Mastering Perl

Creating Professional Programs with Perl

Randal L. Schwartz 作序推荐

精通Perl

brian d foy 著

韩殿飞 译

O'REILLY®

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

O'REILLY®

精通 Perl

Mastering Perl

brian d foy 著

韩殿飞 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书是 O'Reilly 出版社 Perl 系列教程的第 3 本图书，介绍了 Perl 独特的工作机理和编程思想，以及如何把前两本的所有知识综合到一起，让你能够随心所欲地使用 Perl。本书并不是一本小窍门的集合，而是着重介绍了 Perl 编程的思维方式。它能够帮助你解决在日常工作中遇到的诸如调试、维护、配置之类的各种问题。本书将带你一路揭示这些问题的答案，让你成为能够发现并解决各种问题的专家。

本书适合于所有想成为 Perl 大师的中高级用户。

版权贸易合同登记号 图字：01-2007-4643

图书在版编目（CIP）数据

精通 Perl /（美）福瓦（foy,b.d.）著；韩殿飞译. —北京：电子工业出版社，2009.1

书名原文：Mastering Perl

ISBN 978-7-121-07713-5

I. 精… II. ①福…②韩… III. PERL 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2008）第 174777 号

责任编辑：王继花

项目管理：梁 晶

封面设计：Karen Montgomery 张 健

印 刷：北京市天竺颖华印刷厂

装 订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：21.75 字数：529 千字

印 次：2009 年 1 月第 1 次印刷

定 价：68.00 元

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

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

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

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

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

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

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

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

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

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

译者序

翻译一本 300 页的英文技术书籍需要多长的时间？在开始翻译这本书之前，我的想法很简单：如果每天下班以后翻译 2 页，只要 150 天也就是 5 个月的时间。可是我的编辑晓菲不能等那么长的时间。我只好调整成每天翻译 3 页，总共 100 天。当我真正开始着手翻译本书的时候，才发现保持每天 3 页的进度并不是一件容易的事情。有的时候工作上会突然出现一些紧急的事情，你须要晚上加班处理。有的时候恰好碰上节假日，你很想和家人、朋友一起放松放松。这就意味着你须要在状态好、干扰少的时候多做一点，才能弥补加班和休息带来的进度损失。不过，除这些外部的因素以外，更让人“烦恼”的是书中的一些问题。

一个问题是 Perl 中各种术语的翻译。有的术语有多种中文译法，让我在如何取舍方面颇费思量。比如“hash”，有人译作哈希，也有人称为散列。一个取音，一个取意，倒还都好理解。又如“dereference”，网上的翻译居然有不下 10 种。遇到这种情况，只好放下进度，慢慢斟酌了。也有些术语几乎找不到中文译名，比如“shebang”、“sigil”。遇到这种情况只好在第一次出现的地方加上译者注，详细解释它的中文含义，之后一律使用英文名。还有一些术语虽然网上也有翻译，但是个人觉得翻译欠妥，还不如就用英文名，比如“typeglob”。

另一个问题是作者的行文风格。我可不是要攻击 brian d foy——这样大逆不道的事情我可不敢做。我的意思是 brian d foy 的行文风格和其他作者的大不相同。很多时候作者都是以第一人称出现——这在技术类书籍中可不常见。读这本书的时候，你会感觉到他仿佛就站在你的面前，手把手地给你讲解 Perl 的各种功能、绘声绘色地描述他自己的亲身经历。如果用书中介绍的单词计数程序统计一下英文原书中各个单词的频率，你也许会发现书中出现最多的单词不是 the 而是 I。有的时候你会发现作者翻来倒去地重复一句话。这个时候千万不要怀疑是译者或作者的电脑坏了。作为一个长期给人讲授 Perl 的程序员，brian d foy 非常清楚地知道要想让读者记住一个东西的最好方法就是重复。不过请放心，brian d foy 是不会像国内的某些广告那样重复羊羊羊牛牛牛狗狗狗的。

不过，所有的这些困难算不了什么，尤其是当我读到本书最后的“作者简介”时。作者的简介是这样开头的：“Brian d foy 从 1998 年以来一直是 Stonehenge Consulting Services 的一名培训师。”1998 年？嗯，确实是 1998 年。Brian d foy 能够十年如一日地坚持做一件事情，难道我就不能坚持几个月翻译完他的这本呕心沥血之作吗？

最后，作为一个琢磨了这本书几个月的译者，给有兴趣阅读此书的朋友们两点建议。

1. **从你感兴趣的地方开始。**和该系列的前两本书不同，本书各章节之间的依赖关系不强。除了第 9 章和第 10 章需要第 8 章的基础之外，你可以从感兴趣的任何一章开始阅读。

2. **多动手多动脑**。正如作者在第 1 章中所说的："这本书并不会把你变成 Perl 大师。要成为 Perl 大师，你必须写大量的 Perl 程序、尝试各种新的东西、犯很多的错误。"像读小说一样漫不经心地阅读本书对你的帮助不会太大。你须要试试书中的例子，多多琢磨作者那些风趣幽默而又意味深长的话，并且努力把学到的东西用到你的工作中。

韩殿飞
2008 年 10 月于北京

目录

Table of Contents

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

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

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

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

序言

Foreword

作为 Stonehenge 的专业培训人员，遇到的一个问题是确保我们写的培训材料能够在多次报告中重复使用。开发一套课程教材的昂贵费用迫使我们必须要招到 200 到 400 人，而且须要他们有差不多的基础、希望达到差不多的水平，并且支付得起课程的费用。

对于我们的旗舰产品——《Learning Perl》的课程而言，内容选择是很容易的：选择几乎所有人须要知道的、写单文件脚本程序所需要的、适合 Perl 大量应用的所有内容，以及我们可以在讲授 Perl 的第一阶段教授的内容。

为《Intermediate Perl》选题时，我们面临的挑战就要稍微大一点，因为“明显的”选择远没有那么明显。我们的结论是：在学习 Perl 的第二阶段，人们更愿意学习构造复杂数据结构和对象的知识，以及如何组织团队工作（模块、测试和发布）。又一次撞了个正着——我们的课程和教材都很受欢迎。

在更新《Learning Perl》和《Intermediate Perl》之后不久，brian d foy 意识到除了这两个教程之外关于 Perl 还有很多的东西可说，但是这些不一定适合所有的人。

在《Mastering Perl》中，brian 抓住了很多有趣的话题并围绕它们列举了大量例子。各章之间几乎完全独立。你可能发现并不是所有的这些都和自己的工作有关，但是当你有空闲和动力的时候，这本书完全值得重新捧起细细品味——这在课堂上是不可能做到的。虽然你不能享受到我们一对一细致地讲解和交流，但 brian 做了出色的工作使得这些内容深入浅出、自成一体。

很奇怪的是，我用 Perl 编程几乎有 20 年了，还是从这本书里学到了一些东西，所以说 brian 确实写得很棒。我希望你和我一样喜欢读这本书。

——Randal L.Schwartz

前言
Preface

《Mastering Perl》是 Perl 系列教程的第三本书。第一本书是《Learning Perl》，介绍 Perl 的基础语法。第二本书是《Intermediate Perl》，介绍如何编写可重用的 Perl 程序。这本书会告诉你如何把所有的这些组合到一起，让 Perl 乖乖地听你的吩咐。这不是一本小窍门的合集，而是介绍 Perl 编程的思维方式。它能够帮助你解决程序员在日常工作中遇到的诸如调试、维护、配置之类的各种问题。这本书能够使你得到答案，即使不行，也能让你知道如何找到答案或发现问题。

本书的结构
Structure of This Book

第 1 章　引言：成为大师

介绍本书的适用范围和目的。

第 2 章　高级正则表达式

介绍更多的正则表达式功能，包括全局匹配、前后查询匹配、可读化的正则表达式，以及如何调试正则表达式。

第 3 章　安全编程技术

运用本章的技术（包括污点检测和 gotchas）避免常见的编程问题。

第 4 章　调试 Perl 程序

关于 Perl 调试器的一些内容：编写你自己的调试器，使用别人的调试器。

第 5 章　剖析 Perl 程序

在改进 Perl 程序之前，找出须要集中精力来改进的地方。

第 6 章　Perl 基准测试

分析哪个版本在时间、空间和其他指标上做得更好，介绍如何解读数据的真实含义。

第 7 章　清理 Perl 程序

用 `Perl::Tidy` 或 `Perl::Critic` 把别人（或者是你）写的 Perl 程序变得更具表达力和可读性。

第 8 章　符号表和 typeglob

了解 Perl 是如何维护软件包变量的，并使用该机制实现更强大的功能。

第 9 章　动态子程序

动态定义子程序，克服普通的过程式编程带来的劣势。构造循环遍历子程序的列表（而不是数据的列表），使你的程序更加高效和易于维护。

第 10 章　修改模块和临时调整模块

在不修改原始代码的情况下解决代码存在的问题，以便能够随时恢复到原始状态。

第 11 章　配置 Perl 程序

让用户在不触及代码的情况下自行对程序进行配置。

第 12 章　检查和汇报错误

了解 Perl 汇报错误的机制，侦测 Perl 没有报告的错误，学习如何向用户汇报错误。

第 13 章　日志

使用极度灵活和强大的 `Log4perl` 日志模块，让 Perl 程序能够和你对话。

第 14 章　数据持久化

把数据保存起来，以便之后再次运行该程序时或运行其他程序时可以访问这些数据。介绍如何通过网络发送数据。

第 15 章　使用 Pod

把纯文本格式的文档转换成你喜欢的格式，并测试文档的合法性和覆盖率。

第 16 章　位操作

利用位操作和位向量来高效地存储大量的数据

第 17 章　奇妙的绑定变量

实现自己的基本数据类型并进行精巧的操作，避免干扰用户。

第 18 章　以模块的形式编写程序

以 Perl 模块的形式编写应用程序，以便享受 Perl 的模块发布工具、安装工具和测试工具的便利。

排版约定
Conventions Used in This Book

本书采用如下的约定：

等宽字体（Courier New）

用于表示方法名、模块名、环境变量、代码片段和其他直接量（literal）。

斜体（*Italics*）

用于表示强调、Perl 文档、文件名及定义的新名词。

代码示例
Using Code Examples

本书的目的是帮助你完成工作。一般来说，可以在你的程序和文档中使用本书中的代码。你不须要联系 O'Reilly 获得许可，除非你用了代码的大部分。举例来说：写一个程序只用到本书中的几段代码不需要许可；通过 CD-ROM 销售或发行 O'Reilly 书中的代码需要许可；通过引用本书和示例程序来回答问题不需要许可；把书中大部分的代码放入你的产品手册中需要许可。

我们非常感谢但并不要求引用致谢。致谢通常应该包括：标题、作者、出版社和 ISBN 号。比如："*Mastering Perl* by brian d foy. Copyright 2007 O'Reilly Media, Inc., 978-0-596- 52724-2."

如果你觉得使用代码的方式超出了上述许可范围，欢迎通过 *permissions@oreilly.com* 联系我们。

评论与问题
Comments and Questions

如果你想就本书发表评论或有任何疑问，敬请联系出版社：

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

奥莱利技术咨询（北京）有限公司

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

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室
邮政编码：430074
电话：（027）87690813　　传真：（027）87690595
网页：*http://bv.csdn.net*
读者服务信箱：
reader@broadview.com.cn（读者信箱）
bvtougao@gmail.com　　（投稿信箱）

本书的网站提供了勘误、示例和其他信息。网站的地址是：
http://www.oreilly.com/catalog/9780596527242（原书）
http://www.oreilly.com.cn/book.php?bn=978-7-121-07713-5（中文版）

此外，在本书的写作过程中，作者为本书设立了一个网站，你可以在上面找到额外的信息，包括相关资源和内容更新：
http://www.pair.com/comdog/mastering_perl

如果你想就本书发表评论或提问技术问题，请发送 E-mail 至：
bookquestions@oreilly.com

致谢

Acknowledgments

在本书的写作过程中，很多人给予了我帮助。《Mastering Perl》邮件列表的很多读者不断地给书稿提出意见和程序补丁，其中的很多我直接用在了书稿中。他们包括：Andy Armstrong、David H. Adler、Renéè Bäcker、Anthony R. J. Ball、Daniel Bosold、Alessio Bragadini、Philippe Bruhat、Katharine Farah、Shlomi Fish、David Golden、Bob Goolsby、Ask Bjørn Hansen、Jarkko Hietaniemi、Joseph Hourcle、Adrian Howard、Offer Kaye、Stefan Lidman、Eric Maki、Josh McAdams、Florian Merges、Jason Messmer、Thomas Nagel、Xavier Noria、Les Peters、Bill Riker、Yitzchak Scott-Thoennes、Ian Sealy、Sagar R. Shah、Alberto Simoes、Derek B. Smith、Kurt Starsinic、Adam Turoff、David Westbrook 和 Evan Zacks。他们持续不断的严格审查让我确信自己正走在正确的道路上。

Tim Bunce 给 profiling 一章提出了无私的建议，包括 `DBI::Profile`。Jeffrey Thalhammer 则提供了他的 `Perl::Critic` 模块最新的开发进展。

Perrin Harkins、Rob Kinyon 和 Randal Schwartz 在最后给手稿做了一个彻底的分析。他们的意见总是那么一针见血，我很高兴选择了他们作为本书的技术评论员。

Allison Randal 针对 Perl 提出了宝贵的意见并在编辑方面给予了大量指导，尽管她好像被我不停地打扰弄得精疲力竭。在年末的时候，Andy Oram 接替了编辑工作，帮助我把手稿变成了一本书。所有 O'Reilly Media 的员工，编辑、印刷、市场、销售等，都非常友善地给予了帮助，和他们合作很愉快。完成一本书需要的远不止一个作者，所以以后无论见到 O'Reilly 的哪位员工都希望你能向他表示感谢。

我在 Stonehenge Consulting 的合作伙伴 Randal Schwartz 曾经告诫我写一本书须要做大量的工作，他还是慷慨地允许我休息近一年的时间以完成此书。我是从阅读他的《Learning Perl》开始学习 Perl 的，现在很高兴能够为这个系列又添加一本。Randal 多次告诉我："你在星巴克（写书稿）会得到更多的报酬，也能得到健康保险。"一般作者写书是为了和世界分享他的想法，我们写书是为了让别人成为更优秀的程序员。

最后，我要感谢 Perl 社区。在过去的 10 年里，Perl 社区对我难以置信地友好，给予我大量的支持。很多伟大的程序员和经理帮助我成为了一名更优秀的程序员，我希望这本书同样也能帮助那些刚加入 Perl 社区的人们。

第1章

引言：成为大师

Introduction: Becoming a Master

这本书并不会把你变成 Perl 大师。要想成为 Perl 大师须要靠自我努力，你必须写大量的 Perl 程序、尝试各种新的东西，还要犯很多的错误。在这本书中我只是在帮你走上正确的道路，而作为 Perl 大师，你应当能够解答自己和他人的问题。

在行会的全盛时期，工匠们在学习手艺的过程中，无论是实质上还是象征上都遵循着某条道路。他们从学徒开始，先做一些繁琐的工作直至他们掌握了足够多的技巧成为更值得信任的熟练工（journeyman）。熟练工拥有更大的责任，但是依然在一个公认的大师手下工作。在掌握了足够的技艺之后，他须要完成一件作品（master work）来证明他的能力。如果其他的大师认为这件作品达到了足够的水平，他就成为了一个公认的大师。

熟练工和大师也会旅行到（虽然有些人不同意这是 journeyman 一词的来源）其他大师那里，学习新的技术和技巧。每个大师都知道一些别人不知道的东西，或许是故意隐藏的，或许是从不同的途径得到的。熟练工的一部分教育来自多位大师。

与其他大师和熟练工的交流也是大师的教育的延续。他可以从其他更有经验的大师那里学习，也会在教熟练工的过程中（间接地）学到熟练工的老师的技巧。

学徒遵循的道路会影响他所能学到的东西。有机会跟从多位大师的学徒会接触到更多的视角和教学的方法，从而把这些变成自己的风格。来自某位大师的奇特的方法会被其他大师的方法所中和，从而使学徒获得一个平衡的视角。此外，不管是要成为木匠还是泥瓦匠，不同的大师把这些技巧用于不同的目的，从而给学徒提供了了解不同的应用和处事方式的机会。

不幸的是，我们没有行会。大部分 Perl 程序员都是靠他们自己来学习 Perl（作为 Perl 教员，我很遗憾地这样说）、编写程序，从来没有机会得到指导。这也是我入门的方式。我当时购买了《Learning Perl》的第 1 版，完全靠自己看完了全书。当时我是周围的人中唯一了解 Perl 的人。我已经开始反复研究它了，而大部分人只是简单地使用别人做好的东西。不久之后，我发现了讨论列表 comp.lang.perl.misc 并开始回答所有我能够回答的问题。这就像是布置给自己的家庭作业一样。我的技巧得到了提高。我得到了几乎即时的反馈，有好的也有坏的，这些又促使我学到了更多 Perl 的知识。后来，我的工作变成了整天用 Perl 写程序，虽然我是那个公司中唯一一个那样的人。我一直坚持在 comp.lang.perl.misc 上回答问题。

最后，我引起了 Randal Schwartz 的注意，并在他的翅膀的庇护下开始了我的 Perl 学徒生涯。他邀请我加入 Stonehenge Consulting Services 并成为一名 Perl 教员。从此，我开始了教授 Perl 的职业生涯。讲课，意味着找出你所知道的东西并解释给别人，这是学习的最佳方法。之后不久，我开始写一些和 Perl 教学有关的文章。（大部分文章都语法正确，有一个编辑帮忙纠正错误。）

这给本书带来了一个问题。本书是《Learning Perl》、《Intermediate Perl》（两书我都有所参与）系列的第 3 本，所有的这些书都是 300 多页，所以本书也有这个限制。怎么才能把我多年的经验浓缩到这么小的篇幅上呢？

我做不到。我会告诉你应该知道什么，你还须要从其他地方学习什么。就像古代和大师学习一样，你不能只听一个人的。你须要找到别的大师。这也是 Perl 很棒的地方：你可以用很多不同的方式完成一件事情。很多大师写了非常不错的书，各个出版社的都有，所以我就不再重复了。

成为大师的含义
What It Means to Be a Master

本书采取了一个与《Learning Perl》和《Intermediate Perl》不同的方式。前两个都是指导教程，主要介绍 Perl 语言的细节，很少涉及编程实践。本书则把更多的任务留给了你——我们的读者。

你已经学习 Perl 多年，你正通过自己找到问题的答案来提高自己的能力。这比简单地问别人须要付出更多的努力，不过你正是靠这个来积累经验，同时避免过度地打扰同事。

虽然我没有像 Sriram Srinivasan（O'Reilly）的《Advance Perl Programming》第 1 版和 Jeffrey Friedl（O'Reilly）的《Mastering Regular Expressions》（译注 1）那样涉及其他语言，你也应

译注 1：《Mastering Regular Expressions》已由电子工业出版社博文视点资讯有限公司引进出版，中文书名为《精通正则表达式，第 3 版》，目前已经是第 3 次印刷。

该学习一些其他的语言。这有助于丰富你的 Perl 知识、得到更多的视角，从而使你更好地欣赏 Perl，更好地理解它的局限之处。

作为大师，你会遇到 Perl 的局限之处。如果不能列出 5 个讨厌 Perl 的理由和事实，就说明你对 Perl 的理解还不够。这并不是 Perl 的问题。所有的语言都会有缺陷。大师知道这些缺陷但是依然使用 Perl，因为它的优点超过了不足。你是一个大师，因为你知道问题的两个方面、能做出合理的选择、能给别人清楚的解释。

所有的这些意味着成为大师须要工作、阅读，以及和别人的交流。你做得越多，就学得越多，没有捷径。可能你能很快地学完了语法，像任何语言一样，那只是经验中最小的一部分。既然你知道了大部分 Perl 的知识，你就应该花时间读一些关于编程实践的“元”编程的书，而不是满足于语法。这些书也许用的不是 Perl，不过我已经说过，你须要学点别的语言。作为大师，你须要一直学习。

成为大师意味着要理解大量的东西（超过你所需要的）、独自完成很多工作、从别人的经验中学到尽可能多的东西。这不仅仅包括你写的代码，也包括如何处理来自许多其他作者的代码。

也许这听起来很难，不过这是成为大师的必经之路。这些是值得的，不要放弃！祝你好运！

本书适合的读者
Who Should Read This Book

《Intermediate Perl》介绍了引用、对象、模块的基础知识。作为《Intermediate Perl》的后继，我假设读者已经对这些概念非常熟悉。需要的时候，我会引用《Intermediate Perl》中的内容帮助你复习。

如果刚从其他的语言转过来还没有用过 Perl，或者用得很少，你可能须要浏览一下《Learning Perl》和《Intermediate Perl》以了解 Perl 的大概。你可能会对一些需要经验和实践的习惯用法感到陌生。我不是说不要你买这本书（嘿，我需要钱还抵押贷款），只是想告诉你，可能你不能完全领会书中的内容。

如何阅读本书
How to Read This Book

我写的不是《Yet More Perl Features》的第 3 卷。我希望教你如何靠自己来学习 Perl。我把你带到成为大师的路上，作为学徒，须要独自完成很多工作。有时，这意味着我会告诉你去 Perl 手册的某个地方寻找答案（这也意味着我可以节约空间以讨论更多的话题）。

你应该已经知道的内容
What Should You Know Already?

我假设读者已经知道我们在《Learning Perl》和《Intermediate Perl》中讲到的内容。这里的我们指的是 Stonehenge Consulting Services 的工作人员，Perl 畅销书的合作者 Randal Schwartz、Tom Phoenix 和我。

最重要的是，你应该熟悉这些话题，以及它们涵盖的相关内容：

- 使用 Perl 模块
- 编写 Perl 模块
- 引用变量、子程序和文件句柄
- 基本正则表达式的语法和原理
- 面向对象

如果涉及以上两本书以外的内容，我会深入地介绍。有些内容如果非常重要的话，即使前两本书中已经涉及了我也会再介绍一遍。

本书涵盖的内容
What I Cover

在学习了《Learning Perl》中 Perl 的基本语法和《Intermediate Perl》中模块和团队工作的基础知识之后，读者须要学习的是 Perl 编程的常见用法，以及把学到的技巧融汇起来写出可重用的、健壮的、可扩展的应用。

我会更加详细地论述前两本书中的一些话题。正如《Learning Perl》中所说的那样，我们有时会说一些小谎话来简化细节，避免陷入泥潭。而现在则到了对某些问题深入研究的时候。

不过请不要把我对某个问题的论述当成教条。世界上有上百万的 Perl 程序员，每个人都有他自己做事情的方法。成为 Perl 大师须要读大量的 Perl 程序，即使你不用自己写。当我认为你不必做某件事的时候，我会尽力告诉你，不过那只是个人意见。在努力成为一名优秀的程序员的过程中，你须要知道很多东西，尽管不一定用得上。有些时候我会介绍一些不希望你用的东西，因为你会从别人的代码中看到。哎，这不是一个完美的世界。

添加和修改代码的功能并不是编程的全部。有些时候须要把代码拆开来分析。有些时候要去掉不需要的代码。编程时间不限于编写新的程序，也包含管理代码和为代码而争论。书中的有些技巧是用于分析问题的，而不是用于开发代码的。

本书没有涵盖的内容

What I Don't Cover

我和编辑讨论过本书的涵盖范围，我们决定不再重复其他的书中已经论述得很充分的内容。你也须要从其他的大师那里学习。我也不希望本书过多地占据你的书架。忽略这些问题既避免了重复工作，又为其他话题留下了空间。不管怎么说，你确实应该读读其他的书。

不过，这不是说你可以忽略这些内容。我会在合适的地方向你推荐适合的书。附录 A 列出了我认为应该放在你的书架上的书。这些来自其他 Perl 大师的书每一本都有值得你学习的内容。每章的末尾也会列出相关资源。真正的大师永远不会停止学习。

既然你读到了这里，我干脆列出本书不会涉及的话题：Perl 的内部原理、Perl 的内嵌、多线程、最佳实践、面向对象编程、源文件过滤器和海豚。这是一本与海豚无关的书。

第2章

高级正则表达式

Advanced Regular Expressions

正则表达式，简称正则式，是 Perl 文本处理的核心，毫无疑问也是使 Perl 如此流行的众多特性中的一个。所有的 Perl 程序员都经历过一个尝试把所有的东西都用正则式表示的阶段，有的甚至觉得还不够有挑战性，把所有的都用一个正则式来表示。Perl 的正则式的特性远远超过了我能够和希望在本章中介绍的范围。所以这里我只涵盖一些高级特性。我认为这些是最有用的，希望其他的 Perl 程序员能够掌握而无须查阅 *perlre*——正则表达式手册。

引用正则表达式

References to Regular Expressions

我们不必在编程的时候就知道所有的模式。Perl 允许在正则式中插入变量。变量的值可以通过硬编码指定、从用户输入得到或从其他的途径得到。这里是一个和 `grep` 有同样功能的 Perl 小程序。它从命令行得到第一个参数作为 `while` 循环中的正则表达式。这并没有什么特别的，我们在《Learning Perl》中曾演示过。我们把`$regex`中的字符串用作匹配的模式，Perl 在解析到匹配操作符时会对它进行编译（注 1）：

```
#!/usr/bin/perl
# perl-grep.pl

my $regex = "shift @ARGV";

print "Regex is /$regex/\n";

while (<>) {

    print if m/$regex/;
}
```

注 1：在 Perl 的 5.6 版中，如果字符串没有变化，Perl 不会在每次循环中重新编译正则表达式。在 Perl 的 5.6 版之前，需要用/o 选项才不会重新编译。在 5.6 版中，如果不希望重新编译模式，即使字符串发生了变化，也可以用/o 选项。

我们可以通过命令行用这个程序查找文件中包含的模式。此处我们在当前目录的所有 Perl 程序中搜索 new：

```
% perl-grep.pl new *.pl
Regex is [new]
my $regexp = Regexp::English->new
my $graph = GraphViz::Regex->new($regex);
                [ qr/\G(\n)/,                "newline"    ],
                                               { ( $1, "newline char" ) }
print YAPE::Regex::Explain->new( $ARGV[0] )->explain;
```

如果我们给它传了一个非法的正则式会发生什么呢？让我们尝试一个有开括号而没有对应的闭括号的模式：

```
$ ./perl-grep.pl "(perl" *.pl
Regex is [(perl]
Unmatched ( in regex; marked by <-- HERE in m/( <-- HERE perl/
        at ./perl-grep.pl line 10, <> line 1.
```

当 Perl 解析到匹配运算符中的插入的正则式时，Perl 会编译正则式并立刻报告错误退出。要想捕获这个错误，就须要在试图使用正则式之前编译它。

引用运算符 qr//把正则式保存在一个标量中(作为引用运算符,它的说明文档在 *perlop* 中)。qr//运算符会编译完模式，这样在匹配运算符中插入$regex 的时候就已经可以使用了。我们用 eval 包住 qr//来捕获错误，尽管最后还会用 die 来退出：

```
#!/usr/bin/perl
# perl-grep2.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Check your pattern! $@" if $@;

while( <> )
        {
        print if m/$regex/;
        }
```

$regex 中的正则式有匹配运算符的所有功能，包括后向引用和记忆变量。下面的这个模式用来查找 3 个字符的序列，其中第 1 个和第 3 个相同，所有的字符不能是空格。输入是从 perldoc -t 得到的 *perl* 手册的纯文本版本：

```
% perldoc -t  perl | perl-grep2.pl "\b(\S)\S\1\b"
    perl583delta        Perl changes in version 5.8.3
    perl582delta        Perl changes in version 5.8.2
    perl581delta        Perl changes in version 5.8.1
    perl58delta         Perl changes in version 5.8.0
    perl573delta        Perl changes in version 5.7.3
    perl572delta        Perl changes in version 5.7.2
    perl571delta        Perl changes in version 5.7.1
    perl570delta        Perl changes in version 5.7.0
```

```
    perl561delta       Perl changes in version 5.6.1
http://www.perl.com/      the Perl Home Page
http://www.cpan.org/      the Comprehensive Perl Archive
http://www.perl.org/      Perl Mongers (Perl user groups)
```

至少对我来说，很难发现哪些能匹配上，所以我改了改我们的 grep 程序。变量$&保存了匹配的字符串部分：

```
#!/usr/bin/perl
# perl-grep3.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Check your pattern! $@" if $@;

while( <> )
        {
        print "$_\t\tmatched >>>$&<<<\n" if m/$regex/;
        }
```

原来这个模式匹配到了点、字符、点，比如.8.：

```
% perldoc -t perl | perl-grep3.pl  "\b(\S)\S\1\b"
                perl587delta        Perl changes in version 5.8.7
                                 matched >>>.8.<<<
                perl586delta        Perl changes in version 5.8.6
                                 matched >>>.8.<<<
                perl585delta        Perl changes in version 5.8.5
                                 matched >>>.8.<<<
```

一件有趣的事是看看每段记忆区（变量$1、$2 等）都匹配到了什么。我们可以打印出它们的内容，不管是否有捕获分组。不过须要打印多少份呢？Perl 已经知道答案了，因为它用特殊数组@-和@+记录了所有的信息。@-和@+分别记录了每个匹配上的字符串的开始和结束位置。对于$_中的字符串，记忆组的数目等于@_和@+中最后的下标值（两者相等）。两个数组的第一个元素对应于匹配上的字符串（即$&），下一个元素的下标是 1，对应于$1，依次类推。$1 的值和下面这个 substr 调用的结果相同：

```
my $one = substr(
        $_,             # string
        $-[1],          # start position for $1
        $+[1] - $-[1]   # length of $1 (not end position!)
        );
```

遍历数组@-的所有下标就可以打印出这些变量：

```
#!/usr/bin/perl
# perl-grep4.pl

my $pattern = shift @ARGV;
```

```
my $regex = eval { qr/$pattern/ };
die "Check your pattern! $@" if $@;

while( <> )
        {
        if( m/$regex/ )
                {

                 print "$_";

                 print "\t\t\$&: ",
                        substr( $_, $-[$i], $+[$i] - $-[$i] ),
                        "\n";

                foreach my $i ( 1 .. $#- )
                        {
                        print "\t\t\$$i: ",
                                substr( $_, $-[$i], $+[$i] - $-[$i] ),
                                "\n";
                         }
                }
        }
```

于是我们可以看到匹配的字符串和子匹配：

```
% perldoc -t perl | perl-grep4.pl  "\b(\S)\S\1\b"
                perl587delta        Perl changes in version 5.8.7
                                    $&: .8.
                                    $1: .
```

即使模式中加入了更多的子匹配，不用改变任何代码就可以看到添加的匹配：

```
% perldoc -t perl | perl-grep4.pl  "\b(\S)(\S)\1\b"
                perl587delta        Perl changes in version 5.8.7
                                    $&: .8.
                                    $1: .
                                    $2: 8
```

(?imsx-imsx:PATTERN)

如果我们想在 grep 程序中做一些更复杂的事情，比如设定匹配关键字大小写无关，该怎么做呢？用我们的程序查找“Perl”或“perl”有很多种方法，都不须要做太多的额外工作：

```
% perl-grep.pl "[pP]erl"
% perl-grep.pl "(p|P)erl"
```

如果要让整个模式大小写无关，就须要做更多的工作——我可不喜欢那样。对于匹配操作符，我可以简单地在末尾加上/i 选项：

```
print if m/$regex/i;
```

对于 qr//操作符，也可以这样做。不过这样所有的模式都是大小写无关的了：

```
my $regex = qr/$pattern/i;
```

为了避免这个问题，可以把选项放在模式内。特殊序列(?imsx)可以用来打开指定的功能。如果需要大小写无关，可以在模式中使用(?i)。该选项对模式中(?i)之后的部分（或闭括弧之前的部分）有效：

```
% perl-grep.pl "(?i)perl"
```

一般而言，可以只对模式的部分内容使用选项。只须要把感兴趣的模式部分放在括号内，如表 2-1 所示。

表 2-1：(?options:PATTERN)中支持的选项

内联选项	描述
(?i:PATTERN)	大小写无关
(?m:PATTERN)	使用多行匹配模式
(?s:PATTERN)	用.匹配新行
(?x:PATTERN)	打开解释模式

我们也可以同时使用多个选项：

```
(?si:PATTERN)          让 . 匹配新行且大小写无关
```

在选项前加上-号，就能关掉该功能：

```
(?-s:PATTERN)          不让 . 匹配新行
```

从命令行得到模式的时候这个功能尤其有用。事实上，用 qr//操作符创建正则式的时候，我们就已经在使用这个功能了。下面我们修改一下程序，在用 qr//创建正则式之后和使用之前打印出正则式：

```
#!/usr/bin/perl
# perl-grep3.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Check your pattern! $@" if $@;

print "Regex ---> $regex\n";

while( <> )
        {
        print if m/$regex/;
        }
```

打印出正则表达式后，可以看到默认状态下所有的选项都被关掉了。这个作为字符串的正则式用(?-OPTIONS::PATTERN)关掉了所有的选项：

```
% perl-grep3.pl "perl"
Regex ---> (?-xism:perl)
```

我们可以打开大小写无关，虽然结果看起来会有点奇怪——好像关掉 i 是为了能够打开它：

```
% perl-grep3.pl "(?i)perl"
Regex ---> (?-xism:(?i)perl)
```

Perl 的正则式有很多类似的控制序列以括号开始，本章中会涉及一些。每个以开括号开始的序列都是用一些字符来指示它们的功能的。文档 *perlre* 中有完整的列表。

作为参数的引用

References As Arguments

由于引用是标量，所以编译后的正则表达式也可以像其他标量那样使用：保存在数组或哈希表中，作为参数传递给子过程。举一个例子：Test::More 模块 like 函数的第二个参数就是一个正则表达式。like 函数用来测试一个字符串是否匹配某个模式，并在无法匹配时给出详细的输出：

```
use Test::More 'no_plan';

my $string = "Just another Perl programmer,";
like( $string, qr/(\S+) hacker/, "Some sort of hacker!" );
```

由于$string 用了 programmer 而不是 hacker，匹配没有成功。输出告诉我们字符串的值、期望的结果和匹配用的模式：

```
not ok 1 - Some sort of hacker!
1..1
#   Failed test 'Some sort of hacker!'
#                   'Just another Perl programmer,'
#     doesn't match '(?-xism:(\S+) hacker)'
# Looks like you failed 1 test of 1.
```

函数 like 不用做任何特别的事情就可以使用作为参数传进来的正则表达式，虽然它在使用之前还是检查了引用的类型（注 2）：

```
if( ref $regex eq 'Regexp' ) { ... }
```

$regex 是一个引用（类型是 Rexexp），所以我们可以像普通的引用一样使用它。比如用 isa 检查类型，或者用 ref 获得它的类型：

```
print "I have a regex!\n" if $regex->isa( 'Regexp' );
print "Reference type is ", ref( $regex ), "\n";
```

注 2：其实这发生在 Test::Builder 模块的 maybe_regex 方法中。

非捕获分组，(?:PATTERN)

Noncapturing Grouping, (?:PATTERN)

正则式中的括号不一定会触发记忆。我们可以用特殊序列(?:PATTERN)来对字符串进行分组。这样，捕获分组中就不会有不需要的数据。

有时我们须要匹配 and 和 or 两边的人名。在@array 中有些这样的人名对。因为组合关系可能会变化，所以正则式中要用到选择操作 and|or。于是优先级就成了问题。选择操作的优先级低于顺序操作，所以须要用括号包住选择操作才行，(\S+) (and|or) (\S+)：

```
#!/usr/bin/perl
my @strings = (
          "Fred and Barney",
          "Gilligan or Skipper",
          "Fred and Ginger",
          );

foreach my $string ( @strings )
        {
        # $string =~ m/(\S+) and|or (\S+)/; # doesn't work
        $string =~ m/(\S+) (and|or) (\S+)/;

        print "\$1: $1\n\$2: $2\n\$3: $3\n";
        print "-" x 10, "\n";
        }
```

由于把选择操作括在了一起，输出结果中混入了不需要的东西：括号中的字符串部分出现在了记忆变量$2 中（表 2-2）。这是人为的意外。

表 2-2：意料之外的记忆变量

没有括住 and|or	括住 and|or
$1: Fred	$1: Fred
$2:	$2: and
$3:	$3: Barney
----------	----------
$1:	$1: Gilligan
$2: Skipper	$2: or
$3:	$3: Skipper
----------	----------
$1: Fred	$1: Fred
$2:	$2: and
$3:	$3: Ginger
----------	----------

用括号解决了优先级问题，可是现在我们有了多余的记忆变量。我们把匹配后的结果放到列表中后，所有的记忆变量——包括组合部分，都出现在了@names 中：

```
# extra element!
my @names = ( $string =~ m/(\S+) (and|or) (\S+)/ );
```

我们只是想简单地把东西放在一起而不触发记忆。在开括号后加入`?:`，就可以把括号变成无记忆的。之前的`(and|or)`变成了`(?:and|or)`。这样就不会触发记忆变量，也不会改变记忆变量的总数了。这种修饰符可以像括号一样使用。现在`@names`中没有额外的元素了：

```
# just the names now
my @names = ( $string =~ m/(\S+) (?:and|or) (\S+)/ );
```

易读的正则式，/x 和(?#...)

Readable Regex, /x and (?#...)

正则表达式有一个名副其实的坏名声——非常难读。正则式有它自己的简练的语言。这种语言用尽可能少的字符表达几乎无限的可能，而且这还只是大多数人天天用的部分。

幸运的是，Perl 提供了一种让正则式更加易读的方法。利用一些格式上的技巧，人们就能弄明白我想匹配的是什么。即使在写完程序的数周之后，我也还能弄明白。在《Learning Perl》中我们涉及过这个话题。这个想法非常之好，这里我们再深入讨论一下。Damian Conway 著的《Perl Best Practices》（O'Reilly）中也有所介绍。

把`/x`选项加到匹配或替换操作符中，Perl 就会忽略模式中的空格。这样就可以拆开模式的各个部分把模式变得容易辨认。Gisle Aas 的`HTTP::Date`模块使用了几种不同的正则表达式来解析日期。下面是其中的一个正则表达式。我把它改成一行并加入换行符以适合页面大小：

```
/^(\d\d?)(?:\s+|[-\/])(\w+)(?:\s+|[-\/])↵
(\d+)(?:(?:\s+|:)(\d\d?):(\d\d)(?::(\d\d))↵
?)?\s*([-+]?\d{2,4}|(?![APap][Mm]\b)[A-Za-z]+)?\s*(?:\(\w+\))?\s*$/
```

快速问答：你能说出它解析的是哪种日期格式吗？我不行。幸运的是，Gisle 用`/x`选项来帮助分隔正则式的各个部分并用注释解释了各部分的含义。有了`/x`，Perl 会忽略空格和正则式中 Perl 风格的注释。这里是 Gisle 实际的代码，比上面的容易理解多了：

```
/^
 (\d\d?)                # day
    (?:\s+|[-\/])
 (\w+)                  # month
    (?:\s+|[-\/])
 (\d+)                  # year
 (?:
      (?:\s+|:)         # separator before clock
```

```
    (\d\d?):(\d\d)    # hour:min
      (?::(\d\d))?    # optional seconds
    )?                # optional clock
            \s*
    ([-+]?\d{2,4}|(?![APap][Mm]\b)[A-Za-z]+)? # timezone
            \s*
    (?:\(\w+\))?      # ASCII representation of timezone in parens.
            \s*$
    /x
```

有了/x之后，要匹配空格就须要显式地指定了。可以用\s匹配所有的空白，或者用\f\r\n\t中的任意一个，或者用它们的十进制或十六进制序列，比如\040或\x20来表示空白（注3）。类似的，匹配哈希符号#也须要转义，如\#。

在正则式中加入注释也可以不用/x。我们也可以用(?#COMMENT)序列。乍一看，这没有让正则表达式更加易读。用这个序列就可以在字符串的右边注上表示的模式。不过，可以做并不意味着你应该这样做。我认为还是用/x更加易读：

```
$isbn = '0-596-10206-2';

$isbn =~ m/(\d+)(?#country)-(\d+)(?#publisher)-(\d+)(?#item)-([\dX])/i;

print <<"HERE";
Country code:     $1
Publisher code:   $2
Item:             $3
Checksum:         $4
HERE
```

全局匹配
Global Matching

《Learning Perl》中我们曾经告诉读者/g选项会进行所有的替换，不过它的威力远不止这点儿。我们可以在匹配操作符中使用它，它对于标量和列表有不同的效果。我们曾经说过匹配操作符在匹配上的时候会返回真，匹配不上的时候会返回假。这些还是适用的（我们不会对你撒谎），不过不只是一个布尔值了。/g在列表上下文中的行为是最有用的。加上/g选项后，匹配操作符会返回所有记住了的匹配内容：

```
$_ = "Just another Perl hacker,";
my @words = /(\S+)/g; # "Just" "another" "Perl" "hacker,"
```

即使正则表达式中只有一个记忆用的括号，它也匹配尽可能多的部分。一旦找到了一个匹

注3：也可以用\来转义空白，不过由于看不见空白，因此我们更喜欢用看得到的，比如\x20。

配的，Perl 也会从匹配的地方重新开始搜寻。待会再详细地讨论这个。有一种常见的用法和它有密切的关系——不要匹配的内容，只要匹配的数目：

```
my $word_count = () = /(\S+)/g;
```

这里涉及一个很少人知道但是很重要的规则：列表赋值操作的结果是等号右边列表中元素的个数。在这个例子中，就是匹配操作符返回的元素个数。这只对列表赋值操作有效。列表赋值操作就是把右边的列表赋给左边的列表。这也是我们在上面的例子中加入额外的()的原因。

在标量上下文中，/g 选项会做一些之前没有提到的额外工作。匹配成功后，Perl 会记住当前位置，再次匹配同样的字符串时，Perl 会从上次的位置开始。下面的这个例子每次返回一个成功匹配的结果：

```
$_ = "Just another Perl hacker,";
my @words = /(\S+)/g; # "Just" "another" "Perl" "hacker,"

while( / (\S+)/g ) # scalar context
        {
        print "Next word is '$1'\n";
        }
```

当我们再次匹配同一个字符串的时候，Perl 找到了下一个匹配：

```
Next word is 'Just'
Next word is 'another'
Next word is 'Perl'
Next word is 'hacker,'
```

我们甚至可以得到匹配的位置。对于给定的字符串（或者是默认的$_），内置的 pos() 操作符可以返回匹配的位置。每个字符串都会维护一个匹配的位置。字符串的偏移从 0 开始，所以没有匹配上的时候 pos() 会返回 undef 并且重置内部状态。这个功能只在/g 选项打开的时候有效（否则就没有必要用 pos() 了）：

```
$_ = "Just another Perl hacker,";
my $pos = pos( $_ );            # same as pos()
print "I'm at position [$pos]\n"; # undef

/(Just)/g;
$pos = pos();
print "[$1] ends at position $pos\n"; # 4
```

匹配失败的时候，Perl 会把 pos() 的值设为 undef。如果要继续匹配，Perl 会重新开始（可能造成死循环）：

```
my( $third word ) = /(Java)/g;
print "The next position is " . pos() . "\n";
```

一个题外话，我非常讨厌上面这种在 print 中用连接符把函数调用的结果拼在一起的方法。

Perl 没有专门的插入函数调用的机制，所以只好用点小技巧。我们在一个匿名的数组构造器[...]中调用函数，再立刻用@{...}来解除引用（注 4）：

```
print "The next position is @{ [ pos( $line ) ] }\n";
```

pos()运算符也可以作为左值，这是一种花哨的技巧，表示可以赋值来改变它。这样就可以让匹配操作符从希望的位置开始搜索。在下面的例子中，匹配到$line 中的第一个词后，匹配位置到了字符串的起始位置之后。然后用 index 查找当前位置之后的 h，再把 h 的偏移量赋给 pos($line)，下一次匹配就会从 h 的位置开始：

```
my $line = "Just another regex hacker,";

$line =~ /(\S+)/g;
print "The first word is $1\n";
print "The next position is @{ [ pos( $line ) ] }\n";

pos( $line ) = index( $line, 'h', pos( $line) );

$line =~ /(\S+)/g;
print "The next word is $1\n";
print "The next position is @{ [ pos( $line ) ] }\n";
```

全局查找锚定
Global Match Anchors

目前为止，后续的匹配会发生“漂移”，就是说匹配可能会从任何位置开始。为了让下次匹配严格地从上次的位置开始，可以用\G 锚定。它就像字符串起始锚定 ^ 一样，只是它是从当前匹配位置开始。如果匹配失败，Perl 会重置 pos()，匹配会从头开始。

在下面的例子中，我们先打开了\G 锚定，之后用非捕获括号把可能有的空格、\s*和单词、\w+括在了一起。为了增强可读性，我们打开\x 选项以分散模式的各个部分。这个模式只匹配上了前 4 个单词，因为第一个 hacker 后的逗号不在\w 中。因为下一个匹配必须从上次结束的地方开始——也就是逗号，而我们只能匹配空格或单词，所以匹配无法进行下去。匹配失败了，Perl 把当前位置重置为$line 的起始位置：

```
my $line = "Just another regex hacker, Perl hacker,";

while( $line =~ /  \G (?: \s* (\w+) )  /xg )
        {
        print "Found the word '$1'\n";
        print "Pos is now @{ [ pos( $line ) ] }\n";
        }
```

注 4：这和在字符串中插入函数调用是一个技巧：print "Result is : @{ [func (@args)] }"。

有一种方法能避免 Perl 重置匹配的位置。如果想要 Perl 继续匹配下去，可以加上/c 选项。它会让 Perl 在匹配失败的情况下也不会重置匹配位置。这样就可以没有代价地试一些东西。如果一个模式不行，可以在同样的位置试试别的。这个功能是可怜人的词法解析器。这里是一个头脑简单的句子解析器：

```
my $line = "Just another regex hacker, Perl hacker, and that's it!\n";

while( 1 )
        {
        my( $found, $type )= do {
                if( $line =~ /\G([a-z]+(?:'[ts])?)/igc )
                        { ( $1, "a word"          ) }
                elsif( $line =~ /\G (\n) /xgc           )
                        { ( $1, "newline char"    ) }
                elsif( $line =~ /\G (\s+) /xgc          )
                        { ( $1, "whitespace"      ) }
                elsif( $line =~ /\G ( [[:punct:]] ) /xgc  )
                        { ( $1, "punctuation char" ) }
                else
                        { last; ()                }
                };

        print "Found a $type [$found]\n";
        }
```

让我们再看看这个例子。如果我们想加入更多匹配的内容该怎么办？我们只能给决策结构再加入一个新的分支。这可不好玩。这里有大量的重复代码做着同样的事情：匹配某个东西，返回$1 和一个说明。其实可以不用这样。我重写代码去掉了这个重复的结构。我们把正则表达式保存在数组@items 中。用前面提到的 qr//引用正则表达式，再把它们按希望尝试的顺序排列起来。循环 foreach 依次遍历所有的模式直到找到能够匹配的位置。找到之后，会打印出一条消息和$1 的值。这样如果想加入更多的 token，只用在@items 中添加说明就可以了：

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

my $line = "Just another regex hacker, Perl hacker, and that's it!\n";

my @items = (
        [ qr/\G([a-z]+(?:'[ts])?)/i,  "word"        ],
        [ qr/\G(\n)/,                 "newline"     ],
        [ qr/\G(\s+)/,                "whitespace"  ],
        [ qr/\G([[:punct:]])/,        "punctuation" ],
        );

LOOP: while( 1 )
       {
       MATCH: foreach my $item ( @items )
              {
```

```
            my( $regex, $description ) = @$item;
            my( $type, $found );

            next unless $line =~ /$regex/gc;

            print "Found a $description [$1]\n";
            last LOOP if $1 eq "\n";

            next LOOP;
            }
    }
```


让我们分析一下这个例子。所有的匹配都需要/gc 选项，所以它放在了循环 foreach 的匹配操作符中。可是匹配单词的时候还需要/i 选项。我们不能把它加到匹配操作符上，因为其他的分支可能不需要。所以选项/i 加到了@items 中单词的正则表达式中，把它变成了大小写无关的。如果要让正则表达式像之前的那样易读，可以用选项(?ix)。顺便插一句，如果大部分的正则表达式是大小写无关的，我们可以把/i 加入匹配操作符中，再在需要的地方用(?-i)关掉大小写无关。

前后查找
Lookarounds

前后查找是正则式的通用锚定符。在《Learning Perl》中，介绍了多种锚定符，如^、$和\b，我们刚刚还介绍了\G 锚定符。利用前后查找，可以用正则式定义自己的锚定符。和其他锚定符一样，它们不属于模式部分，也不会消耗字符串的任何部分。它们指定了一种必须满足的条件，但是不算在整个模式匹配的字符串中。

前后查找有两种类型：**前向查找**，在当前匹配位置之后判断某个条件是否满足；**后向查找**，在当前匹配位置之前判断某个条件是否满足。听起来很简单，但是这些规则很容易用错。关键是要记住让它先锚定当前匹配位置，然后再找出是在哪边检查。

前向查找和后向查找都有两种类型：**正向**和**负向**。正向前后查找要求某个模式必须被匹配上。负向前后查找要求某个模式不能被匹配上。不管用的是哪一种，都须要记住是相对于当前的匹配位置，而不是字符串的其他地方。

前向查找断言，(?=PATTERN)和(?!PATTERN)
Lookahead Assertions, (?=PATTERN) and (?!PATTERN)

前向查找断言用来检查紧挨着当前匹配位置之前的部分。它不会消耗字符串。如果结果为真，匹配会从当前的匹配位置进行下去。

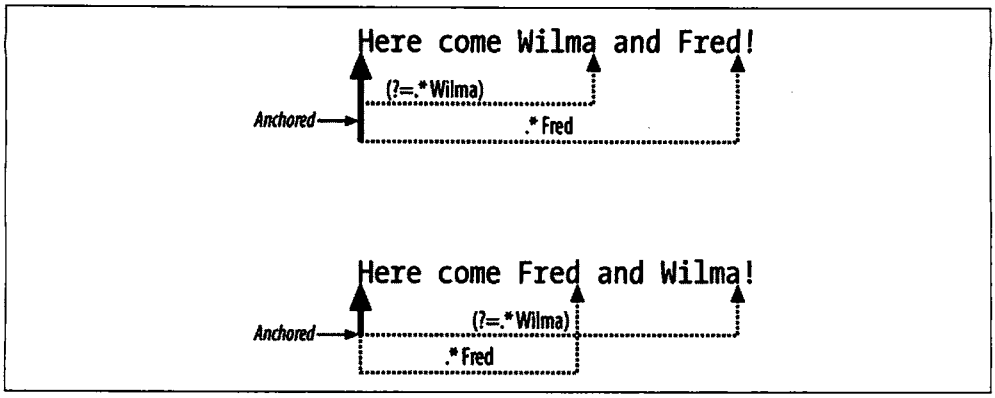


图 2-1：正向前向查找断言(?=.*Wilma)把模式锚定在了字符串的开始

正向前向查找断言

在《Learning Perl》中，有一个练习是查找在输入中同时出现“Fred”和“Wilma”的行，忽略它们的出现顺序。我们当时想告诉 Perl 新手的窍门是两个表达式会比一个表达式简单。一个方法是利用选择操作把两种顺序都试一遍。另一种方法是把他们分成两个正则表达式。

```
#/usr/bin/perl
# fred-and-wilma.pl

$_ = "Here come Wilma and Fred!";
print "Matches: $_" if /Fred.*Wilma|Wilma.*Fred/;
print "Matches: $_" if /Fred/ && /Wilma/;
```

我们可以用**正向前向查找断言**(?=PATTERN)来构造一个简单的正则式。正则式中的断言不会消耗字符串中的文字，但是当它失败的时候，整个匹配也会失败。在这个例子中，我们在正向前向查找断言中用了.*Wilma。这个模式必须匹配上当前匹配位置之后的内容：

```
$_ = "Here come Wilma and Fred!";
print "Matches: $_" if /(?=.*Wilma).*Fred/;
```

由于我们把它用在了模式的开头，它必须匹配上字符串的开头部分。也就是说，字符串的开头必须是任意多个字符（非换行）加上 Wilma。如果匹配成功，它就把模式的其余部分锚定在当前位置（字符串的开始）。图 2-1 展示了两种可能的方式，取决于字符串中 Fred 和 Wilma 的顺序。.*Wilma 锚定了开始匹配的位置。橡皮筋.*——可以用来匹配任意多个非换行的字符，锚定在字符串的开始位置。

不过，观察前后查找失败的例子有助于理解前后查找。我们稍微改变了一下模式，去掉了前向查找断言中的.*。对于第一个例子，它还是能工作的。但是当字符串中 Fred 和 Wilma 倒过来的时候就不行了：

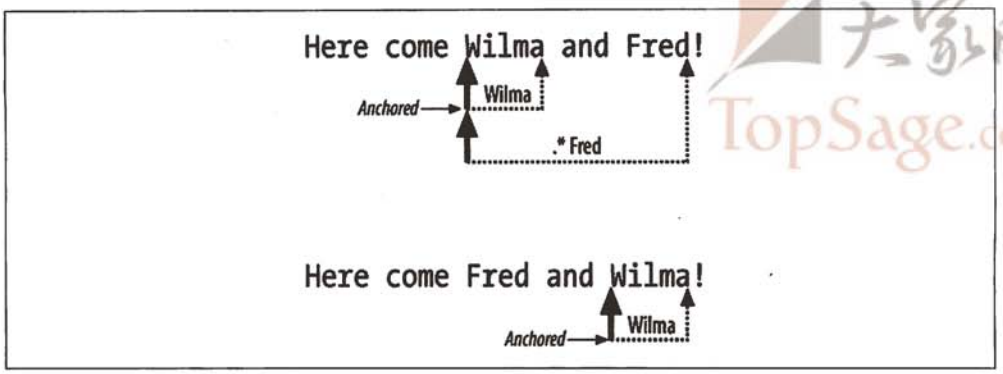


图 2-2：正向前向查找断言(?=Wilma)把模式锚定在了 Wilma

```
$_ = "Here come Wilma and Fred!";
print "Matches: $_" if /(?=Wilma).*Fred/; # Works

$_ = "Here come Fred and Wilma!";
print "Matches: $_" if /(?=Wilma).*Fred/; # Doesn't work
```

图 2-2 显示了发生的事情。在第一个例子中，前向查找锚定在了 `Wilma` 的开始位置。正则表达式引擎从字符串的起始位置开始检查断言，发现不成立，然后移到下一个位置再次检查。这个过程一直持续下去直至遇到 `Wilma`。于是锚定在了 `Wilma` 处。之后的模式匹配就得从这个位置开始。

在第一个例子中，`.*Fred` 从锚定位置开始能够匹配上，因为 `Fred` 在 `Wilma` 之后。图 2-2 的第二个例子也是同样的过程。正则表达式引擎从字符串的起始位置开始检查断言，发现不成立，然后移到下一个位置再次检查。当前向查找断言成立的时候，已经越过了 `Fred`。模式的剩余部分只能从锚定位置开始，于是无法匹配。

因为前向匹配断言不消耗字符串的任何部分，所以当我们不想丢掉匹配上的某部分模式时，可以把它放在 `split` 的模式参数中。在这个例子中，我们想把 studly cap 样式的字符串中的单词分割开。我们希望基于开头的大写字母分隔，并且保留首字母，所以可以用前向查找断言而不是会消耗字符的普通字符串。这和分隔符保留模式也不一样，因为分隔用的模式并不是分隔符，它只是一个锚定符：

```
my @words = split /(?=[A-Z])/, 'CamelCaseString';
print join '_', map { lc } @words; # camel_case_string
```

负向前向断言

假设我们要找到输入中包含 `Perl` 但不是 `Perl6` 或 `Perl 6` 的行。我们可以在模式中用减号指定 `perl` 的 `l` 后面的字符不是 `6`，同时还须要用单词边界锚定符`\b` 确保不会匹配到其他单词的中间，比如“BioPerl”或“PerlPoint”：

```
#!/usr/bin/perl
# not-perl6.pl

print "Trying negated character class:\n";
while( <> )
        {
        print if /\bPerl[^6]\b/;  #
        }
```

我们可以试试一些例子：

```
# sample input
Perl6 comes after Perl 5.
Perl 6 has a space in it.
I just say "Perl".
This is a Perl 5 line
Perl 5 is the current version.
Just another Perl 5 hacker,
At the end is Perl
PerlPoint is PowerPoint
BioPerl is genetic
```

它并不是对所有应该生效的行都有效。它只找到了 4 个在 Perl 后没有跟着 6 的行，和 1 个在 Perl 和 6 之间有个空格的行：

```
Trying negated character class:
        Perl6 comes after Perl 5.
        Perl 6 has a space in it.
        This is a Perl 5 line
        Perl 5 is the current version.
        Just another Perl 5 hacker,
```

它不能正常工作是因为它要求 Perl 的 l 后面必须有一个字符。不仅如此，我们指定了单词边界。如果 l 后面的字符不是单词字符，比如 I just say "Perl"中的"，单词边界的条件就不成立了。如果去掉\b，PerlPoint 又会被匹配上。我们甚至还没有来得及处理 Perl 和 6 之间有一个空格的情况。因此，我们需要一些更好的工具。

为了简化这些问题，我们可以用负向前向断言。我们不想匹配 l 后面的字符，而断言不消耗字符，它就是最合适的工具。我们只用声明如果 Perl 后面有字符，它不能是 6，即使在它们之间有空格分隔。负向前向断言使用(?!PATTERN)。为了解决这个问题，我们用\s?6 作为断言中的模式，它表示 6 之前可能存在空格：

```
print "Trying negative lookahead assertion:\n";
while( <> )
        {
        print if /\bPerl(?!\s?6)\b/;  # or /\bPerl[^6]/
        }
```

现在输出结果中有了所有应该有的行：

```
Trying negative lookahead assertion:
        Perl6 comes after Perl 5.
```

```
I just say "Perl".
This is a Perl 5 line
Perl 5 is the current version.
Just another Perl 5 hacker,
At the end is Perl
```


记住(?!PATTERN)是前向查找断言，所以它是从当前匹配位置之后开始查找。这正是下面例子中的模式能匹配上的原因。下面的前向查找声明在 bar 的 b 之前的不是 foo。因为下一个是 bar 而不是 foo，所以匹配成功。人们常常把这个错误地理解为 bar 之前的不能是 foo。其实(?!foo)和 bar 都是从相同的位置开始匹配的，由于 bar 不是 foo，所以两个条件都能满足：

```
if( 'foobar' =~ /(?!foo)bar/ )
        {
        print "Matches! That's not what I wanted!\n";
        }
else
        {
        print "Doesn't match! Whew!\n";
        }
```

后向查找断言，(?<!PATTERN)和(?<=PATTERN)

Lookbehind Assertions, (?<!PATTERN) and (?<=PATTERN)

除了向前查找将要出现的字符串，我们也可以用后向查找检查正则表达式引擎已经处理过的字符串部分。由于 Perl 的实现原因，后向查找断言必须是固定宽度的，所以无法在断言中使用可变宽度的修饰符。

现在我们可以试着匹配没有跟在 foo 后面的 bar。我们不能用前一节介绍的**负向前向查找**，因为它会向前查找。负向后向查找，记为(?<!PATTERN)，可以往后查找。这正是我们需要的。现在我们可以得到正确的结果了：

```
#!/usr/bin/perl
# correct-foobar.pl

if( 'foobar' =~ /(?<!foo)bar/ )
        {
        print "Matches! That's not what I wanted!\n";
        }
else
        {
        print "Doesn't match! Whew!\n";
        }
```

当正则表达式引擎解析到 bar 的时候，它已经处理完了之前的部分，所以后向查找断言不能是变长的模式。我们不能用变长修饰符构造一个变长模式，因为正则表达式引擎不会在字符串中回溯。我们没法检查 fooo 中可变数目的 o：

```
'fooooobar' =~ /(?<!fo+)bar/;
```

当我们试图这样做的时候，会得到错误提示。即使它只是说还没有实现，你千万不要屏住呼吸等待它实现：

```
Variable length lookbehind not implemented in regex...
```

正向后向断言也会向后查找，不过它的模式一定**不**能匹配上。我唯一用到它的时候是在替换操作中和其他断言一起使用。同时用前向和后向断言，可以让一些替换操作易于阅读。

一个例子是本书中我们大量地用到了有连字符（-）分隔和没有连字符分隔的单词，因为我不确定应该用哪一种。应该用 builtin 还是用 built-in？我会根据心情和打字技巧选用其中一种（注 5）。

我须要清理这种不一致地方。我们知道连字符左边的单词和右边的单词，当它们碰到一起的时候，应该加一个连字符。如果稍微考虑一下这个问题，就会发现刚才描述的是一个使用前后查找的绝佳情况：我们希望在某个特定位置加上一些东西，而且知道它的前后是什么。下面的例程用正向后向查找检查左边的文本，用正向前向查找检查右边的文本。正则式只在两边连在一起时才能匹配上，所以它表示发现了遗漏的连字符。进行替换操作之后，它就把连字符放在了匹配的位置上。我们也不用担心具体的文本内容是什么：

```
@hyphenated = qw( built-in );

foreach my $word ( @hyphenated )
        {
        my( $front, $back ) = split /-/, $word;

        $text =~ s/(?<=$front)(?=$back)/-/g;
        }
```

如果上面的例子还不够复杂的话，看看这个吧。让我们用前后查找在数字中加上逗号。Jeffery Friedl 在《Mastering Regular Expressions》中给出了一个解法，他给美国人口数加上了逗号（注 6）：

```
$pop = 301139843;  # that's for Feb 10, 2007

# From Jeffrey Friedl
$pop =~ s/(?<=\d)(?=(?:\d\d\d)+$)/,/g;
```

这个方法在大部分情况下是有效的。正向后向查找 (?<=\d) 会匹配一个数字，正向前向查找 (?=(?:\d\d\d)+$) 则找到所有的 3 个一组的数字直到字符串结束。当我们遇到浮点数，比

注 5：作为出版社，O'Reilly Media 有丰富的处理这个问题的经验，它维护了一个单词的列表。虽然很多作者（比如我）没有读过它：*http://www.oreilly.com/oreilly/author/stylesheet.html*。

注 6：如果您是在本书出版后很久读到这本书，您可以用美国统计署的最新统计数据：*http://www.census.gov/main/www/popclock.html*。

如价格时，这个方法就不行了。一个例子就是我的股票金额，它会精确到小数点后 4 位。当我们用这个方法来处理股票金额时，得到的结果中小数点左边没有逗号，小数部分却有一个逗号。出现这种问题是因为模式中有个行尾锚定符：

```
$money = '$1234.5678';

$money =~ s/(?<=\d)(?=(?:\d\d\d)+$)/,/g;  # $1234.5,678
```

我们可以对模式稍加修改，把行尾锚定换成单词边界锚定，\b。这看起来有点奇怪，不过考虑到数字也是单词字符就能理解了。于是结果中小数点左边有了逗号，不过小数部分还是有多余的逗号：

```
$money = '$1234.5678';

$money =~ s/(?<=\d)(?=(?:\d\d\d)+\b)/,/g;  # $1,234.5,678
```

其实我们是想在正则表达式的前一部分用后向查找匹配一个数字，但是数字不能在小数点之后。这是一个负向后向查找：(?<!\.\d)。它们都匹配在一个位置，所以它们检查的范围有所重叠也没有关系，只要它们都能按我希望的方式工作：

```
$money = $'1234.5678';

$money =~ s/(?<!\.\d)(?<=\d)(?=(?:\d\d\d)+\b)/,/g; # $1,234.5678
```

成功了！有点糟糕的是我还很想找个借口加入一个负向前向查找呢。不过它已经非常复杂了，所以我现在就身体力行一下，用/x 把模式字符串变得更加易读：

```
$money =~ s/
        (?<!\.\d)          # 匹配位置之前不能是.和一个数字

        (?<=\d)            # 匹配位置之前必须有一个数字
                           # <--- 当前匹配位置
        (?=                # 当前匹配位置之后必须有的字符组
        (?:\d\d\d)+        # 一组或多组三个连续的数字
          \b               # 匹配单词边界（数字的右边或字符串的结尾）
        )

        /,/xg;
```

解读正则表达式

Deciphering Regular Expressions

当尝试分析一个正则式的时候——不管是别人的代码还是自己（可能是很久以前）写的，我们可以打开 Perl 正则式的调试模式（注 7）。Perl 的-D 开关会打开 Perl 解析器的调试选项

注 7：使用正则表达式的调试模式需要解析器是用-DDEBUGGING 选项编译的。运行 perl -v 可以显示解析器的编译选项。

（不是你的程序，见第 4 章）。这个开关需要一系列字母和数字来表示应该打开的功能。选项-Dr 会打开解析和执行正则式时的调试功能。

我们可以用一个小程序来检查一个正则式。程序的第一个参数是待匹配的字符串，第二个是正则式。我们把这个程序保存为 *explain-regex*：

```
#!/usr/bin/perl

$ARGV[0] =~ /$ARGV[1]/;
```

我们用字符串 `Just another Perl hacker` 和正则式 `Just another (\S+) hacker` 试试这个程序，输出的结果主要有两段，文档 *perldebguts* 中有详尽的介绍。首先，Perl 编译正则式，-Dr 会显示出 Perl 是如何解析正则式的。它会显示出正则式节点，比如 EXACT 和 NSPACE，以及所有的优化措施，比如锚定的"Just another"。然后，Perl 会尝试匹配目标串，并逐个节点地显示出匹配的过程。输出中包含了大量的信息，它准确地告诉了我们 Perl 做的事情：

```
$ perl -Dr explain-regex 'Just another Perl hacker,' 'Just another (\S+) hacker,'
Omitting $` $& $' support.

EXECUTING...

Compiling REx `Just another (\S+) hacker,'
size 15 Got 124 bytes for offset annotations.
first at 1
rarest char k at 4
rarest char J at 0
   1: EXACT <Just another >(6)
   6: OPEN1(8)
   8:   PLUS(10)
   9:     NSPACE(0)
  10: CLOSE1(12)
  12: EXACT < hacker,>(15)
  15: END(0)
anchored "Just another " at 0 floating " hacker," at 14..2147483647 (checking anchored) minlen 22
Offsets: [15]
               1[13] 0[0] 0[0] 0[0] 0[0] 14[1] 0[0] 17[1] 15[2] 18[1] 0[0] 19[8] 0[0] 0[0] 27[0]
Guessing start of match, REx "Just another (\S+) hacker," against "Just another Perl hacker,"...
Found anchored substr "Just another " at offset 0...
Found floating substr " hacker," at offset 17...
Guessed: match at offset 0
Matching REx "Just another (\S+) hacker," against "Just another Perl hacker,"
  Setting an EVAL scope, savestack=3
   0 <> <Just another>    |  1:  EXACT <Just another >
  13 <ther > <Perl ha>    |  6:  OPEN1
  13 <ther > <Perl ha>    |  8:  PLUS
                             NSPACE can match 4 times out of 2147483647...
  Setting an EVAL scope, savestack=3
  17 < Perl> < hacker>    | 10:    CLOSE1
  17 < Perl> < hacker>    | 12:    EXACT < hacker,>
  25 <Perl hacker,> <>    | 15:    END
```

```
Match successful!
Freeing REx: `"Just another (\\S+) hacker,"'
```

Perl 自带的编译器指令 re 有一个调试模式，该模式不须要解析器打开-DDEBUGGING 选项。用 use re 'debug'打开调试模式后，它就会对整个程序有效。它不像大部分编译器指令那样有词义范围。我们修改前面的程序用编译器指令 re 替换掉命令行开关：

```
#!/usr/bin/perl

use re 'debug';

$ARGV[0] =~ /$ARGV[1]/;
```

其实不修改程序也可以用 re。我们可以通过命令行参数来指定：

```
$ perl -Mre=debug explain-regex 'Just another Perl hacker,' 'Just another (\S+) hacker,'
```

把正则表达式作为参数传给这个程序后，我们得到的结果和前面那个用-Dr 的程序几乎完全一样。

尽管有点过时了，模块 YAPE::Regex::Explain 可以用非常朴实的英语解释正则式。它会解析正则表达式并解释每个部分的功能。它不能解释语义上的目的，但是我们不能奢求太多。用一个小小的程序就可以分析通过命令行指定的正则表达式：

```
#!/usr/bin/perl

use YAPE::Regex::Explain;

print YAPE::Regex::Explain->new( $ARGV[0] )->explain;
```

即使是一个短小简单的正则式，我们也会得到大量的输出：

```
$ perl yape-explain 'Just another (\S+) hacker,'
The regular expression:

(?-imsx:Just another (\S+) hacker,)

matches as follows:

NODE                    EXPLANATION
-----------------------------------------------------------------------
(?-imsx:                group, but do not capture (case-sensitive)
                        (with ^ and $ matching normally) (with . not
                        matching \n) (matching whitespace and #
                        normally):
-----------------------------------------------------------------------
  Just another            'Just another '
-----------------------------------------------------------------------
  (                       group and capture to \1:
-----------------------------------------------------------------------
    \S+                     non-whitespace (all but \n, \r, \t, \f,
                            and " ") (1 or more times (matching the
                            most amount possible))
```

```
 )                      end of \1
-------------------------------------------------------------------------
  hacker,                ' hacker,'
-------------------------------------------------------------------------
)                     end of grouping
-------------------------------------------------------------------------
```

最后的思考

Final Thoughts

快到本章的结尾了，但是还有很多有用的正则表达式的功能没有介绍。读者可以把这一节看做一个可以自己阅读的快速介绍。

除了使用\w（单词字母）、\d（数字）等转义序列所表示的简单的字符类之外，我们还可以用 POSIX 字符类。我们在名字的两边用方括号和冒号括住来表示一些字符类：

```
print "Found alphabetic character!\n" if  $string =~ m/[:alpha:]/;
print "Found hex digit!\n"            if  $string =~ m/[:xdigit:]/;
```

我们可以在第一个冒号之后加上^来表示取反：

```
print "Didn't find alphabetic characters!\n" if  $string =~ m/[:^alpha:]/;
print "Didn't find spaces!\n" if  $string =~ m/[:^space:]/;
```

我们也可以通过指定一个具名属性来表示同样的功能。\p{Name}序列（小写 p）包含了该具名属性的所有字符，\P{Name}序列（大写 P）表示它的补集：

```
print "Found ASCII character!\n"      if  $string =~ m/\p{IsASCII}/;
print "Found control characters!\n"   if  $string =~ m/\p{IsCntrl}/;

print "Didn't find punctuation characters!\n" if  $string =~ m/\P{IsPunct}/;
print "Didn't find uppercase characters!\n"  if  $string =~ m/\P{IsUpper}/;
```

模块 Regexp::Common 提供了很多经过测试的可靠的正则式。它们涵盖了常见的功能，如网络地址、数字、邮编，甚至是脏话。它提供了一个多级的哈希表%RE，其中哈希表的值是正则式。如果不喜欢，我们也可以用它的函数接口：

```
use Regexp::Common;

print "Found a real number\n" if $string =~ /$RE{num}{real}/;

print "Found a real number\n" if $string =~ RE_num_real;
```

如果想积累自己的模式，我们可以用 Regexp::English。它用一系列的嵌套调用的方法返回一个正则表达式的对象。在真正的程序中，这可能不是你想要的，不过研究一下它还是蛮有趣的：

```
use Regexp::English;

my $regexp = Regexp::English->new
        ->literal( 'Just' )
              ->whitespace_char
        ->word_chars
              ->whitespace_char
        ->remember( \$type_of_hacker )
        ->word_chars
        ->end
              ->whitespace_char
        ->literal( 'hacker' );

$regexp->match( 'Just another Perl hacker,' );

print "The type of hacker is [$type_of_hacker]\n";
```


如果你真的想知道正则表达式的具体细节，我推荐你去看看 O'Reilly 出版的 Jeffrey Friedl 的《Mastering Regular Expressions》。你不仅会学到一些高级功能，还会了解正则表达式的工作原理，掌握优化你的正则式的方法。

总结
Summary

本章涵盖了 Perl 正则式引擎的一些最常用的高级功能。引用操作符 `qr()` 可以编译正则表达式供之后使用，并能把结果作为引用返回。特殊的 `(?)` 序列可以让我们的正则表达式更加强大和简单。锚定符 `\G` 可以让我们从上次离开的地方继续匹配。选项 `/c` 可以让我们尝试多种模式而不会在匹配失败的时候重置当前匹配的位置。

深入阅读
Further Reading

perlre 是 Perl 正则式的说明文档，*perlretut* 是正则式的入门教程。不要把它和 *perlreftut* 弄混了，后者是引用的入门教程。更复杂的是，*perlreref* 是正则表达式的速查手册。

perldebguts 中有正则表达式调试的详细介绍。它解释了 `-Dr` 和 `re 'debug'` 的输出格式。

《Perl Best Practices》有一章是关于正则表达式的，该章节详细地介绍了 `\x` "扩展格式"的功能。

《Mastering Regular Expressions》介绍了正则表达式的基本知识，比较了正则式在不同语言中的实现。Jeffrey Friedl 对前向查找和后向查找操作符的介绍尤其精彩。如果你真的想了解正则表达式，一定要看这本书。

Simon Cozens 在 Perl.com 的两篇文章中介绍了正则表达式的高级功能："Regexp Power"（*http://www.perl.com/pub/a/2003/06/06/regexps.html*）和"Power Regexps, Part II"（*http://www.perl.com/ pub/a/2003/07/01/regexps.html*）。

网站 *http://www.regular-expressions.info* 上有很多很棒的关于正则表达式和它在不同语言中的实现的讨论。

第3章
安全编程技术
Secure Programming Techniques

我们无法控制别人运行我们的程序的方式，也不能控制别人的输入。而一旦有机会，人们又总是会做各种出乎我们意料之外的事情。所以当我们的程序要把输入传递给其他程序的时候，这就成了一个问题。如果允许任何人运行我们的程序——就像 CGI 程序那样，我们就必须格外小心。Perl 有很多功能可以帮助我们免受恶意输入的威胁，不过它们只在被聪明地使用时才是有效的。

不好的数据会浪费你的一整天
Bad Data Can Ruin Your Day

如果不对传给系统函数的参数倍加小心，我们就会给自己带来麻烦。看看这行“貌似”没有问题的打开文件的代码：

```
open my($fh), $file or die "Could not open '$file': $!";
```

它乍看起来没有什么问题，那么问题出在哪里呢？和大多数情况一样，问题来自于多种因素的综合作用。变量$file的值是什么，它的值又是从哪里得到的呢？在实际的代码审查过程中，我见过人们从@ARGV或环境变量中获取$file，这两种方式都使得我们——程序员无法控制$file的值：

```
my $file = $ARGV[0];

# OR
my $file = $ENV{FOO_CONFIG};
```

这怎么会造成问题呢？看看 Perl 文档中对 open 函数的介绍吧。你完整地读过 *perlfunc* 中对该函数 400 多行的说明或它的手册——*perlopentut* 吗？在 Perl 中有如此多的打开资源的方法，以至于人们不得不给 open 函数提供专门的说明文档。其中许多方法都须要打开一个到其他程序的管道：

```
open my($fh), "wc -l $pod |";

open my($fh), "| mail joe@example.com";
```

要想滥用这些程序很容易。我们只须把合适的内容传到$file，这样就能够执行打开管道的操作而不是打开文件的操作了：

```
$ perl program.pl "| mail joe@example.com"

$ FOO_CONFIG="rm -rf / |" perl program
```

如果允许别人为我运行程序的话，这个问题会变得非常严重。千里之堤溃于蚁穴。只要把足够多的漏洞组合在一起，黑客最终可以控制整个系统。

当然，我们也有别的手段来防止这个问题的发生，本章结尾会讨论到这些方法。但一般而言，当我们得到外部输入的时候，须要在使用前确保它是我们所期望的。如果写程序的时候足够小心，我们就不用知道 open 函数能做的所有事情。这种方法的工作量并不会比粗枝大叶的方法大很多，而且用了这种方法，我们就不必再为这类问题操心了。

污点检测
Taint Checking

使用配置文件是一种从程序外部获得数据的方法。当用户选择了输入内容的时候，他们就已经控制了程序所做的事情。这一点在我们为其他人写程序的时候变得更加重要。通常，我们相信自己会给自己的程序提供正确的数据。但是，对于别人，即使是那些意图最单纯的用户来说，他们也可能会弄错。

如果打开了污点检测，Perl 就不允许我们用源代码之外未经检测的数据去影响程序外部的东西了。Perl 会停止程序并报告错误。在我们深入之前，请记住污点检测不会阻止不好的事情发生。它只会帮助我们找到可能出现问题的位置，并通知我们去解决那些问题。

用-T 选项打开污点检测后，Perl 会把所有来自程序外部的数据标记为有污点的，就是说它们是不安全的。Perl 不会让我们用这些数据去处理程序外部的任何东西。这样，我们就避免了很多和其他程序通信时可能产生的安全问题。这是一个“非此即彼”的功能：一旦打开，它就对整个程序和所有的数据起作用。

Perl 会在编译时完成污点检测的准备工作，因此污点检测在程序的整个执行过程中都会起作用。要打开污点检测的功能，必须很早就通知 Perl。下面这个小程序用外部命令 echo 打印一条消息。我们可以在 shebang 行（译注 1）加上-T 开关：

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

system qq|echo "Args are @ARGV"|;
```

