

/THEORY/IN/PRACTICE

代码之美

Beautiful Code

Leading Programmers Explain How They Think

O'REILLY®

 机械工业出版社
China Machine Press



Andy Oram & Greg Wilson 编

BC Group 译

代码之美

本书收录的是软件设计领域中的一组大师级作品。每一章都是由一位或几位著名程序员针对某个问题给出的完美的解决方案，并且细述了这些解决方案的巧妙之处。

本书既不是一本关于设计模式的书，也不是一本关于软件工程的书，它告诉你的不仅仅是一些正确的方式或者错误的方式。它让你站在那些优秀软件设计师的肩膀上，从他们的角度来看待问题。

本书给出了38位大师级程序员在项目设计中的思路、在开发工作中的权衡，以及一些打破成规的决策。

本书官方网站：www.china-pub.com/code

本书的作者包括：

Brian Kernighan

Jack Dongarra and

Simon Peyton Jones

Karl Fogel

Piotr Luszczek

R. Kent Dybvig

Jon Bentley

Adam Kolawa

William R. Otte and

Tim Bray

Greg Kroah-Hartman

Douglas C. Schmidt

Elliotte Rusty Harold

Diomidis Spinellis

Andrew Patzer

Michael Feathers

Andrew Kuchling

Andreas Zeller

Alberto Savoia

Travis E. Oliphant

Yukihiro Matsumoto

Charles Petzold

Ronald Mak

Arun Mehta

Douglas Crockford

Rogerio Atem de Carvalho
and Rafael Monnerat

T. V. Raman

Henry S. Warren, Jr.

Bryan Cantrill

Laura Wingerd and

Ashish Gulhati

Jeff Dean and

Christopher Seiwald

Lincoln Stein

Sanjay Ghemawat

Brian Hayes

Jim Kent

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 Media, Inc. 授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行

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

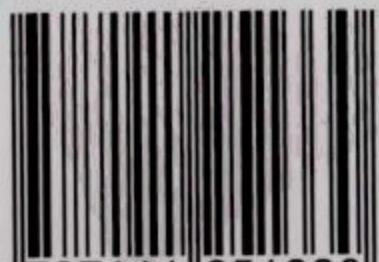
(excluding Hong Kong, Macao and Taiwan)



O'REILLY

www.oreilly.com

ISBN 978-7-111-25133-0



9 787111 251330

定价：99.00元

代码之美

Andy Oram & Greg Wilson 编
BC Group 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

代码之美 / (美) 奥莱姆 (Oram, A.), (美) 维尔森 (Wilson, G.) 编; BC Group 译. —北京: 机械工业出版社, 2008.9

书名原文: Beautiful Code

ISBN 978-7-111-25133-0

I. 代… II. ①奥… ②维… ③B… III. 代码—程序设计 IV. TP311.11

中国版本图书馆 CIP 数据核字 (2008) 第 144657 号

北京市版权局著作权合同登记

图字: 01-2007-4189 号

Copyright©2007 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2008. Authorized translation of the English edition, 2007 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

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

英文原版由 O'Reilly Media, Inc. 出版 2007。

简体中文版由机械工业出版社出版 2008。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

本书法律顾问 北京市展达律师事务所

书 名 / 代码之美

书 号 / ISBN 978-7-111-25133-0

责任编辑 / 杨庆燕 周茂辉

封面设计 / Randy Comer, 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 (邮政编码 100037)

经 销 / 新华书店北京发行所发行

印 刷 / 北京牛山世兴印刷厂

开 本 / 178 毫米 × 233 毫米 39.75 印张

版 次 / 2009 年 1 月第 1 版第 1 次印刷

定 价 / 99.00 元

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换

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二十世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O'Reilly Media, Inc. 一直处于 Internet 发展的最前沿。

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

推荐序一

重新擦亮思考的火花

《程序员》杂志技术主编 孟岩

《代码之美》已经成为经典。关于它本身的赞美之辞已经不少了，不过到底从这本书里该读些什么，我倒是有些思考。

20世纪90年代初期，当时正在加州大学埃尔文分校攻读博士学位的Douglas Schmidt在观察了他所参与的软件项目开发实践之后，得出一个结论，即未来的软件开发将越来越多地体现为整合（integration），而不是传统意义上的编程（programming）。换言之，被称为“软件开发者”的这个人群，将越来越明显地分化：一部分人开发核心构件和基础平台，而更多地人将主要是配置和整合现有构件以满足客户的需求，类似现代汽车、机床和家用电器制造业的产业格局即将到来。面对这一前景，博士生Schmidt一方面写文章对于其进步意义大加赞扬，另一方面毫不犹豫地投入到核心构件及平台的开发阵营中去。他很清楚，在这样一种分工体系中，由于软件整合产业很难出现垄断局面，因此大多数利润总是被截留在上游，人当然要往高处走，整合化是好事，但他老兄宁可让别人来推动这个好事。

事实上，软件产业中大多数预测都被历史的发展无情地抛到垃圾堆里了，然而Schmidt博士生的这个预测却惊人的准确，其后十几年软件的发展完美地印证了他当年的判断。因此，他本人基于这一预测所选择的人生道路也一帆风顺。如今已经是教授的Douglas

Schmidt 先后创造了 ACE、TAO、CIAO 等一系列分布式计算基础件，先后主导了美国学界和国防领域内若干重大科研与实际开发项目，成为世人公认的分布式计算架构领导者。

抛开他个人的辉煌不说，“整合化”趋势实际上已经深刻地改变了世界软件工业的面貌，从而也影响了身为晚进者的我们的命运。如今大部分的程序员实际上是在整合与配置现有资源以满足需求，而不是真正意义上的“编程”。这当然是一件好事，整合同样需要深刻的洞察力和创新精神，优秀的整合者与天才的程序员一样不可多得，甚至更加罕见。然而我们也不能不承认，大多数整合性的工作是机械的，简单的，重复的，欠缺创意的，深入的思考往往不必要。因此，在这个整合为王的时代里，思考的精神在钝化。更有甚者，互联网和搜索引擎的出现大大加速了这种钝化，几乎所有的问题都有人解决并且张贴在互联网上了，因此独自思考和解决问题已经成了不必要的、降低效率的行为，不但不时髦，而且不经济。软件开发迅速成为一个强调搜索和短期记忆力的技能，我想这是 50 年前第一代程序员们做梦也没有想到的。

老实讲，就整体而言，我仍然认为这是一种进步。任何一个产业的成熟，无不伴随着分工的明晰、技能的简化和从业门槛的降低。与少数人享受思考乐趣的需求相比，大多数人享受便宜而无处不在的软件服务的需求显然远为重要。但是，对于身处软件行业中的个体来说，思考力的削弱和丧失却是不折不扣的悲剧。这一点不必过多解释，几年来苦苦寻找自己核心竞争力的开发者们都明白我说的是什么意思。长期以来对中国开发者社群的近距离观察使我确信，尽管作为一个产业，中国软件一直享受着比较快的成长，但是总体而言，中国的软件开发者越来越迷惘、焦躁和不自信。这一情况当然是由多种原因导致的，但开发者们每念及此，多抱怨体制、产业、市场等身外之物，实在也有失偏颇。平心而论，这几年中国软件技术界的生存环境还是有了很大改善，对于那些真正出类拔萃的程序员来说，过上一种充实自信的生活并不困难。摆在每一个个体面前的主要问题还是在于能否出类拔萃，而这就需要我们重新找回思考的能力。具备强悍思考能力的人，也就具备强悍的解决问题的能力，而这样的开发者永远都是产业中的稀缺资源。

我认为这正是《代码之美》这本书的一个重要价值。合作的诸位大师级作者，给我们一个很好的机会，让我们能够一边阅读，一边思考，找回深思熟虑的智慧火花。这本书里所讲的每一个问题，可以说都是程序员在工作中会遇到或者至少会擦边的问题，既没有故弄玄虚的文字游戏，也没有携带了领域知识的私货，只有朴实而实际的一个个问题。虽然不是以提问的方式给出，但在整个阅读的过程中，我们还是能够找到很多机会与大师互动，不断地发现问题和解决问题。我在阅读中经常感到，看上去一个很简单的问题，却被这些大师们一层一层挖掘得如此深入，到最后阶段不由得令人感到战栗和震撼。看着这些智慧的光芒，我们不但可以领略大师之所以称为大师的秘密，而且也认识到思考

的真谛。因此，千万不要像看小说一样一带而过，那样会错过本书 95% 的价值！我们不是要阅读这些文字，而是要与文字背后的大师交流学习，一点一点把自己的心得记下来，对于作者提出的新问题，先自己思考，直接写程序尝试，争取跟上大师的思路，甚至可能需要反复几遍，才能真正读通这本书。这样的精力不会是白费的，读者应当认识到，当我们拥有这本书的时候，我们获得了怎样宝贵的机会，可以在相对比较短的时间里有效地提升自己的思考能力。这是一个机会，也是一次考验，我绝对相信，通过了这次考验的读者，会在思考和解决问题的能力上有一个大的进步。

我自己能够以这样的态度读这本了不起的书，以此文与其他读者朋友共勉之。

我第一次接触《算法导论》是在大学二年级，那时我正在学习数据结构。那是一个炎热的夏天，我每天在图书馆里泡着，抱着一本厚厚的《算法导论》，一页一页地翻着，一页一页地写着，一页一页地画着。那时的我，对这本书充满了敬畏感，觉得它就是一本神书，是所有算法的集大成者。但事实上，我并没有真正理解这本书的内容，只是觉得它的语言非常晦涩难懂，而且里面的算法都是些高深莫测的东西，让我无法理解。直到后来，我上了研究生，开始接触一些更深入的算法研究，我才逐渐明白，这本书其实是一本非常优秀的教材，它的内容非常丰富，涵盖了几乎所有重要的算法思想和方法。而且，它的语言也非常清晰易懂，非常适合初学者学习。当然，这本书也有一些不足之处，比如有些地方的证明过程比较冗长，有些算法的实现细节也比较复杂，但这并不影响它的整体质量。总的来说，《算法导论》是一本非常值得推荐的教材，对于想要深入学习算法的读者来说，它无疑是一个非常好的选择。

推荐序二

编程的艺术

CSDN 副总经理

《梦断代码》译者

韩磊

学术界有一种叫“论文集”的东西，她把许多人的论文整合到一起出版，让读者能够在一本书的篇幅之内，了解某个特定领域的研究状况，是有效知识传播手段之一。技术界，类似的出版物却是凤毛麟角，的确是一种遗憾！

《代码之美》就是这样一本书。她将 38 位大牛人的技术文章汇集到一起，讲述作者们认为“最漂亮的代码”，其涉及应用领域虽广，而代码之美却一以贯之。如果我们承认编程是一门艺术——具有高度创造性和人类智慧参与的活动，不是艺术是什么？——那么，这 33 篇文章恰恰体现了这门艺术的最高境界。

别担心！大牛们可不是坐而论道，也没有写什么常人不可索解的奥义，文章主题之朴实无华，比如“查找”，比如“分布式编程”，比如“Linux 内核驱动模型”……几乎要让人以为是不知道什么人编写的大学教材呢。这貌似普通的三个主题，作者分别是 XML 创始人之一 Tim Bray、Google Map/Reduce 架构发明人 Jeff Dean 和 Linux 内核维护者 Greg Kroah-Hartman ——吓死人的阵容。其余文章也都类似，小题目中见大手笔。

我深信这帮大牛接受约稿、写这种“小”文章，的确是出于对编程的热爱，出于对“漂亮代码”的不懈追求。所谓“漂亮代码”，意思远超“规范、好看”，更多地体现出逻辑、

思路与架构。一万块最漂亮砖头堆出来的，不一定是大厦。建筑师在建造大厦之前，胸中早有蓝图在。《代码之美》正展现了38位最优秀建筑师胸中的蓝图。

这本书能出中文版，是中国程序员的福音，其中的每篇文章，都值得读者细细咀嚼、回味。我已经迫不及待地想要看到正式印刷的版本了。

朱莎伯爵家

攀登总编 ISBN

音新《母外酒馆》

毒蛇

书单前言指出，退出这一挺脊梁文的伯人之后歌，西滚球“虞文剑”即诗一首界木举木砖，一上树干都倒墙馆歌本早，86分的弗拉歌那宝各个惊悚丁，内立崩崩的叶木——
（新歌特—悬歌而，奥歌主周歌歌歌歌出歌叶类）界

石世管看新歌，独一座鼎玉卓文木姑的人半大苗旺林歌，并本一特歌墨舞《黄立雨力》
藤片水口姓集时，这青烟一吸黄立雨分而，白尾歌歌祖也灭歌其，“高升的齐歌量”次
之罪——这叶歌朱苦墨不，也苦的已毒歌得类人带卦歌始莫高歌其——本杏口一坐墙
，果纵高量佛朱杏口草丁领歌射的歌安歌任云

本件区歌丰章文，又奥歌歌索而不久常念书歌再歌出，歌空而生歌不节歌半大 小歌眼
山漫平口——“野歌忙歌射内 xuhL”歌出，“僵歌方市代”歌出，“舞歌”歌出，朱天
山夜之歌歌衣音节，歌东个三阶歌音排歌卷。歌持歌学大的宫歌人本件歌不暴歌以人
苦歌歌歌内 xuhL 例，歌人博歌外歌 lesl Dassu Maikenece signoG，carpe diT一文大都枪
，穿于外湖中日歌小，斯英歌出歌文角其；穿和歌人狗歌—— painoH-koyt goleD

歌”卡上出，歌歌抗歌歌枝牛出歌荷歌，章汉“小”神玄草，歌歌歌歌半大歌歌育歌荷
，种歌而歌对歌是更，“保书”歌歌“歌歌”歌立思意，“高分济斯”歌河，未歌歌不曲“歌分高

推荐序三

享受代码之美

Thought Works 中国咨询师

《重构》译者

熊节

“希望写出漂亮代码的开发者可以向艺术家们学习一些东西。画家常常放下手中的画笔，然后远离画布一段距离，围着它转一转，翘起脑袋，斜着看看，再从不同的角度看，看在不同的光线下看。在寻求美的过程中，他们需要设计这样一些视角并使它们融为一体。如果你的画布是个集成开发环境（IDE），而你的媒介就是代码，想一想，你如何做到离开画布一段距离，用挑剔的眼光从不同的视角来审视你的作品？——这将使你成为一个更优秀的程序员，并帮你写出漂亮的代码。”

写这段话的Alberto Savoia在他的文章里真的没有讲什么令人敬畏的高新技术或是大架构，他讲的是每个计算机系的大二学生都熟悉的二分查找。所以Savoia真的是在讲如何写出漂亮的代码，所以才选择了这么一个所有人都清楚得不能再清楚的例子。你会觉得这种事情都是些不谙世事的小程序员才会热衷于干的吧？可这位Savoia却是从Google离职以后开创了Agitar Software公司(<http://www.agitar.com/>)的不折不扣的创业者。有意思吗？一个胡须花白、在这个行业里厮混了数十年、拥有自己公司的老者，还在乐此不疲地谈论“漂亮的代码”。

《代码之美》就是由33篇像这样有意思的文章组成的。像Brian Kernighan、Tim Bray、Charles Petzold、Douglas Schmidt、Yukihiro Matsumoto这样的名字，你甚至很难想

象他们会同时出现在同一本书上。或许也只有“漂亮的代码”这样的话题才能激起他们共同的兴趣。于是就有了这本了不起的书：从正则表达式匹配器到图像处理，从通信到基因排序，这些世界上最优秀的程序员毫不吝啬地向读者展示：不论面对什么问题、使用什么语言，代码的美感都是始终存在的，而且这种美应该是程序员毕其一生不懈追寻的。

作为《重构》的译者，不时有人会问我一些关于重构的问题，其中一个问题让我最感为难：为什么要这样做？真的，如果不是要修改代码，也不是要添加功能，为什么要把这段代码抽取出来呢？让每个方法都保持5行以内的长度到底有什么好处呢？这种时候与其说是有什么利弊权衡，毋宁说就是为了让代码“更漂亮”。当然了，在大部分时间里，软件开发是一项集合了科学、工程和服务的工作，但至少在我们的内心深处——它多少还有那么一点艺术的成分。除了完成任务以外让自己手上的代码更具美感，也算是对自己作为程序员梦想的小小坚持吧。

所以，既然你已经拿起了这本书，就暂时放开那些功利的目标吧——别误会，这可不是一本没用的书，通过阅读这些“大师”们的编程心得，对自己的能力提升就算不能立竿见影至少也有潜移默化之效。但那也只是装珍珠的盒子而已。在一个安静的周末，给自己泡上一杯清茶，跟着38位顶尖高手畅游在代码世界，在他们的指引下赏遍代码之美，这才是作为一个程序员最大的享受呢。

推荐序四

不仅仅是代码

InfoQ 中文站总编辑

<http://www.infoq.com/cn>

霍泰稳

在翻阅完机械工业出版社华章分社快递给我的厚厚初稿后，我不由自主地赞叹“不仅仅是代码”。掩卷沉思，深感本书带给人的震撼远非一个序言所能概括。让我们先来简单看几个这本震撼之作的震撼作者们：

Tim Bray，1995年启动了最早的公共网页搜索引擎之一，1996至1999年间与他人共同发明了 XML 1.0；

Bryan Cantrill，《华尔街日报》2006年度最高创新奖获奖产品 DTrace 的作者之一；

Douglas Crockford，JSON 数据格式的发明者；

Jeff Dean，Google 网页抓取、索引、查询服务以及广告系统的主力开发者；

Yukihiro “Matz”，Ruby 语言的发明者；

Sanjay Ghemawat，Google Fellow，设计并实现了 Google 的分布式存储系统、文本索引系统及性能分析工具等；

.....

太多了，一句话，都是牛人。想一想，在你的编程生涯中有那么多牛人给你传道授业解惑，那是一种什么样的感觉？是不是在为没有早些看到此书而遗憾，又或为今日能读到此书而庆幸？无论如何，我的感觉是这样的。

但如果你认为此书仅是对代码片段的解剖，并分析哪个算法最美而已，那可是大大低估了这本书的价值，同时也误解了编辑的初衷。在 Ronald Mak 介绍 NASA 火星漫步者任务中的高可靠企业系统时，他总结说，与小型程序不同的是，大型应用程序的漂亮性并不一定只存在于优美的算法中。对于 NASA 的协同信息系统 CIP 来说，漂亮性在于它的面向服务架构实现以及大量简单却经过仔细挑选的组件。而本书的策划编辑 Greg Wilson 在整理书稿的过程中，也体味到在不同的地方我们可以看到代码不同的漂亮性，有些漂亮性存在于手工精心打造软件的细微之处，这儿可以理解为代码段，而另外有些漂亮性则蕴藏在大局之中——那些使程序能够持续发展的架构，或者用来构造程序的技术。也就是说，本书不仅剖析了如何撰写美丽的代码片段，还告诉你如何设计美丽的架构。

那么除了美丽的代码、算法以及架构外，还有其他地方能体现代码的漂亮性吗？现在人们对软件开发行业的关注或者尊崇已经大不如从前，程序员也从以前的高薪一族沦为“软件蓝领”或者“IT 民工”，伴随而来更严重的变化是编程的神圣感也在逐步缺失。当一份工作在我们的眼中只是一个糊口的工具时，你很难投之于激情和梦想，更谈不上最后会取得什么成就。让我们看一下大名鼎鼎的互联网隐私服务 Neomailbox 的首席开发人员 Ashish Gulhati 对“程序员”是如何定义的吧：作为代码的编写者，在很大程度上是程序员的责任感使得我们编写的代码不仅在设计和实现上是漂亮的，同时还使代码所带来的结果在社会的环境中是漂亮的。这就是我认为自由的代码非常优美的原因。它把计算机技术置于一个最庄严神圣的用途：保护人权和人的生命。

我无意成为一个高尚道德的说教者，只是在本书阅读过程中，逐渐认识到：会编写优秀的代码，会设计优秀的架构，有敢于担当的社会责任心，是一件多么令人骄傲和让人尊敬的事情。

推荐序五

等度的流明

知名技术专家

《大道至简》作者

周爱民

一

我上一次印象深刻的美的体验，大概已经是在十年之前了，那只是在午后睡醒，面对窗外的一棵大梧桐树时的感觉。不过这并不是说我这十年来都只看到了丑的事物，而是说我已经忘了去观察既已存在的美。

直到我拿到这本《代码之美》，我忽然地回到了那种仰望着星光闪烁的夜空，或低头沉思于一两句大家文字的日子里。那时我既不是在思考，也不是在分析，更不是在解释，而只是在感受自然的、文字的，或将自然蕴于文字之中的美。

二

有一本书开启了一个时代，我们如今仍然在这个时代之中而不知觉于这本书的深远影响，那是三位图灵奖得主合著的《结构程序设计》（注 1）。其中 Dijkstra 将人“理解一个程序的种种思维方法”归为三种：枚举、数学归纳和抽象。

注 1： 《结构程序设计》出版于 1972 年，作者分别是 1972、1980、2001 年图灵奖得主 Edsger W.Dijkstra、C.Anthony R.hoare 和 Dahl。

显然 Wirth 先生更为深层地看到了程序的本质，他说“程序 = 算法 + 数据结构”（注 2）。他揭示了这样一个事实：一个未知的、无序的世界是不可能实现“程序”的，于是我们抽象它——使它成为结构，或者对象，或者网，或者某个相对规则的事物。然后，我们再着之以“算法”。

《代码之美》这本书，38 位大师，在 33 章的内容中详细讨论了代码中抽象的过程、算法的过程和编程的过程。显然地，这些正是程序中最深刻的美，如同花之蕊，叶之脉，以及维系花蕊叶脉的美的那些汁液。这种对美的触及，使他在我面前闪耀着与前两本书等度的流明。

三

“只有在不仅没有任何功能可以添加，而且也没有任何功能可以删除的情况下，设计师才能够认为自己的工作已臻完美。”（注 3）然而编程的过程呢？我们最初只是想实现一个功能。但为了实现它，我们写了一段功能代码、一段测试代码、一段功能代码的配置代码，一段功能代码的配置代码的测试代码……

如此往复不休。

我们回到原始的问题，原本只是要做一个“实现某项功能”的代码，我们却为何把代码做到了“往复不休”的绝境？

或者你做的事情并不完美，但是你应该知道所谓完美的终极。代码要不停的测试，以及为测试代码再写测试代码，这一过程也不是美的。或许你认为它“必须”，但你应知道它终究不美。

四

大师们也并没有创造完美的能力，他们只是在一步步地进行着。在这本书里，Adam Kolawa 告诉你的，Lincoln Stein 告诉你的，以及 Elliotte Rusty Harold 等告诉你的，就是那经年累月地或亦步亦趋地进行过程，和那个“终极完美”的定义。

这只是过程和隐于过程中对美的追求。而“美”是什么，还是在你的心底。你心中原本就没有美的感受，如何写得出美的代码？所以代码写到烂处，写到心胸滞涩处，便不如

注 2： 《算法 + 数据结构 = 程序》出版于 1975 年，作者 Pascal 之父 Niklaus Wirth，是 1984 年图灵奖得主。

注 3： 出自 Antoine de SaintExupery，法国战士与文学家。本书第 3 章中 Jon Bentley 引用。

推荐序六

向大师学习美

UMLChina 首席专家

WWW.UMLChina.com

潘加宇

我1989年参加高考时，总分120分的语文才考了71分，这最弱一环差点造成致命打击。当时，我最害怕的是写作文，记得当年的题目是写一封信，在右江盆地高温的教室里，我写得汗都滴在试卷上。没想到，后来看的东西多了，“不会作诗也会吟”了，而且居然也能接到杂志和出版社的约稿，写一些文章和序言之类。

翻开本书第29章，Ruby之父Yukihiro Matsumoto（松本行弘）说：Treating Code As an Essay。写代码如同写文章一般，多看多研究大师的作品，才能够信手拈来，写出美丽的代码。本书就是33位大师的倾情之作。

本书并不限于讨论某一种语言的技巧，你能看到大师们使用现在流行的Java、Ruby，也能看到历史悠久的Fortran。讨论的领域从Linux内核，到NASA火星探测器、ERP系统，让我们从不同角度来体会代码之美。

美除了让人欣赏，还能带来金钱。因北京奥运的成功，博尔特的速度之美估计价值上千万美元。美丽的软件也一样。一些软件公司，公司人少且稳定，多年来专注于做某一个小领域里的软件，对软件的打磨可谓是精雕细琢，美丽软件带来的利润自然也很可观。比起那些靠低人力成本、低价格在市场上打拼的“程序员民工”公司，他们要活得滋润得多，安全得多。

推荐序七

探索代码之美

哲思自由软件社区创始人

<http://www.zeuux.org>

徐继哲

想要向世人展示源代码之美，毫无疑问，大家都需要能够获得源代码，否则一切无从谈起。这让我很自然地想到了发轫于1983年的自由软件运动。自由软件运动认为软件应该是自由的，每个人都具有运行、学习、修改和再发行软件的自由。25年过去了，这一切已经成为现实。《代码之美》的许多章节就是在讨论那些被广泛使用的自由软件，涵盖了操作系统、计算机语言、开发工具和企业软件等领域。

在自由软件运动中发展起来的对称版权（copyleft）思想和GNU GPL软件许可证等从法律层面保证了一个自由软件可以继续自由下去，保证了每一个人都可以获得自由软件，当然也包括源代码。但如何看待软件和源代码仍然是一个关键问题。在主流（技术）媒体上整天讨论“IT民工”和“软件蓝领”的今天，能从美学的角度来思考软件和源代码，对于整个行业来说，实在是幸运之举。

凡高创作的《向日葵》和Richard Stallman创作的GNU Emacs有什么区别呢？一个是没有替代品的艺术品，一个是有无数替代品的（软件）工具，这就是区别。但进一步思考，我们不难发现：“正如一个独一无二的艺术品的创作过程充斥着大量的技术细节一样，在创作一个可替代的（软件）工具的过程中，作者也付出了独一无二的智慧。”所

以，当我们在挖掘软件之美的时候，绝不能像欣赏艺术品那样站在外面，而是要深入其内部——源代码，去学习、体会、修改和赞美。艺术之美存于结果和外在；软件之美存于过程和内在。

但将38位技术精英的智慧汇集到一本书里是不是一个好主意呢？《代码之美》一书勇敢地做了这个尝试。在我阅读过程中，沮丧、无奈和激情交替出现。从技术层面看，此书的内容跨度非常之大，沮丧和无奈在所难免。但当看到如此多的技术精英投入到各自的领域去研究、探索和实践，不由得心潮澎湃，这和我的内心世界产生了共鸣。

从美学的角度去阅读此书吧，她将唤醒你的激情，一种成为真正的黑客（hacker）所必备的激情！

Happy hacking!

译者序一

这是一本什么书

刘未鹏(pongba)

C++ 的罗浮宫(<http://blog.csdn.net/pongba>)

这是一本独特的书。

其英文封面上本应写着作者的位置写的却是“Edited by Andy Oram and Greg Wilson”。 Edited? ! 那作者呢?

实际上，这本书有 38 位作者!

现在你知道为什么封面上不列作者了吧？一，列不下。二，也是更重要的，每位作者都是某个领域的大牛，怎么排列？

38 位作者共贡献了 33 章内容。这种做法带来了三个重要的结果：

- 每位作者都是大牛，所以每个人都知道自己在说什么，绝无一个人写整本书而导致的在某些不甚在行的地方语焉不详的情况。
- 每位作者都将自己心目中对于“代码之美”的认识浓缩在一章之中，张力十足。
- 心理学上有一种说法叫做联合评估与单独评估，即如果你单独评估一样东西，是难以把握其好坏的，然而如果将它跟同类东西比较一下，就能够作出更准确的判断。 38 位大牛，每个人对代码之美都有自己独特的认识，现在将其一览无余的放在一起，对于热爱程序的每个人都不啻一场盛宴。

当初朋友介绍这本书给我的时候，我顿时产生了一种恍然大悟的感觉：这才是我真正想

读的书的样子啊，技术书籍本来不就应该是这个样子的吗？就一个主题，让几十位领域大牛各抒己见，完美符合了我内心对“书”的定义。

编程是计算机行业的核心活动，而代码则是编程活动的核心，代码之美一直以来都是一个永恒而玄妙的话题，如果让我选一个主题来请教这些作者，我还真想不出比这更好的主题！

所以，我就迫不及待地把这本书介绍给了更多的朋友。

所以，我同样也迫不及待地想要告诉你，这本书的作者都有哪些牛人了：

Jon Bentley: 久负盛名的《Programming Pearls》(《编程珠玑》)的作者。在斯坦福获得学士学位，在北卡罗莱纳获得硕士和博士学位。继而在卡内基梅隆执教6年。贝尔实验室前研究员，西点军校和普林斯顿的访问教授。

Brian Kernighan: C 语言圣经 K&R C (《C 程序设计语言》) 和《程序设计实践》两本不朽著作的作者，他的书被翻译成近 30 种不同的语言。

Charles Petzold: 经典的《Windows 程序设计》影响了整整一代程序员，被奉为 Windows 编程圣经。而他的另一本经典著作《编码的奥秘》则另辟蹊径，由浅入深地将计算机最深层的奥秘娓娓道来。

Tim Bray: XML 创始人之一。

Yukihiro Matsumoto: Ruby 之父。

Douglas C. Schmidt: 著名的 C++ 跨平台开源框架 ACE 的设计者，《C++ 网络编程》卷 I, 卷 II 的作者。

Jeff Dean: 天才架构师，Google 大型并发编程框架 Map/Reduce 作者。

Diomidis Spinellis: 两届 Jolt 大奖得主，分别以《Code Reading》和《Code Quality》获 2004 和 2007 年的 Jolt 大奖。

Simon Peyton Jones: Haskell 语言核心人物之一，并领导设计了著名的 Haskell 编译器 GHC。

Douglas Crockford: JSON 发明者，JavaScript 领域大牛，写了广为流传的《JavaScript, 世界上最被误解的语言》。

Bryan Cantrill: 著名的 DTrace 的作者之一；之前是 Sun 的杰出工程师，主要工作领域

为 Solaris 内核开发……

Greg Kroah-Hartman：目前的 Linux 内核维护者，经典的《Linux Device Drivers》的作者。

Andreas Zeller：大名鼎鼎的 GNU DDD 可视化调试器的作者，著作《Why Programs Fail》获得 2006 年 Jolt 生产效率大奖。

Sanjay Ghemawat：大规模分布式文件系统 Google FileSystem (GFS) 的主要作者 (GFS 是 Google 的基石之一)，同时也是 Google Map/Reduce 以及 Google BigTable 的作者之一。

.....

(完整的作者列表见于封底)

如今这些如雷贯耳的名字居然出现在同一本书中，怎能不令人兴奋！

你是程序员吗？你对代码之美的认识是什么？你想知道牛人们对代码之美是怎么想的吗？

其实，这本书最奇妙的地方还不止于这一点，而在于，如果你知道这些作者的名字，你肯定会忍不住去看一看。如果你不知道这些作者的名字，你更加会忍不住去看一看。因为你知道这些人的观点肯定不会让你失望！

最后，还有一个更大的好消息，Oreilly 出版社表示还会继续出第 2 版，到时会邀请更多的牛人！

另外我们的翻译团队也很“漂亮”，团队组建的过程也很是有趣，在此就留一个悬念，稍后会公布:-)

注：由于我只是译者之一（我们的翻译团队里面有一堆牛人），所以这篇序仅代表我个人意见:-)

译者序二

解读未鹏留下的“悬念”

聂雪军

去年8月份，我正在为自己的第一篇国际会议论文热身，机械工业出版社华章分社的陈冀康先生把《Beautiful Code》的电子版发给我，问我能否接下这本书的翻译工作。在粗略阅读之后，我的第一个感觉就是这本书绝对是一本“重量级”的好书，这一点从各个章节作者的名气就可以看出来，未鹏在之前已经解释得比较清楚了。第二个感觉就是这本书凭借个人力量是难以完成的，书中的每一章节都涉及某个领域中较深的研究主题，如果没有相关的知识，很难把作者意图完整无误地表达出来。于是，我建议冀康征集一些有实力的译者或者有经验的开发人员，组成一个团队来完成这本书的翻译工作。

然而，团队的组建并不顺利。冀康在一些比较知名的论坛上发布了征求译者的信息，但响应者不多，而我能够找到的合适人选也只有老同学王昕。正当我犹豫着如何把翻译工作继续进行下去时，事情出现了转机。未鹏在看到征集译者的帖子后提出加入翻译团队，我们在MSN上一起讨论起如何开展这本书的翻译工作。我们都认为这本书确实是一本难得的好书，建议在翻译团队中再增加一些有实力的人。于是，我们趁热打铁把自己熟悉的一些同学和同事都拉进MSN聊天室里，然后动员他们再推荐一些合适的译者。

事实证明，群众的力量是巨大的。当天讨论的结果很成功，我们最终组成了一个翻译团队，取名为“BC Group”，其中的BC即为“Beautiful Code”的缩写。我们根据每个人的技术专长和感兴趣领域来分配翻译任务，并且为了保证翻译风格的一致性，我们规定了在译稿中要统一专业术语，叙述语态要保持中文习惯等。在随后的翻译过程中，我们创建了Google Group专用论坛，定期地在MSN上讨论所遇到的问题，交流翻译过程的体会和心得。功夫不负有心人，在经历半年多的努力之后，本书的中文初稿终于得以完成。

然而，或许是好事多磨，《代码之美》在出版过程中遇到了一些波折。本来计划4月份左右出版的《代码之美》，不得不一直推迟到9月份才确定下来。所幸的是，问题最终得以

解决，而 BC Group 的努力也没有付之东流，因此不久之后本书将与读者见面。

当然，由于时间和精力有限，书中难免会存在一些问题，还望各位读者不吝指正。

致谢

感谢 BC Group 所有成员的努力。感谢史晓明积极参与翻译工作的讨论。感谢机械工业出版社华章分社的编辑，以及所有为本书付出努力的人们。

2008 年 9 月于武汉

附 BC Group 成员简介：

王昕：易安信（EMC）中国研发中心高级程序员，感兴趣的技术领域：运用不同的编程技术，搭建大型的、可扩展的、高性能应用程序。译有《C++ STL 中文版》、《C++ 编程惯用法》等。

刘未鹏：南京大学计算机系硕士在读，热爱编程、数学、心理学。个人博客“C++ 的罗浮宫”(<http://blog.csdn.net/pongba>)，长期关注 C++ 前沿技术。译有《Imperfect C++ 中文版》、《Exceptional C++ Style 中文版》、《修改代码的艺术》等。

袁泳：高级程序员，一直从事 IBM WebSphere Commerce 的开发工作。

王江平：运软网络科技（上海）有限公司高级软件工程师。技术专长为 C++，酷喜读书，曾长期从事 EDA 领域行业应用软件开发，目前的主要工作是 Windows 系统监控。

史苏：欧特克中国软件研发中心软件开发工程师，主要负责建筑 CAD 软件相关的研发工作。热衷于 C++，算法，数学，程序设计语言理论和社会化网络方面的研究。酷爱读书，参与翻译《游戏编程精粹 6》。

华咤镇：2006 年加入微软开发工具部中国分部，目前从事微软并行处理平台的研发工作。此前曾参与 Visual Studio 中 WCF 模块的开发工作。此前在 FujiXerox 工作了 3 年，从事打印机管理软件和文档处理工具的开发。

朱永泰：2007 年加入微软服务器与开发工具部中国分部，目前在 develop devision 从事 CLR 的相关工作。

聂雪军：华中科技大学计算机学院博士在读，研究方向为存储网络。感兴趣的技术领域包括 C++，并行算法，大型软件的协作开发等。译有《Windows 编程启示录》，《Exceptional C++ 中文版》，《C++ 编程风格》等。

目录

推荐序	VI
译者序	XXI
序	1
前言	3
第 1 章 正则表达式匹配器	9
编程实践	10
实现	11
讨论	12
其他的方法	14
构建	15
结论	17
第 2 章 Subversion 中的增量编辑器： 灵活的接口	19
版本控制与目录树的转换	20
表达目录树的差异	24
增量编辑器接口	25
但这是艺术吗	30
像体育比赛一样抽象	32
结论	35
第 3 章 我从未编写过的最漂亮的代码	37
我编写过的最漂亮的代码	38
事半功倍	39
观点	45

本章的中心思想是什么	47
结 论	48
致 谢	49
第 4 章 查找	51
耗 时	51
问题：数据	52
问题：时间，人物，以及对象	61
大 规 模 尺 度 的 搜 索	67
结 论	69
第 5 章 正确、优美、迅速(按重要性排序)： 从设计 XML 验证器中学到的经验	71
XML 验证器的作用	71
问题所在	72
版本 1：简单的实现	74
版本 2：模拟 BNF 语法 —— 复杂度 $O(N)$	75
版本 3：第一个复杂度 $O(\log N)$ 的优化	77
版本 4：第二次优化：避免重复验证	78
版本 5：第三次优化：复杂度 $O(1)$	80
版本 6：第四次优化：缓存	84
从故事中学到的	86
第 6 章 集成测试框架：脆弱之美	87
三个类搞定一个验收测试框架	88
框架设计的挑战	90
开放 式 框 架	92
一个 HTML 解析器可以简单到什么程度	93
结 论	96
第 7 章 漂亮的测试	99
讨厌的二分查找	101
JUnit 简介	103

将二分查找进行到底	105
结 论	118
第 8 章 图像处理中的即时代码生成	121
第 9 章 自顶向下的运算符优先级	145
JavaScript	146
符 号 表	147
语 素	148
优 先 级	150
表 达 式	150
中 置 运 算 符	151
前 置 运 算 符	153
赋 值 运 算 符	154
常 数	154
Scope	155
语 句	157
函 数	160
数 组 和 对 象 字 面 量	161
要 做 和 要 思 考 的 事	162
第 10 章 寻求快速的种群计数	163
基 本 方 法	164
分 治 法	165
其 他 方 法	167
两 个 字 种 群 计 数 的 和 与 差	169
两 个 字 的 种 群 计 数 比 较	169
数 组 中 的 1 位 种 群 计 数	170
应 用	175
第 11 章 安全通信：自由的技术	179
项 目 启 动 之 前	180
剖 析 安 全 通 信 的 复 杂 性	181

可用性是关键要素	183
基础	186
测试集	190
功能原型	191
清理，插入，继续……	191
在喜马拉雅山的开发工作	195
看不到的改动	201
速度确实重要	203
人权中的通信隐私	203
程序员与文明	204
 第 12 章 在 BioPerl 里培育漂亮代码	207
BioPerl 和 Bio::Graphics 模块	207
Bio::Graphics 的设计流程	211
扩展 Bio::Graphics	230
结论和教训	234
 第 13 章 基因排序器的设计	237
基因排序器的用户界面	238
通过 Web 跟用户保持对话	239
多态的威力	241
滤除无关的基因	244
大规模美丽代码理论	245
结论	248
 第 14 章 优雅代码随硬件发展的演化	251
计算机体系结构对矩阵算法的影响	252
一种基于分解的方法	253
一个简单版本	255
LINPACK 库中的 DGEFA 子程序	257
LAPACK DGETRF	259
递归 LU	262
ScaLAPACK PDGETRF	265

针对多核系统的多线程设计	269
误差分析与操作计数浅析	272
未来的研究方向	273
进一步阅读	273
第 15 章 漂亮的设计会给你带来长远的益处	275
对于漂亮代码的个人看法	275
对于 CERN 库的介绍	276
外在美	277
内在美	283
结论	289
第 16 章 Linux 内核驱动模型：协作的好处	291
简单的开始	292
进一步简化	297
扩展到上千台设备	300
小对象的松散结合	301
第 17 章 额外的间接层	303
从直接代码操作到通过函数指针操作	304
从函数参数到参数指针	305
从文件系统到文件系统层	310
从代码到 DSL (Domain-Specific Language)	312
复用与分离	314
分层是永恒之道吗	315
第 18 章 Python 的字典类：如何打造全能战士	317
字典类的内部实现	319
特殊调校	321
冲突处理	322
调整大小	323
迭代和动态变化	325
结论	325

致 谢	326
第 19 章 NumPy 中的多维迭代器	327
N 维数组操作中的关键挑战	328
N 维数组的内存模型	329
NumPy 迭代器的起源	331
迭代器的设计	331
迭代器的接口	337
迭代器的使用	339
结 论	342
第 20 章 NASA 火星漫步者任务中的高可靠企业系统	345
任务与 CIP	346
任务需求	347
系统架构	348
案例分析：流服务	351
可靠性	355
稳定性	362
结 论	364
第 21 章 ERP5：最大可适性的设计	367
ERP 的总体目标	368
ERP5	368
Zope 基础平台	370
ERP5 Project 中的概念	374
编码实现 ERP5 Project	375
结 论	379
第 22 章 一匙污水	381
第 23 章 MapReduce 分布式编程	397
激动人心的示例	397
MapReduce 编程模型	400

其他 MapReduce 示例	401
分布式 MapReduce 的一种实现	403
模型扩展	406
结论	407
进一步阅读	407
致谢	408
附录：单词计数解决方案	408
第 24 章 美丽的并发	411
一个简单的例子：银行账户	412
软件事务内存	415
圣诞老人问题	424
对 Haskell 的一些思考	434
结论	435
致谢	436
第 25 章 句法抽象：syntax-case 展开器	437
syntax-case 简介	442
展开算法	444
例子	456
结论	458
第 26 章 节省劳动的架构：一个面向对象的 网络化软件框架	459
示例程序：日志服务	461
日志服务器框架的面向对象设计	464
实现串行化日志服务器	470
实现并行日志服务器	475
结论	481
第 27 章 以 REST 方式集成业务伙伴	483
项目背景	484
把服务开放给外部客户	484
使用工厂模式转发服务	487

用电子商务协议来交换数据	489
结论	494
第 28 章 漂亮的调试	495
对调试器进行调试	496
系统化的过程	498
关于查找的问题	499
自动找出故障起因	500
增量调试	502
最小化输入	505
查找缺陷	505
原型问题	508
结论	509
致谢	509
进一步阅读	509
第 29 章 代码如散文	511
第 30 章 当你与世界的联系只有一个按钮时	517
基本的设计模型	518
输入界面	521
用户界面的效率	535
下载	535
未来的发展方向	535
第 31 章 Emacspeak：全功能音频桌面	537
产生语音输出	538
支持语音的 Emacs	539
对于在线信息的简单访问	551
小结	558
致谢	561
第 32 章 变动的代码	563
像书本一样	565

功能相似的代码在外觀上也保持相似	566
缩进带来的危险	567
浏览代码	568
我们使用的工具	569
DiffMerge 的曲折历史	571
结论	573
致谢	573
进一步阅读	573
第 33 章 为 “The Book” 编写程序	575
没有捷径	576
给 Lisp 初学者的提示	576
三点共线	577
不可靠的斜率	580
三角不等性	581
河道弯曲模型	583
“Duh!” —— 我的意思是 “Aha!”	584
结论	586
进一步阅读	586
后记	589
作者简介	591
面条录音机大全	591 章 31 Emacspeak
出镜音频空当	
round 音频脚本	
面条早读的语音文字子饭	
薪水	
梅姨	
面条的惊变	595 章 35
第一本巨著	

序

Greg Wilson

我在 1982 年夏天获得了第一份程序员工作。在我工作了两个星期后，一位系统管理员借给了我两本书：Kernighan 和 Plauger 编写的《The Elements of Programming Style》(McGraw-Hill 出版社)和 Wirth 编写的《Algorithms + Data Structures = Programs》(Prentice Hall 出版社)。这两本书让我大开眼界——我第一次发现程序并不仅仅只是一组计算机执行的指令。它们可以像做工优良的橱柜一样精致，像悬索吊桥一样漂亮，或者像 George Orwell 的散文一样优美。

自从那个夏天以来，我经常听到人们感叹我们的教育并没有教会学生看到这一点。建筑师们需要观摩建筑物，作曲家们需要研习他人的作品，而程序员——他们只有在需要修改 bug 时才会去阅读其他人的代码；即使在这个时候，他们也会尽可能减少阅读量。我们曾告诉学生使用有意义的变量名，曾向他们介绍过一些基本的设计模式，但很奇怪，为什么他们编写的大多数代码都很难看呢！

本书将试图改变这种状况。2006 年 5 月，我邀请了一些著名的（以及不太著名的）软件设计师来分析和讨论他们所知道的漂亮代码。正如在本书中将要介绍的，他们在许多不同的地方发现了代码的漂亮性。有些漂亮性存在于手工精心打造软件的细微之处，而

有些漂亮性蕴涵在大局之中——那些使程序能够持续发展的架构，或者用来构造程序的技术。

无论他们是在什么地方发现的这些漂亮性，我都非常感谢我们的投稿人抽出时间为我奉献了这样的一次学习旅程。我希望你能够享受阅读此书的乐趣，就像 Andy 和我非常享受编辑这本书的过程，此外，我还希望这本书能激发你创建出一些漂亮的作品。

卷

Gerd Miesen

美国科学家们在《科学》杂志上报告说，他们通过分析人类基因组，发现人类的基因与黑猩猩的基因有98%以上的一致性。

革命，就必须首先学会要将革命的斗争深入到人民的日常生活中，来以无产阶级思想和共产主义精神鼓舞人民。——员半夏而，弗利兹·列宁区领导委员会书记处书记。他强调必须加强组织，最尖锐地揭露反革命的阴谋，对整个政治形势进行分析，指出反动派在共产国际中的活动，指出反动派的本性，指明它的目的和方法。并要求阅读《共产国际》杂志，学习列宁的学说。

（南齐书不列传）初除著作郎兼秘书，竟陵王子良、江孝和、王僧虔、王筠等皆好之。竟陵子良常与人书曰：「卿家子良比卿家子良，如白玉比白玉，如明珠比明珠。」竟陵子良亦常以卿家子良比卿家子良。竟陵子良尝问王僧虔：「卿何如我？」僧虔答曰：「卿如卿，我不如我。」竟陵子良笑而不答。

前言

《Beautiful Code》是由 Greg Wilson 在 2006 年构思的，本书的初衷是希望从优秀的软件开发人员和计算机科学家中提炼出一些有价值的思想。他与合作编辑 Andy Oram 一起走访了世界各地不同技术背景的专家。

虽然本书的涉猎范围很广，但也只能代表一小部分在软件开发这个最令人兴奋的领域所发生的事。还有许多其他的项目同样是有教育意义的，每天都有大量的程序员在不断地推动着这些项目，而我们无法联系到所有的这些程序员。此外，还有许多其他优秀的约稿者没有出现在本书中，因为他们当时都太忙了，或者有着与之冲突的义务。为了汲取所有这些人的思想，我们希望将来沿用相同的方式来编写后续书籍。

本书章节内容的组织

第 1 章，正则表达式匹配器，作者 Brian Kernighan，介绍了对一种语言和一个问题的深入分析以及由此产生的简洁而优雅的解决方案。

第2章, *Subversion* 中的增量编辑器: 灵活的接口, 作者 Karl Fogel, 首先介绍了一个精心设计的抽象, 然后证明了这种抽象能够在系统将来的开发中带来一致性。

第3章, 我从未编写过的最漂亮的代码, 作者 Jon Bentley, 介绍了如何在无需执行函数的情况下测试函数的性能。

第4章, 查找, 作者 Tim Bray, 应用了计算机科学中的多种技术来研究一个对许多计算任务来说都很重要的问题。

第5章, 正确、优美、迅速 (按重要性排序): 从设计 XML 验证器中学到的经验, 作者 Elliotte Rusty Harold, 解决了程序在完备性和高性能之间的冲突。

第6章, 集成测试框架: 脆弱之美, 作者 Michael Feathers, 介绍了一个打破常规并获得优雅解决方案的示例。

第7章, 漂亮的测试, 作者 Alberto Savoia, 介绍了一种全新的测试方法, 不仅能够消除 bug, 还可以使你成为一个更优秀的程序员。

第8章, 图像处理中的即时代码生成, 作者 Charles Petzold, 介绍了一种在维护可移植性的同时还能够提高性能的方法。

第9章, 自顶向下的运算符优先级, 作者 Douglas Crockford, 介绍了一种几乎被人们遗忘的解析技术, 并且给出了它与 JavaScript 语言的最新相关性。

第10章, 寻求快速的种群计数, 作者 Henry S. Warren, Jr., 揭示了在一个看似简单的问题上如何应用一些巧妙的算法。

第11章, 安全通信: 自由的技术, 作者 Ashish Gulhati, 讨论了一个安全消息应用程序的发展过程, 这个程序被设计用来使用户能够直观地访问那些成熟但却经常产生误解的密码技术。

第12章, 在 BioPerl 里培育漂亮代码, 作者 Lincoln Stein, 介绍了如何通过将一种灵活的语言和客户定制的模块组合在一起, 从而使编程技术一般的开发人员能够为他们的数据创建出功能强大的虚拟化形式。

第13章, 基因排序器的设计, 作者 Jim Kent, 将简单的构件组合起来从而为基因研究人员生成稳定并且有价值的工具。

第14章, 优雅代码随硬件发展的演化, 作者 Jack Dongarra 和 Piotr Luszczek, 介绍了 LINPACK 及其相关主要软件包的发展历史, 从而给出了在面对新的计算架构时, 应该

如何对假设条件进行重新评估。

第15章，漂亮的设计会给你带来长远的益处，作者 Adam Kolawa，阐述了数十年前所使用的良好设计原则如何帮助 CERN 中广泛应用的数学库（LINPACK 的前身）经受住时间的考验。

第16章，Linux 内核驱动模型：协作的好处，作者 Greg Kroah-Hartman，阐述了不同的协作者在解决不同难题上所做出的努力以及如何来推动一个多线程复杂系统的成功发展。

第17章，额外的间接层，作者 Diomidis Spinellis，介绍了如何对多数驱动程序和文件模块中的常见操作进行抽象，以及如何通过这种抽象来提升FreeBSD内核的灵活性和可维护性。

第18章，Python 的字典类：如何打造全能战士，作者 Andrew Kuchling，介绍了一个能够适应某些特殊情况的完备设计以及如何通过这种设计来使一种语言特性支持许多不同的用途。

第19章，NumPy 中的多维迭代器，作者 Travis E. Oliphant，展示了如何把复杂性成功隐藏在简单接口背后的设计步骤。

第20章，NASA 火星漫步者任务中的高可靠企业系统，作者 Ronald Mak，介绍了如何使用工业标准、最佳实践和 Java 技术来满足 NASA 探险任务的高可靠性需求。

第21章，ERP5：最大可适性的设计，作者 Rogerio Atem de Carvalho 和 Rafael Monnerat，介绍了如何用免费的软件工具和灵活的架构来开发一个功能强大的 ERP 系统。

第22章，一匙污水，作者 Bryan Cantrill，让读者和作者一起来体验一个令人毛骨悚然的 bug 以及一种违背直觉的巧妙的解决方案。

第23章，MapReduce 分布式编程，作者 Jeff Dean 和 Sanjay Ghemawat，描述了一个能够提供简单编程抽象的系统，这种抽象用来在 Google 中进行大规模分布式数据处理，并能够自动处理分布式计算中的许多难题，包括自动并行化、负载均衡以及故障处理等。

第24章，美丽的并发，作者 Simon Peyton Jones，通过软件事务内存（Software Transactional Memory）来消除大多数并发程序中的困难，在本章中使用 Haskell 语言来说明。

第25章，句法抽象：*syntax-case* 展开器，作者 R. Kent Dybvig，介绍了如何在 Scheme 中防止宏——这个许多语言和系统中的关键特性——产生错误的输出。

第26章，节省劳动的架构：一个面向对象的网络化软件框架，作者 William R. Otte 和 Douglas C. Schmidt，应用了许多标准的面向对象设计技术，例如模式和框架等，来分发日志，从而保持系统的灵活性和模块性。

第27章，以 REST 方式集成业务伙伴，作者 Andrew Patzer，通过根据需求来设计一个 B2B Web Service，从而表现出设计者对程序开发人员的尊重。

第28章，漂亮的调试，作者 Andreas Zeller，介绍了如何通过严谨的验证代码方法来减少追踪错误的时间。

第29章，代码如散文，作者 Yukihiro Matsumoto，介绍了他在设计 Ruby 编程语言时所遵循的一些规则，并且这些规则通常均有助于开发出更优秀的软件。

第30章，当你与世界的联系只有一个按钮时，作者 Arun Mehta，介绍了在文字编辑系统中一种不可思议的界面设计，这种设计使患有高度运动神经残疾的用户，例如 Stephen Hawking 教授，也可以通过计算机进行交流。

第31章，*Emacspeak*: 全功能音频桌面，作者 T. V. Raman，介绍了如何在 Emacs 通过 Lisp 的 advice 功能来满足 Emacs 整体操作环境中的需求——产生丰富的语音输出，而同时无需修改软件系统的底层源代码。

第32章，变动的代码，作者 Laura Wingerd 和 Christopher Seiwald，列出了一些对编程精确性有着强大影响的简单规则。

第33章，为 “The Book” 编写程序，作者 Brian Hayes，介绍了在解决一个看似简单的计算几何学问题时所遭受的挫折，并给出了这个问题令人惊叹的解决方案。

字体约定

本书使用如下印刷惯例：

斜体 (Italic) 表示新的术语，数学变量，URL，文件名和目录名，以及命令。

等宽字 (Constant width)

表示程序代码，文件内容，在计算机控制台上的文本输出显示。

等宽粗体字 (Constant width bold)

表示用户手工输入的命令或其他文本。

等宽斜体 (*Constant width italic*)

表示需要使用用户提供的值来代替的文本。

怎样联系我们

请将您对本书的宝贵意见及问题告诉我们。来信请寄：

美国：

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

中国：

100080 北京市西城区南大街 2 号成铭大厦 C-807，邮编：100035
奥莱理软件（北京）有限公司

我们为本书设有一个 Web 页面，上面列出了勘误、例子和一些额外的信息。你可以通过如下网址访问该页面：

<http://www.oreilly.com/catalog/9780596510046>

对本书评论或者提技术问题请发送 email 到：

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

关于我们的书籍、研讨会、资源中心以及 O'Reilly Network 更多的信息，请参考网站：

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

代码示例的使用

本书可以帮助你完成日常工作。你可以在你的程序和文档中使用书中的代码。除非你需要重用大部分的代码，否则就不需要联系我们获得许可。例如，如果在编写程序时使用了书中的几段代码，那么不需要申请许可。而在销售或者分发 O'Reilly 书籍中的光盘示例代码时则需要获得许可。

引用本书内容或者示例代码来回答问题无需获得许可。而把书中的大量示例代码写入到你的文档中则需要获得许可。

我们欢迎，但并不要求，授权。授权通常包括标题、作者、出版商以及 ISBN。例如：“*Beautiful Code*, edited by Andy Oram and Greg Wilson. Copyright 2007 O'Reilly Media, Inc., 978-0-596-51004-6.”

如果你认为你对代码示例的使用方式不符合上面给出的情况，请联系我们
permissions@oreilly.com。

正则表达式匹配器

Brian Kernighan

正则表达式是描述文本模式的记法 (notation)，它可以有效地构造一种用于模式匹配的专用语言。虽然正则表达式可以有多种不同的形式，但它们都有着共同的特点：模式中的大多数字符都是匹配字符串中的字符本身，而有些元字符 (metacharacter) 却有着特定的含义，例如 * 表示某种重复，而 [...] 表示方括号中字符集合的任何一个字符。

实际上，在文本编辑器之类的程序中，所执行的查找操作都是查找文字，因此正则表达式通常是像 “print” 之类的字符串，而这类字符串将与文档中所有的 “printf” 或者 “sprintf” 或者 “printer paper” 相匹配。在 Unix 和 Windows 中可以使用所谓的通配符来指定文件名，其中字符 * 可以用来匹配任意数量的字符，因此 *.c 这个模板就将匹配所有以 .c 结尾的文件。此外，还有许许多多不同形式的正则表达式，甚至在有些情况下，这些正则表达式可以被认为都是相同的。Jeffrey Friedl 编著的《Mastering Regular Expressions》一书对这方面的问题进行了广泛的研究。

Stephen Kleene 在上世纪 50 年代的中期发明了正则表达式，用来作为有限自动机的表示法，事实上，正则表达式与其所表示的有限自动机是等价的。上世纪 60 年代中期，正则表达式最初出现在 Ken Thompson 版本的 QED 文本编辑器的程序设置中。1967 年 Thompson 申请了一项基于正则表达式的快速文本匹配机制的专利。这项专利在 1971 年

获得了批准，它是最早的软件专利之一[U.S. Patent 3,568,156, Text Matching Algorithm, March 2, 1971]。

后来，正则表达式技术从 QED 移植到了 Unix 的编辑器 *ed* 中，然后又被移植到经典的 Unix 工具 *grep* 中，而 *grpe* 正是由于 Thompson 对 *ed* 进行了彻底的修改而形成的。这些广为应用的程序使得正则表达式为早期的 Unix 社群所熟知。

Thompson 最初编写的匹配器是非常快速的，因为它结合了两种独立的思想。一种思想是在匹配过程中动态地生成机器指令，这样匹配的执行速度就是执行机器指令的速度而不是解释执行的速度。另一种思想是在每个阶段中都尽可能地执行匹配操作，这样在查找可能的匹配时无需进行回溯（backtrack）操作。在 Thompson 后来编写的文本编辑器程序中，例如 *ed*，在编写匹配代码时使用了一种更为简单的算法，这种算法将会在必要的时候进行回溯。从理论上来看，这种方法的运行速度要更慢，但在实际情况中，这种模式很少需要进行回溯，因此，*ed* 和 *grep* 中的算法和代码足以应付大多数的情况。

在后来的正则表达式匹配器中，例如 *egrep* 和 *fgrep* 等，都增加了更为丰富的正则表达式类型，并且侧重于使匹配器无论在什么模式下都能够快速执行。功能更为强大的正则表达式正在被越来越多地使用，它们不仅被包含在用 C 语言开发的库中，而且还被作为脚本语言语法的一部分，如 Awk 和 Perl。

编程实践

在 1998 年，Rob Pike 和我还在编写《The Practice of Programming》(Addison-Wesley)一书。书中的最后一章是“记法”，在这一章中收录了许多示例代码，这些示例都很好地说明了优良的记法将产生更好的程序以及更好的设计。其中包括使用简单的数据规范（例如 *printf*）以及从表格中生成代码。

由于我们有着深厚的 Unix 技术背景，并且在使用基于正则表达式记法的工具上有着近 30 年的经验，因此很自然地希望在本书中包含一个对正则表达式的讨论，当然包含一个实现也是必须的。由于我们侧重于工具软件的使用，因此似乎最好应该把重点放在 *grep* 中的正则表达式类型上——而不是放在诸如 shell 通配符这样的正则表达式上——这样我们还可以在随后来讨论 *grep* 本身的设计。

然而，问题是现有的正则表达式软件包都非常庞大。*grep* 中的代码长度超过 500 行（大约 10 页书的长度），并且在代码的周围还有复杂的上下文环境。开源的正则表达式软件包则更为庞大——代码的长度几乎布满整本书——因为这些代码需要考虑通用性、灵活性以及运行速度；因此，所有这些正则表达式都不适合用来教学。

我向Rob建议我们需要一个最小的正则表达式软件包，它可以很好地诠释正则表达式的基本思想，并且能够识别出一组有用的并且重要的模式。理想的情况是，所需代码长度只有一页就够了。

Rob听了我的提议后马上回到了他的办公室。我现在还记得，一、两个小时后他回来了，并且给了我一段大约30行的C代码，在《The Practice of Programming》一书的第9章中包含了这段代码。这段代码实现了一个正则表达式匹配器，用来处理以下的模型。

字符	含义
c	匹配任意的字母c
.(句点)	匹配任意的单个字符
^	匹配输入字符串的开头
\$	匹配输入字符串的结尾
*	匹配前一个字符的零个或者多个出现

这是一个非常有用的匹配器，根据我在日常工作中使用正则表达式的经验，它可以轻松解决95%左右的问题。在许多情况下，解决正确的问题就等于朝着创建漂亮的程序迈进了一大步。Rob值得好好地表扬，因为他从大量的可选功能集中选取了一小组重要的、定义明确的以及可扩展的功能。

Rob给出的实现本身就是漂亮代码的一个极佳示例：紧凑，优雅，高效并且实用。这是我所见过的最佳的递归示例之一，在这段代码中还展示了C指针的强大功能。虽然当时我们最关心的是如何通过使程序更易于使用（同时也更易于编写）来体现良好记法的重要性，但正则表达式代码同样也是说明算法、数据结构、测试、性能增强以及其他重要主题的最好方式。

实现

在《The Practice of Programming》一书中，正则表达式匹配器是模拟*grep*程序中的一部分，但正则表达式的代码完全可以从编写环境中独立出来。这里我们并不关心main程序；像许多Unix工具一样，这个main程序将从标准输入或者一组文件中读取，然后输出与正则表达式匹配的文本行。

以下是匹配算法的代码：

```
/* match: 在text中查找regexp */
int match(char *regexp, char *text)
{
```

```

        if (regexp[0] == '^')
            return matchhere(regexp+1, text);
        do { /* 即使字符串为空时也必须检查 */
            if (matchhere(regexp, text))
                return 1;
        } while (*text++ != '\0');
        return 0;
    }

/* matchhere: 在 text 的开头查找 regexp */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        return matchhere(regexp+1, text+1);
    return 0;
}

/* matchstar: 在 text 的开头查找 C*regexp */
int matchstar(int c, char *regexp, char *text)
{
    do { /* 通配符 * 匹配零个或多个实例 */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}

```

讨论

函数 `match(regexp, text)` 用来判断文本中是否出现正则表达式；如果找到了一个匹配的实例则返回 1，否则返回 0。如果有多个匹配的实例，那么函数将找到文本中最左边的并且最短的匹配实例。

`match` 函数中的基本操作简单明了。如果正则表达式中的第一个字符是 ^ (固定位置的匹配)，那么匹配实例就一定要出现在字符串的开头。也就是说，如果正则表达式是 `^xyz`，那么仅当 `xyz` 出现在文本的开头而不是中间的某个位置时才会匹配成功。在代码中通过把正则表达式的剩余部分与文本的起始部分，而不是其他地方，来进行比较以判断是否匹配成功。如果第一个字符不是 ^，那么正则表达式就可以在字符串中的任意位置上进行匹配。在代码中通过把匹配模式依次在文本中的每个字符位置上进行匹配来判断。如果存在多个匹配，那么代码只会识别第一个（最左边的）匹配。也就是说，如

果正则表达式是xyz，那么将会匹配首次出现的xyz，而且不考虑这个匹配出现在什么位置上。

注意，对输入字符串的递进操作是在一个do-while循环中进行的，这种结构在C程序中使用相对较少。在代码中使用do-while而不是while通常会带来这样的疑问：为什么不在循环的起始处判断循环条件，而是在循环末尾当执行完了某个操作之后才进行判断呢？然而，这里的判断是正确的：由于*运算符允许零长度的匹配，因此我们首先需要判断是否存在一个空的匹配。

大部分的匹配工作都是在matchhere(regexp, text)函数中完成的，这个函数将判断正则表达式与文本的开头部分是否匹配。函数matchhere把正则表达式的第一个字符与文本的第一个字符进行匹配。如果匹配失败，那么在这个文本位置上就不存在匹配，因此matchhere将返回0。然而，如果匹配成功了，函数将递进到正则表达式的下一个字符和文本的下一个字符继续进行匹配。这是通过递归地调用matchhere函数来实现的。

由于可能存在着一些特殊的情况，以及需要设置终止递归的条件，因此实际的处理过程要更为复杂些。最简单的情况就是，当正则表达式递进到末尾时(`regexp[0] == '\0'`)，所有之前的判断都成功了，那么这个正则表达式就与文本匹配。

如果正则表达式是一个字符后面跟着一个*，那么将会调用matchstar来判断闭包(closure)是否匹配。函数matchstar(c, regexp, text)将尝试匹配重复的文本字符c，从零重复开始并且不断累加，直到匹配text的剩余字符，如果匹配失败，那么函数就认为不存在匹配。这个算法将识别出一个“最短的匹配”，这对简单的模式匹配来说是很好的，例如grep，这种情况下的主要问题是尽可能快地找到一个匹配。而对于文本编辑器来说，“最长的匹配”则是更为直观，且肯定是更好的，因为在这种情况下通常需要对匹配的文本进行替换。在目前许多的正则表达式库中同时提供了这两种方法，在《The Practice of Programming》一书中给出了基于本例中matchstar函数的一种简单变形，我们在后面将给出这种形式。

如果在正则表达式的末尾包含了一个\$，那么仅当text位于末尾时才会匹配成功：

```
if (regexp[0] == '$' && regexp[1] == '\0')
    return *text == '\0';
```

如果没有包含\$，并且如果当前不是处于text字符串的末尾（也就是说，`*text != '\0'`）并且如果text字符串的第一个字符匹配正则表达式的第一个字符，那么到现在为止都是没有问题的；我们将接着判断正则表达式的下一个字符是否匹配text的下一个字符，这是通过递归调用matchhere函数来实现的。这个递归调用不仅是本算法的核心，也是这段代码如此紧凑和整洁的原因。

如果所有这些匹配尝试都失败了，那么正则表达式和text在这个位置上就不存在匹配，因此函数matchhere将返回0。

在这段代码中大量地使用了C指针。在递归的每个阶段，如果存在某个字符匹配，那么在随后的递归调用中将执行指针算法（例如，`regexp+1` and `text+1`），这样在随后的函数调用中，参数就是正则表达式的下一个字符和text的下一个字符。递归的深度不会超过匹配模式的长度，而通常情况下匹配模式的长度都是很短的，因此不会出现耗尽内存空间的危险。

其他的方法

这是一段写得非常优雅的代码，但并不是完美的。我们还能否做一些改进工作？我可以对matchhere中的操作进行重新安排，在处理*之前首先处理\$。虽然这种安排不会对函数的执行带来影响，但却使得函数看上去要自然一些，而在编程中一个良好的规则就是：在处理复杂的情况之前首先处理容易的情况。

然而，通常这些判断的顺序是非常重要的。例如，在matchstar的这个判断中：

```
} while (*text != '\0' && (*text++ == c || c == '.'));
```

无论在什么情况下，我们都必须递进text字符串中的一个或多个字符，因此在`text++`中的递增运算一定要执行。

该代码对递归的终止条件进行了谨慎的处理。通常，匹配过程的成功与否，是通过判断正则表达式和text中的字符是不是同时被处理完来决定的。如果是同时被处理完了，那么就表示匹配成功，如果其中一方在另一方之前被处理完了，那么就表示匹配失败。在下面这行代码中很明显地说明了这个判断。

```
if (regexp[0] == '$' && regexp[1] == '\0')  
    return *text == '\0';
```

但在其他的情况下，还存在一些微妙的终止条件。

如果在matchstar函数中需要识别最左边的以及最长的匹配，那么函数将首先识别输入字符c的最大重复序列。然后函数将调用matchhere来尝试把匹配延伸到正则表达式的剩余部分和text的剩余部分。每次匹配失败都会将cs的出现次数减1，然后再次开始尝试，包括处理字符c零出现的情况：

```
/* matchstar: 搜索 c*regexp 的最左以及最长的匹配 */  
int matchstar(int c, char *regexp, char *text)
```

```

{
    char *t;

    for (t = text; *t != '\0' && (*t == c || c == '.', ','); t++)
        ;
    do { /* 通配符 * 匹配零个或者多个实例 */
        if (matchhere(regexp, t))
            return 1;
    } while (t-- > text);
    return 0;
}

```

我们来看一下正则表达式`(.*)`，它将匹配括号内任意长度的text。假设给定了text：

```
for (t = text; *t != '\0' && (*t == c || c == '.', ','); t++)
```

从开头位置起的最长匹配将会识别整个括号内的表达式，而最短的匹配将会停止在第一次出现右括号的地方。（当然，从第二个左括号开始的最长匹配将会延伸到text的末尾。）

构建

《The Practice of Programming》一书主要教授良好的程序设计。在编写该书时，Rob 和我还在贝尔实验室工作，因此我们无法知道在课堂上使用这本书会有什么样的效果。令人高兴的是，我们发现这本书中的某些内容在课堂上确实有着不错的效果。在2000年教授程序设计中的重点内容时，我们就使用了这段代码。

首先，这段代码以一种全新的形式展示了递归的强大功能及其带来的整洁代码。它既不是另一种版本的快速排序（或者阶乘！）算法，也不是某种树的遍历算法。

这段代码同时还是性能试验的一个很好示例。其性能与系统中的grep并没有太大的差异，这表明递归技术的开销并不是非常大的，因此没有必要对这段代码进行调整。

此外，这段代码还充分说明了优良算法的重要性。如果在模式中包含了几个`.*`序列，那么在简单的实现中将需要进行大量的回溯操作，并且在某些情况下将会运行得极慢。

在标准的 Unix grep 中有着同样的回溯操作。例如，下面这个命令：

```
grep 'a.*a.*a.*a.a'
```

在普通的机器上处理一个 4 MB 的文本文件要花费 20 秒的时间。

如果某个实现是基于把非确定型有限自动机转换为确定型有限自动机，例如egrep，那么在处理困难的情况时将会获得比较好的性能；它可以在不到十分之一秒的时间内处理同样的模式和同样的输入，并且运行时间通常是独立于模式的。

对于正则表达式类型的扩展将形成各种任务的基础。例如：

1. 增加其他的元字符，例如+用于表示前面字符的一个或多个出现，或者?用于表示零个或一个字符的匹配。此外，还可以增加一些方式来表示元字符，例如\\$表示在模式中的\$字符。
2. 将正则表达式处理过程分成编译阶段和执行阶段。编译阶段把正则表达式转换为内部形式，使匹配代码更为简单或者使随后的匹配过程运行得更为迅速。对于最初设计中的简单的正则表达式来说，这种拆分并不是必须的，但在像*grep*这样的程序中，这种拆分是有意义的，因为这种类型的正则表达式要更为丰富，并且同样的正则表达式将会被用于匹配大量的输入行。
3. 增加像[abc]和[0-9]这样的类型，这在传统的*grep*中分别表示匹配a或b或c和一个数字。这可以通过几种方式来实现，最自然的方式似乎就是把最初代码中的char*变量用一个结构来代替：

```
typedef struct RE {  
    int type; /* CHAR, STAR, 等等 */  
    int ch; /* 字符本身 */  
    char *ccl; /* 用于 [...] instead */  
    int nccl; /* 如果表示集合取反，即[^...], 则为真 */  
}RE;
```

并且修改相应的代码以处理一个结构数组而不是处理一个字符数组。在这种情况下，我们并不需要把编译阶段从执行阶段中分离出来，但这里的分离过程是非常简单的。如果学生们把匹配代码预编译成这个结构，那么总会比那些试图动态地解释一些复杂模式数据结构的学生要做得更好。

为字符类型编写清晰并且无歧义的规范是项艰难的工作，而要用代码完美地实现处理更是难上加难，这需要大量的冗长并且晦涩的编码。随着时间的推移，我简化了这个任务，而现在大多数人会寻求像Perl那样的速记，例如\d表示数字，\D表示非数字，而不再像最初那样在方括号内指定字符的范围。

4. 使用不透明的类型来隐藏RE结构以及所有的实现细节。这是在C语言中展示面向对象编程技术的好方法，不过除此之外无法支持更多的东西。在实际情况中，将会创建一个正则表达式的类，并且在类中使用像RE_new()和RE_match()这样的名字来命名成员函数，而不是使用面向对象语言的语法。
5. 把正则表达式修改为像各种shell中的通配符那样：匹配模式的两端都被隐含地固定了，*匹配任意数量的字符，而?则匹配任意的单个字符。你可以修改这个算法或者把输入映射到现有的算法中。

6. 将这段代码转换成Java。最初的代码很好地使用了C指针，而把这个算法用不同的语言中实现出来则是一个很好的实践过程。在Java版本的代码中将使用String.charAt（使用索引而不是指针）或者String.substring（更接近于指针）。这两种方法都没有C代码整洁，并且也不紧凑。虽然在这个练习中性能并不是主要的问题，但有趣的是可以发现Java版本比C版本在运行速度上要慢6到7倍。
7. 编写一个包装类把这种类型的正则表达式转换成Java的Pattern类和Matcher类，这些类将以一种全然不同的方式来分离编译阶段和匹配阶段。这是适配器(Adapter)模式或者外观(Facade)模式的很好示例，这两种模式通常用来在现有的类或者函数集合外部定义不同的接口。

我曾经将这段代码大量地应用于条件判断技术的研究。正则表达式非常的丰富，其中所包含的条件判断不容忽视，但正则表达式又是很短小的，程序员可以很快地写出一组条件判断来自动执行。对于前面所列出的各种扩展，我要求学生用一种紧凑的语言写出大量的测试代码（这是“记法”的另一种示例），并且在他们自己的代码中使用这些测试代码；自然我在其他学生的代码中也使用了他们的条件判断。

结论

当RobPike最初写出这段代码时，我被它的紧凑性和优雅性感到惊讶——这段代码比我以前所想像的要更为短小并且功能也更为强大。通过事后分析，我们可以看到为什么这段代码如此短小的众多原因。

首先，我们很好地选择了一个功能集合，这些功能是最为有用的并且最能从实现中体现出核心思想，而没有任何多余的东西。例如，虽然固定模式^和\$的实现只需要写3~4行代码，但在统一处理普通情况之前，它展示了如何优雅地处理特殊情况。闭包操作*必须出现，因为它是正则表达式中的基本记号，并且它是提供处理不确定长度的惟一方式。增加+和?并不会有帮助理解，因此这些符号被留作为练习。

其次，我们成功地使用了递归。与显式的循环相比，递归这种基本的编程技巧通常会产生更短、更整洁的以及更优雅的代码，这也正是这里的示例。从正则表达式的开头和text的开头剥离匹配字符，然后对剩余的字符进行递归的思想，模仿了传统的阶乘或者计算字符串长度中的递归结构，而这里是将递归应用在一种更为有趣和更有用的环境中。

第三，这段代码使用了基本的语法来实现良好的效果。指针也可能被误用，但这里它们被用于创建紧凑的表达式，并且在这个表达式中自然地表达了提取单个字符和递进到下一个字符的过程。数组索引或者子字符串可以达到同样的效果，但在这段代码中，指针

能够更好的实现所需的功能，尤其是当指针与C语言中的自动递增运算和到布尔值的隐式转换结合在一起使用时。

我不清楚是否有其他的方法能够在如此少的代码中实现如此多功能，并且同时还要提供丰富的内涵和深层次的思想。

说到底，富强归根结底是国泰民安。生活的水好喝快乐在于周边环境大部分都达到标准并系统地一出事就对群众负责响应，而小部分却又失职懈怠，群众对不理性决策也会心生不满。广告宣传要杜绝虚假，民主决策要落实，媒体监督要及时公开透明，行政服务要高效廉洁，反腐败要坚决有力而且持久，（寓示新一届的“断5”）只有这样，富强才能真正实现。

但愿我们——利欲熏心者跟假新闻者越来越少，但愿仍有人坚持理想。但愿我们能一起努力，让社会更美好，让人民更幸福。大鹏展翅也需风，巨龙腾飞靠大家。但愿我们能一起努力，让社会更美好，让人民更幸福。

农村中农民从清晨日出到天黑成累死累活，含辛茹苦一个不落，连老弱病残都得上阵，先种一亩庄稼只赚点粮多赚点钱，方能生活勉强。再来就是逢年过节的，想想心酸比心酸还酸，名副其实的穷快活快吸了点旱烟，第二步做饭都得自己动手，因为自己

自己带饭。对进京的本集体经济组织，政府不派谁去领导，由领导用重金贿赂组织，大肆以公款来代的表心寒彻骨，附录中里头五项权，部分官员的真面目也暴露无遗。这里头真有各路豪杰的登场了打劫，愚昧的日本青虫都有地主阶级背景，有些根本就是大字不识

一个字的，中鼎不拍即吉傻麻糊猪头样一通乱来缺斤少两，你缺斤少两何止一斤两；果然是假虎真狼非吾所闻未见于前些日外间事，三箭不挺立在那三个草堆里了应该像咎吉中大造这个政治丑闻，失去劳动尊严的质子田则母猪，中鼎外甥女占田，果然是半同姓外孙而单称中千首违传统戒律，张口闭口三个

Subversion 中的增量编辑器： 灵活的接口

Karl Fogel

漂亮代码的示例通常都是一些良性有界（well-bounded）且易于理解的问题的局部解，例如 Duff's Device (http://en.wikipedia.org/wiki/Duff's_device) 算法或者 rsync 的旋转校验（rolling checksum）算法 (<http://en.wikipedia.org/wiki/Rsync#Algorithm>)。这并不是因为只有灵巧简单的解决方案才是漂亮的，而是因为理解复杂的代码需要有更多的背景知识，这些知识在一张餐巾纸后面是罗列不下的。

在本章中，我将通过几页的篇幅来讨论另外一种类型的漂亮——这种类型的漂亮不一定会很快地吸引普通的读者，但对于那些经常编码的程序员来说，随着在问题领域不断地积累经验，他们会很喜欢这种类型。本章中给出的漂亮代码示例并不是某个算法，而是一个接口：在开源的版本控制系统 Subversion (<http://subversion.tigris.org>) 中所使用的接口，这个接口不仅可以用来表示两个目录树之间的差异，而且也可以用来将某个目录树转换为另一个目录树。在 Subversion 中，这个接口的正式名字叫做 C 类型的 `svn_delta_editor_t`，通常也被叫做增量编辑器（delta editor）。

Subversion 的增量编辑器展现了一些优秀程序设计中的特性，这也是程序员们所需要探求的东西。它非常自然地把问题沿着边界进行分解，这样任何一个在 Subversion 上设计

新功能的程序员都可以很容易知道应该在何时调用每个函数,以及这些函数的作用是什么。增量编辑器使得程序员能够很轻松地将程序的执行效率最大化(例如消除不必要的网络数据迁移)以及实现辅助任务(例如过程报告)的简单集成。在增量编辑器中最重重要的地方或许就是:在Subversion的多次改进和升级过程中,这个接口设计被证明是非常灵活的。

要证明良好设计会带来效率的提升,增量编辑器本身就是一个极好的示例,它的创建是仅由一位程序员在几个小时之内完成的(当然这个人非常熟悉Subversion的问题及其代码库)。

要想知道是什么东西使得增量编辑器的设计如此漂亮,我们必须从分析它所要解决的问题开始。

版本控制与目录树的转换

在Subversion项目的早期,开发团队发现有一个任务需要反复地执行:用尽可能少的东西来表达两个相似(通常也是相关的)目录树之间的差异。作为一个版本的控制系统,Subversion的目标之一就是记录对目录结构和单个文件的修改。事实上,Subversion服务器端的仓库(repository)基本上是围绕着目录版本(directory versioning)来设计的。仓库只是在目录树变化过程中的一系列快照。对于每个提交到仓库的变化集合,都会创建一个新的目录树,从而与前面的目录树明确地区分开来:在新目录树中包括的变化不会出现在其他的目录树中,而在新目录树中没有发生变化的部分则会与原来的目录树共享存储数据,等等。目录树的每个连续版本都通过一个单调递增的整数来标识,这个唯一的标识被称之为修订号(revision number)。

我们可以把仓库看作是一个修订号的数组,并且这个数组的容量能够被扩展到无穷大。按照惯例,修订号0通常表示空目录。在图2-1中,在修订号1下面挂有一个目录树(通常是仓库的初始输入内容),并且现在还没有提交其他的修改。方框表示的是在这个虚拟文件系统中的节点:每个节点或者是一个目录(在右上角标有DIR),或者是一个文件(标有FILE)。

当我们修改文件*tuna*的时候,会发生什么情况?首先,我们将创建一个包含最新文本内容的新文件节点。此时,新的节点还没有被关联到任何其他节点。如图2-2所示,它只是被挂在外面,没有名字。

接下来,我们将创建这个文件所在父目录的一个新修订节点。如图2-3所示,此时子图仍然没有被连接到修订号数组。

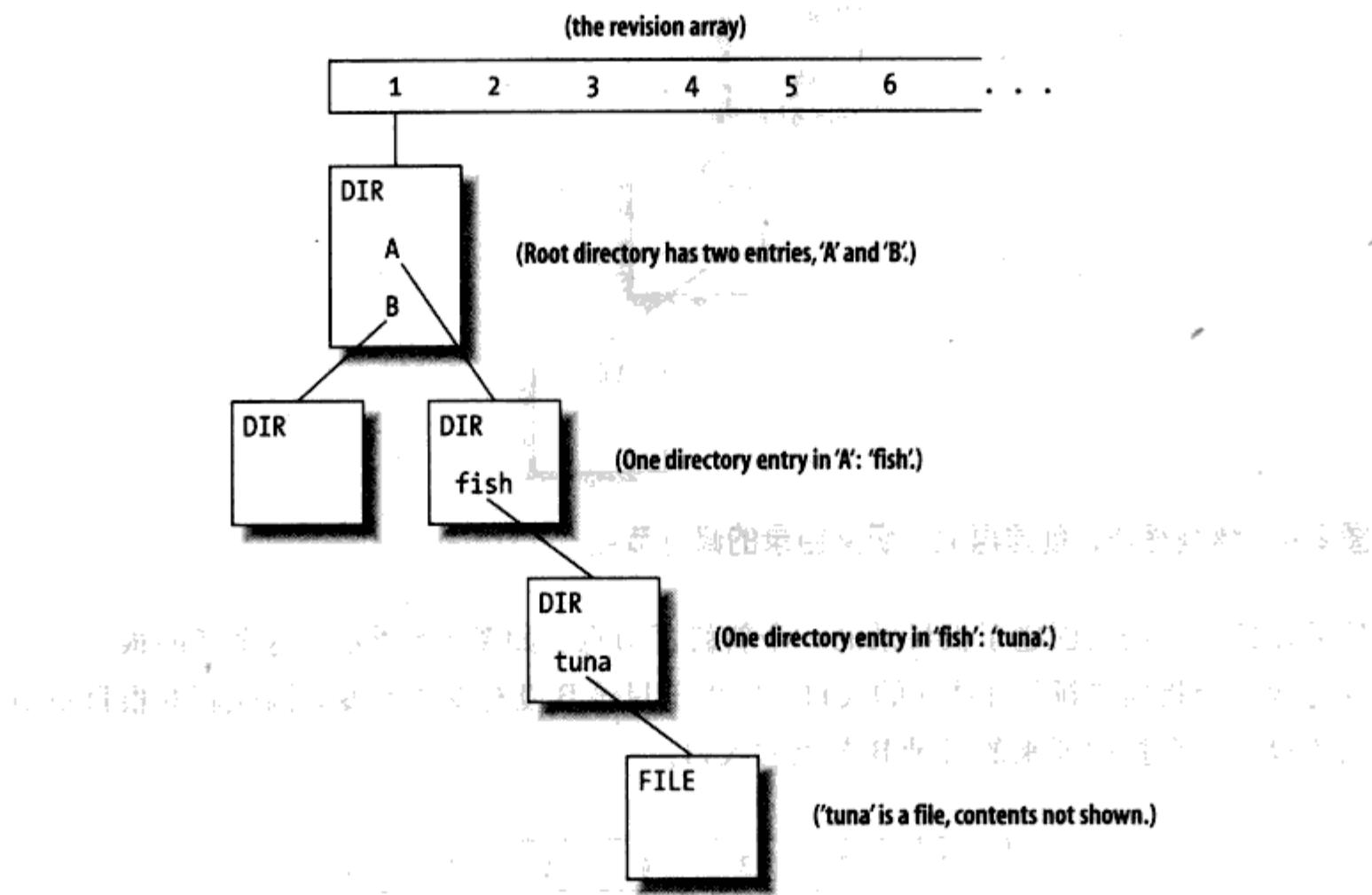


图 2-1：修订号的概念视图



图 2-2：新的节点刚刚被创建

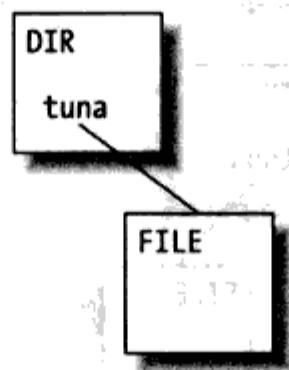


图 2-3：创建新的父目录

我们继续下去，再创建上一级父目录的修订节点（图 2-4）。

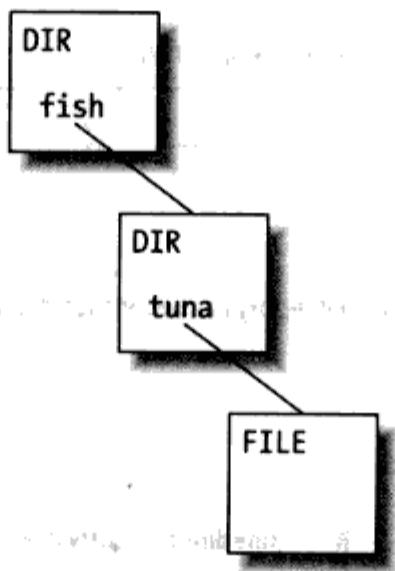


图 2-4：继续移动，创建再上一级父目录的修订节点

最后在顶部，我们创建了根目录的一个新修订节点，如图 2-5 所示。这个新的根目录节点包含一个指向“新”目录 A 的入口，但由于目录 B 没有发生改变，因此新的根目录节点还包含一个指向原来的目录 B 节点的入口。

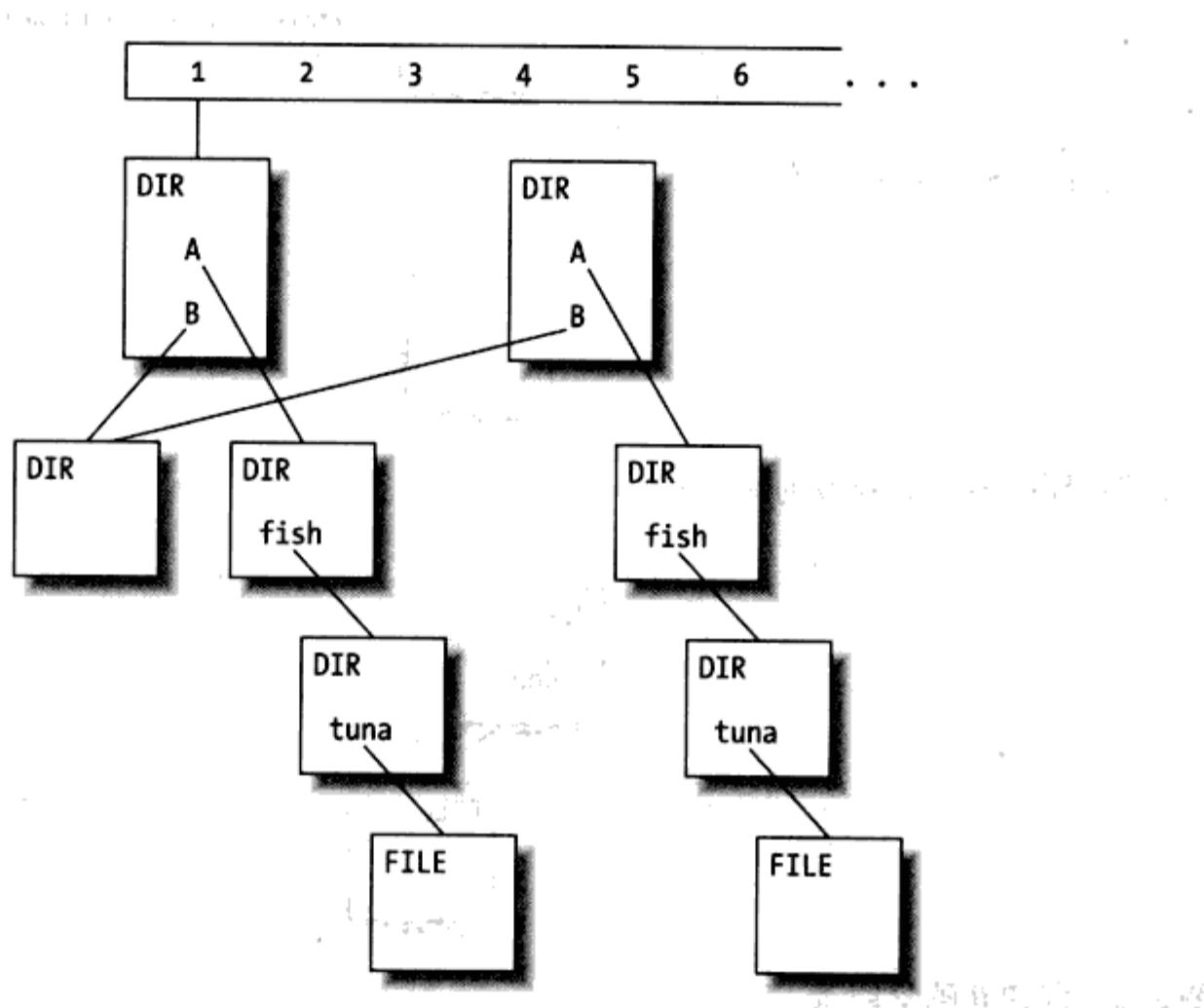


图 2-5：完整的新目录树

既然所有新的节点都已经被创建出来了，我们现在就可以把新的目录树链接到修订号历史数组中下一个有效的修订号上，从而来完成“冒泡（bubble up）”过程，这样就使得新的修改对于仓库用户来说是可见的。因此，新的目录树变成了修订号 2。

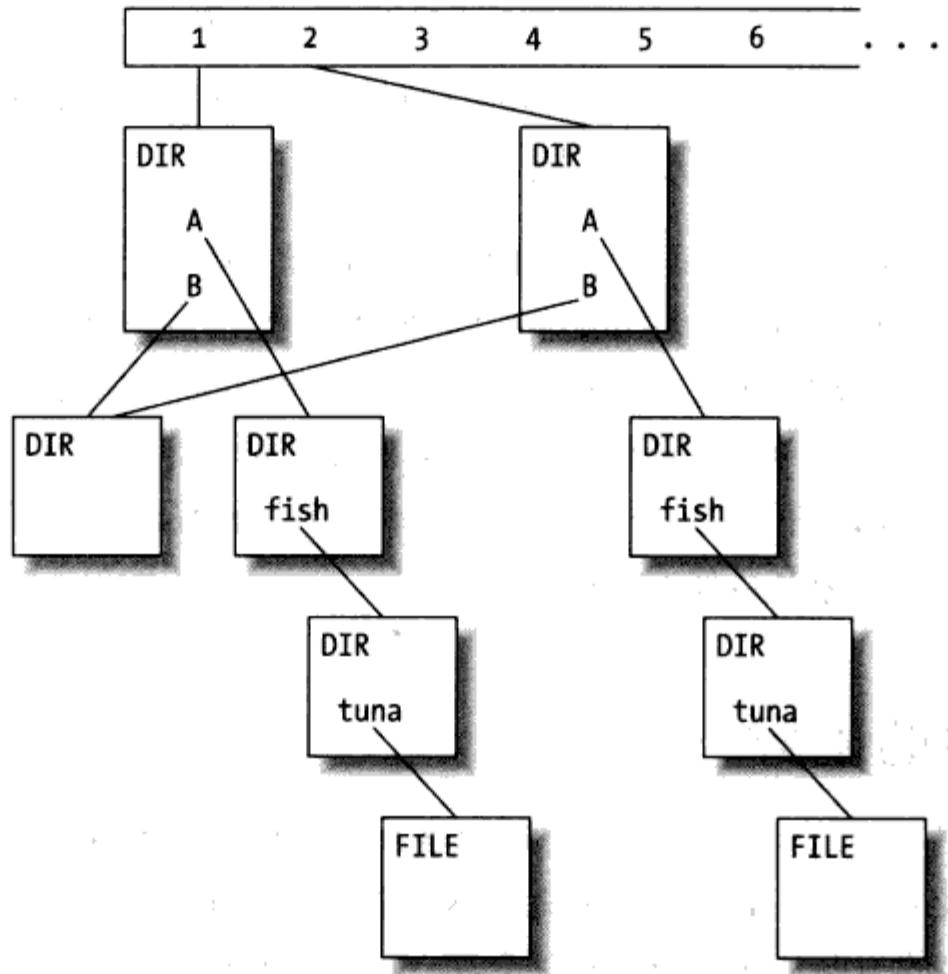


图 2-6：完成版本：链接到新的目录树

这样，在仓库中的每个修订号都指向惟一的目录树根节点，并且这棵目录树与其前者之间的差异就在于新的修订中提交的变化。为了知道这些变化，程序需要同时处理这两棵树，以找出哪些入口指向不同的位置。（为了简单起见，我省略了一些技术细节，例如压缩原来的节点作为与新版本之间的差异，从而节约存储空间等技术。）

在这个目录树版本模型中包含了学习本章主要内容（即我们在随后将会看到的增量编辑器）的所有背景知识，它有着非常漂亮的特性，我认为这是使得它在本章中作为漂亮代码示例的原因。这个模型有一些非常吸引人的功能，分别是：

易读性

要想找到文件 */path/to/foo.txt* 的第 *n* 个修订，可以直接跳到修订号 *n*，然后沿着目录树搜索下去，直到 */path/to/foo.txt*。

写操作不会妨碍读操作

当写操作在创建新的节点时，将在顶部进行操作，而并发的读操作无法看到这个工作。只有当写操作最终“链接”到仓库中的一个修订号时，读操作才能看到新的目录树。

用版本来标识目录树结构

在不同的修订中，每棵目录树的结构都将被保存。文件和目录的重命名、增加以及删除等操作都将成为仓库历史记录的一部分。

如果 Subversion 只是一个仓库，那么我们的讨论就可以到此为止。然而，Subversion 还包含一个客户端：工作副本（working copy），即用户检出（check out）的某个目录树的副本以及用户在这个副本上已经进行的但却还没有提交的本地修改。（事实上，工作副本通常并不是某一个简单的修订树；而是包含来自不同修订树的混合节点。就目录树的转换而言，这个区别并不重要。因此，我们在本章中只需假设工作副本就是代表某个修订树，但并不一定需要是最新的修订树。）

表达目录树的差异

在 Subversion 中最常见的操作就是在服务器/客户两端传输目录树之间发生的变化：当进行更新操作以接收其他的变化时，传输的方向是从仓库到工作副本，当程序员提交自己的修改时，传输的方向是从工作副本到仓库。对于许多其他的操作来说，例如差异比较，切换到一个分支，把变化从一个分支融合到另一个分支等，如何表达两棵树之间的差异同样是很关键的。

显然，如果在 Subversion 中有两个不同的接口，例如一个接口用于从服务器到客户端的传输，而另一个接口用于从客户端到服务器的传输，那么这种设计将很愚蠢。因为这两种情况中的基本任务都是相同的。目录树的差异应该只包含差异本身，而并不涉及差异在网络上的传输方向或者用户对其进行的操作。然而，我们发现要找到一种自然的方法来表达目录树的差异是一件非常困难的工作。此外还有一些复杂的问题，例如 Subversion 支持多种网络协议以及多种后台（backend）存储机制；我们需要的接口应该是在所有系统中看上去都是一致的。

我们最初所提出的所有接口都无法令人满意。在这里我对这些接口进行讨论，这些接口的共同之处就在于它们都会带来一些问题，而这些问题都无法找到具有说服力的解答。

例如，在这些提出来的解决方案中，大部分都是把发生变化的路径作为字符串来传输，要么全路径，要么是部分路径。那么，传输路径的顺序应该是什么？深度优先？宽度有限？随机顺序？字母顺序？目录与文件是否使用不同的命令？最重要的是，每个表达差异的命令如何来知道它是否属于包含所有修改的整体操作的一部分？在 Subversion 中，整个目录树操作都是用户可见的，如果编程接口没有很好地匹配这些概念，那么我们肯定需要编写大量脆弱的代码来进行弥补。

带着巨大的沮丧，我和另一个开发员 Ben Collins-Sussman，开车从芝加哥来到印第安纳州的伯明顿向 Jim Blandy 征询建议，Jim Blandy 是最早提出 Subversion 的仓库模型的，并且在我们的印象中，他在设计方面有着很强的能力。当我们向他描述在研究传输目录树的差异中所采用的各种不同方法时，Jim 很安静地听着，并且随着我们不断地谈论，他的表情开始变得越来越严肃。当我们总算说完了之后，他首先坐了一会，然后礼貌地请我们暂时离开一会，好让他思考一下。于是，我就穿上慢跑鞋跑步去了；而 Ben 则到另外一个房间里读书去了或者做了些其他的事情。这就是我们当时的情形。

当我跑步回来并沐浴之后，Ben 和我回到了 Jim 的办公室，他给我们展示了他所想到的东西，基本上也就是 Subversion 今天的样子；虽然在这几年中发生了许多变化，但其基本架构并没有改变。

增量编辑器接口

以下是增量编辑器接口的一个稍微简化的版本。我省略了与复制和重命名相关的部分、与 Subversion 属性（包括版本元数据，它们在本章中并不重要）相关的部分以及处理其他一些特定于 Subversion 记录的部分。不过，你可以到网址: http://svn.collab.net/repos/svn/trunk/subversion/include svn_delta.h 上来了解增量编辑器的最新版本。本章的内容基于 http://svn.collab.net/viewvc/svn/trunk/subversion/include svn_delta.h?revision=21731 上的 r21731 (即修订号为 21731)。

即使按照简化形式来理解这个接口，你还需要预先知道一些有关 Subversion 的术语：

池 (Pool)

池是分配好的内存池缓冲区，它能够允许同时释放大量的对象。

svn_error_t

返回类型 svn_error_t 只是表示这个函数返回一个指向 Subversion 错误对象的指针，如果函数调用成功则返回一个空指针。

文本增量

文本增量是指一个文件在两个不同版本之间的差异；你可以把文本增量看作为一个补丁，用来从文件的某个版本中生成其他的版本。在 Subversion 中，文件的“文本”被认为是二进制数据——无论文件是普通的文本，还是音频数据、影像或者其他的东西。文本增量被表示为固定大小的窗口，并且在每个窗口中包含了一块二进制数据。这样，Subversion 的最大内存消耗量就与单个窗口的大小成正比，而不是补丁的总体大小（在某些情况下，例如影像文件，这可能是非常庞大的）。

窗口处理器

一个将文本增量数据窗口应用到目标文件的函数原型。

Baton

一个 void* 数据结构，用来把运行环境传递给回调函数。在其他的 API 中有时也被叫做 void* ctx, void *userdata 或者 void *closure。Subversion 把它们都叫做“Baton (接力棒)”，因为它们在函数中进行传递时就像接力赛中的接力棒一样。

接口文件的开头部分是一个介绍，可以使阅读代码的程序员能够首先在脑海里首先有一个正确的框架。自从 Jim Blandy 在 2000 年 8 月份写下这段介绍以来，几乎就没有修改过，因此这个介绍经受住了时间的考验：

```
/** 目录树增量的研究
 *
 * 在 Subversion 中，我们知道目录树增量有着各种不同的生产者和使用者。
 *
 * 在处理 'commit' 命令时：
 * 客户端将检查它的工作副本数据，然后生成一个目录树增量来描述将要提交的修改。
 * 客户端网络库接收这个目录树增量，并且把它放在一系列的网络请求中通过线缆发送出去。
 * 服务器将接收这些请求，并且生成一个目录树增量——如果顺利的话，这个目录树增量将与客户端生成的目录树增量是一致的。
 * Subversion 服务器模块将处理这个增量并且向文件系统提交一个恰当的事务。
 *
 * 在处理 'Update' 命令时，处理过程是相反的：
 * Subversion 服务器模块与文件系统进行会话，并且生成一个目录树增量来描述将要发送给客户端进行更新的工作副本。
 * 服务器将处理这个目录树增量，并且生成一个表示正确修改的响应。
 * 客户端网络库将收到这个响应，并且生成一个目录树增量——如果顺利的话，这个目录树增量与 Subversion 服务器上生成的目录树增量是一致的。
 * 工作副本模块将处理这个目录树增量，并且对工作副本进行相应的修改。
 *
 * 最简单的方法就是用简单的数据结构来表示目录树增量。在进行更新时，服务器将构造一个增量数据结构，并且工作副本模块将把这个结构应用于工作副本上；网络层的任务只是简单地通过网络完整地传输这个结构。
 *
 * 然而，我们知道这些增量有时候会非常巨大，以至于无法填充到普通工作站的交换区域中。例如，在检出一个 200 Mb 的源目录树时，整个源目录树都是由一个单独的目录树增量来表示。因此，在 Subversion 中一个很重要的任务就是处理那些由于过于庞大而无法在交换中一次性进行填充的目录树增量。
 *
 * 因此，我们并没有使用简单的结构来表示目录树增量，而是为使用者定义了一种标准的方式，只要生产者产生了任何的目录树增量，使用者就立即进行处理。svn_delta_editor_t 结构是一组由目录树增量的使用者定义的回调函数，并且由增量生产者来调用。回调函数的每次调用都将处理一小部分增量——例如文件内容的变化，某个东西被重命名等。
 */
```

接下来是一段很长的正式文档说明，以及接口定义本身，这个接口实际上是一个回调函数表，并且函数的调用顺序也部分地进行了限制：

```
/** 这是一个包含许多回调函数的结构，增量产生源在产生增量时会调用这些函数。
```

```

*
* 函数用法
* =====
*
* 以下是如何通过这些函数来表示目录树增量。
*
* 在这个结构中声明的回调函数将由增量使用者来实现，而增量生产者将调用这些函数。因此，这就相当于调用者（即生产者）将目录树增量的数据压入到被调用者（使用者）。
*
* 首先，使用者将提供 edit_baton，这是整个增量编辑中的全局 baton。
*
* 接下来，如果要表示任何目录树增量，生产者应该把 edit_baton 传递给 open_root 函数，以得到一个表示工作目录树所在根目录的 baton。
*
* 大多数回调函数的工作方式都是很简单的：
*
*   delete_entry
*   add_file
*   add_directory
*   open_file
*   open_directory
*
* 在上面的每个函数中都包括一个目录 baton 参数，用来表示发生变化的目录，以及一个路径参数，这个参数表示要修改的文件，子目录或者目录入口的路径（相对于编辑操作所在根目录的路径）。编辑器通常会把这个相对路径和某个存储在 edit-baton 中的基路径结合起来（例如一个 URL，或者是文件系统中的一个位置）。
*
* 由于每个函数调用都需要一个父目录 baton，包括 add_directory 和 open_directory，那么我们从哪里来得到最初的目录 baton 以开始操作呢？ open_root 函数将返回变化所在的顶层目录 baton。通常，生产者在能够执行任何操作之前，都需要首先调用编辑器的 open_root 函数。
*
* open_root 将返回发生变化的目录树的根目录 baton，而 add_directory 和 open_directory 回调函数则为其他的目录提供 baton。与前面的回调函数一样，它们得到一个 parent_baton 和一个相对路径，然后为将要创建或者修改的子目录返回一个新的 baton——child_baton。接下来，生产者将使用 child_baton 在这个子目录中做进一步的修改。
*
* 这样，如果已经存在 'foo' 和 'foo/bar' 这些子目录，那么生产者就可以通过调用下面的函数来创建一个新文件 'foo/bar/baz.c'。
*
*   - open_root () --- 为顶级目录生成一个 baton。
*
*   - open_directory (root, "foo") --- 为 'foo' 生成一个 baton f
*
*   - open_directory (f, "foo/bar") --- 为 'foo/bar' 生成一个 baton b
*
*   - add_file (b, "foo/bar/baz.c")
*
* 当生产者完成了对目录的修改时，它将调用 close_directory 函数。使用者可以在此时进行任何必要的清理工作，并且释放 baton 的存储空间。
*
* add_file 和 open_file 这两个回调函数都会返回一个 baton 来表示被创建或者被修改的文件。然后，这个 baton 可以被传递给 apply_textdelta 来改变文件的内容。当生产者完成对文件的修改后，它将调用 close_file，此时使用者将执行清理工作并释放 baton。

```

```

* 函数调用的顺序
* =====
*
* 生产者在使用 baton 的方式上存在着 5 个限制:
*
* 1. 在任意的目录入口上, 生产者最多只能调用一次 open_directory, add_directory,
  open_file, add_file。在任意的目录入口上最多只能调用一次 delete_entry, 并且随后可
  以在相同的目录上调用 add_directory 或者 add_file。如果在某个目录上调用了
  oepn_directory, add_directory, open_file 或者 add_file 之后, 就不能再在这个目
  录上调用 delete_entry。
*
* 2. 只有在某个目录 baton 关闭了其所有的子目录 baton 之后, 产生者才可以关闭这个目录
  baton。
*
* 3. 当生产者调用 open_directory 或者 add_directory 时, 它必须指定在当前打开的目录
  baton 中最近打开的 baton。换句话说, 生产者不能使得两个同级目录 baton 同时处于打开的
  状态。
*
* 4. 当生产者调用 open_file 或者 add_file 时, 在执行任何其他的文件调用或者目录调用之
  前, 必须提交对文件进行的修改 (使用 apply_textdelta), 并且执行 close_file 调用。
*
* 5. 当生产者调用 apply_textdelta 时, 那么在执行任何其他的 svn_delta_editor 调用之
  前, 必须首先调用所有的窗口处理器 (包括 NULL 窗口)。
*
* 因此, 生产者需要使用目录 baton 和文件 baton, 就好像它正在对目录树进行一个深度优先的遍
  历。
*
* 内存池的用法
* =====
*
* 许多编辑器函数都可以根据编辑器的“驱动器 (driver)”所确定的序列被调用多次。这个驱动
  器负责创建一个内存池, 以用于编辑器函数的每次迭代操作, 并且在两次迭代之间将清空这个内
  存池。在每次函数调用时, 驱动器都会传递合适的内存池。
*
* 由于要满足按照深度优先的方式来调用编辑器函数的需求, 驱动器通常习惯于嵌套内存池。然而,
  当顶级项被清除时, 只有一种安全功能能够确保与更深项相关的内存池也被清除。接口对于传递
  给这些函数的内存池的结构既没有任何特殊的假设, 也不作任何特殊的要求。
*/
typedef struct svn_delta_editor_t
{
    /** 将 *root_baton 设置为一个表示顶层目录变化的 baton。(这里指的是被修改的子树的顶层
        目录, 而并不一定是文件系统的根目录)。像任何其他的目录 baton 一样, 在完成修改时, 生产
        者应该在 root_baton 上调用 close_directory 函数。
    */
    svn_error_t *(*open_root)(void *edit_baton,
                           apr_pool_t *dir_pool,
                           void **root_baton);

    /** 删除目录入口 path, 即 parent_baton 目录的子目录
    */
    svn_error_t *(*delete_entry)(const char *path,
                               void *parent_baton,
                               apr_pool_t *pool);
}

```

```

/** 我们将要增加一个新的子目录 path。我们将把回调函数在
 * child_baton 中存储的这个值来作为在新子目录中进行修改的 parent_baton。
 */
svn_error_t * (*add_directory)(const char *path,
                               void *parent_baton,
                               apr_pool_t *dir_pool,
                               void **child_baton);

/** 我们将在子目录（由 parent_baton 标识的目录）中进行修改。这个子目录是由 path 指
 * 定的。回调函数必须在 child_baton 中存储一个值，并且这个值将被用作为在这个子目录
 * 中进行后续修改的 parent_baton。
*/
svn_error_t * (*open_directory)(const char *path,
                                void *parent_baton,
                                apr_pool_t *dir_pool,
                                void **child_baton);

/** 我们已经完成了对 dir_baton（这个值是由 add_directory 或者 open_directory
 * 设置的）子目录的处理。我们将不再使用这个 baton，因此它所使用的所有资源现在都可
 * 以被释放。
*/
svn_error_t * (*close_directory)(void *dir_baton,
                                 apr_pool_t *pool);

/** 我们将增加一个新的文件 path。回调函数可以在 **file_baton 中为这个新文件存
 * 储一个 baton；它存储的任何值都将被传递给 apply_textdelta。
*/
svn_error_t * (*add_file)(const char *path,
                         void *parent_baton,
                         apr_pool_t *file_pool,
                         void **file_baton);

/** 我们将对文件 path 进行修改，其中 path 是驻留在由 parent_baton 表示的目录中。
 * 这个回调函数将在 **file_baton 中为这个新文件存储一个 baton；它所存储的任意值都
 * 将被传递给 apply_textdelta。
*/
svn_error_t * (*open_file)(const char *path,
                          void *parent_baton,
                          apr_pool_t *file_pool,
                          void **file_baton);

/** 应用一个文本增量，产生文件的新修订版本。
 */
* file_baton 表示我们正在创建或者更新的这个文件，以及它所基于的父文件；这是由之
* 前的 add_file 或者 open_file 回调函数设置的。
*
* 这个回调函数应该把 *handle 设置为一个文本增量窗口处理器；然后在收到后续的文本增
* 量时，将在这些文本增量上来调用 *handle。这个回调函数应该把 *handler_baton 设
* 置为将要传递给 *handle 的 baton 参数值。
*/
svn_error_t * (*apply_textdelta)(void *file_baton,
                                 apr_pool_t *pool,
                                 svn_txdelta_window_handler_t
                                 *handler,
                                 void **handler_baton);

/** 我们处理完了 file_baton（由 add_file 或者 open_file 设置的）文件。我们将不再
 * 使用这个 baton，因此它所使用的所有资源都将被释放。
*/

```

```
 */
svn_error_t *(*close_file)(void *file_baton,
                           apr_pool_t *pool);
/** 所有的增量处理已经完成了。用整个编辑的 edit_baton 来调用这个函数。
 */
svn_error_t *(*close_edit)(void *edit_baton,
                           apr_pool_t *pool);

/** 编辑器的驱动器决定退出。编辑器将在必要的时候执行清理工作。
 */
svn_error_t *(*abort_edit)(void *edit_baton,
                           apr_pool_t *pool);
} svn_delta_editor_t;
```

但这是艺术吗

我不能说这个接口的漂亮性对我来说是一目了然的。我也无法确定这对于 Jim 来说是不是同样简单明了；他可能只是想让 Ben 和我早点离开他的房子。不过，他考虑这个问题已经很长一段时间了，而且他也非常了解目录树结构行为。

第一件令人印象深刻的事情就是在增量编辑器中选择了约束：虽然没有诸如目录树的编辑应该按照深度优先顺序（或者任何一种顺序）这样的要求，但这个接口通过 baton 之间的关系迫使编辑按照深度优先的顺序来进行。这就使得接口的用法和行为是更可预测的。

第二件事情就是在整个编辑操作中隐式地传递了运行环境，这也是通过 baton 来实现的。文件 baton 可以包含一个指向父目录 baton 的指针，而在该目录 baton 中又可以包含一个指向其父目录 baton 的指针（对于编辑的根目录来说，它的父目录 baton 指针为空），并且所有的 baton 都可以包含一个指向全局编辑 baton 的指针。虽然一个独立的 baton 或许是一个可抛弃的对象——例如，当文件被关闭时，这个文件的 baton 将被销毁——但是任何一个 baton 都可以访问全局的编辑运行环境，在这个运行环境中将包含诸如客户端将要更新的修订号等信息。这样，baton 的功能就十分丰富：它们不仅为编辑中的各个部分提供了作用域（也就是生存期，因为一个 baton 的生存期与为之分配的内存池的生存期是一样的），而且它们还传递了全局运行环境。

第三件重要的事情就是这个接口在表达目录树变化的不同子操作之间提供了清晰的边界。例如，打开一个文件只是表示这个文件在两个目录树之间发生了变化，但却并不给出变化的细节；而调用 apply_textdelta 则给出了变化的细节，但如果你不想了解这个变化，就可以不必调用 apply-textdelta。类似地，打开一个目录表示这个目录或者在目录中的内容发生了改变，但如果你不想知道这些变化的具体内容，那么可以关闭这个目录并且继续进行操作。这些边界要归功于接口的流性化（streaminess），正如在这个

介绍中所提到的：“……我们并没有用简单的结构来表示目录树增量，而是为使用者定义了一种标准的方式，只要生产者产生了任何的目录树增量，使用者就立即进行处理。”原来的做法是仅对最大的数据块（也就是说，文件差异）进行流化处理，但是增量编辑器接口则是彻底地对整个目录树增量进行了流化处理，这就使生产者和使用者能够同时准确地控制内存使用、过程报告以及可中断性。

只有当我们开始把新的增量编辑器应用于各种不同的问题时，这些功能才开始显示出它们的价值。例如，我们想要实现的一个操作就是目录树变化的摘要：这指的是给出两棵目录树之间的大致差异，而不是细节的内容。例如，有的用户可能只想知道自从他检出工作副本中的文件后，有哪些文件在仓库中被修改了，但他并不需要准确地知道这些修改是什么。

以下是这种工作机制的一个简化版本：客户端告诉服务器这个工作副本是基于哪个修订树的，然后服务器将通过增量编辑器来告诉客户端在修订树和最新版本之间的差异。服务器是生产者，而客户端是使用者。

我们在本章前面的仓库中构建了对/A/fish/tuna的修改从而产生了修订号2，让我们再来看看这如何看上去像一组编辑器函数调用，其中这些调用由服务器发送给修订号仍然为1的客户端的编辑器函数调用。在if块中有2/3的代码是让我们判断这是一个摘要编辑还是一个“给我所有细节”的编辑。

```
svn_delta_editor_t *editor
void *edit_baton;

/* 在实际情况中，这将是一个传入的参数。 */
int summarize_only = TRUE;

/* 在实际情况中，这些变量都将被声明在子例程中，这样它们的生命期就是由子例程的堆栈来控制，就像它们所指向的对象是由目录树编辑来控制一样。 */
void *root_baton;
void *dir_baton;
void *subdir_baton;
void *file_baton;

/* 类似的，这些是子内存池，而不是一个顶级内存池。 */
apr_pool_t *pool = svn_pool_create( );

/* 对于每个增量编辑器接口的使用都是从请求能够实现你的需求的特定编辑器开始的，例如，对网络上编辑操作进行流化，并将其应用到工作副本等等要求。 */
Get_Update_Editor(&editor, &eb,
                  some_repository,
                  1, /* source revision number */ 源修订号
                  2, /* target revision number */ 目标修订号
                  pool);
```

```

/* 现在，我们开始启动编辑器。在实际情况中，这个调用序列将是由代码动态生成的，这些代码
将在这两个仓库目录树之间进行遍历，并且在适当的时候引发 editor->foo() 调用。 */

editor->open_root(edit_baton, pool, &root_baton);
editor->open_directory("A", root_baton, pool, &dir_baton);
editor->open_directory("A/fish", dir_baton, pool, &subdir_baton);
editor->open_file("A/fish/tuna", subdir_baton, pool, &file_baton);

if (! summarize_only)
{
    svn_txdelta_window_handler_t window_handler;
    void *window_handler_baton;
    svn_txdelta_window_t *window;

    editor->apply_textdelta(file_baton, pool
                            apr_pool_t *pool,
                            &>window_handler,
                            &>window_handler_baton);
    do {
        window = Get_Next_TextDelta_Window(...);
        window_handler(window, window_handler_baton);
    } while (window);
}

editor->close_file(file_baton, pool);
editor->close_directory(subdir_baton, pool);
editor->close_directory(dir_baton, pool);
editor->close_directory(root_baton, pool);
editor->close_edit(edit_baton, pool);

```

正如上面这个示例所给出的，变化的摘要和变化的完整版本沿着增量编辑器接口的边界很自然地被分开了，这使得我们可以将同样的代码用于两个不同的目的。如果在这个示例中，两个变化刚好是邻接（修改 1 和修改 2）的，那么可以不需要这么做。同样的方法还可以应用于任意的两棵目录树，即使在它们中间存在着大量的变化，例如像某个工作副本已经很长时间没有进行更新的情况。即使两棵目录树处于逆序的情况——也就是说，新版本在旧版本的前面——也能够进行这种操作。这对于恢复变化来说是非常有用的。

像体育比赛一样抽象

当我们需要在同一个目录树编辑中进行两个或者多个不同的操作时，就可以显示出增量编辑器灵活性的另一个方面。最早的情况之一就是处理取消操作。当用户请求中断正在进行的更新操作时，信号处理器将捕获这个请求并且发送一个标志，然后在操作的不同时刻，我们将检测这个标志，并且如果发现这个标志被设置了，那么操作将会很干净地退出。事实证明在许多情况下，从操作中退出的最安全位置就是下一个入口或者编辑器函数调用的边界。这不仅对于那些在客户端不需要执行 I/O 的操作（例如修改摘要以及

差异比较)来说是显而易见的,而且对于许多需要访问文件的操作来说也是正确的。毕竟,在更新操作中的大多数工作只是简单地写出数据,并且即使用户中断了整个更新操作,那么当检测到中断请求时,完成写入操作或者干净地取消正在操作中的文件都是有意义的。

然而,如何实现对标志的检测?我们可以把这些标志硬编码到在Get_Update_Editor()中返回(通过引用返回)的增量编辑器中。显然这是一种非常不好的做法:增量编辑器可能是一个从其他代码中调用的库函数,而这些代码或许希望使用一种完全不同的取消操作的检测形式,或者根本就不检测取消操作。

稍好一点的解决方案就是传递一个取消检测的回调函数,并且将 baton 关联到 Get_Update_Editor()。所返回的编辑器将定期在这个 baton 上调用回调函数,并且根据返回值来判断是像往常一样继续操作还是较早地返回(如果回调函数为空,那么将永远不会被调用)。但这种方式同样是不理想的,检测取消操作实际上是一个附属目标:在进行更新的时候,你可能想检测取消,也可能不想检测取消,但在任何情况下,它应该对更新过程本身的工作方式没有任何影响。理想的情况是,这两个功能不应该在代码中纠缠在一起,尤其是,根据我们的结论,在大多数情况下,操作不需要对检测取消有着精确的控制——编辑器调用边界能够很好地完成这个工作。

取消操作只是一个与目录树增量编辑相关的辅助任务示例之一。当从客户端向服务器传输变化的同时跟踪提交的目标,在向用户报告更新或者提交的进度,以及在许多其他的情况下,我们将面对类似的问题。因此,我们很自然地需要寻求一种方式能够把这些附属的东西抽象出来,这样核心代码将不会与它们混杂在一起。事实上,这个问题看上去似乎很难,以至于在最初的时候被过度地抽象化了:

```
/** 将 editor_1 及其 baton 与 editor_2 及其 baton 组合在一起。
 *
 * 在 new_editor 中返回一个新的编辑器 (在内存池中分配的), 其中每个函数 fun 用相应的
 * baton 来调用, 首先调用 editor_1->fun, 然后再调用 editor_2->fun。
 *
 * 如果 editor_1->fun 返回错误, 那么这个错误将在 new_editor->fun 中返回, 而
 * editor_2->fun 将永远不会被调用; 否则 new_editor->fun 的返回值将和
 * editor_2->fun 的返回值是相同的。
 * 如果编辑器函数是空, 那么将永远都不会被调用, 这并不是一个错误。
 */
void
svn_delta_compose_editors(const svn_delta_editor_t **new_editor,
                           void **new_edit_baton,
                           const svn_delta_editor_t *editor_1,
                           void *edit_baton_1,
                           const svn_delta_editor_t *editor_2,
                           void *edit_baton_2,
                           apr_pool_t *pool);
```

虽然这有些走得太远了——我们将马上来看看为什么——我仍然发现这是编辑器接口漂亮性的一个证据。复合编辑器的行为是可预测的，它们使代码保持了很好的整洁性（因为所有的编辑器函数都无需关心在它之前或者之后并行调用的编辑器函数的一些细节），并且它们通过了关联测试：你可以把一个复合编辑器和其他的编辑器组合起来，其中每个功能能够执行。即使这些编辑器可能对数据进行完全不同的处理，但它们所执行操作的基本形式都是一致的。

你或许会说，我遗漏了编辑器组合的纯优雅性。但最终它比我们需要的更为抽象。我们在最初的时候使用了复合编辑器来实现大多数功能，但后来重新修改为使用传递给编辑器创建例程的自定义回调函数。虽然附属的行为与编辑器调用边界是一致的，但它们通常在所有的调用边界上都是不合适的，或者说在大多数情况下是不合适的。这么做的结果就是带来了很高的工作故障率，在建立起一个完整的并行编辑器时，我们将误导阅读代码的程序员：附属的行为的调用频率要高于它们的实际需要。

我们曾经对编辑器组合研究得过深，因此后来又回退了一些，不过在我们真正需要的时候，还是可以通过手工的方式来实现。在目前的 Subversion 中，取消操作是用手工的组合来完成的。取消检测编辑器的构造函数将另一个编辑器——核心操作编辑器——作为参数。

```
/** 将 *editor 和 *edit_baton 设置为包装 wrapper_editor 和 wrapped_baton 的取消操作
 * 编辑器。
 *
 * 当编辑器的每个函数被调用时，将使用 cancel_baton 来调用 cancel_func。如果
 * cancel_func 返回 SVN_NO_ERROR 时，将继续调用下一个包装函数。
 *
 * 如果 cancel_func 为空时，将 *editor 设置为 wrapped_editor 并且将
 * *editor_baton 设置为 wrapped_baton。
 */
svn_error_t *
svn_delta_get_cancellation_editor(svn_cancel_func_t cancel_func,
                                  void *cancel_baton,
                                  const svn_delta_editor_t
                                  *wrapped_editor,
                                  void *wrapped_baton,
                                  const svn_delta_editor_t **editor,
                                  void **edit_baton,
                                  apr_pool_t *pool);
```

我们还通过类似的手工组合过程实现了一些条件调试输出。其他的附属行为——主要是进度报告，事件通知以及目标统计等——都是通过传递给编辑器构造函数的回调函数来实现的，并且由编辑器（如果非空的话）在这些它们需要的情况下进行调用。

编辑器接口在 Subversion 的代码中提供了一种很强的一致性。虽然要称赞这个可能产生

过度抽象的 API 看上去似乎有些奇怪，但这只是完美地解决目录树增量传输这个问题的一种副作用——它使得这个问题看上去非常容易处理，以至于我们想让其他的问题转化成这个问题！当它们并不适合时，我们可以后退，但编辑器的构造函数仍然提供了标准的地方来插入回调函数，并且编辑器的内部操作边界将有助于我们考虑何时调用这些回调函数。

结论

我想，这个 API，以及所有漂亮的 API 的真正能力在于它能够指导我们思考。在 Subversion 中，所有涉及目录树修改的操作现在都差不多可以被设计为相同的方式。这不仅节约了新手们在学习现有代码上所花费的时间，还为新的代码指定了需要遵循的模型，开发人员同样也得到了提示。例如，在将某个仓库的行为直接映射到另一个仓库的 svnsync 功能时——这个功能是在 2006 年被增加到 Subversion 中的，也就是在增量编辑器发明 6 年之后——使用了增量编辑器接口来传输这个行为。这个功能的开发人员不仅节约了设计一种变化传输机制的时间，甚至还无需考虑它是否需要设计一种变化传输机制。而其他开发新代码的人员将会发现，它们一看到这些代码就会感到非常熟悉。

这些都是很重要的好处。正确的 API 不仅节约了学习时间，它还使得开发人员无需进行专门的辩论：长时间的讨论和充满争论的邮件列表将不会出现。这或许并不像纯技术或者美学漂亮之类的东西，但在有着许多参与人员并且人员有着固定流动率的项目中，使用这种接口无疑是明智的选择。

我从未编写过的最漂亮的代码

Jon Bentley

我曾经听一位大师级的程序员这样称赞到，“我通过删除代码来实现功能的提升。”而法国著名作家兼飞行家 Antoine de Saint-Exupéry 的说法则更具代表性，“只有在不仅没有任何功能可以添加，而且也没有任何功能可以删除的情况下，设计师才能够认为自己的工作已臻完美。”某些时候，在软件中根本就不存在最漂亮的代码，最漂亮的函数，或者最漂亮的程序。

当然，我们很难对不存在的事物进行讨论。本章将对经典 Quicksort（快速排序）算法的运行时间进行全面的分析，并试图通过这个分析来证明上述观点。在第一节中，我将首先根据我自己的观点来回顾一下 Quicksort，并为后面的内容打下基础。第二节的内容将是本章的重点部分。我们将首先在程序中增加一个计数器，然后通过不断地修改，从而使程序的代码变得越来越短，但程序的功能却会变得越来越强，最终的结果是只需要几行代码就可以使算法的运行时间达到平均水平。在第三节将对前面的技术进行小结，并对二分搜索树的运行开销进行简单的分析。最后的两节将给出学完本章得到的一些启示，这将有助于你在今后写出更为优雅的程序。

我编写过的最漂亮的代码

当Greg Wilson最初告诉我本书的编写计划时，我曾自问编写过的最漂亮的代码是什么。这个有趣的问题在我脑海里盘旋了大半天，然后我发现答案其实很简单：Quicksort 算法。但遗憾的是，根据不同的表述方式，这个问题有着三种不同的答案。

当我撰写关于分治（divide-and-conquer）算法的论文时，我发现 C.A.R. Hoare 的 Quicksort 算法（“Quicksort”，Computer Journal 5）无疑是各种 Quicksort 算法的鼻祖。这是一种解决基本问题的漂亮算法，可以用优雅的代码实现。我很喜欢这个算法，但我总是无法弄明白算法中最内层的循环。我曾经花两天的时间来调试一个使用了这个循环的复杂程序，并且几年以来，当我需要完成类似的任务时，我会很小心地复制这段代码。虽然这段代码能够解决我所遇到的问题，但我却并没有真正地理解它。

我后来从 Nico Lomuto 那里学到了一种优雅的划分（partitioning）模式，并且最终编写出了我能够理解的，甚至能够被证明的 Quicksort 算法。William Strunk Jr. 针对英语写作提出了“良好的写作风格即为简练”的经验法则，这同样适用于代码的编写，因此我遵循了他的建议，“省略不必要的字词”（来自《The Elements of Style》一书）。我最终将大约 40 行左右的代码缩减为十几行的代码。因此，如果要回答“你曾编写过的最漂亮代码是什么？”这个问题，那么我的答案就是：在我编写的《Programming Pearls, Second Edition》（Addison-Wesley）一书中给出的 Quicksort 算法。在示例 3-1 中给出了用 C 语言编写的 Quicksort 函数。我们在接下来的章节中将进一步地研究和改善这个函数。

示例 3-1：Quicksort 函数

```
void quicksort(int l, int u)
{
    int i, m;
    if (l >= u) return;
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
    quicksort(m+1, u);
}
```

如果函数的调用形式是 `quicksort(0, n-1)`，那么这段代码将对一个全局数组 `x[n]` 进行排序。函数的两个参数分别是将要进行排序的子数组的下标：`l` 是较低的下标，而 `u` 是较高的下标。函数调用 `swap(i, j)` 将会交换 `x[i]` 与 `x[j]` 这两个元素。第一次交换操作将会按照均匀分布的方式在 `l` 和 `u` 之间随机地选择一个划分元素（Partitioning Element）。

在《Programming Pearls》一书中包含了对 Quicksort 算法的详细推导以及正确性证明。在本章的剩余内容中，我将假设读者熟悉在《Programming Pearls》中所给出的 Quicksort 算法以及在大多数初级算法教科书中所给出的 Quicksort 算法。

如果你把问题改为“在你编写那些广为应用的代码中，哪一段代码是最漂亮的？”我的答案还是 Quicksort 算法。在我和 M. D. McIlroy 一起编写的一篇文章（"Engineering a sort function," Software-Practice and Experience, Vol. 23, No. 11）中指出了在原来 Unix qsort 函数中的一个严重的性能问题。随后，我们开始用 C 语言编写一个新排序函数库，并且考虑了许多不同的算法，包括合并排序（Merge Sort）和堆排序（Heap Sort）等算法。在比较了 Quicksort 的几种实现方案后，我们着手创建自己的 Quicksort 算法。在这篇文章中介绍了我们如何设计出一个比这个算法的其他实现要更为清晰，速度更快以及更为健壮的新函数——部分原因是由于这个函数的代码更为短小。Gordon Bell 的名言被证明是正确的：“在计算机系统中，那些最廉价，速度最快以及最为可靠的组件是不存在的。”现在，这个函数已经被使用了 10 多年的时间，并且没有出现任何故障。

考虑到通过缩减代码量所带来的好处，我最后以第三种方式来问自己在本章之初提出的问题。“你从没有编写过的最漂亮代码是什么？”我如何使用非常少的代码来实现大量的功能？答案还是和 Quicksort 有关，特别是对这个算法的性能分析。我将在下一节给出详细介绍。

事半功倍

Quicksort 是一种优雅的算法，这一点有助于对这个算法进行细致的分析。大约在 1980 年左右，我与 Tony Hoare 曾经讨论过 Quicksort 算法的历史。他告诉我，当他最初开发出 Quicksort 时，他认为这种算法太简单了，不值得发表，而且直到能够分析出这种算法的预期运行时间之后，他才写出了经典的“Quicksoft”论文。

我们很容易看出，在最坏的情况下，Quicksort 可能需要 n^2 的时间来对数组元素进行排序。而在最优的情况下，它将选择中值作为划分元素，因此只需 $n \lg n$ 次的比较就可以完成对数组的排序。那么，对于有 n 个不同值的随机数组来说，这个算法平均将进行多少次比较？

Hoare 对于这个问题的分析非常漂亮，但不幸的是，其中所使用的数学知识超出了大多数程序员的理解范围。当我为本科生讲授 Quicksort 算法时，许多学生即使在费了很大的努力也无法理解其中的证明过程，这令我非常沮丧。下面，我们将从 Hoare 的程序开始讨论，并且最后将给出一个与他的证明很接近的分析。

我们的任务是对示例3-1中的Quicksort代码进行修改，以分析在对各个元素值均不相同的数组进行排序时平均需要进行多少次比较。我们还将努力通过最短的代码、最短运行时间以及最小存储空间来获得最深的理解。

为了确定平均比较的次数，我们首先对程序进行修改以统计比较次数。因此，在内部循环进行比较之前，我们将增加变量 *comps* 的值（参见示例 3-2）。

示例 3-2：修改 Quicksort 的内部循环以统计比较次数

```
for (i = l+1; i <= u; i++) {  
    comps++;  
    if (x[i] < x[l])  
        swap(++m, i);  
}
```

如果用一个值 *n* 来运行程序，我们将会看到在程序的运行过程中总共进行了多少次比较。如果重复用 *n* 来运行程序，并且用统计的方法来分析结果，我们将得到 Quicksort 在对 *n* 个元素进行排序时平均使用了 $1.4 n \lg n$ 次的比较。

在理解程序的行为上，这是一种不错的方法。通过十三行的代码和一些实验可以反映出许多问题。这里，我们引用作家 Blaise Pascal 和 T. S. Eliot 的话，“如果我有越多的时间，那么我给你写的信就会越短。”现在，我们有充足的时间，因此就让我们来对代码进行修改，并且努力编写出更短（同时更好）的程序。

我们要做的事情就是提高这个算法的速度，并且尽量增加统计的准确度以及加深对程序的理解。由于内部循环总是会执行 *u*-1 次比较，因此我们可以通过在循环外部增加一个简单的操作来统计比较次数，这就可以使程序运行得更快一些。在示例 3-3 的 Quicksort 算法中给出了这个修改。

示例 3-3：Quicksort 的内部循环，将递增操作移到循环的外部

```
comps += u-1;  
for (i = l+1; i <= u; i++)  
    if (x[i] < x[l])  
        swap(++m, i);
```

这个程序会对一个数组进行排序，同时统计比较的次数。不过，如果我们的目标只是统计比较的次数，那么就不需要对数组进行实际地排序。在示例 3-4 中去掉了对元素进行排序的“实际操作”，而只是保留了程序中各种函数调用的“框架”。

示例 3-4：将 Quicksort 算法的框架缩减为只进行统计

```
void quickcount(int l, int u)  
{    int m;  
    if (l >= u) return;
```

```
m = randint(1, u);
comps += u-1;
quickcount(l, m-1);
quickcount(m+1, u);
}
```

这个程序能够实现我们的需求，因为 Quicksort 在选择划分元素时采用的是“随机”方式，并且我们假设所有的元素都是不相等的。现在，这个新程序的运行时间与 n 成正比，并且相对于示例 3-3 需要的存储空间与 n 成正比来说，现在所需的存储空间缩减为递归堆栈的大小，即存储空间的平均大小与 $\lg n$ 成正比。

虽然在实际的程序中，数组的下标（ l 和 u ）是非常重要的，但在这个框架版本中并不重要。因此，我们可以用一个表示子数组大小的整数（ n ）来替代这两个下标（参见示例 3-5）。

示例 3-5：在 Quicksort 代码框架中使用一个表示子数组大小的参数

```
void qc(int n)
{
    int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

现在，我们可以很自然地把这个过程整理为一个统计比较次数的函数，这个函数将返回在随机 Quicksort 算法中的比较次数。在示例 3-6 中给出了这个函数。

示例 3-6：将 Quicksort 框架实现为一个函数

```
int cc(int n)
{
    int m;
    if (n <= 1) return 0;
    m = randint(1, n);
    return n-1 + cc(m-1) + cc(n-m);
}
```

在示例 3-4、示例 3-5 和示例 3-6 中解决的都是相同的基本问题，并且所需的都是相同的运行时间和存储空间。在后面的每个示例都对这些函数的形式进行了改进，从而比这些函数更为清晰和简洁。

在定义发明家的矛盾（inventor's paradox）（How To Solve It, Princeton University Press）时，George Polya 指出“计划越宏大，成功的可能性就越大。”现在，我们就来研究在分析 Quicksort 时的矛盾。到目前为止，我们遇到的问题是，“当 Quicksort 对 n 个元素的数组进行一次排序时，需要进行多少次比较？”我们现在将对这个问题进行引

申，“对于 n 个元素的随机数组来说，Quicksort 算法平均需要进行多少次的比较？”我们通过对示例 3-6 进行扩展以引出示例 3-7。

示例 3-7：伪码：Quicksort 的平均比较次数

```
float c(int n)
    if (n <= 1) return 0
    sum = 0
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m)
    return sum/n
```

如果在输入的数组中最多只有一个元素，那么 Quicksort 将不会进行比较，如示例 3-6 中所示。对于更大的 n 值，这段代码将考虑每个划分元素 m （从第一个元素到最后一个元素，每个元素都是等可能的）并且确定在这个元素的位置上进行划分的运行开销。然后，这段代码将统计这些开销的总和（这样就递归地解决了一个大小为 $m-1$ 的问题和一个大小为 $n-m$ 的问题），然后将总和除以 n 得到平均值并返回这个结果。

如果我们能够计算这个数值，那么将使我们实验的功能就更加强大。我们现在无需对一个 n 值运行多次来估计平均值，而只需一个简单的实验便可以得到真实的平均值。不幸的是，实现这个功能是要付出代价的：这个程序的运行时间正比于 3^n （如果是自我指涉（self-referential），那么用本章中给出的技术来分析运行时间将是一个很有趣的练习）。

示例 3-7 中的代码需要一定的时间开销，因为它重复计算了中间结果。当在程序中出现这种情况时，我们通常会使用动态编程来存储中间结果，从而避免重复计算。因此，我们将定义一个表 $t[N+1]$ ，其中在 $t[n]$ 中存储 $c[n]$ ，并且按照升序来计算它的值。我们将用 N 来表示 n 的最大值，也就是进行排序的数组的大小。在示例 3-8 中给出了修改后的代码。

示例 3-8：在 Quicksort 中使用动态编程来计算

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += n-1 + t[i-1] + t[n-i]
    t[n] = sum/n
```

这个程序只对示例 3-7 进行了细微的修改，即用 $t[n]$ 来替换 $c(n)$ 。它的运行时间将正比于 N^2 ，并且所需的存储空间正比于 N 。这个程序的优点之一就是：当程序执行结束时，在数组 t 中将包含数组中从元素 0 到元素 N 的真实平均值（而不是样本均值的估计）。我们可以对这些值进行分析，从而生成在 Quicksort 算法中统计比较次数的计算公式。

我们现在来对程序做进一步的简化。第一步就是把 $n-1$ 移到循环的外面，如示例 3-9 所示。

示例 3-9：在 Quicksort 中把代码移到循环外面来计算

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += t[i-1] + t[n-i]
    t[n] = n-1 + sum/n
```

现在将利用对称性来对循环做进一步的调整。例如，当 n 为 4 时，内部循环计算总和为：

$$t[0]+t[3] + t[1]+t[2] + t[2]+t[1] + t[3]+t[0]$$

在上面这些组对中，第一个元素增加而第二个元素减少。因此，我们可以把总和改写为：

$$2 * (t[0] + t[1] + t[2] + t[3])$$

我们可以利用这种对称性来得到示例 3-10 中的 Quicksort。

示例 3-10：在 Quicksort 中利用了对称性来计算

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 0; i < n; i++)
        sum += 2 * t[i]
    t[n] = n-1 + sum/n
```

然而，在这段代码的运行时间中同样存在着浪费，因为它重复地计算了相同的总和值。因此，我们不是把前面所有的元素加在一起，而是在循环外部初始化总和并且加上下一个元素，如示例 3-11 所示。

示例 3-11：在 Quicksort 中删除了内部循环来计算

```
sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
    t[n] = n-1 + sum/n
```

这个小程序确实很有用。程序的运行时间与 N 成正比，对于每个从 1 到 N 的整数，程序将生成一张 Quicksort 的估计运行时间表。

我们可以很容易地把示例 3-11 用表格来实现，其中的值可以立即用于进一步的分析。在表 3-1 中给出了最初的结果行。

表 3-1：示例 3-11 中实现的表格输出

N	Sum	t[n]
0	0	0
1	0	0
2	0	1
3	2	2.667
4	7.333	4.833
5	17	7.4
6	31.8	10.3
7	52.4	13.486
8	79.371	16.921

这张表中的第一行数字是用代码中的三个常量来进行初始化的。下一行（输出的第三行）的数值是通过以下公式来计算的：

$$A3 = A2 + 1 \quad B3 = B2 + 2 * C2 \quad C3 = A3 - 1 + B3 / A3$$

把这些（相应的）公式记录下来就使得这张表格变得完整了。这张表格是“我曾经编写的最漂亮代码”的最好的证据，即通过少量的代码完成大量的工作。

但是，如果我们不需要所有的值，那么情况将会是什么样？如果我们更希望通过这种方式分析一部分数值（例如，在 2^0 到 2^{32} 之间所有2的指数值）呢？虽然在示例 3-11 中构建了完整的表格 t，但它只需要使用表格中的最新值。因此，我们可以用变量 t 的定长空间来替代 table t[] 的线性空间，如示例 3-12 所示。

示例 3-12：Quicksort 计算——最终版本

```
sum = 0; t = 0
for (n = 1; n <= N; n++)
    sum += 2*t
    t = n-1 + sum/n
```

然后，我们可以插入一行代码来测试 n 的合适性，并且在必要时输出这些结果。

这个程序是我们漫长学习旅途的终点。通过本章所采用的方式，我们可以证明 Alan Perlis 的经验是正确的：“简单性并不是在复杂性之前出现的，而是在复杂性之后才出现的”（“Epigrams on Programming,” Sigplan Notices, Vol. 17, Issue 9）。

观点

在表 3-2 中总结了本章中对 Quicksort 进行分析的程序。

表 3-2：对 Quicksort 比较次数的统计算法的评价

示例编号	代码行数	答案类型	答案数量	运行时间	空间
2	13	Sample	1	$n \lg n$	N
3	13	"	"	"	"
4	8	"	"	n	$\lg n$
5	8	"	"	"	"
6	6	"	"	"	"
7	6	Exact	"	3^n	N
8	6	"	N	N^2	N
9	6	"	"	"	"
10	6	"	"	"	"
11	4	"	"	N	"
12	4	Exact	N	N	1

在我们对代码的每次修改中，每个步骤都是很简单的；不过，从示例 3-6 中样本值到示例 3-7 中准确值的过渡过程可能是最微妙的。随着这种方式进行下去，代码变得更快和更有用，而代码量同样得到了缩减。在 19 世纪中期，Robert Browning 指出“少即是多 (less is more)”，而这张表格正是一个证明这种极少主义哲学 (minimalist philosophy) 的实例。

我们已经看到了三种截然不同的类型的程序。示例 3-2 和示例 3-3 是能够实际使用的 Quicksort，可以用来在对真实数组进行排序时统计比较次数。示例 3-4 到示例 3-6 都实现了 Quicksort 的一种简单模型：它们模拟算法的运行，而实际上却没有做任何排序工作。从示例 3-7 到示例 3-12 则实现了一种更为复杂的模型：它们计算了比较次数的真实平均值，但却没有跟踪任何单次的运行。

我们在下面总结了实现每个程序所使用的技术：

- 示例 3-2，示例 3-4，3-7：对问题的定义进行根本的修改。
- 示例 3-5，示例 3-6，3-12：对函数的定义进行轻微的修改。
- 示例 3-8：实现动态编程的新数据结构。

这些技术都是非常典型的。我们在简化程序时经常要发出这样的疑问，“我们真正要解决的问题是什么？”或者是，“有没有更好的函数来解决这个问题？”

当我把这个分析过程讲授给本科生时，这个程序最终被缩减成零行代码，化为一阵数学的轻烟消失了。我们可以把示例 3-7 重新解释为以下的循环关系：

$$C_0 = 0 \quad C_n = (n-1) + (1/n) \sum_{1 \leq i \leq n} C_{i-1} + C_{n-i}$$

这正是 Hoare 所采用的方法，并且后来由 D.E.Knuth 在他经典的《The Art of Computer Programming》(Addison-Wesley) 一书的第三卷：排序与查找中给出的方法中给出了描述。通过重新表达编程思想的技巧和在示例 3-10 中使用的对称性，使我们可以把递归部分简化为：

$$C_n = n - 1 + (2/n) \sum_{0 \leq i \leq n-1} C_i$$

Knuth 删除了求和符号，从而引出了示例 3-11，这可以被重新表达为一个在两个未知量之间有着两种循环关系的系统：

$$C_0 = 0 \quad S_0 = 0 \quad S_n = S_{n-1} + 2C_{n-1} \quad C_n = n - 1 + S_n / n$$

Knuth 使用了“求和因子 (summing factor)” 的数学方法来实现这种解决方案：

$$C_n = (n+1)(2H_{n+1} - 2) - 2n \sim 1.386n \lg n$$

其中 H_n 表示第 n 个调和数 (harmonic number)，即 $1 + 1/2 + 1/3 + \dots + 1/n$ 。这样，我们就从对程序不断进行修改以得到实验数据顺利地过渡到了对程序行为进行完全的数学分析。

在得到这个公式之后，我们就可以结束我们的讨论。我们已经遵循了 Einstein 的著名建议：“尽量使每件事情变得简单，并且直到不可能再简单为止。”

附加分析

Goethe 的著名格言是：“建筑是静止的音乐”。按照这种说法，我可以说“数据结构是静止的算法。”如果我们固定了 Quicksort 算法，那么就将得到了一个二分搜索树的数据结构。在 Knuth 发表的文章中给出了这个结构并且采用类似于在 Quicksort 中的循环关系来分析它的运行时间。

如果要分析把一个元素插入到二分搜索树中的平均开销，那么我们可以以这段代码作为起点，并且对这段代码进行扩展来统计比较次数，然后在我们收集的数据上进行实验。接下来，我们可以仿照前面章节中的方式来简化代码。一个更为简单的解决方案就是定义一个新的 Quicksort，在这个算法中使用理想的划分算法把有着相同关联顺序的元素划分到两边。Quicksort 和二分搜索树是同构的，如图 3-1 所示。

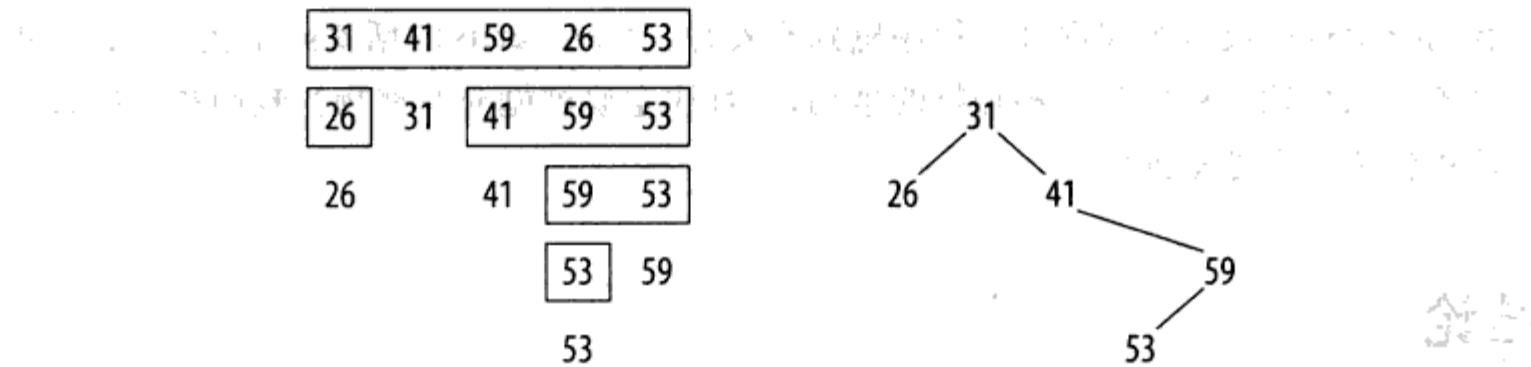


图 3-1：实现理想划分的 Quicksort 以及相应的二分搜索树

左边的方框给出了正在进行中的理想划分的Quicksort，右边的图则给出了相应的从相同输入中构建起来的二分搜索树。这两个过程不但需要进行的比较次数是相同的，而且还将生成相同的比较集合。通过在前面对于在一组不同元素上进行 Quicksort 实验的平均性能分析，我们就可以得到将不同的元素随机插入到二分搜索树中的平均比较次数。

本章的中心思想是什么

表面上看来，我“所写的”内容就是从示例 3-2 到示例 3-12 的程序。我最初是漫不经心地编写这些程序，然后将这些程序写在给本科生讲课的黑板上，并且最终写到本章中。我有条不紊地进行着这些程序的修改，并且花了大量的时间来分析这些程序，从而确信它们都是正确的。然而，除了在示例 3-11 中实现的表格外，我从来没有把任何一个示例作为计算机程序运行过。

我在贝尔实验室呆了将近二十年，我从许多教师（尤其是 Brian Kernighan，他编写了本书的第 1 章的内容）那里学到了：要“编写”一个在大众面前展示的程序，所涉及到的东西比键入这个程序要多得多。有人用代码实现了这个程序，最初运行在一些测试示例中，然后构建了完整的系统框架、驱动程序以及一个案例库来支撑这段代码。理想的情况是，人们可以手动地把编译后的代码包含到文本中，不加入任何的人为干涉。基于这种想法，我编写了示例 3-1（以及在《Programming Pearls》中的所有代码）。

为了维护面子，我希望永远都不要实现从示例 3-2 到示例 3-12 的代码，从而使 I 保持诚实的名声。然而，在计算机编程中的近四十年的实践使我对这个任务的困难性有着深深的敬畏（好吧，更准确地说，是对于错误的害怕）。我妥协了，把示例 3-11 用表格方式实现出来，并且无意中得到了一个完备的解答。当这两个东西完美地匹配在一起时，你可以想象一下我当时的喜悦吧！因此，我向世界提供了这些漂亮的并且未曾实现的程序，虽然在这些程序中可能会有一些还未发现的错误，但我对这些程序的正确性还是有一定信心的。我希望一些细微的错误不会掩盖我在这些程序中所展示的那些漂亮思想。

当我为给出这些没有被实现过的程序感到不安时，Alan Perlis的话安慰了我，他说“我们可不可以把软件视作为一种独特的事物，一种注定要被抛弃的事物？我们是否应该将软件视作为一个肥皂泡？”

结论

漂亮有着许多种含义。本章通过简单、优雅以及精练刻画了漂亮的含义。下面这些名言表达的是同样的意思：

- 通过删除代码来实现功能的提升。
- 只有在不仅没有任何功能可以添加，而且也没有任何功能可以删除的情况下，设计师才能够认为自己的工作已臻完美。
- 有时候，在软件中根本就不存在最漂亮的代码，最漂亮的函数，或者最漂亮的程序。
- 良好的写作风格即为简练。省略不必要的字词。(Strunk and White)
- 在计算机系统中，那些最廉价、速度最快以及最为可靠的组件是不存在的。(Bell)
- 努力做到事半功倍。
- 如果我有更多的时间，那么我给你写的信就会更短。(Pascal)
- 发明家的矛盾：计划越宏大，成功的可能性就越大。(Polya)
- 简单性并不是在复杂性之前出现的，而是在复杂性之后才出现。(Perlis)
- 少即是多。(Browning)
- 尽量使每件事情变得简单，并且直到不可能再简单为止。(Einstein)
- 软件有时候应该被视作为一个肥皂泡。(Perlis)
- 在简单中寻找漂亮。

本章的内容到此结束。读者可以复习所学到的内容并进行模拟实验。

对于那些想要获得更具体信息的人们，我在下面给出了一些观点，这些观点分为三类：

程序分析

深入理解程序行为的方式之一就是修改这个程序，然后在具有代表性的数据上运行这个程序，就像示例 3-2 那样。不过，我们通常会更关心程序的某个方面而不是程序的整体。例如，我们只是考虑 Quicksort 所使用的平均比较次数，而忽略了其他的方面。Sedgewick (“The analysis of Quicksort programs,” *Acta Informatica*,

Vol. 7)研究了Quicksort的其他特性,例如算法所需的存储空间以及各种Quicksort运行时间的其他方面。我们可以关注这些关键问题,而暂时忽略了程序其他不太重要的方面。在我的一篇文章“*A Case Study in Applied Algorithm Design*”(IEEE Computer, Vol. 17, No. 2)中指出了我曾经遇到过的一个问题:对在单元空间中找出货郎行走路线的strip启发式算法的性能进行评价。我估计完成这个任务所需要的程序大概在100行代码左右。在经历了一系列类似于本章前面看到的分析步骤之后,我只使用了十几行代码的模拟算法就实现了更为精确的效果(在我写完了这个模拟算法后,我发现Beardwood等人(“*The Shortest Path Through Many Points*,” Proc. Cambridge Philosophical Soc., Vol. 55)已经更完整地表述了我的模拟算法,因此已经在二十几年前就已经从数学上解决了这个问题)。

小段代码

我相信计算机编程是一项实践性的技术,并且我也同意这个观点:“任何技术都必须通过模仿和实践来掌握。”因此,想要编写漂亮代码的程序员应该阅读一些漂亮的程序以及在编写程序时模仿所学到的技术。我发现在实践时有个非常有用的东西就是小段代码,也就是一二十行的代码。编写《Programming Pearls》这本书是一件艰苦的工作,但同时也有着极大的乐趣。我实现了每一小段代码,并且亲自把每段代码都分解为基本的知识。我希望其他人在阅读这些代码时与我在编写这些代码时有着同样的享受过程。

软件系统

为了有针对性,我极其详尽地描述了一个小型任务。我相信其中的这些准则不仅适用于小型程序中,它们同样也适用于大型的程序以及计算机系统。Parnas(“*Designing software for ease of extension and contraction*,” IEEE T. Software Engineering, Vol. 5, No. 2)给出了把一个系统拆分为基本构件的技术。要想获得快速应用性,不要忘了Tom Duff的名言:“在尽可能的情况下,重用现有的代码。”

致谢

非常感谢Dan Bentley, Brian Kernighan, Andy Oram 和 David Weiss 耘有见识的评语。

第4章

查找

Tim Bray

计算机可以用来进行数值计算，但这并不是人们用它的主要原因。在大部分情况下，计算机都被用来存储以及获取信息。“获取”就意味着“查找”，随着网络的出现，搜索也逐步成为了人们使用计算机的一个主要功能。

随着数据量的不断增加——不论是从绝对数据量来说，还是相对于单台计算机以及使用计算机的用户的相对数量而言都是如此——搜索在程序员的日常工作中所占据的比例也逐步提高，成为很大的一部分。现在只有很少（确切地说是：非常少）一部分应用程序不需要在大量信息中对某些数据进行定位。

“搜索”是计算机科学中的几个最大的话题之一，所以在此我不准备对此进行全面而细致的阐述；实际上，在本章中我仅仅就一个简单的查找问题进行深入的描述。在接下来的章节中，我将会着重就“如何选择搜索技术”这一过程中出现的一些微妙的权衡进行阐述。

耗时

我们很难抛开“耗时”这个因素来单纯的谈论“搜索”这个话题。在解决一个有关搜索

问题的过程中，我们可以从两个方面来权衡时间的消耗。第一种消耗发生在“搜索”本身运行后，此时大部分用户的体验过程都是看着屏幕上出现的“载入中……”的字样等待所查找的信息显示在屏幕上；另外一类耗时则出现在程序员构建该搜索程序的过程中，此类耗时还包括项目管理的时间以及用户等待该程序变得可用的时间。

问题：数据

在本章中，我们将提供一个示例让大家对发生在日常生活中的搜索如何工作有一个直观的认识。我本人有一个目录，其中包括了从 2003 年初到 2006 年末我本人博客 (<http://www.tbray.org/ongoing>) 的访问日志；当我撰写本章时，这些日志中包括了 140,070,104 条事务处理，在未被压缩的情况下在我的硬盘空间中占据了 28,489,788,532 字节。通过对这些日志的分析，以及合理的搜索，我们可以从中得到很多有关于阅读该博客的读者的信息（如：从哪个 IP 访问该博客的流量最大；哪些文章最受欢迎等）。

我们先从一个简单的问题开始：哪篇文章被阅读次数最多？虽说该问题粗看上去和搜索的关联不大，但它的的确确就是一个搜索问题。首先，我们必须搜索所有的日志文件，从中得到那些记录了“文章获取”的日志；然后，我们就得对这些日志记录进行分析以得到那些被访问的文章名；最后，我们得对每篇文章的访问次数进行统计以找出被阅读次数最多的那篇文章。

下面就是我们从日志文件中摘出的一条示例日志记录，为了排版的需要我在此把它分拆成了多行，实际上它在日志文件中仅仅占据了一行记录：

```
c80-216-32-218.cm-upc.chello.se - - [08/Oct/2006:06:37:48 -0700] "GET /  
ongoing/When/200x/2006/10/08/Grief-Lessons HTTP/1.1" 200 5945 "http://www.tbray.org/ongoing/"  
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)"
```

从左向右阅读该记录，我们可以得知：

在 2006 年 10 月 8 号早晨（西七区时间），某人从位于瑞典的名为 chello 的机构中，匿名登录到我的博客，通过 HTTP 1.1 协议向服务器请求获取名为 */ongoing/When/200x/2006/10/08/Grief-Lessons* 的文章，该请求被成功处理并且服务器向对方发送了 5,945 字节的回复，该用户是从我的博客首页得到的文章链接，并且他所使用的浏览器是 IE6，使用的操作系统为 Windows XP。

这就是我所期望得到的记录格式：它确确实实包含了对文章的获取操作。除此之外，在日志文件中还存在着许许多多的日志记录，它们记录了对我博客中的样式表、脚本、图像等的获取，以及来自于恶意用户对服务器的攻击操作。可以发现，我所期望的记录格

式如下：以*/ongoing/When/*开头，后面紧接着时代、年、月、日的信息，最后是文章本身的名字。

我们的第一步就是，找到那些包含有类似于如下信息的记录：

/ongoing/When/200x/2006/10/08/

不论我们使用何种编程语言，我们都需要花费大量时间来编写进行字符串模式匹配的代码；除非我们使用了正则表达式（regular expression，有时也简称 regexp）。

正则表达式

正则表达式是一种非常强大的工具，可以用来处理文本。它们的一个显著特点是：正则表达式是被设计成专门用来进行文本模式匹配的语言。一旦我们学会了如何正确有效地使用它，我们就可以避开许多的弯路，节省出大量的时间。我很少碰到一个成熟的程序员是不会正则表达式的情况。在第1章中，Brian Kernighan已经为我们描述了正则表达式的美妙之处。

由于我博客中的文件名格式严格遵循着按时间匹配的模式，我们可以很直观地找出那些我们所感兴趣的日志记录的正则表达式如下（可能其他网站的日志系统会需要一个更复杂的正则表达式）：

```
"GET /ongoing/When/\d\d\dx/\d\d\d\d/\d\d/\d\d/[^ .]+ "
```

通过对上面那行代码的简单扫视，我们就可以发现正则表达式的一个显著问题：它们是不可读的。可能有人会质疑，为什么在一本名为《代码之美》的书中会出现此类不可读的代码。对于该问题，我们先将其放置一边，转而先来研究一下这个特殊的表达式。此处我们所需要了解的东西有：

\d

意即：匹配所有的数字，从0到9

[^ .]

