/Theory/In/Practice

架构之美

Beautiful Architecture

顶级业界专家揭密软件设计之美

Diomidis Spinellis & Georgios Gousios 编

王海鹏 蔡黄辉 徐锋 等译

O'REILLY®

机械工业出版社
China Machine Press

HZ BOOKS

Software Engineering

架构之美

“本书的作者们在介绍软件架构的基本实践和最佳实践方面得心应手，他们也同样漂亮地介绍了各式各样的现代系统。我特别喜欢他们谈及的架构的广泛性，从Emacs到Facebook，从非常正式的系统到非常有灵气的系统。简而言之，本书对软件架构的艺术、科学和实践作出了非常及时和有益的贡献。”

——Grady Booch，IBM院士

健壮、优雅、灵活和易维护的软件架构是怎样炼成的？本书通过一系列优秀的文章回答了这个问题，这些文章来自于十几位当今一流的软件设计师和架构师。在每篇文章中，作者都向我们展示了一个著名的软件架构，并分析了什么让其具有创新性，让其极其符合设计目标。

本书内容包括：

- Facebook的架构如何建立在以数据为中心的应用生态系统之上。
- Xen的创新架构对操作系统未来的影响。
- KDE项目的社区过程如何让软件的架构从粗略的草图演进为漂亮的系统。
- 不断滋长的特征如何让GNU Emacs获得从未预料到的功能。
- Jikes RVM自优化、自足执行的运行时环境背后的魔法。

本书作者包括：

John Klein and David Weiss
Pete Goodliffe
Jim Waldo
Michael Nygard
Brian Sletten
Dave Fetterman
Derek Murray and Keir Fraser
Greg Lehey
Rhys Newman and Christopher Dennis
Ian Rogers and Dave Grove
Jim Blandy
Till Adam and Mirko Boehm
Bertrand Meyer
Panagiotis Louridas

所有作者版税捐献给无国界医生组织。

客服热线：(010) 88378991，88361066
购书热线：(010) 68326294，88379649，68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：*http://www.hzbook.com*
网上购书：*www.china-pub.com*

O'REILLY®
www.oreilly.com

ISBN 978-7-111-28356-0

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)

定价：69.00元

一L

架构之美

Diomidis Spinellis & Georgios Gousios 编

王海鹏 蔡黄辉 徐锋 等译

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

架构之美 /（美）斯宾耐立思（Spinellis, D.），（美）郭西奥斯（Gousios, G.）编；王海鹏等译. —北京：机械工业出版社，2010.1

书名原文：Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design

ISBN 978-7-111-28356-0

Ⅰ.架… Ⅱ.①斯… ②郭… ③王… Ⅲ.软件设计 Ⅳ.TP311.5

中国版本图书馆CIP数据核字（2009）第172092号

北京市版权局著作权合同登记

图字：01-2009-1579

书　　名 / 架构之美
书　　号 / ISBN 978-7-111-28356-0
责任编辑 / 周茂辉
封面设计 / Karen Montgomery，张健
出版发行 / 机械工业出版社
地　　址 / 北京市西城区百万庄大街22号（邮编　100037）
印　　刷 / 北京京师印务有限公司
开　　本 / 178毫米×233毫米　16开本　24.5印张
版　　次 / 2010年1月第1版第1次印刷
定　　价 / 69.00元（册）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

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，同时也是联机出版的先锋。

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

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

推荐序一

如何看到一滴水的美丽

支付宝（中国）公司业务架构师
《大道至简》作者
周爱民（aimingoo）

【一】

架构是一个过程，而非一个结果。

【二】

在大多数人的谈论中，架构是一个目标产物，而作为架构师的责任就是去生产它。所以无论如何，架构是可以“做”出来的，而且也应该有一些“做”的方法、技术、技巧。

有人问过我：架构的最主要产出是什么？我的答案是：图。这里面有两层含义：一层含义是如同建筑师描绘的蓝图一样，用于引导实施者；另一层含义是架构师头脑中清晰的目标系统。如果架构师头脑中没有系统清晰的图像，他是没有办法把它画出来的。

【三】

画家画的无非是物我。画物的画家，最终画的还是我见。所以，画家的笔最终描绘的是他自己心里的映像。

【四】

艺术是不可能被“生产”出来的，生产出来的，叫“艺术品”。

【五】

架构这个过程，是架构师洞见系统内在结构、规律、原则和逻辑的过程。真正的架构师是可以将自己放在系统中去的（例如作为系统中的任何一个角色），只有清晰地理解系统，才能简洁地描述它。而当架构师拿出了他所描述的“作品”的时候，架构这一过程就已经结束了。

【六】

一滴水滴落的过程中，有多少个形态的变化？

推荐序二

架构的架构

北京无限讯奇信息技术有限公司产品技术高级总监
黄冬

从编辑手里拿到厚厚的《架构之美》译稿时，恰巧是我刚刚讲完一场消息系统架构的讲座之后。而正是在昨天，一位想要创业的朋友跟我说要寻找一位懂得“架构”的高人与他一起创业。要知道与代码不同的是，“虚幻”的架构常常让人认为其有很多玄妙之处，只因它大多难以落在纸上。特别是与很多大师谈及架构时，经常落入他们的一些“陷阱”，并往往为自己达不到大师的精明与技巧而叹息。殊不知，被我们所津津乐道的这些架构，是他们在日常工作里经历了大量的错误、重复的尝试、无数的代码、长久的考验所积淀下来的只言片语。

本书将数十人的经历与只言片语，经过深思熟虑后抽象出规律，使之可以不断复用。而另一方面，又将架构的过程娓娓道来，尝试让读者思考架构的过程与思路。在这里，更多的过程与思考被展现出来，更多的原因与为什么让我们了解。

这本书里展现了很多绚丽的故事，犹如士兵阅读将军的传记一样，阅读本书将会让你更鼓起勇气追寻大师们的脚步，但永远要记住，每一滴汗水才真正是你成长路上的每一个记号，要在自己的工作里更深地去理解每一处不同，架构出属于自己的系统。

感谢译者和出版者为我们带来这样一本传奇的架构故事书。

推荐序三

美丽架构的含义

腾讯R&D研发总监 (Tencent Director Of R&D Development)
资深技术专家（Senior Technology Expert)
王速瑜

古人形容美女之美：“……增之一分则太长，减之一分则太短……”，深刻地揭示了“恰到好处”的美丽含义。当我拿到《架构之美》书稿时，我发现美丽的含义如此相似。

美丽至简。美丽的架构应尽可能简单，但不要过于简单。书中通过多种例子表达了这个最基本的道理。我见过很多大型的软件架构，从大型的电信网络管理系统，到大规模应用的互联网架构，以及企业级的ERP软件，系统总是遵循从无到有，从简单到复杂，再到简单这样的过程。最终，支撑这些大型系统稳定可靠运行的就是这个最基本的道理。

美丽的架构应尽可能精益，并且是演进式发展的。当你架构一个亿万人同时在线的大规模网站系统的时候，你无法从一开始就提供最完善的解决方案，它应该是随着用户的增长而可扩展的。精益的思想让你避免了过度设计，也使架构不断演进，趋于完美。书中从企业级应用架构、用户级应用架构等多个角度提供了相应的解决方案，对于架构师无不是一顿美味的大餐。

深夜看完这本书稿后，我发现，架构之美并不简单，它没有定法。但是，它将为架构师们提供一把进入“美丽架构艺术馆”大门的钥匙。拿起它，您将会开启这扇大门！

推荐序四

美丽架构之道

《构建高性能Web站点》作者
Web架构实践者
郭欣

我无法给架构下一个简单的定义，因为任何定义都会束缚你对架构的无限想象。不可否认，架构师早已出现在人类几千年前的各项生产活动中，比如建筑、音乐。而在计算机软件及Web领域，架构的设计直接影响着系统的生产，比如开发过程和效率、代码和组件复用性等，同时也影响着系统的可用性、可伸缩性、性能、容量可预测性等。

本书更加关注架构之美。美丽的架构同样无法定义，可它却一定是自然的、简单的、可复用的、人文的，甚至是外行人也可以细细品味其思想的。当我看到超市的多个收银台排满长队时，便想到服务器并发处理性能和容量；当我看到十字路口的车辆等待转弯时，便想到它通过缓存思想来提高交通吞吐率。

那么如何设计出美丽的架构呢？从代码逻辑到物理网络，从单机到分布式，无数的技术可供架构师选择，如分层、组件化、服务化、标准化、缓存、分离、队列、复制、冗余、代理等，不过它们仍然只是“术”的范畴，而何时何处如何恰到好处地使用它们才是“道”的范畴，比如顿悟变化的道理，在博弈中寻找平衡，以系统化的角度来分析问题，寻找相对与绝对的奥秘、开放的心态……

然而，这个领域实在是太年轻了，我们需要更多的例子和经验，本书将让你大开眼界！

译者序

架构与美

王海鹏

人们在生活和工作中发现美并创造美，软件开发和架构设计也不例外。

架构之美体现了关注点的分离与结合。在软件设计中，设计师需要考虑多方面的关注点。漂亮的架构设计让这些关注点尽可能分离，然后以最简单的机制结合在一起，从而得到高内聚、低耦合的系统。例如在Darkstar项目中，架构师们考虑的重点就是如何将多人在线游戏的游戏逻辑与系统的可伸缩性分离开来，让游戏的开发者只要遵守少量的规则，就能够像编写单机游戏一样编写大规模多人在线游戏。又如REST架构风格，体现了对资源命名、请求处理和资源物理表现形式的关注点分离。资源的名称与请求资源时服务器的处理方式无关，请求者无需知道服务器端采取的技术，并且请求者本来就不关心服务器端的处理技术。资源的物理表示形式可以通过内容协商来决定，使系统可以支持多种物理表示形式，并可以方便地扩展。

架构之美注重表达的简洁性。“Don’t Repeat Yourself”，好的架构致力于消除各种类型的信息重复。从结构化程序设计中的子程序和函数，到面向对象程序设计中的继承，无不体现了对表达简洁性的特殊偏爱。在敏捷方法学中，消除重复则是重构的主要目的之一。爱因斯坦说：“让它尽可能简单，但不要过于简单。”我们需要考虑所有必须考虑的关注点，然后用简洁漂亮的架构体现我们的关注。同时，简洁的架构之美也降低了软件的总体成本，从这个意义上说，“简洁性”又可以称为“经济性”。

架构之美需要解决实际问题，它既是艺术，也是生活。软件像建筑一样，它的美不能脱离它的实用价值。Bjarne Stroustrup说，人类文明运行于软件之上。每一个软件都有自己的架构，这些架构有的很美，有的不太美。从艺术的角度来说，美是创造矛盾并解决矛盾。架构的多关注点和表达简洁性就是一种矛盾，美丽的架构提供了这一矛盾的解决方法，让我们的内心产生一种愉快的感觉。

架构之美需要经过专业的学习才能更好地欣赏和创造。和所有的艺术之美一样，不是说不经过专业学习就不能欣赏，但是经过了专业的学习，就能更好地欣赏这种美的种种精妙之处。如果想要创造出这种美，那就必然要经过长期的专业学习。

架构之美经过时间打磨。像Facebook面向数据的Web服务、FQL和FML架构，是在对应不同实际需求的过程中逐渐发展起来。在应用程序架构形成的过程中，设计者不断面对新的关注点需求，不断对已有的架构进行修改，并发展出新的架构组件。这就是所谓的“演进式架构”。只有变化是永恒不变的。在架构设计初期，设计者会将一些关注点有意推迟到将来考虑，例如持久和并发。对于这些暂不考虑的关注点，设计者对它们的实现方式尽可能不做任何假定，从而保留更多的可能性，让不同关注点之间的耦合尽可能小。

架构之美没有定法。虽然有一些法则可供我们参考，却没有非如此不可的。《金刚经》云：“一切贤圣，皆以无为法而有差别。”

参加本书翻译工作的人员还有蔡黄辉、徐锋、王海燕、李国安、周建鸣、范俊、张海洲、谢伟奇、林冀、钱立强、甘莉萍。

在这本书的翻译过程中，我受益良多，因此郑重地向大家推荐它。

作译者简介

作者简介

Till Adam在年轻时学习了哲学、比较文学、美国研究和音乐学，职业是音乐人。由于没有发财和出名，他转而攻读科学硕士，学习了数学、计算机科学和商业。多年从事自由软件的经历（特别是对KDE的贡献）教会了他编程，也为他带来了在Klarälvdalens Datakonsult AB工作的机会，目前他在该公司负责协调KDE的开发和其他与自由软件相关的活动。他和他的妻子、女儿住在德国柏林。

Jim Blandy在1990年至1993年间为自由软件基金会维护GNU Emacs，和Richard Stallman一起发布了Emacs的第19个版本。他是Subversion版本控制系统的最初设计者之一。他也是CVS版本控制系统、GNU调试器（GDB）、Guile扩展语言库和一个编辑基因序列的Emacs程序的贡献者。他现在为Mozilla公司工作，工作内容是SpiderMonkey，即Mozilla的Javascript编程语言的实现。Jim和他的妻子、两个女儿住在俄勒冈州的波特兰。

Mirko Boehm从1997开始就是KDE的开发者，在1996年至2006年间是KDE e.V.委员会的成员。他毕业于德国汉堡Helmut Schmidt大学的商业专业。在他的闲暇时间里，他阅读纸版书籍、与家人在一起，试图远离计算机。他目前在德国柏林为Klarälvdalens Datakonsult AB工作，负责跨平台软件和嵌入式软件开发。

Christopher Dennis自2005年JCP项目开始时就是项目的主要开发者。Chris在牛津大学读博士时开始使用Java。此前，他使用过各种编程语言，从十六进制小键盘上编写的Z80机器码到PHP和JavaScript。他对特殊情况、编码技巧和偶尔有点丑陋的临时编码很有兴趣，喜欢用各种语言编写紧凑的、优雅的代码。

Dave Fetterman是Facebook的工程经理，他在那里创建了Facebook平台项目。在2006年加入Facebook之前，他是一名软件工程师，参加Microsoft开发者部门的项目，包括.NET的通用语言运行环境（CLR）。他喜欢为其他开发者创建软件，也喜欢对愿意听的人发表长篇大论。他拥有应用数学的学士学位，并在2003年获得了哈佛大学的计算机科学硕士学位。

Keir Fraser是XenSource的创始人之一，XenSource现在是Citrix Systems公司的一部分。他也是Xen系统管理程序的首席架构师。Keir在2002实现了Xen的第一个版本，作为他

在剑桥计算机实验室攻读博士学位时的一项娱乐。在该项目成为大规模的社区合作的过程中，他继续作为主要的开发者。他因在无锁并发控制方面的工作于2004年获得了博士学位，并在同年成为一名教师。

Pete Goodliffe是一名程序员、专栏作家、演说家和作家，从来不在同一软件领域做过多的停留。Pete的热门书籍《Code Craft》(No Starch Press)是对整个编程追求的实际而有趣的调查——大约600页，真是了不起！他对制革很有热情，而且不穿鞋。

Georgios Gousios是一名职业研究者，接受的教育和软件工程有关，热衷于软件开发。目前，他正在希腊的雅典经济与商业大学完成他的博士论文。他的研究兴趣包括软件工程、软件质量、虚拟机和操作系统，他拥有英国曼彻斯特大学的科学硕士学位。Gousios为多个开源软件项目贡献过代码，并参与了各种学术项目和商业项目的研究与开发。他是SQO-OSS项目的项目经理、设计权威和主要开发成员，为评估软件质量探索一些创新的方法。在他的学术生涯中，Gousios在会议和杂志上发表了10篇技术论文。Gousios是ACM、IEEE、Usenix Association和Technical Chamber of Greece的成员。

Dave Grove是IBM的T.J. Watson研究中心动态优化组的一名研究员。他的主要研究兴趣包括分析和优化面向对象语言、虚拟机设计和实现、JIT编译、在线反馈导向的优化和垃圾收集。他在1998年参加了Jalapeño项目，是这个优化编译器和适应式优化系统首个实现的主要贡献者。自Jalapeño在2001年作为Jikes RVM开放源码以来，他一直是Jikes RVM核心团队和指导委员会的活跃成员。

John Klein是软件工程研究所（SEI）的高级技术人员，他的研究方向是“众系统之系统”的架构方法，并帮助个人、团队和组织机构改进他们的软件架构能力。在加入SEI之前，John是Avaya公司的首席架构师。在Avaya，他负责开发多模式的代理、通信分析的架构，以及为各种客户交互产品创建并改进架构。在此之前，John是Quintus的一名软件架构师，在那里他设计了第一款获得商业成功的多渠道集成联系中心产品，并导致了Quintus兼并了另外两家公司，实现了产品组合的技术集成。在加入Quintus之前，John曾为多家视频会议和视频网络业的公司服务。他的职业生涯开始于Raytheon，在那里他为雷达信号处理、多光谱图像处理、并行处理架构和算法提供硬件和软件解决方案。John拥有Stevens技术学院的学士学位和Northeastern大学的硕士学位。他是ACM和IEEE计算机学会的成员。

Greg Lehey的漫长职业生涯在德国和澳大利业度过，他曾为德国空间研究所工作，也曾为Univac、Tandem、Siemens-Nixdorf和IBM等计算机制造商工作，也曾作为一些没名气的软件公司的大客户，还曾做过独立的咨询顾问。他的活动范围很广，包括从内核开发到产品管理，从系统编程到系统管理，从处理卫星数据到为油泵编程，从生产

CD-ROM到把自由软件移植到DSP指令集上。他是FreeBSD核心团队的成员，也是澳大利业UNIX用户协会的主席。他是FreeBSD和NetBSD项目的开发者，也是《Porting UNIX Software》和《The Complete FreeBSD, Fourth Edition 》(O'Reilly)的作者。他还以编写商业应用软件而闻名。Greg在2007年退休，将多出来的时间用于品味生活。现在，休闲活动占据了他的大多数时间，但这还不够，他还听古典木纹唱片、烹饪、酿啤酒（他开发了一个计算机控制的发酵系统）、做园艺、骑马和摄影。他也对一些历史题材感兴趣，包括古代的难解的欧洲语言。他的主页是：*http://www.lemis.com/grop/*。

Panagiotis Louridas在20世纪80年代通过一台Sinclair ZX Spectrum开始涉足计算机。从那时起，他就开始用机器语言进行编程，而且非常喜欢编程。他在雅典大学信息系获得了计算机科学学士学位，在曼彻斯特大学获得了计算机硕士和博士学位。这些年来，他一直为私人机构开发软件，现在，他在希腊研究和教育网络（GRNET）工作。他也是雅典经济与商业大学（AUEB）软件工程和安全（SENSE）研究组的成员。他发表的文章范围很广，从人类学到加密，从仪表展示到软件工程。他特别喜欢寻找计算机世界和其他领域的联系。

Stephen J. Mellor在为软件开发创建有效的工程方法方面，是国际公认的先行者。在1985年，他出版了广为阅读的Ward-Mellor三卷本《Structured Development for Real-Time Systems》(Prentice Hall)；在1998年，他的书首次定义了面向对象分析。Stephen还在2002年出版了《Executable UML: A Foundation for Model-Driven Architecture》(Addison-Wesley Professional)。他最近的一本书《MDA Distilled: Principles of Model-Driven Architecture》(Addison-Wesley Professional)在2004年出版。他在对象管理组织（OMG）中活动积极，是为UML添加可执行动作的协会的主席，他最近完成了可执行UML的标准。他是敏捷宣言的签名者之一。他是OMG架构委员会的两任成员，IEEE软件顾问委员会的主席，最近，他成为Mentor Graphics的嵌入式软件部门的首席科学家。

Bertrand Meyer是ETH Zurich的软件工程教授，也是Eiffel软件的首席架构师，他领导并设计了EiffelStudio环境和大量的库。他是一些畅销书的作者，其中包括获得Jolt大奖的《Object-Oriented Software Construction》(Prentice Hall)。他也因为在对象技术和Eiffel方面的工作获得了ACM软件系统大奖和Dahl-Nygaard大奖，并获得了St. Petersburg州立技术大学的荣誉博士学位。他的研究对象涉及面向对象技术、编程语言、软件验证（包括测试、并发和规范方法）。他也是一名活跃的顾问和讲师。

William J. Mitchell是MIT架构和媒体艺术与科学系的Alexander Dreyfoos教授，他领导着MIT媒体实验室和MIT设计实验室的Smart Cities团队。他以前曾担任MIT架构和计划学院的院长。他最近的新书是《World's Greatest Architect》和《Imagining MIT》（都由

MIT出版社出版）。

Derek Murray是剑桥大学计算机实验室的博士生。他在2006年加入Xen项目，主要工作是通过重新设计控制栈来改进Xen的安全性。他现在的研究主要是改进大规模分布式系统的容错性，但他还是偶尔会涉及系统核心。Derek在2006年从爱丁堡大学获得了高性能计算专业的硕士学位，2005年获得了Glasgow大学的计算机学士学位。

Rhys Newman在十多年前于牛津大学完成博士学位时，就开始使用Java，那时Java还只有几年历史。在他早期的研究中，他利用纯Java环境展示了高性能实时场景处理的实现方法，即使当时还是使用早期JIT化的JVM。从那时起，他同时在学界和业界工作，一次次证明Java平台实际上有多灵活、多高效、多快。在超过20年的软件工程生涯中，他获得了多个业界杰出技术奖项，最近他回到了牛津，承担了网格计算领域的突破性研究工作。JPC是最新研究工作的一部分。

Michael Nygard致力于在全国帮助开发者提高水平和减少痛苦。他和他遇到的每一个人分享他对改进的热情和活力，有时甚至没有得到对方的同意。Michael花了20年中的大部分时间学习对专业程序员有意义的事，他关心艺术、质量和技艺。他总是愿意在那些全职的、真心投入工作的开发者（那些“觉醒的”开发者）身上花时间。在另一方面，他不能容忍缺乏兴趣或浪费潜力。Michael在近20年来一直是专业的程序员和架构师。在这段时间里，他为美国政府、军方、银行、金融业、农业和零售业交付了运营系统。通常，Michael都要面对他自己开发的系统。这种实际运营的经历改变了他对软件架构和开发的看法。他参与了一个Tier 1零售网站的初期开发，并且常常作为其他在线业务的“流动解决问题专家”。这些经验让他对在相当不友好的环境下构建高性能、高可靠性的软件有了独特的看法。最近，Michael编写了《Release It! Design and Deploy Production-Ready Software》(Pragmatic Programmers)，该书获得了2008年的Jolt生产力大奖。他的其他文字可以在*http://www.michaelnygard.com/blog*上读到。

Ian Rogers是曼彻斯特大学高级处理器技术研究组的研究员。他的博士研究工作是关于Dynamite二进制翻译器的，该技术实现了商用，现在是许多二进制翻译器产品的一部分，包括Apple的Rosetta。他最近的学术研究工作一直是编程语言设计、运行时环境和虚拟机环境，特别是如何自动创建它们并有效地使用并行技术。他是Jikes研究虚拟机的主要贡献者，是开发团队的核心成员。

Brian Sletten是自由的、受过艺术教育的软件工程师，关注forward-learning技术。他曾担任过系统架构师、开发者、现场指导者和培训师。他在世界各地的会议上发表演讲，

编辑注：此书由机械工业出版社引进出版，书名为《代码质量》（注释版），书号为：978-7-111-22671-0。

并为一些在线出版物编写关于面向Web技术的文章。他的经验涉及国防、金融和商业领域。他曾设计并建造了网络矩阵式交换控制系统、在线游戏、3D仿真/可视化环境、因特网分布式计算平台、P2P和基于Web的语义系统。他拥有William and Mary大学的计算机科学学士学位，目前居住在弗吉尼亚的Fairfax。他是Bosatsu咨询公司的总裁，该公司为Web架构、面向资源的计算、语义Web、高级用户界面、可伸缩系统、安全和其他20世纪末21世纪初的技术提供专业的咨询服务。

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、计算机安全和编程语言。他也编写了两本“开放源码方面”的书，由Addison-Wesley出版：《Code Reading》（获得了2004年的软件开发生产力大奖）和《Code Quality》（获得了2007年软件开发生产力大奖，编辑注）。他也写了几十篇科学论文。他是IEEE Software编辑委员会的成员，负责定期的“Tools of the Trade”栏目。Diomidis是FreeBSD的提交者，也是UMLGraph和其他开源软件包、库和工具的开发者。他拥有软件工程的硕士学位和计算机科学博士学位，都是在Imperial College London获得的。Diomidis是ACM的高级成员，也是IEEE和Usenix Association的成员。

Jim Waldo是Sun微系统实验室的杰出工程师，负责研究下一代大规模分布式系统。他目前是Project Darkstar的技术负责人，该系统是针对大规模多人在线游戏和虚拟世界而设计的多线程、分布式基础设施。在此之前，他曾是Jini的首席架构师，Jini是基于Java的分布式编程系统。Jim编写了《The Evolution of C++: Language Design in the Marketplace of Ideas》（MIT出版社），也是《The Jini Specification《Addison-Wesley》的合著者之一。他曾是美国国家学术委员会的共同主席，编辑并出版了《Engaging Privacy and Information Technology in a Digital Age》一书。Jim也是哈佛大学的辅助教师，在计算机科学系教授分布式计算和策略与技术相关的内容。Jim拥有马萨诸塞大学（Amherst）的哲学博士学位。

David Weiss拥有Union College的计算机科学学士学位，并拥有马里兰大学的计算机科学硕士和博士学位。他目前是Avaya实验室的软件技术研究部的领导，他关注软件开发效率改进的普遍问题和Avaya软件开发过程改进的特殊问题。在第二个问题上，他领导了Avaya软件技术研究中心。以前，他曾是朗迅技术贝尔实验室软件生产研究部的主任，该部门负责研究如何改进软件开发的效率。在加入贝尔实验室之前，他是软件生产力协会（SPC）复用和度量部门的主任，该协会由14个大型的美国航空公司组成。在加入SPC之前，Weiss博士在技术评估办公室度过了一年的时间，在那里他与同事共同完成了Strategic Defense Initiative的技术评估。在1985—1986学年，他是Wang Institute的访问学者，在许多年里，他一直是华盛顿特区Naval研究实验室（NRL）计算机科学和系统部门的研究员。他也是一名程序员和数学家。Dave的主要研究方向是软件工程领域，特

别是软件开发过程和方法学、软件设计和软件测量。他最为人知的是发明了软件测量的“目标-问题-测量指标”方法，软件系统模块化结构的工作，以及软件生产线工程的工作。他是Synthesis过程及其后继FAST过程的共同发明人。他与别人共同编著了两本书：《Software Product-Line Engineering》和《Software Fundamentals: Collected Papers of David L. Parnas》（都由Addison-Wesley出版）。

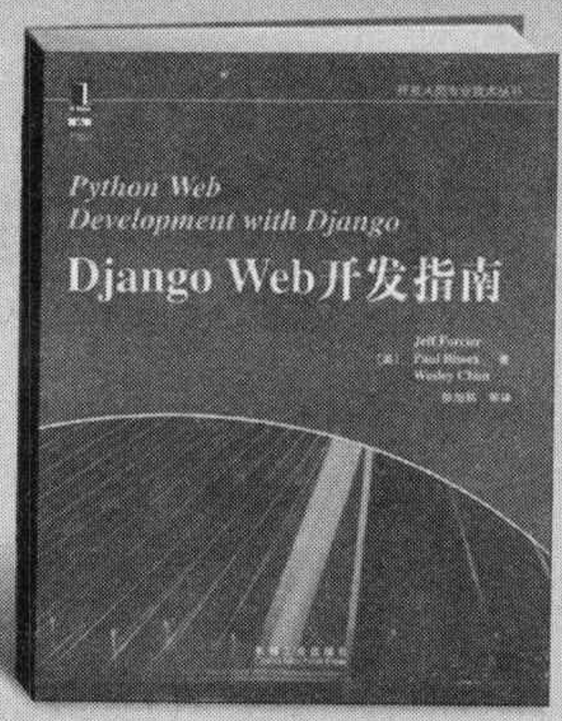
译者简介

王海鹏　1994年毕业于华东师范大学。拥有理学士（物理）和文学士（英国语言文学）学位。独立的咨询顾问、培训讲师、译者和软件开发者。已翻译十余本软件开发书籍，主题涵盖敏捷方法学、需求工程、UML建模和测试。拥有15年软件开发经验，目前主要的研究领域是软件架构和方法学，致力于提高软件开发的品质和效率。

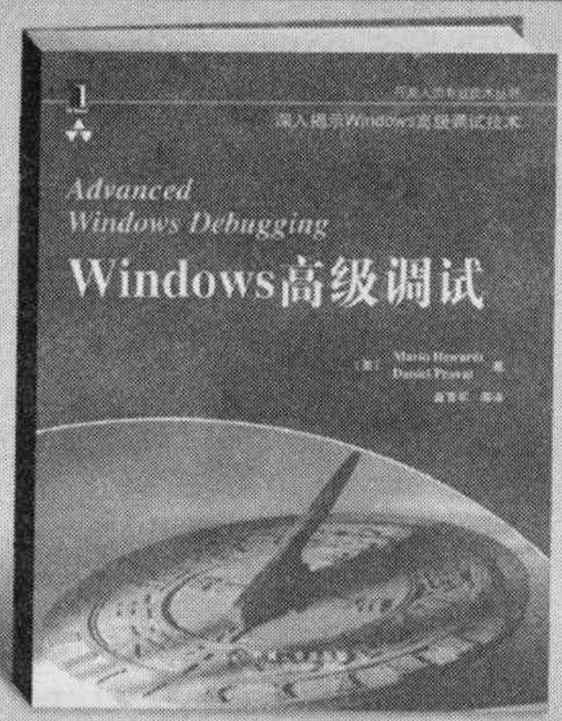
蔡黄辉　江苏启东人。1999年毕业于上海交通大学，毕业后一直从事软件开发工作，主要使用Java做Web方面的底层开发。现居住在上海。联系方式：caihuanghui @ hotmail.com。

徐锋　中国系统分析员顾问团（CSAI）软件工程首席顾问，中国软件技术大会杰出贡献专家，资深咨询顾问。主要研究领域为需求工程、系统分析与设计、软件估算，致力于推动软件工程方法论的落地应用。曾在《程序员》等媒体发表了《实战OO》、《项目管理三步曲》、《大话Design》等多个专栏文章，著有《软件需求最佳实践》、《UML面向对象建模基础》等多本书籍，翻译了《UML 2.0实战》、《AOSD中文版》、《Cloud to Code中文版》等多本相关技术书籍。

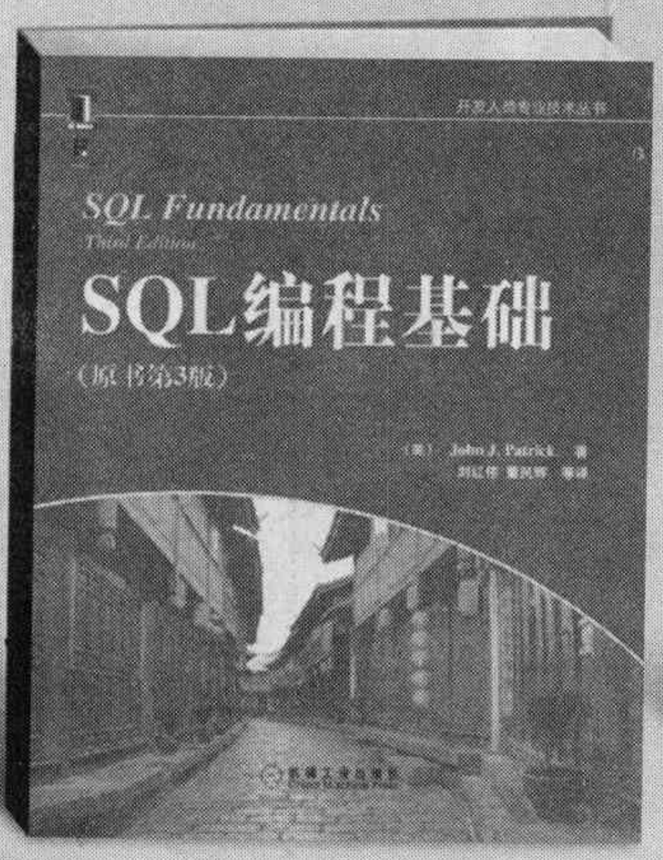
开发人员专业技术丛书



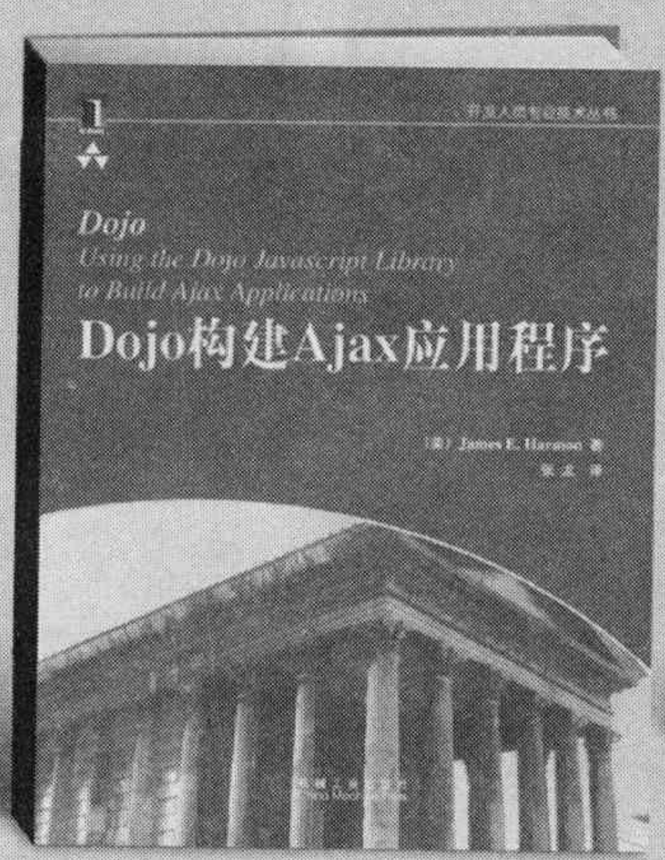
书号：978-7-111-27028-7
定价：49.00元



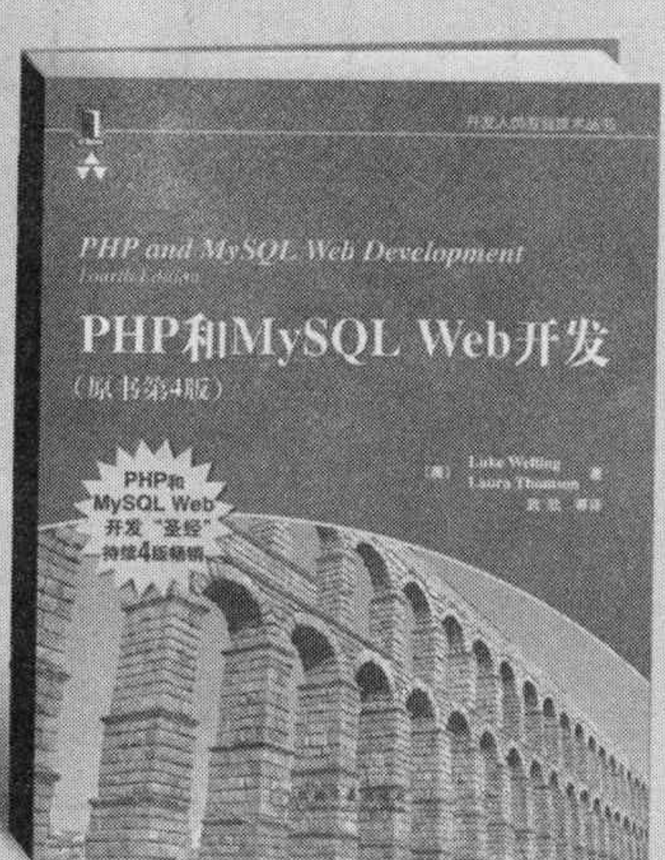
书号：978-7-111-26639-6
定价：79.00元



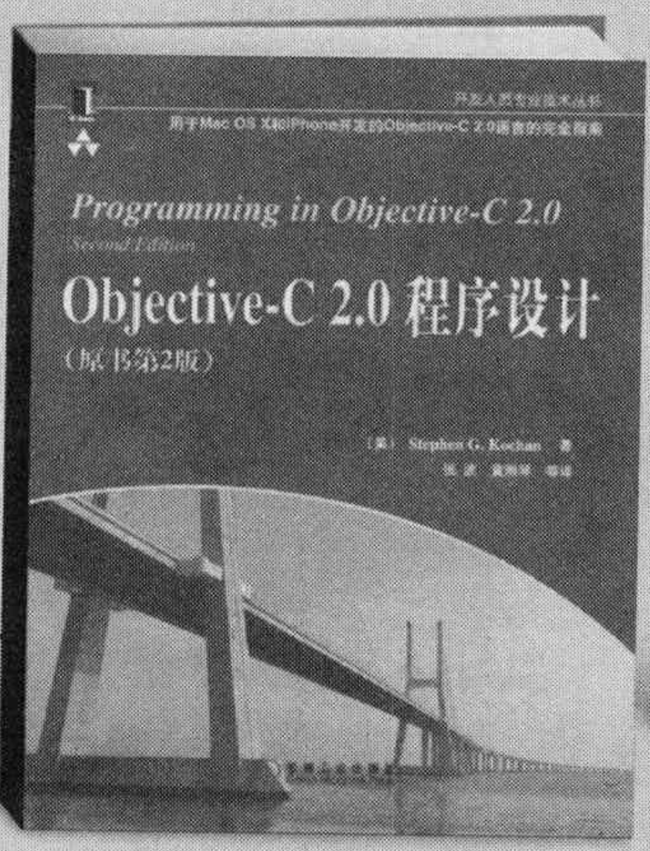
书号：978-7-111-26541-2
定价：69.00元



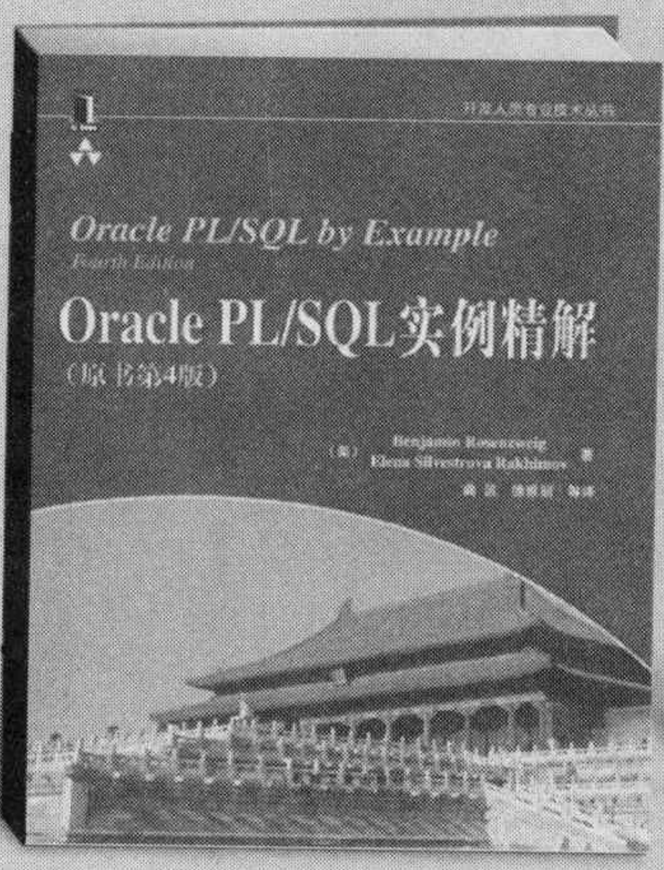
书号：978-7-111-26664-8
定价：45.00元



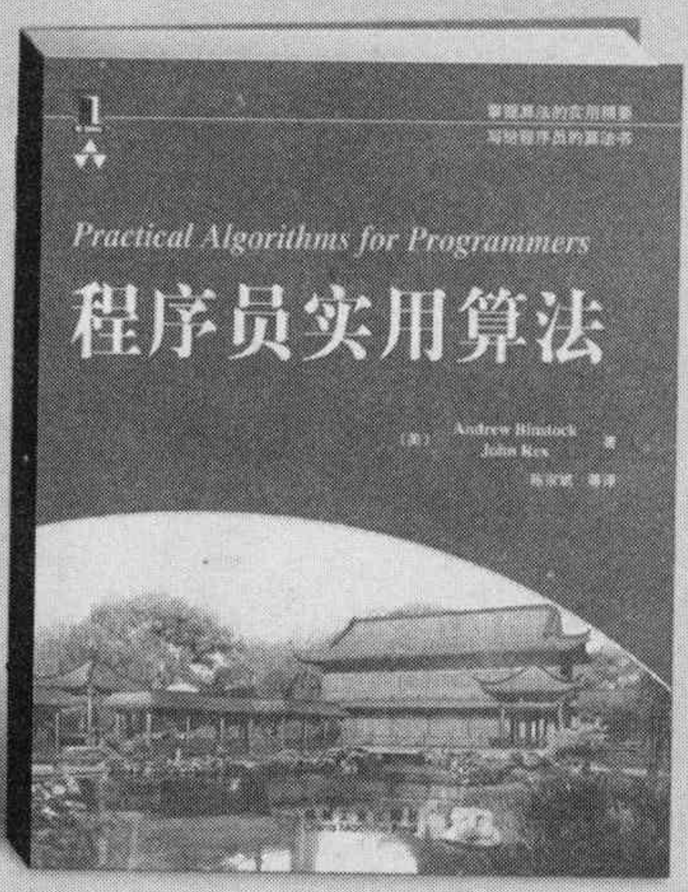
书号：978-7-111-26281-7
定价：95.00元



书号：978-7-111-27686-9
定价：66.00元



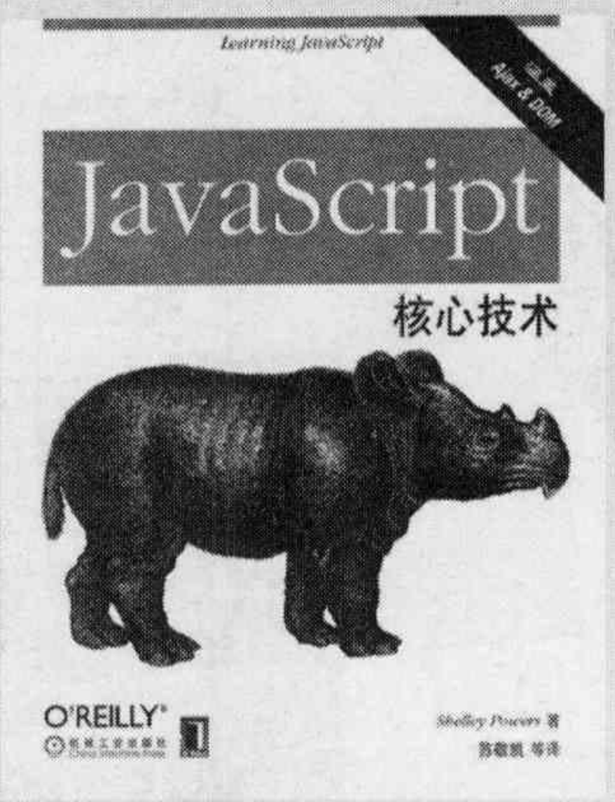
书号：978-7-111-26803-1
定价：85.00元



书号：978-7-111-267396-0
定价：65.00元

一本打开的书，
一扇开启的门，
通向科学圣殿的阶梯，
托起一流人才的基石。

O'Reilly系列专业图书盛宴



1 **JaveScript权威指南 第5版**
作者:David Flanagan
书号:978-7-111-21632-2
定价:109.00元

2 **JaveScript核心技术**
作者:Shelley Powers
书号:978-7-111-21297-3
定价:45.00元

3 **ActionScript 3.0编程精髓**
作者:Colin Moock
书号:978-7-111-23992-5
定价:99.00元

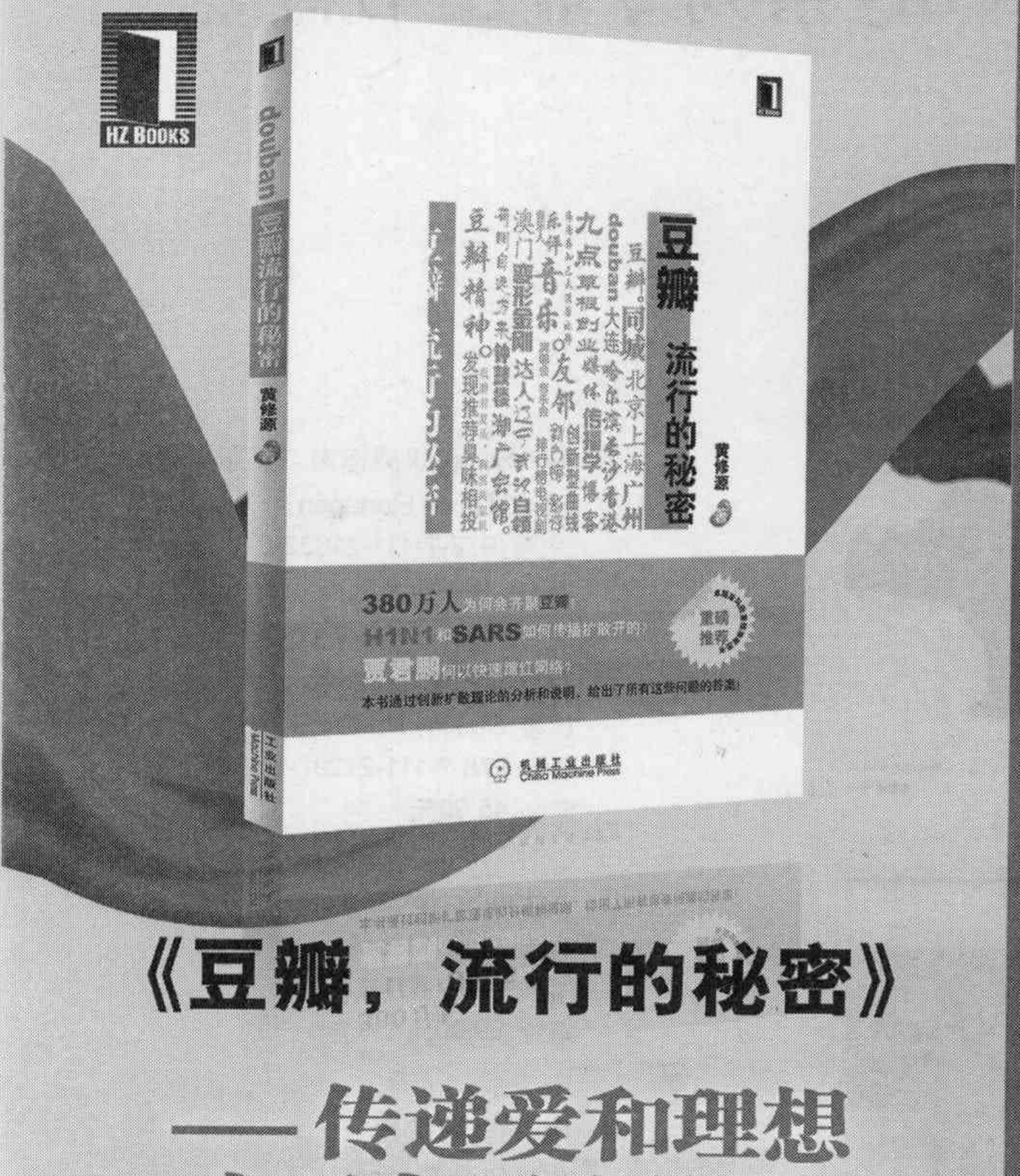
4 **Rails高级编程**
作者:Brad Ediger
书号:978-7-111-24601-5
定价:55.00元

