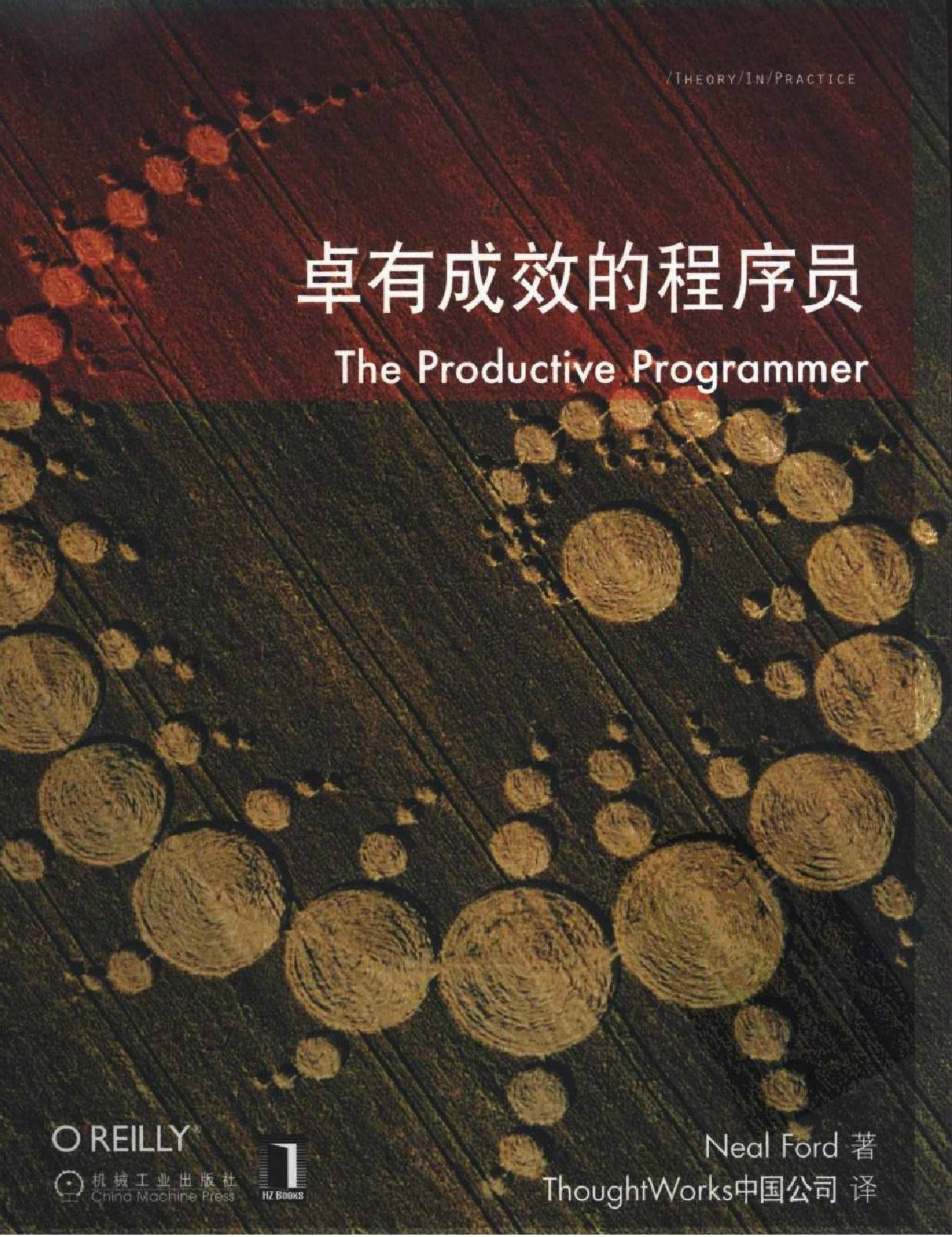
/THEORY/IN/PRACTICE

卓有成效的程序员

The Productive Programmer



O REILLY®
机械工业出版社
China Machine Press
HZ BOOKS

Neal Ford 著
ThoughtWorks中国公司 译

Software Engineering

卓有成效的程序员

任何打算以开发软件为生的人都需要一种经过实践检验的方式，来使自己的工作更好、更快、更高效。本书在“如何节省时间”方面提供了宝贵的建议和实用的工具，不论你使用什么平台都能立即从中获益。作为大师级的开发者，Neal Ford提出了大量有助于提高生产率的建议：如何更明智地工作，如何排除干扰，如何充分利用计算机，以及如何避免重复等。此外，他还详细介绍了很多有价值的实践经验，帮你回避常见的陷阱，改善代码，从而为团队创造更大的价值。

你将会学到：

- 在编写代码之前先写测试。
- 有效管理对象的生命周期。
- 只构建当前一定需要的，不构建将来可能需要的。
- 在软件开发中运用古老的哲学。
- 质疑权威，而非盲从标准。
- 借助元编程，让困难的事变容易，让不可能成为可能。
- 确保同一方法中的所有代码具有同样的抽象层面。
- 选择正确的编辑器，打造最合适的工具组合。

这些不是空谈的理论，而是Ford丰富经验的精华。不论你是刚入行的新手还是从业多年的专家，本书中这些简单而直白的原则都将对你的工作和职业生涯有所助益。

Neal Ford是ThoughtWorks的软件架构师。他曾在美国和其他国家进行现场授课，客户包括军方和很多《财富》500强的企业。

ThoughtWorks是一家全球IT咨询公司。该公司交付客户定制应用软件，提供注重实效的咨询服务，为企业开发软件，帮助企业敏捷开发。

投稿热线：(010) 88379604
购书热线：(010) 68995259，68995264
读者信箱：hzjsj@hzbook.com
华章网站：*http://www.hzbook.com*
网上购书：*www.china-pub.com*

O'REILLY®

www.oreilly.com

O'Reilly Media, Inc.授权机械工业出版社出版

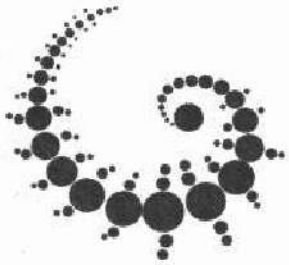
ISBN 978-7-111-26406-4

9 787111 264064

定价：45.00元

卓有成效的程序员

The Productive Programmer



Neal Ford 著

ThoughtWorks 中国公司 译

O'REILLY®

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

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

图书在版编目（CIP）数据

卓有成效的程序员 /（美）弗德（Ford, N.）著；ThoughtWorks 中国公司译 .
—北京：机械工业出版社，2009.3
（O'Reilly 精品图书系列）
书名原文：The Productive Programmer
ISBN 978-7-111-26406-4

I. 卓… II. ①弗… ②T… III. 程序设计 IV. TP311.1
中国版本图书馆 CIP 数据核字（2009）第 023187 号

北京市版权局著作权合同登记
图字：01-2009-1162 号

书　　名 / 卓有成效的程序员
书　　号 / ISBN 978-7-111-26406-4
责任编辑 / 周茂辉
封面设计 / Mark Paglietti，张健
出版发行 / 机械工业出版社
地　　址 / 北京市西城区百万庄大街 22 号（邮编 100037）
印　　刷 / 北京京师印务有限公司印刷
开　　本 / 178 毫米 × 233 毫米　16 开本　14.5 印张
版　　次 / 2009 年 3 月第 1 版　2009 年 3 月第 1 次印刷
定　　价 / 45.00 元（册）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：(010)68326294

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. 授权机械工业出版社，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，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为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。

业内好评

“对于程序员，过去我们一直习惯于用单纯的技术水平（也就是实现程序功能的能力）来衡量。然而这个时代其实已经过去了。

虽然技术仍然很重要，但企业越来越多地认识到，对于程序员更全面的衡量标准，应当是生产率。只有能够以较高的效率完成对项目、对企业有价值的工作，才是团队和组织所真正需要的人才。反之，只有技术好，但不能真正促进整体价值，甚至其反作用，这样的‘技术牛人’已经没有生存空间了。”

——孟岩

《程序员》杂志总编

“‘卓有成效是可以学习的’，让编程工作变得卓有成效也是可以学习的，方法就在这里……我会要求我们团队的所有程序员认真研读这本书。”

——黄晶

校内网高级技术总监

“如果你想做一个真正的懒人，就请读完这本书，因为这本书是天堂；如果你不想做一个真正的懒人，那也可以读完这本书，因为它至少可以教会你掌握一些耍酷的小窍门，而且要比从前那种一遍遍敲 ls 或者 dir 滚屏更加专业。”

——李剑

InfoQ 中文站敏捷社区首席编辑

“通俗不掩睿智，腕从何妨心悟——用这句话可概括我喜欢本书的原因。”

——温昱

资深咨询顾问、《软件架构设计》作者

“如果你不能持续提高自己的价值，恐怕迟早会沦为公司的鸡肋。赢得竞争，就是让自己比别人更有效率！好习惯决定高效率，同时也希望本书能够让你的代码走正确的路……Neal Ford 的这本书中有这样的一句话：‘我的使命，是让作为个体的程序员通过掌握恰当的工具和思想变得更加高效。’这本书倾注了 ThoughtWorks 公司精英们的心血，他们把自己的宝贵经验都分享了出来，那些工具和方法，定会让你受益匪浅。熊节，这位优秀的咨询师，他犀利的文字也为这本书凭添了几分独特，让人爱不释手。”

——胡铭娅

IT168 技术频道资深编辑

“有些知识，你可能需要工作和学习体验很久才能掌握，而本书作者Neal Ford已经直接把这些经验方法告诉你了……在熊节的翻译下，Neal Ford的书读起来到也比较对味，很是流畅。”

——华亮

CSDN博客专家、Tencent工程师

“一直以来，我都未曾在书店找到过一本关于代码和工程以外的、实战类的通用技巧性的书籍……如果你想在效率的竞赛中上百尺竿头更进一步，那这本书就是你需要的。”

——乔坦

.NET程序员

“程序员总有学不完的东西，许多看过我写的‘程序员的十层楼’的人觉得自己仍然是‘菜鸟’。同样，当我看到Neal的这本书时，发现自己十几年的程序员生涯仍然是一个低效的程序员，书中介绍的许多提高效率的工具和方法以前没有用过或没有用好。

要是在‘菜鸟’或‘大虾’阶段就能看到这样一本好书多好啊！不仅能及时掌握各种提高效率的工具和方法，更重要的是变成‘牛人’或‘大牛’后，它可以为设计高效的软件提供非常好的借鉴。”

——周伟明

多核编程专家

译者序

消除浪费，始于细节

在一次关于敏捷的讨论中，我说了一句令很多人不解的话：我不要敏捷。和很多话一样，断章取义地理解很容易造成误会。我当时说的整句话是：我不要敏捷，我要致力于消除软件开发中的一切浪费。当“敏捷”渐渐变成一个人见人爱的“大词”，越来越多的人开始发现，其实自己要的不是“be agile”（变得敏捷），而是切实地消除浪费、提高效率。

所以，作为ThoughtWorks员工的Neal Ford在他的这本书里闭口不谈“敏捷”。他只是实实在在地告诉你，作为一个程序员，你每天都在什么地方浪费着自己的生产率，以及如何去有效地消除这些浪费。

也许你甚至意识不到这些细小环节上浪费的存在。随便举个例子吧，在你一天的工作中，你有多少次从资源管理器里导航到源代码文件夹查看代码，然后又导航到另一个文件夹寻找文档，然后打开命令行窗口并进入项目目录，以及在密密麻麻的任务栏里找到正确的浏览器窗口？Neal Ford说，这些都是浪费：做这些与核心任务（即软件开发）无关的事情是在浪费生产率。有兴趣知道这些自己每天做无数次的事还能如何改进吗？即便不是专业程序员，本书的第2章也将对你不无裨益。

从某种意义上来说，Neal Ford在本书里做的事，正是现代科学管理理论的鼻祖弗雷德里克·泰勒在伯利恒钢铁厂做过的“泰勒实验”：剖析每个个体日常工作中的每个细节，对细节进行持续优化，通过对细节的改进提升生产率。在钢铁厂，泰勒的科学管理方法让一个搬运铁块的工人每天的工作效率提高了3倍；而在软件开发中对细节的重视甚至能让程序员的效率提升更多，因为人的体力终归有限，而脑力的开发程度还远未达到极限。

这并非痴人说梦，因为ThoughtWorks就是这样的例证。ThoughtWorks有一群天才的程序员，只有近距离接触才会发现，这些人之所以能做到如此高效，很大程度上是因为他们有一些根深蒂固的好习惯，而且不断在细节上精益求精。

ThoughtWorks中国公司的几位同事一起参与翻译本书，这也正是为了把我们的经验分享给更多人。

从注意每天的细节开始，让自己成为一个高效的程序员，其实每个人都能做到。

熊节

ThoughtWorks咨询师

2008年11月17日

作者简介

作为一名软件架构师与意见领袖，**Neal Ford** 供职于ThoughtWorks（一家专注于端到端软件开发与交付的跨国IT咨询公司）。在加入ThoughtWorks之前，Neal是The DSW Group, Ltd.的技术总监——这是一家在美国还算有名的培训与软件开发公司。Neal毕业于乔治亚州立大学，他拥有计算机科学的学位，专攻语言与编译器；同时他还辅修数学，专攻统计分析。现在他是一名软件设计师和开发者，此外也编撰培训材料、杂志文章和视频演讲，他还是几本图书的作者，包括《Developing with Delphi:Object-Oriented Techniques》（由Prentice-Hall出版）、《Jbuilder 3 Unleashed》（由Sams出版）和《Art of Java Web Development》（由Manning出版）等。他曾担任2006和2007版《No Fluff, Just Stuff文选》（Pragmatic Bookshelf）的编辑和作者。他擅长的编程语言包括Java、C#/.NET、Ruby、Groovy、函数式语言、Scheme、Object Pascal、C++和C等。他的咨询工作主要针对大规模企业应用的设计和开发。Neal曾在美国和其他国家进行现场授课，客户包括军方和很多世界500强的企业。作为演讲者，他同样在全球享有盛名，曾在世界各地举办的各种大型开发者会议上发表超过600场演讲。如果有兴趣了解更多关于Neal的信息，请访问他的网站：*http://www.nealford.com*。他也希望得到读者的反馈，他的邮件地址是*nford@thoughtworks.com*。

封面介绍

封面图像是Corbis的素材照片。

目录

序言 1
前言 3

第 1 章 概述 9

为什么要写一本关于程序员生产率的书 10
本书涵盖的内容 12
如何读本书 14

第一部分 机制

第 2 章 加速法则 17

启动面板 18
加速器 27
宏 42
小结 44

第 3 章 专注法则 45

排除干扰 46
搜索优于导航 48
找出难找的目标 50
使用有根视图 52
设好“粘性属性” 54
使用基于项目的快捷方式 55
使用多显示器 56
用虚拟桌面拆分工作空间 56
小结 58

第 4 章 自动化法则 59

不要重新发明轮子 61
建立本地缓存 61

自动访问网站 62
与RSS源交互 63
在构建之外使用Ant 64
用Rake执行常见任务 65
用Selenium浏览网页 67
用bash统计异常数 67
用Windows Power Shell替代批处理文件 69
用Mac OS X的Automator来删除过时的下载文件 70
驯服Subversion命令行 72
用Ruby编写SQL拆分工具 73
我应该把它自动化吗 74
别给牦牛剪毛 76
小结 76

第5章 规范性法则 79

DRY版本控制 80
使用标准的构建服务器 82
间接机制 83
利用虚拟平台 90
DRY阻抗失配 91
DRY文档 99
小结 105

第二部分 实践

第6章 测试驱动设计 109

不断演化的测试 111
代码覆盖率 118

第7章 静态分析 121

字节码分析 122
源代码分析 124
用Panopticode生成统计数据 125
动态语言分析 128

第 8 章 当个好公民 131
破坏封装 132
构造函数 133
静态方法 134
犯罪行为 138
第 9 章 YAGNI 141
第 10 章 古代哲人 147
亚里斯多德的“事物的本质性质和附属性质”理论 148
奥卡姆剃刀原理 149
笛米特法则 153
“古老的”软件学说 154
第 11 章 质疑权威 157
愤怒的猴子 158
连贯接口 159
反目标 162
第 12 章 元编程 163
Java 和反射 164
用 Groovy 测试 Java 166
编写连贯接口 167
元编程的归处 168
第 13 章 组合方法和 SLAP 171
组合方法实践 172
SLAP 176
第 14 章 多语言编程 181
历史与现状 182
路在何方 185
Ola 的金字塔 190
第 15 章 寻找完美工具 193
寻找完美编辑器 194
编辑器参考列表 197

选择正确的工具 198
丢弃错误的工具 204

第 16 章 结束语：继续对话 207

附录 构建块 209

序言

在我们这个行业里，不同程序员的个人生产率可谓判若云泥——大多数人也许要花一周时间才能干完的活，有些人一天之内就搞定了。这是为什么？简单来说，这些程序员比大多数同行掌握了更多趁手的工具。说得更明白一点，他们真正了解各种工具的功用，并且掌握了使用这些工具所需的思维方式。这些“卓有成效的程序员”的秘密是采用了正确的方法论与原理，而Neal在本书中准确地捕捉到了这种神秘的东西。

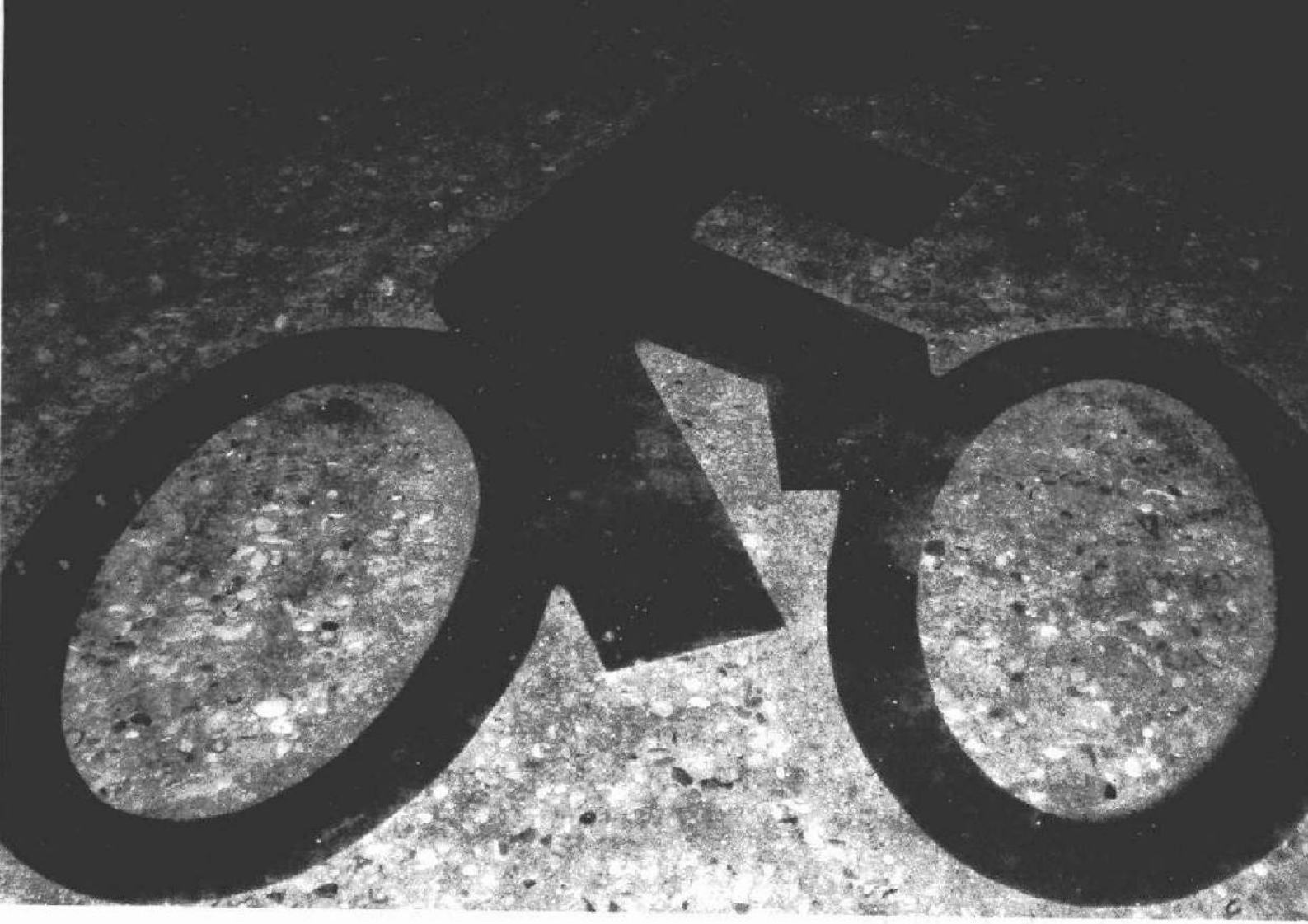
2005年的某日，在去机场的车上，Neal问我：“你认为这个世界需要再多一本关于正则表达式的书吗？”然后话题就变成了“我们希望有什么样的书”，并从此播下了你手上这本书的种子。回望自己的职业生涯中从“好程序员”跃升为“卓有成效的程序员”的那个阶段，思索当时的情景和前因后果，我这样说道：“书名我还没想好，不过副标题应该叫‘用命令行作为集成开发环境’。”那时我把自己的生产率提升归功于使用bash shell带来的加速，但这并不是全部，更重要的是我对这些工具更加熟悉，我无须思索怎么完成一些日常工作，而是自然而然地就把它们做好。我们还花了一些时间讨论过度生产（译注）以及控制这种情况的办法。几年以后，在经过无数次的私下讨论，以及围绕这个主题做了一系列演讲之后，Neal的大作终于得以付梓了。

在《Programming Perl》（O'Reilly出版）一书中，Larry Wall说到“懒惰、傲慢和缺乏耐性”是程序员的三大美德。懒惰，因为你一直致力于减少需要完成的工作总量；缺乏耐性，因为一旦让你浪费时间去做本该计算机做的事，你就会怒不可遏；还有傲慢，因为被荣誉感冲昏头的你会把程序写得让谁都挑不出毛病来。本书不会使用这几个字眼（我已经用grep检查过了），但你会发现同样的理念在本书的内容中得到了继承和发扬。

曾经有那么几本书，它们影响了我的职业生涯，甚至改变了我看待这个世界的方式。说实话，我真地希望早10年看到这本书，因为我确信它会给读者带来极其深远的影响。

David Bock
首席咨询师
CodeSherpas

译注：过度生产（hyperproductivity）是指在高效的工具和工作流程之下工作的工人得不到休息而过度疲劳、压力过大的情况。



前言

很多年前，我曾经给一些有经验的软件开发人员上课，教他们学习新的技术（例如Java等）。这些学生之间生产率的差异一直让我感到惊讶：有些人的效率能比另一些人高出几个数量级——而且这还不是指他们使用工具的过程，而是他们与计算机之间的一般交互。我曾经跟同事开玩笑说，这个班上有些人的电脑压根不是在跑（running），简直就是在散步（walking）。这自然让我开始反思自己的生产率：我有没有让手边的（正在跑或在走的）这台电脑物尽其用？

那以后又过了几年，David Bock和我谈论起这件事。很多比较年轻的同事从来就没有认真用过命令行工具，自然也就无法理解为何这些工具能比时下那些漂亮的IDE还要高效。正如David在序言里说的，我们讨论这个问题并决定要写一本关于“高效使用命令行”的书。我们联系了出版商，然后开始从朋友、同事那里搜集各种各样的命令行“巫术”。

随后又发生了几件事：David创办了他自己的咨询公司，他的孩子也呱呱坠地——三胞胎！所以，显然已经有足够多的事情让David焦头烂额了。与此同时，我也明白了一件事：一本单纯讲述命令行技巧的书很可能会成为有史以来最乏味的书。差不多就在那个时候，我在班加罗尔的一个项目里工作，和我结对编程的搭档Mujir和我聊起代码中的模式以及如何识别这些模式。如同醍醐灌顶一般，我突然意识到在自己搜集的所有技巧中都可以看到模式的踪影。我真正想要介绍的不是一堆命令行技巧，而是那些使得软件开发者们卓有成效的的模式。于是，就有了你手中的这本书。

本书的目标读者

这不是一本帮助最终用户更有效使用计算机的书。这是一本写给程序员的、关于如何提高生产率的书，这意味着我可以对读者作很多假设，很多基本概念也不需要浪费很多时间去解释，因为软件开发者是极其强大的计算机用户。当然，没有技术背景的用户也应该能够从本书（尤其是它的第一部分）中学到一些东西，但我的目标读者是软件开发者。

本书没有明确指定阅读顺序，所以尽情地随性翻阅吧，当然如果你喜欢从头读到尾也没有问题。书中的各个主题之间只有少许有意的关联，所以尽管从头读到尾的方式会略有优势，但还不足以成为阅读本书的不二法门。

体例

本书使用下列排版样式：

斜体（Italic）

用于新名词、URL、邮件地址、文件名和文件扩展名。

等宽字体（Constant width）

用于程序代码，同时也包括在段落正文中引用的程序元素，例如变量名、函数名、数据库、数据类型、环境变量、语句、关键字等。

加粗的等宽字体（**Constant width bold**）

用于命令或其他需要由用户来输入的文字。

加斜体的等宽字体（*Constant width italic*）

表示这里的文本应该替换为用户提供的值或是根据上下文决定的值。

使用代码示例

本书是为了帮助你完成工作而写的。一般而言，你可以在自己的程序或者文档中使用本书中的代码而无须征求我们的许可，除非你打算重新发布其中很大部分的代码。例如，在编程时用到书中的几段代码不需要征求许可，然而把O'Reilly书中的代码示例做成光盘销售或者发行就需要征求许可；引用本书中的代码示例来解答别人的问题不需要征求许可，然而把大量出自本书的示例代码放进你自己的产品文档就需要征求许可。

如果你在引用本书内容时注明它的版权，我们会非常感激，当然这不是必须的要求。一本书的版权信息通常包括书名、作者、出版社和ISBN编号，例如“《The Productive Programmer》 by Neal Ford. Copyright 2008 Neal Ford, 978-0-596-51978-0”。

如果你以别的方式需要使用本书中的代码示例，请通过 permissions@oreilly.com 与我们取得联系。

如何联系我们

关于本书的评论和疑问都可以发送给出版社：

美国：

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

中国：

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

我们为本书建立了一个网页，在上面列出勘误、示例以及其他附加信息。请通过下列地址访问这个网页：

http://www.oreilly.com/catalog/9780596519780（英文版）
http://www.oreilly.com.cn/book.php?bn=978-7-111-26406-4（中文版）

如果要评论或者咨询技术性问题，也可以发送邮件到：

bookquestions@oreilly.com

更多关于我们的书、会议、资源中心和O'Reilly Network的信息参见我们的网站：

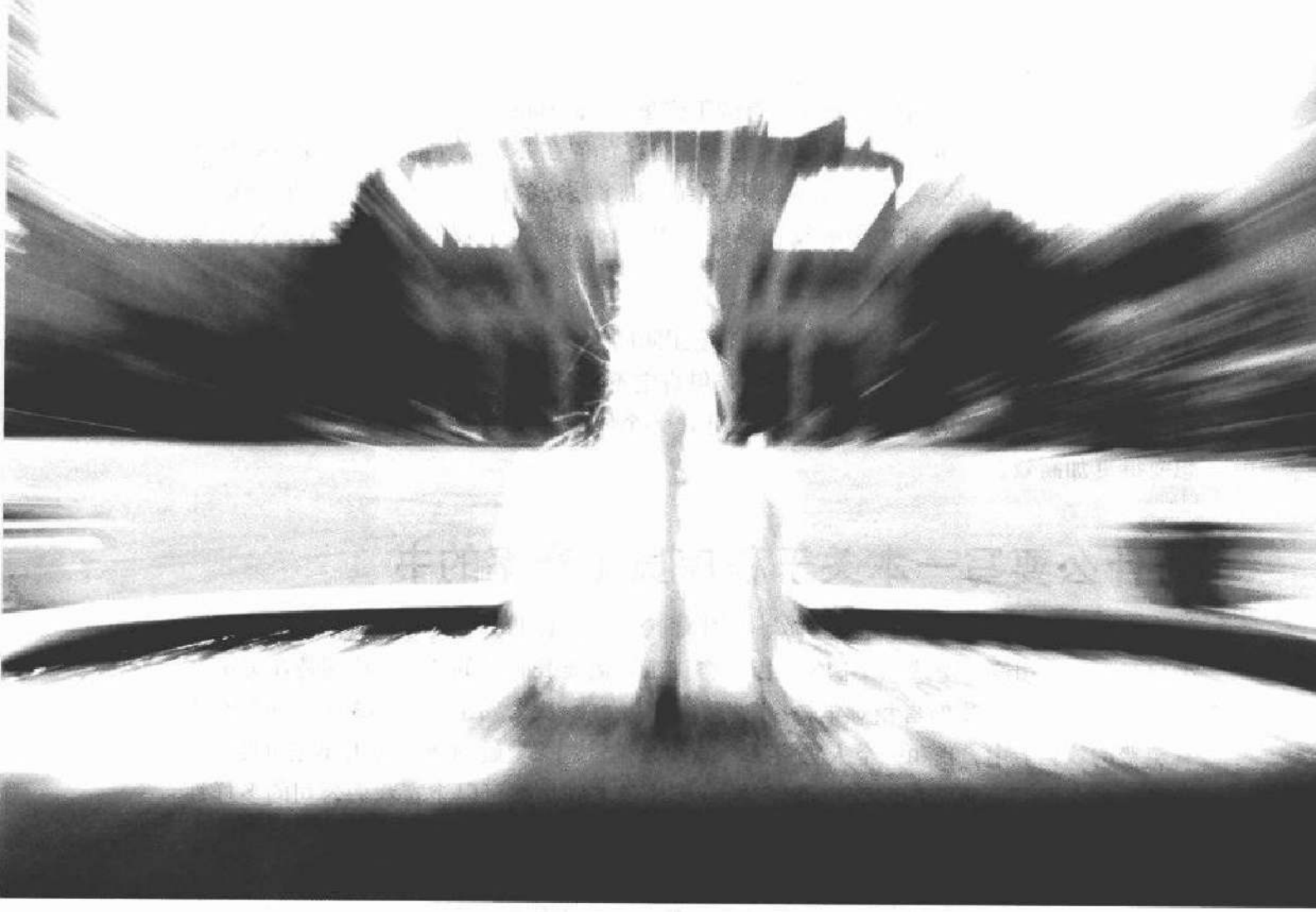
http://www.oreilly.com
http://www.oreilly.com.cn

致谢

在这整本书里，我那些非技术背景的朋友只会读这一部分，所以我最好把它弄得漂亮点。在漫长的写作过程中，我身边的亲友帮了我大忙，他们是我生命的源泉。首先，我得感谢我的家人，特别是母亲Hazel和父亲Geary，当然也少不了其他亲人，包括继母Sherrie和继父Lloyd。"No Fluff, Just Stuff"（注1）的演讲者和参与者们以及该活动的组织者Jay Zimmerman给了我很多素材，尤其是那些演讲者，他们使得飞越千山万水去参加这些研讨会是值得的。我要特别感谢ThoughtWorks的同事，能与这群人共事我深感荣幸。在加入ThoughtWorks之前，我从未见过一家公司能汇聚这样一群高智商、满怀激情、专注而又无私的人，他们一直在致力于推动软件开发方式的革命。这种企业文化至少应该部分归功于Roy Singham，ThoughtWorks的创始人。必须承认，他的个人魅力令我着迷。感谢我所有的邻居，他们不关心甚至压根不知道我写的这些技术。我要感谢Kitty Lee、Diane Coll和Jamie Coll夫妻、Betty Smith，以及所有以前和现在住在Executive Park的邻居们（没错，那也包括你，Margie）。还要特别感谢那些如今住在全球各地的朋友们：Masoud Kamali、Frank Stepan、Sebastian Meyen以及其他S&S的老伙计们。当然，还有那些只在其他国家见过的朋友，比如Michael Li，以及那些尽管住得只有五英里远，作息时间却与我难以重合的朋友，比如Terry Dietzler和他的妻子Stacy。感谢Isabella、Winston和Parker（尽管他们不会看到这些文字），他们不关心技术，但非常在意是否被关注（当然了，这是他们自己说的）。感谢我的朋友Chuck，尽管他的来访越来越少，但他的每次到访都让我感到愉快。最后，我最想感谢我了不起的妻

注1：*http://www.nofluffjuststuff.com*。

子Candy。我那些做演讲的朋友们认为她是个圣女，因为她允许我在世界各地游荡、跟人谈论和编写软件。她纵容我享受自己的工作，因为她知道我爱这工作，而且她也知道我更爱她。她一直耐心地等待我退休或者厌倦这一切，那时我就能用所有的时间来陪伴她了。



第 1 章

概述

生产率是指在一定的时间内所完成的有效工作量。在同样的时间内，生产率高的人能比生产率低的人完成更多的有效工作。本书就是讲述如何在开发软件的过程中变得更加高效。同时，本书的讲述将会跨语言和操作系统：很多技巧的讲述都会伴随多种程序语言的例子，并且会跨越三种主要的操作系统：Windows（多个版本）、Mac OS X以及*-nix （UNIX或者Linux）。

本书讨论的是程序员个体的生产率，而不是团队的生产率，所以它不会涉及方法论（好吧，可能总会在这里或那里谈论到一些，但肯定不会深入讨论）。同时，本书也不会讨论生产率对整个团队的影响。我的使命是让作为个体的程序员通过掌握恰当的工具和思想变得更加高效。

为什么要写一本关于程序员生产率的书

我是ThoughtWorks的一名员工。作为一家跨国咨询公司，ThoughtWorks拥有大约1 000名雇员，分公司遍布全球六个国家。因为咨询工作需要长时间的旅行（特别是在美国），我们公司的员工整体而言相对年轻。记得有一次，在一次公司组织的郊游活动（当然还有免费的饮料）中，我和一个人力资源部的同事闲谈起来。她问我有多大年纪，我告诉了她，她立即“恭维”地对我说道：“哇，你已经老到足够可以丰富我们公司的多样性了！”这激起了我的一些思考：原来我已经在软件开发领域干了很多年了（莫名的伤感……在我的那个年代，计算机甚至还是由煤油驱动的呢）。这些年来，我观察到一个有趣的现象：软件开发人员正在变得越来越低效，而不是更加高效。在古老的时代（对于计算机的时代而言，那是20年之前），让计算机运行起来都是一件非常困难的事情，更不要说编写程序这些事了。你得是一个足够聪明的开发人员，才能让那难以驾驭的机器变得对你有用。如此残酷的现实，逼迫当时一些非常聪明的人发明了各种各样的方法来和“难以驾驭”的计算机交互。

正是因为这些程序员的努力，计算机慢慢地开始变得易用。层出不穷的创新让计算机用户的抱怨也不再那么多。这些聪明的家伙开始为他们所取得的成就庆祝（就像所有其他能让用户“闭嘴”的程序员一样）。然后，一件有趣的事情发生了：对于整整一代程序员来说，他们不再需要“奇技淫巧”，计算机就会乖乖地满足他们的要求，他们也和普通的计算机用户一样，习惯了如今易用的计算机。那这有什么问题呢？毕竟，你不会拒绝提高生产率，对不对？

其实问题的关键在于，那些对普通用户而言能提高其生产率的东西（比如漂亮的图形界面、鼠标、下拉菜单等），对于其他一些人（程序开发者们）来说却是他们获得计算机最佳性能的障碍。“易用”和“高效”在很多时候其实是不相关的。那些在使用图形界

面（好吧，直截了当地说，就是Windows）的过程中长大的程序开发者们，对那些老一代“聪明人”所使用的不仅酷而且高效的技巧一无所知。他们的计算机在大部分时间里根本不是在跑，简直就是在散步。我写此书，就是为了解决这个问题。

浏览器的地址补全

在这里我举一个简单的例子：你每天会访问多少网站？我们知道大多数网址都以“www.”开头并以“.com”结束。但很少人知道现在的浏览器有一个很方便的快捷键：地址补全（address completion）。地址补全使用热键组合，来为你在浏览器地址栏中输入的字符串前后分别加上“www.”和“.com”。地址补全功能在不同浏览器中的使用会有细微的差别。（注意，地址补全和浏览器的自动补全是不同的，现今的浏览器也都有自动补全功能。）它们之间的差别在于效率。自动补全功能会到网络中寻找与你输入的名字相符的站点。如果没有找到，浏览器会为它加上前缀和后缀，再次尝试到网络中寻找。如果网速够快，你可能根本注意不到这个微小的延迟，但这些错误的点击累积起来会影响整个网络的速度。

Internet Explorer

Internet Explorer（IE）的地址补全功能，会使输入有标准前缀和后缀的网址变得更加快捷。使用快捷键Ctrl-Enter，浏览器中的地址就会分别在前后加上“www.”和“.com”。

Firefox

Windows版的Firefox浏览器，它的快捷键跟在IE中一样。而在Macintosh上，快捷键是Apple-Enter。Firefox还有一个快捷键Alt-Enter，它会给地址加上“.org”后缀，这个快捷键在所有支持Firefox的平台上都一样。

Firefox还有其他一些似乎很少有人用到，但却很方便的快捷键。比如在Windows下使用快捷键Ctrl + <标签号>或者在OS X下使用快捷键：Apple + <标签号>，就可以直接跳到某个标签。

好吧，用上这个快捷键，无非在每个页面上少敲8下键盘而已，看起来似乎没有太多价值。但是，想想看你一天要访问多少页面，这每个页面上的8次击键就会体现出它的价值。这是加速法则的一个很好的例子，我们将会在第2章中详细介绍这个法则。

当然，节省每个页面上的8次击键，并不是谈论这个例子的真正目的。我曾经在所有我认识的开发人员中做过一个非正式的调查，得到的结果是只有20%的人知道这个快捷

键。他们都是名副其实的计算机专家，但他们从来没有使用过这些非常简单的方法来提高他们的生产率。我的使命，就是改变这样的现状。

本书涵盖的内容

本书包含两部分。第一部分讨论生产率的机制，以及一些能使你在软件开发过程中变得更加高效的工具。第二部分讨论一些提高生产率的实践，以及如何利用你的知识和他人的知识来更快更好地开发软件。在这两部分中，你都可能会看到一些你已经了解的或者从未接触的东西。

第一部分：机制（生产率法则）

你可以认为此书仅仅是教你如何使用命令行以及另外一些能让你变得更加高效的方法的书籍，你确实也可以通过学习这些方法获得收益。但是，如果你能通过阅读本书理解为什么有些方法可以提高生产率，你就拥有了一双能识别这些方法的慧眼。本书通过创造模式来描述一些东西，从而为这些事物命名：一旦一件东西有了名字，当再次看到它时就会更容易认出来。本书的其中一个目标是定义一系列的生产率法则，来帮助你更好地定义自己的提高生产率的技术。就像所有的模式一样，在有了名字之后，就能更简单地识别出来。同样，知道为什么一些东西会让你更加高效，会帮助你更快地识别出其他一些能使你工作得更有效率的东西。

本书会更专注于讨论程序员的生产率，而不仅仅讨论如何更加有效地使用计算机（虽然肯定会有这个“副作用”）。所以，本书不会涉及很多对于业余（甚至专业）用户来说显而易见的内容（虽然之前的“浏览器地址补全”一节介绍了一个显而易见的技巧，但那是个例外）。程序员是一群特殊的计算机用户。我们显然应该比其他用户更懂得如何让计算机更加高效地工作，因为我们是最了解计算机工作原理的人。本书的大多数内容，讲述的是如何通过与计算机交互而使工作更加简单、快速和高效。不过我也会介绍一些唾手可得的、提高生产率的办法。

第一部分的内容涵盖了我所创造的、获取的，以及通过“威逼利诱”从朋友那里得到的，或者从书本里读到的所有能提高生产率的方法。我的初衷是致力于做一个世界上最好的提高生产率的方法大全。我不知道我是否做到了，但无论如何，你会发现我所收集的这些方法不会让你失望。

当我开始整理这些绝妙的提高生产率的方法时，我注意到一个个模式开始显现。看着这些技术，我开始为它们分门别类。最终，我建立了一些关于程序员生产率的“法则”

——坦白地说，我想不出比这更自命不凡的名字了。这些法则包括加速法则、专注法则、自动化法则以及规范性法则，它们描述了一些能让程序员更加高效的实践。

第2章，加速法则，描述了如何通过提高速度来变得更加高效。很明显，如果一个人在完成某件任务时要比另外一个人速度更快，那么在这件事情上，他肯定比另外一个人更加高效。很显然的一个例子就是本书会经常提到的数目繁多的键盘快捷键。启动应用程序、管理剪贴板以及搜索和导航等，都属于加速法则的范畴。

第3章，专注法则，描述了如何通过利用工具和环境的因素，来达到超级生产率的状态。该章讨论了如何减少环境中（包括物理环境和心理环境）混乱，如何有效地去搜索，以及如何消除干扰。

通过让计算机为你完成一些额外的工作，当然会使你更加高效。第4章，自动化法则，描述的就是如何让计算机为你做更多的工作。每天的工作中要做的很多事情其实都可以（而且应该）让计算机自动为你完成。该章还包括了一些让计算机为你工作的例子和策略。

规范性（canonicality）其实只是给DRY（Don't Repeat Yourself）法则起的一个漂亮名字，后者第一次被大力推崇是在Andy Hunt和Dave Thomas的《The Pragmatic Programmer》（Addison-Wesley）中。DRY法则建议程序员去除重复存在的信息，为每个信息创建唯一的存放处。《The Pragmatic Programmer》已经很形象地描述了这个法则，在第5章中，我会列举几个具体的例子。

第二部分：实践（方法）

作为一个咨询师，我有很多年软件开发的经验。相较于经常会在同一个代码库上工作多年的开发人员，咨询师的优势是可以接触很多不同的项目、不同的方法。当然，我们看到的经常是一些“火车残骸”（很少有咨询师会被叫去“修复”健康的项目）。我们还可以接触软件开发的不同阶段：从开始就参与构建，或者在中途提供一些建议，或者在一个项目遭受重创之后才去拯救。随着阅历增长，即使是最不专心的人，也会锻炼出察觉对错的能力。

我在这些年中看到了很多对提高生产率有正面或负面影响的事情，在第二部分中，我为它们做了一些提炼。它们基本上被无序地打包在一起（虽然你可能会经常惊讶地发现，同一个想法会在不同的地方出现）。我没有说这些东西应成为“如何提高开发效率”的最终纲领，它们只是我所观察到的一些事情的一个列表。对于所有的可能性而言，这只是其中一个很小的子集。

如何读本书

本书的两部分是相互独立的，所以你可以按任何顺序去阅读。虽然第二部分的内容因为有点类似于讲故事，可能会存在一些无意的相互联系，但是大部分的内容还是无序的，你仍旧可以放心地按照自己所喜欢的顺序去读。

最后给出一个小小的警示：如果对一些基本的命令行操作（管道、重定向等）不够熟悉，你应该先阅读一下附录。第一部分中的很多窍门和技巧需要依赖于一个环境，附录会指导你如何去搭建这个环境。我相信对你来说这不会是件难事。

第一部分

机制

第一部分“机制”探讨了生产率的——机制。这一部分里介绍的未必都是软件开发工具，很多是能够给任何熟练的高级用户带来帮助的工具。当然了，软件开发者应该是最高级的计算机用户，这本书里列举的工具他们几乎都能用得上。

第 2 章

加速法则

使用计算机有许多繁文缛节。首先你需要启动它，然后你得知道如何去加载应用程序，而且还必须了解应用程序的交互模式，而不同应用的交互模式又会有差异。与你的计算机交互越少，你就能前进得越快。换句话说，去除繁文缛节使你有更多的时间来针对问题的实质。比如说，与其把时间浪费在一个庞大的文件系统层次结构中东翻西找，不如做一些更有建设性的工作。计算机只是工具，你花越多的时间来关注工具本身，你能完成的工作就越少。科幻小说作者道格拉斯·亚当斯有句名言："我们真正需要的只是已经能工作的东西，但我们能得到的却只是技术。"

> **提示：**
> 关注本质，而非形式。

本章提供一些方法来加速你与计算机的交互。这可以是更快地加载应用程序，更迅速地找到文件，或者是更少地使用鼠标。

启动面板

看看你计算机中的应用程序列表。如果你使用的是Windows，点击"开始"->"程序"。你得到了多少列？两列？三列？甚至是四列？！随着硬盘容量的增大，应用程序种类（我们必须使用的工具）越来越多，我们使用的应用程序数量爆炸式增长。当然，有了常见的100GB硬盘，我们可以把许多东西都装进系统中。但这需要付出代价。

> **提示：**
> 一个应用程序列表的有用程度与它的长度成反比。

列表越长，它就会变得越没用。在Windows中有三列应用程序，或者在Mac OS X中用一个快捷工具栏（Dock）将所有条目挤成显微镜可见的大小，我们要在其中找到所需要的内容就会越来越困难。这对于开发人员尤其费劲，因为我们有许多偶然一用的应用程序：有些特殊用途的工具可能一个月中只有一天会用到，但当那一天到来时我们又会迫切需要它。

加载器

加载器（Launcher）是一类应用程序，允许你输入应用程序（或文档）名称的第一部分来加载它。通常，这是一种更高效地加载应用程序的方式。

提示：

华而不实的东西中看不中用。

如果你知道所找内容的名称（比如应用程序的名字），就应该直接告诉计算机你要什么，而不是在大块列表中或是在一堆图标中来搜索它。加载器能够跳过华而不实的图形化界面，准确而快速找到需要的内容。

所有的主流操作系统都有开源和免费的加载器，允许你输入想要加载的应用程序名称（或者名称的一部分）。其中一些值得一试：Launchy（注 1）Colibri（注 2）以及 Enso（注 3）。Launchy 和 Colibri 都开源（因此也免费），两者都允许你打开一个小窗口，你输入应用程序名称，它就会弹出一个应用列表。Launchy 是现在最流行的开源加载器。Colibri 则尝试模仿一个名为“Quicksilver”的 Mac OS X 工具（这将在随后的“Mac OS X”一节中来介绍）。

Enso 加载器有一些有趣的附加特性。它也是免费的（但不开源），由 Humanized 公司开发。该公司是 Jef Raskin（早期的 Mac 系统用户界面设计师之一）所创立的。Enso 中融入了许多他自己的用户界面观点——有时有点偏激，但很有效。比如，Raskin 提出的想法之一是 Quasimode 键，它的行为类似于 Shift 键（也就是说，在按下去之后改变键盘模式）。Enso 取代了非常没有使用价值的 Caps Lock 键，将它用于启动应用程序并执行其他任务。你按下 Caps Lock 键并输入一个命令，比如“OPEN FIREFOX”，它就会打开 Firefox。当然，这样输入很麻烦，所以 Enso 提供另外一个命令“LEARN AS FF FIREFOX”，它告诉 Enso，FF 命令将加载 Firefox。Enso 所做的不仅仅是加载。如果在一篇文档中有像“4 + 3”这样的表达式，你可以高亮选中这部分内容并调用“CALCULATE”命令，Enso 就会用计算值替换高亮部分的正文。Enso 值得一试，看看 Raskin 对于应用程序加载的观点是否和你的一致。

如果你正在使用 Windows Vista，它已经包括了这些加载器程序的加载功能。点击“开始”按钮，在紧接着出现的菜单下方有一个搜索框，允许你输入要使用的应用程序名称，并进行增量式搜索。但它有上述加载器所没有的一个缺点（也许是一个 bug）：如果你输入的内容不存在，Windows Vista 会花很长时间才能返回，并告诉你找不到。当这种事情发生的时候，你的机器就跟一块砖头没多大区别。我们希望这只是当前版本的一个怪异行为，不久就会被解决。

注 1：从 *http://www.launchy.net* 下载。

注 2：从 *http://colibri.leetspeak.org* 下载。

注 3：从 *http://www.humanized.com* 下载。

创建一个 Windows 启动面板

你可以很容易地利用Windows中的文件夹结构来创建你自己的启动面板。在“开始”按钮之下创建一个文件夹，其中包含你每天都会用到的应用程序快捷方式。你可以将这个文件夹命名为“跳转”（jump）并用“j”作为快捷键，这样你就可以键入Windows-J来访问它。这样一个“跳转”窗口的示例如图2-1。要注意的是，其中每一个菜单项都以该文件夹范围内唯一的单个字母作为前缀，这样有利于应用程序的快速加载。加载文件夹中的每一个应用程序只需两个按键：Windows-J[<唯一字母>]便可加载该应用程序。

这仅是一个目录，因此你可以将其他目录嵌套在其中来构建自己的应用程序微型层次结构。对于我的大部分高负荷机器，26项还不够，所以我倾向于创建一个dev加载文件夹来容纳所有的开发工具。我是有意使用一种层次结构来替换另一种，但两者有一点很大的不同：不像Windows中的程序分组，我对这个组织结构有着完全的控制权。我会定期重组这个文件夹，当一些应用程序失宠时，新的应用程序便会取代他们的位置。

图2-1：定制加载器窗口

创建一个加载文件夹很简单，但它依赖于你使用的“开始”按钮的风格。Windows XP和Vista支持两种“开始”配置：“经典”（从Windows 95到2000所使用的风格）与“现代”（Windows XP与Vista）。对于“经典”Windows，创建加载文件夹相当简单。右键“开始”按钮，选择“打开”（如果你只是想为当前登录的用户添加一个加载菜单）或者“打开所有用户”（针对所有用户做修改）。这会打开控制“开始”菜单内容的底层文件夹，你可以在这里为你喜欢的内容添加快捷键。或者，你可以去“开始”菜单的物理位置——在当前用户的“Documents and Settings”目录结构下。有种简便的方法可以只

将你一直在用的东西添加到加载菜单：从庞大的“程序”菜单中将它们右键拖放到加载文件夹中，创建一份快捷方式的副本。

如果你使用“现代”Windows“开始”菜单，创建一个加载菜单会有点困难，但仍然是可行的。你可以在与上述相同的目录下创建一个加载文件夹，但由于某些奇特的原因，它不会在你按下Windows键时立即出现；而仅当你展开“程序”分组时才出现。这一点很烦人，因为此时，我们用于加载的加速方法又需要一个额外的按键；然而，有办法可以解决这个问题。如果你在桌面上创建了“跳转”文件夹并将它拖放到“开始”菜单，便会创建一个立即显示的文件夹。“现代”版“开始”菜单仅剩的让人头疼的是热键问题。在“现代”式菜单中，不同的应用程序会根据使用情况移进移出（Windows将你留心记住的路径随机化以快速查找内容）。所以，如果你使用“现代”“开始”菜单，你应该为加载菜单选择一个不会冲突的首字母，比如“~”或者“(”键。

因为如此麻烦，所以我倾向于使用“开始”菜单的“经典”版本。你可以通过任务栏属性将Windows XP或Vista的“现代”模式改成“经典”模式，如图2-2所示。

重写Windows中的特殊文件夹

微软发布了称为“PowerToys（注4）”的一组工具集（但并不为其提供支持），其中包括Tweak UI，它允许你通过图形界面的方式来修改Windows注册表。“我的文档”通常位于这个很繁琐的路径下：*C:\Documents and Settings\<你的登录名>\My Documents*（在Windows Vista中仁慈地改成了“Documents”，直接位于根目录下）。Tweak UI允许你修改“我的文档”的默认位置，这样你就可以将它放到一个更合理的位置“*C:\Documents*（Windows Vista默认将你的文档存放在这里）”。

但是，小心！如果你要转移“我的文档”，请在安装好操作系统之后尽快做。许多Windows应用会依赖于这个目录中的内容。如果你在已经安装好的Windows机器中这样做，接着就会有许多应用程序被破坏。

如果你不想这么做，也可以选择一个文件夹（比如“我的文档”），右键得到属性对话框，然后告诉Windows修改它的位置。这样“我的文档”中的所有文件就会被复制到新的位置。你也可以使用古老的“subst”命令（用一个文件夹替换另一个），但它会破坏许多应用程序，所以请小心使用。Junction是一个真正能够用一个目录来代替另一个目录的工具，如果你使用NTFS文件系统，它会工作得更好。更多细节请参见第5章的“间接机制”一节。

注4：　从*http://www.microsoft.com/windowsxp/downloads/powertoys/xppowertoys.mspx*下载。

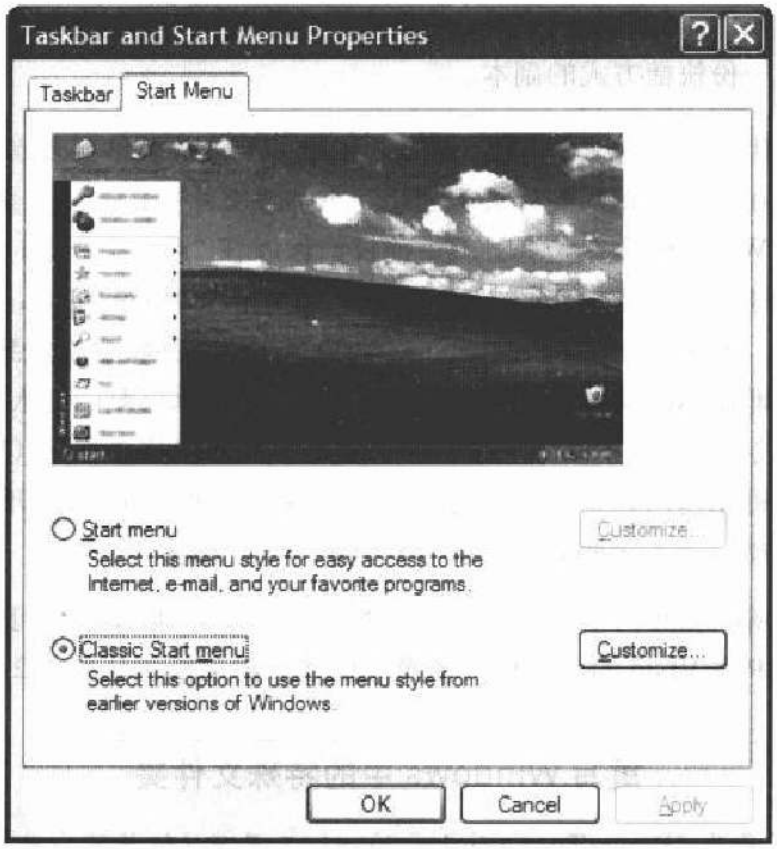


图 2-2：改回经典的开始菜单

Windows确实有一个加载少量几个应用程序的便捷机制：快速启动栏。这是出现在任务栏上的快捷方式区域，通常在“开始”按钮旁边。如果你没有看见它，需要右击任务栏并选择“快速启动栏”来开启该功能。你可以将快捷方式拖放到这里并作为加载器来使用。而且，由于这是一个目录（与其他所有地方一样），你可以直接将内容放在快速启动文件夹中。就像所有其他的快捷方式一样，你也可以为这些快速启动项指定操作系统级别的快捷键，不过应用程序的快捷键会覆盖它们。

提示：

键盘输入总比导航快。

Windows Vista有一个使用快速启动栏的新技巧。你可以通过Windows-<数字>键来运行与快捷方式关联的应用程序。也就是说，Windows-1选择并按下第一个快速启动项，Windows-2加载第二项，如此类推。这个机制可以工作得很好……只要你经常使用的应用不超过10个！虽然它不能像真正的加载器一样容纳那么多的应用，把一些非常重要的应用放在这里还是挺趁手的。

为什么不只是为你最爱的应用程序指定热键？

所有主流操作系统都允许你创建快捷键（也就是热键）来加载应用程序。既然这样，为什么不干脆直接定义一系列快捷键来完成所有的这些加载工作呢？如果你总是将桌面作为当前焦点，那这种方式会工作得很好。但开发人员几乎从来都不会只有桌面（或资源管理器）是打开的。典型情况下，一个开发人员会有20个特定用途的工具同时打开，每一个都有它自己的神奇键盘组合。试图使用操作系统热键来加载应用程序会加剧这种“巴别塔”（译注1）效应。想找到不会影响当前打开着的应用程序的热键几乎是不可能的。尽管使用操作系统级热键听起来是一个很吸引人的主意，但在实际使用的时候却会失败。

