$sum = 2 + 2';
print "The sum is $sum\n";
```

Perl 在词法上下文中执行 eval 语句块前后的代码，这意味着好像我们放入已经求值的代码。eval 语句块的结果即为最后一个表达式的值，因此我们不必将整个语句放入 eval 语句块中：

```
#!/usr/bin/perl

foreach my $operator ( qw(+ - * /) ) {
    my $result = eval "2 $operator 2";
    print "2 $operator 2 is $result\n";
}
```



注意

在 eval 语句块执行后续语句之前，Perl 先插入由双引号引用的字符串。如果我们想要将变量放入已经完成求值的代码中，就必须确保它不能够插入该变量。

此时，我们仔细检查操作符+、-、*和/，并且在 eval 语句块的代码中依次使用这些操作符。在传递至 eval 的字符串中，我们将\$operator 的值插入字符串。eval 语句块执行字符串所表述的代码，并且返回最后一个表达式的值，我们将最后一个表达式的值赋给\$result。

如果 eval 语句块不能够正确编译并且不能正确运行我们传递的 Perl 代码，它就将如同在块形式中一样设定\$@变量。在本例中，我们捕获所有“除数为零”的错误，并且我们没有设定所有数据为除数（另一种类型的错误）：

```
print 'The quotient is ', eval '5 /', "\n";
warn $@ if $@;
```

eval 语句块捕获语法错误并且将捕获的消息放置于特殊变量\$@中，我们将立即在调用 eval 语句之后检查该消息：

```
The quotient is
syntax error at (eval 1) line 2, at EOF
```

如果你之前没有注意到我们的警告，我们就再次重申：小心这类形式的 eval 语句块。如果能够找到其他方法代替 eval 方法完成需要的工作，就先尝试其他方法。我们将在第 11 章中使用 eval 语句块，从外部文件中加载代码，但在那时，我们也将展示一个更好的替代方案。

3.4 使用 do 语句块

do 语句块是 Perl 中一个强大但被忽视的特性。它提供了一个把一组语句聚集为单个表达式的方法，我们可以在其他表达式中使用该表达式。这很类似于内联子例程。与子例程一样，do 语句块的执行结果是最后一个表达式的值。

首先，考虑如下所示的小段代码，将三个可能的值赋给一个变量。声明\$bowler 为一个词法变量，并且使用一个 if-elsif-else 结构选择需要赋值的内容。我们以输入 4 次变量名得到单个赋值作为结束：

```
my $bowler;
if( ...some condition... ) {
    $bowler = 'Mary Ann';
}
elsif( ... some condition ... ) {
    $bowler = 'Ginger';
}
else {
    $bowler = 'The Professor';
}
```

然而，通过使用 do 语句块，只需要使用一次变量名。可以在声明的同时赋值，因为我们能够在 do 语句块中像以单个表达式的形式合并所有其他语句：

```
my $bowler = do {
    if( ... some condition ... ) { 'Mary Ann' }
    elsif( ... some condition ... ) { 'Ginger' }
    else { 'The Professor' }
};
```

do 语句块也非常适合创建一个操作的作用域。我们可能希望将一个文件的所有内容读入变量，如下所示，完成该操作的一个 Perl 习语是在 do 语句块中，为\$/变量（\$/为输入记录分隔符）和@ARGV 的本地化版本提供一个作用域，因此我们能够使用<>处理文件句柄的所有细节：

```
my $file_contents = do {
    local $/;
    local @ARGV = ( $filename );
    <>
};
```

与 eval 语句块类似，do 语句也有一个字符串参数形式。向 do 语句块提供一个字符串而不是代码块，就会尝试通过该字符串名称加载、编译文件，并且按照如下方式执行代码：

```
do $filename;
```

do 语句查找文件并且读取该文件，然后切换内容为 eval 语句块的字符串形式以执行它。do 语句忽视任何错误；程序将继续执行。不但如此，而且 do 语句将仔细检查其整个流程，甚至它已经加载的文件。由于这个原因，几乎没人这样使用 do 语句。还有更好的方法。

之前的章节展示了使用 use 语句加载模块的方法，并且提到该方法于编译时执行。此外还有一个方法加载模块。内置的 require 函数同样也能够加载模块，但该函数于运行时执行：

```
require List::Util;
```

如下所示，use 语句实际上是一个 BEGIN 块中的 require 语句和调用类导入的内容：

```
BEGIN { # what use is really doing
    require List::Util;
    List::Util->import(...);
}
```

我们应该向 ues 语句提供一个模块名称，但是我们能够像 do 语句一样给 require 语句提供一个文件名：

```
require $filename;
```

换句话说，require 能够记忆所加载的文件，并且能够避免重复加载同样的文件。

第 12 章将展示关于这些内容的更多信息。

3.5 练习

可以在附录中的“第 3 章答案”部分找到这些练习的答案。

1. [15 分钟] 编写一个程序，在命令行读取文件名列表，并且使用 grep 语句选取文件容量小于 1000 字节的文件。使用 map 语句转换列表中的字符串，在每个列表元素之前放置 4 个空格并且在之后放置换行符。最后输出结果列表。
2. [25 分钟] 编写一个程序，要求用户输入一个模式（正则表达式），然后通过标准输入读取数据，而不是命令行参数，最后在一些硬编码目录中输出匹配文件名称的列表（例如"/etc"或者'C:\\Windows'）。重复执行该命令，直到用户输入空字符串而不是模式。用户不应该输入 Perl 中传统上使用的斜杠作为模式匹配限定符，输入模式是由结尾的换行符分割。确保一个错误的模式（例如一对不匹配的圆括号）不会使整个程序崩溃。

引用简介

引用是处理复杂数据结构、面向对象编程和精美子例程的基础。在 Perl 的第 4 版和第 5 版加入该特性使这些魔术成为可能。

Perl 的标量变量保存单个值；数组保存一个有序的标量列表；散列保存一个无序的标量集合作为值，字符串作为键。尽管标量可以为任意字符串，但是该字符串允许我们在数组或数列中对复杂数据进行编码，这三种数据类型都不适合用于表示复杂的数据关系。这是引用所擅长的工作。我们从一个示例开始，探究引用的重要性。

4.1 在多个数组上完成相同任务

在 Minnow 离开港口做短途旅行之前（例如，一个三小时的旅程），我们需要检查每个乘客和全体船员，以确保他们都携带旅途中所需要的物品。为了海上安全，每个登上 Minnow 的人需要有一套救生用具、一瓶防晒霜、一个水壶和一件雨衣。如下所示，我们可以先编写一些代码，检查船长（Skipper）所携带的供应物品。

```
my @required = qw(preserver sunscreen water_bottle jacket);
my %skipper = map { $_, 1 }
    qw(blue_shirt hat jacket preserver sunscreen);

foreach my $item (@required) {
    unless ( $skipper{$item} ) { # not found in list?
        print "Skipper is missing $item.\n";
    }
}
```

注意，我们通过船长提供的物品列表创建了一个散列。这是常见并且有用的操作。如果我们想要检查某个特定物品是否在船长提供的列表上，最简单的方法是使所有的物品作为散列的键，然后使用 `exist` 命令检查散列。现在，我们向每个 key 提供一个 true

值，因此我们在此时不必使用 exist 命令。我们可以通过对列表中的元素使用 map 命令创建一个散列，而不必完整输入散列的全部内容。

如果需要检查 Gilligan 和 Professor，就需要编写如下所示的代码：

```
my %gilligan = map { $_, 1 } qw(red_shirt hat lucky_socks water_bottle);
foreach my $item (@required) {
    unless ( $gilligan{$item} ) { # not found in list?
        print "Gilligan is missing $item.\n";
    }
}

my %professor = map { $_, 1 }
    qw(sunscreen water_bottle slide_rule batteries radio);
for my $item (@required) {
    unless ( $professor{$item} ) { # not found in list?
        print "The Professor is missing $item.\n";
    }
}
```

如果我们按照上面的方式编写代码，就开始发现有太多重复的代码，我们可以考虑将上面的代码重构，并且放入一个通用的子例程中，这样就可以重用该子例程：

```
sub check_required_items {
    my $who = shift;
    my %whos_items = map { $_, 1 } @_ ; # the rest are the person's items

    my @_required = qw(preserver sunscreen water_bottle jacket);

    for my $item (@required) {
        unless ( $whos_items{$item} ) { # not found in list?
            print "$who is missing $item.\n";
        }
    }
}

my @_gilligan = qw(red_shirt hat lucky_socks water_bottle);
check_required_items('gilligan', @_gilligan);
```

Perl 通过参数列表，也就是 @_ 数组，向子例程传递 5 项，首先：字符串‘gilligan’和属于 @_gilligan 数组的 4 项。在 shift 语句之后， @_ 数组只包含这些项。因此 grep 针对列表检查每一个需要的项。



注意

我们在 perlsub 文档和 *Learning Perl* 的第 4 章中介绍 Perl 的子例程。

到目前为止还不错。我们使用略微多一些的代码检查 Skipper 和 Professor：

```
my @_skipper = qw(blue_shirt hat jacket preserver sunscreen);
my @_professor = qw(sunscreen water_bottle slide_rule batteries radio);
check_required_items('skipper', @_skipper);
check_required_items('professor', @_professor);
```

对于其他乘客，我们按照需要重复操作。尽管这些代码符合初始需要，我们还是遇到两个待解决的问题：

- 为了创建`@_`数组，Perl 复制了我们希望检查的全部数组内容。这样的操作对于少量数据还可以接受，但是如果数据量非常大，这样大规模地复制数据，仅仅是为了传递数据到子例程就显得有些浪费。
- 假定我们想要修改原始数组，强迫供应列表包含强制性项。因为我们在子例程中有一个副本（按值传递），所以任何对于`@_`数组的修改将不会自动影响相应的供应项数组。



注意

在 `shift` 语句修改已传递的相应变量之后，将新标量赋值给`@_`数组的元素，但这仍然不允许我们通过额外的强制性供应项扩展数组。

为了解决这类问题，我们需要按引用传递而不是按值传递。并且这也是 doctor（或者 Professor）的要求。

4.2 PeGS：Perl 图形结构

然而，在开始讨论引用之前，我们希望先介绍 Perl 图形结构（Perl Graphical Structure），也叫做 PeGS。Perl 数据结构的这些图形表示方式由 Joseph Hall 开发。通过一张漂亮的图片和一些图解说明数据的分布要比简单的文字描述要好。

大多数 PeGS 图由两部分组成：变量名称和该变量引用的数据。名称部分在图的顶部，由一个方框和一个指向右边的尖角构成（参考图 4-1）。变量名放置于方框中。



图 4-1 显示标识符部分的局部 PeGS 图

对于标量，用单个方框放置于变量名的下方，方框内为该标量的单一值（参考图 4-2）。

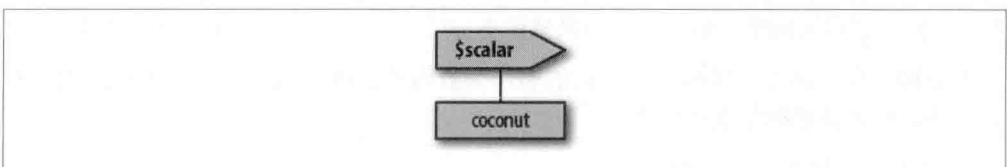


图 4-2 标量的 PeGS 图

对于数组而言（见图 4-3），因为数组可以拥有多个值，所以我们要多做一些。数据部分有一个填充的实心带状线置于顶部，作为集合的标志（以区别单一元素的数组和标量）。

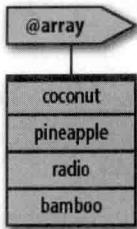


图 4-3 数组的 PeGS 图

散列就更精美了。与数组类似，数据部分从一个黑色的带状线开始，但该带状线下方有两部分：左边的是键，放在一个指向右边的尖角方框中，而右边的方框内是与键相对应的值（参考图 4-4）。

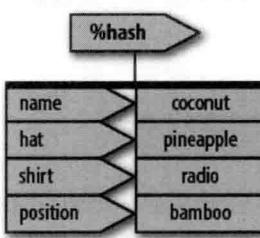


图 4-4 散列的 PeGS 图

我们用这些图绘制一些复杂的数据结构，随着我们继续深入学习，后续将介绍 PeGS 的一些其他特性。

4.3 对数组取引用

反斜杠 (\) 字符有很多含义，其中的一个就是作为“取引用”操作符。当我们将它放置于数组名称之前时，例如：\@skipper，将得到该数组的引用。数组的引用和指针类似^{注1}：但是引用指向整个数组，而不是数组本身的第一个元素的地址（参考图 4-5）。

引用适用于所有适合标量的场景。它能够作为数组或者散列的元素，或者按照如下所示方法，放入普通标量变量中：

```
my $ref_to_skipper = \@skipper;
```

注 1：我们并不是在 C 语言中探讨指针，而是从狗的感知角度探讨，考虑用英语指向鸭子的位置，而不是指向内存地址。

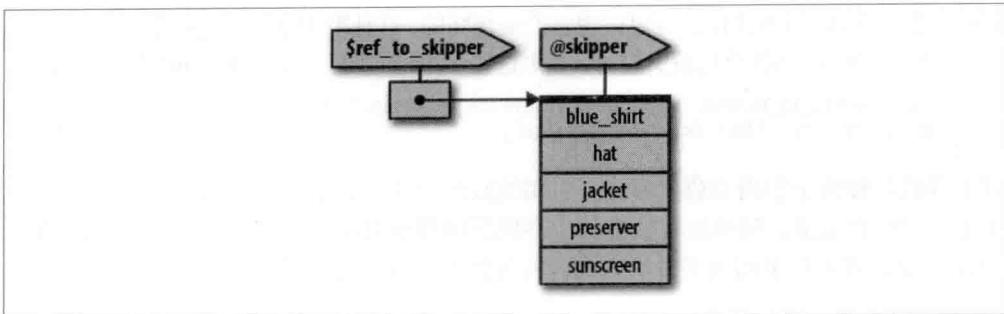


图 4-5 引用的 PeGS 图

在 PeGS 符号中，引用就是标量，因此引用的图看起来就像是标量。然而，在数据部分，引用指向所引用的数据。注意，从引用指向数据的箭头特别指向数据，而不是同样指向引用该数据的变量名称。马上这将会变得非常重要。

我们能够复制引用到另一个引用，这两个引用都指向同一数据（参考图 4-6）。

```
my $second_ref_to_skipper = $reference_to_skipper;
```

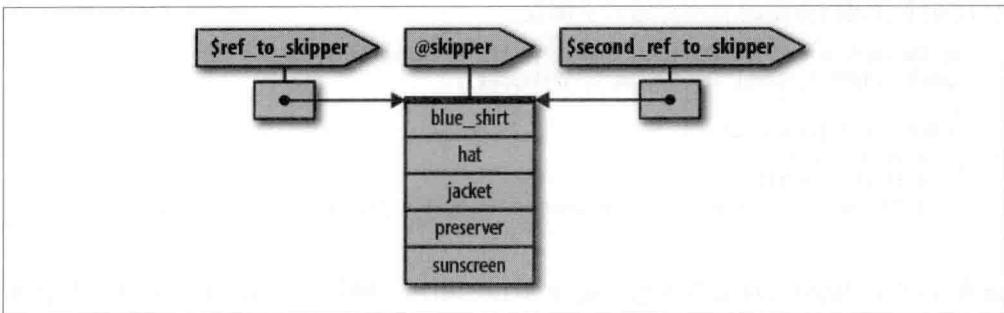


图 4-6 @skipper 数组和另一个指向同一数据的引用

我们甚至可以再次这样做（参考图 4-7）。

```
my $third_ref_to_skipper = \@skipper;
```

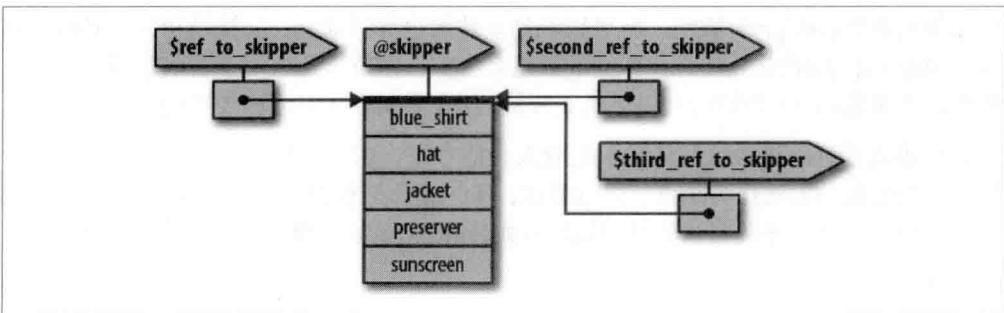


图 4-7 @skipper 数组和其他两个指向同一数据的引用

我们可以互换这三个引用，甚至可以说它们是相同的，这是因为他们指向完全相同的事物。当我们使用“==”操作符比较引用时，如果这些引用指向同一数据地址，返回值就为 true：

```
if ($reference_to_skipper == $second_reference_to_skipper) {  
    print "They are identical references.\n";  
}
```

以上等式比较两个引用的数值形式。引用的数值形式是@skipper 数组在内部数据结构中唯一的内存地址，该地址在数据的生存周期内都不会改变。如果我们使用 eq 操作符或者 print，查看该引用的字符串形式，将得到如下所示的结果：

```
ARRAY(0xa2b3c)
```

该数组中的字符串形式是唯一的，因为它包含该数组十六进制形式的内存地址。调试字符串也将它标注为数组引用。如果在我们的程序的输出中看到这样的内容，这通常意味着程序存在 bug；程序的用户可对十六进制形式的存储地址一点兴趣都没有！

同样，引用需要指向 Perl 内存中相同的数据，而不是在内存中恰好有相同值的两个不同数据。

因为我们复制一个引用，并且将该引用作为参数的别名向子例程传递，所以我们可以使用如下代码将数组的引用传递给子例程。

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);  
check_required_items("The Skipper", \@skipper);  
  
sub check_required_items {  
    my $who = shift;  
    my $items = shift;  
    my @required = qw(preserver sunscreen water_bottle jacket);  
    ...  
}
```

现在子例程中的\$items 是指向@skipper 数组的引用。但是我们如何从引用中获取初始数组呢？这就需要对该数组引用进行解引用操作。

4.4 对数组引用进行解引用操作

如果我们查看@skipper 数组，就可以看到该数组包含两部分：@ 符号和数组名称。同样，\$skipper[1] 的语法是由位于中间的数组名称和一些位于两边用于获取数组第二个元素的语法组成（因为数组的索引值从 0 开始，所以索引值 1 对应该数组的第二个元素）。

这就是难点：可以将数组的任意引用放入大括号中，用于替换数组名称，最后以一个访问原始数组的方法作为结束。这就是说，无论在什么位置使用 skipper 命名数组，都可以使用在大括号中放置该数组引用的方式 \${\$items} 替换。例如，下面这些行都指向整个数组。

```
@ skipper  
@{ $items }
```

以下两种方法都指向数组的第二个元素^{注2}:

```
$ skipper [1]
${ $items }[1]
```

通过使用引用的形式，我们已经从实际数组中对代码和访问数组的方法进行解耦。下面查看修改子例程剩余部分的方法:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;

    my %whos_items = map { $_, 1 } @{$items};

    my @required = qw(preserver sunscreen water_bottle jacket);
    for my $item (@required) {
        unless ( $whos_items{$item} ) { # not found in list?
            print "$who is missing $item.\n";
        }
    }
}
```

我们所做的是用@{\$items}替换 @_ (供应列表的副本)，对初始供应数组的引用进行解引用操作。现在，可以按照如下所示和之前一样调用几次该子例程:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
check_required_items('The Skipper', \@skipper);

my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
check_required_items('Professor', \@professor);

my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
check_required_items('Gilligan', \@gilligan);
```

以上每个情况下，\$items 指向不同的数组，因此每次调用同样的代码应用于不同的数组。这就是引用的其中一个最重要的用途：从代码所操作的数据结构中解耦出代码，这样使我们更容易重用代码。

通过引用传递数组解决了之前提到的两个问题中的第一个问题。现在，不是将整个供应列表复制到 @_ 数组中，而是通过传递单个元素，该元素是指向供应数组的引用。

可以在子例程的开始部分减少两个 shift 操作吗？当然可以，但这样就牺牲了程序的可读性：

```
sub check_required_items {
    my %whos_items = map { $_, 1 } @{$_[1]};

    my @required = qw(preserver sunscreen water_bottle jacket);
    for my $item (@required) {
        unless ( $whos_items{$item} ) { # not found in list?
            print "${_}[0] is missing $item.\n";
        }
    }
}
```

注 2：我们在这两个显示内容之间添加空格，使类似的部分排列整齐。这些空格在程序中是合法的，哪怕大多数程序不使用它。

我们在 @_ 数组中仍然有两个元素。第一个元素是乘客或者船员的名字，我们将在错误消息中使用该名字。第二个元素是一个指向正确供应数组的引用，我们将在 grep 表达式中使用该引用。

4.5 去除大括号

大多数时候，我们希望解引用的数组引用是一个简单的标量变量，例如 @{\$items} 或者 \${\$items}[1]。在下例中，去掉大括号，明确地构造 @\$items 或者 \$\$items[1]。

然而，如果大括号内的值不是由一个或者多个 \$ 符号引导的裸字标识符，我们就不能删除大括号。例如，对于最后一个子例程改写的 @{\$_[1]}，我们就不能移除大括号。这是对于数组中单一元素的访问，而不是标量变量，因此它包含更多的标识符和 \$ 符号。尽管如此，我们可以从 @{\$items} 和 @{\$\$items} 删除大括号。

该规则也意味着我们很容易分辨在哪里放置“丢失的”大括号。当我们看到 \$\$items[1] 这一种比较繁琐的语法时，我们可以知道大括号必须放在简单的标量变量 \$items 两侧。因此，\$items 必须是一个数组的引用。

因此，如下所示为该子例程的一个更易阅读的版本。

```
sub check_required_items {
    my $who    = shift;
    my $items  = shift;

    my @required = qw(preserver sunscreen water_bottle jacket);
    foreach my $item (@required) {
        unless (grep $item eq $_, @$items) { # not found in list?
            print "$who is missing $item.\n";
        }
    }
}
```

此处的不同之处在于移除了 @\$items 两侧的大括号。

4.6 修改数组

在之前的示例中，通过数组的引用解决了过多复制的问题。现在，我们开始修改初始数组。

对于每个遗失的供应项，将它们放入一个数组中，并且提醒乘客考虑其中的每一项：

```
sub check_required_items {
    my $who    = shift;
    my $items  = shift;

    my %whose_items = map { $_, 1 } @$items;
```

```

my @required = qw(preserver sunscreen water_bottle jacket);
my @missing = ( );

for my $item (@required) {
    unless ( $whose_items{$item} ) { # not found in list?
        print "$who is missing $item.\n";
        push @missing, $item;
    }
}

if (@missing) {
    print "Adding @missing to @$items for $who.\n";
    push @$items, @missing;
}
}

```

注意@missing 数组中额外添加的项。如果我们在扫描过程中找到任何丢失的项，就将这些丢失的项放入@missing 数组中。如果在扫描结束时还有任何其他项，我们将这些项添加到初始供应列表中。

关键就在于该子例程的最后一行。我们对\$items 数组引用进行解引用操作，访问初始数组，并且添加来自于@missing 数组的元素。如果没有按引用传递，我们就不得不修改数据的本地副本，而这将对于初始数组没有任何影响。

同样，@\$items（或者它更通用的形式@{\$items}）在双引号引用的字符串内也可以工作，并且可以像常见的命名数组引用一样进行插入操作。在正常的 Perl 代码中，尽管我们能够在大括号中放入任意空格，但此时我们不能在@符号和后面紧跟的字符之间插入空格。

4.7 嵌套的数据结构

在下一个示例中，@_数组包含两个元素，其中一个是数组的引用。如果我们对一个数组取引用，而该数组却包含另一个数组的引用，将会怎样？我们最终得到一个复杂的数据结构，它可能非常有用。

例如，我们能够遍历 Skipper、Gilligan 和 Professor 的数据，方法是先建立一个包含整个供应清单列表的大型数据结构：

```

my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my @skipper_with_name = ('Skipper' => \@skipper);

my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
my @professor_with_name = ('Professor' => \@professor);

my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
my @gilligan_with_name = ('Gilligan' => \@gilligan);

```

此时，@skipper_with_name 数组有两个元素，其中第二个元素是数组的引用，这与传递至子例程的内容类似。现在，把它们组合起来：

```
my @all_with_names = (
    \@skipper_with_name,
    \@professor_with_name,
    \@gilligan_with_name,
);
```

在@all_with_names 数组中有三个元素，其中的每一个元素都是数组引用，并且每个数组引用包含两个元素：人员名称和相应的初始供应列表。具体内容如图 4-8 所示。

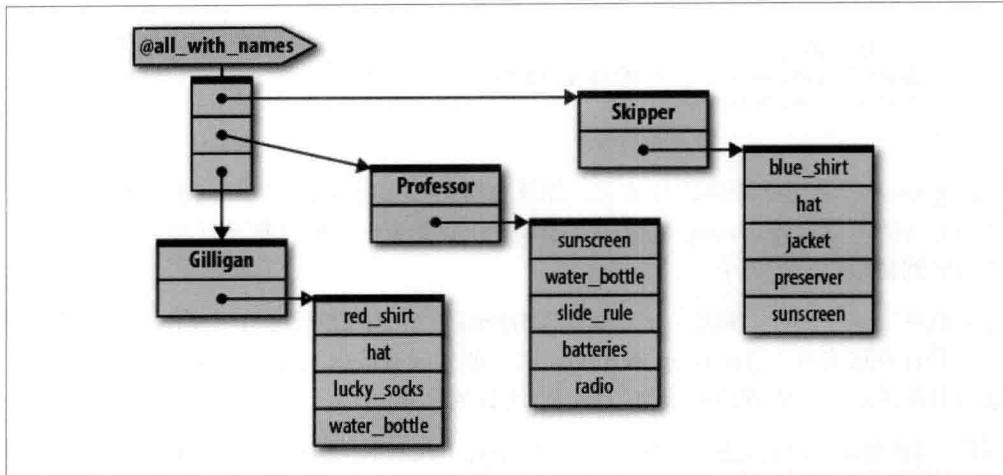


图 4-8 @all_with_names 数组拥有一个包含字符串和数组引用的多层次数据结构

因此，\$all_with_names[2]用来存储 Gilligan 的数据的数组引用。如果以 @{\$all_with_names[2]} 的方式对其进行解引用操作，就将得到一个拥有两个元素的数组“Gilligan”和另一个数组引用。

我们如何访问数组引用呢？再次运用之前提到的规则：以 \${\$all_with_names[2]}[1] 的方式。换句话说，在一个表达式中，对于 \$all_with_names[2]，我们对其进行解引用操作，使用 \${\$all_with_names[2]} 替换 DUMMY。

那么如何用这个数据结构调用现有的 check_required_items()子例程呢？如下所示的代码已经非常简单：

```
for my $person (@all_with_names) {
    my $who = $$person[0];
    my $provisions_reference = $$person[1];
    check_required_items($who, $provisions_reference);
}
```

这不需要对于子例程做任何修改。随着循环过程的进行，控制变量 \$person 遍历 \$all_with_names[0]、\$all_with_names[1] 和 \$all_with_names[2] 中每个元素的内容。当对 \$\$person[0] 解引用时，将分别得到“Skipper”、“Professor”和“Gilligan”。\$\$person[1] 是每个人的供应列表中相对应数组的引用。

因为整个解引用数组与参数列表精确匹配，所以同样可以按照如下方式简化列表：

```
for my $person (@all_with_names) {  
    check_required_items(@$person);  
}
```

或者甚至可以：

```
check_required_items(@$_) for @all_with_names;
```

不同层次的优化会导致混淆。我们需要考虑当一个月之后重新阅读自己的代码时，应当如何理解，如果这还不够，就需要考虑一旦当我们离职之后，有新人接替我们的工作时，是否能够看懂这段代码^{注3}。

4.8 用箭头简化嵌套元素的引用

再次查看解引用时所使用的大括号。在之前的示例中，Gilligan 供应列表的数组引用是\${\$all_with_names[2]}[1]。现在，如果我们想知道 Gilligan 的第一个供应项，该怎么操作呢？我们需要对于该项的下一层级进行解引用操作，因此就需要加上一个层级的大括号：\${\${\$all_with_names[2]}[1]}[0]。这肯定是非常繁琐的语法。我们能够简化吗？当然能！

在编写\${DUMMY}[\$y]的任何地方，都可以使用 DUMMY->[\$y]这种方式代替。换句话说，可以按照如下方式对一个引用数组进行解引用操作：通过在表达式后面用一个箭头和一个带方括号的下标定义数组引用，就可以选取数组中一个特定的元素。

对于本示例，这意味着对于 Gilligan，能够通过简单的\$all_with_names[2]->[1]语句，获取数组引用，并且 Gilligan 的第一个供应项为\$all_with_names[2]->[1]->[0]。这样看起来确实好多了！

如果你觉得还不够简洁，还有一条规则：如果箭头位于“下标类的符号”之间，例如方括号，因为多重下标已经表达了解引用，所以也可以删除这些箭头。例如，\$all_with_names[2]->[1]->[0]可以表达为\$all_with_names[2][1][0]。这样看起来更好了。

箭头必须放置于非下标之间。它为什么不能放置于下标之间呢？如下例所示，假定@all_with_names 数组的引用为：

```
my $root = \@all_with_names;
```

现在如何获取 Gilligan 的第一项？按照下列方法排列下标：

```
$root -> [2] -> [1] -> [0]
```

进一步简化，使用“删除箭头”的规则，可以使用：

注3：O'Reilly 出版社有一本很棒的书有助于编写可复用的代码：*Perl Best Practices*，由 Damian Conway 编写，该书有 256 个诀窍用于编写更具可读性和可维护性的 Perl 代码。

```
$root -> [2][1][0]
```

然而，不能删除第一个箭头，因为这样做就意味着指向数组@root 的第三个元素，一个完全不相关的数据结构。再次用这种形式与完整的大括号形式相比较：

```
$${$root}[2][1][0]
```

使用箭头的形式看起来好很多。然而，要注意，没有通过数组引用获取整个数组的快捷方式。如果想要访问整个 Gilligan 的供应列表，必须按照如下方式：

```
@{$root->[2][1]}
```

按照如下顺序从内部读取数据，可以按照如下方法实现：

1. 提取\$root。
2. 以数组引用的方式对\$root 解引用，获取该数组中的第三个元素（索引编号为 2）：
\$root->[2]
3. 以数组引用的方式对\$root->[2]解引用，获取该数组中的第二个元素（索引编号为 1）：
\$root->[2][1]
4. 以数组引用的方式对\$root->[2][1]解引用，获取整个数组：@{\$root->[2][1]}

最后一步没有使用箭头形式的快捷方式。

4.9 散列的引用

与可以对数组取引用一致，也可以对散列取引用。再一次使用反斜杠作为“取引用”操作符，然后将结果存入标量（参考图 4-9）。

```
my %gilligan_info = (  
    name    => 'Gilligan',  
    hat     => 'White',  
    shirt   => 'Red',  
    position => 'First Mate',  
);  
my $hash_ref = \%gilligan_info;
```

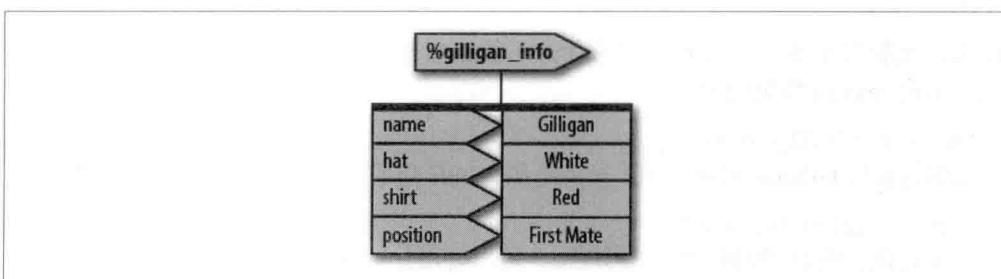


图 4-9 %gilligan_info 散列的 PeGS 结构

我们能够对一个散列引用进行解引用操作以获取初始数据，操作策略与对数组引用进行解引用一致。我们按照没有使用引用的方式编写读取散列中数据的语法，然后用一对大括号包围着的引用名称替换散列名称。例如，对于给定的键挑选一个特殊的值，可以按照如下方式实现：

```
my $name = $ gilligan_info { 'name' };
my $name = $ { $hash_ref } { 'name' };
```

此时，大括号有两层不同的含义。第一组大括号表示其中的表达式返回值为引用，第二组大括号定界其中的表达式作为散列的键。

将操作应用于整个散列，按照如下方式进行。

```
my @keys = keys % gilligan_info;
my @keys = keys % { $hash_ref };
```

作为数组引用，我们可以在某些场景下使用快捷方式替换复杂的大括号形式。例如，如果只是将一个简单的标量变量放置于大括号中（正如到目前为止所展示的示例），就可以按照如下方式删除大括号。

```
my $name = $$hash_ref{'name'};
my @keys = keys %$hash_ref;
```

与数组引用类似，当需要获取一个特定散列元素时，也能够使用箭头形式（参考图 4-10）。

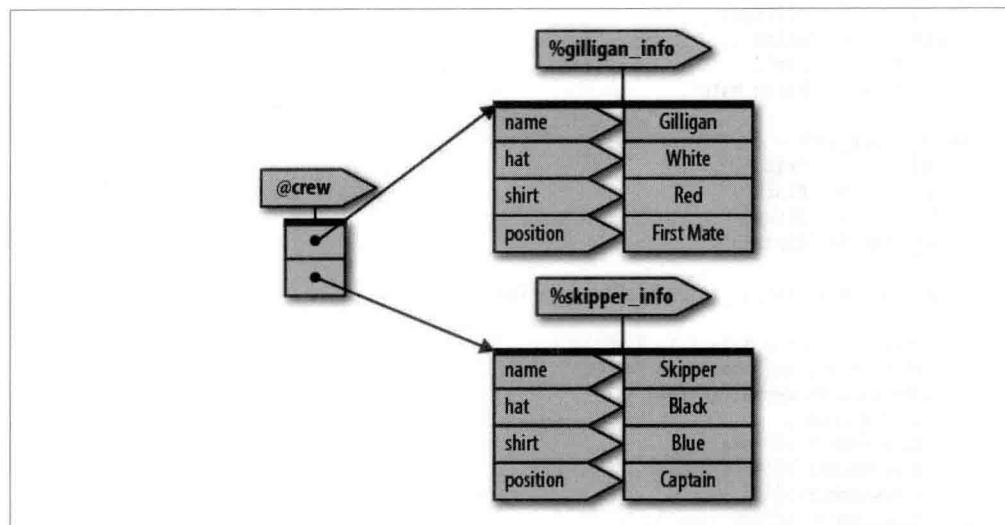


图 4-10 船员花名册的 PeGS

```
my $name = $hash_ref->{ 'name' };
```

因为散列引用适合于一切标量适合的场景，所以可以按照如下方式创建一个散列引用数组：

```

my %gilligan_info = (
    name      => 'Gilligan',
    hat       => 'White',
    shirt     => 'Red',
    position  => 'First Mate',
);
my %skipper_info = (
    name      => 'Skipper',
    hat       => 'Black',
    shirt     => 'Blue',
    position  => 'Captain',
);
my @crew = (\%gilligan_info, \%skipper_info);

```

因此，\$crew[0]是一个指向关于 Gilligan 信息的散列引用。能够通过以下任意一个表达式获取 Gilligan 的名称：

```

${ $crew[0] }{ 'name' }
my $ref = $crew[0]; $$ref{'name'}
$crew[0]->{'name'}
$crew[0]{'name'}

```

通过最后两个表达式，可以在“下标类的符号”之间删除箭头，即使一个是数组的方括号，另一个是散列的大括号。

按照如下方式输出船员花名册：

```

my %gilligan_info = (
    name      => 'Gilligan',
    hat       => 'White',
    shirt     => 'Red',
    position  => 'First Mate',
);
my %skipper_info = (
    name      => 'Skipper',
    hat       => 'Black',
    shirt     => 'Blue',
    position  => 'Captain',
);
my @crew = (\%gilligan_info, \%skipper_info);

my $format = "%-15s %-7s %-7s %-15s\n";
printf $format, qw(Name Shirt Hat Position);
foreach my $crewmember (@crew) {
    printf $format,
        $crewmember->{'name'},
        $crewmember->{'shirt'},
        $crewmember->{'hat'},
        $crewmember->{'position'};
}

```

最后的部分看起来是重复的。我们能够使用散列切片将它们缩减。如下所示，如果初始语法再一次为：

```

@ gilligan_info { qw(name position) }

```

如下所示，来自于引用的散列切片记号为：

```
@ { $hash_ref } { qw(name position) }
```



注意

可以参考 *Learning Perl* 第 17 章，复习散列切片的知识。这些知识也同样记录于 perldata 文档中。

如下所示，因为大括号中是一个简单的标量值，所以可以删除第一组大括号，得到：

```
@ $hash_ref { qw(name position) }
```

因此，可以用如下语句替换最终的循环：

```
for my $crewmember (@crew) {
    printf $format, @{$crewmember}{qw(name shirt hat position)};
}
```

对于数组切片或散列切片，没有带箭头 (\rightarrow) 的快捷方式。

直接以字符串的形式输出散列引用，得到的结果类似于 HASH (0x1a2b3c)，显示该散列的十六进制内存地址。这对于最终用户毫无用处，并且对程序员也几乎没有用处，除非作为没有适当的解引用操作的回应。

4.10 检查引用类型

一旦我们开始使用和传递引用时，就必须确保我们知道正在使用哪种类型的引用。如果我们没有按照正确的类型使用引用，程序就将乱套了：

```
show_hash( \@array );

sub show_hash {
    my $hash_ref = shift;

    foreach my $key ( %$hash_ref ) {
        ...
    }
}
```

show_hash 子例程期望传入一个散列引用作为参数，并且信任我们传入的参数。然而，因为传递的是一个数组引用，所以程序就乱套了：

```
Not a HASH reference at line ...
```

如果我们想要更仔细一些，就应当检查传入 show_hash 子例程的参数，确保传入的确实是散列的引用。有很多方法可以做到，最简单的是使用 ref 函数，该内置函数的返回值为引用类型。按照如下方法比较 ref 函数的返回值和期望传入的值：

```
use Carp qw(croak);

sub show_hash {
    my $hash_ref = shift;
    my $ref_type = ref $hash_ref;
```

```
croak "I expected a hash reference!"
unless $ref_type eq 'HASH';

foreach my $key ( %$hash_ref ) {
    ...
}
```

然而，这样看起来有点奇怪，因为我们不得不对于 HASH 字面量字符串进行硬编码。我们之前从不喜欢这样做。然而，我们能够通过两次使用 ref 函数去除字面量字符串。如下所示，我们通过所期望的引用类型的一个普通版本，再次调用 ref 函数：

```
croak "I expected a hash reference!"
unless $ref_type eq ref {};
```

另外，我们能够使用 constant 模块存储散列引用字符串：

```
use constant HASH => ref {};
croak "I expected a hash reference!"
unless $ref_type eq HASH;
```

使用的 constant 模块看起来很像字面量字符串，但它有很大的不同：如果使用未定义的错误名称，常量就将运行失败，但错误的字面量字符串永远不会运行失败，这是因为 Perl 永远不可能知道我们使用了错误的字符串。

使用 ref 函数带来另一个问题，直到介绍对象之前，我们不能对此做完整的解释。因为 ref 函数返回一个提供引用类型的字符串，如果有一个表现类似于散列引用的对象，程序将运行失败，因为字符串是不同的。Perl 自带的 Scalar::Util 模块，能够使用完成同样工作的 reftype 函数，避免这类事情发生：

```
use Carp qw(croak);
use Scalar::Util qw(reftype);

sub show_hash {
    my $hash_ref = shift;
    my $ref_type = reftype $hash_ref; # works with objects
    croak "I expected a hash reference!"
    unless $ref_type eq ref {};
}

foreach my $key ( %$hash_ref ) {
    ...
}
```

然而，对象是接口。因此不是基于散列引用的事物，仍然能够表现得像散列一样。如果那样，ref 函数不必返回 HASH 字符串。

我们不是问：“你是什么？”而是问：“你能做什么？”我们真的只想知道 show_hash 的参数是否表现得像散列引用，那样就不会乱套了。我们没有特别在意它是否确实是散列引用。

如果那样，我们可能使用 eval 函数做类似于散列的事情。如果 eval 函数运行失败并且返回值为 false，如下所示，我们将不会得到一个散列：

```
croak "I expected a hash reference!"
unless eval { keys %$ref_type; 1 }
```

如果我们期望经常检查，就可能需要将检查包装在它自己的子例程中：

```
sub is_hash_ref {
    my $hash_ref = shift;

    return eval { keys %$ref_type; 1 };
}

croak "I expected a hash reference!"
unless is_hash_ref( $ref_type );
```

4.11 练习

可以在附录中的“第 4 章答案”部分找到这些练习的答案。

1. [5 分钟]下列各式指向多少个不同的表达式？为每个表达式绘制 PeGS 结构图。

```
$ginger->[2][1]
${$ginger[2]}[1]
$ginger->[2]->[1]
${$ginger->[2]}[1]
```

2. [30 分钟]使用 `check_required_items` 子例程的最终版本，编写一个 `check_required_items_for_all` 子例程，该子例程调用的唯一参数是一个散列的引用，该散列的键为在 Minnow 上的人，相应的值为这些人需要带上 Minnow 的物品的数组引用。

例如，散列引用可能按照如下方式构造：

```
my @gilligan = (... gilligan items ...);
my @skipper = (... skipper items ...);
my @professor = (... professor items ...);

my %all = (
    Gilligan => \@gilligan,
    Skipper => \@skipper,
    Professor => \@professor,
);
```

```
check_items_for_all(\%all);
```

最新构造的子例程应当为散列中的每个人调用 `check_required_items` 子例程，更新这些人的供应列表，用来加入需要的物品。

一些初始代码可通过 <http://www.intermediateperl.com/> 网站的下载页面下载。

3. [20 分钟]修改船员花名册程序，为每个遇难的乘客添加位置字段。在开始部分，设定每个人的位置为“The Island”。在每个人的散列中添加字段之后，修改 Howell 夫妇的位置为“The Island Country Club”。最后，按照如下方法为每个人的位置生成报表：

Gilligan at The Island

Skipper at The Island

Mr. Howell at The Island Country Club

Mrs. Howell at The Island Country Club

一些初始代码可通过 <http://www.intermediateperl.com/> 网站的下载页面下载。

引用和作用域

我们能够像任意标量一样复制并且传递引用。在任何时候，Perl 都知道指向某个特定数据项的引用数目。Perl 也能够创建没有显式名称的匿名数据结构的引用，并按照某种需求自动创建引用。我们将展示复制引用的方法，以及使用该方法将如何影响作用域及内存使用。

5.1 关于数据引用的更多信息

第 4 章讲述了对一个数组 @skipper 取引用并将该引用放置于一个新的标量变量中的方法：

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my $ref_to_skipper = \@skipper;
```

如下所示，此时可以复制该引用或者再次取引用，它们全部都指向同一数组并且可交换：

```
my $second_ref_to_skipper = $reference_to_skipper;
my $third_ref_to_skipper = \@skipper;
```

此时此刻，有 4 种不同的方式来访问保存在 @skipper 数组中的数据：

```
@skipper
$ref_to_skipper
$second_ref_to_skipper
$third_ref_to_skipper
```

Perl 通过一个叫做“引用计数”的机制追踪有多少种访问数据的方法。初始名称的个数为 1，创建的每个额外引用（包括引用的副本）的个数也为 1。因此对于当前该供应列表的引用总数为 4。

可以随意添加和删除引用，并且只要引用计数没有减少到 0，Perl 就将在内存中保留该数组，并且该数组仍然可以通过其他访问路径访问。如下所示，例如，可能需要一个

临时引用：

```
check_provisions_list(\@skipper)
```

当该子例程执行时，Perl 将创建第 5 个指向这个数组的引用，然后为子例程将该引用复制到 @_ 数组中。一旦 Perl 检测到需要时，子例程就会自由地为该引用创建额外的副本。一般而言，当子例程返回时，Perl 会自动丢弃所有这些引用，然后该数组的引用计数将再次变成 4 个。

当给变量赋的值不是指向数组 @skipper 的引用时，就能够销毁该引用。如下所示，例如，可以把 undef 赋给变量：

```
$ref_to_skipper = undef;
```

或者，可能使该变量超出作用域：

```
my @skipper = ...;  
{ # bare block  
...  
my $ref = \@skipper;  
...  
}  
} # $ref goes out of scope at this point
```

特殊情况下，保存在子例程的私有（词法）变量中的引用，将会在子例程的结尾销毁。

无论改变值，还是变量本身销毁，Perl 都将其记录为适当减少数据引用的次数。

仅当所有引用（包括数组的名称）销毁时，Perl 才会回收数组占用的内存。在上例中，当 @skipper 和它创建的所有引用都消失后，Perl 才回收内存。

Perl 会在本程序后续调用的其他数据中使用这些已经释放的内存中的数据，并且通常情况下，Perl 不会将这些内存中的数据返还给操作系统。

5.2 如果它曾是变量名将会怎样

通常情况下，变量的所有引用在变量销毁之前都销毁了。但是如果其中一个引用比变量名存在的更久将会怎样？例如，考虑运行如下代码将会得到的输出：

```
my $ref;  
{  
    my @skipper = qw(blue_shirt hat jacket preserver sunscreen); # ref count is 1  
    $ref        = \@skipper; # ref count is 2  
  
    print "$ref->[2]\n"; # prints jacket\n  
}  
  
print "$ref->[2]\n"; # still prints jacket\n # ref count is 1
```

如上例所示，在声明@skipper 数组之后，紧接着有一个指向该 5 个元素列表的引用。在初始化\$ref 之后，直到语句块的结束，我们将拥有该数组的两个引用。当语句块结束时，数组@skipper 的名称也销毁。然而，这是两个访问数据的方法之一！因此，5 个元素的列表仍然存在于内存中，并且\$ref 仍然指向该列表中的数据。

此时此刻，该 5 个元素的列表为一个匿名数组，匿名数组是一个专门术语，用于指代没有名称的数组。

当数组名称消失后，我们仍然能够使用之前使用过的所有解引用策略对数组进行操作，除非\$ref 的值改变，或者\$ref 本身消失。该数组仍然是一个功能齐全的数组，我们可以像对待任何其他 Perl 数组一样对它删减或者增加元素：

```
push @$ref, 'sextant'; # add a new provision  
print "$ref->[-1]\n"; # prints sextant\n
```

此时此刻，我们甚至可以按照如下方法增加引用计数：

```
my $copy_of_ref = $ref;
```

上式等价于：

```
my $copy_of_ref = \@$ref;
```

这些数据将一直存在，直到我们销毁了最后指向这些数据的引用：

```
$ref = undef; # not yet...  
$copy_of_ref = undef; # poof!
```

5.3 引用计数和嵌套数据结构

直到最后一个引用被销毁，数据一直存在，即使在一个大型的活动数据结构中的引用也是如此。假定数组元素是它本身的引用，回想一下第 4 章见过的示例：

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);  
my @skipper_with_name = ('The Skipper' => \@skipper);  
  
my @professor = qw(sunscreen water_bottle slide_rule batteries radio);  
my @professor_with_name = ('The Professor' => \@professor);  
  
my @gilligan = qw(red_shirt hat lucky_socks water_bottle);  
my @gilligan_with_name = ('Gilligan' => \@gilligan);  
  
my @all_with_names = (  
    \@skipper_with_name,  
    \@professor_with_name,  
    \@gilligan_with_name,  
,);
```

想象一下中间变量是所有子例程的一部分：

```
my @all_with_names;  
  
sub initialize_provisions_list {
```

```

my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my @_skipper_with_name = ('The Skipper' => \@skipper);

my @_professor = qw(sunscreen water_bottle slide_rule batteries radio);
my @_professor_with_name = ('The Professor' => \@professor);

my @_gilligan = qw(red_shirt hat lucky_socks water_bottle);
my @_gilligan_with_name = ('Gilligan' => \@gilligan);

@all_with_names = ( # set global
  @_skipper_with_name,
  @_professor_with_name,
  @_gilligan_with_name,
);
}

initialize_provisions_list();

```

设定@all_with_names 数组包含三个引用。在该子例程内部，先将命名数组的引用放入其他命名数组中。最终，所有值都放置于全局数组@all_with_names 中。然而，一旦子例程返回，6 个数组的名称就被销毁，因为每个数组有另一个指向它们的引用，所以将引用计数临时设置为 2，但是当数组名称销毁之后，引用计数变为 1。因为引用计数不为零，所以该数据将继续存在，尽管它现在只是被@all_with_names 数组的元素所引用。

与其对全局变量赋值，还不如按照如下方式不用@all_with_names 数组重写代码，并且直接返回列表：

```

sub get_provisions_list {
  my @_skipper = qw(blue_shirt hat jacket preserver sunscreen);
  my @_skipper_with_name = ('The Skipper', \@skipper);

  my @_professor = qw(sunscreen water_bottle slide_rule batteries radio);
  my @_professor_with_name = ('The Professor', \@professor);

  my @_gilligan = qw(red_shirt hat lucky_socks water_bottle);
  my @_gilligan_with_name = ('Gilligan', \@gilligan);

  return (
    @_skipper_with_name,
    @_professor_with_name,
    @_gilligan_with_name,
  );
}

my @all_with_names = get_provisions_list( );

```

在上例中，所创建的最终保存在@all_with_names 数组中的值，是对子例程最后一条执行语句求值得到的。该子例程返回一个三元素列表。一旦子例程中的命名数组拥有至少一个指向它们的引用，并且该引用是返回值的一部分，数据就一直存在。如果改变或者丢弃@all_with_names 中的引用，Perl 将减少相应数组的引用计数。如果这意味着引用计数变为零（正如本例中所示），Perl 就销毁数组本身。因为@all_with_names 数组内部的每个数组元素也包含一个引用（例如指向@skipper 数组的引用），Perl 将它的引

用计数减少 1。一旦引用计数减少为 0，也就以层叠效应（cascading effect）的方式释放内存。

删除数据树顶端的节点通常就意味着删除该数据树包含的所有数据。例外情况是当对嵌套数据的引用添加额外的副本时，数据就不会丢失。例如，如下所示，如果复制 Gilligan 的供应列表：

```
my $gilligan_stuff = $all_with_names[2][1];
```

那么当删除@all_with_names 数组之后，我们仍然有一个指向之前的@gilligan 数组内容的实时引用，并且从该引用之后的数据内容依然保留。

底线是这样：Perl 总是做正确的事情。如果我们仍然拥有一个指向数据的引用，就将仍然拥有该数据。

5.4 当引用计数出现问题时

使用引用计数管理内存是一种盛行已久的方式，它的历史的确非常长。引用计数有一个缺点，就是当数据结构不是有向图时，并且当数据结构中的一些部分以循环的方式指向数据结构的其他部分时，就会出现问题。例如，假定两块数据结构都包含指向对方的引用（参考图 5-1）。

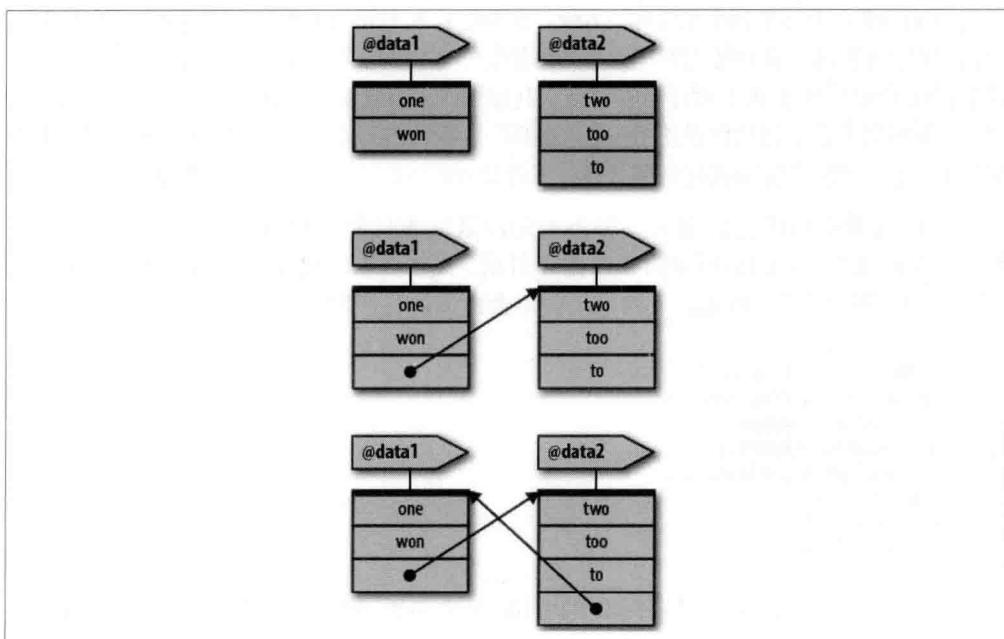


图 5-1 当数据结构中的引用形成一个循环时，Perl 的引用计数系统就可能无法识别和回收不再需要的内存空间

```
my @data1 = qw(one won);
my @data2 = qw(two too to);

push @data2, \@data1;
push @data1, \@data2;
```

此时此刻，对于@data1 中的数据，拥有两个名称：@data1 本身和 {@data2[3]}，并且在@data2 中的数据也有两个名称：@data2 本身和 {@data1[2]}。这样，就创建了一个循环，通过该循环，能够通过无限多个名称（例如\$data1[2][3][2][3][2][3][1]）访问 won。

当这两个数组名称超出作用域将发生什么？这两个数组的引用计数将会从 2 降为 1，但不会是 0！因为没有变为 0，所以 Perl 认为仍然可能有方法访问数据，尽管实际上没有。所以，我们创建了一个内存泄漏。随着时间的流逝，程序中的内存泄漏将导致程序消耗越来越多的内存。

此时此刻，你认为这个例子是有意设计出来的，确实如此，我们绝不会在真实程序中设计一个循环的数据结构！实际上，程序员通常将这类循环作为部分的双向链表、双向链接环或者其他一些数据结构或者甚至是不小心创建的循环。关键是 Perl 程序员基本不会犯这样的错误，最重要的原因是我们在 Perl 中使用这些数据结构。大多数处理内存管理和链接不连续的内存块的操作，Perl 已经自动完成。

如果你曾使用过其他语言，就可能注意到在 Perl 中编程相对更容易。例如，对于列表中的元素进行排序或者添加和删除元素都很方便，甚至是列表中间的元素。这些同样的任务用其他语言实现就很困难，并且使用循环数据结构是绕开这些语言限制的常见方法。我们为什么在这里提这些呢？因为 Perl 程序员有时会从其他编程语言复制一个算法。这样做本身没有错误，尽管最好考虑下初始作者使用一个“循环”数据结构的原因，并且使用 Perl 的优势重新编写该算法。可能你需要使用散列替换，或者放入数组的数据可能后期需要排序。

Perl 的后续版本可能会提出另一种不同的垃圾回收机制，额外添加或者替换引用计数^{注1}。直到现在，我们必须谨防创建循环引用，或者如果不得不这样做，在变量超出作用域之前打断“环”。例如，下列代码就不会造成内存泄漏：

```
{
    my @data1 = qw(one won);
    my @data2 = qw(two too to);
    push @data2, \@data1;
    push @data1, \@data2;
    ... use @data1, @data2 ...
    # at the end:
    @data1 = ();
    @data2 = ();
}
```

我们最终销毁了@data1 中对@data2 的引用，反之亦然。现在每个数据只有唯一的引用，

注 1：我们可能通过 Test::MemoryCycle 模块能够找到问题的原因。

在代码块末尾，引用数目将归零。我们能够清除任意一个引用而不影响另一个，程序依旧工作正常。第 18 章展示创建弱引用的方法，该方法能够有助于解决这类问题。

5.5 直接创建匿名数组

在之前的 `get_provisions_list` 程序中，创建了 6 个数组名称，仅仅是为了随后对它们执行取引用操作。当子例程退出时，数组名称将全部销毁，但引用仍将保留。

虽然创建临时命名数组在最简单的情况下可行，但是一旦数据结构变得更细节化，创建这些名称就变得越来越复杂。我们就不得不记住这些数组的名称，哪怕此后很快就要将它们忘掉。

可以通过缩小不同数组名称的作用域，降低命名空间的复杂性。通过创建一个临时语句块，而不是在子例程的作用域中声明它们：

```
my @skipper_with_name;
{
    my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
    @skipper_with_name = ('The Skipper', \@skipper);
}
```

此时此刻，`@skipper_with_name` 数组的第二个元素是指向之前定义的`@skipper` 数组的引用。然而，该名称已与此程序无关。

为了说明“第二个元素应为指向包含这些元素的数组的引用”，就显得很麻烦。可以创建一个直接使用匿名数组构造函数的值，这也是方括号的另一种用法：

```
my $ref_to_skipper_provisions =
    [ qw(blue_shirt hat jacket preserver sunscreen) ];
```

取方括号内的值（在列表上下文中求值）；为这些元素创建一个新的匿名数组；并且（这里是最重要的部分）返回该数组的引用。如前所述：

```
my $ref_to_skipper_provisions;
{
    my @temporary_name =
        ( qw(blue_shirt hat jacket preserver sunscreen) );
    $ref_to_skipper_provisions = \@temporary_name;
}
```

不必提出一个临时名称，也不需要创建一个不和谐的临时语句块。使用方括号匿名数组构造函数得到的结果是一个数组引用，该引用适用于标量变量适用的所有场合。

现在可以使用它构建更大的列表：

```
my $ref_to_skipper_provisions =
    [ qw(blue_shirt hat jacket preserver sunscreen) ];
my @skipper_with_name = ('The Skipper', $ref_to_skipper_provisions);
```

我们实际上也不需要这个临时标量。如下所示，可以将标量引用的内容放入数组中，作为更大数组的一部分：

```
my @skipper_with_name = (
    'The Skipper',
    [qw(blue_shirt hat jacket preserver sunscreen) ]
);
```

我们已经声明@skipper_with_name 数组，该数组的第一个元素是“*The Skipper*”的名称字符串，第二个元素是数组引用，通过把 5 个供应项放入数组并取引用得到。因此 @skipper_with_name 数组和之前一致，拥有两个元素。

不要对方括号中的圆括号感到迷惑。它们各自有着不同的目的。如果我们用圆括号替换方括号，就最终得到一个包含 6 个元素的列表。如果用方括号替换最外层圆括号（在第一行和最后一行），就将构造一个拥有两个元素的匿名数组，并且对该数组取引用作为最终 @skipper_with_name 数组的唯一元素^{注2}。因此，如果最终使用如下所示的语法：

```
my $fruits;
{
    my @secret_variable = ('pineapple', 'papaya', 'mango');
    $fruits = \@secret_variable;
}
```

就可以将上面的代码替换为：

```
my $fruits = ['pineapple', 'papaya', 'mango'];
```

上面的方法能够用于处理更复杂的数据结构吗？当然能！每当我们需要一个列表的元素成为数组的引用时，就可以通过匿名数组构造函数创建该引用。如下所示，也能够在供应列表中嵌套它们：

```
sub get_provisions_list {
    return (
        ['The Skipper',   [qw(blue_shirt hat jacket preserver sunscreen) ] ],
        ['The Professor', [qw(sunscreen water_bottle slide_rule batteries radio) ] ],
        ['Gilligan',      [qw(red_shirt hat lucky_socks water_bottle) ] ],
    );
}

my @all_with_names = get_provisions_list( );
```

由外到内查看子例程，我们能够得到一个包含三个元素的返回值。每个元素都是一个数组引用，指向一个有两个元素的匿名数组。该匿名数组的第一个元素是一个人员名称字符串，第二个元素是一个匿名数组引用，该长度不等的匿名数组包含所有供应项名称——对于任意中间层不必提出临时名称。

对于该子例程的调用方，返回值与之前的版本完全相同。然而，从维护的角度来看，不使用所有的中间名称将节省屏幕和大脑空间。

可以使用空匿名数组构造器函数，说明空匿名数组的引用。例如，如果需要添加一个“Mrs. Howell”到人员名称列表，假如此人的包裹相当“轻”，可以按照如下方式插入：

```
['Mrs. Howell',
 [ ]  # anonymous empty array reference
],
```

注 2：在教室中，当使用引用时，我们看到很多间接（或者不是完全间接）往往产生大多数常见错误。

这是大型列表中的单个元素。该元素是一个拥有两个元素的数组引用，第一个元素是人员名称字符串，第二个元素是一个指向空匿名数组的引用。第二个元素指向的数组为空，因为 Mrs. Howell 没有在旅程中携带任何物品。

5.6 创建匿名散列

与创建匿名数组类似，也能够创建匿名散列。如下所示，参考第 4 章中的船员花名册：

```
my %gilligan_info = (
    name      => 'Gilligan',
    hat       => 'White',
    shirt     => 'Red',
    position  => 'First Mate',
);

my %skipper_info = (
    name      => 'Skipper',
    hat       => 'Black',
    shirt     => 'Blue',
    position  => 'Captain',
);

my @crew = (\%gilligan_info, \%skipper_info);
```

变量`%gilligan_info` 和`%skipper_info` 是创建最终数据结构的散列所需的临时变量。我们能够使用匿名散列构造函数直接构造引用，而匿名散列构造函数也是大括号的另一层含义。可以按照如下方式定义散列，然后赋值给散列的引用，使用两个步骤：

```
my $ref_to_gilligan_info;

{
    my %gilligan_info = (
        name      => 'Gilligan',
        hat       => 'White',
        shirt     => 'Red',
        position  => 'First Mate',
    );
    $ref_to_gilligan_info = \%gilligan_info;
}
```

也可以按照如下方式，使用单个步骤替换：

```
my $ref_to_gilligan_info = {
    name      => 'Gilligan',
    hat       => 'White',
    shirt     => 'Red',
    position  => 'First Mate',
};
```

在左大括号和右大括号之间是一个 8 元素列表。该 8 元素列表成为一个包含 4 个元素

的匿名散列（4个键值对）。Perl对该散列取引用，并且作为单个标量值返回，我们将引用赋值给该标量值。因而，我们能够按照如下方式重写之前的船员花名册示例：

```
my $ref_to_gilligan_info = {  
    name      => 'Gilligan',  
    hat       => 'White',  
    shirt     => 'Red',  
    position  => 'First Mate',  
};  
  
my $ref_to_skipper_info = {  
    name      => 'Skipper',  
    hat       => 'Black',  
    shirt     => 'Blue',  
    position  => 'Captain',  
};  
  
my @crew = ($ref_to_gilligan_info, $ref_to_skipper_info);
```

和之前一致，现在可以避免使用临时变量，在顶层列表直接插入值：

```
my @crew = (  
    {  
        name      => 'Gilligan',  
        hat       => 'White',  
        shirt     => 'Red',  
        position  => 'First Mate',  
    },  
    {  
        name      => 'Skipper',  
        hat       => 'Black',  
        shirt     => 'Blue',  
        position  => 'Captain',  
    },  
);
```

注意，当列表末尾的元素并非紧接着右花括号、方括号或小括号时，可以以逗号结尾。可采取这个很好的编码风格，因为这样做有利于代码的维护。可以很方便地添加和重排行，或者在不破坏列表完整性的情况下注释掉代码行。

现在，@crew 数组与它之前的内容完全相同，但是我们不再需要为中间数据结构发明一些新的名字。和之前一样，@crew 数组包含两个元素，其中的每一个都是一个散列引用，该散列引用包含关于一个特定乘客的基于关键字的信息。

匿名散列构造函数总是在列表上下文中对它的内容求值，然后通过键值对构造一个散列，与我们将列表赋值给一个命名散列一样。Perl 将返回一个散列引用作为单个标量值，该散列引用适用于标量适用的任何场合。

现在，解析器是这样解释说明的：因为代码块和匿名散列构造函数都在语法树中几乎相同的位置使用大括号，编译器不得不猜测我们究竟想要使用哪一个。如果编译器曾经做过错误的决定，我们可能需要向编译器提供一个提示以明确获取想要的内容。如果向编译器说明我们想要一个匿名散列构造函数，就在左大括号之前放置一个加号：

+{ ... }。如果想得到一个代码块，就在语句块开始处放置一个分号（表示一条空语句）：
{; ... }。

5.7 自动带入

再次查看之前的供应列表。假定我们已经按照如下格式从一个文件中读取数据：

```
The Skipper
  blue_shirt
  hat
  jacket
  preserver
  sunscreen
Professor
  sunscreen
  water_bottle
  slide_rule
Gilligan
  red_shirt
  hat
  lucky_socks
  water_bottle
```

我们用一些空格缩进供应项，对人员名称的行没有缩进。这样，我们就构造出一个供应列表散列，散列的键为人员名称，值为一个包含一个供应列表数组的引用。

最初，可以通过一个简单的循环收集数据：

```
my %provisions;
my $person;

while (<>) {
    if (/^(\S.*)/) { # a person's name (no leading whitespace)
        $person = $1;
        $provisions{$person} = [ ] unless $provisions{$person};
    } elsif (/^\s+(\S.*)/) { # a provision
        die 'No person yet!' unless defined $person;
        push @{$provisions{$person}}, $1;
    } else {
        die "I don't understand: $_";
    }
}
```

首先，声明用于供应列表的结果散列和当前人员名称的变量。对于读入的每一行，判断读入的内容为人员名称还是供应项。如果是人员名称，我们就将记住该名称，并且为该人员创建一个散列元素。在数据文件中，如果他或她的供应列表中的内容分别放置在两个不同的地方，unless 测试语句保证我们不会删除任何人的供应列表。

例如，假定“*The Skipper*” 和 “*sextant*”（注意单词前的空格）在数据文件的末尾，为了罗列一个额外的数据项。

键是人员名称，值最初为指向一个空匿名数组的引用。如果该行是一个供应项，就使

用数组引用，将该供应项推入当前数组的尾部。

代码正常工作，但是实际上并不需要写这么多。为什么？因为我们可以不考虑初始化散列某个键值对中的值为指向一个空数组的引用：

```
my %provisions;
my $person;

while (<>) {
    if (/^(\S.*)/) { # a person's name (no leading whitespace)
        $person = $1;
        ## $provisions{$person} = [ ] unless $provisions{$person};
    } elsif (/^\s+(\S.*)/) { # a provision
        die 'No person yet!' unless defined $person;
        push @{$provisions{$person}}, $1;
    } else {
        die "I don't understand: $_";
    }
}
```

当我们尝试为 Skipper 存储 “blue shirt” 的时候将发生什么？当查看第二行输入时，我们将以下列形式结束：

```
push @{$provisions{'The Skipper'}}, "blue_shirt";
```

此时此刻，\$provisions{"The Skipper"}并不存在，但是我们试图以一个数组引用的方式使用它。为了处理这种情况，Perl 自动在变量中插入一个指向新的空匿名数组的引用，并且继续操作。然后，新创建的空匿名数组的引用被解引用，并且将 “blue shirt” 插入供应列表。

这样的过程叫做自动带入。只要我没有给变量（或者访问数组或者散列中的单个元素）赋值，Perl 将自动创建我们假定存在的引用类型。如果值为 `undef`，但是我们将它作为数组引用操作，Perl 就用一个新的数组引用替换 `undef`。

这实际上与我们一直以来使用 Perl 的方式一致。Perl 根据需要创建新的变量。在此语句之前，\$provisions{"The Skipper"}并不存在，因此 Perl 自动创建%provisions 散列，之后我们才能够访问该散列中的数据。同样，@{\$provisions{"The Skipper"}不存在，Perl 自动创建名为 “The Skipper”的键，之后我们才能够访问它。我们没有给该键赋值，所以它的值为 `undef`。当我们尝试使用未定义的值作为数组引用时，Perl 将该值作为一个数组引用，因此可以将该值解引用为一个数组。

例如：如下代码能够运行：

```
my $not_yet;                      # new undefined variable
@$not_yet = (1, 2, 3);
```

以数组引用的方式对\$not_yet 的值进行解引用。但是因为它初始化为 `undef`，所以 Perl 表现为似乎我们已经将\$not_yet 显式初始化为一个空数组引用：

```
my $not_yet;
$not_yet = [ ]; # inserted through autovivification
@$not_yet = (1, 2, 3);
```

换句话说，一个最初为空的数组变为一个拥有三个元素的数组。

这种自动带入也能够用于多层赋值：

```
my $top;
$top->[2]->[4] = 'lee-lou';
```

最初，\$top 的值为 `undef`，但因为以数组引用的方式对它解引用，Perl 就将一个空匿名数组的引用插入\$top。然后 Perl 访问该数组第三个元素（索引值为 2），这将导致 Perl 将该数组增长为三个元素的长度。因为所访问的第三个元素依旧是 `undef`，所以 Perl 用另外一个指向空匿名数组的引用填充它。最后，我们沿着新创建的数组延长，设定第五个元素为 `lee-lou`。

5.8 自动带入和散列

自动带入也适用于散列引用^{注3}。如果以散列引用的方式对一个包含 `undef` 的变量解引用，就插入一个空匿名散列的引用，并且操作将继续进行。

自动带入的一个便捷之处在于典型的数据压缩任务。例如，Professor 建立和运行一个岛屿大小的网络（可能使用 Coco-Net 或者 Vines），现在希望跟踪主机间的流量。他现在开始把传输的字节数记录到日志文件中，包括源主机、目标主机和传输字节数：

```
professor.hut gilligan.crew.hut 1250
professor.hut lovey.howell.hut 910
thurston.howell.hut lovey.howell.hut 1250
professor.hut lovey.howell.hut 450
professor.hut laser3.copyroom.hut 2924
ginger.girl.hut professor.hut 1218
ginger.girl.hut maryann.girl.hut 199
...
```

现在，Professor 想要生成某天关于源主机、目标主机和传输字节总数的报表。尽可能简单地逐行读取数据，分割数据，然后对于同类数据，添加最新的值到之前的数据中，通过这样的方式把所有数据制成表格。

```
my %total_bytes;
while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}
```

如果我们想要查看第一行输入的生成方式，将执行下列语句：

```
$total_bytes{'professor.hut'}{'gilligan.crew.hut'} += 1250;
```

如上面的代码所示，因为散列%total_bytes 的初始值为空，所以 Perl 没有查找到 professor.hut 的第一个键，但它会以散列引用的方式解引用，建立一个 `undef` 值。（记住，在此处的两个大括号间有隐式的箭头。）Perl 在该元素中插入指向一个空的匿名散列的

注 3：或者，确实是，散列引用。

引用，这样它就会立即扩展为包含一个键为 gilligan.crew.hut 的散列。该键的对应的值的初始值为 undef，即初始值与零等同，当该值与 1250 相加时，相加的结果 1250 也被插入回散列（参考图 5-2）。

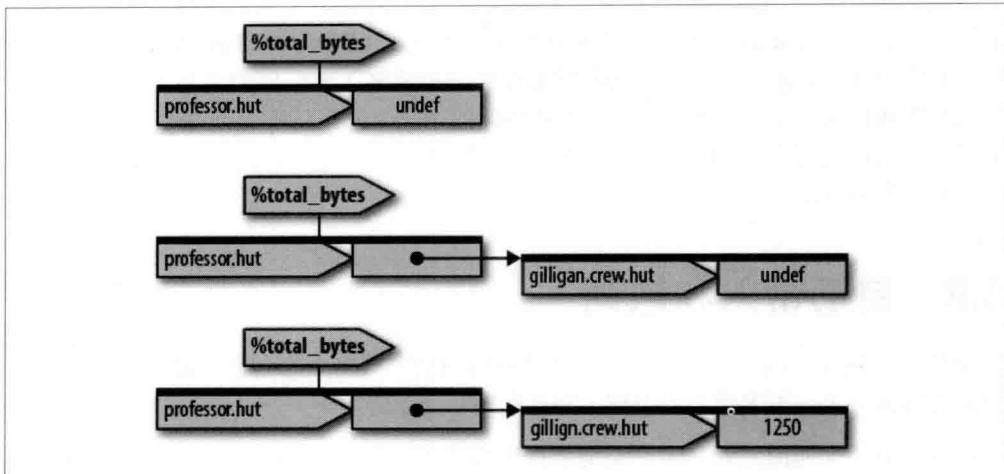


图 5-2 %total_bytes 散列的自动带入

后续任意数据行如果包含相同的源主机和目标主机，就将重用这个值，再加上新读入的字节值计算总数。但是每个新的目标主机会自动扩展散列，用来包括一个新的初始 undef 字节的值，而且每个源主机使用自动带入来创建目标主机散列。换句话说，Perl 总是做对的事情。