学习Ruby
作者:Michael Fitzgerald
书号:978-7-111-22371-5
定价:36.00元

C语言核心技术
作者:Peter Prinz,Tony Crawford
书号:978-7-111-22050-3
定价:69.00元

学习Web设计
作者:Jennifer Niederst Robbins
书号:978-7-111-23876-6
定价:65.00元

学习PHP和MySQL
作者:Michele E.Davis
书号:978-7-111-24081-5
定价:55.00元



《豆瓣，流行的秘密》

——传递爱和理想
一本正在漂流的图书

当你在KFC、麦当劳、星巴克甚至在地铁站里发现一本《豆瓣，流行的秘密》时，不要以为这是一本被丢弃的图书，请把它拣起来，这是一本正在全国漂流的图书，这本漂流的图书，它传播的不仅是一位年轻创业者的理想，也传承着一种阅读的文化。

看了这本书，也许不能让你的产品或网站突然流行起来，但至少，它能让你知道你的产品或网站为什么没能流行起来。下一次，也许，你也能创造出一个大流行。

如果你想了解为什么很多网站吸引客户的秘诀，如果你想知道为什么长尾理论只会在互联网的产品中出现，那么我推荐你阅读这本书籍。它很短小，语言简单，例子很实际，它希望用最简单的语言告诉你一件我已经发现的或许你还不知道的秘密，你可以在闲暇的周末泡杯咖啡，一边慢慢喝一边阅读它，我想这会是一个很充实的周末。

作者黄修源，正在和他的团队一起在实践很多书中所说的理论，番茄树正是他们正在运营的网站。2个月的时间从ALEXA的100多万名到24000名，没有花一分钱广告费，“其实我们只是在实践我写的书中的一些有意思的理论而已！”修源这样说。

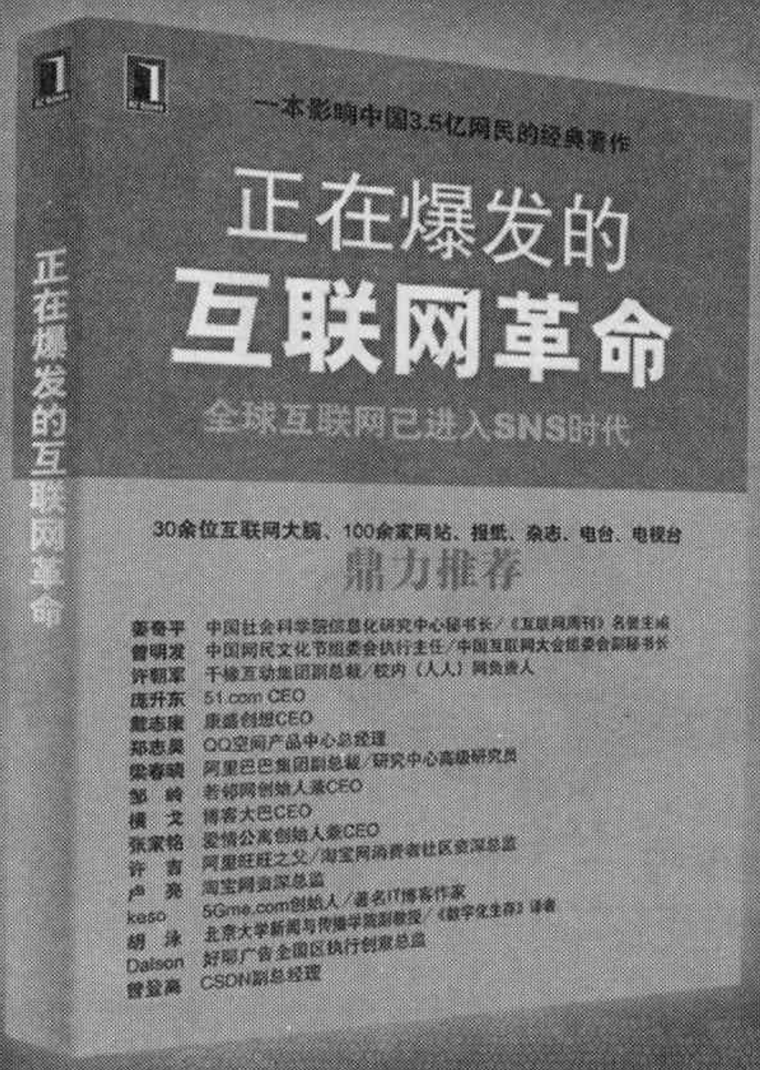
正在爆发的互联网革命

短短20年里，互联网经历了门户网站和Web 2.0时代的重大变革，如今已经迈入SNS时代。新一轮的互联网革命即将爆发，而其影响已经开始……

1. 为何偷菜、抢车位这样的无聊小游戏能让无数智商正常的人痴迷?
2. 为何全球互联网巨头都为SNS而疯狂?
3. SNS掀起的第三次互联网革命能给我们带来哪些机遇?
4. 你想知道奥巴马是如何利用SNS问鼎美国总统宝座的吗?
5. 为何流行天王迈克尔·杰克逊去世不到一小时，噩耗就传遍了全世界?
6. 为何“你妈妈喊你回家吃饭”能在几天时间内红遍整个网络?
7. SNS能让我们免费环游世界，你相信吗?
8. 为何好莱坞的大导演们热衷于以Facebook和Twitter为题材的电影?
9. 你还在通过报纸和门户网站等传统渠道获取每天的新闻消息?
10. 为了找工作，你还去人山人海的招聘会现场排队?还去51job海发简历?
11. 什么是“微弱关系”?为何它有时比多年的“老交情”还管用?
12. SNS开创的免费营销时代已来临，你还在为推广你的产品砸下巨额的广告费?

……

这一切都与SNS有关，都能在本书中找到答案!



这将是一本影响中国3.5亿网民的经典著作，国内30余位互联网大腕一致推荐：姜奇平（中国社会科学院信息化研究中心秘书长）、许朝军、庞升东、戴志康、梁春晓（阿里巴巴集团副总裁）、郑志昊（QQ空间产品中心总经理）、邹岭（若邻网创始人兼CEO）、横戈（博客大巴CEO）、张家铭（爱情公寓创始人兼CEO）、许吉（阿里旺旺之父）、keso、胡泳（北京大学新闻与传播学院副教授/《数字化生存》译者）、曾登高、卢亮（淘宝网资深总监）、Dalson（好耶广告全国区执行创意总监）、李国德（康盛UCenter Home创始人）、韩笑（奇虎网络社区研究机构总经理）、梁柱（腾讯创新中心资深总监）、蒲忠杰（雅虎口碑SNS产品总监）、陶为民（安瑞索思ECD）、姚鸿（POCO.CN CEO）、刘勇（SNS网站亿友网创始人/SNS APP服务商热酷创始人）、姬十三（科学作家/科学松鼠会创始人）、陆贤清（上汽乘用车媒介与互动营销高级经理）、图王（中国最有声望的个人站长之一）……

同时，本书还得到了《IT时代周刊》、《计世资讯》、《互联网周刊》、《销售与市场》、《北京晚报》、《沈阳晚报》等100余家网站、报纸、杂志、电台、电视台及其相关负责人的鼎力推荐。

www.hzbook.com

填写读者调查表　加入华章书友会
获赠精彩技术书　参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：　　书号：7-111-(　　)

姓名：	性别：□男　□女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

□朋友推荐　□书店　□图书目录　□杂志、报纸、网络等　□其他

2. 您从哪里购买本书：

□新华书店　□计算机专业书店　□网上书店　□其他

3. 您对本书的评价是：

技术内容	□很好	□一般	□较差	□理由_____
文字质量	□很好	□一般	□较差	□理由_____
版式封面	□很好	□一般	□较差	□理由_____
印装质量	□很好	□一般	□较差	□理由_____
图书定价	□太高	□合适	□较低	□理由_____

4. 您希望我们的图书在哪些方面进行改进？

_____

_____

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

_____

_____

6. 您有没有写作或翻译技术图书的想法？

□是，我的计划是_____　□否

7. 您希望获取图书信息的形式：

□邮件　□信函　□短信　□其他_____

请寄：北京市西城区百万庄南街1号　机械工业出版社　华章公司　计算机图书策划部收

邮编：100037　电话：（010）88379512　传真：（010）68311602　E-mail: hzjsj@hzbook.com

目录

序

Stephen J. Mellor

使用随意的、非正式的工程技术去开发高性能、高可靠性和高品质的软件系统会遇到非常多的困难。这些技术对于过去要求较低的系统也许还能对付，而目前系统的复杂性已经达到了这样一种程度：如果不开发并维护一个基础架构，利用它将系统组织成一致的整体并避免零碎的实现，那么我们就无法应对，必将导致测试和集成失败。

但是，建立一个架构是一项复杂的任务。很难得到合适的例子，因为要么必须考虑专有权，要么刚好相反，有人需要向各式各样的环境推销某一种架构风格。架构是很大的概念，这使得很难在不吓坏读者的情况下来记录或描述。

但是，美丽的架构展示了一些普遍原则，下面我列出了几点：

一处一个事实

重复导致错误，所以应该避免。每个事实应该是单一的、不可分解的单元，每个事实必须独立于其他事实。当改变发生时（这是不可避免的），只有一个地方需要修改。数据库设计者很熟悉这一原则，它以范式（normalization）之名规范化了。这个原则也不那么规范地应用于行为，名为提取公因式（factoring），即同样的功能提取出来，放到独立的模块中。

美丽的架构找到了一些方法来定位信息和行为。在运行时，形成了分层，即系统可以按层来划分，每个层代表了一层抽象（abstraction）或一个领域（domain）。

自动传播

一处一个事实听起来不错，但出于效率的考虑，某些数据或行为常常会重复。为了维护一致性和正确性，这些事实的传播必须在构建时自动进行。

美丽的架构是由一些构建工具支持的，结果导致了*元编程*（metaprogramming），将一个事实从一处传播到多处，在那些地方有效地使用它们。

架构也包含构建

架构不仅包含运行时系统，而且必须包含它的构建方式。只关注运行时代码会导致架构随时间的推移而退化。

美丽的架构是经过深思熟虑的。它们不仅在运行时是美丽的，在构建时也是美丽的；利用同样的数据、函数和技术来构建系统，就像在运行时使用的那样。

最少量机制

实现某个功能的最佳方式要视情况而定，但是美丽的架构不会追求“最佳”。例如，有许多种方式可以存储和查找数据，但如果系统利用一种机制达到了性能要求，就会考虑编写、验证和维护较少的代码，以及占用较少的内存。

美丽的架构使用一组最少的机制来满足整体的需求。找到每种情况下的“最佳”，会导致各种容易出错的机制的产生，而用极简的方式来添加机制则会得到更小的、更快的、更健壮的系统。

构建引擎

如果您希望构建脆弱的系统，就采用Ivar Jacobson的建议，将架构建立在用例的基础上，每次实现一个功能（例如，使用“控制器”对象）。但是，可扩展的系统依赖于虚拟机的构建，即由更高层提供的数据来“编程”的引擎，一次实现多个应用功能。

这个原则以多种外观出现。虚拟机的“分层”可以追溯到Edsger Dijkstra。“数据驱动的系统”提供了引擎，依赖于系统中的编码常量，让数据来定义特定情况下的具体功能。这些引擎是高度可复用的，也是美丽的。

O(G)，增长的阶

在以前，我们会考虑算法的“阶”，例如，根据对一组一定数量的元素进行排序的时间来分析排序的性能。许多书籍都是围绕这个主题来写的。

这同样适用于架构。例如，一个投票程序在少量数据的情况下工作得很好，但数据量增大时响应时间就变得难以接受。按照中断或事件来组织所有的工作，在开始时

会工作得很好，但后来可能突然出现问题。美丽的架构会考虑可能的增长方向，并能够支持这种增长。

抵制熵增

美丽的架构设计了一条最容易维护的路线，随着时间的推移仍能够保持架构，所以延缓了“系统熵增定律”的效果。该定律指出，系统会随着时间的推移变得越来越乱。维护者必须适应该架构，这样变更才能与架构保持一致，不增加系统的熵。

一种方法是利用敏捷开发的隐喻（Metaphor）概念，它是一种简单的方式，说明架构像什么。另一种方法是使用大量的文档，并以解雇相威胁，虽然这种方法难以取得长久的效果。然而，这通常与工具有关系，特别是那些生成系统的工具。美丽的架构必须保持美丽。

这些原则是高度相关的。只有在你实现了自动传播时，一处一个事实才行得通，而自动传播又只有在架构考虑到构建时才有效果。类似地，构建引擎和最少量机制支持一处一个事实。抵制熵增是长时间保持架构的要求，它靠的是架构包含构建方法并支持自动传播。而且，如果忘记考虑系统将以何种方式增长，将导致架构不稳定，最终会在极端但可预见的环境下失败。将最少量机制与构建引擎的观点结合起来，就意味着美丽的架构通常包含一组有限的模式，同时支持构建任意的系统扩展，这相当于“按模式扩展”。简而言之，美丽的架构用更少的机制做更多的工作。

本书由Diomidis Spinellis和Georgios Gousios汇编而成，当你阅读本书时，你可以通过每章提供的具体例子，寻找这些原则并思考它们的意义。你也可以寻找违反这些原则的情况，想想这是否导致架构变得丑陋，或涉及某些更高的原则。

在写这篇序时，作者问我能否说说如何才能成为好的架构师。我笑了起来。如果我知道的话……但是接着我从自己的经历中想到，有一种特效方法（注）可以帮助人们成为美丽的架构师。这种方法就是永远不要相信你最近创建的系统是唯一的，应设法寻找不同方法来解决相同类型的问题。本书提供的美丽架构的示例就能帮助你朝着实现这个目标迈出一步。

注：　或者是锻炼更多，吃得更少。

前言

出版本书的想法是在2007年确立的，它是获奖的畅销书《Beautiful Code》（编辑注）的姊妹篇。在这本书中，虽然范围和目的不一样，但关注点是类似的：让最优秀的设计师和架构师来描述他们所选的软件架构，剥开他们作品的各层，展示他们如何让软件实现功能性、可靠性、易用性、高效率、可维护性、可移植性，当然，还有优雅性。

为了编成这本书，我们联系了一些著名软件项目的主要架构师和一些不太著名但高度创新的软件项目的主要架构师。许多人快速回应，将引人思索的想法寄回给我们。有些想法甚至让我们大吃一惊，他们建议不要写具体的系统，而是调查架构在软件工程中产生影响的深度和广度。

当本书的所有作者听说这本书的版税将捐给Medécins Sans Frontières（无国界医生组织）时，都感到十分高兴。Medécins Sans Frontières是一个国际人道主义援助组织，为困难的人提供紧急医疗救助。

本书的组织

我们围绕5个主题领域来组织本书的内容：概述、企业应用、系统、最终用户应用和编程语言。很明显，缺少有关桌面软件架构的章节，但这不是故意的。我们联系了50多位

编辑注：本书由机械工业出版社引进出版，其中文版书名为《代码之美》，标准书号为978-7-111-25133-0。

软件架构师，这个结果让我们吃惊不小。难道真的没有美丽桌面软件架构的漂亮例子吗？或者那些天才的架构师避而不谈架构，是因为忙着应付需求，为应用堆彻更多的功能？我们非常期望听听你对于这些问题的见解。

第一部分：论架构

本书的第一部分探讨了软件架构的广度和范围，以及它对软件的开发和演进意味着什么。

第1章由John Klein和David Weiss编写，他们从架构的品质考虑和架构结构的角度，对架构进行了探讨。

第2章由Pete Goodliffe编写，他写了一篇寓言，揭示了软件架构如何影响系统的演进和开发者在项目中的参与情况。

第二部分：企业级应用架构

企业级系统是许多组织机构的IT支柱，它们不仅巨大，而且常常是度身定制的软件集，一般由许多分散的组件构成。它们服务于大量的、支持事务的工作，必须延伸到它们所支持的企业的各个角落，时刻准备着适应不断变化的业务需求。在为这样的系统设计架构时，可伸缩性、正确性、稳定性和可扩展性是最重要的关注点。本书的第二部分包含了一些企业级软件架构的优秀范例。

第3章由Jim Waldo编写，展示了构建大规模多人在线游戏所需的架构技术。

第4章由Michael Nygard编写，介绍了一个多阶段、多地点的数据处理系统的架构，展示了让系统能工作所必需的折中。

第5章由Brian Sletten编写，讨论了创建数据驱动的应用时资源映射的威力，提供了纯面向资源架构的一个优雅的例子。

第6章由Dave Fetterman编写，他提倡以数据为中心的系统，解释了好的架构如何能够创造并支持应用生态系统。

第三部分：系统架构

系统软件可能是在设计方面要求最高的软件，部分原因是因为高效率地使用硬件是少数人才能掌握的神秘艺术，还有部分原因是因为许多人认为系统软件的架构是理所当然的存在。了不起的系统架构很少是从一张白纸开始的，我们今天使用的大多数系统是基于20世纪60年代的设想。第三部分的这几章介绍了4种创新的系统架构，讨论了架构决定背后的复杂性，这正是它们美丽的原因。

第7章由Derek Murray和Keir Fraser编写，提供了一个例子说明深思熟虑的架构如何能够改变操作系统演进的方式。

第8章由Greg Lehey编写，回顾了Tandem的架构选择和组成部分（包括硬件和软件），正是这些让Tandem成为近二十年的高可用性环境的首选平台。

第9章由Rhys Newman和Christopher Dennis编写，描述了如何通过小心地设计软件和很好地理解领域需求来克服编程系统中那些可以察觉到的不足。

第10章由Ian Rogers和Dave Grove编写，介绍了为高级语言创建自优化的、自支持的运行时环境所需的架构选择。

第四部分：最终用户应用架构

最终用户应用程序是我们每天的计算生活都要使用的那些应用程序，是占据我们最多CPU指令周期的软件。这种软件通常不需要仔细地管理资源，也不需要执行大量的事务。但是，它需要易于使用、安全、可定制和可扩展。这些属性可以导致软件广泛流行和使用，以免费开源软件为例，会有大量的自愿者愿意改进它。在第四部分，作者们分析了两个非常流行的桌面软件包的架构和需要的社区过程。

第11章由Jim Blandy编写，解释了一组非常简单的组件和一门扩展语言如何将一个不起眼的文本编辑器变成了一个操作系统（注），成为程序员工具箱中的瑞士军刀。

第12章由Till Adam和Mirko Boehm编写，展示了冲刺和同级评审这样的社区过程如何帮助软件架构从简单的骨架演变为美丽的系统。

第五部分：语言与架构

正如许多人在他们的著作中指出的那样，我们使用的编程语言影响了我们解决问题的方式。但是编程语言也能影响系统的架构吗？如果是这样，那么是如何影响的？在建筑的架构中，新的材料和CAD系统的采用让人们能够表达更复杂、有时更惊人的漂亮设计。对于计算机编程来说也是如此吗？第五部分包含最后的两章，探讨了我们使用的工具和得到的设计之间的关系。

第13章由Bertrand Meyer编写，比较了面向对象和函数式架构风格的适用性。

第14章由Panagiotis Louridas编写，探讨了现代和经典面向对象软件语言的组件背后的架构选择。

最后，在引人深思的跋中，William J. Mitchell（MIT架构与媒体艺术科学系教授）将真实世界中建筑架构的美丽概念与硅基芯片上软件架构的美丽概念联系在了一起。

注：　正如某些忠心的用户所说的："Emacs是我的操作系统，Linux只是提供了设备驱动。"

原则、特性与结构

在这本书评阅过程的后期，一位审阅者要求我们提供个人的观点作为一种补充，告诉读者可以从每一章中学到些什么。这个想法很有趣，但我们不想再猜测各章作者的意思。要求作者们自己提供他们文章的元分析，就会得到一堆的定义、术语和架构结构，这肯定把读者搞糊涂。我们需要一组通用的架构术语，谢天谢地，我们意识到我们已经有了。

在序中，Stephen Mellor讨论了7个原则，所有美丽的架构都基于这7个原则。在第1章中，John Klein和David Weiss提供了4种架构组件和美丽架构的6种特性。细心的读者会发现，Mellor的原则与Klein和Weiss的特性不是彼此无关的。实际上，它们大多数是一致的，这是因为英雄所见略同。他们三人都是非常有经验的架构师，人们曾多次在实际工作中看到过他们描述的概念的重要性。

我们将Mellor的架构原则与Klein和Weiss合并成为两个列表：一个包含了原则和特性（表P-1），另一个包含了结构（表P-2）。然后我们要求各章的作者标出他们认为适用于自己那一章的术语，得到了每一章对应的说明。在这些表中，你可以看到各章说明中出现的每个原则、特性或架构结构的定义。我们希望这些说明对你的阅读提供帮助，它们清晰地总结每一章的内容。但是我们强烈建议你深入阅读每章的内容，而不只是停留在这些说明上。

表P-1：架构原则与特性

原则或特性	架构能够……
功能多样性	……提供“足够好”的机制，利用简洁的表达来处理各种问题
概念完整性	……提供单一的、最优的、无冗余的方式来表示一组类似问题的解决方案
修改独立性	……保持它的元素的独立性，这样就能够让需要的修改最少，从而适应变化
自动传播	……通过在模块之间传播数据或行为，保持一致性和正确性
可构建性	……指导软件进行一致、正确的构建
增长适应性	……考虑到可能的增长
熵增抵抗力	……通过适应、限制和隔离变化的影响来保持有序

表P-2：架构结构

结构	结构能够……
模块	……将设计或实现决定隐藏在一个稳定的接口之后
依赖关系	……按照一个模块使用另一个模块的功能的方式来组织模块
进程	……封装并隔离一个模块的运行时状态
数据访问	……隔离数据，设置数据访问权限

本书体例

我们在本书中使用下面的印刷惯例：

斜体（*Italic*）

表示URL、邮件地址。

等宽字体（Constant width）

用于程序清单，以及在段落中引用程序元素，如变量名或函数名、数据库、数据类型、环境变量、语句和关键字。

等宽粗体（**Constant width bold**）

表示命令或其他由用户输入的文本。

等宽斜体（*Constant width italic*）

表示应该由用户输入的值或上下文相关的值来替代的文本。

使用代码示例

这本书的目的是帮助你完成自己的工作。一般来说，你可以在程序中使用这本书中的代码和文档。你不需要联系我们获得许可，除非你打算复制大量的代码。例如，写一个程序使用本书中的几段代码是不需要获得许可的。销售或发行来自O'Reilly书籍的示例光盘需要许可。引用本书来回答问题或引用示例代码不需要许可。在产品文档中包含大量本书的示例代码需要获得许可。

我们希望在引用时说明来源，但这不是必需的。来源说明通常包括标题、作者、出版商和ISBN号。例如："*Beautiful Architecture*, edited by Diomidis Spinellis and Georgios Gousios. Copyright 2009 O'Reilly Media, Inc., 978-0-596-51798-4."

如果你觉得对示例代码的使用超过了这里的许可，请通过permissions@oreilly.com联系我们。

我们的联系方式

请把涉及本书的评论和疑问邮寄至以下地址：

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

100035北京市西城区西直门南大街2号成铭大厦C座807室

奥莱利技术咨询（北京）有限公司

我们为本书制作了网页，上面列出了勘误表、示例和任何附加的信息。该网页地址为：

http://www.oreilly.com/catalog/9780596517984

http://www.oreilly.com.cn/book.php?bn=978-7-111-28356-0

为了评论或获得本书的技术支持，请发E-mail至以下地址：

bookquestions@oreilly.com

info@mail.oreilly.com.cm

欲获得其他图书、会议、资源中心和O'Reilly网络的更多信息，请访问以下网址：

http://www.oreilly.com

致谢

一本书的出版是一个团队努力的结果，一本合集更是如此。我们要感谢许多人。首先，我们要感谢本书的诸位作者，他们及时地提供了优秀的素材，然后满足了我们多次改动和修订的要求。这本书的审阅者包括Robert A. Maksimchuk、Gary Pollice、David West、Greg Wilson和Bobbi Young，他们给出了许多极好的建议，改进了每一章和整本书。在O'Reilly，我们的编辑Mary Treseler帮助我们联系作者，组织复审过程，高效地管理本书的出版过程。后来，Sarah Schneider作为本书的制作编辑，巧妙地处理了紧张的进度计划和经常发生冲突的要求。文字编辑Genevieve d'Entremont和编索引者Fred Brown灵巧地理顺了来自世界各地的作者的材料，让它们就像是一个人写出来的那样流畅。插图作者Robert Romano设法将我们提供的各种图像格式（包括一些手绘的草图）变成专业的图表。封面设计者Karen Montgomery设计了漂亮的、让人兴奋的封面，切合本书的内容。版式设计者David Futato设计了创新而实用的布局结构，将每章的说明与图书的设计集成在一起。最后，我们要感谢我们的家人和朋友在我们将精力集中在这本书上时给予的支持，这些时间本应属于他们。

第一部分

论架构

第 1 章

架构概述

John Klein
David Weiss

1.1 简介

建筑师、音乐家、作家、计算机设计师、网络设计师和软件开发者都在使用“架构”这个术语，其他人也用（你有没有听说过“食物架构”？），然而不同的用法其结果也不同。建筑与交响乐完全不同，但都有架构。而且，所有的架构师都在谈论他们工作中的美，以及因此而导致的结果。建筑师可能会说，一座建筑应该提供适合工作或生活的环境，而且它应该看起来很美。音乐家可能会说，音乐应该能演奏，包含能够辨明的主题，而且它应该听起来很美。软件架构师可能会说，系统应该对用户友好、响应及时、可维护、没有重大错误、易于安装、可靠，应该通过标准的方式与其他系统通信，而且也应该是美的。

这本书为你提供了一些美丽架构的详细例子，它们来自于各类计算机系统。相对来说，计算机是比较年轻的一个学科。因为年轻，所以不像建筑、音乐或写作等领域那样，有那么多的例子；也因为年轻，则需要更多的例子。我们希望这本书能满足这种需要。

在你开始研究这些例子之前，我们希望你考虑以下两个问题：1) 什么是架构？2) 美丽的架构都有哪些特性？你会在这一章中看到架构的不同定义，每个学科都有自己的定义，所以我们将首先探讨不同学科中的架构有何共同点，以及人们试图用架构解决哪些问题。具体来说，架构有助于确保系统能够满足其利益相关人的关注点，在构想、计划、构建和维护系统时，架构有助于处理复杂性。

然后我们将介绍架构的定义，展示如何将这个定义应用于软件架构，因为软件是本书后面大部分例子关注的核心。这个定义的关键在于，架构由一组结构组成，这些结构的设计目的是让架构师、构建者，以及其他利益相关人看到他们的关注点是如何得到满足的。

在本章末尾，我们将讨论美丽架构的特性，并引用一些例子。美的核心在于概念完整性——即一组抽象和规则，在整个系统中尽可能简单地应用它们。

在讨论中，我们将“架构”作为一个名词，它意味着一组工件，包括像蓝图和构建规范这样的文档。这些工件描述了要构建的对象，在这种描述中，该对象被视为一组结构。某些人也把“架构”作为一个动词，用来描述创建这些工件的过程，包括由此而导致的工作。然而，正如Jim Waldo和其他人曾指出的，没有什么过程可以保证你学了以后就能创造出好的系统架构，更不必说美的架构了（Waldo 2006），所以我们将更关注工件，而非过程。

> 架构：“建造的艺术或科学；特别是设计和建造人类使用的建筑时的艺术或实践，同时考虑到美学因素和实用因素。”
>
> ——《The Shorter Oxford English Dictionary》（小型牛津英语字典，第5版）

在所有学科中，架构都提供了一种方式来解决共同的问题：确保建筑、桥梁、乐曲、书籍、计算机、网络或系统在完成后具有某些属性或行为。换言之，架构既是所构建系统的计划，确保由此得到期望的特性，同时也是所构建系统的描述。维基百科上说：“根据这方面已知最早的著作，即Vitruvius的‘On Architecture’，好的建筑应该美观（Venustas）、坚固（Firmitas）、实用（Utilitas）；架构可以说是这三方面的一种平衡和配合，没有哪一个方面比其他方面更重要。”

> 我们谈到交响乐的“架构（architecture）”，反过来，又将架构（architecture）称为“凝固的音乐”。
>
> ——Deryck Cooke，《The Language of Music》（音乐的语言）

好的系统架构展示了架构完整性。也就是说，它来自于一组设计规则，这组规则有助于减少复杂性，并可以用于指导详细设计和系统验证。设计规则可能包含特定的抽象，这些抽象总是以同样的方式使用，诸如虚拟设备等。这些规则可能表现为一种模式，如管道和过滤器。在最理想的情况下，存在一些可以用于验证的规则，如“在设备失效时，所有某一类的虚拟设备都可以用任何其他同类的虚拟设备代替”，或“所有竞争同一资源的进程必须具有相同的调度优先级”。

当代的架构师可能会说，待构建的对象或系统必须具有以下特征：

- 具备客户要求的功能。
- 能够在要求的工期内安全地构建。

- 性能足够好。
- 可靠的。
- 可用的，并且使用时不会造成伤害。
- 安全的。
- 成本是可以接受的。
- 符合法规标准。
- 将超越前人及其竞争者。

> 我们将计算机系统的架构定义为一组最小的特征集，它们决定了哪些程序将运行，以及这些程序将得到什么结果。
>
> ——Gerrit Blaauw 和Frederick Brooks, 《Computer Architecture》（计算机体系结构）

我们从来没有看到过一个复杂系统能够很好地满足上述特征。架构是一种折中——决定改进其中一个特征常常会对其他特征产生负面影响。架构师必须确定怎样做是足够好的，方法就是发现特定系统的重要关注点，以及充分满足这些关注点的条件。

架构观点中的常见思想是结构，每种结构都由各种类型的组件及其关系构成：它们如何组合、相互调用、通信、同步，以及进行其他交互。组件可以是建筑中的支架横梁或内部腔室、交响乐中的旋律、故事中的章节或人物、计算机中的CPU和内存、通信栈中的层或连接到一个网络上的处理器、协作的顺序过程、对象、编译时的宏、构建时的脚本。每个学科都有自己的一套组件和组件间的相互关系。

> 从更大的范围来说，术语“架构”总是意味着“不变的深层次结构”。
>
> ——Stewart Brand, 《How Buildings Learn》

面对不断增长的系统复杂性，以及它们内部和相互之间的交互，由一组结构形成的架构提供了对付复杂性的主要手段，目的是确保得到的系统具备所要求的特征。结构为我们提供途径，将系统化解为一些交互的组件。

每种结构都是为了帮助架构师理解如何来满足特定的关注点，如可变性或性能。展示某些关注点得到满足时，可能会影响到其他方面的关注点，但架构师必须能够说明所有关注点都已得到满足。

> 网络架构：构成一个网络的通信设备、协议和传输链路，以及它们的组织方式。
>
> ——*http://www.wtcs.org/snmp4tpc/jton.htm*

1.1.1 建筑师的角色

在设计、构建和修复建筑时，我们指定关键的设计师为“建筑师（architects）”，并赋予他们广泛的职责。建筑师准备建筑最初的草图，展示外观和内部布局，与客户讨论这些草图，直至所有相关方都达成一致意见，认为展示的就是他们想要的。这些草图是抽象：它们关注建筑中某些方面的适当细节，而忽略其他的内容。

当客户和建筑师在这些抽象上达成一致意见之后，建筑师会准备或监督准备更为详细的图纸，以及相关的文字规格说明。这些图纸和规格说明描述了建筑的许多“实质性”细节，如管道、壁板材料、窗户玻璃和电线等。

在极少的情况下，建筑师简单地将详细规划交给建造者，建造者将根据规划完成项目。对更重要一些的项目，建筑师会继续参与，定期检查工作，并且可能会建议变更，或接受来自建造者和客户的变更建议。如果建筑师监督项目，仅当他确认项目充分符合了规划和规格说明的要求，项目才算完工。

我们请一名建筑师是为了确保：1）设计满足客户的需要，包括前面提到的那些特征；2）设计具有概念完整性，处处运用了相同的设计原则；3）设计满足法规和安全的要求。建筑师职责的一个重要方面是确保设计概念在实现时得到一致的体现。有时候，建筑师也充当建造者和客户之间的协调人。哪些决定需要由建筑师做出，哪些决定由其他人做出，人们对这个问题常有不同意见，但我们清楚，建筑师将做出重要决定，包括所有对结构的可用性、安全性和可维护性产生影响的那些决定。

音乐作曲与软件架构

虽然人们常用建筑架构设计来类比软件架构，但音乐作曲可能是更好的类比。建筑师创建的是相对静止的结构（该架构必须考虑到人员和服务在建筑内的移动，以及承重结构）的静态描述（蓝图或其他图纸）。在音乐作曲和软件设计中，作曲家（软件架构师）创建一段音乐的静态描述（架构描述和代码），这段音乐以后将演奏（执行）许多次。在音乐和软件中，设计都依靠许多组件的交互来得到期望的结果，结果依赖于演奏者、演奏环境，以及演奏者所做的诠释。

1.1.2 软件架构师的角色

软件开发项目需要一些人在软件构建时扮演架构师的角色，就像构建或修复建筑时传统的建筑师的角色一样。但是，对于软件系统来说，从来就弄不清楚哪些决定属于架构师的职责范围，哪些决定要留给实现者。定义架构师在软件项目中做什么，比建筑师的类

似定义更困难，原因有3个因素：缺少传统、产品无形性和系统复杂性。（参见Grinter[1999]，其中描述了软件架构师如何在一个大型软件开发组织中实现她的职责。）

具体来说：

- 建筑师可以回顾几千年的历史，看看过去的建筑师都做过些什么。他们可以参观并研究那些矗立了几百年的建筑，有时甚至有上千年历史的建筑，而它们仍在使用。在软件业，我们只有几十年的历史，并且我们的设计常常是不公开的。此外，建筑师拥有并利用标准来描述他们制作的图纸和规格说明，这让现在的建筑师能够从记录下来的架构历史中受益。
- 建筑是有形的产品，在建筑师制作的规划和工人修造的建筑之间存在着明显的区别。

架构复用

圣索菲亚大教堂（Hagia Sophia，上图），建造于公元6世纪，率先使用了所谓的“穹顶”结构来支撑巨大的圆形屋顶，它是拜占庭建筑之美的代表。在1100年之后，Christopher Wren使用了同样的设计来建造圣保罗大教堂的穹顶（St. Paul's Cathedral，下图），它成为伦敦的地标性建筑。这两座建筑在今天仍在使用。

在大的软件项目中，常常会有许多架构师。某些架构师相当专注于特定领域，如数据库和网络，他们一般作为团队的一部分，但目前我们假定只有唯一一位架构师。

1.1.3 软件架构的含义

如果认为“架构”是一个简单的实体，能够用一份文档或一张图纸来描述，那就错了。架构师必须做出许多设计决定。要想有用，这些决定必须用文档记录下来，这样就能够进行复审、讨论、修改和批准，然后作为后续决定和构建时的约束。对于软件系统，这些设计决定包括行为上的和结构上的。

外部行为描述展示了产品如何与它的用户、其他系统和外部设备进行交互，这应该表现为需求。结构描述展示了产品如何划分为多个部分，以及这些部分之间的关系。我们还需要内部行为描述，用于描述组件之间的交互接口。结构上的描述常常展示相同部分的一些不同视图，因为不可能把所有信息以有意义的方式组织到一张图纸或一份文档中。一个视图中的组件，可能是另一个视图中一个组件的一个部分。

软件架构常常表现为分层的层次结构，这种层次结构将几种不同的结构放在一张图中。20世纪70年代，Parnas指出“层次结构”这个术语已经被滥用，然后精确地定义了它，并给出了几个不同结构的例子，它们在设计不同系统时实现了不同的目的(Parnas 1974)。将架构的结构描述为一组视图（view），每个视图关注不同的部分，现在已成为了广泛接受的标准架构实践（Clements等 2003; IEEE 2000）。我们将使用“架构”这个词来代指一组有标注的图纸和功能描述，它说明了设计和构建一个系统时所使用的结构。在软件开发社区中，针对这样的图纸和描述，人们使用并建议了许多不同的形式。在Hoffman和Weiss（2000，第14章和第16章）的著作中可以看到一些例子。

> 一个程序或计算系统的软件架构是系统的一种结构或一组结构，它包含软件元素、这些元素的外部可见的属性，以及元素之间的关系。
>
> “外部可见”的属性是其他元素对该元素可以做出的假定，诸如它提供的服务、执行时的特征、错误处理、共享资源的使用等。
>
> ——Len Bass、Paul Clements和Rick Kazman
>
> 《Software Architecture in Practice, Second Edition》

1.1.4 架构与设计

架构是系统设计的一部分，它突出了某些细节，并通过抽象省略掉另一些细节。所以，架构是设计的一个子集。关注实现系统组件的开发者可能不会特别关心所有组件如何装配在一起，而是主要关注少数组件的设计和开发，包括他们必须遵守的架构约束和可以应用的规则。因此，开发者和架构师面对的是系统设计的不同方面。

如果说架构关注的是组件之间的关系和系统组件外部可见的属性，那么设计还要关注这些组件的内部结构。例如，如果一组组件包含了一些信息隐藏的模块，那么这些外部可见的属性就构成了这些组件的接口，内部的结构与模块内的数据结构和控制流一同考虑（Hoffman和Weiss 2000，第7章和第16章）。

1.2 创建软件架构

到目前为止，我们已经讨论了一般意义上的架构，并分析了软件架构与其他领域的架构之间有何相似与差异。接下来我们将注意力转到“如何”设计软件架构。当架构师创建软件系统的架构时，她应该关注什么？

软件架构师的首要关注点不是系统的功能。

这是正确的——软件架构师的首要关注点不是系统的功能。

例如，如果我们请你来设计一个“基于Web的应用”，你首先问我们页面布局和导航树，还是问下面这些问题：

- 谁提供应用主机托管？托管的环境有什么技术限制吗？
- 你想运行在Windows服务器上还是在LAMP栈上？
- 你想支持多少并发用户？
- 应用需要怎样的安全性？有需要保护的数据吗？应用将运行在公网上还是在私有的内部网上？
- 你能为这些答案排列优先级吗？例如，用户数是否比响应时间更重要？

根据我们对这些问题和一些其他问题的回答，你就可以开始画出系统架构的草图。我们还没有谈到应用的功能。

好吧，我们承认耍了点计谋，因为我们问的是“基于Web的应用”，这是一个大家熟悉的领域，所以你已经知道了哪些决定会对你的架构产生最大的影响。类似地，如果我们问的是一个电信系统或一个航空电子控制系统，在这些领域有经验的架构师将考虑到一些功能需求。但是，你仍然可以不必过多担心功能就开始设计架构。你关注的是需要满足的品质。

品质关注点指明了功能必须以何种方式交付，才能被系统的利益相关人所接受，系统的结果包含这些人的既定利益。利益相关人有一些关注点，架构师必须重视。稍后，我们将讨论为了确保系统具有要求的品质，通常会提出的一些关注点。正如我们前面所说的，架构师的一项职责是确保系统设计能满足客户的需要，我们将利用品质关注点来帮助我们理解这些需要。

这个例子突出了成功架构师的两项关键实践：让利益相关人参与以及同时关注功能和品质。作为一名架构师，你首先问我们想从系统中得到什么，有怎样的优先级。在实际项目中，你会找出其他的利益相关人。典型的利益相关人和他们的关注点包括：

- 投资人，他们想知道项目是否能够在给定的资源和进度约束下完成。
- 架构师、开发人员和测试人员，他们首先考虑的是最初的构建和以后的维护与演进。
- 项目经理，他们需要组织团队，制定迭代计划。
- 市场人员，他们想通过品质特点实现与竞争者的差异化。
- 用户，包括最终用户、系统管理员，以及安装、部署、准备、配置人员。
- 技术支持人员，他们关注帮助平台电话呼入的数目和复杂性。

每个系统都有自己的品质关注点。有些关注点可能定义得很好，如性能、安全、可伸缩性等。但是，另一些同样重要的关注点却可能没有详细规定，如可变性、可维护性和可用性等。利益相关人希望把功能放到软件上，而不是放到硬件上，这主要是为了很容易、很快速地修改，然后通常在品质关注方面又对可变性轻描淡写。这很奇怪，不是吗？哪些改变能够迅速、容易地实现，哪些改变需要花时间并且很难实现，架构决定将对此产生重要影响。所以，架构师难道不应该在理解功能需求的同时，也理解利益相关人在“可变性”这样的品质方面的期望吗？

当架构师理解了利益相关人的品质关注点之后，接下来该做些什么？考虑折中。例如，对信息加密将加强安全性，但会损失性能。利用配置文件将增加可变性，但会降低可用性，除非我们能够验证配置是有效的。我们是否应该对这些文件使用标准的表示方式，如XML，还是使用自己发明的格式？创建系统的架构将涉及许多这样的艰难折中。

架构师的第一项任务，就是与利益相关人协作，理解这些品质关注点和约束，并为它们排列优先级。为什么不从功能需求开始？因为通常有许多种可能的系统分解方式。例如，从数据模型开始可能得到一种架构，而从业务处理模型开始则可能得到不同的架构。在极端的情况下，系统没有分解，被开发成单一的软件。这可能会满足所有的功能需求，但可能不会满足品质需求，如可变性、可维护性、可伸缩性等。架构师通常必须进行架构层面的系统重构，例如为了满足伸缩性或性能的要求，将单机部署迁移到分布式部署，从单线程转向多线程，或者将硬编码的参数移到外部配置文件中，因为原来从不改变的参数现在需要修改了。

尽管有许多架构都能满足功能需求，其中却只有一少部分能够满足品质需求。让我们回到Web应用的例子。请考虑提供Web页面的诸多方式——Apache和静态页面、CGI、Servlet、JSP、JSF、PHP、Ruby on Rails、ASP.NET等。选择其中的一种技术是一种架构决定，它将对你满足特定品质需求的能力产生重要影响。例如，像Ruby on Rails这样