Mac OS X

Mac OS X的dock结合了Windows中快速启动菜单和任务栏按钮的功能。它鼓励你把经常要用的应用放在dock上，并将其他的拖出来（在它们消失的时候会发出一声动听的“噗”声）。和快速启动栏一样，空间有限的制约会让你用得不爽：把有用的应用程序都放进dock会使它膨胀以至变得臃肿。这造就了一个小规模的Mac OS X加载器行业。尽管有些知名的加载器已经存在很多年了，但现在较专业的用户大多都在使用Quicksilver。

Quicksilver（注5）是历来各种把命令行图形化的尝试里最出色的一个。就像bash提示符一样，Quicksilver允许加载应用，进行文件维护，并支持其他行为。Quicksilver本身是以浮动窗口的形式出现，通过可定制的热键来启动（Quicksilver的所有东西都是可定制的，包括浮动窗口的感观）。浮动窗口一出现，你就可以在目标框中执行各种行为。

获得 Quicksilver

目前，Quicksilver是免费的，可以从*http://quicksilver.blacktree.com*下载。Quicksilver的创建者正积极鼓励开发人员为Quicksilver开发更多的插件。现在已有Subversion、PathFinder、Automator及许多其他应用的插件，包括核心操作系统和第三方应用的插件。

译注1：“巴别塔”（Tower of Babel）是圣经中一座通天高塔的名字。后来成为混乱和语言不通的代名词。

注5：从*http://quicksilver.blacktree.com/*下载。

> 使用Quicksilver绝对会上瘾。相比其他的软件，它从根本上改变了我与计算机交互的方式。Quicksilver代表了软件最弥足珍贵的品质：简单而优雅。在你第一眼看到它的时候，你会想："没什么了不起的，只不过是加载应用程序的一种新方式"。然而，你用得越多，就会看到越多的微妙之处，越来越多的功能会自己逐步揭示出来。

Quicksilver通过一个热键和一对按键来简化应用程序的加载。在Quicksilver中有三个框：最上端的是文件或应用（名词），中间的是动作（动词），而第三个（如果需要的话）是动作的目标对象（直接宾语）。当你在Quicksilver中搜索项目时，它会将你键入的任何内容都视为带有通配符一样来对待。例如，如果你在Quicksilver中键入"shcmem"，它会为你找到名为ShoppingCartMemento.java的文件。

Quicksilver不仅可以加载应用程序。它允许你将任意（上下文相关的）命令应用于任意文件。在图2-3中，我选择一个名为acceleration_quicksilver_regex.tiff的文件，并指定"Move To...（移动到）"的动作。第三个框允许我选择移动的目标位置，我用同样的方式在目标框中选择了文件名（就是说，使用之前描述过的特定的通配符行为）。

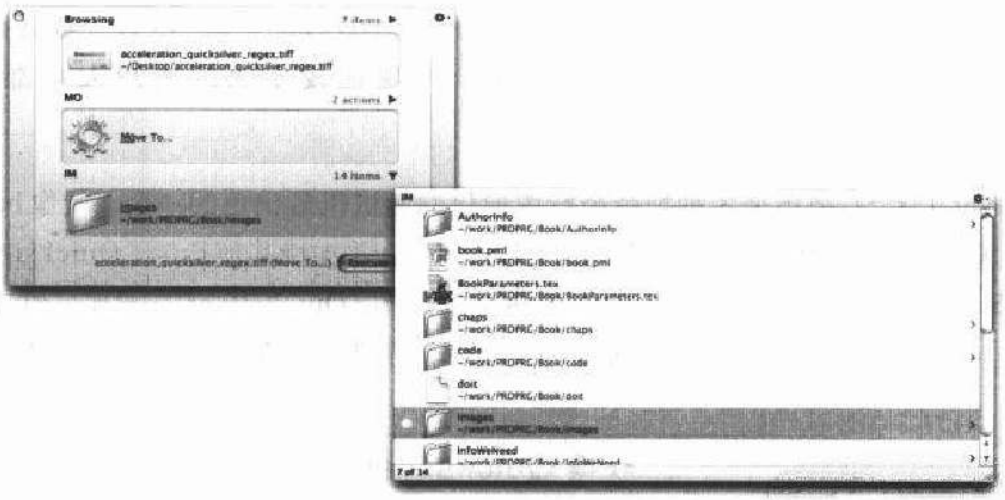


图2-3：Quicksilver的第三个框指出移动的目标

为什么这对开发人员如此重要？Quicksilver通过插件来工作，而有相当数量的以开发人员为中心的插件存在。比如，Quicksilver具有优良的Subversion整合功能。你可以更新代码库，提交修改，获取状态，还可以做很多别的事情。尽管不如命令行Subversion那么强大（没有什么工具能真正与其媲美），但它提供一种便捷的图形化方式，只需几个按键就可以完成日常杂务。

关于Quicksilver还有一点值得一提：触发器（trigger）。一个触发器是一个主－谓－宾的组合，就好像你通过常规的用户界面所做的事情一样，它永久存储在一个热键之下。比如，我有一些项目每次都会使用相同的键盘操作序列：

1. 启动Quicksilver。
2. 选择项目目录（名词）。
3. 选择“用...打开”动作（动词）。
4. 选择TextMate作为应用程序（直接宾语）。

我经常做这件事，所以我为此指定一个触发器。现在，只需要一个单独的按键（我使用Alt-1），我就可以调用这一命令序列。触发器的设计目的是允许你将常用的Quicksilver操作保存为一个单独的热键。我还用触发器来做一些如启动和停止Servlet引擎（比如Tomcat和Jetty）这样的事情。这的确非常有用。

我对Quicksilver强大的功能只是浅尝辄止。你可以加载应用程序，将命令作用到一个或多个文件上，在iTunes中转换歌曲，以及许多其他的事情。它改变了你使用操作系统的方式。有了Quicksilver，你就可以只把dock用作任务管理器来显示当前正在运行的应用程序，而把Quicksilver作为加载器来使用。Quicksilver的定制可以通过一个公开的插件API来实现（关于如何下载Quicksilver及其插件的相关信息请见前面的“获得Quicksilver”部分）。

Quicksilver是一个很好的例子，这类应用在第一次安装时会看起来过于简单而没有什么用处。许多朋友都跟我说“我装好Quicksilver了，现在我该干嘛？”所以后来，一些朋友和我创建了一个围绕着Mac上生产率的博客，称为PragMactic-OSXer（*http://pragmactic-osxer.blogspot.com*）。

为什么Spotlight还不够

Quicksilver的功能与Mac OS X中内置的搜索工具Spotlight有重叠。但Quicksilver不只用于快速搜索。它可以从根本上代替Mac OS X的Finder，因为所有典型的文件操作在Quicksilver中都能够更快地完成（键盘输入总比导航快）。Quicksilver允许你指定想要检索的条目（Spotlight则会索引整个硬盘），这使得Quicksilver通过自己的索引来查找文件时更快。而且，Quicksilver使用了很酷的“在每个字符之间使用正则表达式”的方式来指定搜索项，这是Spotlight所没有的。我几乎再也没有用过Finder了。所有的文件操作（甚至几乎所有与计算机的交互）都通过Quicksilver来进行。我变得

如此依赖它，以致于如果它不能工作（这很少发生，但确有发生；毕竟，它还只是Beta版本），就好像我的整个机器突然废掉了。相比我用过的其他工具，它对我工作方式的改变是最大的。

Leopard的Spotlight版本要比以前版本快很多，但是当然，这两个工具并不相互冲突。在Leopard的Spotlight版本中，你可以跨多台机器进行搜索（这是Quicksilver不会做的）。为了使用该功能，你必须登录到另一台机器（因为很显然的安全原因）。现在，当你再执行一个Spotlight搜索时，你可以在工具栏中选择你想搜索哪台机器。在图2-4所示的例子中，我从自己的笔记本登录到桌面电脑（称为*Neal-office*）上并选择了home目录（叫做*nealford*）。当我进行Spotlight搜索时，我可以在顶端的工具栏中选择搜索目标。文件*music.rb*只在我的桌面电脑上才有。

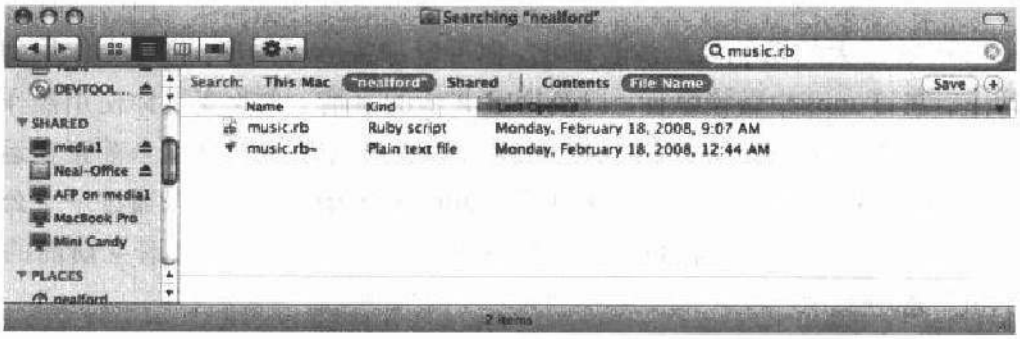


图2-4：Leopard上的Spotlight允许跨机搜索

遗憾的是，在Windows和Linux世界中不存在一个像Quicksilver这样卓越的工具。前面提到的Colibri实现了Quicksilver的一小部分功能（主要围绕它的加载功能，但不包括图形化的命令行部分）。希望有人最终会将Quicksilver移植到其他平台，或者开发一个可以媲美的克隆版本。Quicksilver是迄今为止所有操作系统中最为精妙的加载器。

在Linux中加载

大多数桌面Linux系统都运行GNOME或KDE。两者都有着从Windows借鉴而来的任务栏风格用户界面。然而，定制它们的开始选项要困难得多，因为它们的菜单结构不是以简单的目录项存在。GNOME的现代版本包括了一个功能强大的加载器，默认与Alt-F2按键相关联。它会显示一系列可运行的应用程序，并允许你通过键入字母来精简可选项。在图2-5中，我们输入“fi”两个字母就使列表缩小了。

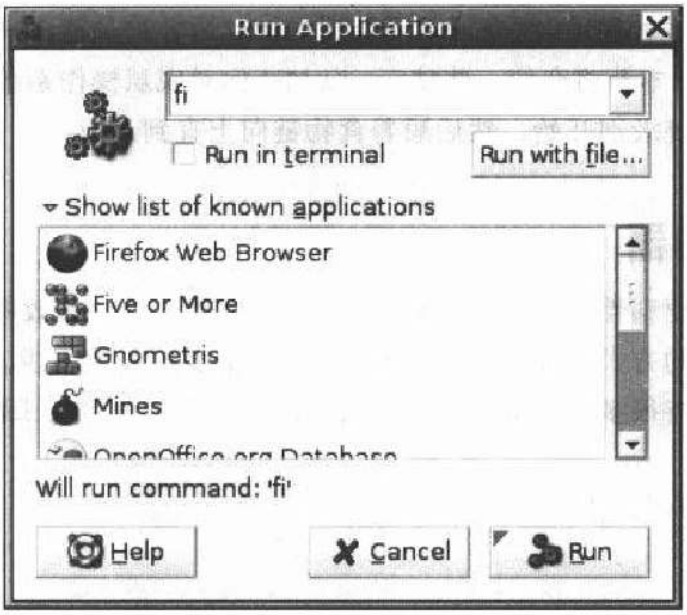


图 2-5：GNOME 的“运行应用程序”加载器

加速器

> **提示：**
>
> 首选键盘而非鼠标。

开发人员实质上是特殊的数据录入职员。我们输入计算机的数据不是来自外界资源，而是来自我们的大脑。但是数据录入操作员的教训仍能使我们产生共鸣。根据他们所输入的信息量来收费的数据录入工人知道，使用鼠标会以数量级程度降低他们的速度。开发人员可以从中学到重要的一课。

VI 编辑器就是一个经典的“不需要鼠标”应用。旁观一个经验丰富的 VI 用户会使人心生敬畏。光标看起来像是在跟随着他们的眼睛。遗憾的是，它的学习曲线太陡峭，大概需要两年坚持使用 VI 才能达到这种程度 —— 就算已经用了一年又 364 天，你还是会有遇到难题的时候。另一款经典的 UNIX 编辑器是 Emacs，同样非常以键盘为中心。然而，Emacs 是一个原始的 IDE：通过插件结构，它可以做的远不止是编辑文件。VI 用户轻蔑地将 Emacs 称为：“一个只有有限文本编辑能力的伟大的操作系统”。

VI 和 Emacs 都支持一个非常重要的加速器：永远不要将你的双手从字符按键上移开。即使是下移到键盘上的箭头按键都会使你慢下来，因为你必须再次回到主排键来输入字符。真正有用的编辑器会使你的手保持在最佳位置，同时进行输入和导航。

如果使用VI达不到这种程度，你也可以领会到如何利用各种加速器来加速你与操作系统和应用程序的交互。本节将介绍一些技巧，以加速你对底层操作系统和IDE等工具的使用。我将从操作系统级别开始，然后顺着食物链向上直到IDE。

操作系统加速器

图形化操作系统对便利性（以及吸引眼球）的重视远胜于纯粹的效率。命令行仍然是与计算机交互最有效的方式，因为用户得到预期结果需要的步骤很少。不过，大多数现代的操作系统仍然支持很多快捷键和其他导航帮助功能来加速你的工作。

Windows 地址栏

> **提示：**
>
> 地址栏是 Windows 资源管理器界面中最高效的部分。

自动补全是命令行中一个很棒的导航辅助功能：你按下Tab键，shell就会自动补全当前目录中匹配的元素。如果有多项匹配，它会生成共同的部分，并允许你输入更多的字符来补全元素（一个目录、一个文件名等）的完整名称。现在所有主流的操作系统都有命令行补全功能，通常都使用 Tab 字符。

如果我仍然使用 Windows 2000 怎么办？

虽然 Windows 2000 默认不会执行命令行中的 Tab 键文件名补全，那也只需要简单修改注册表即可。要打开 Windows 2000 中的文件名补全功能，你只需：

1. 运行 regedit。
2. 找到*HKEY_CURRENT_USER\Software\Microsoft\Command Processor*。
3. 将 EnableExtensions 的 DWORD 值改为 1。
4. 将 CompletionChar 的 DWORD 值改为 9。

许多开发人员都不知道Windows资源管理器的地址栏也提供Tab文件名补全，就像命令提示符一样。切换到地址栏的快捷键是 Alt-D。在那里，你可以输入一个目录的部分名称，敲下 Tab 键，资源管理器就会为你补全。

Mac OS X Finder

Mac OS X提供了大量的快捷键，而且每个应用程序都有一组自己的快捷键。很讽刺的是，考虑到Apple对可用性的普遍关注，OS X各个应用程序中快捷键的一致性就比不上多数的Windows应用程序。微软做了大量的工作来建立和推行一些共同的标准，键盘映射可能是其中最大的成功。不过，Mac OS X有一些很不错的内置快捷键，而另外一些不是那么显而易见。就像Mac的许多其他功能一样，在你发现它们之前还需要有人演示给你看。

关于这一点，Finder和"打开/保存"对话框中的键盘导航是一个很好的例子。在Windows资源管理器中，地址栏很明显。在Finder中，你也可以使用Tab补全来导航到任意文件夹（就像使用资源管理器中的地址栏一样），但这需要键入Apple-Shift-G来显示一个对话框，然后你才可以在其中输入地址。

不要只使用Finder或终端（terminal）（参见本章后面的"你指尖上的命令提示符"一节）。它们能够很好地与对方互动。你可以从Finder中将文件夹拖入终端来作为触发cd命令的快捷方法。也可以使用open命令从终端打开文件，而不必在Finder中双击打开。重点是在适当的上下文环境中，学习手边工具的各种功能，这样你就能够恰当运用它们。

> **提示：**
>
> 花点时间来学习你手边所有隐藏的快捷键。

Windows用户在Mac OS X中最怀念的快捷键是应用程序的Alt键加速器。Mac OS有这些加速器，但它们是基于增量搜索而不是显式的按键关联。Ctrl-F2键将焦点移到菜单栏上，你可以键入所找菜单项的第一部分。当该项被高亮显示时，敲下Enter并逐步键入附属的下级菜单项。这听起来很复杂，但用起来很方便，而且对于所有应用程序都适用。你也可以用Ctrl-F8将焦点移到菜单栏的最右端，所有的服务图标都在那里。

我觉得这个功能最大的问题在于输入Ctrl-F2时高难度的手部操作，所以我使用标准的Mac OS X快捷键对话框来将它重新映射到Ctrl-Alt-Apple-Spacebar（这听起来更糟，但因为它们都排成一行，所以是一个容易键入的简单组合）。再加上我的Quicksilver调用器映射为Apple-Enter，所以我所有的元导航都映射到了差不多同一区域。

如果你使用的是Mac OS X的最新版本，选择菜单项更加简单。Leopard的一个帮助功能会在你输入名字时（或只是部分名称）为你找到菜单项。如果菜单项在嵌套很深的菜单中，或者你只记住了它的名字而忘记了存放路径，或者你认为应用程序应该有某个功能而你不知道对应的菜单项在哪儿，这都会是一个很棒的方法来访问该菜单项。如果你

按了Apple-?键，帮助搜索选项就会出现。键入你需要的菜单项名称的任一部分，Leopard会为你高亮显示该项并在你按下Enter时触发它。就像大量的键盘魔术一样，解释起来很费劲，实际操作一下很容易（图2-6）。

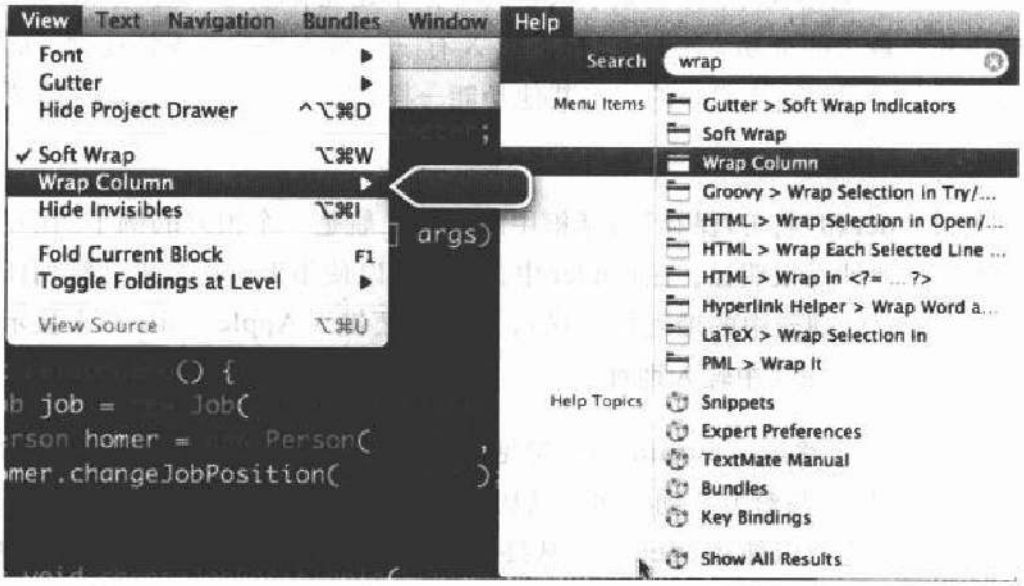


图2-6：Leopard为你找到菜单项

剪贴板

有时候很惊讶我们会倒退到多远。过去的两个传奇编辑器（VI和Emacs）都有多重剪贴板（也叫寄存器）。然而，两个主要的操作系统都限制我们只能用一个可怜的剪贴板。你甚至会认为，剪贴板是一种稀有的自然资源，必须仔细地少量分配来使用，以免有一天我们会用光它。这是一个很好的例子，说明我们因为群体之间的知识传递不足而从一代又一代的开发人员那里丢掉了多少有用的信息。我们总是反复重新发明同样的东西，因为我们没有意识到有人在10年前就已经解决了这个问题。

> **提示：**
> 环境切换会消耗时间。

使用多重剪贴板可能看起来不像是可以极大地提高生产率。可是一旦你习惯了使用它们，就会改变你的工作方式。比如说，如果一个任务需要从一个文件复制粘贴一些不连续的内容到另一个文件，多数开发人员会复制，然后跳转到另外一个文件，粘贴，再跳转回第一个文件，并重复这个繁琐的步骤。很显然，这不是一个高效的工作方式。你最终花了太多的时间在应用程序之间切换。然而，如果你有一个剪贴板堆栈，你就可以从第一

个文件中得到所有要复制的值（将它们塞进剪贴板中），然后跳转到目标文件，并将所有的内容一次性粘贴到适当的位置。

提示：

成批复制粘贴要比反复多次复制粘贴快。

有趣的是，这样一个简单的机制也需要时间去消化它。即使你安装了一个多重剪贴板工具，也需要花一些时间来了解它适用的所有场景。情况往往是，你安装之后很快就忘了它的存在。就像本书中许多提高生产率的建议一样，你需要保持积极清醒的头脑来利用这些技巧。识别出某种技巧所适用的恰当场景就成功了一半。我坚持不懈地使用剪贴板历史记录；我现在无法想象没有它的生活。

幸运的是，Windows和Mac OS X都有多种剪贴板增强软件，开源的和商业的都有。Windows下一个不错的开源替代产品是CLCL（注6），它给你一个可配置的剪贴板堆栈，允许你指定自己的快捷键。对于Mac OS X，JumpCut（注7）是一个简单的开源剪贴板堆栈。想要更强大的（商业的）功能，jClip（注8）很不错：它不仅提供剪贴板堆栈，还提供可配置数目的剪贴板，彼此互不干扰。如果你有一大堆的条目要复制，又不想污染了主剪贴板堆栈，拥有多个独立的剪贴板会很不错。

但是，当你开始习惯于使用剪贴板堆栈的时候要小心。你可能会向一个穿着拖鞋的UNIX用户夸夸其谈地讲起它，结果引发长达一小时的演讲，内容都是关于他如何在你还上小学的时候就使用多重剪贴板，以及那个有20年历史的文本编辑器所制定的所有其他用法。

记住历史

提示：

忘记历史就意味着你得再输入一遍。

所有的shell都有一种历史记录机制，允许你再次调用先前的命令，必要的话可以作一些修改。这是shell相比图形化环境一个很大的优势：你无法在图形化环境中轻易重复有细

注6：*从 http://www.nakka.com/soft/clcl/index_eng.html 下载。*

注7：*从 http://jumpcut.sourceforge.net/ 下载。*

注8：*从 http://inventive.us/iClip/ 下载。*

微变化的行为。因此，学会在命令提示符下执行操作意味着你能够与计算机更有效地沟通。

历史记录查询通常使用上下箭头键，这是一种获得以前使用过的命令的蛮力方法。正如前文所述，搜索比导航更有效。你可以搜索历史纪录找到使用过的命令，这比你倒过来一条一条扫描所有记录更快。

在Windows中，键入先前命令的前一部分然后按一下F8。Shell就会对匹配你刚才输入的前一部分内容的历史命令执行一次反向搜索。你可以不断地按下F8来继续查看匹配的命令列表。如果你想看命令的历史记录，键入F7，你最近的历史记录会显示在一个列表中，你可以利用上下箭头键来选择命令。

在基于UNIX的系统（包括Cygwin）上，你可以选择你想使用的命令行按键语法类型：Emacs（通常这是默认的）或者VI。正如之前提过的，VI有一个超级强大的键盘导航机制，但是从头开始学习非常艰巨。你可以通过在~/.profile文件中添加如下命令为你的*-nix环境设置VI模式：

```
set -o vi
```

当你有了VI模式设定，你可以按下Escape（这会将你置于命令模式下），然后按“/”将自己置于搜索模式下。键入搜索文本，然后按Enter。第一个匹配项将是匹配该搜索字串的最新一次执行的命令。如果那不是你想要的，按下“/”紧接着按Enter来搜索下一个匹配项。同样在bash中，如果你最近执行了一个命令，你可以按入热键“！”连同该命令的首字母来重新运行。“!”直接让你访问到历史记录。如果你想看自己的命令行历史记录，执行history命令，它会以反序提供一个你执行过的命令的编号列表（换句话说，最近的命令是在列表的底部）。你可以使用感叹号（!）+你想调用命令的历史编号来执行该历史记录中的命令。如果你有一些想重新执行的复杂命令，这是非常有用的。

去那儿再回来

作为开发人员，我们不断围着文件系统打转。我们总是需要获取一个JAR文件，去找一些文档，复制一个软件包，或者在bar上安装一个foo。因此，我们必须提高自己的导航和定位技术。正如我反复强调的，图形化的浏览器和查找器很不适合这种跳转（因为那从来都不仅仅是一个跳转……通常是到某些地方去处理一些零星小事的一个往返，因为我们还需要回到出发的地方）。

隐藏的 Alt-Tab 条目

Windows 中的 Alt-Tab 查看器只能容纳 21 项。一旦超过这个数目，多出的条目就不会再显示出来（即使应用程序仍然在运行）。你要么控制资源管理器的打开数量，要么依照下面的两种方案之一，而这两者都涉及 Windows 的 PowerToys。第一个方案是使用 Tweak UI，它允许你配置 Alt-Tab 对话框中显示的条目数。另一个方案是通过 Virtual Desktop PowerToy 来安装多个桌面，我将在第 3 章的“利用虚拟桌面隔离工作空间”一节来讨论该方案。

Mac OS X 允许你在进行 Apple-Tab 切换时终止应用进程，只要在选定的应用程序获得焦点时按下 Q 键就可以关闭它。类似地，如果你使用应用程序管理器 Witch，可以在窗口获得焦点时使用 W 键来关闭单个窗口，这个功能很适合用来关闭繁多的 Finder 窗口。当然，如果你使用 Quicksilver，就不需要那么多的 Finder 窗口了。

一对老牌的命令行工具提供了一种很好的可选方案，让你不必每次在需要跳转到另一个位置时都打开新的资源管理器。你可以暂时切换到另外一个地方，做完任何要做的事情，然后返回到起始的位置。*pushd* 命令执行两个动作：将你置于你作为参数传进去的目录下，并将当前目录入栈。因此，*pushd* 是相对呆板的 *cd* 命令的一个替代物。一旦完成了工作，调用一个 *popd* 命令就可以回到起始的位置。

pushd 与 *popd* 工作在一个目录栈上。这是一个计算机科学意义上的栈，换句话说它是一个 FILO（先进后出）列表（那个经典而恰当的比喻就是自助餐厅的一摞碟子）。因为它是栈，你可以入栈任意多次，出栈的顺序则恰好相反。

所有的 UNIX 系统中都有 *pushd* 和 *popd* 命令（包括 Mac OS X）。Windows 也同样提供了这一对命令 —— 它们不是只有 Cygwin 用户才能享受的稀奇小玩意。

```
jNf ~/work = pushd ~/Documents/
~/Documents ~/work
jNf ~/Documents = pushd /opt/local/lib/ruby/1.
/opt/local/lib/ruby/1. ~/Documents ~/work
jNf /opt/local/lib/ruby/1. 8 =>
```

在这个例子中，从 ~/work 目录开始，跳转到 ~/Documents 目录，然后再到 Ruby 的安装目录。每次 *pushd* 命令被触发时，它都会显示栈中已有的目录。这在我尝试过的所有 UNIX 版本上都是如此。这个命令的 Windows 版本仅执行上述的前两个任务：它不给任何线索让你知道当前堆栈中都有些什么。然而，在使用的时候这不会是一个很大的负担，因为这对命令大多数时候都是被用于快速跳转到另一个位置然后很快就回来。

触手可及的命令提示符

想象你自己正坐在高效程序员的理疗椅上。“我真的愿意花更多的时间在命令行上，但我要做的大多数事情都是在资源管理器中进行”。好吧，我这就来帮你。有几种方法可以使你在图形化视图和命令提示符之间来回切换更便捷。

Command Prompt Explorer Bar

> **提示：**
>
> 嵌入图形化工具的命令提示符让你鱼与熊掌兼得。

对于Windows来说，Command Prompt Explorer Bar（注9）是一个伟大的开源工具：它允许你用快捷键Ctrl-M打开一个命令提示符窗口，该窗口依附在你当前所使用的资源管理器视图底部。在图2-7中可以看到该工具运行的情景。这个工具最棒的特性是它对自己所依附的管理器视图中目录的“粘性”。当你改变了资源管理器中的目录，下方命令提示符窗口中的目录就会自动改变。可惜这种关系不是双向的：改变命令提示符窗口中的目录不会改变资源管理器视图中的目录。尽管如此，这仍是一个有用的工具。

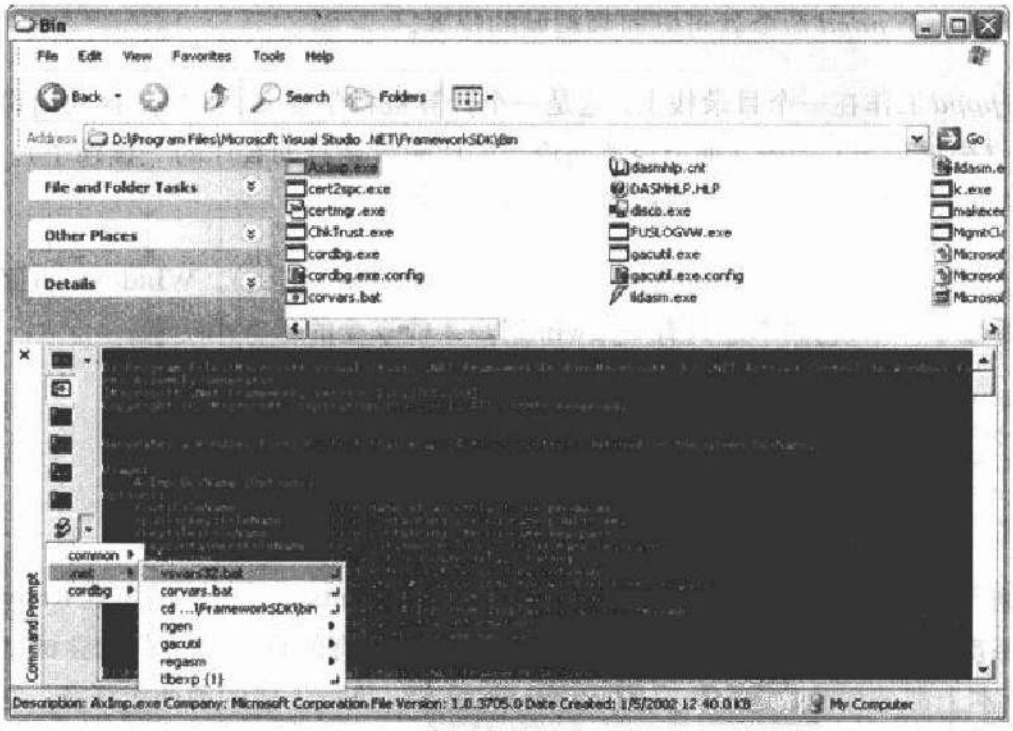


图2-7：Command Prompt Explorer Bar

注9：　从 *http://www.codeproject.com/csharp/CommandBar.asp* 下载。

可惜 Mac OS X 默认并不提供这样的功能。不过，商业的 Finder 替代品 Path Finder（注 10）能做到这一点，如图 2-8 所示。这个终端就和 Mac OS X 中的任何其他终端窗口一样（它会读取 home profile 文件等），而且它可以通过 Alt-Apple-B 快捷键来加载。一旦你习惯了使用如此便于访问的终端，你就会更多地将它用于其适用的任务。

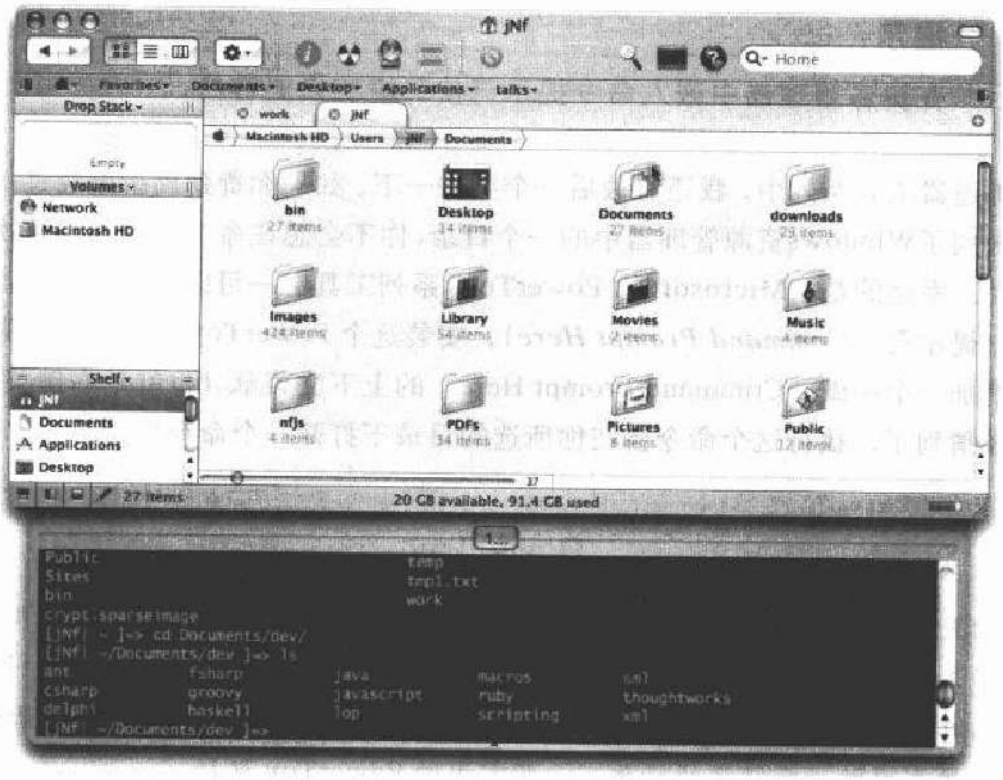


图 2-8：Path Finder 的附属终端

目录结构的图形化视图（Windows 的资源管理器，Mac OS X 的 Finder）也可以通过拖拽这么一种略为隐晦的方式与命令行视图（命令提示符、终端）交互。在 Windows 和 Mac OS X 中，你可以将一个目录拖入命令行视图中来复制路径。因此，如果你想在命令行提示符中进入某个特定目录，只要将 Windows 资源管理器打开到其父目录下（或者到任何可以获得目标目录的地方），键入 *cd*，然后将目录拖到命令提示符中，它就会填充该目录的名称。你也可以使用下一节中要讨论的 *Command Prompt Here*。

另外 Mac OS X 中还有一个更酷的小窍门。如果你使用 Finder 复制了一些文件，可以通过执行一个粘贴操作（Apple-V）在终端窗口中访问它们；Mac OS X 会携带含有文件名的完整路径信息。你也可以使用 *pbcopy*（它将内容复制到剪贴板上）和 *pbpaste*（它

注 10：从 *http://www.cocoatech.com/* 下载。

会从剪贴板上将内容粘贴到命令行中）命令与剪贴板交互来进行管道操作。但是，注意 *pbpaste* 只会粘贴文件名，而不是整个路径。

这里！

提示：

在资源管理器中嵌入命令提示符使环境切换更容易。

在这个加速器工具系列中，我还有最后一个要讲一下。如果你费劲周折地经过一条漫长的路线来到了Windows资源管理器中的一个目录，你不会想在命令提示符中再次重复同样的路径。幸运的是，Microsoft的 PowerToys 系列工具之一可以拯救你：在当前路径打开命令提示符（*Command Prompt Here*）。安装这个 PowerToy 会对注册表做一些修改，并增加一个叫做“Command Prompt Here”的上下文菜单（也就是右键菜单）。你可能已经猜到了，执行这个命令会在你所选的目录下打开一个命令提示符。

PowerToy

微软发布了一组广为人知的PowerToy工具（但并不为其提供支持）（注11）。这些工具从Windows 95时期开始就为Windows增添各种有趣的功能，并且它们都是免费的。其中许多工具（比如Tweak UI）实际上只是一些对话框，对注册表项做一些特定的修改。一些常用的PowerToy包括：

Tweak UI
: 允许你控制Windows的各种视觉效果，比如什么图标出现在桌面上，鼠标的行为，以及其他隐藏的功能。

TaskSwitch
: 改进的任务转换器（与Alt-Tab键相挂钩，该键显示运行着的应用程序的小图标）。

虚拟桌面管理器（Virtual Desktop Manager）
: Windows的虚拟桌面（请见第3章的“用虚拟桌面分割工作空间”一节）。

微软不为这些小工具提供支持。如果你从来没有用过它们，去看看脚注上给出的工具列表页面。很有可能你一直想要Windows为你做的一些事情那里都已经有了。

注11： 从 *http://www.microsoft.com/windowsxp/downloads/powertoys/xppowertoys.mspx* 下载。

Cygwin也不逊色：你可以从Cygwin中运行chere，从而得到一个“Bash Here”右键菜单，它会在选中的位置打开一个Cygwin的bash shell（而不是命令提示符）。这两个工具能很好地一起工作，所以你可以把两个都装上，并根据不同的情况来决定你是需要一个命令提示符还是一个bash shell。命令：

```
chere -i
```

会安装Bash Here上下文菜单，而：

```
chere -u -s bash
```

会卸载它。实际上，通过命令：

```
chere -i -s cmd
```

也可以安装一个Windows命令提示符“Command Prompt Here”右键菜单（就像Windows PowerToy一样）。

所以，如果你有Cygwin，就没有必要去下载“Command Prompt Here”PowerToy，用chere就行了。

Mac OS X下的Path Finder也有一个“在终端中打开”右键菜单选项，但是它会打开另一个终端窗口（不是图2-8中所描述的简单版本，而是一个功能完备的独立窗口）。而且Quicksilver有一个叫作“在终端中到达那个目录（Go to the directory in Terminal）”的动作。

开发加速器

测验一下：什么是你屏幕上最大的可点击目标？答案是：正位于你光标下的那个目标，这就是为什么右键菜单中应当放置最重要的内容。从执行点击的角度来说，你鼠标下的目标对象实际上是无穷大的。第二个问题：什么是第二大的目标？是屏幕的边缘，因为你可以用最快速度把鼠标甩到边缘，却不会越过它。这说明真正重要的内容应该放到屏幕的边缘。这些结论来自于Fitt定律：“需要移动的距离”和“目标的大小”共同决定了用鼠标点击一个目标的容易程度。

Mac OS X的设计者知道这个定律，这就是为什么菜单栏位于屏幕的顶端。当你使用鼠标点击一个菜单项时，你可以飞速将鼠标指针移到屏幕顶端，就到了你想要去的位置。而另一方面，Windows在每个窗口顶端都有一个标题栏。即使窗口被最大化，你在快速到达顶端之后还必须仔细找到你的目标，然后精确地操作鼠标来点击目标。

对于一些Windows应用程序，有一种方式来改善这种情况。微软Office系列有一种“全屏”模式，该模式屏弃了标题栏而将菜单放在了屏幕顶端的右侧，就象Mac OS X一样，而且上面也有开发人员用的帮助项。Visual Studio采用了同样的全屏模式，正如IntelliJ为Java开发者所做的。如果一定要用鼠标的话，在全屏模式下更容易点中菜单。

但我确实不提倡过多使用鼠标。编程（除了用户界面设计）是一项基于文本的行为，所以你应该尽量将手放在键盘上。

提示：

编程时始终优先使用键盘而非鼠标。

你整天都用IDE编码，而IDE有很多快捷键。把它们都学会！用键盘游走于源码中通常比用鼠标快。但快捷键数量之多，完全了解可能很困难。学习它们的最好办法是让使用快捷键成为你的本能。背诵长长的快捷键列表没有用，因为你不在使用它们的上下文中。Eclipse IDE有一个很好用的快捷键“Ctrl-Shift-L”，能够为某个特定的视图显示所有其他的快捷键。这是个很棒的助记符，因为它已经在适当的上下文环境中了。学习快捷键的最好时机是你需要执行这个动作的时候。当你打开菜单时，留意一下上面的快捷键，然后先不要选择菜单项，而是记住快捷键，再退出菜单，在键盘上使用快捷键。这样会强化目标任务和快捷键之间的联系。另外，不管你信不信，在使用的同时大声说出快捷键也会有帮助，因为这会迫使大脑的更多部分接收这个快捷键的信息。当然，你的同事可能会认为你很神经，但不久你运指如飞的功力就会征服他们。

我的一位同事有个教人使用快捷键的好方法。无论你什么时候和他结对编程，如果你用鼠标选择菜单或工具按钮，他都会让你撤销操作，然后用键盘操作3次。的确，最初这样会使你慢下来，但是这种强化记忆的方法（再加上你忘记快捷键时他凶狠的眼神）就像高压锅一样，能让你很快地掌握快捷键。

提示：

在上下文中学习IDE快捷键，而不要去背长长的列表。

另外一种记忆快捷键的好方法是让某人（或某物）提醒你这一点。IntelliJ就有这样一个很妙的插件叫做“按键提示器（Key Prompter）”。每次你使用菜单进行选择时，一个对话框就会弹出来告诉你可以用的快捷键，以及你已经做错了多少次（见图2-9）。在

Eclipse中也有同样的工具，也叫做Key Prompter（注12）。这个Key Prompter更强大：你可以设置一种模式，该模式拒绝菜单选择，强迫你使用快捷键！

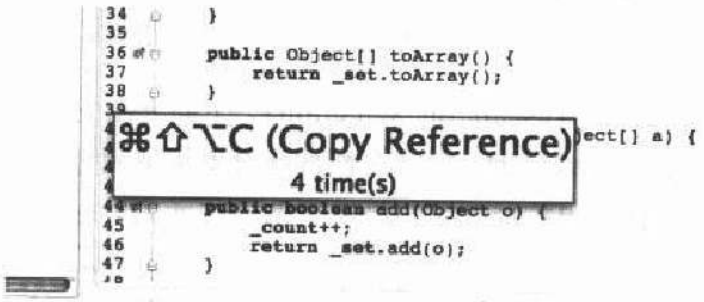


图2-9：Key Prompter是IntelliJ的一个有用的快捷键工具

遗憾的是，许多有用的快捷键根本不会标注在菜单选项上：它们被淹没在了长长的快捷键列表中。你应当找出你所使用的IDE中那些很酷的快捷键。表2-1是为Java开发者准备的一个简短的快捷键列表，其中列出了那些很酷的、被隐藏的IntelliJ和Eclipse（Windows版本）快捷键。

表2-1：IntelliJ和Eclipse快捷键精选

描述	IntelliJ	Eclipse
跳转到某个类	Ctrl-N	Ctrl-Shift-T
符号列表	Alt-Ctrl-Shift-N	Ctrl-O
增量搜索	Alt-F3	Ctrl-J
最近打开或编辑的文件	Ctrl-E	Ctrl-E
引入变量	Ctrl-Alt-V	Alt-Shift-L
逐级选择	Ctrl-W	Alt-Shift-Up箭头

其中有几条需要深入解释一下。“最近打开或编辑的文件”一项在两种IDE中工作的方式不同：在IntelliJ中，它会给出你最近编辑过的文档列表，按访问次序逆序排列（所以最新的文档在最顶部）。在Eclipse中，快捷键会提供一列被打开的缓存。这个功能对开发人员很重要，因为我们倾向于在一小部分经常会访问的文件上工作，所以能便捷地访问一小组文件会很有帮助。

变量引入（introduce variable）从技术角度来讲是一种重构，但我常常用它来输入表达

注12： 从*http://www.mousefeed.com*下载。

式的左边。在两种IDE中，你都可以输入表达式的右侧（比如Calender.getInstance();）然后让IDE来提供左侧内容（本例中是Calendar calendar =）。IDE在给变量起名方面几乎和你做得一样好，这样就少了许多输入工作，而且不用去考虑以什么来命名变量。（这个快捷键使我在用Java编码时变得很懒。）

最后一条特殊项是逐级选择（escalating selection），下面介绍它如何工作。当你把光标放置在某个对象上并调用该命令时，它就会把被选对象扩展到更高一级的语法元素。再按这个键，它会再把所选对象扩展到下一级更大的语法元素分组。因为IDE理解Java语法，它知道是什么组成了一个标识、一个代码块、一个方法等。不用创建多个快捷键来选择每种元素，你可以反复使用相同的按键逐渐扩展选择范围。这描述起来会有点抽象，但是试一试，你就会很快喜欢上它。

这里有个方法来学习和记忆你在大量快捷键中偶然看到的相当酷的快捷键：通读一遍快捷键列表，把那些确实有用但你不知道的快捷键复制到一个单独的文件中（甚至是写到纸上！）。设法记住都有哪些功能，并在下次需要用的时候从你的小抄上去查。这代表了“我知道可以这样做”和“这如何去做”之间欠缺的一环。

另外一个利用IDE提高生产率的关键是代码模板（live template）。你总是会用到相似的代码块。大多数IDE都允许你为模板设置参数，并在编辑器中使用模板时给参数赋值。例如，下面是Intellij中的一个参数化模板，用于在Java中做数组遍历：

```
for (int $INDEX$ = 0; $INDEX$ < $ARRAY$.length; $INDEX$++) {
  $ELEMENT_TYPE$ $VAR$ = $ARRAY$[$INDEX$];
  $END$
}
```

当这个模板被打开后，IDE首先将光标定位到第一个$定界值处，让你输入下标名称，然后按Tab键到下一个参数。在这个模板语言中，END标志是所有扩展完成之后光标结束的位置。

每个IDE都有不同的模板语法，但几乎每个值得一用的IDE都支持这个概念。学习IDE中的模板语言，并尽可能多地使用它。出色的模板支持是TextMate（注13）和E-Text（注14）编辑器流行的原因之一。使用模板不会造成拼写错误，为复杂的语言结构生成模板可以在编码时节省你的时间和精力。

注13：从*http://macromates.com/*下载。

注14：从*http://www.e-texteditor.com/*下载。

提示：

当你第二次输入一个复杂结构时，将它做成模板。

在工具中也使用查找手段来导航

代码结构层次太深是无益的。一旦达到一定的规模，不管是文件系统、包结构，还是其他的分层系统，都会因层次太深而影响有效导航。大的Java项目就因此受到影响，因为包结构是与目录结构联系在一起的。即使是一个小的项目，你也必须顺着树结构不断展开节点，从而找到一个文件，即便是你已经知道了文件名称。如果你发现自己在这样做，那你就工作得太辛苦了。

现代的Java IDE允许你在当前工程中快速找到任意Java源文件。以IntelliJ为例，在Windows中“打开Java源文件”的快捷键是Ctrl-N，在Mac中是Apple-N；Eclipse的快捷键则是Ctrl-Shift-T。图2-10显示的例子来自IntelliJ；它在编辑器中打开一个文本框，你可以输入所找文件的名称。

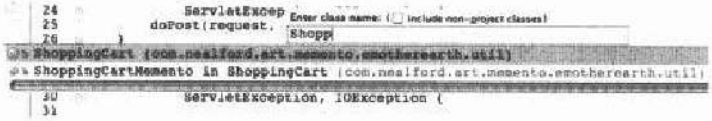


图2-10：IntelliJ的“查找文件”文本框

输入完整名称（甚至名称的一部分）很麻烦。如果IDE知道你如何命名文件就好了——它的确知道。你只需输入大写字母，不必输入文件名，它就会查找有着相同大写字母模式的名字。比如说，如果你要查找文件*ShoppingCartMemento*，可以输入SCM，IDE就会忽略中间的小写字母找到与大写字母匹配的模式，如图2-11所示。

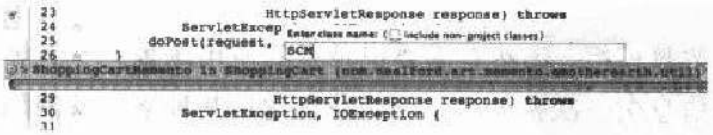


图2-11：IntelliJ的智能名字模式匹配

这种文件查找的技巧也适用于非Java源文件（在IntelliJ中你只需多按一个Shift，在Eclipse中则是Ctrl-Shift-R）。“查找资源”文本框的工作方式就像“查找文件”文本框一样。不要再在巨大的源文件树结构中埋头苦干：你知道你想要什么，那就直接去找到它。

对于.NET开发人员，常见的开发环境是Visual Studio 2005（或是它的后续版本）。它提供的快捷键不是特别多，不过你可以结合使用商业化的Resharper（来自IntelliJ Java IDE的开发商JetBrains）使其更高效。很多开发人员都认为Resharper主要关注于对重构的支持，但是精明的开发人员意识到它还增加了大量的快捷键，包括之前描述的“查找文件”功能。

宏

宏是被记录下来的与计算机交互的片段。一般来说，宏记录器是每个工具自带的（因为只有那个工具知道如何来处理按键）。当然，那就意味着没有标准的宏语法，有时候甚至在同一产品的不同版本之间都会不同。许多年来，微软的Word和Excel的宏语法都相去甚远，即使它们是出自同一个公司，并且是在同一个Office套件中。直到Office 2000，微软才终于统一了宏语法。尽管工具之间存在着巴别塔，宏仍然能帮你解决每天都要面对的特定问题。

宏记录器

> **提示：**
>
> 如果要对多行文本做同样的操作，就应该找出其中的模式，并把它记录为一个宏。

你有多少次发现自己在某种模式下工作？你从一个XML文档中剪贴一些信息，现在你必须删除实际数据周围的所有不必要的XML标记来进行清理。宏曾经在开发人员中风靡一时，但现在似乎不受青睐了。我猜想可能是因为大多数现代IDE的模板功能减少了对宏的需求。

但是，无论你多么依赖模板，仍然有需要记录宏的使用场景。一个常见的场景是以前就被关注的：对某些信息做一次性处理，可能是将它从某种格式下消除标记，或者是为它添加标记使其能为某些工具所使用。如果你能转换任务的视角，将它看作是一系列可重复的步骤，你就会发现宏可以被用来处理许多琐碎的事情。

> **提示：**
>
> 在一段文本上执行某个特定操作的次数越多，就越有可能会再次重复它。

即便使用Eclipse（它没有宏记录器），你仍然可以单为这件事转去使用某个文本编辑器的宏记录器。选择文本编辑器的一个重要依据就是它的宏记录工具和记录宏的格式。如果宏能够产生一些可以手工修改的可读代码，能够创建以后一直都可以使用的可重用的内容，那将会更棒。毕竟，如果你现在需要从一个格式剪贴一些内容到另一种格式，那很有可能以后你还需要做这件事。

键盘宏工具

编辑器中正式的宏对于处理文本、代码和格式转换都很不错，而另一类的宏工具却能帮你解决一些日常事务。所有主流操作系统都有开源和（或）商业的键盘宏工具。键盘宏工具在后台运行，等待着某种文本模式来展开。它们允许你输入一个缩写来代替全文。大多数情况下，这种工具都做一些像根据邮件地址自动输入问候语这样的事情。但是，作为开发人员，我们经常在一些没有代码模板的地方（比如在命令行或者Web浏览器中）输入重复的文本。

> **提示：**
>
> 不要总是重复输入相同的命令。

我经常要给别人展示如何使用Selenium的远程控制功能。为了让它工作，我必须开启一个代理服务器并发出隐秘的命令来给它指令，这些命令实际上只是命令行中的一些口头禅。由于不在IDE中，所以我不能使用模板甚至宏。我甚至不能使用批处理或者shell脚本，因为我在运行一个交互性的代理。不久我就发现，应当将这些命令保存在我的键盘宏工具中：

```
cmd=getNewBrowserSession&1=*firefox&2=8080
cmd=open&1=/art_emotherearth_memento/welcome&sessionId=
```

在我启动Selenium的代理服务器后，这行丑陋的代码就会以“远程控制Selenium”要求的特定格式被发出。如果完全不了解Selenium，你不会明白这些命令的含义。但这个例子并不是要你明白这些命令的含义。这只是我不时要输入的令人讨厌的命令串之一。每个开发人员都会遇到这样的字符串，而且在它们的上下文环境之外很难理解（往往在上下文环境中还是不太容易理解）。但是现在，不必再到处复制粘贴这些命令，我只需输入*rcsl1*来产生第一行，输入*rcsl2*来产生第二行，对于需要展示的那十多条命令同样如此。

一些键盘宏工具允许你在操作系统级记录下键盘操作并回放（有时候甚至可以捕捉鼠标点击和其他交互操作）。另外一些宏工具要求你输入想与特定宏关联起来的命令。在这

两种情况下，你都是在捕捉一些需要重复进行的操作系统级的交互，并保存为特定的格式以便于重用。

将键盘宏工具用于必须反复输入的常见词句也很棒。比如你必须在Word中输入的项目进度信息文本，或者是在你的时间表和报销系统中输入工作小时数。键盘宏工具属于那一类的工具：你之前甚至不知道它的存在，在以后某一天就变成了“没有它我还怎么活”。

Windows中最流行的键盘宏工具是AutoHotKey（注15）（它是开源的）。Mac OS X有两个属于“商业的但不贵”的那一类，比如TextExpander（注16）和Typinator（注17）。

小结

谈论如何使用加载器、剪贴板管理器、IDE快捷方式和本章提到的所有各类工具来加速你与计算机的交互是一件事，但运用它们又是另外一件事。你知道什么可以加速你的工作，但是你觉得没有时间去运用它们。“我知道有个快捷键可以做这件事，但是我赶时间，所以我代之以使用鼠标，以后再去查找那个快捷键”。这个“以后的事情”永远都不会发生。要想变得更有效率，就得找到一种平衡：既要不断追求提升生产率，同时又不降低当前的生产率（我知道，这听起来很有些反讽）。尝试每周掌握一种提高生产率的方式，将精力集中在那一个方法上直到根深蒂固，然后再尝试下一个。这种方式对当前生产率产生的影响很小，同时又让你逐渐提高生产率。