一旦处理完文件，就该显示所有数据报表。首先，确定所有源主机：

```
for my $source (keys %total_bytes) {  
    ...
```

此时，我们将获取全部目标主机，这些语法有点棘手。我们想要散列的所有键，就在第一个结构中，通过对散列元素的值解引用得到：

```
for my $source (keys %total_bytes) {  
    for my $destination (keys %{ $total_bytes{$source} }) {  
        ...
```

为了更好地度量，对散列键的列表进行排序，保持一致性：

```
for my $source (sort keys %total_bytes) {  
    for my $destination (sort keys %{ $total_bytes{$source} }) {  
        print "$source => $destination:  
              " $total_bytes{$source}{$destination} bytes\n";  
    }  
    print "\n";  
}
```

这是一个为了减少报表数据量的典型策略。创建一个散列引用（可能嵌套更深，后续会看到），根据需要在上层数据结构中使用自动带入填补空白，并且遍历结果数据结构，显示结果。



注意

Perl 数据结构的“菜谱” perldsc 拥有更多创建、访问和输出复杂数据结构的示例。

5.9 练习

可以在附录中的“第 5 章答案”部分找到这些练习的答案。

1. [5 分钟] 不运行，你可以看出下列代码有什么问题吗？如果你一两分钟后仍不能查找出问题，查看运行这段代码是否能给你一些关于如何解决问题的提示（你可能需要打开警告）：

```
my %passenger_1 = {  
    name      => 'Ginger',  
    age       => 22,  
    occupation => 'Movie Star',  
    real_age   => 35,  
    hat        => undef,  
};  
  
my %passenger_2 = {  
    name      => 'Mary Ann',  
    age       => 19,  
    hat        => 'bonnet',  
    favorite_food => 'corn',  
};  
  
my @passengers = (\%passenger_1, \%passenger_2);
```

2. [40 分钟] Professor 的数据文件（本章之前部分提及）叫做 coconet.dat，该文件可以通过 <http://www.intermediateperl.com/> 的下载页面下载。可能有注释行（从#符号开始）；跳过这些行。（就是说，程序将跳过这些行的内容，如果你阅读它们，就能找到很多有用的提示。）如下所示的内容为文件中一些初始的数据行：

```
gilligan.crew.hut lovey.howell.hut 4721  
thurston.howell.hut lovey.howell.hut 4046  
professor.hut ginger.girl.hut 5768  
gilligan.crew.hut laser3.copyroom.hut 9352  
gilligan.crew.hut maryann.girl.hut 1180  
fileserver.copyroom.hut thurston.howell.hut 2548  
skipper.crew.hut gilligan.crew.hut 1259  
fileserver.copyroom.hut maryann.girl.hut 248  
fileserver.copyroom.hut maryann.girl.hut 798  
skipper.crew.hut maryann.girl.hut 1921
```

修改本章中的代码，输出中每个源主机的部分显示该主机输出的总字节数。根据输出的字节数目大小对源主机排序，在每一组中，根据从源主机传至目标主机的传输字节数大小排列目标主机：

```
professor.hut => gilligan.hut: 1845  
professor.hut => maryann.hut: 90
```

最终结果是发送数据最多的主机为列表中的第一个源主机，并且其中第一个目标主机将为接收数据最多的主机。Professor 能够使用该报表，根据效率重新配置网络。

3. [40 分钟] 以上面练习 2 中程序的数据结构作为开始，以同样的格式重写 coconet.dat 文件，但是对源主机进行排序。对每个目标主机每次随着源主机传输的总字节数进行报表。目标主机将在源主机名后内缩，并且按照主机名排序：

```
ginger.hut  
    maryann.hut 13744  
professor.hut  
    gilligan.hut 1845  
    maryann.hut 90  
thurston.howell.hut  
    lovey.howell.hut 97560  
...  
...
```

操作复杂的数据结构

在介绍了引用的基本内容后，我们后续将介绍操作复杂数据的其他方法。首先，我们通过使用调试器检查复杂的数据结构，然后使用 Data::Dumper 模块显示程序控制下的数据。下一步，我们将展示如何简单、快速地使用 Storable 模块存储和检索复杂数据。最终我们将回顾总结 grep 命令和 map 命令，查看这两条命令如何应用于复杂的数据。

6.1 使用调试器查看复杂的数据

Perl 调试器能够很容易地显示复杂数据。例如，我们能够单步执行第 5 章中如下版本的字节计数程序：

```
my %total_bytes;
while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}
for my $source (sort keys %total_bytes) {
    for my $destination (sort keys %{$total_bytes{$source}}) {
        print "$source => $destination:", 
              " $total_bytes{$source}{$destination} bytes\n";
    }
    print "\n";
}
```

我们将要使用如下所示的数据测试该程序：

```
professor.hut gilligan.crew.hut 1250
professor.hut lovey.howell.hut 910
thurston.howell.hut lovey.howell.hut 1250
professor.hut lovey.howell.hut 450
ginger.girl.hut professor.hut 1218
ginger.girl.hut maryann.girl.hut 199
```

这可以使用多种方法实现。其中最简单的一种是在命令行条件下调用 perl 并带上-d 参数：

```
myhost% perl -d bytecounts bytecounts-in

Loading DB routines from perl5db.pl version 1.19
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main:::(bytecounts:2):      my %total_bytes;
DB<1> s
main:::(bytecounts:3):      while (<>) {
DB<1> s
main:::(bytecounts:4):      my ($source, $destination, $bytes) = split;
DB<1> s
main:::(bytecounts:5):      $total_bytes{$source}{$destination} += $bytes;
DB<1> x $source, $destination, $bytes
0 'professor.hut'
1 'gilligan.crew.hut'
2 1250
```

每个新发布的调试器都与之前发布的调试器在工作方式上略有不同，因此你屏幕中显示的内容可能与这里所展示的并不完全一致。如果我们在任何时刻遇到困难，就输入 h 获取帮助，或者查询 perldebug 文档。

调试器会在执行前显示每一行正在调试的代码。这就意味着，此时此刻，我们将调用自动带入，并且已经建立了我们自己的键。s 命令将单步执行程序，而 x 命令用一个漂亮的格式转储列表中的所有值。我们能够看到\$source、\$destination 和\$bytes 的值是正确的，现在是更新数据的时候了：

```
DB<2> s
main:::(bytecounts:3):      while (<>) {
```

我们通过自动带入创建散列的条目。下面是得到的结果：

```
DB<2> x \%total_bytes
0 HASH(0x132dc)
'professor.hut' => HASH(0x37a34)
'gilligan.crew.hut' => 1250
```

当向 x 命令提供一个散列引用时，它就将以键值对的方式转储散列中的全部内容。如果散列中的某个值也为散列引用，它同样会递归地转储那些内容。如上例所示，我们看到的%total_bytes 散列有一个键为 professor.hut，该键相对应的值是另一个散列引用。输出该散列引用的结果同预期一致：键为 gilligan.crew.hut，值为 1250 的键值对。

如下所示的内容将在下一次赋值之后发生：

```
DB<3> s
main:::(bytecounts:4):      my ($source, $destination, $bytes) = split;
DB<3> s
main:::(bytecounts:5):      $total_bytes{$source}{$destination} += $bytes;
DB<3> x $source, $destination, $bytes
0 'professor.hut'
1 'lovey.howell.hut'
2 910
```

```

DB<4> s
main:::(bytecounts:3):      while (<>) {
DB<4> x \%total_bytes
0 HASH(0x132dc)
'professor.hut' => HASH(0x37a34)
'gilligan.crew.hut' => 1250
'lovey.howell.hut' => 910

```

现在，我们添加从 professor.hut 流向 lovey.howell.hut 的字节数。顶层的散列内容没有变化，但是第二层的散列增添了新的条目。我们按照如下方式继续执行：

```

DB<5> s
main:::(bytecounts:4):      my ($source, $destination, $bytes) = split;
DB<6> s
main:::(bytecounts:5):      $total_bytes{$source}{$destination} += $bytes;
DB<6> x $source, $destination, $bytes
0 'thurston.howell.hut'
1 'lovey.howell.hut'
2 1250
DB<7> s
main:::(bytecounts:3):      while (<>) {
DB<7> x \%total_bytes
0 HASH(0x132dc)
'professor.hut' => HASH(0x37a34)
'gilligan.crew.hut' => 1250
'lovey.howell.hut' => 910
'thurston.howell.hut' => HASH(0x2f9538)
'lovey.howell.hut' => 1250

```

现在变得越来越有趣了。在顶层散列的一个新条目拥有一个键 thurston.howell.hut 和一个新的散列引用，该散列引用指向最初自动带入的一个空散列。在一个空散列放置到位之后，立即添加了一个键值对，表明从 thurston.howell.hut 到 lovey.howell.hut 传输了 1250 字节。如下所示，我们继续单步执行：

```

DB<8> s
main:::(bytecounts:4):      my ($source, $destination, $bytes) = split;
DB<8> s
main:::(bytecounts:5):      $total_bytes{$source}{$destination} += $bytes;
DB<8> x $source, $destination, $bytes
0 'professor.hut'
1 'lovey.howell.hut'
2 450
DB<9> s
main:::(bytecounts:3):      while (<>) {
DB<9> x \%total_bytes
0 HASH(0x132dc)
'professor.hut' => HASH(0x37a34)
'gilligan.crew.hut' => 1250
'lovey.howell.hut' => 1360
'thurston.howell.hut' => HASH(0x2f9538)
'lovey.howell.hut' => 1250

```

现在，我们已经从 professor.hut 到 lovey.howell.hut 添加了一些更多的字节，重用已存在值的位置。这些步骤都没有什么特别之处，我们按如下方式继续单步执行：

```

DB<10> s
main::(bytecounts:4):           my ($source, $destination, $bytes) = split;
DB<10> s
main::(bytecounts:5):           $total_bytes{$source}{$destination} += $bytes;
DB<10> x $source, $destination, $bytes
0 'ginger.girl.hut'
1 'professor.hut'
2 1218
DB<11> s
main::(bytecounts:3):           while (<>) {
DB<11> x \%total_bytes
0 HASH(0x132dc)
'ginger.girl.hut' => HASH(0x297474)
'professor.hut' => 1218
'professor.hut' => HASH(0x37a34)
'gilligan.crew.hut' => 1250
'lovey.howell.hut' => 1360
'thurston.howell.hut' => HASH(0x2f9538)
'lovey.howell.hut' => 1250

```

此时此刻，我们添加了一个新的源主机：ginger.girl.hut。注意，顶层散列现在拥有三个元素，并且每个元素各自拥有一个不同的散列引用值。我们按照如下方式继续单步执行：

```

DB<12> s
main::(bytecounts:4):           my ($source, $destination, $bytes) = split;
DB<12> s
main::(bytecounts:5):           $total_bytes{$source}{$destination} += $bytes;
DB<12> x $source, $destination, $bytes
0 'ginger.girl.hut'
1 'maryann.girl.hut'
2 199
DB<13> s
main::(bytecounts:3):           while (<>) {
DB<13> x \%total_bytes
0 HASH(0x132dc)
'ginger.girl.hut' => HASH(0x297474)
'maryann.girl.hut' => 199
'professor.hut' => 1218
'professor.hut' => HASH(0x37a34)
'gilligan.crew.hut' => 1250
'lovey.howell.hut' => 1360


```

现在，我们向散列中添加第二个目标主机，该目标主机记录所有源自于 ginger.girl.hut 的字节信息。因为这是数据的最后一行（在本次运行中），所以下一次单步执行将我们带入更低层次的 foreach 循环：

```

DB<14> s
main::(bytecounts:8):           for my $source (sort keys %total_bytes) {

```

即使我们不能从这些圆括号的内部直接检查列表的值，我们也能够按如下方式显示它们：

```

DB<14> x sort keys %total_bytes
0 'ginger.girl.hut'
1 'professor.hut'
2 'thurston.howell.hut'

```

这是 foreach 循环现在正在扫描的列表。这些是在特定日志文件中可以看到的用于传输字节的所有源主机。下面是单步执行内层循环时发生的状况：

```
DB<15> s
main::(bytecounts:9):           for my $destination (sort keys %{ $total_bytes{
    $source} }) {
```

此时此刻，我们能够由内而外精确地确定从圆括号内的列表值将得到什么结果。按如下方式继续查看：

```
DB<15> x $source
0  'ginger.girl.hut'
DB<16> x $total_bytes{$source}
0  HASH(0x297474)
'maryann.girl.hut' => 199
'professor.hut' => 1218
DB<18> x keys %{ $total_bytes{$source} }
0  'maryann.girl.hut'
1  'professor.hut'
DB<19> x sort keys %{ $total_bytes{$source} }
0  'maryann.girl.hut'
1  'professor.hut'
```

转储\$total_bytes{\$source}的值，将显示该值实际是一个散列引用。同样，sort 语句似乎什么都没有做，然而输出的键也不必用排序的方式输出。下一步是查找数据：

```
DB<20> s
main::(bytecounts:10):          print "$source => $destination:",
main::(bytecounts:11):          " $total_bytes{$source}{$destination} bytes\n";
DB<20> x $source, $destination
0  'ginger.girl.hut'
1  'maryann.girl.hut'
DB<21> x $total_bytes{$source}{$destination}
0  199
```

正如我们在调试器中所见，我们可以很方便地在调试器中显示数据，甚至是结构化的数据，来帮助理解程序。

6.2 使用 Data::Dumper 模块查看复杂的数据

另一个将复杂数据结构快速可视化的方法是转储它们。Data::Dumper 模块属于 Perl 的核心模块，该模块提供了一个基本方法，将 Perl 的数据结构显示为 Perl 代码。如下所示，用一个简单的 Data::Dumper 模块调用，改写字节计数程序的后半部分：

```
use Data::Dumper;

my %total_bytes;
while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}

print Dumper(\%total_bytes);
```

Data::Dumper 模块定义了 Dumper 子例程。该子例程类似于调试器中的 x 命令。我们能够向 Dumper 子例程提供一个或者多个值，Dumper 子例程就能够将这些值转化为一个可输出的字符串。我们传递引用参数以保持散列作为散列的形式，而不是分离参数的列表。然而，调试器的 x 命令与 Dumper 子例程的区别在于，由 Dumper 子例程生成的字符串是 Perl 代码：

```
% perl bytecounts < bytecounts-in
$VAR1 = {
    'thurston.howell.hut' => {
        'lovey.howell.hut' => 1250
    },
    'ginger.girl.hut' => {
        'maryann.girl.hut' => 199,
        'professor.hut' => 1218
    },
    'professor.hut' => {
        'gilligan.crew.hut' => 1250,
        'lovey.howell.hut' => 1360
    }
};
```

这样的 Perl 代码相当易于理解；以上代码显示我们拥有一个指向包含三个键值对散列的引用，散列的每个键指向一个嵌套散列的引用。我们能够对这些代码求值，得到的值与原始散列相同。然而，如果你考虑通过这样的方式保存一个复杂的数据结构，持续在不同的程序中调用该数据结构，就请继续往下阅读。

与调试器的 x 命令类似，Data::Dumper 模块能够恰当地处理共享数据。例如，回顾第 5 章中的“内存泄漏”的示例：

```
use Data::Dumper;
$Data::Dumper::Purity = 1; # declare possibly self-referencing structures
my @data1 = qw(one won);
my @data2 = qw(two too to);
push @data2, \@data1;
push @data1, \@data2;
print Dumper(\@data1, \@data2);
```

如下所示为以上程序输出的结果：

```
$VAR1 = [
    'one',
    'won',
    [
        'two',
        'too',
        'to',
        [
        ]
    ]
];
$VAR1->[2][3] = $VAR1;
$VAR2 = $VAR1->[2];
```

请注意现在如何创建两个不同的变量，因为有两个参数传递到 Dumper 子例程。元素 \$VAR1 对应 @data1 数组的引用，而元素 \$VAR2 对应 @data2 数组的引用。此外，如果

使用调试器的 x 命令，得到的输出结果与之类似：

```
DB<1> x \@data1, \@data2
 0  ARRAY(0xf914)
 0  'one'
 1  'won'
 2  ARRAY(0x3122a8)
 0  'two'
 1  'too'
 2  'to'
 3  ARRAY(0xf914)
    -> REUSED_ADDRESS
 1  ARRAY(0x3122a8)
-> REUSED_ADDRESS
```

短语 REUSED_ADDRESS 表明某些部分的数据实际上是我们已经看到的数据的引用。

其他形式的转储程序

作为 Perl 核心模块的 Data::Dumper 模块非常易于使用。然而，有一个基本的设计决策导致它的输出变得有一些丑陋。该模块被专门用于将 Perl 的数据结构显示为一个字符串，并且该字符串是有效的 Perl 代码。这意味着大部分情况下，我们能够对该字符串使用 eval 操作，然后重新得到数据结构。如果我们不在意重建数据结构，就不必使用 Data::Dumper 模块。

如果其他模块不在乎是否创建有效的 Perl 代码作为输出，就能够有更佳的输出格式。所付出的代价仅仅是通过 CPAN 安装这些模块。我们使用不同的模块查看同样的数据结构的转储值。如下所示为散列%total_bytes 的值：

```
my %total_bytes = (
  'thurston.howell.hut' => {
    'lovey.howell.hut' => 1250
  },
  'ginger.girl.hut' => {
    'maryann.girl.hut' => 199,
    'professor.hut' => 1218
  },
  'professor.hut' => {
    'gilligan.crew.hut' => 1250,
    'lovey.howell.hut' => 1360
  }
);
```

Data::Dump 模块有一个叫做 dump 的子例程，与 Data::Dumper 模块的用法一致，向 dump 子例程提供一个引用作为参数：

```
use Data::Dump qw(dump);

dump( \%total_bytes );
```

它的输出比 Data::Dumper 模块的输出看起来更优美一些，但是看起来仍然很像 Perl 代码。按照如下方式将输出行包裹起来：

```

{
    "ginger.girl.hut"      =>
        { "maryann.girl.hut" => 199, "professor.hut" => 1218 },
    "professor.hut"         =>
        { "gilligan.crew.hut" => 1250, "lovey.howell.hut" => 1360 },
    "thurston.howell.hut"  =>
        { "lovey.howell.hut" => 1250 },
}

```

Data::Printer 模块甚至能够去掉更多 Perl 风格的字符。该模块的 p 子例程并不需要一个引用作为参数，因为该子例程能够自动检测参数类型：

```
use Data::Printer;
```

```
p( %total_bytes );
```

输出非常易于阅读：

```

{
    ginger.girl.hut      {
        maryann.girl.hut  199,
        professor.hut     1218
    },
    professor.hut         {
        gilligan.crew.hut 1250,
        lovey.howell.hut  1360
    },
    thurston.howell.hut  {
        lovey.howell.hut  1250
    }
}

```

6.3 数据编组

我们能够将 Data::Dumper 模块 Dumper 子例程的输出放入一个文件中，然后用另一个程序加载该文件。如下所示的程序拥有一个希望保留的循环数据结构：

```

use Data::Dumper;

my @data1 = qw(one won);
my @data2 = qw(two too to);
push @data2, \@data1;
push @data1, \@data2;
my $string = Dumper( \@data1, \@data2 ); # to some filehandle

```

\$string 中的文本是 Perl 代码，该段代码定义了两个变量，\$VAR1 和\$VAR2：

```

$VAR1 = [
    'one',
    'won',
    [
        'two',
        'too',
        'to',
        $VAR1
    ]
]

```

```
];
$VAR2 = $VAR1->[2];
```

可以使用第3章介绍的eval语句块的字符串形式，将上面的代码转换回到Perl数据结构。可以在当前程序或者不同的程序中按照如下所示的方式进行操作：

```
my $data_structure = eval $string
```

这样的代码编写风格并不美观，这些变量之前就已定义，并且现在拥有普通的标识符。使用eval语句块之后，拥有名称为\$VAR1和\$VAR2的变量。

为了解决这些问题，可以调用Dump方法转储两个数组引用。第一个数组引用是希望转储的变量列表，第二个拥有希望使用的名称列表：

```
print Data::Dumper->Dump(
    [ \@data1, \@data2 ],
    [ qw(*data1 *data2) ]
);
```

我们并未在此介绍符号表(typeglob)，但是我们将使用*前缀，该前缀告诉Data::Dumper模块查找引用，用于了解在字符串中需要使用哪些变量类型：

```
@data1 = (
    'one',
    'won',
    [
        'two',
        'too',
        'to',
        \@data1
    ]
);
@data2 = @{$data1[2]};
```

当对这些代码使用eval语句块时，就能获得同样的数据，而且指向这些数据的变量拥有同样的名字。



注意

了解关于符号表的更多信息，请查阅*Mastering Perl*。

6.3.1 使用Storable模块对复杂数据排序

当将代码作为Perl代码执行时，以两个包变量\$VAR1和\$VAR2作为结束，这两个包变量等价于初始数据。这就叫做编组数据：将复杂数据转换成一种能够作为字节流写入文件的一种形式，用于后续重建数据。

然而，另一个Perl核心模块更适合编组数据：Storable模块。与Data::Dumper模块相比，它更适合的原因是Storable模块生成更短小并且更易于处理的文件。（Storable模块在Perl的近期版本中是核心模块，但是如果它不存在，始终可以通过CPAN安装。）

除了必须将所有数据放入一个引用中以外，该模块与Data::Dumper模块的接口类似。

`freeze` 子例程返回一个二进制字符串，该字符串用于表述所需要输出的数据结构：

```
use Storable;
my @data1 = qw(one won);
my @data2 = qw(two too to);
push @data2, \@data1;
push @data1, \@data2;
my $frozen = freeze [\@data1, \@data2];
```

`$frozen` 变量中的字符串有 68 个字节，这比使用 `Data::Dumper` 模块输出同样的内容要小很多，可读性也更差。将该字符串保存到一个文件中，通过套接字发送，或者输出到文件句柄。无论谁得到该字符串都能按照如下方式重建数据结构：

```
use Storable;
my $data = thaw( $input );
```

我们需要知道关于数据结构的一些信息才能使用它。

如果我们想要以二进制表示方式存入文件中，就可以按照如下方式，使用 `nstore` 子例程固化（`freeze`）数据，这样就可以只用一个命令存入文件中：

```
nstore [\@data1, \@data2], $filename;
```

另一方面，使用 `retrieve` 重建数据结构：

```
my $array_ref = retrieve $filename;
```

`Storable` 模块使用的二进制格式，默认是字节顺序依赖的架构，但这里使用 `nstore` 替代 `store`，字母 n 在这里代表“网络字节序”。按照惯例，不同的架构一致同意用于存储数字的“网络字节序”，因此不同的架构知道如何解释数据。

还有一个 `Storable` 模块能够自动完成有用的任务。如果我们想要复制一个能够完全独立存在的数据结构，就可以先固化，然后再解冻（`thaw`）。初始和重建的数据结构相互之间不影响。

为什么这很有趣呢？因为我们知道，如果我们复制一些数组，然后改变其中的一个而不影响其他数组：

```
use Data::Dumper;

my @provisions = qw( hat sunscreen );
my @packed = @provisions;

push @packed, 'blue_shirt';

print Data::Dumper->Dump(
    [ \@provisions ],
    [ qw( *provisions ) ]
);
print Data::Dumper->Dump(
    [ \@packed ],
    [ qw( *packed ) ]
);
```

如下所示，将`@provisions` 数组复制到`@packed` 数组中，然后修改`@packed` 数组，但是`@provisions` 数组中的内容保持不变：

```

@provisions = (
    'hat',
    'sunscreen'
);
@packed = (
    'hat',
    'sunscreen',
    'blue_shirt'
);

```

这恰巧行得通，因为复制命名数组是做浅复制（shallow copy）。如果其中的某个数组包含一个引用，情况就不一样了。如下所示为同样的程序，只是换了一种形式。将`@science_kit`数组的引用加入`@provisions`数组中：

```

use Data::Dumper;

my @provisions = qw( hat sunscreen );
my @science_kit = qw( microscope radio );
push @provisions, \@science_kit;

my @packed = @provisions;

push @packed, 'blue_shirt';

print Data::Dumper->Dump(
    [ \@provisions ],
    [ qw( *provisions ) ]
);
print Data::Dumper->Dump(
    [ \@packed ],
    [ qw( *packed ) ]
);

```



注意

数组或者散列的浅复制只能复制表层的数值。然而，如果其中的某个元素是引用，新的数组或者散列将得到同样的引用。引用所指向的全部内容（更深层的部分）都是一样的。所以，这就是浅复制。

输出显示将出现同样的结果。我们复制数组，然后添加一个元素到其中的一个数组中。到目前为止还不错。如下所示，我们将修改其中的一个数组，然后将它添加到`@science_kit`数组中：

```

use Data::Dumper;

my @provisions = qw( hat sunscreen );
my @science_kit = qw( microscope radio );
push @provisions, \@science_kit;

my @packed = @provisions;

push @packed, 'blue_shirt';

push @{$packed[2]}, 'batteries';

print Data::Dumper->Dump(

```

```

[ \@provisions ],
[ qw( *provisions ) ]
);
print Data::Dumper->Dump(
[ \@packed ],
[ qw( *packed ) ]
);

```

输出显示两个数组有变化，现在每个数组都拥有 batteries 元素！

```

@provisions = (
    'hat',
    'sunscreen',
    [
        'microscope',
        'radio',
        'batteries'
    ]
);
@packed = (
    'hat',
    'sunscreen',
    [
        'microscope',
        'radio',
        'batteries'
    ],
    'blue_shirt'
);

```

这些数组都拥有一个指向同样数据的引用，如图 6-1 所示，因此改变共享引用的值，就会影响其他所有数组。

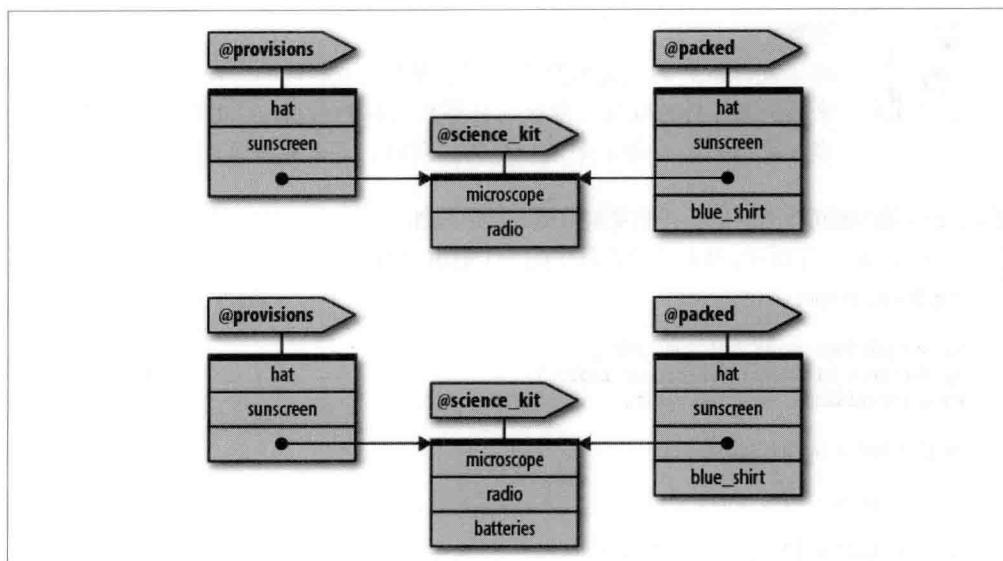


图 6-1 `@provisions` 数组和 `@packed` 数组都包含指向 `@science_kit` 数组的引用

通常情况下不期望浅复制。相反，很多情况下希望深复制，如下所示，通过固化 Storable 模块的子例程，然后再解冻子例程能够自动实现：

```
use Data::Dumper;
use Storable qw(freeze thaw);

my @provisions = qw( hat sunscreen );
my @science_kit = qw( microscope radio );
push @provisions, \@science_kit;

my $frozen = freeze \@provisions;
my @packed = @{ thaw $frozen };

push @packed, 'blue_shirt';

push @{ $packed[2] }, 'batteries';

print Data::Dumper->Dump(
    [ \@provisions ],
    [ qw( *provisions ) ]
);
print Data::Dumper->Dump(
    [ \@packed ],
    [ qw( *packed ) ]
);
```

现在，@provisions 数组和@packed 数组是完全分离的。只有@packed 数组得到 batteries 项（参考图 6-2）。

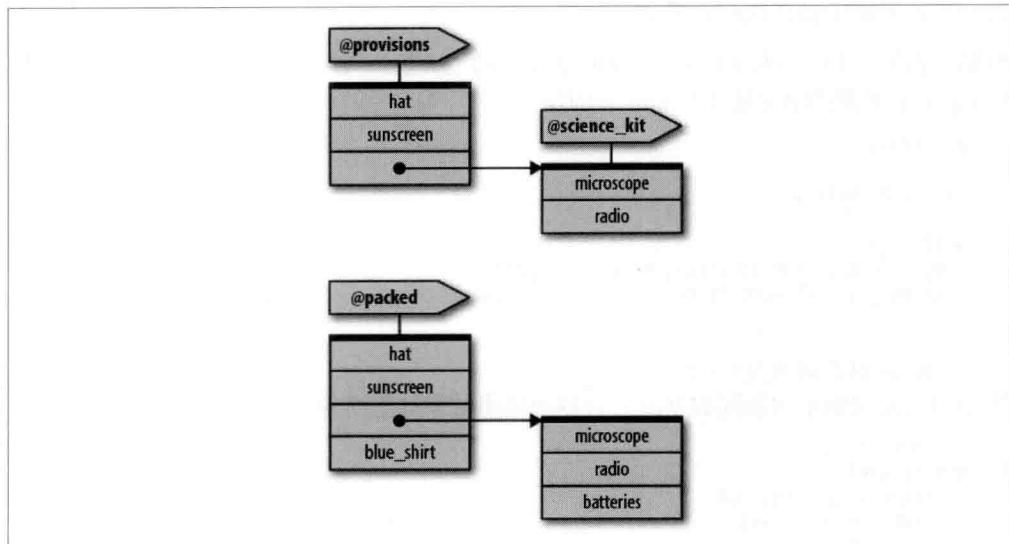


图 6-2 在深复制之后的@provisions 数组和@packed 数组

```
@provisions = (
    'hat',
    'sunscreen',
    [

```

```

        'microscope',
        'radio'
    ];
);
@packed = (
    'hat',
    'suncreen',
    [
        'microscope',
        'radio',
        'batteries'
    ],
    'blue_shirt'
);

```

我们不必分两步完成这些。Storable 模块已经自动预见到这类情况，并能够通过 dclone 子例程一次性完成：

```
my @packed = @{ dclone \@provisions };
```

这将使 Storable 模块非常有用，即使我们不想与任何人交换数据。

6.3.2 YAML 模块

Ingy dot Net^{注1}提出另一种标记语言（Yet Another Mark Language, YAML）以提供一种更易于阅读（并且更简洁的）格式，其他不同的编程语言都可以处理该格式^{注2}。它的工作方式与 Data::Dumper 模块相同。当后续讨论模块时，将会看到更多关于 YAML 模块的介绍，因此不必在此过多描述。

根据之前的示例，在使用 Data::Dumper 模块的地方插入 YAML 模块，然后在之前使用 Dumper 子例程的地方使用 Dump 子例程：

```

use YAML;

my %total_bytes;

while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}

print Dump(\%total_bytes);

```

当使用之前示例中的相同数据时，得到的输出结果如下所示：

```

--- #YAML:1.0
ginger.girl.hut:
maryann.girl.hut: 199
professor.hut: 1218
professor.hut:
gilligan.crew.hut: 1250
lovey.howell.hut: 1360

```

注 1：是的，这就是他的名字。Ingy 编写了很多有趣的模块，可以在 <https://www.metacpan.org/author/INGY> 中查看。

注 2：“诗人相信他们的想法能够跨越所有的语言传播。”参见 <http://acmeism.org/>。

```
thurston.howell.hut:  
    lovey.howell.hut: 1250
```

以上内容更易于阅读，因为它占据更少的屏幕空间，当需要显示深度嵌套的数据结构时，这就显得非常方便。并且，如果我们想要与使用 Ruby 或者 Python 的程序员分享这些数据，他们也能够很轻易地理解这些数据。

6.3.3 JSON 模块

JavaScript 对象表示法 (JavaScript Object Notation)，或者 JSON，是另一种流行的格式。我们能够很方便地在不同程序中交换 JSON 数据，即使它们由不同的语言实现。例如，Perl 程序能够发送数据到一个 Web 页面，Web 页面的 JavaScript 能够很轻易地立即使用数据。JSON 模块有很多种创建输出的方法，其中就包括 `to_json`:

```
use JSON;  
  
print to_json( \%total_bytes, { pretty => 1 } );
```

使用 `pretty` 属性使该页面的输出看起来更漂亮（也使我们更容易阅读）：

```
{  
    "thurston.howell.hut" : {  
        "lovey.howell.hut" : 1250  
    },  
    "ginger.girl.hut" : {  
        "maryann.girl.hut" : 199,  
        "professor.hut" : 1218  
    },  
    "professor.hut" : {  
        "gilligan.crew.hut" : 1250,  
        "lovey.howell.hut" : 1360  
    }  
}
```

可能通过一个文件、Web 请求或者来自其他程序的输出获取 JSON 文本。也能够很容易地重新创建 Perl 的数据结构：

```
use JSON;  
  
my $hash_ref = from_json( $json_string );
```

6.4 使用 map 和 grep 操作符

随着数据结构变得越来越复杂，这些操作符将有助于使用高阶的结构处理普通的任务，例如选择和转换。就这一点而言，Perl 的 `grep` 操作符和 `map` 操作符非常值得掌握。

6.5 应用一点间接方法

一旦我们看过一两个解决方案之后，某些貌似复杂的问题实际上就很简单。例如，假定我们想要查找列表项所拥有的所有奇数数字的和，但不想得到项本身。我们只是想

要知道它们在初始列表中占据的位置。

全部所需要的是一点间接法^{注3}。首先，我们有一个选择性问题，因此使用 grep。我们不想得到 grep 返回的每项本身的值，而要每项的索引：

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @indices_of_odd_digit_sums = grep {
    ...
} 0..$#input_numbers;
```

此时此刻，表达式 0..\$#input_numbers 将为数组的索引列表。在语句块内部，\$_ 是一个很小的整数，值为 0 ~ 6（总共 7 个数字）。现在我们不想要确定 \$_ 是否为奇数数字的和。因为我们想知道相应索引位置的数组元素是否拥有奇数和。如下所示，不使用 \$_ 变量获取感兴趣的数字，而使用 \$input_numbers[\$_]:

```
my @indices_of_odd_digit_sums = grep {
    my $number = $input_numbers[$_];
    my $sum;
    $sum += $_ for split //, $number;
    $sum % 2;
} 0..$#input_numbers;
```

结果是在列表中 1、16 和 32 对应的索引：0、4 和 5。如下所示，可以使用这些数组切片中的索引再次获取初始值：

```
my @odd_digit_sums = @input_numbers[ @indices_of_odd_digit_sums ];
```

此时 grep 操作符或者 map 操作符的间接策略是认为 \$_ 标识一个特别感兴趣的项，例如散列的键值或者数组的索引，然后在语句块或者表达式中使用该标识，用于访问实际值。

如下所示是另一个示例：选择 @x 数组的元素比数组 @y 中相应的值要大一些。再一次，我们将使用 @x 的索引作为 \$_ 项：

```
my @bigger_indices = grep {
    if ($_ > ${\@y} or ${\@x[$_] > ${\@y[$_]}) {
        1; # yes, select it
    } else {
        0; # no, don't select it
    }
} 0..$#x;
my @bigger = @x[@bigger_indices];
```

在 grep 语句中，\$_ 从 0 变化到 @x 数组的最高索引值。如果元素数目超出 @y 数组的边界，将自动选择它。否则，查看两个数组的各个对应值，仅选取匹配该条件的那些值。

然而，这样比需要的更繁琐。我们能够返回布尔表达式而不是确切的 1 或者 0：

```
my @bigger_indices = grep {
    $_ > ${\@y} or ${\@x[$_] > ${\@y[$_]};
} 0..$#x;
my @bigger = @x[@bigger_indices];
```

注 3：一句著名的计算格言是这么说的：“没有哪个问题能够复杂到适当地添加额外的间接层之后，还没有办法解决。” 随着间接层变得模糊，就会在某处神奇地出现一个中间结果。

如果要更容易，可以忽略建立中间数组的步骤，通过 map 返回感兴趣的项：

```
my @bigger = map {
    if ($_ > $y or $x[$_] > $y[$_]) {
        $x[$_];
    } else {
        ();
    }
} 0..$#x;
```

如果索引是正确的，返回结果数组的值。如果索引是错误的，返回一个空列表，使操作项消失。

6.6 选择和改变复杂数据

我们可以在更复杂的数据上使用这些操作符。以第 5 章中的 provisions 列表为例（见图 6-3）：

```
my %provisions = (
    'The Skipper'  => [qw(blue_shirt hat jacket preserver sunscreen)],
    'The Professor' => [qw(sunscreen water_bottle slide_rule batteries radio)],
    'Gilligan'       => [qw(red_shirt hat lucky_socks water_bottle)],
);
```

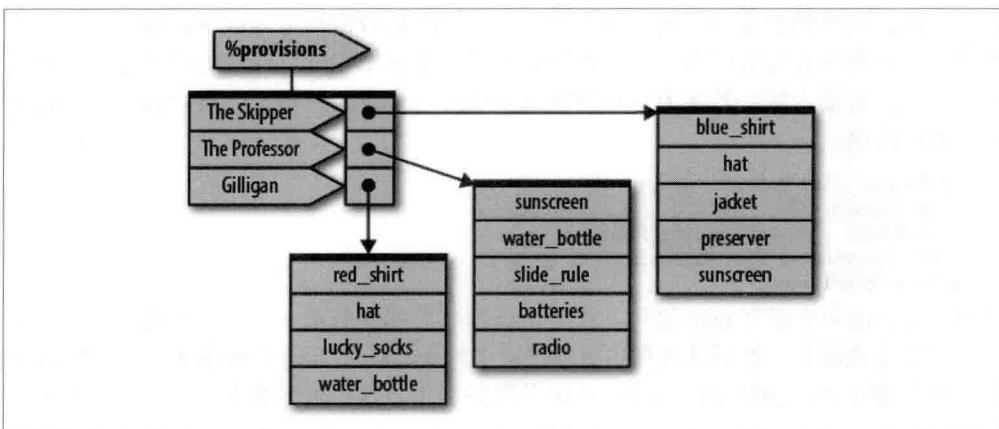


图 6-3 provisions 的 PeGS

此时，\$provisions{"The Professor"}提供一个由 Professor 引入的供应项数组引用，\$provisions{"Gilligan"}[-1]提供 Gilligan 想要导入的最后一项。

我们针对这些数据运行一系列查询。谁提供的项少于 5 个？

```
my @packed_light = grep {@{ $provisions{$_} } < 5, keys %provisions;
```

此时此刻，\$_是人员姓名。通过该人员的姓名，查找该人员对应的供应项数组引用，

在标量上下文中对其解引用，以获取供应项的数目，然后将其与数字 5 相比。我们将会知道，唯一的人员姓名是 Gilligan。

下面是一个更棘手的问题。谁携带饮水壶（water bottle）？

```
my @all_wet = grep {  
    my @items = @{ $provisions{$_} };  
    grep $_ eq 'water_bottle', @items;  
} keys %provisions;
```

再次从姓名列表（keys %provisions）开始，首先取出所有打包的项，然后使用一个内层 grep 操作符处理列表，统计与 water_bottle 项相等的计数。如果计数为 0，也就是说没有水壶，因此对于外层 grep 操作符而言，结果是错误的。如果总数非 0，就说明有一个水壶，因此对于外层 grep 操作符而言，结果是正确的。现在我们看到 Skipper 将在后期会感到有一点口渴，而没有任何减轻。

我们也能够对数据进行格式转换。例如，我们能够将一个散列转换为一个数组引用列表，其中的每个数组包含两个元素。第一个元素是初始人员的姓名；第二个元素是对于该人员的供应项数组的引用：

```
my @remapped_list = map {  
    [ $_ => $provisions{$_} ];  
} keys %provisions;
```

%provisions 的键是人员姓名。对于每个姓名，构建一个两元素的列表，其中包含姓名和相应的供应数组的引用。由于这个列表在一个匿名数组构造函数中，因此为每个人取回一个新创建数组的引用。三个姓名在内，三个引用在外。或者能够通过另一种不同的方式。将输入散列转换为一系列数组引用。每个数组将拥有人员姓名和他们携带的一个供应项：

```
my @person_item_pairs = map {  
    my $person = $_;  
    my @items = @{ $provisions{$person} };  
    map [$person => $_], @items;  
} keys %provisions;
```

是的，map 语句中嵌套 map 语句。外层 map 操作符每次选取一个人员姓名。在\$person 变量中保存该姓名，然后从散列中提取供应项列表。内层 map 操作符遍历该供应项列表，执行表达式，为每个供应项构造匿名数组引用。这个匿名数组包含人员姓名和供应项。

我们不得不在此使用\$person 临时存储外层的\$_ 变量。否则，我们不能查询到外层 map 操作符和内层 map 操作符的临时值。

6.7 练习

可以在附录中的“第 6 章答案”部分找到这些练习的答案。

1. [20 分钟] 第 5 章的练习 2 需要在每次运行时读取整个数据文件。然而，Professor 每天有一个新的路由器日志文件，并且不希望把所有数据保存在一个巨型文件中，这样需要越来越长的时间处理。

修改该程序，继续运行计算数据文件中总数的函数，因此 Professor 可以处理每天的日志文件，得到新的结果。可以在此使用 Storable 模块。

2. [20 分钟] 修改练习 1 中的程序，使用 JSON 模块替换 Storable 模块实现。

第 7 章

对子例程的引用

目前，你已经看到了对 Perl 三种数据类型的引用：标量、数组和散列。同样，也可以对一个子例程进行引用（有时称其为代码引用）。

这样做的原因是什么？在对数组取引用时，可以使同一代码在不同时间处理不同的数组。与之相同，对子例程取引用可以让同一代码在不同时间调用不同的子例程。同样，引用允许构造复杂的数据结构。一个指向子例程的引用，使子例程实际上成为复杂数据结构的一部分成为可能。

换一种说法，一个变量或者一个复杂数据结构是整个程序的数值仓库。一个子例程的引用可以被想象成一个程序中的行为仓库。本节的例子将揭示这一点。

7.1 对命名子例程的引用

Skipper 和 Gilligan 之间有一段对话：

```
sub skipper_greets {
    my $person = shift;
    print "Skipper: Hey there, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq "Skipper") {
        print "Gilligan: Sir, yes, sir, $person!\n";
    } else {
        print "Gilligan: Hi, $person!\n";
    }
}

skipper_greets("Gilligan");
gilligan_greets("Skipper");
```

其输出结果如下：

```
Skipper: Hey there, Gilligan!
Gilligan: Sir, yes, sir, Skipper!
```

到现在为止，一切正常。然而，注意，Gilligan 有两个不同的行为，这取决于它是对 Skipper 说话，还是对其他人说话。

现在，Skipper 和 Gilligan 准备与 Professor 打招呼：

```
skipper_greets('Professor');
gilligan_greets('Professor');
```

其输出是：

```
Skipper: Hey there, Professor!
Gilligan: Hi, Professor!
```

这下轮到 Professor 要作出回应。

```
sub professor_greets {
    my $person = shift;
    print "Professor: By my calculations, you must be $person!\n";
}

professor_greets('Gilligan');
professor_greets('Skipper');
```

输出结果是：

```
Professor: By my calculations, you must be Gilligan!
Professor: By my calculations, you must be Skipper!
```

该程序真麻烦，而且一点也不通用。如果每个乘客的行为以不同的子例程命名，并且当有新乘客到来时，我们不得不指定需要调用的子例程。当然，可以用大量难以维护的代码来处理这件事，但是，就像对于数组和散列所做的处理那样，只要加一些小技巧，就可以简化处理过程。

首先，使用“取引用”操作符。实际上这也不用介绍，因为它与之前的反斜杠的形式相同：

```
my $ref_to_greeter = \&skipper_greets;
```

我们现在对子例程 skipper_greets() 取引用。注意，前导字符“&”在这里是必需的，但有意删除了其后的小括号。Perl 会把这个子例程的引用（coderef）存储于标量 \$ref_to_greeter 中，并且同其他引用一样，它适用于可以使用标量的任何地方。

通过解引用还原一个子例程引用的唯一目的就是调用它。对代码引用的解引用和对其他数据类型引用的解引用是相似的。首先，可以采用在知道引用以前的方法来处理（包括可选的前缀&号）：

```
& skipper_greets( 'Gilligan' )
```

下一步，把子例程的名字用大括号外加大括号内引用的名字替换：

```
& { $ref_to_greeter }( 'Gilligan' )
```

这样就有了它。这条语句调用当前被 \$ref_to_greeter 引用的子例程，并把它传递给单个

字符串参数：Gilligan。

不过，这样是不是太丑陋了？幸运的是，同样的简化规则也适用于此。如果大括号里的值是简单的标量变量，大括号可以删除：

```
& $ref_to_greeter ( 'Gilligan' )
```

也可以把它转换成带箭头的形式：

```
$ref_to_greeter -> ( 'Gilligan' )
```

最后一种形式特别适用于在一个大数据结构中进行代码引用，你一会儿就会看到。

如果让 Gilligan 和 Skipper 向 Professor 问好，我们只需要迭代调用所有的子例程：

```
for my $greet (\&skipper_greets, \&gilligan_greets) {
    $greet->('Professor');
}
```

首先，在小括号里面，创建一个包含两个元素的列表，而且每个元素是一个代码引用，每个代码引用都各自被解引用，即调用相应的子例程并且传递"Professor"字符串。

我们已经看到在标量变量中的代码引用，并且将代码引用作为列表中的一个元素。是否可以把这些代码引用放入一个更大的数据结构中呢？当然可以。可以创建一个表，映射乘客与他们相互问候的行为，然后使用表重写之前的例子：

```
sub skipper_greets {
    my $person = shift;
    print "Skipper: Hey there, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq 'Skipper') {
        print "Gilligan: Sir, yes, sir, $person!\n";
    } else {
        print "Gilligan: Hi, $person!\n";
    }
}

sub professor_greets {
    my $person = shift;
    print "Professor: By my calculations, you must be $person!\n";
}

my %greets = (
    Gilligan => \&gilligan_greets,
    Skipper   => \&skipper_greets,
    Professor => \&professor_greets,
);

for my $person (qw(Skipper Gilligan)) {
    $greets{$person}->('Professor');
}
```

注意，变量 \$person 是一个名字，在散列中查找它，以调用相应的代码引用。然后对那个代码引用进行解引用操作，传递将要问候的人名，并且获得正确的行为，其输出结

果如下：

```
Skipper: Hey there, Professor!  
Gilligan: Hi, Professor!
```

现在可以在一个气氛十分友好的房间里让大家互相问候了：

```
sub skipper_greets {  
    my $person = shift;  
    print "Skipper: Hey there, $person!\n";  
}  
  
sub gilligan_greets {  
    my $person = shift;  
    if ($person eq 'Skipper') {  
        print "Gilligan: Sir, yes, sir, $person!\n";  
    } else {  
        print "Gilligan: Hi, $person!\n";  
    }  
}  
  
sub professor_greets {  
    my $person = shift;  
    print "Professor: By my calculations, you must be $person!\n";  
}  
  
my %greets = (  
    Gilligan => \&gilligan_greets,  
    Skipper => \&skipper_greets,  
    Professor => \&professor_greets,  
);  
  
my @everyone = sort keys %greets;  
for my $greeter (@everyone) {  
    for my $greeted (@everyone) {  
        $greets{$greeter}->($greeted)  
        unless $greeter eq $greeted; # no talking to yourself  
    }  
}
```

其输出结果如下所示：

```
Gilligan: Hi, Professor!  
Gilligan: Sir, yes, sir, Skipper!  
Professor: By my calculations, you must be Gilligan!  
Professor: By my calculations, you must be Skipper!  
Skipper: Hey there, Gilligan!  
Skipper: Hey there, Professor!
```

这有些复杂。我们让他们一个个走进房间。

```
sub skipper_greets {  
    my $person = shift;  
    print "Skipper: Hey there, $person!\n";  
}  
  
sub gilligan_greets {  
    my $person = shift;  
    if ($person eq 'Skipper') {
```

```

    print "Gilligan: Sir, yes, sir, $person!\n";
} else {
    print "Gilligan: Hi, $person!\n";
}
}

sub professor_greets {
    my $person = shift;
    print "Professor: By my calculations, you must be $person!\n";
}

my %greets = (
    Gilligan => \&gilligan_greets,
    Skipper   => \&skipper_greets,
    Professor => \&professor_greets,
);

my @room; # initially empty
for my $person (qw(Gilligan Skipper Professor)) {
    print "\n";
    print "$person walks into the room.\n";
    for my $room_person (@room) {
        $greets{$person}->($room_person); # speaks
        $greets{$room_person}->($person); # gets reply
    }
    push @room, $person; # come in, get comfy
}

```

输出结果是在热带岛屿上典型的一天：

```

Gilligan walks into the room.

Skipper walks into the room.
Skipper: Hey there, Gilligan!
Gilligan: Sir, yes, sir, Skipper!

Professor walks into the room.
Professor: By my calculations, you must be Gilligan!
Gilligan: Hi, Professor!
Professor: By my calculations, you must be Skipper!
Skipper: Hey there, Professor!

```

7.2 匿名子例程

在上一个例子中，我们并没有显式地调用子例程，例如 `profressor_greets`；我们只是通过代码引用间接来调用它们。所以，为了初始化一个仅在其他地方使用一次的数据结构，而给子例程提供名字纯属浪费脑细胞。但是，就像我们可以创建匿名数组和匿名散列一样，我们也能够创建匿名子例程！

让我们再添加岛上的一个居民：Ginger。但是不同于用一个命名子例程来定义她的行为，我们创建一个匿名子例程：

```

my $ginger = sub {
    my $person = shift;
    print "Ginger: (in a sultry voice) Well hello, $person!\n";
};
$ginger->('Skipper');

```

匿名子例程看上去像是普通子例程，只是在 `sub` 关键字和紧随的代码块之间没有名字（或原型声明）。因为这同样是语句的一部分，所以通常需要拖着一个分号，或者其他表达式的分隔符在结尾。

```
sub { ... body of subroutine ... };
```

`$ginger` 的值是一个代码引用，正如我们在其后定义了后面的块作为子例程，然后对该子例程取引用。当我们到达最后一行语句时，我们看到：

```
Ginger: (in a sultry voice) Well hello, Skipper!
```

尽管我们可以把值作为标量变量存储，但我们还可以直接把 `sub {...}` 结构的代码块放入 `greetings` 散列的初始化中：

```

my %greets = (
    Skipper => sub {
        my $person = shift;
        print "Skipper: Hey there, $person!\n";
    },
    Gilligan => sub {
        my $person = shift;
        if ($person eq 'Skipper') {
            print "Gilligan: Sir, yes, sir, $person!\n";
        } else {
            print "Gilligan: Hi, $person!\n";
        }
    },
    Professor => sub {
        my $person = shift;
        print "Professor: By my calculations, you must be $person!\n";
    },
    Ginger => sub {
        my $person = shift;
        print "Ginger: (in a sultry voice) Well hello, $person!\n";
    },
);

my @room; # initially empty
for my $person (qw(Gilligan Skipper Professor Ginger)) {
    print "\n";
    print "$person walks into the room.\n";
    for my $room_person (@room) {
        $greets{$person}->($room_person); # speaks
        $greets{$room_person}->($person); # gets reply
    }
    push @room, $person; # come in, get comfy
}

```

注意这简化了多少行代码。子例程的定义在右边直接引用的数据结构之中。结果相当直观：

```
Gilligan walks into the room.
```

```
Skipper walks into the room.
```

```
Skipper: Hey there, Gilligan!
```

```
Gilligan: Sir, yes, sir, Skipper!
```

```
Professor walks into the room.
```

```
Professor: By my calculations, you must be Gilligan!
```

```
Gilligan: Hi, Professor!
```

```
Professor: By my calculations, you must be Skipper!
```

```
Skipper: Hey there, Professor!
```

```
Ginger walks into the room.
```

```
Ginger: (in a sultry voice) Well hello, Gilligan!
```

```
Gilligan: Hi, Ginger!
```

```
Ginger: (in a sultry voice) Well hello, Skipper!
```

```
Skipper: Hey there, Ginger!
```

```
Ginger: (in a sultry voice) Well hello, Professor!
```

```
Professor: By my calculations, you must be Ginger!
```

添加更多的旅客就变成了简单地把问候行为放入散列中，并把他们添加到进入房间的人名清单中。我们在效率上得到提升，因为我们把行为保存为数据，并通过它可以查找和迭代，这要感谢友好的子例程引用。

7.3 回调

一个子例程引用经常被用于回调。回调定义在一个算法中当子例程运行到一个特定位时所做的事情。它给我们一个机会来提供自己的子例程。回调在如下情况下使用。

例如：File::Find 模块导出一个 find 子例程，它用来以可移植的方式非常高效地遍历给定文件系统的层次结构。在其最简单的形式中，传给 find 子例程两个参数：一个表示目录开始点的字符串，另一个是对子例程的引用。该子例程会从给定的起始目录开始，通过递归搜索的方法，找到其下的每个文件或目录，并对它们“干些什么”。“干些什么”指定为子例程引用。

```
use File::Find;
sub what_to_do {
    print "$File::Find::name found\n";
}
my @starting_directories = qw(.);

find(\&what_to_do, @starting_directories);
```

find 例程开始于当前目录（.），并且定位所有目录或文件。对于找到的每个条目，我们会调用子例程 what_to_do()，通过全局变量给它传递一些记录值。特别是，\$File::Find::name 的值是项目的完整路径名（以开始搜索的目录为起点）。

在此例中，作为 find 例程参数传递数据（开始目录的列表）和行为。

为只使用一次的子例程起个名字好像有些愚蠢，所以用匿名子例程编写之前的代码，例如：

```
use File::Find;
my @starting_directories = qw(.);

find(
    sub {
        print "$File::Find::name found\n";
    },
    @starting_directories,
);
```

7.4 闭包

我们还可以用 File::Find 来查找文件的一些其他属性，比如文件大小。为了回调方便，当前工作目录被设为文件所在的目录，目录中的文件名也放在变量\$_ 中。

在前面的代码中，使用\$File::Find::name 作为返回的文件名。哪个名字是真的，\$_ 还是 \$File::Find::name? \$File::Find::name 给出文件自起始搜索目录的相对路径名，而在回调程序中，工作目录就是条目所在目录。例如，假定我们想要用 find() 在当前工作目录找一些文件，所以我们给出（"."）作为要搜索的目录列表。如果调用 find()，当前工作目录是 /usr 时，则 find 例程会往下找这个目录。当 find 定位到/usr/bin/perl 时，当前工作目录（在回调程序中）是 /usr/bin。变量\$_ 保存 perl，并且\$File::Find::name 保存 ./bin/perl，这就是相对起始搜索目录的相对路径。

所有这一切都意味着进行文件测试，例如-s，对于即时查询的文件自动报表。尽管这很方便，但在回调中的当前目录与搜索的起始目录是不同的。

如果我们想要使用 File::Find 模块累加查询所得到的所有文件大小，该如何实现？callback 子例程不能使用参数，而且 caller 子例程忽略返回的结果。但这没关系，当解引用时，如果子例程引用被调用，子例程引用就能够查看所有可见的词法变量。例如：

```
use File::Find;

my $total_size = 0;
find(sub { $total_size += -s if -f }, '.');
print $total_size, "\n";
```

与之前一致，当调用 find 子例程时，使用两个参数：匿名子例程的引用和起始目录。当在目录（或者它的子目录中）中找到需要查询的文件名时，调用匿名子例程处理。

该匿名子例程访问\$total_size 变量。在匿名子例程作用域的外部声明该变量，但该变量对于子例程仍然是可见的。因此，尽管 find 子例程调用回调子例程（这将不会直接访问\$total_size），回调子例程仍然访问和更新变量。

这种能够访问声明时就存在的所有词法变量的子例程叫做闭包（从数学领域借用的专有名词）。在 Perl 术语中，闭包是一个包含已经超出作用域的词法变量的子例程。

更进一步，从闭包内部访问变量能够确保只要子例程引用存在，访问的变量就存在。例如，对输出的文件进行计数：

```
use File::Find;  
  
my $callback;  
{  
    my $count = 0;  
    $callback = sub { print ++$count, ": $File::Find::name\n" };  
}  
find($callback, '.');
```

声明一个保存回调匿名子例程引用的变量。但不能在裸块中声明该变量（后续的语句块不是大型 Perl 语法结构的一部分），否则 perl 将在语句块末尾回收变量。下一步，词法变量\$count 初始化为 0。然后声明一个匿名子例程，并且将它的引用放入\$callback。该匿名子例程是一个闭包，因为它指向的是在块末尾超出作用域的词法变量\$count。记住匿名子例程之后的分号；这是一条语句，而不是通常的子例程定义。

在裸块末尾，\$count 变量超出作用域。然而，因为该变量仍由\$callback 变量指向的匿名子例程所引用，所以它仍然以匿名标量变量的形式存在。当\$callback 变量被 find 子例程调用时，之前叫做\$count 变量的值开始从 1 到 2 到 3，持续增长。



注意

闭包声明增加了相应用对象的引用计数，就像是另一个引用被显式调用一样。在裸块的末尾，\$count 的引用计数为 2，但在块语句运行结束后，引用计数的值仍然为 1。尽管没有其他代码访问\$count，但是它仍将保留在内存中，直到子例程的引用可以在\$callback 或者其他地方使用。

7.5 从一个子例程返回另一个子例程

尽管裸语句块能够完美地定义回调，但是使子例程返回值为该子例程的引用可能更佳：

```
use File::Find;  
  
sub create_find_callback_that_counts {  
    my $count = 0;  
    return sub { print ++$count, ": $File::Find::name\n" };  
}  
  
my $callback = create_find_callback_that_counts();  
find($callback, '.');
```

此处的操作过程是相同的，只是编写方法略有不同。当调用 `create_find_callback_that_counts` 子例程时，将词法变量 `$count` 初始化为 0。来自该子例程的返回值是一个匿名子例程的引用，因为它访问 `$count` 变量，所以该匿名子例程引用同样也是一个闭包。即使 `$count` 变量在 `create_find_callback_that_counts` 子例程末尾超出作用域，在该变量和返回的子例程引用间也仍有绑定，因此变量仍然存在，直到子例程引用最终被销毁。

如果重用回调，同样的变量也将拥有它最近使用的值。如下所示，在初始子例程 (`create_find_callback_that_counts`) 处发生初始化，而不是 `callback` (匿名) 子例程：

```
use File::Find;

sub create_find_callback_that_counts {
    my $count = 0;
    return sub { print ++$count, " : $File::Find::name\n" };
}

my $callback = create_find_callback_that_counts();
print "my bin:\n";
find($callback, 'bin');
print "my lib:\n";
find($callback, 'lib');
```

以上示例对于 `bin` 目录下的所有文件输出从 1 开始的连续计数，当进入 `lib` 目录时，继续对所有文件计数。这两种情形下使用相同的 `$count` 变量。然而，如下所示，如果调用 `create_find_callback_that_counts` 子例程两次，将得到不同的 `$count` 变量：

```
use File::Find;

sub create_find_callback_that_counts {
    my $count = 0;
    return sub { print ++$count, " : $File::Find::name\n" };
}

my $callback1 = create_find_callback_that_counts( );
my $callback2 = create_find_callback_that_counts( );
print "my bin:\n";
find($callback1, 'bin');
print "my lib:\n";
find($callback2, 'lib');
```

如上例，现在拥有两个独立的 `$count` 变量，每一个都能被它们自身的回调子例程访问。

如何通过回调得到所有文件大小的总数呢？在前一章的示例中，做法是使 `$total_size` 变量全局可见。如果把将 `$total_size` 变量的定义插入返回值为回调引用的子例程中，我们将不能访问该变量。但是我们能够做点手脚。首先，我们能够确定将绝不会使用任何参数调用该回调子例程，如下所示，因此如果该子例程收到任意一个参数，就返回总字节数：

```

use File::Find;

sub create_find_callback_that_sums_the_size {
    my $total_size = 0;
    return sub {
        if (@_) { # it's our dummy invocation
            return $total_size;
        } else { # it's a callback from File::Find:
            $total_size += -s if -f;
        }
    };
}

my $callback = create_find_callback_that_sums_the_size( );
find($callback, 'bin');
my $total_size = $callback->('dummy'); # dummy parameter to get size
print "total size of bin is $total_size\n";

```

通过参数的存在或者不存在来识别程序行为并不是一个通用的解决方案。如下所示，很幸运，我们能够在 `create_find_callback_that_counts` 子例程中创建多个子例程引用：

```

use File::Find;

sub create_find_callbacks_that_sum_the_size {
    my $total_size = 0;
    return(sub { $total_size += -s if -f }, sub { return $total_size });
}

my ($count_em, $get_results) = create_find_callbacks_that_sum_the_size( );
find($count_em, 'bin');
my $total_size = &$get_results( );
print "total size of bin is $total_size\n";

```

因为我们通过同样的作用域创建了两个子例程引用，并且它们都能够访问同一个 `$total_size` 变量。尽管在调用这两个子例程中的任意一个子例程之前，变量已经超出作用域，但它们仍然能够分享这个相同的继承得到的变量，并且能够使用该变量与计算的结果交互。

通过创建子例程来返回两个子例程引用将不会调用它们。此时的引用仅仅是数据。直到作为回调程序或者显式的子例程解引用调用它们，它们才会真正地执行。

如果多次调用这个新的子例程将会发生什么？

```

use File::Find;

sub create_find_callbacks_that_sum_the_size {
    my $total_size = 0;
    return(sub { $total_size += -s if -f }, sub { return $total_size });
}

## set up the subroutines
my %subs;
foreach my $dir (qw(bin lib man)) {
    my ($callback, $getter) = create_find_callbacks_that_sum_the_size( );
    $subs{$dir}{CALLBACK} = $callback;
    $subs{$dir}{GETTER} = $getter;
}

```

```

}

## gather the data
for (keys %subs) {
    find($subs{$_}{CALLBACK}, $_);
}

## show the data
for (sort keys %subs) {
    my $sum = $subs{$_}{GETTER}->();
    print "$_ has $sum bytes\n";
}

```

在建立子例程部分，成对创建了三个 callback-and-getter 实例。每个回调程序都有一个相应的子例程用于获取结果。下一步，在收集数据部分，通过每个相应的回调子例程引用调用 `find` 例程三次。这将更新与每个回调程序相关联的独立的`$total_size` 变量。最后，在显示数据部分，通过调用 `getter` 子例程获取结果。

6 个子例程（和它们共享的三个`$total_size` 变量）都是引用计数。当修改`%subs` 或者它运行超出作用域时，拥有引用计数的值将减少，并且回收所包含的数据（如果数据同样引用其他数据，其他数据的这些引用计数也将相应地减少。）

7.6 作为输入参数的闭包变量

上一节的示例展示了闭包变量是如何修改的，闭包变量也同样用于为子例程提供初始变量或者持续输入。例如，我们编写一个子例程用于创建 `File::Find` 回调，该回调程序输出容量超过规定大小的文件名称：

```

use File::Find;

sub print_bigger_than {
    my $minimum_size = shift;
    return sub { print "$File::Find::name\n" if -f and -s >= $minimum_size };
}

my $bigger_than_1024 = print_bigger_than(1024);
find($bigger_than_1024, 'bin');

```

将数字 1024 作为参数传入 `print_bigger_than` 子例程，然后子例程通过 `shift` 语句将参数传入词法变量`$minimum_size` 中。因为通过 `print_bigger_than` 变量的返回值，访问在子例程之内引用的变量，所以该变量为闭包变量，值在子例程引用存在的时间内都将保持。再一次，多次调用该子例程，为`$minimum_size` 变量创建独特的“锁定”值，每个值都与它们各自相应的子例程引用绑定。

因为词法变量最终将超出作用域，所以闭包仅仅是在词法变量上“封闭的”。又因为包变量（该包变量为全局变量）不会超出作用域，所以闭包绝不会关闭一个包变量。对于所有的子例程而言，它们都引用全局变量的同一个实例。

为了在本书中展示这些内容，创建了 File::Find::Closures 模块，每个都能返回两个闭包的一批生成器子例程。如下所示，其中一个闭包用于 find，另一个用于获取匹配文件的列表：

```
use File::Find;
use File::Find::Closures;

my( $wanted, $list_reporter ) = find_by_name( qw(README) );
find( $wanted, @directories );

my @readmes = $list_reporter->();
```

我们没有打算要所有人真正使用这个模块甚至于从中窃取信息。如下所示的 find_by_min_size 子例程创建一个闭包，用于查找任何大于或者等于输入字节数的文件：

```
use File::Spec::Functions qw(canonpath no_upwards);

sub find_by_min_size {
    my $min   = shift;
    my @files = ();

    sub { push @files, canonpath( $File::Find::name )
          if -s $_ >= $min };
    sub { @files = no_upwards( @files );
          wantarray ? @files : [ @files ] }
}
}
```

我们能够轻易地根据需要修改这些代码，然后将其放入我们的程序中。

7.7 闭包变量作为静态局部变量

子例程并不一定要是匿名子例程才能成为闭包。如果一个命名子例程访问词法变量并且这些变量超出作用域，命名子例程保留一个指向词法变量的引用，正如用匿名子例程展示的一样。例如，考虑如下两个为 Gilligan 统计椰子的例程：

```
{
    my $count;
    sub count_one { ++$count }
    sub count_so_far { return $count }
}
```

如果在程序开始部分放置这些代码，就在裸块作用域内部声明变量\$count。引用该变量的两个子例程就变成闭包。然而，因为它们都有一个名称，所以它们将会像其他命名子例程一样，直到块作用域结束仍将保留该名称。因为子例程在作用域外仍将保持，并且在该作用域中访问声明的变量，所以它们就成为闭包，因此我们可以在本程序的生命周期内持续访问\$count 变量。

所以，经过几次调用，我们可以看到计数的增长：

```
count_one();
count_one();
count_one();
print 'we have seen ', count_so_far(), " coconuts!\n";
```

\$count 变量在调用 count_one 或者 count_so_far 之间保留它的值，但是根本没有其他部分的代码能够访问这个\$count 变量。

如果要递减计数该如何操作？如下所示的代码将会实现递减计数：

```
{
my $countdown = 10;
sub count_down { $countdown-- }
sub count_remaining { $countdown }
}

count_down();
count_down();
count_down();
print "we're down to ", count_remaining(), " coconuts!\n";
```

也就是说，只要将它放入程序的起始部分，在 count_down 子例程或者 count_remaining 子例程的任何调用之前，程序就能工作。为什么？

因为在第一行程序有两个功能，所以当将如下语句放在调用语句之后时，该语句块将不会工作：

```
my $countdown = 10;
```

\$countdown 声明的一部分功能是作为词法变量，在编译阶段，该部分作为已经解析的程序被识别和处理。第二部分是将 10 赋值到相应的存储空间，在运行阶段，该部分作为 perl 解释器执行的代码。除非 perl 在运行阶段执行这些代码，否则该变量的初始值始终为 undef。

该问题的一个实际解决方案是把静态局部变量所在的语句块转换成 BEGIN 语句块：

```
BEGIN {
    my $countdown = 10;
    sub count_down { $countdown-- }
    sub count_remaining { $countdown }
}
```

BEGIN 关键字告诉 Perl 编译器：一旦成功解析到该语句块的位置（在编译阶段），就立即跳转到运行阶段，并且也运行该语句块。如果执行该 BEGIN 语句块中的语句没有产生致命错误，就会编译然后继续执行后续语句块中的语句。该语句块本身也被丢弃，保证其中的代码在程序中精确地执行一次，即使它依照语法规则在一个循环语句或者子例程中出现。

state 变量

Perl v5.10 版开始为子例程引入了另一种方法生成私有的、持续的变量。我们在 *Learning Perl* 一书中介绍过这些内容，但是在此将做一个扼要的回顾。如下所示，可

以使用 state 语句在子例程内部声明变量，而不必使用为词法变量创建 BEGIN 语句块的方法构造作用域：

```
use v5.10;
sub countdown {
    state $countdown = 10;
    $countdown--;
}
```

我们能够在某些人一般不认为是“子例程”的某些地方使用 state 语句，比如 sort 语句块。如下所示，如果我们想要查看这些数字比较的结果，就能够按照下列方式跟踪这些比较数字，而不必在 sort 语句块外部额外创建一个变量：

```
use v5.10;

my @array = qw( a b c d e f 1 2 3 );

print sort {
    state $n = 0;
    print $n++, ": a[$a] b[$b]\n";
    $a cmp $b;
} @array;
```

也能够在 map 语句块中使用 state 语句。如果我们想要对行排序，但仍然想记住它们的初始位置，就能够使用 state 变量跟踪行号：

```
use v5.10;

my @sorted_lines_tuples =
    sort { $b->[1] <=> $a->[1] }
    map { state $l = 0; [ $l++, $_ ] }
        <>;
```

然而，state 变量有一个限制：到目前为止，我们仅能使用 state 语句初始化标量变量。我们能够使用 state 语句声明任意类型的变量，但是不能初始化它们：

```
use v5.10;
sub add_to_tab {
    state @castaways = qw(Ginger Mary Ann Gilligan); # compilation error
    state %tab = map { $_, 0 } @castaways; # compilation error
    $countdown{'main'}--;
}
```

如果我们只希望使用一次，而不是每次运行子例程，初始化这些变量将变得很麻烦。为了避免这些麻烦，我们将坚持使用标量。但是，等一下！引用是标量，因此我们能够初始化数组引用或者散列引用：

```
use v5.10;
sub add_to_tab {
    my $castaway = shift;
    state $castaways = qw(Ginger Mary Ann Gilligan); # works!
    state %tab = map { $_, 0 } @$castaways; # works!
    $tab->{$castaway}++;
}
```

7.8 查询我们自己的身份

匿名子例程有一个身份问题：它们不知道它们是谁！虽然我们不在乎它们是否有名称，但是当需要告诉我们它们的名称是什么时，有一个名称就显得很便捷。假定我们想要使用匿名子例程编写一个递归子例程。当它还没有完成创建时，我们将使用什么名称再次调用相同的子例程？

```
my $countdown = sub {
    state $n = 5;
    return unless $n > -1;
    say $n--;
    WHAT_NAME??->();
};
```

这能够按照以下两个步骤完成，因此保存引用的变量已经存在：

```
my $countdown;
$countdown = sub {
    state $n = 5;
    return unless $n > -1;
    say $n--;
    $countdown->();
};
$countdown->();
```

输出为递减计数：

```
5
4
3
2
1
0
```

以上程序能够工作，因为 Perl 解释器不在乎\$countdown 变量的内容，直到该变量实际上需要使用它。如下所示，为了避开这一点，Perl v5.16 版引入__SUB__ 标记 (token)，返回一个指向当前子例程的引用：

```
my $sub = sub {
    state $n = 5;
    return unless $n > -1;
    say $n--;
    __SUB__->();
};
$sub->();
```

以下代码也能与命名子例程一起使用：

```
sub countdown {
    state $n = 5;
    return unless $n > -1;
    say $n--;
    __SUB__->();
};
countdown();
```

7.8.1 令人着迷的子例程

我们将如何调试匿名子例程？当我们开始自由地传递它们时，如何知道正在传递的是哪一个？对于标量的值、数组的值和散列的值，我们能够按照如下方式输出它们^{注1}：

```
use v5.10;

my @array = ( \ 'xyz', [qw(a b c)], sub { say 'Buster' } );

foreach ( @array ) {
    when( ref eq ref \ '' ) { say "Scalar $$" }
    when( ref eq ref [] ) { say "Array @$" }
    when( ref eq ref sub {} ) { say "Sub ???" }
}
```

除了匿名子例程以外，还可以通过输出得到究竟是什么在里面，对于匿名子例程，我们必须实际运行它才能得到它内部的值，而不能够仅仅解引用：