只要我们直接运行该程序，污点检测就能够正常工作。操作系统会通过 shebang 行找到须要运行的解析器（Perl）和要传递的开关，而 Perl 会抓住 PATH 的安全漏洞。在 system 调用中只使用程序名时，system 命令会使用 PATH 变量中的路径找到程序。用户可以在运行我们的程序前把 PATH 改成他所希望的任何值，而且我们允许程序外部的数据影响程序的工

译注 1：shebang 指的是脚本文件开头的字符“#!”。UNIX 类系统把这两个字符的出现当作文件是脚本的信号，并尝试用第一行指定的解析器执行该文件。更多介绍参见 *http://en.wikipedia.org/wiki/Shebang_line*。

作。运行程序的时候，Perl 发现 PATH 有可能会被篡改，于是就停止执行程序并提示我们它有安全问题：

```
$ ./tainted-args.pl foo
Insecure $ENV{PATH} while running with -T switch at
./tainted-args.pl line 3.
```

如果我们直接运行 perl 命令，它就无法及时得到 shebang 行的开关，从而无法打开污点检测功能。因为污点检测对整个程序有效，所以 perl 命令须要一开始就知道它才能打开该功能。当我们运行程序的时候，会得到一个严重错误。具体的错误消息取决于你的 perl 的版本，在这里我们将展示其中的两种。早期版本的 perl 显示的消息在上面，比较简洁；后来的版本显示的消息在下面，有更多的信息：

```
$ perl tainted-args.pl foo
Too late for -T at peek-taint.pl line 1.

"-T" is on the #! line, it must also be used on the command
line at tainted-args.pl line 1.
```

最新版本的错误消息准确地告诉了我们应该做的事情。如果 shebang 行有-T 开关，用 perl 命令显式地运行程序时，我们就须要在命令行中加上-T 开关。这样，用户就没法替换 perl 程序从而绕过污点检测了：

```
$ perl -T tainted-args.pl foo
```

一个小小的安全提示——我可能有点胡思乱想（如果考虑安全问题时没有胡思乱想，很可能你有什么地方弄错了），没有什么能够阻止别人修改 perl 解析器的源代码以忽略-T 开关，或者改写我的源代码以去掉-T 开关。不要因为打开了污点检测就天真地觉得安全了。要记住，污点检测是一个开发工具，而不是保险箱。

这儿就有一个假装自己是真正的 perl 的程序，它利用了真正的 Perl 解析器能够捕获的 PATH 变量的安全漏洞。如果我能让你相信这个程序就是 perl（有可能是把它放在你的搜索目录的前面），那么污点检测就完全失效了。它会从参数列表和 shebang 行中删除-T，保存好新的程序，再从 PATH 中（当然，要排除掉它自己）找到真正的 perl 来运行新的程序。污点检测是一个工具，而不是灵丹妙药。它只会告诉我们哪里还须要做更多的工作。我说得够清楚了吧？

```
#!/usr/bin/perl
# perl-untaint (rename as just 'perl')
use File::Basename;

# 去掉命令行中的-T
my @args = grep { ! /-T/ } @ARGV;

# 确定程序的名字。通常程序的名字是所有开关之后的
# 第一个字符串（或者是在结束所有的开关的'--'之后）。
# 这段程序在最后的开关有参数的情况下不能正常工作，
# 不过处理这种情况只需要一些额外的工作而已
my( $double )    = grep  { $args[$_] eq '--'  }  0 .. $#args;
my  @single      = grep  { $args[$_] =~ m/^-/ }  0 .. $#args;
```

```
my $program_index = do {
          if( !$double and !@single ) { 0 }
        elsif( $double )              { $double + 1 }
        elsif( @single )              { $single[-1] + 1 }
        };

my $program = splice @args, $program_index, 1, undef;

unless( -e $program )
        {
        warn qq|Can't open perl program "$program": No such file or directory\n|;
        exit;
        }

# 把程序保存到另一个位置（放在当前目录下可能也能正常工作）
my $modified_program = basename( $program ) . ".evil";
splice @args, $program_index, 1, $modified_program;

open FILE, $program;
open TMP, "> $modified_program" or exit; # quiet!

my $shebang = <FILE>;
$shebang =~ s/-T//;

print TMP $shebang, <FILE>;

# 找到我自己(路径中的第一个)，并从搜索路径中删除
# 当.也在路径中的时候这个处理就显得尤其有用
my $my_dir = dirname( `which perl` );
$ENV{PATH} = join ":", grep { $_ ne $my_dir } split /:/, $ENV{PATH};

# 重新设置路径找到真正的perl程序
chomp( my $Real_perl = `which perl` );

# 用真正的perl运行程序（没有污点检查）
system("$Real_perl @args");

# 清理痕迹，看起来就像没有我们这个程序一样。
unlink $modified_program;
```

用警告代替严重错误

Warnings Instead of Fatal Errors

打开-T开关之后，违反污点检测规则的行为会被当作严重错误处理。一般而言，这是一件好事情。但是，有时拿到的是一个没有经过污点检查的程序，我们可能还是希望能够运行这个程序。它没有通过污点检测，可是这不是我的过错，因此Perl提供了一种比较温和的污点检测方式。

开关-t（-T的小兄弟）做的事情和普通的污点检测完全一样，只是它在发现问题的时候只会给出警告。这是一个仅为开发提供的功能，供我们在向外界发布程序之前检测程序的问题：

```
$ perl -t print_args.pl foo bar
Insecure $ENV{PATH} while running with -t switch at print_args.pl line 3.
Insecure dependency in system while running with -t switch at print_args.pl line 3.
```

类似地，开关-U 允许 Perl 执行不安全的操作，它实际上关掉了污点检测功能。可能我已经给一个没有通过污点检测的程序加了-T 开关，但是由于正在解决这个问题，所以须要查看它的运行情况：

```
$ perl -TU print_args.pl foo bar
Args are foo bar
```

不过，我们还是须要在命令行使用-T 开关。不然会得到和前面一样的“too late”消息，程序也不会运行：

```
$ perl -U print_args.pl foo bar
Too late for "-T" option at print_args.pl line 1.
```

如果打开了警告（就像我们一直做的那样，对吧？），我们就会得到没有通过污点检测的警告。警告的内容和用-t 得到的一样：

```
$ perl -TU -w print_args.pl foo bar
Insecure $ENV{PATH} while running with -T switch at print_args.pl line 3.
Insecure dependency in system while running with -T switch at print_args.pl line 3.
Args are foo bar
```

在程序内部，我们可以通过检测 Perl 的特殊变量${^TAINT}的值来了解实际情况。如果打开了任何污点检测模式（包括-U），这个变量的值就为真，否则为假（译注 2）。对于普通的会汇报严重错误的污点检测，它的值为 1；对于只汇报警告的模式，它的值为-1。不要试图修改它，它是个只读变量。记住，污点检测是个非此即彼的功能。

自动污点检测模式

Automatic Taint Mode

有些时候 Perl 会自动为我们打开污点检测。当 Perl 发现实际的和有效的用户或用户组不一样的时候（就是说我们在用和登录的用户不一样的用户或用户组运行程序），Perl 意识到有可能我们得到了过高的系统权限，于是会自动打开污点检测功能。这样，当其他用户用我们的程序处理系统资源时，他们就没有机会利用精心构造的输入来做他们不应该做的事情。这并不意味着程序是安全的，它只和聪明地使用污点检测时一样安全。

mod_perl

因为我们必须在 Perl 运行的早期打开污点检测，所以 mod_perl 须要在运行程序之前就知道污点检测是否须要打开。在 Apache 服务器的配置中，对于 mod_perl 1.x 我们使用了 PerlTaintCheck 指令：

```
PerlTaintCheck On
```

译注 2：关闭的时候，它的值为 0。

在 mod_perl 2 中，我们须要在 PerlSwitches 指令后加上-T 开关：

```
PerlSwitches -T
```

我们无法在*.htaccess* 或之后的其他配置文件中使用这些指令。我们只能对 mod_perl 的所有部分打开污点检测，这意味着所有通过 mod_perl 运行的程序——包括用 ModPerl::PerlRun 或 ModPerl::Registry（注 1）运行的明显没有问题的 CGI 程序——都要使用它。这可能会让用户有点心烦，不过在熟悉了这个更好的编程技术之后，他们就会抱怨新发现的其他的问题了。

有污点的数据

Tainted Data

数据不是有污点的，就是没有污点的。不存在部分或一半有污点的数据。Perl 只会把标量（数值和变量）标记为有污点的，所以即使一个数组或哈希表中的数据有污点，也不代表整个集合是有污点的。Perl 也不会把哈希表的键标记为有污点的，因为它们并不是真正的标量——它们没有标量的所有结构。先记住这一点，之后我们会详细解释。

有很多种方法可以检测数据是否有污点。最简单的是调用 Scalar::Util 模块的 tainted 函数：

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

use Scalar::Util qw(tainted);

# this one won't work
print "ARGV is tainted\n" if tainted( @ARGV );

# this one will work
print "Argument [$ARGV[0]] is tainted\n" if tainted( $ARGV[0] );
```

当我们从命令行指定参数的时候，由于它们来自程序之外，所以 Perl 认为它们是有污点的。数组@ARGV 没有问题，但是它的内容——$ARGV[0]却是有污点的：

```
$ perl tainted-args.pl foo
Argument [foo] is tainted
```

所有涉及有污点的数据的子表达式都是有污点的。有污点的数据就像病毒一样。下面的程序用 File::Spec 创建了一个路径，路径的第一部分是我的 home 目录。我们希望打开一个文件、逐行读入并把每一行打印到标准输出。这应该很简单，对吧？

```
#!/usr/bin/perl -T
use strict;
use warnings;

use File::Spec;
use Scalar::Util qw(tainted);

my $path = File::Spec->catfile( $ENV{HOME}, "data.txt" );
```

注 1：如果我们用的是 Apache 1.x 而不是 Apache 2.x，那么使用的模块应该是 Apache::PerlRun 和 Apache::Registry。

```
print "Result [$path] is tainted\n" if tainted( $path );
open my($fh), $path or die "Could not open $path";
print while( <$fh> );
```


问题出在环境变量上。哈希表%ENV中的所有值都来自程序外部，所以Perl把它们标记为有污点的。所有通过处理有污点的数据得到的值也是有污点的。这是个好事情，因为$ENV{HOME}可以是用户想要的任何值，包括恶意的内容，比如下面这一行的以“|”开始、之后运行一个命令的代码串。实际上，这种攻击程序的某些变体成功地拿到了一些大型Web站点的密码文件，这些网站的CGI程序做着类似的操作。即使得不到密码，一旦知道了系统中所有用户的名字，也可以用来发垃圾邮件：

```
$ HOME="| cat /../../../etc/passwd;" ./sub*
```

有了污点检测，我们会得到一个错误消息，因为Perl发现了我试图塞进文件名中的字符“|”：

```
Insecure dependency in piped open while running with -T switch at ./subexpression.pl?
line 12.
```

污点检测的副作用
Side Effects of Taint Checking

打开污点检测功能后，Perl做的工作不仅仅是标记有污点的数据，它也会忽略一些其他的东西，因为它们可能很危险。Perl在污点检测时会忽略PERL5LIB和PERLLIB，因为用户可以随意修改这两个变量让Perl执行他希望的任何代码。如果一个冒名顶替的*File/Spec.pm*模块出现在了Perl搜索路径的前面，我们的程序会找到它而不是Perl标准库的File::Spec模块。当我们运行程序的时候，Perl会找到这个File::Spec模块，然后当Perl试图调用它的某个方法时，可能会发生一些完全不同的事情。

要想避免忽略PERL5LIB，我们可以使用lib模块或-I开关。污点检测认为它们没有问题（不过这并不意味着这样做是安全的）：

```
$ perl -Mlib=/Users/brian/lib/perl5 program.pl
$ perl -I/Users/brian/lib/perl5 program.pl
```

我们甚至可以在命令行中引用变量PERL5LIB。我并不赞同这种做法，但是你须要知道这是一种别人可能会利用的绕过你的安全设置的方法：

```
$ perl -I$PERL5LIB program.pl
```

同样，Perl也认为PATH是安全漏洞的。否则，我们就可以用特权模式下运行的程序往某个不应该写入的地方写东西了。即使Perl认为它是安全的，我们也不能信任PATH，原因和我们不信任PERL5LIB一样。如果不知道程序在哪里，我们就不知道真正运行的是什么程序。

在这个例子中，我们调用 system 运行 cat 命令。因为依赖 PATH 找到可执行程序，所以我们并不知道真正运行的是哪个文件：

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

system "cat /Users/brian/.bashrc"
```

Perl 的污点检测功能会捕捉到这个问题：

```
Insecure $ENV{PATH} while running with -T switch at ./cat.pl line 3.
```

在 system 命令中用 cat 的完整路径也没有用。如果不知道 PATH 变量可以使用和不能使用的范围就永远是不安全的：

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

delete $ENV{PATH};

system "/bin/cat /Users/brian/.bashrc"
```

类似地，其他的环境变量如 IFS、CDPATH、ENV 或 BASH_ENV 也有同样的问题。它们的值对我们的程序所做的事情可能有潜在的影响。

去除数据的污点
Untainting Data

去除数据污点的唯一允许的方法是用正则表达式的记忆变量提取出数据中没有问题的部分。从设计上，Perl 就不会把正则表达式记忆变量中的数据标记为有污点的，即使源字符串已经被标记为有污点的了。Perl 相信我们能够写出安全的正则表达式。又一次，完全由我们自己来保证程序的安全性。

在下面这行代码中，我们去除了@ARGV的第一个元素的污点，从中得到文件名。我们用字符的类别来准确地指明需要的内容。在这个例子中，我们只需要字母、数字、下划线、点和连接符。我们不需要任何目录分隔符：

```
my( $file ) = $ARGV[0] =~ m/^([A-Z0-9_.-]+)$/ig;
```

注意，我还限定了正则表达式必须匹配整个字符串。就是说，如果字符串中有任何字符不在我们指定的字符类别中，匹配就会失败。我不会试图把不好的数据变成好的数据。你须要考虑一下在各种情况下该如何处理。

这个功能也非常容易被误用。很多人被严格的污点检测弄得很恼火，于是试图绕过它。用一个非常简单的能匹配任何东西的正则表达式就可以去除数据的污点：

```
my( $file ) = $ARGV[0] =~ m/(.*)/i;
```

要是像这样做，我压根就不会用污点检测。如果要求程序员使用污点检测的话，你就要多

加注意，他们可能会为了省事而像上面这样干。我在很多次代码审查中遇到过这样的问题。让我惊讶的是，从来没有人提出异议。

不过，即使我们更加努力也有可能出错。字符类别的快捷方式：\w 和\W（以及 POSIX 版本的[:alpha:]），其实是和区域设置有关的。一个聪明的黑客，可以通过操纵区域设置来得到他所需要的危险的字符。我们应该明确地指明想要的字符，而不是使用快捷方式隐式提供的范围。无论怎样小心也不为过。列出允许的字符之后再添加遗漏的要比列出不允许的安全得多，因为前者排除了我们还不知道的有问题的字符。

如果关掉区域设置，这就不成问题了，我们就可以用字符类别的快捷方式。Perl 会使用内部的区域设置，而不是用户设置的（对于正则表达式则是从 LC_CTYPE 中得到的）。关掉区域设置后，\w 只包含 ASCII 字母、数字和下划线：

```
{
no locale;

my( $file ) = $ARGV[0] =~ m/^([\w.-]+)$/;
}
```

Mark Jason Dominus 在他的一次 Perl 课程上提到了两种用正则表达式去除数据的污点的方法，他称它们为 Prussian Stance 和 American Stance（注 2）。Prussian Stance 明确地列出了允许的字符。我们知道这些字符都是安全的：

```
# Prussian = safer
my( $file ) = $ARGV[0] =~ m/([a-z0-9_.-]+)/i;
```

American Stance 就没有那么可靠了。它是在要排除的字符类别中列出不允许的字符，只要漏掉了一个就会有问题。和只允许安全的输入字符的 Prussian Stance 不一样，这种方法要求我们知道所有有问题的字符。我怎么知道自己知道了所有的呢？

```
# American = uncertainty
my( $file ) = $ARGV[0] =~ m/([^$%;|]+)/i;
```

我倾向于使用一种更加严格的方案——不从输入中提取部分内容。如果输入中有一些是不安全的，那么整个就是不安全的。我们把安全的字符类锚定在了字符串的开始和结尾。我们没有用锚定符$，因为它允许结尾有换行符：

```
# Prussian = safer
my( $file ) = $ARGV[0] =~ m/^([a-z0-9_.-]+)\z/i;
```

有些情况下，我们不希望正则表达式去除数据的污点。即使按照希望的方式匹配上了数据，我们可能也不希望那些数据跑到程序外面。这时，我们可以用编译指令 re 关掉正则表达式记忆变量的去污点功能。一种方式是：

注 2：也有把这个称为“白名单”和“黑名单”的。

```
{
use re 'taint';

# $file依然是有污点的
my( $file ) = $ARGV[0] =~ m/^([\w.-]+)$/;
}
```

另一种更有用和更安全的方式是在全局中关掉正则表达式的去污点功能，只在须要用到的地方打开。因为我们只在真正希望去除数据污点的地方用到它，所以这样更加安全：

```
use re 'taint';

{
no re 'taint';

# $file没有污点
my( $file ) = $ARGV[0] =~ m/^([\w.-]+)$/;
}
```

IO:Handle::untaint

模块 IO::Handle 在很多地方是行输入操作符的基础，它会帮助我们去除数据的污点。来自文件的输入也是外部数据，在污点检测时它通常被标记为有污点的：

```
use Scalar::Util qw(tainted);

open my($fh), $ARGV[0] or die "Could not open myself! $!";

my $line = <$fh>;

print "Line is tainted!\n" if tainted( $line );
```

我们可以告诉 IO::Handle 信任来自某个文件的数据。就像前面多次提到的，这不表示我们是安全的。它只意味着 Perl 不会把数据标记为有污点的，不代表数据是安全的。不过，我们须要显式地使用 IO::Handle 模块才能奏效：

```
use IO::Handle;
use Scalar::Util qw(tainted);

open my($fh), $ARGV[0] or die "Could not open myself! $!";

$fh->untaint;

my $line = <$fh>;

print "Line is not tainted!\n" unless tainted( $line );
```

这可能是一个危险的操作，因为我们用类似于正则表达式/(.*)/的方法绕过了污点检测。

哈希表的键
Hash Keys

你不应该这样做，但是作为 Perl 大师（或者是在问答竞赛上），当别人告诉你去除数据污点的唯一方式是用正则表达式时，你可以告诉他错了。你不应该做我将要展示的事情，但这是你应该知道的，这样当有人试图这样做的时候就可以阻止他了。

哈希表的键不是完整的标量，它们没有提供足够的空间和审计信息来让 Perl 保存污点检测的信息。如果我们把数据传入一个特别的过滤器中，该过滤器把数据作为哈希表的键并返回哈希表的键，得到的输出就不再是有污点的了，无论数据的来源或内容是什么：

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

use Scalar::Util qw(tainted);

print "The first argument is tainted\n"
        if tainted( $ARGV[0] );

@ARGV = keys %{ { map { $_, 1 } @ARGV } };

print "The first argument isn't tainted anymore\n"
        unless tainted( $ARGV[0] );
```

不要这样做。本来想把前面这句话变成全部大写的，不过我知道编辑们不会允许这样做。所以我只好再重复一遍：不要这样做。把这个技巧留到 Perl 问答竞赛上吧，或者在你把本书借给同事之前撕掉这一页。

用有污点的数据选择无污点的数据
Choosing Untainted Data with Tainted Data

污点检测规则的另一个例外是三元操作符。之前我说过有污点的数据也会污染表达式的结果。对于三元操作符来说，如果有污点的数据只是在条件表达式中用于判断应该取哪个值，那么之前所说的就不成立了。只要备选的两个值没有污点，结果就没有污点：

```
my $value = $tainted_scalar ? "Fred" : "Barney";
```

在这个例子中，`$value` 没有污点，因为三元操作符其实只是 `if-else` 块的缩写，而有污点的数据不在和`$value` 有联系的表达式中。有污点的数据只出现在条件表达式中：

```
my $value = do {
        if( $tainted_scalar ) { "Fred"   }
        else                  { "Barney" }
        };
```

system 和 exec 命令的列表形式

List Forms of system and exec

如果调用 system 或 exec 时只用一个参数，Perl 会在参数中找到 shell 元字符（metacharacters）。如果 Perl 找到了元字符，它会把参数传给相应的 shell 程序处理。了解了这一点，我们就可以构造一个 shell 命令来做一些程序本来不希望做的事情。比如说，我有一个看起来无害的 system 调用，它会调用 echo 程序：

```
system( "/bin/echo $message" );
```

作为这个程序的使用者，我可以构造输入的内容让$message 做的事情不仅仅是为 echo 提供一个参数。下面的这个字符串用分号终止了 echo 命令，然后启动 mail 命令并对输入进行重定向：

```
'Hello World!'; mail joe@example.com < /etc/passwd
```

污点检测可以捕获这个问题，不过它还是依赖于我来正确地去除数据的污点。就像前面展示过的那样，我们不能依赖污点检测来保证安全。我们可以使用列表模式的 system 和 exec 命令。这样，Perl 会把第一个参数作为程序名，直接调用 execvp，从而避免 shell 做任何可能的替换或解释操作：

```
system "/bin/echo", $message;
```

在 system 命令中使用数组不会自动触发列表模式。如果数据只有一个元素，system 就只会得到一个参数。如果 system 在唯一的一个参数中看到任何 shell 的元字符，它就会把整个命令行（特殊字符和所有的内容）传给 shell：

```
@args = ( "/bin/echo $message" );
system @args; # single argument form still, might go to shell

@args = ( "/bin/echo", $message );
system @args; # list form, which is fine.
```

为了处理这种特殊情况，我们可以在这两个函数中使用间接对象表示法（indirect object notation）。Perl 会把间接对象作为要调用的程序的名字处理，并且把参数解析成一个列表，即使它只有一个元素。这个例子看起来好像包含了$arg[0]两次，其实并没有。这只是一个特殊的间接对象表示法，它会打开列表处理模式，并假设第一个参数是命令的名字（注 3）：

```
system { $args[0] } @args;
```

在这种表示法下，如果@args 只有一个元素("/bin/echo 'Hello'")，system 会认为要执行的命令是整个字符串。当然，执行会失败，因为没有/bin/echo 'Hello'这个命令。我们须要修改程序的某个地方来保证它们会作为单独的元素出现在@args 中。

注 3：system 命令的间接对象标记的说明文档其实是在 *perlfunc* 手册对 exec 命令的说明中。

如果想要更加安全，我们也可以把允许 system 命令执行的程序保存在一个哈希表中。如果程序不在哈希表中，就不执行它：

```
if( exists $Allowed_programs{ $args[0] } )
        {
        system { $args[0] } @args;
        }
else
        {
        warn qq|"$args[0]" is not an allowed program|;
        }
```

3 个参数的 open 函数

Three-Argument open

从 Perl5.6 版开始，内置的 open 函数有一种 3 个（或更多）参数的形式，这种形式把访问模式从文件名中分离了出来。前面用 open 函数的例子出了问题就是因为文件名字符串中也指定了 open 函数处理文件的方式。如果能够篡改文件名，就可以欺骗 open 函数做一些程序员本来不希望做的事情。在三参数模式中，$file 中的字符都会出现在文件名中，即使有|、>，等等：

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

my( $file ) = $ARGV[0] =~ m/([A-Z0-9_.-]+)/gi;

open my( $fh ), ">>", $file or die "Could not open for append: $file";
```

这种方法并没有绕过污点检测，不过它安全多了。在《Intermediate Perl》的第 8 章和 *perlopentut* 中可以找到关于 open 函数的这种形式的更详细的讨论。

sysopen

函数 sysopen 给我们提供了对文件访问的更多的控制方式。它的 3 个参数的形式把访问模式和文件名分离了开来，并提供了一种可以精确地配置访问控制的 exotic 模式。比如说，open 的 append 模式会在文件不存在的时候新建一个文件。在 sysopen 中，这是两个独立的选项：一个是追加，一个是新建：

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

use Fcntl (:DEFAULT);

my( $file ) = $ARGV[0] =~ m/([A-Z0-9_.-]+)/gi;

sysopen( my( $fh ), $file, O_APPEND|O_CREAT )
        or die "Could not open file: $!\n";
```

它们是独立的选项，所以可以分开使用。如果不希望创建新的文件，可以去掉 O_CREAT。这样如果文件不存在，Perl 就不会创建它，别人也就没法欺骗我的程序创建一个文件来攻击系统了：

```
#!/usr/bin/perl

use Fcntl qw(:DEFAULT);

my( $file ) = $ARGV[0] =~ m/([A-Z0-9_.-]+)/gi;

sysopen( my( $fh ), $file, O_APPEND )
        or die "Could not append to file: $!";
```

限制特权
Limit Special Privileges

因为当我们以其他用户的身份运行程序时，Perl 会自动打开污点检测，所以应该限制使用特权的范围。我们可以 `fork` 一个进程来专门处理需要更高的权限的部分，或者在不再需要特权的时候放弃特权。可以把真实用户和有效用户都设置成真实用户，这样我们就放弃了不需要的特权。我们可以利用 `POSIX` 模块这样做：

```
use POSIX qw(setuid);

setuid( $<, $< );
```

当然也有其他的方法，不过它们超过了本章的范围（甚至超过了本书的范围），而且它们依赖于你的操作系统。在其他的语言中，你也可以这样做。这不是 Perl 特有的问题，因此处理的方法和其他语言一样：隔离出需要特权访问的部分。

总结
Summary

Perl 知道不明智地到处传递数据会招来很多问题，所以给我们——程序员提供了处理这个问题的方法。污点检测是一个工具，能够帮助我们找到程序中试图把外部数据传递到程序外面的部分。Perl 帮助我们检测数据，并在使用之前把它们变成可以信任的数据。检测和过滤数据并不是答案的全部，我们还须要防御性地利用 Perl 提供的其他安全特性。即使这样，污点检测也不能保证我们完全安全，我们还是须要仔细地考虑整体的安全状况，和使用其他任何的编程语言一样。

深入阅读
Further Reading

进一步的阅读可以从 *perlsec* 文档开始，该文档提供了一个 Perl 安全编程技术的概览。

文档 *perltaint* 提供了污点检测的全部细节。文档 *perlfunc* 中关于 `system` 和 `exec` 的部分介绍了它们的安全特性。

文档 *perlfunc* 介绍了内置的 `open` 函数的所有功能，而文档 *perlopentut* 中则包含了更多的内容。

虽然是针对网络应用的，Open Web Application Security 项目（OWASP，*http://www.owasp.org*）为所有类型的应用给出了大量很好的建议。

即使你不愿意阅读计算机应急响应组（CERT，*http://www.cert.org*）和安全焦点（*http://securityfocus.com/*）的所有警告信息，读一些它们中关于 `perl` 解析器和程序的建议往往也是很有帮助的。

第4章 调试Perl程序
Debugging Perl

Perl标准发布版带有一个调试器——perl5db.pl，虽然它只是一个Perl程序。不过，正因为它只是一个程序，我们可以以它为基础写一个自己的调试器来满足需求，也可以用perl5db.pl提供的接口来配置它的行为。不过，这仅仅只是一个开始。我们还可以写自己的调试器，或者从其他的Perl大师写的众多调试器中选择一个使用。

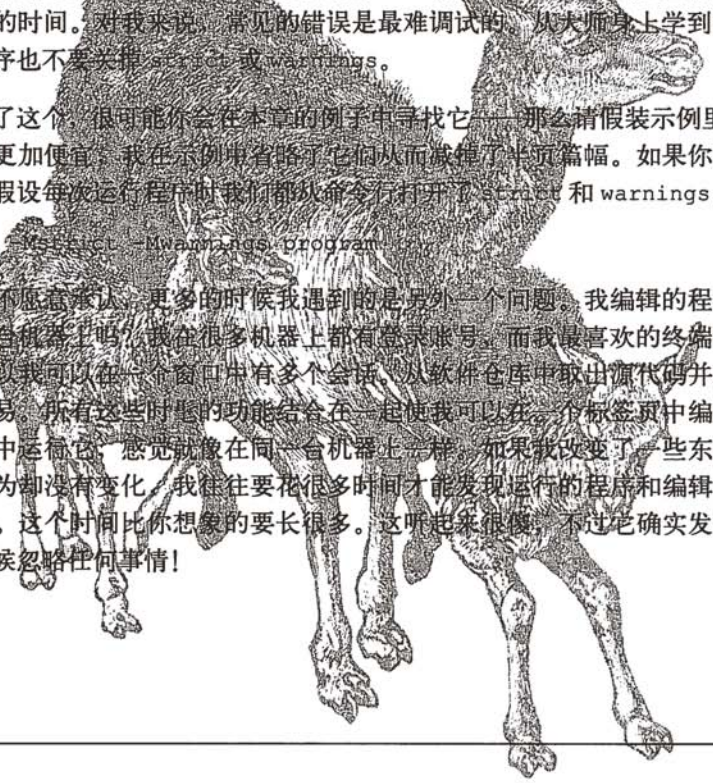
避免浪费太多的时间
Before You waste Too Much Time

首先，我要给你们介绍一下Perl提供的两个非常有用的调试助手：strict和warnings。我的大部分麻烦都是小程序带来的。对于小程序，我觉得不需要strict，结果犯了很多strict模式能够发现的愚蠢的错误。我在跟踪Perl能够立刻发现的问题上花的时间大大超过了应该花的时间。对我来说，常见的错误是最难调试的。从大师身上学到的经验：即使是很小的程序也不要关掉strict或warnings。

既然我介绍了这个，很可能你会在本章的例子中寻找它——那么请假装示例里有这些代码。为了让本书更加便宜，我在示例中省略了它们从而减掉了半页篇幅。如果你不喜欢这个想法，也可以假设每次运行程序时我们都从命令行打开了strict和warnings：

```
$ perl -Mstrict -Mwarnings program.pl
```

此外，尽管不愿意承认，更多的时候我遇到的是另外一个问题。我编辑的程序和运行的程序是在同一台机器上吗？我在很多机器上都有登录账号，而我最喜欢的终端程序是支持标签页的，所以我可以在一个窗口中有多个会话。从软件仓库中取出源代码并在任意地点工作也非常容易。所有这些时髦的功能结合在一起使我可以在一个标签页中编辑一个文件，而在另一个中运行它，感觉就像在同一台机器上一样。如果我改变了一些东西而程序的输出或它的行为却没有变化，我往往要花很多时间才能发现运行的程序和编辑的程序不在同一台机器上。这个时间比你想象的要长很多。这听起来很傻，不过它确实发生了。请不要在调试的时候忽略任何事情！



这是一个比较有趣的故事。我希望通过它来说明一点：在调试的时候，谦卑是维护代码的程序员（注 1）的一个主要的美德。调试时最好的赌注是假设问题是自己造成的。这样做，就不会把任何因素排除在外，或者试图怪罪其他的东西——我经常在各种 Perl 讨论区中看到标题为“Perl 的可能的问题”的帖子。事实证明，当首先怀疑自己时，我往往是对的。附录 B 是我的一个解决问题的指导手册，即使不能解决问题，至少它能够帮助人们找出问题所在。

世界上最好的调试器
The Best Debugger in the World

不管我用过多少不同的调试器或集成开发环境，我仍然认为简明的 print 是最好的调试器。我可以把源代码载入调试器，设置好输入和断点，观察发生的事情；更多的时候会插入几个 print，直接像通常那样运行程序（注 2）。我在变量周围加上了括号，以便于发现任何的空白字符：

```
print "The value of var before is [$var]\n";

#... 会改变$var 的一些操作

print "The value of var after is [$var]\n";
```

其实也可以不用 print。我们可以用 warn 做同样的事情，它会把输出发送到标准错误输出中：

```
warn "The value of var before is [$var]";

#... 会改变$var 的一些操作

warn "The value of var after is [$var]";
```

由于没有在 warn 消息末尾使用换行符，输出会告诉我们 warn 所在的文件名和行号：

```
The value of var before is [$var] at program.pl line 123.
```

对于复杂的数据结构，我们可以用 Data::Dumper 来显示它。它能够很好地处理哈希表和数组。在这里，我们用不同的字符——尖括号来分隔来自 Data::Dumper 的输出：

```
use Data::Dumper qw(Dumper);
warn "The value of the hash is <\n" . Dumper( \%hash ) . "\n>\n";
```

注 1：Larry Wall 说懒惰、没有耐心和傲慢是程序员的主要美德。不过这些只对写代码的程序员适用。处于软件开发周期的其他环节的所有人员需要的是老练、谦卑和低血压。

注 2：在第 10 章中，我们将会介绍如何处理第三方模块：把源文件拷贝到一个私有目录中，并把该目录添加到数组@INC 的前面。这样我们就可以编辑副本而不用担心破坏原始版本。

这些 warn 语句显示出了 warn 语句的行号。其实这不是很有用——我们已经知道 warn 在哪里了，是我们把它放在那里的！我们真正希望知道的是当出问题的时候它是从哪里调用的。考虑下面这个返回两个数字的商的子程序 divide。由于某种原因，代码的某个地方调用该子程序时导致了除零操作（注 3）：

```
sub divide
        {
        my( $numerator, $denominator ) = @_;

        return $numerator / $denominator;
        }
```

Perl 会非常清楚地告诉我们代码的什么位置出了问题：

```
Illegal division by zero at program.pl line 123.
```

我们可以在子程序中加入一些调试代码，使用 warn 检查参数的值：

```
sub divide
        {
        my( $numerator, $denominator ) = @_;
        warn "N: [$numerator] D: [$denominator]";

        return $numerator / $denominator;
        }
```

代码的很多地方可能会调用 divide，所以我们真正想知道的是哪个调用出了问题。上面的这个 warn 语句除了显示出参数之外并没有多大帮助。

尽管把 print 称为世界上最好的调试工具，我真正使用的是 print 的另一种形式——Perl 标准发布版的 Carp 模块的 carp 函数。它和 warn 类似，不过它能够报告调用子程序的代码的文件名和行号：

```
#!/usr/bin/perl
use Carp qw(carp);

printf "%.2f\n", divide( 3, 4 );
printf "%.2f\n", divide( 1, 0 );
printf "%.2f\n", divide( 5, 4 );

sub divide
        {
        my( $numerator, $denominator ) = @_;
        carp "N: [$numerator] D: [$denominator]";

        return $numerator / $denominator;
        }
```

注 3：我知道应该用 eval 包住表达式，不过在这里我需要一个有错误的例子。即使我把它放在了 eval 中，我也须要找出造成错误的罪魁祸首。

这次的输出信息变得有用多了。我们不仅得到了错误信息，而且还得到了调用的代码行的信息及调用子程序时的参数。可以看到第 4 行的调用没有问题，而第 5 行的调用是 Perl 结束程序之前执行的最后一个调用：

```
$ perl show-args.pl
N: [3] D: [4] at show-args.pl line 11
                main::divide(3, 4) called at show-args.pl line 4

0.75
N: [1] D: [0] at show-args.pl line 11
                main::divide(1, 0) called at show-args.pl line 5
Illegal division by zero at show-args.pl line 13.
```

函数 carp 比 warn 提供了更详尽的输出。如果想用 die 做同样的事情，我们可以用 croak 函数。它能够提供和 carp 一样详尽的输出，不过 croak 会和 die 一样终止程序。

随心所欲
Doing Whatever I Want

我们可以通过干预%SIG 来改变 warn 和 die 函数。我喜欢用它们来分析代码，不过从来不用它们为代码添加新的功能。它们只是我的调试工具箱的一部分。

伪键__WARN__和__DIE__指向的函数会在调用 warn 和 die 时运行。我们可以使用具名或匿名子程序的引用指定要运行的函数：

```
$SIG{__DIE__} = \&my_die_handler;
$SIG{__DIE__} = sub { print "I'm about to die!" }
```

利用这个功能，无须遍历所有的代码，就可以把所有的 die 调用变成提示信息更丰富的 croak 调用（注 4）。在下面的例子中，我们在子程序调用前加上了&并去掉括号，这样 Perl 会把当前的参数列表传递给下一个子程序，也就是 croak：

```
use Carp;
$SIG{__DIE__} = sub { &Carp::croak };

die "I'm going now!";  # really calls croak now
```

如果只希望对部分代码使用这个功能，可以用 local 限定作用的范围（因为%SIG 是一个永远在 main::中的特殊变量）。我们的重定义在跳出作用域之前会一直有效：

```
local $SIG{__DIE__} = sub { &Carp::croak };
```

注 4：它会改变所有的调用。如果我们正在使用 mod_perl，那么所有的程序都会被改变，因为它们共享同样的全局变量。这真是糟糕。

运行修改后的程序时，这两个函数会执行相应的操作：warn 会让程序继续运行，而 die 会进行异常处理并最终终止程序（注 5）。

因为 croak 会显示出堆栈的每一层，而我们是从一个匿名的子程序中调用它的，所以会在输出中看到一个虚构的子程序名：

```
use Carp;

print "Starting program...\n";

$SIG{__DIE__} = sub {
        local $Carp::CarpLevel = 0;

        &Carp::croak;
        };


foo(); # program dies here

sub foo { bar() }

sub bar { die "Dying from bar!\n"; }
```

在堆栈踪迹中，可以看到 bar()和 foo()之后有一个__ANON__调用：

```
Starting program...
Dying from bar!
 at die.pl line 12
                main::__ANON__('Dying from bar!\x{a}') called at die.pl line 20
                main::bar() called at die.pl line 18
                main::foo() called at die.pl line 16
```

下面我们修改一下匿名子程序，调整 croak 开始汇报的位置。我们把$Carp::CarpLevel 的值设定为希望跳过的层次，在这个例子中是 1：

```
$SIG{__DIE__} = sub {
        local $Carp::CarpLevel = 1;

        &Carp::croak;
        };
```

现在，我们就看不到不需要的输出了：

```
Starting program...
Dying from bar!
 at die.pl line 12
                main::bar() called at die.pl line 18
                main::foo() called at die.pl line 16
```

注 5：我们的__DIE__处理程序调用 exit 或 die 时可以绕过 die 的异常处理。处理程序中的 die 不会被重定义。想知道更多的细节请参见 *perlvar* 中关于%SIG 的讨论。

如果你想看一个正在被使用的真实的例子的话，可以看看 CGI::Carp 模块。Lincoln Stein 用%SIG 重新定义了 warn 和 die，把它们变得对 Web 友好。只要载入这个模块，我们就可以得到有用的错误消息而不是讨厌的“服务器错误 500”。载入模块时，CGI::Carp 会设置好$SIG{__WARN__}和$SIG{__DIE__}：

```
use CGI::Carp qw(fatalsToBrowser);
```

函数 fatalsToBrowser 会在结果页面显示出错误信息。CGI::Carp 模块还有一些其他的比较有意思的函数：set_message，它能够捕获编译时的错误；warningsToBrowser，它能够在输出中以 HTML 注释的形式嵌入警告信息。

当然，我并不建议你在产品代码中使用它们。我们不希望用户看到程序的错误。当我们在远程服务器上调试程序的时候，这些错误信息非常方便。一旦找到了问题，我们就不再需要它们了。留下错误信息会让公众了解程序的工作方式，从安全的角度看这是不好的。

跟踪程序
Program Tracing

Carp 模块也提供了 cluck 和 confess 子程序来输出堆栈轨迹（stack traces）。与 warn（或 carp）类似，cluck 可以打印出错误消息，不过它会让程序继续运行。confess 能做同样的事情，不过和 die 一样，它会在打印完一堆乱七八糟的消息（注 6）后停止执行程序。

cluck 和 confess 都能打印出堆栈轨迹，从而显示出子程序调用的列表和它们的参数。每一个子程序调用都会在堆栈上创建一个 *frame* 并把所有的相关信息保存在里面。当子程序结束的时候，Perl 会去掉子程序的 frame，然后再继续处理堆栈中的下一个 frame。相反的，当一个子程序调用另一个子程序的时候，会在堆栈上增加一个 frame。

下面是一个有着一串子程序调用的小程序。我们先调用 do_it 函数，然后它会调用 multiply_and_divide，multiply_and_divide 接下来又会调用 divide。最后，我们没有得到 4 除以 5 的正确答案。对于这个短小的例子，也许你能立刻发现出问题的地方。让我们想象这里有一大堆乱七八糟的参数、子程序调用和其他的问题：

```
#!/usr/bin/perl
use warnings;
use Carp qw(cluck);

print join " ", do_it( 4, 5 ), "\n";

sub do_it
        {
        my( $n, $m ) = @_;

        my $sum = $n + $m;
```

注 6：确实是乱七八糟的。这些函数都会调用 Carp::longmess。一旦看到了输出结果，你就会同意我的说法的。

```
        my( $product, $quotient ) =
              multiply_and_divide( [ $n, $m ], 6, { cat => 'buster' } );

        return ( $sum, $product, $quotient );
        }

sub multiply_and_divide
        {
        my( $n, $m ) = @{$_[0]};

        my $product = $n * $m;
        my $quotient = divide( $n, $n );

        return ( $product, $quotient );
        }

sub divide
        {
        my( $n, $m ) = @_;
        my $quotient = $n / $m;
        }
```

我们怀疑问题出在 divide 子程序中。但是我们也知道它在子程序调用链的最末端，因此我们须要检查调用 divide 的路径，也就是说须要得到堆栈轨迹。下面我们修改 divide 以利用 cluck——Carp 的 warn 版本的堆栈追踪功能。为了让结果更加清晰易读，我们在 cluck() 前后分别加入了一行连字符来分隔输出结果：

```
sub divide
        {
        print "-" x 73, "\n";
        cluck();
        print "-" x 73, "\n";

        my( $n, $m ) = @_;

        my $quotient = $n / $m;
        }
```

输出结果给出了子程序调用的列表，其中最近的子程序调用在前面（就是说列表和堆栈的顺序是一致的）。堆栈踪迹中显示了软件包的名字、子程序名和参数。观察一下 divide 的参数，我们会发现一个重复的 4。其中的一个参数应该是 5。这压根不是 divide 的问题：

```
---------------------------------------------------------------------------
 at confess.pl line 68
---------------------------------------------------------------------------
            main::divide(4, 4) called at confess.pl line 60
          main::multiply_and_divide('ARRAY(0x180064c)') called at confess.pl line 49
            main::do_it(4, 5) called at confess.pl line 41
9 20 1
```

这不是 divide 函数的问题，而是传给 divide 的参数的问题。参数来自 multiply_and_divide。在调用 divide 的地方，同样的参数我们传了两次。如果戴上眼镜的话，我可能会注意到$n 和$m 很像，但是它们不同：

```
my $quotient = divide( $n, $n );  # WRONG

my $quotient = divide( $n, $m );  # SHOULD BE LIKE THIS
```

这只是一个简单的例子，不过 Carp 处理它还是有些问题。在 multiply_and_divide 的参数列表中，我们得到的只是'ARRAY(0x180064c)'。这并没有多大的帮助。幸运的是，我知道如何定制模块（第 9 章和第 10 章）。通过研究 Carp 的代码，我们发现 Carp::Heavy 可以对参数进行格式化。该子程序中有一个分支可以处理引用：

```
package Carp;
# This is in Carp/Heavy.pm

sub format_arg {
  my $arg = shift;

        ...

  elsif (ref($arg)) {
         $arg = defined($overload::VERSION) ? overload::StrVal($arg) : "$arg";
  }
        ...

  return $arg;
}
```

当 format_arg 看到一个引用时，它会检查 overload 模块是否存在。overload 模块可以用来定义我们自己的 Perl 操作，包括把变量变成字符串。如果 Carp 发现我们已经加载了 overload，就会调用子程序 overload::StrVal 把引用变成可以阅读的东西。如果没有加载 overload，它就会简单地解析双引号中的引用，从而得到前面见到的 ARRAY(0x180064c) 之类的东西。

不过，函数 format_arg 有点过于简单了。有可能我们在一个软件包中使用了 overload 模块，但是在另一个软件包中没有用。简单地检查是否在程序的某个地方使用了并不代表它对所有的引用都有效。此外，我们可能忘记了调用它来把引用转化成字符串。最后，我们不能在一个很长的堆栈轨迹中对所有的对象和引用使用 overload 模块，尤其是在大部分的模块都不是自己写的的时候。我们需要一个更好的办法。

我们可以覆盖 Carp 的 format_arg 函数来做须要做的事情。为了能够随心所欲地修改代码，我把已有的代码拷贝到 BEGIN 块中。首先，我们加载原来的源文件，Carp::Heavy，这样就载入了原来的定义。通过赋值给 typeglob，我们替换了子程序的定义。如果子程序的参数是引用，我们就载入 Data::Dumper，并设置一些 Dumper 的参数来调整输出的格式，然后得到参数的字符串形式：

```
BEGIN {
use Carp::Heavy;

no warnings 'redefine';
*Carp::format_arg = sub {
        package Carp;
        my $arg = shift;

        if( not defined $arg )
                { $arg = 'undef' }
        elsif( ref $arg )
                {
                use Data::Dumper;
                local $Data::Dumper::Indent = 0; # salt to taste
                local $Data::Dumper::Terse  = 0;
                $arg = Dumper( $arg );
                $arg =~ s/^\$VAR\d+\s*=\s*//;
                $arg =~ s/;\s*$//;
                }
        else
                {
                $arg =~ s/'/\\'/g;
                $arg = str_len_trim($arg, $MaxArgLen);
                $arg = "'$arg'" unless $arg =~ /^-?[\d.]+\z/;
                }

        $arg =~ s/([[:cntrl:]]|[[:^ascii:]])/sprintf("\\x{%x}",ord($1))/eg;
        return $arg;
        };
}
```

对于 Dumper 的输出，我们还做了一些额外的工作。Dumper 返回的结果是能够在 eval 中使用的。它其实是一个 Perl 表达式，包含了对标量的赋值操作和结尾的分号。因此我们用一些替换操作去掉了它们。我还希望去掉 Data::Dumper 添加在末尾的东西：

```
$VAR = ... ; # leave just the ...
```

现在，运行同样的程序，我们得到的输出要清晰很多。我们可以看到传给 multiply_and_divide 的匿名数组中的元素：

```
-------------------------------------------------------------------------
 at confess.pl line 65
                main::divide(4, 4) called at confess.pl line 57
                main::multiply_and_divide([4,5]) called at confess.pl line 46
                main::do_it(4, 5) called at confess.pl line 38
-------------------------------------------------------------------------
9 20 1
```

当然，这种方法最棒的地方在于，我们只须要在一个子程序中加入 cluck 就可以得到所有的这些信息。我曾经在有很多参数和复杂的数据结构的情况下用过它，得到的是一个 Perl 风格的堆栈转储（stack dump）。分析它也许很困难，但是打开或关掉它是非常容易的。

安全地修改模块
Safely Changing Modules

在前一节中，我们修改&Carp::format_arg来做了一点不同的事情。这个基本的想法对于调试来说非常有用。我们要找的 bug 不仅在自己写的代码中，更多的时候是在用的模块中或是别人写的代码中。

当我们要调试其他文件中的程序时，须要加入一些调试语句或修改代码来观察程序的执行过程。可是，我不希望修改源代码。因为每次这样做的时候都会把事情弄得更糟，不管我是多么小心地试图把代码恢复成原来的状态。不管做什么，我希望没有任何的危害，也不会影响任何人。

一个简单的方法是：把有问题的模块的文件拷贝到一个新的地方。我专门为调试建立了一个特殊的目录，这样修改过的模块就不会影响任何其他的东西。然后修改环境变量PERL5LIB让 Perl 首先找到修改的版本。我们完成调试后，再恢复PERL5LIB以使用原有的版本。

举个例子，最近我须要检查一下Net::SMTP的内部工作方式，因为我认为它没有正确地处理套接字。我选择了一个目录来存放我的拷贝，在这个例子中是*~/my_debug_lib*，并设置PERL5LIB指向它。然后创建须要存放修改的版本的目录，并拷贝模块到该目录：

```
$ export PERL5LIB=~/my_debug_lib
$ mkdir -p ~/my_debug_lib/Net/
$ cp `perldoc -l Net::SMTP` ~/my_debug_lib/Net/.
```

现在，我们就可以修改*~/my_debug_lib/Net/SMTP.pm*，运行代码来观察程序的执行过程，进而找到解决方案了。所有的这些都不会影响任何人。对这个模块，我可以做本章中介绍的所有事情，包括在合适的位置插入confess语句来得到调用栈的 dump。每次须要研究一个新的模块的时候，我就把它拷贝到我的临时调试目录中。

封装子程序
Wrapping Subroutines

其实，不用把模块拷贝一份也可以修改它的行为。我们可以直接在自己的代码中覆盖它的部分内容。Damian Conway 写了一个奇妙的Hook::LexWrap模块。这个模块可以给一个子程序加上一个外壳变成另一个子程序。这意味着我们的外壳子程序可以看到传入的参数和传出的结果。从而我们可以在希望的时候检查甚至修改它们。

我们可以从一个简单的例子程序开始。这个程序把一些数加了起来。和前面一样，它有一些问题，因为我传错了参数。我没有区分出$n和$m，在调用add时使用了$n两次。运行这个程序会得到错误的答案，不过我们还不知道问题在哪里：

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

```
# @ARGV = qw( 5 6 );

my $n = shift @ARGV;
my $m = $ARGV[0];

print "The sum of $n and $m is " . add( $n, $n ) . "\n";

sub add
        {
        my( $n, $m ) = @_;

        my $sum = $n + $m;

        return $sum;
        }
```

我不想改变任何代码，或者应该说，希望在不影响已有的语句的条件下看看发生了什么事情。和前面一样，我希望调试结束的时候能够把所有的东西恢复原样。不修改子程序的话会容易得多。

模块 Hook::LexWrap 让我们有机会在调用子程序之前和子程序返回之后做一些事情。正如它的名字所暗示的，它用另一个子程序封装住要研究的子程序来实现它的魔法。函数 Hook::LexWrap::wrap 有 3 个参数：须要封装的子程序名，这里是 add；分别用来做预处理和后处理的两个匿名子程序。

```
#!/usr/bin/perl

use Hook::LexWrap qw(wrap);

my $n = shift @ARGV;
my $m = $ARGV[0];

wrap add,
        pre  => sub { print "I got the arguments: [@_]\n" },
        post => sub { print "The return value is going to be $_[-1]\n" }
        ;

# this line has the error
print "The sum of $n and $m is " . add( $n, $n ) . "\n";

sub add
        {
        my( $n, $m ) = @_;

        my $sum = $n + $m;

        return $sum;
        }
```

预处理程序得到的参数列表和 add 的一样。在这个例子里，我们可以输出参数列表来看看它的值是什么。后处理程序也会得到同样的参数列表，不过 Hook::Lex::Wrap 在@_的最后

添加了另外的一个元素——返回值。在后处理程序中，$_[-1]永远是返回值。现在，我们的程序能够输出一些有用的调试信息了。可以看到，我把同样的参数传了两次：

```
$ perl add_numbers.pl 5 6
I got the arguments: [5 5 ]
The return value is going to be 10
The sum of 5 and 6 is 10
```

注意输出中最后的 5 后面的空格。wrap 会在@_中添加一个元素，即使它是 undef。当我们解析双引号中的数组时，返回值和之前的 5 之间就有了一个空格。

Hook::LexWrap 还能够处理调用的所有上下文。它可以处理标量、列表和空上下文。在列表上下文中，后期处理程序的@_的最后一个元素是数组的引用。在空上下文中，它什么也不做。

不过，它能做的事情还有很多。Hook::LexWrap 会在做任何事情之前往@_中加入额外的元素。如果仔细观察上面的输出的话，会发现第二个元素之后在第二个 5 和反方括号之间有一个空格。这是 5 和@_中的额外元素 undef 之间的空格。

在预处理程序中，我可以给返回值赋值，从而告诉 Hook::LexWrap 已经有了返回值。这样它就不会运行原来的子程序了。如果子程序做的事情和所期望的不一样，我可以强制它返回正确的结果：

```
#!/usr/bin/perl

use Hook::LexWrap;

my $n = shift @ARGV;
my $m = $ARGV[0];

{

wrap add,
        pre  => sub {
                print "I got the arguments: [@_]\n";
                $_[-1] = "11";
                },
        post => sub { print "The return value is going to be $_[-1]\n" }
        ;
print "The sum of $n and $m is " . add( $n, $m ) . "\n";
}

sub add
       {
       my( $n, $m ) = @_;

       my $sum = $n + $m;

       return $sum;
       }
```

在预处理程序中给$_[-1]赋值之后，输出结果就不一样了。子程序和后处理程序都没有运行，我们得到了11：

```
$ perl add_numbers.pl 5 6
I got the arguments: [5 6 ]
The sum of 5 and 6 is 11
```


利用伪造的返回值，我们可以得到正确的结果，从而在不修改程序的条件下继续分析程序。当我们在分析一个很多地方都有问题的大程序的时候，这个工具就会变得特别方便。我们知道某个子程序须要返回什么值，于是就先让它返回正确的值。这样就可以研究程序的其他部分。有的时候去掉一个错误，即使是临时性的，也会使其他问题变得更容易解决。

perl5db.pl

为了检查复杂的数据结构，我们在《Intermediate Perl》中介绍了Perl的标准调试器。手册*perldebug*中对它有详细的介绍，Richard Foley甚至为它专门写了一本书——《Pro Perl Debugging》(Appress)。在本章中，我们只讨论一些须要用到的基础内容，然后就介绍那些更精巧的调试器。

通过-d选项可以启动Perl调试器：

```
perl -d add_number.pl 5 6
```

Perl会编译好程序，停止在要运行的语句之前，并给出提示。调试器会显示出程序名、行号和下一个要执行的语句：

```
Loading DB routines from perl5db.pl version 1.25
Editor support available.

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

main::(Scripts/debugging/add_numbers.pl:3):
3:      my $n = shift @ARGV;
  D<1>
```

从这里开始，我们就可以进行常见的调试操作了，比如单步执行、设置断点和检查程序状态。

我们也可以启动调试器运行命令行参数-e指定的程序。我们同样也能得到调试器的提示行，不过对于调试程序来说不是非常有用。更有用的是，我们可以在调试器的提示符下尝试Perl语句：

```
$ perl -d -e 0

Loading DB routines from perl5db.pl version 1.25
Editor support available.

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

```
main::(-e:1):   0
  D<1> $n = 1 + 2;

  D<2> x $n
0  3
  D<3>
```

我们在《Intermediate Perl》中介绍过这个调试器，文档 *perldebug* 和很多其他教程中对它也有详细介绍。这里，我们就不再讨论它了。如果需要关于它的更多信息，请参见本章最后一节“深入阅读”中的参考文献。

备选的调试器
Alternative Debuggers

除了标准的 *perl5db.pl* 之外，还有很多其他的调试器可以使用。很多的代码分析工具也会利用这些调试工具。在 CPAN 上，有一个很长的 `Devel::`模块的列表，有可能它们其中的某一个能够满足你的需求。

用-D 启用别的调试器
Using a Different Debugger with -D

给`-d` 开关传递一个参数，我们就可以使用别的调试器。在这个例子中，我希望用 `Devel::Dprof` 模块运行我的程序。`-d` 选项默认使用的是 `Devel::`，所以可以省略掉这一部分。在第 5 章中我会详细地介绍 profilers。

```
$ perl -d:DProf program.pl
```

如果写了自己的调试模块，我们可以像使用`-M` 开关一样把参数传递给该模块。我们还可以在模块名和等号后面加入以逗号分隔的参数列表。在这个例子中，我们加载 `Devel::MyDebugger` 模块并带上参数 `foo` 和 `bar`：

```
$ perl -d:MyDebugger=foo,bar
```

作为普通的 Perl 代码，这和用 `use` 加载 `Devel::MyDebugger` 一样：

```
use Devel::MyDebugger qw( foo bar );
```

Devel::ptkdb

这是一个基于 Tk 的调试器，它为 *perl5db.pl* 的所有功能提供了一个图形化的界面。`Devel::ptkdb` 模块不是 Perl 自带的，所以我们必须自行安装（注 7）。用`-d` 选项指定 `ptkdb` 作为调试器后就可以启动它：

```
$ perl -d:ptkdb program.pl
```

注 7：这意味着我们也须要安装 `Tk` 模块。一旦安装了它，我们也需要用某种窗口管理器来显示它。在我的 Powerbook 上，我用的是苹果公司的 X11 程序（对苹果电脑以外的世界来说，它其实是 Xfree86）。Windows 用户可能须要使用诸如 ReflectionX 之类的程序。

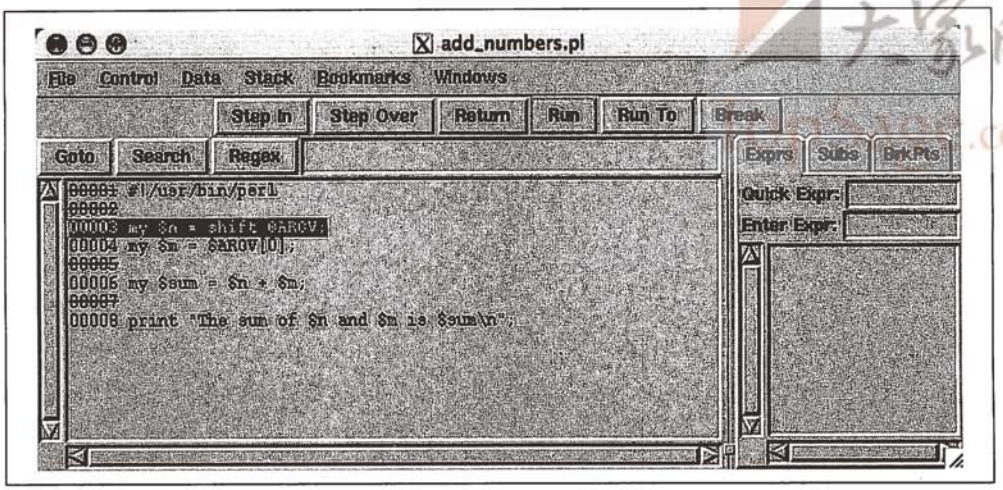


图 4-1：模块 Devel::ptkdb 用 Tk 提供了一个图形化的调试器

ptkdb 启动后会显示一个应用程序的窗口。在窗口的左边，可以看到程序的代码、当前行及它们的行号（图 4-1）。代码上方的按钮可以用来在代码中搜索。右边的标签页可以用来检查表达式、子程序和当前的断点列表。

标签页“Subs”能显示出软件包的层次结构和其中的子程序（图 4-2）。这些都是已经加载的模块。选择须要查看的函数后，可以立刻看到函数的代码。选择函数可以用鼠标双击，或者用方向键选择并用<RETURN>来确定。这些操作不会改变程序的状态。我们可以用“Subs”标签页决定是进入某个函数观察执行过程还是跳过该函数继续运行。

标签页“Exprs”非常有用。它的顶部有两个文本框。“Quick Expr”允许我们输入一个 Perl 表达式，然后它会返回结果并替换掉表达式。如果我在快速表达式中设置或修改了程序的变量，它就会改变程序的状态。这和在终端调试器中尝试一个一次性的表达式是一样的。这个功能不错，不过更好的还是“Enter Expr”。我们输入一个 Perl 表达式，调试器会把它添加到标签页下半部分的表达式的列表中（图 4-3）。运行程序的时候，这些表达式会根据程序的当前状态更新自己的值。我们可以把须要跟踪的变量添加进来，观察它们的变化。

让我们从一个简单的程序开始。这个程序把两个数加到一起。其实，我们并不须要调试它（我希望）。不过，我可以用这个程序来演示“Exprs”标签页的功能。启动调试器后，我们停在了程序的开始，还没有运行任何代码。在“Enter Expr”中加入$m 和$n，单步跟踪走过第一行代码后，就可以看到$m 和$n 的值了。我们也可以输入更复杂的表达式，ptkdb 也会随着代码的执行更新它们的值。

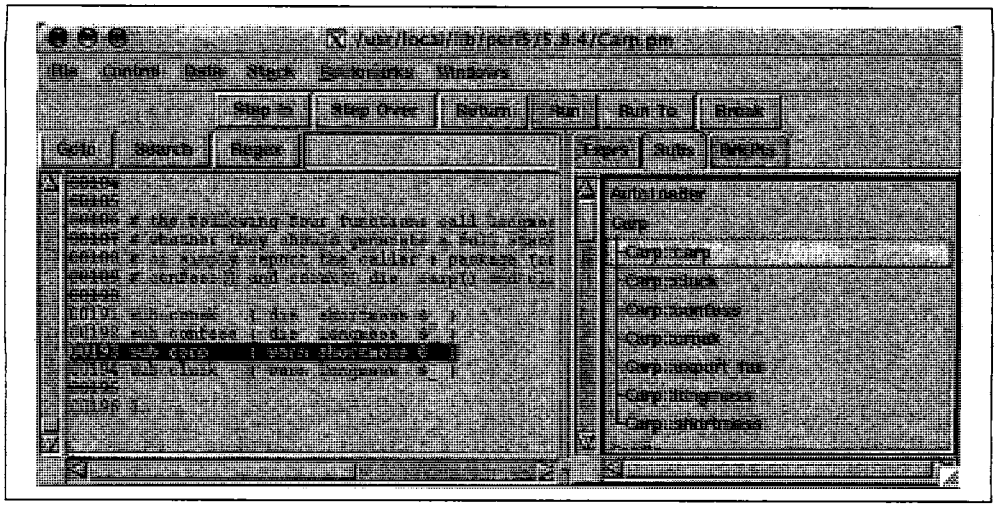


图 4-2：在 Subs 标签页中，可以看到任何已加载的模块中的子程序

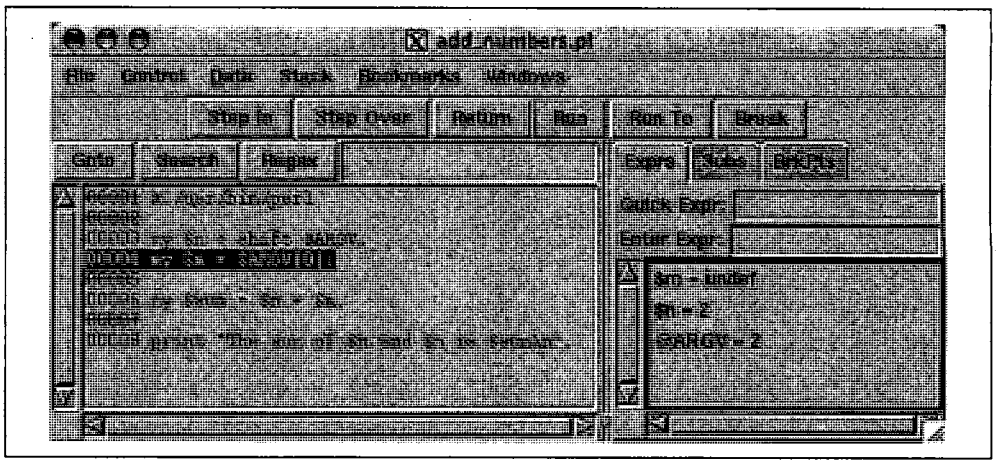


图 4-3：可以在 Expr 标签页中跟踪变量的值

Devel::ebug

Leon Brocard 的 `Devel::ebug` 模块为 Perl 的调试器提供了一个面向对象的接口。该模块还没有完全完成，所以我这里介绍的可能和你读到的有所不同。不过，主要的功能应该没有太大的变化。

它有一个自带的基于命令行的调试器，叫做 `ebug`。这个名字有点奇怪。不过当你注意到你怎么念它的时候，就不会那么奇怪了。它的名字中缺少的 `d` 正好出现在 Perl 的`-d` 开关中：

```
$ perl -d:ebug program.pl
```

不过，也可以不用-d开关。把整个命令行放在引号中（注8）就可以直接用 ebug 运行程序：

```
$ ebug "add_numbers.pl 5 6"
* Welcome to Devel::ebug 0.46
main(add_numbers.pl#3):
my $n = shift @ARGV;
ebug: x @ARGV
--- 5
--- 6

main(add_numbers.pl#3):
my $n = shift @ARGV;
ebug: s
main(add_numbers.pl#4):
my $m = $ARGV[0];
ebug: x $n
--- 5
```

程序 ebug 其实只是 Devel::ebug::Console 的一个外壳。我们可以用多种方式调用 Devel::ebug。它的核心是一个单独运行的进程。程序的后端在调试器下运行程序，前端通过 TCP 和后端通信。这意味着，我们可以在一台机器上调试运行在另一台机器上的程序。

Devel::ebug::HTTP 模块使用了和 Devel::ebug 同样的后端，不过它启动了一个微型的 Web 服务器（注9）。我们可以用和命令行版本同样的方式启动 ebug_http。不过在启动后，它会告诉我们一个能够看到调试器的 URL（注10），而不是命令行提示：

```
$ ebug_http "add_numbers.pl 4 5"
You can connect to your server at http://albook.local:8321
```

网页中给我们显示了一个非常简洁的调试界面（图4-4）。记住，这基本上只是一个概念型的产品。即便如此，它也给人留下了深刻的印象，可以作为你的程序的基础。

注8：调用 Devel::ebug::Console 时，程序会把所有@ARGV 中的元素用空格连接起来。所以如果不加引号的话，会试图运行名为 add_numbers.pl56 的程序，并且没有任何参数。

注9：安装好所有的东西后，一定要把 Devel::ebug::HTTP 安装文件的 *root/* 目录拷贝到 Devel::ebug:: HTTP 模块的相同目录下。可以用 perldoc -l Devel::ebug::HTTP 找到该目录。*root/* 目录下有 Catalyst 需要的生成网页的文件。

注10：我们也可以猜出 URL，因为知道机器的名字，也有办法找出程序用的端口。

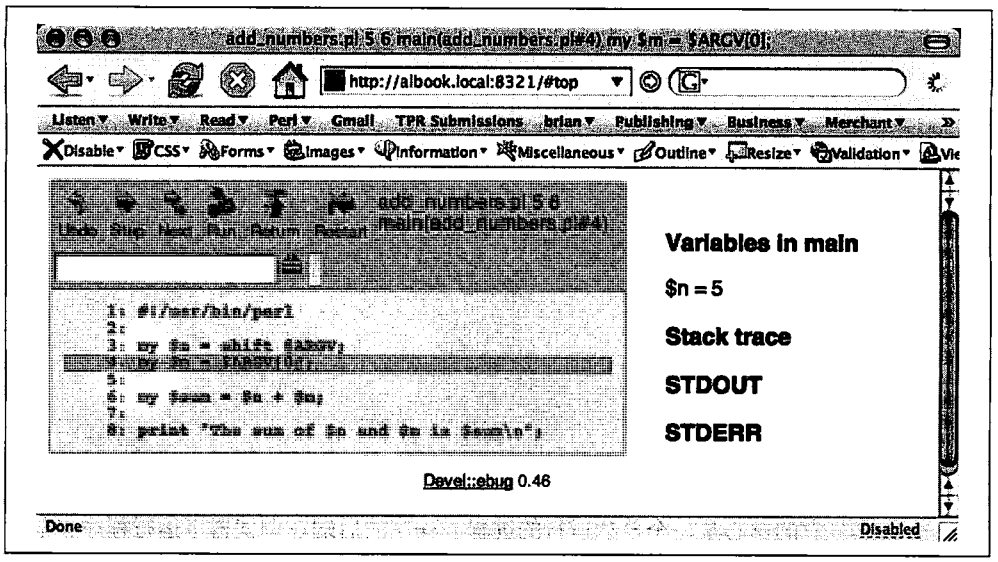


图 4-4：Devel::ebug::HTTP 模块允许我从浏览器调试运行在远程计算机上的程序

其他的调试器
Other Debuggers

EPIC

Eclipse（注 11）是一个能够运行在多种平台上的开源的开发环境。它是一个 Java 应用程序——不要被这点吓着了。它有着很好的模块化设计，允许人们进行扩展以满足自己的需求。EPIC（注 12）是 Eclipse 上的 Perl 插件。

不过，Eclipse 不仅仅是一个调试器，调试也不是它最有趣的功能。从 Perl 的源代码中，我们可以查看类的定义、调阅 Perl 的文档，以及做很多其他的事情。

Komodo

ActiveState 的 Komodo（图 4-5）是一个 Perl 的集成开发环境，最早只能运行在微软的 Windows 上，现在支持的平台包括 Solaris、Linux 和 Mac OS X。它除了支持 Perl 之外也支持其他的多种语言，包括 Tcl、Ruby、PHP 和 Python。

注 11：Eclipse Foundation（*http://www.eclipse.org*）。
注 12：Eclipse Perl Integration（*http://e-p-i-c.sourceforge.net*）。

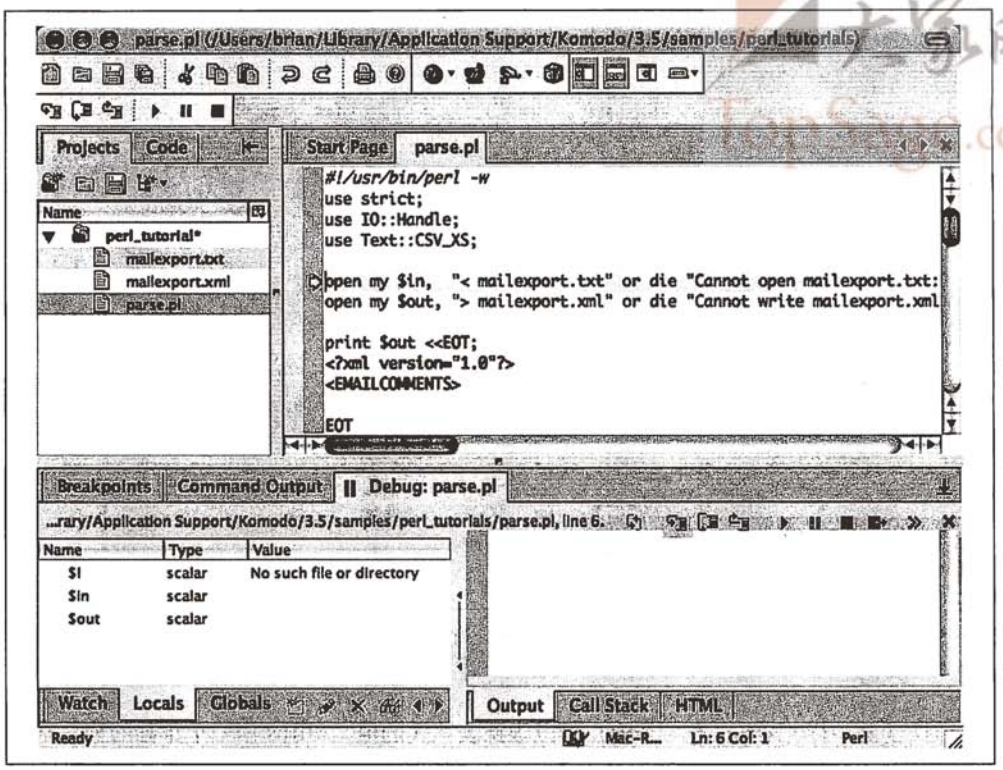


图 4-5：ActiveState 的 Komodo 是一个完整的开发环境。它甚至还包含了一个介绍使用方法的教程

Affrus

Affrus 是 Late Night Software 公司（注 13）出品的 Mac OS X 上的 Perl 专用的调试器。由于几乎只在 Mac 平台上工作，我非常喜欢这个 Mac 风格的调试器。Late Night Software 公司最早是从 AppleScript 的 Script Debugger 起家的，所以他们的产品是专门为 Mac 设计的。此外，Affrus 拥有常用的调试功能。

它的众多功能中我觉得非常有用的一个是 Affrus 的 Arguments 窗口。我可以在调试器中添加多种启动程序的方式，然后选择须要运行的方式。在图 4-6 中，我添加了两个不同的命令行并且选择了第一个。被选中的内容旁边有一个实心的钻石符号。当运行程序的时候，`@ARGV` 会得到元素 5 和 6。如果把它保存为 Affrus 文件的话，下次打开程序的时候依然可以使用它们。

注 13：Late Night Software（*http://www.latenightsw.com*）。

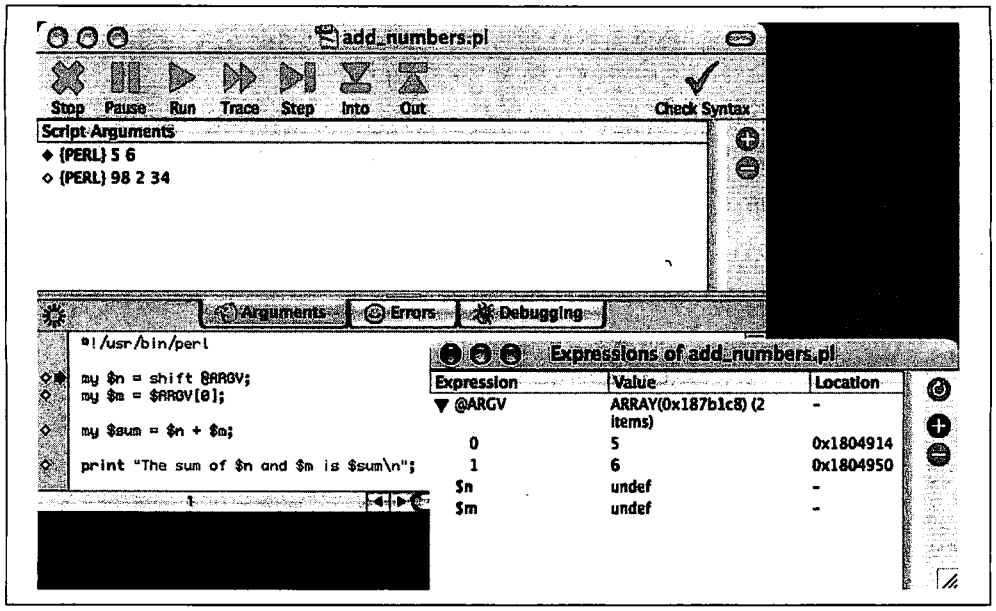


图 4-6：Affrus 允许我们配置多种启动程序的命令行参数。随着程序的运行，它也会更新表达式

和其他调试器一样，Affrus 有一个窗口可以跟踪表达式的值。Affrus 用一个单独的窗口来显示它们。在任何时候，我们也可以从 Debugging 窗口中看到所有变量的列表（图 4-7）。

总结
Summary

我们可以在希望的任何层次上调试 Perl 程序：可以给希望检查的部分加上调试代码，也可以用集成开发环境从外面调整它，甚至可以在运行程序的机器以外的另一台电脑上调试它。我们不必守着一种方法，也可以同时使用多种。如果对现有的调试器不满意，我们甚至可以为某个特定任务写一个自己的调试器。

深入阅读
Further Reading

Addison-Wesley 出版的 Peter Scott 和 Ed Wright 著的《Perl Debugged》是关于 Perl 编程的最好的书之一。它不仅告诉我们如何高效地调试 Perl 程序，还展示了如何避免陷入调试程序

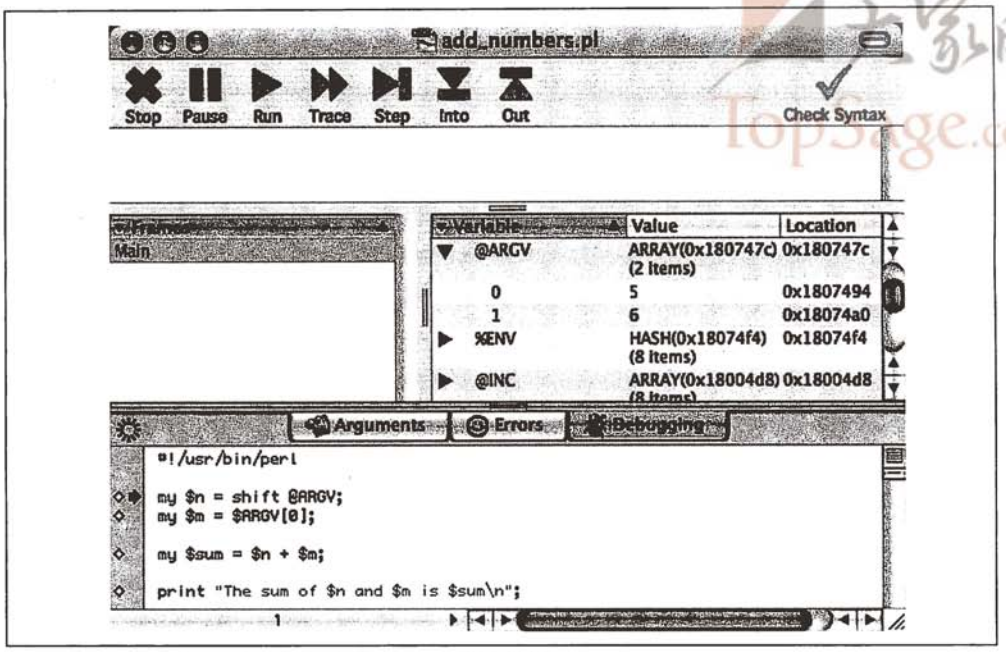


图 4-7：Affrus 在调试页中给我们显示了变量的值

的常见陷阱。遗憾的是，这本书好像已经不再印刷。不过，不要让 Amazon.com 上$1.99 的二手书的价钱误导了你对它的用途的判断。

Richard Foley 著的《Pro Perl Debugging》（Apress）告诉了你应该知道的关于 Perl 自带的调试器 `perl5db.pl` 的所有东西。如果喜欢 Perl 的默认调试器的话，这本书会告诉你所有希望知道的关于它的东西。

我的第一篇关于 Perl 的文章是为《The Perl Journal》第 9 期写的“Die-ing on the Web”。它在我的个人网站上：*http://www.pair.com/comdog/Articles/Die_and_the_Web.txt*。

我在《The Perl Journal》2005 年 7 月号的“Wrapping Subroutines”中详细地讨论了 `Hook::LexWrap`。这篇文章最早出现在《The Perl Journal》上，现在它也在《Dr. Dobb's Journal》的“Lightweight Languages”一节中：*http://www.ddj.com/dept/lightlan/1884416218*。

Addison-Wesley 出版的 Brian W. Kernighan 著的《The Practice of Programming》讨论了调试程序的方法。这不是一本关于 Perl 的书——其实它不是针对某种语言的。书中给出的很多切实可行的建议对任何语言都是适用的。

第5章

剖析 Perl 程序

Profiling Perl

在改进程序之前，我们要先确定须要改进的部分。在花时间改进程序之前，须要找出要集中精力解决的部分。怎样才能用最少的时间得到最大的提高呢？我们应该先做什么呢？通过剖析程序——记录并总结出程序做的工作，我们可以做出这些决定。幸运的是，Perl 为我们提供了很多剖析程序的工具。

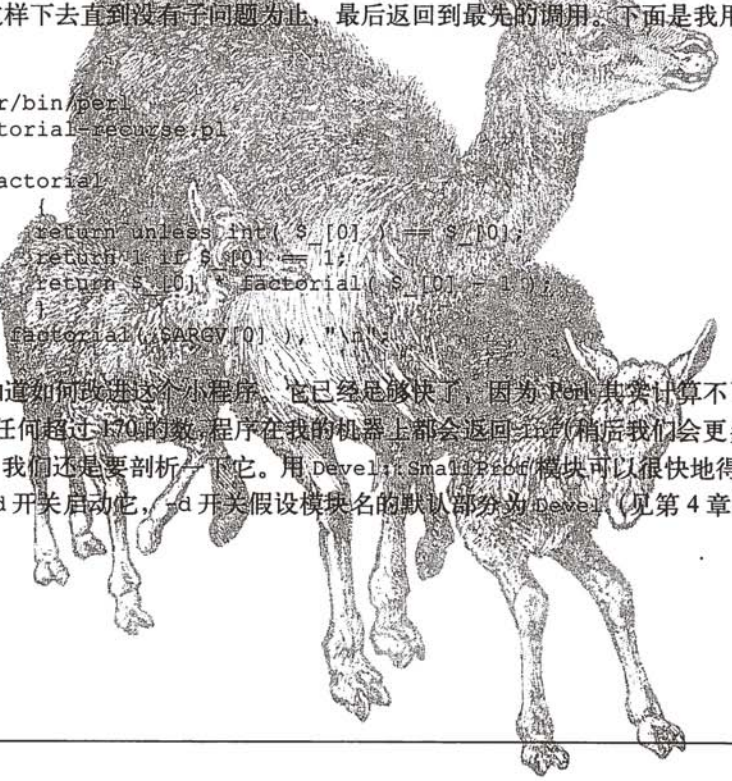
找到罪魁祸首

Finding the Culprit

我们想计算一个阶乘。这是一个讨论性能的老生常谈。不过，很快我们会看到一些更有趣的东西。在 Google 上搜索“阶乘子程序”时，几乎所有的实现（除了用汇编语言写的）都采用递归算法。这意味着一个子程序必须解决部分的问题，再调用它自己去解决一个子问题，一直这样下去直到没有子问题为止，最后返回到最先的调用。下面是我用 Perl 写出来的程序：