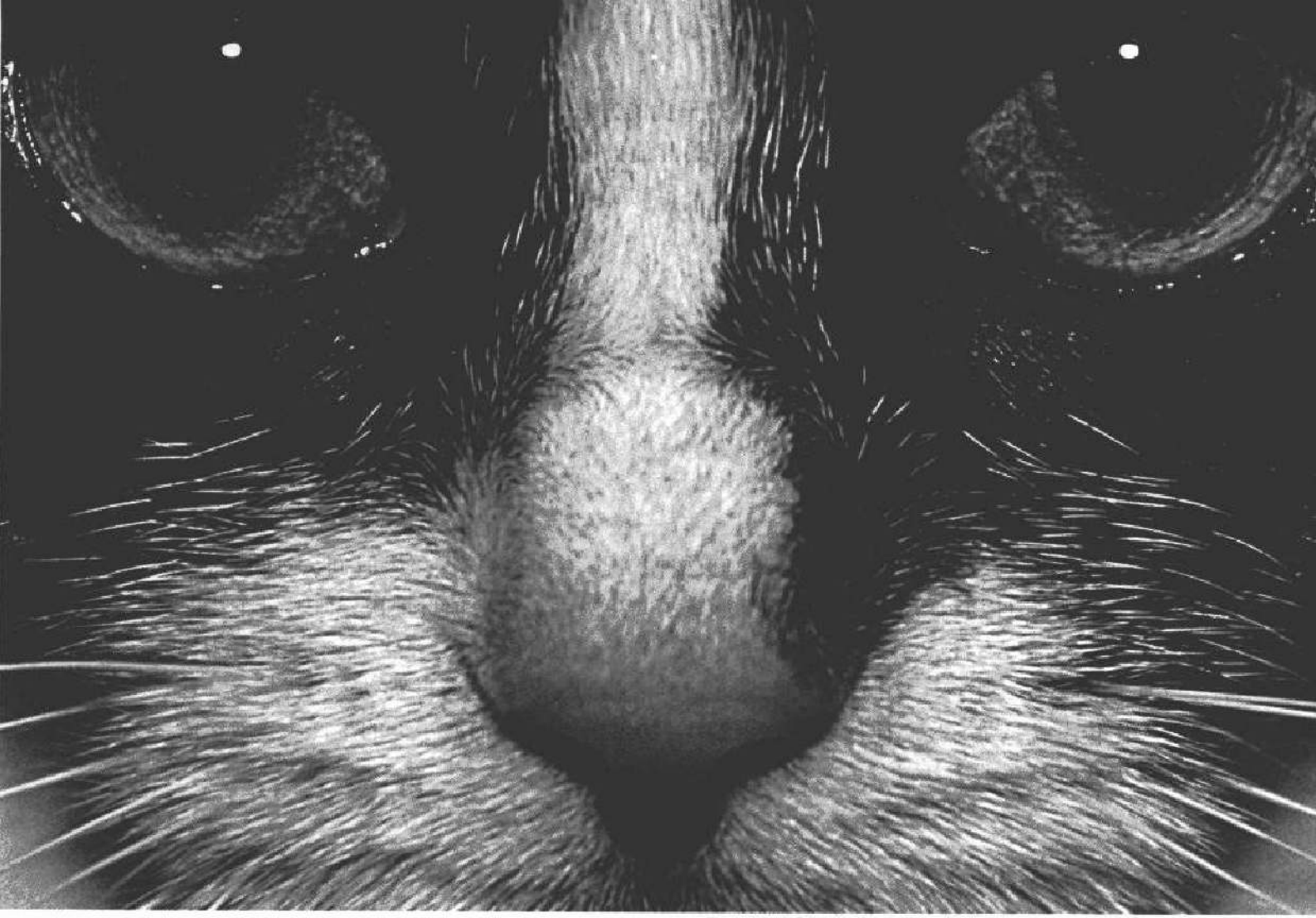
> **注意：**
>
> 每天花一点点时间来使每一天都更高效。

运用加速方法有两个条件：对加速器的了解，以及使用它们的适当场景。比如，在你的机器上安装一个剪贴板工具，并在你每次需要复制粘贴的时候都强迫自己去考虑使用它。你会逐渐开始了解到它是如何为你节省时间的，因为你一次性地复制多个值，然后又将这些值作为一组进行粘贴。在掌握了这个工具之后，再转向下一个。一切的要点都在于寻找一种平衡：一方面需要花时间去学习，另一方面这些学习会让你变得更高效。

注15：从 *http://www.autohotkey.com/* 下载。

注16：从 *http://www.smileonmymac.com/textexpander/index.html* 下载。

注17：从 *http://www.ergonis.com/products/typinator/* 下载。



第 3 章

专注法则

本章主要介绍一些去除低效率和不必要的干扰的方法，来帮你集中注意力。你很可能有过这样的经历：在工作中被计算机或外界的其他因素不停地分散注意力。从本章里，你会学到一些集中注意力的方法，例如使用特定的工具、以特定的方式与计算机交互，以及让你的同事停止打扰你以便能完成手上的工作。本章的主要目的是使你重新找到征服一座山峰后兴奋得头晕目眩的感觉。

排除干扰

作为一个知识工作者，你的收入来自于创造性的工作。不论是在办公桌还是在电脑桌面上，不断的干扰都会使你无法为项目作出最大贡献。开发人员渴望达到流畅的工作状态，这个话题在很多地方被讨论过（甚至有一整本书专门来讨论这话题，作者是Csikszentmihalyi）。所有开发人员都知道这种状况：当你集中精力时，时间会过得很快。你几乎与机器（以及你所应对的问题）形成了共生关系。你会惊讶地说："哇，4个小时过去了？我都没有注意到。"但这种流畅的工作状态是脆弱的：一旦分心，就得努力回到原来的状态。到了一天中较晚的时候，回到流畅的工作状态会更费劲。干扰越多，回到工作状态就越难。分心使你不能集中解决手头的问题，从而减少你的生产率。幸运的是，用几个简单的方法就能有效防止分散注意力。

> **提示：**
>
> 精力越集中，思维越缜密。

隔离策略

注意力是难以维持的，尤其是当电脑似乎决心把你的注意力从工作中拖开时。阻止视觉及听觉的打扰可以帮你保持良好的状态。对于听觉的打扰（尤其是如果你没有一个可以关上门的办公室时），你可以戴耳机（即使不是在听音乐）。当别人看到你戴着耳机，他们就不太可能打扰你。如果办公环境不容许戴耳机，考虑把"请勿打扰"标志贴到工作隔间的入口。这样应该能让其他人在打搅你之前三思。

对于视觉打扰，你应该关掉机器上所有分散注意力的东西。电子邮件通知很有害，因为它会让你感到紧迫。在一天中接收的电子邮件中，有多少是真正需要立即作出反应呢？关闭电子邮件客户端，然后当到了自然的休息时间，再一次性检查你的邮件。这让你可以自己决定什么时候从工作状态中抽离。

关掉不需要的提示

Windows的气球式提示和Mac OS X上的growl通知也会分散注意力。在Mac OS X上，growl是可定制的，你可以只查看需要的通知。但Windows的气球式提示是“全有或全无”的。大量的邮件提示毫无用处。你真的要停止工作来清理桌面上未用的图标吗？有时，你也会得到Windows自动调整虚拟内存的信息通知——我可不想知道这个，更不希望为它打断工作。Windows有时就像一个被惯坏的3岁小孩，一直大喊大叫着想要引起你的注意。

有两种方法可以关闭气球式的提示。如果已经有了Tweak UI PowerToy，其中的一项设置就是禁用气球提示（如图3-1）。另一个方式需要编辑注册表（这其实就是PowerToy在幕后做的工作）：

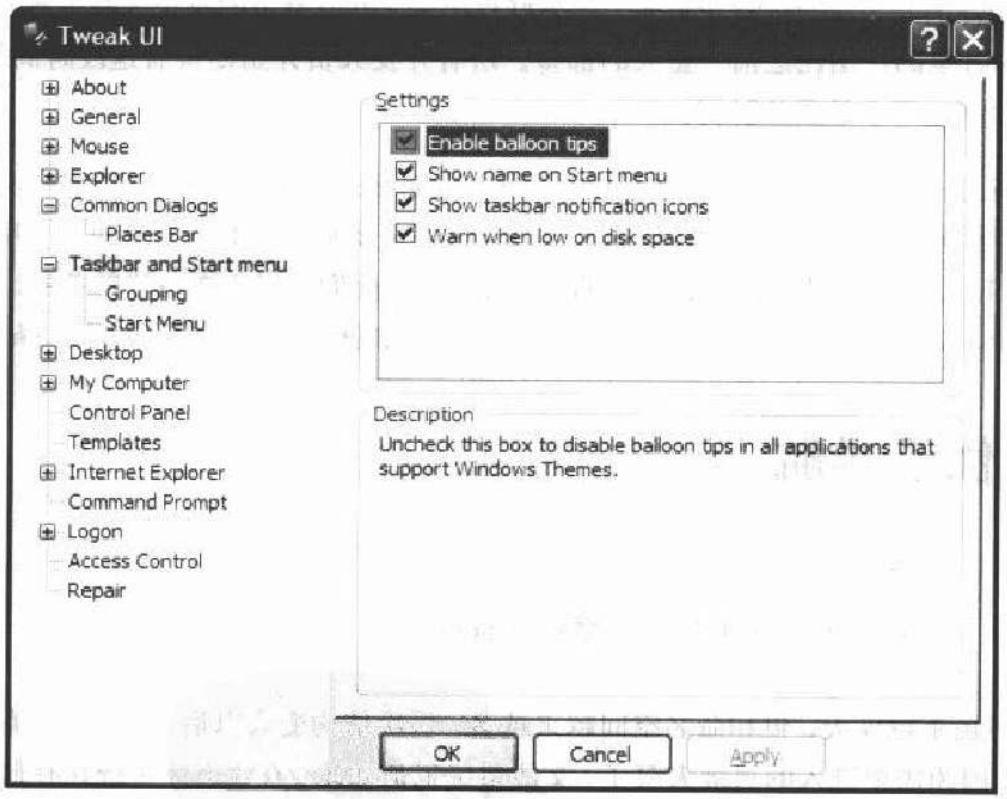


图3-1：用Tweak UI关掉气球式的提示

1. 运行 regedit。
2. 找到 *HKEY_CURRENT_USER \Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced*。
3. 创建一个叫EnableBalloonTips名字的DWORD值（或者编辑它，如果它已经存在的话），将其设为0。

4. 退出系统，重新登录。

如果你在工作中经常创建大量互相重叠的窗口，它们也会使你分心。有几个免费的工具软件可以使背景变黑，把不用的应用淡出，这样你就可以把注意力集中在手头的任务上。

Windows 上的 jediconcentrate（注 1）就是干这个用的，与之类似的还有 Mac OS X 上的 Doodim（注 2）。它们的工作方式都一样，你可以订制背景变暗的程度。

创造安静时间

如果办公室里有很多其他开发人员，可以考虑规定一段“安静时间”，例如上午 9 时至 11 时、下午 3 时至 5 时。在这段时间内，每个人都把电子邮件关掉，不开任何会议，除非有紧急情况（例如，如不解决某个问题就不能继续工作）否则不允许打电话或交谈。我在一家咨询公司实验过这个规定，效果好得出奇：办公室里的每个人都发现，我们在那 4 个小时里的产出比之前一整天的都多。所有开发人员开始期待着这段时间的到来，这成了大家一天中最爱的时间。

我还知道另一个开发团队在他们的共享日历中定期召开会议，而这些“会议”其实只是隔离出时间来做工作。公司里其他的人可以从共享日历看到这帮人都在开会，所以不去打扰他们。办公环境是如此损害员工的生产率，逼得他们不得不要这样的花招以完成应做的工作，这实在有些可悲。为了完成工作，有时你就要冲破办公环境中的条条框框。

搜索优于导航

> **提示：**
>
> 草堆越大，从中找到一根针就越难。

项目变得越来越庞大，包和命名空间越来越多。层次结构变大以后，在其中导航就会变得费劲，因为需要进入的层级太深了。文件系统也是如此：有些文件系统在存储 200MB 数据时很好用，而 200GB 数据就没法应付了。文件系统就像一个硕大的干草堆，而我们总是需要从中找出一根又一根的针。四下寻找文件，这个费时费力的操作会使你分心，无法集中精力考虑真正重要的问题。还好，在新的搜索工具帮助下，你几乎可以完全抛弃麻烦的文件系统导航。

注 1：可以在 *http://www.gyrolabs.com/2006/09/25/jediconcentrate-mod/* 下载。

注 2：可以在 *http://www.lachoseinteractive.net/en/products/doodim/* 下载。

一些强大的搜索工具最近开始在操作系统层面上出现，例如Mac OS X的Spotlight和Vista的Windows Search。这些搜索工具跟以前Windows上拙朴的搜索功能全然不同——后者唯一的用途恐怕就是展示一条小狗的动画。新的搜索工具会索引硬盘上一切有趣的东西，于是搜索就会飞快完成——不仅限于文件名，它们还会索引文件的内容。

即便尚未升级到Windows Vista，也有一些桌面搜索插件可用。我目前最欣赏的是免费的Google Desktop Search（注3）。默认设置下它只能搜索“普通”文件（例如Excel数据表、Word文档、邮件等），但Google Desktop Search的一大优点是它的插件API，开发者可以用它来添加搜索插件。比如说，Larry's的Any Text File Indexer（注4）就可以让Google Desktop Search对源代码进行搜索。

只要安装Larry's Any Text File Indexer，并允许它对硬盘进行索引（这是在后台利用空闲时间完成的），就可以搜索文件内容中的代码片段。比如说，Java规定文件名必须与文件中定义的public类名匹配，大多数编程语言（例如C#、Ruby、Python）的命名规范也是文件名与类名匹配。或者如果你想查找所有用到某个类的文件，也可以用已经知道的代码片段来进行搜索。例如用下列片段：

```
new OrderDb();
```

就能找出所有创建了`OrderDb`实例的类。

搜索文件内容是一个极其强大的功能。就算不记得文件名，你总能记起其中的一部分内容。

> **提示：**
>
> 不要文件树，要搜索。

索引搜索工具使你得以摆脱文件系统的暴政统治。Google Desktop Search这样的工具很可能需要你花些时间来适应，因为“亲手找文件”的习惯已经在我们脑子里根深蒂固。你并不需要这些搜索工具来帮你找到源代码（因为IDE能帮你搞定），但你经常需要访问某个文件所在的位置，以便在那里执行别的操作（例如版本控制、diff或是从另一个项目复制某些东西过来）。Google Desktop Search允许你右击找到的文件，然后打开它所在的目录。

Mac OS X里的Spotlight也有同样的功能：找到一个文件以后，如果你按回车键，它会

注3：可以在*http://desktop.google.com*下载。

注4：可以在*http://desktop.google.com/plugins/i/indexitall.html*下载。

用相关的应用程序来打开这个文件；如果按 Apple- 回车键，就会打开该文件所在的目录。跟 Google Desktop Search 一样，你也可以下载 Spotlight 插件使之支持源代码文件索引。比如说，可以从苹果网站下载一个 Spotlight 插件来支持 Ruby 代码索引（注 5）。

而且 Spotlight 还支持对搜索结果过滤。比如说，你可以给搜索字符串加上 "kind:email" 限制，这样 Spotlight 就只会搜索电子邮件。这种设计体现了搜索的未来趋势：定制属性的搜索（参考下面的 "不远的将来：按属性搜索" 部分）。

不远的将来：按属性搜索

只按文件名搜索已经被证明用处不大。要准确地记住文件名，一点都不比记住它放在哪儿来得容易。按内容搜索则有用得多，因为你更有可能记得文件中的某一部分。

一种更强大的搜索方式已经初露端倪：根据可定制的属性来搜索文件。比如说，假设有一些文件同属于一个项目，其中包括 Java 源文件、SQL 表结构、项目备忘录、进度跟踪数据表等。把这些文件放在同一个目录下是合理的，因为它们都与同一个项目相关。但如果需要在几个项目之间共享某些文件该怎么办呢？好的搜索方式可以让你在需要用到这些文件的地方（而非实际存放这些文件的地方）找到它们。

总有一天，我们会得到更 "聪明" 的文件系统，允许你给文件打上任意属性的标记（tag）。现在 Mac OS X 的 Spotlight Comments 就有这个功能，你可以给逻辑上属于同一项目的文件打上统一的标记，而不用管这些文件存放的物理位置。Windows Vista 也提供了类似的功能。如果你的操作系统有这个功能，把它用起来！这是一种比目录树好得多的文件组织方式。

找出难找的目标

提示：

在诉诸高级搜索之前，先尝试简单的搜索。

Google Desktop Search、Spotlight 和 Vista 都很擅长找到你需要的文件，只要你知道其中的部分内容。但有时你还需要一些更复杂的搜索功能。前面提到的这些工具都不支持

注 5：　可以在 *http://www.apple.com/downloads/macosx/spotlight/rubyimporter.html* 下载。

正则表达式，这实在有些难堪，因为正则表达式早已被证明是一种极其强大的搜索机制。找东西用的时间越少，你就能越快把注意力集中到真正需要解决的问题上。

各种UNIX（包括Mac OS X、Linux以及Windows上的Cygwin）都有一个叫做find的工具，它可以从指定的目录起向下递归遍历目录结构查找文件。find提供大量参数来对搜索加以优化，其中包括以正则表达式匹配文件名。比如说，下列find调用会找到扩展名之前两个字母为“Db”的所有Java源文件。

```
find . -regex ".*Db\.java"
```

在这个搜索中，*find*从当前目录（“.”）开始，查找所有符合下列条件的文件：文件名中包含“Db”字样，其前面有0个或多个字符（“.*”），其后面是“.”符号（必须用反斜杠转义，因为正则表达式中的“.”符号表示“任意单个字符”），紧跟着是“java”扩展名。

*find*本身就是有用的，如果再加上*grep*那就真是如虎添翼。*find*有一个*-exec*选项，可以用于执行其后的命令，并传入找到的文件名作为参数。换句话说，find先找到与条件匹配的文件，然后把这些文件逐一传给-exec右边的命令。请看下列命令（表3-1有详细的解释）：

```
find . -name "*.java" -exec grep -n  -H "new .*Db.*" {} \;
```

表3-1：详解find命令

字符	作用
`find`	执行find命令
`.`	在当前目录下查找
`-name`	文件名匹配“*.java”（请注意，这不是正则表达式，只是文件系统的通配符：*可以匹配任意字符串）
`-exec`	针对每个找到的文件，执行下列命令
`grep`	*grep*命令，强大的*-nix工具，用于在文件中查找字符串
`-n`	显示匹配的行号
`-H`	显示匹配的文件名
`"new .*Db.*"`	用于匹配的正则表达式，它表示“字符串‘new’后面跟着若干字符，然后是‘Db’，然后又是若干字符”
`{}`	占位符，代表find找到的文件名
`\;`	结束-exec后面的命令。由于这是一个UNIX命令，你可以把它的结果管道给另一个命令，所以find命令需要这个结束符来知道“exec”部分到哪里为止

虽然这很麻烦，但你应该能看出这个命令组合的威力了（在附录的“命令行”一节中可以看到另外两个有着异曲同工之妙的组合）。学会语法之后，就可以对代码库进行文本查询了。下面是另一个稍稍复杂的例子：

```
find -name "*.java" -not -regex ".*Db\.java" -exec grep -H -n "new .*Db" {} \;
```

可以在代码评审时使用这个*find+grep*的组合 —— 实际上这个例子正是我在代码评审中用到的，当时我们开发的应用程序采用典型的分层架构设计：模型、控制器和视图。所有访问数据库的类都以“Db”结尾，我们的设计原则是：除了控制器以外，其他的类都不应该实例化这些与数据库相关的类。

这里还有另一个用来找东西的命令行小技巧。如果想要去路径上某个应用程序所在的目录，该怎么做？比如说，假设你想临时去可执行命令java所在的目录，可以组合使用*pushd*和*which*来达到目的：

```
pushd `which java`/..
```

记住，反引号（“`”字符）中的命令会在其他命令之前被执行。在这里，*which*命令（找出路径上某个应用程序所在的位置）找出java所在的位置。但java是一个应用程序，而不是目录，所以我们*pushd*到它的父目录。这个例子很好地展示了*-nix命令的可组合性。

使用有根视图

所谓有根视图（rooted view）是一种“以某个特定子目录为根”的目录结构视图，从中只能看到这个根目录以下的内容。假如你正在一个项目上工作，这时你不会关心其他项目的文件，有根视图就能帮你免受其他无关文件的打扰，把注意力完全集中在工作需要的文件上。各个平台都支持这个概念，不过实现略有差异。

Windows的有根视图

图3-2展示了一个有根的资源管理器视图（它的“根”位于*c:\work\sample code\art\art_emotherearth_memento*目录）。

这就是一个普通的资源管理器窗口，不过是用下列参数来打开的：

```
explorer /e,/root,c:\work\cit
```

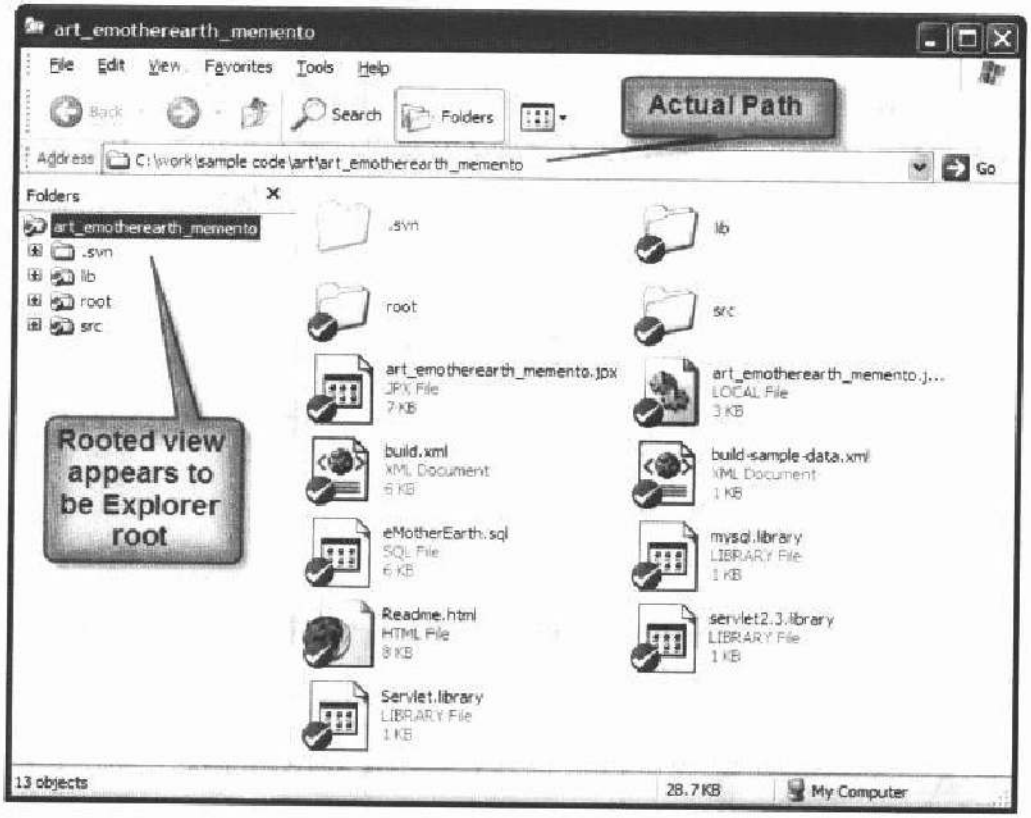


图 3-2：Windows 上的有根视图

有根视图的效果只在这个资源管理器实例中有效。如果用惯用的方式打开另一个资源管理器窗口，就会看到一个普通的资源管理器。为了充分利用有根视图，在创建资源管理器快捷方式时就应该加上根目录参数。有根视图在所有版本的 Windows（从 Windows 95 直到 Windows Vista）上都有支持。

提示：

有根视图把资源管理器变成了项目管理工具。

有根视图特别适用于项目工作。当你使用基于文件或者目录的版本控制系统（例如 Subversion 或者 CVS）时尤其有用。打开有根的资源管理器，你的项目文件和目录就一览无遗。在有根视图中的任何位置，你都可以使用 Tortoise（注 6）插件（一个在资源管理器中的 Subversion 管理工具）。更要紧的是，你无须再被那些与项目全然无关的目录和文件打扰了。

注 6：　可以在 *http://tortoisesvn.tigris.org/* 下载。

OS X 的有根视图

Mac OS X 上的有根视图稍微有些不同。尽管不能把 Finder 限制在一个目录内（就像 Windows 上的有根资源管理器那样），不过你还是可以创建有根视图，从而穿过复杂的目录结构直达目的地。在 Finder 里，你可以把希望到达的目录拖到边栏或者 dock 上，这样就能创建目录的快捷方式，从而可以在 Finder 中直接打开这些目录（如图 3-3）。

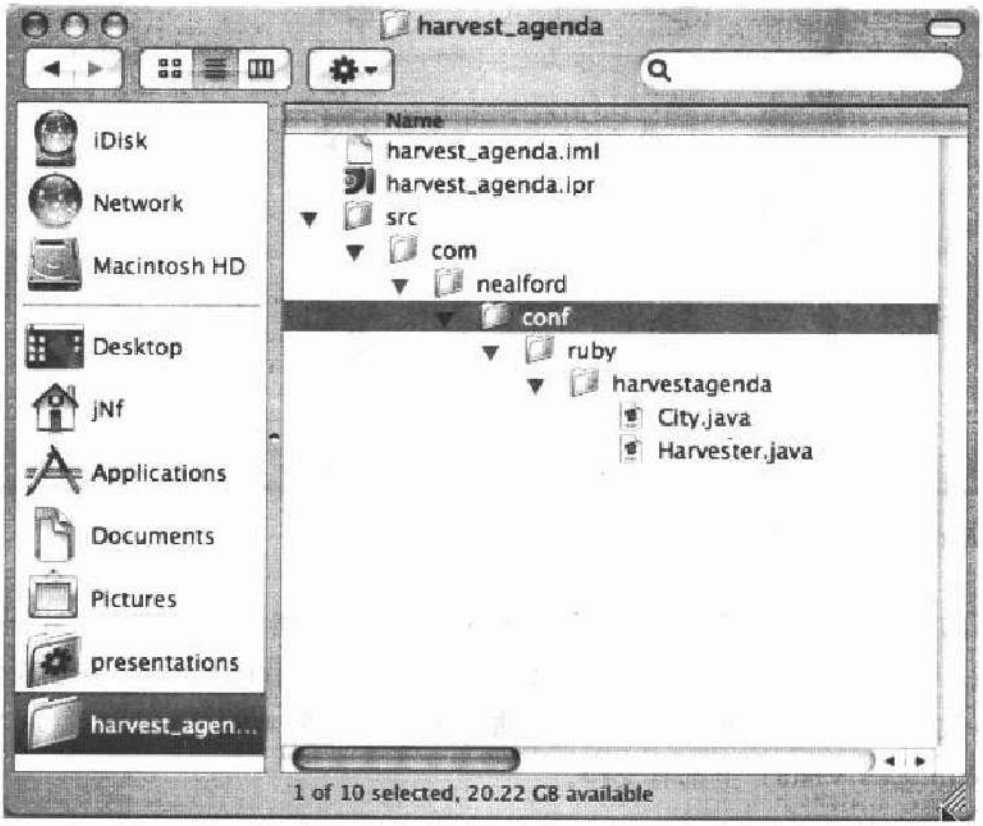


图 3-3：Finder 中的有根视图

设好“粘性属性”

Windows的命令行有一个特别恶心的默认属性：快速编辑模式。这个属性决定了是否可以在命令行窗口中用鼠标选择文本。默认设置下，只要点击命令行窗口就会开始一次拖拽操作，高亮显示选中的文本，并准备将其复制到剪切板（奇怪的是，标准的快捷键 Ctrl-C在这里不管用，你需要敲回车键来复制文本）。这里有个问题：由于是在命令行窗口中选择文本，只要你开始拖动鼠标，所有其他操作（也就是该窗口中所有的进程和线程）都会被冻结住。这也是有道理的：要想复制一段不断翻滚的文本会很烦人。不过，一般而言，在一个多窗口环境中你总是可以放心地点击一个窗口的任意位置使之获得焦点；但如果你点击的目标是命令行窗口，就会在不经意间发起一次拖拽操作，进而冻结

所有进程。当你想要看看命令行窗口在干什么时，你的点击可能无意间让这个窗口停下脚步什么都不干，而你就像丈二和尚摸不着头脑。想要给窗口焦点，却让你失去了工作的焦点，这可真是一种讽刺。还好，可以借助“粘性属性”来摆脱这个“功能”。

提示：

充分利用内建的机制（例如颜色）来帮助你集中注意力。

Windows会根据命令行窗口的标题来跟踪其中的自定义设置。当你关闭命令行窗口时，Windows就会问你是否要保存这些设置以供将来同样标题的窗口使用（于是这些属性就有了“粘性”）。利用这个功能，可以创建特别的命令行窗口：创建一个快捷方式来打开一个指定标题的命令行窗口，设置属性，然后在关闭窗口时保存这些属性。对于开发工作，建议对命令行作如下设置：

- 近乎无限的滚屏长度。默认的滚屏长度只有可怜的300行，一旦你想干点什么有意思的事，信息很容易就滚出了命令行窗口之外。把这个值设为9999行（这时会有一个来自19世纪90年代的警告说你的命令行窗口会用掉2MB内存，不用理它）。
- 在不出现水平滚动条的前提下尽可能加大宽度。在命令行中阅读折行的信息既烦人又容易出错。
- 指定位置。如果某个命令行窗口只用来干一件事（例如开启Servlet引擎，或是Ant/Nant/Rake的执行窗口），它就应该总是出现在同一个位置，这样你可以很快习惯这个位置，连看也不用看就可以知道这个命令行窗口是为干什么而开的。
- 设置独一无二的前景和背景颜色。对于常用的命令行窗口（例如Servlet引擎），颜色成为了醒目的标志，让你一眼看清窗口的用途：你能马上指出绿底黄字的窗口是Tomcat，蓝底绿字的窗口是MySQL。当你在一大堆命令行窗口间穿梭时，靠颜色（还有位置）来判断这些窗口的用途比细看其中的文字要快得多。
- 还有，当然了，记得关掉快速编辑模式。

使用基于项目的快捷方式

所有主流操作系统都支持别名、链接或者快捷方式。我们都见过这样的项目，它们的文档在硬盘上散布得到处都是：需求/用例/故事卡放在一处，源代码在另一处，数据库定义又在别处。在这些目录之间来来往往就是在浪费时间。你没必要强行把所有跟项目相关的文件都放到一个地方，只要让它们“看起来在一起”就行了。可以为这个项目建立

一个文件夹，在其中保存整个项目的快捷方式和链接，你会发现自己在文件系统中四下游逛的时间大大减少了。

提示：

用链接来创建虚拟的项目管理目录。

把项目管理目录放在Windows的“快速启动”菜单或者Mac OS X的dock上。这两块区域放不下太多东西，不过用来放几个项目入口目录倒是挺合适的。

使用多显示器

显示器越来越便宜，完全应该给程序员们配置更好的装备。说实话，要是还舍不得给程序员配备最快的电脑和双显示器，那才叫丢了西瓜拣芝麻。像程序员这样的知识工作者，哪怕只用一丁点时间盯着屏幕上的沙漏，那都是生产率的净损失。同样，在狭小的显示器上管理彼此重叠的窗口也是在浪费时间。

使用多显示器，你可以在一个显示器上写代码在另一个显示器上调试，或是在编程的同时始终把文档放在旁边。不过多显示器只是第一步，因为你还可以利用虚拟桌面把这两个工作空间各自变成一组具有特定功能的视图。

用虚拟桌面拆分工作空间

提示：

虚拟桌面可以让原本杂乱无章的一大堆窗口变得整洁。

虚拟桌面是来自UNIX世界的一个绝妙功能。虚拟桌面就跟普通的桌面一样，上面摆着各样的窗口。不过既然说它们是“虚拟”的，也就是说这样的桌面可以有多个。无须再把IDE、数据库命令行、邮件、聊天软件、浏览器都堆在可怜的桌面上了，你可以把一组逻辑上相关的操作放进一个专用的桌面。桌面上摆着一大堆杂乱的窗口会扰乱你的注意力，因为你经常得去整理这些窗口。

曾经只有高端UNIX工作站才有虚拟桌面（因为这些机器才有足够的图形处理能力来支持这种东西）。不过现在所有主流平台都支持虚拟桌面了。在Linux上，GNOME和KDE都内建虚拟桌面支持。

Mac OS X的Leopard版本（10.5版）也增加了这一功能，叫做Spaces。不过老版本的Mac OS X用户也没有被抛弃：有几个开源或商业的虚拟桌面软件可供选择，例如VirtueDesktops（注7）。它提供了一些很强大的功能，例如把应用程序窗口“钉”在某个桌面上（于是这个应用程序就一定会在这个桌面上出现，当你选中这个应用程序时，你的注意力也会同时转移到这个桌面）。对于程序员来说，这是个很贴心的功能，因为我们经常用一个桌面来专门做一件事（开发、写文档、调试等）。

我们最近有个项目用了Ruby on Rails，我们用Mac Mini（比面包盒子还小、需要外接显示器和键盘的机器）来结对编程。出人意料，Mac Mini是绝佳的开发用机，尤其是配上两套键盘鼠标和显示器以后。但使得它们成为绝佳开发环境的关键还是在于虚拟桌面。我们把所有机器都设置成一样（以便在交换结对时保持使用习惯）：所有开发工具在一个桌面上，文档在另一个桌面上，第三个桌面用来运行应用程序（一个命令行窗口以debug模式运行着Web服务器，还有一个打开的浏览器）。每个桌面都是完全自足的，在切换应用程序时对应的桌面也会旋转上来。有了这样一个环境，我们就可以把所有需要用到的窗口一直开着，把它们放在各自的桌面上，几乎不会有堆叠的情况发生。在另一个项目里，我独自一人在Windows上编程，但我还是设置了“交流”、“文档”、“开发”等桌面，这让我的工作环境不再混乱，让我时刻保持头脑清醒。

Windows PowerToy里有一个叫做Virtual Desktop Manager的工具，可以在Windows 2000和XP中支持虚拟桌面（如图3-4）。你可以用它来管理最多4个虚拟桌面，每个桌面都有自己独立的任务栏、墙纸和热键。虚拟桌面并不是对底层操作系统的根本改变，它只是在背后管理窗口的展现和状态而已。

虚拟桌面能够很好地帮你集中注意力。它们总是在你需要的时候给你恰到好处的信息和工具，没有多余的打扰。我总是根据工作的需要来创建虚拟桌面。Spaces和Virtual Desktop Manager都提供了一个很好的功能：你可以总览所有的桌面。当我有一个单独的任务需要完成时，我就会打开执行这个任务所需的应用程序，然后把它们都挪到同一个桌面去。这样一来，我就可以专心工作于这个项目，而不受电脑上其他东西的打扰。实际上，这一章文字就是在我的2号桌面上敲出来的。

注7：　可以在*http://virtuedesktops.info/*下载。

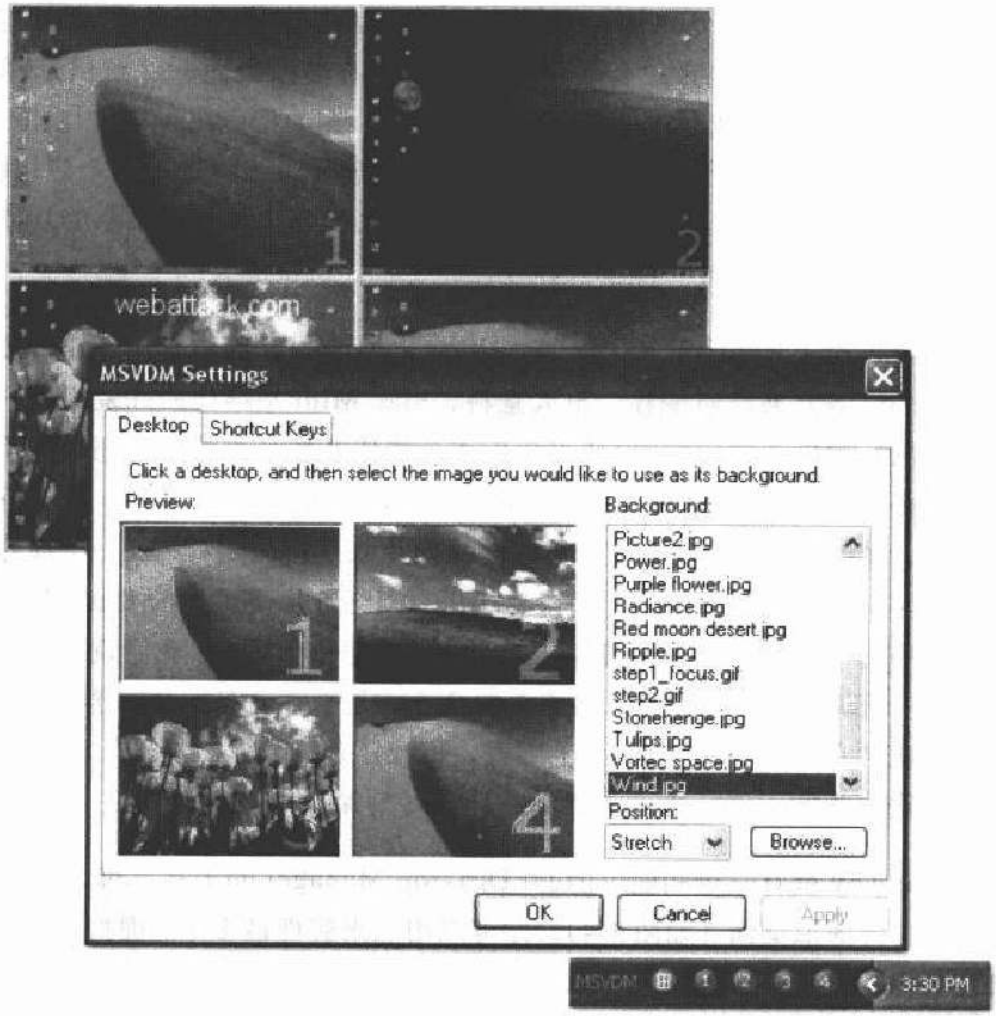
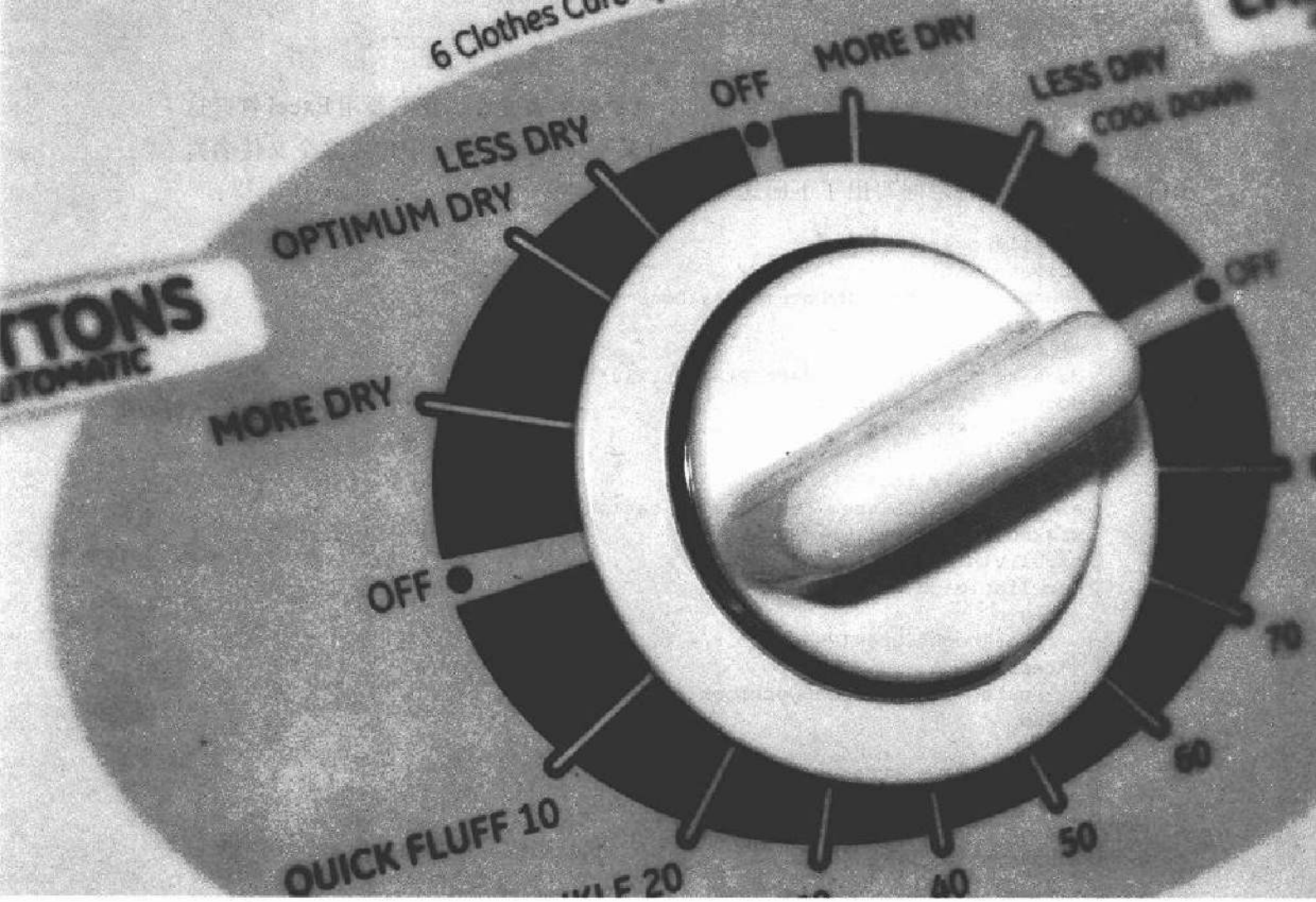


图 3-4：用 Virtual Desktop Manager 管理桌面

小结

本章涵盖了“专注”的几个方面：想办法改变环境以减少打扰，减少来自电脑的打扰，以及借助工具保持专注。现在你应该能明白为什么我要围绕着“生产率原则”来组织这些内容了：要不是有“专注”这条主线贯穿着，这些主题之间看起来根本毫不相干。

现代社会让我们很难保持专注。但为了充分发挥潜力，你必须根据周围的情况创造出一个能让自己专心工作的空间和环境来，这将极大地提高你的生产率。


OFF
MORE DRY
LESS DRY
LESS DRY
OPTIMUM DRY
MORE DRY
OFF
QUICK FLUFF 10
50
70

第 4 章

自动化法则

在从前的一个项目中，我们需要定时更新几个电子数据表文件。我需要用Excel打开这些文件，但手工做这件事实在费劲（偏偏Excel又不允许在命令行中传入多个文件名）。所以，我花了几分钟时间写出了下面这段Ruby小脚本：

```
class DailyLogs
    private
    @@Home_Dir = "c:\\MyDocuments\\Documents\\"
  def doc_list
    docs = Array.new
    docs << "Sisyphus Project Planner.xls"
    docs << "TimeLog.xls"
    docs << "NFR.xls"
  end
  def open_daily_logs
    excel = WIN32OLE.new("excel.application")
    workbooks = excel.WorkBooks
    excel.Visible = true
    doc_list.each do |f|
      begin
        workbooks.Open(@@Home_Dir + f, true)
      rescue
        puts "Cannot open workbook:", @@Home_Dir + f
      end
    end
    excel.Windows.Arrange(7)
  end
end
DailyLogs.daily_logs
```

虽说手工打开这几个文件花不了多少工夫，但那也是在浪费时间，所以我将这件事自动化了。而且我还从中学到一点东西：在Windows上可以用Ruby来驱动COM对象，比如Excel。

计算机原本就该从事简单重复的工作。只要把任务指派给它们，它们就可以一遍又一遍毫不走样地重复执行，而且速度很快。但我却经常看见一种奇怪的现象：人们在计算机上手工做一些简单重复的工作，计算机们则在大半夜里扎堆闲聊取笑这些可怜的用户。怎么会这样？

图形化环境是用来帮助新手的。微软在Windows桌面的左下角放上一个大大的“开始”按钮，因为很多刚从旧版本切换过来的用户不知道该怎么开始操作。（奇怪的是，关机操作也是从“开始”按钮开始的。）但正是这些提高新手效率的东西却恰巧成为高级用户的束缚：比如软件开发中的各种琐事，在命令行里处理几乎一定比图形用户界面来得快。在过去20年里，高级用户执行常规任务的速度反而降低了，这不能不说是一个大大的反讽。过去那些典型的UNIX用户比如今习惯了图形界面的用户更高效，因为他们把一切都自动化了。

要是你去过老木匠的作坊，一定会看见那儿到处都摆着专用的工具（说不定还有一台激光定位螺旋平衡的车床，只是你认不出来）。尽管有那么多工具，在大部分项目里，木匠师傅还是会从地上找一两块边角废料，临时用来隔开两个零件、或是把两个零件夹在一起。按照工程学术语，这样的小块材料称为“填木”或是“夹铁”。作为软件开发者，我们很少创造这种小巧的小工具，很多时候是因为我们还没意识到这些也是工具。

软件开发中有很多显而易见的东西需要自动化：构建、持续集成和文档。本章会介绍一些不那么显而易见、价值却毫不逊色的自动化方法。小到一次敲击键盘，大到小规模的应用程序，我们都会有所介绍。

不要重新发明轮子

每个项目都需要准备一些通用的基础设施：版本控制、持续集成、用户账号之类的。对于Java项目，Buildix（注1）（ThoughtWorks开发的一个开源项目）能够大大简化这些准备工作。很多Linux发行版本会提供“Live CD”选项，用这张CD就可以直接试用这个Linux版本。Buildix也采用了这种发行方式，只不过加上了预先配置好的项目基础设施：它本身其实是一张Ubuntu Live CD，预装了一些软件开发中常用的工具。Buildix预装了下列工具：

- Subversion，流行的开源版本控制工具
- CruiseControl，开源的持续集成服务器
- Trac，开源的问题跟踪和wiki工具
- Mingle，ThoughtWorks推出的敏捷项目管理工具

用Buildix CD启动，你的项目基础设施就准备好了。你也可以用这张CD在已有的Ubuntu系统上进行安装。

建立本地缓存

在开发软件时你经常需要到互联网上查资料。不管网络速度有多快，在互联网上浏览网页终究是要花时间的。所以，对于经常查阅的资料（例如编程API），你应该把它缓存到本地（这样你在飞机上也可以看了）。有些东西很容易保存，只要在浏览器里保存页面就行了；但很多时候你需要保存一大堆的网页，这时就应该求助于工具。

注1：可以在 *http://buildix.thoughtworks.com/* 下载。

*wget*是一个*-nix工具，用于将互联网上的内容保存到本地。在所有*-nix平台上都可以找到这个工具，在Windows上也可以通过Cygwin找到它。在抓取网页方面，*wget*有很多选项，其中最常用的是*mirror*选项：将整个网站镜像到本地。比如说，下列命令就可以在本地创建一个网站的镜像：

```
wget --mirror -w 2 --html-extension --convert-links -P c:\wget_files\example1
```

表4-1：使用wget命令

字符（串）	含义
wget	启动wget工具
--mirror	给网站建立本地镜像。wget会递归地跟踪网站上的链接，下载所有需要的文件。默认情况下，它只会下载上次镜像操作之后有更新的文件，以避免做无用功
--html-extension	很多网站使用非HTML的文件扩展名（例如cgi或是php），实际上它们最终也生成HTML页面。这个选项告诉wget，应该把这些文件的扩展名改为HTML
--convert-links	把页面上所有的链接转为本地链接，以免因为页面上有指向绝对URI的链接而导致页面无法使用。wget会转换页面上所有的链接，使其指向本地资源
-P c:\wget_files\example1	指定保存网站镜像的本地目录

自动访问网站

有些网站需要你登录或是做些别的操作才能得到你需要的信息，cURL能帮你自动化这些交互。cURL也是一个开源工具，有所有主要操作系统的安装版本。它和wget有些相似，不过更偏重于与页面交互以获取内容或是抓取资源。例如，假设网站上有以下表单：

```
<form method="GET" action="junk.cgi">
        <input type=text name="birthyear">
        <input type=submit name=press value="OK">
</form>
```

cURL就可以提供两个参数，获得表单提交后的结果：

```
curl "www.hotmail.com/when/junk.cgi?birthyear=1905&press=OK"
```

在命令行输入“-d”参数，就可以用HTML POST方法（而非默认的GET方法）与页面交互：

```
curl -d "birthyear=1905&press=%20OK%20" www.hotmail.com/when/junk.cgi
```

cURL 最妙的一点是它能用各种协议（例如HTTPS）与加密的网站交互。cURL 网站上有关于这方面的详细介绍。能够使用安全协议访问网站，再加上其他功能，使得cURL成为了与网站交互的绝佳工具。Mac OS X和大部分Linux发行版本都默认安装了cURL，也可以在 *http://www.curl.org* 网站下载它的 Windows 版本。

与 RSS 源交互

Yahoo!有一个名叫 Pipes 的服务（一直处于 beta 状态，目前还是），可以用来操作 RSS 源（例如 blog）。你可以组合、过滤和处理 RSS 信息，用于创建网页或是新的 RSS 源。Pipes 借鉴了 UNIX 的“命令行管道”的概念，提供了一个基于 Web 的拖拽界面来创建 RSS“管道”：信息从一个 RSS 源流向另一个。从使用的角度来说，这个功能很类似于 Mac OS X 的 Automator 工具 —— 每个命令（或者管道中的一个阶段）的输出成为下一个管道的输入。

举例来说，图 4-1 所示的管道会抓取“No Fluff，Just Stuff”会议网站的 blog 聚合 RSS，后者包含了最近的 blog 文章。每个 blog 条目都遵循“作者－标题”的格式，而我又只想要其中的作者信息，所以我用了一个正则表达式管道将“作者－标题”替换成作者姓名。

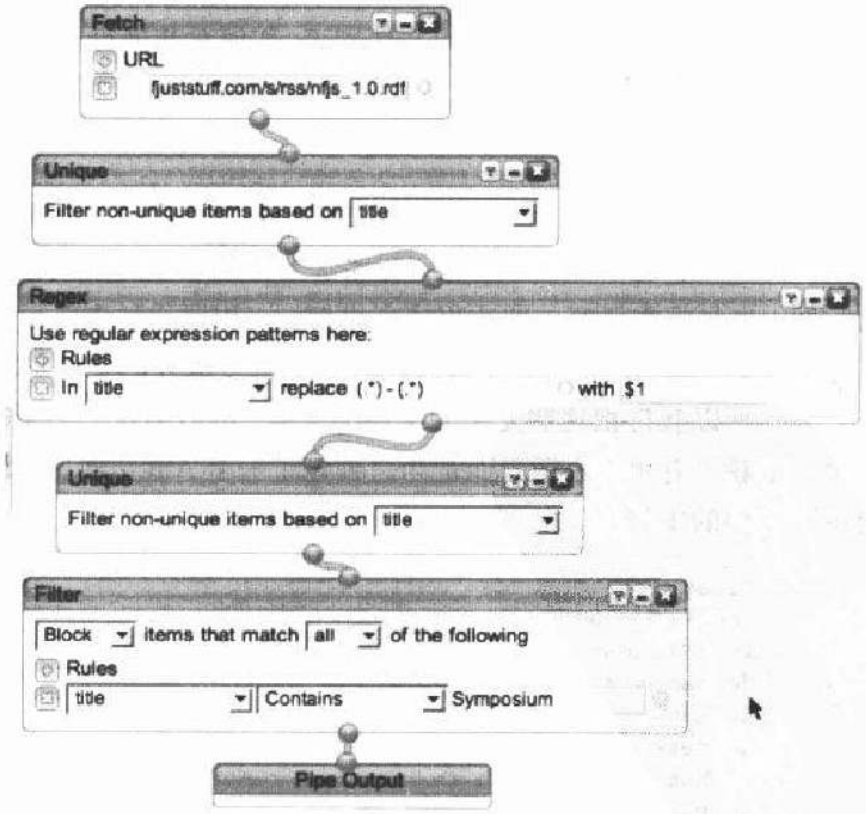


图 4-1：实用正则表达式管道

管道的输出可以是一个HTML页面，也可以是另一个RSS源（从中获取RSS时就会导致管道被执行）。

RSS日益成为一种重要的信息格式，尤其是对于软件开发者们关注的那些信息，而Yahoo! Pipes则允许你以编程的方式处理RSS，对信息加以提炼。不仅如此，Pipes还逐渐加上了"从网页上采集信息"的功能，从而可以自动获取各种基于Web的信息。

在构建之外使用Ant

> **提示：**
>
> 即便不是工具最初的设计意图，只要是合适的场合，同样可以使用这些工具。

在操作系统的层面上，批处理文件和bash脚本都可以用于将工作自动化，但这两者的语法都不够灵活，命令也常常显得笨拙。举例来说，如果你需要对一大批文件执行某一操作，用批处理文件或是bash脚本的原生命令想要得到这些文件的列表就是一件麻烦事。为什么不用专门为此设计的工具呢？

其实我们常用的构建工具已经知道如何获取文件列表、如何对其进行过滤或是执行别的操作。在对文件进行批量操作这件事情上，Ant、Nant和Rake的语法都比批处理文件或是bash脚本要友好得多。

下面就是用Ant来处理文件的一个例子——这用批处理做起来实在太难，以致于我根本就不打算尝试。我曾经教过很多编程课程，在课堂上常会写一些示例程序。为了解答学生们提出的问题，也经常需要写一些小程序来讲解。一周课程结束以后，所有学生都想要复制我在课上写的小程序。但这些小程序还产生了很多额外的文件（输出文件、JAR文件、临时文件等），所以我得把这些无关的文件清理掉，然后创建一个干净的ZIP包复制给学生们。这件事我没有手工去做，而是为之创建了一个Ant文件。Ant的一大好处就是它内建对批量文件的支持：

```
<target name="clean-all" depends="init">
    <delete verbose="true" includeEmptyDirs="true">
    <fileset dir="${clean.dir}">
        <include name="**/*.war" />
        <include name="**/*.ear" />
        <include name="**/*.jar" />
        <include name="**/*.scc" />
        <include name="**/vssver.scc" />
        <include name="**/*.*~" />
```

```
            <include name="**/*.~*~" />
            <include name="**/*.ser" />
            <include name="**/*.class" />
            <containsregexp expression=".*~$" />
        </fileset>
        </delete>
        <delete verbose="true" includeEmptyDirs="true" >
            <fileset dir="${clean.dir}" defaultexcludes="no">
                <patternset refid="generated-dirs" />
            </fileset>
        </delete>
    </target>
```

在Ant的帮助下，我可以写一个高级任务来执行以前手工完成的那些步骤：

```
<target name="zip-samples" depends="clean-all" >
    <delete file="${class-zip-name}" />
    <echo message="Your file name is ${class-zip-name}" />
    <zip destfile="${class-zip-name}.zip" basedir="." compress="true"
    excludes="*.xml,*.zip, *.cmd" />
</target>
```

同样的事情要用批处理文件写出来，不啻是一场噩梦！哪怕用Java写都会很麻烦：Java并没有内置对批量文件进行模式匹配的支持。使用构建工具，你不需要创建main方法，也无须操心太多基础设施的问题，因为构建工具通常已经提供了足够的支持。

Ant最糟糕的一点莫过于对XML的依赖，这使得Ant脚本既难写又难读，同样难以重构，连diff都变得困难。Gant（注2）是一个不错的替代品：它能够与已有的Ant任务交互，但它的构建脚本是用Groovy编写的——和XML不同，那是一种真正的编程语言。

用Rake执行常见任务

Rake就是Ruby的make工具（同时它本身也是用Ruby编写的）。Rake是shell脚本的完美替代品，因为它不仅具备了Ruby强大的表现力，而且能够与操作系统轻松交互。

下面这个例子是我经常使用的。我在各种开发者大会上作很多演讲，这也就意味着我有很多幻灯片和对应的示例代码。长期以来，我总是先打开幻灯片，然后凭记忆打开其他需要打开的工具和示例。难免有时，我会忘记打开某个示例，于是就不得不在演讲中途手忙脚乱地翻找。有这么几次经验之后，我意识到这个问题，于是把这个过程自动化了：

```
require File.dirname(__FILE__) + '/../base'
TARGET = File.dirname(__FILE__)
```

注2：在*http://gant.codehaus.org/*下载。

```
FILES = [
  "#{PRESENTATIONS}/building_dsls.key",
  "#{DEV}/java/intellij/conf_dsl_builder/conf_dsl_builder.ipr",
  "#{DEV}/java/intellij/conf_dsl_logging/conf_dsl_logging.ipr",
  "#{DEV}/java/intellij/conf_dsl_calendar_stopping/conf_dsl_calendar_stopping.ipr",
  "#{DEV}/thoughtworks/rbs/intarch/common/common.ipr"
]
APPS = [
  "#{TEXTMATE} #{GROOVY}/dsls/",
  "#{TEXTMATE} #{RUBY}/conf_dsl_calendar/",
  "#{TEXTMATE} #{RUBY}/conf_dsl_context"
]
```