```
Scalar xyz
Array a b c
Sub ???
```

在此做一些有用的事情并不难，但是我们不得不拿出很多在此前没有完整介绍过的“黑魔法”。这些可是高级功夫，因此，如果你对此感到不适，请不必担心。反正这不是你需要每天完成的编程任务。



注意

对于更详细信息，查阅 overload 模块。

首先，我们在这里将匿名子例程作为对象使用，尽管我们没有告诉你如何实现（这将在第 15 章中详细介绍）。在这个对象内部，重载了字符串化。我们将告诉 Perl 解释器如何字符串化该引用，然后也将对它添加一些有用的信息。这些有用的信息将来自于 B 模块，该模块能够观察 Perl 的解析树以完成多种多样的工作。我们将使用 B 模块获取文件和子例程定义的行号。这样我们才能知道去哪儿进一步解决这类问题。下一步，我们将使用 B::Deparse 模块中的 coderef2txt 子例程将 Perl 的内部代码调整回更可读的 Perl 代码。最后，我们将所有这些放到一起，构成子例程引用的字符串版本：

```
use v5.14;

package MagicalCodeRef 1.00 {
    use overload '"' => sub {
        require B;

        my $ref = shift;
        my $gv = B::svref_2object($ref)->GV;

        require B::Deparse;
```

注 1：一些这类实例在作者的博客“Enchant closures for better clebugging output”中首次出现，<http://www.effectiveperlprogramming.com/blog/1345>。

```

my $deparse = B::Deparse->new;
my $code = $deparse->coderef2text($ref);

my $string = sprintf "---code ref---\n%s:%d\n%s\n---",
    $gv->FILE, $gv->LINE, $code;
};

sub enchant { bless $_[1], $_[0] }
}

```

当创建匿名子例程时，就通过新的子类传递它，然后在我们想要对它字符串化时，对它做检查：

```

my $sub = MagicalCodeRef->enchant( sub { say 'Gilligan!!!' } );

my @array = ( \ 'xyz', [qw(a b c)], $sub );

foreach ( @array ) {
    when( ref eq ref '' )      { say "Scalar $$" }
    when( ref eq ref [] )     { say "Array @$" }
    when( ref eq 'MagicalCodeRef' ) { say "Sub $sub" }
}

```

现在，输出显示了子例程定义的位置，以及子例程内部的内容（包括在创建子例程时，所有有效的编译指令的设定）：

```

Scalar xyz
Array a b c
Sub ---code ref---
enchant.pl:26
{
    use warnings;
    use strict;
    no feature;
    use feature ':5.16';
    say 'Gilligan!!!';
}
---
```

这里给出另一个妙计：通过将它变为一个闭包，因此名称的值可以不在子例程中：

```

my $sub = do {
    my $name = 'Gilligan';
    MagicalCodeRef->enchant( sub { say "$name!!!" } );
};

```

现在，因为我们不知道\$name 的值是什么，所以输出就变得不那么有用：

```

Scalar xyz
Array a b c
Sub ---code ref---
debug.pl:28
{
    use warnings;
    use strict;
    no feature;
    use feature ':5.16';
    say "$name!!!";
}
---
```

有一个叫做 PadWalker 的模块，能够回溯 Perl 的解析，以查找这些闭包变量。如下所示，使用该模块的 close_over 子例程获取这些变量的散列，然后用 Data::Dumper 模块输出它们：

```
use v5.14;

package MagicalCodeRef 1.01 {
    use overload '""' => sub {
        require B;

        my $ref = shift;
        my $gv = B::svref_2object($ref)->GV;

        require B::Deparse;
        my $deparse = B::Deparse->new;
        my $code = $deparse->coderef2text($ref);

        require PadWalker;
        my $hash = PadWalker::closed_over( $ref );

        require Data::Dumper;
        local $Data::Dumper::Terse = 1;
        my $string = sprintf "---code ref---\n%s:%d\n%s\n---\n%s---",
            $gv->FILE, $gv->LINE,
            $code,
            Data::Dumper::Dumper( $hash );
    };

    sub enchant { bless $_[1], $_[0] }
}
```

现在可以查看\$name 的值：

```
Scalar xyz
Array a b c
Sub ---code ref---
debug.pl:38
{
    use warnings;
    use strict 'refs';
    BEGIN {
        $^H->{'feature_unicode'} = q(1);
        $^H->{'feature_say'} = q(1);
        $^H->{'feature_state'} = q(1);
        $^H->{'feature_switch'} = q(1);
    }
    say "$name!!!";
}
---
{
    '$name' => \`Gilligan'
}
---
```

还在继续看吗？放轻松些，这些确实是棘手的内容。现在你已经理解它，我们希望你从来不必使用它。我们不希望你的同事陷入一个孤立无援的境地。然而，下一节将揭

示幕后所发生的事。

7.8.2 转储闭包

既然已经描述了转储闭包的“笨办法”，下面将展示更灵巧的方法。这就像是一个谋杀之谜一样，第一嫌犯绝对不是真正的凶手。

Data::Dump::Streamer 模块是增强版的 Data::Dumper 模块，如下所示，而且它甚至能够处理代码引用和闭包：

```
use Data::Dump::Streamer;

my @luxuries = qw(Diamonds Furs Caviar);

my $hash = {
    Gilligan => sub { say 'Howdy Skipper!' },
    Skipper   => sub { say 'Gilligan!!!!' },
    'Mr. Howell' => sub { say 'Money money money!' },
    Ginger    => sub { say $luxuries[rand @luxuries] },
};

Dump $hash;
```

转储散列的值，就得到如下所示的输出（忽略一些令人厌烦的数据）。注意，该语句也转储@luxuries 数组的内容，因为它知道 Gringer 子例程也需要它：

```
my (@luxuries);
@luxuries = (
    'Diamonds',
    'Furs',
    'Caviar'
);
$HASH1 = {
    Gilligan => sub {...},
    Ginger   => sub {
        use warnings;
        use strict 'refs';
        BEGIN {
            $^H{'feature_unicode'} = q(1);
            $^H{'feature_say'} = q(1);
            $^H{'feature_state'} = q(1);
            $^H{'feature_switch'} = q(1);
        }
        say $luxuries[rand @luxuries];
    },
    "Mr. Howell" => sub {...},
    Skipper   => sub {...}
};
```

7.9 练习

可以在附录中的“第 7 章答案”部分找到这些练习的答案。



注意

你不必输入所有的代码。本书中的代码可以通过 <http://www.intermediateperl.com> 网站下载，此时只需从下载的文件中找到一个名叫 ex7-1.pl 的文件。

1. [50 分钟] Professor 在周一下午修改了一些文件，但是现在他忘记修改了哪一些。这类事情总是会发生。他希望你编写一个叫做 `gather_mtime_between` 的子例程，该子例程提供一个开始和结束的时间戳，返回一对代码引用。其中第一项将使用 `File::Find` 模块收集在这个期间内修改过的文件名称；第二项可用于收集所查找到的文件列表。

下面的代码需要尝试；它仅会列出在上星期一最后修改的文件，尽管你可以轻易地修改它为在另一天使用。

提示：可以通过以下代码查找文件的时间戳（`mtime`）：

```
my $timestamp = (stat $file_name)[9];
```

因为这是一个数组切片，所以记住这里的圆括号是必需的。不要忘记，在回调内部的工作目录不必是调用 `find` 方法的起始目录：

```
use File::Find;
use Time::Local;

my $target_dow = 1;          # Sunday is 0, Monday is 1, ...
my @starting_directories = (".");

my $seconds_per_day = 24 * 60 * 60;
my($sec, $min, $hour, $day, $mon, $yr, $dow) = localtime;
my $start = timelocal(0, 0, 0, $day, $mon, $yr);           # midnight today
while ($dow != $target_dow) {
    # Back up one day
    $start -= $seconds_per_day;           # hope no DST! :-)
    if (--$dow < 0) {
        $dow += 7;
    }
}
my $stop = $start + $seconds_per_day;

my($gather, $yield) = gather_mtime_between($start, $stop);
find($gather, @starting_directories);
my @files = $yield->();

for my $file (@files) {
    my $mtime = (stat $file)[9];          # mtime via slice
    my $when = localtime $mtime;
    print "$when: $file\n";
}
```

注意关于 DST 的注释。在世界上的很多地方，在白天缩短或者使用夏令时的日子中，一天就不再是 86 400 秒。该程序掩盖住这些问题，但是更仔细的程序员会采用某种适当的方式处理这类问题。

文件句柄引用

我们已经介绍过通过引用传递数组、散列和子例程，允许添加一个“中间层”用于解决特定类型的问题。我们也能够在引用中存储文件句柄，而且能够不止为文件打开文件句柄。现在，我们先查看一下解决问题的旧方法，然后看看这些问题新的解决方案。

8.1 旧方法

在过去，Perl 使用裸字作为程序员定义的文件句柄名称，而且也使用诸如 STDIN、ARGV 等其他一些特殊的文件句柄。文件句柄是另一种 Perl 数据类型，尽管人们并不将其作为一个数据类型讨论，因为它没有它本身的特殊魔符。你可能已经看到很多使用裸字文件句柄的代码，例如^{注1}：

```
open LOG_FH, '>>', 'castaways.log'  
or die "Could not open castaways.log: $!";
```

如果我们传递这些文件句柄，我们就能够与代码的其他部分共享，比如库文件，将会发生什么呢？你可能见过一些看起来比较复杂的代码，使用符号表（typeglob）或者指向符号表的引用^{注2}：

```
log_message( *LOG_FH, 'The Globetrotters are stranded with us!' );  
  
log_message( \*LOG_FH, 'An astronaut passes overhead' );
```

在 log_message 子例程中，从参数列表中取出第一个元素，然后将它保存于另一个符号表中。符号表存储所有指向该名称包变量的指针，在此没有深入叙述太多的细节。当对符号表做赋值操作时，实际上只是为同样的数据创建别名。我们能够通过其他名称访问数据，其中可以包括文件句柄的全部细节内容。然后，当使用该名称作为文件句

注 1：Learning Perl 介绍了文件句柄的基础知识。

注 2：Mastering Perl 中有更多关于符号表的知识。

柄时，Perl 就知道通过符号表查找文件句柄部分。如果文件句柄有魔符，操作起来就会变得更容易！

```
sub log_message {
    local *FH = shift;

    print FH @_ , "\n";
}
```

注意，在此使用 local 关键字定义变量。符号表和记号表（symbol table）一起工作，这就意味着使用符号表处理包变量。因为包变量不能是词法变量，我们不能在此使用 my 关键字。因为不能破坏脚本中其他部分可能命名为 FH 的文件句柄，所以必须在此使用 local 关键字，将 FH 文件句柄标注为 log_message 子例程在作用域内的临时变量，当子例程运行结束时，Perl 将还原 FH 文件句柄的值，就好像从未使用过一样。

如果所做的这一切使你感到焦虑不安，而且希望所有这一切都不存在，这非常好。再也别这样做了！我们将它放入一个叫做“旧方法”的小节中，是因为现在我们有更好的方法实现同样的功能。假装本节的内容从未存在过，直接跳到下一节。

8.2 改进的方法

从 Perl v5.6 开始，open 命令能够以普通的标量变量形式创建一个文件句柄的引用，前提是只要该标量变量的值未定义。相比于使用裸字表示文件句柄名称，可以按照如下方式，使用一个值未初始化的标量变量作为文件句柄：

```
my $log_fh;
open $log_fh, '>>', 'castaways.log'
or die "Could not open castaways.log: $!";
```

如果标量已经拥有值，以上方式将不能工作，因为 Perl 解释器不会破坏数据。在下面的示例中，Perl 解释器尝试使用数值为 5 的变量作为符号引用（symbolic reference），因此 Perl 解释器就将查找一个名为“5”的文件句柄。也就是说，它将查找该文件句柄，但 strict 编译提示符将阻止这类的操作：

```
my $log_fh = 5;
open $log_fh, '>>', 'castaways.log'
or die "Could not open castaways.log: $!";
print $log_fh "We need more coconuts!\n"; # doesn't work
```

然而，Perl 的习语是做任何事情尽可能一步到位。我们能够在 open 语句的右边声明变量。一开始它看起来有点好笑，但使用了两三次（也可能多次）之后，你会习惯并且喜欢这样的方式：

```
open my $log_fh, '>>', 'castaways.log'
or die "Could not open castaways.log: $!";
```

如下所示，当我们想要将内容输出到文件句柄时，就使用标量变量而不是裸字。注意，文件句柄之后依旧没有逗号：

```
print $log_fh "We have no bananas today!\n";
```

尽管这样的语法可能看起来有些可笑，即使这对你而言看起来并不可笑，它对于可能使后续需要阅读你的代码的人看起来也很奇怪。然而，还有另一个问题。你可能记得`$_`是`print`语句的默认参数，即使你按如下方式指定使用的裸字文件句柄：

```
$_ = 'Salt water batteries';
print;
print STDOUT;
print STDERR;
```

然而，如下所示，如果该默认参数是`print`语句后的唯一参数，Perl解释器不知道在编译时`$log_fh`究竟是什么。这是一个文件句柄还是发送至标准输出的值？Perl解释器会猜测你想要将`$log_fh`的值发送至标准输出：

```
print $log_fh; # sends "GLOB(0x9bcd5c)" to standard output
```

如果我们不小心在标量文件句柄之后放置一个逗号，就将遇到同样的问题。Perl解释器再一次认为你想要将`$log_fh`的值输出到标准输出：

```
# sends "GLOB(0x9bcd5c)Salt water batteries" to standard output
print $log_fh, 'Salt water batteries';
```

在*Perl Best Practices*这本书中，作者Damian Conway建议在文件句柄部分加上大括号，以显式声明我们的意图。这种语法使它看起来更像是带内联语句块的`grep`和`map`：

```
print {$log_fh} "We have no bananas today!\n";
```

现在，我们将文件句柄引用视为与其他标量变量一样。我们不必施展棘手的“魔法”使其工作：

```
log_message( $log_fh, 'My name is Mr. Ed' );

sub log_message {
    my $fh = shift;

    print $fh @_;
}
```

我们也能够通过读操作创建文件句柄引用。如下所示，在第二个参数中放置合适的参数：

```
open my $fh, '<', 'castaways.log'
or die "Could not open castaways.log: $!";
```

现在，我们在行输入操作符中使用标量变量替换裸字。此前，我们可以看到如下所示的钻石操作符（详见*Learning Perl*）中的裸字：

```
while( <LOG_FH> ) { ... }
```

如下所示，现在，我们看到标量变量占据同样的位置：

```
while( <$log_fh> ) { ... }
```

一般，所有我们看到使用裸字文件句柄的位置，都能够使用标量变量的文件句柄引用替换：

```
while( <$log_fh> ) { ... }
if( -t $log_fh ) { ... }
my $line = readline $log_fh;
close $log_fh;
```

在以上的任意形式中，当标量变量超出作用域（或者对它赋其他值）时，Perl解释器关

闭这些文件。我们自己不必显式地关闭文件。

8.3 指向字符串的文件句柄

从 Perl v5.6 开始，我们能够打开一个标量引用形式的文件句柄，而不是一个文件。该文件句柄要么从该字符串读取要么向该字符串写入，而不是文件（或者管道、套接字）。

可能我们需要捕获通常流向文件句柄的输出。如果我们打开一个指向字符串的写入文件句柄并且使用它，如下所示，输出就绝不会真的离开程序，而且我们也不必使用任何技巧捕获它：

```
open my $string_fh, '>', \ my $string;
```

例如，可以保存 CGI.pm 程序的状态，但是不得不向 save 提供一个文件句柄。如果使用字符串句柄\$string_fh，数据绝不应离开程序：

```
use CGI;
```

```
open my $string_fh, '>', \ my $string;
CGI->save( $string_fh );
```

同样地，可以使用 Storable 模块将数据打包到一个字符串中（参考第 6 章），但是 nstore 函数想要将数据保存到一个命名文件中，并且 nstore_fd 函数将数据发送至一个文件句柄。如下所示，如果我们想要在字符串中捕获这些数据，就可以再次使用字符串句柄：

```
use Storable;
```

```
open my $string_fh, '>', \ my $string;
nstore_fd \@data, $string_fh;
```

也可以使用指向字符串的文件句柄，为这些想要塞满整个屏幕的程序，捕获标准输出（STDOUT）或者标准错误输出（STDERR）。有时候，我们只是想要使程序保持安静，而且有时候，我们也只是想要重定向输出。因此，我们必须先关闭标准输出，然后重新打开文件句柄。因为我们通常不想丢失真正的标准输出，所以我们能够将标准输出局部化到一个作用域，这样替换就有一个限定效果：



注意

这类技巧不会通过系统将新版本的标准输出传至外部程序。

```
print "1. This goes to the real standard output\n";

my $string;
{
    local *STDOUT;
    open STDOUT, '>', \ $string;

    print "2. This goes to the string\n";
```

```
$some_obj->noisy_method(); # this STDOUT goes to $string too  
}  
  
print "3. This goes to the real standard output\n";
```

可以灵活地设计我们自己的程序，这样其他人能够决定将他们自己的数据发往何处。如果我们将输出的流向指定为文件句柄，这样就能够决定输出是否进入文件、套接字或者字符串。这种灵活性的最佳之处在于它的实现非常简单，而且如下所示，它们中每个的用法都一致：

```
sub output_to_fh {  
    my( $fh, @data ) = @_;  
    print $fh @data;  
}
```

如果我们想要使 `output_to_fh` 子例程专业化，就必须对它打包，向它提供合适类型的文件句柄：

```
sub as_string {  
    my( @data ) = @_;  
    open my $string_fh, '>', \$my $string;  
    output_to_fh( $string_fh, @data );  
    $string;  
}
```

逐行处理字符串

当像处理文件一样处理字符串时，因为能够使用文件句柄接口，所以很多常见的任务将变得非常简单。例如，如下所示，我们能够使用 `split` 语句将多行字符串分成不同的行：

```
my @lines = split '/', $multiline_string;  
foreach my $line ( @lines ) {  
    ... do something ...  
}
```

然而，到目前为止，我们在两个地方存放数据，而且对当前解决方案的脆弱性有稍许不满，该解决方案说明我们用于分割数据的模式可能不是正确的。相反，如下所示，我们能够打开一个文件句柄，用于从一个标量引用中读取数据，然后通过行输入操作符获取行信息，逐行进行处理：

```
open my $string_fh, '<', \$multiline_string;  
while( <$string_fh> ) {  
    ... do something ...  
}
```

如果后续数据从另一个源进入，我们就需要一个能够读取的文件句柄。

8.4 文件句柄集合

与其他引用类似，因为文件句柄引用也是标量，所以可以对它们按照处理标量的方式

进行操作。特别地，我们能够将文件句柄引用作为数组元素或者散列值存储。我们可以同时打开几个文件句柄，然后决定使用哪一个。

例如，我们可以仔细检查如下来自第 6 章的数据，然后根据源主机名称（第一列）将数据分割到文件中：

```
professor.hut gilligan.crew.hut 1250
professor.hut lovey.howell.hut 910
thurston.howell.hut lovey.howell.hut 1250
professor.hut lovey.howell.hut 450
ginger.girl.hut professor.hut 1218
ginger.girl.hut maryann.girl.hut 199
```

如下所示，我们能够读取一行数据，基于第一列的源主机名称打开一个文件句柄，将文件句柄存入散列中，然后将该行的其他内容写入文件句柄中：

```
use v5.10; # for state

while( <> ) {
    state $fhs;

    my( $source, $destination, $bytes ) = split;

    unless( $fhs->{$source} ) {
        open my $fh, '>>', $source or die '...';
        $fhs->{$source} = $fh;
    }

    say { $fhs->{$source} } "$destination $bytes";
}
```

我们使用 state 关键字声明 \$fh 变量，因此该变量对于 while 循环而言是私有的，但会在多次迭代过程中保留值的内容不变。每次需要创建一个新文件句柄时，我们就将该文件句柄作为值添加到散列引用中。当 while 循环完成时，\$fhs 文件句柄超出作用域，就会自动关闭文件句柄。

如果你正在使用一个老版本的 Perl，你可以通过将 \$fhs 文件句柄移出 while 循环，但需要用裸块控制它的作用域：

```
use v5.10; # for state

{ # bare block to scope $fhs
    my $fhs;

    while( <> ) {
        ...
    }
}
```

8.5 IO::Handle 模块和其他相应的模块

在幕后，Perl 实际上使用 IO::Handle 模块实现文件句柄操作，因此，文件句柄标量实际

上是 IO::Handle 模块的对象。所以，如下所示，我们能够通过使用该模块的方法替换写入操作：

```
use IO::Handle;  
  
open my $fh, '>', $filename or die "...";  
$fh->print('Coconut headphones');  
$fh->close;
```



注意

你曾经考虑过 print 语句中为什么在文件句柄之后的部分没有逗号吗？这实际上是间接对象标记(之前没有提及这一点，除非在你阅读这个“注意”之前，你已经浏览整本书，与我们在前言中要求的一样！)。

自 Perl v5.14 之后，不必显式加载 IO::Handle 模块，但是对于之前的版本，需要自己手动完成。

IO::Handle 包对于输入、输出操作而言是基础类，因此它能够处理很多事情，而不仅仅是文件。大多数时候，我们想要使用一些方便快捷的模块，在这些模块上构建，而不是直接使用它们。我们还没有介绍关于面向对象编程的知识（这将在第 13 章中介绍），但是在这种情况下，你仅需要遵循模块文档中的示例。

这些模块做我们已经能够使用 Perl 内置的 open 函数完成（取决于我们使用的 Perl 版本）的一些工作，但是当我们想要尽可能晚地决定使用哪个模块处理输入、输出问题时，这些事情就变得很方便：不使用内置 open 函数，而使用模块接口。我们改变模块名称来切换该行为。因为我们使用一个模块接口建立代码，所以切换模块就不必做太多工作。

8.5.1 IO::File 模块

IO::File 模块是 IO::Handle 模块用于操作文件的子集。该模块来自于 Perl 标准发行版，因此你已经拥有它。我们有很多种创建 IO::File 对象的方法。

我们能够使用构造函数的单一参数形式创建文件句柄的引用。如下所示，我们通过查看文件句柄引用变量中定义的值，来检查操作的结果。

```
use IO::File;  
  
my $fh = IO::File->new( '> castaways.log' )  
or die "Could not create filehandle: $!";
```

因为我们不喜欢将 open 模式和文件名组合起来（与常规 open 函数操作的原因一致），我们将使用另外一种调用惯例。如下所示，第二个可选参数是文件句柄模式：

```
my $read_fh = IO::File->new( 'castaways.log', 'r' );  
  
my $write_fh = IO::File->new( 'castaways.log', 'w' );
```



注意

这些是 ANSI C 风格的 fopen 命令模式字符串。我们也能够通过内置 open 命令使用这些。确实如此，IO::File 模块在幕后使用内置的 open 命令。

使用位掩码作为模式允许更细粒度的控制。IO::File 模块提供如下常量：

```
my $append_fh = IO::File->new( 'castaways.log', O_WRONLY|O_APPEND );
```

除了打开命名文件之外，我们可能还想要打开匿名的临时文件。如下所示，在支持这类特性的操作系统上，创建一个新对象，以获取读/写文件句柄：

```
my $temp_fh = IO::File->new_tmpfile;
```

和之前一样，当标量变量超出作用域时，Perl 解释器自动关闭这些文件，但是如果这还不够，如下所示，我们能够通过显式调用 close 语句或者 undef 命令操作文件句柄完成：

```
$temp_fh->close or die "Could not close file: $!";
```

```
undef $append_fh;
```

如果将 undef 作为文件名，Perl v5.6 及后续版本能够打开一个匿名、临时的文件。如下所示，我们可能想要同时对该文件进行读取和写入操作。显然，拥有我们后续不能找到的文件就变得毫无意义：

```
open my $fh, '>', undef  
or die "Could not open temp file: $!";
```

8.5.2 IO::Scalar 模块

如果我们使用 Perl 的某个古老版本，可能该版本不能创建标量引用形式的文件句柄，此时，我们就可以使用 IO::Scalar 模块。该模块在幕后使用绑定（tie）的威力，提供一个附加在标量上的文件句柄引用。该模块并不是 Perl 标准发行版的模块，因此必须自行安装它：

```
use IO::Scalar;  
  
my $string_log = '';  
my $scalar_fh = IO::Scalar->new( \$string_log );  
  
print $scalar_fh "The Howells' private beach club is closed\n";
```

现在，我们最终将日志消息写入一个标量变量\$string_log 而不是一个文件。可是如果我们想要读取日志文件，该怎么操作？如下所示，我们将做同样的事情。在本示例中，我们和之前一样创建\$scalar_fh，然后通过行输入操作符读取数据。在 while 循环中，我们将提取包含有 Gilligan 的日志消息（应当是日志中的大多数消息，它总是在日志中出现）：

```
use IO::Scalar;
```

```

my $string_log = '';
my $scalar_fh = IO::Scalar->new( \$string_log );

while( <$scalar_fh> ) {
    next unless /Gilligan/;
    print;
}

```

8.5.3 IO::Tee 模块

如果我们想要同时发送输出到多个地方，该如何操作？如果我们想要同时将信息发送至一个文件并同时将信息保存为一个字符串，该如何操作？使用我们已知的方法，我们将不得不按照如下方式操作：

```

open my $log_fh, '>>', 'castaways.log'
or die "Could not open castaways.log";
open my $scalar_fh, '>>', \$my $string;

my $log_message = "The Minnow is taking on water!\n"
print $log_fh    $log_message;
print $scalar_fh $log_message;

```

我们可以将代码缩短，这样就可以只有一条 print 语句。我们使用 foreach 循环遍历该文件句柄引用列表，依次使用 \$fh 别名，然后输出到每一个文件句柄之中：

```

foreach my $fh ( $log_fh, $scalar_fh ) {
    print $fh $log_message;
}

```

这样操作的工作量还是有点大。在 foreach 循环中，我们必须决定导入哪个文件句柄。如果我们想要定义一组能够同时响应的文件句柄应当如何实现？这时就可以使用 IO::Tee 模块帮我们实现。想象一下，这就与船舱中输出管道的 T 型汇流连接器一致；当水流到达 T 型汇流连接器时，它就同时流向两个不同的方向。当输出到达 IO::Tee 模块时，它就能够同时流向两个（甚至多个）不同的管道。这就是 IO::Tee 模块的多路通道输出。在以下示例中，castaways 的日志消息同时流向日志文件和标量变量：

```

use IO::Tee;

my $tee_fh = IO::Tee->new( $log_fh, $scalar_fh );

print $tee_fh "The radio works in the middle of the ocean!\n";

```

但这还不是全部。如果 IO::Tee 模块的第一个参数是输入文件句柄（随后的参数必须是输出文件句柄），我们可以使用同样的 T 型文件句柄从输入读取，并且写入输出。源通道和目标通道是不同的，但是我们视它们为单个文件句柄。IO::Tee 模块在这种情况下做了一些特别的操作：当它从输入句柄读取一行数据时，它就立即将该数据写入输出文件句柄：

```

use IO::Tee;

my $tee_fh = IO::Tee->new( $read_fh, $log_fh, $scalar_fh );

# reads from $read_fh, prints to $log_fh and $scalar_fh
my $message = <$tee_fh>;

```

\$read_fh 也不必连接到文件，它也可能连接到一个套接字、标量变量或者外部命令的输出。我们能够使用下面将要介绍的 IO::Pipe 模块或者其他能够创建的外部命令创建可读取的文件句柄。

8.5.4 IO::Pipe 模块

有时候数据不是来自于文件或者套接字，而是来自于外部命令。我们能够使用一个管道化的 open 语句读取命令的输出。如下所示，与 Shell 中的管道类似，符号 “|” 之后的 \$command 将会记录来自于命令的输出，然后通过管道传入程序。

```
open my $pipe, '-|', $command  
or die "Could not open filehandle: $!";  
  
while( <$pipe> ) {  
    print "Read: $_";  
}
```

使用模块自动处理这些细节可能有点简单。IO::Pipe 模块是 IO::Handle 模块的前端，只要提供一条命令，它就自动处理 fork 命令和 exec 命令，然后留下一个文件句柄，我们能够从该文件句柄读取命令的输出：

```
use IO::Pipe;  
  
my $pipe = IO::Pipe->new;  
  
$pipe->reader( "$^X -V" ); # $^X is the current perl executable  
  
while( <$pipe> ) {  
    print "Read: $_";  
}
```

同样，也能够将数据流写入命令。如下所示，在 \$command 之前放置管道，以显示正在流入程序的数据：

```
open my $pipe, "| $command"  
or die "Could not open filehandle: $!";  
  
foreach ( 1 .. 10 ) {  
    print $pipe "I can count to $_\n";  
}
```

也可以使用 IO::Pipe 模块处理这些内容：

```
use IO::Pipe;  
  
my $pipe = IO::Pipe->new;  
  
$pipe->writer( $command );  
  
foreach ( 1 .. 10 ) {  
    print $pipe "I can count to $_\n";  
}
```

8.5.5 IO::Null 模块和 IO::Interactive 模块

有时候我们不想发送输出到任何地方，但是我们不得不将它们发送到某些地方。在这种情况下，我们能够使用 IO::Null 模块创建一个文件句柄，该文件句柄将丢弃我们提供的所有内容。如下所示，它的外表和行为像是一个文件句柄，但是什么都不做：

```
use IO::Null;

my $null_fh = IO::Null->new;

some_printing_thing( $null_fh, @args );
```

其他时候，我们想要在某些情况下输出而在其他情况下不输出。如果我们进入终端，然后运行程序，就将看到很多输出。然而，如果我们通过 cron 安排我们的工作，只要工作能够正常进行，我们就不在意输出的内容。IO::Interactive 模块有足够的智能分辨两者之间的差别：

```
use IO::Interactive;

print { interactive } 'Bamboo car frame';
```

interactive 子例程返回一个文件句柄。因为该子例程的调用不是一个简单的标量变量，所以用大括号把它包括起来，告诉 Perl 解释器这是一个文件句柄。

既然我们已经知道那个“什么都不做”的文件句柄，我们就能够替换每个人倾向于编写的丑陋代码。有时我们想要输出，有时我们不想要，因此，如下所示，在某些情况下，我们可以使用一个语句修改器在某些情况下关闭语句：

```
print STDOUT "Hey, the radio's not working!" if $Debug;
```

相反，无论在我们想要的何种情况下，都能够为 \$debug_fh 赋不同的值，然后，如下所示，我们就可以在每条 print 语句的末尾停止使用丑陋的 if \$Debug 语句：

```
use IO::Null;

my $debug_fh = $Debug ? *STDOUT : IO::Null->new;

$debug_fh->print( "Hey, the radio's not working!" );
```

尽管 IO::Null 模块非常好用，但它背后的魔力可能通过间接对象符号（例如：print \$debug_fh），提供一个关于“print() on unopened filehandle GLOB”的警告。我们将不会得到箭头形式的警告。

8.6 目录句柄

与创建文件句柄引用的方式一致，我们同样也能够按照如下方式创建目录句柄引用：

```
opendir my $dh, '.' or die "Could not open directory: $!";  
  
foreach my $file ( readdir( $dh ) ) {  
    print "Skipper, I found $file!\n";  
}
```

目录句柄引用同样遵循之前展示的规则。仅当标量变量没有值时这才可行，并且当变量超出作用域或者为它赋新的值时，句柄自动关闭。

目录句柄引用

我们也能够将面向对象接口用于目录处理目录句柄。自 v5.6 起，IO::Dir 模块就成为 Perl 标准发行版的一部分。如下所示，该模块没有添加有趣的新特性，但是能够对 perl 的内置函数打包^{注3}：

```
use IO::Dir;  
  
my $dir_fh = IO::Dir->new( '.' )  
or die "Could not open dirhandle! $!\n";  
  
while( defined( my $file = $dir_fh->read ) ) {  
    print "Skipper, I found $file!\n";  
}
```

如果我们决定再次遍历上面示例中的列表（可能在本程序的后续部分），就不必重新创建一个新的目录句柄。可以按照如下方法倒回文件句柄重新开始：

```
while( defined( my $file = $dir_fh->read ) ) {  
    print "I found $file!\n";  
}  
  
# time passes  
$dir_fh->rewind;  
  
while( defined( my $file = $dir_fh->read ) ) {  
    print "I can still find $file!\n";  
}
```

8.7 练习

可以在附录中的“第 8 章答案”部分找到这些练习的答案。

1. [20 分钟] 编写一个程序，输出当天的日期和当天是周几，但是允许用户选择将输出发送至文件、标量或者同时发送至两者。无论用户选择哪一个，仅使用单条 print 语句发送输出。如果用户选择将输出发送至一个标量，就在本程序的末尾将标量的值输出到标准输出。
2. [30 分钟] Professor 必须读取一个如下所示的日志文件。你可以通过 <http://www.intermediateperl.com>/网页的 Download 部分获取该示例数据文件：

注 3：对于每个 IO::Dir 模块中方法的名称，添加“dir”，然后在 perldoc 文档中获取更多信息。

```
Gilligan: 1 coconut
Skipper: 3 coconuts
Gilligan: 1 banana
Ginger: 2 papayas
Professor: 3 coconuts
MaryAnn: 2 papayas
...
```

他想要写入一系列文件，分别叫做 gilligan.info、maryann.info 等。每个文件将包含以该名称开始的所有行（名称总是由名称尾部的冒号限定）。在最后，gilligan.info 文件将以如下行作为文件的开始：

```
Gilligan: 1 coconut
Gilligan: 1 banana
```

现在，日志文件非常大，并且以 coconut 电池供电的计算机并不是非常快，因此他想要一次性完成输入文件的处理，然后并行写入所有的输出文件。他该怎么做？

提示：使用遇难者的名称作为散列的键，值为每个输出文件的 IO::File 对象。如果它们不存在，就创建这些文件；如果它们存在，就覆盖它们。

3. [15 分钟] 编写一个程序，从命令行获取多个目录名称，然后输出这些目录中的内容。使用一个函数，该函数使用你用 opendir 生成的目录句柄引用。

第9章

正则表达式引用

从 Perl v5.5 开始，我们能够编译正则表达式并且对正则表达式取引用，而不必使用匹配或者替换操作符。我们能够在实际使用正则表达式之前，完成全部的处理和准备工作。因为正则表达式引用仅仅是一个与其他引用类似的标量，所以我们在数组或者散列中存储它们，作为参数传递它们，将它们插入字符串，或者在很多其他能够适用标量的场合下使用。

9.1 正则表达式引用之前

大多数人看到的正则表达式是如下匹配或替换操作符的一部分：

```
m/\bcoco.*/
s/\bcoconut\b/coconet/
split /coconut/, $string
```

然而，由操作符分割出来的模式和操作符只是将模式应用于字符串。

我们可能已经有了如下一个关于这类操作的暗示，因为我们能够将正则表达式插入其中一个操作符中：

```
my $pattern = 'coco.*';
if( m/$pattern/ ) {
    ...
}
```

在这种情况下，最初\$pattern 仅仅是一个与其他字符串类似的字符串，并且它不知道我们如何使用它。在匹配操作符插入变量之后，它必须编译生成的正则表达式。就像在一个文字模式之中，匹配操作符不能预先判断该模式是否有效。如果提供一个无效的模式，就可能导致匹配操作符生成一个致命的运行时错误。如下所示，如果想要处理这类可能发生的错误，我们就可以使用 eval 语句在匹配的时候捕获它们：

```

print 'Enter a pattern: ';
chomp( my $pattern = <STDIN> );

print "Enter some lines:\n";
while( <STDIN> ) {
    if( eval { m/$pattern/ } ) {
        print "Match: $_";
    }

    if( $@ ) {
        die "There was a regex problem: $@\n";
    }
}

```

当使用一个包含不匹配圆括号的无效模式时，错误消息显示模式在何处出错：

```

% perl match.pl
Enter a pattern: Gilligan)
Enter some lines:
Gilligan & Skipper
There was a regex problem: Unmatched ) in regex;
marked by <-- HERE in m/Fred) <-- HERE /
at test line 8, <STDIN> line 2.

```

我们可以做类似的事情：使用正则表达式作为子例程参数。如下所示，传递一个字符串作为参数，然后在子例程内部编译模式：

```

find_match( 'Gilligan' );

sub find_match {
    my( $pattern ) = @_;
    if( eval { m/$pattern/ } ) {
        ...
    }
}

```

正如上面的示例所示，由于我们获取错误太晚，以至于无法实施任何应对措施。我们在尝试使用一个模式之前，能够编写一个子例程测试该模式。按照如下方式对一个空字符串尝试进行匹配操作：

```

sub is_valid_pattern {
    my( $pattern ) = @_;
    local( $@ );

    eval { '' =~ $pattern };
    return defined $@ ? 0 : 1;
}

```

此时，我们能够在实际使用模式之前，使用上面的子例程测试该模式。这样做并不理想，原因有很多，包括匹配变量的副作用和正则表达式能够运行任何代码的可能性。我们不想仅仅为了验证一个模式，就去触发这两个的任何一个。此外，这样做显得非常杂乱而且令人厌烦。如果我们能够在不实际匹配的情况下预编译正则表达式，实现起来就会更容易。

9.2 预编译模式

Perl v5.5 引入一个新的引用机制：qr//操作符。如下所示，这与其他常见的引用机制类似，但是它向我们提供了一个编译完成的正则表达式的引用：

```
my $regex = qr/Gilligan|Skipper/;
```



注意

在 perlop 文档中查看“引号和类引号操作符”。

此时，我们还没有将模式应用于其他地方；但我们后续将做一些这类的尝试。如下所示，我们能够通过输出语句检查模式：

```
print $regex
```

Perl stringifies the pattern:

```
(?^:Gilligan|Skipper)
```



注意

不同版本的 Perl 可能使用不同的方式字符串化引用，因此我们将不会依赖某个特定的字符串形式。上面的形式来自于 Perl v5.14。

后续我们将展示在字符串化版本中关于这些内容的更多细节。

然而，qr//操作符并没有解决无效模式问题，因此，如下所示，一旦需要插入一个字符串到模式中，就将使用 eval 语句：

```
my $regex = eval { qr/$pattern/ };
```

与其他常见的引用操作符类似，我们能够选择不同的分隔符。实际上，如下所示，我们通常都这样做，以避免在模式中使用的文字字符问题：

```
my $regex = qr(/usr/local/bin/\w+);
```

和匹配操作符一样，斜线分隔符是一个特别的分隔符。如果使用单引号作为分隔符，Perl 解释器就将模式视为单引号字符串。也就是说，Perl 解释器将不会做任何双引号插入操作：

```
my $regex = qr'$var'; # four characters, not two
```

尽管单引号分隔符禁用了插入操作，但正则表达式元字符还是特殊字符。例如，\$ 符号仍然表示字符串结尾的“锚位”。如下所示，如果我们想要使用\$字符，就需要对它转义；因此，Perl 解释器将不认为这是字符串结尾的“锚位”：

```
my $regex = qr'\$\d+\.\d+';
```

正则表达式的其余部分仍然和正常的一样，\d 仍然表示数字字符类，\+ 仍然表示限定符，等等。

9.2.1 正则表达式选项

如下所示，当使用匹配或者替换操作符时，我们将所有的标志（flag）放置于匹配操作的末尾，在最后一个分隔符之后：

```
m/Gilligan$/migc  
s/(\d+)/ $1 + 1 /eg
```

因为我们将所有选项放置于操作符的末尾，所以我们将不会区分影响模式的标志和影响操作符的标志^{注1}。

当使用 qr//操作符时，仅能够添加影响模式的标志（/x、/I、/s、/m、/p、/o、/a、/l、/d、或者 /u）。这有两种方法实现。第一种方法，是在 qr//操作符末尾添加标志，如下所示，这与我们之前使用操作符的方法一致：

```
qr/Gilligan$/mi;
```

我们也能够直接在模式本身中添加标记，这不是 qr//操作符的特性，也就是说，可以在任何模式中使用这个特性。如下所示，特殊序列（?flags:pattern）允许在模式本身中指定修饰符：

```
qr/(?mi:Gilligan)/;
```

修饰符仅应用于圆括号中封闭的模式部分。如下所示，假如我们仅想要模式的某个部分区分大小写，我们就能够通过问号（?）把这些模式分组并应用我们想用的模式标记：

```
qr/abc(?i:Gilligan)def/;
```

我们也能够通过在标记之前添加一个“-”，从一部分模式中移除修饰符：

```
qr/abc(?-i:Gilligan)def/i;
```

我们甚至能够同时添加和移除修饰符。如下所示，首先，我们指定我们想要添加的部分；然后，移除我们不想要的部分：

```
qr/abc(?x-i:G i l l i g a n)def/i;
```

9.2.2 应用正则表达式引用

当我们想要应用正则表达式引用时，有很多选项。如下所示，我们能够将正则表达式引用插入一个匹配或者替换操作符中：

```
my $regex = qr/Gilligan/;  
$string =~ m/$regex/;  
$string =~ s/$regex/Skipper/;
```

如下所示，我们也能够直接绑定而不必使用显式匹配操作符。绑定操作符将识别正则表达式引用然后运行匹配操作：

```
$string =~ $regex;
```

智能匹配也一样：

```
$string ~~ $regex;
```

注 1：“Know the difference between regex and match operator flags” <http://www.effectiveperlprogramming.com/blog/174>。

9.3 作为标量的正则表达式

既然我们能够在标量中保存预编译的正则表达式，我们就能够以使用其他标量的同样方式使用它们：在数组和散列中存储它们，作为参数把它们传递至子例程。

假如你想要同时匹配多个模式。如下所示，可以在数组中存储这些模式，然后逐个仔细检查，直到找到一个匹配项：

```
use v5.10.1;

my @patterns = (
    qr/(?:Willie )?Gilligan/,
    qr/Mary Ann/,
    qr/Ginger/,
    qr/(?:The )?Professor/,
    qr/Skipper/,
    qr/Mrs?. Howell/,
);
my $name = 'Ginger';

foreach my $pattern ( @patterns ) {
    if( $name ~~ $pattern ) {
        say "Match!";
        last;
    }
}
```

这样的代码并不好。我们在每个 `foreach` 循环语句中都有一个 `if` 语句。通过智能匹配，我们能够匹配数组中的所有模式。如下所示，智能匹配将遍历数组中的每个元素：

```
use v5.10.1;

my @patterns = (
    qr/(?:Willie )?Gilligan|,
    qr/Mary Ann|,
    qr/Ginger|,
    qr/(?:The )?Professor|,
    qr/Skipper|,
    qr/Mrs?. Howell|,
);
my $name = 'Ginger';
say "Match!" if $name ~~ @patterns;
```

这样做依旧不够完美。谁能知道匹配了哪个模式？如下所示，我们现在能够将模式放入散列，这样我们就可以给它们提供标签：

```
use v5.10.1;

my %patterns = (
    Gilligan => qr/(?:Willie )?Gilligan|,
    'Mary Ann' => qr/Mary Ann|,
    Ginger => qr/Ginger|,
    Professor => qr/(?:The )?Professor|,
```

```

Skipper    => qr/Skipper/,
'A Howell' => qr/Mrs?. Howell/,
);

my $name = 'Ginger';

my( $match ) = grep { $name =~ $patterns{$_} } keys %patterns;

```

say "Matched \$match" if \$match;

如下所示，我们将对以上代码做进一步的提炼：哪怕 grep 语句已经找到了一个匹配项，它还将继续查询，因此我们将使用 List::Util 模块中的 first 函数：

```

use v5.10.1;
use List::Util qw(first);

my %patterns = (
    Gilligan    => qr/(?:Willie )?Gilligan/,
    'Mary Ann'   => qr/Mary Ann/,
    Ginger      => qr/Ginger/,
    Professor   => qr/(?:The )?Professor/,
    Skipper     => qr/Skipper/,
    'A Howell'  => qr/Mrs?. Howell/,
);

my $name = 'Ginger';

my( $match ) = first { $name =~ $patterns{$_} } keys %patterns;

```

say "Matched \$match" if \$match;

这样我们就得到了更佳的结果。如果需要匹配多个名称，但是我们想要得到最右边的匹配，该如何实现？正如我们已经在 *Learning Perl* 一书中介绍过的，Perl 的正则表达式引擎会查找最右边最长的匹配。如下所示，我们将创建一个叫做 rightmost 的子例程，向该子例程提供一个字符串和一个模式列表，返回最右边匹配的起始位置：

```

my $position = rightmost(
    'Mary Ann and Ginger',
    qr/Mary/, qr/Gin/,
);

```

下面的代码就是我们实现的方法。我们将\$rightmost 的值初始化为-1。如果我们获取的位置与返回值一致，我们就知道什么都不匹配。记住，第一个位置是 0，如果我们获取这个值，就知道模式匹配到了字符串的起始部分。

我们可以通过 each 函数遍历@patterns 数组。在 Perl v5.12 中，该操作将返回下一个元素的索引和值^{#2}。在 while 循环语句内部，基于匹配的结果选择值，使用条件操作符。如果匹配成功，就选择\$?[0]，否则选择-1。@?特殊变量能够记忆整个模式匹配的起始

注 2：不使用 each 语句完成这些仅仅使人感到略微的厌烦。遍历这些索引，然后自行提取值。然而，你将错过新特性的有趣之处。

位置并且能够捕获分组。`$?[]`中的数值是整个匹配操作的起始位置^{注3}。如果匹配的位置比之前所记忆的位置高一些（也就是说，更靠右边），我们将新的位置信息存入`$rightmost`之中，从而更新`$rightmost`的值：

```
use v5.12;

sub rightmost {
    my( $string, @patterns ) = @_;
    my $rightmost = -1;
    while( my( $i, $pattern ) = each @patterns ) {
        $position = $string =~ m/$pattern/ ? $-[0] : -1;
        $rightmost = $position if $position > $rightmost;
    }
    return $rightmost;
}
```

我们现在将所有这些放在一起。我们将使用来自于`%pattern`散列中已排序的键列表，以散列切片的方式获取多个模式：

```
use v5.12;

my %patterns = (
    Gilligan => qr/(?:Willie )?Gilligan/,
    'Mary Ann' => qr/Mary Ann/,
    Ginger => qr/Ginger/,
    Professor => qr/(?:The )?Professor/,
    Skipper => qr/Skipper/,
    'A Howell' => qr/Mrs?. Howell/,
);
my $position = rightmost(
    'There is Mrs. Howell, Ginger, and Gilligan',
    @patterns{ sort keys %patterns }
);
say "Rightmost match at position $position";
```

这还能够使用一些其他方法改进，但是我们将这部分有趣的内容留在本章末尾的练习中。你不必记住这里所发生事情的全部细节；仅仅需要知道你能够传递一个正则表达式引用作为子例程参数。毕竟，和所有的引用一样，这只是标量。

9.4 建立正则表达式

我们可以预编译模式，然后将它们插入匹配操作符。我们也能够将模式插入其他模式之中。如下所示，通过这种方法，我们能够通过更小、更可控的模式组成复杂的模式：

注 3：也还有@+操作符，该操作符表示结尾位置。

```

my $howells    = qr/Thurston|Mrs/;
my $tagalongs  = qr/Ginger|Mary Ann/;
my $passengers = qr/$howells|$tagalongs/;
my $crew        = qr/Gilligan|Skipper/;

my $everyone   = qr/$crew|$passengers/;

```

当在一个更大的模式中使用正则表达式引用时，任何正则表达式的元字符仍然是特殊字符。\$howells 和\$tagalongs 中的择一匹配在\$passengers 中仍然是择一匹配。

这允许我们将长的、复杂的模式分解成更小的、易于理解的数据块，我们可以按照任意想要的方式将这些数据块组合起来。不仅如此，我们还能够将它们以不同的方式组合起来。如下所示，如果我们想要匹配一组不同的人，就将不同的部分组合起来：

```
my $poor_people = qr/$tagalongs|$passengers|$crew/;
```

如下示例是一个更长的示例，尽管你自己从未曾经编写过这类正则表达式。RFC 1738 文档规定了 URL 的格式，我们能够将该规范变为正则表达式。该系列正则表达式引用几乎是 RFC 1738 直接翻译的结果：

```

my $alpha      = qr/[a-z]/;
my $digit      = qr/\d/;
my $alphadigit = qr/(?:$alpha|$digit)/;
my $safe        = qr/[\$_+-]/;
my $extra       = qr/[!*'\(\)\,]/;
my $national   = qr/{\}|\\^\\[\\`]/;
my $reserved   = qr|[/:@&=]|;
my $hex         = qr/(?:$digit|[A-F])/;
my $escape      = qr/%$hex$hex/;
my $unreserved = qr/$alpha|$digit|$safe|$extra/;
my $uchar      = qr/$unreserved|$escape/;
my $xchar      = qr/$unreserved|$reserved|$escape/;
my $ucharplus  = qr/(?:$uchar|[:?&=])*/;
my $digits     = qr/(?:$digit){1,}/;

my $hsegment   = $ucharplus;
my $xpath      = qr|$hsegment(?:/$hsegment)*|;
my $search     = $ucharplus;
my $scheme     = qr|(?:https?://)|;
my $port        = qr/$digits/;
my $password   = $ucharplus;
my $user        = $ucharplus;

my $toplevel   = qr/$alpha|$alpha(?:$alphadigit|-)*$alphadigit/;
my $domainlabel = qr/$alphadigit|$alphadigit
  (?:$alphadigit|-)*$alphadigit/x;
my $hostname   = qr/(?:$domainlabel\.)*$toplevel/;
my $hostnumber = qr/$digits\.$digits\.$digits\.$digits/;
my $host        = qr/$hostname|$hostnumber/;
my $hostport   = qr/$host(?::$port)?/;
my $login       = qr/(?:$user(?::$password)\@)?/;

my $urlpath    = qr/(?:($xchar)*)/;

```

因为 qr// 操作符在模式中自动放置实质上的非捕获圆括号，所以这些模式相互之间并不

会干扰。如下所示，最终，我们将所有这些放在一起表述 URL，使程序在文本中查找 URL：

```
# ... all of those other lines
use v5.10.1;

my $httpurl = qr|$scheme$hostport(?:/$hpath(?:\?[$search])?)?|;

while( <> ) {
    say if !$httpurl;
```

就像这样，我们能够将所有内容放到一起，但是还有更佳的实现方法。请继续阅读。

9.5 创建正则表达式的模块

因为我们能够创建预编译的正则表达式，所以许多人编写了很多构建模式的模块。我们可以直接使用这些模块中的模式而不必自己重复构建复杂的模式，这样就可以避免可能遗漏一些边际情况或者错误地指定部分模式。我们能够依赖这些众所周知的模块为我们构造模式。

9.5.1 使用常见的模式

Abigail, Perl 的一位正则表达式大师，将大部分复杂的模式放入一个模块中，这样就能够替代人们总是尝试自己编写这些模式（通常情况下变得更糟）。我们能够直接使用 Regexp::Common 模块提供的某个模式，而不是创建我们自己的模式。该模块导出一个叫做\$RE 的散列引用，它以我们所需要的正则表达式引用作为其值。如下所示，来自于上一节的长程序将缩减为更简单的版本：

```
use v5.10.1;
use Regexp::Common qw(URI);
while( <> ) {
    print if /$RE{URL}{HTTP}/;
}
```

散列为我们提供一个正则表达式引用。如下所示，与其他引用一样，我们能够将该引用字符串化：

```
use v5.10.1;
use Regexp::Common qw(zip);

say $RE{zip}{US};
```

我们获得的字符串有一点复杂：

```
(?:(?:(?:USA?)-){0,1}(?:(?:[0-9]{3})(?:[0-9]{2}))  
(?:(?:-){0,1}(?:[0-9]{2})(?:[0-9]{2})){0,1}))
```

我们还可以从该模块中获取很多其他种类的正则表达式。如果我们想要查找 IPv4 地址，例如 10.1.0.37，我们就能够使用该模块的 net 工具衍生出的一个模式：

```
use v5.10.1;
use Regexp::Common qw(net);

while( <> ) {
    say if /$RE{net}{IPv4}/;
}
```

在幕后，Regexp::Common 模块使用 tie，因此它能激活我们决定要使用的键的很多特殊魔力。如下所示，假如我们想要匹配数字，我们就可以从 Regexp::Common 模块中一个用于匹配数字的模式开始：

```
use v5.10.1;
use Regexp::Common qw(number);

while( <> ) {
    say if /$RE{num}{int}/;
}
```



注意

想要学习关于 tie 的更多信息，请查阅 *Mastering Perl* 一书或者查询 perltie 文档。

尽管上例适合用于查找十进制整数，但是我们还可以修改这个模式，以匹配十六进制整数：

```
use v5.10.1;
use Regexp::Common qw(number);

while( <> ) {
    say if /$RE{num}{int}{ -base => 16 }/;
}
```

该程序输出包含数字的行，但是如果我只想要数字，而不是整行，我们就能够添加一个{-keep}键，这样模式将只捕获匹配到的内容：

```
use v5.10.1;
use Regexp::Common qw(number);

while( <> ) {
    say $1 if /$RE{num}{int}{ -base => 16 }{-keep}/;
}
```

因为\$RE 是一个“神奇的”散列，所以我们甚至不必按任何特定的序列放置键。如下所示，我们可以在我们想要的位置放置以“-”开始的特殊键：

```
use v5.10.1;
use Regexp::Common qw(number);

while( <DATA> ) {
    say $1 if /$RE{ -base => 16 }{num}{ -keep }{int}/;
}
```

9.5.2 组装正则表达式

Regexp::Common 模块给我们提供预定义模式，但也有其他模块能够帮助我们建立正则

表达式。例如，`Regexp::Assemble` 模块帮助我们建立高效的择一匹配。考虑到我们拥有在大多数分支中都有相同前缀的择一匹配，在这类情况下，假如我们想要匹配 Mr. Howell、Mrs. Howell 或者 Mary Ann。我们能够按照如下方式生成一个简单的择一匹配：

```
my $passenger = qr/(?:Mr. Howell|Mrs. Howell|Mary Ann)/;
```

所有择一匹配都以字母 M 开头，但这里的简单方法每次检查字母 M。它们中的两个使用字母 r 作为第二个字母。

这样做依然并不非常高效，因为匹配可能必须多次查看同样的字符，以确保与它上一次匹配的内容一致。如下所示，使用 `Regexp::Assemble` 模块，我们将不同的部分放到一起：

```
use v5.10.1;
use Regexp::Assemble;

my $ra = Regexp::Assemble->new;
for ( 'Mr. Howell', 'Mrs. Howell', 'Mary Ann' ) {
    $ra->add( "\Q$_" );
}
```

```
say $ra->re;
```

该模块找到一个将这些内容聚合在一起作为择一选择的好方法，这样我们就不必重复检查任意字符：

```
(?^:M(?:rs?\.\. Howell|ary Ann))
```

如果你在使用 v5.10 或者后续版本，Perl 已经自动完成这些工作。

9.6 练习

可以在附录中的“第 9 章答案”部分找到这些练习的答案。

- [30 分钟] 使 `rightmost` 程序运行（如果你不想自己输入程序的整个内容，可以在 <http://www.intermediateperl.com/> 网站的 Download 部分获取该程序）。一旦你的示例工作，修改 `rightmost` 程序，使用散列引用的模式，然后返回最右边匹配的键。而不是像如下方式一样调用它：

```
my $position = rightmost(
    'There is Mrs. Howell, Ginger, and Gilligan',
    \%patterns
);
```

或者像下面的方式一样调用它：

```
my $key = rightmost(
    'There is Mrs. Howell, Ginger, and Gilligan',
    \%patterns
);
```

- [45 分钟] 编写一个程序，从文件中读入一个模式列表。预编译模式，然后将它们存入数组。例如，你的模式文件可能如下所示：

```
cocoa?n[ue]t  
Mary[-\s]+Anne?  
(The\s+)?(Skipper|Professor)
```

提示用户对于输入的行，输出匹配的行号和每行的文本。\$.变量在此处非常有用。

3. 使用 `Regexp::Assemble` 模块修改练习 2 中的程序，因此你将拥有一个模式而不是一个模式数组。

第 10 章

实用的引用技巧

本章将介绍优化、排序和处理递归定义的数据。

10.1 更佳的输出

Perl 的内置 sort 操作符在默认情况下对文本字符串按照字符顺序进行排序^{注1}。如下所示，这在我们对于字符串排序时没有任何问题：

```
my @sorted = sort qw(Gilligan Skipper Professor Ginger Mary Ann);  
但是，当我们想要对数字排序时，一切就变得一团糟：
```

```
my @wrongly_sorted = sort 1, 2, 4, 8, 16, 32;  
结果列表是 1, 16, 2, 32, 4, 8。为什么不能按照正确的顺序进行排序呢？这是因为  
sort 操作符将这些数字按照字符串的方式处理，并且以字符串的顺序进行排序。任何以  
3 开头的字符串都放置于以 4 开头的字符串之前。
```

如果我们不想使用默认的排序顺序，就不必编写整个排序算法，这是因为 Perl 已经拥有了一个非常好的方法处理这类问题。但是无论使用哪种排序算法，在某些时刻，我们都必须查看 A 项和 B 项，并且决定哪一个靠前。这就是我们需要编写的部分：编写处理两个数据项的代码。Perl 将处理余下的部分。

默认情况下，当 Perl 对于数据项进行排序时，它采用比较字符串的方式。我们能够使用 sort 代码块指定一个新的比较方式，将该代码块放置于 sort 关键字和需要排序的列表之间^{注2}。在该 sort 代码块中，\$a 和 \$b 代表将要进行比较的两个元素。如果对数字进

注 1：我们将以“字符串顺序”作为它们代码中数字的升序，忽略标准化、大小写和其他一切重要的内容。一些人称之为“ASCIIbetical”排序。然而，现代 Perl 并不适用 ASCII 码；而是根据当前语言环境和字符集，使用默认排序顺序。我们可以参考 perllocale 查询更多信息。

注 2：我们也能够使用一个为每项比较调用 sort 内置函数的命名子例程。

行排序，那么 \$a 和 \$b 将是来自于列表中的两个数字。

sort 语句块必须返回一个编码值来指明排序的顺序。如果在期望的排序顺序中 \$a 在 \$b 之前，它就将返回 -1；如果 \$b 在 \$a 之前，它就返回 +1；如果顺序无关紧要，它就将返回 0。顺序可能是无关紧要的，例如，在一个不区分大小写的排序中，比较 “FRED” 和 “Fred”，或者在数字排序中比较 42 和 42^{注3}。

例如，要以适当的顺序对这些数字进行排序，我们能够按照如下方式，使用 sort 语句块比较 \$a 和 \$b：

```
my @numerically_sorted = sort {
    if ($a < $b) { -1 }
    elsif ($a > $b) { +1 }
    else { 0 }
} 1, 2, 4, 8, 16, 32;
```

现在，我们拥有一个适当的数字比较方法，因此我们就有了适当的数字排序方法。然而，这样做还是非常繁琐，如下所示，可以使用飞船操作符 “<=>” 代替：

```
my @numerically_sorted = sort { $a <=> $b } 1, 2, 4, 8, 16, 32;
```

根据之前安排的规则，飞船操作符返回 -1、0 和 +1。如下所示，在 Perl 中实现降序排序是很简单的^{注4}：

```
my @numerically_descending =
    reverse sort { $a <=> $b } 1, 2, 4, 8, 16, 32;
```

但是完成一件事可以有许多种办法。飞船操作符是“近视”的；它不能看出参数中的哪一个来自于 \$a，哪一个来自于 \$b；它只能看出哪一个值放置于左边，哪一个值放置于右边。如下所示，如果交换 \$a 和 \$b 的位置，飞船操作符将以相反的顺序进行排序：

```
my @numerically_descending =
    sort { $b <=> $a } 1, 2, 4, 8, 16, 32;
```

之前示例中 sort 表达式的每次比较结果都返回 -1，但是上面的表达式返回 +1，反之亦然。因此，该排序是逆序排序，而且此时不需要 reverse 操作符。这样的表达式也很容易记忆，因为如果 \$a 在 \$b 的左边，我们就将获得较小的值。如果 \$b 在表达式的最左边，我们将获得较大的值。

哪种方法更好呢？何时应当对 sort 语句使用 reverse 操作符排序？何时应当切换 \$a 和 \$b 的位置？这通常在性能上没有什么差异，因此，如果我们为了语句表述更清晰而优化表述的方式，就使用 reverse 操作符。然而，对于更复杂的比较，单个 reverse 语句可能不能满足任务的需要。

注 3：实际上，我们能够使用任意负数或者正数替换 -1 和 +1。最新的 Perl 版本包含一个稳定的默认排序引擎，因此，由 sort 语句块返回的零值将会影响 \$a 和 \$b 的相对次序，最终影响它们在初始列表中的顺序。旧版本的 Perl 不能保证这类稳定性，未来的 Perl 版本可能不用一个稳定的 sort，因此不要依赖于这些：使用 sort 的“稳定性”声明确保稳定性的或者报告失败。

注 4：作为 Perl v5.8.6，Perl 解释器识别 reverse sort 并且实现的时候，没有产生临时的中间列表。

与飞船操作符类似，我们能够使用 cmp 操作符指定一个字符串排序，因为这是 sort 语句的默认比较方法，所以很少单独使用。cmp 操作符通常在更复杂的比较中使用，我们将会在后续部分详细介绍。

10.2 用索引排序

在第 3 章中，我们通过 grep 语句和 map 语句以及索引，以同样的方式解决了一些问题，我们也将通过 sort 语句以及索引以获取一些更有趣的结果。如下所示，例如，对之前的名称列表进行排序：

```
my @sorted = sort qw(Gilligan Skipper Professor Ginger Mary Ann);
print "@sorted\n";
```

结果必定为：

```
Gilligan Ginger Mary Ann Professor Skipper
```

但是，如果我们要查看初始列表，然后决定初始列表中的哪个元素现在出现在排序列表中的第 1 个、第 2 个、第 3 个等，该如何实现呢？例如，Ginger 是排序列表中的第 2 个元素，并且是初始列表中的第四个元素。我们该如何确定最终排序列表的第 2 个元素就是初始列表的第 4 个元素呢？

这样，我们能够使用一些间接方法。我们对名字在数组中位置的索引排序，而不是对实际名字本身排序。

```
#          0      1      2      3      4
my @input = qw(Gilligan Skipper Professor Ginger Mary Ann);
my @sorted_positions =
    sort { $input[$a] cmp $input[$b] } 0 .. $#input;
print "@sorted_positions\n";
```

此时此刻，\$a 和 \$b 并非列表中的元素，而是索引。因此，我们不是在对 \$a 和 \$b 进行比较，而是使用 cmp 操作符以字符串的形式比较 \$input[\$a] 和 \$input[\$b]。排序的结果是以 @input 数组中相应元素所定义的顺序得到的索引。输出为 0 3 4 2 1，这意味着排序列表中的第 1 个元素是初始列表中的第 0 个元素：Gilligan。排序列表中的第 2 个元素是初始列表中的第 3 个元素：Ginger，以此类推。现在，我们能够排列信息，而不是仅仅移动名字。

事实上，有逆序排列。我们仍然不知道初始列表中一个给定的名称在输出列表中所占据的位置是哪一个。当然，通过使用更多的魔力，也可以按照如下方式实现：

```
#          0      1      2      3      4
my @input = qw(Gilligan Skipper Professor Ginger Mary Ann);
my @sorted_positions =
    sort { $input[$a] cmp $input[$b] } 0 .. $#input;
my @ranks;
@ranks[@sorted_positions] = (0..$#sorted_positions);
print "@ranks\n";
```

这段代码的输出为：0 4 3 1 2。这就是说，在输出列表中 Gilligan 的位置为 0，Skipper 的位置为 4，Professor 的位置为 3，以此类推。此时的位置信息基于 0，因此可以加 1，这样听

起来更像通常“人类”使用的序数值。其中的一个技巧是使用`1..@sorted_positions`，而不是使用`0..$#sorted_positions`。因此，如下为转储全部内容的方法：

```
#          0      1      2      3      4
my @input = qw(Gilligan Skipper Professor Ginger Mary Ann);
my @sorted_positions = sort { $input[$a] cmp $input[$b] } 0..$#input;
my @ranks;
@ranks[@sorted_positions] = (1 .. @sorted_positions);
foreach (0..$#ranks) {
    print "$input[$_] sorts into position $ranks[$_]\n";
}
```

得到的结果为：

```
Gilligan sorts into position 1
Skipper sorts into position 5
Professor sorts into position 4
Ginger sorts into position 2
Mary Ann sorts into position 3
```

如果我们需要从不同的角度看数据，这种通用方法是非常便捷的。可能由于效率的原因，对记录以数字顺序排列，但是我们偶尔也想要以字母顺序查看它们。或者，可能数据项本身不可以排序，例如一个月中服务器日志的价值。

10.3 更为高效的排序

因为 Professor 要维护社区的计算设备（全都由竹子、椰子和菠萝组成，并由一个经过 Perl 黑客级别认证的猴子来提供支援），他连续发现有些人把大量数据给猴子支援的文件系统来处理，所以他决定打印出一份违规者名单。

Professor 编写了一个叫做`ask_monkey_about()`的子例程，该子例程接受一个`castaway`的名字，然后返回它们使用的菠萝储量的数目。我们不得不询问猴子，因为他负责掌管这些菠萝。如下所示，从大到小找出违规者的初始方法可能是下面的代码：

```
my @castaways =
qw(Gilligan Skipper Professor Ginger Mary Ann Thurston Lovey);
my @wasters = sort {
    ask_monkey_about($b) <=> ask_monkey_about($a)
} @castaways;
```

从理论上而言，这个程序没有什么问题。对于第一对名字（Gilligan 和 Skipper），我们问猴子：“Gilligan 有多少菠萝？”和“Skipper 有多少菠萝？”这将会从猴子那里得到两个值，然后以此为依据将 Gilligan 和 Skipper 在最终的列表中排序。

然而，在某些时候，我们不得不把 Gilligan 手中持有的菠萝数量与其他成员手中持有的菠萝数目相比较。例如，假定我们需要对比的两个成员是 Ginger 和 Gilligan。我们先问猴子关于 Ginger 的信息，得到一个数目，然后再问猴子关于 Gilligan 的信息……不断地重复。这可能会让猴子感到厌烦，因为我们早前已经问过它了。但我们必须再二再三，甚至再四地去询问每个值，直到把 7 个值按顺序排列。

这可能会成为一个问题，因为这将激怒猴子。

那我们怎么才能将询问猴子的次数减少到最低呢？这样，我们先建一个表格，然后使用 map 语句和七个输入项/输出项，将每个成员的元素变为一个独立的匿名数组引用，每个数组引用包含两个元素，一个是成员名字，另一个是由猴子报告的菠萝数目：

```
my @names_and_pineapples = map {
    [ $_, ask_monkey_about($_) ]
} @castaways;
```

此时此刻，我们一次性向猴子问了 7 个问题，但这也是我们最后一次询问猴子！现在，我们已经有了用于完成任务的所有信息。

如下所示，下一步，按照猴子报告的数值，对数组引用进行排序：

```
my @sorted_names_and_pineapples = sort {
    $b->[1] <=gt; $a->[1];
} @names_and_pineapples;
```

在以上子例程中，\$a 和 \$b 是列表中将要排序的两个元素。当对数字进行排序时，\$a 和 \$b 就是数字；当对引用进行排序时，\$a 和 \$b 就是引用。将它们解引用，还原成相应的数组，然后将数组中第 1 个元素取出（即猴子所报告的菠萝数目）。因为 \$b 排在 \$a 之前，所以这将是一个降序排列。使用降序排列的原因是 Professor 想要知道列表中拿走菠萝最多的那个人是哪一个人。

我们差不多就要完成了，但如果我们要拿走菠萝最多的名字，而不要其他名字和菠萝的数目，该如何操作呢？如下所示，我们不得不在此使用另一个 map 语句，将引用转换回初始数据：

```
my @names = map $_->[0], @sorted_names_and_pineapples;
```

列表中每个元素都在默认变量 \$_ 之中，所以，我们将它解引用，然后取出数组的第 0 个元素，这就是需要的名字。

现在，我们就有一个名字列表，并以他们所持菠萝的数目降序排列，仅仅使用三个简单的步骤，就使猴子的工作减轻了很多。

10.4 施瓦茨变换

这些步骤之间的中间变量并不是必需的，除非作为到下一步的输入。我们将所有这些中间步骤合并到一起，这也能省点力气。

```
my @names =
  map $_->[0],
  sort { $b->[1] <=gt; $a->[1] }
  map [ $_, ask_monkey_about($_) ],
  @castaways;
```

因为 map 操作符和 sort 操作符是从右到左执行的，所以我们应该由下而上地读这个结构：先取数组 @castaways，通过问猴子一些问题之后，创建一些数组引用，将数组引用列表排

序，然后从数组引用中提取相应的名字。我们将得到以期望的顺序排序的名字列表。

这个结构通常叫做施瓦茨变换^{注5}，这个名字以 Randal 命名（但不是由 Randal 本人命名），感谢他多年前在 Usenet 上发表的一篇文章。施瓦茨变换已经被证明是我们的排序技巧的武器库中非常有效的利器。

如果你觉得这个变换看起来太复杂而难以记忆，或者从基本原理得到，考虑一下灵活的部分和固定的部分可能有所帮助：

```
my @output_data =
  map { EXTRACTION },
  sort { COMPARISON }
  map [ CONSTRUCTION ],
  @input_data;
```

基本结构将初始列表映射到一个数组引用列表上，为初始列表中每个成员只进行一次昂贵的计算；然后对数组引用排序，考虑一下每个昂贵函数调用的缓存值^{注6}；然后按新的顺序提取出初始值。我们所做的所有工作就是插入恰当的两个操作，最后就完成了。例如，使用施瓦茨变换执行一个不区分大小写的排序，我们可以按照如下方式使用代码^{注7}：

```
use v5.16;

my @output_data =
  map $_->[0],
  sort { $a->[1] cmp $b->[1] }
  map [ $_, "\F$_" ], # \F is the full case folder from v5.16
  @input_data;
```

关于数组引用，此处没有什么特别要注意的。如果我们不记得放入哪个数组元素的内容，我们可以使用散列引用替换。我们使用 FOLDED 键描述排序值所表述的内容：

```
my @output_data =
  map $_->{ORIGINAL},
  sort { $a->{FOLDED} cmp $b->{FOLDED} }
  map { ORIGINAL => $_, FOLDED => "\F$_" },
  @input_data;
```

10.5 使用施瓦茨变换实现多级排序

如果我们需要以多个判据进行排序，施瓦茨变换依旧可以处理这类任务。如下所示，我们仅仅需要添加一个排序键到匿名数组，然后在比较过程中使用它们：

注 5：施瓦茨变化实际上是一个来自于 Tom Christiansen 的俏皮话，他称这种变换为“黑变换”，但是，Schwartz 在德文中是黑色的意思，这同样也是这种变换发明人的姓氏，因此，Tom 就以这个名字命名，现在变得众所周知了。

注 6：一个昂贵的操作是使用相对较长的时间或者相对较大的内存的操作。

注 7：只有大写转换操作的代价足够高昂，可能是我们的字符串足够的长，或者我们有一个足够大的数字，这个才是一个高效的实现方法。对于少量不是很长的字符串，一个简单的 my @output_data = sort { "\F \$a" cmp "\F \$b" } @input_data 语句可能就已经足够高效。如果还是怀疑，就做基准测试（benchmark）。

```

my @output_data =
  map $_ ->[0],
  sort {
    $a->[1] cmp $b->[1] or
    $a->[2] <=> $b->[2] or
    $a->[3] cmp $b->[3] }
  map [ $_, lc, get_id($_), get_name($_) ],
  @input_data;

```

这段代码拥有一个三层的排序比较结构，使用三个计算的值，存入匿名数组（原始数据项同时也排序，这总是先发生）。

但是这有一点难以记忆。数组引用没有什么特别的。如下所示，我们能够使用散列引用，而且向每个比较层级提供一个描述性的名称：