```
#!/usr/bin/perl
# factorial-recurse.pl

sub factorial
    {
    return unless int( $_[0] ) == $_[0];
    return 1 if $_[0] == 1;
    return $_[0] * factorial( $_[0] - 1 );
    }

print factorial( $ARGV[0] ), "\n";
```

下面我想知道如何改进这个小程序。它已经足够快了，因为 Perl 其实计算不了太大的数的阶乘。对于任何超过 170 的数，程序在我的机器上都会返回 Inf（稍后我们会更多地介绍它）。尽管如此，我们还是要剖析一下它。用 `Devel::SmallProf` 模块可以很快地得到一个概要。我们通过 `-d` 开关启动它，`-d` 开关假设模块名的默认部分为 `Devel`（见第 4 章）：



```
% perl -d:SmallProf factorial.pl 170
```

Devel::SmallProf 模块生成了一个清晰易读的文本文件 smallprof.out。它按列显示出了程序各行的执行时间、真实时间和 CPU 时间：

```
================ SmallProf version 1.15 ========================
                 Profile of factorial.pl                 Page 1
================================================================
 count wall tm  cpu time line
     0 0.000000 0.000000     1:#!/usr/bin/perl
     0 0.000000 0.000000     2:
   170 0.000000 0.000000     3:sub factorial {
   170 0.001451 0.000000     4: return unless int( $_[0] ) == $_[0];
   170 0.004367 0.000000     5: return 1 if $_[0] == 1;
   169 0.004371 0.000000     6: return $_[0] * factorial( $_[0] - 1 );
     0 0.000000 0.000000     7: }
     0 0.000000 0.000000     8:
     1 0.000009 0.000000     9:print factorial( $ARGV[0] ), "\n";
```

计算 170 的阶乘须要调用子程序 170 次。每次（除了 1 的时候!）调用子程序，都须要执行子程序中的所有代码。每次都须要检查参数是不是整数及参数是不是 1，而且几乎所有的情况下，都须要再次调用子程序。这是非常冗繁的一系列步骤。通过剖析程序，可以发现那些占用了大部分时间的部分，然后可以集中精力去改善它们。

解决这些问题的最好方法是找到一个更好的算法。更好的算法带来的性能提升往往会超过其他的所有方法。下面我们用迭代算法替换掉递归。利用范围操作符，我们可以轻易地得到整数的范围。在其他语言中，可以用一个 C 语言风格的循环：

```
#!/usr/bin/perl
# factorial-iterate.pl

sub factorial {
               return unless int( $_[0] ) == $_[0];
               my $f = 1;
               foreach ( 2 .. $_[0] ) { $f *= $_ };
               $f;
               }

print factorial( $ARGV[0] ), "\n";
```

剖析这个程序之后，我发现它做的计算比之前的少了很多。这个程序没有多少代码须要执行。它只须要检查一次参数，不须要检查参数是不是 1，也不须要重复地调用子程序：

```
================ SmallProf version 1.15 ========================
                 Profile of factorial2.pl                Page 1
================================================================
 count wall tm  cpu time line
     0 0.000000 0.000000     1:#!/usr/bin/perl
```

```
     0 0.000000 0.000000      2:
     1 0.000000 0.000000      3:sub factorial {
     1 0.000021 0.000000      4: return unless int( $_[0] ) == $_[0];
     1 0.000000 0.000000      5: my $f = 1;
   170 0.001632 0.000000      6: foreach ( 2 .. $_[0] ) { $f *= $_ };
     1 0.002697 0.000000      7: $f;
     0 0.000000 0.000000      8: }
     0 0.000000 0.000000      9:
     1 0.000006 0.000000      10:print factorial( $ARGV[0] ), "\n";
```

之前说过调用超过 170 时我们的程序会溢出。如果告诉 Perl 使用 bignum 编译指令，我们就可以超过这个上限：

```
#!/usr/bin/perl
# factorial-recurse-bignum.pl

use bignum;

sub factorial {
       return unless int( $_[0] ) == $_[0];
       return 1 if $_[0] == 1;
       return $_[0] * factorial( $_[0] - 1 );
       }

print factorial( $ARGV[0] ), "\n";
```

现在，比较很大的数的阶乘可以看到明显的性能差别了。当我快完成这本书的时候，我转到了一台 MacBook Pro 上继续工作。它的双核架构使得它处理任何一种方法都很快。只有在计算很大的数的时候递归的方法才慢了下来。

不过，故事还没有完。我们只看了一个非常简单的只计算一个数的程序。真正的程序很可能会用不同的值多次调用 factorial 过程。如果剖析应用程序的话，我们会看到整个过程中子程序的每行代码被执行的次数。

如果把结果缓存起来，两种方法都可以从中受益。下面的程序反复地提示我输入一个数。它会计算出阶乘并把结果缓存起来，从而用空间换取速度的提升。第一次让它计算 10 000 的阶乘时，它花了好几秒。之后，再计算 10 000 以下的数的阶乘时，它只须做一个非常快的查找操作就完成了：

```
#!/usr/bin/perl
# factorial-iterate-bignum-memo.pl

use bignum;

{
my @Memo      = (1);

sub factorial {
       my $number = shift;

       return unless int( $number ) == $number;
```

```
        return $Memo[$number] if $Memo[$number];

        foreach ( @Memo .. $number )
                {
                $Memo[$_] = $Memo[$_ - 1] * $_;
                }

        $Memo[ $number ];
        }
}

{
print "Enter a number> ";
chomp( my $number = <STDIN> );
exit unless defined $number;

print factorial( $number ), "\n";
redo;
}
```

我们也可以对递归的版本做同样的事情。不过，Memoize 模块替我做了这些工作：

```
#!/usr/bin/perl
# factorial-recurse-bignum-memo.pl

use bignum;

use Memoize;

memoize( factorial );

sub factorial {
       return unless int( $_[0] ) == $_[0];
       return 1 if $_[0] == 1;
       return $_[0] * factorial( $_[0] - 1 );
       }

{
print "Enter a number> ";
chomp( my $number = <STDIN> );
exit unless defined $number;

print factorial( $number ), "\n";
redo;
}
```

剖析程序的时候，我们必须记住隔离的部分并不能反映全部。剖析可以帮助我们做决定，不过进行思考的是我们而不是计算机。

通用的方法
The General Approach

剖析意味着统计。要想统计点东西，我们须要写一些语句来统计一些数据。比说如，可以用第 4 章中介绍的功能给子程序添加一些统计的代码。不过，这样做太繁琐了。我们可以让所有的代码都经过一个单独的控制子程序，而不是单独地统计每个子程序。对于小程序来说，也许这样做意味着太多的额外工作，不过在一个大系统中，在我们优化程序时额外的计算会得到回报。

我做统计最频繁的地方是在数据库的代码中。对于数据库，我们希望跟踪提交的查询，这样就可以了解哪些查询花费的时间较多，哪些查询最常使用。从这些数据里，我们就可以找出须要优化的部分。

下面的这个例子可以处理所有的查询，它可以用来剖析数据库的代码。我简化了这个例子，去掉了一些和代码剖析无关的部分，不过它和真实使用的代码很接近。变量%Queries 用来保存剖析的数据，它在整个软件包范围内都是有效的。方法 simple_query 其实是把 prepare 和 execute 方法及一些统计代码包了一层：

```
package My::Database;

my %Queries;

sub simple_query
        {
        my( $self, @args ) = @_;

        my $sql_statement = shift @args;

        $Queries{$sql_statement}++;  # <--- Profiling hook

        my $sth = $self->dbh->prepare( $sql_statement );
        unless( ref $sth ) { warn $@; return }

        my $rc   = $sth->execute( @args );

        wantarray ? ( $sth, $rc ) : $rc;
        }
```

在其余的数据库代码中，有的函数会调用 simple_query 而不是直接使用 DBI 接口。我们的函数 get_postage_rates_by_country 可以计算发信到国外的邮费。它把 SQL 语句和一个 bind 参数传给了 simple_query。和之前一样，这也是真实的代码。不过我只显示出了相关的部分：

```
sub get_postage_rates_by_country
          {
          my( $self, $country ) = @_;

          my( $sth ) = $self->simple_query( <<"SQL", $country );
                SELECT
```

```
                PostageRates.ounces,
                PostageRates.rate,
                PostageServices.name
        FROM
                PostageRates, Countries, PostageServices
        WHERE
                Countries.pk = ?
        AND
                Countries.usps_zone = PostageRates.usps_zone
        AND
                PostageRates.service = PostageServices.pk
        ORDER BY
                PostageRates.ounces
SQL

        return $sth->fetchall_arrayref;
        }
```

当我们的程序工作时，所有的查询会经过 simple_query，simple_query 会统计和记录所有发生的事情。为了得到剖析的数据，我们用一个 END 块生成报告。具体的格式取决于运行时收集的数据。在这个例子中，我们只统计了语句被执行的次数。不过我们可以用哈希表%Queries 保存任何需要的东西，包括 bind 的参数、调用 simple_query 的函数，等等：

```
END {
        foreach my $statement ( sort { $b <=> $a } keys %Queries )
                {
                printf "%5d %s\n\n", $Queries{$statement}, $statement;
                }
}
```

我们可能会在长长的报告中发现程序在反复地获取每个国家的邮费数据，即使它不会变化。当我们分析数据发现这个问题后，就可以优化一下代码，把一些数据缓存在内存中，而不是每次到数据库中请求同样的内容。

其实，我已经有很长一段时间在这样实现我的数据库代码了。最近我发现 Tim Bunce 直接把这些功能加到了 DBI 中。他做了同样的事情，让所有的东西都经过一个集中处理的函数。对于 DBI 来说这非常的容易，因为 DBI 已经对所有的查询这样做了。

Profiling DBI

DBI::Profile 模块可以使用 Perl 的数据库接口模块 DBI 做类似的事情。数据库操作往往是程序最大的性能瓶颈，通常它也是我们最早开始努力改进的部分。就像在上面那个例子中不必要的子程序调用一样，我们可能进行了不必要的数据库查询。

下面的这个短小的程序由于进行了 2 000 次数据库查询而花了很长的时间。在这个例子中，我们想建立一个表来存放数字的名字，这样给定一个数字就可以得到它的名字（比如 9 的名字是“Nine”），或者从名字得到数字。我们可以使用某个 Lingua::*模块，不过在这里我不想用一些太高级的东西。在这个例子中，我们使用 DBD::CSV 模块用以逗号分隔值的文件作为数据库的存储。我们先创建一个表来保存数字-名字对，然后开始往表里添加内容。我们先把前 19 个名字放到表格中，然后再查找已有的名字来创建更多的名字：

```
#!/usr/bin/perl
# dbi-number-inserter.pl
use strict;

use DBI;

my $dbh = DBI->connect( "DBI:CSV:f_dir=." );

$dbh->do( "DROP TABLE names" );
$dbh->do( "CREATE TABLE names ( id INTEGER, name CHAR(64) )" );

my $sth = $dbh->prepare( "INSERT INTO names VALUES ( ?, ? )" );

my $id = 1;
foreach my $name (
        qw(One Two Three Four Five Six Seven Eight Nine Ten),
        qw(Eleven Twelve Thirteen Fourteen Fifteen Sixteen Seventeen Eighteen
                Nineteen)
        )
        {
        $sth->execute( $id++, $name );
        }

foreach my $name ( qw( Twenty Thirty Forty Fifty Sixty Seventy Eighty Ninety ) )
       {
       $sth->execute( $id++, $name );

       foreach my $ones_digit ( 1 .. 9 )
               {
               my( $ones_name ) = map { lc } $dbh->selectrow_array(
                       "SELECT name FROM names WHERE id = $ones_digit"
                       );
               $sth->execute( $id++, "$name $ones_name" );
               }
       }

foreach my $digit ( 1 .. 9 )
        {
        my( $hundreds ) = $dbh->selectrow_array(
               "SELECT name FROM names WHERE id = $digit"
               );

        $sth->execute( $id++, "$hundreds hundred" );

        foreach my $tens_digit ( 1 .. 99 )
```

```
        {
        my( $tens_name ) = map { lc } $dbh->selectrow_array(
                "SELECT name FROM names WHERE id = $tens_digit"
                );
        $sth->execute( $id++, "$hundreds hundred $tens_name" );
        }
    }
```

从命令行运行这个程序时，它在我的 Powerbook G4 上几乎花了两分钟。还行，正好我需要一个比较慢的例子。现在，我们剖析一下这个程序看看如何改进它。假设我们刚拿到这个程序并不知道它是如何工作的。我们设置好环境变量 DBI_PROFILE 以打开数据库剖析功能（注 1）。为了让报告根据语句排序，我们设定 DBI_PROFILE='!Statement'。排序用的键前面有一个感叹号。运行结束后，我们得到了一个很长的报告。这里是开头的几行：

```
$ env DBI_PROFILE='!Statement' perl dbi-profile.pl
DBI::Profile: 109.671362s 99.70% (1986 calls) dbi-profile.pl @ 2006-10-10 02:18:40
'' =>
        0.000784s / 10 = 0.000078s avg (first 0.000023s, min 0.000001s, max 0.000618s)
'CREATE TABLE names ( id INTEGER, name CHAR(64) )' =>
        0.004258s
'DROP TABLE names' =>
        0.008017s
'INSERT INTO names VALUES ( ?, ? )' =>
        3.229462s / 1002 = 0.003223s avg (first 0.001767s, min 0.000037s, max 0.108636s)
'SELECT name FROM names WHERE id = 1' =>
        1.204614s / 18 = 0.066923s avg (first 0.012831s, min 0.010301s, max 0.274951s)
'SELECT name FROM names WHERE id = 10' =>
        1.118565s / 9 = 0.124285s avg (first 0.027711s, min 0.027711s, max 0.341782s)
'SELECT name FROM names WHERE id = 11' =>
        1.136748s / 9 = 0.126305s avg (first 0.032328s, min 0.032328s, max 0.378916s)
```

最上面的行给出了挂钟时间和 DBI 方法被调用的总次数。它是 DBI 被调用的次数，而不是查询的次数。之后的部分给出了每个查询的报告，按词汇顺序排列。它看起来像是按照总时间或查询的次数排序的。不要忘了看看报告的其他部分。它其实是按查询的字母顺序排列的。

对于每个查询，DBI::Profile 报告了总的挂钟时间和调用的次数。它没有汇报 CPU 时间，因为这个不是很有用。数据库服务器可能是另一台机器，即使是在本地，它通常是一个单独的进程。报告给出了每个查询的平均时间、第一次调用花的时间、调用所花的最短时间和最长时间。这不是一个简单的记时程序。即使是相同的数据，数据库服务器也可能会有不同的响应——可能它在处理一些别的任务，或者数据的长短不同，或者有一些其他的原因。

注 1：当然，我们也可以在程序里设置$dbh->{Profile}。

从完整的报告中可以看到，大部分的调用花的时间大致相同，因为它们都非常快。所以通过优化一个查询并不能获得多大的提高。去掉索引或调整归并操作也不会有太大的帮助。

我们真正须要的是降低查询的次数以减少和数据库的交互。我们没法去掉 INSERT 语句，因为我们须要生成每一行，但是我们不需要所有的选择语句。我们应该把结果缓存下来，这样就不用两次取同样的数据了（甚至一次也不用）：

```
#!/usr/bin/perl
# dbi-number-inserter-cached.pl
use strict;

use DBI;

my $dbh = DBI->connect( "DBI:CSV:f_dir=." );

$dbh->do( "DROP TABLE names" );
$dbh->do( "CREATE TABLE names ( id INTEGER, name CHAR(64) )" );

my $insert = $dbh->prepare( "INSERT INTO names VALUES ( ?, ? )" );


my @array = ( qw( Zero ),
        qw(One Two Three Four Five Six Seven Eight Nine Ten),
        qw(Eleven Twelve Thirteen Fourteen Fifteen Sixteen Seventeen Eighteen
                Nineteen)
        );

my $id = 0;
foreach my $name ( @array )
        {
        $insert->execute( $id++, $name );
        }

foreach my $name ( qw( Twenty Thirty Forty Fifty Sixty Seventy Eighty Ninety ) )
        {
        $array[ $id ] = $name;
        $insert->execute( $id++, $name );
        foreach my $ones_digit ( 1 .. 9 )
                {
                my $full_name = $array[ $id ] = "$name $array[$ones_digit]";
                $insert->execute( $id++, $full_name );
                }
        }

foreach my $digit ( 1 .. 9 )
        {
        my( $hundreds ) = $array[ $digit ];
        my $name = $array[$id] = "$hundreds hundred";
        $insert->execute( $id++, $name );

        foreach my $tens_digit ( 1 .. 99 )
```

```
            {
            my( $tens_name ) = lc $array[ $tens_digit ];
            $array[$id] = "$hundreds hundred $tens_name";
            $insert->execute( $id++, "$name $tens_name" );
            }
    }
```

第一轮优化后，由于结果被缓存了起来，所以完全去掉了选择语句。这样就砍掉了程序大部分的运行时间。优化前后两个程序的运行时间有明显的不同。不过要记住，我们在时间和内存方面做了一个折中。第二个程序要快很多，但是它消耗了更多的内存：

```
$ time  perl dbi-profile.pl
real     1m48.676s
user     1m21.136s
sys      0m1.698s

$ time  perl dbi-profile2.pl
real     0m2.638s
user     0m1.736s
sys      0m0.307s
```

下面是新程序的完整的剖析报告。新程序的运行时间只有原有的百分之二。大部分的操作都是 INSERT：

```
$ env DBI_PROFILE='!Statement' perl dbi-profile2.pl
DBI::Profile: 2.118577s 105.93% (1015 calls) dbi-profile2.pl @ 2006-10-10 02:31:10
'' =>
       0.000757s / 10 = 0.000076s avg (first 0.000021s, min 0.000001s, max 0.000584s)
'CREATE TABLE names ( id INTEGER, name CHAR(64) )' =>
       0.004216s
'DROP TABLE names' =>
       0.006906s
'INSERT INTO names VALUES ( ?, ? )' =>
       2.106698s / 1003 = 0.002100s avg (first 0.001713s, min 0.000037s, max 0.005587s)
```

通过分析剖析报告，我们找到了程序的关键部分进行优化。虽然它没有告诉我们如何改进，至少我们知道应该把时间花在哪里了。

DBI::Profile 的其他报告

Other DBI::Profile Reports

上面的运行时间报告并不是我们能够得到的唯一报告。使用 DBI_PROFILE='!MethodName' 后，DBI 会根据 DBI 函数的名字对报告的内容排序。报告会根据 ASCII 字符顺序排列，大写字符在小写字符的前面（我部分修改了下面的报告，因为报告显示了所有的方法，有的方法我甚至不知道自己是否使用过）：

```
$ env DBI_PROFILE='!MethodName' perl dbi-profile2.pl
DBI::Profile: 2.168271s 72.28% (1015 calls) dbi-profile2.pl @ 2006-10-10 02:37:16
'DESTROY' =>
       0.000141s / 2 = 0.000070s avg (first 0.000040s, min 0.000040s, max 0.000101s)
'FETCH' =>
        0.000001s
'STORE' =>
```

```
        0.000067s / 5 = 0.000013s avg (first 0.000022s, min 0.000006s, max 0.000022s)
'do' =>
        0.010498s / 2 = 0.005249s avg (first 0.006602s, min 0.003896s, max 0.006602s)
'execute' =>
        2.155318s / 1000 = 0.002155s avg (first 0.002481s, min 0.001777s, max 0.007023s)
'prepare' =>
        0.001570s
```

我们也可以把上面的两个报告合并起来。用冒号把多个键连起来后，DBI::Profile 就可以根据多个键排序。使用 DBI_PROFILE='!Statement:!MethodName'后，DBI 会给出一个两层的报告。在每个 SQL 语句下面，它会逐个列出函数运行花费的时间。利用这个报告，我们可以比较数据库查询花在 DBI 内部和和花在获取数据上的时间：

```
$ env DBI_PROFILE='!Statement:!MethodName' perl dbi-profile2.pl
DBI::Profile: 2.123325s 106.17% (1015 calls) dbi-profile2.pl @ 2006-10-10 02:38:22
'' =>
        'FETCH' =>
                0.000001s
        'STORE' =>
                0.000069s / 5 = 0.000014s avg (first 0.000024s, min 0.000005s,↵
                max 0.000024s)
        'connect' =>
                0.000644s
        'default_user' =>
                0.000030s
        'disconnect' =>
                0.000050s
        'disconnect_all' =>
                0.000024s
'CREATE TABLE names ( id INTEGER, name CHAR(64) )' =>
        'do' =>
                0.004616s
'DROP TABLE names' =>
        'do' =>
                0.007191s
'INSERT INTO names VALUES ( ?, ? )' =>
        'DESTROY' =>
                0.000149s / 2 = 0.000075s avg (first 0.000050s, min 0.000050s,↵
                max 0.000099s)
        'execute' =>
                2.108945s / 1000 = 0.002109s avg (first 0.002713s, min 0.001796s,↵
                max 0.005454s)
        'prepare' =>
                0.001606s
```

我们也可以使用 DBI_PROFILE='!MethodName: !Statement'把上面的报告倒过来。下面的结果先按 DBI 方法分类，再根据 SQL 语句细分：

```
$ env DBI_PROFILE='!MethodName:!Statement' perl dbi-profile2.pl
DBI::Profile: 2.431843s 81.06% (1015 calls) dbi-profile2.pl @ 2006-10-10 02:40:40
'DESTROY' =>
        'INSERT INTO names VALUES ( ?, ? )' =>
                0.000142s / 2 = 0.000071s avg (first 0.000039s, min 0.000039s,↵
                max 0.000103s)
```

```
'FETCH' =>
       '' =>
              0.000001s
'STORE' =>
       '' =>
              0.000065s / 5 = 0.000013s avg (first 0.000022s, min 0.000005s,↵
              max 0.000022s)
'connect' =>
        '' =>
              0.000685s
'default_user' =>
       '' =>
              0.000024s
'disconnect' =>
       '' =>
              0.000050s
'disconnect_all' =>
       '' =>
              0.000023s
'do' =>
       'CREATE TABLE names ( id INTEGER, name CHAR(64) )' =>
              0.004287s
       'DROP TABLE names' =>
              0.006389s
'execute' =>
       'INSERT INTO names VALUES ( ?, ? )' =>
              2.418587s / 1000 = 0.002419s avg (first 0.002549s, min 0.001819s,?
              max 0.013104s)
'prepare' =>
       'INSERT INTO names VALUES ( ?, ? )' =>
              0.001589s
```

进一步简化

Making It Even Easier

Sam Tregar 的 `DBI::ProfileDumper` 模块有着和 `DBI::Profile` 一样的功能，不过它会把结果保存在文件中而不是输出到标准输出（注 2）。默认的文件名是 *dbi.prof*，不过我们也可以使用任何喜欢的名字。对于大部分稍大点的程序，我们可能须要做很多切片操作来提取需要的信息：

首先，我们在 `DBI_PROFILE` 中指定类名，告诉 DBI 须要用的类。我们用“/”把类的名字和剖析时排序用的键连起来：

```
$ env DBI_PROFILE='!Statement'/DBI::ProfileDumper ./program.pl
```

这样，命令执行完后，*dbi.prof* 中会有所有剖析的数据。如果须要改变文件名，我们可以把它加在类名的后面：

```
$ env DBI_PROFILE='!Statement'/DBI::ProfileDumper/File:dbi.prof ./program.pl
```

注 2：Sam 也为 mod_perl 下的剖析程序编写了 `DBI::ProfileDumper::Apache` 模块。

一旦得到了数据，我们就可以用 dbiprof 分析数据。dbiprof 有很多选项可以用来选择希望显示的数据、排序的方式（甚至是基于多个键）及很多其他的功能：

```
$ dbiprof --number all --sort longest
```


切换数据库
Switching Databases

我们其实是从一个比较糟糕的程序开始的。那个程序做了很多不必要的数据库调用，也做了很多其他的事情。实际上，我们可以巧妙地利用 Perl 的列表操作把那个程序变得在风格上更像 Perl 程序。我们可以把数据放在一个数组中，而不用索引计数。这样在完成同样的工作的同时，代码会变得更加简短。我们还可以把所有数据库操作挪到最后（稍后我会介绍我的秘密计划），而不是一路不停地插入元素。现在，程序几乎和前面的例子运行得一样快：

```
#!/usr/bin/perl
# dbi-number-inserter-end.pl
use strict;

use DBI;

my @array = ( qw( Zero ),
        qw(One Two Three Four Five Six Seven Eight Nine Ten),
        qw(Eleven Twelve Thirteen Fourteen Fifteen Sixteen Seventeen Eighteen
              Nineteen)
        );

Foreach  my $name ( qw( Twenty Thirty Forty Fifty Sixty Seventy Eighty Ninety ) )
        {
        push @array, $name;
        push @array, map { "$name $array[$_]" } 1 .. 9
        }

foreach  my $digit ( 1 .. 9 )
        {
        push @array, "$array[$digit] hundred";
        push @array, map { "$array[$digit] hundred $array[$_]" } 1 .. 99;
        }

my $dbh = DBI->connect( "DBI:CSV:f_dir=." );

$dbh->do( "DROP TABLE names" );
$dbh->do( "CREATE TABLE names ( id INTEGER, name CHAR(64) )" );

my $insert = $dbh->prepare( "INSERT INTO names VALUES ( ?, ? )" );

foreach  my $index ( 0 .. $#array )
        {
        $insert->execute( $index, $array[$index] );
        }
```

现在，我想用一个更复杂的数据库服务器而不是 CSV 文件，因为我认为把数据写到磁盘上性能会更好。既然我们有工具来分析程序的性能了，为什么不试试呢？我们将使用 SQLite，它是一个可以与 DBI 通信的轻量级的数据库服务器。不须要改变太多的程序，因为 DBI 帮我们把很多细节都隐藏了起来。我们只用改变 DBI 连接：

```
# dbi-number-inserter-sqlite.pl
my $dbh = DBI->connect( "DBI:SQLite:dbname=names.sqlite.db" );
```

重新运行时，程序非常地慢。它往 SQLite 数据库中插入数据时花了很多时间：

```
$ time perl dbi-profile-sqlite.pl
real    5m7.038s
user    0m0.572s
sys     0m8.220s
```

真是糟糕透了！当我们剖析程序的时候，可以看到 INSERT 花费的时间是前一个程序的 100 多倍。真是夸张！

```
% DBI_PROFILE='!Statement' perl dbi-profile-sqlite.pl
DBI::Profile: 266.582027s 99.63% (1015 calls) dbi-profile-sqlite.pl @ 2006-03-22↵
17:19:51
'' =>
        0.004483s / 10 = 0.000448s avg (first 0.000007s, min 0.000003s, max 0.004329s)
'CREATE TABLE names ( id INTEGER, name CHAR(64) )' =>
         0.413145s
'DROP TABLE names' =>
         0.294514s
'INSERT INTO names VALUES ( ?, ? )' =>
         265.869885s / 1003 = 0.265075s avg (first 0.000306s, min 0.000016s,↵
         max 0.771342s)
```

不过这是 SQLite 和一些其他数据库的众所周知的问题。因为它们会自动地提交每一个查询，并且一直等到数据写到磁盘上后再继续下面的操作。其实我们可以把所有的操作放在一个交易中，而不是单独插入每行。这样就不用为每个 INSERT 操作写数据库了。写操作会在 COMMIT 时一次完成：

```
# dbi-profile-sqlite-transaction.pl
$dbh->do( "BEGIN TRANSACTION" );
foreach my $index ( 0 .. $#array )
         {
         $insert->execute( $index, $array[$index] );
         }
$dbh->do( "COMMIT" );
```

现在，剖析的结果就大不相同了。从结果中可以看到，插入的速度有了几十倍的提高。从任何方面看，现在的程序都比之前的快了很多，虽然两个程序做的事情完全一样：

```
%  DBI_PROFILE='!Statement' perl dbi-profile-sqlite2.pl
DBI::Profile: 1.334367s 54.19% (1016 calls) dbi-profile-sqlite2.pl @ 2006-03-22 17:25:44
'' =>
        0.004087s / 9 = 0.000454s avg (first 0.000007s, min 0.000003s, max 0.003950s)
```

```
'BEGIN TRANSACTION' =>
       0.000257s
'COMMIT' =>
       0.255082s / 2 = 0.127541s avg (first 0.254737s, min 0.000345s, max 0.254737s)
'CREATE TABLE names ( id INTEGER, name CHAR(64) )' =>
       0.271928s
'DROP TABLE names' =>
       0.715443s
'INSERT INTO names VALUES ( ?, ? )' =>
       0.087570s / 1002 = 0.000087s avg (first 0.000317s, min 0.000004s, max 0.003396s)
```

Devel::DProf

在本章中，我是从 Devel::SmallProf 开始介绍的。这样做的唯一原因是用 Devel::SmallProf 可以很快地得到结果。Devel::DProf 能做的事情基本差不多，只是它会把结果保存为自己的格式。这样就可以用这些数据做很多事情，比如画出非常漂亮的图。调用它的方法和 Devel::SmallProf 一样，也是用-d 开关。

我有一个程序可以通过 SOAP 接口获取 Use.Perl 期刊（注 3）的内容。我们用-d 开关指定 Devel::DProf 作为调试模块启动该程序：

```
% perl -d:DProf journals
```

程序执行完成后，我们得到了一个新文件 *tmon.out*。我们也可以用环境变量 PERL_DPROF_OUT_FILE_NAME 改变文件名。这个文件不适合直接阅读，须要用 dprofpp 把它转换成适合的格式。程序的挂钟时间是 53 秒，CPU 时间不到 1 秒。我们可以改进程序的某些部分，不过大部分时间都消耗在了网络延迟和下载上：

```
$ dprofpp
LWP::Protocol::collect has 17 unstacked calls in outer
Compress::Zlib::__ANON__ has 5 unstacked calls in outer
...snipped several more lines like these...
HTTP::Message::__ANON__ has 8 unstacked calls in outer
Total Elapsed Time = 53.08383 Seconds
  User+System Time = 0.943839 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 8.37   0.079  0.000     84   0.0009 0.0000  utf8::SWASHNEW
 6.25   0.059  0.146      5   0.0118 0.0292  main::BEGIN
 5.83   0.055  0.073     24   0.0023 0.0030  Text::Reform::form
 5.09   0.048  0.067      2   0.0242 0.0334  HTTP::Cookies::Netscape::load
 4.24   0.040  0.040     10   0.0040 0.0040  LWP::UserAgent::BEGIN
 4.24   0.040  0.049      9   0.0044 0.0054  Text::Autoformat::BEGIN
 3.71   0.035  0.035    697   0.0001 0.0001  HTTP::Headers::_header
 3.18   0.030  0.030      8   0.0037 0.0037  DynaLoader::dl_load_file
 3.18   0.030  0.079      2   0.0149 0.0393  Text::Template::GEN0::BEGIN
 3.18   0.030  0.068     17   0.0017 0.0040  LWP::Protocol::implementor
 2.65   0.025  0.045    221   0.0001 0.0002  SOAP::SOM::_traverse_tree
```

注 3：Use.Perl 由 Chris Nandor 负责维护（*http://use.perl.org*）。

```
 2.54   0.024  0.024   892    0.0000 0.0000  HTTP::Cookies::set_cookie
 2.44   0.023  0.023  1060    0.0000 0.0000  Text::Reform::_debug
 2.44   0.023  0.061   293    0.0001 0.0002  SOAP::SOM::_traverse
 2.12   0.020  0.030     5    0.0040 0.0059  HTTP::Cookies::BEGIN
```

如果我们不希望根据执行时间排序的话，也可以选择其他的排序方式。使用-l（小写的L），输出的结果会按照子程序的被调用次数排列。

```
$ dprofpp -l
LWP::Protocol::collect has 17 unstacked calls in outer
Compress::Zlib::__ANON__ has 5 unstacked calls in outer
... snip ...
HTTP::Message::__ANON__ has 8 unstacked calls in outer
Total Elapsed Time = 53.08383 Seconds
  User+System Time = 0.943839 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 1.06   0.010  0.010   1525   0.0000 0.0000  UNIVERSAL::isa
 0.21   0.002  0.002   1184   0.0000 0.0000  LWP::Debug::debug
   -       - -0.009   1156        -      -  SOAP::Data::new
 2.44   0.023  0.023   1060   0.0000 0.0000  Text::Reform::_debug
 2.54   0.024  0.024    892   0.0000 0.0000  HTTP::Cookies::set_cookie
   -       - -0.005    753        -      -  SOAP::Serializer::new
   -       - -0.014    734        -      -  SOAP::Serializer::__ANON__
 3.71   0.035  0.035    697   0.0001 0.0001  HTTP::Headers::_header
   -       -  0.000    527        - 0.0000  HTTP::Message::__ANON__
 0.74   0.007  0.007    439   0.0000 0.0000  UNIVERSAL::can
 0.64   0.006  0.004    425   0.0000 0.0000  HTTP::Headers::__ANON__
   -       - -0.002    382        -      -  SOAP::Utils::o_lattr
   -       - -0.002    369        -      -  SOAP::Trace::__ANON__
   -       - -0.002    323        -      -  HTTP::Message::_elem
 0.64   0.006  0.024    323   0.0000 0.0001  HTTP::Headers::push_header
```

使用dprofpp的-T开关，可以看到子程序的调用顺序。我的journals程序总共进行了25 000次调用（当我写这个程序的时候并没有觉得它有这么复杂），下面是输出中SOAP模块的部分：

```
$ dprofpp -T
... snip ...
SOAP::Serializer::encode_object
   SOAP::Serializer::multiref_object
        SOAP::Serializer::gen_id
        SOAP::Serializer::__ANON__
             SOAP::Serializer::new
  UNIVERSAL::isa
  UNIVERSAL::isa
  SOAP::Serializer::encode_scalar
        SOAP::XMLSchema::Serializer::xmlschemaclass
        SOAP::Serializer::maptypetouri
        SOAP::Serializer::encode_object
             SOAP::Serializer::multiref_object
                  SOAP::Serializer::gen_id
                   SOAP::Serializer::__ANON__
```

```
                                SOAP::Serializer::new
    ... snip ...
```

不过，我们应该注意到，Devel::DProf 有时会遇到问题产生段错误。在那种情况下，我们可以用 Devel::Profiler。它全部是用 Perl 写的，可以用来替代 Devel::DProf，只是速度上要慢一些。

实现自己的剖析程序
Writing My Own Profiler

第一个例子中用到的 Devel::SmallProf 模块并没有多么复杂。如果看看该模块的实现的话，就会发现它并没有多少代码。写一个自己的剖析程序其实非常容易（注 4）。

Devel::LineCounter

在这一节里，我们将实现一个能够统计 Perl 运行程序时每行代码执行了多少次的简单的剖析程序。模块 Devel::SmallProf 已经为我们做了这个工作，不过一旦我们知道如何自己实现一些基本的功能，就可以修改程序做任何的事情。

用-d 开关运行下面的程序时，对于每行语句 Perl 都会调用一个特殊的子程序&DB::DB（Perl 默认的调试器只是一个普通的程序，它的工作方式也基本一样）。这是软件包 DB 中名为 DB 的子程序。我们可以在这个子程序中做任何想做的事情。在这里我们统计每行程序执行的次数：

```
package Devel::LineCounter;

package DB;
use strict;
use warnings;

my @counter = ();

sub DB
        {
        my( $file, $line ) = ( caller )[1,2];

        next unless $file eq $0;

        $counter[$line]++;
        }
```

为了不改变原先的程序就可以得到剖析的输出，我们给 LineCounter 模块添加了一个 END 块，它会在程序结束的时候执行。不过也可能后面还有其他的 END 块：

注 4：这些讨论中的一部分内容也出现在 Dr. Dobb 的在线文章《Creating a Perl Debugger》中（*http://www.ddj.com/documents/s=1498/ddj0103bpl/*）。

```
END
        {
        print "\nLine summary for $0\n\n";

        open FILE, $0 or die "Could not open $0\n$!";

        while( <FILE> )
                {
                printf "%6d %s", $counter[++$count] || 0, $_;
                }
        }
```

我们把新的模块保存在合适的位置（比如保存为 *Devel/LineCounter.pm*，放在 Perl 搜索模块的路径中），然后通过-d 开关启动程序：

```
% perl -d:LineCounter factorial.pl
```

剖析测试套件
Profiling Test Suites

模块 Devel::Cover 可以剖析测试套件，告诉我们测试所覆盖的代码范围。它可以统计测试套件运行的每行代码的次数，也可以记录下所有的代码分支的执行情况。理想情况下，我们的测试应该覆盖所有的代码，并能够执行所有的条件分支。

Devel::Cover

Devel::Cover 带有一个 cover 命令，该命令可以汇报代码的覆盖率。要想使用这个工具，我们须要先清除之前的所有状态。不过，我们不一定要这样做。有时我们可能希望把这次运行的数据添加到之前的结果中，或者和工程中其他部分的数据合并到一起：

```
$ cover -delete
```

清除之前的数据后，我们加载 Devel::Cover 模块并运行程序。命令 make test 会使用 Test::Harness 运行所有的测试程序。所以我们可以在命令行设置 HARNESS_PERL_SWITCHES 来让 Test::Harness 加载模块 Devel::Cover：

```
$ HARNESS_PERL_SWITCHES=-MDevel::Cover make test
```

如果使用的是 Modeul::Build 而不是 ExtUtils::MakeMaker，我们就不用做这么多工作：

```
$ ./Build testcover
```

就像其他的 Devel::模块一样，Devel::Cover 会监测代码的执行，并有很多奇妙的方法来判定发生的事情。它会把所有的信息保存在一个名为 cover_db 的目录下。

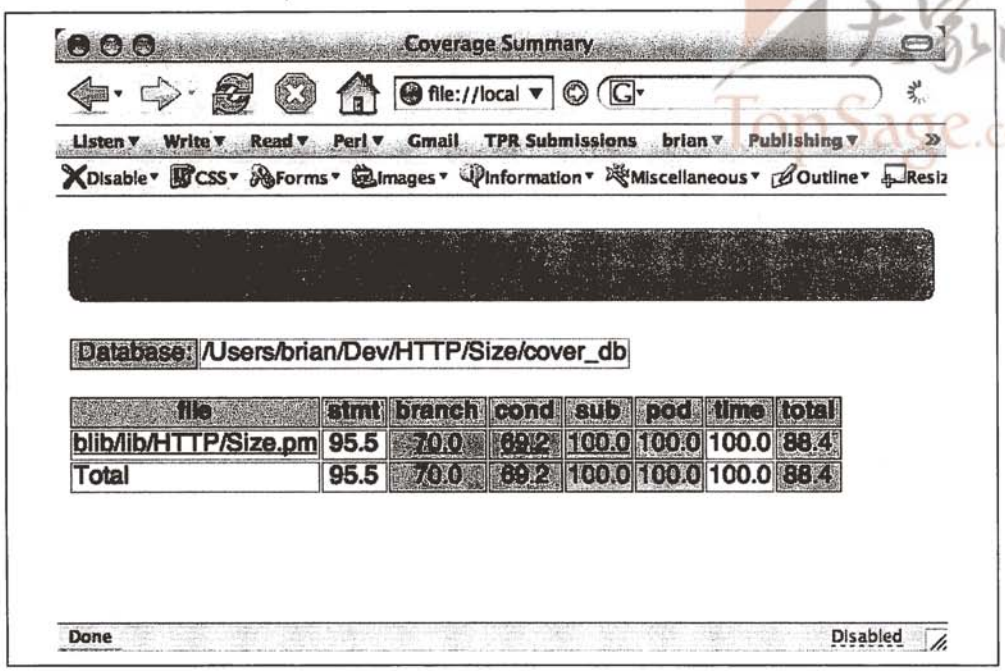


图 5-1：cover 会生成一个 HTML 格式的报告

最后，`cover` 命令把所有的数据都转化成易读的格式，把报告的摘要发送到 STDOUT，并把详细的报告保存在 *coverage.html* 中。文件 HTML 会链接到很多其他的 HTML 文件，从而可以让我们详细地分析程序。下面我们来分析一下 `HTTP::Size` 模块：

```
$ cover
Reading database from /Users/brian/Dev/HTTP/Size/cover_db

---------------------------- ------ ------ ------ ------ ------ ------ ------
File                          stmt branch   cond    sub    pod   time  total
---------------------------- ------ ------ ------ ------ ------ ------ ------
blib/lib/HTTP/Size.pm        95.5   70.0   69.2  100.0  100.0  100.0   88.4
Total                        95.5   70.0   69.2  100.0  100.0  100.0   88.4
---------------------------- ------ ------ ------ ------ ------ ------ ------
```

摘要告诉我们测试套件运行了模块中 95.5%的代码。此外，我们只测试了所有可能的路径的 70%和所有可能条件组合的 69.2%。不过，这只是一个摘要而已。文件 *coverage.html* 中 HTML 格式的报告（图 5-1）提供了更多的信息。

HTML 格式的报告中每一种覆盖率都有一列相应的数据。它用绿色表示某行代码的某个覆盖率指标是 100%，用红色表示该代码还需要更多的测试（图 5-2）。

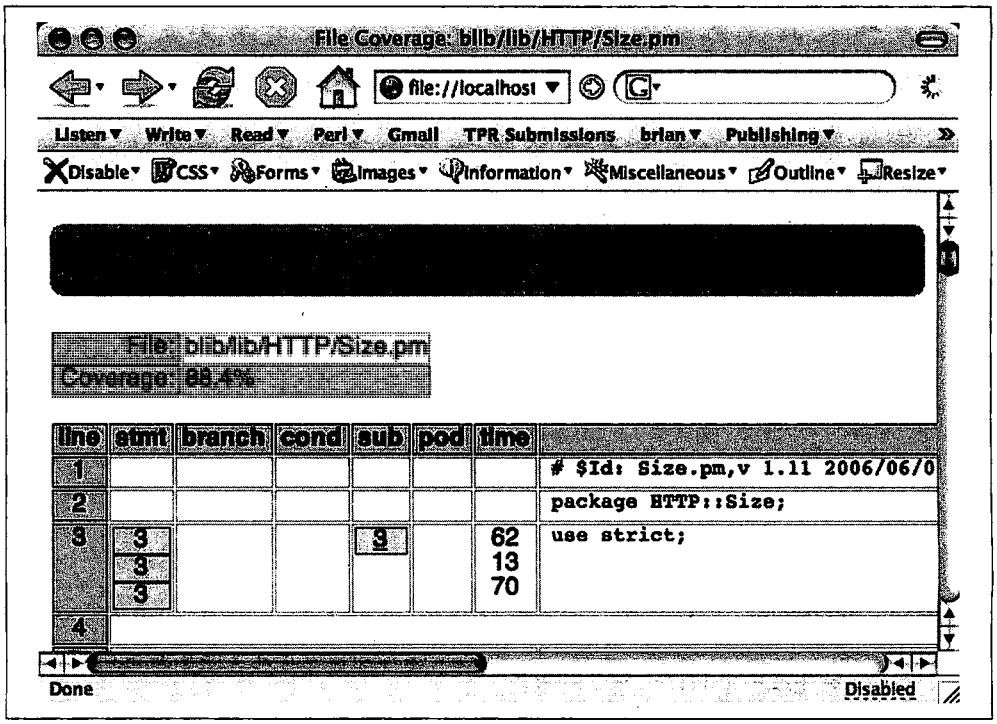


图 5-2：单个文件的覆盖率报告可以告诉我们对该文件的测试是否充分

总结
Summary

在我们确定如何改进 Perl 程序时，须要对程序进行剖析以找到须要重点解决的部分。Perl 的剖析程序只是一种专门化的调试器，如果不喜欢它，也可以实现自己的剖析程序。

深入阅读
Further Reading

文档 *perldebguts* 介绍了如何创建一个调试程序。我在《The Perl Journal》中的文章中更详细地讨论了这些话题："Creating a Perl Debugger"（*http://www.ddj.com/184404522*）和"Profiling in Perl"（*http://www.ddj.com/184404580*）。

"The Perl Profiler"是 Larry Wall、Tom Christiansen 和 Jon Orwant 合著的《Programming Perl, Third Edition》一书中第 20 章的内容。所有试图掌握 Perl 的程序员都应该拥有这本书。

Perl.com 有两篇关于剖析程序的很有意思的文章：Simon Cozens 的“Profiling Perl”（*http://www.perl.com/lpt/a/850*）和 Frank Wiles 的“Debugging and Profiling mod_perl Applications”（*http://www.perl.com/pub/a/2006/02/09/debug_mod_perl.html*）。

Randal L. Schwartz 在《Unix Review》的“Speeding up Your Perl Programs”（*http://www.stonehenge.com/merlyn/UnixReview/col49.html*）和《Linux Magazine》的“Profiling in Template Toolkit via Overriding”（*http://www.stonehenge.com/merlyn/LinuxMag/col75.html*）中详细讨论了如何剖析程序。

第6章
Perl基准测试
Benchmarking Perl

Tony Hoare有句名言——“过早的优化是一切麻烦的根源”。然而，人们往往不知道它的另一层含义——“在97%的时间里，我们不应该做细微的优化”。就是说，除非你别无选择，否则不要榨取细微的性能提升空间。在本章中，我将演示如何分析Perl程序以发现性能瓶颈：在开始优化程序的性能之前，先看看程序究竟干了些什么；一旦找到影响性能的部分，就集中精力解决这部分的问题。

基准测试理论
Benchmarking Theory

基准测试这个词来自勘探人员。它们在某些东西上标上一个物理记号来指示一个已知的海拔，然后用这个标记来确定其他位置的海拔。只有在最初的记号准确的前提下，这些计算出来的海拔才可能是准确的。即使最初的记号是准确的，它也可能改变：比如地壳下沉，地震引起的地壳运动，或者全球变暖导致最终基准——海平面（注1）的改变。基准是相对的，不是绝对的。

对计算机而言，基准测试对一个系统和另一个系统的性能进行比较。评价的指标有很多，包括完成时间、资源占用情况、网络使用或内存使用情况。现在已经有很多工具能够测量Perl以外的部分，所以这里不再赘述。我们希望看看从Perl内部能够发现什么，希望知道一段代码是不是更快或占用内存更少。

测量并提取数字非常容易，我们也很容易相信计算机给我们的数字。这种情况使基准测试变得很危险。不像那些勘探人员，我们没法站在山顶上，用目测来判断我们是比另一座山高还是低。我们不得不仔细分析从基准测试中得到的数字，认真考虑我们所用的方法。

注1：海平面也不是一个好的基准，因为事实上不存在这样的一个平面。潮汐会影响水面高度，地球自转也会使海面倾斜。由于海水的高度不同，世界各地的海平面其实是不一样的。

现在，基准测试没有过去那么流行了。计算机的速度和容量，以及网络带宽已经不像过去那么有限，所以人们认为没有必要拼命节省了。（在大多数情况下，）我们不必为 CPU 时钟付钱，所以我们也不必在意实际使用了多少。至少我们不像过去那样在乎了。毕竟你是在用 Perl，对吧？

所有的测量都可能出现问题。如果我们不理解在测量什么、什么会影响测量，或者这些数字究竟意味着什么，就很容易错误地解读测量的结果。如果我们测量时不够仔细，得到的数字可能会毫无意义。我们可以让计算机为我们做基准测试，但是没法让它代替我们思考。

Perl 程序无法独立运行，它依赖于 Perl 解释器、操作系统和硬件，而所有的这些又依赖于其他的东西。即使用了相同的硬件、不同的 Perl 解释器，甚至是相同版本的 Perl 也会给出不同的结果。我们可能用了不同优化级别的 C 编译器，可能在一个解释器里包含了不同的功能，如此等等。在章末讨论 `perlbench` 时会更多地分析这个问题。

也许你不曾想象过在一个平台开发程序，在另外一个平台部署程序。作为 Stonehenge 的一个顾问，我拜访了很多公司，所以看到了很多不同的情况。很普遍的情况是，开发小组在只有自己使用的一个系统上开发，然后部署到一个非常繁忙的服务器上。那里的 Perl 版本不同、系统不同、负荷的状况也完全不同。服务器空闲时运行速度很快，可当人们都开始使用它的时候，它的速度就会变到慢得不能忍受。一个很好的例子是 CGI 程序，当负荷增加的时候它就会变得很慢，而 `mod_perl` 则有很好的可扩展性。

任何基准测试的结果都仅仅适用于测试时的环境。根据基准测试进行外推也许不会带来麻烦，但外推的结果往往不能反应真实的情况。对我而言，知道在一个特定条件下会发生什么事情的唯一方法就是在那个条件下测试。而且，除了数据之外，还得记录细节。举例来说，只说我用的是 Powerbook G4 的电脑、Mac 10.4.4 版本的系统是不够的。我必须告诉你 Perl 解释器的详细情况、编译文件用的开关（输入 `perl -V` 即可得到相关信息），以及是如何调整系统的。

此外，我们没法在不干扰一个东西的情况下测量它。如果想观察 Perl 的内存管理情况，我们就得用`-DDEBUGGING_MSTATS` 来编译 Perl，这样一来所用的编译器就不同了。尽管现在可以观察内存的使用状况了，很可能它已经使整个程序慢了下来（在本章末介绍 `perlbench` 的时候会证实这一点）。如果添加代码来计算程序的运行时间，就必须得执行那段代码，而

这意味着程序会运行更长的时间。在任何情况下，如果用到了其他模块，Perl 还须要查找代码、载入代码并编译更多的代码。

测量时间
Benchmarking Time

要测量程序运行需要的时间，只须要在程序开始和结束时看看挂钟。这是最简单的方法，也是最初级的方法。这种方法可能在极端条件下也适用。如果可以把程序的运行时间从一个整工作日缩短到几分钟，就不必在乎挂钟够不够精确。

其实我们可以在程序中直接测量时间，不必真的看挂钟，就像这样：

```
#!/usr/bin/perl

my $start = time;

#... 程序的主体部分

my $end   = time;

print "The total time is ", $end - $start;
```

对于一个运行时间短的程序，这种方法只测量了一部分的运行时间。对于 Perl 编译代码所花的时间怎么办？如果用了很多模块，整个过程中很大一部分的时间可能花在了 Perl 开始执行程序之前。Jean-Louis Leroy 为 Perl.com（注 2）写的一篇文章中曾提到一个启动很慢的 Perl FTP 程序。Perl 须要查找 23 个不同的目录才能找到所有 `Net::FTP` 须要加载的模块。这个程序运行时还是很快的，但是启动的时间则相当长。记住每次运行程序时 Perl 都须要编译它（先不提像 `mod_perl` 这样的程序）。如果用了很多模块，每次运行程序的时候 Perl 就要做很多工作来找到并编译它们。

如果要测量整个过程——编译时间和运行时间，可以通过创建一个母程序调用该程序来测量时间，然后通过比较这个结果和运行时间来估计编译时间：

```
#!/usr/bin/perl

my $start = time;

system( "@ARGV" );

my $end   = time;

printf "The whole time was %d seconds", $end - $start;
```

注 2：“一种快速的启动方法”（*http://www.perl.com/lpt/a/2005/12/21/a_timely_start.html*）。

然而，挂钟方法是有问题的。因为系统会进行任务切换，甚至同时执行不同的任务。不能只看看挂钟就知道计算机在程序上花了多少时间。当程序须要等待繁忙的资源或是网络延迟的时候，这个问题就更加严重。在这些情况下，就不能简单地埋怨程序慢了。

为了解决这个问题，大部分 Unix 类系统自带的 time 程序（不是 Perl 内置的）只记录操作系统在程序上所花的时间。你的 shell 可能也有一个能完成这种任务的内置命令（注 3）。

通过命令行告诉 time 程序测量什么程序，它就运行指定的程序并汇报结果。它会把运行时间细分为实际时间、用户时间和系统时间。实际时间就是挂钟时间。其他的两个时间和操作系统如何在系统和程序之间分配任务有关。大部分时候不用在乎它们的差别，只有它们的总和才有意义。

当测试 sleep 程序（不是 Perl 内置的那个）的时候，它所花的时间是它休眠的时间。由于休眠是它做的所有事情，用户时间和系统时间会少得可以忽略不计。以下是 time 程序的输出结果，可能会根据你的版本而有所不同：

```
$ time sleep 5

real    0m5.094s
user    0m0.002s
sys     0m0.011s
```

在内部，time 程序只是调用了标准 C 库提供的 times 函数，而 times 函数提供了所有的统计信息（虽然我们很幸运不再须要为 CPU 时钟付钱）。 Perl 内置的 times 也能做同样的事情。它通过列表返回了 4 个时间：全部的用户时间和系统时间、子进程的用户时间和系统时间。我从结束时间中减去开始时间来得到运行时间：

```
#!/usr/bin/perl

use Benchmark;

my @start = times;

#... 程序的主体部分

my @end   = times;

my @diffs = map { $end[$_] - $start[$_] } 0 .. $#end;

print "The total time is @diffs";
```

注 3：如果你没有这个工具，Perl Power Tools 项目（*http://search.cpan.org/dist/ppt/*）提供了一个 Perl 的实现，很快我也会给出一个自己的实现。

我们甚至也不必亲自做这些计算，因为 Perl 提供的 Benchmark 模块已经为我们实现了这些。当然，这种方法只测量了运行时间：

```
#!/usr/bin/perl

use Benchmark;

my $start = Benchmark->new;

#... 程序的主体部分

my $end = Benchmark->new;

my $diff = timediff( $t1, $t0 );

    print "My program took: " . timestr( $diff ) . "\n";

( $real, $child_user, $child_system ) = @$diff[0,3,4];

# 我十分肯定这是 POSIX 格式的
printf STDERR "\nreal\t%.3f\nuser\t%.3f\nsys\t%.3f\n",
                          $real, $child_user, $child_system;
```

程序的输出和之前 times 程序的输出相似，不过现在它是完全来自我的 Perl 程序，而且只测量 Benchmark->new 之间的部分程序。我们没有测量整个程序，而是把注意力放在了想检查的部分。

这正是 David Kulp 在创建 Perl Power Tools 版本的 time 时所做的：记录下一个基准时间，用 system 命令运行感兴趣的程序，当 system 命令返回时再记录一个基准时间。因为这个版本的 time 是纯 Perl 的，它可以在所有能运行 Perl 的地方运行：

```
#!/usr/bin/perl

use Benchmark;

$t0 = Benchmark->new;

$rc = system( @ARGV );

$t1 = Benchmark->new;

$diffs = timediff( $t1, $t0 );

printf STDERR "\nreal %.2f\nuser %.2f\nsys  %.2f\n", @$diffs[0,3,4];

$rc &= 0xffff;
if ($rc == 0xff00) { exit 127; } else { exit ($rc >> 8); }
```

这种用测量 CPU 时间来比较程序性能的方法有一个大问题：它只测量了程序使用的 CPU

时间，而没有包含等待输入、发送输出或获得其他资源的时间。有些时候这些时间会比 CPU 时间更重要。

比较代码
Comparing Code

基准测试本身不是非常有用。我把它归为决策支持。我们可能会用它来决定要改变一个程序以提高某个指标，但是那个指标本身并没有告诉我们要做什么。当然，我们会知道程序运行花了多少时间，但是它没有告诉我们程序能否变得更快。我们须要把一个实现和另一个实现做比较。

我们可以比较整个程序，但是这往往不是特别有效。如果想提高某个程序的执行速度，须要改进我们认为慢的部分。其他的部分则保持原样，它们所花的时间也是一样的。只须要比较不同的部分。其他部分所花的时间会影响整体的结果，因此我们须要隔离出要比较的部分。

如果我们提取出不同的部分，就可以用它们来创建小的程序。这样这些小程序所花的大部分时间都集中在我们感兴趣的部分。稍后我会详细地讨论这个问题。但是现在须要记住，我们做的任何事情都有额外的开销、所有的测量都会改变本来的状况，所以须要在接受这些数字之前认真地考虑它们。现在，让我们先回到 Benchmark 模块。

如果我们想比较程序的两个小部分而不是整个程序，可以利用 Benchmark 中的一些函数。我们可以把某部分运行一定的次数来比较运行时间，或者反过来，运行固定的时间来比较运行的次数。

在 Benchmark 的 timethese 函数中，第一个参数是迭代的次数，第二个参数是一个匿名的哈希表。哈希表的键是指定的代码片段的名字；哈希表的值是想比较的代码，这里是须要 Perl 用 eval 来计算的字符串。在这个例子中，要比较 opendir 和 glob 在获得文件列表时的速度：

```
#!/usr/bin/perl

use Benchmark;

my $iterations = 10_000;

timethese( $iterations, {
        'Opendir'  => 'opendir my( $dh ), "."; my @f = readdir( $dh )',
        'Glob'     => 'my @f = glob("*")',
        }
        );
```

timethese 函数打印出了一个优美的报告，显示了前面提到的三个时间：

```
$ perl dir-benchmark.pl
 Benchmark:  timing 10000 iterations of Glob, Opendir...
       Glob:  6 wallclock secs ( 2.12 usr + 3.47 sys = 5.59 CPU) @ 1788.91/s (n=10000)
    Opendir:  3 wallclock secs ( 0.85 usr + 1.70 sys = 2.55 CPU) @ 3921.57/s (n=10000)
```

然而，这些不是真正的结果。很多人只测了一次就试图立刻离开。应该重试一次，再试一次。每次运行的结果都有一点点不同，这些只是近似误差：

```
$ perl dir-benchmark.pl
Benchmark: timing 10000 iterations of Glob, Opendir...
       Glob: 6 wallclock secs ( 2.10 usr + 3.47 sys = 5.57 CPU) @ 1795.33/s (n=10000)
    Opendir: 3 wallclock secs ( 0.86 usr + 1.70 sys = 2.56 CPU) @ 3906.25/s (n=10000)

$ perl dir-benchmark.pl
Benchmark: timing 10000 iterations of Glob, Opendir...
       Glob: 7 wallclock secs ( 2.11 usr + 3.51 sys = 5.62 CPU) @ 1779.36/s (n=10000)
    Opendir: 3 wallclock secs ( 0.87 usr + 1.71 sys = 2.58 CPU) @ 3875.97/s (n=10000)

$ perl dir-benchmark.pl
Benchmark: timing 10000 iterations of Glob, Opendir...
       Glob: 7 wallclock secs ( 2.11 usr + 3.47 sys = 5.58 CPU) @ 1792.11/s (n=10000)
    Opendir: 3 wallclock secs ( 0.85 usr + 1.69 sys = 2.54 CPU) @ 3937.01/s (n=10000)
```

不要放弃思考

Don't Turn Off Your Thinking Cap

如果让计算机代替我们思考，基准测试会具有欺骗性。Benchmark 模块可以整天不停地输出结果，但是如果我们不考虑我在做什么、这些数字意味着什么，它们就没有任何意义。它们甚至会让我相信一些错误的东西。我的亲身经历就是一个极好的例子。

Stonehenge 的《Intermediate Perl》中讲到了 Schwartzian 变换，它利用缓存的已排序的键值来避免排序时的重复工作。因此，Schwartzian 变换会更快，尤其是在有很多的元素和复杂的键值计算时。我们在《Intermediate Perl》的第 9 章中讨论过这些。

在一道习题中，为了向学生证明变换提升了性能，我们让学生根据文件的修改时间排列文件。查找修改时间是一个耗时的操作，尤其是当我不得不重复 N*log(N) 次的时候。由于得到了想要的结果，我们没有进行彻底地研究。

我们过去在教程的材料中提供的答案其实不是最好的。它很简短，可以在一张幻灯片上放下，但是它让结果比真实的情况差了很多。Schwartzian 变换如期望的那样排在了最前面，但是我一直认为它应该更快。

在示例中，我们用 Benchmark 的 timethese 来比较两种根据修改时间排列文件的方法。一种“Ordinary”的方法在每次须要比较的时候都要调用-M $a 计算修改时间。Schwartzian 方法则用 Schwartzian 变换来计算，每个文件只计算一次，计算的结果和文件名保存在一起。这是一个缓存了键值的排序：

```
use Benchmark qw{ timethese };
timethese( -2, {
        Ordinary      =>
                q{ my @results = sort { -M $a <=> -M $b } glob "/bin/*"; },
        Schwartzian =>
             q{ map $_->[0], sort { $a->[1] <=> $b->[1] } map [$_, -M], glob "/bin/*"; },
        });
```

这段代码有很多的问题。如果要比较两个东西，应该把它们变得尽可能相似。注意在“普通”方法中我们把结果赋给了@results，在 Schwartzian 方法的一个空上下文中用了 map()。它们做的事情是不同的：一个是一次比较和存储，一个只有比较。为了比较这两种方法，它们须要生成相同的东西。在这个例子中，它们都应该保存计算的结果。

此外，须要隔离不同的部分、抽出相同的部分。在每个代码串中，我们用到了 glob()，而我们知道这是个耗时的操作。glob()破坏了测试结果，因为它给两种排序（译注 1）添加了额外的时间。

在一堂课上，当学生们在做练习的时候，我也做了我的作业：按照基准测试的标准流程重写了我们的基准测试。

我把整个任务分解成若干个部分，分开测量来确定它们对整体的影响。总共有三个部分须要测量：创建文件列表，对文件进行排序，把排序后的列表赋给另一个变量。既要分别测量每个部分，也要测量整个任务。这看起来是个很简单的任务，就是比较两部分代码而已，但是如果我不小心的话会有很多种把事情弄糟的可能。

我也想看看改进之后的程序和课程幻灯片中的例子相比有多少提高，所以把原来的代码也包含了进来。我还尝试了各种不同的方法看看每部分对最终结果有多少贡献：列表赋值花了多少时间，通过 glob()生成文件列表花了多少时间。这样，我从各种公共部分得到了一堆代码串。

首先，我声明了一些软件包公用的变量。Benchmark 会把我的代码串转化为子过程，我需要这些子过程来找到这些变量，所以它们必须是全局（整个软件包）的变量。虽然我知道 Benchmark 会把这些子过程放在 main::包中，我用了 L::*，也就是局部（Local）的缩写。我这样做并不是很重要。相比之下，更重要的是能把公共部分抽象出来以尽可能减少对结果的影响。

$L::glob 变量是我希望 glob 用的，从@ARGV 得到该变量，这样我就可以指定不同的目录

译注 1：作者在这里用英文开了一个小小的玩笑，原文是：it adds to the time for the two sorts of, um, sorts。

来观察文件数目对结果的影响了。我只给$L::glob进行了一次赋值，之后所有的地方都用glob()。这样，所有的代码串都会得到相同的文件列表。我期望Schwartzian变换在文件数目增大的情况下表现会越来越好。

我也想看看没有glob()的情况，所以预先把glob()得到的结果保存在了@L::files中。我认为glob()会明显地影响时间，所以要看看有它和没有它的结果。

我们在匿名哈希数组$code中保存测试用的代码串。我希望测试各个部分和整体，所以我在代码串的开头加上了一些控制串：把文件列表赋给一个变量；运行glob()。Benchmark模块在内部也会运行一个空的子过程，这样也就排除了子进程的开销。我觉得赋值需要的时间是微不足道的，而glob()会花掉很多时间。一开始的时候，我怀疑glob()会占据整个测试1/3的时间，虽然这只是一个猜测而已。

接下来的一组代码串测量排序的时间。sort_names运行在一个空上下文（empty context）中，sort_names_assign也是的，只是它把结果传给了一个数组。我觉得两者的差别应该能测量到，而且应该等于赋值需要的时间。

然后我加入了练习题中的原来那段代码，我把它叫做oridinary_orig。它用了glob()，我觉得这会明显地增加执行时间。oridinary_mod代码则用了@L::files中的文件名列表，这个列表和glob()得到的是一样的。我认为这两者的差别完全来自glob()。

最后的一组代码串比较了3个东西：schwartz_orig是最早的代码。在schwartz_orig_assign中，我把结果赋给了一个数组，就像其他的代码那样。（因为如果我要比较两个东西，它们就必须完全一样。）最后，在schwartz_mod中，我去掉了glob()：

```
#!/usr/bin/perl
# schwartzian-benchmark.pl
 use strict;
 use Benchmark;

$L::glob = $ARGV[0];
@L::files = glob $L::glob;

 print "Testing with " . @L::files . " files\n";

my $transform = q|map $_->[0], sort { $a->[1] <=> $b->[1] } map [ $_, -M ]|;
my $sort      = q|sort { -M $a <=> -M $b }|;

my $code = {
        assign              =>  q| my @r = @L::files |,
        'glob'              =>  q| my @files = glob $L::glob |,
        sort_names          =>  q| sort { $a cmp $b } @L::files |,
        sort_names_assign   =>  q| my @r = sort { $a cmp $b } @L::files |,
        sort_times_assign   => qq| my \@r = $sort \@L::files |,
```

```
        ordinary_orig         => qq| my \@r = $sort glob \$L::glob |,
        ordinary_mod          => qq| my \@r = $sort \@L::files |,
        schwartz_orig         => qq| $transform, glob \$L::glob |,
        schwartz_orig_assign  => qq| my \@r = $transform, glob \$L::glob |,
        schwartz_mod          => qq| my \@r = $transform, \@L::files |,
};

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
print "Timing for 2 CPU seconds...\n";
timethese( -2, $code );

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
my $iterations = 1_000;
print "\n", "-" x 73, "\n\n";
print "Timing for $iterations iterations\n";

timethese( $iterations, $code );
```

Benchmark 模块本身提供了报表。为了让结果更加易读，我对它略加修改（去掉了某些输出，缩短了某些行）。测试的结果并不让人吃惊。我非常希望让学生们看看，它们并没有浪费一个小时听我说 Schwartzian 变换有多么棒：

```
$ perl benchmark

Testing with 380 files Timing for 2 CPU seconds...
 Benchmark: running assign, glob, ordinary_mod, ordinary_orig,
        schwartz_mod, schwartz_orig, schwartz_orig_assign, sort_names,
        sort_names_assign for at least 2 + CPU seconds...
assign: (2.03 usr + 0.00 sys = 2.03 CPU) (n= 6063)
glob: (0.81 usr + 1.27 sys = 2.08 CPU) (n= 372)
ordinary_mod: (0.46 usr + 1.70 sys = 2.16 CPU) (n= 80)
ordinary_orig: (0.51 usr + 1.64 sys = 2.15 CPU) (n= 66)
schwartz_mod: (1.54 usr + 0.51 sys = 2.05 CPU) (n= 271)
schwartz_orig: (1.06 usr + 1.03 sys = 2.09 CPU) (n= 174)
schwartz_orig_assign: (1.20 usr + 0.87 sys = 2.07 CPU) (n= 156)
sort_names: (2.09 usr + 0.01 sys = 2.10 CPU) (n=3595626)
sort_names_assign: (2.16 usr + 0.00 sys = 2.16 CPU) (n= 5698)

---------------------------------------------------------------------------

Timing for 1000 iterations Benchmark: timing 1000 iterations of assign,
        glob, ordinary_mod, ordinary_orig, schwartz_mod, schwartz_orig,
        schwartz_orig_assign, sort_names, sort_names_assign ...
assign: 1 secs ( 0.33 usr + 0.00 sys = 0.33 CPU)
glob: 6 secs ( 2.31 usr + 3.30 sys = 5.61 CPU)
ordinary_mod: 28 secs ( 5.57 usr + 21.49 sys = 27.06 CPU)
ordinary_orig: 34 secs ( 7.86 usr + 24.74 sys = 32.60 CPU)
schwartz_mod: 8 secs ( 5.12 usr + 2.47 sys = 7.59 CPU)
schwartz_orig: 12 secs ( 6.63 usr + 5.52 sys = 12.15 CPU)
schwartz_orig_assign: 14 secs ( 7.76 usr + 5.41 sys = 13.17 CPU)
sort_names: 0 secs ( 0.00 usr + 0.00 sys = 0.00 CPU)
sort_names_assign: 0 secs ( 0.39 usr + 0.00 sys = 0.39 CPU)
```

sort_names 排在了最前面。它几乎每秒钟运行两百万次。其实它什么也没有做，因为它是在一个空上下文中。它运行得非常快，而且无论我在 sort() 块中放入什么都是那么快。空上下文中的 sort() 永远是最快的。然而，空上下文中的 sort() 和 map() 的差别在 schwarts_orig 和 schwartz_orig_assign 中不是那么明显，因为只有最后一个 map 是在空上下文中。对于 schwarts_orig 和 schwartz_orig_assign 而言，只有计算完最右边的 map() 和 sort() 之后才能对空上下文进行优化。schwatz_orig 能够执行的循环数目比 schwatz_assign 多了大约 10%，可见省略赋值操作给它带来了一点明显的但不可靠的性能提升。

再看看第二组结果，这次须要用挂钟时间来比较，即使它们没有 CPU 时间精确。记住 CPU 时间是花在 CPU 上的时间，而我在这里花了很多时间在文件系统上。所以 CPU 时间在这里没有挂钟时间准确。

glob 花了 6 秒，schwartz_orig_assign 花了 14 秒。glob 超过了总时间的 1/3！如果从 14 里减去 6，我得到的是 schwartz_mod 的挂钟时间。正如期望的那样，schwartz_mod 恰好是 8 秒。oridinary_* 也要减去 6 秒，不过是从 34 秒到 28 秒，从比例上来看没有那么严重。

我用更多的文件重复了同样的基准测试。随着文件数目的增加，我们应该看到 Schwartzian 变换越来越好。在下面的比较中，会采用实际 CPU 时间，因为近似误差已经大了很多。

873 个文件

尝试 873 个文件是因为我有一个目录有那么多的文件。注意 glob() 依然占了大量的时间，在空上下文中的变换依然耗了10%的时间。ordinary_mod 和 schwartz_mod 的比例是 73 / 20 = 3.7，比前面的结果（译注 2）稍微高了一些，不过比 ordinary_orig 和 schwartz_orig 的 2.9 要好很多：

```
Benchmark: timing 1000 iterations of glob, ordinary_mod, schwartz_mod...
glob: 14 secs ( 6.28 usr + 8.00 sys = 14.28 CPU)
ordinary_mod: 73 secs (14.25 usr + 57.05 sys = 71.30 CPU)
ordinary_orig: 93 secs (20.83 usr + 66.14 sys = 86.97 CPU)
schwartz_mod: 20 secs (14.06 usr + 5.52 sys = 19.58 CPU)
schwartz_orig: 32 secs (17.38 usr + 13.59 sys = 30.97 CPU)
schwartz_orig_assign: 34 secs (19.95 usr + 13.60 sys = 33.55 CPU)
```

3 162 个文件

空闲的 CPU 时间其实是浪费掉的 CPU 时间。可我宁可有一个空闲的 CPU 也不愿让别人再做一次这个测试。运行这个测试的时候，我的磁盘转了很久很久。ordinary_mod 和 schwartz_mod 的比例是 675 / 151 = 4.5，明显 Schwarzian 变换要好很多。但是

译注 2：前面的结果是 28 / 8 = 3.5。

ordinary_orig 和 schwartz_orig 的比例是 2.8，远小于前者。看来错误的基准测试缩小了本来的差距。这是不应该发生的！

看看 glob() 的巨大损耗！glob() 几乎占用了和变换一样的时间。如果我还是用原来的方法，学生们会认为 Schwarzian 变换并没有什么了不起：

```
Benchmark: timing 1000 iterations of glob, ordinary_mod, schwartz_mod...
glob: 148 secs ( 31.26 usr + 102.59 sys = 133.85 CPU)
ordinary_mod: 675 secs ( 86.64 usr + 517.19 sys = 603.83 CPU)
ordinary_orig: 825 secs (116.55 usr + 617.62 sys = 734.17 CPU)
schwartz_mod: 151 secs ( 68.88 usr + 67.32 sys = 136.20 CPU)
schwartz_orig: 297 secs ( 89.33 usr + 174.51 sys = 263.84 CPU)
schwartz_orig_assign: 294 secs ( 96.68 usr + 168.76 sys = 265.44 CPU)
```

内存使用
Memory Use

当一个程序员提到基准测试的时候，很有可能指的是速度。不管怎么样，Perl 的 Benchmark 模块所测量的和大部分关于基准测试的文章讨论的都是速度。时间是一个容易测量的东西，人们测量能够测量的东西，这也是可以理解的——虽然不一定能够测量正确。不过，有的时候时间不是主要的限制因素。有些其他的因素，比如内存会成为主要的问题。

Perldebguts 文档中曾经提到：

> 对于估计 Perl 的内存使用有这样的一种观点：假设一个比较合理的内存使用算法，把它的估计值乘以 10 就是 Perl 的使用状况。这样即使实际情况仍然超过了估计值，你也不会非常惊讶了。

在设计上，Perl 用内存开销换取了处理速度的提高。Perl 做了大量的查找工作，而不是计算。高级语言会自动进行内存管理以便程序员更多地思考手头的问题，而不是考虑如何获得内存、释放内存或是引入内存管理的 bug（注 4）。

不过，这种易用性是有代价的。由于我们不控制内存，Perl 也不会事先知道我们打算干什么，所以 Perl 只有去猜。当 Perl 需要更多的内存时，它会申请一大块内存。当我们需要内存时，很可能之后还需要更多的，所以 Perl 一次会多申请一些。如果仔细观察程序的内存使用情况，就会发现它每次都有个很大的跳跃，之后停留一段时间，然后又是一个跳跃。Perl 也不会把内存还给操作系统。如果它之前需要内存，可能之后还会需要。它会重用不再使用的内存，不过它也会保留得到的所有内存。

此外，Perl 是构建在 C 之上的，可是它没有 C 的数据类型。Perl 有标量、数组、哈希表，等等。Perl 没有把真实的存储情况暴露给我们，所以我们从来不用去考虑它。不仅如此，

注 4：详尽地介绍 Perl 内存管理机制需要一整本书的篇幅。在这里我只介绍一些基本的东西，有兴趣深入了解的读者可以参阅 *perldebguts*。

Perl 还得去处理上下文。这些数据是字符串，还是数字，还是都有可能？数组的元素放在内存的什么地方？还有什么其他的变量会引用它？

在 Perl 中，一个数远远不止一个数那么简单。它就像一位有着宏大配景的电影明星。它可能是一个 32 位的整数，不过其实它有 12 个字节。我们可以用 Devel::Peek 模块检查变量的数据结构从而了解 Perl 是如何存储变量的：