这个Rake文件列出所有我需要打开的文件，以及在演讲中需要用到的应用程序。Rake的妙处之一在于它可以使用Ruby文件来作为辅助。前面这个Rake文件本身实际上只起声明作用，实际的工作都在名为base的Rake文件中实现了，别的Rake文件则依赖于它。

```
require 'rake'
require File.dirname(__FILE__) + '/locations'
require File.dirname(__FILE__) + '/talks_helper'

task :open do
  TalksHelper.new(FILES, APPS).open_everything
end
```

可以看到，在这个文件的顶部，我又引入了一个名为task_helper的文件：

```
class TalksHelper
  attr_writer :openers,:processes
  def initialize(openers, processes)
    @openers, @processes = openers, processes
  end

  def open_everything
    @openers.each { |f| `open #{f.gsub /\s/, '\\ '}` } unless @openers.nil?
    @processes.each do |p|
      pid = fork {system p}
      Process.detach(pid)
    end unless @processes.nil?
  end
end
```

实际起作用的代码都在这个辅助类里。这样一来，我就可以为每次演讲写一个简单的Rake文件，然后就可以自动打开所有我需要的东西。Rake的一大优势是能够非常轻松地与操作系统交互。只要把一个字符串用反引号（`）括起来，这句话就会被当作shell命令被执行。所以，包含了`open #{f.gsub /\s/, '\\ '}`的这行代码实际上是在操作系统层面上执行了*open*命令（我的操作系统是Mac OS X，对于Windows可以替换成*start*命令），并传入前面定义的变量作为参数。用Ruby驱动底层操作系统比编写bash脚本或者批处理文件要容易多了。

用 Selenium 浏览网页

Selenium（注3）是一个开源的测试工具，用于Web应用程序的用户验收测试。Selenium借助JavaScript自动化了浏览器操作，从而可以模拟用户的行为。Selenium完全是用浏览器端技术编写的，所以在所有主流浏览器中都能使用。这是一个极其有用的Web应用程序测试工具，不论被测的Web应用程序是用什么技术开发的。不过现在我不是来介绍怎么用Selenium做测试的。Selenium有一个叫做Selenium IDE的派生项目，那是一个Firefox浏览器插件，能将用户与网站的交互记录为Selenium脚本，然后可以用Selenium的TestRunner或者Selenium IDE运行这些脚本。这个工具在用来创建测试时非常有用，而如果你需要将你与网站的交互自动化，那它更是一个无价之宝。

下面就是一个常见的情景：假设你要开发一个向导形式的Web应用，现在前三个页面都已经完成，它们的行为（包括输入校验等）都运转良好，你正在编写第四个页面。为了调试这个页面上的行为，你必须反复经过前三个页面，一遍，又一遍，又一遍……你不断地对自己说："好吧，这是最后一遍了，这次我肯定能把问题都解决掉。"但似乎永远没有最后一次！要不然，你的测试数据库里怎么会充斥着那么多"Fred Flintstone"、"Homer Simpson"，还有在焦躁中输入的"ASDF"之类的名字？

Selenium IDE可以帮你做这些烦琐的事情。你只要点击导航来到第四个页面，用Selenium IDE把你的操作记录下来（就像图4-2这样）。下一次当你需要到达第四个页面并填入合法的输入值时，只要回放这段Selenium脚本就行了。

顺理成章地，Selenium的另一个绝妙用途也浮出水面了。当QA部门的同事发现一个bug时，他们通常会用某些原始的方法来记录bug出现的情况：写下他们的操作，附上一张模糊不清的截屏图或是别的什么帮不上忙的东西。现在，请让他们用Selenium IDE把发现bug的过程记录下来，然后提交他们的Selenium脚本；然后你就可以反复地、毫厘不差地重复他们的操作步骤，直到bug被修复为止。这不仅节约时间，而且省了很多麻烦。Selenium可以把用户与网站的交互变成一段可执行的脚本，请使用它吧！

> **提示：**
>
> 不要浪费时间动手去做可以被自动化的事情。

用 bash 统计异常数

这里有一个使用bash的例子，你可能会在一个典型的项目中遇到类似的情况。当时我在

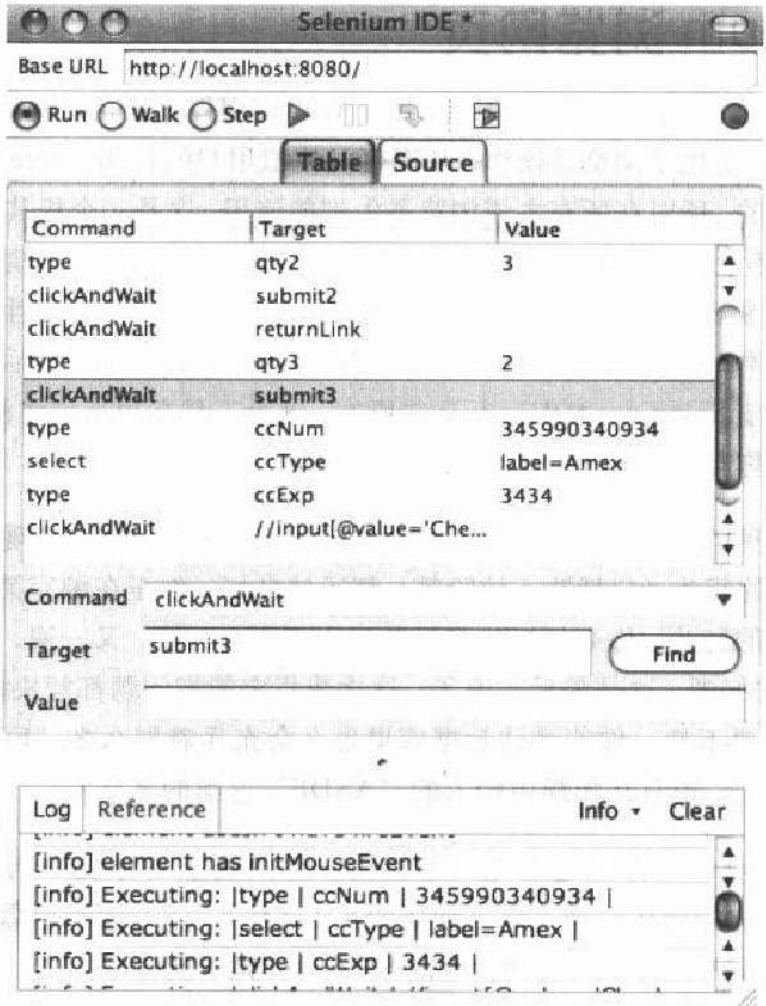


图 4-2：Selenium IDE 和录制好的脚本

一个已经有6年历史的大型Java项目中工作（我只是一个访客，在第6年进入这个项目，并在上面工作了大概8个月）。我的任务之一就是清理一些经常发生的异常，为此我做的第一件事就是提问："哪些异常会被抛出？以什么样的频率？"当然了，没人知道，所以我的第一个任务就是自己动手找到答案。但问题是这个应用程序每星期会生成出超过2 GB的日志，很快我就意识到：即便只是尝试用文本编辑器打开这个文件，那都是在浪费时间。于是我坐下来，写了这么一段脚本：

```
#!/bin/bash
for X in $(egrep -o "[A-Z]\w*Exception" log_week.txt | sort | uniq) ;
do
    echo -n -e "processing $X\t"
    grep -c "$X" log_week.txt
done
```

表 4-2 解释了这段 bash 小脚本的作用。

表 4-2：用于统计异常数量的复杂 bash 命令

字符（串）	用途
`egrep -o`	找出日志文件中出现在“Exception”字眼之前的文字，对它们进行排序，然后得到一个消除重复之后的列表
`"[A-Z]\w*Exception"`	用于定位异常信息的正则模式
`log_week.txt`	庞大的日志文件
`| sort`	将前面的查找结果管道给 sort，生成一个排序后的异常列表
`| uniq`	去掉重复的异常信息
`for X in $(. . .) ;`	针对前面生成的异常列表中的每个异常循环执行这些代码
`echo -n -e "processing $X\t"`	把找到的异常输出到控制台上（这样我才知道这段脚本还在工作）
`grep -c "$X" log_week.txt`	在庞大的日志文件中找出这个异常出现的次数

这个项目到现在还在使用这段小程序。这是一个好例子：借助自动化工具，你可以从项目中找出一些从未有人发现的、有价值的信息。与其绞尽脑汁地猜测有哪些异常被抛出，不如把它们都找出来，这样也可以更有目的性、更容易地修复这些抛出异常的程序。

用 Windows Power Shell 替代批处理文件

作为 Windows Vista 的一部分，Microsoft 对批处理语言作了大幅度的改进。新的批处理环境代号叫 Monad，不过在发行版中叫做 Windows Power Shell。（为了避免每次都写出这么长的名字，为了节约纸张保护树木，我打算继续称它为“Monad”。）它是内建在 Windows Vista 中的，如果你想在 Windows XP 上使用，可以从 Microsoft 网站下载。

Monad 从 bash 和 DOS 等类似的命令行 shell 语言中借鉴了很多理念，例如你可以把一个命令的输出管道给另一个命令作为输入。它们之间最大的区别在于 Monad 不是使用文本（像 bash 那样），而是使用对象：Monad 的命令（称为 cmdlet）知道一组代表操作系统构造的对象，像文件、目录、甚至 Windows 事件查看器（event viewer）之类的东西。Monad 的语义和 bash 大同小异（管道操作符还是那个历史悠久的“|”符号），但它的能力实在强大。下面就是一个例子：假设你需要把在 2006 年 12 月 1 日以后更新过的文件复制到 DestFolder 目录中，相应的 Monad 命令大概会是这样：

```
dir | where-object { $_.LastWriteTime -gt "12/1/2006" } |
    move-item -destination c:\DestFolder
```

由于Monad的cmdlet能“理解”其他cmdlet，也能“理解”它们输出的东西，所以Monad的脚本可以写得比其他脚本语言更简洁。例如，假设你想要关闭所有占用超过15 MB内存的进程，用bash来做大概是这样：

```
ps -el | awk '{ if ( $6 > (1024*15)) { print $3 } }'
  | grep -v PID | xargs kill
```

这可不怎么好看！这里用到了5个不同的bash命令，包括用*awk*来解析*ps*命令的结果。再看看用Monad怎么做同样的事：

```
get-process | where { $_.VS -gt 15M } | stop-process
```

针对*get-process*的结果，你可以用*where*命令来根据某个特定属性加以过滤（在这里我们根据VS属性过滤，也就是占用内存的大小）。

Monad是用.NET编写的，也就是说，你还可以使用标准的.NET类型。字符串处理对于命令行shell来说一直是个难题，现在你可以用.NET的String类来解决这个问题了。例如，下列Monad命令：

```
get-member -input "String" -membertype method
```

会输出`String`类的所有方法。这就有点像在*-nix中使用*man*工具一样。

Monad是Windows世界的一大进步。它把操作系统层面上的编程当作一等公民来对待，很多原本需要求助于Perl、Python和Ruby等脚本语言的任务现在可以用Monad轻松完成。由于它是操作系统核心的一部分，你还可以用它来查找和操作操作系统特有的对象（例如事件查看器）。

用Mac OS X的Automator来删除过时的下载文件

Mac OS X提供了一种以图形化方式编写批处理文件的途径，那就是Automator。从很多方面来说，这简直就是一个图形化版本的Monad，而且比后者早出现了几年。要创建一个Automator工作流（也就是Mac OS X版本的脚本），只要从Automator的工作区拖拽命令，然后把命令之间的输入输出“连接”起来就行了。每个应用程序在安装时都会向Automator注册，告诉后者自己能做什么。还可以用Objective C（Mac OS X底层的开发语言）来编写代码扩展Automator。

下面是一个Automator工作流的例子：删除所有在硬盘上放了超过两周的下载文件。这个工作流（如图4-3所示）由以下步骤组成：

1. 这个工作流会把最近两周的下载文件暂存在 *recent* 目录下。
2. 清空 *recent* 目录，为保存新下载的文件作好准备。
3. 找出所有修改日期在两周以内的下载文件。
4. 把上述文件移到 *recent* 目录下。
5. 找出 *downloads* 目录下所有非目录的文件。
6. 删除上述文件。

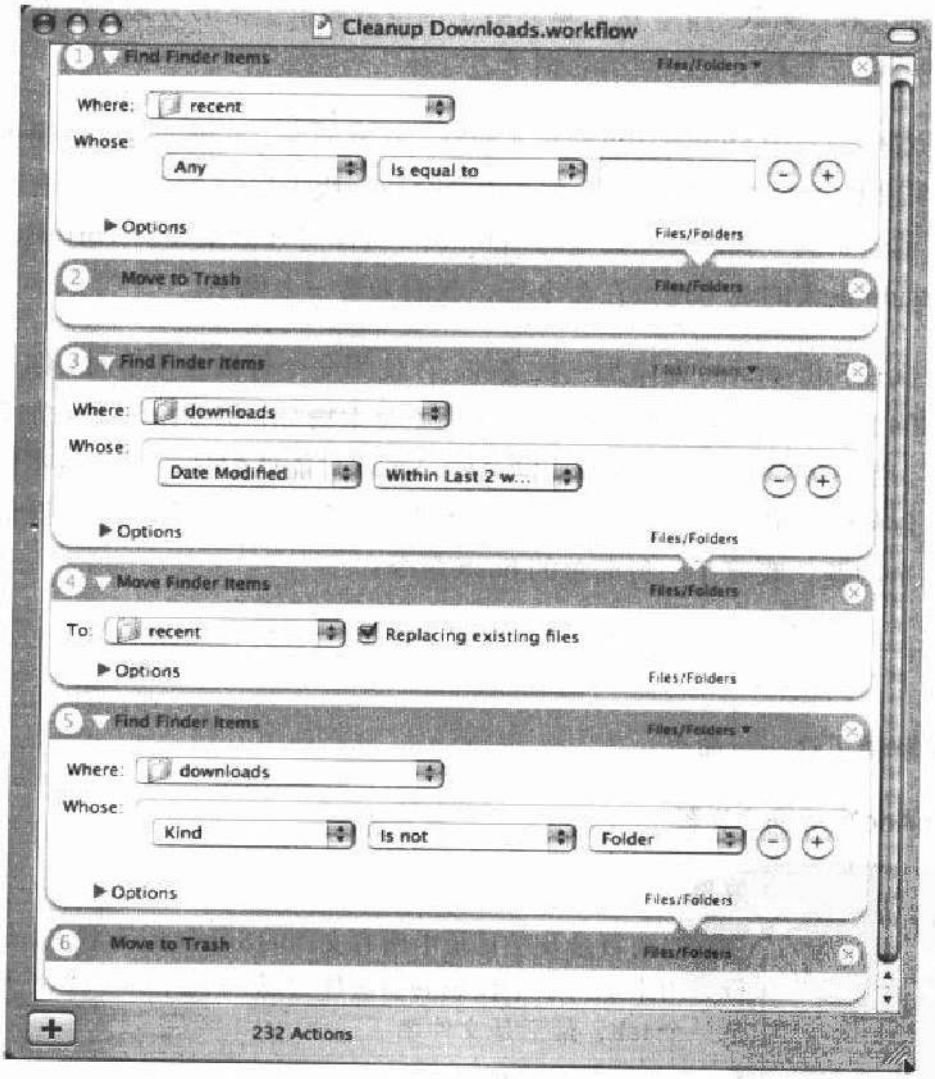


图 4-3：用于删除过时下载文件的 Mac OS X Automator 工作流

这个工作流比前面的Monad脚本做了更多的工作，因为在工作流中没有一种简单的办法能描述“在过去两周内没有被修改过的文件”，于是最好的办法就是找出那些在过去两周中被修改过的文件，把这些文件移到一个暂存目录（也就是 recent 目录），然后删除

downloads 目录下所有的文件。你永远也不会手工去干这些麻烦事，但由于这是自动化工具，做些额外工作也无妨。另一种办法是编写一段 bash shell 脚本，然后在工作流中使用它（可以调用这段 bash 脚本），但这样一来你又回到了“解析 shell 脚本的结果来找出文件名”的老问题上。如果你真的要这么做，还不如把整件事情都用 shell 脚本解决。

驯服 Subversion 命令行

总有些时候，没有别的什么工具或是开源项目能恰好满足你的需要，这时就该你自己动手制造“填木”和“夹铁”了。这一章介绍了很多种制造工具的方式，下面就是一些在真实项目中用这些工具来解决问题的例子。

我是开源版本控制系统 Subversion 的忠实粉丝。在我看来，它就是强大、简单和易用的完美结合。归根到底 Subversion 是一个基于命令行的版本控制系统，不过有很多开发者为它开发了前端工具（我最喜欢的是与 Windows 资源管理器集成的 Tortoise）。尽管如此，Subversion 最大的威力还是在命令行，我们来看一个例子。

我经常会一次往 Subversion 里添加一批文件。在使用命令行做这件事时，你必须指定所有想要添加的文件名。如果文件不多的话这还不算太糟糕，但如果你要添加 20 个文件，那就费事了。当然你也可以用通配符，但这样一来就可能匹配到已经在版本控制之下的文件（这不会有什么损害，只不过会输出一堆错误信息，可能会跟别的错误信息混淆）。为了解决这个问题，我写了一行简单的 bash 命令：

```
svn st | grep '^\?' | tr '^\?' ' ' | sed 's/[ ]*//' | sed 's/[ ]/\\ /g' |
xargs svn add
```

表 4-3 详细解释了这一行命令。

表 4-3：svnAddNew 命令详解

命令	效果
`svn st`	获取当前目录及子目录中所有文件的 Subversion 状态，每个文件一行。尚未加入版本控制的新文件会以一个问号（“?”）开头，随后是一个 tab，最后是文件名
`grep '^\?'`	找出所有以“?”开头的行
`tr '^\?' ' '`	把“?”替换成空格（tr 命令会把一个字符替换为另一个字符）
`sed 's/[ ]*//'`	用 sed（基于流的编辑器）把每行开头的空格去掉
`sed 's/[ ]/\\ /g'`	文件名内部也可能包含空格，所以再用一次 sed，把文件名中的空格替换成转义字符（也就是在空格符前面加上“\”字符）
`xargs svn add`	针对前面处理的结果，逐一调用 svn add 命令

我大概花了15分钟写出这条命令，然后用了它成百上千次。

用 Ruby 编写 SQL 拆分工具

在从前的一个项目中，我和一个同事需要解析一个巨大（38 000行）的遗留SQL文件。为了让解析的工作变得容易一点，我们想把这个大块的文件分成每块1 000行左右的小块。我们稍微考虑了一下手工做这件事，不过很快就明白将其自动化会是更好的办法。我们也考虑用*sed*来实现，不过似乎会很复杂。最终，我们选定了Ruby。大约一个小时以后，我们得到了这段代码：

```
SQL_FILE = "./GeneratedTestData.sql"
OUTPUT_PATH = "./chunks of sql/"

line_num = 1
file_num = 0
Dir.mkdir(OUTPUT_PATH) unless File.exists? OUTPUT_PATH
file = File.new(OUTPUT_PATH + "chunk " + file_num.to_s + ".sql",
    File::CREAT|File::TRUNC|File::RDWR, 0644)

done, seen_1k_lines = false
IO.readlines(SQL_FILE).each do |line|
  file.puts(line)
  seen_1k_lines = (line_num % 1000 == 0) unless seen_1k_lines
  line_num += 1
  done = (line.downcase =~ /^\W*go\W*$/ or
          line.downcase =~ /^\W*end\W*$/) != nil
  if done and seen_1k_lines
    file_num += 1
    file = File.new(OUTPUT_PATH + "chunk " + file_num.to_s + ".sql",
          File::CREAT|File::TRUNC|File::RDWR, 0644)
    done, seen_1k_lines = false
  end
end
```

这个Ruby小程序从源文件中逐行读取，直到读满1 000行为止，然后从中寻找包含*GO*或者*END*的行，如果找到就结束当前文件的查找，开始下一个文件。

我们计算了一下，如果手工强行分解这个文件，大概需要10分钟；而我们将其自动化用了大概1小时。后来我们又把分解的工作做了5次，所以花在自动化上面的时间基本上算是相同。但这不是重点。手工执行简单重复的任务会让你变傻，会消耗你的注意力，而注意力是最重要的生产率之源。

> **提示：**
>
> 做简单重复的事是在浪费注意力。

相反，找出一种聪明的方法来自动化这些任务，这会让你变聪明，因为你能从中学到一些东西。我们之所以用了这么长时间来写这段程序，原因之一是我们不熟悉Ruby的底层文件处理机制。现在我们学会了，于是我们可以把这些知识应用到别的项目。而且我们学会了如何对部分项目基础设施进行自动化，这样将来我们会更有可能找出别的一些方式来自动化简单重复的任务。

提示：

以创造性的方式解决问题，有助于在将来解决类似的问题。

我应该把它自动化吗

有那么一个应用程序，它的部署只需要三个步骤：在数据库上运行“create tables”脚本，把应用程序文件复制到Web服务器上，然后更新配置文件使之反映应用程序的路由变更。很简单的几个步骤。你需要每隔两三天就部署一次，那又怎么样呢？毕竟做这一切只需要15分钟。

假如这个项目持续8个月又如何呢？你要把这个过程重复64遍（实际上到项目后期这个速率还会提高，那时你会更加频繁地部署）。把它加起来：64次 × 15分钟 = 960分钟 = 16小时 = 2工作日。整整两个工作日不干别的，就是一遍又一遍地重复这同一件事！这还没考虑你会因为一时粗心忘记其中的某个步骤，然后不得不花更多的时间来调试和修复错误。所以，只要将其自动化的工作量不超过两天，这笔买卖就根本不需要考虑，因为你完全是在节约时间。但如果需要三天时间来将其自动化呢——还值得去做吗？

我见过一些系统管理员，他们写bash脚本来执行每项任务。这样做的原因有两条。第一，既然你已经做了一次，那么几乎可以肯定以后你还会做同样的事。bash命令本身非常简洁，有时就连有经验的程序员也得花上好几分钟才能弄对；但如果你需要再次执行这个任务，保存下来的命令就能节省你的时间。第二，把一切有用的命令都保存在脚本中，就等于给你做的事情（甚至还可能包括为什么要做这些事）创建了一份活的文档。记录所做的每件事有些极端，但存储设备非常便宜——比起重新创造某些东西所需的时间来要便宜多了。你也可以折中一下：不必保存做过的每件事，不过一旦发现自己第二次做某件事，就将其自动化。一旦某件事需要你做两次，很可能你还需要做100次。

几乎所有*-nix用户都会在自己的.bash_profile配置文件里创建各种别名，给常用的命令行工具创造捷径。下面的例子展示了别名的语法：

```
alias catout='tail -f /Users/nealford/bin/apache-tomcat-6.0.14/logs/
```

```
catalina.out'
alias derby='~/bin/db-derby-10.1.3.1-bin/frameworks/embedded/bin/ij.ksh'
alias mysql='/usr/local/mysql/bin/mysql -u root'
```

所有常用的命令都可以放在这个文件里，这样你就不必费心记住那些复杂的魔法咒语了。实际上，这个功能与键盘宏工具（参见第2章“键盘宏工具”一节）有很大程度的重合。对于大部分命令，我都比较喜欢用bash别名（从而不必进行宏展开），但有那么一种重要的情况会让我使用键盘宏工具：如果一条命令里同时包含双引号和单引号，就很难为它创建别名，因为很难把引号转码弄对。键盘宏工具能更好地处理这种情况。比如说svnAddNew这个脚本（在前面的“驯服Subversion命令行”中展示过了）一开始是一个bash别名，但为了把引号转码弄对几乎把我给搞疯了，所以后来我把它实现为一个键盘宏，生活从此变得轻松。

提示：

是否应该自动化的关键在于投资回报率和缓解风险。

项目中会有很多琐事让你想要自动化，这时你应该先拿下列问题来问自己（并且诚实地作答）：

- 长期来看，将其自动化能节省时间吗？
- 这件任务是否很容易出错（因为其中包含很多复杂的步骤）？一旦出错是否会浪费大量时间？
- 执行这件任务是否在浪费注意力？（几乎所有任务都会使注意力为之转移，你必须花些工夫才能再回到全神贯注的状态。）
- 如果手工操作失误会造成什么危害？

最后一个问题之所以重要，因为它涉及风险的考量。在我以前工作过的一个项目里，人们由于历史原因把代码和测试输出在同一个目录里。要运行测试，我们需要创建三个不同的测试套件，分别对应一种类型的测试（单元测试、功能测试和集成测试）。项目经理建议我们手工创建这些测试套件，但我们还是决定借助反射机制来自动创建它们。手工更新测试套件很容易出错，开发者很可能会写了测试却忘记把它加入到测试套件，从而导致新增的测试不被运行。我们认为，不将这个环节自动化可能造成很大的危害。

当你打算把某个任务自动化时，项目经理可能会担心你的工作失控。我们都有过这样的经验：原本以为只要两个小时就能搞定的事，最终用了4天才做完。要控制这种风险，最好的办法就是“时间盒”（timebox）：首先定好一段时间来探索和了解情况，时间一到

就客观地评估是否值得去做这件事。使用时间盒是为了掌握更多信息，以便作出切合实际的决策。时间盒到期以后，如果掌握的信息不够，你也可以再增加一个时间盒，以便找出更多信息。我知道，巧妙的自动化脚本比那些无聊的项目工作更让你感兴趣，但还是现实点吧，你的老板一定比较喜欢脚踏实地的工作量估算。

提示：

研究性的工作应该放在时间盒里做。

别给牦牛剪毛

最后，别让自动化的努力变成剪牦牛毛（yak shaving）——这是一句在计算机科学界源远流长的行话，它代表了诸如此类的情况：

1. 你打算根据 Subversion 日志自动生成一些文档。
2. 你尝试给 Subversion 加上一个钩子，然后发现当前使用的 Subversion 版本与你的 Web 服务器不兼容。
3. 你开始更新 Web 服务器的版本，随后又发现这个新版本在操作系统当前的这个补丁级别上不被支持，于是你开始更新操作系统。
4. 操作系统的更新包存在一个已知的问题，与用于备份的磁盘阵列不兼容。
5. 你下载了尚未正式发布的针对磁盘阵列的操作系统补丁，它应该能用。它确实能用，但又导致显卡驱动出了问题。

终于在某个时候，你停下来回想自己一开始到底是想干什么。然后你发现自己正在给牦牛剪毛，这时你就应该停下来想想：这一大堆牦牛毛跟“从 Subversion 日志生成文档”到底有什么关系呢？

剪牦牛毛是件危险的事，因为它会吃掉你大把的时间。这也能解释为什么任务工作量估算常常出现偏差：剪光一头牦牛的毛需要多少时间？始终牢记你到底要做什么，如果情况开始失控就及时抽身而出。

小结

本章用大量示例展示了如何将工作中的任务自动化。但这些例子本身并非重点所在，它们只是用于展示我和其他人业已发现的自动化手段而已。计算机之所以存在就是为了执

行简单重复的任务的，你应该让它们去工作！找出那些每天、每周都在做的重复工作，问问你自己：我能把这件事自动化吗？把重复的工作自动化，就能给你更多的时间来做有用的事，而不是一遍又一遍地解决没有价值的问题。手工做那些简单重复的事会浪费注意力，将这些烦人的琐事自动化，你就可以把宝贵的精力用来做其他更有价值的事。

第 5 章

规范性法则

现在距你给大老板做演示还有两个小时，而一个关键的特性在你的机器上不工作了。这不应该，它上周在Bob的机器上还是工作的。于是你跑到Bob的机器上，它果然可以很漂亮地运行，但是在你机器上工作得很好的其他几个特性却在Bob的机器上失效了。现在是你心慌意乱的时刻了。

不久之后，整个开发团队都围站在Bob的机器旁，试图弄清为什么他构建出来的版本跟别人的都不一样。这一切发生得真不是时候——恰好在一个重要的项目里程碑之前，但它已经不可避免地发生了。结果原来是Bob为他的IDE安装了某个插件的新版本，而这个新版本改变了应用程序的运行方式。当然，在Bob机器上安装同样版本的插件会破坏其他的东西。你、Bob，还有你所有的同事们正遭受折磨，因为你们使用了某个重要东西的多个版本，事实证明这样的东西一定会失去同步。

规范表示（canonical representation）指的是不损失信息的前提下最简单的表示形式。规范性（canonicality）指的是消除重复。在启蒙书籍《The Pragmatic Programmer》(Addison-Wesley) 中，Andrew Hunt 和 David Thomas 制定了这条法则："不要重复你自己"（Don't Repeat yourself，DRY）。这句话尽管只有三个单词，却对软件开发有着深远的影响。Glenn Vanderburg 把重复称作 "软件开发中最大的阻力（没有之一）"——我猜你已经同意了这种说法。那在软件开发中如何实现规范化呢？在很多情形下甚至连注意到这些问题都很困难，尤其在人们还没有重现 DRY 的情况下。

本章借助示例来说明怎么获得规范性。我们将会介绍非 DRY 情形的三种常见来源：数据库的对象－关系映射、文档和沟通。这些故事都来自真实的项目，并且在每个故事里，开发者们最终都找到了遵守 DRY 原则的方法。

DRY 版本控制

作为一种显而易见的规范性的应用，版本控制在绝大多数开发企业中已经成为常态。版本控制是一种规范化实践，因为"真正的"文件只存在于版本控制中。版本控制在处理文件版本方面具有明显的好处，同时它也是一种强大的备份机制，把你的源代码保存在一个安全的场所，远离开发者机器上单一的文件实例。

我倾向于优先选择不会锁住文件的版本控制系统。如果有不止一个开发者作了改动它会合并文件内容（这叫做乐观修订）。这是一个很好的例子：工具应该鼓励好的行为。惩罚坏的行为。尽早、尽可能频繁地提交文件到版本控制中鼓励你进行小步改动。如果进行了长时间的改动你就会面临合并冲突的问题，认识到这一点将鼓励你越发经常地提交。这种工具产生了一种有用的张力，以微妙但有益的方式改变了你的工作方式。好的工具

应该鼓励好的行为。因此，我喜欢开源的 Subversion 版本控制系统：它非常轻量，它是免费的，并且它只做你期望它做的，从不节外生枝。

尽管版本控制系统的使用相当普遍，但通常并没有用到它全部的潜能。版本控制能够让你的项目工件尽可能地DRY。构建你的项目所需要的任何东西都应该进入版本控制，包括二进制文件（库、框架、JAR 文件、构建脚本等）。唯一不应该在版本控制系统中的是因为路径、IP地址等原因而特定于开发者机器的配置文件。即使在这种情形下，也只有特定于开发者机器的信息应该保存在本地文件中。构建工具（像 Ant 和 Nant）允许你把特定的信息抽取到构建脚本之外，从而隔离变化。

为什么要把二进制文件也提交到版本控制系统中？今天的项目依赖于一长串的外部工具和库。假设你正在使用某个流行的日志框架（比如Log4J或者Log4Net）。如果不把日志库的构建作为项目构建过程的一部分，那么就应该把它提交到版本控制中。这样即使框架或库都已经不存在了（或者，更可能发生的，框架或库的新版本引入了一个不兼容的改动），你依然能够构建你的软件。要始终把构建软件所需要的一切事物都保持在版本控制中。（操作系统除外 —— 但其实这也是有可能的，如果使用虚拟化技术的话；参考本章后面的“利用虚拟平台”一节）。你可以作些优化，把二进制文件同时保存在版本控制系统中和共享网络驱动器上。那样的话，你就不需要每隔一小会就去版本控制中探查一下它们，但即使你一年之后需要重新构建某个东西。它们依然被保存在那儿。你永远都不知道你是否需要重新构建某个东西。在它能够工作之前你会一直构建它，但随后你就不管它了。如果有一天你发现需要重新构建两年前的某个东西，但却没有它全部的组成部分。就该手足无措了。

> **提示：**
>
> 对于任何你不自己去构建的东西，只在版本控制中保存一份副本。

当然，二进制文件会让版本控制占用更多空间，可能会导致存储（额外的空间）和带宽（签出项目的时间）方面的问题。有两个可接受的替代方案。一些版本控制工具（像 Subversion）有一个外部（externals）选项，允许你从一个项目中引用另外一个项目。你可以把你所有的共享库都保存在一个外部项目中，其他的项目都可以引用这个外部项目。这些二进制文件还是存在于版本控制中，但它们只占一块空间。这解决了存储问题但没有解决带宽问题。

另外一种方案是把二进制文件放在映射的网络驱动器上，可以被每一台开发机器所引用。

这个方案听起来有些吓人，因为你有些构建项目必需的文件不再存在于版本控制中了，但有时这是唯一合适的替代方案。

遗憾的是，大多数项目却偏偏选择了一种不可接受的替代方案：每个程序员在自己的机器上保存库文件，有时甚至是在不同的目录中。任何一个经历过这种项目的人都知道维护这样冗余混乱的基础设施是怎样的恶梦。

你知道你碰到配置问题了，当……

在一家我曾经工作过的咨询公司中，有这样一个客户，我们几年之前曾经在他们的应用程序上面做了一些工作。我们很久没有碰它了，是他们内部的一个开发者一直在维护和增强。他后来离开去做了冲浪运动员，或其他能够让他“找回他自己”的事业。而我的客户却没办法让这个应用程序在这个开发者的机器之外的任何一台机器上构建出来。他们确实努力了几个星期来构建这个项目，然而地球上唯一能够构建它的机器就是那个开发者的笔记本电脑。最终，他们把那台笔记本交给了我们，以便我们能够发现他到底搞了什么魔法。结果是他利用了一个很少人知道的 Java “特性”，就是运行时环境里的“ext”目录，因为他太懒了，没有把它加到 classpath 里面（或者不知道怎么加）。当你把笔记本交给咨询公司来弄清楚如何构建你自己的软件时，你就应该知道：你碰到配置问题了。

使用标准的构建服务器

每个开发组织都需要的另外一种实践是持续集成。持续集成是一个过程，在这个过程中你定期编译整个项目，运行测试，生成文档，以及执行其他一切构建软件所需要的活动（越频繁越好，通常你应该在每次提交代码到版本控制系统后都进行集成）。持续集成有专门的软件来支持。理想情况下，持续集成服务器运行在一台单独的机器上，监控着版本控制系统中的提交。每次你提交代码后，持续集成服务器就出场了，运行你指定的构建命令（诸如 Ant、Nant、Rake 或 Make 等构建脚本），内容通常包括执行一次完全编译，设置测试数据库，运行整个单元测试套件，代码静态分析，部署应用程序并执行一次“冒烟测试”。持续集成服务器把构建的责任从个人的机器上剥离出来，并创建了一个标准的构建场所。

这台标准的构建服务器不应该包含用于创建工程的开发工具，只应该包含构建应用所需的库和框架。这就阻止了难以察觉的工具依赖潜入你的构建过程。有 Bob 和他倒霉的同

事为鉴，你希望确保每个人都构建出同样的东西。拥有一台标准的构建服务器使它成为这个项目唯一“官方”的构建。开发工具的变化不会影响它。

用持续集成服务器作为单独的构建机器，这对每个开发者都有益。它防止了你无意中让工具依赖进入项目。如果只用一条命令就能在一台独立的机器上构建应用程序，那么显然所有配置都是正确的。

有很多持续集成服务器可用，既有商业的也有开源的。CruiseControl（注1）是ThoughtWorks创立的开源项目，它有Java、.NET和Ruby版本。其他的持续集成服务器包括Bamboo（注2）、Hudson（注3）、TeamCity（注4）、以及LuntBuild（注5）等。

间接机制

平台提供了重量级软件的基本结构。开发工具也通过提供稳固的基础创建了它们自己的平台，你可以基于此来构建软件。但平台的一部分是基础设施，而大量的开发工具创建了你无法控制的基础设施。间接机制允许你夺回控制权并变得更高效。

驯服 Eclipse 插件

> **提示：**
>
> 使用间接机制创建友善的工作空间（workspace）。

Eclipse最好的优点之一是丰富的插件生态系统。最坏的缺点之一也恰恰是这个丰富的插件生态系统！团队成员会下载不同的插件版本。通常这不是一个问题，但是偶尔插件版本之间存在不兼容，突然之间你便有了Bob那样不可复制的构建。这代表了一个“规范性”方面的问题。

解决方案是确保项目里的每个人使用完全相同的一组插件（精确到次版本号）。开发团队越大，这就越难管理。Eclipse的创建者早就预见到了这个问题，并允许你配置多个插

注1：在 *http://cruisecontrol.sourceforge.net/* 下载。

注2：在 *http://www.atlassian.com/software/bamboo/* 下载。

注3：在 *https://hudson.dev.java.net/* 下载。

注4：在 *http://www.jetbrains.com/teamcity/* 下载。

注5：在 *http://luntbuild.javaforge.com/* 下载。

件和特性的存放位置。令人费解的是，这个选项在“帮助”菜单里，准确说是在“软件更新→管理配置（Software Updates → Manage Configurations）”中。创建一个新的配置需要以下步骤：

1. 新建一个子目录叫 eclipse。这个目录不要放在 Eclipse 默认的目录结构里。
2. 在这个新目录下创建一个空的占位文件.eclipseextension。作为 Windows 和 Eclipse 不对眼的一个例子，你无法在 Windows 资源管理器中创建这个文件，因为它不允许创建“.”开头的文件。因此，你必须打开命令行窗口（Windows shell 或者 bash shell 都可以）来创建这个文件。在有这个命令的操作系统上做这件事最简单的方式就是 `touch .eclipseextension`。
3. 在这个新目录下创建两个（空）目录：*features* 和 *plugins*。

你必须在 Eclipse 允许你创建指向新目录的新配置前执行这些步骤。我不确定为什么 Eclipse 为什么不干脆直接帮你做了这些事，但它确实不会。无论如何，现在你可以使用“Product Configuration（产品配置）”对话框了（可以从“Manage Configuration（管理配置）”菜单项中启动它）。图 5-1 显示了产品配置对话框，里面定义了两个额外的配置。从这里你可以定义整个插件和特性的工作集，包括 JDK 和所有的 Eclipse 特性（第二个配置，目录是 *c:\work\eclipse*，就包含了整个 SDK）。你也可以只是指向插件的一个子集（就像第三个配置显示的那样，目录是 *c:\work\IVE\eclipse*）。

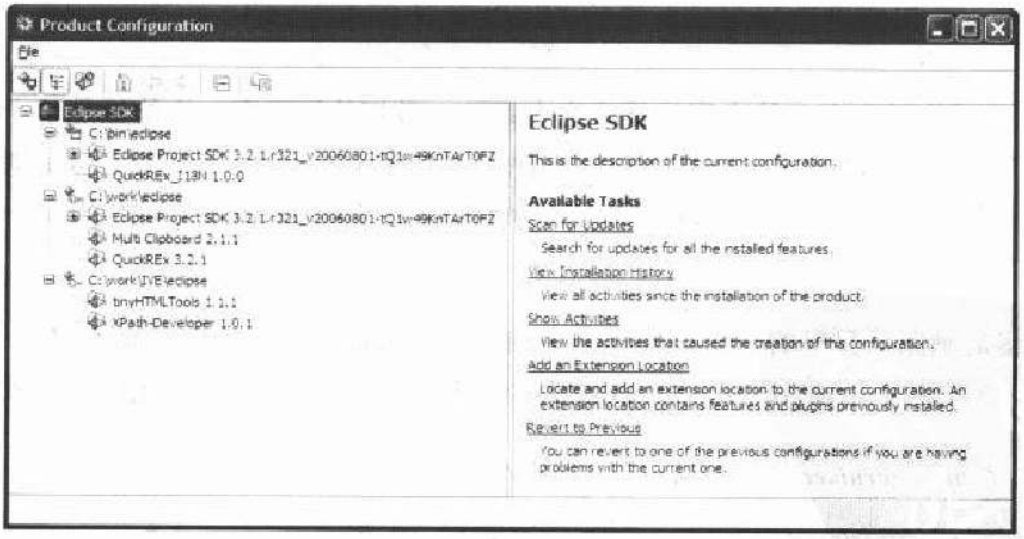


图 5-1：Eclipse 的产品配置（Product Configuration）对话框

在 Eclipse 中，你有两种不同的方式来安装插件和特性。可以自己下载文件并解压到特定的目标目录中，此时你可以在外部配置的目录里解压你的插件；也还可以使用“Find

and Install（查找并安装...）”菜单项指向一个预先定义好的URL并直接下载插件和特性，Eclipse在下载过程中提供了一个按钮，来询问你打算把特性或插件保存在哪个配置里（如图5-2）。

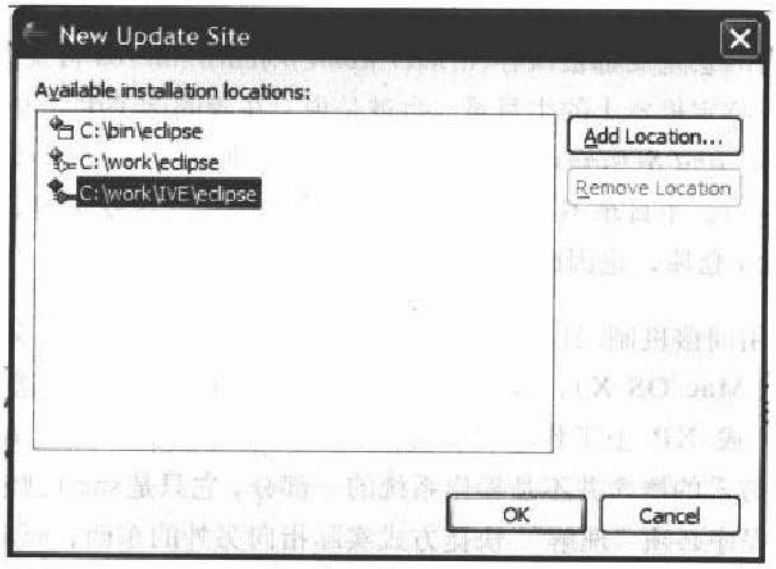


图5-2：额外配置的更新站点

剩下的事就很简单了：在新目录中创建产品配置并提交到版本控制中；设置这个项目中你需要的任何插件并提交到版本控制中。现在，在其他开发者的机器上签出新配置，并在全新安装的Eclipse中使用“Product Configuration（产品配置）”对话框指向它。当某个开发者的工作站发生变化时，这些变化会在下次版本控制系统更新并重启Eclipse后出现在其他工作站上。

当你创建一个新的配置时，它独立存在于“主”配置之外。你可以有选择地启用或禁用配置。这非常方便，尤其是你工作在不同项目上，而每个项目都有自己的插件集。如果配置发生变化，你还是必须重启Eclipse。但这比安装和卸载插件简单多了。

在我发明了这个办法之后，出现了一个叫做Pulse的整体解决方案来管理Eclipse的插件（注6）。当然，我介绍的技巧依然有效，但是有人认识到这确实是个问题并创建了一个工具来解决它！

同步JEdit宏定义

我喜欢用JEdit作为通用的文本编辑器。它有一个很棒的特性：它可以记录和保存宏。

注6：　在*http://www.poweredbypulse.com/*下载。

我需要在几台不同的机器上工作(家里的几台Windows和移动的Macintosh笔记本)。为了让文档保持同步，我在因特网上的另外一台机器上创建了一个Subversion仓库。这个仓库包含了我整个 Documents 目录（在Windows上它被保存在 My Documents 中，在 Mac 上则是 ~/Documents)。

在JEdit里，所有的宏定义都被保存在*[user home]/.jedit/macros*目录下，其中“[user home]”是用户在特定机器上的主目录。也就是说，在我的例子中，主目录在Windows上是*c:\Documents and Settings\nford*，而在Mac上则是*/Users/neal/*（或者，更方便的UNIX记法，~/)。主目录不在那个Documents目录中。这意味着JEdit的宏定义没有进入Subversion仓库，也因此无法在不同的机器上同步。

解决方案就是利用间接机制：让JEdit在你希望的地方寻找它的宏定义。在 *-nix 操作系统上（Linux、Mac OS X)，你可以用符号链接来做到这一点。遗憾的是，如果在Windows 2000 或 XP 上工作的话，你无法创建符号链接，快捷方式也无法胜任：Windows上快捷方式的概念并不是操作系统的一部分，它只是shell创建的一种表示法。其结果是，应用程序必须“理解”快捷方式实际指向另外的东西，而JEdit并不了解这一点。如果你使用 Windows Vista 之前的Windows版本，你需要某种类似符号链接的机制来支持间接操作（Vista则带有一个*mklink*命令可以创建真正的符号链接)。幸运的是，有一个叫做的 Junction 的免费工具可以帮你做到这一点。

针对 Windows 2000 和 XP 用户的 junction

UNIX、Linux和Mac OS X的开发者有操作系统内建的符号链接。Windows用户则需要快捷方式之外的某种东西。Junction在文件系统的内容表中创建了重复的表项，允许你创建指向其他目录的指针。它工作在操作系统级别，因此所有的应用程序（包括Windows自身）都能够理解它创建的指针。

junction是一个命令行工具，允许你创建和删除硬链接。例如，如果你想在当前目录下创建一个叫做“myproject”的链接指向另外一个（深层嵌套）的目录，你可以键入以下命令：

```
junction myproject \My Documents\Projects\Sisyphus\NextGen
```

假文件夹 *myproject* 在操作系统级别指向你的*\My Documents\Projects\Sisyphus\NextGen*文件夹。

现在，我们在所有的操作系统上都有了真正的间接机制，于是我可以在 Documents 下创建一个目录来保存我所有的JEdit宏定义。在Windows上，我使用junction链接在

c:\Documents and Settings\nford\.jedit 下创建叫做“macros”的指针。类似地，在OS X上，我在 *~/.jedit* 下创建叫做 “macros”的符号链接。junction链接和符号链接都指向 *Documents* 下的目录（因此可以放进Subversion仓库中）。图5-3展示了这个方案。

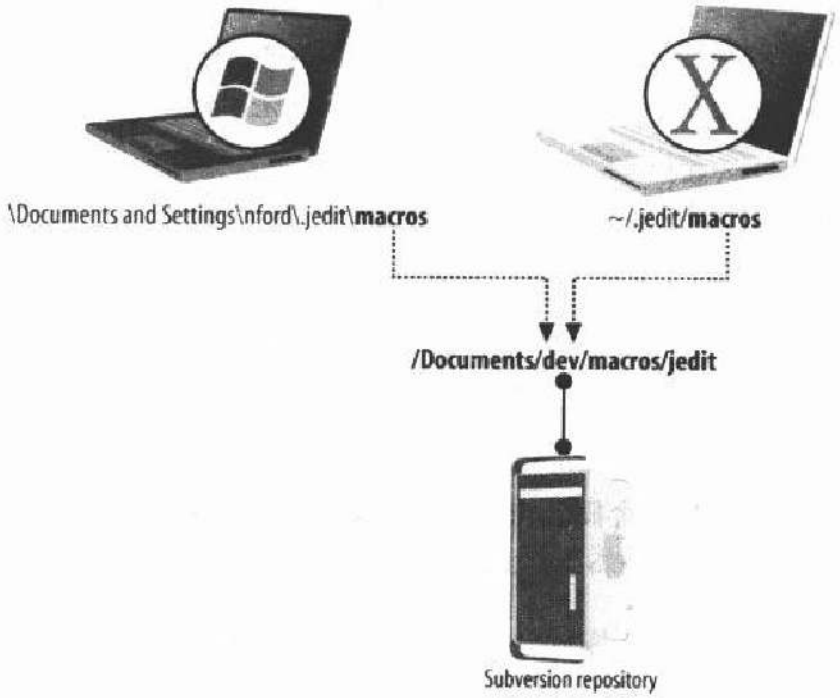


图5-3：不同操作系统上的JEdit宏的位置

现在，我能够很愉快地在任何一台机器上定义宏并为我所有的工作进行一次Subversion提交。当我在其他机器上时，所有的宏定义都下载到了Documents文件夹中。因为我“愚弄”了JEdit，让它去Documents（而不是默认的位置）浏览宏定义，所有我曾经记录的宏都立即可用了。

> **提示：**
>
> 使用间接机制来保持文件同步。

间接可以多级嵌套。曾经有一次我为多个项目工作，每一个都有自定义的宏。我想用间接原则来共享这些宏定义，但其他团队只对适合他们项目的宏定义感兴趣。如果通过间接把JEdit宏定义文件夹里所有的宏都塞给他们，这就违反了专注生产率原则——在那个原则里，我们一直关注如何消除干扰。

JEdit支持在它的宏定义文件夹中放置子目录，因此解决办法很简单：每个项目依然把包含各自宏定义的文件夹放置到各自的版本控制仓库中，并创建符号链接或者junction链

接指向各自的宏定义文件夹。而在我的机器上，JEdit 寻找宏定义的文件夹依然是个符号链接（或junction 链接），但它包含着指向版本控制中其他外部目录的链接。

可以让一个间接链接指向另外一个间接链接，这样把间接分成任意多层，从而创建一个间接链。最主要的限制因素是你使用的工具：如果需要共享的资源是文件（而非目录）并且你使用Windows，那么你将无法使用这个技巧。不过，有这种情况的工具并不多。如果你发现自己不得不使用一个束手束脚的工具，也许是时候换一个了。

TextMate Bundles

TextMate（注 7）是 Mac OS X 上程序员使用的专业编辑器。它的一个强大功能就是代码片段（snippet）的概念（其他工具中叫做“模板”）。TextMate已经移植到了Windows上，称作“E 文本编辑器（E Text Editor（注 8））”。

TextMate 中的代码片段保存在 *bundle* 中。Bundle 是 Mac OS X 上以打包形式存在的一组文件。Mac OS X 使用这个概念为包含大量文件的应用提供安装支持。TextMate 之类的应用程序也可用把 Mac OS X 包的概念用于自己的 bundle。

TextMate有一个很棒的功能：你可以在文件系统上查找Bundle，把它们拖放到另外一个地方（一般是一个共享的网络驱动器），从而共享这些bundle。Bundle存放在 *~/Library/Application Support/TextMate/Bundles* 目录中。只要把一个bundle解压到另一台机器的这个目录里，就可以双击安装它。换句话说，TextMate 考虑到了如何共享它的bundle，并给它们提供了一种可以自安装到其他机器上的格式。向TextMate的创建者脱帽致敬。他们坚守着高效程序员的理想！

不过通过复制粘贴来共享并不是最理想的。当你添加新的片段到 bundle 中时怎么办？Bundle的其他副本很显然无法享受新的代码片段带来的好处。复制粘贴式的复用是邪恶的，即使只是复制配置。

> **提示：**
>
> 通过复制粘贴来复用是邪恶的，不论你复制粘贴的是什么。

尽管在 OS X 的 Finder 程序中 bundle 看起来像是一个个单独的文件，但实际上它们是文件夹。这就意味着你可以通过链接来表示它们。就像前面的JEdit例子一样，你可以在

注 7：在 *http://macromates.com/* 下载。

注 8：在 *http://www.e-texteditor.com/* 下载。

*~/Library/Application Support/TextMate/Bundles*文件夹中创建链接指向版本控制中真正的文件夹。这允许开发团队中的每个人都可以访问相同的bundle集合，因此当有人创建了一段极其有用的代码片段时，整个团队就会在下次版本控制更新周期后马上受益。

规范的配置

把所有的机器都安装和设置为一模一样总是项目中令人头疼的一件事。一些项目采取了在操作系统级别创建镜像的方式（对某些开发环境来说，这是唯一的方式；参见下一节“利用虚拟平台”）。而借助间接机制，你能够显著地简化这个问题。

例如，Emacs把它所有的配置信息保存在一个叫做 *.emacs* 的文件里。它位于用户的主目录下。（它把另外一些信息（比如历史记录等）保存在一个叫做 *.emacs.d* 的目录中）。如果你想跨机器来共享配置怎么办？你可以使用间接和符号链接（在 *-nix 系统上）把真正的 *.emacs* 文件保存在版本控制中而让Emacs指向链接。遗憾的是，在 Windows 2000 和 XP 上 junction 并不能做到这一点，不过你可以在 Windows Vista 上用符号链接完成。

项目中另一个常见的麻烦与开发者用来加速编码的代码片段模板有关。我在前面介绍了在 TextMate 中如何解决这个问题，但是对不使用 TextMate 的人帮助不大。你可以在各种流行的 IDE 中创建代码片段。但遗憾的是，没有标准的方法来做这件事。因此我只会讨论两种流行的 Java IDE: IntelliJ 和 Eclipse。

对IntelliJ来说，共享代码片段（在 IntelliJ 中叫做 live templates）相当容易，因为它们保存在一个目录中（例如，在 Mac OS X 上它们在 */Users/nealford/Library/Preferences/IntelliJIDEA70/templates*中），并且每段片段都对应一个自己的文件。因此，要共享IntelliJ的代码片段，只需要把片段目录的一个标准版本移到版本控制中，然后在原来的位置创建符号链接（或者 junction 链接）指向它在版本控制中的新家。

遗憾的是，在Eclipse中则要难得多，因为Eclipse把这些片段埋藏在了 Java 属性文件中。自定义的代码片段被编码为XML，作为某个属性的值。不，我没有开玩笑，就好像是为了尽可能让它们难以通过编程来获得。文件中的片段看起来就像这样（我加上了换行）：

```
#Tue Feb 12 09:45:01 EST 2008
org.eclipse.jdt.ui.overrideannotation=true
spelling_locale_initialized=true
org.eclipse.jdt.ui.javadoclocations.migrated=true
proposalOrderMigrated=true
org.eclipse.jdt.ui.formatterprofiles.version=11
useQuickDiffPrefPage=true
```

```
org.eclipse.jdt.ui.text.custom_templates=<?xml version="1.0"
  encoding="UTF-8"?><templates><template autoinsert="true" context="java"
  deleted="false" description="" enabled="true" name="my_test"> @Test
public void ${var}() { }</template></templates>
org.eclipse.jdt.ui.text.code_templates_migrated=true
```

Eclipse 确实提供了一种方法：通过 Preferences（首选项）对话框来导入和导出代码片段。导入和导出本质上是复制粘贴，但这确实是共享代码片段最简单的方式。另一个抱怨是插件作者可能把他们的片段保存在不同的目录中：在 Eclipse 中没有一个标准的位置来存放代码片段。Eclipse 把代码片段保存在文本文件中，因此你可以编写一个脚本来周期性的保存和重新生成属性文件，但这需要大量的工作。

利用虚拟平台

提示：

利用虚拟平台使项目依赖标准化。

几年前，当我试图重建在另外一个项目上用过的Visual Studio环境时，我运用了规范性的“间接”手法来简化这一过程。Visual Studio 有一个丰富的第三方组件的生态系统，但你必须清楚：大量使用第三方组件意味着每个应用程序开发环境有着微妙的差异。

比如说，客户A使用了某个窗口组件（widget），而你却必须确保不会为客户B使用它，因为客户B并没有购买它的许可。一旦你在开发者的机器上安装了组件，它就变成了操作系统的一部分。有些客户的安装设置很复杂，仅仅把环境搭好就能耗费一个星期。问题在于隔离：你无法在比操作系统低的层面上封装开发环境（或者要开发的应用程序）。

最后我们使用了虚拟操作系统来开发应用。当时做这件事的首选工具是 VMWare，而它确实相当不错. 我们可以拿一个通用的Windows ghost映象在VMWare虚拟机中安装所有必需的开发工具，并在上面开发。虚拟机的速度还可以，而且我们可以为每个不同的客户创建一个开发净室。当那一阶段项目结束时，我们把 VMWare 的映像文件保存到了服务器上。

两年以后，当那个客户回过头来找我们做些增强的时候，我们启动了跟我们离开那天一样的应用开发环境。这节省了好几天的停工期，并且让为多个用户开发变得轻而易举。当我为客户B的应用程序工作的时候，客户A需要一些小的修改。这都没问题，在虚拟机映象间来回切换就可以了。这种方法同时带来了另外一种明显的好处：在一个干净的操作系统和开发工具环境中开发消除了那些操作系统、工具、办公套件等之间隐藏的依赖。

