

揭示高效程序员的思考模式

卓有成效的程序员

精选版



Neal Ford 著

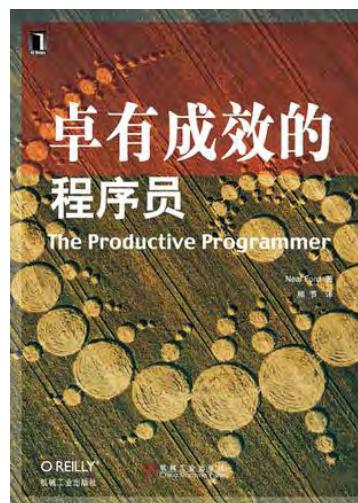
ThoughtWorks (中国) 公司 译

InfoQ企业软件开发丛书

免费在线版本

(非印刷免费在线版)

[登录China-Pub网站购买此书完整版](#)



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品

InfoQ 中文站
www.infoq.com/cn

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本书主页为

<http://infoq.com/cn/minibooks/productive-programmer>

注：封面图片选自 <http://www.flickr.com/photos/kevinkemmerer/3000910963/>，此图片为 CC 授权



QCon 全球企业开发大会(QCon Enterprise Software Development Conference)是由 C4Media 媒体集团 InfoQ 网站主办的全球顶级技术盛会，每年在伦敦和旧金山召开。自 2007 年 3 月份在伦敦召开首次举办以来，已经有包括金融、电信、互联网、航空航天等领域的近万名架构师、项目经理、团队领导者和高级开发人员参加过 QCon 大会。

2009 年，这一在全球企业开发领域享有盛名的大会将首次来到亚洲，来到北京和东京。[QCon 北京大会将在 2009.4.7~4.9 在清华科技园国际会议中心举行](#)。秉承 QCon 伦敦、QCon 旧金山的高品质特性，QCon 北京大会将不仅是一次顶级技术盛宴，还是一次众星云集的大会！

QCon 北京大会将围绕着企业软件开发领域的几大热点主题展开，其中包括：Java、敏捷、云计算、架构、互联网应用等。大会邀请到了迄今为止、国内软件开发领域最大阵容的国际讲师团，其中包括，Spring 创始人 Rod Johnson、Agile 领域伟人和 ThoughtWorks 首席科学家 Martin Fowler、全球知名电子商务网站 eBay 的架构师 Randy Shoup、技术领先公司 Amazon 的云计算负责人 Jeff Bar 等。此外，结合本地实践与案例，大会还邀请了国内众多知名专家与讲师，其中包括，Google、IBM、西门子、盛大、淘宝网、腾讯、搜狐、豆瓣等国内公司有多年经验的架构师或技术负责人，以及在技术领域享有盛名的专家学者。

相关网站：

[QCon Beijing](#)

[QCon Global](#)

[QCon News](#)

目录

序	1
前言	2
本书的目标读者	3
字体约定	3
使用代码示例	4
如何联系我们	4
致谢	5
概述	6
为什么要写一本关于程序员生产率的书	7
本书涵盖的内容	9
如何读此书	11
自动化法则	12
不要重新发明轮子	14
自动访问网站	15
与 RSS 源交互	16
在构建之外使用 Ant	17
用 Rake 执行常见任务	18
用 Selenium 浏览页面	20
用 Bash 统计异常数	20
用 Mac OS X 的 Automator 来删除过时的下载文件	23
驯服 Subversion 命令行	25
用 Ruby 编写 SQL 执行工具	26
我应该把它自动化吗	27
别给牦牛剪毛	29
小结	29
古代哲人	31
亚里斯多德的“事物的本质性质和附属性质”理论	32
笛米特法则	37
“古老的”软件学说	38
多语言编程	40
历史与现状	41
路在何方	44
Ola 的金字塔	49

序

在我们这个行业里，不同程序员的个人生产效率可谓判若云泥——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学，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效率提升归功于使用 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^{*} 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，我真的希望早 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



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）。这些学生之间生产效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）。顺理成章地，这自然让我开始反思自己的生产效率：我有没有让手边的（正在跑或走的）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、关于如何提高生产效率的书，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。

字体约定

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。比如说，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(100080)
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>

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。在漫长写作过程中，我身边的亲支帮了我大忙，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” *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。



概述

工作效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力对整个团队的影响。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力，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工作效率的东西（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力。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。第一部分讨论生产力的机制，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力的实践，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（生产力法则）

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力，你就拥有了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力法则，来帮助你更好地定义自己的提高生产力的技术。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力，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、提高工作效率的办法。

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力的方法。我的初衷是致力于做一个世界上最好的提高生产力的方法大全。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力的方法时，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力的“法则”——

一坦白地说，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，来达到超级生产力的状态。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力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。

如何读此书

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。



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。

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

注1：

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
<i>wget</i>	启动 <i>wget</i> 工具
--mirror	给网站建立本地镜像。 <i>wget</i> 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 <i>wget</i> ，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。 <i>wget</i> 会转换页面上所有的链接，使其指向本地资源
-P c:\wget_files\example1	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。