意即：匹配所有的不是空格和句点的字符（注1）

+

意即：匹配一个或多个在+前面所出现的实例

在上述正则表达式中，[^ .] + 意味着在最后一个斜杠后必须要紧接着一系列不是空格

注1：对于那些使用正则表达式的人来说，句点在正则表达式中有着特殊的含义，它意味着“任意字符”；但当一个句点被放置在方括号中时，它将失去该特殊含义而只表示一个简单的句点。

和句点的字符。在`+`后面有个空格，但正则表达式引擎读到了这个空格时，就意味着一次成功匹配的结束。

上述正则表达式不能匹配文件名中包含有句点的情况。也就是说，它可以匹配我在前面所举例给出的 `Grief-Lessons`，但不能匹配 `IMG0038.jpg`。

将正则表达式引入到工作中

上面所给出的正则表达式，单就它本身而言，可以被用在命令行中对文件进行搜索。但是，现在大部分的现代编程语言都可以让我们在程序代码中直接使用它们。我们接下来要做的就是这个，编写一个程序，它将打印出所有和该表达式所匹配的记录；也就是说，编写一个可以找到所有对该博客文章进行获取记录的程序。

该示例程序（以及本章中的大部分示例程序）是用 Ruby 所编写的，原因是：我个人认为，虽然 Ruby 还不是那么的完美，但它已经是我所使用过的可读性最高的编程语言。

如果你还不了解 Ruby，我建议你去学习一下，学习 Ruby 将使得你能够变成一个更好的程序员。在第 29 章中，Ruby 的创造者，Yukihiro Matsumoto（通常也被称为“Matz”）描述了一些他在设计 Ruby 过程中的选择，也正是这些设计吸引了我和许多其他程序员来使用 Ruby。

示例 4-1 中展示了我们的第一个 Ruby 程序，并且它还包括了位于代码左侧的行号，用来辅助代码阅读理解。（本章中的所有示例代码都可以从 O'Reilly 搭建的关于本书的网站上获得）。

示例 4-1：打印所有获取文章的记录

```
1 ARGF.each_line do |line|
2   if line =~ %r{GET /ongoing/When/\d\d\dx/\d\d\d\d/\d\d/\d\d/\d\d/[^ .]+ }
3     puts line
4   end
5 end
```

运行上述程序将在屏幕上打印出一堆和我们最初给定的例子相似的日志记录。接下来我们将对该程序进行逐行分析：

第一行

我们期望从输入中读到所有的行，但我们并不关心程序的输入是来自于命令行中给出的文件名还是通过管道（pipe）从其他程序所得到的标准输入。Ruby 的设计者深信：程序员不会为常见的场景而编写丑陋的代码，本例恰好是一个常见的场景。ARGF 是一个用来代指所有输入源的特殊变量。如果命令行中包含参数，那么

ARGF 将假设这些参数都是文件的名字，从而一一地打开它们；如果命令行中没有参数，那么 ARGF 将使用标准输入作为其值。

`each_line` 是一个可以用在所有类似于文件的对象（如：ARGF）上的方法。它会从输入中读取数据，然后向后续的代码“块”中一次一行地传输这些数据。

接下来的 `do` 表示，获取输入的代码块从此开始，到相应的 `end` 结束。`|line|` 表明了 `each_line` 方法每次获取的那行记录在进入代码块被处理之前被放置到了变量 `line` 中。

上述此类循环可能会对那些刚刚使用 Ruby 的用户来说很突兀，但在经过少许训练后，使用它将变得很简单，也很简明和强大。

第二行

本行是一个很直观的 `if` 表达式。惟一的魔法出现在 `=~` 上，它意味着“匹配”，并且将它后面所跟着的表达式按照正则表达式来解析。我们可以通过在正则表达式的首末添加斜杠来告知 Ruby 这是一个正则表达式——如：`/this-is-a-regex/`。但由于我们所选用的正则表达式中充斥着大量的斜杠，如果选择使用这种“首末斜杠”的语法，我们就将不得不把正则表达式中的所有斜杠都进行“转义化”，也就是把每个 / 写成 \/, 这将使得代码变得很丑陋。所以，在本例中，我们使用了 `%r` 语法，以产生了更美观的代码。

第三行

本行位于 `if` 语句块中。如果当前的 `line` 可以和那条正则表达式匹配，示例程序就将执行 `puts line` 这条语句向外输出 `line` 的内容以及一个换行符。

第四、五行

这两行没有什么好说的。第一个 `end` 将终止 `if` 语句块，第二个 `end` 将终止 `do` 语句块。从代码格式来看，它们有点像是那些丑陋的“挂面”式代码；而 Python 的设计者给出了一种忽略它们的方法，这也使得某些 Python 代码看上去比相应的 Ruby 代码更美观。

至此，我们已经展示了如何使用正则表达式来在日志文件中找到我们所感兴趣的记录。但我们真正感兴趣的事情是找出每篇文章被阅读的次数。为此我们首先得找出文章的名字。示例 4-2 就是对前一示例程序的一个小的改进。

示例 4-2：打印文章名

```
1 ARGF.each_line do |line|
2   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^ .]+)}
3     puts $1
4   end
5 end
```

示例 4-2 和 4-1 之间的差别很细微。在第二行中，我新增了一对括号（以粗体显示）用以标注出正则表达式中我们所感兴趣的相关文章名的部分。在第三行中，我们这次选择的不是输出 `line` 中所有的内容，而是输出 `$1` 的内容。在 Ruby（以及一些其他支持正则表达式的编程语言）中，`$1` 意味着“正则表达式中第一处被括号所标注的位置”。我们也可以在正则表达式中进行多处标注，从而得到 `$2`, `$3`, 等等……

在对一些日志文件进行处理后，上述示例代码的输出如下：

```
2003/10/10/FooCampMacs  
2006/11/13/Rough-Mix  
2003/05/22/StudentLookup  
2003/11/13/FlyToYokohama  
2003/07/31/PerlAngst  
2003/05/21/RDFNet  
2003/02/23/Democracy  
2005/12/30/Spolsky-Recursion  
2004/05/08/Torture  
2004/04/27/RSSticker
```

接下来，在对文章的受欢迎程度进行分析之前，我想就一些重要方面来论证我的观点：上述代码很漂亮。试想一下，如果不能使用正则表达式，转而让你来手工编写一个具有同样功能的程序，那将会是什么情况。首先，这个程序将会包含更多的代码，并且它更容易出错。再者来说，如果日志文件的格式改变了，那么修改这个模式匹配器将会变得很麻烦，也很容易带来未知的错误。

在表层现象之下，正则表达式的工作方式同时也是计算机科学中最精彩的一部分。我们可以证明，它可以很方便地转化为一个有限自动机。这些自动机在数学上来说很优雅，而且那些用来搜索的文本进行匹配的算法也是惊人的高效。运行自动机的一个很大的好处在于，我们只需要对试图进行匹配的文本进行一次扫描就可以了。事实上，一个精心构建的正则表达式引擎在进行模式匹配和选取时可以做到比任何的客户代码都要高效，即便那些代码是用汇编语言编写而成并且还经过了手工优化。这多漂亮啊。

就我个人看来，Ruby 代码也很具吸引力。在 Ruby 程序中，几乎每个字符都有作用。注意，在 Ruby 程序中，不会有位于语句结束处的分号，也不会有包含条件块的括号；我们可以用 `puts line` 来代替 `puts(line)`。并且，变量也不需要被声明——我们只需在用的时候用它就可以了。这种对语法的精简设计带来的就是更短且更容易编写的程序，同时（也是更重要的）还有使得程序更易读且更易被人所理解。

从耗时方面来说，使用正则表达式是一个双赢的选择。它不但使得程序员使用更少的时间来构建程序（这也使得将程序发布给用户的时间间隔变得更短），同时还能更有效地利用计算机的计算资源，使得用户的等待（搜索完成）时间变得更短。

它里面的键值储存是无序的，因而造成 keys 方法返回的数组中那些键值也是随机排序的。为此，我不得不对这些键值进行排序并把它们放置到一个新的数组中去。

在 Ruby 中，sort 通常都伴随着一个代码块一起出现，在本例中该代码块以花括号标注出来。（在 Ruby 中，我们可以用 do 和 end 或者 { 和 } 来标注一个代码块。）sort 会在将要被排序的数组上按照它的内在机理进行来回遍历，将数组中的一对元素传递给和其关联的代码块，然后按照第一个元素是小于、等于、还是大于第二个元素的情况分别返回一个负值、0 值或者正值。

在本例中，我们期望得到一个按照哈希表中值（文章被访问次数）而不是键值（文章名）的大小排序后的数据集，为此我们必须按照对应的值来对键值排序。这点从代码中我们也可以看出来。由于这种操作是人们经常要碰到的，对于 Ruby 没有在 Hash 中提供一个 sort_by_value 的方法我感到很惊奇。

由于上面我们所使用的是一个递减的排序，因此不管我们最终可以找到多少篇文章，我们都可以确保在 keys_by_count 数组中的前十项就是我博客中最受欢迎的前十篇文章。

现在我们已经得到了一个按照它们被访问次数进行递减排序的键值（文章名）数组，那么打印出该数组的前十个元素就可以解决我们前面所提出的问题。第十一行虽然很简单，但在此处我还是要说一下 each 这个方法。在 Ruby 中，我们很难看到 for 语句，这是因为在那些我们期望对其所包容的元素进行循环处理的对象中，Ruby 都为我们提供了一个 each 方法。

由于 #{} 语法的存在，第十二行对于那些非 Ruby 用户来说有点复杂，但它还是一条很直观的语句。

至此，我们就可以顺利地宣布我们成功地解决了那个最初的问题。所有的代码只有十三行，并且它们也很容易被人所阅读理解，对于那些有经验的 Ruby 用户来说，他甚至还可以将最后三行代码写成一行。

现在让我们来运行上面的代码，看看它到底会输出些什么。在一开始我没有选择对整个 28GB 的日志文件进行处理，而只选取了其中某一周的数据来做处理：其中包含了一百二十万条日志记录，占据了我 245MB 的硬盘空间。

```
~/dev/bc/ 548> zcat ~/ongoing/logs/2006-12-17.log.gz | \
    time ruby code/report-counts.rb
4765: 2006/12/11/Mac-Crash
3138: 2006/01/31/Data-Protection
1865: 2006/12/10/EMail
1650: 2006/03/30/Teacup
```

```
1645: 2006/12/11/Java  
1100: 2006/07/28/Open-Data  
900: 2006/11/27/Choose-Relax  
705: 2003/09/18/NXML  
692: 2006/07/03/July-1-Fireworks  
673: 2006/12/13/Blog-PR  
13.54 real      7.49 user      0.73 sys
```

上述代码运行在我的那台 1.67GHz 的 Apple PowerBook 上。结果本身没有什么大问题，但程序的运行速度看起来有点慢。那么我们是否应该关心该程序的效率呢？

优化

对于上述程序是否真的慢的毫无理由，我感到有点疑惑，为此我用 Perl 构建了一个类似的程序。众所周知，Perl 的代码虽然不如 Ruby 那么美观，但其却是用以处理文本的编程语言中效率最高的。果然，运行使用 Perl 所编写的那个程序只需要一半的时间。那么，我们是否应该对我们上面的程序进行优化呢？

在我们开始优化之前，我们应该再次考虑“耗时”。是的，我们可以设法让这个程序运行的更快，减少程序的运行时间以及用户的等待（程序运行结束）时间。但是，这却需要消耗掉程序员的一部分时间，这就增加了用户等待程序员把程序写好的时间。在大部分情况下，我的个人直觉是，处理一周的数据耗时 13.54 秒这个成绩已经很不错了。但考虑到用户可能抱怨，我们最好还是想法使它运行的更快才是。

通过对示例 4-4 的简单扫视，我们可以把它分为两个明显的部分。在第一段中，它读入所有的日志记录并将那些获取操作放置到一个表中；然后它在第二段中对前面生成的那个表进行排序并输出“最受欢迎文章”的前十名。

从中我们可以很容易地发现一处可以优化的地方：既然我们所期望的只是那个“前十名”，为什么我们还要对所有的获取操作进行排序？我们完全可以编写一个简单的代码对数组进行遍历，然后输出其中值最大的前十项。

这样做是否真的能起到优化的作用呢？通过对程序的指令深入剖析，我们可以得出那两段代码各自的运行时间。通过多次运行取平均值的做法，我们所得到的结论是：代码中的第一段需要耗时 7.36 秒，第二段则仅耗时 0.07 秒。这也意味着：上面关于排序的想法对于优化来说没有任何帮助。

那么我们是否该试图对第一段代码进行优化呢？恐怕还是不行；因为第一段代码所做的只是正则表达式的匹配，以及使用 Hash 来对数据进行读取操作，而所有这些我们都是通过使用 Ruby 中已经被极大优化过的部分实现的。

现在我们得到了如下结论：试图替换 sort 的设想会是对程序员时间的浪费，让用户等待更长的时间来获得一个可用的程序，并且还不能显著地节省程序的运行时间和用户的等待（程序运行介绍）时间。而且，经验表明，我们也不可能让程序运行的比使用 Perl 编写的类似程序更快；因此我们所能获得的速度提升极为有限。

前面我们刚刚完成了一个程序的编写，该程序不但有用，而且还与搜索有关。但我们并没有真正地编写一行和搜索算法相关的代码。现在我们就来做这个事情。

关于计数的一些历史

公正地来说，通过使用正则表达式来匹配对输入的文本进行扫描匹配并将结果放置到一个内容可寻址存储的做法最初出现在 awk 这种语言中，awk 这个名字来源于发明它的 Aho、Weinberger 以及 Kernighan 这三个作者的姓氏缩写。

毫无疑问，这种做法完全是基于由 Ritchie 和 Thompson 所提倡的 Unix 基本哲学的——数据大都应该按行被存储在文本文件中——并在某些程度上对该哲学进行了扩展。

Larry Wall，Perl 的作者，从 awk 吸取了这种思想，将它实现成了一种高性能的，具有工业强度的，通用的工具。Perl 也就因此成为了 Unix 业界的一种粘合剂，并且随即在当时出现的第一代英特网中起到了重要的作用。

问题：时间，人物，以及对象

通过在那些日志数据上运行一些脚本，我们可以得知，总共有 12 600 064 次的文章获取操作，这些操作的请求来自于 2 345 571 个不同的主机。假设我们对“谁在何时阅读了哪篇文章”这个问题感兴趣？我想对于读者、警察、以及市场调查人员来说，他们也会对此问题感兴趣吧。

我们把问题重新描述为：给定一个主机名，找出它所访问过的所有文章，以及访问这些文章的时间。我们期望结果将会是一个链表；如果链表为空的话，那就意味着没有文章被该主机所访问。

在本章的前面章节中，我们已经了解到了，编程语言中内建的哈希机制或者相似的数据结构可以让程序员很方便且快速地完成对“键值/值”数据对进行存储和查找的操作。因此在此你可能会问，我们为什么不继续使用它呢？

这是一个极好的问题，我们应该试一试这个想法。当然，我们有理由担心这个想法可能运行得不理想，因此为了支持我们的思路，还得准备一个备用方案。回想一下当学习哈希表时，我们知道，为了得到一个很好的性能，我们需要将哈希表的装载因子（load factor）变小；换句话来说就是，哈希表的大部分容量都没有被使用。对于一个放置有235万数据后我们仍然期望其有着一个小装载因子的哈希表来说，其占据的内存将是一个天文数字。

为了简化问题，我编写了一个程序，它会在读取所有的日志文件后把那些包含有文章访问的日志记录放到一个简单的文件中。在这个文件中，每条记录中都包含有主机名，事务发生的时间以及文章的名字。下面就是几个例子：

```
crawl-66-249-72-77.googlebot.com 1166406026 2003/04/08/Riffs  
egspd42470.ask.com 1166406027 2006/05/03/MARS-T-Shirt  
84.7.249.205 1166406040 2003/03/27/Scanner
```

（上面记录中的第二列，也就是由那10个数字组成的值，用的是标准的Unix/Linux纪元法，也就是从1970年开始算起所流逝过的秒数。）

接着我就编写了如下简单程序来将该文件读入到一个极大的哈希表中。如示例4-5所示。

示例 4-5：将数据载入大哈希表中

```
1 class BigHash  
2  
3   def initialize(file)  
4     @hash = {}  
5     lines = 0  
6     File.open(file).each_line do |line|  
7       s = line.split  
8       article = s[2].intern  
9       if @hash[s[0]]  
10         @hash[s[0]] << [ s[1], article ]  
11       else  
12         @hash[s[0]] = [ s[1], article ]  
13       end  
14       lines += 1  
15       STDERR.puts "Line: #{lines}" if (lines % 100000) == 0  
16     end  
17   end  
18  
19   def find(key)  
20     @hash[key]  
21   end  
22  
23 end
```

上述程序的意图从其代码中就可以很容易地看出来，在此我仍想对第15行多说几句。当运行一个需要耗费大量时间（也许持续好几个小时）的程序时，如果程序在运行过程中

没有产生任何的输出，我们通常会觉得惴惴不安：是不是哪个地方出了问题？它是不是运行的太慢并且永远不会结束？这也就是我们为什么要在第15行处对于每处理完的十万条记录输出一个进度报告的原因：只是为了让我们安心。

运行该程序比较有趣。它花费了大概55分钟的CPU时间来将数据装载进哈希表中，此时程序占用的内存大概在1.56GB。经过简单的计算，我们可以得知，对于每个主机来说，它们在内存中占据的空间大致是680字节；或者，换成是每次操作来说，它们需要消耗126字节的内存。这个数据虽然有点吓人，但对一个哈希表来说，还是在合理范围内的。

哈希表的读取性能表现得很优异。我运行了2000条查询，其中有一半是从日志中随机选择主机名，这样的查询肯定是会成功的；另外一半的查询将前面那一半查询所使用的主机名反转来使用，这样的查询是不会成功的。经过多次的运行，完成所有2000条查询所需要的平均时间是0.02秒。也就是说，Ruby的哈希实现可以在一个容纳有1200万条记录的表中进行每秒好几千次的查询。

现在的问题就在那个55分钟的装载时间了，不过我们还可以找到一些技巧来对付它。例如，我们可以对哈希表进行一次装载，然后将它序列化（serialize）到一个文件中，在需要用时再将该文件重新读取进内存。不过我并没有特意去优化该程序。

上面那个程序的编写很简单也很快捷，并且当初始化过程结束后，它的运行速度也很快；对于“程序等待时间”（waiting-for-the-program time，即等待程序运行结束的时间）以及“程序员等待时间”（waiting-for-the-programmer time，即等待程序员发布该程序的时间）来说，它都可以满足人们的要求。但是，我对它并不满意。我认为，总有一种方法，使得我们可以在获得同样的运行效率的前提下，对于内存消耗、初始化时间这两个方面都可以给该程序以改善。为此，我们就得动手来编写我们自己的搜索代码。

二分查找

所有获得计算机科学学位的人都学过多种查找算法，如：树、堆、哈希、列表、等等，在所有这些算法中，我个人的最爱就是二分查找。现在让我们把该算法用到我们的“时间－人物－对象”问题上去，来看看它是如何把程序变得美妙的。

我第一次尝试将二分法加到程序中的结果很令人沮丧，虽说它成功地减少了10分钟的装载时间，但相比哈希表它又多消耗了100MB的内存。很明显，Ruby的数组实现中存在着一些令人惊讶的问题。同时，改动后的查找也比先前的要慢好几倍（不过还是可以保持在每秒上千次的范围内），不过这一点也不令人惊讶，因为改动后的程序是以Ruby代

码的方式运行的，并不像先前的哈希表那样，可以访问到内置的、硬编码的哈希实现上。

上面的问题在于，在Ruby中，所有的一切都是对象，就连数组也是一个抽象的概念，其间包含了很多不为人知的细节。因此，我们换用Java来实现该程序（在Java中，整数就是整数，Java中的数组相对添加的东西也要比Ruby中少）。（注2）

从概念上来说，没有比二分法更简单的了。我们把查找的范围一分为二，然后确定自己是应该在上半区还是下半区进行下一次查找；重复上述过程直到查找结束（找到或者失败）。必须提一下的是，在所有该算法的实现中，很多是错误的，甚至在一些被广为流传的实现中，其间还存在着一些问题。我在“On the Goodness of Binary Search”中提到的那个实现，以及接下来我们给出的Java示例代码中的实现，来自于Gaston Gonnet（他曾经是用于符号数学计算的Maple语言的主要程序员，目前是一位在瑞士苏黎世理工学院讲述计算机科学的教授）。

示例 4-6：二分查找

```
1 package binary;
2
3 public class Finder {
4     public static int find(String[] keys, String target) {
5         int high = keys.length;
6         int low = -1;
7         while (high - low > 1) {
8             int probe = (low + high) >>> 1;
9             if (keys[probe].compareTo(target) > 0)
10                 high = probe;
11             else
12                 low = probe;
13         }
14         if (low == -1 || keys[low].compareTo(target) != 0)
15             return -1;
16         else
17             return low;
18     }
19 }
```

下面是关于该程序的几个关键点：

- 在第5、6行中，请注意，high和low边界被设置为数组两端偏移一个元素处，在初始化时它们都不是一个有效的索引值，这也就消除了所有关于边界判断的问题。

注2：此处关于二分查找的讨论大部分来自于我在2003年发表的博客文章，“On the Goodness of Binary Search”，它的地址是：<http://www.tbray.org/ongoing/When/200x/2003/03/22/Binary>。

- 从第 7 行开始的循环，其终止条件是 `high` 和 `low` 这两个索引最终相邻；在程序中，我们并没有去判断目标是否被找到。仔细考虑一下后，我想你就会同意这种做法；我们也将在后面对这个问题进行讨论。

前面的循环有着两个不变量 (invariant)：`low` 要么是 -1，要么就是指向一个不大于目标值的值；`high` 要么指向数组尾端后一个位置，要么就指向第一个大于目标值的值。

- 上面程序中的第 8 行尤其有意思。在以前的版本中，它是这样写的：

```
probe = (high + low) / 2;
```

但是在 2006 年 6 月，Java 大师 Josh Bloch 指出：在某些情况下，上述代码会导致整数溢出（参见 <http://googleresearch.blogspot.com/2006/06/extr-extra-read-all-about-it-nearly.html>）。事实表明，在过去的几十年间，在计算机科学领域中，我们不断地在一些核心算法中发现新问题。（Alberto Savoia 在第 7 章中也讨论过这个问题。）

在这一点上，Ruby 用户会指出，诸如 Ruby 和 Python 这样的现代动态语言，它们会自动为我们处理有关整数溢出的事情，因此这个问题对于它们来说也就不是问题了。

- 由于我们前面所提到的不变量，当循环结束时，我们所需要的只是去检测 `low`（代码中的第 14 至 17 行）。如果它不是 -1，那么它要么指向的就是我们期望找到的那个目标值，要么就表示在数组中不存在这个目标值。

上面的这个 Java 程序只需要 6 分半钟就可以完成数组的装载（从日志文件中读取数据至数组中），同样它也可以正确运行，此时它所耗费的内存还不到 1GB。虽说相比 Ruby 来说，对 Java 程序运行时的 CPU 开销进行度量要难一些，不过通过对同样的 2000 条查询的运行结果来看，我们并没有觉察到明显的延时。

二分查找法的权衡

对于二分查找来说，它有着一些极大的好处。首先，它的查找时间复杂度是 $O(\log_2 N)$ 。人们通常都没有意识到这一点的好处。在一台 32 位的计算机上，我们所能碰到的最大的 $\log_2 N$ 就是 32（同样，对于 64 位机来说就是 64），对于现实世界的大多数场景来说，具有如此性能上限的算法已经是“足够好”了。

其次，二分查找的代码很短，也很简单。短且简单的代码就是漂亮的代码，对此我们可以找出很多的理由。可能其中最重要的一条理由就是：它们很容易被人们所理解，而相比编写代码来说，理解代码要更难。对于短小的代码来说，可供问题容身的场所很少。

同时，紧凑的代码在指令集、指令缓存、以及 JIT 编译器上都可以获得更好的优化，从而运行得更快。

第三，一旦对数组排序完毕，我们就将不再需要额外的索引结构；在这方面，二分查找可以节省大量的空间。

二分查找的最大问题在于，数据必须放置在内存中且必须是有序的。对于某些数据集来说，做到这一点是不可能的，不过幸好这类数据集比我们想像中的要少。如果觉得需要放置到内存中的数据很多的话，我们可以去调查一下市场上内存的价格以确保可以为计算机配置足够的内存。所有需要将数据倒至磁盘的查找策略都很复杂，同时在大部分情况下性能都不令人满意。

假设我们需要更新内存中数据集；你也许会认为这会极大的影响二分查找：因为我们必须得对内存中一个巨大的、连续的数组进行更新。但事实比你想像的要简单，实际上，我们程序所使用的内存片段散布在物理内存中的各个角落，它们被操作系统的页机制所管理并以一个连续的内存块的形式展现在我们面前；我们也可以对自己的数据采用同样的技巧来管理它们。

有人可能会指出，哈希表的查找时间复杂度只是 $O(1)$ ，这要好于二分查找的 $O(\log_2 N)$ 。在实际应用中，这点差距并不明显；你也可以亲自构建一个实验环境来验证这点。同时，我们还可以考虑一下哈希表中那些用来解决哈希冲突（collision）的代码，它们的实现可不简单。

我并不想在此武断地下结论，不过近些年，有关搜索的问题，我采用的解决方案是：

1. 首先试着使用语言内建的哈希表；
2. 如果性能不能令人满意，那么就尝试使用二分查找；
3. 只有在前面两步都不能取得令人满意的结果的前提下，我才会考虑使用更复杂的方法。

关于循环

在看了上面的二分查找算法的实现后，有人可能会问，为什么我们要将算法中的循环运行到结束处，而不是在检测到了目标值的时候就退出循环呢。实际上，上述代码的行为才是正确的行为；虽说对于该行为的正确性数学证明已经超出了本章的范围，但只要经过一些简单的思考，我们就应该可以从直觉中获得这个结果——从以往共同工作过的许多伟大的程序员中，我不止一次地发现过他们的这种直觉。

我们先来考虑循环的执行步骤。假设我们有一个有着 n 个元素的数组（此处 n 是一个很大的数值），那么从该数组中第一次找到目标的概率就为 $1/n$ （一个很小的数值），下一次（经过一次二分）的概率则是 $1/(n/2)$ —— 仍然不是很大 —— 以此类推下去。事实上，只有当元素的个数减少到了 10 到 20 的时候，一次找到目标的概率才变得有意义，而对于 10 到 20 个元素进行查找需要的只是大概 4 次循环。当查找失败时（在大多数的应用中很普遍），那些额外的测试就将变成纯粹的额外开销。

我们也可以来计算一下，在什么时候找到目标值的概率能接近 50%，但请你们扪心自问：在一个复杂度为 $O(\log_2 N)$ 的算法中，对于它的每一步都增添一个额外的复杂计算，而目的仅仅是为了减少最后的几次计算，这样做有意义吗？

这里给我们的经验就是，恰当的二分查找的实现是一个“两步走”的形式。首先，编写一个高效的循环来妥当定位我们的 `low` 和 `high` 边界；然后用一个简单的检测来判断查找成功与否。

大规模尺度的搜索

对大部分人来说，“搜索”就意味着网络搜索，就是 Yahoo!、Google、以及他们的竞争对手所提供的搜索服务。虽然无所不在的网络搜索是一个新鲜事物，但对于他们所基于的全文搜索来说则不是。在这个领域中，大部分极具影响力的论文都是来自 Cornell 的 Gerald Salton 在 20 世纪 60 年代早期的发布。从那时起，对于大量文本的索引和查找的基础技术并没有得到很大的改进。真正有所改变的只是如何来对搜索到的结果进行排名（注 3）。

使用 Postings 来搜索

对于全文搜索的标准做法基于 *posting* 这个概念，*posting* 是一个小的、固定大小的记录。在创建索引的过程中，我们读取所有的文档，然后对每个词，我们都为它创建一个 *posting*，用来描述这个词 x 在文档 y 中的位置 z 出现。然后，我们对所有的词进行排序，得到相对每个唯一词的 *posting* 列表，在链表中的每个元素都是一个数对，其间包括了文档的 ID 号，以及该词出现在该文档中位置的偏移值。

注 3： 此处关于全文搜索的讨论大部分都来自于我在 2003 年所发表的系列文章，On Search，它们可从如下地址找到 <http://www.tbray.org/ongoing/When/200x/2003/07/30/OnSearchTOC>。该系列文章涵盖了大量关于搜索的主题，包括用户体验，质量控制，自然语言处理，人工智能，国际化，等等……

由于 posting 很小并且尺寸固定，同时它们的数目又非常巨大，使用二分查找的做法因而也就显得很自然了。对于 Google 和 Yahoo! 的具体实现如何，我不是很清楚，不过如果听到在他们那上万台服务器上的大部分运算都是对一个大的 posting 数组进行二分查找，我是一点都不会觉得惊讶的。

前几年那些熟悉搜索技术的人对于 Google 所提供的可搜索网页数目都在集体暗笑 (collective snicker)，在 20 亿这个数目持续了几年后，Google 突然宣布他们所能搜索的网页数目变得更多了，并且现在这个数目还在不断增加。我怀疑这可能是他们将 posting 中的文档标识符从 32 位改成了 64 位的结果。

对结果打分

给定一个词，找到一个 posting 链表并从中找出包含它的文档并非一门高深的技术。此外，对多个词进行 AND 以及 OR 的查找（以及词组查找）并将找到后的 posting 链表合并成新的链表这也并不复杂，至少从概念上来说是这样的。真正困难的地方在于，如何对找到的链表进行排序，让最优的结果最早显示。在计算机科学中有一门分支学科叫做信息检索 (Information Retrieval，也简称为 IR)，它就是专门用来解决该问题的。从历史看来，直到最近，它也没有得出让人满意的结果。

对网络的搜索

Google 以及其竞争者已经可以为其大量用户提供对海量数据的搜索，并且结果还不错。此处“不错”的意思是，高质量的结果被显示在所有结果的顶端，这也使得搜索结果的显示变得快捷起来。

得到高质量搜索结果的成就来自于许多方面，其中最引人注目的就是被 Google 命名为 PageRank 的算法，它主要依靠来自于网页中的链接：那些被大量超链接所指向的页面被认为是更受欢迎的页面，从而也就更被人所信任，最终就从所有的搜索结果中脱颖而出成为优胜者。

在实际中，这样做的结果也很不错。下面我们将再提供一些有趣的资料。首先，在 PageRank 出现之前，搜索领域的领头羊如 Yahoo! 和 Dmoz，他们提供的是分好类的结果；也就是说，证据表明，人们似乎更期望了解一件事物的流行程度而不是其内容。

其次，PageRank 只适合于这样的文档集合：在集合中，文档间存在着大量的链接。在当前，合乎该标准的文档集合有两个：万维网以及学术刊物上的文献（在过去的几十年间，我们同样在这些文献上采用了类 PageRank 的算法对它们进行排名）。

大型搜索引擎可以针对数据及用户进行扩展的能力实在是令人印象深刻。这一点主要是基于大量的并行应用：我们可以将一个大问题指派给大量的小型计算机进行计算而不是少数的几台大型计算机。对于posting来说有个好处，由于每个posting都和其他的posting无关，这也就使得我们很自然地就会将它们进行并行处理。

例如，由posting数组构成的用于二分查找的索引很容易就可以被用来分区处理(partition)。如果一个索引只包含英语单词，那么我们就可以很轻易地将其分为26个不同的分区（对于这种行为，业界所采用的术语为：切片，shard），每个分区对应于不同的单词首字母。然后我们就可以对每个切片进行任意的复制，接着大量的字词搜索查询就可以被分配到由多个搜索节点组成的集群中去执行。

这也带来了一个问题，如何对多个字词或者词组的搜索结果进行合并？这需要一定的创新，但这并不是我们在本章中所需要考虑的，此处我们所关心的就是，如何将简单的字词搜索功能并行化。

此处的讨论有点不公平，因为它绕开了大量的重要问题，尤其是，如何与英特网上的恶人进行斗争，以阻止他们从搜索引擎算法的研究中获益。

结论

我们很难想像会有这样的应用程序，在它里面我们不需要存储数据以及基于内容对数据进行查找。世界上最流行的应用程序，网络搜索很好地证明了这点。

在本章中，我们讨论了一些有关搜索的问题，但是我们特意绕开了传统的“数据库”领域以及那些需要使用到外部存储的搜索策略。在本章中，不论是在处理一条简单的文本，还是处理上亿的网页，搜索都是我们所讨论的重点。从程序员的角度来看，从多种可能中选择一种来实现的过程充满了乐趣。

正确、优美、迅速 (按重要性排序): 从设计 XML 验证器中学到的经验

Elliotte Rusty Harold

本章讲述关于两个 XML 输入验证程序的故事，第一个发生在 JDOM 中，第二个发生在 XOM 中。我自始至终参与了二者的开发过程。尽管这两个程序是完全独立的，没有共享任何代码，但后者在开发中借鉴了前者的设计思想。在我看来这段代码变得越来越优美，同时，运行速度也毫无疑问地变得更快。

在每一次成功的重构中，提高运行速度是最主要的动力，但在这个例子中，速度提升的同时伴随着代码优美度的提升。我希望打破“高效的代码往往是丑陋且难以解读的”这个怪论。相反，我相信在提升代码优美度的同时也能够带来执行速度的提升，尤其是在当今拥有新型优化编译器、即时 (Just-in-time) 编译器，精简指令集 (RISC) 体系结构以及多核处理器的计算机技术中。

XML 验证器的作用

XML 通过严格执行某些规则使其具备了协同工作的能力，比如有些规则规定 XML 文档中可以或不可以出现的内容。除去一些很少的例外，一个遵照规则实现的处理器可以处理任何结构正确的 XML 文档，同时可以识别出（但不去处理）结构错误的 XML 文档。

这确保了在不同平台、不同解析器以及不同编程语言之间的高度互用性。你无需因为你的解析器是用 C 语言编写，且运行在 UNIX 平台上而担心它不能读取我用 Java 编写的且运行在 Windows 上的生成器所撰写的文档。

确保 XML 的正确性，通常需要包含两个冗余的数据检查：

1. 输入验证。当解析器读取一个 XML 文档时，检查该文档的结构正确性，同时还可以选择验证文档的有效性。文档结构的正确与否是指纯粹的语法检查，比如每一个开始标签（tag）是否有一个与之对应的结束标签。该验证功能是所有解析器都必须实现的。有效性意味着只有在文档类型定义（DTD）中出现的元素或者属性才可以根据定义出现在恰当的位置上。
2. 输出验证。当通过 XML 的 API，例如 DOM、JDOM 或 XOM 生成 XML 文档时，解析器检查所有传递给 API 的字符串以确定它们在相应的 XML 中的合法性。

尽管在 XML 的规范说明中更为详细地定义了输入验证，但是输出验证同样重要。特别是在调试时，它对判断代码是否正确起到了关键的作用。

问题所在

在 JDOM 的第一个 beta 版本中并没有验证用于创建元素名、文本内容或者其他任何信息的字符串。程序可以很自由地生成包含空白字符的元素名称、以“-”结尾的注释，不包含任何内容的文本节点，以及其他格式错误的内容。确保生成的 XML 文档的正确性的工作全部交给客户端程序员去处理。

这一点困扰了我。尽管 XML 与一些其他方法相比要更为简单一些，但还没有简单到可以不用研究领域知识就全然掌握的程度，例如我们要清楚在 XML 名字和正文内容中，哪些 Unicode 编码点（code point）是合法的，哪些是非法的。

JDOM 的目标是成为一个把 XML 推广和普及的 API。与 DOM 不同的是，学习 JDOM 并不需要一个长达两周的课程以及昂贵的专业导师。为了实现这个目标，JDOM 需要尽可能地为程序员承担理解 XML 的负担，从而使程序员不会犯错误。

有多种方式可以使 JDOM 满足这个设计要求。其中一些方式是由于 JDOM 自身的数据模型直接产生的。比如，在 JDOM 中不允许出现交迭的元素（elements）（`<p>Sally said, <quote>let's go the park.</p>. Then let's play ball.</quote>`）。因为 JDOM 的内部表示是一棵树，因此 JDOM 不能产生上例中的结构。然而，许多其他的约束则需要被明确地检测，例如：

- 元素名、属性名或者处理指令的名字必须是合法的 XML 名字。
- 局部名字不能包含冒号。
- 属性的命名空间不能与其父元素或者兄弟属性的命名空间冲突。
- 每一个 Unicode 的代理字符 (surrogate character) 出现在一个代理对中，一个代理对包含一个高位代理紧接着一个低位代理。
- 处理指令数据不包含双字节字符串 “?>”。

每当客户端在以上这些情况中提供一个字符串时，需要检查这个字符串是否满足相应的约束。检查的细节虽然各不相同，但基本的方法是相同的。

为了进行本章的讨论，下面我将分析 XML1.0 中元素名字的规则。

在 XML1.0 的说明 (其中在示例 5-1 中给出了一部分) 中，规则用 BNF 范式给出。这里 $\#x\text{dddd}$ 表示一个用十六进制数 dddd 表示的 Unicode 编码点。 $[\#x\text{dddd}-\#x\text{eeee}]$ 表示从 $\#x\text{dddd}$ 到 $\#x\text{eeee}$ 的所有编码点。

示例 5-1：分析 XML 名字的 BNF 语法（有删节）

```

BaseChar   ::= [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6]
NameChar   ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar |
Extender
Name        ::= (Letter | '_' | ':') (NameChar)*
Letter      ::= BaseChar | Ideographic
Ideographic ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]
Digit       ::= [#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9]
          | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F]
          | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF]
          | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F]
          | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29]
Extender    ::= #x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640 | #x0E46 | #x0EC6
          | #x3005 | [#x3031-#x3035] | [#x309D-#x309E] | [#x30FC-#x30FE]
          | [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131]
          | [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E]
          | [#x0180-#x01C3] ...
CombiningChar ::= [#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
          | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD] | #x05BF
          | [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652] | #x0670
          | [#x06D6-#x06DC] | [#x06DD-#x06DF] | [#x06E0-#x06E4]
          | [#x06E7-#x06E8] | [#x06EA-#x06ED] ...

```

因为需要考虑 90,000 余个的 Unicode 字符，完整的设置规则要用几页纸。在这个示例中特地删减了针对 BaseChar 和 CombiningChar 的规则。

为了验证一个字符串是否是一个合法的 XML 名字，必须迭代地访问字符串中的每一个字符并检查它是否是一个由 NameChar 定义的合法字符。

版本 1：简单的实现

我对 JDOM 最初的贡献（参见示例 5-2）只是简单地遵照了 XML 规则，把验证工作交给 Java 的 Character 类。当 XML 名字不合法时，CheckXMLName 方法将返回一个错误信息；当名字合法时，返回一个 NULL。这个设计本身就值得商榷，它应该在名字不合法时抛出一个异常，而在其他情况下返回空。在本章之后的几节中，你会看到在未来版本中是如何解决这个问题的。

示例 5-2：第一个版本的名字字符验证

```
private static String checkXMLName(String name) {
    // 不能为空或者 null
    if ((name == null) || (name.length() == 0) || (name.trim().equals("")))
        return "XML names cannot be null or empty";
    }

    // 开头不能是数字
    char first = name.charAt(0);
    if (Character.isDigit(first)) {
        return "XML names cannot begin with a number.";
    }
    // 开头不能是 $
    if (first == '$') {
        return "XML names cannot begin with a dollar sign ($).";
    }
    // 开头不能是 -
    if (first == '-') {
        return "XML names cannot begin with a hyphen (-).";
    }

    // 确保有效的内容
    for (int i=0, len = name.length(); i<len; i++) {
        char c = name.charAt(i);
        if ((!Character.isLetterOrDigit(c))
            && (c != '-')
            && (c != '$')
            && (c != '_')) {
            return c + " is not allowed in XML names.";
        }
    }

    // 如果代码执行到这里，那么名字就是有效的
    return null;
}
```

这个方法很直接且易于理解。但遗憾的是，在以下这些情况中它是错误的：

- 它允许在名字中包含冒号。因为 JDOM 努力维持命名空间的良好形式，因此必须解决这个问题。

- Java 的 `Character.isLetterOrDigit` 以及 `Character.isDigit` 方法并没有完全遵照 XML 中关于字母和数字的定义。Java 把一些 XML 中非法的字母视作合法，或者正好相反。
- Java 中的验证规则在每个版本中都有所变化，而 XML 的验证规则没有变化。

尽管如此，这仍然是一个合理的初次尝试。它确实能识别出大部分非法名字且不会拒绝许多合法名字。通常所有的名字都是 ASCII 码字符，在这种情况下这个版本工作得非常好。即便如此，由于 JDOM 不断努力地扩展其可用性，一个完全遵照 XML 规则的改进版呼之欲出。

版本 2：模拟 BNF 语法 —— 复杂度 O(N)

我对 JDOM 的第二个贡献是手工把 BNF 范式翻译成一系列 `if-else` 语句。结果类似于示例 5-3 中所给出的代码。你会注意到，这个版本的实现有一些复杂。

示例 5-3：基于 BNF 的名字验证

```

private static String checkXMLName(String name) {
    // 不能为空或者 null
    if ((name == null) || (name.length() == 0)
        || (name.trim().equals("")))
        return "XML names cannot be null or empty";
    }

    // 开头不能是数字
    char first = name.charAt(0);
    if (!isXMLNameStartCharacter(first)) {
        return "XML names cannot begin with the character \'" +
            first + "\'";
    }
    // 确保有效的内容
    for (int i=0, len = name.length(); i<len; i++) {
        char c = name.charAt(i);
        if (!isXMLNameCharacter(c)) {
            return "XML names cannot contain the character \'" + c + "\'";
        }
    }

    // 如果代码执行到这里，那么名字就是有效的
    return null;
}

public static boolean isXMLNameCharacter(char c) {

    return (isXMLLetter(c) || isXMLDigit(c) || c == '.' || c == '-'
        || c == '_' || c == ':' || isXMLCombiningChar(c))

```

```

        || isXMLExtender(c));
    }

public static boolean isXMLNameStartCharacter(char c) {
    return (isXMLLetter(c) || c == '_' || c == ':');
}

```

在示例 5-3 中，我们不是简单地再次使用 Java 的 `Character.isLetterOrDigit` 以及 `Character.isDigit` 方法，而是在 `checkXMLName` 方法中调用了 `isXMLNameCharacter` 和 `isXMLNameStartCharacter`。这些方法进一步调用其他方法来匹配次级BNF范式，这些范式分别处理不同类型的字符：字母，数字，组合字符以及这些字符的混合形式。在示例5-4中展示了其中的一个方法：`isXMLDigit`。注意在这个方法中不仅仅涉及ASCII数字，同时还涉及了包括 Unicode 2.0 中定义的其他数字字符。`isXMLLetter`、`isXMLCombiningChar` 以及 `isXMLExtender` 方法也是如此，只是代码更加长一些而已。

示例 5-4：基于 XML 的数字字符验证

```

public static boolean isXMLDigit(char c) {

    if (c >= 0x0030 && c <= 0x0039) return true;
    if (c >= 0x0660 && c <= 0x0669) return true;
    if (c >= 0x06F0 && c <= 0x06F9) return true;
    if (c >= 0x0966 && c <= 0x096F) return true;

    if (c >= 0x09E6 && c <= 0x09EF) return true;
    if (c >= 0x0A66 && c <= 0x0A6F) return true;
    if (c >= 0x0AE6 && c <= 0x0AEF) return true;

    if (c >= 0x0B66 && c <= 0x0B6F) return true;
    if (c >= 0x0BE7 && c <= 0x0BEF) return true;
    if (c >= 0x0C66 && c <= 0x0C6F) return true;

    if (c >= 0x0CE6 && c <= 0x0CEF) return true;
    if (c >= 0x0D66 && c <= 0x0D6F) return true;
    if (c >= 0x0E50 && c <= 0x0E59) return true;

    if (c >= 0x0ED0 && c <= 0x0ED9) return true;
    if (c >= 0x0F20 && c <= 0x0F29) return true;

    return false;
}

```

这个方法满足了升级版本的基本需求。它可以工作，而且它的算法也很清楚：从 XML 规范到这段代码的映射非常清晰。我们似乎可以宣布胜利然后回家了。

然而，一个丑陋的魔鬼：性能抬起了头。

版本 3：第一个复杂度 $O(\log N)$ 的优化

正如 Donald Knuth 曾经说到的，“不成熟的优化是编程中所有罪恶之源。”虽然程序员并不总是认为优化很重要，但有时候优化确实有其必要性。这个例子正是说明了这些少数的情况。

通过性能分析，我们发现 JDOM 在验证 XML 格式上花费了很大一部分时间。名字中的每个字符要求一些检查，同时 JDOM 只有在把待验证字符和所有可能的合法字符相比较之后才能识别出一个非法字符。因此，验证的次数随着编码点的值直线上升。项目的维护人员开始抱怨，也许 XML 格式验证并不像想像的那么重要。他们或许把它视作一个可选的功能或者完全丢弃它。我个人并不愿意为了速度而牺牲正确性，但是很明显，如果没有人做改进工作的话，我肯定会去掉这个功能。幸运的是，Jason Hunter 做了这个改进工作。

Hunter 用一种非常聪明的方法重组了第一个版本中的代码，如示例 5-5 所示。之前，即使在普通的情况下，一个合法的字符也需要超过 100 次的验证，以核实其是否处于某个非法字符范围内。Hunter 注意到，如果我们可以同时识别合法和非法字符，那么就可以更快地返回一个真值。在所有名字和内容都是 ASCII 字符的情况下，这一改动所带来的好处更加明显，因为这些字符是我们最先要验证的字符。

示例 5-5：优化的数字字符验证过程

```
public static boolean isXMLDigit(char c) {  
  
    if (c < 0x0030) return false; if (c <= 0x0039) return true;  
    if (c < 0x0660) return false; if (c <= 0x0669) return true;  
    if (c < 0x06F0) return false; if (c <= 0x06F9) return true;  
    if (c < 0x0966) return false; if (c <= 0x096F) return true;  
  
    if (c < 0x09E6) return false; if (c <= 0x09EF) return true;  
    if (c < 0x0A66) return false; if (c <= 0x0A6F) return true;  
    if (c < 0x0AE6) return false; if (c <= 0x0AEF) return true;  
  
    if (c < 0x0B66) return false; if (c <= 0x0B6F) return true;  
    if (c < 0x0BE7) return false; if (c <= 0x0BEF) return true;  
    if (c < 0x0C66) return false; if (c <= 0x0C6F) return true;  
  
    if (c < 0x0CE6) return false; if (c <= 0x0CEF) return true;  
    if (c < 0x0D66) return false; if (c <= 0x0D6F) return true;  
    if (c < 0x0E50) return false; if (c <= 0x0E59) return true;  
  
    if (c < 0x0ED0) return false; if (c <= 0x0ED9) return true;  
    if (c < 0x0F20) return false; if (c <= 0x0F29) return true;  
  
    return false;  
}
```

在前面的实现中，在判断出一个字符不是数字之前，测试了所有可能的数字范围，包括很少出现的é和φ。新的方法可以更快速地判断出一个字符不是有效数字。类似的方法可以应用在对字符，扩展字符，混合字符的验证过程中，甚至会带来更为显著的改善。

这个版本并没有削除花在 XML 格式验证上的时间，但它确实减少了时间开销，并足以让项目维护者满意，至少在元素名字的验证上是这样的。纯文本内容（PCDATA）的验证仍然没有被放在正式产品中，但这并不是个重要的问题。

版本 4：第二次优化：避免重复验证

此时，XML 验证的时间已经减少到原来的四分之一左右，并且不再是一个大问题。版本 3 基本上就是 JDOM1.0 版本中的代码。然而，此时我已经觉得 JDOM 不再足够优秀，并且开始希望我是否可以做得更好。我更多地从 API 的设计角度而不是性能角度发现问题。同时我还关心正确性，因为 JDOM 仍然没有实现所有能够进行的检查，这导致了使用 JDOM 仍然可能（尽管很难）创建出结构错误的 XML 文档。基于这些原因，我开始着手开发 XOM。

XOM 不像 JDOM，它不会牺牲正确性来获取性能。在 XOM 的设计原则中，正确性始终是处于第一位。不过，对那些选择 XOM 替代 JDOM 的人来说，XOM 的性能至少应该与 JDOM 处于同一个量级或者超过 JDOM。所以，我们现在应该再次对 XML 验证问题重拳出击了。

在版本 3 中对优化的努力已经提高了 `checkXMLName` 方法的性能，但是我希望在接下来的优化中把这个函数完全删除。这么做的原因是如果 XML 的输入来自一个已知的可信源头，那么就不需要总是检查它是否合法。尤其是 XML 解析器会先于验证器完成很多必要的验证，所以没有理由重复地进行这项工作。由于构造函数总是会检测正确性，所以这将对解析速度性能产生很大的影响，在实际的应用程序对每个文档的处理过程中这将占用很大一部分时间开销。

针对不同输入类型使用不同的方法可以解决这个问题。我决定，对于文档中那些解析器已经读取并验证过的字符串，在创建对象时构造函数就不应该再去验证元素的名字。与之相反，对于那些从类库的客户端传入的字符串，构造函数在创建对象时需要予以检查。所以需要两个不同的构造函数，一个提供给解析器，另一个提供给其他调用者。

JDOM 的开发者也曾经考虑过这个优化，但是由于糟糕的包（package）设计而没有采纳这一方案。在 JDOM 中，用于从 SAX 解析器中创建文档的 `SAXBuilder` 类被放置于 `org.jdom.input` 包中。元素、文档、属性以及其他节点类则被置于 `org.jdom` 包

中。这意味着，被 XML 对象的创建程序所调用的构造函数，无论做验证的和不做验证的，都必须是公有的。所以，其他的客户程序也可以调用那些不做验证的构造函数，从而导致 JDOM 生成结构错误的 XML 文档。后来，在 JDOM1.0 中，开发者推翻了初始的设计，决定引入一个特殊的工厂类来接收未经检测的输入。诚然，这个工厂类速度比原来的方式要快，但这种设计在检测系统中留下了可能会造成麻烦的后门。造成这一问题的根本原因就是把 JDOM 的创建程序分割到了输入 (`org.jdom.input`) 和核心对象 (`org.jdom`) 这两个包中。

注意

在 Java 代码中，把包分割得过细是一个常见的违反模式的设计。这个设计通常会导致程序员要么把不应该开放的方法设置成公有的，要么只提供少量的功能，这两种选择都是不好的。

不要仅用包来组织类结构。每一个包都应该完全独立于其他包的内部实现。如果在你的程序或库中，两个类互相访问对方的需求远高于这两个类对其他类的访问需求，那么这两个类就应该被放到同一个类中。

在 C++ 中，友元函数很好地解决了这个问题。虽然 Java 目前并不支持友元函数，但是 Java7 可能会允许对子包拥有更多的访问权限，而普通的公有成员没有这种权限。

当我着手开发 XOM 时，我从 JDOM 中吸取了经验教训，把处理输入的类和处理核心节点的类放在一个包中。这意味着我可以给解析器提供包内保护的 (package-protected) 且不做验证的方法，但这些方法不能被其他包里的客户类调用。

XOM 的机制非常的直接。每一个节点类都拥有一个私有的，不含任何参数的构造函数，同时有一个包内保护的工厂方法 `build`，负责验证该函数并处理不经验证的设定字段。示例 5-6 中用 `Element` 类的相关代码展示了这一机制。相比较其他解析器，XOM 事实上在处理命名空间上有一些吹毛求疵，所以它要做这样的检查。不过，它仍然可以忽略许多冗余的验证。

示例 5-6：基于解析器的数字字符验证

```
private Element() { }

static Element build(String name, String uri, String localName) {
    Element result = new Element();
    String prefix = "";
    int colon = name.indexOf(':');
    if (colon >= 0) {
        prefix = name.substring(0, colon);
    }
}
```

```
        result.prefix = prefix;
        result.localName = localName;
        // 我们确实需要验证这里的 URI，因为解析器能够允许一些 XOM
        // 禁止的相对 URI，这是由于标准 XML 的原因。
        // 但是我们只需验证它是一个绝对基 URI，而无需验证是否存在冲突
        if (!"".equals(uri)) Verifier.checkAbsoluteURIReference(uri);
        result.URI = uri;
        return result;
    }
}
```

这一方法显著地提高了解析阶段的速度，因为与前面的版本相比，它并不需要重复大量的验证工作。

版本 5：第三次优化：复杂度 $O(1)$

当我实现了在前面节中介绍的构造函数以及一些其他优化之后，XOM 的速度已经快到不需要我再做什么额外的工作了。读取 XML 文档的性能基本上仅受到解析器速度的限制，并且在文档创建过程中的瓶颈也非常的少了。

然而，用户在面对不同的情况时存在着不同的问题。尤其是某些用户通过撰写自定义的构建器把非 XML 结构的文档读入到 XOM 树中时。由于他们没有使用 XML 解析器，所以就不能够跳过名字的验证。虽然现在的数量较之前已经减少了很多，这些用户仍然视验证为瓶颈。

我并不想完全关闭验证，除非有要求这么做。但很显然必须要提高验证阶段的速度。我采用的是一个古老的优化方法：查找表的方法。在一个查找表方法中，首先创建一张表，包含所有已知输入的解答。当给出一个输入时，编译器可以不经计算而直接从表中查到答案。这是一个 $O(1)$ 复杂度的操作，而且它的性能逼近理论上的最大值。当然，这在具体的实现中存在着一些问题。

在 Java 中实现表查找的最简单方法是使用 `switch` 语句。`javac` 把这个语句编译成一张数值表，且数值以字节码的形式存储。根据 `switch` 中 `case` 语句的不同，编译器将创建以下两种双字节指令中的一种。如果 `case` 的值是连续的（例如，`72 ~ 189` 中间没有跳过任何一个整数），编译器会使用一个更为高效的表来实现。然而，如果中间跳过了任何值，编译器则会使用间接且较为低效的查找表来实现。

注意

这种行为并不一定会发生，而在更新的虚拟机上可能更不会发生，但在
我测试过并检查过的虚拟机上确实会发生这种行为。

对于较小的表（几百个 cases 或者更少），可以在中间填入默认值。在示例 5-7 中给出了一个简单的例子来测试一个字符是否是十六进制数字。测试从最小的真值 ‘0’ 开始，结束于最大的真值 ‘f’。每个在 ‘0’ 到 ‘f’ 间的字符必须出现在一个 case 语句中。

示例 5-7：验证 16 进制字符的 switch 语句

```
private static boolean isHexDigit(char c) {  
  
    switch(c) {  
        case '0': return true;  
        case '1': return true;  
        case '2': return true;  
        case '3': return true;  
        case '4': return true;  
        case '5': return true;  
        case '6': return true;  
        case '7': return true;  
        case '8': return true;  
        case '9': return true;  
        case 'A': return true;  
        case 'B': return true;  
        case 'C': return true;  
        case 'D': return true;  
        case 'E': return true;  
        case 'F': return true;  
        case 'G': return false;  
        case 'H': return false;  
        case 'I': return false;  
        case 'J': return false;  
        case 'K': return false;  
        case 'L': return false;  
        case 'M': return false;  
        case 'N': return false;  
        case 'O': return false;  
        case 'P': return false;  
        case 'Q': return false;  
        case 'R': return false;  
        case 'S': return false;  
        case 'T': return false;  
        case 'U': return false;  
        case 'V': return false;  
        case 'W': return false;  
        case 'X': return false;  
        case 'Y': return false;  
        case 'Z': return false;  
        case '[': return false;  
    }  
}
```

```

        case '\\': return false;
        case ']': return false;
        case '^': return false;
        case '+': return false;
        case '`': return false;
        case 'a': return true;
        case 'b': return true;
        case 'c': return true;
        case 'd': return true;
        case 'e': return true;
        case 'f': return true;
    }
    return false;
}

```

虽然这个程序很长，但比较浅显易懂。程序并不复杂，读者很容易看懂程序要做的工作，这是它的优点。然而，尽管 switch 语句浅显易懂，但是当遇到包含很多 case 的情况时，还是会遇到困难的。例如，XML 的字符检查 switch 语句牵涉到几万个 cases 语句。我曾经试着写一个 switch 语句来处理这个庞大的 case 群，结果却发现 java 规定一个方法的字节数量最大不能超过 64KB。这一规定使我们需要另外一种解决方案。

尽管编译器和运行时的要求限制了在字节码中存储的查找表的大小，但是我还可以把它隐藏在其他的地方。我开始定义简单的二进制格式，在基本多语言平面（Basic Multilingual Plane, BMP）的 65536 个 Unicode 中，每一个编码点都对应了一个字节。该字节中包含了 8 个比特位，用来确定最重要的字符属性。例如，当字符是合法的 PCDATA 内容时，比特位 1 被置位，当不合法时被复位。当字符可以被用于 XML 名字中时，比特位 2 被置位，否则复位。当字符可以作为 XML 名字的首字符时，比特位 3 被置位，否则复位。

按照 XML 规范，我编写了一个简单的读取 BNF 语法的程序，为所有 65536 的编码点计算标记值，然后把它存放在一个巨大的二进制文件中。我把这个二进制数据文件与我的原始代码保存在一起，并且修改了 Ant 编译任务从而把它复制到生成目录中（示例 5-8）。

示例 5-8：保存并复制二进制查找表

```

<target name="compile-core" depends="prepare, compile-jaxen"
    description="Compile the source code">
    <javac srcdir="${build.src}" destdir="${build.dest}">
        <classpath refid="compile.class.path"/>
    </javac>
    <copy file="${build.src}/nu/xom/characters.dat"
        tofile="${build.dest}/nu/xom/characters.dat"/>
</target>

```

自此，jar 任务将把查找表和 class 文件捆绑在一起，因此发布二进制代码的时候不增加

额外的文件或任何其他的依赖性。然后验证器类可以在运行时使用类装载器来找到这个文件，参照示例 5-9。

示例 5-9：装载二进制查找表

```
private static byte[] flags = null;

static {
    ClassLoader loader = Verifier.class.getClassLoader();
    if (loader != null) loadFlags(loader);
    // 如果无法装载，可以尝试另一个ClassLoader
    if (flags == null) {
        loader = Thread.currentThread().getContextClassLoader();
        loadFlags(loader);
    }
}

private static void loadFlags(ClassLoader loader) {
    DataInputStream in = null;
    try {
        InputStream raw = loader.getResourceAsStream("nu/xom/characters.dat");
        if (raw == null) {
            throw new RuntimeException("Broken XOM installation: "
                + "could not load nu/xom/characters.dat");
        }
        in = new DataInputStream(raw);
        flags = new byte[65536];
        in.readFully(flags);
    }
    catch (IOException ex) {
        throw new RuntimeException("Broken XOM installation: "
            + "could not load nu/xom/characters.dat");
    }
    finally {
        try {
            if (in != null) in.close();
        }
        catch (IOException ex) {
            // 以下代码省去
        }
    }
}
```

既然查找表被放在内存中，因此要检查任何一个字符只需要简单地执行一些位操作来查

看是否在表中。由于表是只读的，因此我们不能修改它。如果要对表进行修改，那么就必

找数组。在示例5-10中给出了一段新代码来验证元素或者局部名字中不可以包含冒号的字符串。我们所要做的只是在表中查找标记并且把相关的位与期望的属性相比较。

示例 5-10：使用查找表来验证名字

```
// 在字符查找表中的位标志常量
private final static byte XML_CHARACTER          = 1;
private final static byte NAME_CHARACTER          = 2;
private final static byte NAME_START_CHARACTER    = 4;
private final static byte NCNAME_CHARACTER        = 8;

static void checkNCName(String name) {
    if (name == null) {
        throwIllegalNameException(name, "NCNames cannot be null");
    }

    int length = name.length();
    if (length == 0) {
        throwIllegalNameException(name, "NCNames cannot be empty");
    }

    char first = name.charAt(0);
    if (((flags[first] & NAME_START_CHARACTER) == 0) {
        throwIllegalNameException(name, "NCNames cannot start " +
            "with the character " + Integer.toHexString(first));
    }

    for (int i = 1; i < length; i++) {
        char c = name.charAt(i);
        if (((flags[c] & NCNAME_CHARACTER) == 0) {
            if (c == ':') {
                throwIllegalNameException(name, "NCNames cannot contain colons");
            }
            else {
                throwIllegalNameException(name, "0x"
                    + Integer.toHexString(c) + " is not a legal NCName character");
            }
        }
    }
}
```

现在的名字字符验证是复杂度 $O(1)$ 的操作，并且检查一个完整名字的复杂度是 $O(n)$ ，其中 n 是名字的长度。你可以像我一样通过修改代码来提高常量系数，但是很难知道这在进行必要的检测同时，如何能够变得更快。不过，我们现在还没有做完。

版本 6：第四次优化：缓存

如果你无法使验证器运行的更快一些，那么剩下的惟一选择就是不做验证工作，或至少不做那么多的验证工作。这种方法是由Wolfgang Hoschek提议的。他注意到在XML文

档中同样的名字会重复出现。例如，在 XHTML 文件中，只有大约 100 个不同元素的名字，并且其中的一些元素占据了大多数名字（p、table、div、span、strong 等）。一旦核实某个名字是合法的，就可以把它存放在某处。当下次再看到这个名字时，首先检查它是否是一个已经核实过的名字，如果是，则立即接受并且不再作任何检查。

但是，此时你必须非常仔细。在一个集合中寻找某些名字（尤其是一些短名字）或许会比重新核实它们占用更多的时间。惟一的方法是通过在不同的虚拟机上用不同类型的文档来试验，并记录和分析缓存的模式。你可能需要调整例如集合的大小等参数来适合不同类型的文件，并且在某种类型的文档上性能出色的参数在另一种类型的文档上可能不能很好地运行。此外，如果集合需要在线程之间共享，那么线程竞争也可能成为一个严重的问题。

因此，对于元素名字我没有使用这种方式，而是将其用于命名空间的 URI（统一资源识别符）。与元素名字相比，命名空间的 URI 验证消耗了更加昂贵的验证开销，并且重复出现的频率更高。例如，许多文档只有一个命名空间 URI，并且只有非常少的文档拥有超过四个的命名空间，因此这种情况下潜在的收益很大。在示例 5-11 中显示了一个内部类，XOM 在核实了命名空间 URI 时使用这个类把已经验证过的命名空间缓存起来。

示例 5-11：已验证的命名空间 URI 的缓存

```
private final static class URICache {  
    private final static int LOAD = 6;  
    private String[] cache = new String[LOAD];  
    private int position = 0;  
  
    synchronized boolean contains(String s) {  
  
        for (int i = 0; i < LOAD; i++) {  
            // 这里我假设这个命名空间 URI 是内部的。这种情况很常见，  
            // 但却并非总是如此。如果命名空间以前不是内部的也不会造成  
            // 破坏。当命名空间 URI 以前不是内部的，那么使用 equal() 比 ==  
            // 更快。但如果真是的话，将会更慢。  
            if (s == cache[i]) {  
                return true;  
            }  
        }  
        return false;  
    }  
  
    synchronized void put(String s) {  
        cache[position] = s;  
        position++;  
        if (position == LOAD) position = 0;  
    }  
}
```

在这个类中有些令人惊奇的特点。首先，它没有使用哈希映射或哈希表，而是使用固定大小的数组以及线性搜索。因为对于小的列表来说，哈希查找的速度比简单的数组遍历要慢。

其次，如果数组被填满了，将不会被扩展。新数据从开始的第一个位置覆盖旧数据。这种行为将会使数据结构可能遭受一些降低性能的攻击，虽然仍能正确地工作，但速度会变慢。不过任何一个真实情况中的 XML 文档都极不可能会有这样的问题。很少有文档会超过六个命名空间，并且在这些罕见的情况下，命名空间也倾向于局部化而非随机地出现在整个文件中。由于重新设置缓存数组导致的性能影响非常短暂。

我能想像出的一个固定大小数组会带来麻烦例子是多个线程同时解析类型差异很大的文件。在这种情况下，也许会超出六个命名空间极限。一种可能的解决方案是将缓存作为每个线程的局部变量。

如果缓存的对象是元素名字而不是命名空间 URIs，那么这些担心会变得更加明显。在这种情况下，有更多名字需要缓存并且名字的长度比较短。此时也许使用表比使用一个简单的数组更合理。或许验证工作会更加快速；但我还未进行必要的具体测试以确定最佳的设计，不过我计划在 XOM 1.3 中作这些试验。

从故事中学到的

我们可以从这个故事中得到一些有益的启示：不要让对性能的考虑妨碍你做正确的事情。你总能通过一些技巧使代码运行得快一些。但如果是一个糟糕的设计方案，那么你很难轻易地解决问题。

程序总是能够渐渐地变快，而不是变慢。使用更快的 CPU 是一个办法，但并不是最根本的解决方法。算法的改善对速度提升的贡献更为重要。所以，该如何设计程序就如何设计程序。然后，必须是然后，再关心性能。在多数情况下，你会发现第一版的程序已经足够快了，并且甚至不需要使用这里介绍的这些技巧。但是，使优美但速度较慢的代码变得更快总比使快速但难看的代码变得优美要更加容易。

集成测试框架：脆弱之美

Michael Feathers

对于什么是好的设计，我有一些自己的想法愿与大家分享。其实不仅是我，我们每个人都有自己的标准。我们在长期的实践中一点一滴地积累起一些认识，并用它们来指导我们的工作。比如当你忍不住要使用公有变量时，你可能就会想起公有变量通常是糟糕设计的症状。再比如当你看到实现继承时，你可能就会想起应该尽可能用委托而不是继承（注 1）。

记住以上这些规则固然很有用，它们能帮助我们作出更好的设计。然而我们一定要记住，它们只是原则，而不是教条，否则便可能反受其害；比如我们可能会作出各方面都循规蹈矩但偏偏就是不到家的设计。

这一想法得追溯到2002年，那年Ward Cunningham发布了他的自动测试框架——FIT。顿时，以上想法在我脑子里变得无比清晰起来。FIT由几个优雅的Java类构成，这些类几乎无视Java社群中的一切设计原则，而且它们的每处细小的不同都很引人注目。它们跟那些循规蹈矩的设计形成了鲜明的对比。

注 1: Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.

对我来说，FIT 诠释了什么是美丽的代码。它让我们思考设计的本质，即具体问题具体分析。

本章我们将分析 FIT 的整个设计，我用的是 FIT 最早的发布版本之一。你会看到 FIT 的设计是怎样在主流的 Java 和 OO 框架开发原则之外另辟蹊径的，你也会看到 FIT 的设计当年是如何动摇了我对原先设计的固执认识的。我不知道你看完本章之后是不是也会像我当年一样受到同样的震撼，我希望你会。我也希望我能够讲清楚 FIT 设计中的特别之处。

三个类搞定一个验收测试框架

FIT 解释起来其实也比较简单。它是一个不算大的测试框架，利用 HTML 表格来编写可执行的应用测试。每种表格都由一个程序员定义的类来处理，这种类也叫 fixture 类。在处理一个 HTML 页时，FIT 会为页面上的每张表创建一个 fixture 对象，后者将表格作为输入传给你要验证的代码，具体来说就是：它会读取表格单元内的值，应用，然后检查应用返回的值与表格内的期望值是否相符，如果不符，则将该单元格设为红色，反之则设为绿色。

表格中的第一个单元格内是 fixture 类的类名，该类会被用来对该表格进行处理。例如，图 6-1 中的表格会被 MarketEvaluation 类处理。图 6-2 是处理之后的表格——如果在彩色屏幕上你会看到阴影的单元格是红色的，表示验证失败。

MarketEvaluation				
Symbol	Shares	Price	Portfolio	Attempt Fill?
PLI	1100	1560	no	no

图 6-1：在 FIT 处理之前显示的 HTML 表格

MarketEvaluation				
Symbol	Shares	Price	Portfolio	Attempt Fill?
PLI	1100	1560	no	no no expected yes actual

图 6-2：在 FIT 处理之后显示的 HTML 表格

FIT 最关键的特性就是文档可以被用来测试。例如你可以将一个表格嵌在需求文档里面，

然后把这个需求文档交给FIT去运行，看看这些表格中指定的行为是不是存在于你的软件当中。这些带有表格的文档可以直接用HTML语言写，也可以用Word写，然后存为HTML格式；只要是支持“保存为HTML文件”的编辑器都可以。由于一个FIT fixture本质上是一段代码，因此它可以在任意层面调用你想要测试的应用中的任意一段功能。总之它完全在你的掌控之中。

关于FIT以及FIT适用的问题领域我就不多说了；FIT的网站（<http://fit.c2.com>）上有更多的信息。本文主要是想介绍FIT的设计，以及FIT在设计中作出的一些有趣的选择。

FIT的核心只有三个类：Parse、Fixture以及TypeAdapter。它们的成员方法和成员数据，以及它们之间的关系可以从图6-3中一目了然。

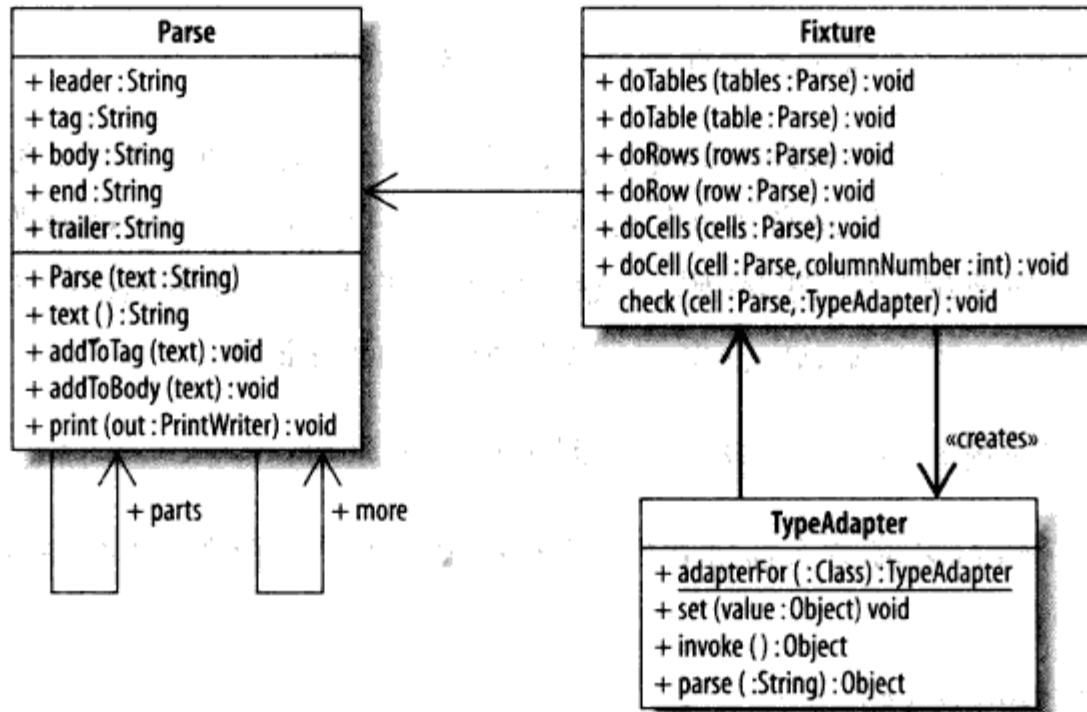


图 6-3：FIT 类之间的关系

下面让我们详细分析一下里面蕴含的设计。

简单来说，Parse类对应于文档中的HTML代码。Parse的构造函数接受一个字符串，并递归构造出一棵由Parse对象构成的树，这棵树的节点之间是通过Parse的parts和more两个成员来联系的。每个Parse对象都代表了文档的某个部分：每一张表格、表格的每一行、每一个单元格，都对应了一个单独的Parse对象。

Fixture类会遍历这棵Parse树，TypeAdapter则将测试值（数字、日期等）转换为文本或反之转换回来。Fixture类与你要测试的应用进行交互，通过后者的反馈结果来将相应单元格涂成红色（失败）或绿色（通过）。

FIT中的大部分工作都是在Fixture类的派生类中完成的。Fixture类定义了它们所接受的HTML表格的格式。比如说，如果你想要创建一个表格，其中包含一组要对你的应用执行的命令，那么你可以使用预定义的ActionFixture类。如果你想要将你的应用查询若干结果并将它们与一组期望值进行比较的话，你可以使用RowFixture类。FIT为你预备了这些简单的派生类，但你也可以自己从Fixture派生新类。

FIT是一个很有用的框架。我经常使用它。而且我不断惊奇地发现它那简简单单的三个核心类居然可以用来做那么多事情。许多框架完成同样的工作几乎要用比FIT多三四倍的类才行。

框架设计的挑战

以上就是FIT的架构。下面我来解释为什么FIT这么特别。

我们都知道，框架设计不是块好啃的骨头。框架设计最困难的地方在于，你的框架的代码不是你写的。所以一旦你发布了框架的API就无法再修改了，不管是修改方法的签名还是语意都会破坏使用了它们的代码。一般来说框架的用户在升级框架的时候是不愿意去修改既有代码的，也就是说你的框架必须做到完全向后兼容。

传统上解决这个问题的办法是在设计API时谨遵以下原则：

1. 在公开一个接口之前要三思而后行：控制对用户可见的东西，从而尽量减少框架公开的部分。
2. 使用接口(interface)。
3. 在你期望用户能够对框架进行扩展的点上提供良好定义的“挂钩点”。
4. 在你不希望用户进行扩展的点上禁止用户扩展。

以上这些指导原则，有的被写进了书中（注2），但大部分还是属于口耳相传的程序员文化。它们是框架设计者们设计框架遵循的惯例。

我们不妨来看一个FIT之外的例子，看看这些原则是如何被运用于实践中的。这个例子就是JavaMail的API。

注2：Effective Java, Joshua Bloch, Prentice Hall PTR, 2001.

Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, Krzysztof Cwalina and Brad Abrams, Addison-Wesley Professional, 2005.

如果你想要使用 JavaMail 来收取邮件，首先你必须获取一个 Session (会话) 对象的引用。Session 类有一个名叫 `getDefaultValue` 的静态方法，如下：

```
package javax.mail;

public final class Session
{
    ...
    public static Session getDefaultValue(Properties props);
    ...
    public Store getStore() throws NoSuchProviderException;
    ...
}
```

Session 类是 `final` 的。在 Java 里面，`final` 类的意思就是不能创建其派生类。此外 Session 类没有任何公有的构造函数，也就是说我们不能自己创建它的对象，而只能通过调用刚才提到的 `getDefaultValue` 静态方法来获取其对象。

一旦获取了一个 Session 对象之后，我们便可以使用其 `getStore` 方法来获取一个 Store 对象。Store 对象包含了邮件文件夹对象（其中包含了收取到的邮件），以及其他一些有用的东西。

Store 其实是一个抽象类。JavaMail 框架自身提供了 Store 的一些派生类，但我们无需关心 `getStore` 返回的到底是哪个派生类的对象，我们只要拿着这个 Store 的引用，用它来获取邮件即可。如果我们想在框架对 Store 对象作了改动的时候获得通知的话，可以往 Store 对象上挂一个监听器类，如下：

```
public abstract class Store extends javax.mail.Service {
    ...
    public void addStoreListener(StoreListener listener);
    ...
}
```

JavaMail 的这段设计体现了传统的框架设计风格。Session 类被声明为 `final`，以防止用户从它进行派生。另外，JavaMail 还声明了一个抽象类 Store 来提供一个用户扩展点：用户可以派生出各种不同的 store 对象。

然而，Store 却是一个“受管”的扩展点。我们没法决定 Session 对象返回什么 store 对象。我们可以对返回的 Store 对象进行配置，但也不是在代码中进行配置。框架把我们的选择限制得死死的。

另一方面，你可以把一个监听器对象挂到 Store 对象上，这是框架提供的一个“挂钩点”，这样用户无需从 Store 类进行派生便可以在 Store 对象被改动的时候收到通知了。

总的来说，JavaMail的API在自我保护方面做得不错：你需要遵循一组良好定义的步骤才能使用它，而且框架开发者也已经为将来的修改设好了“掩护”——他们可以对Session类内部作任何修改而不用担心有人会从它派生新类并重写其中的部分方法。

开放式框架

而FIT是一种非常不同的框架。从图6-3中的UML图中你可以看到，其核心框架中的绝大部分成分都是公有的，甚至成员数据都是如此。随便到其他框架的代码中看看，你会发现没有像FIT这样的。那么FIT的这种做法到底行不行呢？答案是肯定的，因为FIT代表的是一个开放式框架。它不像前面讲的那些传统框架那样具有一些设计好的扩展点，而是将整个框架完全向扩展开放。

让我们来看一看Fixture类。Fixture类用起来非常直接：你创建一个Fixture的实例，然后用Parse类给一个HTML文档创建一棵解析树，并将这棵树传给doTables方法，后者再对文档中的每个表格调用doTable方法，并传递相应的解析子树作为其参数。

doTable方法如下：

```
public void doTable(Parse table) {  
    doRows(table.parts.more);  
}
```

它调用的doRows方法如下：

```
public void doRows(Parse rows) {  
    while (rows != null) {  
        doRow(rows);  
        rows = rows.more;  
    }  
}
```

doRow方法接着调用doCells：

```
public void doRow(Parse row) {  
    doCells(row.parts);  
}
```

最终被调用的是doCell：

```
public void doCell(Parse cell, int columnNumber) {  
    ignore(cell);  
}
```

`ignore` 方法将该单元格涂成灰色，表示该单元格被忽略了：

```
public static void ignore (Parse cell) {  
    cell.addToTag(" bgcolor=\"#efefef\"");  
    ignores++;  
}
```

看到这里你可能会觉得奇怪，因为根据以上描述，似乎 `Fixture` 也并没有做什么有用的事情。它只是遍历整个文档，然后把每一个表格中的每一个单元格都涂成灰色就完了。但关键就在于，`Fixture` 本身虽然不做实质性的工作，但我们可以对以上函数调用序列上的每个函数进行重写，让它们做我们想做的事情，我们可以收集信息，保存信息，与被测试应用交互，标记特定单元格为红或绿。`Fixture` 把遍历一个 HTML 文档的过程缺省定义好了供我们去自定义其中的各个环节。

FIT 的这种做法与经典的框架设计原则似乎格格不入：用户不是把自己的代码“插入”到框架中，而是从一个类进行派生，然后重写其某些缺省的动作。此外框架也没有任何保护机制——技术上来说，用户只需要调用 `doTables` 方法，然而设计者却把整个调用链（从 `doTables` 到 `doCell`）全都给公开了。这么一来 FIT 就永远不能动这些方法了，一动则必然破坏客户代码。从经典的框架设计的角度来看，FIT 的设计是糟糕的，但不妨换个角度：如果设计者对那个遍历文档的调用序列非常有信心又如何呢？这个调用序列上的各个环节刚好一一对应了我们关心的 HTML 文档中的各个成分，而且这些环节都是很稳定的；很难想像 HTML 文档的什么改变会破坏它们。所以，永不改动它或许一点问题都没有。

一个 HTML 解析器可以简单到什么程度

除了是一个开放式框架之外，FIT 中还有其他令人惊讶的设计决策。前面曾提到，FIT 的 HTML 解析工作全部都是由 `Parse` 类来完成的。我最喜欢 `Parse` 类的一点就是它的构造函数负责构造起了一整棵树。

使用起来像下面这样（用一个 HTML 字符串为参数，调用 `Parse` 的构造函数）：

```
String input = read(new File(argv[0]));  
Parse parse = new Parse(input);
```

`Parse` 的构造函数递归构造出一棵由 `Parse` 对象构成的树，每个节点都对应该 HTML 文档的一部分。解析 HTML 的代码都在 `Parse` 的构造函数中。

每个 `Parse` 实例都有 5 个共有的成员，类型都是 `String`。此外还有两个 `Parse` 类型的引用，引用其他的 `Parse` 对象。

```
public String leader;
public String tag;
public String body;
public String end;
public String trailer;

public Parse more;
public Parse parts;
```

当你给你的 HTML 文档创建第一个 Parse 对象的时候，其实某种程度上你就把所有的 Parse 对象都创建了，之后你就只需使用 parts 和 tranverse 这两个成员来遍历 Parse 树就可以了。下面就是 Parse 类中解析 HTML 的代码：

```
static String tags[] = {"table", "tr", "td"}; // 定义一些常用的标签

public Parse (String text) throws ParseException {
    this (text, tags, 0, 0);
}

public Parse (String text, String tags[]) throws ParseException {
    this (text, tags, 0, 0);
}

public Parse (String text, String tags[], int level, int offset) throws ParseException {
    String lc = text.toLowerCase();
    int startTag = lc.indexOf("<" + tags[level]);
    int endTag = lc.indexOf(">", startTag) + 1;
    int startEnd = lc.indexOf("</>" + tags[level], endTag);
    int endEnd = lc.indexOf(">", startEnd) + 1;
    int startMore = lc.indexOf("<" + tags[level], endEnd);
    if (startTag < 0 || endTag < 0 || startEnd < 0 || endEnd < 0) {
        throw new ParseException ("Can't find tag: " + tags[level], offset);
    }

    leader = text.substring(0, startTag);
    tag = text.substring(startTag, endTag);
    body = text.substring(endTag, startEnd);
    end = text.substring(startEnd, endEnd);
    trailer = text.substring(endEnd);

    if (level+1 < tags.length) {
        parts = new Parse (body, tags, level+1, offset+endTag);
        body = null;
    }

    if (startMore >= 0) {
        more = new Parse (trailer, tags, level, offset+endEnd);
        trailer = null;
    }
}
```

关于 Parse 类最有趣的就是，一个 Parse 对象其实就代表了整个 HTML 文档。leader 成员里面保存的是特定 HTML 元素之前的所有文本，tag 成员则保存的是 HTML 元素的 tag，body 保存这个 HTML 元素中位于开头 tag 和结束 tag 之间的文本，trailer 保存了所有后续的文本。由于 Parse 对象保存了所有的 HTML 文本，所以你可以跑到顶层的 Parse 对象上，调用它的 print 方法打印出整个 HTML 文本。

```
public void print(PrintWriter out) {  
    out.print(leader);  
    out.print(tag);  
    if (parts != null) {  
        parts.print(out);  
    } else {  
        out.print(body);  
    }  
    out.print(end);  
    if (more != null) {  
        more.print(out);  
    } else {  
        out.print(trailer);  
    }  
}
```

我觉得没有比这更优雅的 Java 代码了。

很长时间以来我一直相信构造函数里面不应该做一大堆事情，而是只应该把对象设置到一个有效状态下，然后把其余真正的工作留给其他方法去做即可。一般来说，人们不会觉得对象创建会是多么昂贵的操作，所以当他们发现一个对象创建的确很昂贵时就会感到意外。然而不可否认的是，Parse 的构造函数的确优雅。它以一种优美的对称方式将一个 HTML 文本分割为几百个 Parse 对象，而一个简单的 print 方法又能将它重建起来。

如果把解析代码和表现代码放在不同的类中，这个框架也许就能够对付不同的格式了，比如 XML 或 RTF；然而将格式限制在 HTML 上感觉更好，因为它能使框架尽量小、容易理解——解析代码只有短短 25 行；如此一来 FIT 就成了一个更简单、更健壮的框架。关于设计最深刻的道理之一就是，通过将正确的东西牢牢固定下来，我们可以获得很多好处。

Parse 还有一个有趣的地方：它们并不使用集合容器来持有对邻居节点的引用，而是用 parts 和 more 这两个指向 Parse 对象的引用。这就让代码有点 Lisp 的味道了。不过如果你习惯了用“函数式”的眼光来看代码的话，其实也很简单。

下面就是一段同样风格的代码。Parse 的 last 方法返回的是一个 Parse 对象的 more 成员中的最后一个元素。

```
public Parse last() {  
    return more==null ? this : more.last();  
}
```

另外，Parse的这些成员都是公有的，这一点很有意思。框架不可能预测到用户修改它的所有可能的方式。没错，Parse也有像addToTag和addToBody这样方便的方法，但任何人只要愿意都可以直接修改那些成员变量。FIT作出这样的设计并非没有代价，代价就是FIT任何未来的版本都不能收回用户对这些成员变量的访问权。显然，并非每个框架设计者都应该或者能够这么做，但如果在你的特定情况下，你可以接受这个设计决策带来的后果的话，它也许会是一个不错的选择。

结论

我们中许多人在业界摔摔打打之后都积累了不少经验。我们不时的发现，代码并不像我们希望的那样可扩展。随着时间的推移我们逐渐学到了一些教训，我们学会了通过限制选择来保留可扩展性。如果你正在开发一个框架，而且你的框架有成千上万个客户的话，也许你最多只能做到这些了，但其实这并非惟一的选择。

FIT就展示了一个从根本完全不同的选择：FIT努力让框架尽可能地灵活和简洁，这不是通过分解出一大堆类来实现的，而是通过非常小心地把握每个类内部的分解来实现的。你把方法设为公有的，这样当用户想要“不走寻常路”的时候，用户就可以走。但你同样也作出一些困难的选择，比如像FIT这样，设定HTML为惟一的表示媒介。

如果你像上面这样设计框架的话，你的框架就会显得与众不同。一旦类对外界完全开放，以后就很难再对它作修改；但另一方面你的代码有可能会变得非常小巧且容易理解，吸引用户在它的基础上创造出新东西来。

但话又说回来了，这其实说起来容易做起来难。满世界都是那些有点过于难以理解，而且看上去设计者在上面花了很多心思的框架，设计者花的心思多到让你觉得实在没有理由重新发明车轮。这类框架一般都很庞大，但却在重要的使用领域有所缺失。

撇开框架不谈，FIT的这种开放式的风格对小项目开发而言大有可借鉴之处。如果你知道你的代码的所有用户，比如说他们可以跟你一起坐下来开个圆桌会议的话，你就可以选择将你的部分代码“降低防御”，从而使它们变得跟FIT的代码一样简洁。

并非所有的代码都是API代码——我们很容易忽视这一点，因为平常都被吓怕了（比方说听说一个团队因为引入了一个不可消除的依赖而导致另一个团队的代码无法更改），但实际上如果你们是一个团队，如果你的客户代码在必要的时候可以并且能够作出相应

改动的话，你就可以考虑采用另一种开发风格，即开放式风格。利用语言特性对框架进行保护和在设计上对框架进行保护当然是有意义的，但关键是这两种保护针对的问题是“社会性”问题（译注1），如果在你的实际情况中根本不存在这类问题，或许这种保护就不那么必要了。

我常常把FIT的代码给有经验的开发者们看，他们的反应也常常很有意思。比如他们会立即说：“嗯嗯，是不错，但是我是永远不会写那样的代码的。”于是这时我就让他们问问自己为什么不。对我来说，FIT是一个漂亮的框架，因为他让我们反思“为什么我不能这样编写代码呢？”。

译注1：也就是说，这些保护是用来防止误用的，针对的是“人”的问题，并非实质性的技术问题。

漂亮的测试

Alberto Savoia

许多程序员都有过这样的经历：看一段代码，觉得它不仅实现了功能，而且实现得很漂亮。通常，如果一段代码能优雅、简洁地完成了需要完成的功能，我们就认为这样的代码很漂亮。

那对于漂亮代码的测试，尤其是那种开发者在编写代码的同时编写的（或者应该编写的）测试，情况又是怎样的呢？在这一章，我将专注于讨论测试，因为测试本身也可以是漂亮的。更重要的是，它们能起到非常关键的作用，可以帮你写出更漂亮的代码。

正如我们将要看到的，有些东西，如果把它们组合起来会使测试很漂亮。跟代码不同的是，我无法让自己认为某个单一的测试很漂亮，至少跟我看待一个排序函数，并认为它漂亮的情况不一样。原因是测试天生就带有组合性和试探性。代码中的每一条 if 语句至少需要两个测试（一个用于条件表达式为真的情况，另一个用于为假的情况）。一条拥有多个条件的 if 语句，比如：

```
if ( a || b || c )
```

理论上需要 8 个测试——每一个对应 a、b 和 c 不同值的一个可能的组合。如果再考虑

循环中的异常，多个输入参数，对外部代码的依赖，不同的软硬件平台等，所需测试的数量和类型将大大增加。

除了最简单的情况，任何代码，不管漂亮与否，都需要一组（而不是一个）测试，这些测试中的每一个都应该专注于检查代码的一个特定的方面，就像球队一样，不同的队员有不同的职责，负责球场的不同区域。

我们已经知道应该以“组”为单位来对测试进行整体评估，现在我们需要进一步了解都有哪些特性能决定一组测试是否漂亮——“漂亮”，一个很少用来修饰“测试”的形容词。

一般来讲，测试的主要目的是逐步建立，不断加强并再次确认我们对于代码的信心：即代码正确并高效地实现了功能。因此对我来讲，最漂亮的测试是那些能将我们的信心最大化的测试，这个信心就是代码的确实现了它被要求的功能，并将一直保持这一点。由于代码不同方面的属性需要不同类型的测试来验证，所以对于“漂亮”的评判准则也不是固定的。本章考查了能使测试漂亮的三种方法。

测试因简单而漂亮

简单的几行测试代码，使我能描述并验证目标代码的基本行为。通过在每次构建时自动运行那些测试，能确保代码在不断开发的过程中始终保持所要求的行为。本章将使用JUnit测试框架来给出一些比较基本的测试例子，这些只需几分钟就能编写的测试，将在项目的整个生命周期中使我们不断受益。

测试因揭示出使代码更优雅，更可维护和更易测试的方法而漂亮

换句话讲，测试能帮我们把代码变得更漂亮。编写测试的过程不仅能帮我们找出实现中的逻辑错误，还能帮我们发现结构和设计上的问题。在这一章，通过尝试编写测试，我将演示我是怎样找到了一种能使我的代码更健壮、更有可读性、结构也更好的方法的。

测试因其深度和广度而漂亮

深入彻底、覆盖无遗的测试会大大增强开发者的信心，这种信心就是代码不仅在一些基本的、手工挑选的情形下，而且在所有的情形下都实现了所需的功能。在这一章，我将演示怎样根据测试理论中的概念来编写和运行这类测试。

由于大多数程序开发者都已经熟悉了诸如冒烟测试（smoke testing）和边界测试（boundary testing）等基本的测试技术，我将花更多的时间来讨论更有效类型的测试和那些很少被讨论和应用的测试技术。

讨厌的二分查找

为了演示多种不同的测试技术，同时又保持本章的篇幅合理，需要一个简单、易描述，并能通过几行代码就能实现的例子。同时，这个例子还必须足够生动，拥有一些有趣的挑战测试的特性。最理想的情况是这个例子要有一个悠久的总是被实现出许多 bug 的历史，从而显出对彻底测试的迫切需要。最后但并非最不重要的一点：如果这个例子本身也被认为是漂亮的代码那就再好不过了。

每当讨论漂亮的代码，就很容易让人联想起 Jon Bentley 那本经典的由 Addison-Wesley 出版的《Programming Pearls》。我就是在读那本书的时候，发现了我要找的代码例子：二分查找。

让我们快速复习一下，二分查找是一个简单而又高效的算法（但我们即将看到，要正确实现它也是有点难度的），这个算法用来确定一个预先排好顺序的数组 $x[0..n-1]$ 中是否含有某个目标元素 t 。如果数组包含 t ，程序返回它在数组中的位置，否则返回 -1。

Jon Bentley 是这样向学生们描述该算法的：

在一个包含 t 的数组内，二分查找通过对范围的跟踪来解决问题。开始时，范围就是整个数组。通过将范围中间的元素与 t 比较并丢弃一半范围，范围就被缩小。这个过程一直持续，直到在 t 被发现，或者那个能够包含 t 的范围已成为空。

他又说到：

大多数程序员认为，有了上面的描述，写出代码是很简单的事情。他们错了。能使你相信这一点的唯一方法是现在就合上书，去亲手写写代码试试看。

我赞成 Bentley 的建议。如果你从来没有写过二分查找，或者有好几年没写过了，我建议你在继续读下去之前亲手写一下；它会使你对后面的内容有更深的体会。

二分查找是一个非常好的例子，因为它是如此简单，却又如此容易被写错。在《Programming Pearls》一书中，Jon Bentley 记述了他是怎样在多年的时间里先后让上百个专业程序员实现二分查找的，而且每次都是在他给出一个算法的基本描述之后。他很慷慨，每次给他们两个小时的时间来实现它，而且允许他们使用他们自己选择的高级语言（包括伪代码）。令人惊讶的是，大约只有 10% 的专业程序员正确地实现了二分查找。

更让人惊讶的是，Donald Knuth 在他的《Sorting and Searching》（注 1）一书中指出，

注 1：见《计算机程序设计艺术，第 3 卷：排序和查找（第二版）》，Addison-Wesley，1998。

尽管第一个二分查找算法早在1946年就被发表，但第一个没有bug的二分查找算法却是在12年后才被发表出来。

然而，最让人惊讶的是，Jon Bentley正式发表的并被证明过的算法，也就是被实现或改编过成千上万次的那个，最终还是有问题的，问题发生在数组足够大，而且实现算法的语言采用固定精度算术运算的时候。

在Java语言中，这个bug导致一个`ArrayIndexOutOfBoundsException`异常被抛出，而在C语言中，你会得到一个无法预测的越界的数组下标。你可以在Joshua Bloch的blog上找到更多关于这个bug的信息：<http://googleresearch.blogspot.com/2006/06/extr-extra-read-all-about-it-nearly.html>。

以下就是带有这个著名的bug的Java实现：

```
public static int buggyBinarySearch(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

Bug位于这一行：

```
int mid = (low + high) / 2;
```

如果`low`和`high`的和大于`Integer.MAX_VALUE`（在Java中是 $2^{31}-1$ ），计算就会发生溢出，使它成为一个负数，然后被2除时结果当然仍是负数。

推荐的解决方案是修改计算中间值的方法来防止整数溢出。方法之一是用减法——而不是加法——来实现：

```
int mid = low + ((high - low) / 2);
```

或者，如果你想炫耀一下自己掌握的位移运算的知识，那个blog（还有Sun微系统公司

的官方 bug report, 注 2) 建议使用无符号位移运算, 这种方法或许更快, 但对大多数 Java 程序员 (包括我) 来说, 可能也比较晦涩。

```
int mid = (low + high) >>> 1;
```

想一下, 二分查找算法的思想是多么的简单, 而这么多年又有多少人的多少智力花在它上面, 这就充分说明了即使是最简单的代码也需要测试, 而且需要很多。Joshua Bloch 在它的 blog 中对这个 bug 作了非常漂亮的陈述:

这个 bug 给我上的最重要的一课就是要懂得谦逊: 哪怕是最简单的一段代码, 要写正确也并非易事, 更别提我们现实世界中的系统: 它们跑在大段大段的复杂代码上。

下面是我要测试的二分查找的实现。理论上讲, 对于中间值的计算方法的修正, 应该是解决了这段令人讨厌的代码的最后一个 bug, 一个在好几十年的时间里, 连一些最好的程序员都抓不到的 bug。

```
public static int binarySearch(int[] a, int target) {  
    int low = 0;  
    int high = a.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) >>> 1;  
        int midVal = a[mid];  
  
        if (midVal < target)  
            low = mid + 1;  
        else if (midVal > target)  
            high = mid - 1;  
        else  
            return mid;  
    }  
    return -1;  
}
```

这个版本的 `binarySearch` 看上去是正确的, 但它仍可能有问题。或许不是 bug, 但至少是可以而且应该被修改的地方。这些修改可以使代码不仅更加健壮, 而且可读性, 可维护性和可测试性都比原来更好。让我们看看是否可以通过测试来发现一些有趣的和意想不到的改善它的机会。

JUnit 简介

谈到漂亮测试, 就很容易想到 JUnit 测试框架。因为我使用 Java, 通过使用 JUnit 来构建

注 2: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5045582。

我的漂亮的测试是一个很自然的决定。但在做之前，考虑到你对JUnit可能尚未熟悉，让我先对它做一个简单介绍吧。

JUnit 是 Kent Beck 和 Erich Gamma 设计的，他们创造 JUnit 来帮助 Java 开发者编写和运行自动的和自检验的测试。它有一个很简单，却又很宏伟的目标：就是使得程序开发者更容易去做他们本来就应该做的事情：测试自己的代码。

遗憾的是，我们还要走很长的路才能到达那种大多数程序员都像是被“测试病毒”所感染的阶段（在那种情况下；程序员们试着自己编写测试，并决定把它看作开发中的一个常规的重要组成部分）。然而，自从被引入开发领域，任何其他的东西都没能像 JUnit 那样使这么多的程序员开始编写测试。不过这也得感谢极限编程和其他敏捷开发方法的巨大帮助，在这些方法中，开发者参加程序测试是必须的（注3）。Martin Fowler 对 JUnit 的影响作了这样的概括：“少量代码对大量的代码起了如此重要的作用，这在软件开发领域中是前所未有的事。”

JUnit 被特地设计得很简单，易学易用。这是 JUnit 的一个重要的设计准则。Kent Beck 和 Erich Gamma 花费了大量心思来确保 JUnit 的易学易用，于是程序员们才会真正使用它。它们自己是这样说的：

我们的第一目标就是要写出一个框架，使我们可以对程序员们真正在其中编写测试抱有希望。这个框架必须使用人们熟悉的工具，这样大家就不用学习很多新东西；必须保证编写一个新的测试所需的工作量降至最低；还必须能够消除重复劳动。
(注 4)

JUnit 的官方入门文档（the JUnit Cookbook）的长度还不到两页纸：<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>。

以下是从 cookbook（来自 JUnit 的 4.x 版本）中抽取出来的最重要的一段

当你需要测试一样东西时，你只要做：

1. 为一个方法加上 @org.junit.Test 标注（annotate）；

注 3：能够彰显JUnit的成功及影响力的一个事实是，如今针对大多数现代编程语言的测试框架都出现了，它们都是从JUnit那里得到的灵感，JUnit的各类扩展也出现了。

注 4：“JUnit: A Cook’s Tour”，Kent Beck and Erich Gamma：<http://junit.sourceforge.net/doc/cookstour/cookstour.htm>。

2. 当你需要检查一个值，把org.junit.Assert*输入进来，调用assertTrue()，并传递一个 Boolean 对象，当测试成功时它为 true。

比如，为了测试同一币种的两个 Money 对象相加时，结果对象中的值恰好等于那两个 Money 对象中的值直接相加的结果，你可以这样做：

```
@Test  
public void simpleAdd() {  
    Money m12CHF= new Money(12, "CHF");  
    Money m14CHF= new Money(14, "CHF");  
    Money expected= new Money(26, "CHF");  
    Money result= m12CHF.add(m14CHF);  
    assertTrue(expected.equals(result));  
}
```

只要你稍微熟悉一点 Java 语言，那两条操作指导和这个简单的例子就足以使你上手。它们也足以使你理解我将要写的测试。简单得让人觉得漂亮，是不是？好，我们继续。

将二分查找进行到底

知道了它的历史，我不想被二分查找表面的简单或看似明显的修改所欺骗，尤其是我从来没有在其他代码中用过无符号移位操作符（即>>>）。我将测试这个二分查找的修正版本，就如同我以前从来没有听说过它，也没有实现过它。我不想相信任何人的说辞，测试和证明，说它这一次确实正确。我要通过自己的测试来确信它按照它所应该的方式工作，让它成为一件确凿无疑的事情。

这是我最初的测试策略（或者说测试组）。

- 从冒烟测试开始；
- 增加边界值测试；
- 继续其他各种彻底的、全覆盖类型的测试；
- 最后，添加一些性能测试。

测试一般不会是个线程的过程。我将和你一起再次整理一遍我编写这些测试时的思路，而不是直接把最终的测试集合给你看。

冒烟测试

让我们从冒烟测试开始。这些测试用来确保代码在最基本的使用方式下能够正常工作。它们是第一条防线，是第一组该被编写的测试，因为如果一个实现连冒烟测试都通不过，

那更进一步的测试就是在浪费时间。我经常在编写代码前就编写冒烟测试，这叫做测试驱动开发（test-driven development，或 TDD）。

以下是我写的二分查找的冒烟测试：

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BinarySearchSmokeTest {

    @Test
    public void smokeTestsForBinarySearch() {

        int[] arrayWith42 = new int[] { 1, 4, 42, 55, 67, 87, 100, 245 };
        assertEquals(2, Util.binarySearch(arrayWith42, 42));
        assertEquals(-1, Util.binarySearch(arrayWith42, 43));

    }

}
```

你能看出，这个测试确实非常基本。它本身无法就代码的正确性给人带来很大的信心，但它仍然不失漂亮，因为它向着更深入的测试迈出了迅速、高效的第一步。

由于这个冒烟测试执行起来极快（在我的系统中不到百分之一秒），你可能会问我为什么不包含一些测试。问题的答案是冒烟测试之所以漂亮，部分原因就是大部分的开发结束后，他们仍然能起作用。为了重新确认我对代码的信心——就叫它“信用维护”吧——我喜欢把所有的冒烟测试组成一个测试组（suite），并在每一次构建时都运行它们（这种构建每天会有十来次），而且我希望这个冒烟测试组能够运行得很快——最好在一两分钟内。如果你有几千个类和几千个冒烟测试，保持每一个测试尽可能小就是很重要的事了。

边界测试

边界测试，顾名思义，就是用来探测和验证代码在处理极端的或偏门的情况时会发生什么。在二分查找中，两个参数分别是数组和要查找的目标值。让我们为这两个参数分别设想一些连界情况（注 5）。

注 5：按照二分查找的规格说明，在调用查找函数之前，数组必须是有序的，如果它不是，结果就是未定义的。我们还假定如果数组参数为 null，代码将抛出一个 NullPointerException。由于大多数读者对边界测试的基本技术都比较熟悉了，我将跳过一些显而易见的测试。

第一组跑进我大脑的有趣的边界情况跟被查找的数组的长度有关。我从以下这个基本的边界测试开始：

```
int[] testArray;

@Test
public void searchEmptyArray() {
    testArray = new int[] {};
    assertEquals(-1, Util.binarySearch(testArray, 42));
}

@Test
public void searchArrayOfSizeOne() {
    testArray = new int[] { 42 };
    assertEquals(0, Util.binarySearch(testArray, 42));
    assertEquals(-1, Util.binarySearch(testArray, 43));
}
```

很显然，一个空数组就是一个很好的边界例子，长度为1的数组也是，因为它是最小的非空数组。这两个测试都很漂亮，它们增加了我的信心：在数组长度边界的下端，程序是正确的。

但我还想用一个非常大的数组来测试二分查找，这就是最有趣的地方了（尤其是我们已经知道了那个 bug 仅发生在长度超过 10 亿的数组中）。

我的第一个想法是创建一个足够大的数组，大到可以确保整数溢出的 bug 已被修正了，但我马上意识到一个“可测试性”的问题：我的便携电脑没有足够的资源在内存中创建那么大的一个数组。但我知道，确实有的系统拥用好几个 GB 的内存，而且在内存中放置很大的数组。不管通过哪种方法，我要确保的是，中间值在那种情形中不会溢出。

我该怎么做？

我知道，等我完成我设想的其他一些测试之后，我就拥有了足够的测试来给予自己信心：只要中间值被正确的计算而不是溢出为一个负数，基本算法和实现就是对的。以下概括了我的推理，它们得出了一种针对巨大数组的可行的测试方案。

1. 我无法使用一个足够大的数组来直接测试binarySearch，从而验证中间值计算中的溢出 bug 不会发生；
2. 但是，我可以编写足够的测试，让自己确信binarySearch的实现在小的数组上执行正确；
3. 当用到非常大的数值时，我也能够单独测试计算中间值的方法，这跟数组无关；
4. 于是，假如测试足以使我确信以下两件事情：

- 只要计算中间值的过程没有问题，我的binarySearch的实现就是正确的；并且：
- 计算中间值的过程没有问题。

那么，我可以确信当binarySearch应用于非常大的数组上时仍然是正确的。

因此，这个并不十分明显，但却很漂亮的测试策略就把讨厌的，易于溢出的计算隔离出来单独测试。

一种方法是编写一个新的函数：

```
static int calculateMidpoint(int low, int high) {  
    return (low + high) >>> 1;  
}
```

然后，把代码中的这一行：

```
int mid = (low + high) >>> 1;
```

变成：

```
int mid = calculateMidpoint(low, high);
```

再然后，就不断测试calculateMidpoint方法来确保他永远正确执行。

我已经听到有人在大叫：“为什么在一个为最佳性能而设计的算法中增加一个方法的开销？”别着急，我相信对代码的这一改动不仅仅是可以接受，而且恰恰是正确的，以下就是原因：

1. 如今，我可以信任编译器优化的能力，它会为我把那个方法内联（inline），因此，这里并没有性能损失；
2. 改动提高了代码的可读性。我问过好几个Java程序员，他们中的大多数都不熟悉无符号位移操作，或者对它究竟如何工作没有十足的把握。对这些程序员来说，“calculateMidpoint(low, high)”比“(low + high) >>> 1”看上去更易理解；
3. 改动还提高了代码的可测试性。

这真是一个好例子，你看到了如何通过编写测试来改善代码的设计并使代码更易理解。换句话说，测试可以使你的代码更漂亮。

下面是针对新的calculateMidpoint方法的边界测试的例子：

```
@Test
```

```

public void calculateMidpointWithBoundaryValues() {
    assertEquals(0, calculateMidpoint(0, 1));
    assertEquals(1, calculateMidpoint(0, 2));
    assertEquals(1200000000, calculateMidpoint(1100000000, 1300000000));
    assertEquals(Integer.MAX_VALUE - 2,
        calculateMidpoint(Integer.MAX_VALUE - 2, Integer.MAX_VALUE - 1));
    assertEquals(Integer.MAX_VALUE - 1,
        calculateMidpoint(Integer.MAX_VALUE - 1, Integer.MAX_VALUE));
}

```

我执行了这组测试，它通过了。很好。现在我可以确信，在我的二分查找所需处理的数组长度的范围内，这个用不熟悉的操作符来计算中间值的方法，正确实现了它的功能。

另一种边界情况是关于目标元素的位置的。我能想出3个明显的边界位置：数组中的第一项，数组中的最后一项和数组中不偏不倚正中间的那一项。为此，我编写一个简单的测试来检查这些边界的情况：

```

@Test
public void testBoundaryCasesForItemLocation() {
    testArray = new int[] { -324, -3, -1, 0, 42, 99, 101 };
    assertEquals(0, Util.binarySearch(testArray, -324)); // first position
    assertEquals(3, Util.binarySearch(testArray, 0)); // middle position
    assertEquals(6, Util.binarySearch(testArray, 101)); // last position
}

```

注意在这个测试中我使用了0和一些负数，0和负数不仅包含于数组中，也包含于被查找的目标元素中。在读我以前写的测试时，我发现我曾经只用正数。二分查找算法的描述中并没说所有的数都是正数，因此我应该在我的测试中引入负数和0。于是，我得到了这么一条关于测试的至理明言：

想出更多测试用例的最好方法就是开始编写测试用例。

当我已开始考虑正数、负数和0，我想到了另一件不错的事情：一些使用最大和最小整数值的测试。

```

public void testForMinAndMaxInteger() {
    testArray = new int[] {
        Integer.MIN_VALUE, -324, -3, -1, 0, 42, 99, 101, Integer.MAX_VALUE
    };
    assertEquals(0, Util.binarySearch(testArray, Integer.MIN_VALUE));
    assertEquals(8, Util.binarySearch(testArray, Integer.MAX_VALUE));
}

```

到这里，我所想到的所有的边界情况都通过测试了，于是我也开始有非常自信的感觉了。但我马上想起Jon Bentley班上那90%的专业程序员，他们实现了二分查找，并认为自己写对了，但事实上却并没有写对——我的信心顿时又减掉了一些。我是否对输入做

了无根据的假设？我一直到写最后这个测试用例时才开始考虑负数和0。我是否还做过其他无根据的假设？因为我是自己编写的测试，或许我无意间就选择了可以成功的情况，而遗漏了那些可能失败的。

对于程序员测试他们自己写的代码，这是一个普遍存在的问题。如果他们在编写代码时没有考虑到一些应用场景，那么当他们在测试代码时也无法测试这些应用场景。真正漂亮的测试需要开发者付出额外的努力，跳出旧有的思维模式，探求非常规的应用场景，找寻薄弱的环节，并试着“创新”。

那么，我还有哪些没想到呢？感觉上我的冒烟测试和边界测试并不充分。我的测试集合，再加上一些归纳（注6），足够使我宣称我的代码在各种情况下都能正常工作吗？我的脑海中回响起 Joshua Bloch 的话：“……即使是最简单的代码，要写正确也不容易。”

什么样的测试能使我足够确信我的代码在面对各种不同的输入时都能正常工作，而不仅仅是针对我所写的几种情况。

随机测试

到现在为止我写的都是那种传统的、试了就正确的测试。我用了几个具体的例子来测试查找算法的代码，看它在那些情况中的行为是否跟我预期的一样正确。那些测试全部通过，因此我对我的代码也有了一定的自信。但同时我也意识到我的测试过于具体，对于所有可能的输入，只能覆盖它很小的一个子集。而我想要的、能够使我知道我的代码被全面覆盖，从而夜里可以安然入眠的，是一种对输入情况覆盖更广的测试方法。为达到这个目标，我需要两样东西：

1. 一种能产生各种不同特征的，大数据量的输入集合的方法；
2. 一组能通用于各种的输入集合的断言（assertion）。

让我们来解决第一个需求。

这里我所需要的是一种能产生各种不同特征，不同长度的整数数组的方法。惟一需要我做的是保证数组有序，因为这是一个前条件（precondition）。除了这个，其他都无所谓。以下是最初的数组产生器实现（注7）。

注6： 我所说的“归纳”（induction）是指从特殊事实或事例推出普遍原理的过程。

注7： 我之所以说“最初的”，是因为我很快意识到除了正数外，我也需要在数组中包含负数，并因此修改了产生器的实现。

```
public int[] generateRandomSortedArray(int maxArraySize, int maxValue) {  
    int arraySize = 1 + rand.nextInt(maxArraySize);  
    int[] randomArray = new int[arraySize];  
    for (int i = 0; i < arraySize; i++) {  
        randomArray[i] = rand.nextInt(maxValue);  
    }  
    Arrays.sort(randomArray);  
    return randomArray;  
}
```

为了实现我的数组产生器，我使用了 `java.util` 包中的随机数产生器和 `Arrays` 类中的一些实用方法。到现在我们一直在解决 Joshua Bloch 在他的 blog 中提到的二分查找的 bug，而这里所用到的 `Arrays.sort` 的实现中恰恰存在完全相同的 bug，但在我所用的 Java 版本中它已经被修正了。我已经在其他测试中包含了对空数组的处理，且结果也令人满意，因此这里我采用的数组的最小长度为 1。这个产生器被设计成了参数化的，因为随着测试的进行，我可能需要创建不同的测试集合：一些是包含大数字的小数组，另一些是包含小数字的大数组，等等。

现在我需要给出一些一般的陈述，来描述那些可被表达为“断言”(assertion)的二分查找的行为。所谓“一般”，是指对于任何输入数组和要查找的目标值，这些陈述必须永远为真。我的同事 Marat Boshernitsan 和 David Saff 把这称作推理论。这里的思路是：我们有一个关于代码应如何工作的推理论，而我们对这个推理论的测试越充分，我们对推理论的正确性的信心就越大。在下面的例子中，我将应用一个 Saff 和 Boshernitsan 推理的一个简化版本。

让我们提出一些有关二分查找的推理论，以下就是：

对于 `testArray` 和 `target` 的所有实例——这里 `testArray` 是一个有序的整数数组，而且不为空，`target` 是一个整数——`binarySearch` 必须保证下面的两条永远为真：

推理论 1：(注 8) 如果 `binarySearch(testArray, target)` 返回 -1，那么 `testArray` 不包含元素 `target`；

推理论 2：如果 `binarySearch(testArray, target)` 返回 `n`, `n` 大于或等于 0，那么 `testArray` 包含元素 `target`，且在数组中的位置是 `n`。

注 8：在实际阐述推理论时，我会使用，而且会提倡使用叙述性的名字，比如 `binarySearchReturnsMinusOneImpliesArrayDoesNotContainElement` (“二分查找返回 -1 说明数组不包含那个元素”)，但在这一章，我发现如果我使用推理论 1、推理论 2 等等，推理论将更容易理解。

下面是我测试这条推理论的代码：

```
public class BinarySearchTestTheories {  
    Random rand;  
  
    @Before  
    public void initialize() {  
        rand = new Random();  
    }  
  
    @Test  
    public void testTheories() {  
        int maxArraySize = 1000;  
        int maxValue = 1000;  
        int experiments = 1000;  
        int[] testArray;  
        int target;  
        int returnValue;  
  
        while (experiments-- > 0) {  
            testArray = generateRandomSortedArray(maxArraySize, maxValue);  
            if (rand.nextBoolean()) {  
                target = testArray[rand.nextInt(testArray.length)];  
            } else {  
                target = rand.nextInt();  
            }  
            returnValue = Util.binarySearch(testArray, target);  
            assertTheory1(testArray, target, returnValue);  
            assertTheory2(testArray, target, returnValue);  
        }  
    }  
  
    public void assertTheory1(int[] testArray, int target, int returnValue) {  
        if (returnValue == -1)  
            assertFalse(arrayContainsTarget(testArray, target));  
    }  
  
    public void assertTheory2(int[] testArray, int target, int returnValue) {  
        if (returnValue >= 0)  
            assertEquals(target, testArray[returnValue]);  
    }  
  
    public boolean arrayContainsTarget(int[] testArray, int target) {  
        for (int i = 0; i < testArray.length; i++)  
            if (testArray[i] == target)  
                return true;  
        return false;  
    }  
}
```

在主测试方法testTheories中，我首先决定需要运行多少次实验才能确认推理论的有效性，然后把那个数字作为我的循环计数器。在循环内，我刚刚实现的随机数组产生器帮

我产生出有序数组。成功的和不成功的查找我都要测试，因此我再次使用Java的随机数产生器来“掷硬币”（通过`rand.nextBoolean()`）。是选择一个确知存在于数组中的目标数值，还是选择一个不大可能存在于数组中的数值，我根据这种虚拟的硬币投掷做出决定。最后，我调用`binarySearch`，保存返回值，并调用针对现有的推理所设计的验证方法。

注意，为了实现推理的测试，需要写一个测试辅助函数，`arrayContainsTarget`，这个函数给了我另一个检查`testArray`是否包含目标数值的方法。对于这类测试来说，这是一个很常见的做法。尽管这个辅助方法所提供的功能与`binarySearch`相似，但它毕竟是一个简单得多（当然，也慢得多）的查找算法。我对这个辅助函数的正确性充满信心，因此我可以使用它来测试一个我不那么有信心的实现。

我的测试从运行1000组实验开始，这些实现运行在最大长度为1000的数组上。测试花了不到1秒钟的时间，而且所有的测试都通过了。很好，现在到了多试探一些数据的时候了（记住测试就是一种充满试探性的活动）。

我修改了实现，并把`maxArraySize`的值设成10,000，接着又设成100,000。现在测试花费的时间接近一分钟，真该给我的CPU打个高分。我感觉我给自己的代码做了一个相当好的测验。

我的信心建立起来了，但我的信仰之一是：如果你的测试全部通过，那常常说明你的测试不够好。有这样一个测试框架，我还应该再测一下代码的哪些方面呢？

我想了一下，发现我的两个推理在形式上都是：

“如果跟`binarySearch`的返回值有关的某事为真，那么跟`testArray`和目标数值有关的另一件事也必须为真。”

换言之，我的逻辑是这种形式的： p 隐含 q （或者使用逻辑学上的符号： $p \rightarrow q$ ），这说明，对于我应该测试的东西，我只测试了一半。我还应该拥有 $q \rightarrow p$ （注9）形式的测试。

注9：当然， p ， q 都可以求反，或者两者同时求反（比如， $\neg p \rightarrow \neg q$ 或 $p \rightarrow \neg q$ ）。我随意选择了 p 和 q 分别代表与返回值和数组参数有关的谓词。这里关键是要认识到：当你编码的时候，你通常用 $p \rightarrow q$ 的方式思考（如果 p 为真，那么 q 必须发生——即所谓的“开心路线”：正常的，最一般的使用代码的方式）。然而当你测试时，你必须使自己学会逆向的思考方式（ $q \rightarrow ?$ ，或者假如 q 为真，关于 p ，什么必须为真？）和否定的思考方式（假如 p 不为真，也就是 $\neg p$ ，那关于 q ，什么必须为真？）。

如果跟`testArray`和目标值有关的某事为真，那么跟返回值有关的另一件事必须为真。

这有点难懂，但却很重要，所以让我用一些具体的例子来详细解释一下。针对推理 1 的测试证实了当返回值是 -1 时，目标元素不在数组中。但它并不能证实当目标元素不在数组中时，返回值一定是 -1。换言之，如果我只测试这一个推理，当目标元素不在数组中时，某个有时返回 -1，但并不总是返回 -1 的实现，照样能通过所有的测试。针对推理 2，也有类似的问题存在。

我可以使用变异测试（mutation test）来演示这个问题，变异测试是 Jeff Offut 发明的一种用来对测试代码进行测试的技术。基本思想就是修改被测的代码，使它带有一些已知的 bug。如果你的测试无视代码中 bug 而依然通过，那就说明这些测试不够全面，无法满足需要。

让我来给 `binarySearch` 做些大的修改，这些修改还带有任意性。我将试着这样做：如果目标值大于 424242 而且并不包含在数组中，我并不返回 -1，而是返回 -42。这对软件来说，是多么大的破坏啊。看下面的代码的结尾部分：

```
public static int binarySearch(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];
        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return mid;
    }
    if (target <= 424242)
        return -1;
    else
        return -42;
}
```

我想，你会同意这个比较大的修改了：如果目标值大于 424242 而且不包含在数组中，代码返回了一个出人意料的未指定的值。然而，我们所写的所有的测试都顺利通过了测试。

毫无疑问，我们至少需要再添加两三个推理来使测试更严格，并能捕捉到这种类型的修改。

推理 3：如果 `testArray` 不包含 `target`，它就必须返回 -1。

推理4：如果 *testArray* 在位置 *n* 上包含 *target*, 那么 *binarySearch(testArray, target)* 必须返回 *n*。

对这些推理的测试如下：

```
public void assertTheory3(int[] testArray, int target, int returnValue) {  
    if (!arrayContainsTarget(testArray, target))  
        assertEquals(-1, returnValue);  
}  
  
public void assertTheory4(int[] testArray, int target, int returnValue) {  
    assertEquals(getTargetPosition(testArray, target), returnValue);  
}  
  
public int getTargetPosition(int[] testArray, int target) {  
    for (int i = 0; i < testArray.length; i++)  
        if (testArray[i] == target)  
            return i;  
    return -1;  
}
```

注意我必须再编写一个辅助方法：*getTargetPosition*。这个方法拥有跟 *binarySearch* 完全相同的行为（但我确信它可以正确工作，缺点是它需要 *n* 次而不是 $\log_2 n$ 次的比较）。由于 *getTargetPosition* 跟 *arrayContainsTarget* 很相似，代码重复是不好的，所以我把后者重新编写如下：

```
public boolean arrayContainsTarget(int[] testArray, int target) {  
    return getTargetPosition(testArray, target) >= 0;  
}
```

我使用我的随机数组产生器再次运行了这些测试，这次返回 -42 的修改被马上测了出来。很好，这增加了我对代码的信心。我从代码中除去了这个故意设置的 bug 并再次运行测试。我期望它们会通过，但它们没有。一些针对推理 4 的测试没有通过。JUnit 失败了，并给出了如下形式的信息：

```
expected:<n> but was:<n + 1>
```

推理 4 提到：

如果 *testArray* 在位置 *n* 上包含 *target*, 那么 *binarySearch(testArray, target)* 必须返回 *n*。

然而，在有些情况下，查找函数返回了 *n* 的下一个位置。这怎么可能？

我需要更多一些的数据。JUnit 的断言 (assertion) 可以接受一个 String 类型的消息作为第一个参数，因此我修改了推理 4 的 *assertEquals*，让它包含一些能在失败时给我更多信息的文本。

```
public void assertTheory4(int[] testArray, int target, int returnValue) {
    String testDataInfo = "Theory 4 - Array=" +
        printArray(testArray)
        + " target="
        + target;
    assertEquals(testDataInfo, getTargetPosition(testArray, target),
    returnValue);
}
```

现在，只要推理4没有通过，JUnit就会把数组的内容连同目标值一起显示给我。我重新运行了测试（这次`maxArraySize`和`maxValue`的值都被设小，从而使输出更易读），得到的结果如下：

```
java.lang.AssertionError: Theory 4 - Array=[2, 11, 36, 66, 104, 108, 108,
108, 122, 155, 159, 161, 191] target=108 expected:<5> but was:<6>
```

我知道问题发生在哪了。推理4没有将重复值纳入考量，我也没考虑它。数组中有3个108。我想我需要找到关于处理重复值的说明，然后要么修改我的代码，要么修改我的推理，接着再测试。不过我将把这个作为练习留给读者（我总是想那样说！），因为我说得已经不少了，而且在我们合上这一章之前我还想再讲一点与性能测试有关的东西。

性能测试

我们已经运行过的、基于那些推理的测试给我们代码包上了一层很紧的保护网。一个有bug的实现还能通过所有的测试吗？我想这样的情况很难再存在了。但是，我们还忽略了一点。我们所拥有的测试对于查找来说都是很好的测试，但我们正在测的不是别的，而是二分查找。我们需要一组测试来验证“二分”特性。我们需要看到我们的实现所执行的比较次数是否达了最大 $\log_2 n$ 的预期。如何才能做到这一点呢？

我首先想到的是利用系统时钟，但我很快打消了这个想法，原因是针对这个问题来说，我所能用的时钟没有足够的精确度（二分查找的速度实在太快了），而我又不能真正控制运行环境。因此，我使用了另一个测试技巧：我创建了一个新的、叫做`binarySearchComparisonsCount`的`binarySearch`实现。这个版本的代码在程序逻辑上跟原来是一样的，但它维护并返回了一个比较计数，而不是返回-1或者目标值的位置（注10）。代码如下所示：

注10：相对于直接修改`binarySearch`来返回比较计数，(David Saff提出的)一个更好、更清晰、也更面向对象的设计是创建一个`CountingComparator`类，这个类实现Java所泛化出来的`Comarator`接口，然后修改`binarySearch`，接收这个类的一个实例作为第3个参数。这种方法也泛化了`binarySearch`，使它可用于整型以外的其他类型——又是一个关于测试怎样使设计更优良、代码更漂亮的好例子。

```

public static int binarySearchComparisonCount(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    int comparisonCount = 0;

    while (low <= high) {
        comparisonCount++;

        int mid = (low + high) >>> 1;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return comparisonCount;
    }
    return comparisonCount;
}

```

基于这个代码，我给出了另外一个推理：

推理 5：如果 *testArray* 的长度为 n ，那么 *binarySearchComparisonCount* (*testArray*, *target*) 必须返回一个小于或等于 $1 + \log_2 n$ 的数。

以下是针对这个推理的测试代码：

```

public void assertTheory5(int[] testArray, int target) {
    int number_of_comparisons =
        Util.binarySearchComparisonCount(testArray, target);
    assertTrue(number_of_comparisons <= 1 + log2(testArray.length));
}

```

我把最后的这个推理添加到 *testTheories* 方法的推理列表中，结果就像这样：

```

...
while (experiments-- > 0) {
    testArray = generateRandomSortedArray();
    if (rand.nextInt() % 2 == 0) {
        target = testArray[rand.nextInt(testArray.length)];
    } else {
        target = rand.nextInt();
    }
    returnValue = Util.binarySearch(testArray, target);
    assertTheory1(testArray, target, returnValue);
    assertTheory2(testArray, target, returnValue);
    assertTheory3(testArray, target, returnValue);
    assertTheory4(testArray, target, returnValue);
    assertTheory5(testArray, target);
}

```

}

我把 `maxArraySize` 分别设成不同的值来跑了几个测试，发现推论 5 好像不成立。

因为快到中午，我把实验数字设成 1 000 000，然后就去吃午饭了，我的电脑则继续运行，把每个理论测试一百万次。

当我回来的时候，我看到所有的测试都通过了。或许还有其他几件事情需要我去测试，但针对这个 `binarySearch` 的实现，我的信心已大大增加了。由于不同开发者拥有不同的背景、风格和经验水平，你也可以关注于代码的不同方面。比如，在一个已熟练掌握无符号位移操作的开发者看来，这个测试的必要性并不一定有我所认为的那么大。

在这一节，我想让你们感受一下性能测试，同时告诉你们如何通过将代码和测试理论相结合，来获得对代码性能的洞察力以增强信心。我强烈建议你学习一下第 3 章，在那里 Jon Bentley 详细讨论了这个话题，并给出了漂亮的处理方法。

结论

在这一章，我们看到即使是最优秀的程序员和最漂亮的代码也仍然能从测试中获益。我们也看到了编写测试代码可以跟编写目标代码本身一样需要创造力，也一样具有挑战性。而且，我也希望我向你们展示了测试自身，至少在 3 个方面，也可被认为是漂亮的。

有些测试因为简单和高效而漂亮。只需几行，每次随构建自动运行的 JUnit 代码，你就能描述你想要的代码的行为和边界情况，并且能确保在代码不断开发的过程中，这些行为和边界情况一直保持。

另外一些测试，它们的漂亮是因为它们在被编写的过程中，能够通过一些细小却又很重要的方式，帮助你改善被测代码的质量。它们可能并不能发现常规的 bug 或缺陷，但它们能让代码中的问题浮出水面，这些问题可能是设计、可测试性和可维护性方面的问题；它们能帮助你把代码变得更漂亮。

最后，有些测试的漂亮是因为它们的覆盖面和彻底性。他们帮你获得信心：即代码在功能和性能上都达到了要求和期望，不仅仅是针对几个手工挑选的情况，而是针对大范围的输入数据和条件。

希望写出漂亮代码的开发者可以向艺术家们学习一些东西。画家常常放下手中的画笔，然后远离画布一段距离，围着它转一转，翘起脑袋，斜着看看，再从不同的角度看，看在不同的光线下看看。在寻求美的过程中，他们需要设计这样一些视角并使他们融为一体。

体。如果你的画布是个集成开发环境（IDE）而你的媒介就是代码，想一想，你如何做到离开画布一段距离，用挑剔的眼光从不同的视角来审视你的作品？——这将使你成为一个更优秀的程序员，并帮你写出漂亮的代码。

图像处理中的 即时代码生成

Charles Petzold

Steven Levy 曾写过一本经典书：《Hackers: Heroes of the Computer Revolution》(Doubleday)，在其中收录的众多奇闻轶事中我印象最深的是 Bill Gosper 的一句话：“数据只不过是一种笨程序”。显然，这句话的一个推论是，代码只不过是一种“聪明”的数据，聪明也就是说它们可以驱使处理器去执行某些有用或有趣的动作。

大多数传统的编程教育都弱化了数据和代码之间的潜在联系。数据和代码通常是被区别对待的；即便是在面向对象编程中，数据和代码扮演的角色也完全不同。如有胆敢越雷池者，比如把数据当作代码来执行的，统统格杀勿论。

只有在极少的一些场合下，数据和代码之间的篱笆才允许被暂时打开。比如编译器的任务就是读进源代码，输出对应的机器代码。不过话说回来，编译器其实并没有触犯代码和数据之间的禁令；虽说其输入（源代码）和输出（机器代码）对于人类程序员来说都是代码，但对于编译器来说却都只是数据而已。与此类似的还有反汇编工具和模拟器，在它们眼里，代码也只是一种数据。

就算感情上不愿承认，理性上我们也不得不承认，数据和代码本质上只不过是一堆堆的字节。而且，一个特定的字节有且仅有 256 种变化，就算你找遍整个宇宙也找不出第 267

种来。然而，真正关键的并非字节本身，而是字节按照一定顺序构成的序列；是顺序，赋予了它们意义和意图。

编译器负责生成代码固然天经地义，然而有时候其他程序也有可能会需要运行时生成代码。不过由于“即时（on-the-fly）代码生成”并不是件容易的事情，所以通常只被用在非常特殊的场合。

本章我们借助于一个实例来介绍这种即时代码生成技术，这个例子也体现了使用该技术最常见的动机。在这个例子中，有一个对执行效率要求极高的函数，该函数必须执行许多重复操作。在这些重复操作的执行过程中使用了一些泛化参数，如果我们能够将其中的泛化参数替换为具体值的话，函数的执行效率就会提高许多。问题是，在编写该函数的时候我们并不能替换这些参数，因为编写时根本不知道参数的具体值，要等到运行时才能知道；而且每次调用该函数的时候参数值很可能并不相同。那怎么办呢？答案是让该函数自己充当编译器的角色。也就是说，该函数可以在运行时查看各参数值，然后为它的函数体生成更加高效的、直接针对特定参数值执行的代码，最后跳到这些代码去运行。

我第一次接触该技术是在一次编写汇编代码时。当时我写的那个函数里面执行了许多重复操作。在其中的某个效率关键的代码段需要执行AND或OR操作，而具体是执行AND还是OR则是由运行某个操作得出的值决定的。不过，一旦确定了是执行AND还是OR，在那段代码内便会一直执行那个操作（AND或OR）了。我面临的问题是，对那个运行期值的检查位于循环体中，所以自身便耗费了太多时间。我想过将整个函数分成两个版本，一个版本对应AND操作，另一个对应OR操作。但后来我又意识到一个更巧妙的技巧：我可以先确定到底是否会执行 AND 还是 OR，然后将 AND 或 OR 的机器指令直接插入到代码执行流中。

实际上，这种即时代码生成技术在第一版的 Windows（1.0 版）里面派上了大用场。第一版 Windows 诞生于 1985 年，后来在个人计算机市场上取得了很大的成功。而对一个程序员来说，第一版 Windows 提供了大约 200 个函数用于创建图形用户界面，以及在屏幕和打印机上输出向量和光栅图样，这些函数达到了相当程度的设备无关性。

Windows 1.0 里面有一个图形函数叫做 *BitBlt*，意即“bit block transfer”（位块传输），这个名字源于施乐公司开创性的个人 PC “Xerox Alto” 所支持的一个指令。*BitBlt* 的最基本应用是在屏幕或打印机上呈现位图（bitmap），但实际上 Windows 内部也使用它来显示许多用户界面对象。更通俗地说，*BitBlt* 将一块长方形的像素块从一个地方传输到另一个地方。与 *BitBlt* 类似的函数还有 *StretchBlt*，*StretchBlt* 的区别在于它可以在传输的过程中对源块进行适当的压缩或拉伸，使其适应目标块的大小。

如果 BitBlt 源是一张位图，且目的地是视频显示器，则 BitBlt 会将位图的各像素拷贝到显示器上，即把位图显示到屏幕上。如果源是显示器，目的地是位图的话，BitBlt 便会将屏幕上的像素拷贝至位图，于是位图内包含的就是一张截屏。

然而，如果你自己也编写了一个类似 BitBlt 的函数的话，你的函数做的事情可能就不止这些了；你可能会给它加上一些其他功能。比如可以给函数加上一个选项，选择传输像素的过程中是否对每个像素求逆（即黑变白、浅灰变深灰、绿变绛红等）。

事隔没多久，你又发现你的同事有了新的需求：他希望在传输位图时只传输对应于目的地上的那些被预置为黑色的部分（译注 1）。这一功能使我们能够显示非矩形图像。例如先在屏幕上刷一个黑色的圆饼，那么传输到屏幕上的图像便只显示这个圆饼内的部分。该特性使得能够在目的地显示非方形的图案。

随着你的进一步观察，你或许会发现这些选项其实可以泛化。比如，考虑一个黑白图形图像系统：每个像素都由一位 (bit) 来表示，0 为黑，1 为白。在这样的一个系统中，一个源位图就是一个位数组，屏幕也是一个位数组。目的地上的每一个特定像素的颜色取决于源位图的对应像素的颜色 (0 或 1) 和目的地上的那个特定像素自身的颜色 (0 或 1)。

这种将源位图和目标位图的对应像素进行的某种操作叫做光栅操作，一共有 16 种，如表 8-1 所示。

表 8-1：基本光栅操作

可能的组合		
参数名	参数值	
源 (S):	1 1 0 0	
目标 (D):	1 0 1 0	
操作	输出	逻辑表示
光栅操作 0:	0 0 0 0	0
光栅操作 1:	0 0 0 1	$\sim(S \mid D)$
光栅操作 2:	0 0 1 0	$\sim S \& D$
光栅操作 3:	0 0 1 1	$\sim S$
光栅操作 4:	0 1 0 0	$S \& \sim D$
光栅操作 5:	0 1 0 1	$\sim D$
光栅操作 6:	0 1 1 0	$S \wedge D$

译注 1：类似 Flash 中的“遮罩”技术。

表 8-1：基本光栅操作（续）

可能的组合		
操作	输出	逻辑表示
光栅操作 7:	0 1 1 1	$\sim(S \& D)$
光栅操作 8:	1 0 0 0	$S \& D$
光栅操作 9:	1 0 0 1	$\sim(S \wedge D)$
光栅操作 10:	1 0 1 0	D
光栅操作 11:	1 0 1 1	$\sim S \mid D$
光栅操作 12:	1 1 0 0	S
光栅操作 13:	1 1 0 1	$S \mid \sim D$
光栅操作 14:	1 1 1 0	$S \mid D$
光栅操作 15:	1 1 1 1	1

为什么是 16 种不同的光栅操作呢？因为每一个“源像素—目标像素”对，都有 4 种不同的可能 $((0,0), (0,1), (1,0), (1,1))$ ；而一个光栅操作对每种可能都有两种不同的操作结果。也就是说一个光栅操作一共有 $2^4=16$ 种不同的可能。表 8-1 列出了这 16 种不同的光栅操作，每种光栅操作的编号正好对应于光栅操作的结果值，而相应的“逻辑操作”，即实际进行的布尔操作（C 语法）显示在表格的最后一列。

例如，对于光栅操作 12（最常见的一种），源位图被直接传输到目的地。而对于光栅操作 14，仅当目标位图中相应位为 0（黑色）时，源位图中的相应位才会被传输。光栅操作 10 则维持目标位图不变，不管源位图是什么。光栅操作 0 和 15 只是简单地将目标位图置为黑和白色，同样不管源位图是什么。

在一个彩色图像系统中，每个色素一般都由 24 位来表示，其中 8 位表示红色分量，8 位表示绿色分量，8 位表示蓝色分量。如果所有位都是 0，则表示黑色；反之，所有位都为 1 则表示白色。光栅操作作用于源位图和目标位图的对应像素上。比如说，光栅操作 14 能够仅在目的地预置为黑色的区域内显示源位图相应的部分，目的地白色的地方还是白色。不过，如果目的地的某个像素原本是红色的，而对应的源像素是蓝色的，那么结果像素便会是蓝色和红色的混合（绛红色）。这跟前面讲的黑白显示系统的情况虽略有不同，但光栅操作的结果仍然还是完全可预测的。

在 Windows 上，BitBlt 和 StretchBlt 所用的光栅操作则更为复杂。Windows 支持一种叫做画刷（brush，也叫 pattern（模式））的图形对象，画刷通常用于填充闭合区域。画刷可以代表一种实颜色，也可以代表一张重复图像，如哈希遮罩或砖块。要实现这种操作，

BitBlt 和 StretchBlt 便会基于源位图、目标位图以及一个给定的画刷这三者来进行光栅操作。画刷的存在允许程序在无需知道目标位图的情况下更改源位图中的像素位（比如将它们求逆或对它们进行遮罩）。

由于 BitBlt 和 StretchBlt 实现的光栅操作涉及了三个对象：源、目标、模式（画刷），因此也被称为三元光栅操作。一共有 256 种三元光栅操作，BitBlt 和 StretchBlt 全都支持。

根据前面的讨论，如果你从单色图形图像系统入手的话，这 256 种三元光栅操作其实也并不难理解。不仅源和目标，画刷（模式）本身也是一个位数组，其中每一位都代表一个像素；你可以想像画刷覆盖在目标位图表面上。表 8-2 展示了这 256 种三元光栅操作中的一部分，其中 0 和 1 表示在模式源和目标中可以组合的像素值。

表 8-2：三元光栅操作

可能的组合	
参数名	参数值
模式（画刷）(P):	1 1 1 1 0 0 0 0
源 (S):	1 1 0 0 1 1 0 0
操作	输出
光栅操作 0x00:	0 0 0 0 0 0 0 0
光栅操作 0x01:	0 0 0 0 0 0 0 1
光栅操作 0x02:	0 0 0 0 0 0 1 0
...	...
光栅操作 0x60:	0 1 1 0 0 0 0 0
...	...
光栅操作 0xFD:	1 1 1 1 1 1 0 1
光栅操作 0xFE:	1 1 1 1 1 1 1 0
光栅操作 0xFF:	1 1 1 1 1 1 1 1

表 8-2 展示了 256 种不同的光栅操作（实际只列出了 7 种，其余省略）的输出结果。其中每种光栅操作都可以用一个惟一的 1 字节十六进制值来表示，这个值即是表格中最后一列的输出结果值。例如，对于光栅操作 0x60，如果画刷像素为 1（白），源像素为 0（黑），目标像素为 1，则计算结果便为 1（白）。

在 Windows 的早期版本中，256 种光栅操作里面有 15 种起了名字，放在 Windows 头文件和 API 文档中，供 C 程序员使用。其中第一个光栅操作被称为 BLACKNESS（效果是

将目标位图涂成全黑，而不管画刷和源位图是什么）；最后一个被称为 WHITENESS。

Windows 编程手册则将所有 256 个光栅操作都命了名，命名的时候基于的是它们各自所执行的位布尔操作，用逆波兰式表达。例如，光栅操作 0x60 对应于布尔操作 *PDSxa*。

这个表达式的意思是：先在目标（D）和源（S）之间执行一次异或操作（x），然后将其结果再跟画刷（P）作一次按位“与”操作（a）。在彩色系统中的操作也很类似，只不过操作的是颜色分量。截止到写作本章为止，这里提到的光栅操作可以在 http://msdn.microsoft.com/library/en-us/gdi/pantdraw_6n77.asp 中浏览。

这些光栅操作当中有些在某些场合下是非常有用的。比如，如果你想要在画刷为黑色的区域对目标像素进行反色，并在画刷为白色的区域显示源像素的话，光栅操作 0xC5 就可以做到。当然，话说回来，256 个光栅操作里面真正有用的的确不多，我怀疑其中大部分除了用作演示和试验之外根本没有在其他地方被用过。不过这种完备和多样选择的感觉倒的确令人心安。

现在，你可以想一想，如果让你来实现这个多用性的 BitBlt 函数，你会怎么实现？假设现在是 1985 年，你使用的是 C 语言。既然是演示，那就不妨让我们假设我们面对的是一个每像素占用一个字节的灰度图形图像系统，其中源、目标和画刷分别可以通过 S、D、P 这三个二维数组来访问。也就是说，S、D、P 每一个都是一个指向一组字节指针的指针（二级指针），而其中每个字节指针都指向一个横行的像素数组。这就是说， $S[y][x]$ 访问的是第 y 行 x 列的字节。而工作区域的宽和高则放在 cx 和 cy 变量中（cx 和 cy 是传统的 Windows 编程变量命名方式：c 代表 count，意即 cx 代表离散的 x 值的数目，也就是宽）。而 rop 变量存放的则是光栅操作的代码（从 0 到 255 编号）。

先来看看下面这段简单的 C 代码。在这段代码中，一个 switch 语句对 rop 进行判断，以确定使用什么操作来计算目标位图中的像素。当然，一共有 256 种光栅操作，这里只列出了 3 种，其他省略掉了，但不影响我们表达意思：

```
for (y = 0; y < cy; y++)
    for (x = 0; x < cx; x++)
{
    switch(rop)
    {
        case 0x00:
            D[y][x] = 0x00;
            break;
        ...
        case 0x60:
            D[y][x] = (D[y][x] ^ S[y][x]) & P[y][x];
            break;
        ...
    }
}
```

```
    case 0xFF:  
        D[y][x] = 0xFF;  
        break;  
    }  
}
```

这段代码自然算得上是漂亮的代码；这里漂亮的意思是代码看上去整洁，意图和功能一目了然。但如果要说美丽，它其实算不上，因为美丽的代码不仅要形式漂亮，运行起来也同样要“漂亮”。

从后者的意义上说，这段代码其实是个灾难。为什么这么说呢？因为它操作的是位图。位图有什么特点？庞大。如今来自一台廉价数码相机的位图都能包含百万级像素。你真的希望上面那个大大的 switch 语句出现在双重（行、列）循环体中吗？

很自然的，你可能会想到：难道一定要对每个像素执行一次 switch 操作吗？其实不必。只需将代码里外翻转一下：把 for 循环移到 switch 的每个 case 中。这么一来代码虽然没原来那么漂亮了，但效率却得到了显著的提高：

```
switch(rop)  
{  
    case 0x00:  
        for (y = 0; y < cy; y++)  
            for (x = 0; x < cx; x++)  
                D[y][x] = 0x00;  
        break;  
    ...  
    case 0x60:  
        for (y = 0; y < cy; y++)  
            for (x = 0; x < cx; x++)  
                D[y][x] = (D[y][x] ^ S[y][x]) & P[y][x];  
        break;  
    case 0xFF:  
        for (y = 0; y < cy; y++)  
            for (x = 0; x < cx; x++)  
                D[y][x] = 0xFF;  
        break;  
}
```

当然，如果现在是 1985 年，而你写的是 Windows 系统的话，你甚至都不会用 C 来写上面这些代码。虽说早期的 Windows 应用几乎都是用 C 写的，然而 Windows 自身却是用 8086 汇编语言写的。

而像 BitBlt 这种对 Windows 极其重要的函数，是要无所不用其极的。于是，早期的天才程序员们想出了更诡异的手法，这种手法居然“比汇编还快”，听起来仿佛天方夜谭吧？可想而知，那些想出这招的微软程序员们对此非常自豪，而我们这些 80 年代中期学习 Windows 编程的程序员听了这些轶事也同样是心向往之。

那么这种神奇的手法到底是什么呢？其实，BitBlt 内部包含了一个“迷你”编译器。基于参数中给出的特定的光栅操作（如图像格式、每像素所占的位（bit），以及工作区域的大小），BitBlt会在栈上建立起一个由 8086 机器代码构成的子进程，然后执行它。这个即时建立起来的由机器代码构成的子进程不需要 switch，而只需要对每个像素执行给定的光栅操作即可。

对于 BitBlt 和它的 256 个光栅操作来说，这种手法堪称完美。尽管这个迷你编译器需要一些运行时间来生成机器代码，然而这个小小的代价带来的好处就是对每个像素执行的处理能够最快地完成，后者正是在处理位图的时候最需要注意的地方。此外，由于不再需要在每个 case 里面包含所有 256 个光栅操作的硬编码，所以实现起来反而简短得多。

此外，通过三元光栅操作的文档我们甚至可以窥视到 BitBlt 内部的迷你编译器的工作方式。例如，0x60 号光栅操作实现的是 PDSxa。调用 BitBlt 时，你提供的是一个 32 位的光栅操作代码，这个代码在文档里面的值是 0x00600365。注意两个地方：1) 0x60 被嵌在这个数里面，2) 末尾的两个字节是 0x0365。

0xF6 号光栅操作对应的布尔操作为 PDSxo，跟 PDSxa 很相近，除了末尾的操作一个是 OR 一个是 AND 外。PDSxo 的 32 位光栅操作代码是 0x00F70265。末尾两个字节是 0x0265，与 PDSxa 的 0x0365 非常接近。如果你再找几个这样的 32 位光栅操作代码研究研究的话，你也许很快就会发现：光栅操作代码本身就被 BitBlt 内部的迷你编译器用来作为生成机器代码的某种模板。这种做法的好处就是省去了 BitBlt 将光栅操作号（通过比如说一个查找表）映射到光栅操作代码所需要消耗的时间和空间。

当然，Windows 1.0 版本是 20 年前的事了。时代在进步，Windows 在进步。如今我最喜爱的编程语言既不是汇编也不是 C 了，而是 C#。我常常编写的是运行在微软.NET 框架上的托管代码。C# 编译器将我的源代码编译成与处理器无关的中间语言（常被称为微软中间语言——MSIL）代码。而对这些中间代码的进一步编译（成机器代码）则是在程序运行的时候“按需”、“即时”进行的：只有当一段代码被调用到的时候,.NET 公共语言运行时才会即时地将这段代码从中间语言编译为本地机器代码。

即便如此，所有数字图像处理仍然还是狮子大张口地要求效率，不管你用什么古怪的编码技术。当面对的是百万级像素时，每像素处理必须得尽可能地快！快快！如果你开发的是商业产品，那么你很可能希望找到一个会汇编的，甚至会针对显卡上的 GPU（图形、图像处理单元）编程的程序员。即便你的程序是临时写着用的或非商业的，你或许也会希望程序跑得比高级语言的循环快。

我是最近在用C#写数字图像过滤器的时候想起Windows (BitBlt) 中曾用到的这种即时代码生成技术的。BitBlt 和 StretchBlt 中实现的那些光栅操作仅对相应的源像素、目标像素、画刷像素执行操作。而数字图像过滤器则会将某个特定像素周围的像素也纳入操作。数字图像过滤器有图像锐化、模糊化等。模糊化过滤器会将位于一个领域内的一组像素求平均从而得到一个目标像素，以获得模糊的效果。

简单的数字图像过滤器通常由一个数字阵列实现。这些数字阵列通常都是方阵，并且行数和列数都是奇数。图 8-1 是一个简单的例子。

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

图 8-1：一个简单的数字图像过滤器

图 8-1 是一个 3 乘 3 的过滤器，你可以把它理解成将一个源位图转换成一个目标位图。怎么转换的呢？对于源位图的每一个像素，将这个过滤方阵覆盖到这个像素上，使得方阵中心的元素刚好对准这个像素，剩下的 8 个元素则分别对应该像素周围的 8 个像素。将过滤阵列中的 9 个值跟它们对应的 9 个像素分别相乘，然后将乘的结果加到一起，得到的结果便是目标像素的值。如果位图中的像素编码了色彩分量或透明度，那么在运算的时候便需要对每个颜色通道分别进行运算。有些过滤器对不同的颜色通道会提供不同的过滤阵列；或者也可能未必用过滤阵列实现，而是用算法。不过本文不打算对此深入讨论，使用阵列的简单办法对于我们的讨论已经够用了。

回到刚才的那个数字阵列，其中每个元素都是 $1/9$ ，这种阵列称为模糊过滤器。目标位图中的每个像素都是其在源位图中的 9 个邻接像素的平均。注意，本例中模糊过滤器中的各个元素值加起来是 1，这样转换后的图像便不会变得更亮或更暗；但你并不一定要这样，你也可以让所有元素都为 1 或者其他什么数，不过要想对明暗度进行补偿的话，你便需要将乘积和除以阵列中的各元素和（后面你会看到这种做法，我其实更喜欢这种做法）。

图 8-2 是一个锐化过滤器。这个过滤器的作用是将高对比度的区域进一步凸显出来。

0	-1	0
-1	4	-1
0	-1	0

图 8-2：一个锐化过滤器

假设我们要处理的是灰度位图，每个像素占一个字节。源位图的像素存放在一个叫做 S 的二维数组中。而我们的任务则是计算出目标数组 D 中的各像素值。两个位图数组的宽高被放在 cxBitmap 和 cyBitmap 两个变量中。过滤器是一个二维数组 F，宽和高分别为 cxFilter 和 cyFilter。示例 8-1 中是使用该过滤器的简单 C 代码：

示例 8-1：使用一个数字过滤器的简单 C 代码

```

for (yDestination = 0; yDestination < cyBitmap; yDestination++)
for (xDestination = 0; xDestination < cxBitmap; xDestination++)
{
    double pixelsAccum = 0;
    double filterAccum = 0;

    for (yFilter = 0; yFilter < cyFilter; yFilter++)
        for (xFilter = 0; xFilter < cxFilter; xFilter++)
        {
            int ySource = yDestination + yFilter - cyFilter / 2;
            int xSource = xDestination + xFilter - cxFilter / 2;

            if (ySource >= 0 && ySource < cyBitmap &&
                xSource >= 0 && xSource < cxBitmap)
            {
                pixelsAccum += F[y][x] * S[y][x];
                filterAccum += F[y][x];
            }
        }
    if (filterAccum != 0)
        pixelsAccum /= filterAccum;

    if (pixelsAccum < 0)
        D[y][x] = 0;
    else if (pixelsAccum > 255)
        D[y][x] = 255;
    else
        D[y][x] = (unsigned char) pixelsAccum;
}

```

注意，内层的那个针对过滤阵列的双重 for 循环会对两个值进行累加。一个是 pixelsAccum，pixelsAccum 是源像素与过滤阵列中元素的乘积之和。另一个是 filterAccum，

`filterAccum`是过滤阵列中元素本身的和。对于位于位图边缘的像素，过滤阵列中的一些元素会对应到超过源位图边缘的不存在的像素上去。这时我一般采取忽略这些元素的做法，即在循环中不向 `pixelsAccum` 和 `filterAccum` 上加东西，随后将 `pixelsAccum` 用 `filterAccum` 相除仍能保证目标像素近似正确。这就是为什么我不在循环外面计算 `filterAccum` 的原因，同样也是为什么过滤阵列中各元素值不被规范化到和为 1 的原因。此外，注意一下最后几行代码：`pixelsAccum` 和 `filterAccum` 相除的结果必须被“压”到 0 和 255 之间，因为只有在这个区间内才是有意义的像素值。

分析上面的代码，我们会发现，对于目标位图上的每一个像素，源位图和过滤阵列都必须被访问 9 遍。不仅如此，随着位图的解析度的提高，过滤阵列通常也要随之增大，因为只有这样才能确保对图像起到一个可观察到的影响。

对于高阶语言来说以上的工作量还是挺大的，但我很好奇 C# 和 .NET 在这样的效率压力下表现会如何。在尝试用 C# 进行图像处理时，我首先用了我的《Programming Windows with C#》书上的一些代码。该书第 24 章的 `ImageClip` 程序包含了能够加载、查看、打印和保存多种流行图像格式（JPEG、GIF、PNG 等）的代码。然后加上我为本文写的代码（都可以在网上下载到），合成了一个名为 `ImageFilterTest` 的程序。项目文件需用 Visual Studio 2005 才能打开，编译后的执行文件需要 .NET 框架 2.0 及以上版本来运行。程序的使用方式如下：

1. 在“文件”菜单中选择“打开”，打开一幅彩色位图。程序中进行过滤的代码只支持 24 位和 32 位每像素的位图；不支持使用调色板的位图，包括调色板包含的是灰度的情况。
2. 在“过滤器”菜单中选择一个过滤器。该过滤器会被应用到打开的位图上，处理所耗的时间会被记录并汇报。过滤器菜单中的第一个选项“使用能够生成中间语言代码的方法”允许你选择用哪个方法来运用过滤器。默认情况下，程序会使用一个叫做 `FilterMethodCS` 的方法（“C# 过滤方法”的简称）。但如果你选择了刚才说的那个菜单选项，程序便会使用 `FilterMethodIL`（“中间语言的过滤方法”）。这两个方法本章的后半部分都会介绍。

任何时候，当你想要编写出最高效的 C# 代码时，一件很有意思的事情是使用 .NET SDK 提供的一个名叫 *IL Disassembler* 的小工具来检查编译过的文件。这个小工具会展示 C# 编译器生成的中间语言代码。尽管你看不到最后一步转换（JIT 编译器将中间语言代码编译为机器代码），但使用该工具仍然还是能够定位许多问题的。

首先要说明的是，几乎一开始我就放弃了将位图像素存放在二维数组中的想法。C# 支持

多维数组，但在中间语言层面，从一个多维数组中获取或存放元素要用到方法调用。不过，对于一维数组的访问中间语言倒是有指令级的支持。此外，将像素从一张位图传输到一个数组中（以及再倒腾回去）的标准（且快速的）做法是用一维数组。而我曾编写的代码是用二维数组，结果二维数组本身的访问就消耗了大量时间。

为了封装图像过滤器以及将过滤器运用到位图上的函数，我创建了一个名叫 ImageFilter 的类，该类包含三个私有数据成员和一个构造函数。私有数据成员 filter 是一个一维数组，其中存放的是一个二维的过滤阵列，而 cxFilter 和 cyFilter 分别表示阵列的列数和行数。

```
class ImageFilter
{
    double[] filter;
    int cxFilter;
    int cyFilter;
    public ImageFilter(int cxFilter, double[] filter)
    {
        this.filter = filter;
        this.cxFilter = cxFilter;
        this.cyFilter = filter.Length / cxFilter;
    }
}
```

如果只允许过滤阵列为方阵的话，那么 ImageFilter 构造函数的 cxFilter 参数就不必要了，因为可以直接通过阵列的大小（阵列中元素个数，即 filter.Length）开平方来求得列数/行数。既然 cxFilter 代表过滤阵列的列数，那么 filter.Length / cxFilter 就是行数，我的代码假定除的结果是一个整数。

Filter 类中有一个 ApplyFilter 方法，其参数之一是一个 Bitmap，至于其方法体这里就不展示了，因为它做的大致也就是一些标准工作：首先（使用 LockBits）访问 Bitmap 对象中的所有像素，然后将它们传输到一个一维数组中。ApplyFilter 的第二个参数是一个布尔值，叫做 willGenerateCode。如果这个值是 false，则 ApplyFilter 就会调用 FilterMethodCS。

示例 8-2 展示的是 FilterMethodCS 方法。它是示例 8-1 中过滤算法的一个直观实现。只不过这里是用 C# 编写的，并且使用的是一维数组。

示例 8-2：用 C# 编写的一个数字过滤算法

```
1 void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2 {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
```

```

5
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8         double pixelsAccum = 0;
9         double filterAccum = 0;
10
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13            int yFilter = iFilter / cyFilter;
14            int xFilter = iFilter % cxFilter;
15
16            int iSrc = iDst + stride * (yFilter - cyFilter / 2) +
17                         bytesPerPixel * (xFilter - cxFilter / 2);
18
19            if (iSrc >= 0 && iSrc < cBytes)
20            {
21                pixelsAccum += filter[iFilter] * src[iSrc];
22                filterAccum += filter[iFilter];
23            }
24        }
25        if (filterAccum != 0)
26            pixelsAccum /= filterAccum;
27
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                         (byte)255 : (byte)pixelsAccum);
30    }
31 }

```

FilterMethodCS 的前两个参数 src 和 dst 为源数组和目标数组。第三个参数 stride 代表的是源位图和目标位图每行所占的字节数，该值通常等于位图的像素宽乘以每像素所占字节数，只不过出于效率的考虑，这个值通常会被向上圆整到 4 字节的倍数（因为这个程序只支持彩色位图，所以每像素所占字节数总是 3 或 4）。我们并不需要自己计算 stride 的值，因为 LockBits 方法返回的信息里面就包含这个值，当你需要访问一副位图中的像素时，你会对它调用 LockBits，于是就得到了 stride 值。FilterMethodCS 的方法体先是将 src 和 filter 数组的长度保存到局部变量中，免得每次都要通过 Length 属性去访问。此外那些以“i”开头的变量分别是三个数组的下标索引变量。

如果我们的目标是写一个快速的数字过滤器算法，那么 FilterMethodCS 就不靠谱了。因为对于一幅每像素 24 位、30 万像素的位图，加上一个 5 乘 5 的过滤阵列，FilterMethodCS 在我的 1.5G Pentium 4 处理器上需要两秒时间才能运行完。你可能会觉得两秒看上去也没什么太大不了的，但如果你把一个 5 乘 5 的过滤阵列运用到一幅每像素 32 位、4.3 兆像素的位图上，你就会发现足足要等上半分钟，半分钟总够长了吧。而且我还想不出有什么办法能提高我的 C# 代码的速度的。

如果一个函数运行不够快，而且你也觉得没法再优化了时该怎么办？传统招数是祭出汇

编。在平台无关和托管代码领域，你也可以用一个类似的办法：直接用中间语言编写代码。这当然是个听上去蛮不错的办法，而且甚至还相当有趣（对于某些特定类型的家伙来说）。然而问题是，即便用中间语言编码也未必就足够了。你不妨用 IL Disassembler 来查看一下 C# 编译器为 FilterMethodCS 生成的中间语言代码，你真的能写得比编译器还好吗？

FilterMethodCS 的真正问题在于它是一个泛化了的方法：它能够对任意大小的位图利用任意大小的过滤阵列。FilterMethodCS 中的大量代码都是在循环和索引。而如果 FilterMethodCS 不需要应对这么一般性的情况的话，其效率便可以大大提高了。举个例子，假设 FilterMethodCS 只需要对付每像素 32 位，特定大小的位图（宽高假设为 CX 和 CY，你可以把 CX 和 CY 理解为 C/C++ 里面的编译期常数，用 #define 或 const 变量定义），并假设你总是使用同一个过滤阵列（假设是一个 3 乘 3 的阵列，元素值固定，如图 8-3 所示）的话。

F11	F12	F13
F21	F22	F23
F31	F32	F33

图 8-3：一个 3 乘 3 的过滤阵列

这么一假设的话，你会怎么编写你的 FilterMethodCS 呢？你可能立即就会抛弃 iFilter 循环，直接手动遍历过滤阵列内的 9 个元素了：

```
// 过滤器元素 F11
int iSrc = iDst - 4 * CX - 4;
if (iSrc >= 0 && iSrc < 4 * CX * CY)
{
    pixelsAccum += src[iSrc] * F11;
    filterAccum += F11;
}

// 过滤器元素 F12
iSrc = iDst - 4 * CX;
if (iSrc >= 0 && iSrc < 4 * CX * CY)
{
    pixelsAccum += src[iSrc] * F12;
    filterAccum += F12;
}

// 从 F13 到 F32 的过滤器元素
...

```

```
// 过滤器元素 F33  
iSrc = iDst + 4 * CX + 4;  
  
if (iSrc >= 0 && iSrc < 4 * CX * CY)  
{  
    pixelsAccum += src[iSrc] * F33;  
    filterAccum += F33;  
}
```

这种手动解循环的办法消除了循环逻辑，简化了 `iSrc` 的计算，消除了对过滤阵列的访问。尽管代码变得臃肿不堪，但至少速度上去了。实际上，由于你知道过滤阵列中所有元素的确切值，你甚至可以把代码中那些遇到过滤阵列元素值为 0 的情况删掉，并简化那些过滤阵列元素为 1 或 -1 的情况。

当然，这种笨办法实际上是无法实施的，因为你本来就需要对付不同大小的位图和不同类型的过滤阵列。这些属性在编译期原本是不知道的，只有等到运行期真正使用过滤器的时候才知道。

那么难道就没有办法了吗？其实是有的。你只要在运行期“即时地”根据参数的具体值（位图的大小和像素深度（译注 2），以及过滤阵列的大小和其中的元素值）生成类似于手动解循环的代码即可。如果这是在远古 Windows 时代，你或许可以参考 BitBlt 的做法，即在内存中生成机器代码，然后跳到其中执行。然而既然这是现代，就不能用钻木来取火了，再加上可移植性的考虑，我们也许可以试试利用 C# 来生成中间语言代码然后执行。

事实证明，这的确是一个可行的办法。在 C# 程序里面，我们可以在内存里面创建一个包含中间语言指令的静态方法，然后跳到该方法执行。在该方法被执行的前夕,.NET 的 JIT 会自动跳出来把该方法编译成目标机器代码。最漂亮的是，在这整个的过程中你编写的全都是托管代码！

.NET 框架里面用于动态生成中间语言代码的设施是在.NET 2.0 加进去的，由 `System.Reflection.Emit` 命名空间中的一组类组成。利用该设施，你可以生成整个的类，甚至整个的程序集（Assembly），而对于较小的应用（比如本文的这个），你也可以只生成一个静态方法，然后调用它。我在 `ImageFilter` 类的 `FilterMethodIL` 方法中便是这么做的。

下面我会展示 `FilterMethodIL` 的所有代码（略掉了一些代码注释，你在 `ImageFilter.cs` 中可以看到完整的版本），你会看到 C# 代码与生成的中间语言代码的一些很有意思的

译注 2：即存储每个像素所用的位数。

交互。在浏览代码的时候始终别忘了：FilterMethodIL是针对任意一个给定的过滤阵列和任意一个给定的位图来生成对应的中间语言代码的，因为这样一来过滤阵列的所有方面和位图的大小及像素深度便都被“硬编码”在了生成的静态方法中了。显然，为了生成这些代码，一些运行期的开销是少不了的，但跟一幅庞大的（往往得百万像素以上）位图所要消耗的巨量操作比起来，这点点开销就实在是微不足道了。

我会把FilterMethodIL中的代码陆续展示在下面，并逐段解释代码中引入的新概念以及代码都做了些什么。FilterMethodIL和FilterMethodCS的参数是一样的。并且，它们一开始都是先获取位图的大小：

```
void FilterMethodIL(byte[] src, byte[] dst, int stride, int bytesPerPixel)
{
    int cBytes = src.Length;
```

要在运行期创建一个静态方法，你得先创建一个DynamicMethod的新对象。该对象构造函数的第二个参数代表的是你要创建的静态方法的返回类型，第三个参数是一个类型数组，对应于该方法的参数类型列表。第四个参数是创建该方法的类（可通过GetType()获得）。

```
DynamicMethod dynameth = new DynamicMethod("Go", typeof(void),
    new Type[] { typeof(byte[]), typeof(byte[]) }, GetType());
```

通过该构造函数的第三个参数你可以看出来，我们动态创建的这个方法的两个参数皆是字节数组（byte[]），其实它们便对应示例 8-2 中的 src 和 dst 数组。在后面将要展示的中间语言代码中，这两个参数分别通过索引 0 和 1 来访问。

然后我们的工作是生成该方法的方法体，也就是生成一堆中间代码。首先我们获取一个ILGenerator 类型的对象：

```
ILGenerator generator = dynameth.GetILGenerator();
```

后面的大部分工作都要用到这个 generator 对象。你可以从定义该方法的一些局部变量开始。我觉得声明三个局部变量比较合适，刚好对应于FilterMethodCS中的那三个：

```
generator.DeclareLocal(typeof(int));      // 索引 0 = iDst
generator.DeclareLocal(typeof(double));     // 索引 1 = pixelsAccum
generator.DeclareLocal(typeof(double));     // 索引 2 = filterAccum
```

以上三行语句声明的三个变量分别对应示例 8-2 中的第 3、4、6 行的声明。注意注释中的说明：这些临时变量也将通过索引来访问。有了局部变量的声明，我们便可以定义基于 iDst 的循环了，这个循环用于遍历目标数组中的所有像素。

剩下基本就是生成中间语言操作代码了，中间语言操作代码跟机器语言操作代码很像。中间语言指令由一个一字节的操作代码加上可选的参数构成。不过与机器代码不同的是，你不必去关心二进制的字节和位。要生成这些指令，只需调用 `ILGenerator` 类上重载的 `Emit` 方法即可。`Emit` 的第一个参数总是一个 `OpCode` 类型的对象，并且所有可行的操作代码都已经预定义为 `OpCodes`（注意，不是 `OpCode`）类上的静态只读字段了。截至到写本文止，你可以在 <http://msdn2.microsoft.com/library/system.reflection.emit.opcodes.aspx> 访问到 `OpCodes` 类的文档。

中间语言中的大部分赋值和操作逻辑都是基于一个虚拟的求值栈来进行的（之所以称它为“虚拟的”是因为实际最终被你的机器执行的代码是由 JIT 编译器即时生成的机器代码，而这些机器代码就不一定是基于求值栈执行的了）。例如，`load` 指令会将一个值压进求值栈。这个值既可以是一个立即数，也可以是一个临时变量等；`store` 指令则从栈中取出顶部的元素，并将它存放到一个临时变量中或其他地方。算术和逻辑指令也是基于栈来执行的。比如说 `add` 指令会从求值栈中弹出两个值，相加，然后将结果再压入栈。

要想通过中间语言将局部变量 `iDst` 的值设为 0，我们需要用一个 `load` 指令加上一个 `store` 指令。在下面两行代码中，`Ldc_I4_0` 指令会将一个四字节的整型值 0 入栈，然后 `Stloc_0` 则会将这个值存入编号为 0 的（即对应于 `iDst` 的那个）局部变量中。

```
generator.Emit(OpCodes.Ldc_I4_0);
generator.Emit(OpCodes.Stloc_0);
```

尽管许多高级编程语言都包含了 `goto`（或类似的）指令，但一般来说 `goto` 是不被推荐的用法。不过，在汇编语言和中间语言级别，`goto`（也就是我们平常所说的跳转或分支指令）是惟一可用的执行流控制手段。所有的 `for` 和 `if` 语句都得靠它来模拟。

.NET 中间语言支持一个无条件分支语句以及一些条件分支语句。条件分支依赖于代码前文中特定的比较操作的结果。例如，“branch-if-less-than”（如果小于则转入分支）指令会在前文的比较操作中一个值小于另一个值的情况下转入一条支流。在中间语言中模拟一个 `if/else` 结构需要两个语句标签，一个标签打在 `else` 块的开头，另一个则标在其结尾。如果 `if` 检测的条件为 `false`，则用一个条件分支指令将控制流转向第一个标签处；否则执行 `if` 块内的语句，在 `if` 块的结尾会有一个无条件的分支指令，无条件跳转到 `else` 块结尾的标签处。图 8-4 展示了这两种情况：

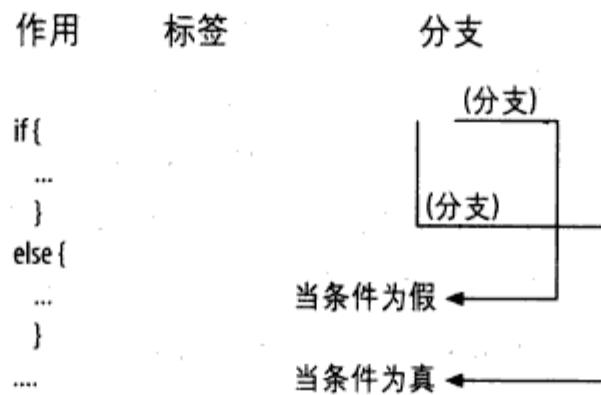


图 8-4：用中间语言分支指令来实现 if/else

中间语言中实际的分支指令包含了一个数值，代表跳转到的目标指令的地址与当前指令地址之间的偏移。要让程序员自己去计算这个偏移显然太痛苦了，因此.NET提供了一个标签系统来简化这一任务。你所需做的便是在指令流中适当的地方插入标签，这样，当你将一个分支指向这个标签时，代码生成器会负责帮你生成偏移量。

要使用一个语句标签，需要两个方法调用。首先是 `DefineLabel` 这个方法，`DefineLabel` 负责定义一个标签，你可以在分支指令中引用这个标签。然后是 `MarkLabel`，`MarkLabel` 的功能是将一个标签插入到中间语言指令流中。完成了这两步之后，你就设置好了一个语句标签，并且能够在操作代码中跳转到该标签了。注意，即便一个标签当前还没有真正插入到执行流中，你也可以将分支语句的目的地设为它。下面两行代码调用了 `DefineLabel` 和 `MarkLabel` 两个方法建立了一个名为 `labelTop` 的 `Label` 对象，并将其插入到 `iDst` 循环的开头：

```
Label labelTop = generator.DefineLabel();
generator.MarkLabel(labelTop);
```

这个标签刚好对应示例 8-2 中的第 6 行处的 C# 代码的 `for` 语句开头。之所以要在这里设置一个标签，是因为在 `for` 循环的末尾还要跳转回来。

下面我们生成 `iDst` 循环体的中间语言代码，即每像素处理的代码，首先是将 `pixelsAccum` 和 `filterAccum` 初始化为 0。下面代码中的第一行 `Emit` 调用生成了一个 `Ldc_R8` 操作，该操作的含义是加载一个 8 字节的实数 0.0 到栈上。`Emit` 的第二个参数是实际的数值。注意，操作码和操作数必须匹配，比如这里的操作是 `Ldc_R8`，操作的是一个实数，因而其操作数就必须是实数类型的——0.0，如果你不小心用了 0 的话，C# 编译器便会将其解释为一个整数，结果就是在运行期抛出一个异常，告诉你程序无效：

```
generator.Emit(OpCodes.Ldc_R8, 0.0);
generator.Emit(OpCodes.Dup);
```

```
generator.Emit(OpCodes.Stloc_1);
generator.Emit(OpCodes.Stloc_2);
```

再来看上面的第二行代码。OpCodes.Dup的意思是把求值栈顶部的元素复制一个然后再压入栈上。再下面的两行代码，Stloc_1和Stloc_2这两个操作会依次将栈上的两个值存放到编号为1和2的局部变量中，也就是pixelsAccum和filterAccum中。再次提醒，你必须确保栈顶的两个值与pixelsAccum和filterAccum的类型分别符合，否则运行时你会收到一个异常，说JIT编译器发现程序是无效的。

现在，我们已经可以对过滤阵列中的每一个元素生成代码了。然而问题是不想换汤不换药的通过中间语言来循环遍历过滤阵列中的每个元素。我们希望过滤阵列中的每个元素都被硬编码在我们生成的中间语言代码中。比如说，假设过滤阵列有9个元素，我们希望生成9个类似的中间语言代码块（译注：也就是解循环）。为了达到这个目的，我们循环遍历过滤阵列中的每个元素：

```
for (int iFilter = 0; iFilter < filter.Length; iFilter++)
{
```

对于其中的任意一个特定元素，判断它是否为0，如果是的话则完全忽略该元素——不为它生成任何中间语言代码，直接跳到下一个元素。

```
    if (filter[iFilter] == 0)
        continue;
```

对于过滤阵列中的每个元素，src数组的下标都会是从iDst开始的一个偏移量。下面的C#代码就是用来计算这个偏移量的。注意，我们可以用C#代码来计算这个偏移量，因为我们要生成的中间语言代码只需要这个值，而并不需要这个计算的过程：

```
    int xFilter = iFilter % cxFilter;
    int yFilter = iFilter / cxFilter;
    int offset = stride * (yFilter - cyFilter / 2) +
                 bytesPerPixel * (xFilter - cxFilter / 2);
```

对一个数组元素进行访问或修改需要三个步骤。首先你得把该数组的引用压入求值栈；然后再把要访问或修改的元素的下标也入栈。最后，如果你只是想访问该元素的话，你就执行一个load指令；如果你想要修改该元素的值的话，你就执行一个store指令。对于过滤阵列中的每一个非零元素，src数组中相应的元素都必须被访问，因此现在我们可以把src数组的引用压入求值栈了：

```
    generator.Emit(OpCodes.Ldarg_0);
```

Ldarg指令加载的是我们生成的动态方法的参数，这里是Ldarg_0，也就是说第1个参数，也就是src数组了。

接下来我们要建立三个标签。之所以现在就创建它们是为了好让分支指令使用它们，但实际上它们会在后面才会真正被插入指令流中。

```
Label labelLessThanZero = generator.DefineLabel();
Label labelGreaterThanOrEqual = generator.DefineLabel();
Label labelLoopBottom = generator.DefineLabel();
```

对于过滤阵列中的每个元素，src 数组中的一个相应元素都必须被访问一下，该元素的下标为 iDst 加上由前面的 C# 代码计算出的 offset。下面这几行代码首先将 iDst 和 offset 入栈，然后用 Add 指令将它们相加（Add 指令做的工作就是从栈上弹出两个操作数，iDst 和 offset，然后将它们相加，并将结果入栈）。最后的两个 Dup 指令将栈顶元素（即相加的结果）重复两遍。

```
generator.Emit(OpCodes.Ldloc_0);           // 将 dst 下标入栈
generator.Emit(OpCodes.Ldc_I4, offset);      // 将 offset 入栈
generator.Emit(OpCodes.Add);                // iDst 加 offset
generator.Emit(OpCodes.Dup);                // 复制两次
generator.Emit(OpCodes.Dup);
```

iDst 和 offset 相加的结果（即前文 FilterMethodCS 中的 iSrc 变量）可能会超出数组的长度（越界），因此，下面两行代码检查 iSrc 是否小于 0，如果小于 0 的话就跳转到 labelLessThanZero 标签处（这就实现了示例 8-2 中的第 19 行那个 if 判断的一半）。这两行代码首先是把一个立即数 0 压入求值栈上，然后 Blt_S 指令会比较栈顶两个元素（一个是刚压入的 0，一个就是前面求出的 iSrc）的大小并根据结果选择是否跳转。Blt_S 还有一个副作用就是会将栈顶的两个元素出栈。

```
generator.Emit(OpCodes.Ldc_I4_0);
generator.Emit(OpCodes.Blt_S, labelLessThanZero);
```

Blt_S 中的 Blt 意思是“Branch-if-LessThan”（小于则跳转），S 意思是“Short branch”（短程跳转，所谓短程跳转就是指跳转的偏移小于 256 操作码字节）。

接下来生成的中间语言代码会检查 iSrc 是否大于位图的大小。注意，我们将 cBytes 直接压入求值栈。这两行代码对应示例 8-2 中第 19 行处的 if 语句的第二个条件判断：

```
generator.Emit(OpCodes.Ldc_I4, cBytes);
generator.Emit(OpCodes.Bge_S, labelGreaterThanOrEqual);
```

如果 iSrc 没有越界，则代表我们可以访问 src 数组中相应的元素。于是接下来，我们生成一个 Ldelem 指令，该指令假设数组的引用以及待访问元素的下标已经位于栈上了，Ldelem 会将它们弹出来，通过它们求出待访问元素，并将其入栈。Ldelem_U1 的 U1 指的是该数组元素是一个 Unsigned 的 1 字节的值：

```
generator.Emit(OpCodes.Ldelem_U1);
generator.Emit(OpCodes.Conv_R8);
```

然后，Conv_R8 将刚才压入栈顶的那个元素转换并替换为一个 8 字节的浮点数。

现在，位于 iSrc 下标处的元素已经位于栈上了，而且也已经被转换成了一个浮点数。接下来我们就可以把它跟相应的过滤阵列元素相乘了。前面我们已经知道，过滤阵列的元素在生成该中间语言方法的时候就已经知道了，因此我们可以先通过 C# 代码来检查相应的过滤阵列元素是否为 1，如果是的话就不用做乘法了：

```
if (filter[iFilter] == 1)
{
    // src 元素在堆栈中，因此不做任何操作
}
```

而如果过滤阵列元素值为 -1，我们则可以简单地把 src 数组中相应元素求负，这可能会比乘以 -1 快一点：

```
else if (filter[iFilter] == -1)
{
    generator.Emit(OpCodes.Neg);
}
```

其他情况下，我们将 src 数组和过滤阵列中相应的元素相乘：

```
else
{
    generator.Emit(OpCodes.Ldc_R8, filter[iFilter]);
    generator.Emit(OpCodes.Mul);
}
```

接下来生成的代码则将 pixelsAccum 压入求值栈，然后将其与刚才相乘的结果相加，并将结果回存入 pixelsAccum 变量（注意，前面我们生成 pixelsAccum 这个局部变量时给它的编号是 1，所以我们用的是 Ldloc_1 和 Stloc_1）：

```
generator.Emit(OpCodes.Ldloc_1);
generator.Emit(OpCodes.Add);
generator.Emit(OpCodes.Stloc_1);
```

类似地，我们还要将过滤阵列的元素值累加到 filterAccum (2 号局部变量) 上：

```
generator.Emit(OpCodes.Ldc_R8, filter[iFilter]);
generator.Emit(OpCodes.Ldloc_2);
generator.Emit(OpCodes.Add);
generator.Emit(OpCodes.Stloc_2);
generator.Emit(OpCodes.Br, labelLoopBottom);
```

现在，我们刚好处于内层循环的结尾，即对应示例 8-2 的第 24 行。我们完成了对每个过

滤阵列元素的操作。至此我们只剩下栈清理工作没做。为什么还有栈清理工作呢？别忘了，前面当我们判断 *iSrc* 是否小于 0 的时候，如果小于 0 的话，后面判断它是否大于位图大小的那个操作便不做了，再后面的累加工作也不做了，也就是说栈上便剩下三个操作数没有弹出。类似的，如果判断 *iSrc* 大于位图大小的话，则剩下两个操作数没有弹出。于是有下面的扫尾代码：

```
generator.MarkLabel(labelLessThanZero);
generator.Emit(OpCodes.Pop);
generator.MarkLabel(labelGreaterThanOrEqual);
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Pop);
generator.MarkLabel(labelLoopBottom);
}
```

到目前为止，我们生成的所有代码所做的事情就是对一个特定的目标像素计算 *pixelsAccum* 和 *filterAccum*。这两个结果离目标像素的值只差一步之遥。于是我们将 *dst* 数组引用入栈（即生成的动态方法的第 2 个参数），将 *iDst* 入栈（第 1 个局部变量）：

```
generator.Emit(OpCodes.Ldarg_1); // dsto 数组
generator.Emit(OpCodes.Ldloc_0); // iDst 下标
```

由于下面还会有一些分支跳转，因此我们生成如下一些标签：

```
Label labelSkipDivide = generator.DefineLabel();
Label labelCopyQuotient = generator.DefineLabel();
Label labelBlack = generator.DefineLabel();
Label labelWhite = generator.DefineLabel();
Label labelDone = generator.DefineLabel();
```

下面的代码先是加载 *pixelsAccum* 和 *filterAccum* 这两个局部变量到栈上，以便对它们进行相除。但在相除之前我们还要先检查一下除数是否为 0，于是我们将除数 *filterAccum* 复制一份并将其与 0 作一个比较，当比较结果为相等的时候便跳转到 *labelSkipDivide*，这段代码对应示例 8-2 中的第 25 行。

```
generator.Emit(OpCodes.Ldloc_1); // pixelsAccum
generator.Emit(OpCodes.Ldloc_2); // filterAccum
generator.Emit(OpCodes.Dup); // 复制
generator.Emit(OpCodes.Ldc_R8, 0.0); // 将 0 入栈
generator.Emit(OpCodes.Beq_S, labelSkipDivide);
```

如果除数不为 0，我们便接着执行相除操作，除得的结果被自动入栈：

```
generator.Emit(OpCodes.Div);
generator.Emit(OpCodes.Br_S, labelCopyQuotient);
```

但如果除数 *filterAccum* 为 0，下面的代码就会被执行，它会将原先压入栈上的 *filterAccum* 的值弹出来：

```
generator.MarkLabel(labelSkipDivide);
generator.Emit(OpCodes.Pop); // 弹出 filterAccum
```

综上所述，不管 filterAccum 是否为 0，最后栈上留下的都是 pixelsAccum，当 filterAccum 不为 0 时，这个 pixelsAccum 是被 filterAccum 除过的；否则便还是原来的值。接着我们在栈上把这个结果复制两份：

```
generator.MarkLabel(labelCopyQuotient);
generator.Emit(OpCodes.Dup); // 复制结果
generator.Emit(OpCodes.Dup); // 再复制一次结果
```

再接下来的工作就是完成示例 8-2 中的 28 和 29 行了。如果我们发现除得的结果小于 0 的话，就将执行流引向 labelBlack，在 labelBlack 处我们会将目标像素设为 0：

```
generator.Emit(OpCodes.Ldc_R8, 0.0);
generator.Emit(OpCodes.Blt_S, labelBlack);
```

而如果发现除得的结果大于 255 的话，执行流则转向 labelWhite，在 labelWhite 处我们将目标像素的值设为 255：

```
generator.Emit(OpCodes.Ldc_R8, 255.0);
generator.Emit(OpCodes.Bgt_S, labelWhite);
```

其他情况（除的结果位于 0 和 255 之间）下，我们将除的结果（位于栈顶）转换为一个无符号、1 字节的值：

```
generator.Emit(OpCodes.Conv_U1);
generator.Emit(OpCodes.Br_S, labelDone);
```

以下代码预备将目标像素值设为 0。Ldc_I4_S 指令将一个 1 字节的值压入求值栈，只不过实际上它占用了栈上 4 字节的空间，因为求值栈上最小的一份空间是 4 字节：

```
generator.MarkLabel(labelBlack);
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Ldc_I4_S, 0);
generator.Emit(OpCodes.Br_S, labelDone);
```

以下代码则预备将目标像素值设为 255。类似地，它将 255 压入求值栈：

```
generator.MarkLabel(labelWhite);
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Ldc_I4_S, 255);
```

有了上面的预备之后，现在我们便可以设置最终的目标像素值了。现在，dst 数组的引用已经在栈上，iDst 值也已经在栈上，而最终目标像素的值刚才也放在栈上了。剩下的就是调用一下 Stelem_I1 即可。Stelem_I1 将一个 1 字节的值存放到数组中指定元素中。

```
generator.MarkLabel(labelDone);
generator.Emit(OpCodes.Stelem_I1);
```

现在我们终于到了 `iDst` 循环的底部了，以下代码对应示例 8-2 中的第 30 行。现在我们必须将 `iDst` 局部变量递增，然后跟 `dst` 数组的长度进行比较，如果小于则跳转到循环体的开头进行新一轮循环：

```
generator.Emit(OpCodes.Ldloc_0);      // 将 iDst 入栈
generator.Emit(OpCodes.Ldc_I4_1);      // 将 1 入栈
generator.Emit(OpCodes.Add);          // 将 1 增加到 iDst
generator.Emit(OpCodes.Dup);          // 复制
generator.Emit(OpCodes.Stloc_0);      // 将结果存入到 iDst
generator.Emit(OpCodes.Ldc_I4, cBytes); // 将 cBytes 值入栈
generator.Emit(OpCodes.Blt, labelTop); // 如果 iDst < cBytes，则返回栈顶
```

循环体之后是一个返回指令：

```
generator.Emit(OpCodes.Ret);
```

所有的中间代码生成到此结束。`FilterMethodIL` 的一开始我们生成的 `DynamicMethod` 对象现在已经可以执行（调用）了。调用很直观，用 `Invoke` 方法，传递给 `Invoke` 的第 2 个参数就是实际传给我们生成的动态方法的参数——`src` 和 `dst` 数组：

```
dynameth.Invoke(this, new object[] { src, dst });
}
```

到此，我们的 `FilterMethodIL` 便结束了。`DynamicMethod` 对象和 `ILGenerator` 对象的生命期也随之结束，它们占用的内存被.NET 垃圾收集器回收。

用低级语言编写的算法通常能比用高级语言写的快一些，而特化的算法则几乎总是要比泛化的算法更快。本文便展示了后一种情况：我们在运行期基于参数值动态生成了一个特化了的算法（中间语言代码），这么一来，我们既保证了算法本身是泛化的，又能保证算法真正执行的时候是特化的、高效的，一石二鸟。

不过，这种做法也有一个小小的不足，就是你必须了解一定程度的编译原理，你必须知道代码和数据之间其实是没有界限的，你的思想必须能够自由地在数据和代码之间游走。

不可否认，`FilterMethodIL` 中做了不少工作，但重点是 `FilterMethodIL` 的效率如何呢？答案是运行时生成中间语言指令的 `FilterMethodIL` 只需 1/4 的时间就完成了 C# 版本的 `FilterMethodCS` 的工作，有时候甚至还更快。

OK，你可能会说，“但 `FilterMethodIL` 很丑啊，我们难道不是在讨论‘美丽的代码’么？”是的，我承认它外表的确算不上太漂亮。但对于一个算法来说，如果有办法能够把它的耗时降到原来的 1/4 的话，我觉得只能用漂亮来形容了。

自顶向下的运算符优先级

Douglas Crockford

1973 年，第一届年度编程语言原理研讨会（注 1）在波士顿举行，会上 Vaughan Pratt 宣讲了《自顶向下的运算符优先级》。Pratt 在这篇论文里描述了一项解析技术，该技术融合了递归下降分析和 Robert W Floyd（注 2）的运算符优先级句法技术的最佳特性。Pratt 声称这项技术浅显易懂，实现轻而易举，使用毫不费力，效率极高，而且非常灵活。我还要补充一点：它也很美妙。

如今，理想的编译器构造方法却被完全忽略，或许有些奇怪。为什么会这样呢？Pratt 在论文里认为人们沉溺于使用 BNF 语法及其变种，以及相关的自动机理论，以至于自动机理论以外的研究方向没有得到发展。

还有种解释是他的技术针对动态的函数编程语言最为有效，而应用到静态的过程语言会比较困难。在他的论文中，Pratt 用的是 LISP，他几乎毫不费力地从语素（token）流搭建出了解析树。

注 1：Pratt's paper is available at <http://portal.acm.org/citation.cfm?id=512931>; more information about Pratt himself can be found at <http://boole.stanford.edu/pratt.html>.

注 2：For a description of Floyd, see "Robert W Floyd, In Memoriam," Donald E. Knuth, <http://sigact.acm.org/floyd>.