```
#!/usr/bin/perl

use Devel::Peek;

my $a;

print_message( "Before I do anything" );
Dump( $a );

print_message( "After I assign a string" );
$a = '123456789';
Dump( $a ),

print_message( "After I use it as a number" );
$b = $a + 1;
Dump( $a );

sub print_message
        {
        print STDERR "\n", "-" x 50,
                "\n$_[0]\n", "-" x 50, "\n"
        }
```

从输出中可以看到，Perl 在程序的每个地方都在跟踪这个标量。当我们第一次创建该变量的时候，还没有给它赋值。我们可以看到 Perl 创建了该标量（内部的说法是 SV，或者是“scalar value”），把引用计数设定成 1，还设置了一些标志位。SV 中没有任何东西（值为 NULL(0x0)），不过它有一个地址，0x1808248。这样整个标量的基础设施就设置好了，可以随时接受赋值。

当我们把一个字符串赋给$a 时，它就拥有了更多的符号位，而且有了一个 PV，也就是“指针值”，其实就是说它是一个字符串（对于熟悉 C 的读者来说是 char *）。现在，标量值指向了字符串数据。

当我们第一次把这个标量当作数字使用时，Perl 必须把它转化成一个数。之后，Perl 也会把数值保存下来，这样标量就变成了 PVIV，这意味着它有一个指针值和整数值。Perl 也会设置更多的标志位以表示它完成了转换，并且有两种值。这样下次就可以直接使用数字了：

```
--------------------------------------------------
Before I do anything
--------------------------------------------------
SV = NULL(0x0) at 0x1808248
  REFCNT = 1
```

```
  FLAGS = (PADBUSY,PADMY)

-------------------------------------------------
After I assign a string
-------------------------------------------------
SV = PV(0x1800908) at 0x1808248
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY,POK,pPOK)
  PV = 0x301c10 "123456789"\0
  CUR = 9
  LEN = 10

-------------------------------------------------
After I use it as a number
-------------------------------------------------
SV = PVIV(0x1800c20) at 0x1808248
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY,IOK,POK,pIOK,pPOK)
  IV = 123456789
  PV = 0x301c10 "123456789"\0
  CUR = 9
  LEN = 10
```

从上面可以看到 Perl 做了大量的工作。每个 Perl 变量都有很多的额外开销，即使它没有赋值。不过这也没有什么问题，因为 Perl 的这些开销非常值得。

`Devel::Size` 模块会告诉我们每个变量占了多少内存空间。不过，必须记住，实际使用的空间可能还会多一些，因为 Perl 必须把值和字节边界对齐。它不能把一个值保存在任何一个起始位置：

```
#!/usr/bin/perl

use Devel::Size qw(size);

my $n;

print_message( "Before I do anything" );
print "Size is ", size( \$n );

print_message( "After I assign a string" );
$n = '1';
print "Size is ", size( \$n );

print_message( "After I use it as a number" );

my $m = $n + 1;
print "Size is ", size( \$n );

sub print_message { print "\n", "-" x 50, "\n$_[0]\n", "-" x 50, "\n" }
```

可以看到，在做任何事情之前，我们的标量`$n`就至少占据了 12 字节。当我们把一个字符串

赋给它的时候，标量占用的空间又变大了一些，稍稍超过了字符串的长度。Perl 用一个空字符标记了字符串的结尾，而且可能还会分配一些额外的空间以免字符串以后变长。当我们把字符串作为数字使用时，Perl 还会保存数字的版本，所以变量会变得更大。所有的这些都会占用一些意料之外的内存空间：

```
------------------------------------------------
Before I do anything
------------------------------------------------
Size is 12

------------------------------------------------
After I assign a string
------------------------------------------------
Size is 26

------------------------------------------------
After I use it as a number
------------------------------------------------
Size is 31
```

那么引用呢？引用也是标量，它只须要知道在哪里找到数值，不须要自己保存数值。即使值发生了变化，它们的大小也不会变化。引用的大小永远不会变化。不过，使用 `Devel::Size` 的时候须要小心。如果传给它一个引用，它会找出引用指向的变量的大小。当我们用引用指向数组或哈希表时，这是一件好事。不过，如果我们用引用指向另一个引用的话，二级引用的大小是引用指向的内容的大小，而它所指向的内容只是一个引用：

```
#!/usr/bin/perl

use LWP::Simple;
use Devel::Size qw(size);

# watch out! This is 50+ MB big!
my $data = get( "http://www.cpan.org/src/stable.tar.gz" );

print "The size of the data is " , size( $data ), "\n";

my $ref = \$data;

print "The size of the reference is " , size( $ref ), "\n";

my $ref2 = \$ref;

print "The size of the second reference is " , size( $ref2 ), "\n";
```

输出中显示二级引用的大小只有 16 个字节。它并没有包括保存在最终的标量中的数据。待会我会解释为什么须要明白这一点。不过现在我们须要先看看 Perl 的容器：

```
The size of the data is 12829217
The size of the reference is 12829217
The size of the second reference is 16
```

对于 Perl 的容器来说，情况就不一样了。数组是标量的集合，哈希表中有标量的键和标量的值。这些标量有可能是普通的保存有值的标量，也有可能是引用。Devel::Size 模块的 size 函数会告诉我们数据结构的大小。记住，引用可能指向很大的数值，但是它们本身不会占用多少空间：

```
#!/usr/bin/perl

use Devel::Size qw(size);

my @array = ( 1 ) x 500;

print "The size of the array is ", size( \@array ), "\n";
```

我们可以看到数组占用的空间。Devel::Size 的文档很小心地说明了该模块统计的不是数组中元素的大小，而是数组本身的大小。注意，含有 500 个元素的数组的大小要远远超过 500 个 16 字节的标量：

```
The size of the array is 2052
```

即使如此，这个数字并没有包含数组中的内容。不管标量是什么，数组占用的大小都是一样的：

```
#!/usr/bin/perl

use Devel::Size qw(size);

my $data  = '-' x 500;
print "The size of the scalar is ", size( $data ), "\n";

my @array = ( $data ) x 500;
print "The size of the array is ", size( \@array ), "\n";
```

上面的例子中，我们创建了一个 500 个字符的标量，加上一些额外的开销，整个标量会占用 525 字节。而数组占用的空间和前面相同：

```
The size of the scalar is 525
The size of the array is 2052
```

对于这个问题，Devel::Size 提供了一个解决方案。我们须要检查容器中的所有标量，得到它们的大小。引用可能会指向别的容器，而这个容器又可能包含更多的引用。一个数组可能会看起来非常小，但是当我们试图进行一个深拷贝、保存它或者须要把所有的数据都保存在一个地方的时候，情况就会大不一样了：

```
#!/usr/bin/perl

use Devel::Size qw(size total_size);

my $data  = '-' x 500;
print "The       size of the scalar is ", size( $data ), "\n";
print "The total size of the scalar is ", total_size( $data ), "\n";
```

```
print "\n";

my @array = ( $data ) x 500;
print "The      size of the array is ", size( \@array ), "\n";
print "The total size of the array is ", total_size( \@array ), "\n";
```

使用 total_size 后，标量的大小保持不变，而数组的大小会包含所有标量。数组的总体大小 264 522 是 525 的 500 倍再加上 2 052，也就是数组的大小：

```
The       size of the scalar is 525
The total size of the scalar is 525

The       size of the array is 2052
The total size of the array is 264552
```

不过，我们须要知道这个数字的真实含义。它只是数组中所有指向的元素的大小之和。如果我们按这样来把所有的数据结构加起来的话，可能得到的并不是程序的内存大小，因为有些数据结构可能会指向相同的数据。

perlbench 工具
The perlbench Tool

对于不同的 perl 二进制程序，同样的代码表现可能会有所不同。差别可能来自于编译选项、使用的编译器、包含的功能，等等。比如说，多线程版本的 Perl 要比共享库版本的稍微慢一点。如果你须要得到一些别的好处的话，慢一些不一定是坏事，不过有的时候你不一定能够选择使用的版本。比如说，很多 Linux 发行版自带的 Perl 是多线程版本的。如果你觉得使用一个单线程的解释器有可能加快程序速度的话，须要在重新配置系统之前找到充分的证据。

为了比较不同的 perl 解析器，Gisle Aas 编写了 perlbench。我们把希望测试的解析器的路径传给它后，它就会运行一系列的测试，最后生成一个报告。Perlbench 的发布版带有一个 perlbench-run 命令，给定须要测试的 perl 解析器的地址后，它就会用它运行一系列的基准测试。程序会根据第一个指定的解析器对时间进行归一化处理：

```
perlbench-run /usr/local/bin/perl5*
```

输出会首先显示每个测试的解释器的详细信息，并给它们分配一个字母。字母一一对应于接下来输出的表格中的各列。其中特别有意思的是 ccflags。在这次执行中，我用的是以 -DDEBUGGING_MSTATS 选项编译的 perl-5.8.7。另一个有意思的地方是编译器的信息。看上去我的 gcc 编译器版本非常老。这可能是个好事，也可能是个坏事。不同的 Perl 版本，甚至不同的编译器在优化代码方面有的做得好，有的做得差。这些数字只有在一台机器上相互比较才有意义：

```
A) perl-5.6.1
        version      = 5.006001
        path         = /usr/local/bin/perl5.6.1
```

```
        ccflags       = -fno-strict-aliasing -I/usr/local/include
        gccversion    = 2.95.2 19991024 (release)
        optimize      = -O
        usemymalloc   = n

B) perl-5.8.0
        version       = 5.008
        path          = /usr/local/bin/perl5.8.0
        ccflags       = -DHAS_FPSETMASK -DHAS_FLOATINGPOINT_H -fno-strict-aliasing
        gccversion    = 2.95.2 19991024 (release)
        optimize      = -O
        usemymalloc   = n

C) perl-5.8.7
        version       = 5.008007
        path          = /usr/local/bin/perl5.8.7
        ccflags       = -DDEBUGGING_MSTATS
        gccversion    = 2.95.4 20020320 [FreeBSD]
        optimize      = -g
        usemymalloc   = y

D) perl-5.8.8
        version       = 5.008008
        path          = /usr/local/bin/perl5.8.8
        ccflags       = -DHAS_FPSETMASK -DHAS_FLOATINGPOINT_H -fno-strict-aliasing
        gccversion    = 2.95.4 20020320 [FreeBSD]
        optimize      = -O
        usemymalloc   = n
```

perlbench-run 汇报完解析器的详细信息后，会用每个解析器运行一系列的 Perl 程序。它会测量执行时间，就像 Benchmark 的 timethese 一样。当它运行完所有的解析器后，会对结果进行归一化，第一个解析器（也就是标记为 A 的）的时间为 100。其他列中小于 100 的就意味着该解析器运行该测试比较慢，大于 100（可以超过 100）的意味着运行该测试比较快。这个数值只对该测试有意义，我们没法把它和其他的测试进行比较，哪怕是同一批执行的。

我去掉了输出结果中的部分内容，不过下面的这个表会让你对比较有所了解。以 -DDEBUGGING_MSTATS 编译的解析器 C 始终比其他的解析器慢。对于某些测试，有时 Perl 5.6.1 要快些，有时 Perl5 .8.8 要快些。这并不是一个普遍的现象，因为我们只用它们编译了一小部分代码。不过，总体上看来，Perl 5.6.1 是最快的。但是，这也不意味着我一定会选择它，因为 Perl 5.8 有很多很棒的功能，只是速度上稍微慢一点：

	A	B	C	D
arith/mixed	100	85	73	79
arith/trig	100	87	82	81
array/copy	100	99	81	92
array/foreach	100	93	87	99
array/shift	100	100	94	91

```
array/sort-num          100     89      92      151
array/sort              100     95      80      94
call/0arg               100     107     79      91
call/1arg               100     92      69      78
call/wantarray          100     95      76      80
hash/copy               100     130     94      124
hash/each               100     119     90      110
hash/foreach-sort       100     103     78      102
loop/for-c              100     102     88      104
loop/for-range-const    100     101     94      106
loop/for-range          100     100     94      104
re/const                100     92      81      88
string/base64           100     86      67      72
string/htmlparser       100     91      75      74
string/tr               100     105     51      111

AVERAGE                 100     97      80      91

Results saved in file:///home/brian/perlbench-0.93/benchres-002/index.html
```

如果我们有一些特别的东西须要测试的话，也可以添加自己的测试文件。大部分的基础设施都已经就位了。perlbench 的 *README* 文件介绍了测试文件的基本格式。我们可以创建一个自己的测试文件并把它放在 perlbench 的 *benchmark* 目录下。perlbench 的发布版也提供了一个样例文件：

```
# Name: My name goes here
# Require: 4

require 'benchlib.pl';

# YOUR SETUP CODE HERE
$a = 0;

&runtest(100, <<'ENDTEST');
        # YOUR TESTING CODE HERE
ENDTEST
```

总结
Summary

基准测试是一个技巧性很强的话题。它涉及了大量的数据，须要对真正发生的事情有很好的理解。我们不仅须要分析 Perl 程序，还须要考虑其他的因素，比如选择的操作系统、使用的 Perl 解析器、编译解析器的方式，以及其他一切可能影响程序的因素。而且，基准测试不仅限于速度。我们可能须要比较两种方法的内存使用情况，或者看看哪种方法占用了更多的带宽。不同的情况下有不同的制约因素。不管我们在做什么，我们须要在决定如何改进程序之前尽最大努力弄明白正在发生的事情。

深入阅读
Further Reading

模块 Benchmark 提供了详细的使用说明。该模块随 Perl 一起发布，所以你应该已经有它了。

在“A Timely Start”中，Jean-Louis Leroy 发现他的 Perl 程序运行很慢的原因是它花了很多的时间查找须要加载的模块：*http://www.perl.com/lpt/a/2005/12/21/a_timely_start.html*。

在“When Perl Isn't Quite Fast Enough”中，Perl 开发者 Nick Clark 介绍了为什么程序一般会很慢，并且解释了设计上导致 Perl 变慢的原因。他的演讲最精彩的部分——最早是在 YAPC::EU 2002 上，是他加快程序速度的解决方案。我在 PerlWhirl 2005 上听过他的演讲，他的很多探讨都和他加快 Perl 处理 Unicode 的速度的工作有关。如果你有机会参加他的演讲会，一定要去听听！ 我想你一定会感到既轻松愉快又受益匪浅。

关于评测 Schwartzian Transform 的内容，最初是我为 Perl monks 的名为“Wasting Time Thinking about Wasted Time”的文章写的。我几乎原封不动地把它从原来的帖子里摘了过来 *http://www.perlmonks.org/index.pl?node=393128*。我现在在 Stonehenge 的 Perl 课程上还在用这个帖子说明即使专家也可能在基准测试上犯错误。

我写的第二篇关于 Perl 的文章是为《The Perl Journal》第 11 期写的“Benchmarking Perl”。在这篇文章中，我介绍了 Benchamark 的其他一些函数 *http://www.pair.com/comdog/Articles/benchmark.1_4.txt*.

当我写这本书的时候，perlbench 并没有被 CPAN 索引，不过你依然可以通过 CPAN Search 找到它：*http://search.cpan.org*。如果需要它的说明文档，请阅读它的 *README* 文件。

第 7 章

清理 Perl 程序

Cleaning Up Perl

掌握 Perl 的一个要求是管理好源代码，无论源代码来自何处。人们通常能够阅读自己写的代码，而对别人写的代码抱怨不休。在本章中，我们将把那些总让人抱怨的代码变得清晰易读。我们将要处理的代码也包括 Perl 扰码程序（Perl obfuscator）的输出结果。Perl 扰码程序的大部分工作只是简单地去掉空格。你是程序员，它是源代码，你应该让它知道谁说了算。

好的风格

Good Style

我不会给出任何有关代码风格、括号位置或是在哪里放多少空格的建议。这些问题往往是那些对工作没有任何帮助的白热化争论的导火索。Perl 解析器或计算机根本不在乎代码风格。不过，归根结底，我们写程序是为了满足人的需求。

在我的观念中，好的代码是技术熟练的人能够轻松读懂的。特别须要指出的是，好的代码并不是任何人都可以读懂的代码。新手读不懂并不意味着是代码写得不好。首要的前提是代码的读者需要有该语言的知识；如果没有的话，他们也应该知道在哪里能够找到须要掌握哪些内容。此外，一个好的程序员应该能够轻松地阅读多种主要风格的源代码。

其次，一致性是好代码的一个主要特性。我们不仅应该每次用同样的方法做同样的事情（这可能意味着小组的所有人都要用同样的方法），而且也应该使用同样的格式。当然，也会有特殊情况。不过，在大部分情况下，每次用同样的方法有助于让新的读者明白我们试图做的事情。

最后，我喜欢在代码中大量使用空格。这个习惯早在我的视力开始变差之前就有了。空格可以把 token 分隔开，空行可以把聚在一起的代码块分隔开，就像在写散文一样。本书如果不分段落的话，一定会非常难读，代码也有同样的问题。

我有自己喜欢的独特的代码风格，但是我并不反对使用其他的风格。在编辑代码或为别人的代码提供补丁的时候，我会试着模仿他的风格。记住，一致性是好风格的主要因素。向已有的代码中添加自己风格的代码会破坏代码的一致性。

如果你还没有形成自己的风格或没有任何风格上的限制，文档 *perlstyle* 和 Damian Conway 著的《Perl Best Practices》（O'Reilly）可以帮助你为你或你的编程小组制定一个标准。

perltidy

程序 `perltidy` 可以调整 Perl 程序的格式以使它们更加易读。对于那些缩进风格古怪（或者完全没有缩进）、token 间空格很少或没有、或者用其他方法扰乱的代码，`perltidy` 都能把它们变得清晰易读。

下面是一段我故意用不好的风格写成的程序（注 1）。在不破坏程序正常功能的前提下，我去掉了所有的空格。除此之外，我没有做任何其他扰乱代码的事情：

```
#!/usr/bin/perl
# yucky
use strict;use warnings;my %Words;while(<>){chomp;s{^\s+}{};s{\s+$}{};
my $line=lc;my @words=split/\s+/,$line;foreach my $word(@words){
$word=~s{\W}{}g;next unless length $word;$Words{$word}++;}}foreach
my $word(sort{$Words{$b}<=>$Words{$a}}keys %Words){last
if $Words{$word}<10;printf"%5d  %s\n",$Words{$word},$word;}
```

如果这个程序是别人给我的，我能够弄明白它在干什么吗？我可能会知道程序做了什么，但是不会知道它是怎么工作的。当然，我可以慢慢地分析它，用脑子记住整个过程，也可以在语句间加上换行。这样做是可以的。不过，即使是处理这个小程序也需要大量的工作。

我把这个程序保存在文件 *yucky* 中，然后用默认选项运行 `perltidy` 处理它。`perltidy` 不会覆盖我的文件，它会把调整格式后的文件保存在 *yucky.tdy* 中：

```
$ perltidy yucky
```

下面是 `perltidy` 根据文档 *perlstyle* 的建议对格式进行调整后的结果：

```
#!/usr/bin/perl
# yucky
use strict;
use warnings;
my %Words;
while (<>) {
        chomp;
```

注 1：其实，我是先按正常方式写出程序，然后再把格式弄乱。

```
        s{^\s+}{};
        s{\s+$}{};
        my $line = lc;
        my @words = split /\s+/, $line;
        foreach my $word (@words) {
                $word =~ s{\W}{}g;
                next unless length $word;
                $Words{$word}++;
        }
}
foreach my $word ( sort { $Words{$b} <=> $Words{$a} } keys %Words ) {
        last
          if $Words{$word} < 10;
        printf "%5d  %s\n", $Words{$word}, $word;
}
```

如果我偏爱 GNU 的代码风格的话，也可以使用它的格式。我们给 perltidy 传递一个-gnu 开关：

```
$ perltidy -gnu yucky
```

现在，括号和缩进与之前相比稍微有些不同，它比最初的版本要易读很多：

```
#!/usr/bin/perl
# yucky
use strict;
use warnings;
my %Words;
while (<>)
{
        chomp;
        s{^\s+}{};
        s{\s+$}{};
        my $line = lc;
        my @words = split /\s+/, $line;
        foreach my $word (@words)
        {
                $word =~ s{\W}{}g;
                next unless length $word;
                $Words{$word}++;
        }
}
foreach my $word (sort { $Words{$b} <=> $Words{$a} } keys %Words)
{
        last
          if $Words{$word} < 10;
        printf "%5d  %s\n", $Words{$word}, $word;
}
```

如果想让结果更加精美的话，我们也可以让 perltidy 把程序变成 HTML 格式。选项-html 不会重排程序的格式，但是它会添加 HTML 标记，并对结果应用一个样式表。为了把重排格式后的程序变得更精美，我们把 *yucky.tdy* 转化成了 HTML 格式：

```
$ perltidy yucky
$ perltidy -html yucky.tdy
```

perltidy 还可以做很多其他的事情。它有一些选项可以根据个人喜好对格式进行精细的控制，也有很多选项可以把输出从一个地方发送到另一个地方（包括同页编辑功能）。

去除扰乱
De-Obfuscation

有些人有一种奇怪的观念，认为应该让他们的 Perl 代码难以阅读。有的时候他们这样做是为了隐藏秘密，比如处理授权管理的代码，或者是不希望人们在没有他们许可的情况下分发他们的代码。不管出于什么原因，他们的工作最终都会是竹篮打水一场空。那些不知道如何恢复源代码的人不会为此而烦恼，而那些知道的人只会对这个挑战更加感兴趣。

解开隐藏的源代码
De-Encoding Hidden Source

Perl 代码非常容易做逆向工程，因为不管代码的发布者如何处理它，最终都需要 Perl 来运行它。如果 Perl 能够得到源代码，只须要做一点点工作，我们也可以得到。如果你正在花时间把你的代码从将要得到它的人面前隐藏起来，那么你其实是在浪费时间。

Perl 扰乱程序最喜欢使用的方法也是那些希望在 Perl 扰乱代码大赛（the Obfuscated Perl Contest）中获胜的人们最喜欢使用的方法。它是 Perl 社区为人们试图卖给你的东西举办的比赛，因此 Perl 社区积累了很多方法来恢复源代码。

我先演示一下扰码程序使用的技术。一旦你掌握了这些技术，恢复代码就是很直接的事情了（很麻烦但是还是能够控制的）。我们先从文件 *japhplaintext.pl* 开始：

```
#/usr/bin/perl
# japh-plaintext.pl

print "Just another Perl hacker,\n";
```

我想把这个文件中所有的字符都变成别的字符。我们可以使用 ROT-13，它会把所有的字母右移 13 个位置，如果到达 z 的话会折回到 a。一个真正的扰码程序应该更鲁棒些，须要处理诸如分隔符之类的特殊情况。不过在这里我们不须要考虑它，我们感兴趣的是如何打败那些扰码程序。下面，我们从代码行开始读入文件，并输出编码后的版本：

```
#!/usr/bin/perl
# japh-encoder-rot13.pl

my $source = do {
        local $/; open my($fh),
        $ARGV[0] or die "$!"; <$fh>
```

```
        };

$source =~ tr/a-zA-Z/n-za-mN-ZA-M/;

print $source;
```


我们得到的结果看起来像是某种外星语言：

```
$ perl japh-encoder.pl japh-p*
#/hfe/ova/crey
# wncu-cynvagrkg.cy

cevag "Whfg nabgure Crey unpxre,\a";
```

我们没法运行这个程序，因为它已经不是 Perl 程序了。我们须要在后面添加一些代码把它变回 Perl 代码。为此，我们须要做一个反变换，然后用 `eval` 执行反变换后得到的字符串：

```
#!/usr/bin/perl
# japh-encoder-decoder-rot13.pl

my $source = do {
       local $/; open my($fh),
       $ARGV[0] or die "$!"; <$fh>
       };

$source =~ tr/a-zA-Z/n-za-mN-ZA-M/;

print <<"HERE";
my \$v = q($source);
\$v =~ tr/n-za-mN-ZA-M/a-zA-Z/;
eval \$v;
HERE
```

现在，我们的编码程序具备了解码的代码。一个真正的扰码程序还会去掉空格和其他有助于阅读的东西，不过对于演示来说它已经够用了：

```
$ perl japh-encoder-decoder-rot13.pl japh-plaintext.pl
my $v = q(#/hfe/ova/crey
# wncu-cynvagrkg.cy

cevag "Whfg nabgure Crey unpxre,\a";
);
$v =~ tr/n-za-mN-ZA-M/a-zA-Z/;
eval $v;
```

这就是基本的思路。扰码程序输出的结果必须还是 Perl 代码，为此只须要做一些对源代码进行编码的工作。这可能会简单得像我们的例子一样，也可能会复杂得须要使用某种秘密（比如一个注册码）才能解开代码。有的程序甚至会使用多种变换。下面的这个扰码程序的工作原理和 ROT-13 差不多，不过它工作在整个 8 比特的范围内（所以被称为 ROT-255）：

```
#!/usr/bin/perl
# japh-encoder-decoder-rot255.pl
```

```
my $source = do {
        local $/; open my($fh),
        $ARGV[0] or die "$!"; <$fh>
        };

$source =~ tr/\000-\377/\200-\377\000-\177/;

print <<"HERE";
my \$v = q($source);
\$v =~ tr/\200-\377\000-\177/\000-\377/;
eval \$v;
HERE
```

我们把程序 ROT-13 编码后的结果传给它再次进行编码。得到的输出就像 goobledygook 一样，有些内容甚至无法从屏幕上看到，因为有些 8 位的字符是打印不出来的：

```
$ perl japh-encoder-decoder-rot13.pl japh-p* |
       perl japh-encoder-decoder-rot255.pl -
my $v = q( íù ¤ö ½ ñ¨£¯èæå¯ïöá¯ãòåù£ ÷îãõãùîöáçÒëç®ãùãåöáç ¢xèæç ).
                q( îáâçõòå Ãòåù õîðøòå¬Ùá¢»©»¤ö ½þ ôò¯îúáíÎÚÁÍ¯áúÁÚ¯»).
                q(åöái ¤ö»);
$v =~ tr/-ÿ-/-ÿ/;
eval $v;
```

我们已经演示完了所有的技巧，下面开始恢复源代码。在输出的最后，我们看到了字符串 `eval`。我们把它改成 `print` 来看看结果：

```
my $v = q( íù ¤ö ½ ñ¨£¯èæå¯ïöá¯ãòåù£ ÷îãõãùîöáçÒëç®ãùãåöáç ¢xèæç ).
                q( îáâçõòå Ãòåù õîðøòå¬Ùá¢»©»¤ö ½þ ôò¯îúáíÎÚÁÍ¯áúÁÚ¯»).
                q(åöái ¤ö»);
$v =~ tr/-ÿ-/-ÿ/;
print $v;
```

运行它之后我们得到了第一次编码后的结果：

```
my $v = q(#/hfe/ova/crey
# wncu-cynvagrkg.cy

cevag "Whfg nabgure Crey unpxre,\a";
);
$v =~ tr/n-za-mN-ZA-M/a-zA-Z/;
eval $v;
```

再次把 `eval` 改成 `print`，我们就得到了最初的代码：

```
#/usr/bin/perl
# japh-plaintext.pl

print "Just another Perl hacker,\n";
```

于是我们破解了这种编码方法。不过，这不是唯一的技巧。稍后我会展示更多的技巧。

用 B::Deparse 逆解析代码

Unparsing Code with B::Deparse

这些技术并不都是用来看别人的代码的。有的时候我们不明白 Perl 为什么会做某件事情，为此我们可以先编译该代码，再反编译得到的结果，从而了解 Perl 是如何处理代码的。模块 B::Deparse 会把代码变成 Perl 内部使用的结构，然后再反过来恢复源代码。因为它没有保留任何中间状态，所以输出不会和最初的代码一模一样。

下面的这段代码展示了 Perl 的一个鲜为人知的功能。我们知道在替换操作符中，可以使用其他的分隔符，所以我耍了点小聪明使用点号作为分隔符。为什么它没有按照期望的方式工作呢？我本来希望去掉字符串中间的点的：

```
$_ = "foo.bar";
s.\...;
print "$_\n";
```

可是，点并没有被去掉。消失的是 f 而不是点。我们已经对点做了转义，问题出在哪里呢？使用 B::Deparse 模块后，我们可以看到 Perl 看到的是不一样的代码段：

```
$ perl -MO=Deparse test
$_ = 'foo.bar';
s/.//;
print "$_\n";
test syntax OK
```

转义操作首先把作为分隔符使用的点保护了起来，而没有把它作为模式中的一个普通的点处理。

下面的这个例子来自 Stunnix 的 Perl 扰码程序（注 2）。它通过更换变量名、把字符串变成转义的十六进制数、把数字变成算术运算等方式把代码变得更加难以阅读。它还能使用前一节中所展示的编码技巧，不过在这个例子里没有使用：

```
#!/usr/bin/perl

=head1 SYNOPSYS

A small program that does trivial things.

=cut
 sub zc47cc8b9f5 { ( my ( $z9e1f91fa38 ) = @_ ) ; print ( ( (
"\x69\x74\x27\x73\x20" . ( $z9e1f91fa38 + time ) ) .
"\x20\x73\x65\x63\x6f\x6e\x64\x73\x20\x73\x69\x6e\x63\x65\x20\x65\x70\x6f\x63\x68\x0a"
 ) ) ; } zc47cc8b9f5 ( (0x1963+ 433-0x1b12) ) ;
```

使用 B::Deparse 可以轻易地破解大部分技巧。B::Deparse 的输出会显示出解码后的字符和数字：

```
$ perl -MO=Deparse stunnix-do-it-encoded.pl
sub zc47cc8b9f5 {
```

注 2：Stunnix Perl-obfus（*http://www.stunnix.com/prod/po/overview.sthml*）。

```
        my($z9e1f91fa38) = @_;
        print q[it's ] .  ($z9e1f91fa38 + time) . " seconds since epoch\n";
}
zc47cc8b9f5 2;
```

Stunnix 程序聪明地选择了明显随机的字符串作为标识符的名字。为了解决这个问题，Joshua ben Jore 在 `B::Deobfuscate` 模块中对 `B::Deparse` 进行了扩展。虽然得不到原来的变量名，但是可以得到一些更加容易阅读的名字并对应起来。Joshua 选择使用了一些花卉的名字：

```
$ perl -MO=Deobfuscate stunnix-do-it-encoded.pl
sub SacramentoMountainsPricklyPoppy {
    my($Low) = @_;
    print q[it's ] . ($Low + time) . " seconds since epoch\n";
}
SacramentoMountainsPricklyPoppy 2;
```

此外，`B::Deparse` 能够做的工作也不只这些。你不知道那些只有一行的 Perl 代码在干什么吧？我们可以把`-MO=Deparse` 添加到命令行中，然后观察输出的结果：

```
$ perl -MO=Deparse -naF: -le 'print $F[2]'
```

该模块根据命令行的开关添加了相应的代码。`-n` 选项加上了 `while` 循环，`-d` 选项加上了 `split`，`-F` 把 `split` 使用的模式变成了冒号。`-l` 是我最喜欢的选项之一，它会在 `print` 的末尾自动加上换行符，所以我们得到了`$\ = "\n"`：

```
BEGIN { $/ = "\n"; $\ = "\n"; }
LINE: while (defined($_ = <ARGV>)) {
        chomp $_;
        our(@F) = split(/:/, $_, 0);
        print $F[2];
}
```

Perl::Critic

在《Perl Best Practices》中，Damian Conway 针对如何写出易于阅读和维护的代码这个问题给出了 256 条建议。Jeffrey Thalhammer 把 Damian 的建议和 Adam Kennedy 的 `PPI` 模块（一个 Perl 的解析器）融合在一起创造了 `Perl::Critic` 模块，从而给人们提供了一种查找代码中违反风格的部分的方法。它不仅是一个清理 Perl 代码的工具，还可以让我在开发新的代码时保持诚实。我们不用等到写完代码就可以使用它。我们应该用它频繁地检查自己（和同事）的代码。

安装好 `Perl::Critic` 模块之后（注 3），就可以运行 `perlcritic` 命令了。在下面的例子中我们用默认的参数来测试那个阅读 Use.Perl 期刊的程序。`perlcritic` 会告诉我们每一个

注 3：如果你不想安装它，也可以试试 *http://www.perlcritic.com*。它允许你上传一个文件进行远程分析。

违背规则的地方有什么问题，并给出《Perl Best Practices》中的引用位置和问题的严重程度。数字越大表示问题越严重，其中 5 是最严重的：

```
$ perlcritic ~/bin/journals
Two-argument "open" used at line 105, column 1.  See page 207 of PBP.  (Severity: 5)
Bareword file handle opened at line 105, column 1.  See pages 202,204 of PBP.↵
(Severity: 5)
Integer with leading zeros at line 111, column 29.  See page 58 of PBP.  (Severity: 5)
```

我们可能会非常高兴，因为 perlcritic 只对三个地方给出了警告——稍后我会更详细地讨论这个问题。perlcritic 报告的三个问题中，有两个问题可以立刻得到解决。因为这个程序是我很久以前写的，我没有使用文本形式的文件句柄或三个参数的 open。程序的第 105 行是一个旧式风格的 Perl 代码：

```
Open OUT, "| $Pager";
```

让我们把这行代码改成更好、更现代的形式。我们不得不修改几个地方以使用新的文件句柄名字，不过这没什么大不了的。这些改动去掉了两个警告：

```
Open my($out),  "!-", $Pager;
```

那么第三个警告呢？第 111 行使用了 dbmopen，并使用了八进制数指定文件许可。这并不奇怪，它是帮助文档中提到的第三个参数：

```
dbmopen my %hash, $Counter, 0640 or die $!;
```

在《Perl Best Practices》的第 58 页，Damian 给出的建议是改用 oct：

```
dbmopen my %hash, $Counter, oct(640) or die $!;
```

我不会去改变代码，这样做很傻。不过也没有什么问题，因为在《Perl Best Practices》中，Damian 的意图是让程序员仔细考虑他们所做的事情，形成一套一致、可靠、能让大部分 Perl 程序员理解的编程风格。他提出的最佳实践并不是命令，大部分只是好方法的建议。在多数情况下，使用八进制的文本量并不是 Perl 中的一个严重问题，在这个问题上他有一点吹毛求疵。不过也没什么问题，因为 perlcritic 允许我们修改违背的规则。

每个 Perl::Critic 都是作为一条规则来实现的，每条规则都是一个 Perl 模块，用来检查某种编码规范。在关掉不喜欢的警告之前，我们须要知道对应的是哪个规则。我们可以使用--verbose 选项给 perlcritic 指定输出报告的格式。格式字符串看起来和 printf 使用的类似，占位符%p 表示的是规则的名字。这样，我们就得到了有问题的规则的名字——ValuesAndExpressions::ProhibitLeadingZeros：

```
$ perlcritic --verbose '%p\n' ~/bin/journals
ValuesAndExpressions::ProhibitLeadingZeros
```

如果想了解被违背的规则的更多信息，可以给--verbose 传递一个数字：

```
$ perlcritic --verbose 9  ~/bin/journals
Integer with leading zeros at line 111, column 29.
  ValuesAndExpressions::ProhibitLeadingZeros (Severity: 5)
       Perl interprets numbers with leading zeros as octal. If that's what you
       really want, its better to use `oct' and make it obvious.

         $var = 041;     #not ok, actually 33
         $var = oct(41); #ok
```

一旦知道了规则的名字，我们就可以把它放在 home 目录的*.perlcriticrc* 文件中来关掉它。我们把规则名放在方括号中，并加上-表示分析时不使用该规则：

```
# perlcriticrc
[-ValuesAndExpressions::ProhibitLeadingZeros]
```

再次运行 perlcritic 时，所有的警告都消失了：

```
$ perlcritic --verbose '%p\n' ~/bin/journals
/Users/brian/bin/journals source OK
```

处理完这些之后，我们就可以开始着手解决一些轻微的问题。我们用--severity 开关把严重程度降低一级。就像其他的调试工作一样，我们在处理次要的问题之前先处理最严重的问题。如果一开始就打开比较低的级别，很可能严重的错误会被淹没在大量重复的不重要的错误中，比如说没有在程序中使用 Perl 的警告功能：

```
$ perlcritic --severity 4 ~/bin/journals
Code before warnings are enabled at line 79, column 1.  See page 431 of PBP.?
(Severity: 4)
Code before warnings are enabled at line 79, column 6.  See page 431 of PBP.?
(Severity: 4)
... snip a couple hundred more lines ...
```

我们也可以指定严重级别的名字。表 7-1 显示了 perlcritic 使用的级别。严重级别 4，是最严重的级别的下一个级别，它的名字是-stern：

```
$ perlcritic -stern ~/bin/journals
Code before warnings are enabled at line 79, column 1.  See page 431 of PBP.↵
(Severity: 4)
Code before warnings are enabled at line 79, column 6.  See page 431 of PBP.↵
(Severity: 4)
... snip a couple hundred more lines ...
```

表 7-1：perlcritic 可以使用的严重级别的编号或名字

编号	名字
--serverity 5	-gentle
--serverity 4	-stern
--serverity 3	-harsh

续表

编号	名字
--serverity 2	-cruel
--serverity 1	-brutal

我们发现与这些警告相关的规则是 TestingAndDebugging::RequireUseWarnings。因为我们既不是在测试也不是在调试，所以我们须要关掉警告（注 4）。现在我们的*.perlcriticrc* 变得更长了：

```
# perlcriticrc
[-ValuesAndExpressions::ProhibitLeadingZeros]
[-TestingAndDebugging::RequireUseWarnings]
```

我们可以继续降低严重级别以得到更加挑剔的警告。级别越低，得到的警告越顽固。比如说，perlcritic 开始抱怨使用了 die 而不是 croak，虽然在我们的程序里 croak 并不能为我们做任何事，因为我是在代码的最外层使用的 die 而不是在子程序中。croak 可以为调用的子程序调整输出的报告，不过在这里没有调用的子程序：

```
"die" used instead of "croak" at line 114, column 8. See page 283 of PBP. (Severity: 3)
```

如果想继续使用 perlcritic，我们须要针对这个程序调整配置文件。不过对于这些严重程度较低的警告，我们可能不希望在所有的分析中都禁用它们。我们可以把*.perlcriticrc* 拷贝到 *journal-critic-profile* 中，并用--profile 开关告诉 perlcirtic 使用新的配置文件：

```
$ perlcritic --profile journal-critic-profile ~/bin/journals
```

有的时候，完全关掉一条规则并不是最佳的选择。有一条规则是不要在字符串上下文中使用 eval。通常情况下，这是一个好主意。不过，在动态加载模块时，我们就须要在字符串上下文中使用 eval。在 eval 中我们须要“require”一个变量，而这个变量可能是一个字符串或一个裸词（bareword）。

```
Eval "require $module";
```

和往常一样，Perl::Critic 会给出警告，因为它不知道这种用法是达到目的的唯一方法。为了处理这些情况，Ricardo Signes 创建了 Perl::Critic::Lax 模块。该模块添加了一些规则，会对 construct 发出警告，除非是 use，就像我们的 eval-require 一样。这是一个很好的想法。它的规则 Perl::Critic::Policy::Lax::ProhibitStringyEval::ExceptForRequire 就可以解决上面的问题。在上下文中使用 eval 固然是不好的，不过这个例子不同。我快完成本书的时候，他刚发布了这个模块。我相信这个模块会变得更加有用。当你看到这本书的时候，应该会有更多的 Perl::Critic 规则，所以请及时查看 CPAN。

注 4：一般来说，我建议在产品代码中关掉警告。只在须要测试和调试程序的时候才打开警告。完成这些事情后，你就不再需要它了。警告有可能只会塞满你的日志文件。

创建自己的 Perl::Critic 规则
Creating My Own Perl::Critic Policy

这些只是使用 Perl::Critic 的开始。我们刚才演示了如何改变它的工作方式以关掉一些规则。其实，我们也可以添加自己的规则。每个规则都是一个 Perl 模块。规则模块在 Perl::Critic::Policy::*名字空间中，它们继承自 Perl::Critic::Policy 模块(注 5)：

```
package Perl::Critic::Policy::Subroutines::ProhibitMagicReturnValues;

use strict;
use warnings;
use Perl::Critic::Utils;
use base 'Perl::Critic::Policy';

our $VERSION = 0.01;

my $desc = q{returning magic values};


sub default_severity { return $SEVERITY_HIGHEST   }
sub default_themes   { return qw(pbp danger)      }
sub applies_to       { return 'PPI::Token::Word'  }


sub violates
        {
        my( $self, $elem ) = @_;
        return unless $elem eq 'return';
        return if is_hash_key( $elem );

        my $sib = $elem->snext_sibling();

        return unless $sib;
        return unless $sib->isa('PPI::Token::Number');
        return unless $sib =~ m/^\d+\z/;

        return $self->violation( $desc, [ 'n/a' ], $elem );
        }

1;
```

能用模块 Perl::Critic 做的事情还有很多。使用 Test::Perl::Critic 模块，我们可以把分析添加到自动测试中。这样每次运行 make test 时就可以知道是否破坏了代码风格。编译选项 criticism 可以给程序添加一个类似于 warnings 的功能，这样在运行程序的时候就可能得到 Perl::Critic 的警告了（如果有的话）。

可能我们不认同某些规则，但是这丝毫不影响 Perl::Critic 的价值。它是可以配置和扩展的，我们完全可以让它适应具体的情况。更详细的信息请参阅本章结尾的参考文献。

注 5：文档 Perl::Critic:: DEVELOPER 对它有详细的介绍。

总结
Summary

我们拿到的代码可能是各种格式、各种编码或经过其他技巧处理后难以阅读的代码。不过我们有很多工具可以清理代码并分析出它所做的事情。只须要做一点点工作，我们就可以得到格式优美的代码，而不用忍受之前的程序员对我们的折磨。

深入阅读
Further Reading

`perltidy` 的网站有更多的细节和例子：*http://perltidy.sourceforge.net/*。可以通过安装 `Perl::Tidy` 模块来安装 `perltidy`。它也有 Vim 和 Emacs 及其他编辑器的插件。

文档 `perlstyle` 是 Larry Wall 关于风格问题的观点的汇总。你不一定要遵循他的风格，不过大部分的 Perl 程序员都是这么做的。Damian Conway 在《Perl Best Practices》中给出了自己关于风格问题的建议。

Josh McAdams 为《The Perl Review 2.3》（2006 年夏季）写了一篇名为“Perl Critic”的文章：*http://www.theperlreview.com*。

`Perl::Critic` 有自己的 Web 站点。你可以上传代码让它分析：*http://perlcritic.com/*。在 Tigris 上也有一个该项目的网页：*http://perlcritic.tigris.org/*。

第 8 章
符号表和 typeglob
Symbol Tables and Typeglobs

虽然通常我们不操作 typeglob 或符号表，我们还是须要理解它们才能使用在后面的章节中介绍的技术。在这一章里，我们将会为后面介绍动态子程序、临时调整代码等高级功能打下基础。

符号表用于组织和存储 Perl 的软件包（全局）变量。我们可以通过 typeglob 操作符号表。通过操作 Perl 的变量的记录，我们可以实现一些强大的功能。你可能已经得到了这些技巧的好处，虽然你并不知道。

软件包变量和词法变量
Package and Lexical Variables

在深入讨论之前，我希望回顾一下软件包变量和词法变量的区别。符号表会记录软件包变量，但是不会记录词法变量。当我们操作符号表或 typeglob 时，我们改变的是软件包变量。软件包变量也被称为全局变量，因为它们在整个程序的范围内都可见。

在《Learning Perl》和《Intermediate Perl》中，我们尽可能地使用了词法变量。我们用 my 声明词法变量，这些变量只在它们的作用域内可见。因为词法变量的作用域有限，我们不必知道程序的所有内容也可以避免名字冲突。访问词法变量也比软件包变量要快些，因为 Perl 不须要操作符号表。

词法变量的作用域有限，它们只会影响部分程序。下面的代码片段在不同的作用域中两次声明了变量$n，从而创建了两个互不影响的变量。

```
my $n = 10; # outer scope

my $square = square( 15 );

print "n is $n, square is $square\n";

sub square { my $n = shift; $n ** 2; }
```

这种重复使用$n的情况没有问题。子程序中的声明是在一个不同的作用域中，所以得到的变量屏蔽了外面的变量。子程序运行结束后，子程序中的$n会被销毁，就像从来没有存在过一样。外层的$n依然还是10。

软件包变量则完全不同。用类似的方法处理软件包变量会覆盖之前$n的定义：

```
$n = 10;

my $square = square( 15 );

print "n is $n, square is $square\n";

sub square { $n = shift; $n ** 2; }
```

不过，Perl有一种方法能处理重复命名的软件包变量。Perl内置的local会暂时把当前的值——10，保存起来直到作用域结束。这样整个程序就会看到新的值——15，直到local的作用域结束：

```
$n = 10;

my $square = square( 15 );

print "n is $n, square is $square\n";

sub square { local $n = shift; $n ** 2; }
```

我们在《Intermediate Perl》中展示过它们的区别。使用local声明的变量会改变包括作用域之外的所有东西，而词法变量只在作用域范围内有效。下面的小程序演示了这两种情况。我们定义一个软件包变量$global，然后观察用不同的方式重新定义它时会发生什么。为了观察发生的事情，我们用子程序show_me输出$global的类型。我们先在开始之前调用一次show_me，然后再在每个重新定义$global的子程序中调用show_me。记住show_me在其他任何子程序的词法作用域之外：

```
#!/usr/bin/perl

#  现在还看不出来，稍微等等
$global = "I'm the global version";

show_me('At start');
lexical();
localized();
show_me('At end');

sub show_me
        {
        my $tag = shift;

        print "$tag: $global\n"
        }
```

子程序 lexical 先定义了名为$global 的词法变量。在子程序中，很明显$global 的值是我们在子程序中给定的值。不过，调用 show_me 的时候，代码跳出了子程序之外。在子程序之外，词法变量不再有效。所以在输出中，以“From lexical()”标记的行显示的是“I'm the global version”：

```
sub lexical
        {
        my $global = "I'm in the lexical version";
        print "In lexical(), \$global is --> $global\n";
        show_me('From lexical()');
        }
```

使用 local 时则完全不同，因为它处理的是软件包变量。当我们用 local 声明一个变量时，Perl 会为接下来的整个作用域保留当前值。在整个程序中，一直到作用域结束之前，新赋给该变量的值会一直可见。调用 show_me 的时候，虽然跳出了子程序，我们在子程序中给 $global 设置的值依然可见：

```
sub localized
        {
        local $global = "I'm in the localized version";
        print "In localized(), \$global is --> $global\n";
        show_me('From localized');
        }
```

输出的结果显示出了两者的不同。起初，$global 是最初的值。在 lexical()中，我们给它赋了一个新的值，不过 show_me 看不到，它看到的仍然是全局的版本。在 localized()中，新的值出现在了 show_me 中。不过，调用完 localized()后，$global 回到了最初的值：

```
At start: I'm the global version
In lexical(), $global is --> I'm in the lexical version
From lexical: I'm the global version
In localized(), $global is --> I'm in the localized version
From localized: I'm in the localized version
At end: I'm the global version
```

好好研究一下输出的结果吧。之后在介绍 typeglob 时我们还会用到这个例子。

获取软件包变量

Getting the Package version

无论是在程序的什么部分或是哪个软件包中，我们都可以在变量名前加上软件包的名字得到软件包变量。在前面的 lexical()中，即使该变量名被同名的词法变量覆盖了，我们也可以看到软件包版本的变量$global。只须要给它加上软件包的完整名字——$main::global 就可以了：

```
sub lexical
        {
        my $global = "I'm in the lexical version";
        print "In lexical(), \$global is --> $global\n";
```

```
        print "The package version is still --> $main::global\n";
        show_me('From lexical()');
        }
```

输出结果显示两个变量都可以访问：

```
In lexical, $global is  --> I'm the lexical version
The package version is still --> I'm the global version
```

不过，我们能做的事情不止于此。如果由于某种奇怪的原因，软件包变量和当前作用域内的词法变量有着同样的名字，我们也可以使用 our（Perl 5.6 引入的）来告诉 Perl 在作用域其余部分使用软件包变量：

```
sub lexical
        {
        my $global = "I'm in the lexical version";
        our $global;
        print "In lexical with our, \$global is --> $global\n";
        show_me('In lexical()');
        }
```

现在，输出结果显示我们从来没有得到词法版本的变量：

```
In lexical with our, $global is  --> I'm the global version
```

这种使用方式看起来非常的傻，因为它在子程序的剩余部分屏蔽了词法版本的变量。如果我们只须要在子程序的部分区域使用软件包版本的变量，我们可以为它创建一个作用域，这样就可以在该部分使用软件包变量，而在其他部分使用词法变量：

```
sub lexical
        {
        my $global = "I'm in the lexical version";

                {
                our $global;
                print "In the naked block, our \$global is --> $global\n";
                }

        print "In lexical, my \$global is --> $global\n";
        print "The package version is still --> $main::global\n";
        show_me('In lexical()');
        }
```

现在，输出结果显示了所有可能使用的$gobal 的方式：

```
In the naked block, our $global is --> I'm the global version
In lexical, my $global is  --> I'm the lexical version
The package version is still --> I'm the global version
```

符号表
The Symbol Table

每个软件包都有一个类似于哈希表的符号表，它包含了该软件包的所有 typeglob。它不是一个真正的 Perl 的哈希表，只是行为上有些相像。它的名字是软件包的名字加上两个冒号。

它不是一个普通的哈希表，不过我们可以用操作符 keys 来查看它的内容。想看看软件包 main 中定义的所有符号名吗？我们可以简单地打印出这个特殊的哈希表的所有键：

```
#!/usr/bin/perl

foreach my $entry ( keys %main:: )
        {
        print "$entry\n";
        }
```

在这里我就不展示它的输出了，因为输出实在太长了。当我们查看输出的时候，一定要记住变量的名字没有 sigil。当我们看到标识符_时，须要记住它有指向变量$_、@_等的引用。其实，如果在前面加上 sigil 的话，大部分的 Perl 程序员都会认出下面这些特殊的变量名：

```
/
"
ARGV
INC
ENV
$
-
0
@
```

如果查看另外的一个软件包 Foo，我们没有看到任何东西——因为该软件包还没有定义任何变量：

```
#!/usr/bin/perl

foreach my $entry ( keys %Foo:: )
        {
        print "$entry\n";
        }
```

如果在软件包 Foo 中定义一些变量，就会看到一些输出：

```
#!/usr/bin/perl

package Foo;

@n       = 1 .. 5;
$string  = "Hello Perl!\n";
%dict    = ( 1 => 'one' );

sub add ( $_[0] + $_[1] )

foreach my $entry ( keys %Foo:: )
        {
        print "$entry\n";
        }
```

输出中显示了一列不带 sigil 的标识符。符号表中存放了这些标识符：

```
n
add
string
dict
```

这些只是名字，而不是我们定义的变量。从这个输出中看不出定义的变量的类型。如果想找出变量的类型，可以在符号引用中使用变量的名字——我们将在第 9 章中详细介绍：

```
#!/usr/bin/perl

foreach my $entry ( keys %main:: )
        {
        print "-" x 30, "Name: $entry\n";

              print "\tscalar is defined\n" if defined ${$entry};
              print "\tarray  is defined\n" if defined @{$entry};
              print "\thash   is defined\n" if defined %{$entry};
              print "\tsub    is defined\n" if defined &{$entry};
        }
```

我们也可以对这些哈希表进行其他的哈希表操作。我们可以删除有着同样名字的所有变量。在下面的程序中，我们定义了变量$n 和$m，并赋给它们初值。然后调用 show_foo 列出软件包 Foo 中的所有变量名——使用软件包 Foo 是因为它没有 main 软件包的各种特殊符号：

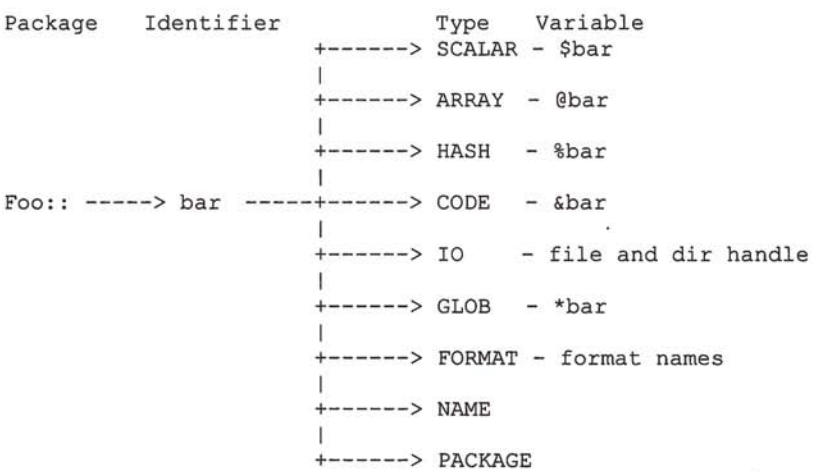
```
#!/usr/bin/perl
# show_foo.pl

package Foo;

$n = 10;
$m = 20;

show_foo( "After assignment" );

delete $Foo::{'n'};
delete $Foo::{'m'};

show_foo( "After delete" );

sub show_foo
        {
        print "-" x 10, $_[0], "-" x 10, "\n";
        print "\$n is $n\n\$m is $m\n";
        foreach my $name ( keys %Foo:: )
              {
              print "$name\n";
              }
        }
```

输出结果显示 Foo::的符号表中有变量名 n 和 m 及 show_foo 的记录。这些都是我们定义的变量名：两个标量和一个子程序。调用 delete 之后，n 和 m 的记录就没有了：

```
----------After assignment----------
$n is 10
$m is 20
show_foo
n
m
----------After delete----------
$n is 10
$m is 20
show_foo
```

typeglob

默认情况下，Perl 的变量是全局变量。这意味着只要知道它们的名字，就可以在程序的任何地方使用它们。Perl 会在符号表中维护它们的记录，该符号表在整个程序内都有效。每个软件包都有一串定义好的标识符，就像在前一节中显示的那样。每个标识符都有一个指针（虽然不是 C 语言意义上的）指向每一个变量类型的 slot。此外，还有两个额外的 slot 留给变量 NAME 和 PACKAGE，稍后我们会用到它们。下图显示了软件包、标识符和变量类型之间的关系：



总共有 7 种变量类型。最常见的 3 个是 SCALAR、ARRAY 和 HASH。不过 Perl 也为子程序提供了 CODE 类型，为文件和目录句柄提供了 IO 类型，为所有这些变量提供了 GLOB 类型。一旦我们拿到了这个结构块，就可以通过访问正确的项目得到某个该名字的特殊变量的引用。比如说，要得到 typeglob *bar 的标量，我们可以像访问哈希表一样来获得它。

不过，typeglob 并不是哈希表，我们既不能对它们使用哈希表的操作符，也不能添加更多的键：

```
$foo = *bar{SCALAR}

@baz = *bar{ARRAY}
```

我们也不能把 typeglob 作为左值：

```
*bar{SCALAR} = 5;
```

否则我们会得到一个严重错误：

```
Can't modify glob elem in scalar assignment ...
```

不过，我们可以对 typeglob 整体赋值。Perl 会找出合适的位置存放值。我们会在本章稍后的“别名”一节中详细介绍。

在 typeglob 中有两个额外的项目——PACKAGE 和 NAME，通过它们可以知道得到的 typeglob 是哪一个变量的。我并不认为这个非常有用，不过有可能某天会在 *Perl Quiz Show* 上用到：

```
#!/usr/bin/perl
# typeglob-name-package.pl

$foo = "Some value";
$bar = "Another value";

who_am_i( *foo );
who_am_i( *bar );

sub who_am_i
        {
        local $glob = shift;

        print "I'm from package " . *{$glob}{PACKAGE}  . "\n";
        print "My name is "       . *{$glob}{NAME}     . "\n";
        }
```

这个功能的作用有限，不过至少在不使用调试器的情况下，可以知道传给子函数的 typeglob 的更多信息：

```
I'm from package main
My name is foo
I'm from package main
My name is bar
```

即使有了变量名，我们也不知道这些变量的类型。一个 typeglob 表示了使用该名字的所有变量。要想知道变量的类型，我们必须使用前面使用过的 defined 技巧来判断：

```
my $name = *{$glob}{NAME};

print "Scalar $name is defined\n" if defined ${$name};
```

别名
Aliasing

通过把一个 typeglob 赋值给另一个 typeglob，我们可以创建变量的别名。在下面的这个例子中，当 Perl 把*foo typeglob 赋值给*bar typeglob 之后，所有标识符为 bar 的变量都变成了标识符为 foo 的变量的别名：

```
#!/usr/bin/perl

$foo = "Foo scalar";
@foo = 1 .. 5;
%foo = qw(One 1 Two 2 Three 3);
sub foo { 'I'm a subroutine!' }

*bar = *foo;  # typeglob assignment

print "Scalar is <$bar>, array is <@bar>\n";
print 'Sub returns <', bar(), ">\n";

$bar = 'Bar scalar';
@bar = 6 .. 10;

print "Scalar is <$foo>, array is <@foo>\n";
```

当我们改变变量 bar 或 foo 的时候，另一个变量也会发生变化，因为现在它们已经是有着不同名字的同一个东西了。

我们也可以不对整个 typeglob 进行赋值。如果把一个引用赋给一个 typeglob，只有 typeglob 中的引用变量会受到影响。把标量的引用\$scalar 赋值给 typeglob *foo 只会影响 typeglob 中的 SCALAR 部分。在下一行代码中，当我们把\@array 赋值给 typeglob 时，只有 typeglob 的 ARRAY 部分会受到影响。通过这些操作，我们把*foo 变成了科学怪人（译注 1）一样的东西，它的所有的值来自于其他变量：

```
#!/usr/bin/perl

$scalar = 'foo';
@array  = 1 .. 5;

*foo = \$scalar;
*foo = \@array;

print "Scalar foo is $foo\n";
print "Array foo is @foo\n";
```

当一个变量的名字很长而我们希望使用一个不同的名字的时候，这个功能就非常有用。这也是 Exporter 模块向我们的名字空间导入符号时所做的事情。使用该模块，我们可以在自己的名字空间中得到导入的变量，而不用指定完整的软件包名字。Exporter 会从导出的软件包中得到变量的名字，赋值给要导入的软件包的 typeglob：

```
package Exporter;

sub import {
```

译注 1：科学怪人是美国电影《Frankenstein》中的一个人造的怪物。

```
my $pkg = shift;
my $callpkg = caller($ExportLevel);

# ...
*{"$callpkg\::$_"} = \&{"$pkg\::$_"} foreach @_;
}
```

旧代码中的文件句柄参数
Filehandle Arguments in Older Code

在 Perl 5.6 引入文件句柄的引用之前，如果我们想给一个子程序传递一个文件句柄，须要使用 typeglob。这是在老的代码中最有可能看到的使用 typeglob 的地方。比如说，CGI 模块可以从指定的文件句柄中读取输入而不是 STDIN：

```
use CGI;

open FH, $cgi_data_file or die "Could not open $cgi_data_file: $!";

CGI->new( *FH ); # can't new( FH ), need a typeglob
```

这种方式对 typeglob 的引用也有效：

```
CGI->new( \*FH ); # can't new( FH ), need a typeglob
```

这是一种旧的使用文件句柄的方式。新的方式下可以使用一个含有文件句柄的引用的标量：

```
use CGI;
open my( $fh ), $cgi_data_file or die "Could not open $cgi_data_file: $!";
CGI->new( $fh );
```

在旧的方式中，文件句柄是软件包变量，它们不能是词法变量。因此，把它们传给一个子程序会产生一个问题。在子程序中它们的名字是什么呢？我们不希望使用已经在使用的名字，因为这样会覆盖它的值，我们也不能对文件句柄使用 local：

```
local( FH ) = shift; # won't work.
```

上面的这行代码会造成一个编译错误：

```
Can't modify constant item in local ...
```

我们必须使用 typeglob。Perl 允许我们对 typeglob FH 的 IO 部分进行赋值：

```
local( *FH ) = shift; # will work.
```

之后，我们就可以像通常那样使用文件句柄 FH 了。即使它是通过给 typeglob 赋值得到的也没有关系。因为我把它变成了局部变量，程序任何地方使用该名字的文件句柄时都会使用新的值，就像之前的 local 例子一样。在新的代码中，我们可以使用文件句柄的引用$fh，把 typeglob 留给旧代码使用（除非须要处理特殊文件句柄 STDOUT、STDERR 和 STDIN）。

给匿名子程序起名字
Naming Anonymous Subroutines

通过给 typeglob 赋值，我们可以给匿名的子程序一个名字。这样，就可以使用一个具名的子程序，而不用使用反引用得到子程序了。

模块 File::Find 通过使用一个回调函数来从一个目录的列表中选择文件：

```
use File::Find;

find( \&wanted, @dirs );

sub wanted { ... }
```

在模块 File::Find::Closures 中，有很多函数都能够返回可以在 File::Find 中使用的两个代码块(closure)。使用它们可以执行常见的 find 操作，而不用重复定义需要的&wanted 函数：

```
package File::Find::Closures;

sub find_by_name
        {
        my %hash  = map { $_, 1 } @_;
        my @files = ();

        (
        sub { push @files, canonpath( $File::Find::name )
                if exists $hash{$_} },
        sub { wantarray ? @files : [ @files ] }
        )
        }
```

下面，我们从模块 File::Find::Closures 中导入须要使用的生成函数——在这里是 find_by_name。然后用这个函数创建两个匿名的子程序：一个供 find 使用，一个之后用来获取结果：

```
use File::Find;
use File::Find::Closures qw( find_by_name );

my( $wanted, $get_file_list ) = find_by_name( 'index.html' );

find( $wanted, @directories );

foreach my file ( $get_file_list->() )
        {
        ...
        }
```

可能我们不想使用子程序引用——不管是出于什么原因。这种情况下，我们可以把匿名子程序赋值给一个 typeglob。因为只是对引用赋值，它只会影响 typeglob 中的子程序部分。赋值之后，我们就可以像上一节中操作文件句柄那样使用具名的子程序了。当我们把 find_by_name 的返回值赋给 typeglob *wanted 和*get_file_list 之后，就得到了以 wanted 和 get_file_list 为名字的子程序：

```
( *wanted, *get_file_list ) = find_by_name( 'index.html' );

find( \&wanted, @directories );

foreach my file ( get_file_list() )
        {
        ...
        }
```

在第9章中，我会用这个技巧操作AUTOLOAD来动态地定义子程序并替换已有子程序的定义。

总结
Summary

符号表是记录Perl的全局变量的系统，typeglob是操作符号表的途径。在某些情况下——比如给子程序传递文件句柄的时候，由于没有办法对文件句柄这个软件包变量使用引用，我们必须使用typeglob。为了绕过早期Perl的这个限制，程序员可以使用typeglob得到需要的变量。不过，这并不意味着typeglob过时了。一些有着特殊功能的模块，比如Exporter，就使用了这个功能，即使我们不知道。在实现一些特殊功能时，Typeglob非常有用。

深入阅读
Further Reading

Larry Wall、Tom Christiansen和Jon Orwant在《Programming Perl》第3版的第10章和12章中介绍了符号表和Perl内部是如何操作符号表的。

Phil Crow在Perl.com的“Symbol Table Manipulation”中演示了一些操作符号表的技巧：*http://www.perl.com/pub/a/2005/03/17/symtables.html*。

Randal Schwartz 2003年5月在Unix Review专栏中介绍了作用域：*http://www.stonehenge.com/merlyn/UnixReview/col46.html*。

第9章 动态子程序

Dynamic Subroutines

在本章中，我们把所有没有用 sub some_name 明确定义的子程序或直到运行时才存在的子程序称为“动态子程序”。Perl 非常灵活，能够在运行的时候指定须要运行的代码，甚至能够执行能生成代码的代码。在这一章里，我们将要涉及很多不同的有关子程序的话题——因为把它们分开反而不好。

我们第一次接触匿名子程序是在《Learning Perl》中。在该书中，我们演示了如何自定义排序，虽然当时我们没有说明这就是匿名子程序。在《Intermediate Perl》中，我们使用动态子程序创建了 closure，并在 map 和 grep 及其他地方使用它们。在本章中，我们会接着《Intermediate Perl》继续介绍匿名子程序的巨大威力。在这些技巧中，不必事先知道所有的东西会非常方便。

把子程序作为数据使用

Subroutines As Data

我们可以把匿名子程序保存在变量中，它们在调用之前甚至不会被运行。这种变量保存的是行为，而不是值。下面的匿名子程序把前两个参数加在一起并作为结果返回。不过，操作只有在运行它的时候才会发生。在这里，我们只是定义了子程序并把它保存在$add_sub中：

```
my $add_sub = sub { $_[0] + $_[1] };
```

这样，我们就可以通过选择合适的变量来决定做什么事情了。一个简单的程序可能会使用一系列 if-elsif 的测试和分支，因为它须要为每个可能的子程序调用硬编码的分支语句。这里我们写了一个能够处理简单的算术运算的计算器。它从命令行接受三个参数并进行计算。每个操作都有自己的分支代码：

```
#!/usr/bin/perl
# basic-arithmetic.pl

use strict;

while( 1 )
```

```
        {
        my( $operator, @operand ) = get_line();

        if(    $operator eq '+' ) { add(     @operand  ) }
        elsif( $operator eq '-' ) { subtract( @operand ) }
        elsif( $operator eq '*' ) { multiply( @operand ) }
        elsif( $operator eq '/' ) { divide(   @operand  ) }
        else
               {
               print "No such operator [$operator ]!\n";
               last;
               }
        }

print "Done, exiting...\n";
sub get_line
         {
         # 我们可以把它变得更复杂一些，但是它不是我们关心的重点
         print "\nprompt> ";

         my $line = <STDIN>;

         $line =~ s/^\s+|\s+$//g;

         ( split /\s+/, $line )[1,0,2];
         }

sub add      { print $_[0] + $_[1] }

sub subtract { print $_[0] - $_[1] }

sub multiply { print $_[0] * $_[1] }

sub divide   { print $_[1] ? $_[0] / $_[1] : 'NaN' }
```

这些分支几乎完全相同：它们接受两个操作数，进行运算，然后打印出结果。每个分支中唯一的不同是子程序的名字。如果我们希望添加更多的操作，就须要添加几乎完全一样的代码分支。不仅如此，添加的代码须要放在 `while` 循环中，这样会弱化循环的意图。如果有任何变动，都须要处理所有的分支。这实在是太麻烦了。

我们也可以完全去掉它们，从而避免编写和维护冗长的分支语句。我希望从分支中把子程序的名字抽出来，这样只用一块代码就能处理所有的运算。理想情况下，`while` 循环永远不用改变，只须要做获取数据、发送到正确的子程序这些基本的工作即可：

```
while( 1 )
         {
         my( $operator, @operand ) = get_line();

         my $some_sub = ....;
```

```
        print $some_sub->( @operands );
        }
```

现在，子程序被保存在了变量$some_sub 中，我们只须要决定如何得到正确的匿名子程序。我们可以建立一个分配表（一个保存所有匿名子程序的哈希表），然后通过键选择合适的子程序。在这个例子中，我们使用操作符作为键。同时，我们也可以捕获错误的输入，因为我们知道哪些操作符是合法的——合法的操作符一定是哈希表的键.。

这样，即使添加更多的操作符，处理循环也可以始终保持不变，我们把循环标记为 REPL（Read-Evaluate-Print 的缩写），这样就可以在子程序中使用它以控制循环：

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

use vars qw( %Operators );
%Operators = (
        '+'  => sub { $_[0] + $_[1] },
        '-'  => sub { $_[0] - $_[1] },
        '*'  => sub { $_[0] * $_[1] },
        '/'  => sub { $_[1] ? eval { $_[0] / $_[1] } : 'NaN' },
        );

while( 1 )
        {
        my( $operator, @operand ) = get_line();
        my $some_sub = $Operators{ $operator };
        unless( defined $some_sub )
                {
                print "Unknown operator [$operator]\n";
                last;
                }

        print $Operators{ $operator }->( @operand );
        }

print "Done, exiting...\n";


sub get_line
        {
        print "\nprompt> ";

        my $line = <STDIN>;

        $line =~ s/^\s+|\s+$//g;

        ( split /\s+/, $line )[1,0,2];
        }
```

如果想添加更多的操作符，只须要在哈希表中加入新的项目。我们可以添加新的操作符，比如用操作符%表示取模，用操作符 x 表示乘法操作*：

```
use vars qw( %Operators );
%Operators = (
        '+' => sub { $_[0] + $_[1] },
        '-' => sub { $_[0] - $_[1] },
        '*' => sub { $_[0] * $_[1] },
        '/' => sub { eval { $_[0] / $_[1] } || 'NaN' },
        '%' => sub { $_[0] % $_[1] },
        );
$Operators{ 'x' } = $Operators{ '*' };
```

挺好的，一切都能正常工作。可能我们希望修改程序以使用逆波兰表达式（Reverse Polish Notation）（操作数在前面，操作符在后面），而不是简单的算术表示。这也容易处理，我们只用改变选择匿名子函数的方式。我们可以检查最后的参数，而不是中间的参数。这些处理都发生在 get_line 子程序中。我们只须要对它稍作调整，其他的都和原来一样：

```
sub get_line
        {
        print "\nprompt> ";

        my $line = <STDIN>;

        $line =~ s/^\s+|\s+$//g;
        my @list = split /\s+/, $line;

        unshift( @list, pop @list );

        @list;
        }
```

换成后波兰表示法之后，我们可以进一步做些改变以处理二元操作之外的运算。如果我们要处理的操作的参数超过两个，我们处理的方式可以保持不变：把最后一个参数作为运算符，把其余的参数传给子程序。除了添加一个新的操作符之外，我们不必修改任何地方。下面，我们定义一个"操作符，该操作符会使用 List::Util 模块的 max 函数查找所有参数中最大的一个。它类似于我们在《Learning Perl》中介绍过的例子，Perl 不在乎传给子程序的参数有多少个：

```
%Operators = (
        # ... same stuff as before

        '"' => sub {
                my $max = shift;
                foreach ( @_ ) { $max = $_ if $_ > $max }
                $max
                },
        );
```

我们也可以处理一元操作符。我们的代码并不在乎操作数的个数，只有一个元素的列表和其他的列表是一样的。下面的功能才是我写这个程序的主要目的。我经常须要在各种进制的数之间转化，或者是把 Unix 时间转换成能够阅读的时间：

```
%Operators = (
        # ... same stuff as before

        'dh' => sub { sprintf "%x",     $_[0]   },
        'hd' => sub { sprintf "%d", hex $_[0]   },
        't'  => sub { scalar localtime( $_[0] ) },
        );
```

最后，如果一个操作没有参数会怎么样呢？这只是一个退化的情况。我们的程序没有一种退出程序的方法。如果运行它们，我必须强行中断程序。现在我们可以加上一个 q 运算符，它其实并不是一个真正的运算符，而是一种停止程序的方式。我们耍了个小聪明使用 last 退出 while 循环（注 1）。不过，我们也可以使用任何其他的方式，包括使用 exit 直接退出。在这里，我们使用 last 并指定了 while 循环的标记：

```
%Operators = (
        # ... same stuff as before

        'q' => sub { last REPL },
        );
```

如果需要更多的操作符，我们只须要在哈希表中添加子程序的引用，并在子程序中实现该功能。我们不必增加任何逻辑或改变程序的结构。我们只用描述一下新增的功能（虽然是用代码的方式描述的）。

创建和替换具名子程序

Creating and Replacing Named Subroutines

在上一节中，我们把匿名子程序保存在了一个变量中。其实匿名子程序只是 typeglob 的一个 slot（参见第 8 章）。我们也可以把子程序直接保存在那里。当我们把一个匿名子程序赋值给一个 typeglob 的时候，Perl 会把它保存在 CODE 区。之后，我们就可以像具名子程序那样使用它了：

```
print "Foo is defined before\n" if defined( &foo );

*foo = sub { print "Here I am!\n" };
foo();

print "Foo is defined afterward\n" if defined( &foo );
```

如果须要替换其他模块中的某些代码，这个功能就非常有用。我们会在第 10 章中用到这个功能。往往我们不希望编辑别的模块。因此我们把它保持不变，只替换掉须要改变的子程

注 1：通常情况下，使用 next、last 或 redo 退出一个子程序不是一个好的方式。这也不是很糟糕，但是这样做很奇怪，所以 *perldiag* 中有不要这样使用的警告。

序即可。由于子程序保存在符号表中，我们可以使用完整的软件包路径来替换一个子程序：

```
#!/usr/bin/perl

package Some::Module;
sub bar { print "I'm in " . __PACKAGE__ . "\n" }


package main;

Some::Module::bar();

*Some::Module::bar = sub { print "Now I'm in " . __PACKAGE__ . "\n" };

Some::Module::bar();
```

如果打开警告运行程序，Perl 会捕获并汇报可疑的行为。如果没有一个正当的理由，我们不应该替换子程序：

```
$ perl -w replace_sub.pl
I'm in Some::Module
Subroutine Some::Module::bar redefined at replace_sub.pl line 11.
Now I'm in main
```

下面我们修改代码以去掉这个警告。我们没有关掉所有的警告，而是把相关代码隔离在一个代码块中，并在该代码块中关掉所有 redefine 类型的警告：

```
{
no warnings 'redefine';
*Some::Module::bar = sub { print "Now I'm in " . __PACKAGE__ . "\n" };
}
```

上面我们修改了一个已有的子程序的定义。即使之前没有定义子程序，我们也可以这样做。只须要稍加修改，我们就可以在软件包 main 中定义软件包 Some::Module 的一个新的子程序 quux：

```
package Some::Module;
# has no subroutines

package main;

{
no warnings 'redefine';
*Some::Module::quux = sub { print "Now I'm in " . __PACKAGE__ . "\n" };
}

Some::Module::quux();
```

看到任何熟悉的东西了吗？如果我再做些修改，它看起来就有点类似于之前见过的把符号导入到另一个名字空间中的技巧。你可能已经这样做很久了，虽然你并不知道：

```
package Some::Module;

sub import
        {
        *main::quux = sub { print "I came from " . __PACKAGE__ . "\n" };
        }


package main;

Some::Module->import();

quux();
```

这正是模块 Exporter 把一个软件包中的定义导入到另一个中时所做的事情。只是 Exporter 比这个稍微复杂一点，它须要找出调用者，并对@EXPORT 和@EXPORT-OK 做一些操作。除此之外，就只是一些简单重复地对 typeglob 的赋值操作了。

符号引用
Symbolic References

在上一节中，我们用一个匿名子程序替换了一个合法子程序的定义。我们通过操作符号表来实现替换。现在，我们开始扩展这个功能的使用范围。

符号引用——也叫符号表的引用，是一种使用字符串来选择变量的名字的方式，它看起来像是在用一个反引用操作来访问变量：

```
my $name = 'foo';
my $value_in_foo = ${ $name }; # $foo
```

这往往不是一个好主意，所以 strict 模式会禁止这样做。如果在例程中加入 use strict，我们会得到一个严重错误：

```
use strict;
my $name = 'foo';
my $value_in_foo = ${ $name }; # $foo
```

strict 模式的 refs 部分发现了这个问题：

```
Can't use string ("foo") as a SCALAR ref while "strict refs" in use at program.pl line 3.
```

我们可以关掉与 refs 相关的规则暂时绕过这个问题：

```
use strict;

{
no strict 'refs';
```

```
my $name = 'foo';
my $value_in_foo = ${ $name }; # $foo
}
```

我们也可以不打开 strict 模式的 refs 部分，不过更好的方式还是只在需要的时候关掉它，这样 Perl 就能够捕获意料之外的使用：

```
use strict qw(subs vars); # no 'refs'
```

为了使用动态子程序，我们须要把子程序名字保存在一个变量中，然后把它变成一个子程序。

首先，我们把名字 foo 保存在标量$good_name 中。然后把它当成 typeglob 的引用对它进行反引用，这样就可以把一个匿名子程序赋给它。由于$good_name 不是一个引用，Perl 会把它的值当作一个符号引用处理。它的值变成了 Perl 要检查和修改的 typeglob 的名字。当我们把匿名子程序传给*{$good_name}时，Perl 就会在当前软件包的符号表中为名为&foo 的子程序创建一个项目。这种方式也适用于完整的软件包表示法，所以我们也可以创建诸如 &Some::Module::foo 之类的子程序：

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

{
no strict 'refs';

my $good_name = "foo";
*{ $good_name } = sub { print "Hi, how are you?\n" };

my $remote_name = "Some::Module::foo";
*{ $remote_name } = sub { print "Hi, are you from Maine?\n" };
}

foo();  # no problem
Some::Module::foo();  # no problem
```

不过，还有更严重的滥用方式。这种方式是我们永远不应该使用的，至少不应该在有用的或重要的代码中使用。把这种方式留到 Perl 的混淆代码大赛 (Obfuscated Perl Contest) 上吧。

我们可以把名字放在一个变量中来绕过 Perl 的变量命名规则。通常，变量名须要以字母或下划线开头，之后可以接字母、下划线或数字。通过使用符号引用，我们可以绕过这些规则，创建一个名字为<=>的子程序：