的方式可能提供快速推向市场的好处，但可能更难维护，因为Ruby语言和Rails框架都在不断地快速发展。也许我们的应用是基于Web的电话，我们需要让电话“响铃”。如果你为了满足性能的要求，需要从服务器向Web页面发出真正异步的事件，那么基于Servlet的架构可能更容易测试和修改。

在真实的项目中，满足利益相关人的关注点需要做出更多的决定，而不仅是选择一个Web框架。你是否真的需要一个“架构”，并需要一名“架构师”来做出这些决定？谁将做出这些决定？是编程人员吗？他们可能会做出许多无意识的、隐含的决定。还是由架构师来做出这些决定？他们全面了解整个系统、利益相关人和系统的演进，然后做出明确的决定。不论哪种方式，你会有一个架构。它是否应该明确地形成并记入文档？或者它应该是隐式的，需要通过阅读代码才能发现？

当然，这种选择通常不是这么死板。但是，随着系统的规模、复杂度和开发人员数目的增长，这些早期决定以及它们的记录方式将产生越来越大的影响。

我们希望你现在已经理解，如果你的系统要满足其品质要求，架构决定是很重要的，你需要注意架构，有意识地做出这些决定，而不只是“让架构自动出现”。

如果系统非常大，情况会怎样？我们之所以运用“分而治之”这样的架构原则，一个原因就是为了降低复杂性，让工作能够并发进行。这让我们能够创建越来越大的系统。架构本身是否能够分解为多个部分，这些部分是否能由不同的人并行开发？考虑到计算机的架构，Gerrit Blaauw和Fred Brooks断定：

> ……如果，在采取了所有让任务能够由单人处理的方法之后，架构任务仍然巨大而复杂，不能由一人来完成，那么产品肯定是太复杂了，以致不实用且不应构建。换言之，单个用户必须能够理解计算机的架构。如果计划的架构不能由一个人设计，那它也不能被一个人理解。(1997)

你是否需要理解架构的所有方面，才能使用它？架构会分离关注点，所以在大多数情况下，利用架构来构建或维护系统的开发人员或测试人员，不需要一下面对全部的架构，而是只要面对必要的部分，就能完成指定的功能。这让我们能够创建超出个人可以理解的、更大的系统。但是，在我们完全忽略IBM System/360（最长寿的计算机架构之一）创造者的建议之前，让我们先来看看他们为什么这样说。

Fred Brooks说，概念完整性是架构最重要的特征：“最好是让系统……反映一组设计思想，而不是让系统包含许多好的思想，而这些思想却彼此独立而不协调”(1995)。正是这种概念完整性，让开发者在知道了系统的一部分之后，能够迅速理解系统的另一部分。概念完整性来自于处理问题的一致性，如分解的判据、设计模式的应用和数据模式。这让开发者运用在系统中的一部分工作的经验，来开发和维护系统的其他部分。同样的规

则应用于整个系统各处。当我们转向“众系统之系统”时，在集成了这些系统的架构中也必须保持概念完整性。例如，可以选择发布/订阅消息总线这样的架构风格，然后将这种风格统一地应用于“众系统之系统”的系统集成中。

架构团队的挑战在于，在创建架构时保持同一种思考方式和同一种哲学。让团队保持尽可能小，让他们在充分沟通、高度协作的环境工作，让一两个“首席架构师”担任仁慈的独裁者，最终做出所有决定。这种架构模式常见于成功的团队，不论是公司开发还是开源开发，由此而得到的概念完整性是美丽架构的一种特性。

好的架构师通常来自于更好的架构师提供的现场指导（Waldo 2006）。原因之一可能是有一些关注点几乎在所有项目中都会出现。我们已经提到过一些，但这里有一份更完整的清单。每个关注点都以问题的方式表述，架构师在项目过程中可能需要考虑它。当然，具体系统会有其他关键的关注点。

功能性（Functionality）

产品向它的用户提供哪些功能？

可变性（Changeability）

软件将来可能需要哪些改变？哪些改变不太可能发生，不需要特别容易进行这些改变？

性能（Performance）

产品将达到怎样的性能？

容量（Capacity）

多少用户将并发使用该系统？该系统将为用户保存多少数据？

生态系统（Ecosystem）

在部署的生态环境中，该系统将与其他系统进行哪些交互？

模块化（Modularity）

如何将编写软件的任务分解为工作指派（模块），特别是这些模块可以独立地开发，并能够准确而容易地满足彼此的需要？

可构建性（Buildability）

如何将软件构建为一组组件，并能够独立实现和验证这些组件？哪些组件应该复用其他的产品，哪些应该从外部供应商处获得？

产品化（Producibility）

如果产品将以几种变体的形式存在，如何开发一个产品线，并利用这些变体的共性？产品线中的产品以怎样的步骤开发（Weiss和Lai 1999）？在创建一条软件产品线时，要进行哪些投资？开发产品线中不同变体的选择，预期会得到怎样的回报？

特别是，是否可能先开发最小的有用产品，然后再添加（扩展）组件，在不改变以前编写的代码的情况下，开发产品线的其他成员？

安全性（Security）

产品是否需要用户认证，或者必须限制对数据的访问？数据的安全性如何得到保证？如何抵挡“拒绝服务”攻击或其他攻击？

最后，一个好的架构师会认识到，架构会影响组织机构。Conway指出，系统的结构会反映构建它的组织机构的结构（1968）。架构师可能会认识到，Conway法则可以反过来应用。换言之，一个好的架构可能对组织机构产生影响，让组织机构发生改变，从而更有效地从该架构构建出系统。

1.3 架构结构

那么，好的架构师如何来处理这些关注点？我们曾经提到过，需要将系统组织成一些结构，每种结构都定义了特定类型的组件之间的具体关系。架构师的主要关注点就是对系统进行组织，让每种结构有助于解答一个关注点所定义的问题。关键的结构决定将产品划分为组件，并定义了这些组件之间的关系（Bass、Clements和Kazman 2003; Booch、 Rumbaugh和Jacobson 1999; IEEE 2000; Garlan和Perry 1995）。对于任何产品，都有许多结构需要设计。每种结构都必须单独设计，这样它就表现为一个独立的关注点。在接下来的几节中我们会讨论一些结构，你可以利用它们来考虑前面列表中的关注点。例如，“信息隐藏结构”展示了如何将系统组织成一些工作指派。这种结构也可以用作改变的路线图，展示了建议的改变，以及哪些模块支持这些改变。针对每种结构，我们描述了一些组件及其之间的关系，正是这些组件和关系确定了这种结构。对照前面的列表，我们认为下面的结构是最重要的。

1.3.1 信息隐藏结构

组件与关系：主要组件是一些“信息隐藏模块”，每个模块都是针对一组开发人员的工作指派，每个模块都包含了一种设计决定。如果一项决定可以改变，同时又不影响任何其他模块，我们就说这项设计决定是一个模块的秘密（Hoffman和Weiss 2000，第7章和第16章）。模块间最基本的关系是“整体-部分”关系。如果“信息隐藏模块A”的秘密是“信息隐藏模块B”的秘密的一部分，那么A就是B的一部分。请注意，必须能够在改变A的秘密的同时，不改变B的其他部分。否则，根据我们的定义，A就不是B的一个子模块。例如，许多架构都有一些虚拟设备模块，它们的秘密是如何与特定的物理设备通

信。如果虚拟设备分成不同类型，那么每种类型可能构成该虚拟设备模块的一个子模块，其中每种虚拟设备类型的秘密将是如何与这种类型的设备进行通信。

每个模块都是一份工作指派，包含了一组要写的程序。根据不同的语言、平台、环境，“程序”可以是能在计算机上执行的方法、过程、函数、子程序、脚本、宏或其他指令序列。第二种信息隐藏模块结构是基于程序和模块之间的“包含”关系。如果模块M的一部分工作指派是要编写程序P，那么M就包含P。请注意，每个程序都包含在一个模块中，因为每个程序必然是某些开发人员的工作指派的一部分。

这些程序中的一些可以通过模块的接口来访问，而另一些则是内部的。模块也可能通过接口发生关系。A模块的接口是一组假定，这些假定包括该模块之外的程序可以对该模块做出的假定，也包括该模块中的程序对其他模块的程序和数据结构所做的假定。如果改变B的接口就要求A也发生改变，那么我们就说A“依赖”B的接口。

“整体–部分”结构是层次状的。在这个层次结构的叶节点上的模块不包含可识别的子模块。“包含”结构也是层次状的，因为每个程序都只包含在一个模块之中。“依赖”关系不一定是层次状的，因为两个模块可能互相依赖，要么是直接互相依赖，要么是通过一个较长的“依赖”关系形成的环。请注意，“依赖”不应该与后面小节中定义的“使用”混淆起来。

信息隐藏结构是面向对象设计方法的基础。如果一个信息隐藏模块设计为一个类，这个类的公有方法就属于该模块的接口。

满足的关注点：信息隐藏结构的设计应该能满足可变性、模块化和可构建性的要求。

1.3.2 使用结构

组件与关系：根据前面我们的定义，信息隐藏模块包含一个或多个程序（在上一小节中定义）。当且仅当两个程序共享一个秘密时，它们才属于同一个模块。“使用结构”（Uses Structure）的组件是一些可以单独调用的程序。请注意，程序可以相互调用，或被硬件调用（例如，被一个中断例程调用），调用也可能来自于不同命名空间的程序，如操作系统例程或远程过程。而且，调用发生的时间可以是任何时候，从编译时到运行时。

只有在相同绑定时间操作的程序之间，我们才考虑形成一种使用结构。首先只考虑运行时操作的程序可能最容易。以后，我们也可以考虑那些编译时或载入时操作的程序之间的使用关系。

如果程序B必须存在并满足其规格说明，程序A才能满足其规格说明，我们就说A使用了B。换言之，B必须存在且操作正常，A才能操作正常。使用关系有时候也称为“要求存在正确的版本”。进一步的解释和例子，参见（Hoffman和Weiss 2000）的第14章。

使用结构确定了我们可以构建并测试怎样的工作子集。在软件系统的使用结构中，期望的属性是它定义了一种层次结构，这意味着其中不出现环。如果在使用关系中出现环，那么环中所有程序都必须存在且正常工作，才能让其他的程序正常工作。由于也许不能够创建完全没有环的使用关系，架构师可能将使用环中的所有程序作为单一的程序，以这种方法来创建子集。子集必须要么包含全部程序，要么都不包含。

如果在使用关系中没有环，软件采用的就是一种层次结构。在最底层，即第0层，是所有不使用其他程序的程序。第n层包含了所有的程序，它们使用了第$n-1$层或以下层的程序。这些层常常描绘为一系列的层次，每个层次表示了使用关系中的一个或几个层。在使用结构中对相邻的层分组，有助于简化表示，并允许在关系中出现小环的情况。进行这种分组有一个指导原则，即一个层次中的程序应该比它上一个层次中的程序执行速度快9倍，执行频率高9倍（Courtois 1977）。

具有层次使用结构的系统可以同时构造一层或几层。这些层次有时候称为“抽象层”，但这是一种错误的名称。因为这些组件是独立的程序，而不是完整的模块，它们不一定抽象（隐藏）了什么东西。

通常大型的软件系统包含太多的程序，这让程序间使用关系的描述不太容易理解。在这种情况下，使用关系可以用于程序的组合，如模块、类或包。这样的组合描述丧失了重要的信息，但有助于展示“全局”。例如，你有时候可以在信息隐藏模块之间建立使用关系，但是除非一个模块中所有的程序都属于实际使用层次的同一层，否则就会丧失重要的信息。

在某些项目中，系统的使用关系开始并没有完全确定，要到系统实现时才能确定，因为开发者会在实现过程中决定他们使用哪些程序。但是，系统的架构师可能在设计时创建一种“允许使用”关系，约束开发者的选择。今后，我们不会区分“使用”和“允许使用”。

定义良好的使用结构将创建系统的适当子集，可以用于驱动迭代式或增量式的开发循环。

满足的关注点：产品化和生态系统。

1.3.3 进程结构

组件与关系：信息隐藏模块结构和使用结构是静态的结构，存在于设计时和编码时。我们现在转向运行时结构。参与进程结构的组件是进程。进程是运行时的事件序列，由程序控制（Dijkstra 1968）。每个程序都作为一个或多个进程的一部分执行。一个进程中的事件序列的执行独立于另一进程中的事件序列，除非这两个进程彼此同步，例如一个进程等待来自另一个进程的信号或消息。进程由支持系统分配资源，包括内存和处理器时间。系统可能包含固定数量的进程，也可能在运行时创建和销毁进程。请注意，在

Linux和Windows操作系统中实现的线程也符合这个进程定义。进程是几种不同关系中的组件。下面是一些例子。

进程提供工作

一个进程可能会创建工作，该项工作必须由其他进程完成。这种结构在确定系统是否死锁时是很重要的。

满足的关注点：性能和容量。

进程取得资源

在动态分配资源的系统中，一个进程可能控制由另一个进程使用的资源，后者必须请求并归还这些资源。因为发起请求的进程可能从几个控制器那里请求资源，所以每项资源可能都有一个不同的控制进程。

满足的关注点：性能和容量。

进程共享资源

两个进程可能共享资源，如打印机、内存或端口等。如果两个进程共享一项资源，就需要通过同步来防止使用冲突。每一种资源可能有不同的关系。

满足的关注点：性能和容量。

进程包含在模块中

每个进程由一个程序控制，正如前面提到的，每个程序包含在一个模块之中。因此，我们可以认为进程包含在模块之中。

满足的关注点：性能和容量。

1.3.4 访问结构

系统中的数据可能划分成具有属性的段，如果程序对段中的任何数据拥有访问权，就对该段中的所有数据拥有了访问权。请注意，为了简化描述，我们应该让段的规模最大化，具体做法是添加一个条件，即如果两个段被同一组程序访问，这两个段就应该合并。数据访问结构包含两种类型的组件：程序和段。这种关系被命名“有权访问”，它是程序和数据段之间的关系。如果这种结构让程序访问的权限最小化，并且严格执行，我们就认为系统更安全。

满足的关注点：安全性。

1.3.5 结构小结

表1-1总结了前面的软件结构，包括它们的定义和它们满足的关注点。

表1-1：结构小结

结构	组件	关系	关注点
信息隐藏	信息隐藏模块	整体—部分 包含	可变性 模块化 可构建性
使用	程序	使用	产品化 生态系统
进程	进程（任务、线程）	提供工作 取得资源 共享资源 包含在模块中 ……	性能 可变性 容量
数据访问	程序和数据段	有权访问	安全性 生态系统

1.4 好的架构

我们曾提到，架构师玩的是折中的游戏。对于一组给定的功能需求和品质需求，没有唯一的正确架构和唯一的“正确答案”。我们从经验中得知，应该对架构进行评估，确定它是否满足其需求，然后再投入资金来构建、测试和部署这个系统。评估试图回答前面小节中讨论的一个或多个一般关注点问题，或回答特定系统的具体关注问题。

架构评估有两种常见的方式（Clements、Kazman和Klein 2002）。第一种评估方式是确定架构的属性，通常通过建模或模拟系统的一个或多个方面。例如，通过性能建模来评估吞吐量和伸缩性，通过失效树模型来评估可靠性和可访问性。其他类型的模型包括复杂性和耦合指标，用于评估可变性和可维护性。

第二种评估方式，也是最广泛使用的方式，就是通过对架构师提出质询来评估该架构。有许多结构化的质询方法。例如，贝尔实验室提出的软件架构复查委员会（Software Architecture Review Board，SARB）过程利用了组织机构之内的专家，以及他们在电信和相关应用中的深厚领域经验（Maranzano等 2005）。

质询方法的另一种变体是架构折中分析方法（Architecture Trade-off Analysis Method，ATAM）（Clements、Kazman和Klein 2002），它寻找架构不能满足品质关注点的风险。ATAM使用了场景分析，每种场景都描述了特定的利益相关人对系统的品质关注点。架构师然后解释该架构如何支持每一种场景。

主动复审是另一种质询方法，它改变了复审过程的开始方式，要求架构师向复审者提供架构师认为重要而需要回答的问题（Hoffman和Weiss 2000，第17章）。然后，复查者利

用已有的架构文档和描述来回答这些问题。最后，在网络上查找“software architecture review checklist（软件架构复审检查清单）”，可以得到几十份检查清单，其中某些清单非常通用，另一些则是针对具体的应用领域或技术框架。

1.5 美丽的架构

所有前面的方法都有助于我们判断一个架构是否“足够好”——也就是说，是否有可能指导开发者和测试者构建一个系统，并满足系统的利益相关人的功能和质量关注点。在我们每天使用的系统中存在着许多好的架构。

但是，超越足够好的架构是怎样的呢？如果有一个“软件架构名人堂”，那会怎样？哪些架构会陈列在这个艺术馆的墙上？这个想法可能没有你想象的那么遥远——在软件产品线领域，这样的“名人堂”的确存在。（注1）进入“软件产品线名人堂”的条件包括获得商业上的成功、影响其他产品线的架构（其他产品线可能“借用、复制、窃取”这个架构）、拥有足够的文档从而让其他人“不必通过道听途说”就能够理解该架构。

我们将为更一般的“架构名人堂”或“美丽架构艺术馆”的候选者设立怎样的条件呢？

首先我们应该认识到，这是一个软件系统的艺术馆，而不是其他艺术馆，我们的系统构建的目的是为了使用。所以，我们也许从一开始就应该关注该架构的实用性：它应该每天被许多人使用。

但是，在使用架构之前，它必须先构建，所以我们应该关注该架构的可构建性。我们会寻找那些具有定义良好的使用结构的架构，它们支持增量式构建，这样，通过每次构建迭代我们都能得到一个有用的、可测试的系统。我们也会寻找那些具有定义良好的模块接口、本来就很好测试的架构，这样，构建的过程就是透明的、可见的。

接来下，我们想寻找那些展示了持久性的架构——也就是说，那些经过了时间考验的架构。我们生活在一个技术环境以从未有过的加速度变化的年代。美丽的架构应该预期到变更的需要，允许期望的修改能够容易而有效地进行。我们想寻找那些避免了“衰老地平线”（Klein 2005）的架构，超过了这条“衰老地平线”，维护将变得代价极大，以至于不可能进行。

最后，我们还想寻找这样一些架构，它们的特征让使用、构建、测试这些架构的开发人员和测试人员，以及由它而形成的系统的用户感到由衷的高兴。为什么开发人员会高兴？因为它让他们的工作变得容易，而且更有可能得到一个高品质的系统。为什么测试人员会高兴？作为测试过程的一部分，他们必须尝试模拟用户的行为。如果他们高兴，

注1： 参见 *http://www.sei.cmu.edu/productlines/plp_hof.html.*

可能用户也会感到高兴。如果厨师对他的烹调的菜品感到不高兴，那么品尝这些菜品的顾客也可能会感到不高兴。

不同的系统和应用领域为这些架构提供了许多机会，展示它们特有的令人高兴的特征，但概念完整性是一项跨越所有领域的特征，并且总是让人感到高兴。一致的架构学习起来更容易、更快，当知道了一点之后，你就可以开始预测其余的部分。不需要记住并处理特殊的情况，代码更干净，测试集更小。一致的架构不会为做同一件事情提供两种（或更多）方法，不会让用户浪费时间进行选择。正如Ludwig Mies van der Rohe所说的，好的设计，“少即是多”。爱因斯坦可能会说，美丽的架构就是尽可能简单，但不要过于简单。

有了这些判别条件，我们推荐第一批进入“美丽架构艺术馆”的候选者。

第一个是A-7E舰载飞行处理器（Onboard Flight Processor，OFP）的架构，它由海军研究实验室（Naval Research Laboratory，NRL）在20世纪70年代后期开发，在（Bass、 Clements和Kazman 2003)有介绍。尽管这个系统从未实现量产，但它满足了所有其他的判别条件。这个架构对软件架构的实践曾经产生了巨大的影响，它展示在真实世界的系统中，将设计时的信息隐藏模块和使用结构与运行时的进程结构分离。因为美国政府资助并开发了这个架构，所以所有项目文档都提供给了公共领域。（注2）这个架构具有定义良好的使用结构，促进了系统的增量式构建。最后，信息隐藏模块结构为分解系统提供了清晰一致的准则，实现了很强的概念完整性。作为嵌入式系统软件架构的榜样，A-7E OFP当然属于我们的艺术馆。

我们想放入艺术馆的另一个架构是朗讯5ESS电话交换机的软件架构（Carney等 1985）。5ESS取得了全球范围的商业成功，为世界各国的网络提供了核心电话网络交换。它成为性能和可靠性的标准，每个单元每小时能处理超过100万次的连接，平均每年非计划宕机时间少于10秒钟（Alcatel-Lucent 1999）。该架构的一些统一概念，如管理电话连接的“半通话模型”，已经成为电话和网络协议领域的标准模式（Hanmer 2001）。除了保持必须处理的通话类型的数目为2*n*（其中*n*是通话协议的数目）之外，半通话模式还在操作系统的进程概念和电话的通话类型概念之间建立起了联系，从而提供了简单的设计原则，引入了漂亮的架构一致性。在过去的25年中，开发团队涉及多达3000个人，他们发展并增强该系统。基于它的商业成功、持久性和影响，5ESS架构是我们艺术馆的一件好藏品。

还有一个我们想放入美丽架构艺术馆的系统，它就是万维网（World Wide Web，WWW）的架构。它由Tim Berners-Lee在CERN创建，在（Bass、Clements和Kazman 2003）中有介绍。万维网当然已经取得了商业上的成功，它转变了人们使用因特网的方式。即使创建了新的应用、引入了新的功能，它的架构仍然保持不变。该架构的整体简单性促成

注2： 参见CHoffman和Weiss2000）的第6章、第15章和第16章，或在NRL Digital Archives (*http://torpedo.nrl.navy.mil/tu/ps*)中查找“A-7E”。

了它的概念完整性，但有一些决定导致了该架构的完整性保持不变，如客户端和服务器端使用同一个库，创建分层架构以实现分离关注点等。核心万维网架构的持久性和它对新扩展、新功能持续支持的能力，使它当之无愧地进入了我们的艺术馆。

什么是建筑师？

夏天很热的一个日子里，一个外乡人沿着一条路在行走。他走着走着，来到一个人跟前，此人正在路边敲碎石头。

“你在做什么？”他问那个人。

那个人抬头看着他，“我在敲碎石头。你以为我看起来像在干什么？现在不要妨碍我，让我继续干活。”

这个外乡人继续沿着路走，不久他遇到了第二个在大太阳下敲碎石头的人。这个人正努力工作，汗滴如雨。

“你在做什么？”外乡人问道。

这个人抬头看他，露出微笑。

“我在为谋生而工作，”他说，“但这个工作太辛苦了。也许你能给我一份更好的工作？”

外乡人摇了摇头，继续前行。没多久，他遇到了第三个敲碎石头的人。太阳正是最炙热的时候，这个人非常卖力，汗流如注。

“你在做什么？”外乡人问道。

这个人停了一下，喝了一口水，微笑着抬起他的手，指向天空。

“我在建一座大教堂。”他喘着气说。

外乡人看了他一会儿，说：“我们正打算开一家新公司。你来做我们的总建筑师怎么样？”

我们的最后一个例子是UNIX系统，它展示了概念完整性，使用广泛，拥有巨大的影响力。管道和过滤器的设计是讨人喜欢的抽象，允许我们快速构建新的应用。

在描述架构、架构师的角色和创建架构时的考虑等方面，我们已经谈了很多，我们也简单介绍了一些美丽架构的例子。接下来我们邀请你阅读后续章节中详细的例子，这些例子来自于那些技艺精湛的架构师，本书介绍了他们创建并使用过的那些美丽架构。

致谢

David Parnas在几篇论文中定义了我们描述的许多结构，其中包括他的“术语滥用”论文（Parnas 1974）。Jon Bentley为这本书提供了创作灵感，他和Deborah Hill、Mark Klein对早期的草稿提出了许多有价值的建议。

参考文献

Alcatel-Lucent. 1999. "Lucent's record-breaking reliability continues to lead the industry according to latest quality report." *Alcatel-Lucent Press Releases. June 2. http://www.alcatel-lucent. com/ wps/ portal/ News Releases/ DetailLucent? LMSG_CABINET= Docs_and_ Resource_ Ctr&LMSG _CONTENT_FILE=News_Releases_LU_1999/LU_News_Article_007318.xml* (accessed May 15, 2008).

Bass, L., P. Clements, and R. Kazman. 2003. *Software Architecture in Practice*, Second Edition. Boston, MA: Addison-Wesley.

Blaauw, G., and F. Brooks. 1997. *Computer Architecture: Concepts and Evolution*. Boston, MA: Addison-Wesley.

Booch, G., J. Rumbaugh, and I. Jacobson. 1999. *The UML Modeling Language User Guide*. Boston, MA: Addison-Wesley.

Brooks, F. 1995. *The Mythical Man-Month*. Boston, MA: Addison-Wesley.

Carney, D. L., et al. 1985. "The 5ESS switching system: Architectural overview." *AT&T Technical Journal*, vol. 64, no. 6, p. 1339.

Clements, P., et al. 2003. *Documenting Software Architectures: Views and* Beyond. Boston, MA: Addison-Wesley.

Clements, P., R. Kazman, and M. Klein. 2002. *Evaluating Software Architectures*. Boston: Addison-Wesley.

Conway, M. 1968. "How do committees invent." *Datamation*, vol. 14, no. 4.

Courtois, P. J. 1977. *Decomposability: Queuing and Computer Systems*. New York, NY: Academic Press.

Dijkstra, E. W. 1968. "Co-operating sequential processes." *Programming Languages*. Ed. F. Genuys. New York, NY: Academic Press.

Garlan, D., and D. Perry. 1995. "Introduction to the special issue on software architecture." *IEEE Transactions on Software Engineering*, vol. 21, no. 4.

Grinter, R. E. 1999. "Systems architecture: Product designing and social engineering." *Proceedings of ACM Conference on Work Activities Coordination and Collaboration (WACC '99)*. 11-18. San Francisco, CA.

Hanmer, R. 2001. "Call processing." *Pattern Languages of Programming (PLoP)*. Monticello, IL. *http://hillside.net/plop/plop2001/accepted_submissions/PLoP2001/rhanmer0/PLoP2001_rhanmer0_1.pdf*.

Hoffman, D., and D. Weiss. 2000. *Software Fundamentals: Collected Papers by David L. Parnas*. Boston, MA: Addison-Wesley.

IEEE. 2000. "Recommended practice for architectural description of software intensive systems." Std 1471. Los Alamitos, CA: IEEE.

Klein, John. 2005. "How does the architect's role change as the software ages?" *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Washington, DC: IEEE Computer Society.

Maranzano, J., et al. 2005. "Architecture reviews: Practice and experience." *IEEE Software*, March/April 2005.

Parnas, David L. 1974. "On a buzzword: Hierarchical structure." *Proceedings of IFIP Congress. Amsterdam*, North Holland. [Reprinted as Chapter 9 in Hoffman and Weiss (2000).]

Waldo, J. 2006. "On system design." *OOPLSA '06*. October 22-26. Portland, OR.

Weiss, D., and C. T. R. Lai. 1999. *Software Product Line Engineering*. Boston, MA: Addison-Wesley.

第 2 章

两个系统的故事：现代软件神话

Pete Goodliffe

架构是一种很浪费空间的艺术。

——Philip Johnson

软件系统就像一座由建筑和后面的路构成的城市——由公路和旅馆构成的错综复杂的网络。在繁忙的城市里发生着许多事情，控制流不断产生，它们的生命在城市中交织在一起，然后死亡。丰富的数据积聚在一起、存储起来，然后销毁。有各式各样的建筑：有的高大美丽，有的低矮实用，还有的坍塌破损。随着数据围绕着它们流动，形成了交通堵塞和追尾、高峰时段和道路维护。软件之城的品质直接与其中包含多少城市规划有关。

某些软件系统很幸运，创建时由有经验的架构师进行了深思熟虑的设计，在构建时体现出了优雅和平衡，有很好的地图，便于导航。另一些软件系统就没有这么幸运，基本上是一些通过偶然聚集的代码渐渐形成的，交通基础设施不足，建筑单调而平凡，置身于其中时会完全迷失，找不着路。

你的代码愿意待在怎样的“城市”中？你愿意构建哪一种城市？

在本章中，我将讲述这样两个软件城市的故事。这是真实的故事，就像所有好的故事一样，这个故事最终是有教育意义的。人们说经验是伟大的老师，但最好是别人的经验，如果你能从这些项目的错误和成功之中学习，你（和你的软件）可能会避免很多的痛苦。

本章中的这两个系统特别有趣，因为它们有很大不同，尽管从表面上看非常相似：

- 它们具有相似的规模（大约500 000行代码）。
- 它们都是“嵌入式”消费音频设备。
- 每种软件的生态系统都是成熟的，已经经历了许多的产品版本。
- 两种解决方案都是基于Linux的。
- 编码采用C++语言。
- 它们都是由“有经验的”程序员开发的（在某些情况下，他们本应知道得更多）。
- 程序员本身就是架构师。

在这个故事中，人名都已改变，目的是保护那些无辜的人（和有罪的人）。

2.1 混乱大都市

你们修筑、修筑，预备道路，将绊脚石从我百姓的路中除掉。

——《以赛亚书》第57章14节

我们要看的第一个软件系统名为“混乱大都市”。它是我喜欢回顾的一个系统——既不是因为它很好，也不是因为它让参与开发的人感到舒服，而是因为当我第一次参与它的开发时，它教给了我有价值的软件开发经验。

我第一次接触“混乱大都市”，是在我加入了创建它的公司时。初看上去这是一份有前途的工作。我将加入一个团队，参与基于Linux的、“现代”的C++代码集开发，已有的代码集已经开发几年了。如果你像我一样拥有特殊的技术崇拜，就会觉得很兴奋。

工作起初并不顺利，但是你不能指望在加入一个新团队、面对新的代码集时会觉得很轻松。然而，日复一日（周复一周），情况却没有任何好转。这些代码要花极长的时间来学习，没有显而易见的进入系统中的路径。这是个警告信号。从微观的层面来说，也就是从每行程序、每个方法、每个组件来看，代码都是混乱而粗糙地垒在一起的。不存在一致性、不存在风格、也没有统一的概念能够将不同的部分组织在一起。这是另一个警告信号。系统中的控制流让人觉得不舒服，无法预测。这又是一个警告信号。系统中有太多的“坏味道”（Fowler 1999），整个代码集散发着腐烂的气味，是在大热天里散发着刺激性气体的一个垃圾堆。这是一个清晰的警告信号。数据很少放在使用它的地方。经常引入额外的巴罗克式缓存层，目的是试图让数据停留在更方便的地方。这又是一个警告信号。

当我试图在大脑中建立“大都市”的全图时，没有人能解释它的结构；没有人知道它的所有层、它的藤蔓，以及那些黑暗、隔离的角落。实际上，没有人知道它究竟有多少部分是真正能工作的（它实际上靠的是运气和英雄式的维护程序员）。人们知道他们面对

的那一小部分区域，但没人了解整个系统。很自然，没有任何文档。这也是一个警告信号。我需要的是一份地图。

这是一个悲伤的故事，我曾是其中的一部分："大都市"是城市规划的恶梦。在你开始整治混乱之前，先要理解混乱，所以我们花了很大的精力和毅力，得到了一份"架构图"。我们标出了每一条公路、每一条主干道、每一条很少人了解的小路、所有灯光昏暗的辅路，并将它们画在一张主图上。我们第一次看到了这个软件的样子，并不令人赏心悦目。它是一些混乱的区块和线条。为了让它更好理解一些，我们用颜色标出了控制路径，突出了它们的类型。然后我们后退一步看着它。

它令人吃惊。它令人目眩神迷。它就像一只喝醉了的蜘蛛，跌进了一些海报颜料罐里，然后在一张纸上织成了一张彩色的网。它看起来就像图2-1那样（这是一个简化后的版本，细节已经修改了，为了保护那些有罪的人）。事情变得很清楚了。我们画出了伦敦地铁图。它甚至有环线。

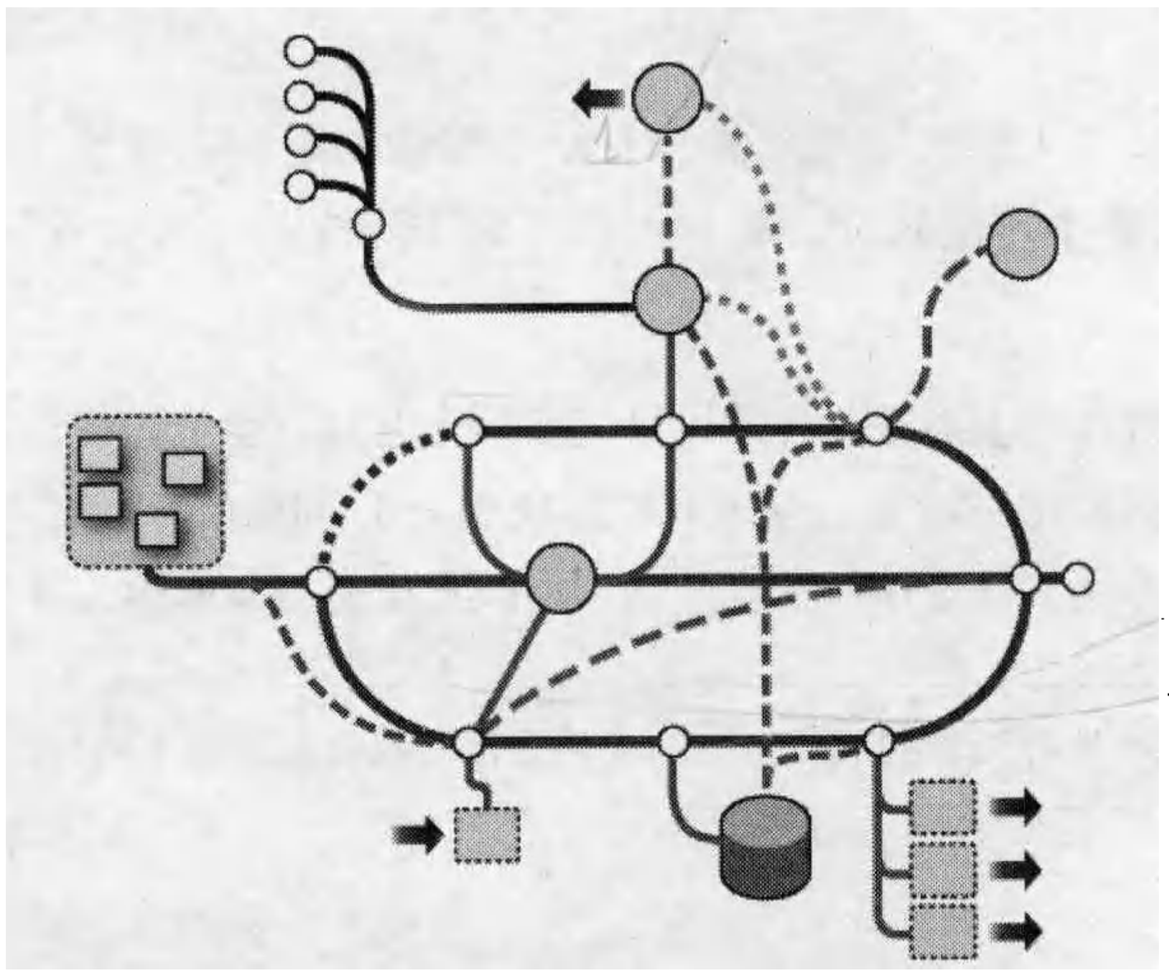


图2-1："混乱大都市"的"架构"

这就是那种让跑遍各地的销售员恼怒的系统。实际上，它与伦敦地铁的相似性让人印象深刻：从系统的一端到另一端有很多条路线，哪条路最好通常是不明显的。地理位置很近的目的地常常很难到达，你希望能在两点之间再挖掘一条隧道。有时候，走出地铁换乘巴士实际上是更好的选择。或者干脆步行。

无论从哪个角度来看，这都不是一个"好的"架构。"大都市"的问题超出了设计的范畴，它涉及开发过程和企业文化。这些问题实际上导致了许多架构腐烂。代码经过几年的"有机"生长。没有人进行过任何架构设计，而且各个部分是随着时间推移，没有经过太多思考就栓在一起的。我们这么说真的算是客气的了。没有人停下来为代码建立一

个理智的结构。它增长、膨胀，成为绝对没有任何架构设计的系统的一个典型。代码集从来不会没有架构。这个系统只是拥有一个很糟糕的架构。

如果我们回顾创建“大都市”的公司的历史，它所处的状态是可以理解的（但是不可宽恕）：这是一个初创的公司，快速提供许多新版本的压力很大。延期是不可容忍的——这会带来财务灾难。软件工程师被迫尽其极限，快速交付。所以代码是以一系列疯狂冲刺的方式垒在一起的。

注意：不好的公司结构和不健康的开发过程将在糟糕的软件架构中得到反映。

2.1.1 后果

“大都市”缺少城市规划，这带来了许多后果，我们将在这里进行分析。这些后果的影响是很严重的，远远超出了你对不良设计的天真想象。地铁变成了云宵飞车，飞速地朝下猛冲。

不可理解

正如你已经看到的，“大都市”的架构以及缺乏强制的结构，导致了一个很难理解的软件系统，实际上几乎不可能修改。新加入项目的团队成员（譬如我）会被复杂性惊呆，不能够搞清楚状况。

坏的设计确实会招致在它上面叠加坏的设计（实际上它简直就是迫使你那样做），因为没有一种明智的方式可以扩展该设计。在所有能解决手上工作的方法之中，阻力最小的总会被采用，没有明显的办法来修复这些结构问题，所以只要能减少麻烦，就会扔进去新的功能。

注意：重要的是要保持软件设计的品质。坏的架构设计会招致更坏的架构设计。

缺乏内聚

系统的组件完全没有内聚性。每个组件本来都应该有一个定义良好的角色，但是它们却包含了一堆杂乱的、不一定相关的功能。这使我们很难确定组件存在的原因，也很难弄明白系统中已经实现了哪项具体的功能。

很自然，这让缺陷修复成为了一场噩梦，严重地影响了软件的品质和可靠性。

功能和数据都放在了系统中错误的地方。许多你认为是“核心服务”的部分却没有在系统的核心部分实现，而是由边远的模块来模拟实现（非常痛苦并且代价很大）。

进一步的软件历史考察揭示了原因：原来的团队中存在个人斗争，所以一些关键程序员开始创建他们自己的软件小帝国。他们把自己认为酷的功能放到他们的模块中，即使它不应该属于那里。为了做到这一点，他们于是又实现了更为巴罗克式的通信机制，把控

制连回到正确的地方。

注意： 开发团队中健康的工作关系将直接有益于软件设计。不健康的关系和个性膨胀会导致不健康的软件。

内聚和耦合

软件设计的关键品质是内聚和耦合。这不是什么新奇的“面向对象”概念；自从20世纪70年代出现结构化设计开始，开发者对这一概念已经谈论了许多年。我们的目标是通过设计使系统的组件具备下列品质：

- 高内聚（Strong cohesion）

内聚是一个测量指标，说明相关的功能如何聚集在一起，模块内的各部分作为一个整体工作得如何。内聚性是将模块粘成一个整体的胶水。

弱内聚的模块是不良分解的信号。每个模块都必须具有清晰定义的角色，而不只是一堆不相关的功能。

- 低耦合（Low coupling）

耦合是模块之间独立性的测量指标——它们之间进出“电线”的数量。在最简单的设计中，模块几乎没有什么耦合，所以彼此间的依赖关系较少。显然，模块不能够完全解耦，否则它们将根本不能够一起工作！

模块之间的联系有多种方式，有的是直接的，有的是间接的。模块可以调用其他模块中的函数，或被其他模块所调用。它可能使用其他模块提供的Web服务或设施，可能使用其他模块的数据类型，或提供某些数据让其他模块使用（可能是变量或文件）。

好的软件设计会限制通信的线路，只提供那些绝对需要的。这种通信线路是确定架构的一部分因素。

不必要的耦合

“大都市”没有清晰的分层。模块之间的依赖关系不是单向的，耦合常常是双向的。组件A会到达组件B的内部，目的是完成它的一项任务。在其他的地方，组件B又通过硬编码调用了组件A。系统没有最底层，也没有控制中心。它是整体式的一大块软件。

这意味着系统的各部分之间耦合非常紧密，你想启动系统骨架就不得不创建所有的组件。单个组件的任何改变都会波及其他组件，需要修改许多依赖它的组件。孤立地看代码组件没有任何意义。

这使得低层次的测试不能够进行。不仅是代码层次的测试不可能进行，而且组件层次的集成测试也不能够创建，因为每个组件都依赖于几乎所有其他组件。当然，在公司中，测试从来也不具有很高的优先级（我们根本没有时间来做这种测试），所以这“不成为问题”。不必说，这个软件不太可靠。

注意：好的设计考虑到内部组件连接的连接机制和连接数（以及连接性质）。系统的单个部分应该能够独立存在。紧耦合将导致不可测试的代码。

代码问题

不良的顶层设计所带来的问题也影响到了代码层面。问题会引起其他问题（参见Hunt和Davis[1999]中关于破窗理论的讨论）。因为没有通用的设计，也没有整体项目“风格”，所以也没有人关心共同的编码标准、使用共同的库，或采用共同的惯例。组件、类和文件都没有命名惯例。甚至都没有共同的构建系统。胶带、Shell脚本、Perl胶水与makefile和Visual Studio项目文件混在一起。编译这个怪物被视为一场复杂的成人仪式！

“大都市”最微妙而又最严重的问题是重复。由于没有清晰的设计，也不清楚功能应该处于的位置，所以轮子在整个代码集中不断重新发明。一些简单的东西，如通用算法和数据结构，在许多模块中重复出现，每种实现都带有自己的一些未知的缺陷和怪异的行为特征。更大范围的关注点，如外部通信和数据缓存，也实现了许多次。

更多的软件历史考察揭示了原因：“大都市”开始是从一系列独立的原型拼起来的，这些原型本该抛弃。“大都市”实际上是偶然形成的城市群。当代码组件缝合在一起时，组件之间匹配得不好。随着时间的推移，这种随意的缝合开始破裂，所以组件互相拉扯，导致代码集破碎，而不是和谐地协作。

注意：松弛而模糊的架构将导致每个代码组件编写得不好，并且相互之间匹配得不好。它也会导致重复的代码和工作。

代码以外的问题

“大都市”内部的问题已经超越了代码集，在公司中其他的地方导致了混乱。不仅开发团队中有问题，而且架构的腐烂也影响到了支持和使用该产品的人。

开发团队

项目的新成员（例如我）被复杂性惊呆了，不能够搞清楚状况。这很好地解释了为什么很少有新人能在公司里待下来——员工流失率非常高。

那些留下来的人非常努力地工作，项目的压力非常大。规划新的功能会导致极大的恐惧。

缓慢的开发周期

由于维护“大都市”是一项恐怖的任务，所以即使是最简单的变更或“很小的”缺陷修复都不知道要花多少时间。管理软件开发周期非常难。客户只好等着实现重要的特征，管理层对开发团队不能满足业务目标感到越来越沮丧。

支持工程师

在支持这个极不寻常的产品时，产品支持工程师度过了可怕的时光，他们要设法弄明白很小的软件版本差异之间错综复杂的行为差异。

第三方支持

项目开发了一个外部支持协议，支持其他设备远程控制“大都市”。由于它只是软件内部结构上面薄薄的一层，所以它反映了“大都市”的架构，这意味着它也是巴罗克式的、难以理解的、容易偶尔出错的、不可能使用的。第三方工程师的生活也被“大都市”的可怕结构搞得一团糟。

公司内部政治

开发问题导致了公司内部不同“种族”的分裂。开发团队与营销销售团队之间关系紧张，每次新版本要推出时，制造部门总是要承受巨大的压力。经理们已经绝望了。

注意：不良架构的影响不仅限于代码。它会进一步影响到人、团队、过程和时间表。

清晰的需求

软件历史考察凸显了“混乱大都市”之所以混乱的一个重要原因：在项目开始之初，团队并不知道要构建的是什么。

本来的初创公司知道它要占领哪个市场，但不知道哪种产品能占领这个市场。所以他们两面下注，要求一个可以做许多事情的软件平台。噢，我们昨天就想得到它了。所以程序员们急急忙忙创建了一个毫无希望的总体基础设施，它具有做任何事情的潜力（但做得不好），而不是创建一个把一件事情做好的架构，并能够在将来进行扩展，做更多的事情。

注意：重要的是要在开始设计系统之前知道你打算设计什么。如果你不知道它是什么，也不知道它将做什么，暂时不要开始设计它。只设计你知道需要的东西。

在规划“大都市”的早期阶段，有太多的架构师。面对糊涂的需求，他们都拿着一块拼

不起来的拼图，试图独自工作。他们在工作时没有考虑到整个项目，所以当他们试图将这些拼图拼在一起时，就拼不起来了。没有时间进一步思考架构，软件设计的各个部分有一些重叠，于是开始了“大都市”的城市规划灾难。