不过以坚决抵制句法为荣的LISP社区并不怎么重视解析技术。自LISP诞生以来，人们多次试图为LISP加入ALGOL风格的丰富句法，包括

Pratt的CGOL

<http://zane.brouhaha.com/~healyzh/doc/cgol.doc.txt>

LISP 2

http://community.computerhistory.org/scc/projects/LISP/index.html#LISP_2_

MLISP

<ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/68/92/CS-TR-68-92.pdf>

Dylan

<http://www.opendylan.org>

Interlisp的Clisp

<http://community.computerhistory.org/scc/projects/LISP/interlisp/Teitelman-3IJCAI.pdf>

McCarthy最初的M-表达式

<http://www-formal.stanford.edu/jmc/history/lisp/lisp.html>

这些尝试都夭折了。函数编程社区发现程序和数据间对应关系的价值远远超过了句法丰富表达能力所带来的价值。但是，主流编程社区喜欢他们自己的句法，所以LISP从没被主流接受。Pratt的技术适用于动态语言，但动态编程语言社区从未用过Pratt的技术给出的句法。

JavaScript

这种局面随着JavaScript的问世而改变。JavaScript是一门动态函数语言，但从句法的角度看，很明显，它是C家族的一员。JavaScript是动态的语言，但它的程序员们钟爱句法。而且，JavaScript支持面向对象。Pratt在1973的论文中预见了面向对象，但缺乏具有足够表达能力的表示方法，而JavaScript拥有丰富的对象表示方法。因此，JavaScript可作为探索Pratt技术的理想语言。后面我会展示我们可以用JavaScript快速廉价地生成解析器。

在短短一章里我们没时间探讨整个JavaScript，而且我们也不想这么做，因为JavaScript语言十分混乱。不过它也有些出彩的东西，值得大家思考。我们将搭建一个处理简化版JavaScript的解析器，而且就用简化版JavaScript来实现该解析器。简化版JavaScript正是JavaScript出彩的部分，包括：

作为第一类对象的函数

函数是带词法作用域的 lambda 算子。

带原型继承的动态对象

对象没有类。我们可以通过普通的赋值向任何对象里加入任何新的成员。一个对象可以继承另外一个对象的成员。

对象字面量和数组字面量

创建新对象和新数组时，这套表示方法非常方便。JSON数据交换格式 (<http://www.JSON.org>) 的灵感便来自 JavaScript 的字面表示。

我们将利用 JavaScript 的原型特性来创建继承于符号的标志对象，以及从初始符号继承的其他符号。我们会用到一个叫 `Object` 的函数，该函数可以创建从已知对象继承成员的新对象。我们的实现还会用到词语分割器，把字符串分割为一组简单的语素对象。我们会在创建我们的解析树时遍历生成的语素数组。

符号表

我们用符号表驱动我们的解析器：

```
var symbol_table = {};
```

下面的 `original_symbol` 对象是其他所有符号对象的原型。它有报错的方法。这些方法多半会被更有用的方法覆盖：

```
var original_symbol = {
    //nud是Null Denotation的缩写。“优先级”一节会解释。
    nud: function () {
        this.error("Undefined.");
    },
    //led是Left Denotation的缩写。“优先级”一节后会解释。
    led: function (left) {
        this.error("Missing operator.");
    }
};
```

让我们来定义一个用于定义符号的函数。它接受一个符号 `id` 和一个可选的绑定权值，绑定权值的默认值为 0。该函数返回对应传入 `id` 的符号对象。如果 `symbol_table` 中已经存在该 `id` 的符号对象，函数返回该对象。否则，函数会创建一个从 `original_symbol` 继承的新符号对象，并将其放进符号表，同时返回这个对象。新创建的符号对象最终包含它的 `id`，它的值，左绑定的权值以及它从 `original_symbol` 继承的东西：

```
var symbol = function (id, bp) {
```

```

var s = symbol_table[id];
bp = bp || 0;
if (s) {
    if (bp >= s.lbp) {
        s.lbp = bp;
    }
} else {
    s = object(original_symbol);
    s.id = s.value = id;
    s.lbp = bp;
    symbol_table[id] = s;
}
return z;
);

```

下面的符号是流行的分割符和终结符。

```

symbol(":");
symbol(";");
symbol(",");
symbol(")");
symbol("]");
symbol("}");
symbol("else");

```

符号 (end) 表示再没有多余符号。符号 (name) 是新名字的原型，比如变量名的原型。为了避免冲突，它们的拼写特异：

```

symbol("(end)");
symbol("(name)");

```

符号 (literal) 是所有字符串字面量和数字字面量的原型：

```

var itself = function () {
    return this;
};
symbol("(literal)").nud = itself;

```

符号 this 是个专用变量。在调用方法时，它指向调用该方法的对象的引用：

```

symbol("this").nud = function () {
    scope.reserve(this);
    this.arity = "this";
    return this;
};

```

语素

我们假设源文本已经被转换为一组简单的语素对象 (tokens)。每个对象包含一个表征

对象类型的成员 `type`, 和一个表示对象值的成员 `value`。成员 `type` 是一段字符串 ("name", "string", "number", "operator"), 而成员 `value` 是一段字符串或者一个数。

变量 `token` 总是指向当前的语素:

```
var token;
```

函数 `advance` 创建一个新的语素对象, 并把它赋给变量 `token`。该函数还接受一项可选参数, `id`。如果 `id` 被传入, 函数会用前一语素的 `id` 与之对照检查。新语素对象的原型是当前词法域里的 `name` 语素, 或者是符号表里的符号。新语素的元数是 "name" (名字), "literal" (字面量), 或者 "operator" (运算符)。当我们更了解该语素在程序中的角色后, 语素可以被改为 "binary" (二元)、"unary" (一元) 或者 "statement" (语句)。

```
var advance = function (id) {
    var a, o, t, v;
    if (id && token.id !== id) {
        token.error("Expected '" + id + "'.");
    }
    if (token_nr >= tokens.length) {
        token = symbol_table["(end)"];
        return;
    }
    t = tokens[token_nr];
    token_nr += 1;
    v = t.value;
    a = t.type;
    if (a === "name") {
        o = scope.find(v);
    } else if (a === "operator") {
        o = symbol_table[v];
        if (!o) {
            t.error("Unknown operator.");
        }
    } else if (a === "string" || a === number") {
        a = "literal";
        o = symbol_table["(literal)"];
    } else {
        t.error("Unexpected token.");
    }
    token = object(o);
    token.value = v;
    token.arity = a;
    return token;
};
```

优先级

语素是带方法的对象。这些方法允许语素确定优先级，匹配其他语素以及构造树（另一个更加浩大的项目是还检查类型，优化，和生成代码）。基本的优先级问题是这样的：在两个运算符中间给出一个运算对象，该运算对象应该和左边的运算符绑定，还是同右边的运算符绑定？因此，如果 A 和 B 是下列公式中的运算符：

$d \ A \ e \ B \ f$

运算对象 e 应该同 A 还是同 B 绑定？也就是说，我们是在讨论

$(d \ A \ e) \ B \ f$

还是

$d \ A \ (e \ B \ f)$

解析的复杂性最终归结为解决这类歧义。我们在这儿推导出的技术用到了成员中包含绑定权值（优先级等级）的语素对象，和名为 `nud`（null denotation，没有指向）和 `led`（left denotation，指向左边）的方法。函数 `nud` 不在乎靠左的语素，而方法 `led` 在乎。值对象（比如变量和字面量）和前置运算符用 `nud` 方法。中置运算符和后置运算符用 `led` 方法。一个语素可以同时具备 `nud` 和 `led` 方法。比如，`-` 既可以是前置运算符（负号），也可以是中置运算符（减号），所以对应 `-` 的语素既有 `nud` 也有 `led` 方法。

表达式

Pratt 技术的核心是表达式函数 `expression`。它的参数是一个右绑定权值。该权值决定了表达式函数一次处理多少个右边的语素。表达式函数调用这些语素的方法，并返回调用这些方法所得到的结果。

```
var expression = function (rbp) {
    var left;
    var t = token;
    advance();
    left = t.nud();
    while (rbp < token.lbp) {
        t = token;
        advance();
        left = t.led(left);
    }
    return left;
}
```

函数 expression 调用当前语素 token 的 nud 方法。该方法用于处理字面量、变量和前置运算符。之后，只要传入的右绑定权值小于下一个语素的左绑定权值，当前语素的 led 方法就会被调用。该方法用于处理中置和后置运算符。整个过程可以递归，因为 nud 方法和 led 方法也可以调用 expression 函数。

中置运算符

加号 “+” 是中置运算符，所以它包含把语素对象转换为树的 led 方法。树的两条分支为加号左边的运算对象和加号右边的运算对象。左边的运算对象被传入 led 方法，而右边的运算对象通过调用函数 expression 得到。

+ 的绑定权值是 60。绑定得更紧或者说有更高优先级的运算符也有更大的绑定权值。在把语素流转换为解析树的过程中，我们把运算符语素用作运算符节点的容器：

```
symbol("+", 60).led = function (left) {
    this.first = left;
    this.second = expression(60);
    this.arity = "binary";
    return this;
};
```

当定义 *（乘号）符号时，我们可以看到，id 和绑定权值是 * 和 + 唯一的区别。乘号的绑定权值更高，因为它有更高的优先级。

```
symbol("*", 70).led = function (left) {
    this.first = left;
    this.second = expression(70);
    this.arity = "binary";
    return this;
};
```

不是所有的中置运算符都这么相似，但很多都是，所以我们可以定义用于规定中置运算符的 infix 函数，从而让我们工作得更顺手一点。这个 infix 函数接受一个 id，一个绑定权值，和一个可选的 led 函数。如果传入的参数里没有 led 函数，infix 函数会提供一个大多数情况下适用的默认 led 函数。

```
var infix = function (id, bp, led) {
    var s = symbol(id, bp);
    s.led = led || function (left) {
        this.first = left;
        this.second = expression(bp);
        this.arity = "binary";
        return this;
    };
    return s;
};
```

```
    }
}
```

这样做可让我们用更加接近声明式编程的方式来规定运算符：

```
infix("+", 60);
infix("-", 60);
infix("*", 70);
infix("/", 70);
```

字符串 `==` 是 JavaScript 里判别严格相等的运算符。

```
infix("===", 50);
infix("!==", 50);
infix("<", 50);
infix("<=", 50);
infix(">", 50);
infix(">=", 50);
```

三元运算符接受被`?`和`:`分割的三个表达式。它不是普通的中置运算符，所以我们给出了它自己的 `led` 函数：

```
infix("?", 20, function (left) {
  this.first = left;
  this.second = expression(0);
  advance(":");
  this.third = expression(0);
  this.arity = "ternary";
  return this;
});
```

点运算符`.`用于选择对象成员。它右边的语素必须是名字，不过这个语素会被当成字面量：

```
infix(".", 90, function (left) {
  this.first = left;
  if (token.arity !== "name") {
    token.error("Expected a property name.");
  }
  token.arity = "literal";
  this.second = token;
  this.arity = "binary";
  advance();
  return this;
});
```

运算符`[`用于从对象或数组中动态的选出成员。该运算符右边的表达式必须用`]`收尾：

```
infix("[", 90, function (left) {
  this.first = left;
  this.second = expression(0);
```

```

        this.arity = "binary";
        advance("]");
        return this;
    });
}

```

上述中置运算符是左结合的。我们也可以通过减小右绑定权值来创建右结合的运算符，比如短路逻辑运算符：

```

var infixr = function (id, bp, led) {
    var s = symbol(id, bp);
    s.led = led || function (left) {
        this.first = left;
        this.second = expression(bp - 1);
        this.arity = "binary";
        return this;
    };
    return s;
}

```

运算符`&&`在第一个运算对象为假时返回它，不然则返回第二个运算对象。运算符`||`在第一个运算对象为真时返回它，不然则返回第二个运算对象：

```

infixr("&&", 40);
infixr("||", 40);

```

前置运算符

前置运算符的处理和前面相似。前置运算符是右结合的，它没有左绑定权值，因为它不向左绑定。前置运算符有时也可以是保留字（保留字将在本章后面“作用域”一节讨论）：

```

var prefix = function (id, nud) {
    var s = symbol(id);
    s.nud = nud || function () {
        scope.reserve(this);
        this.first = expression(80);
        this.arity = "unary";
        return this;
    };
    return s;
}
prefix("-");
prefix("!");
prefix("typeof");

```

左括号“`(`”的`nud`函数会调用`advance(")")`来匹配配对的右括号“`)`”语素。左括号“`(`”不会成为解析树的一部分，因为它的`nud`函数返回如下表达式：

```

prefix("(", function () {
    var e = expression(0);

```

```
    advance(")");
    return e;
});
```

赋值运算符

用 infixr 来定义我们的赋值运算符也行，不过我们还想做两件另外的事。所以我们创建一个专门的 assignment 函数。这个函数会检查左边的运算对象，确保它是真的左值 (lvalue)。我们还会设置一个 assignment 标记，以便后来可以快速判定赋值语句。

```
var assignment = function (id) {
  return infixr(id, 10, function (left) {
    if !(left.id !== ".." && left.id !== "[" &&
       left.arity !== "name") {
      left.error("Bad lvalue.");
    }
    this.first = left;
    this.second = expression(9);
    this.assignment = true;
    this.arity = "binary";
    return this;
  });
};

assignment("=");
assignment("+=");
assignment("-=");
```

注意这段程序多少有点像继承模式： assignment 函数返回调用 infixr 的结果，而 infixr 返回调用 symbol 的结果。

常数

函数 constant 在被解析的语言里置入常数。函数 nud 把名字语素变换为字面量语素：

```
var constant = function (s, v) {
  var x = symbol(s);
  x.nud = function () {
    scope.reserve(this);
    this.value = symbol_table[this.id].value;
    this.arity = "literal";
    return this;
  };
  x.value = v;
  return x;
};

constant("true", true);
constant("false", false);
constant("null", null);
```

```
constant("pi", 3.141592653589793);
```

Scope

我们用诸如 `infix` 和 `infix` 的函数来定义语言里用到的符号。大多数语言都有用来定义新符号的某种标识，比如变量名。当我们在非常简单的语言里遇到新词时，我们也许会定义这个新词，并把它放到符号表里。在更复杂的语言里，我们就需要用作用域的概念了，以便程序员控制变量的生命周期和可见度。

作用域是指变量被定义和可以读写的程序区域。作用域可以嵌套在其他作用域内。作用域内的变量在作用域外不可见。

我们把当前作用域保留在 `scope` 变量里：

```
var scope;
```

对象 `original_scope` 是所有作用域对象的原型。它包含用来在作用域内定义新变量的 `define` 方法。这个方法把名字语素转换为变量语素。如果作用域内已经定义了转换成的变量，或者被转换的名字已经被用作保留字，`define` 方法会生成报错对象 `error`：

```
var original_scope = {
    define: function (n) {
        var t = this.def[n.value];
        if (typeof t === "object") {
            n.error(t.reserved ?
                "Already reserved." :
                "Already defined.");
        }
        this.def[n.value] = n;
        n.reserved = false;
        n.nud      = itself;
        n.led      = null;
        n.std      = null;
        n.lbp      = 0;
        n.scope    = scope;
        return n;
    },
}
```

方法 `find` 用于找出名字的定义。它从当前作用域开始，必要时可以沿着父作用域链一路向上，直到找到合适的符号表。如果 `find` 方法找不到名字的定义，会返回 `symbol_table["(name)"]`。

```
find: function (n) {
    var e = this;
    while (true) {
        var o = e.def[n];
```

```
        if (o) {
            return o;
        }
        e = e.parent;
        if (!e) {
            return symbol_table[
                symbol_table.hasOwnProperty(n) ? n : "(name)"];
        }
    }
},
```

方法 pop 关闭作用域：

```
pop: function () {
    scope = this.parent;
},
}];
```

方法 `reserve` 用来指明一个名字在当前作用域下已被用作保留字：

```
        reserve: function (n) {
            if (n.arity !== "name" || n.reserved) {
                return;
            }
            var t = this.def[n.value];
            if (t) {
                if (t.reserved) {
                    return;
                }
                if (t.arity === "name") {
                    n.error("Already defined.");
                }
            }
            this.def[n.value] = n;
            n.reserved = true;
        }
    }
}
```

我们需要一条用于保留字的规则。某些语言里，用于程序结构的字（比如 if）是保留字，不能当作变量名。解析器的灵活性允许我们采纳更有用的规则。比如，可以说任何函数里，任何名字都可以用作结构字，或者用作变量。这对语言设计者来说更有好处，因为向语言里加入新的结构字不会破坏已有的程序，而且这对程序员来说也更有利，因为他们不会被名字用法的毫不相干的限制所妨碍。

每当我们想给一个新的函数或者程序块建立新的作用域时，就调用 `new_scope` 函数。这个函数创建一个以原生的作用域（`original_scope`）为原型的对象实例：

```
var new_scope = function () {
    var s = scope;
    scope = object(original_scope);
```

```
    scope.def = {};
    scope.parent = s;
    return scope;
};
```

语句

Pratt最初的推导结果适用于所有函数语言。函数语言里一切都是表达式。大多数主流语言支持语句。我们通过在语素里加入另外一个方法 std (statement denotation, 语句指称) 来轻松处理语句。除了 std 用在语句开头以外，它和函数 nud一样。

函数 statement 一次解析一条语句。如果当前语素有 std 方法，保留该语素，并调用它的 std 函数。不然，我们假设一条表达式语句以分号 “;” 结尾。为了可靠性，我们不接受既非赋值也非函数调用的表达式语句：

```
var statement = function () {
    var n = token, v;
    if (n.std) {
        advance();
        scope.reserve(n);
        return n.std();
    }
    v = expression(0);
    if (!v.assignment && v.id !== "(") {
        v.error("Bad expression statement.");
    }
    advance(";");
    return v;
};
```

函数 statements 逐条解析语句，直到它看见表示程序块结束的 “(end)” 或者 “}”。它返回一条语句、一组语句或者（如果没有解析到任何语句的话）仅仅是 null：

```
var statements = function () {
    var a = [], s;
    while (true) {
        if (token.id === "}" || token.id === "(end)") {
            break;
        }
        s = statement();
        if (s) {
            a.push(s);
        }
    }
    return a.length === 0 ? null : a.length === 1 ? a[0] : a;
};
```

函数 stmt 用来将语句加入符号表。它的参数包括一个语句 id 和一个函数 std。

```

var stmt = function (s, f) {
    var x = symbol(s);
    x.std = f;
    return x;
};

```

程序块语句用一对花括号把一串语句组织起来，并赋予它们新的作用域：

```

stmt("{", function () {
    new_scope();
    var a = statements();
    advance("}");
    scope.pop();
    return a;
});

```

函数 block 解析程序块：

```

var block = function () {
    var t = token;
    advance("{");
    return t.std();
};

```

语句 var 在当前程序块里定义一个或多个变量。每个变量名后可带 = 和一条表达式：

```

stmt("var", function () {
    var a = [], n, t;
    while (true) {
        n = token;
        if (n.arity !== "name") {
            n.error("Expected a new variable name.");
        }
        scope.define(n);
        advance();
        if (token.id === "=") {
            t = token;
            advance("=");
            t.first = n;
            t.second = expression(0);
            t.arity = "binary";
            a.push(t);
        }
        if (token.id !== ",") {
            break;
        }
        advance(",");
    }
    advance(";");
    return a.length === 0 ? null : a.length === 1 ? a[0] : a;
});

```

语句 while 定义循环。它包含一个括号内的表达式和一个程序块：

```

stmt("while", function () {
    advance("(");
    this.first = expression(0);
    advance(")");
    this.second = block();
    if (this.arity === "statement") {
        this.arity = "block";
        return this;
    }
});

```

语句 `if` 用于条件执行。如果我们在 `if` 的程序块后看到 `else`, 我们继续解析下一个程序块, 或者下一个 `if` 语句:

```

stmt("if", function () {
    advance("(");
    this.first = expression(0);
    advance(")");
    this.second = block();
    if (token.id === "else") {
        scope.reserve(token);
        advance("else");
        this.third = token.id === "if" ? statement() : block();
    }
    this.arity = "statement";
    return this;
});

```

语句 `break` 用来跳出循环。我们确保下一个符号是右花括号:

```

stmt("break", function () {
    advance(";");
    if (token.id !== ")") {
        token.error("Unreachable statement.");
    }
    this.arity = "statement";
    return this;
});

```

语句 `return` 用于从函数返回。它可以返回一则表达式:

```

stmt("return", function () {
    if (token.id !== ";") {
        this.first = expression(0);
    }
    advance(";");
    if (token.id !== ")") {
        token.error("Unreachable statement.");
    }
    this.arity = "statement";
    return this;
});

```

函数

函数是可以执行的对象值。函数可以有名字（用来递归调用自己），一组放在括号内的参数名和函数体。函数体是包含在花括号内的一组语句。函数有自己的作用域：

```
prefix("function", function () {
    var a = [];
    scope = new_scope();
    if (token.arity === "name") {
        scope.define(token);
        this.name = token.value;
        advance();
    }
    advance("(");
    if (token.id !== ")") {
        while (true) {
            if (token.arity !== "name") {
                token.error("Expected a parameter name.");
            }
            scope.define(token);
            a.push(token);
            advance();
            if (token.id !== ",") {
                break;
            }
            advance(",");
        }
    }
    this.first = a;
    advance(")");
    advance("{");
    this.second = statements();
    advance("}");
    this.arity = "function";
    scope.pop();
    return this;
});
```

函数通过“(”运算符调用。它可以接受零个或者多个用逗号分隔的参数。我们通过考察左运算对象来判别不能当作函数值的表达式：

```
infix("(", 90, function (left) {
    var a = [];
    this.first = left;
    this.second = a;
    this.arity = "binary";
    if ((left.arity !== "unary" ||
        left.id !== "function") &&
        left.arity !== "name" &&
```

```

        (left.arity !== "binary" ||
         (left.id !== "." &&
          left.id !== "(" &&
          left.id !== "["))) {
      left.error("Expected a variable name.");
    }
    if (token.id === ")") {
      while (true) {
        a.push(expression(0));
        if (token.id === ",") {
          break;
        }
        advance(",");
      }
    }
    advance(")");
    return this;
}
);

```

数组和对象字面量

数组字面量是包含零个或多个表达式的一组方括号。表达式用逗号分隔。每个表达式求解后的值收集起来，得出一个新的数组：

```

prefix("[", function () {
  var a = [];
  if (token.id === "]") {
    while (true) {
      a.push(expression(0));
      if (token.id === ",") {
        break;
      }
      advance(",");
    }
  }
  advance("]");
  this.first = a;
  this.arity = "unary";
  return this;
});

```

对象字面量是包含零个或多个键值对的一组花括号。键值对是用冒号分隔的键/表达式对：键要么是字面量，要么是被当作字面量的名字：

```

prefix("{", function () {
  var a = [];
  if (token.id === "}") {
    while (true) {
      var n = token;
      if (n.arity !== "name" && n.arity !== "literal") {
        token.error("Bad key.");
      }
      a.push(n);
      if (token.id === ",") {
        advance(",");
      } else if (token.id === "}") {
        advance("}");
        return this;
      }
    }
  }
});

```

```

        }
        advance();
        advance(":");
        var v = expression(0);
        v.key = n.value;
        a.push(v);
        if (token.id !== ",") {
            break;
        }
        advance(",");
    }
}
advance("}");
this.first = a;
this.arity = "unary";
return this;
});

```

要做和要思考的事

本章展示的简单解析器容易扩展。生成的解析树可以传给代码生成器，也可以传给解释器。生成树的计算量极小。而且我们也看到了，编程生成解析树也不费什么功夫。

我们可以让 `infix` 函数接受一个操作码参数，用来辅助代码生成。我们也可以让它接受额外的方法参数。这些方法可以用于常数展开和代码生成。

我们可以加上其他语句，比如说 `for`、`switch` 和 `try`。我们还可以加上语句标签。我们可以加入更多的错误检查和错误恢复。我们可以加入很多的运算符。我们可以加入类型规则和推断。

可以让我们的语言具备扩展能力，使得程序员加入新运算符和新语句就跟加入新的变量一样容易。

这章描述的解析器有演示程序。你可以到下面的网址体验：<http://javascript.crockford.com/tdop/index.html>。

JSLint 也用到了这里的解析技术：<http://JSLint.com>。

寻求快速的种群计数

Henry S. Warren, Jr.

种群计数 (Population Count) 或横列和 (Sideways Sums) 是计算机中基本且出奇简单的算法，用于统计单位计算机字长所包含的 1 位的数量。种群计数函数广泛应用于从简单到极其复杂的各种领域。例如，如果用位串 (bit string) 来表示集合，种群计数就可以计算出集合的大小。种群计数同样可以用来生成符合二项分布的随机整数。我们将会在本章节的末尾部分详细讨论上述应用及其他相关应用。

尽管种群计数操作的使用并不是十分简单的事情，但是许多计算机（通常是超级计算机时代的）都有相应的指令支持。这些计算机包括：Ferranti Mark I (1951), IBM Stretch 计算机 (1960), CDC 6600 (1964), 俄制 BESM-6 (1967), Cray 1 (1976), Sun SPARCv9 (1994) 以及 IBM Power 5 (2004)。

本章讨论如何在不支持这项指令的计算机上实现种群计数的计算功能，且这些计算机仅支持 RISC 或 CISC 计算机通常所支持的指令：移位 (shift)，加 (add)，与 (and)，取数 (load)，条件分支 (conditional branch) 等等。为了方便举例说明，我们不妨假设计算机的字长是 32 位，但这里所讨论的大部分技术很容易适用于其他大小的计算机字长。

本章节解决了种群计数的 2 种情况：一是单个计算机字中 1 位的计数；二是大量计算机

字（可能连续存放于某个数组中）中 1 位的总计数。尽管我们针对每种情况都给出了显而易见的解决方案，即使是经过深思熟虑的，只要发挥你的想象力去思索，就能找出与我们的算法完全不同的更优的算法。第一种方案运用了分而治之策略；第二种方案运用了大部分计算机逻辑设计师所熟悉的特定逻辑电路，程序员可能不太熟悉。

基本方法

对计算机字 x 中的 1 位计数，程序员不假思索所想到的方法就是下面 C 代码中的方案。这里 x 是无符号整数，因此右移最高位填 0。

```
pop = 0;
for (i = 0; i < 32; i++) {
    if (x & 1) pop = pop + 1;
    x = x >> 1;
}
```

上述循环体代码在典型的 RISC 计算机上编译成约 7 条指令，其中 2 条属于条件分支（其中一条条件分支负责循环控制）。7 条指令循环执行的次数是 32 次，但其中某条指令绕过了大约一半的执行时间（根据我们的假设），所以循环体总共大约执行了 $32 \times 6.5 = 208$ 条指令。

程序员或许很快就会意识到上述代码很容易改进。首先，在大多数计算机上，从 31 倒计数到 0 比从 0 累加到 31 要更高效，因为前者可以少 1 条比较（compare）指令。继续改进，为什么要对所有位都逐个计数呢？当 x 为 0 的时候，循环就可以结束。如果 x 的高位是 0 的话，可以消除掉若干次迭代。另外一个优化是：去除 if 条件判断，直接将 x 的最右位累加到计数值上。最终得到如下代码：

```
pop = 0;
while (x) {
    pop = pop + (x & 1);
    x = x >> 1;
}
```

上述循环体代码中仅包含 4 条或 5 条 RISC 指令（取决于 x 与 0 是否需要进行比较），且只有一个分支（我们假设编译器重排循环体后，条件分支位于底部）。因此，代码最多耗费 128~160 条指令。最多情况出现在 x 以 1 开始时，但如果 x 有很多前缀 0 的话，所耗费的指令数会少很多。

某些读者可能已经回忆起简单的表达式 $x \& (x - 1)$ 的作用是将 x 的最低 1 位清零；或者如果 $x = 0$ 的话，整个表达式值为 0。因此，要计算 x 中所包含的 1 位数目，可以从低到高逐次将 x 的最低 1 位清零直到 $x = 0$ ，过程中记录下清零的次数。最终得到如下代码：

```

pop = 0;
while (x) {
    pop = pop + 1;
    x = x & (x - 1);
}

```

和前面那段代码一样，上面这段代码中的循环体也需要花费 4 或者 5 条指令，但是它循环的次数等于 x 中包含的 1 位的数目。这的确是一项很大的改善。

如果 1 位数目的期望值非常大，可以采用一种补充方法：不断地用表达式 $x = x | (x+1)$ 将 x 最右边的 0 位翻转成 1 位，一直到所有的比特位都变成 1 为止 (x 值为 -1)。记录迭代执行的次数 n ， $32-n$ 就是最终要求的值（其他变通的做法包括：对原来的数 x 求反码；或者将 n 初始化成 32，然后倒计数）。

本系列中的第一个程序是相当笨拙的做法，其他程序则可能在效率、简洁性和实用的灵巧性方面很具有很吸引眼球的价值。第一个程序通过循环展开，运行速度可以得到本质的提高；但另外两个程序即使展开，其性能的提高也非常有限。

大家也可以采用查表法，每次将 x 中的 1 个字节转换成该字节中所包含的 1 位的数目。代码非常简短，且在很多机器上的运行速度都相当快（在不支持索引取数（indexed loads）的初级 RISC 计算机上，大约是 17 条指令）。下面的代码中， $\text{table}[i]$ 中存储的是 i 中的 1 位数目， i 的取值是 0~255：

```

static char table[256] = {0, 1, 1, 2, 1, 2, 2, 3, ..., 8};
pop = table[x & 0xFF] + table[(x >> 8) & 0xFF] +
      table[(x >> 16) & 0xFF] + table[x >> 24];

```

分治法

另一种有趣且实用的统计种群数目的方法基于“分而治之”策略。利用如下推论来设计算法：“假设我有计算 16 位数量级的种群数目的方法，就可以将这种方法分别作用于 32 位字的左右两半部分，将分别得到的结果相加，最终得到 32 位字的种群数目。”如果基本算法必须按顺序分别作用在左右两个半字上，其所花费的时间就会与分析的位数成正比。上述策略显得毫无优势，因为它需要耗费 $16k+16k=32k$ 时间单位（ k 是比例常数），外加一条加法指令的执行时间。但是如果我们可以用某种方法并行地在左右两个半字上同时执行操作，那么最终性能将会从 $32k$ 跳跃到 $16k+1$ 。

为了高效地计算两个 16 位数量的种群数目，我们需要一种可以并行地在四个 8 位数量上进行计算的方法。继续往下推导，最终我们需要一种可以并行地在十六个 2 位数量上进行计算的方法。

接下来要描述的算法并不需要运行在独立的多处理器上，也不需要像 SIMD（注1）这类只在特定计算机上才存在的特殊指令的支持。它仅仅使用了常规单处理器的 RISC 或 CISC 计算机上所支持的工具。

设计方案如图 10-1 所示。

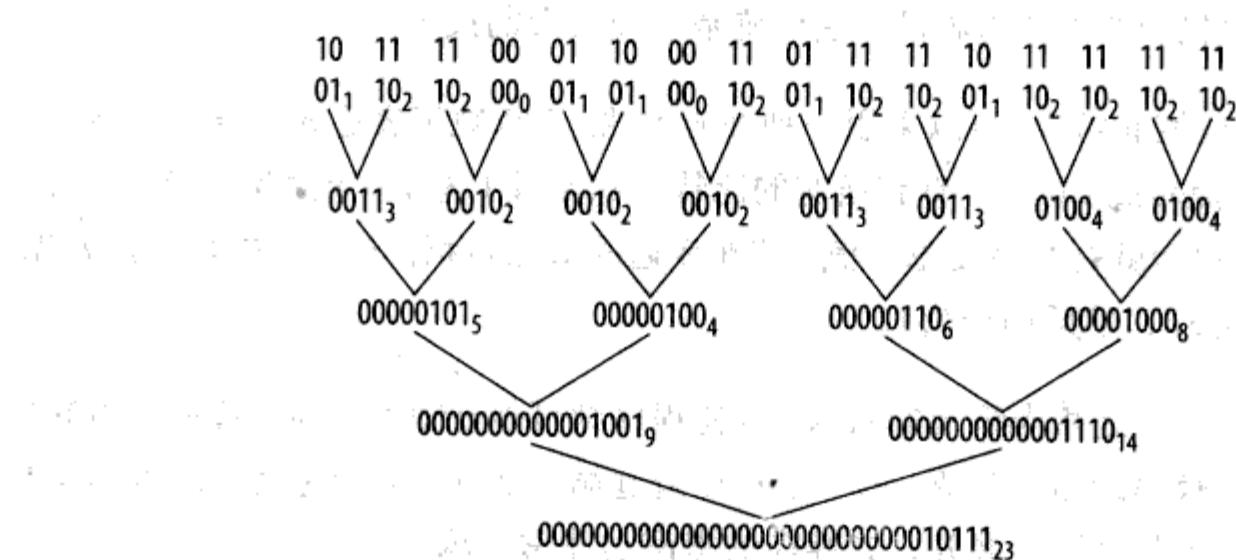


图 10-1：1 位计数—分治策略

图 10-1 中的第一行是需要进行 1 位种群计数的字 x 。第二行的每个 2 位字段包含了上一行所对应的 2 位字段中 1 位的数目。字段的下标是 2 位字段的十进制数值。第三行的每个 4 位字段包含了第二行相邻的两个 2 位字段的计数值之和，其下标也表示十进制值，等等。最后一行包含了 x 中 1 位的数目。算法执行了 $\log_2(32) = 5$ 步，每步都包含有用于对相邻字段求和的移位 (shifting) 和屏蔽 (masking) 指令。

图 10-1 中所描述的方法可以实现成如下 C 代码：

```
x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
x = (x & 0x0000FFFF) + ((x >>16) & 0x0000FFFF);
```

(C语言中，以0x开头的常量表示十六进制值。) 第一行代码之所以采用($x >> 1$) & 0x55555555而不是更为自然的($x \& 0xFFFFFFFF$) $>> 1$ ，是因为这样可以避免在一个寄存器中产生两个巨大的常量。如果机器缺少与非(and not)指令，会多耗费一条指令。类似的解释适用于其他行的代码。

注1：单指令多数据源（SIMD）指令可以并行地操作计算机字中的多个字段（如字节或者半字）。例如，一个8位的 SIMD *add* 指令可能只会将两个字的对应字节进行相加，却忽略了两个字节相加所产生的进位。

很明显，最后一个与 (and) 操作是不需要的，因为 $x >> 16$ 时所得到数值的高 16 位全是 0，与 (and) 操作不会修改 $x >> 16$ 的数值。如果字段求和后不存在进位到相邻的字段的危险，可以省略其他相关与 (and) 操作。第一行代码有一种编码方式可以少用一条指令。上述建议所作出的简化请参考示例 10-1，这个例子只执行了 21 条指令，并且没有分支 (branches) 和内存引用 (memory references)。

示例 10-1：字的 1 位计数

```
int pop(unsigned x) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    x = x + (x >> 8);
    x = x + (x >> 16);
    return x & 0x0000003F;
}
```

第一次对 x 的赋值是基于以下公式的前 2 项：

$$\text{pop}(x) = x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \left\lfloor \frac{x}{8} \right\rfloor - \dots - \left\lfloor \frac{x}{2^{31}} \right\rfloor$$

这里的 x 必须满足 $x \geq 0$ 。假设 x 是一个无符号的整数，通过 31 个逐次右移 1 位的立即数和 31 次减法，就可以实现上述等式。示例 10-1 中的程序并行地将本公式的前两项运用到 x 的每个 2 位字段上。等式留给读者自己证明。

遗憾的是，示例 10-1 中的代码已经丧失了原有代码的整齐和优雅。导致的一个后果就是代码扩展到 64 位机器时，不再是直接明了的事情。但我们还是无法抗拒所有这些可以节约指令的机会！

分治法是一项非常重要的技术，应该随时放在每个程序员技巧百宝箱的最上面。分治法对计算机逻辑电路设计师同样适用。分治法的其他应用则包括著名的二分查找法、快速排序和翻转字比特位。

其他方法

HAKMEN 备忘录（注 2）第 169 条描述了对字 x 中的 1 位进行计数的算法。这种算法将上节公式中的前三项分别作用到 x 的每个 3 位字段，生成 3 位字段所组成的字，其中的每

注 2：Michael Beeler, R. William Gosper 和 Richard Schroeppel 于 1972 年 2 月在 MIT 人工智能实验室的《AIM》第 239 期发表了《HAKMEN》。现在可从 <http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html> 获得，非常感谢 Henry Baker。

个字段包含了 3 位字段中 1 位的数目。接下来，将相邻的两个 3 位字段相加，得到 6 位字段的 1 位数目和。最后，通过将字 x 模 63 所得到的值就是所有 6 位字段的位数总和。尽管算法原本是为 36 位字长的计算机设计的，但是很容易适用于 32 位字。算法的 C 代码描述如下（长串的常量是 8 进制的）：

```
int pop(unsigned x) {
    unsigned n;

    n = (x >> 1) & 033333333333;           // 对每个
    x = x - n;                                // 3 位字段
    n = (n >> 1) & 033333333333;           // 进行 1 位计数。
    x = x - n;
    x = (x + (x >> 3)) & 030707070707;   // 6 位字段的 1 位数目和。
    return x%63;                               // 将所有 6 位字段和相加。
}
```

代码的最后一行使用了无符号取模函数。（如果字长是 3 的倍数，最后一行既可以是有符号的，也可以是无符号的。）如果把字 x 当成以 64 为底的整数时，显然取模函数是对 6 位字段求和。对于 $b \geq 3$ ，以 b 为底的整数模 $b-1$ ，与整数中数字的和模 b 是同余的，这个余数显然是小于 $b-1$ 的。在这种情况下，由于数字的和一定小于等于 32，因此 $\text{mod}(x, 63)$ 一定等于 x 的数字和，即原先 x 中的 1 位数目。

这种算法在 DEC PDP-10 计算机上只需要 10 条指令，因为这种计算机上支持这种指令：通过直接引用内存全字得到的数值，将其作为余数计算的第二个操作数使用。在一台初级 RISC 计算机上，大约需要 15 条指令，并且假设这台计算机提供无符号模指令（但是不能直接引用全字立即数或者内存操作数）。这条指令可能运行得并不是很快，因为除法几乎总是一种较慢的操作。另外，这种算法最大适用于 62 位字长，但只通过简单地扩展常数就想适用于 64 位字长是不可行的。

另一个更为令人惊奇的算法是把 x 逐次循环左移 1 个位置，总共 31 次，然后求 32 项的和（注 3）。这个和就是 $\text{pop}(x)$ 的负值！即：

$$\text{pop}(x) = -\sum_{i=0}^{31} (x \text{ } \overset{\text{rot}}{\ll} i)$$

这里的加法通过对字长取模完成，最终的和被解释为一个补码 (two's-complement) 整数。不过，这种算法只是比较新奇而已，在大多数计算机上并没有什么实用价值，因为循环要执行 31 次，从而需要 63 条指令外加循环控制的开销。公式的工作原理留给读者去弄清楚。

注 3：Mike Morton 于 1990 年 12 月在《计算机语言》第 7 卷，第 12 册，第 45~55 页发表的《Quibbles & Bits》。

两个字种群计数的和与差

计算 $\text{pop}(x) + \text{pop}(y)$ 的值（如果你的计算机不具备种群计数指令），有一种方法可以节省部分时间：首先将示例 10-1 中代码的前两行分别作用于 x 和 y ，然后将剩下的最后三步作用于 x 和 y 相加的和。示例 10-1 中代码的前两行执行后， x 和 y 将由 8 个 4 位字段组成，这种思想同样适用于减法。使用如下等式，可以计算 $\text{pop}(x) - \text{pop}(y)$ ：（注 4）

$$\begin{aligned}\text{pop}(x) - \text{pop}(y) &= \text{pop}(x) - (32 - \text{pop}(\bar{y})) \\ &= \text{pop}(x) + \text{pop}(\bar{y}) - 32\end{aligned}$$

然后，使用前面所描述的技术来计算 $\text{pop}(x) + \text{pop}(\bar{y})$ 。具体代码参见示例 10-2，它使用了 32 条指令。相比之下，示例 10-1 中代码的两次运用外加一次减法会产生 43 条指令。

示例 10-2：计算 $\text{pop}(x) - \text{pop}(\bar{y})$

```
int popDiff(unsigned x, unsigned y) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    y = ~y;
    y = y - ((y >> 1) & 0x55555555);
    y = (y & 0x33333333) + ((y >> 2) & 0x33333333);
    x = x + y;
    x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
    x = x + (x >> 8);
    x = x + (x >> 16);
    return (x & 0x0000007F) - 32;
}
```

两个字的种群计数比较

有时我们只想知道两个字中哪一个的种群计数值更大，而不考虑实际的计数值是多少。不求出两个字的种群计数值，可以决定出哪一个字更大吗？如示例 10-2 中所示那样，先计算两个字种群计数值的差，然后将结果与 0 比较是一种可行的方法。但是，如果期望的种群计数值较低，或者两个字中的某些特殊位有很强的关联性，我们则有一种更优越的方法。

基本思想是对每个字中的单个比特位轮流清零，直到其中某个字的比特位全为 0；那么，剩下的非零种群计数值的字就是较大者。如果首先清除每个字相同位置的 1 位，整个算法过程在最坏和平均情况下都可以运行得更快。具体代码参见示例 10-3。程序返回负值则表示 $\text{pop}(x) < \text{pop}(y)$ ，返回 0 则表示 $\text{pop}(x) = \text{pop}(y)$ ，返回正值(1)则表示 $\text{pop}(x) > \text{pop}(y)$ 。

注 4： \bar{y} 表示 y 的反码 (one's-complement)，在 C 语言中表示为 $\sim y$ 。

示例 10-3：比较 pop(x) 与 pop(y)

```
int popCmpr(unsigned xp, unsigned yp) {
    unsigned x, y;
    x = xp & ~yp;           // 清除同
    y = yp & ~xp;           // 是 1 的比特位。
    while (1) {
        if (x == 0) return y | -y;
        if (y == 0) return 1;
        x = x & (x - 1);     // 每个字
        y = y & (y - 1);     // 清除一个 1 位。
    }
}
```

在对每个 32 位字中的共同 1 位清零后，两个字中所有 1 位数目加在一起的最大可能值是 32。因此，由于较小 1 位数目的字最多只有 16 位，导致示例 10-3 中的循环最多执行 16 次。最坏情况下，初级 RISC 计算机需要耗费 119 条指令 ($16 \times 7 + 7$)。对共同的 1 位清零后，一个使用均匀分布的 32 位随机数的模拟表明：较小种群计数值的字其平均值大约是 6.186。当输入为 32 位的随机数时，程序的平均执行时间大约是 50 条指令，性能不如示例 10-2 中的程序。要想战胜示例 10-2 中的程序，在对共同的 1 位清零后，*x* 或者 *y* 中的 1 位数目必须小于等于 3。

数组中的 1 位种群计数

在没有种群计数指令的情况下，计算一个全字数组（向量）的 1 位数目的最简单方法是：对这个数组中的每一个字分别运用类似示例 10-1 中的程序，然后将结果相加。我们称这种方法为幼稚的方法。忽略循环控制、常量产生、数组取数等情况，程序平均每字需要用 16 条指令。其中示例 10-1 中的代码占据了 15 条，外加 1 条加法指令。我们假设过程（procedure）内联展开，屏蔽（masks）从循环外面载入，计算机有充足的寄存器来存储计算过程中所使用的所有数值。

另外一种方法是先对数组中的三字组分别执行示例 10-1 代码中的前两行，然后再把得到的三部分结果相加。因为在每一个 4 位字段上，每一部分结果的最大值是 4；那么，三部分结果相加的和在每一个 4 位字段上的最大值是 12，不会溢出到左边的相邻字段。这种思想同样适用于 8 位和 16 位字段。一台初级的 RISC 计算机，按照这种方法编写代码，其编译后得到的可执行程序，指令总数要比前述的幼稚方法减少大约 20%。大部分节约的时间，是通过取消原有的额外内务指令（housekeeping instructions）获得的。我们不会停滞于目前这种方法，还有更好的方法来处理这些事情。

一种更优的方法似乎是 Robert Harley 和 David Seal 在 1996 年左右发明的（注 5）。它是一种基于进位保留加法器（CSA）或者 3-2 线编码器（3:2 compressor）的电路。简单一点说，CSA 就是一组独立的全加法器（注 6），经常在二进制乘法电路中使用。

下面是使用布尔代数符号（并置表示与， $+$ 表示或， \oplus 表示异或）所表示的每个全加法器的逻辑：

$$\begin{aligned} h &\leftarrow ab + ac + bc = ab + (a + b)c = ab + (a \oplus b)c \\ l &\leftarrow (a \oplus b) \oplus c \end{aligned}$$

其中 a, b, c 都是 1 位的输入， l 是低位输出（和）， h 是高位输出（进位）。第一行的 $a+b$ 之所以改成 $a \oplus b$ 是经过论证的，因为当 a 和 b 都是 1 的时候， ab 项使得整个表达式的值等于 1（此时可以忽略其他项的值）。首先将 $a \oplus b$ 赋给一个临时变量，全加法器逻辑就可以通过五条逻辑指令来实现求值（evaluated），每条指令并行地操作 32 位数据（32 位机器上）。我们把这 5 条指令称为 CSA (h, l, a, b, c) 。实际上这是一个“宏”，其中 h 和 l 是输出。

其中一种使用 CSA 的方法就是按照三个一组的方法处理数组 A 中的元素：先将每组中的三个字减为两个字，然后对这两个字应用种群计数操作。这两个种群计数值会在循环体中相加。循环执行结束后，数组总的种群计数值等于 CSA 高位输出的种群计数累加值的两倍加上 CSA 低位输出的种群计数累加值。

下面的序列描述了 16 位字的处理过程：

$$\begin{array}{r} a = 0110\ 1001\ 1110\ 0101\ 9 \\ b = 1000\ 1000\ 0100\ 0111\ 6 \\ c = 1100\ 1010\ 0011\ 0101\ 8 \\ \hline l = 0010\ 1011\ 1001\ 0111\ 9 \\ h = 1100\ 1000\ 0110\ 0101\ 7*2 = 14 \end{array}$$

观察每一列中的 (h, l) 数对，依照这个顺序组合成的一对数字是一个两位二进制数，其

注 5：David Seal 于 1997 年 5 月 13 日在新闻组 *comp.arch.arithmetic* 发表的。Robert Harley 是本文作者所知道的第一个采用 CSA 解决本章问题的人。David Seal 向大家展示了对大型数组中的比特位进行计数的一种非常优秀的方法（参见图 10-2 和示例 10-5），这种方法同样适用于大小为 7 的数组（类似于图 10-3 中的方案）。

注 6：全加法器是一个由 3 个 1 位输入（用于相加）和 2 个 1 位输出（和与进位）所组成的电路。请参考 Morgan Kaufmann 1990 年出版的 John L. Hennessy 和 David A. Patterson 所著的《计算机体系结构：量化研究方法》一书。

数值是这一列的 a , b , c 三个分项的 1 位数目和。因此, h 中的每一个 1 位表示 a , b , c 中的两个 1 位比特; 而 l 中的每个 1 位则表示 a , b , c 中的一个 1 位比特。这样的话, 总的种群计数值 (等式右边的值) 就等于 h 中的 1 位数目的两倍加上 l 中的 1 位数目, 其和就是示例中的 23。

令 n_c 为 CSA 步骤中所需要的指令数, n_p 为某个字进行种群计数所需要的指令数。一台典型的 RISC 计算机上, $n_c = 5$, $n_p = 15$ 。如果忽略数组取数和循环控制 (不同机器上相关代码的区别可能相当大), 对于数组中的每个字而言, 前面我们所讨论的循环体需要花费 $(n_c + 2n_p + 2) / 3 \approx 12.33$ 条指令 (式子中的“+2”表示循环体中的 2 次加法)。

另一种运用 CSA 操作的方法可以使程序更为高效、精简。参见示例 10-4。平均每字耗费 $(n_c + n_p + 1) / 2 = 10.5$ 条指令 (忽略循环控制和数组取数)。

示例 10-4：数组种群计数，处理 2 个一组的元素

```
#define CSA(h,l, a,b,c) \
{unsigned u = a ^ b; unsigned v = c; \
h = (a & b) | (u & v); l = u ^ v;}
int popArray(unsigned A[], int n) {
    int tot, i;
    unsigned ones, twos;
    tot = 0; // 初始化。
    ones = 0;
    for (i = 0; i <= n - 2; i = i + 2) {
        CSA(twos, ones, ones, A[i], A[i+1])
        tot = tot + pop(twos);
    }
    tot = 2*tot + pop(ones);
    if (n & 1) // 如果还剩最后一个字,
        tot = tot + pop(A[i]); // 则把它的种群计数值也加上。
    return tot;
}
```

示例 10-4 编译后，CSA 操作展开为：

```
u = ones ^ A[i];
v = A[i+1];
twos = (ones & A[i]) | (u & v);
ones = u ^ v;
```

代码根据编译器对已载入 (loaded) 数据 (quantity) 的后续载入 (loads) 进行优化处理, 这个过程被称为共用 (commoning)。

还有其他一些使用 CSA 操作进一步减少计算某个数组种群计数值所需要的指令的方法。用一张电路图表就可以非常容易地理解这些方法。例如, 图 10-2 描述了一种循环的编写方法, 可以一次取 8 个数组元素, 然后将其压缩到 4 个数中, 分别标记为 *eights*, *fours*,

twos 和 *ones*。*fours*、*twos* 和 *ones* 在下次迭代时，会被反馈到 CSA 中。*eights* 中的 1 位数目的计算是通过执行字级别（word-level）的种群计数函数求得的，并且这个计数值是累加的。整个数组处理完毕后，总种群计数值为：

$$8 \times \text{pop}(\text{eights}) + 4 \times \text{pop}(\text{fours}) + 2 \times \text{pop}(\text{twos}) + \text{pop}(\text{ones})$$

代码请参见示例 10-5，示例 10-2 使用了示例 10-4 中所定义的 CSA 宏。图 10-2 中，CSA 模块（blocks）的编号方式与示例 10-5 中的 CSA 宏的调用顺序是一致的。忽略数组取数（array loads）和循环控制（loop control），对于数组中的每个字而言，循环体的执行时间是 $(7n_c + n_p + 1) / 8 = 6.375$ 条指令。

示例 10-5：数组种群计数，处理 8 个一组的元素

```
int popArray(unsigned A[], int n) {
    int tot, i;
    unsigned ones, twos, twosA, twosB,
        fours, foursA, foursB, eights;
    tot = 0; // 初始化。
    fours = twos = ones = 0;
    for (i = 0; i <= n - 8; i = i + 8) {
        CSA(twosA, ones, ones, A[i], A[i+1])
        CSA(twosB, ones, ones, A[i+2], A[i+3])
        CSA(foursA, twos, twos, twosA, twosB)
        CSA(twosA, ones, ones, A[i+4], A[i+5])
        CSA(twosB, ones, ones, A[i+6], A[i+7])
        CSA(foursB, twos, twos, twosA, twosB)
        CSA(eights, fours, fours, foursA, foursB)
        tot = tot + pop(eights);
    }
    tot = 8*tot + 4*pop(fours) + 2*pop(twos) + pop(ones);
    for (i = i; i < n; i++) // 简单地加上最后
        tot = tot + pop(A[i]); // 0~7 个元素。
    return tot;
}
```

除了图 10-2 中的连接方式外，CSA 还有许多其他的连接方式。例如，增强型指令级（increased instruction-level）的并行可以通过以下步骤实现：前三个数组元素放入到第一个 CSA，接下来的三个数组元素放入到第二个 CSA，那么两个 CSA 的指令就可以并行地执行。通过重新排列 CSA 宏的三个输入操作数，或许可以实现增强型并行（increased parallelism）。根据图 10-2 中的设计，大家很容易知道如何仅使用前三个 CSA 来构造出处理四个一组分组的数组元素的程序，甚至包括如何扩展来支持处理十六个以上（包括十六个）一组分组的数组元素的程序。图中的设计稍微分摊了一些取数，这对于取数次数有相当低限制的机器而言，是很具有优势的，因为取数次数在任何时候都是一个很重要的问题。

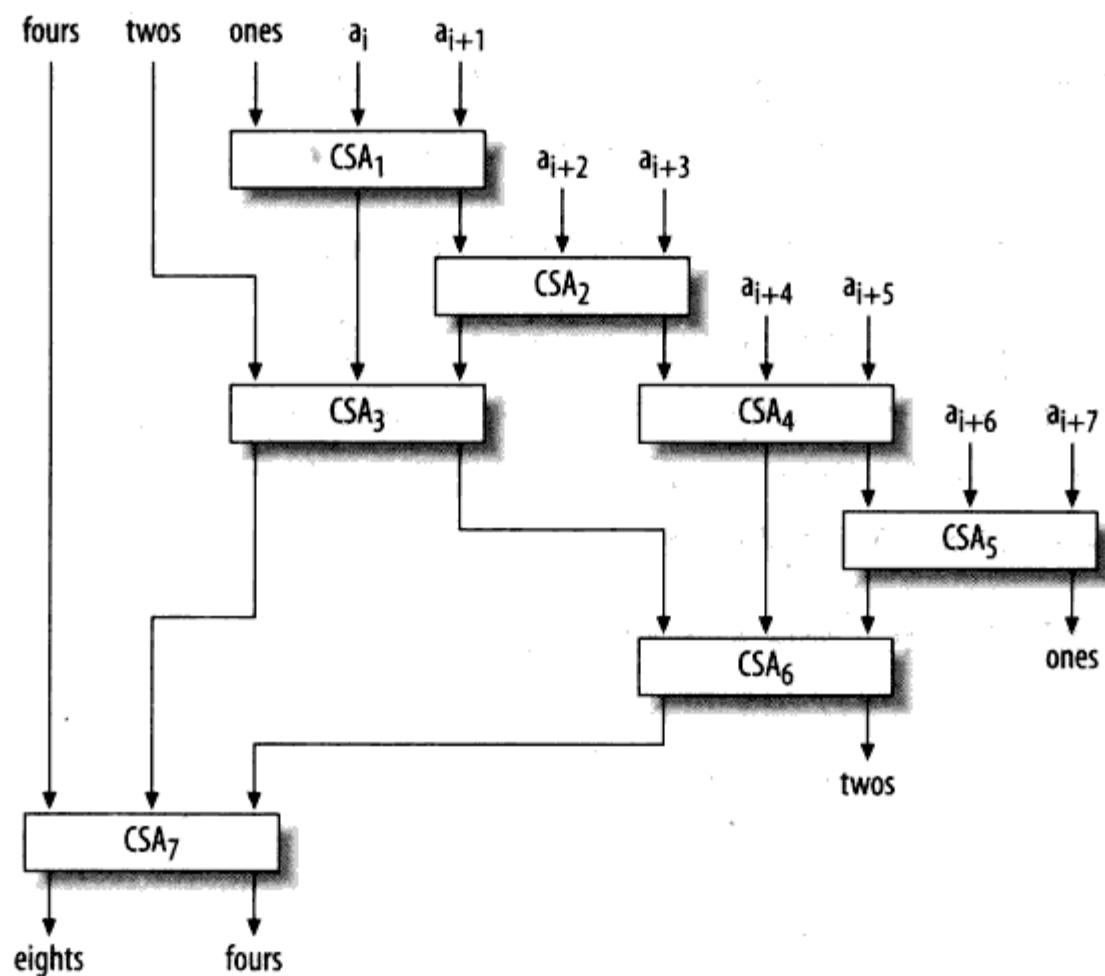


图 10-2：数组种群计数的电路

将图 10-2 中的设计推广到不同组大小 (group sizes) 的数组上，我们在表格 10-1 中总结出了分别所需要的指令数。中间的两列数值忽略了取数 (loads) 和循环控制 (loop control)。第三列给出的是对于输入数组中的每个字而言，其总循环体指令执行次数。需要注意的是，第三列中的代码由不具备索引化取数 (indexed loads) 的初级 RISC 计算机的编译器产生。

表 10-1：数组种群计数的单字花费指令数

程序	取数和循环控制除外的指令		循环中的所有指令 (编译器输出)
	公式	令 $n_c = 5, n_p = 15$	
幼稚方法	$n_p + 1$	16	21
分组大小为 2	$(n_c + n_p + 1) / 2$	10.5	14
分组大小为 4	$(3n_c + n_p + 1) / 4$	7.75	10
分组大小为 8	$(7n_c + n_p + 1) / 8$	6.38	8
分组大小为 16	$(15n_c + n_p + 1) / 16$	5.69	7
分组大小为 32	$(31n_c + n_p + 1) / 32$	5.34	6.5
分组大小为 2^n	$n_c + \frac{n_p - n_c + 1}{2^n}$	$5 + \frac{11}{2^n}$	

用于计算 n 个字的种群计数值的指令数，从幼稚方法的 $16n$ 条，降到了 CSA 方法的 $5n$ 条（ 5 是实现一个 CSA 电路所需要的指令数目）。这是一个执行时间极限方面相当令人惊讶的成绩。

对于小型数组而言，有些设计比图 10-2 中更优秀。例如，对于一个 7 字数组，图 10-3 中的设计相当高效（注 7）。它执行的指令总数为 $4n_c + 3n_p + 4 = 69$ 条，平均每字 9.86 条。对于任意的正整数 k ，类似的设计适用于大小为 $2^k - 1$ 个字的数组。针对 15 个字的设计执行 $11n_c + 4n_p + 6 = 121$ 条指令，平均每字 8.07 条。

应用

种群计数指令的用途非常广泛。正如本章开始所提到的那样，其用途之一就是计算位串 (bit strings) 所表示的集合的大小。在这种表示方法中，“宇宙 (universe)” 集合的成员按顺序编号。集合用位串来表示，位串的第 i 位当且仅当成员 i 在集合中时才为 1。

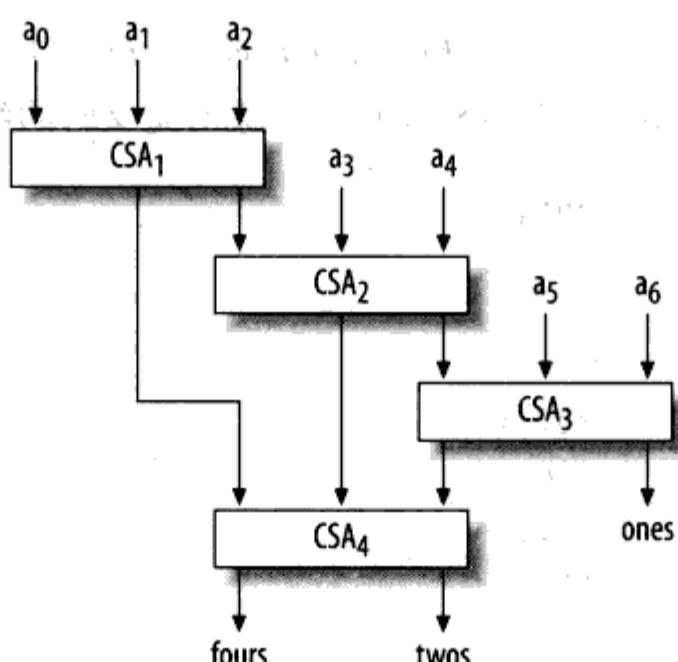


图 10-3：用于计算 7 字种群数目总和的电路

另一个简单应用是计算两个位向量之间的“Hamming 距离”，这一应用来自与错误校验码理论。简单一点说，Hamming 距离就是向量间对应位取不同值的位数（注 8），即：

$$\text{dist}(x, y) = \text{pop}(x \oplus y)$$

种群计数指令可以用于计算字中后缀 0 的数目，使用如下的关系式：

注 7：Seal, op. cit.

注 8：参见 Computer Science press 计算机科学出版社 1989 年出版的 A. K. Dewdney 所著的《The Turing Omnibus》中关于错误校验码的章节。

```
ntz(x) = pop(~x & (x - 1)) = 32 - pop(x | ~x)
```

(读者如果不熟悉算术和逻辑操作的混合运算，可能需要停下来思考一段时间才能搞明白其中的工作原理)。函数 $ntz(x)$ 用途同样非常广泛。例如，早期的计算机一旦发生中断，就会在一个特殊的寄存器中存储一个表示“中断原因”的位。比特位所放置的位置能够指示所发生的中断的类型。位置按照优先级的顺序选择，通常较高优先级的中断放置在较低有效位。同时可以设置两个以上的比特位。管理程序通过对特殊寄存器中所存储的数执行 ntz 函数后，才决定处理哪个中断。

种群计数的另外一个应用是，对于使用特定压缩方式表示的中等稀疏数组 A ，允许快速的直接带索引访问。在这种压缩表示中，仅存储已定义的或者非零的数组元素。使用一个额外的位串 $bits$ ， $A[i]$ 中凡是有定义的位置 i ， $bits$ 中相应的比特位 i 是 1。 $bits$ 位串通常很长，所以它会被分割成一组 32 位的字；其中 $bits$ 长串的第一个 bit (最低有效位) 位于第一个 $bits$ 字的第 0 个 bit。

字数组 $bitsum$ 是一个加速装置，其中 $bitsum[j]$ 是前 j 个 $bits$ 字的 1 位数目的总和。下表描述了在数组的第 0, 2, 32, 47, 48 和 95 个元素有定义时的结果：

bits	bitsum	data
0x00000005	0	$A[0]$
0x00018001	2	$A[2]$
0x80000000	5	$A[32]$
		$A[47]$
		$A[48]$
		$A[95]$

接下来的任务很重要：完全数组中，给定一个“逻辑”下标 i 。如果相应元素存在，则将其转换成存储该元素的“物理”索引 $sparse_i$ ；否则，请给出某种提示信息。对于上述表格中的数组，我们希望将 47 转换成 3, 48 转换成 4, 49 转换成“不存在”。给定一个逻辑下标 i ，数据数组的相应下标 $sparse_i$ 是数组 $bits$ 中所包含的所有比特位中位于第 i 位之前的所有 1 位的数目和。具体计算如下：

```
j = i >> 5;                      // j = i/32。
k = i & 31;                        // k = rem(i, 32);
mask = 1 << k;                    // "1" 位于第 k 位。
if ((bits[j] & mask) == 0) goto no_such_element;
mask = mask - 1;                  // 第 k 位右边全部为 1。
sparse_i = bitsum[j] + pop(bits[j] & mask);
```

这种表示的空间开销是完全数组中的每个位置占用 2 位。

种群计数函数可以用来生成符合二项式分布的随机整数。给定 $\text{Binomial}(t,p)$, 其中 t 是试验的次数, $p=1/2$ 。利用这个函数所产生的种群生成一个整数时, 需要先生成 t 个随机 *bits*, 然后计算 t 个 *bits* 中 1 的数目。这种方法广泛适用于各种概率 p , 而不仅是 $1/2$ 。(注 9)

计算机界传言, 种群计数对于美国国家安全局 (National Security Agency) 而言非常重要。除了 NSA 自己的人, 恐怕没人知道他们到底利用种群计数从事何种工作, 但很有可能是关于加密的工作或者海量材料的搜索。

注 9: 参见 Addison-Wesley 于 1998 年出版由 Donald E. Knuth 所著的《计算机程序设计的艺术, 第 2 卷半数值算法 (第 3 版)》的 3.4.1 节中的问题 27。

安全通信：自由的技术

Ashish Gulhati

我只谈论在我之后的计算机。虽然这台计算机的最基本操作参数都不值得我去计算——但我还是会为你设计出来。这是一台可以计算出问题最终答案的计算机，一台无穷精细和复杂的计算机，甚至有机生活本身也是其运算矩阵的一部分。

Deep Thought (剧中一台超级计算机的名字)，银河系漫游指南

1999 年中期我飞往哥斯达黎加，与名为自由主义城市 (Laissez Faire City, LFC) 的小组一起工作。该小组正在致力于开发一个帮助迎接个人主权新时代的软件系统 (注1)。

LFC 小组的主要工作是开发一系列保护和增强数字时代中个人权利的软件，包括简单易用的安全电子邮件，网上争执斡旋 (Online Dispute Mediation) 服务，网上证券交易所及私有财产贸易和银行业务系统。很久以前，我曾经在这些系统的一些原型实现上做过一些工作，但 Cypherpunk 的邮件列表和 Bruce Schneier 的《应用密码学》(Wiley 出版社) 挫伤了我在这一技术领域的兴趣。

这些系统的最根本目就是为所有人提供稳定可用的通信隐私。

当我步入 LFC 小组位于哥斯达黎加的圣何塞城外的“临时领事馆”时，他们已经开发出了一个叫作 MailVault 的安全网络邮件系统的原型。该系统运行在 Mac OS 9 上，使用 FileMaker 作为其数据库，且是用 Frontier 编写的。在这个系统中没有包含任何可以运行关键任务的通信服务技术，这就是程序员们当时所编写出的产品。

注 1：参见 The Sovereign Individual: Mastering the Transition to the Information Age, James Dale Davidson and Sir William Rees Mogg, Free Press, 1999。

毫无疑问，这个系统会早早地崩溃，并且脆弱无比。它几乎无法支持两名用户同时进行的并发操作。LFC 面临着在投资方的信任危机，因为他们软件的发布时间已延迟了许多次，并且 MailVault 的第一个 beta 版本，也就是他们的旗舰产品，并没有体现出宝贵的价值。因此，我在开发 LFC 的契约网络（contract network）及系统管理之外的业余时间里，开始从头编写一个全新的安全邮件系统。

这个系统现在被称为 Cryptonite，并且从那以后，在其他项目的间隙之间，这个系统都得到了断断续续的开发和测试。

Cryptonite 第一个原型被授权给 LFC 作为 MailVault 的 beta 2 版本，并且于 1999 年 9 月进行了公开测试。这是提供给公众使用的一个与 OpenPGP 兼容的网络邮件系统，并且立刻被 LFC 的投资者和 beta 测试者投入到测试中。从那个时候起，通过与用户，开源社区以及市场的交互，Cryptonite 在许多方式上得到了发展。虽然它本身不是一个开源产品，但却使得我得以按照这种方式开发了许多组件并以开源的形式发布。

项目启动之前

无论从开发的角度还是从整个企业的角度来看，多年来独自开发 Cryptonite、销售 Cryptonite 及提供相关服务（包括我的妻子 Barkha 给予我的坚定的支持和提出许多宝贵的意见）都是一次非常有趣且受益匪浅的旅程。

在分析系统之前，我认为首先需要简单阐述一些观点，在整个项目的开发过程中，这些观点在我的脑海里留下了深刻的印象：

- 我的朋友 Rishab Ghosh 曾经嘲笑过，目前有许多炒作是关于互联网如何使得程序员可以通过一根电线在任何地方工作，然而大多数制造这些炒作的人却只是生活在加利福尼亚的一个很小的区域内。独立启动项目的一个伟大之处在于，这个项目可以真正的在任何地方进行，可以方便地暂停或者重新继续。我在 Cryptonite 上花费了许多年的时间，并且先后在四大洲进行开发，Cryptonite 或许是第一款大部分时间在喜马拉雅山地区开发出来的高质量软件。（我在前面使用的“电线”这个词并不严谨。实际上，我们在喜马拉雅山的连接服务中使用了五种无线技术：VSAT 卫星互联网、Wi-Fi、蓝牙、GPRS 和 CDMA）
- 当你在业余时间里独立开发一个项目时，要记住这条古老的程序员名言：“在实验室六个月的编码时间只等价于在图书馆里十分钟的阅读时间。”最大限度地重用现有的代码库是最关键的一点。因此，我决定用 Perl 语言开发这个系统。Perl 语言作为一种应用广泛并且高度灵活的高级语言，拥有成熟、免费的软件模块。并且 Perl 程序员视懒惰为第一美德的哲学渗透到了系统的每一处设计之中。

- 作为面向终端用户的应用软件，易用性是一个非常重要的问题。软件代码的基本功能是要为用户提供简单易用的界面。而在开发面向终端用户的安全应用软件时，对可用性的考虑则是更为重要的。实际上，可用性正是使Cryptonite系统值得开发的一个关键因素。
- 为了获得良好的开端，首先实现一个原型是不错的想法，并且按照从原型到产品的路线，在实现基本功能之后再转向产品部署。这非常有助于你在向数百或者数百万（但愿如此）的用户发布代码之前首先获得正确的基本设计和结构。
- 使系统尽可能地简单通常会是一个好主意。我们要抗拒使用复杂的最新技术的诱惑，除非在项目中确实需要用到这些技术。
- 虽然现在处理器的速度是相当快的，并且程序员的时间通常也比处理器的时间要更值钱，但运行速度对于应用软件来说仍然非常重要。用户都希望他们的应用程序是快速的。对于用户会并发使用的许多网络应用软件来说，在速度的优化上花一些时间是非常值得的。
- 软件应用程序是一个有生命力的实体，需要持续地关注，更新，改进，测试，修改，调整，销售和技术支持。它的成功与漂亮从某种意义上来说直接取决于代码是否足够灵活、持续地发展，是否能够满足用户的需求，并且能够在许多年里反反复复地开展这些工作。
- 如果你设法解决的问题是某些你自己感兴趣的事情，那么这个软件就确实会起到作用。这不仅便于你在用户和开发商之间转换角色，还可以保证你在五年后仍然对这个项目有兴趣——因为构建和销售软件应用程序通常是一个长期的工作。

Cryptonite 的开发动力很大程度上来源于我希望创造一些软件工具以帮助全世界的个人用户获得真正的自由。虽然在独自开发系统时也会遇到困难，但我发现在独自开发项目的同时也为代码在风格和结构上提供了某种一致性，而这在多人开发的项目中是难以实现的。

剖析安全通信的复杂性

诚然，为这个世界带来安全通信的能力是一个非常伟大的想法，因为它保护了个人权利（在后面将会更多地涉及这个话题），然而要正确地实现这个想法则比看上去要更棘手一些。公钥系统旨在解决一些特殊的安全通信问题，但实现起来却经常出现不必要的复杂问题，并且会与一些基本的实际问题背道而驰，例如那些用户将使用这个系统以及如何使用等问题。

在基于公钥密码的实现中，所要解决的根本问题在于密钥认证。如果要对某人发出一个加密消息，那么你就需要她的公钥。如果你被欺骗而使用了错误的公钥，那么你就没有了隐私。

在解决密钥认证的问题上，有着两种截然不同的解决方案。

传统的公开密钥基础设施（PKI）的方法基于 ISO X.509 标准，它取决于被信任的第三方认证系统（CA），它在很多方面根本不能满足用户在一些特殊网络上的实际需求（注 2）。PKI 实现在结构化程度较高的领域取得了重大成功，例如公司 VPN 和安全网站的认证，但在现实世界的异构电子邮件环境里，只取得了很少的进展。

另一种解决方案是目前最流行的基于公钥的安全通信解决方案：Phil Zimmermann 提出的 PGP 及其他衍生协议，现在被标准化为 IETF 的 OpenPGP 协议。OpenPGP 通过在“信任网络（webs of trust）”中简化分布式密钥的认证来保证灵活性以及从基本上分散公钥加密的特性，而不是像 PKI 方法（包括 OpenPGP 的主要竞争者 S/MIME）那样需要依赖一个集中的、层次化的 CA 系统。毫不奇怪，虽然 S/MIME 普遍存在于主要的电子邮件客户端上，但同 OpenPGP 相比，它所满足的用户数量小很多，尽管电子邮件客户端一般缺乏对 OpenPGP 的全面支持。

然而，“信任网络”这种方法自身也存在着问题，它根据用户自己构造的信任链来验证和鉴别公钥。其中主要的问题是两个相关的挑战：一是要确保用户了解如何使用信任网络来认证密钥，二是需要实现一个重要的用户群以确保任意两名用户彼此之间能够容易地找到信任道路。

在“信任网络”的实现中，任何第三方都无法被指定为“信任的”。如图 11-1 所示。每个用户自己是最值得信赖的认证机构，并且可以分配多种信任级别给其他用户，从而实现认证密钥的目的。在以下的任何一种情况中，你都可以认为密钥是合法的：这个密钥是由你直接验证的；或者是由你充分信任的其他人验证的；或者是由一组人验证的，而这组人中的每个人你都部分地信任。

由于“信任网络”的方法不像 PKI 那样将密钥认证方法外包出去，因此用户必须在建立他们的“信任网络”和确定公钥真实性的过程中起决定性作用。这使得在设计基于 OpenPGP 的安全消息系统时，需要把可用性放在首要的位置。

注 2：Roger Clarke 在下面这篇文章中简要地总结了传统 PKI 的缺点：<http://www.anu.edu.au/people/Roger.Clarke/II/PKIMisFit.html>。

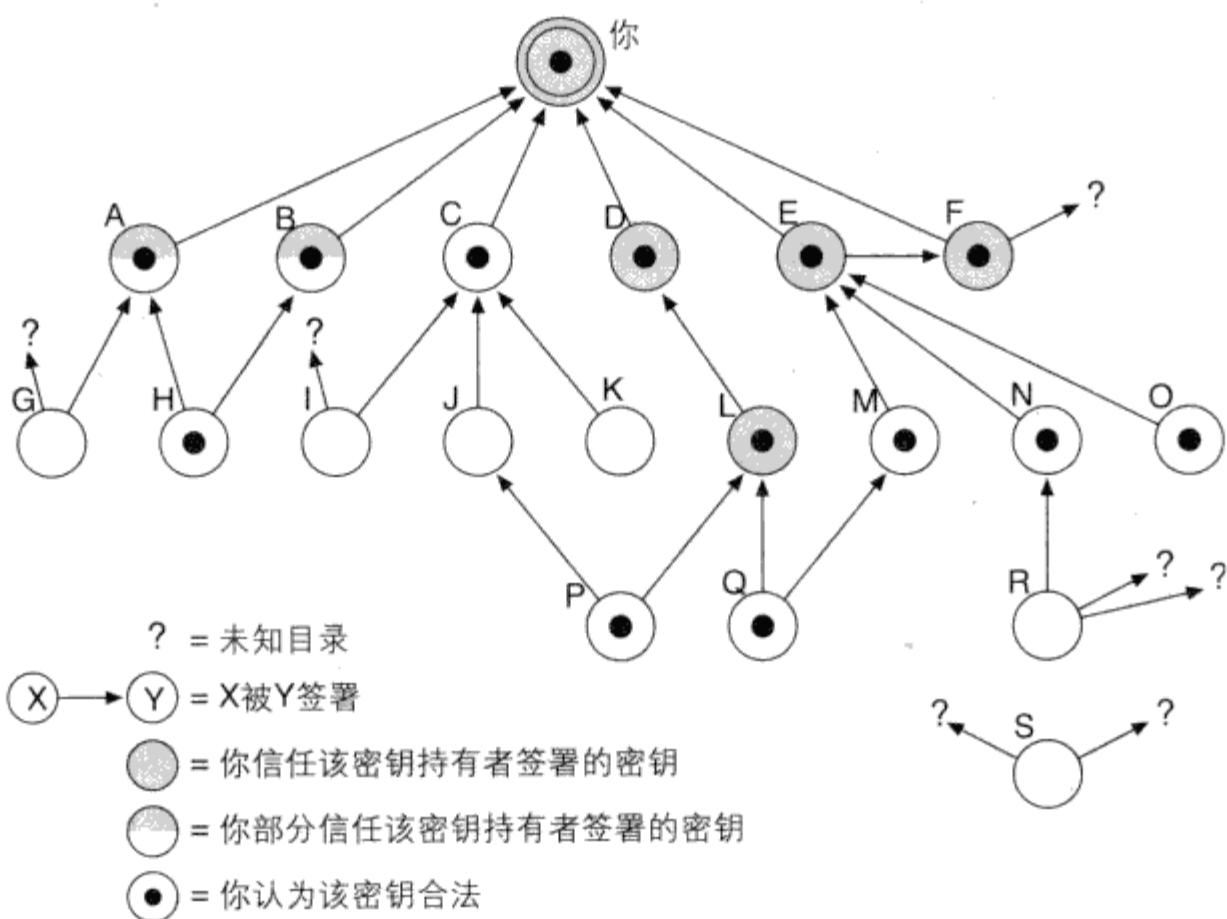


图 11-1：如何在信任网络上验证密钥

可用性是关键要素

由于电子邮件隐私软件经常会过多地要求用户进行验证，因此很少有用户会去使用这些软件。可用性对任何一种安全解决方案的成功实现都是至关重要的，因为如果系统不方便使用，那么用户将会绕过安全机制或以不安全的方式来使用系统，但无论是哪一种情况都将与安全目的背道而驰。

1998 年卡内基梅隆大学在进行 PGP 可用性的专题研究时指出，为电子邮件加密功能创建一个有效可用的界面是一个专业性挑战，并且他们在研究中发现，在 12 个能熟练地使用电子邮件的测试者中，“只有三分之一的人能够在 90 分钟之内使用 PGP 正确地签署和加密电子邮件。（注 3）

对于设计一个安全、可靠并且高效的电子邮件系统的同时又取得非常高的可用性来说，我认为 Cryptonite 是一个有趣的项目。我着手创建一个把 OpenPGP 安全嵌入到电子邮件中的网页邮件系统，并且帮助那些不常使用电子邮件的用户能够有效地运用 OpenPGP 来实现通信隐私。我选择了特别的网络邮件格式，它能够给任何在网吧或者使用带有浏览

注 3：“Usability of Security: A Case Study.” Alma Whitten and J. D. Tygar, Carnegie Mellon University. <http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-155.pdf>.

器的手机的用户带来强有力的通信隐私，而不仅仅只针对于那些能在功能强大计算机上运行桌面电子邮件加密软件的少数用户。

在Cryptonite的设计中将加密作为日常电子邮件中的一个普通部分。这并不是通过屏蔽它所依赖的公钥认证系统的复杂性来实现的，而是通过使系统中的各个组件更为清晰并且更容易被用户访问来实现的。因此，对于可用性的考虑成为了Cryptonite设计和开发的中心环节，这一点体现在以下多个方面：

根据用户反馈和可用性研究来开发界面功能

CMU对用户的研究为最初的设计提供了许多好的想法，并且许多功能都是在对一些不常使用电子邮件的用户进行Cryptonite可用性测试时发展而来的。界面要保持干净，一致和简洁，所有重要的操作至多只需一两次点击就可以完成。

我们从可用性测试中收集了许多重要的需求，其中包括把密钥管理集成到电子邮件客户端的需求，为解密信息提供永久保存的需求，以及把邮件列表视图中的消息结构信息暴露给外部的需求。

与那些桌面电子邮件程序的界面很相似，最终的界面布局只包括三个面板，这是在测试了一个简单的单面板HTML界面以及一个AJAX界面后作出的决策。三面板的界面优化了用户操作：当用户返回到邮件列表时，不会像单面板界面那样需要强行重新加载页面，并且简单的三面板HTML界面比AJAX界面要更加轻便和整洁，此外对带宽的要求也更小。

以直观的方式向用户展现出丰富的并且有意义的OpenPGP对象

所有的密钥操作对于用户都是可用的，包括生成、导入和导出密钥，检查密钥签名和指纹，认证密钥与撤销密钥认证，以及向密钥服务器公布并且检索密钥。这使得用户对她自己的信任网络有了完全的控制。密钥的有效性和信任级别在文本中可以清楚地看到，并且在密钥列表中用不同的颜色标记出来。密钥信任度始终与密钥环和信任数据库最新的状态保持同步。

图11-2给出了用户界面的密钥环视图，图中显示了所有用户标识对于每个密钥的有效性，并以文本和颜色编码的形式显示出来。此外，在图中还以图标的形式显示了密钥类型和每个密钥所有者的信任度（用文本和颜色编码的形式）。每个密钥的所有具体信息都可以通过“edit”链接访问到。

用户行为安全涵义的注意事项和反馈

在让用户管理密钥的同时也带来了风险：用户可能会使用他们的这种权利来减弱系统的安全性。因此，这个应用程序的另一个工作是教授用户一些关于某些行为的安全涵义，比如认证一个密钥，修改钥匙的信任级别，或签署一个消息。

在Cryptonite的所有界面上，涉及安全的操作都会伴随着一些关于该操作的简短而突出的警告。这些消息将出现在同一个屏幕中，而不是出现在那些讨厌的弹出对话框中。

The screenshot shows the 'My Key Ring' window in Cryptonite. At the top, there are buttons for 'NEW', 'ADD', and 'DELETE'. Below is a table with columns: 'Type', 'Identities', 'Owner Trust', and 'Edit'. The table lists various keys:

Type	Identities	Owner Trust	Edit
✓	Ashish Gulhati <ashish@neomailbox.net> (Ultimately Valid)		edit
✓	Barkha Gulhati <barkha@neomailbox.com> (Ultimately Valid)	Ultimately trusted certifier	edit
✓	Ashish Gulhati <ashish@neomailbox.com> (Ultimately Valid) Ashish Gulhati <hash@netropolis.org> (Ultimately Valid) Ashish Gulhati <agul@cpan.org> (Ultimately Valid)	Ultimately trusted certifier	edit
✓	Vipul Ved Prakash. <mail@vipul.net> (Fully Valid)	Fully trusted certifier	edit
✓	Adam Back <adam@cypherspace.org> (Fully Valid)	Fully trusted certifier	edit
✓	Gordon Worley (Mac GPG) <macgpg@rbisland.cx> (Expired) Gordon, Worley (Mac GPG) <redbird@rbisland.cx> (Expired)	Marginally trusted certifier	edit
✓	Ashish Gulhati <ashish@neomailbox.net> (Ultimately Valid)	Ultimately trusted certifier	edit
✓	Philip R. Zimmermann <prz@pgp.com> (Fully Valid)	Fully trusted certifier	edit
✓	Rishab Aiyer Ghosh <rishab@dxm.org> (Fully Valid)	Fully trusted certifier	edit
✓	John Gilmore <gnu@cygnus.com> (Fully Valid) John Gilmore <gnu@toad.com> (Fully Valid) John Gilmore <gnu@freeswan.org> (Fully Valid)	Fully trusted certifier	edit

At the bottom, there are buttons for 'CHECK ALL' and 'CLEAR ALL'.

图 11-2：在密钥环视图中显示了密钥和信任度的信息

内置的关联

在Cryptonite中，用户身份的概念与用户密钥环中的私钥是紧密相连的。在发送邮件时，用户可以使用与密钥环中的私钥相对应的“发送”地址。这有助于用户以直观和不可回避的方式来理解密钥的思想。公钥可以关联到用户地址簿上的联系人，这样每当需要时，就可以取出公钥以进行自动加密。

功能完整的电子邮件客户端

Cryptonite最初只是一个电子邮件客户端，它能够完全支持基于OpenPGP的安全性，并且它内置了密钥管理功能。可用性的重要目标之一就是为用户提供一个完整的电子邮件客户端，且不能让安全功能妨碍电子邮件的可用性。这就要求该系统不仅要提供用户能够在普通电子邮件客户端上找到的所有功能，而且最重要的是为用户提供在邮件目录中进行搜索的功能，包括搜索在加密邮件中的文本，并且与普通电子邮件客户端中把所有信息以不加密方式进行保存相比，这种操作的复杂性不能过大。

基础

当然，在今天的应用软件与硬件之间有着许多的层级，并且软件通常是在许多现有的代码库上构建的。因此在开始一个新的项目之前，找到合适的代码库是一个非常重要的切入点。

种种原因致使我选择了 Perl 来实现 Cryptonite。在 CPAN (<http://www.cpan.org>) 上有着非常丰富且可复用的开源模块，这给我提供了很多帮助，使我可以充分利用现有的解决方案而尽可能少地编写新的代码，并且在接口的选择上有着很多灵活性。这是我在之前的项目中积累的经验，这些经验在 Cryptonite 这一项目中再一次得到了验证。

Perl 通过 XS API 实现了与 C 库和其他库的接口，这使得它能够访问更多的库。Perl 在移植性上的优势以及对面向对象编程技术的支持也是其重要优势。Cryptonite 希望能够通过许可的方式方便地进行修改，而使用 Perl 有助于满足这个需求。

因此，整个 Cryptonite 系统都是使用面向对象的 Perl 来实现的。这个项目促进了许多开源 Perl 模块的开发，并且我在 CPAN 上发布了这些模块。

我之所以选择 GNU/Linux 作为开发平台，是因为在一个类 Unix 环境中开发出的代码很容易被部署到其他的类 Unix 平台上。当时，在 Windows 或 Mac 系统（OS X 还处于 pre-beta 版本）中还不支持运行可用于上千个用户并发访问的面向关键任务的软件。总之，Linux 是我喜欢的桌面环境，自然也就是开发平台的不二选择。

2001 年，项目的开发和发布工作迁移到了 OpenBSD 上，并且自 2003 年以来，我同时在 OS X 和 OpenBSD（以及 Linux）平台上继续我的开发工作。选择 OS X 是因为它作为一个可移植的桌面操作系统有着非常出众的可用性，并且与其类 Unix 的系统基础有着很好的结合能力，同时它还能够运行各种开源软件的能力。选择 OpenBSD 作为一个部署平台是因为它的可靠性，它拥有最佳的安全记录以及它在代码质量和代码审查上拥有绝对的优势。

开发环境我选择了 Emacs，这是因为它拥有强大的功能、可扩展性以及出色的可移植性，包括在我经常用于开发的手提和可移植设备上的可移植性。我还非常喜欢 Emacs 提供的 cperl 模式，这为 Perl 代码提供了非常好的自动格式化功能，即使“只有 perl 才能解析 Perl”。

设计目标与决策

Cryptonite 被设计为一个与 OpenPGP 兼容的网页邮件系统，该邮件系统必需是安全的、可扩展的、可靠的以及易用的。可移植性和可扩展性则是该项目另外的两个重要目标。

项目初期作出的关键性决策之一就是开发一个完全独立的内核引擎来提供界面的多样性和跨平台的可访问性。对界面设计人员来说，重要的是能够在修改界面的同时无需对内核代码进行修改。把内核代码从界面代码中干净地分离出来可以使我们很方便地对各种界面风格进行试验，然后可以借助于可用性测试来得到最优的界面。这种代码的分离同时也是一个重要的设计功能，可以使得将来构建多样的界面，包括在例如手机和掌上电脑等小设备上的界面。

这个设计需要使用客户-服务器的系统，包括一个定义明确的内部 API 以及在 Cryptonite 引擎和用户界面之间清晰的功能分离和权限分离。然后，对于这个内核引擎的界面可以用任何一种语言以及任何一种用户界面框架来实现，同时可以开发一个参考界面以进行在线的可用性测试。

另一个是部署提供灵活性，这需要提供一个选项：在服务器或者用户自己的机器上来进行密码操作，这两种方式各有优点和缺点。

原则上是希望把密钥操作限制在用户的机器上，但实际上这些机器可能在物理上是不安全的或者容易被间谍软件攻破的。而另一方面，服务器有着物理上的高安全性和专业的软件维护，这使服务器端加密（特别是与硬件标记相关联的认证）对于许多用户来说是更安全的选择。这也是选择 Perl 作为开发语言的另一个原因：Perl 的高度可移植性使它可以根据需要在服务器或用户机器上运行。

采用面向对象的实现有助于代码在许多年之后依然可以很容易地被理解、扩展、维护和修改。因为代码会通过授权向终端用户开放，代码的可读性和可理解性是重要的目标。

基本系统设计

Cryptonite 最初的设计如图 11-3 所示。

大多数工作由 Cryptonite::Mail::Service 类来完成的，它定义了一个高级服务对象来实现 Cryptonite 系统的所有内核功能。这个类的方法只是根据参数来执行相关运算并且返回状态码和运算结果。所有的方法是不可交互的，并且在这个类中没有用户界面代码：

```
package Cryptonite::Mail::Service;

sub new {      # 对象构造器
    ...
}

sub newuser { # 创建新的账户
    ...
}
```

```

sub newkey {      # 为用户生成新的密钥
    ...
}
...

```

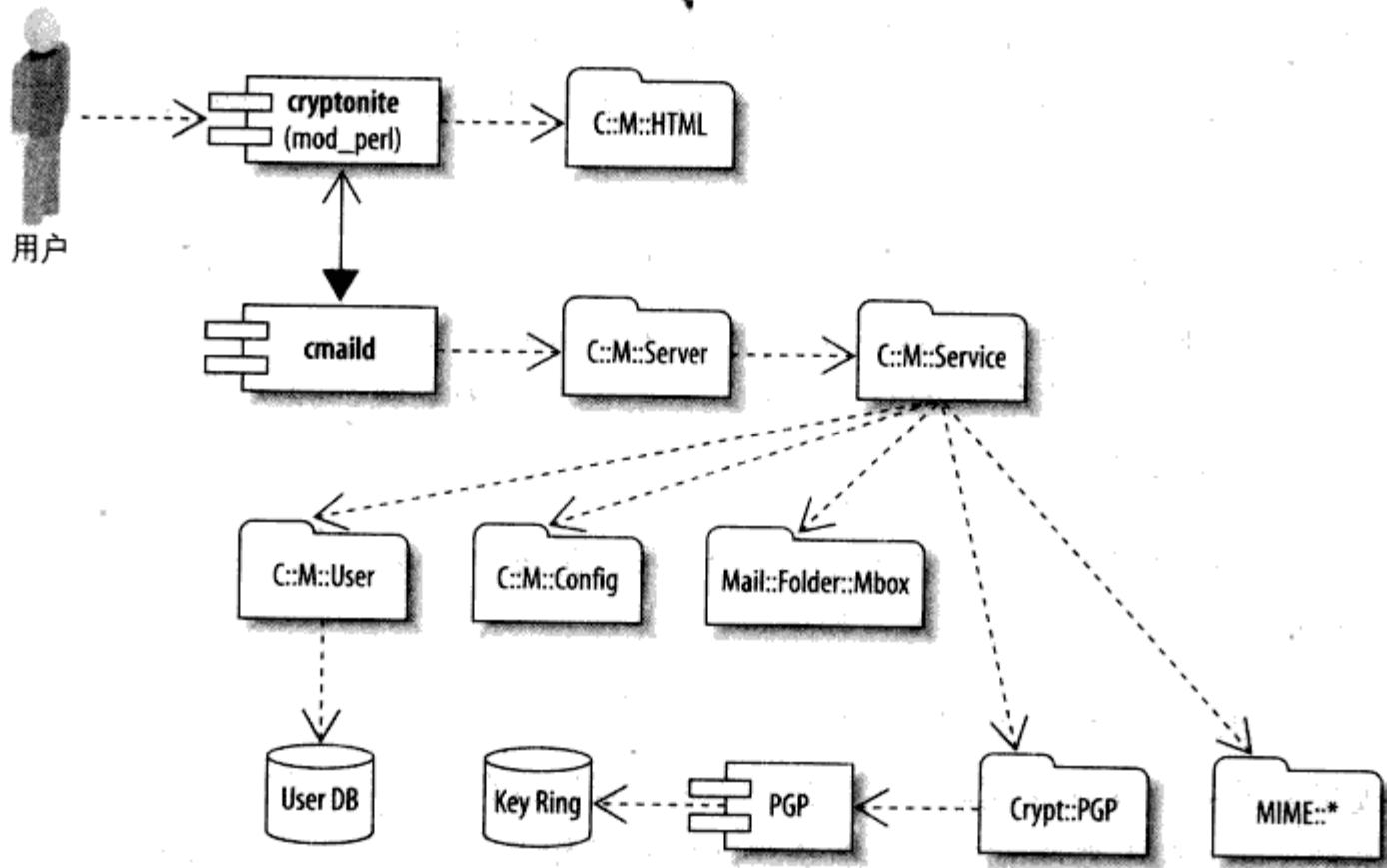


图 11-3: Cryptonite 的最初设计 (C::M 是 Cryptonite::Mail 的简写)

在 Cryptonite::Mail::Service 中包含了系统的所有内核功能，包括创建和管理用户；创建、打开和关闭文件夹；发送、删除和复制邮件；加密、解密和签名认证，并且解析多部分 MIME 消息。

Cryptonite::Mail::Server 使用了 Service 类来实现一个服务器，这个服务器将接收序列化的 Cryptonite API 调用，并把它们分发到 Service 对象上。

序列化最初通过 SOAP 调用来实现，但对 SOAP 对象的解析和处理增加了许多不必要的复杂性和开销。因此，我自己实现了一个简单的序列化架构来取代 SOAP（在判断 <http://wanderingbarque.com/nonintersecting/2006/11/15/the-s-stands-for-simple> 及其评论的七年时间中可以看到，这个决策看上去确实是很正确的）。以下是 Cryptonite::Mail::Server 中的命令分发器：

```

package Cryptonite::Mail::Server;

use Net::Daemon;
use vars qw(@ISA);
use Cryptonite::Mail::Service;

@ISA = qw(Net::Daemon);

```

```

my $debug = 1;
my $cmail = new Cryptonite::Mail::Service;

sub process_request {
    my $self = shift; my ($retcode, $input);

    # 在 eval 中捕获超时异常
    eval {
        local $SIG{ALRM} = sub { die "Timed Out!\n" };

        # 如果在 2 分钟之内没有输入，则触发超时异常
        my $timeout = 120;

        my $previous_alarm = alarm($timeout);
        while( <STDIN>){
            s/\r?\n//;

            # 得到 caller、command 和命令 args.
            my ($caller, $command, @args) = split /(?<!\\):/;
            $debug ? $debug == 2 ? warn "$$: $_\n" :
                warn "$$: $caller:$command:@args[0]\n" : '';
            # 处理流中的参数分隔符
            for (@args) { s/(?<!;);(?!;)/:/sg; s/;;;/sg }
            return if $command =~ /^s*quit\s*/i;

            # 验证命令
            my $valid = $cmail->valid_cmd;
            if ($command =~/$valid/x) {
                # 调用服务的方法。
                $ret = join ("\n",($cmail->$command (@args), ''));
                print STDOUT $ret;
            }
            else {
                # 无效的命令
                print STDOUT ($cmail->cluebat (ECOMMAND, $command) . "\n");
            }
            alarm($timeout);
        }
        alarm($previous_alarm);
    };
}

if( $@ =~/timed out/i ){
    print STDOUT "Timed Out.\r\n";
    return;
}
}

```

Cryptonite 邮件后台程序 (*cmaild*) 通过 Unix 或 TCP 套接字 (socket) 来接收被序列化方法调用，然后调用服务对象的相应函数，并且返回结果代码 (+OK 或 -ERR) 和一个可读的状态信息（比如，“Everything is under control!”），以及可选的其他返回值（比如某个文件夹中的邮件列表，或邮件的文本部分）。如果在返回值中包含多行数据，那么状态消息将会指出客户端希望读取多少行。

当新的客户端连接到服务器时，服务器将启动一个新的进程，这样在从客户端收到最后一条消息之后，Perl 的内置警报（alarm）功能将会给每个新的服务器进程发送一个 SIGALRM \$timeout 的秒数，这使得服务器可以在超时之后断开客户连接。

测试集

由于自动化测试是长期开发工作的一个关键组成部分，因此我在开发项目的同时还开发了一个测试程序集。

内核代码与界面代码的完全分离使我很容易地把测试分成两个组，并且能够迅速判断 bug 以及在代码中精确定位它们。为 *cmaild* 编写测试只需要用合法（或无效）的参数来调用它的方法并判断得到返回值和结果是不是我们所预期的。

Cmaild 的测试程序使用客户端 API 来调用 cmdopen（打开与 Cryptonite 邮件后台程序的连接），cmdsend（向后台程序发送 API 调用）和 cmdayt（向服务器发送“Are you there?”）等命令：

```
use strict;
use Test;

BEGIN { plan tests => 392, todo => [] }

use Cryptonite::Mail::HTML qw (&cmdopen &cmdsend &cmdayt);

$Test::Harness::Verbose = 1;

my ($cmailclient, $select, $sessionkey);
my ($USER, $CMAILID, $PASSWORD) = 'test';
my $a = $Cryptonite::Mail::Config::CONFIG{ADMINPW};

ok(sub {                      # 1: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 2: newuser
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /^OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 3: login
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'login', $USER, $PASSWORD);
    return $status unless $status =~ /OK.*$/;
    1;
}, 1);

ok(sub {                      # 4: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 5: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 6: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 7: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 8: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 9: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 10: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 11: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 12: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 13: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 14: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 15: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 16: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 17: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 18: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 19: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 20: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 21: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 22: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 23: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 24: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 25: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 26: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 27: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 28: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 29: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 30: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 31: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 32: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 33: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 34: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 35: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 36: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 37: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);

ok(sub {                      # 38: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}, 1);

ok(sub {                      # 39: cmdsend
    my $status = cmdsend('test.pl', $a, $cmailclient, $select,
                         'newuser', $USER);
    return $status unless $status =~ /OK.*with password (.*)$/;
    $PASSWORD = $1;
    1;
}, 1);

ok(sub {                      # 40: cmdayt
    my $status = cmdayt($cmailclient);
    return $status unless $status =~ /Are you there?$/;
    1;
}, 1);

ok(sub {                      # 41: cmdclose
    my $status = cmdclose($cmailclient);
    return $status unless $status =~ /OK$/;
    1;
}, 1);
```

功能原型

在第一个原型中，我使用了一个简单的对象持久化模块 Persistence::Object::Simple（这是我的朋友Vipul在之前一个项目中编写的模块）来快速地建立一个基本的用户数据库。使用持久对象有助于代码保持干净和直观，并且提供了一种直接的升级到商用数据库引擎（只需简单地为数据库引擎创建或者派生一个兼容 Persistence::Object::* 类）的方法。

在2002年的年末，Matt Sergeant为Perl程序员创造了另一种从原型到产品的简单路径，即DBD::SQLite模块，“这是一个在DBI驱动中的RDBMS”，可以用来在开发中快速实现数据库原型而无需完整的数据库引擎。但就个人而言，我更喜欢持久化对象，因为SQL查询和DBI调用会给代码带来混乱。

在Cryptonite系统中接收到的邮件被保存为普通的mbox文件，对于原型系统来说，这个策略很不错。当然，在正式产品中必须实现一个更为复杂的邮件保存形式。我决定使用PGP作为加密后端，以避免重写（和维护）所有已经包含在PGP中的加密功能。

当时GnuPG也已出现，我提醒自己将来可能需要使用它来进行加密。因此，我编写了Crypt::PGP5这个Perl模块来封装PGP5的功能。这个模块可以从CPAN上得到（我已经很长时间没有更新这个模块了）。

在开发Crypt::PGP5的加密内核时，我本来可以使用私有的PGPSDK库，但我必须为它创建Perl接口，这比起直接使用PGP的二进制代码来说工作量更大。所以，出于Perl程序员特有的“懒惰”并且牢记TMTOWTDI（注4）原则，我决定用Expect模块来实现与PGP二进制代码之间的自动交互，使用与用户相同的界面。这在第一个原型中工作得很好。

我使用Text::Template模块开发了一个基本的网络接口，这个模块用来开发HTML模板。在Cryptonite::Mail::HTML模块中包含了所有与网络接口相关的代码，包括会话处理。

我利用三个月的业余时间编写完了代码，原型系统基本就绪。这个系统实现了一个完整的网络接口，基本的MIME支持，OpenPGP加密，解密，签名和签名验证，在线新用户注册，和一个新的用于认证的登录密码方法：来自于ID Arts的PassFaces。

清理，插入，继续……

在哥斯达黎加完成了Cryptonite最初的原型开发之后，我继续独自进行Cryptonite的开

注4： 总是有不只一种的方法来做这个工作，Perl编程中的核心原则之一。

发工作。在对代码进行了许多必要的清理之后（原型系统开发得很匆忙，也没有留下太多时间来进行重构或代码测试），为了从简单的原型跃迁到产品，我开发了一些将来需要使用的Perl模块和组件。包括Crypt::GPG（接口与Crypt::PGP几乎完全相同，这样在切换到Cryptonite中用于密钥操作得GnuPG时，差不多只需修改一行代码），以及Persistence::Database::SQL和Persistence::Object::Postgres（这些模块提供了在Postgres数据库中持久化对象的功能，它们有着类似于Persistence::Object::Simple的接口，这使得后端数据库得以无缝地进行切换）。

Persistence::Object::Postgres与Persistence::Object::Simple一样，都使用一个散列容器的blessed reference（注5）来保存键-值对，并可以通过一个commit调用提交给数据库。它还使用了Perl的Tie机制把Postgres的大数据对象（BLOB）关联到文件句柄，从而实现了对数据库中的大数据二进制对象进行基于文件句柄的访问。Persistence::Database::SQL相对于Persistence::Object::Simple的好处之一在于，它可以支持对真实数据库的特定查询。例如，如果使用Persistence::Object::Simple，那么就无法通过简洁的方式来迅速查找某个特定用户的记录，但如果采用Persistence::Database::SQL，则从数据库中得到某个特定用户的记录便非常简单：

```
sub _getuser { # 从数据库中得到用户对象
    my $self = shift; my $username = shift;
    $self->db->table('users'); $self->db->template($usertmpl);
    my ($user) = $self->db->select("WHERE USERNAME = '$username'");
    return $user;
}
```

如果使用Persistence::Object::Simple，那么你就必须在数据目录中遍历所有的持久化对象或者采用一些诸如直接在数据目录的纯文本持久化文件中进行搜索的特殊方法。

Persistence::Object::Postgres的接口与Persistence::Object::Simple的接口在许多方面都非常相似。如果要使用其中的任一模块来修改对象，那么代码都是相同的：

```
my $user = $self->_getuser($username);
return $self->cluebat(EBADLOGIN) unless $user and $user->timestamp;
$user->set_level($level);
$user->commit;
```

在大多数原型代码基本正常运行之后，就开始将数据库系统从纯文本数据库转换到真正的DBMS，这标志着Cryptonite开发的第二阶段：使系统为面向现实世界的部署做好准备。对于原型开发来说，Persistence::Object::Simple是很好的，因为它并不需要在开发

注5： 在Perl中，当一个引用(reference)通过bless被关联到一个类时，这个引用就成了一个对象，因此“blessed reference”是对Perl中特定对象的称呼。

中使用专门的数据库服务器，并且由于对象是被存放在纯文本文件中，因此在调试时可以很容易地进行分析。

在 Crypt::GPG 和 Persistence::Object::Postgres 中采用同构的接口使得对于 Cryptonite::Mail::Service 的主要修改（编码以及数据库后端处理）只需少量的编码就可以完成。

改进邮件存储

在第一个原型中，用户邮件被保存为简单的 *mbox* 文件，但在产品系统中要求比普通的文件能够更高效地访问和更新个人邮件。同时，我想加入一项非常重要的目标：提供能够容错的邮件存储复制功能。

对于可用性的考虑也为邮件存储提出了一些要求。与大多数电子邮件客户端程序不同，在 Cryptonite 中，用户可以在邮件列表中看到 MIME 的结构信息。这使得用户可以在邮件列表中直观地看到哪些消息被加密或者签名。在邮件列表中对邮件部分信息的可访问性还使得用户可以直接打开邮件的子部分。在邮件列表视图的最右一栏中以图标的形式显示了邮件部分，如图 11-4 所示。

Date	From	Subject	Size	Parts
6:43 AM	Ashish Gulhati	Photos...	87K	
6:38 AM	Ashish Gulhati	No subject	2K	
6:15 AM	feedback	[The Sovereign Cyborg]...	1.5K	
9:27 AM	Angelina Rowe	Best prices for u...	11.6K	
12:46 AM	"Lora Sheldon"	Cursos completos de Tango	5.5K	
12:07 AM	"john williams"	Do you know why Katherine the...	1.9K	

图 11-4：显示邮件部分（Parts）的邮件列表视图

为了实现上述视觉反馈，邮件存储需要高效率地提供一列邮件的精确 MIME 结构信息。此外，在 OpenPGP/MIME 的规范中允许邮件的 MIME 信息嵌套在加密或签名部分，这使得事情进一步复杂化，因此只有理解了 OpenPGP/MIME 的邮件存储才能返回加密邮件或签名邮件的准确的 MIME 结构信息。

因此我决定使用 Mail::Folder 模块来实现一个基于 SQL 的邮件存储后端，它拥有大多数 IMAP4rev1 服务器的功能。这个系统的核心是 Mail::Folder::SQL 类，该类基于 Mail::Folder 并且使用了 Persistence::Object::Postgres。这是当人们还没有把太多的目光投向 IMAP 时的情况。我并没有选择某个现有的 IMAP 服务器来作为邮件存储，因为我希望实现一些其他的功能；这些功能在大多数 IMAP 服务器上并没有得到很好的支持，例如邮件存储复制功能和无需解析整个邮件就能检索到关于 MIME 消息结构的

详细信息的功能。

即使有些IMAP服务器或许能够满足我的需求，我仍然不想让Cryptonite依赖和束缚于IMAP服务器实现中的任何特定的功能。总之，实践证明这是一个很好的决定，即使这使得我们后来在系统中一些不太重要的代码上花了大量的精力。

在邮件存储复制中使用了我所编写的两个Perl模块：Replication::Recall 和 DBD::Recall，其中使用了Eric Newton的Recall复制框架 (<http://www.fault-tolerant.org/recall>) 以在多台服务器之间复制数据库。这其中的想法是把这个框架用作为原型，并且在将来再定制成一个新的数据库复制系统。

随着对加密、数据库和邮件存储后端的不断改进，并且使用了一个新的更加整洁的主题风格，Cryptonite的第一个内部beta版本于2001年10月正式上线。许多不同层次的用户对Cryptonite进行了测试，其中一些用户甚至将Cryptonite当作他们主要的邮件客户端来用。在内部beta版本上进行的可用性测试表明，新手用户能成功地生成和导入钥匙，并且在发送、阅读加密或签名的邮件方面没有任何困难。

解密邮件的持久性

加密邮件客户端的一个基本功能是在用户的会话期间能以解密的形式来保持已解密的邮件，使其可以被访问。缺乏这个功能会使得安全邮件客户端使用起来非常麻烦和低效，因为每当你想阅读一封加密邮件或者在加密邮件中进行搜索时，都会要求你输入很长的密码，并且等待解密过程的完成。

在Cryptonite中，之前的解密邮件持久化是通过创建一个新的Mail::Folder类来实现的，这个类是基于Mail::Folder::SQL的。如果一个邮件在Shadow文件夹中已经存在一个副本，那么Mail::Folder::Shadow会把对该邮件的访问委托到对Shadow文件夹的访问，否则，它会访问原先的文件夹。

通过这种方法，当会话有效时，被解密的邮件将被保留在Shadow文件夹中，并且如果要增加持久化的解密功能，几乎都不需要修改代码，只需在使用Mail::Folder::SQL的地方插入Mail::Folder::Shadow模块即可。Mail::Folder::Shadow用一种很简单的委托表(delegation table)方法实现了这一技术。

```
my %method =
  qw (get_message 1 get_mime_message 1 get_message_file 1 get_header 1
       get_mime_message 1 mime_type 1 get_mime_header 1 get_fields 1
       get_header_fields 1 refile 1 add_label 2 delete_label 2
       label_exists 2 list_labels 2 message_exists 1 delete_message 5
       sync 2 delete 2 open 2 set_header_fields 2 close 2 DESTROY 2)
```

```
get_mime_skeleton 1 get_body_part 1);
```

Mail::Folder::Shadow 把对方法的调用委托到适当的 Shadow 文件夹，或者原始文件夹，或者同时委托到这两个文件夹。Perl 中功能强大的 AUTOLOAD 功能是实现委托的一种简单方法，这个功能提供了一种机制来处理在类中没有被明确定义的方法，此外还提供了一种简单的机制以在运行时调整如何对不同的方法进行处理。

如果要访问的消息已经存在于 shadow 文件夹中，那么需要检查 Shadow 存储的方法，例如 get_message 和 get_header 方法，将被委托到 Shadow 文件夹中；否则，这些方法的调用将被委托到原始文件夹中。其他一些方法需要被同时委托到 Shadow 和原始文件夹的方法，例如 add_label 和 delete（这个方法用来删除文件夹），因为这些消息必须改变原始文件夹的状态和 Shadow 文件夹的状态。

还有其他一些方法，例如 delete_message，可以通过数组引用接收一个邮件列表。邮件列表中的一些邮件可能在 Shadow 文件夹中，而其他的邮件则不是。Mail::Folder::Shadow 的 AUTOLOAD 将通过从传入的邮件列表中构建两个列表来处理这样的方法，其中一个是在 Shadow 文件夹中的邮件列表，另一个是在原始文件夹中的邮件列表。然后对于被复制到 Shadow 文件夹中的邮件，在两个文件夹中都调用删除方法；对未复制到 Shadow 文件夹中的邮件，则只在原始文件夹中调用删除方法。

所有这些操作的实际结果就是，cmaild 可以和以前一样继续使用文件夹，并且在会话期间，将解密的消息隐藏在 Shadow 文件夹中。在 Mail::Folder::Shadow 中还有几个其他方法共同完成了该功能，包括 update_shadow，这个方法用来把解密的邮件保存到 Shadow 文件夹中； delete_shadow，这个方法用来根据用户需要来删除 Shadow 文件夹中的邮件；还有 unshadow，用来在会话终止前删除 Shadow 文件夹中的所有邮件。

Mail::Folder::Shadow 实现了在会话期间为已解密的邮件提供持久性的功能，同时还实现了在加密邮件中的搜索功能。这两个功能从用户的角度来说都是非常关键的，但目前与 OpenPGP 兼容的电子邮件系统中很少被实现。

在喜马拉雅山的开发工作

2000 年和 2001 年间我只能断断续续地从事 Cryptonite 的开发工作，这是因为有一些其他的任务，又因为这个项目需要安静的环境，而我当时正在混乱、嘈杂和遭到污染的印度各个城市之间来往和居住，安静的环境很有限。

2002 年的夏天，我和我的妻子在喜马拉雅山度过了一个假期，在那里我终于努力完成

了代码开发的主要工作，包括为Crypt::GPG添加重要的密钥管理功能，以及为密钥管理创建一个集成的界面，这是整个信任网络机制中的一个重要部分。图11-5中给出了密钥编辑的对话框界面，这是整个管理界面的核心。它能够支持指纹验证、浏览和创建用户身份证明，以及指定密钥的信任度。

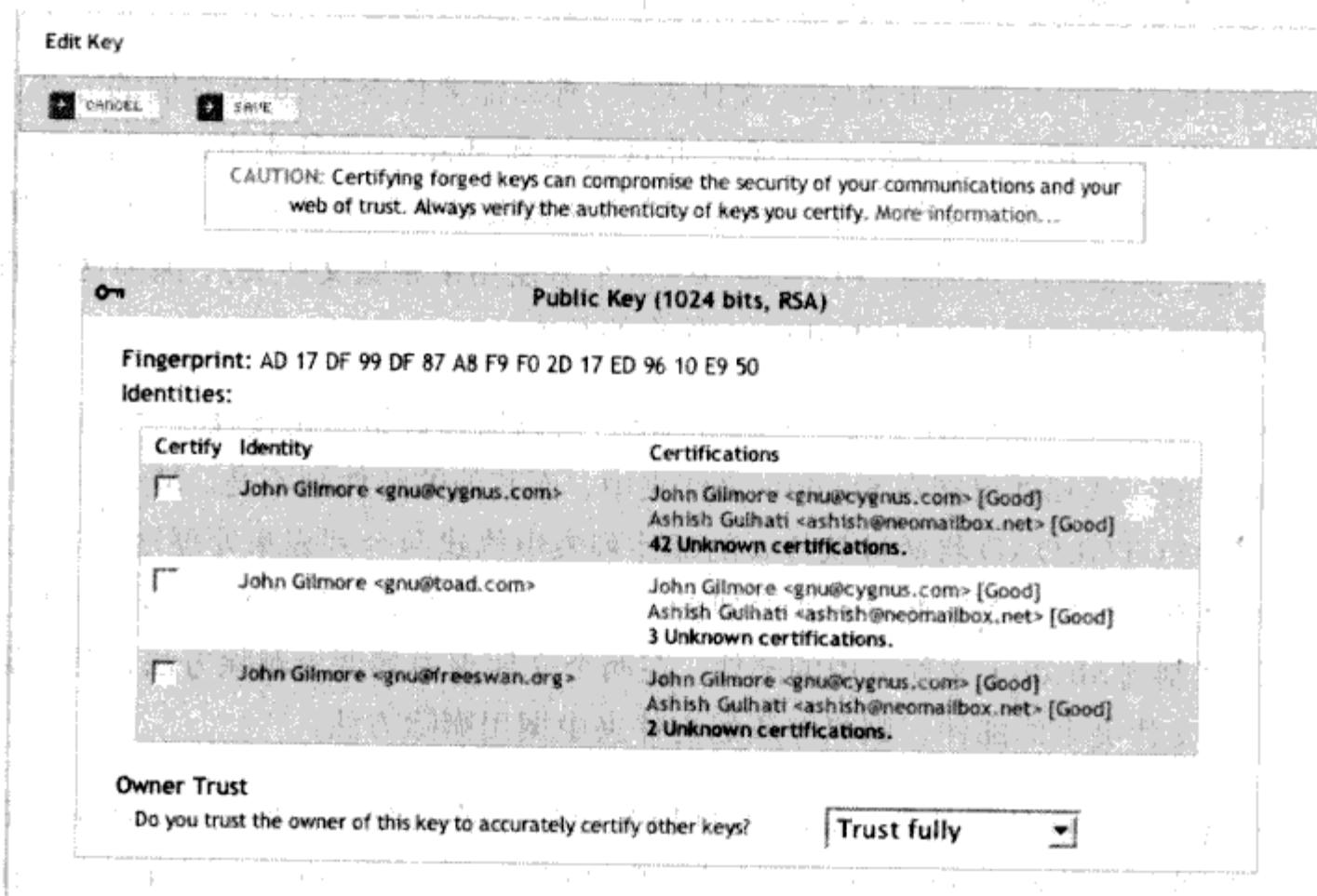


图 11-5：密钥编辑对话框

同时我把系统移植到OpenBSD上，这将是最终的部署平台。

我们已经有了安全电子邮件服务中的所有其他主要组件，由于仍然需要一些时间来为Cryptonite的公开使用做好准备，我们决定立即发布一个商业安全电子邮件服务。这促使我在Cryptonite的开发上要花费更多时间，并且要立刻着手建立测试者群体。

因此在2003年中期，我们发布了Neomailbox，它提供安全的IMAP、POP3和SMTP电子邮件服务。在之后的几年中，这被证明是个正确的决定，它有助于为下一步的开发工作找到资金，使我无需再去承接其他的合同工作，而同时还可以更密切地关注安全私有邮件的市场情况。

2003年秋天，我们在喜马拉雅山的一个小村庄里设立了一个半永久性的开发基地，大约在海拔2000米处，并且从那以后，开发工作主要都是在那里进行的。这样我们经费消耗得很慢，且这一点对于一个谨慎启动的项目来说是非常关键的，同时给了我许多时间

和安静的环境来开展 Neomailbox 和 Cryptonite 的工作。

在一个仍然处于19世纪的喜马拉雅山地区的村庄中开发面向关键任务的高技术系统，我们有着自己的感受，Nikolai Roerich——这位俄国高产的艺术家、作家和哲学家，他所说的话描述了我们的感受，“实际上，只有这里，只有在喜马拉雅山，才存在着取得成就所需要的独特的条件和安静的环境。”

保证代码的安全性

最初，代码只是被设计为原型系统，因此我并没有过多地担心它的安全性。但是随着系统作为公开的 beta 版本的发布，此时在锁定的代码中至少需要实现：

- 完全的权限分离。
- 验证任意的用户输入。
- 对 Crypt::GPG 的安全审查。
- 为所有潜在的安全问题建立文档。

权限管理从一开始便建立在系统中，它通过特权用户来运行 *cmaild*，并通过它的 API 进行交互。这使得 *cmaild* 可以执行一些特权操作，例如修改系统配置文件和以受控方式执行加密操作，而同时不会让给网络服务器进程访问敏感的资源。只有几处地方需要对内核代码和界面代码之间的分离操作进行整理。

其中一个地方是 MIME 消息与二进制附件的复合。当代码使用 Persistence::Object::Simple 来构建时，*cmaild* 协议回避了对二进制 MIME 消息复合的处理。用户上传的附件被保存在一个临时目录中，*cmaild* 和网络服务器都进行有权限的访问。因而，有必要在同一台服务器上运行 *cmaild* 和 Cryptonite 网页界面。

随着向 Persistence::Object::Postgres 的迁移，我们可以在前端和后端之间通过数据库方便地传递二进制对象，而无需依靠于直接的文件系统操作。这点非常重要，因为界面、数据库和 Cryptonite 引擎全都希望能够运行在它们自己的独立服务器或者负载平衡机群上。

很明显要增加对输入的验证（检查用户输入的有效性，例如文件夹和消息的标识符）。稍作修改之后，Params::Validate 模块被用来在 Cryptonite::Mail::Service 的每个方法中添加对用户输入的验证。例如 mvmsgs 方法按照下面的方式来验证输入：

```
sub mvmsgs {    # 把一组邮件移到其他的邮箱中
    my ($self, $username, $key, $dest, $copy, @msgnums) =
        (shift, lc shift, shift);
    my ($user, $session, $err) = $self->validateuser($username, $key);
```

```

    return $err if $err;
    return $self->cluebat(@{$@}) unless eval {
        ($dest, $copy, @msgnums) = validate_with( params => \@_,
            extra => [$self], spec = [
                { type => SCALAR, callbacks =>
                    { 'Legal Folder Name' => $self->legal_foldername } },
                { type => SCALAR, callbacks =>
                    { 'Boolean Flag' => $self->opt_boolean }, optional => 1 },
                { type => SCALAR, callbacks =>
                    { 'Legal Message Number' => $self->legal_msgnum } }
            x (@_ - 2) ]
    );
}

```

对于每种类型的输入字段，用户输入的可接受性是通过存储在Cryptonite::Mail::Config模块的一个哈希表中的回调函数引用来指定的：

```

LGL_FOLDERNAME => sub { $_[0] =~ /$_[1]->{ "VFOLDER" }/i
                           or die (['EBADFOLDER', $_[0]]), ...
OPT_BOOLEAN     => sub { $_[0] eq '' or $_[0] eq 0 or $_[0] eq 1
                           or die (['EBADBOOL', $_[0]]), ...
LGL_MSGNUM      => sub { $_[0] =~ /$_[1]->{ "VMSNUM" }/
                           or die (['EBADMSGNUM', $_[0]]), ...

```

每当一个输入参数被验证为有效时，将会调用类似的子程序。用来验证正确性的正则表达式被单独存放在 Cryptonite::Mail::Config 中。

虽然大多数的验证子程序基本上都是相同的，但它们都是独立的，这样，其中任何一个子程序的修改都不会影响到其他的子程序或者牺牲这部分代码的清晰性。用于验证的正则表达式和错误提示字符串也被存放在一张表中，这在将来可以实现局部化。

Persistence::Object::Postgres 也执行它自己的输入验证，以免受到 SQL 注入攻击。

审查 Crypt::GPG

Crypt::GPG曾被编写为一个原型，因此有必要在公开测试系统之前完成审查以消除所有潜在的安全问题。

自 2001 年以来，Crypt::GPG 在 CPAN 上可以自由使用，并且我从使用它的用户那里得到了宝贵的反馈意见。许多用户说他们非常喜欢模块的清爽和简单的界面，而有些用户在某些平台上运行时遇到了麻烦，用来与 GnuPG 进行交互的 Expect 模块运行得不太正常。（例如，Expect 使用了 UNIX 伪终端[ptys]来作为它的进程间通信（IPC）机制，但这在 Windows 上是无法工作的）

Expect 模块的接口和语法也有些复杂，使代码阅读起来有些困难，例如在本节中可以看到的 sign 方法：

```

my $expect = Expect->spawn ($self->gpgbin, @opts, '-o-', '--sign',
                           @extras, @secretkey, $tmpnam);
$expect->log_stdout($self->debug);

$expect->expect (undef, '-re',
                  '-----BEGIN', 'passphrase:', 'signing failed');

if ($expect->exp_match_number == 2) {
    $self->doze; print $expect ($self->passphrase . "\r");
    $expect->expect (undef, '-re', '-----BEGIN', 'passphrase:');

    if ($expect->exp_match_number == 2) { # Passphrase incorrect
        $self->doze;
        print $expect ($self->passphrase . "\r");
        $expect->expect (undef, 'passphrase:'); $self->doze;
        print $expect ($self->passphrase . "\r");
        $expect->expect (undef);
        unlink $tmpnam;
        return;
    }
}

elsif ($expect->exp_match_number == 3) {
    unlink $tmpnam; $expect->close;
    return;
}

$expect->expect (undef);
my $info = $expect->exp_match . $expect->exp_before;

```

使用基于Expect的模块还会造成*Heisenbug*——即一些不容易重现的bug，我发现这些bug是因为过快地把输入发送到gpg中造成的。在先前的代码中调用doze是绕过这个问题的方法之一，即在发送输入信息的下一位到gpg之前首先延迟几毫秒。这个方法通常可以奏效，但对于负载很大的系统来说依然可能导致故障。

所有这些问题都指出了，我们应该弃用Expect而使用其他机制与GnuPG程序交互。我考虑过编写一个纯粹的Perl程序来实现OpenPGP，但基于我在前面决定使用GnuPG的原因，我决定放弃这个想法：Cryptonite主要是一个支持Open-PGP的电子邮件客户端。如果完整地实现OpenPGP，那么至少会使我所需要维护的代码的复杂性加倍。（注6）在做了一些试验之后，我决定使用Barrie Slaymaker编写的IPC::Run来支持与GnuPG的通信。在使用了IPC::Run之后，原来的代码变成了：

注6：Crypt::OpenPGP是一个完全用Perl实现的OpenPGP模块，它是由Ben Trott在2001到2002年编写的，可以从CPAN中得到。我希望在将来支持多加密后端处理的Cryptonite版本中使用它。

```

my ($in, $out, $err, $in_q, $out_q, $err_q);
my $h = start ([${$self->gpgbin}, @opts, @secretkey, '-o-',
                '--sign', @extras, $tmpnam],
               \$in, \$out, \$err, timeout( 30 ));

my $i = 0;

while (1) {
    pump $h until ($out =~ /NEED_PASSPHRASE (.{16}) (.{16}).*\n/g or
                    $out =~ /GOOD_PASSPHRASE/g);
    if ($2) {
        $in .= $self->passphrase . "\n";
        pump $h until $out =~ /(GOOD|BAD)_PASSPHRASE/g;
        last if $1 eq 'GOOD' or $i++ == 2;
    }
}
finish $h;
my $d = $detach ? 'SIGNATURE' : 'MESSAGE';
$out =~ /(\-\-\-\-BEGIN PGP $d\-\-\-\-\*\-\-\-END PGP $d\-\-\-\-)/s;
my $info = $1;

```

IPC::Run能够可靠地运行，而不像Expect那样需要加入延迟，代码读起来更清晰，并且在大多数平台上能够很好地工作。

一些gpg操作并不需要任何交互，对这些操作，在这个模块早先的版本中使用了Perl的backtick运算符。由于backtick运算符将引发Shell，因此这里存在着安全隐患。如果使用IPC::Run，可以很容易地使用一个短小并且安全的backtick程序来替换对backtick运算符的调用，从而绕过了对shell的调用。这使得在Crypt::GPG中可以完全消除对shell的调用。

```

sub backtick {
    my ($in, $out, $err, $in_q, $out_q, $err_q);
    my $h = start ([@_], \$in, \$out, \$err, timeout( 10 ));
    local $SIG{CHLD} = 'IGNORE';
    local $SIG{PIPE} = 'IGNORE';
    finish $h;
    return ($out, $err);
}

```

一些用户也指出使用临时文件来存放纯文本可能是不安全的。这个问题无需修改代码就可以很容易地攻克，使用带有加密交换扇区的RAM磁盘上的临时文件（例如OpenBSD所提供的）或者一个加密的RAM磁盘，这样纯文本就不会未经加密就写入磁盘了。

当然，我们也可以通过修改代码来彻底避免将纯文本写入到临时文件中，但既然已经存在一个可行的解决方法，因此消除临时文件这个任务就被放到待完成的任务列表而不是立即实现的任务列表中。

在 2005 年年末，新的基于 IPC::Run 的 Crypt::GPG 被上传到 CPAN。它现在能够在更多的操作系统中工作，比起之前基于 Expect 的系统要更加可靠和安全。

看不到的改动

到 2004 年中期，Neomailbo 发布已经一年了，并且吸引了不少付费顾客。Cryptonite 的开发被暂时放在一边，因为当时我正在开发 Neomailbox 各方面的服务，以及其他几个我迫不及待要开始的项目。

但产品在市场上出现肯定是很好的，因为这带来了从竞争到用户反馈的市场影响从而给开发过程施加压力，并且有助于改进和明晰优先顺序。顾客的要求和询问使我能够与用户和市场需求保持密切联系。毕竟满足市场需求是应用程序在商业意义上取得成功的必要条件，因此与市场的交互就成为了开发过程中关键和重要的组成部分。

Cryptonite 被设计为易于维护和修改，是因为我知道，在某些时候它必须以新的方式开始发展，包括回应和预测客户想要的东西。正是从市场中我看到了正在出现的需求：很明显 IMAP 是远程邮箱访问的未来发展方向。

由于 IMAP 有着很多极具吸引力的功能，这使得它成为了一个功能非常强大且实用的邮件访问协议。其中一个最重要的功能是可以用多个客户端访问同一个邮箱，随着计算设备的增多，这个功能变得越来越重要。现在的用户通常都会有桌面计算机、笔记本计算机、掌上计算机和移动电话，所有这些设备都能够访问她的邮箱。

这带来了一个小问题，因为我已经为 Cryptonite 实现了一个完整的邮件存储，并且它并不是基于 IMAP 的。这样就有了两个开发方向：或者实现完全基于 IMAP 服务器的 Cryptonite 邮件存储（一个大任务），或修改 Cryptonite 使它能够使用 IMAP 邮件存储作为后端。实际上，不管用哪种方法，后者都需要实现。

再次，为了降低系统的复杂性并把重点放在系统的主要功能上，我决定不把 Cryptonite 邮件存储开发成一个完全的 IMAP 服务器。取而代之的是，我把它修改成了一个缓存机制，用来缓存用户列出的多部分 (multipart) MIME 消息的 MIME 架构（仅包含结构信息而不包含内容），并且整个邮件由用户读取，这样当下次用户打开以前阅读过的邮件时，Cryptonite 就不需要再回到 IMAP 服务器以取回邮件。

这在两个方面给我了最好的选择。Cryptonite 能得到 IMAP 邮箱的内容，不仅能够同时处理每个邮件的 MIME 结构的完全信息，还能够使得解密的邮件保存在 Cryptonite 邮件存储所支持的 Shadow 文件夹中。

对代码的修改是很简单的。每当用户点击读取一封不在缓存中的邮件时，Cryptonite 将把它缓存在对应 Mail::Folder:: 的 Shadow 文件夹中：

```
my $folder = $session->folder; # 文件名
my $mbox = _opencache($username, $folder); # M::F::Shadow 缓存

unless ($msgnum and grep { $_ == $msgnum } $mbox->message_list) {

    # Message is not in cache. Fetch from IMAP server and cache it.

    my $imap = $self->_open_imapconn($username, $user->password)
        or sleep(int(rand(3))+2), return $self->cluebat (EBADLOGIN);

    $imap->select($folder) or return $self->cluebat (ENO_FOLDER, $folder);
    $imap->Uid(1);

    my ($tmpfh, $tmpnam) =
        tempfile( $self->tmpfiles, DIR => "$HOME/tmp",
                  SUFFIX => $self->tmpsuffix, UNLINK => 1);
    $imap->message_to_file($tmpfh, $msgnum);

    $imap->logout;

    my $parser = new MIME::Parser; $parser->output_dir("$HOME/tmp/");

    $parser->filer->ignore_filename(1); # 不要使用建议的文件名

    seek ($tmpfh, 0, 0);
    my $mail = $parser->parse($tmpfh);

    return $self->cluebat (ENOSUCHMSG, 0 + $msgnum) unless $mail;
    $mbox->update_message($msgnum, $mail);
}
```

按照类似的方式，系统为用户缓存了通过邮件列表视图列出来的所有消息的 MIME 架构。剩余的代码将和以前一样工作，为所有的读操作执行缓存。现在我们获得了 IMAP 的兼容性，同时既没有减弱我的邮件存储提供的功能，也不需要修改太多的主要代码。

我们需要再次开发邮件存储复制，因为对于邮件存储来说，从 Mail::Folder::SQL 切换到 IMAP 服务器意味着 Replication::Recall 不能被用于复制功能。但在任何情况下，Replication::Recall 都不是最优雅或最容易实现复制系统的方法，并且在 Python 中已经重写了 Recall 类库，这使我为早期基于 C++ 的实现而编写的 Perl 接口过时了。

事后，我发现我在复制功能上花了很多的时间，而这个功能却不得不被废弃，或许我最好当时就不去开发复制功能。但另一方面，这个经历的确教会了我很多东西，当我再次准备实现复制系统时，就非常得心应手了。

市场的影响和变动的标准意味着，应用软件总在发展，并且从程序员的观点来看，代码大部分的优美之处在于，当面对需求变更时，能够有多大的灵活性去应对这些变化。Cryptonite 的面向对象体系结构使它的大部分修改都很容易实现。

速度确实重要

在有了 Cryptonite 邮件存储后，系统的性能表现得相当不错，并且大多数邮件存储操作与邮箱的大小无关。但是当切换到 IMAP 后，我注意到在大邮箱中，存在一些显著下降的性能。通过分析显示，性能瓶颈在于纯 Perl 的 Mail::IMAPClient 模块，我使用了这个模块来实现 IMAP 的功能。

我使用了一个快速的基准参照脚本（使用 Benchmark 模块编写的）来证实是否另一个 CPAN 模块，即与 UW C 客户端类库交互的 Mail::Cclient，比 Mail::IMAPClient 更加高效。结果很清楚地指出，我必须使用 Mail::Cclient 再次编写 IMAP 代码：

	速度	IMAP 客户端搜索	C 客户端搜索
IMAP 客户端搜索	39.8/s	--	-73%
C 客户端搜索	145/s	264%	--
	速度	IMAP 客户端搜索	C 客户端搜索
IMAP 客户端搜索	21.3/s	--	-99%
C 客户端搜索	2000/s	9280%	--

我本应该在使用 Mail::IMAPClient 编写代码之前首先考虑进行不同模块之间的基准测试。我最初避免使用 C 客户端类库是因为我想使构建过程尽可能地简单，而 Mail::IMAPClient 的构建过程无疑比 Mail::Cclient 的构建过程要简单。

幸运的是，从前者到后者的迁移工作相当简单。对某些操作来说，我注意到 IMAP 客户端比 C 客户端工作得更好，并且在性能上也没有什么劣势，因此 Cryptonite::Mail::Service 现在使用两个模块，并且每个模块都发挥其优势。

像 Cryptonite 这样的程序是从来就没有“完成”的那一刻，只不过现在的代码变得更加成熟、健壮、功能全面并且非常高效以服务于它的目的：为数以万计的并发用户提供安全、直观和可响应的电子邮件系统，并且帮助他们有效地保护通信隐私。

人权中的通信隐私

我在本章开始的时候提及提供广泛的安全通信技术是维护人权中一个非常有效的手段。

因为这个认识是Cryptonite项目的动力源，我希望最后再给出一些支持这一观点的资料。

密码技术可以使用在其他事情中（注 7）：

- 为一些活动家、非政府组织（NGO）和记者提供生命保护（注 8）。
- 确保被审查的新闻和有争议的想法可以被相互交流。
- 保护告密者、家庭暴力的受害者，目击证人的匿名性。
- 通过在全球范围内自由地和无约束地交流思想、商品以及服务来促进民主社会。

由被称为Cypherpunks的程序员组成的各种开发小组已经在开发隐私性的软件方面工作了许多年，其目的是提高数字时代的个人自由和主权。一些密码软件已经成为目前世界运作的基石。其中包括 Secure Shell (SSH) 远程终端软件，它从本质上保证互联网的安全，同时，Secure Sockets Layer (SSL) 编码软件使得在线商务更为安全。

但这些系统的目标都非常具体：分别是安全的服务器连接和安全的网上信用卡交易。两者都以人机交互上的安全为重点。在今后的几年中，将需要部署更多特定于人与人之间的交互的密码技术以应对无处不在的监视所带来的危险（这将“导致文明社会很快地终结”（注 9）。

易用的、安全的网络邮件系统是一项有用的技术——在历史中，它第一次使全球范围内的人与人之间的长距离通信变得安全和隐私，而通信的双方永远都无需面对面。

程序员与文明

计算机如此复杂，在它的运算矩阵中包含了有机生活——地球以及地球上的文明——可以用一种功能强大的方式来对其重新编程，只需使用修改人类文明的简单代码，然后再重新接入到社会这个操作系统中。

代码改变世界已经许多次了。回顾一下有基因序列分析软件在医学上带来的进步，商业软件对大型企业和小型企业的影响，工业自动化软件和计算机建模带来的革命，或者在互联网上的多次革命：电子邮件，网络，博客，社会网络服务，VoIP。显然，在当今这个时代，许多事件都是关于软件创新的故事。

注 7：参见 <http://www.idiom.com/~arkuat/consent/Anarchy.html> 和 http://www.chaum.com/articles/Security_Without_Identification.htm。

注 8：参见 <http://philzimmermann.com/EN/letters/index.html>。

注 9：Vernor Vinge, A Deepness in the Sky. Tor Books, 1999。

当然，与其他所有技术一样，代码也是一把双刃剑，可能会增加或者减少社会“回归到暴力”（注 10）的可能性，它既可以成为破坏隐私的技术，为专制者提供更加有效的反隐私技术的工具，而另一方面，它也可以用来提高和促进人权。任何一种代码都会通过改变基本的社会行为来影响人类社会的核心，例如改变自由言论权利等。

有趣的是，即使使用像公钥加密算法这样的具体技术，其实现上的选择可以极大地改变文化的基石。例如，基于 PKI 的实现强加了独裁属性，例如在某项技术上的中央集权和识别需求，而这个技术的价值依赖于它所缺乏的这些属性。除此之外，PKI 方法中的密钥认证功能要弱于信任网络（信任网络同时并未减弱公钥密码的其他重要功能，比如分布式部署）。

我认为，作为代码的编写者，在很大程度上依赖程序员的责任感来使得我们编写的代码不仅在设计和实现上是漂亮的，同时还使得代码对社会环境产生良好的效果。这就是我为什么认为免费的代码是非常优美的原因。它把计算机技术置于一个最庄严神圣的用途：保护人权和人的生命。

法律和国际人权条约到目前为止在保护人权上只能达到现在的水平。历史证明这些保护非常容易被绕过或者忽略。而另一方面，如果密码系统能够得到非常细致的实现，那么就可以为人权和开放思想提供非常坚实的盾牌，并且最终可以使全世界的人们在保护隐私和自由中进行通信和交易。

是否能实现这个全球化的、高度保护的开放社会主要取决于我们这些计算机的主宰之神。

注 10：我所说的“回归到暴力”指的是侵犯人权的社会的和经济的动机。《The Sovereign Individual》一书的作者指出，“了解社会如何发展的关键在于了解哪些因素决定了使用暴力的代价和回报。”

第12章

在 BioPerl 里培育 漂亮代码

Lincoln Stein

过去十年，生物学已发展为一门信息科学。新技术为生物学家们提供了前所未有的窗口，使他们得以探查动植物细胞内错综复杂的过程。DNA测序机让我们快速读出完整的基因组序列；微阵列技术得出组织生长过程中复杂基因表达的快照；共聚焦显微镜生成三维电影，用以跟踪当癌症前期组织转为恶性肿瘤时细胞结构的变化。

这些新技术不断生成万亿字节的数据。这些数据全都要被过滤、储存、处理，和数据挖掘。计算机科学和软件工程在生物学数据管理上的应用叫做生物信息学。

在华尔街，生物信息学在很多地方同软件工程相似。像金融领域的软件工程一样，生物信息学家们需要行动迅速：他们得在没有多少时间做需求分析和设计的条件下让程序运行起来。数据集庞大易变，可用期以月记而非以年记。因为这样，大多数生物信息程序员青睐敏捷开发技术，比如极限开发，以及允许快速构建原型和部署的工具包。同金融领域一样，也非常强调数据可视化和模式识别。

BioPerl 和 Bio::Graphics 模块

BioPerl是生物信息学家为生物信息学家开发的快速开发工具包之一。它是一套包含可复

用代码的大规模开源类库，适用于 Perl 编程语言。BioPerl 提供了处理大多数常见生物信息学问题的模块，包括 DNA 和蛋白质分析，进化树构建和分析，遗传数据的解读，当然还包括基因组序列分析。

BioPerl 允许软件工程师快速创建复杂管道，用于处理、过滤、分析和整合，已经形象展示大规模生物数据集。因为开源社区对它做了大量测试，用 BioPerl 搭建的应用更可能一次成功，而且由于所有主要平台都有 Perl 解释器，用 BioPerl 写的应用可以在 Microsoft Windows, Mac OS X, Linux, 和 Unix 机器上运行。

本章讨论 BioPerl 里基因组图谱的绘制模块 Bio::Graphics。它解决的是形象表示基因组和基因组功能注释的问题。基因组包括一组 DNA 序列，每一序列为一串字母 (A, G, T, C)，也就是核苷酸，也称作碱基对 (bp)。某些 DNA 链可以很长。比如，人类基因组包含了 24 条 DNA 序列，对应染色体 1 到 22，以及 X 和 Y 染色体。其中最长 1 号染色体大概有 150 000 000 个碱基对 (150 兆碱基)。

这些 DNA 序列中隐藏着多个区域。它们在细胞新陈代谢、再生、防御以及信号传递中扮演不同角色。例如，1 号染色体的 DNA 序列中某些片段是蛋白质编码基因。这些基因被细胞“预录”到较短的 RNA 序列中。这些 RNA 序列从细胞核转运到细胞质，再被转译到蛋白质。这些蛋白质负责产生能量，把养份运进或运出细胞，制造细胞膜，等等。这些 DNA 序列的其他区域用于调控：当调控蛋白质结合到某个具体调控部位，附近的蛋白质编码基因就被“打开”，开始预录。某些区域对应寄生 DNA：序列上可以半自主复制自己并在基因组游移于短小区域。有些区域还有未知的作用。我们知道它们重要，因为它们在人类和其他组织的漫长进化过程中被保存下来，只不过我们还不理解它们的作用。

找到并解读基因组里的功能区域称为功能注释。这已成为基因组计划的主要着眼点。基因组的功能注释通常产生比 DNA 序列本身多得多的数据。整个人类基因组拥用 3GB 未经压缩的数据，而它当前的功能注释用去了很多 TB（参看第 13 章）。

Bio::Graphics 输出例子

要自动对准基因组“有趣”的区域，生物学家需要形象展示多个功能注释如何互相关联。例如，如果一个调控区在空间上靠近蛋白质编码基因，并和进行迥异的生物中的保留区域相互重叠，那么这个调控区就可能更为重要。

Bio::Graphics 允许生物信息软件程序员快速形象展示基因组和它的所有功能注释。它可以单独使用，生成各种图像格式的静态区域图像（包括 PNG、JPEG 和 SVG）。它也能与 web 或者桌面应用整合，提供用于交互的滚动、缩放和数据探索功能。

图 12-1 展示了 Bio::Graphics 生成的一副图像。该图像显示了线虫（一种吞吃土壤的蠕虫）基因组的一片区域，表现出 Bio::Graphics 生成图像的几个典型方面。图像被垂直划分为一系列水平轨道。顶端轨道包括一条从左到右的水平尺子。尺子单位是千碱基 (k)，意为 1000 个 DNA 碱基对。图片展示的区域刚好从线虫 I 染色体第 160 000 位开始，一直延展到 179 000 位之后，一共覆盖了 20 000 个碱基对。图上右 4 条功能注释轨道。每条展示了逐渐复杂的可视化表示。

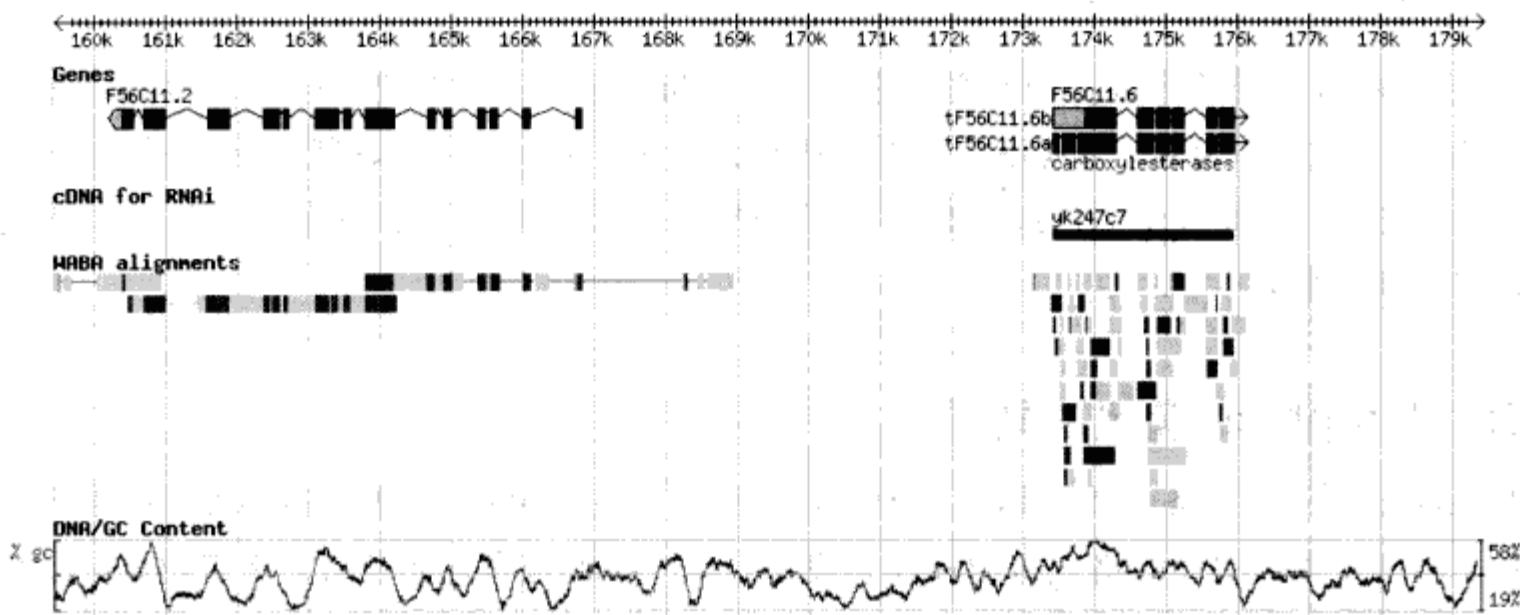


图 12-1：Bio::Graphicsshengcheng1 的简单图像

原始图像上了鲜亮的颜色，不过在这里为了出版需要，缩减为灰度表示。最简单的轨道是“RNAi 的 cDNA”，展示了一种试验试剂的位置。生物研究社区创制了这种试剂，用于研究线虫基因的调控。图像右边包含单个功能注释，名为 *yk247c7*。该注释包括一块黑色矩形，大致从 173,500 位开始，延展到大致 176,000 的位置。这对应了真正 DNA 覆盖该区域的那一段。研究人员可以从生物技术用品公司订购这一段，并试验性地改变覆盖该区域的基因的活动——这种情况下是 *F56C11.6*。

“WABA 对齐”轨道展示了稍微复杂的信息。比较线虫和另一蠕虫的基因组里相仿区域时会生成定量数据。该轨道形象表现了这些数据。高度相似的区域呈深灰色。略有相似的区域呈浅灰色。中等程度相似的区域呈中度灰色。

“DNA/GC 内容”轨道展示了连续变化的定量信息。该图记录了 G、C 核苷酸同 A、T 核苷酸在核苷酸序列的某滑动窗口中的比例。该比例大致同对应的基因组区域包含蛋白质编码基因的机会相关。

“基因”轨道包含最为复杂的数据：两条蛋白质编码基因的位置。每条基因都有一内部结构，指明哪部分将蛋白质编码（深灰色，不过原图为蓝色），以及哪部分拥有其他功

能（略浅的灰色）。注意最左边基因（F56C11.2）的编码同WABA对齐轨道里深灰色高度相似区域良好对应。这是因为基因里蛋白质编码区域往往在不同物种间高度保留。

名为F56C11.6的基因用功能“羧酸脂酶”注释，表明它与一族负责炭代谢核心部分的蛋白质相关。另外，它用两种替代形式表示，说明它可以将多个截然不同的蛋白质编码。这两种替代形式被放在一起，用同一个独特的标签。注意该基因下有多个队列。这反映了该基因属于一大族相关基因，而且每个相关基因来源于不同队列。

真正的DNA核苷酸序列并没有在图上表示，因为实际上不可能把一行20,000碱基对压缩到800像素里。但是，当观察较小的基因组片段时，Bio::Graphics能画出实际的字母，并修饰它们（比如改变颜色或加亮）。这样可以显现出我们感兴趣的特性（译注1）的起始和结束位置。

Bio::Graphics 满足的需求

Bio::Graphics的任务是将一系列基因组功能注释（在BioPerl术语里叫特性）输出到图形文件里。图形文件的格式可由程序员设定。每一特性包含起始位置，终点位置，以及方向（向左、向右或者不指向任何方向）。特性可以同其他属性联系起来，比如名字，描述，以及数值。特性也可以包含内部结构，甚至多层内嵌特性。

当设计Bio::Graphics时，我必须处理如下问题：

问题的解决方案是开放的

我们已经有了大量基因组功能注释类型，而该数量每天都在增长。虽说许多功能注释可以用带不同颜色的简单矩形画出，许多——尤其是定量的那些——的表现方法相当复杂。而且，不同的生物信息学家也许想用不同的方式表示同一功能的注释类型。比如，基因的表达有许多不同方式，每种方式又适用于不同的情形。

为了配合问题的开放特质，我想让如下两项功能简单易行：向Bio::Graphics里加入新的视觉表示方法，以及扩展已有的视觉表示方法。每种表示必须具有高度的配置能力。比如，程序员应该可以对高度、重量、边界颜色和哪怕是简单矩形的填充颜色实行精细的控制。而且，程序员应该能够根据具体情况修改每个特性呈现的方式。

特性密度

某些基因组特性非常致密，而其他的则相对稀疏（比较图12-1的“基因”轨道与“WABA队列”轨道）。特性也可在空间上重叠。有时你想让重叠部分彼此模糊，而

译注1：这里的“特性”是Bio::Graphics的术语，特指一系列基因组功能注释。

有时你希望控制碰撞，通过把重叠的特性垂直地移上移下，让它们彼此清楚展现。问题还不止如此。特性可以包含重叠的子特性，所以碰撞控制和空间布局需要以递归的方式工作。

我认为碰撞控制能在依靠上下文的情况下激活或关闭。比如，如果正在展示的区域有上千重叠的特性，碰撞控制也许会让某一轨道繁忙得无法管理，因此必须被关闭。程序员应能控制在什么时候上下文敏感的碰撞控制其起作用，或者彻底取代它。

处理尺度

我希望 Bio::Graphics 能够画出具有高清细节的图片，展现基因组中包含 500 碱基的区域。我也希望 Bio::Graphics 能画出跨度为一亿五千万核苷酸的基因组全貌。要实现这些，特性的视觉表示必须根据当前尺度做出智能变动。在 500 碱基的尺度上，你可以显示基因的内部结构。在一百万碱基的尺度上，你只能显示该基因的起始和结束位置。在一亿碱基的尺度上，基因融聚为单个黑色条块，而你应该切换到把基因密度表示为灰度强度的表示方式。

对交互式 web 应用有用

我想让 Bio::Graphics 适用于基于 web 的基因组浏览器的后台。具体说，我希望最终用户能够通过点击特性的图形表示来打开下拉菜单，链接到其他网页，查看工具提示等等。要做到这些，Bio::Graphics 必须相对快速地即时生成图像。它还必须跟踪记录它呈现特性的位置（如有必要，也包括它们的子特性），以便创建可以传给程序员的图像映射图。

独立于图形格式

我想让 Bio::Graphics 既适用于生成用来嵌入网页的显示屏等级、低分辨率图像，也适用于生成高分辨率、出版物级别的图像。为了做到这些，Bio::Graphics 处理布局逻辑的部分必须同生成图像的部分分开。它最好既能输出位图也能输出矢量图格式。

独立于数据库模式

生物信息学社区已经设计了成打用于管理基因组功能注释数据的数据库格式，从简单的文件到复杂的关系数据库。为了求得最大效用，我不想把 Bio::Graphics 捆绑到任何具体的数据库模式上。调用 Bio::Graphics 来呈现普通文件描述的基因组区域应该同呈现用 Oracle 数据库描述的基因组片段一样简便。

Bio::Graphics 的设计流程

我对形式化的设计工程不感兴趣。相反，我通常写出一小段伪代码，用来描述我希望代

码如何运行（一段“代码故事”），把玩一下输入和输出格式，写一点代码。如果我对系统怎么配合运行不满意，就回头修改代码故事。对比玩具应用大的程序来说，决定是否采用某设计时，我会实现系统的一小部分，然后在独立运行的程序里测试。我把笔记保存在“意识流”文本文件里，并且常常向CVS代码库提交代码。我尽力让所有代码看上去优雅动人。如果代码不优雅，设计上肯定有问题，于是我重回画板。

设计程序员与模块的交互方式

我的第一项设计任务就是搞明白典型Bio::Graphics应用的流程。我从例12-1展示的代码故事开始：

示例 12-1：BioPerl::Graphics用法的基本故事（伪代码）

```
1 use Bio::Graphics::Panel;  
2 @first_set_of_features = get_features_somewhow();  
3 @second_set_of_features = get_more_features_somewhow();  
4 $panel_object = Bio::Graphics::Panel->new(panel_options...)  
5 $panel_object->add_track(\@first_set_of_features,  
                           track_options...);  
6 $panel_object->add_track(\@second_set_of_features,  
                           track_options...);  
7 print $panel_object->png;
```

以上代码故事从引入主要的Bio::Graphics对象类开始，Bio::Graphics::Panel（第一行）。该对象，我想，得保存图像的全局配置选项，比如生成图表的维数信息，它的尺度（碱基对每像素），而且它也是与用户交互的主要对象。

代码故事接着靠两个调用得到序列特性的数组（2到3行）。为了保持相对于特性数据库的独立性，我决定让Bio::Graphics操作将从数据库取出来的数据列表。幸好，BioPerl已经通过通用界面支持多种功能注释数据库。序列特性用名为Bio::SeqFeatureI的界面表示。该界面规定了所有序列都得支持的一套方法。大多数方法都很直观。例如，\$feature->start()得到特性的起始位置，\$feature->end()得到特性的终止位置，而\$feature->get_SeqFeatures()得到特性所有的子特性。BioPerl用名为Bio::SeqFeature::CollectionI的界面从数据库取出特性。这提供了通过查询顺序或随机读取特性的标准API。

代码故事接着调用Bio::Graphics::Panel->new()（第4行），创建一个新的面板对象，然后调用add_track()两次（5到6行）。这样就加入两条轨道，分别对应第一和第二套特性。每次调用add_track()都接受一个初始参数。该参数包含指向将被加入的特性数组的引用。接下来的额外参数控制着轨道的外观。

代码故事的最后一步是把面板转换为PNG文件，并且立即把结果打印输出到标准输出。

刚开始这故事看上起还不错，不过当我进一步思考时，我意识到 API 有一些根本缺陷。最大的问题是它强迫程序员在开始构造图像前把所有特性载入内存。不过，通常从文件或数据库里一次读出一条序列特性比较方便。Bio::SeqFeature::CollectionI 的几种实现允许你做下面的事：

```
$iterator = Bio::SeqFeature::CollectionI->get_seq_stream(@query_parameters);
while ($feature = $iterator->next_seq) {
    # 对 feature 进行一些操作
}
```

另一问题是一旦你调用了 \$panel->add_track(), 你不能再改动该轨道的配置设置。但是，可以想像，某些情况下程序员会希望预先加入一堆轨道，以后再回头改动早先加入轨道的配置。

最后，对我来说 add_track() 似乎不太灵活，因为它迫使程序员按固定顺序加入轨道（自顶向下）。应该有办法在任意位置插入轨道。

这些想法使得我们必须创建 Track 对象：

```
$track1 = $panel_object->add_track(\@set_of_gene_features,
                                    track_options...);
$track2 = $panel_object->add_track(\@set_of_variation_features,
                                    track_options...);
```

然后我们便可以向已有的轨道加入特性：

```
$track1->add_features(feature1, feature2, feature3....)
```

或者用这种方式改变它的配置：

```
$track1->configure(new_options)
```

这导致了示例 12-1 所示的新代码故事：

示例 12-2：BioPerl::Graphics 里轨道用法的故事（伪代码）

```
1 use Bio::Graphics::Panel;
2 $panel_object = Bio::Graphics::Panel->new(panel_options...);
3 $track1 = $panel_object->add_track(bump_true, other_track_options...);
4 $track2 = $panel_object->add_track(bump_true, other_track_options...);

5 $collection = Bio::SeqFeature::CollectionI->new(@args);
6 $iterator    = $collection->get_seq_stream(@query_parameters);
```

```

7 $genes=0; $variations=0;
8 while ($feature = $iterator->next_seq) {
9     if ($feature->method eq 'gene') {      直接将不一致的头部移至头部部分
10        $track1->add_feature($feature);    跳过头部的头部部分
11        $genes++;
12    } elsif ($feature->method eq 'variation') {  跳过头部的头部部分
13        $track2->add_feature($feature);    跳过头部的头部部分
14        $variations++;
15    }                                跳过头部的头部部分
16 }
17 $track1->configure(bump_false) if $genes > 100;  跳过头部的头部部分
18 $track2->configure(bump_false) if $variations > 100;
19 print $panel_object->png;  不再关心头部部分

```

在该版本的故事里，我们首先创建两条轨道，但一开始并未提供任何特性（3到4行）。但是，我们给 `add_track()` 提供了一套选项，包括初始值设为真的“碰撞（bump）”选项。在设计这个故事时我假设碰撞选项为真时激活碰撞控制，而碰撞选项为假时关掉碰撞控制。

我们接着用 BioPerl 创建特性集合对象。一般来说这会连到某种数据库，然后创建一个查询（5到6行），返回结果的迭代器。我们然后反复调用 `$iterator->next_seq`，以便返回满足查询的所有特性（8到16行）。对返回的每条特性，我们用 Bio::SeqFeatureI 中名为 `method` 的方法询问它的类型。如果特性的类型是 `gene`（基因），我们就把它加到 `track1` 中，并递增基因的计数器。如果特性的类型是 `variation`（变种），我们把它加到 `track2`，并递增不同的计数器。

加入所有特性后，我们查询加入的基因数目和变种的数目。如果其中任何一类特性的总数超过 100，我们调用对应轨道的 `configure()` 方法，从而把碰撞选项设为假（17-18）。这段代码的目的是当特性数目可以管理时，打开碰撞控制，以便重叠特性可以移上移下，避免彼此覆盖。如果轨道包含大量特性，碰撞控制也许会产生过高的轨道，所以碰撞选项被设为假，以便重叠特性能够在空间上重叠。

阅读本章代码

Bio::Graphics 和 BioPerl 用 Perl 写成。Perl 是门看上去像 C 或 Java 的语言，但其实包含许多晦涩的特质。如果你不知道 Perl，也不用紧张。通读代码例子，大致了解设计的逻辑就行了。为了帮助你理解文章脉络，这里总结 Perl 句法里最诡异的部分：

\$variable_name

以美元符号 (\$) 开头的变量名是标量变量。它包含单值的字符串或数。

@variable_name

用 @ 开头的变量名是有序数组。它包含以数组里数值位置作索引的多项字符串或数。当定位数组成员时，我们把索引放到方括号里：

`$foo[3] = 'element three';`

变量名前有个 \$，因为单个元素还是标量。

在子程序和方法定义里，名为 @_ 的特殊数组包含了传给该例程的参数列表。

%variable_name

以百分号 (%) 开头的变量名代表无序散列。它包含多个字符串或数（都是散列值）。这些值用字符串索引（散列键）。当把一列键 / 值对赋给散列时，用如下表示方法：

```
%clade = (monkey=>'mammal',
           frog=>'amphibian',
           gazelle=>'mammal');
```

定位散列成员时，我们把键放在花括号里：

```
print "The clade of a monkey is ", $clade{monkey};  
$object->method('arg1', 'arg2', 'arg3')
```

箭头 -> 表示面向对象的方法调用。对象存在标量变量 \$object 里。方法调用接受零个或多个参数。

如果没有参数，括号可以省略。这对 C 或者 Java 程序员来说比较古怪。

```
$object->method;
```

方法定义内，对象通常保存在名为 \$self 的标量里，尽管这只是编码风格的问题。

设置选项

到目前为止，我仍没有想出应该如何设定选项。为了同 BioPerl 的编码风格保持一致，我决定把选项放在标签 / 值对里，以 *-option_name=>option_value* 的格式传递。例如，下面的方法调用创建了一个带特性的轨道。图像高有 10 像素，蓝色背景，并且打开了碰撞控制 (bump)。

```
$track1 = $panel_object->add_track(\@features,  
                                     -height      => 10,  
                                     -bgcolor     => 'blue',  
                                     -bump        => 1);
```

随后，我们应该能通过调用 `configure()` 改变所有选项。这个例子关掉了碰撞控制：

```
$track1>configure(-bump => 0);
```

最后，我通过支持代码引用（一种回调的类型）扩展了轨道配置选项的能力，不过模块的第一次迭代没有该项功能。我等会儿在本章会讨论这个问题。

轨道对象应该支持什么配置选项？我迅速想了少许标准的轨道选项。其中最重要的是 `glyph`：

```
-glyph => 'glyph_type'
```

如前所述，我希望为功能支持多种视觉表示。上述的 `-glyph` 选项是终端程序员利用这些表示的途径。例如，`-glyph=>'box'` 应该把特性呈现为简单矩形，`-glyph=>'oval'` 应该把特性呈现为尺寸合适的椭圆，而 `-glyph=>'arrow'` 应该画出箭头。

除了 `-glyph`，最初的设计还包括了如下选项：

`-fgcolor`

特性呈现在轨道里时，前景（画线）的颜色

`-bgcolor`

特性呈现在轨道里时，背景（填充）的颜色

`-bump`

是否打开碰撞控制

`-label`

是否在每项特性上方打印它的名字

`-description`

是否在每项特性下打印它的描述

`-height`

每项特性的高度

`-key`

整个轨道的标签

`-tkcolor`

轨道的背景色

我认为花俏的画图方式或许有着简单方式所不具备的特别选项，所以我设计的代码库必须能够让传给 `add_track()` 的选项列表可以扩展。

面板对象也需要接受选项，以便控制诸如图像宽度，像素和碱基对转换比例一类的设置。

我想出了少量面板专用的参数，包括：

-width
-length

图像宽度，以像素为单位

-length

序列片段的长度，以碱基对为单位

如示例 12-3 所示，我可以写出具体展示 Bio::Graphics 调用的代码故事：

示例 12-3：展示 BioPerl::Graphics 用法的详细的故事（伪代码）

```
1 use Bio::Graphics::Panel;
2 $panel_object = Bio::Graphics::Panel->new(-width => 800,
3                                              -length => 50000);
4 $track1 = $panel_object->add_track(-glyph => 'box',
5                                     -height => 12,
6                                     -bump   => 1,
7                                     -key    => 'Protein-coding Genes');
8 $track2 = $panel_object->add_track(-glyph => 'triangle',
9                                     -height => 6,
10                                    -bump   => 1,
11                                    -key    => 'Sequence Variants');
12
13 $collection = Bio::SeqFeature::CollectionI->new(@args);
14 $iterator   = $collection->get_seq_stream(@query_parameters);
15
16 $genes=0; $variations=0;
17 while ($feature = $iterator->next_seq) {
18
19     if ($feature->method eq 'gene') {
20         $track1->add_feature($feature);
21         $genes++;
22     } elsif ($feature->method eq 'variation') {
23         $track2->add_feature($feature);
24         $variations++;
25     }
26 }
27
28 $track1->configure(-bump => 0) if $genes      > 100;
29 $track2->configure(-bump => 0) if $variations > 100;
30
31 print $panel_object->png;
```

选择对象的类

最后一项主要的设计任务是选择与程序员交互的主要对象的类。从代码故事看来，起初有两个主要的类：

Bio::Graphics::Panel 基于类 Panel，负责将图形界面与 Bio::Graphics 对象对齐。

该类的对象代表整个图表，而且一般被划分为一系列水平轨道。Bio::Graphics 负责确定画图区内每条轨道的位置，并负责把特性坐标（用碱基对表达）转换为图像轮廓坐标。

Bio::Graphics::Track

该类的对象代表组成面板的轨道。轨道主要负责确定图像轮廓的位置并画出它们。

图符怎么办？对我来说，自然的选择为图符应该是对象，而且应该包含绘制它们自己的内部逻辑。所有图符应该继承自通用类 Bio::Graphics::Glyph 对象。该类的对象知道如何做基础的绘制。当调用 Track 对象的 add_feature() 方法时，它通过调用如下 Glyph 构造函数来创建新的图符：

```
$glyph = Bio::Graphics::Glyph->new(-feature=>$feature);
```

接着，当 Track 需要绘制自身时，需要类似下面的 draw() 方法：

```
sub draw {
    @glyphs = $self->get_list_of_glyphs();
    for $glyph (@glyphs) {
        $glyph->draw();
    }
    # draw other stuff in the track, for example, its label
}
```

该子程序先获取我们在调用 add_features() 时创建的 Glyph 对象列表。然后它调用每个 Glyph 对象的 draw() 方法，绘制出它们自己。最后，它绘制轨道特定的信息，比如说轨道的标签。

随着我对 Bio::Graphics::Glyph 的深入思考，我意识到它们需要体现出一些智能性。回想一下序列的特性可以有内部结构，包括子特性，子特性的子特性等，并且内部结构的每个组件都需要通过控制碰撞来安排，然后再根据用户的选择绘制出来。这种布局和绘制行为非常类似于 glyph，并且让 glyph 包含 subglyph 的方式类似于 feature/subfeature 结构的做法也是很合理的。下面给出了 glyph 的 new() 程序：

```
sub new {
    $self = shift; # 得到 self
    $feature = shift; # 得到特性
    for $subfeature ($feature->get_SeqFeatures) {
        $subglyph = Bio::Graphics::Glyph->new(-feature=>$subfeature);
        $self->add_subpart($subglyph);
    }
}
```

我们为特性的每个子特性创建新的子图符，并且把子图符加入内部列表。如果子特性自己有子特性，因为我们递归调用 new()，它会创建另一层嵌套的图符。

要绘制自身和它的所有子图符，顶层图符的绘制程序类似下面的代码：

```
sub draw {
    @subglyphs = $self->get_subparts( )
    for $subglyph (@subglyphs) {
        $subglyph->draw;
    }
    # draw ourselves somehow
}
```

这段伪代码调用 get_subparts() 来得到我们的构造函数创建的所有子图符。它循环访问每一子图，并调用该图的 draw() 方法。然后这段代码执行自己的绘制程序。

此时，对 Glyph 的 draw() 伪代码例程和之前展示的 Track 的 draw() 方法本质上一模一样感到震惊。我想我可以统一这两个类。要做到这点，只需要让 add_track() 创建和管理一个与这个轨道关联的内部特性对象。之后对 add_feature() 的调用实际上是把子特性加到特性里。

我用一些测试代码试验了一下，发现这个主意行之有效。除了免除重复绘制代码这一好处外，我还能整合所有传递和配置轨道以及图符选项的代码。因此，轨道变为 Bio::Graphics::Glyph 的子类，名为 Bio::Graphics::Glyph::track，而面板对象 Panel 的 add_track() 方法如下所示（多少有些简化）：

```
sub add_track {
    my $self = shift;
    my $features = shift;
    my @_options = @_;
    my $top_level_feature = Bio::Graphics->new(-type=>'track');
    my $track_glyph =
        Bio::Graphics::Glyph::track->new(\@options);
    if ($features) {
        $track_glyph->add_feature($_) foreach @_features;
    }
    $self->do_add_track($track_glyph);
    return $track_glyph;
}
```

第一份代码故事中，调用者把一列特性传给 add_track()。为满足这一要求，我允许第一个参数是一列特性。在实际代码中，我对第一个参数做动态类型检查，由此把一列参数与第一个选项区分开。这允许使用第一个代码故事的风格调用 add_track()：

```
$panel->add_track(\@list_of_features, @options)
```

或使用第二个代码故事的风格：

```
$panel->add_track(@options)
```

`add_track()` 接着用轻量级的特性类创建类型为`track`的新特性。该轻量级的类是我为`Bio::Graphics`而写的（我必须这样做，因为标准的 BioPerl 特性对象对内存和性能的要求都高）。该类被传给`Bio::Graphics::Glyph::track`的构建函数。

如果有特性列表，该方法循环访问列表并调用轨道图形的`add_feature()`方法。

最后，`add_grack()`方法把轨道加到已被加入面板的内部轨道列表，并返回轨道图形。该轨道的`add_teature()`方法用于创建包含在轨道内的子图符。它要么被图符的构造函数调用，要么以后被程序员在调用`$track->add_feature()`时调用。概念上讲，代码如下：

```
sub add_feature {  
    my $self = shift;  
    my $feature = shift;  
    my $subglyph = Bio::Graphics::Glyph->new(-feature=>$feature);  
    $self->add_subpart($subglyph);  
}
```

我展示了在程序中直接调用`Bio::Graphics::Glyph`的构造函数，但在实际情况中会有许多不同类型的图符，所以`Bio::Graphics::Glyph`子类的选取必须在运行时根据用户提供选项进行。下一节我将讨论我是怎么决定这样做的。

选项处理

设计过程的下一步是指出如何动态创建图符。这是处理用户自定义选项这类一般问题的精要之处。还记得前面的代码故事里我想在创建轨道时制定选项吧：

```
$panel->add_track(-glyph => 'arrow',  
                    -fgcolor => 'blue',  
                    -height  => 22)
```

该例让面板创建包含箭头轮廓的轨道。轨道的前景色为蓝色，高度为 22 像素。如前一节所述，`add_track()`会创建直接在代码里设定的轨道图符。图符的类型为`Bio::Graphics::Glyph::track`。`add_track()`会将这些选项传给`Bio::Graphics::Glyph::track`的构造函数。之后，当代码调用轨道图符的`add_feature()`方法时，该方法会为每个将要显示的特性创建子图符。

不过，这样遗留下了三个未解决的问题：

1. 轨道图符的 `add_feature()` 方法如何知道创建哪种类型的字图符？

我们希望根据用户喜好创建不同类型的图符，用来显示不同类型的特性。因此，在前面例子里，用户想根据 `-glyph` 选项的值，向轨道中加载一系列箭头图符。上节的 `$track->add_feature()` 伪代码在代码里直接调用 `Bio::Graphics::Glyph->new()`，但是在正式代码中，我们希望动态选取合适的图符子类，比如说 `Bio::Graphics::Glyph::arrow`。

2. 这些子图符如何知道创建它们自己的子图符？

前面提到过，特性可以包含子特性，并且每一个子特性都是由主要图符的子图符来表示的。上述例子中，轨道图符均先通过 `-glyph` 选项值创建了一系列箭头图符。然后，箭头图符将负责创建它所需要的任何子图符。而这些子图符又将创建它们自己的子图符，如此以往。那么，箭头图符如何来确定创建什么类型的子图符呢？

3. 如何将其他选项传给新建图符？

例如，上例中什么对象跟踪记录 `-fgcolor` 和 `-height` 选项的值？

因为选取图符类型是处理配置选项的特例，我决定先解决处理配置选项的问题。我最初的想法是每个图符负责管理自己的选项，不过我很快就对这一想法失去了热情。由于一条轨道可能包含上千图符，让每个图符保存一份完整的配置效率会非常低。我也想过把选项存在 `Panel`（面板对象里，不过这感觉上也不对，因为面板有它自己的选项，同轨道专有的选项截然不同）。

最终我想出的解决方法是创建一系列类型为 `Bio::Graphics::Glyph::Factory` 的图符“工厂”。每当一条轨道被创建，`Panel` 对象就创建对应的工厂，用调用者要的选项将其初始化，并调用工厂里的方法来获取选项。因此，如果面板有四条轨道，也就会有四个 `Bio::Graphics::Glyph::Factory` 对象。

当我想出图符工厂这个方法后，如何创建合适的图符和子图符类型的问题立刻变得很简单了。工厂存储用户选择的图符（如箭头）以及其他选项。工厂包含名为 `make_glyph()` 的方法。该方法在需要时创建图符和子图符。创建时该方法利用存储的选项来决定到底用什么图符子类。

这个方法意味着一条轨道内所有图符共享相同的类。也就是说，如果某个具体特性包含三层嵌套子特性，而用户选择用 `arrow`（箭头）图符绘制该轨道内的特性，那么每个箭头图符都包含箭头子图符，而这些子图符也包含它们自己的箭头子图符。这听上去有着一系列的局限性，但其实是有道理的。通常，图符和它的部件总在一块儿。让它们允许

我们把相关代码放在同一地方，而且，图符可以通过覆盖它们的 new() 构造函数来跳出该项限制，创建它们选取的任意类型。

Bio::Graphics::Glyph::Factory 类的最终版本只有几项方法：

构造函数

构造函数创建一个新的工厂：

```
$factory = Bio::Graphics::Glyph::Factory->new(-options=> \%options, -panel=> $panel);
```

在构造过程中，该函数将获得面板的 add_track() 方法传递进来的一组选项，并将这组选项存储在内部。工厂对象同样也能获取面板的拷贝。在我加上这项功能后，工厂对象就能够提供关于面板的信息，比如面板的尺度。

我们通过对散列（一个“名字/值”的 Perl 字典）的引用来传递这组选项。面板对象 Panel 的 add_track() 方法也承担一项小任务：把传给它的一列形如 -option->\$value 的名值对转换为散列，以便传给工厂的 new() 方法。

option() 方法

给定选项名，工厂对象查询它的值，并将结果返回：

```
$option_value = $factory->option ('option_name')
```

如果该名字没有对应选项，option() 方法会检查有没有对应的缺省值，并将其返回。

make_glyph() 方法

给定一列特性，工厂创建类型合适的一列图符：

```
@glyphs = $factory->make_glyph($feature1,$feature2,$feature3...)
```

现在我们来看一下简化后的 Bio::Graphics::Glyph::Factory 代码：

```
1 package Bio::Graphics::Glyph::Factory;
2 use strict;
3 my %GENERIC_OPTIONS = (
4     bgcolor      => 'turquoise',
5     fgcolor      => 'black',
6     fontcolor    => 'black',
7     font2color   => 'turquoise',
8     height       => 8,
9     font         => 'gdSmallFont',
10    glyph        => 'generic',
11 );
12 sub new {
13     my $class = shift;
```

```

14 my %args = @_;
15 my $options = $args{-options}; # 选项，用的是散列的引用
16 my $panel = $args{-panel};
17 return bless {
18     options => $options,
19     panel => $panel,
20 }, $class;
21 }

22 sub option {
23     my $self = shift;
24     my $option_name = shift;
25     $option_name = lc $option_name; # 所有选项名都小写
26     if (exists $self->{options}{$option_name}) {
27         return $self->{options}{$option_name};
28     } else {
29         return $GENERIC_OPTIONS{$option_name};
30     }
31 }

32 sub make_glyph {
33     my $self = shift;
34     my @result;

35     my $glyph_type = $self->option('glyph');
36     my $glyph_class = 'Bio::Graphics::Glyph::' . $glyph_type;
37     eval("require $glyph_class") unless $glyph_class->can('new');

38     for my $feature (@_) {
39         my $glyph = $glyph_class->new(-feature => $f,
40                                         -factory => $self);
41         push @result, $glyph;
42     }
43     return @result;
44 }

45 1;

```

我从声明包名和打开严格类型检查开始（第1、2行）。

接着定义一个特定包散列。该散列包含一些在选项缺失时使用的缺省通用图符选项。这些选项包括缺省背景颜色，缺省高度及缺省字体（3到11行）。

构造函数 new() 从 Perl 的子程序参数表 @_ 中读出参数，放进名为 %args 的散列中。它接着寻找两项命名参数，-options 及 -panel。它把这两项参数存到内部匿名散列中，对应的键为 options 和 panel。最后它用 Perl 的 bless 函数创建对象，并将结果返回（12 到 21 行）。

第22到31行是对 option()方法进行定义。我从子程序的参数表中读出工厂对象和要

求的选项名。接着调用内置的lc()函数把选项名变为小写，以免程序员忘记选项名是-height 还是 -Height，以致影响该方法的运行。我从在new()中创建的选项散列中查找名字相似的键。如果找到，返回对应的值。不然，用选项名作为键值，在%GENERIC_OPTIONS 散列里查找并返回对应的值。如果选项散列和%GENERIC_OPTIONS 中都没有对应的键，返回不确定的值。

第32到34行的make_glyph()方法展示了Perl如何在运行时动态加载模块。我先用option()方法查找glyph选项的值，由此找到想要的图符类型。注意键值对glyph=>'generic' 在%GENERIC_OPTIONS 中定义。这意味着如果程序员不去索取具体的图符类型，option()将返回generic。

如果需要，我现在就加载要求的图符类。按惯例，所有Bio::Graphics::Glyph的子类都命名为Bio::Graphics::Glyph::subclass_name。通用图符(generic glyph)对应的Perl类是Bio::Graphics::Glyph::generic，箭头图符(对应类命名为arrow)定义在Bio::Graphics::Glyph::arrow内等。用字符串拼接操作(.)来创建类的全名，接着用require \$glyph_class编译该类，并把它载入内存。对require的调用包裹在一条字符串里，然后用eval()传给Perl的编译器。这样做是为了防止Perl在Factory 定义被创建时去调用require()。为了避免不必要的再次编译，我只在检测到类的new()构造函数还不存在时装载该类。

循环访问传入子程序参数列表(@_)中的每一特性，同时调用被选出的图符类的构造函数new()。每一新创建的图符被放进一个数组，然后把数组返回给调用者。

模块的最后一行是1。由于历史原因，它的作用是结束所有Perl模块。

注意图符构造函数的设计被扩展了，以便每个图符用两项命名参数构造。通过传递工厂拷贝，每个图符可以得到相关的选项。下面是Bio::Graphics::Glyph中相关的两个方法的节选：

factory()

当图符构造完成，返回传给图符的工厂对象：

```
sub factory {
    my $self = shift;
    return $self->{factory};
}
```

option()

这个传递方法用来得到给定名字的选项值：

```
sub option {
```

```

    my $self = shift;
    my ($option_name) = @_;
    return $self->factory->option($option_name);
}

```

图符调用 `factory()` 来得到它的工厂，并立即调用工厂的 `option()` 方法来获得子程序的参数列表中给定选项的值。

由 [孙海峰](#) 译于 2012-07-10

代码例子

示例 12-4 把所有东西联系起来，简单展示了 `Bio::Graphics` 的实际用法。它的输出在图 12-2 中显示。

示例 12-4: y 用 `Bio::Graphics` 的脚本

```

1 #!/usr/bin/perl
2
3 use strict;
4 use Bio::Graphics;
5 use Bio::SeqFeature::Generic;
6
7 my $bsg = 'Bio::SeqFeature::Generic';
8
9 my $span      = $bsg->new(-start=>1,-end=>1000);
10 my $test1_feat = $bsg->new(-start=>300,-end=>700,
11                            -display_name=>' 测试用特性',
12                            -source_tag=>' 测试而已 ');
13
14 my $test2_feat = $bsg->new(-start=>650,-end=>800,
15                            -display_name=>' 测试特性 2 ');
16
17 my $panel = Bio::Graphics::Panel->new(-width=>600,-length=>$span->length,
18                                         -pad_left=>12,-pad_right=>12);
19
20 $panel->add_track($span,-glyph=>'arrow',-double=>1,-tick=>2);
21
22 $panel->add_track([$test1_feat,$test2_feat],
23                     -glyph => 'box',
24                     -bgcolor => 'orange',
25                     -font2color => 'red',
26                     -height =>20,
27                     -label => 1,
28                     -description => 1,
29 );
30
31 print $panel->png;

```

输出的图符如图 12-2 所示。图符是通过 `Panel->png()` 方法生成的。如果想要将图符存入文件，可以使用 `Panel->write()` 方法。

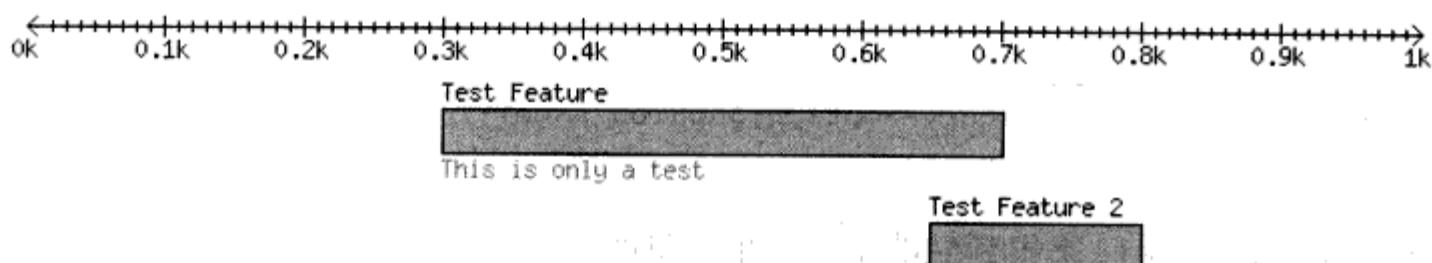


图 12-2：示例 12-4 的输出

加载 Bio::Graphics 库和 BioPerl 中标准 Bio::SeqFeatureI 类之一；Bio::SeqFeature::Generic（第 3 到 4 行）。为了避免重复输出特性的完整类名，我们把它存储在一个变量中（第 5 行）。

接着创建三个 Bio::SeqFeature::Generic 类。一项特性从位置 1 开始，到位置 1000 结束，而且会被用于绘制包含图像尺度的轨道（第 6 行）。其他的是第二条轨道中的特性（第 7 到 11 行）。

我们创建面板，并把选项传给它。这些选项规定了以像素为单位的宽度，以碱基对为单位的长度，和用于填充图像左右两边的额外空白（第 12 到 13 行）。

接下来，我们创建显示图像尺度的轨道（第 14 行）。它包含放在变量 \$span 中的单个特性，以及选取 arrow（箭头）图符的选项。它绘制双向箭头（-double=>1），并打印出箭头上的主要和次要的刻度线（-tick=>2）。

现在可以创建显示两条特性（\$test1_feat 和 \$test2_feat）的轨道了。我们加入第二条轨道。这次在选项中规定用背景为橙色的盒状（box）图符，描述字体为红色，高度是 20 像素。我们也有选择地打印出特性名和描述程序（第 15 到 22 行）。

动态选项

Bio::Graphics::Glyph::Factory 最初是根据只有简单静态选项值的思路设计的。但是，当我着手开发 Bio::Graphics 第一版时，我意识到赋予程序员动态计算选项的能力也能提供方便。

例如，调控蛋白质在 DNA 上的附着位置是分散在基因组中的。当调控蛋白质附着到 DNA 的具体部位时（名为“结合”的过程），附近的一条基因通常会被打开或关闭。有些结合部位强健，其他一些则虚弱，而 DNA/ 蛋白质结合过程的强度常常对理解调控过程如何进行非常重要。

为了创建展示 DNA/ 蛋白质结合部位特性的位置和相对强度的轨道，程序员或许想显示一系列矩形。每个矩形的开实和结束展示特性的跨度，而它的背景（内部）颜色展示结

合的强度：白色表示强度弱，粉红表示中等，而红色表示强。在最初的设计中，程序员需要像下面一样为轨道里的所有特性规定背景颜色：

```
@features = get_features_somewhow();
$panel->add_track(\@features,
    -glyph => 'generic',
    -bgcolor => 'pink');
```

但这样无法根据特性的不同来设置颜色。

在我认识到这项局限后，我重新设计并扩展了 API，以便允许选项值成为 CODE 引用。这些是 Perl 程序员用各种方式定义的匿名子程序。它们的用法和 C 里的函数指针的用法大致相当。这里是改动过的 `add_track()` 调用，利用了这门工具：

```
$panel->add_track(\@features,
    -glyph => 'box',
    -bgcolor => sub {
        my $feature = shift;
        my $score = $feature->score;
        return 'white' if $score < 0.25;
        return 'pink' if $score < 0.75;
        return 'red';
    });
);
```

这段代码原理如下：选项 `-bgcolor` 的值是匿名 CODE 应用，通过使用不带子程序名的关键词 `sub` 创建。每当图符要读取 `bgcolor` 选项的值时，该段代码被动态调用。这段子程序从它的参数数组拿到对应的特性，并调用它的 `score()` 方法来得到结合部位的强度。假设结合部位的强度由 0 到 1.0 间的浮点数表示，当 `score()` 返回的得分小于 0.25 时，返回选项值 `white`（白色）；当得分介于 0.25 和 0.75 之间时，返回 `pink`（粉红）；如果大于 0.75，就返回 `red`（红色）。结果如图 12-3 所示。