DRY 阻抗失配

你有过打电话交谈时伴有恼人的回声的体验吗？ 这就是阻抗失配（impedance mismatch），是由信号无法精确同步引起的。阻抗失配是由电子工程领域渗透到软件世界里的一个词汇，因为它描述了我们的一些问题。

在软件开发中，阻抗失配是导致违反 DRY 原则的常见原因之一。阻抗失配发生在两种抽象风格的边界处：从基于集合到基于对象，或从过程式到面向对象。因为你试图融合两种抽象风格，在边界处就出现了重复。

数据映射

> **提示：**
>
> 不要让对象 - 关系映射工具（O/R 映射器）违反规范原则。

在需要处理数据的项目中总有一件事让我们头疼：关系型数据库和面向对象语言之间的阻抗失配。为了解决这个问题，我们逐渐走向了 O/R 映射器，例如 Hibernate、NHibernate、iBatis 以及其他一些工具。但使用 O/R 映射器在项目中引入了重复，我们在三个地方拥有本质上相同的信息：数据库结构定义文件（schema）、XML 格式的映射文件以及类文件。这表示我们违反了 DRY 原则两次。

这个问题的解决方案是创建唯一表示，并以它为基准生成另外两种。第一步，我们需要决定谁是这些信息的“正式”持有者。举例来说，如果数据库是标准的信息来源，那就基于它生成 XML 格式的映射文件和对应的类文件。

在这个例子里，我使用 Groovy（一种 Java 的脚本方言）来解决阻抗失配。在例子中的这个项目里，开发人员对数据库结构没有控制权。因此，我认为应该以数据库作为数据的标准表示。我使用开源的 iBatis（注 9）SQL 映射工具（它不产生 SQL，只是把类映射到 SQL 结果）。

首先，我们需要从数据库中得到其结构（schema）信息：

```
class GenerateEventSqlMap {
    static final SQL =
        ["sqlUrl":"jdbc:derby:/Users/jNf/work/derby_data/schedule",
            "driverClass":"org.apache.derby.jdbc.EmbeddedDriver"]
    def _file_name
```

注 9：*在 http://ibatis.apache.org/ 下载。*

```
def types = [:]
def GenerateEventSqlMap(file_name) {
    _file_name = file_name
}

def columnNames() {❶
    Class.forName(SQL["driverClass"])
    def rs = DriverManager.getConnection(SQL["sqlUrl"]).createStatement().
        executeQuery("select * from event where 1=0")

    def rsmd = rs.getMetaData()
    def columns = []
    for (index in 1..rsmd.getColumnCount()) {
        columns << rsmd.getColumnName(index)
        types.put(camelize(rsmd.getColumnName(index)),
    rsmd.getColumnTypeName(index))
    }
    return columns
}

def camelized_columns() {❷
    def cc = []
    columnNames().each { c ->
        cc << camelize(c)
    }
    cc
}

def camelize(name) {
    def newName = name.toLowerCase().split("_").collect() {
        it.substring(0,1).toUpperCase() + it.substring(1,it.length())
    }.join()
    newName.substring(0,1).toLowerCase() +
  newName.substring(1,newName.length())
}

def columnMap() {
    def columnMap = [:]
    for (colName in columnNames())
        columnMap.put(camelize(colName), colName)
    return columnMap
}

def create_mapping_file() {❸
    def writer = new StringWriter()
    def xml = new MarkupBuilder(writer)
    xml.sqlMap(namespace:'event') {
        typeAlias(alias:'Event',
    type:'com.nealford.conf.canonicality.Event')
        resultMap(id:'eventResult',class:'Event') {
            columnMap().each() {key, value ->❹
      result(property:"${key}", column:"${value}")
        })
        select(id:'getEvents',resultMap:'eventResult',
                'select * from event where id = ?')
```

```
                select(id:"getEvent",
            resultClass:"com.nealford.conf.canonicality.Event",
                        "select * from event where id = #value#")
            }
            new File(_file_name).withWriter { w ->
                w.writeLine("${writer.toString()}")
            }
        }
    }
```

❶ columnNames 方法使用底层的 JDBC 从数据库中获得列名。

❷ camelized_columns 根据数据库列名生成典型的 Java 方法名。

❸ create_mapping_file 使用一个 Groovy 构建器来方便地输出 XML 文档。

❹ 使用 builder 的一个好处是：它创建 XML 文档的语法比 DOM 等技术更简洁。你也可以利用循环（这里是通过 each 方法）来从代码中产生 XML。

在 BuildEventSqlMap 类外部，我构造了它的一个对象并要求它产生 XML 映射文件：

```
def generator = new GenerateEventSqlMap("/Users/jNf/temp/EventSqlMap.xml")
generator.create_mapping_file()
```

这个调用的结果就是能够作为 iBatis 输入的映射文件：

```
<sqlMap namespace='event'>
    <typeAlias type='com.nealford.conf.canonicality.Event' alias='Event' />
    <resultMap id='eventResult' class='Event'>
      <result property='description' column='DESCRIPTION' />
      <result property='eventKey' column='EVENT_KEY' />
      <result property='start' column='START' />
      <result property='eventType' column='EVENT_TYPE' />
      <result property='duration' column='DURATION' />
    </resultMap>
    <select resultMap='eventResult' id='getEvents'>
      select * from event where id = ?
    </select>
    <select resultClass='com.nealford.conf.canonicality.Event' id='getEvent'>
      select * from event where id = #value#
    </select>
  </sqlMap>
```

生成 XML 和 SQL 间的映射消除了一处重复（“根据数据库结构生成映射文件”现在已经成为构建过程的一部分）。可以用相同的技术来产生类文件。事实上，我可以利用同样的基础设施，因为我已经从数据库中得到了列名。为了产生类文件，我构造了 ClassBuilder 类：

```
class ClassBuilder {
      def imports = []
      def fields = [:]
```

```
    def file_name
    def package_name

    def ClassBuilder(imports, fields, file_name, package_name) {
        this.imports = imports
        this.fields = fields
        this.file_name = file_name
        this.package_name = package_name
    }

    def write_imports(w) {
        imports.each { i ->
            w.writeLine("import ${i};")
        }
        w.writeLine("")
    }

    def write_classname(w) {
        def class_name_with_extension = file_name.substring(
        file_name.lastIndexOf("/") + 1, file_name.length());
        w.writeLine("public class " +
            class_name_with_extension.substring(0,
        class_name_with_extension.length() - 5) + " {")
    }

    def write_fields(w) {
        fields.each { name, type ->
            w.writeLine("\t${type} ${name};");
        }
        w.writeLine("")
    }

    def write_properties(w) {❶
        fields.each { name, type ->
            def cap_name = name.charAt(0).toString().toUpperCase() +
        name.substring(1)
            w.writeLine("\tpublic ${type} get${cap_name}() {")
            w.writeLine("\t\treturn ${name};\n\t}\n");
            w.writeLine("\tpublic void set${cap_name}(${type} ${name}) {")
            w.writeLine("\t\tthis.${name} = ${name};\n\t}\n")
        }
    }

    def generate_class_file() {❷
        new File(file_name).withWriter { w ->
            w.writeLine("package ${package_name};\n")
            write_imports(w)
            write_classname(w)
            write_fields(w)
            write_properties(w)
            w.writeLine("}")
        }
    }
}
```

❶ Groovy灵活的字符串语法（像Ruby一样，它允许你在字符串中替代成员变量）使产生标准的Java结构（例如get/set方法等）变得很容易。

❷ 这个方法使用前面的辅助方法来输出标准的Java文件。

在脚本中，在调用产生XML映射文件的方法之后，我又调用了ClassBuilder来构造对应同样信息的Java文件：

```
TYPE_MAPPING = ["INTEGER" : "int", "VARCHAR" : "String"]
def fields = [:]
generator.camelized_columns().each { name ->
    fields.put(name, TYPE_MAPPING[generator.types[name]]);
}
new ClassBuilder(["java.util.Date"], fields,
    "/Users/jNf/temp/Event.java", "com.nealford.conf.canonicality").
  generate_class_file()
```

调用的结果就是以下的Java文件：

```
package com.nealford.conf.canonicality;

import java.util.Date;

public class Event {

  String description;
  int eventKey;
  String start;
  int eventType;
  int duration;

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public int getEventKey() {
    return eventKey;
  }

  public void setEventKey(int eventKey) {
    this.eventKey = eventKey;
  }

  public String getStart() {
    return start;
  }

  public void setStart(String start) {
    this.start = start;
```

```
    }

    public int getEventType() {
      return eventType;
    }

    public void setEventType(int eventType) {
      this.eventType = eventType;
    }

    public int getDuration() {
      return duration;
    }

    public void setDuration(int duration) {
      this.duration = duration;
    }
}
```

当然，这只是一个没有行为的DAO（数据存取对象）：它仅仅是一组 get/set 方法的集合。如果你需要为这个实体增加行为，可以继承这个DAO。永远不要手动编辑一个自动生成的文件，因为在你下次执行构建的时候它会被重新生成。

> **提示：**
>
> 通过扩展。开放类（open classe），或者部分类（partial classe）来为生成的代码增加行为。

要为自动生成的代码增加行为，你可以继承（在类似Java的语言中），使用开放类（在类似 Ruby，Groovy，Python的语言中），或者使用部分类（在类似 C# 之类的语言中）。

现在，我已经解决了 O/R 映射代码中所有违背 DRY 原则的问题。这段 Groovy 脚本作为构建过程的一部分来运行，因此每次数据库结构改变后，它会自动生成对应的映射文件和Java文件。

数据迁移

另外一种导致项目中出现重复的情形同样来自于代码和SQL间的失配。大量的项目把源代码和SQL看作完全无关的产物，经常是由完全独立的开发组分别创建的。然而，为了使代码能够正常工作，它必须依靠于数据库结构和数据的某个特定版本。我将介绍两种解决这个问题的办法，其中一种依赖于特定的框架，另一种则与具体的框架或语言无关。

提示：

始终保持代码和数据库结构的同步。

Rake 迁移

作为一个 Web 开发框架，Ruby on Rails 拥有很多精妙的特性，迁移（migration）就是其中之一。一个迁移就是一个 Ruby 源文件，它能够处理你的数据库结构的版本，保持它与源代码的同步。通常，你会同时修改代码和数据库（包括结构和测试数据）。用迁移来管理数据库，你就可以把两者的修改一起提交到版本控制中，这样版本控制就能够保存“代码 + 数据”的系统完整快照。

Rails 迁移是由 Rails 自带的一个工厂生成的，它本质上是一个 Ruby 脚本文件，其中包含了两个方法：up（用来存放你对数据库的改动）和 down（用来存放与 up 相对的操作，也就是撤销 up 所做的一切）。迁移的命名都带有一个数字前缀（例如，001_create_user_table.rb）。Rails 带有 Rake 任务，在修改数据库时以从前向后的顺序运行迁移，而在撤销修改时则以相反的顺序进行。

这是一个 Rails 迁移的例子，它会创建由若干字段组成的一张表：

```
class CreateProducts < ActiveRecord::Migration
    def self.up
      create_table :products do |t|
        t.column :title,:string
        t.column :description,:text
        t.column :image_url,:string
      end
    end

    def self.down
       drop_table :products
    end
  end
```

在这次迁移中，我在 up 方法中创建了带有三个字段的 Product 表，并在 down 中删除了它。

迁移使你可以把表结构信息保存在源代码（而不是数据库）中，从而保持 DRY。这种设计还有一个额外的好处：Rails 因此可以支持多个目标部署环境。在 Rails 的配置文件 *database.yml* 中，你可以定义多个环境（比如说，“开发”、“测试”、“产品”环境）。迁移允许你把任何一个环境的数据库设置到某个特定状态，只要针对那个环境运行迁移就好了。

迁移唯一的缺点是它紧密绑定在 Rails 框架上。如果你用 Rails，那很好；否则它就帮不上什么忙了。

dbDeploy

不过就算不用 Rails，也不会无计可施。dbDeploy 是一个开源的框架，以一种平台无关的方式实现了迁移的部分功能。dbDeploy 以 Java 编写，支持大量数据库服务器（而且这个列表还在不断增长），其中包括所有主流数据库。

dbDeploy 会创建数据库的基线 SQL 快照（包括 DDL 和数据）。当开发者修改数据时，他们创建包含改动的脚本文件，并按顺序编号。dbDeploy 会负责生成真正的 SQL 脚本，并在数据库上运行。它把改动的历史轨迹保存在名为 dbdeploy 的数据库和你的数据库中的一张表（默认叫做 changelog）中。对于它所支持的各种数据库，dbDeploy 都提供脚本来创建 changelog 表。例如为 MS-SQL Server 建表的脚本就像这样：

```
USE dbdeploy
  GO

  CREATE TABLE changelog (
    change_number INTEGER NOT NULL,
    delta_set VARCHAR(10) NOT NULL,
    start_dt DATETIME NOT NULL,
    complete_dt DATETIME NULL,
    applied_by VARCHAR(100) NOT NULL,
    description VARCHAR(500) NOT NULL
  )
  GO

  ALTER TABLE changelog ADD CONSTRAINT Pkchangelog PRIMARY KEY
(change_number, delta_set)
  GO
```

> **提示：**
>
> 使用迁移为数据库结构的改动创建可重复的快照。

尽管不像迁移那样功能全面，dbDeploy 依然部分解决了把数据库结构和代码放在两个不同地方带来的问题。允许通过编程来管理数据库的改动，这让你能更好地保持这两部分的同步，并避免了代码和数据定义间固有的阻抗失配。

DRY 文档

提示：

过时的文档比没有文档更糟，因为它会主动误导你。

文档总是管理者和开发者之间的战场：管理者想要更多，而开发者想给更少。它也是对付信息重复的战场。开发者应该能够积极主动的修改代码，改进它的结构，使它能够逐渐演化改进。如果所有的代码都必须有文档记录，那么文档也必须同时演化。但绝大多数时候，代码和文档会失去同步，因为进度的压力，没有动力（承认吧，写代码比写文档有趣多了），以及其他原因。

提示：

对管理者来说，文档意味着缓解风险。

过时的文档带来了传播错误信息的风险（讽刺的是，它原来的目的是降低风险）。阻止文档过时的最好方法是尽可能自动生成它。这一节将提到几种可行的生成文档的办法。

SVN2Wiki

在一个项目中，我们遇到了发布信息的问题。开发者们分布在班加罗尔、纽约和芝加哥。我们共享一个版本控制仓库（在芝加哥），并用 wiki 跟踪重要决定（我们使用开源的 Instiki）。在每天结束时，每个开发者要负责更新 wiki，以说明他或她今天干了什么。这个规定的效果可想而知 —— 只要想想人们在下班时冲出办公室去挤地铁的情景就知道了。我们试图在开发者们耳边不断念叨这条规定，但这只是让他们恼羞成怒。

然后我们意识到：我们实际上违反了规范性原则，因为我们要求开发者记录他们已经记录过的东西 —— 就在提交到版本控制时的注释里。所有的开发者都善于编写描述性的注释。我们决定利用已经存在的资源，而最终，我们创造出了 SVN2Wiki，一个 Subversion 的小插件。当 Subversion 执行一些操作时，你可以编写程序让它来运行。SVN2Wiki 就坐在那里等着当有人提交代码时 Subversion 来调用它。然后它就获得了结对的开发者添加的注释，并把它们发布到 wiki 上去。

在能够自动发布注释之后，我们又意识到这个 wiki 支持 RSS。这意味所有开发者（当然，也包括管理者）能够订阅 wiki，从而获知从上次他们浏览之后代码库又发生了什么。完整的设置如图 5-4。

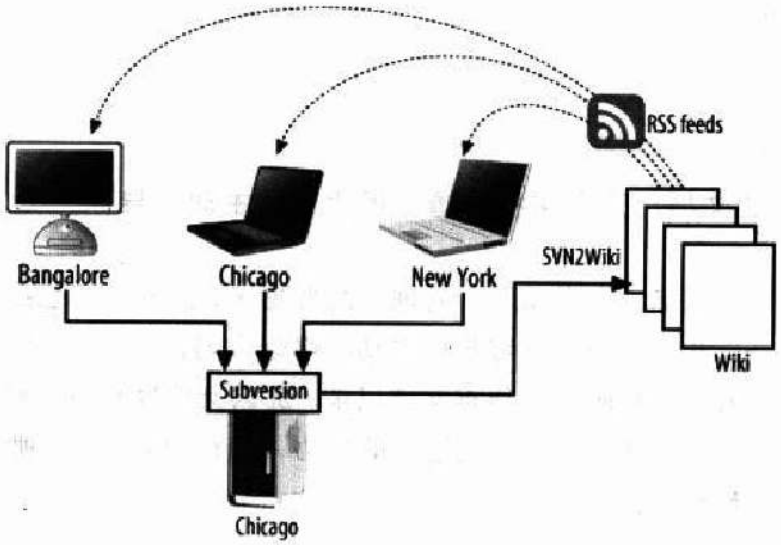


图 5-4：SVNWiki 为地理分隔的人们搭起沟通的桥梁

SVN2Wiki 的代码本身很简单．我们用的是 C#（它应该很容易移植到其他语言）。事实上，SVN2Wiki 中最复杂的代码是关于怎么把记录贴到特定日期的 wiki 页上。

```
namespace Tools.SVN2Wiki {
    public class SVN2Wiki {
      private const CONFIG =
          "c:/repository/hooks/svn2wiki-config.xml";
      private SubversionViewer subversionViewer;
      private string revision;
      private string repository;
      private SVN2WikiConfiguration config;
      private Wiki wiki;

      private static void Main(string[] args) {
        string repository = args[0];
        string revision = args[1];

        //get configuration
        SVN2WikiConfiguration config =
            new SVN2WikiConfiguration(CONFIG);
        config.loadConfiguration();

        Wiki wiki = new WikiUpdates(new HttpInvokerImpl(),
            config.WikiURL);
        SVN2Wiki svn2wiki = new SVN2Wiki(new SubversionViewerImpl(),
            revision, repository, config, wiki);
        svn2wiki.processUpdate();
      }

      public SVN2Wiki(SubversionViewer subversionViewer,
                      string revision,
                      string repository,
                      SVN2WikiConfiguration config,
```

```
                Wiki wiki)  {
    this.subversionViewer = subversionViewer;
    this.repository = repository;
    this.revision = revision;
    this.config = config;
    this.wiki = wiki;
  }
  public SVNCommit getCommitData() {
    string machine = subversionViewer.svnLook(
        "author -r " + revision + " " + repository);
    string date = subversionViewer.svnLook(
        "date -r " + revision + " " + repository);
    string comments = subversionViewer.svnLook(
        "log -r " + revision + " " + repository);
    string[] dateToParse = date.Split(' ');
    date = dateToParse[0] + " " + dateToParse[1] + " " + dateToParse[2];
    return new SVNCommit(machine, DateTime.Parse(date), comments);
  }
  public void processUpdate() {
    SVNCommit commit = getCommitData();
    //for each updater in the config file
    foreach (UpdaterConfiguration updater in config.Updaters) {
      if (needToPostSVNCommit(commit, updater)) {
        Console.WriteLine("Posting to " + updater.MenuPage);
        wiki.UpdatesListPage = updater.MenuPage;
        wiki.UpdatesPageNamePrefix = updater.UpdatePagePrefix;
        Console.WriteLine("posting commit:");
        Console.WriteLine(commit.Machine + " " +
            commit.CommittedOn);
        wiki.postUpdate(commit);
      }
    }
  }
  public bool needToPostSVNCommit(SVNCommit commit,
                                  UpdaterConfiguration updater) {
    string[] users = updater.ExcludeUsers.Split(',');
    if (arrayContainsString(users, commit.Machine))
      return false;
    if (updater.ExcludePaths.Length == 0)
      return true;
    else
    {
      string[] paths = updater.ExcludePaths.Split(',');
      string[] changedDirectories =
          getChangedDirectories(repository, revision);
      foreach (string changedDir in changedDirectories) {
        bool changedDirInExcludePaths = false;
        foreach (string path in paths) {
          Console.WriteLine("Path = " + path);
        if (changedDir.StartsWith(path))
            changedDirInExcludePaths = true;
        }
        if (!changedDirInExcludePaths)
          return true;
      }
```

```
        }
        return false;
      }

      private bool arrayContainsString(string[] array, string toFind) {
        foreach (string a in array)
          if (a == toFind) return true;
       return false;
      }

      private string[] getChangedDirectories(string repository,
                                             string revision) {
        return subversionViewer.svnLook(
            "dirs-changed -r " + revision + " " + repository).Split('\n');
      }
    }
  }
```

SVN2Wiki是一个“活文档”好例子。大部分项目的大部分文档都是很差劲的，因为它们与项目全然无关。由于我们在地理上太分散了，使用wiki被证明是对项目最好的事情，尤其是从文档化所有决定、甚至源代码的提交注释（使用SVN2Wiki）的角度来说。我们在这里发布项目所有重要的信息：会议日程和结论汇总，画在白板上并用数码相机拍下来的非正式的图表等。wiki都支持搜索（我们使用的wiki则支持使用正则表达式搜索，因此我们可以随时重新翻阅之前的决定）。项目结束时，我们把整个wiki导出为HTML。我们拥有的文档如此之棒，甚至都几乎可以根据它重建我们的项目，这正是文档的最初目的：创建一个关于“你做了什么”和“为什么做”的可信赖的信息来源。

提示：

始终保持“活”的文档。

类图

提示：

任何需要费劲创造的东西，都让它的创造者欲罢不能。

尽管敏捷开发尽量让自己看起来不那么正式，你有时还是会需要一些图来说明类或其他产物之间的关系。你可能被诱惑着去使用精心打造的工具来画这些图，但应该抵抗住诱惑。如果最终需要费劲来创造某件东西，就意味着需要费劲去改变它。对一幅图不理智的坚持程度与创造它所花费的努力成正比。如果它花了你15分钟，你会下意识地试图避免修改它，因为在脑海深处，你正考虑这修改需要多久来完成。

提示：

白板 + 数码相机强过任何 CASE 工具。

因此，不那么一本正经的产物是最好的，而我的最爱是在白板上慢慢地画。它几乎没有成本（因此你不介意改变它）。几个人合作来画也很容易（而大多数工具并不允许这么做）。当你画完时，用数码相机给它拍个照片，并作为文档的一部分。只有在用代码将其描述出来之前你才会需要它，那之后代码本身就会说话。

提示：

尽量生成所有技术文档。

为了让代码本身能说话，你应该找个工具从代码中生成框图。一个很好的例子是yDoc，这是一个商业的画图工具，能够从代码产生 UML 图，图中还带有超链接直接关联到源代码。图 5-5 显示了某个项目中的一个 UML 图。有些 IDE（像 Visual Studio）可以生成框图，但它们通常很难自动化。理想情况下，你应该在编译代码的同时生成框图（使用类似持续集成服务器的东西）。用这种方式，你就永远不必担心“我的框图是最新的吗”，因为它们永远处于最新的状态。

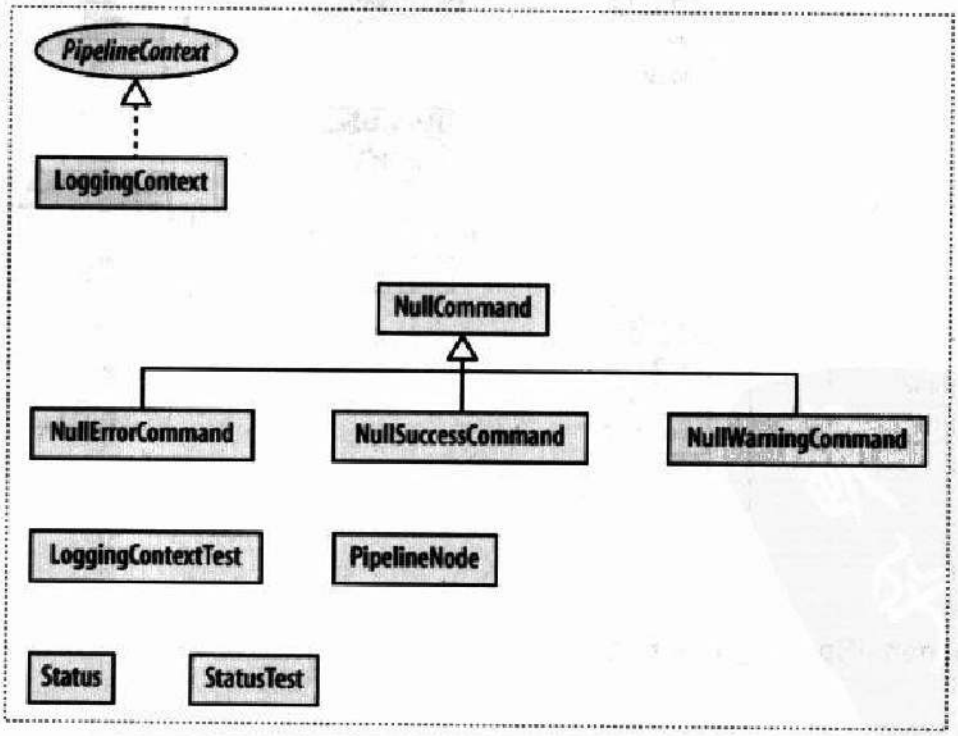


图 5-5：yDoc 创建的 UML 图

提示：

永远不要为同一份信息保存两份拷贝（比如代码和描述它的图）。

如果你打算先画图，那么用工具来生成代码。如果用白板创建了非正式的图，而稍后又需要更正式的版本，那么从代码中生成它们。否则，它们一定会失去同步。

数据库结构

就像类图一样，数据库结构是产生不必要重复的危险地带。SchemaSpy（注 10）是一个开源工具，就像 yDoc 对代码做的那样，它会产生数据库实体/关系图。它会连接到数据库，并产生表信息（包括元数据），以及类似图 5-6 那样的关系图。

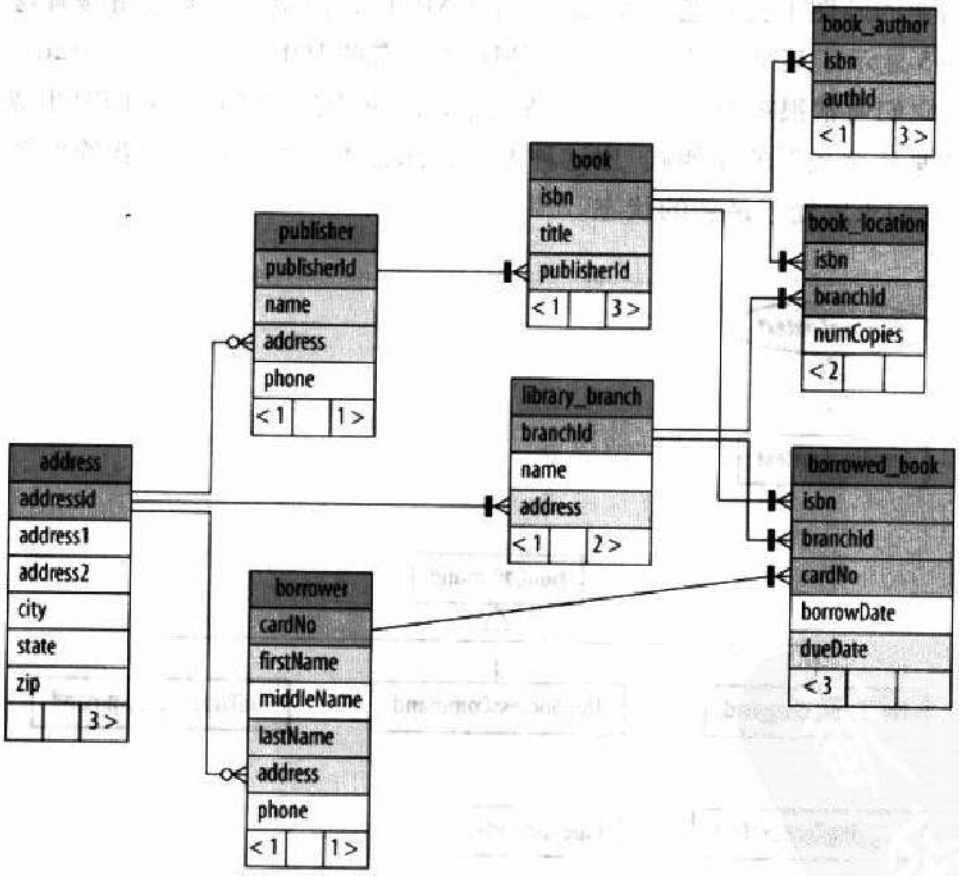


图 5-6：SchemaSpy 产生的关系图

注 10：在 *http://schemaspy.sourceforge.net/* 下载。

小结

提示：

重复是软件开发中最大的阻力。

重复以悄然声息的方式潜入项目中。有时需要足够的智慧来找出解决它的办法。Glenn Vanderburg，一位睿智的软件开发先行者，精确地概括了它：

重复是邪恶的。我是说，邪恶的！

疯狂地追求 DRY 具有几种正面的效果：

- 你逐渐精于重构。
- 你最终会得到一个令人满意的设计——通常并非最佳，但是比团队通常凭空想出的那些好多了。
- 你会重新发现了一些更通用的模式，并因此真正理解它们。
- 团队中有经验的人有更多机会来传授各式各样的封装机制、设计策略等，因为初学者经常无法找出消除重复的办法而不得不寻求帮助（针对具体问题解释这些知识通常会有更好的效果）。

随着程序员的成长，他会发现这条原则有时可以稍稍放松，因为如果太过于强调消除重复，有时会付出可读性或性能方面的代价。但这并不会改变一个事实：DRY 是写出良好代码的根本原则。

第二部分

实践

第二部分中的章节提出了很多改进代码的建议。这些建议大多不限于具体的语言、领域或开发方法。很可能你早已尝试过这些实践中的一部分（甚至是大部分），但就算你一丝不苟地管理着每个对象的生命周期（好公民），本书里的解决方案还是可能令你耳目一新。如果你发现某些内容已经很熟悉了，大可以放心地扫读甚至跳过某些章节，但我还是要给你一个善意的提醒：我在书中不时地加上了一些惊喜，就是为了吸引你的视线。

第 6 章

测试驱动设计

单元测试是一项提升代码质量的极佳实践。经过测试的代码能更好地保证编码意图和实际结果相符。测试驱动开发，因为坚持先写测试后写代码的方式，带来了更多的好处。当把软件工程和其他工程学科进行比较时（经常需要一些好的隐喻来表述），它们之间的重要差异就会显现。在软件科学领域，我们还没有积淀很久的数学理论可以依赖，软件开发科学的历史也还很短(而且我们可能永远也无法达到传统工程那么高的水平)。当然，我们也不能简单地以“拿来主义”利用传统工程中的“规模效应”。例如，金门大桥的建造使用了一百多万个铆钉，你可以期望设计大桥的工程师们都了解每个铆钉的抗压特性，然后用那个数字乘以1 000 000即可得到整座大桥的抗压能力。同样，一件软件可能也有1 000 000件组件，但是它的每件组件都是不同的。作为软件开发人员，我们不能利用“常规”工程师们所使用的“规模”和乘法来度量，但我们的确有一个优势，那就是：我们可以很简单地生产组件，并通过编写测试来验证它们正如我们所期望的一样工作。因为编写软件来测试软件是极其廉价的，我们可以利用各种级别的测试对软件进行核查：比如单元测试、功能测试、集成测试以及用户可用性测试等。

提示：

测试代表着软件开发行为中工程式的严谨部分。

严格遵守TDD（测试驱动开发）的方式还会给设计带来非常多的好处，以致于我经常把TDD说成是测试驱动设计。TDD迫使你用不同的思维方式来看待代码：不是先写一堆代码然后为它们编写测试，而是在编写代码之前先考虑测试流程。TDD建立了一种“消费意识”：当你编写一个单元测试时，其实你正在创建待开发代码的第一个“消费者”。这会让你去思考，在测试之外我们会如何使用这个类。每个开发人员都会有这样的经历：在写一个大类的过程中，一路伴随着很多假设或者臆断。然而当你真的开始使用这个类时，你才认识到一些假设是错误的。然后，你只好去重构代码。TDD要求你在写代码之前先创建它的第一个“消费者”，这会让你去思考其他代码最终会如何使用这些待开发的代码。

TDD也迫使你去模拟（mock）被依赖的对象。比如，如果你要创建一个Customer类，它的addOrder方法需要跟一个Order对象协作。如果要在addOrder方法中创建被依赖的对象，那么就需要Order对象在开始测试Customer类之前已经存在。而Mock对象允许你为测试创建“虚假”的被依赖类。但同时，因为它们都是第一次被开发，你必须得考虑两个对象之间会如何相互作用。

TDD鼓励你通过变量或参数的方式传递被依赖对象，而使被依赖对象的创建放置在其他地方（因为如果你在方法内调用构造函数，你就不能模拟被依赖对象）。这往往会促使

你把对象的创建封装在一个定义得更好的层次上，从而使跟踪对象的分配和引用更加简单（这样你就不会一不小心在其他地方错误地持有一些对象的引用，而使它们永远也跳不出那个范围，导致它们不会被垃圾收集）。TDD事实上还会迫使你编写小而内聚的方法，因为测试一般只测试一件事情。同时，让每个方法只做一件事情，会使它们更好地遵守SLAP法则（我们将会在第13章中谈到）。

不断演化的测试

让我们举一个例子来看看TDD对设计所带来的优化作用。为了演示它们，我们需要一个不是太简单而显得粗浅、但又不是太复杂而导致被涉入细节中的例子。一个完美的候选方案就是“寻找完全数”：完全数是指因数之和（除了它自己本身之外）等于它本身的数字。比如，6就是一个完全数，因为把它的因数（1、2、3和6）去除6之后相加等于6本身。下面写一些Java代码来寻找完全数。

以TDD的方式写单元测试

下面的代码应用了一些简单的逻辑和小小的算法优化，但并不是在测试驱动的方式下编写的：

```
public class PerfectNumberFinder {

    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < Math.sqrt(number) + 1; i++)❶
            if (number % i == 0) {
                factors.add(i);
                if ( number / i != i )❷
                    factors.add(number / i);
            }

        // sum the factors
        int sum = 0;
        for (Integer i : factors)
            sum += i;

        // decide if its perfect
        return sum - number == number;
    }
}
```

❶ 如果能成对地获取数字，只要遍历到数字的平方根即可。比如，对于数字28，如果找到了因数2，那么也同时找到了对称的因数14。

❷ `number / i != i`这条判断是为了确保同一个数字不会被放入列表两次。由于我们成对地获取数字，如果两个因数相等会发生什么？以16为例，在搜寻到因数4时，它应该只被放入列表一次。

这段代码只包含一个静态方法，它会根据传入的数字是否是完全数而返回`true`或`false`。第一步，先取得因数。我们知道1和数字本身就是因数，所以先把它们加上。然后，使用一个`for`循环来搜索，直到这个数字的平方根。这是一个小小的优化，因为如果能成对地获取因数，那么只要搜寻到平方根即可。

没错，它仅仅是一段很小的代码。如果应用了TDD，它看起来会有什么不同？第一个测试应该是超级简单的。首先，我只要能正确地拿到数字1的因数即可。

```
@Test public void factors_for_1() {
    int[] expected = new int[] {1};
    Classifier c = new Classifier(1);
    assertThat(c.getFactors(), is(expected));
}
```

我使用了带有Hamcrest（注1）匹配器的JUnit 4.4测试框架来编写测试（Hamcrest匹配器提供了更多类似英语的语法，比如`assertThat(expected,is(c.getFactors()))`）。这个测试看起来怎么用处不大呢？它似乎过于简单了。其实，像这样确实非常简单的测试并不是为了真正测试，它只是帮助你正确地搭建起基础设施。比如，把测试库添加到classpath，创建一个名叫`Classifier`的类，解决所有包的依赖问题等。太多工作要做了！虽然只是一个看起来很傻很简单的测试，却可以帮助我们在开始思考如何测试真实问题之前，把所有的基础设施搭建起来。

在这个测试通过之后，我把它稍微改进了一下，使它看起来更像一个真实的测试。数组被替换成了大小可变的`List<Integer>`：

```
@Test public void factors_for_1() {
    List<Integer> expected = new ArrayList<Integer>(1);
    expected.add(1);
    Classifier c = new Classifier(1);
    assertThat(c.getFactors(), is(expected));
}
```

通过这个测试之后，我应该把它保留吗？没错，应该保留！我把这些非常简单的测试叫做金丝雀测试（canary test），因为它正如被矿工带入煤矿的金丝雀会警示煤气的侵害一样，这个测试也会持续不断地对实况进行检查并发出警告。如果这个测试某一天失败了，那么你的代码基础设施肯定就存在严重的问题了，比如：一个JAR包被放置到了错误的

注1：从 *http://code.google.com/p/hamcrest/* 下载。

地方，或者代码本身被移动了等。这些简单的测试可以提醒你是不是有些基本的东西被破坏了。

下一个测试，我想着手查找一个数字的真实因数：

```
@Test public void factors_for_6() {
    List<Integer> expected = new ArrayList<Integer>(
        Arrays.asList(1, 2, 3, 6));
    Classifier c = new Classifier(6);
    assertThat(c.getFactors(), is(expected));
}
```

这是我想写的测试，虽然很短，但它表述了很多不同的功能。为了让这个测试通过，我必须知道怎样辨别一个数字是不是因数，怎样计算因数，以及怎样获取已经找到的因数。一个简单的测试揭示了所有需要实现的功能，这样的事情其实经常发生在TDD的过程中。要实现这些令人望而却步的大量功能，最好的办法是：先跳出来，然后想想怎样让这个测试通过。随着逐层剥开，我创建了如下的测试（以及使之通过的相应代码）：

```
@Test public void is_factor() {
    assertTrue(Classifier.isFactor(1, 10));
    assertTrue(Classifier.isFactor(5, 25));
    assertFalse(Classifier.isFactor(6, 25));
}

@Test public void add_factors() {
    Classifier c = new Classifier(20);
    c.addFactor(2);
    c.addFactor(4);
    c.addFactor(5);
    c.addFactor(10);
    List<Integer> expectation = new ArrayList<Integer>(
            Arrays.asList(1, 2, 4, 5, 10, 20));
    assertThat(c.getFactors(), is(expectation));
}
```

在Classifier类中，我们已经把1和数字本身（20）自动添加上了，所以现在只要把余下的所有因数加上即可。在实现了Classifier类中的代码来生成一个ArrayList之后，第一个测试仍旧通过，但第二个测试却失败了，它的错误信息是：

```
java.lang.AssertionError: Expected: is <[1, 2, 4, 5, 10, 20] got: <[1, 20,
2, 10, 4, 5]>
```

失败的原因在于因数是成对获取的，这就产生了一个很基本的问题：是不是应该在Classifier类中加上一些代码来去除重复，还是现在使用的提取方法本身就是错误的呢？其实，最终取得的结果实际上是一个集合，并没有固有的顺序，这提醒我应该修改Classifier类，让它使用HashSet而不是ArrayList。TDD的优点就在于能帮助

你在早期发现一些错误的假设，在那时代码还不是很多，重构也不致于太痛苦。在这个例子里面有一件有趣的事情要注意：addFactor()方法在类Classifier中实际上是私有的。我会在第12章的“Java和反射”一节中解释怎样测试这些私有方法。

顺着这个思考过程，我最终得出了Classifier类的处理逻辑，实现如下：

```
public class Classifier {
    private int _number;
    private Set<Integer> _factors;
    public Classifier(int number) {
        if (number < 0) throw new InvalidNumberException();
        setNumber(number);
    }

    public Classifier() {}

    public Set<Integer> getFactors() {
        return _factors;
    }

    public boolean isPerfect() {
        return sumOfFactorsFor(_number) - _number == _number;
    }

    public void calculateFactors() {
        for (int i = 2; i < Math.sqrt(_number) + 1; i++)
            addFactor(i);
    }

    private void addFactor(int i) {
        if (isFactor(i)) {
            _factors.add(i);
            _factors.add(_number / i);
        }
    }

    private int sumOfFactorsFor(int number) {
        calculateFactors();
        int sum = 0;
        for (int i : _factors)
            sum += i;
        return sum;
    }

    private boolean isFactor(int factor) {
        return _number % factor == 0;
    }

    public int getNumber() {
        return _number;
    }

    public void setNumber(int value) {
```

```
            _number = value;
            _factors = new HashSet<Integer>();
            _factors.add(1);
            _factors.add(_number);
        }
    }
```

代码的度量

如果拿TDD版的代码和非TDD版的比较一下，你会发现TDD版的代码量比较大，其中有很多小方法。更多更小的方法是件好事。当你读到这些方法名时，会有一种看到很多原子操作的感觉。事实上，如果你留意一下最初的代码中的注释，你就会发现，在TDD方式下，这些注释所表述的功能（或许更多）都被实现成为方法了。

6个月之后，当你需要再次修改这块代码时，就可以放心大胆地去改动了。如果一些东西被破坏了，你很容易在每个方法的少量代码中找出错误。测试驱动下开发的代码，其方法名描述的就是一个原子操作，所以如果测试失败了，你会更快地发现是什么破坏了测试。而在处理一个长方法时，要找到错误的根源就需要花费更多的时间，因为在你开始修改之前要对整个方法的上下文理解清楚。而理解一个只有三行代码的方法就要快得多，几乎不需要花费什么时间。所以，如果你发现代码中加入了一些注释，那说明这个方法应该更加精炼。带有很多注释的长方法，往往意味着解决方案没有被很好地组织。通过把注释重构成方法，就可以清除这些又长又臭的大方法。

> **提示：**
>
> 把注释重构成方法。

McCabe的圈复杂度计算工具是（为数不多的）代码测量工具中非常有用的一个（参见下面的版块：圈复杂度）。非TDD版PerfectNumberFinder类的平均圈复杂度是5（类中只有一个方法，所以这个方法的圈复杂度也就是类的平均复杂度）。而TDD版代码的平均圈复杂度是1.5，这意味着它的方法更加简单（同样，这个类也是）。

圈复杂度

Thomas McCabe创造了一种叫做圈复杂度的代码度量工具来测量代码的复杂度。它的计算公式非常简单：边数－节点数＋2，边代表执行路径，节点代表代码的行数。比如，下面的代码：

```
public void doit() {
```

```
        if (c1) {
          f1();
        } else {
          f2();
        }
        if (c2) {
          f3();
        } else {
          f4();
        }
    }
```

如果把这个方法用一个流程图（见图6-1）画出来，你会发现它有8条边和7个节点，这意味着这段代码的圈复杂度是3。

有很多用于测量圈复杂度的工具（见第7章中介绍的一些开源工具），其中包含IntelliJ IDE的“分析”菜单。

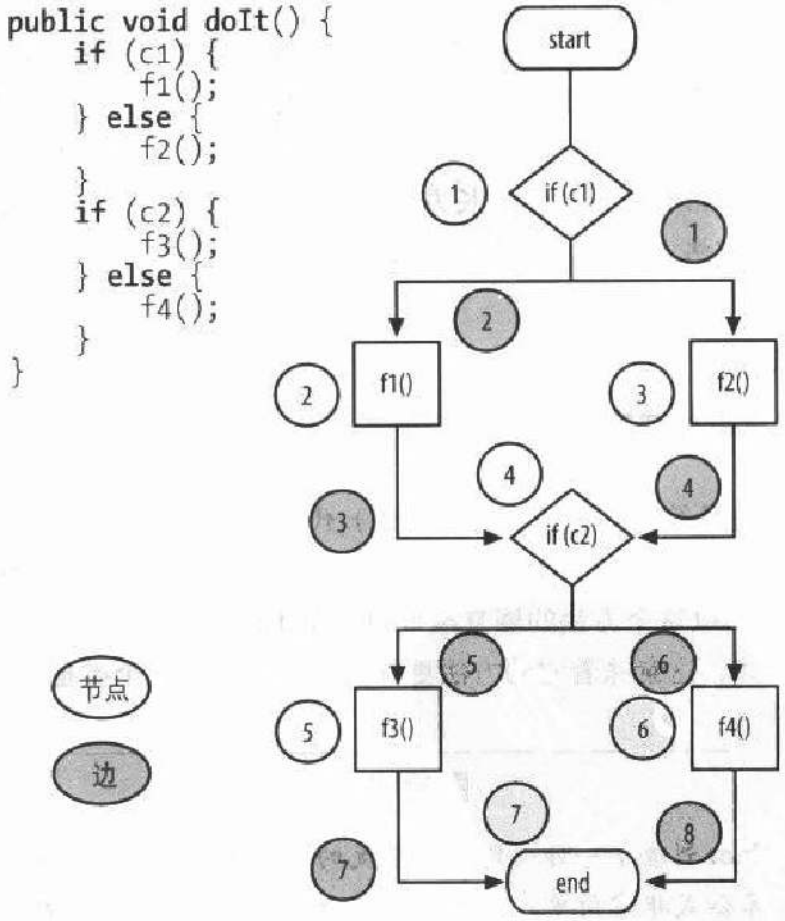


图6-1：用一个流程图来表示圈复杂度的计算

对设计的影响

对设计的影响是TDD对软件设计所带来好处的最后一个重要指标。如果说，某一天那些永不满足的用户又跳出来了，并决定他们想要的不仅可以寻找完全数，而且还应该可以计算过剩数（除本身之外的因数之和大于它本身的数）和亏数（除本身之外的因数之和小于它本身的数）。在第一个代码版本中，你需要进行大量的重构，直到把它打散成类似于第二个版本的样子，才能进行修改。而对于TDD版本的代码呢？只要加上两个方法即可：

```
public boolean isDeficient() {
    return sumOfFactorsFor(_number) - _number < _number;
}

public boolean isAbundant() {
    return sumOfFactorsFor(_number) - _number > _number;
}
```

因为所有的“积木”都已经存在了。测试驱动开发的代码往往有更多可以重用的元素，因为测试驱动开发驱使你写出更加内聚的方法，而这些高内聚的方法就是代码中真正可重用的“积木”。

TDD通过以下形式，来改进代码的设计：

- 它帮你养成更好的代码“消费意识”，因为在开始写代码之前，就需要创建第一个“消费者”。
- 保持对极其简单的一些情况进行测试（以及持续不断地测试），能在你不小心把至关重要的基础设施破坏了的时候及时地发出警告。
- 对边界情况的测试是必不可少的。那些难以被测试的代码，可以把它们重构得更加简单，而如果真的无法简化它们的话，也应该想办法严格地测试它们，不管多么困难。因为复杂的事情更需要测试！
- 永远保持把测试作为构建过程的一部分。软件中最奇怪的事情，莫过于在修改一块完全不相关的代码时，不小心引发的“副作用”。把单元测试作为回归测试的一部分，能帮助你迅速地发现那些“副作用”。有了单元测试这张“安全网”，确实能替你节省很多的时间和力气。
- 有一套强健的单元测试，允许你进行一些“异想天开”的重构游戏（进行大量的修改，然后运行测试来看看这些修改所带来的影响）。记得第一次和一些已经习惯于单元测试的开发人员一起动手开始修改代码时，我也是非常紧张，因为大量的修改往往会破坏很多东西，但他们看起来丝毫没有犹豫。逐渐地，我也放下心来，因为我慢慢地认识到：有了测试的保证，完全可以放心大胆地去修改代码。

代码覆盖率

最后一个至关重要的关于测试的话题是代码覆盖率（code coverage）。代码覆盖率是指被测试执行过的代码行数和分支数。现在，几乎每种语言都有一些开源和商业的代码覆盖率计算工具。

对于编译式语言（像 Java 和 C#）的代码覆盖率计算，首先要在已编译的字节码上进行标记处理，然后开始在被标记的代码上运行整套测试，来测量哪些代码被执行了。测量的详细结果会先以一些中间形式保留下来，并最终生成报告。这个报告会显示行和分支的测试覆盖率。以上的内容，都总结在图 6-2 中。

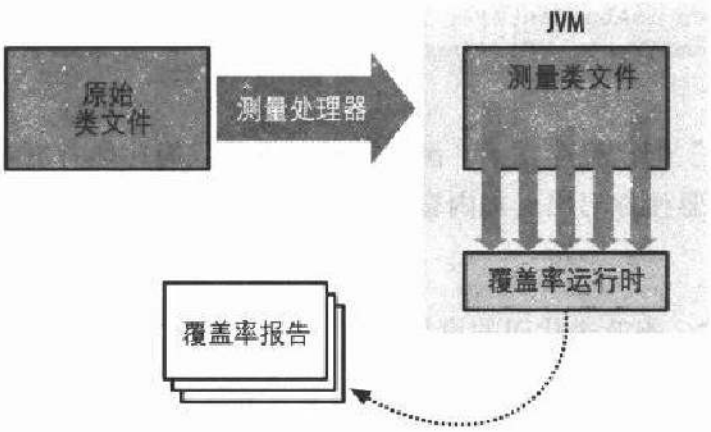


图 6-2：如何生成代码覆盖率

对于动态语言，这个过程会有细微的差别，但最终结果都是一样的：得到一个代码覆盖率报告，显示有多少代码被测试执行了。报告会出现在你的编辑器中，或者以 XML 和 HTML 的形式提供给你。

这个数据是非常关键的，因为它会告诉你哪些代码没有被测试执行过。测试是软件严谨性的保障，没有被测试的代码，往往是错误最有可能存在的地方。如果你是的TDD的忠实拥护者，那你的所有代码都会被自动地测试到 —— 当然，一些难以考虑到的意外情况除外（你也应该为它们加上测试）。

很多开发人员都会问，代码覆盖率到底要多高才能达到可接受的水平。我曾经很乐观地以为，80%以上的覆盖率已经足够了。然而，我注意到一个有趣的现象：如果只有 80% 的覆盖率，那些最需要被测试的代码往往都没有被测到。即使是很审慎的开发人员，也会写出一些复杂的代码，生成测试覆盖率报告，并说："呦！ 82.3%。足够了，我不再需要为那块'复杂'的代码添加更多的测试了！"

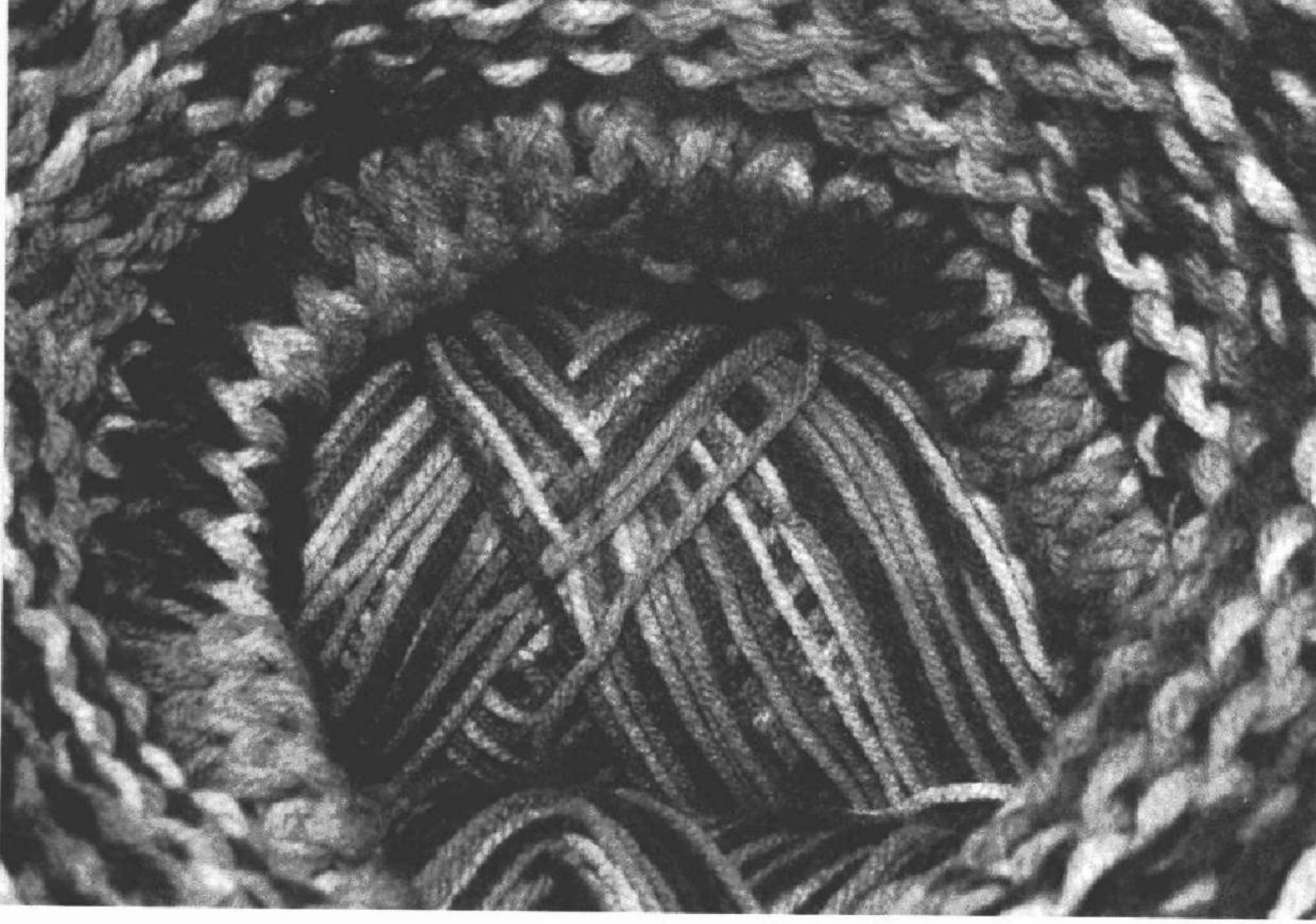
最终，我得出一个结论，任何少于100%测试覆盖率的代码都处在危险之中。只要你坚持那个最高标准，你永远也不会觉得一些代码“太复杂以致于难以被测试覆盖”。首先，你必须编写简单的代码，但如果真的遇到一个非常复杂的情况，你也必须要想办法来测试它。只要记住“没有任何东西是不需要测试的”，那么你就会对代码的健康更加勤勉。

但是，如果遇到有一个很大的代码库，却没有任何测试的情况，怎么办？当然，为了提高测试覆盖率，而停止几个月的开发工作，是极其不现实的幻想。那么，首先设定一个不远的日期（比如下个礼拜四）。然后，让整个团队保证，从那天开始，代码覆盖率要永远保持增加。这意味着：

- 所有的新代码都要达到100%的单元测试覆盖率（当然，最好是用测试驱动开发）。
- 每当修复一个错误的时候，都要为其添加相应的测试。

要达到100%代码覆盖率的目标，需要大量的努力。你需要为所有的新代码编写测试（这会保持代码的简单性），而且在发现错误的时候，也要为其添加测试以减少错误再次发生的可能性。

就如我刚才所说，100%的单元测试代码覆盖率是一个很难达到的标准。然而，我确实曾在一些项目中成功地做到了，而且它也确实帮助我们改进了代码的客观特征（客观特征可以用静态分析和其他一些方法来测量，见第7章）。



第 7 章

静态分析

如果你正在使用一种静态类型语言（例如Java或C#），你就有一种隔离和发现某几类bug的强大手段 —— 这些bug通过代码评审或是别的传统方式很难发现。静态分析（static analysis）就是用软件工具对程序代码进行验证的机制，通过它可以找出其中存在的已知的bug模式。

静态分析工具大致分两类：对编译结果（class文件或字节码）进行分析，或对源代码进行分析。本章分别举例介绍这两类工具。这些例子大多是Java的，因为Java世界里有大量免费的静态分析工具。但这些技术本身并不局限于Java代码，其他各种主流的静态类型语言都有相应的工具。

字节码分析

字节码分析的目标是在编译后的代码中找出已知的bug模式。这意味着两件事：首先，人们已经对某些语言进行了足够多的研究，以致于能够从编译后的字节码中找到一些常见的bug模式；其次，这些工具不可能找出所有bug —— 它们只能找到那些已经被识别出固定模式的bug。但这并不表示这些工具没多大用，因为它们能找到的某些bug用其他方式（例如耗费几个小时毫无头绪地盯着调试器）极难发现。