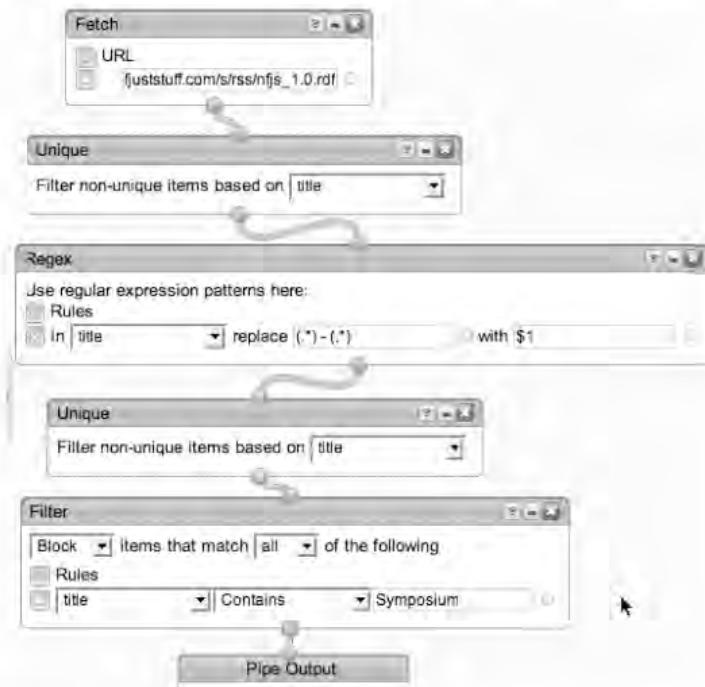


图 4-1：实用正则表达式管道

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，以至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__)
FILES = [
    "#{PRESENTATIONS}/building_dsls.key",

```

注 2：

```

    "#{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}/dss/",
    "#{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

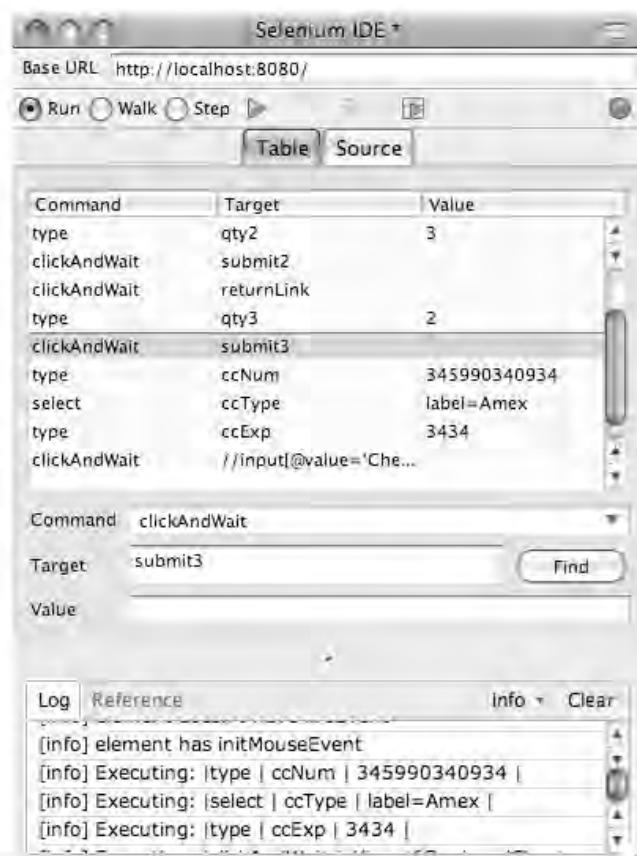


图 4-2：Selenium IDE 和录制好的脚本

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
<code>egrep -o</code>	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
<code>"[A-Z]\w*Exception"</code>	用于定位异常信息的正则模式
<code>log_week.txt</code>	庞大的日志文件
<code> sort</code>	将前面的查找结果管道给 <code>sort</code> ，生成一个排序后的异常列表
<code> uniq</code>	去掉重复的异常信息
<code>for x in \$(. . .) ;</code>	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
<code>echo -n -e "processing \$x\t"</code>	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)
<code>grep -c "\$x" log_week.txt</code>	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。