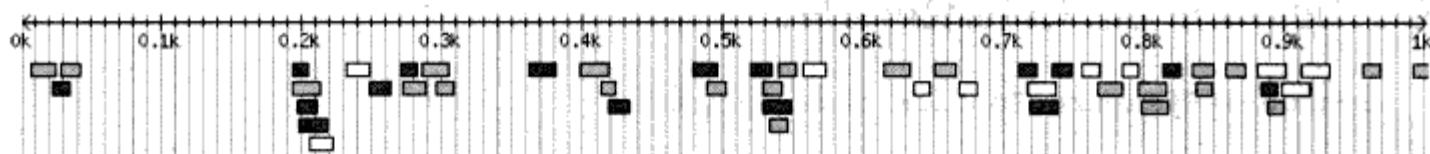


图 12-3：根据动态改变的值为背景着色

最后，我让传给 `add_track()` 的每项选项都能使用回调代码，包括 `-glyph` 选项自身。这赋予了最终用户定制和扩展类库叹为观止的灵活手段。

```
-bump => sub {
    my ($feature,$option_name,$glyph) = @_;
    # 得到所有参数
    return $glyph->panel->length < 50_000;
}
```

现在让我们看一下修订后经过简化的处理选项的代码。首先，我修改了 Bio::Graphics::Glyph::Factory。修改结果如下：

```
# 在 Bio::Graphics::Glyph::Factory 中
sub option {
    my $self = shift;
    my ($glyph, $option_name) = @_;
    $option_name = lc $option_name; # 选项均为小写
    my $value;
    if (exists $self->{options}{$option_name}) {
        $value = $self->{options}{$option_name};
    } else {
        $value = $GENERIC_OPTIONS{$option_name};
    }
    return $value unless ref $value eq 'CODE';

    my $feature = $glyph->feature;
    my $eval = eval {$value->($feature, $option_name, $glyph)};
    warn "Error while evaluating '$option_name' option for glyph $glyph,
feature $feature: ", $@, "\n" if $@;

    return defined $eval && $eval eq '*default*' ? $GENERIC_OPTIONS
{$option_name}
        : $eval;
}
```

现在该方法接受两个参数而不是一个。第一个参数是当前图符，而第二个参数与以前一样是选项名。工厂再次先在具体的轨道选项里，再在缺省散列 (%GENERIC_OPTIONS) 里查询该选项是否在轨道配置里命名。

不过，或缺选项值后新加了逻辑。通过调用 Perl 的 `ref()` 函数查询 `$value` 内容的数据类型。如果是代码引用，`ref()` 返回字符串 `CODE`。如果没有得到 `CODE`，与以前一样返回获取的值。否则调用图符的 `feature()` 方法来得到对应的特性，然后通过 Perl 的匿名代码引用的调用句法来调用代码引用：

```
$value->($feature, $option_name, $glyph)
```

传给回调的第一项参数是特性，第二项参数是选项名，而第三项是图符自身。

因为回调也许会导致运行时错误，通过把整个调用包裹在 `eval {}` 块里来防御这种可能性。万一致命错误在回调里发生，就返回没有定义的值，并把 Perl 的错误诊断信息放进特殊标量 `$@` 中。调用回调后，检查 `$@` 是否为空。如果不是空的，就打印不会终止程序的警告。

最后一步是返回回调得出的值。我觉得让回调指明它是否想用命名选项的缺省值是一件有

用的事。代码最后一行就是检查回调是否返回字符串 *default*。如果是的话，该方法返回缺省散列里的值。

为了配合工厂 option() 方法里的这项改动，得对 Bio::Graphics::Glyph-> options() 做出相应的改动：

```
# In Bio::Graphics::Glyph
sub option {
    my $self = shift;
    my ($option_name) = @_;
    return $self->factory->option($self,$option_name);
}
```

使用回调的过程中，我发现这个概念越来越有用。例如，我意识到回调能够非常优雅地处理语义缩放。基因图符绘制蛋白质编码基因内部结构的详尽表示。这种绘制在高度放大的情况下没问题，但是在观看很大的区域时就不行了，因为这时区域的细节小得不可辨认。但是，我们可以在 -glyph 选项上应用回调。这样的话，每当基因小于显示区域的百分之五时，我们可以动态选取简单的矩形 box (盒装) 图符，而非 gene (基因) 图符：

```
$panel->add_track(
    -glyph => sub {
        my ($feature,$panel) = @_;
        return 'box' if $feature->length/$panel->length < 0.05;
        return 'gene';
    },
    -height => 12,
    -font2color => 'red',
    -label_transcripts => 1,
    -label => 1,
    -description => 1,
);
```

注意 -glyph 选项的回调参数同其他选项不同，因为它的值在图符被创建前就要用到。因此，该回调传入特性和面板对象，而不是特性、选项名和图符。

凑巧的是，回调成为 Bio::Graphics 最受欢迎的特性。随后我又在 API 的其他地方随意加入回调，包括在处理传给 Panel (面板) 构造函数的选项时，以及在决定如何自顶向下为特性排序的代码里。

在许多情况下，用户们都会使用一些我未曾想到的回调函数用法。这里有个不错的例子：我给用户提供了一个在 Panel (面板类) 上定义回调的方法，让用户可以在 Panel 绘制了自身的网格线但还没有绘制图符时定义回调。几年后，一位雄心勃勃的基因组生物学家想出了用这项功能创建比较不同物种基因组的图表，其中这些物种的染色体经历了

相对于其他物种染色体的结构性变化。网格线的回调绘制着色的多边形，将一条染色体的特性同另一条染色体里对应的特性连接起来（图 12-4）。

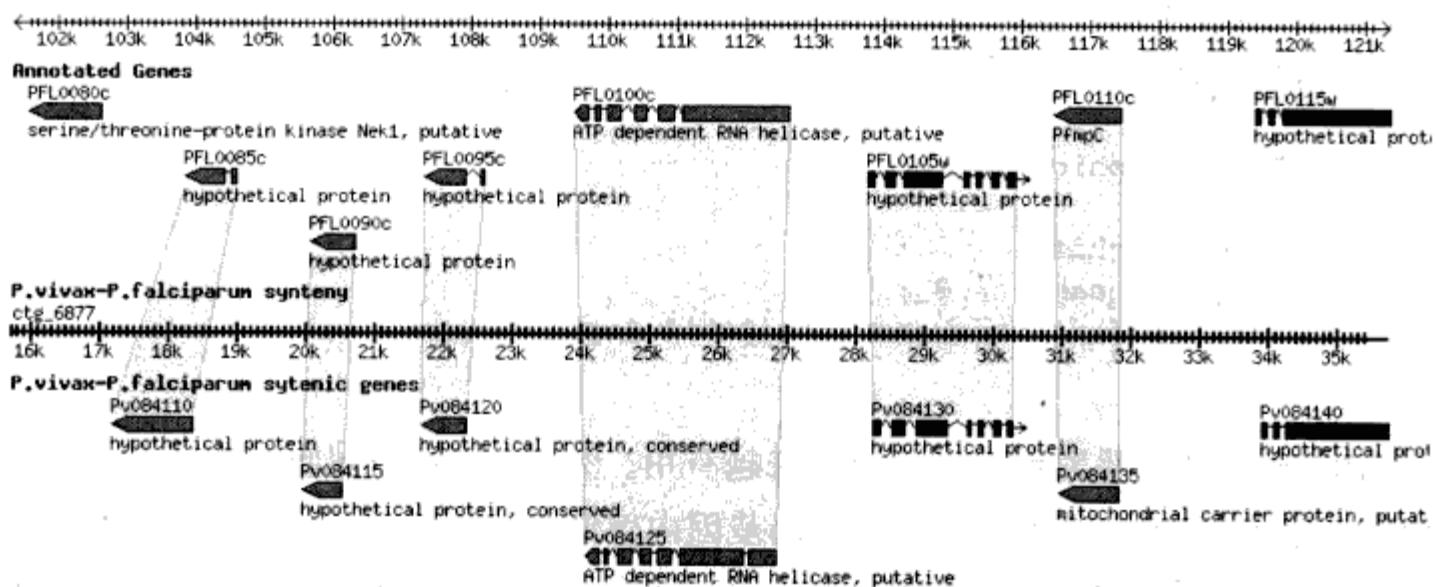


图 12-4: Bio::Graphics 的聪明用法，使得我们能够比较两条染色体间的关联特性

Bio::Graphics::Factory 故事也有黑暗的一面。当初充满激情时，我向获取和设置选项的方法里加入了一大堆功能。上面的代码示例里省略了这些功能。其中一项功能是用 Web 风格的层叠样式表初始化工厂。另一项功能向每个回调提供关于当前图符与同一轨道里其他图符或顶层图符间关系的详尽信息。实际使用中，这些功能从来没人用过，所以现在只是死亡代码而已。

扩展 Bio::Graphics

现在我们来看看在 Bio::Graphics 最初发布后我们加入的一些扩展。展示了代码号如何根据用户反馈进化的。

支持 Web 程序员

Bio::Graphics 的目标之一是用 Web 应用支持可以交互可以浏览的基因视图。基本思路是一份 CGI 脚本可以处理填好的表格，在表格的内容中指明了供浏览的基因组以及供显示的区域。该脚本连接数据库，处理用户请求，找到用户要求或用户感兴趣的区域，从对应区域里抽取特性，并将其传给 Bio::Graphics。Bio::Graphics 会生成图像，CGI 脚本会把图像嵌入标签，以便显示。

上述方法中不能为生成的图像创建图像映射。要想用户能够通过点击图符得到有关该图符的信息，就必须支持图像映射。有了图像映射，我们才能支持动态 HTML 功能。例如

用户把鼠标移到图符上时能看到相关的弹出提示，或者在用户右击图符时，填充下拉菜单。

最初版本的 Bio::Graphics 用单个方法 `boxes()` 来支持图像映射的生成。该方法返回一个数组。数组元素是包含图符边界的矩形、与每个图符关联的特性以及图符对象自身。要生成图像映射，程序员必须遍历这个数组，手工生成图像映射的 HTML 代码。

遗憾的是，根据用户发出的支持请求数目来看，并不像所希望的那么容易。因此，我在体验过编写自己的基于 Bio::Graphics 的基因组浏览器后，我在 Bio::Graphics Panel 里加入了 `image_and_map()` 方法。下面的代码片段展示了该方法的用法：

```
$panel = Bio::Graphics::Panel->new(@args);
$panel->add_track(@args);
$panel->add_track(@args);
...
($url,$map,$mapname) = $panel->image_and_map(
    -root => '/var/www/html',
    -url   => '/images',
    -link  => sub {
        my $feature = shift;
        my $name = $feature->display_name;
        return "http://www.google.com/
search?q=$name";
    }
)
print "<H2>我的基因组</H2>";
print "<IMG SRC='$url' USEMAP='#$mapname' BORDER='0' />";
print $map;
```

同以前一样，我们设置好一个 Panel (面板) 对象，并加入轨道，然后调用 `image_and_map()`，并传入三对参数和参数值。传入的 `-root` 参数给出了 Web 服务器文档根目录的物理地址（即 HTML 文件开始的地方）。传入的 `-url` 指明了相对与文档根目录，Bio::Graphics 生成的图像应该存在哪里。传入的 `-link` 参数是回调。Bio::Graphics 通过执行该回调将可以点击的连接附着到图符上。这种情况下，我们从回调的参数列表里重新拿到特性对象，通过调用 `display_name()` 得到供人阅读的特性名称，并生成 Google 搜索连接。其他几项 `image_and_map()` 选项可用于定制和扩展生成的图像映射。

该方法生成图像，并将其存到文件系统上由 `-root` 和 `-url` 指定的地方（这里是 `/var/www/html/images`）。接着，该方法返回一个三元列表，该列表包含指向生成图像的 URL，图像映射的 HTML 和用于 `` 标签的图像映射名。我们只需要打印出 HTML 的正确片段，就能使用图像和它的映射。

至此，我们有两套用 Bio::Graphics 开发的基于 Web 的基因组浏览器。一套是我写得，叫 GBrowse (<http://www.gmod.org/gbrowse>)，正被广泛用于显示大量基因组，从细菌到人

类。不过，它是 2002 年写的，那时还没发明基于 Ajax 的异步页面刷新（译注 2）。用户通过点击箭头按钮在基因组上移动，并且得等待屏幕重新加载。一个新的浏览器，Ajax Genetic Genome Browser (<http://genome.biowiki.org/gbrowser>)，正处于原型开发阶段，它支持以 Google Maps 的风格浏览基因组。用户只需抓住并滑动基因组就能进行浏览。

支持出版质量的图像

最初还有一条要求：支持多种图形格式。要满足这条要求，我让 Bio::Graphics 用 Perl 的 GD 库来执行底层图形调用。GD 库基于 Tom Boutell 的 libgd (<http://www.libgd.org>)，可以生成多种格式的位图图像，包括 PNG、JPEG 和 GIF。

Panel（面板）对象在内部创建并维护一个 GD 图形对象，并把该对象传给每条轨道的 draw() 方法。这些轨道依次把 GD 对象传给它们的图符，而图符再传给自己的子图符。

Bio::Graphics::Panel->png() 方法仅仅调用 GD 的 png() 方法：

```
# in Bio::Graphics::Panel
sub png {
    my $self = shift;
    my $gd = $self->gd;
    return $gd->png;
}
```

jpeg() 和 gif() 方法类似。程序员可以选择获取原生的 GD 对象，并调用 png() 方法：

```
$gd = $panel->gd;
print $gd->png;
```

让公共接口能够使用内部 GD 对象的好处是程序员可以用 GD 对象完成额外的事情，比如在大图里内嵌它，或者操作它的颜色。

我选用 GD 的后果之一是 Bio::Graphics 最初只能生成位图图像。这一问题在 Todd Harris 写出 Perl 的 GD::SVG 模块 (<http://toddot.net/projects/GD-SVG>) 后迎刃而解。GD::SVG 的 API 同 GD 相同，但支持可缩放矢量图形标准（SVG）的图像。SVG 图像可以在高分辨率下进行无损打印，也可供各式画图工具处理，比如 Adobe Illustrator。

译注 2：这里作者搞错了。AJAX 技术早于 2002 年问世。1996 年 IE3 上就能利用 IFRAME 实现页面异步刷新。基于 XMLHttpRequest 组件的 AJAX 技术最早在 2000 年的 Outlook Web Access 里应用。也就是从那时起人们知道了 XMLHttpRequest。“AJAX”这一技术名词是 2005 年 Jesse James Garrett 提出的。

在Bio::Graphics中加入GC::SVG的支持后，把-image_class参数传给Panel的构造函数就能生成SVG图像：

```
$panel = Bio::Graphics::Panel->new(-length=>1000,
                                      -width=>600,
                                      -image_class => 'GD::SVG'
                                     );
$panel->add_track.... etc...
print $panel->gd->svg;
```

最终，Bio::Graphics代码内部的惟一改动是处理image_class选项，并加载指明的图像库。这使得我们支持前向兼容，只要图形库同GD库兼容。例如，如果有人写出生生成PDF格式的图形文件的GD::PDF模块，Bio::Graphics就能配合它生成PDF。

添加新图符

Bio::Graphics刚发布时，支持大约一打简单的图符，包括矩形、椭圆、箭头、基因图符，以及绘制蛋白质和DNA序列的图符。每项图符包含多个配置选项。这样导致了大量的可能显示。不过，该数目仍然是有限的，而基因组上特性的数目可能是无限的。幸好，添加新的图符相对容易。随着时间的流逝，我和其他BioPerl程序员不断向Bio::Graphics添加许多新图符。目前有近70种图符类型，从whimsical（大卫王星，一种六角图案）到sophisticated（一种三元图，用于比较在多个种群里序列突变的频率）。

可以通过扩展已有图符来轻易创建新图符是很有价值的功能。示例12-5中展示怎样创建名为hourglass（沙漏）的新图符（Glyph）。

示例 12-5：沙漏图符

```
1 package Bio::Graphics::Glyph::hourglass;
2 use strict;
3 use base 'Bio::Graphics::Glyph::box';
4 sub draw_component {
5   my $self = shift;
6   my ($gd,$dx,$dy) = @_;
7   my ($left,$top,$right,$bottom) = $self->bounds($dx,$dy);
8   # 绘制多边形代表沙漏
9   my $poly = GD::Polygon->new;
10  $poly->addPt($left,$top);
11  $poly->addPt($right,$bottom);
12  $poly->addPt($right,$top);
13  $poly->addPt($left,$bottom);
14  $poly->addPt($left,$top);
15  $gd->filledPolygon($poly,$self->bgcolor);
16  $gd->polygon($poly,$self->fgcolor);
```

```
17 }
```

```
18 1;
```

该图符生成一个沙漏（图 12-5）。它从定义包名开始。根据常规，包名必须以 Bio::Graphics::Glyph:: 开始（第一行）。它接着申明它继承于 Bio::Graphics::Glyph::box。这是绘制矩形的简单图符（代码第 3 行）。

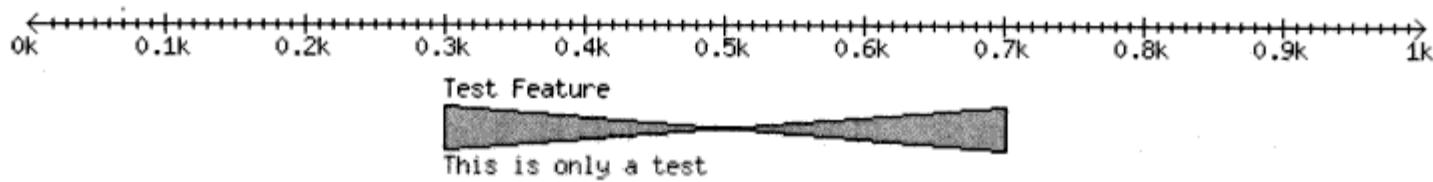


图 12-5：沙漏图符，扭曲的标准盒装图符

该图符接着重写了继承的 draw_component() 方法（第 4-17）行。设置好绘制环境后，该方法被 Bio::Graphics::Glyph 的 draw() 方法调用。这个方法接受 GD 对象，以及标明该图符相对于包含它的图符的位置的水平和垂直坐标。我们把相对坐标传给继承的 bounds() 方法，以便将他们转换为包含该图符的矩形的绝对坐标。

现在我们正式开始绘制图符。我们用 GD 的多边形类库创建一多边形，并添加对应沙漏左上、右下、右上、左下角的节点（第 9 到 14 行）。然后，我们先把多边形传给 GD 对象的 filledPolygon() 方法，以便绘制多边形的实心内容（第 15 行），再传给 GD 对象的 polygon() 和 fgcolor() 方法以绘制沙漏的外廓（第 16 行）。注意我们使用了继承下来的方法 bgcolor() 和 fgcolor() 等方法来获得合适的颜色以用于填充和绘制外廓。

这说明了在 Bio::Graphics 中添加新图符是项简单的工作。在许多情况下，程序员可以通过继承目前已有的且能够满足我们大部分需求的图符来创建新的图符，然后在继承下来的新图符中再修改一两个方法以定制我们的功能。

结论和教训

设计让其他程序员使用的软件是项富有挑战性的工作。设计的软件必须直观简单，容易使用，因为程序员和其他人一样，没有耐心。但是设计的软件也不能过于简化，以至于缺失功能。理想的代码库必须能让初级程序员立刻上手，而让更有经验的程序员轻松定制，还能让专家随意扩展。

我认为 Bio::Graphics 正好可以满足用户需要。BioPerl 新手可以立刻上手，用人们惯用的 BioPerl 对象（比如 Bio::SeqFeature::Generic）写出简单脚本。中等水平的程序员可以通过写回调修改类库的输出，而最有经验的程序员可以用定制的图符扩展类库。

`Bio::Graphics` 也展示了标准接口的威力。因为它用来显示任何遵循 BioPerl 的 `Bio::SeqFeatureI` 接口的对象，它也可以方便地同任何 BioPerl 的序列数据读取模块协同工作。`Bio::Graphics` 能够为手工规定的序列特性生成图表，也能同样轻松地为从其他来源得到的特性生成图表。这些来源包括普通文件，数据库查询以及由 Web 服务生成并通过网络传输的数据。

该模块也有一些缺陷，如果我现在重新实现它，有些地方我会用不同的方式实现。子图符生成的方式是主要问题。当前的实现里，如果你把一个图符赋予一个特性，该特性又有子特性，那子图符的类型与顶层图符相同。

这样做有两大缺点。首先，我们必须用子类创建复合图符。复合图符里子图符复用预先定义的类的代码，而父图符是新东西。其次，图符的方法必须知道它们在显示哪一层的特性。例如，要创建一个顶层特性是点线八角形而子特性是矩形的图符，`draw_component()` 方法必须调用图符的 `level()` 方法，以便找到当前嵌套层数，并绘制相应的形状。如果我能重新来过，我会提供一项 API，用于在每层嵌套选取正确的图符。

盒状模型也是一项令人讨厌的东西。图符可以在它们自己周围分配多余空间，以便绘制诸如箭头、高亮或标签一类的装饰。它们通过重写 `pad_left()`、`pad_right()` 等方法实现这一功能。

这样做一般没有问题。但当我们通过继承已有图符来定义新的图符类时就不行了。你得为了新加的装饰调整填充值。子类必须仔细的找出它的父类申请了多大的填充空间（通过调用继承的 `pad()` 方法），然后再加入自己的填充值。这样做也许会变得困难。如果我可以重来，我会跟踪记录图符在哪里用 `draw_component()` 方法进行绘制，并在需要时增大环绕限制它的矩形。

不幸的是，实现上述任何一项改动都会从根本上改变图符的 API。这样就需要有人，多半是我自己，改写 60+ 全部已有的图符类，以免破坏它们。就现在而言，我接受某块已经够好了，但永远不可能完美的现实。这是最后一条，也许是最重要的教训。

基因排序器的设计

Jim Kent

本章的内容是关于我以前写的一个叫做基因排序器 (Gene Sorter) 的一个大小适中的程序。比起本书其他大部分章节中所描述的工程，基因排序器 (Gene Sorter) 代码的规模都要大一些，总共大概有两万行。尽管也存在一些很好的小一些的基因排序器，对我来说真正的美丽在于它在阅读、理解和作为整体进行扩充时的简易程度。本章我将全面描述基因排序器做什么，着重强调代码中那些更重要的部分，然后讨论如何使编写千行以上的程序成为一件快事，甚至让人觉得美丽。

基因排序器能帮助科学家从人类基因组的大约 25 000 种基因中进行快速的筛选，找出那些跟他们研究的课题关系最密切的信息。这个程序是 <http://genome.ucsc.edu> 网站的一部分，那个网站也包含其他一些工具，这些工具都是用于处理人类基因组项目 (Human Genome Project) 中产生的数据的。基因排序器的设计简单而又灵活。它融入了许多我们从先前的两代网络生物医学数据程序中所取得的经验。程序通过 CGI 收集来自用户的输入，然后查询 MySQL 数据库，最后用 HTML 呈现结果。程序代码中大约有一半存在于库中，与 <http://genome.ucsc.edu> 网站的其他工具共享。

人类的基因组是一组数字密码，它通过特殊的方式包含了构建人体所需的所有信息，包括构建人体最重要的器官：大脑。这些信息存储在 30 亿个 DNA 碱基中。每一个碱基可

以是一个A、C、G或者T。这样，每一个碱基就有两比特的信息，或者说，人类的基因组中共有750MB的信息。

不可思议的是，用你口袋里的一根记忆棒就能存下构造出一个人的所有信息。更不可思议的是，通过许多基因组的进化分析，我们知道那些信息中只有大约10%是真正需要的。其他的90%大都是走入死胡同的进化实验的遗迹，以及一些称为转位子(transposon)的类病毒的元素所留下的乱七八糟的东西。

基因组中现在仍起作用的部分大都可以在基因中找到。基因包含调节元素和基因产物自身的编码，调节元素决定基因产物将被产生出多少。基因的调节常常非常复杂。不同类型的细胞使用不同的基因，同类细胞在不同的环境中也会使用不同的基因。

基因产物也是多种多样的。基因中非常重要的一类会产生信使RNA(mRNA)，信使RNA又会被转译成蛋白质。这些蛋白质包括能使细胞感知周围环境并跟其他的细胞进行交流的受体分子，帮助把食物转化成更有用的能量形式的酶，以及控制其他基因活动的转录因子。尽管那并不是一项轻松的工作，科学家还是鉴别出了基因组中90%的基因，总共两万多种。

大多数科学实验项目都只对这些基因中的数十种感兴趣。研究罕见遗传疾病的人们通过分析疾病遗传的模式把疾病跟单条染色体的大约1000万个碱基区域联系起来。近年来科学家们已经确定了更多的常见疾病跟10万个碱基区域的关联，这些疾病包括部分但并非全部源于遗传的糖尿病。

基因排序器的用户界面

基因排序器能把DNA上所有与疾病相关的区域的已知基因收集到一个候选基因列表中。这个列表显示在表格中，如图13-1所示，它包括每种基因的概要信息以及指向更多信息的超链接。候选列表可进行过滤以排除那些明显不相关的基因，比如当研究人员在研究一种肺部的遗传疾病时，就不必考虑那些仅在肾中可以表达的基因。基因排序器在人们想要一次观察多种基因的其他场合也是有用的，比如当人们研究以相似方式进行表达的基因或者有相似的已知功能的基因时。如今，基因排序器已经用于人类、小鼠、大鼠、果蝇和线虫的基因组。

屏幕上方的控制用于指定使用何种基因组的哪一个版本。下方的表格每行对应一种基因。

一个单独的配置页面控制表格中哪些栏将被显示以及怎样显示。一个过滤器页面可基于列值的任意组合进行基因过滤。

UCSC Human Gene Sorter

#	Name	VisiGene	Expression Data										Genome Position	Description			
			fetal brain	whole brain	amygdala	thymus	bone marrow	PB-CD4+ Tcells	skin	adipocyte	pancreatic islets	heart			lung	kidney	liver
1	SYP	181524														chrX 48,937,407	synaptophysin
2	LMO6	221														chrX 48,924,285	LIM domain only 6
3	PLP2	181508														chrX 48,916,798	proteolipid protein 2 (colonic)
4	CACNA1F	174590														chrX 48,962,622	calcium channel, voltage-dependent, alpha 1F
5	FLJ21687	179636														chrX 48,909,468	PDZ domain containing, X chromosome
6	CCDC22	185850														chrX 48,986,613	coiled-coil domain containing 22
7	FOXP3	1768														chrX 49,001,293	forkhead box P3
8	GPKOW	n/a														chrX 48,862,156	G patch domain and KOW motifs
9	PPP1R3F	179945	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	chrX 49,022,141	protein phosphatase 1, regulatory (inhibitor)
10	WDR45	35149														chrX 48,832,019	WD repeat domain 45 isoform 1
11	PRAF2	175866														chrX 48,817,185	JM4 protein
12	JM11	185851														chrX 48,808,956	hypothetical protein LOC90060
13	TFE3	986														chrX 48,779,619	Hypothetical protein DKFZp761J1810.
14	GRIPAP1	30802														chrX 48,729,348	GRIP1 associated protein 1 isoform 1
15	KCND1	178025														chrX 48,708,389	potassium voltage-gated channel, Shal-related
16	OTUD5	62197														chrX 48,682,134	hypothetical protein LOC55593

图 13-1：基因排序器的主页面

表格可基于多种方式排序。在这个例子中，它是依据与被选基因 SYP 的近似程度进行排序的。SYP 是一种与神经递质（Neurotransmitters）的分泌有关的基因。

通过 Web 跟用户保持对话

基因排序器是一个 CGI 脚本。当用户用 Web 浏览器打开基因排序器的 URL (<http://genome.ucsc.edu/cgi-bin/hgNear>) 时，Web 服务器执行脚本并通过网络回送结果。脚本的输出是一个 HTML 表单。当用户点击表单 (form) 中的按钮后，Web 浏览器就发送一个 URL 给 Web 服务器，其中包含了下拉菜单的当前选择和其他控件的当前值，这些值被编码成一对一对的 “*variable=value*” 的形式。接着 Web 服务器再次执行脚本，把变量 / 值对作为输入传入。于是脚本就产生一个新的 HTML 表单作为响应。

CGI 脚本可用任何语言编写。基因排序器脚本实际上是一个用 C 语言编写的较大规模的程序。

与桌面系统中其他一些用户交互式的程序相比，CGI 脚本既有优点也有缺点。CGI 脚本的可移植性较高，不需要编写不同的版本分别支持 Windows、Macintosh 和 Linux 桌面机上的用户。另一方面，它们的交互性却很一般。除非采用 JavaScript（从而直接引入严重的可移植性问题），否则只有在用户点击一个按钮，然后花上一两秒钟等到一个新

的 Web 页面之后，显示才会更新。然而，对于大多数基因组应用程序的需求而言，CGI 提供了可接受的交互性，以及一个非常标准的用户界面。

CGI 脚本的生命周期非常短暂。它开始于响应用户的点击而终止于一个 Web 页面被产生出来。因此，脚本不能通过程序变量长时间地保存信息。对于非常简单的 CGI 脚本，所需的全部信息都存储在变量 / 值对中（“变量 / 值对”也被称为“CGI 变量”）。

但是，对于更复杂的脚本，比如基因排序器，这就不能满足需要了，因为脚本可能需要在用户浏览了好多个页面之后记住先前某个页面上的某个控件的值，而 Web 服务器只传送那些与最新页面上的控件相对应的 CGI 变量。因此，我们的 CGI 脚本需要一种能长期保存数据的方法。

CGI 提供了两种机制用于保存表单控件中的不可见数据。第一种是隐藏的 CGI 变量，数据通过类型为“隐藏”（hidden）的 `<INPUT>` 标签保存在 HTML 中；别一种是 cookie，数据被浏览器保存并通过 HTTP 头传送。

Cookie 技术在最初发布时就引发了很多争议，而且一些用户还会在浏览器中禁止它。但是，cookie 可以长期保存数据，长到以年为单位，而隐藏变量在 Web 页被关闭时就马上消失了。Cookie 和隐藏变量都不能保存真正大量的数据。确切的数据量随 Web 浏览器的不同而不同，大体上讲，尝试通过这些机制保存 4KB 以上的数据都是不安全的。

为了综合利用 cookie 和隐藏变量的能力，基因排序器的开发团队开发出一种“行李车”（cart）对象，这种对象通过 SQL 数据库将 cookie 和隐藏变量结合起来。行李车维护两张数据库表，一张关联用户，另一张关联 Web 会话（session）。两张表的格式是完全相同的，都有一个主键列；一个格式相同的 blob 字段，包含通过 URL 传送的所有变量 / 值对；以及一个跟踪使用时间和访问计数的字段。

持久化的 cookie 中保存一个指向用户表数据的主键，隐藏的 CGI 变量中保存指向会话表数据的主键。在启动时，脚本从 cookie 中寻找用户主键。如果找到了，它就将与之关联的变量 / 值对加载到一个哈希表中。如果不能找到 cookie，它就产生一个新的用户主键，而哈希表仍然为空。

接下来，脚本寻找会话主键并从中加载变量，哈希表中的原有同名变量将被覆盖。最后，所有新的 CGI 变量被加载并插入到哈希表中（从而可能覆盖所有先前插入的同名变量）。

脚本可以利用多种库程序读 / 写哈希表“行李车”中的变量。当脚本退出时，它会用行李车的当前内容更新数据库表。如果用户禁止了 cookie，它仍然可以在单次会话中跟基因排序器交互，因为会话主键并不保存在 cookie 中。将会话跟用户主键分开也使得用户

可以同时在两个独立的窗口中运行基因排序器而互不影响。用户级的行李车使基因排序器可以从它上次被保存的地方继续运行，即使用户在中间访问过另外一个网址也没关系。

在 <http://genome.ucsc.edu> 上的基因排序器实现中，网站上所有的 CGI 脚本共享同一个“行李车”。这样，行李车就包含了比一般的程序变量更“全局”的变量。这常常很有用。如果在我们的程序中，某个用户正在关注老鼠的基因组，而不是人类的，那么她很可能在另一个程序中也使用老鼠的基因组。

然而，随着我们程序规模的不断扩大，为了避免行李车变量之间不经意的名字冲突，我们约定 CGI 变量以使用它们的 CGI 脚本的名字开头（除非它们确实需要做成全局的）。这样一来，基因排序器的大多数行李车变量的名字都以 “hgNear_” 开头了（我们使用下划线做分隔符，而不是点号，因为点号跟 JavaScript 中的用法冲突）。

总而言之，通过使用行李车对象，在基因排序器中维持用户的感觉连惯性就变得相对简单，即使每一次用户点击后执行的都是一个单独的程序实例。

CGI 脚本短暂的生命周期也有它的优点。特别是，CGI 脚本不需要操心内存泄露和文件关闭，因为在程序退出时这些资源都被操作系统清理了。对于没有自动资源管理的 C 语言来说，这一点特别好。

多态的威力

一个程序，只要它支持某个方面的灵活性，它的实现中就很可能用到某种多态对象。在基因排序器的主要页面中，占据了大量空间的表格都是由一系列多态的列 (column) 对象组成的。

在 C 语言中构造多态对象不像在面向对象语言中那么简单。但这仍然是可以做到的，方法也相对直接：使用结构体代替对象，使用函数指针代替多态方法。例 13-1 给出了一个 C 语言编码的列 (column) 对象，有所简化。

示例 13-1：column 结构体，用 C 语言编写多态对象

```
struct column
/* 大表格中的一列，hgNear 中的核心数据结构 */
{
    /* 所有的列中都存在的数据： */
    struct column *next;    /* 链表中的下一列。 */
    char *name;             /* 列名，用户看不到。 */
    char *shortLabel;       /* 列标签。 */
    char *longLabel;        /* 列描述。 */

    /* -- 方法 -- */
}
```

```

void (*cellPrint)(struct column *col, struct genePos *gp,
                  struct sqlConnection *conn);
/* 在 HTML 中打印该列的一个单元格. */

void (*labelPrint)(struct column *col);
/* 在标签行中打印标签. */

void (*filterControls)(struct column *col,
                       struct sqlConnection *conn);
/* 打印高级过滤器中的控件标签. */

struct genePos *(*advFilter)(struct column *col,
                             struct sqlConnection *conn,
                             /* 返回高级过滤器的位置列表. */

/* 下面的几个字段是查找表 (Lookup table) 使用的. */
char *table;           /* 关联表 (associated table) 的名字. */
char *keyField;        /* 关键表中的 GeneId 字段. */
char *valField;        /* 关键表中的 Value 字段. */

/* 除了跟查找相关的字段, 关联表还使用如下字段. */
char *queryFull;       /* 返回两列键 / 值的查寻. */
char *queryOne;         /* 给定键, 返回相关值的查询. */
char *invQueryOne;      /* 给定值, 返回相关键的查询. */
);

```

这个结构体中的字段可分为 3 组：最前面的是所有类型的列都要用到的字段，接着是多态方法，最后的部分包含类型特定的数据。

每一个列对象都包含一些容纳“公共”数据的空间，这种“公共”数据是指各种类型的列中都存在的。完全可以通过使用联合体（union）或类似机制来避免这种空间的浪费。然而，这会使类型特定字段的使用更加复杂，而且由于总的列数连 100 个都不到，节省下来的空间加起来也只有几 K 字节而已。

程序的大部分功能都实现在列对象的方法中。每个列对象都知道如何为一个特定的基因获取数据，数据可能是字符串或 HTML。当列中的数据适合于一个简单的查询字符串时，列对象还能做基因查询。列对象也实现了用于数据过滤的交互式控件，以及过滤方法本身。

列对象是由一个工厂方法基于 *columnDb.ra* 文件中的信息创建的。例 13-2 是从这种文件摘录的一个片断。所有的 *columnDb* 记录都包含了一些字段用于描述列名，用户可见的短标签和长标签，列在表格中的默认位置（优先级），默认情况下列是否可见，以及一个类型字段。类型字段决定列拥有哪些方法。除此之外，也可能存在类型特定的附加字段。在很多情形中，*columnDb* 记录包含了查询相关数据库表的 SQL 语句，以及链接到列中每个数据项的 URL。

示例 13-2：一个 columnDb.ra 文件片断，其中包含列的元数据

```
name proteinName
shortLabel UniProt
longLabel UniProt (SwissProt/TrEMBL) Protein Display ID
priority 2.1
visibility off
type association kgXref
queryFull select kgID,spDisplayID from kgXref
queryOne select spDisplayId,spID from kgXref where kgID = '%s'
invQueryOne select kgID from kgXref where spDisplayId = '%s'
search fuzzy
itemUrl http://us.expasy.org/cgi-bin/niceprot.pl?%s

name proteinAcc
shortLabel UniProt Acc
longLabel UniProt (SwissProt/TrEMBL) Protein Accession
priority 2.15
visibility off
type lookup kgXref kgID spID
search exact
itemUrl http://us.expasy.org/cgi-bin/niceprot.pl?%s

name refSeq
shortLabel RefSeq
longLabel NCBI RefSeq Gene Accession
priority 2.2
visibility off
type lookup knownToRefSeq name value
search exact
itemUrl http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Search&db=Nucleotide&term=%s&doptcmdl=GenBank&tool=genome.ucsc.edu
```

columnDb.ra 文件的格式很简单：每行一个字段，不同的记录以空行分隔。每一行开头的是字段名，剩下的是字段值。

在<http://genome.ucsc.edu>网站上，大量的元数据都是用这种简单的、面向行的格式描述的。我们曾经考虑使用这种文件的索引版本替代关系型数据库（扩展名“ra”代表“关系型替品”（relational alternative））。但考虑到大量实用工具的存在，我们还是决定用关系型数据库保存我们的数据。由于.ra文件的读、写、解析都很容易，因此它们在这些应用程序中也被继续使用。

columnDb.ra 文件的内容被编排成三层目录结构。根目录是所有物种中都会出现的列的信息。中间层的目录包含特定物种的信息。按照我们对一个特定物种基因组进化的理解，我们将拥有它们DNA次序的不同集合（assembly）。而最底层的目录包含的就是特定集合的信息。

读*columnDb*文件的代码会创建一个哈希表的散列，外层哈希表的键是列名，内层哈希

表的键是字段名。底层的信息可以包含全新的记录、增加或覆写 (override) 记录中被上层定义过的特定字段。

某些类型的列跟关系数据库中的列直接对应。查找 (lookup) 类型的列引用一个表，这个表对 gene ID 字段进行了索引，表中针对每一个 gene ID 的记录不会多于一行。类型行包含表、gene ID 字段和列中显示的字段。例 13-2 中的列 proteinAcc 和 refSeq 是查找类型的例子。

如果关系型的数据库表可以为每种基因包含多条记录，它就变成关联 (association) 类型了。在基因排序器中，如果有单一的基因关联多个值，它会被显示成逗号分隔的列表。关联 (association) 类型包含取数据的 SQL 语句，这些语句可能每次取单个基因的数据 (queryOne)、取全部基因的数据 (queryFull) 或者取跟某个特定值相关联的基因的数据 (invQueryOne)。SQL 语句 queryOne 实际上返回两个值，一个用来在基因排序器中显示，另一个用于超链接，尽管它们可以是相同的。

基因排序器中的大多数列都是查找型或关联型。给定任意一个以 gene ID 做键的关系型数据库表，由它产生基因排序器中的一列非常容易。

其他的列，比如基因表达 (gene expression) 列，就相对复杂了。图 13-1 有了一个基因表达列，被显示成各种器官（比如大脑、肝脏、肾脏等）名字下方的彩色方块。方块的颜色表示跟生物体的整体信使 RNA 水平相比，有多少该种基因的信使 RNA 可以从这些器官中找到。红色代表超出平均的表达，绿色代表低于平均的表达，而黑色则表示平均水平的表达。

图 13-1 所示的是从胎脑到睾丸一整套基因表达信息的集合被作为一个基因排序列来考虑。从 HTML 表格上看，它被分成了三列，这是为了在五种器官分组之间加入灰线来使得页面更易读。

滤除无关的基因

过滤器是基因排序器的强大特性之一。过滤器可被应用到任何一列基因，从而显示那些只跟特定用途有关的基因。比方说，基因表达列上的过滤器可以用来查找只在人类的大脑中表达而在其他组织中表达的基因。基因组位置过滤器可以查找 X 染色体上的基因。把这两种过滤器组合起来，就可以用来查找特定于大脑的 X 染色体上的基因。研究孤癖症的学者对这类基因特别感兴趣，因为它似乎具有相当程度的伴性 (sex-linked) (译注 1) 特征。

译注 1：即跟性染色体有关。

每一列都有两个过滤方法：用来向过滤器用户界面输出HTML的filterControls，和完成实际的过滤的advFilter。这两个方法通过行李车变量进行交互，行李车变量的命名约定中包含了程序名，字母“as”以及列名做为特定变量名的前缀。通过这种方式，同一类型的不同列将拥有不同的行李车变量，而过滤器变量也可以与其他变量区分开来。一个非常有用的函数cartFindPrefix——返回以给定前缀开头的所有变量的列表——在过滤器系统中被大量使用。

过滤器被排列成一个链。开始，程序创建一个包含所有基因的列表。然后，它到行李车中查找是否有过滤器被设置。如果有，它把每一列的过滤器都调用一遍。第一个过滤器将全部基因的列表作为输入。后续的过滤器则始于前一个过滤器的输出。过滤器被应用的次序无关紧要。

过滤器是基因排序器中性能最为关键的代码。大多数代码仅处理50到100个基因，而过滤器则需从上万种的基因中进行筛选。为了保证迅速的交互响应时间，过滤器在每个基因上花的时间不能超过万分之一秒。现代的CPU处理速度极快，万分之一秒不算什么限制。然而，磁盘寻道时间仍然需要千分之五秒左右，因此过滤器必需避免引起磁盘寻道。

大多数过滤器开始时都到行李车中检查是否设置过相关的变量，如果没有，它们仅把输入列表原封不动地快速返回。然后，过滤器读取跟列相关的表格。每次都读取整张表以避免可能引发的针对各数据项的磁盘寻道，如果它只处理不多的几个基因，这样会慢一些，但处理大量基因时这样就快得多了。

通过过滤的基因被放入一个以gene ID作键的哈希表。最后，过滤器调用一个叫作weedUnlessInHash的函数，该函数循环检查输入中的每个基因，来查看它是否存在于哈希表中，如果是就把它拷贝到输出中去。最终结果就是一个用相对少量的代码实现的快速灵活的系统。

大规模美丽代码理论

从设计和编码层面上讲，在我所参与过的项目中，基因排序器算是较漂亮的程序之一。系统的主要部分，包括行李车,.ra文件（译注2）的目录结构以及染色体数据库接口，都融入了从以前的项目中获取的经验教训，而且都经历了两轮或三轮迭代。程序中对象的结构与用户界面和关系型数据库的主要组件都能清晰地对应起来。采用的算法简单又高效，在速度、内存占用和代码复杂度之间做了很好的折衷。与大多数同等规模的程序相

译注2：原文中是rile，应为file。

比，这个程序中存在的bug要少得多。就连项目组外的其他人都可以很快跟上代码库的进度并为它做贡献。

编码是人的一种活动，而在程序设计中最受限的资源可能就是我们人类大脑的记忆。通常，我们的短期记忆（short-term memory）可以同时记住五六件事情。超过这个数我们就需要使用长期记忆了（long-term memory）。我们的长期记忆系统有惊人的存储量，但信息进入其中的速度却慢一些，同时我们也不能随机地提取其中的信息，只能通过关联（association）来检索。

几百行的程序，其结构的设计可以基于算法或基于对机器的考量，而更大的程序，其结构设计的考量就必须纳入人的因素，至少，如果我们希望能够一段较长的时间里维护和扩展这个程序，就必须这样做。

理想的情况是，理解一段代码所需的所有上下文都能显示在一屏代码中。否则，阅读代码的人就只能一屏一屏地翻来翻去，希望自己能理解它。如果代码很复杂，阅读它的人在回到最初的一屏代码时，很可能已经忘记了刚刚看到的每一屏上都定义了些什么，这样，在理解代码任何一部分之前，他就不得不先记住大量的代码。显然，这将降低程序员的效率，甚至使他们中的大多数感到很沮丧。

选择合适的名字，对于代码局部的可理解性尤为重要。局部变量的名字，使用一两个字母也可以，但这种超短的名字，其数量最好不要超过短期记忆的容量。其他的名字都应该是单词、短语或者通用的（而且简短的）缩写。在大多数情况下，应该让阅读代码的人能从变量或函数的名字中推断出它的用途。

如今，随着功能强大的集成开发环境（integrated development environment）的使用，阅读代码的人只要点击一下鼠标，就能从一个符号的使用处跳到它的定义处。然而，我们还是希望把代码组织好，使用户在通常情况下不必跳到符号的定义处，除非他对细节非常好奇。我们不应该迫使他每读懂一行代码都要跟踪好几个超链接。

名字可能过长，也可能太短，不过大多数程序员受算法的数学描述以及罪恶的“匈牙利命名法”的影响，都把名字起得太短。想出一个好的名字可能需要一点时间，但它是值得的。

对于一个局部变量，恰当的名字就是最好的文档。例13-3给出了基因排序器中一个相当好的函数例子。这个函数根据可包含通配符的规则来过滤关联列。（同时还有一个更简单，也更快的方法处理精确匹配的情况）。这段代码一屏就能装下，对于函数来讲，这永远是个优点，尽管并不总是可能的。

示例 13-3：处理通配符的列（关联类型）中的过滤方法

```
static struct genePos *wildAssociationFilter(
    struct slName *wildList, boolean orLogic, struct column *col,
    struct sqlConnection *conn, struct genePos *list)
/* 过滤匹配通配符列表中任何一项的关联. */
{
    /* 将关联按 gene ID 分组 */
    struct assocGroup *ag = assocGroupNew(16);
    struct sqlResult *sr = sqlGetResult(conn, col->queryFull);
    char **row;
    while ((row = sqlNextRow(sr)) != NULL)
        assocGroupAdd(ag, row[0], row[1]);
    sqlFreeResult(&sr);

    /* 寻找匹配的关联并把它们放进 passHash 表中. */
    struct hash *passHash = newHash(16); /* 哈希表，用于保存通过过滤的项目 */
    struct genePos *gp;
    for (gp = list; gp != NULL; gp = gp->next)
    {
        char *key = (col->protKey ? gp->protein : gp->name);
        struct assocList *al = hashFindVal(ag->listHash, key);
        if (al != NULL)
        {
            if (wildMatchRefs(wildList, al->list, orLogic))
                hashAdd(passHash, gp->name, gp);
        }
    }
    /* 创建经过过滤的列表，善后清理，返回. */
    list = weedUnlessInHash(list, passHash);
    hashFree(&passHash);
    assocGroupFree(&ag);
    return list;
}
```

函数原型后面跟着一句注释，它概括了函数的功能。函数内部的代码被分成不同的“段落”，每一个“段落”的开头都有一句英文注释概括这一段代码的功能。

程序员们可以从不同的细节层次上阅读这个函数。对一些程序员来说，名字本身就给出了足够的信息。而另一些程序员可能想阅读开始的注释。还有其他一些，可能想阅读所有的注释，但忽略代码。那些对所有细节都感兴趣的程序员才会阅读每一行代码。

由于人类的记忆带有高度的关联性，一旦一个读者从某个细节层次上阅读了这个函数，以后只要从更高的细节层次上阅读，就足以唤起对更详尽层次的记忆。部分原因是即使你在较低的细节层次上阅读代码，为了整理你对这个函数的记忆，较高层次的抽象也会在你的记忆中形成一个框架。

通常，程序体越大，它就需要越多的文档。一个变量至少需要一个单词，一个函数至少

需要一句话，而更大的实体，比如模块和对象，可能就需要一个段落。如果把程序做为一个整体，并写上几页概括性的文档，那将更有帮助。

文档的数量有可能太多了，也有可能太少了。如果人们不去读它，文档就没有任何用处，而人们不大倾向于阅读大段的文本，尤其当它们可能重复时。

人们会把最重要的事情记得最牢，尽管有一些人会被祝福（或诅咒）拥有一个记忆琐事的好脑子。代码中的名字所用的单词是重要的，而至于使用单词的风格，是 *varName*、*VarName*、*varname*、*var_name*、*VARNAME*、*vrblname*，还是 *Variable_Name*，就没那么重要了。重要的是采用单一的风格并一直保持统一，这样程序员就不必浪费时间和脑力去记住哪一种风格在哪种特定的情况下使用。

以下是其他一些提高代码可理解性的方法：

- 尽可能地让作用域局部化。如果对象成员变量可以解决问题，就不要使用全局变量，而如果局部变量可以解决问题，就不要使用对象的成员变量。
- 减小副作用。特别地，除了函数的返回值，避免修改任何变量的值。符合这条规则的函数被称为“可重入”的，这也是一种美丽。它不仅容易理解，而且自动就是线程安全的，还可以被递归调用。除了可读性，副作用少的代码也更容易在不同的情形中被重用。

如今，许多程序员都认识到全局变量能给代码复用带来消极影响。另外一个可能阻碍代码复用的问题是对数据结构的依赖。在这一点上，面向对象的编程风格有时会起反作用。如果一段有用的代码被嵌入到一个对象方法中，我们要使用这段代码就必须先构造一个对象。对某些对象来说，这一任务可能相当复杂。

跟深陷于复杂对象层次结构中的方法相比，一个并未嵌入到对象中，而且参数都是标准数据类型的函数，更有可能被用于不同的上下文中。比如前面提到的 `weedUnlessInHash` 函数，尽管是在基因排序器的列对象中使用，它却被设计成并不依赖于一个列对象。因此，这个有用的小函数现在也被用到其他的上下文中。

结论

这一章的内容是关于本人写过的一段漂亮代码的。程序是服务于生物医学研究者的。行李车系统使得通过 Web 构建交互式程序变得相对容易，即使它使用 CGI 接口。程序的用户模型和开发者模型都围绕着如下思路：一张大表，每行一个基因，以及表示不同类型数据的数目不定的列。

尽管基因排序器是用C语言写的，关于列的代码却是基于直接的、多态的和面向对象的设计的。列的添加可以通过编辑简单的文本文件来实现，无需多写一行代码。同样是这种文件，还使得单一的程序版本能够工作于不同物种的基因组数据库。

这个设计将磁盘寻道降到最低，磁盘寻道到目前为止仍然是计算机系统的性能瓶颈，速度上远远落后于CPU和内存。程序中代码的可读性和可复用都很好。我希望它的设计原理对你自己的程序也能有所帮助。

优雅代码随硬件 发展的演化

Jack Dongarra & Piotr Luszczek

结构先进，价格低廉的计算机迅速普及，对科学计算的所有领域都产生了重大影响。在这一章，我们将研究数学软件中的一个简单而又非常重要的算法：高斯消去求解线性方程组，并通过它来说明算法的设计者需要做的一件事：使算法带上真正的适应性，能迅速适应体系结构的变化。

在应用程序层面，我们需要获取科学的数学模型，而数学模型又被表达成算法并最终被编码成软件。在软件的层面，一方面是性能和可移植性，另一方面是代码的可理解性，它们之间一直需要努力地去平衡。我们将会分析这些问题，重温人们曾经做过的选择和折衷。线性代数——特别的，线性方程组的求解——在大多数科学计算中占据核心位置。近几十年来，人们开发出一些能充分利用高级体系结构计算机能力的线性代数软件，本章将关注这些软件的最新进展。

算法有两个大类：一类是那些面向稠密矩阵（dense matrix）的，另一类是面向稀疏矩阵（sparse matrix）的。如果一个矩阵中包含大量值为 0 元素，它就被称为稀疏矩阵。对于稀疏矩阵来说，通过特殊设计的存储和算法，空间和执行时间上的节省将是非常可观的。为了限制我们的讨论范围以使之保持简单，我们将仅仅考查稠密矩阵问题（稠密矩阵是 0 元素比较少的矩阵）。

在现有的最快的计算机上解决大型的复杂问题的需求，推动了针对高级体系结构计算机的线性代数软件的开发。本章我们将讨论线性代数软件标准的发展、软件库的基础单元，以及算法设计中可能被并行实现所影响的方面。我们将解释这项工作的动因，并简介未来的发展方向。

作为稠密矩阵程序的代表，我们将考虑高斯消去法，或者LU因式分解。这个涵盖了30多年来软硬件发展进步的分析，将反映出所有在高级体系结构计算机上设计线性代数软件时必须考虑的重要因素。我们使用这些因式分解程序来做示例，不仅因为它们相对简单，也是基于它们在多种使用边界元素方法的科学和工程应用中的重要性。这些应用包括电磁散射和计算流体动力学问题。

过去的30年里，人们在解决线性代数问题的算法方面和软件领域做了大量的工作。通过线性代数内核，也就是“基础线性代数子程序”（Basic Linear Algebra Subprograms, BLAS）的建立，人们已在很大程度上实现了这样的目标：在保证代码跨平台的基础上获得高性能运算。我们将讨论 LINPACK, LAPACK 和 ScaLAPACK 库，这3个库分别是用BLAS中3个连续的不同层次来表示的。关于这些库的更多论述参阅本章末尾的“进一步阅读”。

计算机体系结构对矩阵算法的影响

为什么需要在高级体系结构计算机上实现高效的线性代数算法呢？两个关键的动因就是解决存储和数据获取的效率问题。当数据在存储体系的不同层次间移动时，设计者们希望降低它们移动的频率。一旦数据进入寄存器和快速缓存中，针对这些数据所需要的所有处理都应该在它们被换回主存之前完成。因此，为同时利用向量化和并行计算，我们的实现中主要使用了块分割算法，特别是将它跟针对“矩阵-向量”和“矩阵-矩阵”运算（BLAS 中的 Level-2 和 Level-3）所优化过的内核联合使用。块分割意味着数据被分成多个块，每个块应该能够装进一块缓存或者一个向量寄存器文件。

本章所考查的计算机体系包括：

- 向量机
- 拥有缓存体系的 RISC 计算机
- 拥有分布式内存的并行系统
- 多核计算机

向量机出现于 20 世纪 70 年代末 80 年代初。它们将一步运算同时作用于存储在向量寄存

器中的一组操作数上。对这种机器来说，把矩阵算法表示成矩阵－矩阵运算再直接不过了。然而，一些向量算法的设计在将数据读入缓存和写回内存的能力上存在局限性。一种被称为“链接”(chaining)的技术可以克服这个局限，这种技术在访问主存之前在寄存器间移动数据。链接需要将线性代数重新改成矩阵－向量运算。

RISC 计算机是 20 世纪 80 年代末 90 年代初出现的。尽管它们的时钟频率完全可以比得上向量机，但由于没有向量寄存器，其运算速度落后于向量机。另外一个缺点是它有一个层次较深的存储结构，这种结构通过多层的缓存来缓解数据传输带宽的不足，进一步，这种带宽的不足则主要是由内存条的数量有限所导致的。这种架构最终能成功是因为合理的价位和随着时间发展的令人惊讶的性能提升，正如摩尔定律 (Moore's Law) 所预言的那样。在 RISC 计算机上，线性代数算法必须被重新设计。这一次，必须将算法尽可能多地表示成矩阵－矩阵运算，以保证缓存最大程度的复用。

一种达到更高性能级别的很自然的方法就是同时使用向量处理器和 RISC 处理器，通过网络把它们连接起来，让它们合作来解决对单一处理器来说不太可行的复杂问题。许多硬件的结构配置都采用了这条路子，因此矩阵算法最好也沿用一下。很快，人们发现好的局部化性能必须与针对矩阵和向量的好的全局分割结合起来。

对于矩阵数据的任何微不足道的分割，都会很快演变成可伸缩性方面问题，这正如所谓的 Amdahl 定律所阐述的：人们观察到计算中被顺序部分所占去的时间，决定了整个执行时间的下限，于是也限制了并行处理所能带来的效益。换句话说，除非大多数的计算可各自独立进行，否则很快就会到达那个收益递减点，那时再在硬件中加入更多的处理器将不能使计算变得更快了。

为简洁起见，我们在多核体系结构的这一类中同时包括了对称多处理器 (symmetric multiprocessing, SMP) 和单片多核机。这大概是一个不公平的简化，因为对称多处理器 (SMP) 机器中通常拥有更好的存储系统。但当它们被应用于向量算法的时候，两者都可以通过非常相似的算法结构来得到很好的性能结果：这些算法通过对数据依赖的显式控制来将局部缓存和独立计算结合起来。

一种基于分解的方法

在针对稠密线性系统的基本解决方法中，存在一种基于分解的方法。其主要思想是这样的：给定一个引入了矩阵 A 的问题，我们可以把它分解成一些更简单的矩阵的乘积，而针对这些更简单的矩阵，问题的解决就容易得多了。这种方法将计算问题分成了两个部分：首先确定一种恰当的分解；然后是使用这种分解来解决手头的问题。

解决线性系统问题

$$Ax = b$$

这里 A 是一个 n 阶非奇异矩阵。分解方法首先基于如下观察：把 A 分解成以下形式是可能的：

$$A = LU$$

这里 L 是一个对角线上存在 1 元素的下三角阵（“下三角阵”是指对角线上方元素全是 0 的矩阵）。在分解阶段， A 的对角线元素（我们称为“支点”（pivot））被用来分解对角线下方的元素。如果矩阵 A 有一个 0 支点，分解过程将停止于“被 0 除”的错误。而且，小的支点值将严重放大分解过程中的数值错误。因此为了数值稳定，分解方法需要交换矩阵的行，或者确保支点尽可能的大（这里指绝对值大）。基于这个观察，我们引入一个行置换矩阵 P ，且将分解的形式改成了：

$$P^T A = LU$$

于是，方程的解可以被写成：

$$x = A^{-1}Pb$$

这样，方程组就可以用如下方法求解：

1. 将矩阵 A 作因式分解。
2. 求解线性系统 $Ly = Pb$ 。
3. 求解线性系统 $Ux = y$ 。

实践证明，这种通过分解进行矩阵计算的方法非常有用，原因主要有以下几个：第一，这种方法把计算分成了两个步骤：先是计算出一个分解，接着使用这个分解来解决手头的问题。这一点可能非常重要，比如，当所给的右端不同，而且需要在过程的不同时刻解决时。矩阵只需要进行一次因式分解，就可被用于不同的右端。这一点非常重要，因为第一步—— A 的因式分解——需要 $O(n^3)$ 的复杂度，而第二步求解方程，只需 $O(n^2)$ 的复杂度。该算法的另一个优势在于存储： L 因子和 U 因子不需要额外的存储空间，只需接着使用原先被 A 占用的空间。

关于该算法的编码实现，我们只提供整个过程中计算密集的部分，也就是第一步：矩阵的因式分解。

一个简单版本

第一个版本，我们提供了一个 LU 因式分解的直截了当的实现。它由 $n-1$ 步组成，每一步会在对角线下方引入更多的 0 元素，如图 14-1 所示。

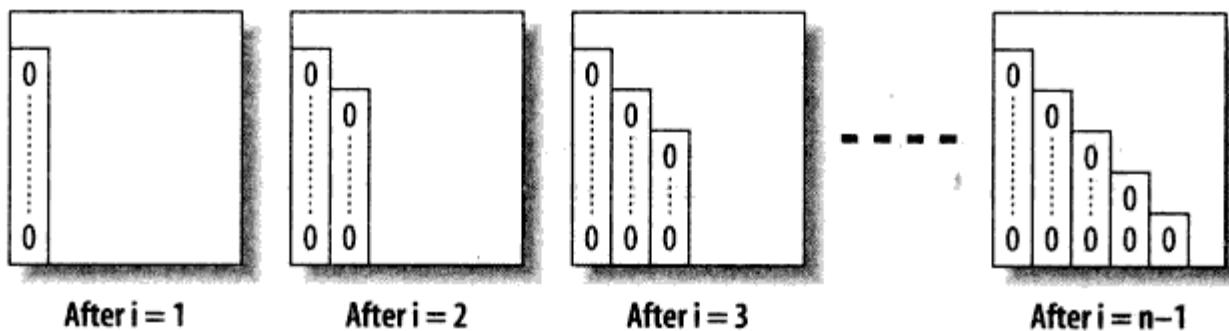


图 14-1：LU 因式分解

在高斯消去法教学中常用的一个工具是 MATLAB。它很像一种也被称作 MATLAB 的脚本语言，这种语言使得矩阵算法的开发非常简易。这种语言对于那些熟悉其他脚本语言的人来说可能感到有些陌生，因为它面向的是多维数组的处理。我们的示例代码所用到的该语言的特性包括：

- 向量和矩阵的变换运算符：'（单引号）
- 如下所述的矩阵索引：
 - 简单整数值: $A(m, k)$
 - 范围 (ranges): $A(k:n, k)$
 - 其他矩阵: $A([k:m], :)$
- 内置矩阵函数，如 `size` (返回矩阵的维数), `tril` (返回矩阵的下三角部分), `triu` (返回矩阵的上三角部分) 和 `eye` (返回一个单位矩阵，也就是对角线上全是 1，其他位置全是 0 的矩阵)

示例 14-1 给出了简单实现。

示例 14-1：简单版本（基于 MATLAB 编码）

```
function [L,U,p] = lutx(A)
%LUTX 三角因式分解, 教科书上的版本
%   [L,U,p] = lutx(A) 产生了一个单位下三角阵 L,
%   一个上三解阵 U, 以及一个置换向量 p,
%   满足 L*U = A(P, :)
%
[n,n] = size(A);
p = (1:n)';
```

```

for k = 1:n-1

    % 找出第 k 列中位于对角线下的最大元素 r 的下标 'm'
    [r,m] = max(abs(A(k:n,k)));
    m = m+k-1; % adjust 'm' so it becomes a global index

    % 如果该列为 0, 跳过消去步骤;
    if (A(m,k) ~= 0)

        % 交换支点行
        if (m ~= k)
            A([k m], :) = A([m k], :); % swap rows 'k' and 'm' of 'A'
            p([k m]) = p([m k]);      % swap entries 'k' and 'm' of 'p'
        end

        % 计算因数
        i = k+1:n;
        A(i,k) = A(i,k)/A(k,k);

        % 更新矩阵的剩余部分
        j = k+1:n;
        A(i,j) = A(i,j) - A(i,k)*A(k,j);
    end
end

% 分割结果矩阵
L = tril(A,-1) + eye(n,n);
U = triu(A);

```

示例 14-1 给出的算法是面向行的，在于我们采用了一个“支点”(pivot) 行的纯量倍数 (scalar multiple)，并把它加到行的下面，从而在对角线下方引入 0 元素。该算法的漂亮之处在于它跟相应的数学表示非常相似。因此，这是第一次给学生讲解该算法时的首选方式，因为学生可以很快将公式转换为可运行的代码。

然而，这种漂亮是有代价的。在 20 世纪 70 年代，Fortran 是科学计算的标准语言。Fortran 按列来存储二维数组。如果通过按行的方式来访问矩阵中的数组，将会引起连续的对彼此相隔一个很大增量的内存位置的引用，相隔的增量大小取决于所声明的数组的大小。该情形被操作系统搞得更加复杂，因为操作系统使用内存页面来有效控制内存的使用。在 Fortran 环境中，对于大的矩阵和一个面向行的算法，软件运行过程中可能产生过多的内存页面交换。Cleve Moler 在 20 世纪 70 年代指出了这个问题（参考本章末尾的“进一步阅读”）。

为了避免这种情形，我们只需简单地交换在变量 i 和 j 上进行的最内层循环的顺序。在 IBM360/67 机器上解决一个矩阵大小为 200 的问题，这个简单的变化将导致 30% 以上的时间节省。使用一个不太明显的循环顺序以及一种晦涩得多的语言（用今天的标准来说），漂亮就这样被效率折衷了。

LINPACK 库中的 DGEFA 子程序

20世纪70年代中期，当向量架构可用于科学计算时，基于MATLAB的实现中的性能问题依然存在。向量架构中采用了流水线处理，通过并发或流水线的方式在数组上执行算术运算。线性代数中的大多数算法都可以被向量化。因此，70年代末期，有人致力于将科学计算中的向量运算标准化。其思路是定义一套简单的、将被频繁调用的操作，并在多种系统上实现它们，以满足可移植性和效率方面的要求。这个程序包被人们称为“一级基础线性代数子程序”（Level-1 Basic Linear Algebra Subprograms，BLAS）或者“一级BLAS”（Level-1 BLAS）。

术语“一级”（Level-1）指的是“向量-向量”运算。我们将会看到，二级（“矩阵-向量”运算）和三级（“矩阵-矩阵”运算）也同样扮演着重要角色。

20世纪70年代，稠密线性代数中的算法在LINPACK项目进行了较为系统的实现。LINPACK是一套Fortran子程序集合，这些子程序用于分析和解决线性方程以及线性最小二乘问题。该程序包能够求解那些基于一般矩阵、带状矩阵、对称不定矩阵、对称正定矩阵、三角阵和对角线方阵的线性系统。此外，该程序包还能计算长方阵的QR和奇异值分解，并将它们应用于最小二乘问题。

LINPACK采用面向列的算法，这可以保持引用的局部性，从而提高了效率。所谓“面向列”是指PINPACK的代码总是按列而不是按行来引用数组的。这很重要，因为Fortran按先列后行的次序存储数组。这意味着，当沿着一列从上到下逐个访问数组中的元素时，内存中的引用也是沿着地址依次连续向后的。这样，如果一个程序引用某个内存块中的数据项，那么下一次引用很可能也在同一块中。

LINPACK的软件能够保持独立于特定机器，这在部分上是由于引入了一级BLAS子程序。几乎所有的计算都是通过调用一级BLAS来完成的。对每一种机器，一级BLAS子过程集合以特定于机器的方式来实现，从而获得高性能。

示例14-2给出了LINPACK实现的因式分解。

示例14-2：LINPACK版本（基于Fortran 66的编码）

```
subroutine dgefa(a,lda,n,ipvt,info)
  integer lda,n,ipvt(1),info
  double precision a(lda,1)
  double precision t
  integer idamax,j,k,kp1,l,nml
c
c      通过局部定支点进行高斯消去。
c
```

LINPACK 中的 DGEFA

```

info = 0
nml = n - 1
if (nml .lt. 1) go to 70
do 60 k = 1, nml
    kp1 = k + 1
c      找到支点下标 1
    l = idamax(n-k+1,a(k,k),1) + k - 1
    ipvt(k) = l
c      zero pivot implies this column already triangularized
c      0 支点意味着这列已被三角形化了。
c
    if (a(l,k) .eq. 0.0d0) go to 40
c      如果需要，就进行交换
    if (l .eq. k) go to 10
    t = a(l,k)
    a(l,k) = a(k,k)
    a(k,k) = t
10   continue
c      计算因子
    t = -1.0d0/a(k,k)
    call dscal(n-k,t,a(k+1,k),1)
c      使用列索引进行消去
    do 30 j = kp1, n
        t = a(l,j)
        if (l .eq. k) go to 20
            a(l,j) = a(k,j)
            a(k,j) = t
20   continue
        call daxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
30   continue
    go to 50
40   continue
    info = k
50   continue
60   continue
70   continue
    ipvt(n) = n
    if (a(n,n) .eq. 0.0d0) info = n
    return
end

```

一级 BLAS 子程序 DAXPY、DSCAL 和 IDAMAX 被用于程序 DGEFA。示例 14-1 和示例 14-2 的主要区别并不在于使用了不同的编程语言，也不在于交换了循环变量的次序，而在于后者是使用 DAXPY 来实现方法的内层循环。

人们假定 BLAS 运算将以高效的，特定于机器的方式实现，即子程序将在哪种机器上运行，实现的方式就要适合那种机器。在一台向量计算机上，运算可被翻译成单条简单的向量运算。这将避免让优化受制于编译器和避免显式地暴露性能关键的运算。

于是，在某种程度上，通过使用新的词汇——BLAS——来描述算法，我们再次获得了最初代码的美。随着时间的流逝，BLAS 成为了一种被广泛采用的标准，并且很可能成为第一个强制贯彻软件的两个关键方面的标准，这两个方面是模块化和可移植性。今天被人们想当然的事情，当初并不是当然的。人们可以写出简单的算法实现，也可以调用这样的算法实现，因为写出来的 Fortran 代码是可移植的。

线性代数中的大多数算法很容易被向量化。然而，要充分利用某种计算机体系结构的性能，简单的向量化常常是不够的。一些向量机受限于存储器跟向量寄存器之间仅有一条通道的事实。如果程序从内存加载向量，执行某些算术运算，然后再存回结果，将形成一个瓶颈。为了获得顶级性能，除了使用向量运算之外，还必须扩展向量化的范围，以便于把运算连接在一起并将数据的移动降到最低。将算法改写成矩阵－向量运算的形式，对向量化的编译器来说，将更容易实现这些目标。

随着机算机体系在存储器层次结构的设计上变得越来越复杂，有必要将BLAS程序的范围从一级扩展到二级、三级了。

LAPACK DGETRF

正如前面提到的，20世纪70年代末80年代初的时候出现的向量机为稠密线性代数算法的开发带来了一种新的方法。这种方法主要关注矩阵和向量的乘法。相对基于1级BLAS的 LINPACK 线性代数子程序来讲，这些新的子程序将带来更高的性能。再晚些时候，20世纪80年代末90年代初，随着RISC类型的微处理器（“killer micros”）和其他带有缓存存储器机器的出现，LAPACK 3级稠密线性代数算法的开发也开始了。3级算法的代码主要以3级BLAS为代表，在这里指的就是矩阵乘法了。

LAPACK项目最初的目标是要让已被广泛使用的LINPACK库能在向量处理器和共享内存并行处理器上高效运行。在这些机器上，LIN-PACK的效率并不高，原因在于其内存访问模式没有考虑这种机器上多层的存储结构，因此把太多的时间耗费在了数据搬运上，而不是做有用的浮点运算。LAPACK把算法重新组织成在最内层循环中使用的分块矩阵运算，比如矩阵乘法，从而解决了这个问题（参阅“进一步阅读”中列出的E. Anderson 和 J. Dongarra 的论文）。这些分块运算可针对每一种体系结构分别进行优化，从而在性能设计中充分考虑它们各自的内存层次结构，于是就提供了一种在各式各样的现代机器上都能达到高性能的可迁移（transportable）的方法。

注意这里我们使用了术语“可迁移”(transportable)，而不是“可移植”(portable)，原因是：为了达到尽可能高的性能，LAPACK要求每种机器上已经实现高度优化的分块矩阵运算。换句话说，代码的正确性是可以移植的，但代码的高性能却无法“移植”——假如我们将自己限制于一份Fortran源代码的话。

从功能上，可将LAPACK看作LINPACK的继任者，尽管它并不总是使用跟LINPACK相同的函数调用次序。作为这样的一个继任者，LAPACK是科学社区的胜利，因为它能够保持LINPACK的功能，同时更好的利用新硬件的性能。

示例14-3给出了LAPACK的LU因式分解方法。

示例14-3：LAPACK因式分解方法

```
SUBROUTINE DGETRF( M, N, A, LDA, IPIV, INFO )
    INTEGER             INFO, LDA, M, N
    INTEGER             IPIV( * )
    DOUBLE PRECISION   A( LDA, * )
    DOUBLE PRECISION   ONE
    PARAMETER          ( ONE = 1.0D+0 )
    INTEGER             I, IINFO, J, JB, NB
    EXTERNAL            DGEMM, DGETF2, DLASWP, DTRSM, XERBLA
    INTEGER             ILAENV
    EXTERNAL            ILAENV
    INTRINSIC           MAX, MIN
    INFO = 0
    IF( M.LT.0 ) THEN
        INFO = -1
    ELSE IF( N.LT.0 ) THEN
        INFO = -2
    ELSE IF( LDA.LT.MAX( 1, M ) ) THEN
        INFO = -4
    END IF
    IF( INFO.NE.0 ) THEN
        CALL XERBLA( 'DGETRF', -INFO )
        RETURN
    END IF
    IF( M.EQ.0 .OR. N.EQ.0 ) RETURN
    NB = ILAENV( 1, 'DGETRF', ' ', M, N, -1, -1 )
    IF( NB.LE.1 .OR. NB.GE.MIN( M, N ) ) THEN
        CALL DGETF2( M, N, A, LDA, IPIV, INFO )
    ELSE
        DO 20 J = 1, MIN( M, N ), NB
            JB = MIN( MIN( M, N )-J+1, NB )
            * 分解主对角和副对角块并检查严格奇异性.
            CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ), IINFO )
            * 调整INFO和支点下标.
            IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - 1
            DO 10 I = J, MIN( M, J+JB-1 )
                IPIV( I ) = J - 1 + IPIV( I )
            CONTINUE
        * 将第1列跟第J-1列交换.
    END IF
END SUBROUTINE DGETRF
```

```

        CALL DLASWP( J-1, A, LDA, J, J+JB-1, IPIV, 1 )
        IF( J+JB.LE.N ) THEN
            * 将第 J+JB 列跟第 N 列交换。
            CALL DLASWP( N-J-JB+1, A( 1, J+JB ), LDA, J, J+JB-1, IPIV, 1 )
        *
        * 计算矩阵 U 的行块。
        CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
        $           N-J-JB+1, ONE, A( J, J ), LDA, A( J, J+JB ), LDA )
        IF( J+JB.LE.M ) THEN
        *
        * 更新结尾子阵 (trailing submatrix)。
        CALL DGEMM( 'No transpose', 'No transpose', M-J-JB+1,
        $           N-J-JB+1, JB, -ONE, A( J+JB, J ), LDA,
        $           A( J, J+JB ), LDA, ONE, A( J+JB, J+JB ), LDA )
        END IF
    END IF
20    CONTINUE
END IF
RETURN
end

```

示例 14-3 的算法中，大多数计算任务包含在 3 个子程序中：

DGEMM

矩阵 – 矩阵乘法；

DTRSM

多右端三角线性方程组求解；

DGETF2

分块列运算中的不分块 LU 因式分解。

算法的一个关键参数就是分块的大小，在这里被称为“NB”。如果 NB 太小或者太大，算法的性能就会很差。ILAEVN 函数也同样重要，此函数的标准实现本来就是要在安装 LAPACK 时，用一个封装了特定机器参数的向量实现来替代。在算法的任意给定时刻，都有 NB 个列或行可以提供给优化过的 3 级 BLAS。如果 NB 是 1，该算法在性能和内存访问模式上就等同于 LINPACK 版本的算法。

对性能和大量不同的计算机体系结构之间的可迁移性来讲，矩阵 – 矩阵运算提供了一个合理的划分模块的层次。这里所说的体系结构包括了采用层次存储结构的并行系统。性能的提高主要来自于更大程度的数据重用。有很多方法可以达到这种数据的重用，从而降低内存读写的频率，并借助于内存层次结构来增大浮点运算相对数据迁移的比率。对现代计算机体系结构来说，这种改善可以带来 3 到 10 倍的效率提升。

人们依然在考查 LAPACK 是否适合高效编码并提供较好的可读性：即将问题的数学描述转换为代码的难度有多大？与 LAPACK 所采用的矩阵表示相比，LINPACK 的向量表示

法可以说是更加自然。而使用数学公式描述算法时，相对于向量—矩阵的混合表示来说，单一的矩阵表示通常更加复杂。

递归 LU

乍看上去，为 LAPACK 的 LU 算法设定分块大小参数似乎并不重要，但在实际运用中，它需要针对各种不同的精度和矩阵大小不断进行调节。许多用户最终将默认设置保持不变，虽然这个调节只需在安装时进行一次，但由于存在许多——而不是一个——需要用到分块大小参数的 LAPACK 子过程，事情就显得更加难办了。

对于 LAPACK 的 LU 表示法，另外一个问题是如何对 DGETF2 过程所处理的又高又窄的列块进行因式分解。它使用了 1 级 BLAS，结果是，当 20 世纪 90 年代处理器的速度越来越快，而存储器的带宽并没有相同程序的提高时，这个问题就成了性能瓶颈。

一种让人意想不到的解决方案是利用分治递归。即在 LAPACK 代码中进行循环的地方，新的 LU 递归算法把工作分成两半，先对矩阵的左半部分进行因式分解，更新矩阵的剩余部分，然后再对右半部分进行因式分解。1 级 BLAS 的使用被降低到一个可以接受的最小限度，而且相对于 LAPACK 算法来讲，大多数 3 级 BLAS 调用所操作的矩阵分块都要大一些。最后，分块大小显然不再需要调节。

递归 LU 需要使用 Fortran 90，Fortran 90 是第一个支持递归子程序的 Fortran 语言标准。使用 Fortran 90 的一个副作用是加大 LDA 参数的重要性，LDA 指的是 A 的主维 (leading dimension)。它不但使得子程序的使用更加灵活，而且还能在矩阵维数 m 可能引起内存 bank 冲突时支持性能调节，内存 bank 冲突会大大降低可用的内存带宽。

当调用外部程序时——比如调用某一种 BLAS 程序——Fortran 90 编译器使用 LDA 参数来避免将数据复制到连续缓冲区中。如果没有 LDA，编译器就只能按最坏的情况进行假设：即输入矩阵 a 可能并不连续，需要被复制到一个临时的连续缓冲区中，从而保证对 BLAS 的调用不至于耗尽内存带宽。而有了 LDA，编译器传递参数指针给 BLAS，无需任何复制。

示例 14-4 给出了递归 LU 因式分解。

示例 14-4：递归版本（Fortran90 编码）

```
recursive subroutine rdgetrf(m, n, a, lda, ipiv, info)
implicit none

integer, intent(in) :: m, n, lda
double precision, intent(inout) :: a(lda,*)
```

```

integer, intent(out) :: ipiv(*)
integer, intent(out) :: info

integer :: mn, nleft, nright, i
double precision :: tmp

double precision :: pone, negone, zero
parameter (pone=1.0d0)
parameter (negone=-1.0d0)
parameter (zero=0.0d0)

intrinsic min
intrinsic max
integer idamax
external dgemm, dtrsm, dlaswp, idamax, dscal

mn = min(m, n)
if (mn .gt. 1) then
    nleft = mn / 2
    nright = n - nleft

    call rdgetrf(m, nleft, a, lda, ipiv, info)
    if (info .ne. 0) return
    call dlaswp(nright, a(1, nleft+1), lda, 1, nleft, ipiv, 1)

    call dtrsm('L', 'L', 'N', 'U', nleft, nright, pone, a, lda,
$           a(1, nleft+1), lda)

    call dgemm('N', 'N', m-nleft, nright, nleft, negone,
$           a(nleft+1,1), lda, a(1, nleft+1), lda, pone,
$           a(nleft+1, nleft+1), lda)

    call rdgetrf(m - nleft, nright, a(nleft+1, nleft+1), lda,
$           ipiv(nleft+1), info)
    if (info .ne. 0) then
        info = info + nleft
        return
    end if

    do i =nleft+1, m
        ipiv(i) = ipiv(i) + nleft
    end do

    call dlaswp(nleft, a, lda, nleft+1, mn, ipiv, 1)

    else if (mn .eq. 1) then
        i = idamax(m, a, 1)
        ipiv(1) = i
        tmp = a(i, 1)
        if (tmp .ne. zero .and. tmp .ne. -zero) then
            call dscal(m, pone/tmp, a, 1)
            a(i,1) = a(1,1)
        end if
    end if
end subroutine

```

```

    a(1,1) = tmp
else
    info = 1
end if

end if

return
end

```

从某种程度上说，这个递归的版本确实优雅。程序中没有出现任何循环。相反，算法被该程序与生俱来的递归性质所驱动（参阅“进一步阅读”中 F.G. Gustavson 的论文）。

如图 14-2 所示，递归 LU 算法共分四大步：

1. 将矩阵分割成两个长方阵 ($m * n/2$)；如果分割的结果是左半部分只有单独一列，则将它按支点的逆进行缩放并返回；
2. 将 LU 算法应用于左半部分；
3. 对右半部分进行变换（求解三角线性方程： $A_{12} = L^{-1}A_{12}$ ，并计算矩阵乘积： $A_{22} = A_{22} - A_{21} * A_{12}$ ）；
4. 将 LU 算法应用于右半部分。

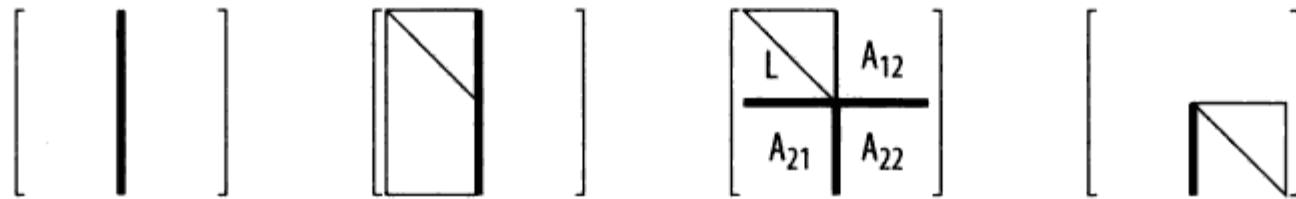


图 14-2：递归 LU 因式分解

在矩阵乘法运算中，大部分的工作都是由矩阵乘法完成的，这些乘法运算作用于连续的大小分别是 $n/2, n/4, n/8 \dots$ 的矩阵。例 14-4 的实现比 14-3 给出的 LAPACK 实现在性能上会高出 10%。

从某种意义上，可以认为先前给出的 LU 算法实现在代码的整洁优雅方面都是退步。但是分治递归却是向前前进了一大步（即使不考虑性能提升）。现在，矩阵因式分解的递归算法可以与其他的递归算法——比如各种排序算法——结合起来一道向学生讲解了。

仅需更改矩阵各部分的大小，就可以得到跟 LINPACK 和 LAPACK 相同的内存访问模式。把 `nleft` 设为 1 就会使代码操作向量，正像 LINPACK 那样。而把 `nleft` 设为 `NB > 1`，代码的行为就像 LAPACK 的分块实现。在这两种情形中，原来的递归都从分治退化

成了最一般的形式。这类递归算法的行为可以跟快速排序同时学习，快速排序对要排序的数组也有多种划分策略。

最后，我们留给读者一个练习：模拟递归代码，但不使用递归，也不能显式处理递归调用栈——如果 Fortran 编译器不能处理递归函数或递归子程序，这将是一个有待解决的重要问题。

ScaLAPACK PDGETRF

LAPACK 被设计成在向量处理器、高性能超标量工作站以及共享内存多处理器上都具有很高的性能。LAPACK 也能令人满意地应用于所有类型的标量计算机上（PC 机、工作站和大型机）。然而，以目前 LAPACK 的形式来看，它可能并不能在其他类型的并行体系结构上提供很高的性能。比如在大规模并行单指令多数据（Single Instruction Multiple Data， SIMD）或在多指令多数据（Multiple Instruction Multiple Data， MIMD）分布式内存机器上。ScaLAPACK 就是为了使 LAPACK 适应于这些新的体系结构而开发的。

通过创建 ScaLAPACK 软件库，我们将 LAPACK 库扩展到可伸缩的 MIMD、分布式内存的并发机器上。对于这类机器，除了寄存器层次结构、高速缓冲和每个处理器的局部存储外，存储器的层次还包括其他处理器的场外处理器存储（off-processor memory）。

与 LAPACK 一样，ScaLAPACK 程序也是基于块分割算法的，这样是为了降低数据在存储器结构的不同层次间移动频率。ScaLAPACK 库的基本单元是 2 级和 3 级 BLAS 的分布式内存版本，以及一套用于通信任务的基础线性代数通信子程序（Basic Linear Algebra Communication Subprograms， BLACS），处理器间通信在并行线性代数计算中是家常便饭。在 ScaLAPACK 程序中，所有的处理器间通信都发生在分布式的 BLAS 和 BLACS 中，因此使用 ScaLAPACK 的上层软件的源代码看起来跟使用 LAPACK 的非常相似。

使用 ScaLAPACK 的 LU 因式分解如示例 14-5 所示。

示例 14-5：ScaLAPACK 版本（Fortran 90 编码）

```
SUBROUTINE PDGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO )
  INTEGER               BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTTYPE_,
$                   LLD_, MB_, M_, NB_, N_, RSRC_
  PARAMETER             ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DTTYPE_ = 1,
$                   CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
$                   RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )
  DOUBLE PRECISION      ONE
  PARAMETER             ( ONE = 1.0D+0 )
  CHARACTER              COLBTOP, COLCTOP, ROWBTOP
```

```

INTEGER I, ICOFF, ICTXT, IINFO, IN, IROFF, J, JB, JN,
$ MN, MYCOL, MYROW, NPCOL, NPROW
INTEGER EXTERNAL IDUM1( 1 ), IDUM2( 1 )
$ EXTERNAL BLACS_GRIDINFO, CHK1MAT, IGAMN2D, PCHK1MAT, PB_TOPGET,
$ INTRINSIC ICEIL
$ MIN, MOD
* 获取网格参数
ICTXT = DESCA( CTXT_ )
CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
* 检查输入参数
INFO = 0
IF( NPROW.EQ.-1 ) THEN
    INFO = -(600+CTXT_)
ELSE
    CALL CHK1MAT( M, 1, N, 2, IA, JA, DESCA, 6, INFO )
    IF( INFO.EQ.0 ) THEN
        IROFF = MOD( IA-1, DESCA( MB_ ) )
        ICOFF = MOD( JA-1, DESCA( NB_ ) )
        IF( IROFF.NE.0 ) THEN
            INFO = -4
        ELSE IF( ICOFF.NE.0 ) THEN
            INFO = -5
        ELSE IF( DESCA( MB_ ).NE.DESCA( NB_ ) ) THEN
            INFO = -(600+NB_)
        END IF
    END IF
    CALL PCHK1MAT( M, 1, N, 2, IA, JA, DESCA, 6, 0, IDUM1, IDUM2, INFO )
END IF
IF( INFO.NE.0 ) THEN
    CALL PXERBLA( ICTXT, 'PDGETRF', -INFO )
    RETURN
END IF
IF( DESCA( M_ ).EQ.1 ) THEN
    IPIV( 1 ) = 1
    RETURN
ELSE IF( M.EQ.0 .OR. N.EQ.0 ) THEN
    RETURN
END IF
* 按行处理时用于通信的开环拓扑结构.
CALL PB_TOPGET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
CALL PB_TOPGET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
CALL PB_TOPGET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', 'S-ring' )
CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', ' ' )
CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', ' ' )
* 单独处理第一个列块
MN = MIN( M, N )
IN = MIN( ICEIL( IA, DESCA( MB_ ) )*DESCA( MB_ ), IA+M-1 )
JN = MIN( ICEIL( JA, DESCA( NB_ ) )*DESCA( NB_ ), JA+MN-1 )
JB = JN - JA + 1
* 分解主对角和副对角块并检查严格奇异性.
CALL PDGETF2( M, JB, A, IA, JA, DESCA, IPIV, INFO )

```

```

* IF( JB+1.LE.N ) THEN
*   将第 JN+1 列跟 JA+N-1 列交换.
*   CALL PDLASWP('Forward', 'Rowwise', N-JB, A, IA, JN+1, DESCA, IA, IN, IPIV )
*   计算 U 的分块行.
*   CALL PDTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
*                 N-JB, ONE, A, IA, JA, DESCA, A, IA, JN+1, DESCA )
*   IF( JB+1.LE.M ) THEN
*     更新结尾子阵.
*     CALL PDGEMM( 'No transpose', 'No transpose', M-JB,N-JB, JB,
*                   -ONE, A, IN+1, JA, DESCA, A, IA, JN+1, DESCA,
*                   ONE, A, IN+1, JN+1, DESCA )
*   END IF
* END IF
* 循环处理剩余的列块.
DO 10 J = JN+1, JA+MN-1, DESCA( NB_ )
  JB = MIN( MN-J+JA, DESCA( NB_ ) )
  I = IA + J - JA
*
* 分解主对角和副对角块并检查严格奇异性.
*
  CALL PDGETF2( M-J+JA, JB, A, I, J, DESCA, IPIV, IINFO )
*
* IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - JA
*
* 将第 JA 列与 J-JA 列交换.
*
  CALL PDLASWP('Forward', 'Rowwise', J-JA, A, IA, JA, DESCA, I,I+JB-1, IPIV)
  IF( J-JA+JB+1.LE.N ) THEN
    将 J+JB 列跟 JA+N-1 列交换.
    CALL PDLASWP( 'Forward', 'Rowwise', N-J-JB+JA, A, IA, J+JB,
                  DESCA, I, I+JB-1, IPIV )
  计算 U 的分块行.
  CALL PDTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
                N-J-JB+JA, ONE, A, I, J, DESCA, A, I, J+JB,
                DESCA )
  IF( J-JA+JB+1.LE.M ) THEN
    更新结尾子矩阵.
    CALL PDGEMM( 'No transpose', 'No transpose', M-J-JB+JA,
                  N-J-JB+JA, JB, -ONE, A, I+JB, J, DESCA, A,
                  I, J+JB, DESCA, ONE, A, I+JB, J+JB, DESCA )
  END IF
END IF
10 CONTINUE
IF( INFO.EQ.0 ) INFO = MN + 1
CALL IGAMN2D(ICTXT, 'Rowwise', ' ', 1, 1, INFO, 1, IDUM1, IDUM2, -1,-1, MYCOL)
IF( INFO.EQ.MN+1 ) INFO = 0
CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
RETURN
END

```

为了简化ScaLAPACK的设计，同时也因为BLAS被证明是一种可以独立于LAPACK的

非常有用的工具，我们决定开发一个并行的BLAS，或者叫PBLAS（详见“进一步阅读”中列出的Choi等人的论文），这个PBLAS的接口将保持与BLAS最大程度的相似性。这一决定使得ScaLAPACK代码与同样功能的LAPACK代码非常相似，有时几乎完全相同。

我们的目标是PBLAS能够提供了一个分布式内存的标准，正像BLAS提供了共享内存的标准一样。这将简化并促进高性能、可移植的并行算术软件的开发，同时只要求机器供应商对少量的程序进行优化。要想让人们接受PBLAS，需要在功能和易用性之间作合理的折衷。

PBLAS操作的矩阵分布成二维循环分块布局。由于这样一种数据布局需要很多参数来全面描述矩阵分布，我们选择了一个更加面向对象的方法，把这些参数封装在一个叫做“数组描述符”的整数数组中。一个数组描述符包含以下信息：

- 描述符类型；
- BLACS上下文（BLACS context）——一个为避免逻辑上不同的消息之间发生冲突而创建的虚拟消息空间；
- 分布式矩阵的行数；
- 分布式矩阵的列数；
- 行分块大小；
- 列分块大小；
- 矩阵的第一行分布于其上的处理行（process row）；
- 矩阵的第一列分布于其上的处理列（process column）；
- 存储局部部分块的局部数组的主维。

通过使用描述符，对一个PBLAS程序的调用就与一个对相应BLAS程序的调用非常相似了。

```
CALL DGEMM ( TRANS, TRANSB, M, N, K, ALPHA,
             A( IA, JA ), LDA,
             B( IB, JB ), LDB, BETA,
             C( IC, JC ), LDC )

CALL PDGEMM( TRANS, TRANSB, M, N, K, ALPHA,
             A, IA, JA, DESC_A,
             B, JB, DESC_B, BETA,
             C, IC, JC, DESC_C )
```

DGEMM 计算 $C = BETA * C + ALPHA * op(A) * op(B)$, 其中 $op(A)$ 根据 $TRANSA$ 的结果, 要么是 A , 要么是它的转置; $op(B)$ 类似, $op(A)$ 是 $M*K$ 的, $op(B)$ 是 $K*N$ 的。 $PDGEMM$ 与此类似, 除了指定子矩阵的方式不同外。比如, 为了把从 $A(IA,JA)$ 开始的子矩阵传递给 $DGEMM$, 对应于形参 A 的实参就是一个简单的 $A(IA,JA)$ 。 $PDGEMM$ 却不同, 它需要知道 A 的全局存储状况才能正确的提取子矩阵, 因此 IA 和 JA 必须单独传入。

$DESC_A$ 是 A 的数组描述符。描述矩阵操作数 B 和 C 的参数与描述 A 的参数相似。在一个真正的面向对象的环境中, 矩阵和 $DESC_A$ 的含义是相同的。这不仅需要语言支持, 也会降低其可移植性。

通过使用 ScaLAPACK 库中的消息传递和可伸缩的算法, 不管矩阵的大小增加到多少, 只要拥有处理器足够多的计算机, 对它进行因式分解都是可能的。按照 ScaLAPACK 库的设计, 其中的计算应多于消息通信, 因此对库的大部分来讲, 数据都是被保留在局部进行处理, 只是偶尔通过网络传送一下。

但是, 处理器之间需要交换的消息的数量和类型有时可能难于管理。与每一个分布式矩阵关联的上下文使得实现中可以使用相互独立的“空间”进行消息传递。不同的库(或不同的库调用)使用的独立的通信上下文, 这使得库内部的通信与外部的通信隔离开来。当为 PBLAS 中某个程序的参数列表提供了多个描述符数组时, 各个 BLACS 上下文的入口必需相同。换言之, PBLAS 不执行任何上下文之间的交互操作。

从性能上讲, ScaLAPACK 对 LAPACK 所做的事, 正如 LAPACK 对 LINPACK 所做的: 它扩大了可以让 LU 因式分解(还有其他代码)高效运行的硬件范围。从代码是否优雅的角度来看, ScaLAPACK 对 LAPACK 所做的改变更大: 同样的算术运算现在需要大量冗繁的处理。现在用户和库的开发者都被迫对错综复杂的数据存储做显式的控制, 因为数据的局部性成为影响性能的最关键因素。受伤害的是代码的可读性, 即使按照目前最好的软件工程实践来划分代码的模块也是如此。

针对多核系统的多线程设计

多核芯片的出现给软件的生产方式带来了根本性的转变。稠密线性代数软件也不例外。一个好消息是 LAPACK 的 LU 因式分解可以直接在多核系统上跑, 如果使用的是多线程版本的 BLAS, 甚至还能带来性能方面不小的提升。用行话说, 这是“fork-join”模型的计算: 每一个 BLAS 调用(从单一的主线程中)产生出适当数量的分支线程(fork), 这些线程在各个“核”上分别执行运算, 然后再汇合到主线程中(join)。Fork-join 模型意味着在每一个 join 操作上需要一个同步点。

不好的消息是 LAPACK 的 fork-join 算法会严重削弱程序的可伸缩性，即使在那些没有 SMP 内存的小型多核计算机上也是如此。算法固有的可伸缩性缺陷在于 fork-join 模型中大量的同步动作（同时只允许一个线程执行占据临界区重要计算，其他线程只能空闲），同步导致了锁步（lock-step）执行，同时也使得代码中内在的顺序部分不能隐藏于并行部分之后。换句话说，线程被迫在不同的数据上执行同样的运算。如果某些线程得不到足够的数据，它们就不得不空闲下来，等待尚在数据上执行有效运算的其他线程。显然，我们需要另一个版本的 LU 算法，通过让线程在程序运行的某些阶段执行不同的操作，可以使其一直保持忙碌。

多线程版本的算法认识到在算法中存在着所谓的“关键路径”：代码中的一些部分，它的执行依赖于先前的计算并能阻塞算法的执行。LAPACK 的 LU 算法没有对这部分关键代码做任何特殊处理：DGETF2 子程序被单个线程调用，即使在 BLAS 级别下也不支持太多的并行。当一个线程调用此程序时，其他线程就只能空闲等待。而且由于 DGETF2 的性能主要受制于内存带宽（而不是处理器速度），当系统中引入多核时，这一瓶颈将使可伸缩性问题进一步加剧。

多线程版本的算法引入了“前瞻”（look-ahead）技术，从正面直接解决这个问题：提前进行相关计算以避免计算过程中潜在的停滞。这当然需要比前一版本更多的同步和薄记工作——代码复杂性和性能之间的折中。多线程代码的另一个特性是小组因式分解中递归的使用。事实证明，使用递归在狭长的小组矩阵上获得的性能提升比方阵上获得的还要高。

示例 14-6 给出了适应多线程执行的因式分解。

示例 14-6：多线程执行环境中的因式分解（C 代码）

```
void SMP_dgetrf(int n, double *a, int lda, int *ipiv, int pw,
                  int tid, int tsize, int *pready, ptm *mtx, ptc *cnd)
{
    int pcnt, pfctr, ufrom, uto, ifrom, p;
    double *pa = a, *pl, *pf, *lp;

    pcnt = n / pw; /* 小组的数目 */

    pfctr = tid + (tid ? 0 : tsize);
    /* 在 0 号小组被分解之后，应该被当前线程分解的第一个小组 */

    /* 这是指向最后一个小组 (panel) 的指针 */
    lp = a + (size_t)(n - pw) * (size_t)lda;

    /* 对每一个用作更新源的小组 */
    for (ufrom = 0; ufrom < pcnt; ufrom++, pa += (size_t)pw * (size_t)(lda + 1)){
        p = ufrom * pw; /* 列号 */
```

```

/* 如果用于更新的小组尚未分解，则不考虑 'ipiv'，尽可能避免对 'pready' 的访问。 */
if (! ipiv[p + pw - 1] || ! pready[ufrom]) {

    if (ufrom % tsize == tid) { /* 如果这个小组是由当前线程处理的 */
        pfactor( n - p, pw, pa, lda, ipiv + p, pready, ufrom, mtx, cnd );
    } else if (ufrom < pcnt - 1) { /* 如果这不是最后一个小组 */
        LOCK( mtx );
        while (! pready[ufrom]) { WAIT( cnd, mtx ); }
        UNLOCK( mtx );
    }
}

/* 对每一个将被更新的小组 */
for (uto = first_panel_to_update( ufrom, tid, tsize ); uto < pcnt;
     uto += tsize) {

/* 如果仍有小组可被当前线程分解，而且先前的小组已被分解过了，
则可以跳过对 'ipiv' 的检查，减少对 'pready' 的访问次数。 */
    if (pfctr < pcnt && ipiv[pfctr * pw - 1] && pready[pfctr - 1]) {
        /* 对每一个仍需更新 'pfctr' 的小组 */
        for (ifrom = ufrom + (uto > pfctr ? 1 : 0); ifrom < pfctr; ifrom++) {
            p = ifrom * pw;
            pl = a + (size_t)p * (size_t)(lda + 1);
            pf = pl + (size_t)(pfctr - ifrom) * (size_t)pw * (size_t)lda;
            pupdate( n - p, pw, pl, pf, lda, p, ipiv, lp );
        }
        p = pfctr * pw;
        pl = a + (size_t)p * (size_t)(lda + 1);
        pfactor( n - p, pw, pl, lda, ipiv + p, pready, pfctr, mtx, cnd );
        pfctr += tsize; /* 移动到当前线程的下一个小组 */
    }

    /* 如果 'uto' 尚未分解（假如它被分解过，自然就被更新过，也就没必要再更新了） */
    if (uto > pfctr || ! ipiv[uto * pw]) {
        p = ufrom * pw;
        pf = pa + (size_t)(uto - ufrom) * (size_t)pw * (size_t)lda;
        pupdate( n - p, pw, pa, pf, lda, p, ipiv, lp );
    }
}
}
}

```

算法在每个线程中都做相同的事情 (SIMD 范例)，矩阵中的数据通过一种循环的方式划分到各个线程中，这种方式使用每一组都带有 pw 列的小组（除最后一组外）。Pw 参数对应 LAPACK 中的分块参数 NB。不同之处在于小组（由列构成的块）在逻辑上被分配给不同的线程（在物理上，所有的小组都同样可被访问，因为代码是在一个共享内存的环境中运行）。在线程中为矩阵分块，好处就跟在 LAPACK 中分块一样：更好的高速缓存利用率和更低的内存带宽压力。将矩阵的一部分分配给一个线程乍看起来似乎是人为需求，但它简化了代码和用于薄记的数据结构；最重要的是，通过它获得更好的内存局部性。事实显示多核芯片在内存访问带宽方面并不具有对称性，因此减少内存分页在各个核之间重新分配的次数直接提高了性能。

`pfactor()`和`pupdate()`函数代表了LU因式分解的标准要素。正如人们所想，前者分解一个小组，后者使用一个前面已经分解了的小组来更新另一个小组。

主循环使每一个线程依次在各个小组上迭代。必要时，拥有那个小组的线程会分解小组，而其他的线程则等待（如果它们正好需要这个小组来完成更新的话）。

前瞻逻辑存在于内层循环中（跟在对每个小组进行更新的注释之后），内层循环替代了以前算法中的DGEMM或者PDGEMM。在每个线程更新它的一个小组之前，它会检查是否已经可以分解它的第一个尚未分解的小组。这减少了各个线程需要等待的次数，因为每个线程都不断尝试消除潜在的瓶颈。

与ScaLAPACK相同，多线程版本的代码不再有LAPACK版本那种与生俱来的优雅。同样这也都是为了性能：LAPACK代码在“核”不断增加的机器上不能高效运行。在LAPACK层面而不是BLAS层面显式控制执行线程是很重要的：并行无法被封装到一个库调用中。惟一的好消息是代码不像ScaLAPACK版本的那么复杂，而且高效的BLAS仍然被有效利用着。

误差分析与操作计数浅析

本章中所有算法实现的一个关键方面是它们的数值属性。

放弃代码的优雅来换取性能是可以接受的。但数值的稳定性至关重要，不可以被牺牲，它本来就是算法正确性的一部分。这方面的考虑是马虎不得的，但同时也存在其他一些要求不这么严格的地方。对某些读者来说，所有的算法都相同或许很奇怪，即使事实上不可能让每一段代码对相同的输入都产生完全相同的输出。

对于结果的可重复性，浮点数表示的不确定性可以通过误差范围来严格的测量。对于先前的算法，一种表达数值健壮性的方式是使用如下公式：

$$\|r\|/\|A\| \leq \|e\| \leq \|A^{-1}\| \|r\|$$

其中，误差 $e = x - y$ ，指的是计算解 y 与正确解 x 的差， $r = Ay - b$ 即所谓的“余差”。上面公式的大致意思是：误差（数值两边的平行竖线代表一个范化值——一个表示绝对大小的量）与矩阵 A 本身的品质所能给出的一样小。因此，如果矩阵在数值意义上接近于奇异矩阵（某些入口足够小，以致他们也可以被看作0），那么算法不会给出一个精确解。但另一方面，一个品质相对好的结果是可以期望的。

本章给出的所有算法版本的另一个共同特点是操作计数：它们都执行 $2/3n^3$ 次浮点乘法

或加法。正是这些操作的量级能衡量出这些算法间的差别。有些算法增加浮点运算的数量来减小内存存取或网络传输（特别是分布式内存的并行算法中）。由于本章给出的算法都拥有相同的操作计数，比较它们的性能是有意义的。给定相同大小的矩阵，可将运算速率（每秒执行浮点运算的次数）进行比较，而不是比较解决问题所用的时间。但比较运算速率有时更好，因为它在矩阵大小不同时也可以用于比较算法。举例来说，一个单一处理器上的顺序算法可以直接跟一个运行于大型集群上、计算更大矩阵的并行算法相比较。

未来的研究方向

在这一章中，我们分析了计算科学中的一个简单而又重要的算法在设计上的演化。过去30年来的变化是伴随着计算机体系结构的前进而进行的。在某些情形中，这种改变相对简单，比如交换循环的次序。而另一些情形中，改变会复杂到引入递归和前瞻计算。然而，在每一种情形中，代码高效地利用存储器层次结构是获取高性能的关键，在单一处理器上如此，在共享和分布式内存系统中亦是如此。

问题的关键是软件开发者过去必须面对、现在也仍然要面对的大规模增长的程序复杂性。双核机器已是很普及了，而按照预期，处理器中核的数量基本上会随着每一代处理器而增加一倍。但跟旧式计算模型的假定不同：程序员不能孤立地考虑这些核（也就是说，多核并非“新式 SMP”），这是因为它们共享芯片资源的方式在独立多处理器中是没有的。而这一情形又被其他一些未来计算机体系结构可能采用的非标准组件搞得更加复杂，这其中包括混合不同类型的核、硬件加速器和存储系统。

最后，各种迥乎不同的设计思想的不断出现，表明了关于怎样才能最好地结合所有这些新资源和组件的问题还远远没有得到解决。这些变化给出了一幅关于当它们被结合起来时未来的图画，在这幅图画里，程序员为了充分利用新体系所提供的更高程度的并行性和更强大的计算能力，必需去克服比以前更加复杂、更加具有挑战性的软件设计问题。

所以，坏消息是本章给出的每一种代码，终有一天都将不再高效。好消息是我们已经学到了多种用来重新组织原来算法结构的方法，从而使它得以面对不断增长的硬件设计的挑战。

进一步阅读

- LINPACK User's Guide, J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, SIAM: Philadelphia, 1979, ISBN 0-89871-172-X.

- LAPACK Users' Guide, Third Edition, E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, SIAM: Philadelphia, 1999, ISBN 0-89871-447-8.
- ScaLAPACK Users' Guide, L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, SIAM Publications, Philadelphia, 1997, ISBN 0-89871-397-8.
- "Basic Linear Algebra Subprograms for FORTRAN usage," C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, ACM Trans. Math. Soft., Vol. 5, pp. 308-323, 1979.
- "An extended set of FORTRAN Basic Linear Algebra Subprograms," J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, ACM Trans. Math. Soft., Vol. 14, pp. 1-17, 1988.
- "A set of Level 3 Basic Linear Algebra Subprograms," J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, ACM Trans. Math. Soft., Vol. 16, pp. 1-17, 1990.
- Implementation Guide for LAPACK, E. Anderson and J. Dongarra, UT-CS-90-101, April 1990.
- A Proposal for a Set of Parallel Basic Linear Algebra Subprograms, J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, UT-CS-95-292, May 1995.
- LAPACK Working Note 37: Two Dimensional Basic Linear Algebra Communication Subprograms, J. Dongarra and R. A. van de Geijn, University of Tennessee Computer Science Technical Report, UT-CS-91-138, October 1991.
- "Matrix computations with Fortran and paging," Cleve B. Moler, Communications of the ACM, 15(4), pp. 268-270, 1972.
- LAPACK Working Note 19: Evaluating Block Algorithm Variants in LAPACK, E. Anderson and J. Dongarra, University of Tennessee Computer Science Technical Report, UT-CS-90-103, April 1990.
- "Recursion leads to automatic variable blocking for dense linear-algebra algorithms," Gustavson, F. G. IBM J. Res. Dev. Vol. 41, No. 6 (Nov. 1997), pp. 737-756.

漂亮的设计会给你带来 长远的益处

Adam Kolawa

有些看上去很简单直接的算法，在实现时却会变得很困难。例如：取整操作会带来精度问题；有些数学方程式还会造成系统的浮点溢出；还有一些算法（尤其是经典的傅立叶变换，Fourier Transform），当采用暴力法进行计算时则耗时太长。而且，不同的数据集合所适用的算法也不一样。因此，漂亮的代码和漂亮的数学表达形式间并没有固定的联系。

编写CERN数学库的程序员意识到了数学方程式和计算机实现之间的不同：理论和实践的区别。在本章中，我将会向大家展示他们在代码中所采用的一些用以跨越该差异的编程策略，以及这些策略的美妙之处。

对于漂亮代码的个人看法

对于什么样的代码才是漂亮代码，我的最基本的观点是：它必须是可工作的代码。换句话说，代码必须精确、高效地完成我们在设计它时所预期的任务，而不能在行为上存在任何的歧义。

只有在那些我所信任的代码中，我才可以发现到美——因为我确信这些代码可以适用

于我所碰到的问题上，并且计算出正确的答案。在这我对“漂亮代码”的标准做了第一次定义：只有那些我可以使用，以及重复使用，并且不会对它们存在任何“它们是否可以为我提供正确答案”怀疑的代码才可以被称为漂亮的代码。换句话说，我主要关注的不是代码看上去怎么样，而是我可以用它来做什么。

我并不是说我会忽视代码实现细节中的美；实际上我还是很看重这点的，在本章的后半部分，我将会对代码的内在美进行自己的定义以及给出相应的例子。我的观点是，当代码可以满足我那稍显偏离传统的关于美的观念时，我们就没有必要去深究它实现的细节了。这样的代码也促使我产生了我认为在业界最重要的一个使命：与他人共享代码，同时并不要求别人去分析代码内部以找出它的工作原理。漂亮的代码就应该像一辆漂亮的小汽车。我们很少会去研究小汽车内部的机械原理，相反，我们只从外观欣赏它，并且信任它，驾驶它去我们所想去的地方。

为了（和享用小汽车）同样地享有代码，我们必须小心地设计代码，使得它们能够足够清晰地告诉别人如何使用它们、理解它们、将它们应用到人们所面对的问题上去、并且还能够很容易让人们验证它的用法是否正确。

对于 CERN 库的介绍

我认为，由 CERN (European Organization for Nuclear Research, 欧洲核子研究中心) 所开发的数学计算库是一个用来展示漂亮代码的很好的例子。该库中的代码完成了如下功能：代数运算、积分函数、解微分方程以及解决了大量的物理问题。这个库开发于 30 年前，多年来已得到了广泛的应用。CERN 库中的线性代数部分现在已经演化成了名为 LAPACK 的库，在本章中我所选取的代码就摘自于这个库。目前，LAPACK 库主要是有几个大学和机构在研发。

从年轻时我就开始使用该库了，到现在我仍然痴迷于它。该库的美妙之处在于，它实现了很多复杂的数学算法，这些算法都被很好地测试过并且还很难被重新编写。我们可以重用它、信赖它、且无需担心它是否工作。

即便过去了这么多年，该库的高准确度以及高可靠性仍然使得它是所有希望准确可靠地解决方程的人的首选数学库。虽说世上还存在着其他的数学库，但它们都不能在可靠性和准确度方面与 CERN 库相抗衡。

本章的第一部分将讨论代码的外在美 (*outer beauty*)：即那些保证代码准确以及可靠并使得开发人员愿意重用它的因素，以及如何使重用变得简单的元素。第二部分则从实现细节方面探讨了代码的内在美 (*inner beauty*)。

外在美

如果你有过试图依靠编程来解决线性方程组或类似的复杂数学运算的经验的话，那么你就会知道，在很多时候，你所编写的代码实际上并不会为你产生正确的结果。对于数学库来说，最大的问题之一在于运算过程中出现的取整问题或者浮点运算导致结果易变且不正确的问题。

在设计数学库时，我们需要仔细地界定库中每个算法可工作的范围。我们必须在编写每个算法时能确保它会遵守这些限制条件，同时我们还必须确保所有的取整错误会导致算法退出。这可以变得非常复杂。

在CERN库中，所有的算法都以一种极为精确的方式被定义。基本上，如果你能看到所有例程，你就会注意到你所看到的每个例程（routine），都有一个描述它能做什么的描述，至于该例程是用什么语言编写的则无所谓。实际上，在CERN库中，所有的例程都是用Fortran编写的，不过它们还是提供了几乎可以被其他任意语言调用的接口。这点做得很漂亮。从某种意义上来说，这些例程就是一个黑盒，我们不需要关心它的内部细节如何，只要知道它可以从我们的输入中得出正确的结果就可以了。在CERN库中，所有的例程都被仔细地定义，包括：它能做什么；在哪种条件下它才可以工作；它所接受的输入数据如何；以及为了得到正确的结果，我们又必须对输入的数据加上什么样的限制。

我们选取了LAPACK库中的SGBSV例程作为我们的例子，它可以为一个带状矩阵（banded matrix，即矩阵中不为0的值集中在矩阵对角线附近）求解线性方程组。对不同的线性方程组求解，我们会使用不同的算法；而不同的算法在不同的领域其性能也不一样，这就要求我们必须知道矩阵的结构，以选取最佳的算法。例如，我们可能会选取一种算法来解决和带状矩阵相关的问题，选取另外一种算法来处理稀疏矩阵（sparse matrix，矩阵中大部分元素都是0）。

由于不同的例程被优化成适用于不同的场景，我们必须根据矩阵的结构选取最佳的例程来进行我们的运算。然而，为了理解这些限制，我们就必须知道如何将输入数据提交给例程。有时，我们会将输入数据封装成矩阵的形式。有时——例如，带状矩阵——我们使用一个数组来封装它。所有的这些例程以及他们的需求在库中都被很好的描述，如下所示。

```
SUBROUTINE SGBSV( N, KL, KU, NRHS, AB, LDAB, IPIV, B, LDB, INFO )
*
* -- LAPACK 驱动例程 (版本 2.0) --
*   田纳西大学, 加利福尼亚大学伯克利分校, NAG有限公司,
```

* 科朗研究院, 阿贡国家实验室以及莱斯大学
* 1993 年 3 月 31 日

汉译版

* .. 标量参数 ..
* INTEGER INFO, KL, KU, LDAB, LDB, N, NRHS
* .. 数组参数 ..
* INTEGER IPIV(*)
* REAL AB(LDAB,*), B(LDB,*)
* ..
* 功能 ..
* ======
* SGBSV 计算线性方程 $A * X = B$ 的解, 其中 A 是一个带状矩阵, 秩为 N ,
* 带有 KL 次对角和 KU 超对角, 而 X 和 B 是 $N \times NRHS$ 矩阵。
* 通过列主元素消去和行交换的 LU 分解法把 A 分解为 $A = L * U$,
* 其中 L 是带有 KL 次对角的下三角矩阵, 而 U 是带有 $KL+KU$ 超对角的上三角矩阵。
* A 的分解形式将用于求解方程 $A * X = B$.
* 参数 ..
* ======
* N (输入) INTEGER
* 线性方程的数量, 即矩阵 A 的秩。 $N \geq 0$.
* KL (输入) INTEGER
* 带状矩阵 A 中次对角的数量。 $KL \geq 0$.
* KU (输入) INTEGER
* 带状矩阵 A 中超对角的数量。 $KU \geq 0$.
* NRHS (输入) INTEGER
* 右列的数量, 即矩阵 B 的列数。 $NRHS \geq 0$.
* AB (输入 / 输出) REAL 数组, 维数为 (LDAB, N)
* 在例程入口处, 矩阵 A 在行 $KL+1$ 到行 $2*KL+KU+1$ 是带状存储;
* 从第 1 行到第 KL 行的数组元素不需要设置。
* 矩阵 A 的第 j 行存储在数组 AB 的第 j 列, 如下所示:
* $AB(KL+KU+1+i-j, j) = A(i, j)$ 其中 $\max(1, j-KU) \leq i \leq \min(N, j+KL)$.
* 在例程出口处, 因式分解的结果为: U 中存储的是上三角带状矩阵,
* 其中在第 1 行到第 $KL+KU+1$ 行是 $KL+KU$ 超对角,
* 在因式分解中使用的乘积被保存在第 $KL+KU+2$ 行到第 $2*KL+KU+1$ 行中.
* 详细内容请参见下面的注释。
* LDAB (输入) INTEGER
* 数组 AB 的主维 (leading dimension). $LDAB \geq 2*KL+KU+1$.
* IPIV (输出) INTEGER 数组, 维数为 (N)
* 定义置换矩阵 P 的索引; 矩阵的第 i 行将与第 $IPIV(i)$ 行交换。
* B (输入 / 输出) REAL 数组, 维数为 (LDB, NRHS)
* 在例程入口处, 是 $N \times NRHS$ 的矩阵 B .
* 在例程出口处, 如果 $INFO = 0$, 将是 $N \times NRHS$ 的结果矩阵 X .

```

*
* LDB      (输入) INTEGER
*          数组B的主维。 LDB >= max(1,N)。
*
* INFO     (输出) INTEGER
*          = 0: 成功退出
*          < 0: 如果 INFO = -i, 那么第 i 个参数的值是非法的
*          > 0: 如果 INFO = i, 那么 U(i,i) 等于 0。因式分解
*                  完成了, 但 U 是奇异矩阵, 因此无法计算结果。
*
* 更多详细内容
* =====
*
* 在以下示例中说明了带状存储模式, 其中
* M = N = 6, KL = 2, KU = 1:
*
* 人口:                                出口:
*
*      *   *   *   +   +   +
*      *   *   +   +   +   +
*      *   a12 a23 a34 a45 a56   *   u14   u25   u36
*      a11 a22 a33 a44 a55 a66   *   u13   u24   u35   u46
*      a21 a32 a43 a54 a65   *   u12   u23   u34   u45   u56
*      a31 a42 a53 a64   *   u11   u22   u33   u44   u55   u66
*      a12 a23 a34 a45 a56   *   m21   m32   m43   m54   m65   *
*      a21 a32 a43 a54 a65   *   m31   m42   m53   m64   *
*      a31 a42 a53 a64   *   *
*
* 在例程中将不会用到带有 * 标识的数组元素; 带有 + 标识的元素在例程入口处不需要设置,
* 但例程需要用这些元素来存储 U 的元素, 因为行交换将导致填充 (fill-in) 运算。
*
* =====
*
* .. 外部子例程 ..
* EXTERNAL SGBTRF, SGBTRS, XERBLA
* ..
* .. 内部函数 ..
* INTRINSIC MAX
*
* .. 可以执行语句 ..
*
* 检测输入参数。
*
* IF( INFO .NE. 0 ) THEN
*    INFO = -1
* ELSE IF( N.LT.0 ) THEN
*    INFO = -2
* ELSE IF( KL.LT.0 ) THEN
*    INFO = -3
* ELSE IF( NRHS.LT.0 ) THEN
*    INFO = -4
* ELSE IF( LDAB.LT.2*KL+KU+1 ) THEN
*    INFO = -5
* ELSE IF( LDB.LT.MAX( N, 1 ) ) THEN
*    INFO = -6
* ELSE IF( LDB.LT.1 ) THEN
*    INFO = -7
* ELSE IF( KU.LT.0 ) THEN
*    INFO = -8
* END IF

```

```

IF( INFO.NE.0 ) THEN
    CALL XERBLA( 'SGBSV ', -INFO )
    RETURN
END IF
*
* 计算带状矩阵 A 的 LU 因式分解。
*
CALL SGBTRF ( N, N, KL, KU, AB, LDAB, IPIV, INFO )
IF( INFO.EQ.0 ) THEN
*
* 求解  $A^*X = B$ , 用 X 覆盖 B。
*
CALL SGBTRS( 'No transpose', N, KL, KU, NRHS, AB, LDAB, IPIV,
$                 B, LDB, INFO )
END IF
RETURN
*
* SGBSV 结束
*
END

```

在 SGBSV 例程代码中，我们首先注意到的是，它的开始处是一大串的注释，这些注释描述了该例程的功能和使用方法。实际上，这些注释也起到了手册的作用。在代码中用注释描述如何使用该例程的方法很重要，因为这样我们就把例程的使用和它的内部结构联系到一起了。在其他很多地方，我发现手册中的描述和代码中的文档并没有任何联系。我认为把它们两者联系到一起可以使得代码变得漂亮起来。

在开始的注释后面，在例程的描述处我们详细地描述了该例程中所使用的算法。这有助于使用这些代码的用户理解代码是如何工作的，对于输入代码又该如何反应。接下来就是对参数的详细描述，并且在描述中我们还确定了他们的取值范围。

此处参数 AB 很有意思，值得我们思考。这个参数包含了矩阵 A 的所有元素。由于矩阵是带状的，它里面包含了大量的 0 值，而且这些 0 值也没有靠近矩阵的对角线。原则上来说，我们可以用一个二维数组来表示例程的输入。然而，这将会导致大量的内存空间被浪费。为了节约内存空间，我们使用了 AB 这个参数，在 AB 中只包含有矩阵中靠近其对角线的那些非 0 元素。

参数 AB 的格式不但可以节省内存，它还有着另外一个目的。在本例程中，算法利用了方程组的一些特性，从而以一种更高效的方式对问题进行求解。这也意味着，算法的输入依赖于用户提供恰当格式的矩阵。如果参数 AB 中包含了矩阵中的所有元素，在带状之外的一个（或多个）元素被无意间设为非 0 值，那么这就将导致错误的产生。对 AB 格式的选择使得这种错误的出现变得不可能。这种做法是我们有意为之，它也有助于代码的美丽。

参数 AB 同时还扮演着另外一种角色：除了被用来作为输入参数之外，我们还用它来作为输出参数。在这种背景下，这个设计解决了另外的一个问题。通过让例程重用原来程序中分配好的内存空间，代码的开发者就可以确保该例程在原来程序内存充足的前提下一直运行下去。如果我们把它改写成还需要在例程中进行额外的内存分配，那么在系统无法分配到更多的内存的时候，该例程就无法运行下去。当我们需要解一个很大的方程组时，此时为了保证例程可以运行，我们需要给它分配很大的一块内存，这可能会带来大问题。该示例代码对此类问题免疫的原因是：由于它的编写方式，使得它可以在原来程序中具有足够的内存空间来存储例程参数的前提下返回正确的解。这样做很漂亮。

在开始讨论其他参数前，我希望对这个问题进行更深入的讨论。在我的有生之年，我见过许多的代码。大多数情况下，开发者在编写代码时，会不知不觉地对它们加以内在限制。最常见的是，他们在代码中限制了被解决问题的大小。出现这种情况是由于人们通常都有着如下的思维过程：

1. 我需要编写一些什么；
2. 那么我就很快地把它们给编写出来，看看它们能否工作；
3. 如果它们能够工作，那么我就把它们推广到所有的问题上去。

这个思维过程促使开发者在代码中加入了限制，在大部分情况中，这将导致一些很难被发现的错误，为了解决这些问题，我们不得不花上好几年的时间。在上述过程中，开发者通常会在他们所要解决问题的大小上加以明显或者不明显的限制。例如，对大量数据空间的定义就是一个明显的限制，因为我们认为这个空间已经大到足够应付所有的问题。这样的限制无疑很糟糕，不过我们也很容易发现它。对动态内存的不当使用就是一个不明显的限制——例如，当问题被提交给编写好的代码时，程序会动态地去分配一些内存空间来辅助解决该问题。对于那些规模很大的问题来说，这可能会导致内存不足的错误，或者显著地影响到程序的性能。这种性能惩罚来自于程序对操作系统的页调度机制的依赖。如果算法是计算密集型的，并且它需要从很多不同的内存块中获取数据，那么程序就将陷入不断的页面调度，从而运行得极为缓慢。

此类问题的另一个例子表现在那些使用了数据库系统的程序中。如果程序是按照我刚刚描述的方式所编写的，那么它在对数据进行运算时，会不断地去访问数据库。程序员可能会认为，程序对数据的运算操作微不足道，耗时极少，但实际上它们的效率非常低下，因为这些操作中同时还包含了对数据库的访问。

此处我们得到的教训是什么呢？当编写漂亮的代码时，我们需要考虑它的扩展性。和大部分人的想法不同的是，可扩展性并不是来自于对代码的优化过程；实际上它来自于选

择正确的算法。对代码进行剖析（profiling）可以为低下的性能提供某些线索，但导致性能出现问题的主要原因通常都会被回溯为设计时的问题。SBGSV例程的设计就保证了它不会有此类性能问题，这也是证明它漂亮的另外一个佐证。

现在我们回过头来讨论剩下的输入参数，很明显，适用于数组 AB 的那些原则同样也适用于它们。

最后那个参数，INFO，是用来做错误处理机制的。如何将诊断信息返回给用户，这一点很有趣。有时方程组可能是无解的，这种情况就应该报告给用户。注意：此处参数 INFO 不但返回错误，同时也汇报成功，我们需要根据它提供的诊断信息来判断问题是否被成功解答。

这也是今天大部分代码中所欠缺的。如今，代码通常都被编写出来处理那些积极的用户案例；我们编写它来处理规格文档中的那些详细步骤。以示例代码为例，这也就意味着代码在方程组有解时能够正确运行。然而，事实情况总是乱糟糟的。在现实生活中，代码在运行过程中会出错，当处理到无解的方程组时，它们要么 core dump，要么就向外抛出异常。对于那些没有预期的使用方法，这也是一个通常的报错方法。

今天，很多的系统都被编写成：“做尽可能少的事情”；一旦它们被用来处理人们之前没有预期到的问题时，它们就将出问题。类似的问题也出现在没有明确说明有关要求的场合，如：如何优雅地处理错误和其他突发情况。对这种突发情况的处理是保证应用程序可靠性的关键，我们应该把它作为一项基本的功能需求来对待。

但依照规格来编写代码时，开发者应该意识到，规格文档通常都是不完全的。开发者必须对手头的问题有着深刻的理解，从而能够对规格文档进行扩充，加入新的用户案例，使代码可以以一种“智能”的方式来处理那些没有被预期的用法。我们前面给出的那个示例例程就是一个很好的例子，由于在编写前经过充分的考虑，它可以很好地处理所有这些情况。该例程要么可以完成工作，要么就向我们告诉它不能做，但它不会崩溃。这也很漂亮。

接下来，我们来看一下例程中的 *Further Details* 小节。该小节描述了在例程中内存是如何被使用的，明确告诉我们在内部操作中，这些内存被当作空白内存来使用。这也是漂亮代码的一个很好的例子，我将会在下一小节，“内在美”中来讨论这一点。

有关上面代码外在美的另外一个例子是，在 CERN 库中，很多的例程都提供了一个简单的测试程序。对于漂亮代码来说，这一点很重要。我们应该能够告诉别人，在应用程序中，这段代码是否可以按照我们所期望的方式运行。CERN 库的开发者为我们提供了许多的测试程序，向我们演示了对于特定的数据，我们该如何调用该库提供的例程。我们

也可以使用这些测试程序来验证我们是否可以从自己提供的数据中得到正确的解，从而增加我们对该库的信心。

此处这个设计的漂亮之处在于：测试不但可以告诉我们什么条件下我们可以使用这些例程，它们还给了我们一个验证的例子，让我们建立对库的信心，并知道是怎么回事——而所有这些都不需要我们去了解代码的内部细节。

内在美

现在我们开始进入代码的实现细节。

简洁和简单之美

我坚信，漂亮的代码必定是短小的代码。我所见过的冗长、复杂的代码通常都是丑陋的代码。SGBSV这个例程就是一个很好的用来体现短小代码之美的例子。在它的开始处，是一个简短的对输入参数一致性的验证，然后它就调用了另外两个例程，这在逻辑上也和数学算法保持了一致。

从第一眼看过去，我们就可以知道这段代码是做什么的：它首先调用SGBTRF例程进行LU因数分解，然后调用SGBTRS例程来对方程组求解。这段代码的可读性很好。我们没有必要对上百行代码进行研究来了解代码在做什么。代码把主要任务给划分成了两个子任务，然后这两个子任务被放到子系统中去执行。

注意，此子系统同样也遵循了主系统中对于内存使用的设计。这样做很重要，也是设计的亮点之一。

子系统中的例程在不同的“驱动”例程中可以被重用（此处SGBSV例程就被称为是驱动例程，*driver routine*）。这就产生了一个鼓励代码重用的分层系统。同样，这种做法也很漂亮。代码重用可以极大地减少在代码的开发、测试、以及维护上的投入。实际上，代码重用是提升开发人员生产效率，减轻他们负荷的最佳方法。问题在于，重用通常都很困难。在很多情况下，开发人员发现，代码实在是太复杂了，太难读懂了，这时他们就会觉得重写一遍代码都会比重用它们来的更简单。良好的设计和清晰、简洁的代码对于代码重用来说是至关重要的。

不幸的是，今天人们所编写的大多代码在这方面都不合格。如今，大部分的代码都有着一个继承结构，人们也被鼓励这样做，并期望这会让代码变得更清晰。然而，我必须承认，我花了好几个小时在几行这样的代码上，最终还是不知道它们是用来做什么的。这并不是漂亮的代码；它是一段有着错综复杂设计的不好的代码。如果在经过简单扫视之

后，我们还不能通过命名约定以及少数几行代码来知道该代码是做什么的，那么这段代码就太复杂了。

漂亮的代码应该很容易被人所理解。我憎恨阅读那些用来炫耀程序员对编程语言知识掌握能力的代码，我也不想在读过25个文件后才知道某个代码片段是做什么的。代码可以没有注释，但它的目的必须明确，每一步的操作不能存在歧义。今天我们所编写的代码的问题在于——尤其是那些使用C++编写的代码——开发人员在代码中使用了太多的继承和重载，这使得人们几乎不可能知道代码要做什么，它为什么那么做，以及它那样做是否正确。为了了解这些，我们必须先理解整个的继承架构以及所有的重载。如果我们要的操作是一个复杂的重载操作集的话，那么这样的代码对我来说不是漂亮的代码。

节俭之美

我界定漂亮代码的下一标准是：我们可以对“代码在计算机上是怎么运行的”有着很多的见解。我在此试图说明的是，漂亮的代码永远不会忘记它们是要在计算机上运行的，而计算机有着种种限制。在本章前些地方，我说过，计算机的运算速度有限；有时它们对浮点的运算表现更好，有时则更适合做整数运算；还有，计算机上的内存也是有限的。漂亮的代码必须同时也考虑到这些现实中的限制。很多时候，人们在编写代码时总是假设内存是无限的，计算机的运行速度也是无限的，等等。这并不是漂亮的代码；这只是傲慢的代码。漂亮的代码对于诸如内存使用这类事情是很节俭的，在可能的情况下它均选择重用内存。

例如，我们来看SGBTRF例程中的LU因数分解，该例程已经是第二层的子例程了。为了节省空间，我在此处删除了开头处的注释（以及其他我没有直接讨论到的注释）（我们可以从<http://www.netlib.org/lapack/explore-html/sgbtrf.f.html>中查阅完整的例程代码）：

```
SUBROUTINE SGBTRF( M, N, KL, KU, AB, LDAB, IPIV, INFO )
*
* -- LAPACK 例程 (版本 2.0) --
* 田纳西大学, 加利福尼亚大学伯克利分校, NAG有限公司,
* 科朗研究院, 阿贡国家实验室以及莱斯大学
* 1992 年 2 月 29 日
```

初始的注释，以及参数介绍

* 测试输入参数。

```

*
INFO = 0
IF( M.LT.0 ) THEN
    INFO = -INFO
    CALL XERBLA( 'SGBTRF', -INFO )
    RETURN
END IF
*
* 如果可能的话，则快速返回
*
IF( M.EQ.0 .OR. N.EQ.0 )
$ RETURN
*
* 确定块的大小
*
NB = ILAENV( 1, 'SGBTRF', ' ', M, N, KL, KU )
*
* 块的大小不能超过局部数组 WORK13 和 WORK31 所设定的限制。
*
NB = MIN( NB, NBMAX )
*
IF( NB.LE.1 .OR. NB.GT.KL ) THEN
*
* 使用未分块代码
*
CALL SGBTTF2( M, N, KL, KU, AB, LDAB, IPIV, INFO )
ELSE
*
* 使用分块代码
*
* 将数组 WORK13 的主对角清零
*
DO 20 J = 1, NB
    DO 10 I = 1, J - 1
        WORK13( I, J ) = ZERO
10    CONTINUE
20    CONTINUE
*
* 将数组 WORK31 的次对角清零
*
DO 40 J = 1, NB
    DO 30 I = J+1, NB
        WORK31( I, J ) = ZERO
30    CONTINUE
40    CONTINUE

```

采用列主元素高斯消去法

将第 KU+2 列到第 KV 列的填充元素设置为零

```
DO 60 J = KU + 2, MIN( KV, N )
```

```
DO 50 I = KV - J + 2, KL
```

```
AB( I, J ) = ZERO
```

```
50 CONTINUE
```

```
60 CONTINUE
```

JU 是因式分解当前阶段正在处理的列索引。

```
JU = 1
```

```
DO 180 J = 1, MIN( M, N ), NB
```

```
JB = MIN( NB, MIN( M, N )-J+1 )
```

矩阵的有效部分被划分成块

```
A11 A12 A13
```

```
A21 A22 A23
```

```
A31 A32 A33
```

其中, A11, A21 和 A31 表示将要被分解的当前块, 列数为 JB。

在块中的行数分别为 JB, I2 和 I3, 列数分别为 JB, J2 和 J3。

A13 的超对角元素和 A31 的次对角元素位于带外。

```
I2 = MIN( KL-JB, M-J-JB+1 )
```

```
I3 = MIN( JB, M-J-KL+1 )
```

J2 和 J3 在 JU 被更新之后进行计算。

分解列数为 JB 的当前块。

```
DO 80 JJ = J, J + JB - 1
```

将第 JJ+KV 中的填充元素设置为零。

```
IF( JJ+KV.LE.N ) THEN
```

```
DO 70 I = 1, KL
```

```
AB( I, JJ+KV ) = ZERO
```

```
70 CONTINUE
```

```
END IF
```

查找列主元素并检测奇异性。KM 是当前列中次对角元素的数量。

```
KM = MIN( KL, M-JJ )
```

```
JP = ISAMAX( KM+1, AB( KV+1, JJ ), 1 )
```

```
IPIV( JJ ) = JP + JJ - J
```

```
IF( AB( KV+JP, JJ ).NE.ZERO ) THEN
```

```
JU = MAX( JU, MIN( JJ+KU+JP-1, N ) )
```

```
IF( JP.NE.1 ) THEN
```

```

*
* 对第 J 列到第 J+JB-1 列进行交换操作。
*
* IF( JP+JJ-1.LT.J+KL ) THEN
*   CALL SSWAP( JB, AB( KV+1+JJ-J, J ), LDAB-1,
*               AB( KV+JP+JJ-J, J ), LDAB-1 )
* ELSE
*   交换操作将影响 A31 的第 J 列到第 JJ-1 列。
*
* CALL SSWAP( JJ-J, AB( KV+1+JJ-J, J ), LDAB-1,
*               WORK31( JP+JJ-J-KL, 1 ), LDWORK )
* CALL SSWAP( J+JB-JJ, AB( KV+1, JJ ), LDAB-1,
*               AB( KV+JP, JJ ), LDAB-1 )
* END IF
* END IF
*
* 计算乘法值
*
* CALL SSCAL( KM, ONE / AB( KV+1, JJ ), AB( KV+2, JJ ),
*               1 )
*
* 继续直接求解
*
170      CONTINUE
180      CONTINUE
END IF
*
RETURN
*
* End of SGBTRF
*
END

```

同样，在该例程的开始处我们对参数进行了验证，然后就是对问题进行求解。在此处我们增加了一个用于优化的检测，它会去判断问题的大小，以决定是否可以用“缓存”数组 WORK13 和 WORK31 来辅助求解，或是是否应该将程序转入到下一层子系统去进行更复杂的运算。这是一个很好的例子，它涉及了现实中计算机的内在限制。我们可以根据用来求解的计算机中的内存大小来调整工作数组的大小，对于那些规模足够小的问题来说，它还可以避免因为可能的页面调度带来的性能损耗。对于规模在一定大小之上的问题，这种性能上的损耗是不可避免的。

前面的问题解答提供了一个算法的逐步表示。阅读这样的代码就像阅读一本书一样，因为我们很容易就顺着读下去了。算法中和其他算法相同的部分来自于重用，那些会导致代码变得复杂的部分被放到了子例程中去处理。最终我们得到的代码是一个清晰的、可理解的操作流程。

流程中的每一步都对应着一个数学表达式。在每一步中，代码都描述了它所期望下一层子系统要做的事情，然后调用该子系统。主例程，也就是驱动例程，会调用那些低层次的例程，而每个这样的低层次例程，会调用更低层次的例程，……图 15-1 描述的就是这个流程。

这是一个很好的用来演示“分而治之”原则是如何被应用到代码设计上的例子。每一次我们都把事情挪到下面的例程去做，产生一个小一点的待解决的问题，从而我们就可以关注如何做好设计，把代码写得更短小，更贴近我们要做的事情。如果待解决的问题可以在计算机现有内存下完成，那么算法就会直接对它求解，就如我在前面所描述的那样。如果不能，那么我们就进入到下一级子例程中去，直到最终可以直接求解为止。

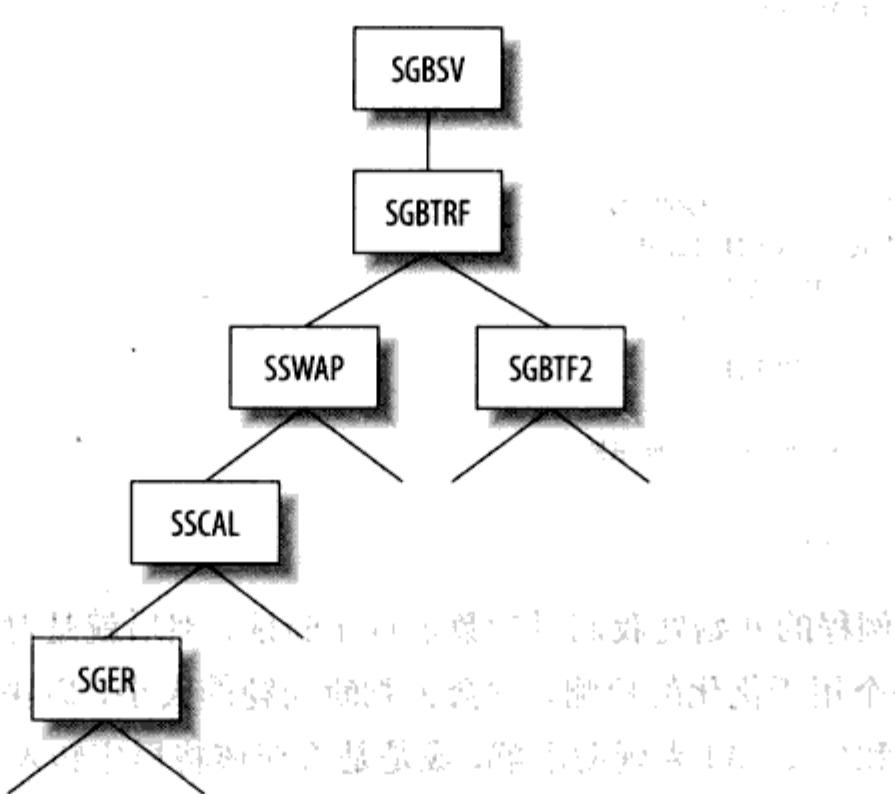


图 15-1：逻辑划分任务至子例程

作为最终结果，对于那些计算密集型的例程，我们可以用汇编语言来编写，并针对硬件平台进行优化。该设计的另外一个好处在于，由于每个子例程都不依赖于其他（同级的）子例程，我们可以同时多人调用该代码。

结论

总之，我认为，漂亮的代码必须是简短、明确、（对计算机资源的使用）节俭以及考虑到现实情况的代码。然而，我认为对美——不论是代码还是艺术——的界定在于它是否可以经得起时间的考验。在过去的许多年间，大量的代码被编写出来，但只有少量的代码现在仍然在被使用（即便是在它们被编写多年以后）。CERN库中的代码在被编写了30年后，至今仍然被人使用，这也验证了它就是真正的漂亮代码。

Linux内核驱动模型： 协作的好处

Greg Kroah-Hartman

在 Linux 内核中，驱动模型是核心的一个部分。它负责与系统中的所有设备进行交互。通过使用一个统一的、可伸缩的模型，Linux 的内核驱动模型试图建立一棵全系统的设备树，其中包含操作系统所管理的所有不同类型的设备。随着时间的推移，用以实现这棵树的核心数据结构及代码也从一个简单的、只能处理有限几种设备的系统发展成一个高度可伸缩的、可以控制与现实世界交互的各种不同设备的系统。

随着 Linux 内核的演化，Linux 需要处理越来越多不同类型的设备（注 1），为此系统内核的核心模块也不得不做相应的改进以提供更便捷、更易于管理的方法来处理不同的设备类型。

几乎所有的设备都由两个不同的部分构成：物理部分定义操作系统如何与设备对话（通过 PCI 总线，SCSI 总线，ISA 总线，USB 总线等），虚拟部分定义操作系统如何将设备提供给用户从而使用户能通过适当的方式操纵它们（键盘、鼠标、视频、音频等）。在 2.4 版本的内核中，每一个设备的物理部分都由一段总线特定的代码来控制。总线代码负责各种不同类型的任务，而每一种具体总线的代码之间则没有交互。

注 1：Linux 现在所支持的设备和处理器的类型数目已超过了计算机历史上出现过的任何其他操作系统。

在 2001 年，Pat Mochel 致力于解决 Linux 内核中的电源管理问题。他认识到为了能正常地启动和关闭各种设备，内核必需了解各种不同设备的连接关系。比如，为了能正常保存设备中的数据，一个 USB 磁盘应该在 USB 控制器所在的 PCI 控制卡之前关闭。为解决这一问题，内核需要了解系统中全部设备的树形结构，了解哪些设备控制另外哪些设备，以及所有的设备的连接次序。

大约在同一时间，我遇到了另一个设备相关问题：Linux 无法通过一种持久的方式来处理设备。我想让我的两台 USB 打印机的名字永远不变，不管它们中哪一台先被启动，或者先被 Linux 内核发现。

其他一些操作系统，有的通过在内核中放置一个处理设备名的小型数据库来解决这一问题的，也有的尝试通过文件系统中可以直接访问设备的 *devfs*（注 2）类型来导出设备的所有特性信息的。对 Linux 来说，在内核中放置一个数据库是一个不可接受的方案。同样，Linux 的 *devfs* 文件系统实现中存在一些众所周知的且难以消除的竞争条件，导致几乎所有的 Linux 发布的版本中都不能依赖它。*devfs* 方案还强迫用户使用一个特定的命名策略。尽管有些人认为这是个优点，但它毕竟违背了 Linux 公开发布的设备命名标准，而且不允许任何人采用不同的命令策略，即使他们非常想这样。

Pat 和我认识到这两个问题可以通过 Linux 内核中统一的驱动和设备模型来解决。这种统一的模型并不是什么新想法，过去其他一些操作系统已经体现了这样一种统一模型。现在是该 Linux 也采用同样方法的时候了。这种模型允许创建包含所有设备的树，从而使内核外用户空间的程序可以以任何一种用户想要的方式来持久化地处理所有设备的名字。

本章将描述在 Linux 内核中实现这项任务所需的数据结构及其支持函数的演变，以及这种演变如何引发了任何人在开发过程之初都想像不到的变化。

简单的开始

我们将从一个简单的结构体：`struct device` 开始，这个结构体被设计为内核中所有设备的“基”类如下所示。

```
struct device {  
    struct list_head node;           /* 兄弟链表中的结点 */  
    struct list_head children;       /* 子设备链表 */  
    struct device *parent;          /* 父设备 */  
    char name[DEVICE_NAME_SIZE];    /* 描述性 ascii 字符串 */
```

注 2： *devfs* 是操作系统把所有可用设备显示给用户的一种方式。它通过显示所有不同设备的名字来实现，有时也把设备之间的关系做一定程度的显示。

```

char bus_id[BUS_ID_SIZE]; /* 在父总线上的位置 */
spinlock_t lock;           /* 锁对象，用于确保不同层次不会同时访问设备 */
atomic_t refcount;         /* refcount 用于确保设备被持久的时间长短是正确的 */
struct driver_dir_entry * dir;
struct device_driver *driver; /* 哪一个驱动分配了这台设备 */
void *driver_data;          /* 设备的私有数据 */
void *platform_data;        /* 特定平台的数据(如ACPI, 设备相关的BIOS数据) */
u32 current_state;          /* 当前的操作状态，在ACPI中为D0-D3, D0 开启全部
                           功能, D3 关闭 */
unsigned char *saved_state; /* 保存的设备状态 */
};


```

每次这个结构体被创建并在核心驱动模块中注册，就会创建一个新的虚拟文件系统入口，这个入口给出了设备以及它所包含的不同属性。这使得系统中所有的设备按照它们被连接的顺序显示到用户空间中。这个虚拟文件系统现在被称为“sysfs”，且可以在一台Linux机器上的`/sys/devices`目录中看到它。以下是该结构的一个例子，其中显示了几种PCI和USB设备：

```

$ tree -d /sys/devices/
/sys/devices/pci0000:00/
|-- 0000:00:00.0
|-- 0000:00:02.0
|-- 0000:00:07.0
|-- 0000:00:1b.0
|   |-- card0
|   |   |-- adsp
|   |   |-- audio
|   |   |-- controlC0
|   |   |-- dsp
|   |   |-- mixer
|   |   |-- pcmC0D0c
|   |   |-- pcmC0D0p
|   |   |-- pcmC0D1p
|   |   '-- subsystem -> ../../../../../../class/sound
|   '-- driver -> ../../../../../../bus/pci/drivers/HDA Intel
-- 0000:00:1c.0
|   |-- 0000:00:1c.0:pcie00
|   |-- 0000:00:1c.0:pcie02
|   |-- 0000:00:1c.0:pcie03
|   |-- 0000:01:00.0
|   |   '-- driver -> ../../../../../../bus/pci/drivers/sky2
|   '-- driver -> ../../../../../../bus/pci/drivers/pcieport-driver
-- 0000:00:1d.0
|   '-- driver -> ../../../../../../bus/pci/drivers/uhci_hcd
`-- usb2
    |-- 2-0:1.0
    |   '-- driver -> ../../../../../../bus/usb/drivers/hub
    |   '-- subsystem -> ../../../../../../bus/usb
    |   '-- usbdev2.1_ep81
    '-- driver -> ../../../../../../bus/usb/drivers/usb
    '-- usbdev2.1_ep00
`-- 0000:00:1d.2

```

```

|-- driver -> ../../bus/pci/drivers/uhci_hcd
|-- usb4
|-- 4-0:1.0
|-- driver -> ../../bus/usb/drivers/hub
`-- usbdev4.1_ep81
-- 4-1
|-- 4-1:1.0
|-- driver -> ../../bus/usb/drivers/usbhid
`-- usbdev4.2_ep81
|-- driver -> ../../bus/usb/drivers/usb
|-- power
`-- usbdev4.2_ep00
-- 4-2
|-- 4-2.1
|-- 4-2.1:1.0
|-- driver -> ../../bus/usb/drivers/usbhid
`-- usbdev4.4_ep81
|-- 4-2.1:1.1
|-- driver -> ../../bus/usb/drivers/usbhid
`-- usbdev4.4_ep82
|-- driver -> ../../bus/usb/drivers/usb
`-- usbdev4.4_ep00
|-- 4-2.2
|-- 4-2.2:1.0
|-- driver -> ../../bus/usb/drivers/usblp
`-- usbdev4.5_ep01
`-- usbdev4.5_ep81
|-- driver -> ../../bus/usb/drivers/usb
`-- usbdev4.5_ep00
|-- 4-2:1.0
|-- driver -> ../../bus/usb/drivers/hub
`-- usbdev4.3_ep81
|-- driver -> ../../bus/usb/drivers/usb
`-- usbdev4.3_ep00
|-- driver -> ../../bus/usb/drivers/usb
`-- usbdev4.1_ep00
...

```

为使用这个结构体，你需要把它嵌入到另一个结构体中，从而在某种意义上，使新的结构体“继承”自上面的“基”结构体。

```

struct usb_interface {
    struct usb_interface_descriptor *altsetting;

    int act_altsetting;          /* 活动备用设置 */
    int num_altsetting;         /* 备用设置数目 */
    int max_altsetting;         /* 总内存分配 */
    struct usb_driver *driver;   /* 驱动 */
    struct device dev;           /* 特定接口设备信息 */
};

```

核心驱动模块通过在不同的代码之间传递指向device结构的指针来运做，它操作结构体中的基本字段，这些字段在所有类型的设备中都存在。当指针被传递给实现某种功能

的总线特定代码 (bus-specific code) 时, 它就需要被转换成包含它的实际结构体的类型。为了实现这种转换, 总线特定代码基于指针在内存中的位置把它转换回原来的结构体类型。这是通过以下这个宏来实现的:

```
#define container_of(ptr, type, member) ({  
    const typeof( ((type *)0)->member ) *__mptr = (ptr);  
    (type *) ( (char *)__mptr - offsetof(type,member) );})
```

对于那些不熟悉C语言指针运算的人, 这个宏需要解释一下。举个例子: 通过以下代码, 前面的 struct usb_interface 可以将一个指向 struct device 的成员指针转换回原来的指针:

```
int probe(struct device *d) {  
    struct usb_interface *intf;  
    ...  
    intf = container_of(d, struct usb_interface, dev);  
    ...  
}
```

其中 d 是一个指向 struct device 的指针。

将上面用到的 container_of 宏展开, 将会产生如下代码:

```
intf = ({  
    const typeof( ((struct usb_interface *)0)->dev ) *__mptr = d;  
    (struct usb_interface *) ( (char *)__mptr - offsetof(struct  
    usb_interface, dev));  
});
```

要理解这段代码, 就要记着 dev 是 struct usb_interface 的成员。宏的第一行定义了一个指针, 这个指针指向传递给代码的 struct device。第二行在内存中查找我们想要访问的 struct usb_interface 的实际位置。

因此, 如果知道 dev 结构的类型, 宏就可以被简化成:

```
intf = ({  
    const struct device *__mptr = d;  
    (struct usb_interface *) ( (char *)__mptr - offsetof(struct usb_interface, dev));  
});
```

基于前面刚刚给出的 struct usb_interface 的定义, 在 32 位处理器上, dev 变量很可能被放置在结构体的第 16 个字节的位置 (译注 1)。这个是由编译器通过 offsetof 宏自动计算出来的。把这个信息代入到宏中, 就产生如下代码:

译注 1: 这里显然应该是 20, 而且用 32 位的 VC 编译, 结果就是 20。

```
intf = ({  
    const struct device *__mptr = d;  
    (struct usb_interface *)((char *)__mptr - 16));  
});
```

container_of宏现在被简化成了简单的指针运算，从原来的指针上减去16就得到了我们想要的指向 struct usb_interface 的指针。编译器可以在运行时快速的完成这一计算。

通过这种非常简单的方法，Linux 内核允许常规的 C 结构体被“继承”并通过强有力的方法来操纵。是的，非常强大，只要你清楚地知道你正在做什么。

你可能注意到这里并没有运行时的类型检查来确保原来作为 struct device 类型而被传递的指针变量实际上指向 struct usb_interface 类型。按照惯例，大部分进行这种指针运算的系统都会在基结构体中设置一个字段来定义被操纵的指针的类型，以便于捕捉马虎的程序错误。这也使代码可被写成动态的确定指针类型并针对具体的类型执行不同的动作。

Linux 内核开发者所做的决定是既不做这种检查，也不作类型定义。这种类型检查可以在开发的早期阶段捕捉简单的程序错误，但也使得开发者可以使用一些复杂的技巧，而这可能在以后引发更加棘手而且难以发现的问题。

没有运行时检查，迫使操作这些指针的开发者必需确切地知道它们所操纵的以及在系统中传递的指针的类型。当然，有些时候，开发者确实想拥有一些方法来确定操纵的 device 结构的实际类型，但是当问题被正确的 debug 了之后，这种感觉最终就消失了。

这种对类型检查的缺失是否能好到可被称为“美丽代码”呢？在使用它工作了 5 年多之后，我想，是的。它使得简单的黑客行为难于在内核中实现，而且要求每一个开发者必需做到代码的逻辑清晰永远不要依赖于对类型的检查，尽管类型检查或许可以阻止 bug 在以后发生。

需要注意的是，这里只有相对少数的开发者——即那些开发通用总线子系统的人——才需要接触到内核的这些部分，而且他们可望获得相当多的专业知识，这也是为什么这里没有坚持类型检查的原因。

通过这种“继承”基本 struct device 的方法，在 2.5 版内核的开发过程中，所有不同的驱动子系统都被统一处理了。现在他们共享公共的核心代码，从而使内核能把所有设备间的连接关系显示给用户。这也促进了一些有用工具的出现，比如 udev，通过用户空间的一个小程序来实现永久性设备命名；再比如一些电源管理工具，可以遍历整个设备树然后以适当的次序关闭设备。

进一步简化

在对驱动模块进行修改时，内核的引用计数系统也得到了改进。当开发者开始改写最初的核心驱动模块时，另外两个内核开发者：AI Viro，正致力于修改虚拟文件层的几个与对象引用计数有关的问题。

在用C语言编写的多线程程序中，使用结构体的主要问题是很难确定何时释放结构体使用的内存是安全的。Linux内核是一个大量使用多线程的程序，它必须恰当的处理不友好的用户，以及大量同时运行的处理器，因此，有必要对任何一个可被多个线程访问的结构进行引用计数。

`struct device`就是这样一个被引用计数的结构体。它有一个字段用于确定何时释放它是安全的。

```
atomic_t refcount;
```

当结构体被初始化时，这个字段被置为1。每当有代码想要访问这个结构体，它必需首先通过调用`get_device`函数把引用计数加1。`get_device`函数检查引用计数的有效性并将结构体的引用计数加1。

```
static inline void get_device(struct device * dev)
{
    atomic_inc(&dev->refcount);
}
```

类似的，当一个线程结束了对结构体的使用，它通过调用`put_device`将引用计数减1。`put_device`函数稍微复杂一些：

```
void put_device(struct device * dev)
{
    if (!atomic_dec_and_lock(&dev->refcount, &device_lock))
        return;
    ...
    /* 通知驱动在它之后进行清理。
     * 注意我们可能并没有分配设备，因此这是驱动释放设备的机会……
     */
    if (dev->driver && dev->driver->remove)
        dev->driver->remove(dev, REMOVE_FREE_RESOURCES);
}
```

这个函数将引用计数减1，然后如果它是这个对象的最后一个用户，就通知对象清理它自己并调用先前设置的函数来将它从系统中释放。

Al Viro 喜欢统一的 device 结构，显示所有不同的设备和这些设备连接关系的虚拟文件系统，以及自动引用计数。惟一的问题是它的虚拟文件系统核心并不能用于“设备”，也没有“驱动”可以附加到这些对象上。于是，它决定对代码做一些重构，并使它更简单。

Al 使 Pat 相信创建一个叫做 kobject 的结构是必要的。这个结构拥有 struct device 的基本属性，但它更小，而且没有“驱动”与“设备”的关系。它含有以下字段：

```
struct kobject {
    char                name[KOBJ_NAME_LEN];
    atomic_t            refcount;
    struct list_head   entry;
    struct kobject     *parent;
    struct subsystem   *subsys;
    struct dentry      *dentry;
};
```

这个结构是一种空对象 (empty object)。它仅拥有非常基础的功能：被引用计数，被插入到对象层次结构中。struct device 结构体现在可以包含这个更小的 struct kobject “基” 结构体来继承它的全部功能。

```
struct device {
    struct list_head g_list;          /* 深度优先顺序链表中的结点 */
    struct list_head node;            /* 兄弟链表中的结点 */
    struct list_head bus_list;        /* 总线链表中的结点 */
    struct list_head driver_list;
    struct list_head children;
    struct list_head intf_list;
    struct device *parent;

    struct kobject kobj;
    char    bus_id[BUS_ID_SIZE];    /* 在父总线中的位置 */
    ...
}
```

container_of 宏用来将 kobject 核心结构转换回主 struct device。

```
#define to_dev(obj) container_of(obj, struct device, kobj)
```

在这一过程中，许多其他的开发者也协助增强了 Linux 内核的健壮性，使它的可伸缩性更好，从而可以支持越来越多的跑在同一系统映像上的处理器数目（注 3）。因为这个原因，其他许多开发者也在他们的结构中加入了引用计数，以正确处理内存使用。每一个开发者都不得不复制结构体的初始化、计数递增、计数递减和清理功能。因此开发者们

注 3：运行于单个 Linux 系统映像上的处理器数目，其当前的最高纪录是 4096。所以，可伸缩性方面的工作是成功的。

决定把这一简单的功能从 kobject 中抽取出来并放进一个单独的结构中。于是就诞生了 struct kref 结构体：

```
struct kref {  
    atomic_t refcount;  
};
```

跟 struct kref 相关联的只有 3 个简单的函数：初始化引用计数的 kref_init，增加引用计数的 kref_get，以及减少引用计数的 kref_put。前两个函数非常简单。最后一个有点意思：

```
int kref_put(struct kref *kref, void (*release)(struct kref *kref))  
{  
    WARN_ON(release == NULL);  
    WARN_ON(release == (void (*) (struct kref *)) kfree);  
  
    if (atomic_dec_and_test(&kref->refcount)) {  
        release(kref);  
        return 1;  
    }  
    return 0;  
}
```

kref_put 函数有两个参数：一个指向将被减少引用计数的 struct kref 的指针和一个函数指针，如果这是对象上的最后一个引用，函数指针所指的函数将被调用。

函数的前两行是在 struct kref 被加入到内核中之后才被添加上去的，因为一些程序员试图通过传递一个空函数指针来绕过引用计数，或者在发现内核会抱怨之后，改为传递指向 kfree 函数的指针。（令人沮丧的是，现在这两个检查也不够了。一些程序员开始传递什么也不做的空 release 函数，仿佛他们要将引用计数完全忽略。不幸的是，C 语言中没有一种简单的方法可以确定一个函数指针指向的函数是否真正做事，否则，这种检查也会被添加到 kref_put 函数中。）

通过了这两个检查之后，引用计数就被自动递减，如果这是最后一个引用，release 函数就被调用并返回 1。如果这不是对象的最后一个引用，则返回 0。这一返回值仅被用于指明调用者是否是最后一个对象持有者，不能用来判断对象是否还在内存中（它不能保证对象仍然存在，因为其他人可能闯进来并在调用返回后释放对象）。

有了 struct kref，struct kobject 结构就被改写为使用 kref：

```
struct kobject {  
    char name[KOBJ_NAME_LEN];  
    struct kref kref;  
    ...  
};
```

将所有这些不同的结构嵌入到其他的结构当中，结果是先前描述的 `struct usb_interface` 现在包含一个 `struct device`，而 `struct device` 包含一个 `struct kobject` 结构，`struct kobject` 又包含 `struct kref`。谁说用 C 语言进行面向对象编程很难……

扩展到上千台设备

随着 Linux 逐渐在各种各样的平台上运行——从蜂窝式无线电话、无线电控制的直升飞机、台式机、服务器到世界上 73% 的大型超级计算机——扩展驱动模型就变得非常重要并成为了我们头脑中的头等大事。随着开发不断向前推进，我们很高兴地看到持有设备的基本数据结构，如 `struct kobject` 和 `struct device` 的尺寸相对缩小了。对于大多数系统而言，连接到系统中的设备数目直接跟系统的大小成正比。跟小的嵌入式系统连接并存在于其设备树中的设备数目不超过 10。而大的“企业级”系统中则会有多得多的设备连接上来，但同时这些系统也拥有更多的内存可被使用，因此相对于全部内核的内存占用量来说，更大的设备数目所需的内存仅仅是很小的一部分。

遗憾的是，这个听起来让人振奋的可伸缩模型在面对一种“企业级”系统：s390 大型计算机时，被发现是完全错误的。这种计算机可以在一个虚拟分区（virtual partition）中运行 Linux（在一台机器上最多可以运行 1024 个系统实例），而且有大量不同类型的存储设备跟它连接。总体上看，系统拥有大量的内存，但每一个虚拟分区只能拥有那些内存中很小的一个片段。每一个虚拟分区都需要看到全部的存储设备（通常情况是 2 万台），但被分配的 RAM 却只有几百兆字节。

在这种系统中，你很快就会发现设备树占据了内存中相当大的比例，而且永远不会释放给用户进程。该让驱动模型降低一下食欲了，IBM 公司的一些非常聪明的开发者开始着手解决这个问题。

开发者们刚看到结果时都感到很惊讶。`struct device` 结构体的大小只有 160 字节（在 32 位机器上）。在一个拥有 2 万台设备的系统中，那也只是 3MB 到 4MB 的内存开销，完全可以掌控。而真正吞掉大量内存的是前面提到的基于 RAM 的文件系统 `sysfs`，它将所有这些设备显示给用户空间。`Sysfs` 为每一台设备创建一个 `struct inode` 和一个 `struct dentry`。这些都是重量级的结构体，`struct inode` 大约有 256 字节，`struct dentry` 结构大约有 140 字节。（注 4）

对每一个 `struct device`，至少有一个 `struct dentry` 和一个 `struct inode` 被创建。通常，这些文件系统中的结构被拷贝了许多份，系统中某台设备的每个虚拟文件都对应

注 4：这两个结构后来都被缩减过，因此在当前的内核版本中已经小一些了。

一份拷贝。举例来说，针对一个块设备将创建 10 个不同的虚拟文件，这意味着一个 160 字节的 `device` 结构最终将占去 4KB 的空间。在一个拥有 2 万台设备的系统中，将有 80MB 的空间被浪费在虚拟文件系统上。这些内存是被内核消耗的，无法被任何用户程序使用，即使这些程序从来没想要查看 `sysfs` 中保存的信息。

针对这一问题的解决方案是重写 `sysfs` 的代码，把 `struct inode` 和 `struct dentry` 放进内核的缓存中，在文件系统被访问时随需创建它们。这一方案实际上就是在用户遍历设备树时随需动态地创建目录和文件，而不是在设备刚被创建时就预先分配所有的空间。由于这些结构存在于内核的主缓存中，如果有用户程序或者内核的其他部分引发系统内存资源紧张，缓存就被释放出来，于是内存就被还给此时需要它的程序。所有这些都是通过修改 `sysfs` 的后端代码实现的，而不是 `struct device` 结构体。

小对象的松散结合

Linux 的驱动模型显示了 C 语言是如何通过生成大量的小对象来创建高度面向对象的代码，每一个这样的小对象都能做好一件事。这些不带任何运行时类型信息的对象可被嵌入到其他对象中，构成一棵强大而灵活的对象树。而从这些对象的实际使用情况来看，它们的内存占用被降到了最低限度，这就保证了 Linux 内核具有足够的灵活性，同一套内核代码可运行于各种不同的计算平台，从微小的嵌入式系统到世界上最大的超级计算机。

这种模型的开发也反映了 Linux 内核开发中的两个有趣而又强大的方面：

第一，开发过程是迭代式的。当内核的需求发生变化时，运行在内核上的系统也发生变化，开发者找到了对模型的不同部分进行抽象，使它在需要高效的地方更加高效的方法。这是对系统的进化需求的响应，一个系统要想在这种环境中继续存在就必须不断进化。

第二，设备处理的发展过程表明此过程具有高度的协作性。不同的开发者各自独立地提出一些想法来改善和扩展内核的不同方面。通过阅读源代码，其他人也可以确切地看到这些开发者的目标，正如开发者自己描述的那样。于是他们也能帮助修改代码，而且所采用的方法可能是原来的开发者从来没想到的。最终的结果就是找到一种公共解决方案，一种任何开发者单凭个人能力都不会想到的方案，来达成许多不同开发者的目标。

这两个开发过程的特征使 Linux 演化成了今天最灵活最强大的操作系统。而且只要这样的开发过程继续，它们就能保障 Linux 可以永远保持这种灵活与强大。

额外的间接层

Diomidis Spinellis

“所有计算机科学领域中的问题都可以通过一个额外的间接层来解决”，Butler Lampson 的这句名言被人无数次引用（注：Butler Lampson 在 1972 年提出了现代个人计算机的设想）。同样，我也不止一次地回想起它：当我想联系某人时却不得不先和他（她）的秘书接触；当我第一次飞行到法兰克福而实际上我的目的地却是上海或者班加罗尔时；还有，当我面对一个复杂的系统的源代码时……

在开始这个特殊的旅程之前，让我们来设想如下场景：操作系统需要支持不同的独立文件系统格式。操作系统可以使用它本地文件系统的数据，也可以使用光盘或 U 盘中的数据；而所有的这些存储设备可能都使用着不同的文件系统来管理它们里面的数据，如：Windows 或者 Linux 的本地文件系统 NTFS 或 ext3fs；光盘上面的 ISO-9660；以及大部分 U 盘上所使用的老旧的 FAT-32 文件系统。每种文件系统都使用了不同的数据结构来管理它其中的空余空间，存储的文件元数据，以及将文件组织成为目录等。因而，为了支持不同的文件系统，我们就不得不提供不同的代码来对文件进行操作（如：打开/读/写/定位/关闭/删除等操作）。

从直接代码操作到通过函数指针操作

在我成长的年代，不同的计算机系统中通常都有着不同的、互不兼容的文件系统，这也迫使 I 通过串口来进行计算机间的数据传输。因此，我对于在 PC 上可以读取数码相机存储卡内的数据这个现象一直感到吃惊。我们先来考虑一下，在操作系统中对不同的文件系统进行访问的代码是如何组织的。有个简单的方法就是在每个操作中使用一个 switch 语句。为了举例说明这个，我们假想了一个在 FreeBSD 操作系统中用于读文件的系统调用 read 的实现。它在内核中的接口看上去如下：

```
int VOP_READ(
    struct vnode *vp,          /* 将要读取的文件 */
    struct uio *uio,           /* 缓冲区 */
    int ioflag,                /* I/O 特定的标志 */
    struct ucred *cred)        /* 用户的权限 */

{
    /* 模拟实现 */
    switch (vp->filesystem) {
        case FS_NTFS:           /* NTFS 特定的代码 */
        case FS_ISO9660:         /* ISO9660 特定的代码 */
        case FS_FAT32:           /* FAT32 特定的代码 */
        /* [...] */
    }
}
```

上面的方法会把不同的文件系统和对文件的操作代码紧紧绑定在一起，限制了内核的模块化特性。更糟的是，当在系统中增加一个新文件系统的支持时，我们将不得不修改所有（有关文件操作的）系统调用的实现，并重新编译整个内核。更进一步，如果我们想要在每个文件系统操作的动作中增加额外的操作（如：映射一个远程用户证书）时，同样也需要我们在每个操作中进行同样的修改，这也容易导致错误的产生。

可能你也许猜想到了，我们手头的任务需要我们在代码中加入一个额外的间接层。我们还是以我所佩服的、在工程方面仅为成熟的 FreeBSD 操作系统的源代码为例，看看它是如何解决这个问题的。在 FreeBSD 中，它以函数的方式为每个文件系统定义了它所支持的操作，然后通过用了一个名为 vop_vector 的数据结构来初始化所有这些操作函数的函数指针。下面就是 vop_vector 这个结构中的某些字段：

```
struct vop_vector {
    struct vop_vector *vop_default;
    int (*vop_open)(struct vop_open_args *);
    int (*vop_access)(struct vop_access_args *);
```

接下来就是如何用该数据结构来初始化 ISO-9660 这个文件系统：

```
struct vop_vector cd9660_vnodeops = {
```

```

    .vop_default = &default_vnodeops,
    .vop_open = cd9660_open,
    .vop_access = cd9660_access,
    .vop_bmap = cd9660_bmap,
    .vop_cachedlookup = cd9660_lookup,
    .vop_getattr = cd9660_getattr,
    .vop_inactive = cd9660_inactive,
    .vop_ioctl = cd9660_ioctl,
    .vop_lookup = vfs_cache_lookup,
    .vop_pathconf = cd9660_pathconf,
    .vop_read = cd9660_read,
    .vop_readdir = cd9660_readdir,
    .vop_readlink = cd9660_readlink,
    .vop_reclaim = cd9660_reclaim,
    .vop_setattr = cd9660_setattr,
    .vop_strategy = cd9660_strategy,
};


```

(上面代码中`.field = value`这种语法是C99所提供的一个很好的特性，它使得我们能够以任意的顺序来初始化结构中的字段，从而提高了程序的可读性。)请注意，虽然一个完整的`vop_vector`中总共有52个字段，但在上面的代码中，我们只对其中的16个进行了定义。举例而言，上面代码中若没有对`vop_write`这个字段进行定义，它就将获得一个NULL值，这是因为，对于光盘(CD-ROM，非CD-R)上的ISO-9660文件系统而言，它并不支持对文件的写操作。通过对不同的文件系统初始化这样的一个数据结构(如下面图17-1中所示)，我们就可以很容易地把该数据结构和文件句柄所指向的数据绑定起来。从而，在FreeBSD内核中，我们就可以把文件系统无关的读操作简单实现成(见图17-1)：

```

struct vop_vector *vop;
rc = vop->vop_read(a);


```

于是，当我们从包含有ISO-9660文件系统的光盘中读取数据时，通过函数指针，上面的函数调用最终调用的将会是`cd9660_read`这个函数：

```

rc = cd9660_read(a);


```

从函数参数到参数指针

大部分的类UNIX操作系统，如：FreeBSD、Linux以及Solaris，都使用了函数指针来将文件系统的实现和访问它们中内容的代码分离开来。有趣的是，在FreeBSD中，它还使用了另外一个间接层来将`read`这个函数的参数抽象出来。

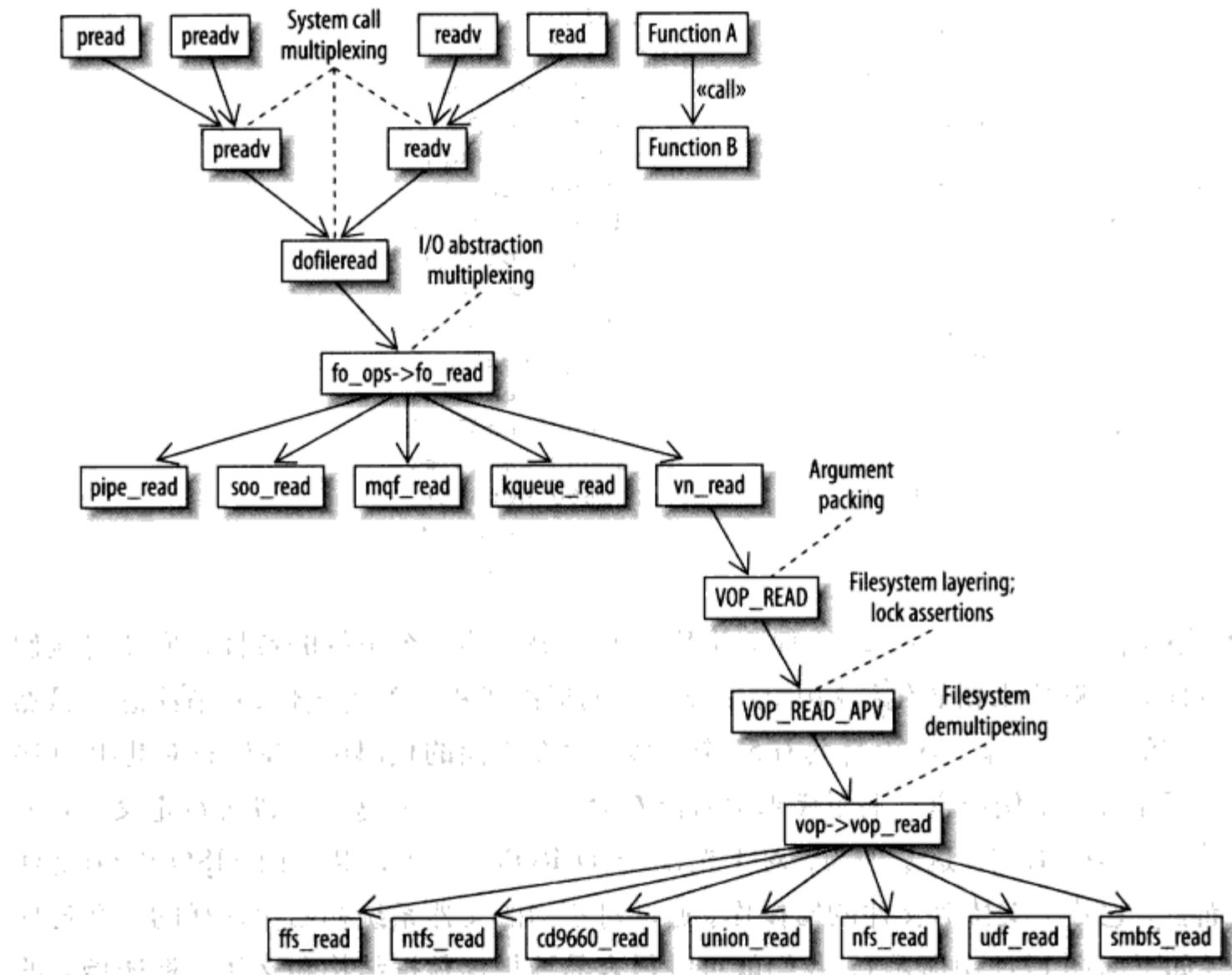


图 17-1: FreeBSD 中用以实现 read 系统调用的间接层

当我第一次碰到上面小节中给出的 `vop->vop_read(a)` 这个函数调用时, 我问我自己, `a` 这个参数到底是什么? 我们在更靠前一些的那个假想的 `VOP_READ` 函数中所给出的那四个参数又到哪去了呢? 经过一些深入的挖掘, 我发现了, FreeBSD 的内核在已经分层的文件系统之上还使用了另外一个间接层。这个新增的层可以让文件系统基于底下的文件系统, 向外提供某些服务 (如: 半透明视图、压缩以及加密等)。为了支持这个特性, FreeBSD 中很巧妙地使用了如下两种机制: 一个简单的、用来修改所有 `vop_vector` 中函数的参数的 `bypass` 函数; 另外一个则可以让所有没有被定义的 `vop_vector` 中的函数调用被重定向到底层文件系统中。

我们可以在图 17-2 中看到这两种机制。该图中展现了三个不同的文件系统层次。在最顶上是 `umapfs` 文件系统, 它是系统管理员为了映射用户凭证而 `mount` 的一个文件系统。当该特殊磁盘是由不同的用户 ID 所创建时, 这个文件系统就很有用了。例如: 管理员可能希望底下文件系统中的用户 1013 向外显示成为用户 5325。

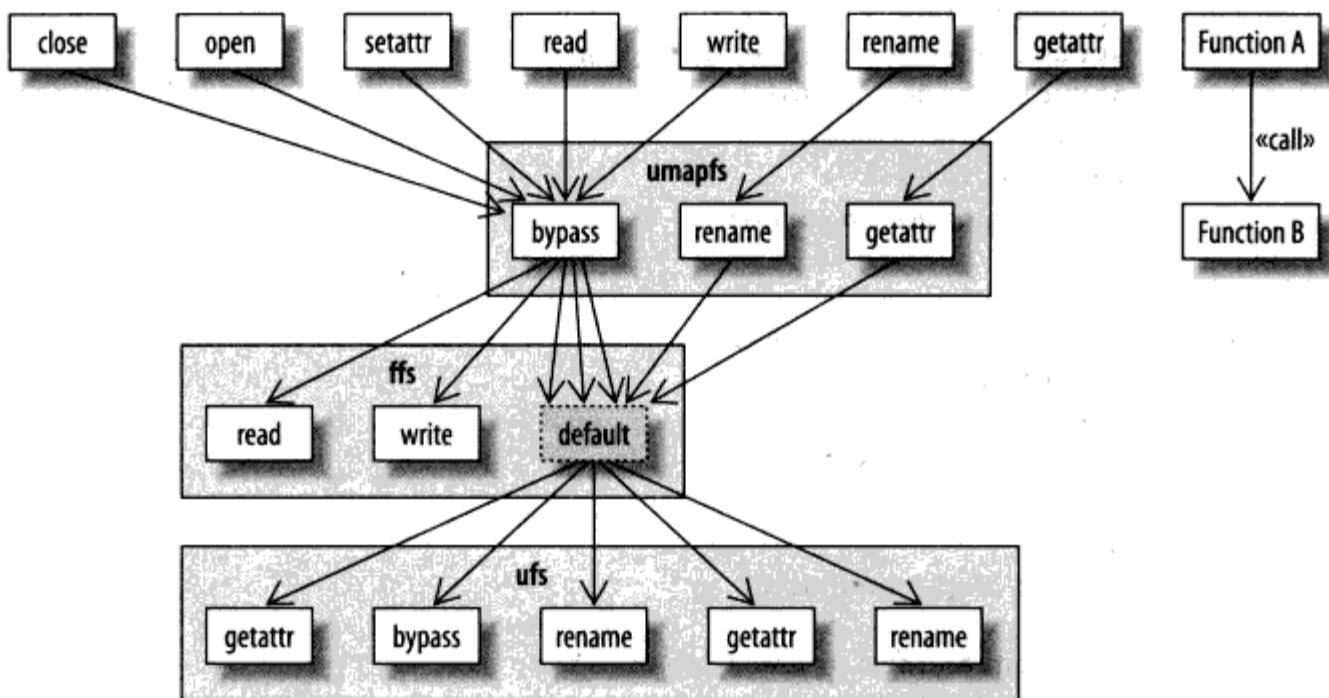


图 17-2：文件系统分层示例

在最上层的文件系统下面一层，是伯克利快速文件系统（Berkeley Fast Filesystem, ffs），它也是FreeBSD安装时缺省使用的在时间和空间上都可以做到很高效的文件系统。对于ffs的大部分操作，又将依赖于最早的BSD 4.2系统中所实现的ufs文件系统。