```
{
no strict 'refs';
my $evil_name = "<=>";
*{ $evil_name } = sub { print "How did you ever call me?\n" };

# <=>()  是啊，这样是不行的。

*{ $evil_name }{CODE}->();
```

```
    &{$evil_name}();    # Another way ;-)
    }
```

不过，我们没法像通常那样使用名字非法的子程序，但是我们可以通过它的 typeglob 或符号引用来调用它。

遍历子程序列表

Iterating Through Subroutine Lists

我们须要在模块 Data::Constraint 中提供一种验证数据的方法。用户可以用该方法轻松地设定复杂的限制条件而不用写任何代码。这样验证数据的工作就只是一些配置的事情了，而不是编程。

我们可以使用一系列子程序，每次验证一个数据，而不是用一个子程序验证一组数据。每个特定的数据都有自己的一组验证过程，因此我们希望分别验证每个数据（虽然可能还是使用某种循环）。每个子程序都给数据添加了某种限制条件。

我们先定义一些检查一个数值的子程序。假设事先不知道数据的含义和用户对它的限制。我们只定义一些通用的子程序，完全由程序员来按照他喜欢的方式对这些子程序进行组合。每个子程序都会返回真或假：

```
my %Constraints = (
        is_defined        => sub { defined $_[0] },
        not_empty         => sub { length $_[0] > 0 },
        is_long           => sub { length $_[0] > 8 },
        has_whitespace    => sub { $_[0] =~ m/\s/ },
        no_whitespace     => sub { $_[0] =~ m/\s/ },
        has_digit         => sub { $_[0] =~ m/\d/ },
        only_digits   => sub { $_[0] !~ m/\D/ },
        has_special   => sub { $_[0] =~ m/[^a-z0-9]/ },
        );
```

哈希表%Constraints 现在成了一个子程序库，库中有很多的验证数据的子程序。定义好子程序之后，我们再来看看如何使用它们。

比如说，我希望写一个检查密码的程序，要求密码至少有 8 个字符、没有空格、至少有 1 个数字和 1 个特殊字符。由于已经把子程序保存在了哈希表中，我们可以从哈希表中找出需要的子程序，并把备选的密码传给它们：

```
chomp( my $password = <STDIN> );
my $fails = grep {
        ! $Constraints{ $_ }->( $password )
        } qw( is_long no_whitespace has_digit has_special );
```

我们是在标量上下文中使用 grep 的，所以它会返回那些返回值为真的代码块的个数。由于我们需要的是返回值为假的个数，所以我们对返回值进行取反。如果$fails 不等于 0，就说明某个条件没有满足。

当我们要验证很多不同的数值而每个数值都有自己的要求时，这种方法的优势就体现出来了。使用的技术还是一样的，不过我们须要把它变得更加通用：

```
my $fails = grep {
        ! $Constraints{ $_ }->( $input{$key} )
        } @constraint_names;
```

这样，参数检查就变成了简单的配置：

```
password          is_long no_whitespace has_digit has_special
employee_id       not_empty only_digits
last_name         not_empty
```

我们指定需要的配置，并把它载入到程序中。这个功能对于那些须要改变程序行为的非程序员来说非常有用。他们可以不用接触任何代码。我们可以把配置保存在文件中，逐行读入，建立一个数据结构来保存每个变量的名字和相应的限制条件。完成这些工作之后，我们就可以像前面的例子那样正确地访问所有的东西了：

```
while( <CONFIG> )
        {
        chomp;
        my( $key, @constraints ) = split;
        $Config{$key} = \@constraints;
        }

my %input = get_input(); # 做一些处理输入的工作

foreach my $key ( keys %input )
        {
        my $failed = grep {
                ! $Constraints{ $_ }->( $input{$key} )
                } @{ $Config{$key} };

        push @failed, $key if $failed;
        }

print "These values failed: @failed\n";
```

不管有多少输入参数，不管每个参数的需求有多么特别，我们的代码都可以保持短小不变。

这是 Data::Constraint 模块的基本思路。不过，该模块做了更多的工作，它会设置好环境，返回没有满足的限制条件。我们可以稍微修改一下这个程序以返回没有满足的条件：

```
my @failed = grep {
        $Constraints{ $_ }->( $value ) ? () : $_
        } @constraint_names;
```

处理流水线
Processing Pipelines

就像上面的例子中遍历限制条件的列表那样，我们可能也想建立一个处理流水线。我们可以用同样的方式：确定须要使用的子程序，然后遍历子程序列表，对数值分别使用每个子程序进行处理。

在对数值进行归一化处理的时候我们须要确定一个合适的变换。我们把所有的变换操作作为子程序保存在%Transformations 中，然后在@process 中列出要使用的子程序。之后，我们从标准输入逐行读入数据，分别对每行使用每个子程序进行处理：

```
#!/usr/bin/perl
# sub-pipeline.pl
my %Transformations = (
        lowercase                => sub { $_[0] = lc $_[0] },
        uppercase                => sub { $_[0] = uc $_[0] },
        trim                     => sub { $_[0] =~ s/^\s+|\s+$//g },
        collapse_whitespace      => sub { $_[0] =~ s/\s+/ /g },
        remove_specials          => sub { $_[0] =~ s/[^a-z0-9\s]//ig },
        );

my @process = qw( remove_specials lowercase collapse_whitespace trim );

while( <STDIN> )
        {
        foreach my $step ( @process )
                {
                $Transformations{ $step }->( $_ );
                print "Processed value is now [$_]\n";
                }
        }
```

我们甚至可以把这些和上一节的限制条件检查合并在一起。先对数值进行清理，再检查它们的合法性。数据输入和处理部分的代码都非常简短，它们会保持不变。复杂的部分完全被隔离在数据流动过程之外了。

方法列表
Method Lists

其实这一节和上面两节并不一样，不过每当我谈到前面这些技术的时候我都会想到它。在《Intermediate Perl》中，我们曾经提到，只要一个变量是标量（不能是引用或其他东西），就可以把它当作一个方法的名字使用——只要对象能够处理 foo 方法就没有问题：

```
my $method_name = 'foo';
$object->$method_name;
```

如果想执行某个对象的一系列方法，我们可以像处理匿名子程序那样遍历方法的列表。其

实对于 Perl 来说它们是不一样的。不过，对于程序员来说，思考的方式是相同的。我们可以遍历所有的方法名，使用 map 得到需要的值：

```
my $isbn = Business::ISBN->new( '0596101058' );

my( $country, $publisher, $item ) =
        map { $isbn->$_ }
        qw( country_code publisher_code article_code );
```

我们不须要重复同样的代码。和上面一样，提取所需数值的代码会变得非常简短，而且选择并找出所需方法的复杂代码也和代码流的重要部分分离了开来。

把子程序作为参数使用
Subroutines As Arguments

由于子程序引用是标量，我们可以把它们作为参数传给其他的子程序：

```
my $nameless_sub = sub { ... };
foo( $nameless_sub );
```

不过，在这里我们不想把它们当成标量传递。我们希望像 sort、map 和 grep 那样使用内联代码做一些比较炫的事情：

```
my @odd_numbers  = grep { $_ % 2 } 0 .. 100;

my @squares      = map  { $_ * $_ } 0 .. 100;

my @sorted       = sort { $a <=> $b } qw( 1 5 2 0 4 7 );
```

为了实现这个功能，我们须要使用 Perl 的子程序原型。可能有人曾经告诉你子程序原型没有什么用，而且是不好的，不过在这里我们须要使用这个功能来告诉 Perl 该代码块是一个子程序。

作为例子，我想写个程序根据传给它的某个代码块把一个列表化简成一个值。在模块 List::Util 中，Graham Barr 用 reduce 函数实现了这个功能。函数 reduce 可以用传给它的子程序把一个列表变成一个值。下面的代码片段可以把一串数加起来：

```
use List::Util;
my $sum = reduce { $a + $b } @list;
```

函数 reduce 是一个非常有名的处理列表的函数，你会在很多其他的语言中见到它。它先从列表中取出两个参数，用内联函数计算结果。之后，它对结果和列表中的下一个元素进行同样的操作，直到遍历完列表中的所有元素。

和 map、grep 和 sort 类似，在 reduce 函数的内联子程序后面没有逗号。不过，为了让整个程序能够工作，我须要使用 Perl 子程序原型告诉程序要使用内联函数。

为了提高运行速度，List::Util 模块中的函数是用 XS 实现的。不过 Graham 也提供了一个纯 Perl 的版本，可以供我们无法使用 XS 时使用：


```
package List::Util;

sub reduce (&@) {
  my $code = shift;
  no strict 'refs';

  return shift unless @_ > 1;

  use vars qw($a $b);

  my $caller = caller;
  local(*{$caller."::a"}) = \my $a;
  local(*{$caller."::b"}) = \my $b;

  $a = shift;
  foreach (@_) {
       $b = $_;
       $a = &{$code}();
  }

  $a;
}
```

在他的原型中，Graham 使用了（&@）。符号&告诉 Perl 第一个参数是一个子程序，符号@表示剩下的部分是一个列表。文档 *perlsub* 中有一个所有原型的符号和它们的含义的列表。不过在这里我们只须要使用&@。

reduce 的剩余部分和 sort 的工作方式类似，每次把两个元素放在软件包变量$a 和$b 中。Graham 用这两个名字定义了词法变量，随后立刻用符号引用把它们赋给了$a 和$b 的 typeglobs。之后使用的$a 和$b 则是词法变量。当它调用子程序参数&{$code}()时，代码使用的是软件包变量，也就是我们写程序时使用的版本。明白了吗？在 reduce 中，我们使用的是词法变量；在$code 中，我们使用的是调用软件包的软件包变量。这正是 Graham 把一个变成另一个别名的原因。

我们也可以去掉全局变量$a 和$b。使用@_可以达到此目的：

```
my $count = reduce { $_[0] + $_[1] } @list;
```

由于@_永远是软件包 main::中 Perl 的一个特殊变量，我们不用担心调用的软件包是什么，也不用担心如何把列表的元素放到变量中。我们可以直接操作@_，调用匿名子程序处理@_中的前两个元素，并把结果放回到@_中。这个过程一直进行下去直到@_中剩下一个元素，最后把该元素返回给调用程序：

```
sub reduce(&@)
        {
        my $sub = shift;
```

```
        while( @_ > 1 )
                {
                unshift @_, $sub->( shift, shift );
                }

        return $_[0];
        }
```

到目前为止，这种方法只适用于扁平结构的列表。如果想处理一个复杂的数据结构该怎么办呢？在 Object::Iterate 模块中，我创建了能处理任何数据结构的 map 和 grep。我们可以这样调用我的 imap 和 igrep（注 2）：

```
use Object:Iterate;

my @filtered     = igrep {...} $object;

my @transformed  = imap  {...} $object;
```

我使用了和前面一样的技巧——原型。不过这次第二个元素是一个标量，因为我们处理的是一个对象而不是一个列表。我们使用了原型（&$）：

```
sub igrep (&$)
        {
        my $sub    = shift;
        my $object = shift;

        $object->_check_object;

        my @output = ();

        while( $object->__more__ )
                {
                local $_ = $object->__next__;

                push @output, $_ if $sub->();
                }

        $object->__final__ if $object->can( __final__ );

        wantarray ? @output : scalar @output;
        }

sub _check_object
        {
        croak( "iterate object has no __next__ method" )
                unless eval { $_[0]->can( '__next__' ) };
```

注 2：我想 Mark Jason Dominus 在我之前就使用了这些名字。不过当我想出这些名字的时候并没有读过他在 Perl 的邮件列表上题目为的 Higher-Order 的文章。在 Perl Review 0.5 上我的文章《Iterator Design Pattern》的一个脚注中，我认为这只是一个巧合。我们当时都在考虑迭代，只是我认为设计模式很棒，而他认为设计模式很傻。可能我们都是对的。

```
    croak( "iterate object has no __more__ method" )
           unless eval { $_[0]->can( '__more__' ) };

    $_[0]->__init__ if eval { $_[0]->isa( '__init__' ) };

    return 1;
    }
```


在 igrep 中，我们把内联子程序放在了$sub 中，把对象放在了$object 中。Object::Iterate 依赖于对象来在循环中找到下一个的元素。我们调用_check_object 检查对象是否有某个方法，从而确保对象可以处理某个方法。

方法__more__会告诉 igrep 是否还有更多的元素须要处理。如果有的话，igrep 会调用__next__从该对象得到下一个元素。无论我们在对象中保存了什么样的数据，igrep 都可以处理，因为它可以通过对象找出数据的类型。

得到一个元素之后，我们把它赋给了$_，就像普通版本的 map 和 grep 一样。在内联函数中，我们用$_表示当前的元素。

下面这个短小的例子使用了 Netscape::Bookmarks 模块。我们希望遍历它的所有类别和链接构成的树以检查所有的链接。得到$bookmark 对象之后，我们调用 igrep 处理它。在内联函数中，我们使用 HTTP::SimpleLinkChecker 模块的 check_link 函数得到链接的 HTTP 状态。200 表示链接没有问题。不过我们希望找到的是失效的链接，所以我用 igrep 查找那些状态不是 200 的链接。最后，我们把失效链接的总数和地址打印出来：

```
#!/usr/bin/perl
# bookmark-checker.pl

use HTTP::SimpleLinkChecker qw(check_link);
use Netscape::Bookmarks;
use Object::Iterate qw(igrep);

my $bookmarks = Netscape::Bookmarks->new( $ARGV[0] );
die "Did not get Bookmarks object!" unless ref $bookmarks;

my @bad_links = igrep {
        200 != check_link($_);
        } $bookmarks;

{
local $/ = "\n\t";
print "There are " . @bad_links . " bad links$/@bad_links\n";
}
```

比较富有技巧性的代码在程序的后半部分。我们定义了供 Object::Iterate 使用的特殊的方法。然后创建了一个作用域，在该作用域定义了 Netscape::Bookmarks::Category 模块的一些方法，并为词法变量@links 提供了一个作用域。我们的__more__方法只是简单地返回@links 中的元素的个数，__next__只是返回@links 中的第一个元素。我们还可以

用__next__遍历整个数据结构，而不是使用__init__一次得到所有的元素。不过这会占用太多的篇幅。不管我们怎么做，我们只须要使用 Object::Iterate 的接口：

```
{
package Netscape::Bookmarks::Category;
my @links = ();

sub __more__ { scalar @links }
sub __next__ { shift  @links }

sub __init__
        {
        my $self = shift;

        my @categories = ( $self );

        while( my $category = shift @categories )
               {
               push @categories, $category->categories;
               push @links, map { $_->href } $category->links;
               }

        print "There are " . @links . " links\n";
        }
}
```

自动加载的方法
Autoloaded Methods

当 Perl 在一个模块里或所有的派生类中找不到一个方法时，它会回到最初的类中查找一个特殊的子程序——AUTOLOAD。为了捕获所有的这种问题，Perl 把软件包变量$AUTOLOAD 的值设定为正在查找的方法的名字，并传给 AUTOLOAD 同样的参数列表。之后，完全由我们来决定如何处理。

为了定义一个基于 AUTOLOAD 的方法，我们先找出正确的方法名。Perl 会把完整的软件包名也放在$AUTOLOAD 中，通常我们需要的只是名字的最后一部分，所以可以用一个正则表达式把它提取出来：

```
if( $AUTOLOAD =~ m/::(\w+)$/ )
        {
        # 结果保存在$1 中
        }
```

有些代码也会使用替换操作，抛弃掉所有其他部分，只留下方法名。不过这种方法的问题在于会破坏$AUTOLOAD 的原始的值——可能之后会用到原始的值：

```
$AUTOLOAD =~ s/.*::///;  # 破坏性的，不推荐使用
```

一旦得到了方法名，我们可以而做任何想做的事情。我们可以给 typeglob 赋值来定义一个

具名的子程序（就像第 8 章中我承诺的那样）。变量$AUTOLOAD 依然存有完整的数据包名，我们可以把它当成一个符号引用来使用。由于$AUTOLOAD 不是一个引用，Perl 会把 typeglob 的反引用解析成定义该名字的一个变量、访问 typeglob，最后进行赋值操作：

```
*{$AUTOLOAD} = sub { ... };
```

如果$AUTOLOAD 是 Foo::bar，它会变成：

```
*{'Foo::bar'} = sub { ... };
```

这行代码设置了正确的软件包，定义了子程序名，但是没有实现子程序，最后把一个匿名子程序赋给了它。如果有足够多的时间，我会把代码写成这样：

```
{
package Foo;

sub bar;

*bar = sub { ... }
}
```

定义好子程序后，我们希望用之前试图使用的参数运行该子程序。不过，我们希望让它看起来就像和 AUTOLOAD 没有关系一样，而且不希望 AUTOLOAD 出现在调用栈中。这正是少数几个须要使用 goto 的地方之一。使用 goto 可以从调用栈中去掉 AUTOLOAD，并执行刚定义好的子程序。在子程序名字前使用&后，Perl 会把当前的@_当作子程序的参数（注 3）：

```
goto &{$AUTOLOAD};
```

在《Intermediate Perl》的第 14 章里，我们使用了 AUTOLOAD 动态地定义子程序。下面我们看看$AUTOLOAD 的内部实现。如果方法名和@elements 中的某个东西相同，我们创建一个匿名子程序返回哈希表中与该键对应的值。我们把匿名子程序赋给该名字的 typeglob。这是一个符号引用，所以我们在它周围加了一个括号以限定 no strict 'refs'的作用域。最后，在 typeglob 赋值之后，我们使用 goto 把方法调用转到刚定义好的子程序上。从效果上来看，该子程序好像一直就是定义好了的，第二次调用该方法的时候，Perl 就不须要再次查找它了：

```
sub AUTOLOAD {
        my @elements = qw(color age weight height);

        our $AUTOLOAD;

        if ($AUTOLOAD =~ /::(\w+)$/ and grep $1 eq $_, @elements) {
              my $field = ucfirst $1;
```

注 3：Nathan Torkington 在《The Perl Journal》第 9 期的“CriptoContext”中对它进行了详细的介绍。

```
        {
        no strict 'refs';
        *{$AUTOLOAD} = sub { $_[0]->{$field} };
        }
        goto &{$AUTOLOAD};
        }

    if ($AUTOLOAD =~ /::set_(\w+)$/ and grep $1 eq $_, @elements) {
        my $field = ucfirst $1;
        {
        no strict 'refs';
        *{$AUTOLOAD} = sub { $_[0]->{$field} = $_[1] };
        }
        goto &{$AUTOLOAD};
        }

    die "$_[0] does not understand $method\n";
    }
```

作为对象使用的哈希表

Hashes As Objects

我最喜欢的一个使用 AUTOLOAD 的地方是 Paul Hoffman 的 Hash::AsObject 模块。他在他的 AUTOLOAD 子程序中做了一些奇妙的事情，从而可以像通常那样通过键访问哈希表中的值，也可以通过方法名来访问相应的对象：

```
use Hash::AsObject;

my $hash = Hash::AsObject->new;

$hash->{foo} = 42;   # 以普通的方式访问一个哈希表的引用

print $hash->foo, "\n"; # as an object;

$hash->bar( 137 ),      # set a value;
```

它甚至能处理多层哈希表：

```
$hash->{baz}{quux} = 149;

$hash->baz->quux;
```

它的技巧是$hash 只是软件表中的一个哈希表的引用。当我们调用该引用的一个方法时，由于该方法不存在，Perl 会调用 Hash::AsObject::AUTOLOAD。这段代码因为要处理很多种特殊情况，所以比较长。在这里我们就不展示代码了。不过基本的思路和前一节是一样的，都是动态地定义子程序。

自动切分

AutoSplit

自动切分是另一种类似于AUTOLOAD的技术，不过它的使用没有过去那么频繁了。AutoSplit

模块得到一个模块后，会解析子程序的定义，把每个子程序保存在单独的文件中。它只在调用子程序的时候才加载相应的文件。在一个复杂的有上百个子程序的 API 中，如果只须要使用其中几个的话，就不须要让 Perl 编译所有的子程序。一旦加载某个子程序后，在同一个程序中 Perl 就不用再次编译它了。简单地说，我们把编译时间推迟到了使用之前。

使用 AutoSplit 时，须要把子程序的定义放在__END__符号后面，这样 Perl 就不会解析和编译它们了。下面我们告诉 AutoSplit 使用这些定义，并把它们分别存在文件中：

```
$ perl -e 'use AutoSplit; autosplit( "MyModule.pm", "auto_dir", 0, 1, 1 );
```

通常我们并不须要自己切分文件，因为 ExtUtils::MakeMaker 模块会在编译时帮我们处理。切分模块之后，我们会发现结果出现在 Perl 库路径的某个 auto 目录下。每个.al 文件中都保存有一个子程序的定义：

```
ls ./site_perl/5.8.4/auto/Text/CSV
_bite.al      combine.al       fields.al    parse.al          string.al
autosplit.ix error_input.al   new.al       status.al         version.al
```

为了在需要的时候加载方法的定义，我们可以使用 AutoLoader 模块的 AUTOLOAD 方法，并且把它赋值给一个 typeglob。该方法知道如何找到正确的文件，加载、编译该文件，最后定义好子程序：

```
use AutoLoader;
*AUTOLOAD = \&AutoLoader::AUTOLOAD;
```

你可能已经在工作中遇到了 AutoSplit。下面的这个错误信息表示 AutoLoader 在文件中没有找到某个方法。由于它没有找到文件，它会汇报无法找到文件。模块 Text::CSV 使用了 AutoLoader，所以当我们加载模块并使用了一个未定义的方法时，我们会得到这个错误：

```
$ perl -MText::CSV -e '$q = Text::CSV->new; $q->foobar'
Can't locate auto/Text/CSV/foobar.al in @INC ( ... ).
```

几乎所有的情况下，这种类型的错误意味着我们使用了一个在模块接口范围之外的方法名。

总结
Summary

通过使用子程序引用，我们可以用数据来表示行为。我们可以像任何其他标量一样使用子程序引用。

深入阅读
Further Reading

关于原型的说明文档在 *perlsub* 中。

Mark Jason Dominus 也使用函数名 `imap` 和 `igrep` 做了和我一样的事情，不过他在 Higher-Order Perl 上关于遍历器的讨论更加深入。参见 *http://hop.perl.plover.com/*。我在《The Perl Review 0.5》（2002 年 9 月）的“The Iterator Design Pattern”中讨论了遍历器，你可以从网上免费得到：*http:// www.theperlreview.com/Issues/The_Perl_Review_0_5.pdf*. Mark Jason 的书涉及了 Perl 的 functional programming，通过已有的函数构造新的函数，该书主要关注的是 fancy 的子程序技巧。

Randy Ray 在《The Perl Journal》第 6 期中介绍了自动切分模块。很长的一段时间，这篇文章是我最喜欢的和读得最多的关于 Perl 的文章。

Nathan Torkington 的“CryptoContext”发表在《The Perl Journal》的第 9 期上和合辑《The Best of The Perl Journal: Computer Science & Perl Programming》中。

第10章 修改模块和临时调整模块

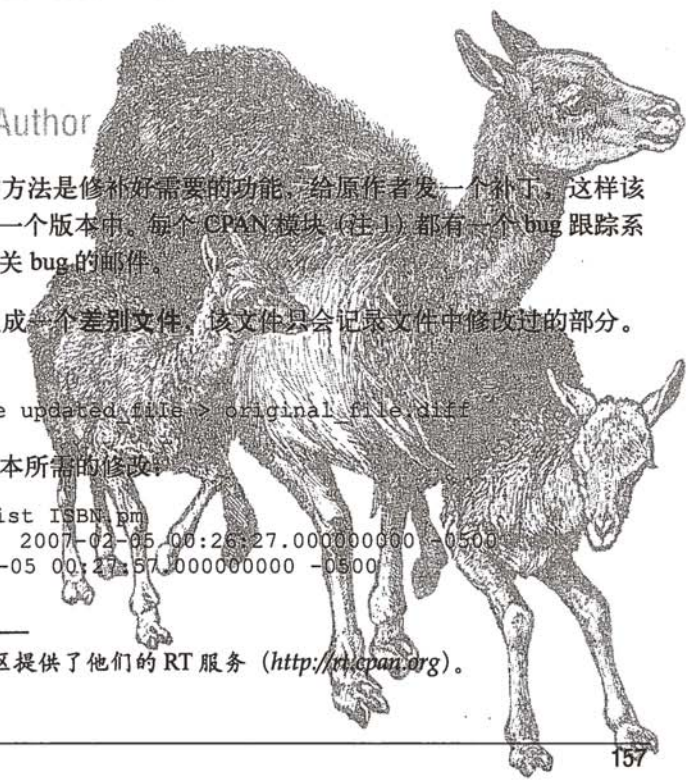
Modifying and Jury-Rigging Modules

虽然 CPAN 上有超过 10 000 个软件包，有的时候我还是找不到正好满足需要的。有的时候某个模块有个 bug，或者缺乏某个功能。我们有多种方法来解决这个问题——不管模块的作者是否同意我的修改。关键是要在不修改源代码的条件下解决问题。

选择正确的解决办法

Choosing the Right Solution

解决模块存在的问题可能有很多种方法，没有一个方法能适应所有的情况。我往往喜欢选择那些所需工作量较小、但是对 Perl 社区帮助最大的方法，虽然这样做不一定总能保持兼容性。在本章中，我不会直接给出问题的答案。我所能做的事情是指出牵涉到的一些问题，这样读者就可以决定哪个方案最适合于自己的情况。



把补丁寄给作者

Sending Patches to the Author

在大部分情况下，工作量最小的方法是修补好需要的功能，给原作者发一个补丁，这样该模块的作者就可以把它包括在下一个版本中。每个 CPAN 模块（注 1）都有一个 bug 跟踪系统，模块的作者会自动地收到相关 bug 的邮件。

当我们完成修补后，我们可以生成一个**差别文件**，该文件只会记录文件中修改过的部分。我们用 `diff` 命令生成**补丁文件**：

```
$ diff -u original_file updated_file > original_file.diff
```

补丁文件显示了从原版本到新版本所需的修改：

```
% diff -u -d ISBN.pm.dist ISBN.pm
--- ISBN.pm.dist        2007-02-05 00:26:27.000000000 -0500
+++ ISBN.pm     2007-02-05 00:27:57.000000000 -0500
@@ -59,8 +59,8 @@
```

注 1：Best Practice 免费为 Perl 社区提供了他们的 RT 服务（*http://rt.cpan.org*）。

```
                $self->{'isbn'}      = $common_data;
                if($isbn13)
                {
-               $self->{'positions'} = [12];
-               ${$self->{'positions'}}[3] = 3;
+               $self->{'positions'}     = [12];
+               $self->{'positions'}[3] = 3;
                }
                else
                { $self->{'positions'} = [9]; }
```

作者收到补丁文件之后可以用 patch（注 2）程序把它应用到原来的代码上。patch 程序通过读取差别文件判断出须要修改的部分并进行相应的更新：

```
$ patch < original_file.diff
```

有的时候作者正好有空，有工夫改进该模块并发布一个新的版本。在这种情况下，我们的任务就完成了。不过，CPAN 上的大部分工作都是志愿的，作者可能没有足够的空闲时间来做一些不能帮助他付房租或填饱肚子的事情。即使最认真的模块维护者也可能有的时间会很忙。

公平地说，对模块的维护者来说，即使那些看起来很简单的修补也不一定是很简单的事情。提交的补丁很少有相应的测试程序和文档，补丁有可能会对其他的模块造成影响，或者破坏可移植性。进一步来说，补丁的提交者往往倾向于改变接口以方便他们完成工作，这有可能让接口变得和其余的部分不一致。在补丁提交者看来只要五分钟的工作可能意味着模块维护者很多小时的工作。因此很多补丁最后被放在了“将要添加的功能”的列表上，而不是“已完成”的列表上。

本地修补
Local Patches

如果不能得到模块的维护者足够的重视，我们也可以自行修改代码。这样做可能会在一段时间内有效，不过当我们从 CPAN 上更新模块时，模块的新版本可能会覆盖我们的修改。我们可以把本地模块的版本改得很高，这样真正的版本不会高于我们选择的版本号，这种方法能够部分解决这个问题：

```
our $VERSION = 99999;
```

不过这样做的缺点是：当我们想安装一个修正了某个问题的正式版本时会遇到很大的麻烦。很有可能正式版的版本号会比当前版本的小，于是很多工具比如 CPAN.pm 和 CPANPLUS 会认为我们当前的版本是最新的，从而就不会安装看起来是旧的、其实是更新一些的版本。

注 2：Perl 的创造者 Larry Wall 也是 patch 最初的作者。现在 patch 由自由软件基金会维护。大部分的 Unix 类系统都有 patch 程序，Windows 用户也可以从多个来源获得它，包括 Win32 版的 GNU 工具（*http://unxutils.sourceforge.net/*）和 Perl Power Tools（*http://ppt.perl.org*）。

别人在使用我的模块的时候可能也会遇到同样的问题，不过当程序在他们更新一些看起来无关的模块之后出现问题时，他们可能不会意识到出了什么问题。为了解决这个问题，有些软件厂商会创建一个单独的模块目录仅供他们的程序使用，并把所有经过验证的模块和修补后的模块放在该目录下。这比我打算做的工作要多了很多，不过这种方法是有效的。

接管一个模块
Taking over a Module

如果一个模块对你（或者你的业务）很重要，而作者又联系不上，你可能要考虑接管这个模块。虽然 CPAN 上的所有模块都有所有者，Perl Authors Upload Server（PAUSE）（注 3）的管理员可以把你变成共同所有者，甚至把模块的所有权转交给你。

整个流程很简单，虽然不是自动的。首先，给 *modules@perl.org* 发一封邮件询问模块的状况。很多时候当你找不到模块的作者时，管理员可以找到。因为作者能够认出管理员的名字。然后，管理员会让你公开接管该模块的意向，这意味着整个社区的大部分人都会知道这个消息。随后，就是等待。这种事情并不会太快，因为管理员须要给作者足够的时间回信。他们不希望当一个作者还在度假的时候就把所有权转交给别人。

一旦你接管过该模块之后，你就得对该模块负责。可能你会发现过去自己是这山望着那山高，有可能更多地强调是自己在维护自由软件，从而又开始一个不能及时更新——转交所有权的循环。

分支
Forking

最后的一个选择是分支，或者说是创建一个和正式版本并行的版本。对于任何流行的开源项目来说，这都是一件危险的事情，不过对于 Perl 模块来说，这种事情只在很少的情况下发生过。PUASE 允许我们用另一个作者的名字上传一个模块。该模块会出现在 CPAN 上，但是 PAUSE 不会索引它。因为它不在索引中，所以那些能和 CPAN 一起工作的工具不会看到它，虽然 CPAN 上存储了它。

我们也可以使用不一样的模块名。如果我们选择了一个不同的名字，就可以把修改过的模块上传到 CPAN 上，从而 PAUSE 会用新的名字索引它，CPAN 的工具也可以自动安装它。没有人会知道我的模块，因为所有人使用的都是他们早已知道名字的和一直在使用该接口的原来的模块。所以如果新模块的接口和原模块的兼容，或者能够提供某种兼容层，可能会有助于人们接受我们的模块。

注 3：参见 *http://pause.perl.org*。当我在写作本书的时候，我是 PAUSE 的众多管理员之一，所以可能你会看到我在 *modules@perl.org* 上。不要不好意思在这个邮件列表上问问题。

自己从头开始
Start Over on My Own

可能我们会决定完全不用第三方的模块。如果一个模块是我自己写的，我永远可以找到该模块的维护者。当然，既然我同时是作者和维护者，可能我会成为所有其他人抱怨的对象。自己做意味着很多事情得自己处理。这和我的用最少的时间来完成工作的目标不太一致。只有在极少的情况下这些替代的模块才能赶得上原模块，我们应该在做大量工作之前好好考虑这些。

替换模块的部分内容
Replacing Module Parts

我曾经须要调试一个用 Email::Stuff 模块通过 Gmail 信箱发信的程序。就像其他的邮件服务器一样，这个程序应该连接上邮件服务器并发送邮件，不过它却停在了本地处理部分。它有一个很长的调用链，从 Email::Stuff 模块开始，然后经过 Email::Simple、Email::Send::SMTP、Net::SMTP::SSL、Net::SMTP，最后到达 IO::Socket::INET。这里的某个地方出了问题。顺便说一下，正是这个问题促使我写了第 4 章介绍的 Carp 模块，这样我就可以完整地看到每层的参数列表了。

我最后跟踪到了 Net::SMTP 中的某个地方。由于某个原因，本来应该自动选择好的本地端口和地址并没有被设置好。下面是从 Net::SMTP 模块中提取出的真实的 new 方法：

```
package Net::SMTP;

sub new
{
 my $self = shift;
 my $type = ref($self) || $self;

 ...
 my $h;
 foreach $h (@{ref($hosts) ? $hosts : [ $hosts ]})
  {
   $obj = $type->SUPER::new(PeerAddr => ($host = $h),
       PeerPort => $arg{Port} || 'smtp(25)',
       LocalAddr => $arg{LocalAddr},
       LocalPort => $arg{LocalPort},
       Proto    =>   'tcp',
       Timeout  => defined $arg{Timeout}
                       ? $arg{Timeout}
                       : 120
       ) and last;
  }

...
 $obj;
}
```

代码使用了一个典型的 new 调用，远程主机名是第一个参数，随后是一系列参数对。因为 Google 的服务用的不是标准 SMTP 端口，我们须要自己指定端口：

```
my $mailer = Net::SMTP->new(
'smtp.gmail.com',
        Port => 465,
        ...
        );
```

问题出在我们没有指定 LocalAddr 和 LocalPort 参数。我们不应该漏掉这个，底层的代码应该能为默认的本地地址找到一个空闲的端口。由于某种原因，当这两个变量没有得到赋值时产生了问题。当它们的值是 undef 的时候是不能工作的。我们的代码在把它们当成数字使用时应该转化成 0，而且应该告诉底层的代码自行选择一些合适的值：

```
LocalAddr => $arg{LocalAddr},
LocalPort => $arg{LocalPort},
```

为了调查这个问题，我希望改变 Net::SMTP 模块，但是我不希望直接编辑 *Net/SMTP.pm* 文件。我对于修改标配的模块往往很保守。我选择替换该模块的某些部分，而不是修改它。我希望能处理 LocalAddr 和 LocalPort 没有赋值的情况，同时也希望保留给它们赋值的功能。我把完整的解决方案中相关的部分摘录在了这里：

```
BEGIN {
use Net::SMTP;

no warnings 'redefine';

*Net::SMTP::new = sub
{
print "In my Net::SMTP::new...\n";

package Net::SMTP;

# ... snip

my $hosts = defined $host ? $host : $NetConfig{smtp_hosts};
 my $obj;

 my $h;
 foreach $h (@{ref($hosts) ? $hosts : [ $hosts ]})
  {
   $obj = $type->SUPER::new(PeerAddr => ($host = $h),
       PeerPort => $arg{Port} || 'smtp(25)',
       $arg{LocalAddr} ? ( LocalAddr => $arg{LocalAddr} ) : (),
       $arg{LocalPort} ? ( LocalPort => $arg{LocalPort} ) : (),
       Proto    => 'tcp',
       Timeout  => defined $arg{Timeout}
                       ? $arg{Timeout}
                       : 120
       );
```

```
    last if $obj;
    }

# ... snip

 $obj;
}
```

为了让所有的一切能够正常工作，我们得做很多事情。首先，我们把整个东西封装在 BEGIN 块中，这样代码就会在任何 Net::SMTP 的代码运行前得到执行。在 BEGIN 模块内，我们立刻加载 Net::SMTP，这样所有定义的内容就能生效了。我不希望 Perl 加载原来的代码而掩盖掉我所有的辛勤工作（注 4）。加载 Net::SMTP 模块之后，我告诉 Perl 不要对我接下来做的事情发出警告。这只是一个告诉我不应该做这个的线索，但是不足以阻止我这样做。

一旦完成了这些之后，我须要给 typeglob 赋值以重新定义 Net::SMTP::new()。最大的变化是在 foreach 循环中。如果参数列表中的 LocalAddr 和 LocalPort 值不为真，我们就不把它们放在 SUPER 类的参数列表中。

```
$arg{LocalAddr} ? ( LocalAddr => $arg{LocalAddr} ) : (),
$arg{LocalPort} ? ( LocalPort => $arg{LocalPort} ) : (),
```

这是一个灵巧的技巧。如果$arg{LocalAddr}值为真，它就会选择三元操作符中的第一个选择，这样我们就把 LocalAddr => $arg{LocalAddr}放在了参数列表中。如果$arg{LocalAddr}值不为真，我们就选择三元操作符的第二个值，也就是一个空列表。在这种情况下，底层的代码会自行选择合适的值。

现在我们解决了 Net::SMTP 模块的问题，而且没有改变原来的文件。即使我不想在产品代码中使用这个技巧，这个方法在帮助找出问题上也非常有效。使用这个方法，我可以修改有问题的模块，也可以立刻抛弃我们的修改得到原来的文件。我们还可以把它作为一个例子发送给模块的原作者以汇报问题。

派生子类
Subclassing

如果可能的话，最好的解决方法是从须要修改的模块中派生一个子类。我们把修改的内容存放在自己的文件中，而不接触原来模块的源文件。我们在《Intermedia Perl》的 barnyard 例子中介绍过这些内容，所以这里不再重复（注 5）。

注 4：我们假设这个程序没有使用诸如重设%INC 的值、重载模块之类的技巧。

注 5：如果你的手头没有羊驼书也没有关系，因为 Randal 把它作为 *perlboot* 的文档添加到了标准 Perl 发行版中。

在做大量的工作之前，我们先创建一个空的类。如果我们没有改变什么程序就不能正常工作了，我就不用做太多无用功了。在这个例子中，我们从 Foo 模块派生出一个子类，这样就可以添加一个新的功能了。我们可以使用 Local 名字空间，它永远也不会和一个真正的模块名字冲突。我们使用 base 编译指令从须要修改的模块 Foo 派生出 Local::Foo 模块：

```
package Local::Foo

use base qw(Foo);

1;
```

如果我们能够从这个模块派生一个类，我们也能够简单地改变使用的类名，所有的一切都应该能照常工作。在程序中，我们使用了和原类一样的名字。因为我们还没有覆盖任何东西，程序的行为应该和原来的一样。这种方法被称作“空子类测试”：

```
#!/usr/bin/perl

# use Foo
use Local::Foo;

#my $object = Foo->new();
my $object = Local::Foo->new( ... );
```

接下来的事情取决于我们想做什么。我们是想完全替换掉一个功能或方法，还是只想添加一点功能？我们给子类添加一个方法。我们可能想先调用 super 方法以便让原来的方法完成它的工作：

```
package Local::Foo

use base qw(Foo);

sub new
        {
        my( $class, @args ) = @_;

        ... munge arguments here

        my $self = $class->SUPER::new( @_ );

        ... do my new stuff here.
        }

1;
```

不过，有的时候这种方法不行，因为没法从原来的模块派生子类——可能是出于设计的原因或意外。举例来说，粗心的模块作者可能设计了一个只有一个参数的 bless 模块。由于没有第二个参数，bless 会把当前的软件包当成对象类型。无论我们在子类中做什么，只有一个参数的 bless 会在返回对象时忽略子类：

```
sub new
        {
        my( $class, @args ) = @_;

        my $self = { ... };

        bless $self;
        }
```

为了能从该类派生子类，我须要使用 new 的第一个参数，该参数被我们保存在了$class 中，并被作为第二个参数传给 bless：

```
sub new
        {
        my( $class, @args ) = @_;

        my $self = { ... };

        bless $self, $class;
        }
```

$class 中的值是我们使用过的原来的类名，而不是当前的软件包的。除非我们有足够充分的理由忽略原来的名字，我们应该一直在 bless 中使用它。

要测试这个模块，我们须要检查两个东西。首先，我须要确定继承类是能够工作的。这意味着我须要在继承树的某个地方找到父类 Foo 和用来创建对象 Local::Foo 的类：

```
# some file in t/
use Test::More;

my $object = Local::Foo->new();

foreach my $isa_class ( qw( Foo Local::Foo ) )
        {
        isa_ok( $object, $isa_class, "Inherits from $isa_class" );
        }
```

通常情况下，这些就足够了。如果我希望对象属于某个特殊的类而不是只是继承于该类，我就可以用 ref 准确地检查类的名字：

```
is( ref $object, 'Local::Foo', 'Object is type Local::Foo' );
```

内置的 ref 没有 Perl 5.8 之后的 Scalar::Util 模块的 blessed 函数好。它们的功能基本相同，但是内置的 ref 会在参数有问题的时候返回 undef。Blessed 函数不会在引用有问题时返回真：

```
use Scalar::Util qw(blessed);
is( blessed $object, 'Local::Foo', 'Object is type Local::Foo' );
```

一旦我们对自己能够派生子类感到满意，我们就可以覆盖子类的方法得到需要的行为。

一个例子：ExtUtils::MakeMaker
An ExtUtils::MakeMaker Example

有的时候人们知道他们的模块不会满足所有人的需要，因此会提供一种方法绕过程序的默认行为。

ExtUtils::MakeMaker 适用于大部分模块的安装程序，当它不能满足需要的时候我们也能非常方便地通过派生子类实现需要的功能。ExtUtils::MakeMaker 会使用特殊的子类名字 My。在作为它的硬编码的方法之前，它会在软件包 My 中查找同名的方法并且使用它们。

当 MakeMaker 工作的时候，它会把它的方法要求它做的事情写在文件 *Makefile* 中。它要写入的内容来自基类 ExtUtils::MM_Any，有可能也会来自于为了解决某个平台相关的问题而派生的子类 ExtUtils::MM_Unix 或 ExtUtils::MM_Win32。

在 Test::Manifest 模块中，我们须要改变测试工作的方式。我们希望 make test 能够根据指定的顺序运行测试文件，而不是根据 glob 命令从目录 t 返回的文件名的顺序。函数 test_via_harness 会生成 *Makefile* 的一节。我知道这一点，因为我在 *Makefile* 中找到了须要修改的部分并在模块中查找了相应的文本所在的函数：

```
package ExtUtils::MakeMaker;

sub test_via_harness {
        my($self, $perl, $tests) = @_;

        return qq{\t$perl "-MExtUtils::Command::MM" }.
               qq{"-e" "test_harness(\$(TEST_VERBOSE),
               '\$(INST_LIB)', '\$(INST_ARCHLIB)')" $tests\n};
}
```

进行一些解析和替换操作后，*Makefile* 中的输出大致是这个样子（根据平台结果可能会有所不同）：

```
test_dynamic :: pure_all
                PERL_DL_NONLAZY=1 $(FULLPERLRUN) "-MExtUtils::Command::MM" "-e"
                "test_harness($(TEST_VERBOSE), '$(INST_LIB)', '$(INST_ARCHLIB)')"
                $(TEST_FILES)
```

解决完所有的这些问题后，make test 会运行一个命令取得目录 t 下的所有文件名，并根据这个顺序执行它们。这促使模块的作者使用一些奇怪的名字比如 *00.load.t* 或 *99.pod.t* 来确保测试按照他们希望的顺序进行：

```
perl -MExtUtils::Command::MM -e 'test_harness( ... )' t/*.t
```

只要我们替换的代码能够做同样的事情，test_harness 是怎么工作的并不重要。在这里，我们不希望测试文件名来自于@ARGV，我们希望能够控制它们的顺序。

为了改变这个，须要用我们的函数来替换 test_harness。我们在软件包 MY 中定义了自己的 test_via_harness 子程序，这样就可以用任何喜欢的文本替换 test_via_harness 了。我们希望在软件包 Test::Manifest 中使用自己的函数。下面的程序指定了完整的软件包名字以使用正确的名字空间：

```
package Test::Manifest;

sub MY::test_via_harness
        {
        my($self, $perl, $tests) = @_;

        return qq|\t$perl "-MTest::Manifest" | .
               qq|"-e" "run_t_manifest(\$(TEST_VERBOSE), '\$(INST_LIB)', | .
        qq|'\$(INST_ARCHLIB)', \$(TEST_LEVEL) )"\n|;
        };
```

我们没有把文件名列表作为参数传进来，而是让子程序 run_t_manifest 调用了 get_t_files()，从文件 *t/test_manifest* 中获得文件名。Run_t_manifest()得到文件列表之后，会把它传给 Test::Harness::runtests()，这也是 test_harness()最后会调用的：

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

my $Manifest = catfile( "t", "test_manifest" );

sub run_t_manifest
        {
        ...;

        my @files = get_t_files( $level );

        ...;
        Test::Harness::runtests( @files );
        }

sub get_t_files
        {
        return unless open my( $fh ), $Manifest;

        my @tests = ();

        while( <$fh> )
               {
               ...;

               push @tests, catfile( "t", $test ) if -e catfile( "t", $test );
               }
        close $fh;

        return wantarray ? @tests : join " ", @tests;
        }
```

在文件 *t/test_manifest* 中，我们列出了须要运行的测试文件，也可以注释掉须要忽略的行。我们按照须要运行的顺序排列它们，运行时会遵循这个顺序：

```
load.t
pod.t
pod_coverage.t
#prereq.t
new.t
feature.t
other_feature.t
```


通过派生子类，我们就不用折腾模块 `ExtUtils::MakeMaker` 了。这样也避免了做不希望做的事情。我们得到了希望的功能，并且没有破坏模块。`ExtUtils::MakeMaker` 模块的源代码依然和其他人的一样。如果我们要改变 `ExtUtils::MakeMaker` 的其他行为的话，也可以做类似的修改。

其他的例子
Other Examples

在第15章中可以找到另外一个派生子类的例子。我们在该章中从 `Pod::Simple` 派生了子类。Sean Burke 专门为其他人派生子类写了这个模块。本书的大部分是用 pseudopod 写成的。它是 O'Reilly Media 的一种特殊的纯文本格式。我创建了自己的 `Pod::PseudoPod` 子类来把源文件转换成 HTML 页面供网站（注 6）和印刷部门使用。

对子程序进行封装
Wrapping Subroutines

有的时候我们只想把一个子程序或方法封装在另外一个里面，而不是替换掉它。这样我们就可以在运行子程序之前检查输入，并在返回结果给调用者之前清理、检查返回值。基本的想法如下：

```
sub wrapped_foo
        {
        my @args = @_;

        ...; # prepare @args for next step;

        my $result = foo( @args );

        ...; # clean up $result

        return $result;
        }
```

注 6：*Mastering Perl* 网站有本书的文本部分和源代码，网址是 *http://www.pair.com/comdog/mastering_perl*。

不过，为了正确地处理，我们须要处理不同的上下文。如果我们在列表上下文中调用 wrapped_foo，我们也须要在列表上下文中调用 foo。Perl 的子程序在不同的上下文中行为不同，Perl 程序员也期望子程序在不同的上下文中行为不同，这些都很常见。下面的这个基本的模板能够处理标量上下文、列表上下文和空上下文：

```
sub wrapped_foo
        {
        my @args = @_;

        ...; # prepare @args for next step;

        if( wantarray )             # list context
                {
                my @result = foo( @args );

                return @result;
                }
        elsif( defined wantarray ) # scalar context
                {
                my $result = foo( @args );
                ...; # clean up $result
                return $result;
                }
        else                        # void context
                {
                foo( @args );
                }
        }
```

事情甚至比这个还要复杂一些，不过 Damian Conway 的 Hook::LexWrap 模块简化了这一切。该模块允许我们在被封装的子程序前后分别加入预处理函数和后处理函数，其余中间的事情由该模块负责处理。该模块的接口简单：我们只须要使用 wrap 子程序并通过匿名子程序指定处理函数。封装后的程序是 sub_to_watch()，我们可以像调用一个普通的子程序一样调用它：

```
#!/usr/bin/perl

use Hook::LexWrap;

wrap 'sub_to_watch',
        pre  => sub { print "The arguments are [@_]\n" },
        post => sub { print "Result was [$_[-1]]\n" };

sub_to_watch( @args );
```

Hook::LexWrap 向@_中添加了一个额外的元素来保存返回值，这样我们就可以在后处理函数中访问$_[-1]得到返回值。

我们可以使用这个方法重写第 4 章的例子 divide 程序。在那个例子中，我们有一个子程序返回两个数的比例。在我虚构的例子中，我们传入了错误的参数，所以得到了错误的结果。这里是我们的那个子程序：

```
sub divide
        {
        my( $n, $m ) = @_;
        my $quotient = $n / $m;
        }
```


现在我们希望在传递参数之前检查一下参数、并在返回之前检查一下返回值。如果传入的参数和商是吻合的，那么子程序工作是正常的——可惜有人用错了参数。如果参数是正确的而商是错的，那么就是子程序有问题：

```
#!/usr/bin/perl

use Hook::LexWrap;

sub divide
        {
        my( $n, $m ) = @_;
        my $quotient = $n / $m;
        }

wrap 'divide',
        pre  => sub { print "The arguments are [@_]\n" },
        post => sub { print "Result was [$_[-1]]\n" };

my $result = divide( 4, 4 );
```

封装好子程序之后，我们可以像通常一样调用 `divide`。不过更重要的是，我们不须要改变程序中调用 `divide` 的地方，因为 `Hook::LexWrap` 做了一些奇妙的工作来替换子程序的定义，这样我们的程序看到的是封装后的版本。从而我们在没有编辑源代码的情况下修改了子程序。不用修改子程序，每次调用的时候，我们都能够看到额外的输出：

```
The arguments are [4 4 ]
Result was [1]
```

当我们删除 `wrap` 的时候，所有的东西会和之前的一样，我不必担心如何恢复我们的改动。

总结
Summary

我们不必改动模块的代码就能改变模块工作的方式。对于一个面向对象的模块来说，我们可以创建一个子程序来改掉我们不喜欢的部分。如果由于某种原因我们不希望派生子类，我们可以替换掉该模块的部分内容，就像我们可以这样处理任何模块一样。不过，不管我们怎么做，通常情况下我都不希望改变源代码（除非这是我们的模块，我们须要修复它），这样我们就不会把事情弄得更糟。

深入阅读
Further Reading

文档 *perlboot* 中有一个关于派生子类的 extended 的例子。该例子也在《Intermediate Perl》中。

我在 2005 年 7 月的《The Perl Journal》的文章“Wrapping Subroutines to Trace Code Execution”中介绍了 `Hook::Lex::Wrap` 模块：*http://www.ddj.com/dept/lightlang/184416218*。

命令 `diff` 和 `patch` 的手册讨论了它们的使用方法。`patch` 的手册尤其有指导性，它在文档末尾的一节中讨论了使用这些工具的注意事项和与其他程序员合作时应该考虑的一些实际问题。

第11章 配置Perl程序

Configuring Perl Programs

一旦有人知道你熟悉 Perl，他们可能会请你给他们写一个程序，或者要你修改一下你的某个程序。很有可能别人找到了你的某个灵巧的小程序，希望能够使用它，只是使用的方式稍微有些不同。

千万不要陷入创建或维护多个版本的程序的陷阱中。你应该让你的程序变成可配置的，这样你的用户就不须要接触源代码了。当用户接触到源代码的时候，任何意想不到的问题都有可能出现。他们所做的任何细小的改变都可能破坏整个程序——可能只是因为漏了一个分号。如果他们想修补一个问题应该找谁呢？对了——应该找你。只要做一点点额外的工作把你的程序变成可配置的，就能避免之后很多让人头痛的问题。

不要做的事情

Things Not to Do

最容易也是最糟糕的配置 Perl 程序的方法是简单地把一堆变量放在程序中，告诉用户在需要不同的功能的时候改变他们。然后用户须要打开我们的程序，改变变量的值来改变程序的行为。这会给用户足够的信心去改变其他的东西，即使我们警告过用户不要改变配置部分之外的任何东西。就算用户只在我们期望的范围内修改代码，他也可能会犯一个语法错误。不仅如此，如果他必须在多台机器上安装这个程序，那么最终他可能在每台机器上都有一个不同版本的程序。对该程序的任何修改或更新都须要他去编辑每一个版本：

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

my $Debug   = 0;
my $Verbose = 1;
my $Email   = 'alice@example.com';
my $DB      = 'DBI:mysql';

#### DON'T EDIT BEYOND THIS LINE !!! ###
```

我们并不希望用户了解我们的程序是什么，他们只须要知道它能够做什么、如何和它交互。我们并不在乎用户是否知道我们使用的语言、程序的工作方式，等等。我们希望他们能够完成他们的工作，这意味着我不希望他们不得不寻求我的帮助。我也不希望他们看我们的代码，因为我并不期望他们了解 Perl 语言。当然，他们还是可以看源代码（归根结底，我们确实喜欢开源软件），但是如果我们的程序写得足够好的话，他们就不须要看源代码。

虽然我已经说了这么多，有的时候把一个值写死在代码中也不是那么糟糕。不过下面的这种方法并不能算是真正的配置。当我们须要给某个数值一个名字以便重用的时候，我们可以使用编译选项 constant，它会创建一个返回该数值的子程序。下面我们把 PI 定义为一个常数，然后在需要它的地方使用它的名字 PI：

```
use constant PI => 3.14159;

my $radius = 1;
my $circumference = 2 * PI * $radius;
```

下面这种定义一个自己的子程序的方法要更加易读些，因为它表明了定义一个常量的意图。我们使用了一个空的参数列表，这样 Perl 不会试图把子程序名字后面的任何东西当作参数来处理。我们可以在程序的任何地方使用这个子程序，就像使用其他的子程序一样。可以把它们从模块中导出，也可以通过指定完整的软件包路径访问它们：

```
sub PI () { 3.14159 }
```

这种方式非常方便，能够找出某个数值，而且提供了一个简单的访问它的方法。虽然下面的这个例子并没有做很多事情，其实我们也可以把它弄得复杂点，可以通过访问数据库、从网络上下载某个东西或其他的方式来计算这个数：

```
{
my $days_per_year = $ENV{DAYS_PER_YEAR} || 365.24;
my $secs_per_year = 60 * 60 * 24 * $days_per_year;

sub SECS_PER_YEAR { $secs_per_year }
}
```

令人好奇的是，PI 和 SECS_PER_YEAR 这两个数除了相差一千万倍之外几乎都是相同的。如果我是在餐巾的背后计算的话，每年的秒数（忽略特殊的日子）大概是 3.15e7，这个数和 Pi 的一千万倍非常接近。

类似的，如果对 Perl 的变量更熟悉的话，我们也可以使用 Readonly 模块。如果我们试图修改任何一个变量，Perl 会给出警告。该模块也允许我们定义词法变量：

```
use Readonly;

Readonly::Scalar my $Pi       => 3.14159;
```

```
Readonly::Array  my @Fibonacci => qw( 1 1 2 3 5 8 13 21 );

Readonly::Hash   my %Natural   => ( e => 2.72, Pi => 3.14, Phi => 1.618 );
```

在 Perl5.8 及其之后的版本，我们可以去掉第二级的软件包名字，让 Perl 根据传递的值找出具体的名字：

```
use 5.8;
use Readonly;

Readonly my $Pi        => 3.14159;

Readonly my @Fibonacci => qw(1 1 2 3 5 8 13 21 );

Readonly my %Natural   => ( e => 2.72, Pi => 3.14, Phi => 1.618 );
```

把配置代码放在单独的文件中
Code in a Separate File

一个更复杂但是依然不太好的方法是把同样的配置代码放在一个单独的文件中，再从主程序中读入。我们把之前在程序开头的代码放在 *config.pl* 中。我们不能使用词法变量，因为它们的作用域仅仅限于它们的文件。这样做 *config.pl* 之外的程序会看不到它们，这可不是我希望的配置文件：

```
# config.pl
use vars qw( $Debug $Verbose $Email $DB );

$Debug   = 0;
$Verbose = 1;
$Email   = 'alice@example.com';
$DB      = 'DBI:mysql';
```

我们使用 `require` 导入配置信息，只是我们须要把它放在一个 `BEGIN` 块中，这样 Perl 会在编译程序的其他部分之前先看到 `use vars` 的声明。我们在《Intermediate Perl》的第 3 章中开始介绍模块时详细地介绍过 `BEGIN` 块：

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

BEGIN { require "config.pl"; }
```

当然，如果不介意去掉 `use strict` 的话（我并不想这样做），我们也可以不涉及这些技巧。不过还是有很多人会这样做，使用 Google（注 1）可以找到大量的使用 *config.pl* 的例子。

注 1：Google 有一个搜索开源代码的服务。试试在 *http://codesearch.google.com* 上查找引用 *config.pl* 的代码。

更好的方法
Better Ways

配置的含义是把希望用户能够改变的数据和代码的其他部分分离开来。这些数据可以来自多个数据源，不过完全由我们决定哪个数据源是适合我们的应用的。不是所有的情况都需要同样的方法。

环境变量
Environment Variables

环境变量为一个外壳程序（shell）下的所有进程指定了可以访问和使用的值。子程序可以使用同样的值，只是子程序没法为它们的父进程改变它们。大部分的外壳程序都会自动设置一些环境变量，比如把 HOME 作为家目录，把 PWD 作为当前工作的目录。在 Perl 中，这些变量会出现在哈希表%ENV 中。在大部分机器上，我们可以写一个 *testenv* 程序来看看环境是如何设置的：

```
#!/usr/bin/perl

print "Content-type: text/plain\n\n" if $ENV{REQUEST_METHOD};

foreach my $key ( sort keys %ENV )
        {
        printf "%-20s %s\n", $key, $ENV{$key};
        }
```

注意使用$ENV{REQUEST_METHOD}的那行代码。如果我们是把这个程序当作 CGI 程序使用，Web 服务器会设置一些环境变量，其中也包括一个名为 REQUEST_METHOD 的变量。如果我们的程序发现它正作为一个 CGI 程序在运行，它就打印出一个 CGI 响应头。否则，它发现我们是在终端运行，从而忽略该部分。

我特别喜欢在 CGI 程序中使用环境变量，因为我们可以在*.htaccess* 文件中设置环境变量。这个例子是 Apache 的，它需要模块 mod_env，不过其他的服务器可能也有类似的功能：

```
# Apache .htaccess
SetEnv DB_NAME mousedb
SetEnv DB_USER buster
SetEnv DB_PASS pitrpat
```

我们在*.htaccess* 设置的任何变量都会出现在程序中，而且可以在所有受该文件影响的程序中使用。如果要改变密码，我就只用在一个地方改变。不过要小心的是，既然网页服务器的用户可以读到这个文件，其他的用户也有可能得到这个信息。不过，不管你怎么处理它，最终 Web 服务器还是须要知道这些值，所以我们没法永远把它们隐藏起来。

特殊环境变量
Special Environment Variables

Perl 会使用一些环境变量。环境变量 PERL5OPT 会模拟在命令行中使用的开关。PERL5LIB

可以把目录添加到搜索模块的路径上。这样，我们可以在不改变程序的条件下改变 Perl 的行为。

如果要想添加更多的选项，就像在 shebang 行指定的一样，我们可以把选项添加到 PERL5OPT。这种方式会非常方便，比如说我们希望永远在运行时打开警告：

```
% export PERL5OPT=w
```

变量 PERL5LIB 的值会出现在代码的 use lib 指令中。当我们想在不同的机器上运行同样的程序时我们往往不得不使用它。虽然我希望世界上的所有的文件系统的布局、存储模块的目录、家目录的位置和其他文件都是一样的，不幸的是人们并没有这样做。不过我们可以从外部设置这些变量，而不是编辑程序来改变本地模块的路径。一旦在登录程序或者 *Makefile* 中设置好之后，这些设置就会一直不变，我们再也不用考虑它。当 Perl 函数库使用新的目录时，我们也不用编辑所有的程序：

```
% export PERL5LIB=/Users/brian/lib/perl5
```

打开额外的输出
Turning on Extra Output

在开发程序查找某个 bug 的时候，我往往会添加大量额外的 print 语句来检查程序的状态。我们的程序会离最终的正常工作状态越来越接近，程序中会留下大量的 print 语句。但是我们不须要每次运行程序的时候都执行它们；我们只希望在有问题的时候执行它们。

类似的，在某些情况下我可能希望在终端运行程序的时候，我们的程序能够显示正常的输出，而从外壳程序 cron 中运行时输出比较少，如此等等。

不管是哪种情况，我们可以定义一个环境变量来打开或关掉额外的 print。使用环境变量避免了修改程序。环境变量的作用域可长可短。通过在运行程序时设置环境变量，我们的设置在整个程序运行的时候都有效：

```
$ DEBUG=1 ./program.pl
```

如果在整个会话中设定了环境变量，也可以在整个会话中有效：

```
$ export DEBUG=1
$ ./program.pl
```

现在我可以使用这些变量来配置我们的程序。我们可以从环境变量中得到这些配置，而不是把它们写在程序中：

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

my $Debug   = $ENV{DEBUG};
my $Verbose = $ENV{VERBOSE};
```

```
...
print "Starting processing\n" if $Verbose;
...
warn "Stopping program unexpectedly" if $Debug;
```

我们可以在命令行上直接设定环境变量，这样环境变量只对该进程有效。我们可以用程序 *testenv* 验证变量的设置。有的时候我可能会犯一些奇怪的引用或特殊字符处理的错误，所以 *testenv* 对于帮助我们找出为什么变量的值不是我们期望的非常方便：

```
% DEBUG=1 testenv
```

我们也可以为一个会话中的所有进程设定环境变量。不过不同的外壳程序的语法略有不同：

```
% export DEBUG=2   # bash
$ setenv DEBUG=2   # csh
C:> set DEBUG=2    # Windows
```

如果没有设置程序中使用的环境变量，Perl 会报告有变量没有初始化，因为我们打开了警报。在上面的程序中当我们在 `if` 修饰符中检查变量的值时，会因为使用没有定义的值而得到警告。为了去掉警告，我们可以设置默认的值。短路操作符`||`在这里非常地方便：

```
my $Debug   = $ENV{DEBUG}     || 0;
my $Verbose = $ENV{VERBOSE}   || 1;
```

有的时候 `0` 是一个合法的值，即使它表示假，我们不希望在变量有定义的情况下使用默认值。在这些情况下，使用三元操作符和 `defined` 会非常方便：

```
my $Debug   = defined $ENV{DEBUG} ? $ENV{DEBUG} : 0;
my $Verbose = defined $ENV{VERBOSE} ? $ENV{VERBOSE} : 1;
```

Perl 5.10 中还有一个 defined-or（`//`）操作符。它会计算左边的操作数的值，如果有定义的时候会返回该值，即使返回值为假。否则，它会返回下一个数：

```
my $Verbose = $ENV{VERBOSE} // 1;  # new in Perl 5.10?
```

操作符`//`最早作为一个新的语法出现在 Perl 6 中。由于它实在太棒了，人们也把它加到了 Perl 5.10 中。和所有的新特性一样，我须要把使用它得到的好处和损失的向后兼容性相比较。

有些变量可能会影响别的变量。我们可能希望`$DEBUG` 为真的时候`$VERBOSE` 也为真，否则为假：

```
my $Debug   = $ENV{DEBUG}     || 0;
my $Verbose = $ENV{VERBOSE}   || $ENV{DEBUG} || 0;
```

当我们决定大量使用环境变量之前，还须要考虑一下程序的目标用户和使用的平台。如果运行的平台不支持环境变量，我们就得考虑其他的配置程序的方法。

命令行开关
Command-Line Switches

命令行开关是程序的参数。它们往往被用来改变程序的行为，不过在一些特殊情况下它们什么也不做，只是为了保持程序外部接口的兼容性。在《Advanced Perl Programming》中，Simon Cozens 谈到了 Perl 程序员不停地重复发明的各种东西（和 reinventing consistently 不同）。命令行参数是其中的一个。确实如此，当我在 CPAN 上查找这些模块的时候，我找到了 `Getopt::Std`、`Getopt::Long` 和其他 87 个名字中有 `Getopt` 的模块。

我们可以用多种方式处理命令行开关。如何处理它们完全由我们决定。它们只是我的 Perl 程序的参数，这些处理命令行开关的程序只是从`@ARGV` 中把它们取出来，做一些必要的处理，把它们变成程序中能用的形式，同时避免破坏其他非开关的参数。只要考虑一下人们处理命令行参数的多种不同的方式，我们就不会对有这么多个处理它们的模块而感到惊讶了。即使非 Perl 程序也在使用命令行参数上没有多少一致性。

下面的模块列表并不是完整的，每种情况我至少涵盖了两个 Perl 模块。我对精巧的参数处理并没有什么兴趣，自然也没有使用过这些模块中的绝大部分。虽然 CPAN 有 89 个模块与“Getopt”匹配，我只看了那些可以正常安装的模块和文档中介绍不需要太多工作就可以使用的模块。

1. 单字符的开关，每个开关前面都有连字符。我们须要单独处理这些开关（`Getopt::Easy`、`Getopt::Std`，Perl 的`-s` 开关）

```
% foo -i -t -r
```

2. 单字符的开关，每个开关前面都有连字符。开关可能带有数值（必须的或可选的），在开关和数值之间可能有分隔字符（`Getopt::Easy`、`Getopt::Std`、`Getopt::Mixed`、Perl 的`-s` 开关）

```
% foo -i -t -d/usr/local
% foo -i -t -d=/usr/local
% foo -i -t -d /usr/local
```

3. 放在一起的单字符的开关，也被称作捆绑的或是聚集的开关。每个字符表示单独的东西（`Getopt::Easy`、`Getopt::Mixed`、`Getopt::Std`）：

```
% foo -itr
```

4. 单连字符的多字母开关，可能带有值。（Perl 的`-s` 开关）：

```
% foo -debug -verbose=1
```

5. 双连字符的多字母开关，也支持可能会放在一起的单连字符的单字母开关（Getopt::Atrribute、Getopt::Long、Getopts::Mixed）：

```
% foo --debug=1 -i -t
% foo --debug=1 -it
```

6. 用双连字符表示开关的结束。有的时候参数也会以连字符开始，所以外壳程序提供了一种方式表示开关的结束（Getopt::Long、Getopts::Mixed和-s，如果我们不考虑错误的诸如${-debug}之类的变量名的话）：

```
% foo -i -t --debug -- --this_is_an_argument
```

7. 可以用不同形式表示相同意思的开关（Getopt::Lucid、Getopts::Mixed）：

```
% foo -d
% foo -debug
```

8. 完全特殊的表示方法，可能有各种sigils或压根没有（Getopt::Declare）：

```
% foo input=bar.txt --line 10-20
```

开关-s
The -s Switch

我们也可以不用外部模块来处理命令行开关。如果需求不是太复杂的话，可以使用Perl的-s开关。使用这个编译选项后，Perl会把程序的开关转化成软件包变量。它可以处理单连字符和双连字符（双连字符只是一个单连字符和一个以单连字符开头的名字）。开关可以带参数，也可以不带。我们可以在命令行或shebang行指定-s：

```
#!/usr/bin/perl -sw
# perl-s-abc.pl
use strict;

use vars qw( $a $abc );

print "The value of the -a switch is [$a]\n";
print "The value of the -abc switch is [$abc]\n";
```

开关没有值的时候，Perl会把该开关对应的变量设置为1。开关中用等号指定了值的时候（这是唯一指定值的方式），Perl会把变量设置为指定的值：

```
% perl -s ./perl-s-abc.pl -abc=fred -a
The value of the -a switch is [1]
The value of the -abc switch is [fred]
```

我们也可以使用双连字符供-s处理：

```
% perl -s ./perl-s-debug.pl --debug=11
```

这种方式会导致 Perl 产生一个非法的名为${'-debug'}的变量，虽然该变量不能在 strict 模式中使用。该方式使用符号引用以绕过 Perl 的变量命名规则，所以我们不得不把变量名字放在花括号中。该方式还绕过了声明变量的 strict 规则，所以我们不得不关掉 strict 模式的 refs 检查才能使用这些变量：

```
#!/usr/bin/perl -s
# perl-s-debug.pl
use strict;

{
no strict 'refs';
print "The value of the --debug switch is [${'-debug'}]\n";
print "The value of the --help switch is [${'-help'}]\n";
}
```

之前的命令行的输出如下：

```
The value of the --debug switch is [11]
The value of the --help switch is []
```

其实我们并不须要使用双连字符。Perl 的开关-s 并不支持把开关聚在一起，所以我们不须要使用双连字符来表示长的开关名。使用以非法字符开头的变量名是一种便捷的把配置用的变量和普通变量区分出来的方法。不过，我不太赞成这种做法。

模块 Getopt

Getopt Modules

我没法介绍可能用到或刚才提到的所有模块，所以这里我只介绍两个和 Perl 一起发布的模块：Getopt::Std 和 Getopt::Long（从 Perl 5 开始两个模块都可以使用了）。你须要考虑一下你需要的功能是否超过了这两个模块支持的范围。标准的 Perl 安装版一定会有这些模块，这两个模块也不会处理那些可能会让用户感到困惑的奇怪的语法。

Getopt::Std

模块 Getopt::Std 可以处理单字符的开关，可以把开关聚到一起，也可以指定开关的值。该模块有两个函数，一个有“s”，getopt，一个没有 s，getopts。不过它们的行为只有细微的不同（我从来没有找到一种方法把它们区分开）。

函数 getopt 期望每个开关都有一个值（比如：-n=1），如果开关没有值的话不会接受任何开关（比如：-n）。它的第一个参数是一个表示期望的开关的字符串。第二个参数是一个哈希表的引用，参数的名字和值会被设置为哈希表的键和值。下面我们在程序的开头调用 getopt：

```
#!/usr/bin/perl
# getopt-std.pl
use strict;

use Getopt::Std;
```

```
getopt('dog', \ my %opts );

print <<"HERE";
The value of
   d      $opts{d}
   o      $opts{o}
   g      $opts{g}
HERE
```

当我们用一个开关和一个值调用这个程序的时候，我们看到 getopt 把开关设定成了指定的值：

```
$ perl getopt-std.pl -d 1
The value of
   d      1
   o
   g
```

当我们用同样的参数调用同样的程序但是没有指定值的时候，getopt 不会设置任何值：

```
$ perl getopt-std.pl -d
The value of
   d
   o
   g
```

getopt 也有一种单参数的形式。不过这里我们不会介绍，因为它会创建全局变量，而我们一直避免创建全局变量。

函数 getopts（带 s 的那个）工作方式稍微有些不同。它可以处理没有参数的开关，并把开关对应的值设置为 1。为了区别有参数和没有参数的开关，我们在有参数的开关后面加上了一个冒号。

在这个例子中，开关 d 和 o 是二进制开关，开关 g 可以带一个参数：

```
#!/usr/bin/perl
# getopts-std.pl

use Getopt::Std;

getopts('dog:', \ my %opts );

print <<"HERE";
The value of
   d      $opts{d}
   o      $opts{o}
   g      $opts{g}
HERE
```

当我们给这个程序指定开关 g 及其对应的参数 foo 和开关-d 的时候，getopts 会设置这些开关对应的值：

```
$ perl getopts-std.pl -g foo -d
The value of
   d      1
   o
   g      foo
```


如果一个开关会带一个参数，它会把之后的任何东西作为参数处理。比如说如果我忘记了为开关-g提供一个值，它会把下一个开关作为-g的值：

```
% ./getopts.pl -g -d -o
The value of
   d
   o
   g      -d
```

反过来，如果给一个不带参数的开关传递了一个参数，程序就会不能正常工作。给-d传一个值会让getopts停止处理参数：

```
$ perl getopts-std.pl  -d foo -g bar -o
The value of
   d      1
   o
   g
```

Getopt::Long

模块Getopt::Long可以处理单字符开关、聚在一起的单字符开关和以双连字符开始的开关。使用的时候，我们给函数GetOptions传递一列键和值的对，键指定了开关的名字，值对应的是GetOptions用来存放开关的值的变量的引用：

```
#!/usr/bin/perl
# getoptions-v.pl

use Getopt::Long;

my $result = GetOptions(
        'debug|d'   => \ my $debug,
        'verbose|v' => \ my $verbose,
        );

print <<"HERE";
The value of
   debug          $debug
   verbose        $verbose
HERE
```

在这个例子中，我也用竖线|指定了某些开关的别名。我们必须把pair中的键引起来，因为|是一个Perl操作符（我们会在第16章中介绍）。由于-verbose和-v都会让Perl设置变量$verbose，我们可以使用-verbose或-v让程序输出更详细的信息：

```
$ perl getoptions-v.pl -verbose
The value of
   debug
   verbose       1

$ perl getoptions-v.pl -v
The value of
   debug
   verbose 1

$ perl getoptions-v.pl -v -d
The value of
   debug              1
   verbose       1

$ perl getoptions-v.pl -v -debug
The value of
   debug         1
   verbose       1

$ perl getoptions-v.pl -v --debug
The value of
   debug         1
   verbose       1
```

如果只指定键的名字，开关会被当成布尔量处理，所以我们得到的结果是真或是假。我们也可以给 GetOPtions 提供更多关于开关的信息，这样 Perl 就能够知道期望的值的类型。在 GetOptions 中，我们可以在开关的名字后面用等号指定选项。选项=i 表示整数，=s 表示字符串，空表示只是一个标志量，就像我们之前的那样。还有其他的类型。如果我们给开关传递了错误类型的数据，比如说在需要数的时候传了一个字符串，GetOptions 不会设置相应的变量（所以说它不会把字符串转化成数字 0）：

```
#!/usr/bin/perl
# getopt-long-args.pl

use Getopt::Long;

my $result = GetOptions(
        "file=s" => \ my $file,
        "line=i" => \ my $line,
        );

print <<"HERE";
The value of
        file            $file
        line            $line
HERE
```

在下面的例子中，开关-line 希望接收一个整数。当我们把 1 传给它的时候一切正常。当我们传给它一个实数的时候会得到警告：

```
$ perl getopt-long-args.pl -line=-9
The value of
        file
        line          -9
$ perl getopt-long-args.pl -line=9.9
Value "9.9" invalid for option line (number expected)
The value of
        file
        line
```


我们可以使用@告诉 GetOptions 某个开关可以接受多个值。要想让-file 能够接受多个值，我们把@放在=s 的后面，并且把得到的值赋给数组@files，而不是一个标量：

```
#!/usr/bin/perl
# getopt-long-mult.pl

use Getopt::Long;

my $result = GetOptions(
        "file=s@" => \ my @files,
        );


{
local $" = ", ";

print <<"HERE";
The value of
        file          @files
HERE
}
```

要想使用这个功能，我们须要在命令行多次使用该开关：

```
$ perl getopt-long-mult.pl --file foo --file bar
The value of
        file            foo, bar
```

配置文件
Configuration Files

如果我们想多次使用同一个值，或者希望知道多个值，我们可以把它们放在一个可以被程序读到的文件中。而且，就像有多种命令行选项解析器可以供我们选择一样，我们也有一些配置文件解析器可以选用。

我的建议是先根据你的情况选择合适的配置文件格式，然后选择合适的模块来处理该格式。

ConfigReader::Simple

我对模块 ConfigReader::Simple 有些偏心，因为是我维护着它（虽然我不是原作者）。它可以处理多个文件（比如说，支持用户专有的配置文件覆盖全局的配置文件），并且有一个简单的单行语法：

```
# configreader-simple.txt
file=foo.dat
line=453
field value
field2 = value2
long_continued_field This is a long \
        line spanning two lines
```

这个模块能够处理所有的这些情况：

```
#!/usr/bin/perl
# configreader-simple.pl

use ConfigReader::Simple;

my $config = ConfigReader::Simple->new(
        "configreader-simple.txt" );
die "Could not read config! $ConfigReader::Simple::ERROR\n"
        unless ref $config;

print "The line number is ", $config->get( "line" ), "\n";
```

Config::IniFiles

Windows 的用户更熟悉 INI 文件。也有模块能够处理这种格式。这种文件的基本格式是把配置分成以放在方括号中的标题的若干组。每个标题下的参数只适用于该标题。参数的键和值用等号分隔（在有些格式中用的是逗号）。注释行以分号开头。INI 格式甚至还有一种续行功能。模块 Config::IniFiles 及一些其他模块可以支持这种格式。下面这个 INI 文件是我们在本书中将要用到的：

```
[Debugging]
;ComplainNeedlessly=1
ShowPodErrors=1

[Network]
email=brian.d.foy@gmail.com

[Book]
title=Mastering Perl
publisher=O'Reilly Media
author=brian d foy
```

我们可以解析这个文件并得到不同的节的值：

```
#!/usr/bin/perl
# config-ini.pl
```

```
use Config::IniFiles;

my $file = "mastering_perl.ini";

my $ini = Config::IniFiles->new(
        -file    => $file
        ) or die "Could not open $file!";

my $email = $ini->val( 'Network', 'email' );
my $author = $ini->val( 'Book', 'author' );

print "Kindly send complaints to $author ($email)\n";
```

除了读取配置文件之外，还可以使用模块 Config::IniFiles 改变参数的值，添加或减少参数，或者重写 INI 文件。

Config::Scoped

Config::Scoped 支持的格式和 INI 文件比较类似，可以把参数限定在某一节内，不过它更加复杂。它允许嵌套的节，可以执行 Perl 代码（记得我之前提到的吗？），并支持有多个值的键：

```
book {
       author = {
              name="brian d foy";
              email="brian.d.foy@gmail.com";
              };
       title="Mastering Perl";
       publisher="O'Reilly Media";
}
```

该模块可以解析配置文件并把结果放在一个 Perl 数据结构中返回：