2.1.2 现状

“大都市”的设计几乎完全是无可救药的——相信我，随着时间的推移，我们也尝试过修复它。修复工作需要返工、重构、修改代码结构中的问题，这些已经成为不可能的任务。重写也不是省事的方案，因为支持老的、巴罗克式的控制协议是需求的一部分。

你可以看到，“大都市”的“设计”产生的后果是残酷的，并且会无情地变得更糟。很难加入新的特性，所以人们只是忙着添加更多不完善的功能、救急补丁和编造的谎言。没有人在面对代码时感到愉快，项目正盘旋着向下栽。缺乏设计导致了不良的代码，从而又导致了不良的团队精神和不断变长的开发周期。这最终导致了公司严重的财务问题。

最后，管理层宣布“混乱大都市”已经不盈利了，它被抛弃了。对于任何组织机构来说，这都是勇敢的一步，特别是这个公司一直都眼高手低，同时又试图避免沉沦。带着团队从以前版本中得到的所有C++和Linux经验，他们在Windows上用C#重写了系统。猜猜看会怎么样。

2.1.3 来自“大都市”的名信片

那么我们学到了什么？不良的架构会产生深远的影响和严重的反弹。在“混乱大都市”中缺少预见性和架构设计，导致了下面的问题：

- 低品质的软件和漫长的版本发布周期。
- 系统没有弹性，不能够适应变更或添加新的功能。
- 无处不在的代码问题。
- 员工问题（压力大、士气低、跳槽等）。
- 大量混乱的公司内部政治。
- 公司不能成功。
- 许多痛苦和面对代码深夜加班。

2.2 设计之城

形式永远服从功能。

——Louis Henry Sullivan

“设计之城”软件项目表面上与“混乱大都市”非常相似。它也是用C++写的消费音频

产品，运行在Linux操作系统上。但是，它的构建方式有很大不同，所以内部结构也非常不同。

我从一开始就参加了“设计之城”项目。我们用有能力的开发者组成了一个全新的团队，从头开始构建这个产品。团队很小（开始有4名程序员），像“大都市”项目一样，团队的结构是扁平的。幸运的是，没有出现“大都市”项目中的个人对抗，在团队中也没有出现任何争权夺利的事。在此之前，团队成员之间并不非常熟悉，不知道我们在一起可以配合得多好，但我们对这个项目都很热心，喜欢这项挑战。

这样很好。

Linux和C++是项目早期的决定，这项决定确定了团队成员的组成。从一开始，项目就有清晰定义的目标：具体的首个产品和将来功能的路线图，代码集必须能够支持这些功能。这将是一个通用目标的代码集，可以适用于多种产品配置。

开发过程采用了极限编程（XP）（Beck和Andres 2004），很多人相信这种开发过程避开了设计：直接开始编码，不要想太远。实际上，一些旁观者对我们的选择感到震惊，并预言项目将以泪收场，就像“大都市”一样。但这是一种常见的误解。XP没有贬低设计，它贬低的是不必要的工作（即YAGNI原则，You Aren't Going To Need It）。但是，如果需要前端设计，XP就要求你进行设计。它也鼓励使用快速原型（所谓的“spike”），快速展现并验证设计的有效性。这些都非常有用，对最终的软件设计产生了极大的影响。

2.2.1 设计之城的第一步

在设计过程的早期，我们确定了主要的功能领域（这包括核心的音频通道、内容管理和用户控制/界面 ）。我们考虑了它们如何在系统中适配，推出了初步的架构，包括了实现性能需求所必需的核心线程模型。

系统中各独立部分的相对位置关系体现为传统的分层结构，图2-2展示了简化后的结果。请注意，这并不是庞大的前端设计。它是有意为之的“设计之城”的简单概念模型：图中只有一些大块，这是一个基本的系统设计，可以随着功能模块的添加而轻松地增长。虽然很基本，但这个初始架构为增长提供了坚实的基础。“大都市 ”没有总体规划，在“方便”的地方嫁接（或修补）功能。

我们在系统的核心上花了额外的设计时间：音频通道。它实际上是整个系统的一个内部子架构。为了确定它，我们考虑了穿过一系列组件的数据流，最后得到了一个“过滤器和管道”音频架构，如图2-3所示。根据产品的不同物理配置，它包含了这样一些管道。同样，开始时这些管道只是一个概念，即图中的一些方块。我们当时还没有决定如何将所有模块拼装在一起。

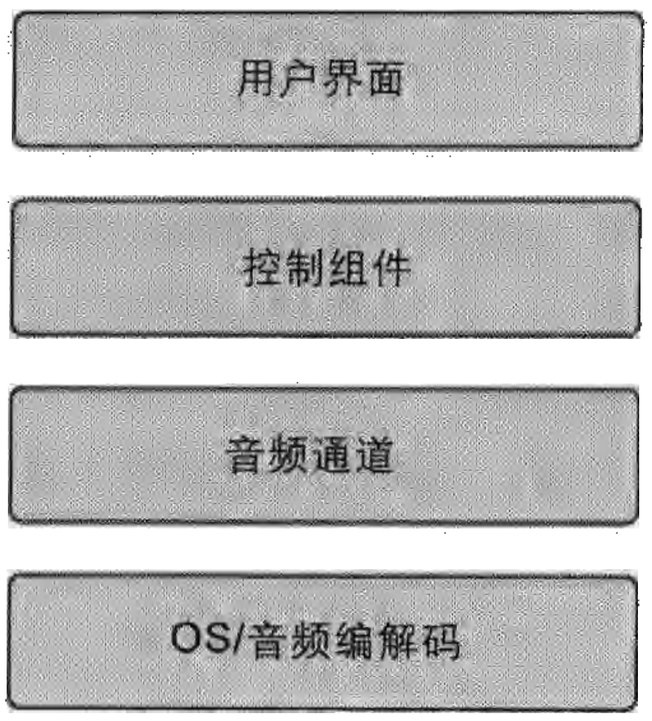


图2-2：“设计之城”的初始架构

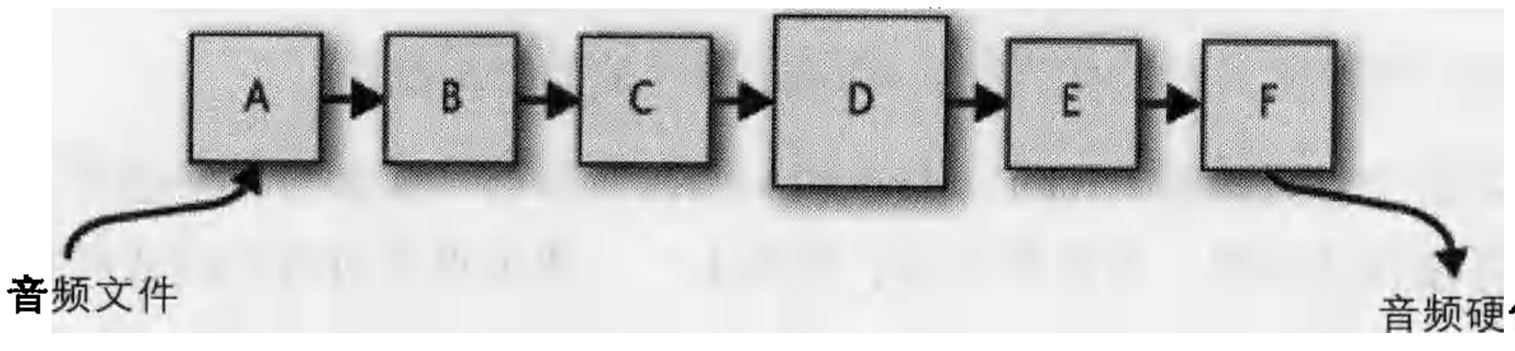


图2-3：“设计之城”的音频管道

我们在早期也选择了项目将采用的支持库（例如，可以从*http://www.boost.org*获得的Boost C++库和一组支持数据库的库）。关于一些基本关注点的决定是这时候做出的，目的是确保代码能够容易而一致地增长，这些决定包括：

- 顶层文件结构。
- 我们如何对事物命名。
- “内部”展示的风格。
- 共用的编码惯例。
- 选择单元测试框架。
- 支持基础设施（例如版本控制、适合的构建系统和持续集成）。

这些“细节”完美的因素非常重要：它们与软件架构密切相当，影响到后来的许多决定。

2.2.2 故事展开

在团队完成了初始设计之后，“设计之城”项目按照XP过程推进。设计和编码要么以结对的方式完成，要么经过仔细的复审，确保工作的正确性。

随着时间的推移，设计和代码不断发展和成熟；随着“设计之城”的故事逐渐展开，产生了下面的结果。

定位功能

由于从一开始我们就有系统结构的清晰总体视图，所以新的功能单元可以一致地添加到代码集的正确功能区域。代码应该属于哪一块从来就不是问题。在扩展功能或修复问题时，我们总是很容易找到已有功能的实现代码。

现在，把新的代码放到“正确”的位置有时候比简单“嫁接”到方便而不妥的地方而更难一些。所以，架构规划的存在有时候让开发者的工作变得更难一些。这些额外工作的回报就是今后的生活要容易很多，当我们维护或扩展系统时，不愉快的事情会很少。

注意：架构有助于定位功能：添加功能、修改功能或修复缺陷。它为你提供了一个模板，让你将工作纳入到一张系统导航图中。

一致性

整个系统是一致的。各个层次的所有决定都是在整个设计的背景下做出的。开发者从一开始就有意为之，这样得到的所有代码都完全符合系统设计，并与编写的所有其他代码相匹配。

在项目的历史中，尽管有许多变更，涉及代码集的各处（从单行代码到系统结构），但这些变更都符合最初的设计模板。

注意：清晰的架构设计将导致一致的系统。所有决定都应该在架构设计的背景下做出。

顶层设计的好风格和优雅很自然会为较低的层带来好处。即使在最低层，代码也是统一而整洁的。清晰定义的软件设计确保了没有重复，熟悉的设计模式到处使用，熟悉的接口惯例普遍采用，没有特殊的对象生命周期或奇怪的资源管理问题。代码是在城市规划的背景之中写成的。

注意：清晰的架构有助于减少功能重复。

架构的增长

有一些全新的功能领域出现在了设计“全图”中，例如存储管理和外部控制功能。在“大都市”项目中，这是致命的一击，难度超乎想象。但在“设计之城”项目中，事情就不一样了。

系统设计就像代码一样，被认为是可扩展、可重构的。开发团队的一项核心原则就是保持敏捷，没有什么是一成不变的，所以在需要时架构也可以修改。这促使我们让设计保持简单并易于修改。这样一来，代码可以快速地增长，同时又保持好的内部结构。添加新的功能块不是问题。

注意：软件架构不是一成不变的。需要时就改变它。要想做到可以修改，架构就必须保持简单。牺牲简单性的修改要抵制。

延迟设计决定

有一项XP原则确实提高了“设计之城”的架构品质，这就是YAGNI（如果你不是马上需要，就不要去做）。这促使我们在早期只设计了重要的部分，将所有余下的决定推迟，直到我们对实际的需求有了更清晰的理解并知道如何放到系统中最好时，再做出这些决定。这是一种非常强大的设计方法，在很大的程度上解放了我们的思想。

- 当你还不理解问题时就开始设计，这是一件糟糕的事。YAGNI迫使你等待，直到你知道真正的问题是什么，它应该怎样由设计来体现为止。这消除了猜测的工作，确保设计是正确的。

- 当你开始创建软件设计时就加入所有可能需要的东西（包括厨房水槽）是危险的。你的大部分设计工作会变成无用功，得到的只是额外的负担，你不得不在软件的整个变更生命周期中支持这些设计。它一开始就增加了成本，而且在项目的生命周期中不断增加成本。

注意：延迟设计决定，直到你必须做出这些决定为止。不要在你还不知道需求的时候就做出架构决定。不要猜测。

保持品质

从一开始，“设计之城”就准备好了一些品质控制过程：

- 结对编程。
- 对没有结对编程的工作进行代码/设计复审。
- 对每一段代码进行单元测试。

这些过程确保了系统中从未加入不正确的、不合适的变更。所有不符合软件设计的内容都被拒之门外。这可能听起来有点过于严厉，但这些是开发者们坚信的过程。

这种信念凸显了一个重要的态度：开发者们相信设计，认为设计对项目相当重要。他们拥有设计，对设计负责。

注意：必须保持架构品质。只有当开发者们相信它并对它负责时，才能做到这一点。

管理技术债务

除了这些品质管理方法之外，“设计之城”的开发是相当注重实效的。随着最后期限的临近，一些不太重要的功能被砍掉，让产品能够准时推出。小的代码“瑕疵”或设计问题允许存在于代码集中，要么是为了让功能快一点实现，要么是为了在接近发布时避免高风险的改动。

但是，与“混乱大都市”项目不同的是，这些逃避职责的地方被标记为技术债务，并安排在后续的版本发布中修正。这些问题很清楚，开发者对它们不满意，直到将它们处理掉为止。同样，我们看到了开发者对设计的品质负责。

单元测试打造了设计

关于代码集的一项核心决定就是所有代码都要有单元测试（这也是在XP开发中强制要求做到的）。单元测试带来了许多好处，其中一点就是能够修改软件的一些部分，而不必担心在修改的过程中破坏其他的东西。我们对“设计之城”内部结构的某些部分进行了相当激进的返工，单元测试给了我们信心，让我们相信系统的其他部分没有被破坏。例如，线程模型和音频管道的内部连接接口都进行了彻底的改变。这是在子系统开发较晚的时候发生的严重设计变更，但与音频通道接口的其他代码仍然执行得很好。单元测试让我们能够改变设计。

随着“设计之城”的逐渐成熟，这种类型的“主要”设计变更越来越少了。在经过一些设计返工之后，情况稳定下来，此后只有一些不重要的设计变更。系统开发得很快：以迭代的方式进行，每一次迭代都改进了设计，直到它达到了相对稳定的状态。

注意：你的系统应该有一组不错的自动化测试，它们让你在进行根本的架构变更时风险最小。这为你提供了工作的空间。

单元测试的另一个主要好处在于，它们在很大程度上定型了代码设计：它们实际上迫使我们实现好的结构。每个小的代码组件都被定型成定义良好的实体，可以独立存在，因为它必须能够在单元测试中构造出来，不需要围绕它构造系统的其他部分。编写单元测试确保了每个代码模块的内聚性，也确保了与系统其他部分之间的松耦合。单元测试迫使我们仔细考虑每个单元的接口，确保该单元的API是有意义的，内部是一致的。

注意：对你的代码进行单元测试将带来更好的软件设计，所以设计时要考虑可测试性。

设计时间

“设计之城”成功的另一个因素是分配的开发时间段，它既不长也不短（就像金发歌蒂的粥，既不热也不冷，刚刚好）。项目需要一个有利的环境才能获得成功。

如果时间太多，程序员常常会想创建他们的巨作（那种总是快要好了，但永远不会实现的东西）。有一点压力是好事，紧迫感有助于完成事情。但是，如果时间太少，就不可能得到任何有价值的设计，你只会得到半生不熟的解决方案，就像“大都市”那样。

注意：好的项目计划将带来优质的设计。分配足够的时间来创建架构杰作，它们不会立即出现。

与设计同行

尽管代码集很大，但它是一致而易于理解的。新的程序员可以比较容易地拿起代码并开始工作。不需要去理解不必要的复杂内部关系，也不需要面对奇怪的遗留代码。

由于代码中产生的问题比较少，工作起来有乐趣，所以团队人员的流失率很低。这是因为开发者们负责设计，并不断希望改进它。

看着开发团队动态地遵守架构设计是一件有趣的事情。“设计之城”的项目原则规定没有人“拥有”哪一部分设计，这意味着任何开发者都可以改动系统的所有地方。每个人都应该写出高品质的代码。“大都市”是许多不协作的、互相争斗的程序创造的一团混乱，而“设计之城”则是由密切合作的同事创建的一组干净、一致、密切合作的软件组件。在很大程度上，Conway法则（注）反过来也生效，团队的组织方式就像软件的组织方式一样。

注意：团队的组织方式必然对它产生的代码有影响。随着时间的推移，架构也会影响到团队协作的好坏。当团队瓦解时，代码的交互就很糟糕。当团队协作时，架构就集成得很好。

2.2.3 现状

在一段时间之后，“设计之城”的架构如图2-4所示。也就是说，它与最初的设计非常相似，同时也包含了一些值得注意的变更。此外，它还包含了大量的经验，证明这个设计是正确的。健康的开发过程，小的、更善于思考的开发团队，适当注意确保一致性，带来了极为简单、清晰、一致的设计。这种简单性为“设计之城”带来了好处，得到了可扩展的代码和快速开发的产品。

在编写本书时，“设计之城”项目已走过了3年。代码集仍在使用，而且扩展出了一些成功的产品。它还在开发、成长、扩展，还在每天发生变化。下一个月它的设计可能与这个很不同，但也可能没有不同。

我要澄清一点：这些代码并不完美。有些地方存在着技术争论，但是它们在整洁的背景下显得特别突出，会在将来得到解决。没有什么是一成不变的，由于适应性强的架构和灵活的代码结构，这些问题都可以解决。几乎所有东西都各就各位，因为架构很好。

注：　Conway法则指出，代码结构符合团队的结构。简而言之，“如果你让4个小组开发一个编译器，就会得到一个4阶段编译器。”

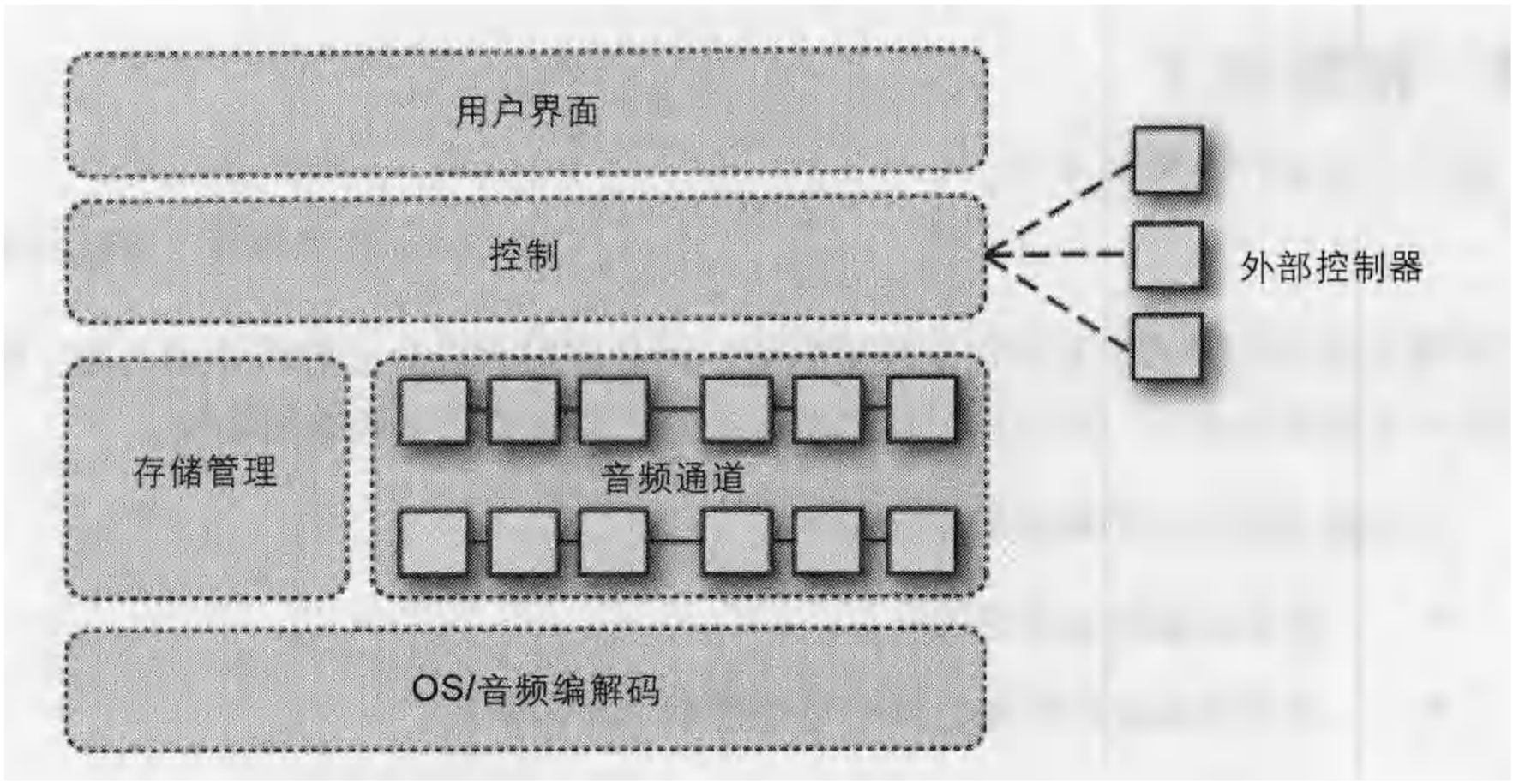


图2-4：“设计之城”的最终架构

2.3 说明什么问题

> 等那完全的来到，这有限的必归于无有了。
>
> ——《哥林多前书》第13章10节

这个关于两个软件系统的简单故事当然不是软件架构的全面介绍，但我已展示了架构如何对软件项目产生深远的影响。架构几乎会影响所有与之相关的人和事，它决定了代码集的健康，也决定了相关领域的健康。就像一个繁荣的城市会为当地带来成功和声望，好的软件架构将帮助项目获得发展，为依赖于它的人带来成功。

好的架构是很多因素的结果，包括以下方面（但不限于此）：

- 确实进行有意为之的前端设计。（许多项目甚至还没开始，就因为这一点而失败了。）
- 设计者的素质和经验。（以前犯过一些错误是有帮助的，这能在下一次为你指出正确方向！“大都市”项目肯定教会了我一些东西。）
- 在开发过程中，保持清晰的设计观点。
- 授权团队负责软件的整体设计，而团队也承担起这一责任。
- 不要害怕改变设计：没有什么是一成不变的。
- 让合适的人加入到团队中，包括设计者、程序员和经理，确保开发团队的规模合适。确保他们具有健康的工作关系，因为这些关系将不可避免地影响代码的结构。
- 在合适的时候做出设计决定，当你知道所有必要信息时再做出决定。延迟那些暂时不能做出的决定。
- 好的项目管理，以及合适的最后期限。

2.4 轮到你了

绝不要失去神圣的好奇心。

——阿尔伯特·爱因斯坦

你正在读这本书是因为你对软件架构感兴趣，而且你对改进自己的软件感兴趣。所以这里就有一个极好的机会。对于你目前的软件经验，请考虑以下简单的问题：

1. 什么是你看到过的最好的系统架构？
   - 你怎么知道它是好的？
   - 这个架构在代码集之内和之外带来了什么结果？
   - 你从中学到了什么？
2. 什么是你看到过的最差的系统架构？
   - 你怎么知道它是差的？
   - 这个架构在代码集之内和之外带来了什么结果？
   - 你从中学到了什么？

参考文献

Beck, Kent, with Cynthia Andres. 2004. *Extreme Programming Explained*, Second Edition. Boston, MA: Addison-Wesley Professional.

Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Professional.

Hunt, Andrew, and David Thomas. 1999. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley Professional.

第二部分

企业级应用架构

	原则与特性		结构
√	功能多样性		模块
√	概念完整性	√	依赖关系
	修改独立性		进程
	自动传播	√	数据访问
	可构建性		
√	增长适应性		
	熵增抵抗力		

第3章

伸缩性架构设计

Jim Waldo

3.1 简介

在设计系统架构时，一个比较有趣的问题就是确保系统在伸缩时的弹性。随着越来越多的系统运行在网络上或在互联网上提供访问，伸缩性正变得越来越重要。对于这样的系统，如果你希望误差的范围在几个数量级以内，那么容量规划的想法显然是荒谬的。如果你架起一个网站，然后它火了，你可能会突然发现有几百万的用户访问你的站点。同样容易出现的情况是，你架起了一个网站，却发现没有人感兴趣，你投入的所有设备都闲置着，消耗着能源和管理成本，浪费钱财（这同样是一种灾难）。在网络世界里，一个站点可以在几分钟内从其中一种状态转变成另一种状态。

只要是将系统连接到网络上，每个人都会遇到伸缩性问题，但是“大型多人在线游戏”（MMO）和虚拟世界特别关注这一点。这些系统必须具备伸缩性，以满足大量的用户。Web服务器的用户常常读取的是静态的内容，而且彼此之间没有交互，但MMO中的玩家或虚拟世界中的居民则不同，他们既需要与所处的世界进行交互（这改变了世界的基本信息），也需要彼此之间的交互。这些交互行为使得这类系统基础设施的伸缩性问题变得更复杂，因为用户与系统的交互几乎是独立的（除了那些不独立的情况），而且不会让世界的状态改变太多。对于一个世界里的任意两个参与者，他们在某个时刻进行交互的几率是非常小的。但是，几乎所有玩家在所有时候都在与他人交互。结果是这种系统并行程度非常高，但只有少数的交互是互相依赖的。

由于这些系统所培育起来的文化，MMO和虚拟世界的伸缩性问题进一步复杂化了。MMO和虚拟世界都源自于视频游戏产品。这是一种从PC游戏和游戏机游戏传统中成长起来的文化，在这种传统中，程序员会假定游戏运行在一台独立的机器或游戏机上。在这样的环境中，机器所有的资源都受游戏程序控制，程序的问题也限于单个用户玩游戏的情况（实际上，缺陷或奇怪的行为常常会被认为是游戏本身逻辑的一部分）。

这些游戏和编写、出品、扩展它们的公司，都属于娱乐行业。编写游戏的团队由一个出品人领导，有剧本和背景故事。游戏的目标是刺激、打动人，最重要的一点，要好玩。可靠性很好，但不一定必需。可扩展性是游戏的特性，让游戏在升级时能够加入新的故事情节和主题，但可扩展性不是代码的特性，不必让代码能以新的、不同的方式使用。

在线游戏和虚拟世界的兴起将这种文化带入了一个新的环境，在这个环境中，需求与企业应用开发者所面对的类似。多个用户通过网络在服务器上交互时，由于一个玩家的意外动作而导致的服务器崩溃将影响许多其他玩家。随着这些世界发展出自己的经济（有些经济与现实世界的经济有关系），在线世界的稳定性和一致性就超出了一个游戏的要求。随着这些世界中玩家或居民的人数达到百万级，伸缩的能力就成为了任何架构的首要需求。

Darkstar项目（本章后面将简称为Darkstar）是对这些游戏和虚拟世界创建者的需求挑战的回答。这个项目由Sun公司实验室的一个研究小组承担，它将在架构的伸缩性领域不断探索。这个项目特别有趣之处在于，它是针对MMO和虚拟世界的创建者，这些程序员与我们（作为系统设计者）所熟悉的那些程序员相比，有着非常不同的需求。得到的架构看起来似乎很眼熟，但如果你仔细查看，会发现它的不同之处，它与你的经验有所不同。得到的架构有着属于它自己的美丽，同时它也是一堂实践课，说明了不同的需求如何改变你所想到的构建系统的方式。

3.2 背景

像一座建筑或一个城市的物理架构一样，系统的架构必须适应环境，利用该架构创建的工件将存在于该环境之中。对于物理架构来说，这个环境包括工作的历史环境、它所处位置的气候、本地工人的技能、可以获得的建筑材料，以及建筑的使用意图。对于软件架构，这个环境不仅包括使用该架构的应用程序，也包括那些要使用该架构的程序员，以及由此受到的系统约束。

在创建Darkstar架构时，我们（注1）意识到的第一件事就是所有针对伸缩性而设计的架构都需要包含多台机器。我们不清楚，就算是最大的大主机系统是否能够满足今天的一

注1：在提到Darkstar项目的架构开发时，我通常会说“我们”做了什么，而不是“我”做了什么。这不仅是编辑意义上的“我们”。该架构的设计是一个协作项目，由Jeffrey Kesselman、Seth Proctor和James Megquier发起，并经过Seth、James、Tim Blackman、Ann Wollrath、Jane Loizeaux和我的努力发展到现在的状态。

些在线游戏的要求（例如《魔兽世界》，据报道它有500万注册用户，几十万的同时在线用户）。就算单台机器能够处理这种负载，我们也不能在一开始就假定游戏会取得如此成功，需要这样的硬件投资。这在经济上是不可行的。这种应用需要能够从很小的系统开始，然后随着用户数的增长而增加处理能力，最后随着大家对游戏兴趣的衰退而降低处理能力。这与分布系统的特点相符，在分布式系统中，我们可以随着请求增长而添加（合理的小）机器，当请求下降时移走机器。所以我们从一开始就知道，总体架构必须是一个分布式系统。

我们也知道系统利用了芯片架构的当前趋势。MMO和虚拟世界（在较小程度上）曾经针对伸缩性利用过摩尔定律。随着处理器的速度倍增，可以创造的世界会在复杂度、丰富程度和互动性方面倍增。没有其他计算领域像游戏世界这样探索过处理器速度的增长所带来的好处。为游戏而设计的个人计算机总是将CPU速度、内存和图形性能推向极致。游戏机更激进地将这些方面推向极致，它们包含的图形系统远远超过了高端工作站上的图形系统，整个机器完全是为了游戏玩家的特殊需要而打造的。

芯片演进方面最近的变化是从不断增加的时钟速度转为实现多核处理器，这已经对游戏中能做的事情产生了影响。新芯片的设计目标不是将一件事做得更快，而是同时做多件事。如果在芯片上运行的多项任务实际上可以同时执行，那么在芯片层面上引入并发执行将带来更好的总体性能。在不改变时钟速度的情况下，4核的芯片应该能比单核的芯片多做3倍的事情。实际上，这种增长不是呈线性的，因为系统的其他一些部分没有以同样的方式实现并发。但是可以通过并发来实现系统总体性能的增长，而且制造这种并发的芯片比制造增加时钟速度的芯片要容易得多。

基于这一事实，MMO和虚拟世界应该是多核芯片和分布式系统的理想候选者。在MMO或虚拟世界中发生的大多数事情就像在真实世界中发生的大多数事情一样，与该世界中发生的其他事情是无关的。玩家继续他们的搜索或装饰他们的房间。他们与怪物交战或设计衣服。即使他们与该世界中的另一个玩家或居民发生交互，也只是与该世界的很少一部分居民发生交互。这正是令人为难的并行计算任务的特点，也正是多核和多机系统应该擅长处理的那种任务。

尽管这些系统中的任务的并行度让人为难，但为这样的系统编程的程序员却没有接受过分布式计算或并发编程方面的训练，也没有这方面的经验。这是极为微妙的领域，即使是在这个领域接受过训练的人和对这些技术相当有经验的人也会感到困难。要让大多数游戏程序员来开发高度并发的、分布式游戏服务器，就是要求他们做超出自己的专长和经验的事。

3.2.1 首要目标

这样的背景为我们确立了该架构的首要目标。对伸缩性的需求表明，系统应该是分布式

的、并发的，但我们需要为游戏开发者提供简单得多的编程模型。简而言之，目标就是游戏程序员应该把该系统视为一台单机，运行着一个线程，所有允许部署到多线程和多计算机上的机制都应该由Darkstar项目的基础设施来考虑。

在一般的情况下，对应用程序隐藏分布式和并发是不可能的。但MMO和虚拟世界不是一般的情况。我们试图实现的这种隐藏，其代价就是必需一种非常特别的、严格限制的编程模型。幸运的是，这种模型恰好非常适合游戏服务器和虚拟世界已经采用的编程方式。

Darkstar项目要求的一般编程模型是反应式的，在这种编程模型中，游戏的服务器端写成了事件监听器，监听客户端生成的事件（客户端就是游戏玩家使用的机器，通常是一台PC或游戏机）。如果检测到事件，游戏服务器就应该生成一项任务，这个任务是一个短期的计算序列，包括操作虚拟世界中的信息，并与最初生成事件的客户端或其他一些客户端进行通信。任务也可以由游戏服务器自己生成，要么是响应某些内部的变化，要么是周期性地根据时间来生成任务。在这种情况下，游戏服务器可以在游戏或虚拟世界中生成一些角色，这些角色是不受外部玩家控制的。

这种编程模型非常适合游戏和虚拟世界，但它也应用于一些企业级的架构中，如J2EE和Web服务。之所以需要创建一个不同于这些企业计算机制的架构，是因为MMO和虚拟世界存在的环境非常不一样。这种环境几乎刚好和经典企业环境相反，这意味着如果你接受过企业环境方面的训练，你知道的所有事情在这个新世界中几乎都是错的。

经典的企业环境可以描述为一个“瘦”客户端连接到一个“胖”服务器 （这个服务器又常常连接到一个更“胖”的数据库服务器）。服务器将保存客户端需要的绝大部分信息，在最理想的情况下，客户端内存不多，没有自己的硬盘，它是服务器的一个称职的显示设备，绝大多数真正的工作在服务器上完成。

3.2.2 游戏世界

MMO和虚拟世界的环境始于一个非常胖的客户端：它通常是顶级的PC、具有最强劲的CPU、很大的内存，以及本身计算能力就很强的显卡。它也可以是一台游戏机，专门为图形密集的、高度交互的任务而设计。只要有可能，数据就会存放在这些客户端，特别是那些不会改变的信息，如地理信息、材质贴图和规则集。服务器保持尽可能的简单，通常只保存非常抽象的世界表示和其世界中的实体的表示。而且，服务器的设计目标是尽可能少地进行计算。绝大部分的计算留给了客户端。服务器的真正工作是保存共享的世界真实状态，确保不同客户端对世界的看法差异可以根据需要得到纠正。真实状态需要由服务器来保存，因为控制客户端的玩家很有兴趣让他们的表现变成最强，所以可能会受到诱惑，根据他们的喜好修改共享的真实（如果他们可以）。在一般情况下，如果有可能，玩家就会作弊，所以服务器必须是共享真实的最终来源。

MMO和虚拟世界的数据访问模式也和企业中看到的情况有着很大的区别。企业中的一般经验法则是90%的数据访问都是只读的，大多数任务会读取大量数据，然后再改写少量数据。在MMO和虚拟世界的环境中，大多数任务只访问服务器上少量的状态数据，但在它们访问的数据中，大约一半会被改写。

3.2.3 延迟是敌人

但是，这两种环境中最大的不同要追溯到用户所做的事情的不同。在企业环境中，目标是管理业务，如果总吞吐量得到改进，在处理中有一点延迟是可以接受的。在MMO和虚拟世界的环境中，目标是开心，而延迟是开心的敌人。所以MMO或虚拟世界的基础设施需要围绕着尽可能限定延迟的需求来设计，即便以吞吐量为代价也在所不惜。

在线游戏和虚拟世界显然已经找到了办法来实现伸缩性，以应对数量巨大的用户。目前的方法可以分成两大类。第一类实质上是基于地理位置来实现的。游戏设计成包含一组不同的区域，每个区域运行在一台服务器上。它可能是虚拟世界的一个岛或房间，也可能是在线游戏中的一个小镇或山谷。游戏设计试图让每个区域无关，限定地理区域的大小，这样服务器不会因太多用户进入这个区域而拥塞。在实践中，这样的区域常常能实现自我限制，因为当服务拥塞时，游戏就会变得响应比较慢，趣味性下降。因此，玩家就会转向更有趣的区域，这使得以前拥塞的区域人数减少，响应时间得到改进。

将不同地理区域分配给不同服务器来实现伸缩性的方法有一个问题，即必须在游戏编写时决定哪些区域应该放到一台服务器上。虽然在游戏或虚拟世界中添加新的区域相当容易，但是改变已经分配给服务器的区域却可能需要改动代码。决定哪些区域组成一个伸缩性单位，这必须是开发工作的一部分。

第二种处理游戏或虚拟世界中拥塞区域的方法叫做分区（sharding）。一个分区是该区域的一份副本，运行在它自己的服务器上，独立于其他的分区，它代表了游戏中相同的部分，即原来的区域。这样，分区可能代表了某个房间或村庄的不同副本，允许成倍的玩家进入到世界的这个部分中。分区的缺点是它们不允许处于不同分区的玩家彼此之间进行交互。随着游戏和虚拟世界变成更多的社交体验，而不仅仅是玩游戏，这种缺点就明显了。玩家的目标不只是要出现在虚拟世界中，而是要和他们的朋友（真实的和虚拟的）一起进入虚拟世界。分区阻碍了这个目标的实现。

因此，Darkstar架构的另一个主要目标就是支持随时伸缩，同时不要求游戏逻辑受到伸缩的影响。这个架构应该支持游戏动态地响应负载，而不是让这种响应成为游戏设计工作的一部分。

3.3 架构

Darkstar由一组独立的服务构成，这些服务可以在游戏或虚拟世界的服务器端的地址空间内获得。每个服务都定义为一个小的编程接口。尽管不是出于本意，但Darkstar项目提供的一些基本服务很像经典操作系统的服务，它们支持游戏或虚拟世界的服务器端访问持久存储，调度并执行任务，与游戏或虚拟世界的客户端进行通信。

用一组相互联系的服务来构建这个系统，显然是开始了“分而治之”的过程，分而治之是设计所有大型计算机系统的基本方法。每种服务都可以用一个接口来描述，这让使用该服务的程序不会受到底层实现变更的影响，同时也支持这些实现可以独立地完成。对一个服务的实现进行变更不应该影响到另一个服务的实现，即使其他的服务会利用到这个变更的服务（假定接口和接口的语义没有变更）。

我们采用服务分解的方法还有其他的原因。从一开始，Darkstar项目就设计成一个开放源代码的项目，希望我们能够放大核心团队的工作，支持其他社区成员创建更多的服务，丰富核心的功能。维护一个开源社区在任何情况下都是复杂的，我们相信在组成架构的服务之间拥有最大程度的隔离，将支持在不同服务实现层次之间的更高级的隔离。此外，当时并不清楚是否存在单一一组服务能够适合所有的MMO和虚拟世界。将基础设施设计为一组独立的服务，使得这些服务的不同组合可以在不同的情况下使用，这由使用该基础设施的具体项目的需求来决定。Darkstar栈中具体包含哪些服务可以由一个配置文件来设置。

3.3.1 宏观结构

图3-1展示了基于Darkstar项目基础设施的游戏或虚拟世界的基本结构。一些服务器构成了游戏或虚拟世界的后端。每个服务器运行着一组选定服务的副本（称为Darkstar栈）和游戏逻辑的副本。客户端将连接到其中一个服务器，与服务器保存的该世界的抽象表示进行交互。

与大多数的复制策略不同，游戏逻辑的不同副本不需要处理相同的事件。每个副本可以独立地与客户端进行交互。在这个设计中，复制主要用于支持伸缩性，而不是确保容错（虽然我们后面会看到，容错也实现了）。而且，游戏逻辑本身不知道、也不需要知道在其他机器上运行着服务器的其他副本。游戏程序员编写的代码就像在一台机器上执行一样，不同副本之间的协作由Darkstar项目的基础设施来完成。实际上，如果游戏的容量只需要一台服务器，基于Darkstar的游戏就能够在一台服务器上运行。

客户端连接到游戏逻辑使用的通信机制是基础设施的一部分。这些机制支持客户端到服务器的直接通信，也支持一种“发布-订阅”通道，任何发往通道的消息都会送达该通道的所有订阅者。

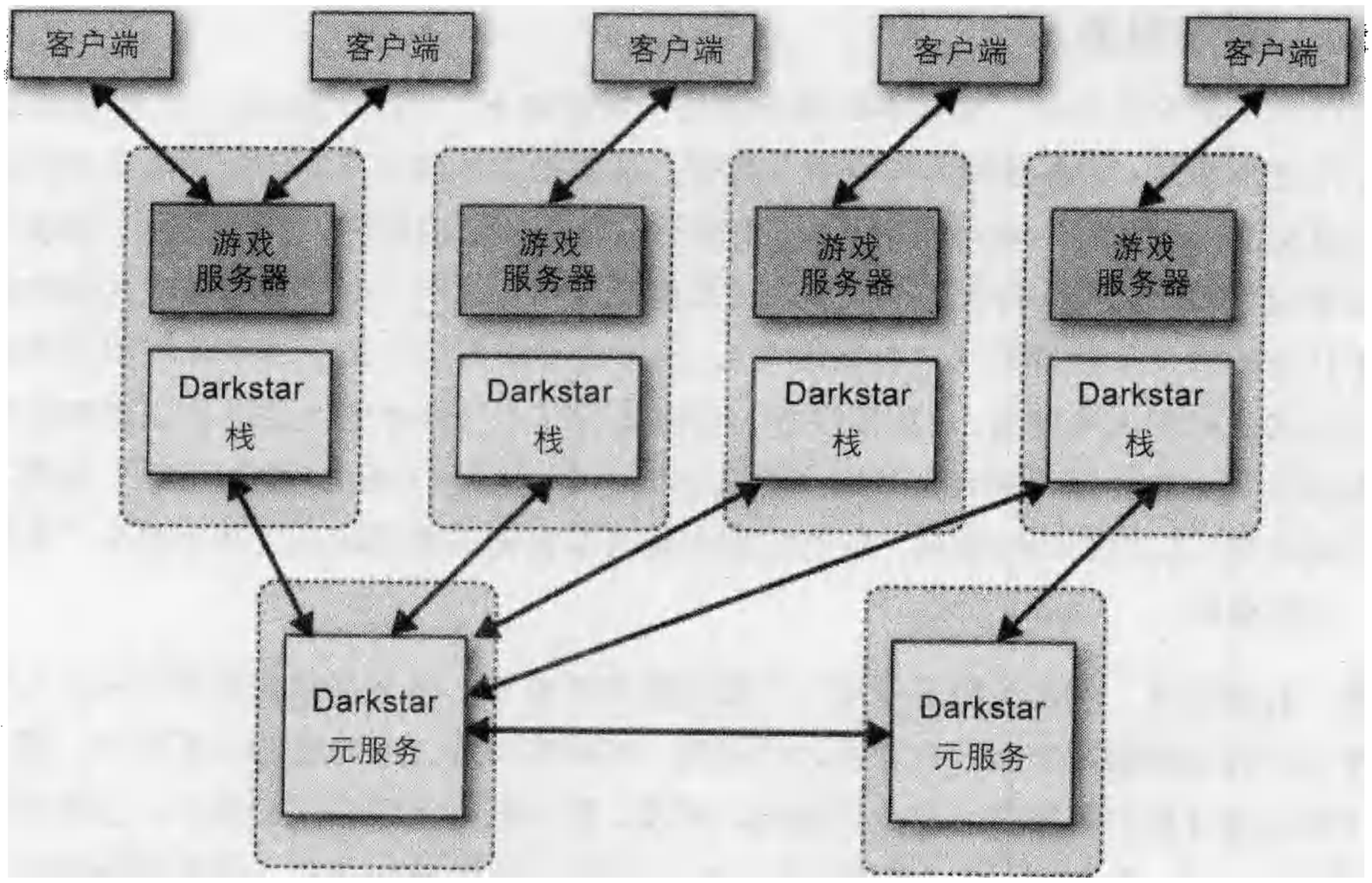


图3-1：Darkstar项目的高层架构

Darkstar栈由一组元服务来协调，这是一组网络访问服务，游戏或虚拟世界的程序员是不可见的。这些元服务支持栈的各个副本之间进行协作，共同运营整个游戏。例如，这些元服务将所有独立的副本持续工作，如果某个副本失效，就会发起失效恢复。这些元服务还会跟踪各副本的负载，在需要的时候重新分配负载，或者随时添加新的服务器，增加总体容量。由于这些服务对于Darkstar项目的用户来说是完全隐藏的，所以它们可以随时改变或移除，或者添加新的服务，这都不需要修改游戏或虚拟世界的代码。

对于在Darkstar项目环境中创建游戏或虚拟世界的程序员来说，可见的架构就是栈中包含的一组服务。服务的全集是可以改变和配置的，但4个基本服务必须存在，它们构成了运营环境的核心，如图3-2所示。