```

map $_ ->{VALUE},
sort {
  $a->{LOWER} cmp $b->{LOWER}  or
  $a->{ID}     <=> $b->{ID}      or
  $a->{NAME}   AND $b->{NAME}  }
map {
  VALUE  => $_,
  LOWER  => lc,
  ID     => get_id($_),
  NAME   => get_name($_),
},
@input_data;

```

10.6 递归定义的数据

我们当前处理的数据引用到目前为止都是固定的架构，有时候我们不得不处理层次结构的数据，而这些数据通常是递归定义的。

第一个示例考虑如果在一个 HTML 表格中行的单元格里面甚至可能还包含整个表格。第二个示例是由一个目录包含一些文件和其他一些目录的文件系统的可视化表示。第三个示例是公司组织结构图，其中经理们向他们的老板报告，而其中的一些本身可能是经理。第四个示例是更复杂的组织图，包含第一个示例中的 HTML 表格实例，第二个示例中的文件系统表示，甚至第三个示例中整个公司的组织结构图。

我们可以使用引用来获取、存储和处理这些层次化信息。通常情况下，管理数据结构的例程通常都是以递归子例程结束。

递归算法通过从基线条件开始，并且基于这个基线条件处理无限复杂度的数据。递归函数都将有一个基线条件或次要条件，次要条件不需要递归操作，并且所有其他的递归算法表达式都最终能够实现。也就是说，除非我们有很多时间，使函数的递归操作一直运行下去。基线条件考虑在最简单的情况下做什么：叶节点没有分支、数组为空或者计数器为零。在递归算法的不同分支上拥有多个基线条件是很常见的。没有基线条件的递归算法将是无限循环。

递归子例程有一个调用它本身的分支用于处理部分任务，以及一个不调用它本身的分支用于处理基线条件。在第一个示例中，基线条件可以是一个空的表格单元格。也可以是空的表格和空的表格行。在第二个示例中，文件和可能为空的目录将需要基线条件。

例如，如下所示，使用递归子例程处理阶乘函数，这是一个最简单的递归函数：

```
sub factorial {
    my $n = shift;
    if ($n <= 1) {
        return 1;
    } else {
        return $n * factorial($n - 1);
    }
}
```



注意

其他语言有一个叫做尾递归的概念，它们的编译器能够识别尾递归，然后将尾递归转换为一个迭代解决方案，因此，尾递归实际上不是递归。类似于 Perl 的动态语言不能这样做，这是因为再调用子例程之前，子例程定义可能改变。这就意味着我们不得不小心使用 Perl 的递归。

我们的基线条件是 \$n 必须要小于等于 1，这将不会调用递归的实例，如果得到 \$n 大于 1 的递归条件，就将调用例程处理问题中需要计算的部分（例如，计算下一个更小数字的阶乘）。我们宁愿使用迭代更好地解决这个任务，也不使用递归，尽管阶乘的经典定义通常作为递归操作给出。

10.7 构建递归定义的数据

假定我们想要捕获关于文件系统的信息，包括文件名称和目录名称，以及它们包含的内容。我们将目录表述为一个散列，其中散列的键为目录中每个条目的名称，对于普通文件，值为 `undef`。以 `/bin` 目录为例，它可能如下所示：

```
my $bin_directory = {
    cat  => undef,
    cp   => undef,
    date => undef,
    ... and so on ...
};
```

同样，`Skipper` 的主目录可能也包含一个私有的 `bin` 目录（可能类似于 `~/skipper/bin`），其中包含一些私人工具：

```
my $skipper_bin = {
    navigate      => undef,
    discipline_gilligan => undef,
    eat          => undef,
};
```

以上两个示例没有说明目录在一个层次结构里面。它仅仅描述一些目录的内容。

向上一个层级到 Skipper 的主目录，如下所示，该目录可能包含一些文件和私有的 bin 目录：

```
my $skipper_home = {  
    '.cshrc'          => undef,  
    'Please_rescue_us.pdf'  => undef,  
    'Things_I_should_have_packed' => undef,  
    bin                => $skipper_bin,  
};
```

注意，我们现在有三个文件，但是第四项 bin 目录对应的值并不是 undef，而是之前创建的指向 Skipper 的私有 bin 目录的散列引用。这就是指向子目录的方法。如果值为 undef，这个文件就是普通文件；如果值为散列引用，我们就有了一个子目录，该子目录包含它自己的文件和次级子目录。我们合并了这两种初始化（见图 10-1）：

```
my $skipper_home = {  
    '.cshrc'          => undef,  
    'Please_rescue_us.pdf'  => undef,  
    'Things_I_should_have_packed' => undef,  
  
    bin => {  
        navigate      => undef,  
        discipline_gilligan => undef,  
        eat           => undef,  
    },  
};
```

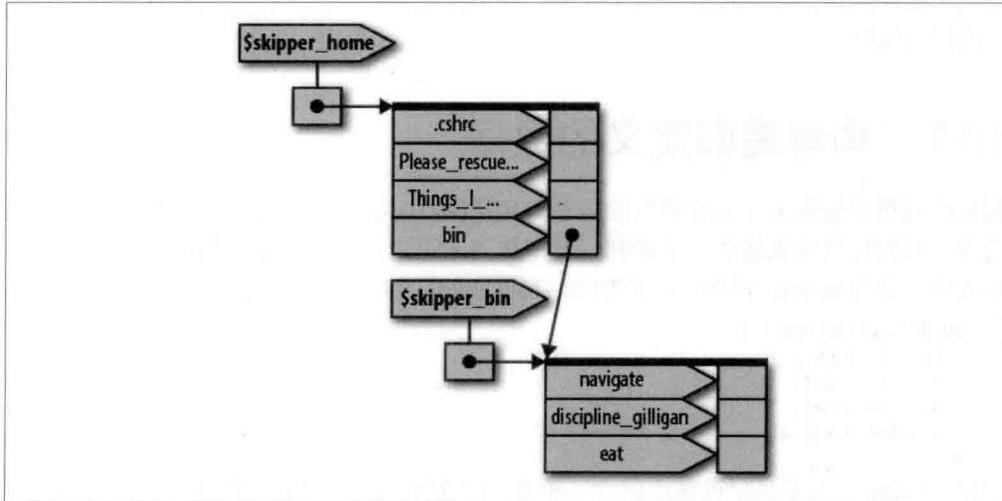


图 10-1 Skipper home PeGS

现在，数据的层次结构开始发挥作用。

显然，我们不想通过改变程序中的文本来创建和维护一个数据结构。我们将通过使用子例

程获取数据。如果路径为文件，就为给定路径名编写一个返回值为 `undef` 的子例程，如果路径是一个目录，就为路径名编写一个返回值为目录内容的散列引用。查看一个文件的基本条件是最简单的，因此我们按照如下方式编写代码：

```
sub data_for_path {
    my $path = shift;
    if (-f $path) {
        return undef;
    }
    if (-d $path) {
        ...
    }
    warn "$path is neither a file nor a directory\n";
    return undef;
}
```

如果 Skipper 在.cshrc 文件中调用上面的程序，他将得到的返回值为 `undef`，表示所看到的是文件。

现在对于目录部分，我们需要一个散列引用，我们在子例程内部将该散列引用声明为一个命名散列。对于散列的每个元素，我们调用自身以填充散列元素的值。这一切将按照如下方式操作：

```
sub data_for_path {
    my $path = shift;
    if (-f $path or -l $path) {          # files or symbolic links
        return undef;
    }
    if (-d $path) {
        my %directory;
        opendir PATH, $path or die "Cannot opendir $path: $!";
        my @names = readdir PATH;
        closedir PATH;
        for my $name (@names) {
            next if $name eq '.' or $name eq '..';
            $directory{$name} = data_for_path("$path/$name");
        }
        return \%directory;
    }
    warn "$path is neither a file nor a directory\n";
    return undef;
}
```

这个递归算法的基本条件是文件和符号链接。如果这个算法允许目录的符号链接，就好像符号链接是真实的（硬）链接，它将不能正确地遍历整个文件系统，因为如果符号链接指向一个包含这个符号链接的目录，这最终将以一个死循环作为结束^{注8}。这样不能正确地遍历一个格式错误的文件系统，格式错误的文件系统是指其中一个目录形成一个环形结构而不是树状结构。尽管格式错误的文件系统可能不一定成为问题，但递归算法通常在遇到递归数据的结构时是十分容易出错的。

注 8：不是所有人都遇到过这类情况，而且想要理解程序死循环的原因。第二次遇到真的不是我们的错误，而且第三次遇到仅仅只是运气糟糕。这就是我们的故事，而且我们将继续下去。

对于目录中的每个文件都要检查，递归调用 `data_for_path` 的响应是 `undef`。这将填充散列中的大多数元素。当返回命名散列的引用时，引用变为匿名散列的引用，这是因为该名称即将超出作用域。（数据本身没有变，但是我们能够访问数据的方法种类发生了变化。）

如果有一个子目录，嵌套的子例程调用 `readdir` 内置函数解析目录的内容，然后返回一个散列引用，该散列引用插入一个由 `caller` 内置函数创建的散列结构中。

首先，这可能看起来有点令人迷惑，但是如果我们慢慢地读完整个程序，我们将看到它总是在做正确的事情。如下所示，通过在“.”（当前目录）上调用它来测试这个子例程的结果，然后检查结果：

```
use Data::Dumper;
print Dumper(data_for_path('.'));
```

显然，如果我们当前目录包含子目录，那就更有趣了。

10.8 显示递归定义的数据

`Data::Dumper` 模块的 `Dumper` 子例程能够非常优雅地显示输出，但是如果我不喜欢这个正在使用的格式该怎么做？我们能够编写一个用于显示数据的程序。再一次，对于递归定义的数据，递归子例程通常就是键。

为了转储数据，我们需要知道在树状结构顶部的目录名称，因为它没有存储在结构中：

```
sub dump_data_for_path {
    my $path = shift;
    my $data = shift;

    if (not defined $data) { # plain file
        print "$path\n";
        return;
    }
    ...
}
```

如下所示，对于普通文件，转储路径名称；对于目录，`$data` 是一个散列引用。遍历所有键，然后转储它们的值：

```
sub dump_data_for_path {
    my $path = shift;
    my $data = shift;

    if (not defined $data) { # plain file
        print "$path\n";
        return;
    }

    foreach (sort keys %$data) {
        dump_data_for_path("$path/$_", $directory{$_});
    }
}
```

对于目录中的每个元素，传递一个路径，该路径由输入路径和当前目录项组成，并且数据指针要么是对于文件的 `undef` 值，要么是对于其他目录的子目录散列引用。如下所示，我们能够通过运行代码得到想要的结果：

```
dump_data_for_path('.');


```

再一次，在一个拥有子目录的目录中这会更有趣，但是，如下所示，得到的输出将与在 shell 交互环境下调用 `find` 命令类似：

```
% find . -print
```

10.9 避免递归

我们在之前的示例中使用递归，因而我们展示过递归的使用方式，但是这并不是完成工作的唯一方法。现在，我们将使用迭代解决方案编写解决方案的代码。为什么？大多数人学习递归，因为其他语言有一种能够将看起来像是递归的代码转换成迭代解决方案的特性。用迭代算法构思很容易，但是这并不意味着程序实际上就按照递归的方式执行。然而，在 Perl 语言中，我们并未从幕后这些重新配置中获益。

迭代解决方案还有其他好处。在 `data_for_path` 的递归版本中，我们构建了目录树深度优先，然后它只能按照深度优先完成。在递归方案中，我们必须一直操作到底部，然后才能移动到下一个顶层继续执行。

要生成一个迭代解决方案，可以使用一个基本模板。我们必须管理我们需要处理的事务列表。一旦有事务在列表中，就继续处理。当处理完列表中的所有事务时，就将返回创建的数据结构。模板如下所示：

```
sub iterative_solution {
    my( $start ) = @_;

    my $data = {};
    my @queue = ( [ $start, $data ] );

    while( my $next = shift @queue ) {
        ... process current element ...
        ... add new things to @queue ...
    }

    return $data;
}
```

在模板中，`@queue` 数组中的每项都携带希望处理的全部元素，其中每个元素都是匿名数组。此时，`$start` 是需要处理的项，并且`$data` 是用于存储结果的引用。尽管在匿名数组中只有两项，但我们将在后续添加另一个吸引人的特性。

首先，如下所示，我们仅仅将递归解决方案转换成迭代解决方案，使用相同的深度优先行为：

```

use File::Basename;
use File::Spec::Functions;

my $data = data_for_path( '/Users/Gilligan/Desktop' );

sub data_for_path {
    my( $path ) = @_;
    my $data = {};
    my @queue = ( [ $path, $data ] );
    while( my $next = shift @queue ) {
        my( $path, $ref ) = @$next;
        my $basename = basename( $path );
        $ref->{$basename} = do {
            if( -f $path or -l $path ) { undef }
            else {
                my $hash = {};
                opendir my $dh, $path;
                my @new_paths = map {
                    catfile( $path, $_ )
                } grep { ! /^\.\.\?\.z/ } readdir $dh;
                unshift @queue, map { [ $_, $hash ] } @new_paths;
                $hash;
            }
        };
        $data;
    }
}

```

在 while 循环内部，我们先获取下一个待处理的元素。取出需要处理的路径和处理结果的引用。因为\$path 是完整路径，并且我们只是想要文件名，所以我们使用 basename 子例程获取想要的文件名作为键。一旦将键添加到\$ref 中，就必须决定这个键对应的值是什么。在 do 语句块内部，有两个分支：如果是文件或者软链接，值就为 undef；如果是目录，就必须创建用于处理的新项，然后将它们添加到@queue 数组中。

为了创建新元素，创建一个叫做\$hash 的匿名散列引用。对于需要处理的新项，将处理的结果存入这个匿名散列引用中。对于需要处理的项，这些引用也将要放入匿名数组内。这是一种很酷的引用技巧：即使分开创建引用，当将它作为\$ref->{\$basename} 的值赋值时，甚至作为空匿名散列时，它也已经成为数据结构的一部分。我们不知道它是数据结构的哪一部分，但是我们没有必要知道。

因为我们想要使用这个深度优先算法，所以当有新元素需要处理时，我们就通过使用 unshift 语句将这个新元素放入@queue 数组最前面。在这个版本中，并没有使它做得比递归解决方案更出色。代码有一些长，第一次看到它时不是那么容易理解，并且它的运行速度已经足够快。

广度优先方案

既然我们已经创建了迭代解决方案，我们就能够比之前的递归版本做得更多一些。我们能够很容易将它改造为广度优先算法，所需要做的只是改变单个关键字。如下所示，通过 `unshift` 语句，将新发现的项放入`@queue` 数组的前方。如果使用 `push` 语句替换，就将新得到的数据项放入`@queue` 数组的末尾：

```
# unshift @queue, map { [ $_, $hash ] } @new_paths;
push @queue, map { [ $_, $hash ] } @new_paths;
```

深度优先算法的版本是后入先出（Last In-First Out, LIFO），广度优先算法的版本是先入先出 First In-First Out, FIFO。两个版本的代码是基本相同的。如果我们是纯粹的计算机科学理论研究者，我们可能指出深度优先版本实际上是一个栈而不是队列，但是我们将只说这是一个有 VIP 部分的队列，其中更重要的 VIP 总是在插队。

广度优先算法有一种极其吸引人的功能：我们能够在任意喜欢的层级很轻易地停留。如果我们仅仅想要到第三层深度，就不必添加任何比该深度更深的数据项用于处理。我们也可以修改递归解决方案实现同样的操作，但是那样做看起来并不友好。

在迭代解决方案中，将存储在`@queue` 数组的匿名数组中追踪层级。在 `do` 语句块的 `else` 分支中，如果当前层级低于阈值层级，就会在`@queue` 数组中添加元素，因为添加的元素会更深一层。

```
use Data::Dumper;
use File::Basename;
use File::Spec::Functions;

my $data = data_for_path( "/Users/brian/Desktop", 2 );
print Dumper( $data );

sub data_for_path {
    my( $path, $threshold ) = @_;
    my $data = {};
    my @queue = ( [ $path, 0, $data ] );

    while( my $next = shift @queue ) {
        my( $path, $level, $ref ) = @$next;
        my $basename = basename( $path );
        $ref->{$basename} = do {
            if( -f $path or -l $path ) { undef }
            else {
                my $hash = {};
                if( $level < $threshold ) {
                    opendir my $dh, $path;
                    my @new_paths = map {
                        catfile( $path, $_ )
                    } readdir( $dh );
                    push @queue, map { [ $_, $hash ] } @new_paths;
                }
            }
        };
    }
}
```

```

        } grep { ! /^\.\.?\z/ } readdir $dh;

        push @queue, map { [ $_, $level + 1, $hash ] } @new_paths;
    }
    $hash;
}
};

$data;
}

```

很漂亮，对吧？尽管它变得更好了。让用户决定他或者她想要深度优先遍历还是广度优先遍历实在是太难了。如果用户想要一次性处理特定数目的项这可能就变得很有趣，留下一些暂不处理。我们不会破坏掉这个乐趣，因为你将在练习 5 中完成这个内容。

10.10 练习

可以在附录中的“第 10 章答案”部分找到这些练习的答案。

1. [15 分钟] 使用 glob 操作符，对你的主目录中的每个名字，通过它们的相对大小进行简单排序，可能得到如下所示的结果：

```

chdir; # the default is our home directory
my @sorted = sort { -s $a <=> -s $b } glob '*';
运用施瓦茨变换技术重写以上代码。

```

2. [15 分钟] 仔细研读 Perl 的核心模块 Benchmark 的文档。编写一个将回答下列问题的程序，“在练习 1 的执行中使用施瓦茨变换将提速多少？”
3. [10 分钟] 使用一个施瓦茨变换，读取一个单词列表，然后对它们按“字典顺序”排序，字典排序忽略所有字母的大写和内部标点符号。提示：后续的转换操作可能有些用处：

```

my $string = 'Mary Ann';
$string =~ tr/A-Z/a-z/;      # force all lowercase
$string =~ tr/a-z//cd;       # strip all but a-z from the string
print $string;               # prints "maryann"

```

确定你没有损坏数据！如果输入的内容包含 Professor 和 Skipper，输出就将以同样的顺序罗列，并且首字母大写。

4. [20 分钟] 修改递归目录的转储子例程，这样就会以缩进方式显示嵌套目录。一个空目录将按照如下格式显示：

```
sandbar, an empty directory
```

而一个非空目录将使用缩进两个空格的方式显示嵌套内容：

```
uss_minnow, with contents:  
  anchor  
  broken_radio  
  galley, with contents:  
    captain_crunch_cereal  
    gallon_of_milk  
    tuna_fish_sandwich  
    life_preservers
```

5. [20分钟]修改 `data_for_path` 的递归版本，同时处理深度优先遍历和广度优先遍历。

使用一个可选的参数作为第三个参数，让用户判断使用哪一种方式实现：

```
my $depth =  
  data_for_path( $start_dir, $threshold, 'depth-first' );  
  
my $breadth =  
  data_for_path( $start_dir, $threshold, 'breadth-first' );
```

第 11 章

构建更大型的程序

当程序变得越来越大时，我们开始意识到一些代码可以应用于其他工作之中。我们能够移动一些代码到库文件中，这样就可以在一些不同程序之间，甚至与其他一些人共享该库文件。我们也能够使用库，通过库函数或者 `use` 语句隔离代码，使其实现与不相关任务之间的代码分离。

11.1 修改通用代码

应 Minnow 的请求，Skipper 编写很多 Perl 程序，为所有通用港口提供导航服务。如下所示，他发现自己总是不停地在不同的程序之间复制、粘贴一个通用的子例程：

```
sub turn_toward_heading {
    my $new_heading = shift;
    my $current_heading = current_heading();
    print "Current heading is ", $current_heading, ".\n";
    print "Come about to $new_heading ";
    my $direction = 'right';
    my $turn = ($new_heading - $current_heading) % 360;
    if ($turn > 180) { # long way around
        $turn = 360 - $turn;
        $direction = 'left';
    }
    print "by turning $direction $turn degrees.\n";
}
```

该例程通过由 `current_heading` 子例程返回的当前航向，然后向最短转向提供到由子例程的第一个参数表示的新航向的方法。

该子例程的第一行可能如下所示：

```
my ($new_heading) = @_;
```

这通常是一种风格调用：在两种情况中，第一个参数以`$new_heading` 结束。然而，从

`@_` 数组移除已经被识别的数据项比较方便。所以，我们坚持(大多数情况下)使用“shift”风格的参数解析。现在回到手上的程序。

通过使用该例程编写多个程序之后，Skipper 发现当他花时间调整到正确的航向(或者可能开始在合适的航向前行)时，得到的输出非常口语化。毕竟，如果当前航向是 234° ，并且他需要转向 234° ，我们将看到如下输出：

```
Current heading is 234.  
Come about to 234 by turning right 0 degrees.
```

真烦人，Skipper 决定通过检查转向值为 0 修正这个问题：

```
sub turn_toward_heading {  
    my $new_heading = shift;  
    my $current_heading = current_heading();  
    print "Current heading is ", $current_heading, ".\n";  
    my $direction = 'right';  
    my $turn = ($new_heading - $current_heading) % 360;  
    unless ($turn) {  
        print "On course (good job!).\n";  
        return;  
    }  
    print "Come about to $new_heading ";  
    if ($turn > 180) { # long way around  
        $turn = 360 - $turn;  
        $direction = 'left';  
    }  
    print "by turning $direction $turn degrees.\n";  
}
```

非常好！新版本的子例程在当前导航程序中非常好用。然而，因为他之前已经使用复制-粘贴的方法将该子例程添加到多个其他的导航程序中，所以这些其他程序仍然使 Skipper 对于无关的转向消息感到厌烦。

Skipper 需要一种在一个地方编写代码然后与其他一些程序共享的方法。与 Perl 中的很多事物类似，这有很多方法可以实现。

11.2 使用 eval 插入代码

Skipper 能够通过将 `turn_toward_heading` 的定义放入一个独立的文件中，以节省磁盘容量(和大脑空间)。例如，假如 Skipper 发现多个与导航相关的常用子例程中，Minnow 似乎在为任务所编写的大多数或者所有程序里使用。这样，他就能够将这段代码放置到一个叫做 `Navigation.pm` 的独立文件之中，该文件由所需的多个子例程组成。



注意

.pm 扩展名是“Perl 模块”的意思，而且这个扩展名是很重要的，这将会在后续部分中详细叙述。因为我们告知解释器我们想要的确切文件名，所以我们在此不需要一些特殊的文件名或者扩展名。在 Perl 的早

期版本中，人们使用.pl（Perl Library 的缩写），但是，现在人们也对于 Perl 程序使用该扩展名。

但是现在，我们如何使 Perl 解释器从其他文件中导入那个程序片段？我们可以运用硬编码的方法，使用第3章中介绍过的 eval 表达式的字符串形式：

```
sub load_common_subroutines {
    open my $more_fh, '<', 'Navigation.pm' or die "Navigation.pm: $!";
    undef $/; # enable slurp mode
    my $more_code = <$more_fh>;
    close $more_fh;
    eval $more_code;
    die $@ if $@;
}
```

Perl 将来自于 Navigation.pm 的代码读入 \$more_code 变量。然后使用 eval 表达式以 Perl 代码的方式处理读入的文本。\$more_code 中的任何词法变量当作本地变量求值。如果有任何语法错误，Perl 解释器就以适当的错误消息设定 \$@ 变量，然后程序退出。



注意

eval 语句能够在调用它的作用域内访问任何词法变量。它并不像子例程调用一样创建新作用域。

现在，不必把多行相同的子例程部署于每个文件之中，我们可以将以下子例程插入到每个文件中实现同样的功能：

```
load_common_subroutines();
```

但这样并不漂亮，特别是当我们需要持续重复实现这类工作的时候。很幸运，Perl 有很多种方法解决这类问题。

11.3 使用 do 语句

Skipper 在 Navigation.pm 中部署了一个常见的导航子例程。如果 Skipper 只插入如下语句：

```
do 'Navigation.pm';
die $@ if $@;
```

到他的标准导航程序中，就与我们在程序中同一位置使用 eval 语句执行代码得到的结果几乎相同^{注1}。

也就是说，do 操作符的功能就像是把来自 Navigation.pm 的代码合并进当前程序中，尽管导入的变量还在它自己的作用域语句块中，因此所导入文件中的词法（my 变量）和大多数指令（例如 use strict）不会泄漏到主程序之中。

注 1：除了关于“@INC, %INC”和丢失文件的处理，后续详细介绍这些内容。

现在 Skipper 能够安全地更新和维护一个常见的子例程副本，而不必将所有修改和扩展的代码一遍又一遍地复制到他所创建和使用的众多独立的导航程序中。

这需要一点规则，因为现在破坏当前给定子例程的接口，就会摧毁很多程序而不仅仅是一个^{注2}。一旦我们使其他人使用我们的代码，我们实际上就被我们的接口困住了，因为我们的用户将会抱怨任何迫使修改自己代码的事情。

通过把一些代码放入一个分离的文件中，其他程序员能够重用 Skipper 的例程，反过来也一样，通过共享文件而不是共享整个应用。如下所示，如果 Gilligan 编写了一个 drop_anchor 例程，然后将它放入 DropAnchor.pm 文件中，此时 Skipper 能够通过导入他的库文件的方式，使用 Gilligan 的代码：

```
do 'DropAnchor.pm';
die $@ if $@;
...
drop_anchor() if at_dock() or in_port();
```

因此，我们通过不同的独立文件导入的代码能够很方便地维护和协同编程。这并不意味着这样就足够了，我们仍然要做好的团队成员。



注意

do 语句并不会像第 2 章中所展示的 use 语句和 require 语句一样搜索模块目录。我们应该向 do 语句提供指向文件的绝对路径或者相对路径。

当加载了一个模块时，就从一个.pm 文件导入代码，并且能够直接运行模块中可执行的语句，而这比使用 do 语句加载文件的方式定义子例程更为常用。

我们再回到 Drop_anchor.pm 库，如果 Skipper 想要编写一个既要导航又要“drop anchor”的程序，该如何操作呢？如下所示，他需要加载两个文件：

```
do 'DropAnchor.pm';
die $@ if $@;
do 'Navigation.pm';
die $@ if $@;
...
turn_toward_heading(90);
...
drop_anchor() if at_dock();
```

这些语句工作得很好也很顺利。子例程在两个库中定义，并且都在该程序中可用。

11.4 使用 require 语句

假定 Navigation.pm 模块本身由于一些常见的导航任务需要将 DropAnchor.pm 模块导入。Perl 解释器将直接一次性读取文件，然后在处理 Navigation.pm 模块时再次读取

注 2：在第 14 章将展示建立测试的方法，用于维护重用的代码。

DropAnchor.pm 模块。因此，在这里重复定义 drop_anchor 是毫无必要的。更糟糕的是，如果我们打开警告，我们将从 Perl 解释器得到一个关于重复定义子例程的警告，尽管子例程的内容是相同的。

我们需要一个机制，来追踪哪些文件已经导入，然后仅导入一次。Perl 提供这类操作，叫做 require。如下所示，将之前的代码修改为：

```
require 'DropAnchor.pm';
require 'Navigation.pm';
```

require 操作符追踪 Perl 解释器读取的文件。一旦 Perl 解释器成功处理了一个文件，它就忽略在相同文件上任何进一步的 require 操作。这就意味着即使 Navigation.pm 包含 require "DropAnchor.pm" 语句，Perl 解释器也只会导入 DropAnchor.pm 一次。因此，我们就不会再收到关于重复子例程定义的令人厌烦的错误消息（参考图 11-1）。更重要的是，我们不必多次处理相同的文件，这样也节省时间。



注意

如 perlfunc 文档中的 require 操作符条目所述，Perl 解释器使用 %INC 散列追踪加载的模块。

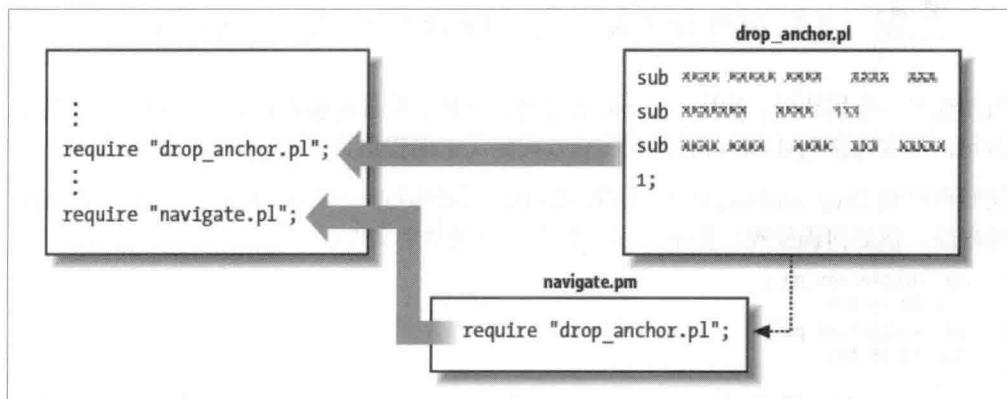


图 11-1 一旦 Perl 解释器导入 DropAnchor.pm 文件，它就忽略其他对该文件的导入请求

require 操作符也有两个额外的特性：

- 导入文件中的任何语法错误都将终止程序，所以不再需要很多 “die \$@ if \$@” 语句；
- 文件中的最后一个求值表达式必须返回一个真值，因此 require 语句才能知道该文件正确导入。

正是由于第二点，大多数用于 require 导入的文件中，最后一行代码总有一个神秘的 1。这将确保最后的求值表达式为真。努力保持这个传统吧。

11.5 命名空间冲突的问题

通常情况下 Skipper 能够顺利地驾驶一艘船驶向一座岛屿，但有时在 Perl 程序中，仅仅只是两个名字陷入冲突。假如 Skipper 将他所有很酷并且很实用的例程加入 Navigation.pm 模块中，而且 Gilligan 将这个库合并到他自己的 head_toward_island 导航包中：

```
require 'Navigation.pm';

sub turn_toward_port {
    turn_toward_heading(compute_heading_to_island( ));
}

sub compute_heading_to_island {
    .. code here ..
}

.. more program here ..
```

Gilligan 此时需要调试他的程序（可能在 Professor 的帮助下），然后程序能够正常工作。

然而，现在 Skipper 决定修改他的 Navitgation.pm 库，添加一个叫做 turn_toward_port 的例程，该例程提供一个向左转 45° 的操作（在航海术语中叫做“左弦”）。

一旦 Gilligan 尝试左转 45° 的操作，他的程序将以一个灾难性的方式结束：他将控制船开始转圈！问题在于 Perl 编译器首先从 Gilligan 的主程序编译 turn_toward_port 子例程，然后当 Perl 解释器在运行时对于 require 语句求值时，它将作为 Skipper 的定义重新定义 turn_toward_port 子例程。的确如此，如果 Gilligan 打开警告，他将注意到这类错误的发生，但是他怎么会意料到这些呢？

问题在于 Gilligan 定义 turn_toward_port 子例程作为“转向岛屿的方向”的含义，而 Skipper 将该子例程定义作为“向左转”。我们该怎么解决这个问题呢？

其中一个导入方法是在库文件定义的每个名称之前放置一个显式的前缀，例如使用 navigation_。然而，Gilligan 的程序就以如下方式结束：

```
require 'Navigation.pm';

sub turn_toward_port {
    navigation_turn_toward_heading(compute_heading_to_island( ));
}

sub compute_heading_to_island {
    .. code here ..
}

.. more program here ..
```

很明显，navigation_turn_toward_heading 子例程是包含在 Navigation.pm 文件之中。这

对于 Gilligan 非常棒，但是对于 Skipper 就很尴尬，因为他的文件现在拥有更长的子例程名称：

```
sub navigation_turn_toward_heading {
    .. code here ..
}

sub navigation_turn_toward_port {
    .. code here ..
}

1;
```

是的，现在每个标量、数组、散列、文件句柄或者子例程，前面都有一个 `navigation_` 前缀，以确保该名称不会与库文件任何潜在的用户发生命名冲突。很明显，对于老水手而言，这将不会使他们的船浮起。我们怎么做才更好呢？

11.6 使用包作为命名空间分隔符

如果上一节中最后一个示例的名称前缀不必每次使用时都清楚地拼写，事情就会简单很多。如下所示，可以使用包改进这个情况：

```
package Navigation;

sub turn_toward_heading {
    .. code here ..
}

sub turn_toward_port {
    .. code here ..
}

1;
```

在文件初始部分的包声明，告诉 Perl 解释器实际上将 `Navigation::` 插入文件中的大多数（变量）名称之前。这样，对于该子例程，之前的代码隐式地使用完全限定的包名称：

```
sub Navigation::turn_toward_heading {
    .. code here ..
}

sub Navigation::turn_toward_port {
    .. code here ..
}

1;
```

现在，当 Gilligan 使用该文件时，他只需在库函数中定义的子例程名称前添加 `Navigation::` 前缀，在他自己定义的同名子例程前则不必添加 `Navigation::` 前缀：

```
require 'Navigation.pm';

sub turn_toward_port {
    Navigation::turn_toward_heading(compute_heading_to_island(  ));
}

sub compute_heading_to_island {
    .. code here ..
}

.. more program here ..
```

包名与变量名相同：它们由字母、数字和下划线组成，但不能以数字开头。同样，由于 perlmodlib 文档中解释的原因，包名应当以一个大写字母开头，而且不会与已经存在于 CPAN 的模块或者 Perl 的核心模块名称重叠。



注意

Perl 作者上传服务器 (Perl Authors Upload Server, PAUSE) 对于命名规范和你的模块如何适应于 CPAN 上已有的模块提出一些建议。可以参考第 21 章以获得关于这类问题的更多信息。

包名也可以有由双冒号分割的多个名称，例如 Minnow::Navigation 和 Minnow::Food::Storage。

几乎所有的标量、数组、散列、子例程和文件句柄名称^{注3}实际上已经隐式添加了当前包名作为前缀，除非名已经包含一个或者多个双冒号标记。

因此，在 Navigation.pm 模块中，能够按照如下方式使用变量：

```
package Navigation;
@homeport = (21.283, -157.842);

sub turn_toward_port {
    .. code ..
}
```



注意

小注：北纬 21.283°，西经 157.842° 是现实生活中码头的位置，该码头也是一个著名的系列电视剧的开机拍摄地点。如果你不相信，可以在 Google maps 上查看。

如下所示，可以在主程序代码中使用完整的包规范引用 @homeport 变量：

```
@destination = @Navigation::homeport;
```

如果每个名字前都插入包名，那么主程序的名称是什么？是的，它们同样在一个叫做 main 的包中。就好像 main 包放置在每个文件的起始位置。因此，要想使 Gilligan 避免

注 3：除了专门词汇 (lexical) 之外，我们将在后续详细描述。

使用 `Navigation::turn_toward_heading`, `Navigation.pm` 模块文件可以按照如下方式表述:

```
sub main::turn_toward_heading {  
    .. code here ..  
}
```

现在子例程在 `main` 包中定义, 而不是 `Navigation` 包中。这不是一个最优解决方案(当第 17 章讨论 Exporter 时, 将展示更好的解决方案), 但是, 至少目前与任何其他包相比, `main` 包没有什么神圣或者极度的特殊。



注意

如果我们了解包, 就能够从程序中的任何位置访问包的变量和包的子例程。

这就是在第 2 章中导入符号到脚本中时模块所做的事情, 只是我们在当时没有介绍全部内容。这些模块导入子例程和变量到当前包中(再一次, 这通常是脚本中的 `main` 包)。也就是说, 除非使用完整的包规范, 否则这些符号仅在那个包中可用。我们后续将深入介绍更多细节。

11.7 Package 指令的作用域



注意

Perl 不会强迫我们像 C 语言一样创建显式的 `main` 循环。Perl 解释器知道每个脚本都需要一个, 因此它就将自动做这个工作。

所有 Perl 文件运行时就好像我们有一个 `main` 包定义在起始部分。直到声明下一条包命令之前, 当前所有包指令仍然有效, 除非包指令在一个带大括号的作用域中。在那种情况下, Perl 解释器会记住之前的包指令, 当带大括号的作用域结束时, 就还原之前的包指令。可以参考如下示例:

```
package Navigation;  
  
{ # start scope block  
    package main; # now in package main  
  
    sub turn_toward_heading { # main::turn_toward_heading  
        .. code here ..  
    }  
  
} # end scope block  
  
#. back to package Navigation  
  
sub turn_toward_port { # Navigation::turn_toward_port  
    .. code here ..  
}
```

当前包是在词法作用域的，这与 my 变量的作用域类似，范围限制在我们定义包指令时的最内层的一对封闭大括号或者文件中。

大多数库在文件的起始部分仅有一个包声明。大多数程序把 main 包作为默认的包名。然而，知道我们能够临时拥有另一个当前包也是不错的。



注意

不论当前包的定义如何，一些名称总是在 main 包中：ARGV、ARGVOUT、ENV、INC、SIG、STDERR、STDIN 和 STDOUT。我们能够总是引用@INC 数组，而且确定能够得到的是@main::INC 数组。标点符号变量（例如\$_、\$2 和 \$!）要么是专门词汇，要么被强制放入 main 包中。因此当我们写\$.的时候，我们绝不会错误地得到\$Navigation::这样的结果。

11.8 包和专门词汇

词法变量（通过 my 关键字引入的变量）不使用当前包作为前缀，因为包变量总是全局变量：如果我们知道包变量的完整名称，就可以始终对该包变量取引用。词法变量对于程序的一部分而言通常是临时的并且可访问的。如果我们声明一个词法变量，然后不带包前缀使用该变量名称，就将得到该词法变量。包前缀能够确保我们访问的是包变量，而绝不会访问词法变量。

例如，假定 Navigation.pm 中的一个子例程声明了一个词法变量@homeport。@homeport 变量的任何引用将重新在当前包中创建词法变量，但是一个完全限定的@Navigation::homeport 名称将访问 Navigation.pm 包中的@homeport 变量。

```
package Navigation;
our @homeport = (21.283, -157.842); # package version
sub get_me_home {
    my @homeport;

    .. @homeport .. # refers to the lexical variable
    .. @Navigation::homeport .. # refers to the package variable
}
```

.. @homeport .. # refers to the package variable

很明显，这样将导致令人迷惑的代码，因此我们将不会介绍这些不必要的重复内容。然而，如果我们知道规则，结果就将完全可预测。

包语句块

从 Perl v5.12 开始，我们能够使用一个新语法，如下所示，允许我们在包语句中使用

语句块：

```
package Navigation {  
    my @homeport = (21.283, -157.842); # package version  
  
    sub get_me_home {  
        my @homeport;  
  
        .. @homeport .. # refers to the lexical variable  
        .. @Navigation::homeport .. # refers to the package variable  
  
    }  
  
    .. @homeport .. # refers to the package variable  
}
```

这与包在裸语句块中的操作没有什么不同，如下所示，这样实现甚至看起来更好：

```
{  
    package Navigation;  
    my @homeport = (21.283, -157.842); # package version  
  
    sub get_me_home {  
        my @homeport;  
  
        .. @homeport .. # refers to the lexical variable  
        .. @Navigation::homeport .. # refers to the package variable  
  
    }  
  
    .. @homeport .. # refers to the package variable  
}
```

不管使用哪种方法，我们都能够在该语句块的作用域中使用词法变量，其中在语句块中恰巧使用包变量。我们不必一次性定义所有包，因此，如果在别处定义包的其他部分，并且期望能够访问这些变量，这些语句才有效。

如下所示，这对于在同一文件中定义多个小的包也非常便捷：

```
use v5.12;  
  
package Navigation {  
    ...  
}  
  
package DropAnchor {  
    ...  
}
```

通常情况下，每个文件使用一个包，这就可以达到基本相同的效果，但是通常情况下巧合的是，因为我们使用的词法变量的作用域碰巧也是这个文件（而且不是这个包）。表示包的作用域也显式地表明了我们所使用词法变量的作用域。

直到第 12 章都不会介绍太多关于版本的问题，如下所示，但是新的包语法允许我们指定一个版本，使用或者不使用语句块均可：

```
use v5.12;  
  
package Navigation 0.01;  
  
package DropAnchor 1.23 { ... }
```

这对于设定\$VERSION 变量而言确实是一条捷径，当其他代码想要包的版本信息时，就可以查找这个变量。然而，这是一个正规的 Perl 标量，如果需要，我们自己可以直接设定它。

11.9 练习

可以在附录中的“第 11 章答案”部分找到这些练习的答案。

1. [30 分钟] 岛上的 Oogaboogoo 土著人对于日期和月份有特别的名称。如下代码由 Gilligan 编写，但是并不完善。修改下面的代码，添加一个月份名称转换函数，然后将所有这些放入一个库文件中。更进一步，添加合适的错误检查，并且考虑文档中应该写入哪些部分。

```
@day = qw(ark dip wap sen pop sep kir);  
sub number_to_day_name { my $num = shift @_; $day[$num]; }  
@month = qw(diz pod bod rod sip wax lin sen kun fiz nap dep);
```

2. [10 分钟] 使用你的库文件和下面的代码，编写一个程序，用于输出消息，例如今天是 dip sen 15 2011，意思是今天是 8 月的一个星期一。你可能需要使用 localtime 内置函数：

```
my($sec, $min, $hour, $mday, $mon, $year, $wday) = localtime;
```

提示：由内置函数 localtime 返回的年和月的数字可能不是你所期望的，因此你可能需要查询文档以获取更多的信息。

第 12 章

创建你自己的发行版

到目前为止，我们一直将 Perl 作为一门语言来介绍。本书剩下的部分将讲述 Perl 的开发流程。我们基本上已经可以开发模块，但在此之前，我们将展示如何创建 Perl 模块的发行版。其实我们并非一定需要用这些生成模块框架的发行版来创建模块，但使用它们会让模块开发变得简单一些；Perl 的绝大多数工具是按照发行版形式构建的。除了这些之外，我们可以在开发时就对代码进行测试，而不是拖到最后。

12.1 Perl 模块的两个构建系统

构建系统将我们发布的所有文件合并到我们实际上独立安装的文件中，这些文件在安装时可能需要进行编译，在代码中插入配置信息，或者其他开发者想要实现的一些其他内容。总之，一旦文件转换完成，构建系统就完成发行版的构建。

Perl 有两种常见的发行版构建系统。ExtUtils::Makemaker 模块基于 make 构建，make 是一个源自 UNIX 社区的依赖关系管理工具。使用这个模块构建的发行版使用一个叫做 Makefile.PL 的文件来控制构建过程。如果我们想定制构建脚本，就不得不了解 make 工具配置语言，并保证我们增加的部分是可移植的。尽管 ExtUtils::Makemaker 模块得到很好的支持，但它太老了，不会再增加新的特性。尽管如此，CPAN 上的许多发行版仍然继续使用 ExtUtils::Maker 模块构建，这么多开发者继续使用它的原因就是：它很稳定，在大多数情况下都工作得很好。

较新的系统使用 Module::Build 模块构建，这是一个纯粹的 Perl 工具。如果我们想要安装 Perl 模块，就一定已经有 Perl 运行环境了。使用一个只需要 Perl 的构建系统，意味着我们不需要安装任何额外的东西。然而，Module::Build 模块只是从 Perl v5.10 才被加入到标准库之中。使用这个模块生成的发行版都有一个 Build.PL 文件。

一些发行版的安装包里不仅有 Makefile.PL，还有 Build.PL。其中的一些安装包可以使用任何

一个安装方式，而另外一些安装包就只是其中一种安装方式对另一种提供的简单封装。

不管我们选择哪一个，发行版的文件结构是几乎相同的，所以本章中所有的内容都适用于任何一种构建系统。不过我们会提及这两种方式的区别。

12.1.1 在 Makefile.PL 内部

ExtUtils::Makemaker 模块通常使用 Makefile.PL 构建，但这名字实在没有什么特别的，只是相关工具链按照惯例使用它而已。根据其文件扩展名可以知道，这只是一个 Perl 程序。其中，WriteMakefile 子例程以键值对形式获取配置信息，然后将它们转换生成一个 Makefile 格式的文件——源自于 Unix 社区强大的依赖关系管理系统。

如下所示，由 module-starter 程序创建的启动执行代码 Makefile.PL 的内容：

```
use 5.006;
use strict;
use warnings;
use ExtUtils::MakeMaker;

WriteMakefile(
    NAME          => 'Animal',
    AUTHOR        => q{Willie Gilligan <gilligan@island.example.com>},
    VERSION_FROM  => 'lib/Animal.pm',
    ABSTRACT_FROM => 'lib/Animal.pm',
    PL_FILES      => {},
    PREREQ_PM     => {
        'Test::More' => 0,
    },
    dist          => { COMPRESS => 'gzip -9f', SUFFIX => 'gz', },
    clean         => { FILES => 'Animal2-*' },
);

```

ExtUtils::Makemaker 模块的文档解释了这些配置的细节。我们此时感兴趣的是 PREREQ_PM 键对应的值，它列出在运行代码时需要的依赖模块及其版本号。当在这里列出依赖的模块时，CPAN 客户端都能够自动获取、构建并且安装它们。

PREREQ_PM 的设置信息都是很笼统的，但是我们可以使用其他键向 CPAN 客户端提供更多信息，配置项 CONFIGURE_REQUIRES 和构建项 BUILD_REQUIRES 使用相同的配置信息格式，但是为这些步骤列出依赖关系。我们需要一些模块来安装和运行该构建，但是安装完成这些模块之后，就不再需要了。而测试模块，在第 14 章和第 20 章提到的，只是在构建阶段才需要。如果我们需要一个特定的模块或者版本来运行构建文件，我们可以做相应的修改。ExtUtils::Makemaker 模块直到 6.56 版才支持构建项 BUILD_RREQUIRES 的配置，所以，如果我们想使用这个内容，就必须指定 6.56 或以上的版本。如下所示，这个可以在 Makefile.PL 中作为参数设置 CONFIGURE_REQUIRES 配置项：

```
use ExtUtils::Makemaker 6.56;

WriteMakefile(
    ...
)
```

```
CONFIGURE_REQUIRES => {
    'ExtUtils::Makemaker' => 6.56,
},
BUILD_REQUIRES => {
    'Test::More' => 0,
},
PREREQ_PM => {
    ...
},
...
);
```

当我们需要运行 ExuUtils::Makemaker 模块时，将它本身作为一个依赖条件写进配置信息，似乎显得有点奇怪，因为我们已经在用它了。但实际上当我们将所有信息打包放入发行版时，构建程序将创建包含所有配置信息的 META.yml 或 META.json 文件。CPAN 客户端会解压发行版，然后查看 META 文件，确定在运行 Makefile.PL 之前需要做的事情。

如下所示，另一个有用的设置是 EXE_FILES 键，在它对应的值里面列出发行版中包含的可安装程序。使用一个数组引用指向这些信息：

```
use ExtUtils::Makemaker 6.56;

WriteMakefile(
    ...
    EXE_FILES => [ qw( scripts/barnyard.pl ) ],
    ...
);
```

当在示例中创建程序时，我们将它们放入 scripts 目录。该目录的名称不是很特别；它仅仅是提供给 EXE_FILES 键的相对路径。

尽管我们通常将 Perl 发行版认为是模块的归档，但使用 EXE_FILES 键，我们可以发布没有模块的程序。

12.1.2 在 Build.PL 文件内部

如果使用 Module::Build 模块，我们就要一个 Build.PL 文件而不是 Makefile.PL 文件。它们看起来挺像，但略有不同。我们不是调用一个子例程，而是创建一个对象，然后调用 `create_build_script` 程序。通常，我们并不在意这些。对于复杂的构建流程，我们可以通过继承的方式扩展 Module::Build 模块来完成我们想要做的任何事情。事实上，开发 Module::Build 模块的一个动机就是增加构建的灵活性。

```
use 5.006;
use strict;
use warnings;
use Module::Build;

my $builder = Module::Build->new(
    module_name      => 'Animal',
    license          => 'perl',
    dist_author      => q{Willie Gilligan <gilligan@island.example.com>},
```

```
dist_version_from => 'lib/Animal.pm',
build_requires => {
    'Test::More' => 0,
},
requires => {
    'perl' => 5.006,
},
add_to_cleanup     => [ 'Animal-*' ],
);

$builder->create_build_script();
```

Module::Build::API 的文档为新手解释了有效键的所有信息。如下所示，对于 EXE_FILES 键，Module::Build 模块使用 script_files 键替换。

```
my $builder = Module::Build->new(
    ...
    script_files      => [ qw(scripts/barnyard.pl) ],
    ...
);
```

12.2 我们的第一个发行版

有好几种方法可以构建一个 Perl 发行版，但我们只打算介绍其中两个。我们使用哪个工具取决于我们想要的控制程度和是否喜欢工具为我们做的默认选择。大部分工具只是生成相同的基本目录结构，因此一旦生成了发行版，这些工具也就没用了。

我们将要创建一些描述牧场中动物的 Animal 模块。

12.2.1 h2xs 工具

h2xs 工具来自 Perl 语言。顾名思义，设计这个工具用来将 C 语言的.h 头文件转换成_xs 文件，作为连接 C 和 Perl 的胶水语言。现在，它不止做这些，而且它的主要优点是作为 Perl 标准库中创建模块工具。如果我们已经安装 Perl，我们就已经拥有了这个工具。

要开始创建 Animal 模块，带-A 和-X 参数运行 h2xs，用来关掉 AUTOLOAD 和 XS^{注1} 特性（说来有趣，这两个特性是 h2xs 得以存在的最主要原因），然后使用-n 开关设置模块名称：

```
% h2xs -AX -n Animal

Writing Animal/lib/Animal.pm
Writing Animal/Makefile.PL
Writing Animal/README
Writing Animal/t/Animal.t
Writing Animal/Changes
Writing Animal/MANIFEST
```

注 1：大家如果想了解更多关于 XS 的内容，可以查阅 Simon Czens 和 Tim Jenness 合著的 *Extending and Embedding Perl* (Manning 出版社)。

输出显示 h2xs 创了一个 Animal 目录和它下面的一些文件。我们后续将解释每个文件的用途。h2xs 创建的文件是完全可用的，但是有另外一些工具能够为我们做更多。

12.2.2 Module::Starter 模块

实际中更加通用的做法是使用 Module::Starter 模块，尽管该模块并不包含在标准库中。通过填入一些更详细的信息，该模块能够为我们对输出结果提供更强大的控制。通过 module-starter 程序，可以设定我们的姓名和电子邮件，这些信息将被插入相关文件合适的地方。我们还喜欢它的一个特性就是--verbose 选项，如下所示，通过开启这个设置就可以看到模块正在为我们做什么。

```
% module-starter --module=Animal --author="Gilligan" /  
--email=gilligan@island.example.com --verbose  
Created Animal  
Created Animal/lib  
Created Animal/lib/Animal.pm  
Created Animal/t  
Created Animal/t/pod-coverage.t  
Created Animal/t/pod.t  
Created Animal/t/boilerplate.t  
Created Animal/t/oo-load.t  
Created Animal/.cvignore  
Created Animal/Makefile.PL  
Created Animal/Changes  
Created Animal/README  
Created Animal/MANIFEST  
Created starter directories and files
```

默认情况下，module-starter 程序会创建一个带 Makefile.PL 的安装目录系统发行版。如果想使用 Module::Build 模块替代，如下所示，我们就可以使用--builder 开关来设置我们想要使用的构建系统：

```
% module-starter --builder="Module::Build" --author="Gilligan" /  
--email=gilligan@island.example.com --verbose
```

其实我们没必要输入--builder 选项的完整内容，因为 module-starter 提供--mb 的一个缩写形式以实现相同的功能：

```
% module-starter --mb --author="Gilligan" /  
--email=gilligan@island.example.com --verbose
```

我们不想每次都在命令行中输入这么长的命令，以便 module-starter 能从配置文件 \$HOME/.module-starter/config 中读取信息。如果我们使用 Windows 系统，那么.module-starter 文件的名称可能有点小问题，因此我们可以设定 MODULE_STarter_DIR 环境变量来指定包含 config 的目录名称。

在配置文件内部，可以列出用冒号分割的所有参数名称和值：

```
author: Willie Gilligan  
email: gilligan@island.example.com  
builder: Module::Build  
verbose: 1
```

一旦我们建立配置文件，一切事情就会变得很轻松，因为我们只需要设置想要创建的发行版名称就可以：

```
% module-starter --module=Animal
```

12.2.3 定制模版

在我们使用自己的偏好设定开发编写多个发行版之前，Module::Starter 模块创建的发行版做得已经足够好。然而，最终，假如我们想要更多地超出 module-starter 程序的定制功能，我们还是有很多种方法处理这些。

如果我们真的喜欢 Module::Starter 模块，就需要做一点点修改，我们可以使用插件或者创建自己的插件，定制并且改变该模块的工作方式。在 CPAN 上已经有好几个这样的模块，Module::Starter::Plugin 模块的文档详细展示了如何创建我们自己的插件。

更简单一点的方法，就是设置我们喜欢的一些模板，当每次需要创建一个新的发行版时，仅仅处理它们。我们可以加入我们想要的任何文件。这就很接近 Distribution::Cooker 模块的形式。我们只是使用 Template Toolkit 来创建我们想要的模板，我们甚至可以使用 module-starter 的输出来启动初始模板。当我们已经准备好生成一个新的发行版时，我们就运行一个叫 dist_cooker 的程序。

对于真正复杂的模块创建，可以使用 Dist::Zilla 模块。这个模块不但可以自动创建发行版，而且在我们修改了发行版中的文件之后，它还知道如何更新发行包。如果我们想改变版权信息、作者简介或者其他类似的信息，Dist::Zilla 模块完全能够处理这些，而不必重新创建。

或者，我们可以开发自己的发行版生成器，因为有许多人似乎已经在这么做了^{注2}。

12.3 在你的发行版内部

我们已经创建了 Animal 发行版，而且它包含一个发行版的框架。现在它什么事情也做不了，但这确实是一个运行良好的完整发行版。通过运行 Build.PL 创建构建脚本：

```
% perl Build.PL
Checking whether your kit is complete...
Looks good
Checking prerequisites...
Looks good

Creating new 'Build' script for 'Animal' version '0.01'
```

注 2：而且，某个已经创建他自己的发行版生成器的作者，也在读上面的这段文字。

输出的第一行显示 Build.PL 脚本检查发行版，确保它需要的所有文件是完整的。每个发行版将在 MANIFEST 文件记录检查的结果。因为我们还没有加入任何文件，所以不会产生任何问题。当我们把发行版提供给其他人时，MANIFEST 文件将帮助人们发现我们是否已经提供全部所需的信息。

输出的下一个部分显示 Build.PL 脚本检查发行版所依赖的先决条件(即所依赖的模块)，我们后续将更仔细地查看 Build.PL 文件，再介绍如何指定其他所需要的模块，使代码能够正常工作。

一旦 Build.PL 文件完成了检查，它将创建一个程序，其中包含 Perl 的设置、模块的路径和其他一些信息。我们已经准备好生成新发行版的全部信息。先按照如下方式构建它：

```
% ./Build  
Copying lib/Animal.pm -> blib/lib/Animal.pm  
Manifying blib/lib/Animal.pm -> blib/libdoc/Animal.3
```

当运行构建脚本之后，发生了两件事：首先，它从 lib 目录将模块文件复制到构建库 blib 目录中，这是构建系统用来保存所有准备安装文件的位置。其次，Module::Build 模块将 Animal 模块内置的文档转换成 UNIX 系统中类似的手册页 (manpage)，然后放置到 blib/libdoc 目录中。

在构建发行版后，可以测试它。这是最常运行的命令。我们将对模块中的 lib/Animal.pm 文件做一些改动，然后运行测试，查看我们的修改造成了多糟糕的结果。module-starter 程序在 t 目录中创建了若干测试存根，在什么都没有改动之前，所有测试都是应当通过的：

```
% ./Build test  
t/00-load.t ..... ok  
t/boilerplate.t ... ok  
t/pod-coverage.t .. skipped: Test::Pod::Coverage 1.08 required for testing POD coverage  
t/pod.t ..... ok  
All tests successful.  
Files=4, Tests=5, 0 wallclock secs ( ... )  
Result: PASS
```

想了解测试的更多相关细节，请参照第 14 章中关于 Perl 的测试框架部分。

在发布我们的发行版之前，我们可以先运行安装测试命令的 disttest 参数。这个参数和其他参数有一点不同。在运行 disttest 参数时，Build 会为它即将创建的压缩包文件创建一个子目录，将 MANIFEST 中列出的所有文件复制进去，切换到子目录的位置，然后再次运行测试。这个测试用来检查我们即将放入压缩包和发布的内容已经包含测试所需的所有信息。

```
% ./Build disttest  
Creating Makefile.PL
```

```
Added to MANIFEST: Makefile.PL
Creating META.yml
Added to MANIFEST: META.yml
Creating Animal-0.01
/usr/local/perl5/perl-5.10.0/bin/perl Build.PL
Checking whether your kit is complete...
Looks good

Checking prerequisites...
Looks good

Creating new 'Build' script for 'Animal' version '0.01'
/usr/local/perl5/perl-5.10.0/bin/perl Build
Copying lib/Animal.pm -> blib/lib/Animal.pm
Manifying blib/lib/Animal.pm -> blib/libdoc/Animal.3
/usr/local/perl5/perl-5.10.0/bin/perl Build test
t/00-load.t ..... ok
t/pod-coverage.t .. skipped: Test::Pod::Coverage 1.08 required for testing POD coverage
t/pod.t ..... ok
All tests successful.
Files=3, Tests=2, 1 wallclock secs ( ... )
Result: PASS
```

当我们真正准备好发布发行版时，执行 dist 参数，MANIFEST 文件中列出的所有文件都将重新被组织：

```
% ./Build dist
Creating Makefile.PL
Deleting META.yml
Creating META.yml
Deleting Animal-0.01
Creating Animal-0.01
Creating Animal-0.01.tar.gz
Deleting Animal-0.01
```

现在，我们有一个名为 Animal-0.01.tar.gz 的压缩包，我们可以把它发给我们的朋友和家人，或者上传到 CPAN，第 21 章将介绍关于上传至 CPAN 的内容。

12.3.1 META 文件

build dist 命令的输出中有一行显示 Creating META.yml 文件，并且根据构建工具的版本，会显示 Creating META.json。这些由构造文件产生且显示为一种与语言无关的文本格式的特殊文件，如第 6 章介绍的 YAML 和 JSON，包含一些概述性的信息。这样，CPAN 客户端可以加载这些信息然后判断它需要做什么事情。客户端尤其关注文件中包含 `_requires` 字段列出的信息，这些信息可以帮助客户端在运行之前检查安装环境是否满足条件。在下面的 META.yml 示例文件中，客户端将在运行 Build.PL 之前，先检查系统是否已经安装了 Module::Build 模块的 0.38 版本（或者更新的版本）。

```
---
abstract: 'The great new Animal!'
author:
- 'Willie Gilligan <gilligan@island.example.com>'
```

```
build_requires:
  Test::More: 0
configure_requires:
  Module::Build: 0.38
dynamic_config: 1
generated_by: 'Module::Build version 0.38, CPAN::Meta::Converter version 2.112150'
license: perl
meta-spec:
  url: http://module-build.sourceforge.net/META-spec-v1.4.html
  version: 1.4
name: Animal
provides:
  Animal:
    file: lib/Animal.pm
    version: 0.01
  Horse:
    file: lib/Horse.pm
    version: 0.01
requires:
  perl: 5.006
resources:
  license: http://dev.perl.org/licenses/
  version: 0.01
```

我们也可以将 Makefile.PL 文件中的 META_MERGE 键或者 Build.PL 文件中的 meta_merge 键添加到这个文件中。要知道我们能做什么，可以检查 META 的规范，该规范将列出 meta spec 中的键。

12.3.2 添加额外的模块

最后，我们想要将另一个模块文件添加到发行版中，我们将在后续章节中使用我们创建的面向对象的模块来实现这一需求。

如果一开始我们就知道要把多个模块文件放入一个发行版中，如下所示，我们可以在初始运行 module-starter 时，在--module=后指定用逗号分割的参数列表：

```
% module-starter --module=Animal,Cow,Horse,Mouse
```

如果有可能，我们在创建初始发行版之后，还想要往发行版中添加新的模块文件。使用 Module::Starter::AddModule 模块能够满足这类需求，如下所示，但我们需要自己安装这个模块，并在 module-starter 程序的配置文件中作为一个插件添加：

```
author: Willie Gilligan
email: gilligan@island.example.com
builder: Module::Build
verbose: 1
plugins: Module::Starter::AddModule
```

在 Animal 目录中再次运行 module-starter 程序。我们使用--dist 参数加句号选项来指定在当前工作目录下运行安装包。得到的输出有一些杂乱，但是有两行显示我们已经创建了新的 Sheep 模块：

```
% module-starter --module=Sheep --dist=.
Found .. Use --force if you want to stomp on it.
```

```
Skipped lib/Animal.pm
Skipped lib/Cow.pm
Skipped lib/Horse.pm
Skipped lib/Mouse.pm
Created lib/Sheep.pm
Skipped t/pod-coverage.t
Skipped t/pod.t
Skipped t/manifest.t
Skipped t-boilerplate.t
Skipped t/oo-load.t
Created ./ignore.txt
Skipped ./Build.PL
Skipped ./Changes
Skipped ./README
Regenerating MANIFEST
Created MYMETA.yml and MYMETA.json
Creating new 'Build' script for 'Animal' version '0.01'
File 'MANIFEST.SKIP' does not exist: Creating a temporary 'MANIFEST.SKIP'
Added to MANIFEST: lib/Sheep.pm
Created starter directories and files
```

如果我们的发行版不在当前工作目录中，那么可以通过名称指定发行版所在的目录。例如，如果我们突然意识到忘记创建 Sheep 模块的目录，我们可以按照如下方式创建目录然后添加到刚创建的发行版中：

```
% module-starter --module=Animal,Cow,Horse,Mouse
% module-starter --module=Sheep --dist=Animal
```

如下所示，如果我们当前的位置已经在发行版的目录中，就可以使用.（点号）作为发行版的当前目录位置：

```
% module-starter --module=Sheep --dist=.
```

12.4 模块内部

既然我们已经构造了一个模块的框架，我们就查看模块的内部，检查一下已经为我们创建好了什么样的工具。如下为 lib/Animal.pm 文件的大部分内容，然而，我们已经去掉了一些令人生厌的重复代码的模板，以节省本书的篇幅：

```
package Animal;

use 5.006;
use strict;
use warnings;

=head1 NAME

Animal - The great new Animal!

=head1 VERSION

Version 0.01
```

```
=cut  
our $VERSION = '0.01';
```

=head1 SYNOPSIS

Quick summary of what the module does.

Perhaps a little code snippet.

```
use Animal;  
  
my $foo = Animal->new();  
...
```

=head1 EXPORT

A list of functions that can be exported. You can delete this section if you don't export anything, such as for a purely object-oriented module.

=head1 SUBROUTINES/METHODS

=head2 function1

=cut

```
sub function1 {  
}
```

=head2 function2

=cut

```
sub function2 {  
}
```

=head1 AUTHOR

Willie Gilligan, << gilligan at island.example.com >>

=head1 BUGS

...

=head1 SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc Animal
```

...

=head1 LICENSE AND COPYRIGHT

Copyright 2012 Willie Gilligan.

=cut

1; # End of Animal

我们并没有逐行分析上面代码中的内容，而是抛弃了非代码的部分。Perl 有一个嵌入式的文档格式称为 Pod，它是 Plain ol' Documentation 的简称。可以在代码部分之间放置 Pod，这样文件就是一些代码接着一些文档，再接着一些代码的格式。

12.5 老式文档

本节将对 Pod 做一个扼要介绍，虽然本书中还有许多没有讲述到的地方，但我们所介绍的内容是最通用的部分。我们可以在 perlpod 和 perlpodspect 两篇文档中参考 Pod 的文本格式规范，我们可以按照这个顺序来检查它们。

当 Perl 解释器期望遇到新语句的起始部分时，却发现语句行以“=”（等号）开始，例如=head1，Perl 解释器就将切换到 Pod 解析模式。为了编译代码，Perl 解释器将忽略所有的这类文档（就像大部分程序员一样）。perldoc 程序将会做相反的事情，它将忽略代码部分，只解析 Pod 部分，然后输出结果。到目前为止，我们已经知道如何使用 perldoc 来阅读已经安装模块的文档，但是，如下所示，我们同样也能阅读模块文件的 Pod 文档部分：

```
% perldoc lib/Animal.pm
```

默认情况下，perldoc 使用 nroff 命令（或它的一个变体），这样得到的输出看起来是沉闷的黑色和白色，但在终端中可以是彩色：

```
Animal(3) User Contributed Perl Documentation Animal(3)
```

NAME

Animal - The great new Animal!

VERSION

Version 0.01

SYNOPSIS

Quick summary of what the module does.