```
#!/usr/bin/perl
# config-scoped.pl

use Config::Scoped;

my $config = Config::Scoped->new( file => 'config-scoped.txt' )->parse;
die "Could not read config!\n" unless ref $config;

print "The author is ", $config->{book}{author}{name}, "\n";
```

AppConfig

可能 Andy Wardley 的模块 AppConfig 是所有处理配置文件的程序中功能最强大的，它提供了一个统一的接口，能够处理命令行参数、配置文件、环境变量、CGI 参数和很多其他的东西。它可以处理 ConfigReader::Simple 支持的单行模式的配置，Config::INI 支持的 INI 格式，和很多其他的格式。Andy 把 AppConfig 用在了他的非常流行的模板系统 Template Toolkit 中。

下面是用 AppConfig 写的 INI 文件的阅读器，使用的是之前用过的同样的 INI 文件：

```
#!/usr/bin/perl
# appconfig-ini.pl

use AppConfig;

my $config = AppConfig->new;

$config->define( 'network_email=s'    );
$config->define( 'book_author=s'      );
$config->define( 'book_title=s'       );
$config->define( 'book_publisher=s'   );

$config->file( 'config.ini' );

my $email  = $config->get( 'network_email' );
my $author = $config->get( 'book_author' );

print "Kindly send complaints to $author ($email)\n";
```

这个程序稍微要复杂些。因为 AppConfig 可以做很多不同的事情，我们要告诉它须要做什么。创建$config 对象后，我们须要告诉它希望的参数和参数的类型。AppConfig 使用了和 Getopt::Long 一样的语法格式。对于 INI 格式，AppConfig 会把配置节的名字作为参数的前缀来处理，从而把配置文件变成扁平格式的。我们的程序会汇报没有定义的参数，AppConfig 在处理 INI 文件的注释行;complainneedlessly 时也遇到了困难：

```
debugging_;complainneedlessly: no such variable at config.ini line 2
debugging_showpoderrors: no such variable at config.ini line 3
Kindly send complaints to brian d foy (brian.d.foy@gmail.com)
```

有了程序 AppConfig 后，我们可以在不改变程序的情况下改变配置文件的格式。AppConfig 会自动适应新的格式。只要我们更新配置文件的格式，我们之前的程序就能正常工作。下面是我们新的文件格式：

```
network_email=brian.d.foy@gmail.com
book_author=brian d foy
```

只要稍微做一些修改，就可以让我们的程序能够处理命令行参数。当我们不带参数调用 $config->args()时，AppConfig 会调用 Getopt::Long 处理@ARGV：

```
#!/usr/bin/perl
# appconfig-args.pl

use AppConfig;

my $config = AppConfig->new;

$config->define( 'network_email=s'  );
```

```
$config->define( 'book_author=s'    );
$config->define( 'book_title=s'     );
$config->define( 'book_publisher=s' );

$config->file( 'config.ini' );

$config->args();

my $email  = $config->get( 'network_email' );
my $author = $config->get( 'book_author' );

print "Kindly send complaints to $author ($email)\n";
```

当我们重新运行程序并从命令行给 network_email 指定另一个值的时候，指定的值会覆盖文件中的配置，因为我们是在$config->file 之后调用的$config->args：

```
$ perl appconfig-args.pl
Kindly send complaints to brian d foy (brian.d.foy@gmail.com)

$ perl appconfig-args.pl -network_email bdfoy@cpan.org
Kindly send complaints to brian d foy (bdfoy@cpan.org)
```

其实，AppConfig 比我目前展示的要复杂得多，它可以做很多事情。我在本章结尾的“深入阅读”中列出了一些关于 AppConfig 的文章。

其他格式的配置文件
Other Configuration Formats

还有很多其他格式的配置文件，很可能每种格式都有一个相应的 Perl 模块。Win32::Registry 可以访问 Windows 注册表，Mac::PropertyList 可以处理 Mac OS X 的 plist 格式，Config::ApacheFile 可以解析 Apache 的配置文件。你可以浏览 CPAN 上 Config:: 模块的列表来找到你需要的格式。

有不同名字的脚本程序
Scripts with a Different Name

我们的程序也可以根据名字的不同来做不同的事情。程序的名字会出现在 Perl 的特殊变量 $0 中，你可能在 shell 编程中接触过。通常情况下，每个程序只有一个名字。不过，我们可以给文件创建链接（符号链接或硬链接）。当我们用某个名字运行程序时，可以相应地设置不同的配置：

```
if( $0 eq ... )       { ... do this init ... }
elsif( $0 eq ... )    { ... do this init ... }
...
else                  { ... default init ... }
```

除了改变程序的名字之外，我们也可以把程序嵌入到另一个程序中，由外层的程序负责设置环境变量、用正确的命令行开关和参数调用程序。这样，我们节省了很多设置变量的输入工作：

```
#!/bin/sh

DEBUG=0
VERBOSE=0
DBI_PROFILE=2

./program -n some_value -m some_other_value
```

交互和非交互程序
Interactive and Noninteractive Programs

有的时候我们希望程序能够自己判断出是应该给出输出结果还是要求给一些输入。当我们从命令行运行的时候，我们希望能够看到输出以了解程序在做什么。当我们从 cron（或者一些其他的任务调度程序）运行的时候，我们不希望看到输出。

真正的问题并不一定是程序是不是交互的，而是我们能否发送输出到终端或从终端知道结果。

我们可以通过检查 STDOUT 来判断输出是否会被送到终端。使用文件测试-t 可以知道分解句柄是否被连接到了终端。通常情况下，通过命令行调用时都是连着终端的：

```
$ perl -le 'print "Interactive!" if -t STDOUT'
Interactive!
```

如果我们重定向了 STDOUT，可能是重定向到了 *output.txt*，它就不再连接到终端，我们的测试程序也就不会有任何输出：

```
$ perl -le 'print "Interactive!" if -t STDOUT' > output.txt
```

不过，这可能不是我们的本意。因为我们是从命令行运行程序的，我们可能还是希望能够得到同样的输出。

如果我们希望知道是否可以给用户输入提示符，我们可以检查 STDIN 是否连接着终端。不过，我们也应该检查一下我们的提示符是否会出现在用户能够看到的地方：

```
$ perl -le 'print "Interactive!" if( -t STDIN and -t STDOUT )'
Interactive!
```

我们必须注意自己的真正意图并确保所做的检查是正确的。Damian Conway 的 IO::Interactive 可能会有很多的帮助，它可以处理各种特殊情况，还能判断一个程序是否处在交互状态下：

```
use IO::Interactive qw(is_interactive);

my $can_talk = is_interactive();
print "Hello World!\n" if $can_talk;
```

Damian 还提供了一个非常有用的功能：函数 interactive。这样我们就不用在所有的 print 语句中使用条件判断了。他的 interactive 函数能够在程序处于交互状态的时候返回文件句柄 STDOUT，在其他情况下返回 null 句柄。我们可以这样写一个正常的 print 语句：

```
use IO::Interactive qw(interactive);

print { interactive() } "Hello World!\n";
```


我们必须在 interactive() 调用时加上花括号，因为它不是一个简单的引用。我们也没有在括号后加上逗号。程序处于交互模式的时候会给出输出，不在的时候不会有输出。

还有很多其他的方法使用 interactive 函数。我们可以把 interactive 的返回值赋给一个标量，然后在 print 语句中把该标量作为文件句柄：

```
use IO::Interactive qw(interactive);

my $STDOUT = interactive();

print $STDOUT "Hello World!\n";
```

perl 的 Config 模块

perl's Config

模块 Config 提供了一个含有 perl 二进制程序的编译选项的哈希表。其中大部分选项表示的是 Configure 程序发现的系统的功能或编译时我们提供的问题的答案。

比如说，如果我们想抱怨 perl 的某个功能，我们可以检查一下 cf_email 的值。它应该是当你有关于 perl 程序的问题时联系的人（或者是角色），不过能否得到回信就要看运气了！

```
#!/usr/bin/perl

use Config;

print "Send complaints to $Config{cf_email}\n";
```

如果我们想看看 perl 程序的机器名（就是说 Config 是否正确地找到了它，或者编译 perl 时是否在同一台机器上），我们可以看看 myhostname 和 mydomain（虽然我们可以通过其他方式得到）：

```
#!/usr/bin/perl

use Config;

print "I was compiled on $Config{myhostname}.$Config{mydomain}\n";
```

如果想要看看我们的 perl 是否支持多线程，也可以检查相应的编译选项：

```
#!/usr/bin/perl

use Config;

print "has thread support\n" if $Config{usethreads};
```

不同的操作系统
Different Operating Systems

我们可能希望程序能够根据运行平台的不同而做不同的事情。在 Unix 平台上，我们可能会加载某个模块，而在 Windows 上可能会加载另外一个。Perl 知道它运行的平台是什么，并且会在$^O 中保存一个唯一的字符串（助记符：O 是操作系统 Operating system 的第一个字母），我们可以用这个字符串来判断该做什么。Perl 在编译和安装的时候确定该字符串的值。$^O 的值和$Config{'osname'}相同。如果我们需要更加详细的信息，可以使用$Config{'archname'}。

不过，在准确地指定需要的操作系统时我们要格外小心。表 11-1 显示了$^O 中流行的操作系统的名字，文档 *perlport* 中列出了更多的系统。注意我们不能通过模式 m/win/i 来检查 Windows，因为 Mac OS X 的名字是 darwin。

表 11-1：各种操作系统的$^O 的值

操作系统	$^O
Mac OS X	darwin
Mac Classic	Mac
Windows	Win32
OS2	OS2
VMS	VMS
Cygwin	Cygwin

我们可以根据操作系统来选择性地加载模块。举例来说，和 Perl 一起分发的模块 File::Spec 其实是一个 facade，它封装了多个操作系统相关的模块。下面是该模块的全部代码。它定义了一个哈希表%module 来把$^O 的值映射到须要加载的模块。然后，它使用 requires 来确保相应的模块必须存在。因为所有的模块都有同样的接口，使用该模块的程序员不会注意到不同平台的区别：

```
package File::Spec;

use strict;
use vars qw(@ISA $VERSION);

$VERSION = '0.87';

my %module = (MacOS   => 'Mac',
              MSWin32 => 'Win32',
              os2     => 'OS2',
              VMS     => 'VMS',
              epoc    => 'Epoc',
              NetWare => 'Win32',  # Yes, File::Spec::Win32 works on↵
                           NetWare.
              dos     => 'OS2',    # Yes, File::Spec::OS2 works on↵
                           DJGPP.
              Cygwin  => 'Cygwin');
```

```
my $module = $module{$^O} || 'Unix';

require "File/Spec/$module.pm";
@ISA = ("File::Spec::$module");

1;
```


总结
Summary

我们不用在程序中把用户定义的数据写成代码。我们有各种方法来让用户不用看源代码就可以指定配置和运行选项。Perl 有很多的模块可以处理命令行参数，在 CPAN 上有更多的模块，有些格式甚至有多种模块可供选择。虽然没有一种技术可以适应所有的情况，使用这些模块可以避免用户与源代码打交道，从而也减少了破坏源代码的可能。

深入阅读
Further Readming

文档 *perlport* 讨论了各种平台的不同之处，并介绍了如何在程序中区分它们。

Teodor Zlatanov 为 IBM developerWorks 写了一系列关于 `AppConfig` 的文章："Application Configuration with Perl"（*http://www-128.ibm.com/developerworks/linux/library/l-perl3/index.html*），"Application Configuration with Perl, Part 2"（*http://www-128.ibm.com/developerworks/linux/library/l-appcon2.html*）和"Complex Layered Configurations with AppConfig"（*http://www-128.ibm.com/developerworks/open source/library/l-cpappconf.html*）。

Randal Schwartz 在 2005 年 7 月的《Unix Review》的专栏中讨论了 `Config::Scoped` *http://www.stonehenge.com/merlyn/UnixReview/col59.html*。

第 12 章

检查和汇报错误

Detecting and Reporting Errors

在程序中，很多地方都有可能出问题，包括编程时带来的问题、错误或遗漏的输入、外部资源无法访问，等等。Perl 并没有任何内置的错误处理机制，但是它知道什么样的错误它无法处理，并且能够告诉我们错误的类型，然后完全由我们——Perl 程序员确保程序所做的事情是正确的，并且由我们决定如何在程序出错的时候做一些弥补错误的工作。

Perl 错误处理的基础知识

Perl Error Basics

Perl 有 4 个特殊变量用来汇报错误：`$!`、`$?`、`$@`和`$^E`。每个变量用于汇报不同类型的错误。表 12-1 显示了这 4 个变量及其相应的描述，*perlvar* 中也有对它们的介绍。

表 12-1：Perl 的汇报错误的特殊变量

变量	英文名的变量	描述
`$!`	`$ERRNO and $OS_ERROR`	从操作系统或库函数调用得到的错误
`$?`	`$CHILD_ERROR`	最近一次调用 `wait()` 得到的返回值
`$@`	`$EVAL_ERROR`	最近一次调用 `eval()` 得到的错误
`$^E`	`$EXTENDED_OS_ERROR`	操作系统特有的错误信息

操作系统错误

Operating System Errors

最简单的错误发生在当 Perl 要求系统做一些事情，而系统由于某些原因无法完成或没有完成的时候。在大部分情况下，Perl 的内置函数会返回 false 并把错误消息保存在`$!`中。如当我们试图读取一个不存在的文件的时候，`open` 函数会返回 false 并把失败的原因保存在`$!`中：

```
open my( $fh ), '<', 'does_not_exist.txt'
       or die "Couldn't open file! $!";
```

Perl 解析器是一个 C 程序，它的很多工作都是通过 C 语言的函数库来完成的。变量`$!`中存

放的是调用底层 C 函数的返回值。返回值的定义在头文件 *errno.h* 中。其他的应用程序可能也有相同名字的文件。文件 *errno.h* 把符号常量和错误值关联起来，并给每个错误提供了一个文字描述。下面是 Mac OS X 上的 *errno.h* 文件的片段：

```
#define EPERM   1   /* Operation not permitted */
#define ENOENT  2   /* No such file or directory */
#define ESRCH   3   /* No such process */
```

在 open 函数的例子中，我们按字符串方式解析变量$!得到了一个易读的错误消息。不过，变量$!其实有两重含义。这种有着不同的字符串值和数值的标量被称作**双重标量**（dualvars）（注 1）。它的数值是 C 函数返回的 errno 的值，字符串值则是易读的消息。我们可以通过设置$!看到两个值。下面，我们使用 printf 的格式符来得到同一变量的数值和字符串值：

```
for ($! = 0; $! <= 102; $!++)
        {
        printf("%d: %s\n", $!, $! );
        }
```

输出中显示了数值和字符串值：

```
1: Operation not permitted
2: No such file or directory
3: No such process
...
```

变量$!的值只有在调用库函数之后立刻使用才有效。我们应该在执行完要检查的表达式之后立刻使用$!，因为下一个 Perl 语句可能会做另一个库函数调用，从而可能改变$!的值，并生成一条不同的错误消息。库函数调用失败会设置$!，而调用成功则不会修改$!，也不会重置它。如果不立刻检查$!的值，我们可能会把它和错误的语句关联起来。

不过，这并不是问题的全部。哈希表%!和标量$!有一些奇妙的联系。%!的键是 *errno.h* 中定义的符号常量，比如 ENOENT。它是一个特殊的哈希表，只有和当前的$!相同的键才有对应的值。举例来说，当 Perl 没法打开 *does_not_exit.txt* 文件时，会把$!设置成 ENOENT 所代表的值。同时，Perl 会给$!{ENOENT}设定一个值。%!中的其他键不会有值。这意味着我们可以检查发生了什么事情，从而根据错误的类型采取相应的补救措施，从 open 函数的错误中恢复。

当 Perl 在程序的任何地方看到%!时，会自动加载 Errno 模块。该模块提供了和 *errno.h* 的符号常量同名的函数，我们可以利用这些函数得到任何错误的号码。不过，不使用%!也可以使用这些函数。我们可以自己加载该模块，甚至直接导入须要使用的符号：

注 1：我们可以使用 Scalar::Util 模块的 dualvar 函数创建自己的双重标量。

```
use Errno qw(ENOENT);

print "ENOENT has the number " . ENOENT . "\n";
```

在下面的这个例子中，我要把一些信息写到磁盘上。这些是非常重要的信息，所以我们须要格外小心确保数据保存好了。我不能简单地调用 die 然后期望别人会注意到发生了错误。事实上，如果是由于磁盘满了而不能写文件，甚至连警告消息也不会写到日志文件中：

```
#!/usr/bin/perl

use File::Spec;

my $file = 'does_not_exist.txt';
my $dir  = 'some_dir';
my $fh;

my $try = 0;
OPEN: {
last if $try++ >= 2;
my $path = File::Spec->catfile( $dir, $file );
last if open $fh, '>', $path;

warn "Could not open file: $!...\n";

if( $!{ENOENT} )      # 文件不存在
        {                   # 请确保目录存在
        warn "\tTrying to make directory $dir...\n";
        mkdir $dir, 0755;
        }
elsif( $!{ENOSPC} )  # Full disk
        {                   # 请尝试使用一个不同磁盘或挂载点
        warn "\tDisk full, try another partition...\n";
        $dir = File::Spec->catfile(
               File::Spec->rootdir,
               'some_other_disk',
               'some_other_dir'
               );
        }
elsif( $!{EACCES} )  # 没有权限
        {
        warn "\tNo permission! Trying to reset permissions...\n";
        system( '/usr/local/bin/reset_perms' );
        }
else
        {
        # 放弃尝试，通过邮件直接发送...
        last;
        }

redo;
}

print $fh "Something very important\n";
```

从这个小例子中可以看到，可以做很多工作从系统错误中恢复正常。我们总共尝试了 4 种方法试图从错误中恢复，并且会反复运行标记为 OPEN 的代码块直到成功或重试了足够多的次数（从错误中恢复的希望不大，所以放弃）为止。如果可以打开文件句柄，我们会用 last 跳出代码块。否则，我们将检查%!中哪个键的值为真。只会有一个键的值为真，它对应的应该是$!的值。如果得到的错误是文件不存在，我们会试图建立需要的目录。如果错误是磁盘上没有足够的空间，则尝试使用别的磁盘。如果没有正确的访问权限，则试图重置文件的权限。这些都是我在工作中遇到的问题。很多人有管理员的特权，会不恰当地改变一些重要文件的访问权限。因此我写了一个 setuid 程序来从数据库中得到正确的权限并重置文件的权限。我可以从任何程序中调用它，然后重新尝试打开文件。这种方法有效地避免了午夜的求救电话。从此以后，我意识到让任何人成为 root 用户是不明智的。

子程序错误
Child Process Errors

要想知道我们的程序启动的子程序出了什么问题，可以用$?获取子程序的退出状态。Perl 有多种方式可以和外部的程序通信，其中包括：

```
system( ... );
`....`;
open my($pipe), "| some_command";
exec( 'some command' );
my $pid = fork(); ...; wait( $pid );
```

如果什么地方出了问题，我们不会立刻知道错误。执行外部程序的时候，Perl 须要先分支（fork）出一个进程或创建一个当前进程的拷贝，然后调用 exec 把该进程变成须要执行的进程。因为 Perl 进程已经在运行了，大部分情况下我们一定能够运行它的拷贝，除非达到了进程数的上限，或者用完了内存。第一种情况——也就是 fork，不会有问题。$!中不会有任何错误，因为不会有 C 函数库错误。不过，一旦另一个进程准备好开始运行，它就不会把它的错误保存到父进程的$!变量中。它会在停止运行的时候把返回值传给父进程，Perl 会把返回值放在$?中。我们只有在用 close 或 wait 清理子程序的时候才会看到错误值：

```
close( $pipe ) or die "Child error: $?";
wait( $pid ) or die "Child error: $?";
```

变量$?的值比其他的错误变量要复杂很多。它其实是一个词（两个字节）。高字节才是子程序的返回值。我们可以把所有的字节右移 8 位来得到该值。这个值是运行的程序所特有的，所以我们须要查看它的文档才能知道具体的含义：

```
my $exit_value = $? >> 8;
```

$?的低 7 位保存的是终止程序的信号的代码，它只会在程序是由于某个信号而终止时才有效：

```
my $signal = $? & 127;  # or use 0b0111_1111
```

如果子程序造成了内核转储，低字节的第 8 位会被设为 1：

```
my $core_dumped = $? & 128; # or use 0b1000_000;
```

当我们使用 Perl 的 exit 时，exit 的参数会被作为进程的返回值。如果是其他的 Perl 程序调用它的话，该返回值会变成$?的高字节。我们的 *exit-with-value.pl* 程序能以不同的返回值退出：

```
#!/usr/bin/perl
# exit-with-value.pl

# 以随机值退出
exit time % 256;
```

我们可以用程序 *exit-with-value-call.pl* 调用 *exit-with-value.pl*。我们先用 system 调用该程序，然后把$?右移 8 位得到 exit 的值：

```
#!/usr/bin/perl
# exit-with-value-call.pl

system( "perl exit-with-value.pl" );

my $rc = $? >> 8;

print "exit value was $rc\n";
```

当我们运行程序的时候，可以看到每次会有不同的返回值：

```
$ perl exit-with-value-call.pl
exit value was 102
$ perl exit-with-value-call.pl
exit value was 103
$ perl exit-with-value-call.pl
exit value was 104
```

如果使用的是 die 而不是 exit，Perl 会把 255 作为 exit 的返回值。我们可以使用 END 代码块改变返回值。我们可以在 Perl 结束程序之前把需要的返回值赋给$?。Perl 会在 die 之后立刻进入 END 代码块，$?中保存的是 Perl 本来准备使用的 exit 的返回值。如果看到返回值是 255，我们就知道之前运行的是 die，从而可以把 exit 的值改成某些更有意义的值：

```
END { $? = 37 if $? == 255 }
```

操作系统相关的错误
Errors Specific to the Operating System

在某些系统上，Perl 可能可以通过变量$^E 提供更多关于错误的信息。很多错误往往来自

Perl 之外，所以即使 Perl 用外部的函数库没有检测到问题，操作系统也可能会设置它自己的错误变量。

通常情况下，只要是标准的 Perl，变量$^E 的值就会和$!的一样。对于 Perl 语言可以处理的问题，我们不会在$^E 中得到额外的信息。不过，在 VMS、OS/2、Windows 或 MacPerl 上，我们可能会得到额外的信息。

这并不意味着和平台相关的模块不能用$^E 来把信息传递回去。当这些模块和其他函数库或资源交互的时候，Perl 不一定会处理这些操作的错误。如果一个库函数调用返回了一个表示失败的返回值，Perl 可能不会把它当作错误来特殊对待。但是，调用的模块可能会解析返回值，判断出它是一个错误，然后自行设定好$^E。

模块 Mac::Carbon 会通过$^E 返回来自 Mac 接口的错误信息。我们可以用 Mac::Errors 把错误信息转换成错误代码、符号常量或错误的描述。我有一个叫 Mac::Glue 的程序，它可以运行另一台机器上的 RealPlayer 来操作与之相连的家庭影院系统，该程序通过检查变量$^E 来判断发生了什么问题：

```
#!/usr/bin/perl
# mac-realplayer.pl

use Mac::Errors qw($MacError);
use Mac::Glue;

print "Trying machine $ENV{REALPLAYER_MACHINE}...\n";

my $realplayer = Mac::Glue->new(
        'Realplayer',
        eppc => 'RealPlayer',
        $ENV{REALPLAYER_MACHINE},
        undef, undef,
        map { $ENV{"REALPLAYER_MACHICE_$_"} } qw( USER PASS )
        );

$realplayer->open_clip( with_url => $ARGV[0] );

if( $^E )
        {
        my $number = $^E + 0;
        die "$number: $MacError\n";
        }
```

这个程序的很多地方都有可能出问题。我们须要使用很多环境变量和一个命令行参数。如果我忘记了设置$ENV{REALPLAYER_MACHINE}或忘记了在命令行指定 URL，我们会得到一个错误消息告诉我们有地方出了问题：

```
$ perl mac-realplayer.pl
Trying machine ...
-1715: a required parameter was not accessed
```

如果我忘记了设置$ENV{REALPLAYER_MACHINE_USER}或$ENV{REALPLAYER_MACHINE-PASS}，

Mac OS X 会提示输入用户名和密码以访问远程的机器。如果取消该对话框，会得到一个不同的错误消息，告诉我们没有通过认证：

```
$ perl mac-realplayer.pl
Trying machine realplayer.local...
-128: userCanceledErr
```


对于 Windows 系统，$^E 的值和 Win32::GetLastError() 返回的一样。Win32 系列的模块会使用$^E 返回错误消息。我们可以使用 Win32::FormatMessage() 把错误代码变成一个字符串。比如说，模块 Text::Template::Simple 会尝试调用 Win32 模块得到 Windows 路径，如果失败，会调用 GetLastError：

```
package Text::Template::Simple;

if(IS_WINDOWS) {
 require Win32;
 $wdir = Win32::GetFullPathName($self->{cache_dir});
 if( Win32::GetLastError() ) {
    warn "[  FAIL  ] Win32::GetFullPathName\n" if DEBUG;
    $wdir = ''; # croak "Win32::GetFullPathName: $^E";
 }
 else {
    $wdir = '' unless -e $wdir && -d _;
 }
     }
```

在 VMS 上，如果$!{VMSERR}为真，我们会在$^E 中得到更多的信息。其他的操作系统可能也会使用这个变量。

汇报模块的错误
Reporting Module Errors

到目前为止，我们讨论的都是 Perl 如何告诉我们错误。如果我们想告诉程序员模块出了问题该怎么办呢？我们有很多种方法。我将使用 Andy Wardley 的 Template 工具箱来演示这个问题，因为它里面有我需要的所有例子。其他的模块可能会有自己的汇报错误的方法。

最简单可能也是最让人恼火的方法是设置一个软件包变量并让用户去检查它。我们甚至可以自己设置$!。我的意思是，我们可以这样做，但是这不意味着应该这样做。模块 Template 会在出错的时候设置变量$Template::ERROR：

```
my $tt = Template->new() || carp $Template::ERROR, "\n";
```

使用软件包变量并不太好，程序员应该尽可能避免使用它们。除了软件包变量之外，更好的方式是使用 error 类。这样即使创建对象的时候失败了，Template 对象也会返回错误：

```
my $tt = Template->new() || carp Template->error, "\n";
```

如果已经得到了对象，我们可以使用 error 来找出上次操作对象时发生的错误。error 方法会从 Template::Exception 中返回一个错误对象，供我们检查错误的类型和描述：

```
$tt->process( 'index.html' );
if( my $error = $tt->error )
        {
        croak $error->type . ": " . $error->info;
        }
```

在这种情况下，我们不用自己构造错误消息，as_string 已经为我们完成了这个工作：

```
$tt->process( 'index.html' );
if( my $error = $tt->error )
        {
        croak $error->as_string;
        }
```

我们甚至不用调用 as_string，因为该对象会自动把它转换成字符串：

```
$tt->process( 'index.html' );
if( my $error = $tt->error )
        {
        croak $error;
        }
```

分解问题
Separation of Concerns

Template 模块处理错误的主要思想是不使用函数或方法的返回值汇报错误。返回值做的事情不应该超过函数应该做的事。我们不应该把返回值当作汇报错误的通道。确实如此。我们可以在出问题的时候什么也不返回，即使返回假也会有问题。因为 0、空字符串、空列表可能都是某个子程序运行成功时的返回值。这也是我们在设计系统时须要考虑的问题。

假设有一个名为 foo 的函数须要返回一个字符串，它在出问题的时候什么也不返回。由于没有传给 return 任何值，调用函数也不会得到任何值——不管是在标量上下文中还是列表上下文中（Perl 会把返回值当作 undef 或空列表来处理）：

```
sub foo {
        ...
        return unless $it_worked;
        ...
        return $string;
        }
```

这种方式很容易描述清楚，也便于理解。当然，我不想把返回值弄得一团糟。不过即使这样也会导致代码不断地膨胀，因为即使最简单的函数最后也会被错误处理的代码占据大部

分空间。如果在 foo 中给每种错误都返回一个值，我们会把 foo 中有趣的代码淹没在错误处理中：

```
sub foo {
        ...
        return -1 if $this_error;
        return -2 if $that_error;
        ...
        return $string;
        }
```

与之相反的，我们可以把错误信息保存起来，这样当程序员注意到调用出了问题的时候就可以去读取错误。我们只用给实例或类添加一个保存错误的项。在 Template 的 process 方法中，如果任何地方出了问题，系统的另一部分会处理它并把错误保存起来。process 方法只是简单地返回错误：

```
# From Template.pm
sub process {
        my ($self, $template, $vars, $outstream, @opts) = @_;

        ...

        if (defined $output) {
                ...
                return 1;
        }
        else {
                return $self->error($self->{ SERVICE }->error);
        }
}
```

error 方法其实是在 Template::Base 中定义的，它既负责设置错误消息又负责读取错误消息。这个函数在基类中，因此可以供 Template 家族的所有模块使用。实际上它非常巧妙，既简单又实用：

```
# From Template/Base.pm
sub error {
        my $self = shift;
        my $errvar;

        {
        no strict qw( refs );
        $errvar = ref $self ? \$self->{ _ERROR } : \${"$self\::ERROR"};
        }
        if (@_) {
        $$errvar = ref($_[0]) ? shift : join('', @_);
        return undef;
        }
        else {
        return $$errvar;
        }
}
```

得到第一个参数后，它会设置$errvar。如果$self 是一个引用（比如说是以$tt->error 方式调用的），它所指向的一定是一个实例，所以程序会检查$self->{_ERROR}。如果$self 不是一个引用，它一定是一个类的名字（以 Template->error 方式调用的），所以程序会从软件包变量中得到错误对象的引用。注意，Andy 关掉了对符号引用的检查以便为$self 中的任何类的名字构造完整的软件包名。$self 中的类可能是任何一个 Template 模块。

如果在@_中还有别的参数，就意味着我们正在调用 error 设置错误消息。因此 error 设置好错误消息并返回 undef（注 2）。process 的返回值正好就是 error 方法的返回值。另一方面，如果@_中没有别的参数，就意味着我们正在试图得到错误消息，所以它会返回对 $errvar 取反引用后的结果。这就是我们在程序的$error 变量中得到的结果。

解释完了。虽然我可能不会像 Andy 这样来处理这个问题，但是基本的想法是一样的：把数据放在某个地方并给程序员提供一种方式找到它；返回一个没有定义的值来表示错误。

异常
Exceptions

我们必须把这一点说清楚，Perl 没有异常。不过，和很多其他的 Perl 中没有的功能一样，人们找到了模拟它的方法。如果你对其他的语言（比如 Java 或 Python）很熟悉，请把你的期望降低一点，这样你不会太失望。在其他的一些语言中，异常处理是语言的基本组成部分，也是用于处理所有错误的方式。异常不是 Perl 设计中的组成部分，也不是 Perl 程序员习惯的处理错误的方式。

虽然我对在 Perl 中使用异常没有什么兴趣，但是支持它的人们确实有一个合理的理由：程序员必须得处理错误，不然程序会停止运行。

eval

前面已经说过，我们可以模拟出基本的异常。最简单的方式是使用 die 和 eval 的组合。我们可以用 die 抛出异常（意味着我们必须得自己调用它），然后用 eval 捕获它并进行处理。当 eval 捕获到一个错误的时候，它会停止正在执行的代码块并把错误消息放到$@中。最后，我们在 eval 之后通过检查$@判断是否有错误发生：

```
eval {
        ...;
        open my($fh), ">", $file
                or die "Could not open file! $!";
        };
if( $@)
        {
```

注 2：这往往不是一个好主意，因为在列表上下文中我们会得到(undef)——一个 undef 的列表。

```
    ...; # 捕获 die 的消息并处理它
    }
```

eval 可能会捕获一个很多层之外的 die。处理这种发生在很远的地方的事情可能会非常麻烦，尤其是在无法处理错误的时候，要想从错误中恢复就更困难了。这意味着我们应该尽可能在发生错误的地方附近处理它们。

传递了很多层的 die

Multiple Levels of die

如果把 die 当作异常处理机制使用，我们可以把它的错误消息在多层的 eval 中传递。如果没有给 die 传递一个消息，它就会使用$@的值：

```
#!/usr/bin/perl
# chained-die.pl

eval{
        eval {
                eval {
                        # start here
                        open my($fh), ">", "/etc/passwd" or die "$!";
                        };
                if( $@ )
                        {
                        die; # first catch
                        }
                };
        if( $@ )
                {
                die; # second catch
                }
        };
if( $@ )
        {
        print "I got $@"; # finally
        }
```

当我们得到错误消息的时候，就可以看到消息传播的路径。最原始的消息“Permission denied”来自于第一个 die，后面的每个 die 都会在消息中加上“…propagated”直到最后错误得到了处理：

```
I got Permission denied at chained-die.pl line 8.
        ...propagated at chained-die.pl line 12.
        ...propagated at chained-die.pl line 17.
```

我们可以用这种方式来处理错误：如果某一层无法处理错误，就把错误传到上一层。下面我们修改一下第一个捕获错误的语句，给$@追加一些额外的信息。不过我们还是使用没有参数的 die：

```
#!/usr/bin/perl
# chained-die-more-info.pl

eval{
       eval {
```

```
                my $file = "/etc/passwd";

                eval {
                        # start here
                        open my($fh), ">", $file or die "$!";
                        };
                if( $@ )
                        {
                        my $user = getpwuid( $< );
                        my $mode = ( stat $file )[2];
                        $@ .= sprintf "\t%s mode is %o\n", $file, $mode;
                      $@ .= sprintf( "\t%s is not writable by %s\n", $file, $user )
                                unless -w $file;
                        die; # first catch
                        }
                };
        if( $@)
                {
                die; # second catch
                }
        };
if( $@)
        {
        print "I got $@"; # finally
        }
```

我们得到的输出和之前的基本相同，不过有了更多的额外信息。后面的 die 继续传递了消息“...propageted”。

```
I got Permission denied at chained-die-more-info.pl line 10.
        /etc/passwd mode is 100644
        /etc/passwd is not writable by brian
        ...propagated at chained-die-more-info.pl line 19.
        ...propagated at chained-die-more-info.pl line 24.
```

在 die 中使用引用

die with a Reference

一个有用的异常消息至少应该包含三个东西：错误的类型、错误的来源和发生错误时程序的状态。因为 eval 可能离抛出异常的地方很远，我们需要大量的信息来跟踪错误。一个字符串并不足以提供足够的信息。

我们可以传给 die 一个引用而不是一个字符串。传递的参数是什么类型并没有什么多大关系。如果我们可以在 eval 中捕获 die，引用就会出现在$@中。这意味着我们可以创建一个异常的类，然后在程序中传递异常的对象。当我们检查$@的时候，就可以利用对象的所有特性来传递错误的信息。

在下面的这个小程序中，我们给 die 传递了一个匿名的哈希表。使用 Perl 编译器指令__LINE__和__PACKAGE__来提供当前的行号和软件包名字，并且让__LINE__出现在须要报告错误的地方（使用 die 的地方）。我们的哈希表也包含了错误的类型和一条文本消息。查看$@的时候，可以像使用哈希表一样对它进行反引用操作：

```
#!/usr/bin/perl
# die-with-reference.pl

eval {
        die {   'line'    => __LINE__,
                'package' => __PACKAGE__,
                'type'    => 'Demonstration',
                'message' => 'See, it works!',
                };
        };

if( $@ )
        {
        print "Error type: $@->{type}\n" .
              "\t$@->{message}\n",
              "\tat $@->{package} at line $@->{line}\n";
        }
```

这种方式对于对象也是有效的，因为它们只是被保佑的引用（blessed reference），不过我们须要做一个重要的修改。一旦$@得到对象之后，须要把它保存在另一个变量中，这样才不会丢失该对象。对于一个简单的引用来说这不是问题，因为我们没有做任何可能改变$@的操作。但是对于一个对象来说，我们不确定调用的方法是否会改变它：

```
#!/usr/bin/perl
# die-with-blessed-reference.pl

use Hash::AsObject;
use Data::Dumper;

eval {
        my $error = Hash::AsObject->new(
              {       'line'    => __LINE__ - 1,
                      'package' => __PACKAGE__,
                      'type'    => 'Demonstration',
                      'message' => 'See, it works!',
              } );

        die $error;
        };

if( $@ )
        {
        my $error = $@; # 保存好! $@的值可能之后会被重置

        print "Error type: " . $error->type . "\n" .
              "\t"    . $error->message . "\n",
              "\tat " . $error->package . " at line " . $error->line . "\n";
        }
```

用 die 传递对象

Propagating Objects with die

因为 die 在没有参数的时候会传递$@中的任何内容，所以如果$@中保存的是引用它也会这

样做。下面的这个程序和之前链式的 die 的例子很像，只是我们把信息保存在了一个匿名的哈希表中。这使得之后使用错误消息变得更加方便，因为可以在需要的时候取出需要的部分。当须要修改$@的时候，先对$@进行深拷贝（参见第 14 章）——因为我们的操作可能会重置$@。把$@的拷贝放在$error 中，然后就可以在 die 中使用它了。一旦得到引用之后，就不用解析字符串来得到需要的错误信息了：

```
#!/usr/bin/perl
# chanined-die-reference.pl

eval{
        eval {
                my $file = "/etc/passwd";

                eval {
                        # start here
                        open my($fh), ">", $file or die { errno => $! }
                        };
                if( $@ )
                        {
                        use Storable qw(dclone);
                        my $error = dclone( $@ );
                        @{ $error }{ qw( user file mode time ) } = (
                                scalar getpwuid( $< ),
                                $file,
                                (stat $file)[2],
                                time,
                                );

                        die $error; # first catch
                        }
                };
        if( $@ )
                {
                die; # second catch
                }
        };
if( $@ )
        {
        use Data::Dumper;
        print "I got " . Dumper($@) . "\n"; # finally
        }
```

当引用的内容是一个对象的时候会更好，因为我们可以自己处理错误的传递。如果特殊函数 PROPAGATE 存在的话，就可以用它改变$@并用它的返回值替代$@的当前值。下面我们修改一下前面的程序，使用非常简单的 Local::Error 软件包来处理错误。在 Local::Error 中，为了演示整个过程，我忽略了通常遵守的模块编程的好习惯。在 new 中，我们只是简单地把第一个参数“bless”到软件包中，并返回它。在第一个 die 中，我们把 Local::Error 对象作为 die 的参数。之后的 die 都没有参数，它们会使用$@。因为$@是一个对象，所以 Perl 会调用它的 PROPAGATE 方法，我们在该方法中给$self->{chain}添加了一个新的元素来显示发生错误的文件和行数：

```
#!/usr/bin/perl
# chained-die-propagate.pl
use strict;
use warnings;

package Local::Error;

sub new { bless $_[1], $_[0] }

sub PROPAGATE
        {
        my( $self, $file, $line ) = @_;
        $self->{chain} = [] unless ref $self->{chain};
        push @{ $self->{chain} }, [ $file, $line ];

        $self;
        }

package main;

eval{
        eval {
                my $file = "/etc/passwd";

                eval {
                        # start here
                        unless( open my($fh), ">", $file )
                                {
                                die Local::Error->new( { errno => $! } );
                                }

                        };
                if( $@ )
                        {
                        die; # first catch
                        }

                };
        if( $@ )
                {
                die; # second catch
                }
        else
                {
                print "Here I am!\n";

                }
        };
if( $@ )
        {
        use Data::Dumper;
        print "I got " . Dumper($@) . "\n"; # finally
        }
```

下面我们把输出显示出来，可以看到可以轻松地访问对象中的所有信息：

```
I got $VAR1 = bless( {
                        'chain' => [
                                     [
                                       'chained-die-propagate.pl',
                                       37
                                     ],
                                     [
                                       'chained-die-propagate.pl',
                                       42
                                     ]
                                   ],
                        'errno' => 'Permission denied'
                      }, 'Local::Error' );
```

我们的例子非常简单，不过我们可以非常轻松地修改它以使用一个更有用的对象来表示异常或某种类型的错误。

严重错误

Fatal

模块 Fatal 把 Perl 内置函数在失败时返回的错误转变成了异常。它使用了我们在第 10 章中介绍的封装子程序的技巧。由于错误会被作为异常捕获，我们就不用检查返回值了。不过，我们须要在 Fatal 模块的导入列表中指定须要使用该功能的函数：

```
use Fatal qw(open);

open my($fh), '>', $file;
```

Fatal 模块会让程序在 open 调用失败的时候自动调用 die。产生的错误消息是一个字符串，不过不是太美观：

```
Can't open(GLOB(0x1800664), <, does_not_exist): No such file or directory at (eval 1) line 3
        main::__ANON__('GLOB(0x1800664)', '<', 'does_not_exist') called at
                /Users/brian/Dev/mastering_perl/trunk/Scripts/Errors/Fatals.pl line 5
```

为了捕获错误，我们用 eval 包住 open 以便捕获模块 Fatal 设置的 die 调用：

```
use Fatal qw(open);

eval {
        open my($fh), '>', $file;
        }
if( $@ )
        {
        print "Could not open $file: $@";
        }
```

更方便的一种方式是在导入列表中指定:void。当我们这样做的时候，Fatal 只会在不使用返回值的时候才起作用。在下面的代码片段中，我们在 or 语句中用到了返回值，所以 Fatal 不会起作用：

```
use Fatal qw(:void open);

open my($fh), ">", $file or die "..."; # no exception

eval {

        open my($fh), '>', $file;
        }
if( $@ )
        {
        print "Could not open $file: $@";
        }
```

除了 `system` 和 `exec` 之外，我们可以对 Perl 的任何内置函数使用 `Fatal` 模块。不过我们必须在导入列表中指定须要改变的函数。虽然并不喜欢异常，我也不得不承认这是一个非常方便的模块，可以帮助我们知道哪些 `open` 函数或其他须要检查结果的函数调用没有检查返回值。

总结
Summary

Perl 有很多种汇报错误的方法，我们必须知道哪种方法适合我们所做的事情。此外，Perl 也可以发现来自操作系统函数库和其他模块的错误。

深入阅读
Further Reading

文档 *perlfunc* 中 `die` 和 `eval` 的部分介绍了更多的细节。

Arun Udaya Shankar 在 *Perl.com* 上介绍了 “Object Oriented Exception Handling in Perl”：*http://www.perl.com/pub/a/2002/11/14/exception.html*。他介绍了 `Error` 模块，该模块的异常处理语法是按照 try-catch-finally 方式组织的，它看起来更像 Java 而不是 Perl。

第13章 日志 Logging

日志是一种在程序中记录消息以便观察程序的执行进展或事后分析的技术手段。它不仅仅限于在出问题的时候记录错误和警告，也可以在事情进展顺利的时候记录消息。除了配置文件之外，日志功能是大部分的 Perl 程序所缺乏的一个功能，不过其实它是非常容易添加的。

记录错误和其他信息
Recording Errors and Other Information

Web 应用程序已经具备了日志功能。它们可以把消息发送到标准错误输出（通过程序已有的某种机制或接口）中，消息最终会出现在网页服务器的错误日志中（注 1）。要想给其他的程序添加日志功能须要做更多的工作。一般来说，记录日志并不是简单地打开一个文件、添加一些信息、然后关闭文件，如果程序同时只运行一次，并且一定是在重新运行之前结束，这种方式是可行的。在任何其他情况下，有可能同一程序的两个实例会同时试图写同一个文件。由于输出缓冲和对输出文件的控制权的竞争，只会有一个实例能成功地输出日志，不是所有的内容都能写到文件中。

不过，日志并不只是把一些内容添加到一个文件中。我们可能也希望把消息保存在数据库中、显示在屏幕上、发送到一个系统日志程序中（比如 *syslogd*），或者同时使用多种方式。我们也可能想把它们发送到邮件或寻呼机中。确实如此，如果某台机器上什么地方出了问题，为了确保能看到错误消息，我们会希望把它发送到另一台机器上。我们可能也想同时把消息发送到多个地方，比如同时发送到日志文件中和屏幕上。我们可能还希望程序的不同部分能够用不同的方式记录日志：应用程序的错误应该发给客服人员，数据库错误应该发给数据库管理员。

如果这些还不够复杂，还可以考虑给日志指定不同的级别。给每条消息指定一个重要程度之后，我们可以决定记录什么样的消息。比如说，可能有一个调试的级别，该级别会输出

注 1：如果使用一些定制的日志模块的话，日志甚至可以写到数据库中。

大量的信息，因为我们须要在解决某个问题的时候看到尽可能多的东西。在程序交付使用之后，我们就不想把日志文件塞满调试信息了，不过我们依然希望能看到其他重要的消息。

总结起来，我们须要：

- 输出日志到多个地方
- 程序的不同部分有不同的日志
- 支持不同的日志级别

有很多模块可以提供全部的这些功能，包括 Michael Schilli 的 `Log::Log4perl` 和 Dave Rolsky 的 `Log::Dispatch`，不过在这里我只介绍它们中的一个。它们的基本概念都是相同的，不同之处只是接口的细微差别。实际上，`Log::Log4perl` 中使用了 `Log::Dispatch` 模块。

Log4perl

Apache/Jakarta 项目创建了一个名为 `log4j` 的 Java 日志机制，它具有我们在上一节中提到的所有功能。除了不是 Perl 语言的之外，它是一个非常好的软件包。事实上，它非常棒，因此 Mike Schilli 和 Kevin Goess 把它移植到了 Perl 上。

使用日志其实非常容易。下面这个短小的例程先加载了 `Log::Log4perl` 模块，然后导入`::easy`，这样我们就可以使用`$ERROR` 了：它本身是一个表示日志级别的常量，`ERROR` 则是记录该级别消息的函数。我们告诉 `easy_init` 须要记录的日志消息的级别——在这里是 `$ERROR`，从而完成了对默认日志的设置。之后，我们就可以使用 `ERROR` 了。由于没有指定日志的输出地点，`Log::Log4perl` 会把它发送到终端显示器上：

```
#!/usr/bin/perl
# log4perl-easy1.pl

use Log::Log4perl qw(:easy);

Log::Log4perl->easy_init( $ERROR );

ERROR( "I've got something to say!" );
```

从屏幕上可以看到，错误消息包含日期、时间及发送的消息：

```
2006/10/22 19:26:20 I've got something to say!
```

如果不希望发送到屏幕，也可以给 `easy_init` 传递一些额外的信息来让它知道。可以用一个匿名哈希表来指定日志的级别和输出的文件。我们希望把日志追加到文件中，所以在文件名之前使用了>>，这和 Perl 的 `open` 函数一样。下面这个例子所做的事情和前面的一样，只是把输出保存在了 *error_log* 中。用 `Log::Log4perl` 的术语来说，我们配置了一个**追加器**（appender）：

```
#!/usr/bin/perl
# log4perl-easy2.pl

use Log::Log4perl qw(:easy);

Log::Log4perl->easy_init(
        {
        level => $ERROR,
        file  => ">> error_log",
        }
        );

ERROR( "I've got something to say!" );
```

也可以稍微修改一下程序。可能须要输出一些调试信息，为此可以使用 DEBUG 函数。设置调试消息的目的地时，我们使用了表示标准错误输出的特殊文件名 *STDERR*：

```
#!/usr/bin/perl
# log4perl-easy3.pl

use strict;
use warnings;

use Log::Log4perl qw(:easy);

Log::Log4perl->easy_init(
        {
        file  => ">> error_log",
        level => $ERROR,
        },

        {
        file  => "STDERR",
        level => $DEBUG,
        }
        );

ERROR( "I've got something to say!" );

DEBUG( "Hey! What's going on in there?" );
```

这样，错误消息会输出到文件中，调试消息会出现在标准错误输出中。不过，还是在屏幕上同时看到了它们！

```
2006/10/22 20:02:45 I've got something to say!
2006/10/22 20:02:45 Hey! What's going on in there?
```

我们不必满足于这种简单的错误消息。除了使用字符串作为输出参数之外，还可以给日志程序提供一个匿名的子程序。该子程序只会在记录它对应级别的日志时运行。我们可以在子程序中做任何事情，而返回值会被当作日志消息来处理：

```
#!/usr/bin/perl
# log4perl-runsub.pl
```

```
use strict;
use warnings;

use Log::Log4perl qw(:easy);

Log::Log4perl->easy_init(
        {
        file  => "STDERR",
        level => $DEBUG,
        }
        );

DEBUG( sub {
        print "Here I was!";      # To STDOUT
        return "I was debugging!" # the log message
        } );
```

日志消息是有层次关系的。因此设定错误级别（比如说$DEBUG）意味着该级别的消息和所有低于该级别的消息都会被输出到相应的追加器中。Log::Log4perl 定义了 5 个级别，其中调试（DEBUG）是最高级别（该级别的输出最多）。DEBUG 级别会得到所有级别的错误消息，而错误（ERROR）级别只会得到它自己的和严重错误（FATAL）的消息。下面是各个级别和它们的层次关系：

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

须要记住的是，我们配置的其实是追加器。每一个追加器都能得到日志消息，它们都会检查消息并决定是否须要记录该消息。在上面的例子中，追加器 *error_log* 和 STDERR 都知道它们须要记录 ERROR 级别的消息，所以 ERROR 消息同时出现在了两个地方。只有 STDERR 追加器认为它应该记录 DEBUG 级别的消息，所以 DEBUG 级别的消息只出现在屏幕上。

配置 Log4Perl
Configuring Log4perl

上面使用简单的方法定义了一个默认的记录器（logger）并自动使用了它。要想对日志进行更多的控制，可以直接创建自己的记录器。在大部分应用程序中，我们都须要这样做。这包括两步：首先，加载 Log::Log4perl 并配置它；然后，获取记录器的一个实例。

为了加载自己的配置，我们把之前的 easy_init 替换成了 init。init 需要一个文件名作为参数。初始化记录器后，须要使用 get_logger 得到记录器的一个实例（因为我们可能同

时有多个实例)。easy_init 方法在幕后自动处理了这些事情。现在为了更大的灵活性，我们须要自己做更多的工作，把实例保存在$logger 中，并且调用该实例的方法(比如 error)：

```
#!/usr/bin/perl
# root-logger.pl

use Log::Log4perl;

Log::Log4perl::init( 'root-logger.conf' );

my $logger = Log::Log4perl->get_logger;

$logger->error( "I've got something to say!" );
```

接下来我们必须配置 Log::Log4perl 模块。之前让 easy_init 替我们做了所有的决定，现在须要自己来做决定。在这个例子中，我只用一个记录器。Log::Log4perl 可以在同一个程序中使用不同的记录器，不过暂且忽略这些。其实下面的这个配置文件做的事情和之前的是相同的，都是把 ERROR 级别和低于 ERROR 级别的消息追加到文件 *error_log* 中：

```
# root-logger.conf
log4perl.rootLogger                = ERROR, myFILE

log4perl.appender.myFILE           = Log::Log4perl::Appender::File
log4perl.appender.myFILE.filename= error_log
log4perl.appender.myFILE.mode      = append
log4perl.appender.myFILE.layout    = Log::Log4perl::Layout::PatternLayout
log4perl.appender.myFILE.layout.ConversionPattern = [%c] (%F line %L) %m%n
```

第一行配置了 rootLogger。第一个参数是日志的级别，第二个是使用的追加器。我选择了一个希望使用的名字，*myFILE* 看起来就不错。

配置好日志模块之后，继续配置追加器。虽然已经指定了追加器的名字（*myFILE*）并把它传给了日志模块，追加器并没有被创建——其实什么事情也没有发生。我们须要告诉 Log4perl 模块 *myFile* 方法应该做什么。

首先，我们配置 *myFile* 以使用 Log::Log4perl::Appender::File。也可以使用任何随该模块一起提供的追加器（而且有很多），不过在这里我们尽量保持简单。由于想把消息发送到文件中，我们须要告诉 Log::Log4perl::Appender::file 要使用的文件和模式。和前面的例子 easy 一样，我们想把消息追加到文件 *error_log* 中。我们还须要告诉它要往文件里写的内容。

为了指定消息的格式，我们告诉追加器使用 Log::Log4perl::Layout::PatternLayout。要想使用这个功能，还须要提供格式的字符串。格式的字符串类似于 sprintf 的，不过 Log::Log4perl 可以替我们处理占位符%。我们从文档中选择使用了一些占位符构造了下面的模式：

```
[%p] (%F line %L) %m%n
```

这个模式会输出一条错误消息，包括错误的级别（%p 表示优先级）、记录消息的文件名和行号（%F 和%L）、消息本身（%m）和最后的换行符（%n）：

```
[ERROR] (root-logger.pl line 10) I've got something to say!
```

我们不能忘了使用换行符，因为该模块并没有对消息的格式做任何假设。对于这个问题人们一直有争议。不过我认为该模块的处理方式是正确的：它只做我们所说的事情，而不是让我们适应模块。只用记住要自己添加换行符就行了——不论是在格式字符串中还是在消息中（见表 13-1）。

表 13-1：PatternLayout 方法的占位符

占位符	描述
%c	消息的类别
%C	调用软件包（类）的名字
%d	日期和时间 *YYYY MM DD HH:MM:SS*
%F	文件名
%H	主机名
%l	%c %f（%L）的缩写
%L	行号
%m	消息
%M	方法或函数名
%n	换行符
%p	优先级（比如：ERROR、DEBUG、INFO）
%P	进程的标识号
%r	程序已运行的时间（毫秒）

持久化的日志
Persistent Logging

如果在一个驻留程序中使用日志，比如说是在 mod_perl 上运行的程序，就不用每次都加载配置文件了。我们可以告诉 log::Log4perl 如果已经载入了配置文件就不必重新载入。使用 init_once 方法可以保证只加载一次配置文件：

```
Log::Log4perl::init_once( 'logger.conf' );
```

有些情况下，我们可能会希望 Log::Log4perl 能够反复检查配置文件，并在文件发生变化的时候重新载入。这样，只用修改配置文件（记得我承诺过可以在不改变程序的条件下做到）就可以在程序运行的时候改变记录日志的方式。比如说，我们可能想提高日志的级别以便看到更多的消息（或者把日志发到不同的地方）。方法 init_and_watch 的第二个参数是刷新间隔（单位是秒）：

```
Log::Log4perl::init_and_watch( 'logger.conf', 30 );
```

Log::Log4perl 的其他功能

Other Log::Log4perl Features

目前为止我只介绍了 Log::Log4perl 的基本功能。其实，它的功能比这些要强大得多，而且已经有很多非常棒的教程。因为 Log::Log4perl 最早是作为 Log4j——一个 Java 库出现的，所以它的很多的接口和配置都和 Log4j 一样，你可能也想读一读 Log4j 的文档和教程。

介绍了这些之后，我最后还想再介绍一个功能。因为 Log::Log4per 是用 Perl 写的，可以用 Perl 的钩子函数来动态地改变程序的配置。举例来说，可以使用 Log::Log4perl::Appender::DBI 给数据库发消息，不过通常须要使用用户名和密码才能访问数据库。我不希望把这些信息保存在文件中，因此我们从环境变量中获取它们。在这个追加器的例子中，我从%ENV 中得到需要的消息。Log::Log4perl 看到 sub{}中的配置变量后，会把它当作 Perl 代码来执行（注 2）：

```
# dbi-logger.conf
log4perl.category = WARN, CSV
log4perl.appender.CSV                 = Log::Log4perl::Appender::DBI
log4perl.appender.CSV.datasource      = DBI:CSV:f_dir=.
log4perl.appender.CSV.username        = sub { $ENV{CSV_USERNAME} }
log4perl.appender.CSV.password        = sub { $ENV{CSV_PASSWORD} }
log4perl.appender.CSV.sql             = \
   insert into csvdb                    \
   (pid, level, file, line, message) values (?,?,?,?,?)

log4perl.appender.CSV.params.1        = %P
log4perl.appender.CSV.params.2        = %p
log4perl.appender.CSV.params.3        = %F
log4perl.appender.CSV.params.4        = %L
log4perl.appender.CSV.usePreparedStmt = 1

log4perl.appender.CSV.layout          = Log::Log4perl::Layout::NoopLayout
log4perl.appender.CSV.warp_message    = 0
```

我们的程序还是像之前一样使用日志模块，只是我们须要在 BEGIN 代码块中添加一些初始化代码以便在数据库文件不存在的时候创建该文件：

```
#!/usr/bin/perl
# log4perl-dbi.pl

use Log::Log4perl;

Log::Log4perl::init( 'dbi-logger.conf' );

my $logger = Log::Log4perl->get_logger;

$logger->warn( "I've got something to say!" );

BEGIN {
        # 如果不存在则创建一个数据库
```

注 2：如果由于不信任有权限访问配置文件的人而不喜欢这种灵活性，也可以只用下面这行代码关闭配置文件中的 Perl 钩子函数功能：Log::Log4perl::Config→allow_code(0);

```
        unless( -e 'csvdb' )
                {
                use DBI;

                $dbh = DBI->connect("DBI:CSV:f_dir=.")
                       or die "Cannot connect: " . $DBI::errstr;
                $dbh->do(<<"HERE") or die "Cannot prepare: " . $dbh->errstr();
CREATE TABLE csvdb (
        pid INTEGER,
        level   CHAR(64),
        file    CHAR(64),
        line    INTEGER,
        message CHAR(64)
        )
HERE

                $dbh->disconnect();
          }
        }
```

只要能够写出代码，我们还可以用配置文件的 Perl 钩子函数来做更复杂的事情。

总结
Summary

我们可以使用模块 `Log::Log4perl` 轻松地给程序添加日志。`Log::Log4perl` 是软件包 Log4j 的 Perl 版本。我们可以使用 easy 提供的简单配置，也可以自己指定更复杂的配置。配置好日志之后，我们可以调用子程序生成错误消息并传给 `Log::Log4perl`。`Log::Log4perl` 会决定把消息发到什么地方，或者是否应该忽略它。我们只用负责发送消息。

深入阅读
Further Reading

Sourceforge 的 log4perl 项目（*http://log4perl.sourceforge.net/*）有 `Log4perl` 的常见问题、教程和该软件包的其他配套材料。你可以找到大部分关于使用该模块的基本问题的答案，比如说“如何自动循环日志文件？”

Michael Schilli 在 *Perl.com* 上的“Retire Your Debugger, Log Smartly with Log::Log4perl”一文中介绍了 `Log4perl`（*http://www.perl.com/pub/a/2002/09/11/log4perl.html*）。

Log4Perl 和 Java 的日志库 Log4j（*http://logging.apache.org/log4j/docs/*）有密切的联系，所以你可以用同样的方式使用它们。你可以在这里找到关于 `log4j` 的很好的文档，它们对于 `Log4perl` 也适用。

第 14 章
数据持久化
Data Persistence

我们的程序应该能和其他程序或后来运行的程序本身共享数据。要想实现数据的共享，须要把数据保存在程序内存之外的地方，然后再在使用的时候从该数据源读入数据。我们可以把数据存放在文件或数据库中，通过网络连接发送，或者用任何其他希望的方式处理。

我们甚至可以在不同的程序之间共享数据。除了简单的应用程序之外，我们往往都想使用一个可靠的数据库服务器和 DBI 模块。在这里我们不会讨论诸如 MySQL、PostgreSQL 和 Oracle 之类的数据库服务器。Perl 是通过 DBI 和这些服务器交互的。我在本章结尾的“深入阅读”一节中列出了一本很棒的关于 DBI 的书。本章主要介绍一些轻量级的不需要服务器后台的技术。

扁平结构的文件
Flat Files

无论是从概念上还是实际上来讲，最简单的保存和重用数据的方法是把它们作为文本写到文件中。只要做很少的工作就可以检查文件、在需要的时候修改数据、把文件发送给别人而不用担心字节顺序或内部的数据大小等问题。当想再次使用数据的时候，我们可以把它读回到程序中。即使使用的不是真正的文件，也可以用这些技术把数据通过套接字或邮件发送出去。

打包
pack

内置的 pack 函数可以把数据变成一个字符串，它使用一个模板字符串来决定如何把数据放到一起。这和 sprintf 类似，不过正如它的名字所暗示的，输出的字符串会尽可能地节省空格：

```
#!/usr/bin/perl
# pack.pl

my $packed = pack( 'NCA*', 31415926, 32, 'Perl' );

print 'Packed string has length [' . length( $packed ) . "]\n";
print "Packed string is [$packed]\n";
```

pack 函数生成的字符串比直接把数据的所有字符连在一起都要短，自然得到的结果不可能是便于阅读的：

```
Packed string has length [9]
Packed string is [□Ä¶Ë+ Perl]
```

格式化字符串 NCA*的每个字母都对应着之后的一个参数，它们告诉 pack 该如何解析数据。字母 N 会把参数按网络字节顺序和无符号长整型来处理。字母 C 会把参数按照无符号字符处理。字母 A 会把参数按照 ASCII 字符处理。字母 A 之后的*是**重复的次数**，表示要把参数中的所有字符按字母 A 的方式处理。如果没有*，pack 只会处理第一个字符。

得到包装好的字符串后，我们就可以把它写到文件中，通过套接字发送出去，或者用任何处理字符串的方式来处理它了。当我们希望恢复数据的时候，可以用 unpack 使用同样的模板字符串来恢复数据：

```
my( $long, $char, $ascii ) = unpack( "NCA*", $packed );

print <<"HERE";
Long: $long
Char: $char
ASCII: $ascii
HERE
```

只要所有的操作都是正确的，我们就可以得到之前的数据：

```
Long: 31415926
Char: 32
ASCII: Perl
```

我们可以把多个数据包装在一起来构成扁平的文件数据库的一条记录。假设记录包括一本书的 ISBN 号、书名和作者。我们可以使用 3 个不同的 A 格式，给每个格式指定一个长度的修饰符。对于每个指定的长度，pack 会在参数过长的时候截断参数，也会在太短的时候用空格补齐：

```
my( $isbn, $title, $author ) = (
      '0596527241', 'Mastering Perl', 'brian d foy'
      );

my $record = pack( "A10 A20 A20", $isbn, $title, $author );

print "Record: [$record]\n";
```

无论传给它什么样的数据，记录都会正好是 50 个字符的长度。

```
Record: [0596527241Mastering Perl     brian d foy        ]
```

当我们把它和其他的记录一起保存在文件中的时候，我们就知道接下来的 50 个字节一定是另一个记录。内置的 seek 函数可以把读指针移动到正确的位置，然后我们就可以用 sysread 不多不少地读出需要的字符：

```
open my($fh), "books.dat" or die ...;

seek $fh, 50 * $ARGV[0];        # 移动到记录的正确位置
```

```
sysread $fh, my( $record ), 50;  # read next record.
```

在模板字符串中还可以使用很多其他的格式，包括各种类型的数字格式和存储方式。如果希望看看某个字符串中的内容，可以用 H 格式来解开包裹，把它变成一个十六进制数的字符串。展开$packed 中的字符串时所使用的模板字符串不一定要和创建字符串时的相同：

```
my $hex = unpack( "H*", $packed );
print "Hex is [$hex]\n";
```

现在我们可以看到字符串中每个字符的十六进制形式：

```
Hex is [01df5e76205065726c]
```

内置的 unpack 在读取二进制的文件时也非常方便。下面的这段代码来自于 Gisle Aas 的 Image::Info 软件包，它能够读取 Portable Network Graphics（PNG）的数据。在 while 循环中，它每次读取 8 个字节的数据，然后展开成一个长整数和一个 4 个字符的 ASCII 字符串。长整数和字符串分别是下一块数据的长度和类型。子程序的后半部分将更加频繁地使用 unpack：

```
package Image::Info::PNG;

sub process_file {
        my $signature = my_read($fh, 8);
        die "Bad PNG signature"
        unless $signature eq "\x89PNG\x0d\x0a\x1a\x0a";

        $info->push_info(0, "file_media_type" => "image/png");
        $info->push_info(0, "file_ext" => "png");

        my @chunks;

        while (1) {
               my($len, $type) = unpack("Na4", my_read($fh, 8));

               ...
               }

         ...
         }
```

Data::Dumper

我们可以几乎不费什么力气就把 Perl 的数据结构变成（几乎）可读的文本。和 Perl 一起发行的 Data::Dumper 模块可以把它的参数变成文本形式，并且还可以把文本形式变回原始的数据。下面我们给 Dumper 传递一个要转化成字符串的引用的列表：

```
#!/usr/bin/perl
#  data-dumper.pl

use Data::Dumper qw(Dumper);
```

```
my %hash = qw(
        Fred    Flintstone
        Barney  Rubble
        );

my @array = qw(Fred Barney Betty Wilma);

print Dumper( \%hash, \@array );
```

程序会把数据结构按照 Perl 代码的方式输出：

```
$VAR1 = {
            'Barney' => 'Rubble',
            'Fred' => 'Flintstone'
          };
$VAR2 = [
            'Fred',
            'Barney',
            'Betty',
            'Wilma'
          ];
```

须要记住的是，我们传给它的必须是哈希表或数组的引用。否则 Perl 会给 Dumper 传递一个扁平化后的元素的列表，Dumper 就没法按照数据结构来输出了。如果不喜欢变量的名字，我们也可以自己指定。我们可以给 Data::Dumper->new 传递一个要输出的引用的匿名数组和变量名字的匿名数组：

```
#!/usr/bin/perl
# data-dumper-named.pl

use Data::Dumper qw(Dumper);

my %hash = qw(
        Fred    Flintstone
        Barney  Rubble
        );

my @array = qw(Fred Barney Betty Wilma);

my $dd = Data::Dumper->new(
        [ \%hash, \@array ],
        [ qw(hash array) ]
        );

print $dd->Dump;
```

然后我们就可以调用对象的 Dump 方法得到转化为字符串后的结果。现在，引用使用的名字是我们指定的：

```
$hash = {
          'Barney' => 'Rubble',
          'Fred' => 'Flintstone'
        };
$array = [
          'Fred',
```

```
        'Barney',
        'Betty',
        'Wilma'
       ];
```


不过，转化为字符串后的结构和程序中的数据并不相同。转化之前是一个哈希表和一个数组，现在则变成了它们的引用。如果调用 `Data::Dumper->new` 则在使用的名字之前加上星号：

```
my $dd = Data::Dumper->new(
        [ \%hash, \@array ],
        [ qw(*hash *array) ]
        );
```

转换成字符串之后就不再是引用了：

```
%hash = (
       'Barney' => 'Rubble',
       'Fred' => 'Flintstone'
        );
@array = (
          'Fred',
          'Barney',
          'Betty',
          'Wilma'
         );
```

然后，我们可以把这些变成字符串的数据读回到程序中，或者把它们发送给另一个程序。它们已经是 Perl 代码的形式了，所以可以调用 `eval` 来运行它们。我们把之前的输出保存在了文件 *data-dumped.txt* 中，现在希望把数据载入到程序中。通过使用 `eval`，我们可以在同样的词法作用域内运行它的参数中的代码。在程序中把 `%hash` 和 `@array` 定义成词法变量，但是没有给它们赋值。这些变量会从 `eval` 得到赋值，因此 `strict` 模式不会报告任何错误：

```
#!/usr/bin/perl
# data-dumper-reload.pl
use strict;

my $data = do {
        if( open my $fh, '<', 'data-dumped.txt' ) { local $/; <$fh> }
        else { undef }
        };

my %hash;
my @array;

eval $data;

print "Fred's last name is $hash{Fred}\n";
```

由于我们把变量保存在了文件中，我们也可以使用 `do`。我们在《Intermediate Perl》中稍微做过一些介绍，不过当时是在介绍如何从其他文件中加载子程序。当时的建议是不要这样做，因为 `require` 和 `use` 都比它更好。在这里，要做的事情是重新载入数据，内置的 `do`

要比 eval 有些优势。对于这个任务，须要给 do 传递一个文件名，然后它会搜索@INC 中的目录找到该文件。找到文件之后，它会在%INC 中更新该文件的路径。这和 require 几乎是一样的，不过 do 会每次都重新解析文件，而 require 和 use 只会在第一次时解析文件。它们都会设置%INC，所以当它们看到该文件之后就不用重新解析文件了。和 require 或 use 不一样，do 可能会返回一个假。如果 do 找不到文件，它会返回 undef 并在$!中设置错误消息。如果它找到了文件但却没法读取或解析该文件，它会返回 undef 并设置$@。下面让我们修改一下之前的程序以使用 do：

```
#!/usr/bin/perl
# data-dumper-reload-do.pl
use strict;

use Data::Dumper;

my $file = "data-dumped.txt";
print "Before do, \$INC{$file} is [$INC{$file}]\n";

{
no strict 'vars';

do $file;
print "After do, \$INC{$file} is [$INC{$file}]\n";

print "Fred's last name is $hash{Fred}\n";
}
```

使用 do 的时候，我们失去了 eval 的一个重要功能。由于 eval 是在当前的上下文中执行代码，它可以看到在作用域之内的词法变量。而 do 不能这样做，它不能给词法变量赋值，因为这种做法不是绝对安全的。

如果希望在邮件中传递数据，使用 dumping 方法会非常地方便。有些程序，比如 CGI 程序，可以帮我们收集一些数据以便之后处理。我们可以把数据变成某种格式的字符串，之后再用程序来解析它们。不过使用 Data::Dumper 会更加方便，它也能处理对象。我们可以使用 Business:ISBN 模块来解析一本书的书号，然后用 Data::Dumper 来把对象变成字符串，这样就可以在另一个程序中使用该对象了。最后，我们把导出的数据保存在 *isbn-dumped.txt* 中：

```
#!/usr/bin/perl
# data-dumper-object.pl

use Business::ISBN;
use Data::Dumper;

my $isbn = Business::ISBN->new( '0596102062' );

my $dd = Data::Dumper->new( [ $isbn ], [ qw(isbn) ] );

open my( $fh ), ">", 'isbn-dumped.txt'
        or die "Could not save ISBN: $!";
```

```
print $fh $dd->Dump();
```

当我们把数据读回到程序中的时候，看起来就像数据一直在程序中一样，因为 Data::Dumper 使用了 bless 调用来输出数据：

```
$isbn = bless( {
                'country' => 'English',
                'country_code' => '0',
                'publisher_code' => 596,
                'valid' => 1,
                'checksum' => '2',
                'positions' => [
                                 9,
                                 4,
                                 1
                               ],
               'isbn' => '0596102062',
               'article_code' => '10206'
              }, 'Business::ISBN' );
```

我们不用做任何特殊的事情就能把它变成一个对象，不过还是要加载合适的模块才能调用该对象的方法。可以“bless”软件包中的某个东西并不意味着该软件包存在或里面有任何东西：

```
#!/usr/bin/perl
# data-dumper-object-reload.pl

use Business::ISBN;

my $data = do {
        if( open my $fh, '<', 'isbn-dumped.txt' ) { local $/; <$fh> }
        else { undef }
        };

my $isbn;

eval $data;

print "The ISBN is ", $isbn->as_string, "\n";
```

类似的模块
Similar Modules

Data::Dumper 模块可能并不能在所有的时候都满足我们的要求。在 CPAN 上也有很多其他的模块，它们会用稍微不同的方式做同样的工作。不过基本的概念是一样的：把数据变成文本文件，之后再把文本文件变成数据。我们来转储一个匿名子程序：

```
use Data::Dumper;

my $closure = do {
        my $n = 10;

        sub { return $n++ }
```

```
        };

print Dumper( $closure );
```

不过，我们不会得到任何有用的东西。Data::Dumper 知道它是一个子程序，但是它不可能告诉我们这个子程序的功能是什么：

```
$VAR1 = sub { "DUMMY" };
```

模块 Data::Dump::Streamer 可以在一定程度上处理这些状况，不过它在处理作用域时有些问题。由于它必须把代码引用所引用的变量串行化，它会把这些变量放到和代码引用相同的作用域中：

```
use Data::Dump::Streamer;

my $closure = do {
                my $n = 10;

               sub { return $n++ }
               };


print Dump( $closure );
```

使用 Data::Dumper::Streamer，我们可以得到词法变量和匿名子程序的代码：

```
my ($n);
$n = 10;
$CODE1 = sub {
                  return $n++;
                };
```

由于 Data::Dump::Streamer 会把在同一作用域中的所有代码引用都串行化，所以代码引用中使用的所有变量都会出现在相同的作用域中。有一些方法能够绕过这个问题，不过它们不是始终有效的。请小心使用。

如果不喜欢 Data::Dumper 必须创建的那些变量，我们可以使用 Data::Dump，它可以简化创建数据的过程：

```
#!/usr/bin/perl
use Business::ISBN;
use Data::Dump qw(dump);

my $isbn = Business::ISBN->new( '0596102062' );

print dump( $isbn );
```

输出的结果和 Data::Dumper 的几乎一样，而且没有$VARn 之类的东西：

```
bless({
  article_code => 10_206,
  checksum => 2,
  country => "English",
  country_code => 0,
```

```
    isbn => "0596102062",
    positions => [9, 4, 1],
    publisher_code => 596,
    valid => 1,
}, "Business::ISBN")
```


当我们“eval”这些数据的时候，不会创建任何变量。我们必须把 eval 的结果保存起来才能使用。得到对象的唯一方法是把 eval 的结果赋值给$isbn：

```
#!/usr/bin/perl
# data-dump-reload.pl

use Business::ISBN;

my $data = do {
        if( open my $fh, '<', 'data-dump.txt' ) { local $/; <$fh> }
        else { undef }
        };

my $isbn = eval $data;

print "The ISBN is ", $isbn->as_string, "\n";
```

在 CPAN 上还有很多其他的模块可以转储数据，如果不喜欢上面的这些格式，我们也有很多其他的选择。

YAML

YAML（YAML Ain't Markup Language 的缩写）和 Data::Dumper 的思路相同，不过它更加简洁和易读。YAML 在 Perl 社区中正变得越来越流行，已经在模块的一些维护工作中得到了应用。很多创建模块发布版的工具生成的 *Meta.yml* 文件就是 YAML。比较偶然的，JaveScript Object Notation（JSON）也是一个合法的 YAML 格式。下面我们把数据写到一个后缀名为*.yml* 的文件中：

```
#!/usr/bin/perl
# yaml-dump.pl

use Business::ISBN;
use YAML qw(Dump);

my %hash = qw(
        Fred    Flintstone
        Barney  Rubble
        );

my @array = qw(Fred Barney Betty Wilma);

my $isbn = Business::ISBN->new( '0596102062' );

open my($fh), ">", 'dump.yml' or die "Could not write to file: $!\n";
print $fh Dump( \%hash, \@array, $isbn );
```

数据结构的输出非常紧凑，不过一旦理解它的格式之后还是非常易读的。我们不必经过 Data::Dumper 那样麻烦的步骤就可以取回数据：

```
---
Barney: Rubble
Fred: Flintstone
---
- Fred
- Barney
- Betty
- Wilma
--- !perl/Business::ISBN
article_code: 10206
checksum: 2
country: English
country_code: 0
isbn: 0596102062
positions:
  - 9
  - 4
  - 1
publisher_code: 596
valid: 1
```

模块 YAML 提供了 Load 函数来帮助我们载入数据，不过基本的概念都是一样的。我们从文件中读入数据，并把文本传给 Load：

```
#!/usr/bin/perl
# yaml-load.pl

use Business::ISBN;
use YAML;

my $data = do {
        if( open my $fh, '<', 'dump.yml' ) { local $/; <$fh> }
        else { undef }
        };

my( $hash, $array, $isbn ) = Load( $data );

print "The ISBN is ", $isbn->as_string, "\n";
```

YAML的唯一不足是它不在Perl标准发行版中，而且它还依赖于一些非核心的模块。随着 YAML 越来越流行，这个问题可能会得到解决。有些人也提供了一些更简单的 YAML 的实现，包括 Adam Kennedy 的 YAML::Tiny 和 Audrey Tang 的 YAML::Syck。

Storable

随 Perl 5.7 及其之后版本一起发布的模块 Storable，要比上一节中介绍的可供人阅读的方案更进一步。它的输出是可以让人读懂的，但一般来说它不是为人们的阅读而设计的。该

模块大部分是用 C 写的，其中的一部分暴露出了我们编译 perl 时所使用的系统的架构——数据的字节顺序依赖于系统架构。在大字节序（big-endian）的机器上，比如说我的 G4 Powerbook，得到的结果和在小字节序（little-endian）的 MacBook 上的结果就不同。我们稍后会讨论如何绕开这个问题。

函数 store 可以把数据串行化并保存在文件中。Storable 会把遇到的问题当作异常处理（意味着出问题的时候它会调用 die，而不是试图恢复），所以我们把该函数的调用放在了 eval 中，并检查 eval 的错误变量$@来判断是否有严重问题。一些次要的问题，比如输出错误，会导致返回 undef 而不是调用 die。所以这类错误也须要检查。当错误和系统有关时（比如无法打开输出），我们会在$!中找到错误消息：

```
#!/usr/bin/perl
# storable-store.pl

use Business::ISBN;
use Storable qw(store);

my $isbn = Business::ISBN->new( '0596102062' );

my $result = eval { store( $isbn, 'isbn-stored.dat' ) };

if( $@ )
        { warn "Serious error from Storable: $@" }
elsif( not defined $result )
        { warn "I/O error from Storable: $!" }
```

当须要重新载入数据的时候，我们可以使用 retrieve。和使用 store 类似，把调用放在 eval 中来捕获所有的错误。我们还在 if 结构中添加了一个检查来确保得到的内容是我们所希望的。在这里得到的是一个 Business::ISBN 对象：