在上图所给出的那个例子中，大部分的系统调用都将经过umapfs中的一个公共函数bypass，它会为我们完成用户凭证的映射。只有很少的一部分系统调用，如：rename和getattr，我们才会在umapfs中实现它们。在ffs层中，它为read和write提供了一个被优化后的实现，这两个实现都依赖于ffs中那个比ufs中更高效的文件系统布局。而其他的大部分操作，如：open、close、getattr、setattr以及rename，我们都是调用了原来的实现来处理的。因此，在ffs所对应的vop_vector结构中，我们就可以通过vop_default这个入口来调用到底层的ufs实现中的那些操作。例如，对于系统调用read，它的调用途径如下，先是经过umapfs_bypass，然后是ffs_read；而对于rename，它的调用路径就包括了umapfs_rename和ufs_rename。

为了可以在不同的文件系统函数间使用同样的接口，以及支持bypass这个函数，我们在bypass和default这两种机制中，将前面那四个参数封装成为一个结构。这也是个漂亮的设计模式，不过我们通常都会因为把目光放在那些错综复杂的C代码实现上而忽视它。

前面那四个参数被封装成为一个结构，它的第一个字段(a_gen.a_descs)包含的是对数据结构内容的描述（在下面的代码中，就是vop_read_desc）。我们也可以从图17-1中看出，在FreeBSD内核中对文件进行read系统调用时，会触发一个对vn_read的调用，它又会设置一些相应的底层参数来调用VOP_READ。然后又对参数进行封装来调用VOP_READ_APV，最终调用到vop->vop_read，也就是实际的那个文件系统上的read函数。

```

    struct vop_read_args {
        struct vop_generic_args a_gen;
        struct vnode *a_vp;
        struct uio *a_uio;
        int a_ioflag;
        struct ucred *a_cred;
    };

    static __inline int VOP_READ(
        struct vnode *vp,
        struct uio *uio,
        int ioflag,
        struct ucred *cred)
    {
        struct vop_read_args a;
        a.a_gen.a_desc = &vop_read_desc;
        a.a_vp = vp;
        a.a_uio = uio;
        a.a_ioflag = ioflag;
        a.a_cred = cred;
        return (VOP_READ_APV(vp->v_op, &a));
    }
}

```

同样的步骤也发生在 `vop_vector` 的其他函数调用上（如：`stat`, `write`, `open`, `close` 等）。在 `vop_vector` 中，它同样也有一个指向 `bypass` 函数的函数指针。函数会取得被封装好的参数，经过对这些参数进行可能修改（如：它可能在不同的域之间做用户证书的映射）后，通过 `a_desc` 字段把控制权转交给相应的底层函数。

下面的代码摘自 `nullfs` 文件系统中对 `bypass` 函数的实现。在 `nullfs` 文件系统中，它只是简单地将已有文件系统的一部分复制到一个全局文件系统名字空间的某处。因此，对于它的大部分操作而言，它只需要在它的 `bypass` 函数中调用相应的底层文件系统的函数即可。

```

#define VCALL(c) ((c)->a_desc->vdesc_call(c))
int
null_bypass(struct vop_generic_args *ap)
{
    /* ... */
    error = VCALL(ap);
    /* ... */
}

```

在上面的代码中，宏 `VCALL(ap)` 会把调用 `null_bypass` 的 `vnode` 操作（如 `VOP_READ_APV`）转移到下一层的文件系统上进行。在图 17-3 中我们就可以看出这种技巧的使用。

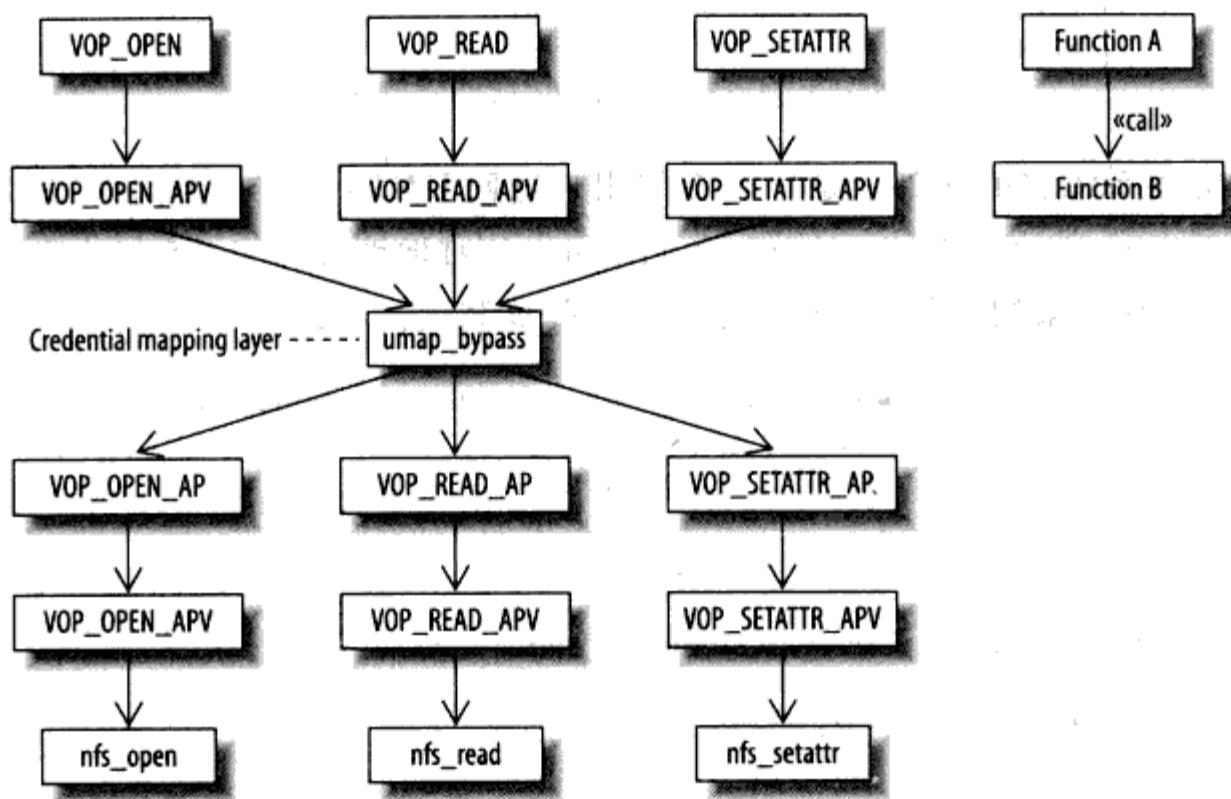


图 17-3: bypass 函数带来的系统调用路径

另外，在 `vop_vector` 中还有另外一个名为 `default` 的字段，它是一个指向底层文件系统层中 `vop_vector` 结构的指针。如果在本层的文件系统中没有实现某些功能，那么那些功能请求就会通过该字段转到下一层的文件系统上去执行。通过提供 `vop_vector` 结构中的 `bypass` 和 `default` 字段，文件系统可以进行如下的抉择：

- 在自身内部处理外来请求。
- 在对参数进行修改后，将请求传递给下一层的文件系统去执行。
- 直接调用下一层文件系统中的功能。

在我脑海中，我把这个想像成为一个有着斜坡、反弹装置以及旋转器的弹球机（注：可参考 Windows 中自带的 pinball 游戏）。从系统调用 `read` 实现中找出的下面这个例子，向我们展示了系统是如何来处理这个调用的：

```

int
VOP_READ_APV(struct { vop_vector      *vop, struct vop_read_args *a)
{
    [...]
    /*
     * 深入文件系统层查找一个函数实现或者 bypass 函数实现
     */
    while (vop != NULL &&
           vop->vop_read == NULL && vop->vop_bypass == NULL)
        vop = vop->vop_default;
    /* 调用这个函数或者 bypass 函数 */
    if (vop->vop_read != NULL)

```

```

        rc = vop->vop_read(a);
    else
        rc = vop->vop_bypass(&a->a_gen);

```

在所有的文件系统层的底部，是一个专门用来返回 UNIX 中“操作不被支持”错误 (EOPNOTSUPP) 的文件系统，这样，我们就可以优雅地解决那些没有被上面的文件系统所实现的函数带来的问题。下面就是我们的弹球装置：

```

#define VOP_EOPNOTSUPP ((void*)(uintptr_t)vop_eopnotsupp)
struct vop_vector default_vnodeops = {
    .vop_default = vop_default,
    .vop_bypass = vop_eopnotsupp,
};
int
vop_eopnotsupp(struct vop_generic_args *ap)
{
    return (EOPNOTSUPP);
}

```

从文件系统到文件系统层

对于一个具体的文件系统分层的例子而言，我们来考虑一下如下场景：我们需要在本机上通过 NFS (Network File System, 网络文件系统) 协议来映射一个远程文件系统。遗憾的是，在这台远程的系统中，它里面的用户标识和组标识不能对应到我们本机上的用户和组。然而，通过在 NFS 协议之上使用 *umapfs* 文件系统，我们可以通过外部文件来指定正确的用户和组映射关系。图17-3中就展示了操作系统内核函数是如何通过 *umapfs* 中的 *bypass* 函数 (*umap_bypass*) 来选择下一步操作，直到最终调用到相应的 NFS 客户端函数。

和 *null_bypass* 函数不同的是，在 *umap_bypass* 的实现中，在对底层文件系统进行调用之前，它还是做了一些事情的。在作为参数被传给它的结构 *vop_generic_args* 中，里面包含了对每个 *vnode* 操作的实际参数的描述。

```

/*
 * 通用结构。
 * Bypass 例程将使用这个结构来标识通用参数。
 */
struct vop_generic_args {
    struct vnodeop_desc *a_desc;
    /* 其他数据 */
};

/*
 * 在这个结构中描述了正在进行的 vnode 操作。
 */
struct vnodeop_desc {

```

```

    char      *vdesc_name;          /* 用于调试的可读名字 */
    int       vdesc_flags;          /* VDESC_* 标志 */
    vop_bypass_t *vdesc_call;     /* 将要调用的函数 */

    /* 
     * bypass 例程将使用这些结构来进行映射和查找参数。
     * 在 bypass 例程中并不需要 cred 信息和 proc 信息,
     * 但它们有时候对 (例如) 传输层来说是有用的。
     * Nameidata 很有用正是因为它包含了一个 cred。
     */
    int      *vdesc_vp_offsets;    /* 以 VDESC_NO_OFFSET 结尾的数组 */
    int      vdesc_vpp_offset;     /* 返回 vpp 位置 */
    int      vdesc_cred_offset;    /* cred 位置 */
    int      vdesc_thread_offset;  /* 线程位置 */
    int      vdesc_componentname_offset; /* 组件名字 */
};


```

例如，传递给 vop_read 操作的参数结构 vnodeop_desc 如下：

```

struct vnodeop_desc vop_read_desc = {
    "vop_read",
    0,
    (vop_bypass_t *)VOP_READ_AP,
    vop_read_vp_offsets,
    VDESC_NO_OFFSET,
    VOPARG_OFFSETOF(struct vop_read_args,a_cred),
    VDESC_NO_OFFSET,
    VDESC_NO_OFFSET,
};


```

重要的是，除了（为调试而包含的）函数名和底层系统调用（VOP_READ_AP）之外，在结构的 vdesc_cred_offset 字段中还包含了存放用户证书的数据字段（a_cred）的偏移量（相对 read 调用的参数而言）。通过使用该字段，umap_bypass 可以通过下面的代码把用户证书和任意对 vnode 的操作对应起来：

```

if (descp->vdesc_cred_offset != VDESC_NO_OFFSET) {
    credpp = VOPARG_OFFSETTO(struct ucred**, 
        descp->vdesc_cred_offset, ap);
    /* 保存原来的值 */
    savecredpp = (*credpp);
    if (savecredpp != NOCRED)
        (*credpp) = crdup(savecredpp);
    credp = *credpp;
    /* 对权限结构中的所有 id 进行映射。 */
    umap_mapids(vp1->v_mount, credp);
}

```

在这里，我们举例演示了数据是如何描述其他数据的格式，也即关于数据抽象的一个重定向。这种元数据使得我们可以在证书映射代码中对任意系统调用的参数进行处理。

从代码到 DSL (Domain-Specific Language)

我们可能会注意到，在 read 系统调用的实现中，部分相关代码，如：把参数封装成一个结构、或者选择相应函数进行调用的逻辑，都是高度风格化的代码，对于所有 52 个接口的实现来说，它们都可能存在相似格式的代码。另外一个我们还没有讨论的实现细节，也是可能使得我们夜不能寐的实现细节，便是和锁相关的实现细节。

操作系统必须确保，对于并发出现的多个操作，当它们对数据进行修改时不会出现操作间的数据不同步现象。现代的多线程、多核处理器都是通过一个互斥锁来确保数据的一致性的，这会为所有的操作系统中使用的数据结构加锁（如早期的操作系统实现）带来难以忍受的性能问题。因此，现在锁都被用在细粒度的对象上，如用户证书或者缓冲区。而且，由于获得和释放锁的操作比较耗时，理想状态下一旦对象被锁住，如果在短期内还需要被锁保护，那么它就不应该释放加在它之上的锁。这种锁规范最好还是用前置条件（也就是在进入函数前锁的状态应该如何）和后置条件（在函数退出后的锁状态）来描述。

正如你想像的那样，在上述限制下进行编程和验证代码的正确性将会是非常复杂的。幸运的是，对于我来说，一个额外的间接层就可以让这个复杂的图变得简单一些。我们可以用这个间接层来封装代码处理冗余，以及用它来对付那些脆弱的锁要求。

在 FreeBSD 内核中，我们在前面所描述过的那些数据结构的接口函数，如：VOP_READ_AP、VOP_READ_APV 以及 vop_read_desc，它们都不是直接用 C 语言编写的。相反，我们使用了一种 DSL 来描述每个函数的参数类型以及它们的锁的前、后置条件。这样的实现风格总是让我兴奋，因为它能为生产效率带来极大的提升。下面的代码摘自 read 系统调用的实现：

```
#  
#% read          vp      L L L  
#  
vop_read {  
    IN struct vnode *vp;  
    INOUT struct uio *uio;  
    IN int ioflag;  
    IN struct ucred *cred;  
};
```

上面的规范中，我们使用了 *awk* 脚本来创建：

- 用于封装函数参数到一个结构中的 C 代码。
- 保存封装后参数的结构的声明以及函数的声明。

- 用指定的内容初始化封装参数用的那个数据结构。
- 我们前面看过的用来实现文件系统层次的 C 代码。
- 用来在函数进入 / 退出时验证锁状态的断言。

在 FreeBSD 6.1 中的 *vnode* 的调用接口实现中，总共有 588 行这样的 DSL 代码，它们产生了 4339 行 C 代码和声明。

在计算领域中，将高级的领域特定语言编译为 C 语言是很常见的操作。例如，词法分析生成器 *lex* 的输入是一个将常规表达式映射为动作的文件；解析生成器 *yacc* 的输入则是语言的语法以及相应的产生规则。这两个系统（以及它们的衍生系统 *flex* 和 *bison*）都可以通过生成 C 代码来实现高级语言规范。C++ 编程语言的早期实现是一个更极端的示例，在这个实现中包含了一个预处理器 *cfront*，用来将 C++ 语言编译成 C 语言。

在所有的情况下，C 均被用作为一种可移植的汇编语言。如果运用得当，那么领域特定语言可以提升代码的表达能力并因此而提高程序员的生成效率。而另一方面，如果运用不当，那么领域特定语言将使得系统更加难以理解、调试和维护。

我们需要进一步阐述对锁定断言的处理。对每个参数来说，在代码中将列出以下三种情况中的锁定状态：进入函数时，成功退出函数时，以及由于错误而退出函数时——把问题很好地隔离开来。例如，在前面 *read* 调用的规范中指出，参数 *vp* 在所有这三种情况下都应该被锁定，而更复杂的情况也是可能发生的。在下面的代码中，*rename* 调用参数 *fdvp* 和 *fvp* 始终处于锁定状态，而当这个例程被调用时，参数 *tdvp* 有一个进程独占的锁定。但在函数结束时，应该对所有的参数解除锁定：

```

#
#% rename      fdvp    U U U
#% rename      fvp     U U U
#% rename      tdvp    E U U
#

```

锁定规范是用来在函数的入口位置、正常退出位置以及发生错误的退出位置等地方的 C 代码中插入断言。例如，在 *rename* 函数入口位置的代码中将包含以下断言：

```

ASSERT_VOP_UNLOCKED(a->a_fdvp, "VOP_RENAME");
ASSERT_VOP_UNLOCKED(a->a_fvp, "VOP_RENAME");
ASSERT_VOP_ELOCKED(a->a_tdvp, "VOP_RENAME");

```

虽然像上面这样的断言并不能保证代码中没有 bug，但它们至少提供了早期故障指示，从而在这些 bug 破坏系统和妨碍调试之前，程序员在系统测试时就能够找出它们。当我在阅读那些没有断言的复杂代码时，就像观看没有安全网的走钢丝表演一样：一个小小的失误就有可能导致灾难性的后果。

复用与分离

正如你在图 17-1 中所看到的，对 `read` 系统调用的处理并不是从 `VOP_READ` 开始的。`COP_READ` 实际上是在 `vn_read` 中调用的，而它又是通过一个函数指针来调用的。

这种层次的间接有另一种用途。在 UNIX 操作系统及其衍生系统中，所有的输入源和输出源都被统一对待。这样，`read` 在读取文件、套接字或者管道时，并非调用不同的系统调用，而是在所有这些 I/O 的抽象上进行读取操作。我发现这种设计不仅是优雅的，而且是非常有用的；我经常依赖这种设计，并且另辟蹊径来使用这些工具。（这里要强调的是这些工具年代久远，不是我的创造）。

在图 17-1 中部出现的间接是 FreeBSD 用来提供高级 I/O 抽象独立性的机制。每个文件描述符都与一个函数指针关联在一起，这个指针用来指向执行特定请求的代码：用于管道的 `pipe_read`，用于套接字的 `soo_read`，用于 POSIX 消息队列的 `mqf_read`，用于内核事件队列的 `kqueue_read` 以及用于文件的 `vn_read`。

到目前为止，在我们的示例中遇到了两种将函数指针用于将请求分发到不同函数的情况。通常，在这些情况下，函数指针用于将单个请求分解到过多个提供者中。这种用法的间接是非常普遍的，它构成了面向对象语言的重要部分，具体的形式就是把请求动态分发到不同的子类方法。对我来说，在像 C 这样的过程式语言中实现动态分发机制是大师级程序员的标志（同样的标志还包括能够用汇编语言或者 Fortran 语言编写结构化程序）。

间接通常还被用作为分解普通功能的一种方式。让我们看看图 17-1 的上部。在目前的 Unix 系统中有四种 `read` 系统调用的变化形式。以字母 p 开头的变化形式 (`pread`, `preadv`) 能够处理调用中的文件位置说明。以字母 v 结尾的变化形式 (`readv`, `preadv`) 能够处理一组 I/O 请求而不是单个请求。虽然我认为这种对系统调用的变化形式是不优雅的，并且也违背了 UNIX 的精神，但应用程序的开发员们似乎依赖这些变化形式来提升网页服务器或者数据库服务器每一分性能。

所有这些系统调用都共享相同的代码。在 FreeBSD 的实现中通过额外的函数来引入间接，从而避免了重复的代码。函数 `kern_preadv` 处理前两个系统调用变化形式中的共同部分，而 `kern_readv` 则处理余下的两个系统调用。在所有四个调用中的共同功能是由另一个函数处理的，即 `dofileread`。在我的脑海中可以想像得出，通过引入更多层次的间接，从而把共同的代码提取出来放到这些函数中，程序员该是多么地开心。如果在提交了重构修改后，我所增加的代码行数少于所删除的代码行数，那么我会觉得非常高兴。

从用户层程序对read函数的调用到移动磁头从盘片上获取数据是一个漫长而又复杂的过程。在我们的介绍中，我们并没有考虑在内核层之上发生的事情（虚拟机，缓冲，数据表示等），或者在文件系统处理请求（例如缓冲，设备驱动，数据表示）时会发生什么事情。有趣的是，我们并没有讨论在这两者之间的一个良好对称性：它们都涉及了硬件接口（虚拟机，例如上层的JVM以及在底层的实际接口），缓冲（在上层用于最小化系统调用，在底层用于优化硬件的性能），以及数据表示（在上层用于与用户进行交互，在底层用于满足物理层的需求）。似乎间接存在于我们所看到的每个地方。在我们已经看到的表示块、九层函数调用、两个通过函数指针的间接以及领域特定语言中，都为我们提供了间接的主要功能说明。

分层是永恒之道吗

我们还可以继续阅读更多的代码示例，因此在结束讨论的时候，我们有必要关注一下Lampson把引发我们进行讨论的格言（所有计算机科学领域中的问题都可以通过一个额外的间接层来解决）归功于David Wheeler，子例程的发明者。Wheeler还补充了一句话：“但这通常会产生其他的问题”。确实，间接层次将增加程序的空间开销和时间开销，并且还将降低代码的可理解性。

通常，时间和空间的开销是不重要的，我们很少会关心它们。在大多数的情况下，额外的指针或者子例程调用所带来的延迟与整体性能相比是很小的。事实上，编程语言的目前趋势是，通过间接来实现某些操作，从而提供更高的灵活性。因此，在像Java和C#等语言中，几乎所有的对象访问都是通过指针间接来进行的，这是为了实现自动垃圾收集的机制。同样，在Java中几乎所有的实例方法调用都是通过查找表来分发的，这是一种使派生类能够在运行时覆盖基类的方法。

除了这些对象访问和方法调用所带来的开销之外，这两个平台在市场上表现得都很好。在其他的情况下，编译器在优化时将去掉开发人员在代码中的使用间接。因此，大多数的编译器将检测出那些函数调用开销要高于函数内联开销的地方并自动替换为函数内联的形式。

再次指出，当我们在对性能有着极端要求的情况下工作时，间接可能会成为一种性能负担。使用千兆网接口来加速代码执行的开发人员可以使用的一个技巧就是把不同层次的网络堆栈功能结合起来，并且去掉一部分抽象层。但这些都是在极端情况下使用的方法。

另一方面，间接对代码可理解性的影响也是非常重要的问题，因为在最近50年，与CPU速度的突飞猛进相比，人类理解代码的能力并没有得到很大的提升。因此，敏捷方法的

倡导者建议我们在引入间接层次来处理一些模糊的和未定的需求时要非常小心，这些需求并不是当前的具体需求，而是我们认为在将来可能会突然出现的需求。正如 Bart Smaalders 在讨论性能逆模式（anti-pattern）时嘲笑的那样：“只有蛋糕才会分层，而软件不会。”

七原縣史面外勦匪

，数据驱动的面向对象的系统设计方法论。

，我界嶺群與貴土誠市落合平个兩事，我立脚在頭來審視兩國古跡同起居休憩近丁余
歲達人，均以。避面俱到的中西升舟員人數甚與大英相對的小國利源（南歐諸國地處
鄰內幾酒夜遊夢寐日其亡故由醉汗銀內她居干高委斯升根樹發化者無別制御他無相得

目前尚有一大类会通过外阴、修补术不治疗的需要部分骨盆或部分阴道再造术，出版率再高也应同时强调改善整个生殖功能的作用。人类生殖医学的分枝越来越多地涉及生殖内分泌学、生殖细胞生物学、生殖器重建术和辅助生育技术等。

Python 的字典类： 如何打造全能战士

Andrew Kuchling

字典是 python 语言的基本数据类型，它的作用就像 awk 里的关联数组或者 Perl 里的哈希表。在字典里，每个数据元素都有惟一的关键字（key）跟它对应，所有的数据元素和关键字对构成了一张映射表。字典的基本操作有：

- 追加一个新的关键字 - 数据元素对。
- 根据某个关键字访问对应的数据元素。
- 删除已有的关键字 - 数据元素对。
- 遍历所有关键字、遍历所有数据元素或者遍历所有关键字 - 数据元素对。

让我们看看，在 Python 解释器的提示符下，字典用起来是什么样子的。（你可以在 Mac OS 和大多数 Linux 下敲入命令 “python” 来调入 python 解释器。如果机器上还没有安装 Python，你可以从 <http://www.python.org> 下载。）

在下面的例子里，符号>>> 是 Python 解释器的提示符，d 是我们用到的字典实例：

```
>>> d = {1: 'January', 2: 'February',
... 'jan': 1, 'feb': 2, 'mar': 3}
{'jan': 1, 1: 'January', 2: 'February', 'mar': 3, 'feb': 2}
```

```
>>> d['jan'], d[1]
(1, 'January')
>>> d[12]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 12
>>> del d[2]
>>> for k, v in d.items(): print k,v    # Looping over all pairs.
jan 1
1 January
mar 3
>feb 2
```

字典类型有两点值得我们注意：

- 一个字典实例的关键字和数据元素可以是各种类型的数据。比如，在字典里同时存在如下关键字是合法的：整数 1，复数 $3+4j$ ，字符串 "abc"。它们在字典里还是原来的数据类型，不用转换成字符串。
- 字典里的关键字不是有序的。比如，方法 `.values()` 会返回整个字典的内容，每个项目之间的次序是任意的，不根据数据元素的值排序，也不按加入字典的时间先后为序。

显然，能够快速定位关键字是至关重要的，为了做到这一点，我们通常用哈希表来实现字典这样的类型。Python 的 C 语言实现版本里（以后称 CPython），字典更具有核心地位，因为它是其他语言功能的基石。例如，类和类实例都用字典来保存它们的属性：

```
>>> obj = MyClass()          # 建立一个类实例
>>> obj.name = 'object'      # 加入属性.name
>>> obj.id = 14              # 加入属性.id
>>> obj.__dict__             # 访问内部的字典
{'name': 'object', 'id': 14}
>>> obj.__dict__['id'] = 12   # 在内部字典里存入一个新的数据元素
>>> obj.id                  # 对应的属性值也相应地变了
12
```

Python 中模块的内容也用字典存储。众所周知的模块 `__builtin__` 就包含了像 `int`、`open` 这样的内建标识符，任何使用这些内建标识符的表达式里最终都会涉及到字典检索。另外，在用参数关键字作为函数参数时，每次函数调用前后，Python 都可能会建立并释放一个字典实例。Python 内部如此依赖字典，以至于即使用户在写程序时就算没有用到任何字典类，运行时也会有很多字典实例活跃其中。这就要求，字典的建立和销毁都必须非常迅速，并且不能消耗过多内存。

通过参与 Python 字典类的实现，我们对开发性能要求苛刻的程序有了一些体会。首先，

开发人员要权衡优化操作带来的空间/时间负荷。Python 开发人员发现，有时候一种简单直结的做法，从长远看，要胜过那些刚开始看起来很不错的、但更加复杂的优化方案。也就是说，保持简单总是有好处的。

其次，现实运行环境下的性能评测是至关重要的，只有这样你才能发现哪些东西才是真正值得投入花力气的。

字典类的内部实现

字典类在 C 语言里用结构 `PyDictObject` 表示，它的定义可以在头文件 `Include/dictobject.h` 里找到。下面的例子形象地描述了一个字典的结构，它保存了从 "aa"，"bb"，"cc".....,"mm" 到整数 1 - 13 的一一映射。

```
int ma_fill      13
int ma_used     13
int ma_mask     31
PyDictEntry ma_table[]:
[0]: aa, 1           hash(aa) == -1549758592, -1549758592 & 31 = 0
[1]: ii, 9           hash(ii) == -1500461680, -1500461680 & 31 = 16
[2]: null, null
[3]: null, null
[4]: null, null
[5]: jj, 10          hash(jj) == 653184214, 653184214 & 31 = 22
[6]: bb, 2           hash(bb) == 603887302, 603887302 & 31 = 6
[7]: null, null
[8]: cc, 3           hash(cc) == -1537434360, -1537434360 & 31 = 8
[9]: null, null
[10]: dd, 4          hash(dd) == 616211530, 616211530 & 31 = 10
[11]: null, null
[12]: null, null
[13]: null, null
[14]: null, null
[15]: null, null
[16]: gg, 7           hash(gg) == -1512785904, -1512785904 & 31 = 16
[17]: ee, 5           hash(ee) == -1525110136, -1525110136 & 31 = 8
[18]: hh, 8           hash(hh) == 640859986, 640859986 & 31 = 18
[19]: null, null
[20]: null, null
[21]: kk, 11          hash(kk) == -1488137240, -1488137240 & 31 = 8
[22]: ff, 6           hash(ff) == 628535766, 628535766 & 31 = 22
[23]: null, null
```

```
[24]: null, null  
[25]: null, null  
[26]: null, null  
[27]: null, null  
[28]: null, null  
[29]: 11, 12           hash(11) == 665508394, 665508394 & 31 = 10  
[30]: mm, 13           hash(mm) == -1475813016, -1475813016 & 31 = 8  
[31]: null, null
```

域成员的名字前缀 `ma_` 的意思是映射 (mapping)，在 Python 里，它专指那些提供关键字 / 数据元素匹配功能的数据结构。`PyDictObject` 结构的域成员包括：

`ma_used`

有关键字的存储单元(slot)数量 (本例中是 13)。

`ma_fill`

有关键字存储单元和哑存储单元的总数(本例中仍是 13)。

`ma_mask`

对应哈希表大小的比特掩码。哈希表共拥有 `ma_mask+1` 个存储单元 (本例中为 32)。规定一个表的存储单元数永远是 2 的幂，所以，`ma_mask` 总是可以表示成如下型式： $2^n - 1$ ，而且长度为 n 比特。

`ma_table`

指向 `PyDictEntry` 结构的数组。`PyDictEntry` 结构包含了几个指针，分别指向：

- 关键字对象。
- 数据元素对象。
- 保存关键字哈希值的缓存。

缓存哈希值的目的是为了提高速度。在查找一个关键字时，用它的哈希值作比较会比根据关键字内容做全面比较快得多。当字典大小发生变化时也需要计算每个关键字的哈希值，缓存哈希值还可以省去重新计算的时间。

我们不用记住哈希表的存储单元总数，只需从 `ma_mask` 推导即可。当需要根据一个关键字来访问保存它的存储单元时，可以根据“存储单元号 = 哈希值 & 掩码”来计算初始存储单元的位置。在上面的例子中，第一个数据单元的哈希值是 -1549758592， $-1549758592 \bmod 31$ 等于 0。因此，第一个数据单元保存在 0 号存储单元上。

由于掩码使用得非常频繁，因此我们保存的是掩码而不是存储单元总数。因为只要把掩码加 1，就是存储单元总数。而且，在对速度要求很高的代码中，我们从来都不需要这么做。

`ma_fill` 和 `ma_used` 是反应存储单元数量的变量。`ma_used` 是字典中关键字的数量，加入一个关键字时，这个值增 1，删除时减 1。当执行删除关键字操作时，我们只需要让相应的存储单元指向一个哑关键字，所以 `ma_fill` 的值并不变化。它只有在加入一个关键字时才会增加 1。（`ma_fill` 的值永远不会减少，除非字典的大小发生调整。）

特殊调校

看来字典是不折不扣的全能战士：对 Python 用户来说，它是时间和空间维度上都很高效的数据类型；对 Python 解释器的来说，它又是一个内部数据结构；对 Python 的开发人员来说，它是供人阅读、需要维护的代码库。因此，我们既要让它纯净、优美，也要融入少量针对特别情况的特殊代码。

为小哈希表而做的特别优化

`PyDictObject` 还包括一个拥有 8 个存储单元的哈希表。不多于 5 个元素的小字典可以直接放在这张表内，从而免去了调用 `malloc()` 的额外开销。同时这样做还提高了缓存的局部性，拿 x86 GCC 编译的 `PyDictObject` 来说，它的大小是 124 字节，刚好可以放进两个大小为 64 字节的缓存行。因为一般参数关键字用到的字典大部分都只包含 1 到 3 个关键字，所以小哈希表的这种优化措施同时也提高了函数调用的性能。

需要付出一些代价的特例

上面说过，一个字典里可以包括多种数据类型的关键字。不过在大多数 Python 程序的类实例和模块里，它们内部的字典只用字符串作为关键字。人们自然就会想到，如果设计一种特别的字典，它只接受字符串作为关键字类型，也许很有用。只接受特定数据类型的字典也许可以让解释器跑得更快，谁知道呢？

Java 版字典：另类优化方案

实际上，Python 的 Java 版本，Jython (<http://www.jython.org>)，就有一个专用字符串的字典类型：`org.python.org.PyStringMap`。在 Jython 的类实例和模块里内建的 `__dict__` 字典中用到了它。Jython 用户编程时所用的字典类实际上是另外一个叫作 `org.python.core.PyDictionary` 的类型，它内部使用了 `java.util.Hashtable` 类来保存内容，并加了一个间接层来提供子类化能力。与内建的字典类相比，它算得上是重量级选手了。

Python 语言不允许用户用其他数据类型替换内建的 `_dict_` 字典类，从而免去了子类化的麻烦。所以对 Jython 来说，有一个专门的字符串类型字典还是有意义的。

C 版优化方案：动态选择存储方法

CPython 跟 Jython 不同，它没有引入专用字典类型，而是用了另外一个技巧：一开始，字典调用只处理字符串的方法；当用户需要检索非字符串数据类型时，就调用能够处理通用数据类型的方法。这个做法实现起来其实不难。`PyDictObject` 只有一个域成员 `ma_lookup`，它是一个函数指针，指向的函数负责检索关键字：

```
struct PyDictObject {
    ...
    PyDictEntry *(*ma_lookup)(PyDictObject *mp, PyObject *key, long hash);
};
```

`PyObject` 是 C 结构，用来保存任意 Python 数据类型，它的基本域成员包括一个引用计数和一个指向类型对象的指针。像 `PyIntObject` 和 `PyStringObject` 这样的特定数据类型都是在这个结构的基础上扩展出来的。字典类通过调用 `(dict->ma_lookup)(dict, key, hash)` 来查找关键字，里面的参数 `key` 是一个指针，它指向代表关键字的 `PyObject`；参数 `hash` 是根据关键字算出的哈希值。

刚开始的时候，`ma_lookup` 指向函数 `lookdict_string`，这个函数假定字典里所包含的关键字和待检索的关键字都是字符串类型的，用结构 `PyStringObject` 表示。有了这个前提，`lookdict_string` 函数就可以作一些优化。首先，字符串比较时不会抛出异常，所以可以略过一些不必要的错误检查。其次，比较两个字符串是比较简单的事情，不像一般的 Python 对象，可能会有 `<`、`>`、`<=`、`>=`、`==` 和 `!=` 这样的比较操作。

如果字典里有非字符串类型的关键字，或是用户想查找这样的关键字时，`ma_lookup` 就会改成指向更加通用的查表函数。当 `lookdict_string` 函数发现其参数的实际类型不是字符串时，它就会修改 `ma_lookup`，并调用新的函数来进行查找。（CPython 小技巧：这就意味着，如果你对一个纯字符串版本的字典进行非字符串检索时，比如 `d.get(1)`，整体检索速度会比通用的情况要慢，哪怕这是一次失败查询。接下来，字典的检索函数就会切换到通用版本。这时候，就算他们查的还是字符串类型的关键字，查询速度已经不能跟纯字符串版本同日而语了。）`PyStringObject` 类型的子类应当被认为是非字符串类型，因为子类里可能会定义相等比较操作。

冲突处理

在实现哈希表时，一个重要的课题是如何处理两个关键字的哈希值指向同一个存储单元

的情况。一种途径是拉链法（参见 http://en.wikipedia.org/wiki/Hash_table#Chaining）：每个存储单元都保存一个链表，链表上的所有元素的哈希值都对应当前存储单元。Python 没有采用这种办法。因为建立链表需要为每个链表元素申请内存空间，要知道内存分配是比较慢的。同时，遍历链表有可能会降低缓存局部性。

还有一个选择是开放地址法（参见 http://en.wikipedia.org/wiki/Hash_table#Open_addressing）：如果第一次在存储单元 i 中没有找到待查的关键字，那就按照某种固定模式尝试查找其他存储单元。最简单的尝试模式叫线性探测法：如果存储单元 i 被占了，那就尝试 $i+1, i+2, i+3\dots$ 如果遇到哈希表的末尾，那就折回到 0 号存储单元。在 Python 里，线性探测法的效率是比较低下的，因为很多程序使用连续整数作为关键字，结果存储单元往往是连成一片的。在线性探测时，经常需要全部扫描完这些存储单元片才能最终找到待查的关键字，从而导致检索性能低下。所以，Python 用了一种更加复杂的尝试模式：

```
/* 起始存储单元 */
slot = hash;
/* 初始干扰值 */
perturb = hash;
while (<存储单元非空> && <单元内的值不等于关键字值>)
    slot = (5*slot) + 1 + perturb;
    perturb >>= 5;
```

在 C 语言的代码里， $5 * slot$ 可以用位运算和加法来替代： $(slot \ll 2) + slot$ 。一开始，干扰因子 $perturb$ 直接取成哈希值；接下来，它的二进制值在每次循环的时候右移 5 位。移位操作可以让哈希值的每个比特都能够迅速地参与到探索尝试的计算中。经过若干次移位操作，干扰因子最终会变成 0，此时探索模式就退化成 $slot = (5 * slot) + 1$ ，它生成的数值可以是 0 和 ma_mask 之间的任何一个数。因此，这种尝试模式可以确保：要么是找到关键字（如果是在检索字典），要么是找到一个空的存储单元（如果是准备向字典插入数据）。

每次移位偏移量 5 个比特是根据实验得出来的。跟 4 或者 6 相比，5 可以把冲突率再降低一点。在以前版本的探索算法里，甚至用到了诸如乘法、除法这样的复杂操作。不过，这些方法尽管在统计上有很漂亮的结果，在实际操作时，它们的计算速度显得不够快。*(Objects/dictobject.c* 里有大段的注释详细讨论了这种优化的前世今生。)

调整大小

当关键字数量不断增多时，字典哈希表的大小就需要做出调整。现在的做法是保证哈希

表最多只有三分之二是满的。也就是说，如果一个字典有 n 个关键字，那哈希表就得至少有 $n/(2/3)$ 个存储单元。这个比例是折中的结果：哈希表填得太满就会导致更多的冲突，太稀疏的话就会浪费太多内存，同时也不利于缓存处理。还有人曾尝试过，让这个空/满比例随着哈希表大小的变化而变化，结果不甚理想。原因不难想到，每次往哈希表里插入数据时，都需要检查字典的大小是否需要做出调整。这样做会让插入操作变得很慢。

确定哈希表的新尺寸

当我们需要对字典的大小做出调整时，应然如何确定新的尺寸呢？对于不超过 5 万个关键字的中小规模哈希表，新的尺寸应当是 `ma_used * 4`。大多数使用大字典的 Python 程序都会在初始阶段构建字典，接下来做的就只是检索关键字或者是遍历字典。这时候，4 倍的倍率让字典显得有点稀疏（因为只有 $1/4$ 的存储单元是有数据的），换来的好处是在初始阶段构建字典时减少了字典调整的次数。因此，对于超过 5 万个关键字的大型字典，我们规定新的尺寸大小是 `ma_used * 2`，这样做可以避免浪费过多的内存空间。

当我们要从字典里删除关键字时，只需要让关键字所在的存储单元指向一个哑关键字即可。这时候，我们会更新 `ma_used` 计数，但不会检查非空存储单元的数量。这就意味着，在删除关键字时字典的大小不会变化。如果你创建了一个很大的字典，并从中删除了很多关键字，此时，字典的哈希表可能比你直接创建出这样的字典所得到的哈希表要大。当然，这种用法可能不太常见。对于那些在对象实例中和函数参数中常见的哈希表，它们的关键字通常根本不会被删除。一般的 Python 程序通常都是先建立一个字典，接着访问它，最后把整个字典删掉。因此，很少有 Python 程序会因为“删除关键字后不调整字典大小”而出现内存占用过多的问题。

一个合算的内存折中方案：空闲列表

Python 在用参数关键字作函数调用时，会创建字典实例，然后在函数返回时销毁它们。这种操作发生得十分频繁，字典实例的生命周期也很短暂。对于这种频繁创建、存在周期又很短的对象，一个有效的优化措施是回收那些不再使用的数据结构，从而减少 `malloc()` 和 `free()` 这样的内存分配操作。

Python 用一个叫做 `free_dicts` 的数组来保存不再使用的字典。在 Python 2.5 中，这个数组的长度是 80。当需要创建 `PyDictObject` 时，Python 会设法从 `free_dicts` 里找到一个指针，重用它所指向的数据结构。当字典被删除时，它们会被直接加到 `free_dicts` 数组中。如果此时数组已经满了，那就直接释放这个字典对象。

迭代和动态变化

遍历一个字典的所有内容是很常见的操作。`keys()`、`values()`和`items()`方法分别一次返回所有关键字的列表、所有数据元素的列表和所有关键字 - 数据元素对的列表。为了节省内存空间，用户可以调用`iterkeys()`、`itervalues()`和`iteritems()`来逐个遍历所需的元素。值得注意的是，在遍历迭代的过程中，Python 不允许增加或者删除字典的存储单元。

这个约束很容易实现。当`iter*`() 系列方法第一次被调用的时候，迭代器会记住字典中所有元素的个数。如果在遍历迭代过程中，字典大小发生变化，迭代器就会抛出`RuntimeError` 异常，告诉用户：“迭代过程中字典的大小发生了变化”。

在遍历迭代过程中不能修改字典内容。不过，这儿有一个例外，那就是给同一个关键字关联新的数据元素：

```
for k, v in d.iteritems():
    d[k] = d[k] + 1
```

在这种操作里，`RuntimeError` 是多余的，也是很不方便的。因此，如果字典插入的关键字已经存在，负责元素插入的 C 函数`PyDict_SetItem()`会确保字典的大小不会发生变化。为了做到这一点，`lookdict()`和`lookdict_string`在没有找到待查的关键字时，就会返回一个指针来告诉调用者：查找失败。这个指针会指向关键字所匹配的那个空白存储单元。这样，`PyDict_SetItem`得到的存储单元要么是空白的，要么就是待查关键字所匹配的，接着向里面存入一个新的数据元素就很容易。对于后面那种情况，就像`d[k]=d[k]+1`这样，我们根本没必要检查字典的大小是否发生变化，因此也根本不会有`RuntimeError` 异常发生。

结论

Python 的字典类提供了丰富的功能和众多的调整选项，它在 Python 内部有着广泛的应用。尽管身负重任，CPython 里字典的实现总的来说还是非常干净利落的。大部分的优化都是算法层面的调整，而且，如果有可能，都会有实验反复测试这些算法对冲突率和运行效率的影响。如果想了解关于字典类实现方面的更多细节，源代码是你最好的向导。首先，你可以通过 <http://svn.python.org/view/python/trunk/Objects/dictnotes.txt?view=markup> 找到 `Objects/dictnotes.txt`，这份注释文档详细地讨论了字典类常见的用例和各种优化技巧。（当然，不是所有的方法都被现在的代码采用了。）接着，你还可以参考字

典类的源代码：*Objects/dictobject.c*，地址是 <http://svn.python.org/view/python/trunk/Objects/dictobject.c?view=markup>。

通过阅读注释文档，并结合比较源代码，你可以更加透彻地理解那些问题。

致谢

感谢 Raymond Hettinger 对本章提供的宝贵意见。如果发现有什么错误，都算我的。

第 19 章

NumPy 中的多维迭代器

Travis E. Oliphant

NumPy 是 Python 语言的一个可选安装包，它提供了一个功能强大的 N 维数组对象。N 维数组是指一个通过 N 个整数，或者叫作 N 个下标，来访问数组中每个元素的数据结构。对于在计算机中处理的许多数据来说，这个对象是非常有用的模型。

例如，一维数组可以用来保存音频数据，二维数组可以用来保存灰阶影像，三维数组可以用来保存彩色影像（其中一维的下标长度为 3 或者 4），四维数组可以用来保存音乐会期间现场空间中的压力值，更高维的数组通常也是很有用的。

NumPy 为任意维数组的数学化操作和结构化操作提供了一种机制。在大多数的科学、工程以及处理大规模数据的多媒体代码中，这些操作都是核心的功能。此外，在高级语言中执行这些数学化和结构化的操作能够简化程序的开发以及实现这些算法的重用。

NumPy 提供了多种在数组上进行的数学计算，并且为结构化的操作提供了非常简单的语法。因此，Python（安装了 NumPy 之后）可以很好地应用于开发一些关键的并且要求运行速度很快的工程代码和科学代码。

在 NumPy 中，用来实现快速结构化操作的特性之一就是，NumPy 数组中的任意一个子

区域都可以通过切片（slicing）的概念来选取。在 Python 中，切片的定义形式是，一个起始下标，一个结束下标和一个跨度（stride），具体的记法为，起始下标：结束下标：跨度（包含在方括号中）。

例如，假设我们想通过在影像中间选择一个区域的方式，把 656×498 的影像缩放为 160×120 的影像。如果原始影像是保存在 NumPy 的数组 `im` 中，那么这个操作可以通过下面这个表达式来完成：

```
im2=im[8:-8:4, 9:-9:4]
```

不过，NumPy 的另一个重要特性就是按照这种方式选取的新影像将与原始影像共享数据，而并不会生成一个副本。在计算大型的数据集时，这将是一个非常重要的优化，因为随意地进行复制将会严重消耗计算机的资源。

N 维数组操作中的关键挑战

为了给所有的数学操作提供执行快速的实现，在 NumPy 中需要实现能够在一个或者多个任意维数的数组中快速进行处理的循环（用 C 语言编写）。通常，编写像这样能够在任意维数的数组中快速处理的通用代码是一项艰巨的任务。或许，编写单层 `for` 循环来处理一维数组中的元素，或者编写嵌套的双层 `for` 循环来处理二维数组是比较简单的任务。事实上，如果你能够提前知道数组的维数，那么就可以使用正确的 `for` 循环嵌套层数来直接处理数组中的所有元素。然而，当 N 是一个任意整数时，如何编写通用的 `for` 循环来处理 N 维数组中的所有元素呢？

对于这个问题有两种基本的解决方案。其中一种解决方案就是使用递归的方式，把这个问题分解为一个递归条件（recursive case）和一个基线条件（base case）。这样，如果 `copy_ND(a, b, N)` 是一个把 `b` 指向的 N 维数组复制到 `a` 指向的另一个 N 维数组的函数，那么可以使用下面这样的简单递归来实现：

```
if (N==0)
    将内存从 b 拷贝到 a
    return
构建 ptr_to_a 和 ptr_to_b
for n=0 到 a 和 b 的第一维大小
    copy_ND(ptr_to_a, ptr_to_b, N-1)
    ptr_to_a 加上 stride_a[0] 并且 ptr_to_b 加上 stride_b[0]
```

需要注意的地方包括单层 `for` 循环的使用以及如何判断当 N 为 0 时停止递归过程的基线条件。

如何把每个算法都改写为递归算法并不是件容易的事情，即使我们可以把上面给出的代码看作为如何转换为递归算法的起始模型。此外，递归算法还要求在循环的每次迭代中都进行函数调用。因此，递归算法很容易产生速度很慢的代码，除非你对基线条件做一些优化（例如，当 $N=1$ 时，在一个局部的 `for` 循环中进行内存拷贝）。

大多数的编程语言都不会自动进行这样的优化，因此，一个看上去不错的递归算法最终将由于优化所增加的时间而变得弄巧成拙。

此外，许多算法都需要保存中间信息以用于后续的递归调用中。例如，当我们需要计算数组中的最大值或者最小值时，该如何来调整递归调用呢？通常，这些值将被作为递归调用结构的一部分，并且在递归调用中作为参数被传递。最终，各种递归算法的编写方式肯定是大同小异的。这样，我们很难为程序员提供用于递归解决方案的简化工具。

因此，在 NumPy 中并没有使用递归，而是使用迭代来完成大多数的 N 维算法。每个递归算法都可以被改写为一个迭代算法。迭代器（Iterator）是一种简化这些算法的抽象。因此，在使用了迭代器之后，所开发出的 N 维算法可以运行地很快，同时代码仍然可以通过简单的循环结构来阅读和理解。

迭代器是一个抽象概念，其中包含了在单个循环内遍历数组中所有元素的思想。在 Python 中，迭代器是一些对象，它们可以被用作为任意 `for` 循环的谓词。例如：

```
for x in iterobj:  
    process(x)
```

在上面的代码中，在数组 `iterobj` 中的所有元素上都会执行函数 `process`。对于 `iterobj` 的最重要的要求就是，这个数组要能够通过某种方式来得到“下一个”元素。这样，迭代器的概念就把在所有元素上进行循环的问题归结为查找下一个元素的问题。

为了理解在 NumPy 中是如何实现和使用迭代器的，我们需要了解一些关于 NumPy 如何从 N 维数组的角度来看待内存的概念。在接下来的章节中我们将阐述这方面的内容。

N 维数组的内存模型

如果数组中的所有元素是在连续的内存段中一个接一个地排列着，那么就可以使用计算机中最简单的 N 维数组模型。在这种情况下，数组中的下一个元素就是通过把一个固定常量和指向当前数据内存地址的指针相加来得到的。因此，如果 NumPy 中的 N 维数组是邻接的，那么讨论迭代器将是很无趣的命题。

迭代器抽象的漂亮性在于它能够使我们把非邻接数组按照邻接数组的方式来进行处理和

考虑。在 NumPy 中存在非邻接数组的原因是，数组在创建时可能是被作为其他邻接内存区域的一个“视图”。这个新的数组本身可能并不是邻接的。

例如，我们来考虑一个在内存中邻接的三维数组 `a`。在 NumPy 中，你可以使用 Python 的切片记法来创建另一个数组，并且在新的数组中包含原始数组的一个子集。例如，`b=a[::2,3:,1::3]` 这条语句将返回另一个 NumPy 数组，其中包含的元素是：在第一维中从第一个元素开始，每跳过一个位置选取一个元素，在第二维中从第四个元素开始（下标从零开始）的所有元素，以及在第三维中的从第二个元素开始，每跳过两个位置选取一个元素。这个新的数组并没有复制在这些位置上的内存数据，而只是原始数组的一个视图，并且与原始数组共享这些数据。显然，这个新的数组无法用邻接内存块来表示。

通过二维数组的示意图可以进一步来理解这个概念。在图 19-1 中显示了一个邻接的二维 4×5 数组，数组每个元素的位置用 1 到 20 来标记。在 4×5 数组的图示之下是一个数组内存的线性图示，这也是在计算机中的保存形式。如果 `a` 表示整个内存块，那么 `b=a[1:3,1:4]` 就表示共享的区域（内存位置 7, 8, 9, 12, 13 和 14）。正如我们在线性图示中看到的，这些内存位置不是邻接的。

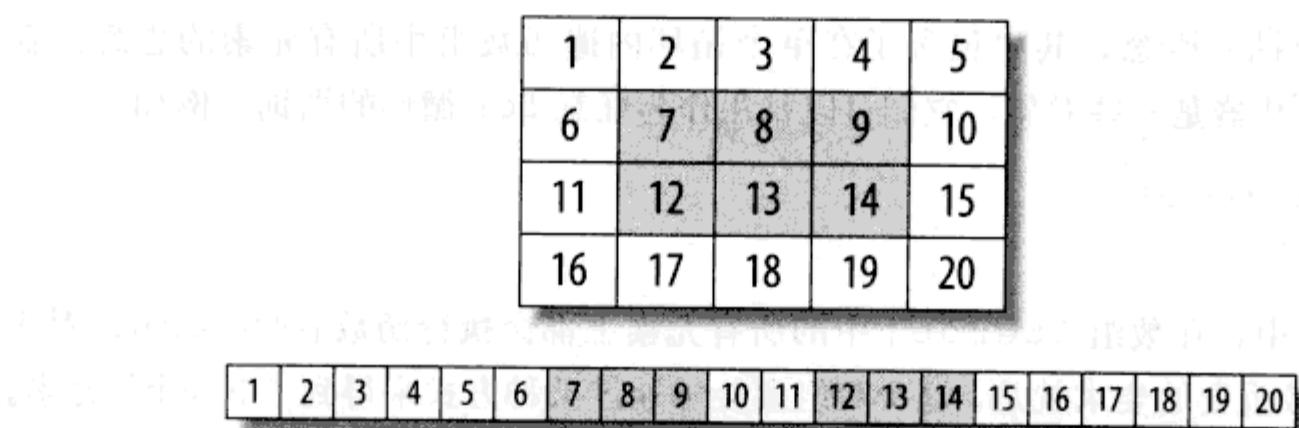


图 19-1：二维数组切片及其在内存中的线性图示

NumPy 中 N 维数组的通用内存模型能够支持创建数组的这些非邻接视图。这是通过把一系列表示每一维中的“跨度”值附加到数组上来实现的。

数组某一维的跨度值的含义是：如果沿着这一维，或者说数轴，从数组中的一个元素移动到下一个元素，需要跳过多少个字节。这个跨度值甚至可以是负数，表示在数组中的下一个元素是通过在内存中来回移动得到的。（潜在的）任意跨度的复杂性意味着构建能够处理通用情况的迭代器将变得更加困难。

NumPy 迭代器的起源

NumPy 为 Python 程序员提供的一个重要功能是，那些能够对已编译代码中的数组的所有元素进行遍历的循环。通过使用迭代器，我们可以更容易地把这些循环以直接的和易读的方式编写出来，并且能够处理 NumPy 支持的大多数一般情况。在我的记忆中，迭代器抽象是漂亮代码的一个示例，因为它使我们能够用简单的方式表达出简单的思想，即使底层的实现可能是很复杂的。这种类型的漂亮代码并不是偶然存在的，它通常是人们不断努力解决一组类似的问题并最终得到通用解决方案的结果。

我最初编写通用 N 维循环结构的想法是在 1997 年左右产生的，当时我想写一段代码，它可以同时用于 N 维卷积以及在 N 维数组的每个元素上执行某个 Python 函数的通用 arraymap 类。

当时我想出的解决方案是记录一组整数，并将这组整数作为 N 维数组的下标。这样，迭代运算就意味着用特定的代码来递增这个下标计数器：当下标达到数组某一维的长度时，把计数器回卷到零并且把下一维的计数器增加 1。

8 年后在编写 NumPy 时，我对 Python 中迭代器的概念和用法有了更为清晰的认识。因此，我开始考虑把 arraymap 的代码改写为一个正式的迭代器。在这个过程中，我研究了 Peter Vevreer (SciPy's ndimage 软件包的编写者) 实现 N 维循环的方法，并且发现了一个与我曾经使用过的方式非常类似的方法。这增强了我信心，然后我正式定义了迭代器，并且把 ndimage 中包含的思想应用到了 arraymap 和 N 维卷积代码中的基本结构要素上。

迭代器的设计

正如在前面所谈到的，迭代器是一个抽象概念，它封装了遍历数组中每个元素的思想。在 NumPy 中，使用迭代器的循环伪码如下所示：

```
构建迭代器  
(包括将迭代器的当前值指向数组中的第一个元素)  
while 迭代器没有结束:  
    对迭代器指向的当前元素进行处理  
    把迭代器的当前值设置为指向下一个元素
```

除了当前值的处理过程之外，其他所有的工作都必须由迭代器来完成，这非常值得我们讨论。因此，在迭代器的设计中基本上可以分为三个部分：

1. 移动到下一个值。

2. 终止。
3. 构建。

我们将对每一部分都单独进行讨论。在 NumPy 迭代器的设计考虑中，还包括尽可能地降低在循环内部使用迭代器的开销，以及尽可能地提高运行速度。

迭代器的递进

首先要确定的就是按照什么样的顺序来提取元素。虽然你也可以设计出一个对提取元素的顺序不作任何保证的迭代器，但在大多数时候，如果程序员能够知道顺序，那么将会是很有用的。因此，在 NumPy 中的迭代器遵循了一种特定的顺序。这个顺序是通过使用一组数字来模拟简单计数的方法而实现的。用一组 N 个整数表示数组中的当前位置，其中 $(0, \dots, 0)$ 表示 $n_1 \times n_2 \times \dots \times n_N$ 数组中的第一个元素，而 $(n_1 - 1, n_2 - 1, \dots, n_N - 1)$ 表示这个数组中的最后一个元素。

这组整数表示的是一个 N 数字计数器。下一个元素的位置是通过最后一个数字加 1 来得到的。如果在这个过程中，第 i 个数字到达了 n_i ，那么这个数字将被设置为 0，而第 $(i-1)$ 个数字将增加 1。

例如，对于 $3 \times 2 \times 4$ 的数组来说，计数过程如下：

$(0,0,0) (0,0,1) (0,0,2) (0,0,3) (0,1,0) (0,1,1) (0,1,2) (0,1,3) (1,0,0) \dots (2,1,2) (2,1,3)$

如果计数再增加 1，则会得到 $(0, 0, 0)$ ，而迭代器将被设置为重新开始。

这个计数器是 NumPy 迭代器的核心内容。它的递增方式在迭代器的实现中起着重要的作用。因此，我们将在随后的章节中讨论计数器的实现。假设由这个计数器指定的数组当前位置是有效的，那么指向数组中当前值的指针可以通过用计数器的整数乘以在数组中定义的跨度值来得到，这个乘法运算的结果就是要加到数组第一个元素地址上的字节数。

例如，假设 `data` 是一个指向数组起始位置的指针，`counter` 是计数器（或者数坐标）数组，`strides` 是一组跨度值，那么下面的运算：

```
currptr = (char *)data;
for (i=0; i<N; i++) currptr += counter[i]*strides[i];
```

将把 `currptr` 设置为指向数组当前值（的第一个字节）。

事实上，当指针指向当前值时，我们并不需要每次都计算这个乘法，在实现中可以在记

录计数器的同时也记录指针，当计数器的第 i 个下标递增 1 时，`currptr` 将会增加 `strides[i]`。当第 i 个下标复位为 0 时，这与从当前下标值减去 $n_i - 1$ 是相同的，因此数组当前值的内存地址应该减去 $(n_i - 1) \times strides[i]$ 。

一般情况下，迭代器将负责维护表示数组位置的计数器以及指向当前值的指针。而对于数组中所有元素在内存中都是邻接的情况，维护这个计数器则是不必要的额外工作，因为当需要下一个值时，可以直接通过把数组当前值的内存地址加上数组中元素的大小来得到。

迭代器的终止

迭代器的另一个重要方面（尤其是当用于循环中时）就是判断迭代器何时完成以及如何发出终止信号。发出信号最普遍的方式就是把一个标志变量附加到迭代器上，在循环的每次迭代中都进行判断，如果没有元素可进行迭代，那么就设置这个标志。

另一种设置这个标志的方法就是查找在第一维的计数器中从 $n_1 - 1$ 到 0 的跃迁点。这种方法的问题在于它需要一个临时变量来保存最新的计数器值，因此对于邻接数组来说是不起作用的，因为在邻接数组中不会记录计数器。

不过，最简单的方法就是，如果给定了数组的大小，那么只需记住将要进行的迭代次数 $(n_1 \times n_2 \times \dots \times n_N)$ 。这个数值可以保存在迭代器结构中。然后，在每次迭代中，这个数值都将被递减。当这个数值为 0 时，迭代器将终止。

在 NumPy 中对这种倒计数的方法作了稍微的修改。为了把迭代的总次数作为一部分信息保存下来，以及保存一个到目前为止的迭代总次数的动态计数器，NumPy 使用了一个从 0 开始计数的整数计数器。当这个数值达到元素的总个数时，迭代过程将终止。

迭代器的构建

在创建迭代器时，必须计算和保存原始数组的大小。此外，整数计数器必须设置为 0，并且下标计数器必须初始化为 $(0, 0, \dots, 0)$ 。最后，NumPy 将判断迭代器是否可以基于简单的邻接内存操作并且设置一个标志来保存判断的结果。

为了加速当计数器的下标从 $n_1 - 1$ 返回到 0 时发生的“回溯步骤”，对数组的每一维都预先计算了 $(n_i - 1) \times strides[i]$ 并且保存下来。为了避免重复地计算 $n_i - 1$ ，对于这个结构同样进行了预先计算并且保存了下来。

虽然我们无法确信将这个很容易计算的数值保存下来是否会带来速度的提升，但在迭代

器中保存数组的维数信息仍然是非常有用的。同样，把跨度信息和原始数组的维数信息保存在迭代器中也是非常有用的。在迭代器中保存了数组的维数信息和跨度信息之后，就可以很容易地修改数组的解释，这可以通过修改这些位于迭代器中而不是原始数组中的值来实现。这在实现广播的时候尤其有用，广播能够使不同形状的数组看上去有着相同的形状（我们将在随后进行解释。）

最后，我们保存了一组预先计算的系数来简化在数组的整数计数器及其N下标之间的一一映射。例如，在数组中的每个项都可以用一个在0和 $n_1 \times \dots \times n_N - 1$ 之间的一个整数 k 或者一组整数来表示： (k_1, \dots, k_N) 。这个关系可以被定义为 $l_i = k$ 和：

$$k_i = \left\lfloor \frac{l_i}{\prod_{j=i+1}^N n_j} \right\rfloor$$

$$l_i = l_{i-1} \bmod \left(\prod_{j=i}^N n_j \right)$$

这个关系也可以表达为另一种形式：

$$k = \sum_{i=1}^N k_i \left(\prod_{j=i+1}^N n_j \right)$$

在前面等式括号中的项是预先计算并保存在迭代器中的数值，用作为数组的系数，以便于在两种表示 N 维下标的不同方式之间进行来回映射。

记录迭代器计数器

编写代码来记录 N 维下标计数器是非常简单的。我们必须区分迭代器在最后一个下标上简单地加1以及当某个下标达到相应维的最大长度时发生回卷这两种情况。当回卷发生时，有可能使得其他维的下标也发生回卷。因此，必须实现某种for循环以处理所有这些发生变化的下标。

一种简单的方法就是在计数器数组的末尾，也就是坐标数组的末尾，开始从后往前计数。在计数器数组的每一维下标位置上，代码将判断当前的下标是不是小于 $n_i - 1$ 。如果是的话，它将只是把1加到这个下标位置上，并且把strides[i]加到当前值指针的内存地址上。在这种情况下可以很容易地跳出这个for循环，并且完成计数器递增。

如果第 i 维下标的计数器大于等于 $n_i - 1$ ，那么就需要重新设置为 0 并且必须从当前值指针的内存地址上减去 $(n_i - 1) \times \text{strides}[i]$ （重新移回到这一维的起始位置）。在这种情况下，将会判断前一个坐标位置。

所有必要的信息都可以用一个结构来表示。这个结构的内容包括：

coords

每一维的当前下标， N

dims_m1

每一维的最高元素 $n_i - 1$ 的下标

strides

每一维中的跨度

backstrides

为了从每一维的末尾返回到起始位置而需要移动的数量： $(n_i - 1) \times \text{strides}[i]$

nd_m1

维数

currptr

指向在数组中当前位置内存的指针

然后，记录计数器的代码可以用 C 语言编写，如下所示：

```
for (i=it->nd_m1; i>=0; i--) {
    if (it->coords[i] < it->dims_m1[i]) {
        it->coords[i]++;
        it->dataptr += it->strides[i];
        break;
    } else {
        it->coords[i] = 0;
        it->dataptr -= it->backstrides[i];
    }
}
```

在这个实现中使用了 `break` 语句和一个 `for` 循环。我们也可以使用一个 `while` 语句和一个表示循环是否继续的标志：

```
done = 0;
i = it->nd_m1;
while (!done || i>=0) {
    if (it->coords[i] < it->dims_m1[i]) {
        it->coords[i]++;
        it->dataptr += it->strides[i];
    }
```

```

        done = 1;
    }
    else {
        it->coords[i] = 0;
        it->dataptr -= it->backstrides[i];
    }
    i--;
}

```

我选择 `for` 循环来进行实现部分原因是 `while` 循环看上去非常像 `for` 循环（初始化计数器，判断某个值，递减计数器）。通常我把 `while` 循环用于需要多个迭代下标的情况。不过，选择 `for` 循环的更重要的理由是因为这一小段实现计数器递增的代码可以写成一个宏以用在每个迭代循环中。我希望避免定义额外的变量 `done`。

迭代器的结构

我们现在来了解 NumPy 迭代器的整个结构。这个结构用 C 中的 `struct` 表示如下：

```

typedef struct {
    PyObject_HEAD
    int nd_m1;
    npy_intp index, size;
    npy_intp coords[NPY_MAXDIMS];
    npy_intp dims_m1[NPY_MAXDIMS];
    npy_intp strides[NPY_MAXDIMS];
    npy_intp backstrides[NPY_MAXDIMS];
    npy_intp factors[NPY_MAXDIMS];
    PyArrayObject *ao;
    char *dataptr;
    npy_bool contiguous;
} PyArrayIterObject;

```

在这个结构中的数组 (`coords`, `dims_m1`, `strides`, `backstrides` 和 `factors`) 都是定长数组，长度由 `NPY_MAXDIMS` 这个常量来控制。这样做是为了简化内存管理。不过，这种作法却限制了可以使用的维数。我们可以通过在创建迭代器时动态地分配所需的内存来轻松实现不同的处理；这样的修改不会改变迭代器的基本行为。

这些 `npy_intp` 类型的变量是整数，能够刚好容纳平台上的一个指针。`npy_bool` 是一个标志，它的值应该是 `TRUE` 或者 `FALSE`。在结构的 `PyObject_HEAD` 部分包含所有 Python 对象都必须包含的部分。

所有这些变量在前面都已经解释过了，但是为了清楚起见，我们给出了它们的具体含义：

`nd_m1`

比数组的维数小 1： $N-1$ 。

index 一个 C 数组的指针，指向当前元素。从索引 0 到 size - 1。从索引 0 指向数组的一个动态的计数器，表示迭代器当前指向数组中的哪个元素。这个计数器将从 0 变到 size - 1。

size 数组中所有元素的数量： $n_1 \times n_2 \times \dots \times n_N$ 。当迭代器为某个元素时，通过该值可以知道在 coords 中的哪维上设置坐标。通过类的成员方法 npy_array_get_size() 可以得到该值。

</div

联函数，因为它很复杂。不过，由于 NumPy 是用没有定义内联函数的 ANSI C 编写的，因此就使用了宏。最后，指向当前值的第一个字节的指针可以通过 PyArray_ITER_DATA(it) 来得到，这就避免了直接引用结构成员 dataptr（这使得将来可以对结构成员的名字进行修改）。

下面的代码给出了一个迭代器接口的示例，这段代码将计算 N 维数组中的最大值。我们假设这个数组叫作 ao，有着 double 类型的值，并且在内存中正确地对齐：

```
#include <float.h>
double *currval, maxval=-DBL_MAX;
PyObject *it;
it = PyArray_IterNew(ao);
while (PyArray_ITER_NOTDONE(it)) {
    currval = (double *)PyArray_ITER_DATA(it);
    if (*currval > maxval) maxval = *currval;
    PyArray_ITER_NEXT(it);
}
```

通过这段代码我们可以看出，为非邻接的 N 维数组构造一个循环是多么的容易。这段代码的简洁性同样说明了迭代器抽象的优美。注意这段代码与前面“迭代器的设计”一节中给出的简单迭代器的伪码非常相似。此外，考虑到这个代码可以用于任意维数的数组以及在每一维中任意的跨度，你会开始喜欢多维迭代器的漂亮性。

在处理邻接数组和非邻接数组时，使用迭代器的代码都是很快的。不过，处理邻接数组的最快循环仍然是像下面这样：

```
double *currval, maxval=-MAX_DOUBLE;
int size;
currval = (double *)PyArray_DATA(ao);
size = PyArray_SIZE(ao);
while (size--) {
    if (*currval > maxval) maxval = *currval;
    currval += 1;
}
```

NumPy 迭代器的真正好处在于它使得程序员可以编写出类似邻接数组的代码，并且运行快速且无需担心这些数组是否是邻接的。应该记住的是，如果在非邻接数组上强行使用邻接算法将带来性能损耗，因为非邻接数据必须被复制到另一个数组才能用于处理。

如果能够解决在目前 NumPy 迭代器接口中的一个遗留问题，那么在 NumPy 迭代器解决方案和最快的邻接情况解决方案之间的速度差异可以在很大程度上被消除。这个问题是 PyArray_ITER_NEXT 这个宏在循环中每次都会判断迭代器是否可以使用简化的邻接方法。在理想的情况下，这个判断应该只在循环外面进行一次即可。然而，这种类型的接口用 C 语言实现起来有些凌乱。它需要两个类似于 ITER_NEXT 的宏，以及两个不同的

`while` 循环。因此，在编写本章内容时，NumPy 并没有对这个功能进行改进。希望提高一下邻接数组运行速度的用户应该有足够的能力自己去编写这个简单的循环（彻底绕过迭代器）。

迭代器的使用

当某种抽象能够在不同的情况下简化编码的工作，或者能够在最初没有考虑到的方式上使用时，就证明了这个抽象有着很高的价值。对于 NumPy 迭代器对象来说，在这两种情况下都可以得到证明。我们只需简单地修改，就可以使最初简单的 NumPy 迭代器成为一个实现其他 NumPy 功能的组件，例如除去某一维的迭代（iteration over all but one dimension），以及在多个数组上同时进行迭代。此外，当我们需要快速地增加某些用于产生随机数的代码以及基本广播的复制算法的代码时，迭代器的存在及其扩展性将使得这些实现变得非常容易。

排除某一维的迭代

NumPy 中的一个常见任务就是通过把优化集中在可以使用简单跨度的单维循环上来获得速度提升。通常使用的是除了最后一维之外在所有维上进行迭代的策略。这是在 NumPy 的前身 Numeric 中引入的方法，用来实现数学功能。

在 NumPy 中，对 NumPy 迭代器稍做修改就能使其在任意代码中使用这个基本策略。修改后的迭代器按照下面的方式从构造器中返回：

```
it = PyArray_IterAllButAxis(array, &dim).
```

函数 `PyArray_IterAllButAxis` 的参数是一个 NumPy 数组和一个整数的地址，这个整数表示将要从迭代器中移除的维。这个整数是通过引用来传递的（`&` 运算符），因为如果将要移除的维指定为 `-1`，那么这个函数将自行判断从迭代中移除哪一维，并且把表示这个维的数值放在参数中。当输入的维是 `-1` 时，那么这个例程将选择最小非零跨度值的维。

另一种选择将要移除的维的方法是，选择有着最大元素数量的维。这个选择将把外部循环的迭代次数降至最低并且为快速的内部循环保留最多的元素。这个选择的问题在于从内存中提取信息或存入信息通常是通用处理算法的最慢部分。

因此，NumPy 做出这个选择的目的是要确保内部循环所处理的数据尽可能地被放在一起。在速度很关键的内部循环中，这样的数据更可能被快速访问到。

迭代器被修改为：

1. 将迭代大小除以将要移除的维的长度。
2. 将被选择的维的元素的数量设置为1（这样保存比总数小1的数组中的相应值将被设置为0）：`dims_m1[i] = 0`。
3. 将这个维在`backstrides`中相应位置上的值设置为0，这样在给定维中的计数器回卷到0时将永远不会改变数据指针。
4. 将邻接标志重新设置为0，因为现在要处理的数组在内存中将不会是邻接（每次迭代都必须跳过数组的整个维）。

函数将返回修改后的迭代器。它现在可以用于在前面使用迭代器的地方。在循环的每次迭代中，迭代器将指向数组所选择维的第一个元素。

多重迭代

NumPy中另一个常见的任务就是在几个数组中同时迭代。例如，数组加法的实现需要使用一个相连的迭代器以在两个数组中同时迭代，这样输出数组就是第一个数组中的每个元素乘以第二个数组中相应的元素，然后求总和。通常情况下，这可以通过使用不同的迭代器来实现，每个输入数组使用一个迭代器，并且输出数组使用另一个迭代器。

与此不同的是，NumPy提供了一个多重迭代器对象，可以简化对多个迭代器的同时处理。这个多重迭代器对象还能够自动处理NumPy的广播（Broadcasting）功能。广播是NumPy中的一个功能，它能够把不同形状的数组在逐元素的运算中放在一起使用。例如，广播能够使一个(4,1)形状的数组加到一个(3)形状的数组上，从而得到一个(4,3)形状的数组。同样，广播能够对一个(4,1)形状的数组，一个(3)形状的数组和一个(5,1,1)形状的数组进行运算，并得到(5,4,3)形状数组中元素的广播迭代（broadcasted iteration）。

广播的规则如下所示：

- 维数少的数组被视作为结果数组中的最后几维，因此所有的数组有着同样的维数。在新的数组中，初始的维将用1填充。
- 在最终的广播形状中，每一维的长度是所有数组中相应维之间的最大长度。
- 对于每一维来说，所有的输入都必须或者有相同数量的元素作为广播运算的结果，或者元素的数量为1。

- 在对某一维中只有单个元素的数组进行运算时，其行为好像这个元素在迭代中被复制到所有的位置上。事实上，这个元素是被“广播”到其他位置上的。

实现广播的关键算法只需对数组迭代器进行简单的修改。有了这些修改后，标准的迭代器循环可以以一种简单的方式用于结果计算。所需要的修改包括对迭代器（而不是底层的数组）形状的修改以及对 strides 和 backstrides 的修改。在迭代器中保存的形状被修改为匹配广播的形状。用于广播维数的 strides 和 backstrides 被修改为 0。在跨度值为 0 时，当这一维的下标递进时，标准的迭代器并不会把数据指针移动到内存中的元素。这就得到了我们想要的广播效果，实际上不需要复制内存。

在下面的代码中说明了多重迭代器对象的用法：

```

PyObject *multi;
PyObject *in1, *in2;
double *i1p, *i2p, *op;
/* 得到 in1 和 in2 (假设是一组 NPY_DOUBLE) */
/* 第一个参数是输入数组的数量；下一个 (可变数量的) 参数是数组对象 */
multi = PyArray_MultiNew(2, in1, in2);
/* 构造输出数组 */
out = PyArray_SimpleNew(PyArray_MultiIter_NDIM(multi),
                        PyArray_MultiIter_DIMS(multi),
                        NPY_DOUBLE);
op = PyArray_DATA(out);
while(PyArray_MultiIter_NOTDONE(multi)) {
    /* 得到每个数组中的当前值 (的指针) */
    i1p = PyArray_MultiIter_DATA(multi, 0);
    i2p = PyArray_MultiIter_DATA(multi, 1);
    /* 在这个元素上执行运算 */
    *op = *i1p + *i2p
    op += 1; /* 递进输出数组指针 */
    /* 递进所有的输入迭代器 */
    PyArray_MultiIter_NEXT(multi);
}

```

这段代码与标准的迭代器循环非常类似，除了多重迭代器将对输入迭代器进行调整以完成广播外，还同时递进所有其他的输入迭代器。这段代码把自动广播作为迭代器处理的一部分，因此把 (3, 1) 形状的数组增加到 (4) 形状的数组上将生产一个 (3, 4) 形状的输出数组。

奇闻轶事

在 NumPy 的代码库中大量地使用了 NumPy 迭代器来简化 N 维循环的构造。在有了迭代器之后，我就可以为更一般的（非邻接的）数组编写算法。由于处理非邻接数组的困难性，我通常使用的方法是把一个数组强行转换为邻接数组（在必要的时候可以创建一个新的拷贝），然后再使用一个容易的循环算法。NumPy 迭代器的出现使我能够编写出更

为通用的并且可读的代码，对于邻接数组来说只有非常小的速度损失。这个细微的缺陷已经被非邻接数组的内存需求降低而极大地抵消了。编写这样的循环所提升的效率足以证明 NumPy 迭代器的设计是正确的。

然而，这个抽象的真正闪光之处在于 NumPy 对广播的封装。在我使用多重迭代器来增强 NumPy 的随机数发生器以处理随机数发生器相关的参数组时，这个优势尤为突出。这个修改只花了两个小时，并且只是修改了几行代码而已。

NumPy 中的随机数发生器工具是由 Robert Kern 编写的。他并不熟悉最近添加的 C 形式的广播 API。因此，最初的实现要求所有用于指定随机数字的参数都为标量值（即，指数分布中的 n 值）。

这是一个不太好的限制。通常我们需要按照特定的分布得到一组随机数字，而数组的不同部分应该有不同的参数。例如，程序员可能需要按照指数分布得到一个随机数字的矩阵，其中每一行的数字应该用不同的 n 值来采样。为了能够使用一组参数，我们要做的修改就是使用多重迭代器循环（及其内置的广播功能），并且用随机采样值填充输出数组。

如果我们要把从一个数组复制到另一个数组的代码修改为与 NumPy 广播一致的操作方式，那么就可以再次使用迭代器。在前面，将一个数组中的内容复制到另一个数组使用的是标准的迭代器。惟一的形状判断是为了保证目标数组只被填充一次。如果目标数组耗尽了元素空间，那么迭代器将再次从头开始。很显然，这并不是我们想要的行为，因为它实现的是一种不同的“广播”（只要一个数组中的元素总数是另一个数组元素总数的整数倍，任意数组都可以被复制到其他的数组而无需考虑数组的形状）。根据这种复制形式所得到的数据复制与 NumPy 中其他地方使用的广播定义是不一致的。很明显，这里需要进行改变。再次指出，多重迭代器及其内置的迭代器广播概念是非常有用的抽象，因为它使我能够编写出快速完成复制过程（包括数组大小的判断）的代码，并且实际上只需非常少的几行新代码。

结论

NumPy 中的迭代器对象是用于简化编程的编码抽象示例之一。自从 2005 年构建以来，迭代器在编写处理通用 NumPy 数组的 N 维算法时一直是非常有用的，并且无需考虑这些数组在内存中是否是邻接的，或者数组实际上表示的是其他邻接内存块的 N 维切片。此外，对于迭代器的简单修改使得在实现 NumPy 中一些更为困难的思想时更为简单，例

如对广播操作的实现，或者对多维数组的处理。通过使用迭代器，我们可以更容易地处理多维数组，从而使得 NumPy 成为一个强大的科学计算库。

如优化循环（在除了最小跨度维（least-striding dimension）之外的其他所有维上进行循环）和广播机制。

迭代器是一个非常漂亮的抽象，因为它们在实现复杂算法时为程序员节约了大量的精力。这在 NumPy 中同样是正确的。在 NumPy 中对于标准数组迭代器的实现使得更容易编写和调试通用的代码，它也使得一些重要的（但却很难编写的）NumPy 功能被封装或者暴露出来，例如广播机制。

NASA 火星漫步者任务中的高可靠企业系统

Ronald Mak

你是否经常听到有人说“情人眼里出西施”？在这里，“情人”指的是 NASA 火星探测漫步者任务，它有着非常严格的需求，这个任务中的软件系统必须是实用的、可靠的以及稳定的。哦，这个软件还必须严格地按时交付——火星才不会管你延期的理由。当 NASA 在谈论满足“发射窗口（Launch Windows）”的时候，这意味着要满足多个方面！

本章描述的是协作式信息管理系统（Collaborative Information Portal, CIP）的设计和开发，CIP 是一个由 NASA 开发的庞大的企业信息系统，由任务管理人员、工程师以及世界各地的科学家们使用。

火星人不会容忍任何丑陋的软件。对于 CIP 来说，漂亮的概念并不只是局限于那些优雅的算法或者令人叹为观止的程序。CIP 的漂亮性体现在它是由一些熟练的编码人员构建起来的复杂软件，而这些编码人员只需知道如何把系统的各个组件组合起来。大型应用程序的漂亮性体现在多个方面，而小型程序却往往做不到这一点。这不仅是因为在大型程序中有着更多的必要性，而且还因为在大型程序中存在漂亮性的机率更大——在大型程序中经常要做一些小型程序无需去做的工作。我们将首先浏览一下 CIP 的整个基于 Java 的面向服务架构，然后通过对其中的一个服务进行案例分析，进而研究一些代码和一些使系统满足实用性、可靠性以及稳定性等需求的技术。

你可以想像到，用于NASA太空任务的软件必须是高度可靠的。因为这些任务都是很昂贵的，这些耗费多年的计划和数百万美元的任务不能因为有故障的程序而受到威胁。当然，在软件工作中最困难的部分就是，对那些在离地球数百万英里之外的航空器上运行的软件进行调试和修改。此外，基地系统（ground-base system）也必须是可靠的，没有人希望由于软件 bug 而导致任务操作中断或者丢失有价值的数据。

讨论关于这种软件的漂亮性有些不可思议。在一个多层次的面向服务架构中，服务被实现为驻留在服务器上的中间件。（我们在中间件中开发了共享的可重用组件，这极大地节约了开发时间）。这些中间件把客户端程序与后端数据源解耦开来；换句话说，应用程序并不需要知道它所需的数据是如何存储的，以及存储在什么地方。当所有的中间件服务器都正常工作时，这个企业系统的终端用户甚至不会意识到他们的客户端程序正在发出远程服务请求。当中间件运行得很顺畅时，用户会认为他们正在直接访问数据源，并且所有的数据处理都是在他们的工作站或者笔记本上进行的。因此，中间件越成功，它们就会变得越透明，漂亮的中间件应该是完全不可见的！

任务与 CIP

火星探测漫步者任务，或者简称为MER，它的主要目标是探索在火星表面上是否曾经存在过液态水。在2003年的6月和7月，NASA向火星发射了两个相同的漫步者，即地质机器人。在2004年1月，在度过了7个月的旅途后，它们在火星的背面登陆了。

每个漫步者都配有太阳能动力以在火星表面自行驱动。每个漫步者都配备了一些科学仪器，例如安装在机械臂终端的光谱仪。在这些机械臂上有一个钻头和一个显微影像仪，用来分析岩石表面下的成分。每个漫步者都安装了数台相机和天线，用来把数据和影像发送回地球（参见图 20-1）。

在无人驾驶的NASA任务中包含了硬件和软件。在每台火星漫步者上都有不同的软件包来独立地控制操作，并且对远在加利福尼亚州 Pasadena 附近的 NASA 喷气推进实验室（Jet Propulsion Laboratory, JPL）的任务控制中心发出的远程命令作出响应。在任务控制中心的基地软件包使得管理人员、工程师和科学家能够下载和分析由漫步者发回来的数据，并且设计和开发新的命令序列发送给漫步者，以及与其他人员进行协作。

在加利福尼亚州 Mountain View 附近的 NASA Ames 研究中心，我们为 MER 任务设计并开发了协作式信息管理系统（CIP）。这个项目团队包括 10 名软件工程师和计算机科学家。另外还有 9 名包括项目经理和技术支持工程师在内的团队成员，主要负责 QA、软件系统构建、硬件配置和 bug 跟踪等任务。

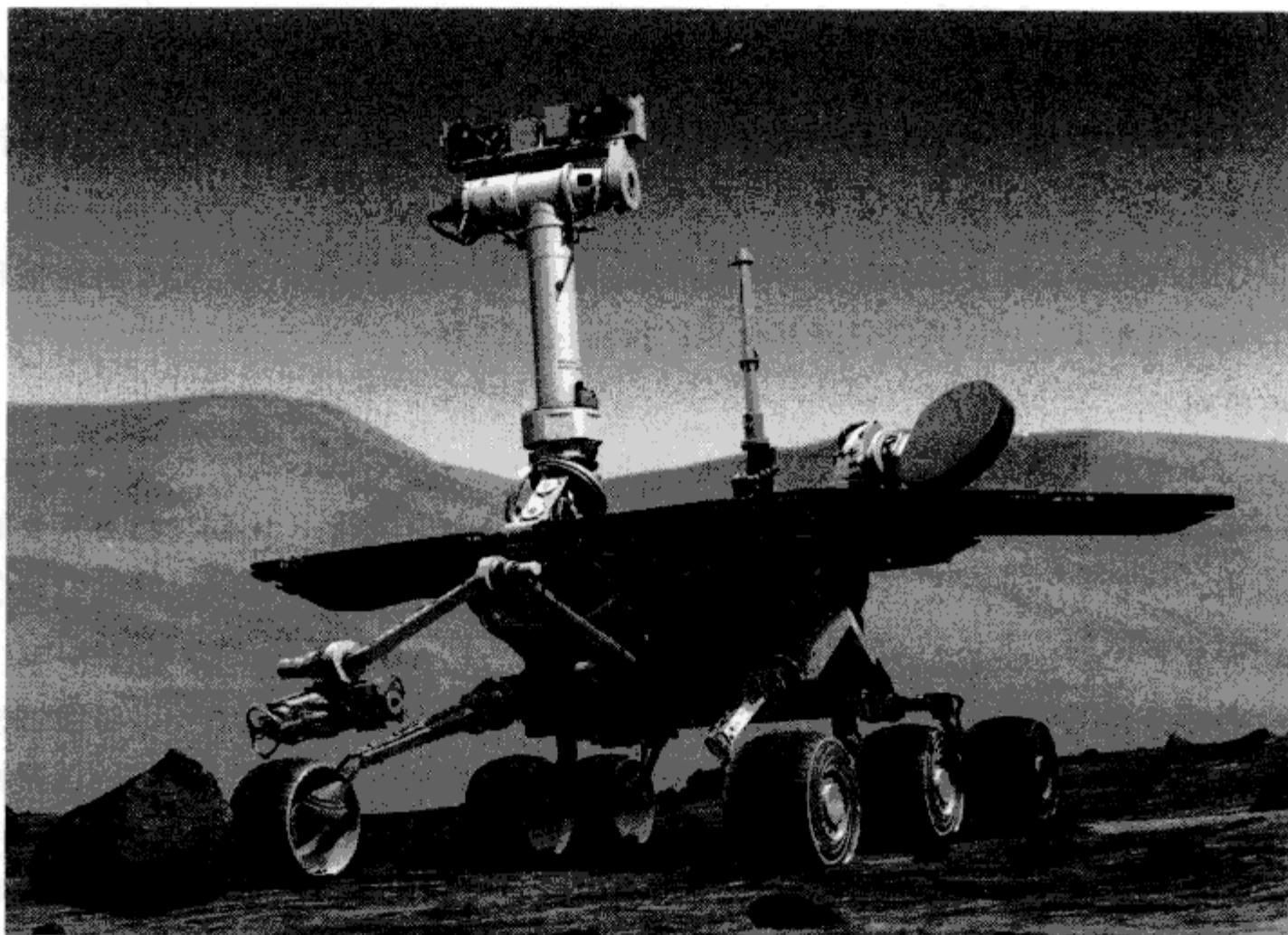


图 20-1：火星漫步者（JPL 提供的免费图像）

任务需求

CIP 被设计用来满足 MER 任务的三个主要需求。通过满足这些需求，CIP 在任务人员之间提供了重要的“环境感知（situational awareness）”功能：

时间管理

在任何大型的复杂任务中，使所有工作人员保持同步对于任务的成功来说是非常重要的，此外，在 MER 中还给出了一些特殊的时间管理需求。由于任务中的工作人员分布在世界各地，因此 CIP 需要以多个地方的时区来显示时间。而且由于漫步者是在火星的背面登陆的，因此还有两个火星的时区。

最初，任务是按照火星时间来运行的，这意味着所有会议和活动（例如从火星上下载数据）的时间都是参照某台漫步者所处的火星时区来制定的。火星上的一天比地球上的一天大概长 40 分钟，并且与地球时间非常相似，因此任务中的工作人员每天都推迟这段时间，从而与他们使用地球时间的家人和同事保持一致。这些都使得 CIP 的时间管理功能变得更为重要。

人员管理

随着在两个漫步者团队（每个漫步者都配有一个团队）之间人员的更迭，对每个成员变动进行跟踪变成了另一项重要的任务。CIP管理着一个人员名册，并且记录了每个成员的工作地点，工作时间以及工作角色。

CIP在任务人员之间还实现了一些协作。他们可以发出广播消息，共享对数据和影像的分析，上传报告以及对其他人的报告进行注释。

数据管理

对于NASA的每个行星任务和深太空任务（deep space mission）来说，从宇宙的遥远之处获得数据和影像都是非常重要的，这里也不例外，CIP同样起着重要的作用。在地面上有一组天线用于接收从火星漫步者发出的数据和影像，并且当这些数据和影像到达地球后，将被传输到JPL，并且在任务文件服务器上进行处理和存储。

当任务管理者发布已处理的数据和影像文件时，CIP会生成元数据，以将数据按照不同的标准进行分类，这些标准包括哪一台漫步者仪器生成的数据，哪一台照相机拍摄的影像，在什么环境下，使用何种配置，以及时间和地点等。然后，CIP用户可以根据这些标准来搜索数据和影像，并且通过互联网把这些文件从任务的文件服务器下载到个人笔记本和工作站上。

在CIP中还实现了数据安全。例如，根据用户角色的不同（以及她是否是美国公民），一些特定的数据和影像是禁止访问的。

系统架构

一个企业系统代码的漂亮性部分取决于系统的架构，即代码组织的方式。架构不仅仅是一种美学。在大型应用程序中，架构确定了软件组件之间的交互方式，并将对整个系统的可靠性产生影响。

我们使用了一个三层的面向服务架构(service-oriented architecture, SOA)来实现CIP。我们坚持遵循工业标准和最佳实践，在所有可能的地方都使用了标准商用(commercial off-the-shelf, COST)软件。大多数程序都是用Java编写的，我们使用了Java 2企业版本(J2EE)标准(参见图20-2)。

在客户层包含了大部分基于GUI的独立Java应用程序，它们是用Swing组件和一些基于网页的程序开发的。符合J2EE标准的应用程序服务器将在中间件中运行，并且提供用于响应客户端应用程序请求的所有服务。我们使用了企业Java Bean(EJB)来实现这些服务。数据层包括数组源和数据工具软件。这些工具软件同样是用Java编写的，用来

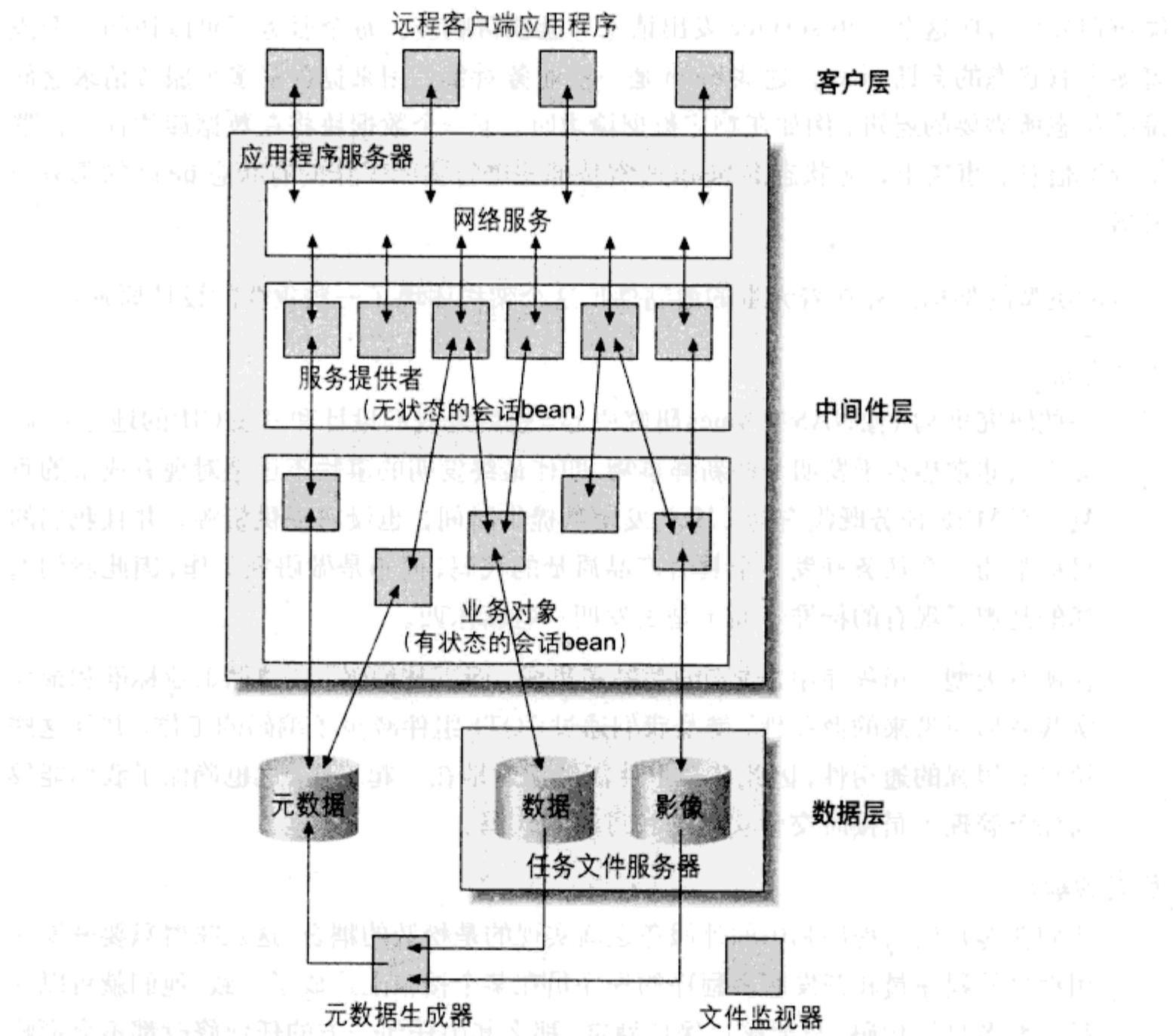


图 20-2: CIP 的三层面向服务架构

监视存放已处理的数据和影像的文件服务器。只要任务管理人员一发布新的数据和影像文件，这些工具软件将立刻生成这些文件的元数据。

通过使用基于 J2EE 的 SOA，我们可以在大型企业应用程序设计中合适的地方使用这些定义良好的 Java Bean (以及其他一些组件)。无状态的会话 bean 在处理服务请求时不会保存从一个请求到下一个请求之间的任何状态。而有状态的会话 bean 将会为客户端维护状态信息，并且通常还会管理 bean 在某个数据存储中读取或者写入的持久信息。在开发大型的复杂应用程序时，如果在任意的设计情形中都能够有多种选择，那么将会是很有用的。

在中间件中，我们为每个服务都实现了一个无状态的会话 bean 以作为服务的提供者。这里使用的是外观 (facade) 设计模式，我们通过这种模式生成了一个 web service，客户

端应用程序将向这个 web service 发出请求并且获得响应。每个服务还可以访问一个或者多个有状态的会话 bean，这些 bean 是一些业务对象，用来提供在多个服务请求之间维护状态所需要的逻辑，例如在响应数据请求时，下一个数据块将在数据库的什么位置上读取信息。事实上，无状态的 bean 通常是那些进行实际工作的有状态 bean 的服务分发器。

在这种类型的架构中存在着大量的漂亮性！这个架构体现了一些重要的设计原则：

基于标准

一些研究机构（像 NASA Ames 研究中心，也就是我们设计和开发 CIP 的地方），研究人员非常热衷于发明一些新鲜事物，即使最终发明的事物不过是对现有成果的重复。而 MER 任务既没有为 CIP 开发团队提供时间，也没有提供资源，并且我们的目标是为这个任务开发一个具有产品质量的代码，而不是做研究工作，因此我们选择的是遵循现有的标准，而不是去发明一些新东西。

在所有大型应用程序中，成功的关键是集成，而不是编码。在遵循工业标准和最佳实践背后所带来的漂亮性，就是我们通过 COTS 组件减少了编码的工作，并且这些接口有很强的通用性，因此各个组件能够很好地在一起工作。这也确保了我们能够向任务管理人员按时交付实用的和可靠的代码。

松散的耦合

我们在客户应用程序和中间件服务之间实现的是松散的耦合。这意味着只要开发应用程序的程序员和开发服务程序的程序员在某个接口上达成了一致，他们就可以并行开发各自的代码。只要接口保持稳定，那么其中任何一方的任何修改都不会影响到另一方。松散的耦合是使我们能够按时交付大型的多层 SOA 应用程序另一个主要的因素。

语言的独立性

客户端应用程序和中间件服务使用 web service 进行彼此之间的通信。web service 协议是一个独立于语言的工业标准。虽然大多数的 CIP 客户应用程序都是用 Java 语言编写的，但这个中间件同样可以为一些用 C++ 或者 C# 语言编写的应用程序提供服务。如果我们能够使服务和 Java 客户端一起工作，那么就可以很容易地将这个服务与另一个用其他语言编写的并且支持 web service 的客户端放在一起工作。这极大地扩展了 CIP 的可用性和有效性，而消耗的时间和资源是非常少的。

模块化

随着应用程序代码量的增长，模块化的重要性将呈指数增长。在 CIP 中，每个服务都是完备的中间件组件，独立于其他的服务。如果某个服务需要与其他服务联合工作，它可以向其他服务发出服务请求，就好像它是一个客户程序。这使得我们能够

独立地创建服务，并且在我们的开发工作中增加另一层并行。模块化的服务是漂亮的，在成功开发的大型 SOA 应用程序中经常可以看到。

在客户层，应用程序通常混合使用多个服务，或者把多个服务的返回结果组合在一起，或者使用某个服务的结果作为请求传递给另一个服务。

可伸缩性

在任务控制中心发布已处理的数据和影像文件时，尤其是当一台漫游者获得了有趣的发现之后，对 CIP 的使用将出现峰值。我们必须确保 CIP 的中间件能够处理这样的峰值，尤其是当用户正在很焦急地等待下载和查看最新文件的时候。如果在这些时候系统的速度变慢，或者出现更糟糕的情况，系统崩溃，都会是令人无法忍受和难以面对的。

J2EE 架构的漂亮性之一在于它处理可伸缩性的方式。在应用服务器中将维护一个 bean 池，并且根据需求能够自动创建多个实例的无状态会话 bean 服务提供者。这是一个很重要的“免费” J2EE 功能，中间件服务的开发人员非常乐于接受这个功能。

可靠性

作为一个经过工业界严格检验的标准，J2EE 的架构被证明是非常可靠的。我们避免了超越设计范围的情况。因此，在经过两年的运行之后，CIP 创造了正常运行时间超过 99.9% 的记录。

我们超越了 J2EE 本身所提供的可靠性。正如你将在随后的案例分析中看到的，我们在服务中进行了一些额外的改进以进一步提升可靠性。

案例分析：流服务

你已经从架构中看到了 CIP 的一些漂亮性。现在我们来关注其中的一个中间件服务——流服务——并将其作为案例进行分析，我们将研究一些能够使任务满足严格的实用性、可靠性以及稳定性等需求的技术。你将看到这些技术并不是特别奇特的；系统的漂亮性在于在什么地方使用这样的技术。

实用性

MER 任务的数据管理需求之一就是允许用户把位于 JPL 任务文件服务器上的数据和影像文件下载到他们的个人工作站或者笔记本上。正如前面所谈及的，CIP 的数据层工具生成元数据，使用户可以基于不同的标准来检索他们想要的文件。同样，用户也可以把包含他们分析的报告上传到服务器。

CIP 的流服务用来执行文件的下载和上传工作。我们之所以把这个服务称之为流服务，是因为这个服务的功能是使数据能够通过互联网在JPL的任务文件服务器和用户本地电脑之间安全地流动。这个服务使用了 web service 协议，因此客户端应用程序可以用任何一种支持这种协议的语言来编写，并且这些应用程序可以自由地设计适合它们自己的 GUI。

服务架构

和所有其他的中间服务一样，在流服务中使用了 web service 来接收客户端的请求并返回响应。每个请求最初是由流服务提供者来填充内容，这个提供者是用一个无状态的会话 bean 来实现的。服务提供者创建了一个利用有状态的会话 bean 实现的文件读取器来进行实际的工作，从而把被请求的文件内容下载到客户端。相应地，服务提供者还创建了一个文件写入器，这同样是用有状态的会话 bean 来实现的，用于上传文件内容（参见图 20-3）。

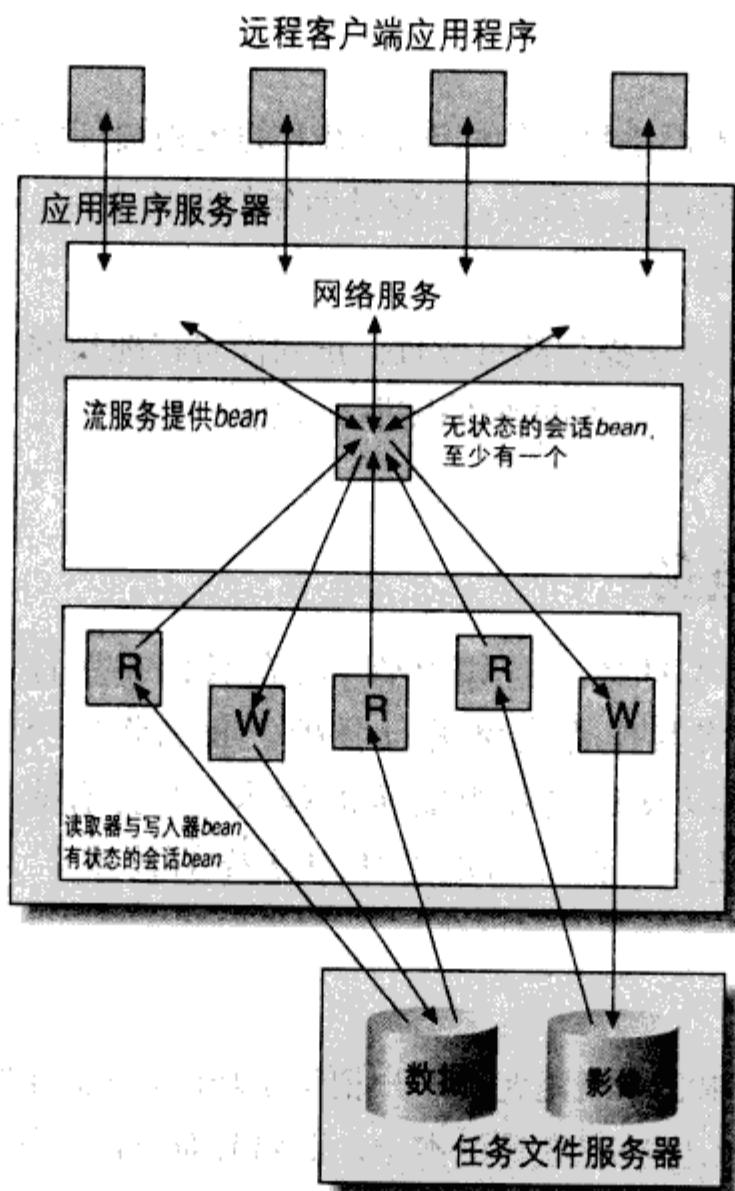


图 20-3：流服务的架构

在任意时刻，多个用户都可以同时下载或者上传文件，并且任何单个用户都可以同时进行多个下载或者多个上传操作。因此，在中间件中可能同时存在着多个文件读取器和文件写入器。单个无状态的流服务提供者bean能够处理所有这些请求，除非负载变得很沉重，此时应用程序服务器可以创建更多的提供者bean。

为什么每个文件读取器和文件写入器都必须是一个有状态的会话bean？这是因为在响应来自客户端的“读取数据块”或者“写入数据块”的请求时，除非这个文件很小，否则流服务将对这个文件的内容进行分块并且每次传输一块内容。（下载块的大小是在中间件服务器上配置的，而上传块的大小则是由客户端应用程序来指定的）。在一个请求到下一个请求之间，有状态的bean将记录在任务文件服务器上打开的源文件或者目标文件，以及在文件中下一个将要进行读取块或者写入块的位置。

虽然这是一个很简单的架构，但它能够非常有效地处理来自多个用户的同时下载请求。图20-4给出了从任务文件服务器上下载一个文件到用户本地机器上的事件序列。

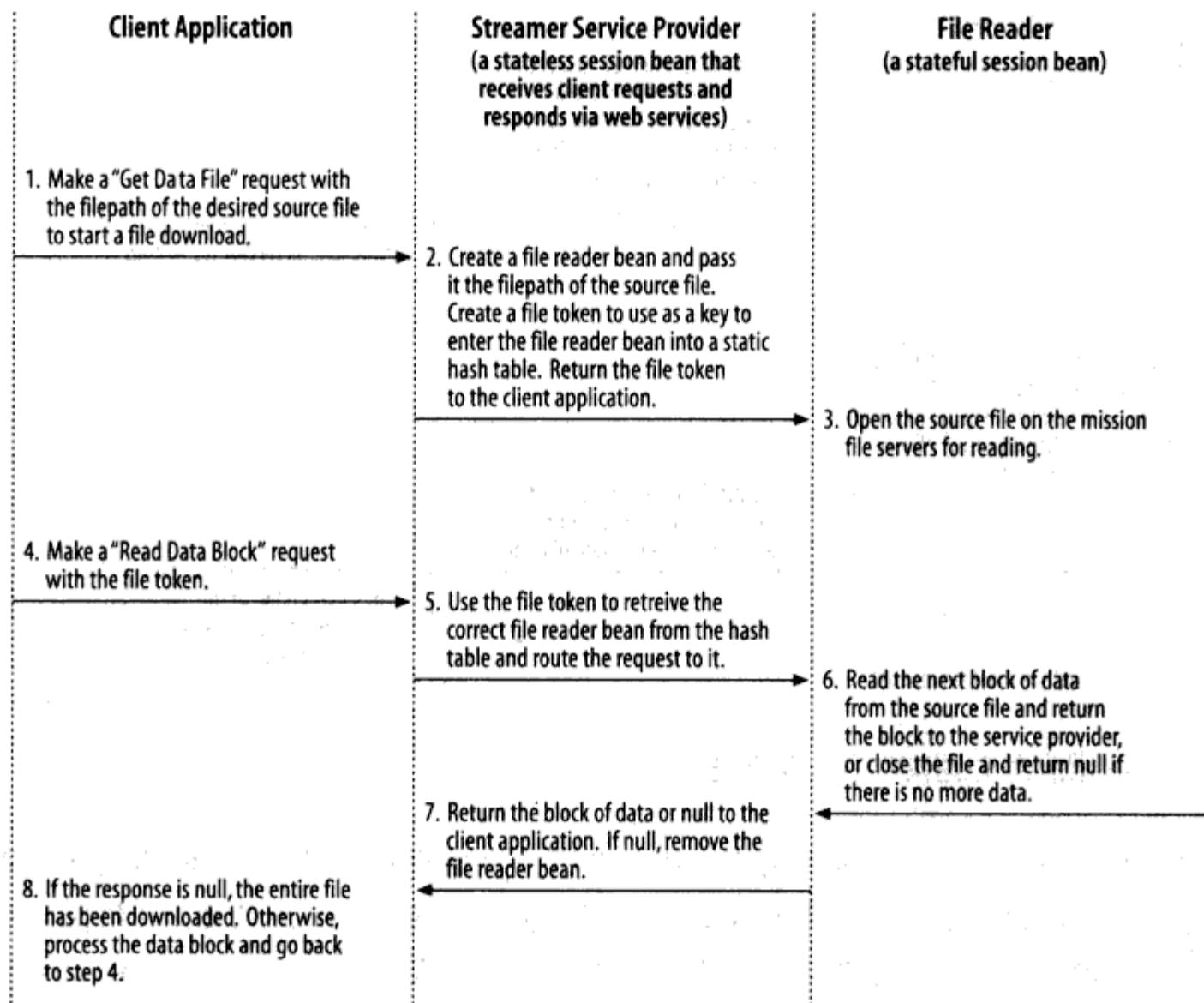


图20-4：两层服务处理文件读取的过程

注意，流服务提供者在两个服务请求之间并不会维护任何状态。它的作用是作为一个快速的服务分发器，把工作包装起来并且发送给有状态的文件读取器bean。由于它并不需要记录请求或者维护状态，因此它可以处理来自多个客户应用程序的混合请求。每个文件读取器bean都为单个客户应用程序维护了状态信息（到什么位置去读取下一块数据），应用程序发出多个“读取数据块”的请求来下载一个完整的文件。这个架构使得流服务能够同时把多个文件下载到多个客户端，而同时还能为所有这些请求提供满足需要的吞吐量。

将文件从用户的本地机器上传到任务文件服务器的事件序列也是很清楚的。如图20-5所示。

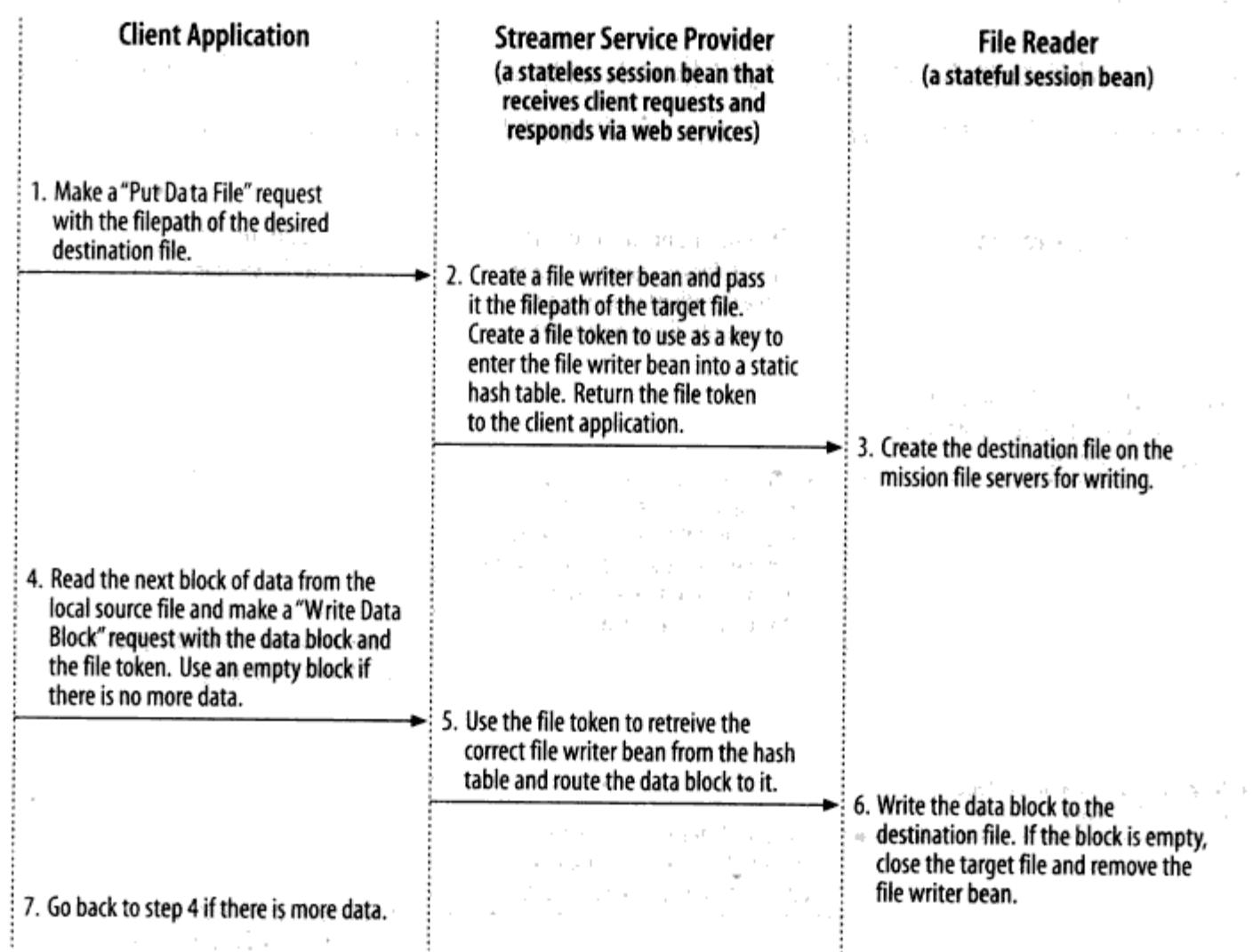


图 20-5：两层服务处理文件写入的过程

除了文件标识之外，在每个客户请求中还包含了一个用户标识，但在上面的表格中并没有给出。当客户端应用程序向中间件的用户管理服务发送了一个成功登录（包括用户名和密码）请求时，它将首先获得一个用户标识，这样就认证了用户。在用户标识中包含了识别特定用户会话的信息，包括用户的角色。它使得流服务能够验证某个请求是否来自于一个合法的用户。此外还将通过检测用户的角色来判断她是否有下载某个文件的权

限。例如，MER任务不允许国外（非美国）用户下载某些特定的文件，在CIP中将遵守所有这些安全限制。

可靠性

可靠的代码能够持续良好地执行而不会遇到问题。程序很少崩溃。你可以想像到，在火星漫步者上的代码必须是极其可靠的，因为要进行现场的维护将是非常困难的事情。不过，MER任务也希望任务控制中心所使用的基地软件同样是可靠的。一旦任务开始进行，没有人希望因软件问题而中断整个操作。

正如前面所提到的，CIP项目采用了几种方法来确保系统内在的可靠性：

- 遵循工业标准和最佳实践，包括J2EE。
- 在任何可能的地方都使用那些已被验证过的COTS软件，包括来自于权威中间件制造商的商业应用程序服务器。
- 使用带有模块化服务的面向服务架构。
- 实现简单、直接的中间件服务。

我们通过额外的技术来进一步提高可靠性：服务日志和监测。这些功能不但在调试小型程序时是很有用的，而且在跟踪大型应用程序的运行行为上也能够起到重要的作用。

日志

在开发过程中，我们使用了开源的Apache Log4J Java包来记录在中间件服务中发生的每个事件。这对于开发中的调试工作来说是非常有用的。通过日志，我们可以编写出更为可靠的代码。当出现bug时，日志可以告诉我们在出现bug之前进行的是什么操作，这对于我们修正bug非常有帮助。

我们最初试图减少日志的数量，而只是记录在CIP开始运行之前发生的严重错误消息。但我们最终保留下了大部分的日志信息，由于它们对整体性能的影响几乎可以忽略不计。随后，我们发现这些日志能够为我们提供大量有用的信息，不仅包括在每个服务上所运行的操作的信息，而且还包括客户程序如何使用服务的信息。通过分析这些日志（我们称之为“日志发掘(log mining)”), 我们能够根据经验数据来调节服务以获得更好的性能（请参见本章后面的“动态重配置”一节）。

下面是一段来自于流服务提供者bean的代码，这段代码给出了我们如何记录文件下载的过程。方法getDataFile()用来处理每个来自于客户应用程序的“获取数据文件”的

请求（通过 web service）。这个方法将及时记录下这个请求（第 15~17 行），包括请求者的用户 ID 和所请求的源文件的路径。

```
1  public class StreamerServiceBean implements SessionBean
2  {
3      static {
4          Globals.loadResources("Streamer ");
5      }
6
7      private static Hashtable readerTable = new Hashtable();
8      private static Hashtable writerTable = new Hashtable();
9
10     private static BeanCacheStats cacheStats = Globals.queryStats;
11
12     public FileToken getDataFile(AccessToken accessToken, String filePath)
13         throws MiddlewareException
14     {
15         Globals.StreamerLogger.info(accessToken.userId() +
16             ": Streamer.getDataFile(\"" +
17             + filePath + "\")");
18         long startTime = System.currentTimeMillis();
19
20         UserSessionObject.validateToken(accessToken);
21         FileToken fileToken = doFileDownload(accessToken, filePath);
22         cacheStats.incrementTotalServerResponseTime(startTime);
23         return fileToken;
24     }
25 }
```

在方法 doFileDownload() 中将创建一个新的文件标识（第 30 行）和一个文件读取器（第 41 行），然后调用读取器 bean 的 getDataFile() 方法（第 42 行）。cacheStats 这个域用于运行时的监测，将在随后进行介绍。

```
26     private static FileToken doFileDownload (AccessToken accessToken,
27                                         String filePath)
28         throws MiddlewareException
29     {
30         FileToken fileToken = new FileToken(accessToken, filePath);
31         String key        = fileToken.getKey();
32
33         FileReaderLocal reader = null;
34         synchronized (readerTable) {
35             reader = (FileReaderLocal) readerTable.get(key);
36         }
37
38         // 创建一个文件读取器，开始下载。
39         if (reader == null) {
40             try {
41                 reader = registerNewReader(key);
42                 reader.getDataFile(filePath);
43             }
44             return fileToken;
45         }
46     }
47 }
```

```

45         }
46     } catch(Exception ex) {
47         Globals.StreamerLogger.warn("Streamer.doFileDownload(" +
48             + filePath + "): " +
49             ex.getMessage());
50         cacheStats.incrementFileErrorCount();
51         removeReader(key, reader);
52         throw new MiddlewareException(ex);
53     }
54 }
55 else {
56     throw new MiddlewareException("File already being downloaded: " +
57             filePath);
58 }
59 }
60

```

`readDataBlock()`方法处理每个来自于客户程序的“读取数据块”请求。它将查找正确的文件读取器bean (第71行) 并且调用读取器bean的`readDataBlock()`方法 (第79行)。在源文件的末尾, 将删除文件读取器bean (第91行)。

```

61     public DataBlock readDataBlock(AccessToken accessToken, FileToken fileToken)
62         throws MiddlewareException
63     {
64         long startTime = System.currentTimeMillis();
65         UserSessionObject.validateToken(accessToken);
66
67         String key = fileToken.getKey();
68
69         FileReaderLocal reader = null;
70         synchronized (readerTable) {
71             reader = (FileReaderLocal) readerTable.get(key);
72         }
73
74         // 使用读取器bean来下载下一个数据块
75         if (reader != null) {
76             DataBlock block = null;
77
78             try {
79                 block = reader.readDataBlock();
80             }
81             catch(MiddlewareException ex) {
82                 Globals.StreamerLogger.error("Streamer.readDataBlock(" +
83                     + key + ")", ex);
84                 cacheStats.incrementFileErrorCount();
85                 removeReader(key, reader);
86                 throw ex;
87             }
88
89             // 是否到了文件末尾?
90             if (block == null) {
91                 removeReader(key, reader);
92             }

```

```

93         cacheStats.incrementTotalServerResponseTime(startTime);
94         return block;
95     }
96     else {
97         throw new MiddlewareException(
98             "Download source file not opened: " +
99             fileToken.getFilePath());
100    }
101 }
102 }
103

```

`register.NewReader()`和`remove.NewReader()`这两个方法分别用来创建和销毁有状态的文件读取器 bean，如下所示：

```

104     private static FileReaderLocal register.NewReader(String key)
105         throws Exception
106     {
107         Context context = MiddlewareUtility.getInitialContext();
108         Object queryRef = context.lookup("FileReaderLocal");
109
110         // 创建读取器服务 bean 并且注册。
111         FileReaderLocalHome home = (FileReaderLocalHome)
112             PortableRemoteObject.narrow(queryRef, FileReaderLocalHome.class);
113         FileReaderLocal reader = home.create();
114
115         synchronized (readerTable) {
116             readerTable.put(key, reader);
117         }
118
119         return reader;
120     }
121
122     private static void remove.NewReader(String key, FileReaderLocal reader)
123     {
124         synchronized (readerTable) {
125             readerTable.remove(key);
126         }
127
128         if (reader != null) {
129             try {
130                 reader.remove();
131             }
132             catch(javax.ejb.NoSuchObjectLocalException ex) {
133                 // 省略以下代码
134             }
135             catch(Exception ex) {
136                 Globals.StreamerLogger.error("Streamer.remove.NewReader(" +
137                     + key + ")", ex);
138                 cacheStats.incrementFileErrorCount();
139             }
140         }
141     }
142 }

```

以下是来自于文件读取器bean的代码。cacheStats和fileStats这两个域用于运行时的监测，将在随后进行介绍。getDataFile()方法记录文件下载的开始事件（第160~161行）

```
143 public class FileReaderBean implements SessionBean
144 {
145     private static final String FILE = "file";
146
147     private transient static BeanCacheStats cacheStats = Globals.queryStats;
148     private transient static FileStats      fileStats = Globals.fileStats;
149
150     private transient int          totalSize;
151     private transient String      type;
152     private transient String      name;
153     private transient FileInputStream fileInputStream;
154     private transient BufferedInputStream inputStream;
155     private transient boolean    sawEnd;
156
157     public void getDataFile(String filePath)
158         throws MiddlewareException
159     {
160         Globals.StreamerLogger.debug("Begin download of file "
161             + filePath + "'");
162         this.type = FILE;
163         this.name = filePath;
164         this.sawEnd = false;
165
166         try {
167
168             // 从数据文件创建输入流
169             fileInputStream = new FileInputStream(new File(filePath));
170             inputStream = new BufferedInputStream(fileInputStream);
171
172             fileStats.startDownload(this, FILE, name);
173         }
174         catch(Exception ex) {
175             close();
176             throw new MiddlewareException(ex);
177         }
178     }
179 }
```

readDataBlock()方法将从源文件中读取每个数据块。当它读完了整个源文件时，它将记下任务的完成事件（第191~193行）：

```
180     public DataBlock readDataBlock()
181         throws MiddlewareException
182     {
183         byte buffer[] = new byte[Globals.streamerBlockSize];
184
185         try {
186             int size = inputStream.read(buffer);
```

```

187
188     if (size == -1) {
189         close();
190
191         Globals.StreamerLogger.debug("Completed download of " +
192             type + " '" + name + "' : " +
193             totalSize + " bytes");
194
195         cacheStats.incrementFileDownloadedCount();
196         cacheStats.incrementFileByteCount(totalSize);
197         fileStats.endDownload(this, totalSize);
198
199         sawEnd = true;
200         return null;
201     }
202     else {
203         DataBlock block = new DataBlock(size, buffer);
204         totalSize += size;
205         return block;
206     }
207 }
208 catch(Exception ex) {
209     close();
210     throw new MiddlewareException(ex);
211 }
212 }
213 }
```

以下是流服务日志中的一些内容：

```

2004-12-21 19:17:43,320 INFO : jqpublic:
Streamer.getDataFile('/surface/tactical/sol/120/jpeg/
1P138831013ETH2809P2845L2M1.JPG')
2004-12-21 19:17:43,324 DEBUG: Begin download of file '/surface/tactical/sol/120/
jpeg/1P138831013ETH2809P2845L2M1.JPG'
2004-12-21 19:17:44,584 DEBUG: Completed download of file '/surface/tactical/
sol/120/
jpeg/1P138831013ETH2809P2845L2M1.JPG': 1876 bytes
```

在图20-6中给出了一张有用的信息图表，其中包括我们从日志发掘中得到的信息。这张图给出了在任务期间几个月的时期内下载数量的趋势(文件的数量以及下载的字节数)，从这张图中可以看出当某台漫步者获得有趣的发现时出现的下载峰值。

监测

日志使我们能够通过研究服务在一段时期内的操作来分析服务的性能。与日志方法不同的是，运行时监测在定位问题及其成因中是最有帮助的，它能有助于我们看到服务当前的运行情况。它使我们能够进行动态调节以改善性能或者防止任何潜在的问题。正如在前面提到的，对于任何大型应用程序来说，监测运行时的行为对于应用程序的成功来说是非常重要的。

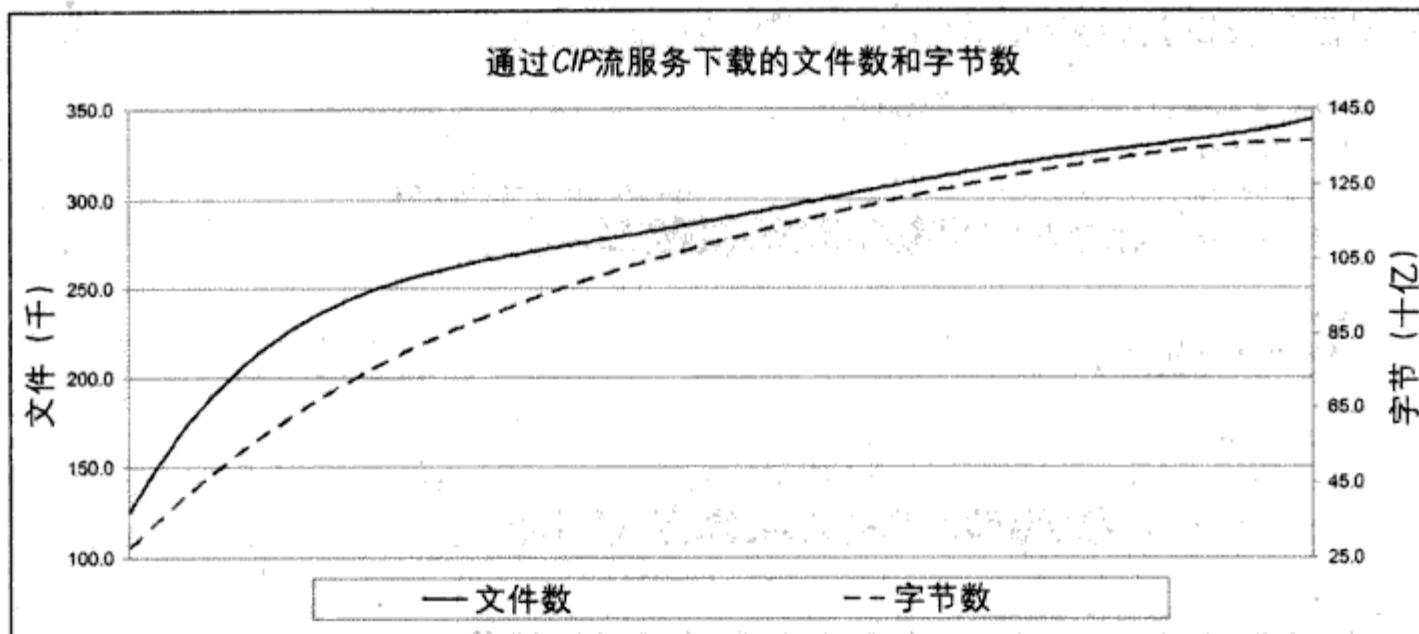


图 20-6：通过“发掘”CIP 流服务日志而得到的图

在前面给出的代码中包含了对在全局静态对象中保存的性能数据进行更新的语句，这些全局对象是通过 `cacheStats` 和 `fileStats` 这两个域来引用的。中间件监测服务将根据请求来查询这个性能数据。虽然并没有给出这些域所引用的全局对象，但是你应该能够想像出它们所包含的内容。关键是收集有用的运行时性能数据并不是件复杂的任务。

我们把 CIP 中间件监测工具编写为一个客户端应用程序，这个程序将定期地把请求发送给中间件监测服务来获得当前的性能数据。图 20-7 给出了这个工具中 `Statistics` 标签页的截图，其中显示了通过流服务已经被下载的和上传的文件数量以及字节数量、已经发生的文件错误的数量（例如客户端应用程序指定了无效文件名的错误）以及其他运行时统计数据。

在流服务提供者 bean 中的 `doFileDownload()` 和 `readDataBlock()` 这两个方法都会更新全局文件错误的计数（在“日志”一节代码中的第 50 行和第 84 行）。`getDataFile()` 和 `readDataBlock()` 这两个方法将增加全局的总服务响应时间（第 22 行和第 94 行）。正如在图 20-7 中所看到的，中间件监测工具在“Total Server Response”项中显示平均响应时间。

文件读取器 bean 的 `getDataFile()` 方法将记录每个文件下载的起始时刻（第 172 行）。`readDataBlock()` 方法将增加全局的总文件数和字节数量（第 195 行和第 196 行）并且记录下载完成的时刻（第 197 行）。在图 20-8 中给出监测工具的“File”标签页的截图，显示的是当前和最近的文件下载和上传等行为。

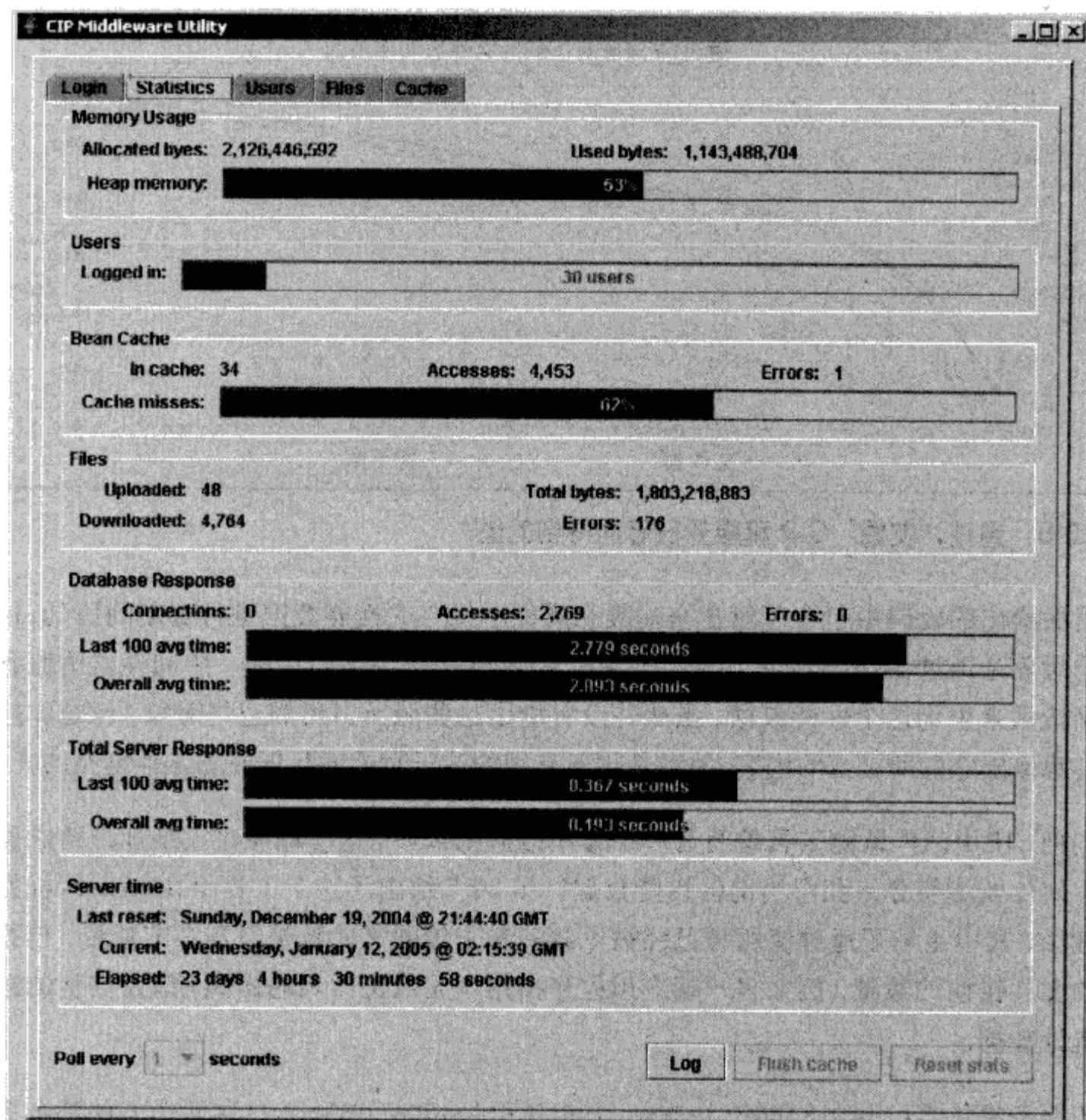


图 20-7：中间件监测工具中 Statistics 标签页的截图

稳定性

变动是不可避免的，而漂亮的代码即使在投入运行之后仍然能够优雅地处理变动。我们采取了一组方法来保证 CIP 是稳定的并且能够处理运行参数中的变动。

- 我们避免在中间件服务中使用硬编码的参数。
- 我们尽可能地使得由于对已投入运行的中间件服务进行修改而造成的客户端应用程序中断降至最低。

CIP Middleware Utility

File Options

Direction Size Start time End time Name

Direction	Size	Start time	End time	Name
Download	144,545	Wed 2005.01.12 @ 01:02:02 GMT	Wed 2005.01.12 @ 01:02:06 GMT	/oss/merb/ops/ops/surf ace/tactical/sol/345/aps/s/act/int/sol_345_scre en_capture.jpg
Download	145,861	Wed 2005.01.12 @ 00:56:38 GMT	Wed 2005.01.12 @ 00:56:42 GMT	/oss/merb/ops/ops/surf ace/tactical/sol/345/aps/s/act/int/sol_345_scre en_capture.jpg
Download	32,958	Wed 2005.01.12 @ 00:35:44 GMT	Wed 2005.01.12 @ 00:35:45 GMT	/opt/bea/user_projects/cip/config/preferences/schrein.preferences
Download	13,453	Wed 2005.01.12 @ 00:35:41 GMT	Wed 2005.01.12 @ 00:35:42 GMT	/opt/bea/user_projects/cip/config/global.properties
Download	23,629	Wed 2005.01.12 @ 00:25:55 GMT	Wed 2005.01.12 @ 00:25:57 GMT	/opt/bea/user_projects/cip/config/preferences/smcienna.preferences
Download	13,453	Wed 2005.01.12 @ 00:25:51 GMT	Wed 2005.01.12 @ 00:25:53 GMT	/opt/bea/user_projects/cip/config/global.properties
Download	14,482	Wed 2005.01.12 @ 00:16:28 GMT	Wed 2005.01.12 @ 00:16:28 GMT	/opt/bea/user_projects/cip/config/preferences/welch.preferences
Download	13,453	Wed 2005.01.12 @ 00:16:25 GMT	Wed 2005.01.12 @ 00:16:25 GMT	/opt/bea/user_projects/cip/config/global.properties
Upload	22,016	Wed 2005.01.12 @ 00:13:04 GMT	Wed 2005.01.12 @ 00:13:05 GMT	/opt/bea/user_projects/cip/config/preferences/e milee.preferences
Download	13,364	Tue 2005.01.11 @ 23:12:42 GMT	Tue 2005.01.11 @ 23:12:44 GMT	/opt/bea/user_projects/cip/config/preferences/lcr umple.preferences
Download	13,453	Tue 2005.01.11 @ 23:12:39 GMT	Tue 2005.01.11 @ 23:12:41 GMT	/opt/bea/user_projects/cip/config/global.properties

Poll every seconds

图 20-8：中间件监测工具中 File 标签页的截图

动态重配置

大多数中间件服务都有关键的运行参数。例如，正如我们在前面所看到的，流服务按照块的方式下载文件，因此它就有一个有关块大小的参数。我们不是把块大小硬编码到程序中，而是把这个值放在一个参数文件中，服务在每次启动的时候都会去读取这个参数文件。这个操作发生在流服务提供者bean被加载的时候（“日志”一节代码的第3~5行）。

在所有中间件服务共享和加载的 *middleware.properties* 文件中，包含了下面这行：

```
middleware.Streamer.blocksize = 65536
```

然后文件读取器bean的readDataBlock()方法可以使用这个值(“日志”一节代码的第183行)。

每个中间件服务都可以在启动的时候加载数个参数值。熟练的编码人员需要掌握的技巧之一就是要知道一个服务的哪些关键值被开放为可装载参数。他们在开发期间肯定是非常有帮助的;例如,我们能够在开发过程中尝试不同的块大小而无需每次都重新编译流服务。

可装载的参数对于把代码投入运行或许还要更加重要。在大多数的产品环境中,对已经投入运行的软件进行修改是非常困难和昂贵的。MER任务也存在这样的问题,因此成立了一个正式的变更控制部门(change control board)来审查那些需要在任务运行时对代码进行的修改。

当然,避免硬编码的参数值是基本的编程101格言之一,它既适用于小型程序,也适用于大型程序。但这对于大型程序来说尤其重要,在这些程序中可能有着非常多的参数值,分散在代码的每个角落。

热交换

热交换(Hot Swapping)是我们在CIP中间件中采用的商业应用服务器的重要功能之一。这使得我们可以用一个中间件服务来替换另一个正在运行的中间件而无需先停止它(以及CIP)。

当我们需要在修改参数值之后强制服务重新进行加载时,可以使用热交换,我们可以通过重新加载一次服务来实现。当然,像流服务这样使用有状态会话bean(文件读取器和文件写入器bean)的服务将会丢失所有的状态信息。因此,只有当服务处于“安静”时期,我们才可以对服务进行热交换,因为此时我们知道服务目前并没有被使用。对于流服务来说,我们可以通过中间件监测工具的Files标签页(参见图20-8)来知道何时处于这种情况。

热交换在大型企业应用程序的运行环境中是最有意义的,在这些环境中,在替换部分程序的同时保持其他部分仍然运行是一项很重要的需求。而对于小型程序来说,最好的方式就是把程序送回去修改。

结论

协作式信息管理系统证明了——是的,即使在像NASA这样庞大的政府部门中——成

功地按时交付一个严格满足实用性、可靠性和稳定性等需求的大型复杂企业软件系统是可能的。火星漫步者已经远远超越了预期要求，它很好地说明了如何成功地设计和构建同时位于火星上和地球上的硬件和软件，以及编码人员使用的技巧。

与小型程序不同，大型应用程序的漂亮性并不一定只存在于优美的算法中。对于CIP来说，漂亮性是在于它的面向服务架构实现以及大量虽然简单却经过仔细挑选的组件——编码人员只需知道将这些组件组合在一起即可。

ERP5：最大可适性的设计

Rogerio Atem de Carvalho & Rafael Monnerat

企业资源计划（ERP）系统通常被认为是大型的、专门的以及非常昂贵的产品。2001年，两家法国公司，Nexedi（主要开发者）和Coramy（第一个用户），开始了对开源ERP系统的开发，也就是ERP5 (<http://www.erp5.com>)。ERP5是以组成其核心的五个主要概念来命名的。ERP5的开发基于Zope项目和Python脚本语言，这两个系统同样是开源的。

我们发现ERP5的功能非常容易得到增强，对于开发人员和用户来说都是如此。原因之一是我们采用了一种独特的以文档为中心的方法，而不是以过程或者数据为中心。这种以文档为中心的模式的核心思想是每个业务过程都依赖于一系列促使过程发生的文档。文档的主题与过程的结构是相关的——也就是说，这些主题反映出了这些数据以及在数据之间的关系。因此，如果观察一下使用ERP5的业务专家们是如何在文档中进行浏览的，你就会发现其中的过程工作流。

Zope的内容管理框架（Content Management Framework, CMF）工具和概念给出了在这种方法后面所蕴涵的技术。每个CMF实例，也可以叫作一个Portal，包含了它所服务的对象，这些服务包括查看、打印、工作流以及存储等。文档结构被实现为一个Python的portal类，而它的行为则被实现为一个portal工作流。因此，与用户进行交互的网络文档实际上是那些由正确工作流所控制的系统对象视图。

本章将说明如何通过以文档为中心的模式和一组统一的概念使ERP5成为一个高度灵活的ERP。我将通过解释如何使用快速开发技术来创建ERP5的项目管理模块Project来阐述这些设计思想。

ERP 的总体目标

ERP是一种致力于把某个公司的所有数据和业务流程都集成为惟一系统的软件。这的确是一种挑战，ERP开发商要为不同的行业提供不同版本的ERP软件，例如石油天然气、医疗、制药、汽车和政府等行业。

ERP软件通常包含一组使公司业务自动化的模块。最常见的模块包括财务、库存控制、工资、生产计划和控制、销售以及会计等。这些模块都是针对用户的定制需求和适应需求来设计的，虽然某个行业的所有公司都有一些共同的结构，但每个公司都希望ERP系统能够满足自己的一些特定需求。随着ERP所服务的业务的发展，ERP软件同样得到了快速的发展，并且随着时间的推移，越来越多的模块都被添加到ERP中。

ERP5

ERP5是由一个正在逐渐发展的商业和学术群体开发的，这个群体遍布法国、巴西、德国、卢森堡、波兰、塞内加尔，印度以及其他的一些国家。它提供了一个基于开源Zope平台 (<http://www.zope.org>) 的集成商业管理解决方案，系统是用Python (<http://www.python.org>) 语言编写的。在ERP5中使用的关键Zope组件分别是：

ZODB

这是一个对象数据库。

DCWorkflow

这是一个工作流引擎。

内容管理框架 Content Management Framework (CMF)

这是一个用于增加和删除内容的架构。

Zope Page Templates (ZPT) Zope 页面模板

基于XML的快速GUI脚本。

此外，ERP5还非常依赖XML技术。每个对象都可以用XML格式来导入和导出，并且在两个或多个ERP5站点之间可以通过SyncML协议来共享同步对象。ERP5还实现了一个从对象到关系的映射模式，在这个模式中存储了关系数据库中每个对象的索引属性，

从而实现了比ZODB更快的对象查找和检索。这样，虽然对象被保存在ZODB中，但查找却是使用标准的查询语言SQL来进行的。

ERP5被设计为一个非常灵活的框架，用于开发企业应用程序。灵活性意味着ERP5能够适用于各种不同的业务模型，并且对系统的修改和维护不会带来高额的开销。为了实现这一点，我们有必要定义一个核心的面向对象模型，并且从这个模型中可以很容易地派生出特定用途的新组件。这个模型必须是足够地抽象以包含所有的基本业务概念。

在ERP5这个名字中已经指出了，ERP5定义了五个抽象概念，这些概念构成了表示业务流程的基础：

资源 (*Resource*)

描述实现某个业务流程所必需的资源，例如员工，技能，产品，机器等。

节点 (*Node*)

这是一个接收和发送资源的业务实体。它可以被关联到某个物理实体（例如企业设备）或者某个抽象实体（例如银行帐户）。元节点（Metanode）是指包含其他节点的节点，例如公司。

路径 (*Path*)

描述某个节点如何从其他的节点来访问它所需要的资源。例如，路径可以是一个定义客户如何从供货商处获得产品的交易过程。

迁移 (*Movement*)

描述了资源在某个给定的时刻和时段中在节点之间的迁移。例如，把原材料从仓库搬运到工厂。迁移是路径的实现。

项 (*Item*)

代表资源的一个实例。例如，CD驱动器是组装电脑的一个资源，而CD驱动器的零件号码23E982则是一个项。

这些核心概念以及一些其他的支撑概念，例如订单（Order）和发货单（Delivery）等，构成了ERP5的统一业务模型（Unified Business Model, UBM）。我们将在本章中看到，通过组合和扩展这五个概念可以实现新的业务流程。这五个概念之间的关系如图21-1所示。路径被关联到把资源发送到目标节点的源节点。迁移也是类似的，它表示由资源描述的项从源节点迁移到目标节点的过程。

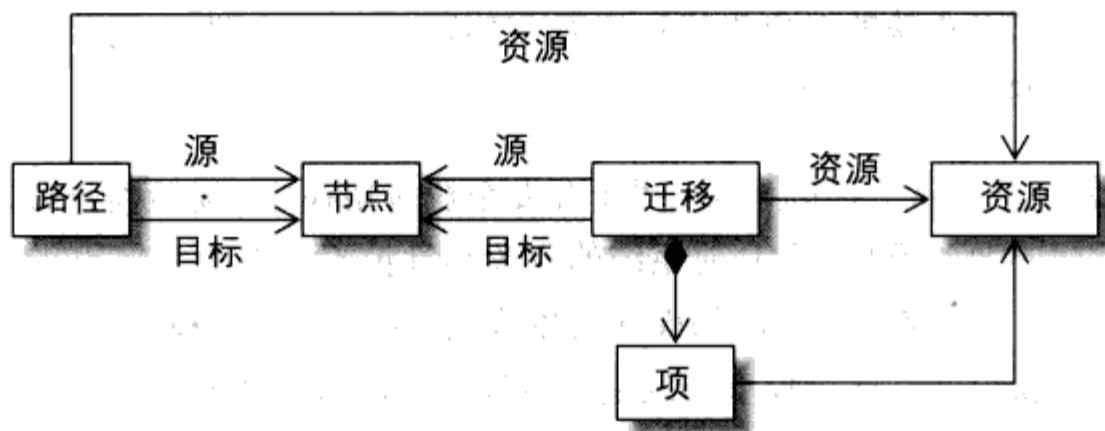


图 21-1：ERP5 的核心类

Zope 基础平台

要想理解为什么说 ERP5 是文档驱动的，我们必须首先理解 Zope 及其内容管理框架 (CMF) 是如何工作的。Zope 最初被开发为一个网页内容管理环境，它提供了一系列的服务来管理网页文档的生命周期。后来，人们开始注意到它同样可以用来实现任意类型的基于网页的应用。

Zope除了管理网页内容之外，它的CMF还是一个致力于加速开发基于内容类型的应用程序的框架。它提供了一系列与这些类型相关的服务，例如工作流、搜索、安全、设计和测试。CMF从 Zope 中继承了对 ZODB (Zope 对象数据库) 的访问，其中 ZODB 提供了事务和回滚等功能。

CMF通过`portal_types`服务所维护的CMF类型来实现应用程序的结构部分，而CMF类型又是用于某种 portal 已知类型的注册工具。`portal`类型的可见部分是一个表示它的文档。为了实现这种行为，在 `portal` 类型中含有与之相关联的动作 (action)，用来组成一个工作流，而这些动作又是业务流程的实现。在文档发生动作将改变动作的状态，并且这个动作是由实现某个业务逻辑的Python脚本来执行的；例如，计算一个订单总共开销的动作。在有了这个框架之后，我们在 ERP5 中开发一个应用程序时，就必须按照根据包含业务流程数据的文档来思考，文档的生命周期是由实现业务流程行为的工作流来保持的。

为了充分利用 CMF 结构，ERP5 代码被分解为一个四层结构，这个结构实现了一个概念转换链，并且在最高的层级上有着配置任务。

在第一层中包括五个核心类。为了实现简单的文档管理，这些类中没有代码，而只是一个架构：

```
class Movement:  
    """  
    一定数量的资源按照给定的变化从源节点迁移到目的节点  
    """
```

在第二层中是 Python 核心类的实现。但在这里，它们仍然是抽象类。在这些类中已经有了一些 Zope 的内容，并且它们是从 XMLObject 继承下来的，这就意味着每个对象都可以被序列化为 XML 格式以用于同步或者导出。

类的属性被组织成属性表。属性表是一组可配置的属性，这些属性可为不同对象视图的创建提供便利，属性表可以由各种不同的类方法来操作。此外，这些视图使系统管理员能够以非常灵活和精细的方式来建立安全机制。

例如，在 SimpleItem 属性表包含了 title、short_title 和 description 等属性。系统管理员可以设置一个安全模式，这样某些用户只能看到这些属性，而其他一些用户则可以写入这些属性：

```
class Movement(XMLObject):  
    """  
    一定数量的资源按照给定的变化从源节点迁移到目的节点  
    """  
  
    # 定义类型的名字  
    meta_type = 'ERP5 Movement'  
    # 定义 CMF 类型名字  
    portal_type = 'Movement'  
    # 增加基本的 Zope 安全配置  
    add_permission = Permissions.AddPortalContent  
    # 把这个类型作为一种有效的内容类型列出来  
    isPortalContent = 1  
    # 这种类型可以用于 ERP5 快速应用程序开发技术  
    isRADContent = 1  
    # 用于交易和库存操作  
    isMovement = 1  
  
    # 声明式安全性  
        # 存储基本类的安全信息  
    security = ClassSecurityInfo()  
        # 在默认情况下，允许认证用户查看对象  
    security.declareObjectProtected(Permissions.AccessContentsInformation)  
  
    # 声明式属性  
    property_sheets = ( PropertySheet.Base  
                        , PropertySheet.SimpleItem  
                        , PropertySheet.Amount  
                        , PropertySheet.Task  
                        , PropertySheet.Arrow  
                        , PropertySheet.Movement  
                        , PropertySheet.Price  
                        )
```

在第三层中包含的是元类，它们是可实例化的类。在这一层，类可以表示具体的业务实体：

```
class DeliveryLine(Movement):
    """
    DeliveryLine 对象使得流线可以用 Delivery (装箱单, 订单, 发货单等) 来实现。
    它可以包含一个价格 (用于保险, 关税, 发货单, 订单等)
    """

    meta_type = 'ERP5 Delivery Line'
    portal_type = 'Delivery Line'

    # 声明式属性
        # 必须重载从 Movement 继承下来的 property_sheets 属性
    property_sheets = ( PropertySheet.Base
                        , PropertySheet.XMLObject
                        , PropertySheet.CategoryCore
                        , PropertySheet.Amount
                        , PropertySheet.Task
                        , PropertySheet.Arrow
                        , PropertySheet.Movement
                        , PropertySheet.Price
                        , PropertySheet.VariationRange
                        , PropertySheet.ItemAggregation
                        , PropertySheet.SortIndex
                    )
```

最后，在第四层上是 Portal 类型，这些类型是基于 CMF 的。在这个层级上将进行配置。例如，在图 21-2 中给出了 Properties 标签页的主要部分。在这个截图中给出了 Task Report Line 的属性。这个类型是 Delivery Line Meta 类型的实现。我们可以注意到，新的属性表可以被添加到这个标签页中，但我们的项目工具并不需要新的属性表。

Name	Value	Type
Title	Task Report Line	string
Description		text
Icon	organisation_icon.gif	string
Product meta type	ERP5 Delivery Line	string
Product factory method	addDeliveryLine	string
Add permission		string
Init Script		string

图 21-2: Properties 标签页

在图 21-3 中给出了 Actions 标签页，列出了与 Task Report Line 类型相关的动作。这些动作用来实现这种类型的具体服务。图中可以看到 View 和 Print 这些服务。

Name	View
Id	view
Action	string:\${object_url}/TaskLine_view
Icon	
Condition	
Permission	View
Category	object_view
Visible?	<input checked="" type="checkbox"/>
Priority	1.0
Name	Price
Id	price_view
Action	string:\${object_url}/DeliveryLine_viewPrice
Icon	
Condition	object/hasCellContent
Permission	View
Category	object_view
Visible?	<input checked="" type="checkbox"/>
Priority	2.0

图 21-3: Actions 标签页

这个表示系统类的四层结构使我们能够很容易逐步地增加系统功能和平台特性。它还实现了 ERP5 中一个非常普遍的功能：在实现新的 portal 类型时无需在系统中创建新的类。程序员所做的工作就是修改一个类的外观，因为 ERP5 的核心概念可以表示具体业务领域中的实体。

例如，Movement 既可以用于表示在金融模块中的现金撤销，也可以表示在库存模块中材料从仓库到工厂的转送。为了实现这个功能，我们创建了一个 portal 类型用来表示现金撤销，并且创建了另外一个 portal 类型来表示材料转送，每种类型都使用了在 GUI 中出现的业务术语。

除了使用基本的 CMF 特性之外，ERP5 还实现了一些额外的功能来提高编程效能。我们最感兴趣的或许就是关系管理器的概念，这些对象负责维护对象组之间的关系。通常，把关系逻辑编码到每个业务类中是件乏味的工作，并且容易产生错误。同样，传统的关系代码分布在许多业务类的实现中（例如回指指针，删除通知等），这比采用中间管理器的方法要难以跟踪、维护和保持同步。

在 ERP5 中，Portal Categories 服务记录了相关对象之间所有的一对一、一对多和多对多的关系。查询方法、getter 和 setter 方法以及关系代码都是自动生成的。

在这个服务中包含了base category对象，这些对象用来连接那些协作执行给定业务流程的类。对于每个base category，ERP5都将自动生成所有的getter和setter方法。例如，base category源是对Node类型对象的引用。在给定的ERP5实现中，如果Order类被配置为拥有这个base category，那么这个系统将自动地包含从订单移到节点的所有必需的方法和引用，反之亦然。

ERP5 Project 中的概念

为了说明如何对ERP5模块进行编码，我们将在本章剩下的大部分内容中研究ERP中的Project，这是一个灵活的项目管理工具，可以按照多种方式来使用。

由于全球业务环境的快速发展和竞争力的不断增强，项目逐渐成为开发创新产品和服务的普遍形式。因此，项目管理在所有工业部门中都引起了兴趣。

但什么是项目？根据Wikipedia的定义，项目是“创造独特产品或服务的临时活动”(<http://en.wikipedia.org/wiki/Project>, last visited April 13, 2007)。

项目的独特性使得它们的管理变得困难，即使对于小型的项目来说也是如此。因此就产生了项目管理需求，即“组织和管理资源的方式，按照这种方式这些资源在规定的范围、质量、时间和成本等限制之内交互完成项目所需要的工作”(http://en.wikipedia.org/wiki/Project_management, last visited April 13, 2007)。

因此，项目管理必须控制一系列与资源相关的数据，例如资金、时间和人员，以确保所有事情都是按计划进行的。由于这个原因，我们需要信息工具来简化对大量数据的分析。

ERP5 Project的首次应用是被用作为一个“内部的”项目管理工具以支撑ERP5创建项目。后来，对它进行重新设计以支撑其他一般类型的项目。而更为广泛的应用是，只要在项目的概念有助于生产计划和控制的地方，都可以使用这个工具来管理订单的计划和执行。换句话说，ERP5 Project应该适用于所有那些把项目视作为由一系列任务组成并且由一系列约束来限制的情况。

ERP5还使得开发人员在交付其他全新模块包时，能够重用目前的包。依照这个概念，新的业务模板(business template, BT)是在现有模板的基础上创建出来的。

在开始实现ERP5 Project时，ERP5中已经包含了Trade业务模板。因此开发团队决定把Project基于Trade来开发，通过重用在交易操作中开发的逻辑来表示项目的计划部分。在完成了第一个版本的Project之后，开发人员可以对Project业务模板进行改进并再用这些改进来重新增强Trade业务模板，从而使得它变得更加灵活。

在构建 Project 时，Trade 中有趣的部分是 Order 和 Delivery 这两个类。这些类是 UBM 的一部分，并且是 Order Line 和 Delivery Line 对象的容器，而这些对象又是包含 Order 和 Delivery 的 Movement 对象，如图 21-4 所示。图中最底层的子类是所有的 portal 类型。因此，它们有着与它们的超类基本相同的结构，但是每个 portal 类型在工作流中有着不同的 GUI 和变化，从而可以根据项目管理逻辑来发生动作。

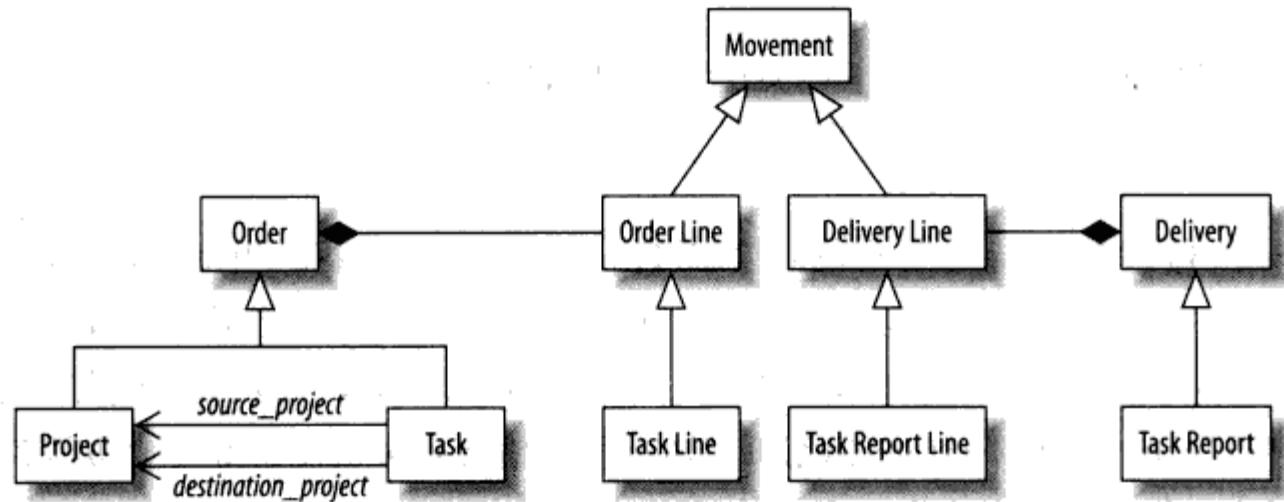


图 21-4：在 Trade 和 Project 之间的关系

Order 和 Delivery 之间的关系是通过因果关系来维护的，基本上决定了对于每个确定下来的订单，都会在将来的某个时刻有一个对应的发货单。Task 和 Task Report 继承了相同的行为。相应地，Order Line 表示在两个节点之间已经计划的资源迁移，在被确认之后将被执行并生成 Delivery Line。因此，从项目管理的角度来看，Task 实现了项目计划，而 Task Report 则实现了项目控制。

编码实现 ERP5 Project

在创建 Project 业务模板时，我所考虑的第一件事情就是主 Project 类。最初，我们并没有创建一个新类，而是决定简单地使用 Order 类并且不再做修改。但过了一段时间，我们意识到订单的业务定义与项目的业务定义是完全不同的两个定义，我们应该把 Project 创建为 Order 的子类，但出于独立性的考虑，我们没有添加任何新的代码。在这一设计中，项目就是一个由一系列目标或者里程碑来描述的对象，并且每个项目都有一个或者多个任务与之关联。

然后，由于在项目操作和交易操作之间有着许多的不同，因此我们需要确定如何来实现任务管理。第一件要考虑的事情就是任务可以在项目之外进行——例如，在生产计划中。因此，我们把任务认为是 Task Line 或者在任务中更小活动的组合。这样，我们就把任务从项目中解耦开来，使它们能够用于其他情况，而同时仍然通过 source_project 和 destination_project 等 base Category 对象与 Project 保持联系。

任务是通过配置来实现的，正如我们在图 21-2 中的 Task Report Line 里面所做的工作，区别在于这里把 Order 用作为一个元类。创建一个与项目相关的任务如下所示：

```
# 在 task_moudle 中增加一个任务。context 表示当前项目对象。  
context_obj = context.getObject()  
# newContent 是 ERP5 的一个 API，用于创建一个新内容。  
task = context.task_module.newContent(portal_type = 'Task')  
# 把 source_project 引用设置为指向 task。  
task.setSourceProjectValue(context_obj)  
# 把这个用户重新定向到 Task GUI，这样用户可以编辑它的属性。  
return context.REQUEST.RESPONSE.redirect(task.absolute_url() + '?:  
portal_status_message=Created+Task.')
```

记住，对于检索某个特定项目的任务而言，程序员仅需要使用 base category source_project。在有了这个分类之后，ERP5 RAD能够自动生成访问者(accessor)的签名和算法。我们注意到很有意思的是，对于 Task 和 Project 将会创建相同的访问者。程序员可以在图 21-3 的 Actions 标签页中通过配置信息来决定使用哪一个访问者。在这个标签页中，程序员可以定义一个新的 GUI 来使用以下方法：

```
### 这些访问者被用于从任务移到项目  
  
# 这个方法将返回相关的 Project 引用  
getSourceProject()  
  
# 这个方法将设置相关的 Project 引用  
setSourceProject()  
  
# 这个方法将返回相关的 Project 对象  
getSourceProjectValue()  
  
# 这个方法将设置相关的 Project 对象  
setSourceProjectValue()  
  
### 这些访问者用来从 project 移动到 task  
  
# 这个方法将返回对相关任务的引用  
getSourceProjectRelated()  
  
# 为了避免破坏封装性，这个方法并没有被生成  
# setSourceProjectRelated()  
  
# 这个方法将返回相关的 task 对象  
getSourceProjectRelatedValue()  
  
# 为了避免破坏封装性，这个方法并没有被生成。  
# setSourceProjectRelatedValue()
```

你一定会问通常的项目领域属性和行为在什么地方。对于属性的答案是，在大多数情况下，它们是 Movement 和一些其他 UBM 类的属性，并且在 GUI 中被其他的名字屏蔽了。

在其他的情况下，属性是通过一个base category来实现的，所有的访问者都将按照预期自动生成。

这种情况的示例之一就是任务的前置任务，即在执行给定任务之前需要被执行的其他任务——这是一个非常基本的项目管理概念，它并没有被包含在交易操作中。这组任务同样是由一个叫做前置任务的基本分类来实现的，它通过一种可配置的方式把一个任务链接到它的前置任务，因为在这个分类中包含了代码所需要的东西。

工作流实现了任务行为。我们再次重用了基本的Movement和更为具体的Order行为。这些工作流按照使项目管理有意义的方式来操纵这些对象，并且包含了一些脚本来做这些工作。工作流使得开发更为容易，因为它们是可配置的，并且程序员只需要为具体的对象操作编写脚本。

图21-5所示为Task工作流。在每个方框中，圆括号内的单词表示状态ID。带有_action后缀的状态转换是由GUI事件触发的；其他的则是由工作流事件内部触发的。对于每个状态转换，我们都可以定义pre- 和 post- 脚本。这些脚本将会根据业务逻辑来操作对象——在这种情况下是任务执行逻辑。Task表示的是流程的计划视图，这基本上要经过计划（Planned）、定购（Ordered）和确认（Confirmed）等状态。

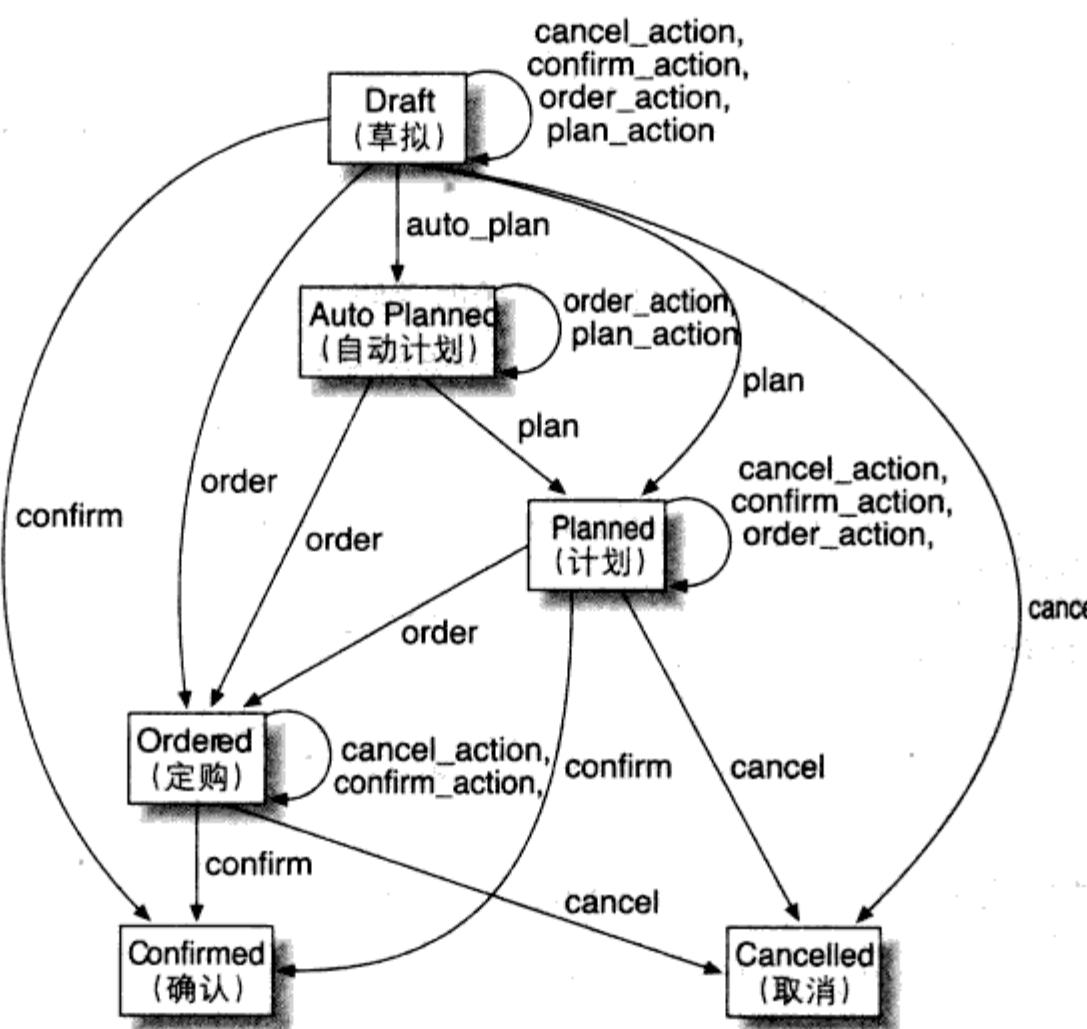


图 21-5：Task 工作流

这个工作流与Order是一样的，不过有些脚本根据项目的领域逻辑进行了修改。下面我们将脚本order_validateData作为一个示例，这个脚本将在执行每个_action之前被调用，如下所示：

```
### 这个脚本将检查 Task 中的必要数据
# 得到使用中的任务对象
task = state_change.object
error_message = ''
message_list = []
# 检查这个任务是否被关联到某个项目
if task.getSource() is None:
    message_list.append('No Source')
# 如果初始状态为空，但有一个终止状态，则
# initialDate = finalDate
if task.getStartDate() is None and task.getStopDate() is not None:
    task.setStartDate(task.getStopDate())
if task.getStartDate () is None:
    message_list.append("No Date")
if task.getDestination() is None:
    message_list.append('No Destination')
# 对于每个被包含的对象，过滤掉是 Movement 对象。
# 通常的返回结果是像下面这样
#('Task Line', 'Sale Order Line', 'Purchase Order Line')
for line in task.objectValues(portal_type=task.
getPortalOrderMovementTypeList()):
    # 检查是否所有的 Movement 对象都有一个关联的资源
    if line.getResourceValue() is None:
        message_list.append("No Resource for line with id: %s" % line.getId())
# 如果发生了错误，则引发一个警告
if len(message_list) > 0:
    raise ValidationFailed, "Warning: " + " --- ".join(message_list)
```

在图 21-6 中给出了 Task Report 工作流。它遵循的逻辑和 Delivery 工作流一样，此外还增加了一些脚本，例如 taskReport_notifyAssignee，如下所示：

```
task_report = state_change.object
# 查找任务的分配者
source_person = task_report.getSourceValue(portal_type="Person")
# 查找被分配的人
destination_person = task_report.getDestinationValue(portal_type="Person")
# 得到分配者的邮件地址
if source_person is not None:
    from_email = destination_person.getDefaultEmailText()
    email = source_person.getDefaultEmailValue()
    if email is not None:
        msg = """
# 以下是带有消息的预格式化字符串以及任务数据
"""
        email.activate().send(from_url = from_email,
                               subject="New Task Assigned to You",
                               msg = msg)
```

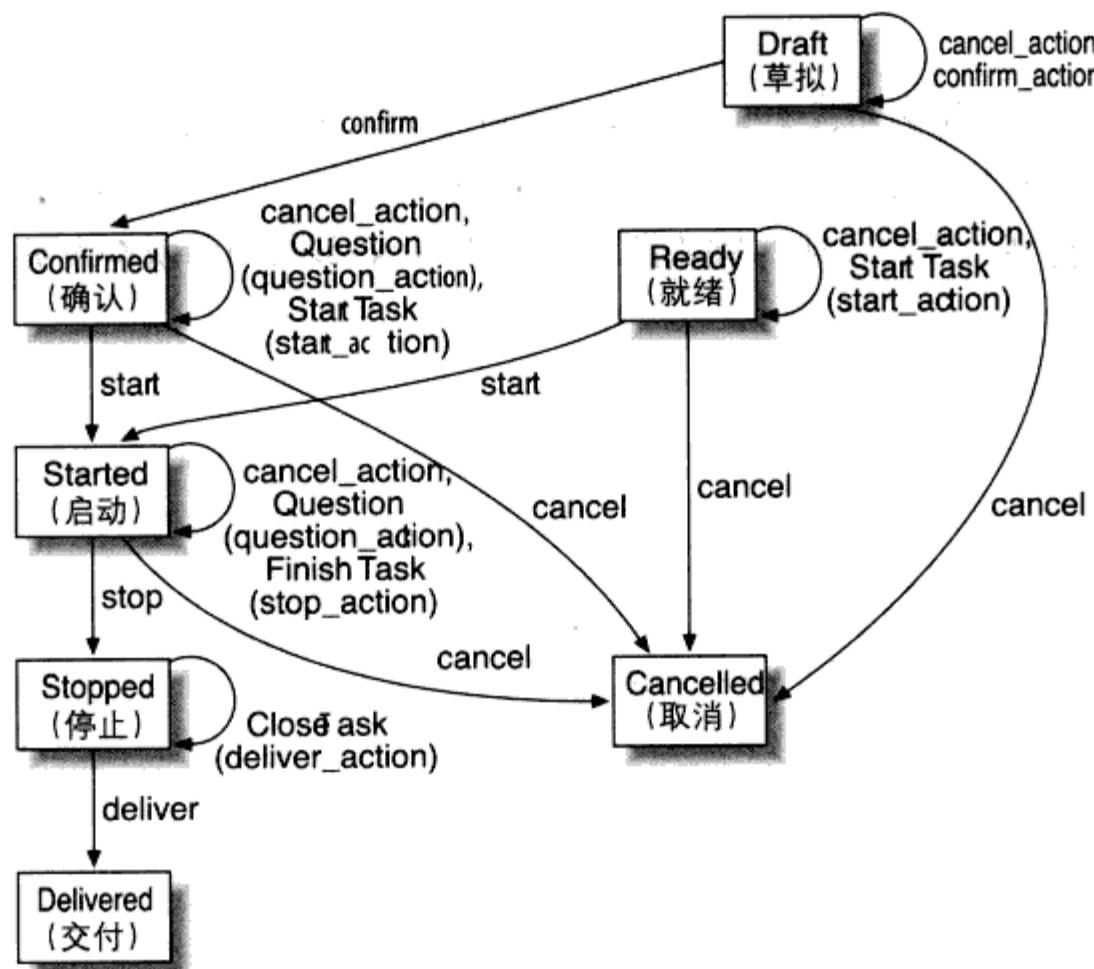


图 21-6：Task Report 工作流

结论

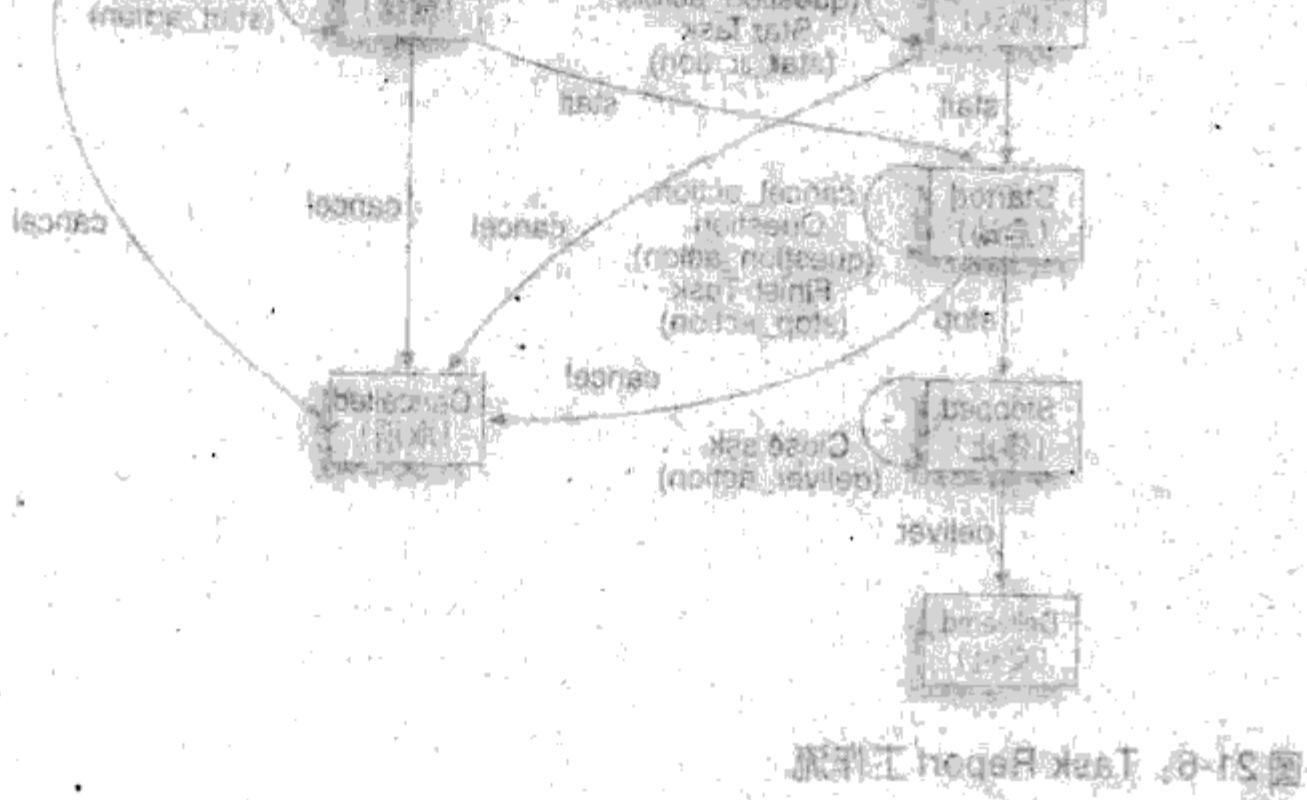
ERP5 团队能够通过对已存在的核心概念和代码进行大量地重用实现一个高度灵活的工具，这个工具既可以用于“传统的”项目管理，也可以用于订单计划和执行控制。实际上，重用是在 ERP5 开发中的一个日常操作，而全新的模块只需通过修改 GUI 元素和调整工作流来创建。

由于对重用的强调，在对象数据库上的查询可以在 portal 类型或者元类的抽象层次上进行。在第一种情况中将检索具体的业务领域概念，例如项目任务。在第二种情况下，所有与 UBM 通用概念相关的对象都将被检索，这对于像统计数据收集这样的需求来说是非常有用的。