FindBugs就是这样的一个工具，它是由马里兰大学发起的一个开源项目。FindBugs有几种工作模式：命令行、Ant任务以及图形界面。图7-1就是FindBugs的GUI。

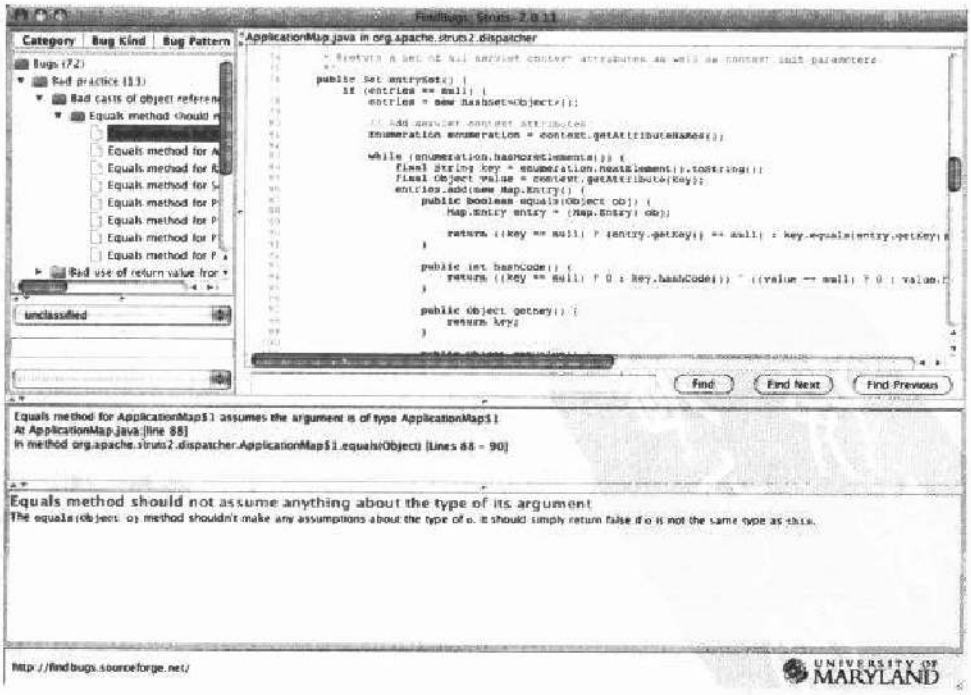


图7-1：FindBugs的图形化客户端

FindBugs会寻找以下几类bug：

正确性

可能是逻辑错误的地方

不良实践

违反基本编码实践（例如，覆盖equals()方法而没有同时覆盖hashCode()方法）

迷惑

难懂的代码、奇怪的用法、反常的做法、糟糕的代码

为了展示FindBugs的效果，我得先找到一个靶子，于是我下载了著名的开源Web框架Struts，然后用FindBugs对它做了一番检查。FindBugs找出了一些（可能是）“不良实践”的征兆：一些代码违背了“Equals方法不应对传入参数的类型有任何假设”这一原则。在实现equals()方法时，Java推荐的最佳实践是检查传入对象是否与this对象同一类型，以确保等义性比较是有意义的。下面就是Struts中违反了这一原则的代码，它位于*ApplicationMap.java*文件中：

```
entries.add(new Map.Entry() {
   public boolean equals(Object obj) {
       Map.Entry entry = (Map.Entry) obj;

       return ((key == null) ? (entry.getKey() == null) :
            key.equals(entry.getKey())) && ((value == null) ?
                (entry.getValue() == null) :
                value.equals(entry.getValue()));
   }
```

之所以说这只是一个征兆，是因为这段代码出现在一个匿名内部类中，也许作者能够确保传入参数的类型。但不管怎么说，这多少有些不可靠。

下面这个被FindBugs发现的bug就是确凿无疑的了。下列代码出自*IteratorGeneratorTag.java*：

```
if (countAttr != null && countAttr.length() > 0) {
       Object countObj = findValue(countAttr);
       if (countObj instanceof Integer) {
           count = ((Integer)countObj).intValue();
       }
       else if (countObj instanceof Float) {
           count = ((Float)countObj).intValue();
       }
       else if (countObj instanceof Long) {
           count = ((Long)countObj).intValue();
       }
```

```
else if (countObj instanceof Double) {
    count = ((Long)countObj).intValue();
}
```

仔细看看最后一行。FindBugs把它列入“正确性”一类，它尝试进行“不可能的转型”：上面代码的最后一行必定会抛出转型异常。实际上这段代码无论如何都是有错的：它首先要求`countObj`的类型是`Double`，然后又把它转型成`Long`。再看看前面的`if`语句你就会明白到底是怎么回事了：复制粘贴产生的错误。这种错误在代码评审中很难发现，因为你很容易一扫而过。显然Struts的代码库里没有任何单元测试执行过这行代码，不然问题马上就会暴露出来。更糟糕的是，同样的代码在Struts的代码库中出现了三次：前面提到的这一次在*IteratorGeneratorTag.java*中，还有两次在*SubsetIteratorTag.java*中。原因是什么呢？你已经猜到了。同样的代码被复制粘贴到三个地方。（FindBugs并不能发现复制粘贴的代码，我只是注意到这些有问题的代码看起来是如此相似。）

可以通过Ant或者Maven来自动执行FindBugs，并将其变成构建流程的一部分，于是你就有了一份非常实惠的保险：至少FindBugs所知道的那些bug能被马上发现。它不承诺能找出代码里所有缺陷（也不能让你免于编写代码测试），但它至少能帮你找出一些烦人的bug。它甚至还能找到一些单元测试也难以使之暴露的bug，例如线程同步问题。

> **提示：**
>
> 静态分析工具提供了便宜实惠的验证手段。

源代码分析

从名字上就能看出来，源码分析是从源代码中搜寻bug模式。下面我要展示的工具是PMD，这是一个开源的Java源代码分析工具。PMD提供了命令行的版本，支持Ant，而且有各种主流开发环境的插件。它主要搜寻代码中存在的以下几类问题：

可能的bug

例如，空的`try...catch`块

死代码

没有被用到的本地变量、参数和私有变量等

欠优化的代码

无节制的字符串操作

过于复杂的表达式

以复制粘贴的方式来“复用”代码

重复代码（需要一个叫CPD的辅助工具）

以复制粘贴的方式来“复用”代码

PMD介于纯代码风格检查器（例如，CheckStyle，用于确保代码符合某些风格规定，比如有适当的缩进）和字节码检查器（例如，FindBugs）之间。我们可以用下列Java方法来试试看PMD能发现哪些问题：

```
private void insertLineItems(ShoppingCart cart, int orderKey) {
    Iterator it = cart.getItemList().iterator();
    while (it.hasNext()) {
        CartItem ci = (CartItem) it.next();
        addLineItem(connection, orderKey, ci.getProduct().getId(),
            ci.getQuantity());
    }
}
```

PMD会指出这个方法的可改进之处：第一个参数（ShoppingCart cart）可以加上final修饰，这样编译器可以帮你做更多的事：由于Java中所有对象都是按值传递的，将cart变量标记为final就可以确保在方法内部无法把另一个对象的引用赋值给它，如果尝试这样做就会引发编译错误。PMD还提供了几个类似这样的、能够让现有工具（例如编译器）工作得更有效的改进建议。

PMD的辅助工具CPD（Cut-Paste Detector）可以从源码中搜寻到可疑的重复代码（例如Struts里复制粘贴的代码）。CPD有一个基于Swing的用户界面，能直观地显示搜寻状态和结果（如图7-2）。当然，这直接反映出我们在第5章里讨论过的DRY原则。

类似的分析工具大多提供可定制的API，允许你创建自己的规则集（FindBugs和PMD都有这样的API）。这些工具一般也会提供交互模式，同时又可以将它们作为自动化流程（例如持续集成）的一部分来执行。对每个程序员每次提交的代码都用这些工具来检查一遍，就能避免很多低级错误——而且成本极其低廉。别人已经做了很多工作来识别出这些低级错误，你应该坐享其成。

用Panopticode生成统计数据

“统计”这个主题已经超出了本书的范畴，不过“如何高效生成统计数据”还是值得一提的。对于静态语言（例如Java和C#），我一贯主张持续搜集统计数据，以确保尽快发现问题解决问题。换句话说，我会在持续集成中运行一组统计工具（包括FindBugs和PMD/CPD）。

要把这一大堆工具组装起来还真得费一番工夫。我希望所有这些基础设施能预先配置好，

就像 Buildix 那样（参见第 4 章“不要重新发明轮子”一节）。这就是 Panopticode 的价值所在。

我的一个同事（Julias Shaw）也遇到了同样的问题。不过他没有像我一样抱怨，而是动手来解决。Panopticode（注 1）是一个开源项目，它预先配置了很多常用的统计工具。Panopticode 由两部分组成：一个 Ant 构建文件，以及预先配置好的一大堆开源项目以及它们的 JAR 文件。你只需指定源代码路径、库文件路径（构建项目所需的 JAR 文件所在的位置）和测试目录，然后执行 Panopticode 的构建脚本，它就会完成剩下的事情。

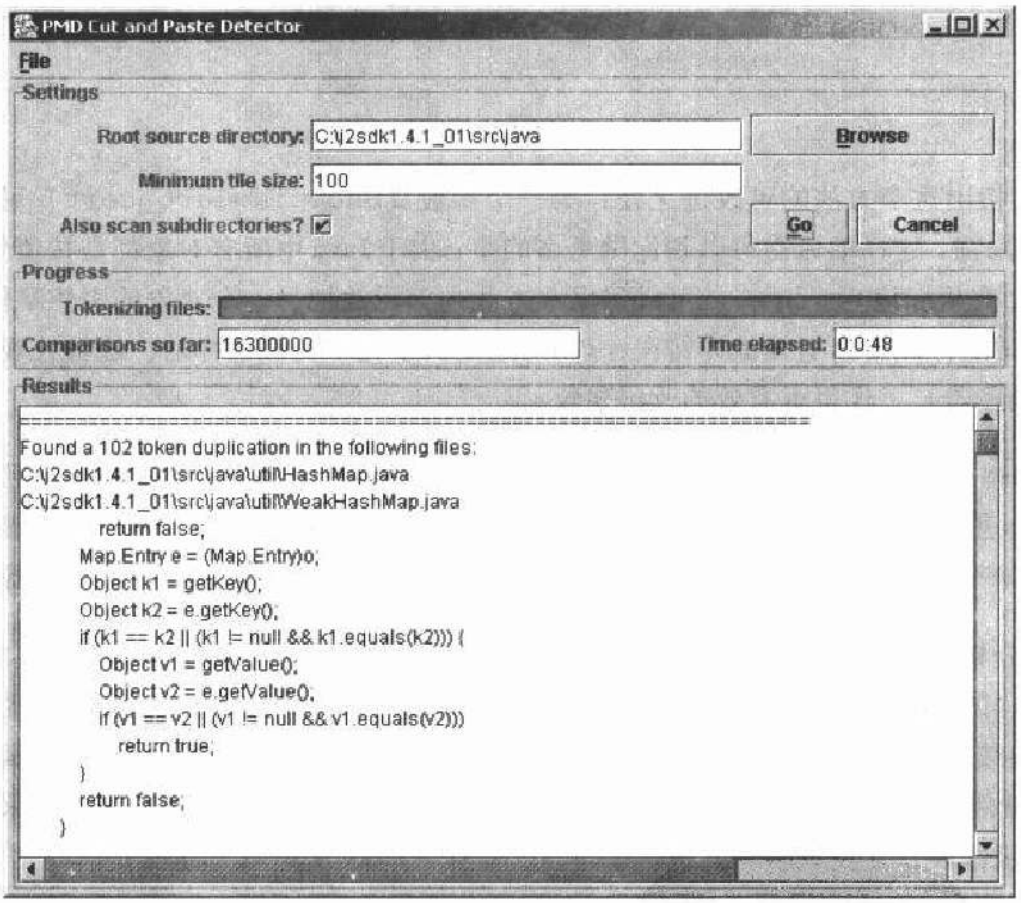


图 7-2：PMD 的“复制粘贴检测器”

Panopticode 包含了下列预先配置好的统计工具：

Emma

开源的代码测试覆盖率统计工具（参加第 6 章“代码覆盖率”一节）。而且只要在 Panopticode 的构建文件中修改一行就可以切换到 Cobertura（另一个 Java 的开源代码测试覆盖率统计工具）。

注 1：可以在 *http://www.panopticode.org/* 下载。

CheckStyle

开源的代码风格检验工具。在Panopticode的构建脚本里可以定制规则集。

Jdepend

开源的统计工具，可以在包层面上搜集统计数据。

JavaNCSS

开源的圈复杂度统计工具（参加第6章“圈复杂度”版块）。

Simian

商业的重复代码搜寻工具。Panopticode所带的这个版本有限时15天的使用许可，之后你可以选择购买或删除它。Panopticode计划在将来用CPD作为该工具的替代方案。

Panopticode Aggregator

Panopticode生成的报表，以文字和图形（树状图）的格式来展示所有统计的结果。

当你运行Panopticode时，它会把你的代码审查一遍，然后生成反映代码统计结果的报表。它还会生成一些漂亮的树状图，这些图形都是复杂的可缩放矢量图（SVG，现在大部分浏览器都支持SVG图片）。这些树状图能给你两方面的帮助。首先，你可以直观地总览某项统计的结果。例如图7-3所示的树状图就展示了CruiseControl项目的圈复杂度，其中每个小方块代表一个方法的圈复杂度，不同的灰度代表不同的结果值范围。粗白线代表包边界，细白线代表类边界，每个方块代表一个方法。

树状图的另一个好处是交互性。这不仅是一张漂亮的图片，你还可以与它交互。在浏览器中打开这幅树状图，再点击其中的任意一个方块，对应这个方法的详细统计数据就会显示在右边。借助这些树状图，你可以对代码进行分析，找出到底是哪些类和方法有问题。

Panopticode提供了两项很好的服务。第一，你不需要一次又一次地为每个项目配置同样的工具。如果项目结构大同小异的话，配置Panopticode只要5分钟就够了。图形化的树状图则是第二个好东西，因为它能够很好地扮演“信息辐射器”的角色——在敏捷项目中，人们会在显眼的地方（例如咖啡机旁边）摆出重要的项目状态信息，这种机制就叫“信息辐射器”。要了解项目的状态，团队成员不需要打开邮件附件，在泡咖啡的路上就能看到了。

Panopticode生成的一个树状图是代码测试覆盖率图（参加第6章“代码覆盖率“一节）。如果覆盖率图上出现一大块黑色区域，那就不妙了。在开始尝试提升覆盖率之前，首先找一个尽可能大的彩色打印机，把覆盖率图打印出来（实际上对于大多数项目来说，第

一次打印用黑白打印机就行了，因为那时候的覆盖率图反正也是一片黑），挂在一个醒目的地方。过一段时间后（比如到了下一个里程碑时），把最新版本打出来，并挂在第一份图的边上。树状图是开发人员无可回避的驱动力：没有人愿意看到树状图突然如往日那样全黑一片。同时，树状图也很好地展示给管理人员：代码的测试覆盖率正在逐步提升。管理人员就是喜欢这样大而显眼的图表，更何况这张图表真正提供了有用的信息！

图 7-3：用 Panopticode 给 CruiseControl 项目生成的圈复杂度图

动态语言分析

动态语言在大多数情况下开发效率更高，但它们缺乏像静态类型语言那样丰富的分析工具。动态语言的分析工具更难开发，因为它们没有静态类型系统之类的特性可供依赖。

动态语言世界里开发分析工具的努力主要围绕着圈复杂度（因为这项指标的分析方法对几乎所有基于代码块的语言都一样）和代码测试覆盖率进行。例如，说在 Ruby 世界里，rcov 就是一种常用的覆盖率统计工具。实际上 Ruby on Rails 就已经预先配置好了 rcov

（参见图15-1的rcov报表）。至于圈复杂度统计，可以使用开源的Saikuro（注2）来实现。

由于缺少“传统的”静态分析工具，Ruby程序员们也学聪明了，他们发起了一些有趣的项目，用一些非传统的方式来度量代码质量。首先值得一提的是flog（注3）。它用“ABC”值来度量代码：赋值（assignment）、分支（branch）和调用（call），并且给调用加上权值。flog会给方法中的每行代码赋一个权值，然后生成类似下面这样的结果报告（这是针对第15章“重构SqlSplitter使之更具可测试性”一节中的SqlSplitter示例程序运行flog的结果）：

```
SqlSplitter#generate_sql_chunks: (32.4)
     20.8: assignment
      7.0: branch
      3.4: downcase
      3.0: +
      2.9: ==
      2.8: create_output_file_from_number
      2.8: close
      2.0: lit_fixnum
      1.7: %
      1.4: puts
      1.4: lines_o_sql
      1.2: each
      1.2: make_a_place_for_output_files
SqlSplitter#create_output_file_from_number: (11.2)
      4.8: +
      3.0: |
      1.8: to_s
      1.2: assignment
      1.2: new
      0.4: lit_fixnum
```

这个结果表明generate_sql_chunks方法是这个类里最复杂的方法，它的得分是32.4（把它下面列出的值加起来）。这个方法中最复杂的又是与赋值相关的部分。所以如果想简化这段代码，就应该首先从这个方法里大量的赋值语句入手。

Groovy的情况比较特殊，因为尽管它是动态语言，它生成的目标码却是Java字节码，也就是说，你可以用标准的Java静态分析工具来对字节码进行分析。但这样分析的结果可能并不理想，因为Groovy必须生成大量的代理类和包装类，以便实现动态语言的特性，于是你就会在分析报告中看到很多不认识的类。Java统计工具的开发者们已经开始考虑

注2：可以在*http://saikuro.rubyforge.org/*下载。

注3：可以在*http://ruby.sadi.st/Flog.html*下载。

Groovy（例如，代码覆盖率工具Cobertura现在就已经支持Groovy了），但这些工作都仍处在早期阶段。

第 8 章

当个好公民

在改进代码的讨论里面，“公民责任”可能看起来是一个奇怪的话题，但是“公民责任”在这里说的是一个既知道自己的状态、又对周围事物的状态友善的对象。虽然听起来很容易，但是软件开发者常会在不知不觉中违反这个原则。在这一章让我们一起看看几种不同的对公民责任的违规行为，以及一些成为尽职公民的方法。

破坏封装

在面向对象程序设计里，其中一个主要信条就是封装：保护内部域（field）不受外界的干扰。然而，我看到过很多的开发者因为不假思索地开发代码，而破坏了封装的原意。

这里有一个场景：你创建了一个新类，在其中添加几个私有成员变量（域），然后让IDE自动生成一些属性（Java里的getter / setter或C#里的properties），然后才去考虑到底要怎么使用它们。为每一个私有域创建public的属性，就彻底破坏了属性机制的原意。倒不如把所有的成员变量变为public的，因为属性（properties）对你没有任何帮助（反而令你的代码更累赘）。

例如，你有一个`Customer`（顾客）类，它用几个字段来共同表示地址（例如常见的`addressLine`[地址栏]、`city`[城市]，`state`[州、省]、`zip`[邮编]）。如果你为每个字段都创建了赋值方法（mutator），你就制造了让别人把你的`Customer`变为一个不良公民（即它有一个不合法的状态）的机会。在现实生活里，一个顾客有一个不完整的地址是不合常理的。一般来说，他们要么有一个完整的地址，要么就没有地址。不要让代码把Customer对象（它应该反映现实中的顾客）置于对业务而言没有意义的状态。那些取值方法可能没有问题，但是你应该创建一个整体的原子赋值方法，而不是为每一个字段做一个：

```
class Customer {
    private String _adrLine;
    private String _city;
    private String _state;
    private String _zip;

    public void addAddress(String adrLine, String city,
                String state, String zip) {
        _adrLine  = adrLine;
        _city     = city;
        _state    = state;
        _zip      = zip;
    }
}
```

有了一个原子赋值方法，你只需一步就能把对象从一个合法状态切换到另一个合法状态。这样做有以下几个好处：首先，你可以不用写校验代码来确认地址是否合法 —— 如果

你没有任何机会去创建一个非法的地址，你就不用去防止它发生；其次，它使你的顾客模型更加贴近一个真实的顾客。当你的真实顾客改变时，你的代码可以随着它一起改变，因为它们语义上非常相近。

在这里，我们不是盲目地创建属性，而是只有在需要从其他的代码调用这个属性时才创建它。这个策略有几个作用。首先，不会有不需要的代码。开发者很多时候都会创建太多预期性的代码，因为他们在想："反正我以后肯定需要这个，倒不如现在就做了吧。"如果你是用工具来自动创建属性，就算在你需要这个属性之后才创建，也不会让你格外费劲。其次，可以避免让代码有不必要的臃肿。属性在代码里面所占空间很多，这些自动生成的代码会减慢你阅读代码的速度。最后，不用担心要为属性写测试。因为所有的属性都在真正使用这些属性的方法里被调用了，所以它们会自然而然地被其他的测试覆盖。我永远都不会用测试驱动开发属性（在Java里的getter/setter或在C#里的properties）；我只会在需要时才生成它们。

构造函数

在大多数的面向对象语言里，我们都习惯了构造函数的存在。我们只把它当作一个创建对象的机制。其实构造函数有一个更重要的使命：它们告诉你怎样才可以创建出这个类的合法对象。构造函数和这个对象的消费者构成了一个合约性的关系，它显示了要使这个对象合法而需要填入的字段。

很遗憾的是，编程语言的权威们反对有意义的构造函数。大多数的语言实质上都坚持所有类都有一个默认构造函数（不需要任何参数的构造函数）。从公民责任的角度来说，这是不合理的。你什么时候听过你的业务人员说："我们要运送这件东西给这个客户，但是我们没有他的地址。"你不可能运送东西给一个没有内部状态的客户。对象是状态的保持者，一个对象没有状态是不合理的。实际上每一个对象都应该一开始就至少维持一些最小限度的状态。在你的公司里，可能有一个没有名字的客户吗？

要转变这个默认构造函数的习惯是不容易的。很多框架坚持使用它们，如果你不提供便会触怒这些框架。这个"一定要有默认构造函数"的规矩，在Java里的JavaBeans规范里甚至成了金科玉律。如果一个框架或者一个语言规范要一意孤行，那也没办法（除非你可以用一个更加友善的代替它）。在这种情况下，把这个默认构造函数当作一个例外，就像那些加在领域对象（domain object）上的丑陋的序列化代码一样。

静态方法

静态方法有一个很好的用处：它可以作为一个黑箱式的独立、无状态的方法。Java里的Math类很好地说明了静态方法的好处。当你调用Math.sqrt()方法时，你不需要担心因为sqrt()里面的状态改变而使下一次调用返回的是立方根而不是平方根。当静态方法完全无状态时，它们可以很好地工作。但当你把静态和状态混合起来时，你就会遇到麻烦。

静态方法与夏威夷

我曾经在一家讲授Java的咨询及培训公司里工作了很多年。有一次，我有幸得到了一个培训差事之中的“圣杯”：去夏威夷为两组开发人员分别讲授两个课程。根据行程的安排，我在那里逗留了两个星期（每周一个课程），然后回家三周，之后再去为课程的下半部分讲授两周。当然，我需要在夏威夷度过周末，作为一个对公司尽心尽职的好员工，我尽可能地应付了过来。但说到课程方面，还是遇到了一定的难度，因为所有学生都是出身于大型机（mainframe）的背景。我记得有一个学生，他不太明白大括号首尾对应的概念，每当碰到编译错误时，他就在文档末多加一些括号。我去指导他时，发现他在同一行里有一打的闭括号。

言归正传，我熬过了前两周，三周后回来开始下半部课程。其中一个学生立即骄傲地来到我的面前说：“当你不在时，我们领悟了Java是怎么回事。”我很惊讶，想看看他们写的代码。她拿给我看，而我看到的代码全都像这样的：

```
public static Hashtable updateCustomer(
            Hashtable customerInfo, Hashtable newInfo) {
    customerInfo.put("name", newInfo.get("name");
    // . . .
}
```

他们达到了一个我本来以为不可能达到的境地：把Java变成过程式语言！还有弱（loosely）类型的变量。不用说，之后的时间我都在纠正他们使用Java的错误方法。

这段轶事告诉我们静态方法的过度使用如何显示一个人的过程式思维方式。如果你发现自己使用很多的静态方法，你就应该检查一下你的抽象是否正确。

“静态性”和“状态”的邪恶混合常常可在Singleton模式里见到。Singleton模式是用来创造一个只能实例化一次的类，之后的再尝试创建都返回原先的实例。Singleton模式通常是这样实现的（以下是Java代码，但在其他语言中也大同小异）：

```
public class ConfigSingleton {
    private static ConfigSingleton myInstance;
    private Point _initialPosition;

    public Point getInitialPosition() {
       return _initialPosition;
    }

    private ConfigSingleton() {
        Dimension screenSize =
                Toolkit.getDefaultToolkit().getScreenSize();
        _initialPosition = new Point();
        _initialPosition.x = (int) screenSize.getWidth() / 2;
        _initialPosition.y = (int) screenSize.getHeight() / 2;
    }

    public static ConfigSingleton getInstance() {
        if (myInstance == null)
            myInstance = new ConfigSingleton();
        return myInstance;
    }
}
```

在以上的代码中，getInstance()方法首先检查是否已存在一个实例，如没有就创建一个实例，并返回指向它的引用。这个方法不是线程安全的，但在这里我们不讨论线程安全问题，因为这只会把事情复杂化。Singleton模式邪恶的地方在于它内藏的状态使它不能被测试。单元测试需要操纵对象的状态，但我们没有办法去操纵这个Singleton对象的状态。因为在Java里，对象构造的过程是原子性的，所以你只能通过initialPosition方法获得现有的屏幕大小，而没有办法获得其他值来测试这个类。Singleton是面向对象版本的全局变量，而每一个人都知道全局变量是不好的。

提示：

不要创建全局变量，即使是对象层次的全局变量。

说到底，singleton之所以有“臭味”，是因为它承担了两个职责：一个职责是监管自己的实例，另一个是提供配置信息。当一个类里有多个不相关的职责时，它就会有“代码臭味”。

但是，能保证只有一个Configuration（配置）对象是很有用的。那么怎样可以不用Singleton而达到这个效果呢？你可以用一个简单的对象加上一个工厂（factory），让它们各自担负自己的职责。工厂负责监管实例，而这个简单对象（Java上叫POJO，Plain Old Java Object）只管理关于配置的信息和行为。

下面就是更改为POJO后的Configuration对象：

```
public class Configuration {
    private Point _initialPosition;

    private Configuration(Dimension screenSize) {
        _initialPosition = new Point();
        _initialPosition.x = (int) screenSize.getWidth() / 2;
        _initialPosition.y = (int) screenSize.getHeight() / 2;
    }

    public int getInitialX() {
        return _initialPosition.x;
    }

    public int getInitialY() {
        return _initialPosition.y;
    }
}
```

这个类也很容易测试。单元测试和工厂都可以用反射（reflection）来创建这个类。Java里的私有（private）访问控制有点类似建议性质的文档。在现代语言里，有必要时基本上可以通过反射来绕过它。这里就是一个好例子：你不希望让别人直接创建一个实例，所以构造函数是私有的。

以下的代码是这个类的单元测试，包括使用反射来创建实例，以及利用反射访问它的私有域来测试它拥有不同值时的行为：

```
public class TestConfiguration {
      Configuration c;

      @Before public void setUp() {❶
          try {
              Constructor cxtor[] =
                      Configuration.class.getDeclaredConstructors();
              cxtor[0].setAccessible(true);
              c = (Configuration) cxtor[0].newInstance(
                      Toolkit.getDefaultToolkit().getScreenSize());
          } catch (Throwable e) {
              fail();
          }
      }

      @Test
      public void initial_position_set_correctly_upon_instantiation() {
          Configuration specialConfig = null;
          Dimension screenSize = null;
          try {
              Constructor cxtor[] =
                      Configuration.class.getDeclaredConstructors();
              cxtor[0].setAccessible(true);
```

```
            screenSize = new Dimension(26,26);
            specialConfig = (Configuration) cxtor[0].newInstance(screenSize);
        } catch (Throwable e) {
            fail();
        }

        Point expected = new Point();
        expected.x = (int) screenSize.getWidth() / 2;
        expected.y = (int) screenSize.getHeight() / 2;
        assertEquals(expected.x, specialConfig.getInitialX());
        assertEquals(expected.y, specialConfig.getInitialY());
    }

    @Test
    public void initial_postion_can_be_changed_after_instantiation() {
        Field f = null;
        try {
            f = Configuration.class.getDeclaredField("_initialPosition");❷
            f.setAccessible(true);
            f.set(c, new Point(10, 10));
        } catch (Throwable t) {
            fail();
        }
           Assert.assertEquals(10, c.getInitialX());
    }
}
```

❶ setUp()方法利用反射创建Configuration对象，然后调用initialize()来创建一个可用于大多数测试的合法对象。

❷ 可以利用反射来访问_initialPosition这个私有域，看看如果初始位置（initial position）不是默认值的话，会发生什么事情。

将Configuration做成一个简单对象，可以使它更容易被测试，同时也不会影响它的功能。

负责创建Configuration的工厂类同样既简单又容易测试。ConfigurationFactory的代码如下：

```
public class ConfigurationFactory {
    private static Configuration myConfig;

    public static Configuration getConfiguration() {
        if (myConfig == null) {
            try {
                Constructor cxtor[] =
                    Configuration.class.getDeclaredConstructors();
                cxtor[0].setAccessible(true);
                myConfig = (Configuration) cxtor[0].newInstance(
                        Toolkit.getDefaultToolkit().getScreenSize());
            } catch (Throwable e) {
```

```
                throw new RuntimeException("can't construct Configuration");
            }
        }
        return myConfig;
    }
}
```

并不奇怪的是，这段代码跟原先singleton中创建对象的代码看起来很相似。主要的区别是这里的代码只做一件事：监管Configuration类的实例。ConfigurationFactory这个类也很容易测试，如下所示：

```
public class TestConfigurationFactory {

    @Test
    public void creation_creates_a_single_instance() {
        Configuration config1 = ConfigurationFactory.getConfiguration();
        assertNotNull(config1);
        Configuration config2 = ConfigurationFactory.getConfiguration();
        assertNotNull(config2);
        assertSame(config1, config2);
    }
}
```

静态方法还有另一个陷阱：Java允许你通过对象实例来调用它们，这样会引起混淆，因为你不能覆盖静态方法。当你通过一个对象（并非它的类）来调用一个静态方法时，Java不会发出任何警告。当父类和子类一起混合使用时，静态方法也会引起混淆。看以下的例子：

```
Derived d = new Derived();
Base b = d;
int x = d.getNumber();
int y = b.getNumber();
int z = ((Base)(null)).getNumber();
System.out.println("x = " + x + "\ty = "
                   + y + "\tz = " + z);
```

上面的代码假设Base类有一个getNumber()方法，而且Derived这个类是 Base的子类。你可以合法地用上面的任何一种形式去调用getNumber()方法。

静态方法虽然能为我们带来一些好处，但是它的各种陷阱也预示着Java可能应该创造另一种不那么危险的机制。

犯罪行为

当一个反社会罪犯闯进你的社区时会发生什么？ java.util.Calendar就是个Java世界里的反社会罪犯，它呈现了很多对于其他公民不友善的行为。它的工程洁癖盖过了常

理。比如，那些定义月份的常数从0开始计数（和Java其他地方一致），也就是说当你传进2这个数字时，它实际上是三月。我明白从0开始是为了保持一致性，但颠覆一个众所周知的对应关系（月份与其数字）无论如何都是极其荒谬的。

Calendar也不能正确地维持自己的内在状态。当你执行下面的代码时会发生什么呢？

```
c = Calendar.getInstance();
c.set(Calendar.MONTH, Calendar.FEBRUARY);
c.set(Calendar.DATE, 31);
System.out.println(c.get(Calendar.MONTH));
System.out.println(c.get(Calendar.DATE));
```

输出会是2和2。在解读之后，它觉得正确的日期应该是3月2日。你叫它去定义一个“2月31日”的日期，而它却静静地返回“3月2日”。你在何时试过告诉你的朋友：“我约你在2月31日见面”，而他会回答：“你的意思是3月2日，对吧？”Calendar不知道自己的内在状态，它允许你去设定一个不存在的日期，也不抛出任何异常，只是静静地给你一个完全不同的日期。对象应该是状态的保持者，不过Calendar看来对自己的状态一无所知。

为什么Calendar会有这样异常的行为呢？问题在于它允许你分开地设定每一个字段。而它本应该强制对日期执行原子操作，即同时设定年、月、日，那么它就有机会去验证得到的日期是否合法。但是Calendar并没有这么做，因为这样方法签名就会很长。原因是什么？是因为Calendar管理了太多的信息：除了日期外，它还管理着时间的信息。那么你不仅要设日期，而且要设时间，这样就会形成一个长得让人讨厌的方法签名。你有没有试过当人家问你“现在几点？”时，而你却回答：“等等，我要查查日历”？Calendar有太多的职责，既损害了它作为一个状态保持者的职责，又降低了它的可用性。

当社区巡察员看到一个罪犯时会怎么做呢？把他踢出社区！开源的Joda库（注）完全能取而代之。不要创建具有坏公民品质的类，也不要使用它们。当你尝试去克服一些烂代码的奇怪之处时，你也会为自己的代码平添不必要的复杂性。

注：　可以在 *http://joda-time.sourceforge.net/* 下载。

第9章

YAGNI

YAGNI是“You Ain't Gonna Need It”（你不会需要它）的缩写，也是敏捷开发对预想开发的战斗宣言。预想开发（speculative development）的例子可谓俯拾皆是。程序员们对自己说：“我肯定以后会需要这项额外的功能，所以现在就提前把它实现了吧。”这是滑向深渊的第一步。更好的方式是：只开发当下需要的东西。

预想开发会给软件造成伤害，因为它过早地给代码引入了复杂度。正如Andrew Hunt和David Thomas在《The Pragmatic Programmer》（Addison-Wesley）一书中所说，熵会损害软件——“熵”是一个数学术语，用于度量系统中的复杂度。熵是软件的大敌，因为软件系统越复杂，理解代码、修改代码、添加功能就越困难。在物理世界里，事物通常倾向于朝着简单的方向发展，除非外界对其注入能量。软件则正好相反：由于创造软件是如此容易（而且创造复杂的软件和简单的软件所需的物理工作量没有太大区别），因此软件倾向于朝复杂的方向发展。将软件从复杂变回简单可能需要花很大力气。

“给软件贴金”是个迷人的陷阱，程序员一不小心就会掉进去。预想开发是一个不易改变的习惯：当你正在兴头上时，真的很难用客观的眼光来看待自己刚想出的绝妙想法——它究竟是会让代码变得更好，还是仅仅增加复杂度？说起来，这正是结对编程的好处之一：旁边有个人能给你的妙主意提点客观的意见，这是价值无限的，因为程序员们总是很难客观地看待自己的想法，特别是在这个想法刚冒出来的时候。

要是程序员沉溺于预想开发（不管是哪种形式），软件的健康一定会大受损害。在最糟糕的情况下，预想开发会造就框架！框架并非十恶不赦之物，但它们经常是预想开发病的外在症状——这种病在Java世界里尤为严重：哪怕你把其他语言的所有框架加在一起，也比不上Java的框架多。Java世界里甚至还有元框架，也就是让人们能够轻松炮制出又一个框架的框架。该停止这种疯狂的举动了！

当框架完全出自预想时，它们就一无是处。Java世界里有几个经典的例子：EJB（第1版和第2版）和JSF。它们都是严重过度工程（over-engineering）的产物，用它们干点活真是难上加难。EJB几乎已经成了一个过度工程的传奇，因为它是如此复杂，只为了解决如此稀少的项目才面对的问题。而当时Java世界的普遍观点却鼓励使用EJB。JSF的情况略有些不同，不过同样具有代表性。JSF的特性之一就是可以定制渲染管道（rendering pipeline），你不仅可以用它生成HTML，还可以生成WML（Wireless Markup Language），甚至是原生XML。迄今为止我还没见过任何一个程序员真的用到这项特性，但所有JSF用户都得为它的存在而交复杂度税。这是以下情形的一个典型例子：象牙塔里的设计师们拍脑袋想出些很酷的东西，然后把它们放进框架里——更可恶的是这些东西让程序员们听起来也觉得很酷，于是市场营销就很容易了。但归根到底，一项用不到的特性只会给软件增加熵。

提示：

如无必要，勿增复杂度。

框架并非十恶不赦之物。恰好相反，框架已成为最受喜爱的一种抽象方式。面向对象开发和组件潮流中许诺的代码重用大多是靠框架来兑现的。但要是框架的功能远超出你的需要，它就会对项目造成伤害，因为框架必然增加复杂度。最好的框架不是出自那些坐在象牙塔里揣测程序员需求的设计师，而是从真实的代码中抽取出来的。人们开发了真实可用的应用程序，然后到开发下一个应用程序时，程序员们到前一个应用程序中学习经验，把好的东西抽取出来用到下一个程序里。这正是Ruby on Rails简洁的原因之一：它是从真实工作的代码中抽取出来的。

YAGNI并不是叫你绝不使用框架，只是老实承认框架不是银弹。你应该仔细考察框架提供的东西，如果它很大程度上覆盖了你的需求，那么当然值得去用它，这样你就不必自己去写那些代码了；但要是有人拿来一个EJB这样的框架，那还是小心点好。

不要说出来！

我们有一个需要用.NET开发的小项目。作为技术带头人，我不想用上nHibernate或是iBatis.net之类全套的O/R映射，因为项目的规模看起来没有那么大。但.NET原生的数据库接口库又太难用，因为它们难以单元测试是出了名的。我告诉项目经理，我要在ADO.NET基础上构造一个非常小的框架，就提供我们需要的那些功能，同时又是可测试的。当时项目经理都快要发飙了。“框架这种东西，提都别跟我提！”他冲我吼道。原来他以前的一个项目就被框架所困而没能按时交付，他不想重蹈覆辙。不过在我的坚持之下（技术带头人的角色也起了作用），他终于还是被我说服了。

当然，后来事实证明他是对的。我开发了这个小框架（没再跟项目经理提过），随之项目也启动了。一开始框架用起来还挺好：我们需要的东西很快就能开发出来，而且还能够测试。紧跟着宿命就降临了：我们要开发的一个东西在框架中支持得不太好，于是我往框架里加了些功能，然后一次又一次。没过多久，我就得把一半的时间都用来维护这个框架，项目经理则早已怒不可遏了。

最终我们还是按时完成了项目。我的魔幻小框架一开始节省的那些时间最后都被向其中添加功能的维护工作吞了回去——连本带利。一开始我以为已经清楚我们需要些什么，但软件项目中的细微之处根本就无法一一预知。我对项目经理道了歉。要是给我一个从头再来的机会，我会选择已有的框架，例

如nHibernate或者iBatis，因为我们所做的一切无非是重新实现这些框架一开始就提供的功能中的一小部分 —— 而且bug成堆。

YAGNI的另一个缘由是对功能的贪婪 —— 这会给商业软件造成严重的伤害。具体的情况大概是这样：

市场部："我们需要X，这样才能超过Y功能 —— 那是竞争对手Z的卖点。"
工程师："嘿，用户真的会在乎X还是Y吗？"
市场部："他们当然在乎。你得相信我，要不我怎么在市场部工作呢？"
工程师："好吧。"

这是个难题，因为市场人员认为他们知道什么是最好的 —— 也许他们真的知道。但他们并不完全理解增加软件的复杂度会带来什么影响，也不明白为什么功能A比功能B需要的时间长出一个数量级 —— 在非程序员看来这两个功能根本就一模一样。

保持业务用户与开发者之间的交流渠道畅通非常重要。作为软件开发者，我们应该主动给客户提出建议，帮助客户找到他们真正的所需。大部分时候，用户和业务分析师对于软件功能的工作方式有自己的想法，我们应该尝试抓住这些功能的核心价值，然后看看有没有更简单的解决方案。我曾经在项目中建议过一些非常好的替代方案，它们能满足与原来的方案相同的业务需求，实现起来却容易得多。沟通非常重要：如果没有良好的沟通渠道，你就会发现贪心的用户与沉默的程序员之间的矛盾实在难以调和。别忘了大船瓦沙的教训（请看接下来的故事）。

提示：

软件开发首先是一场沟通博弈。

大船瓦沙

1625年，瑞典国王古斯塔夫二世阿道夫打算建造有史以来最棒的战舰。他雇了最好的造船工匠，专门种植了一片最刚硬的橡树林，都为了建造大船瓦沙。国王不断提出各种要求，希望把船建得更宏伟壮丽，到处都加上华美的装饰。有那么一天，他甚至决定在船上配备两套甲板炮，这是当时世界上绝无仅有的。他的船会是海上的霸王。而且由于迫在眉睫的外交问题，他还希望大船瓦沙能尽快造好。当然了，工匠们一开始的设计只有一套甲板炮，不过既然国王提出了要求，自然就得再加一套。由于工期太短，工匠们来不及做"摇

晃测试”——让一群水手从船的一边快跑到另一边，船在这种情况下不应该摇晃得太厉害（换句话说，没有头重脚轻）。结果，它的处女航只有几个小时，然后就沉入了海底：在给它加上所有华丽特性的同时，国王和工匠也让这条船变得根本无法航行。大船瓦沙在北海的海底长眠了几百年，直到20世纪初才被打捞出来，陈列在博物馆里。

一个有趣的问题：到底是谁导致了瓦沙的沉没？是国王，因为他要求了太多的特性？还是工匠，因为他们只顾着满足国王的要求，没有大声说出自己的担忧？看看你正身处其中的项目吧：你是不是正在建造又一艘瓦沙？

应该集中关注给软件增加功能而非复杂度。假如有两个代码库，它们实现同样的功能。其中一个严格遵守简单原则，每一步都贯彻YAGNI的思想；另一个则加入了很多当前并不需要、只是可能在将来会用到的特性——这些特性不会马上创造价值，但从可重构性的角度来说你却马上就得为它们付出成本，从而降低在项目中作出修改的速度。用不到的特性也会增加代码维护和扩展的难度。代码量事关紧要，把没用的代码从代码库中剔除，在修改功能代码时就不会那么费力。如果特性可以称重，特性与可重构性之间的关系大概会如图9-1。

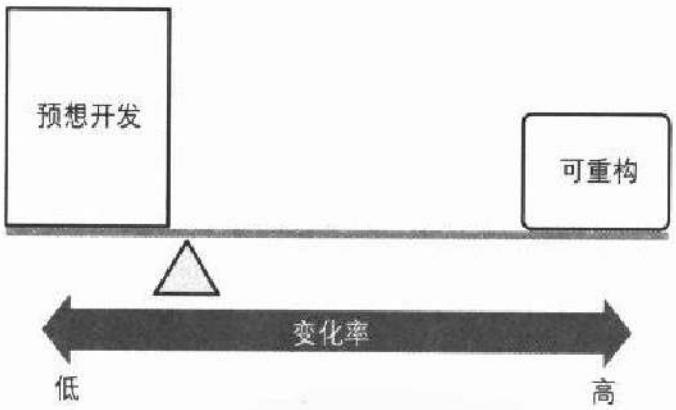


图9-1：如果特性可以称重，哪一种开发方式得到的代码库更容易修改，是预想开发还是YAGNI

只做当下需要的。一开始这会很难，但最终你会得到一个更好的代码库。如果不引入不必要的复杂度，在修改或重构时就不用对抗这些复杂度。程序员应该谨记：熵会杀死软件，所以，如无必要，勿增特性。

第 10 章

古代哲人

在一本关于程序员生产率的书籍中看到讲述古代哲人的章节或许会让你感到突兀，但这是真的。结果表明，由古老（或者不是那么古老）的哲学家们发现的一些哲学思想对构建高质量软件有直接的影响。我们看看这几个哲学家是怎么评说代码的吧。

亚里斯多德的“事物的本质性质和附属性质”理论

亚里斯多德建立了很多我们今天熟知的科学分支，事实上，多数科学研究都起源于他。他把自己对自然世界的所有思考进行分类、编目，并给出定义。同时，他也是逻辑和形式思维的奠基者。

亚里斯多德定义的一个逻辑原理是：事物本质性质和附属性质之间的区别。比方说，有五个单身汉，他们都有棕色的眼睛。未婚是他们的本质性质，而棕色眼睛是一个附属性质。因为你不能由此推断说：所有的单身汉都有棕色的眼睛，因为眼睛的颜色确实只是一个巧合。

好吧，但这跟软件有什么关联？稍微延伸一下这个概念，便会让我们想到事物的本质复杂性和附属复杂性。本质复杂性是指要被解决之问题的核心，由软件中的难点问题组成。大多数软件的问题都包含一些复杂性。附属复杂性是指跟解决方案没有必要直接关联的那些东西，但无论如何我们仍然要解决它们。

举个例子。譬如说有一个问题，它的本质复杂性在于对客户数据的跟踪：从网页获取数据，并把它们保存到数据库。这是一个很简单很直观的问题，但是要让它在你的组织内工作，你必须使用一个由低劣驱动所支持的“古老”数据库。当然，你还要担心数据库的访问权限问题。如果在其他某处主机上的一些数据库包含相似的数据，还必须要用数据交叉检查以保证一致性。现在，就是要想办法连接到主机，然后以一种你可使用的格式把数据提取出来。这时，你却发现你无法直接连接到数据库，因为在你使用的工具中竟然没有连接器。没办法，你只能让其他人为你提取数据，并把它们存放到某个数据仓库，然后你可以从那里获取。这听起来像不像你正在干的工作？本质复杂性往往可以一言以蔽之，而附属复杂性描述起来却常常没完没了。

没人希望在附属复杂性上花费比在本质复杂性上还多的时间，但随着它的累积，很多组织最终在附属复杂性上要花费比本质复杂性更多的时间。SOA（面向服务的架构）之所以在当前这么流行，就是因为很多公司试图从日积月累的大量附属复杂性中抽身而出。SOA是一个把迥然相异的两个应用程序绑定在一起，使之能够相互通信的架构风格。很少人会认为这是驱使你使用SOA的原因。然而，这的确就是在你有很多需要分享信息却

不可通信的程序时想做的事情。但在我看来，这纯粹是平添附属复杂性的行为。在厂商嘴里，SOA架构风格就等于企业服务总线（ESB），其主要卖点是：中间件问题的解决方案是使之更加中间化。难道增加复杂性会降低复杂性么？几乎不可能。所以，要对厂商驱动的解决方案慎之又慎。他们的首要目的是销售他们的产品，其次才（或许）是让你的生活更加美好。

识别问题是摆脱附属复杂性的第一步。思考一下你所使用的过程、策略以及正在处理的技术难题。认识清楚怎样的改进可能从根本上让你抛弃一些对整个问题贡献不多却增加麻烦的东西。就像刚才那个问题，你可能认为你需要的是一个数据仓库，但其实它所能带来的好处远比其增加的复杂性要少。你不可能把软件中的附属复杂性统统去掉，但是你可以致力于不断减少它们。

提示：

致力本质复杂性，去除附属复杂性。

奥卡姆剃刀原理

奥卡姆（译注1）的威廉爵士是一个厌恶华美装饰以及复杂解释的修士。他对哲学和科学的贡献是奥卡姆剃刀原理：如果对于一个现象有好几种解释，那么最简单的解释往往是最正确的。显然，这跟我们讨论的事物本质和附属性质理论紧密关联。这个原理对于软件的影响度也是出乎我们意料的。

作为软件行业中的一员，过去十年我们一直在进行着某项实验。这个实验始于20世纪90年代中期，主要是由于开发人员发现其开发进度远远跟不上软件需求的增长而引发的（其实在那时这已经不是一个新问题，这个问题自商业软件的想法出现之后就一直存在）。实验的目的是：创造一些工具和环境来提高那些普通开发人员的生产率，即使一些人比如Fred Brooks（他撰写了《人月神话》）已经告诉我们软件开发中的一些混乱事实。此实验试图验证：我们是否可以创造一种能限制程序员破坏力的语言而使人摆脱麻烦；我们是否可以无需支付荒唐的大量金钱给那些令人生厌的软件技工（即使在那时候你可能还为找不到足够的软件技工而发愁），而同样生产出软件呢？这些思考让我们创造出了如dBase、PowerBuilder、Clipper和Access这样的工具，并促成了工具和语言相结合的4GL（第四代语言）的崛起，比如FoxPro和Access。

但问题是，即使有这样的工具和环境你也不能完成所有的工作。我同事Terry Dietzler

译注1：奥卡姆（Ockham）在英格兰的萨里郡，是威廉出生的地方。

为Access创建了一个叫做“80-10-10”的准则（而我喜欢把它称之为Dietzler定律）。这个定律说的是：80%的客户需求可以很快完成；下一个10%需要花很大的努力才能完成；而最后的10%却几乎是不可能完成的，因为你不能把所有的工具和框架都“招至麾下”。而用户却希望能满足一切需求，所以作为通用目的语言的4GL（Visual BASIC、Java、Delphi以及C#）应运而生。Java和C#的出现主要是由于C++的复杂性和易错性，语言开发者们为了让一般程序员摆脱这些麻烦而在其内建了一些相当严格的限制。在此之后“80-10-10准则”才发生了改变，无法完成的工作已经微乎其微。这些语言都是通用目的语言，只要付出足够的努力，大多数工作都可以完成。但Java虽然比较易用却常常需要大量编码，所以框架出现了，组件增加了，大量其他框架蜂拥而至。

下面有一个例子。这段Java代码是从一个广泛使用的开源框架中提取出来的，试着明了它的功能吧（关于该方法的名字我只会提示你一点点）：

```
public static boolean xxXxxxx(String str) {
      int strLen;
      if (str == null || (strLen = str.length()) == 0) {
          return true;
      }
      for (int i = 0; i < strLen; i++) {
          if ((Character.isWhitespace(str.charAt(i)) == false)) {
              return false;
          }
      }
      return true;
  }
```

花了多少时间？这实际上是一个从Jakarta Commons框架（它提供了一些或许本该内置于Java的帮助类和方法）中提取出来的`isBlank`方法。一个字符串是否为“空白”由两个条件决定：这个字符串是空字符串，或者它只由空格组成。这段代码的计算公式非常复杂，因为要考虑参数是null的情况，而且还要迭代所有的字符。当然，你还要把字符包装成`Character`类型以确定它是否空白字符（空格、制表符、换行符等）。总之，太麻烦了！

下面是用Ruby实现的版本：

```
class String
  def blank?
    empty? || strip.empty?
  end
end
```

这个定义跟之前的那个非常相近。在Ruby里，你可以把`String`类打开并且加入一些新方法。这个`blank?`方法（Ruby里返回布尔值的方法通常用问号结尾）会检查字符

串是否为空，或者在去除所有空格之后是否为空。在Ruby里方法的最后一行就是返回值，所以你可以忽略可选的return关键字。

这段代码在一些预想不到的情况下同样工作。看看这段代码的单元测试吧：

```
class BlankTest < Test::Unit::TestCase
  def test_blank
    assert "".blank?
    assert " ".blank?
    assert nil.to_s.blank?❶
    assert ! "x".blank?
  end
end
```

❶ 在Ruby里，nil是NilClass类的一个对象，就是说它也有to_s方法（等价于Java和C#中的toString方法）。

其实我讲这些的主旨是：一些在企业级开发中非常流行的主流静态类型语言有很多附属复杂性。Java中的基本数据类型就是一个语言附属复杂性的绝佳例子。当Java还是新语言的时候它们确实非常有用，但如今它们只是让代码更加难以看懂而已。自动装箱（译注2）能起到一些帮助，但是它会引起其他一些异常问题。看看下面这段代码吧，肯定会让你挠头：

```
public void test_Compiler_is_sane_with_lists() {
    ArrayList<String> list = new ArrayList<String>();
    list.add("one");
    list.add("two");
    list.add("three");
    list.remove(0);
    assertEquals(2, list.size());
}
```

这个测试通过了。再看看下面这个版本，它们之间的区别只有一个单词（ArrayList换成了Collection）：

```
public void test_Compiler_is_broken_with_collections() {
    Collection<String> list = new ArrayList<String>();
    list.add("one");
    list.add("two");
    list.add("three");
    list.remove(0);
    assertEquals(2, list.size());
}
```

译注2：自动装箱（Autoboxing）是指将基本的数据类型自动地转换为封装类型。

这个测试失败了，抱怨说list的大小还是3。它说明了什么问题？它很好地解释了当使用泛型和自动装箱来改装一些复杂的库（比如集合）时会发生什么事情。测试失败的原因在于：Collection接口虽然也有remove方法，但是它删除的是与其参数内容匹配的项，而不是索引指定的项。在这个例子中，Java把整数0自动装箱成一个Integer类的对象，然后在列表中寻找一个内容是0的项，当然最后没有找到，所以也就不删除任何东西。

现代语言非但没有让程序员摆脱麻烦，而且其附属复杂性还迫使他们艰难地寻找复杂的解决方法。这个趋势影响了构建复杂软件的效率。我们真正想要的，是4GL作为强大的通用目的语言所具有的通用性和灵活性所带来的高效率。让我们看看用DSL（领域特定语言）构建的框架，目前一个很好的范例是Ruby on Rails。当编写Rails程序时，你不用写太多“纯”Ruby代码（即使写，大部分也只在处理商业逻辑的模型层）。你主要是在Rails的DSL部分编写代码，这意味着编码工作都投入在最有价值的部分，当然就带来了更大的产出。

```
validates_presence_of :name, :sales_description, :logo_image_url
validates_numericality_of :account_balance
validates_uniqueness_of :name
validates_format_of :logo_image_url,
                    :with => %r{\.(gif|jpg|png)}i,
                    :message => "must be a URL for a GIF, JPG, or PNG image"
```

只要小小的一段代码，却实现了大量的功能。DSL构建的框架提供了4GL级别的生产率，但它们有一个关键的差异。用4GL（目前一些主流的静态类型语言）做一些非常强大的东西（比如元编程）是极其困难或者说不可能的，而在一个基于一种超级强大语言的DSL里，你不仅可以用少量的代码完成大量的功能，而且可以跳到底层语言去实现任何你想做的东西。

“强大的语言加上特定领域元层次”提供了目前最好的解决方案。生产率来自DSL跟问题域的紧密相连，能力来自于表层之下的强大语言。基于强大语言并且易于表达的DSL将会成为一个新的标准。框架将会用DSL来编写，而不再是语法拘束且无必要规范过多的静态类型语言。请注意这并不代表只能使用动态语言（比如Ruby），只要有合适的语法，静态类型推断语言也非常有可能从这种设计风格中获益。例如Jaskell（注1），特别是基于它的DSL：Neptune（注2）。Neptune拥有和Ant一样的基本功能，但它是基于Jaskell的领域特定语言。Neptune展示了在一个熟悉的问题域之内，你可以用Jaskell写出多么易读以及简洁的代码。

注1：从 *http://jaskell.codehaus.org/* 下载。

注2：从 *http://jaskell.codehaus.org/Neptune* 下载。

提示：

Dietzler定律：即使是通用目的编程语言也逃不出“80-10-10准则”的魔咒。

笛米特法则

笛米特法则是20世纪80年代晚期在美国西北大学发展起来的。可以用一句话总结这个法则：只跟最亲密的朋友讲话。它的主要思想是：任何对象都不需要知道与之交互的那些对象的任何内部细节。这个法则的名字来自于古罗马掌管农业(也就是掌管食物分配)的女神笛米特。虽然严格地讲她不是一个古代哲学家，但她的名字听起来确实很有哲学家味道。