```
#!/usr/bin/perl
# storable-retreive.pl

use Business::ISBN;
use Storable qw(retrieve);

my $isbn = eval { retrieve( 'isbn-stored.dat' ) };

if( $@ )
        { warn "Serious error from Storable: $@" }
elsif( not defined $isbn )
        { warn "I/O error from Storable: $!" }
elsif( not eval { $isbn->isa( 'Business::ISBN' ) } )
        { warn "Didn't get back Business::ISBN object\n" }

print "I loaded the ISBN ", $isbn->as_string, "\n";
```

为了绕过刚才提到的和机器相关的格式问题，我们可以在 Storable 中使用**网络字节顺序**。它与架构无关，可以被正确地转化为本地的字节顺序。Storable 提供了一个名字几乎相同但是有一个字母“n”的前缀的函数来完成该工作。就是说，要想把数据按照网络字节顺序

保存，我们须要使用nstore。函数retrieve可以自己找出使用的格式，所以没有nretrieve函数。在这个例子中，我们还使用了Storable的函数来直接操作文件句柄，而不是文件名。这些函数的名字中都有fd：

```
my $result = eval { nstore( $isbn, 'isbn-stored.dat' ) };

open my $fh, ">", $file or die "Could not open $file: $!";
my $result = eval{ nstore_fd $isbn, $fh };

my $result = eval{ nstore_fd $isbn, \*STDOUT  };
my $result = eval{ nstore_fd $isbn, \*SOCKET  };

$isbn = eval { fd_retrieve(\*SOCKET) };
```

可以看到，我们在Storable的函数中使用了文件句柄的引用作为参数。为此须要说明，Storable改变的是那些文件句柄所对应的数据，而不是文件句柄本身。我不能用这些函数来保存某个文件句柄或套接字的状态以供之后使用。这种方式不会工作，不管曾经有多少人在邮件列表中提出过这个问题。

冻结数据
Freezing Data

模块Storable是和Perl一起发行的。它还能把数据“冻结”到一个标量中。我们不必把数据保存在文件中或发送给一个文件句柄，我们可以把它保存在内存中——虽然是串行化处理之后的。我们还可以把它保存在数据库中或对它做一些其他的处理。之后我们可以使用thaw把它变回成原来的数据结构：

```
#!/usr/bin/perl
# storable-thaw.pl

use Business::ISBN;
use Data::Dumper;
use Storable qw(nfreeze thaw);

my $isbn = Business::ISBN->new( '0596102062' );

my $frozen = eval { nfreeze( $isbn ) };

if( $@ ) { warn "Serious error from Storable: $@" }

my $other_isbn = thaw( $frozen );

print "The ISBN is ", $other_isbn->as_string, "\n";
```

这个方法有个有趣的应用。当我们把数据串行化后，它就完全和原来保存数据的变量断开了联系。所有的数据都在串行化的时候被复制保存了起来。当我们“解冻”数据的时候，它就变成了一个全新的数据结构，和之前的数据结构没有任何关系。

在展示这个之前，我先演示一个**浅拷贝**（shallow copy）。在浅拷贝中我们只拷贝数据结构顶层的内容，其余层次使用的都是同样的引用。这是拷贝数据时常常发生的一个错误：我们以为它们是不同的拷贝，之后才发现修改拷贝的内容也会改变原来的内容。

我们先从一个包含两个匿名数组的匿名数组开始。我希望检查一下第二个匿名数组的第二个元素，该元素起初是 Y。我们在修改拷贝的内容前后分别检查一下原始数据和拷贝的值。我们通过反引用$AoA 进行一个浅拷贝，并在一个新的匿名数组中使用它的元素。这是一个非常幼稚的拷贝数据的方法。不过我已经多次看到这样的例子了，我自己也这样做过很多次：

```
#!/usr/bin/perl
# shallow-copy.pl

my $AoA = [
        [ qw( a b ) ],
        [ qw( X Y ) ],
        ];

# 做一个浅拷贝
my $shallow_copy = [ @$AoA ];

# 在改变之前检查一下所有的内容
show_arrays( $AoA, $shallow_copy );

# 改变 shallow_copy 的内容
$shallow_copy->[1][1] = "Foo";

# 在改变之后检查一下所有的内容
show_arrays( $AoA, $shallow_copy );

print "\nOriginal: $AoA->[1]\nCopy: $shallow_copy->[1]\n";

sub show_arrays {
        foreach my $ref ( @_ ) {
                print "Element [1,1] is $AoA->[1][1]\n";
                }
        }
```

运行程序的时候，可以从输出中看到修改$shallow_copy 也改变了$AoA。把两个数组中的相应元素的引用转换成字符串打印出来后，可以看到它们其实是对同一份数据的引用：

```
Element [1,1] is Y
Element [1,1] is Y
Element [1,1] is Foo
Element [1,1] is Foo

Original: ARRAY(0x18006c4)
Copy: ARRAY(0x18006c4)
```

为了避免浅拷贝的带来问题，可以通过冻结数据再解冻的方法进行**深拷贝**（deep copy）。这样，我们不就用做任何工作来找出数据的结构。当数据被冻结之后，它们和源数据就没有任何关系了。我们使用 nfreeze 来得到网络字节顺序的结果，从而确保把数据发给另一台机器时也不会有问题：

```
use Storable qw(nfreeze thaw);

my $deep_copy = thaw( nfreeze( $isbn ) );
```

这种方法非常有用，因此 Storable 提供了 dclone 函数来一次完成该功能：

```
use Storable qw(dclone);

my $deep_copy = dclone $isbn;
```

其实，Storable 比我在本节中介绍的要有意思和有用得多。它能够处理文件锁，并且有钩子函数，能够和类集成在一起，从而使我们可以在对象中使用它的功能。更多的细节请参考 Storable 的文档。

Matthew Simon Cavalletto 的 Clone::Any 模块也提供了同样的功能，它是多个可以进行深拷贝的模块的一个门面（façade）。使用 Clone::Any 的统一接口，可以不必关心真正使用的是哪个模块，也不用担心模块是否安装在一台远程的计算机上（只要它们中有一个是的）：

```
use Clone::Any qw(clone);

my $deep_copy = clone( $isbn );
```

DBM 文件
DBM Files

介绍完 Storable 之后我们来介绍一下微型轻量级数据库。这些数据库不需要数据库服务器，但是依然能够处理大部分的工作，确保程序可以访问到数据。这样的数据库有很多，不过这里只会涉及它们中的一部分。基本的概念都是相同的，只是接口和具体的细节不同。

dbmopen

至少从 Perl 3 开始，我们就可以连接 DBM 文件了。DBM 文件是存储在磁盘上的哈希表。在 Perl 的早期，该语言和主要的应用都是以 Unix 为中心的，很多系统的数据库使用的都是这种格式，所以访问 DBM 非常重要。DBM 是一个简单的哈希表，我们可以指定键和值。我们使用 dbmopen 把一个哈希表和一个磁盘文件联系起来，然后就可以像正常的哈希表一样使用它了。dbmclose 会保证所有的修改都被保存到磁盘上：

```
#!/usr/bin/perl
# dbmopen.pl

dbmopen %HASH, "dbm-open", 0644;

$HASH{'0596102062'} = 'Intermediate Perl';

while( my( $key, $value ) = each %HASH ) {
        print "$key: $value\n";
        }

dbmclose %HASH;
```

在现在的 Perl 中，情况变得更加复杂了。DBM 格式分支成了几种相互竞争的格式，每一种都有自己的优势和特点：有些只能保存短于某个长度的值，或者只能保存一定数目的键，如此等等。

根据本地的 perl 二进制文件编译时使用的不同选项，我们可能会使用某种实现。这意味着虽然在同一台机器上可以安全地使用 dbmopen，在机器间共享数据时仍可能会遇到问题，因为别的机器可能使用的是不同的 DBM 库。

不过，这些都不是问题，因为 CPAN 提供了一些更好的选择。

DBM::Deep

现在更流行的是 DBM::Deep 模块。如果在以前，我可能会使用其他的 DBM 模块，不过现在我大部分情况下都是使用它。利用这个模块，我们可以创建任何深度的多层哈希表或数组。该模块完全是用 Perl 写的，所以不用担心不同库的实现、底层的细节等问题。只要有 Perl，就有所需要的东西。我们不用担心使用的平台是 Mac、Widnows 还是 Unix，它们之间可以相互共享 DBM::Deep 文件。它最大的优点是它是纯 Perl 的。

Joe Huckaby 创建 DBM::Deep 模块的同时也提供了一个面向对象的接口和一个 tie 接口（参见第 17 章）。该模块的文档推荐使用面向对象的接口，所以在这里我们一直使用该接口。只要有一个作为文件名的参数，它的构造函数就可以打开文件，并在文件不存在的时候创建该文件：

```
use DBM::Deep;

my $isbns = DBM::Deep->new( "isbns.db" );
if( $isbns->error ) {
        warn "Could not create database: " . $isbns->error . "\n";
        }

$isbns->{'0596102062'} = 'Intermediate Perl';
```

得到 DBM::Deep 对象之后，我们可以像普通哈希表的引用一样来处理它，并且可以使用任何哈希操作。

此外，也可以调用该对象的方法来做同样的事情。甚至可以使用额外的功能，比如说在创建对象的时候设置使用文件锁和自动清除缓冲区：

```
#!/usr/bin/perl

use DBM::Deep;

my $isbns = DBM::Deep->new(
        file     => "isbn.db"
        locking  => 1,
        autoflush => 1,
        );
```

```
if( $isbns->error ) {
        warn "Could not create database: " . $isbns->error . "\n";
        }

$isbns->put( '0596102062', 'Intermediate Perl' );

my $value = $isbns->get( '0596102062' );
```

该模块也可以处理基于数组的对象，它们有自己的一组方法。它们也有嵌入内部的钩子函数，可以用这些钩子函数来改变它们的工作方式。

当你读到本书的时候，DBM::Deep 应该已经支持交易（transaction）了。多亏该模块现在的维护者 Rob Kinyon 的工作。我们可以创建对象，然后使用 begin_work 的方法来开始一个交易。一旦调用该方法后，在调用 commit 之前数据不会有任何变化。commit 会把所有的变化写到磁盘上。如果中间出了问题，我们可以调用 rollback 来把数据恢复到开始时的状态：

```
my $db = DBM::Deep->new( 'file.db' );

eval {
        $db->begin_work;

        ...

        die "Something didn't work" if $error;

        $db->commit;
        }

if( $@ )
        {
        $db->rollback;
        }
```

总结
Summary

把数据变成字符串是一种在程序内和程序之间传递数据的轻便方法。更复杂一些的方法是使用二进制格式，Perl 也带有处理二进制格式的模块。不管哪一种方法，我们都有多种选择而不必使用数据库服务器。

深入阅读
Further Reading

O'Reilly 出版的 Simon Cozens 著的《Advanced Perl Programming, Second Edition》在第 4 章"Objects, Databases, and Applications"中介绍了对象存储和对象数据库。Simon 介绍了两个流行的对象存储，Pixie 和 Tangram，你可能会发现它们很有用。

O'Reilly 出版的 Larry Wall、Tom Christiansen 和 Jon Orwant 合著的《Programming Perl, Third Edition》讨论了各种 DBM 文件的实现和每种方案的优缺点。

O'Reilly 出版的 Tim Bunce 和 Alligator Descartes 合著的《Programming the Perl DBI》介绍了 Perl 数据库接口 (DBI)。DBI 是个适用于大部分流行的数据库服务器的通用接口。如果你需要的功能超出了本章介绍的范围，很可能你要使用 DBI。我本可以在本章中介绍 SQLite 的，它是一个非常轻量级的单文件的关系数据库。不过它和其他数据库一样须要通过 DBI 访问，所以我没有介绍它。不过如果想迅速完成一些数据持久化的操作，它是非常方便的。

模块 `BerkeleyDB` 给 BerkeleyDB 库（*http://sleepycat2.inetu.net/products/bdb.html*）提供了一个接口，BerkeleyDB 库提供了另一种保存数据的方法。它使用起来要复杂一些，不过功能非常强大。

Alberto Simoes 为 2006 年冬季的《The Perl Review 3.1》写了“Data::Dumper and Data::Dump::Streamer”一文。

Vladi Belperchinov-Shabanski 在 Perl.com 的“Implementing Flood Control”一文中演示了一个使用 `Storable` 的例子：*http://www.perl.com/pub/a/2004/11/11/floodcontrol.html*。

Randal Schwartz 有一些关于数据持久化的文章：“Persistent Data”，Unix Review, February 1999（*http://www.stonehenge.com/merlyn/UnixReview/col24.html*）；“Persistent Storage for Data”, Linux Magazine, May 2003（*http://www.stonehenge.com/merlyn/LinuxMag/col48.html*）和“Lightweight Persistent Data”, Unix Review, July 2004（*http://www. stonehenge.com/merlyn/UnixReview/col53.html*）。

第15章 使用 Pod

Working with Pod

Perl 有一种被称为 Plain Old Documentation 的默认的文档格式，简称是 Pod。我们可以在程序中甚至是在两段代码之间使用它。其他的程序可以轻松地从代码中抽取出 Pod 并把它转换成人们熟悉的格式，比如 HTML、纯文本甚至是 PDF。在本章中，我会介绍 Pod 最有用的一些功能，介绍如何测试 Pod，以及如何创建自己的转换 Pod 的程序。

Pod 格式

The Pod Format

本章中介绍的大部分内容都是由 Sean Burke 负责维护的。他在文档 *perlpodspec* 中完整地规定了 Pod 格式。本章介绍的是该规范的精华部分，以及如何解析该格式。《Learning Perl》和《Intermediate Perl》中介绍的只是概述文档 *perlpod* 中的基础部分。

Pod 指令

Directives

Pod 指令可以从任何一个新语句所在行的开头开始，每条指令都以等号“=”开始，而且只能出现在 Perl 期望有一个新的语句时（所以不可能出现在一个语句的中间）。当 Perl 试图解析一条新的语句而看到=的时候，它会切换到解析 Pod 的状态。Perl 会继续解析 Pod 直至遇到指令=cut 或文件结束：

```
#!/usr/bin/perl

=head1 First Level Heading

Here's a line of code that won't execute:

        print "How'd you see this?\n";

=over 4

=item First item
```

```
=item Second item

=back

=cut

print "This line executes\n";
```

正文元素
Body Elements

在 Pod 的文本中，可以用 *interior sequences* 指定须要被当作特殊字体或字符显示的无结构的标记。每个标记都以一个字符开始，该字符指定了序列的类型，并把内容放在尖括号中。举例来说，在 Pod 中我们使用“<”来表示一个文本的<。若需要斜体字（如果格式化程序支持斜体字），我们可以使用 I<>：

```
=head1

Alberto Simões helped review I<Mastering Perl>.

In HTML, I would write <i>Mastering Perl</i> to
get italics.

=cut
```

多行注释
Multiline Comments

由于 Perl 可以处理代码中间的 Pod，所以我们可以用它来注释掉多行代码。我们只须要用 Pod 指令把它们包围起来，确保中间没有其他的=cut 就可以了：

```
=pod

....
....
....

=cut
```

转换 Pod
Translating Pod

有两种方法可以把 Pod 转换成其他的格式：使用现成的转换程序或写自己的转换程序。我们可以修改某个已有的程序来尝试这两种方法。如果还想在基本的 Pod 格式之外添加一些其他的格式，我们就须要写一些程序来解析它。

幸运的是，Sean Burke 已经在 Pod::Parser 中完成了大部分的工作。只要遵循基本的规则，就可以解析常用的 Pod 格式；同时我们也能通过派生 Pod::Parser 的子类来解析扩展的格式。

Pod 转换程序

Pod Translators

Perl 有一些自带的 Pod 转换程序。你可能已经使用过，虽然你不知道。其实，命令 perldoc 就是一个从文档中提取出 Pod 并对其进行格式化处理的工具。通常它会根据你的终端设置来对文档进行格式化，可能会使用颜色或字体之类的功能：

```
$ perldoc Some::Module
```

不过，这不是 perldoc 可以做的全部事情。因为它是为输出到终端窗口而进行格式化的，所以当我们把输出重定向到文件的时候，格式上会出一些问题。比如说，标题会看起来很奇怪：

```
$ perldoc CGI > cgi.txt
$ more cgi.txt
CGI(3)      User Contributed Perl Documentation      CGI(3)

NNAAMMEE

           CGI - Simple Common Gateway Interface Class
```

我们可以使用-t 开关告诉 perldoc 输出纯文本格式，而不要为屏幕显示而格式化内容：

```
% perldoc -t CGI > cgi.txt
% more cgi.txt

NAME

       CGI - Simple Common Gateway Interface Class
```

再退一步，perldoc 也可以不格式化任何东西。开关-m 可以简单地输出源文件（当我们需要看看源代码但是又不想自己去找源文件的时候，这个功能非常方便）。perldoc 会在@INC 中查找源文件。perldoc 能够做所有的这些事情，因为它只是其他 Pod 转换程序的一个接口。perldoc 程序非常简单，它只是 Pod::Perldoc 一个简单的封装。使用 perldoc 查看它自己的代码就可以看出来：

```
$ perldoc -m perldoc
#!/usr/bin/perl
        eval 'exec /usr/local/bin/perl -S $0 ${1+"$@"}'
                if 0;

# 文件 "perldoc" 是由 "perldoc.PL" 生成的

require 5;
BEGIN { $^W = 1 if $ENV{'PERLDOCDEBUG'} }
use Pod::Perldoc;
exit( Pod::Perldoc->run() );
```

模块 Pod::Perldoc 做的工作只是解析命令行参数并把任务分发给相应的子类（比如子类 Pod::Perldoc::ToText）来处理。除此之外还有什么别的模块呢？可以使用-l 开关查找这些转换程序的目录：

```
$ perldoc -l Pod::Perldoc::ToText
/usr/local/lib/perl5/5.8.4/Pod/Perldoc/ToText.pm
```

```
$ ls /usr/local/lib/perl5/5.8.4/Pod/Perldoc
BaseTo.pm       ToChecker.pm   ToNroff.pm      ToRtf.pm        ToTk.pm
GetOptsOO.pm    ToMan.pm       ToPod.pm        ToText.pm       ToXml.pm
```

想把所有这些放在一行 Perl 语句中吗？

```
$ perldoc -l Pod::Perldoc::ToText | perl -MFile::Basename=dirname \
        -e 'print dirname( <> )' | xargs ls
```

在 Unix 机器上，使用该平台已有的 *dirname* 工具（不过它不是 Perl 程序）可以让这个命令变得更短：

```
$ perldoc -l Pod::Perldoc::ToText | xargs dirname  | xargs ls
```

如果没有 *dirname* 工具，也可以使用下面这个功能相同的简单的 Perl 程序。它和 Perl Power Tools（注 1）中的 `dirname` 程序非常相似。我经常在移动 Perl 函数库的目录时使用它：

```
#!/usr/bin/perl
use File::Basename qw(dirname);
print dirname( $ARGV[0] );
```

不难看出，可以把 Pod 转换成 nroff（发送到终端的内容）、纯文本、RTF、XML 及一大堆其他的格式。格式转换很快就可以完成。

`perldoc` 并没有为每个格式都提供一个开关。用户可以使用`-o` 开关指定一种格式。这里我想使用 XML 格式，所以我们使用`-oxml`，并加上了`-T` 开关。`-T` 会让 `perldoc` 把所有的内容输出到标准输出。我们也可以使用`-d` 把输出发送到文件中：

```
$ perldoc -T -oxml CGI
```

不过，我们不必局限于这些格式化程序。我们也可以创建自己的程序。比如说，我们可以用`-M` 开关指定使用 `Pod::Perldoc::ToRtf` 模块：

```
$ perldoc -MPod::Perldoc::ToRtf CGI
```

Pod::Perldoc::ToToc

现在我们已经具备了创建自己的格式化程序的条件。在这个例子中，我希望从 Pod 得到一个表格的内容。可以扔掉所有其他的东西，但是须要得到`=head` 指令中的文本，而且希望该文本能够按照大纲的模式缩进。参照已有的转换程序的命名规律，我把这个模块命名为 `Pod::Perldoc::ToToc`。我甚至把它上传到了 CPAN 上。事实上我使用了这个模块来帮助我写作本书。

注 1：你可以在这个地址找到 Perl Power Tools：*http://sourceforge.net/projects/ppt/*。

开始写一个自己的转换程序是非常容易的。除了那些须要特殊处理的地方外，还可以参考其他的转换程序并照葫芦画瓢。这非常容易，因为大部分复杂的工作都是在其他地方完成的：

```
package Pod::Perldoc::ToToc;
use strict;

use base qw(Pod::Perldoc::BaseTo);

use subs qw();
use vars qw();

use Pod::TOC;

$VERSION = '0.10_01';

sub is_pageable       { 1 }
sub write_with_binmode { 0 }
sub output_extension   { 'toc' }

sub parse_from_file
        {
        my( $self, $file, $output_fh ) = @_; # Pod::Perldoc object

        my $parser = Pod::TOC->new();

        $parser->output_fh( $output_fh );

        $parser->parse_file( $file );
        }
```

我们的转换程序派生自 Pod::Perldoc::BaseTo 模块。该模块几乎可以处理所有重要的事情。它把我们在 parse_from_file 中做的事情和 perldoc 的用户接口连接了起来。当 perldoc 试图加载该模块时，它会检查 parse_from_file 是否存在，因为它会在找到要解析的文件之后调用该子程序。如果该子程序不存在，perldoc 会继续查找列表中的下一个格式化程序。之前我们用过的开关-M 并不会告诉 perldoc 使用哪个格式化程序；它只会把指定的程序加入到尝试使用格式化程序的列表的前面。

在 parse_from_file 中，第一个参数是一个 Pod::Perldoc 对象。我们没有使用它。从 Pod::TOC 模块创建了一个新的解析器模块，下一节会介绍 Pod::TOC 模块。该模块派生自 Pod::Simple，它的大部分接口都来自 Pod::Simple。

第二个参数是我们要解析的文件的文件名。第三个是用于输出的文件句柄。创建解析器之后，我们使用$parser->output_fh()设置输出的目标。模块 Pod::Perldoc::BaseTo 会期望从该文件句柄得到输出。我们不能直接把结果打印到 STDOUT，因为这样会绕过 Pod::Perldoc 的输出机制，从而导致该模块抱怨没有得到任何输出结果。在这里，我们又

一次利用了 Pod::Perldoc 的内部机制。如果用户希望把输出保存到文件中，他须要把 $output_fh 指向该文件。当我们设置好它后，只用调用$parser->parse_file()，一切就会正常运行起来了。

Pod::Simple

其实不必在 TOC 的创建程序中解析 Pod，我们可以在底层使用 Pod::Simple 来完成解析工作。该模块提供了简单的接口允许我们在发生某些事件的时候做一些事情。所有把 Pod 抽取出来并确定各个部分代表的内容的工作都是在别的地方做的，我们不必自己处理。下面是 Pod::TOC 模块中从一个 Pod 文件中提取表格内容的所有代码：

```
package Pod::TOC;
use strict;

use base qw( Pod::Simple );

$VERSION = '0.10_01';

sub _handle_element
        {
        my( $self, $element, $args ) = @_;

        my $caller_sub = ( caller(1) )[3];
        return unless $caller_sub =~ s/.*_(start|end)$/${1}_$element/;

        my $sub = $self->can( $caller_sub );

        $sub->( $self, $args ) if $sub;
        }

sub _handle_element_start
        {
        my $self = shift;
        $self->_handle_element( @_ );
        }

sub _handle_element_end
        {
        my $self = shift;
        $self->_handle_element( @_ );
        }

sub _handle_text
        {
        my $self = shift;

        return unless $self->get_flag;

        print { $self->output_fh }
              "\t" x ( $self->_get_flag - 1 ), $_[1], "\n";
        }
```

```
{ #定义一个作用域以隐藏仅供子程序使用的词法变量
my @Head_levels = 0 .. 4;

my %flags = map { ( "head$_", $_ ) } @Head_levels;

foreach my $directive ( keys %flags )
        {
        no strict 'refs';
        foreach my $prepend ( qw( start end ) )
                {
                my $name = "${prepend}_$directive";
                *{$name} = sub { $_[0]->_set_flag( $name ) };
                }
        }

sub _is_valid_tag { exists $flags{ $_[1] } }
sub _get_tag      {       $flags{ $_[1] } }
}

{
my $Flag;

sub _get_flag { $Flag }

sub _set_flag
        {
        my( $self, $caller ) = shift;

        my $on  = $caller =~ m/^start_/ ? 1 : 0;
        my $off = $caller =~ m/^end_/   ? 1 : 0;

        unless( $on or $off ) { return };

        my( $tag ) = $caller =~ m/_(.*)/g;

        return unless $self->_is_valid_tag( $tag );

        $Flag = do {
                   if( $on ) { $self->_get_tag( $tag ) } # 如果$on 为真就设置标记位
                elsif( $off ) { undef }           # 如果$off 为真就清除标记位
                };

        }
}
```

Pod::TOC 模块派生自 Pod::Simple。大部分的操作都是在 Pod::Simple 解析该模块的时候进行的。Pod::Perldoc::ToToc 中没有子程序 parse_file，因为 Pod::Simple 已经定义了，我们不用做任何不同的事情。

不过，我们须要改变 Pod::Simple 模块遇到各种 Pod 元素时的行为。Allison Randal 写了 Pod::Simple::Subclassing 模块来演示派生该模块的各种方法，这里我们只使用最简单

的方法。当 Pod::Simple 遇到一个 Pod 元素的时候，它会调用一个名为_handle_element_start 的子程序，并把该元素的名字作为参数传给它；当它处理完该元素之后，它会用同样的参数调用_handle_element_end。当遇到元素中的文本时，它会调用_handle_text。Pod::Simple 会把所有的文本拼接起来，所以我们可以把它当成一个逻辑块（比如一个段落段）来处理，而不用作为多个单元（比如一行文本再加上多行内容）来分别处理。

_handle_element_start 和_handle_element_end 只是_handle_element 的一个封装。我们通过查看调用者来找出是_handle_element_start 还是_handle_element_end。在_handle_element 中，我们从$caller_sub 中得到调用的子程序，并得到 start 或 end。我们把它和$element 中的元素名拼接在一起。这样最后在$caller_sub 中得到了 start_head1 和 end_head3 之类的东西。下面，我还须要展示更多的代码来说明如何处理这些子程序。

当我们接收到某个开始事件或结束事件的时候，我们并没有得到元素中的文本，所以须要记住正在处理的元素，这样_handle_text 才知道该如何处理。每次当 Pod::Simple 遇到文本的时候——无论是指令=headN、正文中的段落还是列表中的某个东西，它都会调用_handle_text。对于我们的表格来说，我只希望输出来自=head 指令的文本，所以我们在_handle_text 中有一些特殊处理。

在 foreach 循环中，我们遍历了各个级别的=head 指令（注 2）。在外层的循环中，我们为每一个级别的=head 指令创建了两个子程序：start_head0、end_head0、start_head1、end_head1，如此等等。我们使用了符号引用（参见第 8 章）来动态地创建子程序的名字，然后再把一个匿名的子程序赋给该名字的 typeglob（参见第 9 章）。

所有的这些子程序都会设置一个标志位。当 start_headN 子程序开始运行的时候，它会打开该标志位；当 end_headN 子程序运行的时候，它会关掉该标志位。这些操作都是在_set_flag 中完成的，它会操作$Flag。

我们的_handle_text 子程序会根据$Flag 决定做什么。_handle_text 在$Flag 为真的时候输出文本，为假的时候直接返回。我们使用它来屏蔽掉不属于标题的文本。此外，为了确定当前表格内容的缩进大小，我们把=head 的级别保存在了$Flag 中。

整个执行过程是：当我们遇到=head1 时，Pod::Simple 调用_handle_element_start。该子程序会立刻调用_handle_element。_handle_element 会知道当前正处于=head1 的开始部分。所以，_handle_element 会知道它须要调用 start_head1，并动态地创建它。子程序 start_head1 会调用_set_flag('start_head1')，它会根据调用的参数来打开$Flag。接下来，Pod::Simple 遇到了一些文本，所以会调用_handle_text，_handle_text 调用_get_flag 得到返回值为真。它会用文件句柄把文本打印出来。之后，Pod::Simple 处理

注 2：我使用的值是 0 到 4，因为 O'Reilly 在使用的格式——也就是我写作本书时使用的格式中添加了=head0。

完了=head1，会调用_handle_element_end，_handle_element_end 会调用_handle_element，_handle_element 会调用 end_head1。当 end_head1 运行的时候，它会调用_set_flag 关掉$Flag。每次 Pod::Simple 遇到指令=head 的时候都会按这个过程执行。

从 Pod::Simple 派生子类

Subclassing Pod::Simple

我用 Pod 格式写作了本书，不过使用的是 O'Reilly Media 为了满足它的出版需求而扩展的一种格式。比如说，O'Reilly 为脚注（注 3）添加了指令 N。Pod::Parser 依然可以处理它，不过它须要知道在遇到它的时候该如何处理。

Allison Randal 创建了 Pod::PseudoPod 作为 Pod::Simple 的扩展。它可以处理 O'Reilly 添加的这些元素。该模块是一个更长的派生类的例子。我从该模块派生了 Pod::PseudoPod::MyHTML，并使用该模块为本书的网站生成 HTML。你也可以从该网站得到源代码（注 4）。

网页服务器中的 Pod

Pod in Your Web Server

为了从 Apache 网页服务器请求 Perl 的文档，并用喜欢的浏览器阅读文档，Andy Lester 写了 Apache::Pod 模块（基于 Rich Bowen 的 Apache::Perldoc 模块）。与分页输出到终端相比，我当然更喜欢这种方式。我们可以享受到浏览器带来的所有优势，包括显示风格设置、搜索功能和超链接功能。

Sean Burke 的 Pod::Webserver 模块使用了自己的网页服务器来把 Pod 文档转换成网页。它使用了 Pod::Simple 来完成转换工作，它可以运行在任何 Perl 能够运行的地方。使用该模块，即使不想安装 Apache 服务器，我们也可以有自己的文档服务器。

测试 Pod

Testing Pod

完成 Pod 之后，我们可以再检查一下以确保它能够正常工作。当用户阅读我们的文档的时候，他们不应该看到任何有关格式问题的警告信息。即使 Pod 的错误造成解析器出问题也不应该导致用户无法阅读文档。如果用户无法读到文档，那么文档还有什么用处呢？

检查 Pod

Checking Pod

Pod::Checker 是另一种类型的 Pod 转换程序，与那些把 Pod 文本转换成其他格式的程序不同，它检查的是 Pod 和文本。它会在找到可疑的内容时发出警告。Perl 有一个 podchecker 程序，它是一个类似于 perl -c 的成熟可用的程序，不过它是用来检查 Pod 的。其实该程序只是把模块 Pod::Checker 变成了程序。模块 Pod::Checker 是 Pod::Parser 的一个子类：

注 3：你可能会注意到我们很喜欢在《Learning Perl》和《Intermedia Perl》中使用脚注。

注 4：《Mastering Perl》的网站是 *http://www.pair.com/comdog/mastering_perl*。

```
% podchecker Module.pm
```

程序 podchecker 比较适合于手工使用。如果希望在 shell 脚本中使用它，可以直接使用模块 Pod::Simple 来检查错误。Pod::Simple 会在解析输入内容的时候记录遇到的所有错误。我们可以之后再查看它们：

```
*** WARNING: preceding non-item paragraph(s) at line 47 in file test.pod
*** WARNING: No argument for =item at line 153 in file test.pod
*** WARNING: previous =item has no contents at line 255 in file test.pod
*** ERROR: =over on line 23 without closing =back (at head2) at line 255 in file test.pod
*** ERROR: empty =head2 at line 283 in file test.pod
Module.pm has 2 pod syntax errors.
```

很久以前，我希望自动处理我的所有模块，所以我写了模块 Test::Pod。后来 Andy Lester 几乎重写了该模块，他现在维护着该模块。我们可以把 *t/pod.t* 文件放到测试目录下：

```
use Test::More;
eval "use Test::Pod 1.00";
plan skip_all => "Test::Pod 1.00 required for testing POD" if $@;
all_pod_files_ok();
```

Pod 的覆盖率
Pod Coverage

检查完文档的格式之后，我们也希望确保文档覆盖了所有的内容。模块 Pod::Coverage 可以找到一个软件包中的所有函数，并把它们和找到的 Pod 关联起来。除了特殊的函数名字、以下划线开头的函数名和私有方法之外，它会汇报所有没有文档的函数。

最简单的调用方法是从命令行调用。比如说，我们同时使用-M 开关加载 CGI 模块和 Pod::Coverage 模块，并且给后者加上=CGI 来告诉它须要检查的模块。最后，由于不希望运行任何程序，我们使用了-e 1 来指定一个空程序：

```
% perl -MCGI -MPod::Coverage=CGI -e 1
```

输出会给 CGI 模块一个评分，并列出所有没有说明文档的函数：

```
CGI has a Pod::Coverage rating of 0.04
The following are uncovered: add_parameter, all_parameters, binmode, can,
cgi_error, compile, element_id, element_tab, end_form, endform, expand_tags,
init, initialize_globals, new, param, parse_params, print, put, r,
save_request, self_or_CGI, self_or_default, to_filehandle, upload_hook
```

我们也可以自己写一个程序（podcoverage）来遍历命令行中指定的软件包。该程序给出的评分来自于 coverage 方法，评分可能是一个 0 到 1 之间的数，或者是 undef（如果无法分析该模块的话）：

```
#!/usr/bin/perl

use Pod::Coverage;

foreach my $package ( @ARGV )
        {
        my $checker = Pod::Coverage->new(
               package => $package
               );

        my $rating = $checker->coverage;

        if( $rating == 1 )
               {
               print "$package gets a perfect score!\n\n";
               }
        elsif( defined $rating )
               {
               print "$package gets a rating of ", $checker->coverage, "\n",
                      "Uncovered functions:\n\t",
                      join( "\n\t", sort $checker->uncovered ),
                      "\n\n";
               }
        else
               {
               print "$package can't be rated: ", $checker->why_unrated, "\n";
               }

        }
```

使用该程序测试 Module::NotThere 和 HTML::Parser 的时候，它会告诉我们没法给第一个模块打分，因为它没有找到任何 Pod。而在 HTML::Parser 中，它找到了很多没有文档说明的函数：

```
$ podcoverage Module::NotThere HTML::Parser
Module::NotThere can't be rated: couldn't find pod
HTML::Parser gets a rating of 0.925925925925926
Uncovered functions:
        init
        netscape_buggy_comment
```

不过，我们的 podcoverage 程序并没有多大的用处。它可以帮助我们找到模块中隐藏的函数，但是我们并不会使用这些函数，因为它们可能在之后的版本中就不存在了。我们可以使用 podcoverage 来检查自己的模块以确保所有的函数都有说明文档，不过这样做会很麻烦。

幸运的是，Andy Lester 使用 Test::Pod::Coverage 把整个过程变成了自动的。该模块基于 Pod::Checker 模块。只用创建一个测试文件并放到模块发布包的 *t* 目录下，每次运行 make test 的时候就会自动测试 Pod 的覆盖率。下面我把这段代码从文档中摘了出来。它首先检查 Test::Pod::Coverage 是否存在，这样那些没有安装该模块的系统就不会进行该检查，这个行为和 Test::Pod 模块一样：

```
use Test::More;
eval "use Test::Pod::Coverage 1.00";
plan skip_all => "Test::Pod::Coverage 1.00 required for testing POD coverage" if $@;
all_pod_coverage_ok();
```

隐藏和忽略函数

Hiding and Ignoring Functions

之前曾经提到过可以把函数隐藏起来以避免检查 Pod。Perl 并没有提供任何方法来区分应该说明的函数、其他人可以使用的公有函数和不希望用户使用的私有函数。Pod 覆盖率测试程序会看到所有的函数。

不过，这不是全部的情况。`Pod::Coverage` 模块知道哪些函数须要忽略。比如说，所有特殊的 `Tie::`函数（参见第 17 章）其实都是私有函数。根据约定，所有以下划线开头的函数（比如_init）都是仅供内部使用的私有函数，因此 `Pod::Checker` 会忽略它们。如果希望创建私有函数，可以在它们的名字前加上下划线。

但是，并不是所有的函数都能够隐藏。考虑一下之前提到的 `Pod::Perldoc::ToToc` 子类。我们必须覆盖 `parse_from_file` 方法，这样才能调用自己的解析器。我们其实不希望给这个函数提供文档，因为除了使用了一个不同的格式化模块之外，它做的事情都和父类中的方法相同。大部分时候，用户都不会直接调用它，它做的事情其实和父类 `Pod::Simple` 的 `parse_from_file` 方法的文档中所描述的是一样的。我们可以告诉 `Pod::Checker` 忽略某些名字或忽略满足某个正则表达式的名字：

```
my $checker = Pod::Coverage->new(
         package => $package,
         private     => [ qr/^_/ ],
         also_private => [ qw(init import DESTROY AUTOLOAD) ],
         trustme     => [ qr/^get_/ ],
         );
```

`private` 键需要一个正则表达式的列表。它是给真正的私有函数使用的。`also_private` 是那些知道名字的私有函数的列表，使用它可以不用正则表达式。`trustme` 键稍微有些不同。我们使用它告诉 `Pod::Checker` 我们没有也不会给这些公共函数提供文档。我们的例子使用了正则表达式 `qr/^get_/`。有的时候我们会一次介绍一系列的函数，而不是分别介绍每一个。有些函数也有可能是由 AUTOLOAD 生成的。模块 `Test::Pod::Coverage` 也使用了同样的接口来忽略不用检查的函数。

总结

Summary

Pod 是 Perl 的标准文档格式，我们可以使用 Perl 自带的工具轻松地把它转换成其他格式。如果这些工具不能满足需求，我们可以创建自己的 Pod 转换程序来支持新的格式，或者是向已有的格式添加一些新的功能。

使用 Pod 为自己的软件创建文档之后，也可以使用一些工具来检查文档的格式、确保文档覆盖了所有的内容。

深入阅读
Further Reading

文档 *perldoc* 介绍了 Pod 格式的基本内容。文档 *perlpodspec* 介绍了实现的细节。

Allison Randal 的 `Pod::Simple::Subclassing` 演示了一些其他的从 `Pod::Simple` 派生子类的方式。

`Pod::Webserver` 作为 Hack #3 出现在了 O'Reilly 出版的由 chromatic、Damian Conway 和 Curtis "Ovid" Poe 合著的《Perl Hacks》一书中。

我在 2005 年 12 月的《The Perl Journal》的"Playing with Pod"一文中介绍了如何从 `Pod::Simple` 派生子类输出 HTML：*http://www.ddj.com/dept/lightlang/184416231*。

我在 2002 年 12 月的《The Perl Journal》的"Better Documentation Through Testing"一文中介绍了 `Test::Pod`。

第 16 章
位操作
Working with Bits

Perl 是一种高级语言，因此我们不必处理每个比特和每个字节就能完成工作。不过，为此所付出的代价是要让 Perl 来管理所有内容的存储。如果我们须要控制存储的方式该怎么办呢？遇到那些把很多信息都保存在一个字节里的场合（比如 Unix 文件权限）该如何处理呢？如果一个有成千上万个元素的数组占用了太多的内存空间又该怎么办呢？解决这些问题的方法是重新回到比特的级别来操作。

二进制数
Binary Numbers

几乎所有的人使用的都是二进制的计算机，所以再说“二进制”似乎成了多余的了。系统的最底层处理的是关或开，也被称为 0 或 1。把足够多的 1 和 0 连在一起就构成了计算机指令，它们告诉计算机进行某些操作，或者改变磁盘上数据的物理表示。大部分人不用在这个层次和计算机打交道，不过由于有的时候须要处理底层的问题，这种考虑问题的方式会渗透到高级语言编程中。

比如说，考虑一下我们在 `mkdir`、`chmod` 或 `dbmopen` 中使用的设置文件模式（也被称作权限，其实不只是权限）的参数。虽然该模式是用一个八进制数表示的，但是它的含义其实是由每一个比特位决定的：

```
mkdir $dir, 0755;
chmod 0644, @files;
dbmopen %HASH, $db_file, 0644;
```

我们也可以从 `stat` 的返回值中得到文件模式：

```
my $file_mode = (stat( $file ))[2];
```

在 Unix 和类 Unix 系统上，文件模式包含了很多的信息，其中包括：所有者、组用户和其他人的文件访问权限及 setuid、setgid 和其他的设置。得到文件模式之后，我们须要把这些设置提取出来；Perl 定义了完成这些操作所需要的所有操作符，稍后会介绍它们。

以二进制形式输出
Writing In Binary

有些情况下我们希望用一串比特表示一系列的值。通过给每个比特（或一组比特）指定含义，我们可以在一个标量中保存多个值，从而只占用一个标量的内存空间。由于计算机处理位操作很快，所以对比特串的操作不会太慢，不过程序中使用比特串的代码可能会让整个程序慢下来。在第 17 章中，我们会使用比特串保存 DNA 序列。在程序使用的内存有了显著下降的同时，程序的速度明显变慢了。也就是说，我们总得牺牲一个优势来换取另一个。

从 Perl 5.6 开始，我们可以用记号 `0b` 直接指定二进制的数。《Learning Perl》的第 2 章曾经对此有所涉及：

```
my $value = 0b1;      # same as 1, 0x01,  01
my $value = 0b10;     #         2, 0x02,  02
my $value = 0b1000;   #         8, 0x08, 010
```

我们可以加上下划线来让二进制数更加易读；Perl 会直接忽略它们。1 个字节（byte）是 8 个比特，1 个 nybble（注 1）是半个字节：

```
my $value = 0b1010_0101              # by nybbles;
my $value = 0b11110000_00001111      # by bytes
my $value = 0b1111_0000___0000_1111  # by bytes and nybbles
```

目前（将来也不太可能），Perl 并没有内置的像 `oct` 或 `hex` 那样进行进制转换的 `bin`。不过，我也可以写一个自己的 `bin`——之前我确实也写了一个。后来 Randal 告诉我内置的 `oct` 可以处理二进制、八进制和十六进制之间的转换：

```
my $number = oct( "0b110" ); # 6
```

当然，一旦我们把一个数赋给变量之后，Perl 只会把它当成一个数处理，它并没有固有的表示形式。虽然 Perl 在默认情况下是以十进制的形式显示数的，我们可以在 `printf` 或 `sprintf` 中使用 `%b` 格式符来以二进制输出一个数。在下面的这个例子中，我们在值的前面加上了文本串 `0b` 来提醒自己输出的是二进制数。虽然只有 `1` 和 `0` 是二进制数的一个特征，但其他进制的数也可能会使用它们：

```
#!/usr/bin/perl

my $value = 0b0011;

printf "The value is 0b%b\n", $value;
```

注 1：他们故意把“bite（咬）”和“nibble（轻咬）”都拼错了，很可爱吧？

在上面的例子中，我们自己在值的前面加上了 0b。可以使用一个不同的 sprintf 的格式化串来自动得到它。如果在占位符%后面使用哈希符号#，Perl 会在数的前面加上前缀来表示数的基数（注 2）：

```
my $number_string = printf '%#b', 12;   # prints "0b1100"
```

我们还可以在格式字符串中指定宽度以得到更整洁漂亮的输出。为了让二进制数一直有 32 位，我们把宽度放在了格式符之前，并且在宽度前加上 0，这样 Perl 会用 0 填充空着的位置。由于字符串 0b 占据了两个位置，所以总共的列宽宽度是 34：

```
printf "The value is 0b%034b\n", $value;

printf "The value is %#034b\n", $value;
```

因为经常须要在各种进制的数之间转换，我经常使用这样的代码，所以我使用了一些单行的 Perl 程序。我给这些单行的 Perl 程序设置了别名，这样就可以在 bash shell（你的 shell 可能会有所不同）中使用它。d2h 可以把十进制数转换成十六进制数，o2b 可以把八进制数转换成二进制数，如此等等。当你阅读本章的时候，你可能会发现这些小的脚本非常方便：

```
# 适用于 bash。可能你的 shell 会有所不同。

alias d2h="perl -e 'printf qq|%X\n|, int( shift )'"
alias d2o="perl -e 'printf qq|%o\n|, int( shift )'"
alias d2b="perl -e 'printf qq|%b\n|, int( shift )'"

alias h2d="perl -e 'printf qq|%d\n|, hex( shift )'"
alias h2o="perl -e 'printf qq|%o\n|, hex( shift )'"
alias h2b="perl -e 'printf qq|%b\n|, hex( shift )'"

alias o2h="perl -e 'printf qq|%X\n|, oct( shift )'"
alias o2d="perl -e 'printf qq|%d\n|, oct( shift )'"
alias o2b="perl -e 'printf qq|%b\n|, oct( shift )'"
```

位操作
Bit Operators

Perl 的二进制操作符和 C 中的二进制操作符做的事情相同。大部分情况下，它们和你的 Perl 编译时所用的 C 函数库的行为一样。每当所要操作二进制数或查找一些资料的时候，我都会去查看我的 C 语言的书（注 3），不过我这样做的主要原因是我最先是在 C 语言中学习二进制操作的。

注 2：这种方法对其他的基数也有效，使用%#x 可以得到 0x，使用%#o 可以得到 0。不过，如果数是 0，基数是什么都没有关系，所以 Perl 不会给出任何前缀。

注 3：主要是 Waite Group 的《New C Primer Plus》。该出版社也出了一本 C++的书，我把它称为《New C Plus Plus Primer Plus》。在 Amazon 上可以以不到一美元的价钱买到它的旧书——至少上次我查看的时候还是这样。

一元操作符 NOT，~

Unary NOT, ~

一元操作符 NOT（有时也被称作取补操作），~，会根据平台上整数的长度返回每一位的反，或者说是每一位相对于 1 的补（注 4）。这意味着不管数字的符号是什么，Perl 只是简单地翻转所有的位：

```
my $value      = 0b1111_1111;
my $complement = ~ $value;
printf "Complement of\n\t%b\nis\n\t%b\n", $value, $complement;
```

可以看到，即使传入的数是 8 比特的，返回的值也是 32 比特的（因为我的 MacBook 的整数是 32 比特的）：

```
Complement of
       11111111
is
       11111111111111111111111100000000
```

这个输出不是太好。我们希望数字能够很好地对齐。为此，我们须要得到整数的长度。这个可以非常容易地从 Perl 的配置中得到（参见第 11 章）。整数的长度是按字节表示的，所以我们把从 Perl 的配置中得到的值乘以 8：

```
#!/usr/bin/perl
# complement.pl

use Config;
my $int_size   = $Config{intsize} * 8;

print "Int size is $int_size\n";

my $value      = 0b1111_1111;
my $complement = ~ $value;
printf "Complement of\n\t%${int_size}b\nis\n\t%${int_size}b\n",
        $value, $complement;
```

现在数字对齐得很好，不过如果能够把空余的位用 0 补齐就更好了。你可以自己试试如何实现这种效果：

```
Int size is 32
Complement of
                                11111111
is
       11111111111111111111111100000000
```

使用从一元操作符 NOT 得到的结果时须要倍加小心。根据使用方式的不同，我们会得到不同的数。在下面的例子中，我们把逐位取反后的结果保存在`$negated`中。当我们用 `printf` 打印`$negated`的时候，可以看到所有的位都被翻转了，而且负数的绝对值比正数大 1。这

注 4：这是 Perl 中极少会暴露出系统架构的地方。它取决于你处理器的整数长度。

是补码的特点，这里我不会深入介绍它。不过，当我们用老式的无格式的 print 打印数字的时候，Perl 会把它当成无符号的数字处理，所以说比特位的翻转不会影响正数的符号，但是会把负数变成正数：

```
#!/usr/bin/perl
# unary-not.pl

foreach my $value ( 255, 128, 5, 65534  )
        {
        my $negated = ~ $value;

        printf "  value is %#034b  %d\n",   $value,   $value;

        printf "~ value is %#034b  %d\n", $negated, $negated;

        print  "  value is ", $negated, "\n\n";
        }
```

得到的输出结果对于那些不了解究竟发生了什么事的人来说会令人非常困惑（这意味着如果希望以后维护的程序员知道是怎么回事的话我们就不应该使用这段代码）：

```
  value is 0b00000000000000000000000011111111  255
~ value is 0b11111111111111111111111100000000  -256
  value is 4294967040

  value is 0b00000000000000000000000010000000  128
~ value is 0b11111111111111111111111101111111  -129
  value is 4294967167

  value is 0b00000000000000000000000000000101  5
~ value is 0b11111111111111111111111111111010  -6
  value is 4294967290

  value is 0b00000000000000000111111111111110  65534
~ value is 0b11111111111111111000000000000001  -65535
value is 4294901761
```

操作符~的优先级几乎是最高的，所以它会在其他大部分操作符之前得到执行。须要注意这一点。

位与操作，&

Bitwise AND, &

如果不需要前一个例子中的一些比特位的时候该怎么办呢？我们必须使用Perl的整数长度，但可以使用位掩码来去掉额外的位，这样就须要介绍下一个操作符：位与操作，&。

位与操作符会返回第一和第二个参数中都设置了的位。如果两个参数中有一个参数的某一位是 0，结果中的相应位置就会是 0。只有当两个参数的相同位置都是 1 的时候，结果中的相应位置才是 1。通常情况下第二个参数被称为**掩码**，因为它的 0 会把第一个参数中设置的位掩盖起来：

```
  1010      value
& 1101      mask
------
  1000
```

可以使用这种方式选择感兴趣的位。上一节使用~计算一个 8 比特的值的补，结果得到了一个 32 比特的值。如果只需要最后的 8 比特，我们可以使用&和一个只设置了低字节的数相与：

```
my $eight_bits_only = $complement & 0b1111_1111;
```

也可以使用十六进制表示法来让它更加易读。数字 0xFF 表示一个设置了所有位的字节，所以可以使用它作为掩码来隐藏除了低字节之外的所有东西：

```
my $eight_bits_only = $complement & 0xFF;
```

这种方式在我们须要从一个数中提取出需要的比特位时也很有用。比如说，从 stat 中得到的 Unix 文件模式是两个字节，包括所有者、组用户和其他用户的访问权限。每一组权限需要一个 nybble，最高位置的 nybble 包括一些其他信息。要想得到访问权限，我们须要知道（并且使用）正确的位掩码。在这个例子中，我们用八进制数指定掩码，它和我们在 chmod 和 mkdir 中使用的是一样的（无论是在 Perl 还是在命令行中）：

```
my $mode = ( stat($file) )[2];

my $is_group_readable     = $mode & 040;
my $is_group_writable     = $mode & 020;
my $is_group_executable   = $mode & 010;
```

其实，我并不喜欢使用这些魔法数（magic number）作为位掩码。我们可以把它们变成常量（参见第 11 章）：

```
use constant GROUP_READABLE   => 040;
use constant GROUP_WRITABLE   => 020;
use constant GROUP_EXECUTABLE => 010;

my $mode = ( stat($file) )[2];

my $is_group_readable     = $mode & GROUP_READABLE;
my $is_group_writable     = $mode & GROUP_WRITABLE;
my $is_group_executable   = $mode & GROUP_EXECUTABLE;
```

不过，我们并不用这样做。众所周知，在 POSIX 模块中已经有了这样的常量。使用 fcntl_h 导出标签可以得到与文件访问权限有关的 POSIX 常量。你能够通过名字判断出各个常量的功能吗？

```
#!/usr/bin/perl
# posix-mode-constants.pl

use POSIX qw(:fcntl_h);

# S_IRGRP S_IROTH S_IRUSR
# S_IWGRP S_IWOTH S_IWUSR
```

```
# S_IXGRP S_IXOTH S_IXUSR
# S_IRWXG S_IRWXO S_IRWXU
# S_ISGID S_ISUID

my $mode = ( stat( $ARGV[0] ) )[2];

print "Group readable\n"      if $mode & S_IRGRP;
print "Group writable\n"      if $mode & S_IWGRP;
print "Group executable\n"    if $mode & S_IXGRP;
```


位或操作，|

Binary OR, |

位或操作符|会返回在任何一个（或两个）操作数中设置的位。只要某一位在任何一个参数中设置了，结果中该位也会被设置：

```
  1010
| 1110
------
  1110
```

我经常使用它来合并数字，你可能已经在 sysopen 和 flock 的操作符中使用过。这些内置函数需要一个参数指定禁止使用（或允许使用）的操作，我们可以把所有的值相或得到该参数。每一个值都表示一个设置，得到的结果则是所有这些设置的组合。

sysopen 的第三个参数是它的模式。如果知道了模式设置的每一位，我们就可以直接使用它。不过它们随着系统的不同而有所变化。因此我们改而使用从 Fcntl 得到的结果。在第 3 章中，我们使用过它来限制打开文件后可以做的事情：

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

use Fcntl (:DEFAULT);

my( $file ) = $ARGV[0] =~ m/([A-Z0-9_.-]+)/gi;

sysopen( my( $fh ), $file, O_APPEND | O_CREAT )
        or die "Could not open file: $!\n";
```

在锁定文件的时候，我们把各个设置相或得到需要的效果。模块 Fcntl 提供了作为常数的各个数值。这个例子用读写模式打开一个文件后立刻尝试得到该文件的锁。我们同时使用了排他锁 LOCK_EX 和非阻塞锁 LOCK_NB，所以如果无法立刻得到文件锁程序就会调用 die。通过把这些常数相或，我们就得到了供 flock 使用的正确的位模式：

```
use Fcntl qw(:flock);

open my($fh), '<+', $file       or die "Connot open: $!";
flock( $fh, LOCK_EX | LOCK_NB ) or die "Cannot lock: $!";

...;

close $fh; # don't unlock, just close!
```

如果不使用 LOCK_NB，我们的程序会停在 flock 所在的行等待得到文件锁。在这个例子中我们简单地退出了程序，不过有的时候我们可能希望休眠一段时间再重试，或者做一些别的事情直到得到锁为止。

异或操作，^

Exclusive OR, ^

逐位的异或操作符，^，会返回恰好只在一个操作数中设置了的位。这正是排他（exclusive）的含义（译注 1）。如果任何一个参数的某个比特位被设置了，结果中该比特位也会被设置，不过必须要求另一个参数中的相同位没有被设置。这就是说，只有在一个参数中设置了的位才会被设置：

```
  1010
^ 1110
------
  0100
```

位异或操作也可以处理字符串，不过估计除了想参加 Obfuscated Perl Contest 的人之外没有人希望这样做。下面我介绍一个有趣的例子，它适合在比赛的时候使用，其余的部分留给读者去分析。完整的例子在 *perlop* 中也有介绍。

想知道“perl”和“Perl”的区别吗？

```
$ perl -e 'printf "[%s]\n", ("perl" ^ "Perl")'
[ ]
```

呵呵，没法看到结果，下面我们使用 ord 把它变成 ASCII 值：

```
$ perl -e 'printf "[%d]\n", ord("perl" ^ "Perl")'
[32]
```

这是一个空格字符！操作符^会屏蔽掉两个字符串中都设置了的比特位，只有第一个字符是不同的。两个字符的差别只有一个比特。

为了看看是哪个比特造成的，我们使用%b 对内置的 ord 函数返回的结果进行格式化：

```
$ perl -e 'printf "[%#10b]\n", ord("perl" ^ "Perl")'
[0b00100000]
```

我们是怎么得到这个数的呢？首先，我们先得到大写字母 *P* 和小写字母 *p* 的值：

```
$ perl -e 'printf "[%#10b]\n", ord( shift )' P
[0b01010000]
$ perl -e 'printf "[%#10b]\n", ord( shift )' p
[0b01110000]
```

对它们进行异或操作后，我们会得到只在一个字符中设置了的比特位。在 ASCII 字符表中，小写字母和大写字母除了第 5 位之外完全相同。所有的小写字母都在相应的大写字母前第 32 个位置上：

译注 1：异或的英文是 Exclusive OR，直译是排他的或。

```
  0101_0000
^ 0111_0000
-----------
  0010_0000
```


这正是 *perlfaq1* 认为“perl”和“Perl”只有一个比特的差别的原因（注 5）。

左移<<和右移>>操作

Left << and right >> shift operators

移位操作符会把所有的位向左（使用<<）或向右（使用>>）移动，并且用 0 填补空出来的位置。箭头指向的方向是移动的方向，最高位（代表最大的值的位）在左边：

```
my $high_bit_set = 1 << 8;        # 0b1000_0000

my $second_byte  = 0xFF << 8;     # 0x00_00_FF_00
```

移位操作符不会把移出来的值填充到另一端，不过我们可以自己写子程序实现该功能。为此，须要先记住移出的部分，再把它们添加到另一端。前面已经提过，变量的长度取决于使用的平台。

我们往往须要对 `system` 的返回值使用移位操作符，它是一个两字节的数（其他不同版本的 libc 库的 `wait` 函数也是）。返回值的低字节包含触发的信号和内核信息，高字节才是我们需要的外部命令的返回值。可以简单地把所有的位向右移动 8 位，不须要使用掩码，因为低字节会在移位操作后消失：

```
my $rc = system( 'echo', 'Just another perl hacker, ' );
my $exit_status = $rc >> 8;
```

我们也不用保存 `system` 调用的返回值，因为 Perl 把它放在了特殊变量`$?`中：

```
system( 'echo', 'Just another perl hacker, ' );
my $exit_status = $? >> 8;
```

我们可以检查`$?`以判断什么地方出了问题，不过这要使用合适的掩码：

```
my $signal_id   = $? & 0b01111111;     # or 0177, 127, 0x7F
my $dumped_core = $? & 0b10000000;     # or 0200, 128, 0x80
```

注 5：虽然它也介绍了通常人们用"perl" 表示二进制的程序，而用"Perl"表示所有其他的东西。

位向量
Bit Vectors

位向量可以在一个标量中保存多个变量，从而节省内存空间。可以使用一个长的比特串来保存多个变量，而不用使用标量的数组。每一个创建的标量都有空间上的额外开销，即使空的标量也会占据一些内存空间。我们可以使用 Devel::Size 查看一下标量的大小：

```
#!/usr/bin/perl
# devel-size.pl

use Devel::Size qw(size);

my $scalar;

print "Size of scalar is " .
size( $scalar ) . " bytes\n";
```

在我的 MacBook 笔记本上，Perl 5.8.8 的环境下，一个标量在没有赋值的时候就占据了 12 字节！

```
Size of scalar is 12 bytes.
```

我们可以使用 Devel::Peek 来查看标量里面的内容：

```
#!/usr/bin/perl
# devel-peek.pl

use Devel::Peek;

my $scalar;

print Dump( $scalar );
```

输出表明，在没有赋值的时候 Perl 就已经对该变量进行了一些基本的设置：

```
SV = NULL(0x0) at 0x1807058
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
```

即使没有赋值，标量中也有一个引用计数和一些标志位。可以想象，在一个有成百上千个标量的数组中，每个标量都会占用大量的额外空间。即使没有赋值这也是一笔很大的内存开销。

我们可以不使用 Perl 的数组来保存数据。假如有其他方法来存取大量的数据，我们就可以避免 Perl 变量的额外开销，从而节省大量的内存空间。

最简单的方法是用一个长字符串的每一个（或者多个）字符表示一个元素。假设我们在处理 DNA（生物学中的概念，不过你最好使用 BioPerl 来处理它），我们使用字母 T、A、C 和 G 来表示构成 DNA 串的基本碱基对（我们在第 17 章介绍 tied 变量的时候也会使用这些

字母)。我们没有把 DNA 串保存在一个标量的数组中，而是把它们作为一个字符序列保存在了一个字符串中，这样就只会有一个标量的额外开销：

```
my $strand = 'TGACTTTAGCATGACAGATACAGGTACA';
```


我们可以使用 substr() 访问该字符串。为此须要指定开始的位置和长度：

```
my $codon = substr( $strand, 3, 3 );
```

我们甚至可以改变字符串的值，因为 substr() 可以当作左值来使用：

```
substr( $strand, 2, 3 ) = 'GAC';
```

当然，我们可以把这个操作隐藏在函数中，也可以利用字符串构造一个对象，然后调用该对象的方法来访问或改变需要的部分。

更复杂的一种方式是使用 pack()（参见第 14 章）。它也能完成同样的功能，不过它的灵活性更大。我们可以把多种不同类型的变量保存在一个字符串中，之后再把它们分开。这里我忽略掉了例子部分，感兴趣的读者可以参考 Tie::Array::PackedC 模块。该模块把一系列的整数（或浮点数）保存在一个字符串中，而不是把它们的数值和字符串值保存在单独的标量中。

一个位向量也可以实现和前面两种方法相同的功能。它可以在一个标量中保存多个值。就像 DNA 的例子和 pack() 所做的事情一样，完全由我们决定如何分配位向量来表示各个元素。

函数 vec

The vec Function

内置的函数 vec() 会把一个字符串当作位向量来处理。它根据我们指定的长度把一个字符串分成若干个元素，不过长度必须是 2 的幂。它和 substr() 相类似，都是从一个字符串中提取出感兴趣的部分，不过 vec 一次只能取出一个元素。

我们可以用 vec 处理任何的字符串。这个例子使用 8 作为元素的长度，该长度正好是一个单字节字符的长度：

```
#!/usr/bin/perl
# vec-string.pl

my $extract = vec "Just another Perl hacker,", 3, 8;

printf "I extracted %s, which is the character '%s'\n",
        $extract,
        chr($extract);
```

从输出中可以看到，$extract 是一个数字，须要使用 chr 来把它转换成对应的字符：

```
I extracted 116, which is the character 't'
```

我们也可以从头开始建立一个字符串。函数 vec 是一个左值，可以给它赋值。就像 Perl 中

的其他东西一样，第一个元素的下标是 0。由于 vec 是逐位处理的，要想把字符串中的小写 *p* 变成大写的，我们须要使用 ord 得到 P 对应的数字，再赋值给 vec：

```
my $bit_field = "Just another perl hacker,";
vec( $bit_field, 13, 8 ) = ord('P');

print "$bit_field\n"; # "Just another Perl hacker,"
```

之前我曾经演示过，“perl”和“Perl”只有一个比特的差别。其实我们不必改变整个字符，只改变正确的比特位（注 6）就可以了：

```
my $bit_field = "Just another perl hacker,";
vec( $bit_field, 109, 1 ) = 0;

print "$bit_field\n"; # "Just another Perl hacker,"
```

当我们使用 vec 处理字符串的时候，Perl 会把它当成一个单字节元素的字符串，从而扔掉该字符串可能带有的编码。就是说，vec 可以处理任何字符串，不过它会把它变成一个单字节元素的字符串。所以说不要用 vec 处理需要作为字符串处理的串：

```
#!/usr/bin/perl
# vec-drops-encoding.pl

use Devel::Peek;

# 通过包含 unicode 序列以设置 UTF-8 标记
my $string = "Has a unicode smiley --> \x{263a}\n";
Dump( $string );

# 读取操作不会破坏 UTF-8 标记
print STDERR "-" x 50, "\n";
my $first_char = vec( $string, 0, 8 );
Dump( $string );

# 赋值操作会导致 UTF-8 标记丢失
print STDERR "-" x 50, "\n";
vec( $string, 0, 8 ) = ord('W');
Dump( $string );
```

从 Devel::Peek 的输出中可以看到，创建字符串的时候有 UTF8 的标志。我们得到的三个原始字节是\342\230\272，而 Perl 可以根据编码的设置知道它是 Unicode 编码的：

```
SV = PV(0x1801460) at 0x1800fb8
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY,POK,pPOK,UTF8)
  PV = 0x401b10 "Has a unicode smiley --> \342\230\272\n"\0
       [UTF8 "Has a unicode smiley --> \x{263a}\n"]
  CUR = 29
  LEN = 32
```

注 6：我是怎么知道正确的比特位的呢？我比较懒惰。我使用 foreach my $bit (100..116) 进行了遍历，然后再从中找出可以正常工作的一个值。

我们可以用 vec 在不影响 UTF8 标志的情况下提取出字符串中的部分内容。直接通过 vec 访问字符串会对变量造成一些影响，不过它依然还是 UTF8 编码的：

```
----------------------------------------------
SV = PVMG(0x180aca0) at 0x1800fb8
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY,SMG,POK,pPOK,UTF8)
  IV = 0
  NV = 0
  PV = 0x401b10 "Has a unicode smiley --> \342\230\272\n"\0
       [UTF8 "Has a unicode smiley --> \x{263a}\n"]
  CUR = 29
  LEN = 32
  MAGIC = 0x4059d0
        MG_VIRTUAL = &PL_vtbl_utf8
        MG_TYPE = PERL_MAGIC_utf8(w)
        MG_LEN = 27
```

最后，一旦我们通过 vec 改变了字符串，Perl 就会把它当成一个简单字节的序列。当我们把开头的字母 H 变成 W 之后，Perl 会忘记编码的设置。一旦我们把它当作位向量来使用，就完全由我们来决定当前的上下文，由我们来解释每个比特的含义。如果想继续使用字符串，就须要做一些额外的工作：

```
----------------------------------------------
SV = PVMG(0x180aca0) at 0x1800fb8
  REFCNT = 2
  FLAGS = (PADBUSY,PADMY,SMG,POK,pPOK)
  IV = 0
  NV = 0
  PV = 0x401b10 "Was a unicode smiley --> \342\230\272\n"\0
  CUR = 29
  LEN = 32
  MAGIC = 0x4059d0
        MG_VIRTUAL = &PL_vtbl_utf8
        MG_TYPE = PERL_MAGIC_utf8(w)
        MG_LEN = -1
```

位字符串的存储

Bit String Storage

其实，位字符串真正的存储方式是很复杂的。如果改变了一个存有很多变量的标量中的某个部分，然后再检查该标量的内容，结果看起来就会像某个地方出了问题一样。Perl 会把位向量当作字符串保存起来，所以检查时得到的结果几乎毫无意义：

```
#!/usr/bin/perl
# vec-wacky-order.pl

{
my @chars = qw( a b c d 1 2 3 );

my $string = '';

for( my $i = 0; $i < @chars; $i++ )
        {
        vec( $string, $i, 8 ) = ord( $chars[$i] );
```

```
        }

print "\@chars string is ---> [$string]\n";
}

#------

{
my @nums = qw( 9 2 3 12 15 );

my $string = '';

for( my $i = 0; $i < @nums; $i++ )
        {
        vec( $string, $i, 4 ) = 0 + $nums[$i];
        }

print "\@nums string is ---> [$string]\n";

my $bit_string =  unpack( 'B*', $string );

$bit_string =~ s/(....)(?=.)/${1}_/g;

print "\$bit_string is ---> [ $bit_string ]\n";
}
```

每个元素 8 个比特，也就是每个元素占用一个字节。这是非常容易理解的，一点也不复杂。位向量中的第一个元素是第一个字节，第二个元素是第二个字节，依次类推。程序的第一部分创建了一个可见的字符串，可以看到字符的顺序正好和添加的顺序相同：

```
@chars string is ---> [abcd123]
```

程序的第二个部分稍有不同。我们把每个元素的宽度设为 4，并添加了若干个数字。作为字符串，它看起来一点也不像它的元素。不过当我们逐一查看每一位的时候，可以看到每个 4 比特的数，虽然顺序和我们添加的时候不同，而且明显有一个额外的数字：

```
@nums string is ---> [)√]
$bit_string is ---> [ 0010_1001_1100_0011_0000_1111 ]
                        2    9   12    3     0    15
```

如果每个元素的宽度是 1 个、2 个或 4 个比特，Perl 依然会把字符串当成单字节的串处理，不过会按照小字节序方式排列字符串中的比特。下面我们使用字母来演示排列的顺序，假设每个元素两个字节。元素的正确顺序是 A、B、C、D，但是 vec 是从每个字节的低位（也就是右边）开始的，填满一个字节之后再移动到下一个字节：

```
4 bits:       B       A

2 bits:   D   C   B   A

1 bit:  H G F E D C B A
```

我写了一个小程序来演示元素的顺序。对于每一种不同的比特串长度，我计算出最后一个元素的索引（从 0 开始）和该长度的每一位都是 1 的比特串（使用 oct 得到，不过我们须要记得在比特串的前面加上“0b”）。当我们运行该程序的时候，第一行输出显示了比特串的域，之后的一行显示了真正的存储情况：

```
#!/usr/bin/perl
# vec-4bits.pl

foreach my $bit_length ( qw( 4 2 1 ) )
        {
        print "Bit length is $bit_length\n";
        my $last    = (16 / $bit_length) - 1;
        my $on_bits = oct( "0b" . "1" x $bit_length );

        foreach my $index ( 0 .. $last )
               {
               my $string = "\000\000";

               vec( $string, $index, $bit_length ) = $on_bits;

               printf "%2d: ", $index;
               print show_string( $string ), "\n   ", show_ord( $string ), "\n";
               }
        print "\n";
        }

sub show_string
        {
        unpack( "b*", $_[0] );
        }

sub show_ord
        {
        my $result = '';

        foreach my $byte ( split //, $_[0] )
               {
               $result .= sprintf "%08b", ord($byte);
               }

        $result;
        }
```

如果须要看到以 0 和 1 表示的位向量，可以使用带有格式符 b 的 unpack。这样得到的比特串会按照期望的方式排列，而不是那种前面演示的在长度小于 8 时出现的奇怪的顺序：

```
$bit_string = unpack( "b*" , $bit_vector);
```

不过不用担心这个，只要我们每次都使用 vec 访问和存储数据，并且每次使用的比特串的长度都相同就不会有问题。

保存 DNA 序列
Storing DNA

在前面的 DNA 的例子中，我们有 4 个东西须要保存：T、A、C、G。我们其实可以只使用两个比特位来保存每个元素，而不是之前使用的整个字符（8 个比特）。在这个例子中，我们将把一个 12 个字符的字符串压缩成一个只有 3 个字节的位向量：

```
my %bit_codes = (
        T => 0b00,
        A => 0b11,
        C => 0b10,
        G => 0b01,
        );

# 添加反向映射
@bit_codes{values %bit_codes} = keys %bit_codes;

use constant WIDTH => 2;

my $bits = '';
my @bases = split //, 'CCGGAGAGATTA';

foreach my $i ( 0 .. $#bases ) {
        vec( $bits, $i, WIDTH ) = $bit_codes{ $bases[$i] };
        }

print "Length of string is " . length( $bits ) . "\n";
```

这就是我们的 12 个元素的位向量。现在假设须要取出第三个元素。我们传给 `vec()` 三个参数：位向量、元素的个数和每个元素的宽度。然后我们使用从 `vec()` 中得到的值去哈希表中查找原始的元素名。该哈希表是一个双向映射表：

```
my $base = vec $bits, 2, WIDTH;
printf "The third element is %s\n", $bit_codes{ $base };
```

一种更酷的方式是用 4 个比特表示一个元素，每个比特代表一个碱基。看起来这样好像浪费了 3 个比特。如果我们知道碱基的类型，这样做就没有必要。不过有的时候我们不知到碱基的类型。在这种情况下我们可能只知道一个元素不是 A，所以它可能是其他任何一个元素。生物信息学家用其他的字母来表示这些情况（在这种情况下，B 表示不是 A）。不过现在我们不须要这样做。

记录事情
Keeping Track of Things

在《The Perl Review》的“Generating Sudoku”一文中，Eric Maki 使用位向量来表示数独游戏可能的解法。他使用 9 个比特来表示棋盘的每一行，每个比特代表一格，当一个格子有值的时候就设为 1。这样棋盘的一行就可以表示成：

```
0 0 0 1 0 1 1 0 0
```

对于棋盘中的每一行，他都用 9 个比特来表示，这样总共用了一个 81 比特的位串来表示所有的格子。他的算法比这个表示方式要复杂很多，不过这里我们感兴趣的只是位操作。

对他来说检查一个可能的解法非常容易。一旦任何一格有了值，他就可以排除掉任何其他该格子里也有值的方案。不用做太多工作就可以做到，因为他所需要的只是位操作。

他也知道应该排除什么样的解法，因为对备选的行和当前的方案进行位与操作后的结果至少应该有一个共同的位。我们把当前方案中用来和备选方案中的行进行比较的行称为枢轴行（pivot row）。下面例子中的两行有一个比特相同，因此位与的结果为真。和之前一样，我们不须要进行任何移位操作，只须要知道值是否为真，至于真正的值是多少并不重要。稍后我们再讨论这个问题：

```
  0 0 1 0 0 0 1 0 0 # 备选行
& 0 0 0 1 0 1 1 0 0 # 枢轴行
---------------------
  0 0 0 0 0 0 1 0 0 # 位设置，消除行
```

在另一个情况下，备选行和当前解法的相应行没有共同的比特，所以 AND 返回了全 0：

```
  0 1 0 0 1 0 0 0 1 # 仍是备选行
& 0 0 0 1 0 1 1 0 0 # 枢轴行
---------------------
  0 0 0 0 0 0 0 0 0 # false, still okay
```

这里我们须要小心！vec()使用的是字符串，而所有的字符串除了“0”之外都为真（包括“00”等等），因此我们无法立刻根据字符串的值判断结果是否全为 0。

不过， Eric 使用位操作的范围不仅仅限于解决智力问题。他还用位向量来还记录所有不须要考虑的行。总共有 93 个可能的组合，Eric 使用位向量来保存它们。每一位都表示一个备选的行。当某一位被设置的时候，它所代表的行就不再是备选的行了。每一位都对应着另一个数组的下标。通过关掉位向量中的某一位，他避免了从数据结构中删除一个元素，从而节省了 Perl 维护数据结构的大量时间。在这个例子中，他使用了一个位向量来提高速度，虽然消耗了更多的内存。

一旦知道可以忽略某一行后，他就设置$removed 位向量的相应位：

```
vec( $removed, $row, 1 ) = 1;
```

当须要知道所有可用的备选行时，只须要对排除的行取反。这里要小心！不要错误地使用了 binding 操作符：

```
$live_rows = ( ~ $removed );
```

总结
Summary

虽然在大部分情况下，Perl 把我们和计算机的物理细节隔离了开来，不过当我们须要把数据打包到字节中的时候，还是须要处理它们。当 Perl 的数据结构占据了太多的内存时，我们也可能希望把数据压缩到位字符串中以避免 Perl 在内存上的额外开销。一旦有了位向量，我们就可以用和其他语言几乎相同的方法操作它。

深入阅读
Further Reading

文档 *perlop* 介绍了位操作符。文档 *perlfunc* 介绍了内置函数 `vec`。

Mark Jason Dominus 在他的演讲“File Locking Tricks and Traps”的幻灯片中演示了如何使用文件锁和 `Fcntl` 模块。讨论部分有很多位或操作（*http://perl.plover.com/yak/flock/*）。

Eric Maki 在《The Perl Review 2.2》（Spring 2006）上写了“Generating Sudoku”一文，并且使用了 `vec` 来记录状态，避免占用太多内存。

我为《The Perl Review 2.2》（Spring 2006）写了“Working with Bit Vectors”一文，以补充 Eric 关于数独的文章。该文章构成了本章的基础，不过在这里我还是进行了大量地扩充。

Maciej Ceglowski 为 Perl.com 介绍了“Bloom Filters”。Bloom filters 把数据哈希之后当作键保存起来而不用保存值，它大量地使用了位操作。（*http://www.perl.com/lpt/a/2004/04/08/bloom_filters.html*）。

如果 `vec` 和 Perl 的位操作都不能满足你的要求，你可以试一试 Stephen Breyer 的 `Bit::Vector` 模块。它允许在位向量中使用任何长度的元素。

Randal Schwartz 为 1998 年一月的《Unix Review》写了“Bit Operations”一文：*http://www.stonehenge.com/merlyn/UnixReview/col18.html*。

第17章

奇妙的绑定变量

The magic of Tied Variables

Perl 允许我们通过一种被称为**绑定**（tying）的机制深入到变量中。利用这个机制，我们可以在访问变量、保存变量等一切操作变量的时候改变变量的行为。

绑定变量使用的是 Perl 最基础的东西。对于绑定变量，我们可以完全决定 Perl 在访问或保存变量时的行为。在内部，我们须要实现变量行为的所有逻辑。由于这些是我们实现的，因此可以让这些看起来像普通变量的变量来做程序可以完成的任何事情（这可是有非常多的功能的）。虽然在内部使用了很多技巧，对于用户来说，绑定变量看起来就和普通的变量一样。不仅如此，绑定变量是通过 Perl 的 API 来工作的。Perl 的普通变量在内部也使用了这种绑定机制。

似是而非

They Look Like Normal Variables

你可能已经见过使用的绑定变量了，虽然没有使用关键字 `tie`。比如说命令 `dbmopen` 就可以把一个哈希表和一个数据库文件绑定起来：

```
dbmopen %DBHASH, "some_file", 0644;
```

不过，这是 Perl 中一种很老的用法。从那以后，磁盘上的哈希表的数目和种类都有了很大的增长。每一种哈希表的实现都解决了另一种的一些问题。如果想使用这些新的哈希表而不是 Perl 在 `dbmopen` 中使用的版本的话，我们可以使用关键字 `tie` 把哈希表和相应的模块关联起来：

```
tie %DBHASH, 'SDBM_File', $filename, $flags, $mode;
```

这里有一些隐藏的技巧。程序员看到的变量 `%DBHASH` 和一个正常的哈希表一样。为了达到这个效果，Perl 维护了一个和变量 `%DBHASH` 关联在一起的秘密对象。实际上，我们可以通过 `tie` 的返回值得到该对象：

```
my $secret_obj = tie %DBHASH, 'SDBM_File', $filename, $flags, $mode;
```