Perhaps a little code snippet.

```
use Animal;
```

```
my $foo = Animal->new();
```

我们还可以生成其他格式。pod2html 程序可以将 Pod 转换成 HTML 格式：

```
% pod2html lib/Animal.pm
<?xml version="1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
<title>Animal - The great new Animal!</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<link rev="made" href="mailto:gilligan@example.com" />
</head>

<body style="background-color: white">
```

12.5.1 段落的 Pod 命令

段落的 Pod 命令用于分隔段落。`=headn` 指定一个标题。`=head1` 是一级标题，`=head2` 是二级标题，依次类推：

```
=head1 NAME

=head1 DESCRIPTION

=head2 Functions
```

如下所示，当我们需要返回代码模式时，就以`=cut` 语句结束：

```
=head1 NAME

=head1 DESCRIPTION

=head2 Functions

=cut
```

如下所示，要创建列表，可以以`=over n` 开始，然后每个列表元素使用`=item` 来标记，最后以`=back` 标记列表结束：

```
=over 4

=item 1. Gilligan

=item 2. Skipper

=item 3. Ginger
```

在`=item` 之后，我们使用一些标志表示列表的类型。如果是数字类型，像数字 1，就创建一个数字列表。如下所示，如果使用“*”号，就会得到一个项目符号列表：

```
=over 4

=item * Gilligan

=item * Skipper

=item * Ginger

=back
```

12.5.2 Pod 段落

要在文档中加入文本，可以直接添加。完全不需要以任何方式进行标记。在下面的 SYNOPSIS 标题后，有两个段落：

```
=head1 SYNOPSIS
```

```
Quick summary of what the module does.
```

```
Perhaps a little code snippet.
```

Pod 格式化程序可以对这些段落进行重新换行。如果我们不希望文字换行，我们可以使用逐字的段落。如下所示，任何以空格开始的段落将不会换行，所以这类段落格式通常用于表述代码：

```
=head1 SYNOPSIS
```

```
Quick summary of what the module does.
```

```
Perhaps a little code snippet.
```

```
use Animal;
```

```
my $foo = Animal->new();
```

12.5.3 Pod 格式标记

在一个普通的段落内，或在一些命令段落中，我们可以使用格式标记自定义一些文本格式，这也叫做内部序列。每一种格式标记都以一个大写字母开始，并且用“<”和“>”括住所需的内容。例如，我们想定义斜体，我们就用 I。如下为这些格式标记：

- B<粗体文本>;
- C<代码文本>;
- E<实体名称>;
- I<斜体文本>;
- L<链接文本>。

如果格式标记中的文本有尖括号，我们就用双尖括号作为分隔符。Pod 解析器有足够的智能来找到最右边的结束序列。例如，我们得这样写：C<<\$a <=> \$b>>。有意思的是，在本书的 Pod 源代码中，我们不得不用三个连续的尖括号来表示两个连续的尖括号作为分隔符：C<<< C<<\$a <=> \$b>>>>>。等等，要这么做我们不得不……

如果我们需要特殊字符，可以用 E<名称>的格式指定，这种格式能够识别 HTML 实体名称和编号，例如 E<acute>、E<lt>和 E<0x0414>。然而，Pod 解析器能够处理 UTF-8，因此我们能够在声明编码后直接转入这些字符：

```
=encoding utf8
```

```
Gilligan tried to download Björk Guðmundsdóttir's latest album,  
but the Professor's Internet connection was down. The Professor  
pointed out that Gilligan should just say Björk.
```

12.5.4 检查 Pod 格式

一旦在程序或者模块中添加了 Pod，就可以使用 podchecker 程序来检查它的语法是否正确：

```
% podchecker lib/Animal.pm
*** WARNING: =head4 without preceding higher level at line 45 in file lib/Animal.pm
*** ERROR: unterminated L<...> at line 82 in file lib/Animal.pm
*** ERROR: =over on line 74 without closing =back (at head1) at line 93 in
file lib/Animal.pm
*** WARNING: empty section in previous paragraph at line 96 in file lib/Animal.pm
lib/Animal.pm has 2 pod syntax errors.
```

我们通常不会自己去做，因为我们可以使用一个测试文件自动检查，这些内容将在第 14 章中详细介绍。

12.6 模块中的代码

如下所示，当从 Animal.pm 模块中移除掉 Pod 后，就只剩下一点代码了：

```
package Animal;

use 5.006;
use strict;
use warnings;

our $VERSION = '0.01';

sub function1 {
}

sub function2 {
}

1; # End of Animal
```

我们在第 11 章介绍过包的语句，而且对于 strict 和 warnings 早已很熟悉。

按照惯例，Perl 模块使用包变量 \$VERSION 来声明版本号，使用 our 按照如下方式声明代码：

```
our $VERSION = '0.01';
```

版本号是一个字符串，这看起来有点古怪，因为我们通常认为版本号是数字，因为版本号由数字组成的。第 21 章将介绍\$VERSION 对于 PAUSE 索引和 CPAN 客户端的重要性，因此我们需要小心地处理它的值。例如，如果从 1.9 版本开始，并且下一个版本用 1.10，其实就降低了版本“数字”，这是因为 Perl 工具链将以数字的方式比较它们（因此，1.10 就小于 1.9）。当我们在 *Learning Perl* 中介绍 sort 内置函数的时候，也遇到同样的问题。

我们可以使用前导字母 v 以“版本字符串”的形式声明版本。我们使用“.”（点）来分

隔主版本号和次版本号。当我们比较这些时，先以数字的形式比较第一个数字，然后第二个数字，依次类推：

```
use v5.10; # v-strings are unreliable before v5.10;
our $VERSION = v0.1;
our $VERSION = v1.2.3;
```



注意

想了解关于版本的更多信息，请查询 Version 模块相关的文档。

module-starter 程序添加了两个存根函数，function1 和 function2。这并不是想要我们记住这两个子例程或者它们的名称。

模块最后的语句是“1;”。这不必是这个特别的值，但它必须是一个真值。当我们 require 或 use 模块时，如果文件一个返回值为真，Perl 解释器知道它成功加载或者编译了这个模块（编译错误的返回值为假）^{注3}。

以上就是基本模块的建立流程。在本书后续的章节中，我们将其他模块的特性添加到这个基本模块中。

12.7 模块构建的总结

本章介绍了很多内容。因为有太多的步骤，所以我们提供使用基于 Module::Build 模块和基于 ExtUtils::Makemaker 模块构建发行版的步骤概述。

12.7.1 创建基于 Module::Build 模块的发行版

创建初始发行版：

```
% module-starter --mb --name="Animal"
```

运行 Build.PL 来创建 Build 脚本：

```
% perl Build.PL
```

通过运行 Build 脚本来构建发行版：

```
% ./Build
```

我们做任何关于 test 参数的操作之前，确保所有测试都是可以通过的：

```
% ./Build test
```

用 disttest 参数确保测试依旧可以通过：

```
% ./Build disttest
```

注 3：有些人故意使用类似“false”的字符串，在返回的真值上开玩笑。

用 dist 参数创建发行版:

```
% ./Build dist
```

12.7.2 创建 ExtUtils::Makemaker 发行版

尽管本章一直都在使用基于 Module::Build 模块创建的发行版，但我们还想使用 ExtUtils::Makemaker 模块创建一个发行版，如果我们使用 Makefile.PL 文件，我们将遵循相同的流程并且发生相同的事情。

使用 Module::Starter 模块来创建发行版:

```
% module-starter --builder="ExtUtils::Makemaker" --name="Animal"
```

运行 Makefile.PL 来创建 Makefile 文件:

```
% perl Makefile.PL
```

通过运行 make 命令构建发行版:

```
% make
```

使用 test 参数，以确保在进行任何改动之前，所有测试都通过:

```
% make test
```

使用 disttest 参数确保测试依旧可以通过:

```
% make disttest
```

使用 dist 参数来创建发行版:

```
% make dist
```

12.8 练习

可以在附录中的“第 12 章答案”部分找到这些练习的答案。

1. [20 分钟] 使用 Module::Starter 模块，在命令行中运行 module-starter 命令，来创建你自己的 Animal 发行版，构建发行版并且运行测试。如果在此期间你没有做任何改动，所有测试都将通过。

故意在 Animal.pm 文件中设置一些语法错误，然后查看当你的模块中有一个错误时将发生的事情。重新运行测试，测试此时将无法通过。不用担心你将任何事情搞糟，因为你可以重新运行 module-starter 命令。

2. [20 分钟] 在 Module::Starter 模块配置文件中设定你的名字和电子邮件地址，然后重做练习 1，替换 Animal 发行版。
3. [20 分钟] 下载并安装 Module::Starter::AddModule 模块，在你的 Module::Starter 配置文件中添加一个插件。然后将 Cow 模块添加到你的发行版中。

对象简介

面向对象编程（通常称为 OOP）通过将代码组织到能够命名和分割的不同事物中，能够使代码运行速度更快并且维护更容易。我们需要先了解面向对象编程的基础，以便为后续章节的内容做准备。

仅当在程序（包括所有扩展库和模块）的长度超过 N 行后 OOP 的益处才会显露出来。遗憾的是，没有人能就一个确切的 N 值是多少达成一致意见，不过对于 Perl 程序，可认为它是 1000 行左右的代码。如果一个程序只有一二百行代码，使用面向对象的编程方式就有点杀鸡用牛刀了。

就像引用，Perl 面向对象的架构是建立在大量已有的 Perl v5 代码已经在使用的基础上，因此我们必须确保新加入的面向对象特性不会破坏已有的语法。让人惊奇的是，使面向对象编程的实现变得完美，唯一需要增加的语法是方法调用，稍后我们将介绍这个内容。不过这就意味着需要先学习一下这个语法，然后我们将介绍这个语法的基本内容。



注意

如果你想了解关于 OOP 在 Perl 语言中更深入的细节，可以尝试阅读 Damian Conway 所编写的 *Object Oriented Perl* (Manning 出版社)，这本书讲述了面向对象思想在 Perl 中实现的原理和架构。

Perl 的对象架构是在包、子例程和引用的概念上建立的，所以如果你忽略了之前章节中介绍的这部分内容，就需要从头阅读。准备好了吗？让我们开始。

13.1 如果我们可以和动物对话……

显然，遇难者不能只靠椰子和菠萝活下来。幸运的是，在他们到达荒岛不久，一艘载

着随机动物的驳船也搁浅在荒岛上，于是这些遇难者开始修建农场和饲养动物了。

首先我们要用到上一章中创建的 Animal 发行版，如下所示，我们使用 module-starter 程序添加一些特定的动物：

```
% module-starter --module=Cow,Horse,Sheep
```

现在我们在 lib 目录中已有三个文件：Cow.pm、Horse.pm 和 Sheep.pm。在每个文件中，我们将添加一个相应动物的特有 speak 子例程。尽管我们是一点一点地完善这些文件，但我们可以直接跳到本章末尾，查看经过我们全部修改之后每个文件的完整代码。

我们听一会儿这些动物的叫声，来确定它们的说话方式。如下所示，在 module-starter 创建的存根模板中，用我们自己的 speak 子例程替换一些已经有的子例程模板：

在 Cow.pm 中：

```
sub speak {  
    print "a Cow goes moooo!\n";  
}
```

在 Horse.pm 中：

```
sub speak {  
    print "a Horse goes neigh!\n";  
}
```

在 Sheep.pm 中：

```
sub speak {  
    print "a Sheep goes baaaah!\n";  
}
```

我们在 scripts/pasture 目录中创建一个脚本（并将这个文件加到构建文件中）。我们加载所有新模块，然后在每一个类中调用 speak 子例程：

```
use Cow;  
use Horse;  
use Sheep;  
  
Cow::speak;  
Horse::speak;  
Sheep::speak;
```

因为我们还没有安装这些模块，所以一旦运行这个脚本就会报错，因为 Perl 还不知道去哪里找这些模块：

```
% perl scripts/pasture  
Can't locate Cow.pm in @INC (...)
```

除非我们决定安装这些模块，否则我们必须通过第 2 章介绍的方法来告诉 Perl 如何使用我们放在 lib 文件夹中的模块。

```
% perl -Ilib scripts/pasture
```

现在我们得到如下所示的正确输出：

```
a Cow goes moooo!  
a Horse goes neigh!  
a Sheep goes baaaah!
```

这没有什么让人兴奋的：只是从不同的包中，使用完整的包名简单地调用子例程而已。如下所示，现在我们使用自己定义的子例程创建一个完整的牧场。

```
use Cow;  
use Horse;  
use Sheep;  
  
my @pasture = qw(Cow Cow Horse Sheep Sheep);  
foreach my $beast (@pasture) {  
    no strict 'refs';  
    &{$beast."::speak"};           # Symbolic coderef  
}
```

现在我们有许多动物开始嚎叫：

```
a Cow goes moooo!  
a Cow goes moooo!  
a Horse goes neigh!  
a Sheep goes baaaah!  
a Sheep goes baaaah!
```

在循环体中，符号代码引用的解引用代码实在太难看了。我们在 no strict 'refs' 的模式下统计，肯定在大一点的程序中不建议这样使用。但这里为什么必须这么实现？因为包名称似乎不能与我们想要在这个包中调用的子例程的名称分离。

或者是？



注意

尽管本书的所有示例都应当是有效的 Perl 代码，但在本章中，为了更便于理解，一些示例将打破使用 strict 模式的限制。不过在本章的最后，我们将再次展示使用 strict 模式的编码方式。

13.2 介绍方法的调用箭头

类是一组拥有相似行为和特征的事物。目前为止，我们说 Class->method 调用 Class 包中的 method（方法）子例程。这个方法是面向对象风格的子例程，因此从现在开始我们称其为“方法”^{#1}。其实这并不完全准确，但我们将一步一个脚印地学习，就像如下的使用方式：

```
use Cow;  
use Horse;  
use Sheep;
```

注 1：在 Perl 中，方法和子例程没有什么区别。它们都可以获取 @_ 中的参数列表，而且我们必须确保我们所做的事情是正确的。

```
Cow->speak;  
Horse->speak;  
Sheep->speak;
```

如下所示，得到的输出和之前的输出一致：

```
a Cow goes mooool!  
a Horse goes neigh!  
a Sheep goes baaaah!
```

这不是开玩笑，我们得到了相同数量的字符，但都是常量，而不是变量。然而，现在这些部分可以分开了。我们可以将类名放入变量中，然后按照如下方式使用：

```
my $beast = 'Cow';  
$beast->speak; # invokes Cow->speak
```

既然包名和子例程名分开了，就可以使用一个变量作为包名。这次，即使启用 `use strict 'refs'` 也可以让代码正确执行。

如下所示，在之前的牧场示例中使用方法调用箭头的方式：

```
use Cow;  
use Horse;  
use Sheep;  
  
my @pasture = qw(Cow Cow Horse Sheep Sheep);  
foreach my $beast (@pasture) {  
    $beast->speak;  
}
```

看，现在所有动物都在说话了，而且非常安全地实现了，并没有使用符号代码引用。

但是看看这些通用的代码。每个 `speak` 方法都有一个相似的结构：`print` 操作符和包含普通文本的字符串，只有两个单词不同而已。OOP 的一个核心原则就是让代码最少：如果只写一次，就会节省时间。如果只测试并且调试一次，就会节省更多时间。

既然我们理解方法调用箭头实际上做的事情，我们就知道做相同事情的这个更容易的方法。

13.3 方法调用的额外参数

方法调用形式：

```
Class->method(@args)
```

现在按照如下方式调用 `Class::method` 子例程：

```
Class::method('Class', @args);
```

(如果它没有找到相应的方法，就将启用继承机制，不过稍后我们再介绍这部分内容。) 这意味着我们将类名作为传递给所调用方法的第一个参数，或者唯一的参数。如果没有提供任何参数，我们可以按照如下方法重写 `Sheep` 的 `speaking` 方法：

```
sub speak { # In lib/Sheep.pm
    my $class = shift;
    print "a $class goes baaaah!\n";
}
```

如下所示，另外两个动物的输出也类似，我们在 Cow.pm 中做了相同的改动：

```
sub speak { # In lib/Cow.pm
    my $class = shift;
    print "a $class goes mooool!\n";
}
```

在 Horse.pm 中的改动也一样：

```
sub speak { # In lib/Horse.pm
    my $class = shift;
    print "a $class goes neigh!\n";
}
```

在以上的每个示例中，变量\$class 为方法提供合适的数据。但还是有太多相似的结构。我们还能进一步提取出更通用的部分吗？当然能，通过在同一个类中调用另外一个方法实现。

13.4 调用第二个方法进一步简化

我们可以在 speak 方法中调用一个叫做 sound 的帮助方法。该方法为 sound 方法本身提供一个常量文本。如下所示，在 Cow.pm 模块中，调用一个 sound 子例程返回 cow 发出的声音字符串：

```
# In lib/Cow.pm
sub sound { 'moooo' }
sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
}
```

现在，当调用 Cow->speak 时，是在调用 Cow 中的\$class 指向的 speak 方法。这里按照顺序，先选择 Cow->sound 方法，得到返回值 mooool。对于 Horse.pm 有什么不同吗？其实没什么不同：

```
# In lib/Horse.pm
sub sound { 'neigh' }
sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
}
```

只是改变了包的名称和特定的 sound 声音字符串。那么，我们能在 Cow 和 Horse 之间共享 speak 方法的定义吗？当然能，使用继承就可以！对于 Sheep.pm 也一样：

```
# In lib/Sheep.pm
sub sound { 'baaaah' }
sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
}
```

现在，我们定义一个叫做 Animal 的通用方法包，其中定义了通用的 speak 方法和 sound 方法的占位符。

```
# In lib/Animal.pm
sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
}

sub sound {
    die 'You have to define sound() in a subclass'
}
```

尽管我们不想让大家调用 Animal 类中的 sound 方法，因为这会使运行失败，但是我们想提醒大家需要自己定义 sound 方法。

然后，对于每个动物，我们可以说它们继承自 Animal 类，都包含特定于动物的 sound 方法。如下所示，在每个动物的模块中，添加一行包含@ISA 的语句表明这种关系（后续将介绍更详细信息）：

```
use Animal;
our @ISA = qw(Animal);
sub sound { "moooo" }
```

现在调用 Cow->speak 将会发生什么？

首先，Perl 解释器将构建参数列表。在这里，仅仅对于 Cow 构建。然后，Perl 解释器将查找 Cow::speak 方法。没找到，Perl 解释器将检查继承数组@Cow::ISA，它发现该数组中只有一个名字：Animal。

Perl 解释器下一步将在 Animal 类中查找 speak 方法，找到了 Animal::speak 方法。于是 Perl 解释器传入已经暂存的参数列表并且调用该方法，如下所示，正如我们之前曾经使用过的：

```
Animal::speak('Cow');
```

在 Animal::speak 方法内部，\$class 变为 Cow，作为第一个参数被移除。如下所示，在 Animal::speak 方法内部以下面的语句开始：

```
print "a $class goes ", $class->sound, "!\n";
```

我们替换成'Cow'，这正是 \$class 的值：

```
# but $class is Cow, so...
print 'a Cow goes ', Cow->sound, "!\n";
# which invokes Cow->sound, returning 'moooo', so
print 'a Cow goes ', 'moooo', "!\n";
```

于是我们得到了期望的输出。

13.5 关于@ISA 的几个注意事项

神奇变量@ISA（发音为“is a”，而不是“ice-uh”）声明 Cow 是一个（is a）Animal。注意，这是一个数组，不单单只是一个值，因为在一些罕见的场合中，这个变量使我们在多个父类中查找缺失的方法变得有意义。我们随后将详细介绍这部分内容。

如果 Animal 类没有 speak 方法，但也有一个@ISA 数组，Perl 解释器就也会在@ISA 数组中查找。在@ISA 中查找都是递归的，深度优先，并且从左到右进行。一般来说，每个@ISA 仅拥有一个元素（多个元素意味着多重继承和多重混乱），这样我们就得到一个清晰的继承树。



注意

还可以通过 UNIVERSAL 和 AUTOLOAD 继承，想了解完整细节，请查阅 perlobj 文档或者 Programming Perl 获得更完整的内容。

当开启 strict 模式时，我们就会得到关于@ISA 变量的警告信息，因为该变量既不是一个包含显式包名的变量，也不是一个词法（用 my 或 state 声明）变量。我们不能将它当作一个词法变量：但它只属于根据继承机制查询到的那个包。

处理@ISA 的声明和设定有两个直接的方法。如下所示，最简单的方法是加上包名：

```
@Cow::ISA = qw(Animal);
```

如下所示，也可以允许它作为隐式命名的包变量：

```
package Cow;
use vars qw(@ISA);
@ISA = qw(Animal);
```

如下所示，也可以使用 our 关键字声明来简写上面的内容：

```
package Cow;
our @ISA = qw(Animal);
```

然而，如果我们考虑到我们的代码可能会被坚持使用 Perl v5.5 或更早版本的人运行，我们就应当避免使用 our。当然，我们鼓励所有人使用最近十年发布的 Perl 版本^{注2}。

如下所示，要在模块中使用基类，同样，不但要通过@ISA 声明继承关系，还要加载 Animal 模块：

```
package Cow;
use Animal;
our @ISA = qw(Animal);
```

要一次性完成以上全部这些步骤，可以用 use parent 编译提示符：

注 2：Perl v5.6 的第一个发行版于 2000 年 3 月 22 日发布的。更多信息，请参看 perlhist 文档。

```
use v5.10.1;
package Cow;
use parent qw(Animal);
```

这种写法相当简洁。此外，use parent 语句的好处在于它是在编译时执行的，可以提前检查出@ISA 在运行时设置的一些潜在错误，与其他一些解决方案类似。



注意

如果使用 Perl v5.10.1 以前的版本，我们可以用 use base 替代，或者自行安装 parent 模块。我们应当将它声明为必要条件，因为之前的 Perl 语言的标准库中并没有提供。

13.6 方法重写

我们添加一种几乎听不到的 mouse 的声音，如下所示，首先创建 Mouse 包：

```
% module-starter --module=Mouse --dist=.
```

一旦拥有 Mouse.pm 文件，就在文件中添加了它的 sound 子例程，这与之前介绍的从 Animal 继承的其他文件类似。如下所示，在 speak 子例程中做了一点小小改动：

```
package Mouse;
use parent qw(Animal);

sub sound { 'squeak' }

sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
    print "[but you can barely hear it!]\n";
}
```

现在，创建一个 script/mouse 程序来加载这个模块并调用 speak 方法：

```
use Mouse;

Mouse->speak;
```

当执行后，就会看到：

```
a Mouse goes squeak!
[but you can barely hear it!]
```

此时，Mouse 有了自己的 speak 子例程，因此执行 Mouse->speak 语句不会立即调用 Animal->speak。这就是重写。当我们需要一个特定版本的子例程时，就可以在派生类 (Mouse) 中重写方法。我们甚至不需要初始化@mouse::ISA 来说明 Mouse 属于 Animal 类，因为 speak 所需要方法的全部内容都完全在 Mouse 中定义。

我们从 Animal->speak 中复制一些代码；这可能会引发一个维护性问题。例如，如果有人认为在 Animal 类输出的内容中单词 goes 是一种 bug。现在，类的维护人员就会把 goes

更改为 says。但老鼠还按以前的方式“说话”，这就意味着代码依然存在 bug。问题的根源在于我们复制并粘贴了代码，这实在是个愚蠢的决定。我们应当使用继承的方式重用代码，而不是用复制和粘贴。

如何避免这类问题呢？是否可以用某种方式，不但让 Mouse 可以做任何其他从 Animal 类派生出的程序能做的事情，而且增加一些额外的内容？当然可以！

在第一次尝试中，我们直接调用 Animal::speak 方法。如下所示，在 Mouse.pm 中修改了 speak 方法：

```
package Mouse;
use parent qw(Animal);

sub sound { 'squeak' }

sub speak {
    my $class = shift;
    Animal::speak($class);      # MESSY!
    print "[but you can barely hear it!]\n";
}
```

注意，由于我们停止使用方法调用箭头，因此我们不得不包含\$class 参考（几乎可以确定就是 Mouse 的值）作为 Animal::speak 方法的第一个参数。

我们为什么停止使用箭头？原因是这样的，如果我们在此调用 Animal->speak，方法的第一个参数将是“Animal”，而不是“Mouse”，而且当调用 sound 子例程的时候，将无法使用正确的类为该对象选择合适的方法。

然而，直接调用 Animal::speak 会显得很杂乱。如果 Animal::speak 之前并不存在，而且该方法是从@Animal::ISA 中包含的其他类中继承的，将会怎么样呢？例如，创建一个叫做 LivingCreature 的模块：

```
% module-starter --dist=LivingCreature
```

在 LivingCreature.pm 中添加 speak 子例程：

```
package LivingCreature;
```

```
sub speak { ... }
```

也删除 Animal.pm 中的 speak 方法：

```
package Animal;
use parent qw(LivingCreature);
```

因为在 Mouse::speak 中没有使用方法调用箭头，又因为只是在调用一个普通子例程而没有使用继承的方法，所以就只有一次机会来调用正确的方法。我们将在 Animal 类中找这个方法，如果找不到，程序就终止。

Animal 类名现在已经捆绑到所选择的方法上。如果有人维护这段代码，为 Mouse 改变 @ISA 数组的内容，或者没有注意到 Animal 类中已经有了一个 speak 方法，一切就会变得一团糟。因此，这实在不是一个好主意。

13.7 开始从不同的地方查找

一个比较好的解决方案是使 Perl 在继承链上的不同位置进行查询：

```
package Mouse;
use parent qw(Animal);

sub sound { 'squeak' }

sub speak {
    my $class = shift;
    $class->Animal::speak(@_); # tell it where to start
    print "[but you can barely hear it!]\n";
}
```

尽管如此丑陋，但它可行。使用这种语法，开始在 Animal 中查找 speak 方法，如果没有立即找到，就在 Animal 的所有继承链上查找。第一个参数是\$class（因为再一次用了方法调用箭头），因此查询到的 speak 方法使用 Mouse 作为其第一个参数，并且最后返回调用 Mouse::sound 的细节上。

然而，这并不是最好的解决方案。我们仍然必须保持@ISA 和初始查询包的同步（改变其中一个必须考虑同时改变另一个）。更糟糕的是，如果 Mouse 在@ISA 中有多个项，我们就不必知道哪一个实际上定义 speak 方法。

那么，还有更好的办法吗？

13.8 使用 SUPER 的实现方法

在调用方法时，通过改变 Animal 类为 SUPER 类，可以立即自动搜索一遍所有的超类（@ISA 中列出的类）：

```
package Mouse;
use parent qw(Animal);

sub sound { 'squeak' }

sub speak {
    my $class = shift;
    $class->SUPER::speak;
    print "[but you can barely hear it!]\n";
}
```

因此，SUPER::speak 意味着应当在当前包的@ISA 中查找 speak 方法，如果找到多个，就调用第一个。这里，我们查看第一个并且唯一的父类：Animal，查找 Animal::speak，最后将“Mouse”作为唯一的参数传递给它。

13.9 要对 @_ 做些什么

在上一个示例中，对于 speak 方法还会有其他额外的参数（例如多少次，使用什么音调歌唱，等等）吗？这些参数将被 Mouse::speak 方法所忽略。如下所示，如果我们想将这些参数连续地传递给父类，我们可以将它们作为一个参数添加。

```
$class->SUPER::speak(@_);
```

这将调用父类的 speak 方法，包括参数列表中所有没有移除的参数。

哪个是正确的？这根据情况而定。如果添加一个类只是为了添加父类的行为，最好将参数不加改动地进行传递。如果我们想严格控制父类的行为，我们应当显式地确定参数列表，然后传递它们。

13.10 我们在哪里

目前为止，已经使用方法调用箭头的语法调用了一个类字面里的方法：

```
Class->method(@args);
```

如下所示，可以使用一个保存了类名的变量做同样的事情：

```
my $beast = 'Class';
$beast->method(@args);
```

在这些示例中，Perl 隐式地将类名放置于参数列表之前：

```
('Class', @args)
```

如果想用普通的子例程来做同样的事情，就不得不使用完全限定的包名，而且还要将类名加入参数列表中：

```
Class::method('Class', @args);
```

然而，只要我们将它作为方法调用，如果 Perl 解释器找不到 Class::method，它就会检查@Class::ISA（以递归方式）以定位真正拥有 method 的包，然后调用这个版本作为替代。

第 15 章介绍如何通过相关的属性来区分不同的动物，这叫实例变量。

13.11 牧场总结

下面是经过所有改动后放到牧场文件中的代码。

在 lib/Animal 文件中：

```
package Animal;

sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
}

sub sound {
    die 'You have to define sound() in a subclass'
}

1;
```

在 lib/Cow.pm 文件中：

```
package Cow;
use parent qw(Cow);

sub sound { 'neigh' }

1;
```

在 lib/Horse.pm 文件中：

```
package Horse;
use parent qw(Animal);

sub sound { 'neigh' }

1;
```

在 lib/Sheep.pm 文件中：

```
package Sheep;
use parent qw(Animal);

sub sound { 'baaaaah' }

1;
```

在 lib/Mouse.pm 文件中：

```
package Mouse;
use parent qw(Animal);

sub sound { 'squeak' }

sub speak {
    my $class = shift;
    $class->Animal::speak(@_); # tell it where to start
    print "[but you can barely hear it!]\n";
}

1;
```

13.12 练习

可以在附录中的“第 13 章答案”部分找到这些练习的答案。

1. [20 分钟] 创建 Animal、Cow、Horse、Sheep 和 Mouse 类。在发行版中运行测试目标文件，以确保所有内容都正确编译。(也就是说，你的 t/00load.t 测试通过)。修改你想改的内容，并使测试能够通过。
2. [20 分钟] 编写一个程序，使用户输入一个或多个牧场动物的名字。创建一个包含这些动物的牧场，并让所有动物都叫一次。
3. [40 分钟] 增加一个与 Animal 同样层级的 Person 类，它们都从一个叫做 Living Creature 的新类中继承。同时创建一个 speak 方法获取一个“说什么”的参数，如果没有提供参数，就返回到 sound (Person 的哼唱)，因为这个人不是 Doolittle 博士(一个能和动物交流的人)，所以就要确保动物不能说话(也就是说，不要让 animal 的 speak 方法带任何参数)。尽量不要重复任何代码，但要确保能捕获使用过程中任何可能的错误，例如忘记定义某一种动物的 sound。

在 scripts/perlson.pl 脚本中展示 Person 类，让人 (person) 说 “Hello, World!”。

第 14 章

测试简介

在第 13 章中，我们创建了一个新的 Perl 发行版，修改了一些模块，然后为发行版添加了一个程序。既然我们在开发伊始就有一个功能完整的文件包，那么我们能立即开始使用 Perl 的扩展测试框架。实际上，我们已经开始在这么做了。

现在，到了近距离了解已经在安装包中的和我们自己创建的测试用例的时候。当我们继续进行模块开发时，测试将引导我们在正确的道路上前行。

14.1 为什么需要测试

我们为什么在开发过程中就需要测试？简单来说，这样可以帮助我们快速发现问题，而且测试强制我们以非常小的代码块进行编码（因为这比较容易测试），这通常是非常好的编程实践。尽管我们可能认为还有许多额外的工作要做，但是其实这相当短视，因为当我们花费更少的时间进行调试时，我们才是最后的赢家。这是因为当问题出现之前，我们已经解决这些问题，而且测试能精确地告诉我们哪里出了错。

从另一方面说，如果我们少了什么东西，测试将会告诉我们，从心理上更容易修改代码。当我们和老板或同事谈论的时候，我们对于他们的质疑和问题会更有信心。测试会告诉我们，我们所编写代码的健康程度。

我们实际上从没有真正完成测试。甚至当使用模块时，我们也不应该放弃测试套件！除非代码使用神奇的“没有 bug 的模块”，否则用户将给我们发送错误报告。我们可以把每个测试报告转变成一个测试用例。每次修复完 bug 时，现有的测试可以防止代码回归到一个功能更少的版本——这就是回归测试名字的由来。



注意

如果我们为别人的代码提交了一个 bug，我们通常可以肯定地说，如果我们能发送给相应的维护人员一个针对这个 bug 的测试，他会很感激我们。甚至比给他一个修复错误的补丁还高兴！

然后总是需要考虑关于未来版本的问题。当我们想添加新特性时，就从添加测试开始。因为已有测试将确保代码是向前兼容的，所以我们就会自信地认为新版本不仅能做旧版本能做的所有事情，还能做更多。

正如第 21 章的介绍，CPAN 上的 Tester 能使用测试用例，来检查代码在我们所没有的系统和环境下的运行情况。

14.2 Perl 的测试流程

Perl 测试惯例是在 Perl 程序所在的文件夹中建立一个文件夹，我们称之为“测试文件”，或者有时只称之为“测试”。其中的每个程序文件运行后，就会判断测试是否通过，并且输出正确的提示，下一步能告诉发生了什么。

通用测试协议

Perl 用一种简单的方式记录哪些测试通过和哪些测试没有通过。没有人十分确定谁发明了它(Tim Bunce 和 Andreas König 两者之一)，但它作为通用测试协议(Test Anywhere Protocol, TAP) 开始流行起来。这个简单的文本协议起源于 Perl，但也已经被其他语言采用。

如果测试通过，输出 ok 和一个测试编码。

ok 1

我们可以给测试附加一个标签，使我们知道通过了什么：

ok 1 - The boat motor works

如果测试没有通过，输出为 not ok：

not ok 2 - The hull is intact

除了独立的测试集以外，我们想知道我们想运行的所有测试是否实际上都在运行中。如下所示，测试计划给出了我们将要运行的测试范围。计划可以在测试前提供：

```
1..3
ok 1 - The boat motor works
ok 2 - The gas tank is full
not ok 3 - The hull is intact
```

或者在测试后显示：

```
ok 1 - The boat motor works
ok 2 - The gas tank is full
not ok 3 - The hull is intact
1..3
```

在 Perl 的早期，程序员经常使用直接输出测试结果的方式：

```
print $motor_broken ? 'not ' : '', 'ok ', $test++, "\n";
```

当我们这么做的时候，就不得不自己跟踪测试的数量，至少直到 Test::Simple 模块自动完成这项工作。尽管几乎没有人用这个古老的模块，但它仍然是 Perl 核心模块的一部分：

```
use Test::Simple tests => 3;
```

```
use Minnow::Diagnostics;
```

```
ok( try_motor(), 'The boat motor works' );
ok( check_gas() eq 'Full', 'The gas tank is full' );
ok( check_hull(), 'The hull is intact' );
```

我们不再直接处理测试计划和输出的问题。`ok` 函数的第一个参数用于验证测试结果真假。如果为真，就输出 `ok`，否则输出 `not ok`。

当今的测试技术已经与当初大不一样，现在有了 Perl 主要的测试模块：`Test::More` 模块，从名字看它比 `Test::Simple` 功能更强大^{注1}。我们有很多更方便的子例程用于检查值和输出正确的 TAP。本节的其他内容将介绍这些子例程。



注意

我们可以在 `Test::Tutorial` 文档中了解关于测试的更多信息。

`Test::More` 模块能够与 `Test::Simple` 一样处理测试计划以供使用，它也有 `ok` 子例程来做同样的事情：

```
use Test::More tests => 1;
```

```
ok( try_motor(), 'The boat motor works' );
```

在以上示例中，我们明确声明只有一个测试，并且这是测试用具（`test harness`）期望我们的程序输出的测试报表数量。如果我们不知道测试的数量，我们可以在测试代码的末尾使用 `done_testing`。如果我们运行到该行，就可以有理由确定测试程序已经运行到末尾。

```
ok( try_motor(), 'The boat motor works' );
```

```
done_testing();
```

在 `Test::Simple` 模块的示例中，我们使用单个参数来判断我们想要测试内容的真实性。如下所示，我们仍然可以这样做：

```
ok( check_gas() eq 'Full', 'The gas tank is full' );
```

然而，`Test::More` 模块还有 `is` 子例程，该子例程会自动进行比较。我们告诉它我们所拥有的值、所期望的值、和测试标签。

```
is( check_gas(), 'Full', 'The gas tank is full' );
```

注 1：这两个模块来自于同一个发行版。

如果这个测试通过，我们就会得到相同的输出。然而，当测试失败时会更有趣，因为测试已经知道期望得到的结果。TAP 协议允许注释，可以按照如下方式输出：

```
1..3
ok 1 - The boat motor works
not ok 2 - The gas tank is full
#   Failed test 'The gas tank is full'
#   in /Users/Gilligan/test.pl at line 9.
#       got: 'Empty'
#       expected: 'Full'
not ok 3 - The hull is intact
# Looks like you failed 1 test of 3 run.
```

还有更多有趣的子例程。Isnt 的功能与 is 子例程相反。其中第二个参数不应当是我们期望得到的值。

```
isnt( check_hull(), 'Broken', 'The hull is intact' );
```

like 子例程可以使用一个正则表达式：

```
like( 'Mary Ann', qr/Mary[ -]Anne?/, 'Mary Ann is a passenger' );
```

甚至有 unlike 子例程，就是模式匹配中的不匹配：

```
unlike( 'Ginger', qr/Mary[ -]Anne?/, 'Ginger is a passenger' );
```

我们可以测试基本的数据结构：

```
is_deeply( \@this_array, \@that_array, 'The arrays are the same' );
```

就这些了，测试程序就是调用这一系列的子例程。对于那些 Test::More 模块不处理的内容，可以使用 CPAN 上以 Test::名称开始的模块处理，例如 Test::Class 模块、Test::File 模块，或者许多其他特定领域的模块。第 20 章介绍这些内容。

14.3 测试的艺术

好的测试模块也会在相应的文档中提供一些小示例。这是表达同样事情的另外一种方式，并且一些人可以更喜欢其中的一种示例。好的测试模块也会给用户足够的信心，证明我们的代码（和它们全部的依赖关系）在他们的系统上是可移植的。



注意

与文档相比，一些模块的使用更容易从测试用例中学习。所有好的测试用例应当重复出现在模块的文档中。

测试是一门艺术。人们已经编写和阅读过几十本关于如何测试的书（似乎大部分都被遗忘）。主要原因大概是忽略了这个原则：记住编程过程中曾经犯过（或从其他人哪里听说）的错误是很重要的，然后再次测试当前项目中没有测试过的地方。

在创建测试时，我们应尽量从模块用户的角度考虑问题，而不是从编写模块的作者角度。我们知道如何使用我们的模块，因为它是我们自己编写的模块，而且该模块能够

满足特定的需要。其他人可能会使该模块有不同的用途，他们将试图用各种不同的方式来使用它。我们也许已知道可以这样做：用户将会发现使用我们的模块的所有方式。在编写测试时，我们需要站在这个角度来思考。

我们需要测试代码运行中断的情况，以及代码正常工作的情况。需要测试边界和中间情况。还需要测试比边界条件多一次或者少一次的情况。一次只测试一个用例，然后一次性测试很多用例。如果某种情况下应当抛出异常，我们也要确保测试不会有不良的副作用：传递额外的或者多余的参数，或者没有传递足够的参数，搞混命名参数的大小写。简而言之，我们将一直尝试“打破”代码中的常规设置直到我们找不出任何其他“打破”的方法。

一个测试示例

假设我们想要测试 Perl 的 `sqrt` 函数，也就是计算平方根的函数。很明显，我们需要确保在输入一个完全平方数时，例如 0、1、49 或 100，返回正确的值。调用 `sqrt(0.25)` 应当得到的是 0.5。我们应该确保 `sqrt(7)` 的值再进行平方运算之后得到的值在 6.99999~7.00001 之间。



注意

记住，浮点数并不总是准确；通常都会有一个舍入误差。`Test::Number::Delta` 模块可以处理这些情况。

我们用代码来表述这些内容。在这部分测试代码中，如果我们给出合适的值，它就将正常工作。

```
use Test::More tests => 6;

is( sqrt( 0 ), 0, 'The square root of 0 is 0' );
is( sqrt( 1 ), 1, 'The square root of 1 is 1' );
is( sqrt( 49 ), 7, 'The square root of 49 is 7' );
is( sqrt(100), 10, 'The square root of 100 is 10' );

is( sqrt(0.25), 0.5, 'The square root of 0.25 is 0.5' );

my $product = sqrt(7) * sqrt(7);
ok( $product > 6.999 && $product < 7.001,
    "The product [$product] is around 7" );
```

这其实很无聊。乐趣在于“打破”代码中的常规设置。运行 `sqrt(-1)` 将会发生什么？这是一个完全有效的数学运算，但并不是 Perl 的 `sqrt` 版本能够应付的情况。一些程序员会故意或出于其他原因这样做，并且测试应当检查这些内容，可以用 `eval` 捕获得到的结果：

```
{
    $n = -1;
    eval { sqrt($n) };
    ok( $@, '$@ is set after sqrt(-1)' );
}
```

我们可以尝试用其他方法“打破”它的常规设置。如下所示，提供一个未定义的值：

```
eval { sqrt(undef) };
is( $@, '', '$@ is not set after sqrt(undef)' );
```

或者不提供值：

```
is( sqrt(0), 'sqrt() works on $_ (undefined) by default' );
```

尝试使用默认变量 `$_`：

```
$_ = 100;
is( sqrt(10), 'sqrt() works on $_ by default' );
```

对于非常大的数字，会发生什么呢？

```
is( sqrt(10**100), 10**50, 'sqrt() can handle a googol1' );
```

14.4 测试用具

测试程序将输出 TAP 协议的数据。我们在第 12 章创建的发行版在 `t` 文件夹中包含几个测试程序。当我们运行测试的时候，调用了一个叫做测试用具的东西，它将查找所有测试程序（以`.t`结尾的文件），逐个运行它们，捕获输出，并提供结果的一个总体摘要。测试用具内部有好几个部分来分别处理其中的每个内容，但在本书中将不再赘述。



注意

Test::Harness 模块会搜集测试脚本，并运行它们，最后汇总结果。但我们通常不会直接和它交互。

发行版中的测试用例只是测试程序的集合，与我们目前所展示的类似。

14.5 标准测试

当使用 `module-starter`（或者 `h2xs`）程序创建我们自己的发行版时，就自动生成了一些初始测试脚本以及一些其他文件。按照惯例，测试文件位于 `t` 目录并以 `.t` 扩展名结尾。

```
% module-starter --module=Animal
...
Created Animal/t
Created Animal/t/pod-coverage.t
Created Animal/t/pod.t
Created Animal/t/boilerplate.t
Created Animal/t/oo-load.t
...
```

当运行测试程序时，构建程序将运行它在 `t` 目录中所查找到的每个测试文件：

```
% perl Build.PL
...
% ./Build test
Copying lib/Animal.pm -> blib/lib/Animal.pm
t/00-load.....# Testing Animal 0.01, Perl 5.010000, /usr/bin/perl
t/00-load.....ok
t/boilerplate.....ok
t/manifest.....skipped
    all skipped: Author tests not required for installation
t/pod-coverage....skipped
    all skipped: Test::Pod::Coverage 1.08 required for testing POD coverage
t/pod.....ok
    all skipped: Test::Pod 1.22 required for testing POD coverage
All tests successful, 2 tests skipped.
Files=5, Tests=5, 0 wallclock secs ( 0.12 cusr + 0.04 csys = 0.16 CPU)
```

t/manifest.t 文件并没有运行，显示“不需要作者的测试”(Author tests not required)。模块的维护人员可能会对一些测试感兴趣，因此在发布模块前，会针对发行版的特定细节而编写一些测试。然而，当到达用户的时候，运行这些测试已经为时已晚。更糟糕的是，一旦出现问题，通常是一些与代码无关的原因。CPAN 客户端将拒绝安装此发行版和任何依赖于此发行版的其他发行版。因为这个为难的事，测试的作者将检查 RELEASE_TESTING 或 AUTOMATED_TESTING 这两个环境变量，使这些测试只在他们的系统上运行。作者在自己的运行环境中设置 RELEASE_TESTING 环境变量，而 CPAN 上的 Tester 将设置 AUTOMATED_TESTING 变量来进行自动测试。我们可以根据这些结果做出自己的判断。



注意

有些人把作者测试目录设置成 xt，以便与代码测试目录 t 分开。

这次运行的测试同样也跳过了 t/pod-coverage.t 和 t/pod.t，这是因为我们没有安装这些相应的模块。在这些测试做任何事情之前，它们会检查那些需要完成工作的相依赖模块。如果没有，就将跳过这些相应的测试。第 20 章展示这些跳过的测试内容。

14.5.1 模块编译的检查

我们看看 t/00-load.t 文件，这是运行的第一个测试文件，因为默认测试顺序是按照字典顺序执行。如下就是试图编译 Animal 模块的测试脚本：

```
#!/perl -T

use Test::More tests => 1;

BEGIN {
    use_ok( 'Animal' ) || print "Bail out!\n";
}

diag( "Testing Animal $Animal::VERSION, Perl $[, $^X" );
```

首先，测试程序会加载 Test::More 模块，并声明将只有一个测试用例。在 BEGIN 语句块内部，

它使用 Test::More 模块的 use_ok 子例程。提供一个模块的名称，use_ok 子例程将试图加载这个模块。如果发生问题，例如一个语法错误，测试将失败，use_ok 子例程的返回值为 false。

当 use_ok 子例程的返回值为 false 时，就将继续执行||操作符后的 print 语句。如果从标准输出中发现字符串“Bail out！”，测试框架就会立即终止运行。如果我们不能加载模块，继续接下来的测试将毫无意义。与在整屏的错误信息中深入研究错误的原因相比，这是很容易发现语法错误的方式。



注意

我们必须使用 -I 参数来添加模块搜索路径，因为这些测试开启了“污染”检查模式，这个模式下将忽略 PERL5LIB 环境变量。

要看看测试程序究竟在做些什么，我们可以自己运行它。可以使用 -I 参数将 blib/lib 构建目录添加到 @INC 数组中。然而，在我们想要对于 blib 进行测试之前，我们应当重新构建发行版以保证使用的是最新的代码。我们还要在命令行中指定 -T 参数，因为我们必须在测试前打开“污染”检查模式：

```
% ./Build  
% perl -Iblib/lib -T t/oo-load.t  
1..1  
ok 1 - use Animal;  
# Testing Animal 0.01, Perl 5.014002, perl
```

每一项测试都输出单个行，表示要么 ok，要么 not ok，并且后面附带一个测试编号。Test::More 模块自动处理大部分细节问题，所以我们不必担心这些大多数组细节。

与使用 -Iblib/lib 相比，我们可以使用 blib 模块来搜索周围的目录，包括父目录，用于将 blib 添加到 @INC 数组中。如下所示，我们可以在命令行上用 -M 开关来加载它：

```
% perl -Mblib -T t/oo-load.t
```



注意

在运行这个测试之前，我们重新运行构建程序，以确保我们测试的是最新的代码。

因为我们在发行版中添加了额外的类，所以我们也想测试它们，因此，如下所示，我们需要重新安排测试程序来处理不止一个类：

```
#!/perl -T  
  
BEGIN {  
    my @classes = qw(Animal Cow Sheep Horse Mouse);  
    use Test::More tests => scalar @classes;  
  
    foreach my $class ( @classes ) {  
        use_ok( $class ) or print "Bail out! $class did not load!\n"  
    }  
}
```

现在，输出显示测试框架自动做了更多的工作。第一行是一个测试计数：它告诉测试框架所期望的测试数目。每一行显示测试的内容：

```
% ./Build  
% perl -Iblib/lib -T t/00-load.t  
1..4  
ok 1 - use Animal;  
ok 2 - use Cow;  
ok 3 - use Sheep;  
ok 4 - use Horse;
```

14.5.2 模板测试

在查看完 t/00-load.t 文件以后，我们继续查看 t/boilerplate.t 文件。当创建模块文件时，创建工具就自动添加文本。如下所示，在存根模块文件中有一些基本的文档：

```
=head1 SYNOPSIS  
  
Quick summary of what the module does.  
  
Perhaps a little code snippet.  
use Horse;  
  
my $foo = Horse->new();  
...
```

下面是一些基本代码：

```
sub function1 {  
}
```

我们应当用我们实际模块的文档替换这些内容并填入代码。已有的内容就是模板。t/boilerplate.t 测试脚本将查找这些占位符文本，如果找到它，它就会抱怨我们没有修改它。

这个测试文件的开始部分定义一个叫 not_in_file_ok 的子例程：

```
#!/usr/bin/perl -T  
  
use 5.006;  
use strict;  
use warnings;  
use Test::More tests => 6;  
  
sub not_in_file_ok {  
    my ($filename, %regex) = @_;  
    open( my $fh, '<', $filename )  
        or die "couldn't open $filename for reading: $!";  
  
    my %violated;  
  
    while (my $line = <$fh>) {  
        while (my ($desc, $regex) = each %regex) {  
            if ($line =~ $regex) {  
                push @{$violated{$desc}} ||= [], $.;
```

```

    }
}

if (%violated) {
    fail("$filename contains boilerplate text");
    diag "$_ appears on lines @{$violated{$_}}" for keys %violated;
} else {
    pass("$filename contains no boilerplate text");
}
}

```

该子例程使用一个文件名和一个值为正则表达式的散列作为参数。它读取文件并查找与这些正则匹配的行，如果它找到一个匹配的行，就将该行记录在%violated 散列中。一旦它已经检查所有文件，它将执行 if (%violated) 查看它是否发现任何问题。如果发现问题，它就调用 fail 子例程，这是一个 Test::More 模块的子例程，并且仅仅接受一个测试标签，然后输出 not ok。如果没有发现违规记录，它就调用 pass 子例程。

t/boilerplate.t 文件的下一个部分是另外一个子例程：module_boilerplate_ok。它接受一个模块的文件名并且传递一个键值对列表到 not_in_file_ok 子例程中用于检查：

```

sub module_boilerplate_ok {
    my ($module) = @_;
    not_in_file_ok($module =>
        'the great new $MODULENAME'    => qr/- The great new /,
        'boilerplate description'     => qr/Quick summary of what the module/,
        'stub function definition'   => qr/function[12]/,
    );
}

```

文件的下一部分很有趣。有一个语句块的标签为 TODO。这是 Test::More 的特性，允许我们标注我们期望那些测试失败，但不用立即修复测试用例。在语句块内部，它设置 \$TODO 的值作为测试的一个标签，我们稍后会介绍。在此之后，剩下的语句块将调用 not_in_file_ok 子例程和 module_boilerplate_ok 子例程：

```

TODO: {
    local $TODO = "Need to replace the boilerplate text";

    not_in_file_ok(README =>
        "The README is used..."      => qr/The README is used/,
        "'version information here'" => qr/to provide version information/,
    );

    not_in_file_ok(Changes =>
        "placeholder date/time"     => qr(Date/time)
    );

    module_boilerplate_ok('lib/Animal.pm');
    module_boilerplate_ok('lib/Cow.pm');
    module_boilerplate_ok('lib/Horse.pm');
    module_boilerplate_ok('lib/Mouse.pm');
    module_boilerplate_ok('lib/Sheep.pm');
}

```

直至我们更新文件，这些测试将一直失败，但是当我们运行这些测试时，这些测试看起来好像又能够通过。这是因为\$TODO 标签将自身附加在测试标签末尾。当测试用具看到这些时，它不将其视为一个真正的测试失败：

```
% ./Build  
% perl -Iblib/lib t/boilerplate.t  
1..6  
not ok 1 - README contains boilerplate text # TODO Need to replace the boilerplate text  
#   Failed (TODO) test 'README contains boilerplate text'  
#   at t/boilerplate.t line 24.  
# The README is used... appears on lines 3  
# 'version information here' appears on lines 11  
...
```

当我们修正了模板测试中的一个问题之后，测试就将通过。既然 TODO 表示我们期望这些测试无法通过，那么测试用具将添加一个 TODO 在通过情况下的报告：

```
Test Summary Report  
-----  
t/boilerplate.t (Wstat: 0 Tests: 6 Failed: 0)  
    TODO passed: 1  
    Files=5, Tests=20, 0 wallclock secs  
    Result: PASS
```

我们不用保留 t/boilerplate.t 文件，一旦我们替换了占位符文本，就可以删除这个文件了。

14.5.3 测试 Pod

module-starter 程序为我们提供了一些初始文档，并且它创建了检查这些文档的测试用例。标准的 Pod 测试只关心两件事：我们的 Pod 没有任何格式错误和我们已经为每个子例程编写了文档。这些测试都是可选的，并且只有在安装了 Test::Pod 模块和 Test::Pod::Coverage 模块后才会运行。这些测试会自动查找所有模块文件并逐个进行测试，因此我们不用修改这些测试脚本。

如果我们还没有给新添加的子例程编写相应的文档，Pod 覆盖测试就会无法通过：

```
% ./Build test  
t/oo-load.t ..... 1/5  
t/oo-load.t ..... ok  
t/boilerplate.t ... ok  
t/manifest.t ..... skipped: Author tests not required for installation  
t/pod-coverage.t .. 1/5  
#   Failed test 'Pod coverage on Animal'  
#   at .../Test/Pod/Coverage.pm line 126.  
# Coverage for Animal is 0.0%, with 2 naked subroutines:  
#     sound  
#     speak  
  
#   Failed test 'Pod coverage on Cow'  
#   at .../Test/Pod/Coverage.pm line 126.  
# Coverage for Cow is 0.0%, with 1 naked subroutine:  
#     sound  
...
```

```
Failed 5/5 subtests
t/pod.t ..... ok

Test Summary Report
-----
t/boilerplate.t (Wstat: 0 Tests: 6 Failed: 0)
  TODO passed: 3
t/pod-coverage.t (Wstat: 1280 Tests: 5 Failed: 5)
  Failed tests: 1-5
  Non-zero exit status: 5
  Files=5, Tests=20, 1 wallclock secs ( 0.04 usr  0.02 sys +  0.19 cusr  0.03 csys = /
  0.28 CPU)
Result: FAIL
Failed 1/5 test programs. 5/21 subtests failed.
```

要修复这个 Pod 测试，我们需要使用我们自己的方法中的文档替换存根文档中的信息。一旦这样做了，那么 Pod 测试就会通过。

14.6 添加第一个测试

如果需要测试模块，我们可以添加我们自己的测试文件到发行版中，然后逐步建立整个测试。

首先，创建一个名为 t/Animal.t 的文件，我们将用该文件来测试 lib/Animal.pm 中的功能。开始，我们使用 Test::More 模块中的特殊 pass 子例程，该子例程能够使测试一直通过。

```
use strict;
use warnings;

use Test::More tests => 1;

pass();
```

现在，我们再次运行测试套件，就会看到 t/Animal.t 测试运行并且测试通过：

```
% ./Build test
t/00-load.t ..... ok
t/Animal.t ..... ok
t/boilerplate.t ... ok
t/manifest.t ..... skipped: Author tests not required
t/pod-coverage.t .. ok
t/pod.t ..... ok
All tests successful.
Files=6, Tests=11, 1 wallclock secs ( ... )
Result: PASS
```

下一步，我们需要添加一些更有趣的测试。尽管我们已经测试过 lib/Animal.pm 的编译，但是我们可以在 t/Animal.t 中再次测试这项内容。当我们想要通过自身单独运行这个测试文件时，这是非常有用的：

```
use strict;
use warnings;
```

```
use Test::More tests => 1;

BEGIN {
    require_ok( 'Animal' ) || print "Bail out!\n";
}

diag( "Testing Animal $Animal::VERSION, Perl $], $^X" );
```

如果我们有最新的 Test::More 模块版本（而且应当这样做，因为精简版太古老了），我们可以使用 BAIL_OUT 子例程来处理错误。

```
use strict;
use warnings;

use Test::More 0.62 tests => 1;
BEGIN {
    require_ok( 'Animal' ) || BAIL_OUT();
}
diag( "Testing Animal $Animal::VERSION, Perl $], $^X" );
```

我们再次运行测试以确保测试仍然能够通过，这不应该有问题。现在我们想要测试 Animal 模块中的一个方法。到目前为止，我们在该模块中只有两个方法：speak 和 sound。虽然对于示例而言它有些大材小用，但在更加复杂的代码中我们可能也想测试已经定义的子例程。这只是一个容易的测试：使用 Test::More 模块中的 ok 函数：

```
use strict;
use warnings;

use Test::More tests => 3;

BEGIN {
    use_ok( 'Animal' ) || print "Bail out!\n";
}

diag( "Testing Animal $Animal::VERSION, Perl $], $^X" );

# they have to be defined in Animal.pm
ok( defined &Animal::speak, 'Animal::speak is defined' );
ok( defined &Animal::sound, 'Animal::sound is defined' );
```

在 Animal 模块中，使用 defined 内置函数的方法测试模块，因为我们想检查它们实际上是在类中定义，而不是从其他类（例如 LivingCreature 类）继承得到的。

一旦我们确定已经定义了我们的方法，我们就会测试这些方法是否完成所期望做的事情。在这个示例中，我们有点苦恼，因为 Animal 的 sound 子例程像是“死前的哀嚎”。我们在最低层次并且以我们自己的方式进行测试。现在我们想测试这个 sound 子例程的“死前的哀嚎”，而且得到了正确的信息。如下所示，来自 Test::More 模块的 like 子例程检查它的第一个参数是否匹配后面的正则表达式：

```
use strict;
use warnings;

use Test::More tests => 4;
```

```

# same as before
...
# check that sound() dies
eval { Animal->sound() } or my $at = $@;
like( $at, qr/You must/, 'sound() dies with a message' );

```

如果 eval 中的语句运行失败，我们立即将\$@中错误信息保存到一个新变量中。就像 Perl 的其他全局变量一样，这个值可能在后续我们做了其他事情后发生变化。有另外一种方式让我们测试这样的错误，但这不是本章要讲的重点。

下一步我们要测试 speak 子例程。既然它会调用 sound 子例程，它也会运行失败，因此测试方法几乎相同：

```

use strict;
use warnings;

use Test::More tests => 5;

# same as before
...

# check that sound() dies
eval { Animal->sound() } or my $at = $@;
like( $at, qr/You must/, 'sound() dies with a message' );

# check that speak() dies too
eval { Animal->speak() } or my $at = $@;
like( $at, qr/You must/, 'speak() dies with a message' );

```

然而，要完整测试 sound 子例程，就必须在它不会运行失败的一种情况下测试。sound 子例程的大多数内部工作都依赖于一个子类，因此我们可以为此设置一个小的测试子类，然后使用 Test::More 模块的 is 子例程来确保我们获得了正确的信息。如下所示，我们将创建一个名为 Foofle 的子类，并且将该子类的定义打包，然后在一个裸块中测试，以限定它的作用域：

```

use strict;
use warnings;

use Test::More tests => 6;

# same as before
...

{
    package Foofle;
    use parent qw(Animal);
    sub sound { 'foof' }

    is(
        Foofle->speak,
        "A Foofle goes foof!\n",
        'An Animal subclass does the right thing'
    );
}

```

把这些都加起来，我们就有 Animal 类的一个完整的测试：

```
use strict;
use warnings;

use Test::More tests => 6;

BEGIN {
    use_ok( 'Animal' ) || print "Bail out!\n";
}
diag( "Testing Animal $Animal::VERSION, Perl $], $^X" );

# they have to be defined in Animal.pm
ok( defined &Animal::speak, 'Animal::speak is defined' );
ok( defined &Animal::sound, 'Animal::sound is defined' );

# check that sound() dies
eval { Animal->sound() } or my $at = $@;
like( $at, qr/You must/, 'sound() dies with a message' );

# check that speak() dies too
eval { Animal->speak() } or my $at = $@;
like( $at, qr/You must/, 'speak() dies with a message' );

{
    package Foofle;
    use parent qw(Animal);
    sub sound { 'foof' }

    is(
        Foofle->speak,
        "A Foofle goes foof!\n",
        'An Animal subclass does the right thing'
    );
}
```

14.7 测量测试覆盖率

我们的目标是完整地测试全部代码。虽然这可能并不总是可行或经济的，但这仍是我们的目标。有几个覆盖度量指标，每一种分别对应一种不同类型的测试。我们应当记住，完美的测试度量指标并不是目的，我们更希望得到完美的代码，测试指标仅仅只是一个数字而已。

我们使用 CPAN 上的 Devel::Cover 模块来收集这些度量指标。如下所示，如果我们正在使用 Module::Build 模块，我们可以将 testcover 作为 Build 的参数：

```
% ./Build testcover
```

如果我们使用 ExtUtils::Makemaker 模块，就可以按照如下方式使用 HARNESS_PERL_SWITCHES 环境变量：

```
% HARNESS_PERL_SWITCHES=-MDevel::Cover make test
```

在运行测试之后，需要运行 cover 命令来将收集到的统计信息转换成可读的报告。该命

令将输出一个总结性的报告：

```
% cover
```

cover 命令会创建一个总结性的报告，它如下所示：

```
Reading database from /Users/brian/Desktop/Animal/cover_db
```

File	stmt	bzan	cond	sub	pod	time	total
Animal.pm	60.0	0.0	n/a	42.9	100.0	90.7	57.1
Cow.pm	85.7	n/a	n/a	66.7	100.0	0.9	81.8
Horse.pm	85.7	n/a	n/a	66.7	100.0	8.1	81.8
Sheep.pm	85.7	n/a	n/a	66.7	100.0	0.4	81.8
Total	78.9	0.0	n/a	60.0	100.0	100.0	74.5

该命令还创建一个为每个文件显示更多细节的 `cover_db/coverage.html` 文件，因此我们不仅可以逐行查看相应的测试覆盖率，还可以深入探讨相应的文件和度量指标，以便查看仍然需要测试的内容，来提高测试覆盖率。

14.7.1 子例程覆盖率

这个度量指标测量子例程的测试覆盖百分率，我们致力于测试所有子例程，这可能是完全覆盖最简单的度量指标。我们必须运行每一个子例程以便得到一个完美的高分。这是一个可信度较低的度量指标，因为即使我们测试了全部子例程，代码仍然可能会有严重的错误。只要我们运行子例程，它就计数，即使程序出错。

14.7.2 语句覆盖率

运行每一个子例程是不够的。我们还想测试每一条语句，不管它是否在子例程内部。这是一个比子例程覆盖率稍微好一点的度量指标，但它也不能保证正确的代码。一条语句由表达式组成，表达式由术语构成。虽然我们执行了语句，但这并不意味着我们已经测试了所有表达式或术语。

14.7.3 分支覆盖率

程序流可以在 if、unless 和 given-when 语句中出现多个分支：

```
if( ) { ... }
elsif() { ... }
else { ... }

unless()
elsif() { ... }
else { ... }

give( ... )
when { ... }
when { ... }
when { ... }
}
```

要获得完整的分支覆盖率，我们需要测试这些结构的每个分支，这就意味着我们需要使测试代码能够触发每个分支语句块。

14.7.4 条件覆盖率

分支覆盖率内部的条件语句可能使用多个条件语句，但还有其他地方的语句可能会产生条件行为。我们需要建立每一个条件行为的测试，例如，下面这些语句分别有两部分：

```
my $foo = $n || $m;  
  
if( $n && $m ) {  
    ...  
}  
  
while( $n && $m ) {  
    ...  
}  
  
open my($fh), '>', $file  
or die "Could not open file! $!\n";
```

要覆盖这些语句的全部条件，我们需要测试每一种情况下的相应条件。我们需要测试每个逻辑运算符为真或为假的组合情况下的全部可能。

14.8 练习

可以在附录中的“第 14 章答案”部分找到这些练习的答案。

1. [35 分钟] 编写一个模块发行版，从测试开始写。创建一个 My::List::Util 模块，它包含两个子例程：sum 和 shuffle。sum 子例程接受一个值列表并返回它们的数字和。shuffle 子例程将一个数据列表随机地重新排列，并返回这个列表。

从 sum 子例程开始，编写测试脚本，然后添加代码。当测试通过的时候，你就知道已经做完了。现在，对于包括 shuffle 子例程的测试，添加 shuffle 的实现。可以在 perlfaq4 文档中或 List::Util 模块的文档中找到 shuffle 实现的介绍。

确保做这些的同时更新了文档和 MANIFEST 文件。

保存你的发行版以便用于第 17 章和第 20 章的练习。

2. [25 分钟] 添加 t/Animal.t 测试文件到你的发行版中，并使它能够工作。当你添加其他部分的测试时，在添加新的测试文件之前先运行测试套件。
3. [15 分钟] 为 Cow、Horse 和 Sheep 类创建测试文件。添加一个测试来确保每个类都能正确编译。为每一个类添加 sound 方法的测试代码。
4. [5 分钟] 使用 Devel::Cover 模块测量测试套件的覆盖率。既然你还没有实现对 Cow、

Horse 和 Sheep 类的完整测试，你就应当看看代码覆盖率度量指标到底有多低。这很好，因为你会在接下来的练习中解决这个问题。

5. [25 分钟] 完成 Cow、Horse 和 Sheep 类的测试，这样你就得到很高的测试覆盖率指标（或足够高）。测试每一种 Animal 的 sound 和 speak 方法并完成相应文档的编写。

第 15 章

带数据的对象

使用第 13 章介绍的简单语法，我们就可以创建类方法、（多重）继承、重载和扩展。我们能够把代码中共同的部分提取出来，通过一个抽象这些共同部分的函数的方法实现重用。这是对象提供的核心特性之一，但对象也能提供实例数据，本章将介绍这部分内容。

15.1 马属于马类，各从其类是吗

我们看一下在第 11 章用于表示 Animal 类和 Horse 类的代码，Animal 类提供通用的 speak 子例程：

```
package Animal;

sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n"
}
```

Horse 类从 Animal 类中继承，但提供它自己特定的 sound 子例程：

```
package Horse;
@ISA = qw(Animal);
sub sound { 'neigh' }
```

这允许我们将调用 Horse->speak 变成向上调用 Animal::speak，回调 Horse::sound 获取特定的 sound 子例程，然后就会得到如下输出：

```
a Horse goes neigh!
```

但是这样，所有 Horse 对象就都是绝对相同的。如果添加一个方法，所有 Horse 对象就自动共享这个方法。这对于生成相同的马就太方便了，但我们如何捕获个别马的属性呢？例如：我们想给我们的马起个名字，应该有办法使这个名字区别其他马。

我们可以建立一个实例来实现这个功能。一个实例通常是根据类创建的，就好像汽车由汽车工厂生产一样。一个实例会有关联的属性，称为实例变量（或叫做成员变量，如果你有 C++ 或 Java 背景）。每个实例都有一个唯一的标识（就像注册过的马的序列号），多个可共享的属性（马的颜色和特长），以及通用行为（例如，拉缰绳就是告诉马要停下）。

在 Perl 中，实例必须是一个内置类型的引用。从一个最简单的标量引用^{注1}开始，该引用可以保存马的名字。如下所示，添加一个 scripts/horse.pl 脚本到发行版中，该脚本文件将对我们最喜欢的马的名字取引用，然后“bless”它到 Horse 包中（参考图 15-1）：

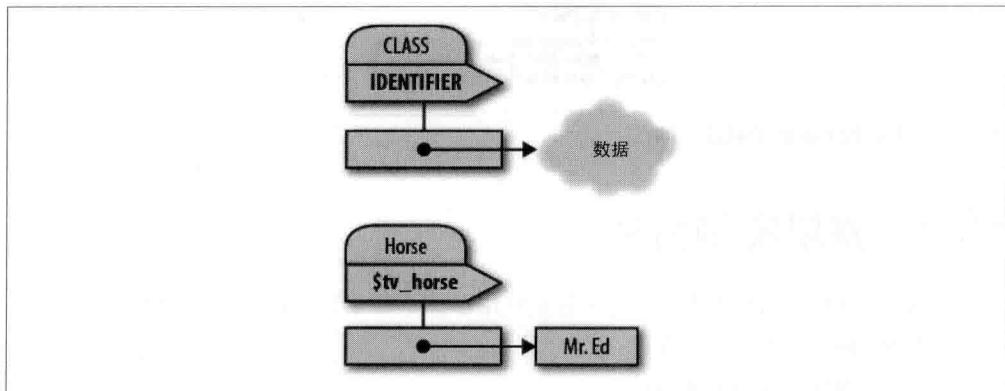


图 15-1 常见对象的 PeGS 结构

```
#!perl
# scripts/horse.pl
my $name = 'Mr. Ed';
my $tv_horse = \$name;

bless $tv_horse, 'Horse';
```

现在 \$tv_horse 是指向一个特定实例数据（名字）的引用。跟随在 bless 操作符后的引用将查找它所指向的变量——在这里就是标量\$name。然后通过给它附加包名称它“bless”该变量，将 \$tv_horse 变量变为一个对象——Horse 对象。（想象一下，刚才说的那匹马带了一个上面写着 \$name 的小标签。）

一个对象的 PeGS 结构看起来和对象引用所表述的内容相同，我们只是给它加了顶表明包名的“帽子”。对于基本的对象，我们并不这么做：

我们已经 bless 了 \$tv_horse，它是一个标量引用，它独有的 PeGS 将如图 15-2 所示。

此时此刻，\$tv_horse 是 Horse 类的一个实例^{注2}。也就是说，它是一匹特别的马（horse）。

注 1：这是最简单的，但很少在实际代码中使用，我们后续将详细介绍原因。

注 2：实际上，\$tv_horse 指向对象，但是在常用的术语中，我们几乎总是通过指向这些对象的引用处理对象。因此，最简单的方式就是说 \$tv_horse 就是马（horse），而不是“\$tv_horse 所引用的东西”。

此外，这个引用并没有什么改变，依旧可以用传统的解引用操作符。^{注3}

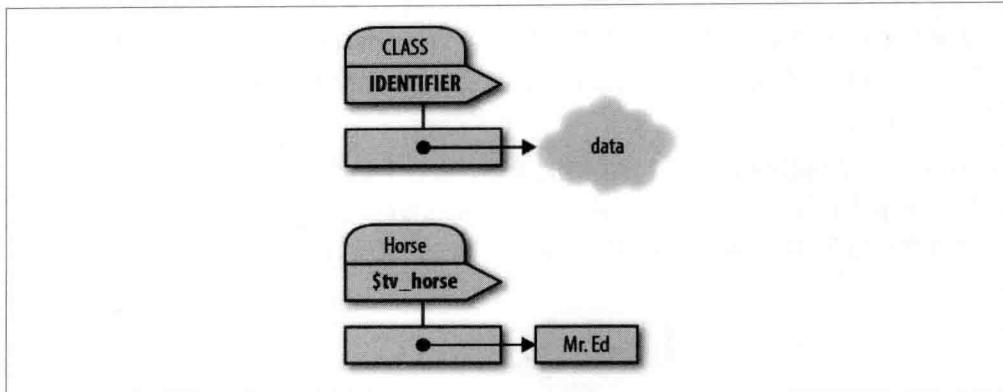


图 15-2 \$tv_horse 的 PeGS 结构

15.2 调用实例方法

方法的箭头可以在实例以及包（类）名上使用。如下所示，我们扩展 scripts/horse.pl 脚本来获取 \$tv_horse 发出的声音：

```
# scripts/horses.pl, as before
my $noise = $tv_horse->sound;
```

调用 sound 的过程是这样：首先 Perl 解释器将记录 \$tv_horse 是一个被 bless 的引用，因而就是一个对象实例。然后，Perl 解释器创建一个参数列表，就像我们用类名加方法调用箭头构建的参数列表一样。在该示例中，它就仅仅是 \$tv_horse。稍后，我们将展示实例变量后面的参数，它与跟在类后面的方式类似。

现在就是最有趣的部分：Perl 从被 bless 过的实例变量中取出类的名称，在这个示例中就是 Horse，然后用它来定位和调用方法，就好像我们声明 Horse->sound，而不是 \$tv_horse->sound。最初 bless 的目的就是把一个类和这个引用关联起来，让 Perl 解释器能找到适当的方法。

在这里，Perl 直接查找 Horse::Sound（没有使用继承），生成最终的方法调用。

```
Horse::sound($tv_horse)
```

此处的第一个参数仍然是实例，而不是与之前一样的类名。neigh 为返回值，以之前的 \$noise 变量结束。

如果 Perl 解释器没有找到 Horse::sound，它将会遍历 @Horse::ISA 列表以试图在其中某个超类中查找该方法，就好像查找一个类方法一样。实例方法和类方法的唯一区别在

注 3：因此在类的外部做这些是一个很糟糕的主意，我们后续将详细介绍原因。

于它们的第一个参数是否为实例（一个被 bless 引用）或者类名（字符串）^{注4}。否则，它们都只是 Perl 子例程。

15.3 访问实例数据

因为我们获取实例作为第一个参数，所以我们现在可以访问特定实例的数据。在这里，添加一个获取名称的方法。如下所示，在 lib/horse 中，添加了一个 name 方法：

```
sub name {
    my $self = shift;
    $$self;
}
```

现在在 scripts/horse.pl 脚本中调用 name 方法：

```
print $tv_horse->name, " says ", $tv_horse->sound, "\n";
```

在 Horse::name 内部，@_ 数组只包含 \$tv_horse 标量，而且该标量通过 shift 语句存入 \$self 中。这是一个很常见的方法：通过 shift 语句，将默认数组的第一个参数，传递给实例方法一个叫做 \$self 的标量，因此，我们将坚持用这个方法，除非我们有其他强有力的理由不这么做。然而，Perl 解释器在 \$self 的名称上并没有赋予什么重要意义^{注5}。然后我们对 \$self 作为标量引用进行解引用操作，得到 Mr. Ed。如下所示，结果为：

```
Mr. Ed says neigh.
```

15.4 如何构建 Horse 的实例

如果我们手动构建所有马（horse）的实例，我们很可能有时会犯错误：使马的“内部”可见，这也违反了 OOP 的一个称为封装的原则。如果我们是兽医，能够查看马的内部结构没有什么不好，但如果只是喜欢马，就不好了。如下所示，我们让 Horse 类自动构建一个新的 Horse 实例：

```
package Horse;
use parent qw(Animal);
sub sound { 'neigh' }
sub name {
    my $self = shift;
    $$self;
}
sub named {
    my $class = shift;
    my $name = shift;
    bless \$name, $class;
}
```

注 4：这可能和你知道的其他 OOP 语言不同。

注 5：如果我们有另一种面向对象语言背景，就可能会选择 \$this 或 \$me 作为变量名，但这样可能会使其他大多数 Perl OO 程序员感到疑惑。

如下所示，现在通过新的 named 方法，我们可以构建 Horse 类的一个实例而不必直接生成引用：

```
# my $name = 'Mr. Ed';
# my $tv_horse = \$name;
my $tv_horse = Horse->named('Mr. Ed');
```

我们返回一个类方法，因此，Horse::named 的两个参数叫做“Horse”和“Mr. Ed”。bless 操作符不但 bless \$name，而且返回指向\$name 的引用，因此就将该引用作为返回值。这就是构建 Horse 实例的方法。

在这里调用 named 构造函数，所以它表示构造函数的参数作为这个特殊的 Horse 类的名称。我们可以使用不同的构造函数和不同的名称，以不同的方法生成对象（如记录其血统或出生日期）。然而，我们会发现大多数人使用一个叫做 new 的构造函数，使用各种不同的方式解释 new 的参数。这些风格都很好，只要我们在文档中记录生成对象的特殊方法。大多数核心模块和 CPAN 的模块都使用 new，也有明显的例外，比如 DBI 模块的 DBI->connect()。这完全取决于模块的作者，而且任何方法都可以是一个构造函数。

15.5 继承构造函数

对于 named 方法中的 Horse 类有什么特殊的设定吗？没有。这是从 Animal 类继承来构建其他类的相同“策略”，因此，如下所示，将它放在 Animal 类中：

```
package Animal;

sub name {
    my $self = shift;
    $$self;
}

sub named {
    my $class = shift;
    my $name = shift;
    bless \$name, $class;
}
```

但是如果我们在一个实例上调用 speak 方法将会发生什么？

```
my $tv_horse = Horse->named('Mr. Ed');
$tv_horse->speak;
```

我们将得到一个调试值：

```
a Horse=SCALAR(0xaca42ac) goes neigh!
```

为什么？这是因为 Animal::speak 方法期望一个类名作为它的第一个参数，而不是一个实例。当传入一个实例替代时，使用一个 bless 的标量引用作为字符串，这表现为我们在上面所展现的内容——类似于一个字符串化的引用，只不过将类名放在最前面。

15.6 编写能够使用类或实例作为参数的方法

我们需要解决的是找到一个途径，来判断方法是类还是实例上调用。最简单的查明办法是使用 ref 操作符。如果该操作符用于一个 bless 的引用，就返回一个字符串（类名）；如果用于一个字符串，就返回 undef（像一个类名）。如下所示，我们先修改 name 方法来提示这个变化：

```
# in lib/Animal.pm
sub name {
    my $either = shift;
    ref $either
        ? $$either           # it's an instance, return name
        : "an unnamed $either"; # it's a class, return generic
}
```

此处“?:”操作符的选择要么是解引用要么是派生的字符串。现在我们可以在实例或者类中使用它。如下所示，我们将第一个参数的持有者变更为\$either，表明这是有意为之：

```
print Horse->name, "\n";      # prints "an unnamed Horse\n"

my $tv_horse = Horse->named('Mr. Ed');
print $tv_horse->name, "\n";  # prints "Mr Ed.\n"
```

现在，我们将按照如下方式修正 speak 方法：

```
sub speak {
    my $either = shift;
    print $either->name, ' goes ', $either->sound, "\n";
}
```

当前的 sound 方法已经既能够操作类又能够操作实例，功能全部完成！

15.7 为方法添加参数

我们想允许动物（animal）吃（eat）东西。我们为 Animal 类添加一个 eat 方法，通过该方法，传递了一些让动物咀嚼的东西：

```
package Animal;
sub named {
    my( $class, $name ) = @_;
    bless \$name, $class;
}
sub name {
    my $either = shift;
    ref $either
        ? $$either           # it's an instance, return name
        : "an unnamed $either"; # it's a class, return generic
}
sub speak {
    my $either = shift;
```

```
    print $either->name, ' goes ', $either->sound, "\n";
}
sub eat {
    my $either = shift;
    my $food = shift;
    print $either->name, " eats $food.\n";
}
```

现在，如下所示，我们尝试在一个新的程序 scripts/horse-and-sheep.pl 中，给动物提供它们最喜欢的食物：

```
my $tv_horse = Horse->named('Mr. Ed');
$tv_horse->eat('hay');
Sheep->eat('grass');
```

得到的输出为：

```
Mr. Ed eats hay.
an unnamed Sheep eats grass.
```

一个带参数的实例方法通过作为第一个参数的实例本身调用，然后是参数列表。第一个调用如下所示：

```
Animal::eat($tv_horse, 'hay');
```

实例的方法构成了对象的应用程序编程接口（API）。设计一个好的对象类的绝大多数努力最终都会进入到 API 的设计之中，因为 API 定义了可重用和可维护的对象以及它的子类的内容。在我们考虑清楚我们（或其他人）如何使用对象之前，不要急于冻结 API 设计。

15.8 更有趣的实例

如果实例需要更多的数据，该如何实现？大多数有趣的实例由许多数据项组成，其中每个数据项都可以轮流作为引用或者另一个对象。存储这些数据项最简单的方法通常是存入一个散列之中。散列的键作为部分对象的名称（也称为实例或成员变量），相应的值就是对应数据项的值。

如何将 Horse 实例变换成为一个散列？回想一下，对象是一个被 bless 操作过的引用，我们可以很容易地使一个 bless 的散列引用成为 bless 的标量引用，只要关注引用的全部内容做出相应改变。

如下所示，我们使一只羊（sheep）拥有名字和颜色：

```
my $lost = bless { Name => 'Bo', Color => 'white' }, 'Sheep';
```

\$lost->{Name} 为 Bo，\$lost->{Color} 为 white。但是我们想让 \$lost->name 来访问名字，但却因为它是一个标量引用而导致混乱。

不要着急，我们可以轻松搞定它：

```
## in Animal
sub name {
    my $either = shift;
    ref $either
        ? $either->{Name}
        : "an unnamed $either";
}
```

如下所示，`named` 方法仍然构建 `sheep` 标量的一个实例，所以我们也做相应的修改：

```
## in Animal
sub named {
    my $class = shift;
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    bless $self, $class;
}
```

`default_color`（默认颜色）是什么？如果 `named` 方法只有 `name` 变量，我们仍然需要设置一种颜色，所以我们将有一个特定于类的初始颜色。如下所示，对于羊（`sheep`），我们可能会将其初始颜色定义为白色：

```
## in Sheep
sub default_color { 'white' }
```

然后，不必为每个额外类定义一种颜色，可以定义一个支撑方法，作为默认的默认颜色，直接用于 `Animal` 类：

```
## in Animal
sub default_color { 'brown' }
```

因此，现在所有动物都是褐色的（也许是泥泞的），除非某个特定 `animal` 类给出了该方法的一个特定重写版本。

现在，因为 `name` 方法和 `named` 方法是引用对象结构的唯一方法，其余方法可以保持不变，所以 `speak` 方法还是像以前一样工作。这支持 OOP 的另一个基本规则：如果只有对象访问其内部数据，当需要修改它们的结构时，只需要改变很少的代码。