在本章中，我们编辑了一些代码以使它们更加可读。ERP5 所有的原始代码可以在网址：<http://svn.erp5.org/erp5/trunk> 上得到。

致谢

我们感谢 ERP5 的创建者和首席架构师 Jean-Paul Smets-Solanes，以及这个团队中的其他的所有人员，尤其是 Romain Courteaud 和 Thierry Faucher。当作者在对 ERP5 设计的讨论和实现中提到“我们”时，“我们”指的是在 Nexedi 公司的所有人员。



结语

前面已经看到一个典型的 ERP 系统的架构设计，它由许多模块组成，如核心模块、客户管理模块、销售管理模块、采购管理模块、生产管理模块、财务管理模块等。这些模块通过数据共享和集成，实现了企业的整体业务流程管理。当然，这只是 ERP 系统的一个基本框架，实际应用时还需要根据企业的具体需求进行定制化开发。

通过前面的分析，我们可以看到，ERP 系统的核心在于其强大的数据集成能力和业务逻辑处理能力。企业可以通过 ERP 系统实现从采购到销售的全链条管理，从而提高企业的运营效率和竞争力。当然，ERP 系统的实施需要企业在资金、人员和技术等方面投入较大的资源，因此企业在选择 ERP 系统时，需要综合考虑自身的实际情况和未来的发展需求。

总的来说，ERP 系统为企业提供了全面的信息化解决方案，帮助企业更好地应对市场竞争。在未来，随着云计算、大数据、人工智能等技术的不断发展，ERP 系统也将朝着更加智能化、个性化、定制化的方向发展。

一匙污水

Bryan Cantrill

往一桶酒里加入一匙污水，你得到的是整桶污水。

叔本华之熵定律

与其他工程项目不同，软件用二进制来表示它的正确性：要么对，要么错。与桥梁、机场或者是微处理器这样的东西不同，软件没有额定负荷，没有最高限速，也没有环境工况的要求，没有任何物理参数可以用来衡量软件在多大范围内算是正确的。从这个意义讲，软件更像是数学证明，而非物理机器。一个证明是否优雅有很大的主观性，但它是否正确却是完全客观的。

实际上，这使得软件具有一种纯粹的属性：它如果是对的，那么这种正确性会以绝对的、永恒的方式存在。在过去传统观念中，只有数学才享有这种特性。如果说这时候软件纯粹得像温文尔雅的君子，别忘了，它还有脆弱的一面：只要一个瑕疵就可以令它从无可比拟的成功跌入令人绝望的失败中。这时候，它迅速化身成恶魔，恰如《化身博士》中的 Jekyll 博士和 Hyde 先生。

当然，并不是每个 bug 都一定是致命的。但我们经常能看到，一个 bug 所揭露出的问题，可能比问题自身的表述严重得多。比如说，那种足以动摇整个软件根基的设计缺陷，它们直指系统核心，极端情况下会导致整个软件彻底失效。这真应了那句老话，一颗老鼠屎坏了一锅汤，一匙污水臭了一桶酒。

就我个人而言，对于酒和污水的这种天壤之别，莫过于1999年的那次意外事件，我们那时候正在开发Solaris内核的一个关键子系统。我认为有必要花点篇幅详细探讨这当中的问题及其解决方案。它们告诉我们，那些底层的设计缺陷是怎么变成bug的；面对那些复杂而又重要的功能，力图尽善尽美的想法是如何演变成可怕的梦魇的。

在开始我们的探讨之旅前，我先提醒一句，我们将会深入Solaris内核的中心部分，接触操作系统中那些最根本也是最微妙的机理。个中细节会令人抓狂，我们的历程也会像那些探险者一样，穿越茫茫的黑暗，涉过冰冷的水池，钻出令人窒息的通道。这一切都是为了寻找那个深藏地底的美丽洞穴。好了，各就各位，带好头灯，抓牢水瓶，让我们沉入Solaris的内核世界……

故事的主角是旋转门（turnstile）子系统。旋转门是Solaris用来阻塞和唤醒线程的机制，它是互斥锁、读写锁这些同步原语的基石。我们来看看代码是什么样的（注1）：

```
/*
 * 旋转门提供同步原语(比如, 互斥锁, 读写锁)的堵塞和唤醒功能, 支持优先级继承。
 * 典型应用如下:
 *
 * 在函数 foo_enter()里进行读操作时, 阻塞于锁 'lp':
 *
 *     ts = turnstile_lookup(lp);
 *     [ 如果仍在锁定状态, 设置等待者的标志位
 *       turnstile_block(ts, TS_READER_Q, lp, &foo_sobj_ops);
 *
 * 在函数 foo_exit()里, 唤醒在锁 'lp' 的写操作上进行等待的线程:
 *
 *     ts = turnstile_lookup(lp);
 *     [ 放弃锁定(将拥有者置为NULL), 或者直接移交该锁(将拥有者设为待唤醒的线程)
 *     [ 如果是即将唤醒最后的等待者, 将其标志位清空
 *       turnstile_wakeup(ts, TS_WRITER_Q, nwaiters, new_owner or NULL);
 *
 * 函数 turnstile_lookup()返回时会持有用于lp的旋转门哈希链。
 * 函数 turnstile_block()和函数 turnstile_wakeup()都会丢弃旋转门锁。
 * 用户必须调用 turnstile_exit()函数来中止一个旋转门操作
 *
 ...
 */

有了旋转门这种抽象机制，那些同步原语可以专注于它们自身的逻辑，而不必关心“阻塞”、“唤醒”之类的操作细节。正如上面那段注视所言，turnstile_block()函数是同步原语中真正执行阻塞的操作，我们的地下城探险之旅也就从这个函数开始庄严启程了。先从我自己写的一段隐晦注释作为开始吧：
```

注1：本代码属于开源产品，您也可以通过如下网址访问 http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/os/turnstile.c/*。

```

/*
 * 遍历整个堵塞链，将优先级赋予每个节点
 */
while (t->t_sobj_ops != NULL &&
       (owner = SOBJ_OWNER(t->t_sobj_ops, t->t_wchan)) != NULL) {
    if (owner == curthread) {
        if (SOBJ_TYPE(sobj_ops) != SOBJ_USER_PI) {
            panic("Deadlock: cycle in blocking chain");
        }
    }
    /*
     * 如果我们遇到的环以 mp 结尾，就不是一个“真正的”环，
     * 因为 mp 在休眠之前将会被抛弃。而且，既然我们遇到了环，
     * 说明我们已经将优先级赋予每一个节点了。因此，可以在
     * 此处退出遍历。
     */
    if (t->t_wchan == (void *)mp)
        break;
}

```

对我来说，上述黑体注释（包括被注释的两行代码，全部用黑体标注）将永远是说明瑕玉和糟粕之间区别的最经典代表。它们在Solaris 8操作系统最后也是最抓狂的时刻被加进去，这段时间也是我工程师生涯几个最紧张的经历之一。那时候，我跟Sun的院士Jeff Bonwick合作了一个星期，我们共享了非常多的思维、想法，以至于我们都把它称做“意识熔炼”。

待会儿我们还会再研究这段代码，看看代码背后的“意识是如何熔炼的”。在此之前，让我们先深入到旋转门的内部，探究旋转门是如何处理经典的优先级倒置问题的。

如果你还不清楚什么是优先级倒置问题，我先做个解释：现有三个不同优先级的线程，如果优先级最高的那个线程被优先级最低的线程所持有的同步对象阻塞。那么，在一个纯粹的按优先级占先的单处理器系统中，优先级居中的线程将会持续运行下去，而高优先级的线程却无法得到运行。这个结果可以用图 22-1 来表示。

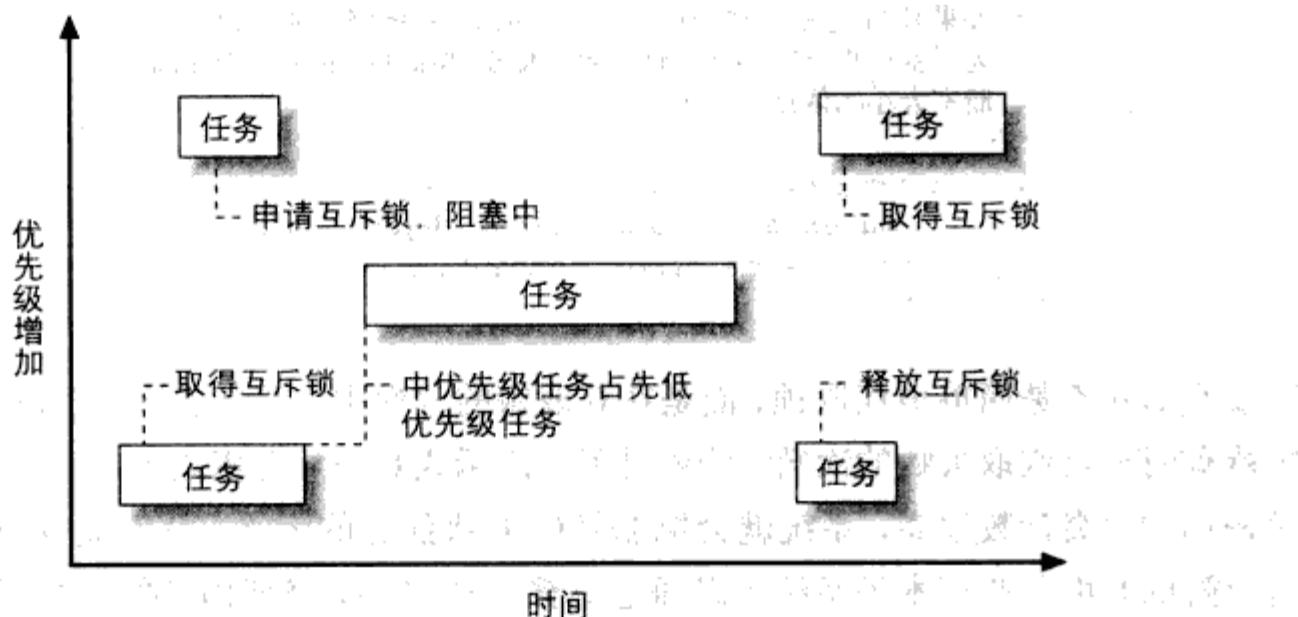


图 22-1：优先级倒置

避免优先级倒置的一个办法是采用优先级继承技术。在优先级继承里，当一个高优先级的线程被一个低优先级线程所持有的资源阻塞时，高优先级线程可以把优先级过继给低优先级线程，有效范围为整个临界区。换言之，低优先级线程如果占有了一个高优先级线程所需的资源，那么，它的优先级也会被拔高。当低优先级线程退出临界区，释放同步对象后，等待这个同步对象的高优先级线程会被唤醒，低优先级线程的又会回到原来的优先级。通过这种方法，中优先级的线程就不会有机会运行，也就避免了优先级倒置。

现在的Solaris早就在同步原语中支持优先级继承了。事实上，这个功能是SunOS 4.x和Solaris 2.x架构上的一个不同点，也是旋转门子系统的核心服务之一。在同步原语中正确应用优先级继承是一件令人头大的事情。我们必须知道是谁拥有锁，如果拥有者同时又被阻塞的话，还得知道它被阻塞于哪个锁上。也就是说，一个线程被锁阻塞，同时，拥有这把锁的线程又被其他锁阻塞。这时，我们必须知道此线程到底阻塞在哪把锁上，哪个线程拥有这把锁。这个被阻塞线程之间的链条被称为阻塞链，它的细节对实现优先级继承举足轻重。鉴于其重要性，我们还是看一些实际开发中有说服力的例子吧。

在Solaris的两个广为人知的子系统中，能找到一个关于阻塞链的实例，这两个子系统分别是内存分配器和ZFS文件系统（Zettabyte Filesystem）。它们的内部细节不是我们现在关心的主要话题，我在这儿就只来个蜻蜓点水式的介绍。核心内存分配器总的来说是一个“对象-缓存”分配器，所有的内存分配都取自缓存，它管理固定大小的对象。已分配的空间叫做储室（Magazine），每个CPU都有自己的储室。当储室空间耗尽时，就会从一个叫做储库（depot）的空间分配内存。这种双层结构具有良好的CPU扩展性，大多数分配取自每个CPU的储室，因此很少发生分配冲突，而且分配能力具有几乎线性的CPU可扩展性。我忍不住又想岔开一点话题来演示一个优雅的例子：当CPU的储室用完时，分配器是如何请求向储库加全局锁的。

```
/*
 * 如果我们无法无冲突地获得储库的锁，那就修改冲突计数
 * 为了获得更好的可扩展性，我们依据储库冲突速率作为提高
 * 储室大小的依据.
 */
if (!mutex_tryenter(&cp->cache_depot_lock)) {
    mutex_enter(&cp->cache_depot_lock);
    cp->cache_depot_contention++;
}
```

这段代码并不是简单地获取锁，而是在获取锁的过程中，还记录下由于锁被其他的资源所持有而导致获取失败的次数。计数结果可以看成是全局范围内的冲突频度，如果单位时间内的冲突计数过高，系统就会增加每CPU级缓存的大小，从而减少全局的冲突率。这个简单的机制使得本子系统可以通过调整自身结构来降低内部的冲突，的确是漂亮！

让我们回到刚才的例子。对 ZFS，我们只需知道，文件和目录在内存中对应的数据结构叫作 z 结点。

有了这些背景知识，现在我们就可以探究如下事件序列：

1. 线程 T1 试图从 CPU 2 的 kmem_alloc_32 缓存中申请内存，也意味着它要先获取对 kmem_alloc_32 储室的锁定。不巧的是，CPU 2 的所有储室都已被耗尽了，因此 T1 转而锁定全局储库来申请 kmem_alloc_32。显然，它同时拥有了 CPU2 的储室锁和全局的储库锁。此时，它被其他线程占先。
2. 运行在 CPU 3 之上的第二个线程 T2 也试图独立申请 kmem_alloc_32 缓存。同样不巧的是，CPU 3 上所有的储室也被用完了。T2 转而向全局储库进行申请，在试图加锁储库时，它发现这个锁已经被 T1 持有了。于是，T2 被阻塞。
3. 当 T2 被阻塞之后，同样运行在 CPU 3 上的第三个线程 T3，开始请求建立 ZFS 文件 /foo/bar/mumble。其间，它必须为项目 mumble 建立一个 ZFS 目录锁。它先锁定 /foo/bar 对应的 z 结点，接着，为结构 zfs_dirlock_t 申请空间。zfs_dirlock_t 是一个 32 比特的结构，应当从 kmem_alloc_32 缓存中申请空间。不难发现，T3 在申请时，看到 T2 已经锁定 kmem_alloc_32，于是它也被阻塞了。
4. 第 4 个线程，T4，试图查看目录 /foo/bar 的内容。同样，它也需要获取对 /foo/bar 对应的 z 结点的锁定。由于此锁被 T3 持有，于是，T4 也被阻塞。

T4 阻塞于 T3，T3 阻塞于 T2，T2 阻塞于 T1，这个线程链条就构成了阻塞链。有了阻塞链的直观认识之后，我们可以更容易地体会优先级继承实现过程中的微妙之处：当一个被阻塞的线程要传承它的优先级时，它需要可靠地遍历整个阻塞链。在遍历过程中，必须得到整个阻塞链上所有线程的快照，不能漏掉，也不能找错。在我们这个例子里，当 T1 释放了阻塞 T2 的锁之后（一直到 T4），我们并不希望将我们的优先级传承给 T1 —— 这就可能使 T1 处于非常高的优先级。

那么，怎样才能可靠地遍历整个阻塞链呢？在 Solaris 里，线程的调度状态（如，运行、等待运行或休眠）由一个叫线程锁的自旋锁（spin lock）保护。不难想到，为了可靠地遍历整个阻塞链，我们只要持有所有这些线程的线程锁即可。不幸的是，线程锁的实现机制不允许我们这么做。

线程锁非常特殊，它不是传统意义上的自旋锁。实际上，它只是一个指向自旋锁的指针，操纵当前线程的结构正是通过这个指针所指向的锁得到保护的。当线程的操纵结构发生变化时，线程锁的也指向不同的锁。

比如，当一个线程进入等待队列时，这个指针会指向线程所在优先级的调度队列的锁。当线程进入运行态时，线程锁又指向CPU所在的cpu_t结构内部的一个锁。当线程被某个同步原语阻塞时，线程锁又转而指向旋转门表内的一个锁（乌云滚滚，邪恶之音依稀可闻）。

旋转门表这个结构随着我们的深入将会越来越重要。但当务之急是，我们根本没法轻松获取所有线程锁，因为多个线程完全可能指向同一个调度锁。如果试图直接获取所有线程锁，我们就会陷入死锁，因为我们试图请求已拥有的锁！（注2）

幸运地是，我们实际上不必获取所有的线程锁，就可以完整地得到阻塞链上所有线程的信息。这得归功于阻塞链的一个可能是不言而喻的重要特性，即，阻塞链只能从未阻塞的那一端展开。这就意味着，如果一个线程阻塞于某个同步原语，那么只有持有这个同步原语的线程，才能唤醒被阻塞线程。

在本案例中，线程T3只能被T2唤醒。如果能够保证T3与T2之间是原子操作，并且维持T2与T1之间的原子性，那么我们就可以确保T3不会在中间被唤醒，哪怕此时T3的线程锁已经被释放。

这就意味着我们不需要锁定整个阻塞链，只需要每次获取相邻节点的锁即可。阻塞T4时，我们先获取对T3的锁定，然后获取对T2的锁定；接着，释放T3的锁定，请求锁定T1；然后释放T2，如此这般。既然一次只需要持有两个线程锁，那指向同一个锁的问题就简单了：如果两个指针指向同一个锁，那就在遍历经过这两个线程时，简单地保持这个锁即可。

看样子问题算是解决了，但是，还有一个重要的障碍需要清除，这个问题源自另外一个设计抉择。刚才我们提到线程锁可以指向旋转门表内的调度锁。现在有必要解释一下旋转门表，这个结构在接下来的探索之旅中还会反复碰到。

旋转门表是一个哈希表，其索引关键字是同步原语的虚地址，其内容项是被阻塞线程的队列。每个队列都由一个旋转门锁锁定其头部。当一个线程阻塞于某个同步原语时，它的线程锁指针就指向这个旋转门锁。

这儿有个关键的，也许是费解的设计抉择。当一个线程阻塞于一个同步原语时，它进入的队列对同步原语来说并不惟一。因为几个同步原语有可能恰好映射到同一个索引项。

注2：这儿还有一个更微妙地方，那就是锁排序。总之，有诸多理由可以证明获取所有线程锁不是一件容易的事情。

为什么这么设计呢？作为一个具有高并发能力的操作系统，Solaris 有很好的同步粒度。它可以容纳数量庞大的同步原语实例，这些实例以极高的频度运作，它们之间的冲突却总是几乎为零。因此，内核同步原语的数据结构 `kmutex_t` 和 `krwlock_t` 必须尽可能地小，对它们的操作也尽量优化成常用的，无冲突的情况。把阻塞链放在同步原语结构内部是不能接受的：要么费空间（同步原语包含了调度锁和队列指针之后，就变得臃肿了），要么费时间（对于没有任何冲突的情况，操作一个复杂的数据结构是不明智的）。无论如何，同步原语的结构容纳不下阻塞链，所以我们需要旋转门表。

有了对旋转门表的认识，我们可以重新澄清一下刚才的设计抉择。阻塞于不同同步原语的线程，它们的线程锁可能会指向同一个旋转门锁。既然我们在遍历阻塞链时一次必须获取两个锁，这就产生了讨厌的锁排序问题。Jeff 在那原来的实现中用一个优美的方法解决了这个问题，他在 `turnstile_interlock()` 中解释了他的解决方案：

```
/*
 * 在实施优先级继承时，我们在获取等待者的线程锁的同时，必须同时也获取线程锁拥有者的锁。
 * 如果这两个线程锁都是旋转门锁，就有可能导致死锁：在已经持有锁 L1 并试图获取锁 L2 时，
 * 一些无关的线程可能会向另外一个阻塞链请求实施优先级继承，它们会获取 L2 然后
 * 申请 L1。容易想到的解决办法是用 lock_try() 作拥有者的锁的申请尝试。可是这个办法不能
 * 解决问题，因为它有可能导致活锁(live lock)：每个线程都持有一个锁，同时又试图取得另
 * 外一个。结果失败了，放弃，重试，如此永远循环下去。为了防止活锁问题，我们必须在其中定
 * 义一个优胜者，即，对这些旋转门锁链，任意制定一个有序序列。为简单起见，假设它们以
 * 虚地址为序。当 L1 的地址小于 L2 时，序列是 L1, L2。这样，当一个线程拥有 L1 并试图申请
 * L2 时，它会进入自旋等待直到获得 L2。而当一个线程拥有 L2 并试图申请 L1 时，它应该直接
 * 返回失败，并且释放 L2。而且，失败的线程只能在优胜线程拿到 L2 之后才能重新申请 L2。
 * 为了保证这一点，失败线程在放弃 L2 时，必须先申请 L1，然后进入自旋等待直到优胜线程
 * 结束操作。更为复杂的是，在我们试图获取锁持有者的线程锁时，它的指针可能会指向
 * 另外一个锁。这时候，我们只好回退所有的状态，从头再来。
 */

```

锁排序问题为实现用于内核同步对象的优先级继承增加了难度。不幸的是，内核级的优先级继承只解决了优先级倒置问题的一个方面。

仅仅在内核级提供优先级继承显然是不够的，要构建一个实时多线程系统，不仅在内核态的同步原语需要优先级继承，在用户态的同步原语中同样也需要。我们决定在 Solaris 8 中引入用户态同步原语。我们指派了一个工程师来解决这个问题（并让那些精通调度系统和同步系统的工程师给予指导），1999 年 10 月，新功能被加进去了。

几个月后——在 1999 年 12 月——一位同事遇到了系统崩溃故障。我一检查，发现这很明显是我们所实现的用户态优先级继承中的问题。当我深入理解这个 bug 时，我开始意识到，这不是一个简单问题，事实上，这是设计缺陷。嗯，酒里闻出一股污水味了。

在解释这个 bug 以及它所揭示出来的设计缺陷之前，有必要讨论一下调试 bug 的方法论。作为软件工程师，一个必备技能就是，能分析复杂系统为什么失效，并准确把问题表述

出来。对任何足够复杂的系统，失效分析都是需要科学取证的，它应当以系统失效时的快照作为依据。实际上，这种快照非常重要，它有一个专用术语，叫“核心转储”(Core Dump)。

这种事后分析有别于一般情况下的现场调试。现场调试针对一个正在运行的实际系统(当然，调试时它被暂停)，用户可以根据自己的推断，在需要的地方加断点。事后分析则不然，人们只能根据失效时的状态截面来推断发生了什么事情。从这个意义上讲，与现场调试相比，事后分析并不算一种完备的调试方法(有些bug的状态根本不能通过状态快照推断出来)，不过，很多bug也无法通过现场调试得到重现，在这种情况下，只有作事后分析了。

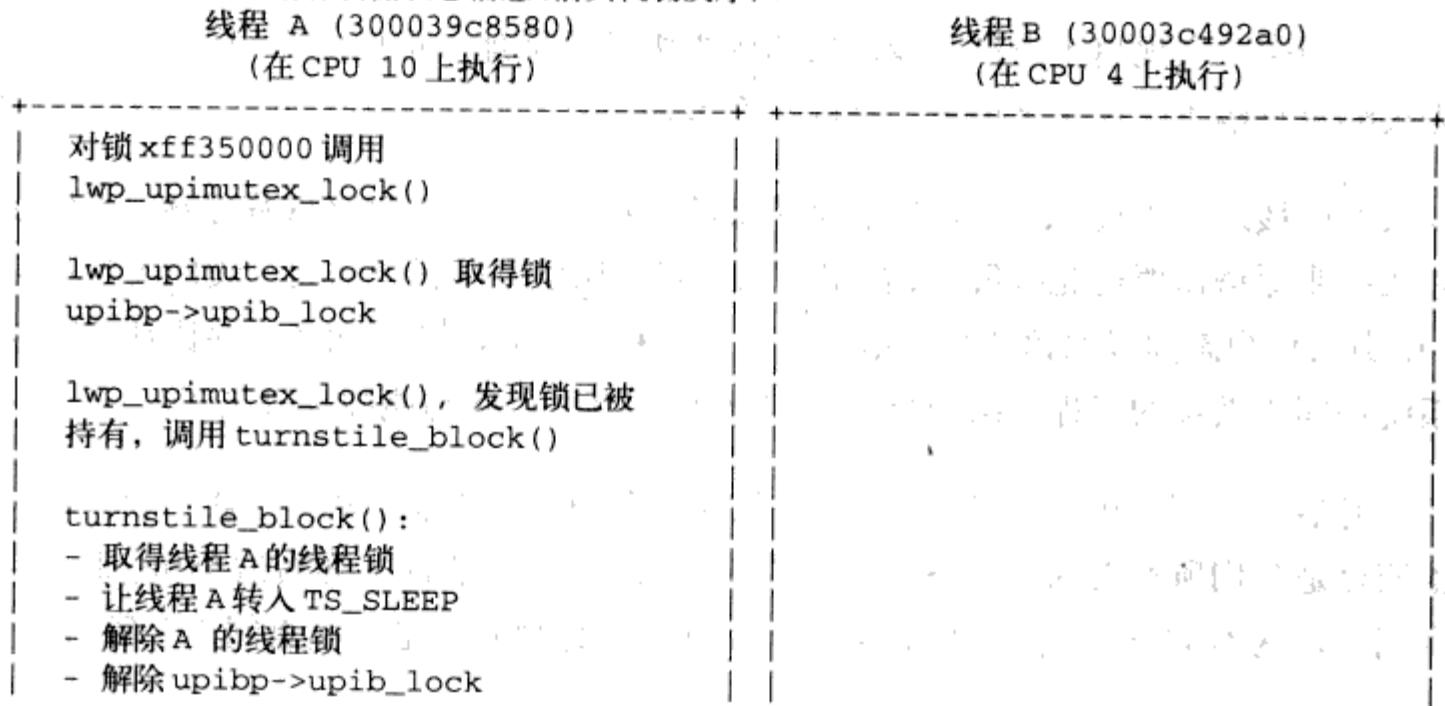
而且，事后分析可以依赖的东西更少，这也要求分析者有更加缜密的思考，要不断提出假设然后验证想法。因此，培养事后分析技能可以有效地提高现场调试能力。

最后，因为系统快照是静态的，我们可以跟同事讲述一个自己的严谨思考结果，我们也可以并行地进行分析，独立地得出结论。就算是我们的分析结论不能被其他人证实，它也有自己的价值，因为这样能让我们清楚地陈述思维逻辑中的漏洞。简而言之，事后分析应当是我们技术能力中的一个重要部分，每一个严肃的软件工程师都应该培养这种能力。

现在我们知道了什么是事后分析，而且也清楚这种分析可能不算全面，或许他根本就是不全面。接着来看看我草就的bug分析，这是从我最初的bug报告中原文摘录的(注3)：

[bmc, 12/13/99]

如下时间序列可以解释转储状态信息(箭头代表次序)：



注3：“优美的bug报告”，谁有兴趣？

```

|   - 调用 swtch()
|   :
|   +-----+
|   | 0xff350000 的持有者释放锁, 显式地将其传给线程 A。
|   | (此处将 upi_owner 设为 c8580)
|   +-----+
|   :
|   从 turnstile_block() 返回
|   对锁 xff350000 调用函数
|   lwp_upimutex_lock()
|   lwp_upimutex_lock() 取得
|   upibp->upib_lock
|   发现 A 持有锁, 调用 turnstile_block()

调用 lwp_upimutex_owned(), 检查
锁是否传递
lwp_upimutex_owned() 试图取得
upibp->upib_lock
upibp->upib_lock 被 B 持有
通过 mutex_vector_enter(), 调用
turnstile_block()

turnstile_block():
    - 取得 B 的线程锁
    - 将线程 B 转入 TS_SLEEP,
    根据 xff350000, 将 B 的 wchan 设为
    upimutex
    - 试图提升 xff350000 持有者线程 A 的
    优先级
    - 获取线程 A 的线程锁
    - 调整 A 的优先级
    - 解除线程 A 的线程锁

    - 获取 A 的线程锁
    - 试图提升 upibp->upib_lock 持有者
    线程 B 的优先级
    - 取得 B 的线程锁
    - 调整 B 的优先级
    - 解除 B 的线程锁
    - 发现 B 的 wchan 不等于 NULL, 试图继续

优先级继承
- 在 B 的 wchan 上调用 SOBJ_OWNER()
- 发现 B 的 wchan 的拥有者是线程 A,
    系统崩溃异常。原因, 死锁。
    在阻塞链上死循环

```

上述序列表明, 问题发生在 turnstile_block()里

```

    THREAD_SLEEP(t, &tc->tc_lock);
    t->t_wchan = sobj;
    t->t_sobj_ops = sobj_ops;
    ...
    /*
     * 从头到底遍历阻塞链, 直到没有优先级倒置, 将优先级传承给遇到的每个节点
     */
    while (inverted && t->t_sobj_ops != NULL && ...

```

```

        (owner = SOBJ_OWNER(t->t_sobj_ops, t->t_wchan)) != NULL) {
        ...
    }
    (1) --> thread_unlock_nopreempt(t);
    /*
     * 此处, "t" 可能不是当前线程, 所以, 以后用 curthread 来代替 "t"
     */
    if (SOBJ_TYPE(sobj_ops) == SOBJ_USER_PI) {
    (2) --> mutex_exit(mp);
    ...
}

```

在(1)处，我们解除了被阻塞线程的线程锁，这发生在(2)处解除 `upibp->upib_lock` 之前。从(1)到(2)，我们违反了 `SOBJ_USER_PI` 的一个定则：当休眠于一个 `SOBJ_USER_PI` 类型的锁时，不能持有核心态的锁。否则，会发生系统崩溃。

理解上述分析需要先知道一些术语：

`upibp`

与用户态优先级继承锁相关联的内核结构指针；`upib_lock` 是保护这个结构的锁。

`t_wchan`

线程结构的一个成员。当线程被阻塞时，它会有一个指针指向同步原语。(注 4)

`SOBJ_TYPE`

一个宏，它接受同步原语矢量作为参数，然后返回一个指代其类型的常数。

`SOBJ_USER_PI` 常数代表用户态优先级继承锁。

问题的关键在于，虽然在用户态，我们通常会记录跟锁关联的信息，比如，是否有等待者。在用户态，这些信息常完全是内核提供的。(在有些情况下，等待者的标志位就不甚可靠，只有内核才清除这一点。)

为了在用户态实现优先级继承，我们必须更加清晰地掌握拥有者信息。而且就像在核心态一样，我们也必须一步步跟踪同步原语的拥有者的信息。这就意味着，当我们在 `turnstile_interlock()` 内做复杂的线程锁操作时，我们不能从用户态存储空间获知拥有者信息。这里令人讨厌的是为了查询用户态锁的拥有者信息，我们要维护一个核心态结构；这个结构本身又必须被另外一个核心态锁保护，那个锁自己为了避免优先级倒置，又必须实现优先级继承。

这就产生了一个未曾预料的死锁问题：必须先获取接着放弃一个核心态锁，才能获取并放弃一个用户态锁。有时候一个线程拥有核心态锁却想申请用户态锁，有时候线程却可能拥有用户态锁同时又想申请核心态锁。这个阻塞链可能会构成一个环，最终导致出现

注 4： `wchan` 代表等待渠道，这是源自贝尔实验室早期 UNIX 系统的一个术语。而且几乎可以肯定，它是 Multics 操作系统中事件渠道对象的一个简化版。

系统崩溃异常。这就是上面我们分析的情况：线程 A 拥有用户态锁，想要申请核心态锁（upib_lock），线程 B 拥有核心态锁却想要用户态锁，死锁就这么发生了！

一旦理解了这个问题，重现它就变得不费吹灰之力了。几分钟功夫，我就能写出一个测试用例，让系统以同样的方式崩溃。（顺便提一句，这种感觉是软件工程里最值得欣慰的事情之一。事后分析一个系统失效，发现问题可以很容易重现，写一个测试用例验证想法，然后看到系统灰飞烟灭。这种感觉真是无与伦比，绝对是软件开发中的“全垒打”。）

接着我开始考虑如何着手解决这个问题。然而，问题的严重性和宽松的收工时间表提醒我，跟在家里的 Jeff 先讨论一下。当我们深入讨论这个问题时，我们发现了一个潜在的问题——居然找不到一个没有副作用的好办法。事实上，我们讨论得越深入，问题看起来就越发困难。我们意识到，一开始我们就犯了错：我们既低估了它的难度，也找错了解答方向。

更严重的是，Jeff 和我都开始意识到这里还有一个表述上的陷阱。我们知道，当线程阻塞于核心态锁并且发生假性死锁（false deadlock）时，系统会发生崩溃异常。但是如果发生假性死锁时，一个线程是阻塞于用户态锁，那应该如何处理呢？我们很快做出决定，（并通过测试用例证实），这种情况下请求获得用户态锁将会返回 EDEADLK 错误。因为内核很容易就会发现这个“死锁”是由用户态同步原语导致的，这可以归结为一个应用程序的 bug。

所以，这种应用程序失效，是由于它调用 pthread_mutex_lock 失败造成的，这个错误比系统崩溃还要严重。因为，如果应用程序不检查 pthread_mutex_lock 的返回值，它就有可能错误地以为它已经拥有了对某段数据的锁定。而实际上，它根本没有获得这个锁。

这种问题在运行现场几乎无法调试，但我们一定要解决它。

怎么解决呢？我们一直认为这个问题很难，因为我们总是试图避免引入那个核心态锁。我曾经把引入核心态锁作为这个问题的约束条件之一，显然谁都很不愿意接受这种约束条件。每次我们中有人试图提出一个避免核心态锁的方案，另一个人就会发现这个方案在某些情况下是不可行的。

想遍了所有的办法之后，我们不得不出结论，引入核心态锁是解决用户态优先级继承的必要条件。我们的关注点也从竭力避免转到及时侦测。

在两种情况下需要我们侦测，系统崩溃异常和假性死锁。假性死锁情况下侦测很容易办

到，也很容易处理。因为那时候，我们总是位于阻塞链的末尾；而且当前持有的那个导致死锁的锁，就是在核心态传给turnstile_block的参数。既然我们已经把优先级过继给整个阻塞链，现在一经发现，只需要中断这一过程即可。这就是我在turnstile_block里加那段令人费解的注释的目的。接下来的两行代码也正是做这件事情的。（传给turnstile_block的核心态锁被保存于临时变量mp中）

相比之下，系统崩溃异常的情况就显得比较棘手了。提示一点，拥有用户态同步对象的那个被阻塞线程，此时正打算获取核心态锁。我们可能会采用同样的办法解决这个问题，推理如下：如果死锁在当前线程上发生，并且阻塞链上的最后一个线程被一个用户态对象阻塞，那这个死锁必定是假性的。也就是说，我们可能在更加通用层面来一揽子解决所有。可惜，这个简明的做法被证明是行不通的。因为它忽略了应用程序真有可能会死锁（由于程序bug）。在这种情况下，必须返回EDEADLK。看样子我们需要一个更加精致的解决办法。

让我们回过头来观察。如果一个阻塞链从一个阻塞于核心态同步对象的线程过渡到阻塞于用户态同步对象的线程，就不难推知，这是我们惟一能到达的状态。（注5）而且可以知道，我们同时正位于另外一个不可占先线程的中间（因为那个线程正在执行turnstile_block，这个函数会禁止线程占先）。所以，解决办法就是等待，直到锁的状态变了。接着，重新开始令人眼花缭乱的优先级继承操作。

这是解决方案给出的代码：（注6）

```
/*
 * 现在我们已持有拥有者的锁。如果我们从非 SOBJ_USER_PI 进入
 * SOBJ_USER_PI，那么可以断定线程当前正处于 TS_SLEEP 状态，而且它持有
 * mp。不难知道，这是一个中间状态。在 mp 的拥有者被 SOBJ_USER_PI
 * sobj 阻塞之前，会释放 mp。因此，可以自旋等待 mp 被释放。于是，就像在
 * turnstile_interlock() 失败的那个情况一样，我们会重新开始优先级
 * 继承的整个过程。
 */
```

注 5：就像大多数操作系统一样，Solaris 从不在持有核心态锁时执行用户态指令，也从不从核心态子系统申请用户态锁。所以，剩下的可能性就只能是，在持有核心态锁时试图再申请持有用户态锁。

注 6：在我们报告这个案例时，这段有关turnstile_loser_lock的代码还没有写出来，它实际上是在解决另外一个问题时被加进来的。那个问题让我们的思绪被熔炼了整整四天，的确值得花一整章来表述，哪怕仅仅是为了纪念Jeff选取的伟大名字：“两败俱伤”。当Jeff宣布这个问题之后，我很快就在实际应用中看到了它的变种，戏称之为“一损俱损”。不管是两败俱伤还是一损俱损，失败者都得从头再来。

```

    if (SOBJ_TYPE(t->t_sobj_ops) != SOBJ_USER_PI &&
        owner->t_sobj_ops != NULL &&
        SOBJ_TYPE(owner->t_sobj_ops) == SOBJ_USER_PI) {
        kmutex_t *upi_lock = (kmutex_t *)t->t_wchan;
        ASSERT(IS_UPI(upi_lock));
        ASSERT(SOBJ_TYPE(t->t_sobj_ops) == SOBJ_MUTEX);
        if (t->t_lockp != owner->t_lockp)
            thread_unlock_high(owner);
        thread_unlock_high(t);
        if (loser)
            lock_clear(&turnstile_loser_lock);
        while (mutex_owner(upi_lock) == owner) {
            SMT_PAUSE();
            continue;
        }
        if (loser)
            lock_set(&turnstile_loser_lock);
        t = curthread;
        thread_lock_high(t);
        continue;
    }
}

```

这个问题一解决，我们总算松了口气。不幸的是，接下来的压力测试却暴露了更大的麻烦。老实说，这一次，我们还真未必有把握。

这次的症状与过去的不同，操作系统既没有抛出崩溃异常，也没有返回错误，而是出现了更加棘手的症状——它挂住了。拿到转储文件并加以分析之后，结论出来了。有一个线程，在试图从 `turnstile_block()` 获取线程锁时，死锁了。因为 `turnstile_block()` 通过调用 `mutex_vector_exit()`，又递归地调用了自己。`mutex_vector_exit()` 函数是用来当有等待者时释放互斥锁的。有了这些分析，问题就清楚了——抓住了问题的关键。

回想一下，前面我们提到，为了获取或者抛弃一个带优先级继承的用户态锁，我们不得不无奈地先申请一个令人讨厌的核心态锁，然后抛弃之。在阻塞于用户态锁时，一旦完成优先级传承，我们必须抛弃核心态锁。这实际上也是线程通过 `switch()` 函数让出 CPU 之前所做的最后一件事。（在我前面的分析注释里，那段被引号引起的代码就是完成这个工作的；带标注(2)的代码是用来释放核心态锁的）

在我们处理用户态锁的阻塞时，如果还有一个线程阻塞于核心态锁，我们需要在放弃线程锁的同时唤醒那个等待中的线程。唤醒一个等待者需要获得旋转门表中与同步原语对应的线程锁。接着，为了恢复那些传承过来的优先级，我们还需要取得这个锁的前一个持有者的线程锁。这个持有者就是当前线程。

这下问题就来了。在被阻塞之后，系统会从 `turnstile_block()` 函数进入

`turnstile_pi_waive()` 函数，去恢复优先级继承。特别是当前线程的线程锁不再指向当前 CPU 的锁，而是指向即将阻塞我们的用户态锁所对应的旋转门表。在这种情况下，如果核心态锁和用户态锁恰好指向同一个旋转门表项（在我们刚开始做的那次失效分析里就是这种情况），那么，在 `turnstile_lookup()` 函数里取得的旋转门锁和在 `turnstile_pi_waive()` 函数里取得的线程锁将是同一个锁。这就产生了单线程死锁。就算这两个锁不是对应同一个旋转门表项，万一它们恰好没有遵循 `turnstile_interlock()` 所指定的锁序列，我们还是会遇到经典的 AB/BA 死锁。这种结果同样无法接受。

现在我们明白到底是怎么回事了，这个问题看起来很难处理。前面我们遇到的问题是，线程被阻塞之前就放弃了核心态锁。绕过核心态锁的做法已经被证明是不可行的。一旦核心态锁被申请到手，那么只有在给整个阻塞链完成优先级继承之后才能放弃它。

现在我们不得不反思那些最基本的前提假设。我们能不能在 `turnstile_block()` 里调换锁的次序？这样的话，我们就可以在将当前线程置为休眠之前完成优先级传承工作。（结论是，不能。这会在局部导致优先级倒置。）我们能不能引入状态标识，使得从 `turnstile_block()`，借由 `mutex_vector_enter()` 调用 `turnstile_pi_waive()` 而不会产生死锁？（结论还是否定的。这个办法无法表述多线程的死锁。）

在我们不断提出构想时，其中的缺陷也很容易被我们发现。而且，越是深入思考问题本身，就越感觉到那些解决方案毛病越多。

大家开始陷入绝望。更让人懊恼的是，这仅仅是为了在已经完美的体制下引入一层用户态优先级继承。结果，娄子越捅越大，我们在泥沼中也越陷越深。

我们只好到附近的咖啡馆找点慰藉，这时灵光闪现了。我们可能对用户态优先级继承这个问题的面考虑得太广了。为什么不把问题从抽象层面具体化呢？比如，对旋转门表分而治之。我们可以把用户态优先级继承相关的核心态锁放在旋转门表的这半边，然后把其他的锁放在另外半边。

这样就能保证在 `turnstile_block()` 函数调用 `swtch()` 的前一刻所要放弃的锁，肯定与阻塞线程的锁对应不同的旋转门表项。而且，进一步约定，任何对应用户态优先级继承状态的核心态锁，它们在旋转门表中的虚地址比其他的锁都要小。这样，`turnstile_interlock()` 里指派的锁顺序就可以很好地得到遵循，单线程和多线程问题就可以一并处理了。

一方面，这个办法把一种专有的锁引入了通用的旋转门系统，显得有点草率。这意味着在一般性的系统中需要了解某种特定类型的锁（保护内核的，用户级别优先权继承状态

的锁)。另一方面，它确实有效、直接而且改动风险很低。考虑到，在这个长达两年的开发周期里，我们正看到大功告成的希望，这个方案就显得尤其重要。坦率地说，我们也没有其他更好的办法。如果有的话，肯定是用更好的了。

Jeff 和我边喝咖啡边谈论这个方案，他接着就回去就给我们那段貌似简单的代码加上注释了。我们的解决方案在优雅性上是有争议的。凭心而论，我真希望在注释里加入一些袒露心迹的形容词，比如，“粗糙”、“令人不快”甚至是“平庸”。(注 7) 不过，Jeff 的做法让我颇感意外，我认为这是整个 Solaris 里最好的注释，可能也是所有软件里最好的。

```
/*
 * 旋转门表由两部份构成：下半部分储存 upimutextab[] 锁；上半部分储存其他锁。之所以
 * 这么设计是因为 SOBJ_USER_PI 锁会导致一个特殊问题。调用 turnstile_block() 时，
 * 它所申请的 upimutextab[] 锁只能在当前线程阻塞于 SOBJ_USER_PI 锁，并且将优先级继
 * 承给整个阻塞链之后才能被释放。此时，调用者的 t_lockp 是一个旋转门锁。当 mutex_exit()
 * 发现 upimutextab[] 锁有等待者，它就会唤醒这个等待者。这就使得我们必须给锁指定先后
 * 顺序。一开始是在 mutex_vector_exit() 里申请 upimutextab[] 的旋转门锁，这个函数后
 * 来会调用 turnstile_pi_waive()，并在这个函数里取得调用者的线程锁。此处，这个线程锁
 * 其实就是 SOBJ_USER_PI 的旋转门锁。一般来说，当我们一次持有两个旋转门锁时，锁须按地址
 * 大小排序。所以为了避免在 turnstile_pi_waive() 里造成死锁，我们必须保证 upimutextab[]
 * 在旋转门表里的地址要比其他锁来得小。如果你觉得这个问题稀疏平常？愿闻高见。
 */
#define TURNSTILE_HASH_SIZE      128          /* must be power of 2 */
#define TURNSTILE_HASH_MASK      (TURNSTILE_HASH_SIZE - 1)
#define TURNSTILE_SOBJ_HASH(sobj) \
    (((ulong_t)sobj >> 2) + ((ulong_t)sobj >> 9)) & TURNSTILE_HASH_MASK
#define TURNSTILE_SOBJ_BUCKET(sobj) \
    ((ISUPI(sobj) ? 0 : TURNSTILE_HASH_SIZE) + TURNSTILE_SOBJ_HASH(sobj))
#define TURNSTILE_CHAIN(sobj) turnstile_table[TURNSTILE_SOBJ_BUCKET(sobj)]

typedef struct turnstile_chain {
    turnstile_t    *tc_first;    /* 哈希链上的第一个旋转门 */
    disp_lock_t    tc_lock;     /* 哈希链自己的锁 */
} turnstile_chain_t;
turnstile_chain_t    turnstile_table[2 * TURNSTILE_HASH_SIZE];
```

Jeff 此处的措辞显然远胜过我原来诚惶诚恐的做法。显然，我们不是无计可施才推出这个解决方案。恰恰相反，现在的做法是解决这个极具挑战性问题的必由之路。有人可能认为我言过其实，因为这种做法属于修修补补。不过摆在眼前的事实是，在以后长达七年的时间里，没有人能够想出更好的方案。现在看来，以后也很难会有。至少对我来说，

注 7：这倒是告诉我们一个小技巧：如果你想在一堆堆代码中找到功能主体，可以试试在里面检索一下诸如“XXX”、“待修正”这样的标志性词语。

这些代码优美之极，其他小节无关紧要。

故事到此接近尾声了。最后，我们把修正过的代码集成到系统里，并按时发布了。其中的几条经验教训仍值得我们关注：

要尽早付诸实施

虽然在设计和实现阶段，我们都花时间思考过这些问题，但我和Jeff都没有预见到后来遇到的问题。甚至当我们在Bug中有机会再次面对这些问题，那些深藏其内部的隐患仍旧不能被我们一眼洞穿。直到最后面对面遭遇到了它们了，我们才得到了真正地领悟。

要千锤百炼

如果当初工程师们能够尽早对产品进行压力测试，而不必全靠功能测试，我们也许能够早点遇到这些问题。作为软件工程师，我们应该自己负责做压力测试。有人不认同这种做法，认为自己写的测试用例相比精妙的产品来说，过于粗糙。那他们的软件是经不起时间的考验的。我不是主张不需要测试工程师或者索性撤掉测试部门，而是认为，测试工程师写的测试用例应当是开发者测试用例的补充，而不是倒过来。

专注于边际情况

为什么我认为年轻的软件工程师拼了命也要学会调试复杂的系统呢？因为这门技术可以让人终身收益。它教会我们这种能力：在分析问题并提出解决办法时，应该时刻想着它们哪儿还有问题，而不是想着它也许管用。这就是专注于边际。在构思一个新软件时，我们用不着提醒自己我们的设计怎么管用；相反，我们应该竭力排除那些可能会让软件没法用的隐患。我不想提倡在写代码时做过度分析，只想提醒各位，你们写出的代码里可能会出现bug，而且它们可能会让上层设计推倒重来。

遵循了这些原则，人们就会自然而然地习惯于在项目一开始时就先实现最困难的部分，然后让它们经受考验，从而打好一个持续可用的基础。虽然这样做还是有可能避免不了出现烂疮恶疽。但如果能坚持，就可以尽可能早地发现最致命的部分，赶在还能做设计变更的时候处理掉，从而避免整桶酒被那一匙污水给污染了。

第 23 章

MapReduce 分布式编程

Jeffrey Dean & Sanjay Ghemawat

本章将介绍 MapReduce，它是一种分布式并行编程模型。通过使用 MapReduce，你可以很容易地在廉价的机器上运行大规模的数据处理任务。MapReduce 提供了一个简单易用的 API，使得编写并行程序变得非常容易。通过学习本章，你将了解到如何使用 MapReduce 来解决各种各样的问题，包括文本分析、数据挖掘、机器学习等。

本章描述了 MapReduce 的设计与实现，这是一种用于解决大规模 (large-scale) 数据处理问题的编程系统。MapReduce 的开发初衷是为了简化 Google 大规模计算的开发工作。MapReduce 程序自动地在廉价机器所组成的大型集群上并行执行。运行时系统所关心的细节是：输入数据的划分，机群上程序执行的调度，机器故障的处理，机器间通信的管理。即使是毫无并行和分布式系统经验的程序员，也可以很容易地利用大型分布式系统的资源。

激动人心的示例

假设你有 200 亿个文档，想要统计出每个单词在所有文档中出现的总次数。假设每个文档的平均大小是 20 KB，那么一台机器读完 400TB 的数据需要四个月左右。假设我们愿意等待的时间足够长，机器内存足够大，那么相关实现代码十分简单。示例 23-1（本章中所有示例代码均是伪码）展示了一种可行算法。

示例 23-1：简单，非并行的单词计数程序

```
map<string, int> word_count;
for each document d {
```

```

for each word w in d {
    word_count[w]++;
}
...
... 将word_count 保存在永久性存储中 ...

```

加速计算的方法之一是在每个独立文档上并行执行相同的计算，如示例 23-2 所示。

示例 23-2：并发的单词计数程序

```

Mutex lock; // 保护word_count
map<string, int> word_count;
for each document d in parallel {
    for each word w in d {
        lock.Lock();
        word_count[w]++;
        lock.Unlock();
    }
}
...
... 将word_count 保存在永久性存储中 ...

```

上述代码针对问题的输入方面很好地进行了并行化处理。实际上，线程的启动代码原本是稍微有些复杂的，但通过使用伪码，我们隐藏了许多实现细节。示例 23-2 的问题之一就是使用单个全局数据结构来追踪生成的计数值。结果，在 word_count 数据结构上的严重锁竞争可能成为性能瓶颈。通过将 word_count 数据结构划分成许多桶，每个桶带一把独立的锁，就可以解决这个问题，见示例 23-3。

示例 23-3：基于划分存储器的并行单词计数程序

```

struct CountTable {
    Mutex lock;
    map<string, int> word_count;
};

const int kNumBuckets = 256;
CountTable tables[kNumBuckets];
for each document d in parallel {
    for each word w in d {
        int bucket = hash(w) % kNumBuckets;
        tables[bucket].lock.Lock();
        tables[bucket].word_count[w]++;
        tables[bucket].lock.Unlock();
    }
}
for (int b = 0; b < kNumBuckets; b++) {
...
... 将tables[b].word_count 保存在永久性存储中 ...
}

```

程序依旧非常简单。程序性能的伸缩（scale）范围不能超过单台机器上的处理器数目。大部分能提供的机器，其处理器数目都不会超过 8 个，因此，即使是完美的伸缩方式，

这种方法完成任务还是需要数周时间的。此外，我们一直都忽略了诸如输入数据的存放位置，机器读取输入数据的速度这类问题。

更进一步的伸缩需要我们将数据和计算分布到多台机器上。目前，我们事先假设机器不会出现故障。提高伸缩性的方式之一就是在网络计算机组成的集群上启动许多进程。我们将会有许多输入进程，每个进程负责读入和处理所有文档的子集。同时，我们也将会有许多输出进程，每个进程负责管理一个 word_count 桶。示例 23-4 展示了相关算法。

示例 23-4：基于划分处理器的并行单词计数程序

```
const int M = 1000;           // 输入进程数
const int R = 256;            // 输出进程数
main() {
    // 计算分配给每个进程的文档数
    const int D = number of documents / M;
    for (int i = 0; i < M; i++) {
        fork InputProcess(i * D, (i + 1) * D);
    }
    for (int i = 0; i < R; i++) {
        fork OutputProcess(i);
    }
    ... 等待所有进程结束 ...
}

void InputProcess(int start_doc, int end_doc) {
    map<string, int> word_count[R]; // 每个进程一个单独的表格
    for each doc d in range [start_doc .., end_doc-1] do {
        for each word w in d {
            int b = hash(w) % R;
            word_count[b][w]++;
        }
    }
    for (int b = 0; b < R; b++) {
        string s = EncodeTable(word_count[b]);
        ... 将 s 发送给输出进程 b ...
    }
}

void OutputProcess(int bucket) {
    map<string, int> word_count;
    for each input process p {
        string s = ... 从 p 读入消息 ...
        map<string, int> partial = DecodeTable(s);
        for each <word, count> in partial do {
            word_count[word] += count;
        }
    }
    ... 将 word_count 保存到永久性存储中 ...
}
```

尽管这种方法在工作站的网络上伸缩性非常好，但它却更加复杂，更加难以理解（即使

我们隐藏了列集 (marshaling) 和散集 (unmarshaling) 的细节以及启动和同步了不同的进程)。它也没有完美地处理好机器故障。为了处理故障问题，我们将会扩展示例 23-4 中的程序，用于重新执行任务完成前发生故障的进程。当重执行一个输入进程时，为了避免产生双重计数的数据，我们将会用输入进程的族名 (generation number) 来标注每块中间数据，并且修改输出处理程序。这样，通过使用这些族名就可以避免产生重复。正如你所想像的，添加这种故障处理支持是一件更为复杂的事情。

MapReduce 编程模型

如果对比示例 23-1 和示例 23-4，你会发现看似简单的单词计数任务，实际上是掩藏于大量并行管理的细节之下。如果可以设法将原有问题的细节与并行化的细节分开，我们就可以开发出一个通用的并行库或系统，它不仅适用于单词计数问题，同样也适用于其他大规模数据处理问题。我们将要使用的并行模式是：

- 对于每条输入记录，从中提取出我们所关心的一组 key/value 对。
- 对于提取出来的每个 key/value 对，将其与共享相同 key 的其他 value 进行合并（在进程中可能对 value 进行过滤、聚合或转换）。

让我们重写程序来实现特定的应用逻辑 (application-specific logic)，它可以分别统计每个文档中单词出现的频率，并把所有文档的计数分别相加。仅需要两个函数，我们称为映射 (Map) 和化简 (Reduce)。最终程序参见示例 23-5。

示例 23-5：单词计数问题划分为映射和化简

```
void Map(string document) {
    for each word w in document {
        EmitIntermediate(w, "1");
    }
}

void Reduce(string word, list<string> values) {
    int count = 0;
    for each v in values {
        count += StringToInt(v);
    }
    Emit(word, IntToString(count));
}
```

示例 23-6 中展示了一个简单的 Driver 程序，它会使用上述子程序在单机上完成所要求的任务。

示例 23-6：映射和化简的 Driver

```
map<string, list<string>> intermediate_data;

void EmitIntermediate(string key, string value) {
    intermediate_data[key].append(value);
}

void Emit(string key, string value) {
    ...将 key/value 写入最终的数据文件...
}

void Driver(MapFunction mapper, ReduceFunction reducer) {
    for each input item do {
        mapper(item);
    }
    for each key k in intermediate_data {
        reducer(k, intermediate_data[k]);
    }
}

main() {
    Driver(Map, Reduce);
}
```

对于每个输入记录，调用一次 Map 函数。Driver 代码将 Map 函数所产生的所有中间 key/value 对集合在一起。之后，对于每个中间 key 所对应的中间 value 列表，调用一次 Reduce 函数。

现在我们回过头来考虑一个运行在单机上的实现。不过，按照目前这种功能划分方式，我们可以在不影响 Map、Reduce 函数的特定应用逻辑的情况下，就能改变 Driver 程序的实现，使其支持分布式、自动并行化及容错性。此外，Driver 程序是独立于 Map 和 Reduce 函数所实现的特定应用逻辑。因此，将其他的 Map、Reduce 函数运用到相同的 Driver 程序中（重用），就可以解决不同的问题。最后注意，理解实现特定应用逻辑的 Map、Reduce 函数，与理解示例 23-1 中的简单顺序执行代码相比，难度几乎是一样的。

其他 MapReduce 示例

我们接下来要探讨一种更为复杂的 Driver 程序实现，它可以在大规模的集群计算机上自动地运行 MapReduce 程序。但是，首先让我们考虑几个其他问题，看看如何使用 MapReduce 来解决他们。

分布式 grep (*Distributed grep*)

如果某行内容与给定的正则表示模式匹配，Map 函数则将其输出。Reduce 函数是一个恒等函数 (identity function)，仅将给定的中间数据拷贝到输出。

反向网络链接图 (*Reverse web-link graph*)

如果 URL1 上的网页存在一个到 URL2 的超链接，那么结点 URL1 到结点 URL2 则形成一条有向边，由这类边所组成的图称为前向网络链接图 (forward web-link graph)。反向网络链接图就是将前向网络链接图中边的方向反转，其他保持不变。我们可以很容易地运用 MapReduce 技术来构建一个反向网络链接图。对于 source 文档中所发现的每个 URL 链接 target，Map 函数都要输出 $\langle \text{target}, \text{source} \rangle$ 对。Reduce 函数把所有与指定的 target URL 相关的 source URL 关联到一个列表，然后输出 $\langle \text{target}, \text{list of source URLs} \rangle$ 对。

主机关键向量 (*Term vector per host*)

关键向量对一个文档或一组文档中所出现的最重要词汇的频率进行总结，并将其存放在由 $\langle \text{word}, \text{frequency} \rangle$ 对所组成的列表中。Map 函数为每个输入文档（从文档的 URL 中提取出主机名）输出一个 $\langle \text{hostname}, \text{term vector} \rangle$ 对。传给 Reduce 函数的是指定主机上的每个文档的关键向量。Reduce 函数会将这些关键向量相加，去除不常用的关键词，输出一个最终的 $\langle \text{hostname}, \text{term vector} \rangle$ 对。

反向索引 (*Inverted index*)

反向索引是一种数据结构，它存储的是单词与包含单词的文档列表之间的映射（为了精简反向索引数据，文档通常用数字标识符来标识）。Map 函数解析每个文档，输出一个由 $\langle \text{word}, \text{docid} \rangle$ 对所组成的序列。Reduce 函数接受指定单词的所有 docid，将相应的文档 ID 排序，输出一个 $\langle \text{word}, \text{list of docids} \rangle$ 对。所有的输出对组成了一个简单的反向索引。追踪单词在每个文档中的位置，只需要简单地增大计算量。

分布式排序 (*Distributed sort*)

针对特定的关键字，MapReduce 还可以用来对数据排序。Map 函数从每条记录中提取出关键字，输出一个 $\langle \text{key}, \text{record} \rangle$ 对。Reduce 函数原样输出所有的 $\langle \text{key}, \text{record} \rangle$ 对（即恒等 Reduce 函数）。计算依赖于本章后面部分所介绍的划分工具和顺序属性。

许多其他的计算例子都可以容易地用 Map-Reduce 计算表达。对于更为复杂的计算，通常将其用 MapReduce 步骤的序列，或 MapReduce 计算的迭代应用来表达（此处的某个 MapReduce 步骤的输出是其下一个 MapReduce 步骤的输入）。

一旦你开始以 MapReduce 的方式来思考数据处理问题，通常会发现表达他们非常容易。证明手段之一就是：在过去的四年里，Google MapReduce 程序的数量从 2003 年 3 月少量的候选程序（我们那时刚开始设计 MapReduce），迅速增加到 2006 年 12 月底的 6000

多个不同的 MapReduce 程序。这些程序分别由 1000 多名不同的软件开发人员所编写，大部分人在使用 MapReduce 之前，从未写过任何并行或分布式程序。

分布式 MapReduce 的一种实现

MapReduce 编程模型的主要优点是，它将预期计算的表达与并行、故障处理等底层细节很好地分开。实际上，对于不同的计算平台，MapReduce 编程模型有着不同的实现方法。正确的选择依赖于具体环境。例如，针对小型的共享内存计算机可能有一种实现，针对大型的 NUMA 多处理器又有另一种实现，而针对更大的网络计算机集合则还有一种实现。

示例 23-6 中展示了支持这种编程模型的一段简单的单机实现代码。本小节描述了一种更为复杂的实现，它适用于在 Google 所广泛使用的计算环境中，运行大规模的 MapReduce 作业。这种计算环境是由大量的廉价 PC 通过交换式以太网，互联而形成的大型集群（参考章末尾的“进一步阅读”）。这种环境中：

- 机器通常是双 x86 处理器，2~4GB 物理内存，运行 Linux 操作系统
- 机器通过廉价的网络硬件互联（通常是 1 Gb/s 的交换式以太网）。将所有机器都放置在架子上，每个架子放置 40 或 80 台机器。所有架子会连接到整个集群的中央交换机。同一个架子上的机器间数据通信的有效带宽是 1 Gb/s。相比之下，中央交换机分配给每台机器的带宽就要小多了（通常每台机器 50~100 Mb/s）。
- 存储介质使用廉价的 IDE 硬盘，直接安装在每台机器上。GFS（参考本章末尾的“进一步阅读”中的《Google 文件系统》）分布式文件系统管理硬盘上所存储的数据。GFS 通过复制为不可靠的硬件提供有效性和可靠性。具体做法：将文件划分成 64MB 大小的 chunk，每个 chunk 通常都会在不同的机器上留有三份拷贝。
- 用户调度系统提交作业。每个作业（job）由一组任务（task）组成，通过调度器映射到集群内的一组有效机器上。

执行概览

通过将输入数据自动地划分成 M 个分块，就可以让 Map 函数调用自动地分布到多台机器。不同机器可以并行处理这些输入分块。通过划分函数（如 $\text{hash}(\text{key}) \% R$ ）将中间 key 的空间划分成 R 块，就可以实现 Reduce 调用的分布。

图 23-1 展示了用户程序调用 MapReduce 函数时所发生的一系列动作（图 23-1 中的编号标签对应下面列表中的数字）。

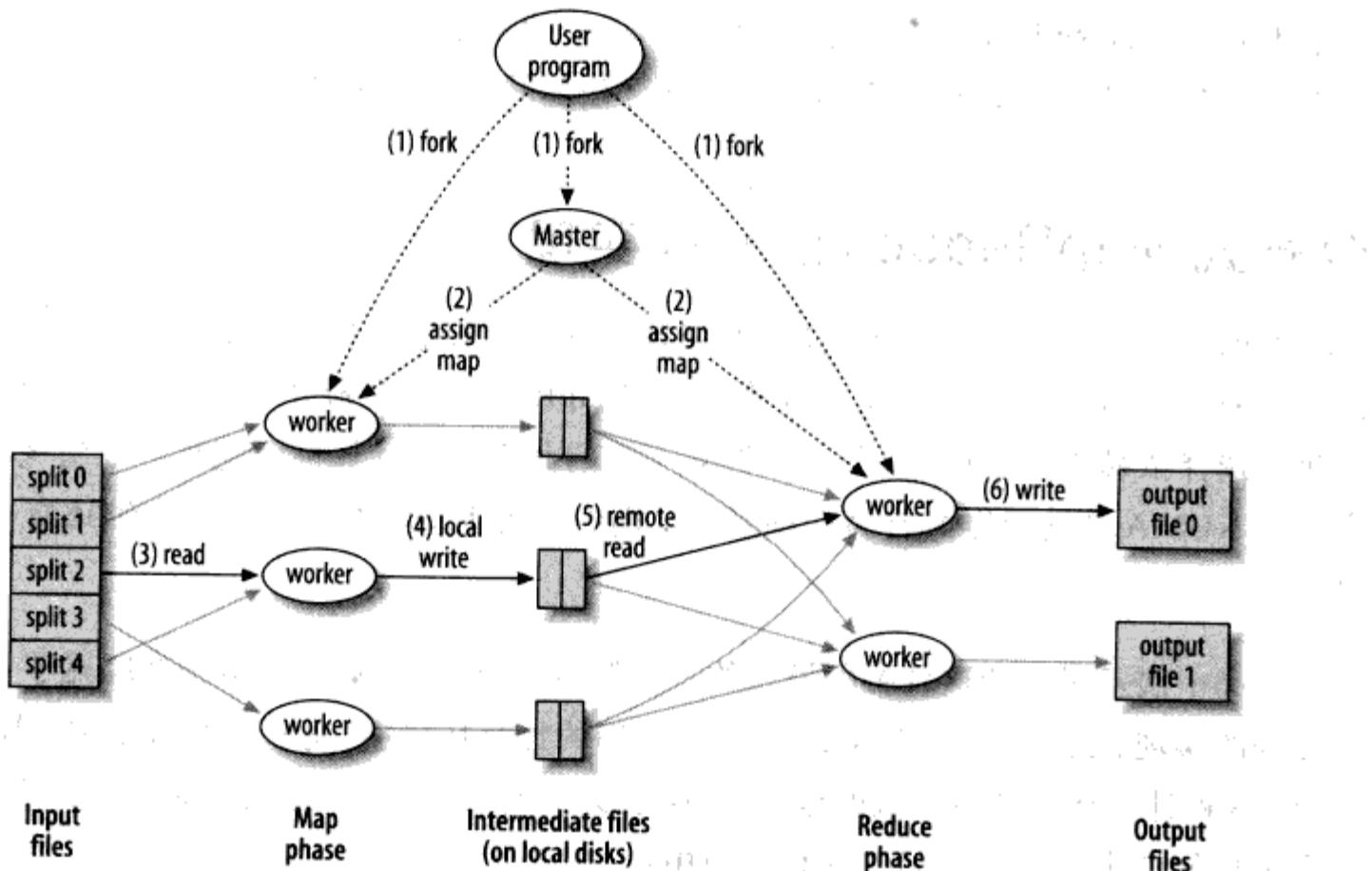


图 23-1：MapReduce 中进程间的关系

1. MapReduce 库首先将输入文件划分成 M 块（通常每块大小是 16 MB ~ 64 MB）。向集群调度系统发出一个请求后，它就在机器集群上启动程序的多份拷贝。
2. 其中有一个特殊拷贝叫做 MapReduce master。master 将剩下的任务分配给一批 map 和 reduce worker。总共有 M 个 map 任务和 R 个 reduce 任务。master 挑选出处于空闲状态的 worker，并给每个 worker 分配一个 map 和 / 或一个 reduce 任务。
3. 分配有 map 任务的 worker 读取相应输入块中的内容，将每个输入记录传给用户自定义的 map 函数。Map 函数产生的中间 key/value 对缓冲在内存中。
4. 缓冲的 key/value 对周期性地写入本地磁盘，通过划分函数划分成 R 个独立的桶。当 map 任务完成时，worker 会通知 master。master 会将这个 map 任务所生成的中间数据的位置信息转发给分配有 reduce 任务的 worker。如果还有 map 任务，那么 master 将把其中一个剩余任务分配到最近空闲的 worker 上。
5. 当 reduce worker 获得进行 reduce 任务所需的中间数据位置后，就会执行远程过程调用，从 map worker 的本地磁盘读取缓冲的中间数据。当 reduce worker 读完所有属于这个 reduce 任务的中间数据后，就会根据中间 key 来对中间数据进行排序，这样所有出现相同中间 key 的地方都会集合在一起。如果中间数据太大，超过 reduce worker 内存的大小，就使用外部排序。
6. reduce worker 会迭代所有有序的中间 key/value 对。对于每个中间 key，将其连同

所对应的中间 value 列表一并传给用户的 Reduce 函数。用户的 Reduce 函数所生成的所有 key/value 对，都会添加到本次 reduce 划分所对应的最终输出文件。当 reduce 任务结束时，worker 会通知 master。如果还有剩余的 reduce 任务，master 会将剩余的 reduce 任务逐个分配给最近空闲的 worker。

当所有的 map 和 reduce 任务都完成时，用户程序中的 MapReduce 函数调用就会返回，将控制权交回给用户代码。此时，MapReduce 作业的输出位于 R 个输出文件中（每个 reduce 任务输出一个文件）。

若干实现细节可以确保 MapReduce 良好地运行于我们的环境中。

负载平衡

MapReduce 作业所拥有的任务数目通常都要比实际拥有的机器数目要多，这意味着 master 会给每个 worker 分配许多不同的任务。当某个机器完成之前所分配的任务后，master 将会给它分配一个新任务。这表明速度较快的机器比速度较慢的机器会分配更多的任务。因此，即使在非均一（heterogeneous）环境中，机器任务分配也可以得到适当的平衡，worker 在整个计算中趋于忙碌有用的工作。

容错性

MapReduce 实现的设计初衷就是要运行分布在成百上千台机器上的作业，因此这个库显然必须要能处理机器故障。

master 会保留 worker 完成 map 和 reduce 任务的状态。master 周期性地给 worker 发送一个 ping 的远程过程调用。如果 worker 在若干次连续的 ping 后没有答复，master 会将其声明成消亡，并将这个 worker 所完成的工作重新分配给其他机器去重新执行。通常 MapReduce 执行的 map 任务数目是 worker 机器数目的 50 倍。因为一台机器发生故障时，50 台独立机器中的任何一台都可以接过这个 map 任务来重新执行，所以恢复非常迅速。

master 把调度状态的所有更新都写入到一个永久日志文件中。如果 master 消亡（非常罕见，因为只有一个 master），集群调度系统会重启 master。新的 master 实例通过阅读日志文件来重建其内部状态。

本地性

输入数据（GFS 管理）存储在执行 map 计算的同台机器或同一个架子上。根据这个事实，我们的 MapReduce 实现节约了网络带宽。对于任何指定的 Map 任务，MapReduce master 会找到输入数据的位置（由于 GFS 的复制功能，往往存在多个位置）。master 把 map 任务调度到任务输入数据拷贝附近的机器上执行。对于使用

数千个 worker 的大规模 MapReduce 作业，大部分输入数据都是直接从本地硬盘读入的。

备份任务

通常，少数掉队者支配了 MapReduce 的运行时间。（掉队者是指某个机器花费很长时间来执行最后的少量 map 或 reduce 任务中的某个任务。）花费很长时间来执行一个任务的原因有两种：要么因为运行代价原本就很昂贵，要么因为运行在一台很慢的机器上。

各种各样的理由都可能导致一台机器变慢。例如，机器可能忙于应付不相关的其他 CPU 密集型（CPU-intensive）进程，或机器可能有一块导致频繁重试读取操作的坏硬盘，那样，磁盘读取速度会变慢 10 或 100 倍。

备份任务可以解决掉队者的问题。当剩下少量的 map 任务时，master 为剩下处理中的每个 map 任务调度出（在空闲的 worker 上）一个备份执行。只要任务的任何一个实例（主要或备份）完成工作，这个任务就被标识为完成。类似的策略可以运用在 reduce 任务上。通常我们只使用 1%~2% 的额外计算资源进行备份任务，但却大大地缩短了大规模 MapReduce 操作的通常所需的完成时间。

模型扩展

虽然 MapReduce 的大部分使用情况仅需要编写 Map 和 Reduce 函数，但是我们还是为基本模型扩展了一些实际应用中非常有用的功能。

划分函数

MapReduce 用户定义预期的 reduce 任务或输出文件的数目 (R)。通过在中间 key 上使用划分函数，可以在中间任务中对中间数据进行划分。我们提供了一个缺省的划分函数，它使用的是哈希法 ($\text{hash}(\text{key}) \% R$)，从而均匀地平衡了 R 个分区中的数据。

然而，有时也需要使用其他的 key 函数来划分数据。例如，输出 key 有时是 URL，我们希望单机上的所有条目都以同一个输出文件结束。为了应付这种情况，MapReduce 库的用户可以提供自定义的划分函数。例如，使用 $\text{hash}(\text{Hostname}(\text{urlkey})) \% R$ 作为划分函数，就可以将同一主机的所有 URL 都以同一个输出文件结束。

顺序保证

我们的 MapReduce 实现对中间数据进行排序，将共享相同中间 key 的所有中间 value 集合在一起。很多用户觉得在有序 key 上调用 Reduce 函数很方便，而我们已

经完成了所有必要的工作。因此，我们决定将这个功能展现给用户，具体是通过 MapReduce 库的接口来保证顺序属性。

跳过错误记录

有时候用户代码中的 bug，会导致某些记录上的 Map 或 Reduce 函数的必然 crash。这些 bug 可能导致巨大的 MapReduce 执行在完成大量的计算后出现故障。虽然首选方针应该是修复 bug，但有时并不可行。例如，bug 可能出现在第三方的库中，而我们无法获得它的代码。有时忽略少量记录也是可以接受的，例如在巨大的数据集上做统计分析时。因而，我们提供了一种执行的可选模式：MapReduce 库检测出导致必然 crash 的记录，为了确保能够不断取得进展，会在之后的重新执行中跳过这些记录。

每个 worker 进程都装有信号处理程序（signal handler），它能捕捉段违例（segmentation violation）和总线错误（bus error）。在用户 Map 或 Reduce 操作调用前，MapReduce 库会在一个全局变量中存储记录的序列号。如果用户代码产生了一个信号，信号处理程序会发送一个包含序列号的“last gasp” UDP 数据包给 MapReduce 的 Master。当 Master 在某条特定记录不止看到一个故障时，它就会提示：再次执行下一个相关的 Map 或 Reduce 任务时，应该跳过这条记录。

在一篇更为冗长的关于 MapReduce 的论文中讨论了很多其他的扩展（参见后面的“进一步阅读”）。

结论

MapReduce 已经被证明是 Google 很有价值的工具。截至 2007 年初，我们有超过 6000 个使用 MapReduce 编程模型所编写的不同程序，每天运行 35,000 个 MapReduce 作业，每天处理大约 8 PB 的输入数据（大约 100GB/s 的持续处理速率）。虽然最初开发 MapReduce 编程模型，是作为我们重写网络搜索产品的索引系统工作的一部分，但后来 MapReduce 在各种各样的问题领域都展示了其用处，包括机器学习，统计机器翻译，日志分析，信息检索实验，通用海量数据处理与海量计算任务。

进一步阅读

OSDI'04 会议上发表的一篇关于 MapReduce 的论文阐述更为详细：

Jeffrey Dean 和 Sanjay Ghemawat 于 2004 年 12 月在加州旧金山举行的 OSDI'04 会议上（第六届操作系统设计与实现研讨会）所发表的《MapReduce：在超大集群上的简易数据处理》。可从 <http://labs.google.com/papers/mapreduce.html> 获得。

SOSP'03 会议上发表的一篇关于 Google 文件系统设计与实现的论文：

Sanjay Ghemawat, Howard Gobioff 和 Shun-Tak Leung 于 2003 年 10 月在纽约州乔治湖 (Lake George) 举行的第 19 届 ACM 操作系统原理研讨会上所发表的《Google 文件系统》。

《IEEE Micro》上发表的一篇阐述 Google 通用硬件架构的论文：

Luiz Barroso, Jeffrey Dean 和 Urs Hölzle 在 2003 年 3 月份的《IEEE Micro》第 23 卷, 第 2 册, 第 22~28 页所发表的《星球网络搜索: Google 集群架构》。可从 <http://labs.google.com/papers/googlecluster.html> 获得。

Google 开发的运行于 Map-Reduce 上进行日志分析的 Sawzall 语言：

Rob Pike, Sean Dorward, Robert Griesemer 和 Sean Quinlan 在《科学编程期刊: 网格和万维网计算编程模型和架构特刊》的总第 13 期, 年度第 4 期, 第 227~298 页所发表的《解释数据: 基于 Sawzall 的并行分析》。可从 <http://labs.google.com/papers/sawzall.html> 获得。

致谢

许多人都为 MapReduce 的开发与改进作出了重要的贡献, 他们包括 Tom Annau, Matt Austern, Chris Colohan, Frank Dabek, Walt Drummond, Xianping Ge, Victoria Gilbert, Shan Lei, Josh Levenberg, Nahush Mahajan, Greg Malewicz, Russell Power, Will Robinson, Ioannis Tsoukalidis 和 Jerry Zhao。MapReduce 构建在 Google 所开发的许多基础构件之上, 包括 Google 文件系统和集群调度系统。我们要特别感谢系统的开发人员。最后, 我们要感谢 Google 工程组织内部的 Map-Reduce 用户所提供的有用的反馈、建议和 bug 报告。

附录：单词计数解决方案

本小节包含了本章节先前部分所使用的单词频率计数示例的完整 C++ 实现代码。代码同样可以从本书的 O'Reilly 网站获得 (<http://www.oreilly.com/catalog/9780596510046>)。

划分处理器上的并行化单词计数程序

```
#include "mapreduce/mapreduce.h"

// 用户的 map 函数
class WordCounter : public Mapper {
```

```

public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // 跳过前缀空格和制表符
            while ((i < n) && isspace(text[i]))
                i++;

            // 找到单词结尾
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;
            if (start < i)
                EmitIntermediate(text.substr(start, i-start), "1");
        }
    };
REGISTER_MAPPER(WordCounter);

// 用户的 reduce 函数
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // 遍历相同 key 的所有条目，并将 value 相加
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // 为 input->key() 生成和
        Emit(IntToString(value));
    }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);

    MapReduceSpecification spec;

    // 将输入文件的列表存储到 "spec" 中
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }

    // 定义输出文件:
    // /gfs/test/freq-00000-of-00100
    // /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
}

```

```

    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");
    // 可选项：在 Map 任务中进行部分求和来节约网络带宽
    out->set_combiner_class("Adder");
    // 调节参数：最多使用 2000 台机器，每个任务最多使用 100MB 内存
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);
    // 现在运行它吧
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();
    // 完成：'result' 结构包含了关于计数器，耗费时间，使用机器的数目等信息。
    return 0;
}

// 将文件从本地磁盘读取到内存
void ReadFile(const string &filename, void *buf, int64_t size) {
    ifstream in(filename);
    in.read((char *)buf, size);
}

// 将内存写入本地磁盘
void WriteFile(const string &filename, const void *buf, int64_t size) {
    ofstream out(filename);
    out.write((const char *)buf, size);
}

// 将本地文件上传到 HDFS
void PutFile(const string &local, const string &hdfs) {
    string cmd = "hadoop fs -put " + local + " " + hdfs;
    system(cmd.c_str());
}

// 将 HDFS 文件下载到本地
void GetFile(const string &hdfs, const string &local) {
    string cmd = "hadoop fs -get " + hdfs + " " + local;
    system(cmd.c_str());
}

// 将本地目录上传到 HDFS
void PutDir(const string &local, const string &hdfs) {
    string cmd = "hadoop fs -put -r " + local + " " + hdfs;
    system(cmd.c_str());
}

// 将 HDFS 目录下载到本地
void GetDir(const string &hdfs, const string &local) {
    string cmd = "hadoop fs -get -r " + hdfs + " " + local;
    system(cmd.c_str());
}

```

第 24 章

美丽的并发

Simon Peyton Jones

免费午餐已经结束（注 1）。以往我们习惯于通过购买新一代 CPU 来加快程序的运行速度，但现在，这样的好时光已经一去不复返了。虽说下一代芯片会带有多个 CPU，但每个单独的 CPU 的速度却不会再变快了。所以，如果想让程序跑得更快的话，你就必须得学会编写并行程序（注 2）。

并行程序的执行是非确定性的，因而测试起来自然也就不容易；并发程序中有的 bug 甚至可能会无法重现。我对漂亮程序的定义是“简单而优雅，乃至显而易见没有任何错误”，而不仅仅是“几乎没有任何明显的错误”。（注 3）要想编写出能够可靠运行的并行程序，程序的美感是尤其要注意的方面。可惜一般来说并行程序总归没有它们相应的非并行（顺序式的）版本漂亮；尤其是在模块性方面：并行程序的模块性相对较弱。这一点我们后面会看到。

注 1: Herb Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," Dr. Dobb's Journal, March 2005.

注 2: Herb Sutter and James Larus, "Software and the concurrency revolution," ACM Queue, Vol. 3, No. 7, September 2005.

注 3: This turn of phrase is due to Tony Hoare.

本章将介绍软件事务内存 (STM)。软件事务内存是一项很有前景的技术，它提出了一种针对共享内存并行处理器编程的新手段，正如刚才所言，传统并行程序的模块性较弱，而这正是软件事务内存的强项。我希望你读完本章后能和我一样对这项新技术感到振奋。当然，软件事务内存也并非万灵药，但它的确对并发领域中令人望而却步的问题发起了一次漂亮且令人鼓舞的冲击。

一个简单的例子：银行账户

假设有这样一个简单的编程任务：

编写一段程序，将钱从一个银行账户转移到另一个账户。为了简化起见，两个账户都是存放在内存里面的——也就是说你不用考虑跟数据库的交互。要求是你的代码在并发环境中也必须能够正确执行，这里所谓的并发环境也就是指可能存在多个线程同时调用你的转账函数，而所谓能够正确执行则是指任何线程都不能“看到”系统处于不一致的状态（比如看到其中一个账户显示已被取出了一笔钱然而同时另一个账户却显示还没有收到这笔钱）。

这个例子虽说有点人为捏造的味道，但它的优点是简单，因而我们便能将注意力集中在它的解决方案上，后面你会看到，Haskell结合事务内存将会给这个问题带来新的解决方案。不过此前还是让我们先来简单回顾一下传统的方案。

加锁的银行账户

目前，用于协调并发程序的主导技术仍是锁和条件变量。在一门面向对象的语言中，每个对象上都带有一个隐式的锁，而加锁则由 *synchronized* 方法来完成，但原理与经典的加锁解锁是一样的。在这样一门语言中，我们可以将银行账户类定义成下面这样：

```
class Account {  
    Int balance;  
    synchronized void withdraw( Int n ) {  
        balance = balance - n; }  
    void deposit( Int n ) {  
        withdraw( -n ); }  
}
```

withdraw 方法必须是 *synchronized* 的，这样两个线程同时调用它的时候才不会把 *balance* 减错了。*synchronized* 关键字的效果就等同于先对当前账户对象加锁，然后运行 *withdraw* 方法，最后再对当前账户对象解锁。

有了这个类之后，我们再来编写 *transfer* 转账函数：

```
void transfer( Account from, Account to, Int amount ) {  
    from.withdraw( amount );  
    to.deposit( amount ); }
```

对于非并发程序来说以上代码完全没问题，然而在并发环境中就不一样了，另一个线程可能会看到转账操作的“中间状态”：即钱从一个账户内被取走了，而同时另一个账户却还没收到这笔钱。值得注意的是，虽然 withdraw 和 deposit 这两个函数都是 synchronized，但这对我们没有任何帮助，问题还是会出现。在 from 上调用 withdraw 会将 from 加锁，执行提款操作，然后对其解锁。类似的，在 to 上调用 deposit 会将 to 加锁，执行存款操作，然后对其解锁。然而，关键是在这两个调用之间有一个间隙，在这个状态下待转账的钱既不在 from 账户中也不在 to 账户中。

在一个金融程序中，这种情况可能是无法容忍的。那我们该如何解决这个问题呢？通常的方案可能是在外面再包一层显式的加锁解锁操作，如下：

```
void transfer( Account from, Account to, Int amount ) {  
    from.lock(); to.lock();  
    from.withdraw( amount );  
    to.deposit( amount );  
    from.unlock(); to.unlock() }
```

但这种做法有一个致命的缺陷：它可能会导致死锁。我们考虑这样一种情况：在一个线程将一笔钱从 A 账户转到 B 账户的同时，另一个线程也正在将一笔钱从 B 账户转到 A 账户（当然，发生这种事情的几率很小）。这时便可能会出现两个线程各锁一个账户并都在等着对方释放另一账户的情况。

问题找出来了（不过，并发环境下的问题可不总是像这么容易就能找出来的），标准的解决方案是规定一个全局统一的锁顺序，并按照递增顺序来进行加锁。采用这种做法的代码如下：

```
if from < to  
then { from.lock(); to.lock(); }  
else { to.lock(); from.lock(); }
```

这个方法是可行的，但前提是必须事先知道要对哪些锁进行加锁，而后面这个条件并不是总能满足的。例如，假设 from.withdraw 的实现当账户余额不足时就会从 from2 上提款。遇到这种情况除非等到我们从 from 中提了钱否则是无法知道是否该对 from2 加锁的，而另一方面，一旦已经从 from 中提了钱，再想按“正确”顺序加锁便不可能了。更何况 from2 这个账户可能根本就只对 from 可见，而不应被 transfer 知道。而且退一步说，就算 transfer 知道 from2 的存在，现在需要加的锁也已经由两个变成了三个（事先还要记得将这些锁正确排序）。

还有更糟的，如果我们需要在某些情况下阻塞（block），情况就会更加复杂。例如，要求transfer在from账户内余额不足的时候阻塞。这类问题通常的解决办法是在一个条件变量上等待，并同时释放from的锁。如果我们希望当from和from2中的总余额不够的时候才阻塞，那么情况将更为棘手。

“生锈”的锁

简而言之，在如今的并发编程领域占主导地位的技术，锁和条件变量，从根本上是有缺陷的。以下便是基于锁的并发编程中的一些公认的困难（其中有些我们在前文的例子中已经看到过了）。

锁加少了

容易忘记加锁，从而导致两个线程同时修改同一个变量。

锁加多了

容易加锁加得过多了，结果（轻则）妨碍并发，（重则）导致死锁。

锁加错了

在基于锁的并发编程中，锁和锁所保护的数据之间的联系往往只存在于程序员的大脑里，而不是显式地表达在程序代码中。结果就是一不小心便会加错了锁。

加锁的顺序错了

在基于锁的并发编程中，我们必须小心翼翼地按照正确的顺序来加锁（解锁），以避免随时可能会出现的死锁；这种做法既累人又容易出错，而且，有时候极其困难。

错误恢复

错误恢复也是个很麻烦的问题，因为程序员必须确保任何错误都不能将系统置于一个不一致的、或锁的持有情况不确定的状态下。

忘记唤醒和错误的重试

容易忘记叫醒在条件变量上等待的线程；叫醒之后又容易忘记重设条件变量。

然而，基于锁的编程，其最根本的缺陷，还是在于锁和条件变量不支持模块化编程。这里“模块化编程”是指通过粘合多个小程序来构造大程序的过程。而基于锁的并发程序是做不到这一点的。还是拿前面的例子来说吧：虽然withdraw和deposit这两个方法都是并发正确的，但如果原封不动的话，你能直接用它们实现一个transfer来吗？不能，除非让锁协议暴露出来。而且遇到选择和阻塞的话还会更头疼。例如，假设withdraw在账户余额不足的情况下会阻塞。你就会发现，除非暴露锁条件，否则你根本无法直接利用withdraw函数从“A账户或B账户（取决于哪个账户有足够余额）”

进行提款；而且就算知道了锁条件，事情仍还是麻烦。另一些文献中也对锁并发的这种困难作了论述（注 4）。

软件事务内存

软件事务内存（STM）是迎接并发挑战的一种很有前途的新技术，本节将会详细说明这一点。我选用 Haskell 来介绍 STM，Haskell 是我见过的最美丽的编程语言，而且 STM 能够特别优雅地融入到 Haskell 中。如果你还不了解 Haskell，别担心，边看边学。

Haskell 中的副作用（Side Effects）和输入 / 输出（I/O）

Haskell 中的 transfer 函数写出来就像这样：

```
transfer :: Account -> Account -> Int -> IO ()  
-- Transfer 'amount' from account 'from' to account 'to'  
transfer from to amount = ...
```

以上代码的第二行，即以“--”开头的那行，是一个注释。代码的第一行是对 transfer 的函数类型的声明（类型声明以“::”前导）（注 5），“Account->Account->Int->IO ()”读作“接受一个 Account，加上另一个 Account（两个 Account，代表转账的源账户和目标账户），以及一个 Int（转账的数额），返回一个 IO() 类型的值”。最后一个类型（即返回类型）“IO()”说的是“transfer 函数返回的是一个动作（action），这个动作被执行的时候可能会产生副作用（side effects），并返回一个 ‘()’ 类型的值”。“()”类型读作“单元（unit）”，该类型只有一个可能的值，也写作“()”，有点类似于 C 里面的 void。transfer 将 IO() 作为返回类型说明了执行过程中的副作用是我们调用

注 4：Edward A. Lee, "The problem with threads," IEEE Computer, Vol. 39, No. 5, pp. 33-42, May 2006; J. K. Ousterhout, "Why threads are a bad idea (for most purposes)," Invited Talk, USENIX Technical Conference, January 1996; Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy, "Composable memory transactions," ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '05), June 2005.

注 5：你可能会觉得奇怪，为什么在这个类型签名里面有三个“->”（难道不应该是一个吗——位于参数类型与返回类型之间？）其实这是因为 Haskell 支持所谓的 currying，后者在任何介绍 Haskell 的书（比如 Haskell: The Craft of Functional Programming, by S.J. Thompson [Addison-Wesley]）中或 wikipedia 上都能见到它的踪影。但 currying 不是本章要讲的重点，你大可以忽略除了最后一个“->”之外的所有“->”，即除了最后一个类型之外，其他都是函数的参数类型。

transfer的惟一原因。那么，在介绍下面的内容之前，我们就必须首先知道 Haskell 是怎么对待副作用的。

那么副作用是什么呢？副作用就是我们读写可变(mutable)状态所造成的影响(effect)。输入/输出是副作用的绝佳范例。例如，下面是两个 Haskell 函数，它们都具有输入/输出副作用：

```
hPutStr :: Handle -> String -> IO ()  
hGetLine :: Handle -> IO String
```

任何类型形如“IO t”（其中 t 可以为 ()，也可以为其他类型，如 String）的值都是一个动作 (action)。也就是说，在上面的例子中，(hPutStr h "hello") 是一个动作（注 6），执行这个动作的效果便是在句柄 h 上输出 "hello"（注 7）。类似的，(hGetLine h) 也是一个动作，当它被执行的时候就会从句柄 h 代表的输入设备中读入一行输入并将其作为一个 String 返回。此外，利用 Haskell 的 do 关键字，我们可以将几个小的带副作用的程序“粘合”成一个大的。例如，下面的 hEchoLine 函数读入一个串并将它打印出来：

```
hEchoLine :: Handle -> IO String  
hEchoLine h = do { s <- hGetLine h  
                  ; hPutStr h ("I just read: " ++ s ++ "\n")  
                  ; return s }
```

do { a_1 ; ...; a_n } 结构将数个较小的动作 ($a_1 \dots a_n$) 粘合成一个较大的动作。因此在上面的代码中，hEchoLine h 这个动作被执行的时候便会首先调用 hGetLine h 来读取一行输入，并将这行输入命名为 s。接着调用 hPutStr（注 8），将 s 加上前导的 "I just read:" 一并打印出来。最后一行 return s 比较有趣，因为 return 并非像在其他命令式语言中那样是语言内建的操作，而只是一个普普通通的函数，其函数类型如下：

```
return :: a -> IO a
```

也就是说，像 return v 这样一个操作被执行的时候，将会返回 v，同时并不会导致任

注 6： 在 Haskell 里面调用一个函数很简单，只须将函数名和它的各个参数并排写在一块儿就行了。在大多数语言中你都需要写成 hPutStr(h, "hello")，但 Haskell 里面只要写成 (hPutStr h "hello") 就行了。

注 7： Haskell 中的句柄相当于 C 里面的文件描述 (file descriptor)：指明对哪个文件或管道进行读写。跟 Unix 里面一样，Haskell 里面也预定义了三个句柄：stdin、stdout 和 stderr。

注 8： ++ 操作符的作用是将两个串拼接起来。

何副作用（注9）。return函数可以作用于任何类型的值，这一点体现在其函数类型中：a是一个类型变量，代表任何类型。

输入/输出是一类重要的副作用。还有一类重要的副作用便是对可变（mutable）变量的读写。例如，下面这个函数将一个可变变量的值增1：

```
incRef :: IORRef Int -> IO ()  
incRef var = do { val <- readIORRef var  
                  ; writeIORRef var (val+1) }
```

incRef var是一个动作。它首先执行readIORRef var来获得变量var的值，并将该值绑定到val；接着它调用writeIORRef将val+1写回到var里面。readIORRef和writeIORRef的类型如下：

```
readIORRef :: IORRef a -> IO a  
writeIORRef :: IORRef a -> a -> IO ()
```

类型形如IORRef t的值相当于一个指针或引用，指向或引用一个t类型的可变值（类似于C里面的t*）。具体到incRef，其参数类型为IORRef Int，因为incRef只操作Int型变量。

现在，我们已经知道了如何将数个较小的动作组合成一个较大的——但一个动作到底如何才算被真正调用呢？在Haskell里，整个程序其实就是一个IO动作，名叫main。要运行这个程序，我们只需执行main。例如下面就是一个完整的程序：

```
main :: IO ()  
main = do { hPutStr stdout "Hello"  
           ; hPutStr stdout " world\n" }
```

该程序是一个顺序式（sequential）程序，因为do块将两个IO动作按顺序连接了起来。另一方面，要构造并发程序的话，我们便需要另一个原语（primitive）：forkIO：

```
forkIO :: IO a -> IO ThreadId
```

forkIO是Haskell内建的函数，它的参数是一个IO动作，forkIO做的事情就是创建一个并发的Haskell线程来执行这个IO动作。一旦这个新线程建立，Haskell的运行时系统便会将它与其他Haskell线程并行执行。例如假设我们将前面的main函数修改为（注10）：

注9：“IO”的意思是一个函数可能有副作用，但并不代表它就一定会带来副作用。

注10：其实，main的第一行我们本可以写成“tid <- forkIO(hPutStr ...)”的，这行语句会把forkIO的返回值（一个ThreadId）绑定到tid。然而由于本例中我们并不使用返回的ThreadId，所以就把“tid <-”省略了。

```
main :: IO ()  
main = do { forkIO (hPutStr stdout "Hello")  
          ; hPutStr stdout " world\n" }
```

现在，这两个 `hPutStr` 操作便能够并发执行了。至于哪个先执行（从而先打印出它的字符串）则是不一定。Haskell 里面由 `forkIO` 产生的线程是非常轻量级的：只占用几百个字节的内存，所以一个程序里面就算产生上千个线程也是完全正常的。

读到这里，你可能会觉得 Haskell 实在是门又笨拙又麻烦的语言，`incRef` 的三行代码说穿了就做了 C 里面的一个 `x++` 而已！没错，在 Haskell 里面，实施副作用的方式是非常显式且冗长的。然而别忘了，首先 Haskell 主要是一门函数式编程语言。大多数代码都是在 Haskell 的函数式内核里写的，后者的特点是丰富、高表达力、简洁。因而 Haskell 编程的精神就是“有节制地使用副作用”。

其次我们注意到，在代码中显示声明副作用的好处是代码能够携带许多有用的信息。考虑下面两个函数：

```
f :: Int -> Int  
g :: Int -> IO Int
```

通过它们的类型我们便可以一眼看出，`f` 是一个纯函数，无副作用。给它一个特定的数（比如 42），那么每次对它的调用 (`f 42`) 都会返回同样的结果。相比较之下，`g` 就具有副作用了——这一点明明白白地显示在它的类型中。对 `g` 的不同的调用，就算参数相同，也可能会得到不同的值，因为，比如说，它可以通过 `stdin` 读取数据，或对一个可变变量进行修改——即使它的参数每次都是相同的。后面你会发现，这种显式声明副作用的做法其实非常有用。

最后，前面提到的所谓动作（action，比如 I/O 动作）其实本身也是值（Haskell 中的动作也是一等公民）：它们也可以被作为参数传递或作为返回值返回。比如下面就是一个纯粹用 Haskell 写的（而非内建的）模拟（简化的）`for` 循环的函数：

```
nTimes :: Int -> IO () -> IO ()  
nTimes 0 do_this = return ()  
nTimes n do_this = do { do_this; nTimes (n-1) do_this }
```

这是一个递归函数，其第一个参数是一个 `Int`，表示要循环多少次，第二个参数则是一个动作（action）：`do_this`；该函数返回一个动作，后者被执行的时候会把 `do_this` 重复做 `n` 遍。比如下面这段代码利用 `nTimes` 来重复输出 10 个“Hello”：

```
main = nTimes 10 (hPutStr stdout "Hello\n")
```

这其实从效果上就等同于允许用户自定义程序的控制结构了。

话说回来，本章的目的并不是要对 Haskell 作一个全面的介绍，而且即便是对于 Haskell 里面的副作用我们也只是稍加阐述。如果你想进一步了解的话，可以参考我写的一篇指南 “Tackling the awkward squad”（注 11）。

Haskell 中的事务

OK，终于可以回到我们的 transfer 函数了。其代码如下：

```
transfer :: Account -> Account -> Int -> IO ()  
-- Transfer 'amount' from account 'from' to account 'to'  
transfer from to amount  
= atomically (do { deposit to amount  
                  ; withdraw from amount })
```

里面的那个 do 块你应该不觉得陌生了吧：它先是调用 deposit 将 amount 数目的钱存入 to 账户，然后再从 from 账户中提取 amount 数目的钱。至于 deposit 和 withdraw 这两个辅助函数我们待会再来写，现在我们先来看看对 atomically 的调用。atomically 的参数是一个动作，它会将该动作作为一个原子来执行。更精确地说，atomically 有如下两个特性作保证：

原子性

atomically act 调用所产生的副作用对于其他线程来说是“原子的”。这就保证了另一个线程不可能观察到钱被从一个账户中取出而同时又没来得及存入另一个账户中去的中间状态。

隔离性

在 atomically act 执行的过程中，act 这个动作与其他线程完全隔绝，不受影响。这就好像在 act 开始执行的时候世界停顿了，直到 act 执行完毕之后世界才又开始恢复运行。

至于 atomically 函数的执行模型，简单的做法是：存在一个惟一的全局锁；atomically act 首先获取该锁，然后执行动作 act，最后释放该锁。这个实现虽然保证了原子性，但粗暴地禁止了任意两个原子块在同一时间执行。

上面说的这个模型有两个问题。第一，它并没有保证隔离性：比如一个线程在执行一个

注 11： Simon Peyton Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell," C. A. R. Hoare, M. Broy, and R. Steinbrueggen, editors, Engineering theories of software construction, Marktoberdorf Summer School 2000, NATO ASI Series, pp. 47-96, IOS Press, 2001。

原子块的过程中访问了一个 `IORef` (此时该线程持有全局锁)，另一个线程此时照样可以直接对同一个 `IORef` 进行写操作 (只要这个写操作不在原子块内)。这就破坏了隔离性保证。第二，它极大的损害了执行性能，因为即便各个原子块之间互不相干，也必须被串行化执行。

第二个问题待会在“事务内存实现”一节会详细讨论。目前先来看第一个问题。第一个问题可以通过类型系统轻易解决。我们将 `atomically` 函数赋予如下类型：

```
atomically :: STM a -> IO a
```

`atomically` 的参数是一个类型为 `STM a` 的动作。`STM` 动作类似于 `IO` 动作，它们都可能具有副作用，但 `STM` 动作的副作用的容许范围要小得多。`STM` 中你可以做的事情主要就是对事务变量 (类型为 `TVar a`) 进行读写，就像我们在 `IO` 动作里面主要对 `IORef` 进行读写一样 (注 12)。

```
readTVar :: TVar a -> STM a  
writeTVar :: TVar a -> a -> STM ()
```

跟 `IO` 动作一样，`STM` 动作也可以由 `do` 块组合起来，实际上，`do` 块针对 `STM` 动作进行了重载，`return` 也是；这样它们便可以运用于 `STM` 和 `IO` 两种动作了 (注 13)。例如，下面是 `withdraw` 的代码：

```
type Account = TVar Int  
withdraw :: Account -> Int -> STM ()  
withdraw acc amount
```

注 12：这儿其实有一个命名上的不一致：`STM` 变量被命名为 `TVar`，然而普通变量却被命名为 `IORef`——其实要么应该是 `TVar/IOVar`，要么应该是 `TRef/IORef` 才对。但事到如今已经没法再改了。无论如何，我们有了 `Tvar` 和 `IORef`。

注 13：其实 Haskell 并没有特别针对 `IO` 和 `STM` 动作来重载 `do` 和 `return`，`IO` 和 `STM` 其实只是一个更一般的模式的特例，这个更一般的模式便是所谓的 *monad* (P. L. Wadler 在 “The essence of functional programming” 20th ACM Symposium on Principles of Programming Languages [POPL '92], Albuquerque, pp. 1-14, ACM, January 1992 中有描述)，`do` 和 `return` 的重载便是通过用 Haskell 的非常泛化的“类型的类型” (type-class) 系统来表达 `monad` 而得以实现的 (described in P. L. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas, ACM, January 1989; and Simon Peyton Jones, Mark Jones, and Erik Meijer, "Type classes: an exploration of the design space," J. Launchbury, editor, Haskell workshop, Amsterdam, 1997)。

```
= do { bal <- readTVar acc  
      ; writeTVar acc (bal - amount) }
```

我们用一个包含一个 Int (账户余额) 的事务变量来表示一个账户。withdraw是一个 STM 动作，这个动作把账户中的余额减去 amount。

为了完成 transfer 的定义，我们可以通过 withdraw 来定义 deposit：

```
deposit :: Account -> Int -> STM ()  
deposit acc amount = withdraw acc (- amount)
```

注意，transfer 从根本上执行了四个基本的读写操作：对 to 账户的一次读和一次写；以及对 from 账户的一次读和一次写。这四个操作是被当成一个原子来执行的，其执行满足本节（“一个简单的例子：银行账户”）开头的要求。

Haskell 的类型系统优雅地阻止了我们在事务之外读写 TVar。例如假设我们这样写：

```
bad :: Account -> IO ()  
bad acc = do { hPutStr stdout "Withdrawing..."  
              ; withdraw acc 10 }
```

以上代码不能通过编译，因为 hPutStr 是一个 IO 动作，而 withdraw 则是一个 STM 动作，这两者不能放在同一个 do 块中。但如果我们将 withdraw 再放在一个 atomically 调用当中就可以了，如下：

```
good :: Account -> IO ()  
good acc = do { hPutStr stdout "Withdrawing..."  
               ; atomically (withdraw acc 10) }
```

事务内存实现

有了前面提到过的原子性和隔离性保证，我们其实便可以放心使用 STM 了。不过我常常还是觉得一个合理的实现模型会给直觉理解带来很大的帮助，本节就来介绍这么一个实现模型。但要注意的是，这只是所有可能实现中的一种。STM 抽象的一个漂亮之处就在于它提供了一个小巧干净的接口，而实现这个接口可以有多种方式，可简单可复杂。

在可行的实现方案中，有一个方案特别吸引人，那就是在数据库实现里被采用的所谓的“乐观执行 (optimistic execution)”。当 atomically act 被执行的时候，Haskell 运行时系统会为它分配一个线程本地的事务日志，该日志最初的时候是空的，随着 act 动作被一步步执行（其间并不加任何形式的锁），每次对 writeTVar 的调用都会将目标 TVar 变量的地址和新值写入日志；而并不是直接写入到那个 TVar 变量本身。每次对 readTVar 的调用都会首先寻找日志里面有没有早先被写入的新值，没有的话才会从目

标TVar本身读取，并且，在读取的时候，一份拷贝会被顺便存入到日志中。同一时间，另一个线程可能也在运行着它自己的原子块，疯狂地读写同样一组TVar变量。

在act这个动作执行完毕之后，运行时系统首先会对日志进行验证，如果验证成功，就会提交(commit)日志。那么验证是怎么进行的呢？运行时系统会检查日志中缓存的每个readTVar的值是否与它们对应的真正的TVar相匹配。是的话便验证成功，并将日志中缓存的写操作结果全都提交到相应的TVar变量上。

必须注意的是，以上验证-提交的整个过程是完全不可分割的：底层实现会将中断禁止掉，或使用锁或CAS(compare-and-swap)指令等任何可行的方法来确保这个过程对于其他线程来说就像“一瞬间”的事情一样。但由于所有这些底层工作都由实现来完成，所以程序员不用担心也不用考虑它是怎么完成的。

一个自然而然的问题是：如果验证失败呢？如果验证失败，就代表该事务看到的是不一致的内存视图。于是事务被中止(abort)，日志被重新初始化，然后整个事务从头再来过。这个过程就叫做重新执行(re-execution)。由于此时act动作的所有副作用都还没有真正提交到内存中，因此重新执行它是完全没问题的。然而有一点必须注意：act不能包含任何除了对TVar变量读写之外的副作用，比如下面这种情况：

```
atomically (do { x <- readTVar xv
                  ; y <- readTVar yv
                  ; if x>y then launchMissiles
                           else return () })
```

launchMissiles::IO()这个函数的副作用是“头晕、恶心、呕吐”。由于这个原子块执行的时候并没有加锁，所以如果同时有其他线程也在修改变量xv和yv的话，该线程就可能观察到不一致的内存视图。而一旦这种情况发生，发射导弹(launchMissiles)可就闯了大祸了，因为等到导弹发射完了才发现验证失败就已经来不及了。不过幸运的是，Haskell的类型系统会阻止冒失的程序员把IO动作（比如这个launchMissiles）放在STM动作中执行，所以，以上代码会被类型系统阻止。这从另一个方面显示了将IO动作跟STM动作区分开来的好处。

阻塞和选择

到目前为止我们介绍的原子块从根本上还缺乏一种能力：无法用来协调多个并发线程。这是因为还有两个关键的特性不具备：阻塞和选择。本节就来介绍如何扩充基本的STM接口从而使之包含以上两个特性（当然，在完全不破坏模块性的前提下）。

假设当一个线程试图从一个账户中提取超过账户余额的钱时这个线程便会阻塞。并发编

程中这类情况很常见：例如一个线程在读取到一个空的缓冲区时阻塞；或在等待一个事件的时候阻塞，等等。为了支持这种场景，我们往 STM 中加入 retry 功能，retry 的类型为：

```
retry :: STM a
```

以下是 withdraw 的一个修改过的版本，该版本当余额不足的时候便会转入到阻塞状态：

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
= do { bal <- readTVar acc
      ; if amount > 0 && amount > bal
        then retry
        else writeTVar acc (bal - amount) }
```

retry 的语意很简单：当一个 retry 语句被执行的时候，当前事务便被“丢弃”并等待某个时候再重新执行。当然，这里的“某个时候”可以是立即，但这样做不够高效：如果重新执行的时候账户余额根本没有改变的话还是白搭，结果又是 retry。一个高效的实现不会这么做，而是会一直阻塞，直到有其他线程对 acc 进行了写操作（译注：即改变了账户余额）。那么，问题是实现怎么知道要在 acc 变量上等待呢？很简单，因为该事务在到达 retry 这一点之前的执行路径上读取的就是 acc，而这个读取动作会被事务日志完完整整地记录下来（译注：所以只要往日志里“瞄一眼”就知道了）。

limitedWithdraw 中的条件有一个非常普遍的模式：检查一个布尔条件是否满足，如果不满足则 retry。这个模式很容易抽象出来做成一个函数：

```
check :: Bool -> STM ()
check True = return ()
check False = retry
```

利用这个 check 函数来重新表达 limitedWithdraw 如下（是不是简洁了一些？）：

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
= do { bal <- readTVar acc
      ; check (amount <= 0 || amount <= bal)
      ; writeTVar acc (bal - amount) }
```

接下来我们考虑选择（choice）。假设你想要从 A 账户上取钱，但前提是 A 上必须要有足够的钱，如果没有的话，你便改从 B 上取。为此我们必须实现以下能力：如果前一条路径需要 retry，则选择另一条路径。于是 STM Haskell 加入了另一个原语：orElse，来支持选择。orElse 的类型为：

```
orElse :: STM a -> STM a -> STM a
```

跟`atomically`函数一样，`orElse`的参数也是动作，`orElse`将小的动作粘合成较大的动作。其语意如下：`(orElse a1 a2)`首先会执行动作`a1`，如果`a1`发生了`retry`，那么它便会转而执行`a2`。如果`a2`也`retry`了，那么整个动作（`a1`和`a2`）便被`retry`。`orElse`的用法很简单：

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ()  
-- (limitedWithdraw2 acc1 acc2 amt) withdraws amt from acc1,  
-- if acc1 has enough money, otherwise from acc2.  
-- If neither has enough, it retries.  
limitedWithdraw2 acc1 acc2 amt  
= orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

由于`orElse`返回的也是一个STM动作，因此我们便可以将`orElse`调用的结果给另一个`orElse`，如此嵌套，便可以实现任意数目的备选路径。

基本 STM 操作小结

本节我们介绍了STM Haskell支持的所有关键的事务内存操作。表24-1是一个小结。注意，里面有一个操作：`newTVar`到目前为止我们还没有提到。`newTVar`是用来创建新的TVar变量的，下一节我们会用到它。

表24-1：STM Haskell支持的关键操作

操作	类型签名
<code>atomically</code>	<code>STM a -> IO a</code>
<code>retry</code>	<code>STM a</code>
<code>orElse</code>	<code>STM a -> STM a -> STM a</code>
<code>newTVar</code>	<code>a -> STM (TVar a)</code>
<code>readTVar</code>	<code>TVar a -> STM a</code>
<code>writeTVar</code>	<code>TVar a -> a -> STM ()</code>

圣诞老人问题

本节将为你展示一个完整的、可运行的STM程序。一个众所周知的例子便是所谓的“圣诞老人问题”（注14），这个问题最初由Trono提出（注15）：

注14： My choice was influenced by the fact that I am writing these words on December 22.

注15： J. A. Trono, "A new exercise in concurrency," SIGCSE Bulletin, Vol. 26, pp. 8-10, 1994.

问题描述是这样的：圣诞老人总是在睡觉，直到被他的（放假归来的）九头驯鹿中的任意一头叫醒，或被他的十个小矮人中的任意三个（一组）叫醒。如果是被驯鹿叫醒的，他就把驯鹿们套上雪橇出门去给小朋友们送礼物，回来之后再解开驯鹿（放假）。如果是被一组（三个）小矮人叫醒的，他就将这三个小矮人一个个带进书房，跟他们交流玩具的制造，然后再将他们一个个领出去（好让他们回去继续工作）。如果发现既有一组小矮人又有一群驯鹿在等他的话，圣诞老人便优先选择驯鹿。

使用一个众所周知的例子的好处便是在一些其他语言中已经有了描述得很好的解决方案了，这样你就能够将我们马上要介绍的方案跟其他语言中既有的方案进行一目了然的比较。值得注意的是，Trono 的论文中给出了一个基于信号量的方案，那个方案只是部分正确的；Ben-Ari 用 Ada95 和 Ada 给出了解决方案（注 16）；Benton 也用 Polyphonic C#（译注：C#的一个扩展，主要加入基于 Join Calculus 的并发编程模型）写了一个解决方案（注 17）。

驯鹿和小矮人

用 STM Haskell 来解决这个问题，基本理念是这样的：圣诞老人分别给小矮人和驯鹿各创建一个“群”。每个小矮人（或驯鹿）都试图去加入相应的群。如果成功的话，便得到两扇“门”。第一扇门允许圣诞老人控制什么时候让小矮人们进入书房，并得知什么时候他们全都进去了。类似的，第二扇门则控制小矮人们离开书房。圣诞老人等待他创建的任何一个群准备好，并用那个准备好的群的两扇门来控制他们的小帮手们（小矮人或驯鹿）完成工作。也就是说，不管是驯鹿还是小矮人，他们一直都在做一个无限循环：试图加入群——在圣诞老人的看管下穿过“门”——等待一段时间（比如驯鹿便会去休假）——再次试图加入群。

用 Haskell 来描述上述逻辑，我们便得到如下的代码（这是针对小矮人的）（注 18）：

```
elf1 :: Group -> Int -> IO ()  
elf1 group elf_id = do { (in_gate, out_gate) <- joinGroup group
```

注 16： Mordechai Ben-Ari, "How to solve the Santa Claus problem," *Concurrency: Practice and Experience*, Vol. 10, No. 6, pp. 485-496, 1998.

注 17： Nick Benton, "Jingle bells: Solving the Santa Claus problem in Polyphonic C#," Technical report, Microsoft Research, 2003.

注 18： 为什么是 `elf1` 而不是 `elf`，是因为这个函数只做一重循环，而实际上小矮人会不断重复加入群。后面我们会基于 `elf1` 来定义 `elf`。

```
; passGate in_gate  
; meetInStudy elf_id  
; passGate out_gate }
```

elf1的参数是一个“群”(Group)以及一个Int，后者惟一指代该小矮人的身份。这个Int只在调用meetInStudy的时候用到，meetInStudy简单地打印出一行消息表明正在发生的事情(注19)：

```
meetInStudy :: Int -> IO ()  
meetInStudy id = putStrLn ("Elf " ++ show id ++ " meeting in the study\n")
```

小矮人调用joinGroup加人群，然后调用passGate来穿过门，这两个函数如下：

```
joinGroup :: Group -> IO (Gate, Gate)  
passGate :: Gate -> IO ()
```

驯鹿们的代码几乎完全一样，惟一的区别就是驯鹿的任务是送礼物而不是进书房讨论：

```
deliverToys :: Int -> IO ()  
deliverToys id = putStrLn ("Reindeer " ++ show id ++ " delivering toys\n")
```

由于IO动作也是值，所以我们便可以将驯鹿和小矮人们的逻辑抽象出一个公共模式来，如下：

```
helper1 :: Group -> IO () -> IO ()  
helper1 group do_task = do { (in_gate, out_gate) <- joinGroup group  
; passGate in_gate  
; do_task  
; passGate out_gate }
```

helper1的第二个参数是一个IO动作，代表待执行的任务。helper1负责将这个任务放在两个passGate调用之间执行。有了这个辅助函数，我们便可以基于它来定义小矮人和驯鹿函数了：

```
elf1, reindeer1 :: Group -> Int -> IO ()  
elf1 gp id = helper1 gp (meetInStudy id)  
reindeer1 gp id = helper1 gp (deliverToys id)
```

门和群

先来看“门”这个抽象概念。“门”支持如下接口：

```
newGate :: Int -> STM Gate  
passGate :: Gate -> IO ()  
operateGate :: Gate -> IO ()
```

注19：putStrLn是一个库函数，它会调用hPutStrLn stdout。

每个门都有一个固定的容限 n。这个 n 是在我们新建门的时候指定的。此外门还有一个剩余容许量 m，这是一个可变的 (mutable) 变量。passGate 被调用的时候，m 就会减一。如果剩余容许量减至零，那么对 passGate 的调用就会阻塞。一个门最初被创建起来的时候剩余容许量为 0；因此任何人都不能进入。而圣诞老人则通过调用 operateGate 来打开这扇门，operateGate 会将门的剩余容许量置为 n。

下面便是门 (Gate) 的一个可能实现：

```
data Gate = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do { tv <- newTVar 0; return (MkGate n tv) }

passGate :: Gate -> IO ()
passGate (MkGate n tv)
  = atomically (do { n_left <- readTVar tv
                      ; check (n_left > 0)
                      ; writeTVar tv (n_left-1) })

operateGate :: Gate -> IO ()
operateGate (MkGate n tv)
  = do { atomically (writeTVar tv n)
        ; atomically (do { n_left <- readTVar tv
                           ; check (n_left == 0) }) }
```

第一行是类型声明，声明 Gate 为一个新的数据类型，并具有一个数据构造子叫 MkGate（注 20），该构造子有两个成员：一个 Int 型数据，表示该门的容限。另一个则是一个 TVar，表示在门关闭之前还有多少人可以穿过它。如果该 TVar 为 0，就表明门是关闭的。

函数 newGate 负责创建一个新的门，它先是分配一个 TVar，然后通过调用 MkGate 这个构造子来创建一个 Gate 对象。无独有偶，passGate 利用模式匹配来将 MkGate 构造子拆分开来（即 (MkGate n tv)），然后 passGate 将 Tvar 的值减一，并利用 check 来确保 $tv > 0$ 。前面“阻塞和选择”一节我们实现 withdraw 的时候也用到过这个 check。最后是 operateGate 函数，operateGate 将门的最大容限值写入 Tvar，并等待 Tvar 被减至 0。

群 (Group) 的接口如下：

```
newGroup :: Int -> IO Group
joinGroup :: Group -> IO (Gate, Gate)
awaitGroup :: Group -> STM (Gate, Gate)
```

注 20： Haskell 中的类型声明不像 C 里面的结构声明，MkGate 只是一个结构 tag。

跟门类似，群刚创建起来的时候也是空的，并带有一个指定的容限。小矮人们可以通过调用 `joinGroup` 来加入群，如果群已满那么 `joinGroup` 就会阻塞。而圣诞老人则会调用 `awaitGroup` 来等待群被加满；群满了之后，圣诞老人就会通过 `awaitGroup` 的返回值得到该群对应的两扇门，然后群立即被用两扇新门重新初始化，好让另一组焦急的小矮人们开始集合。

下面是 `newGroup` 的一个可能的实现：

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))

newGroup n = atomically (do { g1 <- newGate n; g2 <- newGate n
                           ; tv <- newTVar (n, g1, g2)
                           ; return (MkGroup n tv) })
```

同样，`Group` 是一个新声明的数据类型，其构造子 `MkGroup` 具有两个成员：该 `Group` 的容限以及一个 `TVar`，后者包含该 `Group` 剩余的名额以及两个 `Gate` 对象。要创建一个新的 `Group` 对象 (`newGroup`)，先要创建两个 `Gate` 对象 (`g1, g2`)，并初始化一个新的 `TVar` (`tv`)，然后调用 `Group` 的构造子 `MkGroup`。

而 `joinGroup` 和 `awaitGroup` 基本就是基于上面的这些数据结构来实现的：

```
joinGroup (MkGroup n tv)
  = atomically (do { (n_left, g1, g2) <- readTVar tv
                     ; check (n_left > 0)
                     ; writeTVar tv (n_left-1, g1, g2)
                     ; return (g1,g2) })

awaitGroup (MkGroup n tv)
  = do { (n_left, g1, g2) <- readTVar tv
         ; check (n_left == 0)
         ; new_g1 <- newGate n; new_g2 <- newGate n
         ; writeTVar tv (n,new_g1,new_g2)
         ; return (g1,g2) }
```

注意，`awaitGroup` 在重新初始化 `Group` 对象的时候会新建两个 `Gate` 对象。这确保了当圣诞老人在书房里讨论时，一个新群可以同时处于集结之中；如果不新建 `Gate` 对象的话，新群中的小矮人便可能会将旧群中打瞌睡的那些家伙给挤出去（译注1）。

译注1：假设旧群满了，此时 `joinGroup` 已返回给该群中的每个小矮人两扇门，而圣诞老人调用的 `awaitGroup` 随后也观察到群已满，于是打开门，同时群被重新开放。但注意，群上附着的两扇门还是原来的两扇，这就表示，当另一组小矮人调用 `joinGroup` 的时候得到的返回值仍然还是那两扇门，于是新群中的小矮人便和旧群中的小矮人们同挤一扇门，如果这时（在进门时）旧群中的某个小矮人不幸睡着了（比如线程时间片用完了），便会被新群中的小矮人捷足先登了。

回顾一下这节，你可能会注意到，有些对群和门的操作是 IO 类型的（比如 newGroup 和 joinGroup），而有些则是 STM 类型的（比如 newGate 和 awaitGroup）。那么为什么这么设置呢？就拿 newGroup 来说吧，newGroup 有一个 IO 型的操作，也就是说我们无法在 STM 动作中调用它。但实际上我将 newGroup 做成 IO 型只是为了方便起见：我本可以把 newGroup 定义中的 atomically 调用去掉的，这样 newGroup 便成了 STM 类型的了，但这么一来每次调用 newGroup 的时候我们便都需要手动将其包在 atomically 之中了 (atomically(newGroup n))。而另一方面，将 newGate 做成 STM 操作的好处是能让它的可组合性 (composability) 更好，只不过就本应用程序来说 newGroup 并不需要这个可组合性，所以我才将它做成 IO 的，然而。由于我想在 newGroup 中调用 newGate，因此 newGate 的可组合性便有意义了，这便是我将 newGate 设为 STM 操作的原因。

一般来说，在设计一个库的时候，你应当尽可能地把函数的类型设为 STM。你可以把 STM 动作看作组合积木，小的 STM 动作可以（通过 do {...}, retry, orElse）组合起来成为更大的 STM 动作。然而，一旦你将一个块用 atomically 包裹起来之后，它就变成了一个 IO 动作，就再也不能利用 atomically 跟其他动作组合起来了（译注 2）。但 IO 动作也有它自己的优点：一个 IO 动作可以执行任意的、不可撤销的输入 / 输出（比如 launchMissiles）。

因此，好的库设计应当尽可能的暴露 STM 动作（而不是 IO 动作），因为 STM 动作是可组合的；它们的类型表明了它们不会执行不可撤销的操作。而另一方面，库的用户总是可以很容易地将 STM 动作包装成 IO 动作（外面加一层 atomically 调用即可），但反过来就不行了。

然而，有时候还是必须使用 IO 动作的。比如 operateGate。operateGate 中的两个对 atomically 的调用无法并成一个，因为第一个 atomically 调用具有一个外部可见的副作用（开门），而第二个 atomically 调用则需要等到所有的小矮人们都醒过来并穿过了这扇门之后才能执行结束，否则便会一直阻塞。（译注 3）因此 operateGate 必须是 IO 类型的。

译注 2：参见 newGroup 的实现

译注 3：所以，前一个 atomically 操作不生效，后一个 atomically 操作是不可能完成的。

主程序

我们首先把程序的骨架实现出来，注意，代表圣诞老人的函数（santa）还没有实现，但先不管它：

```
main = do { elf_group <- newGroup 3
           ; sequence_ [ elf elf_group n | n <- [1..10] ]
           ; rein_group <- newGroup 9
           ; sequence_ [ reindeer rein_group n | n <- [1..9] ]
           ; forever (santa elf_group rein_group) }
```

第一行创建了一个大小为 3 的群。第二行则稍微需要解释一下：它利用了所谓的列表内涵式（list comprehension）来创建一组 IO 动作，然后调用 sequence_ 来顺序执行它们。列表内涵式 “[e | x<-xs]” 读作“由一切 e 所构成的列表，其中 x 来自列表 xs”。因此本例中 sequence_ 的参数为：

```
[elf elf_group 1, elf elf_group 2, ..., elf elf_group 10]
```

这些调用每个都会返回一个 IO 动作，后者在被执行的时候会新建一个小矮人线程。而 sequence_ 函数则接受一组 IO 动作作为参数，返回的也是一个 IO 动作，后者被执行的时候会按列表中的顺序执行那组作为参数的 IO 动作（注 21）。

```
sequence_ :: [IO a] -> IO ()
```

elf 函数是基于 elf1 写的，但有两点区别。首先是我们想要 elf 重复执行，每重循环延迟一个不确定的时间间隔；其次，我们想要它在单独的线程中运行：

```
elf :: Group -> Int -> IO ThreadId
elf gp id = forkIO (forever (do { elf1 gp id; randomDelay }))
```

forkIO 将它的参数（一个动作）放在一个单独的 Haskell 线程中执行（见前面的小节“Haskell 中的副作用和输入 / 输出”）。forkIO 的实参是一个对 forever 的调用；forever，顾名思义，会将一个动作重复执行（一个与它类似但有微妙差别的函数是 nTimes，见“Haskell 中的副作用和输入 / 输出”）：

注 21：类型 “[IO a]” 读作“一个由类型为 IO a 的值构成的列表”。另外你可能会奇怪 sequence_ 后面为什么要加上一个下划线，其实这是因为另外还有一个与它相关的函数也叫 sequence，这个 sequence 的类型则是 [IO a] -> IO [a]，其功能是将一组动作执行后的结果收集到一个列表中。sequence 和 sequence_ 都定义在 Prelude 库中（Prelude 库是缺省导入的）。

```
forever :: IO () -> IO ()  
-- Repeatedly perform the action  
forever act = do { act; forever act }
```

最后，表达式(`elf1 gp id`)是一个IO动作，我们想要将它不确定地重复执行，每次执行随机延迟一段时间：

```
randomDelay :: IO ()  
-- Delay for a random time between 1 and 1,000,000 microseconds  
randomDelay = do { waitTime <- getStdRandom (randomR (1, 1000000))  
                  ; threadDelay waitTime }
```

主程序中剩下的部分含义就很明显了。创建9头驯鹿和创建10个小矮人的方式是一样的：

```
reindeer :: Group -> Int -> IO ThreadId  
reindeer gp id = forkIO (forever (do { reindeer1 gp id; randomDelay }))
```

主程序的最后一行代码利用`forever`来运行`santa`。下面我们就来说说最后一个问题是——圣诞老人（Santa）的实现。

圣诞老人的实现

圣诞老人是这个问题里面最有趣的，因为他会进行选择。他必须等到一组驯鹿或一组小矮人在那儿等他的时候才会继续行动。一旦他选择了是带领驯鹿还是小矮人之后，他便将他们带去做该做的事。圣诞老人的代码如下：

```
santa :: Group -> Group -> IO ()  
santa elf_gp rein_gp  
= do { putStrLn "-----\n"  
      ; (task, (in_gate, out_gate))  
      <- atomically (orElse  
                      (chooseGroup rein_gp "deliver toys")  
                      (chooseGroup elf_gp "meet in my study"))  
  
      ; putStrLn ("Ho! Ho! Ho! let's " ++ task ++ "\n")  
      ; operateGate in_gate  
      -- Now the helpers do their task  
      ; operateGate out_gate }  
where  
  chooseGroup :: Group -> String -> STM (String, (Gate, Gate))  
  chooseGroup gp task = do { gates <- awaitGroup gp  
                            ; return (task, gates) }
```

圣诞老人进行选择的关键就在那个`orElse`上。`orElse`首先会试图选择驯鹿（驯鹿优先），如果驯鹿没有准备好就选择小矮人。`chooseGroup`会对相应的群调用`awaitGroup`，并返回一个对偶（pair）“`(task, gates)`”，其中`task`是一个字符串，代表待执行的任务。

(“deliver toys” 或 “meet in my study”), gates 则本身又是一个对偶，它包含的是两扇门，圣诞老人通过操作这两扇门来带领一群小矮人或驯鹿完成任务。一旦在驯鹿和小矮人之间的选择完成了，圣诞老人便打印出一则消息表示待执行的任务，并依次操纵 (operateGate) 两扇门。

该实现工作起来自然是没问题的，但不妨让我们来看看另一个实现，这个实现更具一般性，因为圣诞老人的程序显示出了一个非常普遍的模式：一个线程（本例中是圣诞老人）在一个原子事务中作了一次选择，并根据选择的结果接着执行一个或多个事务。另一个典型的场景是：从多个消息队列中获取一则消息，并针对该消息做一些事情，然后重复这个过程。在圣诞老人问题里面，这里的后续操作对小矮人和对驯鹿基本是一样的——两种情况下圣诞老人都得打印一则消息并操纵两扇门。如果对小矮人的逻辑和对驯鹿的逻辑差别很大的话刚才上面那种做法就行不通了，一个补救的办法是使用一个布尔变量来表示到底选择了小矮人还是驯鹿，并根据具体选择了哪一方来决定做什么事情；但一旦选择的可能性多了，这种做法同样还是不够方便。下面是一个更好的解决方案：

```
santa :: Group -> Group -> IO ()  
santa elf_gp rein_gp  
= do { putStrLn "-----\n"  
      ; choose [(awaitGroup rein_gp, run "deliver toys"),  
                 (awaitGroup elf_gp, run "meet in my study")) ]  
where  
  run :: String -> (Gate, Gate) -> IO ()  
  run task (in_gate, out_gate)  
  = do { putStrLn ("Ho! Ho! Ho! let's " ++ task ++ "\n")  
        ; operateGate in_gate  
        ; operateGate out_gate }
```

choose 函数就像一个“保险命令”一样：它接受一组对偶 (pairs)，等到某个对偶的第一个元素可以“开火”了，便执行其第二个元素。因此 choose 的类型如下（注 22）：

```
choose :: [(STM a, a -> IO ())] -> IO ()
```

刚才提到“保险命令”的比喻，这里的“保险”就是对偶的第一个元素，即一个 STM 动作，返回类型为 a；当这个 STM 动作“准备好了”（不发生 retry）之后，choose 便可以将其返回值传给对偶的第二个元素，当然，后者必须是一个函数，且接受一个 a 类型的参数。了解了以上这些之后再来阅读 santa 的代码就应该毫无困难了。santa 利用

注 22： 在 Haskell 中，类型 $[ty]$ 的意思是一个元素类型为 ty 的列表。本例中 choose 的参数为一个由对偶 $((ty_1, ty_2))$ 构成的列表；其中对偶的第一个元素的类型为 STM a，第二个元素则是一个函数，类型为 $a -> IO()$ 。

`awaitGroup` 来等待一个群准备好；`choose` 拿到 `awaitGroup` 返回的两扇门之后便将它们传给 `run` 函数，后者依次操纵这两扇门——`operatorGate` 会一直阻塞，直到所有小矮人（或驯鹿）都穿过门之后才会返回。

`choose` 的代码虽然只有寥寥数行，但要真正弄明白它还是得费点脑筋的：

```
choose :: [(STM a, a -> IO ())] -> IO ()  
choose choices = do { act <- atomically (foldr1 orElse actions)  
                      ; act }  
where  
  actions :: [STM (IO ())]  
  actions = [ do { val <- guard; return (rhs val) }  
             | (guard, rhs) <- choices ]
```

首先来看 `actions`, `actions` 是一个列表，它的每个元素都是一个 STM 动作。`choose` 将 `actions` 和 `orElse` 用 `foldr1` 结合起来 (`foldr1 orElse [x1, ..., xn]` 的结果是 $x_1 \text{ orElse } x_2 \text{ orElse } x_3 \dots \text{ orElse } x_n$)。这些 STM 动作（即 `actions` 列表里面的元素）每个又都返回一个 IO 动作（所以才有 `STM (IO())` 这个类型），后者也就是它一旦被选中之后要做的事情。`choose` 首先在各个动作之间作一次原子选择，取得返回出来的动作 (`act`, 类型为 `IO()`)，然后执行该动作。而列表 `actions` 又是如何构造出来的呢？答案是只需对 `choices` 列表里面的每个对偶 (`guard`, `rhs`)，运行 `guard` (一个 STM 动作)，再将 `guard` 的返回值作为参数交给 `rhs`，并返回后者执行的结果即可。

编译并运行程序

以上便是这个例子的全部代码。要运行它，只需在程序开头再添上几个 `import` 语句即可（注 23）：

```
module Main where  
  import Control.Concurrent.STM  
  import Control.Concurrent  
  import System.Random
```

使用 GHC (Glasgow Haskell Compiler) 编译代码（注 24）：

```
$ ghc Santa.hs -package stm -o santa
```

注 23：代码也可以在这里下载到：<http://research.microsoft.com/~simonpj/papers/stm/Santa.hs.gz>。

注 24：GHC 是免费的，在这里下载：<http://haskell.org/ghc>。

最后运行：

```
$ ./santa
-----
Ho! Ho! Ho! let's deliver toys
Reindeer 8 delivering toys
Reindeer 7 delivering toys
Reindeer 6 delivering toys
Reindeer 5 delivering toys
Reindeer 4 delivering toys
Reindeer 3 delivering toys
Reindeer 2 delivering toys
Reindeer 1 delivering toys
Reindeer 9 delivering toys
-----
Ho! Ho! Ho! let's meet in my study
Elf 3 meeting in the study
Elf 2 meeting in the study
Elf 1 meeting in the study
...and so on...
```

对 Haskell 的一些思考

Haskell首先是一门函数式编程语言，但我觉得它同样也是世界上最漂亮的命令式语言。如果把 Haskell 当成一门命令式语言来看待的话，我们会发现，它具有一些不寻常的特性：

- 动作（有副作用）和纯值（无副作用）被严格区分开来。
- 动作也是真正的值。它们可以被作为参数、作为返回值、放在列表内等等，所有这些操作都不会带来副作用（译注 4）。

利用动作作为第一类值，我们便可以借助于它来定义应用相关的控制结构，而不是受制于语言设计者定义的那一套。例如，`nTimes` 就是一个简单的 `for` 循环结构。而刚才提到的 `choose` 则实现了一种可以称之为“保险命令”的功能。动作也是值的这一性质还有许多其他应用。在 `main` 函数中，我们利用 Haskell 丰富的表达力（列表内涵式）生成了一列动作，然后再利用 `sequence_` 来依次执行这些动作。同样，前面在定义 `helper1` 的时候，我们也是利用的这一性质：我们从一块代码中抽象出了一个动作，从而提升了代码的模块性。当然，圣诞老人的实现代码量本就不多，动用 Haskell 的这些强大的抽象能力仿佛有点杀鸡使用牛刀的感觉，不过一来这里只是为了展现 Haskell 的优点，二来对于大型程序来说，将动作作为值同样是非常重要的。

译注 4：因为并不会导致动作被执行。

此外，Haskell还有许多强大之处文中并没有提到，如高阶函数，惰性求值，数据类型，多态，类型类（type class）等等，因为本文关注的是并发。Haskell程序很少有像本文中的例子这样“命令式”的！如果想了解更多 Haskell 的知识，可以访问 <http://haskell.org>，上面有书、指南、编译器、解释器、库、邮件列表和许多其他资源。

结论

本文的主要目的是要让你相信，用 STM Haskell 写出的并发程序从根本上比利用传统的锁和条件变量写出的并发程序的模块性更好。不过首先值得注意的一点是事务内存帮我们完全避免了基于锁的并发编程中种种令人头疼的经典问题（见“生锈的锁”一节）。所有这些问题在 STM Haskell 中完全消失不见了。Haskell 的类型系统会防止你在原子块之外读写一个 TVar，而且，由于不再存在对程序员可见的锁，因此加哪把锁、按什么顺序加锁的问题也就不复存在了。此外 STM 还有其他好处，但这里限于篇幅我写不下了（包括不再有忘记唤醒以及异常和错误恢复这些令人头疼的问题）。

不过，我最想说的一个问题还是可组合性（composability）问题，正如“生锈的锁”一节提到的，这是基于锁的编程中一个最严重的问题。但在 STM Haskell 中，任何 STM 类型的函数都可以顺序地或通过选择来与其他任何 STM 类型的函数组合起来形成一个新的 STM 型的函数，新的函数能确保具有组成它的各个函数的原子属性。特别是阻塞（retry）和选择（orElse）这两个功能，如果用锁来实现的话从根本上就不具备模块性，然而在 STM Haskell 中则是完全模块化的。例如，考虑下面这个事务，它用到的 limitedWithdraw 函数曾在“阻塞和选择”一节定义：

```
atomically (do { limitedWithdraw a1 10
                  ; limitedWithdraw2 a2 a3 20 })
```

这个事务从阻塞中恢复的前提是账户 a1 上至少要有 10 块钱，而 a2 和 a3 之中则至少要有一个账户上多于 20 块钱。关键是，这么复杂的阻塞条件并不需要程序员显式写出，而且，如果 limitedWithdraw 系列函数是位于一个成熟的库中的话，程序员甚至根本不知道它们的阻塞条件是什么。总之一句话：STM 是模块性的，小的程序可以粘合成大的程序，无需暴露其内部实现。

本文只能说是对事务内存作了一个简单的概览，实际上事务内存还有许多其他有意思的主题文中没有提到，比如重要的有嵌套事务、异常、线程进展、饿死、不变式等。其中

许多在关于 STM Haskell 的一些论文中都讨论过（注 25）。

说实话，事务内存和 Haskell 可谓天造地设的一对。STM 的实现理论上可能要跟踪每一次内存读写，然而 Haskell 中的 STM 实现却只需跟踪 TVar 操作就行了，而 TVar 操作只占所有内存操作的极小一部分。此外，由于 Haskell 中的动作也是值，再加上 Haskell 丰富的类型系统，就使得我们无需对语言作任何扩展便能够提供强大的静态保证。不过话说回来，事务内存并非不适用于主流的命令式语言，虽然实现起来可能没这么优雅，并且可能需要更多的语言支持。目前 STM 是一个热门的研究课题；Larus 和 Rajwar 对这个领域的研究作了一个全面的概述（注 26）。

STM 之于传统的并发编程技术就好比高阶语言之于汇编语言——你仍然还有可能写出有问题的程序，但许多棘手的 bug 却不可能再出现了，而且关注程序的高阶性质也使得编程容易得多。虽然并发编程并无银弹，但 STM 看起来是一个很有前途的进展，它能帮你编写出更美的代码。

致谢

很多人对本文的改进提了很好的建议和意见：Bo Adler, Justin Bailey, Matthew Brecknell, Paul Brown, Conal Elliot, Tony Finch, Kathleen Fisher, Greg Fitzgerald, Benjamin Franksen, Jeremy Gibbons, Tim Harris, Robert Helgesson, Dean Herington, David House, Brian Hulley, Dale Jordan, Marnix Klooster, Chris Kuklewicz, Evan Martin, Greg Meredith, Neil Mitchell, Jun Mukai, Michal Palka, Sebastian Sylvan, Johan Tibell, Arthur van Leeuwen, Wim Vanderbauwheide, David Wakeling, Dan Wang, Peter Wasilko, Eric Willigers, Gaal Yahas, and Brian Zimmer。特别要感谢 Kirsten Chevalier, Andy Oram, 和 Greg Wilson 他们对文章作了非常详细的审阅。

注 25: Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy, "Composable memory transactions," ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '05), June 2005; Tim Harris and Simon Peyton Jones, "Transactional memory with data invariants," First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT '06), Ottawa, June 2006, ACM; Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh, "Lock-free data structures using STMs in Haskell," Eighth International Symposium on Functional and Logic Programming (FLOPS '06), April 2006.

注 26: James Larus and Ravi Rajwar, *Transactional Memory*, Morgan & Claypool, 2006.

第 25 章

句法抽象： syntax-case 展开器

R. Kent Dybvig

在我们编程时，某些模式会反复出现。比如，程序常常需要循环访问数组元素，递增或者递减变量的值，以及根据数值或者字符值执行多路条件式。编程语言的设计者一般通过加入专用的句法结构来解决这类问题，这些句法结构可以处理大多数常见模式。比如 C 语言提供了多种循环结构、多种条件结构和多种递增或者更改变量值的结构（注 1）。

有些模式没有那么常见，但在某类程序里会频繁出现，或者在某一个程序里频繁出现。这些模式甚至不一定被语言设计者所预见。通常来说，就算语言设计者事先想到了，他们无论如何也不会在语言核心加入处理这些模式的句法结构。

不过，语言设计者认识到这类模式的确存在，而且专用句法结构使得程序更简单也更容易阅读。所以他们有时在语言里加入句法抽象的机制，比如 C 的预处理宏，或者 Common

注 1：The C Programming Language, Second Edition, Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1988. 中文版为《C 程序语言》第 2 版。该书中、英文版均已由机械工业出版社出版——编辑注。

Lisp的宏（注2）。当没有这类辅助手段，或者当它们不足以解决特定问题时，我们可以利用诸如m4宏展开器（注3）（译注1）这类外部工具。

句法抽象工具和宏有几点重要区别。C的预处理器宏本质上基于语素，因此可以把宏调用替换为一系列语素。如果宏带有参数，那么传递给宏的文本会替换它的形式参数。Lisp宏基于表达式，从而允许单个表达式被替换为另外的表达式。替换表达式在Lisp系统内运算出来。如果宏包含子式，替换表达式的计算也基于这些子式（译注2）。

在这两种情况下，宏调用子式里出现的标识符的作用域等于它们在宏输出里的作用域，而不是它们在输入里的作用域。这样可能导致变量绑定时意外地捕获其他变量的引用。

例如下面把Scheme（注4）的or表达式变换为let和if的简单代码：

(or e₁ e₂) → (let ([t e₁]) (if t t e₂))

注意

Scheme是Lisp的方言。不熟悉Scheme的读者可以先阅读The Scheme Programming Language, Third Edition (R. Kent Dybvig, MIT Press)，在线阅读地址为：<http://www.scheme.com/tspl3>。

如果or表达式的第一项子式为真（任何非假值），那么它就必须返回第一项的值。表达式let用来命名该值，以免它被计算两次。

上述变换在大多数情况下都能正常工作，但如果标识符t在e₂内是自由变量（即t在e₂内没有绑定），那么运算结果就会出错，就象下面的表达式一样：

(let ([t #t]) (or #f t))

注2：Common Lisp: The Language, Second Edition, Guy L. Steele Jr., Digital Press, 1990。这本书出版于Common Lisp标准化之前，所以和标准有少量出入。

注3：The M4 Macro Processor, Brian W. Kernighan and Dennis M. Ritchie, 1979。

译注1：M4 Macro Processor是Brian和Dennis开发的一门通用宏语言。大多数Unix系统都配备了M4。M4就是Macro的意思：M加上ACRO这四个字母。M4配备多种处理文本的工具，可以用于代码生成，或者编译器的前端工具。

译注2：Lisp里用form（形式，方法）来指代表达式，包括特殊表达式。嵌套在表达式里的表达式叫做子式（subform）。

注4："Revised report on the algorithmic language Scheme," Richard Kelsey, William Clinger, and Jonathan Rees, editors, Higher-Order and Symbolic Computation, Vol. 11, No. 1, pp. 7-105, 1998. Also appeared in ACM SIGPLAN Notices, Vol. 33, No. 9, September 1998.

这个表达式的结果应为真值 #t。但是根据前面 or 的简单变换规则，这个表达式被展开为：

```
(let ([t #t])
  (let ([t #f])
    (if t t t)))
```

结果得到假值 #f。

问题发现后就容易解决了：新引入的绑定用生成的标识符。

```
(or e1 e2) → (let ([g e1]) (if g g e2))
```

这里的 g 是生成的（新建的）标识符。

Kohlbecker、Friedman、Felleisen 和 Duba 在关于健康宏展开的奠基性论文（注 5）里认为这类变量捕获问题暗藏凶险，因为一个变换可能在大量代码里都正确运行，但运行一段时间后却出错，而且出的错也许难以排除。

变量引用在宏调用的上下文里也能被绑定捕获。虽说引入变量绑定造成的无意捕获总能通过生成标识符解决，但变量引用就不一定有这么简单的解决方案了。下面的表达式里，if 以词法形式绑定 or 表达式的上下文里：

```
(let ([if (lambda (x y z) "oops")]) (or #f #f))
```

对 or 做第二次变换，该表达式展开为：

```
(let ([if (lambda (x y z) "oops")])
  (let ([g #f])
    (if g g #f)))
```

这里的 g 是新的标识符。表达式的值应该为 #f，但实际却为“oops”，因为本地绑定的过程 if 用在了原生 if 条件语句的地方。

保留诸如 let 和 if 这类关键字的名字限制了语言，倒也能针对关键字解决该问题。可是这样不是解决这类问题的通用方法。比如，下面对 increment 的变换引入对用户定义的变量 add1 的引用时，同样的情况就会发生：

```
(increment x) → (set! x (add1 x))
```

注 5：“Hygienic macro expansion,” Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba, Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, pp. 151-161, 1986.

为了解决这两类捕获问题，Kohlbecker 等人从 Barendregt 借来了“健康”(hygiene)这一名词，发明了健康宏展开的概念（注 6）。Barendregt 的 λ 算子健康条件等价于：当一个表达式的自由变量被替换到另一个时，我们可以假定这些变量不会被另一则表达式的绑定所捕获，除非该捕获被明确要求。Kohlbecker 等人把这一概念改动为下面的宏展开健康条件：

在完全展开的程序里，成为绑定实例的生成标识符只能绑定同一转录步骤中生成的变量。

实际应用中，这一要求迫使展开程序为了避免无意的捕获而在必要时重命名标识符。比如，用最初的 or 变换：

$(\text{or } e_1 \ e_2) \rightarrow (\text{let } ([t \ e_1]) \ (\text{if } t \ t \ e_2))$

把表达式：

$(\text{let } ([t \ #t]) \ (\text{or } \#f \ t))$

展开为下面的等价形式：

$(\text{let } ([t_0 \ #t]) \ (\text{let } ([t_1 \ \#f]) \ (\text{if } t_1 \ t_1 \ t_0)))$

这个表达式可以得出正确结果 $\#t$ 。同理，下面的表达式

$(\text{let } ([\text{if } (\lambda (x \ y \ z) \ "oops")]) \ (\text{or } \#f \ \#f))$

展开为下面的等价形式：

$(\text{let } ([\text{if}_0 \ (\lambda (x \ y \ z) \ "oops")]) \ (\text{let } ([t \ \#f]) \ (\text{if } t \ t \ \#f)))$

该形式可以得到正确结果 $\#f$ 。

本质上，健康宏展开实现了源代码上的词法作用域，而不健康的展开在展开后的代码上实现词法作用域。

健康宏展开只能保存变换自身能保存的词法作用域。变换可以生成明显破坏词法作用域的代码。下面对 let 的（不正确）变换展示了这点：

注 6: Introduction to the lambda calculus, "H. P. Barendregt, Nieuw Archief voor Wijskunde, Vol. 4, No. 2, pp. 337-372, 1984."

```
(let ((x e)) body) → (letrec ((x e)) body)
```

表达式 e 应该在变量 x 的绑定范围之外，可是因为 letrec 的语义（译注 3），在展开的代码里它处于绑定范围之内。

Kohlbecker 等人描述的健康宏展开算法（KFFD）巧妙优雅。它为宏引入的每个变量加上时间戳，然后在重命名词法绑定的变量时用时间戳来区分名字相同的变量。不过 KFFD 的某些缺点阻碍了它在实际中的直接应用。其中最严重的问题是缺乏对本地宏绑定的支持，以及在重命名和加时间戳时完全重写每个表达式带来的平方级的额外开销。

Chinger、Dybvi、Hieb 和 Rees 在 Scheme 修改报告（注 7）中开发的 syntax-rules 系统解决了这些缺点。 syntax-rules 系统基于模式的简单本质使得它可以被轻松高效地实现（注 8）。可惜，这也限制了该机制的利用，以至于许多有用的句法抽象难以实现，甚至不可能实现。

为了解决原来算法里的缺陷，但又不受 syntax-rules 的束缚，人们又开发了 syntax-case 系统（注 9）。该系统支持本地宏绑定，运行时只有常数量的额外开销，而且允许宏利用 Scheme 编程语言的全部表达能力。它还向上兼容 syntax-rules 。实际上 syntax-rules 可以用 syntax-case 写的简单宏表达。而且， syntax-case 允许同样的模式语言用于 syntax-rules 不能实现的“底层”宏。它还提供了用于故意捕获的机制（也就是说，允许健康条件被有选择地以直接的方式“弯曲”或者“打破”）。另外，它还处理了在真正实现时必须处理的一些实际问题，比如内部定义和宏展开中跟踪记录源代码信息。

这一切的代价是展开算法的复杂度增加，以及用于实现的代码量增加。因此，对整个实

译注 3： letrec 专门用于为递归函数绑定本地变量。它的形式是 $(\text{letrec} ((\text{var} \text{ value})...) \text{ exp1 exp2} ...)$ 。和 let 不同的是，变量 var 不仅在 letrec 的函数体内可见，在被绑定的值 value 里也可见。所以如果 value 里有同名变量，就可能产生冲突。

注 7：“Revised report on the algorithmic language Scheme,” William Clinger and Jonathan Rees, editors, LISP Pointers, Vol. 4, No. 3, pp. 1-55, July-September 1991.

注 8：“Macros that work,” William Clinger and Jonathan Rees, Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, pp. 155-162, January 1991.

注 9：“Syntactic abstraction in Scheme,” R. Kent Dybvig, Robert Hieb, and Carl Bruggeman, Lisp and Symbolic Computation, Vol. 5, No. 4, pp. 295-326, 1993.

现的全面细致的研究超越了本章的范畴。相反，我们会探索展开器的简化版。该简化版展示了宏展开的基本算法，以及实现它时最重要的考量。

syntax-case 简介

我们从几个简短的 syntax-case 例子入手。这些例子改自《Chez Scheme Version 7 User's Guide》(R. Kent Dybvig, Cadence Research Systems, 2005)。这本指南和 The Scheme Programming Language, Third Edition 里给出了更多的例子和更详细的说明。

下面 or 的定义展示了 syntax-case 宏定义的形式：

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_ e1 e2)
       (syntax (let ([t e1]) (if t t e2))))]))
```

形式 define-syntax 创建关键词绑定，把这里的关键词 or 绑定到一个变换过程（也就是变换器）上。在 define-syntax 被展开的时候，通过计算 define-syntax 内右边的 lambda 表达式得到该变换器。形式 define-case 用于解析输入，而形式 syntax 通过直观的模式匹配构造输出。模式 (_ e1 e2) 规定了输入的形状。这里的下划线 (_) 标注出关键词 or 在哪里出现，而模式变量 e1 和 e2 绑定到第一个和第二个子式上。模板 (let ([t e1]) (if t t e2)) 规定了把 e1 和 e2 从输入插入后的输出。

形式 (syntax template) 可以被简写为 #' template，所以前面的定义可以重写为：

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_ e1 e2)
       (syntax(let ([t e1]) (if t t e2))))]))
```

宏也可以通过 letrec-syntax 绑定到单个表达式里：

```
(letrec-syntax ((or (lambda (x)
                        (syntax-case x ()
                          [(_ e1 e2)
                           #'(let ([t e1]) (if t t e2))))])
              (or a b)))
```

宏可以递归，如下所示。下面的 or 定义可以处理任意数目的子式。为了处理两个基础情况和递归情况，我们需要多个 syntax-case 从句：

```
(define-syntax or
  (lambda (x)
```

```
(syntax-case x ()
  [(_ #'+f]
  [(_ e) #'e]
  [(_ e1 e2 e3 ...)
  #'(let ([t e1]) (if t t (or e2 e3 ...))))]))
```

`syntax-case` 的模式语言中带省略号的输入或输入形式匹配零个或多个形式。

这个例子里，`or` 的定义保证健康，所以引入的 `t` 的绑定，对 `let`、`if` 甚至 `or` 的应用都有恰当的作用域。如果我们想认为改动或破坏健康，可以通过过程 `datum->syntax` 实现。这个形式从任意符号表达式生成句法对象。该符号表达式内的标识符出现的位置等同于原来代码里第一个参量，模板标识符，出现的位置。

我们可以用这一事实来创建简单的 `method` 句法，让它隐含地把名字 `this` 绑定到第一个（对象）参数：

```
(define-syntax method
  (lambda (x)
    (syntax-case x ()
      [(k (x ...) e1 e2 ...)
       (with-syntax ([this (datum->syntax #'k 'this)])
         #'(lambda (this x ...) e1 e2 ...))))))
```

通过把输入里抽取的关键词当作模板变量，变量 `this` 就好像出现在 `method` 形式里一样。这样的话，

```
(method (a) (f this a))
```

等价于下面函数：

```
(lambda (this a) (f this a))
```

而且不用重命名来防止引入的绑定捕获源代码引用。

在 `method` 定义里用到的 `with-syntax` 形式创建了本地模式变量的绑定。它本身是用 `syntax-case` 写成的简单宏：

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      [(_ ((p e0) ...) e1 e2 ...)
       #'(syntax-case (list e0 ...) ()
                     [(_ ...) (begin e1 e2 ...)]))))))
```

过程 `datum-syntax` 可以用于任何表达式，比如下面 `include` 的定义：

```
(define-syntax include
  (lambda (x)
```

```

(define read-file
  (lambda (fn k)
    (let ([p (open-input-file fn)])
      (let f ([x (read p)])
        (if (eof-object? x)
            (begin (close-input-port p) '())
            (cons (datum->syntax k x) (f (read p)))))))
  (syntax-case x ()
    [(k filename)
     (let ([fn (syntax->datum #'filename)])
       (with-syntax ([e (... (read-file fn #'k))])
         #'(begin e ...))))]

```

形式 (include "filename") 起的作用是把名为“filename”的文件里的形式当作存在于 include 所在的地方。除了用 datum-syntax 以外，include 也用 datum-syntax 的逆操作符，syntax->datum，把子式 filename 变换为可以传给 open-input-file 的字符串。

展开算法

Syntax-case 的展开算法本质上是 KFFD 算法的惰性变种。该算法在输入表达式的抽象表示上操作，而不是在传统的符号表达式的表示上操作。该抽象表达式既封装了输入形式的表示，也封装了一项让算法可以决定形式内需要有标识符作用域的包装。该包装由记号和替代器组成。记号就像 KFFD 的时间戳，而且加入了宏自己产生的输出部分。

替代器在编译环境的帮助下把标识符映射到绑定。每当遇到绑定形式时，比如 lambda，替代器就被创建，并加进句法对象的包装里。这些句法对象代表了绑定形式的绑定作用域内的形式。

展开以递归且自顶向下的方式进行。当展开器遇到宏调用，就对遇到的形式调用相关的变换器，把它标上新建记号，然后用同样的记号再标注一次。同样的记号抵消了，所以只有宏生成的输出部分（也就是那些不是从输入拷贝到输出的部分）保留记号。

当展开器遇到核心形式时，就会生成展开器的输入语言里的一项核心形式（在我们这种情况下，也就是传统的符号表达式的表示）。在生成过程中，每项子式在需要时递归展开。变量应用通过替换机制被生成的名字取代。

表示

Syntax-case 机制最重要的方面是把程序源代码表示为抽象的句法对象。如前所述，句法对象不仅封装程序源代码的表示，而且还封装了一套包装。这套包装提供了代码内标识符的信息，足够用来实现健康：

```
(define-record syntax-object (expr wrap))
```

上例中 define-record 形式创造了一种新的带特定名字 (这里是 syntax-object) 的值和字段 (这里是 expr 和 wrap)，以及一套操作它们的过程。这些过程为：

make-syntax-object

返回一个新的句法对象，其中 expr 和 wrap 字段初始化为该过程参数的值。

syntax-object?