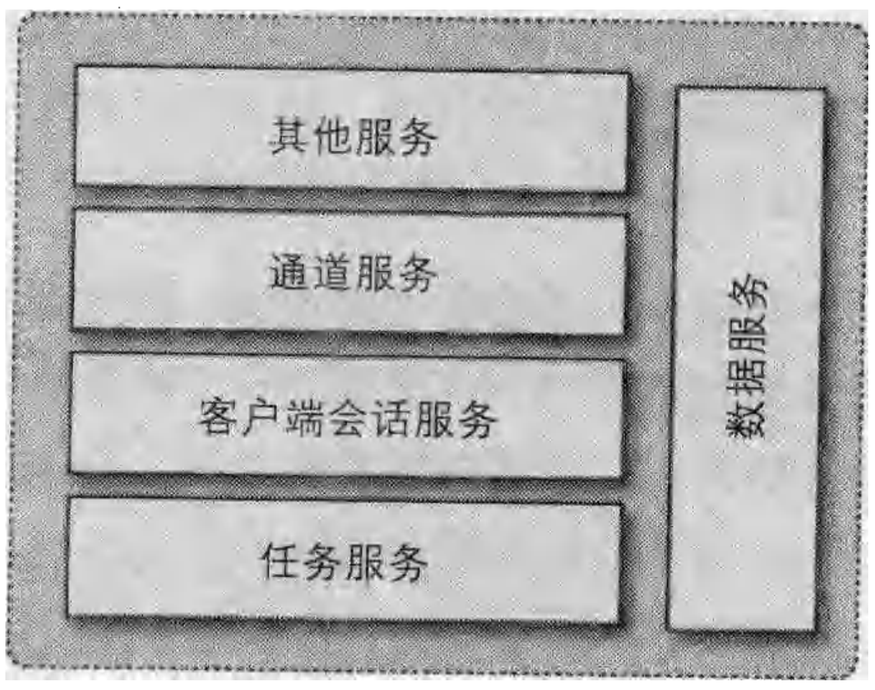


图3-2：Darkstar栈

3.3.2 基本服务

在这些栈层面的服务中，最基本的服务就是“数据服务”（Data Service），游戏或虚拟世界用它来保存、读取和操作所有持久数据。这里的持久概念可能比其他系统中的持久概念更宽泛。对于在Darkstar项目环境中编写的游戏或虚拟世界，任何存在时间超过一个任务的数据都被视为持久的，必须在“数据服务”中保存。我们曾假定在这种编程模型中任务的时间是短暂的（这也是需求），所以几乎所有用于表示游戏或虚拟世界的服务器端的数据都需要持久。“数据服务”也将运行在不同服务器上的游戏或虚拟世界的副本联系在一起，因为所有这些副本都共享同一个（概念上的）“数据服务”实例。所有的副本都会访问相同的数据，所有的副本都可以根据需要读取或改变存储在“数据服务”中的数据。

虽然“数据服务”看起来像是使用一个数据库的好地方，但是存储的需求实际上与通常条件下对标准数据库的需求有着很大的差别。存储的对象之间静态的关系很少，游戏中也不需要对存储的内容进行复杂的查询。相反，简单的命名策略就足够了，包括在编程语言层面上对对象的引用。“数据服务”也必须针对延迟进行优化，而不是针对吞吐量来优化。特定服务要访问的对象个数可能很少（我们初步的测算基于一些游戏和虚拟世界的原型，这些测算表明每个任务大约访问一打对象），在这些访问的对象中，大约一半会在任务执行中改变。

第二个栈层面的服务是“任务服务”（Task Service），它用于调度或执行任务。这些任务要么是响应从客户端收到的某个事件，要么是由游戏或虚拟世界服务器本身的内部逻辑发起的。绝大部分任务是一次性事件，是由于客户端的某种动作产生的，它们从“数据服务”中读取一些数据，操作这些数据，可能还进行一些通信，然后结束。任务也可能生成其他的任务，或者生成定期任务，在特定的时间执行或以特定的时间间隔执行。所有任务的执行时间必须很短，执行一项任务的最大时间是一个可配置的值，但默认值是100毫秒。

游戏或虚拟世界的程序员会看到因事件或服务器逻辑本身而生成的单个任务，但在底层，Darkstar的基础设施正尽其所能调度最多的任务。特别地，由服务器逻辑生成的任务与响应客户发起的事件而生成的任务是并行执行的，就像响应不同客户端而生成的任务一样。这样的并发执行可能导致数据竞争。要处理这种竞争，就需要“任务服务”和“数据服务”协作。在底层，在服务器程序员不可见的地方，“任务服务”调度的每个任务都包装在一个事务中。这个事务确保了任务中的所有操作要么全部完成，要么都不完成。此外，所有改变“数据服务”中对象的值的操作都由该服务作为中介。如果有多个任务试图改变相同的数据对象，只有一个任务会执行，其他任务都会中止，并安排在稍后执行。执行的那个任务会运行到结束。当执行的任务结束时，其他的任务就可以执行了。虽然

服务器程序员可以说明访问的数据将被修改，但这不是必需的。如果数据对象先被读取，然后被修改，“数据服务”会在任务提交之前检测到这种修改。在读取时就说明打算进行修改，这是一种优化，能够更早地检测到冲突，但是不事先说明修改的意图也不会影响程序的正确性。

将任务包装到一个事务中意味着通信机制也必须支持事务，只有当包装了消息发送任务的事务提交时，消息才会发出。这是通过Darkstar栈中余下两项核心服务来完成的。

3.3.3 通信服务

第一个服务是“会话服务”（Session Service），它是客户端和游戏或虚拟世界服务器之间通信的中介。在登录和认证后，客户端与服务器之间就会建立起一个会话。服务器通过会话监听客户端发出的消息，解析消息的内容，确定生成怎样的任务来响应该消息。客户端通过会话接收来自服务器的响应。这些会话隐藏了客户端和服务器的真实端点，这一点对于Darkstar的多机伸缩性策略是很重要的。会话也负责确保维持消息的顺序。如果来自某个客户端的前一条消息所引发的任务还没有完成，后一条消息就不会提交。在“会话服务”对任务进行这样的排序之后，“任务服务”就得到了极大的简化。“任务服务”可以假定它在任何时候收到的任务在本质上都是并发的。对来自特定客户端的消息排序是Darkstar框架中唯一的消息排序保证机制，外部观察者看到的多个客户端之间的消息顺序，与游戏或虚拟世界内看到的顺序有很大不同。

Darkstar栈中总可以得到的第二种通信服务是“通道服务”（Channel Service）。通道是一种一对多的通信机制。在概念上，通道可以由任何数目的客户端加入，任何发往该通道的消息都会送达所有与通道相关的客户端。这里似乎是应用端到端技术的好地方，可以让客户端之间直接通信，不会增加对服务器的负载。但是，这种通信需要由一些受信任的代码来监控，确保玩家不会利用不同的客户端实现来发送不正确的消息或欺骗消息。既然客户端假定是在用户或玩家的控制之下，那么客户端的代码就不能信任，因为很容易把原来的客户端代码换成另外的“定制过的”客户端代码。所以，实际上，所有通道消息都必须经过服务器，（可能）在经过服务器逻辑检查之后。

会话和通道的复杂性有多种原因，其中之一就是它们必须遵守任务的事务语义。因此，会话连接或通道上的实际消息传送不能够在调用相应的`send()`方法时发生，它只能够在该方法所处的任务提交时才能发生。

这些通信机制为我们实现伸缩性机制的第二部分奠定了基础。既然所有通信都必须通过Darkstar会话或通道的抽象层，而这些抽象层又不暴露客户端或服务器通信的真实端点，那么在实体通信和通信起止端的实际位置之间就存在着一个抽象层。这意味着我们可以在Darkstar系统中将服务器通信的端点从一台机器移到另一台机器，同时不会改变客户

端对这次通信的感觉。从游戏或虚拟世界的视角来看，通信也是经过一个会话或通道。但是底层的基础设施可以随着时间的推移和负载的变化，根据负载平衡的需要，将会话或通道从一台机器移到另一台机器。

3.3.4 任务的可移动性

要实现负载均衡的能力，其关键之处在于，对于我们要求的编程模型和必须使用的基本栈服务，响应客户端事件或游戏内部事件的任务可以从任何一台运行着Darkstar栈和游戏或虚拟世界副本的机器上移动到另一台同样的机器上。任务本身是用Java编写的（注2），这意味着只要（物理）机器的运行时栈中包含相同的Java虚拟机，任务就能够运行。任务读取和操作的所有数据必须从“数据服务”获得，“数据服务”是所有机器上的游戏或虚拟世界的实例和Darkstar栈所共享的。通信由“会话服务”或“通道”来实现中介，它们抽象了通信的真实端点，而且支持特定的会话和通道从一个服务器移动到另一个服务器上。因此，所有任务都可以运行在任意一个游戏服务器的实例上，同时不改变任务的语义。

这使得Darkstar的基本伸缩机制看起来很简单。如果有一台机器超载了，只要将一些任务从这台机器移到另一台负载较小的机器就行了。如果所有的机器都超载了，就向运行着Darkstar栈和游戏/虚拟世界的集群中添加新的机器。底层的负载平衡软件会将负载分发给新的机器。

对单台机器的负载进行监控并在需要时重新分配负载，这是元服务的工作。这些元服务是网络层面的服务，对于游戏或虚拟世界的程序员是不可见的，但是它们对Darkstar栈中的服务是相互可见的。例如，这些元服务会监控哪些机器正在运行（并监控是否有机器失效），哪些用户与某台机器有关，不同的机器当前的负载。由于元服务对于游戏或虚拟世界的程序员是不可见的，所以它们在任何时候都可以改变，不会影响到游戏逻辑的正确性。这让我们能够尝试不同的策略和方法，实现系统的动态负载平衡，也让我们能够丰富基础设施所需的元服务集合。

同样，我们使用实现多机伸缩机制来实现系统的高容错。由于任务和通信机制所使用的数据是与具体机器无关的，所以很明显，我们可以将任务从一台机器移到另一台机器上。但是如果机器失效，我们如何恢复在那台机器上执行的任务呢？答案很简单：任务本身也是持久对象，保存在系统的“数据服务”中。因此，如果一台机器失效，该机器上正在执行的所有任务都被视为中断的事务，会重新调度在不同的机器上执行。尽管这种重

注2：更准确地说，所有的任务都由字节码的序列组成，可以在Java虚拟机上执行。我们不关心源语言是什么，我们只关心从源语言编译出来的结果可以运行在所有支持游戏或虚拟世界的分布式环境中。

新调度比在一台机器上重新调度中断的任务的延迟要长，但系统的正确性是不变的。系统的用户（游戏玩家或虚拟世界的居民）顶多会注意到响应时间暂时有点延长。这样的延迟让人有点不舒服，但总好过现在其他游戏或虚拟世界环境中服务器崩溃所造成的影响。在那些环境中，会导致玩家掉线，还可能导致相当一部分游戏状态的丢失。

3.4 关于架构的思考

也许所有人关于架构及其实现的第一个问题就是它的性能。虽然未经深思熟虑就对架构进行优化是诸多罪恶之源，但是我们也可能设计出一个架构，而它的实现根本不可能达到好的性能。由于Darkstar架构的一项基本选择，这种担心是真实的。而且由于游戏行业的特点，确定服务器基础设施的性能是很难的。

要确定游戏或虚拟世界服务器基础设施的性能，其难度源自一个简单的事实：没有针对大规模MMO或虚拟世界的性能测试标准或共同接受的例子。缺少性能测试标准并不奇怪，因为绝大多数游戏或虚拟世界的服务器组件都是针对特定的游戏或虚拟世界从头开发的。只有少数的通用基础设施可以作为可复用的构建块，这些组件一般是事后从特定的游戏或虚拟世界中提取出来的，提供给其他构建类似游戏的人使用。也许是因为游戏行业还比较年轻，或是因为娱乐业中出现重要技术的偶然性，结果是没有共同接受的性能测试标准用于测试新的基础设施，也无法对不同的基础设施进行比较。

关于游戏或虚拟世界的预期计算、数据操作和通信负载也基本上没有什么资料，所以很难创建性能测试标准程序。部分原因是已有的服务器都是定制的。每台服务器都是为特定的游戏或虚拟世界设计的，所以考虑的是那款游戏或虚拟世界的具体负载特征。而且，这种状况也是因为游戏业的强烈的保密性。在游戏业中，关于一款游戏开发的信息是严格保密的，而且发布游戏的实现方式也是严格保密的。同时，行业中的许多人对这些信息是不感兴趣的。大量的思考和讨论集中在艺术设计、故事情节或玩家交互模式上，这些令新游戏更有趣、更好玩，而不是游戏服务器的设计方式和游戏为支持并发玩家（这个数字也是严格保密的）所采取的伸缩性技术。所以，获得现有的游戏和虚拟世界的这种服务器负载信息都很困难。

根据我们的经验，即使我们能够找来开发者，谈论他们的游戏或虚拟世界加在服务器上的负载，他们也常常会报告错误的信息。这不是因为他们想保持商业优势而故意错误报告他们服务器的情况，而是因为他们真的自己不也了解。在游戏服务器上基本不会加入一些手段，让他们收集有关服务器真实性能或完成事务的信息。对于这种服务器的分析一般最多是经验性的。程序员在服务器上工作，直到它让游戏玩起来有趣，这是一种迭代式的工作方式，而不是仔细对代码本身进行测量。在这些系统中，更多的是手工技术活，而不是科学测定。

这并不是说，这些游戏或虚拟世界后面的服务器是一些粗制滥造的代码，也不是说它们做得不好。实际上，许多代码是效率杰作，展示了聪明的编程技巧，也展示了针对高要求的应用构造一次性、专门目的服务器的优势。但是，为每个游戏或虚拟世界构建一个新服务器的习惯意味着人们不太注意积累构建这种服务器所需的知识，也没有共同接受的机制来比较不同的基础设施。

3.4.1 并行与延迟

缺少有关服务器可接受的性能的资料，这一点引起了Darkstar团队的特别关注，因为我们所做的一些关键决定，都是围绕着如何能够从游戏或虚拟世界服务器中获得好的性能。也许Darkstar架构和一般实践之间的最大区别就在于，Darkstar架构拒绝在服务器的主内存中存放任何重要的信息。所有生存周期超过一次具体任务的数据都需要持久在“数据服务”中，这是实现Darkstar基础设施功能的核心。这让基础设施能够检测到并发问题，反过来又让系统能够对程序员隐藏这些问题，同时让服务器能够利用多核架构。它也是实现整体伸缩性的关键组件，支持任务从一台服务器移动到另一台服务器，从而在一组机器上实现负载平衡。

在游戏和虚拟世界服务器领域，任何时候都持久保存游戏状态是一种异端邪说，因为人们普遍很关心延迟。在编写这种服务器时，大家的观点是只有将所有信息都放在内存中才能让延迟足够小，达到要求的响应时间。可以偶尔保存状态的快照，但对交互速度的要求表明，这种长时间的操作只能够偶尔进行，而且要在后台进行。所以，从表面上看，我们的架构似乎绝不可能达到足够好的性能，从而服务于它的目标应用。

虽然要求数据持久肯定是这个架构的主要不同之处，而且要求通过“数据服务”来访问数据会在架构中引入一定的延迟，但我们相信我们所采取的方式更具有竞争力，原因有几点。首先，我们相信能够让访问内存数据和访问“数据服务”中的数据之间的差异远远小于一般人们的看法。虽然在概念上每个生命周期超出一次任务的对象都需要从持久存储中读出，并写入持久存储，但实现这种存储可以利用人们在数据库缓存和一致性方面多年的研究成果，从而减少因这种方式而导致的数据访问延迟。

如果我们能够将访问局限在一个特定服务器上的几组特定对象，就更是如此了。如果用到一组特定对象的那些任务都运行在一台服务器上，那么就可以利用该服务器的缓存，达到接近内存的对象读写速度（受到需要满足的持久性约束的影响）。我们可以识别任务属于哪些玩家或虚拟世界的哪些用户。这样，我们就可以利用基础设施中服务所接收到的数据访问和通信请求，来收集特定时刻游戏或虚拟世界中数据访问模式和通信模式的信息。有了这些信息，我们相信能够非常准确地估计哪些玩家应该与另一些玩家放在一起。因为我们可以根据需要将玩家移动到任何服务器上，所以能够根据观察到的运行

时行为，主动地将相关的玩家尽可能地放在同一台服务器上。这让我们能够运用数据库领域熟悉的标准缓存技术，尽量减少访问和保存持久信息的延迟。

这听起来非常像目前在大规模游戏和虚拟世界中为实现伸缩性而采用的地理区域分解技术。在这种技术中，服务器开发者将世界分解成一些区域，将它们指派给一些服务器，不同的区域就成为用户分区的机制。同一区域的玩家比不同区域的玩家进行交互的可能性更大，所以这种集中在一个服务器上的优势就体现出来了。不同之处在于，目前的地理区域分解是在游戏开发过程中进行的，被编入源代码，放到服务器上。而我们的位置集中基于运行时的信息，可以根据游戏中发生的实际玩法和交互模式来实现动态调整。这类似于编译时优化和运行时优化之间的区别。前一种方法试图针对程序所有可能的运行进行优化，而后一种方法试图针对当前的运行进行优化。

我们不相信我们能够消除内存访问和持久访问之间的差别，而且我们也不认为有必要这样做，最后让这种架构比使用内存的架构性能更好。要知道，通过让所有的数据持久，我们可以支持在服务器上使用多线程（从而也支持多核）。尽管我们不相信并发是完美的（即对于每个添加的核，我们都能充分利用），但我们确实相信在游戏和虚拟世界中可以使用大量的并行运算（初步的结果也证实了这种看法）。如果可使用的并行运算超过我们可能引入的延迟，那么游戏或虚拟世界的总体性能就会更好。

3.4.2 赌未来

我们对多核处理器中多线程的信心基本上是在赌处理器将来的发展方向。目前服务器的处理器提供2～32个核，我们相信将来的芯片设计将集中向更多的核发展，而不是让现有的核以更高的时钟频率运行。当我们在几年前开始这个项目时，这种赌博似乎比现在更具投机性。那时候，我们在做展示时常常说这种设计是一种“假定”的演练，说我们正尝试一种架构，如果芯片性能更好地支持多线程而不是单线程的时钟速度，这种架构将是可行的。这就是在研究实验室中进行这类项目的好处之一，可以接受设计方法中存在很高的风险，探索一个将来也许在商业上可行的领域。我们决定构建一个以多线程为中心的架构，与这个决定做出时相比，目前芯片设计的趋势让我们看得更清楚。（注3）

即使我们只能得到50%的完美并发，如果我们能把使用持久存储的延迟控制在使用内存的延迟的2～16倍，就能够在性能上持平。我们相信在并发方面以及减少访问持久状态和全内存方案之间的差异方面都可以做得更好。但是结果主要取决于构建于这个基础设施之上的应用的使用模式（我们曾提到，这一点很难发现）。

我们也不应认为减少延迟就是这个基础设施的唯一目标。通过将游戏或虚拟世界服务器端的对象全部保存在“数据服务”中，我们把因服务器失效而导致的数据丢失减到了最

注3：　再一次说明，在早期的设计决定中少许的运气是很重要的。

小。实际上，在大多数情况下，服务器失效时用户只会注意到延时有一点增加，因为任务（它们本身也是持久对象）从失效的服务器移到了另一台服务器上。没有数据会丢失。一些缓存机制可能导致丢失几秒钟的游戏成果，但即使是这样，也比在线游戏和虚拟世界目前使用的机制好得多，它们只是将偶尔进行内存快照作为主要的持久方式。在它们的基础设施中，如果服务器在不巧的时间崩溃，可能会造成数小时的游戏成果丢失。只要延迟是可以接受的，Darkstar所使用持久机制的可靠性更高，这对于在这个基础设施上构建系统的开发者和系统的用户来说都是优点。

3.4.3 简化程序员的工作

实际上，如果在支持伸缩性的同时减少延迟是服务器开发者的唯一目标，那么开发者最好的方法就是专门针对特定的游戏，编写自己的分布式和多线程基础设施。但这要求服务器开发者处理复杂的分布式和并发编程。在为速度需求而过度烦恼之前，我们应该想到Darkstar的第二个同样重要的目标，即在支持多线程、分布式游戏产品的同时，为程序员提供一个单机单线程的开发模型。

在相当大的程度上，我们已经实现了这一目标。通过将所有任务封装到事务中，并在“数据服务”中检测数据冲突，程序员就能够享受到多线程的好处，又不必在他们的代码中引入锁协议、同步和信号量。程序员不必担心如何将玩家从一台服务器移到另一台服务器，因为Darkstar为他们提供了透明的负载平衡。编程模型虽然有自己的风格和限制，但社区中的早期成员认为，这对他们开发的那种游戏和虚拟世界来说是比较自然的。

不幸的是，我们发现我们不能够向程序员隐藏所有的东西。当在Darkstar上编写的第一个游戏表现出极少的并行度（以及意料之外的糟糕性能）时，这一点就明确了。通过检查源代码，我们很快就发现了原因。游戏中的数据结构设计导致了游戏中所有的状态改变都只涉及一个对象，并由它来协调所有的工作。使用这个对象实际上使得游戏中所有动作序列化执行，这使得基础设施不能够发现并利用并发计算。

当我们发现这一点时，我们与游戏开发者进行了长时间的讨论，主题是在设计对象时需要考虑到并发访问。通过对游戏中数据对象进行审查，我们发现了一些类似的情况：数据设计方案排除了并发的可能性（并非出自设计者本意）。当这些对象重新设计之后，系统的整体性能增加了好几个数量级。

这告诉我们，不可能让使用Darkstar的开发者完全不知道系统底层的并发和分布式实质。但是，他们对系统这方面特点的知识不需要包括并发控制、锁，以及在系统的各个分布式部分之间的通信。实际上，他们只需要在设计活动中确保他们的数据对象定义能够充分利用并发。这种设计一般只需要确保对象定义是自包含的，它们的操作不需要依赖其他对象的属性。这一点对于任何系统来说，都不是不好的设计原则。

关于Darkstar架构，我们还有许多方面没有测试，或者说我们并未完全理解。虽然我们已经得到了一个系统，使得多台机器能够利用多线程运行一个游戏或虚拟世界，同时对服务器程序员（几乎）保持透明，但是我们还没有通过添加核心服务之外的其他服务，来检验该架构的能力。由于Darkstar任务的事务本质，这可能比我们开始设想的要复杂得多，但我们希望这些添加的服务不需要参与到核心服务的事务中。我们已经开始试验通过不同的方式来收集系统负载的信息和实现负载平衡。幸运的是，因为实现这种负载平衡的机制对于使用系统的程序员是完全不可见的，所以我们可以移除老的方式，引入新的方式，同时又不影响Darkstar的用户。

作为一个架构，Darkstar展示了一些创新的方法，这使它变得很有趣。它试图构造一个游戏或虚拟世界的基础设施，使其具有企业级软件一样的可靠性，同时又满足游戏行业对延迟、通信和伸缩性的要求。它是目前为数不多的这类尝试之一。通过利用更多机器和更多线程来实现效率，我们希望能够抵消因使用持久存储机制而导致的延迟增加。最后，游戏和虚拟世界环境中极为不同的情况，即客户端的处理很多而服务器端的处理很少，与我们常见的高并发、分布式系统环境形成了鲜明的对比。现在说这个架构是否成功还为时尚早，但我们相信它已经很有趣了。

	原则与特性		结构
	功能多样性	√	模块
√	概念完整性	√	依赖关系
√	修改独立性		进程
√	自动传播		数据访问
√	可构建性		
	增长适应性		
	熵增抵抗力		

第4章

记忆留存

Michael Nygard

从最早的锡版照相法和达盖尔银板照相法开始，我们一直都认为照片是具有特别意义的，有时候甚至是神奇的。照片记录了一闪而过的时刻，而我们的记忆容易出错，不能做到这一点。但是最好的肖像不仅保存了一个时刻，还阐释了这个时刻，捕捉了某个眼神或表情，捕捉了有特点的姿势，让主角靓丽照人。

如果你的小孩在美国的学校上学，你可能已经熟悉Lifetouch这个名字。Lifetouch每年都为美国几乎所有的小学生、初中生和高中生照相。但是你可能不知道，Lifetouch还经营着高品质的肖像照相馆。Lifetouch肖像照相馆（LPS）开设在美国各大零售商店中，还在大超市中开设了“Flash!”连锁照相馆。在这些照相馆中，LPS的摄像师拍摄那些可以保留一生的照片。

数字摄影改变了整个摄影行业，LPS也不例外。大圈的胶片和上底片框的照相机不见了，取而代之的是专业级的DSLR和闪存卡。自由摄影师可以四处走动，尝试不同角度，比以往任何时候更接近所拍摄对象。简而言之，它们更自由地拍摄这些了不起的肖像。摄影师利用照相机将光子转变为电子，但通过某种方式，在某些地方，某些系统必须将这些电子转变成墨水原子和纸张。

2005年，我和来自Minneapolis的ATI（Advanced Technologies Integration）的同事，与LPS的开发者一起工作，开发了一个新系统来完成这部分工作。

4.1 功能和约束

这两点影响了系统的架构：它必须做什么？它必须在什么限制条件下工作？这两点也确定了问题空间。

通过解决这些重要问题，在要求的行为和限制条件之间进行探索，我们创造并考察解决方案空间。如果对各项限制条件的解决方案形成了一致的整体，我们就可能创造出一个优雅的、美丽的解决方案。我很高兴地说，Creation Center项目做到了这一点。

在这个项目中，我们面对了一些明确的事实。有些只是业务的特点，另一些会改变，但不在我们考虑的范围内。不管怎样，我们认为这些事实是不会变的。这些事实构成了图4-1中的左边一列。

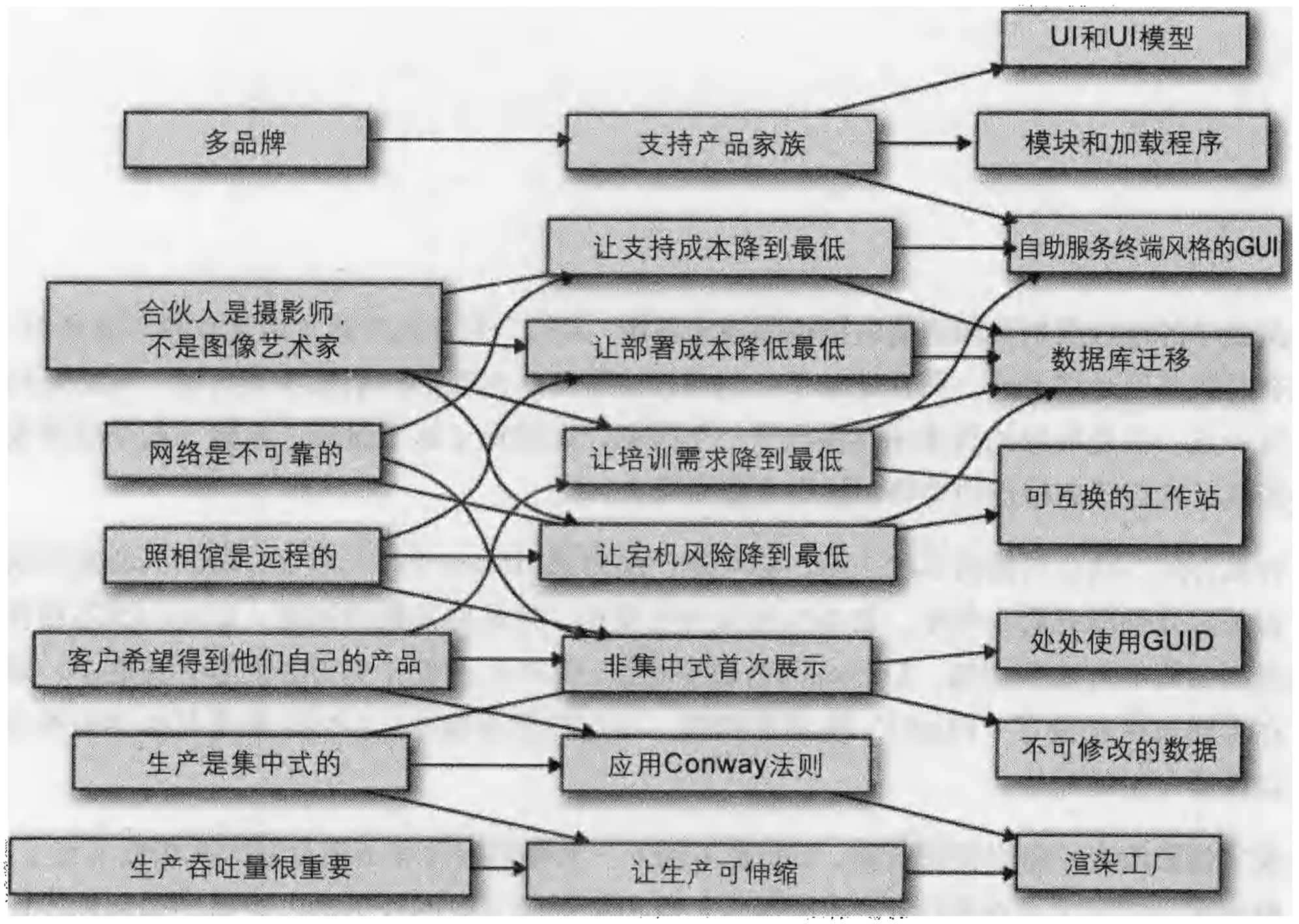


图4-1：事实、重要问题和Creation Center架构的关注点

多品牌

LPS今天支持多个品牌，将来可能加入更多品牌。至少，Creation Center会有两个视觉上截然不同的界面主题，而且添加界面主题应该不需要大量的工作。

合伙人是摄影师，不是图像艺术家

摄影师接受过如何使用照相机的培训，但没有接受过使用Photoshop的培训。当一个没有经验的用户使用Photoshop时，最可能的结果是得到糟糕的图片。对于强大的用户来说，Photoshop是强大的工具，照相馆的摄影师应该没有必要去体验Photoshop的学习曲线。Photoshop和类似的工具会让照相馆的工作流程变慢。照相馆的合伙人需要快速创造出漂亮的图像。

照相馆是远程的

照相馆在地理上是分布式的，很少或没有本地技术支持。硬件交付或替换需要来回运送部件。

网络是不可靠的

某些照相馆没有网络连接。即便对于有网络连接的照相馆，因断网而导致照相馆停工也是不可接受的。

客户希望得到他们自己的产品

客户收到的照片应该包括他们自己的设计和要求的文字。

生产是集中式的

高品质的照片打印机正变得越来越常见，但制作能够保存几十年的产品需要贵得多的设备。

生产吞吐量很重要

使用到相同的打印机也是生产过程中的约束条件。因此，过程中的其他步骤必须服从于这个约束条件。

这些事实导致了一些重要问题，我们必须权衡考虑。人们常常将这些重要问题视为最基本的，但它们不是的。实际上，它们源自于系统存在的环境。如果环境改变，这些重要因素会无效，甚至走向反面。

我们选择了一些结构来解决这些重要问题。图4-1中右边的一列展示了架构的这些关注点。当然，这些并不是Creation Center的全部值得讨论的特性，但架构中的这些考虑是普遍关注的。我也认为它们同时展示了很好的关注点分离和相互支持的结构。

在深入到具体的特性之前，我们需要补充介绍一点背景知识：系统的工作流。

4.2 工作流

典型的照相馆有2~4个摄影室，备有专业的灯光、背景幕布和支架。摄影师在摄影室中

拍照（每张照片被称为一个“姿势”）。在摄影室外面，摄影室也提供客户服务、安排计划，并让客户挑选照片。

当摄影师完成了一个阶段的拍照之后，她会坐到一台工作站前面，从照相机的存储卡中导入照片。

导入了一批照片后，摄影师会删除那些明显不好的照片，包括闭眼的、不高兴表情的、婴儿没有看镜头等照片。删除了这些不好的照片之后，剩下的照片就是“基本照片”。接下来她会从这些基本照片中选择一些进行增强。增强可以是简单色调调整，如变成黑白或深褐色，也可以是精心组合多张照片。例如，摄影师可能为3个小孩拍摄一张合影，然后将照片嵌入设计好的3个“卡槽”中，得到他们的个人照。

在完成了这些增强后，摄影师帮助客户订购打印的各种尺寸和组合。包括8" × 10"这样的大照片，或者5" × 7"、3" × 5"，以及适合放在钱包中的小照片。还有更大的尺寸。客户最大可以订购24" × 30"的照片，配上镜框后挂在墙上。

在完成了客户订购后，摄影师就转向下一组照片的拍摄。

每天工作结束时，照相馆经理会制作一张DVD，包含当天所有的订购照片，并将它送到打印工厂。

打印工厂每天会收到几百张DVD。（稍后我会介绍这些DVD的内容。）这些DVD包含了订单和需要打印的照片，照片打印好后会送回照相馆，这样客户就可以取走自己的照片。但在照片打印之前，最终打印分辨率的照片必须渲染成图片。这些可以打印的图片是很大的。如果把24" × 30"的肖像渲染成高品质的图片，会超过1亿像素，每个像素都由32位的颜色来表示。所有像素都是根据摄影师在照相馆中的设计来合成的。根据不同的合成方式，渲染的过程可能包含6~10个步骤。简单的演染需要2~5分钟，但大照片的复杂渲染需要超过10分钟的时间。

同时，打印机每分钟都会输出一些完成打印的照片。让打印机保持工作是“生产控制系统”（PCS）的职责，这是一个复杂的系统，它负责处理任务调度、协调渲染工厂、管理图像存储，并将图像提供给打印队列。

当完成的订单送达照相馆时，经理会告知客户可以过来取照片了。

这个工作流部分来自于LPS的业务背景，部分来自于我们对于如何划分系统的考虑。接下来让我们来看看图4-1中不同的关注点。

4.3 架构关注点

将一个多维的、动态的系统简化为一种线性的描述形式总是一项挑战，不论我们是在谈

论对还不存在的系统的愿景，还是试图解释已经构建好的系统中各个部分的交互。超链接让我们可以从多个角度来处理这个复杂问题，也许会让事情变得容易一些，但是纸张对超链接的支持还不是很好。

当我们研究每一个架构关注点时，要记住它们是研究整个系统的不同方式。例如，我们使用模块化的架构来支持不同的部署场景。同时，每个模块又是按分层架构来创建的。它们是正交的、相互穿插的考虑。每个模块都按照相同的方式来分层，每个层都可以在所有模块中找到。

实际上，我们感到非常满意，因为我们既能够保持这些关注点的分离，同时又让它们互相支持。

4.3.1 模块和加载程序

一直以来，我们都在考虑“产品家族”而不是单个“应用程序”，因为我们必须支持几种不同的部署场景，而这些场景使用同样的底层代码。特别是，我们从一开始就知道会有以下几种配置：

照相馆客户端

照相馆有2~4台这样的工作站。摄影师使用它们来参与整个工作流，从加载图像到创建订单。

照相馆服务器

每个照相馆的中心服务器上运行着MySQL，保存客户和订单等结构化数据。服务器的存储也比工作站更健壮，使用了RAID来实现可恢复性。我们也利用照相馆服务器将每天的订单刻录到DVD中。

渲染引擎

在生产过程中，我们决定构建自己的渲染引擎。渲染照相馆屏幕和准备打印的图像使用的是相同的代码，所以我们绝对可以肯定客户会拿到他们期望的照片。

最初，我们认为这些不同的部署配置只是一些不同的.jar文件集。我们创建了一些顶层目录来保存每种部署的代码，还有一个Common目录。每个顶层目录都有自己的source、test和bin目录。

没过多久，这种结构就让我们遭遇了挫折。例如，我们有一个庞大的/lib目录，里面积累了许多构建时的库和运行时的库。我们也苦于寻找地方存放那些非代码文件，如图像、颜色描述、Hibernate配置文件、测试图像等。我们中的一些人也觉得必须通过手工的方式来维护.jar文件的依赖关系是不舒服的事情。在刚开始那些日子里，我们常常发现很

多包放在了错误的目录中。而在运行时，有些类又会加载失败，因为它依赖于另一个.jar文件中的类。

大约在项目的第3次迭代时我们引入了Spring（注1），这时转折点出现了。我们采用“敏捷架构”的方式：保持最小的架构，只有当避免新架构特性的代价超过实现它的代价时，才采用新的架构。这在“精益软件开发”中称为“最后负责时刻”。在此之前，我们对Spring只有一些不全面的了解，所以我们决定不依赖它，但是我们都预计后来会用到它。

当我们加入Spring时，.jar文件的依赖问题被配置文件问题放大了。每个部署配置都需要它自己的beans.xml，但大约超过一半的beans会在这些文件中重复出现，这明显违反了“不要重复你自己”（注2）的原则，而且是缺陷的必然来源。所有人都不应该手工同步几千行XML文件中的bean定义。此外，几千行的XML文件本身难道不是一种坏味道吗？

我们需要一个解决方案，让我们能够模块化Spring的beans文件、管理.jar文件的依赖关系、使库贴近使用它们的代码，并管理构建时和运行时的classpath。

ApplicationContext

学习Spring就像探索一片广阔的、不熟悉的地域。它是框架中的NetHack游戏，他们考虑了所有的事情。浏览javadoc文档常常有巨大的收获，在这个项目中，当我无意中看到“应用上下文”类时，我们发现了非常有价值的东西。

所有Spring应用程序的核心都是“bean工厂”（bean factory）。bean工厂支持按名字查找对象，根据需要创建对象，并向其他bean注入配置和引用。简而言之，它管理了一些Java对象和它们的配置。最常用的bean工厂实现会读取XML文件。

应用上下文扩展了bean工厂，提供了一项关键的功能：创建嵌套的上下文链，就像《Design Patterns》（设计模式）（Gamma等 1994）一书中描述的“职责链”模式。

ApplicationContext对象正是我们所需要的：将我们的所有bean分开，放到多个文件中，将每个文件装入各自的应用上下文。

然后我们需要利用一种方法来建立应用上下文链，最好是不用巨大的shell脚本。

模块依赖关系

将每个顶层目录看成一个模块，我认为让每个模块包含它自己的元数据是很自然的。通过这种方式，模块只需要声明它用到的classpath和配置文件，并声明它需要哪些其他模块。

注1：*http://www.springframework.org/*.

注2：参见Andrew Hunt和David Thomas的《Pragmatic Programmer》(Addison-Wesley Professional)。

我为每个模块提供了自己的manifest文件。例如，下面是StudioClient模块的manifest文件：

```
Required-Components: Common StudioCommon
Class-Path: bin/classes/ lib/StudioClient.jar
Spring-Config: config/beans.xml config/screens.xml config/forms.xml
        config/navigation.xml
Purpose: Selling station. Workflow. User Interface. Load images. Burn DVDs.
```

这个格式显然是源自于.jar文件中的manifest文件。我发现，使用熟悉格式来表示概念上的“manifest文件”是很有用的。

请注意，这个模块使用了4个独立的bean文件。按功能分离bean的定义有额外的好处。它减少了对主配置文件的修改和冲突，而且提供了很好的关注点分离。

我们的团队很喜欢自动生成文档，所以我们在构建过程中包含了一些报告生成步骤。由于所有模块依赖关系明确写入了manifest文件，所以在自动化构建中添加报告生成步骤是很容易的。只需要进行一些文本解析，然后提供给Graphviz，生成图4-2中的依赖关系图。

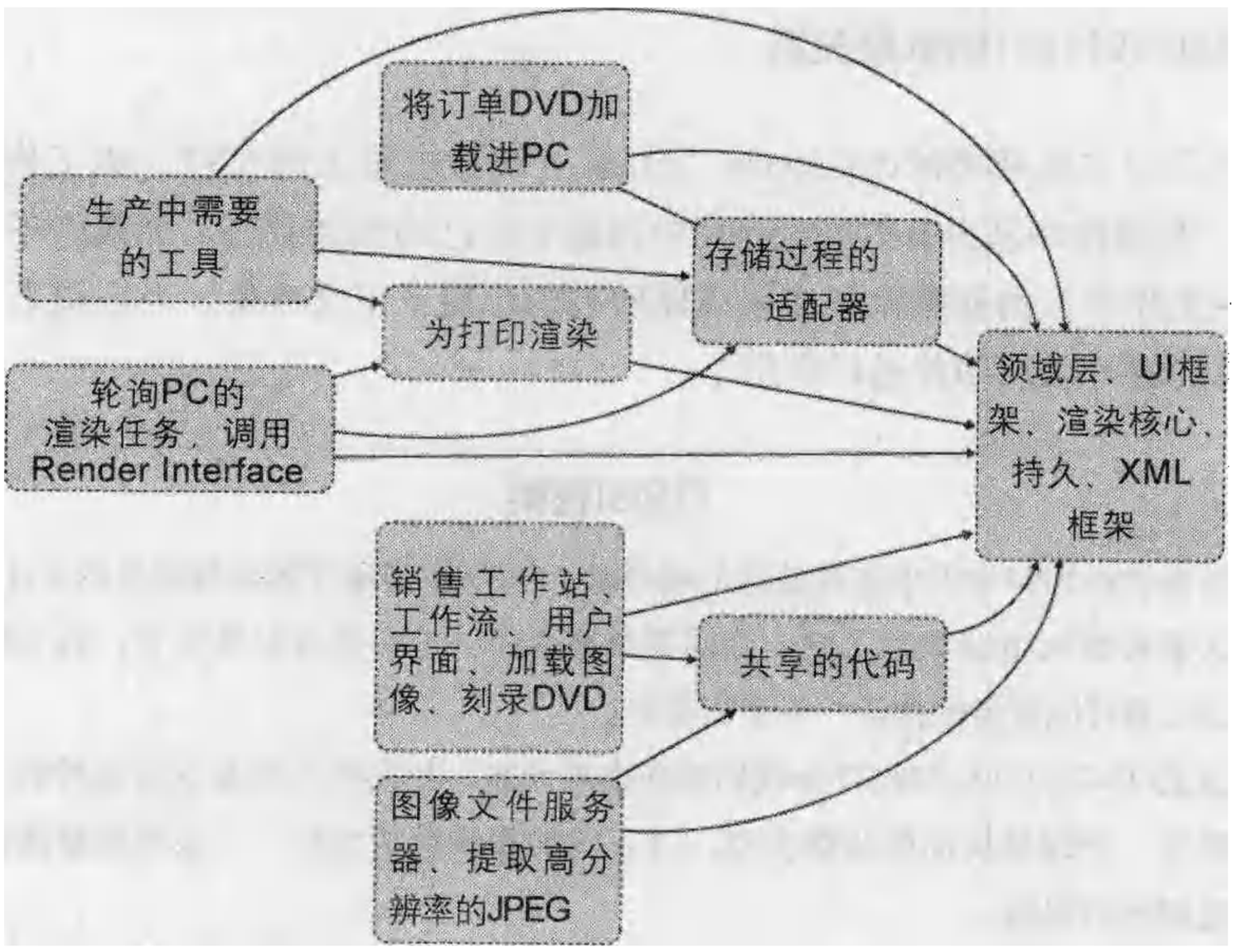


图4-2：模块和依赖关系

有了这些manifest文件，我们只需要通过一种方法来解析它们，然后做一些有用的事。我写了一个加载程序，你可以猜到它就叫“加载程序”（Launcher），来做这些事情。

加载程序

我看到过许多桌面Java应用，它们带有大量的shell脚本或批处理脚本来定位JRE，设置环境变量，设置classpath等。天啊！

对于给定的模块名，加载程序将解析manifest文件，构建模块依赖关系的传递闭包。加载程序很小心，不会将一个模块加载2次，同时它也将一组偏序关系变成全序关系。图4-3展示了StudioClient的全部依赖关系。StudioClient同时将StudioCommon和Common声明为依赖包，但加载程序只会将每个包加载一次。

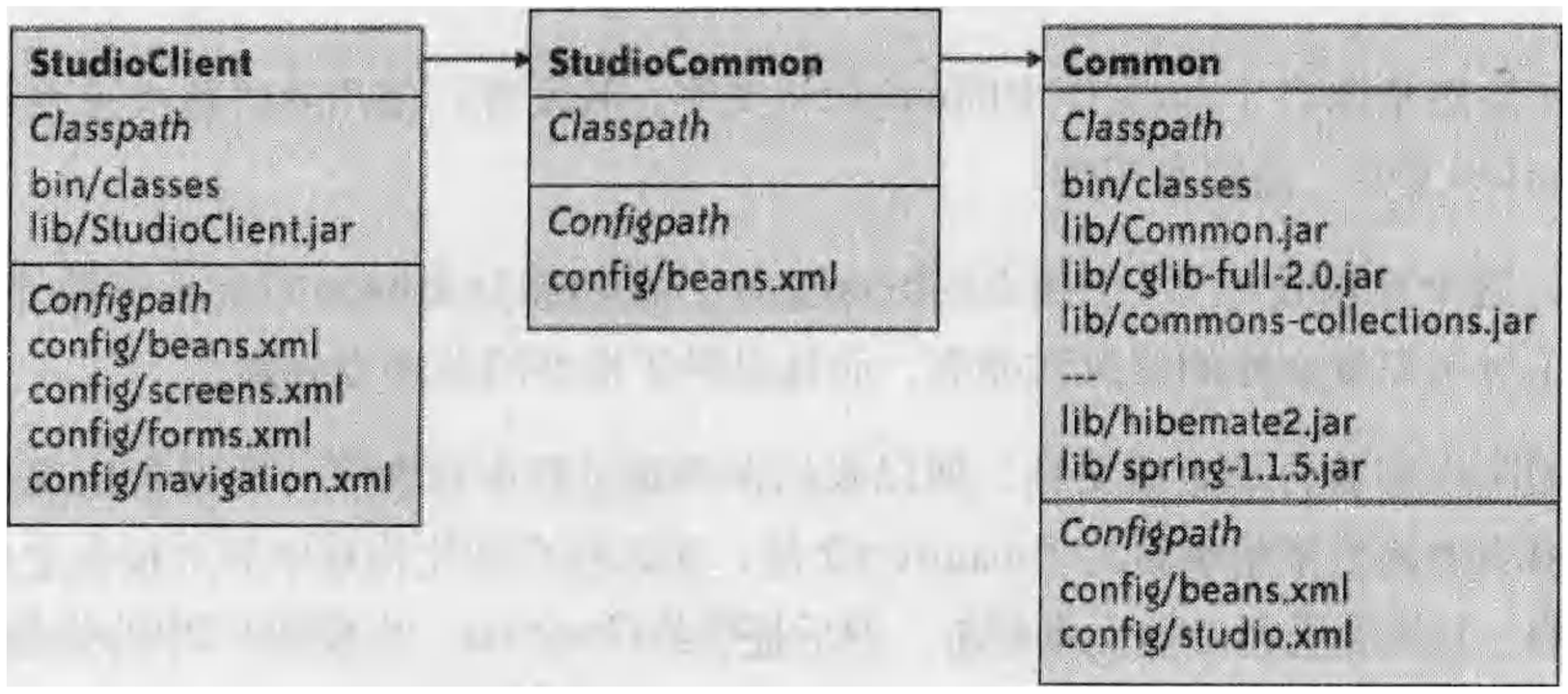


图4-3：StudioClient的依赖关系

为了避免来自于主机环境的classpath“污染”（构建机器上的ANT，或工作站上的JRE classpath），加载程序从组合的classpath中创建它自己的类加载器。加载程序将配置路径传递给初始化程序，由初始化程序来创建所有的应用上下文对象。当应用上下文创建好了之后，我们就可以启动并运行应用了。