15.9 一匹不同颜色的马

如果所有马都是褐色可能会很无趣。如下所示，添加了以下几个方法来获取和设置颜色：

```
## in Animal
sub color {
    my $self = shift;
    $self->{Color};
}

sub set_color {
    my $self = shift;
    $self->{Color} = shift;
}
```

如下所示，现在可以修改 Mr. Ed 的颜色：

```
my $tv_horse = Horse->named('Mr. Ed');
$tv_horse->set_color('black-and-white');
print $tv_horse->name, ' is colored ', $tv_horse->color, "\n";
```

得到的结果为：

```
Mr. Ed is colored black-and-white
```

15.10 收回存款

根据我们写代码的方式，setter 也返回更新的值。当我们编写 setter 时，也应该考虑这些（并且记录它们）。setter 的返回值是什么呢？以下是一些常见的变化：

- 更新后的参数（与传递的相同）
- 之前的值（与 umask 或 select 的单参数形式相似）
- 对象自身
- 成功/失败代码

每一种 setter 都有各自的优点和缺点。例如，如果返回更新后的参数，可以在另一个对象中再次使用它：

```
$tv_horse->set_color(
    $eating->set_color( color_from_user() )
);
```

之前的实现返回最近更新后的值。通常情况下，这是最简单的代码编写方式，而且通常执行也是最快的。

如果返回之前的参数，可以很容易创建“为此临时设置这个值”的功能：

```
{
    my $old_color = $tv_horse->set_color('orange');
    ... do things with $tv_horse ...
    $tv_horse->set_color($old_color);
}
```

这以如下方式实现：

```
sub set_color {
    my $self = shift;
    my $old = $self->{Color};
    $self->{Color} = shift;
    $old;
}
```

如下所示，为了更高效，可以使用 wantarray 函数，来避免当调用空上下文时，存储前一项的值：

```

sub set_color {
    my $self = shift;
    if (defined wantarray) {
        # this method call is not in void context, so
        # the return value matters
        my $old = $self->{Color};
        $self->{Color} = shift;
        $old;
    } else {
        # this method call is in void context
        $self->{Color} = shift;
    }
}

```

如下所示，如果返回对象本身，可以使用这样的链式设定：

```

my $tv_horse =
    Horse->named('Mr. Ed')
    ->set_color('grey')
    ->set_age(4)
    ->set_height('17 hands');

```

这样的设定能够工作的原因，是每个 setter 的返回值是初始对象，成为下一次方法调用的对象。如下所示，再次实现这些就相对更简单：

```

sub set_color {
    my $self = shift;
    $self->{Color} = shift;
    $self;
}

```

空上下文的技巧也能用在这里，因为我们已经在此创建了\$self，即使这样做有尚属疑问的好处。

最终，如果一个更新失败相当常见，返回一个成功状态而不是一个异常事件是很有用的。其他的变化必须通过 die 语句抛出异常来显示失败。



注意

我们可以用我们想要的如何东西，尽可能地保持一致，但依旧记录下它们（并且不要在我们已经发布一个版本之后再修改）。没有任何一个答案能够涵盖所有情况。

15.11 不要查看“盒子”里面的内容

我们可能已经在类的外部按照散列引用的规则获取或者设置颜色：\$tv_horse->{Color}。然而，这样暴露对象的内部结构违背了对象的封装性。对象应该是一个黑盒，但是我们已经撬开它的锁链，窥探内部的细节。

OOP 的目的之一，就是使 Animal 类或者 Horse 类的维护人员，对于方法的实现能够合理地做出独立的改变，而且导出的接口依然能够正常工作。要弄清楚为什么直接访问

散列就将违反了这个规则，我们可以说 Animal 类不再使用一个简单的 color 名称来表述 color 方法，而是使用一个计算过的 RGB 三重值来存储颜色（以数组引用形式保存）。在这个示例中，使用一个虚构的（在撰写本文时）Color::Conversions 模块改变幕后 color 数据的格式：

```
use Color::Conversions qw(color_name_to_rgb rgb_to_color_name);

sub set_color {
    my $self = shift;
    my $new_color = shift;
    $self->{Color} = color_name_to_rgb($new_color); # arrayref
}

sub color {
    my $self = shift;
    $self->{Color} = rgb_to_color_name($self->{Color}); # takes arrayref
}
```

如下所示，如果使用 setter 和 getter，就仍然可以继续维持旧的接口，因为它们可以使用户无须了解所做的改变。也可以添加新接口，启用 RGB 三重值的直接设置和获取：

```
sub set_color_rgb {
    my $self = shift;
    $self->{Color} = [@_]; # set colors to remaining parameters
}

sub get_color_rgb {
    my $self = shift;
    @{$$self->{Color}}; # return RGB list
}
```

如果在类的外部使用代码直接查看 \$tv_horse->{Color} 的值，这种变化是不可能发生的。当它想要一个数组引用 ([0,0,255]) 时，或者使用数组引用作为字符串时，它将不会存储一个字符串（say，“blue”），或使用一个数组引用作为一个字符串。这就是 OOP 鼓励我们调用 setter 和 getter 的原因。

15.12 更快的 setter 和 getter

因为我们想要认真对待，并且总是调用 setter 和 getter 而不是窥视内部的数据结构，所以我们频繁调用 setter 和 getter。如下所示，为了节省很少的一点点时间，我们可能会将这些 setter 和 getter 改写为：

```
## in Animal
sub color { $_[0]->{Color} }
sub set_color { $_[0]->{Color} = $_[1] }
```

当我们这样做时，只能减少一点输入，而且代码也会运行略快，尽管与发生在程序中的其他一切相比，可能不足以引起我们的注意。\$_[0] 只访问 @_ 数组的单一元素，而不是使用 shift 语句将参数移到另一个变量中，我们可以直接使用它。

15.13 getter 作为双倍的 setter

创建两个不同的方法用于获取和设置一个参数的替代方案是创建一个方法，该方法将记录是否获取额外的参数。如果没有参数，它就是一个取值操作；如果有参数，它就是一个设值操作。如下所示，可以用一个简单的版本通过查看参数的个数来决定做什么：

```
sub color {
    my $self = shift;
    if (@_) { # are there any more parameters?
        # yes, it's a setter:
        $self->{Color} = shift;
    } else {
        # no, it's a getter:
        $self->{Color};
    }
}
```

现在可以使用相同的方法来获取或设置颜色：

```
my $tv_horse = Horse->named('Mr. Ed');
$tv_horse->color('black-and-white');
print $tv_horse->name, ' is colored ', $tv_horse->color, "\n";
```

第二行中参数存在表明我们正在设置颜色，而在第三行中参数空缺表明我们正在获取颜色。

这个策略是有吸引力的，因为它很简单，但它也有缺点。它使 getter 的行为变得更复杂，而且也被频繁调用。这也使我们很难搜索代码查找一个特定参数的 setter，这通常比 getter 更重要。在过去，当一个 getter 作为 setter 时，我们已经被折磨得焦头烂额，因为另一个函数在更新后，将返回比预期更多的参数。

15.14 仅仅限制一个类方法或者实例方法

设置一个难以形容并且通用的 Horse 类的名称或许不是一个好主意；在一个实例上调用 named 方法也不是。在 Perl 中，没有哪个方法定义的声明为：“这是一个类方法”或者“这是一个实例方法”，因为它们都只是 Perl 子例程。幸运的是，ref 操作符允许我们检查变量，判断当方法被错误调用时，是否应当抛出一个异常。如果仅仅作为实例或者类方法，考虑以下代码，我们通过检查参数来判断做什么：

```
use Carp qw(croak);

sub instance_only {
    ref(my $self = shift) or croak "instance variable needed";
    ... use $self as the instance ...
}

sub class_only {
    ref(my $class = shift) and croak "class name needed";
    ... use $class as the class ...
}
```

对于一个实例，ref 函数返回 true，实例只是一个被 bless 的引用；如果对于一个类，返回 false，因为此时此刻，类只是一个字符串。如果返回一个不希望得到的值，我们可以使用来自于 Carp 模块（这是标准发行版自带的一个核心模块）的 croak 函数。croak 函数会在看起来像是来自于我们调用方法的位置，而不是错误真正产生的位置，放置关于调用方函数生成的错误消息。如下所示，调用方会生成类似这样的错误消息，在调用错误方法的代码中提供行号：

```
instance variable needed at their_code line 1234
```

就像 croak 函数是另一种形式的 die 函数，Carp 模块提供 carp 函数作为 warn 函数的替代。每个都告诉用户在哪一行代码调用了产生问题的代码。可以使用 Carp 模块来代替模块中的 die 语句和 warn 语句。我们的用户将会为此感谢我们。

15.15 练习

可以在附录中的“第 15 章答案”部分找到这些练习的答案。

1. [5 分钟]给予 Animal 类能够获取和设置名称和颜色的能力。确保你的结果能在开启 use strict 语句的环境下工作。同时确保你的获取方法能够与通用的 animal 实例和特定的 animal 实例工作，使用如下语句测试代码：

```
my $tv_horse = Horse->named('Mr. Ed');
$tv_horse->set_name('Mister Ed');
$tv_horse->set_color('grey');
print $tv_horse->name, ' is ', $tv_horse->color, "\n";
print Sheep->name, ' colored ', Sheep->color, ' goes ', Sheep->sound, "\n";
```

如果要求你设置一个通用 animal 实例的名称或颜色，你将做些什么？

一些高级对象主题

你也许想知道，“是否所有对象都从一个普通的类继承得来？”“如果一个方法无法找到该怎么办？”“多重继承怎么实现？”或者“我如何知道我的对象类型？”无需多问，本章将涵盖这些主题并延伸更多相关知识。

16.1 通用方法

在定义类时，我们在每个包中通过全局变量@ISA 创建了继承层次。在查找一个方法的过程中，Perl 解释器会遍历@ISA 树，直到找到一个匹配的方法或查找失败。

然而，在查找失败后，Perl 解释器通常会在一个叫做 UNIVERSAL 的特殊类中查找，并调用其中的一个方法，如果成功匹配，则该方法就像位于任何其他类或超类中一样。

一种看待这种特性的解释为：UNIVERSAL 是从中派生所有对象的基类。可以将任何方法放在这里，例如：

```
sub UNIVERSAL::fandango {  
    warn 'object ', shift, " can do the fandango!\n";  
}
```

使程序的所有对象都可以被\$some_object->fandango 调用。

通常情况下，我们应当为感兴趣的特定类提供一个 fandango 方法，然后在 UNIVERSAL::fandango 中提供一个定义作为一个支撑方法，以防 Perl 解释器不能找到一个更明确的方法。实际中的一个示例可能是调试过程中的一个数据转储例程，或者可能是一个将所有应用对象转储到文件的编组策略。我们在 UNIVERSAL 类中提供通用方法，并且在特定类中为不常用的对象重写这个方法。

很明显，我们应当尽力避免使用 UNIVERSAL 模块，因为所有对象都在占用这一个空间，而且 fandango 方法可能会和其他加载进来并也调用 fandango 方法的模块相互冲突。因为这

个原因，除了那些必须完成的方法调用外，UNIVERSAL 模块几乎不会用在其他地方。基本上，如果不是像在调试过程中，或者其他想要了解 Perl 解释器的内部机制的情况下，普通程序员可以放心地忽略 UNIVERSAL 模块在其他地方的调用了。

16.2 为了更好的行为而测试对象

除了为我们提供一个空间来放置通用方法以外，UNIVERSAL 包同时预加载了两个非常有用的实际方法：DOES 和 can。因为 UNIVERSAL 定义了这些方法，所以它们能够被所有对象调用。



注意

DOES 方法在 Perl v5.10 以及后续版本中可用。对于在此之前的版本，可以调用 isa 来得到同样的东西，尽管 isa 仅仅只能测试继承关系。

DOES 方法用来测试查看一个给定的类或实例是否提供了某种特定角色，也就是一系列行为。如下所示，继续以之前章节中提到的 Animal 类为例：

```
use v5.10;

if ($tv_horse->DOES('Animal')) {    # does Horse do Animal?
    print "A Horse is an Animal.\n";
}

my $tv_horse = Horse->named("Mr. Ed");
if ($tv_horse->DOES('Animal')) { # is it an Animal?
    print $tv_horse->name, " is an Animal.\n";
    if ($tv_horse->DOES('Horse')) { # is it a Horse?
        print 'In fact, ', $tv_horse->name, " is a Horse.\n";
    } else {
        print "...but it's not a Horse.\n";
    }
}
```

当一个数据结构中混合有多种不同的对象并且想要区分各对象的特定分类时，如下操作就会很便捷：

```
use v5.10;

my @horses = grep $_->DOES('Horse'), @all_animals;
```

结果将仅仅是数组中的马（或赛马）。可以按照如下方式对比：

```
my @horses_only = grep ref $_ eq 'Horse', @all_animals;
```

这将只挑选出马，因为对 RaceHorse 类的 ref 操作将不返回 Horse 类。

通常情况下，我们将不会按照如下方式使用：

```
ref($some_object) eq 'SomeClass'
```

因为在之前给出的程序中，这样做会阻止将来的用户将这个类子类化，并且能够像它

更通用的基类一样调用这个类，于是使用了之前介绍过的 DOES 结构。

在此处 DOES 调用的一个缺点是，它只用于 blessed 的引用或者看起来像是类名的标量。如果我们碰巧传递给它一个没有 blessed 的引用，就将获得一个致命的（但是可捕获的）错误：

```
Can't call method "DOES" on unblessed reference at ...
```

为了使用更稳健的方式调用 DOES，可以以子例程的方式调用它：

```
if (UNIVERSAL::DOES($unknown_thing, 'Animal')) {  
    ... it's an Animal! ...  
}
```

无论\$unknown_thing 包含什么内容，像这样运行都不会再引发错误了。但这打破了 OO 机制，而且也有它自身的一些问题。这是一种用于异常机制的实现，就好像 eval 语句一样。如果\$unknown_thing 的值不是一个引用，那么我们将不能通过其来调用方法。eval 语句捕获这个错误并返回 undef，这意味着有错误，而这恰恰也是这种情况下的正确答案：

```
if (eval { $unknown_thing->DOES('Animal') }) {  
    ... it's an Animal ...  
}
```



注意

如果 Animal 类有一个自定义的 DOES 方法（也许它拒绝家族树里叫声突变的动物），跳过 Animal::DOES 而直接调用 UNIVERSAL::DOES，这样可能会给我们一个错误的答案。

在 DOES 的情况下，也可以调用 can 方法来测试可调用的行为。与诸如 DOES 的宽泛检查不同，can 只查询特定的方法而并不关心它们是如何定义的。例如：

```
if ($tv_horse->can('eat')) {  
    $tv_horse->eat('hay');  
}
```

如果 can 的返回值为真，那么在继承层次中的某些地方，其中的一个类可以调用 eat 方法。此外，关于\$tv_horse 的附加说明，要么是一个被 bless 的引用，要么是作为一个标量仍在应用的类名，因此，稳健的方案是尽可能多地使用 eval 来处理相应事务。

```
if (eval { $tv_horse->can('eat') }) { ... }
```

如果之前定义了 UNIVERSAL::fandango，那么这个方法的 can 检查将一律返回真，因为所有对象都可以调用 fandango 方法。

```
if( $object->can('fandango') ) { ... } # true for all objects
```



注意

can 和 DOES 有一样的问题：我们可以通过定义我们自己的方法来缩短与 UNIVERSAL 之间的路径。

16.3 最后的手段

在 Perl 解释器在继承树和 UNIVERSAL 中查找方法后，如果还不成功，查找将不会停止。Perl 解释器将重复搜寻相同的层次（包括 UNIVERSAL），查找一个叫 AUTOLOAD 的方法。

如果存在一个 AUTOLOAD 方法，将调用子例程来替换初始方法，并将预定的标准参数列表传递给它：类名或实例引用，后面跟着为方法调用提供的所有参数。初始方法名被传递到叫做 \$AUTOLOAD 的包变量中（在之前编译子例程的包里），并包含完全限定的方法名，所以，如果我们想要一个简单的方法名，一般就应当用包含最后双冒号的那个名字。

AUTOLOAD 子例程能执行本身期望的操作，安装一个子例程并跳转到里面，如果要求调用一个未知的方法，可能就会 die。

AUTOLOAD 的一个用处就是直到需要时才编译大型子例程。例如，假设一个 animal 的 eat 方法很复杂，而且在最近的程序调用中没有用到，我们可以推迟对它的编译，直到 Perl 解释器调用 AUTOLOAD。

```
## in Animal
use Carp qw(croak);

sub AUTOLOAD {
    our $AUTOLOAD;
    (my $method = $AUTOLOAD) =~ s/.*://s; # remove package name
    if ($method eq "eat") {
        ## define eat:
        eval q{
            sub eat {
                ...
                long
                definition
                goes
                here
                ...
            }
        };           # End of eval's q{} string
        die $@ if $@; # if typo snuck in
        goto &eat;    # jump into it
    } else {         # unknown method
        croak "$_[0] does not know how to $method\n";
    }
}
```

如果方法名是 eat，我们定义了 eat 方法，该方法之前已经储存在一个字符串内但并未编译；当有一个对 eat 的调用时，程序将跳转到一个调用了 eat 方法的特殊构造中来代替当前的 AUTOLOAD 子例程调用，就好像我们调用的是 &eat 而不是 AUTOLOAD^{注1}。

注1：尽管 goto 语句通常（并且一般情况下确实是）被认为是有害的，goto 语句的这种赋予一个子例程名字并将其作为跳转目标的应用形式，并不是有害的 goto 语句用法；这是一种有益处的 goto 语句用法。甚至可以说，这是“奇妙的 goto 语句”。这种技巧在于 AUTOLOAD 对子例程是完全不可见的。

在第一次 AUTOLOAD 匹配后，eat 子例程就定义了，因此我们不必重复相同的步骤。这就是按需编译程序的奇妙之处，因为这将最小化启动开销。

为了以更自动的方法创建代码来实现这些，我们很容易地在开发和调试过程中关闭自动载入功能，请参照 AutoLoader 和 SelfLoader 的核心模块文档。

16.4 使用 AUTOLOAD 创建访问器

第 15 章介绍了如何创建 color 和 set_color 方法来获取和设置一个动物的颜色。如果我们有 20 个属性，而不是一两个，就将会有很多令人痛苦的重复代码。通过使用一个 AUTOLOAD 方法，我们可以根据需要构造几乎相同的方法访问器，节省编译时间并为开发人员减轻负担。

我们使用一个代码引用作为一个闭包来完成该任务。首先，我们为对象设置一个 AUTOLOAD 方法，并为我们想要的小访问器定义一个散列键列表。

```
use Carp qw(croak);
sub AUTOLOAD {
    my @elements = qw(color age weight height);
```

然后，我们将判断该方法是否是其中一个键的 getter，如果是，则安装一个 getter 并跳转到它：

```
our $AUTOLOAD;
if ($AUTOLOAD =~ /:::(\w+)/$ and grep $1 eq $_, @elements) {
    my $field = ucfirst $1;
    {
        no strict 'refs';
        *$AUTOLOAD = sub { $_[0]->{$field} };
    }
    goto &$AUTOLOAD;
}
```

我们使用了 ucfirst 内置函数，因为我们将 color 用于获取散列元素的方法叫做 Color。typeglob 符号 (*{\$AUTOLOAD}) 安装了一个使用代码引用闭包定义的需要的子例程，用来获取对象散列中的相应的键。考虑到这部分代码十分巧妙，于是我们把该代码剪切并拷贝到程序里。最终，goto 语句结构跳转进了最新定义的子例程中。



注意

请阅读 *Mastering Perl* 来了解关于 typeglob 和符号表操作的更多知识。

否则，可能它就是一个 setter。

```
if ($AUTOLOAD =~ /::set_(\w+)/$ and grep $1 eq $_, @elements) {
    my $field = ucfirst $1;
    {
        no strict 'refs';
    }
```

```
*$AUTOLOAD = sub { $_[0]->{$field} = $_[1] };
}
goto &$AUTOLOAD;
}
```

如果它都不是，就会抛出异常：

```
(my $method = $AUTOLOAD) =~ s/.*/s; # remove package name
croak "$_[0] does not understand $method\n";
```

同样，我们只在第一次加载一个特定的 getter 或 setter 时对于 AUTOLOAD 产生开销。之后，如果一个相关子例程已经定义，我们就可以直接调用它。

16.5 更容易地创建 getter 和 setter

如果所有使用 AUTOLOAD 创建访问器的代码看起来有些凌乱，请放心，我们真的没有必要处理这个问题，因为有一个 CPAN 模块能够更直接地完成同样的操作：Class::MethodMaker^{注2}。

例如，一个简化版的 Animal 类可以定义如下：

```
package Animal;
use Class::MethodMaker
    new_with_init => 'new',
    get_set      => [-eiffel => [qw(color height name age)]],
    abstract      => [qw(sound)],
;

sub init {
    my $self = shift;
    $self->set_color($self->default_color);
}

sub named {
    my $self = shift->new;
    $self->set_name(shift);
    $self;
}
sub speak {
    my $self = shift;
    print $self->name, ' goes ', $self->sound, "\n";
}

sub eat {
    my $self = shift;
    my $food = shift;
    print $self->name, " eats $food\n";
}

sub default_color {
```

注 2：有时，使用 Class::MethodMaker 模块可能大材小用了。我们同样可以检验一下轻量级的 Class::Accessor 模块。

```
'brown';
}
```

4个实例属性（名称、高度、颜色和年龄）的 getter 和 setter 会自动定义，使用 get_color 方法来获取颜色，set_color 方法来设置颜色。（eiffel 语言的标志性语句是：“按照 Eiffel 语言的工作方式去做”，就是在里面的处理方式。）此时，容易让人混乱的 bless 步骤被隐藏到一个简单的 new 方法的后面。因为生成的 new 方法调用了 init 方法，所以与之前一样我们定义初始颜色为默认颜色。

然而，我们仍然可以调用 Horse->named ('Mr. Ed')，由于此时这种调用方法也立刻调用了 new 例程。

Class::MethodMaker 模块创建的 sound 方法作为一个抽象方法。抽象方法只是一个占位符，意味着将要在子类中定义。如果一个子类没有定义这个抽象方法，Class::MethodMaker 模块为 Animal 模块生成的 sound 方法将运行失败。

我们失去在类本身上调用 getter（例如 name）的能力，而不是在实例上失去。相应地，当它们调用了访问器时，会破坏我们对通用动物的 speak 和 eat 方法调用的优先使用权。如下所示，解决该问题的一个办法是定义 name 的一个更通用版本来处理一个类或实例，然后改变其他例程来调用它：

```
sub generic_name {
    my $either = shift;
    ref $either ? $either->name : "an unnamed $either";
}
sub speak {
    my $either = shift;
    print $either->generic_name, ' goes ', $either->sound, "\n";
}
sub eat {
    my $either = shift;
    my $food = shift;
    print $either->generic_name, " eats $food\n";
}
```

在此处，现在它看起来几乎与之前的定义兼容了，除了那些直接引用了在散列中属性名的友员类，与首字母大写版本一致（例如 Color），而未通过访问器（\$self->color）。

这也再次引起了维护问题。从接口（方法名称、参数列表或返回值的类型）的实现过程（使用散列或数组、散列键的名称，或元素的类型）中去耦合得越多，系统就会越灵活并且越可维护。

然而，灵活性并非不受约束。因为执行方法调用的成本是高于使用哈希查找的，所以在某些情况下，通过一个友员类窥视类的内部是很有意义的。

16.6 多重继承

Perl 解释器如何遍历@ISA 树？答案可能是简单的，也可能是复杂的。如果我们没有多重继承（也就是说，没有哪个@ISA 拥有超过一个元素），该问题是简单的：Perl 解释器可以从一个@ISA 遍历到下一个，直到找到@ISA 为空的最终基类。

多重继承更为复杂。它发生在当一个类的@ISA 有不止一个元素的时候。例如，假设有一个叫做 Racer 的类，它定义了任何能够快速行进（race）的事物的基本能力，所以它可以被用作一个跑步者、一辆快速行进的汽车或者一只快速行进的乌龟的基类。如下所示，可以很简单地构建 RaceHorse 类：

```
package RaceHorse;  
use parent qw{ Horse Racer };
```



注意

如果 Horse 和 Racer 中的方法之间有冲突，或者它们的实现过程不能够协调工作，这会变得很困难。在@ISA 中不同类可能不能够很好地一起运行，甚至可能相互影响彼此间的数据。

现在，一个 RaceHorse 类可以做一个 Horse 类和 Racer 类能做的任何事情。当 Perl 解释器查找一个不是由 RaceHorse 直接提供的方法时，它将首先查找完 Horse 类的所有方法（包括它所有的父类，例如 Animal 类）。当在 Horse 类中也不可能找到时，Perl 解释器将转去查找 Racer 类（或者它的一个父类）是否能够提供需要的方法。另一方面，如果我们希望 Perl 解释器在查找 Horse 类之前就查找 Racer 类，把它们按照这样的顺序放进@ISA（见图 16-1）。

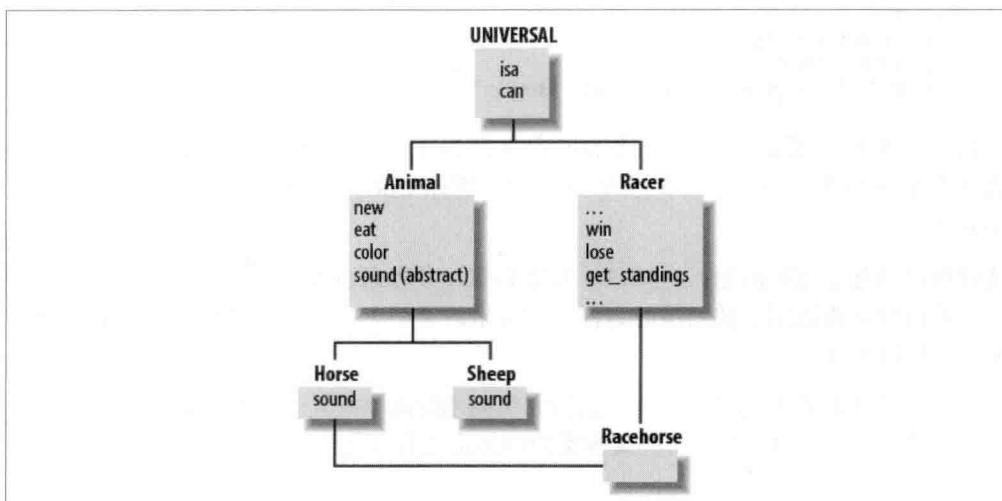


图 16-1 要是一个类通过多重继承从它的父类中来继承所需要的一切，这个类可能不需要实现其自身的任何方法

16.7 练习

可以在附录中的“第 16 章答案”部分找到这些练习的答案。

1. [20 分钟] 写一个名为 MyDate 的模块，该模块有一个 AUTOLOAD 方法，可以用来处理对名字为 day、month 和 year 的方法的调用，每一个调用方法都返回相应的属性值。对其他的方法，AUTOLOAD 方法将调用 Carp 模块来处理未知的方法名。使用你的模块写一个脚本，并且能够输出 date、month 和 year 的值。
2. [20 分钟] 从为上一个练习写的脚本开始，添加一个 UNIVERSAL::debug 函数，该函数能够在你传递给它消息之前输出一个时间戳。对 MyDate 对象调用 debug 方法，看看发生了什么？它是如何绕过 AUTOLOAD 机制的？

第 17 章

Exporter

第 2 章介绍了如何使用模块，其中的一些模块将模块中定义的函数导入当前命名空间中。本章将介绍如何在我们自己的模块中实现这个功能。

17.1 use 语句在做什么

那么，use 语句做了些什么呢？导入列表是如何开始运作的？Perl 解释器将 use 语句后面的列表解释为一种特殊形式的 BEGIN 语句块，该语句块包裹着一条 require 语句和一个方法调用。下面两个操作是等价的：

```
use Island::Plotting::Maps qw( load_map scale_map draw_map );  
  
BEGIN {  
    require Island::Plotting::Maps;  
    Island::Plotting::Maps->import( qw( load_map scale_map draw_map ) );  
}
```

首先，这里的 require 语句是一个对于包名的 require 语句，而不是第 11 章介绍的对于字符串表达式的 require 语句。冒号在类 UNIX 系统中被转换成目录分隔符，最终将以如下方式显示：

```
require "Island/Plotting/Maps.pm";
```



注意

我们不能使用前面使用的 .pl（即 Perl library 的缩写）作为扩展名，因为 use 语句找不到它。它只能使用 .pm 扩展名。

回想一下之前 require 语句的操作，这意味着 Perl 解释器将查找当前 @INC 数组中的值，按顺序检查每个目录，查找一个叫 Island 的子目录中的 Plotting 子目录中包含名为 Maps.pm 的文件。

如果 Perl 解释器查找 @INC 数组后还找不到指定的文件，程序就由于查询不到所调用

的库而终止(这可通过 eval 捕获)。否则,Perl 读取并判断它读取的第一个文件。与 require 语句类似,最后一个表达式的值必须为真,否则程序将认为它遇到编译文件的错误而退出。然后,一旦 Perl 解释器读取文件,如果不再次要求它,它就将不会再次读取。在模块的接口层面上,我们期望在 package 关键字后面定义的名称与所在的文件同名,然后将子例程定义在 package 后面。例如,如果我们取出所有子例程的内容,File::Basename 模块文件的一部分可能如下所示:

```
package File::Basename;
sub dirname { ... }
sub basename { ... }
sub fileparse { ... }
1;
```

这三个子例程定义在 File::Basename 包内,并不是在 use 语句出现的包中。

这些子例程是如何从模块的包进入被调用的包中呢?这就是在 BEGIN 块中的第二步:Perl 解释器在模块的包里自动调用一个叫做 import 的例程,传入整个导入列表。通常情况下,这个例程将对导入的命名空间中的一些名称指定别名(例如:File::Basename 模块),然后映射到当前的命名空间之中(例如 main)。模块的作者负责提供一个适当的导入例程列表。实际上比听起来容易些,这将在本章后续部分详细介绍。

最后,整个过程被封装在一个 BEGIN 语句块内。这意味着 use 语句的操作在编译时间内生效,而不是运行时,并且确实如此。因此,那些在模块中定义的子例程和原型被妥善地关联映射到当前的程序中。

17.2 使用 Exporter 模块导入子例程

在第 2 章中,当我们在 import 例程中将那个 File::Basename::fileparse 子例程导入到调用者的包中,并且设置别名为 fileparse 时,我们跳过了“见证魔法”的部分。

Perl 解释器提供了很多内省的功能。具体来说,我们可以查看符号表(在里面命名所有子例程和许多变量),来看看定义了什么,然后改变这些定义。在之前第 16 章介绍 AUTOLOAD 机制时,我们已经介绍过一些内容。如果我们是 File::Basename 模块的作者,我们想要将当前包中的 filename、basename 和 fileparse 名称传递到调用包中,我们可以写 import 子例程来实现:

```
sub import {
    for (qw(filename basename fileparse)) {
        no strict 'refs';
        *{"main::$_"} = \&$_;
    }
}
```

同学们,这很神秘吧!但有限制。如果调用者不想要 fileparse 该怎么办呢?如果调用者在一个不是 main 的包中调用呢?

谢天谢地,在 Exporter 模块中有一个标准的 import 方法。对于模块的编写者而言,我

们只需要将 Exporter 模块添加为父类就可以了：

```
package Animal::Utils;
use parent qw(Exporter);
```

现在使用 import 调用包名的语句将向上继承到 Exporter 的类，然后提供一个 import 例程，该 import 例程知道将如何使用一个子例程列表^{注1}，并且将这个列表导出到调用者的包中。事实上，现在很多人就是这么做。

我们并不是真的想要直接从 Exporter 模块继承。类通常不会成为 Exporter 模块的一个特殊实现，因此使用 ISA 的关系是不对的。事实上，可以不通过继承就直接导入 import 子例程，如下所示，如果使用 v5.8.3 或者后续版本：

```
use v5.8.3;
package Animal::Utils;
use Exporter qw(import);
```

我们更喜欢后者，但我们在代码中看到了这两种写法。

17.3 @EXPORT 和@EXPORT_OK

由 Exporter 模块提供的 import 子例程将检查模块包中的@EXPORT 变量，判断哪个符号它默认导出。如下所示，File::Basename 模块可能是按照这样的方式工作：

```
package File::Basename;
use Exporter qw(import);
our @EXPORT = qw( basename dirname fileparse );
```

@EXPORT 列表不但定义可以用来导入的符号（公共接口），而且为 Perl 解释器提供一个默认列表，用于当我们没有指定导入列表时。如下两个调用是等效的：

```
use File::Basename;

BEGIN { require File::Basename; File::Basename->import }
```

我们没有传递列表到 import 例程，因此 Export->import 例程将查询@EXPORT 变量，然后提供该列表中的全部内容。记住，没有导入列表与导入空列表是不同的。如果导入的列表为空，模块的 import 方法就根本没有调用。

如果我们有一些子例程并不想将它们作为默认导入列表的一部分，但是如果我们想要调用，这些子例程仍然存在，该如何实现？我们可以添加这些子例程到模块的包中的@EXPORT_OK 列表里面。例如，假如 Gilligan 的模块默认提供 guess_direction_toward 例程，但也想提供 ask_the_skipper_about 例程和 get_north_from_professor 例程，如果需要，可以按照如下方式实现：

```
package Navigate::SeatOfPants;
use Exporter qw(import);
our @EXPORT = qw(guess_direction_toward);
our @EXPORT_OK = qw(ask_the_skipper_about get_north_from_professor);
```

如下调用将是有效的：

注 1：以及变量，尽管没有那么通用，并且会有争议地去做错的事情。

```

use Navigate::SeatOfPants; # gets guess_direction_toward
use Navigate::SeatOfPants qw(guess_direction_toward); # same

use Navigate::SeatOfPants
qw(guess_direction_toward ask_the_skipper_about);

use Navigate::SeatOfPants
qw(ask_the_skipper_about get_north_from_professor);
## does NOT import guess_direction_toward

```

如果指定任何例程的名称，它们必须要么在@EXPORT 列表中，要么在@EXPORT_OK 列表中，因此，如下调用将会被 Export->import 子例程拒绝。

```
use Navigate::SeatOfPants qw(according_to_GPS);
```

因为 according_to_GPS 子例程既不属于 @EXPORT 列表，也不属于 @EXPORT_OK 列表^{#2}。因此，通过使用这两个列表，可以控制公共接口。但这并不能阻止某些人用 Navigate::SeatOfPants::according_to_GPS 的方式调用（如果它存在），但至少在现在，很明显大家都使用我们没有刻意提供的一些内容。

17.4 使用%EXPORT_TAGS 分组

我们也不必列出我们想要导入的每一个函数或者变量。我能够创建快捷方式或者标签，将它们用一个名字统称。在导入列表中，在标签前面放置一个冒号。例如，如下所示，核心模块 Fcntl 可以使用：“flock” 标签作为一组来导入 flock 常量变量：

```
use Fcntl qw( :flock );          # import all flock constants
```

正如 Exporter 模块文档所述，少量快捷方式是自动存在的。DEFAULT 标签将导入与我们没有提供导入列表所导入的相同内容：

```
use Navigate::SeatOfPants qw(:DEFAULT);
```

这样并不是十分有用，但是如果我们要导入默认符号以及更多的内容，我们就不必输入所有名称，因为我们可以按照如下方式提供导入列表：

```
use Navigate::SeatOfPants qw(:DEFAULT get_north_from_professor);
```

这样的方式在实际代码中很少见。为什么？显式地提供导入列表的目的，通常意味着我们想要控制在程序中使用的子例程名称。最后这些示例并没有使我们与模块将来的变更相互隔离，模块可能导入额外与代码相互冲突的子例程。



注意

更新一个已发布模块来引入新的默认导入子例程，通常被认为是一个很糟糕的主意。如果我们知道我们的第一个发行版仍然缺少一个函数，然而，我们仍然没有理由不放置一个占位符：sub according_to_GPS { die "not implemented yet" }。

注 2：这个检查也能捕获拼错和错误的子例程名称，使我们不再琢磨为什么 get_direction_from_professor 例程不能工作。

在一些情况下，模块可能提供几十个或者几百个可能的符号。这些模块能够使用更高级的技术（在 Exporter 模块文档中介绍）使导入多个相关的符号变得更容易。

在模块中，使用%EXPORT_TAGS 散列来定义这些标签。如下所示，散列键是标签的名称（没有冒号），值是一个符号数组引用：

```
package Navigate::SeatOfPants;
use Exporter qw(import);

our @EXPORT      = qw(guess_direction_toward);
our @EXPORT_OK   = qw(
    get_north_from_professor
    according_to_GPS
    ask_the_skipper_about
);

our %EXPORT_TAGS = (
    all      => [ @EXPORT, @EXPORT_OK ],
    gps     => [ qw( according_to_GPS ) ],
    direction => [ qw(
        get_north_from_professor
        according_to_GPS
        guess_direction_toward
        ask_the_skipper_about
    ) ],
);

```

第一个标签 all 包括所有可导出的符号（在@EXPORT 列表和@EXPORT_OK 列表中的所有内容）。gps 标签仅仅包括所有处理 GPS 的函数，direction 标签包括所有处理方向的函数，标签也可以包含重叠项，我们会注意到 according_to_GPS 将出现在每一个中。无论如何定义标签，所有包含的子例程要么在@EXPORT 中，要么在 @EXPORT_OK 之中。

如下所示，一旦定义导出标签，用户就可以在导入列表中使用它们：

```
use Navigate::SeatOfPants qw(:direction);
```

17.5 定制导入例程

在介绍如何编写我们自己的模块之前，我们将使用 CGI.pm 模块来介绍定制导入程序的方法。CGI.pm 模块的作者 Lincoln Stein 并不满足于 Exporter 模块的 import 例程不可思议的灵活性，他为 CGI 模块创建了一个特殊的 import 方法^{注3}。如果你曾经呆呆地看着出现在 use CGI 语句后面令人眩晕的选项数组，而且想知道这个选项数组的工作方式，这只是一个简单的编程问题。我们总是能够查看它们的源代码。

可以将 CGI 模块作为一个面向对象模块使用：

注3：一些人把这个称为“Lincoln 加载器”，同时对 Lincoln 致以深深敬意，主要是对于必须处理他们所不熟悉的东西所产生的纯粹恐惧。

```
use CGI;
my $q = CGI->new;           # create a query object
my $f = $q->param('foo'); # get the foo field
```

通过 Lincoln 的定制 import，我们也可以将 CGI 模块当成一个面向函数的模块^{注4}：

```
use CGI qw(param);          # import the param function
my $f = param('foo');       # get the foo field
```

如果我们不想拼写所有可能用到的子例程，可以这样写：

```
use CGI qw(:all);           # define 'param' and 800-gazillion others
my $f = param('foo');
```

在这里编译开关还是有效的。例如，我们想禁用正常黏着字段处理，只需要添加一个-nosticky 到导入的列表中即可：

```
use CGI qw(-nosticky :all);
```

如果除了其他例程之外我们想要创建 start_table 和 end_table 例程，可以按照如下方式处理：

```
use CGI qw(-nosticky :all *table);
```

不仅如此，我们可以创建新的方法来生成 HTML。定制 import 例程将它不能识别的导入转换为便利的 HTML 功能：

```
use CGI qw(foo bar);

print foo( 'Hello!' );      # <foo>Hello!</foo>
```

真正让人眼花缭乱的选项数组。Lincoln 是怎么让这一切工作起来的？我们可以自己去查看 CGI 模块的源代码，但这里只介绍最基本的内容。

import 方法是一个常规的方法，所以我们可以用这个方法做我们想要做的任何事情，早些时候，我们介绍了一个关于 File::Basename 模块的简单（虽然是假设的）示例。在这种情况下，不使用 Exporter 模块中的 import 方法作为真实模块中的方法，我们编写了我们自己的方法强制将符号导入 main 包中：

```
sub import {
    foreach my $name (qw(filename basename fileparse)) {
        no strict 'refs';
        *{"main::$name"} = \&$_;
    }
}
```

这只适用于 main，因为这已经硬编码到例程之中。然而，可以通过使用内置的 caller 函数在运行中找出所调用的包。在标量上下文中，caller 函数返回正在调用的包名称：

```
sub import {
    my $package = caller;
    foreach my $name (qw(filename basename fileparse)) {
        no strict 'refs';
        *{$package . "::$name"} = \&$_;
    }
}
```

注 4：还有一个对象在幕后，但我们没有看到它。

我们能通过列表上下文从 caller 函数中获取更多信息：

```
sub import {
    my( $package, $file, $line ) = caller;
    warn "I was called by $package in $file\n";
    for (qw(filename basename fileparse)) {
        no strict 'refs';
        ${$package . "::$_"} = \&$_;
    }
}
```

因为 import 是一个方法，所以它的任何参数（请记住是导入列表）都出现在 @_ 数组中，我们可以检查参数列表，然后决定要做什么。仅当 debug 出现在导入列表中时，才打开调试输出。我们不会导入一个命名为 debug 的子例程。如果有\$debug 参数，我们仅仅设置\$debug 的参数为 true。然后做一些之前做过的事情。此时此刻，如果打开 debug 选项，仅仅输出警告消息：

```
sub import {
    my $debug = grep { $_ eq 'debug' } @_;
    my( $package, $file, $line ) = caller;
    warn "I was called by $package in $file\n" if $debug;
    for (qw(filename basename fileparse)) {
        no strict 'refs';
        ${$package . "::$_"} = \&$_;
    }
}
```

这些是 Lincoln 在他的 CGI 魔法中运用的基本技巧，而且它们和 Test::More 模块是同样的东西，第 14 章已经介绍过这部分内容，使用自己的 import 函数来设置测试计划。

17.6 练习

可以在附录的“第 17 章答案”部分找到这些练习的答案。

1. [15 分钟] 把在第 11 章的练习 1 中创建的 Oogabooogoo 库变成一个模块，你可以直接用 use 语句导入。改变调用代码，以便它使用导入的例程（而不是像你之前那样使用完整的包规范），并测试它。
2. [15 分钟] 修改练习 1 的答案以使用 all 导出标签，当用户使用 all 时，你的模块应导入所有子例程名：

```
use Oogabooogoo::date qw(:all);
```
3. [10 分钟] 修改在第 14 章中创建的 My::List::Util 模块，以便它导出其 sum 和 shuffle 子例程。

对象析构

第 13 章和第 15 章介绍对象创建和操作的基本知识。本章将介绍一个同等重要的主题：对象运行结束后发生了什么。在 Perl 解释器内部，调用一个清理对象的方法来销毁它。

正如第 5 章所述，当最后一个指向 Perl 中某一数据结构的引用消失时，Perl 解释器将会自动回收相应数据结构所占的内存，包括销毁该数据结构任何指向其他数据的链接。当然，这可能进而导致 Perl 解释器也销毁其他结构，比如该结构当中包含的结构。

默认情况下，对象以这种方式工作，因为对象使用与基于引用计数的垃圾收集机制相同的方式来构造更复杂的对象。如果一个对象由散列引用构建，一旦最后一个指向该对象的引用消失，Perl 解释器将销毁这个由散列引用构建的对象。如果散列值也是引用，它们也同样被删除，这可能造成进一步的析构。

18.1 清理

假如对象使用一个临时文件来保存数据，这些数据适合完全放置在内存中。此时对象可以在它的实例数据中包含一个指向临时文件的文件句柄。而正常的对象析构过程将关闭这个文件句柄，除非我们采取进一步操作，否则在磁盘上这个临时文件一直存在。

当 Perl 解释器销毁一个对象时，需要执行一些适当的清理操作，所以我们需要知道什么时候发生该操作。谢天谢地，如果有相应请求，Perl 解释器就会根据请求发出这样的通知。我们通过给对象一个 DESTROY 方法来请求这个通知。

当指向对象的最后一个引用消失时，以\$bessie 为例，Perl 解释器自动调用该对象的 DESTROY 方法，就像我们主动调用这个方法一样。

```
$bessie->DESTROY
```

这个方法调用与其他大多数方法调用类似：Perl 解释器从对象所属的类出发并沿着继承层次结构向上直到它找到合适的方法。但是，与调用其他方法不同的是，如果 Perl 解释器没有找到一个合适的方法，并不会产生错误^{注1}。



注意

通常情况下，如果 Perl 解释器没有找到这个方法，就会在我们自己的方法调用过程中产生错误。可以在基类中定义一个空方法来防止这种现象发生。

例如，回到第 13 章中所定义的 Animal 类，我们可以单纯为调试程序而给这个类添加一个 DESTROY 方法，这样就可以知道对象什么时候消失。

```
## in Animal
sub DESTROY {
    my $self = shift;
    print '[', $self->name, " has died.]\n";
}
```

此时此刻，当在程序中创建 Animal 类时，当对象销毁时，我们就可以获得相应的提示信息。例如：

```
## include animal classes from previous chapter...

sub feed_a_cow_named {
    my $name = shift;
    my $cow = Cow->named($name);
    $cow->eat('grass');
    print "Returning from the subroutine.\n";    # $cow is destroyed here
}
print "Start of program.\n";
my $outer_cow = Cow->named('Bessie');
print "Now have a cow named ", $outer_cow->name, ".\n";
feed_a_cow_named('Gwen');
print "Returned from subroutine.\n";
```

这个程序的输出结果为：

```
Start of program.
Now have a cow named Bessie.
Gwen eats grass.
Returning from the subroutine.
[Gwen has died.]
Returned from subroutine.
[Bessie has died.]
```

从上面的程序可以看出，Gwen 是子例程当中的一个变量。然而，当子例程退出后，一旦 Perl 解释器发现不再有指向 Gwen 的引用，它就会自动调用 Gwen 的 DESTROY 方法，输出 Gwen has died 消息。

在一个程序运行的最后发生了什么？由于在程序结束时对象应不再存在，因此 Perl 解

注 1：import 和 unimport 方法也很特殊。

释器最终会忽略所有已有的数据并且销毁它们。无论这些数据是以词法变量还是全局变量的形式存在。因为在程序末尾 Bessie 这个变量还存在，则该变量就需要被回收，所以在程序中所有其他步骤完成之后，我们将获得关于 Bessie 的消息。



注意

Perl 解释器在程序执行完最后一个 END 语句块标志时立即执行最后的清理工作，对于 END 语句块也遵循同样的规则：程序必须正常结束而不是突然终结，如果程序由于超出内存等原因而非正常结束，所有约定都不再存在。

18.2 嵌套对象析构

如果一个对象包含另外一个对象（也就是说，作为一个数组元素或者散列元素的值），Perl 解释器首先销毁这个对象再销毁这个对象中包含的其他对象。这是容易理解的，因为所包含对象正常销毁可能需要用到指向所包含内容的引用。如下所示，为了演示这一特点，我们构建一个“barn”然后销毁它。有趣的是：我们将“barn”构建为被 bless 的数组引用而不是散列引用。

```
{ package Barn;
sub new { bless [ ], shift }
sub add { push @{$shift()}, shift }
sub contents { @{$shift()} }
sub DESTROY {
    my $self = shift;
    print "$self is being destroyed...\n";
    for ($self->contents) {
        print ' ', $_->name, " goes homeless.\n";
    }
}
```

在此处，我们在定义对象时做到尽量简约。在创建一个新的 barn 类时，我们 bless 一个空的数组引用到类名中，作为第一个参数。添加一个 animal 并将其放到 barn 类的最后。如果需要访问 barn 类的内容，只需对数组引用对象进行解引用操作并且返回其内容。

接下来介绍最有趣的析构函数。我们对自身取引用，在一个特定的 barn 对象销毁时显示出一条调试消息。接下来，依次访问 barn 类中所包含的每一个对象。如下所示，实际代码为：

```
my $barn = Barn->new;
$barn->add(Cow->named('Bessie'));
$barn->add(Cow->named('Gwen'));
print "Burn the barn:\n";
$barn = undef;
print "End of program.\n";
```

得到的程序输出为：

```
Burn the barn:  
Barn=ARRAY(0x541c) is being destroyed...  
    Bessie goes homeless.  
    Gwen goes homeless.  
[Gwen has died.]  
[Bessie has died.]  
End of program.
```

注意，Perl 解释器首先销毁 barn 类，使我们能够清楚地获得 barn 类中每一个对象的名称。当 barn 销毁后，这些对象也就没有了额外的引用，它们也要消失，因为 Perl 解释器也调用它们的析构函数。就好比牛（cow）毕生都在牛棚（barn）之外一样。

```
my $barn = Barn->new;  
my @cows = (Cow->named('Bessie'), Cow->named('Gwen'));  
$barn->add($_) for @cows;  
print "Burn the barn:\n";  
$barn = undef;  
print "Lose the cows:\n";  
@cows = ();  
print "End of program.\n";
```

得到的程序输出为：

```
Burn the barn:  
Barn=ARRAY(0x541c) is being destroyed...  
    Bessie goes homeless.  
    Gwen goes homeless.  
Lose the cows:  
[Gwen has died.]  
[Bessie has died.]  
End of program.
```

在上面的代码中，cow 对象将一直存在直到唯一一个指向牛的引用（存在于数组 @cows 中）消失。

一旦 barn 的析构函数调用完成，指向 cow 的引用就会消失。在某些情况下，我们反而需要把 cow 从 barn 类中提取出来，作为我们单独关注的对象。此时此刻，最简单的是破坏性地修改 barn 数组，而不是遍历它。如下所示，将 Barn 改变为 Barn2 来详细介绍：

```
{ package Barn2;  
sub new { bless [ ], shift }  
sub add { push @{$shift()}, shift }  
sub contents { @{$shift()} }  
sub DESTROY {  
    my $self = shift;  
    print "$self is being destroyed...\n";  
    while (@$self) {  
        my $homeless = shift @$self;  
        print ' ', $homeless->name, " goes homeless.\n";  
    }  
}
```



注意

如果我们使用的是散列，我们就在希望立即处理的元素上使用 delete 内置函数。

现在将散列用到前面的场景当中：

```
my $barn = Barn2->new;
$barn->add(Cow->named('Bessie'));
$barn->add(Cow->named('Gwen'));
print "Burn the barn:\n";
$barn = undef;
print "End of program.\n";
```

得到的程序输出为：

```
Burn the barn:
Barn2=ARRAY(0x541c) is being destroyed...
    Bessie goes homeless.
[Bessie has died.]
    Gwen goes homeless.
[Gwen has died.]
End of program.
```

在 barn 消失的时候，Bessie 被立即赶出而没有了生存的场所，所以她也就消失了（可怜的 Gwen 遭受了同样的厄运）。在此时并没有指向她的引用，甚至在 barn 的析构函数完成之前都没有。

接下来，我们回到临时文件的问题：我们使用标准库的 File::Temp 模块提供的临时文件来修改 Animal 类。该模块的 tempfile 例程知道如何生成临时文件，包括在哪儿放置，等等，所以我们并不需要自己完成。tempfile 函数返回一个文件句柄和一个文件名，而且我们需要同时存储它们，因为在析构函数中同时用到。

```
## in Animal
use File::Temp qw(tempfile);

sub named {
    my $class = shift;
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    ## new code here...
    my ($fh, $filename) = tempfile();
    $self->{temp_fh} = $fh;
    $self->{temp_filename} = $filename;
    ## .. to here
    bless $self, $class;
}
```

现在，我们已经有了文件句柄和对应的文件名，以 Animal 类或其派生类的实例变量来存储。如下所示，在析构函数中，关闭文件句柄并且删除（unlink）文件。

```
sub DESTROY {
    my $self = shift;
    my $fh = $self->{temp_fh};
    close $fh;
    unlink $self->{temp_filename};
    print "[", $self->name, " has died.]\n";
}
```

当 Perl 解释器销毁 Animal 类对象的最后一个引用时（甚至在程序的末尾），它将会自动删除临时文件以避免混乱。



注意

在最终，可以使用 File::Temp 模块来自动完成这些工作，但是，这样我们就不能手动展示它的工作细节。手动完成将需要我们做一些额外的处理，比如把总结信息从一个临时文件存储到一个数据库之中。

18.3 终结一个“死去”的 Horse 类

因为子类会像继承超类的其他方法一样继承超类的 DESTROY 方法，所以我们也可以重写和扩展超类的方法。例如，我们想要进一步使用死去的马。在 Horse 类中，重写了继承自 Animal 类的 DESTROY 方法，从而进行一些额外的处理。然而，因为 Animal 类可能会做一些意想不到的事情，所以通过第 13 章介绍的 SUPER::伪类来调用它的 DESTROY 版本。

```
## in Horse
sub DESTROY {
    my $self = shift;
    $self->SUPER::DESTROY if $self->can( 'SUPER::DESTROY' );
    print "[", $self->name, " has gone off to the glue factory.]\n";
}

my @tv_horses = map { Horse->named($_) } ('Trigger', 'Mr. Ed');
$_->eat('an apple') for @tv_horses;      # their last meal
print "End of program.\n";
```

上面的程序将会输出：

```
Trigger eats an apple.
Mr. Ed eats an apple.
End of program.
[Mr. Ed has died.]
[Mr. Ed has gone off to the glue factory.]
[Trigger has died.]
[Trigger has gone off to the glue factory.]
```



注意

在调用 SUPER::DESTROY 之前，我们需要检查是否可以调用它。虽然我们并没有定义这个方法，但是在 Perl 对其进行隐式调用的时候并不会产生不好的结果。如果显式地调用 DESTROY 方法，该方法必须定义。

我们会让马吃最后一顿“晚餐”，在程序结束时，每一个 horse 对象的析构函数将会被调用。

该析构函数的第一步是调用其父类的析构函数。为什么这一点尤为重要？如果没有调

用父类的析构函数，在超类中所需要的清理工作将不会正确执行。如果这只是之前展示过的调试语句，它不会有太大的影响。但是如果它是删除临时文件的清理方法，那么不调用父类的析构函数将不会对文件进行删除。

所以，在编写这类程序时需要遵守的规则为在析构函数中总是包含一个调用\$self->SUPER::DESTROY 的方法（即使这个类没有基类或者父类）。

在定义的析构函数的开始还是末尾调用上述方法是一个值得激烈争论的问题。如果派生类需要使用超类当中的一些实例变量，我们需要在所有的工作结束之后来调用超类的析构函数，因为超类的析构函数会使用令人厌烦的方式改变这些需要使用的变量。另一方面，例如，在添加行为之前调用超类的析构函数，因为我们想要先实现超类的行为。

18.4 间接对象表示法

通过箭头语法来调用一个方法通常被称为是直接对象语法^{#2}。因为也存在间接对象语法，间接对象语法也被视为“只在某些情况下可行”的语法，我们将在随后介绍其中的原因，我们可以通过箭头符号替换我们之前所写的一些语句：

```
Class->class_method(@args);
$instance->instance_method(@other);
```

调用的方法名放在类名的前面，方法所需要的参数放在后面：

```
class_method Class @args;
instance_method $instance @other;
```

这种习惯用法在 Perl v5 早期非常流行，而且我们仍然试图消除这种做法。我们没有必要在此详细介绍这种做法，因为如果你不了解也就不会使用这种做法了。但是，当别人这样使用时，你需要能够识别。这个符号出现在以前的一些好的代码与文档之中，所以你需要了解相应的做法代表什么意思。如下所示，我们希望你总是使用箭头符号：

```
my $obj = Some::Class->new(@constructor_params);
```

但是，如下所示，一些 Perl 程序员在语句前放置 new 符号，使语句更加容易理解。

```
my $obj = new Some::Class @constructor_params;
```

当然，这样的语法会使得 C++ 程序员感到很自然。在 Perl 语言中，对于 new 来说，并没有什么特别值得注意的地方，但是这样的语法会使得大部分人感到熟悉。

当我们可以用间接对象语法替换箭头语法时，为什么通常会发出警告？举个例子，当一个实例是一个比简单的标量变量更加复杂的变量时：

```
$somehash->{$somekey}->[42]->instance_method(@parms);
```

在间接对象表示法中不可以交换它的顺序。

注 2：我们也可以将它称为语法中的与格（与格，或谓第 3 格，是指名词的语法上的格）。但是并没有太多人使用这个术语。

```
instance_method $somehash->{$somekey}->[42] @parms;
```

间接对象语法中唯一可以接受的只有裸字（例如，一个类名），一个简单的标量变量或者表示一个语句块的大括号返回一个被 bless 的引用或者类名^{注3}。这就意味着我们必须像在第 8 章中介绍的文件句柄引用的方式写入：

```
instance_method { $somehash->{$somekey}->[42] } @parms;
```

从简单到丑陋只有一步之遥。此外还有另外一个缺点：二义性的解析。当我们在开发涉及间接对象引用的课堂培训材料时，写出如下代码：

```
my $cow = Cow->named('Bessie');  
print name $cow, " eats.\n";
```

因为我们认为间接对象与下面的语句等价：

```
my $cow = Cow->named('Bessie');  
print $cow->name, " eats.\n";
```

然而，后面的代码能够正常运行，而前面的则不能。我们将不会得到任何输出。最后，我们打开警告，就会得到类似于如下的一系列有趣的消息。

```
Unquoted string "name" may clash with future reserved word at ./foo line 92.  
Name "main::name" used only once: possible typo at ./foo line 92.  
print( ) on unopened filehandle name at ./foo line 92.
```



注意

当 Perl 在做出我们不理解的事情时，首先想到的应该是打开警告。或者我们应该考虑在第“零”步打开，因为我们最好无论何时在编写代码时，都使警告生效。

因此，如下所示，没有输出的那一行被解析为：

```
print name ($cow, " eats.\n");
```

换句话说，输出 name 文件句柄的列表项。这显然并不是我们所希望的。所以，我们必须增加额外的语法来消除这个调用的二义性。



注意

二义性出现的原因是 print 语句自身就是一个由文件句柄调用的方法。你可能通常把它看作是一个函数，但是记住需要在文件句柄后省略掉的逗号。这就与间接对象调用语法十分相似，因为这就是。这最终给出下一个十分重要的建议：无论何时都使用箭头语法。

我们认识到，虽然一些人习惯写成 new Class… 而不是 Class->new(…)，但是我们依旧容许这种做法。老的模块更倾向于在它们的代码示例中使用这个符号，一旦你习惯了这种写法，就继续保持下去。然而，在某些情况下这样依旧会产生二义性（例如，当

注 3：聪明的读者会注意到这些都和间接文件句柄语法的规则相同，就好像间接对象语法是从这些语法中照搬过来的一样，与解引用一个指定引用的规则也相同。

存在一个叫作 new 的例程时，这个类本身的名字就不会被视为一个包）。如果存在这种二义性，就不使用间接对象语法。维护你程序的程序员会感谢你。

18.5 子类中的额外实例

运用散列表这个数据结构的好处，是派生类可以在超类不知情的情况下增加额外的实例变量。例如，我们从 Horse 类中派生出 RaceHorse 子类，子类除了继承超类所有的信息外还需要追踪赛马比赛的结果。如下所示，其中的第一部分是不重要的。在 lib/RaceHorse.pm 中创建 RaceHorse 这个子类。

```
package RaceHorse;
use parent qw(Horse);
```

当创建一个新的 RaceHorse 类时，我们希望初始化这个对象没有参加任何比赛，当然也就没有获胜过。我们通过扩展 named 子例程以及添加四个字段来实现（这四个字段分别为 wins、places、shows 和 losses，用于表示为第一、第二、第三和以上均不是的结果）。

```
package RaceHorse;
use parent qw(Horse);
## extend parent constructor:
sub named {
    my $self = shift->SUPER::named(@_);
    $self->{$_} = 0 for qw(wins places shows losses);
    $self;
}
```

这里，向超类传递所有的参数，它可以返回一个完整的 Horse 对象。然而，因为我们作为类传递 RaceHorse，所以也将其 bless 进入 RaceHorse 类中。当在第 13 章中作为类传递 Horse 时，这与使用 Animal 类的构造函数来创建 Horse 类，而不是 Animal 类是相类似的。接下来，添加了超出超类定义范围的 4 个实例变量，将它们的初值均设置为 0。最后，为调用者返回修改过的 RaceHorse 类。

在这里应当注意的一点是，我们在编写该派生类时，实际上已经“打开了一个盒子”。我们知道超类使用散列引用，超类的层次结构并没有使用为一个派生类选取的 4 个实例变量名称。这是因为 RaceHorse 类是一个“友元类”（在 C++ 或者 Java 术语中），它可以直接访问实例变量。如果 Horse 或者 Animal 类的维护者改变了变量的表示形式或者名字，便会产生冲突，这类情况难以被发现，直到在某个重要的日子，你需要向投资人展示代码的时候才发生。当散列引用变为数组引用时，事情也可能会变得更加有趣。

其中一个消除这种依赖关系的方法是通过组合而不是继承的方式来创建一个派生类。在下面的示例中，我们需要生成一个 Horse 对象（一个 RaceHorse 类的实例变量）并且把剩余的数据放入单独的实例变量中。你还需要通过委托的方式将从 Horse 继承的方法传递到子类 RaceHorse 中。然而，即使 Perl 可以支持这些必要的操作，这种实现方式也是缓慢而笨拙的。关于此主题本书就讨论到这里。

接下来，我们提供了一些访问器方法。

```
package RaceHorse;
use parent qw(Horse);
## extend parent constructor:
sub named {
    my $self = shift->SUPER::named(@_);
    $self->{$_} = 0 for qw(wins places shows losses);
    $self;
}
sub won { shift->{wins}++; }
sub placed { shift->{places}++; }
sub showed { shift->{shows}++; }
sub lost { shift->{losses}++; }
sub standings {
    my $self = shift;
    join ', ', map "$self->{$_} $_[", qw(wins places shows losses);
}
```

在 scripts/racehorse.pl 中，我们编写了一个程序来尝试构造新的 RaceHorse 类的对象。

```
use RaceHorse;
my $racer = RaceHorse->named('Billy Boy');
# record the outcomes: 3 wins, 1 show, 1 loss
$racer->won;
$racer->won;
$racer->won;
$racer->showed;
$racer->lost;
print $racer->name, ' has standings of: ', $racer->standings, ".\n";
```

上面的程序会输出：

```
Billy Boy has standings of: 3 wins, 0 places, 1 shows, 1 losses.
[Billy Boy has died.]
[Billy Boy has gone off to the glue factory.]
```

请注意，我们依然保存了 Animal 类与 Horse 类的析构函数。超类并没有意识到我们在散列中添加了 4 个额外的元素，所以它们仍按照原来的方式工作。

18.6 使用类变量

如果我们想要列出迄今为止我们所构造的所有动物将会如何？所有动物在程序的命名空间中都会一直存在，但是一旦它们被 named 构造函数交还就不再存在了。它们并不是真正地消失，而是我们已经找不到了。

我们可以在散列中记录每一个创建的动物对象并且在散列中遍历。散列的键是字符串化形式的 animal 的引用^{注4}，而值可以是真实的引用，可以允许我们访问它的名字或者类。

例如，我们通过记录每个动物的创建来扩展 named 方法：

注 4：或者任何其他便于呈现和互不相同的字符串。

```

## in Animal
my %REGISTRY;
sub named {
    my $class = shift;
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    bless $self, $class;
    $REGISTRY{$self} = $self; # also returns $self
}

```

大写的%REGISTRY 名称就是为了提醒这个变量比大多数变量拥有更广阔的作用域。在这里，它是一个包含多个实例信息的元变量。我们仍然可以使用词法变量，但这次它在文件作用域内生效^{注5}。

当使用\$self 作为键时，Perl 解释器就将它字符串化，这就意味着它将变为对象的唯一字符串。

我们还需要添加一个新方法：

```

sub registered {
    return map { 'a '.ref($_)." named ".$_->name } values %REGISTRY;
}

```

现在我们可以看到我们所构造的所有动物：

```

my @cows = map Cow->named($_), qw(Bessie Gwen);
my @horses = map Horse->named($_), ('Trigger', 'Mr. Ed');
my @racehorses = RaceHorse->named('Billy Boy');
print "We've seen:\n", map(" $_[\n", Animal->registered);
print "End of program.\n";

```

这个程序输出：

```

We've seen:
a RaceHorse named Billy Boy
a Horse named Mr. Ed
a Horse named Trigger
a Cow named Gwen
a Cow named Bessie
End of program.
[Billy Boy has died.]
[Billy Boy has gone off to the glue factory.]
[Bessie has died.]
[Gwen has died.]
[Trigger has died.]
[Trigger has gone off to the glue factory.]
[Mr. Ed has died.]
[Mr. Ed has gone off to the glue factory.]