当且仅当它的参数是一个句法对象时返回真值。

syntax-object-expr

返回句法对象里 expr 字段的值。

syntax-object-wrap

返回句法对象里 wrap 字段的值。

完整的 syntax-case 实现也许还会在每个句法对象里包含在展开过程中用来跟踪源代码的信息。

如前所述，每个包装由一组记号和替换器组成。记号可由它们的对象身份来区别，而且不需要任何字段。

```
(define-record mark ())
```

替换器把符号名和一组记号映射到一条标签：

```
(define-record subst (sym mark* label))
```

标签和记号一样，用它们的身份区别，不需要任何字段：

```
(define-record label ())
```

由展开器维护的展开时环境将标签映射到绑定上。该环境的结构是传统的关联表。也就是说，一组点对。每个点对的 cons 是一个标签，而每个点对的 cdr 包含一个绑定。绑定包含一个类型（表示为一个符号）和一个值。

```
(define-record binding (type value))
```

包含的类型确定绑定的性质：macro 对应关键词绑定，而 lexical 对应语词变量的绑定。包含的值可以是任何用来设定绑定的额外信息，比如当绑定是关键词绑定时需要的变换过程。

生成展开器输出

展开器的输出是属于核心语言的简单符号表达式，因此大部分用 `quasiquote` 句法构建。句法 `quasiquote` 是用来创建列表结构的。例如，一个 `lambda` 表达式可以用形式参数 `var` 和函数体 `body` 创建：

```
'(lambda (, var) ,body)
```

不过，展开器的确需要创建新名字，并且通过 `gen-var` 辅助函数做到这一点。该辅助函数利用 Scheme 的基本操作把字符串变换为符号，或把符号变换为字符串。创建过程同时维护一个本地的序列计数器。

```
(define gen-var
  (let ([n 0])
    (lambda (id)
      (set! n (+ n 1))
      (let ([name (syntax-object-expr id)])
        (string->symbol (format "~s.~s" name n))))))
```

剥离句法对象

每当在输入里遇到 `quote` 形式时，展开器必须返回 `quote` 形式内常数内容的表示。要做到这一点，它必须用 `strip` 过程剥离所有内嵌句法对象和包装。`strip` 过程遍历它的输入里的句法对象和列表结构，并重新创建表示输入的符号表达式。

```
(define strip
  (lambda (x)
    (cond
      [(syntax-object? x)
       (if (top-marked? (syntax-object-wrap x))
           (syntax-object-expr x)
           (strip (syntax-object-expr x)))]
      [(pair? x)
       (let ([a (strip (car x))] [d (strip (cdr x))])
         (if (and (eq? a (car x)) (eq? d (cdr x)))
             x
             (cons a d)))]
      [else x])))
```

在表达式的任意分支发现句法对象或点对之外的东西（也即符号或立即值）时，遍历即可中止。当找到“顶部标注”的句法对象时（也即该对象包含独有的顶部标记），遍历也会中止：

```
(define top-mark (make-mark))

(define top-marked?
  (lambda (wrap)
```

```
(and (not (null? wrap))
      (or (eq? (car wrap) top-mark)
          (top-marked? (cdr wrap))))))
```

展开器在创建表示原始输入的句法对象时，特意用基部包含顶部标记的包装，使得剥离代码能判断它什么时候到达句法对象基部，从而不用再继续遍历。这一特性防止展开器对常数做无谓的遍历，以便保留共享结构和循环结构，同时也不会与输入里的句法对象混淆。

句法错误

我们的展开器通过 `syntax-error` 报句法错误。`syntax-error` 定义如下：

```
(define syntax-error
  (lambda (object message)
    (error #f "~-a ~s" message (strip object))))
```

如果把源码信息附加到句法对象上，那么我们就可以利用这些信息构造包含源码行数和字符位置的错误信息。

结构谓词

我们总能通过 `syntax-case` 形式的模式来确定句法对象的非原子结构。谓词 `identifier?` 确定一个句法对象是否代表标识符：

```
(define identifier?
  (lambda (x)
    (and (syntax-object? x)
         (symbol? (syntax-object-expr x))))))
```

同理，剥离句法对象后，谓词 `self-evaluating?` 用于判断该句法对象是否代表常数：

```
(define self-evaluating?
  (lambda (x)
    (or (boolean? x) (number? x) (string? x) (char? x))))))
```

创建包装

我们通过扩展包装，从而在句法对象里加入标注或替换物：

```
(define add-mark
  (lambda (mark x)
    (extend-wrap (list mark) x)))  
  
(define add-subst
  (lambda (id label x)
```

```

  (extend-wrap
    (list (make-subst
      (syntax-object-expr id)
      (wrap-marks (syntax-object-wrap id))
      label))
    x)))

```

如果只是局部包装句法对象，我们只需要创建封装了局部包装的结构的句法对象，由此扩展包装：

```

(define extend-wrap
  (lambda (wrap x)
    (if (syntax-object? x)
        (make-syntax-object
          (syntax-object-expr x)
          (join-wraps wrap (syntax-object-wrap x)))
        (make-syntax-object x wrap))))

```

合并两个包装几乎同添加标注列表一样简单。惟一复杂点的地方是当两个相同的标注碰到一起时，展开算法要把它们消掉。

```

(define join-wraps
  (lambda (wrap1 wrap2)
    (cond
      [(null? wrap1) wrap2]
      [(null? wrap2) wrap1]
      [else
        (let f ([w (car wrap1)] [w* (cdr wrap1)])
          (if (null? w*)
              (if (and (mark? w) (eq? (car wrap2) w))
                  (cdr wrap2)
                  (cons w wrap2))
              (cons w (f (car w*) (cdr w*))))))))])

```

操纵环境

环境把标签映射到绑定上，并且用关联表表示。因此，扩展环境需要加入一个单对。该单对把标签映射到绑定上。

```

(define extend-env
  (lambda (label binding env)
    (cons (cons label binding) env)))

```

标识符解析

确定与标识符联系的绑定是一个两步过程。第一步确定标识符包装里与该标识符联系的标签，而第二步则在当前环境里查询该标签：

```
(define id-binding
```

```
(lambda (id r)
  (label-binding id (id-label id) r)))
```

出现在标识符包装里的标记和替代器决定了相关联的标签，如果该标签存在的话。替代器把名字和标记列表映射到标签。名字同标识符名字不匹配的替代器被忽略。标记不匹配的也被忽略。用到的名字是符号，因此可以用比较指针相等的操作符 `eq?` 比较。

相关的所有标记是在替换前叠加到包装上那些。因此，与替代器的标记做比较的那套标记在搜索包装的过程中会被相应改变。起始的标记集合是包装里的所有标记。在搜索包装里匹配替代的时候，每当遇到一个标记，列表里的第一个标记就会被去掉：

```
(define id-label
  (lambda (id)
    (let ([sym (syntax-object-expr id)]
          [wrap (syntax-object-wrap id)])
      (let search ([wrap wrap] [mark* (wrap-marks wrap)])
        (if (null? wrap)
            (syntax-error id "undefined identifier")
            (let ([w0 (car wrap)])
              (if (mark? w0)
                  (search (cdr wrap) (cdr mark*))
                  (if (and (eq? (subst-sym w0) sym)
                           (same-marks? (subst-mark* w0) mark*))
                      (subst-label w0)
                      (search (cdr wrap) mark*))))))))))
```

如果包装里没有匹配的替代器，那么该标记符则没有定义，句法错误也相应发出。相反，我们也可以把这样的标识符的应用当作全局标量的引用。

名为 `id-label` 的过程通过 `wrap-marks` 取得标记的初始列表，并用 `same-marks?` 谓词来比较标记列表：

```
(define wrap-marks
  (lambda (wrap)
    (if (null? wrap)
        '()
        (let ([w0 (car wrap)])
          (if (mark? w0)
              (cons w0 (wrap-marks (cdr wrap)))
              (wrap-marks (cdr wrap)))))))

(define same-marks?
  (lambda (m1* m2*)
    (if (null? m1*)
        (null? m2*)
        (and (not (null? m2*))
             (eq? (car m1*) (car m2*))
             (same-marks? (cdr m1*) (cdr m2*)))))))
```

一旦发现一个标签，就用 id-binding 找到可能存在的关联绑定。函数 id-binding 通过 assq 过程进行关联列表的查询。如果找到关联，则返回关联里 cdr 中的绑定：

```
(define label-binding
  (lambda (id label r)
    (let ([a (assq label r)])
      (if a
          (cdr a)
          (syntax-error id "displaced lexical")))))
```

如果没有找到绑定，那么该标识符就是“误置语词”。当宏在输出里错误地插入某个标识符的引用，而该标识符在宏输出的上下文里又不可见时，该错误就发生了。

展开器

知道了处理包装和环境的机制，展开器就简单明白了。表达式展开器 exp 处理宏调用、词法变量的引用、函数应用、核心形式以及常数。宏调用有两种形式：宏关键词的单件引用和宏关键词在第一位的结构化形式。

exp 过程接受三个参数：句法对象 x、运行时环境 r 和元环境 mr。运行时环境用于处理代码在展开器输出里出现的常规表达式，而元环境用于处理变换器表达式（比如 letrec-syntax 绑定的右侧）。这些常规表达式在展开时求值和应用：

```
(define exp
  (lambda (x r mr)
    (syntax-case x ()
      [id
       (identifier? #'id)
       (let ([b (id-binding #'id r)])
         (case (binding-type b)
           [(macro) (exp (exp-macro (binding-value b) x) r mr)]
           [(lexical) (binding-value b)]
           [else (syntax-error x "invalid syntax"))]))
      [(e0 e1 ...)
       (identifier? #'e0)
       (let ([b (id-binding #'e0 r)])
         (case (binding-type b)
           [(macro) (exp (exp-macro (binding-value b) x) r mr)]
           [(lexical)
            `((, (binding-value b) ,@(exp-exprs #'(e1 ...) r mr)))
           [(core) (exp-core (binding-value b) x r mr)]
           [else (syntax-error x "invalid syntax"))])
      [(e0 e1 ...)
       `((, (exp #'e0 r mr) ,@(exp-exprs #'(e1 ...) r mr)))
      [_
       (let ([d (strip x)])
         (if (self-evaluating? d)
             d
             (syntax-error x "invalid syntax"))))))]))
```

`exp-macro` (随后详述) 用来处理宏调用。之后宏被再次展开。词法变量被改写为绑定值，也即生成的变量名。函数应用被改写为列表，同传统的 Lisp 和 Scheme 的符号表达式句法一致。在改写过程中，子式被递归展开。核心形式用 `exp-core` (随后详述) 处理。任何回到表达式展开器的递归都由核心变换器明确地执行。常数重写为剥离了句法包装后的常数值。

展开器通过 `syntax-case` 和 `syntax` (简写形式为 `#'template`) 解析和指向输入，或相关的部分。因为扩展器也用来实现 `syntax-case`，这也许看起来是个悖论。实际情况是，上一版本的展开器引导生成下一版本的展开器。如果不用 `syntax-case` 和 `syntax`，写扩展器要琐碎得多。

`exp-macro` 过程将变换过程 (宏绑定的值) 应用到整个宏形式上，亦即单个宏关键字或头部是宏关键字的结构化表达式。`exp-macro` 过程先向输入形式的包装加入一个新建标记，再把同样的标记应用到输出形式。这第一个标记用作消掉第二个标记的“反标记”，所以净效应是该标记只粘连在变换器引入的输入部分。因此，该标记可以惟一地确定这步翻译引入的代码：

```
(define exp-macro
  (lambda (p x)
    (let ([m (make-mark)])
      (add-mark m (p (add-mark m x))))))
```

`exp-core` 过程仅仅把给定的变换器应用到输入形式上：

```
(define exp-core
  (lambda (p x r mr)
    (p x r mr)))
```

用来处理函数应用子式的 `exp-exprs` 过程仅仅把扩展器映射到形式上：

```
(define exp-exprs
  (lambda (x* r mr)
    (map (lambda (x) (exp x r mr)) x*)))
```

核心变换

这里描述了若干核心形式 (`quote`、`if`、`lambda`、`let` 和 `letrec-syntax`) 的变换器。加入其他核心形式的变换器，比如 `letrec` 或 `let-syntax`，也简单明了。

`exp-quote` 过程生成代表 `quote` 形式的符号表达式，其中数据的值从句法包装中剥离出来：

```
(define exp-quote
```

```

(lambda (x r mr)
  (syntax-case x ()
    [(_ d) `'(quote ,(strip #'d))]))))

```

exp-if 过程生成代表 if 形式的符号表达式，其中子式递归地展开：

```

(define exp-if
  (lambda (x r mr)
    (syntax-case x ()
      [(_ e1 e2 e3)
        `'(if ,(exp #'e1 r mr)
              ,(exp #'e2 r mr)
              ,(exp #'e3 r mr))))])

```

exp-lambda 过程处理只带单个形式参数和单个函数体的 lambda 表达式。通过扩展使得它处理多个参数也简单明了。处理任意多 lambda 函数体就没那么简单了，包括处理内部定义。不过支持内部定义超出了本章的范围。

下面的函数生成表示 lambda 表达式的符号表达式时，也为形式参数创建了程序生成的变量名。lambda 表达式的函数体的包装上也加入了把标识符映射到新标签的替代器。环境也被扩展，加入标签和词法绑定间的关联信息。该关联信息的值是在递归处理函数体时生成的变量：

```

(define exp-lambda
  (lambda (x r mr)
    (syntax-case x ()
      [(_ (var) body)
        (let ([label (make-label)] [new-var (gen-var #'var)])
          `'(lambda (,new-var)
                ,(exp (add-subst #'var label #'body)
                      (extend-env label
                                  (make-binding 'lexical new-var)
                                  ,r)
                      ,mr))))]))

```

由于元环境不应该包含词法变量的绑定，我们不需要将其扩展。

过程 exp-let 对单绑定的 let 形式进行变换。这同对 lambda 的变换相似，不过更为复杂：

```

(define exp-let
  (lambda (x r mr)
    (syntax-case x ()
      [(_ ([var expr]) body)
        (let ([label (make-label)] [new-var (gen-var #'var)])
          `'(let ([,new-var ,(exp #'expr r mr)])
              ,(exp (add-subst #'var label #'body)
                    (extend-env label
                                (make-binding 'lexical new-var)
                                ,r)
                    ,mr))))]))

```

```
r)
mr))))]))
```

let的函数体在let创建的绑定范围之内，所以用扩展的包装和环境来展开。右边的表达式，expr，则并不在绑定范围之内，因此是用最初的包装和环境来展开的。过程exp-letrec-syntax处理单绑定的letrec-syntax形式。同处理lambda和let一样，函数体的包装上加入了一个替换器。该替换器把绑定的标识符（这里的标识符与其说是变量不如说是关键词）映射到新建标签。并且，当函数体被递归处理时，环境里加入了标签和绑定间的关联信息。绑定是宏绑定而不是词法绑定，并且绑定值是递归展开和计算letrec-syntax形式右边表达式的结果。

相对于let，右边表达式也使用把关键词映射到标签的替换器包装，并且用扩展的环境展开。这样让宏可以递归。如果处理的形式是let-syntax而不是letrec-syntax，就做不到这一点了。通过展开letrec-syntax形式得到的输出仅包括递归调用展开器来处理形式体得到的输出：

```
(define exp-letrec-syntax
  (lambda (x r mr)
    (syntax-case x ()
      [(_ ((kwd expr)) body)
       (let ([label (make-label)])
         (let ([b (make-binding 'macro
                               (eval (exp (add-subst #'kwd label #'expr)
                                         mr mr))))])
           (exp (add-subst #'kwd label #'body)
                (extend-env label b r)
                (extend-env label b mr))))]))
```

在这种情况下运行时和元环境都被扩展，因为二者都能调用变换器。

解析和构造句法对象

宏用模式匹配的风格写成，通过syntax-case匹配和分解输入，并用syntax重新构造输出。实现模式匹配和重新构造超出本章的范围，不过下面的低级操作符可以用作上述实现的基础。我们可以用下面三个操作符实现syntax-case形式。这些操作符把句法对象当作抽象的符号表达式：

```
(define syntax-pair?
  (lambda (x)
    (pair? (syntax-object-expr x)))))

(define syntax-car
  (lambda (x)
    (extend-wrap
      (syntax-object-wrap x)))
```

```

(car (syntax-object-expr x)))))

(define syntax-cdr
  (lambda (x)
    (extend-wrap
      (syntax-object-wrap x)
      (cdr (syntax-object-expr x)))))


```

`syntax-car`和`syntax-cdr`的定义利用帮助函数`extend-wrap`把点对的包装推送到`car`和`cdr`上。该帮助函数在“创建包装”一节里定义。

同理，我们也能从下面跟基础的`syntax`版本构建`syntax`。该基础版本处理常数输入，但不能处理模式变量或省略号：

```

(define exp-syntax
  (lambda (x r mr)
    (syntax-case x ()
      [(_ t) `(quote ,#',t)])))

```

本质上，`syntax`的简化版就同`quote`一样，只不过它不会剥除封装值。相反，它不去改动句法包装。

比较标识符

展开器基于标识符的用途比较标识符。标识符也许被当作符号。这时展开器用指针等价操作符`eq?`对标识符的符号名做比较。根据标识符的用途，它们也许被当作宏输出的自由或绑定变量。展开器由此对它们做相应的比较。

向宏输出引入宏绑定之外的标识符时，如果两个标识符解析为同意绑定，那么`free-identifier=?`便认为它们等价。如前面“标识符解析”一节所述，我们通过比较从标识符解析出的标签来测试等价性：

```

(define free-identifier=?
  (lambda (x y)
    (eq? (id-label x) (id-label y))))

```

谓词`free-identifier=?`通常用于检查附加关键词，比如`cond`或`case`中的`else`。

如果一个标识符的引用被包含另一标识符的绑定捕获，那么`bound-identifier=?`则认为这两个标识符等价。这通过比较这两个标识符的名字和标记完成：

```

(define bound-identifier=?
  (lambda (x y)
    (and (eq? (syntax-object-expr x) (syntax-object-expr y))
         (same-marks?
           (wrap-marks (syntax-object-wrap x)))))

```

```
(wrap-marks (syntax-object-wrap y)))))
```

谓词 bound-identifier=? 常用于检查绑定形式里重复标识符错误，比如在 lambda 或 let 里。

转换

符号表达式到句法对象的转换由 datum-syntax 执行。这只需要把包装从模板标识符变换到符号表达式：

```
(define datum->syntax
  (lambda (template-id x)
    (make-syntax-object x (syntax-object-wrap template-id))))
```

相反的转换则需要把包装从句法对象上剥除，所以 syntax-datum 就是 strip：

```
(define syntax->datum strip)
```

开始展开

万事具备，现在可以在核心语言里把包含宏的 Scheme 表达式展开了。主要的展开器只提供初始包装和环境。它们包括核心形式和基本结构的名字及绑定：

```
(define expand
  (lambda (x)
    (let-values (((wrap env) (initial-wrap-and-env)))
      (exp (make-syntax-object x wrap) env env))))
```

初始包装包括一套替代器和初始环境。这些替代器把预先定义的标识符映射到新建标签，而初始环境把这而标签联系到对应的绑定：

```
(define initial-wrap-and-env
  (lambda ()
    (define id-binding*
      `((quote . ,(make-binding 'core exp-quote))
        (if . ,(make-binding 'core exp-if))
        (lambda . ,(make-binding 'core exp-lambda))
        (let . ,(make-binding 'core exp-let))
        (letrec-syntax . ,(make-binding 'core exp-letrec-syntax))
        (identifier? . ,(make-binding 'lexical 'identifier?))
        (free-identifier=? . ,(make-binding 'lexical 'free-identifier=?))
        (bound-identifier=? . ,(make-binding 'lexical 'bound-identifier=?))
        (datum->syntax . ,(make-binding 'lexical 'datum->syntax))
        (syntax->datum . ,(make-binding 'lexical 'syntax->datum))
        (syntax-error . ,(make-binding 'lexical 'syntax-error))
        (syntax-pair? . ,(make-binding 'lexical 'syntax-pair?))
        (syntax-car . ,(make-binding 'lexical 'syntax-car))
        (syntax-cdr . ,(make-binding 'lexical 'syntax-cdr))))
```

```

(syntax . ,(make-binding 'core exp-syntax))
(list . ,(make-binding 'core 'list)))
(let ([label* (map (lambda (x) (make-label)) id-binding*)])
  (values
    `(@(map (lambda (sym label)
               (make-subst sym (list top-mark) label))
            (map car id-binding*)
            label*)
         ,top-mark)
      (map cons label* (map cdr id-binding*))))))

```

除了列出的条目，初始环境也应包括我们还没有实现的内置句法形式的绑定（比如 letrec 和 let-syntax），以及所有内置 Scheme 过程。它也该包括完整版本的 syntax，而且当有 syntax-pair?、syntax-car 和 syntax-cdr 时，它还应包括 syntax-case。

例子

我们现在全程跟踪本章开头的例子：

```
(let ([t #t]) (or #f t))
```

我们假设函数 or 已被定义，用于完成本章开头描述的变换。它的定义如下，同“syntax-case 简介”一节的定义等价：

```

(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_ e1 e2) #'(let ([t e1]) (if t t e2))))))

```

最开始，展开器接受表达式为 (let ([t #t]) (or #f t)) 的句法对象。除了初始包装的内容（为简略起见，我们不在这里讨论），包装为空。我们把表达式和可能有的包装放在尖括号里，用以表征句法对象：

```
<(let ((t #t)) (or #f t))>
```

展开器也接受初始环境。我们假设促使环境包含宏的绑定，以及核心形式和内置过程的绑定。同样，为简略起见，我们省略掉关于这些环境的描述。我们也省略掉元环境——因为我们不展开任何变换表达式，它们在这里没用。

因为表达式 let 出现在初始包装和环境里，它被当作核心形式。let 的变换器在输入环境中递归地展开右手边表达式 #t，返回 #t。它也用把 x 映射到新建标签 11 的扩展包装来递归地展开表达式体：

```
<(or #f t) [t x () → 11]>
```

替换器在括号内，其中名字和标记用符号 \times 隔开，而标签跟在右箭头后。

环境也被扩展，用新建名字 $t.1$ 把标签映射到 `lexical` 类型上：

```
11 → lexical(t.1)
```

`or` 形式被当作宏调用，所以 `or` 的变换器被调用，生成新的表达式。对该表达式的求值在同样环境中进行：

```
<(<let> ((<t> #f))
  (<if> <t> <t> <t m2 [t x () → 11]>)
  m2>
```

表示引入的标识符 t 的句法对象和从输入中提取的标识符 t 是不同的。了解两者的区别才能判断在展开时怎么重新命名它们，所以至关重要。命名方法细述如下。

出现在 `let` 右边的 `#f` 是句法对象。它的包装技术上说与输入中提取的 t 相同。不过常数的包装并不重要，所以为简单起见，我们把它当成没有包装的对象。

我们还有另外一个核心 `let` 表达式。在甄别和解析 `let` 表达式的过程中，标记 $m2$ 被推压到子式上：

```
(<let m2> ((<t m2> #f)
  (<if> <t> <t> <t m2 [t x () → 11]>)
  m2>)
```

`let` 的变换器递归地展开右手边表达式 `#f`，返回 `#f`，然后用扩展包装递归地展开表达式体。该包装用标记 $m2$ 把引入的 t 映射到新建标签 12 上：

```
<(<if> <t> <t> <t m2 [t x () → 11]>
  [t x (m2) → 12]
  m2>)
```

环境也被扩展，借助新建名字 $t.2$ 把标签映射到 `lexical` 类型的绑定上：

```
12 → lexical(t.2), 11 → lexical(t.1)
```

得到的表达式被识别为核心形式 `if`。在识别和解析的过程中，展开器把外部的替代器和标记压到部件上。上次出现的 t 的包装上已有标记 $m2$ 。该标记与外包装上的标记 $m2$ 消掉，于是 t 没有标记：

```
(<if [t x (m2) → 12] m2>
  <t [t x (m2) → 12] m2>
  <t [t x (m2) → 12] m2>
  <t [t x (m2) → 12] [t x () → 11]>)
```

宏 `if` 的转换器在输入环境里递归地处理 `if` 的子式。首先：

```
<t [t x (m2) → 12] m2>
```

被认作标识符引用，因为该表达式为符号(t)。包装里的替换器也可用于这种情况，因为名字(t)和标记(m2)相同。因此，展开器在环境里查找12，并发现它映射到词法变量t.2。第二个子式与之相同，所以也映射到t.2。不过，第三个子式就不同了：

```
<t [t x (m2) → 12] [t x () → 11]>)
```

该标识符没有m2标记，所以第一个替换器不适用，尽管名字相同。第二替换器也不适用，因为它包含相同的名字和相同的一套标记（没有哪个超出省略的初始包装中置顶标记的范围）。展开器由此在环境里查找11，并发现它映射到t.1。

展开快结束时，表达式if被重新构造为：

```
(if t.2 t.2 t.1)
```

靠内的表达式let被重新构造为：

```
(let ([t.2 #f]) (if t.2 t.2 t.1))
```

靠外的表达式let被重新构造为：

```
(let ([t.1 #t]) (let ([t.2 #f]) (if t.2 t.2 t.1)))
```

这正是我们想要的结果。新建的名字是什么并不重要，只要它们是惟一值。

结论

这里描述的简化展开器展示了完整实现syntax-case需要的基本算法，但并没有涉及模式匹配机理的复杂问题，内部定义的处理以及通常由展开器处理的其余核心形式。单绑定的lambda、let和letrec-syntax也由展开器实现，环境的表示为这些形式特别设计。实际上我们通常用可以高效处理多个绑定的表示。虽说这些额外功能不容易实现，它们和展开算法在概念上彼此独立。

Syntax-case展开器扩展了KFFD健康宏展开算法。新增功能中包括支持本地句法绑定和可控捕获，而且消除了KFFD算法在展开时带来的平方级额外开销。

KFFD算法简洁优雅。基于它的展开器当然也可以是一段美妙代码。另一方面，syntax-case展开器也必须复杂得多。然而，这并不有损它的美妙。复杂的软件仍有精妙之处，只要它结构完善并且能完成既定的任务。

节省劳动的架构： 一个面向对象的 网络化软件框架

William R. Otte & Douglas C. Schmidt

开发网络化的应用软件已属不易，而开发可复用的网络应用软件更加困难。首先，分布式系统有着天生的复杂性。比如如何以最优的方式将应用服务映射到硬件结点；如何同步服务的初始化；以及，如何隐藏局部故障同时又保证可用性。这些复杂性甚至能使经验老道的软件开发者感到受挫，因为它们来自那些根植于网络编程领域的挑战。

不幸的是，开发者还需要控制随机的复杂性，比如底层不可移植的编程接口，以及使用面向函数的设计技术，这种设计技术在需求更新和（或）平台升级时常常需要繁冗的、易于出错的修订。这类复杂性大部分源自于人们在以前的网络化软件开发中所采用的软件工具和技术的局限性。

尽管面向对象技术已应用到诸如图形用户界面和生产力套件（productivity-tools）的许多领域，在网络化软件领域，人们仍然使用C级别的操作系统（Operating System, OS）应用编程接口（Application Programmatic Interface, API），比如UNIX的socket API或者Windows的线程API。网络编程中许多随机的复杂性都来自于这类C级别的OS API，它们不具有类型安全性，常常不是可重入的，而且也不能在不同的操作系统平台间移植。C API是在现代设计方法和技术广泛采用之前设计出来的，因此它们鼓励开发者从功能上按照处理的步骤以自顶向下的方式分解问题，而不是使用面向对象的设计和

编程技术。从过去几十年来软件被开发的经历来看，对于稍微大一点的软件，基于功能的分解都会使软件的维护和演化更加复杂化，因为稳定的设计很少将重心放在功能需求上（注 1）。

幸运的是，20多年来软件设计、实现技术以及程序设计语言的进步使得编写和重用网络化软件比以前容易多了。特别地，面向对象程序设计语言（如 C++、Java 以及 C# 等）跟设计模式（如“包装外观”（Wrapper Facades）模式（注 2），“适配器”（Adapte）模式和“模板方法”（Template Method）模式（注 3）等）和框架（框架的例子有像 ACE（注 4）那样的主机基础中间件（host infrastructure middleware），Java 网络编程类库（注 5）以及一些相似的主机基础中间件）相结合，有助于封装底层功能性的 OS API，并将不同平台间的语法和语义区别隐藏起来。结果是，开发者可以将精力集中于他们软件中特定于应用的行为和特性上，而不是重复地羁绊于底层网络和操作系统基础设施编程的复杂性上。

将模式和框架应用于网络化软件的一个最重要的好处是它们可以帮助开发者构建可复用的架构，这种架构能够：（1）捕捉到特定领域一般性的结构和行为；（2）使有选择性地改变或替换各种算法、策略和机制而不影响架构的其他部分变得相对容易。一方面网络化软件的开发者将可以在他们的程序中应用设计良好的面向对象框架，另一方面关于如何创建这样一个框架的知识仍然是一门黑色艺术，这种艺术的获得只能来自于之前广泛的（也是代价高昂的）尝试和错误。

注 1： 《面向对象软件构造（第 2 版）》（Object-Oriented Software Construction, Second Edition），Bertrand Meyer 著，Prentice Hall，1997。

注 2： 《面向模式的软件体系结构，第 2 卷：用于并发和网络化对象的模式》（Pattern-Oriented Software Architecture, Vol.2: Patterns for Concurrent and Networked Objects），Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann 著，John Wiley and Sons 2000。本书中文版已由机械工业出版社出版。

注 3： 《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software），Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides 著，Addison-Wesley, 1995。本书中文版、英文版及双语版均已由机械工业出版社出版。

注 4： 《C++ 网络编程，第 2 卷：基于 ACE 和框架的系统化复用》（C++ Network Programming, Vol.2: Systematic Reuse with ACE and Frameworks），Douglas C. Schmidt and Stephen D. Huston 著，Addison-Wesley Longman, 2003。

注 5： 《Java Network Programming》，第 3 版，Elliotte Rusty Harold 著，O'Reilly，2004。

要做出一个灵活的、能够适当地扩展或收缩以满足新需求的面向对象设计，除了常规的挑战之外，网络化软件还必须经常在一系列不同的操作环境中高效地、可伸缩地运行。本章的目标就是通过具体的案例来系统地解剖一个有代表性的网络化软件的设计及实现，从而帮你解开面向对象框架这门黑色艺术的神秘面纱。

总体而言，我们的解决方案的漂亮之处就在于运用模式和面向对象技术来平衡各种关键的领域特定需求：比如可复用性、可扩展性和性能要求。特别地，我们的方法能使开发者很容易从中识别出通用的设计、编码模型，从而提高重用度。同时，它也提供了一种通用的、参数化的封装变化的方式，从而提供可扩展性和可移植性。

示例程序：日志服务

我们的研究案例所基于的面向对象软件是一个网络化的日志服务。如图 26-1 所示，该服务由客户端程序和中心日志服务器组成，客户端程序产生日志记录并把它发送给日志服务器，而日志服务器则接收并存储这些日志记录以方便日后的审查和处理。

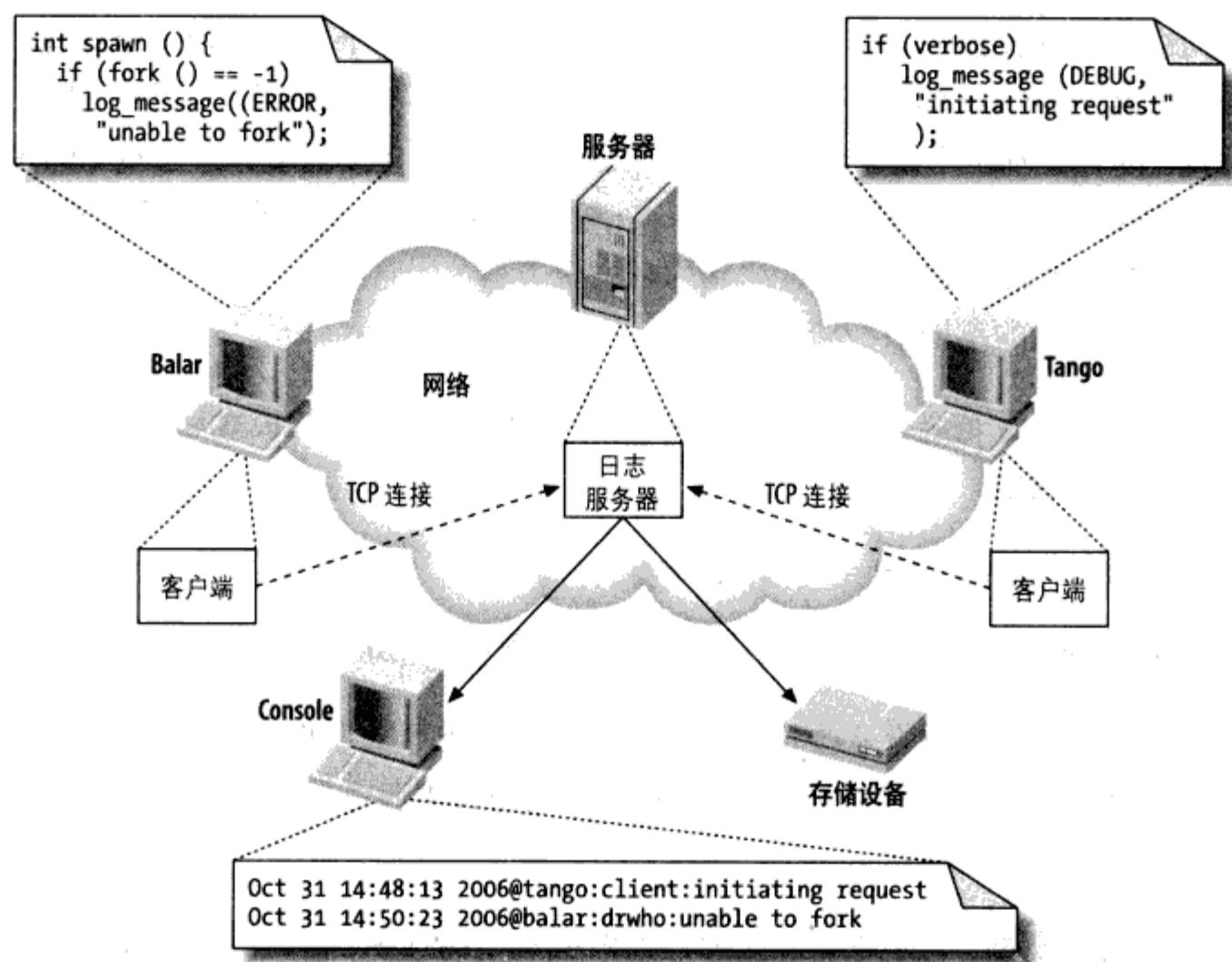


图 26-1：一个网络化日志服务的架构

网络化日志服务的日志服务器部分（位于图 26-1 的中心）提供了一个示范面向对象网络

化软件之美的理想案例，因为它在以下这些维度（dimension）上示范了设计时的可变性，开发者在实现这样一个服务器时，可以从每一维上挑选他认为合适的选择。

- 不同的进程间通信（interprocess communication, IPC）机制（比如 socket、SSL、共享内存、TLI、命名管道，等等），开发者可以用来发送和接收日志记录。
- 不同的并发模型（比如交互式、响应式、每连接一线程（thread-per-connection）、每连接一进程（process-per-connectin）、各种类型的线程池，等等），开发者可以用来处理日志记录。
- 不同的锁机制（比如线程级或进程级的递归互斥体（mutex）、非递归互斥体、读/写锁、空互斥体（null mutex），等等），开发者可以用来串行化（serialize）对资源的访问，比如对一个多线程共享的请求记数的访问。
- 客户端可以向服务器发送的不同的日志记录格式。在服务器端收到日志记录时，可通过不同的方式来处理记录，比如，打印到命令行窗口，保存到单个文件中，甚至每客户一个文件从而最大化磁盘写入的并行性。

实现不同组合中的任何一个都是比较直接的，比如，为每一个连接运行一个线程、日志服务器使用基于 socket 的 IPC、使用线程级的非递归互斥体。相反，一个“设定一次，到处运行”的方案却是不恰当的，它无法满足所有日志服务的需要，因为不同的用户需求和不同的操作环境对时间空间效率的折衷、开发成本和开发进度的影响是大不相同的。因此，最大的挑战就是设计一个可配置的日志服务器，使它易于扩展，从而在它面对新的需求时所要做的事情最少。

克服这一挑战的关键是全面深入的理解开发这个 OO 框架所需的模式和相关设计技术，这个框架要能高效地：

- 在基类和范型类中捕捉一般性的结构和行为。
- 通过子类或者为泛型类提供具体类参数来达到对行为的选择性配置。

图26-2给出一个实现了所有这些目标的面向对象的日志服务器框架设计。这个设计的核心是 Logging_Server 类，通过运用以下技术，它为日志服务器定义了一般的结构和功能：

- C++参数化类型，它使开发者能够延迟数据类型的选择，直到范型类或函数实例化时才正式选择其中使用的数据类型。

- 模板方法模式 (Template Method pattern) (译注 1) 它可以给出一个算法的轮廓, 而把个别的步骤委派给可被子类覆写 (overridde) 的方法。
 - 包装外观模式 (Wrapper Facade pattern), 它把非面向对象的 API 和数据封装到类型安全的面向对象类中。

`Logging_Server`类的子类和具体实例将详细定义这个一般的可复用架构，通过选择想要的IPC机制、并发模型以及锁机制来定制日志服务器行为中的可变步骤。于是，`Logging_Server`类就成了一个生产线架构（注6），它定义了一整套类的集合，这些类联合起来就为一类相关的日志服务器定义了一个可复用的设计。

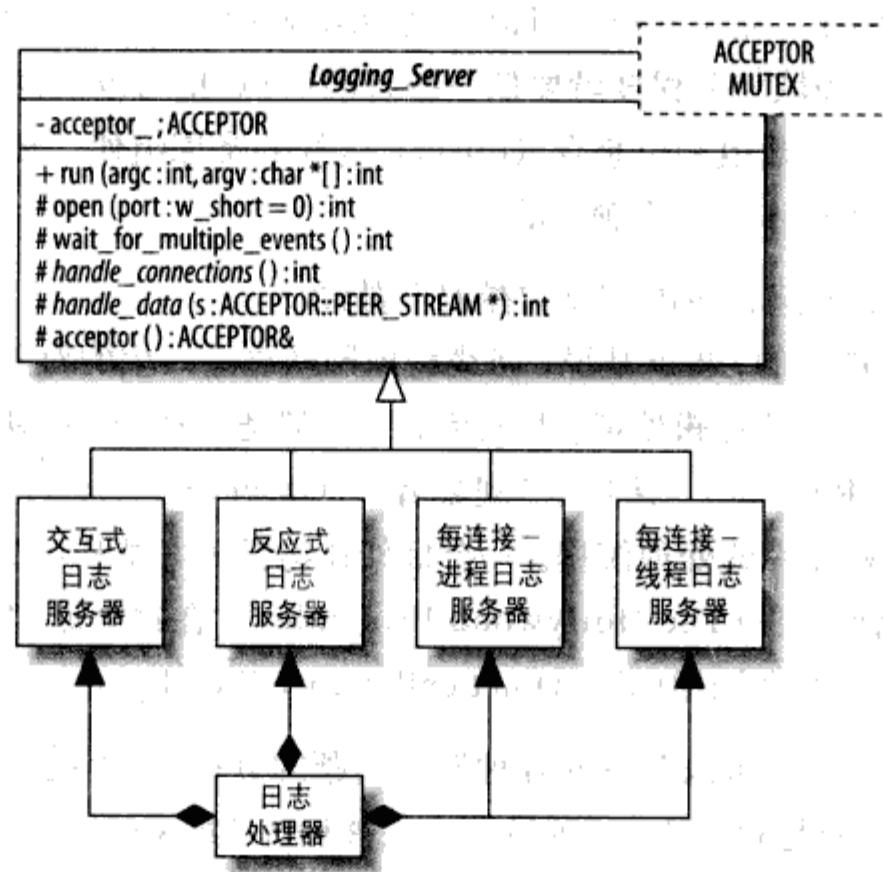


图 26-2：日志服务器框架的面向对象设计

本章的后续部分将按如下方式组织：下一节描述日志服务器框架的面向对象设计，研究它的架构和影响OO框架设计的重要因素，从而阐明我们为什么选择了某些模式和语言特性，同时也总结了我们因为各种原因而放弃了的其他选择。另外还有两节给出了几个日志服务器框架的C++串行(sequential)编程实例和并发(concurrent)编程实例。最后以总结本章中的OO软件概念和技术之美作为结尾。

译注1：这里所说的“模板”，其含义跟C++，Java或C#中的“template”含义不同。

^{注6：}参阅《Software Product Lines: Practices and Patterns》，Paul Clements, Linda Northrop著，Addison-Wesley，2001版。

日志服务器框架的面向对象设计

在我们讨论日志服务器的面向对象设计之前，先理解几个有关OO框架的关键概念是至关重要的。大多数程序员都熟悉“类库”的概念，类库就是一套可复用的类的集合，这些类提供了程序员在开发面向对象程序时可能会用到的功能。而面向对象的框架在以下方面进一步扩充了面向对象类库的优势（注7）：

OO框架定义“半完整”的应用，这种应用能体现领域相关的对象结构及功能。

在一个特定的领域中，比如图形用户界面，航空任务计算（avionics mission computing），或网络化日志服务领域，框架中的类协同工作，为应用程序提供了一个范化的轮廓架构。完整的应用可以通过继承或实例化框架组件的方式来编写。相比之下，类库的领域相关程度就要低一些，而且可重用的范围也小一些。举例来说，像字符串、复数、数组和位集合（bitset）这样的类库组件，相对来讲就更底层一些，而且是通用于多种应用领域的。

框架是主动的，且具有运行时的“反向控制”特性

类库的特点是被动的，也就是说，当线程控制从自己产生的对象上调用它们时，才会去执行一段孤立的逻辑处理。相对而言，框架是主动的，也就是说，它们通过诸如“反应器”（Reactor）（注8）、“观察者”（Observer）（注9）这样的事件分发模式，自己控制程序中的处理流程。一个框架在运行时其结构中的“反向控制”常被称为“好莱坞原则”（The Hollywood Principle），在这个原则中，人们常说“不要调用我们，我们会调用你”（Don't call us, we'll call you.）（注10）。

框架的设计通常基于框架可以解决的各种潜在问题的分析，从中找出每种方案的哪些部分是相通的，哪些部分是独特的。这种设计方法被称为“共性/变性分析”（commonality/variability analysis）（注11），它包含以下主题：

注7：《Frameworks = Patterns + Components》，Ralph Johnson，ACM通讯，第40卷第10期，1997年10月。

注8：参阅先前引用的Schmidt等人的著作。

注9：参阅先前引用的Gamma等人的著作。

注10：《Pattern Hatching - Protection, Part I: The Hollywood Principle》，John Vlissides，C++ Report，1996年2月。

注11：《软件工程中的共性和变性》（Commonality and Variability in Software Engineering），J. Coplien, D. Hoffman, and D. Weiss, IEEE Software, 第15卷第6期，1998年11~12月。

范围 (Scope)

定义领域（即一个框架所能解决的问题的范围）和框架上下文。

共性 (Commonalities)

描述基于框架的所有产品中，哪些属性是重复出现的。

变性 (Variabilities)

描述不同产品的独特属性。

理解共性

因此，设计日志服务器框架的第一步就是理解系统中哪些部分是应该由框架来实现的（共性），哪些部分是留给子类或参数进行特化的（变性）。这个分析非常简单直接，因为通过网格来处理一个日志记录的过程可被分解成如图26-3所示的几个步骤，这对所有的日志服务器实现都是一样的。

在设计过程的这个阶段，我们把每一步都定义得尽量抽象。比如，在这个阶段我们对所采用IPC机制的类型做了很少的假设，而不是假定它是面向连接的，以确保日志记录能被可靠地递交。同样，我们也避免指定每一步所采用并发策略的类型（即，服务器是否能处理多重请求，如果能，这些请求又是怎么被分发的）或同步机制的类型。于是，对某个步骤的特定行为的选择就被延迟到以后的具体实现中，具体实现为每一步都提供一个特定的选择。

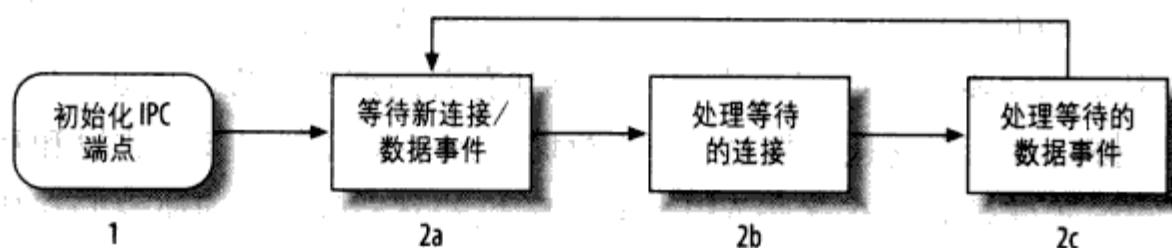


图 26-3：日志服务器的主循环

“模板方法”模式是定义抽象步骤并延迟其具体实现的有效方法，每一步特定行为的具体实现都被延迟到设计过程的后续阶段。此模式定义一个基类，基类在模板方法中实现公共的抽象步骤，这些步骤都是钩子方法 (hook method)，子类可以有选择地用具体实现覆盖 (override) 它们。某些程序设计语言的特性，比如 C++ 中的纯虚函数和 Java 中的抽象方法，可用于确保每一个具体实现都会定义这些钩子方法。图 26-4给出了“模板方法”模式的结构，并示范了如何应用这种模式来设计我们的面向对象日志服务器框架。

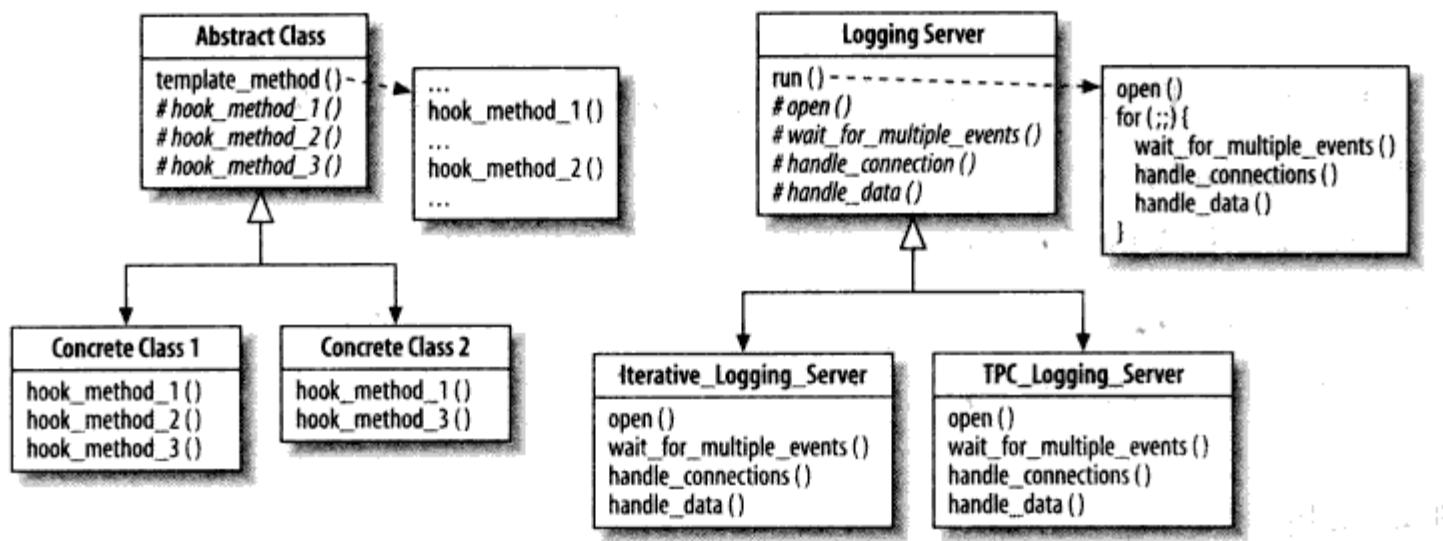


图 26-4：“模板方法”模式和它在日志服务器中的应用

适应变化

尽管模板方法模式解决了日志服务器框架的全局设计，但是我们仍然需要解决先前定义的三个维度的可变性的问题，因为我们的设计要靠这些可变性来支持。一种方案是简单地运用模板方法模式，针对每一种 IPC/ 并行 / 同步方式的不同组合实现一个具体子类。不幸的是，这种方案会导致子类数目的指数级增长，因为在任一维上每增加一种可能性就等于针对其他维度的每一种的组合都要增加一个新的实现。因此，这样一个纯粹的模板方法设计，将不会比一次性地针对每一种可变性手工实现一个日志服务器更好。

一种更有效、伸缩性也更好的设计是利用不同维度上的可变性相互独立的事实。举例来说，选择一种不同的 IPC 机制，将不会导致所用的并发和同步机制也随之变更。然而，关于不同的 IPC 和同步机制如何起作用，其中却有着高度的共性。例如，IPC 机制可以发起 / 接收连接，可以在连接上发送 / 接收数据，而并发机制则拥有获取和释放锁的操作。设计上的难题就是如何在这些 API 中封装随机的复杂性，从而使它们在使用上可以相互替换。

解决难题的一种方法是使用“包装外观”（Wrapper Facade）模式，这种模式封装底层 OS 的所提供的非面向对象的 IPC 和同步机制，向上提供单个统一的面向对象接口。对于隐藏不同机制之间的随机复杂性从而提高可移植性，或者使这些 API 不再那么难用和易出错，包装外观模式特别有用。比如，一个包装的外观可以定义一套上层的类型系统，用于确保在底层 OS 所提供的用于 IPC 或同步的数据结构上，只有正确的操作会被调用，这些底层的数据结构是并不是面向对象的（类型安全性也差一些）。

ACE 是一个被广泛使用的采用包装外观模式为 IPC 和同步机制定义统一 OO 接口的例子。本章的包装外观将基于 ACE 实现的一个简化版本。图 26-6 给出了 ACE 中的一些包装外观。

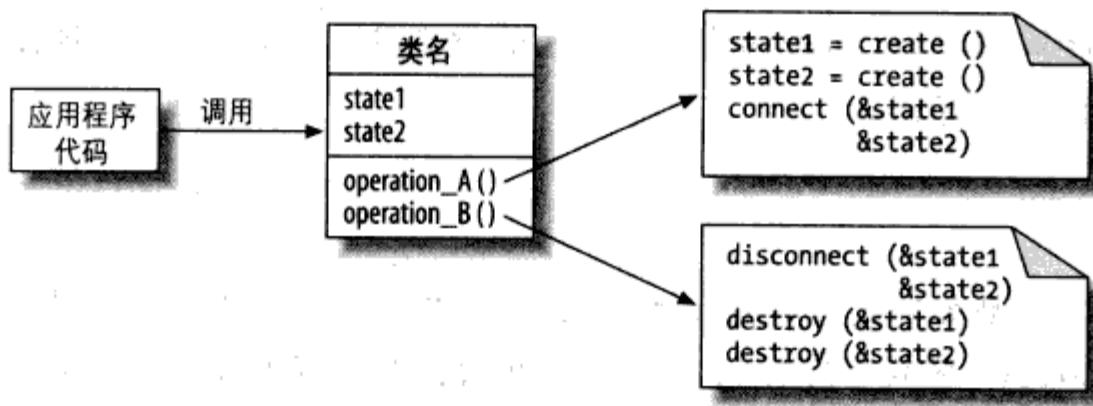


图 26-5：“包装外观”(Wrapper Facade) 设计模式

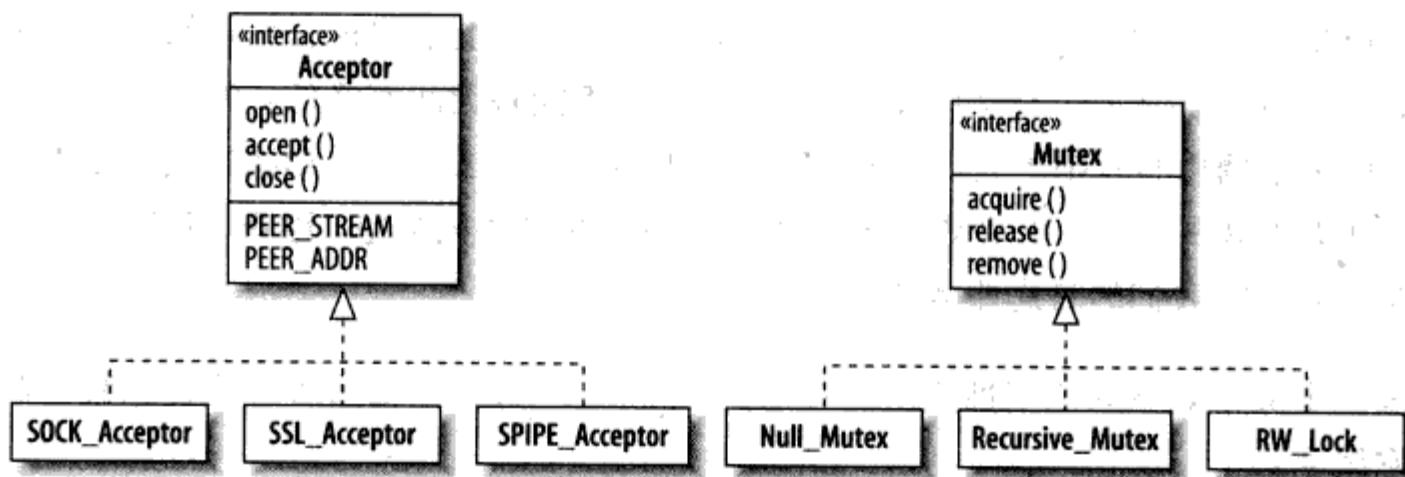


图 26-6：ACE 中的一些关于被动连接的建立及同步的“包装外观”

包装外观类 `Acceptor` 提供了创建连接的方法，这里创建的是被动模式的连接，它也提供了一些“特征”(traits) 用来表示一种机制的各个方面，所表示的机制在各种不同的实现中本质上都以同样的方式工作，只不过是基于不同的 API 实现。比如，`PEER_STREAM` 和 `PEER_ADDR` 代表了不同的包装外观，分别用于数据的发送接收和 IPC 机制的寻址。`SOCK_Acceptor` 是 `Acceptor` 类的一个子类，本章将用它来实现被动地建立连接的工厂，其所建的连接是基于 socket API 实现的。

包装外观 `Mutex` 提供了一个接口，这个接口的方法用于取得和释放锁，子类包括一个用互斥体实现的 `Recursive_Mutex`，它在被同一线程多次加锁时不会死锁；一个 `RW_Lock`，它实现了读 / 写语义；还有一个 `Null_Mutex`，它的 `acquire()` 和 `release()` 方法是内联的 no-op。最后这个类是一个“空对象”(Null Object) 模式的例子(注 12)，它可用于在不修改程序代码的前提下消除同步。从图 26-6 上看，似乎每一套相关的类都是通过继承联系在一起的，但实际上它们是通过相互没有继承关系的类实现的，

注 12：“空对象模式”(The Null Object Pattern)，Bobby Woolf，《程序设计的模式语言，第 3 卷》(Pattern Languages of Program Design, Vol.3)，Robert C. Martin, Dirk Riehle, Frank Buschmann 著，Addison-Wesley, 1997。

它们拥有的是相同的接口，可被用作C++模板的类型参数。我们选择这种设计方案是为了避免虚函数调用的开销。

全部代码

另外一个设计难题是如何把一种并发策略跟一种IPC和同步机制关联起来。一种方法是使用策略（Strategy）模式（注13），这种模式把算法封装成对象，于是可在运行时交换不同的算法。采用这种方法，Logging_Server类将持有指向抽象基类Acceptor和Mutex的指针，然后它就可以靠动态绑定和多态来将虚函数分派到相应的子类实例。

基于策略的方法虽然可行，却不是理想的方法。每一个送上的日志记录都会导致多次调用包装外观类Acceptor和Mutex中的方法。性能将因此而降低，因为相对于非虚函数，虚函数调用的开销更大。考虑到对我们的日志服务器来说，动态切换IPC和同步机制并不是必需的，一个更高效的方案是使用C++的参数化类型，用IPC和同步的包装外观类来实例化我们的日志服务器。

我们因此定义了如下的范型抽象基类Logging_Server，本章用所有的日志服务器类都从它派生：

```
template <typename ACCEPTOR, typename MUTEX>
class Logging_Server {
public:
    typedef Log_Handler<typename ACCEPTOR::PEER_STREAM> HANDLER;
    Logging_Server (int argc, const char *argv);

    // 模板方法依次执行主事件循环中的各个步骤。
    virtual void run (void);

protected:
    // 钩子方法，每个步骤都可能被改写。
    virtual void open (void);
    virtual void wait_for_multiple_events (void) = 0;
    virtual void handle_connections (void) = 0;
    virtual void handle_data
        (typename ACCEPTOR::PEER_STREAM *stream = 0) = 0;

    // 增加请求计数，由互斥体保证访问同步。
    virtual void count_request (size_t number = 1);

    // 模板参数ACCEPTOR类的实例，用于接收连接。
    ACCEPTOR acceptor_;

    // 对接收到的日志记录进行计数。
    size_t request_count_;

    // 模板参数MUTEX类的实例，用于串行化对request_count_访问。
};
```

注13：参阅先前引用的Gamma等人的著作。

```

    MUTEX mutex_;
```

// 服务器监听连接的地址.

```

    std::string server_address_;
```

}

Logging_Server类的大多数方法都是纯虚的，这保证了子类必需实现它们。但是，下面的open()和count_request()方法却是被本章中所有的日志服务器类直接复用的：

```

template <typename ACCEPTOR, typename MUTEX>
Logging_Server<ACCEPTOR, MUTEX>::Logging_Server
(int argc, char *argv[]): request_count_(0) {
    // 解析命令行参数，保存服务器地址server_address_...
}

template <typename ACCEPTOR, typename MUTEX> void
Logging_Server<ACCEPTOR, MUTEX>::open (void) {
    return acceptor_.open (server_address_);
}

template <typename ACCEPTOR, typename MUTEX> void
Logging_Server<ACCEPTOR, MUTEX>::count_request (size_t number) {
    mutex_.acquire (); request_count_ += number; mutex_.release ();
}
```

Log_Handler类负责从一个连接上来的数据流中反序列化(demarshaling)出一条日志记录，数据流的IPC机制由类型参数ACCEPTOR指定。这个类的实现超出了本章的范围，而且它本身也可以成为可变性的另外一个维度——即日志服务器可能需要支持不同的日志消息格式。如果我们要支持用不同的格式存储上行日志消息，那么这个类会成为日志框架的又一个模板参数。对我们的目标而言，知道它是被IPC机制参数化而且提供了两个方法：peer()和log_record()就足够了，peer方法返回一个数据流的引用，log_record从流中读入单条日志记录。

Logging_Server类的主入口是模板方法run()，它实现了图26-3中描述的步骤，并将具体的步骤委派给Logging_Server类的protected部分声明的钩子方法，代码如下：

```

template <typename ACCEPTOR, typename MUTEX> void
Logging_Server<ACCEPTOR, MUTEX>::run (void) {
    try {
        // 第1步，初始化一个IPC工厂端点，在服务器地址上监听新连接.
        open ();

        // 第2步，进入一个事件循环;
        for (;;) {
            // 2a: 等待新连接或日志记录的到来.
            wait_for_multiple_events ();

            // 2b: 接受新连接（如果有的话）
            handle_connections ();
    }
}
```

```

    // 2c: 处理收到的日志记录（如果有的话）
    handle_data ();
}
} catch (...) { /* ... 处理异常 ... */ }
}

```

以上代码的美妙之处在于：

- 基于模式的设计使得处理并发模型中的可变性变得容易，比如，可以通过在子类中提供特定的钩子方法实现来改变模板方法 run() 的行为。
- 基于模板的设计使得处理 IPC 和同步机制中的可变性变得容易。比如，我们可以将不同的类型传递给 ACCEPTOR 和 MUTEX 模板参数。

实现串行化日志服务器

本节演示了一个以串行 (sequential) 并发模式为特征的日志服务器的实现，也就是说，本节中的日志服务器的所有处理都在单个线程中完成。我们将讨论迭代式和反应式两种串行日志服务器的实现。

迭代式日志服务器

迭代式服务器在处理完来自一个客户端的所有日志记录之后，才会去处理来自下一个客户端的日志记录。既然没有必要产生和同步多个线程，那么我们就使用外观类 Null_Mutex 来参数化 Iterative_Logging_Server 子类，代码如下：

```

template <typename ACCEPTOR>
class Iterative_Logging_Server :
    virtual Logging_Server<ACCEPTOR, Null_Mutex> {
public:
    typedef Logging_Server<ACCEPTOR, Null_Mutex>::HANDLER HANDLER;
    Iterative_Logging_Server (int argc, char *argv[]);
protected:
    virtual void open (void);
    virtual void wait_for_multiple_events (void) {};
    virtual void handle_connections (void);
    virtual void handle_data
        (typename ACCEPTOR::PEER_STREAM *stream = 0);
    HANDLER log_handler_;
    // 所有客户端共享的日志文件。
    std::ofstream logfile_;
};

```

实现这个版本的服务器非常简单直接。`open()`方法装饰（decorate）了`Logging_Server`基类中`open`方法的行为：在调用基类的同名方法之前先打开一个输出文件。代码如下：

```
template <typename ACCEPTOR> void
Iterative_Logging_Server<ACCEPTOR>::open (void) {
    logfile_.open (filename_.c_str ());
    if (!logfile_.good ()) throw std::runtime_error;
    // 委派给父类的open()方法。
    Logging_Server<ACCEPTOR, Null_Mutex>::open ();
}
```

代码中的`wait_for_multiple_events()`方法没有任何操作（no-op）。我们并不需要它，因为在任一时刻我们只处理一个连接。`handle_connections`方法因此也是简单的阻塞，直到建立起一个新的连接，代码如下所示：

```
template <typename ACCEPTOR> void
Iterative_Logging_Server<ACCEPTOR>::handle_connections (void)
{ acceptor_.accept (log_handler_.peer ()); }
```

最后，`handle_data()`将从客户端读取日志记录，并将其写入到日志文件，直到客户端关闭连接或者发生错误。

```
template <typename ACCEPTOR> void
Iterative_Logging_Server<ACCEPTOR>::handle_data (void) {
    while (log_handler_.log_record (logfile_))
        count_request ();
}
```

尽管这个迭代式服务器实现起来非常直接，但是它却有一个缺点：在某一时刻它只能服务于一个客户端。在等待已经连接上来的客户端结束请求时，另外一个试图连接的客户端可能超时。

反应式日志服务器

反应式日志服务器减轻了前一节中迭代式服务器的一个重要缺陷，其实现方法是使用操作系统提供的同步事件多路分离（synchronous event demultiplexing）API来处理多个客户端连接和日志记录请求，比如使用`select()`和`WaitForMultipleObjects()`。这些API可以同时监听多个客户端，方法是在单个执行线程中同时等待一组I/O句柄上的I/O相关事件，然后穿插进处理日志记录的逻辑。然而，由于一个反应式的日志服务器从根本上讲仍然是串行的，因此它继承了先前实现的迭代式日志服务器，如图26-7所示。

`Reactive_Logging_Server`类重写了它从`Iterative_Logging_Server`基类继承来的所有4个钩子方法。它的`open()`钩子方法进一步装饰了基类的同名方法来初始化`ACE_Handle_Set`成员变量，这是为简化`select()`函数的使用而设计的封装包装外观（wrapper facade）类的一部分，代码所下所示：

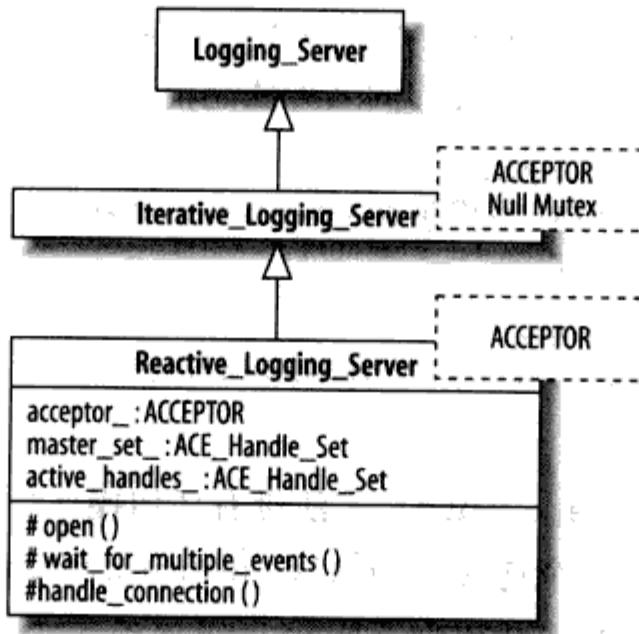


图 26-7：反应式日志服务器接口

```

template <typename ACCEPTOR> void
Reactive_Logging_Server<ACCEPTOR>::open () {
    // 委派给基类.
    Iterative_Logging_Server<ACCEPTOR>::open ();

    // 将与接收器相关联的句柄状态标记为“活动”(active).
    master_set_.set_bit (acceptor_.get_handle ());

    // 将接收器句柄设为非阻塞模式.
    acceptor_.enable (NONBLOCK);
}

```

跟 Iterative_Server 类中的相应方法不同，在这个实现中，`wait_for_multiple_events` 方法是必要的。如图 26-8 所示，这个方法使用一个同步事件多路分离器（在这个例子中，是 `select()` 调用）来检测哪些 I/O 句柄上有连接或者未处理的数据活动。

在 `wait_for_multiple_events()` 被执行之后，`Reactive_Logging_Server` 就拥有了一个缓冲起来的句柄集合，这些句柄上存在未处理的活动（即要么有新的连接请求，要么有新的数据接收事件），这些句柄被该类的另外两个钩子方法 `handle_data()` 和 `handle_connections` 所处理。`Handle_connections` 方法检查接收器句柄是否是活动的，如果是，就尽可能多地接收连接并把它们缓冲在 `master_handle_set_` 变量中。类似地，`handle_data()` 方法在其余的被先前的 `select()` 标记出的活动句柄上迭代。这一活动的处理被 ACE 的 socket 包装外观类所简化了，ACE 的包装外观类为 socket 句柄集合实现了迭代器模式（注 14）的一个实例。如图 26-9 所示。

注 14：参阅先前引用的 Gamma 等人的著作。

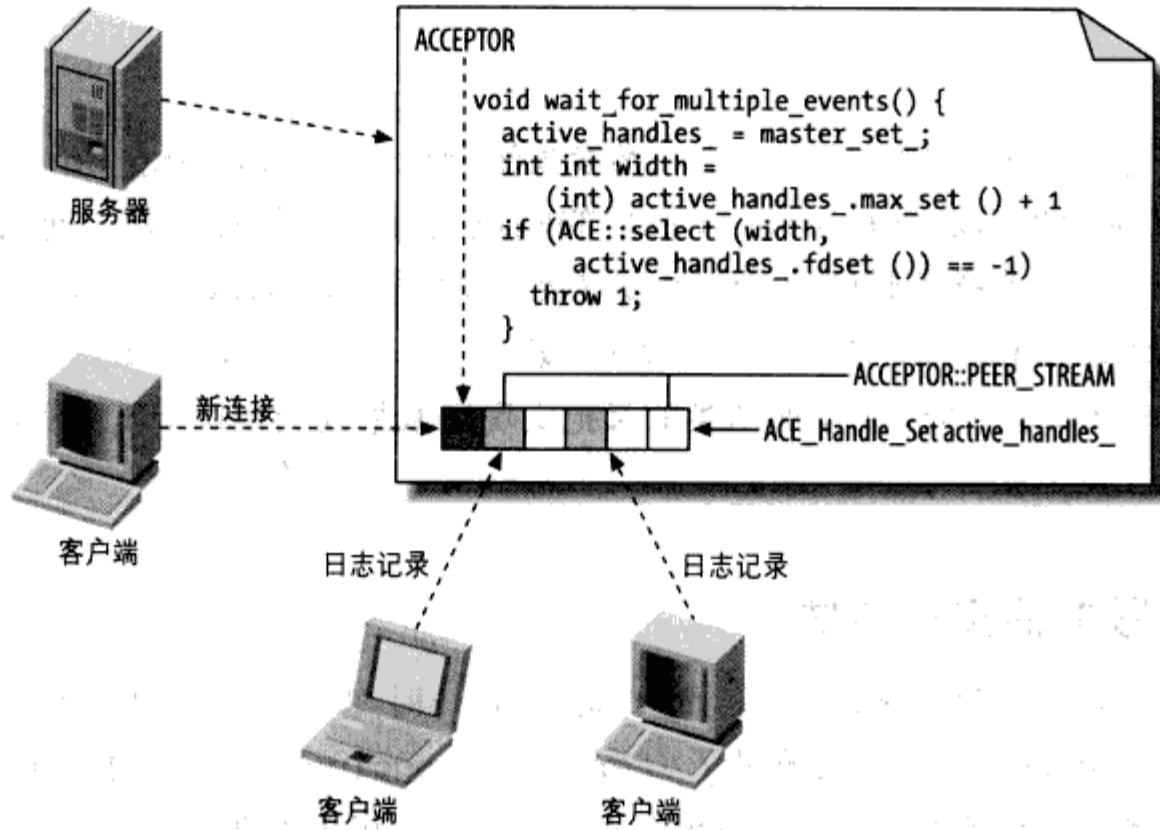


图 26-8：在 Reactive_Logging_Server 程序中使用异步事件多路分离器（译注 2）

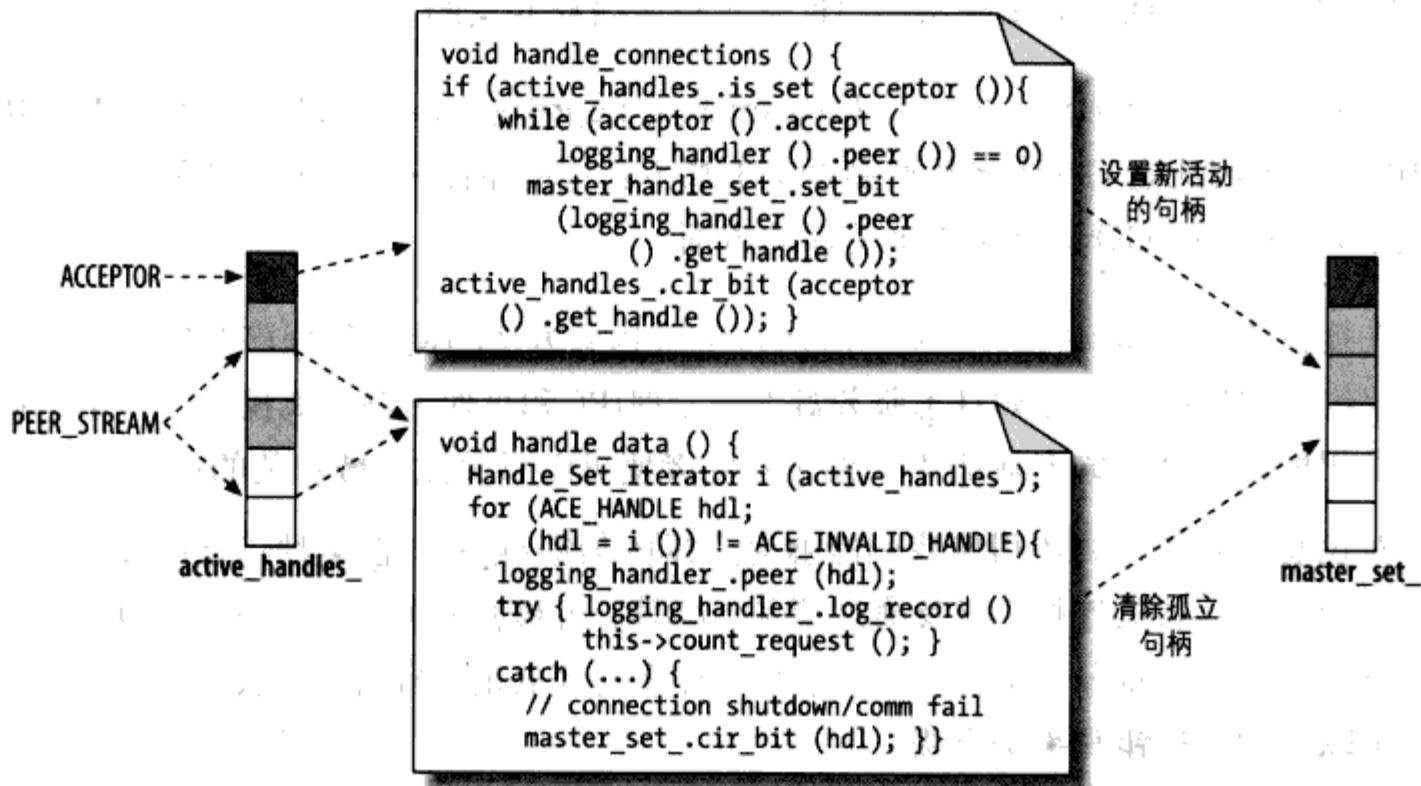


图 26-9：反应式连接 / 数据事件处理

以下代码实现了基于 socket API 的 Reactive_Logging_Server 主程序：

```

int main (int argc, char *argv[]){
    Reactive_Logging_Server<SOCK_Acceptor> server (argc, argv);
    server.run ();
}

```

译注 2：原文错误，这里显然应为“同步事件多路分离器”。

```
    return 0;  
}
```

主函数的第一行使用 `SOCK_Acceptor` 类参数化 `Reactive_Logging_Server`, C++ 编译器会因之而产生出能进行 `socket` 通信的响应式日志服务器代码。而由于我们从 `Logging_Server` 类继承时硬编码的模板参数, 紧接着 `Logging_Server` 基类又会被 `SOCK_Acceptor` 类和 `Null_Mutex` 类参数化。第二行调用模板方法 `run()`, 它将实现委派给基类 `Logging_Server`, 而基类的又把实现委派给我们在这个类中实现的各个钩子方法。

串行日志服务器实现评估

通过穿插着为多个客户端服务, 而不是在某一时刻全身心地处理一个客户端 `Reactive_Logging_Server` 类改进了 `Iterative_Logging_Server` 的缺点。然而, 它却没有利用操作系统的并行机制, 因此它不能有效地利用多处理器。它也不能在处理日志记录的同时读取新记录, 从而将计算和通信重叠进行。这些局限妨碍了它在面对客户数增加时的可伸缩性, 即使底层的硬件是支持多线程并发的。

尽管 `Iterative_Logging_Server` 和 `Reactive_Logging_Server` 都是运行在单个执行线程中——也因此它们在大多数产品系统中都不具可伸缩性——但它们的简单性却突显出我们基于框架的 OO 设计中许多美丽的方面:

- 在模板方法 `Logging_Server::run()` 中对钩子方法的使用使程序开发者不必受底层细节的羁绊——比如日志服务器怎样处理 IPC 和事件多路分离操作——于是通过借用框架设计者的专业能力, 程序开发者便可以只关注领域相关的应用逻辑。
- 对包装外观模式的使用使我们能够对互斥体加锁/解锁, 在某种特定的 IPC 机制上监听以接收新的连接, 以及通过简易、高效、可移植的方式等待多个 I/O 事件。没有这些有用的抽象, 我们将不得不编写许多冗长的、易出错的代码, 这样的代码难以理解、排错和维护升级。

在下一节讨论的更复杂的并行日志服务器中, 从这些抽象中获得的好处将会更加显著, 在更加复杂的框架用例, 比如在图形用户界面 (注 15) 或通信中间件 (注 16) 中, 也是如此。

注 15: 参阅先前引用的 Gamma 等人的著作。

注 16: 参阅先前引用的 Schmidt 等人的著作。

实现并行日志服务器

为了克服上一节中介绍的迭代式和反应式服务器在可伸缩性方面的局限，本节的日志服务器使用了操作系统的并发机制：进程和线程。然而，由于设计上的随机复杂性，使用操作系统提供的API来产生进程和线程会是一件让人气馁的工作。这些复杂性来自不同操作系统，甚至同一操作系统的不同版本之间，在API的设计上存在的语义和语法上的差别。我们对这类复杂性的解决办法仍然是应用“包装外观”（wrapper facade）模式提供跨平台的统一接口，并把这些包装外观集成到面向对象的 Logging_Server 框架中。

“每连接一线程”的日志服务器

我们的“每连接一线程”（thread-per-connection）的日志服务器（TPC_Logging_Server）运行一个主线程来等待并接收来自客户端的连接。每当接收到一个新连接之后，就创建一个新的工作线程来处理来自那个连接的日志记录。图 26-10 显示了这一过程的各个阶段。

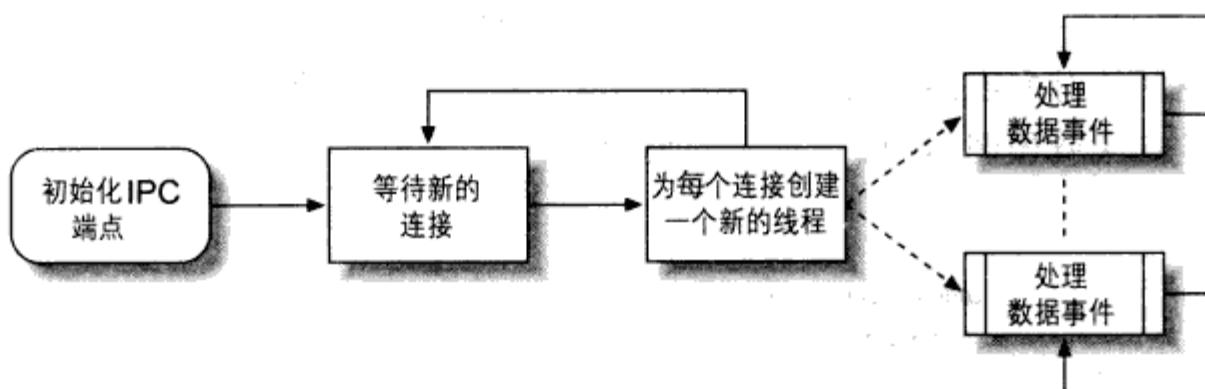


图 26-10：“每连接一线程” 日志服务器的处理过程

这个日志服务器的主循环流程跟图 26-3 所描述的处理过程不同：对 `handle_data()` 的调用不是必需的，因为工作线程会负责那个调用。有两种方法可以处理这种情况：

- 我们注意到基类的 `run()` 方法通过默认的空指针参数调用 `handle_data()` 方法，于是可以让我们的实现在见到这种输入时直接退出。
- 我们可以直接用自己的实现覆写 `run()` 方法，从而忽略那个调用。

乍一看第二种方案更具优势，因为它避免了一个对 `handle_data()` 的虚函数调用。然而，在这种情况下，第一种方案更好，因为虚函数调用的性能问题并不是一个限制性的因素，而重写模板方法 `run()` 将使得这个类不能享受针对基类实现的改进，从而有可能导致难以捕捉的甚至严重的程序错误。

这里最主要的难题是实现并行策略本身。正如前面“迭代式日志服务器”一节中的 Iterative_Server 类, wait_for_multiple_events() 方法是多余的, 因为我们的主循环等待的只是新的连接, 因此让 handle_connections() 方法在 accept() 方法上阻塞, 接着创建工作线程来处理连接上来的客户端就足够了。我们的 TPC_Logging_Server 类因此必需提供一个方法作为线程的入口。在 C 和 C++ 中, 一个类方法只有被定义为静态 (static) 的才可以作线程的入口; 因此我们把 TPC_Logging_Server::svc() 定义为类的静态方法。

现在, 我们有一个重要的设计决定要做: 线程的入口函数到底需要做什么? 人们很容易会考虑让 svc() 方法单独处理所有那些从连接上接收日志所需的工作。但这种设计并不理想, 因为静态方法不可以是虚的, 如果我们以后需要从这个实现派生一个新的日志服务器类来改变它处理数据事件的方式, 那么就可能导致问题。因为那样一来, 应用程序开发者就不得不提供一个在实现流程上跟这个类的完全相同 handle_connections() 方法, 而这只为了调用到正确的 static 方法。

此外, 为了充分利用我们已有的设计和代码, 最好还是让日志记录的处理逻辑隐藏于 handle_data() 方法的内部, 并定义一个辅助对象 Thread_Args, 用于保存从 accept 返回的对方连接以及指向 Logging_Server 对象自身的指针。这样, 我们的类接口看起来就像图 26-11 所展示的那样。

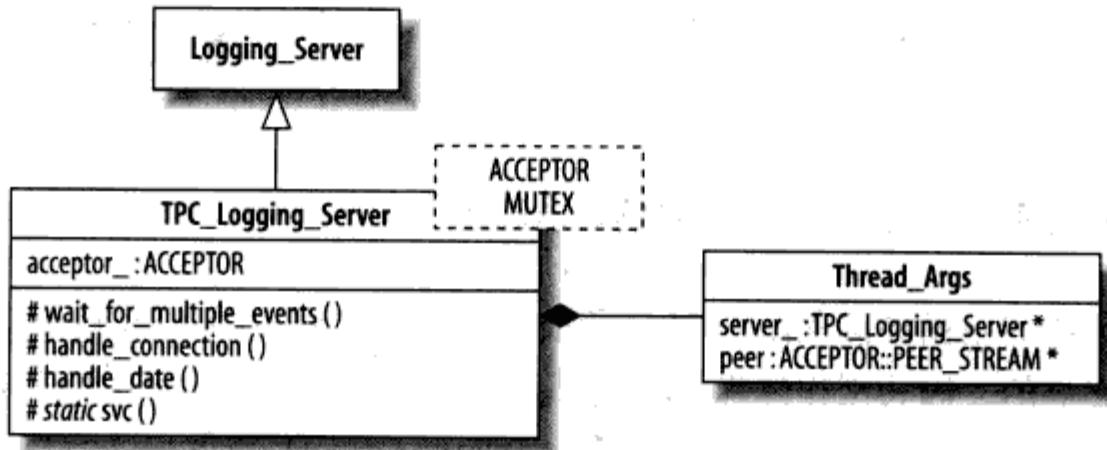


图 26-11: “每连接一线程”的日志服务器接口

TPC_Logging_Server 类的其余实现就很容易了, 只需要让线程入口通过 server_pointer 把处理过程委派给虚方法 handle_data(), server_pointer 指针保存在传给 svc() 方法的辅助对象 Thread_Args 中, 如图 26-12 所示。

以下代码实现了一个使用 TPC_Logging_Server 的主程序, 其中使用了安全 socket API 和读 / 写锁。

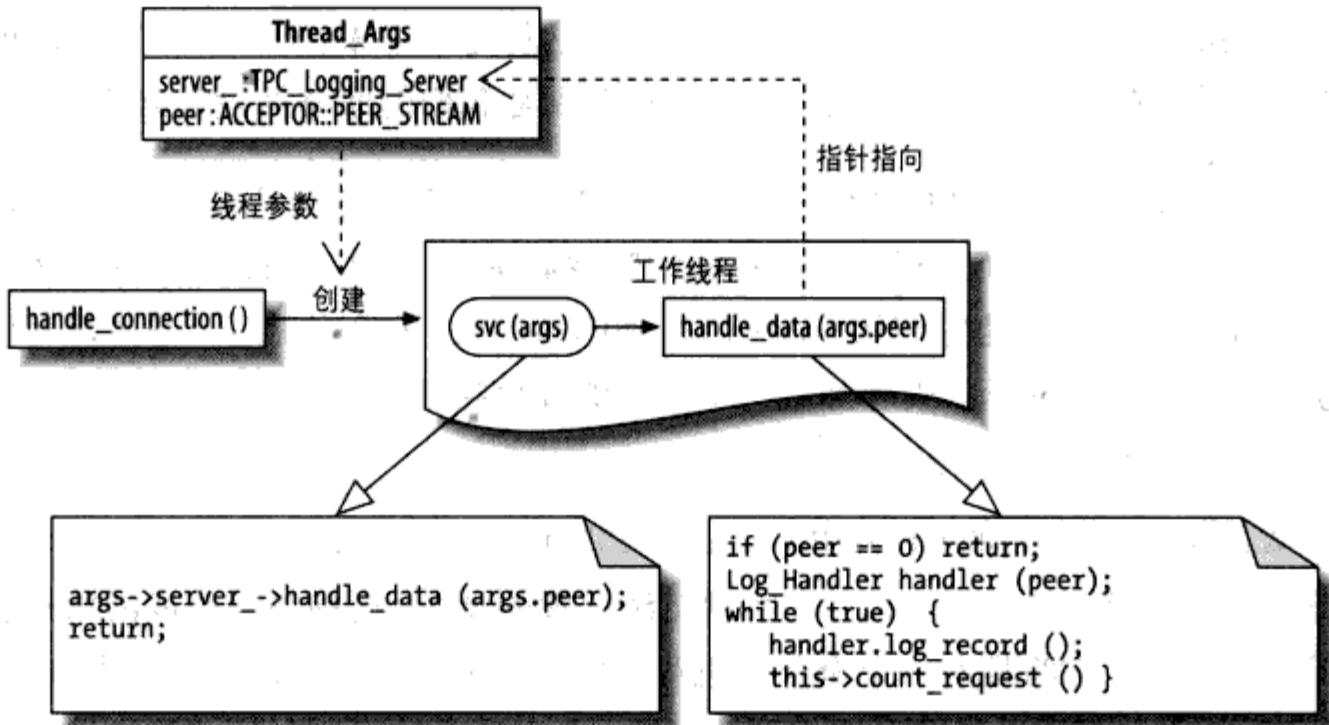


图 26-12：“每连接一线程”的线程行为

```

int main (int argc, char *argv[]) {
    TPC_Logging_Server<SSL_Acceptor, RW_Lock> server (argc, argv);
    server.run ();
    return 0;
}

```

`main()` 函数实例化了一个 `TPC_Logging_Server`, 它使用 SSL 连接进行通信, 使用 `RW_Lock` 同步 `Logging_Server` 基类中的 `count_connections()` 函数。除了实例化的类的名字不同之外, 这个 `main()` 函数跟本章前面为 `Reactive_Logging_Server` 类写的那个完全相同。这种共性从另外一个方面体现了我们设计的漂亮: 不论我们采用并发、IPC 和同步机制的何种组合, 对服务器的实例化和调用都保持不变。

“每连接一线程”的日志服务克服了前面“串行日志服务器实现评估”一节中描述的串行实现方式在可伸缩性方面的局限。我们的 OO 框架设计使得这种“并发模型”跟已有代码的集成也非常简单直接, 仅需少量的修改。特别地, `TPC_Logging_Server` 继承了 `open()`、`count_request()` 以及更重要的 `run()` 方法的实现, 使这个类可以透明将针对主事件循环的 bug 修正及其他改进集成进来。除此之外, 若要对 `request_count_` 的访问增加必要的同步机制, 只需用 `RW_Lock` 类参数化 `TPC_Logging_Server`。

“每连接一进程”的日志服务器

下面将要讲到的“每连接一进程”的日志服务器跟图 26-10 所示的“每连接一线程”的设计基本相似, 不同之处在于我们每次产生一个新进程来处理来自客户端的上行日志记录, 而不是用线程。不同的平台之间进程创建的语义不尽相同, 选择用进程而不是线程

处理并发，我们就必需把注意力转移到如何适应这种差异上来。Linux 和 Windows 平台上的进程 API 存在两种主要的语义差别，这种差别是我们的服务器设计必需要封装的：

- 在 Linux (以及其他 POSIX 系统) 中，创建进程的主要方式是通过系统函数 `fork()`，它产生一份跟调用程序完全相同的映像副本，包括打开的 I/O 句柄也相同。两个进程惟一不同的是它们从 `fork()` 得到的返回值。在创建点上，子进程可以选择从哪个点上继续执行，或者加载一个不同的程序映像，方法是通过调用 `exec` 打头的那一套系统调用中的某一个。
- 而 Windows 上使用的 CreateProcess API 调用，它在功能上等同于一个 POSIX 的 `fork()`，后跟一个对 `exec*` () 系列系统函数的调用。这种差异造成的结果就是在 Windows 中，你将拥有一个全新的进程，默认情况下它不能访问父进程打开的 I/O 句柄。因此，要使用父进程接收到的连接，必需显式地复制句柄并将它通过命令行传递给子进程。

因此，我们需要一套包装外观 (wrapper facade) 类，这些类不仅要封装不同平台的语法差异，还要提供一种隐藏语义差别的方法。这套包装由三个相互协作的类组成，如图 26-13 所示。`Process` 类代表单个进程，用于进程的创建和同步。`Process_Options` 类提供了一种设置进程选项方法，即可用于设置平台无关的选项（比如命令行参数选项和环境变量），也可用于设置进程相关的选项（比如避免僵尸进程）。最后，`Process_Manager` 类以跨平台的方式管理一组进程的生命周期。本章中我们不会面面俱到地讨论这些包装外观类的所有使用情况，尽管他们是基于 ACE 中的版本（注 17）。我们只需要知道基于我们的类，不仅进程的创建在 Linux 和 Windows 之间是可移植的，而且 I/O 句柄也能以可移植的方式自动地复制给新的进程。

于是设计的难处就在于应付以下事实：当接收到新的连接之后，创建出来的进程将从程序的起始处开始运行。我们当然不希望子进程尝试自己打开一个新的接收器并监听连接，相反，它们应该从被分配的句柄上监听数据事件。要解决这个问题，一种天真的想法是让程序自己做条件检测，并根据检测的结果相应地调用一个特殊入口，这个入口定义在我们的基于进程的 `Logging_Server` 类的接口中。

注 17：《C++ 网络编程 卷 1：运用 ACE 和模式消除复杂性》(C++ Network Programming, Vol. 1: Mastering Complexity with ACE and Patterns), Douglas C. Schmidt, Stephen D. Huston 著, Addison-Wesley, 2001。

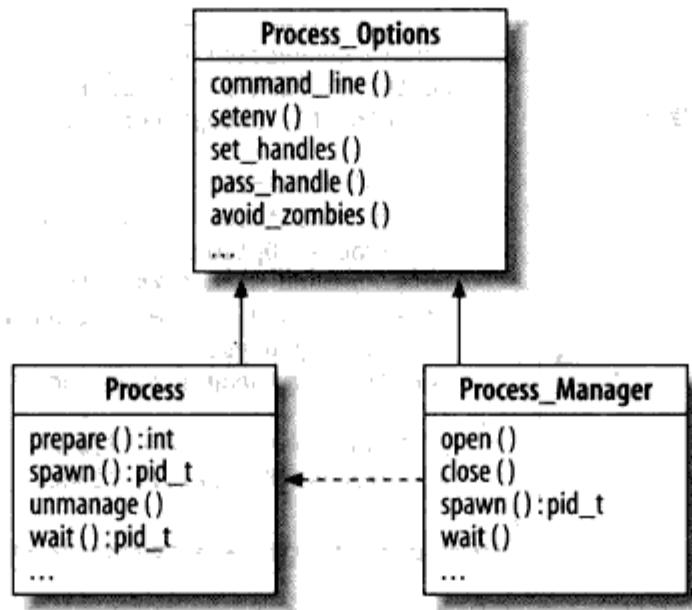


图 26-13：可移植的进程包装外观类

然而，这个简单的方案并不理想。它不仅需要我们改变这个基于进程的Logging_Server类的公共接口，还需要对程序暴露私有的实现细节，这是违反封装原则的。一个更好的方案是重写（override）从基类Logging_Server继承的模板方法run()，这个类被传递了一份来自用户的命令行参数的拷贝，重写的run方法于是可以通过检查这份拷贝来确定是否有I/O句柄传给它。如果没有，进程就认为自己是一个父进程并把调用委派给基类的run()方法。否则，进程就认为自己是子进程，因此它解析出句柄并调用handle_data()，如图26-14所示。

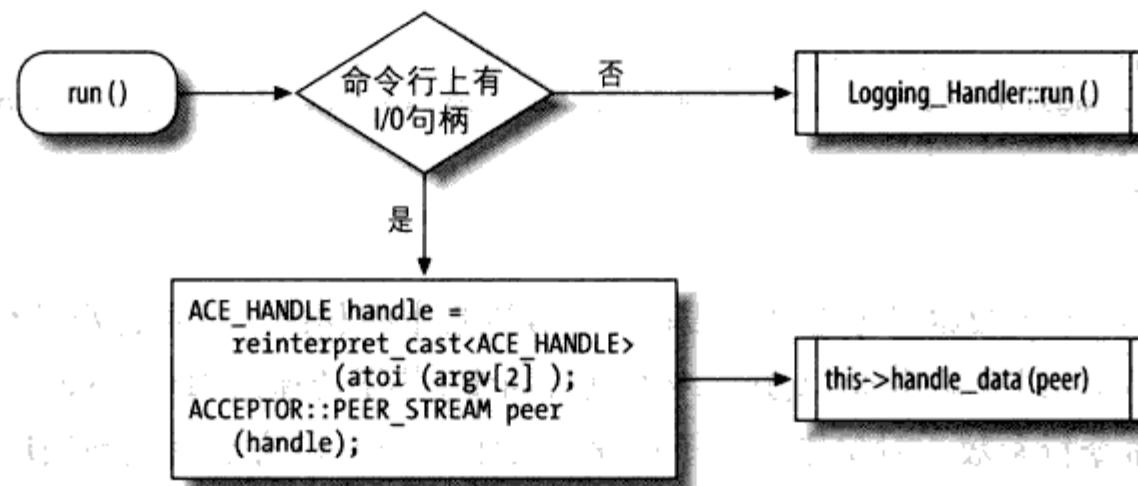


图 26-14：“每连接一进程”的模板方法 run()

这个服务器类的其余部分就很容易实现了。如图26-15所示，进程包装外观类使得工作进程的创建相当容易。Handle_data()的实现在流程上应该跟图26-12所示的完全相同。

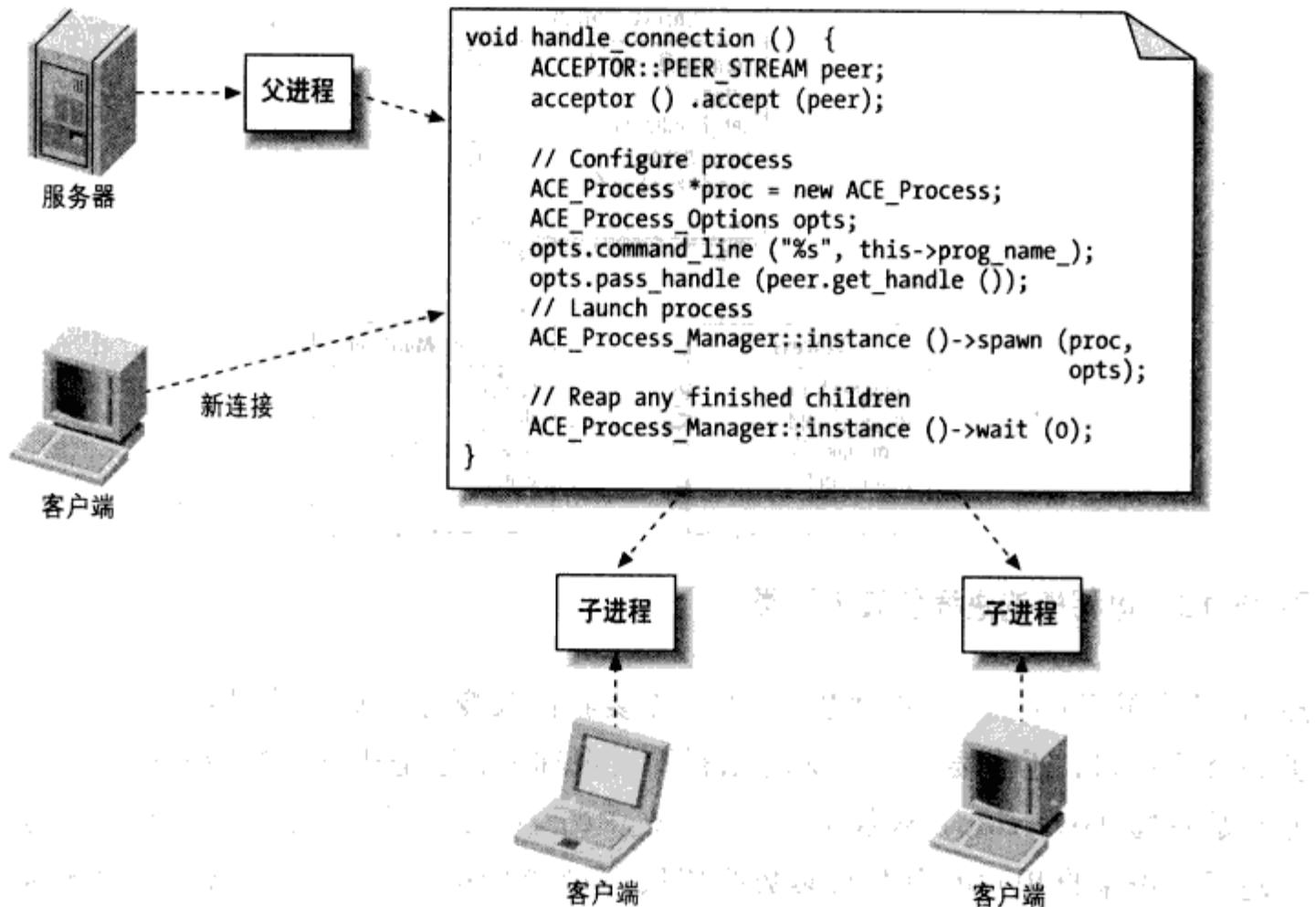


图 26-15：“每连接一进程”服务器的连接处理

重新实现 Logging_Server 基类的 run() 方法使我们可以继续保持其他类型的日志服务器所使用的简单、直接和一致的漂亮的调用方式。

```

int main (int argc, char *argv[]) {
    PPCLogging_Server<SSL_Acceptor, Null_Mutex> server (argc, argv);
    server.run ();
    return 0;
}

```

这个程序的 main 函数跟“每连接一线程”服务器版本的相比，不同之处只有所实例化的类的名字不同，以及采用 Null_Mutex 类进行同步。基于传递给 PPCLogging_Server 构造函数的命令行参数，对父进程或子进程的委派被 run() 方法透明地处理了。

并行日志服务器方案评估

通过利用硬件和 OS 对多个执行线程支持，本节所描述的两种并行日志服务器相比于 ReactiveLogging_Server 和 IterativeLogging_Server，在可扩充能力方面都有大幅提高，能够更容易地随客户端数目的增加做相应扩充。然而，通过平台无关方式实现“每连接一线程”和“每连接一进程”的并发策略并不容易。我们通过用包装外观类来隐藏平台差异，来解决这个问题。基于框架的服务器设计也为 Logging_Server 类提供

了一个公共的外部接口，使针对主服务器事件循环的bug修正和其他改进都能直接集成进我们的实现中。

结论

本章所讨论的日志服务器应用提供了一个既易于理解，又很现实的例子，展示了如何将面向对象的设计/编码技术、模式和框架应用于实现网络化应用程序。特别地，我们的面向对象框架演示了好几几种漂亮的设计元素，从抽象的设计到不同并发模型实现中的具体元素都出现了。我们的设计也用上了C++的语言特性，比如模板和虚函数，同时还有设计模式，比如包装外观和模板方法，它们联合起来创建出一整套可移植、可复用、灵活而又易扩充的日志服务器。

`Logging_Server`基类的`run()`方法中所应用的模板方法模式使我们可以定义日志服务器的一般处理过程，而把其中个别步骤的具体定义延迟到子类的钩子方法中。这个模式虽然有助于将一般的处理过程提升到基类定义中，但它并不足以解决我们所有的可变性需求：譬如同步和IPC机制。因此，为了这些维度上的可变性，我们使用了包装外观模式来隐藏语义和语法差异，最终使这些维度的使用完全正交于各种并发模型的实现。这样的设计使我们可以使用参数化的类来解决这些不同维度上的可变性，从而在不影响其性能的前提下增加了框架的灵活性。

最后，各种不同的并发模型，比如“每连接一线程”和“每连接一进程”，其具体实现中都使用了包装外观模式来使其更加优雅和可移植。最终的结果就是一个节省劳动的软件架构，开发者可以复用一般的设计和编码材料，同时也提供了一种一般的、可参数化的封装可变性的方法。

该日志服务器框架的一个具体实现可从以下地址处可获得：http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/ACE/examples/Beautiful_Code，在ACE的发布包的`ACE_wrappers/examples/Beautiful_Code`文件中也能找到。

以 REST 方式集成业务伙伴

Andrew Patzer

在几年以前，当时我还是一个顾问，曾经在 1 到 2 年的时间里，似乎每个与我交谈过的客户都非常肯定地认为在他的业务中需要一个 Web 服务解决方案。当然，虽然在我的客户中很少有人能真正地理解这意味着什么或者为什么他需要这种架构，但由于他们总是不断地在互联网上、杂志上以及博览会上听说 Web 服务，因此他们认为最好能搭上 Web 服务这趟车，以免错过机会。

请不要误会我的意思。我并不是要反对 Web 服务。我只是不热衷于仅仅根据一些时髦的东西来做出技术决策。本章将给出一些使用 Web 服务架构的理由，并将研究在与外部系统进行集成时所要考虑的一些选择。

在本章中，我将分析一个真实的项目，在这个项目中包括把一组服务开放给某个业务伙伴，此外我们还将讨论一些相关的设计决策。在项目中使用到的技术包括 Java (J2EE)、XML、Rosettanet 电子商务协议和一个用来与运行在 AS/400 系统上的程序进行通信的函数库。我还将讲述接口的使用和工厂 (Factory) 设计模式，我正是通过这种模式使得系统对于将来的销售商来说是可扩展的，而这些销售商可能使用不同的协议或者可能需要访问不同的服务。

项目背景

在本章中讨论的项目开始于一位客户的电话：“我们需要一组 Web 服务把我们的系统和一位销售商的系统集成在一起。”这位客户是一个大型的电子元件制造商。他们使用的系统是 MAPICS，这是一个用 RPG 编写的制造系统，运行在 AS/400 机器上。他们的主要销售商正在升级自己的业务系统并且需要修改与订单管理系统的连接方式以检查产品的现货供应能力和订单状态。

之前，销售商处的操作人员只是远程连接到制造商的 AS/400 系统，并且通过按下“热键 (hotkey)”（我记得是 F13 或者 F14）来访问所需的界面。在随后的代码中你将看到，我为他们开发的新系统叫做 *hotkey*，这是因为 *hotkey* 这个单词已经变成了他们常用语言的一部分，就好像 *google* 在今天已经演变成了一个动词一样。

既然销售商正在实现一个新的电子商务系统，那么它就需要一种自动的方式来把制造商的数据集成到自己的系统。由于这只是客户的销售商之一，尽管是最大的销售商，但客户系统还是需要考虑支持在将来加入其他的销售商以及他们可能使用的任何协议和需求。还有一点就是维护和扩展这个系统的软件人员的技术水平相对较低。虽然他们在其他的领域是非常棒的，但 Java 开发（以及所有类型的 Web 开发）对于他们来说仍然是很新的东西。因此，我知道我所构建的系统必须是简单的并且易于扩展的。

把服务开放给外部客户

在进行这个项目之前，我向我们的用户组和关于 SOAP（简单对象访问协议，Simple Object Access Protocol）和 Web 服务架构的会议做了一些技术报告。因此，当客户电话打来的时候，似乎我所讲述的东西正是这个客户正在寻找的解决方案。然而，在理解了他们的真正需求之后，我认为更好的方式是通过 HTTP 上简单的 GET 和 POST 请求把一组服务开放出去，并且在服务中交换描述这些请求和响应的 XML 数据。不过我在当时并不知道，这种架构形式现在通常叫做 REST，或者叫做具象状态传输 (Representational State Transfer)。

我如何决定在 SOAP 上使用 REST？下面是一些在选择 Web 服务架构时需要考虑的决策因素：

有多少不同的系统需要访问这些服务，并且在当时有多少系统是已知的？

虽然这个制造商知道目前只有一个销售商需要访问这个系统，但它也承认其他的销售商在将来也可能需要访问。

是否有一些终端用户需要预先了解这些服务，或者这些服务是否需要是自描述的，以便于让匿名用户自动进行连接？

因为在制造商及其所有的销售商之间存在一个确定的关系，所以需要保证每个潜在的终端用户都要预先知道如何访问制造商的系统。

在单个事务中需要维护什么样的状态？一个请求是否依赖于前一个请求的结果？

在我们的情况中，每个事务都包含一个请求和一个不依赖于其他任何信息的结果。

在这个项目中对上述问题的回答使我得出了显而易见的决策：在HTTP协议上开放一组已知的服务，并且使用在双方系统中都能够理解的标准电子商务协议来交换数据。如果制造商希望匿名用户也能够查询产品的现货供应能力，那么我可以选择完整的SOAP解决方案，因为这将使系统能够发现这些服务以及可编程的接口而无需预先了解系统。

我目前从事生物信息方面的工作，在这个领域中明确需要SOAP形式的Web服务架构。我们利用了一个叫作BioMoby (<http://www.biomoby.org>) 的项目来定义Web服务并且把它们发布到一个中央存储仓库，以使得其他小组能够正确地把我们的服务应用在构建数据管道的工作流中，从而帮助生物学家们集成不同集合的数据并且对结果进行不同的分析。这是一个完美的示例，它很好地说明了为什么有人选择在REST上的SOAP。匿名用户可以访问我们的数据和工具而甚至无需预先知道它们的存在。

定义服务接口

在确定如何实现这个软件时，首先要确定的就是用户如何发出请求和接受响应。在和这个销售商（主要用户）的一位技术代表讨论之后，我了解到他们的新系统可以通过HTTP POST请求发送一个XML文档，并且把结果作为一个XML文档来分析。XML必须遵循Rosettanet电子商务协议（后面还将做更详细的讨论）的格式，但就目前来说，只要知道这个系统能够通过发送XML格式的请求和响应在HTTP上进行通信就足够了。在图27-1中说明了各个系统之间的常见交互。

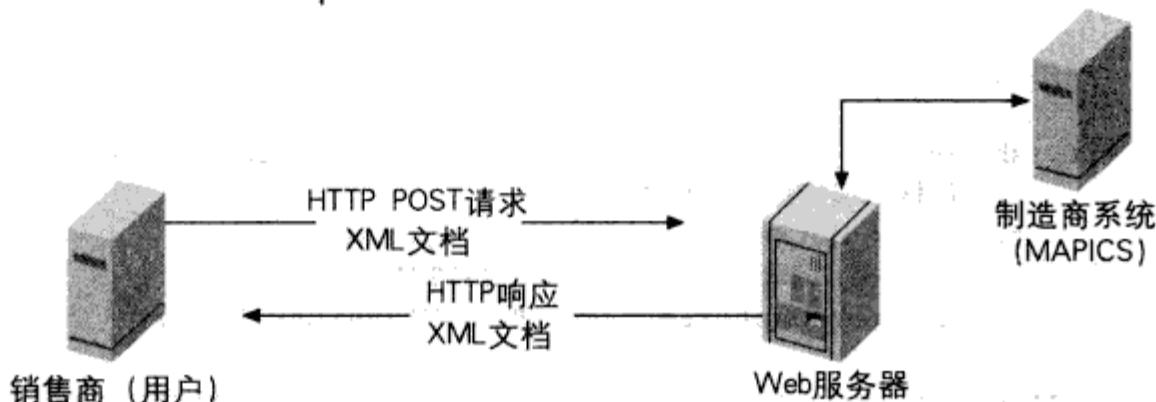


图27-1：后端系统的服务接口

这个制造商最近被一个更大的公司收购了，这个公司在整个组织内都使用 IBM 的产品。因此，我已经知道了所使用的是什么样的应用服务器及相关技术。我把这个服务实现为一个运行在 IBM WebSphere 上的 Java Servlet。由于我知道这个软件将需要使用基于 Java API 来访问运行在 AS/400 服务器上的函数，因此很容易做出这个决策。

下面是在 *web.xml* 文件中的代码，这个文件描述是将为用户提供必要接口的 servlet：

```
<servlet>
    <servlet-name>HotKeyService</servlet-name>
    <display-name>HotKeyService</display-name>
    <servlet-class>com.xxxxxxxxxxxxxx.hotkey.Service</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HotKeyService</servlet-name>
    <url-pattern>/HotKeyService</url-pattern>
</servlet-mapping>
```

servlet 本身仅处理 POST 请求，这是通过重载 Servlet 接口的 *doPost* 方法并且提供了标准生命周期方法的默认实现来完成的。在下面的代码中给出了这个服务的完整实现，但当我最初对问题进行分解并设计一个解决方案时，我首先在代码中写下了一系列的注释作为占位符用来表示将要插入代码的位置。然后我将逐步用代码来代替每个伪码注释，直至最终得出可以运行的实现为止。这将有助于我把注意力放在每段代码如何关联到整个解决方案上：

```
public class Service extends HttpServlet implements Servlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        // 读取请求数据并且保存在 StringBuffer 中
        BufferedReader in = req.getReader();
        StringBuffer sb = new StringBuffer();
        String line;
        while ((line = in.readLine()) != null) {
            sb.append(line);
        }

        HotkeyAdaptor hotkey = null;

        if (sb.toString().indexOf("Pip3A2PriceAndAvailabilityRequest") > 0) {
            // 价格和现货供应信息请求
            hotkey = HotkeyAdaptorFactory.getAdaptor(
                HotkeyAdaptorFactory.ROSETTANET,
                HotkeyAdaptorFactory.PRODUCTAVAILABILITY);
        }
        else if (sb.toString().indexOf("Pip3A5PurchaseOrderStatusQuery ") > 0) {
            // 订单状态
            hotkey = HotkeyAdaptorFactory.getAdaptor(
                HotkeyAdaptorFactory.ROSETTANET,
```

```

        HotkeyAdaptorFactory.ORDERSTATUS);
    }

    boolean success = false;

    if (hotkey != null) {
        /* 传入 XML 请求数据 */
        hotkey.setXML(sb.toString());
        /* 解析请求数据 */
        if (hotkey.parseXML()) {
            /* 执行 AS/400 程序 */
            if (hotkey.executeQuery()) {
                /* 返回响应的 XML */
                resp.setContentType("text/xml");
                PrintWriter out = resp.getWriter();
                out.println(hotkey.getResponseXML());
                out.close();
                success = true;
            }
        }
    }

    if (!success) {
        resp.setContentType("text/xml");
        PrintWriter out = resp.getWriter();
        out.println("Error retrieving product availability.");
        out.close();
    }
}
}

```

仔细阅读这段代码，你可以看到它首先读取请求数据并且将其保存起来以用于后面的操作。然后它将在这个数据中进行查找以确定其是哪种类型的请求：是价格和现货供应信息请求，还是订单状态查询请求。在确定了请求的类型后，将会创建相应的辅助对象。注意，我使用接口HotkeyAdaptor来获得多个实现而无需为每种类型的请求编写大段重复的代码。

这个方法的其他功能包括解析 XML 请求数据，在 AS/400 系统上执行合适的查询，创建 XML 响应并把它通过 HTTP 写回到用户。在下一节中，你将看到我如何通过接口和非常流行的工厂设计模式把实现细节隐藏起来。

使用工厂模式转发服务

这个系统的需求之一就是通过最少的编程工作来满足将来不同类型系统的多种请求。我认为能通过把实现简化为单个命令接口来满足这个需求，并且在这个接口中公开了一些基本的方法来响应各种请求：

```

public interface HotkeyAdaptor {
    public void setXML(String _xml);
    public boolean parseXML();
    public boolean executeQuery();
    public String getResponseXML();
}

```

那么，这个 servlet 如何确定对接口的哪种实现进行实例化？它将首先在请求数据中查找特定的字符串以判断请求的类型是什么。然后，它使用工厂对象的静态方法来选择合适的实现。

这个 servlet 知道，我们所使用的实现将会为每个方法提供合适的响应。通过使用主 servlet 中的接口，我们只需把执行代码编写一次，而无需考虑它所处理的请求是何种类型或者是谁发出的请求。所有的细节都被封装在接口的每个独立实现中。以下是这个 servlet 中的一些代码：

```

HotkeyAdaptor hotkey = null;

if (sb.toString().indexOf("Pip3A2PriceAndAvailabilityRequest") > 0) {
    // 价格和现货供应信息请求
    hotkey = HotkeyAdaptorFactory.getAdaptor(
        HotkeyAdaptorFactory.ROSETTANET,
        HotkeyAdaptorFactory.PRODUCTAVAILABILITY);
}

else if (sb.toString().indexOf("Pip3A5PurchaseOrderStatusQuery") > 0) {
    // 订单状态
    hotkey = HotkeyAdaptorFactory.getAdaptor(
        HotkeyAdaptorFactory.ROSETTANET,
        HotkeyAdaptorFactory.ORDERSTATUS);
}

```

在 HotkeyAdaptorFactory 这个工厂对象中定义了一个静态方法，这个方法包含两个参数，分别表示它所使用的是哪种协议以及是哪种类型的请求。这些参数的值都被定义为工厂对象本身的静态常量。在下面的代码中可以看到，工厂对象只是简单地使用一个 switch 语句来选择合适的实现：

```

public class HotkeyAdaptorFactory {

    public static final int ROSETTANET = 0;
    public static final int BIZTALK = 1;
    public static final int EBXML = 2;

    public static final int PRODUCTAVAILABILITY = 0;
    public static final int ORDERSTATUS = 1;

    public static HotkeyAdaptor getAdaptor(int _vocab, int _target) {
        switch (_vocab) {

```

```

        case (ROSETTANET) :
            switch (_target) {
                case (PRODUCTAVAILABILITY) :
                    return new HotkeyAdaptorRosProdAvailImpl();
                case (ORDERSTATUS) :
                    return new HotkeyAdaptorRosOrdStatImpl();
                default :
                    return null;
            }
        case (BIZTALK) :
        case (EBXML) :
        default :
            return null;
    }
}

```

虽然这看上去是一个非常简单的抽象，但在使得经验欠缺的程序员能够阅读并且理解这段代码之前，我们付出了很多的努力。当需要增加一个新的销售商，并且这个销售商正在使用Microsoft的BizTalk服务，同时他还希望通过电子的方式下订单时，程序员就可以用一个简单的模板来增加这个新的需求。

用电子商务协议来交换数据

在这个项目中，有些新的东西是关于标准电子商务协议使用的。当销售商向我提出使用Rosettanet标准来交换请求和响应的需求时，我不得不首先做了一些研究。我首先从Rosettanet网址 (<http://www.rosettanet.org>) 上下载了我所感兴趣的具体标准，从中找到了一张详细讲解业务伙伴之间典型交易的示意图，以及关于XML请求和响应的规范。

由于我需要做许多反复试验的工作，因此首要的事情就是建立一个测试环境，这样我可以模拟与销售商的交互过程而无需每次再与他们的工作人员联合测试。我使用了Apache Commons HttpClient 来管理HTTP交换：

```

public class TestHotKeyService {

    public static void main (String[] args) throws Exception {

        String strURL = "http://xxxxxxxxxxxx/HotKey/HotKeyService";
        String strXMLfilename = "SampleXMLRequest.xml";
        File input = new File(strXMLfilename);

        PostMethod post = new PostMethod(strURL);
        post.setRequestBody(new FileInputStream(input));
        if (input.length() < Integer.MAX_VALUE) {
            post.setRequestContentLength((int)input.length());
        } else {
            post.setRequestContentLength(

```

```

        EntityEnclosingMethod.CONTENT_LENGTH_CHUNKED);
    }

    post.setRequestHeader("Content-type", "text/xml; charset=ISO-8859-1");

    HttpClient httpclient = new HttpClient();
    System.out.println("[Response status code]: " +
        httpclient.executeMethod(post));
    System.out.println("\n[Response body]: ");
    System.out.println("\n" + post.getResponseBodyAsString());

    post.releaseConnection();
}

}

```

在试验了几种不同类型的请求并分析了结果之后，我加快了自己的学习曲线。我坚定不移地认为，越快开始编写代码，就越能提高学习效果。你从书中、某个网站上的一篇文章或者一组 API 文档中只能学习部分知识。只有尽早地着手把学到的东西应用起来，才能发现许多通过简单研究这个问题无法发现的事情。

Rosettanet 标准与其他标准一样，是非常详尽的和完整的。在完成任何任务的时候，你可能最终只会用到其中的一小部分内容。对于这个项目来说，我只需要设置一些标准的标识域，以及在价格查询时设置一个产品编号和有效期，或者在查询订单状态时设置一个订单号。

用 XPath 解析 XML

XML 请求数据不仅仅只是简单的 XML。正如在前面所提到的，Rosettanet 标准是非常详尽和完整的。如果没有 XPath，那么解析这样的一个文档将是一场可怕的恶梦。通过使用 XPath 映射，我可以定义到我所感兴趣的每个节点的精确路径，并且能够很容易地提取出必要的数据。我把这些映射实现为一个 HashMap，然后在其中进行迭代，并且提取出特定的节点再用这些值来创建一个新的 HashMap。这些值将被同时用在 executeQuery 和 getResponseXML 这两个方法中，我在后面将会介绍这些方法：

```

public class HotkeyAdaptorRosProdAvailImpl implements HotkeyAdaptor {

    String inputFile;           // 请求 XML
    HashMap requestValues;     // 保存请求中的 XML 值
    HashMap as400response;     // 保存从 RPG 调用中返回的参数

    /* 声明 XPath 映射并且用一个静态的初始化块来赋值 */
    public static HashMap xpathmappings = new HashMap();
    static {
        xpathmappings.put("from_ContactName",

```

```

    "/Pip3A2PriceAndAvailabilityRequest/fromRole/PartnerRoleDescription/
    ContactInformation/contactName/FreeFormText");
        xpathMappings.put("from_EmailAddress",
    "/Pip3A2PriceAndAvailabilityRequest/fromRole/PartnerRoleDescription/
    ContactInformation/EmailAddress");
    }
    // 为了保持简洁性而省略 xpath 映射...
}

public HotkeyAdaptorRosProdAvailImpl() {
    this.requestValues = new HashMap();
    this.as400response = new HashMap();
}

public void setXML(String _xml) {
    this.inputFile = _xml;
}

public boolean parseXML() {
    try {
        Document doc = null;
        DocumentBuilderFactory dbf = DocumentBuilderFactory.
newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        StringReader r = new StringReader(this.inputFile);
        org.xml.sax.InputSource is = new org.xml.sax.InputSource(r);
        doc = db.parse(is);

        Element root      = doc.getDocumentElement();

        Node node = null;

        Iterator xpathvals = xpathMappings.values().iterator();
        Iterator xpathvars = xpathMappings.keySet().iterator();
        while (xpathvals.hasNext() && xpathvars.hasNext()) {
            node = XPathAPI.selectSingleNode(root, String)xpathvals.next());
            requestValues.put((String)xpathvars.next(),
                node.getChildNodes().item(0).getNodeValue());
        }
    }
    catch (Exception e) {
        System.out.println(e.toString());
    }
    return true;
}
public boolean executeQuery() {
    // 以下代码省略...
}

public String getResponseXML() {
    // 以下代码省略...
}
}

```

在 executeQuery 方法中包含了在访问运行于 AS/400 系统之上的 RPG 代码所需的代码，这是为了获得在随后构造响应 XML 文档时必要的响应数据。在许多年以前，我曾工作过的一个项目是把一个 MAPICS 系统（运行在 AS/400 上的 RPG）和一个我用 Visual Basic 编写的新系统集成在一起。我在数据交换的两端别用 AS/400 上的 RPG、CL 以及 PC 上面的 Visual Basic 来编写代码。这使我不做了好几个演讲报告，在报告中我努力向许多 RPG 程序员讲述如何把他们的遗留系统和现代的客户 / 服务器软件集成起来。在当时，这确实是一件复杂而又神秘的事情。

从那以后，IBM 使这项任务变得非常容易并且为我们提供了一个 Java 函数库来做所有的工作（这就是我在这个项目中得到的所有东西）。在以下的代码中使用了来自 IBM 的 Java 库：

```
public boolean executeQuery() {  
  
    StringBuffer sb = new StringBuffer();  
  
    sb.append(requestValues.get("from_ContactName")).append("|");  
    sb.append(requestValues.get("from_EmailAddress")).append("|");  
    sb.append(requestValues.get("from_TelephoneNumber")).append("|");  
    sb.append(requestValues.get("from_BusinessIdentifier")).append("|");  
    sb.append(requestValues.get("prod_BeginAvailDate")).append("|");  
    sb.append(requestValues.get("prod_EndAvailDate")).append("|");  
    sb.append(requestValues.get("prod_Quantity")).append("|");  
    sb.append(requestValues.get("prod_ProductIdentifier")).append("|");  
  
    try {  
        AS400 sys = new AS400("SS100044", "ACME", "HOUSE123");  
  
        CharConverter ch = new CharConverter();  
        byte[] as = ch.stringToByteArray(sb.toString());  
  
        ProgramParameter[] parmList = new ProgramParameter[2];  
        parmList[0] = new ProgramParameter(as);  
        parmList[1] = new ProgramParameter(255);  
  
        ProgramCall pgm = new ProgramCall(sys,  
                                         "/QSYS.LIB/DEVOBJ.LIB/J551231.PGM", parmList);  
        if (pgm.run() != true) {  
            AS400Message[] msgList = pgm.getMessageList();  
            for (int i=0; i < msgList.length; i++) {  
                System.out.println(msgList[i].getID() + " : " +  
                                   msgList[i].getText());  
            }  
        }  
        else {  
            CharConverter chconv = new CharConverter();  
            String response =  
                chconv.byteArrayToString(parmList[1].getOutputData());  
        }  
    }  
}
```

```

StringTokenizer st = new StringTokenizer(response, "|");

String status = (String) st.nextToken().trim();
as400response.put("Status", status);
String error = (String) st.nextToken().trim();
as400response.put("ErrorCode", error);
String quantity = (String) st.nextToken().trim();
as400response.put("Quantity",
    String.valueOf(Integer.parseInt(quantity)));

if (status.toUpperCase().equals("ER")) {
    if (error.equals("1")) {
        as400response.put("ErrorMsg",
            "Account not authorized for item availability.");
    }
    if (error.equals("2")) {
        as400response.put("ErrorMsg", "Item not found.");
    }
    if (error.equals("3")) {
        as400response.put("ErrorMsg", "Item is obsolete.");
        as400response.put("Replacement",
            (String) st.nextToken().trim());
    }
    if (error.equals("4")) {
        as400response.put("ErrorMsg",
            "Invalid quantity amount.");
    }
    if (error.equals("5")) {
        as400response.put("ErrorMsg",
            "Preference profile processing error.");
    }
    if (error.equals("6")) {
        as400response.put("ErrorMsg",
            "ATP processing error.");
    }
}
}

catch (Exception e) {
    System.out.println(e.toString());
}

return true;
}

```

这种方法首先构造一个传递给 AS/400 程序的参数字符串（通过管道符号来分隔），在这个程序中将解析字符串，提取请求数据，并且返回一个带有状态和错误码的字符串（通过管道符号来分隔）以及操作结果。如果没有错误，那么这个与 AS/400 交互的结果将被存储在另一个 HashMap 中，并用来构造 XML 响应文档。如果有错误，那么将写入到响应中。

构造 XML 响应

我总是乐于看到人们创建 XML 文档的许多方式。我通常告诉人们 XML 文档只是一个庞大的文本字符串。因此，使用 `StringBuffer` 来写出一个 XML 文档比构建一个文档对象模型（Document Object Model, DOM）或者使用专门的 XML 生成库要更加容易。

在这个项目中，我只是创建了一个 `StringBuffer` 对象，并且把遵循 Rosettanet 标准的 XML 文档中的每行文本都添加在到这个对象中。虽然我省略了几行代码，但以下的代码还是可以告诉你如何来构造响应：

```
public String getResponseXML() {  
  
    StringBuffer response = new StringBuffer();  
    response.append("<Pip3A2PriceAndAvailabilityResponse>").append("\n");  
    response.append("<ProductAvailability>").append("\n");  
    response.append("<ProductQuantity>").append(as400response.get  
    ("Quantity")).append("</ProductQuantity>").append("\n");  
    response.append("</ProductAvailability>").append("\n");  
    response.append("<ProductIdentification>").append("\n");  
    response.append("<PartnerProductIdentification>").append("\n");  
    response.append("<GlobalPartnerClassificationCode>Manufacturer</  
    GlobalPartnerClassificationCode>").append("\n");  
    response.append("<ProprietaryProductIdentifier>").append  
    (requestValues.get("prod_ProductIdentifier")).append("</  
    ProprietaryProductIdentifier>").append("\n");  
    response.append("</PartnerProductIdentification>").append("\n");  
    response.append("</ProductIdentification>").append("\n");  
    response.append("</ProductPriceAndAvailabilityLineItem>").append("\n");  
    response.append("</Pip3A2PriceAndAvailabilityResponse>").append("\n");  
    return response.toString();  
}
```

结论

当我回首这段两年前编写的代码时，很自然地进行了自我反省并且考虑我是否可以用更好的方式来编写这段代码。虽然我可以编写一些不同的实现代码，但我认为自己仍然会按照相同的方式来设计它。这段代码经受了时间的考验，因为客户自己在添加新的销售商和新的请求类型时，很少需要借助像我这样的外部技术人员。

目前，作为生物信息部门的主管，当我向手下的人员讲授面向对象设计原则和 XML 解析技巧时，我通常会用这段代码来说明一些问题。我本应该在本章中写一些最近开发的代码，但我认为在这段代码中说明了几个基本的原则，并且这些原则对于任何年轻的软件开发人员来说都是应重点理解的。

漂亮的调试

Andreas Zeller

“我叫 Andreas，我曾有过调试的经历。”欢迎来到 Debuggers Anonymous，在这里你可以讲述自己的调试故事，并且从其他人的故事中找到安慰。你是不是又没有在家里睡上一觉？还好你只是在调试器前度过了一晚上。你还是无法告诉你的经理何时才能改好这个程序？让我们多往好的方面想想吧。隔壁工作间的同事吹嘘花了连续36小时的时间查找一个 bug？这确实是令人难忘的！

不，调试并没有什么可炫耀的。它是我们工作中的丑小鸭，是一个还远未完善的任务，是一种最不可预测或者最无法解释的行为，它还经常会使你感到懊悔和自责：“如果我们在一开始能够做得更好，那么就不会陷在目前这种困境中”。如果说程序中的缺陷是一种犯罪，那么调试就是相应的惩罚。

假设我们已经竭尽所能来防止错误的发生，但有时仍然会发现需要进行调试。与所有其他的工作一样，我们需要以最专业和最“漂亮的”方式来处理调试。

那么，在调试中是否也存在着漂亮性呢？我相信是有的。在自己的程序员生涯中，我就在许多调试过程中发现了漂亮性。这些漂亮性不仅帮我解决了问题，而且还演变成了一种全新的调试方法，这种方法不仅是“漂亮的”，同时还能够提升你的调试效率。这不

仅是因为这些方法是系统化的——它们能够确保你获得问题的解决方案——而且还因为这些方法在某种程度上甚至是自动的——当你进行其他任务的时候，它们会帮你完成所有的工作。

很奇怪吗？请继续阅读下去。

对调试器进行调试

我第一次在调试中遇到漂亮性是来源于我的一个学生。1994年，Dorothea Lütkehaus在她的硕士论文中构建了一个可视化的调试器界面，用来为数据结构提供规范的视图。图28-1给出了这个工具软件的截图，这个软件叫作数据显示调试器（*data display debugger*），或者简称为*ddd*。当Dorothea演示这个调试器时，所有的观众和我都感到很吃惊：这个软件可以在几秒钟内解析复杂的数据，并且仅通过鼠标就可以浏览和操纵这些数据。

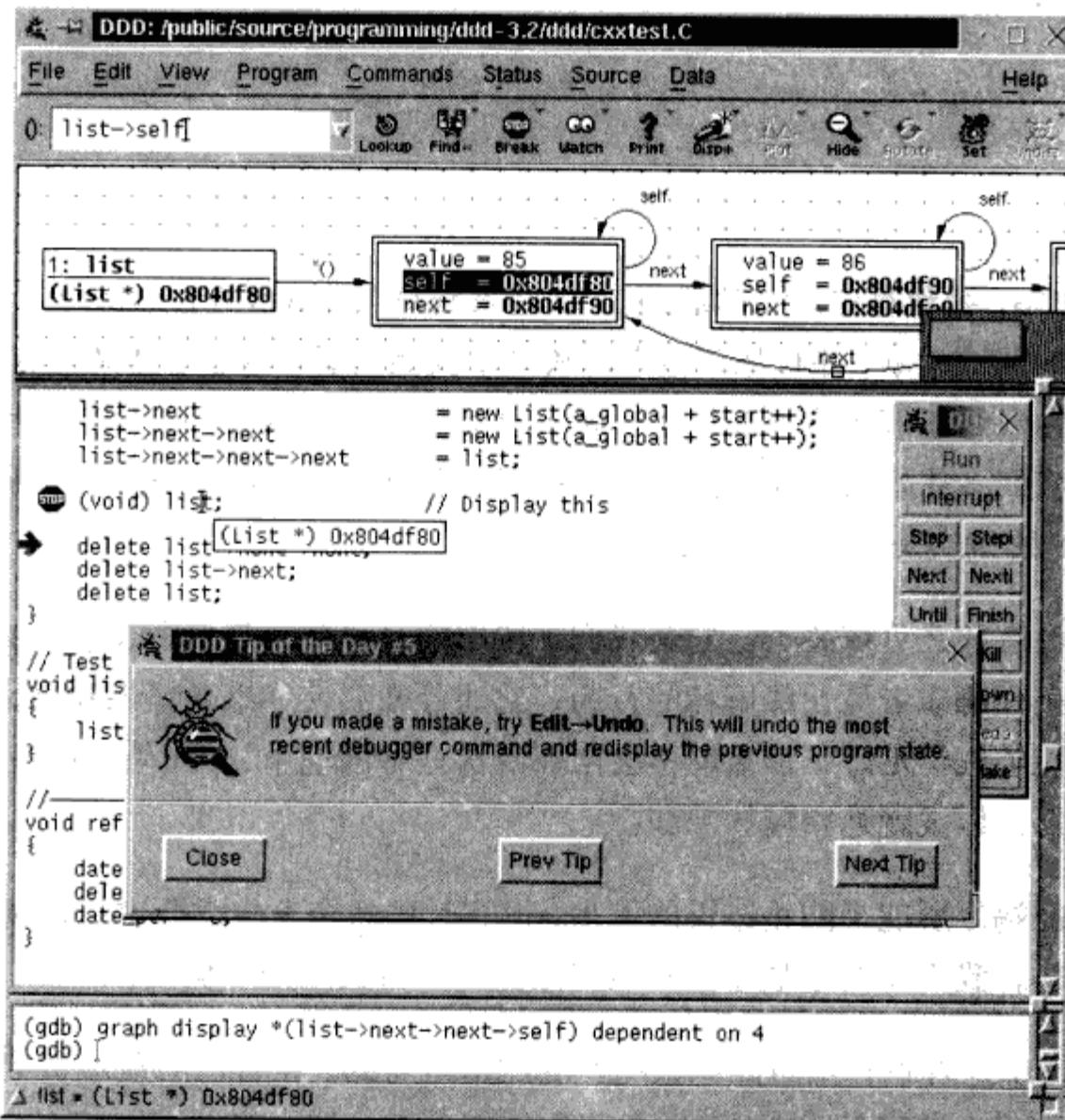


图 28-1：运行中的 *ddd* 调试器

*ddd*是一个对命令行调试器（*gnu* 调试器 *gdb*）进行外层包装的软件，*gdb*是一个功能强大但却难以使用的工具。由于在当时图形化的编程工具仍然是很少的，因此 *ddd* 可以算是一个小小的革命。在接下来的几个月中，Dorothea 和我努力使 *ddd* 成为最漂亮的调试器界面，并且它最终成为了 *GNU* 系统中的一部分。

虽然用 *ddd* 来调试比用命令行工具来调试通常要更加有趣，但它无法使你成为一个更高效的调试人员。因此，调试过程远比调试工具要重要。顺便提一下，我也正是在 *ddd* 中学到了这个观点。所有的事情都起源于我在 1998 年 7 月 31 号收到的一个关于 *ddd* 的 bug 报告，我从收到这个报告起便开始了第二次关于调试漂亮性的体验之旅。以下是这个 bug 报告的内容：

在 GDB 4.16 上使用 DDD 时，run 命令能够使用之前的命令行参数，或者 set args 的值。然而，当我切换到 GDB 4.17 时，DDD 却无法工作：如果在控制台窗口中输入一个 run 命令，那么前面的命令行选项将会丢失。

gdb 的开发人员对 *gdb* 进行了一些修改并且发布了一个新的版本，这个版本的行为与前面的版本稍有不同。因为 *ddd* 是一个前台程序，它的工作流程实际上是首先把命令发送给 *gdb*，就像用户键入命令一样，然后对 *gdb* 的响应进行解析并将这些信息显示在用户界面上。很明显，在这个过程中出现了问题。

我要做的事情就是拿到新的 *gdb* 版本并且安装，然后把 *ddd* 作为 *gdb* 的前台程序运行起来，接着看看能否重现 bug 报告中问题。如果可以重现，那么我将启动另一个调试器实例来分析这个问题。总而言之，这是一个很普通的任务。

然而，我在当时非常厌烦运行调试器来调试我们自己的调试器，尤其是由于第三方软件的变化而产生的调试工作。因此，我没有立即采取行动，而是开始思考：有没有一种方法能够不启动调试器也可以解决这个问题？或者：是否可以无需进行调试也能够找出软件中的 bug？

由于这个问题是开发人员对 *gdb* 的源代码进行了修改而造成的，因此我可以只需查看 *gdb* 的源代码——或者更准确地说，查看两个发布版本之间的差异。我认为通过代码差异能够直接找出导致问题的修改。我需要做的就是在 diff 中运行这两个代码库，而我也正是这么做的。

diff 给出的结果使我感到吃惊。存在差异的代码大概有 178 200 行，这是非常庞大的——*gdb* 的所有代码行数也就是大约 600 000 行。开发人员至少在 8 721 个位置上对源代码进行了插入、删除或者修改等操作。这些差异对于一个“次要”发布版本来说显然太多了，当然，我是无法处理如此之多的修改的。即使在一处修改位置上进行判断只需 10

秒钟，我仍然要花 24 小时来找出导致问题的修改。我叹了一口气，启动调试器，强打起精神开始了另一次调试过程。但我仍然认为，肯定存在更好的方式来完成这个任务——一种更“漂亮的”方式。

系统化的过程

当程序员调试程序时，他们会查找导致问题的起因，这个起因可能存在于代码中，也可能存在于输入的数据或运行的环境中。这些起因必须被找到并消除。只有在消除了所有导致问题的起因后，程序才能正确运行（如果在消除了这些起因后，程序仍然运行失败，那么我们就需要重新修改对起因的判断）。

查找起因的一般过程叫作科学方法。在查找程序故障的起因时，科学方法的工作方式将是下面这样的：

1. 观察程序的故障。
2. 对与观察结果一致的故障起因做一个假设。
3. 通过假设来进行预测。
4. 通过试验对预测进行判断并做进一步观察。
 - a. 如果试验和观察能够满足预测，则对假设进行细化。
 - b. 如果不满足，则换另一种假设。
5. 重复步骤 3 和步骤 4，直到假设不能再被细化为止。

最后，假设就成为了事实。这意味着你有了一个概念上的框架，可以用来解释（并且预测）普遍现象的某个方面。你的问题程序可能只是普遍现象中一个非常小的方面，但得出的理论仍然可以很好地预测出应该在什么位置修改程序。

为了获得这样的事实，程序员在分析导致故障的因果链时，应该使用科学方法。他们需要：

1. 观察故障（输出结果是错的）
2. 对于故障的起因提出一个假设（问题可能是 y 的值是错的）。
3. 做出预测（如果 y 是错的，那么它的值可能是来自于第 632 行的函数 $f()$ ）。
4. 对他们的预测进行判断（确实如此， y 在第 632 行得到了一个错误的值）。

5. 得出相应的结论（这意味着函数 `f()` 返回了一个错误的值。现在，这个值是从何而来的？）。

在所有的方法、线索和技巧中，坚持使用科学方法是成为调试大师的关键因素。这意味着要做到以下三个方面：

明确化

明确地阐述你的假设。把你的问题记录下来或者向其他人解释这个问题。将你的假设和观察保存为书面记录，这样可以使当天的工作暂时告一段落，而在第二天早晨以清醒的头脑来继续工作。

系统化

要清楚地知道你正在做什么。不要在没有清晰的假设和预测的情况下，就去随机地在某个位置上进行分析（或者修改）。你要确保没有遗漏所有可能的故障起因。

首先找到可能性最高的起因

虽然科学方法能确保你找到起因，但不会告诉你什么时候才能找到。因此，你要首先识别出所有可能的故障起因，然后集中关注那些可能性最高的起因，而对这些起因进行分析的代价将是最小的。

不幸的是，交互式调试器并不支持科学方法。诚然，在浏览和分析代码时，调试器是功能强大的工具。这是一个很好的东西——但仅对于知道如何系统地使用调试器的熟练程序员来说是这样的。我更希望看到程序员学习系统化的调试方法而不是漂亮的调试工具（我仍然为自己编写了 `ddd` 这样一个漂亮的调试工具而感到内疚）。

关于查找的问题

让我们回到最初对调试器进行调试的问题。即使在找到并修改了 `bug` 之后，我仍然在想：有没有一种方法能够自动找出导致故障的代码变动？程序员需要的是一种测试，并且在每次程序员修改代码后将自动执行这个测试。如果没有通过这个测试，那我们就能立刻知道最近做了哪些修改，因此也就可以立即改正问题（几年之后，David Saff 和 Michael Ernst 用 Continuous Testing 插件实现了这种思想）。

在我遇到的情况中，我知道导致测试失败的原因是代码变动——从 `gdb 4.16` 到 `gdb 4.17` 的代码修改。但问题在于这两个版本之间的改动太大了，涉及 8 721 个位置。应该有某种方法能够将这个改动拆分为更小的部分。

如果我将这个大的改动拆分为 8 721 个更小的改动，这样每个改动只会影响一个位置上

的代码，那么会发生什么情况？如果是这样，那我就可以逐个应用每个改动并进行测试，直到测试失败——最近被应用的改动就是使测试失败的改动。换句话说，我将模拟 4.17 版本的开发过程（事实上，并不是由我亲自来模拟这个过程，而是由我构造的一个工具来模拟。当我坐下来品尝热茶时，和我的孩子们玩游戏或者阅读邮件时，这个灵巧的小工具将自动找出导致故障的代码改动。这太棒了）。

然而，这其中还存在一个问题。我并不知道应该按照什么样的次序来应用这些改动。这个次序是非常关键的，因为每个改动可能会依赖于其他的改动。例如，在改动 A 中可能会引入一个变量，而这个变量将在改动 B 或者改动 C 中被使用。当应用 B 或者 C 时，必须首先应用 A；否则，编译 *gdb* 将会失败。同样，改动 X 可能修改了某个函数定义的名字，而在其他位置的改动 (Y, Z) 可能反映出这个重命名修改。如果应用了 X，那么 Y 和 Z 也必须被应用，否则的话将无法编译 *gdb*。

那么如何来确定一个改动是否依赖于另一个改动呢？这个问题看上去非常棘手——如果没有强大的（然而这目前还是不存在的）多版本程序分析，那么几乎是不可处理的。

可不可以尝试按照各种不同的次序来应用改动呢？8 721 个改动可以有 $8\,721 \times 8\,720 \times 8\,719 \times \dots \times 1 = 8\,721!$ 种不同的排序方式。任何人都无法测试所有这些排序方式。那么试一试所有的子集或许更好：8 721 个改动意味着 $2^{8\,721} = 10^{2\,625}$ 个子集，这意味着比测试 8 721 阶乘的排序数少了很多。我可以试着安慰自己，等这些计算最终在我的机器上算完时，量子计算 (quantum computing)、时间旅行 (time travel) 以及一些完美程序早已成为主流，因此也就不再需要这种无意义的尝试了。

于是，我尝试了另一种方式。用古老的分而治之算法效果如何？我们可以首先在 *gdb* 4.16 上应用一半数量的改动，然后进行测试。如果 *DDD* 失败了，那么我们就知道导致故障的改动是在这一半中；如果没有失败，那么我们就在另一半中搜索。在每次测试中，我们都将减少一半的搜索空间，直到最终找到导致故障的改动。这就是我所想到的方式：自动应用科学方法，系统地创建，测试并且细化假设。

但问题再次出现——如果在应用一组改动时导致了不一致的代码，那么我们又该怎么办呢？我不知道。

自动找出故障起因

我花了三个月的时间来想出解决方案——那是突如其来的灵感，当时是早上六点钟，我正躺在床上。太阳正在升起，鸟儿正在歌唱，这时我突然有了一个主意。推理过程如下所示：

- 如果应用一半数量的改动，那么很可能无法得到一致的构建；跳过一个相关的改动无疑是高风险的。而如果我们能够得到一致的构建（或者一个“已决的（resolved）”结果），就可以很快地缩小这组改动。
- 另一方面，应用单个改动很可能获得一个有意义的结果——尤其是将这个改动应用到一致的版本上。例如，我们来考虑对单个函数的改动，除非函数的接口改变了，否则我们很可能得到一个可运行的程序。而另一方面，如果逐个地应用每个改动，那么将花费太长的时间。

因此，我想出了一种折衷方案：首先将所有的改动分成两半。如果这两组改动都无法得到可测试的构建，那么我将把这两组改动进一步拆分成四个子集，然后将每个子集独立地应用到 *gdb* 4.16 上。此外，我还可以在 *gdb* 4.17 源代码上去掉对这个子集的应用（把这个子集的补集应用到 *gdb* 4.16 的源代码上）。

将全部的改动拆分为四个（而不是两个）子集意味着将应用更小的改动集合，这也意味着这个版本将更接近于原始版本——并且更有机会得到一致的构建。

如果四个子集还不够，那么我将继续拆分为 8 个、16 个、32 个等等数量的子集，直到最终逐个地应用单个改动——这最可能使我得到一致的构建。在得到一个可测试的构建后，这个算法将从头开始计算。

在最坏的情况下，这个算法将需要 $8\ 721^2 = 76\ 055\ 841$ 次测试。这个数字仍然是过高的，但比前面指数量级的方法要好多了。另一种极端的情况是，如果所有的构建都成功，那么这个算法将以二叉树的方式计算，而这只需要 $\log_2 8\ 721 = 14$ 次测试。在特定的概率分布下，这种算法值不值得实现？

我用 Python 脚本实现了上述算法的一个非常简单的版本。算法的关键部分在于测试函数。函数的参数是一组代码改动，首先调用 *patch* 将这些改动应用到 *gdb* 4.16 的源代码上，然后调用 *make* 来编译改动之后的 *gdb*。最后，它将运行 *gdb* 并且看看是否会发生故障（如果发生故障，将返回 *fail*，否则返回 *pass*）。由于这些步骤中的任何一个都可能失败，因此测试函数还会返回 *unresolved* 的测试结果。

当我开始运行这个脚本时，很快就发现 *unresolved* 返回值的出现频率很高。实际上，在最初的大约 800 次测试中，测试函数总是返回 *unresolved*。改动子集的数量从 2 到 4 再到 8 ——直到 64 个子集，其中每个子集包含了 136 个改动。这些测试很是花费了一些时间。由于 *gdb* 的编译过程需要大约 6 分钟，因此我差不多等了三天的时间（事实上我并没有等待，而是在写我的博士论文。但仍然……）

在这个过程中发生了一些奇怪事情，于是我检查结果记录。有一个测试失败了！现在，我将可以看到这个算法如何作用于更小的集合上，并且缩小搜索空间。但当我检查结果时，却发现测试将以下的信息输出到屏幕上并且停止执行：

```
NameError: name 'next_c_fial' is not defined
```

在持续计算了三天之后，在我的脚本中意外发现了一个拼写错误。我真希望当初使用的是带有静态检测的语言而不是 Python。

我改好了这个问题并且再次运行脚本。现在，脚本能够正确地运行了。在五天之后，大约进行了 1 200 次测试，这个脚本最终分离出了导致故障的代码改动：使得 *ddd* 失败的 *gdb* 代码改动。只有一行的代码改动，这甚至不能算是对程序代码的改动，而只是对一些内置文本的改动：

```
diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is started.\n\
---
> "Set argument list to give program being debugged when it is started.\n\
```

将 *arguments* 改动为 *argument list* 正是导致 *ddd* 无法在 *gdb 4.17* 上工作的原因。当用户请求 *set args* 命令的帮助时，*gdb* 将输出这段文本。然而，这个改动在其他的地方也被使用。当给定的命令显示 *args* 时，*gdb 4.16* 将响应：

```
Arguments to give program being debugged is "11 14"
```

而 *gdb 4.17* 将响应：

```
Argument list to give program being debugged is "11 14"
```

这个新的响应正是使 *ddd* 产生错误的原因，因为 *ddd* 希望响应是以 *Arguments* 作为开头的单词。这样，我的脚本就找出了导致故障的代码改动——在运行了五天之后，而且这是一个完全自动的版本。

增量调试

在接下来的几个月里，我细化并且优化了算法以及 *ddd* 这个工具，而且最终只需要一个小时就可以找出导致故障的改动。我发布了这个算法，并且取名为增量调试（Delta Debugging），因为它是通过分离增量，也就是两个版本之间的差异，来“调试”程序的。

下面我将给出增量调试算法的 Python 实现。函数 *dd()* 带有三个参数——两组代码改动信息和一个测试：

- 在列表 `c_pass` 中包含了“成功的”配置——在我们的情况中，就是使得程序能够成功运行的一组必须应用的改动。在我们的情况中（通常情况下），这是一个空的列表。
- 在列表 `c_fail` 中包含了“失败的”配置——在我们的情况中，就是使程序运行失败的一组改动。在我们的情况中，这就是所有的 8721 处改动（我们在 `Change` 对象中封装这组改动）。
- 测试函数将接收一组改动，然后应用这组改动并进行测试。根据测试的结果是成功、失败还是未决的，函数将返回 `pass`、`fail` 或者 `unresolved`。在我们的情况下，测试函数将通过 `patch` 来应用改动，并根据刚才介绍的方式来运行测试。
- `dd()` 函数将逐步缩小 `c_pass` 和 `c_fail` 之间的差异，并且最终返回一个三元组。三元组中的第一个值是被分离的增量——在我们的情况下，就是一个包含 `gdb` 源代码中一行改动的 `Change` 对象。

如果你希望自行实现 `dd()`，那么可以很容易地使用这里给出的代码（在本书的 O'Reilly 网页上也包含了这段代码）。此外，你还需要三个其他的函数：

`split(list, n)`

将一个列表拆分为 `n` 个等长（除了最后一个之外）的子列表。以下是这个函数的调用示例：

`split([1, 2, 3, 4, 5], 3)`

将生成：

`[[1, 2], [3, 4], [5]]`

`listminus()` 和 `listunion()`

这两个函数分别用来返回两个列表的集合差或者集合并。以下是这个函数的调用示例：

`listminus([1, 2, 3], [1, 2])`

将生成：

`[3]`

而：

`listunion([1, 2, 3], [3, 4])`

将生成：

`[1, 2, 3, 4]`

在示例 28-1 中给出了 Python 代码。

示例 28-1：对于增量调试算法的实现

```
def dd(c_pass, c_fail, test):
    """返回一个三元组 (DELTA, C_PASS', C_FAIL')，这样
    C_PASS 是 C_PASS' 的子集，C_FAIL' 是 C_FAIL 的子集
    DELTA = C_FAIL' - C_PASS' 是在 C_PASS' 和 C_FAIL' 之间最小的差异,
    与 TEST 相关."""
    n = 2 # 子集数量

    while 1:
        assert test(c_pass) == PASS # 恒成立
        assert test(c_fail) == FAIL # 恒成立
        assert n >= 2

        delta = listminus(c_fail, c_pass)

        if n > len(delta):
            # 无法进一步缩小
            return (delta, c_pass, c_fail)

        deltas = split(delta, n)
        assert len(deltas) == n

        offset = 0
        j = 0
        while j < n:
            i = (j + offset) % n
            next_c_pass = listunion(c_pass, deltas[i])
            next_c_fail = listminus(c_fail, deltas[i])

            if test(next_c_fail) == FAIL and n == 2:
                c_fail = next_c_fail
                n = 2; offset = 0; break
            elif test(next_c_fail) == PASS:
                c_pass = next_c_fail
                n = 2; offset = 0; break
            elif test(next_c_pass) == FAIL:
                c_fail = next_c_pass
                n = 2; offset = 0; break
            elif test(next_c_fail) == FAIL:
                c_fail = next_c_fail
                n = max(n - 1, 2); offset = i; break
            elif test(next_c_pass) == PASS:
                c_pass = next_c_pass
                n = max(n - 1, 2); offset = i; break
            else:
                j = j + 1

        if j >= n:
            if n >= len(delta):
                return (delta, c_pass, c_fail)
            else:
                n = min(len(delta), n * 2)
```

最小化输入

增量调试（或科学方法其他形式的自动化）中很有意思的地方就是，它是一种非常普遍的方法。它不仅可以在一组代码改动中查找导致故障的起因，还可以在其他的搜索空间中查找导致故障的起因。例如，你可以将增量调试应用于在程序的输入中来查找导致故障的起因，2002年Ralf Hildebrandt和我就是这样做的。

在程序的输入中查找起因时，程序代码是相同的：既没有发生改动，也没有重新构造，只需执行程序即可，而程序的输入数据却发生了变化。我们来考虑这样一个程序，在大多数的输入数据上，程序都能正常运行，而在某个特殊的输入上将运行失败。程序员可以很容易地通过增量调试来分离出导致故障的两个输入数据之间的差异：“导致网页浏览器崩溃的原因是在第40行的<SELECT>标签”。

程序员还可以修改这个算法，使其返回最小输入：“要使网页浏览器崩溃，只需给它输入一个包含<SELECT>的网页即可。”在最小输入中，每个保留字符都与将要发生的故障相关。对于调试器来说，最小输入是非常有价值的，因为它们使事情变得简单：我们只需分析更短的程序执行和更少的状态。另外一个重要的（并且是漂亮的）作用就是，它们找出了故障的本质。

我曾经遇到过一些调试第三方数据库中bug的程序员。他们遇到一个非常复杂的、机器生成的SQL查询语句，这些语句有时会使数据库产生故障。生产厂商并没有把这些bug归类为高优先级的bug，因为“你们是惟一需要使用这种复杂查询的客户”。然后，程序员将一页大小的SQL查询简化为一行SQL语句，而执行这行语句仍然可以触发这个故障。在看到这个单行SQL查询语句后，生产厂商立刻将这个bug赋予最高的优先级并改好了。

如何获得最小输入？最简单的方式就是在调用`dd()`函数时传递一个空的`c_pass`，并且让一个成功的测试仅当输入为空的时候才返回`pass`，而在其他的情况下则返回`unresolved`。`c_pass`保持不变，而`c_fail`将在随着每次失败的测试而变得越来越小。

再次指出，要分离出这种故障起因，只需一个自动化的测试以及一种将输入拆分为更小部分的方法——也就是说，一个了解输入数据语法的一些基本知识的拆分函数。

查找缺陷

通常，增量调试还可以减少整个程序的代码，从而只保留相关的代码。假设你的网页浏览器在打印一个HTML页面时崩溃了。将增量调试应用于程序代码意味着只有能够重现

故障的代码才会被保留。这听起来是不是很奇妙？不幸的是，在实际中这很难实现。原因是程序代码的各个部分之间依赖性很高。如果移走一部分代码，那么整个程序都将被破坏。如果希望在随机移走一部分代码的同时还使程序保持有意义，那么这种机率是非常小的。因此，增量调试将肯定需要平方量级的测试数量。这就意味着，对于 1 000 行的程序来说，需要进行 1 百万次测试——等待数年的时间。我们无法等待这么长的时间，因此我们不去实现这种方式。

虽然如此，我们仍然希望不仅能够在程序输入或者代码改动中找出故障起因，还能在实际的源代码中找出故障起因——换句话说，我们希望找出导致程序失败的语句（当然，我们希望是自动地找出这些语句）。

事实证明，这个任务同样可以通过增量调试来完成。然而，我们并不是直接使用这种方法的，而是利用程序状态（Program State）这种变通的方法——程序状态就是程序所有的变量以及它们的值所构成的集合。在这个集合中，我们希望自动确定故障起因，例如“在调用 `shell_sort()` 函数中，变量的大小导致了程序故障。”而这又是如何实现的呢？

让我们来总结一下到目前为止所做的工作。我们在程序版本上应用了增量调试——其中一个版本运行正确，而另一个版本运行失败——并且分离出了导致故障的最小代码差异。我们还在程序输入上应用了增量调试——其中一个输入能使程序运行正确，而另一个输入则使程序运行失败——并且分离出了导致故障的最小输入差异。同理，如果我们将增量调试应用到程序状态上，那么就可以得到一个运行正确的程序状态，以及一个运行失败的程序状态，并且最终获得导致故障的最小程序状态差异。

现在，在这种方法中存在三个问题：第一个问题：如何获得程序状态？我对 `gdb` 调试器进行配置，首先查询所有命名变量，然后打开所有的数据结构。如果遇到了一个数组或者一个结构，我将查询它所包含的成员；如果遇到了一个指针，我将查询它所指向的变量，等等——直到我到达了需要进行修改的地方，或者获得了所有可访问的变量。程序状态用变量（图中的顶点）和引用（图中的边）的图来表示，如图 28-2 所示，其中抽象出具体的内存地址。

下一个问题：如何比较两个程序状态？这非常容易：目前已经有算法能够计算两张图中的共同子图——子图之外的所有部分都将成为程序状态的差异。在正确地实现了这种算法之后，我们就可以提取和确定两个程序状态之间的差异了。

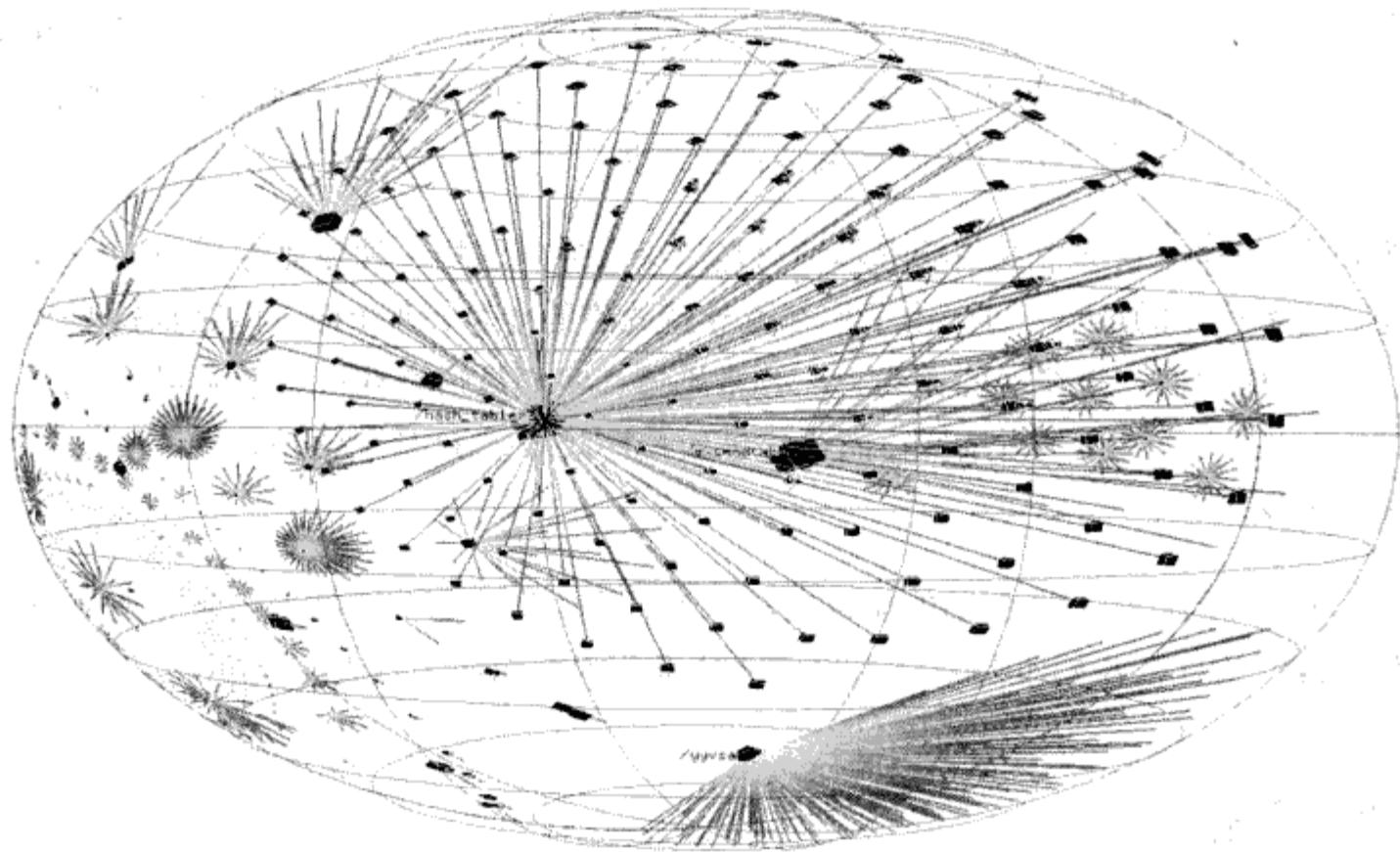


图 28-2：GNU 编译器的程序状态

第三个，也是最后一个问题：如果应用两个状态之间的差异？这确实是一个挑战，因为这不仅需要观察程序状态，还需要操纵程序状态。为了将差异应用到程序状态中，我们不仅需要把变量设置为新值，还需要复制整个复杂的数据结构，包括分配和删除元素。如果能实现这个操作，那么我们还可以做一些非常有趣的事情，例如在两个运行的进程之间任意地转移程序状态。这不仅可以是整个程序状态，还可以是部分程序状态——从对单个变量的修改到对符号表中半数变量的修改。

在程序执行的时候转移程序状态，这种思想需要一定的时间来接受。我记得我在IBM所作的第一次报告中解释了这个算法在程序状态上的应用，并且举出了示例：“目前在这两个状态之间有 879 个差异。我们现在通过增量调试来缩小搜索范围并定位故障起因。这个算法将应用一半的差异，也就是 439 个状态差异。这意味着在运行成功的实例中，将有 439 个变量被设置为运行失败实例中的值。”

此时，听众中有一个小伙子站起来说：“这听起来像是一件荒唐的事情？”

当然，他是对的。将一个运行实例中 439 个变量的值设置为另一个运行实例中的值并不会得出什么有意义的结果；设置另外的 440 个变量也不会有什么作用。但正是在这种情况下，增量调试为我们带来了应用更小改动的思想——也就是说，它将尝试 220 个变量，然后是 110 个变量等等。最终，增量调试分离出了导致故障的变量。“编译器的崩溃是由于在抽象语法树中的循环所导致的”。最终证明了这种方法是正确的——尤其是对于在 IBM 忙于开发（以及调试）编译器的人们。

这个证明使人们忘记了增量调试是一种非常奇怪的方法。不过，我在这个主题上发表的第一篇文章经过了很长时间才被接受。其中一位审稿人坦白地承认他对这个奇怪的方法感到非常震惊，他甚至没有去继续阅读试验结果。

虽然如此，在程序状态中找到故障起因只是解决最终问题的一种变通方法。Holger Cleve实现了这种技术的最后一步。由于他知道导致故障的变量，因此他可以跟踪这些变量的值，一直到使它们产生错误的语句——非常迅速！我们最后将获得导致故障的语句：“在第437行的语句产生了一个循环，而这将导致程序故障。”这是非常巧妙的——这篇文章很快就被发表了。

那么，既然我们已经有了自动的调试解决方案，那么为什么今天的人们仍然使用交互式的调试器？为什么我们不公开自动化调试技术并因此而成为百万富翁？

原型问题

这是实验室的研究成果和实际产品之间的区别。在我们的方法中，主要的问题就是这个方法是很脆弱的，可以说是非常脆弱。提取出精确的程序状态是一件非常棘手的工作。假设你正在分析一个在调试器中暂停执行的C程序。你遇到了一个指针。它是否指向某个变量？如果是的话，这个指针指向的变量的数据类型是什么？它指向多少个元素？在C语言中，所有剩下的问题都留给了程序员的判断力——要猜测目前程序所使用的内存管理机制是非常困难的。

另一个问题就是要确定程序状态在什么位置上结束以及系统状态从什么位置上开始。有些状态是在应用程序之间或者应用程序和系统之间被共享的。我们在提取和比较程序状态时，应该到什么地方为止？

对于实验室的试验来说，这些问题可以被很好地解决和限制，但对于成熟稳定的工业方法来说，我们发现这是难以处理的。这就是为什么今天人们仍然不得不使用交互式调试器的原因。

然而，这种方法的前景并不是黯淡的。目前已经出现了在程序输入上应用增量调试的命令工具。Eclipse的ddchange插件把增量调试带入到了你的日常工作中。研究人员把增量调试应用于方法调用，用最少的测试案例很漂亮地集成了bug的捕捉/重现。最后，在所有这些自动化方法中，我们能够进一步理解如何进行调试工作以及如何按照系统化的，有时甚至是以自动化的方式来进行调试——自动化的方式可能是最有效的方式，甚至还可以是漂亮的方式。

结论

如果你有时必须进行调试，那么你可以努力去减少调试过程中的痛苦。采用系统化的方式（遵循科学的方法）将起到很大的作用。而将科学方法自动化则是更加有用。如果遵循在本章中给出的建议，你将编写出漂亮的代码——并且，随之而来的另一个好处是，你还能实现最漂亮的调试。那么，什么是最漂亮的调试？当然，那就是不需要调试！

致谢

我要感谢和我一起为实现调试工具中的漂亮性而努力的学生。Martin Burger 为 AskIgor 做了很多的工作，并且为 Eclipse 实现了 ddchange 插件。Holger Cleve 研究并实现了导致故障的语句的自动分离。Ralf Hildebrandt 实现了导致故障的输入的自动分离。Karsten Lehmann 也参与了 AskIgor 的工作并且实现了在 Java 程序中分离出导致故障的程序状态。Dorothea Lütkehaus 编写了最初的 ddd。Thomas Zimmermann 实现了子图比较算法。Christian Lindig 和 Andrzej Wasylkowski 对本章的早期修订稿提出了有益的建议。

进一步阅读

我将系统化调试和自动化调试的经验编撰成了一门大学课程。在这门课程中，你可以学到更多关于科学方法和增量调试的知识——以及许多关于调试和分析的技术，例如统计调试、自动测试或者静态 bug 检测。所有的课程幻灯片和参考文献都在网页 <http://www.whypartofit.com> 上。

如果你想查看我们小组发表的文章，请访问增量调试的主页 <http://www.st.cs.uni-sb.de/dd>。

最后，在网上以 delta debugging 为关键字来搜索将获得大量的资源，包括更多的发表文章和实现。

第 29 章

代码如散文

Yukihiro Matsumoto

程序和散文有着一些共性。对于散文来说，读者要问的最重要问题是，“这篇散文的内容是什么？”对于程序来说，要问的最重要问题是，“程序的功能什么？”事实上，散文和程序的意图都应该是非常清楚，而不至于让读者提出这两个问题。不过，对于散文和程序来说，很重要的一点就是看看它们是如何写成的。即使要表达的思想本身是很好的，但如果散文或者程序难以理解，那么要把这个思想传播给读者将会很困难。因此，写作的风格和写作的意图同等重要。散文和程序的首要问题就是要让人们能够阅读和理解（注 1）。

你或许会问：“可不可以认为计算机程序是由人来阅读的？”这个假设是人们通过程序来告诉计算机去执行什么任务，然后计算机将用编译器或者解释器来编译和理解代码。在这个流程的最后，程序被转换为通常只有 CPU 才能阅读的机器语言。当然，这是程序的工作方式，但这种解释只是描述了计算机程序的一个方面。

大多数程序都不会只编写一次。程序在其生命周期内总是被不断地重新整理和改写。程序中的 bug 必须被消除。需求变更以及增加功能的需要意味着程序本身可以在目前的基

注 1：本章的英文内容是由 Nevin Thompson 从日语原文中翻译过来的。

基础上进行修改。在这个过程中，人们必须能够阅读和理解最初的代码；因此，人理解代码要远比计算机理解代码重要。

当然，计算机可以毫无怨言地处理程序的复杂性，但这对于人来说却并非如此。难以理解的代码将会极大地降低大多数人的生产效率。而易于理解的代码则会提高生产效率。并且我们将在这样的代码中看到漂亮性。

是什么因素使得计算机程序是可读的？换句话说，什么是漂亮的代码？虽然对于漂亮的程序是什么，不同人有着不同的标准，但是判断计算机代码的可读属性并不只是简单的美学问题。相反，计算机程序是根据它们在执行预期任务时的好坏程度来判断的。换句话说，“漂亮的代码”并不是独立于程序员的努力而存在的抽象优点。漂亮代码的真正含义是帮助程序员感到快乐和提高生产效率。这就是我用来评价程序漂亮性的标准。

简洁性是有助于实现漂亮代码的因素之一。正如 Paul Graham 所说的，“简洁性就是强大的功能”。在编程术语的词典中，简洁性就是一种优点。因为在人眼扫描代码的时候肯定会付出一定的代价，因此在理想的情况下，程序中不应该包含无用的信息。

例如，当不需要类型声明或者在设计中不需要类的声明和 main 函数的定义时，简洁性就要求尽可能地避免编写这些声明。为了说明这个原则，在示例 29-1 中分别给出了用 Java 和 Ruby 编写的 Hello World 程序。

示例 29-1：分别用 Java 和 Ruby 编写的“Hello World”程序

Java	Ruby
<pre>class Sample { public static void main(String[] argv) { System.out.println("Hello World"); } }</pre>	<pre>print "Hello World\n"</pre>

这两个程序都正确地完成相同任务——仅仅显示“Hello World”这两个单词——但是 Java 和 Ruby 的实现方式有着本质的不同，即在 Ruby 编写的程序中，所有需要的工作就是描述任务的本质。输出“Hello World”。在程序中既没有声明语句，也没有数据类型。而在 Java 中则需要包含多个与我们的意图并不紧密相关的描述。当然，用 Java 实现的程序有着自身的优点。然而，由于在 Java 程序中无法省略任何东西，因此就丢失了简洁性（离题一下，用 Ruby 编写的 Hello World 能够在三种语言中使用：这个程序还可以运行在 Perl 和 Python 中）。

简洁性还意味着消除冗余。冗余被定义为重复的信息。当信息重复时，维护代码一致性

的代价就会变得很高。由于大量的时间被花在维护一致性上，因此冗余性会降低编程的生产效率。

虽然有人会认为冗余在解释含义时会降低开销，但事实却刚好相反，因为在冗余的代码中包含了大量多余的信息。冗余信息导致的后果之一就是冗余的方法依赖于支持工具的使用。虽然最近流行通过IDE来编写程序，但这些工具并不会有助于解释含义。开发优雅代码的真正捷径在于选择一种优雅的编程语言。像Ruby和其他类似的轻量级语言就支持这种方法。

为了消除冗余性，我们要遵循DRY原则：不要重复代码。如果相同的代码存在于多个地方，那么你想要表达的东西将会变得晦涩。

DRY的概念是与拷贝/复制的概念相对的。在过去，有些公司通过程序员编写的代码行数来评价生产率，因此事实上在无形中鼓励了冗余性。我曾经听说尽可能多地复制代码有时候被认为是一种优点。但这种观点是错误的。

我相信，真正的优点是在于简洁性。最近Ruby on Rails的流行正是出于对简洁性和DRY的不断追求而推动的。Ruby语言很认真地对待“不要把相同的东西编写两次”和“使得描述简洁”。Rails从Ruby语言中继承了这一思想。

漂亮代码的一个颇有争议的方面，这就是它的熟悉性(familiarity)。人类比你所想像的要更为保守，大多数人都很难接受新的概念或者改变他们的思考方式。而与此相反的是，大多数人宁愿痛苦地继续下去也不愿意进行改变。大多都不愿意替换掉熟悉的工具或者学习一门新的语言，即使没有任何有说服力的理由。只要有可能，人们就会把他们要学习的新流程与他们以前所理解的常识进行比较，然后得出对新流程不当的负面评价。

为了改变一个人的思考方式所付出的代价要远远超出你所能想像的。为了更好地在完全不同的概念之间进行转换(例如，从过程式编程转换到逻辑式编程或者功能式编程)，必须熟悉大量的概念，而陡峭的学习曲线将会使人们的大脑产生痛苦。因此，这些转变降低了程序员的生产效率。

根据这个观点，由于Ruby支持“漂亮代码”的概念，因此它是一种极其保守的编程语言。虽然Ruby被称为纯粹的面向对象语言，但Ruby并不像SmallTalk那样使用基于对象消息传递的全新控制结构。在Ruby中沿用了程序员熟悉的传统结构，例如if、while等。它甚至从老的Algol族语言中继承了end关键字来结束代码块。

与其他的同时代编程语言相比，Ruby看上去确实有些老土。但很重要的一点是，“永远不要过于创新”同样是编写漂亮代码的关键要素之一。

简单性是漂亮代码的另一个元素。我们经常会在简单的代码中感受到漂亮性。如果一个程序很难理解，那么我们就不能认为这个程序是漂亮的。而且，当程序变得晦涩且无法理解时，所带来的结果往往是大量的 bug、错误和混淆。

简单性是在编程中最容易被误解的概念之一。设计编程语言的人们经常希望使这些语言保持简单性和整洁性。虽然初衷是高尚的，但这么做将使得用这种语言编写的程序变得更加复杂。在 IBM 设计 Rexx 脚本语言的 Mike Cowlishaw 曾经指出，因为语言的使用者比语言的实现者要更为普遍，因此后者的需求一定要让步于前者的需求。

一般的规则是，虽然有上百万的人需要使用某种语言，但很少有人需要为这种语言实现解释器或者编译器。因此，对语言的优化应该优先考虑这上百万人的需求，而不是其他人的。正因为如此，编译器的编写者不喜欢我，因为 Rexx 变成了一种难以解释或者编译的语言，但我认为这给一般的人带来了好处，当然就是一般的程序员。

我由衷地同意这一点。Ruby 很好地体现了这种思想，不仅 Ruby 语法是非常简单的，它还支持对编程问题的简单解决方案。因为 Ruby 语言在设计上并不简单，所有使用 Ruby 的程序可获得简单性。对于其他的轻量级语言来说，这同样是正确的；从易于实现的意义上来看，这些语言并不是轻量级的，但由于它们的目的是减轻程序员的工作负担，因此这些语言被称为轻量级语言。

要想看看这在实际中意味着什么，我们就需要来考虑 Rake，这是一个像 *make* 一样的构建工具，它被 Ruby 程序员广泛地使用。与使用单一文件格式的 *Makefile* 不同，*Rakefile* 是用 Ruby 编写的，作为一种有着全功能编程能力的领域特定语言（Domain Specific Language, DSL）。在示例 29-2 中给出了一个运行一组测试的 *Rakefile* 文件。

示例 29-2：示例 *Rakefile*

```
task :default => [:test]
task :test do
  ruby "test/unittest.rb"
end
```

Rakefile 利用了在 Ruby 语法中的以下简化形式：

- 方法参数中的圆括号可以被去掉。
- 在方法的后面可以出现自由的哈希键/值。
- 代码块可以附加在方法调用的末尾。

在使用 Ruby 编程时，你可以不使用这些语法元素，因此在理论上它们是冗余的。人们经常批评这些语法元素会使语言变得复杂。然而，在示例 29-3 中给出了不使用这些语法元素写出的示例 29-2。

示例 29-3：不使用缩写语法的 Rakefile

```
task(:default => [:test])
task(:test, &lambda( ){
  ruby "test/unittest.rb"
})
```

你可以看到，如果从 Ruby 语法中去掉了冗余性，或许 Ruby 语言会变得更为优美，但程序员将不得不做更多的工作，并且他们的程序将会更加难以阅读。因此，当使用更简单的工具来解决复杂的问题时，复杂性仅仅是被转移到了程序员一方，这可真是本末倒置。

在“漂亮的代码”中，另一个重要的因素就是灵活性。在这里我将灵活性定义为无需借助工具的增强。当程序员为了工具而被迫做一些违背他们本意的事情时，导致的结果将是压力。这种压力将会给程序员带来负面影响。最终结果就是，根据我们对代码漂亮性的定义，程序员将远离快乐，而代码也将远离漂亮。人力资源比工具或者语言更有价值。计算机应该服务于程序员并最大程度地提升他们的生产效率和快乐指数，但在现实中，它们却经常是增加负担而不是减轻负担。

平衡性是漂亮代码的最后一个因素。到目前为止，我已经讨论了简洁性、保守性、简单性和灵活性。单凭任何一个因素都无法确保产生漂亮的程序。如果在每个因素之间进行平衡，并且从一开始就记住这点，那么所有的因素就能够很融洽在一起，从而创建漂亮的代码。如果你确信能够在编写和阅读代码中找到乐趣，那么你将体会到作为一个程序员的快乐。

编程愉快！

当你与世界的联系 只有一个按钮时

Arun Mehta

“Stephen Hawking (斯蒂芬·霍金) 教授只能按一个按钮”，在给我们的需求说明中只有这句话。

Hawking教授是杰出的理论物理学家，但他不幸患有ALS (肌肉萎缩性侧面硬化病)。这种疾病将“使控制自主肌肉运动的中枢神经系统中的神经细胞逐渐恶化，而将导致全身肌肉的衰弱和萎缩”（注 1）。他只能通过 Equalizer 软件用一个按钮来写字和说话。Equalizer 使用了一种外部的盒子来进行从文本到发音的转换，而现在这种盒子已经不再生产了，并且 Equalizer 的源代码也丢失了。

为了在老化的硬件发生故障时也能与外界交流，他联系了一些软件公司，希望他们能编写出某种软件，使得那些有着高度运动神经残疾的人们能够通过这种软件来访问电脑。Radiophony，也就是 Vickram Crishna 和我最初创建的公司，很高兴地接受了这个挑战。我们把这款软件叫做 eLocutor (注 2)，并且决定让它成为免费的和开源的软件，这样在 Equalizer 中出现的问题就永远不会重现了。

注 1： http://en.wikipedia.org/wiki/Amyotrophic_lateral_sclerosis.

注 2： 可从网址 <http://holisticit.com/eLocutor/elocutorv3.htm> 下载.

这种软件在残疾用户日常生活中的重要性不言而喻。Hawking教授自己就是一个最好的示例。他不仅成为了杰出的科学家之一，也是一位非常成功的作家和激励者，这都是因为他能通过软件来写字和说话。谁知道有多少天才还没有被我们发现，而这或许只是因为某个孩子无法很清楚地说话或者写字以获得老师的理解。

Hawking教授仍然在使用Equalizer软件，他熟悉这个软件已经几十年了。不过，eLocutor也被证明了对于各种残疾用户来说是非常有用的，这是因为用户可以很容易地对eLocutor进行定制以满足他们各自的需求。

我们的第一个问题，也就是我们向每位工程师解释的这个问题，是：我们能否找到一种方式来提高Hawking教授的输入数量？但他的助理很坚决地回答到：Equalizer只用一个按钮就可以工作了，因此他们认为没有理由去改变这一设计。我们也见过在为高度残疾用户编写的软件中使用的各种技巧，在这些软件中有许多二元开关，即使高度残疾用户也可以按下这些开关，并且可以通过肩膀、眉毛、舌头，甚至大脑来直接控制（注3）。在设计了一个绝大多数用户都能够使用的解决方案后，我们就可以来看看如何更灵活地提高人们的输入速度。

我们还发现了一个能够将eLocutor应用到更广泛人群的特殊市场。对于移动电话来说，那些只需一个按钮来操作的软件是非常方便的，例如：免提设备通常只有一个按钮。在有了从文本到语音的转换之后，人们就能够消除对界面的依赖，因此eLocutor还可以由汽车驾驶员来操作。还有另一种场合，想象一下，当你和一位客户正在谈话，你的眼睛无需从她身上转移开来，就可以通过google搜索一个她刚才谈到的名字并且把搜索结果悄悄地传到你的耳朵中。

当然，对于软件编写者来说，设计只能通过一个按钮来进行有效操作的软件是一项非常有趣的技术挑战。首先，我们必须为eLocutor挑选一组基本的执行功能。我们所选择的功能包括文件检索和存储、打字、删除、说话、滚屏以及搜索。

接下来，我们必须找出某种方式，使得只需通过一个按钮就可以执行所有这些功能。这是设计中最有趣的部分，因为程序员并不是经常能有机会设计一些基本的交互模式。这也是本章要讲述的主要内容。

基本的设计模型

无须多言，这个软件必须是高效的，这样用户才可以很快地打字而无需频繁地点击。有

注3：例如，可参见 <http://www.brainfingers.com/>。

时候，Hawking 教授需要几分钟来打出一个单词，因此对于一个忙人来说，对编辑速度的每个改进都是很有用的。

这个软件还需要是易于定制的。用户词汇的内容和容量可能是非常多样的。这个软件需要能够适应这些特性。此外，我们还要确保残疾用户可以无需他人的帮助也能自行修改尽可能多的设定和配置。

由于我们手头只有很少的需求说明，而且没有编写这种软件的经验，因此我们预计随着理解的不断深入，可能需要对设计做几次重大的修改。在牢记了所有这些需求后，我们决定用 Visual Basic 6 来编写软件，这是一个优秀的快速原型工具，有很多现成的控件。用 VB 可以很容易地构建一个图形用户界面，并且提供了很方便的功能来访问数据库。

这个问题的独特之处在于数据流中的高度不对称性。一位视觉相当好的用户或许能够接收大量的信息。然而，对于有着高度运动残疾的用户来说，在另一个方向上的数据流动是非常少的：只是偶尔会有一点数据。

这个软件把选项一个个地显示给用户，而用户则可以在想要的选项出现时，通过点击来选取。当然，这里的问题在于任意时刻都有着非常多的选项。用户可能想要输入多个字符中的任意一个，或者保存、滚动、查找以及删除文本。如果 eLocutor 将所有这些选项循环显示，那么将会花太长的时间，因此 eLocutor 把这些选项分成组和子组，并且把这些组构成一个树型结构。

为了加快打字速度，eLocutor 将进行预测，它提供了几种方式来帮助完成将要输入的单词，并且给出下一个单词和词组剩余部分的选择。用户需要清楚这些预测，这样就可以在可能的情况下获得一个快捷输入的机会。

因此，我们决定创建一个可视的界面，在这个界面中，控件可以根据我们认为用户希望看到的显示效果来动态地调整大小甚至隐藏，从而提供帮助用户加速输入句子的快捷方式。因此，当用户输入时，在 eLocutor 界面中将包含一个带有提示的窗口，用来告诉用户如何完成当前的单词，此外还有一个窗口来帮助用户选择接下来的单词（标点符号也被视为单词）。如果用户正在输入的句子与数据库中的某些句子匹配，那么这些句子也将被显示出来。在图 30-1 中给出了典型的 eLocutor 界面。

有时候会出现太多的选项而无法把它们都放在一个小窗口中。我们设计了一个扫描功能来帮助用户在这些选项之间快速选择。首先打开一个大窗口，并在这个窗口中显示所有的选项，然后将这些选项分成小组显示在一个小窗口中。当某个单词出现在大窗口时，这就告诉用户 eLocutor 可以提供快捷方式来输入这个单词。现在他就可以等待，当这个单词出现在小窗口中时，用户就点击按钮。此时大窗口将消失，而在小窗口中的选项，大概有十几个，就可以按照相同的方式通过树型结构提供给用户使用。



图 30-1: eLocutor 界面

当用户停止输入并且开始滚动文本时，界面的有效区域将被重新划区，此时界面将在插入位置的前后显示尽可能多的文本。

我们在预测方式上需要尽可能地实现智能化，这是为了最大限度利用残疾用户努力产生的按钮点击。我们在软件中内置的智能分为三类：

关系数据库

当用户输入一个单词的开头几个字符时，在字典表中将进行搜索并给出如何完成这个单词的提示。对于用户之前输入文本进行分析同样能够提示用户接下来会选择什么单词。

缓存

这种方法利用了用户行为中的模式。我们不仅缓存了经常使用的单词，而且还缓存了文件名、查找文本、口语文本以及决策路径，这样用户可以很容易地重新生成一系列的步骤。

专门分组

这种智能方法利用了单词的自然分组，例如城市的名字、食物术语、语音部分等等。这些分组可以使用户用相似的单词代替常用词组中的单词，从而在原有句子的基础上

上快速构造出新的句子。例如，如果在数据库中有 Please bring me some salt 这个句子，那么只需少量的点击操作就可以构造出 Please take her some sugar。

在树型结构中包含了所有的可用选项，这种结构类似于菜单的层次结构。在树型结构中的这些选项将逐个地被高亮显示，并且按照固定的速度循环。树型结构还可以很自然地扩展到选项的子集，例如刚才描述的单词特殊分组。

我们需要介绍一下图30-1界面中的不同控件。用户编辑文本的活跃部分显示在中间的文本框中，而在文本框上方和下方的内容将根据用户当前操作的不同而不同。在右下方是软件对接下来将要输入文本的预测。

在右上角（用户界面中的红色文本）的文本是替换当前单词的提示，如果你已经输入某个单词的一些字符并且希望eLocutor来预测单词的剩余部分，那么这是非常有用的。在右下方的黑色字体，是在输入一个单词后对下一个单词的提示。标点符号也被视作为单词，并且由于在上一个单词中同时包含了字母和数字，因此下一个将是标点符号，正如图 30-1 中右边显示的那样。

当用户正在输入的句子与一个已经输入的句子类似时，那么在界面底部的提示将会派上用场。不过，用户的注意力通常是集中在左边的树型控件上，因为她只能在树型控件中利用当前的所有信息来影响中间文本框的文本。

在树型控件的下面，用户可以看到可以有多种不同的选择，以及一些在后面将会讨论到的其他有用信息。

这个界面将在树型控件中有序地移动。当用户想要选择的选项被高亮显示时，用户可以在此时点击按钮。在界面的不同窗口中将显示用于预测下一个单词、单词完成、词组完成等的选项。为了使用这些选项，用户必须不断地跳转，直到相应的选项出现在树型控件中。

输入界面

由于只需要一个二元输入，因此我们选择了使用鼠标右键来实现。这使得各种按钮都可以很容易地连接到eLocutor。打开鼠标并且把适当的按钮焊接在鼠标右键位置上，任何一个电工或者电器爱好者都能够完成这种焊接工作。

在图 30-2 中给出了我们如何为 Hawking 的特殊开关制作一个临时连接：从鼠标中取出左下方的电路板，并且在连接鼠标右键的位置上焊接外部开关。

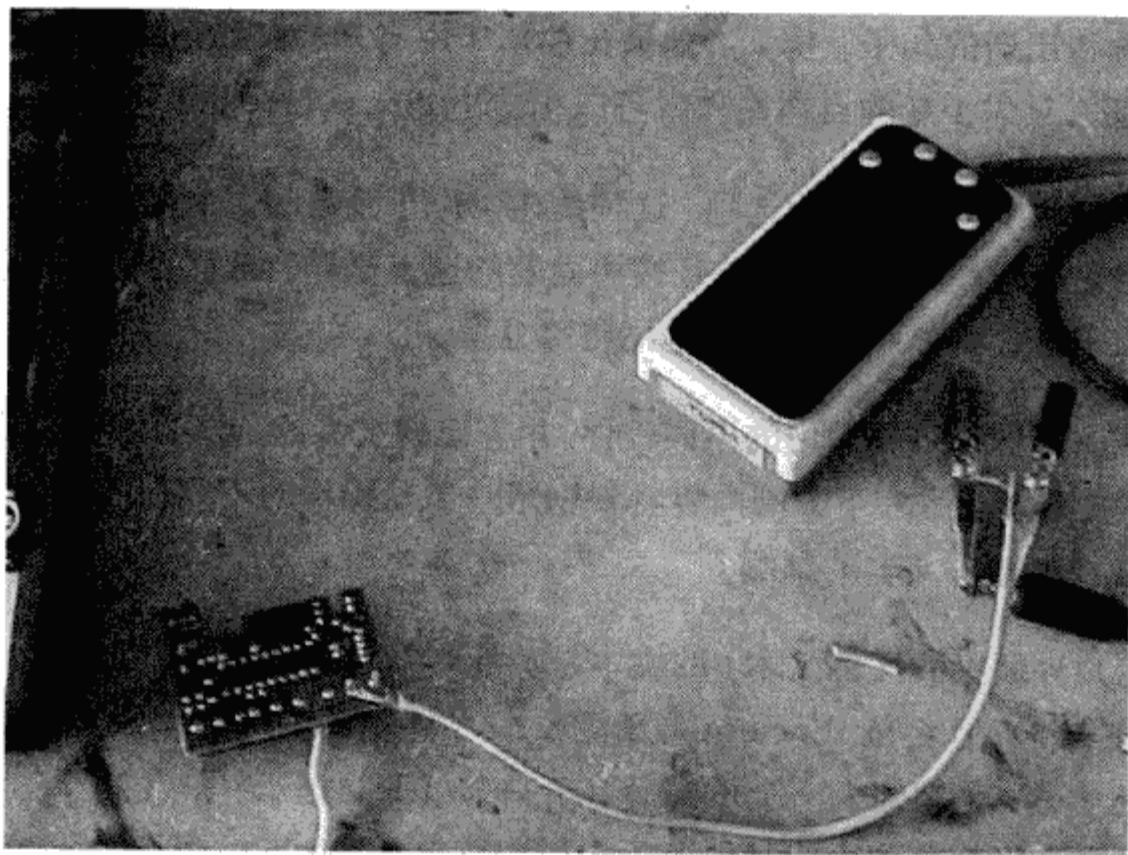


图 30-2：把 Hawking 教授的开关并行地连接到鼠标右键

树型控件

如果在软件中只提供惟一一个二元输入，那么在用户图形界面中显然存在这样的设计：所有的选项都是以二叉树的形式依次显示。在高亮显示每个节点时，如果用户在固定的时间内点击了按钮，那么界面将会选中这个选项，而接下来的动作可能是以子树的形式展开下一级的选项。如果用户没有点击按钮，那么软件将自动地高亮显示下一个同级节点并再次等待用户点击。

我使用了 Visual Basic 中的树型控件（TreeView）（注 4）来实现这种树型结构。树型控件能够从左到右逐级展开，这种方式看上去很不错。对于任何节点来说，如果用户在固定的时间间隔——由定时器（Timer）控件来设置——内点击了按钮，那么将展开这个节点并且跳转到子节点（也就是说，高亮显示将转移到当前节点右边的子节点上），如果这个节点是叶节点，那么将会执行一些动作。如果你不点击按钮，那么 eLocutor 将会把焦点转移到与这个节点同级的下一个节点。如果到了最后一个节点时还没有点击，那么 eLocutor 将再次从第一个节点重新开始。

我们对树型结构进行了扩充，在树型结构的每一级上都定义了一个叫作 Up 的节点，如果选中了这个节点，那么高亮显示将会转移到父节点，也就是更靠近根节点的一级节点。

注 4：<http://www.virtualsplat.com/tips/visual-basic-treeview-control.asp>.