如果我们忘记了在调用 `tie` 的时候得到该对象，也可以稍后使用命令 `tied` 得到它。无论是哪种方法，最后我们都会得到一个看起来很普通的变量和相应的对象，而且我们可以使用任何一个来进行操作：

```
my $secret_obj = tied( %DBHASH );
```

每次操作%DBHASH 的时候，Perl 都会把该操作转换成对$secret_obj 的某个方法的调用。每种类型的变量（标量、数组，等等）都有着不同的行为，所以它们有不同的方法，而这正是我们须要实现的。

你可能已经在不知道绑定变量的情况下使用过它们了。在《Intermediate Perl》的第 5 章中，我们介绍过如何打开一个到标量的引用的文件句柄：

```
open my($fh), ">", \$print_to_this_string
      or die "Could not open string: $!";
```

获得文件句柄之后，我们可以使用它进行各种文件操作。这一切是如何实现的呢？模块 IO::Scalar 实现了一个绑定的文件句柄来处理文件句柄的各种行为。它可以做任何想做的事情。在这个例子中，它把打印的内容输出到了一个标量中，而不是文件中。

在用户层面
At the User Level

在过去那些须要写很多 HTML 代码的日子里，我非常喜欢使用有着隔行交错颜色的表格。这并不难实现，不过却很乏味。我们须要把使用的颜色保存在一个地方，然后在创建新的一行的时候从列表中找出下一个颜色：

```
@colors = qw( AAAAAA CCCCCC EEEEEE );

my $row = 0;
foreach my $item ( @items )
        {
        my $color = $colors[ $row++ % ($#colors + 1) ];

        print qq|<tr><td bgcolor="$color">$item</td></tr>|;
        }
```

这些额外的代码让人非常讨厌。我尤其不喜欢在 foreach 循环的外面定义$row 变量。这其实并不是什么问题，不过从美学角度来看它不太好。这个循环做的事情只是简单地创建一行表格，为什么还须要选择颜色的逻辑呢？

为此，我创建了 Tie::Cycle 模块来解决这个问题。我没有使用数组，而是定义了标量的一种特殊行为：每次访问这个特殊的标量时，它都会返回列表中的下一个颜色。绑定机制帮助我们处理了其他的所有事情。这样做的一个额外的好处是不容易犯差一（off-by-one）的错误——我很容易在实现这个逻辑的时候犯这种错误：

```
use Tie::Cycle;
tie my $color, 'Tie::Cycle', [ qw( AAAAAA CCCCCC EEEEEE ) ];

foreach my $item ( @items )
        {
        print qq|<tr><td bgcolor="$color">$item</td></tr>|;
        }
```

我们还可以多次使用绑定变量$color。不管循环是在什么地方停止的，都可以把它重置成开始状态，这样每一组表格的行都可以从同样的颜色开始。下面使用命令 tied 得到隐藏对象并调用 reset 方法——该方法是在创建模块时实现的：

```
tied( $color )->reset;

foreach my $item ( @other_items )
        {
        print qq|<tr><td bgcolor="$color">$item</td></tr>|;
        }
```

使用模块 Tie::Cycle，我们给一个数组提供了一个标量的接口。不过也可以不这么复杂。我们可以使用普通的接口，只用简单地给这种数据类型增加一些存储和访问的限制。稍后我再详细介绍。

拉开帷幕
Behind the Curtain

在内部，Perl 为绑定变量使用了一个对象。用户虽然没有把绑定变量当作对象使用，Perl 却会找到调用的方法并进行正确的操作。

对于程序员来说，一旦要实现一个变量的行为，就须要告诉变量所有的事情该怎么做。绑定机制会使用一些特殊名字的方法，所有这些方法都须要我们来实现。由于每一种变量类型都有所不同（我们可以对数组进行 unshift 操作，但是无法这样操作标量；我们可以得到一个哈希表的所有键，但是不能这样处理数组），所以每一种类型的变量都有一些特殊的 tie 方法。

Perl 5.8 提供了一些基类来帮助我们。我们可以把 Tie::Scalar、Tie::Array、Tie::Hash 或 Tie::Handle 作为自己的 Tie::*模块的基础。不过我常常发现要做一些特别的事情的时候，这些模块就派不上什么用场了。

每种类型的变量都有一个构造方法，该方法的名字是在变量类型前加上 TIE（比如 TIESCALAR，等等）。变量也可能有 UNTIE 和 DESTROY 方法。除此以外，每一种变量都有和它的特有行为对应的方法。

Perl 会在使用关键字 tie 的时候调用构造方法。这里再看一下之前的一个例子：

```
tie my $color, 'Tie::Cycle',
        [ qw( AAAAAA CCCCCC EEEEEE ) ];
```

Perl 会得到类的名字——Tie::Cycle，调用该类的方法——TIESCALAR，并把命令 tie 的其余参数传给 TIESCALAR：

```
my $secret_object = Tie::Cycle->TIESCALAR(
        [ qw( AAAAAA CCCCCC EEEEEE ) ] );
```

获得隐藏对象后，Perl 会把它和变量$color 关联起来。

跳出$color的作用域之后，Perl会把它解释成对隐藏对象的DESTROY方法的调用，并调用该方法：

```
$secret_object->DESTROY;
```

我们也可以决定不再绑定变量。调用 untie 可以取消秘密对象和变量之间的绑定。这样$color就变成一个普通的标量了：

```
untie $color;
```

Perl会把该命令转换成对UNTIE的调用，从而取消秘密对象和变量之间的绑定：

```
$secret_object->UNTIE;
```

标量
Scalars

绑定标量是最容易实现的，因为标量的功能不多——我们只能存储和访问标量。实现标量的行为须要创建两个方法：STORE——Perl在赋值的时候调用；FETCH——Perl在访问变量的时候调用。除此之外，我们还提供了TIESCALAR方法供Perl在我们使用tie的时候调用，以及可能会用到的DESTROY和UNTIE方法。

TIESCALAR方法工作方式和其他的构造方法一样。第一个参数是类的名字，第二个参数是其余参数的列表。这些参数都直接来自于命令tie。

Tie::Cycle

在例子Tie::Cycle中，要绑定的变量之后的所有东西（类的名字和其余的参数）都会被作为TIESCALAR的参数。除了方法的名字之外，它看起来就像一个普通的构造方法。Perl会替我们处理所有的绑定操作，所以不必自己做：

```
tie $colors, 'Tie::Cycle', [ qw( AAAAAA CCCCCC EEEEEE ) ];
```

这种方式几乎和自己调用TIESCALAR一样：

```
my $object = Tie::Cycle->TIESCALAR( [ qw( AAAAAA CCCCCC EEEEEE ) ] );
```

不过，对于后一种方法，由于没有使用tie，我们得到的只有一个对象，Perl并不知道有特殊接口。它只是一个普通的对象。

在模块Tie::Cycle（从CPAN上可以得到）中，模块的开始部分非常简单：我们声明软件包的名字、设置好常用的部分、定义自己的TIESCALAR方法。我决定给该接口两个参数：类名和一个匿名数组。这样选择并没有什么特别的。TIESCALAR会得到tie传来的所有参数，完全由我们决定如何处理它们——当然也包括如何强化接口。

在这个例子中，我们的目的很简单：只须要确保有一个参数是数组的引用，并且参数多于一个。和其他的构造方法一样，该方法返回一个被保佑的（blessed）引用。虽然绑定的是一个标量，我们却使用了一个匿名数组作为对象。不过只要保持一致，Perl 并不在乎我们怎么做。从外部看来，它就像个标量一样：

```
package Tie::Cycle;
use strict;

use vars qw( $VERSION );

$VERSION = sprintf "%d.%02d", q$Revision: 1.9 $ =~ m/ (\d+) \. (\d+) /xg;

sub TIESCALAR
        {
        my $class    = shift;
        my $list_ref = shift;

        my @shallow_copy = map { $_ } @$list_ref;

        return unless ref $list_ref eq ref [];

        my $self = [ 0, scalar @shallow_copy, \@shallow_copy ];

        bless $self, $class;
        }
```

得到绑定变量之后，我们就可以像使用该类型的普通变量那样使用它了。我们可以像使用其他标量那样使用绑定标量。我们已经把一个匿名数组保存在对象中了，如果想改变它，可以直接给它赋值。在这个例子中，我们须要赋给它一个匿名数组：

```
$colors = [ qw(FF0000 00FF00 0000FF) ];
```

在帷幕之后，Perl 会调用 STORE 方法。再次强调一下，我们不能改变该方法的名字，而且还须要自己处理所有的事情。这里须要做的事情和 TIESCALAR 子程序一样。其实这段代码可以重构一下，不过对于这个小模块来说，重构带来的问题会比它本身的更多：

```
sub STORE
        {
        my $self     = shift;
        my $list_ref  = shift;

        return unless ref $list_ref eq ref [];

        $self = [ 0, scalar @$list_ref, $list_ref ];
        }
```

Perl 会在每次试图获取标量值的时候调用 FETCH。和之前一样，我须要做所有的工作，须要知道如何返回需要的值。在返回一个值的前提下，我们可以做任何事情。在 Tie::Cycle 中，须要得到要访问的元素的下标，然后返回该元素。我们给下标值加一，把结果对数组中元素的个数取模，然后返回结果：

```
sub FETCH
        {
        my $self   = shift;
        my $index  = $self->[0]++;
        $self->[0] %= $self->[1];
        return $self->[2]->[ $index ];
        }
```

这就是所有要做的事情。我们可以也增加一个 UNTIE（或 DESTROY）方法，不过我没有这样做，因为我们没有生成任何须要清理的垃圾。这些方法也没有什么神秘的。它们和你知道的 DESTROY 方法是一样的。

如果看一看 Tie::Cycle 的源代码，你还会看到一些额外的方法。我们没法通过 tie 的接口访问这些方法，但是可以通过对象访问到。它们和 tie 机制没有关系。该对象只是一个普通的对象，我们完全可以把它当作对象来处理，包括添加方法。比如说，可以用 previous 方法得到列表中前一个元素的值，而不改变下标。使用它可以在不改变任何东西的情况下查看列表的元素：

```
my $previous = tied( $colors )->previous;
```

我们可以通过 tied 得到秘密的对象后立刻调用该对象的方法而不用把对象保存到变量中。类似地，我们也可以使用 next 来查看下一个元素：

```
my $next     = tied( $colors )->next;
```

最后，就像之前介绍的那样，我们可以重置循环：

```
tied( $colors )->reset;
```

有界的整数
Bounded Integers

下面我要创建一个绝对值大小有上限的绑定变量，变量中的值只能在一定范围之内。要创建一个支持绑定机制的类，我们须要做的工作和之前的 Tie::Cycle 一样：创建 TIESCALAR、STORE 和 FETCH 方法：

```
package Tie::BoundedInteger;
use strict;

use Carp qw(croak);

use vars qw( $VERSION );

$VERSION = 1.0;

sub TIESCALAR
        {
        my $class = shift;
        my $value = shift;
        my $max   = shift;
```

```
        my $self = bless [ 0, $max ], $class;

        $self->STORE( $value );

        return $self;
        }

sub FETCH { $_[0]->[0] }
sub STORE
        {
        my $self      = shift;
        my $value     = shift;

        my $magnitude = abs $value;

        croak( "The [$value] exceeds the allowed limit [$self->[1]]" )
               if( int($value) != $value || $magnitude > $self->[1] );

        $self->[0] = $value;

        $value;
        }

1;
```

从用户的角度来看，我们做的事情和之前一样。我们调用 `tie`，指定好变量的名字、实现该行为对象的名字和使用的参数。在这个程序中，我希望整数的初始值为 1，绝对值不超过 3。完成这些之后，我们尝试把-5 到 5 之间的整数逐一赋给`$number` 并打印出结果：

```
#!/usr/bin/perl

use Tie::BoundedInteger;

tie my $number, 'Tie::BoundedInteger', 1, 3;

foreach my $try ( -5 .. 5 )
        {
        my $value =  eval { $number = $try };

        print "Tried to assign [$try], ";
        print "but it didn't work, " unless $number == $try;
        print "value is now [$number]\n";
        }
```

从输出中可以看到，`$number` 的初值是 1。当我们试图把 7 赋给它的时候（绝对值大于 5），操作没有成功，值还是 1。通常我的程序会调用 `croak`，这里我们使用了 `eval` 来捕获错误。对于 6 也是一样的结果。当我们尝试 5 的时候，赋值终于成功了：

```
Tried to assign [-5], but it didn't work, value is now [1]
Tried to assign [-4], but it didn't work, value is now [1]
Tried to assign [-3], value is now [-3]
```

```
Tried to assign [-2], value is now [-2]
Tried to assign [-1], value is now [-1]
Tried to assign [0], value is now [0]
Tried to assign [1], value is now [1]
Tried to assign [2], value is now [2]
Tried to assign [3], value is now [3]
Tried to assign [4], but it didn't work, value is now [3]
Tried to assign [5], but it didn't work, value is now [3]
```

自动销毁的值
Self-Destructing Values

例子 Tie::BoundedInteger 通过对数值加以限制改变了存储数值的方式。我们也可以改变访问数值的方式。在下面的例子中，我们会创建 Tie::Timely 模块，该模块给数值设定了一个生命期。超过生命期之后，访问该变量时会得到 undef。

STORE 方法很容易实现。我们只须要把得到的值保存起来就可以了。我们不在乎得到的是一个标量、引用、对象还是别的东西。不过，每次保存值的时候，我们须要记录当前的时间。这样每次改变值的时候，都会重置计时器。

FETCH 方法可以返回两种结果。如果在生命期之内，该方法返回保存的值；否则不返回任何东西：

```
package Tie::Timely;
use strict;

use Carp qw(croak);

use vars qw( $VERSION );

$VERSION = 1.0;

sub TIESCALAR
        {
        my $class         = shift;
        my $value         = shift;
        my $lifetime = shift;

        my $self = bless [ undef, $lifetime, time ], $class;

        $self->STORE( $value );

        return $self;
        }

sub FETCH { time - $_[0]->[2] > $_[0]->[1] ? () : $_[0]->[0] }

sub STORE { @{ $_[0] }[0,2] = ( $_[1], time ) }

1;
```

数组
Arrays

实现绑定数组的方法和实现绑定标量的方法相同，不过我们须要创建更多的方法，因为数组有更多的功能。我们的实现须要处理各种数组操作（移出、移入、压栈、出栈、切片）和其他常见的数组操作：

- 得到或设置数组的最后一个元素的下标。
- 增大数组。
- 检查一个下标是否存在。
- 删除一个元素。
- 清除所有的元素。

一旦决定要自己实现数组的所有行为之后，我们就要对这一切负责到底。我们也可以不给所有的操作定义相应的方法，但是这样这些操作就不能正常工作了。模块 `Tie::Array` 实现了大部分的基本操作，可以作为一个基类使用。它会在程序试图使用某个不存在的功能时调用 `croak`。表 17-1 显示了一些数组操作和绑定的方法的对应关系（文档 *perltie* 有全部的对应关系）。大部分方法都和 Perl 的操作符名字相同，不过名字是全大写的。

表 17-1：一些数组操作和绑定的方法的对应关系

操作名	数组操作	绑定的方法
赋值	`$a[$i] = $n`	`STORE( $i, $n )`
读取元素	`$n = $a[$i];`	`FETCH( $i )`
获取数组长度	`$l = $#a;`	`FETCHSIZE()`
预留空间	`$#a = $n;`	`STORESIZE( $n )`
添加元素到末尾	`push @a, @n`	`PUSH( @n );`
从末尾删除元素	`pop @a;`	`POP()`

重新发明数组
Reinventing Arrays

在介绍 tying 标量的时候，我演示了 `Tie::Cycle` 模块，该模块创建了一个和标量的行为一样的数组。为了公平起见，我也创建一个和数组的行为一样的标量。为了避免每个元素额外的内存开销，我没有把元素保存在数组中。相反地，我们创建了一个标量，把它分割成多段以保存数组的各个元素。从本质上看，我们的程序通过牺牲速度换取了内存空间。由于小于 256 的数可以保存在一个字符中，我们可以重用前面的有界整数的例子。这样非常方便，对吧？

```
package Tie::StringArray;
use strict;

use Carp qw(croak);
```

```
use vars qw( $VERSION );

$VERSION = 1.0;

sub _null { "\x00" }
sub _last () { $_[0]->FETCHSIZE - 1 }

sub _normalize_index { $_[1] == abs $_[1] ? $_[1] : $_[0]->_last + 1 - abs $_[1] }

sub _store  { chr $_[1] }
sub _show   { ord $_[1] }
sub _string { ${ $_[0] } }

sub TIEARRAY
        {
        my( $class, @values ) = @_;

        my $string = '';
        my $self = bless \$string, $class;

        my $index = 0;

        $self->STORE( $index++, $_ ) foreach ( @values );

        $self;
        }

sub FETCH
        {
        my $index = $_[0]->_normalize_index( $_[1] );

        $index > $_[0]->_last ? () : $_[0]->_show(
               substr( $_[0]->_string, $index, 1 )
               );
        }

sub FETCHSIZE { length $_[0]->_string }

sub STORESIZE
        {
        my $self     = shift;
        my $new_size = shift;

        my $size = $self->FETCHSIZE;

        if( $size > $new_size ) # truncate
               {
               $$self = substr( $$self, 0, $size );
               }
        elsif( $size < $new_size ) # extend
               {
               $$self .= join '', ($self->_null) x ( $new_size - $size );
               }
        }
```

```
sub STORE
        {
        my $self      = shift;
        my $index     = shift;
        my $value= shift;

        croak( "The magnitude of [$value] exceeds the allowed limit [255]" )
               if( int($value) != $value || $value > 255 );

        $self->_extend( $index ) if $index >= $self->_last;

        substr( $$self, $index, 1, chr $value );

        $value;
        }

sub _extend
        {
        my $self      = shift;
        my $index     = shift;

        $self->STORE( 0, 1 + $self->_last )
               while( $self->_last >= $index );
        }

sub EXISTS  { $_[0]->_last >= $_[1] ? 1 : 0 }
sub CLEAR   { ${ $_[0] } = '' }

sub SHIFT   { $_[0]->_show( substr ${ $_[0] }, 0, 1, '' ) }
sub POP     { $_[0]->_show( chop   ${ $_[0] }          ) }

sub UNSHIFT
        {
        my $self = shift;

        foreach ( reverse @_ )
               {
               substr ${ $self }, 0, 0, $self->_store( $_ )
               }
        }

sub PUSH
        {
        my $self = shift;

        $self->STORE( 1 + $self->_last, $_ ) foreach ( @_ )
        }

sub SPLICE
        {
        my $self      = shift;

        my $arg_count = @_;
        my( $offset, $length, @list ) = @_;
```

```
        if(    0  == $arg_count )
                {
                ( 0, $self->_last )
                }
        elsif( 1 == $arg_count )
                {
                ( $self->_normalize_index( $offset ), $self->_last )
                }
        elsif( 2 <= $arg_count ) # 只需要变量$offset和$length
                {
                ( $self->_normalize_index( $offset ), do {
                        if( $length < 0 ) { $self->_last - $length }
                        else              { $start + $length - 1   }
                        }
                )
                }

        #@removed = map { $self->POP } $start .. $end;


        if( wantarray )
                {
                @removed;
                }
        else
                {
                defined $removed[-1] ? $removed[-1] : undef;
                }
        }

1;
```

我们把字符串中的每一个位置都作为数组的一个元素。保存一个值时，须要把元素的下标和元素的值作为参数传给 STORE。我们须要把要保存的值转化成一个字符，并且把它保存在字符串中正确的位置上。当我们试图保存在 1 到 255 范围之外的数的时候，会得到错误消息。

要想获取一个值，我们须要从字符串的正确位置上提取出字符并把它转化成数字。FETCH 的参数是元素的下标，我们须要把该下标转换成 substr 可以使用的形式。

对于那些更加复杂的数组操作，我们须要做更多的工作。要想得到数组的一个切片 (splice)，我们须要同时获取多个元素。因为切片也可以作为左值使用，所以我们也须要能够给一个区域赋值。不仅如此，一个用户指定的值的数目可能和切片的数目不同，因此我们也须要能够缩短或扩充字符串。不过这些并不可怕，因为所有的这些都可以使用 substr 操作字符串来实现。

删除一个元素要更复杂些。在一个普通的数组中，我们可以有一个没有定义的元素。但是在字符串的中间该如何处理呢？奇妙的是，有一种方法：我们可以把 undef 作为一个空字

节保存起来。不过，如果要保存的数是在 0 到 255 之间，我们就有麻烦了。怎样才能解决这个问题呢？我很好奇。

Perl 也允许我们扩展一个绑定数组。在普通数组中，我们可以通过扩展一个数组来告诉 Perl 为一定数目的元素预留好空间（从而显式地绕过了 Perl 内置的推测内存使用量的逻辑）。在这个例子中，我们只用扩展一下字符串。

一些现实的东西
Something a Bit More Realistic

上面的例子是我虚构的，所以我可以演示整个过程而不涉及任何复杂的东西。我们也可以用它保存一组字符。现在，我希望调整它来保存 DNA 序列。变量范围从 256 变成了更小的集合{T C G A}，它们分别代表着胸腺嘧啶、胞嘧啶、鸟嘌呤和腺嘌呤。如果把 NULL 的情况也考虑进去的话（可能我们的基因测序器不能告诉我们某个位置的碱基是什么），我们总共有 6 种组合。这样我们就不必使用一个完整的字符来表示它。其实只用 3 个比特就可以了，而且还会有一些未使用的状态。

在我们深入讨论之前，我想估计一下能够节省的内存空间。通常一个 DNA 序列有上千个碱基对。如果使用数组的话，每个标量都要占用额外的空间。假设额外的开销是 10 个字节（保守的估计），那么对于一个 10 000 个碱基对的短序列来说，就是 100 000 字节。标量的额外空间累加起来确实不小！现在，假设我们把所有的字母都保存在一个标量中，这样只会有一个标量的额外开销。10 000 个 3 比特的碱基对，总共占用的空间是 30 000 比特，也就是 3 750 字节。我们把它近似成 4 000 字节。这可是 50 倍的差距！不过要记住，内存上的缩减是以执行的速度为代价的。我们为此须要做更多的计算。

使用 3 个比特可以得到 8 个不同的模式串。我们须要给其中的一些模式赋予含义。幸运的是，在 Perl 中很容易实现。只要是 Perl 5.6 及其之后的版本，我们就可以直接使用二进制串（参见第 16 章的位操作）：

```
use constant N => 0b000;
use constant T => 0b001;
use constant C => 0b100;
use constant G => 0b110;
use constant A => 0b011;

use constant RESERVED1 => 0b111;
use constant RESERVED2 => 0b101;
```

不过，由于没有用字符来表示每个元素，我们不能使用 substr 了。如果使用 vec，我们就得按照 2 的幂来分割比特串，这样会浪费 1 个比特（我们已经浪费了两个状态（注 1））。如果这样做，就会有 10 个不用的模式。如果最终我们会遇到遗传编码更复杂的外星人的话，这样做会非常好，不过目前我们还是维持原有的假设。

在阅读下面的代码之前，请记住我们要做的事情，这样你就不会被代码吓跑了。它解决的问题还是和上一个例子一样：把数字作为字符保存在一个长字符串中。这一次我们是在比特的层次上进行操作，须要用到更多的数学。在这个例子中，具体的细节并不是那么重要，重要的是只要我们愿意，就可以用绑定机制实现任何需要的功能：

```
package Tie::Array::DNA;
use strict;
use base qw(Tie::Array);

use Carp qw(croak carp);

use vars qw( $VERSION );
$VERSION = 1.0;

use constant BITS_PER_ELEMENT     =>  3;
use constant BIT_PERIOD           => 24; # 24 bits
use constant BYTE_LENGTH          =>  8;
use constant BYTE_PERIOD          =>  3; # 24 bits

my %Patterns = (
        T => 0b001,
        A => 0b011,
        C => 0b100,
        G => 0b110,
        N => 0b000,
        );

my @Values   = ();
foreach my $key ( keys %Patterns )
        {
        $Values[ $Patterns{$key} ] = $key
        }

sub _normalize { uc $_[1] }
sub _allowed   { length $_[1] eq 1 and $_[1] =~ tr/TCGAN// }

my %Last;

sub TIEARRAY
        {
        my( $class, @values ) = @_;

        my $string = \'';
```

注 1：如果只关心 DNA 并且能够准确地知道每个位置的碱基，就可以只用两个比特，这样我们就可以使用 vec。如果希望添加其他的符号，比如用 B 表示非 A 的碱基，那么我们就须要使用更多的比特位。

```
        my $self = bless $string, $class;

        $$self = "\x00" x 10_000;
        $Last{ "foo" } = -1;

        my $index = 0;

        $self->STORE( $index++, $_ ) foreach ( @values );

        $self;
        }

sub _get_start_and_length
        {
        my( $self, $index ) = @_;

        my $bytes_to_start = int( $index * BITS_PER_ELEMENT / BYTE_LENGTH );

        my $byte_group = int( $bytes_to_start / BYTE_PERIOD );

        my $start  = $byte_group * BYTE_PERIOD;

        ( $start, BYTE_PERIOD )
        }

sub _get_bytes
        {
        my( $self, $index ) = @_;

        my( $start, $length ) = $self->_get_start_and_length( $index );

        my @chars = split //, substr( $$self, $start, $length );

        (ord( $chars[0] ) << 16) +
                (ord( $chars[1] ) << 8) +
                        ord( $chars[2] );
        }

sub _save_bytes
        {
        my( $self, $index, $bytes ) = @_;

        my( $start, $length ) = $self->_get_start_and_length( $index );

        my $new_string = join '', map {
                chr(
                        ( $bytes & ( 0xFF << $_ ) )
                        >>
                        $_
                        )
                } qw( 16 8 0 );

        substr( $$self, $start, $length, $new_string );
        }
```

```
sub _get_shift
        {
        BIT_PERIOD - BITS_PER_ELEMENT - ($_[1] * BITS_PER_ELEMENT % BIT_PERIOD);
        }

sub _get_clearing_mask
        { ~ ( 0b111 << $_[0]->_get_shift( $_[1] ) ) }

sub _get_setting_mask
        { $_[0]->_get_pattern_by_value( $_[2] ) << $_[0]->_get_shift( $_[1] ) }

sub _get_selecting_mask
        { 0b111 << $_[0]->_get_shift( $_[1] ) }

sub _get_pattern_by_value     {   $Patterns{ $_[1] }   }
sub _get_null_pattern         {   $Patterns{ 'N'   }   }

sub _get_value_by_pattern     {   $Values  [ $_[1] ]   }

sub _string    { $_[0] }

sub _length { length ${$_[0]} }

sub _add_to_string { ${$_[0]} .= $_[1] }

sub STORE
        {
        my( $self, $index, $value )  = @_;

        $value = $self->_normalize( $value );

        carp( qq|Cannot store unallowed element "$value"| )
                unless $self->_allowed( $value );

        $self->_extend( $index ) if $index > $self->_last;

        # get the mask
        my $clear_mask  = $self->_get_clearing_mask( $index );
        my $set_mask    = $self->_get_setting_mask( $index, $value );

        # clear the area
        my $result = ( $self->_get_bytes( $index ) & $clear_mask ) | $set_mask;

        # save the string
        my( $start, $length ) = $self->_get_start_and_length( $index );

        my $new_string = join '', map {
                chr(
                      ( $result & ( 0xFF << $_ ) )
                      >>
                      $_
                      )
               } qw( 16 8 0 );

        substr( $$self, $start, $length, $new_string );
```

```
        $self->_set_last( $index ) if $index > $self->_last;

        $value
        }

sub FETCH
        {
        my( $self, $index ) = @_;

        # get the right substr
        my $bytes = $self->_get_bytes( $index );

        # get the mask
        my $select_mask  = $self->_get_selecting_mask( $index );
        my $shift        = $self->_get_shift( $index );

        # clear the area
        my $pattern = 0 + ( ( $bytes & $select_mask ) >> $shift );

        $self->_get_value_by_pattern( $pattern );
        }

sub FETCHSIZE { $_[0]->_last + 1 }
sub STORESIZE { $_[0]->_set_last( $_[1] ) }

sub EXTEND  { }
sub CLEAR   { ${ $_[0] } = '' }
sub EXISTS  { $_[1] < $Last{ "foo" }  }

sub DESTROY { }

__PACKAGE__;
```

这一段代码比较复杂，因为我们须要实现自己的数组。由于把所有的东西保存在了一个字符串中，并且把该字符串作为一个长的比特串而不是字符串使用，因此我们须要有一种从比特串中得到需要的信息的方法。

我们的每个元素 3 个比特，可是每个字节有 8 个比特。为了简化计算，我决定每次处理 3 个字节（24 个比特），因为 24 是 3 和 8 的最小公倍数。我们在_get_bytes 和_save_bytes 中进行这种处理，这两个方法会找到须要处理的 3 个字节。_get_bytes 方法把 3 个字符转化成一个数，这样之后就可以对它进行位操作了。_save_bytes 方法则进行逆处理。

从_get_bytes 得到数之后，我们须要取出感兴趣的 3 个比特。每一组有 8 个元素，所以我们使用_get_selecting_mask 找到需要的元素并返回正确的位掩码。位掩码通过对 0b111 进行合适的位移得到。方法_get_shift 通过使用常量 BIT_PERIOD 和 BITS_PER_ELEMENT 来处理位移操作。

得到所有这些之后，FETCH 方法会对它们进行处理以返回一个元素。它先获取比特串，然后调用_get_value_by_pattern 把比特串转化成相应的符号（比如：T、A、C、G）。

STORE 方法也会做这些事情，不过整个顺序是反过来的。它把符号转化成比特串，移位到正确的位置，然后进行相应的位操作来在字符串中设置该值。我们先用_get_clearing_mask 返回的掩码清除目标位，再用_get_setting_mask 得到的位掩码来保存元素。

嗨！你现在还跟得上吗？现在我们还没有实现所有的数组操作。我们该如何实现 SHIFT、UNSHIFT 和 SPLICE 呢？这里给个提示：记住 Perl 会给真正的数组和字符串实现这些功能。我们不必在每次改变数据的时候移动它们。相反地，我们可以记录下开始的位置（不一定是在开头）。须要移出一个元素时，只用把偏移量加 3。这样第一个元素是从第 3 位到第 5 位，而不是第 0 位到第 2 位。具体的实现就留给读者了。

哈希表
Hashes

绑定哈希表只比绑定数组稍微复杂一点。我们用同样的方法来实现它。我们须要实现相应的方法处理所有须要绑定哈希表处理的操作。表 17-2 显示了一些哈希表的操作和相应的绑定的方法。

表 17-2：一些哈希操作和绑定的方法的对应关系

操作	哈希操作	绑定的方法
设置值	`$h{$str} = $val;`	`STORE( $str, $val )`
读取值	`$val = $h{$str};`	`FETCH( $str )`
删除一个键	`delete $h{$str};`	`DELETE( $str )`
检查一个键是否存在	`exists $h{$str};`	`EXISTS( $str )`
获取下一个键	`each %h;`	`NEXTKEY( $str )`
清除哈希表	`%h = ();`	`CLEAR( $str )`

一个常见的操作——对我来说至少如此——是在哈希表中累计某些东西的次数。我在讲授 Perl 时最喜欢使用的一个例子是计算单词的频率。当学生们在学习《Learning Perl》教程的第 3 天的时候，它们就具备了足够的知识来写一个简单的单词计数器：

```
my %hash = ();

while( <> )
        {
        chomp;
        my @words = split;
```

```
        foreach my $word ( @words ) { $hash{$word}++ }
        }

foreach my $word ( sort { $hash{$b} <=> $hash{$a} } keys %hash )
        {
        printf "%4d  %-20s\n", $hash{$word}, $word;
        }
```


不过，当学生们真正开始使用它的时候，会发现事情并没有这么简单——单词有不同的大小写，可能带有不同的标点，甚至会有拼写错误。我们可以给上面的例子程序添加一些代码来处理各种特殊情况，也可以在哈希表的赋值操作中进行处理。下面我们把声明哈希表的语句替换成对 tie 的调用，程序的其他部分则不做任何改变：

```
# my %hash = (); # old way
tie my( %hash ), 'Tie::Hash::WordCounter';

while( <> )
        {
        chomp;
        my @words = split;
        foreach my $word ( @words ) { $hash{$word}++ }
        }

foreach my $word ( sort { $hash{$b} <=> $hash{$a} } keys %hash )
        {
        printf "%4d  %-20s\n", $hash{$word}, $word;
        }
```

我们可以在绑定哈希表中做任何事情，所以可以在给哈希表赋值的时候对单词进行正则化处理，这样就能处理各种特殊情况。我们不须要对单词计数器程序做很大的改动，就可以把所有的工作都隐藏在绑定的接口后面。

大部分的工作会在 STORE 方法中完成。其他的部分和普通的哈希表一样。我们在内部也使用了一个哈希表。由于我们也须要在访问一个键的时候能够忽略大小写和标点，所以 FETCH 方法也用同样的方式对参数进行了规范化：

```
package Tie::Hash::WordCounter;
use strict;
use Tie::Hash;

use base qw(Tie::StdHash);

use vars qw( $VERSION );

$VERSION = 1.0;

sub TIEHASH  { bless {}, $_[0] }

sub _normalize
        {
        my( $self, $key ) = @_;
```

```
        $key =~ s/^\s+//;
        $key =~ s/\s+$//;

        $key = lc( $key );

        $key =~ s/[\W_]//g;

        return $key
        }

sub STORE
        {
        my( $self, $key, $value ) = @_;

        $key = $self->_normalize( $key );

        $self->{ $key } = $value;
        }

sub FETCH
        {
        my( $self, $key ) = @_;

        $key = $self->_normalize( $key );

        $self->{ $key };
        }

__PACKAGE__;
```

文件句柄
Filehandles

现在你可能已经猜到我要说什么了：绑定文件句柄和其他绑定变量一样。表 17-3 显示了一些文件操作和绑定的方法的对应关系。我们只须要为需要的行为提供相应的方法。

表 17-3：一些文件句柄操作和绑定的方法的对应关系

操作	文件操作	绑定的方法
打印到文件句柄	`print FH "...";`	`PRINT( @a )`
从文件句柄读	`$line = <FH>;`	`READLINE()`
关闭文件句柄	`close FH;`	`CLOSE()`

举一个小例子，我实现了模块 `Tie::File::Timestamp`，它可以给输出的每一行添加一个时间戳。假设某个程序已经有很多 `print` 语句了，这个程序不是我写的，而我的任务是给每一行添加一个时间戳：

```
# old program
open LOG, ">>", "log.txt" or die "Could not open output.txt! $!";
```

```
print LOG "This is a line of output\n";
print LOG "This is some other line\n";
```

我们可以做很多搜索和输入来进行修改，甚至也可以使用文本编辑器完成大部分的工作。不过可能我们会漏掉一些地方——我总是对大范围的改动比较担心。通过替换文件句柄，我们只用对代码做少量修改。我们可以使用 tie 替换掉 open，程序其余的部分则完全保持不变：

```
# new program
#open LOG, ">>", "log.txt" or die "Could not open output.txt! $!";
tie *LOG, "Tie::File::Timestamp", "log.txt"
        or die "Could not open output.txt! $!";

print LOG "This is a line of output\n";
print LOG "This is some other line\n";
```

现在我们须要让整个程序能够正常工作。这非常容易，只用实现 4 个方法。我们在 TIEHANDLE 方法中打开文件。如果打开文件失败，我们就简单地返回，从而触发程序中的 die。如果打开文件成功，我们返回文件句柄的引用，并把它和我们的绑定类“bless”起来。该对象会被其余的方法作为第一个参数使用。

输出方法也非常简单。它们只是把内置的 print 和 printf 包了一层。我们把 tie 对象当作文件句柄的引用来使用（根据《Perl Best Practices》的建议，把它放在了括号中以提示用户我们的意图）。在 PRINT 中，我们给 print 添加了一些参数。添加的第一个参数是时间戳，第二个是一个空格符，以便让输出好看一些。PRINTF 中做的事情也相同，不过额外的参数放到了$format 中：

```
package Tie::File::Timestamp;
use strict;
use vars qw($VERSION);

use Carp qw(croak);

$VERSION = 0.01;

sub _timestamp { "[" . localtime() . "]" }

sub TIEHANDLE
        {
        my $class     = shift;
        my $file      = shift;

        open my( $fh ), ">> $file" or return;

        bless $fh, $class;
        }

sub PRINT
        {
        my( $self, @args ) = @_;
```

```
        print { $self } $self->_timestamp, " ", @args;
        }

sub PRINTF
        {
        my( $self, $format, @args ) = @_;

        $format = $self->_timestamp . " " . $format;

        printf { $self } $format, @args;
        }

sub CLOSE { close $_[0] }

__PACKAGE__;
```

不过，绑定变量也有一个明显的缺点：只能对文件句柄这样做。从《Learning Perl》开始，我就在告诉读者使用裸字（bareword）作为文件句柄是过去的方式，新的和更好的方式是把文件句柄的引用保存在标量中。

当我们使用标量的时候，`tie` 会查找 `TIESCALAR` 方法及其他绑定标量的方法。它不会查找 `PRINT`、`PRINTF` 及其他的输入输出方法。不过这个问题可以用一个我不太推荐的方法来解决：可以使用一个 `glob` 的引用——`*FH` 在符号表中创建一个记录。我们用 `do` 代码块把该语句包围起来以创建一个作用域并得到返回值（最后一个被执行的表达式）。如果不关掉警告的话，Perl 会告诉我们`*FH` 只被使用了一次。在 `tie` 语句中，我们把`$fh` 当作 `glob` 的引用进行反引用，这样 `tie` 就会查找 `TIEHANDLE` 方法而不是 `TIESCALAR` 方法。很可怕吧？好的。不要这样写程序！

```
my $fh = \do{ no warnings; local *FH };
my $object = tie *{$fh}, $class, $output_file;
```

总结
Summary

我们介绍了很多复杂的用 Perl 语言重新实现 Perl 的数据类型的代码。`Tie` 接口使我们可以做任何想做的事情，不过我们须要做所有的工作才能让变量完成需要的功能。这个强大的功能意味着更大的责任和更多的工作。

如果须要更多的例子，可以查看 CPAN 上的 `Tie` 模块。你可以通过阅读源代码来了解它们的功能、获取一些灵感用在你的工作中。

深入阅读
Further Reading

Teodor Zlatanov 在 2003 年 1 月的《IBM developerWorks》上发表了“Tied Variables”一文：*http://www-128.ibm.com/developerworks/linux/library/l-cptied.html*。

Phil Crow 在 Perl.com 上的“Perl Design Patterns”一文中使用绑定文件句柄用 Perl 实现了一些设计模式：*http://www.perl.com/lpt/a/2003/06/13/design1.html*。

Dave Cross 在 Perl.com 的“Changing Hash Behaviour with tie”一文中讨论了绑定哈希表：*http://www.perl.com/lpt/a/2001/09/04/tiedhash.html*。

Abhijit Menon-Sen 在 Perl.com 的“How Hashes Really Work”一文中使用绑定哈希表实现了一个很酷的字典：*http://www.perl.com/lpt/a/2002/10/01/hashes.html*。

Randal Schwartz 在 2005 年 3 月和 4 月的《Linux Magazine》的文章“Fit to be tied (Parts 1 & 2)”中讨论了绑定：*http://www.stonehenge.com/merlyn/LinuxMag/col68.html* and *http://www.stonehenge.com/merlyn/LinuxMag/col69.html*。

CPAN 上有很多和“`Tie`”有关的模块，你可以通过阅读源代码来了解它们的功能、获取一些灵感用在你的工作中。

第 18 章

以模块的形式编写程序

Modules As Programs

Perl 有很多优秀的用于创建、测试和发布模块的工具。另一方面，使用 Perl 开发独立的程序非常方便，以至于不须要使用任何其他的工具。不过，我还是希望能用开发模块的工具来开发程序，也希望能够用测试模块的方法来测试它。为了达到这个目的，我们可以把程序变成**微型模块**（modulinos）。

主要问题

The main Thing

其他的语言不像 Perl 这样善解人意。它们要求我们实现一个最外层的子程序作为程序的入口。在 C 或 Java 中，我们必须把这个子程序命名为 `main`：

```
/* hello_world.c */

#include <stdio.h>

int main (void) {
        printf( "Hello C World!\n" );

        return 0;
        }
```

而 Perl 语言能够帮助我们判断出这种情况并定义子程序 `main`。在 Perl 中，整个程序都是子程序 `main`。在默认情况下，Perl 最终会把它作为 `main` 软件包来处理。当我们执行 Perl 程序的时候，Perl 就会开始执行整个文件的代码，就像它在子程序 `main` 中一样。

模块中的大部分代码都在方法或子程序中，所以它们不会被立刻执行。它们只有在调用子程序时才会被执行。你可以试试从命令行运行你最喜欢的模块，在大部分情况下，你不会看到任何事情发生。要想验证这一点，可以使用 `perldoc` 的 `-l` 开关来找到模块所在的文件，然后执行该文件：

```
$ perldoc -l Astro::MoonPhase
/usr/local/lib/perl5/site_perl/5.8.7/Astro/MoonPhase.pm
$ perl /usr/local/lib/perl5/site_perl/5.8.7/Astro/MoonPhase.pm
```

我们可以把程序写成一个模块，然后在运行的时候再决定如何处理代码。如果把文件作为程序来运行，它就表现得像一个程序一样；如果把它作为一个模块包含在另一个文件中（可能是在一个测试套件中），它就不执行任何代码直到有代码调用它。这样我们就可以在享受独立程序的便利的同时使用模块的开发工具。

回到过去
Backing Up

我们首先得回到 Perl 的早期。我们须要实现子程序 `main`，然后在须要运行的时候运行它。为了简单起见，我们先用“Just another Perl hacker（译注 1）”（JAPH）程序作为例子，之后再逐步深入。

通常情况下，Perl 版本的“Hello World”程序非常简单。为了有趣点，我们加入了“`package main`”，并且用字符串“Just another Perl hacker”替换了“Hello World”。其实，除了告诉下一个维护者软件包的默认名字之外，这样做没有任何意义。待会我们还会使用这种技巧：

```
#!/usr/bin/perl
package main;

print "Just another Perl hacker, \n";
```

很明显，当我们运行上述程序时，该字符串会出现在输出中。不过，在这个例子中我并不想要这样。我希望该文件更像一个模块：运行该文件的时候不发生任何事情。Perl 会编译代码，但是没有任何代码可以执行。因此，我把整个程序放在一个单独的子程序中：

```
#!/usr/bin/perl
package main;

sub run {
        print "Just another Perl hacker, \n";
        }
```

除非运行子程序 `run`，否则 `print` 语句不会被执行。接下来我们须要知道的是什么时候应该运行子程序。为此，我们须要知道如何区分程序和模块。

谁在调用函数
Who's Calling?

内置的 `caller` 命令可以告诉我们当前的调用栈，这样就可以知道调用发生在程序的什么地方。程序和模块都可以调用 `caller`，并不是只有子程序才能使用它。如果把一个文件当作程序运行，并且在最外层使用 `caller`，它什么也不会返回，因为调用的位置是在最外面的一层，它是整个程序的起点。于是我们知道：当一个文件作为模块使用时 `caller` 会返回一些东西，而作为程序调用时 `caller` 什么也不会返回。利用这一点，我们可以根据调用方式的不同进行相应的操作：

译注 1：中文的意思是只是另一个 Perl 黑客。

```
#!/usr/bin/perl
package main;

run() unless caller();

sub run {
        print "Just another Perl hacker, \n";
        }
```


在把这个程序保存在文件中之前，我们须要给它取个名字。它的四不像的特点暗示我们不应该给它扩展名，不过由于之后会把它作为一个模块来使用，我们可以依据模块的命名规则使用扩展名*.pm*。这样，我们可以“use”它，Perl 可以像查找其他的模块一样查找它。不过，它究竟是**程序**还是**模块**呢？其实，它同时都是。它不是通常意义上的模块，但是，我们可以把它看成是一个微小的模块，所以我称它为微型模块（modulino）。

定义好这些术语后，我把我们的微型模块保存为 *Japh.pm* 中。它就在我的当前目录下，所以我们也须要 Perl 能够从该目录中查找模块（比如说“.”在搜索路径中）。下面我们来检查一下微型模块的行为。首先，我们把它当作一个模块使用。我们在命令行中使用-M 开关加载该模块，并通过-e 开关指定使用一个“空程序”。看来作为模块加载的时候，什么事情也没有发生：

```
$ perl -MJaph -e 0
$
```

Perl 会编译该模块，然后遍历所有能够立刻执行的语句。它会执行 `caller`，而 `caller` 会返回加载了微型模块的程序的 elements 的列表。由于返回值为真，`unless` 语句不会被触发，从而不会运行 `run()`。我们稍后还会这样做。

现在我希望把 *Japh.pm* 作为程序来运行。这一次，由于已经是在最外面一层了，`caller` 不会返回任何结果。于是，`unless` 检查失败，Perl 调用 `run()`，从而我们看到了输出结果。两种方式唯一的区别是执行文件的方式。作为模块执行时它按照模块的方式运行；作为程序执行时它按照程序的方式运行。下面是作为程序运行时的命令和输出结果：

```
$ perl Japh.pm
Just another Perl hacker,
$
```

测试程序
Testing the Program

建立好微型模块的基本框架之后，我们就可以利用它带来的好处了。由于把它作为一个模块包含在其他程序中时它不会运行，我们就可以把它加载到一个测试程序中而不运行它。也可以使用 Perl 的所有测试框架来测试它。

如果程序写得好，能够把工作分隔到小的子程序中，每个子程序只做一件事，我们就可以独立地测试每一个子程序。由于 run 子程序做的工作是打印字符，我们可以使用 Test::Output 来捕捉标准输出，并和期望的结果进行比较：

```
use Test::More tests => 2;
use Test::Output;

use_ok( 'Japh' );

stdout_is( sub{ main::run() }, "Just another Perl hacker, \n" );
```

这样，我们就可以测试程序的每一部分，直到最后把所有的部分在子程序 run() 中连起来。现在，run() 看起来更像 C 语言中的程序——所有的东西都是在 main 中按照正确的顺序调用的。

创建程序的发布包
Creating the Program Distribution

有很多创建 Perl 发布包的方法，我们在《Intermediate Perl》的第 15 章中也介绍过。对于一个已经存在的程序，我比较喜欢使用 scriptdist 程序，它可以在 CPAN 上找到（要小心，很多人都会自己写一个这样的程序）。scriptdist 程序会根据*~/.scriptdist* 的模板创建程序的发布包，因此我可以按照喜欢的方式配置我的发布包。这意味着你也可以把它变成你喜欢的样子。在这里，我们要有基本的测试并且要用文件 *Makefile.PL* 来控制整个过程，就像处理普通的模块一样。所有的东西最后都会放在一个目录下，目录的名字是程序名加上.d。除了作为临时的中转目录之外，通常情况下我不使用该目录，我会立刻把所有的东西都导入版本控制系统中。须要注意的是，我还给自己写了一个提示：须要在导入之前跳转到该目录下。我大概花了 50 到 60 次才弄明白这一点：

```
$ scriptdist Japh.pm
Home directory is /Users/brian
RC directory is /Users/brian/.scriptdist
Processing Japh.pm...
Making directory Japh.pm.d...
Making directory Japh.pm.d/t...
RC directory is /Users/brian/.scriptdist
cwd is /Users/brian/Dev/mastering_perl/trunk/Scripts/Modulinos
Checking for file [.cvsignore]... Adding file [.cvsignore]...
Checking for file [.releaserc]... Adding file [.releaserc]...
Checking for file [Changes]... Adding file [Changes]...
Checking for file [MANIFEST.SKIP]... Adding file [MANIFEST.SKIP]...
Checking for file [Makefile.PL]... Adding file [Makefile.PL]...
Checking for file [t/compile.t]... Adding file [t/compile.t]...
Checking for file [t/pod.t]... Adding file [t/pod.t]...
Checking for file [t/prereq.t]... Adding file [t/prereq.t]...
Checking for file [t/test_manifest]... Adding file [t/test_manifest]...
Adding [Japh.pm]...
Copying script...
Opening input [Japh.pm] for output [Japh.pm.d/Japh.pm]
Copied [Japh.pm] with 0 replacements
```

```
Creating MANIFEST...
-----------------------------------------------------------------
Remember to commit this directory to your source control system.
In fact, why not do that right now?  Remember, `cvs import` works
from within a directory, not above it.
-----------------------------------------------------------------
```

在 *Makefile.PL* 中，只用对安装模块的常用设置做几个小小的调整就可以让它处理程序了。我们把程序的名字放在 EXE_FILES 使用的匿名数组中，ExtUtils::MakeMaker 会自动处理剩下的事情。运行 make install 时，程序会被安装到正确的位置（也取决于 PREFIX 的设置）。命令 MAN3PODS 用于安装开发的支持文档。如果想安装一个使用手册，我们可以使用 MAN1PODS——该命令用于安装应用程序的文档：

```
WriteMakefile(
        'NAME'      => $script_name,
        'VERSION'   => '0.10',

        'EXE_FILES' =>  [ $script_name ],

        'PREREQ_PM' => {},

        'MAN1PODS'  => {
                        $script_name => "\$(INST_MAN1DIR)/$script_name.1",
                        },

        clean => { FILES => "*.bak $script_name-*" },
        );
```

使用 EXE_FILES 的一个好处是 ExtUtils::MakeMaker 会修改 shebang 行以使用运行 *Makefile.PL* 时的 perl 程序。这样我们就不用担心因 perl 的位置不同而带来的问题了。

完成基本的设置之后，我们开始进行一些简单的测试。因为可以从 scriptdist 中看到它创建的测试，这里我们忽略掉具体的细节。测试 compile.t 只是确保所有的东西都能够被编译。如果程序不能正常编译，就没有必要进行后面的测试了。测试 pod.t 检查程序的文档中可能存在的 Pod 错误（有关 Pod 的更多细节请参见第 15 章）。测试 prereq.t 检查所有的依赖关系是否已经在 Perl 程序中声明。这些测试可以发现最常见的错误（至少在我的所有发行包中是这样的）。

在开始之前，我们须要确保一切都正常工作。由于正在把程序作为模块处理，因此我们每一步都会进行测试。不过，除非把它作为程序运行，否则它不会做任何事情：

```
$ cd Japh.pm.d
$ perl Makefile.PL; make test
Checking if your kit is complete...
Looks good
Writing Makefile for Japh.pm
cp Japh.pm blib/lib/Japh.pm
cp Japh.pm blib/script/Japh.pm
```

```
/usr/local/bin/perl "-MExtUtils::MY" -e "MY->fixin(shift)" blib/script/Japh.pm
/usr/local/bin/perl "-MTest::Manifest" "-e" "run_t_manifest(0,?
'blib/lib', 'blib/arch', )"
Level is
Test::Manifest::test_harness found [t/compile.t t/pod.t t/prereq.t]
t/compile....ok
t/pod........ok
t/prereq.....ok
All tests successful.
Files=3, Tests=4,  6 wallclock secs ( 3.73 cusr +  0.48 csys =  4.21 CPU)
```

添砖加瓦
Adding to the Script

完成了所有的基础配置之后，我们可以进行进一步的开发了。由于我们是把它作为模块来处理的，因此希望添加更多的子程序来完成某些工作。这些子程序应该短小且容易测试。最好只须要把我们的微型模块包含在其他程序中就可以重用这些子程序。不管怎么说，它只是一个模块，所以为什么其他的程序不能使用它呢？

首先，我们去掉硬编码的消息。我会一步步地修改以演示微型模块的开发过程。第一件事情是把消息移到自己的子程序中。这样就把要打印的消息封装在了接口里面，之后就可以在不改变子程序 run 的情况下改变得到的消息。我们也须要能够单独地测试消息。同时，把整个程序放在自己的软件包中，并把该软件包命名为 Japh。这些有助于我们把事情分解开，以便测试或在其他程序中使用：

```
#!/usr/bin/perl

package Japh;

run() unless caller();

sub run {
        print message(), "\n";
        }

sub message {
        'Just another Perl hacker, ';
        }
```

现在，我们也可以给 *t*/目录添加其他的测试文件了。添加的第一个测试非常简单：检查能否“use”这个微型模块，并确保新定义的子程序是存在的。我们不必测试真正的消息是什么，因为很快我就会改变它的内容（注 1）：

注 1：如果喜欢测试驱动的开发模式，你只须改变本章中测试和程序的顺序就可以了。在这种模式下，你应该在改变程序之前创建新的测试。

```
# message.t
use Test::More tests => 4;

use_ok( 'Japh.pm' );

ok( defined &message );
```


现在我须要能够配置消息。当前它是英文的，不过可能有的情况下我不希望这样。怎样才能得到其他语言的消息呢？可以通过做各种复杂的工作来实现代码的全球化，不过为了简单起见，我们创建了一个包含语言的场所（locale）和相应的模板字符串的配置文件：

```
en_US "Just another %s hacker, "
eu_ES "apenas otro hacker del %s, "
fr_FR "juste un autre hacker de %s, "
de_DE "gerade ein anderer %s Hacker, "
it_IT "appena un altro hacker del %s, "
```

接下来添加一些代码来读入该文件。我们须要添加一个子程序来读入文件并返回一个数据结构，然后 message 子程序须要选择的正确模板。由于 message 现在返回的是一个模板字符串，我们须要在 run 中使用 sprintf。我们还添加了子程序 topic 来返回我这个黑客的类型。这里我就不再介绍各种可能的获取 topic 的方法了。不过你可以看到我是怎么把一个只做一件事的程序变得更加灵活的：

```
sub run
        {
        my $template = get_template();

        print message( $template ), "\n";
        }

sub message
        {
        my $template = shift;

        return sprintf $template, get_topic();
        }

sub get_topic { 'Perl' }

sub get_template { ... shown later ... }
```

可以添加更多的测试来确保新的子程序能够正常工作，同时也须要保证之前的测试还能通过。

现在我的微型模块可以处理多种语言了，并且消息是可以配置的，我对此非常满意。不过让我失望的是出现了另外一个问题。由于格式化字符串是由用户决定的，他可以做任何 printf 允许他做的事情（注 2）——这可是很多的事情。由于用于运行程序的数据是由用户提供的，我们应该打开污点检查（参见第 3 章）。不过更好的解决方法是彻底避免这个问题而不是等出了问题再在伤口外缠绷带。

我们转而使用 Template 模块，并把格式化字符串变成下面的模板：

```
en_US "Just another [% topic %] hacker, "
eu_ES "apenas otro hacker del [% topic %], "
fr_FR "juste un autre hacker de [% topic %], "
de_DE "gerade ein anderer [% topic %] Hacker, "
it_IT "Solo un altro hacker del [% topic %], "
```

在微型模块中，我们须要包含 Template 模块，并配置 Template 解析器不执行 Perl 代码。由于程序的其他部分不须要知道 message 是如何工作的，因此只用改变 message 子程序就可以了：

```
sub message {
        my $template = shift;

        require Template;

        my $tt = Template->new(
                 INCLUDE_PATH => '',
                 INTERPOLATE  => 0,
                 EVAL_PERL    => 0,
                 );

        $tt->process( \$template, { topic => get_topic() }, \ my $cooked );

        return $cooked;
        }
```

现在我须要做一些发布程序的工作了。我们的微型模块依赖于 Template 模块，所以要把它添加到依赖的列表中。这样，CPAN（或者 CPANPLUS）可以自动地检测依赖关系并在安装微型模块时安装它所依赖的模块。这也是把程序封装在发行包中的另一个好处：

```
WriteMakefile(
        ...

        'PREREQ_PM' => {
                Template => '0';
                },

        ...
        );
```

注 2：模块 Sys::Syslog 曾经有这个问题，它在 bug 报告中详细解释了当时的情况。更多的细节请参见 Dyad Security 的通告：*http://dyadsecurity.com/webmin-0001.html*。

不过，如果没有配置文件会怎么样呢？message 子程序应该依然能够工作。所以我们通过 get_template 给它提供一个默认的消息。如果打开了警告的话，我们也会收到一条警告消息：

```
sub get_template {
        my $default = "Just another [% topic %] hacker, ";

        my $file = "t/config.txt";

        unless( open my( $fh ), "<", $file ) {
                carp "Could not open '$file'";
                return $default;
                }

        my $locale = shift || 'en_US';
        while( <$fh> )
                {
                chomp;
                my( $this_locale, $template ) = m/(\S+)\s+"(.*?)"/g;

                return $template if $this_locale eq $locale;
                }

        return $default;
        }
```

现在你应该知道窍门了吧：程序中新添加的代码须要更多的测试。我再一次把这些工作留给读者去完成。

最后，我须要把程序作为一个整体来测试。我们已经分开测试了各个部分，但是它们连在一起能否正常工作呢？为了得到答案，我使用 Test::Output 模块来运行一个外部命令并捕获输出的结果。我们把得到的结果和期望的进行比较。具体如何做取决于要测试的程序。为了能够在测试文件中运行我们的程序，我把它包在一个子程序中并使用$^X 指定须要运行的 perl 程序。它会和运行测试的 perl 程序一样：

```
#!/usr/bin/perl

use File::Spec;

use Test::More 'no_plan';
use Test::Output;

my $script = File::Spec->catfile( qw(blib script Japh.pm ) );

sub run_program {
        print `$^X $script`;
        }

{ # 测试美国英语
local %ENV;
$ENV{LANG} = 'en_US';
```

```
stdout_is( \&run_program, "Just another Perl hacker, \n" );
}

{ # 测试西班牙语
local %ENV;
$ENV{LANG} = 'eu_ES';

stdout_is( \&run_program, "apenas otro hacker del Perl, \n" );
}

{ # 测试没有设置 LANG 的情况
local %ENV;
delete $ENV{LANG};

stdout_is( \&run_program, "Just another Perl hacker, \n" );
}

{ # 测试设置了无意义的 LANG 的情况
local %ENV;
$ENV{LANG} = 'blah blah';

stdout_is( \&run_program, "Just another Perl hacker, \n" );
}
```

发布程序
Distributing the Programs

创建好程序的发布包后，我们可以把它上传到 CPAN（或者任何你喜欢的地方）上供别人下载。创建 CPAN 上的档案（archive）须要做的事情和创建模块须要做的相同。首先，我们运行 `make disttest`，该命令会创建一个发布包，把它展开到一个新的目录，再运行测试。这样保证了我们上传的发布包中含有所有需要的文件并且能够正常工作（嗯，在大部分情况下）：

```
$ make disttest
```

之后，我们创建喜欢的格式的档案：

```
$ make tardist
==OR==
$ make zipdist
```

最后，我们把它传到 PAUSE 上并且对外公布消息。实际上，我使用的是 `Module::Release` 提供的 `release` 工具，它可以一步完成所有的工作。

作为 CPAN 上的模块，我们的微型模块会被 CPAN 的测试小组测试。这是一个由志愿者和自动运行的计算机组成的松散的组织，他们会测试所有的模块。不过他们不会测试程序，好在我们的微型模块并不像一个程序。

在 CPAN 上还有一个鲜为人知的被称为“脚本（scripts）”的东西。它指的是人们上传的缺乏完整的发布包支持的独立程序（注 3）。Kurt Startsinic 做了一些工作来自动地根据类别给

注 3：*http://www.cpan.org/scripts/index.html*。

这些程序建立索引。他的方法是查找程序的 Pod 文档中的“脚本类别（SCRIPT CATEGORIES）”一节（注 4）。我们也可以在文档的这一节中加上程序的类型，这样索引程序就会在下一轮检查时对它进行索引：

```
=pod SCRIPT CATEGORIES

CPAN/Administrative

=cut
```


总结
Summary

我们可以创建像模块一样的程序。这种程序整个在一个文件中（除第三方模块之外）。我们可以像运行其他的程序一样运行它，也可以像模块一样开发和测试它。我们可以同时得到两边的好处，其中包括：可测试、自动处理依赖关系和安装方便。由于这种程序是一个模块，我们也可以轻松地在其他程序中重用某个部分。

深入阅读
Futher Reading

“How a Script Becomes a Module”最早发表在 Perlmonks 上：*http://www.perlmonks.org/index.pl?node_id=396759*。

我在《The Perl Journal》的“Scripts as Modules”一文中也介绍了这种想法。虽然想法相同，我选择了一个不同的话题：“turning the RSS feed from TPJ into HTML”：*http://www.ddj.com/dept/lightlang/184416165*。

Denis Kosykh 在 2004 年夏天的《The Perl Review 1.0》上发表了“Test-Driven Development”一文：*http://www.theperlreview.com/Issues/subscribers.html*。

注 4：*http://www.cpan.org/scripts/submitting.html*。

深入阅读
Further Reading

正如我在序言中所说的，要想成为大师须要向很多人学习。即使你可以从我们的系列读物《Learning Perl》、《Intermediate Perl》和《Mastering Perl》（甚至是参加 Stonehenge 的 Perl 课程）中学习 Perl 的知识，你还是须要向别人学习。

关键的技巧是要知道应该读谁的书不读谁的书。在本章附录中，我列出了我认为对读者学习 Perl 来说很重要的人物。我并不是为我的出版商促销，其实大部分的书是来自于其他出版商的。

如果你好奇为什么我没有在本书中介绍某些话题（与此同时本书还是有着可观的重量），我可以告诉你是因为这些书更好地介绍了这些内容。

其中有些书不是关于 Perl 的。当你学习 Perl 到这个阶段的时候，你应该能够从别的话题借鉴想法用于丰富你的 Perl 技巧。不要只找那些标题中有 Perl 的书。

关于 Perl 的书
Perl Books

- Dave Cross 著的《Data Munging with Perl》（Manning 出版）
- Tim Jeness 和 Simon Cozens 合著的《Extending and Embedding Perl》（Manning 出版）
- Mark Jason Dominus 著的《Higher-Order Perl: Transforming Programs with Programs》（Morgan Kaufmann 出版）

  Nicholas Clark，perl 5.8 的日常维护者，说："不要只是买这本书，要认真读它。"Mark Jason 对 Perl 编程有着独特的视角，很大程度上是因为他在计算机语言方面有非常强大的背景。该书标题的想法来自于高阶函数，这是一种函数编程的技术，它可以通过组合已有的函数创建新的函数。这本书绝对是一本杰作，读完它之后你会以一种从未想过地方式真正去欣赏 Perl。
- Lincoln Stein 著的《Network Programming with Perl》（Addison-Wesley 出版）

当你手上拿着这本书的时候，Lincoln 的书已经是相当的老了——至少对互联网来说是如此。即便如此，这个话题自从他写了该书之后也没有多少变化。如果你已经了解套接字和 Unix 网络编程了，这本书可以帮助你把这些翻译成 Perl 的版本。如果你不知道，这本书也是一个很好的介绍材料。

- Damian Conway 著的《Object-Oriented Perl》（Manning 出版）
- Damian Conway 著的《Perl Best Practices》（O'Reilly 出版）
- Peter Scott 著的《Perl Debugged and Perl Medic》（Addison-Wesley 出版）

  Perl Scott 在它的书中展现了现实主义者对 Perl 的看法。他介绍了如何处理真实生活中的 Perl 编程问题。并且在 Perl 的编程实践方面给出了实用的建议和信息。

- Darren Chamberlain、David Cross 和 Andy Wardley 合著的《Perl Template Toolkit》（O'Reilly 出版）

  Simon Cozens 在《Advanced Perl Programming》的第 2 版中说过，所有的程序员都会经历一个创建自己的模板引擎的阶段。如果你还没有到达该阶段，你可以使用 *Template Toolkit* 直接越过该阶段。不要犹豫或回头。

- Ian Langworth 和 chromatic 的《Perl Testing: A Developer's Notebook》（O'Reilly 出版）

  虽然我们在《Learning Perl》和《Intermediate Perl》中介绍了一些 Perl 测试的内容，该书的作者更加集中和详尽地介绍了这些内容，并且还包含了有用的模块和技术。

- Tim Bunce 和 Alligator Descartes 合著的《Programming the Perl DBI》（O'Reilly 出版）

  DBI 模块是 Perl 中功能最强大、最有用的模块（在 the *Template Toolkit* 之后这么说很危险）让我惊奇的是，该模块的创造者 Tim Bunce 能够和 Alligator Descartes 写出一本又好又薄的书。

- Sam Tregar 著的《Writing Perl Modules for CPAN》（Apress 出版）

  Apress 在 Sam 告诉他们不可能从本书中赚到大钱的情况下还是出版了这本书。我必须为此赞扬 Apress 出版社。这是除了 Peter Scott 的书之外另一本介绍 Perl 的实用技巧的书。Sam 完整地介绍了创建模块、打包和维护模块的过程，并且覆盖了所有需要的非 Perl 的知识。虽然本书可以从网上免费找到，我还是建议你买一本。

与 Perl 无关的书
Non-Perl Books

- Jeffrey Freidl 著的《Mastering Regular Expressions》（O'Reilly 出版）

Jeffrey 在该书中使用了很多的 Perl，不过现在很多语言都有正则表达式，他在该书中也有所涉及。该书介绍了很多正则表达式的知识，可能远远超过了你打算知道的。介绍的内容包括各种不同的正则表达式的实现，以及它们对性能的影响。读完该书后，即使你不记得所有的这些东西，你也会有意识地改进你的正则表达式。

- Jon Bentley 著的《Programming Pearls》和《More Programming Pearls: Confessions of a Coder》（Addison-Wesley 出版）

  Perl 编程圣经的名字是《Programming Perl》并非偶然。当你读到《Communications of the Association for Computing Machinery》上 Jon Bentley 的专栏文章的时候，你的感觉一定会像读到了 Perl 语言规范的初稿一样。你可以在 *http://www.cs.bell-labs.com/cm/cs/pearls* 上看到该书的部分章节。.

- Brian W. Kernighan 和 Rob Pike 合著的《The Practice of Programming》（Addison-Wesley 出版）

- Steve McConnell 著的《Code Complete》（Microsoft 出版）

brian 的解决任何 Perl 问题的指导手册

brian's Guide to Solving Any Perl Problem

经过多年的 Perl 教学和帮助人们解决 Perl 问题，我写了一个指导手册来说明我是如何考虑问题的。很多网站都有转载，甚至还有很多的翻译版本（注 1）。有些事情是下意识做的，这些东西是最难教给新程序员的。有了这个指导手册之后，别人就可以基于它来积累自己的解决问题的技巧了。也许它不能解决你的所有问题，但是值得尝试一下。

解决问题的哲学

My Philosophy of Problem-Solving

对于编程（甚至是其他的所有事）来说，我相信 3 个东西：

与个人无关

忘记代码的所有权问题。你可能会把自己当作一个艺术家，不过即使是大师也会有很多的垃圾作品。所有人的代码都是垃圾，就是说我的代码是垃圾，你的也是。学着热爱这一点吧！当你遇到一个问题的时候，你的第一个想法应该是“我的垃圾代码出了点问题”。这就意味着你不会去怪罪 Perl。这不是个人的。

忘记你做事的方式。如果你做事情的方式是正确的，你就不会读这一段文字了。不过这并不是什么坏事。只是需要一些时间而已。所有的人都有用错误方法做事情的时候。

个人的责任感

如果你的程序有些问题，那就是你的问题。你应该尽自己的所能去解决它。记住，所有的其他人都有自己的程序，他们也有自己的问题。在向别人请教你的问题之前，自己要先尽力解决它。如果你已经老老实实地尝试了本手册中的所有方法还是无法解决它，那么你已经尽力了，应该去问问别人了。

注 1：法文版：*http://articles.mongueurs.net/traductions/guide_brian.html*; and 中文版：*http://wiki.perlchina.org/main/show/brian's%20Guide%20to%20Solving%20Any%20Perl%20Problem*.

改变自己做事情的方式

我们应该解决自己的问题以避免犯同样的错误。问题往往出在你写代码的方式上，而不是你的代码中。改变你做事情的方式，这样你的生活会更轻松。不要让 Perl 适应你，它永远不会。学着去适应 Perl。它只是一种语言，而不是一种生活的方式。

我的方法
My Method

用 strict 编译程序了吗？

如果你没有使用 `strict`，赶紧把它打开。Perl 的领袖之所以能成为领袖是因为他们使用 `strict`。`strict` 模式给他们留下了更多的时间来解决其他问题、学习新东西、上传工作正常的模块到 CPAN。

你可以在代码中使用编译指令 `strict` 打开 `strict` 模式：

```
use strict;
```

你也可以使用 `perl` 的开关`-M` 从命令行打开 `strict` 模式：

```
perl -Mstrict program.pl
```

刚开始你可能会对 `strict` 模式比较恼火，不过几周之后，你会写出更好的代码，在简单的错误上花更少的时间，甚至不再需要这个指导手册。

警告的含义是什么？

Perl 会针对很多值得商榷的用法给出警告。打开警告让 Perl 来帮助你吧。

你可以在 shebang 行使用 `perl` 的开关`-w`：

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

也可以从命令行打开警告：

```
$ perl -w program.pl
```

Lexical 的警告有各种有意思的特性。想知道细节请查阅编译指令 `warnings` 的文档：

```
use warnings;
```

如果你不理解某个警告的含义，可以在 perldiag 中查找它的详细解释，或者在代码中使用 `diagnostics` 编译指令：

```
use diagnostics;
```

先解决第一个问题！

当 perl 给出错误或警告消息的时候，先解决第一个问题，再看看 perl 是否还会给出其他的错误消息。有的时候后面的消息往往是第一个问题带来的副产品。

查看警告消息中给出的行号之前的代码！

当 Perl 对某个问题有疑问的时候，它会给出警告消息。当 `perl` 给出警告的时候，问题已经发生了，`perl` 当前所在的行很可能已经是在出现问题的代码行**之后**。检查警告消息中的行号之前的表达式。

变量的值是你想象的那样吗？

不要猜测！检查所有的东西！在表达式中使用变量之前要先检查它的值。世界上最好的调试器是 `print`：

```
print STDERR "The value is [$value]\n";
```

我把`$value`放在了括号中，这样可以看到变量前后的任何空格或换行符。对于标量之外的其他类型，我们可以使用 `Data::Dumper` 打印出数据结构：

```
require Data::Dumper;

print STDERR "The hash is ", Data::Dumper::Dumper( %hash ), "\n";
```

如果变量不是你期望的，往后退几步再试试！这样重复下去直到你找到出现问题的地方。

你也可以通过 `perl` 的`-d`开关来使用 Perl 内置的调试器。更多的细节请参考 *perldebug*：

```
perl -d program.pl
```

你还可以使用其他的调试工具或开发环境，比如 ptkdb（一个基于 Tk 的图形调试工具）或 Komodo（ActiveState 的基于 Mozilla 的 Perl 集成开发环境）。我在第 4 章中介绍了调试。

你是否正确地使用了函数？

我写 Perl 程序已经很长时间了，到现在为止几乎每天还要查看 *perlfunc*。有的时候我没法集中精力，有的时候我实在是太缺乏睡眠了，以至于失去了所有的常识开始奇怪为什么 `sprintf()` 没有把内容输出到屏幕。

你可以使用命令 `perldoc` 和`-f`开关查找函数的说明：

```
perldoc -f function_name
```

如果你在使用一个模块，请检查文档以确保使用的方式是正确的。你可以使用 `perldoc` 来查看模块的文档：

```
perldoc Module::Name
```

是否使用了正确的特殊标量？

和函数类似，我总是去查阅 *perlvar*。不过，事实上我发现《The Perl Pocket Reference》更加好用些。

你使用的是模块的正确的版本吗？

有些模块不同版本的行为不同。你的模块的版本是你想象的那样吗？你可以用简单的单行 `perl` 程序来查看安装的版本：

```
perl -MModule::Name -le 'print Module::Name->VERSION';
```

如果你是在网上阅读文档的，比如 *http://perldoc.perl.org* 或 *http://search.cpan.org*，你很有可能会在文档中看到不同版本的差别。

你写了小的测试例吗？

如果你在尝试新东西或觉得某段代码的行为很有意思，可以写个最短的可能的程序来试一试该部分。这样就可以把大部分其他的因素排除在外。如果该测试程序的行为和我们期望的一致，可能问题并不在这段代码中。否则你很可能已经找到了问题的所在。

检查环境变量

有的程序依赖于环境变量。你确定环境变量的设置正确吗？你看到的环境变量和程序运行时看到的一样吗？记住给 CGI 程序或 cron 使用的程序可能会看到和你交互的 shell 不同的环境变量，尤其是当运行的机器不同的时候。

Perl 把环境变量保存在了 `%ENV` 中。如果你须要使用某个变量，记得在变量不存在的时候提供一个默认值，即使是在测试的程序中。

如果还是有问题，检查一下环境变量：

```
require Data::Dumper;
print STDERR Data::Dumper::Dumper( \%ENV );
```

查阅 Google 了吗？

如果你遇到了一个问题，很可能别人已经遇到过这个问题了。你可以通过搜索 Google Groups（*http://groups.google.com*）看看有没有人在 Usenet group *comp.lang.perl.misc* 上讨论相似的问题。在 Usenet 上问问题的人和回答问题的人的区别在于后者能够有效地使用 Google Groups。

你剖析程序了吗？

如果你须要找出程序中运行慢的部分，你剖析过它吗？你可以让 `Devel::SmallProf` 替你完成这些脏活累活。它可以统计 `perl` 执行每行代码的次数和时间，并给出一个漂亮的报告。我在第 5 章中介绍了如何剖析程序。

哪一个测试失败了？

如果你有一个测试套件，你知道哪一个测试失败了吗？你应该能够很快找到错误，因为每个测试只会执行一小部分代码。

如果没有测试套件，为什么不创建一个呢？如果你的程序非常小或这个程序只使用一次，我不会强迫你写大量的测试。除了上述两种情况之外，写一些测试程序会有很多好处。模块 `Test::More` 让这些变得非常容易。如果你像第 18 章中那样把你的脚本写成了一个 modulino，你还可以利用所有开发模块能够使用的工具。

和你的小熊讨论过你的问题吗？

大声地解释你的问题，要说出声来。

我和一个非常棒的程序员一起工作了很多年，非常愉快。他能够解决几乎所有的问题。当我被问题卡住的时候，会走到他的办公桌旁向他解释我的问题。通常我还没有说到第三句就会说“别在意——我想出来了”。几乎每次都这样。

你可能也须要经常这样做，所以我建议你找一个毛绒玩具来当你的 Perl 心理师，这样不会惹火你的同事。我在桌子上有一个小熊，我经常向它解释我的问题。我的女朋友甚至没有注意到我在自言自语。

把问题放在纸上会不会有所不同？

你一直在盯着电脑屏幕看，也许使用一个不同的媒介会让你用新的方式去思考问题。试试把你的程序打印出来看看。

你看了 Jon Stewart 的 Daily Show 吗？

说真的。如果你不喜欢 Jon Stewart，也可以选择别的。休息一下吧。停止思考这个问题一段时间，让你的大脑放松一下。休息之后再思考这个问题，也许你会很快找到答案。

放下你的自尊

如果读到这里还没有解决你的问题，可能问题是心理上的。可能你对某一部分代码有很强的感情，所以不去改变它。你甚至可能会认为除了你之外别人都错了。当你这样考虑的时候，你就不会认真地考虑最有可能出问题的地方——你自己。不要忽略任何东西，要检查所有的地方。

索引 Index

Symbols

A

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

V

W

X

Y

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

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

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

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

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

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

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

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

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

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

作者简介

brian d foy 从 1998 年以来一直是 Stonehenge Consulting Services 的一名培训师。从读物理系的研究生开始他就是 Perl 的使用者。在 Perl 社区中他很有名气。他成立了第一个 Perl 用户讨论组——the New York Perl Mongers，以及 the Perl advocacy nonprofit Perl Mongers, Inc. 他负责维护 Perl 核心文档的 *perlfaq* 部分、CPAN 上的很多模块，以及很多独立的脚本。他是《The Perl Review》的出版者，这是一本专门讨论 Perl 的杂志。他经常在会议上发表演说。brian 也对畅销书《Learning Perl》和《Intermediate Perl》的最新版本做出了贡献。

关于封面

本书封面上的动物是小羊驼母亲和她的孩子。小羊驼生活在南美洲安第斯山脉海拔 4 000 到 5 500 米的地方。它们柔软的羊毛能够制成世界上最好的羊绒。很多世纪以来，小羊驼因此受到人们的珍爱。由小羊驼毛制成的织物贵达每码（0.914 4 米）3 000 美金。

小羊驼在古老的印加人的社会中有着特殊的地位。印加人相信这种动物是一个美丽的少女的化身。她由于屈服于一个衰老的丑陋的国王的求爱而得到了一件金子做的大衣作为回报。每隔四年，印加人都会举行一个名为“chacu”的狩猎仪式来捕获数以千记的小羊驼，剪掉它们的毛，再把它们放生。印加人的法律禁止杀死小羊驼，只有皇室成员才能穿由小羊驼毛制成的衣服。

无节制的捕猎曾经使该物种在 1974 年出现在濒危物种的名单中。当时，它们的数目缩小到了 6 000。不过，在秘鲁政府严密的监管下，该物种的数目得以回升，时至今日，小羊驼的总数已经超过了 120 000。“chacu”仪式现在受到秘鲁政府的监管，所获利润的一部分会被返还给安第斯山脉的村民。