```

注意，所有保存各种动物的变量都将在一个合适的时间消失，因为所有保存动物的变量在最后一步都会被销毁。这并不复杂，因为我们创建了额外需要销毁的引用。下一节将讨论这个主题。

注 5：文件的作用域与 Perl 不使用其他技巧获得私有类变量密切相关。

18.7 削弱参数

%REGISTRY 变量也保存一个指向每个动物的引用，即使我们处理所包含的变量，比如让它们超出作用域：

```
{  
    my @cows = map Cow->named($_), qw(Bessie Gwen);  
    my @horses = map Horse->named($_), ('Trigger', 'Mr. Ed');  
    my @racehorses = RaceHorse->named('Billy Boy');  
}  
print "We've seen:\n", map(" $_[ ]\n", Animal->registered);  
print "End of program.\n";
```

即使没有代码来保存这些表示动物的变量，这些动物变量也不会被销毁。首先想到的是，可以通过修改析构函数来修复这个问题：

```
## in Animal  
sub DESTROY {  
    my $self = shift;  
    print "[", $self->name, " has died.]\n";  
    delete $REGISTRY{$self};  
}  
## this code is bad (see text)
```

但是这仍然会有相同的输出。为什么？因为 Perl 解释器直到对象的最后一个引用消失时才调用它的析构函数。同时，最后一个引用直到析构函数被调用才会被销毁^{注6}。

在 Perl 5.8 和以后的版本中（我们也没必要去使用更早的版本），一个解决方案是使用弱引用。弱引用并不像引用一样计数。对于计数问题，最好通过一个示例来解释。

弱引用的机制是内置于 Perl 5.8 版本的核心中。因此，我们需要一个外部接口来“削弱”程序，该程序可以由 Scalar::Util 模块导入^{注7}：

```
## in Animal  
use Scalar::Util qw/weaken/; # in 5.8 and later  
  
sub named {  
    ref(my $class = shift) and croak 'class only';  
    my $name = shift;  
    my $self = { Name => $name, Color => $class->default_color };  
    bless $self, $class;  
    $REGISTRY{$self} = $self;  
    weaken($REGISTRY{$self});  
    $self;  
}
```

当 Perl 解释器对某对象的活跃引用进行计数时^{注8}，它不会对通过 weaken 函数转化为的

注 6：我们生成一个指向鸡和鸡蛋的引用，但是这将引入另外一个从 Animal 类派生出的类。

注 7：在 Perl v5.6 中，可以通过 WeakRef 这个 CPAN 模块来模拟相同的函数。

注 8：在 Programming Perl 和 Perl 自身文档中所定义的“某物”，可以是引用所指向的任何东西，例如一个对象。如果我们是特别古板的人，我们可以将它称为引用物（referent）。

弱引用进行计数。如果所有普通引用都消失了，Perl 解释器会把这个对象删除并且将所有弱引用变为 `undef`。

现在，我们将可以得到期望的行为：

```
my @horses = map Horse->named($_), ('Trigger', 'Mr. Ed');
print "alive before block:\n", map(" $_\n", Animal->registered);
{
    my @cows = map Cow->named($_), qw(Bessie Gwen);
    my @racehorses = RaceHorse->named('Billy Boy');
    print "alive inside block:\n", map(" $_\n", Animal->registered);
}
print "alive after block:\n", map(" $_\n", Animal->registered);
print "End of program.\n";
```

这个程序输出：

```
alive before block:
a Horse named Trigger
a Horse named Mr. Ed
alive inside block:
a RaceHorse named Billy Boy
a Cow named Gwen
a Horse named Trigger
a Horse named Mr. Ed
a Cow named Bessie
[Billy Boy has died.]
[Billy Boy has gone off to the glue factory.]
[Gwen has died.]
[Bessie has died.]
alive after block:
a Horse named Trigger
a Horse named Mr. Ed
End of program.
[Mr. Ed has died.]
[Mr. Ed has gone off to the glue factory.]
[Trigger has died.]
[Trigger has gone off to the glue factory.]
```

注意，赛马与牛的对象在语句块的末尾消失，但是普通马的对象在程序的最后消失。

成功！

弱引用也可以解决一些内存泄漏问题。比如，假设一个动物进程想要记录它的父进程或者子进程。当每个子进程可能想要为每个父进程保留引用时，父进程想要保留所有子进程的引用。

我们可以削弱父子进程这两个链接中的任意一个（甚至两个全部削弱）。如果我们削弱与子进程有关的链接，当子进程的所有其他引用消失时，Perl 解释器就会销毁该子进程，这时与父进程有关的链接就变成 `undef`（或者可以设置一个析构函数完全地移除它）。然而，只要父进程还包含有子进程，父进程就不会消失。类似地，如果与父进程有关的链接被削弱了，当父进程的引用不再由其他数据结构所引用时，它就会变成是 `undef`。这其实非常灵活。



注意

当使用弱引用的时候，一定要确保你没有解引用一个已经变为 `undef` 的弱引用。

如果没有弱引用这种关系，一旦创建了任何父子进程，父、子进程都会保存在内存当中，直到最终的全局销毁阶段，无论其他保存父进程或子进程的结构是否销毁。

要非常注意：使用弱引用的时候一定要小心，不要在一个循环引用的问题中使用它。如果要销毁由弱引用保存的数据，你可能会遇到一些非常令人困惑的编程问题要解决和调试。

18.8 练习

可以在附录中的“第 18 章答案”部分找到这些练习的答案。

1. [45 分钟] 修改 `RaceHorse` 类，当创建马时，从永久存储（例如 DBM、`Storable`、`JSON` 等）中获得排名，当销毁马时更新排名。

例如，运行这个程序 4 次：

```
my $runner = RaceHorse->named('Billy Boy');
$runner->won;
print $runner->name, ' has standings ', $runner->standings, ".\n";
```

这段程序应该显示 4 次额外的获胜。同时，确保 `RaceHorse` 类继承了 `Horse` 类的全部属性。

Moose 简介

Moose 模块是 Perl 语言一个相对比较新的对象系统，可以通过 CPAN 获取^{注1}。因为 Moose 模块已经在社区中非常流行，我们认为它应该在本书中用单独的一章介绍。我们认为所有人依旧有必要学习基本的 Perl 知识，但是当我们在现实世界中编程时，大家都建议我们使用 Moose 模块。

Moose 模块通过简化我们应当做但通常可能会忽略的部分，使 Perl 的面向对象编程不再那么乏味。通过它的对象元编程协议，它允许强大的编码方式，但本章没有涉及这部分内容。

本章将回顾一下在之前章节中创建的类，并用 Moose 模块的基本特性重建它们。我们仅介绍一些基本的内容；Moose 模块值得用一本完整的书籍来介绍。

19.1 用 Moose 模块创建之前的 Animal 模块

首先，我们将在 Horse.pm 中创建一个拥有名字和颜色的 Horse 类：

```
package Horse;
use Moose;

has 'name' => ( is => 'rw' );
has 'color' => ( is => 'rw' );

no Moose;

__PACKAGE__->meta->make_immutable;

1;
```

首先介绍 Moose 模块中定义的 has，它将后面的内容设置为一个特性的名称和该特性的一个或者多个属性。这里，我们声明这两个特性是“read/write”，或者 rw。Moose

注 1：Moose 有自己的网站：<http://moose.perl.org/>。

模块将自动建立设置 (set) 和取出 (fetch) 这些属性值的访问器，因此我们不必自己去做。

当定义类之后，使用 no Moose 语句来清理 Moose 模块导入的所有子例程。也可以使用 namespace::autoclean 模块，该模块不仅对于 Moose 模块可行，而且对于其他模块可行，非 Moose 模块的写法可能如下所示：

```
package Horse;
use Moose;
use namespace::autoclean;

has 'name' => ( is => 'rw' );
has 'color' => ( is => 'rw' );

__PACKAGE__->meta->make_immutable;

1;
```

最后，调用 meta 对象，然后告诉它使类不可改变，这就意味着我们不必再计划修改它。这将使程序变得更快一点，因为 Moose 模块的底层将不必不断检测 meta 对象系统的变更情况。

现在，如下所示，可以在程序中使用 Horse 类：

```
use Horse;
use v5.10;

my $talking = Horse->new( name => "Mr. Ed" );
$talking->color( "grey" ); # sets the color

say $talking->name, ' is colored ', $talking->color;
```

请注意，我们并没有定义任何 new、color 或者 name 的方法：Moose 模块自动完成这些工作。

正如之前的章节所介绍的，这些方法并不是仅仅针对 Horse 类，因此可以在 Animal 类中也定义它们。我们能够用更简单的方式做这些事情。在 Animal.pm 中，我们使用与之前在 Horse.pm 中一致的写法，尽管我们只修改了包名称：

```
package Animal;
use Moose;
use namespace::autoclean;

has 'name' => ( is => 'rw' );
has 'color' => ( is => 'rw' );

__PACKAGE__->meta->make_immutable;

1;
```

注意



我们不必在 Animal 模块中定义我们自己的构造函数，因为 Moose 模块已经自动定义了。如果需要，也可以自己定义。关于我们没有介绍的 Moose 特性，请参见 Moose::Manual 模块的文档。

要在 Horse 模块中继承 Animal 模块，如下所示，使用 Moose 模块的 extends 函数定义继承关系，而不是 parent：

```
package Horse;
use Moose;
use namespace::autoclean;

extends 'Animal';

__PACKAGE__->meta->make_immutable;

1;
```

通过以上代码，Horse 类就成为一个换了包名的 Animal 类。我们需要添加一个 sound 方法，使我们的马儿可以叫“neigh”。如下所示，可以通过添加一个有默认值的特性来实现。

```
package Horse;
use Moose;
use namespace::autoclean;

extends 'Animal';

has 'sound' => ( is => 'ro', default => 'neigh' );

__PACKAGE__->meta->make_immutable;

1;
```

要使用 sound 方法，需要调用 Animal 模块中的 speak 方法。如下所示，可以不使用 Moose 模块的语法糖，而创建我们自己的方法。我们也添加一个默认的 sound 方法并且附加上如果运行失败的报错消息：

```
package Animal;
use Moose;
use namespace::autoclean;

has 'name' => ( is => 'rw' );
has 'color' => ( is => 'rw' );
has 'sound' => ( is => 'ro', default => sub {
    confess shift, " needs to define sound!"
} );

sub speak {
    my $self = shift;
    print $self->name, " goes ", $self->sound, "\n";
}

__PACKAGE__->meta->make_immutable;

1;
```

如果子类没有定义一个 sound 方法，就使用 confess 方法，这是 Moose 模块免费提供的。如下所示，程序现在如下所示：

```
use Horse;  
  
my $talking = Horse->new( name => "Mr. Ed" );  
$talking->speak;
```

19.1.1 使用“角色”替换“继承”

到目前为止，我们将之前的模块全部转换成了使用 Moose 模块的相同结构。其实，Moose 模块中还有很多特性可以用。不必继承自 Animal 模块，可以使用角色（role）。



注意

角色就像一个混入特性。那些方法并不是被继承过来的，它们只是添加到类中，并不会影响继承关系树。

此时使用 Moose::Role 模块而不是 Moose 模块。当我们这样做时，我们不需要调用默认的 speak 子例程，因为我们可以使用 role 特性声明所有使用这个 role 的类，这些类都需要定义自己的 sound 子例程：

```
package Animal;  
use Moose::Role;  
  
requires qw( sound );  
  
has 'name'  => ( is => 'rw' );  
has 'color' => ( is => 'rw' );  
  
sub speak {  
    my $self = shift;  
    print $self->name, " goes ", $self->sound, "\n";  
}  
  
1;
```

如下所示，要在类中包含一个角色（role），可以使用 with 来代替 extends：

```
package Horse;  
use Moose;  
  
with 'Animal';  
  
sub sound { 'neigh' }  
  
1;
```

如果我们没有定义 sound 子例程，Moose::Role 模块将提供一个长的回溯标记我们的错误。

19.1.2 默认值

Moose 模块支持默认值的概念。如下所示，我们添加一种默认的颜色，并且也设置一个类应有的责任：

```
package Animal;  
use Moose::Role;
```

```

use namespace::autoclean;

requires qw( sound default_color );

has 'name' => ( is => 'rw' );
has 'color' => (
    is => 'rw',
    default => sub { shift->default_color }
);

sub speak {
    my $self = shift;
    print $self->name, " goes ", $self->sound, "\n";
}

1;

```

初始化实例时，如果没有提供颜色，类的默认颜色就应当发挥作用，所以应当保证这个具体类能够提供默认的颜色。如下所示，我们派生的其他动物类现在看起来简单多了。

在 lib/Cow.pm 里面：

```

package Cow;
use Moose;
use namespace::autoclean;

with 'Animal';
sub default_color { 'spotted' }
sub sound { 'moooooo' }

__PACKAGE__->meta->make_immutable;

1;

```

在 lib/Horse.pm 里面：

```

package Horse;
use Moose;
use namespace::autoclean;

with 'Animal';
sub default_color { 'brown' }
sub sound { 'neigh' }

__PACKAGE__->meta->make_immutable;

1;

```

在 lib/Sheep.pm 里面：

```

package Sheep;
use Moose;
use namespace::autoclean;

with 'Animal';
sub default_color { 'black' }
sub sound { 'baaaah' }

__PACKAGE__->meta->make_immutable;

```

```
1;
```

现在我们可以将 sheep 模块作为一个实现的类。

```
use Sheep;
my $baab = Sheep->new( color => 'white', name => 'Baab' );
$baab->speak; # prints "Baab goes baaaah"
```

在上面的示例中，如果我们没有提供一种颜色，就要使用一个子例程引用来设置颜色。如下所示，我们同样可以直接使用一个字符串（或其他值）设置一个值的方式实现：

```
has 'color' => (
    is => 'rw',
    default => 'white',
);
```

19.1.3 约束值

如果我们设置一个动物的颜色，就可能需要先检查一下这个值是不是一个颜色。下面的一些调用将行不通：

```
$sheep->color( 'black' );
$sheep->color( 'Dolly' );      # oops, that's a name!
$sheep->color( 'Porkchop' );   # that's a name, too!
$sheep->color( 1/137 );        # what color is that?
$sheep->color( '#FFCCAA' );    # web safe sheep?
```

在之前的示例中，我们没有做任何操作来限定方法可以接受的值。Moose 模块为我们提供了一种方式来过滤那些无奇不有的非法参数。如下所示，has 可以接受一个 isa 参数：

```
has 'color' => ( is => 'rw', isa => 'Str' );
```

Str 类型在此处并不是特别有用，因为 Perl 解释器仍然将数字转换成字符串。列在 Moose::Util::TypeConstraints 模块中的其他默认类型都不能为我们工作。

我们可以通过定义一个子类型来生成我们自己的类型约束。在以下示例中，我们声明 ColorStr 作为 Str 的子类型，但我们也能够用一个验证子例程作为 where 语句的一部分。我们检查这个值，以\$_ 传入，检查是否存在于%colors 中。如果失败，我们从 message 语句中得到报错的消息。

```
{
  use Moose::Util::TypeConstraints;
  use namespace::autoclean;

  {
    my %colors = map { $_, 1 } qw( white brown black );
    subtype 'ColorStr'
      => as 'Str'
      => where { exists $colors{$_} }
      => message {
          "I don't think [$_] is a real color"
        };
  }
}
```

这样似乎做了太多事情，而只是在一个集合中验证存在性。其实，如下所示，可以用 enum 语句来声明一个子类型作为替代：

```
{  
    use Moose::Util::TypeConstraints;  
    use namespace::autoclean;  
    enum 'ColorStr' => [qw( white brown black )];  
}
```

我们仅仅只需要定义这些约束一次，然后它们将在程序中所有位置都可用。我们已经将这些约束添加到 Moose 系统中，而不仅仅是定义它们的类或文件之中。

现在，尽管我们已经称这些为“类型约束”，而且 Moose 中也称它们为“类型约束”，但它们确实不是我们在其他语言中见到的相同东西。Perl 并不会提前告知我们有一个类型违背。直到我们尝试调用这个方法之前，我们都不会发现。也就是说，Moose 模块并没有改变 Perl 语言的动态本质。只是把这些当成参数校验。

19.1.4 封装方法

Mouse 类有点特别，因为它用另一行语句的输出扩展了 speak 方法。当我们用基于 SUPER:: 的方法调用父类的行为时，因为角色没有使用继承，所以我们不能使用角色机制来完成。然而，Moose 让我们可以封装已有的方法。使用 after 语句，我们可以用一个方法来替换 speak 方法，该方法先调用初始的 speak 方法，然后做额外的工作（合理地保留主调上下文）^{#2}：

```
package Mouse;  
use Moose;  
use namespace::autoclean;  
  
with 'Animal';  
  
sub default_color { 'white' }  
sub sound { 'squeak' }  
after 'speak' => sub {  
    print "[but you can barely hear it!]\n";  
};  
  
__PACKAGE__->meta->make_immutable;  
1;
```

现在我们有了一个带有 speak 方法的 Mouse 类：

```
use Mouse;  
my $mickey = Mouse->new( name => 'Mickey' );  
$mickey->speak;
```

输出包括额外的内容行：

```
Mickey goes squeak  
[but you can barely hear it!]
```

如果我们想在调用原始方法前有一些额外的行为，我们可以使用 before 语句而不是 after

^{#2} 2：我们可以使用 Hook::LexWrap 模块封装任意子例程，该模块可以在 CPAN 上获得。

语句。或者，我们可以使用 around 语句在原始方法调用前后做一些填充的工作。如下所示，与之前所做的类似，我们可以使用如下代码同时处理类和实例方法。

```
package Animal;
...
has 'name' => (is => 'rw');
around 'name' => sub {
    my $next = shift;
    my $self = shift;
    blessed $self ? $self->$next(@_) : "an unnamed $self";
};
};
```

我们使用 has 为‘name’创建基本的行为，但随后用 around 包围它。新子例程的第一个参数是指向原始方法的引用，然后是方法的一些普通参数。我们通常要检查\$self，判断它是否是一个被 bless 操作过的引用。如果它是，就调用这个方法，如果\$self 没有被 bless 操作过，它就一定是一个类名，然后我们使用 an unnamed \$self。

around 语句假设在生成对象时，就会给动物起一个名字。我们不必指定一个名字。如果我们有给 new 提供一个名字或者后续提供，就可以使用基于类名称的默认名字。

```
has 'name' => (
    is => 'rw',
    default => sub { 'an unnamed ' . ref shift },
);
};
```

强制用户定义一个名字，这样我们就不用设置一个默认值，可以通过设置 required 特性实现：

```
has 'name' => (
    is => 'rw',
    required => 1,
);
};
```

如下所示，当我们想生成一个未命名的动物时，就会得到一个错误：

```
Attribute (name) is required at constructor Sheep::new
```

19.1.5 只读属性

如下所示，如果我们不想让人们改变动物的颜色，就可以使用 ro 将属性设置为只读而不是 rw。

```
has 'color' => (
    is => 'ro',
    default => sub { shift->default_color }
);
};
```

如下所示，如果我们尝试使用一个参数调用 color 方法，就会得到一个错误：

```
Cannot assign a value to a read-only accessor
```

这并不意味着我们永远不能改变颜色。如下所示，我们可以重新指定另外一个方法来做这件事。

```
has 'color' => (
    is => 'ro',
    writer => '_private_set_color',
    default => sub { shift->default_color },
);
```

如下所示，尽管如此我们不能直接改变老鼠的颜色：

```
my $m = Mouse->new;
my $color = $m->color;
$m->color('green'); # DIES
```

我们仍然可以使用私有名称替代：

```
$m->_private_set_color('green');
```

根据惯例，一条前导下划线将表示这个方法是私有的，我们不能在类的外部使用。然而，这并不意味着它就是一个真正的私有方法。

19.2 改进的赛马类

我们通过把赛手（racer）的特性添加到 Horse 类中，创建一个赛马类。

我们将与比赛相关的部分定义为一个角色（role）。在前面的示例中，我们必须继承自 Horse 类和 RaceHorse 类。如下所示，现在我们可以摆脱多重继承，因为我们使用角色替换其中一个父类：

```
package Racer;
use Moose::Role;
use namespace::autoclean;

has $_ => (is => 'rw', default => 0)
foreach qw(wins places shows losses);

1;
```

has 实际上就是一个子例程，因此我们可以像其他子例程一样调用它。我们使用一个带后缀的 foreach 语句使一些方法变为都是可读/写的，并且有个默认初始值 0。

我们为每一项统计信息添加了一些增长数值的方法，然后使用 standings 方法生成总结。

```
package Racer;
...
sub won    { my $self = shift; $self->wins($self->wins + 1) }
sub placed { my $self = shift; $self->places($self->places + 1) }
sub showed { my $self = shift; $self->shows($self->shows + 1) }
sub lost   { my $self = shift; $self->losses($self->losses + 1) }

sub standings {
    my $self = shift;
    join ", ", map { $self->$_ . " $" } qw(wins places shows losses);
}
```

有些人可能认为直接访问散列将会更容易一些，因为这样看起来也更简洁。

```
sub won { shift->{wins}++; }
```

然而，我们不知道 wins 方法实际上做了什么。我们展示了 before、after 和 around 语句。当打破封装时，我们就破坏了它们。我们应当总是信任接口。

要创建一个 Race Horse 类，用比赛者（racer）这个角色扩展 Horse 类：

```
package RaceHorse;
use Moose;
use namespace::autoclean;

extends 'Horse';
with 'Racer';

__PACKAGE__->meta->make_immutable;
1;
```

现在我们可以骑小马了：

```
use RaceHorse;
my $s = RaceHorse->new( name => 'Seattle Slew' );
$s->won;
$s->won;
$s->won;
$s->placed;
$s->lost;
print $s->standings, "\n"; # 3 wins, 1 places, 0 shows, 1 losses
```

19.3 进一步学习

我们只是粗略的介绍了 Moose 模块所提供的一些皮毛知识。我们可以定义强大而灵活的定制构造函数（注意，在本章还没有一个构造函数），把参数值强制转化为正确的类型，处理多个角色并且当角色冲突时适当地调度，你可以在 Moose 模块的文档中看到更多的内容。

19.4 练习

可以在附录中的“第 19 章答案”部分找到这部分练习的答案。

1. [45 分钟] 使用 Moose 模块重新实现 Animal 类，创建一个扩展 Animal 类的子类。调整文档并测试这些变化。确保根据新的依赖关系更新了相应的构建文件。
2. [10 分钟] 修改练习 1 中所创建的发行版，使用 Animal 作为角色，而不是用父类。

高级测试

在本章中，我们尝试了一些更受欢迎的测试模块，以及 Test::More 模块的一些高级功能。除非特别说明，这些模块都不是 Perl 标准发行版的一部分（与 Test::More 不同），并且我们需要自己安装它们。你可能会觉得被本章欺骗，因为我们会常常说“参见模块文档”，但是我们是在缓慢的推动你进入 Perl 领域。对于更多的细节，你也可以查看 *Perl Testing, A Developer's Notebook*，该书涵盖了更进一步的主题。

20.1 跳过测试

在某些情况下，我们想跳过测试。例如，我们的一些特性可能只适合某个特定的 Perl 解释器版本、一个特定的操作系统，或者仅仅当可选的模块可用时才能够工作。为了跳过测试，我们会做和 TODO 测试同样多的工作，但是 Test::More 会做一些相对来说更加不同的工作。

我们再次使用一个裸块来创建一段需要跳过的代码，并且将其标注为 SKIP。在测试过程中，与无论如何都会运行的 TODO 块不同，Test::More 将不执行这部分测试。在块的开始部分，我们调用 skip 函数来说明为什么我们想要跳过这些测试，以及一共有多少测试我们想要忽略。

在以下示例中，我们要检查 Mac::Speech 模块是否在尝试测试 say_it_aloud 方法之前就已经安装。如果没有安装，eval 语句块返回 false，与此同时执行 skip 函数：

```
SKIP: {
    skip 'Mac::Speech is not available', 1
    unless eval { require Mac::Speech };

    ok( $tv_horse->say_it_aloud( 'I am Mr. Ed' ) );
}
```

当 Test::More 模块对跳过测试时，它将输出专门的 ok 消息，以此来保持测试编号正确并告诉测试工具发生了什么。之后，测试工具可以报告我们一共跳过多少个测试。

如果是因为程序工作的不正确，我们就不应该跳过这些测试。在这种情况下我们可以使用 TODO 语句块。当我们想要使测试在特定情况下可选时，就可以使用 SKIP 语句块。

20.2 测试面向对象特性

对于面向对象模块，当调用构造函数时，要确保返回一个对象。对于这方面而言，使用 Test::More 模块的 isa_ok 函数和 can_ok 函数都是好的接口测试方法：

```
use Test::More;

BEGIN{ use_ok( 'Horse' ); }
my $trigger = Horse->named('Trigger');
isa_ok($trigger, 'Horse');
isa_ok($trigger, 'Animal');
can_ok($trigger, '_') for qw(eat color);
done_testing();
```

这些测试用例都有默认的测试名称，而测试输出如下：

```
ok 1 - use Horse;
ok 2 - The object isa Horse
ok 3 - The object isa Animal
ok 4 - Horse->can('eat')
ok 5 - Horse->can('color')
1..5
```

如下所示，这里会测试它是马，同时它也是动物，并且它既具有 eat 方法，又可以返回一个 color。我们可以进行进一步的测试，以确保每一匹马都有一个唯一的名字：

```
use Test::More;

BEGIN{ use_ok( 'Horse' ); }

my $trigger = Horse->named('Trigger');
isa_ok($trigger, 'Horse');

my $tv_horse = Horse->named('Mr. Ed');
isa_ok($tv_horse, 'Horse');

# Did making a second horse affect the name of the first horse?
is($trigger->name, 'Trigger', 'Trigger\'s name is correct');
is($tv_horse->name, 'Mr. Ed', 'Mr. Ed\'s name is correct');
is(Horse->name, 'a generic Horse');

done_testing();
```

下面的输出告诉我们，未命名的马并不完全是我们所认为的结果：

```
ok 1 - use Horse;
ok 2 - The object isa Horse
ok 3 - The object isa Horse
ok 4 - Trigger's name is correct
ok 5 - Mr. Ed's name is correct
not ok 6
#   Failed test at t/horse.t line 14.
#       got: 'an unnamed Horse'
#       expected: 'a generic Horse'
1..6
# Looks like you failed 1 test of 6.
```

观察测试结果。我们写了一个通用的 Horse 类，但是这个字符串实际上是一个未命名的 Horse。该错误在测试中而不是在模块中，所以我们要改正测试错误并重试。除非模块的规范确实需要：“一个通用的 Horse 类”^{注1}。我们不应该害怕只是编写测试和测试模块。如果我们发现其中任何一个有错，其他的通常也会捕捉到该错误。

20.3 分组测试

不是每一个测试都能够表现出代码的逻辑测试。我们实际上可能想测试几件事情作为测试单个功能的准备。通过使用 Test::More 的 subtest 特性，我们可以把这些测试组成一个单元来表示一个测试。我们给它一个标签，在此处作为第一个参数和一个对测试分组的代码引用：

```
use Test::More;

subtest 'Major feature works' => sub {
    ok( defined &some_subroutine, 'Target sub is defined' );
    ok( -e $file, 'The necessary file is there' );
    is( some_subroutine(), $expected, 'Does the right thing' );
};

done_testing();
```

在顶层，代码引用中的三个测试作为单个测试。TAP 利用嵌套测试来实现它。该子测试的输出是缩进的，并有自己的计划：

```
ok 1 - Target sub is defined
ok 2 - The necessary file is there
ok 3 - Does the right thing
1..3
  ok 1 - Major feature works
  1..1
```

如果所有子测试都通过，则全部测试通过。可以修改 t/Animal.t 测试来对 speak 和 sound 子例程的测试分组，并且调整计划只对顶层测试计数：

注 1：之后我们会发现这些测试不仅仅会检查代码，而且会以代码的形式创建规范。

```

use strict;
use warnings;

use Test::More tests => 3;

BEGIN {
    use_ok( 'Animal' ) || print "Bail out!\n";
}

subtest 'sound() works' => sub {
    ok( defined &Animal::sound, 'Animal::sound is defined' );
    eval { Animal->sound() } or my $at = $@;
    like( $at, qr/You must/, 'sound() dies with a message' );
};

subtest 'speak() works' => sub {
    ok( defined &Animal::speak, 'Animal::speak is defined' );
    eval { Animal->speak() } or my $at = $@;
    like( $at, qr/You must/, 'speak() dies with a message' );
};

```

20.4 测试大型字符串

第 14 章曾经展示，当一个测试失败时，Test::More 模块可以展示我们所期望的和我们实际上得到的：

```

use Test::More;
is( "Hello Perl", "Hello perl" );
done_testing();

```

当运行这个程序时，Test::More 模块会展示究竟出了什么问题：

```

% perl test.pl
not ok 1
#     Failed test (test.pl at line 5)
#         got: 'Hello Perl'
#         expected: 'Hello perl'
1..1
# Looks like you failed 1 test of 1.

```

如果字符串非常长会怎样？如下所示，我们并不希望看到整个字符串，它可能有数百或数千个字符那么长。我们只想看到它们从什么地方开始不同：

```

use Test::More;
use Test::LongString;

is_string(
    "The quick brown fox jumped over the lazy dog\n" x 10,
    "The quick brown fox jumped over the lazy dog\n" x 9 .
    "The quick brown fox jumped over the lazy camel",
);

done_testing();

```

错误的输出不必展示整个字符串来告诉我们究竟哪里出错。它只需要展示相关的部分以及字符串的长度。尽管该示例有点做作，还是想象对一个 web 页面、配置文件或者我们不希望弄乱测试输出的其他一些大块数据执行这个操作：

```
not ok 1
#   Failed test in long_string.pl at line 6.
#       got: ..." the lazy dog\x{0a}"...
#       length: 450
#   expected: ..." the lazy camel"...
#       length: 451
#   strings begin to differ at char 447
1..1
# Looks like you failed 1 test of 1.
```

20.5 测试文件

用来测试诸如文件是否存在和文件大小这类事情的代码是十分简单的，但是，如果我们编写的代码越多，每一条代码语句就会有更多的部分，这也就更可能会使我们不仅陷入困境，而且也会给维护代码的程序员传递错误的信息。

我们可以很容易地测试文件是否存在。我们在来自 Test::More 的 ok 函数中使用-e 文件测试运算符，这就可以很好地完成工作：

```
use Test::More;
ok( -e 'minnow.db' );

done_testing();
```

如果以上代码是我们打算测试的内容，它就将运行良好，但是在该代码中没有任何部分告诉别人我们打算做什么。如果我们是想要确保在开始测试之前该文件并不存在，该如何实现呢？如下所示，实现该功能的代码仅需要改动一个字符：

```
use Test::More;
ok( ! -e 'minnow.db' );

done_testing();
```

我们可以添加一行代码注释，但是大多数代码注释似乎总假设我们已经知道了应该会发生什么。这行注释可以让我们知道这两种情况中的哪种是我们想要的吗？如果文件存在，我们是否要通过测试呢？

```
use Test::More;
# test if the file is there
ok( ! -e 'minnow.db' );

done_testing();
```

由 brian 所编写的 Test::File 模块，在函数名称中封装了函数的意图。当文件存在时，如果我们想要通过测试，就使用 file_exists_ok 方法：

```
use Test::More;
use Test::File;
```

```
file_exists_ok( 'minnow.db' );
done_testing();
```

当文件不存在时，如果我们想要通过测试，则使用 file_not_exists_ok 方法：

```
use Test::More;
use Test::File;

file_not_exists_ok( 'minnow.db' );

done_testing();
```

这是一个简单的示例，但是如果模块有许多其他函数也遵循同样的命名规范：名字的第一部分告诉我们函数检查什么，file_exists，而最后一部分则告诉我们如果文件存在会发生什么，_ok。当我们不得不把这些敲出来时，错误传达意图这种事情就会很难发生：

```
use Test::More;
use Test::File;

my $file = 'minnow.db';

file_exists_ok( $file );
file_not_empty_ok( $file );
file_readable_ok( $file );
file_min_size_ok( $file, 500 );
file_mode_is( $file, 0775 );

done_testing();
```

所以，显式的函数名称不仅可以用来传递意图，而且有助于代码中的并行结构。

20.6 测试 STDOUT 和 STDERR

使用 ok 函数（和友元）的一个优点是它们不需要直接写入标准输出 STDOUT，而是在测试脚本开始运行时，写入一个从 STDOUT 偷偷地复制过来的文件句柄。如果不改变程序中的 STDOUT，这将成为一个有争议的问题。但是如果我们想要测试一个在 STDOUT 中写入了一些东西的例程，比如确保一个 Horse 类适当地调用 eat 方法，我们则要小心：

```
use Test::More;
use_ok 'Horse';
isa_ok(my $trigger = Horse->named('Trigger'), 'Horse');

open STDOUT, ">test.out" or die "Could not redirect STDOUT! $!";
$trigger->eat("hay");
close STDOUT;

open T, "test.out" or die "Could not read from test.out! $!";
my @contents = <T>;
close T;
is(join("", @contents), "Trigger eats hay.\n", "Trigger ate properly");
done_testing();

END { unlink "test.out" } # clean up after the horses
```

在开始测试这个 eat 方法之前，要对临时输出文件打开或重新打开标准输出 STDOUT，该方法中的输出以 test.out 文件结束。我们将该文件的内容取出来，并将它给予 is 函数。即使我们不关闭 STDOUT，is 函数依然可以访问原始的 STDOUT，因此测试套件将看到合适的 ok 或 not ok 消息。

如果我们创建一个类似的临时文件，需要记得当前的目录与测试脚本相同（即使我们从父目录运行 make test）。此外，如果想让人们能够以可移植的方式使用和测试我们的模块，我们需要选择一个比较安全的跨平台名称。

不过对于这点有一个更好的方法，Test::Output 模块可以自动处理这个问题。该模块可以为我们提供几个能够自动考虑到所有细节的函数。



注意

老版本的 Test::Output 模块在读取一些输出特例时会有一些问题。具体参见 <http://www.dagolden.com/wp-content/uploads/2009/04/how-not-to-capture-output-in-perl.pdf>。

```
use Test::More;
use Test::Output;

sub print_hello { print STDOUT "Welcome Aboard!\n" }
sub print_error { print STDERR "There's a hole in the ship!\n" }

stdout_is( \&print_hello, "Welcome Aboard\n" );
stderr_like( \&print_error, qr/ship/ );

done_testing();
```

以上所有函数都以一个代码引用作为它们的第一个参数，但是这并不成问题，因为第 7 章已经阐述了所有相关的知识。如果我们没有一个子例程要测试，我们就把我们想要测试的代码封装在一个子例程里，并使用它：

```
sub test_this {
    print_error();
    print STDERR "Some other output";
    ...
}

stdout_is( \&test_this, ... );
```

如果代码足够短，我们可能会想要跳过定义一个命名的子例程的步骤，并使用一个匿名子例程：

```
stdout_is( sub { print "Welcome Aboard" }, "Welcome Aboard" );
```

我们甚至可以使用一个内联代码块，比如我们之前对于 grep 函数和 map 函数使用的。与这两个列表操作符一样，注意，在内联代码块之后并没有一个逗号：

```
stdout_is { print "Welcome Aboard" } "Welcome Aboard";
```

除了 Test::Output 模块以外，我们可以使用 Test::Warn 模块执行一些相似的操作，尤其

是在测试警告输出方面。它的接口专门使用内联代码块形式：

```
use Test::More;
use Test::Warn;

sub add_letters { "Skipper" + "Gilligan" }

warning_like { add_letters() }, qr/nonnumeric/;

done_testing();
```

我们总是致力于使代码警告随时发出，并且我们也可以为此测试。Perl 解释器的警告在不同的版本之间变化，在新的警告弹出时我们就能够知道，或者如果 Perl 解释器将在其中一个客户的计算机上发出警告。Test::NoWarnings 模块与我们之前已经展示的模块有点不同。它仅仅通过加载模块来自动添加一个测试，而我们只要确保添加了我们用于 Test::More 模块计数的隐藏测试：

```
use Test::More tests => 1;
use Test::NoWarnings;

my( $n, $m );
# use an uninitialized value
my $sum = $n + $m;
```

当我们试图计算总和时，我们使用了两个还未赋予值的变量。这会触发令人烦恼的警告“use of uninitialized value”（确保警告已经打开！）。毫无疑问，现在我们并不想让这类事件填满日志文件，对吧？当此类事件发生时，Test::NoWarnings 模块会告诉我们如何修复：

```
1..1
not ok 1 - no warnings
#   Failed test 'no warnings'
#   in /usr/local/lib/perl5/5.8.7/Test/NoWarnings.pm at line 45.
# There were 2 warning(s)
#       Previous test 0 ''
#       Use of uninitialized value in addition (+) at nowarnings.pl line 6.
#
# -----
#       Previous test 0 ''
#       Use of uninitialized value in addition (+) at nowarnings.pl line 6.
#
# Looks like you failed 1 test of 1 run.
```

20.7 使用模拟对象

有时我们并不想增强整个系统而只是仅仅测试它其中的一部分。我们可以相当确定，或者至少假设，系统的其他部分能正常工作。我们不需要打开昂贵的数据库连接或者实例化具有大内存的对象来测试代码的每一部分。

Test::MockObject 模块创建了假装的（pretend）对象。我们向它提供我们想要实用的对象接口部分的信息，并且它会假装成为该接口的一部分。基本上，当调用这些“假装

的”方法时，它必须返回正确的结果，而且不需要做任何处理。

与创建一个能够在 boat 上打开各种各样东西的真正的 Minnow 对象不同，我们可以创建一个模拟的对象作为代替。一旦创建了模拟对象并将其储存在\$Minnow 中，我们会告诉它如何对于我们需要调用的方法做出响应。在这种情况下，我们让模拟对象对 engines_on 返回 true，对 moored_to_dock 返回 false。我们并不是真正测试 ship 的对象，我们仅仅是想测试 quartermaster 对象，它把一个 ship 作为一个参数。我们使用模拟对象，而不是测试具有一个真正的 ship 的 quartermaster 对象：

```
use Test::More;
use Test::MockObject;

# my $Minnow = Real::Object::Class->new( ... );
my $Minnow = Test::MockObject->new();

$Minnow->set_true( 'engines_on' );
$Minnow->set_true( 'has_maps' );
$Minnow->set_false( 'moored_to_dock' );

ok( $Minnow->engines_on, "Engines are on" );
ok( ! $Minnow->moored_to_dock, "Not moored to the dock" );

my $Quartermaster = Island::Plotting->new(
    ship => $Minnow,
    # ...
);

ok( $Quartermaster->has_maps, "We can find the maps" );

done_testing();
```

我们可以创建更复杂的方法来做我们想做的任何事情。假设与返回 true 或 false 的方法不同，我们需要一个返回一个列表的方法。或许我们需要假装连接一个数据库并检索一些记录。在我们开发的过程中，我们可能会多次尝试该操作，但在每次我们尝试追踪一个 bug 时，我们不希望与真正的数据库连接和断开。

在这个示例中，我们模拟数据库方法 list_names，该方法将返回给我们三个名字。既然我们已经知道了这些，而且我们实际上正在测试其他内容（在这个虚构的示例中不展现），所以可以创建代替真正的数据库的模拟方法：

```
use Test::More;
use Test::MockObject;
my $db = Test::MockObject->new();

# $db = DBI->connect( ... );
$db->mock(
    list_names => sub { qw( Gilligan Skipper Professor ) }
);

my @names = $db->list_names;
```

```

is( scalar @names, 3, 'Got the right number of results' );
is( $names[0], 'Gilligan', 'The first result is Gilligan' );

print "The names are @names\n";

done_testing();

```

20.8 编写我们自己的 Test::* 测试模块

我们不需要等待别人来写出色的测试模块。如果有一个特定的测试情景，并且想把其封装进一个测试函数，我们可以使用 Test::Builder 模块来编写我们自己的 Test::* 模块，该模块使用 Test::Harness 模块和 Test::More 模块，可以处理所有棘手的集成问题。如果我们回顾许多 Test::* 模块的应用场景，我们很可能会发现使用 Test::Builder 模块的痕迹。

再一次，测试函数的优点是它们可以把可重用的代码封装在一个描述预期行为的函数名中。为了测试一些东西，我们使用函数名而不是输入一串单独的语句。简单的函数名称更易于让人理解测试的意图，但是，如果做同样的事情，重复编写一些语句就显得更难一些。在第 4 章中，我们编写了一些代码来检查所有幸存者是否拥有全部所需的物品（items）。现在，我们把这些操作转变成一个 Test::* 模块。这就是第 4 章介绍的 check_required_items 子例程：

```

sub check_required_items {
    my $who = shift;
    my %whos_items = map { $_, 1 } @_ # the rest are the person's items

    my @required = qw(preserver sunscreen water_bottle jacket);

    for my $item (@required) {
        unless ( $whos_items{$item} ) { # not found in list?
            print "$who is missing $item.\n";
        }
    }
}

```

我们需要将这个子例程转换成一个 Test::* 模块，用于简单地检查物品（所以它并没有添加缺失的物品），然后输出正确的内容。对于任何新的测试模块基础都是相同的。我们调用新模块 Test::Minnow::RequiredItems，并且从此存根（stub）开始：

```

package Test::Minnow::RequiredItems;
use strict;
use warnings;

use Exporter qw(import);
use vars qw(@EXPORT $VERSION);

use Test::Builder;

my $Test = Test::Builder->new();

```

```

$VERSION = '0.10';
@EXPORT  = qw(check_required_items_ok);

sub check_required_items_ok {
    # ...
}

1;

```

我们首先声明包名称，然后打开约束和警告，因为我们想要成为优秀的程序员（即使这 3 小时的过程注定是失败的，这也不会是来源于一个软件错误）。正如第 17 章所示，因为我们想要在主调命名空间中显示函数，所以我们导入 Exporter 模块并把 required_items_ok 添加到@EXPORT 数组中。我们按照第 12 章介绍的方法设置\$VERSION。之前没有介绍过的唯一知识就是 Test::Builder 模块。在测试模块的开始部分，创建一个新的 Test::Builder 对象，我们将该对象赋值给词法变量\$Test，它的作用域是整个文件^{#2}。

\$Test 对象将会自动处理所有的测试细节。我们删除了 check_required_items 函数的所有输出部分，然后去掉了修改输入列表的部分。一旦我们仔细检查了其他逻辑部分，最后要做的唯一事情就是告诉测试套件，测试是 ok 或者是 not_ok：

```

sub check_required_items {
    my $who   = shift;
    my $items = shift;

    my @required = qw(preserver sunscreen water_bottle jacket);
    my @missing = ( );

    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # not found in list?
            push @missing, $item;
        }
    }

    if (@missing) {
        ...
    }
    else {
        ...
    }
}

```

现在我们必须添加一些使函数变成一个测试用例的语句。在\$Test 上调用方法来告诉测试套件所发生的事情。在每种情况下，最后求值的表达式应该是一个对\$Test->ok 的调用，以便成为整个函数的返回值。



注意

因为大多数人都在一个空的上下文环境中调用大部分测试函数，所以我们经常不使用返回值，但是我们可以返回一些有意义的东西。

^{注 2：} 它几乎像一个全局变量，除了它不在包中并且在文件外面不可见之外。

如果我们发现丢失的物品，我们会想要测试失败，这样我们就可以把一个 `false` 值赋予 `$Test->ok`，但是在此之前，我们可以使用 `$Test->diag` 和一条消息来告诉我们究竟什么出错了：

```
sub check_required_items_ok {
    my $who    = shift;
    my $items  = shift;

    my @required = qw(preserver sunscreen water_bottle jacket);
    my @missing = ( );

    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # not found in list?
            push @missing, $item;
        }
    }

    if (@missing) {
        $Test->diag( "$who needs @missing.\n" );
        $Test->ok(0);
    }
    else {
        $Test->ok(1);
    }
}
```

尽管我们还可以做很多事情，此处却没有我们必须做的更多事情。一旦我们保存了 `Test::Minnow::RequiredItems` 模块，我们就可以立刻在一个测试脚本中使用它。我们仍然使用 `Test::More` 模块来设置计划：

```
use Test::More;
use Test::Minnow::RequiredItems;

my @gilligan = (
    Gilligan => [ qw(red_shirt hat lucky_socks water_bottle) ]
);

check_required_items_ok( @gilligan );
done_testing();
```



注意

我们可以为模块设置一个测试计划，但最有可能的情况是测试脚本也会使用其他模块。而只有它们中的一个可以设置测试计划，所以我们让 `Test::More` 模块来处理这类问题。

因为 `Gilligan` 没有他所需的所有物品，所以测试失败。它输出 `not_ok` 和诊断消息：

```
not ok 1
1..1
# Gilligan needs preserver sunscreen jacket.
#     Failed test (/Users/Ginger/Desktop/package_test.pl at line 49)
# Looks like you failed 1 test of 1.
```

而且，既然我们已经创建了 `Test::Minnow::RequiredItems` 模块，那么我们如何测试这个测试模块？此时可以使用 `Test::Builder::Tester` 模块。不过您必须自行研究该模块的细节。

20.9 练习

可以在附录中的“第 20 章答案”部分找到这些练习的答案。

1. [30 分钟] 使用 Test::File 模块来检查在 UNIX 系统中/etc/hosts 文件和在 windows 系统中 C:\windows\system32\drivers\etc\hosts 文件的存在性和可读性（或者，如果你并没有这些文件，就选择你有的一个文件）。通过检查\$^O (大写字母 O) 变量的值，跳过你并不在的平台上的测试。可以添加这个测试文件到你自己的 My::List::Util 发行版中或者作为一个独立的程序使用它。
2. [30 分钟] 编写你自己的测试模块 (Test::My::List::Util)，该模块只有一个测试函数 (sum_ok)，它有两个参数：实际总数和预期总数。如果这两个参数不匹配，输出一条诊断消息：

```
my $sum = sum( 2, 2 );
sum_ok( $sum, 4, 'The sums match' );
```

除了在本章中的示例，你也可以查询 Test::File 模块上的资源（或者大量其他 Test::* 模块），为你的测试模块获取更好的思路。

第 21 章

贡献到 CPAN

除了允许我们社区的其他人获取我们所创建的那些奇妙的模块和各种各样的发行版所带来的好处之外，我们能为整个 Perl 社区做出贡献。分享你们工作的机制就是 Perl 综合典藏网的目的，当我们编写本书时，它已经存在了差不多 20 多年，而且迄今已经拥有超过 10 万个不同的模块。

21.1 Perl 综合典藏网

第 2 章介绍了 CPAN 一些基本的使用机制，但那是从模块用户的角度。现在我们想要向 CPAN 贡献代码，那么我们从模块作者的角度看看吧。

毋庸置疑，CPAN 是非常有用的。这个项目的宗旨就是让任何人都能够贡献代码，而且能够让任何人都能够很容易向大家分享他们的代码。非官方 CPAN 的座右铭是：“尽早上传，尽快更新”。我们没有必要非要等到代码完成后才开始分享给大家。

请记住，CPAN 是一个巨大的存储装置。这就是它的魔力所在。几乎拥有任何问题的解决方案，例如 MetaCPAN (<https://www.metacpan.org/>)、CPAN.pm 和 CPANPLUS，只需拿来用就好，不用再去重新创建。

21.2 准备阶段

既然 CPAN 是一个巨大的文件存储站点，那么我们需要向它上传我们的发行版。要贡献代码到 CPAN 上，我们需要以下两个东西：

- 需要贡献的内容，理想情况下就是一个成型的模块；
- 一个 Perl 作者上传服务器（PAUSE）账户。

PAUSE 账户是我们向 CPAN 贡献代码的通行证。我们只需要简单申请就可以得到一个 PAUSE 账户^{注1}。我们要填写一个只有几项基本内容的网页表单(这里有链接)，例如我们的名字、电子邮件地址和我们想要使用的 PAUSE 账户名称。请注意，PAUSE 账户名称必须在 4~9 个字符之间(一些遗留的 PAUSE 账户名称可能只有三个字符^{注2})。所有账户都是由专人审批的，这样可以有效避免机器人注册或多重账户的情况，因此审批可能大概需要一天左右的时间，直到申请被审批。不过很少会不批准。如果你跳过了本书开始的前言，或者还没有设置一个 PAUSE 账户，请马上就注册一个吧(或者下次你可以访问网络时)。否则在本章后续的部分，当你需要时，就不得不等待审批了。

一旦我们拥有 PAUSE 账户，我们就需要全面仔细地考虑我们所提交的模块。因为我们的模块可能和其他作者的模块在程序中使用，所以我们需要确保模块的包名称不会与现有的模块冲突，也不能让查询 CPAN 的人感到困惑。幸运的是，有一个松散的志愿者团队维护着 Perl 模块列表(`module@perl.org`)，他们已经长时间使用 CPAN 和诸多模块，而且能够帮你解决很多问题。

21.3 PAUSE 的工作方式

通过 PAUSE 可以把模块和程序提交到 CPAN 上。每个账户都有自己的目录。Randal 的目录是 `authors/id/M/ME/MERLYN`。在你浏览 CPAN 时会看到这些目录。当 Randal 上传了一个发行版的安装包，就将它放到自己的目录下。他也可以上传自己喜欢的任何东西，因为这完全是他自己的目录。

然而，Randal 不会独自工作。尽管他可能上传代码，但他不必是编写那些代码的人。同样，他也能自己编写代码，然后让其他人替他上传。一些发行版有多个维护人员，一些项目会特别指定一个发布经理(release manager)。PAUSE 总是将发行版安装包放在上传者的目录中。一些发行版是大家轮流履行发布经理的职责，因此可能会每一个新的发行版出现在不同作者的目录中。就其本身而言，对于本章剩余的部分，我们将介绍上传者，当我们提到“作者”时，就意味着“上传者”。

PAUSE 还做一些索引每个发行版的工作，然后生成一个从命名空间到发行版的映射。在 CPAN 上，有一个 `modules/02packages.details.gz` 文件，该文件包含一个命名空间、版本号和发行版所在的路径：

注 1：如果你已经完成了第 1 章的练习，就应该有一个 PAUSE 账户，但是如果没有，请点击：<http://www.cpan.org/modules/04pause.html>。

注 2：最初，PAUSE 名称必须是五个字符或者更少，直到 Randal 想要 MERLYN 这个名字，而且也是因为他做了适当的调整。

```
File::Finder      0.53 M/ME/MERLYN/File-Finder-0.53.tar.gz
File::Finder::Steps 0.53 M/ME/MERLYN/File-Finder-0.53.tar.gz
File::Findgrep    0.02 S/SB/SBURKE/File-Findgrep-0.02.tar.gz
File::FindLib      0.001001 T/TY/TYEMO/File-FindLib-0.001001.tar.gz
File::Flock        2008.01 M/MU/MUIR/modules/File-Flock-2008.01.tar.gz
```

CPAN 客户端使用这些数据去查找他们想要的发行版，然后获取并且安装模块。如下所示，当我们像这样运行客户端时：

```
% cpan File::Finder
```

程序将在 modules/02packages.details.txt.gz 中查找 File::Finder。它将获取最新的版本号，然后与自身系统中当前所安装的模块版本号相比较。如果所获取索引中的版本号更大，客户端就知道需要获取/M/ME/MERLYN/File-Finder-0.53.tar.gz。这把该地址追加到 CPAN 镜像地址后面以获取完整的 URL 路径，例如 <http://www.cpan.org/authors/id/M/ME/MERLYN/File-Finder-0.53.tar.gz>。

02packages.details.txt.gz 文件仅仅列出每个命名空间一次，并且仅列出索引的最新版本。

21.3.1 索引器

一切从我们上传一个发行版安装包到 PAUSE 开始。我们所上传的任何东西都会进入上传者目录。然而，技巧在于如何上传人们能够使用 CPAN 客户端安装的发行版。



注意

进入 PAUSE 然后在 https://pause.perl.org/pause/authenquery?ACTION=add_uri 上传发行版安装包。

当 PAUSE 发现有一个新的发行版时，它就试图通过解压缩然后查看发行版所包含的命名空间来索引该发行版。我们将发行版中所找到的命名空间与它之前索引的命名空间列表进行比较。一次成功的索引添加命名空间和版本号到 02packages.details.txt.gz 内。

如果 PAUSE 遇到之前从未见过的命名空间，上传者将会获得该名称的“先入手”特权，然后 PAUSE 将会索引命名空间和模块版本号（查询\$VERSION 变量）。上传者仅仅获得在该确切的命名空间上的权限，而不是该命名空间下的所有命名空间的权限。例如，当 Randal 上传了 File::Finder 模块，他将获得了该命名空间的“先入手”特权，但是没有获得 File::Finder::Enhanced 模块的命名空间权限，在 File::Finder 模块层次结构之下的命名空间并不属于该模块。其他任何人都能够上传一个命名空间在 File::Finder 之下的模块，即使所上传的模块与 Randal 的 File::Finder 模块没有什么关系。

如果 PAUSE 先查看了模块的命名空间，而且上传者有维护权限，PAUSE 比较当前上传模块的版本号和它之前索引的版本号。如果新上传的发行版版本号更大，它就将索引命名空间并且更新版本号。如果版本号更低，它就不会索引命名空间，而是发送一个

错误的报告给上传者。发行版仍然能够放入上传者的目录中。

如果 PAUSE 找到了模块的命名空间，但是上传者没有维护权限，PAUSE 就不会索引命名空间。发行版仍然能够放入上传者的目录之中，但是命名空间将不会在元数据文件中展示。其他人仍然能够从 CPAN 上下载该模块，但是他们不得不通过这个路径获取发行版的文件。

因为 PAUSE 将会为它在发行版中所找到的每个命名空间都执行这个遍历流程，所以它可能索引一部分并且忽略一部分。当我们遇到这个问题时，PAUSE 就会给我们发送邮件。这些完全或者部分没有被索引的发行版仍然会出现在搜索结果里面，但是可能被打上“未被认证”的标记。



注意

PAUSE 使用我们自己账户中设定的邮件地址，因此，如果我们尝试使用一个非法的邮件地址绕过，例如 `gilligan@no.spam`，我们就永远看不到任何 PAUSE 的帮助消息。

21.3.2 模块维护人员

我们在之前的章节中提到“第一个”维护者。此外，还有两类维护方式。

- “第一个”维护者就是第一位将模块上传到命名空间的人。一个模块列表维护者拥有完整地添加可选注册命名空间的额外步骤（https://pause.perl.org/pause/authenquery?ACTION=apply_mod）。
- 主要维护者就是被配置为负责命名空间权限的人。默认情况下，这就是“第一个”维护者，但是主要的维护权限也可以传递给其他某人。主要维护者可以给其他人分配共同维护者权限，但是，顾名思义，只能有唯一一个主要维护者。
- 共同维护者拥有上传和索引命名空间的权限，但是不能把共同维护者权限授予其他人。

一个主要维护者能够将这个角色传递给其他人，但是他们也能够放弃这个角色而没有指派其他主要维护者，留下一个没有任何人能够创建新的共同维护者的命名空间。

有些时候，某些模块所有的维护者都消失了。这是因为一些人对于这些已经不能用于他们工作的模块不再感兴趣，其他一些人开始使用其他的模块，还有一些人实在是太忙了。当没有人想要交接一个模块的维护权限时，PAUSE 的管理员就可以提供帮助。在初始作者消失的情况下，PAUSE 的管理员拥有一整套流程用于转移命名空间的控制权到一个新的开发者。



注意

获取命名空间的流程在 <http://pause.perl.org> 的 CPAN FAQ 上以及 <http://www.cpan.org/misc/cpan-faq.html> 上有概述。

很容易查询谁拥有一个命名空间的权限。PAUSE 网站允许搜索模块或作者，然后显示命名空间、作者和权限。

21.4 在我们开始工作之前

在我们开始编写我们自己的模块之前，我们应该做一些研究，这可能最终会节省很多时间。

是否已经有一个模块能够满足我们的需要？我们是在重新发明一个已经存在模块的子集，或者我们可以将我们的工作成果作为另一个模块的补丁？我们可能找到一些非常相似但是需要一些额外的帮助才能实现的模块。与其创建另一个几乎已经完成的模块，还不如使用已有的模块。如果模块最初的作者消失，PAUSE 的管理员会转移模块的维护权限。

我们需要为模块想出一个好名称，而且这个名称应当对于其他人而言更易于理解，而不仅仅只是适合我们使用名称。我们选取的名称如何适用于 CPAN 上的其他名称呢？选取什么样的名称才能帮助人们更容易发现我们的模块？一旦我们选取一个名称，然后公之于众，我们事实上就已经必须接受这个名称了，因为人们已经在他们的代码中使用它，并且不愿意去修改它。大家可以通过阅读 module-authors@perl.org 和 modules@perl.org 这两个邮件列表，来帮助选取一个好的模块名称，而且 PAUSE 管理员也有一个命名指南。



注意

对于 PAUSE 的命名指南，请参阅 https://pause.perl.org/pause/authenquery?ACTION=pause_namingmodules。

如果我们告诉全世界我们正在做的事情，Perl 开发者社区可能知道一个模块已经做了我们想做的。对于在 CPAN 上的 100 000 多个的模块，那个选择还不算得上太糟糕。我们不需要知道 CPAN 上已有的所有模块。我们必须知道大多数人对此的共同理解是什么。

21.5 准备发行版

一旦我们确定了模块名称，并且我们已使用这个新名称（根据需要）测试模块，我

们就应确保它准备开始发布了。如果我们使用了其中一个模块创建工具，我们就应该已经拥有这些文件。如果我们手动生成了模块，我们就应该确保我们已经包含了这些文件。



注意

由 Sam Tregar 编写的 *Writing Perl Modules for CPAN* (Apress 出版社) 是一本关于介绍这部分内容的图书，而我们试图在这一章中总结这本书的全部内容。这本书有点老，但是里面介绍的内容与实际变化不大。

21.5.1 创建或更新 README

创建（或更新）一个 README 文件。这个文件在 CPAN 存档中自动提取到一个单独的文件中，允许大家在获取或者解压缩你的发行版之前查看或下载相关的关键内容。

21.5.2 检查构建文件

生成并且测试你的 Makefile.PL 或 Build.PL 文件。模块没有能正常工作的构建文件，或者没有任何构建文件，仍然能够进入 CPAN，但通常遭到下载用户生气的抱怨。一个常见的抱怨是构建文件中缺少先决条件（见第 12 章）。

21.5.3 更新清单

我们需要确保 MANIFEST 是最新的。如果我们已经添加属于发行版的文件，这些文件也需要在 MANIFEST 之中。发行版的归档仅仅包括我们列在该 MANIFEST 里面的文件。

一个快速的诀窍是保留你想要留在发行版中的文件，然后调用./Build manifest（或者 make manifest），它更新 MANIFEST 文件，以确定在发行版目录中的确切内容。



注意

这些并不是我们想要放入.gitignore 文件或在命令行上使用的 glob 模式。

如果./Build manifest 添加太多的文件，我们可以创建一个 MANIFEST.SKIP 文件，该文件包含有一组 Perl 正则表达式，这些表达式告诉./Build manifest 需要忽略哪些文件。如下是一个由 module-starter 创建的 MANIFEST.SKIP 文件示例：

```
# Avoid configuration metadata file
^MYMETA\.

# Avoid Module::Build generated and utility files.
\bBuild\$
\bBuild.bat$
```

```
\b_build
\bBuild.COM$
\bBUILD.COM$
\bbuild.com$
^MANIFEST.SKIP

# Avoid archives of this distribution
\bAnimal-[d\.\_\_]+
```

这是一个文本文件。可以向其中添加任何喜欢的内容。如果添加了 lib/Kangaroo.pm，./Build manifest 自动添加它，因为它是发行版中的新文件，它不会被 MANIFEST.SKIP 中的任何模式排除：

```
% ./Build manifest
File 'MANIFEST.SKIP' does not exist: Creating a temporary 'MANIFEST.SKIP'
Added to MANIFEST: lib/Kangaroo.pm
```

如果添加了一些文件，比如使发行版成为一个 Git 仓库，./Build manifest 将按照如下方式添加它们：

```
% ./Build manifest
File 'MANIFEST.SKIP' does not exist: Creating a temporary 'MANIFEST.SKIP'
Added to MANIFEST: .git/config
Added to MANIFEST: .git/description
Added to MANIFEST: .git/HEAD
Added to MANIFEST: .git/hooks/applypatch-msg.sample
Added to MANIFEST: .git/hooks/commit-msg.sample
Added to MANIFEST: .git/hooks/post-commit.sample
...

```

我们可以通过添加.git.*到 MANIFEST.SKIP 中排除这些文件。下次运行./Build manifest，这些条目将消失（但文件本身不会改变）：

```
% ./Build manifest
Removed from MANIFEST: .git/config
Removed from MANIFEST: .git/description
Removed from MANIFEST: .git/HEAD
Removed from MANIFEST: .git/hooks/applypatch-msg.sample
Removed from MANIFEST: .git/hooks/commit-msg.sample
Removed from MANIFEST: .git/hooks/post-commit.sample
Removed from MANIFEST: .git/hooks/post-receive.sample
Removed from MANIFEST: .git/hooks/post-update.sample
Removed from MANIFEST: .git/hooks/pre-applypatch.sample
Removed from MANIFEST: .git/hooks/pre-commit.sample
Removed from MANIFEST: .git/hooks/pre-rebase.sample
Removed from MANIFEST: .git/hooks/prepare-commit-msg.sample
Removed from MANIFEST: .git/hooks/update.sample
Removed from MANIFEST: .git/info/exclude
Removed from MANIFEST: .gitignore
```

如果我们每次做了什么都要给 MANIFEST.SKIP 添加条目，我们会发疯的。幸运的是，有一个默认的模式列表，我们可以在文件中加上如下这一行语句：

```
#!include_default
```

这条语句将从 ExtUtils/MANIFEST.SKIP 添加模式到现有的 MANIFEST.SKIP 文件之中（在操作之后，这个文件会改变）。可以使用 perldoc 查看默认列表，尽管这看起来像是

意外的工作：

```
% perldoc -m ExtUtils::MANIFEST.SKIP
```

21.5.4 添加版本字符串

我们需要使用一个便于理解的发行版的版本字符串，该字符串需要比之前 PAUSE 索引的版本字符串更大一些（数值上，而不是字符串上），我们不能太过于苛求：版本 1.9 大于 1.10 版本！实际情况比这个更复杂，但是因为我们还没有结束版本的介绍，所以我们会在此展开详细介绍。



注意

“遗憾的是，在 Perl 中版本号并不乏味而且十分容易”，David Golden 写道，详见 <http://www.dagolden.com/index.php/369/version-numbers-should-be-boring/>。

Build.PL 文件将要么指定一个 VERSION 值要么指定一个 VERSION_FROM 值，如果我们在发行版中有单个模块（例如一个.pm 文件），通常最好从 dist_version_from（或者在 Makefile.PL 中的 VERSION_FROM）中获取版本号。如果有多个.pm 文件，就可以指定其中一个作为定义版本号的源文件：

```
my $builder = Module::Build->new(
    module_name      => 'Animal',
    dist_version_from => 'lib/Animal.pm',
    ...
);
```

我们可以把版本号直接放入 Build.PL 里面：

```
my $builder = Module::Build->new(
    module_name      => 'Animal',
    dist_version     => '1.023',
    ...
);
```

21.5.5 测试发行版

到目前为止，我们已经按照如下方式测试我们的工作：

```
% ./Build test
```

我们在当前的工作目录执行上述语句，其中包括所有已经添加到目录中的文件。当我们归档发行版时，我们只包括 MANIFEST 中的文件。如果我们忘记更新 MANIFEST 文件不会使它或者在 MANIFEST.SKIP 文件中使用一个过于互斥的模式，测试需要的一些文件可能不会使它进入发行版中：

```
% ./Build disttest
```

以上操作将 MANIFEST 中的所有文件构建为一个发行版的归档文件，解压缩该归档文件到另一个单独的目录中，然后在发行版上运行测试。如果这样并不能正常工作，我

们不指望该发行版能够工作在其他任何人从 CPAN 上下载我们发行版的环境下。

21.6 上传发行版

一旦发行版准备共享（甚至可以在此之前），我们就可以通过 PAUSE 页面 https://pause.perl.org/pause/authenquery?ACTION=add_uri 上传发行版。使用我们的 PAUSE 账户和密码登录，然后选择一个上传选项。我们可以上传一个文件，指定获取发行版的 URL 地址，或者声明我们已经通过一个匿名 FTP 上传的一个文件。



注意

一些在线资源的控制服务将归档一个提供一个适当 URL 的目录。

无论我们以何种方式上传文件，它都将会出现在页面底部的已上传文件列表中。我们可能需要等一会，以便让 PAUSE 获取这些远程文件，但是它们通常在一个小时以内出现。如果我们没有看到 PAUSE 名称在发行版名称旁边，我们就不能声明它。



注意

因为我们上传到一个匿名但不公开的 FTP 站点，所以其他 Perl 程序员可以直接从 incoming 目录下载，这样他们可以在 PAUSE 查看这些代码之前使用它。

一旦 PAUSE 获取到文件，并且知道该文件属于谁，它就会索引该文件。你将得到一个来自 PAUSE 索引器的电子邮件，告诉你发生了什么事情。此后，你的发行版就开始向各个镜像站点传输。请记住，N 在 CPAN 中是网络的缩写，所以你的发行版可能需要花几小时或几天才能传输到所有的 CPAN 镜像站点。通常不需要超过两三天时间，使用由 PAUSE 的工作人员开发的快速 rsync 脚本，通常几分钟就可以同步完毕。

如果你有问题，或者想到某些没有发生但是可能会发生的事情，你可以通过发送电子邮件到 modules@perl.org，来询问 PAUSE 管理员。

21.7 在多个平台上测试

CPAN 的 Tester (<http://cpantesters.org>) 自动测试几乎所有上传到 CPAN 的发行版。来自世界各地的志愿者们自动下载并且在他们的环境下测试每一个发行版，他们在每一平台、操作系统和 Perl 版本上测试我们的模块（可能其中的很多环境我们并不在意）。他们发邮件给模块作者告诉他们发生的事情，然后会自动更新 Testers 数据库。我们可

以通过 Tester 网站 (<http://www.cpantesters.org>) 或在 MetaCPAN 上 (<https://www.metacpan.org/>) 查询任何模块的测试结果。

通常情况下，tester 能够帮助我们找出我们不能访问的平台的问题。虽然失败的测试报告可能令人沮丧，我们应该记得要善待我们的测试人员。我们的发行版没有通过测试，(通常)并不是由于他们的过错。

21.8 发布模块

我们发布一个模块，一部分人会使用它，另一部分人会将会改善它。大家需要知道关于我们模块的所有这些信息。我们的模块将自动在多个网站引起注意，其中包括：

- CPAN Search 的 “Recent modules” 页面：<http://search.cpan.org/recent>
- MetaCPAN 的 “new modules” 部分：<https://www.metacpan.org/recent>
- “Perl news” 邮件列表的每日公告

在 Perl 会议上有很多关于模块作者介绍他们工作的简短演讲。毕竟，还有谁比他们更有资格来帮助别人使用我们的模块呢？越多的人对我们的模块感兴趣，我们就能够得到更好的 bug 报告、功能请求和补丁。



注意

如果你想成为一个演讲者，“Nobocty is a good speaker when they start” (<http://blog.yapcna.org/post/17253936133/nobody-is-a-good-speaker-when-they-start>) 中有一些励志文字。

如果在 Perl 会议上演讲的计划有点恐怖（这并不会吓住任何模块作者），或者你不想等那么久，就可以看一下你的本地 Perl 用户组。他们通常寻找演讲者（通常为接下来一两周的会议），这个用户组的大小通常小到可以随意设置安排。你通常可以通过查询 Perl Mongers 网站 (<http://www.pm.org/>) 在你附近找到一个 Perl 用户组。如果你找不到一个当地用户组，就自己建立一个！

21.9 练习

可以在附录的“第 21 章答案”部分找到这些练习的答案。

1. [5 分钟] 如果你还没有 PAUSE 账户，就注册一个。你不会立即得到，但你不需要它也可以开始工作。
2. [10 分钟] 确保你的 Animal 发行版可以通过其自身的 disttest 检查。先检查它是否能

通过自身的 test 检查！做任何需要的修改，直到通过 disttest 测试。

3. [45 分钟] 用你的 PAUSE 账户创建一个新的发行版，使用 Acme::*名称空间。例如，如果你的 PAUSE 账户名是“GILLIGAN”，就创建一个 Acme::GILLIGAN::Utils 模块。创建一个 sum 函数，添加数字，然后为这个函数创建一个测试。准备上传你的发行版，然后更新到 PAUSE 上。
4. [10 分钟] 添加一个 lib/tie/Cycle.pm 到练习 3 中创建的发行版中，或者可以从 Tie::Cycle 发行版中复制一个。准备把你的发行版上传到 PAUSE，确保 lib/Tie/Cycle.pm 出现在 MANIFEST 中。上传你的发行版，等待 PAUSE 索引它，这样你就可以看到关于未授权的命名空间的错误消息。
5. [20 分钟] 使用之前练习中创建的 Acme::*模块，修改 sum 函数，以便函数将数字相乘而不是相加。现在测试将会失败；就让测试失败，因此你可以看到当上传一个损坏的发行版时所发生的事情。
6. [5 分钟] 在 MetaCPAN 上查看您的发行版的页面 (<https://www.metacpan.org>)。该站点从 PAUSE 上更新非常快，因此你能够在一小时内看到你的模块。
7. [10 分钟] 使用你的某个 CPAN 客户端，在 CPAN 上安装你的 Acme::*发行版。你可能要等到你的发行版传输到你配置的 CPAN 镜像站点。

附录

练习答案

本附录包含该书中列出的所有练习的解答。本书部分练习有一些补充资源，读者可以到 <http://www.intermediateperl.com> 网站的 Download 部分进行下载。

第 1 章答案

我们给出了这一章练习的答案，但是在第一部分并没有太多需要做的事情。

练习 1

我们已经在第 21 章介绍过，要申请得到一个 PAUSE 账户，我们可以访问 <http://pause.perl.com>，打开“Request PAUSE account”链接。填写相关信息之后，应用程序会转向 PAUSE admins 以进行人工检查。我们不需要证明我们是否有足够的资格去获得这个账户；只要你是还没有账户的个人（而不是一个公司或角色），你都可以申请，你的申请应该会在一到两天之内得到批准，恰好可以在我们需要时及时使用。

一旦我们的账户被审核通过，我们就会同时拥有一个账户名称以及@cpa.n.org 为结尾的邮箱地址。下一步我们可能会通过该地址创建一个全球认证肖像 (<http://www.gravatar.com/>)。当我们想展现一个作者的头像时，CPAN 搜索界面会查找全球认证头像并附加到我们的 @cpa.n.org 地址上。例如：<https://metacpan.org/author/BDFOY>。

练习 2

在网站 <http://www.intermediateperl.com> 中包含本书的相关资料和羊驼的图片。

第 2 章答案

练习 1

本练习的技巧是让模块做所有困难的工作。很高兴我们已经向您展示了如何使用模块！Cwd 模块（ cwd 是 current working directory 的首字母缩略词）自动导入 getcwd 函数，我们不必担心它如何实现它的魔力，我们应该有信心，它能在绝大多数主要的平台上正确运行。

一旦我们获得了存储当前路径的标量 \$ cwd，我们就可以把它作为第一个参数传递给 File::Spec 模块中的 catfile 方法。catfile 方法的第二个参数来自于 foreach 的输入列表，并以 \$ _ 的形式显示：

```
use Cwd;
use File::Spec;

my $cwd = getcwd;

foreach my $file ( glob( ".* *" ) ) {
    print "    ", File::Spec->catfile( $cwd, $file ), "\n";
}
```

如果我们希望使用 File::Spec::Functions 模块，foreach 中的代码就可以更简短一些：

```
use Cwd;
use File::Spec::Functions;

my $cwd = getcwd;

foreach my $file ( glob( ".* *" ) ) {
    print "    ", catfile( $cwd, $file ), "\n";
}
```

练习 2

安装了 local::lib 模块之后，我们就可以通过使用该模块来安装其他模块了，这就产生了一个先有鸡还是先有蛋的经典问题，我们想要在我们的目录中安装它，但是我们还没有安装 local::lib 模块。幸运的是，在文档中有一个启动进程可以帮助我们。

首先，通过 <https://www.metacpan.org/module/local::lib> 下载 local::lib 模块，然后把它解压并运行 Makefile.PL --bootstrap：

```
% perl Makefile.PL --bootstrap
% make install
```

至此我们已经安装好 local::lib 模块，之后我们就可以使用它和其中的一个 CPAN 工具：

```
% cpan -I Module::CoreList
% cpanm Module::CoreList
```

现在，我们在程序中使用 local::lib，配置跟之前的程序一样：

```

use local::lib;

use Module::CoreList;

my @modules    = sort keys $Module::CoreList::version{5.014002};

my $max_length = 0;
foreach my $module ( @modules ) {
    $max_length = length $module if
        length $module > $max_length;
}

foreach my $module ( @modules ) {
    printf "%*s %s\n",
        - $max_length,
        $module,
        Module::CoreList->first_release( $module );
}

```

如下为得到的输出：

AnyDBM_File	5
App::Cpan	5.011003
App::Prove	5.010001
App::Prove::State	5.010001
App::Prove::State::Result	5.010001
App::Prove::State::Result::Test	5.010001

虽然这个练习的目的是学会使用 local::lib 模块，但是从答案中我们还是可以发现一些有趣的事情。由于第一列有不同的宽度但是左对齐，所以第二列的数字也右对齐。

一旦我们得到 v5.14.2 中的模块列表，我们就遍历所有模块的名称，从而找到最长的模块名称的长度。在 printf 函数中，使用 “%*S” 格式说明符，其中 “*” 告诉 printf 函数从下一个参数中获得域的长度，该长度就是 - \$max_length，其中 “-” 为左对齐的字符串。

我们还没介绍过 map (将在下一章中介绍)。如果我们可以使用 map，就没必要使用第一个 foreach 循环语句，如下所示，我们可以将所有模块名称的长度传递给 List::Util 模块的 max 函数：

```

use List::Util qw(max);

my $max_length = max map { length } @modules;

```

练习 3

首先我们使用一个 CPAN 客户端来安装 Business::ISBN 模块：

```
% cpan -I Business::ISBN
% cpann Business::ISBN
```

一旦我们安装 Business::ISBN 模块，我们就可以按照文档中的示例来实践。如下所示，首先在程序中我们从命令行读取 ISBN 的值，然后创建新的 ISBN 对象，该对象保存到变量\$isbn 中。当获得这个对象之后，我们继续查阅文档中的示例：

```
use Business::ISBN;

my $isbn = Business::ISBN->new( $ARGV[0] );

print "ISBN is " . $isbn->as_string . "\n";
print "Country code: " . $isbn->country_code . "\n";
print "Publisher code: " . $isbn->publisher_code . "\n";
```

第3章答案

练习1

如下为其中一种解决方法：先把命令行输入的参数储存到特殊数组@ARGV 中，然后利用 grep 函数筛选这个数组，所有小于 1000 字节的文件都将储存到数组 @smaller_than_1000 中。该数组也传递给 map 函数，map 函数将分别取出每一个元素，然后在元素的前面加上一些空格，后面加上一个换行符，然后返回：

```
my @smaller_than_1000 = grep { -s < 1000 } @ARGV;

print map { "    $_[0]\n" } @smaller_than_1000;
```

通常，我们也可以不使用中间的数组：

```
print map { "    $_[0]\n" } grep { -s < 1000 } @ARGV;
```

练习2

在这里，我们选择主目录作为硬编码目录。当调用 chdir 时，如果没有传递参数，它就将进入主目录（所以，这也是 Perl 语言中少数几个不使用\$_作为默认值的地方）。

此后，一个无限的 while 循环使代码持续运行，直到 Last 的条件不能满足时，跳出循环。仔细查看一下现有的条件：我们并不需要测试条件为真。如果我们想查找所有包含一个“0”的文件，程序会发生什么呢？我们查找有非零长度的定义值，所以 undef（输入结束）和空字符串（按下输入键）会终止 while 循环。

利用正则表达式，我们重做之前的练习。这次，我们使用 glob 的结果作为输入列表，在 grep 内部使用模式匹配。在模式匹配的两边加上一个 eval {} 语句以防止这个被编译（例如，它有一个不匹配的括号或方括号）：

```
chdir; # go to our home directory

while( 1 ) {
    print "Please enter a regular expression> ";
    chomp( my $regex = <STDIN> );
    last unless( defined $regex && length $regex );
    print
        map { "    $_[0]\n" }
        grep { eval{ $regex =~ /$_/ } }
        glob( "./*" );
}
```

第 4 章答案

练习 1

要求我们区分下列 4 个表达式：

```
$ginger->[2][1]      # 1  
${$ginger[2]}[1]      # 2  
$ginger->[2]->[1]  # 3  
${{$ginger->[2]}[1]} # 4
```

除了第二个表达式 \${\$ginger[2]}[1] 以外，它们所指的都是同一个东西。第二个表达式与 \$ginger[2][1] 是等价的，它基于 @ginger 数组，而不是 \$ginger 标量。

当我们画 PeGS 的时候可以很容易看出这些差别。图 A-1 显示的是 \$ginger 之后带 -> 符号的表达式的图，这是一个引用（同时也是一个标量，因为所有引用都是标量）。图 A-2 显示的是 \${\$ginger[2]}[1] 表达式的图。

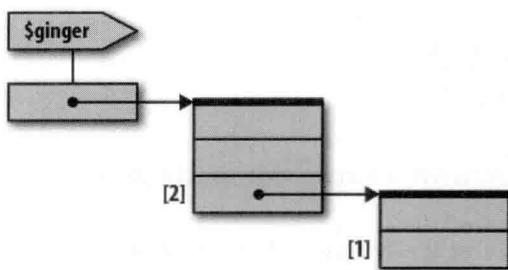


图 A-1 \${\$ginger->[2]}[1]、\$ginger->[2]->[1] 和 \${\$ginger->[2]}[1] 的 PeGS

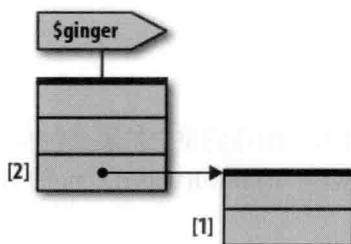


图 A-2 \${\$ginger[2]}[1] 的 PeGS，即带一个数组引用元素的命名数组

练习 2

首先，按照如下方式构建散列结构：

```
my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my %all = (
    "Gilligan" => \@gilligan,
    "Skipper" => \@skipper,
    "Professor" => \@professor,
);
```

然后，将这个散列传递给第一个子例程：

```
check_items_for_all(\%all);
```

在子例程中，第一个参数是一个散列引用，所以我们需要对它进行解引用，来获得键及其对应的值：

```
sub check_items_for_all {
    my $all = shift;
    for my $person (sort keys %$all) {
        check_required_items($person, $all->{$person});
    }
}
```

从那里开始，调用原始子例程：

```
sub check_required_items {
    my $who = shift;
    my $items = shift;
    my @required = qw(preserver sunscreen water_bottle jacket);
    my @missing = ();
    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # not found in list?
            print "$who is missing $item.\n";
            push @missing, $item;
        }
    }
    if (@missing) {
        print "Adding @missing to @$items for $who.\n";
        push @$items, @missing;
    }
}
```

练习 3

我们从船上人员花名册程序开始，但我们将它扩展到包含其余的遇难者。在此并不列出所有遇难者，但是在 <http://www.intermediateperl.com/> 的 Download 部分有一个更详细的文件：

```
my %gilligan_info = (
    name      => 'Gilligan',
    hat       => 'White',
    shirt     => 'Red',
    position  => 'First Mate',
```

```

);
my %skipper_info = (
    name      => 'Skipper',
    hat       => 'Black',
    shirt     => 'Blue',
    position  => 'Captain',
);
my %mr_howell = (
    name      => 'Mr. Howell',
    hat       => undef,
    shirt     => 'White',
    position  => 'Passenger',
);
my @castaways = (\%gilligan_info, \%skipper_info, \%mr_howell);

```

一旦拥有@castaways 数组，就可以从头到尾遍历每个元素，并对其添加一个散列键：

```

foreach my $person (@castaways) {
    $person->{location} = 'The Island';
}