OSGi框架

当我们在2004年下半年开始这个项目时，OSGi框架刚开始得到较多的关注，这要感谢Eclipse采用了它。我们简单地看了一下，但没有采用它，因为缺少大量可以获得的知识、专家和指导。

但是OSGi的目标却很符合我们所面临的问题。相同的代码集支持多种部署配置，管理模块间的依赖关系，以正确的顺序激活它们……很明显解决的是相同的问题。

我想我们之所以没有使用OSGi，部分原因是因为时间紧迫，部分原因是因为我们不愿意采用技术风险更大的东西。我一般会赞成“获取并集成”，而不是“自己来搞一套”，但似乎有一个盈亏平衡点：很少支持的开源项目和较弱的社区与众所周知的、广泛采用的开源项目相比，具有更大的风险。类似地，我倾向于避免准开源框架，它们实际上是供应商公会。他们服务的社区通常是供应商社区，而不是用户社区。

那时我们不清楚OSGi属于哪一种类型。如果我们今天来做这个项目，我想我们可能会使用OSGi，而不是自己搞一套。

在这个项目的过程中，我们数次重构了模块结构。manifest文件和加载程序提供了巨大的帮助，让我们只需进行小小的改变。我们最后得到了6个非常不同的部署配置，它们都由同样的结构来支持。

这些模块都有着类似的结构，但它们不一定是完全一样的。这是这种方法的副作用之一。每个模块都可以隐藏其他模块不关心的部分。

4.3.2 自助服务终端风格的GUI

照相馆合伙人选择的标准是能够很好地使用照相机，并与家庭（特别是孩子）合作，而不是具有很好的计算机技能。在家里，他们可能是Photoshop专家，但在照相馆里，没有人希望他们成为强大的用户。实际上，在忙的季节，照相馆可能会请一些临时工。因此，能够快速上手是很重要的。

一名架构师也承担UI设计师的工作。他总是对界面有着清晰的愿景，即使我们并不总是对实现它的可行性持一致意见。他希望用户界面是友好的、可视的。不会有菜单。用户应该通过直接操作与图像打交道。大的像糖果包装一样的按钮让所有的选项一目了然。简而言之，工作站看起来应该像一个自助服务终端。

剩下来的决定就是选用哪种技术来实现显示。

我们团队中的一个人调查了可用的Java富UI技术，包括主流技术和边缘技术。我们希望找到一个好的声明式UI框架，能够帮助我们避免在与Swing的搏斗中无尽地艰难前行。结果让我们很震惊。

在2005年，即使是Java诞生十年之后，也只有两种基本选择占据了主流：XML地狱或GUI构建工具生成的混乱代码。基于XML的各种技术或多或少地直接将Swing组件映射为XML实体和属性。这对我们没有什么意义。GUI改变要求发布新版本的代码，不论这种改变是直接用Java代码实现的还是用XML文件实现的。为什么要在脑子里记住两种语言（Java和XML schema）？为什么不只使用Java？另外，作为一种编程语言，XML显得很笨拙。

GUI构建工具以前曾对我们都造成过伤害。没有人想将业务逻辑编织到JPanel中嵌入的动作监听者中。

我们很不情愿地采用了纯Swing GUI，但是使用了一些基本原则。经过了几次在本地Applebee的午餐之后，我们想出了一种使用Swing的创新方式，同时又不会陷入其中。

4.3.3 UI和UI模型

典型的分层架构包含“表示层”、“业务领域层”和“持久层”。在实践中，代码的重心集中到了表示层，业务领域层变成了简单的数据容器，持久层则转变成对框架的调用。

但在同时，一些重要的信息在各层之间重复出现。例如，姓名的最大长度会体现为数据库中列的宽度，可能也是业务领域中的一项验证规则，还是UI中JTextField的一项属性设置。

同时，表示层嵌入了这样的逻辑：如果选中了这个复选框，那就让这4个文本字段也生效。这看起来是关于UI的论述，但它实际上反映的是一些业务逻辑：如果客户是“肖像俱乐部”的会员，应用需要记录下他们的会员编号和到期时间。

所以在典型的三层架构中，一种类型的信息会散布在各层中，而另一种类型的重要信息存在于GUI控制逻辑中。

最后，我们的解决方案是将GUI与业务领域层的正常关系颠倒过来。我们让业务领域层负责将屏幕的可视外观与它的值和属性的逻辑操作分离开来。

表格

在这个模型中，表格对象将一个或多个领域对象的属性表示成有类型的属性。表格管理着领域对象的生命周期，也负责调用底层事务和持久的外观方法。每个表格代表了一整屏的交互对象，虽然在少数情况下我们使用了子表格。

但这里的技巧在于，表格是完全不可视的。它不处理UI组件，只包含对象、属性以及这些属性之间的交互。UI可以将一个布尔属性绑定到任何类型的UI表示和控制组件上：复选框、标记按钮、文本框或标记开关。表格并不关心表示。它只知道它有一个属性，取值可以是true或false。

表格从不直接调用屏幕对象。实际上，绝大多数表格甚至不知道它们屏幕上的具体类。表格和屏幕间的所有通信都通过属性和绑定来完成。

属性

不像一般的基于表格的应用，表格暴露出来的属性不只是Java的原生类型或像Java.lang.Integer这样的基本类型。属性包含了一个值和这个值的元数据。属性可以回答它是单值的还是多值的，它是否允许空值，它是否是启用的。它也允许注册监听者对象来监听它的变化。

表格及其属性对象为我们提供了用户界面的清晰模型，同时我们又不必处理实际的GUI组件。我们把这一层称为“UI模型”层，如图4-4所示。

属性的每个子类处理一种不同类型的值。具体的子类有自己的方法来访问其值。例如，StringProperty有getStringValue()和setStringValue(String)方法。属性值总是对象类型，而不是Java原生类型，因为原生类型不允许出现空值。

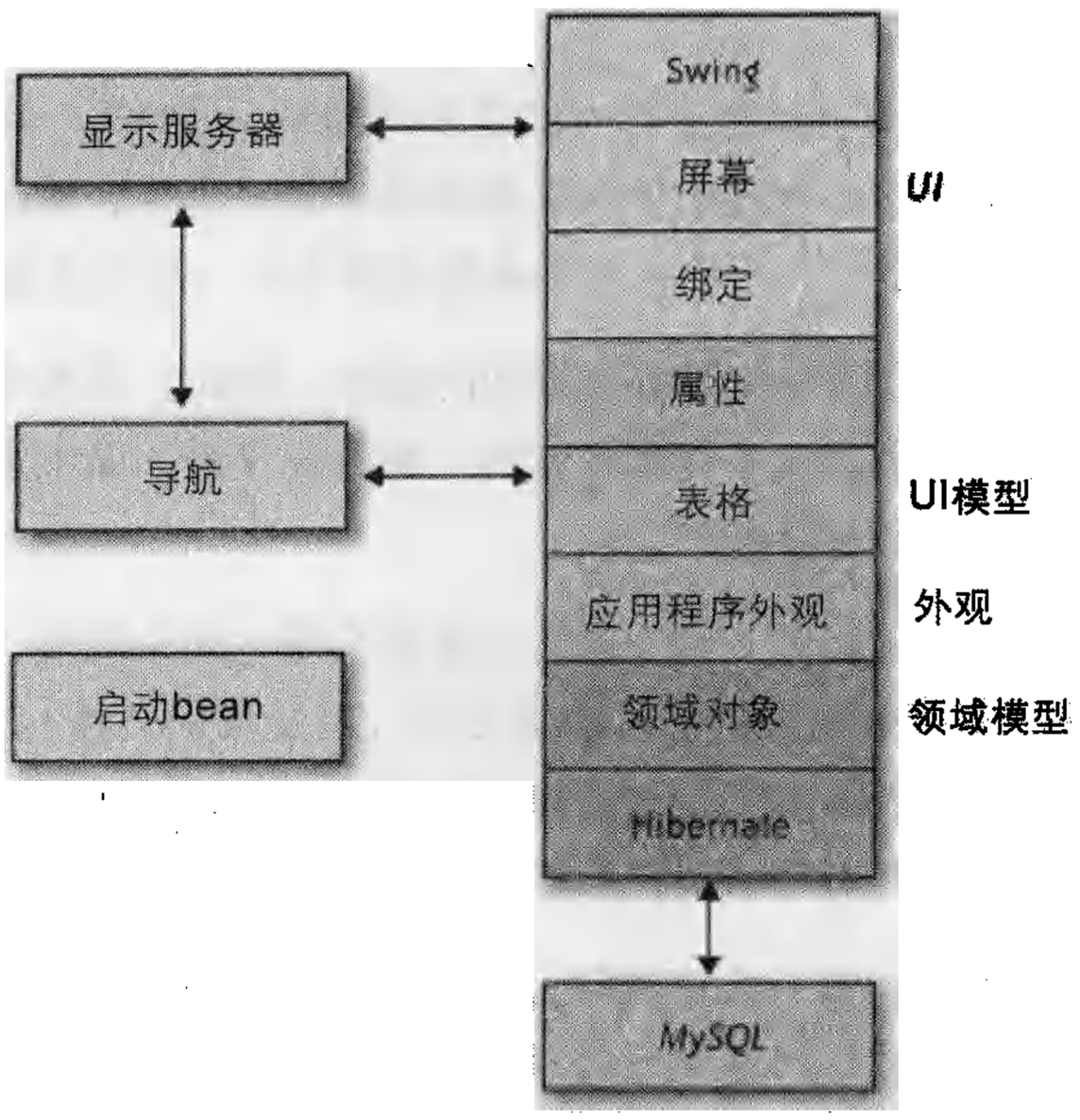


图4-4：分层架构

属性类似乎可以无休止地膨胀下去。如果我们为每个领域对象类创建一个属性类，肯定会出现这种情况。多数时候，表格不会直接暴露出领域对象，而是暴露出代表该领域对象不同方面的多个属性值。例如，客户表格暴露出StringProperty对象来表示客户的姓、名、地址、城市和邮编。它也暴露出一个DateProperty来表示客户会员资格的到期时间。

某些领域对象在应用这种方法时会让人觉得很奇怪。连接一个滚动条来控制图像的缩放，或将图案中的图像嵌入到底层的几何图形上可能需要超过半打的属性。让表格具有这么多属性，目的只是为了拖动一个滚动条，这明显是不好的代码味道。另一方面，添加另一种属性类型似乎又在朝着类型膨胀的方向前进。

最后，我们进行了折中，引入了一个对象属性来保存任意类型的Java对象。关于这个类的讨论意见包括“一步错，步步错”和“垃圾场”。幸运的是，我们对这种类进行了检查——我觉得这是类型检查语言的一种危险。

我们创建了一种“命令属性”来处理动作，它封装了一些命令对象，也表明功能启用。因此，我们可以将命令属性对象绑定到GUI按钮，利用属性的启用值的变化来启用或禁用按钮。

UI模型让我们能够将Swing保持在UI层之内。它也为单元测试提供了巨大的好处。我们的单元测试可以通过UI模型的属性来驱动UI模型，并断言这些动作所导致的属性改变。

所以，表格本身是不可视的，但它们暴露出命名的、强类型的属性。在某个地方，这些属性必须与可视控件联系起来。这就是绑定层的工作。

绑定

属性针对的是它们的值的类型，而绑定针对的是单个的Swing组件。屏幕创建它们自己的组件，然后注册一些绑定，将这些组件和底层表格对象的属性联系起来。每个屏幕并不知道它所处理的具体表格类型，表格也不知道与它联系在一起的具体屏幕类型。

绝大多数的绑定都会在每次GUI变化时更新它们的属性。例如，文本字段会在每次击键时更新。我们使用立即验证的方式来提供经常的、细致的反馈，而不是让用户输入一串不正确的数据之后再弹出一个对话框。

绑定也负责处理从属性的对象类型到组件的合理视觉呈现之间的转换。所以，文本字段绑定知道如何将整型、布尔型和日期型转换成文本（以及反向转换）。但并不是所有的绑定都能够处理所有的值类型。例如，把图像属性转换成文本字段是没有什么意义的。我们确保所有不匹配的情形在应用启动时都能发现。

当我们构建了这个属性绑定的第一次迭代时，发生了一个有趣的插曲。我们尝试的第一个屏幕是客户注册表格。客户注册是相当简单的，只是一些文本字段、一个复选框和几个按钮。第二个屏幕是相册屏幕，在视觉和交互方面的要求更高一些。它使用了多个GUI组件：2个验证表单、一个大图像编辑器、一个滚动条和几个命令按钮。即使这样，表格也负责通过属性来决定所有的选择、可见性和功能启用。所以相册表格对象知道验证表单的选择会影响中央的图像编辑器，但屏幕对象是不知道的。

一个够了吗？

在某些屏幕上，验证表单允许多项选择；在另一些屏幕上，只允许一项选择。更糟糕的是，某些动作只有在选择某一个缩略图时才允许执行。哪个组件负责决定哪个选择模型适用、何时根据选择来启用其他命令？这明显是UI逻辑，所以它属于UI模型层。也就是说，它属于表格。UI模型永远不应该引入Swing类，那么表格如何能够在不涉及Swing代码表达它们关于选择模型的意见呢？

我们决定，没有理由让GUI组件只有一个绑定。换言之，我们可以对组件的某一个方面做多个绑定，这些绑定可以关联到不同的表单属性。

例如，我们常常有不同的绑定，代表组件的内容和它的选择状态。选择绑定可以将组件配置为单项选择和多项选择，这由绑定属性的数目来决定。

虽然花了很长时间来解释属性-绑定架构，我还是认为这是Creation Center最优雅的部分之一。从本质上来说，Creation Center是高度可视化的，带有丰富的用户交互。它的领域是创建和操作照片，所以它不是灰色的、基于表格的业务应用！然而，从一小组容易理解的对象出发，让每个对象都由单一的行为定义，我们组建了非常动态的界面。

客户应用最后支持了拖放、图像中的部分选择、随时图像缩放、主从列表、表格和双击激活等功能。我们从未破坏属性-绑定架构。

应用程序外观

创建强大的领域模型容易犯一个典型的错误。表示层（在这个例子中就是UI模型层）常常与领域模型太过亲密。如果表示层遍历了领域模型中的关系，那么就很难改变领域模型了。像大多数敏捷团队一样，我们需要保持灵活，我们不能允许自己的设计决定导致灵活性随时间的推移而降低。

Martin Fowler的“应用程序外观”模式正适合解决这个问题（参见这一章末尾的第4.6节）。应用程序外观只将领域模型的一部分暴露给表示层。表示层不会跟踪领域对象的关系图，而是让应用程序外观来帮助实现遍历、生命周期管理和激活等操作。

每个表格定义了一个对应的外观接口。实际上，按照应该由客户（而不是提供者）来确定接口的格言，我们将这个外观接口放到了表格的包中。表格要求外观来查询领域对象、建立领域对象之间的关系并将它们持久。实际上，这些外观管理着所有的数据库事务，所以表格从不知道事务边界。

在表格对象和外观对象之间的这些接口，也成为隔离对象、进行单元测试的理想场所。要测试某个表格对象，单元测试会创建一个模拟对象（mock object）来实现外观的接口。测试让模拟对象向表格对象提供一组预期的结果，包括很难通过真正的外观对象再现的错误情况。我想我们都把模拟对象看成是一种双面承诺：尽管我们让单元测试变得可行，但有时候仍然觉得让测试和表格的实现绑得这么紧是不对的。例如，模拟对象必须事先录制预期的方法调用顺序和准确的参数。（新的模拟对象框架更灵活一些。）结果，即使外部可见的行为没有改变，表格内部结构的变化常常也会导致测试失败。在某种程度上来说，这只是你使用模拟对象的代价。

所有的Creation Center应用程序，包括在照相馆和打印工厂中使用的，都使用相同的层。将GUI从驾驶室中挪开让团队不必与Swing进行无休止的搏斗。反向控制也提供了一种统一的结构，每个应用程序，每一对程序员都遵守它。虽然我们创造的架构超出了通常的“三层蛋糕”，但它在分离关注点方面相当有效：Swing限制在UI部分，领域交互在表格中完成，持久在应用外观中实现。

4.3.4 可以互换的工作站

当摄影师完成一段拍摄之后，她可以使用任意一台工作站。她通常会当时就完成这个客户的工作，这取决于照相馆当时的忙闲状况。但是，客户常常也会稍后再回来，甚至可能改天再来。让某个用户只能使用某台工作站是很奇怪的，这不仅在安排上行不通，而且也有风险。工作站可能失效。

所以照相馆中的所有工作站必须能够互换，但是“可互换”带来了一些问题。每次拍摄的图像可能占用将近1GB的空间。

我们简单考虑了一下将工作站连成一个P2P网络，进行分布式复制。最后，我们采用了更为传统的客户端–服务器模式，如图4-5所示。

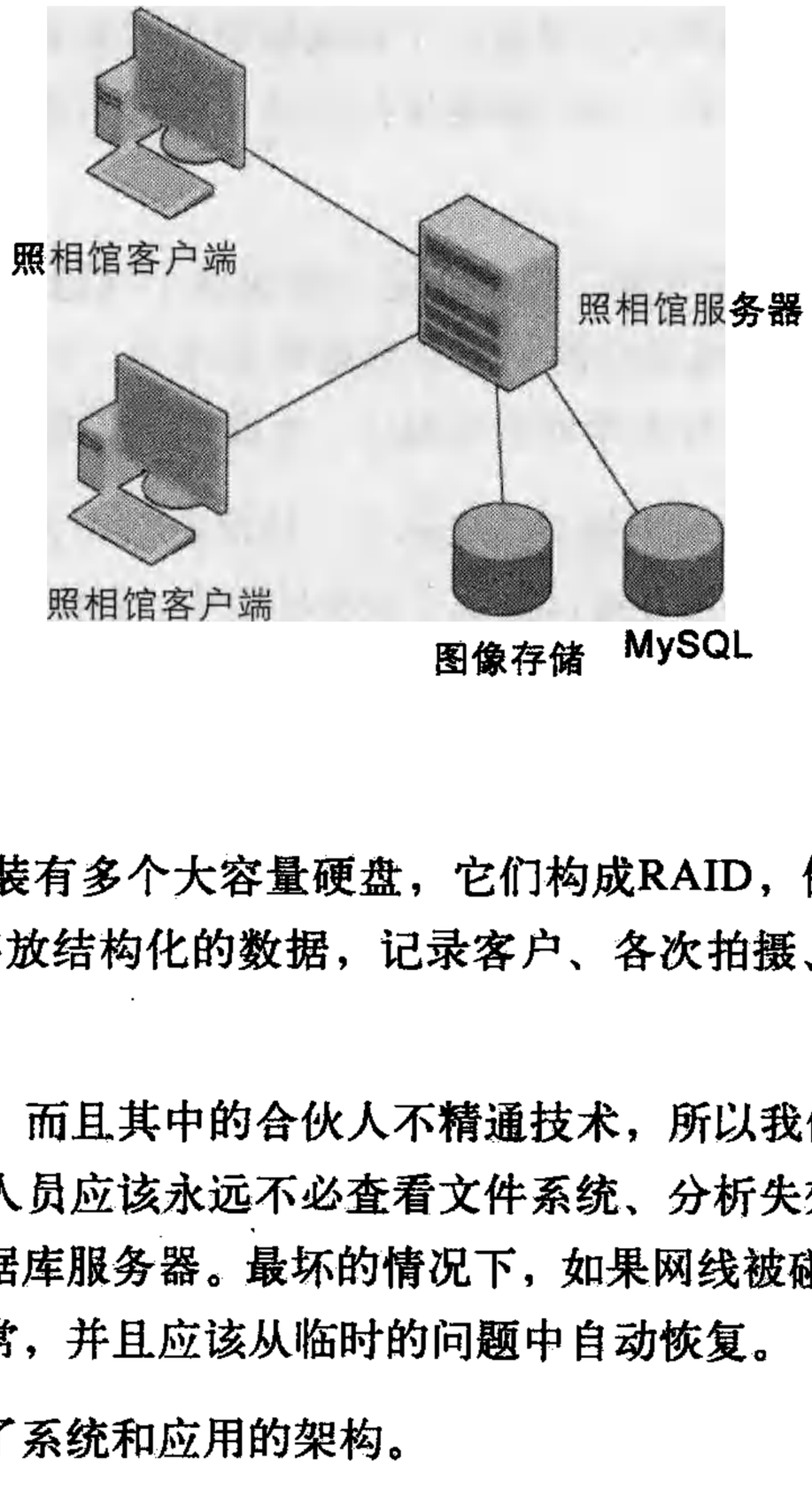


图4-5：照相馆部署

与客户端相比，服务器装有多个大容量硬盘，它们构成RAID，保证安全。服务器上运行着MySQL数据库，存放结构化的数据，记录客户、各次拍摄、订单等。但大部分空间用于存放客户的照片。

因为照相馆的距离很远，而且其中的合伙人不精通技术，所以我们知道把基础设施隐藏起来是很重要的。工作人员应该永远不必查看文件系统、分析失效原因或重启任务。他们当然应该从不登录数据库服务器。最坏的情况下，如果网线被碰松了，当它插回去时，所有工作都应该恢复正常，并且应该从临时的问题中自动恢复。

考虑这一点，我们设计了系统和应用的架构。

图像库

要让工作站可以互换，最基本的功能就是自动传输图像，包括从摄影师加载图像的工作站到服务器，以及从服务器到另一台工作站。

照相馆客户端和照相馆服务器都使用了一个核心组件，名为图像库。它负责处理所有的存储、加载和记录图像的工作，包括图像的元数据。在客户端，我们建立了一个本地的、带缓存的、后写入的代理。当调用者请求一张图片时，客户端图像库要么直接从本地缓存中取出它，要么将文件下载到本地缓存中，再取出它。不管用哪种方式，调用者都不必了解具体的细节。

类似地，当在客户端添加图像时，客户端图像库会将它上传到服务器。我们使用一个后台运行的线程池来进行传输，这样用户就不必等待上传。

客户端和服务器都大量运用了多线程。我们创建了一个锁系统，名为“预订”。预订是协作锁的软件形式。当客户想在图像库中添加一幅图像时，它必须先请求并持有“写预订”。通过这种方式，我们可以保证在我们发出该预订时，没有别的线程在读取图片。自然地，读取程序必须获得“读预订”。

虽然我们没有实现分布式事务或两阶段提交，但在实践中，在客户端图像库确认一次写预订和服务器端确认对应的写预定之间，只有很短的时间。当第二次预订确认时，我们就确信能够避免文件冲突了。

在实践中，甚至锁竞争也是很少见的。只有两个摄影师在两个不同工作站访问同一个客户的照片时才会出现。但是，每个照相馆有几台工作站，每个工作站上有许多线程，所以小心一点是值得的。

NIO图像传输

显然，剩下的问题是将图像从客户端传输到服务器。早期我们曾考虑并否决的一种可选方案是CIFS，即Windows共享驱动器。这里我们主要考虑的是容错，但我们也担心传输速度。这些机器需要来回转输许多数据，摄影师和客户要坐在那里等着。

在我们列出的所有现成的可选方案中，没有哪种方案可以同时确保速度、并行、容错和信息隐藏。虽不情愿，我们也只有决定创建自己的文件传输协议，这将我们带入了Creation Center最复杂的部分。图像转输变成了严峻的考验，但我们最后得到了整个系统最健壮的一项功能。

我以前对Java NIO曾有过一些经验，所以我知道我们可以利用它来实现一种非常快速的图像转输机制。实现NIO数据传输本身并不是太困难。我们使用常见的领导者一追随者（leader-follower）模式来提供并发，同时保证NIO选择器在一个线程中执行。

虽然该协议的实现并不困难，却有一些细小的地方需要处理：

- 套接字的两端都可以关闭它，特别是在客户端进行缓存的情况下。示例代码没有对这种情况提供合适的处理。
- 在处理IO事件时，SelectionKey仍将发出信号，表明它已准备好。如果你不清楚来自这个SelectionKey的兴趣集的操作，就可能导致多个线程调用同一段处理程序。
- 领导者必须处理所有对SelectionKey的兴趣集的变更，否则就会在Selector上产生竞态条件，所以我们必须创建一个队列来存放待执行的SelectionKey变更，领导者线程将在调用select之前执行这些变更。

处理这些小细节导致了不同对象之间的耦合比我预期的要多。如果我们当时是要设计一个框架，这个部分就需要更多地关注松耦合。但对于一个应用程序，我们觉得这是可以接受的，可以将服务器上这些操作的对象看成是一个内聚的单元。

当我们运行一个包嗅探器，想看看是否真地获得了最大可能的吞吐量时，出现了特别有趣的一幕。我们没有做到。最初，当接收方从有数据的套接字中读取数据时，它会读满一个缓冲区，然后返回。我们猜想，如果数据多于8192字节，不用等多少时间就会继续读取后面的数据。事实表明照相馆的网络相当快，在下一个线程再次调用处理程序之前就会填满服务器的TCP窗口，所以实际上每次传输都会停顿几乎一半的时间。我们在响应代码中添加了一个循环，它会不断读取，直到缓冲区读空为止。这使得传输时间几乎缩短了一半，而且减少了线程切换和调度的开销。我觉得这很有趣，因为它只在快速、低延迟的网络中有效，而且要在客户端总数较少的时候。对于网络延迟较高或客户端较多的情况，这种处理方式可能会导致某些客户端饥饿。同样，这是在我们的环境中有效的一种折中。

单元测试和代码复审

这个NIO文件服务器让我发现进行大组复审是很有帮助的，即使是在一个完全进行结对编程的敏捷项目中也是如此。

我的结对伙伴和我在大多数迭代中都一起处理多线程、锁和NIO机制。我们尽自己所能进行了单元测试，但在多线程和低级套接字IO之间，我们发现很难对代码产生自信。所以我们执行了另一件最佳实践：我们让更多的眼睛来关注它。但是我们称之为特例。我们对不能够编写足够的单元测试进行了补偿。

一般来说，在所有时间里让两双眼睛盯着代码，可以提供代码复审的所有好处。将结对编程与自动格式化和代码检查结合，就会使代码复审的价值小于它的代价。如果你能在不花代价的情况下获得好处，那为什么还要进行代码复审？

我们在工作室里放了一个投影仪，通过A/B开关连接到两台电脑。当我们有一项技术要展示，或有一个设计模式要分享时，我们就会在午餐后打开投影仪，走查一些代码。这在项目的早期非常方便，尤其是让架构和设计变得更流畅，以及我们在学习如何使用Spring和Hibernate时。这也有助于共享Eclipse的实践和技巧。

投影仪对于迭代展示也很方便。我们可以让所有利益相关人坐在房间里，不必挤在一个屏幕前面。

（更别说在墙上投影有趣的YouTube视频了。）

我知道，快速构建易碎的东西并不难。真正的挑战是让它变得健壮，特别是整个网络都在几百英里以外的照相馆中。这个网络不能让我们远程登录来调试问题，或在失效后清理现场。这个网络要面对小孩、分心的父母，服务器放在学步儿童视线的高度。想想这个不友好的环境！通过电缆传输数据是不够的，我们需要自动的文件传输并保证成功。

第一层保障是协议本身。对于“put”操作（将文件从客户端上传到服务器），请求的第一个数据包中包含了文件的MD5校验和。当客户端发出最后一个数据包后，它等待来自服务器的响应。服务器的响应代码包括：OK、TIMEOUT、FAILED_CHECKSUM或UNKNOWN_ERROR。如果收到的响应代码不是OK，客户端就会重发整个文件，我们称之为“快重试”。客户端在传输失败之间要经过3次快重试。

文件传输的问题分成两类。一类是“快速短暂的”，这种问题会自己消失，如网络错误。另一类要求人的干预。这意味着问题不会在几毫秒内消失，或者需要几分钟到几小时来修正。反复重试快速文件传输是没有意义的。如果几次尝试都无效，就不太可能在很短的时间内恢复正常。

因此，如果客户端经过3次快重试仍然失败，就会将文件放到一个文件传输任务队列中。一个后台任务会每20分钟执行一次，检查受阻的文件传输任务。它会再次尝试每个任务，如果还是失败，它会将任务放回到队列中。利用Spring的任务调度支持，这种“慢重试”是很容易实现的。

这种混合快重试和慢重试的策略让我们能够分离服务器和客户端的维护和支持。在升级或替换时，不需要“冷启动”整个照相馆的机器。

快速和健壮

本地和远程图像库及其相关文件传输机制变成了一项相当艰难的任务。但当它完成之后，整个系统向服务器上传图像的速度比从存储卡中读取的速度更快。下载到另一台机器的速度也相当快，用户根本注意不到背后的操作。在从一个屏幕转到另一个屏幕时，客户可以下载整个相册的所有缩略图。

下载屏幕大小的图片尺寸的图像进行完整显示是在点击鼠标时进行的。这样的速度让用户不必费时等待“加载中”对话框。

4.3.5 数据库迁移

想象一下操作600个远程数据库服务器，它们跨了4个时区。它们也可能在一个荒凉的小岛上，而且从数字化的角度来说，它们就是如此。如果数据库管理员需要手工进行变更，他就必须走遍几百个地方。

在这种情况下，一种选择就是在第一次发布之前将数据库设计做得完美无缺，然后再

也不改动它。也许还有一些人认为这是可能做到的，但我们团队中的成员肯定都不是这样想的。我们预期在各个层面上都会发生变更，包括数据库，我们甚至盼望着这些变更。

另一种选择就是将发行版声明送到现场。在执行安装时，照相馆经理总是会打电话到服务台，要求一步步的电话指导。也许我们可以将SQL脚本包含在发行版的CD中，让他们输入或复制/粘贴。想到指导他们输入命令时要从“现在输入mysqladmin -u root -p…”开始，我就感到背脊发凉。

最后，我们决定进行自动化数据库更新。Ruby on Rails称之为“数据库迁移”，但在2005年，这还不是一项常见的技术。

将更新作为对象

照相馆的服务器定义了一个bean，名为“数据库更新器”。它保存了一组数据库更新对象，每个对象代表了对数据库的一次原子变更。每个数据库更新对象知道它自己的版本，也知道如何作用于数据库。

在启动时，数据库更新器检查一个表，得知数据库当前的版本。如果它没有找到该表，它就认为没有更新或更新还没有执行。因此，第一次更新将建立这个版本表，并在其中填入一行数据。这行数据包含了一个版本号和一个锁字段。为了避免并发更新，数据库更新器首先会更新这一行，设置锁字段。如果不能设置，那它就认为网络中有别的机器正在执行更新。

我们使用这个迁移功能来执行一些简单的更新和复杂的更新。有一次简单的更新只是为一些影响性能的列加上索引。有一次更新让我们很紧张，它要将所有表的类型从MyISAM改为InnoDB。（MyISAM是默认的表类型，不支持事务和引用完整性。而InnoDB支持。如果我们在发布第一个版本之前就知道这一点，我们开始就会使用InnoDB。）由于我们已经部署了带有生产数据的数据库，所以必须使用一系列的“alter table”语句。它工作得很好。

在送到现场的几个发行版中，我们大约进行了10次更新。没有一次失败。

日常工作

每次我们执行构建时，都会将本地开发数据库恢复到版本0，然后向后升级。这意味着我们每天都会将升级机制执行数十次。

我们还对每次数据库更新进行单元测试。每个测试都会在更新之前对数据库的状态做出一些断言。然后它执行更新，再对之后的状态做出一些断言。

当然，这些测试都是针对“好行为”的数据进行的。但现场会发生奇怪的事情，真实的

数据总是比任务测试数据集更乱。我们的更新会创建表，添加索引，填充数据行，以及创建新列。如果数据不是我们期望的那样，有时候这些变更会导致糟糕的结果。由于我们担心更新时的这种风险，所以要寻找一些方法让这个过程更可靠。

安全特性

假定在一次更新中出了一些问题。照相馆就会关门歇业，直到经营者找到一种方法来恢复数据库。如果更新真出了问题，就会让数据库处于某种冲突状态或中间状态。这样照相馆甚至都不能回到应用的前一个版本。为了避免这种灾难性的场景，数据库更新器在开始执行升级之前会先对数据库进行备份。如果不能够完成备份，它就会停止更新过程。

如果在更新时发生错误，更新器会自动尝试装入原来的备份。如果这一步也失败了，那么至少还留有一个备份，技术支持人员会告诉照相馆经理进行手工恢复。

实际上，在最坏的情况下，打印工厂总有数据库的一份副本，它与最新数据库的差异不会超过一天。我们使用每日DVD的空闲空间来发送数据库的完整副本。这要求数据库较小而存储空间很多。

现场效果

我们花在自动化数据库更新上的时间有几个方面的回报。首先，我们通过一些早期的更新改进了性能和可靠性。在那个发行版之后，用户的积极反馈马上就来了。其次，运维小组非常喜欢容易部署新版本这个特征。以前的系统要求照相馆来回寄送可移动硬盘，这自然会伴随所有物流方面的问题。最后，有了这种更新机制，我们就可以关注于“刚好够”的数据库设计。我们不会预测未来，也不会过度设计数据库schema。相反，我们只设计足够的schema，支持当前的开发迭代。

4.3.6 不可修改的数据和处处使用的GUID

照相馆合伙人与客户一起，创建一些构图，利用多张照片插入某个设计之中。这些设计来自于公司总部的一个设计小组。一些设计是长期的，另一些设计是季节性的。各种设计的圣诞卡销量很大，至少在圣诞节之前的几周内。一旦过了圣诞节，需求就直线下降了。

一个具体的设计包含了一些作为背景的图像，还包含了一段描述，说明有多少空位放基本的图像以及这些空位的几何形状。合伙人可以非常创造性地用照片或其他构图来填充这些空位。

在处理这些设计和放入其中的基本图像时，我们遇到了一些有趣的挑战。例如，在客户下了订单之后，该设计的新版本又发到了照相馆，会产生什么情况？从较小的范围来说，如果合伙人将一个设计嵌套在另一个设计之中（例如将一张深褐色调的照片放在一个边框里），然后又改变或删除了原来的设计，你会怎么办？

最初，这看起来像是一个引用计数或隐式链接的噩梦。我们考虑的每一个创建对象引用网的策略都会导致间断丢失图像或让人吃惊的变化。作为一个团队，我们都相信“最小吃惊法则”，而隐式链接会导致变更从一个产品波及另一个产品，所以不会有好结果。

当我们中最有想象力的同事得到了一个简单而清晰的答案时，没过30秒就得到了大家的一致赞同。这个解决方案包括两条规则：

1. 在创建之后就不要改变。设计和构图是不可修改的。

2. 复制原来的设计，不要引用。

把这两条规则结合在一起，这意味着选择一个设计实际上就是把这个设计复制到了工作空间。如果合伙人把由此产生的构图加到了相册中，那么它就是添加的设计的一个完整的、自包含的副本。同样，将一张处理过的图像嵌套在另一个图像中也会将原来的图像复制并组合到新的构图中。从组合发生之时开始，原来的构图和新的构图就是完全相互独立的。

这些副本不只是内存中对象引用的一个技巧。对构图的实际XML描述也包含了完整的设计副本或嵌入的构图。这个描述存在于照相馆的数据库中，与DVD中发送的描述是一样的。当照相馆经理将当天的订单刻成DVD时，StudioServer就将创建最终效果所需的全部内容都进行打包：源图像、背景、alpha遮罩，以及如何将它们组合成最终图像的指令。

在DVD上包含整个构图的完整描述（包括设计本身）为生产带来了很大的好处。

以前的系统将设计保存在一个库中，订单通过ID来引用设计。这意味着设计者必须同步照相馆和集中式打印工厂的所有设计ID。所以，设计必须先在生产系统中“注册”，然后才能分发到照相馆。一旦ID没有同步（这种情况有时会发生），错误的设计就会打印出来，客户就不会拿到他们期望的照片。同样，不论设计者何时更新一个设计，可能有几天的DVD使用了该设计的老版本。有时候效果很好，有时候不好。

在新系统中，设计永远不需要注册。进入XML中的内容就会打印出来，这让设计者可以自由地进行更频繁的更改，并随时分发设计。新版本的设计不会影响到已有的订单，因为每份订单都是自包含的。当新版本到达照相馆时，它就会开始出现在订单流程中。

唯一不复制的是图像文件本身。它们太大了，所以我们为每个图像（不论是设计的一部分还是照相馆中拍摄的图像）分配了一个GUID。作为一项规定，当图像得到一个GUID后，它就不可修改了。在准备将订单刻录到DVD上时，StudioServer会遍历订单收集GUID（利用有争议的Visitor模式）。它将找到的每个图像添加到DVD中，包括客户的照片和设计背景。

4.3.7 渲染工厂

StudioClient帮助合伙人利用基本图像创建增强的肖像。这些增强的肖像可以是简单的深褐

色效果或黑白效果，目的是让肖像看起来更戏剧化，也可以是复杂的多层结构，带有alpha遮罩的背景、文字和柔焦。不论是哪种效果，照相馆的工作站都不会得到最终渲染的图像。打印工厂拥有各种打印机，支持不同尺寸和分辨率。他们可以自由地更换打印机或在任何时候将任务移到其他打印机上。照相馆对于如何得到打印好的图像了解得并不多。

当那些每日生成的DVD到达时，它们被载入生产控制系统（PCS）。PCS负责做出所有决定，包括何时为订单渲染图像，何时打印，使用哪一台打印机打印等。另一个独立的团队负责开发PCS，他们处于另一个时区的某个地点。以前的项目试图与PCS紧密集成，这导致了大量的冲突。所有团队都怀着良好的意图在工作，但沟通的困难让各个团队都慢了下来。我们需要避免这种冲突，所以我们决定主动应用Conway法则（在下一节中定义），在软件中明确创建一个接口，让大家知道团队工作的边界在哪里。

增量式架构

在敏捷社区中一个反复出现的问题就是："你应该事先创建多少架构？"某些杰出的敏捷思想家会告诉你："一点也不需要。无情地重构，然后架构就会出现。"我还没有达到这个境界。

重构在不改变代码功能的前提下改进了代码的设计。但是，要通过重构得到更好的设计，首先你必须知道好的设计和不好的设计。我们有一组不错的"代码味道"来指导我们，但我们不知道相应的"架构味道"。其次，必须有可能持续地改进，甚至跨越接口边界。这总是让我相信系统的基本架构必须在开始开发时就确定好。

现在，在Creation Center项目之后，我对这个答案的信心减弱了不少。我们在项目较晚的时候加入了一些主要的架构组成部分。下面是一些例子：

- Hibernate：在两三次迭代之后加入。在此之前我们不需要数据库。
- Spring：在第一个发行版进行到大约三分之一时加入。它很快就成为了我们架构的中心。我不记得在没有它时我们是如何工作的，但情况确是如此。
- FIT：在第一个发行版进行到大约一半时加入。
- DVD刻录软件：在初始开发快结束时购买并加入。
- 支持分窗口的UI：在发布前最后两次迭代中加入。

在每一种情况下，我们都会先探索所有可能性，然后再做决定。我们会在"最后可能的时刻"做出决定，即不做决定的代价超过了实现该特征的代价。尽管如果一开始就用Spring，有些事情我们可能会做得不一样，但在后来加入它也没有让我们受苦。在早期的迭代中，我们关注的是发现应用想成为什么样子，而不是Spring希望我们如何构建应用。

应用Conway法则

Conway法则常常在事后被人提起，来解释那些产品中看起来似乎很随意的决定。它说出了有关开发团队的基本事实：在所有团队的边界，你会发现软件的边界。这是因为需要沟通接口而导致的。

我们觉得保持Creation Center对DVD格式和布局的完全控制是很重要的，所以我们在自己的工作范围内增加了一个程序：DvdLoader。DvdLoader运行在打印工厂中，负责读取DVD并调用PCS中的不同存储过程，添加订单、构图和图像。PCS将组合指令看成是一个透明的字符串，我们很小心地避免让PCS“打开”这个字符串中的XML。有时候这意味着我们会有重复信息，诸如对基本图像本身的依赖关系，但这对于维持一个清晰的边界来说是一种可以接受的折中。

类似地，我们定义了一个接口，让RenderEngine从PCS中取得渲染任务，同时保持渲染本身的XML描述处于Creation Center的控制之下。

我们编写了这些接口的规格说明，然后利用运行在开发服务器上的FIT来“敲定”准确的含义。实际上，我们将FIT作为这些接口的可执行的规格说明。后来发现这很重要，因为即使对于参加谈判某个接口的人来说，也会发现在他们认为同意的内容和实际构建的东西之间存在着差异。FIT让我们在开发中就消除了这些差异，不需要在集成测试或生产环境中再来消除，那样会更糟。

DVD加载

DvdLoader程序运行在打印工厂，它实际上是一个批处理器，从DVD上读出订单并将它们装入PCS。像其他程序一样，我们关注的是健壮性。DvdLoader读取一条完整的订单，验证该DVD包含了该订单的所有构成元素，然后将它加入PCS。通过这种方式，我们不会将不完整的订单或有冲突的订单加入数据库。

因为图像可能出现在许多DVD上，加载程序会利用GUID来检查是否某张图像已加载。如果没有，加载程序就加上它。这样，在需要时，订单就可以从照相馆重新发出，就算PCS已经清除了订单和相关的图像也没关系。这也意味着设计中使用的背景图像只在使用该设计的第一份订单到达时加载。

所以，DVD是自包含的，不论加载几次得到的结果都一样。

渲染管道

对于渲染引擎本身，我们使用了经典的管道和过滤器架构。“管道”是对图像渲染的自然隐喻，它将一系列复杂的动作分解成具体的步骤，让单元测试变得简单。

在从PCS取出一项任务时，渲染引擎会创建一个RenderRequest。它将这个RenderRequest

送入渲染管道，其中的每一个步骤都作用于该请求。在最后一个步骤，管道将渲染好的图像保存到PCS指定的路径下。在该请求退出管道时，它只保存得到的结果，包括成功标记以及可能有的问题集合。

管道中的每个步骤都有机会报告问题，将错误信息添加到结果中。如果某个步骤发生错误，管道就会终止，引擎将问题报告给PCS。

快速失败

每个系统都有一些失败模式，唯一的问题在于你是针对失败进行设计还是任其发生。我们仔细设计“安全的”失败，特别是在生产过程中。我们绝不希望因为软件的原因而导致生产线停下来。

还有另一个方面。当客户回来取他的订单时，应该取走正确的照片！也就是说，我们交付的产品需要对产品和客户的订单进行匹配。这看起来是小事一桩，但让产品图像的渲染和屏幕图像的渲染保持方式一致是非常重要的。我们努力确保在生产使用的渲染代码与照相馆中使用的相同。我们也确保渲染引擎在生产时使用相同的字体和背景。

在我们的渲染引擎中，我们采用的哲学是“快速失败、大声失败”。当渲染引擎从PCS取出一项任务时，它会检查所有指令，验证该任务需要的所有资源都实际存在。如果该任务包含文字，那么渲染引擎就立即装入字体。如果该任务包含某些背景图或alpha遮罩，那么渲染引擎就立即装入相关的图像。如果缺了什么，它就马上向PCS报告错误并终止该任务。在渲染管道的16个步骤中，前5个都是进行验证。

在上线几个月之后，我们终于发现了一个渲染引擎没有及早检测到的错误：它没有为前面渲染的图像预订磁盘空间。某天当PCS的存储卷满了的时候，渲染任务很晚才失败，而不是尽早失败。在此之前，没有因为失败的渲染而返工的。

伸缩性

每个渲染引擎都是独立操作的。PCS不会保留现有渲染引擎的清单，每个引擎直接从PCS中取任务。实际上，引擎可以根据需要添加或移除。因为每个引擎在完成一项任务之后就立即请求新的任务，我们自动实现了负载均衡的效果，充分利用了每个引擎的能力。快速的渲染引擎以较快的速度完成任务。不同配置的渲染引擎不会引起问题。

唯一的瓶颈将是PCS本身。因为渲染引擎会调用存储过程来取得任务和更新状态，每个渲染引擎每过三五分钟都产生成两个事务。PCS运行在还算不错的Microsoft SQL Server服务器集群上，所以没有很快成为吞吐量瓶颈的危险。

4.4 用户反应

我们的第一个版本安装在两个本地照相馆中，它们的距离容易让我们产生“驱车过去除错”的冲动。合伙人的反馈很及时，也相当积极。一个照相馆经理认为新系统快了许多，而且更易用，所以她预计在假期能够多接待50%的客户。据说有一个客户曾询问在哪里能够买到这个软件的副本。我们常常听说客户直接拿起鼠标自己修饰照片。你可以想象，客户更愿意订购他们自己创建的照片。

我们在生产过程中遇到了一些缺陷，但很快就纠正了。由于我们在加载程序和渲染集群中设计的弹性，打印工厂能够很容易进行伸缩，处理来自比预期更多的照相馆的照片，同时保持很高的生产品质。

4.5 结论

我本可以花更多的时间和篇幅来描述每个类、每次迭代或每个设计决定，就像初为人父的人描述宝宝的每次打嗝和摇晃一样。但这一章浓缩了一年的努力、探索和心血。它展示了Creation Center架构的结构与功能如何从业务的基本需求及环境中浮现。通过保持关注点分离，坚持增量式设计和开发，Creation Center以一种令人满意的方式平衡了各方面的需求。

参考文献

Buschmann, Frank, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-Oriented Software Architecture: A Pattern for Distributed Computing*, vol. 4. Hoboken, NJ: Wiley.