更正式地说，笛米特法则讲的是任何一个对象或者方法，它应该只能调用下列对象：

- 该对象本身
- 作为参数传进来的对象
- 在方法内创建的对象

在大多数现代语言中，你可以把它解释得更形象更简短：在调用方法时永远不要使用一个以上的“点”。下面就是一个例子。

有一个Person类，它有两个属性：name和Job。Job类同样有两个属性：title和salary。根据笛米特法则，在Person类里面通过调用Job得到其position属性是不被允许的，就如下面一样：

```
Job job = new Job("Safety Engineer", 50000.00);
Person homer = new Person("Homer", job);

homer.getJob().setPosition("Janitor");
```

那么，要遵循笛米特法则，你可以在Person类里面创建一个方法来改变job，然后让Job类去完成余下的工作，看下面：

```
public PersonDemo() {
    Job job = new Job("Safety Engineer", 50000.00);
    Person homer = new Person("Homer", job);
    homer.changeJobPositionTo("Janitor");
}

public void changeJobPositionTo(String newPosition) {
    job.changePositionTo(newPosition);
}
```

这样的改变带来了什么好处？首先请注意，我们不再调用Job类的setPosition方法，而是使用了一个更形象的名字：changePositionTo。这强调了一个事实：除了Job类本身，没有任何其他东西知道Job类内部的位置是如何实现的。虽然它现在看起来像是一个字符串，但在内部实现上可能使用的是一个枚举。这么做的主要目的是信息隐藏：你不想让依赖类知道被依赖类内部工作的实现细节。笛米特法则通过迫使你编写隐藏细节的方法来达到这个目的。

当严格遵守笛米特法则时，你会倾向于为你的类做很多小的包装或者为其编写大量的代理方法，以防止在调用方法时使用多个“点”。那段额外的代码让两个类之间的耦合更加松散，这样能保证一个类的改变不会对另一个类产生影响。如果你想看更详尽的例子，请阅读David Bock（注3）的文章"The Paperboy, The Wallet, and The Law Of Demeter"（也是软件学说的一部分）。

“古老的”软件学说

软件开发者们对“古老的”软件学说基本上一无所知。因为软件技术日新月异，开发者们为了保持与时代的同步已经需要付出很大的努力，而且那些（相对）古老的技术能帮助我们解决当前的问题么？

当然，看一本Smalltalk的语法书并不能帮助你提高Java或者C#水平，但Smalltalk书籍不仅仅讲述语法，比如它们可以让那些第一次使用全新技术（面向对象语言）的开发者们学到很多难得的知识。

> **提示：**
>
> 关注那些“古老的”软件技术学说。

古代哲学家们所创立的一些在现在看来显而易见的思想，在当时却需要非凡的才智和莫大的勇气。有时候，他们因为其思想违背了当时已建立的教条还要遭受极大的迫害。其中一个伟大的“历史背叛者”就是伽利略，他不相信任何人告诉他的任何事情，凡事他都要自己进行尝试。在他之前的时代，人们已经接受了“重物比轻物降落得快”的说法。这是亚里士多德学派的思想，他们认为逻辑思考比实验更有价值。伽利略却不买账，他登上比萨斜塔然后向下扔下石块，他还用大炮射击石头。最后他终于发现这个有悖直觉的事实：所有的物体都以相同的速度降落（如果不计空气阻力）。

注3：*从http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf下载。*

伽利略验证了：有些看起来不符合直觉的事情其实是正确的。同时，这也是他给后人上的非常有价值的一课。很多“实打实”的软件开发知识并不符合直觉。比如，“软件可以预先设计好并按部就班地去实现”的想法看起来是符合逻辑的，但在现实世界的持续变化面前，它并不成立。幸运的是，大量反直觉的软件学说已经编目在反模式目录（*http://c2.com/cgi/wiki?AntiPatternsCatalog*）中。这是软件的古老学说。当你的老板要求你使用一个低质量的代码库时，不需要在崩溃中咬牙切齿，告诉他：你正落入“站在侏儒的肩膀上”的陷阱中，然后他就会明白不仅仅只有你才觉得那是个坏主意。

理解已存在的软件学说，能给你提供很好的资源。比如当你被告诉甚至被一些管理者强迫去做一件你内心知道是错的事情时。理解过去发生的战争能为你当前的战争供给弹药。花一些时间去读读那些数十年之前就已面世但仍被广泛阅读的软件书籍吧，比如《人月神话》，Hunt和Thomas的《程序员修炼之道》（Addison-Wesley）以及Beck的《Smalltalk Best Practice Patterns》（Prentice Hall）。这远远不是一个详尽的列表，但这几本书都提供了无价的知识。

第 11 章

质疑权威

在开发团队和开发社区中推行通用标准是一件好事。它使人们能更容易阅读彼此的代码，更迅速地理解代码中的习语，并有效避免率性而为的随意编程（也许Perl社区（注1）例外）。

但是，盲目固守标准就跟完全没有标准一样糟糕。有时候标准反而限制了一些有价值的变通。对于软件开发中你所做的每一件事，你都要确保自己明白做这件事的原因。否则，你可能会因为愤怒的猴子们而倍受折磨。

愤怒的猴子

我从Dave Thomas的主题演讲“愤怒的猴子与船货崇拜（译注1）”中第一次听到这个故事。我不确定故事是否真实（尽管我做了一些调查），但是没有关系——它很好地阐明了一个观点。

早在20世纪60年代（那时候科学家们可以做任何疯狂的事情），行为科学家们进行了一项实验。他们把五只猴子和一架活梯放在一间屋子里，并在天花板上挂了一串香蕉。这些猴子很快就想到它们可以爬上梯子去吃香蕉，但每当它们靠近活梯的时候，科学家们就用冰水浸满整个屋子。我想你能猜到会发生什么：一群愤怒的猴子。很快，再没有一只猴子会去靠近那个梯子了。

之后，科学家们将其中一只猴子替换成另一只没有忍受过冰水折磨的新猴子。这只新猴子所做的第一件事就是直奔那架梯子，但当它这么做时其他所有猴子都痛打它。它不明白为什么，但很快就学乖了：不要去靠近那架梯子。科学家们逐渐将最初的那些猴子都替换成新猴子，直到这群猴子中谁都没有被冰水浸泡过，然而它们还是会去攻击任何靠近梯子的猴子。

这说明了什么？软件项目中许多惯例之所以存在，就因为“我们一直是那样做的”。换句话说，是因为愤怒的猴子。

这里有一个我以前项目中的例子。大家都知道在Java中我们使用骆驼命名法（CamelCase）来命名方法：以小写字母开头，其后的每个单词之间都以一个大写字母来分界。这对常规编码来说是适用的，但对测试名就不同了。在命名单元测试时，你想要的是一个即长又准确的描述性名字，这样你就可以知道测的是什么。不幸的是，我们用

译注1：美拉尼西亚盛行的一种类似于千禧年主义的社会运动，人们盼望某种超自然的事物给他们带来繁荣昌盛。

注1：我只是在开玩笑。真的。我爱你们！请不要给我发E-mail！

骆驼命名法得到了像LongCamelCaseNamesAreHardToRead这样难懂的名字。在那个项目中，我建议在测试名的每个单词之间加上下划线，就像下面这样：

```
public void testUpdateCacheAndVerifyItemExists() {
}

public void test_Update_cache_and_verify_item_exists() {
}
```

在我看来，采用下划线的名称更具可读性。而观察当时开发团队对我提议的反应，也是件非常有趣的事情。有些开发人员很快就喜欢上了这个主意，但其他人对我这个小小的建议表现得就像愤怒的猴子一样。我们最终采用了这种方式（有时候技术负责人会像一个仁慈的独裁者），并发现这样产生的测试名易读多了，特别是在IDE的测试运行视图中读那一列长长的名字时（请见图11-1）。

对于任何开发习惯，仅仅因为“我们一直是这样做的”而存在是站不住脚的。如果你明白自己为什么一直这样做，而且这样做确实有意义，那么一定要继续。但你应当始终保持对任何假设的质疑，并验证它们的正确性。

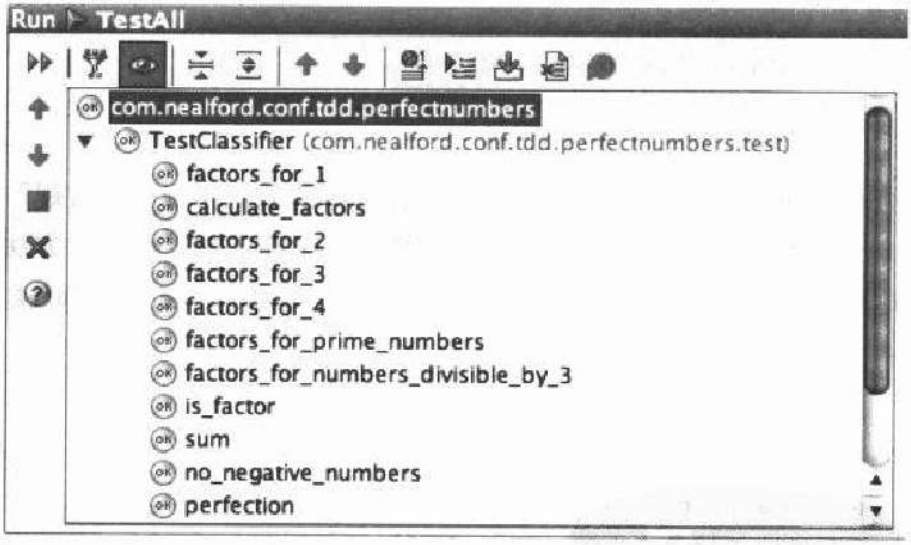


图11-1：采用下划线的测试名更具可读性

连贯接口

连贯接口（fluent interface）是现在正流行的领域特定语言（DSL）的风格之一。所谓连贯接口，就是将长长的一连串代码构建成句子，按照这种方式可以合理地将一个个完整的想法口语化。就像英语句子一样，这种风格的代码更容易阅读，因为这样你就知道一个想法在哪里结束，下一个从哪里开始。

这是根据我的一个项目给出的示例。在那个项目里，我们开发了一个处理列车车厢的应用系统，每种车厢都有一个营销说明。车厢有许多与之相关的规章制度，所以要建立适当的测试场景很难。我们得不停地问我们的业务分析师，对于需要测试的车厢种类，我们是否有绝对细致入微的定义。这是我们给他们所看代码的一个简化版本：

```
Car car = new CarImpl();
MarketingDescription desc = new MarketingDescriptionImpl();
desc.setType("Box");
desc.setSubType("Insulated");
desc.setAttribute("length", "50.5");
desc.setAttribute("ladder", "yes");
desc.setAttribute("lining type", "cork");
car.setDescription(desc);
```

阅读这样的代码对一个Java开发人员来说相当平常，但对我们的业务分析师来说却很困难。“为什么你们要给我看Java代码？只要告诉我你的意思就行了”。当然，“翻译”总会有带来出错的可能。为了解决这个问题，我们创建了一个连贯接口来传达同样的信息，但代之以这样的格式：

```
Car car = Car.describedAs()
              .box()
              .length(50.5)
              .type(Type.INSULATED)
              .includes(Equipment.LADDER)
              .lining(Lining.CORK);
```

我们的业务分析师更喜欢看到这样的代码。我们成功地去除了很多“常规”Java API风格中所需的令人反感的冗余。这实现起来非常简单。让所有的属性设置方法返回this而不是void，这样我们就可以将方法调用链接起来构造成句子。现在，Car类的实现就像下面这样了：

```
public class Car {
      private MarketingDescription _desc;

      public Car() {
          _desc = new MarketingDescriptionImpl();
      }

      public static Car describedAs() {
          return new Car();
      }

      public Car box() {
          _desc.setType("box");
          return this;
      }

      public Car length(double length) {
```

```
        _desc.setLength(length);
        return this;
    }

    public Car type(Type type) {
        _desc.setType(type);
        return this;
    }

    Public Car includes(Equipment equip) {
        _desc.setAttribute("equipment", equip.toString());
        return this;
    }

    public Car lining(Lining lining) {
        _desc.setLining(lining);
        return this;
    }
}
```

这也是DSL模式中的表达式构建器（Expression Builder）模式的一个例子。Car类隐藏了一个事实：它实际上在内部创建了一个MarketingDescription对象。表达式构建器通过在封装好的表达式之上创建公共接口来简化连贯接口。为了使方法链成为可能，Car的每个修改器方法都得返回this。

但是为什么把这个例子放到“质疑权威”这一章呢？要写出像Car这样的连贯接口其实就是要求你毁掉Java中的圣牛（译注2）：Car类将不再是一个JavaBean。尽管这看起来不算什么，但多数的Java程序结构中都坚持采用这一规范。但如果你仔细研究一下JavaBean规范，会发现它的有些做法对整体代码质量非常有害。

JavaBean规范强调每个对象都要有一个默认构造函数（请见第8章的“构造函数”一节），但实际上如果没有状态，没有哪个对象是有效的。JavaBean规范还强制使用Java中丑陋的属性存取语法，要求以getXXX()方法作为访问器（accessor），以返回void类型的setXXX()方法作为修改器（mutator）。我理解为什么会有这些限制存在（比如，默认构造函数会使序列化更容易），但在Java世界中没有人质疑过他们是否应该将自己的对象实现成bean。他们会乖乖地照着其他愤怒的猴子一样，将每个对象都实现成一个bean。

所以，要质疑权威。要是把每个对象都实现成bean，创建一个连贯接口就不可能了。知道你在创建什么，明白它将被用来干什么，然后明智地作出决策。“因为大家都说它应该是这样的”不是充分的理由。

译注2：“圣牛”一词比喻不可批评、侵犯的人或物，有嘲讽含义。

反目标

有时候，你应当质疑的权威是你自己对某个问题特定解决方案的偏好。在2006年的OOPSLA会议上有一篇很棒的论文，叫做“协同扩散：反目标编程”（注2）。这篇论文的作者指出：尽管对象和对象层次结构为多数问题提供了优秀的抽象机制，而同样的这些抽象却会使某些问题更加复杂。反目标（Anti-Objects）背后的思想就是对问题显而易见的原因和背后深层次的原因进行转换，然后去解决更简单，但却不是那么明显的问题。那这里的“显而易见的原因”和“背后深层次的原因”是什么意思呢？下面举例来说明。（警告！如果你仍然很喜欢玩《PacMan》，就不要阅读下面几段——它们会永久地毁掉PacMan在你心中的地位！有时候知识也伴随着代价。）

《PacMan》控制台游戏在20世纪70年代诞生时，它的计算能力比今天的一个廉价手机还弱。然而，它需要解决一个相当有难度的数学问题：如何让那些鬼魂穿过迷宫来追赶PacMan？那也就是说：穿过迷宫到一个移动目标的最短距离是多少？这是一个很大的问题，特别是如果你使用的内存很小或计算能力很弱。所以，《PacMan》的开发人员没有去解决这个问题，他们使用反目标的方法，将智能内建于迷宫自身。

《PacMan》中的迷宫就像一个自动机（很像康威的《生命游戏》）。迷宫的每个房间都有与之相关的简单规则，从左上端开始直到右下端，这些房间每次都依次执行一遍。每个房间会记住一个“PacMan气味”值。当PacMan位于一个房间中，这个房间就会有最大的“PacMan气味”值。如果他刚离开这个房间，该房间的“PacMan气味值”就等于最大值减1。气味随着轮次的进行而递减，直至消失。这时鬼魂们就变得愚钝了：他们只能不断地寻找PacMan气味，每当他们嗅到气味时，他们就会朝着气味最重的房间去。

这个问题“显而易见”的解决方案是给鬼魂们赋予智能。然而，更简单的解决方法是给迷宫赋予智能。这就是反目标方法：反转计算的前因和后因。不要掉入“传统的建模方法总是正确的”陷阱中。也许一个特定的问题完全可以用另一种语言很容易就解决。（请参见第14章了解更多反目标方法背后的原理）。

注2：　下载地址：*http://www.cs.colorado.edu/~ralex/papers/PDF/OOPSLA06antiobjects.pdf*。

第 12 章

元编程

元编程的正式定义是编写“会写程序”的程序，但是其应用范围实际上要广泛得多。通常来说，任何以超出“正常”的方式操作代码的做法都视为元编程。元编程方法比传统的解决方案（例如代码库和构架）更复杂，但是因为你在一个更基础的层面上操作代码，它使得困难的事情变得简单、不可能的事情变得可能。

所有的主流语言都在一定程度上支持元编程。学习元编程技巧将会使你事半功倍，让你打开一条找到解决方案的新路。

在这一章，我会谈到几个元编程的例子，分别以Java、Groovy和Ruby为编程语言，以方便不同的读者理解。每种语言的特性是不同的，下面的例子简单地展示了元编程能解决的几种问题。

Java和反射

Java的反射功能很健壮，同时也很局限。你当然能通过字符串形式的方法名来调用方法，但是安全管理器不会允许你在运行时定义新方法或者覆写已有的方法。面向切面的编程工具（例如AspectJ）能帮你一些忙，但是这种用法很具争议：人们质疑其是否是真正的Java，因为它有自己的语法和编译器等基础设施。

可能需要用Java反射机制的一个例子是测试私有方法。当你使用测试驱动开发方法来写代码的时候，你仍然想要使用语言自身的保护机制。通过反射来调用私有方法是非常顺理成章的，但是需要很多语句来实现。

比方说你需要调用Classifier类的isFactor方法（该方法用于判断一个数是否是另一个数的因子）。想要调用私有方法isFactor，你可以在测试类中创建一个如下所示的辅助方法：

```
private boolean isFactor(int factor, int number) {
    Method m;
    try {
        m = Classifier.class.getDeclaredMethod("isFactor",
                int.class);
        m.setAccessible(true);❶
        return (Boolean) m.invoke(new Classifier(number), factor);❷
    } catch (Throwable t) {
        fail();❸
    }
    return false;
}
```

❶ setAccessible的调用把方法的调用权限改成了public。

❷ invoke方法实现了真正的调用，并将返回结果包装成原方法要求的返回值类型（在这个例子里，自动包装成了基本的boolean类型）。

❸ 不管有什么异常抛出，这个单元测试都会失败，因为肯定有什么东西出错了。

这样单元测试就很容易了：

```
@Test public void is_factor() {
    assertTrue(isFactor(1, 10));
    assertTrue(isFactor(5, 25));
    assertFalse(isFactor(6, 25));
}
```

在前面的例子里，所有由于反射代码发生的异常都被隐藏了（Java很担心反射的使用，要求你捕捉许多不同类型的异常）。大多数时候，你的测试失败是因为方法的实现有问题。但有些时候，你必须小心处理异常。下面是另一个测试辅助方法的例子，在这个例子里我们必须自己来处理反射时产生的异常，并合理地向上层抛出。

```
private void calculateFactors(Classifier c) {
    Method m;
    try {
        m = Classifier.class.getDeclaredMethod("calculateFactors");
        m.setAccessible(true);
        m.invoke(c);
    } catch (InvocationTargetException t) {
        if (t.getTargetException() instanceof InvalidNumberException)
            throw (InvalidNumberException) t.getTargetException();
        else
            fail();
    } catch (Throwable e) {
        fail();
    }
}
```

在上面的例子里，你需要考察是否关注异常。如果需要关注某些异常，就要把它重新抛出，同时把其他的异常屏蔽掉。

通过反射来调用方法的能力使得构建更智能的工厂类成为可能，你可以在运行时加载类。大多数的插件架构都使用了通过反射动态加载类和调用方法的能力，以允许用户基于特定的接口构造新的东西，而无须在编译时就有具体的类实现。

Java的反射（以及其他的元编程）特性与动态语言相比还比较弱。C#要稍微好一些，提供了更详尽的元编程支持，但是总体说来与Java也在同一档次上。

用 Groovy 测试 Java

Groovy 是针对 Java 的动态语言语法。它可以与 Java 代码无缝交互（包括编译好的字节码），允许你使用更灵活的语法。Groovy 使得用户可以实现一些对 Java 而言非常困难或者几乎不可能的功能。

下面的 Groovy 语法使用了标准的 Java 反射机制，可以用来重新实现前面对 isFactor 方法的测试。

```
@Test public void is_factor_via_reflection() {
  def m = Classifier.class.getDeclaredMethod("isFactor", int.class)
  m.accessible = true
  assertTrue m.invoke(new Classifier(10), 10)
  assertTrue m.invoke(new Classifier(25), 5)
  assertFalse m.invoke(new Classifier(25), 6)
}
```

可以看到，反射代码非常简洁，甚至我都没有感觉到把它放到它自己的方法里有什么困难。Groovy 已经完成了烦人的异常检查，使调用反射方法更加简单（至少没有那么拘谨了）。Groovy "懂得" Jave 所有的语法，因此 `m.accessible = true` 这句就相当于 Java 中调用 `m.setAccessible(true)`。Groovy 也放松了使用括号的规则。

这段 Groovy 代码与前面用 Java 编写的单元测试功能一样 —— 它完全使用了同样的 JAR 文件来访问代码。Groovy 使得为 Java 代码编写单元测试更加简单，这是一个很好的借口来把它"偷渡"到保守的组织中（尽管测试代码是项目基础的一部分，但它毕竟不会被部署到生产环境中，所以谁会在意你使用了什么开源的代码库呢）。事实上，如果有非开发人员在场的时候，我推荐不要直接称 Groovy 的名字。我更喜欢称之为企业业务执行语言（缩写为 ebXl —— 管理者们都认为含有 X 的缩写比较酷）。

实际上，前面的测试并不是故事的全部。以它目前的实现，Groovy 完全忽略了关键字 private，甚至是在 Java 代码中声明的 private。因此，先前的测试也可以这样写：

```
@Test public void is_factor() {
    assertTrue new Classifier(10).isFactor(1)
    assertTrue new Classifier(25).isFactor(5)
    assertFalse new Classifier(25).isFactor(6)
}
```

是的，这是调用 JAR 文件中的 private 方法 isFactor 的 Java 代码。Groovy 恰当地忽略了 private 声明，这样你就可以直接调用该方法了。无论如何，Groovy 还是使用了反射来调用方法的，它只是悄悄地替你调用了 setAccessible。从技术上讲，这是 Groovy 的一个缺陷（它从 Groovy 诞生的那天起就存在），但是这一点又是如此有用，没有人觉得

有必要去修复它——我希望他们永远都不要去修复这个缺陷。在任何一门具有很强反射能力的语言中，关键字private都仅仅是文字形式上的：只要有需要，我总是可以使用反射来调用它所修饰的方法。

编写连贯接口

这里有一个简短的例子，它使用了Ruby的元编程特性来编写连贯接口——这是Java力所不及的。关于这个主题能写下一整本书，但这里只是为你提供一个初体验。下面的代码可在任何版本的Ruby下运行，也包括Java平台的JRuby。

你想要编写一个表示食谱的连贯接口。它应该允许开发者编写食谱的原料成分，这看起来像是一个数据格式；但在背后，它还要统计食谱的营养构成。封装代码的实际行为是连贯接口的优点之一。下面是示例语法：

```
recipe = Recipe.new "Spicy bread"
recipe.add 200.grams.of Flour
recipe.add 1.lb.of Nutmeg
```

要让上面的代码工作，你首先必须为Ruby中内建的Numeric类（该类封装了整数和浮点数）添加新的方法。

```
class Numeric❶
  def gram
    self
  end
  alias_method :grams, :gram❷
  def pound
    self * 453.59237
  end
  alias_method :pounds,:pound
  alias_method :lb,:pound
  alias_method :lbs, :pound
end
```

❶ 类Numeric总是在classpath中，所以这个方法打开了这个类，以便于往其中添加新的方法。

❷ alias_method是Ruby内建的功能，允许你为已经存在的方法创建别名（换句话说就是友好的名字）。alias_method不是关键字，它是Ruby内建的元编程功能的一部分。

Ruby的开放类允许你向已存在的类中添加新方法。所需的语法非常简单：当你创建了一个类的定义时，如果它已经在classpath中存在，那你就打开了一个已经存在的类。内建

的Numeric类型很明显已经存在于classpath中了，所以这段代码往Numeric类中添加了新的方法。

对Numeric类的改变使食谱连贯接口的第一部分能运行了。那么第二部分呢？

```
class Numeric
  def of ingredient
    if ingredient.kind_of? String
      ingredient = Ingredient.new(ingredient)
    end
    ingredient.quantity = self
    return ingredient
  end
end
```

我们再次打开Numeric类，给数字添加of方法。这个方法接受String或者Ingredient类型的参数（代码会检查传入的参数类型），把数字值赋值给Ingredient对象的quantity属性，然后返回Ingredient实例（return并不是必须的：Ruby方法的最后一行就是方法的返回值，但是明确地写出return会稍微增加一点代码的可读性）。

下面的单元测试证实了所有的修改都正确地运行：

```
def test_full_recipe
  recipe = Recipe.new
  expected = [] << 2.lbs.of("Flour") << 1.gram.of("Nutmeg")
  expected.each {|i| recipe.add i}
  assert_equal 2, recipe.ingredients.size
  assert_equal("Flour", recipe.ingredients[0].name)
  assert_equal(2 * 453.59237, recipe.ingredients[0].quantity)
  assert_equal("Nutmeg", recipe.ingredients[1].name)
  assert_equal(1, recipe.ingredients[1].quantity)
end
```

用动态语言构建连贯接口要容易得多，因为它们具备开放类、数字量对象之类的特性。对Java和C#开发者来说，以打开类的方式来添加方法有点怪，但是它却是Ruby和Groovy中编写代码的正常方式。

> **提示：**
>
> 元编程改变了你的语法辞典，给你提供了更多表达自己的方式。

元编程的归处

看完这些元编程代码你可能会觉得恶心，因为它违反了编程的基本法则：不要编写可自我更改的代码，但是这恰恰是你应该质疑的地方（参见第11章）。是的，使用不当的话，

这种方式会很危险，但这是每个强大的特性都需要面对的现实。在Java中你也可以使用切面（aspects）做危险的事，只是更难用一些而已。可是如果说强大的语言特性应该难于使用以确保只有专家才能够掌握，那显然是错误的。Java将String对象声明为final的原意是阻止开发人员修改该对象。但是有趣的事情发生了：这种约束并没有把差的开发者变好，反而给优秀的开发者套上了枷锁，他们不得不忍受可笑的枷锁。现在这个经典的事例已经过时了，因为有了Groovy提供的GString，那些优秀的开发者现在关注的是如何充分利用GString——这是Groovy中提供的String，而且提供了比Java更多的特性。有些人也许会说，因为Groovy与Java是天生紧密交互的，所以用户也许可以交替使用String和GString，当然Groovy代码向Java代码传递的一定是String。但是事实上这是错误的。因为String被声明成final，你甚至不能将GString定义为String的子类，这样Java库也就没有办法解读GString了。final的存在代表了语言设计者的态度：他们不信任使用该语言的人们。

具有对元编程良好支持的语言则恰好相反：它们允许开发者使用额外的能力，由开发者来决定什么时候使用这些额外的能力。

第 13 章

组合方法和 SLAP

SLAP是单一抽象层次原则（Single Level of Abstraction Principle）的简写，这个概念来自Kent Beck的《Smalltalk Best Practice Patterns》(Prentice Hall)，我的朋友Glenn Vanderburg在理解透概念之后提出了这个缩写。

在讨论SLAP之前，我想有必要先谈谈Beck书中论及的组合方法（Composed Method）模式。组合方法要求所有的公有方法读起来像一系列执行步骤的概要，而这些步骤的真正实现细节是在私有方法里面。组合方法有助于让代码保持精炼并使之易于复用，下面我们看看它是怎么工作的。

组合方法实践

组合方法鼓励把代码构建（或重构）得更简短、精炼、可读性更高。在我担任技术带头人的项目里面，我们的经验就是，对于Java或C#，不允许方法超过15行，而对于像Groovy或Ruby这样的动态语言，最多允许5行代码。

这带来了什么好处？看看下面这段不遵循组合方法模式的代码，它来自一个小型的电子商务网站：

```
public void populate() throws Exception  {
    Connection c = null;
    try {
        Class.forName(DRIVER_CLASS);
        c = DriverManager.getConnection(DB_URL, USER, PASSWORD);
        Statement stmt = c.createStatement();
        ResultSet rs = stmt.executeQuery(SQL_SELECT_PARTS);
        while (rs.next()) {
          Part p = new Part();
          p.setName(rs.getString("name"));
          p.setBrand(rs.getString("brand"));
          p.setRetailPrice(rs.getDouble("retail_price"));
          partList.add(p);
      }
    } finally {
        c.close();
    }
}
```

这是一个使用底层JDBC库连接数据库、获取数据信息的类中的一个方法。尽管在这段代码里面找不到明显可以复用的东西，但是它不仅违反了“15行代码”约定，而且还承担了大量的职责。该对它重构了。

首先就是要把它分解成一个个工作步骤。把这个方法所做的事情写下来，你就知道怎么对各个新方法命名了。经过首次重构之后，得到如下代码：

```
public class PartDb {
    private static final String DRIVER_CLASS =
            "com.mysql.jdbc.Driver";
    private static final String DB_URL =
            "jdbc:mysql://localhost/orderentry";
    private static final int DEFAULT_INITIAL_LIST_SIZE = 40;
    private static final String SQL_SELECT_PARTS =
            "select name, brand, retail_price from parts";
    private static final Part[] TEMPLATE = new Part[0];
    private ArrayList partList;

    public PartDb() {
        partList = new ArrayList(DEFAULT_INITIAL_LIST_SIZE);
    }

    public Part[] getParts() {
        return (Part[]) partList.toArray(TEMPLATE);
    }

    public void populate() throws Exception {
        Connection c = null;
        try {
            c = getDatabaseConnection();
            ResultSet rs = createResultSet(c);
            while (rs.next())
                addPartToListFromResultSet(rs);
        } finally {
            c.close();
        }
    }

    private ResultSet createResultSet(Connection c)
            throws SQLException {
        return c.createStatement().
                executeQuery(SQL_SELECT_PARTS);
    }

    private Connection getDatabaseConnection()
            throws ClassNotFoundException, SQLException {
        Connection c;
        Class.forName(DRIVER_CLASS);
        c = DriverManager.getConnection(DB_URL,
                "webuser", "webpass");
        return c;
    }

    private void addPartToListFromResultSet(ResultSet rs)
            throws SQLException {
        Part p = new Part();
        p.setName(rs.getString("name"));
        p.setBrand(rs.getString("brand"));
        p.setRetailPrice(rs.getDouble("retail_price"));
        partList.add(p);
    }
}
```

不错，这好多了。你能看到重构出的方法读起来就像一系列步骤的纲要：

1. 获得数据库连接。
2. 通过数据库连接获取结果集。
3. 循环遍历结果集，把每一项添加到Part列表。
4. 关闭数据库连接。

populate方法现在已经遵循组合方法准则了，但别急，看看还能做什么。getDatabaseConnection方法跟结果列表没有任何关系：它只是用来获取数据库连接。你可以顺着继承体系把它上推到父类中去，这样其他需要访问数据库的子类也可以复用它。同样地，在createResultSet方法中我们只需要传入SQL语句，就可以使它成为通用方法。好了，改进完这两点，你可以得到两个类：BoundaryBase和PartDb：

```
abstract public class BoundaryBase {
    private static final String DRIVER_CLASS =
            "com.mysql.jdbc.Driver";
    private static final String DB_URL =
            "jdbc:mysql://localhost/orderentry";

    protected Connection getDatabaseConnection() throws ClassNotFoundException,
            SQLException {
        Connection c;
        Class.forName(DRIVER_CLASS);
        c = DriverManager.getConnection(DB_URL, "webuser", "webpass");
        return c;
    }
    // . . .
```

BoundaryBase类现在包括了getDatabaseConnection方法，以及与之相关的两个常量。

至于createResultSet方法，我们可以使用模板方法模式改进它，该模式详见“四人组(GoF)”的经典书籍《Design Patterns: Elements of Reusable Object-Oriented Software》(Addison-Wesley)(译注1)。模板方法模式是指由父类定义抽象方法，而把实现细节推迟到子类的方法中。这个模式提供了一种办法来让你先定义共同的算法结构，之后再提供实现细节。在这里我们可以这样实现：将createResultSet方法上推到BoundaryBase类，同时引入一个抽象方法来强制子类提供所需要的SQL。在这个例子中，createResultSet方法被分解成两个方法：一个保留原来的名字，另一个是让子类用来提供SQL的新的抽象方法(getSqlForEntity)：

译注1：本书由机械工业出版社出版。其影印版书号为7-111-09507-3，中文版书号为978-7-111-07575-2，双语版书号为978-7-111-21126-6（中文书名为《设计模式：可复用面向对象软件的基础》）。

```
abstract protected String getSqlForEntity();

protected ResultSet createResultSet(Connection c) throws SQLException {
    Statement stmt = c.createStatement();
    return stmt.executeQuery(getSqlForEntity());
}
```

好极了，让我们看看在populate方法中还可以抽象出哪些东西。稍微研究一下，就能发现populate方法只有在while循环里依赖于特定的实体类：从结果集中取出每一项再填充到实体列表。这里你同样可以使用模板方法，把populate方法移到BoundaryBase类里面去：

```
abstract protected void addEntityToListFromResultSet(ResultSet rs)
        throws SQLException;

public void populate() throws Exception {
    Connection c = null;
    try {
        c = getDatabaseConnection();
        ResultSet rs = createResultSet(c);
        while (rs.next())
            addEntityToListFromResultSet(rs);
    } finally {
        c.close();
    }
}
```

跟之前一样，从结果集里面获取数据再填充到实体列表，是很多业务实体类都会用到的共同算法。既然如此，为什么不让它更通用一些呢？

现在，看看完成所有的重构之后的PartDb类：

```
public class PartDb extends BoundaryBase {
    private static final int DEFAULT_INITIAL_LIST_SIZE = 40;
    private static final String SQL_SELECT_PARTS =
            "select name, brand, retail_price from parts";
    private static final Part[] TEMPLATE = new Part[0];
    private ArrayList partList;

    public PartDb() {
        partList = new ArrayList(DEFAULT_INITIAL_LIST_SIZE);
    }

    public Part[] getParts() {
        return (Part[]) partList.toArray(TEMPLATE);
    }

    protected String getSqlForEntity() {
        return SQL_SELECT_PARTS;
    }
```

```
        protected void addEntityToListFromResultSet(ResultSet rs) throws SQLException {
            Part p = new Part();
            p.setName(rs.getString("name"));
            p.setBrand(rs.getString("brand"));
            p.setRetailPrice(rs.getDouble("retail_price"));
            partList.add(p);
        }
    }
```

类里面留下的都是跟 Part 实体紧密相关的方法、变量等。其他所有根据结果集填充实体的处理代码都放到了 BoundaryBase 类，可以被其他的实体复用。

这个例子给了我们三点启示。第一，最原始的代码看上去没有任何可复用的部分，只是一大堆冗长无趣的代码。一旦你应用了组合方法模式，可以复用的部分就会浮现出来。你很难从冗繁的代码中看出可复用的部分，但在强迫自己把它分解成原子片段之后，可复用的代码（兴许之前你都没觉得）就暴露了出来。

提示：

重构成组合方法能暴露出隐藏的可复用代码。

第二，现在你清晰地分离了数据库操作的模板代码和特定实体相关的代码。这意味着你有了一个简单的持久化框架的雏形，而 BoundaryBase 类提供了这个框架里负责处理持久化的原型。记住第 9 章的忠告：最好的框架往往来自于可工作的代码。这个例子就是一个简单的持久化框架的萌芽。

第三，组合方法模式能不知不觉地暴露出重复代码（参阅第 5 章）。重复代码潜伏在软件开发的所有地方，甚至在一些你发誓说不可能有重复的地方。

关于组合方法模式还要注意：如果你严格遵循测试驱动开发 TDD 准则（参阅第 6 章），你几乎自然而然就会得到符合组合方法模式的代码。TDD 鼓励你，事实上是强迫你，编写精炼的方法（你所能测试的最小元素），这些使它成为组合方法。

提示：

TDD 实践推动组合方法模式。

SLAP

SLAP 强调每个方法中的所有代码都处于同一级抽象层次。换句话说，你不应该在一个

方法中既处理底层的数据库连接，又包含高层的业务代码，甚至还包含对Web服务的处理。当然与此同时，这样的方法也违反了Beck的组合方法原则。但即便你有了高内聚的代码，你也应该确保其满足SLAP原则。

这里有个例子。看看下面这个方法，它来自一个基于JEE的电子商务网站示例（比前面的例子要稍微复杂一些）。这个方法实现了往购物车里面添加订单的功能。为了简单起见，它还使用了底层的JDBC，但这不是我们讨论SLAP的重点。

```
public void addOrder(ShoppingCart cart, String userName,
                     Order order) throws SQLException {
    Connection c = null;
    PreparedStatement ps = null;
    Statement s = null;
    ResultSet rs = null;
    boolean transactionState = false;
    try {
        c = dbPool.getConnection();
        s = c.createStatement();
        transactionState = c.getAutoCommit();
        int userKey = getUserKey(userName, c, ps, rs);
        c.setAutoCommit(false);
        addSingleOrder(order, c, ps, userKey);❶
        int orderKey = getOrderKey(s, rs);
        addLineItems(cart, c, orderKey);
        c.commit();
        order.setOrderKeyFrom(orderKey);
    } catch (SQLException sqlx) {
        s = c.createStatement();
        c.rollback();
        throw sqlx;
    } finally {
        try {
            c.setAutoCommit(transactionState);
            dbPool.release(c);
            if (s != null)
                s.close();
            if (ps != null)
                ps.close();
            if (rs != null)
                rs.close();
        } catch (SQLException ignored) {
        }
    }
}
```

❶ 这个方法和在它之前的代码处于不同的抽象层次。

addOrder方法通过一系列步骤来搭建数据库的基础设施，然后又跳到业务层次去使用一些领域方法，比如addSingleOrder。基于每个步骤的不同需求，代码在不同抽象层次上近似于随机地切换，这使它很难阅读。

使用组合方法模式进行重构之后，这段代码看起来更干净一些了：

```
public void addOrder(ShoppingCart cart, String userName,
                     Order order) throws SQLException {
    Connection connection = null;
    PreparedStatement ps = null;
    Statement statement = null;
    ResultSet rs = null;
    boolean transactionState = false;
    try {
        connection = dbPool.getConnection();
        statement = connection.createStatement();
        transactionState =
                setupTransactionStateFor(connection,
                        transactionState);
        addSingleOrder(order, connection,
                ps, userKeyFor(userName, connection));
        order.setOrderKeyFrom(generateOrderKey(statement, rs));
        addLineItems(cart, connection, order.getOrderKey());
        completeTransaction(connection);
    } catch (SQLException sqlx) {
        rollbackTransactionFor(connection);
        throw sqlx;
    } finally {
        cleanUpDatabaseResources(connection,
                transactionState, statement, ps, rs);
    }
}

private void cleanUpDatabaseResources(Connection connection,
        boolean transactionState, Statement s,
        PreparedStatement ps, ResultSet rs) throws SQLException {
    connection.setAutoCommit(transactionState);
    dbPool.release(connection);
    if (s != null)
        s.close();
    if (ps != null)
        ps.close();
    if (rs != null)
        rs.close();
}

private void rollbackTransactionFor(Connection connection)❶
        throws SQLException {
    connection.rollback();
}

private void completeTransaction(Connection c)
        throws SQLException {
    c.commit();
}

private boolean setupTransactionStateFor(Connection c,
        boolean transactionState) throws SQLException {
    transactionState = c.getAutoCommit();
```

```
        c.setAutoCommit(false);
        return transactionState;
    }
```

❶ 即使有些方法只有一行代码，如果有助于使代码更好地遵循SLAP准则，那也是好的。

这段代码包括了更多的方法（其中有一些只有一行代码），但这样有助于更好地保证抽象层次的一致性。当然，有时Java需要你预先写一点繁琐、累赘的代码来处理所有的初始化，而且方法所有的语句都必须放在try...catch块中以使错误发生时可以回滚数据库。addOrder方法现在读起来感觉好多了，显示了真正的组合方法：公有方法读起来就像一系列必需步骤的纲要，虽然Java加进去了一些噪音。

还能进一步改进代码吗？它里面仍然包含一定数量的由Java引进的低层次代码。下面列出的代码在前一份代码的基础上进行了如下改进：把所有底层涉及的对象保存在Map里面；从Map得到所有JDBC需要的参数，而不是单独传入。

```
public void addOrderFrom(ShoppingCart cart, String userName,
                         Order order) throws SQLException {
    Map db = setupDataInfrastructure();
    try {
        int userKey = userKeyBasedOn(userName, db);
        add(order, userKey, db);
        addLineItemsFrom(cart,
                order.getOrderKey(), db);
        completeTransaction(db);
    } catch (SQLException sqlx) {
        rollbackTransactionFor(db);
        throw sqlx;
    } finally {
        cleanUp(db);
    }
}

private Map setupDataInfrastructure() throws SQLException {
    HashMap db = new HashMap();
    Connection c = dbPool.getConnection();
    db.put("connection", c);
    db.put("transaction state",
            Boolean.valueOf(setupTransactionStateFor(c)));
    return db;
}

private void cleanUp(Map db) throws SQLException {
    Connection connection = (Connection) db.get("connection");
    boolean transactionState = ((Boolean)
            db.get("transation state")).booleanValue();
    Statement s = (Statement) db.get("statement");
    PreparedStatement ps = (PreparedStatement)
            db.get("prepared statement");
```

```
        ResultSet rs = (ResultSet) db.get("result set");
        connection.setAutoCommit(transactionState);
        dbPool.release(connection);
        if (s != null)
            s.close();
        if (ps != null)
            ps.close();
        if (rs != null)
            rs.close();
    }

    private void rollbackTransactionFor(Map dbInfrastructure)
            throws SQLException {
        ((Connection) dbInfrastructure.get("connection")).rollback();
    }

    private void completeTransaction(Map dbInfrastructure)
            throws SQLException {
        ((Connection) dbInfrastructure.get("connection")).commit();
    }

    private boolean setupTransactionStateFor(Connection c)
            throws SQLException {
        boolean transactionState = c.getAutoCommit();
        c.setAutoCommit(false);
        return transactionState;
    }
```

这个版本牺牲了辅助方法的可读性来换取公有方法addOrderFrom的可读性（为了使它更易懂，我们修改了方法名）。其中最大的变化就是把所有逻辑上相同但语法不同的数据库属性都放到了Map里面，从而简化了方法签名。你也能注意到为了增加公有方法addOrderFrom的可读性，我把参数dbInfrastructure放在了所有方法的参数的最后一位。

很多时候，为了让代码更加清晰以及更具可读性所进行的重构，也要考虑取舍。这是由于代码本身存在一定的本质复杂性，所以问题就变成了“复杂性应该分布在哪里？”我更倾向于保持简单的公共方法，而把复杂性（这个例子中，复杂性存在于把值放到Map以及从Map取值的过程）放到私有方法里面。在实现一个类的时候，你已经需要专注于代码的细节，所以最好是在那时顺带处理复杂性。之后阅读公共方法时，你肯定不愿去弄清楚细节，而只想尽可能简单地弄清方法的功能即可。

提示：

把所有的实现细节封装在公共方法之外。

无论你是更喜欢第二版还是第三版的代码，它们都遵循了SLAP原则：通过大胆的重构以保证方法内的所有代码都处于同样的抽象层次。

第 14 章

多语言编程

计算机语言就像鲨鱼，要是保持静止就会死。和现实生活中的语言一样，计算机语言也在不断发展演化（不过幸运的是，青少年还不会往计算机语言里添加各种俚语——至少不会像英语里的俚语出现得那么快）。语言的变迁是为了适应周围环境的变化。例如，Java最近加上了泛型（generic）和注解（annotation），这应该归功于它与.NET之间永无休止的“军备竞赛”。不过，在某些时候，语言的变迁也可能反而降低效率。看看从前的一些语言（Algol 68或Ada），你就会发现：语言的发展是有界限的，要是走得太远，它就会变得笨重，最终不堪重负轰然倒下。Java已经接近自己的界限了吗？如果是，我们这些Java程序员的出路在哪儿？

本章将向读者介绍多语言编程的概念——在我看来这将是Java和.NET平台未来的出路，也是所有热爱这两个平台的程序员的出路。不过在深入进去之前，我们应该看看历史和现状：Java怎么了？多语言编程又能带来什么帮助？

历史与现状

如今在企业应用和其他很多软件开发领域里，Java已经成为无庸置疑的主流语言。像我这样曾经在那个人们听到“Java”时只会想到一个印尼小岛或者一种咖啡的年代里生活过的人，能亲眼见证Java的发展确实激动人心。但流行并不等价于完美：Java也有它自己的问题，大多是历史的原因（有趣的是，Java在一开始是作为一种全新的语言发明出来的，并没有任何向后兼容的需求）。现在我们就来看看Java如今是什么样子，以及它是如何走到如今这一步的。

Java的身世和发展

在那遥远的过去，一位传说中的神人（我们称他为James）需要为电烤箱和有线电视机顶盒发明一种新的编程语言。他不想使用那些众人皆知且深受喜爱的语言（C和C++），因为（就连真心喜欢这些语言的人们也承认）这些语言不适合这类用途。由于内存管理的问题每天重启几次计算机似乎还能忍受，但要是有线电视也得这么用就太烦人了。

于是有一天，James决定发明一种新语言，用来填补那些深受喜爱的旧语言力所不及的空缺。他创造了Oak，也就是后来的Java。（我跳过了这段传奇中的一些情节。）Java确实解决了C和C++的很多问题，而且恰好又赶上了第一次互联网浪潮。Bruce Tate把Java的流行称作“完美风暴”：一切条件在那时齐聚，才让Java如同超新星爆发一般迅速风靡全球。

在Java诞生之初，互联网和浏览器让所有人着迷。Java运行在那个年代的硬件和操作系统上还有些慢，但它有一点无可比拟的优势：它能以Applet的形式在浏览器里运行。虽然如今看来有些怪异，但确实是Applet让Java进入了人们的视野。当然了，历史总在轮回：我们仍然编写能在浏览器里运行的富客户端应用程序，不过改为使用JavaScript——现在JavaScript也快要焕发第二春了，这可真是够讽刺的。

当所有人都意识到“在浏览器里运行一个巨大的企业应用程序”不是什么好主意时，服务器端的Java就粉墨登场了，于是人们的字典里又加上了“Servlet”和“Tomcat”等字眼。

Java的阴暗面

出现在恰当的时间和地点并不代表Java就是一个完美的解决方案。Java有一些有趣的包袱。考虑到作为一种全新的语言，它原本不必有任何包袱，这些包袱就更显得有趣。在你学Java的时候，你经常会自言自语：“你在和我开玩笑吗——这也能行？怎么回事？”这些令人费解的东西就是Java的包袱。也许你已经忘了大部分这样的时刻，因为Java就是这样的。不过现在，让我们来回想几件这样的事吧。

那是什么时候的事

想想Java程序初始化的顺序。初始化是构造函数的职责，对吗？呃……对，又不对。你还可以创建静态初始化块（static initializer）或是实例初始化块（instance initializer）。静态初始化块在构造函数之前执行，实例初始化块则在构造函数执行过程中的某个时刻执行。而且初始化块你想写多少就可以写多少。你还可以在声明对象的同时调用其构造函数。那么，谁先来：静态初始化块，实例初始化块、还是被声明（同时被构造）的对象的初始化逻辑？还没明白？那就看下面这个例子：

```
public class LoadEmUp {
    private Parent _parent = new Parent();
    { System.out.println("Told you so");
    }

    static {
        System.out.println("Did too");
    }

    public LoadEmUp() {
        System.out.println("Did not");
        _parent = new Parent(this);
    }

    static {
```

```
        System.out.println("Did not");
    }

    public static void main(String[] args) {
        new LoadEmUp();
        System.out.println("Did too");
        Parent referee = new Parent();
    }
}

class Parent {

    public Parent() {
        System.out.println("stop fighting!");
    }

    public Parent(Object owner) {
        System.out.println("I told you to stop fighting!");
    }
}
```

不查文档，你能预测这几条消息出现的顺序吗？下面是运行这个程序的输出结果：

```
Did too
Did not
stop fighting!
Told you so
Did not
I told you to stop fighting!
Did too stop fighting!
```

要成为Java传道者，你就得学会理解一些神秘的、貌似无迹可寻的事情——例如你刚才看见的。

程序启动时的行为只是Java各种奇特行为的冰山一角。有些怪异的东西是在语言里根深蒂固的，例如数组的索引。

谁喜欢从0开始的数组

答案是那些习惯了大量使用指针的编程语言的人。你有没有问过自己，Java的数组为什么要从0开始？这毫无意义。显然，Java的数组从0开始，仅仅因为C的数组是从0开始的：为了向后兼容一种Java本身并不向后兼容的语言。

从0开始的数组在C语言里绝对有意义，因为C的数组实际上就是指针运算。看看图14-1，C的数组就是操作指针和偏移量的语法糖。（你甚至可以说C语言里几乎所有东西都只是指针的语法糖。）

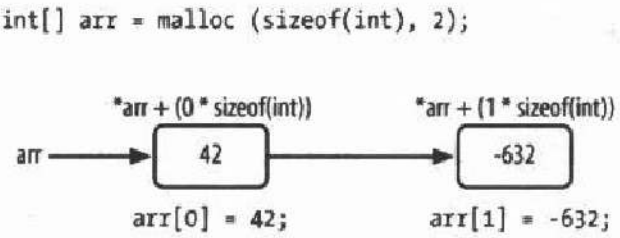


图 14-1：C的数组偏移量

所以，Java的数组之所以是从0开始是因为C的数组是这样的。虽然这能让程序员们感到舒服些（让他们由C/C++到Java的迁移容易些），但从语言的角度来说这毫无意义。在迁移到新的编程语言以后，开发者会很快地扔掉旧的惯用法。要是Java从一开始就提供foreach关键字，根本就不会有人关心数组的索引号。当然，Java最终还是提供了foreach操作符（令人迷惑的是，这个操作符也被叫做for），才只用了8年！

对从0开始的数组的盲目崇拜和向后兼容C语言的奴性让（原来的）for循环语法看起来是这样的：

```
for (int i = 0; i < 10; i++) {
    // some code
}
```

对C程序员来说这简直是宾至如归，可是对于那些从没见过或用过C的程序员（如今这样的人越来越多了）来说，这只能叫怪异。显然，愤怒的猴子们决定了这样的语法（译注1）。

令人郁闷的是，像这样令人费解的怪癖和惯用法在Java里还有很多，有些来自Java的创造者们，有些是前世带来的包袱。所有语言或多或少都有类似的情况。难道没办法躲开这些愚蠢的事情吗？

路在何方

幸运的是，Java的创造者们实际上创造了两样东西：Java语言和Java平台。后者就是我们摆脱历史包袱的途径。如今Java越来越多地被作为一个平台（而非一种语言）来使用，这种趋势会在未来几年中成为主流，最终我们都会被卷入其中——这就是我所说的多语言编程。

译注1：关于“愤怒的猴子”的故事请参见第11章。

如今的多语言编程

今天当我们开发Web应用程序时，我们主要在使用三种语言（如果算上XML就是四种）：Java（或者别的某种基础的通用语言）、SQL以及JavaScript（以Ajax库的形式）。尽管那些专用语言已经渗透到常规的通用语言中，大多数开发者还是会说他们自己是Java程序员（或者.NET程序员、Ruby程序员），而遗漏了其他语言。

多语言编程（polylot programming）是指除了一种通用语言之外，还使用一种或多种专用语言来构造应用程序。我们已经在这样做了，但正因为如此自然，以致于我们甚至没有把它当回事。比如说吧，SQL在开发中的地位是如此根深蒂固，几乎每个应用程序的开发都会用到它。

不过，跟我们熟悉的命令式的通用语言相比，SQL不啻是一只怪兽。脱胎于集合理论的SQL用于操作数据，所以它看起来就不像是“常规的”语言。大多数程序员已经可以接受这样的人格分裂：他们开心地使用SQL（或者用Hibernate来帮忙生成SQL），调试奇怪的SQL问题，甚至根据测量结果帮助数据库来优化SQL语句。这些已经成为每天软件开发中很自然的一部分了。

今天的平台，明天的语言

为什么要使用专用语言？显然，这是为了达成某些专门的目的。SQL无疑就属于这类语言；JavaScript也是，尤其是以人们现在使用它的方式来说。尽管这些语言针对不同的平台（Java运行在虚拟机上，SQL运行在数据库服务器上，JavaScrip运行在浏览器上），但它们共同组成了一个“应用程序”。

我们应该用好这个概念。如今Java平台支持很多种语言，其中一些是高度专门化的。这就是我们从怪异的Java语言脱狱而出的钥匙。

Groovy是一种开源的编程语言，它给Java带来了动态语言的语法和功能。它会生成Java字节码，因此可以在Java平台上运行。但在Java之后十多年里浮现出来的各种语言很大程度上影响着Groovy的语法：Groovy支持闭包、较松散的类型系统、“理解”迭代的集合，以及很多现代编程语言的改进特性。而且它可以编译成纯正的Java字节码。

看一个例子就一目了然了。作为一个经验丰富的Java程序员，你的任务是要编写一个简单的程序，从一个文本文件中读取内容，然后在每一行的前面加上行号打印出来。稍加思索你就会得到类似这样的程序：

```
public class LineNumbers {
    public LineNumbers(String path) {
        File file = new File(path);
        LineNumberReader reader = null;
        try {
            reader = new LineNumberReader(new FileReader(file));
            while (reader.ready()) {
                out.println(reader.getLineNumber() + ":"
                        + reader.readLine());
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
            } catch (IOException ignored) {
            }
        }
    }

    public static void main(String[] args) {
        new LineNumbers(args[0]);
    }
}
```

下面则是实现同样功能的Groovy程序：

```
def number=0
new File (args[0]).eachLine { line ->
    number++
    println "$number: $line"
}
```

这段Java程序有多少行？有时候，Java的语法要求不再是一种帮助，反倒成了约束。再看看Groovy这个版本，里面用到的单词数还不如Java版本的行数多呢！既然Java和Groovy最终产出的是同样的字节码，那为什么还要死抱着古板的Java语法不放呢？

到处都能听见Java程序员的哭喊："Groovy代码不可能有Java代码高效！"这是绝对正确的：Groovy生成的Java字节码有些累赘，加上了必要的声明、构造、异常块以及其他一些Java要求的繁文缛节。Java版本的程序会比Groovy版本的快上几百毫秒。那又如何呢？程序员的生产率比计算机的时钟周期更重要，而且摩尔定律（处理器的计算能力每过18个月翻一番）自然会解决这种问题。我们越来越多地关注如何让程序员顺畅地完成工作，而不是代码的执行性能。想想你最近写的5个程序：绝大多数情况下，网络和数据库的延迟比编程语言的执行速度更影响系统的整体性能，不是吗？

显然，Groovy确实能给Java一些别扭的地方注入新鲜的血液。不过Groovy还只是在字

节码之外做了一层包装而已，多语言编程走得比这还要远：有些种类的应用程序现在看来似乎不切实际，但多语言编程有可能让它们成为现实。

使用 Jaskell

如今的计算机大多拥有多个处理器。例如我现在写书用的这台笔记本电脑就有一个双核CPU，也就是说，从软件开发的角度来看，它也是一台多处理器的计算机。要让应用程序充分发挥多处理器的威力，就必须写出漂亮的线程安全的代码，而这是极其困难的。我们这些人当中就算对自己的编程技术最自信的那些程序员最终还是不得不认真研读Brian Goetz的《Java Concurrency in Practice》（Addison-Wesley），在那本书里Brian精辟地阐述了一个道理：用Java（以及其他任何命令式的语言）编写线程安全的代码是非常困难的。

最多就在5年前，我们炮制的那些丑陋得简直就像把命令行窗口直接塞进浏览器的Web应用程序还能让用户满意。可是那些讨厌的Google程序员毁了这一切：他们发布了Google Maps和Gmail，更要紧的是他们让用户明白Web应用程序不一定是那么糟烂的，于是我们只好跟着开发更好的Web应用程序。并发编程的情况也是一样：我们这些程序员现在还可以幸福地对严重的线程问题视而不见，但一定会有人站出来，让人们看到某种新发明的威力，然后我们又得亦步亦趋地跟上去。我们的计算机有强大的运算能力，而我们却没有能力写出那样的代码来充分利用它们，这对矛盾正在日益凸显。那么，为何不借助多语言编程来简化我们的任务呢？