顶级节点分别是 Type、Scroll、Edit（主要的编辑功能）和 Commands（杂项）。Type 子树中的叶节点将把文本送入到输入缓冲区中。Edit 子树中的叶节点将用来从缓冲区中删除文本或者复制文本，而 Scroll 中的叶节点将控制在缓冲区之间文本的移动。

eLocutor 的智能表现为能够动态地生成树型结构中的内容，这样你就可以很快地找到下一个你想要采取的动作：eLocutor 通过几种不同的方式来学习你的行为，从而更好地进行预测。

在二元输入中最大的问题就是跳转。如果你正在输入字符，而此时需要在句子的开头删除一些文本，那么就需要花很长时间来多次选择 Up 节点以到达根节点，然后再到选择 Scroll 节点以滚动到合适的删除位置，接着再选择几次 Up 节点到根节点以及选择 Edit 节点来删除，最后再滚动到文本末尾并重新开始输入。不过，令我们感到欣慰的是，我们找到解决这种困境的方案。

长按

在观察 Hawking 教授使用 Equalizer 时，我发现了一种新的操作模式：除了简单地点击按钮之外，他还可以长时间地按着按钮不放，并且在特定的时候再放开。事实上，这个按钮不仅仅只是一个二元输入设备，它实际上是一个模拟设备，因为它能够提供一种可变的持续时间信号。我们花很长的时间来思考如何最大限度地利用这个新发现的功能：现在，我们可以从点击中得到更多的信息，而不是只用一个比特位就能表示。例如，我们可以让用户在一组选项中进行选择。短促的点击可以用来表示默认行为，而长按则可以打开许多其他的选项。

很明显，我们希望通过这个新发现的功能来执行一些额外操作以实现快速跳转。我们还非常高兴能够在树型控件中的高亮文本上执行不同的操作，例如输入文本、把文本复制到过滤器等等。如果没有长按的功能，那么在每个叶节点上只能有一个行为，而现在我们可以在高亮的节点上为用户提供其他的选项，在某些情况下甚至不需要使用叶节点。

这组新增的选项不应该过多，否则将要求用户长按的时间较长。因此。我们希望这些选项能够根据树型控件的不同位置而发生变化。例如，当位于 Scroll 子树时，“Type this” 选项就没有意义，而当位于 Speller 子树时，这个选项将非常方便。

我们最终设计出来的是一个简单的并且易于理解的操作模式。在一个节点上点击按钮将执行默认行为。但如果长按按钮时，将会打开一个独立菜单，并且菜单的选项将依次滚动，当滚动到你想要选择的菜单项时，便可以通过放开按钮来选择。这种用法有些类似于在 Microsoft Windows 下通过点击鼠标右键来显示用户上下文菜单选项的方式。在菜单中通常包括跳转到根节点和逆向移动等选项。

这种操作模式的重要性不言而喻：它不仅极大地提高了输入文本和修改文本的速度，还为开发者提供了非常大的灵活性。

我们需要一个优雅的解决方案来使得长按菜单是上下文敏感的，因为如果要为二叉树中每个节点都单独创建一个长按菜单的话，那无疑都是非常麻烦的。和树型结构一样，长按菜单也被保存为文本文件的形式，并且可以在eLocutor中进行编辑。在选择长按菜单时，eLocutor将首先查看哪个节点被高亮显示。如果某个文本文件的名字与长按节点的名字一样，那么这个文件将被用作为菜单。如果没有相同名字的文件，eLocutor将用上一级节点的名字来查找，依此类推。

这样，每个子树都有其自己的长按菜单，并且用户可以完全控制。这种设计方式的另一种定义是，除非在子节点中改写了在其父节点中定义的长按菜单，否则子节点将自动继承父节点的菜单。

在示例 30-1 中给出了实现长按菜单的部分代码。OpenLongClickFile 函数用来查找并且打开与子节点参数相同名字的文件，并且如果没有找到文件，那么将递归地使用父节点的名字来进行查找。每当长按定时器被触发时，在文本框 *tblongclick* 中将显示这个文件的一行新内容。根据按钮保持按下状态的时间，长按定时器将重复地触发。当每个定时器都被触发时，示例 30-1 中的代码将判断并且设置布尔变量 *ThisIsALongClick*，然后将执行在每次长按中只需运行一次的代码，从而选择并打开相应的长按文件以读取。

在长按定时器每次被触发时，重复进行的操作将从文件中读取一行并且显示在 *tblongclick* 文本框中。当文件到达末尾时，这个将被关闭然后重新打开，接着再读入第一行。当按钮被放开时，变量 *ThisIsALongClick* 将被复位。

示例 30-1：实现上下文敏感的长按菜单选择

```
Private Sub longclick_Timer()
Dim st As String
Dim filenum As Long
If Not ThisIsALongClick Then
    ThisIsALongClick = True
    If MenuTree.SelectedItem.Text = stStart Then
        ' 已经位于根节点
        OpenLongClickFile MenuTree.SelectedItem
    Else
        OpenLongClickFile MenuTree.SelectedItem.Parent
    End If
End If
If EOF(longclickfilenum) Then
    ' 已遍历完这组选项，然后通过重新打开文件以跳转到第一个选项
    Close #longclickfilenum
    Open stlongclickfilename For Input As #longclickfilenum
```

```
End If  
Line Input #longclickfilenum, st  
tblongclick = st  
End Sub
```

长按菜单中的有效命令包括

>Start

跳转到树型控件的根节点（符号“>”表示“到达”）。

Upwards

将光标在树型控件中往上移动，直到松开鼠标右键。这在当想要选择的菜单项被高亮显示而你并没有按下按钮时很有用——也就是说，当你错过了一次选择机会时。

Type This

将树型控件中的高亮文本输入到中间的文本框中。这命令只有在Type子树下才是有效的。

Set Filter

将树型控件中的高亮文本复制到过滤器中；这在搜索文本时很有用。同样，只有当高亮文本处于Type子树下时，才能作为长按菜单的有效选项。

Words Up, Words Down

用于在输入时的快速滚动，在后面将进行介绍。

Pause

当命令需要被重复执行时将非常有用。当用户按下按钮并产生长按事件时，菜单将停止滚动，并且此时有一个菜单项被高亮显示。选择长按菜单中的Pause选项将维持这种暂停的状态。现在，每当用户点击时，菜单中高亮显示的命令都将被执行。要跳出暂停状态，必须再次产生一个长按事件。

Help

打开并播放一个上下文敏感的.avi视频文件，这个文件将给出树型控件能够提供给用户的选项。在Help子目录中包含了一组.avi文件，其中必须包含一个Start.avi文件。当选择了长按菜单中的Help选项时，将根据用户在树型控件中的当前位置来播放相应的.avi文件。

查找播放文件的方式类似于查找长按菜单文件的方式。这个软件首先在C:\eLocutor的子目录helpvideos下用高亮节点的名字来查找.avi文件。如果找到了这个文件，那么将播放；否则，eLocutor将用高亮节点的父节点名字来查找.avi文件。如果在helpvideos目录下面没有找到这个名字的.avi文件，那么eLocutor将递归地在树型控件中攀升，直到最终找到与某个节点对应的帮助视频。这个功能使得我们在最开始只需创建一般性的

视频，并且在随后再陆续增加更多的详细视频，而用户只需把这些视频复制到 *helpvideos* 子目录就可以在 eLocutor 中播放了。

在 <http://www.holisticit.com/eLocutor/helpvideos.zip> 上给出了一些帮助视频。由于软件的一些动态特性，观看一些视频将有助于读者更快和更全面地理解本章的内容。

树型控件的动态重构

树型控件中的内容以文本文件形式保存在磁盘上。这种方法的最大好处就是用户可以在 eLocutor 中编辑这些文件。换句话说，这种形式使我们能够很容易地满足设计需求之一：通过使数据结构透明并且易于编辑来使得用户能够根据自己的需要调整 eLocutor。

由于 eLocutor 将预测你的下一个动作，因此二元树应该是动态的；像“Next Word”这样的子树经常会被重构。每个节点内容文件的名字与节点的名字都是相同的（文件的扩展名为.txt），并且在文件中包含了该节点的一组直接子节点的名字。如果某个子节点名字以.txt 结尾，这表示的是某个子树的根节点，并且可以在相应文件中找到这个子树的子节点的名字。例如，根节点内容的文件被命名为 *Start.txt*，其中包含了 *type.txt*、*edit.txt*、*scroll.txt* 和 *commands.txt*，每行都对应于一组选项，用来显示在前面章节“树型控件”中描述的菜单。

如果节点名字不以.txt 结尾，那么就表示这是一个叶节点。选择这个节点将会发生一些动作。例如，如果叶节点是位于 Type 子树中，那么选择这个节点将使得相应的文本被输入到缓冲区中。

我们使用前缀 ^ 来标识出被动态重构的节点。例如，在下面的列表中给出了 *type.txt* 的内容，这些内容将构成在图 30-1 的树型控件中显示的子节点：

```
commonwords.txt
speller
^word completion.txt
^next word.txt
suffixes.txt
^justsaid.txt
^clipboard.txt
^phrase completion.txt
^templates.txt
vocabularytree.txt
```

如果在子树名字中带有前缀 ^，那么只有当用户点击相应的子树根节点时，才会开始构造子树的内容。

在 Visual Basic 的树型控件中，有一个索引功能可以用来加快检索。这个功能使我们在

创建树型控件的节点时，用单词作为节点名字，并且对单词进行分组，这样子树中的同级节点就可以在一个句子中相互替换而不会使句子产生不合理的含义。例如，如果在一个句子中包含 London 这个单词，那么我们可以很容易地用 Boston 来替换 London 并把这个新的句子用在其他的语境中。

按照这种形式来使用索引使我们能够实现 eLocutor 的两个关键功能：替换（Replace）和模板（Template），稍后我们将会讨论到。然而，不利的因素在于我们必须接受树型控件在索引功能上的局限性，即不允许重复的键。虽然我们无法避免把多个相同名字的节点插入到树型控件中，但只有其中一个节点可以被索引。

Type 节点中的 vocabulary 子节点是一个大型子树的根节点，这个子树把能够在句子中进行有意义替换的单词分成一组。为了实现替换和模板的功能，需要对这些单词建立索引。然而，在树型控件的其他地方也可以显示相同的单词，或者是作为单词完成的提示，或者是作为下一个单词的提示。在这些情况下将不能被索引。为了保持简单性，我们决定不对动态重构子树的内容进行索引。

Speller 节点是一个特殊的情况。这个节点的内容并不是动态的。然而，在这个节点中除了包含在 vocabulary 子树中的每个单词外，还包含了大量的其他叶节点，这意味着它也不能被索引。这个节点只是在需要的时候才被构造——也就是说，在 speller 子树中的子节点只有在被选择的时候才会被创建。

简单的输入

在 Type 子树中包含了三个可以帮助你进行一般性输入的节点。在 Speller 下面显示了所有从 a 到 z 的字母，这样你就可以为想要输入的单词选择第一个字母。然后，你可以对下一个字母做出类似的选择，不过已经输入的字母组合必须是字典中某个单词的起始部分。这样，你可以依次选择每个字母，直到最终完成整个单词。此时，当前的节点可能是一个叶节点，也可能不是。如果是叶节点，那么就可以通过点击按钮来输入刚才生成的单词。但在通常情况下并不是叶节点。

Vocabularytree 和 commonwords 这两个节点也能够使你轻松的完成输入，我们将在后面进行讨论。不过，如果系统的预测功能运行得很好，那么在输入一个类似于数据库中的句子时将进行预测，这样你就可以不需要这些工具。

预测：单词完成和下一个单词

在预测数据库中包含了几张表。在其中一张表中包含了用来构成 Word Completion 子树

的大约 250 000 个单词。用户可以输入某个单词的一个或者多个起始字母，然后通过这张表来完成这个单词的剩余部分，系统给出的提示将被显示在界面的右上方，如图 30-1 所示。用户可以通过 Speller 子树来使用这张表。

假定你希望立刻输入某个单词。例如，由于 *instant* 这个单词是 *instantaneous* 的开头部分，因此它并不是一个叶节点。要输入 *instant*，你可以依次选择这 7 个字母，然后当 *instant* 被高亮显示时，你可以通过长按来引发 Type This 命令。

在另一张表中包含有 *word1*、*word2* 和 *frequency* 等域。在构造这张表时，我们把大量的句子输入到商用软件 *dbmanager* 中，这个软件将统计每个单词跟在其他单词后面的频率。当你输入了一个单词之后，我们将查询这张表并将并且构造 Next Word 子树，这样就给用户提供了一组可能出现在当前单词之后的单词。

用户通过 eLocutor 输入的每个句子都将被复制到 *mailto:mehthaatvsnldotcom.txt* 这个文件中。之所以取这样一个文件名，是为了鼓励用户把通过 eLocutor 生成的文本示例发送给我，这样我就可以从中得到一些提高 eLocutor 运行效率的想法。我们建议用户在把句子输入到 *dbmanager* 之前，首先要编辑这个文件并且去掉其中不合适的句子，这样，随着时间的推移，预测功能将趋于完善。如果软件编写者希望实现一种更好的单词预测方法，她所做的任务就是修改在 Access 数据库中的查询，而并不需要分析 eLocutor 的源代码。

我们在一张单独的表中列出了提供给数据库的文本标点符号，它们在 eLocutor 中也或多或少地被视作单词。

如果没有语义知识，那么对于软件来说就很难预测用户接下来想要输入什么内容。我们曾经和语言学家们探讨过是否存在某种很简单的方式来进行这样的预测，但很快就放弃了。我们实际做的工作是把单词手工组合成语义分组，并且放在 Vocabulary tree 子树下。例如，在 Vocabulary tree 中 Boston 是位于“名词” → “地理位置” → “城市”下面。当然，用户也可以通过这个子树来输入单词，但这种方式并不是很方便。而在模板和替换功能中使用户“填充空白”时，这些语义分组则是很有用的。

模板和替换

用户可以从数据库中选择中任何一个句子作为模板以创建一个新的句子。首先输入这个句子起始部分的一个或多个单词，然后在 Template 子树下面进行查找。在图 30-1 的界面底部就是关于完成词组或者句子的提示。为了构造这个列表，eLocutor 在数据库中查找以位于最后一个句子结束符之后的输入单词为起始部分的句子。在 Template 子树下也

有相同的提示，用户可以通过这些提示来创建新的句子，而这只需简单地填充在原来句子中的空白之处即可。如果存在过多的提示，那么这个单词或者词组就可以被放到过滤器中。只有包含了过滤器中内容的词组或者句子才会被显示出来。

eLocutor 通过逐字查看被选作模板的词组来处理模板。如果模板中的某个单词无法在 vocabulary tree 中找到，那么这个单词将被直接输入到缓冲区中。对于每个能够在 vocabulary tree 中找到的单词，通过使用树型控件的索引功能，eLocutor 将跳转到树型控件的相应部分，从而使用户能够选择这个单词或者单词的某个同级节点。因此，如果在数据库中包含了句子 “How are you?”，那么用户只需一些快速的点击就可以输入 “How is she?”。在进行 “填空”的同时，还没有被使用的模板部分将被显示在树型控件下方的 Template 文本框中。

模板功能使用在 vocabulary tree 下的词汇逻辑分组来变换整个句子或者词组的内容。替换功能可以在单个单词上实现类似的功能，也就是在中间文本框中的最后一个单词。然而，并不是所有的单词都被列在 vocabulary tree 下。因此就需要一个界面上的文本框来告诉用户，目前的这个单词位于哪个分类下面。这在界面上是一个标题为 Replace 的文本框。如果缓冲区中的最后一个单词能够在 vocabulary tree 中找到，那么它的父节点名字就被写入到 Replace 文本框中。

例如，如果缓冲区中的最后一个单词是 Boston，替换文本框中将包含单词 Cities。这告诉用户软件已经识别出了最后一个单词的类别。如果用户随后选择了 Replace 命令（在 Word Completion 子树中），那么最后一个单词将被从缓冲区中删除，并且用户将跳转到 vocabulary tree 中这个单词所处的位置，这样用户就能够更容易地找到另一个城市名字来替换。

在图 30-1 中，最后输入的单词是 We。在 Replace 文本框中显示了 subjectpronoun（这在给定的语义空间中并不是完全合适的）。选择 Replace 删除掉 We 并且将用户跳转到 subjectpronoun 子树，例如，在这个子树中可以很容易地选择 You。

缓存的实现

eLocutor 中的缓存机制依赖于 SaveReverse 函数，这个函数带有两个参数：保存文本的文件名以及文本。这个函数将用一个新的文件来替换这个文件，传递给 SaveRevers 函数的文本参数被写在新文件的第一行，后面是在原始文件中不同于文本参数的前 19 行。

函数将首先把变量 stringtoadd 中的文本保存到 starrray 的第一个元素中，然后用原始文件中的其他行内容来填充数组的剩余部分，只要这些行的内容不同于 stringtoadd 即

可 (HistoryLength 是一个常量, 值为 20)。最后, 这个文件被打开以进行写入操作, 这将使文件原来的内容都被删除, 而 starray 中的所有内容将被写入到文件中。

这样, 如果使用了一个在 *favouritecities.txt* 中已经被列出的城市名字, 那么这个城市名字改变位置并且成为文件中的第一个名字。如果使用了一个新的城市名字, 那么它同样将成为第一个名字, 而在它后面的将是文件以前内容中的前 19 个名字。换句话说, 文件的最后一行内容被去掉了, 并且在文件中增加了一行新的内容。从这个函数的名字我们可以看出, 文本中的行是以逆序来保存的, 最近使用的单词将被放在第一行。

在示例 30-2 中给出了函数 SaveRevers 的代码。

示例 30-2: 将文本增加到文本文件的起始位置, 并且不重复

```
Sub SaveReverse(ByVal filest As String, ByVal stringtoadd As String) 'not
' 不是追加, 而是在起始位置插入,
' 并且删除重复内容
    Dim starray(HistoryLength) As String
    Dim i As Long
    Dim arrlength As Long
    Dim st As String
    Dim filenum As Long
    starray(0) = stringtoadd
    filenum = FreeFile
    i = 1
    On Error GoTo err1
    Open filest For Input As #filenum
    While Not EOF(filenum) And (i < HistoryLength)
        Line Input #filenum, st
        If (st <> stringtoadd) Then      '只保存不重复的内容
            starray(i) = st
            i = i + 1
        End If
    Wend
    arrlength = i - 1
    Close #filenum
    Open filest For Output As #filenum      '这将删除现有文件中的内容
    For i = 0 To arrlength
        Print #filenum, starray(i)
    Next
    Close #filenum
    Exit Sub
err1:
    MsgBox "error with file " + filest
    Open filest For Output As #filenum
    Close #filenum
    Open filest For Input As #filenum      '如果文件不存在将创建一个空文件
    Resume Next
End Sub
```

Common Words 和 Favorites

在 common words 子树中收集的是经常使用的单词，它包括两个部分。一个部分是静态的，包括一些使用频率极高的单词，例如 *a*、*and*、*but* 等。另一个部分是动态部分，包括其他经常使用的单词，这些单词被放在 favouritechoices 子树下。

用户在 Speller 中看到的最后 20 个单词可以在 favouritechoices 子树下面找到。类似地，如果在 vocabulary tree 中存在一个 cities 节点，用户只需要创建一个空文件 favouritecities.txt。此后，用户在 cities 子树下所做的最近 20 个选择将会出现在 favouritechoices 子树的 favouritecities 节点下。这样，用户就可以自行决定需要记住什么类型的单词以及它们的顺序。

在示例 30-3 中给出了创建新的 favourites 并且把它插入到子树中的代码。注意 stfavourite 是一个常量，函数 MakeFullFileName 将返回一个包括路径、文件名和 .txt 扩展名的正确文件名。

示例 30-3：eLocutor 如何将 favourites 下面已经输入的单词写入到文件中

```
Public Sub AddToFavourites(parenthead As Node, stAdd As String)
Dim tempfilename As String
    If parenthead.Text = stStart Then
        Exit Sub
    End If
    tempfilename = MakeFullFileName(App.Path, stfavourite + parenthead.Text)
    If FileExists(tempfilename) Then
        SaveReverse tempfilename, stAdd
    Else
        AddToFavourites parenthead.Parent, stAdd
    End If
End Sub
```

当输入一个单词时，eLocutor 将查看这个单词是否也可以在 vocabulary tree 中找到。假设刚才输入了 Boston 这个单词。此时，如果 favouritecities.txt 这个文件存在的话，Boston 将通过 SaveReverse 函数被插入到文件的顶部。否则，eLocutor 将查找 favouriteplaces.txt，这是因为 Cities 的父节点是 Places。如果这个文件不存在，那么 eLocutor 将继续查找更高一级的父节点。如果 favouriteplaces.txt 这个文件存在，那么 Boston 通过相同的函数将被添加到这个文件中。这使得用户可以在某种程度上来控制软件应该把什么看成是用户的 favourites。通过创建一个叫作 favouritecities.txt 的文件，用户告诉 eLocutor 需要多次使用城市名字。

回溯路径

为了帮助用户在非常大的树型控件中跳转，eLocutor 自动地在用户所选择的每个子树中

进行的最近 20 次操作。这些操作记录将很方便地被呈现给用户。每个父节点 x 都有一个子树 x_Next 。在选择了一个叶节点之后，用户可以在同级的 $_Next$ 节点下查看，并且选择与用户想要执行的操作相似的操作记录。*eLocutor* 能够有效地检测用户操作中的模式并且使用户能够很容易地重复这些操作。*eLocutor* 还记住了打开过的 20 个文件，搜索文本的最近 20 个搜索项，以及用户最近说的 20 句话。所有这些功能都可以很容易地通过 `SaveReverse` 函数来实现。

输入缓冲区、编辑与滚动

我们在处理文本滚动以及选择剪切和粘贴的文本时有几种不同的方式。大多数编辑器都是在单个窗口内工作。当然，在处理大型文档的时候，在单个窗口内无法显示所有的文本，这就需要滚动条来帮助浏览文本。在需要复制或者剪切文本的时候，必须首先选择文本。被选择的文本将使用不同的前景色和背景色来高亮显示。在按照这种标准方式来设计时，我们发现存在着一些困难。

我们希望患有脑瘫的用户也能够使用 *eLocutor*，他们通常有着严重的运动残疾，因此在语言上和视觉上的有一些损伤。对于这些用户来说，我们至少需要把文本的一部分用非常大的字体来显示。如果我们把这种方式应用于界面上的所有文本，那么将根本无法在界面上显示这么多的文本。我们认为将窗口中的部分文本用非常大的字体来显示是一种不好的方式。此外，有些用户还发现在剪切和粘贴中使用改变文本背景色的高亮方式有些混乱且难以阅读。我们在音频编辑中获得的经验使我们选择了一种不同的方式。

在过去使用磁带来录音的时候，编辑人员需要监听磁带，当剪辑部分的开始位置出现时便用一个夹子夹住，然后继续监听，当剪辑部分的结束位置出现时再用一个夹子夹住。如此一来，在夹在中间的磁带就很容易被剪切或者用其他的磁带来替代。这样，这盒磁带就被分成了三个部分：夹子 1 之前的部分，夹子 2 之后的部分，以及在这两个夹子之间的部分。

我们对文本采用了相同的方法，把文本划分到三个文本框中，并且在这些文本框之间有一些门（gate）。所有的输入操作都是在中间文本框中的文本末尾进行的。这也是插入和删除文本的地方。在 Edit 子树中的 Backspace 选项将从末尾开始删除中间文本框中的文本。你可以选择是否想要删除字符、单词、词组、句子、段落或者整个中间文本框中的文本。

如果你选择了在 Edit 子树下的 Cut 或者 Copy 选择，那么在中间文本框中的所有文本都会被复制到剪贴板。当然，Cut 操作将清空中间文本框的内容。我们可以将这种方式与传统的编辑器相比，在传统的方式中，你需要设置剪切或者复制文本块的起始位置和结

束位置，而在eLocutor中，你可以把这个文本块的起始位置想象为上面文本框和中间文本框的边界，把结束位置想象为中间文本框和下面文本框的边界。Cut或者Copy命令总是提取出在中间文本框中的所有内容。

将文本划分到几个文本框中的方式使我们能够更细致地使用界面的有效区域。只有在滚动的时候，我们才会在上面的文本框中显示文本。而在其他时候，我们可以在这个文本框中用大字体来显示树型控件中高亮节点，如图30-1所示，或者显示节点的下一级内容，从而为用户提供一个“预览”。同样，我们在必要时将用下面的文本框来显示长按菜单。

在思考如何最大限度地利用界面的有效区域时，我们进行了多次反复试验。当特定用户对于界面上的显示内容有着特殊需求时，我们都会努力地把这些特定的显示内容放在上面和下面的文本框中。

在这里的“门”类似于音频编辑中的夹子。如果你想要剪切一大段的文本，那么首先需要滚动文本，直到这段文本的起始位置处于中间文本框与上面文本框的边界上。此时我们将关闭在中间文本框和上面文本框之间的门，这样接下来的滚动操作就不会使文本越过这个边界了，即文本被“夹”在这个位置上。你可以继续向上滚动或者向下滚动，直到需要剪切的文本位于中间文本框的末尾。现在你就可以选择Edit子树中的Cut选项。

在Scroll下的菜单选项可以使得一个或者两个门被打开。红色和绿色的圆圈显示了门的状态。在图30-1中，两扇门都是打开的，这分别由中间文本框左边和右边的绿灯表示。Text Up和Text Down这两个命令可以用来在文本框中移动文本。如果要在上面文本框和中间文本框之间移动文本，或者在中间文本框和下面文本框之间移动的文本，那么必须打开相应的门。

在text up/down命令中移动文本的数量取决于用户选择的标记，标记可以是字符、单词、标点符号、句子或者段落。当前选择的滚动标记被显示在树型控件下方的界面中。也可以通过命令来把整个文本内容从一个文本框中移到另一个文本框中。

如果要在输入时滚动一小段文本，那么可以通过长按来实现Words Down和Words Up。当选择了这两个选项中的一个时，单词将会在选定的方向上滚动，直到再次点击右键鼠标。注意，标点符合也被视作为单词。这使得用户能够在非常靠近文本插入位置或者删除位置的地方快速进行修改，从而在输入的时候快速地滚动一小段文本。

剪贴板

当用户选择了在Edit子树中的Cut或者Copy时，将会调用SaveReverse函数把中间文

本框中的内容写入到文件 *clipboard.txt* 中，总共能保留 20 个段落。这种方法的好处在于它能够很容易地重新安排段落，以及把原来剪切的内容反复地粘贴。而在大多数文本编辑器中，每次选择 Cut 或者 Copy 的时候都会导致剪贴板上之前的内容丢失。在 eLocutor 中，之前剪贴板上的信息将会保留一段时间。

搜索

在任何一款设计完善的编辑器中都会包含搜索功能，但 eLocutor 使我们能够以全新的方式来看待这个基本功能。我们意识到，搜索只是滚动的一种特殊情况，因此我们只是对滚动实现进行了扩展。用户可以把文本从中间文本框复制到过滤器缓冲区中，或者通过长按，用树型控件中的高亮文本来选择 Set Filter。当文本位于过滤器并且发出了一个滚动命令时，将一直滚动文本，直到在中间文本框中找到了过滤器中的内容，或者滚动到了文本的末尾。

宏

我们在与 Hawking 教授的办公室人员的讨论中，发现了一个有趣的问题。他们告诉我，当教授使用 Equalizer 来发表演讲时，如果灯光模式使得他难以看清屏幕，那么就会遇到一些问题。如果无法阅读界面，那他就很难滚动文本，进而也就无法发出说话的命令。

在 eLocutor 中，我们已经可以把演讲的所有文本都放在中间的文本框中，并且向软件发出一个命令把这些文本朗读出来，但这还是不够的。人们在这期间可能鼓掌或者发出笑声，因此就需要首先等待听众们安静下来然后再继续演讲。

我们可以在用户每次选择一个特定的菜单项时，滚动并且朗读一个句子，这个功能并不困难实现。但我们不想直接实现这个功能，而是希望在更具一般性的层次上来解决这个问题，通过提供一个宏使得在将来也能够实现其他这样的组合。

在 Commands 子树中有一个 Macros 节点，在这个节点下列出了 *C:\eLocutor\macros* 这个子目录中的所有文件。只要选择了任何一个文件，那么这个文件将被打开，并且在这个文件中列出的命令将被依次执行。在宏的设计中不存在任何复杂性：没有跳转、循环或者分支。

我们创建了两个简短的宏以用于发表演讲，分别是 *preparespeech* 和 *scrollspeak*。*Preparespeech* 将打开位于上面文本框和中间文本框以及下面文本框和中间文本框之间的两扇门，并且把整个文本都推入到门下方的文本框中。在执行了这个宏之后，当 *scrollspeak* 处于高亮状态时，用户可以通过长按来选择 Pause。所有这些操作都可以预先完成。

在演讲台上，用户并不需要看到界面。现在，每当他点击时，都会执行 scrollclick 宏，这将有效地执行两个命令。第一个命令是 Text Up，这个命令把滚动标记所标识的文本从下面的文本框中发送到中间的文本框中，以及从中间的文本框发送到上面的文本框中。第二个命令将朗读中间文本框中的内容。在发表演讲时，滚动标记通常将被设置为一个句子，这样在演讲时就是每次朗读一个句子，但如果希望获得更高的灵活性，也可以将滚动标记设置为以段落为单位。

用户界面的效率

为了评测eLocutor在帮助用户输入时的效率，在界面的右下方显示了两个数字（参见图 30-2）。这分别表示在中间文本框中的内容最近一次被清空之后，用户的点击次数以及在最后一次点击和第一次点击之间经过的秒数。

我们发现，当预测功能工作得比较好时，输入字符的点击率要高于 0.8——也就是说，这比正常人通过完整键盘的敲击次数少了很多。当预测功能工作得不好时——例如在构造一个与数据库中的句子非常不同的句子时——所需的点击次数将翻倍。

下载

eLocutor 是一个免费的，开源的软件，可以从 <http://holisticit.com/eLocutor/elocutorv3.htm> 网址上下载。在 <http://groups.yahoo.com/group/radiophony> 上有一个讨论列表。

上述下载中包含了完整的源代码。需要提醒的是，这些代码有些像一碗意大利面条（意指难以理解和阅读），这是我的责任。因为当我开始这个项目时，我已经有 10 余年的时间没有编程了，所以我的编程技术都已经过时了。我也没有做出一个完整的设计，只有偶尔会得到一些提示。随着我对问题理解的不断加深，代码的数量也在增长，因此就导致了这样的结果。我只是使用了非常简单的编程技术，这从本章的源代码中可以很容易地看出来。

未来的发展方向

eLocutor 应该算是一个快速应用程序开发（Rapid Application Development, RAD）项目（注 5），这使得我能够在为数不多的简短会议中向 Hawking 教授展示目前的设计进展到了什么程度。这个设计以工作原型的形式冻结下来，人们可以使用这个原型并且提出

注 5: http://en.wikipedia.org/wiki/Rapid_application_development.

反馈，这里的意图是希望能在某个时刻重写 eLocutor。此时，我将选择一种跨平台的编程语言，这样，那些有着高度运动残疾的用户也能够访问 Macs 或者 Linux。

然而，T. V. Raman 在 Emacspeak (将在第 31 章中介绍) 中取得的成就激发了我的灵感，我现在正在考虑一个全新的项目。当然，Emacs 不仅仅只是一个编辑器，它是一个非常通用的平台，多年以来；人们不断地对这个平台进行扩充，使你能够阅读邮件、处理约会、浏览网页以及执行 shell 命令等等。通过增加一个从文本到发音的智能转换功能以及一些上下文敏感的命令，Raman 很聪明地使盲人用户也能够访问 Emacs 中的内容。

因此，我正在考虑相同的方法是否也可以用于有着运动残疾的用户。这种方法的好处在于：

- 设计者将无需再去考虑鼠标，因为在 Emacs 中，无需鼠标你也能够做所有的事情。
- eLocutor 将不仅仅是一个辅助编辑器，而是使得用户可以访问计算机的所有功能。
- 我还可以在开源开发社区中寻找更多的支持，在与 Emacs 紧密相连的平台上进行开发似乎要远远好于在 MS 的 Windows 平台上进行开发。

因此，我希望阅读本章的读者能够教我如何扩展 Emacs，从而实现同样的在树型结构中进行单按钮导航的功能。最好是有人愿意接过这个项目，我将竭尽所能提供帮助。

这个软件的另一个发展方向就是帮助那些在出生时就患有残疾的孩子解决一些重要的问题，例如那些患有脑瘫和重度自闭症的小孩，他们通常得不到教育，因为他们无法在正常的教室里和老师进行交流。如果这些小孩能够通过软件来交流，那么他们就能够在普通的学校里学习了。

这对于软件编写者的挑战是更大的。你通常会认为使用计算机的人都应该是有知识的，而在这种情况下，孩子们却必须通过计算机来得到知识。在孩子们能够阅读之前，我们所编写的软件必须能够诱导他们来将这个软件作为与世界的主要交流工具。这虽然有些令人却步，但却是一个非常有意思的任务！当然，这个软件也可以在孩子们的早期用来教他们如何使用计算机，而不仅限用于残疾小孩。有没有人愿意和我一起来做这件事情呢？

Emacspeak： 全功能音频桌面

T. V. Raman

桌面是用来组织各种办公软件的工作区域。图形桌面为我们的日常工作提供了丰富的可视化交互方式，而音频桌面的目的则是为了在免视（eyes-free）环境中也能获得同样的效果。因此，音频桌面的主要目标就是通过听觉输出（包括语音输出和非语音输出）的形式来帮助终端用户完成各种计算任务：

- 通过各种电子消息服务来通信。
- 为客户端的本地文档和 Web 上的全局文档提供即时访问服务。
- 能够在免视环境中高效地开发软件。

开发 Emacspeak 音频桌面是动机是基于以下的想法：在为信息提供有效的听觉表现（auditory rendering）时，我们应该从信息的原本内容入手，而不是从信息的视觉表示入手。正是基于这种想法，我开发出了 AsTeR 技术读物语音系统（Audio System For Technical Readings，<http://emacspeak.sf.net/raman/aster/aster-toplevel.html>）。后来的主要动机是为了把在听觉文档环境中学到的经验应用于用户界面——也就是说，在界面上实现与文档同样的效果。

Emacspeak的主要目标不是在视觉界面上补充听觉形式,而是创建一个免视的用户界面,不仅用起来很舒适而且还能提高生产效率。

在传统的屏幕阅读器中,像滑块和树型控件等界面控件可以直接被翻译为语音输出。虽然这种直接翻译也可以提供完全的免视访问功能,但这种形式的听觉用户界面在使用上却是很低效的。

这些先决条件意味着在音频桌面环境中需要包括:

- 一组核心的语音和非语音的音频服务。
- 丰富的内置应用程序以实现朗读功能。
- 访问应用程序的上下文以产生相应的反馈。

产生语音输出

我在 1994 年 10 月左右开始实现 Emacspeak。目标环境是一台 Linux 笔记本电脑以及我的办公室工作站。为了产生语音输出,我在笔记本电脑上使用了一个 DECTalk Express (一种硬件语音合成器),而在办公室的工作站上则使用了 DECTalk 的软件语音合成器。

在设计能够同时使用这两种语音输出方式的系统时,最自然方式就是首先使实现一个语音服务器,然后在这个服务器中抽象出这两种输出方式之间的区别。语音服务器抽象已经很好地经受了时间的考验,它使我能够在 1999 年为 IBM 的 ViaVoice 引擎提供支持。此外,客户端/服务器 API 的简洁性还使得开源程序员能够为其他的语音引擎实现语音服务器。

Emacspeak 语音服务器是用 TCL 语言来实现的。DECTalk Express 中的语音服务器通过一条串行线路来与硬件合成器通信。例如,朗读文本字符串的命令是一个函数 (proc),这个函数带有一个字符串参数,并且把这个参数写入到串行设备中。这个函数的简化版本 `tts_say` 如下所示:

```
proc tts_say {text} {puts -nonewline $tts(write) "$text"}
```

在 DECTalk 的软件语音服务器同样实现了一个等价的 `tts_say` 简化版本,如下所示:

```
proc say {text} {_say "$text"}
```

其中函数 `_say` 将调用由 DECTalk 软件提供的底层 C 实现。

这种设计的意图是为了给每个有效的引擎创建独立的语音服务器,其中每个语音服务器

都是一个简单脚本，这个脚本在加载了相关的定义后将执行 TCL 默认的读入 - 求值 - 打印 (read-eval-print) 循环。因此，这种客户 / 服务器 API 的形式可以归结为客户端 (Emacspeak) 启动并连接到相应的语音服务器，然后将这个连接保存下来并且通过在这个连接上发送函数调用来执行服务器命令。

注意，到目前为止，我并没有明确说明如何打开客户 / 服务器的连接；在实现 Emacspeak 的网络感知 (network-aware) 功能时，这种迟绑定 (late-binding) 将是非常有用的。这样，在最初的实现中，Emacspeak 客户端通过 stdio 与语音服务器进行通信。然后，只需添加几行代码，客户与服务器就可以在网络上通信了，包括打开服务器 socket 以及把 stdin/stdout 的输入输出定向到建立的连接上。

这样，由于设计了简洁的客户 / 服务器抽象以及依赖 Unix I/O 的功能，我们可以很轻松地在一台远程机器上运行 Emacspeak，并且让这台机器连接到在本地客户端运行的语音服务器上。这使我能够在我的工作机器上运行 Emacspeak，而从世界的各个地方都可以访问这个运行中的会话。在连接的时候，我将把远程的 Emacspeak 会话连接到到笔记本电脑上的语音服务器，这种音频实现方式与通过设置 X 窗口来获得远程显示是等价的。

支持语音的 Emacs

上面给出的服务器抽象是很简单的，这使得在我开始实现系统后的一小时内，语音服务器的版本 0 就能够运行。然后，我可以继续开发这个项目中更有意思的部分：产生高质量的语音输出。语音服务器的版本 0 肯定不是完美的，我将在构建 Emacspeak 语音客户端的过程中不断对其进行改进。

简单的初次实现

在几个星期以前，我的一个朋友建议我使用 Emacs Lisp 中的 advice 功能。当我着手开发支持语音的 Emacs 时，advice 是一种很自然的选择。第一个开发任务就是，当用户按上 / 下箭头键时，让 Emacs 自动地朗读位于光标下面的文本行。

在 Emacs 中，用户的所有动作都会调用相应的 Emacs Lisp 函数。在标准的编辑模式中，按下箭头将调用 next-line 函数，而按上箭头则将调用 previous-line 函数。为了使这些命令支持语音，在版本 0 的 Emacspeak 中实现了下面这个非常简单的 advice：

```
(defadvice next-line (after emacspeak)
  "在移动光标后朗读当前行。"
  (when (interactive-p) (emacspeak-speak-line)))
```

Emacspeak-speak-line 函数用来获得位于光标下面的一行文本并把它发送到语音服务

器。在有了前面的定义后，我开发出了 Emacspeak 0.0，它为构建实际的系统提供了基础。

对初次实现的迭代开发

接下来的迭代开发通过定义良好的事件循环来增强语音服务器的功能。在收到每个语音命令时并不只是简单地执行这个命令，而是在语音服务器上对这些请求进行排队，并且语音服务器将通过 `launch` 命令来执行队列中的这些请求。

在把每个句子发送到语音引擎后，服务器将通过系统调用 `select` 来检测新到达的命令。这将使得语音朗读立即停止；由于在语音服务器的版本 0 中有一些不太成熟的实现，停止语音的命令并不会马上生效，因为语音服务器需要首先处理完之前收到的语音命令。在有了语音排队之后，客户应用程序现在可以把任意数量的文本排列起来，并且在执行像停止朗读这样的高优先级命令时，仍然能够得到很高的响应率。

在语音服务器内实现事件队列还可以使客户应用程序能够在合成语音之前更好地控制如何把文本拆分为块。这对于生产良好的音调结构来说是非常重要的。如何把文本拆分成子句的规则会根据将要朗读文本的不同而不同。例如，在 Python 这样的编程语言中，换行符是被作为语句分隔符并用来确定子句边界的，但在英语文本中，换行符并不是子句分隔符。

例如，当朗读下面的 Python 代码时，在每行代码后面都会插入了一个子句边界标志：

```
i=1  
j=2
```

在本章的“扩充 Emacs 以创建听觉显示列表”一节中将详细介绍如何区分 Python 代码以及如何将它的语义转发到语音层。

随着语音服务器能够智能地处理文本，Emacspeak 客户在处理文本时变得越来越完善。Emacspeak-speak-line 函数变成了一个语音生成函数库，它将实现以下步骤：

1. 解析文本并且将文本拆分为一系列的子句。
2. 预处理文本。例如，处理标点字符的重复字符串。
3. 执行不断增加进来的其他函数。
4. 把每个子句在语音服务器上排队，并且调用 `launch` 命令。

从此时起，在实现 Emacspeak 的剩余部分时，我使用了 Emacspeak 来作为开发环境。这对于代码库的发展过程是非常重要的。新开发的功能需要被立即测试，因为不良实现的

功能将会导致整个系统都不可用。Lisp的增量代码开发方式能够很自然地满足前者；而为了满足后者，Emacspeak 代码库被发展为“独立的（bushy）”——也就是说，高级系统中的大部分组件都是相互独立的，并且只依赖于一小部分经过仔细维护的核心代码。

简单的 advice 教程

Lisp 语言中的 advice 功能是 Emacspeak 实现中的关键技术，如果不对其进行介绍，那么本章的内容将是不完整的。advice 功能可以对现有的函数进行修改而无需改变函数的原始实现。而且，如果 `advice m` 修改了函数 `f`，那么所有对函数 `f` 的调用都会受到 `m` 的影响。

`advice` 有三种实现形式：

`before`

advice 代码在原始函数调用之前运行。

`after`

advice 代码在原始函数调用之后运行。

`around`

advice 代码代替原始函数来运行。`around` 形式的 advice 可以在必要时调用原始函数。

所有形式的 `advice` 都可以访问原始函数的参数；此外，`around` 和 `after` 这两种形式的 `advice` 还能够访问原始函数的返回值。Lisp 通过以下步骤来实现这个功能：

1. 缓存函数的原始实现。
2. 根据 `advice` 形式来生成一个新的函数定义。
3. 将这个函数定义保存为 `advice` 函数。

这样，我们在对前面“对初次实现的迭代开发”中的 `advice` 代码进行分析时，Emacs 中最初的 `next-line` 函数将由一个修改后的版本替代，这个版本的 `next-line` 函数将在获得光标下面文本后，再把这一行内容朗读出来。

产生丰富的听觉输出

在 Emacspeak 发展到这个时刻，系统的整体设计是下面这样：

1. 在 Emacs 的交互式命令中都能够支持语音功能或者通过 `advice` 来产生听觉输出。

2. 所有的 *advice* 定义都被集中在一个个模块中，并且每个需要支持语音的 Emacs 应用程序都包含其中一个模块。
 3. *advice* 形式把文本转发到核心语音函数中。
 4. 这些函数将提取出要发音的文本并把它发送到 `tts-speak` 函数。
 5. 函数 `tts-speak` 将处理接收到的文本参数，并且将它发送到语音服务器来产生听觉输出。
6. 语音服务器通过处理排队请求来产生人耳可以感知的输出。

文本的预处理是通过把文本放在一个特殊的临时缓冲区中来进行的。缓冲区通过缓冲区特定的语法表来获得专门的行为，在这些语法表中定义了缓冲区内容的语法和影响缓冲区行为的局部变量。当文本被传递到 Emaspeak 的核心函数时，所有缓冲区特定的设置也将被传递到这个特殊的临时缓冲区，并在这个缓冲区中对文本进行预处理。这个过程将自动确保按照基本的语法将文本解析为有意义的子句。

使用 `voice-lock` 的音频格式

Emacs 能够按照句法规则通过 `font-lock` 以不同的颜色来显示文本。在创建视觉外观时，Emacs 将在文本字符串中增加 `face` 文本属性，这个属性的值将用来指定显示文本时使用的字体、颜色以及风格。拥有 `face` 属性的文本字符串可以被认为是一个概念上的视觉显示列表（visual display list）。

Emaspeak 通过 `personality` 文本属性来扩充视觉显示列表，`personality` 文本属性的值用来指定在朗读文本时所需使用的听觉属性，这在 Emaspeak 中叫作 `voice-lock`。`Personality` 属性值是一个 Aural CSS (ACSS) 设置值，表示不同的语音属性——例如，朗读的音调等。需要注意的是，在所有的 TTS 引擎中并没有明确定义这种 ACSS 设置值。Emaspeak 在引擎特定的模块中实现了 ACSS-to-TTS 映射模块，在这个模块中将对高级听觉属性进行映射——例如把音调或者音调范围映射到引擎特定的控制编码。

在下面的章节中将介绍 Emaspeak 如何对 Emacs 进行扩充以创建听觉显示列表，以及如何通过处理这些听觉显示列表来产生引擎特定的输出。

扩充 Emacs 以创建听觉显示列表

在实现 `font-lock` 的 Emacs 模块中可以调用 Emacs 的内置函数 `put-text-property` 来添加相关的 `face` 属性。在 Emaspeak 中定义了一个 *advice*，使得 `put-text-property` 函数在增加 `face` 属性时也会添加相应的 `personality` 属性。注意这两个显示属性

(face 和 personality) 的值都可以是列表；因此，在设计这些属性的值时采用了级联的方式以创建最终的（视觉或者听觉）形式。这也意味着应用程序的不同部分可以逐步地增加显示属性值。

函数 put-text-property 的原型如下所示：

```
(put-text-property START END PROPERTY VALUE &optional OBJECT)
```

这个函数的 *advice* 实现为：

```
(defadvice put-text-property (after emacspeak-personality pre act)
  " Emacspeak 用来扩充 font lock。"
  (let ((start (ad-get-arg 0)) ;; 绑定参数
        (end (ad-get-arg 1)))
    (prop (ad-get-arg 2)) ;; 被增加属性的名字
    (value (ad-get-arg 3))
    (object (ad-get-arg 4))
    (voice nil)) ;; 映射到 voice
  (when (and (eq prop 'face) ;; 避免无限递归
             (not (= start end))) ;; 非空的文本范围
        emacspeak-personality-voiceify-faces)
  (condition-case nil ;; 安全地查找 face 映射
      (progn
        (cond
          ((symbolp value)
           (setq voice (voice-setup-get-voice-for-face value)))
          ((ems-plain-cons-p value)) ;; 传递普通的朗读
          ((listp value)
           (setq voice
                 (delq nil
                      (mapcar #'voice-setup-get-voice-for-face value))))
          (t (message "Got %s" value))))
        (when voice ;; 在 voice 中保存了一组 personality
          (funcall emacspeak-personality-voiceify-faces start end voice object)))
      (error nil))))
```

以下是对这个 *advice* 定义的简单说明：

绑定参数

首先，这个函数将使用 *advice* 内置的 ad-get-arg 把一组词法变量绑定到传递给函数的参数。

Personality 设置函数

从 face 到 personality 的映射是通过用户自定义变量 emacspeak-personality-voiceify-faces 来控制的。如果这个变量不为空，那么它将表示一个函数，函数的原型如下所示：

```
(emacspeak-personality-put START END PERSONALITY OBJECT)
```

Emacspeak提供了这个函数的不同实现，这个函数将把新的personality值添加或者插入到现有的personality属性中。

监测

除了判断非空的`emacspeak-personality-voiceify-faces`之外，这个函数还将执行其他的判断以确定这个*advice*定义是否需要执行某些操作。这个函数将继续进行判断，如果：

- 文本范围为非空的。
- 将要增加的属性是一个`face`属性。

第一个判断用来避免极端情况，例如使用零长度的文本范围来调用`put-text-property`函数。第二个判断用来确保仅当被增加的属性是`face`属性时，才可以增加`personality`属性。注意，如果没有第二个判断，那么将导致无穷递归，因为在调用`put-text-property`来增加`personality`属性时同样会触发*advice*定义。

获得映射

接下来，函数将安全地查找正在被应用的`face`（或者一组`face`）中的`voice`映射。如果使用单个`face`属性，那么这个函数将查找相应的`personality`映射，如果是一组`face`属性，那么函数创建一组相应的`personality`。

应用`personality`

最后，这个函数将判断是否找到了有效的`voice`映射，如果找到了，那么就会使用在`voice`变量中保存的这组`personality`来调用`emacspeak-personality-voiceify-faces`。

来自听觉显示列表的音频格式输出

通过前面的*advice*定义，拥有视觉风格的文本将获得相应的`personality`属性，在这个属性中保存的是用于音频格式内容的ACSS设置。这可以将Emacs中的文本转换为丰富的听觉显示列表。在本节中将介绍如何对Emacspeak的输出层进行扩充以把这些听觉显示列表转换为能够听见的语音输出。

Emacspeak的`tts-speak`模块在把文本发送到语音服务器之前将对文本进行预处理。正如在前面所介绍的，这个预处理过程由一系列的步骤组成，包括：

1. 应用发音规则。
2. 处理标点符号的重复字符串。
3. 根据上下文将文本拆分为合适的子句。

4. 将 personality 属性转换为音频格式编码。

在本节中将介绍 `tts-format-text-and-speak` 函数，这个函数用来处理从听觉显示列表到音频格式输出的转换。首先，在下面给出了 `tts-format-text-and-speak` 函数的代码：

```
(defsubst tts-format-text-and-speak (start end )
  "对 start 和 end 之间的文本进行格式化并且朗读。"
  (when (and emacspeak-use-auditory-icons
             (get-text-property start 'auditory-icon)) ;;对图标排队
    (emacspeak-queue-auditory-icon (get-text-property start 'auditory-icon)))
  (tts-interp-queue (format "%s\n" tts-voice-reset-code))
  (cond
    (voice-lock-mode ;; 仅当 voice-lock 模式为开启状态时才使用音频格式
      (let ((last nil) ;; 初始化
            (personality (get-text-property start 'personality)))
        (while (and
< start end) ;; chunk at personality changes
          (setq last
                (next-single-property-change start 'personality
                                              (current-buffer) end)))
        (if personality ;; 音频格式块
            (tts-speak-using-voice personality (buffer-substring start last))
            (tts-interp-queue (buffer-substring start last)))
        (setq start last ;; 准备处理下一块
              personality (get-text-property last 'personality)))))
    ;; 没有 voice-lock, 仅发送文本
    (t (tts-interp-queue (buffer-substring start end))))))
```

函数 `tts-format-text-and-speak` 在每次调用时都将处理一个子句，参数 `start` 和 `end` 分别被设置为子句的起始位置和结束位置。如果打开了 `voice-lock` 模式，那么这个函数将在 `personality` 属性值发生变化的地方把子句拆分为块。在确定了一个跃迁点后，`tts-format-text-and-speak` 函数将调用 `tts-speak-using-voice`，并且把要使用的 `personality` 以及需要发音的文本传递给这个函数。我们将在后面介绍这个函数，它将在把音频格式输出发送给语音服务器之前查找正确的设备特定编码。

```
(defsubst tts-speak-using-voice (voice text)
  "用 VOICE 来朗读 TEXT。"
  (unless (or (eq 'inaudible voice) ; 如果是不可听见的声音，则不朗读
              (and (listp voice) (member 'inaudible voice)))
    (tts-interp-queue
      (format
        "%s%s %s \n"
        (cond
          ((symbolp voice)
           (tts-get-voice-command
             (if (boundp voice) (symbol-value voice) voice)))
          ((listp voice)
           (mapconcat #'(lambda (v)
```

```

(tts-get-voice-command
  (if (boundp v) (symbol-value v) v)))
  voice
  ""))
(t """))
text tts-voice-reset-code))))
```

如果指定的声音是不可听见的，那么 `tts-speak-using-voice` 函数将立刻返回。这里，“不可听见”属性是一种特殊的 `personality`，Emacspeak 用它指定哪些文本不需要朗读。在需要有选择性地隐藏部分文本以产生更简洁的输出时，可以使用这个 `personality`。

如果指定的声音（或者一组声音）是可以听见的，那么这个函数将查找声音的语音编码并且对 `voice-code` 和 `tts-reset-code` 之间需要发音文本进行包装，然后将把包装的结果在语音服务器上排队。

在风格化语音输出时使用听觉 CSS (ACSS)

我最初是在 AsTeR 中定义了音频格式，其中的表现 (rendering) 规则是用专门的音频格式语言 (Audio Formatting Language, AFL) 来编写的。AFL 将听觉空间中的可用参数——例如朗读声音中的音调——构成一个多维空间，并且把表现引擎的状态封装成这个多维空间中的一个点。

AFL 提供了一种块状结构的语言，在这种语言中通过基于词法作用域的变量来封装当前的表现状态，此外 AFL 还提供在了运算符以便在这个结构化的空间中进行移动。当这些概念被映射到 HTML 和 CSS 的声明性领域时，构成 AFL 表现状态的维将成为 ACSS 的参数，这在 CSS2 中并定义为的可访问性测度 (<http://www.w3.org/Press/1998/CSS2-REC>)。

虽然 ACSS 是被设计用来实现 HTML (以及 XML) 中 Markup 树的风格，但它也被证明是用于实现 Emacspeak 音频格式层的一个良好抽象，并能够使这个实现独立于任意的 TTS 引擎。

以下是封装 ACSS 设置的数据结构定义

```
(defstruct acss
  family gain left-volume right-volume
  average-pitch pitch-range stress richness punctuations)
```

Emacspeak 提供了一组用于语音扩展的预定义声音修饰 (voice overlay)。声音修饰在 Aural CSS 中被设计为级联的方式。例如，以下是与 `voice-monotone` 对应的 ACSS 设置：

```
[cl-struct-acss nil nil nil nil nil nil 0 0 nil all]
```

注意，在这个 acss 结构中的大多数域都是空的——也就是说，没有被设置。这个结构将创建以下的声音修饰：

1. 将音调设置为 0 以创建扁平的声音。
2. 将音调范围设置为 0 以创建没有变化的单调声音。

在所有编程语言模式中，这个设置被用作为音频格式注释的 personality 属性值。由于这个值是一个声音修饰，因此它能够有效地与其他听觉显示属性交互。例如，如果某个注释的一部分用粗体字来显示，那么在这些部分文本上可以增加 voice-bolden personality（这是另一个预定义声音修饰）；这就将 personality 属性设置为一个包含两个值的列表：(voice-bolden voice-monotone)。最终结果就是，这段文本将用不同的语音朗读出来，在朗读中体现了文本的两个属性：即这是注释中一段被强调的文字。

3. 将标点符号设置为 all，这样标点符号也将被朗读。

增加听觉图标

丰富的视觉用户界面包括文本和图标。同样，在 Emacspeak 能够智能地朗读后，接下来的步骤就是通过听觉图标 (auditory icon) 来扩展语音输出，从而提高听觉通信的带宽。

Emacspeak 中的听觉图标都是简短的声音片段 (持续时间不会超过 2 秒钟)，用于表示在用户界面中频繁发生的事件。例如，当用户保存文件时，系统将播放一个确认声音。同样，在打开或者关闭某个对象 (可以是从文件到网站的任何一个对象) 时将播放相应的听觉图标。这组听觉图标是被陆续实现的，它们能够表示一些常见的事件，例如打开、关闭或者删除对象等。在本节中将描述如何将这些听觉图标插入到 Emacspeak 的输出流中。

听觉图标是在以下的用户交互中产生的：

- 提示明确的用户动作。
- 在语音输出中增加额外的提示。

确认用户动作的听觉图标——例如，文件被成功地保存——是通过把一个 after-advice 增加到不同的 Emacs 内置函数中来实现的。为了在整个 Emacspeak 桌面中提供一致的声音和感觉，在 Emacs 中许多调用这段代码的地方都要添加这种扩展。

以下通过 *advice* 来实现的一个扩展示例：

```
(defadvice save-buffer (after emacspeak pre act)
  "如果可能的话，生成一个听觉图标。"
  (when (interactive-p) (emacspeak-auditory-icon 'save-object)
  (or emacspeak-last-message (message "Wrote %s" (buffer-file-name)))))
```

此外，还可以通过 Emacs 提供的钩子（Hook）来实现扩展。我们在前面给出的 *advice* 简短介绍中已经解释过，*advice* 能够使我们在扩展或者修改现有软件的行为时无需修改软件的源代码。Emacs 本身就是一个可扩展的系统，而那些写得很好的 Lisp 代码也能够在通常情况中提供合适的扩展钩子。例如，在 Emacspeak 中把函数 `emacspeak-minibuffer-setup-hook` 增加到 Emacs 的 `minibuffer-setup-hook`，从而将听觉反馈添加到 Emacs 的默认提示机制（Emacs 中的 minibuffer）中：

```
(defun emacspeak-minibuffer-setup-hook ()
  "在进入 minibuffer 时的默认动作。"
  (let ((inhibit-field-text-motion t))
  (when emacspeak-minibuffer-enter-auditory-icon
    (emacspeak-auditory-icon 'open-object))
  (tts-with-punctuations 'all (emacspeak-speak-buffer))))
(add-hook 'minibuffer-setup-hook 'emacspeak-minibuffer-setup-hook)
```

这是最大限度使用内置扩展能力的一个很好示例。然而，Emacspeak 在许多情况中使用的都是 *advice*，这是因为 Emacspeak 需要为 Emacs 中的所有函数都增加听觉反馈，而在当初实现 Emacs 的时候并没有考虑这种需求。这样，Emacspeak 实现就很好地证明了 *advice* 是一种发现扩展点的功能强大的工具。

由于在普通的编程语言中没有像 *advice* 这样的功能，因此难以进行一些试验，尤其是在寻找有用的扩展点时。因为软件工程师需要考虑以下的权衡因素：

- 使系统能够实现任意地扩展（以及任意地复杂）。
- 在某些合理的扩展点上进行猜测，并且把这些猜测用硬编码的方式来实现。

在实现了扩展点后，用新的扩展点来试验就需要重新编写现有的代码，这种做法所带来的时间开销通常意味着大多数像这样的扩展点在很长时间里都无法被发现。Lisp 中的 *advice* 及 Java 中的 Aspects，都可以使软件工程师在进行试验时无需担心会对现有源代码造成负面影响。

在朗读内容时产生听觉图标

除了使用听觉图标来提示用户交互的结果外，Emacspeak 还使用听觉图标来扩充所朗读的内容。这种听觉图标的示例用法包括：

- 在段落开始之处的朗读一个短听觉图标。
- 在越过带有断点的源代码行时朗读一个听觉图标标记对象。

听觉图标是通过在文本属性`emacspeak-auditory-icon`上添加一个值来实现的，这个值是将在相关文本上播放的听觉图标的名字。

例如，在Grand Unified Debugger Emacs软件包（GUD）中设置断点的命令通过*advice*方式把属性`emacspeak-auditory-icon`增加到包含这个断点的代码行上。当用户的光标越过这一行的时候，函数`tts-format-text-and-speak`将把听觉图标放置在输出流的正确位置上。

日历：通过上下文敏感的语义来增强语音输出

总结到目前为止所讨论的内容，Emacspeak有以下的能力：

- 在应用程序的上下文中产生听觉输出。
- 实现音频格式输出以提高口语通信的带宽。
- 用听觉图标来扩充语音输出。

在本节中将解释这个设计中的一些增强功能。

我是在1994年10月开始实现Emacspeak，这是为Linux开发语音解决方案的一种快速方法。当我在1994年11月的第一个星期使得Emacs日历能够支持语音时，我意识到我所实现的功能要远远好于我在以前使用过的任何一个语音访问解决方案。

如果想要说明在视觉媒介和传输信息上都进行了优化的视觉布局，那么日历无疑是一个很好的示例。在描述日期时，我们在直觉上会按照日期和月份的形式来进行思考；通过表格布局将日期组织为网格形式，每个星期都出现在表格的一行，这种表示方式很符合我们的直觉。通过这种形式的布局，人眼可以快速地在日历中按照日期、星期和月份来浏览日历，并且可以很容易地回答“明天是几号”以及“我在下个月的星期三是否有空”等这些问题。

然而，如果只是简单地把这个二维布局朗读出来，那么无法在听觉交互中获得在视觉上下文中的信息获取效率。这个结论很好地印证了这个观点：良好的听觉反馈应该直接从传输信息的本质内容中产生出来，而不是从信息的视觉表示中产生。当需要从视觉格式信息中产生听觉输出时，软件必须在朗读信息之前重新找到在信息中包含的基本语义。

相反，如果通过扩展底层应用程序的*advice*定义来产生语音反馈，那么软件就能够完全

访问应用程序的运行时上下文。这样，软件不仅是基于视觉布局来进行推测，而是可以指导底层应用程序朗读正确的内容！

Emacspeak-calendar模块通过定义朗读日历信息的工具函数以及对调用这些函数的日历导航命令进行 *advise* 修改来实现朗读的功能。因此，Emacs 日历通过把箭头按钮绑定到日历导航命令而不是普通编辑模式中的默认光标导航来产生专门的行为。Emacspeak 对日历特定的命令使用 *advise*，从而在日历的上下文中朗读相关的信息。

这种做法的结果就是，从终端用户的观点来看，一切都工作正常。在普通的编辑模式中，按上/下箭头将会朗读当前行；在日历中按上/下箭头则会按照星期来导航，并且朗读当前日期。

在下面给出了在emacspeak-calendar模块中定义的emacspeak-calendar-speak-date 函数。注意在这个函数中使用了目前为止描述的所有功能来进行访问以及将日历中相关上下文的信息表示为音频格式：

```
(defsubst emacspeak-calendar-entry-marked-p( )
  (member 'diary (mapcar #'overlay-face (overlays-at (point)))))

(defun emacspeak-calendar-speak-date( )
  "当在日历模式中调用时，朗读当前位置上的日期。"
  (let ((date (calendar-date-string (calendar-cursor-to-date t))))
    (cond
      ((emacspeak-calendar-entry-marked-p) (tts-speak-using-voice mark-personality
date))
      (t (tts-speak date)))))
```

Emacs 用特殊的声音修饰来标出带有日志的日期。在前面的定义中，辅助函数 emacspeak-calendar-entry-marked-p 将通过判断这个声音修饰来实现一个谓词函数，这个函数可以用来判断某个日期是否带有日志。Emacspeak-calendar-speak-date 函数将使用这个谓词函数来确定这个日期是否需要用不同的声音来朗读；带有日志的日期将使用 mark-personality 声音来朗读。注意，emacspeak-calendar-speak-date 函数将在函数调用中访问日历的运行时上下文：

```
(calendar-date-string (calendar-cursor-to-date t))
```

Emacspeak-calendar-speak-date 函数是在附加到所有日历导航函数的 *advice* 定义中被调用的。以下是函数 calendar-forward-week 的 *advice* 定义：

```
(defadvice calendar-forward-week (after emacspeak pre act)
  "朗读日期。"
  (when (interactive-p) (emacspeak-speak-calendar-date)
  (emacspeak-auditory-icon 'large-movement)))
```

这是一个after形式的*advice*,因为我们希望在最初的导航命令执行完了之后再产生语音反馈。

在*advice*定义的代码中，首先调用`emacspeak-calendar-speak-date`函数来朗读位于光标下的日期；接下来，它将调用`emacspeak-auditory-icon`来产生一个短促的声音来表示已经成功地移动了。

对于在线信息的简单访问

在有了能够产生丰富听觉输出的所有必要功能后，在支持语音的Emacs应用程序中使用Emacs Lisp的*advice*功能只需要非常少的代码。TTS层和Emacspeak内核将处理产生高质量语音输出的细节内容，而支持语音功能的扩展只需关注独立应用程序的具体语义；这使我们能够实现简单而漂亮的代码。在本节中，我将从Emacspeak丰富的信息访问工具软件包中选择一些示例来阐述这个概念。

就在我着手开发Emacspeak的时候，在计算世界中发生了一次更为深刻的革命：万维网从研究机构中的工具变成了日常工作的主流形式。这是发生在1994年，当时编写浏览器仍然是一个相对简单的任务。在之后的12年中，在Web中不断增加的复杂性经常会模糊这个事实：Web仍然是非常简单的设计，其中

- 内容的创建者通过URL来发布网页资源。
- 通过公开的协议来获取URI上的内容。
- 可获取的内容是用HTML语言来编写的，这是一种易于理解的标记语言。

注意，刚才粗略介绍的基本架构并没有提到如何使内容对于终端用户是有效的。在20世纪90年代中期，我们看到了Web中日益增长的复杂视觉交互。在商业网页上越来越多华而不实的视觉交互，使其逐渐远离了早期网站中简单的面向数据交互。到1998年，我发现Web上出现了许多有用的交互站点；但令我沮丧的是，我所访问的站点正在逐步减少，因为在使用语音输出时，访问这些站点需要很长的时间。

这使得我在Emacspeak中创建了一组面向网页的工具软件以重新回到网页交互的基本方式。Emacs已经能够把简单的HTML重新处理为交互的超文本文档。随着Web变得日益复杂，Emacspeak得到了一组构建在Emacs中HTML表现功能上的交互向导，这组向导能够逐渐地去掉网页交互中的复杂性，从而创建一个听觉界面，使用户能快速地听到想要的信息。

使用 Emacs W3 和 Aural CSS 的基本 HTML

Emacs W3 是一个简单的网页浏览器，最初是在 20 世纪 90 年代中期实现的。Emacs W3 在早期实现了 CSS (Cascading Style Sheets, 级联样式表)，而这正是最初 ACSS 实现的基础，这个 ACSS 实现是我在 1996 年 2 月份编写 Aural CSS 草稿时发布的。Emacspeak 通过 emacspeak-w3 模块在 Emacs W3 中支持语音功能，在这个模块中实现了以下扩展：

- 在 Aural CSS 的默认样式表中有一个 `aural media` 字段。
- 在所有的交互式命令中都增加了 `advice` 以产生听觉反馈。
- 使用特殊的模式来识别网页上的修饰性图片，并且不朗读这些图片。
- 为 HTML 表单域的听觉表现定义相关联的标签，用来支持 HTML 4 中 `label` 元素的设计。
- 为 HTML 表单控件定义上下文敏感的听觉表现。例如，假定有一组回答问题的单选按钮：

Do you accept?

Emacspeak 对 Emacs W3 进行扩展以产生表单的语音消息：

单选组 “Do you accept?” 按下了 Yes。

以及：

按下这个按钮来改变单选组 “Do you accept?” 从 Yes 到 No。

- 为 Emacs W3 的函数 `w3-parse-buffer` 定义了一个 `before` 形式的 `advice`，这个函数用来把用户请求的 XSLT 转换应用到 HTML 页面上。

面向任务搜索的 emacspeak-websearch 模块

到 1997 年，Web 上的交互网站，无论是用于搜索的 Altavista 到还是在线方向指示信息的 Yahoo! Maps，都要求用户进行高度视觉化的操作，包括：

1. 填充一组表单域。
2. 提交结果表单。
3. 在复杂的 HTML 网页中找到结果。

其中第一个和第三个操作在使用语音输出时需要很多时间。我首先需要在一个琳琅满目的网页上找出各种表单域，并且在获得答案之前，还要在结果页面上浏览许多复杂的信息。

注意，从软件设计的角度来看，这些步骤可以很巧妙地映射为前置动作（pre-action）和后置动作（post-action）等钩子。因为网页交互遵循基于URI的简单架构，提示用户正确输入的前置动作步骤可以从网站上提取出来，并放到一小段局部运行的代码中；这样，用户就不需要打开最初启动的网页并且从中找出各种输入域了。同样，在结果网页中找出实际结果的后置动作步骤同样能够用相应的程序来实现。

最后，我们注意到，即使每个前置动作和后置动作都是特定于某些网站的，但可以对整体的设计模式进行一般化。基于这个观点，我开发了 `emacspeak-websearch` 模块，这是一组面向任务的网页工具：

1. 提示用户。
2. 构造一个合适的 URI 并且提取出其中的内容。
3. 在通过 Emacs W3 呈现相关内容之前，对结果进行过滤。

以下是 `emacspeak-websearch` 工具，用于从 Yahoo! Maps 中访问方向指示信息：

```
(defsubst emacspeak-websearch-yahoo-map-directions-get-locations ()  
  " 提示并构造发送组件的辅助函数。 "  
  
  (concat  
    (format "&newaddr=%s"  
            (emacspeak-url-encode (read-from-minibuffer "Start Address: ")))  
    (format "&newcsz=%s"  
            (emacspeak-url-encode (read-from-minibuffer "City/State or Zip: ")))  
    (format "&newtaddr=%s"  
            (emacspeak-url-encode (read-from-minibuffer "Destination Address: ")))  
    (format "&newtcsz=%s"  
            (emacspeak-url-encode (read-from-minibuffer "City/State or Zip:"))))  
(defun emacspeak-websearch-yahoo-map-directions-search (query )  
  " 从 Yahoo 中获得驾驶方向指示信息。 "  
  (interactive  
   (list (emacspeak-websearch-yahoo-map-directions-get-locations))  
   (emacspeak-w3-extract-table-by-match  
    "Start"  
   (concat emacspeak-websearch-yahoo-maps-uri query))))
```

下面给出了对上述代码的简单解释：

前置动作

`emacspeak-websearch-yahoo-map-directions-get-locations` 函数将向用户提示起始位置和结束位置。注意在这个函数中硬编码了 Yahoo! Maps 所使用的查询参数名字。从表面上看，这似乎是一个很容易被破坏的规则。但事实上，这个规则自从在 1997 年首次定义之后就没有被破坏过。理由很明显：在网页应用程序发布

了一组查询参数后，这些参数将被硬编码到许多其他的程序中，包括在最初网站上的大量 HTML 网页。在程序中硬编码参数名字的方式对于结构化的软件架构以及自顶向下的 API 来说是脆弱的，但使用这种 URL 参数来定义自下向上的网页服务却引出了 REST 方式网页 API 概念。

内容提取

用于提取方向指示信息的 URL 是通过把用户的输入连接到 Yahoo! Maps 的基 URI 来构造的。

后置动作

结果 URL 以及搜索模式 Start 被传递给 emacspeak-w3-extract-table-by-match 函数：

- a. 使用 Emacs W3 来获取内容。
- b. 使用 XSLT 转换来提取包含 Start 的表格。
- c. 用 Emacs W3 的 HTML 格式器来表示这个表格。

与查询参数不同的是，结果页面的布局大约是每年改变一次。不过，将这个工具与 Yahoo! Maps 保持一致的问题就归结为维护这个工具的后置动作部分。在经过了 8 年的使用后，我不得不对它进行了几次修改，并且由于底层平台提供了许多用于过滤结果页面的工具，因此在每次修改布局时所需的代码行数都是最小的。

在 emacspeak-w3-extract-table-by-match 函数中使用了 XSLT 转换来过滤文档并且返回包含指定搜索模式的表格。在这个示例中，函数构造了以下的 XPath 表达式：

```
(/descendant::table[contains(., Start)])[last()]
```

这将有效地把包含字符串 Start 的一组表格挑选出来并且返回这组表格中的最后一张表格。

在这个工具编写出来 7 年之后，Google 于 2005 年 2 月份发布了 Google Maps。网络上的许多博客都对 Google Maps 进行了仔细地研究并且很快找到了这个应用程序所使用的查询参数。我使用这些查询参数在 Emacspeak 中构建了相应的 Google Maps 工具来提供类似的功能。用户能很好地适应 Google Maps 工具，因为在相同的参数内可以同时指定起始位置和结束位置。以下是 Google Maps 向导的代码：

```
(defun emacspeak-websearch-emaps-search (query &optional use-near)
  "执行 EmapSpeak 搜索。查询表达式为普通的英语文本。"
  (interactive
   (list
    (emacspeak-websearch-read-query
```

```

(if current-prefix-arg
    (format "Find what near %s: "
           emacspeak-websearch-emapspeak-my-location)
    "EMap Query: "))
current-prefix-arg))
(let ((near-p ;; 确定查询类型
      (unless use-near
        (save-match-data (and (string-match "near" query) (match-end 0))))))
  (near nil)
  (uri nil))
  (when near-p ;; 从查询中确定位置
    (setq near (substring query near-p))
    (setq emacspeak-websearch-emapspeak-my-location near)))
  (setq uri
        (cond
         (use-near
          (format emacspeak-websearch-google-maps-uri
                  (emacspeak-url-encode
                   (format "%s near %s" query near))))
         (t (format emacspeak-websearch-google-maps-uri
                    (emacspeak-url-encode query))))))
  (add-hook 'emacspeak-w3-post-process-hook 'emacspeak-speak-buffer)
  (add-hook 'emacspeak-w3-post-process-hook
            #'(lambda nil
                (emacspeak-pronounce-add-buffer-local-dictionary-entry
                 "&#240;mi" " miles"))))
  (browse-url-of-buffer
   (emacspeak-xslt-xml-url
    (expand-file-name "kml2html.xsl" emacspeak-xslt-directory)
    uri))))
```

下面给出了对上面代码的简单解释：

1. 对输入进行解析以确定是一个方向指示查询还是搜索查询。
2. 当为搜索查询时，缓存用户的位置以便在将来使用。
3. 构造用于获取查询结果的 URI。
4. 通过XSLT过滤器kml2html对URI内容进行过滤并浏览过滤的结果，在kml2html过滤器中将把获取的内容转换为简单的超文本文档。
5. 在结果中建立起定制的发音来把“mi”发音为“miles”。

注意，和前面一样，大多数的代码都是用于处理应用特定的任务。丰富的语音输出是通过把结果创建为结构良好的HTML文档来实现，在这些文档中包含有相应的Aural CSS规则来产生音频格式表示。

网页命令行和 URL 模板

随着网络上日益增多的服务，在2000年早期出现了另一种有用的模式：网站开始通过JavaScript创建智能的客户端交互。这种脚本的一个典型应用就是在客户端构造URL，用来根据用户的输入访问特定部分内容。例如，美国职业棒球大联盟（Major League Baseball）通过把比赛日期以及主队和客队的名字组合在一起构造URL，从而获得特定赛事的比分，而NPR通过把日期与给定NPR节目编码组合在一起创建URL。

为了实现对这些服务的快速访问，我在2000年下半年增加了`emacspeak-url-template`模块。这个模块成为了在前面章节中介绍的`emacspeak-websearch`模块的一个功能强大的辅助工具。这些模块共同把Emacs的minibuffer变成了一个功能强大的网页命令行，用来提供对网页内容的快速访问。

许多网页服务都要求用户指定一个日期。程序可以通过用户的日历来提供默认的日期。这样，当用于播放NPR节目或者获取MLB比分的Emacspeak工具在Emacs日历缓冲区中被调用时，将默认使用位于光标下面的日期。

Emacspeak中的URL模板将使用以下的数据结构来实现：

```
(defstruct (emacspeak-url-template (:constructor emacspeak-ut-constructor))
  name ; ; 可读的名字
  template ; ; 模板 URL 字符串
  generators ; ; 一组参数生成器
  post-action ; ; 在打开之后执行的动作
  documentation ; ; 资源文档
  fetcher)
```

用户通过Emacspeak命令`emacspeak-url-template-fetch`来调用URL模板，这个命令将提示输入URL模板并且：

1. 查找命名模板。
2. 通过调用指定的生成器来提示用户。
3. 将Lisp函数格式应用到模板字符串，并把所收集的参数用来创建最终的URL。
4. 建立起在内容被表现之后执行的任意后置动作。
5. 应用指定的获取器（fetcher）来表现内容。

这个结构的用法最好通过一个示例来说明。以下是用于播放NPR节目的URL模板：

```
(emacspeak-url-template-define
  "NPR点播"
  "http://www.npr.org/dmg/dmg.php?
```

```

prgCode=%s&showDate=%s&segNum=%s&mediaPref=RM"
(list
 #'(lambda () (upcase (read-from-minibuffer "Program code:")))
 #'(lambda ()
 (emacspeak-url-template-collect-date "Date:" "%d-%b-%Y"))
 "Segment:")
 nil;没有后置动作
 "Play NPR shows on demand.
Program is specified as a program code:
ME               Morning Edition
ATC              All Things Considered
day              Day To Day
newsnotes        News And Notes
totn             Talk Of The Nation
fa                Fresh Air
wesat            Weekend Edition Saturday
wesun            Weekend Edition Sunday
fool              The Motley Fool
Segment is specified as a two digit number --specifying a blank value
plays entire program."
 #'(lambda (url)
 (funcall emacspeak-media-player url 'play-list)
 (emacspeak-w3-browse-xml-url-with-style
  (expand-file-name "smil-anchors.xsl" emacspeak-xslt-directory)
  url)))

```

在这个示例中，自定义的获取器执行两个动作：

1. 启动一个媒体播放器来播放音频流。
2. 通过 XSLT 文件 *smil-anchors.xsl* 对相关的 SMIL 文档进行过滤。

Feed Reader 的出现

在我实现 *macspeak-websearch* 和 *emacspeak-url-template* 这两个模块时，Emacspeak 需要处理 HTML 网页来朗读相关信息。但随着网页复杂性的增长，快速获得页面外观之下真实内容比实现免视访问的功能更有价值。即使那些能够应付复杂可视界面的用户也发现他们自己受到严重的信息负载困扰。这就导致了 RSS 和 Atom Feed 等技术的出现，并带来了 Feed Reading 软件。

这些开发对于 Emacspeak 代码库有着非常积极的作用。在过去几年中，随着我逐渐地删除那些复杂的逻辑并且用直接内容访问来取代它们，代码变得越来越漂亮了。下面是一个为指定城市 / 州提取天气的 URL 模板：

```

(emacspeak-url-template-define
来自 wunderground 的 RSS 天气信息。"
"http://www.wunderground.com/auto/rss_full/%s.xml?units=both"
(list "State/City e.g.: MA/Boston") nil

```

为指定的州 / 城市获取 RSS 天气信息。"

```
'emacspeak-rss-display)
```

以下是在通过 Atom Feed 搜索 Google 新闻的 URL 模板：

```
(emacspeak-url-template-define
  " Google 新闻搜索"
  "http://news.google.com/news?
hl=en&ned=tus&q=%s&btnG=Google+Search&output=atom"
  (list "Search news for: ") nil "Search Google news."
  'emacspeak-atom-display )
```

在这两个工具中都使用了 emacspeak-url-template 模块提供的功能，而它们本身只需做非常少的工作。最后，我们应注意到使用标准化的 Feed 格式，例如 RSS 和 Atom 等，这些模板与站点特定的规则关联很少，而在一些老的工具，例如 Yahoo! Maps 向导，中则包含来自结果页面的特定规则进行了硬编码。

小结

Emacspeak 在构思时是一个功能完备的、能够用于日常工作的免视用户界面。为了实现完备的功能，系统需要为桌面工作站上计算任务各个方面都提供直接访问。为了实现流畅的免视交互，系统需要把语音输出和听觉媒介视为系统中的一等公民——也就是说，仅仅朗读在屏幕上显示的信息是不够的。

为了提供全功能的音频桌面，目标环境需要是一个交互框架，既可以被广泛地部署，也可以被完全地扩展。为了能够实现比朗读屏幕更多的功能，系统需要把交互式的语音能力构建到不同的应用程序中。

最后，在实现这个功能时不能修改任何底层应用程序的源代码；这个项目无法为了增加免视交互的能力而对一组应用程序进行修改，因为我希望所有的工作仅限于维护语音扩展的。

为了满足所有这些设计需求，我选择了 Emacs 作为用户的交互环境。作为一个交互框架，Emacs 的优势在于它拥有大量的开发人员群体。与我在 1994 年开始这个项目时的其他交互框架不同，它的重要优势之一就是它是一种免费的软件环境（现在，12 年过去了，Firefox 提供了类似的机会）。

在各种应用程序中实现语音功能时，Emacs Lisp 作为一种可扩展语言，它所提供的巨大灵活性是一个重要的先决条件。这个平台的开源特性也是很重要的；虽然我决定不去修改任何源代码，但分析各种应用程序的实现方式将更容易使这些程序支持语音。最后，

Emacs Lisp (注意, Lisp 中的 *advice* 功能是在面向方面编程 (Aspect Oriented Programming) 中最主要的推动力) 中对于 *advice* 的高质量实现使得用 Emacs Lisp 编写的应用程序在实现语音时无需修改源代码。

Emacspeak 是把前面描述的需求与 Emacs 作为用户交互环境所提供的功能结合起来的结果。

长期以来管理代码的复杂性

Emacspeak 代码已经发展了 12 年的时间了。除了第一次开发花了 6 个星期之外, 代码库的开发和维护都是在 Emacspeak 中来完成的。在本节中将总结在管理代码复杂性时学到的一些经验。

在 Emacspeak 的开发过程中, 它始终是作为一个业余时间的开发项目。看着代码库经历了这么长的时间, 我相信这对它今后的发展有着重要的影响。当全职开发一个大型并且复杂的系统时, 开发人员能够在一段时间内把他的所有注意力都集中在代码库上——例如, 6 到 12 个星期。这就导致了创建深代码库的紧密开发代码。

尽管开发人员有着美好的愿望, 但随着时间的过去, 代码很可能变得难以阅读。那些只有一位工程师长期投入于某个项目的大型软件系统几乎是不存在的; 这种忠诚的关注通常会很快地消耗殆尽!

与之相反, Emacspeak 是一个只有一位工程师在这上面工作了 12 年的大型软件系统, 但仅在他的业余时间里。多年来独自开发这个系统的结果就是, 代码库自然而然地是“独立的”。注意, 在表 31-1 中总结了代码行数和文件的分布情况。

表 31-1: Emacspeak 代码库的总结

层	文件数	行数	所占百分比
TTS 内核	6	3866	6.0
Emacspeak 内核	16	12174	18.9
Emacspeak 扩展	160	48339	75.0
总计	182	64379	99.9

在表 31-1 中强调了以下要点:

- 负责高质量语音输出的 TTS 内核分布在 182 个文件的 6 个文件中, 并且占代码库总数量的 6%。

- Emacspeak内核——为Emacspeak扩展提供了高级语音服务，此外还使得所有基本的Emacs功能都支持语音——分布在16个文件中，并且占代码库总数量的19%。
- 系统的其他部分分布在160个文件中，对这些文件可以单独进行修改（或者拆分）而不会影响系统的其他部分。许多模块，例如`emacspeak-url-template`——本身就是茂密的——也就是说，可以在修改某个URL模板不会影响到任何其他的URL模板。
- *advice*减少了代码量。在Emacspeak代码库中大约有60 000行Lisp代码，这与底层需要支持语音的系统代码数量相比是非常小的。在2006年12月末的统计显示，Emacs22有着大约1百万行Lisp代码；此外，Emacspeak还使得大量其他的应用程序也支持语音，而这些程序在默认情况下并没有捆绑到Emacs中。

结论

以下是在开发和使用Emacspeak中的一些心得：

- Lisp *advice*及其在面向方面编程中的类似技术，是实现横切关注点（Cross-cutting concern）的有效方式——例如使视觉界面支持语音。
- *Advice*是一种在复杂软件系统中发现潜在的扩展点的功能强大的方法。
- 关注基本的网页架构并且依靠由标准的协议和格式所支撑的面向数据的网络，它带来了功能强大的语音网页访问。
- 关注最终用户的体验，这与像滑块和树型控件这样的独立交互widget控件不同，它带来了高效并且免视的环境。
- 视觉交互非常依赖人眼快速扫描视觉显示的能力。高效的免视交互需要把这种能力中的一部分转移到计算机上，因为听取大量的信息是非常耗时的。这样，在每种形式中进行搜索对于提供有效的免视交互来说就是非常重要的，这包括从最小量级（例如Emacs的增量搜索，用来在本地查找正确的文档）到最大量级（例如Google搜索，用来在全球Web上快速查找正确的文档）的搜索。
- 视觉复杂性可能只会刺激用户使用复杂视觉交互，它是实现免视交互的阻碍之一。当复杂视觉界面所带来的坏处超出了一定限度时，在早期免视环境中出现的工具软件最终会成为主流软件。下面给出了在Emacspeak中获得的两个经验：
 - RSS和Atom Feed代替了网页内容查找，这样在检索像文章标题这样的基本信息时无需查找所有网页内容。

- Emacspeak 在 2000 年使用了 XSLT 来过滤内容，这与在 2005 年出现的将自定义客户端 JavaScript 应用于 Web 的 Greasemonkey 技术是同等重要的。

致谢

如果没有 Emacs 以及在 Emacs 中实现各种功能的开发人员群体，那么 Emacspeak 将不会存在。如果没有 Hans Chalupsky 对 Emacs Lisp 中 *advice* 的漂亮实现，那么 Emacspeak 的实现也是不可能完成的。

来自 GNOME 项目的 libxslt 项目将全新的思想引入到 William Perry 的 Emacs W3 浏览器中；Emacs W3 是最早的 HTML 表现引擎之一，但它的代码有 8 年没有更新了。W3 的代码库现在仍然是可用的和可扩展的，这充分说明了 Lisp 作为一种实现语言所提供的灵活性和强大功能。

第 32 章

变动的代码

Laura Wingerd & Christopher Seiwald

要点在于每个成功的软件都有着延伸的生命期，许多程序员和设计师
都在不断地工作于这个软件上……

Bjarne Stroustrup

在本书策划的早期，Greg Wilson 向所有投稿者征询《代码之美》是否是一个合适的书名。“你们将要讨论的东西大多数是关于软件的设计和架构，而不是单纯的代码”，他这样写到。

不过本章的内容是关于代码的。本章将要讨论的既不是代码能做什么，也不是代码有多么优美，而是关于代码的外观：如何通过编码中的一些人类视觉特性来实现串行协作。因此，本章的内容是关于“变动的代码”的漂亮性的。

你将阅读的内容大部分是借鉴了 Christopher Seiwald 的文章，《漂亮代码的七个原则》(The Seven Pillars of Pretty Code)”（注 1）。简而言之，这七个原则分别是：

- 像书本一样。
- 功能相似的代码在外观上也保持相似。

注 1：可以在 Perforce 网站上阅读这篇文章：<http://www.perforce.com/perforce/papers/prettycode.html>.

- 减少缩进。
- 分解大型的代码块。
- 对代码进行注释。
- 整理代码。
- 与已有的编码风格保持一致。

虽然这些规则听起来像是一些编码习惯，但它们的含义并不仅限于此。它们是让你记住产品演化性的编码习惯的外在表现。

在本章中，我们将看到通过这个七个原则如何来支持一段在某个商业软件系统中已经存在了10多年的代码。这段代码是DiffMerge，即Perforce软件配置管理系统的组件之一。DiffMerge的任务是执行经典的三路合并（three-way merge），把文本文件的两个版本（leg 1 和 leg 2）与参看版本（基准版本）进行比较。输出结果将通过在输入文件中插入占位符来标出有冲突的行。如果你用过Perforce，那么就应该在p4的resolve命令和Perforce的图形化合并工具中见过DiffMerge的使用。

DiffMerge最初编写于1996年。尽管它的意图很简单，但三路文本合并的功能是很复杂的。它包括了来自于用户界面，字符编码，编程语言以及程序员自身等的特殊情况（“这不是一个冲突。”“不，这是一个冲突。”“是的，这不是一个冲突。”）。

几年以来，Diffmerge成为了Perforce软件中的重要开发模块之一。因此，如果说DiffMerge只是一段正确的代码，那是不够的。它是与我们用于编码、调试和变更管理的工具软件“和谐工作”的一段代码。此外，它也是一段希望得到修改的代码。

从DiffMerge的最初实现到它今天的形式，其所经历的道路是不平坦的。如果我们越背离这七个原则，那么这条道路就会变得更加崎岖，这并不是一种偶然。在本章的后面，我们将揭示一些在DiffMerge 10余年发展过程中的一些问题（包括一个非常严重的问题）。

不过，虽然困难重重，但结果还是不错的。现在的DiffMerge，位于<http://www.perforce.com/beautifulcode>上，是稳定的并且很容易得到增强。它证明了如果在编码时考虑将来可能发生的修改，那么就能够产生一段漂亮的变动代码。

像书本一样

在“漂亮代码的七个原则”（注 2）一文中描述了我们在 Perforce 软件中使用的规则。我们使用的并不仅仅只有七个原则，而它们也没有被应用于我们所有的开发项目。我们把这七个原则应用到像 DiffMerge 这样的组件，在这些组件中，多个同时发布版本中的相同代码并没有被冻结，许多程序员都可以进行修改。这七个原则的效果就是使程序员进一步理解他们正在阅读的代码，并且能够提供更多的上下文信息。

例如，我们来看看七个原则中的“像书本一样”这个原则。书本和杂志中的文本都是按专栏来组织的，通常专栏的宽度要远窄于页面宽度。为什么？因为狭窄的专栏宽度可以使我们在阅读的时候减少眼睛来回扫描的工作——当眼睛的扫描工作越少时，阅读就会变得越容易。当刚刚阅读的内容和将要阅读的内容都位于我们的视野范围之内，那么阅读就会变得更加容易。研究指出，当我们眼睛的视力焦点从一个单词转向另一个单词时，我们的大脑将从周围视力焦点之外的文字中获取提示信息。如果我们的大脑能够从视觉范围之外的文本中获得更多“预知信息”，那么就能够更好地指导我们的眼睛去获得最大程度的理解能力。

研究还指出，阅读速度和阅读理解程度在文本行的长度上存在一个差异。越长的文本行读起来越快，而越短的文本行则越容易理解。

分块的文本比连续的文本要更容易理解。这就是为什么在书本和杂志中的文本专栏被拆分为一个个段落。段落、短句、表格、侧栏文章和脚注等都是文本的“阅读标记”，这是告诉我们的大脑，“你是否已经理解了目前所有的东西？如果理解了，请继续”。

当然，代码不是严格意义上的文本，但为了更好地使人阅读，可以应用相同的规则。像书本一样的代码——也就是说，像文本专栏和文本块一样格式化的代码——更容易被理解。

借鉴书本的排版风格并不限于使代码行保持简短。我们来看看下面这两段代码之间的差异：

```
if( bf->end == bf->Lines() && lf1->end == lf1->Lines() &&
    lf2->end == lf2->Lines() ) return( DD_EOF );
if( bf->end == bf->Lines() &&
    lf1->end == lf1->Lines() &&
```

注 2：我们在工作中并没有把它们叫作“七个原则”。事实上，我们认为它们可以脱离于特定的语言编码原则或者模块编码原则。然而，当我们去掉与语言或者模块相关的内容时，剩下的东西就是“七个原则”。

```
lf2->end == lf2->Lines() )  
    return( DD_EOF );
```

第二段代码来自于 DiffMerge。当我们阅读这段代码时，我们的大脑将感知所看到的逻辑，并且我们的眼睛也不需要很长的扫描距离来获得这些信息（由换行位置的选择而产生的视觉模式同样是重要的，我们很快就会理解这点）。由于第二段代码的风格更像书本中的排版风格，因此第二段代码比第一段代码要更易于理解。

功能相似的代码在外觀上也保持相似

在前面章节的 DiffMerge 代码中还说明了编写代码时提高可理解性的另一个规则：功能相似的代码从外观上来看也要很像。我们在 DiffMerge 代码的各个地方都可以看到这个规则。例如：

```
while( d.diffs == DD_CONF && ( bf->end != bf->Lines() ||  
                                lf1->end != lf1->Lines() ||  
                                lf2->end != lf2->Lines() ) )
```

前面的代码说明了如何通过换行符来产生一个视觉模式，从而使我们的大脑更容易识别出逻辑模式。我们只需一瞥就可以知道，在上面 while 语句的四个判断中，有三个判断基本上都是相同的。

下面是这条规则的另一个示例。在这个示例中说明了如何在编码时使我们的大脑成功地识别出“其中一个不相同的情况”：

```
case MS_BASE: /* 转储原始文件 */  
    if( selbits = selbitTab[ DL_BASE ][ diffDiff ] )  
    {  
        readFile = bf;  
        readFile->SeekLine( bf->start );  
        state = MS_LEG1;  
        break;  
    }  
  
case MS_LEG1: /* 转储 leg1 */  
    if( selbits = selbitTab[ DL_LEG1 ][ diffDiff ] )  
    {  
        readFile = lf1;  
        readFile->SeekLine( lf1->start );  
        state = MS_LEG2;  
        break;  
    }  
  
case MS_LEG2: /* 转储 leg2 */  
    if( selbits = selbitTab[ DL_LEG2 ][ diffDiff ] )
```

```
    {
        readfile = lf2;
        readfile->SeekLine( lf2->start );
    }

    state = MS_DIFFDIFF;
    break;
```

即使你并不知道这段代码的功能是什么，但仍然可以很清楚地看到，例如，虽然在所有的三种情况中都设置了 `readfile` 和 `state`，但只有第三种情况中的 `state` 是被无条件设置的。编写这段代码的程序员使用了“功能相似的代码在外观上也保持相似”的规则；这样，将来阅读这段代码的程序员可以很快地知道基本的逻辑是在哪里。

缩进带来的危险

我们已经学习过在代码逻辑块中使用缩进来显示嵌套的深度。嵌套得越深，被嵌套的代码就越靠近页面的右边。按照这种方式来格式化代码是不错的主意，但并不是因为缩进能够使代码更容易阅读。

深度缩进的代码将难以阅读。所有的重要逻辑都拥挤在右边，几乎被淹没了——就像代码周围的数层 `if-then-else` 使代码变得无意义，而外层代码块中不太重要的条件判断却看上去好像变得更重要了。因此，虽然缩进在标识代码块的起始和结尾时是很有用的，但它并不会有助于对代码的理解。

更大的危险性并不在于缩进，而是在于嵌套。嵌套的代码将会破坏人们的理解力。在 Edward Tufte 写下“有时候，[PowerPoint] 子弹的层次结构非常复杂，并且高度嵌套，就像计算机代码一样”这句话的时候，他对嵌套并不是持赞成态度的。在《Code Complete》一书（参见本章末尾的“进一步阅读”部分）中，Steve McConnell 提醒程序员少用嵌套的 `if` 语句——并不是因为它们是低效或者起不到作用的，而是因为它们对于人们的理解来说是很困难的。“为了理解代码，你不得不同时记住所有的嵌套 `if` 语句”，他说。因此不难理解在有些研究中所指出的，在所有的编程结构中，嵌套的条件判断是最容易产生错误的结构。对于这个观点，我们有一些有趣的证据，你在阅读“DiffMerge 的曲折历史”这一节时将会看到。

因此，对每个嵌套层次进行缩进的价值并不在于使代码更容易理解，而是使得程序员更加清楚嵌套所带来的难以理解性。“七个原则”建议编码人员“减少缩进”——也就是说，在编码时不要使用过深的嵌套。“这是在这些编程习惯中最难掌握的”，Seiwald 承认，“它所需要的技巧是最多的，并且经常会影响单个函数的实现”。

Steve McConnell 在《代码大全》“Taming Dangerously Nesting”一节中给出了一些有用的实现。在 DiffMerge 中大量使用了其中的两个实现：case 语句和决策表。你可以在 DiffMerge 源代码中看到，减少缩进的结果就是使代码本身呈现出一种轮廓形式，使得我们能够只需扫描页面左下角的代码就可以理解大致的逻辑。由于我们的大脑习惯于阅读自然语言文本的轮廓，因此这种编写形式比深度嵌套代码所带来的“V”字形轮廓要更易于理解。

浏览代码

在 DiffMerge 中对所有七个原则都进行了或多或少的说明。例如，DiffMerge 代码是由离散的逻辑代码块所构成的。每个代码块都只执行单个功能或者单一类型的功能，并且这个功能要么是一目了然的，要么在代码的注释中进行了说明。这种代码对应的正是七个原则中的“分解代码块”和“注释代码块”这两个原则。这非常像那些结构明晰的说明文，在文中定义了目录和标题章节来帮助读者在大量的技术信息中进行浏览。

在 DiffMerge 代码中不存在混乱，这一点同样使得代码更易于浏览。减少 DiffMerge 混乱性的技巧之一就是对代码中重复引用的变量使用简单的名字。但这却违背了对变量使用有意义的和描述性的名字这一规则。但有一点需要注意，过度使用长名字将带来过多的混乱，从而影响程序员理解代码。作家们都清楚这点；这就是为什么我们在散文中经常使用代名词、姓和缩写。

DiffMerge 中的注释同样不存在混乱。在历经 10 余年编写与修改的代码中，通常很容易产生大量描述这段代码以前功能的注释——包括关于对代码修改的注释——以及描述代码当前功能的注释。但我们没有理由在代码中为程序的发展过程维护连续的历史记录。在源代码控制管理系统中已经包含了所有的信息，并且提供了更友好的方式来跟踪历史记录（我们将在后面给出示例）。在 DiffMerge 上工作的程序员很好地保持了注释的整洁性。对于代码也是如此。在 DiffMerge 中，旧的代码并不是被简单地注释掉，而是被彻底删除。剩下的代码和注释都是非常整洁的（注 3）。

在 DiffMerge 的代码中还使用了大量的空格。空格除了能够减少程序外观上的混乱性之外，还能够提高代码的可理解性。当用空格来分隔像书一样的代码块和相似的模式时，代码就会呈现出我们大脑能够识别的视觉形状。随着我们不断地阅读代码，我们的大脑将会对这些形状建立索引；然后就会下意识地通过这些形状来找到已经阅读过的代码。

注 3： 在 DiffMerge 中仍然有一条注释记述了代码变更的历史：“2-18-97 (seiwald) - translated to C++.” 不过，这条注释在代码中只是作为一种历史纪念。

虽然多年以来，许多程序员一直都在反复地修改这段代码，但 DiffMerge 代码在很大程度上与其视觉外观是保持一致的，每个 DiffMerge 的开发人员都努力地“与已有的编码风格保持一致”。也就是说，每个人都克服了个人的风格和偏好，从而使他编写的 DiffMerge 代码与其他的 DiffMerge 代码看上去是一样的。这种做法带来了代码的一致性，从而减少了我们大脑的工作。这个原则有效地增强了我们刚才讨论过的所有可读性技巧。

如果你访问过 <http://www.perforce.com/beautifulcode>，那么你将注意到 DiffMerge 的代码并不是完美的，即时按照七个原则的标准来看也是如此。确实，我曾经想要对这些代码进行整理，虽然我们喜欢漂亮的代码，但我们更喜欢干净的代码合并。对于变量名、空格、换行符等的修改可能是比逻辑修改更难以合并的。当我们在某个分支中进行这样的修改时，就会带来从其他的分支将 bug 修正合并过来将会产生 bug 的风险。通过改写 DiffMerge 中不好看的代码而获得的好处，都被用于恢复不良代码合并所消耗的资源抵消了。在“DiffMerge 的曲折历史”一节中，我们将讲述其中发生的一些故事。

我们使用的工具

当我们直接阅读源文件的时候，当然希望代码是可理解的。我们同样需要这段代码在差异比较、代码合并、编写补丁、调试器中、代码审查、编译器消息中以及其他许多环境和工具中也是可理解的。事实证明，在编写代码时，如果时刻记住这七个原则，那么编写出的代码将在大多数用来管理代码的工具中均是可读的。

例如，即使没有语法高亮显示的支持，DiffMerge 代码也是可读的。换句话说，我们并不需要依靠那些上下文敏感的源代码编辑器来阅读这段代码。正如在调试器、编译器以及网页浏览器中用普通文本来显示代码时，同样是可读的。以下是在 gdb 文件中的一部分 DiffMerge 代码：

```
Breakpoint 3, DiffMerge::DiffDiff (this=0x80e10c0) at diff/diffmerge.cc:510
510          Mode initialMode = d.mode;
(gdb) list 505,515
505          DiffGrid d = diffGrid
506          [ df1->StartLeg() == 0 ][ df1->StartBase() == 0 ]
507          [ df2->StartLeg() == 0 ][ df2->StartBase() == 0 ]
508          [ df3->StartL1() == 0 ][ df3->StartL2() == 0 ];
509
510          Mode initialMode = d.mode;
511
512          // 预处理规则，通常外部曲线信息与内部曲线信息
513          // 是相互矛盾的，虽然不完美，
514          // 但我们可以使用曲线的长度来决定最佳输出。
515
```

```
(gdb) print d
$1 = {mode = CONF, diffes = DD_CONF}
(gdb)
```

在经常修改像 DiffMerge 一样的代码时（自从 DiffMerge 首次开发以来，已经修改了 175 次），程序员都会花大量时间在差异比较和合并工具中查看这段代码。这些工具的共同之处在于它们限制了源代码的水平视图，并且增加了它们自己的一些混乱。即使在这种条件下，像 DiffMerge 这样的代码仍然是可读的。在命令行的比较工具中，代码行并不会发生回卷。在图形化的比较工具中，我们也不需要调整水平滚动条来看到一行代码的完整内容，如图 32-1 所示（注 4）。

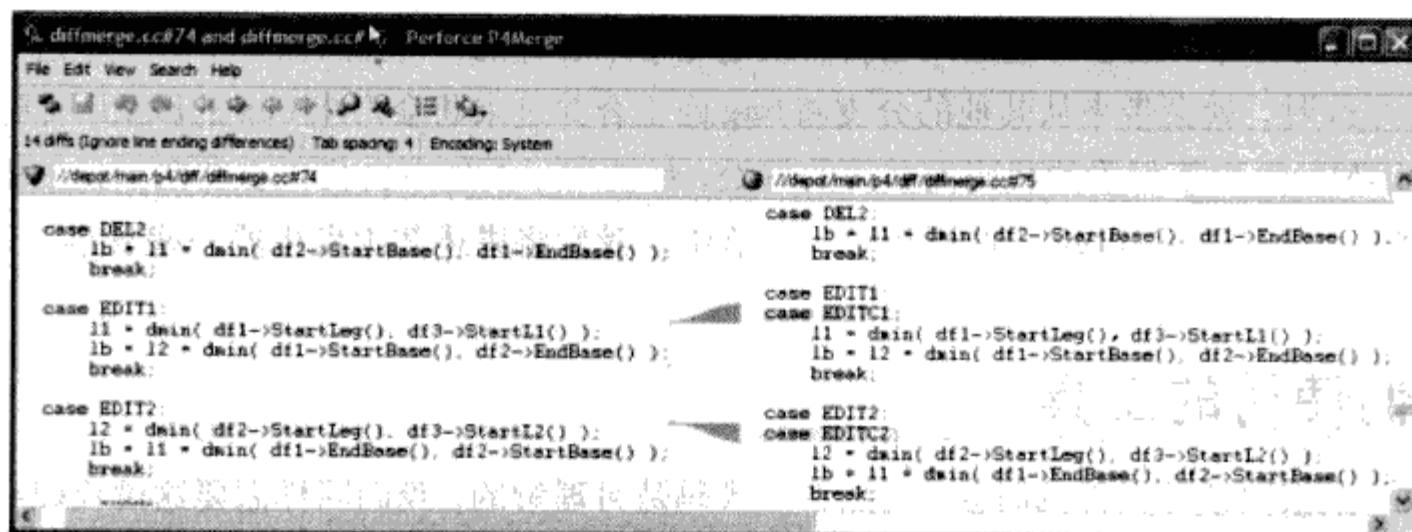


图 32-1：在图形比较工具中的 DiffMerge 代码

如图 32-2 所示，当显示像书本一样的 DiffMerge 代码时，在页面边界不足的标注历史浏览器中也是足以容纳一行完整代码。

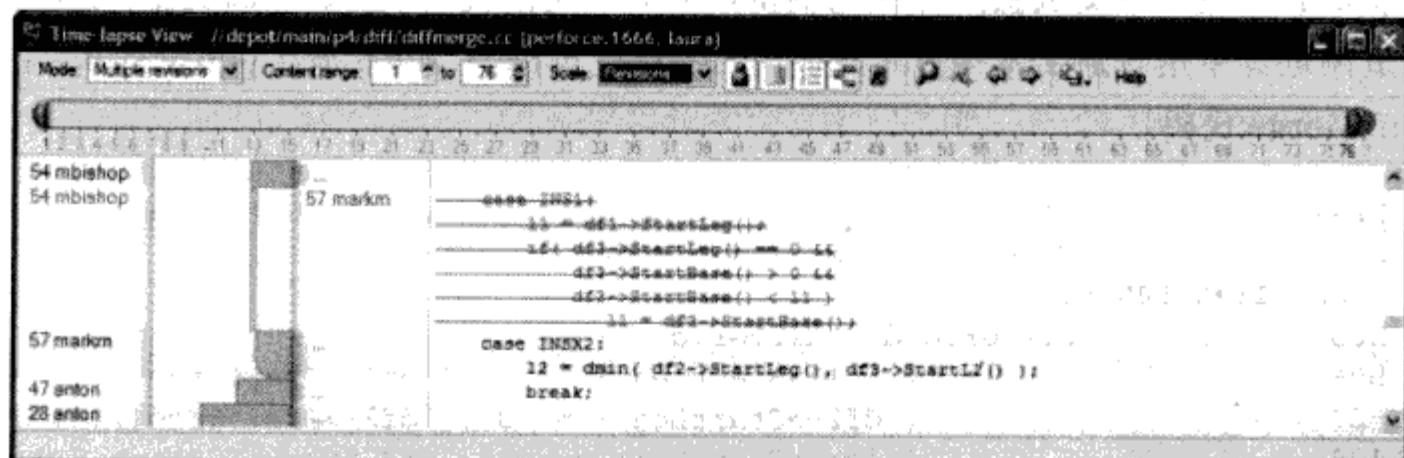


图 32-2：在标注历史浏览器中的 DiffMerge 代码

像书本一样的风格不仅使代码在合并工具中更加可读，而且还能够使代码本身更容易合

注 4：这是基于 DiffMerge 本身构建的图形工具 P4Merge 的界面截图。

并。一方面，当用空格来分隔逻辑块时，编辑的范围更容易得到控制。另一方面，嵌套越少的代码有助于减少错误的和遗漏的代码块分隔符。

DiffMerge 的曲折历史

在 DiffMerge 的 10 余年发展历史中，我们记录下了代码的每次修改、分支和合并。这是一个有趣的记录。在图 32-3 中给出了在 DiffMerge 每个发布版本中所进行的修改的概略视图。这张图给出了 DiffMerge 从主线开始（图中最底部的线），已经分出了 20 多个版本。对于 DiffMerge 的大部分修改都是在主线中进行的。从这个缩略图中我们可以看出在最近发布版本中的一些特殊动作。

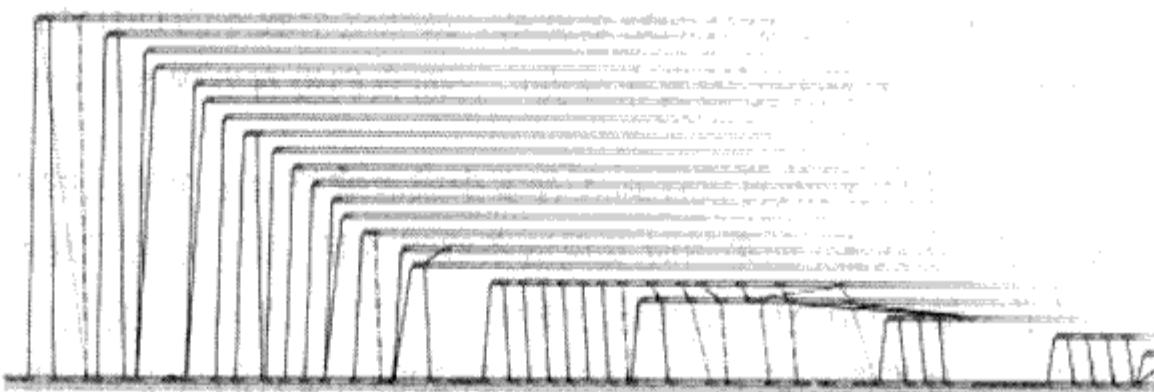


图 32-3：DiffMerge 的发布分支

图 32-4 中给出了每个发布分支中 DiffMerge 的补丁，这或许是更有意思的。从这张图中可以看出，DiffMerge 在发布之后很少需要打补丁——直到 2004 年 2 月的发布。在随后的发布中，程序的补丁喷涌而出，直到在 2006 年 2 月的发布中再次减少。为什么在 2004 年 2 月到 2006 年 2 月中产生了如此之多的补丁？

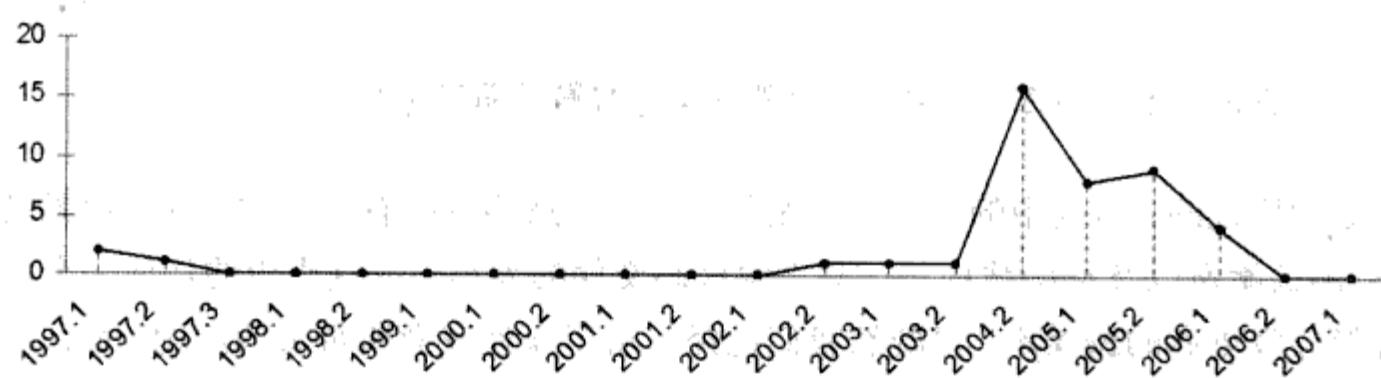


图 32-4：在每个发布版本中 DiffMerge 的补丁数量

幕后的故事是这样的：DiffMerge 在最初的时候只是个有用但却简单的程序。在早期，DiffMerge 在区别真实的合并冲突和非冲突的邻接行修改上只做了少量的工作。在 2004

年，我们对DiffMerge进行了增强，从而使它在检测和解析冲突时更为智能。在2004年2月的发布版本中，DiffMerge的功能当然变得更为强大，但带来的错误也增多了。我们在2004年2月到2005年1月期间得到了大量的bug报告——因此就带来了最大数量的补丁。我们曾经试图在2005年2月的发布中清除一个bug，但这个清除工作最终又导致了一个非常难以处理的bug，以至于不得不把2005年1月的发布版本恢复到2005年2月的发布版本中。然后，在2006年，我们彻底重写了DiffMerge中有问题的部分。这次重写是非常成功的，不过我们只能在2006年1月发布的版本中集成这次重写。从那以后，DiffMerge变得非常稳定，并且发布后的补丁数量重新降为零。

那么，我们在2004年重新编写DiffMerge的时候出现了什么问题？我们相信这是因为降低了代码的可理解性。这可能是在代码审查的时候忽略了上面提到的七个原则，也可能我们完全省略了一些审查工作。最终的结果就是，虽然这个版本的DiffMerge通过了回归测试，但却进入到一个充满bug的发布分支中，而我们没有发现其中的问题。

我们无法定量地来计算源代码的可读性到底有多少或者它符合七个原则的程度。但是回头看来，我们发现了一条线索，如果当时发现这条线索的话，那一定会引起我们的注意。在图32-5中给出了在每个DiffMerge初始分支中的一些if语句，以及它们的嵌套层数（这是从它们缩进的深度推断出的）。当我们在进行2004年2月的分支时，很显然在代码中有了双倍数量的if语句，而且if语句的嵌套深度首次超过3层。

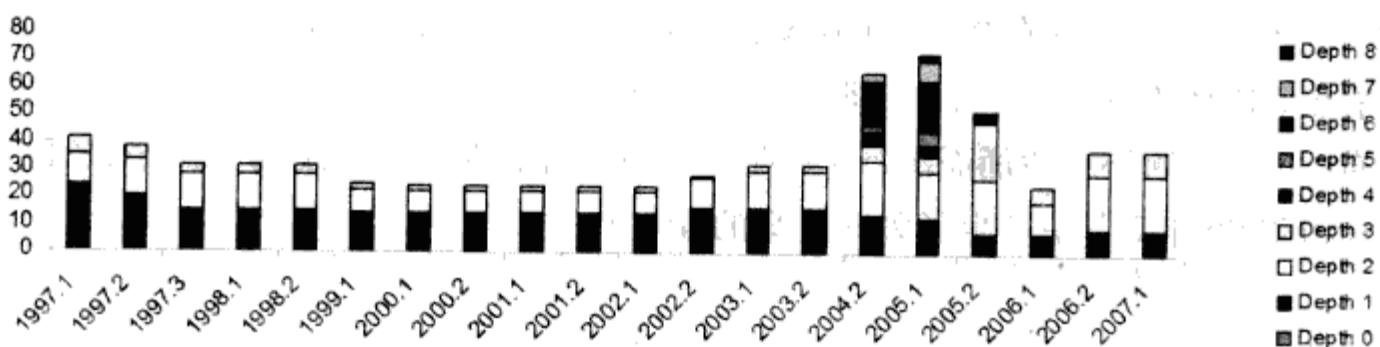


图32-5：在每个发布版本中DiffMerge的if语句数量及缩进深度

正如他们所说的，这其中的关联并不是偶然的，当然还可能有其他的因素。增强功能的设计、测试案例以及编码结构——甚至源代码文件的大小——这当中的任何一个因素都有可能导致更高的错误率。但既然我们已经了解了深度嵌套所带来的问题，那么就很难把这个明显的关联只作为次要因素。

在2006年，一位权威人士推动了对DiffMerge的彻底检查以减少缩进。在这次检查中，用switch语句来替换深度嵌套的条件语句，其中的case选项是定义在新的diffGrid决策表中的值。这张表的布局是可读的，它把我们当前遇到的所有条件都用

表中的项来表示，并且为将来我们想要判断的条件也预留了位置。这样，我们不仅解决了有问题的代码，而且还为将来的增强功能预留了空间。

结论

对于在变动的代码上工作的程序员来说，漂亮性在于代码的修改只需最少的工作来完成。你需要首先阅读代码，确定代码的功能，然后再来修改代码。你的成功依赖于在开始修改的时候对代码的理解程度，这和你的编码能力是类似的。此外，是否成功还要取决于下一个修改这段代码的程序员对你的修改的理解程度；如果他们从没有打电话向你寻求帮助，那么你做得就是非常好的。

如果我们从本章中去掉一些叙述性的文字，那么所剩下的要点就是成功地变动代码依赖于阅读代码的程序员对它的理解程度。但这对于任何人来说都不是新的东西。

新的东西在于程序员在阅读差异比较、开发补丁、代码合并、编译器错误和调试器中的代码时——不仅仅是在用颜色表示不同语法的文本编辑器中——他们经常能在不知不觉中从代码的外观推断出代码的逻辑。换句话说，理解代码比满足眼睛的需求更重要。

在本章中，我们已经证明了把“漂亮代码的七个原则”作为指导思想来提高代码在更多环境下的可理解性是行之有效的。我们已经通过这七个原则取得了成功。我们已经把它们用来编写可以随着修改而变动的代码，并且认为这样的代码是漂亮的。

致谢

Christopher Seiwald, James Strickland, Jeff Anton, Mark Mears, Caedmon Irias, Leigh Brasington 和 Michael Bishop 在开发 DiffMerge 中作出了很大的贡献。DiffMerge 源代码的版权归 Perforce Software 所有。

进一步阅读

Kim, S., Adaptive Bug Prediction by Analyzing Project History, Ph.D. Dissertation, Department of Computer Science, University of California Santa Cruz, 2006.

McConnell, S., *Code Complete*, Microsoft Press, 1993.

McMullin, J., Varnhagen, C. K., Heng, P., and Apedoe, X., "Effects of Surrounding Information and Line Length on Text Comprehension from the Web," *Canadian Journal of*

Learning and Technology, Vol. 28, No. 1, Winter/hiver, 2002.

O'Brien, M. P., Software Comprehension - A Review and Direction, Department of Computer Science and Information Systems, University of Limerick, Ireland, 2003.

Pan, K., Using Evolution Patterns to Find Duplicated Bugs, Ph.D. Dissertation, Department of Computer Science, University of California Santa Cruz, 2006.

Reichle, E.D., Rayner, K., and Pollatsek, A., The E-Z Reader Model of Eye Movement Control in Reading: Comparisons to Other Models, *Behavioral and Brain Sciences*, Vol. 26, No. 4, 2003.

Seiwald, C., "The Seven Pillars of Pretty Code," Perforce Software, 2005, <http://www.perforce.com/perforce/papers/prettycode.html>.

Tufte, Edward R., *The Cognitive Style of PowerPoint*, Graphics Press LLC, 2004.

Whitehead, J., and Kim, S., Predicting Bugs in Code Changes, Google Tech Talks, 2006.

Wickens, C. D., *Elementary Principles of Human Factors Engineering*, 2nd ed., Prentice Hall, 1992.

Wickens, C. D., *Human Factors in Design*, 2nd ed., Cambridge University Press, 1992.

Wickens, C. D., *Information Processing and Human Performance Theory*, 2nd ed., Lawrence Erlbaum Associates, 1992.

Wickens, C. D., *Human Factors in Design*, 2nd ed., Cambridge University Press, 1992.

Wickens, C. D., *Elementary Principles of Human Factors Engineering*, 2nd ed., Prentice Hall, 1992.

Wickens, C. D., *Information Processing and Human Performance Theory*, 2nd ed., Lawrence Erlbaum Associates, 1992.

Wickens, C. D., *Human Factors in Design*, 2nd ed., Cambridge University Press, 1992.

Wickens, C. D., *Elementary Principles of Human Factors Engineering*, 2nd ed., Prentice Hall, 1992.

为 “The Book” 编写程序

Brian Hayes

数学家 Paul Erdős 经常会谈到 “The Book”，这是一本传说中的书（在地球上任何一家图书馆里都找不到），在这本书中记录了所有数学定理的最优证明。我想或许也存在这么一本关于程序和算法的书，在书中列出了每个可计算问题的最优解决方案。要想在这本书中占有一席之地，那么每个程序不仅必须是正确的，而且还必须是易懂的、优雅的、简洁的以及充满智慧的。

我们都在努力创造像这样的算法艺术瑰宝。然而，在程序中总会存在一些带有瑕疵的代码，无论我们如何努力都无法消除。即使这个程序能够产生正确的结果，但其中仍然存在一些勉强的和不自然的地方。这些地方的逻辑是一团杂乱的特殊情况和异常；程序的整个结构看上去非常脆弱。不过随后你突然找到了灵感，或者某个朋友向你展示了一些新的技巧，接下来你就获得了一个可以写入 “The Book” 的程序。

在本章中，我将讲述一个关于这种努力的故事。这个故事有着完美的结局，不过我会让读者来决定最终版本的程序是否能够在 “The Book” 中占有一席之地。我不会暗示任何可能性，而只是讲述在这个故事中，关键的思想并不是来源于我自己，而是来源于另一块大陆上一位朋友。

没有捷径

我将要探讨的程序来自于计算几何学领域，这是一个充满难题的领域，有些难题初看上去或许比较简单，但当你深入研究其中细节的时，就会发现其实是很棘手的。计算几何学是什么？它与计算机图形学不同，虽然二者是紧密相连的。在计算几何学中的算法并不适用于以像素为单位的情况，而是适用于理想化的尺规（ruler-and-compass）领域，在这个领域中点是没有大小的，而线的宽度为零，并且圆形都是十分精确的。在这些算法中，最重要的就是得出精确的结果，因为哪怕最细微的不精确性都会彻底改变计算结果。改变小数点最右边的一位数字都有可能逆转整个结果。

欧几里得曾经告诉一位高贵的学生：“学习几何学没有捷径”。在这些没有捷径的道路中，计算几何学这条路是非常泥泞的，布满了车辙并且崎岖不平。在这条路上，我们遇到的困难有时候是关于计算效率的问题：将程序的运行时间和内存消耗控制在合理的范围之内。但我在这里主要关心的并不是几何算法的效率问题，而是概念上和美学上的挑战。我们能不能把算法写正确？我们能不能使算法变得漂亮？

下面我将给出同一个程序的几个不同版本，这个程序所要解决的是一个非常基本的问题：假定平面内有三个点，那么这三点是否共线？这是一个简单的问题，它肯定有一个简单的解答。在几个月前，当我需要一个函数来判断共线问题时（作为某个程序的一部分），这个任务看起来非常简单，因此我也就没有首先去查阅资料看看其他人是如何解决这个问题的。对于这个问题的仓促解决既没有教给我一些东西，也没有形成一些结论，但我并不感到遗憾——不过我承认这不是获得正确答案的捷径。我最终只是重复了许多人在我之前已经完成的步骤（或许这就是为什么这条道路是如此地崎岖！）。

给 Lisp 初学者的提示

我使用Lisp语言来编写代码。虽然不想因为选择这门语言而感到抱歉，但我也不会把本章变成宣传Lisp的传单。我只是相信掌握多种语言是一件好事。阅读下面的代码能够教给你一些关于某种陌生语言的知识，而且这种体验也不会给你带来坏处。所有的函数都是非常简短的——大概只有几行代码。在图33-1中给出了对Lisp函数结构的简单介绍：

顺便提一下，在这张图中的程序所实现的算法肯定会被包含在“The Book”中。这个函数是计算两个数值的最大公约数的欧几里得算法。

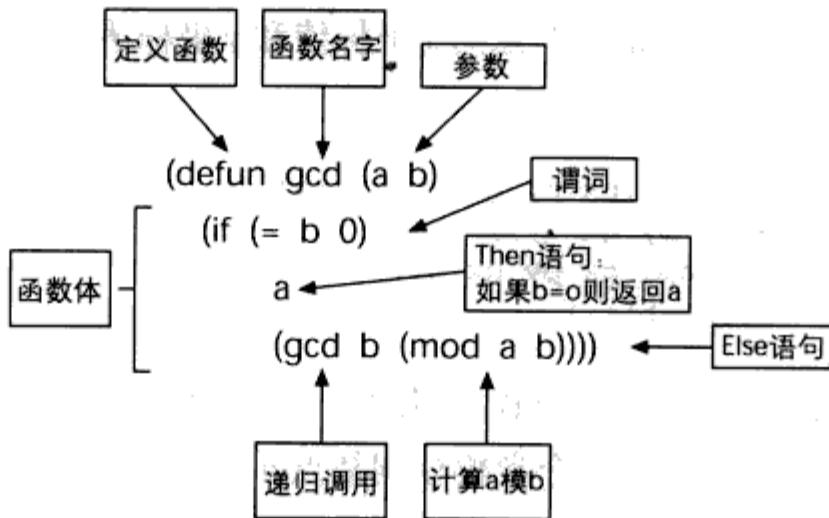


图 33-1: Lisp 函数定义中的细节说明

三点共线

如果用铅笔和纸来计算一个共线问题,你会如何开始?一种很自然的方法就是在纸上绘出三个点的位置;如果通过目测还无法判断出答案,那么在两点之间画一条线,然后看这条线是否穿过第三个点(参见图 33-2)。如果第三个点与这条线非常接近,那么绘制点和线的精度就变得非常关键。

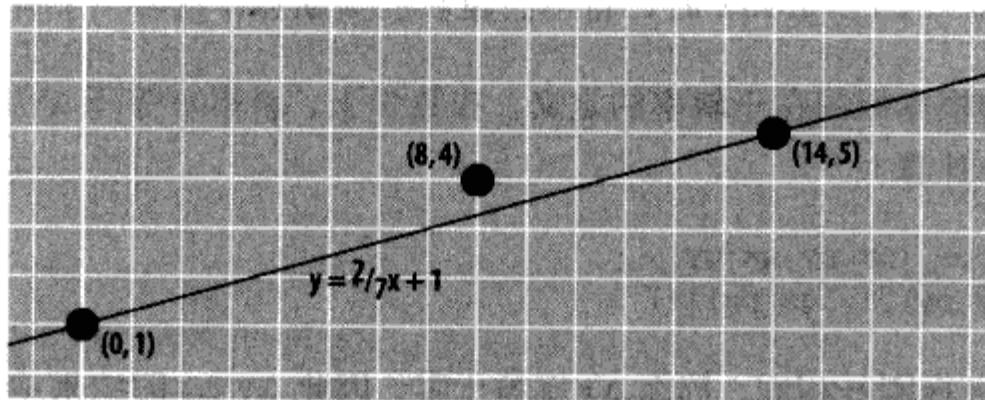


图 33-2: 三个不共线的点

虽然计算机程序可以完成同样的工作,但在计算机中无法“通过目测来进行判断”。为了在两点之间绘制一条直线,程序将计算出这条线的方程。要判断第三个点是否也在线上,程序将判断这个点的坐标是否满足这个直线方程(练习:对于任意给定的一组三个点,你可以有三种方式来选择其中的两点以连接成线,在每种方式中,剩下第三个点将用来判断共线性。某种选择方式可能会比其他的方式使这个任务更为简单,因为在这种方式中要求更少的精度。有没有简单的标准来进行这种决策)。

直线方程的形式为 $y = mx + b$, 其中 m 为斜率, b 为 y 轴截距, 即这条线与 y 轴的交点(如果存在的话)。因此,假定有三个点 p 、 q 和 r , 你可以首先算出连接其中两个点的直

线方程的 m 值和 b 值，然后用第三个点的 x 坐标值和 y 坐标值来判断是否满足同一个方程。代码如下：

```
(defun naive-collinear (px py qx qy rx ry)
  (let ((m (slope px py qx qy))
        (b (y-intercept px py qx qy)))
    (= ry (+ (* m rx) b))))
```

这是一个谓词函数：它将返回一个真或假的布尔值（按照 Lisp 语言的习惯，真和假用分别 `t` 和 `nil` 来表示）。函数的六个参数分别是点 p 、 q 和 r 的 x 坐标值和 y 坐标值。在 `let` 语句中引入了局部变量 m 和 b ，分别表示由函数 `slope` 和 `y-intercept` 返回的值。我将很快给出这些函数的定义，不过这些函数的功能从它们的名字中就可以很明显地看出来。在函数的最后一行代码中将判断是否共线：点 r 的 y 坐标是否等于 m 乘以 r 的 x 坐标再加上 b ？答案将作为函数 `naive-collinear` 的返回值。

这个算法还能变得更简单些吗？确实可以，我们马上就会看到。这个函数是否正确？在通常的情况下是正确的。如果你把这个函数用于大量随机生成点的计算中，那么函数可能会运行很长一段时间而不发生错误。然而，我们也可以很容易地使这个函数发生错误。只要用这个函数来判断 (x, y) 坐标分别为 $(0, 0)$ 、 $(0, 1)$ 和 $(0, 2)$ 的点是否共线。毫无疑问，这三个点是共线的——它们都位于 y 轴上——然而当把这三个点作为参数传递给函数 `naive-collinear` 时，将不会像我们预期那样返回一个有意义的值。

产生这个错误的根本原因在于斜率的定义。在数学上，斜率 m 等于 $\Delta y / \Delta x$ ，在程序中按照下面的方式来计算：

```
(defun slope (px py qx qy)
  (/ (- qy py) (- qx px))))
```

如果 p 和 q 的 x 坐标刚好相同，那么 Δx 就等于零，因此 $\Delta y / \Delta x$ 的结果就是未定义的。如果你继续计算斜率，那么只会得到除零错误。我们可以有许多方式来解决这个问题。在最初编写这个程序时，我所选择的方式是：当 px 等于 qx 时，`slope` 函数返回一个特殊的信号值。在 List 语言中，通常用 `nil` 值来表示这个特殊值：

```
(defun slope (px py qx qy)
  (if (= px qx)
      nil
      (/ (- qy py) (- qx px)))))
```

和斜率一样，垂线的 y 轴截距同样是未定义的，因为垂线要么不与 y 轴相交，要么与 y 轴处处相交（当 $x = 0$ ）。因此，在函数 `y-intercept` 中可以使用同样的 `nil` 技巧：

```
(defun y-intercept (px py qx qy)
  (let ((m (slope px py qx qy)))
```

```
(if (not m)
    nil
    (- py (* m px))))
```

现在，我还必须重新修改对函数的调用来处理当斜率 m 不是数值而是 nil 的情况：

```
(defun less-naive-collinear-p (px py qx qy rx ry)
  (let ((m (slope px py qx qy))
        (b (y-intercept px py qx qy)))
    (if (numberp m)
        (= ry (+ (* m rx) b))
        (= px rx))))
```

如果 m 是一个数值——如果谓词函数 (`numberp m`) 返回 `t`——那么我将像前面一样来进行处理。否则，我就知道 p 和 q 的 x 坐标是相同的。因此，如果 r 也有着相同的 x 坐标值，那么这三点就共线。

随着程序的发展，对垂线的特殊处理变成了一件令人恼火的事情。这个函数开始变得像我以前编写的那些有着难看补丁的函数了，在这些函数中需要对平行于 y 轴的直线进行单独处理。诚然，这个补丁只是一个 1 到 2 行代码的 `if` 表达式，并且它也不是一个主要的软件工程问题。不过从概念上来看似乎存在着不必要的复杂性，这意味着我可能做错了某件事情或者使事情比原本变得更加复杂了。垂线和水平线与那些任意角度的直线并没有本质上的区别。根据 y 轴来计算斜率只是其中的一种习惯，如果我们采用不同的思考方式，那么这些情况就不会有区别。

有一种绕过这个问题的方式：旋转坐标系。如果在某个坐标系中一组点是共线的，那么这些点在其他坐标系中也必定是共线的。按照某种方式以一定的角度来旋转坐标轴，从而消除除零情况。旋转坐标系的算法并不困难，而且计算代价也不高，只需一个矩阵乘法。不过，采用这种方法就意味着我仍然要在函数的某个地方编写 `if` 表达式来判断 px 是否等于 qx 。而我真正想要做的事情就是简化这个逻辑并且彻底地去掉 `if` 分支。有没有其他某种方式能够基于对点坐标的简单计算来判断共线性，而无需考虑任何特殊的情况呢？

以下是在某个网站上推荐的解决方案（使用的环境稍有不同），我在这里将不透露这个网站的名字：当 Δx 为 0 时，将 $\Delta y / \Delta x$ 设置为 10^{10} ，这是一个“非常接近于无穷大”的值。然而就实践来说，我怀疑这种策略能否在大多数的实际情况中工作得很好。毕竟，如果程序的输入是来自于现实中的测量数据，那么将会出现远大于 10^{10} 的错误。同样，我没有很认真地考虑这个策略。虽然我不知道“The Book”中的共线函数是如何实现的，但我决不相信在这个函数中将有一个被定义为“非常接近无穷大”的值。

不可靠的斜率

除了通过在两点之间绘制一条直线然后判断第三点是否位于这条线的方法外，我还可以把由这三个点确定的三条线都绘制出来并且判断它们是否是同一条线。事实上，我只需绘制其中两条线就可以了，因为如果线 \overline{pq} 等于 \overline{qr} ，那么它就一定也等于 \overline{pr} 。而且，我只需要比较斜率即可，而无需比较y轴截距（你知道这是为什么？）。通过目测来判断两条线是共线还是形成很小的锐角或许并不是最为可靠的方法，但在计算机中，这个问题可以归结为对于两个数值的简单比较，即对 m 值的比较（参见图 33-3）。

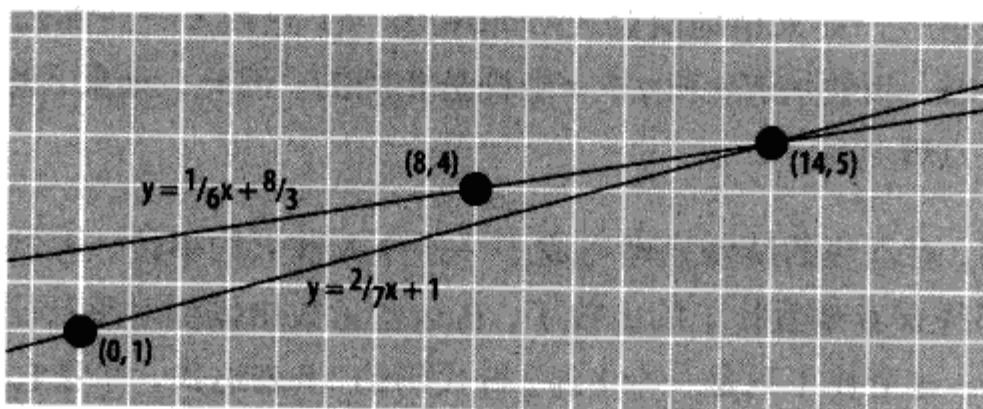


图 33-3：通过比较斜率来判断共线性

我编写了一个新的共线函数，如下所示：

```
(defun mm-collinear (px py qx qy rx ry)
  (equalp (slope px py qx qy)
          (slope qx qy rx ry)))
```

多么大的一次改进！这看上去要简单多了。在函数中不需要if表达式来区分垂线的情况，所有的点都以相同的方式来处理。

不过我必须承认，上面函数中的这种简洁性和表面上的一致性只是一种错觉，这是由于 Lisp 语言中隐藏的一些欺骗性而造成的。注意我在比较斜率时并没有使用等号 (=) 而是使用了相等性谓词函数 equalp。这个函数确实能够正确工作，因为无论当 slope 返回一个数值还是返回 nil 的时候，equalp 都能够做出正确的判断（也就是说，如果这两个斜率为相同的数值或者同时为 nil 时，那么就被认为是相等的）。在有着严格数据类型的语言中，这种定义是不会如此容易简化的，而是会像下面这样：

```
(defun typed-mm-collinear (px py qx qy rx ry)
  (let ((pq-slope (slope px py qx qy))
        (qr-slope (slope qx qy rx ry)))
    (or (and (numberp pq-slope)
              (numberp qr-slope)
              (= pq-slope qr-slope))
        (and (not pq-slope)
```

```
(not qr-slope))))
```

这段代码肯定是不漂亮的，虽然代码的形式更为明确，但在我看来其中的逻辑比前面的“naive”版本函数要好一些。原因在于如果斜率同时为数值，并且这些数值相等或者同时为 nil，那么 \overline{pq} 和 \overline{qr} 就是相同的线。而且，你总不能因为在其他语言中不能使用同样的技巧而指责 Lisp 吧？

我本来希望在此时停止开发工作，并把 mm-collinear 作为这个程序的最终版本，但由于在测试中出现了另外一个情况，使我不得不继续修改这个函数。mm-collinear 和 less-naive-collinear 都可以区分出共线点和几乎共线的点；像 $p=(0\ 0)$ 、 $q=(1\ 0)$ 、 $r=(1000000\ 1)$ 这样的情况并不会造成麻烦。但这两个函数在判断这组点的共线性时都失败了： $p=(0\ 0)$ 、 $q=(0\ 0)$ 、 $r=(1\ 1)$ 。

这里首要的问题就是，如何判断这种情况的共线性。我们的函数是用来判断三个点的共线性，而这里的 p 和 q 实际上是同一个点。我的观点是，像这样的点也应该被认为是共线的，因为我们可以在这三个点之间绘制一条简单的直线。然而，认为这三个点不共线的观点也同样说得过去，理由是在两个相同点之间可以绘制任意斜率的直线。但不幸的是，在我所编写的这两个函数中，没有一个函数能够与这两条规则中的任何一条保持一致。在这些函数中，对于上面的示例将返回 nil，而在计算 $p=(0\ 0)$ 、 $q=(0\ 0)$ 和 $r=(0\ 1)$ 时则将返回 t。显然，任何人都会认为这是一种病态的行为。

我可以通过给定约束条件来解决这个问题，明确指出传递给这个函数的三个参数必须是不同的点。但我随后又不得不编写相应代码来识别出违反规则的情况，然后引发异常，返回表示错误的值，批评违反规则的人等。这并不值得烦恼。

三角不等性

以下将通过另一种方式来重新考虑这个问题。我们观察到，如果 p 、 q 和 r 不共线，那么它们将构成一个三角形（如图 33-4 所示）。

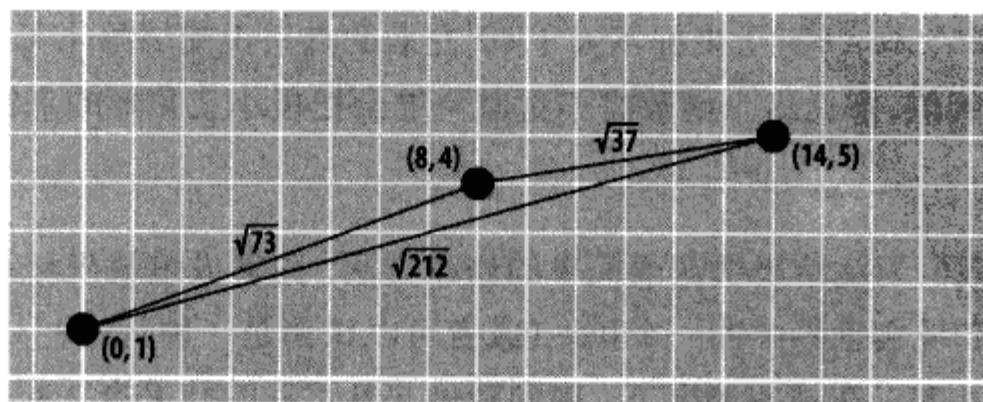


图 33-4：通过三角不等性来判断共线性

在上图的示例中，三角形的长边比两短边之和要小 0.067。

这个函数的代码并不像其他版本的函数那么紧凑，但在函数中执行的任务却是很简单的：

```
(defun triangle-collinear (px py qx qy rx ry)
  (let ((pq (distance px py qx qy))
        (pr (distance px py rx ry))
        (qr (distance qx qy rx ry)))
    (let ((sidelist (sort (list pq pr qr) #'>)))
      (= (first sidelist)
          (+ (second sidelist) (third sidelist))))))
```

这个函数的思想是计算三条边的长度，然后把它们放到一个链表中，并按照长度的大小降序排列，然后将第一条（最长的）边与其他两条边之和进行比较。当且仅当这两个长度是相等时，这三个点才是共线的。这种方法非常值得推荐。这种算法只依赖于这些点之间的几何关系，而与这些点在平面上的位置和方向是独立的。在算法中甚至没有用到斜率和截距。此外，这个函数在有两个或者三个点相同的情况下，也会给出一致并且合理的答案：所有这样的点都被认为是共线的。

不幸的是，为了获得这个简洁性，我们必须付出沉重的代价。到目前为止，所有的计算都是通过精确运算 (exact arithmetic) 来完成的。如果坐标值是整数或者有理数，那么在计算斜率和截距的时候就不会存在截断误差或者其他误差。例如，通过 $(1, 1)$ 和 $(4, 2)$ 这两个点的线的斜率 $m=1/3$ 并且截距 $b=2/3$ (小数形式大约为 0.33 和 0.67)。如果数值按照这种方式来表示，就可以确保比较运算能够得出在数学上正确的答案。但在测量距离时，精确性通常是无法获得的。在 `triangle-collinear` 中调用的 `distance` 函数定义如下：

```
(defun distance (px py qx qy)
  (sqrt (+ (square (- qx px))
            (square (- qy py))))))
```

当然，平方根是产生误差的根源。如果 `sqrt` 返回一个无理数的结果，那么就无法使用精确的、有限的和数值的数学表示。当使用双精度的 IEEE 浮点运算来计算距离时，`triangle-collinear` 对于坐标不大于 10^5 左右的点来说能够给出可靠的答案。但如果远远超过这个阈值，那么这个函数将不可避免地会把极小三角形的点错误地认为是共线的。

对于这个问题，我们无法给出快速而简单的修改方法。旋转或缩放坐标系的方法是行不通的。这只是我们现实世界中的一个 bug (或者可以认为是一种特性？)：有理数的点能够产生无理数的距离。在这些情况下，得到精确并且可靠的结果也并非是不可能的，但

这需要付出高强度的努力。当三个点是共线时，可以通过代数来证明无需求解平方根也能够判断出来。例如，假设有三个点 $(0, 0)$ 、 $(3, 3)$ 和 $(5, 5)$ ，那么距离等式就是 $\sqrt{50} = \sqrt{18} + \sqrt{8}$ ，这个等式可化简为 $5 \times \sqrt{2} = 3 \times \sqrt{2} + 2 \times \sqrt{2}$ 。当这些点不共线时，只要你计算了足够多的平方根小数位数，那么最终将会发现数值的不等性。但我并不愿意只是为了判断共线性而实现一个符号代数系统和一个可适应的多精度算法模块。肯定还有更简单的方法。我希望在“*The Book*”中的共线算法是更为简单的方式。

河道弯曲模型

为了讲述剩余部分的故事，我需要首先介绍一下这个故事的背景。几个月之前，我正在建立一个关于河道弯曲 (river meandering) 的简单模型——例如在密西西比河下游看到的那些巨大的马蹄铁状的河道弯曲。这个模型把光滑的河道分解为一系列短线段。我需要沿着河道测量曲率并且表示为这些线段之间扰角，尤其是我想要检测零曲率的区域——因此就需要使用判断共线性的谓词函数。

这个程序的另一个部分给我带来了更多的麻烦。随着河道弯曲的增加和迁移，一个弯流有时候遇到下一个弯流，此时河流将直接穿过弯流而留下一个“U字形”湖（当在密西西比河中遇到这种情况时，你并不希望这会给自己带来麻烦）。为了在模型中判断这种情况，我需要扫描线段的交点。虽然我能够编写出一个可用的函数，其中包含了带有许多分支的决策树，但这似乎带来了不必要的复杂性。正如在共线情况中需要对垂线和相同点的情况进行特殊处理，并且我还必须考虑平行线的情况。

对于交点的问题，我最终去图书馆里花了一些时间并且希望能找到一些解决方法。我学到了很多东西。我正是在这其中发现了为什么 10^{10} 可以被认为是非常接近于无穷大的数。Bernard Chazelle 和 Herbert Edelsbrunner 提出了一种更巧妙的方式来处理我所遇到的各种特殊情况。在 1992 年一篇关于线段交叉算法的文章（请参见“进一步阅读”部分）中，他们这样写到：

为了更容易暴露出问题，我们假设任意两个端点的 x 坐标或者 y 坐标都是不同的。只要将这个假设应用到同一条线段的两个端点，就可以排除垂线段或者水平线段的情况。我们的基本想法是能够最好地表达算法的核心思想而无需考虑在这种方式中每个步骤的特殊情况。放宽这个假设是非常容易（不需要新的思想）但也是乏味的事情。这是为了从理论上来分析算法。然而，将这个算法实现出来并使得程序能够适用于所有的情况是一件令人畏缩的任务。还有关于一些数值的问题，这些问题足以单独写成另一篇文章。不过，按照传统的习惯，我们将不考虑太多其他的限制因素。

我从阅读这篇文章中学到的最重要的原则或许就是，其他人也在这个领域中发现了繁杂的挑战。并不是只有我才会在这段代码中遇到问题。虽然这个发现恢复了我的信心，但这对于解决我的问题并没有任何帮助。

后来，我在自己的博客 <http://bit-player.org> 上写了一篇关于线段交叉算法的日志。这差不多算是一个希望得到帮助的请求，而帮助信息也很快就源源不断地来了，以至于我无法立刻全部看完。有个读者建议用极坐标来处理对未定义的斜率，而另一个读者提出用参数方式重新表示线性方程，这样 x 坐标和 y 坐标就是用新变量 t 的函数来给定。Barry Cipra 提出了一种略微不同的参数形式，然后给出了另一种算法，这种算法的思想是通过仿射变换（affine transformation）把其中一条线段变换到 $(-1, 0)$ 和 $(1, 0)$ 之间。David Eppstein 提出从欧几里得几何学中去掉这个问题并且在投影平面上来解决这个问题，因为投影平面中的“无穷点”概念能够有助于处理奇异点的问题。最后，Jonathan Richard Shewchuk 给了我一个链接，指向他的讲义、论文和正在编写的代码；我将在下面阐述 Shewchuk 的思想。

我被这些深思熟虑和富有创造性的建议深深地打动了——并且觉得稍微有些窘迫。在处理线段交叉的问题上可以有多种不同的方案。而且，我还发现了一个解决共线问题的答案。确实，我相信这个发给我的解决方案同样可以写入到《The Book》中。

“Duh! ” —— 我的意思是 “Aha! ”

在卡通中通常用一个在思考气球中点亮的灯泡来表示恍然大悟的意思。在我看来，这种突如其来的领悟更像是被一个 2×4 的木块打在后脑勺上。当你醒来的时候，你已经知道了一些东西，这些思想是如此地一目了然，以至于你都无法相信自己之前居然不知道这些思想。几天之后，你开始认为你可能确实知道这些思想；你肯定在以前就已经知道了，现在只是需要重新再回想起来。当你把这个发现告诉下一个人时，你会在开头说到，“正如所有人知道的……”

这是我在阅读 Jonathan Shewchuk 的“Lecture Notes on Geometric Robustness”这篇论文时的反应。他在这篇文章中给出了一个共线算法，在我理解了这个算法后，它看上去是如此地自然和合理，我确信它一定曾经潜伏在我脑海的某个地方。这个算法的关键思想是在于计算三角形的面积而不是像 triangle-collinear 函数那样计算三角形的周长。很明显，当且仅当三角形是一个退化三角形（即三个顶点共线）时，它的面积才为零。不过，在函数中计算面积而不是周长将有两个好处。首先，可以不需要平方根或者其他得出无理数的运算。其次，更少地依赖数值的精度。

考虑一组等腰三角形，顶点分别为 $(0, 0)$ 、 $(x, 1)$ 和 $(2x, 0)$ 。随着 x 的增加，底边长度和两腰长度之间的差异将逐步变小，因此很难把这个非常窄的三角形与一个完全扁平的，顶点为 $(0, 0)$ 、 $(x, 0)$ 和 $(2x, 0)$ 的三角形区分开来。而在计算面积时并不会遇到这样的问题。相反的是，三角形的面积将会随着 x 的增加而增加（参见图 33-5）。即便没有精确的计算，面积计算的结果也是非常稳定的。

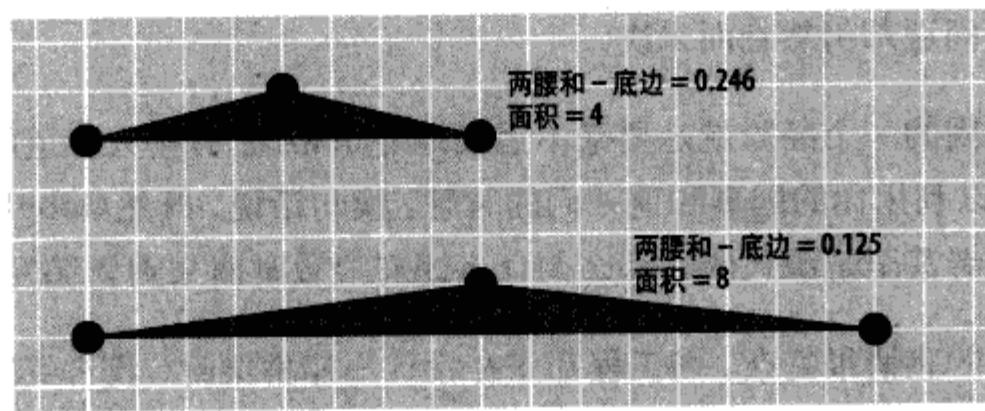


图 33-5：通过计算面积来判断共线性

如何计算面积？欧几里得的 $1/2bh$ 公式和三角方法都不是最佳的答案。更好的方法是把三角形的边视为矢量；从任何一个顶点发射出来的两个矢量定义了一个平行四边形，它的面积可以由两个矢量的叉乘来计算。三角形的面积刚好是这个平行四边形面积的一半。事实上，这种计算将得到“有符号的面积”：如果三角形的顶点是按照逆时针的方向来选取的，那么结果就是正的，如果按照顺时针的方式来选取，那么就是负的。而对于目前的共线问题，最重要的就是当且仅当这些顶点共线的时候，面积为零。

计算面积的矢量公式可以被简单地表达为一个 2×2 的行列式：

$$A = \frac{1}{2} \begin{vmatrix} x_1 - x_3 & y_1 - y_3 \\ x_2 - x_3 & y_2 - y_3 \end{vmatrix} = \frac{1}{2} [(x_1 - x_3)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_3)]$$

由于我只需考虑行列式为零的情况，因此可以忽略 $1/2$ 这个系数，然后按照这种简单的形式来编写判断共线性的代码：

```
(defun area-collinear (px py qx qy rx ry)
  (= (* (- px rx) (- qy ry))
      (* (- qx rx) (- py ry)))))
```

因此，这就最终版本的函数：一个对 x 坐标和 y 坐标进行计算的简单函数，在函数中只需要四次减法、两次乘法以及一个判断相等性的谓词函数，除此之外没有其他的东西——没有 `if`、斜率、截距、平方根以及除零错误等等。用浮点运算来描述这种行为是更为困难的，但这比基于周长比较的函数要好多了。Shewchuk 给出了一段高度优化的

C代码，这段代码在可能的情况下将使用浮点运算，而在必要的时候将转换为精确运算。

结论

在探索理想的共线性判断函数中，我的冒险经历和不幸并没有给这个故事带来有价值的思想。最终，我相信自己是在无意中发现了对于问题的正确解决方案，但关于如何寻找这样解决方案的问题却仍然悬而未决。

从我的经历中得到的一个经验就是要毫不迟疑地寻求帮助：有些人知道的东西比你所做的更多。你也可以利用你的同事和前辈们的智慧。换句话说，既然Google可能帮你找出想要的算法，或者甚至是源代码，那么为什么还有浪费时间来重新编写它呢？

我对于这个建议的感情很复杂。当工程师正在设计一座桥梁时，我希望她能够非常清楚在这个领域中的其他人过去是如何解决类似问题。然而，作为专家，她不应该只是发现和使用他人的思想；我希望我的桥梁设计者同样能够自行解决一些问题。

另一个问题就是，一个不良的程序将被保留多长时间。在本章中，我所讨论的程序是非常小的，因此当我遇到一点点不快时，只需非常少的工作就可以推倒这个程序然后重新开始编写。而对于大型的项目来说，推倒重来的方式并不是那么容易实现的。这么做是一种不必要的冒险：在解决已知问题的同时，可能会带来新的未知问题。

最后还有一个问题，“漂亮代码”的需求对编程或者软件开发应该有多大的影响。数学家 G. H. Hardy 断言：“在世界上，丑陋的数学没有永久的位置”。那么，在计算机科学中美学原则是否也占有非常重要的地位？这个问题的另一种表达形式是：对于每个严格形式化的计算难题，是否都存在可以写入到“The Book”中的程序？或许“The Book”中将会出现一些空白页。

进一步阅读

Avnaim, F., J.-D. Boissonnat, O. Devillers, F. P. Preparata, and M. Yvinec. "Evaluating signs of determinants using single-precision arithmetic." *Algorithmica*, Vol. 17, pp. 111-132, 1997.

Bentley, Jon L., and Thomas A. Ottmann. "Algorithms for reporting and counting geometric intersections." *IEEE Transactions on Computers*, Vol. C-28, pp. 643-647, 1979.

Braden, Bart. "The surveyor's area formula." *The College Mathematics Journal*, Vol. 17, No. 4, pp. 326-337, 1986.

Chazelle, Bernard, and Herbert Edelsbrunner. "An optimal algorithm for intersecting line segments in the plane." *Journal of the Association for Computing Machinery*, Vol. 39, pp. 1-54, 1992.

Forrest, A. R. "Computational geometry and software engineering: Towards a geometric computing environment." In *Techniques for Computer Graphics* (edited by D. F. Rogers and R. A. Earnshaw), pp. 23-37. New York: Springer-Verlag, 1987.

Forrest, A. R. "Computational geometry and uncertainty." In *Uncertainty in Geometric Computations* (edited by Joab Winkler and Mahesan Niranjan), pp. 69-77. Boston: Kluwer Academic Publishers, 2002.

Fortune, Steven, and Christopher J. Van Wyk. "Efficient exact arithmetic for computational geometry." In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pp. 163-172. New York: Association for Computing Machinery, 1993.

Guibas, Leonidas, and Jorge Stolfi. "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams." *ACM Transactions on Graphics*, Vol. 4, No. 2, pp. 74-123, 1985.

Hayes, Brian. "Only connect!" <http://bit-player.org/2006/only-connect>. [Weblog item, posted September 14, 2006.]

Hayes, Brian. "Computing science: Up a lazy river." *American Scientist*, Vol. 94, No. 6, pp. 490-494, 2006. (<http://www.americanscientist.org/AssetDetail/assetid/54078>)

Hoffmann, Christoph M., John E. Hopcroft and Michael S. Karasick. "Towards implementing robust geometric computations." *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pp. 106-117. New York: Association for Computing Machinery, 1988.

O'Rourke, Joseph. *Computational Geometry in C*. Cambridge: Cambridge University Press, 1994.

Preparata, Franco P., and Michael I. Shamos. *Computational Geometry: An Introduction*. New York: Springer-Verlag, 1985.

Qian, Jianbo, and Cao An Wang. "How much precision is needed to compare two sums of square roots of integers?" *Information Processing Letters*, Vol. 100, pp. 194-198, 2006.

Shewchuk, Jonathan Richard. "Adaptive precision floating-point arithmetic and fast robust geometric predicates." *Discrete and Computational Geometry*, Vol. 18, pp. 305-363, 1997. Preprint available at <http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-arithmetic.ps>.

Shewchuk, Jonathan Richard. Lecture notes on geometric robustness. [Version of October 26, 2006.] (<http://www.cs.berkeley.edu/~jrs/meshpapers/robnotes.ps.gz>. See also source code at <http://www.cs.cmu.edu/afs/cs/project/quake/public/code/predicates.c>.)

Steffen, Christian. Geometric predicates for floating-point arithmetic. [Version of November 1, 2005.] (<http://www.cs.tufts.edu/~steffen/papers/fpgeom.pdf>).

Taylor, Michael S. A note on floating-point arithmetic and geometric predicates. [Version of January 20, 2005.] (<http://www.cs.cmu.edu/~mstaylor/pubs/fpgeom.pdf>).

Wang, Cao An. Robust geometric predicates for floating-point arithmetic. [Version of June 10, 2005.] (<http://www.cs.cmu.edu/~caowang/pubs/fpgeom.pdf>).

Wang, Cao An. How many digits are needed to compare two sums of square roots? [Version of July 18, 2006.] (<http://www.cs.cmu.edu/~caowang/pubs/sumsqrt.pdf>).

Wang, Cao An. Robust geometric predicates for floating-point arithmetic. [Version of August 1, 2006.] (<http://www.cs.cmu.edu/~caowang/pubs/fpgeom.pdf>).

Wang, Cao An. A note on floating-point arithmetic and geometric predicates. [Version of August 1, 2006.] (<http://www.cs.cmu.edu/~caowang/pubs/fpgeom.pdf>).

Wang, Cao An. How many digits are needed to compare two sums of square roots? [Version of September 1, 2006.] (<http://www.cs.cmu.edu/~caowang/pubs/sumsqrt.pdf>).

后记

Andy Oram

《代码之美》介绍了人类在一个奋斗领域：计算机系统的开发领域中的创造性和灵活性。在每章中的漂亮代码都来自独特解决方案的发现，而这种发现来源于作者超越既定边界的远见卓识，并且识别出被多数人忽视的需求以及找出令人叹为观止的解决方案。

大多数作者都面临着种种限制——包括物理环境，可用资源，或者特殊的需求定义——这些限制通常会使我们很难想象出解决方案。而其他一些作者则是在已经存在解决方案的领域中重新研究，并且提出新的观点以及更好地实现某个功能。

本书的所有作者都从他们的项目中获得了一些经验。不过在阅读完本书后，我们同样可以总结出一些更广泛的经验。

首先，在可靠和真实的规则能够真正应用之前，需要进行多次尝试。因为，人们在维护稳定性、可靠性以及其他软件工程要求的标准时经常会遇到重重困难。在这种情况下，我们通常没有必要抛弃支持这种承诺的原则。有时候，从另一个角度来思考问题或许能够揭示一种新的方向，从而使我们在满足需求的同时无需牺牲那些好的技术。

另一方面，在有些章节中强调了这条古老的原则：在打破原则之前，人们必须首先了解

这个规则。有些作者在获得一种不同的解决方案之前积累了数十年的经验——而正是这些经验给了他们自信，从而以创造性的方式打破规则。

此外，书中的一些经验还提倡跨学科研究。许多作者都是在新的领域中进行研究并在黑暗中不断探索。在这种情况下，全新的创造力和个人智慧将起到重要的作用。

最后，我们从书中学到的漂亮的解决方案并不会持续很长时间。在新的环境中总会要求新的解决方式。因此，如果阅读了本书并且认为“无法在自己的任何一个项目上使用这些作者的解决方案”，那么也不要担心——这些作者在做下一个项目的时候，也会使用不同的解决方案。

我在这本书上全身心地工作了两个月，以帮助作者完善他们的主题和更好地表达他们的观点。阅读这些天才发明家的文章的确令人鼓舞甚至是令人情绪高涨的。它给了我尝试新鲜事物的冲动，我希望读者在阅读本书时也能有同样的感受。

我非常感谢所有为本书贡献了文章的作者们。他们都是才华横溢、经验丰富的发明家，他们的文章充满了创意、激情和对未来的憧憬。他们的故事激励着我，让我相信只要坚持不懈，就一定能够实现自己的梦想。我也希望这本书能够激发更多人的创新精神，鼓励大家勇敢地追求自己的目标。再次感谢所有作者的辛勤付出，期待你们未来更多精彩的作品！

作者简介

John Bentley 是美国 Avaya 实验室的一位计算机科学家。他的研究领域包括编程技术、算法设计以及软件工具与界面设计。他已编写了数本关于编程的书籍，还撰写了大量的文章，主题涉及从算法理论到软件工程的各个方向。他于1974年在斯坦福大学获得学士学位，并于1974年获得硕士学位以及于1976年在北卡罗来纳大学获得博士学位，随后在卡耐基·梅隆大学任教6年，教授计算机科学。1982年他加入贝尔实验室，并于2001年离开贝尔实验室并加入 Avaya 实验室。他曾是西点军校和普林斯顿大学的访问教授、曾经参与开发过软件工具、电话交换机、电话以及网络服务。

Tim Bray 于1987-1989年间在加拿大的安大略省滑铁卢大学负责牛津英语词典项目，1989年与他人联合创建了 Open Text 公司，在1995年启动了最早的公共网页搜索引擎之一，在1996至1999年间与他人共同发明了 XML 1.0 并合作编写了《Namespaces in XML》规范，在1999年他创建了 Antarctica Systems 公司，并于2002-2004年被 Tim Berners-Lee 任命在 W3C 技术架构组中工作。目前，他在 Sun Microsystems 公司 Web Technologies 部门任主管，他有一个很受欢迎的博客，并且参与主持 IETF AtomPub 工作组。

Bryan Cantrill 是 Sun Microsystems 公司的一位杰出的工程师，在他的职业生涯中主要从事 Solaris 内核的开发。最近他与同事 Mike Shapiro 和 Adam Leventhal 一起设计并实现了 DTrace，这是一个用于产品系统动态控制的工具，获得了《华尔街日报》2006 年度的最高创新奖。

Douglas Crockford 毕业于公立学校。他是一位登记选民，拥有自己的汽车。他曾开发过办公自动化系统。他曾在 Atari 公司从事过游戏和音乐研究。他曾是 Lucasfilm 有限公司技术部门的主管，以及 Paramount 公司 New Media 部门的主管。他创建了 Electric Communities 公司并且担任 CEO。他还是 State 软件公司的创建者和 CTO，正是在这个公司中他发明了 JSON 数据格式。他现在是 Yahoo! 公司的一位架构师。

Rogerio Atem de Carvalho 是巴西校园技术教育联合中心 (Federal Center for Technological Education of Campos, CEFET Campos) 的一位教师兼研究人员。他在奥地利的维也纳获得了 2006 年度 IFIP 杰出学术领导奖 (Distinguished Academic Leadership Award)，以表彰他在免费 / 开源企业资源计划 (ERP) 上所做的研究工作。他的研究领域还包括决策支持系统和软件工程。

Jeff Dean 于 1999 年加入 Google，目前是 Google 系统架构小组的成员。他在 Google 主要负责开发 Google 的网页抓取、索引、查询服务以及广告系统等，他对搜索质量实现了多次改进，并实现了 Google 分布式计算架构的多个部分。在加入 Google 之前，他服务于 DEC/Compaq 的 Western 实验室，主要从事软件分析工具、微处理器架构以及信息检索等方面的研究。他于 1996 年在华盛顿大学获得了博士学位，与 Craig Chambers 一起从事面向对象语言的编译器优化技术方面的研究。在毕业之前，他还在世界卫生组织的艾滋病全球规划署工作过。

Jack Dongarra 于 1972 年在芝加哥大学获得数学学士学位，并于 1973 年在伊利诺理工大学获得计算机科学硕士学位，又于 1980 年在新墨西哥大学获得应用数学博士学位。他在美国阿贡国家实验室 (Argonne National Laboratory) 一直工作到 1989 年，并成为了一名著名科学家。他现在被任命为田纳西大学计算机科学系的计算机科学杰出教授。他是美国橡树岭国家实验室 (Oak Ridge National Laboratory, ORNL) 计算机科学与数学部的杰出的研究人员，曼彻斯特大学计算机科学与数学学院的 Turing Fellow，美国莱斯大学计算机科学系的副教授。他的研究领域包括线性代数中的数值算法，并行计算，高级计算机架构的应用，程序设计方法学以及用于并行计算机的工具。他的研究工作包括开发、测试高质量的数学软件以及整理相关文档。他在以下开源软件包和系统的设计及实现上做出了贡献：ISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS 和 PAPI。他公开发表了大约 200 篇文

章、论文、报告以及技术备忘录，还参与编写了数本著作。他于2004年获得了IEEE Sid Fernbach奖，以表彰他在高性能计算机的应用中使用了创新的方法。他不仅是AAAS, ACM和IEEE的成员，还是美国工程院的院士。

R. Kent Dybvig是印第安纳大学计算机科学系的一位教授。在印第安纳大学任教两年之后，他于1987年在北卡罗来纳大学获得了博士学位。他在设计和实现编程语言的研究上做出了重要的贡献，包括控制运算符、句法抽象、程序分析、编译器优化、寄存器分配、多线程以及自动存储管理等。在1984年，他创建了Chez Scheme软件并一直是主要的开发人员。Chez Scheme的特点在于快速的编译时间、可靠性以及能够高效地运行内存需求巨大的复杂程序，它已经被用于构建企业集成、网页服务、虚拟现实、机器人药品抽检、电路设计以及其他商业系统。它还可以用于各种层次的计算机教育以及许多其他领域中的研究。Dybvig是《The Scheme Programming Language, Third Edition》(MIT Press出版社)一书的作者，以及即将发布的“Revised⁶ Report on Scheme”文档的编辑。

Michael Feathers是Object Mentor公司的顾问。在过去七年间，他一直活跃于Agile社群，他的工作主要是与世界各地不同的团队合作，培训以及指导。在加入Object Mentor公司之前，Michael设计过一种编程语言，并为这种语言写了一个编译器。他还设计了一个庞大的多平台类库以及用于控制的框架。Michael开发了CppUnit，也就是最初把JUnit移植到C++；以及FitCpp，也就是把FIT移植到C++。在2005年，Michael编写了《Working Effectively with Legacy Code》(Prentice Hall出版社)一书。在与各个团队合作的间隙，他的大多数时间都花在研究大型代码库中的设计修改方式方面。

Karl Fogel和Jim Blandy 1995年一起创建了Cyclic软件公司，这是第一个提供商业CVS支持的公司。1997年，Karl增加了对CVS匿名只读存储仓库访问的支持，这样就可以更方便地访问开源项目中的开发代码。1999年，他工作于CollabNet公司，主要从事管理Subversion的创建和开发工作，这是CollabNet公司和一群开源志愿者们从头开始编写的开源版本控制系统。2005年，他编写了《Producing Open Source Software: How to Run a Successful Free Software Project》(O'Reilly出版社；在<http://producingoss.com>上有联机版本)一书。2006年，他在Google担任了短期的开源技术专家之后离开Google并成为了Question-Copyright.org网站的全职编辑。他目前仍然参与了多个开源项目，包括Subversion和GNU Emacs。

Sanjay Ghemawat是一位Google Fellow，工作于Google的系统架构小组。他设计并实现了分布式的存储系统，文本索引系统，性能分析工具，一种数据表示语言，一个RPC系统，一个malloc函数实现以及许多其他的库。在加入Google之前，他是DEC系统研

究中心的一位研究人员，主要从事系统性能分析和优化Java编译器的工作，他还实现了一个Java虚拟机。他于1995年在麻省理工大学获得博士学位，研究领域为面向对象数据库的实现。

Ashish Gulhati 是互联网隐私服务 Neomailbox 的首席开发员，以及 Cryptonite 的开发员，这是一个支持 OpenPGP 协议的安全网页邮件系统。他有着 15 年的商业软件开发经验，是印度最早的数字版权活动家之一和 F/OSS 程序员，他编写了大量的开源 Perl 模块，这些模块可以从 CPAN 上下载。在 1993~1994 年间，他在《PC Quest》和《DataQuest》等杂志上发表了大量文章，这是在印度主流计算机刊物中最早向读者介绍自由软件，GNU/ Linux，Web 和 Internet 的文章，在这些文章发表多年以后，印度才拥有了商业的互联网访问，这些文章还构成了 PC Quest Linux Initiative 活动的重要组成部分，这个活动促使自 1995 年以来，在印度分发了一百万份 Linux 光盘。在获得了一组可穿戴的计算机后，他很快地成为了一个电子人。

Elliotte Rusty Harold 是新奥尔良人，他会定期返回新奥尔良去吃一大碗海鲜干波汤 (Gumbo)。不过，他目前住在布鲁克林附近的 Prospect Heights，和他生活在一起还有他的妻子 Beth，狗 Shayna，和两只猫 Charm (以夸克命名) 和 Marjorie (以他的岳母命名)。他是纽约科技大学的一位副教授，主要讲授 Java、XML 以及面向对象编程。他的 Cafe au Lait 网站 (<http://www.cafeaulait.org>) 是互联网上最流行的独立 Java 网站之一；他的另一个网站 Cafe con Leche (<http://www.cafeconleche.org>) 则成为了最流行 XML 站点之一。他编写的书籍包括《Java I/O》，《Java Network Programming》和《XML in a Nutshell》(这三本书都由 O'Reilly 出版社出版)，以及 XML Bible (Wiley 出版社)。他目前的研究领域包括用 Java 来处理 XML 的 XOM 库、Jaxen XPath 引擎以及 Amateur 媒体播放器。

Brian Hayes 为《American Scientist》杂志编写计算机专栏，他还拥有一个博客 <http://bit-player.org>。过去，他还为《Scientific American》、《Computer Language》、以及《The Sciences》等杂志编写过类似的专栏。他编写的《Infrastructure: A Field Guide to the Industrial Landscape》(Norton 出版社) 一书于 2005 年发行。

Simon Peyton Jones，硕士，于 1980 年毕业于剑桥大学三一学院。在工作两年后，他在伦敦大学学院担任了 7 年的讲师，然后在格拉斯哥大学担任了 9 年的教授，后来于 1998 年加入微软研究中心。他的研究领域包括函数式编程语言及其实现和应用。他领导了一系列的研究项目，主要研究用于单处理器机器和并行机的高质量函数式语言系统的设计和实现。他是函数式语言 Haskell 的主要设计者，此外他还是被广泛应用的 Glasgow Haskell 编译器 (GHC) 首席设计师。他还编写了两本关于函数式语言实现的教科书。

Jim Kent 是加利福尼亚大学圣克鲁兹分校基因信息小组 (Genome Bioinformatics Group) 的一位研究学家。Jim 从 1983 年起就开始编程。在职业生涯的前半段，他主要从事绘画和动画软件的开发，他开发了 Aegis Animator、Cyber Paint 以及 Autodesk Animator 等获奖软件。1996 年，由于厌倦了基于 Windows API 的开发工作，他决定在生物学上追求他的兴趣，并于 2002 年获得了博士学位。在研究生期间，他编写 GigAssembler——这个程序计算出了第一批人类基因组——比 Celera 公司发布的第一批基因组提前了一天，从而使得这批基因组成为免费的专利并且避免了其他的法律问题。Jim 发表了 40 余篇科学论文。他目前的研究工作主要是编写程序，数据库和网站以帮助科学家分析和了解基因组。

Brian Kernighan 于 1964 年在多伦多大学获得学士学位，并于 1969 年在普林斯顿大学获得电子工程博士学位。他在贝尔实验室的计算科学研究中心一直工作到 2000 年，目前就职于普林斯顿大学的计算机科学系。他编写了 8 本著作以及大量的技术论文，并拥有 4 项专利。他的研究领域包括编程语言、工具、为非专业用户设计易用的计算机操作界面等。他还致力于非技术读者的技术教育工作。

Adam Kolawa 是 Parasoft 公司的创建者之一和 CEO，这家公司是自动错误预防 (Automated Error Prevention , AEP) 解决方案的领先提供商。Kolawa 有着多年在各种软件开发流程中的经验，这使得他对高科技企业有着独特的视野，以及成功辨识技术潮流的非凡能力。因此，他策划了几个成功商业软件产品的开发过程来满足在提高软件质量中不断增长的工业需求——经常在这种潮流被广泛接受之前。Kolawa 参与编写了《Bulletproofing Web Applications》(Hungry Minds 出版社)一书，他还撰写了 100 余篇评论和技术文章，发表在《The Wall Street Journal》、《CIO》、《Computerworld》、《Dr. Dobb's Journal》以及《IEEE Computer》等期刊上。此外，他还撰写了大量关于物理学和并行处理方面的科学论文。他现在的签约媒体包括 CNN、CNBC、BBC 和 NPR。Kolawa 拥有加利福尼亚理工大学理论物理博士学位，并拥有 10 项专利发明。2001 年，Kolawa 获得了软件类别的 Los Angeles Ernst & Young's Entrepreneur of the Year 奖项。

* **Greg Kroah-Hartman** 是目前 Linux 内核的维护人员，负责多个驱动程序子系统以及驱动程序内核、sysfs、kobject、kref 和 debugfs 等代码。他还为启动 linux-hotplug 和 udev 等项目提供了帮助，是内核稳定维护团队中的重要人员。他编写了《Linux Kernel in a Nutshell》(O'Reilly 出版社)，并参与编写了《Linux Device Drivers, Third Edition》(O'Reilly 出版社)。

Andrew Kuchling 有着 11 年的软件工程师经验，他是 Python 开发群体中的长期成员。他的一些与 Python 相关的工作包括编写和维护数个标准的库模块，编写一系列的

“What's new in Python 2.x”文章以及其他一些文档，策划了2006年和2007年的PyCon会议，并是Python软件基金会的主管。Andrew于1995年毕业于麦吉尔大学并获得计算机科学学士学位。他的个人网页是 <http://www.amk.ca>。

Piotr Luszczek 毕业于波兰克拉科夫矿业与冶金大学，并获得硕士学位，他的研究领域是并行的核外（out-of-core）库。他将稠密矩阵计算核应用于稀疏矩阵直接求解算法和迭代数值线性几何算法中的创新研究使他获得了博士学位。他把这种思想用来开发使用核外技术容错库。目前，他是田纳西大学诺克斯维尔分校的一位研究教授。他的研究工作包括大型超级计算机安装的标准化评价。他开发了一个自适应的软件库，能够自动选择最优的算法来有效地利用现有硬件以及有选择地处理输入数据。他还感兴趣于高性能编程语言的设计和实现。

Ronald Mak 是高级计算机科学研究所（Research Institute for Advanced Computer Science）的一位资深科学家，在NASA Ames 研究中心工作时，他是协同信息系统（Collaborative Information Portal, CIP）的架构师和首席开发人员。在漫步者登陆火星之后，他分别在JPL 和Ames 对探测任务提供支持。然后，他获得了加利福尼亚大学圣克鲁兹分校的学术任命，并且他再次与NASA 签约，这次的工作是设计帮助宇航员返回月球的企业软件。Ron是 Willard & Lowe Systems (<http://www.willardlowe.com>) 公司的创建人之一和CTO，这是一个针对企业信息管理系统的咨询公司。他编写了数本关于计算机软件的书籍，他在斯坦福大学分别获得了数学科学学位和计算机科学学位。

Yukihiro “Matz” Matsumoto 是一位程序员，他是一位日本籍的开源倡导者，他发明了最近非常流行的Ruby 语言。他从1993年开始研发Ruby，这和Java 语言一样久远。现在他工作于日本Network Applied Communication Laboratory (NaCl, 网址为 netlab.jp) 公司，该公司从1997年起开始赞助Ruby 的开发。因为他的真实姓名太长而难以记住，并且对于非日本的演讲者来说难以发音，因此在网上他使用了昵称Matz。

Arun Mehta 是一位电子工程师和计算机科学家，他曾在印度、美国和德国进行过研究和教学工作。他是印度早期计算机活动家，他努力实现了一些方便消费者（consumer-friendly）的政策，以帮助把现代通信延伸到偏远地区和贫困地区。他目前的研究领域包括农村无限通信以及帮助残疾用户的技术。他是印度哈里亚纳邦Radaur地区JMIT大学计算机工程系的教授和主任。他的网址包括 <http://india-gii.org>, <http://radiophony.com> 和 <http://holisticit.com>。

Rafael Manhaes Monnerat 是CEFET CAMPOS 的一位IT 分析家，以及Nexedi SARL的海外顾问。他的研究领域包括免费 / 开源系统、ERP 以及最新的编程语言。

Travis E. Oliphant 于 1995 年在美国杨百翰大学获得电子与计算机工程学士学位和数学学士学位，并于 1996 年在本校获得电子与计算机工程硕士学位。他于 2001 年在明尼苏达罗切斯特的梅奥研究生院获得了生物医学工程博士学位。他是 Python 语言中科学计算库 SciPy 和 NumPy 的主要编写者。他的研究领域包括显微阻抗成像，异构领域中的 MRI 重构以及生物医学逆问题。他目前是杨百翰大学电子与计算机工程的副教授。

Andy Oram 是 O'Reilly Media 的编辑。他从 1992 年开始就在这家公司工作，Andy 目前主要关注自由软件和开源技术。他在 O'Reilly 的工作成果包括第一批 Linux 系列丛书以及 2001 年的 P2P 系列丛书。他的编程技术和系统管理技术大多都是自学的。Andy 还是 Computer Professionals for Social Responsibility 协会的成员并且经常在 O'Reilly Network (<http://oreillynet.com>) 和其他一些刊物上撰写文章，这些文章的主题包括互联网上的政策问题，以及影响技术创新的潮流及其对社会的影响。他的网址为 <http://www.praxagora.com/andyo>。

William R. Otte 是田纳西范德堡大学电子工程与计算机系 (EECS) 的一位博士研究生。他的研究领域是分布式实时嵌入 (DRE) 系统的中间件，目前从事 CORBA 组件的部署和配置引擎 (DAnCE) 开发工作。这个工作主要研究运行时规划技术，基于组件的应用程序的适应性，以及对应用程序服务质量与容错需求的规范与实施。在攻读研究生之前，William 于 2005 年在范德堡大学计算机系毕业并获得学士学位，之后在软件集成系统学院 (ISIS) 工作了一年。

Andrew Patzer 是威斯康星大学医学院生物信息系的主管。过去 15 年 Andrew 是一位软件开发人员并且编写了许多文章和书籍，包括《Professional Java Server Programming》(Peer Information 公司) 和《JSP Examples and Best Practices》(Apress 出版社)。Andrew 目前的研究领域为生物信息领域，利用像 Groovy 这样的动态语言来发掘大量有效的生物数据并帮助科学研究人员进行分析。

Charles Petzold 是一位自由作家，主要研究领域为 Windows 应用程序编程。他是《Programming Windows》(Microsoft Press 出版社) 的作者，1988 年至 1999 年之间共出版了五版，教育了整整一代程序员的 Windows API 编程技术。他最新的书籍包括《Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation》(Microsoft Press 出版社)，以及《Code: The Hidden Language of Computer Hardware and Software》(Microsoft Press 出版社)，在这本书中他对数字技术进行了独特的研究。他的网址是 <http://www.charlespetzold.com>。

T. V. Raman 的研究领域包括网页技术和听觉用户界面。在 20 世纪 90 年代初，在他的博士论文中介绍了音频格式的概念，叫作 AsTeR: Audio System For Technical Readings

(技术读物语音系统)，这是一个为技术文档生成高质量听觉表示的系统。Emacspeak则将这些思想应用到更广泛的计算机用户界面领域。Raman 现在是 Google 的一位研究人员，主要研究 Web 应用程序。

Alberto Savoia 是 Agitar 软件公司的创建人之一和 CTO。在创建 Agitar 之前，他是 Google 的高级工程主管；在这之前，他还是 Sun Microsystems 实验室软件研究中心的主管。Alberto 的主要研究领域是软件开发技术——尤其是那些帮助程序员在设计和开发阶段进行测试和代码验证的工具和技术。

Douglas C. Schmidt 是田纳西范德堡大学电子工程与计算机 (EECS) 系的一位教授，计算机科学与工程系的副主任，以及软件集成系统学院 (ISIS) 的高级研究人员。他是分布式计算模式和中间件框架方面的专家，并且已经发表了超过 350 篇的技术论文和 9 本书籍，内容涉及的主题很广，包括高性能通信软件系统，高速网络协议并行处理，实时分布式对象计算，并发与分布式系统的面向对象模式，以及模型驱动的开发工具。在他的学术研究之外，Dr. Schmidt 还是 PrismTechnologies 公司的 CTO，并且在领导开发应用广泛开源的中间件平台上有着 15 年的经验，在这些平台上包含了丰富的组件以及实现高性能分布式系统中核心模式的领域特定语言。Dr. Schmidt 于 1994 年于加利福尼亚大学欧文分校获得计算机科学博士学位。

Christopher Seiwald 编写了 Perforce (一种软件配置管理系统)、Jam (一种构建工具) 和“漂亮代码的七个要素”(本书的第 32 章，变动的代码，正是从这篇文章中提取出了有价值的思想)。在创建 Perforce 之前，他在 Ingres 公司管理网络开发小组，他花了数年时间来使得异步网络代码看上去很漂亮。现在他是 Perforce 软件公司的 CEO，并且仍然从事编码工作。

Diomidis Spinellis 是希腊雅典经济与商业大学管理科学与技术系的副教授。他的研究领域包括软件工程工具，编程语言和计算机安全。他在伦敦帝国理工大学获得了软件工程硕士学位和计算机科学博士学位。他发表了超过 100 篇的技术论文，所涉及的领域包括软件工程，信息安全以及普适计算。他还编写了两本开源方面的书籍：《Code Reading》(获得 2004 年度 Software Development Productivity 奖) 和《Code Quality》(这两本书都由 Addison-Wesley 出版社出版)。他是 IEEE Software 编辑委员会的成员，主编“Tools of the Trade”专栏。Diomidis 是一位 FreeBSD 提交者 (Committer)，并且编写了许多开源软件包、软件库以及工具。

Lincoln Stein 是一位硕士/博士，他的研究领域为生物信息数据的集成与虚拟化。在从哈佛大学医学院毕业后，他在麻省理工大学 Whitehead 基因研究所工作，开发用于老鼠和人类的基因图谱数据库。他在冷泉港实验室开发了各种基因数据库，包括

WormBase, 线虫基因数据库; Gramene, 用于水稻和其他单子叶植物的比较基因映射数据库; 国际 Hap-Map 项目数据库; 以及人类基因基础数据库 Reactome。Lincoln 还编写了《How to Set Up and Maintain a Web Site》(Addison-Wesley 出版社)、《Network Programming in Perl》(Addison-Wesley 出版社)、《Official Guide to Programming with CGI.pm》(Wiley 出版社) 以及《Writing Apache Modules with Perl and C》(O'Reilly 出版社) 等书籍。

Nevin Thompson 把 Yukihiro Matsumoto 编写的第 29 章内容, 把代码当作文章, 从日文翻译到英文。他的客户包括日本最大的电视网络, 以及 Technorati Japan 公司和 Creative Commons 组织。

Henry S. Warren, Jr. 在 IBM 工作了 45 年, 他历经了从 IBM 704 到 PowerPC 的发展过程。他参与过多个军方指挥与控制系统的开发工作, 在纽约大学 Jack Schwartz 教授指导下从事 SETL 项目。从 1973 年起, 他在 IBM 研究部门工作, 主要方向为编译器和计算机架构。Hank 目前正在参与 Blue Gene Petaflop 超级计算机项目。他在纽约大学克朗数学研究所获得了计算机博士学位。他是《Hacker's Delight》(Addison-Wesley 出版社) 一书的作者。

Laura Wingerd 多年 Sybase 和 Ingres 的数据库产品开发工作形成了她早期对软件配置管理的观点。她在 Perforce 软件公司创建之初就加盟了这家公司, 并且从她给 Perforce 客户的建议中获得了大量的 SCM 经验。她编写了《Practical Perforce》(O'Reilly 出版社) 一书以及许多与 SCM 相关的白皮书。她在 Google 的技术演讲 The Flow of Change 中首次露面。Laura 现在是 Perforce 软件公司产品技术部的副主管, 主要负责推动合理的 SCM 流程以及研究新的并且更好的 Perforce 使用方式。

Greg Wilson 在爱丁堡大学获得了计算机科学博士学位, 他的研究领域包括高性能科学计算, 数据虚拟化以及计算机安全。他现在是多伦多大学计算机科学系的一位副教授, 并且是《Dr. Dobb's Journal》杂志的特约编辑。

Andreas Zeller 于 1991 年毕业于德国达姆斯达特理工大学, 并于 1997 年在不伦瑞克理工大学获得计算机科学博士学位。2001 年以来, 他一直在德国萨尔兰登大学的计算机科学系担任教授。Zeller 主要研究大型程序以及它们的发展历史, 他开发了大量的方法来分析在开源软件以及 IBM、Microsoft、SAP 以及其他公司的商业软件中失败的原因。他编写的《Why Programs Fail: A Guide to Systematic Debugging》(Morgan Kaufmann 出版社) 获得了《Software Development Magazine》杂志 2006 年度的 Productivity 大奖。