Fowler, Martin. 1996. *Analysis Patterns: Reusable Object Models*. Boston, MA: Addison-Wesley.

Fowler, Martin. “Application facades.” *http://martinfowler.com/apsupp/appfacades.pdf.*

Gamma, Erich, et al. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

Hunt, Andrew, and David Thomas. 1999. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley.

Lea, Doug. 2000. *Concurrent Programming in Java*, Second Edition. Boston, MA: Addison-Wesley.

Martin, Robert C. 2002. *Agile Software Development, Principles, Patterns, and Practices.* Upper Saddle River, NJ: Prentice-Hall.

	原则与特性		结构
	功能多样性		模块
√	概念完整性		依赖关系
	修改独立性		进程
	自动传播	√	数据访问
	可构建性		
√	增长适应性		
√	熵增抵抗力		

第5章

面向资源的架构：在Web中

Brian Sletten

架构是有人居住的雕塑。

——康斯坦丁·布朗库西（Constantin Brâncusi）

在这一章中，我们将说明企业中聚焦信息的架构展示了与Web一样的明确特点：伸缩性、弹性、架构迁移策略、信息驱动的访问控制等。在这个过程中，业务部门有权力根据业务需求来决定资金投入和软件开发，而不是简单地因为选择容易失效的技术而要求业务部门为不断的改变付费。

5.1 简介

作为IT行业从业人员，我们很没面子，因为我们不得不承认这个令人尴尬的事实：大多数的组织机构更容易在Web上找到信息，而不是在他们自己的系统中。想一想这一点。他们更容易通过第三方、在全球信息系统中找到数据，而不是在他们自己能够完全控制和访问的环境中。这种尴尬的事情有许多原因，但最大的问题是我们倾向于在内部使用错误的抽象，过分强调我们的软件和服务，忽视了数据。这种方向性错误在一定程度上导致了业务部门对IT部门感到烦恼。我们忘记了公司并不关心软件，而只是关心它提供的特征和功能。业务部门真正想要的是一些更容易的方法，这些方法能够管理他们收集的数据，对数据进行分析，通过复用数据来支持客户和核心功能。

为什么组织机构内的信息管理与Web有着如此巨大的差别？遗憾的是，答案既与技术选择有关，也与公司策略有关。我們有一些遗留系统，使交互模式变得复杂。我们试图利

用供应商的一些解决方案，而这些供应商的利益并不与我们保持一致。我们希望有银弹来解决所有的问题（尽管Brooks博士在多年前就纠正过我们的观念（注1）。即使你碰巧进入了一个拥有非常合适的技术基础设施的组织机构，数据管理人和数据用户通常也会互相争夺，这抑制了信息共享。这是公司不像Web那么清楚的原因之一：似乎没有合适的动机来分享数据，尽管分享数据的需求是很明确的。最重要的信息是，有些问题与技术无关。在一定程度上，Web技术也会帮助我们绕过策略问题，因为你不总是需要特别许可才能暴露出你通过其他形式得到的信息。

好消息是我们可以借助Web来指导我们，弄明白这种寻找信息的极好环境背后的原因。在组织机构中应用这些概念将帮助我们解决这些问题，得到类似的好处，如低成本的数据管理、架构迁移的策略、信息驱动的访问控制，以及满足法律法规。Web的成功在很大程度上是因为它增大了信息共享的可能性，同时又降低了门槛。我们已经创建了一些工具和协议，不但支持世界领先的科学家之间的知识传递，而且也支持我们的祖母与家人之间保持联系，并找到他们感兴趣的内容和社区。这不是一件小事，我们应该好好考虑一下导致这些现实背后的思想所产生的影响 。我们必须住在我们创建的架构里，所以我们应该建造既能满足我们的需求又能为我们带来灵感的架构。

5.2 传统的Web服务

在我们开始为信息驱动的环境寻找新的架构之前，我们应该简单回顾一下最近我们如何构建了类似的系统，怎样才能做得更好。在最近10年里，我们一直在推销企业级架构的愿景，这种架构是建立在可复用的业务服务的基础之上的。我们需要提醒自己，Web服务的目标是要成为一种业务策略，成为一种方式，让我们能够在一些地方定义功能，在任何地方、以各种编程语言异步地访问这些功能。我们希望能够在不影响客户的情况下升级这些服务。遗憾的是，与这个目标有关的不断变化的技术组合让人们很迷惑，而且也没有解决在真实组织机构的实际架构中所面临的问题。我们在这个新愿景下的目标不是与以往不同，而是要增加价值，改变我们所看到的“面向服务恶化（Service-Oriented Aggravation）”的状况。

有一组技术构成了我们对Web服务的基本理解：SOAP用于服务调用，WSDL用于协议描述，UDDI用于服务元数据发布和发现。SOAP来自于一些不同的技术，包括远程过程调用（RPC）模型和异步XML消息模式（doc/lit）。第一种技术是容易失效的，伸缩性不佳，在以前的DCOM、RMI和CORBA等名称下，它的实际效果并不太好。尖括号既没有带来新问题，也没有解决老问题，我们只不过试图以这种方式构建一些粒度上错误的系统，并过早地将自己与一个明显不稳定的协议绑定在一起。第二种技术提高了艺术

注1： *http://en.wikipedia.org/wiki/No_Silver_Bullet.*

水平，是一种不错的实现策略，但是不太能配得上它从一开始就一直进行的交互性宣传。它甚至把简单的交互弄得更复杂了，因为它的处理过程受到解决大型交互问题的目标的影响。

doc/lit风格让我们用结构化的包来定义请求，这些请求可以在一个工作流中由一些松耦合的参与者进行转手、修正、处理和再处理。就像一颗流动的珍珠，这种信息在由中间节点和最终节点异步处理的过程中，会增加元素和属性。我们通过在一层中添加更多消息处理器来实现水平伸缩性。我们可以在合作伙伴之间和行业边界之间标准化交互方式和业务流程，这些业务流程不能包含在一个单独的环境中。它代表了一种上下文无关的请求，能够处理非常困难的交互模式。

当只靠严格的服务分解和描述（如SOAP和WSDL）被证实不足以解决我们的交互需求时，我们沿着处理栈向上，引入新的业务处理和混合层。于是标准和工具的组合使得已经过于复杂的情况变得更为复杂。当我们跨越领域和组织机构边界时，我们遇到了冲突的术语、业务规则、访问策略和非常真实的WS巴别塔。即使我们信奉这种愿景，我们也没有切实可行的迁移策略，实际上我们对可能实现的交互性撒了谎。Clay Shirky曾把Web服务的交互性总结为“乌龟叠罗汉”（注2）

问题在于，当大多数人希望以语言无关和平台无关的方式来调用可复用的功能时，这些技术却成了牛刀杀鸡，它们太复杂了，也泄露了实现的细节。为了调用这些功能，你必须会使用SOAP语言。这是一个不错的实现选择，但在这个松耦合系统的世界中，我们并不总是希望仅仅为了简单的交互方式，就向客户宣传或要求客户知道这些细节。SOAP的大思路是让请求与上下文无关，并在异步的环境中维持事务完整性。但在真实系统的业务现实中，上下文被加入到请求中。首先，我们必须为请求加上唯一标识，然后是证书，然后对消息签名并为敏感的信息加密，等等。发出SOAP请求的“简单”开销受到了交互方式和业务需求的影响。如果组织机构中的某人希望取得某些信息，为什么不能直接索取？而且，当这些问题被回答过一次之后，为什么问同样问题的10个人（或100个人、1000个人）还要将同样的开销通过相同的查询加到后台系统上去呢？

这些问题凸显了传统Web服务技术栈中存在的一些抽象问题，在一定程度上解释了世界各地IT部门对Web服务技术的普遍不满。这些技术是一些实现技术，将我们的调用行为分解成由服务组成的工作流，但我们不能够仅通过服务的概念来表达组织机构的全部需求。我们不能够脱离上下文来识别和确定信息结构，这种上下文是特定调用中所用到的。当我们将自己与特定协议的请求、特定的端口和特定的机器绑在一起时，我们就丧失了松耦合和异步交互的模式，也不能够跟上数据视图的变化。如果不能够唯一地识别通过

注2： *http://en.wikipedia.org/wiki/Turtles_all_the_way_down.*

服务的数据，那么我们就不能够在某个信息层面上应用访问控制。这使得在不断网络化的世界上保护敏感的、有价值的、私密的信息这一问题变得更为复杂，更加难以应付。

这里的问题并不在于SOAP和WSDL，但它们也不是完整的解决方案。在我接下来要描述的面向资源的架构中，很可能以doc/lit的方式来使用SOAP，但我们不会把它作为唯一的解决方案。如果没有必要，我们也不需要总是宣称背后使用的是这些技术。为了迈出这一步，我们需要看看Web，弄明白为什么它作为一个可伸缩、灵活、可演变的信息共享平台取得了如此的成功。实现细节对于信息消费者来说通常并不重要。

5.3 Web

在大家的概念中，Web是以文档为中心的。具体来说，当我们想到Web时，就会想到在Web浏览器中看一些文档。但真正的神奇之处却在于公开可获得的信息之间的链接、这些链接所代表的意义，以及我们能够方便地通过创建新窗口来访问链接的内容。没有起点，也看不到终点。只要我们知道要找的是什么，我们通常就能找到它。现在已经出现了一些技术，帮助我们知道要找的是什么，即通过一些搜索引擎或某种形式的推荐系统。

我们喜欢为事物命名，因为我们从本质上是面向名称的生物，我们使用名称来区别“这个东西”和“那个东西”。作为儿童，他们的早期沟通行为之一就是为感兴趣的东西命名，指向它们，索取它们。在很大程度上，Web就是这种儿童般的好奇心应用于群体智慧和愚蠢的结果。作为一种对知识永不知足的生物，我们会决定对什么感兴趣，然后就开始索要它。没有集中式的协调，我们自由地记录下我们的探索，出版我们的故事、思想和旅程。我们把Web看成是文档之间的一系列单向链接（参见图5-1）。

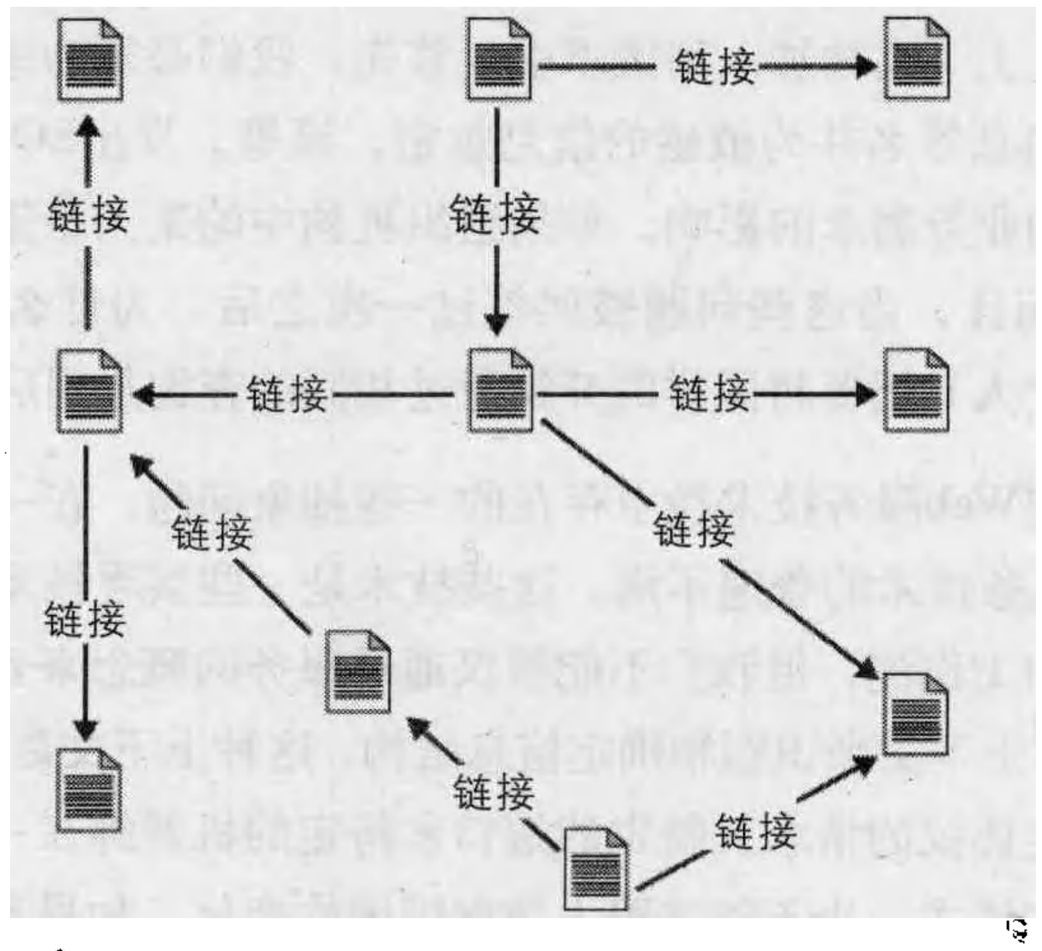


图5-1：Web的传统概念

但是，链接起来的文档只是一部分。Web的愿景中总是包含了数据链接的思想。这种内

容可以通过渲染的视图来展现，或在不同的环境里用合适的方式直接引用或操作它们。你可以设想有一个中间层要求XML文档格式的信息，而表示层则希望通过AJAX调用取得JSON对象。同样的名称代表了不同形式的同样数据。通过允许以这种方式指定数据，我们很容易创建分层的应用，它们包含一致的视图，即使它们要求不同的细节程度，或希望数据有不同的表示形式。应用和环境提供与消费数据的方式是松耦合的，应用不再是“在网上”，而是“在网里”。我们正走向数据的Web，它连接了人、文档、数据、服务和概念，如图5-2所示。

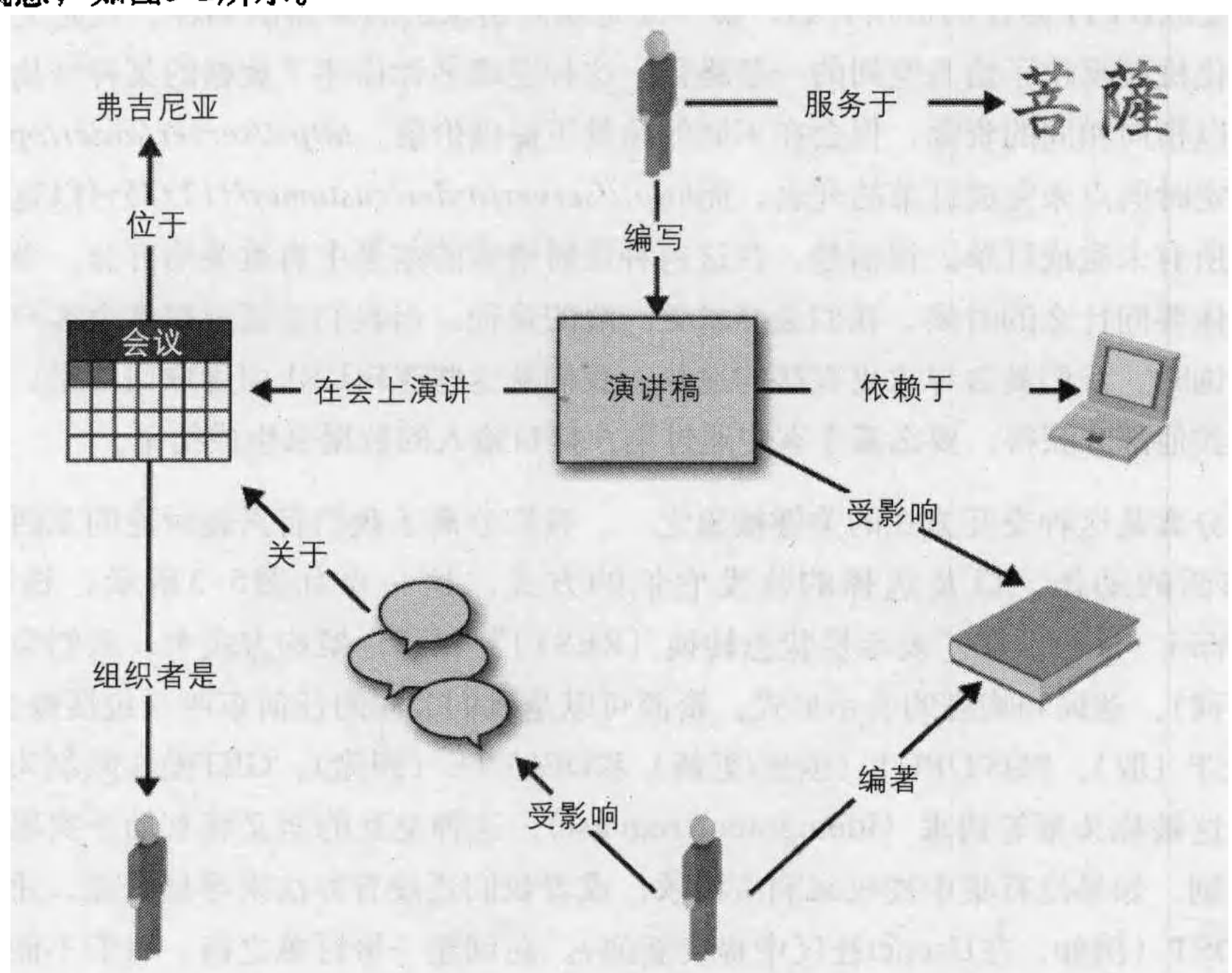


图5-2：数据的Web

在这个环境中，基本的交互是逻辑上的客户端-服务器请求。我们必须有感兴趣的信息的地址。术语“统一资源定位符（URL）”指的是一种标识符，它不仅无歧义地确定了在全球地址空间的一个引用，而且告诉我们如何解释这个请求。在这个过程中，不要求我们事先懂得实现该请求的技术。这让处理过程保持简单，在面对后端变化时有弹性。当我们最喜欢的站点从静态的数据提供变成动态的数据提供，或者改变应用服务供应商时，这些动作对我们来说是不可见的。虽然许多站点在这个过程中没能够有效地处理内容安排，但至少我们能够从相同名字的地址接收不同的表示形式。我们可能希望以不同的格式取回数据，这取决于我们是用计算机发出请求还是用手机发出请求。在后面的讨论中，我们会看到如何利用这个属性来控制细节程度，以满足访问控制和满足法规的要求。

Web使用的命名机制让我们能够标识文档、数据、服务，甚至概念。我们曾经很难区分对Abraham Lincoln的引用和一篇关于它的文章。例如，站点*http://someserver/ abrahamlincoln*可能是两者之一。W3C技术架构组（TAG）提出了一份建议（注3），非网络可寻址的资源（如不存在于Web上，但我们仍感兴趣的东西）可以通过返回码303来说明，而不是通常的200。这相当于暗示客户端："是的，你请求的东西是合法的，也很有趣，但实际并不存在于Web上。你可以在这里找到更多的信息……"

Web地址以HTTP协议的引用开始，接下来是响应请求的服务器的名称。在此之后，一种层次化机制反映了信息空间的一条路径。这种逻辑名称描述了数据的某种结构。多条路径可以指向相同的资源，但会在不同的场景下提供价值。*http://server/order/open*可以返回特定时间点未完成订单的列表，而*http://server/order/customer/112345*可以返回某个客户的所有未完成订单。很清楚，在这两种逻辑请求的结果中有重叠的部分。当我们不知道具体要问什么的时候，我们会寻求更一般的途径。当我们希望根据某个客户的状态进行查询时，我们就会寻求更直接的途径。我们从这些逻辑URL引用取回结果，要么从系统的其他部分获得，要么基于客户通过用户接口输入的数据来生成结果。

关注点分离是这种交互方式的关键抽象之一。我们分离了我们有兴趣讨论的东西、操作这些东西的动作，以及选择的收发它们的方式。这一点如图5-3所示，该图来自RESTWiki（注4）。在"表示层状态转换（REST）"（注5）架构方式中，我们会提到资源（名词）、动词和响应的表示形式。资源可以是我们关注的任何东西（包括概念）。动词是GET（取）、POST/PUT（创建/更新）和DELETE（删除）。GET操作限制为没有副作用。这被称为幂等请求（idempotent request）。这种交互的语义将有助于实现可能的缓存机制。如果没有集中授权来响应请求，或者我们还没有办法来寻址资源，通常就会使用POST（例如，在Usenet社区中提交新闻）。在创建一份订单之前，我们不能够为它提供标识符，因为服务器应用负责生成订单ID。因此，我们通常会向某些功能程序（如Servlet）POST这些请求，这些功能程序代表我们接收请求，并在此过程中生成ID。PUT用于更新和重写一个已命名资源的现有状态。DELETE在公共Web上用得不多，但在内部控制的、面向资源的环境中，声明不再需要或关心某些资源则是资源生命周期管理的重要部分。REST方式的基本工作原理是分离关注逻辑命名资源、操作资源的方式，以及选择的表示资源的格式，如图5-3所示。

这种关注点分离与SOAP服务调用的协议本质形成鲜明的对比，在SOAP服务中，请求的结构、调用的行为和返回类型的格式通常通过Web服务定义语言（WSDL）捆绑到一个协议

注3：*http://lists.w3.org/Archives/Public/www-tag/2005Jun/0039.*

注4：*http://rest.blueoxen.net/cgi-bin/wiki.pl?RestTriangle.*

注5：*http://en.wikipedia.org/wiki/REST.*

上。协议很有用，除非我们想打破这些协议。Web服务技术栈的一个主要目标就是减少耦合，并引入异步处理模型，这样消息的处理程序就可以根据新的业务逻辑进行更新，同时又不影响客户端。WSDL绑定的方式注意到了这个目标，但做法却刚好相反。我们通常不能在不影响客户端的情况下改变一个端口的后台绑定（而这正是我们明确想避免的）。

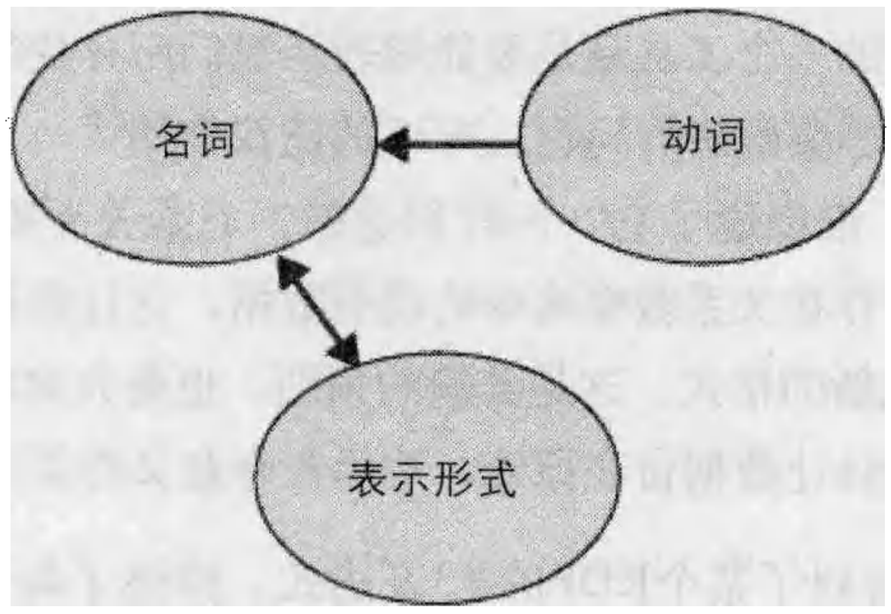


图5-3：REST关注点分离

面向资源的方式让我们在需要时可以强制实现协议，但这并不是必需的。通过分离事物的名称和形式结构，我们可以复用同样的逻辑名称，支持多种类型的交互 。我们可以更新后台而不会影响原有的客户端。如果我们要改变一个模型，所有已有的客户端用第1版的消息结构向一个URL POST消息，那么我们可以在后台添加对第2版消息结构，同时在需要时让业务照常进行。如果我们想停止支持老的消息结构也可以，但我们可以选择什么时间这样做。这种灵活性是面向资源的架构有助于我们重新控制业务后端的原因之一：后端系统的改变不一定要求前端也更新。如果用一个RESTful接口将遗留系统包装起来，我们就可以继续使用它，直到有足够的业务理由来改变它为止。当然，其他技术也支持我们用这种方式包装遗留系统。但正是使用逻辑名称的这种一般方式让我们有更大的机会来避免中间件的不断改变，让情况变得不同。

为了努力促进水平的可伸缩性，RESTful方式要求请求是无状态的。这意味着生成响应需要的所有信息都要放在请求中。这让我们能够通过负载均衡机制，由任意多台后端服务器来处理请求。在面对增加的负载时，可以添加更多的硬件来解决问题，任何服务器都可以接收并处理请求。虽然伸缩性是这种架构方式的目标，但将无状态的请求应用于GET请求的语义还带来了另一个大的副作用：我们可以设想对任意请求的结果进行缓存的可能性。响应者的地址（URL的主要部分）加上请求的全部状态（URL层次路径加上查询参数）构成了结果集的一个复合散列键（例如，数据库查询、对一部分数据进行转换等）。你不会不花力气就享受到缓存的好处，但利用这种可能性的环境忽然就变得很容易想到了。NetKernel面向资源环境（注6）的一个突出特征就是它充分而全面地利用

注6：*http://1060.org.*

了这种可能性带来的好处，实现了一种架构上的记忆化（注7），而你几乎不要做什么额外的工作。我们将在稍后的5.6节中进一步讨论。

对所有我们感兴趣的事物使用通用的命名机制，用一个逻辑处理过程来支持事物随时间的变化或在不同上下文中的变化，这样我们就几乎拥有了一个引领组织机构信息管理所需的基础设施。我们最后需要的一个工具就是要能够表示我们所寻找的事物的元数据。这就是“资源描述框架（RDF）”想要解决的问题。W3C的建议使用了一个图式模型，支持对命名实体信息的开放式表述。谁创建了它？何时创建的？它是关于哪方面的？它和什么有关系？我们可以命名和寻址存在关系数据库中的现有数据，这让我们可以描述任何想描述的数据，不必将它们转换成新的格式。这是普遍的预期，也是大家报怨的原因，因为RDF在实践中撑不住。我们通常会让数据留在原处，然后在有意义的层面上进行集成。

在下面的列表中，我们看到了某个RDF的N3表达式，描述了与特定资源相关的创建者、标题、版权日期和许可证。这个例子向我们展示了3个来自“都柏林核心元数据倡议”（注8）的术语和1个来自Creative Commons（注9）社区的术语。我们可以自由使用来自任何词汇表的术语，或在需要描述新的内容时创建新的术语：

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix cc: <http://creativecommons.org/ns/> .
<http://bosatsu.net/team/brian/index.html> dc:creator
        <http://purl.org/people/briansletten> .
<http://bosatsu.net/team/brian/index.html> dc:title
        "Brian Sletten's Homepage".
<http://bosatsu.net/team/brian/index.html> dc:dateCopyrighted
        "2008-04-26T14:22Z".
<http://bosatsu.net/team/brian/index.html> cc:license
<http://creativecommons.org/licenses/by-nc/3.0/> .
```

我们现在不仅能够使用任何想用的术语，也可以在将来任何时候添加新的术语和关系，不影响原有的关系。这种“无schema”的方式对于修改过XML或关系数据库schema的人来说，具有很大的吸引力。它也代表了这样一个数据模型，即不仅能够在不可避免的社会、过程和技术变化中生存下来，而且拥抱这些变化。

这个RDF可以存储在triplestore或其他数据库中，通过SPARQL或类似的语言进行查询。大多数支持语义的容器现在都支持这种存储和查询RDF的方式。包括Mulgara Semantic Store（注10）、Sesame Engine（注11）、Talis Plateform（注12）、甚至Oracle 10g以后的

注7：*http://en.wikipedia.org/wiki/Memoization.*

注8：*http://dublincore.org.*

注9：*http://creativecommons.org/ns.*

注10：*http://mulgara.org.*

注11：*http://openrdf.org.*

注12：*http://talis.com.*

版本。图中的节点可以通过模式匹配条件来选择，所以我们可以针对资源问这样的问题："谁创建了这个URL？""给我看Brian创建的一些东西。"或"列出所有最近6个月中创建的Creative Commons许可证的素材"。表示"由谁创建""具有哪种许可证"这样的术语是在相关的词汇表中列出的，但很容易转成我们所提到的目标。数据模型的灵活性以及查询语言的表达能力使得描述、发现和调用RESTful的服务变得相当简单。这当然比通过UDDI这样的迟钝的、高阻抗的技术来发现和调用服务要舒服得多。

能够寻址和解析任意资源，能够以不同格式取得这些资源，能够以开放世界和混合词汇的方式描述它们，我们就可以将这些思想应用于企业了。我们将描述一种信息驱动的架构，支持在数据组成的网上"冲浪"，就像在文档组成的网上"冲浪"一样。

5.4 面向资源的架构

面向资源的架构的标识是向命名的资源发起逻辑请求的过程。这些请求由某种引擎解释，转成该资源的物理表现形式（如HTML页面、XML格式、JSON对象等）。参见图5-4。

这张图展示了面向资源的架构（ROA）的基本交互方式。逻辑请求由一个面向资源的引擎来命名、解析，并将结果返回给请求者。命名的请求可能被解释为一个数据库查询，或是某项信息管理功能（如RESTful服务）。响应请求的可能是一个Servlet、一个Restlet（注13）、一个NetKernel模块，或某些可以寻址、能够理解该请求的功能，而对此信息感兴趣的人基本上不关心这些响应请求的程序。这个逻辑步骤隐藏了所有的可能性和技术选择，没有向客户透露一点不必要的细节。它确实不能支持所有的交互方式，但你可能会惊讶于有多少东西可以舒服地藏在URL后面。

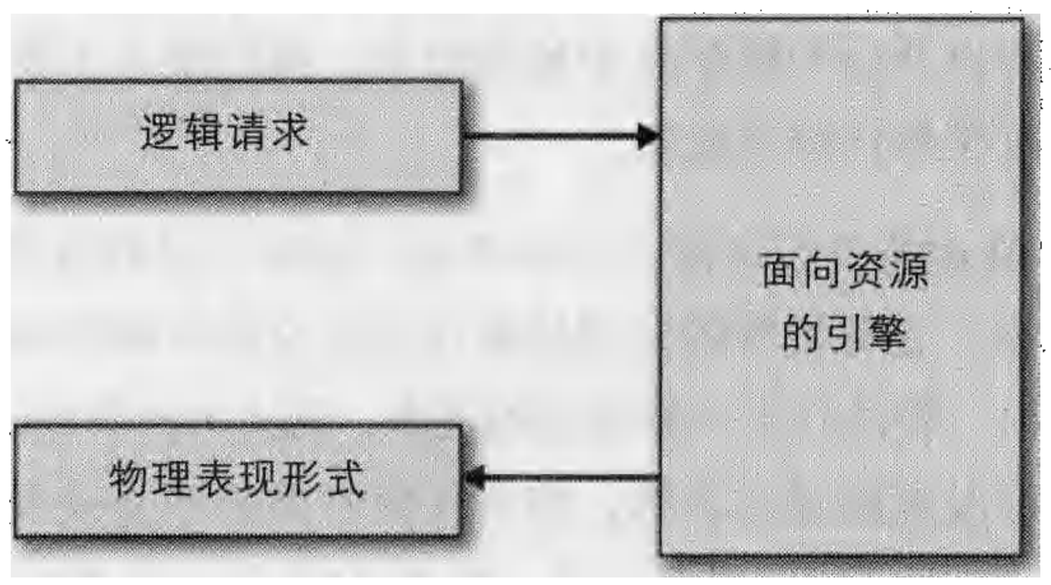


图5-4：面向资源的架构

请考虑这个地址：*http://server/getemployees&type=salaried*。许多人会认为创建这样的地址就是在采用REST。遗憾的是，这不是一个好的REST服务名称（大多数"拉斯塔法里分子(Restafarians)"会说这根本不是REST!），因为它混合了名词和动词。这就是我

注13：*http://restlet.org*.

常说的“通过URL来寻址行为”或“通过URL来实现RPC”。用REST的方式来区分名词和动词没有什么神奇的，它只是能让我们识别我们关心的事物。前面提到的URL不能够用于更新雇员列表，因为POST一条雇员记录到“/getemployees”没有什么意义。相反，如果URL是*http://server/employee/salaried*，那么向它发起GET请求将得到相同的信息，而它将成为“领薪雇员”这个业务概念的长期有效地址，正如*http://server/employee/hourly*可以指按小时付薪的所有雇员一样。我们可以选择不更新这些信息资源，因为它们代表的是对后台数据库的查询。但是，在/employee信息空间内是一致的，我们可以选择通过其他方式来导航。*http://server/employee/12345678*代表了一个具有特定ID的雇员，而*http://server/employee*可能代表所有的雇员。向后面一个URL POST一条记录可以代表雇用了某人。向包含具体雇员ID的URL PUT一条记录可以代表在岗位变更、升职或加薪之后更新雇员记录。向该地址发出DELETE请求可以表示这个命名资源在该组织机构中已经不再感兴趣了（他们或者辞职，或者被解雇了）。

这突出了REST和SOAP之间的一个主要区别，如果人们混淆了这两种方式的意图，就会导致困惑。REST是关于信息管理的，而不一定是通过URL来调用任意的行为。当人们开始挠头思考4个动词是否足够完成他们想做的事情时，他们想的也许不是信息，而是在想调用的行为。如果你打算通过URL来实现RPC，你也可以使用SOAP。如果你将重要的业务概念作为可以寻址的信息资源，并且可以在不同环境下进行操作或表示为不同的形式，那么你就在利用REST的长处，可能看到我们在Web上看到的某些好处。即使后台系统使用SOAP来满足请求，你也可以设想RESTful接口带来的好处。提供这样的地址不仅让用户能够“在数据上冲浪”，还可以引入缓存结果的能力，消除变更WSDL协议所带来的某些痛苦。客户端会通过一些逻辑连接，再翻译成SOAP消息，然后生成响应结果。响应的内容可以从返回的信息中抽取出来。我们根本不需要宣传这一事实，就能够在这个过程中实现架构迁移的策略。

正如图5-5所示，同样名称的资源可能在不同环境下返回不同的物理格式，却保留相同的标识符。我们可以设想，某种类型的公司报表在信息空间中的组织方式支持按时间浏览（例如，先按年再按月）。假设只有一种类型的报表，那么*http://server/report/2008/02*就是一个相当不错的、可以长期使用的名称。将来任何时候都不会改变这一事实：我们有一份2008年2月的报告。我们可能在某种场景下需要XML形式的数据，在另一场景下需要Excel表格，或者渲染成JPEG图像包含在汇总报告中。我们不希望在每种场景下使用不同的名称，所以我们通过内容协商来指定我们的格式偏好。面向资源的引擎需要知道如何响应一种请求类型，但这是很容易支持的。将来可能出现一些现有客户端都不支持的数据格式。已有的客户端不会需要修改，因为我们会在服务器上添加支持，然后某些其他客户端会利用它。这种面对变化的弹性是Web设计中固有的，也是我们在企业系统中希望实现的。客户端和服务器可以在解析过程中，针对命名的资源协商一种特定的格式。

这让同样名称的资源在不同环境下表现为不同的结构（例如，在中间层使用XML，在浏览器中使用JSON）。针对每种格式，如果需要，结构化的格式可以由服务器实现缓存。

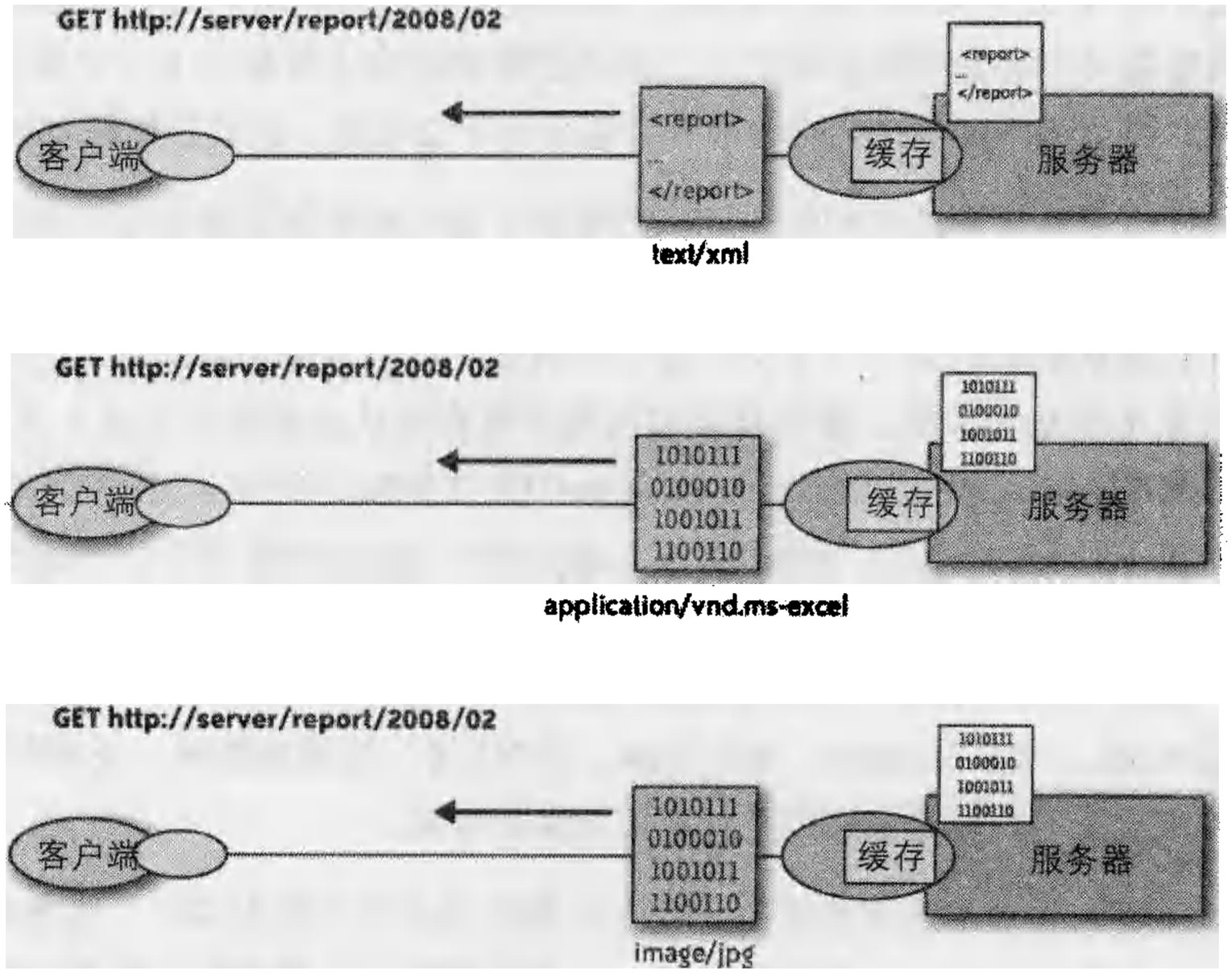


图5-5：在面向资源的环境中协商内容

除了支持在解析请求的上下文中选择物理表示形式之外，我们还可以支持服务根据用户的身份和使用的应用程序等信息来决定返回多少引用数据。我们可以想象这样一种场景：呼叫中心的座席人员需要通过一个相关的应用来访问敏感的信息，以解决问题。这些信息可能包括社会保险号码、信用卡号码（可能只是最后4位数字）、家庭地址等。这是一个特殊的业务，需要允许座席人员访问这些信息，所以我们可以用一个声明式的策略来支持这种情况。如果同样的雇员在不同的环境下使用不同的应用程序（也许是一个市场分析软件），在业务上就不太可能需要访问这些敏感信息，虽然我们可能仍然需要查找同样的客户，访问她的统计信息和购买历史。在这种情况下，上下文环境将不允许访问敏感数据，我们将强制通过一个自化的过滤过程来移除或加密敏感信息。采用哪种方式取决于数据需要发送到哪里。加密的数据要求取得密钥，这会带来额外的管理负担。我们很容易移除这些敏感数据，并在需要的时候将它们放在不同的解析上下文里。

管理单点访问控制对于传统的企业架构来说可能不是什么大问题。但是，考虑到不断增加的工作流，显式的、模型化的业务处理过程以及诸如此类的东西，我们有充分的理由认为一个应用程序的用户会需要在多种情况下调用一项功能或一个服务。如果我们在系统之间传递实际的数据，应用程序的开发者就有责任知道在跨应用边界时的访问控制问

题。如果我们只是传递数据的引用，那么最初的应用就不再需要负责，我们就可以确保实现信息驱动的、集中式的访问控制。许多现有的SOA系统根据身份或角色来限制访问权限，但它们基本上不支持对服务之间传递的具体数据的限制。这种局限性也是传统Web服务既复杂又不够安全的原因之一。访问控制策略应该在某个上下文环境中应用于行为和数据，但如果不能够命名这些数据并包含上下文信息，这就变得非常困难。

当人们开始研究面向资源的架构时，他们考虑到了通过链接暴露敏感信息的问题。在某种程度上说，在透明的查询之后返回大块的数据似乎更安全。他们很难从上下文中分离数据标识和数据解析的行为。上下文中包含的信息足够决定是否要向特定的用户提供信息。它与请求本身是无关的，要符合组织机构中现有的认证和授权系统。从HTTP基本认证，到IBM Tivoli Access Manager，到OpenID或其他联合身份系统，都可以用来保护数据。我们可以检查谁可以访问什么数据，通过单向或双向SSL来加密传输数据，防止窃听。可以寻址并不等于安全性脆弱。实际上，传递引用的策略比传递数据更安全、伸缩性更好。面向资源的方式不会因为它没有复杂的安全特性（如XML加密、XML签名、XKMS、XACML、WS-Security、WS-Trust、XrML等）变得更脆弱，反而可能更安全，因为人们可以真正明白威胁模式以及如何应用保护策略。

当我们面对需要说明满足法规要求这一令人生畏而又非常严肃事实时，这些思想就变得非常的重要。信用卡公司、医疗监控机构、公司治理审计者等组织机构会对公司的审计动真格，要求证明只有在工作中必须访问这些敏感信息的雇员才能看到这些信息。即使你的公司是满足要求的，但如果很难说明这一点（“首先，看系统中的这部分日志，然后跟踪流过这些中介的消息，它在这里被选取并处理成一个查询，你可以在另一个日志中看到这一点……”），那么审计过程的代价会变得巨大。针对逻辑引用解析来使用声明式的访问控制策略，可以明确谁、何时知道什么（而且容易遵守）。

5.5 数据驱动的应用

当组织机构摆脱了数据寻址的麻烦之后，就可以得到一些好处，除了可以让后台系统缓存结果之外，还可以通过谨慎的方式迁移一些新技术。具体来说，我们可以引入一些全新的数据驱动的应用程序和集成策略。当我们可以对数据命名，并通过应用友好的方式来请求数据时，我们就很容易实现一些数据查看、商务智能、知识管理，让大多数分析师看到它的时候感到异常兴奋。Simile项目（注14）是W3C和MIT CSAIL小组的合作项目，他们已经做了大量的工作来展示这些思想，展示真正能够带来多少兴奋。

请考虑这样一种情况：根据网站上的访问量和销售来追踪各种市场策略的效果。我们可能需要从电子表格、数据库和来自Web分析软件的一些日志文件和报表中取得信息。虽

注14： *http://simile.mit.edu.*

然将这些东西结合起来不需要像造火箭那样，但这的确需要不小的工作量来发现、请求、转换和发表这些结果。如果只是生成一份电子表格汇总，然后通过邮件分发它，那么我们实际上在将来需要查看这些结果时，就需要在我们的大堆文件中翻找。采用CMS或其他的文档管理管理系统只会增加得到结果所需的时间。无论生成这些报告的频度如何，我们每次都需要重复这些过程。

在面向资源的架构中，我们可以为每项数据元素确定地址，并请求JSON格式的数据，然后在基于浏览器的环境中使用它。源自Simile的Exhibit项目（注15）有一个Timeline视图（注16），差不多为我们提供了这种功能。花一点工夫将Excel表格转换成JSON对象，我们就有了一个可复用的环境，有了它之后，我们就能在几秒钟里组合并发布这些市场报告。接下来你会想到同样的基础设施可以将其他格式的数据放在一起，很容易地实现不同类型的分析和报表，这样你就会意识到Web和可寻址的数据的价值。这种环境正在企业中出现，如果你的组织机构还不能够这样容易地组织数据，那么它应该做到这一点。

5.6 应用面向资源的架构

最近，我通过重新架构我公司的Persistent URL（PURL）系统，创建一个面向资源的系统。原来的PURL（注17）是在大约15年前实现的。它是从Apache 1.0中分支出来的，用C语言写成，反映了当时的水平（注18）。它从那时起就成了Internet基础设施的一个稳定组成部分，但它逐渐显现老态，需要现代化，特别是需要支持W3C TAG的303建议以及大量的使用。大量的数据可以通过网页或简单的CGI-bin脚本来访问，因为在那时候，浏览器似乎像是唯一要面对的客户端。随着我们逐渐认识到语义网、生命科学、出版业和类似的一些社区需要长期不变的、无歧义的标识符，我们就知道是时候来重新考虑架构，并让它对人和软件更有用了。

PURL系统的设计目的是缓解好的名字和可解析的名字之间的紧张关系。任何在Web上发布过信息的人都知道，当内容被移动时，链接就不能使用了。Persistent URL的思想就是用一个好的、符合逻辑的名称映射到可以解析的位置。例如，一个PURL可以定义从*http://purl.org/people/briansletten*指向*http://bosatsu.net/foaf/brian.rdf*并返回一个303来表示“参见”响应。我不是一个网络可寻址的资源，但是从我“朋友的朋友（FOAF）”文件（注19）可以找到关于我的更多信息。我可以将这个PURL传给所有想链接到我的FOAF文件的人。如果我转到了其他公司，我可以更新PURL，让我的FOAF文件指向一

注15：*http://simile.mit.edu/exhibit.*

注16：*http://simile.mit.edu/timeline* .

注17：*http://purl.org.*

注18：此代码库构成非常成功的服务的基础TinyURL (*http://tinyurl.com*) service.

注19：*http://foaf-project.org.*

个新的位置。所有现有的链接将继续有效，它们只是通过303指向了新的位置。这个过程如图5-6所示。PURL服务器实现了W3C技术架构小组（TAG）的建议，即响应码303可以用于提供非网络可寻址的资源的更多信息。

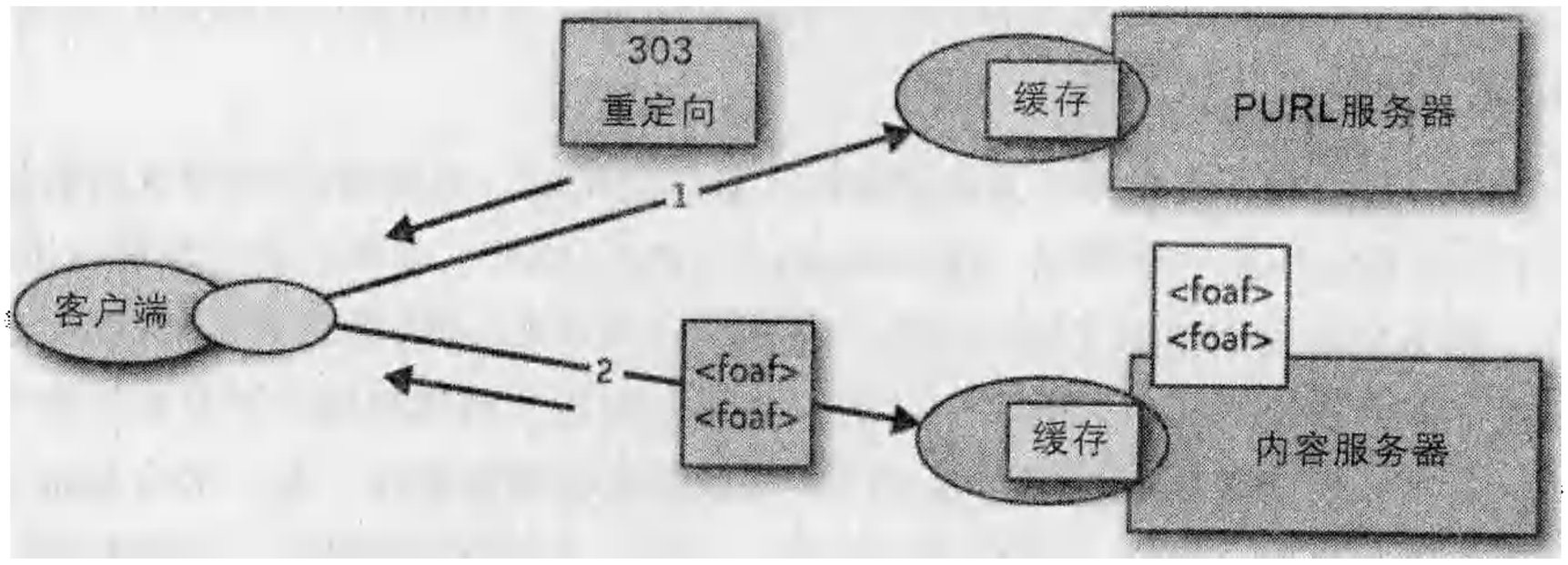


图5-6：PURL的“参见”重定向

除了支持PURL重定向，我们还想将PURL系统中的每个主要数据部分作为一个可寻址的信息资源。这不仅简化了与用户接口的交互，也支持潜在可能的数据复用，也许超出我们原本的计划。对资源的操作要求所有者凭证，但所有人都可以取得一个PURL的定义。对点击*http://purl.org/employee/briansletten*这样的PURL有一个直接的解析过程（它将导致303重定向），也有一个过程处理PURL资源的间接RESTful地址（*http://purl. org/admin/purl/employee/briansletten*），它将返回该PURL的定义，目前看起来是这样的：