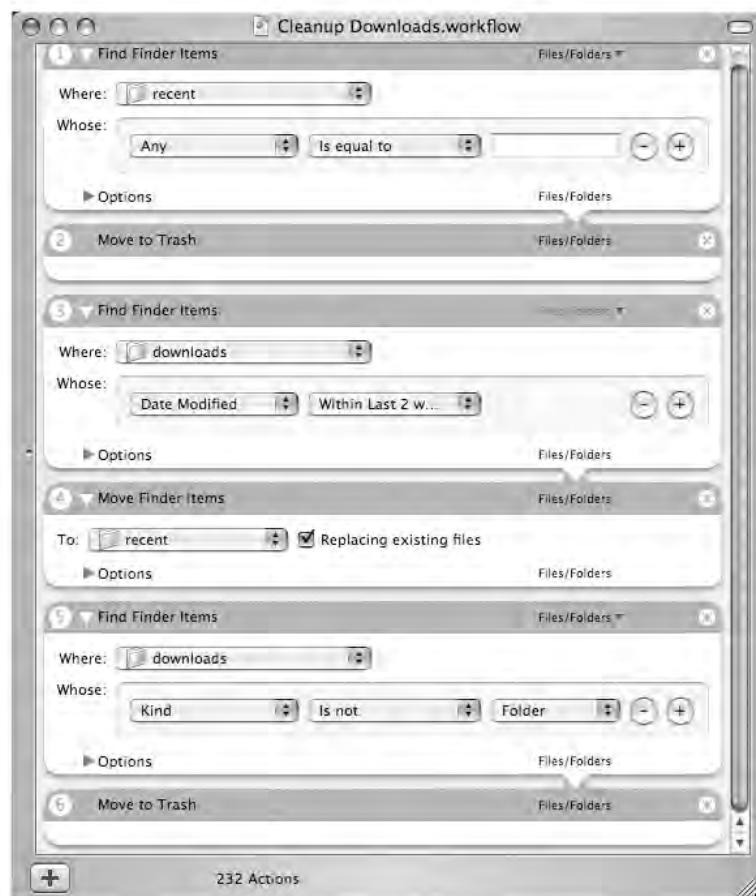


图 4-3：用于删除过时下载文件的 Mac OS X Automator 工作流

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方法能描述“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，但如果我要添加 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
<code>svn st</code>	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
<code>grep '^?*</code>	找出所有以“?”开头的行
<code>tr '^?' ' '</code>	把“?”替换成空格（tr 命令会把一个字符替换成另一个字符）
<code>sed 's/[]*//'</code>	用 sed（基于流的编辑器）把每行开头的空格去掉
<code>sed 's/[]/\ \ /g'</code>	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）
<code>xargs svn add</code>	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 1000 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力之源。

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 \times 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这样做）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？

最后一个问题是之所以重要，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。我们都曾有过这样的经验：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。



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。让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 xxxx(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?①
    assert ! "x" .blank?
  end
end
```

- ① 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.setPosition(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。



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“在浏览器里跑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运行“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

答案是那些习惯于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。）

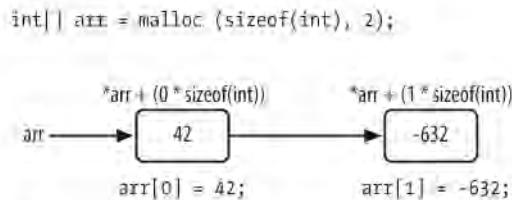


图 14-1：C 的数组偏移量

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）。

令人都闷的是，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（polyglot 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，JavaScript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！”这是绝对的正确：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？程序员的工作效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糕的，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: ? ? ? ? ?

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。mixin 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 Programming, 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推向领域特定语言 (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。



全球企业开发大会（北京）2009.4.7~2009.4.9



QCon 全球企业开发大会（QCon Enterprise Software Development Conference）是由 C4Media 媒体集团 InfoQ 网站主办的全球顶级技术盛会，每年在伦敦和旧金山召开。自 2007 年 3 月份在伦敦召开首次举办以来，已经有包括金融、电信、互联网、航空航天等领域的近万名架构师、项目经理、团队领导者和高级开发人员参加过 QCon 大会。

2009 年，这一在全球企业开发领域享有盛名的大会将首次来到亚洲，来到北京和东京。QCon 北京大会将在 2009.4.7~2009.4.9 在清华科技园国际会议中心举行。秉承 QCon 伦敦、QCon 旧金山的高品质特性，QCon 北京大会将不仅是一次顶级技术盛宴，还是一次众星云集的大会！

六大技术主题：

Java: 企业级 Java 开发 — 毛新生主持

Agile: 敏捷——在路上 — 李剑主持

Cloud Computing: 云计算 — 王翔主持

Case Studies: 网站架构案例分析 — 冯大辉主持

Architecture: 设计优良的架构 — 周爱民主持

RIA: 炫富互联网应用 — 吕德维主持

详细介绍：

<http://www.qconbeijing.com/tracks.shtml>

部分演讲嘉宾：

Rod Johns — Spring 创始人

Martin Fowler — 敏捷宣言缔造者

Randy Shoup — eBay 高级架构师

Jeff Barr — Amazon 公司云计算战略师

Dylan Shiemann — Dojo Toolkit 创始人

Henrik Kniberg — 《硝烟中的Scrum和XP》作者

更多嘉宾：

<http://www.qconbeijing.com/speakers.aspx>