命令式编程语言的很多缺陷在函数式语言中都不存在。函数式语言更严格地遵循着数学的原则，比如说函数式程序中的“函数”（function）就跟数学中说的“函数”一样：输出只与输入相关。换句话说，函数不会改变外界的状态。纯粹的函数式语言压根没有“变量”的概念：它们是无状态的。当然这有些不切实际，但确实存在一些出色的混合型函数式语言，它们既带来了人们期待的特性，又没有严重的可用性问题。这样的函数式语言包括Haskell、OCaml、Erlang、SML等。

尤其值得一提的是，函数式语言对多线程的支持比命令式语言要强得多，这都要归功于它们对无状态程序的鼓励。所以结论是：比起命令式语言来，用函数式语言更容易写出强壮的线程安全的代码。

现在来看看Jaskell：运行在Java平台上的Haskell版本。换句话说，它可以把Haskell代码变成Java字节码（注1）。

注1：　从 *http://jaskell.codehaus.org/* 下载。

下面这个例子来自 Jaskell 的网站。假设我们要用 Java 实现一个数组类，让用户可以线程安全地访问其中的元素，该类如下所示：

```
class SafeArray{
    private final Object[] _arr;
    private final int _begin;
    private final int _len;

    public SafeArray(Object[] arr, int begin, int len){
        _arr = arr;
        _begin = begin;
        _len = len;
    }

    public Object at(int i){
        if(i < 0 || i >= _len){
            throw new ArrayIndexOutOfBoundsException(i);
        }
        return _arr[_begin + i];
    }

    public int getLength(){
        return _len;
    }
}
```

同样的功能可以用 Jaskell 的 *tuple*（本质上就是关联数组）来实现：

```
newSafeArray arr begin len = {
  length = len;
  at i = if i < begin || i >= len then
           throw $ ArrayIndexOutOfBoundsException.new[i]
         else
           arr[begin + i];
}
```

由于 tuple 本质上是关联数组（associative array），所以对 `newSafeArray.at(3)` 的调用就会转发到 tuple 的 at 部分，从而执行在这部分里定义的代码。尽管 Jaskell 不是面向对象的，但继承和多态等机制都可以用 tuple 来模拟，而且一些程序员向往已久的功能（例如 mixin）在 Jaskell 中也可以用 tuple 来实现，而在 Java 语言中就做不到。minxin 允许你在不使用继承的情况下向一个类中注入代码（而不仅仅是增加方法签名），从而提供了一种取代接口与继承机制的可能性。AspectJ 之类工具所支持的面向切面编程（Aspect-Oriented Progrmming，AOP）也可以实现 mixin —— AspectJ 也是我们现在使用的多语言混合体中的成员之一。

Haskell（以及 Jaskell）支持函数的延迟求值，也就是说不到必要时不会对函数求值。比如说下列代码在 Haskell 中完全合法，但在 Java 中就不能工作：

```
makeList = 1 : makeList
```

这行代码的意思是："创建一个list，其中有一个元素；如果这个list的用户用到其中更多的元素，就根据需要将它们求值出来。"这行代码会创建一个拥有无穷多个元素的list，其中所有的元素都是1。

当然了，要想用好Haskell的语法（通过Jaskell），你的开发团队里必须有某个人了解Haskell。正如现在的项目里常常配备数据库管理员一样，今后的项目里还会有各种各样的专家，他们专门编写代码来解决某一领域特别的问题。也许你面前摆着一个复杂的安排算法问题，用Java来实现可能需要1 000行代码，而用Haskell只要50行就搞定了。既然如此，为什么不充分利用Java平台支持多种语言的能力，用其他更适合这项任务的语言来编程呢？

不过，在带来好处的同时，这种开发风格也带来了新的问题。与用一种语言开发的应用程序相比，多语言应用程序更难调试：问问那些曾经调试过JavaScript与Java之间交互的程序员吧，他们会赞同我的观点。就算到了将来，解决这个问题最简单的办法也跟现在一样：靠严格的单元测试来避免在调试器上浪费时间。

Ola的金字塔

多语言的开发风格会把我们带向领域特定语言（Domain-Specific Language，DSL）的方向。不用多久，我们的语言版图就会发生剧变：我们会以一些专门的语言为基础，创造出非常有针对性的、非常专注某一问题域的DSL。抱定一种通用语言不放的年代就快结束了，我们正在进入一个专业细分的新时代。大学时的Haskell教材还在书架顶上蒙尘吗？该给它掸掸灰了。

我的同事Ola Bini给多语言编程的思想又增添了几分色彩：他定义了一个全新的应用程序栈。他对于现代软件开发的世界观大致如图14-2：我们会用一种语言（很可能是某种静态类型语言）作为可靠的基础，用一种彰显开发效率的语言（很可能是某种动态语言，例如JRuby、Groovy或Jython）来完成日常编程任务，用多种领域特定语言[RL1]（参考第11章"连贯接口"一节中的讨论）让我们的代码更贴近业务分析师和最终用户的需求。我认为Ola找到了让多语言编程、领域特定语言和动态语言三者相辅相成的一个最佳方向。

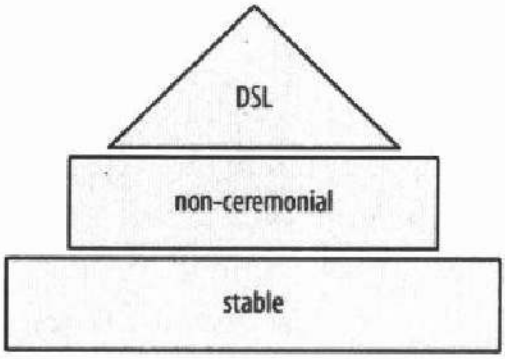
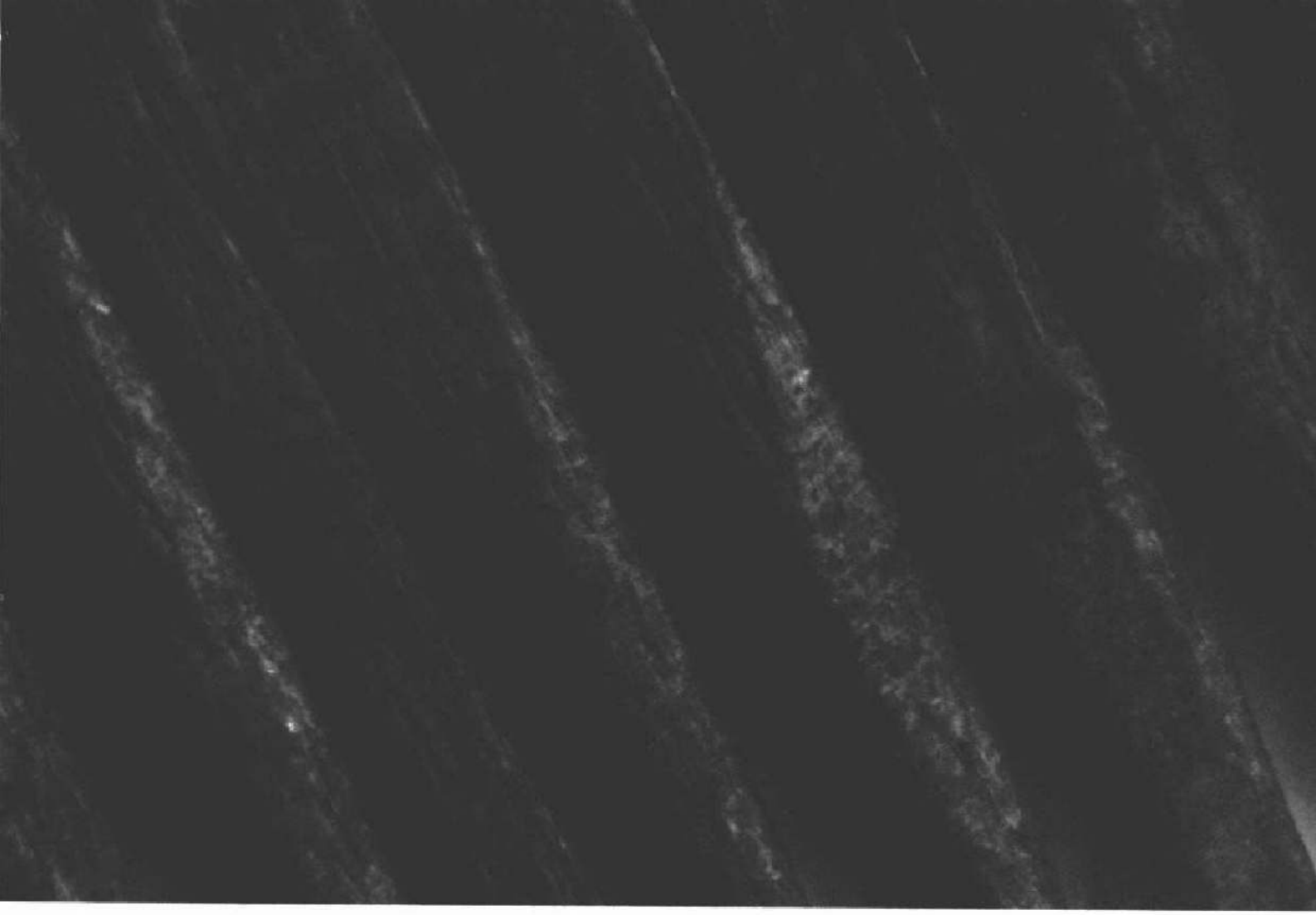


图 14-2：Ola 的金字塔

每个医生都曾经是多面手，但随着在专业领域的深入，分工细化就无可避免。软件的复杂度 —— 不仅因为我们要开发各种各样的应用程序，也因为底层平台本身变得日益复杂 —— 正在以飞快的速度驱使我们进行专业分工。面对这个充满挑战的新世界，我们必须拥抱多语言编程，让它为我们提供更专业的平台级工具，并借助领域特定语言来处理日益困难的问题域。5 年以后，软件开发将变得与如今大不相同。



第 15 章

寻找完美工具

到现在为止，我已经向你展示了一系列工具，它们可以用来解决各种各样的问题。这些工具包括：批处理文件、bash 命令（包括单命令行或整个脚本）、Windows PowerShell、Ruby、Groovy、sed、awk以及O'Reilly系列图书中提及的工具。现在，关键的时候到了，你已经明确了那些困扰你的问题，而且你想通过自动化让它们不再出现；但究竟该使用什么工具呢？第一种你最可能使用的工具是某个不起眼的文本编辑器。但它可能是你工具箱中最重要的一个，因此，我们先谈谈它。

寻找完美编辑器

开发人员的大部分时间仍旧在处理纯文本，因为不管现在有多少生成向导或者代码生成工具，大部分编码仍需通过手工输入纯文本的方式生成。同时，大部分信息仍然需要保存成文本格式，因为你不知道你所用的工具5年后是否仍在使用。因此，你最好继续保持具备阅读纯ASCII（也许是unicode）文件的能力，直至下个世纪。（就像在《The Pragmatic Programmer》（Addison-Wesley）一书中警告过的："把你的知识保存成纯文本"。）

既然有这么多需要围绕文本来解决的问题，那么最好要寻找一个完美的编辑器，而不是IDE。虽然公司的制度或者某些编程语言通常会告诉你，在代码编写方面，使用IDE是非常高效的方式，但我们仍然需要优秀的文本编辑器来编辑纯文本文件。

我承认我曾经有编辑器崇拜心理。在我的机器上曾装有6个编辑器，因为我喜欢它们不同的功能。直到有一天，我开始思考一个完美编辑器应该具有哪些特性。我把这些期望记录下来，然后挨个考察每个编辑器，从而一劳永逸地解决了选择编辑器的难题。

> **提示：**
>
> 寻找属于你的完美编辑器，并从里到外去了解它。

以下就是我创建的完美编辑器所需特性列表，这些特性是我认为完美编辑器应当拥有的，同时我会为每项特性举些例子。当然，你的列表可能有所不同。

Neal的完美编辑器所需特性列表

宏记录器

回想过去（大约几十年以前），宏是开发人员手里最重要的工具之一。我几乎每周都要使用它。如今，IDE里面模板的出现代替了宏的一些基本功能。而且实际上，一些很流

行的Java开发环境（如开源的Eclipse）并没有宏记录器。但如果你告诉一个20世纪80年代的UNIX程序员说一个现存的开发工具不允许记录宏，那么他们肯定会被吓坏。（坦白地说，下一版的Eclipse也许将增加对宏记录的支持。）

> **提示：**
>
> 你可以使用宏来记录所有重复的文本操作。

宏仍旧是一个功能强大的工具，但是你必须以某种特定的方式去思考问题。当你需要去从源代码中移去HTML标签或者用HTML标签包装代码时，你可以在一行上开始记录这些操作，并且在下一行相同的位置结束记录。这样你就可以在随后的所有行中回放这个宏。

完美编辑器应当具有易读的宏语法，因此你可以方便地保存通用宏并修改重用。比如，开源的JEdit编辑器就使用了一个类Java的脚本语言，叫BeanShell。每当你使用JEdit记录宏时，它会开辟一个缓冲区来保存它。这里有一个JEdit记录的宏的实例，它把一行中所有的HTML标签列表（由于显示间距的原因）扩展成每行一个条目。

```
SearchAndReplace.setSearchString("<li>");
SearchAndReplace.setAutoWrapAround(false);
SearchAndReplace.setReverseSearch(false);
SearchAndReplace.setIgnoreCase(true);
SearchAndReplace.setRegexp(true);
SearchAndReplace.setSearchFileSet(new CurrentBufferSet());
SearchAndReplace.find(view);
textArea.goToPrevCharacter(false);
textArea.goToNextWord(false,false);
textArea.goToNextWord(false,false);
textArea.goToNextWord(false,false);
```

正如你所看到的，BeanShell是很具有可读性的，它能保存有用的宏供将来使用。

可以从命令行调用

你应当能够从命令行中调用编辑器，并且可以传入一个或多个文件。TextMate（我的最佳编辑器之一）在这方面做得更好。当从一个目录中运行它时，它自动把这个目录做为一个工程，并显示目录下的所有文件和子目录。

正则表达式查找和替换

你的编辑器应当支持强大的正则表达式，用来提供单个文件或多文件的查找和替换。因为我们需要大量的时间处理文本文件，所以学好正则表达式语法是非常值得的。

学会使用正则表达式可以节省文本处理工作的时间。我曾经经历了一个事情，它让我领教了正则表达式的强大功能。事情发生在一个项目中，那个项目已使用了1000个EJB（Enterprise Java Bean），当时决定所有的非EJB方法（就是说除EJB托管的回调方法之外的所有方法）都需要一个额外的参数。原本估计需要一个人花6天来手工完成（这样的工作已经不用人来做了）。但一个熟悉正则表达式的开发员，用他信赖的编辑器（Emacs）仅用两个小时就做完了所有的替换。也就是那一天开始，我决定好好学习正则表达式。

有一个关于水管专家的很好例子。你雇了一个水管专家去修理一个大楼的水管问题。这个专家双手插兜在大楼里逛了几天，查看了大楼里的所有水管器具。在第三天行将结束时，他爬到了一个东西下面然后拧了一下一个阀门。“请支付2000美元”，你傻呆呆地盯着他，“2000美元？你仅仅拧了一个阀门！”“对”，他回答，“拧阀门1美元，但知道哪个阀门需要被拧则需1999美元”。

具备良好的正则表达式知识能让你变成水管专家那类角色的开发人员。在我刚提到的例子中，开发员花了1小时58分钟去建立正确的语法，然后用了不足两分钟去运行。在一些未曾培训过的人的眼里，他花的大多数时间都没有效率（这就是为什么他们会反对使用正则表达式的原因），但最后，他节省的是几天的工作。

提示：

掌握好正则表达式将为我们节省大量精力。

增强的剪切和复制命令

令人费解的是，大多数现代的IDE工具只有一个剪贴板并只允许操作一个条目。好的编辑器不仅提供剪切和复制功能，而且允许你附加复制和剪切，它能添加文本到现有剪贴板的文本上。这样你就可以在剪贴板里就组织好你的内容，而不用反复地在源文本到目标文体之间不停地切换。（很多时候，我看到开发人员在不停地复制、切换、粘贴、切换、复制、切换、粘贴……）而在你的源文件就做好所有复制相关的工作，再切换到目标文件，并粘贴全部的内容能很大地提高效率。

提示：

如果能批量处理，就不要来回做重复的工作。

多寄存器

拥有额外剪切和复制命令的编辑器通常也会有多个寄存器——实际上“多剪贴板”从

前就叫“多寄存器”。在完美的编辑器里，你能使用同键盘上的键一样多的剪贴板。你能在操作系统级别上创建多个剪贴板（查看第2章的“剪贴板”部分），但是如果你的编辑器也能支持这个功能，那就再好不过了。

跨平台

并非所有程序员都需要跨平台的编辑器，但是那些必须使用多个操作系统（也许一天好几个）的人就需要有一把在哪里都派得上用场的“瑞士军刀”。

编辑器参考列表

这里是一些最能满足我之前列出的特性的编辑器。当然，世界上不仅仅只有这些好的编辑器，这份列表只是一个起点。

VI

列表里肯定不能缺少这个编辑器。之前许多提及的高级功能都源于或基于此工具。VI仍在发展并越发强大。最流行的跨平台版本叫VIM（“VI Improved”），已经存在于多个平台上。美中不足的是它的宏语法不够易读。VI是相当难学的，但是一旦你掌握了它，你将是最高效的文本操作人员。看看富有经验的VI使用者，人们说光标总是能快速地跟随他们的眼球不断变换。当然，在VI和Emacs的用户群之间也有一些低级的争论，但实际上它们并不是一样的东西：VI致力成为终极文本操作工具，而Emacs却努力成为任何语言的IDE工具。VI的用户总是风趣地说“Emacs是伟大的操作系统，但只有最初级的文本编辑功能”。

Emacs

这是一种老式的、深受大众喜爱（未必是狂热）的编辑器。它具有以上所提及的各种功能（如果你认为它的宏语言elisp还算可读的话）。它有若干种版本：Emacs、XEmacs（为某些操作系统，例如windows，提供的带有图形界面的Emacs）和AquaEmacs（为Mac OS X提供的版本，它除了可以使用通用的Emacs命令之外，还可以使用本地的Mac OS X命令）。Emacs有时需要你不停转动手指，才能完成任务（正因如此，一些人总是在嘲弄Emacs，说它是“Escap、Meta Alt Control Shift”的缩写），不过它本身的确具有强大的功能。它可以为不同语言设定“模式”，支持复杂语言语法的增强显示，支持某些特殊功能，还可以承载其他行为。实际上，Emacs就是现代IDE的原型。

JEdit

我必须承认这一工具让我感到非常惊讶。我曾经使用JEdit好几年，后来逐渐不再

使用。但是，在我整理我的这个特性列表时，我再次重新评估了JEdit，它具有列表里的所有特性。JEdit是一个功能齐全的编辑器，并且它还允许使用插件，借此可利用大量的第三方功能（如Ant）并支持其他语言。JEdit本身构建于BeanShell，这意味着它很容易定制以及修改，尤其是对于Java开发员而言。

TextMate（和eEditor）

TextMate是Mac OS X下的编辑器，它赢得了许多用户（包括吸引了一些钟爱Emacs的用户）。公正地讲，它也是一个强大的编辑器，具有以上列表所提到的大部分特性，在Mac OS X下运行得也非常好。尽管它本身不能跨平台，但TextMate是如此流行以至于另外一家公司将其功能移植到Windows操作系统，并给它起了另外一个名字：eEditor。

选择正确的工具

Barry Schwartz在《The Paradox of Choice》（选择的悖论，由Harper Perennial出版社出版）一书中引用了一项研究来说明为什么太多的选择反而会困扰用户。太多的选择非但不能使用户高兴，反倒能让他们感到不乐。比方说，有一个商店在卖果酱，并让顾客免费品尝。他们放了3瓶果酱在桌上，结果销售业绩暴涨，原因在于客户能够在购买他们之前品尝果酱并决定购买哪个。同样的逻辑，商家决定拿出20瓶果酱用来品尝，却发现销售业绩下滑。提供3瓶试尝果酱能带来很好的效果，是因为顾客能够事先品尝；但是20瓶就太多了。顾客仍然在试尝果酱，但是他们面临过多选择，以至于麻痹了他们作出决策的能力，导致了果酱销售下滑。

在解决实际问题中，我们开发人员往往会遇见一样的问题：有许多解决方法可以选择，有时我们甚至未雨绸缪。比如说，在第4章的“用Ruby构建SQL分割器”一节中有个例子，我们创建了一个SqlSplitter来把sql文件分割成多个小块。跟我结对的开发伙伴最初试图使用sed、awk、C#甚至perl去解决这个问题，但是很快认识到这要花费太长的时间。面对多种多样的工具，你会如何选择呢？

我开始越来越多地使用那些“真正的”脚本语言（指一些支持脚本能力的通用语言）来做更多自动化的工作，它们同时也能胜任通用语言的那些繁重任务。你永远不会知道哪些小玩意最终会成为你项目中不可或缺的部分。某天你创建一个很小的工具，它本来用来解决一个很小的问题，但最终会因为你不断地添加更多的东西进去而成为一个新颖的功能点。总有一天，它会真正成为你项目中的一部分，并且你会像“真的”代码一样去对待它们（比如版本控制、单元测试、重构等）。这些“真正的”脚本语言包括Ruby、Python、Groovy、Perl等。

提示：

用“真正的”脚本语言去完成自动化任务。

重构 SqlSplitter 使其具有可测试性

回到第4章的：“用Ruby构建SQL分割器”一节，我描述了一个自动化的解决方案用来将大的SQL文件分割成若干个小的文件，这个方案的名称叫做SqlSplitter。我原先只是想用一次就丢了，但出乎意料的是，它成了我们项目中重要的一部分。因为它是用Ruby写的，你很容易将它由一个小玩具变成一个有价值的东西，包括用Ruby写单元测试使之保持健康也是非常简单的。好吧，那我们先重构一下SqlSplitter，这样你就能继续为其编写单元测试了。下面就是升级版本：

```
class SqlSplitter
  attr_writer :sql_lines❶
def initialize(output_path, input_file)
    @output_path, @input_file = output_path, input_file
  end

  def make_a_place_for_output_files
    Dir.mkdir(@output_path) unless @output_path.nil? or File.exists? @output_path
  end

  def lines_o_sql❷
    @sql_lines.nil? ? IO.readlines(@input_file) : @sql_lines
  end

  def create_output_file_from_number(file_number)
    file = File.new(@output_path + "chunk " + file_number.to_s + ".sql",
        File::CREAT|File::TRUNC|File::RDWR, 0644)
  end

  def generate_sql_chunks
    make_a_place_for_output_files
    line_num = 1
    file_num = 0
    file = create_output_file_from_number(1)
    found_ending_marker, seen_1k_lines = false
    lines_o_sql.each do |line|❸
      file.puts(line)
      seen_1k_lines = (line_num % 1000 == 0) unless seen_1k_lines
      line_num += 1
      found_ending_marker = (line.downcase =~ /^\W*go\W*$/ or
            line.downcase =~ /^\W*end\W*$/) != nil
      if seen_1k_lines and found_ending_marker
        file.close
        file_num += 1
        file = create_output_file_from_number(file_num)
        found_ending_marker, seen_1k_lines = false
      end
```

```
    end
    file.close
  end
end
```

❶ 为sql_lines成员变量增加一个attr_writer属性。这将允许你在类构造完毕后，任何其他任何方法被调用前将一个测试值注入到类里。

❷ lines_o_sql方法封装了类的成员变量，确保在被调用时它都有值。任何其他方法都不需要知道sql_lines内的成员变量是如何被填充等细节。

❸ 用一个方法去轮询源文件行，允许你传入自己的内容行用来测试，而无需总依靠输入文件。

重构代码之后，写单元测试就非常简单了：

```
require "test/unit"
require 'sql_splitter'
require 'rubygems'
require 'mocha'

class TestSqlSplitter < Test::Unit::TestCase
  OUTPUT_PATH = "./output4tests/"
private
  def lots_o_fake_data❶
    fake_data = Array.new
    num_of_lines_of_fake_data = rand(250) + 1
    1.upto 250  do
      1.upto num_of_lines_of_fake_data do
        fake_data << "Lorem ipsum dolor sit amet."
      end
      fake_data << (num_of_lines_of_fake_data % 2 == 0 ? "END" : "GO")
      num_of_lines_of_fake_data = rand(250) + 1
    end
    fake_data
  end

  public
  def test_mocked_out_dir
    ss = SqlSplitter.new("dummy_path", "dummy_file")
    Dir.expects(:mkdir).with("dummy_path")❷
    ss.make_a_place_for_output_files_in(dir)
  end

  def test_that_output_directory_is_created_correctly❸
    ss = SqlSplitter.new(OUTPUT_PATH, nil)
    ss.make_a_place_for_output_files
    assert File.exists? OUTPUT_PATH
  end

  def test_that_lines_o_sql_has_lines_o_sql❹
    lines = %w{Lorem ipsum dolor sit amet consectetur}
    ss = SqlSplitter.new(nil, nil)
```

```
    ss.sql_lines = lines
    assert ss.lines_o_sql.size > 0
    assert_same ss.lines_o_sql, lines
  end

  def test_generate_sql_chunks❺
    ss = SqlSplitter.new(OUTPUT_PATH, nil)
    ss.sql_lines = lots_o_fake_data
    ss.generate_sql_chunks
    assert File.exists? OUTPUT_PATH
    assert Dir.entries(OUTPUT_PATH).size > 0
    Dir.entries(OUTPUT_PATH).each do |f|
      assert f.size > 0
    end
  end

  def teardown
    `rm -fr #{OUTPUT_PATH}` if File.exists? OUTPUT_PATH
  end
end
```

❶ 这块代码会生成一个数组，其中每个元素大致就是一组SQL语句，不过用来分隔的记号（“GO”和“END”）还在其中。

❷ 使用Mocha（Ruby的mock库）把“创建目录”的逻辑mock掉以便测试。

❸ 测试数据目录是否被正确创建。

❹ 测试lines_o_sql方法返回的数组，验证传入的参数是否由传给sql_lines的那些字符串组成的数组。

❺ 这是主要的测试。它测试了SqlSplitter类分析输入文件以及产生输出文件的功能。

这个版本测试了SqlSplitter的各个方面，包括mock文件系统使你能测试这段程序是否与操作系统正确交互。因为这是Ruby代码，我可以运行rcov（一个代码覆盖工具）去保证百分之百的测试覆盖。对于脚本来说，这是非常重要的，尤其是对于偶尔才发生的边缘测试用例而言。

C0 code coverage information

Generated on Mon May 28 08:26:30 CEST 2007 with rcov 0.6.0

Name	Total lines	Lines of code	Total coverage	Code coverage
TOTAL	39	33	100.0%	100.0%
sql_splitter.rb	39	33	100.0%	100.0%

Generated using the rcov code coverage analysis tool for Ruby version 0.6.0.

Valid XHTML 1.1! Valid CSS!

图15-1：SqlSplitter的测试覆盖率

这段代码原先没有测试，但是当把这个小工具作为“真正的”代码来使用时，测试就变得重要了。命令行工具（例如*bash*、*sed*、*awk*）的一个严重缺点是缺少可测试性。当然，一般情况下你不需要测试这三种类型的工具，除非你真的想去测试。Ant的一个缺点是当Ant文件增长到上千行时，它就不具可测试性了。因为Ant是基于XML的，你不能很容易地去比较或者重构（尽管一些IDE支持有限的重构），或者为它做一些对于其他“真正的”编程语言来说自然而然的事情，比如单元测试。

没有人不让你为bash脚本写单元测试，但这的确有点难。没有人想去测试命令行工具，因为他们不认为这些工具能在这方面提供足够的支持。这些小工具总是以很简单的形式出现，但当它们终于变得对项目至关重要时，它们也会变得难以调试的怪兽。

将行为保留在代码中

大部分企业级开发都离不了XML。实际上，一些项目拥有“真”代码一样多的XML。XML开始被引入到开发领域有两个原因。首先，它很容易被解析，大量的标准库使其很容易被开发和使用，这是它取代了那些UNIX风格的“小语言”（用来配置UNIX系统各部分的配置文件）的主要原因。其次是因为我们逐渐认识到，重用代码最好的方法之一就是通过框架，而框架有两个基本部分：框架本身的代码和允许用户驱动框架的配置信息。配置部分通常采取后期绑定，因为它可以改变代码而无需重新编译应用程序。这就是早期EJB版本的巨大卖点：由部署专家来把事务特性与EJB糅合起来。现在听起来有点滑稽可笑，但在当时这可真能打动人。不过直到今天，通过配置实现延迟绑定的功能仍是一个有用的想法。

那么，XML的问题在哪里？问题在于它不是“真正的”代码却要硬逞能。XML的可重构性很差，编写也很麻烦，试图用*diff*命令去区分XML文件是件恐怖的事情。并且，它对计算机语言所应当具有的东西（比如变量）支持很弱。

幸运的是，我们能通过生成而不是手写的方式来解决XML的问题。现代动态语言都有标签构造器（markup builder），专门用来构造XML。来看个例子：*struts-config.xml*文件存在于每一个Struts项目中（Structs是一个流行的Java框架）。*struts-config*配置文件允许你配置数据库连接池，大概是这样一段XML：

```
<data-sources>
    <data-source
            type="com.mysql.jdbc.jdbc2.optional.MysqlDataSource">
      <set-property property="url"
            value="jdbc:mysql://localhost/schedule"  />
      <set-property property="user" value="root"  />
      <set-property property="maxCount" value="5"  />
```

```
      <set-property property="driverClass"
            value="com.mysql.jdbc.Driver"  />
      <set-property value="1" property="minCount" />
    </data-source>
  </data-sources>
```

如果我们想让最小数据库连接数总是5，而且比最大连接数要小，该怎么做？因为XML没有很好的变量机制，所以我们不得不手工管理，当然这很容易出错。看看这一版本的XML，它是用Groovy的标签生成器来生成的：

```
def writer = new StringWriter()
  def xml = new MarkupBuilder(writer)
  def maxCount = 10
  def countDiff = 4

  xml.'struts-config'() {
    'data-sources'() {
      'data-source'(type:'com.mysql.jdbc.jdbc2.optional.MysqlDataSource') {
        'set-property'(property:'url', value:'jdbc:mysql://localhost/schedule')
        'set-property'(property:'user', value:'root')
        'set-property'(property:'maxCount', value:"${maxCount}")
        'set-property'(property:'driverClass', value:'com.mysql.jdbc.Driver')
        'set-property'(property:'minCount', value:"${maxCount - countDiff}")
      }
      //...
```

使用标签生成器，你就能用代码来描述XML的层级结构，用参数和名称/值组合产生XML属性和子元素。这种方法更易读，因为其中的XML噪音（例如尖括号）更少。但真正的好处是，你可以很容易地声明变量。在这个例子中，minCount是基于maxCount的，也就是说你不必再手工同步值。使用Groovy Ant任务，你可以把这个builder文件包含在构建流程中，每次构建时它都会自动生成你需要的XML配置。

很好。但如果你已经有了一个*struts-config.xml*文件，又不想在builder文件里重写一遍，该怎么办？可以采用反向解析它！这是一段Groovy代码，它会将一个现存的XML文件转化成Groovy代码：

```
import javax.xml.parsers.DocumentBuilderFactory
import org.codehaus.groovy.tools.xml.DomToGroovy

def builder     =
    DocumentBuilderFactory.newInstance().newDocumentBuilder()
def inputStream = new FileInputStream("../struts-config.xml")
def document    = builder.parse(inputStream)
def output      = new StringWriter()
def converter   = new DomToGroovy(new PrintWriter(output))

converter.print(document)
println output.toString()
```

Ruby也有类似的生成器（事实上，Ruby生成器是从Groovy那里获得的灵感）。使用生成器产生代码能让开发人员更加方便。记住，不要手写XML，而应自动生成。当然，生成又反向解析XML看上去有点傻。（为什么不移除XML这个“中间人”呢，就像Ruby所做的——Ruby世界里大部分的配置都是Ruby代码和YAML，这样配置就成了内嵌Ruby代码）。当然，你不可能期望其他语言（像Java，C#）开发的框架也用真实代码代替配置，但至少你可以写代码去生成XML，尽量把行为放入代码中。作为第一步，应该用反向解析器将配置信息放入代码中。

提示：

将行为保留在（可测试的）代码中。

如果一开始就用强大的脚本语言来对项目进行自动化，一旦发现这些基础设施的代码开始变得重要，你就能随时把它们变成“真正的”代码。这也意味着你没必要学习一大堆专用的工具。事实上，今天的脚本语言几乎能胜任命令行所能完成的一切任务。

越是有用的东西就越不会消失，它们会持续发展，最终变成开发中的重要部分。各种各样的小工具加在一起会变成系统中重要的部分，因此也需要更多的关怀。如果你在一开始就正确构建它们，就不必日后去重写。尽可能将行为写入代码（而不是放在工具或XML之类的标签语言中）。我们知道各种操作代码的方式：用diff比较文件版本、重构、强壮的测试库。为什么要浪费我们积累的所有代码知识去给某些复杂的工具救火呢？

提示：

关注每个小工具的演化。

丢弃错误的工具

相比选择正确的工具而言，拒绝不好用的工具更为重要。实际上，有一种反模式称之为：船锚（boat anchor）。所谓船锚，是指你被迫使用的工具，即使它并不适合你手边的工作。船锚经常要花费你大量的金钱，而这又恰恰增加了政治压力强迫你在每种条件下去使用它。有个痛苦的但又不失准确的比喻：试想一个木匠正被迫用一个雪橇（强大的功能）去起钉子。这明显是不合适的，但是我们在软件行业中却经常作出类似甚至更糟的决定。

我最近参加了某家大企业的项目启动会。我们介绍了许多敏捷原则给开发人员，他们感

到非常新奇。其中一个决定是采用版本控制。负责基础架构的代表来找我谈，提供了两个选择：Rational ClearCase 或 Serena Version Manager。如果你不是很了解这两个版本控制工具，我可以告诉你它们是相当昂贵而且功能庞大的。那我们如何选择呢？一个都不要。我们建议用Subversion，一款轻量级的开源版本控制工具。我们给代表们介绍了Subversion，他们同意了，而且看起来非常满意。然后，他们抛出了一个最大的问题："每个用户需要多少授权使用费？"当我告诉他们这是免费的时候，他们快吓晕了。他们说："你知道吗，我们在6个月之前给另外一组开发人员安装了ClearCase，他们看起来并不怎么喜欢它。"

大公司总是希望通过采购得到一个能够解决所有问题的工具。对于一家大公司而言，标准化基础设施的确是有意义的。但是从某些角度来看，标准化基础设施成为了一种阻碍而不是好处——这在一些复杂的工具上体现得尤为明显。实际上，我想到了一个词：复杂税（complexitax）。复杂税是指你为一个大而不当的工具所附带的复杂性而付出的额外成本。许多项目陷于工具带来的复杂当中，说它们能提高效率，实在是一种讽刺。

提示：

尽量少交复杂税。

演示较为简单的方案

公司CIO的想法不难理解：如果能把不多的几种工具标准化，我们就不必做太多的培训，还能让雇员更容易地在不同项目中穿插。更糟的是，软件提供商不停地约他们打高尔夫球（我有一个同事，每当他看到一个没用过的工具丢在角落时总是说："但愿他们球打得开心"），无情的市场竞争让他们使尽各种手段让产品得以出售。但你可以打败这些船锚：演示更为简单的工具，证明它确实适合具体项目。以一个Web应用程序为例，你可以启动一个很小的工程来展示Tomcat比Websphere更适合小型Web应用程序，因为我们想用脚本控制部署，而这在Tomcat中更加简单。

有时就连给出这么一个演示的机会都很难得到，尤其是当演示需要花一点工夫的时候。千万别卑躬屈膝，坚持做正确的事，否则你会浪费大把宝贵的时间。

先斩后奏

我的一个朋友，Jared Richardson，接受了一项困难的任务：让一个世界上最大的软件公司变得更加敏捷。经过观察，他发现每晚的编译失败是一个最大的问题。他没有向公司高层要求为某些项目搭建CruiseControl，他只是找了一台没有人用的旧台式机，在上面安装了CruiseControl，然后为几个棘手的项目搭建了持续环境，并且使用E-mail提醒那些破坏了build的开发人员。接下来的几天，几个开发者找到

他并让他关闭E-mail通知功能。他说："很简单，不要破坏build。"最终，开发者们认识到那真的是摆脱E-mail纠缠的唯一方法，于是他们开始清理自己的代码。

于是这家公司从一个月里只有3个成功的build，终于变成一个月只有3个失败的build。一些敏感的经理开始四下探访："是什么玩艺让你做到这样的结果？"现在这家公司已经成为世界上安装CruiseControl最多的公司。

使用柔道方法

柔道是一种武术，他鼓励学习者借助对手的力量克敌。我们曾为一家大公司做过一个项目，他们在整个公司范围强制使用了一套标准的版本控制软件。我们试了后发现它与我们正在做的开发风格截然相反。我们需要尽早、尽可能频繁地提交代码，并且保证不会锁住文件（否则大规模的重构会相当难），但这款工具处理不了。这就破坏了我们的工作流程，大大降低了我们的生产率。基于这些事实，我们提出了反对意见。

最终，我们与客户达成了妥协。他们允许我们使用Subversion，但为了遵循公司政策，会有一个定时任务在每天凌晨两点从Subversion签出（checkout）代码并签入到公司原有的版本控制器中。他们保证了代码保存在一个标准的地方，而我们则能使用适合我们工作的工具。

对抗内部特性蔓延和船锚

软件厂商固然是附属复杂性的推动者，但组织内部也有他们的同谋。船锚不一定是外部的工具，这些令人头疼的东西常常是早已存在于组织内部、由自己开发的。许多项目的构建基于不合适的内部架构或工具（反模式站在侏儒肩膀上的受害者）。业务使用者只想看到他们想要的功能，不想理解背后有多少复杂的东西。开发人员、架构师、技术主管则必须让用户和管理人员了解：不合适的工具、软件库、架构给他们带来复杂性，并且需要为其付出昂贵的代价。

使用了不合适的工具看似是一件微不足道的事情（尤其对于非开发人员而言），但是它对整个团队开发效率的影响可能是巨大的。打个比方，切掉你的上鄂不会要你的命，但不断的疼痛让你不能集中精力在重要的事情上。不合适的工具对于工作来说也是这个道理。过于复杂的工具让事情变得更糟，因为你花了太多的时间来关注工具，却不能把真正该做的工作做好。

第 16 章

结束语：继续对话

编程是一种独特的行为。无论我们搜肠刮肚地找出多少类比，它跟任何其他活动和职业还是非常不一样。编程是工程和技巧的高度结合，它要求开发人员掌握多种多样的技能：分析思维能力，对各层次的细节以及美学的高度敏感，同时关注宏观和微观两个层面，以及对软件服务的目标对象具有敏锐和细致的理解能力。毫不夸张地说，开发人员必须比实际的业务执行者对业务流程更加知根知底：业务人员可以根据经验对新情况作出感性决定，而我们必须把一切都编码成算法和明确的行为。

写此书的初衷是给大家提供一本关于如何提高生产率的技巧手册。但这本书最终演化成了两个部分：第一部分介绍了生产率的机制，第二部分专注于讲述提高开发人员生产率的实践。虽然这本书仍旧介绍了一定数量的技巧，但记住：讲述技巧只能“授之以鱼”，而不能“授之以渔”。给出生产率法则（加速法则、专注法则、自动化法则和规范性法则）的定义，为我们将来识别新技术提供了一套命名系统。所以归根结底，我想写的是一本能让你拥有“捕鱼”能力的书。

第二部分，“实践”，介绍了一些可能你从未用过的构建软件的方法。开发人员有时候会陷入泥沼，需要旁人提供一些帮助或带给他们一些新思维。但愿第二部分能在此方面有所贡献。

实际上，本书背后的目标是推动一场关于程序员生产率的对话，来讨论生产率的机制和实践，而不仅仅是我的独白。我希望能通过这种方式来加强我们对如何提高自身生产率的认知。同时，我也希望其他一些更聪明的人来继续这场对话。一起努力，我们可以想出很多惊人的东西。

这也就意味着，永远也不可能有一本书能巨细靡遗地涵括生产率的各个方面，因为我们对提高生产率的追求永无止境。为了激励大家基于我的独白展开讨论，我创建了一个wiki：*http://productiveprogrammer.com*。当你找到一些让你更加高效的方法时，不要吝啬，告诉其他人吧；当你发现一个关于生产率的模式（或者反模式）时，不要吝啬，发表它吧。作为一个团体，提高我们生产率的唯一途径是合作、分享，并持续地发现。

来吧，让我们继续这个对话。

附录

构建块

命令行是个好东西。只要你懂得那些神奇的命令，命令行通常是把一件事从想法变成现实最快的方式。从前，开发人员必须用心学习所有那些咒语般的命令——他们没有选择。那时的计算机杂志也充斥着有趣的关于DOS工作方式的细微差别（也经常关于它不工作的方式）的内容。当Windows征服了用户的桌面电脑后，开发人员也紧随其后，只有我们这些“老鸟”知道幕后的黑魔法。

虽然众多IDE赋予初级开发人员更大的生产率，最有生产率的开发人员依然大量依赖命令行这方利器。通过脚本编写自动化任务，连接现有工具的输入和输出，以及操纵本地的和远程的文件，完成这些任务的最佳方式仍然在命令行那闪烁的光标处。但是首先，你必须确保拥有正确的“闪烁的光标”。如果你用UNIX或Mac OS X，你可以跳过下一节。但如果使用Windows，你毫无疑问需要阅读这些内容。

Cygwin

作为一个Windows用户，你是否曾嫉妒过Linux平台拥有全套的软件包？针对超过一打语言的编译器、调试器、文本编辑器、绘图工具、Web服务器、数据库、出版工具……这个列表简直望不到头。你在Windows上也可以拥有这些（注1）——感谢Cygwin。

Cygwin是一套工具的合集：

- 一个Linux API模拟层，使得你可以编译和运行Linux上那些很酷的程序
- 一套绝佳的UNIX风格的工具集
- 一个安装程序管理工具来保持所有程序更新

首先，从*http://www.cygwin.com*下载Cygwin的安装包。这份软件不只是一个安装程序，而是一套完整的软件包管理系统。在安装Cygwin后，你仍需要保存这个安装包，因为将来需要用它来安装、更新以及卸载软件包。你需要下载的安装文件很小（大约300kB），但那只是冰山一角——它会下载一堆东西，根据你的安装选项，可能要几百兆字节（运行本书的例子不需要安装这么多）。对那些网络带宽受限的人，可以购买安装光盘，价钱也不贵。

对大部分Windows用户来说，Cygwin的安装程序看起来有点儿奇怪。软件包选择页面列出了一堆要安装的程序，每个都可以单独选择安装、升级或卸载。一开始这可能让人觉得迷惑，但你只要记住，这是一个“软件包管理工具”，而不是“安装程序”，那就很好理解了。

注1：当然，Mac用户也能得到所有这些。如果你需要的UNIX工具没有包含在OS X里，你同样可以下载源码自己构建，或者通过一个包管理工具来安装，例如Fink或MacPorts。

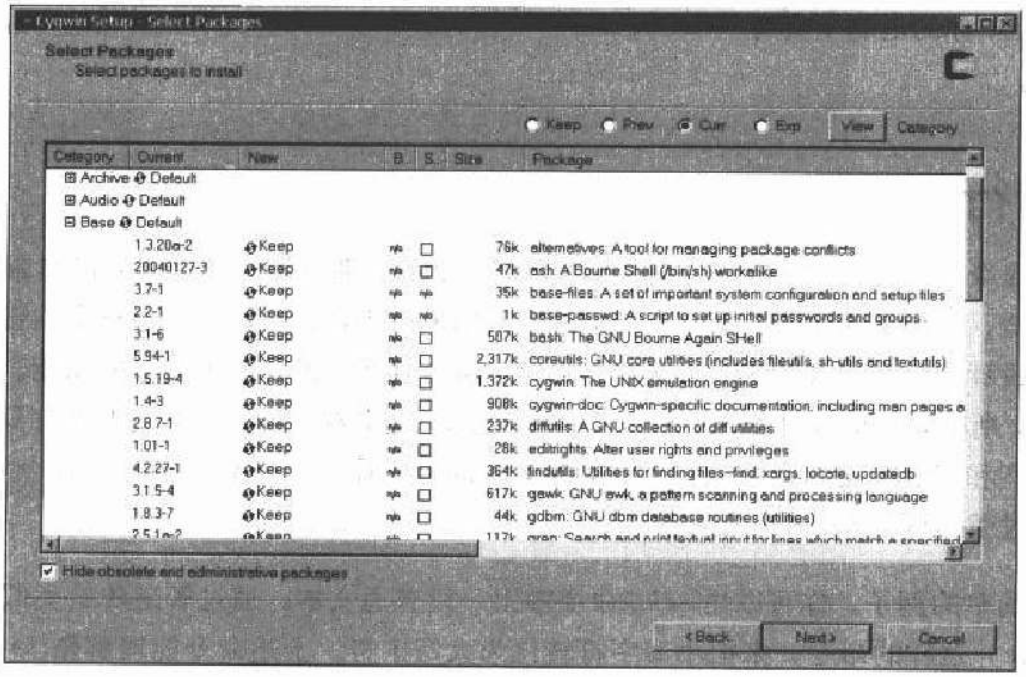


图 A-1：Cygwin 安装程序实际是一个软件包管理工具

下一步，你必须在 DOS 和 UNIX 格式的换行符之间做出选择。本书的答案是：UNIX。原因比较奇妙：文本文件的每行末尾都存在不可见的字符，这些字符的顺序在 DOS 和 UNIX 上不一样。现在你既然准备在自己的机器上安装 UNIX 工具，你的文件系统上将同时存在 DOS 和 UNIX 格式的文件。如果选择了 UNIX 格式的换行符，你有时会看到打开的文件每行末尾都有一个 'M' 字符；但如果选择了 DOS 格式，你在使用所有那些期望 UNIX 格式换行符的工具时将遇到麻烦。从实用的角度讲，当今的大部分应用程序（除了那些特别糟糕的老古董，比如 Windows 记事本）都能处理这种换行符的差异。

最后，DOS 和 UNIX 的路径格式不同。在 Windows 上，文件路径可能像这样：*c:\Documents and Settings\nford*；但 UNIX 上的路径可能是：*/home/nford*'。由于 Cygwin 安装在 Windows 文件系统上，你在 Cygwin 中可能看到文件在 */home/nford/readme.txt*，但是在 Windows 中它在 *c:\cygwin\home\nford\readme.txt*。Cygwin 中提供了从一种路径格式到另一种的转换工具。有些工具要求某种特定的格式，你必须了解如何在不同的路径格式间互相转换。

安装 Cygwin 时，有一个软件包列表供你选择。浏览各项分类，查看其中的好东西，你会发现各种各样的文本工具、数据库工具、shell 工具、Web 服务器、编程语言（比如 Ruby、Python 等），以及大量其他实用工具。对本书示例的需求来说，选择默认配置即可（注意检查选上 wget）。下载和安装过程要花费一点儿时间。

完成之后，你的电脑上会有一个Cygwin的快捷方式，吸引你进入Windows版bash shell的世界。

命令行

为什么命令行如此有用，特别是在UNIX世界？为什么那些UNIX奇客们谈起命令行就充满感情，口水都流到鞋子里？这要追溯到UNIX创造者们构建于命令行中的哲学。他们想要设计出一套强大的工具，这些工具可以混合使用，组合出更强大的功能。为了这个目标，所有东西的设计都基于一个简单的概念：纯文本字符流。事实上所有UNIX工具都生产和消费纯文本字符流。即使文本文件也能被看作字符流（通过cat命令），处理后再通过重定向（>）命令写入文件。

举一个简单的例子，你可以用*echo*命令输出一段普通文本，通过管道传给类似*tr*（或translate）这样的命令，来把小写字符转化为大写字符。上述命令能理解像*:lower:*和*:upper:*这样的字符类，例如（注意$是命令提示符，不是命令串的一部分，留在那里以便你能区分输入和输出）：

```
$ echo "productively lazy" | tr "[:lower:]" "[:upper:]"
PRODUCTIVELY LAZY
```

这里值得注意的概念是管道（就是“|”字符）命令。它获得*echo*命令的输出，传给*tr*命令作为输入，后者忠实地执行小写字符到大写字符的转换。所有的UNIX命令都是这样组合工作的，前一个命令的输出作为后一个命令的输入。对上面的例子来说，你同时也可以通过输出重定向来把结果写入文件。

```
$ echo "productively lazy" | tr "[:lower:]" "[:upper:]" >pl.txt
$ cat pl.txt
PRODUCTIVELY LAZY
```

而且，毫无疑问地，你可以获取一个文件的内容，处理它，然后写入另一个文件。

```
$ cat pl.txt | tr "[:upper:]" "[:lower:]" | tr "z" "Z" > plz.txt
productively laZy
```

这里有一个更实用的例子。假设我有一个很大的Java项目，其中有一些辅助类，它们的名字按照惯例都以Helper结尾。现在我想找出所有这些辅助类的Java文件，尽管它们分散在项目各处。

```
$ find . -name *Helper.java
```

我那些训练有素的仆人忠实地响应：

```
./src/java/org/sample/domain/DomainHelper.java
./src/java/org/sample/gui/WindowHelper.java
./src/java/org/sample/logic/DocumentHelper.java
./src/java/org/sample/logic/GenericHelper.java
./src/java/org/sample/logic/LoginHelper.java
./src/java/org/sample/logic/PersistenceHelper.java
```

没错，你也可以在IDE的查找对话框里得到同样的结果。但命令行的有用之处在于得到了这些结果之后你能做什么。在IDE里，查找结果出现在一个窗口里，如果够幸运的话，你可以从中复制粘贴。但在命令行里，你可以把结果通过管道传给别的工具，比如*wc*。*wc*是一个统计工具，能够统计词、字符、行或者文件的数量。

```
$ find . -name *Helper.java | wc -l
6
```

*wc -l*简单地给出传给它的内容的行数。

这都是谁起的名字？

UNIX命令通常简明扼要。毕竟，它们被设计来在字符终端上工作，并以其简明精炼回报给那些学会这种方式的用户。一旦你掌握了它们，只需要打很少的字就能调起那些超酷的功能。但是你可能会对*grep*命令感到困惑，它的名字是怎么来的？

据传*grep*命令来自ex编辑器中的查找命令。*ex*是行编辑器的先驱，是VI的前身——后者是一款极富传奇色彩的UNIX编辑器，以其陡峭的学习曲线闻名。在*ex*里，你进入命令模式，键入*g*来做全局搜索，接下来是以*/*开头和结尾的正则表达式，最后输入*p*以打印查找结果。总的来说，就是*g/re/p*。在UNIX里，这种做法是如此的普遍以致于它成了一个动词。所以，当UNIX需要一个基于命令行的查找工具时，它已经有了一个完美的名字：*grep*。

我知道有些辅助类继承自别的辅助类。前面我已经可以用*find*命令得到所有的辅助类了，现在我还想看看文件内部的内容，找到那些继承别的类的类文件。现在是*grep*派上用场的时候了。我们可以看到三种组合使用*find*和*grep*的方式。第一种方法是使用*find*命令的扩展项：

```
$ find . -name *Helper.java -exec grep -l "extends .*Helper" {} \;
./src/java/org/sample/logic/DocumentHelper.java
./src/java/org/sample/logic/LoginHelper.java
```

这个命令做了什么？附表A-1给出了分析。

附表 A-1：揭秘命令行魔法

字符串	用途
find	执行 find 命令
.	查找当前目录
-name	匹配像 “*Helper.java” 这样的名字
-exec	对每个搜索到的文件执行后面的命令
grep	grep 命令
-l	打印包含匹配字符串的文件
"extends .*Helper"	待匹配的文本模式：“extends” + 空格 + 0 或多个字符 + “Helper”
{}	占位符，由 find 命令的输出文件替换
\;	结束符，表明自 -exec 往后的命令结束。因为这是在 UNIX 里，你可能把当前命令的结果通过管道传给另一个命令，所以 find 命令必须知道 “exec” 什么时候结束

哇！这里面的东西可不少，而且还是这么一种紧凑的语法！当然，这就是 UNIX 的设计哲学。下面是另一个例子，用来展示 UNIX 命令行有多种方式来做一件事情。其中采用了 xargs 命令，完成同样的功能。

```
$ find . -name *Helper.java | xargs grep -l "extends .*Helper"
./src/java/org/sample/logic/DocumentHelper.java
./src/java/org/sample/logic/LoginHelper.java
```

这个例子大部分与上一个相同，只是采用了管道来把 *find* 命令的输出转发给 *xargs* 命令。*xargs* 是一个辅助工具，它可以把来自管道中的输入放置到它的参数串的末尾，作为一整条命令执行。就像上面的 {} 占位符一样，*xargs* 拿到 *find* 命令输出的文件名，传给 *grep* 作为最后一个参数。你还可以使用 *xargs* 的不同选项来控制 *xargs* 将管道中的输入放置到参数串的不同位置。例如，下面的命令将所有以大写字母开头的文件复制到 *dest* 目录：

```
$ ls -1d [A-Z]* | xargs -J % cp -rp % destdir
```

-J 标志告诉 *xargs* 使用 % 作为占位符，代表来自管道的输入。你可以使用任何字符作为占位符，只要它不与目标命令要求的字符冲突。

这个是 find 命令的最后一个版本：

```
$ grep -l "extends .*Helper" `find . -name *Helper.java`
./src/java/org/sample/logic/DocumentHelper.java
./src/java/org/sample/logic/LoginHelper.java
```

注意上面的反向单引号(`)。我们不是用管道来把一个命令的输出传给另一个命令——那么做的话，*grep*只能看到文件名列表，而不是文件内容。用反向单引号（在U.S.键盘的左上部分）把*find*命令括起来，*find*命令就会首先执行，然后其输出结果被传送给*grep*作为待搜索文件列表（而不是被当作待搜索的文本内容）。

现在，我把上面所有东西串起来：

```
$ grep -l "extends .*Helper" `find . -name *Helper.java` | wc -l
2
```

虽然每个命令本身的功能都很细小，但是通过管道和反向单引号，我可以组合出任意我想要的功能，甚至是它们的原作者都没想到过的功能（这就是UNIX背后主要的设计哲学）。了解一打左右的命令，你也可以表现得像一个UNIX资深用户了。更重要的是，你可以随心所欲地摆弄自己的项目，不必局限于IDE菜单上提供的功能。

需要时如何获得帮助

大部分情况下，你可以直接从命令本身获得帮助。但那可能比较难懂，有时也可能找不到。基本上，类UNIX的系统都提供了两种帮助方式。第一种是直接从命令自身获得，多数情况下你可以使用*--help*（或*-h*）选项，就像这样：

```
$ ls --help
```

第二种是通过大部分类UNIX系统都包含的帮助系统："手册页"（manpages）。你可以使用这样的命令来访问：*man <你想查询的命令>*。

```
$ man wget
```

第三种方式是*info*命令，它类似*man*，在大部分Linux上都有。你可以像*man*一样传入要查询的命令来调用*info*，也可以不带任何参数启动它来浏览所有帮助主题，这和*man*不一样。

遗憾的是，对大部分内建的帮助系统，你至少得先知道自己想要查询哪个命令的帮助。但UNIX命令通常名字晦涩，很多时候你根本不知道自己到底要查什么。比如，你不会想，"我需要知道如何找出哪些文件的内容里引用了我的主目录。我知道我只需键入man grep!"对于这种情况，没有什么比入门指南、指导手册或命令行参考手册之类更好的了。