```

在此之后，再次遍历@castaways 数组，跳过每一个不到达 Howell 的元素。对于最终得到的元素，可以改变它们的位置：

```

foreach my $person (@castaways) {
    next unless $person->{name} =~ /Howell/;
    $person->{location} = 'The Island Country Club';
}

```

最后，再一次遍历@castaways 并输出结果：

```

foreach my $person (@castaways) {
    print "$person->{name} at $person->{location}\n";
}

```

但是并不需要遍历@castaways 三次，我们可以一次性完成所有这些工作，改变位置然后立即输出结果：

```

foreach my $person (@castaways) {
    if( $person->{name} =~ /Howell/ ) {
        $person->{location} = 'The Island Country Club';
    }
    else {
        $person->{location} = 'The Island';
    }

    print "$person->{name} at $person->{location}\n";
}

```

使用条件运算符好像看起来更好一点：

```

foreach my $person (@castaways) {
    $person->{location} = ( $person->{name} =~ /Howell/ ) ?
        'The Island Country Club' : 'The Island';

    print "$person->{name} at $person->{location}\n";
}

```

第 5 章答案

练习 1

匿名散列构造器的花括号返回的是散列引用，该引用是一个标量（和所有的引用一样），所以它不适合单独用来作为散列所期望的键值对。也许这段代码的作者是为了将其赋值给标量变量（如 \$passenger_1 和 \$passenger_2），而不是给散列，但是我们通过将两对花括号变为括号来解决这个问题。

如果我们尝试运行这个程序，Perl 就会以警告的方式给我们提供一条有用的诊断消息。如果我们没有得到警告，那可能是因为我们没有打开警告，此时可以使用 -w 开关或者使用 warnings 编译指示符来打开警告。即使我们平常不使用 Perl 的 warnings，我们也应该在调试期间启用警告（没有 Perl 的警告的帮助，需要花费我们多长时间调试这个程序？打开 Perl 的警告需要多长时间？这就不用多说了吧）。

如果我们得到了警告消息，却不知道警告消息是什么意思，该怎么办呢？这就是 perldiag 的用途所在。警告消息文字一定要简洁明了，因为它们被编译成 Perl 解释器的二进制包（用于运行 Perl 代码的程序）。perldiag 列出所有我们可以从 Perl 获得的消息，以及每一条很长的解释信息，解释每一个的含义，为什么这是一个问题，以及如何解决。

如果想偷懒，我们可以在程序的开始处添加 “use diagnostics;”，这样任何错误消息都会在文档中自动查找其内容，然后显示完整的详细消息。然而，在最终的产品里我们将不会保留这些，除非我们喜欢在每一次程序开始时，不论是否发生错误，都占用大量 CPU 周期。

练习 2

我们想要统计有多少数据已经被发送到所有机器，所以在一开始，我们将变量 \$all 设置成一个能够代表所有机器的名称。这个名字应该是一个永远不会用于任何真实机器的名称。为了方便写程序，我们将其存储在一个变量中，稍后改变它也更容易：

```
my $all = "***all machines***;
```

输入循环几乎与本章给出的一样，不一样的地方就是它跳过注释行。同时，它统计有多少数据已经被发送到所有机器，并存在 \$all 中：

```
my %total_bytes;
while (<>) {
    next if /^#/;
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
    $total_bytes{$source}{$all} += $bytes;
}
```

接下来，我们生成一个排序列表。这个列表包含所有源主机的名称，并按总传输字节的降序进行排列。我们在 for 循环外部使用这个列表，而不是使用一个临时数组 @sources，我们本可以把 sort 操作直接放在 for 循环的圆括号内：

```

my @sources =
    sort { $total_bytes{$b}{$all} <=> $total_bytes{$a}{$all} }
    keys %total_bytes;

for my $source (@sources) {
    my @destinations =
        sort { $total_bytes{$source}{$b} <=> $total_bytes{$source}{$a} }
        keys %{ $total_bytes{$source} };
    print "$source: $total_bytes{$source}{$all} total bytes sent\n";
    for my $destination (@destinations) {
        next if $destination eq $all;
        print " $source => $destination:",
              " $total_bytes{$source}{$destination} bytes\n";
    }
    print "\n";
}

```

在循环内部，我们输出从源主机发出的所有数据的总字节数，然后生成目标文件的排序列表（与@sources 中的列表相似）。当遍历列表时，我们使用 next 跳过空的\$all 项。因为该项位于排序列表的首部，那么为什么我们不用 shift 来丢弃它呢？因为这样可以避免在内层循环内部重复检查它。

即使这个空项会出现在排序列表的首部，它也不必为列表的第一个元素。假如一个主机仅仅发送数据到另一个主机上，那么目标主机的接收数据总量会跟源主机的发送数据总量相等，这样在排序时它们的顺序将不确定。或许，你也可以简化这个程序。子表达式\$total_bytes{\$source} 在 for 循环的大量输出中使用过多次（在输入循环中也使用了两次）。它们可以由一个简单的标量替换，并且该标量在循环开始时初始化：

```

for my $source (@sources) {
    my $tb = $total_bytes{$source};
    my @destinations = sort { $tb->{$b} <=> $tb->{$a} } keys %$tb;
    print "$source: $tb->{$all} total bytes sent\n";
    for my $destination (@destinations) {
        next if $destination eq $all;
        print " $source => $destination: $tb->{$destination} bytes\n";
    }
    print "\n";
}

```

这使得代码更加简短并且（可能）运行速度也稍微快一点。如果你确信这样做会提高运行效率，就给自己更多的信心。如果你感觉这样做可能使代码变得让人困惑进而决定不进行更改那么也给自己更多的信心。

练习 3

我们先从本章中的数据聚合程序开始：

```

my %total_bytes;
while (<>) {
    my ($source, $destination, $bytes) = split;
    $total_bytes{$source}{$destination} += $bytes;
}

```

在填完%total_bytes之后，我们遍历它并输出每个层级。我们对于顶层的键进行排序并输出，然后得到第二层级的散列并把它放到\$dest_hash中，这样可以方便阅读。在foreach循环的内层嵌套一个foreach循环做同样的事情，但是我们会报告目的主机和字节总数。

```
foreach my $source ( sort keys %total_bytes ) {  
    print "$source\n";  
    my $dest_hash = $total_bytes{$source};  
    foreach my $dest ( sort keys %$dest_hash ) {  
        print " $dest $dest_hash->{$dest}\n";  
    }  
}
```

第 6 章答案

练习 1

该练习与第 5 章中的类似，但是在里使用了 Storable 模块：

```
use Storable;  
  
my $all      = "**all machines**";  
my $data_file = "total_bytes.data";  
  
my %total_bytes;  
if (-e $data_file) {  
    my $data = retrieve $data_file;  
    %total_bytes = %$data;  
}  
  
while (<>) {  
    next if /^#/;  
    my ($source, $destination, $bytes) = split;  
    $total_bytes{$source}{$destination} += $bytes;  
    $total_bytes{$source}{$all}       += $bytes;  
}  
  
store \%total_bytes, $data_file;  
  
### remainder of program is unchanged
```

在程序的开始处，我们把文件名放入一个变量当中，之后我们只能够在数据文件存在时检索到数据。

在读取到数据之后，我们再次使用 Storable 模块将数据写回到同一个磁盘文件当中。

如果我们选择通过代码和文件格式来将散列的数据避易就难地写到一个文件中，那我们就太努力工作了。更重要的是，除非我们才华出众，或者在这个练习中花了很长长时间，否则在序列化例程中肯定会存在一些 bug，或者至少在文件格式中存在缺陷。

我们还可以做得更多。或许应该有一些检查，以确保 Storable 模块的写入是成功的。它

会捕获一些错误（和终止），但有时候它将返回 `undef`。因此，我们应该检查 `store` 和 `retrieve` 的返回值。

程序应将原来的数据文件（如果有）保存在一个备份文件中，这样就很容易恢复之前的内容。事实上，它甚至可以保存几个备份，例如上周的重要内容。

在没有任何新的输入数据时，能够输出也是很有用的。正如我们当前所拥有的，我们可以通过传递一个空文件（如`/dev/null`）作为输入。然而，应该有一个更简单的方法。可以使得输出功能完全与更新分离。

练习 2

如果我们想使用 JSON 模块而不是 Storable 模块，就有一些工作是我们必须做的。我们需要分别从文件中读取数据然后分别重建数据。由于 JSON 数据是 UTF-8 格式的，因此我们必须指定编码格式才能正确地读取数据。*Learning Perl* 一书介绍了这部分内容，但第 8 章对它进行更多的说明。

`decode_json` 函数使用一个 UTF-8 编码的字符串，这个字符串跟我们平常所说的字符串是不一样的。我们需要文件中的原始数据，并将它作为单个字符串。`decode_json` 返回一个散列引用，所以在以前我们使用一个散列的地方现在变成了一个标量：

```
use JSON;

my $all      = "**all machines**";
my $data_file = "total_bytes.json";

my $total_bytes; # need a reference now!
if (-e $data_file) {
    local $/;
    open my $fh, '<:raw', $data_file;
    $json_text = <$fh>;
    $total_bytes = decode_json( $json_text );
}
```

程序的中间部分几乎是相同的，除了类型变成了引用而不再是散列：

```
while (<>) {
    next if /^#/;
    my ($source, $destination, $bytes) = split;
    $total_bytes->{$source}{$destination} += $bytes;
    $total_bytes->{$source}{$all}      += $bytes;
}
```

当完成后，我们将数据写入文件，同时确保将其编码设置为 UTF-8。为了方便阅读，我们可以给 `to_json` 函数添加 `pretty` 标记：

```
{
    open my $fh, '>:utf8', $data_file;
    print $fh to_json( $total_bytes, { pretty => 1 } );
}
```

程序的剩余部分是相同的：

```

foreach my $source ( sort keys %$total_bytes ) {
    print "$source\n";
    my $dest_hash = $total_bytes->{$source};
    foreach my $dest ( sort keys %$dest_hash ) {
        print " $dest $dest_hash->{$dest}\n";
    }
}

```

程序的输出部分如下所示：

```

{
    "maryann.girl.hut" : {
        "maryann.girl.hut" : 185108,
        "gilligan.crew.hut" : 248228,
        "thurston.howell.hut" : 257710,
        "ginger.girl.hut" : 208854,
        "professor.hut" : 165854,
        "***all machines***" : 1512000,
        "laser3.copyroom.hut" : 251704,
        "lovey.howell.hut" : 194542
    },
    "gilligan.crew.hut" : {
        "maryann.girl.hut" : 247398,
        "thurston.howell.hut" : 207276,
        "gilligan.crew.hut" : 351378,
        "ginger.girl.hut" : 251480,
        "professor.hut" : 260128,
        "***all machines***" : 1718064,
        "laser3.copyroom.hut" : 224528,
        "lovey.howell.hut" : 175876
    },
    ...
}

```

第 7 章答案

练习 1

可以从 <http://www.intermediateperl.com/> 网站的 Download 部分下载的程序框架开始，如下所示，然后添加 gather_mtime_between 子例程：

```

sub gather_mtime_between {
    my($begin, $end) = @_;
    my @files;
    my $gatherer = sub {
        my $timestamp = (stat $_[0])[9];
        unless (defined $timestamp) {
            warn "Can't stat $File::Find::name: $!, skipping\n";
            return;
        }
        push @files, $File::Find::name
            if $timestamp >= $begin and $timestamp <= $end;
    };
    my $fetcher = sub { @files };
    ($gatherer, $fetcher);
}

```

主要的挑战是正确获取项的名称。当在回调函数中使用 stat 函数时，使用的文件名是 \$_，并不包含路径，但当返回文件的名称（或者报告给用户）时，却是 \$File::Find::name，这个名称包含当前文件的完整路径。

如果因为某些原因 stat 函数执行失败了，那么 timestamp 变量将会是 undef。这是有可能发生的，例如，当它找到一个空符号链接（无效链接）时。在这种情况下，回调函数就会警告用户并提前返回。如果我们忘了检查，那么在与 \$begin 和 \$end 比较的过程中，我们就会获得一个关于未定义值的警告。

当我们运行完包含上面这个子例程的完整程序时，输出应该显示文件的修改日期是之前的某个星期一（除非我们改变了代码，使用了一周的另外一天）。

第 8 章答案

练习 1

在这个练习中，我们必须使用三个不同的输出方法：输出到一个文件，输出到一个标量（此时需要用到 v5.8 及后续版本），或同时输出到这两者。诀窍是将输出文件的句柄存储在同一个变量中，我们将在 print 语句中使用该变量。当文件句柄是一个变量时，我们可以将我们喜欢的任何东西赋值给它，而且我们也可以在运行时决定把什么内容赋给它：

```
use IO::Tee;
use v5.8;

my $fh;
my $scalar;

print 'Enter type of output [Scalar/File/Tee]> ';
my $type = <STDIN>;

if( $type =~ /^s/i ) {
    open $fh, '>', \$scalar;
}
elsif( $type =~ /^f/i ) {
    open $fh, '>', "$0.out";
}
elsif( $type =~ /^t/i ) {
    open my $file_fh, '>', "$0.out"
        or die "Could not open $0.out: $!";
    open my $scalar_fh, '>', \$scalar;
    $fh = IO::Tee->new( $file_fh, $scalar_fh );
}

my $date      = localtime;
my $day_of_week = (localtime)[6];

print $fh <<"HERE";
```

```

This is run $$  

The date is $date  

The day of the week is $day_of_week  

HERE  

print STDOUT <<"HERE" if $type =~ m/^$st/i;  

Scalar contains:  

$scalar  

HERE

```

在这个程序中，我们会提示用户输出的类型，并且我们希望他们的输入类型为“标量”、“文件”或者“tee”（同时输出到文件和屏幕）。一旦我们读取到用户的输入，通过首字符匹配的方法（使用不区分大小写的匹配会更加灵活），我们就能够发现他们所输入的是哪一种类型。

如果用户选择“标量”，我们为\$fh 打开一个标量的引用。如果他们选择“文件”，如前所述，我们会为\$fh 打开一个文件。我们会用程序中\$0 这个变量存储的程序名称来命名该文件，然后加上.out 后缀。如果用户选择“tee”，我们会同时为文件和标量创建文件句柄，然后将两者结合起来放在 IO::Tee 对象中，该对象存储在\$fh 中。不管用户选择哪种方法，但最终转出信道（单个或多个）都会出现在同一个变量之中。

由此开始，就只是编程方面的问题，它与我们实际上输出的东西并无太大联系。在这个练习中，我们通过在标量上下文中使用 localtime 来获得日期字符串，然后得到那一天是星期几的字面量列表切片。

在输出到\$fh 的字符串中，包括进程 ID（包含在特殊变量\$\$中）。因此我们可以分辨程序在不同的运行之间的差别，以及当时的日期和星期几。

最后，如果我们选择把内容输出到一个标量（或者单独发送到标量或者同时发送到一个文件），我们将标量的值输出到 STDOUT 以确保正确的事情在此结束。

练习 2

对于这个问题，我们需要同时保持多个打开的文件句柄：

```

my %output_handles;  

while (<>) {  

    unless (/^([:]+):/) {  

        warn "ignoring the line with missing name: $_";  

        next;  

    }  

    my $name = lc $1;  

    unless( $output_handles{$name} ) {  

        open my $fh, '>', "$name.info"  

        or die "Cannot create $name.info: $!";  

        $output_handles{$name} = $fh;  

    }  

    print { $output_handles{$name} } $_;  

}

```

在 while 循环的开始，我们使用一个模式来从数据行提取人的名字，而且能在没找到的情况下发出警告。

一旦你有了名字，就强制其转换为小写，这样如果输入“Ginger”，将会和“GINGER”储存到同一个地方。这样的操作对文件命名也是很方便的，就像下一条语句所展示的。

每次我们遇到一个名字，我们就会检查在散列%output_handles 中是否已经存储一个文件句柄。如果找不到，就创建一个，并将它存储在这个散列中。

此后，我们将该散列输出到文件句柄。因为我们通过其散列项访问文件句柄，所以我们要将文件句柄参数加上花括号。

练习 3

这里有一种实现方法。首先，遍历@ARGV 数组中的参数，找出哪些不是目录的参数，然后分别输出其中的错误消息。

此后，再次检查@ARGV 数组以找到哪些元素代表有效目录。我们获得由 grep 查找出来的元素组成的列表，并将其发送到 map 函数，在 map 函数中，我们使用 opendir 函数创建目录句柄（尽管我们跳过了错误检查部分）。把文件输出列表最终放入@dir_hs 里面，我们使用 foreach 循环遍历这个列表，然后将得到的目录作为参数传递给 print_contents。

对于 print_contents 子例程没有什么特别之处。首先它读取传入的第一个目录参数，并将其存储在\$dh 中，然后通过该参数来遍历整个目录：

```
my @not_dirs = grep { ! -d } @ARGV;
foreach my $not_dir ( @not_dirs ) {
    print "$not_dir is not a directory!\n";
}

my @dirs = grep { -d } @ARGV;
my @dir_hs = map { opendir my $dh, $_; $dh } grep { -d } @ARGV;
foreach my $dh ( @dir_hs ) { print_contents( $dh ) };

sub print_contents {
    my $dh = shift;

    while( my $file = readdir $dh ) {
        next if( $file eq '.' or $file eq '..');
        print "$file\n";
    }
};
```

第 9 章答案

练习 1

rightmost 程序大部分的内容相同。我们按照如下方式初始化为相同模式散列：

```
my %patterns = (
    Gilligan => qr/(?:Willie )?Gilligan/,
    'Mary Ann' => qr/Mary Ann/,
    Ginger => qr/Ginger/,
    Professor => qr/(?:The )?Professor/,
    Skipper => qr/Skipper/,
    'A Howell' => qr/Mrs?. Howell/,
);

```

尽管如此，我们并不是直接向 rightmost 传递模式，而是以引用的方式传递整个散列，所以 rightmost 子例程能够识别键和值：

```
my $key = rightmost(
    'There is Mrs. Howell, Ginger, and Gilligan, Skipper',
    \%patterns
);
say "Rightmost character is $key";

```

在 rightmost 子例程中，我们使用从参数 patterns 中获得的每一个键值对。这里我们至少需要使用 Perl v5.12 来处理这个数组。但处理散列就不需要。与之前一样，我们通过使用模式的值作为模式来匹配字符串。我们能够获得最佳匹配的位置，同时我们也能知道匹配处的键。在最后，我们返回这个键而不是匹配位置：

```
sub rightmost {
    my( $string, $patterns ) = @_;
    my( $rightmost_position, $rightmost_key ) = ( -1, undef );
    while( my( $key, $value ) = each %$patterns ) {
        my $position = $string =~ m/$value/ ? $-[0] : -1;
        if( $position > $rightmost_position ) {
            $rightmost_position = $position;
            $rightmost_key = $key;
        }
    }
    return $rightmost_key;
}

```

练习 2

本练习包括两部分。首先，我们从一个文件中读取模式的列表，并把它们放到 patterns 数组中。我们读文件的每一行，调用 chomp 函数消除换行符，然后使用 qr// 操作符来预编译模式。我们使用 eval 函数捕获无效的模式：

```

open my $fh, '<', 'patterns.txt'
    or die "Could not open patterns.txt: $!";

while( <$fh> ) {
    chomp;
    my $pattern = eval { qr/$_/ };
    or do { warn "Invalid pattern: $@"; next };
    push @patterns, $pattern;
}

```

我们逐行读取数据，然后针对它测试每一个模式：

```

while( <> ) {
    foreach my $pattern ( @patterns ) {
        print "Match at line $. | $_" if /$pattern/;
    }
}

```

如果我们的要求很简单，我们就可以仅仅报告一个匹配，一旦我们寻找到一个匹配模式就可以停止。我们也可以在找到一个匹配模式的时候直接跳到下一行：

```

LINE: while( <> ) {
    foreach my $pattern ( @patterns ) {
        if( /$pattern/ ) {
            print "Match at line $. | $_" if /$pattern/;
            next LINE;
        }
    }
}

```

如果我们想知道跟哪个模式能够匹配，我们就可以将模式插入输出中：

```

while( <> ) {
    foreach my $pattern ( @patterns ) {
        print "Match of [${$pattern}] at line $. | $_" if /$pattern/;
    }
}

```

练习 3

对于第一部分，使用 `Regexp::Assemble` 模块就像使用数组一样，只是没有数组变量。我们读取一行输入，然后使用 `Regexp::Assemble` 将它添加到`$ra` 对象中：

```

use Regexp::Assemble;

open my $fh, '<', 'patterns.txt'
    or die "Could not open patterns.txt: $!";

my $ra = Regexp::Assemble->new;

while( <$fh> ) {
    chomp;
    $ra->add( $_ );
}

```

当我们添加了所有模式之后，我们就得到整个模式。如果我们想看看 `Regexp::Assemble` 模块为我们创建了什么，我们就可以将其输出：

```
my $overall = $ra->re;
print "Regexp is: $overall\n";
```

当我们读取每一行时，我们只有一个检查：

```
while( <> ) {
    print "Match at line $. | $_" if /$overall/;
}
```

第 10 章答案

练习 1

施瓦茨变换保存了我们后续将要用到的-s 计算结果。第一个 map 函数产生了一个储存初始文件名和大小的匿名数组。然后利用 sort 函数按照文件大小的顺序对这个匿名数组进行排序。最后一个 map 函数将获得文件名并作为唯一的内容输出到@sorted 数组：

```
use v5.10;
chdir;
my @sorted =
  map  $_->[0],
  sort { $a->[1] <=> $b->[1] }
  map  [$_, -s $_],
  glob '*';
say join "\n", @sorted;
```

练习 2

一个基本的 Benchmark 程序按照如下方式使用 timethese 程序，将以标签作为键、代码引用作为值存到一个散列引用中：

```
use Benchmark qw(timethese);
timethese( -2, {
  LABEL => CODE_REF,
  LABEL => CODE_REF,
  ...
});
```

因为我们想比较做同样事情的两个代码段，所以我们使这两个匿名子例程对相同的文件列表进行排序。首先我们要创建文件列表，因为我们不希望把对 glob 的调用作为比较的一部分^{注1}：

```
chdir;
my @files = glob '*';
print 'There are ' . @files . " files to compare\n";
```

注 1：这其实是很重要的，正如我们在 http://www.perlmonks.com/?node_id=393128 中所解释的一样。为了获得关于基准测试的更多用法，请参考 *Mastering Perl* 一书中的“Benchmarking”章节。

```

my $ordinary = sub {
    my @sorted = sort { -s $a <=> -s $b } @files;
};

my $transform = sub {
    my @sorted =
        map $_->[0],
        sort { $a->[1] <=> $b->[1] }
        map [$_, -s $_],
        @files;
};

```

现在我们调用 timethese 方法：

```

timethese( -2, {
    Ordinary => $ordinary,
    Schwartzian => $transform,
});

```

输出表明即使数量很小的文件，结果也是非常庞大的：

```

There are 21 files to compare
Benchmark: running Ordinary, Schwartzian for at least 2 CPU seconds...
Ordinary: 2 wallclock secs ( 0.51 usr + 1.66 sys = 2.17 CPU) @ 3539.17/s (n=7680)
Schwartzian: 2 wallclock secs ( 1.28 usr + 0.79 sys = 2.07 CPU) @ 8618.36/s (n=17840)

```

练习 3

为了进行“词典”排序，当我们使用普通的施瓦茨变换时，我们需要删除那些非字母字符，并将要进行比较的字符串都转换为小写

```

my @dictionary_sorted =
    map $_->[0],
    sort { $a->[1] cmp $b->[1] }
    map {
        my $string = $_;
        $string =~ tr/A-Z/a-z/;
        $string =~ tr/a-z//cd;
        [ $_, $string ];
    } @input_list;

```

在第二个 map 函数中，即先执行的 map 函数，首先复制`$_`。（如果我们不这样做，就会损坏原始数据）。

实际上，这不是一种做这类事情的好方式。我们可以写一整本书来介绍排序。首先，可以假定所有字符都是 ASCII 字符，但是世界并不是由 ASCII 字符组成的。可以先删除非字母字符，这样我们就匹配了非 Unicode 属性，`\P{Letter}`。此后，我们使用来自于 Perl v5.16 的 `fc` 函数，做一些合适的大小写转换：

```

use v5.16;

my @dictionary_sorted =
    map $_->[0],
    sort { $a->[1] cmp $b->[1] }
    map {
        my $string = $_;

```

```
$string =~ s/\P{Letter}//g; # remove nonletters
$string = fc( $string );    # a proper case fold
[ $_, $string ];
} @input_list;
```

通常，这仍然是不够好的，当使用它并不能解决问题的时候，可以使用 `Unicode::Collate` 模块，本书不详细介绍这个部分。

练习 4

在本练习中，使用本章先前的 `data_from_path` 程序创建一个递归子例程，用来遍历一个目录结构。可以从 <http://www.intermediateperl.com/> 网站的 Download 部分获得这个程序。我们的目的是获取目录的结构并生成一个逐行缩进的报告。

我们想要按照如下方式调用这个函数：指定路径、其散列值（不管是未赋值的还是另一个散列引用），及其缩进级别：

```
dump_data_for_path( $path, $data, $level );
```

在子例程中，首先将参数放入命名的变量中。可以立即输出路径，但是这里使用了字符串复制运算符 `x`，它的后面紧跟添加前导空格的缩进级别：

```
sub dump_data_for_path {
    my $path = shift;
    my $data = shift;
    my $path = shift || 0;

    print ' ' x $level, $path;
```

此后，我们来看看`$ data`。如果它不是一个已经定义的值，就转出一个换行符并返回：

```
if( not defined $data ) { # plain file
    print "\n";
    return;
}
```

如果`$ data`是一个已定义的值，它应该是一个散列引用，这样我们将检查它是否有键。虽然一个空目录会生成一个空散列，但它仍然是一个散列。对于每个键，再次调用 `dump_data_for_path` 函数：

```
if( keys %$data ) {
    print ", with contents of:\n";
    foreach (sort keys %$data) {
        dump_data_for_path( $_, $data->{$_}, $indent + 1 );
    }
} else {
    print ", an empty directory\n";
}
```

练习 5

在前面的练习中，使用了递归，尽管不需要使用它。也可以使用迭代的方案来完成这个任务，这将带来很大的灵活性。诀窍就是使用一个队列，该练习希望我们允许迭代我们所

创建的 data_for_path 函数，使用一个广度优先或深度优先搜索。这其实也就是将新元素放到了队列的前面或后面处理（或者甚至不把它们放入队列中）。

这可以用很多种方式实现。对我们而言，最简单的方法也许就是给每种技术编写一个单独的子例程，然后用一个更高级别的子例程来选择调用哪种方式：

```
sub breadth_first {
    ...
    push @queue, ...
    ...
}

sub depth_first {
    ...
    unshift @queue, ...
    ...
}

sub data_for_path {
    my( $path, $type ) = @_;
    if( $type eq 'depth' ) { depth_first( $path ) }
    else                   { breadth_first( $path ) }
}
```

然而，在子例程 breadth_first 和 depth_first 之间，有很多相同的代码。可以先检测遍历类型而不是两个分离的子例程，然后选择调用正确的子例程：

```
sub data_for_path {
    my( $path, $type ) = @_;
    ...
    if( $type eq 'depth' ) { unshift @queue, ... }
    else                  { push @queue, ... }
    ...
}
```

如下即为这个想法的完整实施：

```
sub data_for_path {
    my( $path, $type ) = @_;

    my $data = {};

    my @queue = ( [ $path, 0, $data ] );

    while( my $next = shift @queue ) {
        my( $path, $level, $ref ) = @$next;

        my $basename = basename( $path );

        $ref->{$basename} = do {
            if( -f $path or -l $path ) { undef }
            else {
                my $hash = {};
                opendir my $dh, $path;
                my @new_paths = map {
```

```

        catfile( $path, $_ )
    } grep { ! /^\.|\.\?|\.z/ } readdir $dh;
    if( $type eq 'depth' ) {
        unshift @queue, map { [ $_, $level + 1, $hash ] } @new_paths;
    }
    else {
        push @queue, map { [ $_, $level + 1, $hash ] } @new_paths;
    }
    $hash;
}
};

$data;
}

```

还可以采用其他一些技巧，这样就不需要每次都检查一遍。可以生成一个代码引用，使其要么指向一个将数据添加到数组前面的子例程，要么指向一个将数据添加到数组后面的子例程：

```

sub data_for_path {
    my( $path, $type ) = @_;
    my $coderef = $type eq 'depth' : \&add_to_front : \&add_to_front;
    ...
    $coderef->( \@queue, ... );
    ...
}

```

我们把剩下的工作留给你来完成。

第 11 章答案

练习 1

下面给出一种解决这个问题的方法。从 package 指令和 strict 模式开始：

```

package Oogaboogoo::Date;
use strict;

```

然后，定义常量数组来保存每一天在一周内的映射和月份名称的映射：

```

my @day = qw(ark dip wap sen pop sep kir);
my @mon = qw(diz pod bod rod sip wax lin sen kun fiz nap dep);

```

接下来，定义一个子例程，这个子例程可以把一周中的第几天转化为名称，并且这个子例程可以作为 Oogaboogoo::Date::day 访问：

```

sub day {
    my $num = shift @_;
    die "$num is not a valid day number"
    unless $num >= 0 and $num <= 6;
    $day[$num];
}

```

同样，可以定义一个子例程，可以把一年中哪几个月转化为月份：

```
sub mon {
    my $num = shift @_;
    die "$num is not a valid month number"
        unless $num >= 0 and $num <= 11;
    $mon[$num];
}
```

最后，在包的末尾，可以得到强制性的真值：

```
1;
```

把这个文件命名为 Date.pm，并把它保存到 Oogaboogoo/目录下面，Oogaboogoo 目录位于@INC 变量所包含的路径下面，比如当前目录。

练习 2

下面给出一种处理方法。从@INC 所包含的路径中的某一个地方将.pm 文件加载进来：

```
use strict;
require 'Oogaboogoo/Date.pm';
```

我们可以获得当前的时间信息：

```
my($sec, $min, $hour, $mday, $mon, $year, $wday) = localtime;
```

然后我们使用新定义的子例程进行转换：

```
my $day_name = Oogaboogoo::Date::day($wday);
my $mon_name = Oogaboogoo::Date::mon($mon);
```

因为历史原因，获得的年份是减去 1900 之后的值，所以我们需要解决这个问题：

```
$year += 1900;
```

最终，输出的时间为：

```
print "Today is $day_name, $mon_name $mday, $year.\n";
```

第 12 章答案

练习 1

我们利用 module-starter 来新建一个要发行的模块：

```
% module-starter --module=Animal --name=Gilligan --email="gilligan@example.net"
```

我们切换到 Animal 目录下，并运行构建文件：

```
% cd Animal
% perl Build.PL
Created MYMETA.yml and MYMETA.json
Creating new 'Build' script for 'Animal' version '0.01'
```

在做任何事之前，我们需要运行测试程序。在开始你的工作之前先测试代码是一个好主意，这也确保你拥有一个好的开始。由于我们什么都还没有做，因此每一项都应能通过：

```
% ./Build test
t/00-load.t ..... ok
t/boilerplate.t ... ok
t/manifest.t ..... skipped: Author tests not required
t/pod-coverage.t .. ok
t/pod.t ..... ok
All tests successful.
Files=5, Tests=6, 1 wallclock secs ( ... )
Result: PASS
```

我们可以通过某种方式改变 lib/Animal.pm 来生成一个语法错误。例如，在 use strict 后面移除了分号：

```
use 5.006;
use strict # syntax error
use warnings;
```

再次运行测试， t/00-load.t 测试捕获以下错误：

```
% ./Build test
t/00-load.t ..... 1/1 Bailout called. Further testing stopped:

# Failed test 'use Animal;';
# at t/00-load.t line 6.
# Tried to use 'Animal'.
# Error: syntax error at lib/Animal.pm line 5, near "use strict
# use warnings"
# Compilation failed in require at (eval 4) line 2.
FAILED--Further testing stopped.
```

很高兴我们的测试能够捕获错误。因为我们并不希望程序在不能正常工作的时候通过测试。

练习 2

在根目录.module-starter/ config (或者我们放在 MODULE_STARTER_DIR 中的位置) 下创建 module-starter 配置文件：

```
author: Willie Gilligan
email: gilligan@island.example.com
builder: Module::Build
verbose: 1
```

现在我们调用 module-starter 将不再那么麻烦：

```
% module-starter --module=Animal
```

练习 3

我们使用第 2 章介绍的任意一种技术安装 Module::Starter::AddModule：

```
% cpan -I Module::Starter::AddModule
```

一旦安装了这个模块，就可以更新.module-starter/config 配置文件来使用新安装的插件块：

```
author: Willie Gilligan
email: gilligan@island.example.com
builder: Module::Build
verbose: 1
plugins: Module::Starter::AddModule
```

现在我们可以很容易地添加另一个模块。如果我们在发行目录中，`--dist` 参数就仅是“.”。

```
% module-starter --module=Cow --dist=.
```

第 13 章答案

练习 1

我们应该已经把 `Animal` 类发布版放在了合适的位置，但是如果我们没有这么做，就创建一个新的发布版，并把这个发布版声明给所有我们想要进行此操作的类：

```
% module-starter --module=Animal,Cow,Horse,Sheep
```

如果我们已经有带 `Animal` 类的发布，可以添加模块：

```
% module-starter --module=Cow,Horse,Sheep --dist=.
```

这是我们唯一可以做的。我们在 `lib/Animal.pm` 中定义了带有 `speak` 方法的 `Animal` 类。对于这个答案，我们仅仅会给出简写形式的代码位，但是实际上对于该代码位，我们是有文档记录的。

```
package Animal;
our $VERSION = '0.01';
sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
}
1;
```

我们定义了每个具有其明确 `sound` 方法的子类。

在 `lib/Cow.pm` 文件中：

```
package Cow;
our $VERSION = '0.01';
use parent qw(Animal);
sub sound { 'moooo' }
1;
```

在 `lib/Horse.pm` 文件中：

```
package Horse;
our $VERSION = '0.01';
use parent qw(Animal);
sub sound { 'neigh' }
1;
```

在 `lib/Sheep.pm` 文件中：

```
package Sheep;
our $VERSION = '0.01';
use parent qw(Animal);
sub sound { 'baaaah' }
1;
```

Mouse 包因为额外安静，所以略有不同：

```
package Mouse;
our $VERSION = '0.01';
use parent qw(Animal);
sub sound { 'squeak' }
sub speak {
    my $class = shift;
    $class->SUPER::speak;
    print "[but you can barely hear it!]\n";
}
1;
```

不过我们还没有完成。为了确保我们没有任何语法错误，我们使用 t/00-load.t 来做一个测试。这个测试只检测模块是否正确地编译。如果我们从 Animal 类开始，它将只能检查 lib/Animal.pm。

```
#!/perl -T

use Test::More tests => 1;

BEGIN {
    use_ok('Animal') || print "Bail out!\n";
}

diag("Testing Animal $Animal::VERSION, Perl $], $^X");
```

我们做一个更新以此来检查新类：

```
#!/perl -T

use Test::More tests => 5;

BEGIN {
    foreach my $class ( qw(Animal Cow Horse Sheep Mouse) ) {
        use_ok($class)
            or print "Bail out! $class does not compile!\n";
    }
}
```

使用新测试，再次运行测试：

```
% perl Build.PL
% ./Build test
t/00-load.t ..... 1/1
# Testing Animal , Perl 5.014002, /usr/local/perls/perl-5.14.2/bin/perl
t/00-load.t ..... ok
...
```

如果 t/00-load.t 测试并未通过，我们修复失败的模块并且再试一次。其他测试比如 (t/boilerplate.t 和 t/pod.t) 也可能失败，所以我们修复模块直到它们也通过。

练习 2

现在我们需要一个可以使农庄群声鼎沸的程序，该程序会放入到 script/barnyard.pl 文件中。我们加载了所有需要的模块：

```
use Cow;
use Horse;
use Mouse;
```

```

use Sheep;

my @barnyard = ();
{
    print "enter an animal (empty to finish): ";
    chomp(my $animal = <STDIN>);
    $animal = ucfirst lc $animal;           # canonicalize
    last unless $animal =~ /^(Cow|Horse|Sheep|Mouse)$/;
    push @barnyard, $animal;
    redo;
}

foreach my $beast (@barnyard) {
    $beast->speak;
}

```

这段代码通过一个模式匹配来使用一个简单的检查，以此确保用户没有输入 Alpaca 或者其他不可用的动物，因为如果用户这样做，会导致程序崩溃。

最后，运行程序，得到如下输出：

```

% perl scripts/barnyard.pl
enter an animal (empty to finish): Cow
enter an animal (empty to finish): Cow
enter an animal (empty to finish): Sheep
enter an animal (empty to finish): Horse
enter an animal (empty to finish): Kangaroo
A Cow goes moo!
A Cow goes moo!
A Sheep goes baaaah!
A Horse goes neigh!

```

最后的动物——Kangaroo，跳出循环，并且使它不再进入@barnyard，这就是为什么我们不能够听到它叫。

练习 3

如果我们已经有带 Animal 类的发布版，我们可以添加一个 LivingCreature 和一个 Person 模块：

```
% module-starter --module=LivingCreature,Person --dist=.
```

我们更新 t/00-load.t：

```

#!/perl -T

use Test::More tests => 7;

BEGIN {
    my @classes = qw(Animal Cow Horse Sheep Mouse
                     LivingCreature Person);
    foreach my $class ( @classes ) {
        use_ok( $class )
            or print "Bail out! $class does not compile!\n";
    }
}

```

这有一个方法来实现它。首先，创建一个具有简单的 speak 方法的 LivingCreature 基类：

```
package LivingCreature;
our $VERSION = '0.01';
sub speak {
    my $class = shift;
    if (@_) {           # something to say
        print "a $class goes '@_'\n";
    } else {
        print "a $class goes ", $class->sound, "\n";
    }
}
1;
```

因为一个 person 是一个活着的生物，所以我们继承自 LivingCreature 类并定义一个简单的 sound：

```
package Person;
use parent qw(LivingCreature);
sub sound { "hmmmm" }
```

接下来是 Animal 类，同样继承自 LivingCreature 类。因为在 LivingCreature 类中现在已经没有主要的 speak 例程，所以我们不需要再在 Person 类中把它写一遍。但是在 Animal 类中，你需要检查一下以确保一个 Animal 不会在调用 SUPER::speak 之前就尝试 speak：

```
package Animal;
use parent qw(LivingCreature);
sub sound { die "all Animals should define a sound" }
sub speak {
    my $class = shift;
    die "animals can't talk!" if @_;
    $class->SUPER::speak;
}
```

其他的 Animal 类保持不变。只要能够得到结果，它们并不在意 Animal 类如何实现该过程。

最后，创建 scripts/person.pl。加载新的 Person 类并且调用其 speak 方法：

```
use Person;

Person->speak;
Person->speak("Hello, world!");
```

当运行它时，我们可以看到 Person 发出了它们的默认声音以及我们赋予它们的话“Hello World”。

```
% perl scripts/person.pl
hmmmm
Hello, World!
```

第 14 章答案

练习 1

首先创建我们的发布版并且切换到我们的发布目录：

```
% module-starter --module=My::List::Util
% cd My-List-Util
```

并不是直接运行这些模块，首先要做的是测试。因为我们需要两个子例程：sum 和 shuffle，所以我们创建 t/sum.t 和 t/shuffle.t。在每一个文件里，我们要测试好的数据、坏的数据、非法接口，以及其他任何我们可以想到的会中断其过程的方式。下面是一个 t/sum.t 示例：

```
use Test::More;

BEGIN { use_ok( 'My::List::Util' ) }

ok( defined &My::List::Util::sum,
    'sum() is defined' );
is( My::List::Util::sum( 1, 2, 3 ), 6,
    '1+2+3 is six' );
is( My::List::Util::sum( qw(1 2 3) ), 6,
    '1+2+3 as strings is six' );
is( My::List::Util::sum( 4, -9, 37, 6 ), 38,
    '4-9+37+6 is six' );
is( My::List::Util::sum( 3.14, 2.2 ), 5.34,
    '3.14 + 2.2 is 5.34' );
is( My::List::Util::sum(), undef,
    'No arguments returns undef' );
is( My::List::Util::sum( qw(a b) ), undef,
    'All bad args gives undef' );
is( My::List::Util::sum( qw(a b 4 5) ), 9,
    'Some good args works' );
done_testing();
```

下面是一个 t/shuffle.t 示例：

```
use Test::More;

BEGIN { use_ok( 'My::List::Util' ) }

ok( defined &My::List::Util::shuffle, 'shuffle() is defined' );

{
    my @shuffled = My::List::Util::shuffle();
    is( scalar @shuffled, 0, 'No args returns an empty list' );
}

{
    my @array = 1 .. 10;
    my @shuffled = My::List::Util::shuffle( @array );
    is( scalar @array, scalar @shuffled,
        "The output list is the same size" );
    isnt( "@array", "@shuffled", "The list is shuffled" );
}

done_testing();
```

因为我们仍然没有实现这些子例程，所以这些测试都失败了。尤其是，ok 测试也失败了，这是因为我们甚至还没有定义这些子例程。我们对 lib/My/Util.pm 文件以及它

们的软件文档存根都进行了修复，所以 Pod 测试通过：

```
=head2 sum  
Returns the sum of the numbers passed to it, ignoring arguments  
that don't look like numbers.
```

```
=cut
```

```
sub sum {  
}
```

```
=head2 shuffle  
Returns a shuffled version of the list.
```

```
=cut
```

```
sub shuffle {  
}
```

我们再次运行这些测试，因为我们已经定义了子例程，所以情况应该变得更好点。现在我们需要去做一些略微困难的测试。这个独有的实现没有进程和测试那样重要。下面是我们可能需要对 sum 做的一些操作：

```
=head2 sum  
=cut  
sub sum {  
    my $sum;  
    foreach my $num ( grep { /\A-?\d+\.*\d*\z/ } @_ ) {  
        $sum += $num;  
    }  
    $sum;  
}
```

我们采用 perlfaq4 中的 fisher_yates_shuffle：

```
=head2 shuffle  
=cut  
sub shuffle {  
    my @deck = @_;  
    return unless @deck;  
  
    my $i = @deck;  
    while( --$i ) {  
        my $j = int rand ($i+1);  
        @deck[$i,$j] = @deck[$j,$i];  
    }  
    @deck;  
}
```

现在所有测试都应该可以通过。这并不意味着我们已经完成了或者代码是完全正确的。这是一个永无休止的战斗。

练习 2

这是一个容易的练习。我们采用这一章提到的一个测试，并且将其放入 t/Animal.t 文件中：

```
use strict;
use warnings;

use Test::More tests => 6;

BEGIN {
    use_ok( 'Animal' ) || print "Bail out!\n";
}

diag( "Testing Animal $Animal::VERSION, Perl $[, $^X" );

# they have to be defined in Animal.pm
ok( defined &Animal::speak, 'Animal::speak is defined' );
ok( defined &Animal::sound, 'Animal::sound is defined' );

# check that sound() dies
eval { Animal->sound() } or my $at = $@;
like( $at, qr/You must/, 'sound() dies with a message' );

# check that speak() dies too
eval { Animal->speak() } or my $at = $@;
like( $at, qr/You must/, 'speak() dies with a message' );

{
    package Foofle;
    use parent qw(Animal);
    sub sound { 'foof' }

    is(
        Foofle->speak,
        "A Foofle goes foof!\n",
        'An Animal subclass does the right thing'
    );
}
```

练习 3

我们需要为 Cow、Horse 和 Sheep 类编写测试程序，但是我们从仅测试 sound 方法开始。下面是 t/Horse.t 文件的代码：

```
use Test::More;

BEGIN { use_ok( 'Horse' ) }

is( Horse->sound, 'neigh', 'The horse make the right sound' );

done_testing();
```

其他的测试文件仅仅替换掉该动物及其发出的声音即可。

练习 4

在前面的练习中，创建了测试文件：

```
% ./Build testcover
```

测试之后，得到的结果为：

```
% cover
```

练习 5

我们不会直接显示代码。这个练习的诀窍是 cover 创建的 HTML 报告。通过对其的使用，你会知道你需要测试的是什么：

```
% ./Build testcover  
% cover  
% open cover_db/coverage.html
```

可能该功能对我们的系统并不可用（虽然这是因为我们使用的是 Mac OS X 系统）。我们最喜欢用的浏览器可能已经安装了另一个程序，或者我们可以通过 Web 浏览器的界面打开这个文件。当我们更新这段代码或者测试时，我们会重新运行该进程：

```
% ./Build testcover  
% cover  
% open cover_db/coverage.html
```

虽然这看起来工作量很大，但是我们在前期花费的时间越多后期越轻松。

第 15 章答案

练习 1

首先，从 Animal 包开始。named 方法检查该包是否获得一个字符串参数；我们想把它限制到一个类方法。我们预期一个名称参数，并且使用 default_color 来设置初始颜色：

```
package Animal;  
use Carp qw(croak);  
  
sub named {  
    ref(my $class = shift) and croak "class name needed";  
    my $name = shift;  
    my $self = { Name => $name, Color => $class->default_color };  
    bless $self, $class;  
}  
  
## backstops (should be overridden)  
sub default_color { "brown" }  
sub sound { croak "subclass must define a sound" }
```

接下来，我们定义由一个类或者一个实例来调用的多个方法：

```

sub speak {
    my $either = shift;
    print $either->name, " goes ", $either->sound, "\n";
}

sub name {
    my $either = shift;
    ref $either
    ? $either->{Name}
    : "an unnamed $either";
}

sub color {
    my $either = shift;
    ref $either
    ? $either->{Color}
    : $either->default_color;
}

```

最后，我们添加一些方法，这些方法只能被特定实例调用。如果第一个参数不是一个引用，它也不是一个实例，那么我们会使用 croak：

```

sub set_name {
    ref(my $self = shift) or croak "instance variable needed";
    $self->{Name} = shift;
}
sub set_color {
    ref(my $self = shift) or croak "instance variable needed";
    $self->{Color} = shift;
}

```

具体类（包括 Horse 类）都保持不变。而我们几乎没有为它们编写任何代码。此时，创建 scripts/mr_ed.pl 文件来测试我们的变化：

```

my $tv_horse = Horse->named("Mr. Ed");
$tv_horse->set_name("Mister Ed");
$tv_horse->set_color("grey");
print $tv_horse->name, " is ", $tv_horse->color, "\n";
print Sheep->name, " colored ", Sheep->color, " goes ", Sheep->sound, "\n";

```

第 16 章答案

练习 1

有多种方法可以解决这个问题。在解决方案中，在同一个文件中创建了一个 MyDate 包作为脚本。裸块定义了 MyDate 包中语句的作用域。在这之后，我们将不能够在脚本中使用这个模块，因为 Perl 解释器不能找到其中的文件。我们必须调用 import 方法来将符号导入 main 命名空间。

为了使 AUTOLOAD 子例程只适用于正确的子例程，我们定义了保存可行的方法名的 %Auowed_methods。它们的值是在 localtime 返回的列表中它们的偏移量。这个方法基本上解决了这个问题，但是 localtime 对于年和月使用基于 0 的数字。在 @Offsets 数组中，存储

这些数字，并把它们添加到了 localtime 列表相应的条目中。由于目前只有两个值有偏移量，因此工作量似乎很大，但这样做可以消除两种特殊情况。

我们需要一个 new 方法（或者某个构造函数）来为我们提供对象。在这个例子中，对象实际上是什么无关紧要。我们使用了一个通过 bless 操作进入当前包的空的匿名散列（这是在参数列表中的第一个元素，所以它是\$_[0]）。我们也知道我们需要一个 DESTROY 方法，因为当 Perl 尝试清理对象时，它将会查找它。如果我们没有 DESTROY 方法，当它尝试调用它自己的 DESTROY 方法时，AUTOLOAD 将会提示此处有一个未知方法（注释掉 DESTROY 来看看将会发生什么）。

在 AUTOLOAD 内部，在\$method 中储存了方法名，因此我们可以改变它。我们想去掉包的信息并且得到非限定方法名。这一切都在最后一个之后，所以我们使用了替换操作符来去掉该位置之前的所有内容。一旦我们得到了方法名，我们就会查找其在%Allowed_methods 中的键。如果方法名不在那里，就输出一个包含 carp 的错误。尝试调用一个未知方法，Perl 解释器会在哪一行报错？

如果我们在%Allowed_methods 中找到了方法名，我们会得到其值，即其在 localtime 列表中的位置。我们把这个值储存到\$slice_index 中，并且使用它来从 localtime 得到该值和该值的偏移量。我们把这个值相加并返回结果。

这听起来工作量似乎很大，但是为了给 hour 和 minute 添加新的方法，我们必须完成的工作究竟有多少？我们只需简单地给%Allowed_methods 添加这些名称。其他的一切都已经开始工作：

```
{  
    package MyDate;  
    use vars qw($AUTOLOAD);  
  
    use Carp;  
  
    my %Allowed_methods = qw( date 3 month 4 year 5 );  
    my @Offsets        = qw(0 0 0 0 1 1900 0 0 0);  
  
    sub new      { bless {}, $_[0] }  
    sub DESTROY  {}  
  
    sub AUTOLOAD {  
        my $method = $AUTOLOAD;  
        $method =~ s/.*://;  
  
        unless( exists $Allowed_methods{ $method } ) {  
            carp "Unknown method: $AUTOLOAD";  
            return;  
        }  
  
        my $slice_index = $Allowed_methods{ $method };  
  
        return (localtime)[ $slice_index ] + $Offsets[ $slice_index ];  
    }  
}
```

```

}

MyDate->import; # we don't use it
my $date = MyDate->new();

print "The date is " . $date->date . "\n";
print "The month is " . $date->month . "\n";
print "The year is " . $date->year . "\n";

```

练习 2

该脚本看起来和之前的对 UNIVERSAL::debug 例程添加的答案一样。在脚本结束时，在\$date 对象上调用了 debug 方法，无须改变 MyDate 模块，它就可以正常运行：

```

use MyDate;
my $date = MyDate->new();

sub UNIVERSAL::debug {
    my $self = shift;
    my $when = localtime;
    my $message = join '|', @_;
    print "[${when}] ${message}\n";
}

print "The date is " . $date->date . "\n";
print "The month is " . $date->month . "\n";
print "The year is " . $date->year . "\n";

$date->debug( "I'm all done" );

```

为什么 debug 方法不能使 AUTOLOAD carp? 记住，在 Perl 解释器开始查找任何 AUTOLOAD 方法之前，它会遍历所有的@ISA 和 UNIVERSAL。因此，在它必须使用 AUTOLOAD 魔法之前，Perl 解释器查找 UNIVERSAL::debug。

第 17 章答案

练习 1

模块 Oogabooogoo.pm 的代码如下：

```

package Oogabooogoo::Date;
use strict;
use Exporter qw(import);
our @EXPORT = qw(day mon);

my @day = qw(ark dip wap sen pop sep kir);
my @mon = qw(diz pod bod rod sip wax lin sen kun fiz nap dep);

sub day {
    my $num = shift @_;
    die "$num is not a valid day number"
    unless $num >= 0 and $num <= 6;
}

```

```

    $day[$num];
}

sub mon {
    my $num = shift @_;
    die "$num is not a valid month number"
        unless $num >= 0 and $num <= 11;
    $mon[$num];
}
1;

```

程序现在如下：

```

use strict;
use Oogabooogoo::Date qw(day mon);

my($sec, $min, $hour, $mday, $mon, $year, $wday) = localtime;
my $day_name = day($wday);
my $mon_name = mon($mon);
$year += 1900;
print "Today is $day_name, $mon_name $mday, $year.\n";

```

练习 2

大部分答案与前面的答案相同。我们只需要为导出标记“all”添加相关部分：

```

our @EXPORT = qw(day mon);
our %EXPORT_TAGS = ( all => \@EXPORT );

```

我们放进 %EXPORT_TAGS 中的任何东西也都要放入@EXPORT 或@EXPORT_OK 中。对 all 标记来说，我们直接对@EXPORT 使用一个引用。如果我们不喜欢那样，我们可以创建一个新的副本使这两者不会互相引用：

```

our @EXPORT = qw(day mon);
our %EXPORT_TAGS = ( all => [ @EXPORT ] );

```

我们修改以前练习里的程序，使其使用 import 标记“all”时在导入列表中在 all 前面加一个冒号。主要程序现在如下：

```

use strict;
use Oogabooogoo::Date qw(:all);

```

练习 3

我们对 My::List::Util 的改变是十分容易的。为了导出子例程，添加下面这两行：

```

use Exporter qw(import);
our @EXPORT = qw(sum shuffle);

```

在 t/sum.t 测试中，不需要在每一次调用 sum 前都添加 My::List::Util，这样代码会更容易阅读：

```

use Test::More;

BEGIN { use_ok( 'My::List::Util' ) }

ok( defined &sum, 'sum() is exported' );

```

```

is( sum( 1, 2, 3 ), 6, '1+2+3 is six' );
is( sum( qw(1 2 3) ), 6, '1+2+3 as strings is six' );
is( sum( 4, -9, 37, 6 ), 38, '4-9+37+6 is six' );
is( sum( 3.14, 2.2 ), 5.34, '3.14 + 2.2 is 5.34' );
is( sum(), undef, 'No arguments returns undef' );
is( sum( qw(a b) ), undef, 'All bad args gives undef' );
is( sum( qw(a b 4 5) ), 9, 'Some good args works' );
done_testing();

```

第 18 章答案

练习 1

首先，通过从 Horse 类继承开始写 RaceHorse 类的代码：

```

package RaceHorse;
use parent qw(Horse);

```

接下来，使用一个简单的 dbmopen 来连接%STANDINGS，使其可以永久储存：

```

dbmopen (our %STANDINGS, "standings", 0666)
    or die "Cannot access standings dbm: $!";

```

当命名一个新的 RaceHorse 类时，或者取出数据库中的现有排名或者把一切都初始化为 0：

```

sub named { # class method
    my $self = shift->SUPER::named(@_);
    my $name = $self->name;
    my @standings = split ' ', $STANDINGS{$name} || "0 0 0";
    @{$self{qw(wins places shows losses)}} = @standings;
    $self;
}

```

当对 RaceHorse 类执行 destroy 操作时，我们会更新排名来清除硬盘中储存在内存的对象中的内容：

```

sub DESTROY { # instance method, automatically invoked
    my $self = shift;
    $STANDINGS{$self->name} = "@$self{qw(wins places shows losses)}";
    $self->SUPER::DESTROY if $self->can('SUPER::DESTROY');
}

```

最后，定义 instance 方法来使值递增：

```

## instance methods:
sub won { shift->{wins}++; }
sub placed { shift->{places}++; }
sub showed { shift->{shows}++; }
sub lost { shift->{losses}++; }
sub standings {
    my $self = shift;
    join ", ", map "$self->{$_} $_", qw(wins places shows losses);
}

```

第 19 章答案

练习 1

我们不会在答案中写出整个发布版，但是可以从 <http://www.intermediateperl.com/> 中的 Download 部分获得所有内容。

从创建我们需要的模块的发布版开始：

```
% module-starter --module=Animal,Horse,Cow,Sheep,Mouse
```

当我们有模块存根时，我们把第 19 章的代码移植到模块文件。在删除了 Pod（对它我们做了更新）之后，基类 Animal 的代码如下所示：

```
package Animal;
use strict;
use warnings;

use Moose;

our $VERSION = '0.01';

has 'name'  => ( is => 'rw' );
has 'color' => ( is => 'rw' );
has 'sound' => ( is => 'ro', default => sub { 'Grrrr!' } );

sub speak {
    my $self = shift;
    print $self->name, " goes ", $self->sound, "\n";
}

__PACKAGE__->meta->make_immutable;

1;
```

Horse 类使用 extend 来继承自 Animal 类：

```
package Horse;
use strict;
use warnings;

use Moose;
use namespace::autoclean;

extends 'Animal';

has 'sound' => ( is => 'ro', default => 'neigh' );

__PACKAGE__->meta->make_immutable;

1;
```

对 Cow 和 Sheep 类也做同样的事情，具体视其特有的 sounds 方法而定。Mouse 类略有不同：

```

package Mouse;
use strict;
use warnings;

use Moose;
use namespace::autoclean;

extends 'Animal';

has 'sound' => ( is => 'ro', default => 'squeak' );

after 'speak' => sub {
    print "[but you can barely hear it!]\n";
};

__PACKAGE__->meta->make_immutable;

1;

```

对每一个类的测试都是相似的。下面是对于 Horse 类的一个测试，它位于 t/horse.t 中：

```

use Test::More;
use strict;
use warnings;

BEGIN { use_ok( 'Horse' ) }

can_ok( 'Horse', qw(new sound color name speak) );

my $horse = Horse->new( name => 'Mr. Ed' );
isa_ok( $horse, 'Horse' );
is( $horse->name, 'Mr. Ed', 'Got the name right' );

done_testing();

```

我们对于 t/cow.t、t/sheep.t 以及 t/mouse.t 执行类似的测试。

练习 2

在之前的练习中，动物都从 Animal 类继承而来。现在，我们想要把 Animal 当成一个角色。这与我们在第 19 章编写的代码相同。代码量要比非角色类要短，因为我们不用提供一个 sound 方法，并且也并不必定义一个默认的 Animal 包：

```

package Animal;
use Moose::Role;

requires qw( sound );

has 'name' => ( is => 'rw' );
has 'color' => ( is => 'rw' );

sub speak {
    my $self = shift;
    print $self->name, " goes ", $self->sound, "\n";
}

1;

```

Horse 类使用 with 而不再是 extends:

```
package Horse;
use strict;
use warnings;

use Moose;
use namespace::autoclean;

with 'Animal';

sub sound { 'neigh' }

__PACKAGE__->meta->make_immutable;

1;
```

t/animal.t 测试文件不得不略微改变，因为我们没有创建一个新的 Animal 类；这只是一个角色。我们现在将移除 t/animal.t 文件，因为我们还没有学习关于怎样测试的相关知识。

第 20 章答案

练习 1

如果我们还没有 Test::File，我们将安装它：

```
% cpan -I Test::File
```

我们会检查文档，来看它到底提供了什么子例程，并且找出 file_exists_ok 文件和 file_readable_ok 文件。因为我们想检查这两个文件中的一个，所以我们把 SKIP 块放到了这两个文件两侧，而且不论它们是或不是 Windows，都会执行 skip 操作。

```
use Test::More;
use Test::File;

my $unix_file    = '/etc/hosts';
my $windows_file = 'C:\\windows\\system32\\drivers\\etc\\hosts';

SKIP: {
    skip q(We're not on Windows), 1 unless $^O eq 'MSWin32';
    file_exists_ok( $windows_file );
    file_readable_ok( $windows_file );
}

SKIP: {
    skip q(We're not on Unix), 1 unless $^O ne 'MSWin32';
    file_exists_ok( $unix_file );
    file_readable_ok( $unix_file );
}

done_testing();
```

如果我们没有声明 SKIP 块，我们将只运行这些测试一次，但是会使用\$^O 选择待测试

的文件：

```
use Test::More;
use Test::File;

my $file = $^O eq 'MSWin32' ?
'C:\\windows\\system32\\drivers\\etc\\hosts'
:
'/etc/hosts.txt';

file_exists_ok( $file );
file_readable_ok( $file );

done_testing();
```

练习 2

通过使用在第 20 章中关于 Test::Minnow::RequiredItems 的例子，我们创建了一个 sum_ok 子例程，该子例程使用了一个 Test::Builder 对象来处理测试结果。省略了文档的 Test::My::List::Util 如下所示：

```
package Test::My::List::Util;
use strict;
use warnings;

use v5.10;

use Exporter qw(import);
use Test::Builder;

my $Test = Test::Builder->new();

our $VERSION = '0.10';
our @EXPORT = qw(sum_ok);

sub sum_ok {
    my( $got, $expected, $label ) = @_;
    $label //="The sum is $expected";

    if( $got eq $expected ) {
        $Test->ok( 1, $label );
    }
    else {
        $Test->diag("The sums do not match. Got $got, expected $expected");
        $Test->ok( 0, $label );
    }
}

1;
```

现在我们在一个测试程序中使用新模块：

```
use Test::More;
BEGIN { use_ok( 'Test::My::List::Util' ) }
BEGIN { use_ok( 'My::List::Util' ) }
```

```
ok( defined &sum, 'sum() is exported' );
ok( defined &sum_ok, 'sum_ok() is exported' );

sum_ok( sum( 1, 2, 3 ), 6, '1+2+3 is 6' );

done_testing();
```

第 21 章答案

练习 1

登录网站 <https://pause.perl.org/>，并且单击“Request PAUSE account”链接。填写信息，发送表单，然后等待。在最后部分中的等待，就是在第 1 章的练习中我们让你做该事情的原因。

练习 2

一旦我们有了 Animal 的发布版，我们就会测试它：

```
% ./Build disttest
```

如果所有测试都通过，我们应该构建发布版：

```
% ./Build dist
```

这就是该练习答案。在该练习之后，我们对 Animal 类将不再执行任何操作。

练习 3

我们从创建一个基于 PAUSE 名字的新的发布版开始：

```
% module-starter --module=Acme::GILLIGAN::Utils
```

我们切换到这个目录来开始工作：

```
% cd Acme-GILLIGAN-Utils
```

我们确认所有测试都通过。虽然还没有做过任何事情，但是我们想要从一个好的地方开始：

```
% ./Build test
t/00-load.t ..... ok
t/boilerplate.t ... ok
t/manifest.t ..... skipped: Author tests not required
t/pod-coverage.t .. ok
t/pod.t ..... ok
All tests successful.
Files=5, Tests=6, 0 wallclock secs ( ... )
Result: PASS
```

在新的发布版中，有一个 lib/Acme/GILLIGAN/Utils.pm 文件。我们需要给该文件添加 sum 子例程。怎样实现并不重要，但无论如何都要得到软件文档：

```
=head2 sum( LIST )
```

Numerically sums the argument list and returns the result.

```
=cut
```

```
sub sum {
    my $sum;
    foreach ( @_ ) { $sum += $_ }
    return $sum;
}
```

我们再一次测试，并且一切都应该通过。如果我们跳过了软件文档，`t/pod-coverage.t` 测试将会失败。

既然在模块中有一个新的子例程，我们就应该测试它。这是一个简单的测试，我们放入到 `t/sum.t` 测试文件中。我们测试到我们可以加载模块并且还定义了子例程。这将是十分烦人的工作——不断修改代码最后才发现是子例程名出错（尽管我们不愿意承认，这种事情发生的频率还是很高）。在这之后，测试数字 1~10 的和。这就是一个有着良好的输入以及期望的良好输出的测试。我们也在`@weird_list` 中测试了坏的输入，在该输入列表中没有任何数字。接下来会发生什么？

```
use Test::More tests => 4;

use_ok( 'Acme::GILLIGAN::Utils' );
ok( defined &Acme::GILLIGAN::Utils::sum, 'sum() is defined' );

my @good_list = 1 .. 10;
is( Acme::GILLIGAN::Utils::sum( @good_list ), 55,
    'The sum of 1 to 10 is 55' );

my @weird_list = qw( a b c 1 2 3 123abc );
is( Acme::GILLIGAN::Utils::sum( @weird_list ), 129,
    'The weird sum is 128' );
```

当我们运行这些测试时，我们因为这些怪异的元素而得到警告：

```
% ./Build test
t/00-load.t .... ok
t-boilerplate.t ... ok
t/manifest.t ..... skipped: Author tests not required
t/pod-coverage.t .. ok
t/pod.t ..... ok
t/sum.t ..... 1/2 Argument "a" isn't numeric ...
Argument "b" isn't numeric ...
Argument "c" isn't numeric ...
Argument "123abc" isn't numeric ...
t/sum.t ..... Dubious, test returned 255 (wstat 65280, 0xff00)
All 2 subtests passed
```

Test Summary Report

```
-----
t/sum.t      (Wstat: 65280 Tests: 4 Failed: 2)
Failed tests:  3-4
Non-zero exit status: 255
Files=6, Tests=10,  1 wallclock secs ( ... )
Result: FAIL
Failed 1/6 test programs. 2/10 subtests failed.
```

我们要如何处理这些情况？我们可以把这些警告留在那里，这样程序员就会知道他或她的代码中有错误。我们也可以忽略它们：

```
sub sum {
    no warnings 'numeric';
    my $sum;
    foreach (@_) { $sum += $_ }
    return $sum;
}
```

如果我们想要确保我们已经解决了所有的警告，我们可以使用 Test::NoWarnings（尽管我并不期望这出现在你的答案中）：

```
use Test::More tests => 5;
use Test::NoWarnings;

use_ok( 'Acme::GILLIGAN::Utils' );
ok( defined &Acme::GILLIGAN::Utils::sum, 'sum() is defined' );

my @good_list = 1 .. 10;
is( Acme::GILLIGAN::Utils::sum( @good_list ), 55,
    'The sum of 1 to 10 is 55' );

my @weird_list = qw( a b c 1 2 3 123abc );
is( Acme::GILLIGAN::Utils::sum( @weird_list ), 129,
    'The weird sum is 128' );
```

现在，当我们运行这些测试时，似乎没有什么错误：

```
% ./Build test
t/00-load.t ....... ok
t/boilerplate.t ... ok
t/manifest.t ..... skipped: Author tests not required
t/pod-coverage.t .. ok
t/pod.t ..... ok
t/sum.t ..... ok
All tests successful.
Files=6, Tests=11, 0 wallclock secs ( ... )
Result: PASS
```

一切正常。此时是时候把它上传到 PAUSE 了。我们更新清单，把其列入 t/sum.t 文件中：

```
% ./Build manifest
Added to MANIFEST: t/sum.t
```

我们测试了发布版，并且当其通过时，我们创建存档：

```
% ./Build disttest
% ./Build dist
```

现在准备上传。我们登录 PAUSE 账户，并且单击链接“Upload a file to CPAN”。在那里，我们按照指示并发布代码。不到一个小时，甚至更可能在更短的时间内，我们应该能够在网站 <https://www.metacpan.org/> 上发现我们新的发布版。

如果这是你第一个 CPAN 发布版，恭喜你！

练习 4

对于这个练习，我们将破坏我们的发布版。首先，我们更新 lib/Acme/GILLIGAN/Utils.pm 中的版本号，这样我们就可以上传和索引一个新的发布版存档：

```
our $VERSION = '0.02';
```

我们要通过添加一个由他人控制的名称空间来破坏我们的发布版。我们建议使用由 BDFOY（本书的一个作者，可以使用）控制的 Tie::Cycle。添加一个模块：

```
% module-starter --module=Tie::Cycle
```

现在我们有一个 lib/Tie/Cycle.pm 文件。尽管 module-starter 已经自动更新了 MANIFEST，但是我们可以再次确认：

```
% ./Build manifest
```

在那里，我们像之前做的一样创建了一个新的存档：

```
% ./Build disttest  
% ./Build dist
```

我们上传文件到 PAUSE。当 PAUSE 试图索引新存档时，它将会在 Tie::Cycle 上失败。我们的模块将仍然进入到 CPAN 中，但是它将不会被索引，而且 PAUSE 将发送给我们一封电子邮件来解释到底是哪里出错了。

练习 5

这个练习是另一种类型的失败，但是这一次出错的原因与 CPAN Tester 有关。如果 lib/Tie/Cycle.pm 文件仍然存在，那么我们需要删除它。

我们更改代码，把 sum 中的代码替换成乘法运算：

```
sub sum {  
    my $sum;  
    foreach ( @_ ) { $sum *= $_ }  
    return $sum;  
}
```

我们希望测试失败，而且我们应当确保它们确实失败：

```
% ./Build test
```

我们不会告诉你失败的输出是什么。我们建立一个新的归档文件并将其上传到 PAUSE。当我们的发布版进入 CPAN 时，CPAN Tester 志愿者将开始下载和测试它。在它失败以后，我们应该会收到一封关于错误的电子邮件。

练习 6

当我们查看在 MetaCPAN 中的发布页面时，我们应该看到 Tester 的结果。查看 CPANdeps，我们也应该看到一个表明我们有一个版本通过了一个版本没有通过的矩阵式报表。在此处没有太多关于此练习的答案。详细相关内容可以通过登录网站

<https://www.metacpan.org/>找到我们的 page. Explore 来获得。

你也可以在该页面中寻找你喜欢的其他模块。

练习 7

我们应该可以使用 CPAN 客户端来安装我们在 CPAN 的发布版：

```
% cpan -I Acme::GILLIGAN::Utils
```

如果我们按照这些例子中的程序，我们停止了一个失败的发布版。此时我们应该修复这个发布版，重新上传并稍后再次尝试。

在成功地安装了这个模块之后，我们已经完成了整个流程。再次恭喜你！

作者介绍

Randal L. Schwartz 是一位著名的 Perl 编程语言专家。除了编写 *Learning Perl* 以及 *Programming Perl* 的前两版之外，他还是 *UNIX Review*、*Web Techniques*、*Sys Admin* 以及 *Linux Magazine* 的 Perl 专栏作家。他至今已经贡献了十多本 Perl 图书以及 200 多篇杂志文章。Randal 还经营着一家 Perl 培训和咨询公司（Stonehenge 咨询服务公司），并且凭借其技术技能、善于互动以及超高的人气被公认为 Perl 的发言人。他还是一位相当有水准的 K 歌歌手。

brain d foy 自从 1998 年起就是 Stonehenge 咨询服务公司的一名讲师，自从他成为物理系的研究生起就是一名 Perl 用户，而且自从他拥有了电脑之后，就成了 Mac 的铁杆用户。他成立了第一个 Perl 用户组——New York Perl Mongers，还成立了非盈利的 Perl Mongers 公司，该公司帮助组建了全球 200 多个 Perl 用户组。他是 Perl 核心文档 perlfaq 部分的维护人，还维护 CPAN 的多个模块以及一些独立的脚本文件。他是 *The Perl Review*（一家致力于 Perl 的杂志）的出版人，并经常在各种会议（包括 Perl Conference、Perl University、MaccusEvansBioInformatics' 02 和 YAPC）上发表演讲。他写作的 Perl 作品可以在 *O'Reilly Network*、*The Perl Journal*、*Dr. Dobbs*、*The Perl Review*、use.perl.org 以及多个 Perl 新闻组上找到。

Tom Phoenix 自从 1982 年起投身于教育领域，在科学博物馆工作的 13 年多里，他经历了解剖、爆炸、与有趣的小动物共事以及在冒着火花的高压电下工作等事情，然后从 1996 年起加入了 Stonehenge 咨询服务公司，并教授 Perl 课程。从那时起，他去过了很多有趣的地方，没准你不久之后就会在 Perl Mongers 的会议上见到他。当他空闲时，他就会在 comp.lang.perl.misc 和 comp.lang.perl.moderated 新闻组上回答问题，而且为 Perl 的发展和使用贡献自己的力量。除了 Perl、Perl 破解以及相关主题的工作之外，Tom 还在业余密码学和讲世界语方面投入了很多时间。Tom 住在美国俄勒冈州的波特兰市。

封面图片说明

本书封面上的动物是羊驼。羊驼是南美骆驼科动物家族中的一员，它与大家更为熟悉的亚洲和非洲骆驼具有密切的关系。南美骆驼科动物还包括美洲驼、小羊驼和原驼。羊驼的体型（36 英寸高）要小于美洲驼，但是要比其他骆驼科动物大。世界上大约有 300 万只羊驼，其中 99% 都生活在秘鲁、玻利维亚和智利。

野生骆马大概是在 6000~7000 年前开始向家养羊驼进化。在公元前 500 年，人们开始专业化养殖羊驼，用于生产羊毛纤维。印加人将羊驼发展成两个不同的类型：华卡约（Huacaya）和更少见的苏利（Suri）。它们之间主要区别是生产的羊毛纤维不同。华卡约羊毛为卷曲或波浪状；苏利羊毛柔滑而有光泽，无卷曲。羊驼因其羊毛而备受珍视，它的羊毛软如羊绒，而且要比羊毛更轻、更结实。羊驼的羊毛要比生产纤维的其他动物的颜色更多（大约有 22 种基本颜色，而且有许多变化和混合）。

羊驼的寿命大约为 20 年，其妊娠期为 11.5 个月，每 14~15 个月生产一胎。羊驼是一种改良的反刍动物，吃的草要比其他动物少很多，但是可以高效地将其转换为能量。与真正的反刍动物不同，它们的胃有 3 个室而不是 4 个，因此可以在不适合其他家养动物生存的地区存活下来。羊驼很温顺，不会咬人或顶撞。即使它们咬人或顶撞人了，由于不使用门牙、角、蹄或爪，因此也不会造成伤害。