```
<purl status=" 1" >
    <id>/employee/briansletten</id>
    <type>303</type>
    <maintainers>
       <uid>brian</uid>
    </maintainers>
    <seealso>
       <url>http://bosatsu.net/foaf/brian.rdf</url>
    </seealso>
</purl>
```

PURL服务器的客户可以在数据定义上“冲浪”，作为发现有关PURL资源的一种手段，而不必真正解析它。不需要编写代码来取得这些信息。我们可以在浏览器中查看它，或通过curl在命令行访问它。这样，我们就可以设想编写一些shell脚本，利用来自信息源的数据，检查PURL是否指向了有效的资源，并返回合理的结果。如果没有，我们可以找到PURL的所有者，向账号相关的邮件地址发出一份邮件通知。可寻址、可访问的数据可以用于各种意想不到的设计、脚本、应用和桌面控件，因为这样做很容易，也很有用。

有趣的完整故事是，我们没有在最初的版本中支持JSON的请求格式，这使得AJAX用户接口变得复杂了。JavaScript的XML处理远不是人们所期望的。即使我们在内部使用XML

格式，我们也应该不怕麻烦，提供JSON格式让浏览器来处理。你可以相信我们很快就会弥补这个疏忽，但是我想在这里有必要突出这一点，如果我们一开始就做对，会很有好处。你不需要一开始就支持所有数据格式，但在今天，从XML和JSON开始是很好的。

作为一项有趣的补充说明，我们可以选择几种容器和工具来实现前面介绍的这种架构。所有响应HTTP请求的软件都可以作为PURL服务器。这体现了RESTful接口和面向资源架构的浅显而有用的概念，如图5-7所示。任何Web服务器或应用服务器都可以作为简单的面向资源的引擎。逻辑HTTP请求被解释为对Servlet、Restlet或类似的可寻址功能的请求。

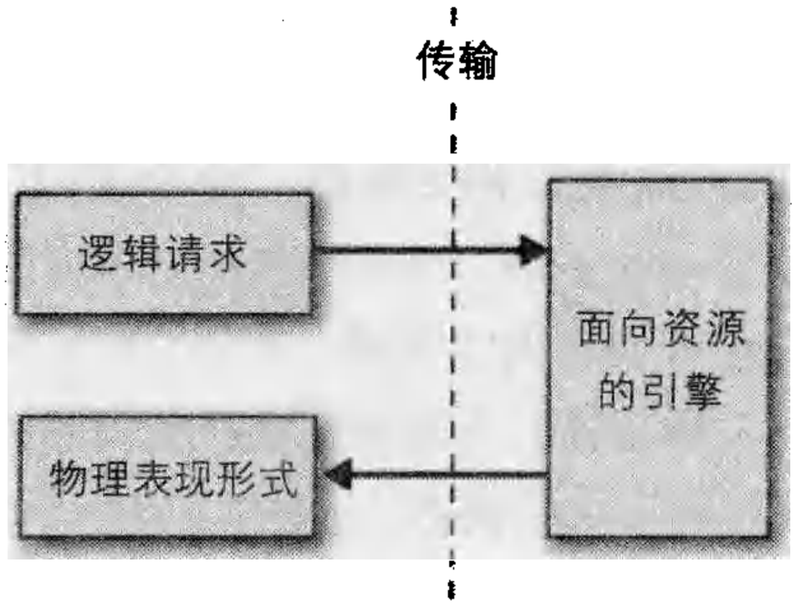


图5-7：简单的面向资源的架构

我们选择使用NetKernel作为这个架构的基础，因为它是面向资源的架构的化身，具有双重许可证，允许用于开源和商业项目中。利用不同的表示方式来实现各层之间的逻辑耦合的思想，被设计进了软件架构中，提供了类似的灵活性、可伸缩性和简单性等好处。层和层之间的联系是通过异步解析、逻辑名称来实现的。面向资源的架构的这种深刻思想如图5-8所示。NetKernel是一个有趣的软件基础设施，因为它在内部利用了资源逻辑相连的概念，所以HTTP逻辑请求转成其他逻辑请求。这种架构在运行时软件环境中反映了Web的特性。

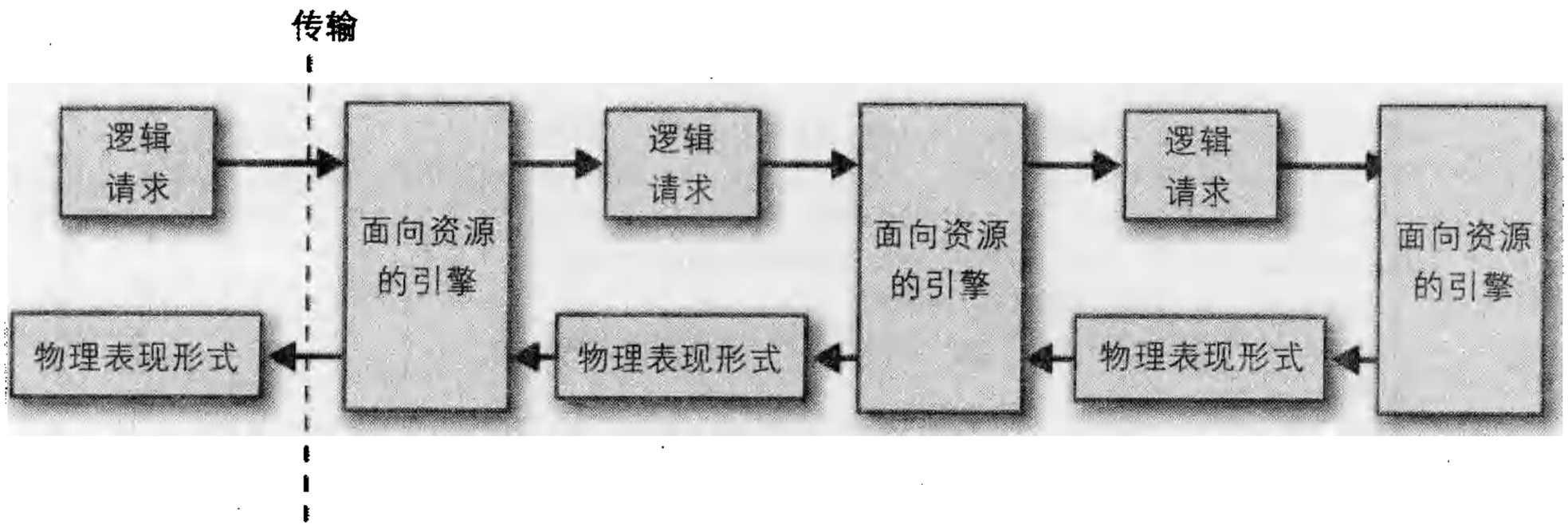


图5-8：深层的面向资源的架构

外部URL *http://purl.org/employee/briansletten* 通过通过地址重写被映射到一段名为“访问者”（accessor）的程序（注20）。访问者存在于模块中，这些模块输出公开的URI定义，代表了它们负责响应的地址空间。这里的方便之处在于，我们能够在模块的新版本中大幅度地改变实现技术，同时只要修改重写规则，让它指向新的实现就可以了。只要返回的是兼容的响应结果，客户端什么都不需要知道。我们可以在现代的面向对象语言中利用接口近似实现这种灵活性，但这还是要求我们在“物理上”耦合接口的定义。通过只包含逻辑的绑定，我们虽然仍旧需要支持现有客户端的期望，但我们不再与任何实现细节耦合在一起。我们不仅可以在Web上通过URI的通信时看到这种价值，而且在本地运行的软件中也是这样。

在内部，我们使用与请求的方法类型相关联的“命令模式”（注21）来实现访问者。HTTP GET方法被映射到GetResourceCommand方法，该方法不保持状态。在收到请求时，我们从映射表中找到命令并向它发出请求。REST的无状态风格确保了请求中已包含回答请求所需的全部信息，所以我们不需要在命令实例中保持状态。我们可以在后面的代码中通过上下文实例来访问请求的状态。这些代码对于Java开发者来说显得很直白。我们在Java对象上调用方法，捕捉异常，取得返回值。重要的是要注意IURAspect接口的使用。从本质上说，我们不关心资源采取何种形式。它可以是一个DOM实例、一个JDOM实例、一个字符串，或一个字节数组。从我们的目的来说，这都不重要。基础设施会将它转换成一个字节流，再带上元数据，作为请求的响应结果发出。如果我们希望基础设施支持特定的格式，我们就会直接向它请求那种格式。这种声明式的、面向资源的方式有助于大幅度减少用于操作数据的代码的数量，让我们能利用正确的工具做正确的事情：

```
if(resStorage.resourceExists(context, uriResolver)) {
    IURAspect asp = resStorage.getResource(context, uriResolver);

    // Filter the response if we have a filter
    if (filter!=null) {
    asp = filter.filter(context, asp);
    }

    // Default response code of 200 is fine
    IURRepresentation rep = NKHelper.setResponseCode(context, asp, 200);
    rep = NKHelper.attachGoldenThread(context, "gt:" + path , rep);
    retValue = context.createResponseFrom(rep);
    retValue.setCacheable();
    retValue.setMimeType(NKHelper.MIME_XML);
} else {
    IURRepresentation rep = NKHelper.setResponseCode(context,
            new StringAspect(" No such resource: "
```

注20：*http://docs.1060.org/docs/3.3.0/book/gettingstarted/doc_intro_code_accessor.html.*

注21：*http://en.wikipedia.org/wiki/Command_pattern.*

```
                        + uriResolver.getDisplayName(path)), 404);
        retValue = context.createResponseFrom(rep);
        retValue.setMimeType(NKHelper.MIME_TEXT);
    }
```

当GET请求发出后，大部分信息资源会返回200。显然，PURL超越了这种行为，可以返回302、303、307、404等。如果我们看看resStorage.getResource()方法的面向PURL的实现，就会知道这种的面向资源的趣事。

```
INKFRequest req = context.createSubRequest("active:purl-storage-query-purl");
req.addArgument("uri", uri);
IURRepresentation res = context.issueSubRequest(req);
return context.transrept(res, IAspectXDA.class);
```

简而言之，我们通过“active:purl-storage-query-purl”这个URI发出一个逻辑请求，带有“ffcpl:/purl/employee/briansletten”作为参数。忽略掉这个不寻常的URI格式，这只是在NetKernel中用于表示内部的请求。我们不知道什么代码会被调用，以取得要求格式的PURL，我们并不关心它。在面向资源的环境中，我们只需说：“响应这个URI的程序将生成响应结果”。现在我们可以自由地在设计阶段先向客户端提供一些静态文件，然后在构建阶段利用基于Hibernate映射到关系数据库。我们可以通过重写对“active:purl-storage-query-purl”这个URI的响应来实现这种转换。客户端代码不需要知道这种区别。如果我们改变了PURL的解析方式，从本地持久层变为远程获取，客户端代码也可以不必关心。这就是我们已经提到的好处，在一个强大的软件环境中，面向资源的企业级计算的思想被具体化了。

我们不仅用这种方式实现了层与层之间的松耦合，而且我们在这个环境中也享受到了幂等的、无状态的请求所带来的好处。取得PURL定义的早期代码片断逐渐在内部扁平化，变成了对“active:purl-storage-query-purl+uri@ ffcpl:/purl/employee/briansletten”这个URI的异步请求。正如我们前面所讨论的，这变成了一个复合散列键，代表了查询持久层而得到的结果。即使我们对调用的代码一无所知，NetKernel仍然能够缓存这些结果。这就是我前面提到过的架构上的记忆化。实际的过程稍稍有点差异，但在概念上，就是这样的。如果其他人试图从内部或通过HTTP RESTful接口来解析同样的PURL，我们就可以从缓存中拿出结果。尽管这一点可能不能打动在Web页面上已经建立了缓存机制的人，但如果你再仔细想想，就会发现这实际上是很令人佩服的。所有可能的URI请求都以这种方式进行缓存，不论我们是打算从磁盘上读取文件、通过HTTP取得文件、通过XSLT文件转换一份XML文档，或是将PI（圆周率）计算到10000位小数。每次调用都得到逻辑的、无状态的、异步的结果，每次结果都可能缓存起来。这种面向资源的架构方式让我们的软件可伸缩、有效率、支持缓存，并且是通过统一的、逻辑的接口来工作的。这导致了本质上更健壮、更灵活、可伸缩的架构，就像Web一样，原因也是一样的。

5.7 结论

面向资源的架构方法很优雅地实现了一些折中。一方面，对于传统的方法来说，这些方法可能看起来有些奇怪，而且没有尝试过。如果人们关心自己的简历，就会希望停留在尝试过的、真正在使用的方法。另一方面，对于那些研究过Web和它的基本组成模块的人来说，它很有意义，代表了人们设想和实现过的最大、最成功的网络软件架构。一方面，它要求一种完全不同的思考方式。另一方面，它支持一种强大的机制，包装并复用已有的代码、服务和基础设施，为它们提供逻辑命名接口，对所有形式的交互都不透露实现细节。我们可以自由地调整服务器端的技术而不会影响原有的客户端。随着时间的推移，我们可以为同样的数据提供新的结构形式支持。我们可以迁移后端的实现，不会影响我们的客户。另外，这些设计选择导致了伸缩性强、易于缓存、信息驱动的访问易控制、好的法规兼容性等特点。

软件开发者通常不关心数据，他们关心算法、对象、服务和其他类似的结构。对于J2EE、.NET和基于SOAP的架构，我们已经拥有一些相当具体的推荐蓝图和技术。遗憾的是，大多数已有的蓝图没有把信息作为一等公民。它们让我们局限于某些具体的绑定，使我们难以在不影响已有客户端的情况下进行变更。这是我们沿用多年的脚踏式水车，业务部门对此已感到疲倦。Web服务本应该是一种退出策略，但不恰当的抽象和过于复杂的边界条件用例，使得整个过程成为了对Web服务完全不满意的经历。现在是时候从以软件为中心的架构中走出来，开始关注信息及其流动了。我们仍会利用我们知道和喜爱的工具来编写软件，只是这些工具不会成为架构绑定的焦点。

面向资源的方法在业务部门和支持它们的技术部门之间提供了引人注目的桥梁。以信息为中心的视图和我们的各系统之间连接的方式，为我们提供了真正的效率和业务价值主张。我们不需要根据供应商们的伟大思想从头干起，我们可以从Web上学到有价值的经验，了解它的架构方式引发的重要特征。架构是有人居住的雕塑，我们不得不在一段时间内承受这些选择的影响。我们应该抓住机会，让功能性、美和弹性渗入到架构之中，使我们在架构中的生活变得更舒适。

	原则与特性		结构
√	功能多样性	√	模块
	概念完整性	√	依赖关系
√	修改独立性		进程
√	自动传播	√	数据访问
	可构建性		
√	增长适应性		
√	熵增抵抗力		

第6章

数据增长：Facebook平台的架构

Dave Fetterman

给我看你的流程图而藏起你的表，我将仍然莫名其妙。如果给我看你的表，那么我将不再需要你的流程图，因为它们太明显了。

——Fred Brooks，《The Mythical Man-Month》（人月神话）

6.1 简介

当前大多数计算机科学的学生将Fred Brooks的这句话理解为：“给我看你的代码而藏起你的数据结构……”信息架构师坚信，处于大多数系统核心的是数据，而不是算法。随着Web的兴起，用户产生和消费的数据比以往更加推动了信息技术的使用。Web用户不会去接触QuickSort（快速排序）。他们会访问一个数据仓库。

这些数据可以是通用的，如一本电话簿；也可以是私有的，如一个在线仓库；也可以是个人的，如一个博客；也可以是开放的，如当地的天气情况；还可以是严格保护的，如在线银行记录。在任何情况下，Web呈现的几乎所有面对用户的功能归根结底都是提供一个界面，访问站点专有的一组核心数据。这些信息构成了几乎所有网站的核心价值，不论它是由顶级员工研究团队创建的还是由世界各地的用户创建的。数据推动了用户喜欢的产品，所以架构师围绕数据创建了其余的传统“*n*层”软件栈（逻辑层与显示层）。

这个故事讲的是Facebook的数据，以及它如何与Facebook平台的创建一起发展。

Facebook（*http://facebook.com*）是一个很有用的围绕数据建立架构的例子，包括用户提交的个人关系映射表、传记信息，以及文本或其他媒体内容。Facebook的工程师在构建

站点其余部分的架构时，关注的是显示和操作这些社会关系数据。这个站点的大多数业务逻辑与这些社会关系数据密切相关，诸如对各种页面的流程和访问模式，搜索的实现，查看新闻内容，以及对内容应用可见性规则。对于用户来说，这个站点的价值直接来自于他和与他有关的人对该系统所贡献的数据的价值。

“Facebook社会关系网站”在概念上是一个标准的*n*层栈，用户的请求会从Facebook的内部库中取出数据，然后通过Facebook的逻辑进行转换，最后通过Facebook的界面输出。Facebook的工程师意识到这些数据的用处超过了这些容器的限制。Facebook平台的创建显著地改变了Facebook数据访问系统的形态，它包含的愿景远远超出了*n*层栈的分离功能，目标是以应用的形式来集成外部的系统。利用居于架构中心的用户社会关系数据，该平台开发了一组Web服务（Facebook平台应用编程接口，或Facebook API）、一门查询语言（Facebook查询语言，或FQL），以及一种数据驱动的标记语言（Facebook标记语言，或FBML），目的是将应用开发者的系统与Facebook的系统结合在一起。

随着某些数据集越来越广泛地提供出来，而且用户要求跨越多个网和桌面应用来统一使用他们的数据，阅读本章的架构师可能会发现自己已经是这样一个平台的消费者，或者围绕着自己站点的数据建立了类似的平台。本章将向读者展示Facebook以一种受控的方式向外界开放数据的过程，跟随数据演进的每一步的架构选择，以及调和数据开放与渗透在社会关系系统中独特的隐私需求的过程。它包括：

- 促进这些类型的集成。
- 将数据功能从内部栈调用移到外部可见的Web服务上（Facebook API）。
- 授权访问这个Web服务，注意保持这个社会关系系统的隐私性。
- 创建一种数据查询语言，减轻这个Web服务的新客户端的负担（Facebook FQL）。
- 创建一种数据驱动的标记语言，将应用的显示集成回Facebook，同时也支持使用其他方式不能访问的数据（Facebook FBML）。

当我们将应用的架构从分离的栈进行了足够的演进之后：

- 创建一些技术来弥补Facebook体验与外部应用体验之间的差异。

对于数据平台的使用者，本章展示了我们所做的设计决定和这些决定背后的理由。用户会话、身份认证、Web服务和各种处理应用逻辑的方式等概念将不断重复出现，它们是Web上所有这些类型的平台的主题。理解它们背后的思想为数据架构提供了巨大的实践机会，而且考虑到这些平台制造者将来可能创建的功能和形式，这种理解也相当重要。

我们鼓励数据平台制造者心里想着自己的数据集，然后从Facebook开放其数据模型的方式中学习。某些设计选择和折中可能只适合Facebook，或只适合处理有隐私保护的社会

关系数据，可能不完全适用于给定的数据集。但不管怎样，在每一步我们都给出了一个实际的问题、一个数据驱动的解决方案，以及该解决方案的高层实现。对于每个新的解决方案，我们基本上会创建一个新的产品或平台，所以在任何时候我们都必须让这个新产品符合用户的预期。反过来，我们会伴随每一步的演进创造一些新技术，有时候会改变围绕应用的Web架构。

Facebook平台的开源版本可以从*http://developers.facebook.com/*获得。就像这个版本一样，本章的代码是用PHP写的。请随意查看，不过请注意，出于清晰性的考虑，这里的代码是缩写过的。

我们从这些类型的集成的动机开始，通过一个例子来讲解一个“外部的”应用逻辑和数据（一个书店）、Facebook的社会关系数据（用户信息和朋友关系），以及它们的集成。

6.1.1 某些应用核心数据

Web应用，即使是不提供也不使用任何的数据平台，基本上仍然是由它们内部的数据来驱动的。以*http://fettermansbooks.com*为例，它一是个假想的网站，提供书籍方面的信息（如果用户感兴趣，它可能也提供购买这些书的功能）。这个站点的功能可能包括可查找的库存索引、关于每件产品的基本信息，以及用户每本书作出的评论。访问这些具体的信息构成了这个应用的核心，驱动了架构的其他部分。该站点可能使用Flash和AJAX技术，支持通过移动设备来访问，并提供一个一流的用户界面。然而*http://fettermansbooks.com*存在的根本理由是让访问者能够利用某些方法得到示例6-1中这样的核心映射关系。

例6-1：书籍数据映射的例子

```
book_get_info : isbn -> {title, author, publisher, price, cover picture}
book_get_reviews: isbn -> set(review_ids)
bookuser_get_reviews: books_user_id -> set(review_ids)
review_get_info: review_id -> {isbn, books_user_id, rating, commentary}
```

所有这些最终都实现为类似简单集合的东西，能够从一个经索引的数据表中取出。这样的书籍站点如果要有存在的价值，可能还会实现其他一些不太简单的功能，如例6-2中的简单“查找”。

例6-2：简单查找映射

```
search_title_string: title_string -> set({isbn, relevance score})
```

这些功能中包含的每个键值通常都会表现为*http://fettermansbooks.com*上的一个或多个页面——有一组特有的逻辑围绕着这批数据，通过一种特有的方式显示出来。例如，要查看评论者*X*提交的一些评论，*http://fettermansbooks.com*的用户可能会被引向页面 *fettermansbooks.com/reviews. php? books_user_id=X*，或者要看ISBN号为*Y*的某本书的所有信息（包括所有对个人评论页面的链接），用户会被引向页面 http:// fettermansbooks. com/ *book.php?isbn=Y*。

像*http://fettermansbooks.com*这样的站点有一个特点是值得注意的，即几乎所有数据都对所有用户开放。它在book_get_info这样的映射中生成所有的内容，帮助用户发现有关某本书的尽可能多的信息。这对于一个卖书的站点可能是好事，但在接下来的使用社会关系数据的例子中，可见性限制决定了数据访问层的许多架构考虑。

6.1.2 一些Facebook核心数据

随着所谓“Web 2.0”的网络技术逐渐流行，数据在系统中的核心地位就变得更明显了。Web 2.0展现的核心主题就是它们是数据驱动的，用户本身提供了绝大部分的数据。Facebook像*http://fettermansbooks.com*一样，主要由一组核心数据映射构成，它们驱动着网站的观感和功能。这些Facebook映射的极端精简集合看起来如例6-3所示。

例6-3：社会关系数据映射示例

```
user_get_friends: uid -> set(uids)
user_get_info: uid -> {name, pic, books, current_location,...}
can_see: {uid_viewer, uid_viewee, table_name, field_name} -> 0 or 1
```

这里的uid指的是（数字化的）Facebook用户标识符，从user_get_info返回的info指的是用户的描述信息（参见Facebook开发文档中的users.getInfo），可能包含了用户最喜欢的书籍名称，因为他们曾在*http://facebook.com*上输入过。这个系统从核心上来看与*http:// fettermansbooks.com*区别不大，只有中心数据不同，因此站点的功能也不同，这些功能围绕着用户与其他用户的联系（“朋友”），用户的内容（“描述信息”），以及内容的可视法则（“can_see”）。

这个can_see数据集是很特殊的。Facebook对于用户生成的数据有一个非常核心的隐私概念，即用户*X*查看用户*Y*的信息的业务规则。这种数据从不直接可见，但它驱动了一些重要的考虑，当我们查看外部应用的逻辑、数据与Facebook的逻辑、数据集成的例子时，会看到这些考虑反复出现。就其本身而言，Facebook到处使用这种数据集令它与*http://fettermansbooks.com*这样的站点区别开来。

Facebook平台和其他社会关系平台认识到这种社会关系映射是有用的，这种用处不仅体现在*http://facebook.com*这样的站点内部，也体现在与*http://fettermansbooks.com*这样的站点功能进行集成时。

6.1.3 Facebook的应用平台

对于*http://fettermansbooks.com*和*http://facebook.com*的共同用户来说，此时因特网应用的图景如图6-1所示。

在一般的*n*层架构中，应用将输入（对于Web来说，就是GET、POST和cookie信息的集合）映射为对原始数据的请求，这些原始数据可能存在于数据库中。它们被转换为内存中的数据，并通过一些业务逻辑进行智能化处理。输出模块将针对显示对这些数据对象进行转换，变成HTML、JavaScript、CSS等。这里，在图的顶部，是运行在基础设施之上的应用程序*n*层栈。在应用出现在Facebook平台之前，Facebook完全运行在同样的架构上。重要的是，在两个架构中，业务逻辑（包括Facebook的隐私）实际上都是根据一些规则来执行的，这些规则建立在系统的某些数据组件之上。

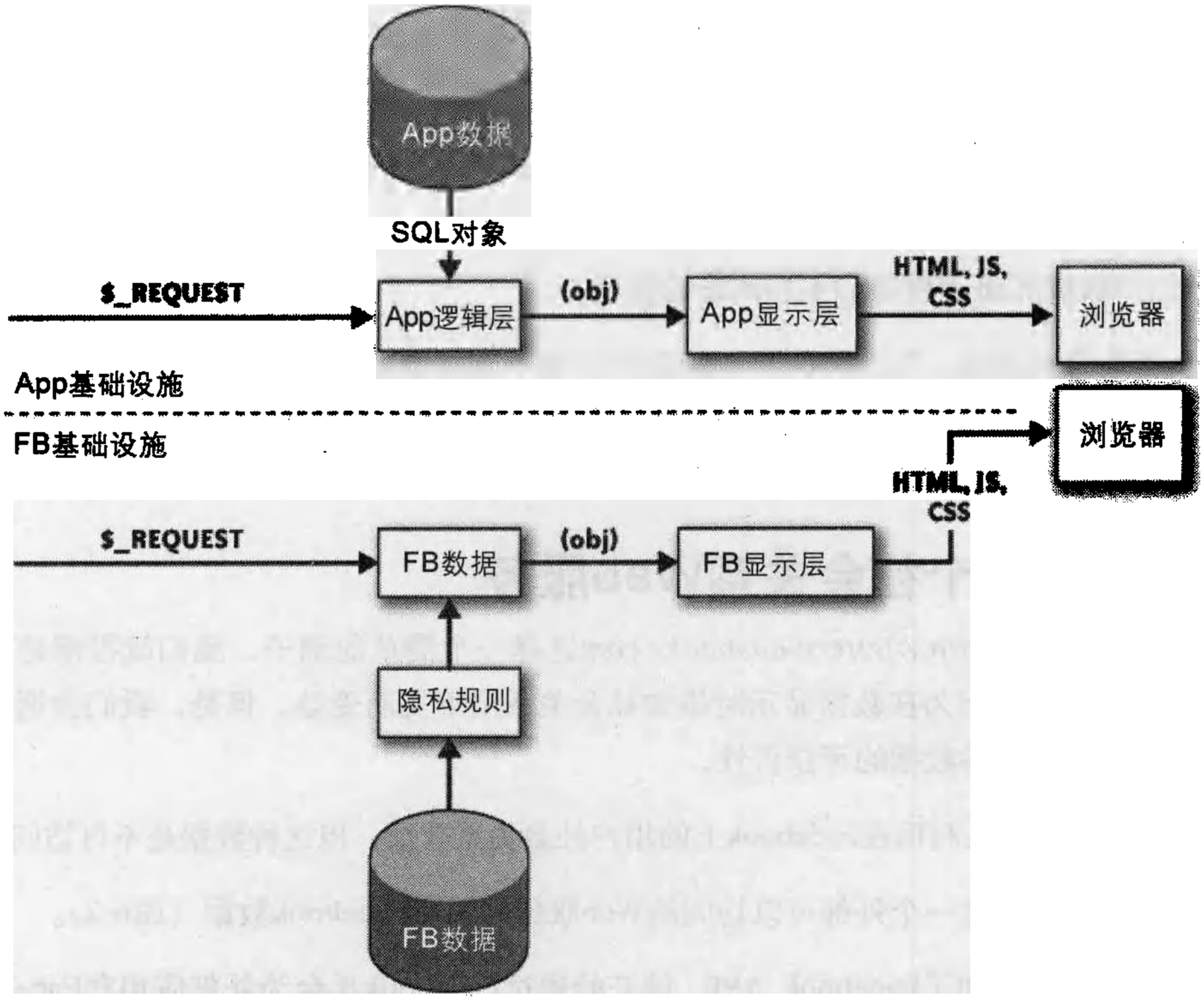


图6-1：分离的Facebook和*n*层应用栈

更大量的相关数据意味着业务逻辑可以提供更多个人定制的内容，所以在*http://fettermansbooks.com*（或其他应用）上浏览书籍，写书评、阅读或购买的体验，会被来自Facebook的用户社会关系数据加强和放大。具体来说，显示朋友的书评、期望清单和购买情况将有助于用户的购买决定，发现新的书籍，或强化与其他用户之间的联系。如果Facebook的内部映射`user_get_friends`可以由*http://fettermansbooks.com*这样的其他外部应用访问，就会为这些原本分离的应用提供强大的社会关系上下文，让应用程序不需要创建它自己的社会关系网络。所有这种类型的应用都可以与这种数据进行很好的

集成，因为开发者可以将这些核心Facebook映射应用于无数其他Web应用，用户在这些应用里提供或消费内容。

Facebook平台的技术通过在社会关系网络和数据架构方面的一系列改进，实现了这一点：

- 应用可以通过Facebook平台的数据服务来访问有用的社会关系数据，为外部的Web应用、桌面操作系统应用和其他设备上的应用提供社会关系上下文。
- 应用可以通过一种名为FBML的数据驱动标记语言来实现显示，在*http://facebook.com*的页面上集成他们的应用体验。
- 通过FBML所要求的架构改变，开发者可以使用Facebook平台的cookie和Facebook JavaScript（FBJS），从而让应用出现在*http://facebook.com*上所需的改动最小。
- 最后，应用可以获得这些功能，同时不必牺牲隐私，也不必放弃对于Facebook为用户数据和显示提供的用户体验的期望。

最后一点是最有趣的。Facebook平台的架构并非一直是美丽的——它主要被看成是社会关系平台领域的先行者。大多数的架构考虑是为了创建统一可用的社会关系上下文，它体现了这样的阴阳关系：数据可获得性和用户隐私。

6.2 创建一个社会关系Web服务

回过头来看一看像*http://fettermansbooks.com*这样一个简单的例子，我们就很清楚大多数因特网应用都会因为在数据显示时添加社会关系上下文而受益。但是，我们会遇到一个实际的问题：这种数据的可获得性。

实际问题：应用可以利用在Facebook上的用户社会关系数据，但这种数据是不可访问的。

数据解决方案：通过一个外部可以访问的Web服务来提供Facebook数据（图6-2）。

为Facebook架构添加了Facebook API，就开始通过Facebook平台为外部应用和Facebook建立了关系，本质上为外部应用栈添加了Facebook数据。对于Facebook用户，当他显式地授权外部应用可以代表他获得社会关系数据时，这种集成就开始了。

例6-4展示了*http://fettermansbooks.com*的登录页面在没有Facebook集成的情况下可能的样子。

例6-4：书籍网站逻辑示例

```
$books_user_id = establish_booksite_userid($_REQUEST);
$book_infos = user_get_likely_books($books_user_id);
display_books($book_infos);
```

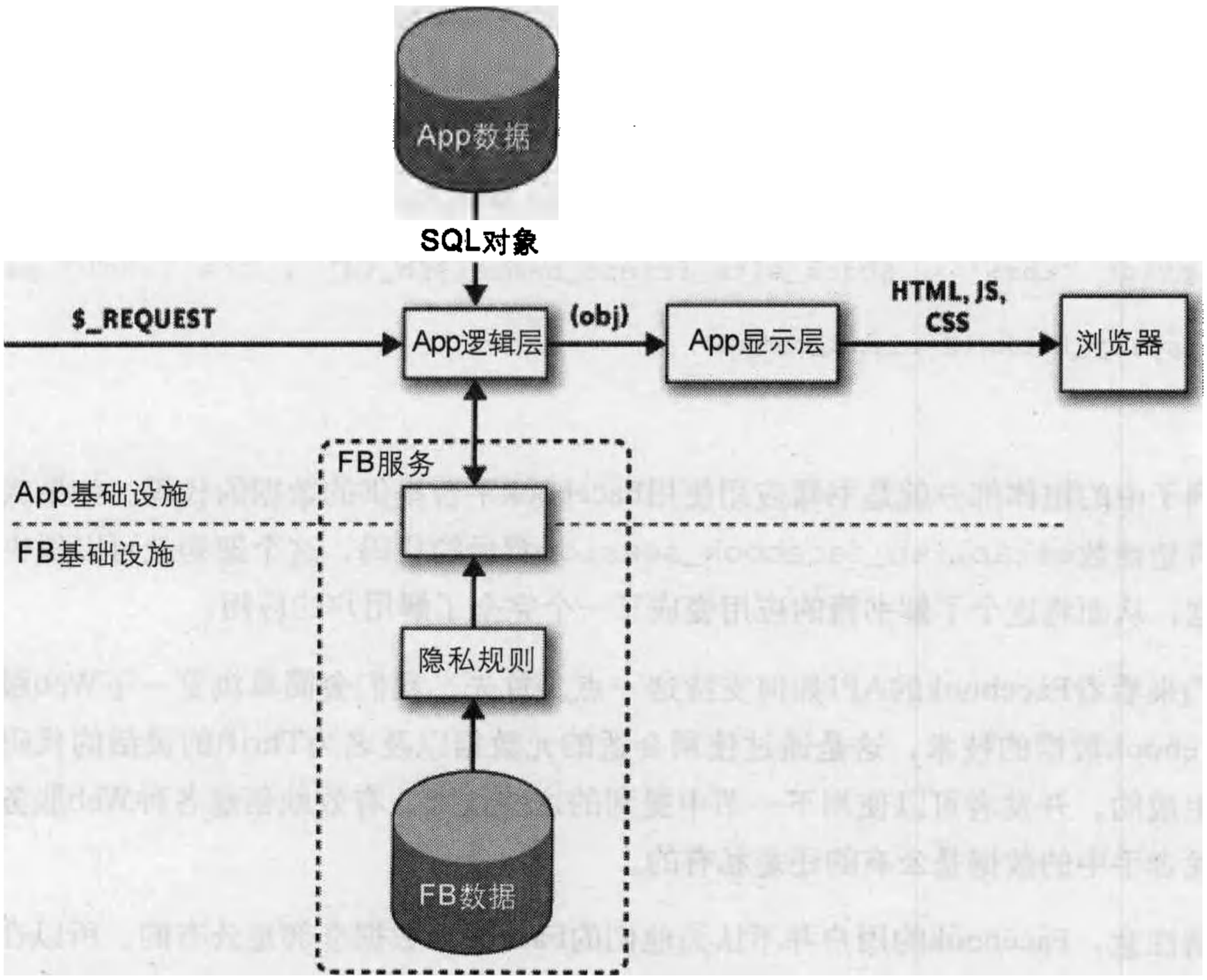


图6-2：应用栈通过Web服务使用Facebook数据

这个user_get_likely_books函数操作完全源自于这个书籍应用控制的数据，可能使用智能的关联技术来猜测用户的兴趣。

但是，假定Facebook为在其他站点的用户提供了两个简单的远程过程调用（RPC）方法：

- friends.get()
- users.getInfo($users, $fields)

通过它们，并添加从*http://fettermansbooks.com*的用户标识符到Facebook的用户标识符的映射关系，我们就可以为*http://fettermansbooks.com*上的所有内容添加社会关系上下文。请考虑这个针对Facebook用户的新流程，如例6-5所示。

例6-5：包含社会关系上下文的书籍站点逻辑

```
$books_user_id = establish_booksite_userid($_REQUEST);
$facebook_client = establish_facebook_session($_REQUEST,$books_user_id);

if ($facebook_client) {
  $facebook_friend_uids = $facebook_client->api_client->friends_get();
  foreach($facebook_friend_uids as $facebook_friend) {
    $book_site_friends[$facebook_friend]
```

```
        = books_user_id_from_facebook_id ($facebook_friend);
  }
  $book_site_friend_names = $facebook->api_client->
      users_getInfo($facebook_friend_uids, 'name');

  foreach($book_site_friends as $fb_id => $booksite_id) {
    $friend_books = user_get_reviewed_books($booksite_id);
     print "<hr>" . $book_site_friend_names[$fb_id] . "'s likely picks:
<br>";
    display_books($friend_books);
  }
}
```

这个例子中的粗体部分就是书籍应用使用Facebook平台提供的数据的代码。如果我们能够弄清楚函数establish_facebook_session背后的代码，这个架构就可以提供更多的数据，从而将这个了解书籍的应用变成了一个完全了解用户的应用。

让我们来看看Facebook的API如何支持这一点。首先，我们会简单浏览一下Web服务包装Facebook数据的技术，这是通过使用合适的元数据以及名为Thrift的灵活的代码生成器来生成的。开发者可以使用下一节中提到的这些技术，有效地创建各种Web服务，不论开发者手中的数据是公有的还是私有的。

但是请注意，Facebook的用户并不认为他们的Facebook数据全部是公有的。所以在技术概述之后，我们会探讨Facebook层面的隐私，这是通过平台API中的主要认证方式来实现的，即用户会话。

6.2.1 数据：创建一个XML Web服务

为了能够在一个示例应用中提供基本的社会关系上下文，我们已经建立了两个远程方法调用，即friends.get和users.getInfo。访问这些数据的内部功能可能存在于Facebook代码树的某个库中，为Facebook站点上的类似请求提供服务。例6-6展示了一些例子。

例6-6：社会关系映射示例

```
function friends_get($session_user) { ... }
function users_getInfo($session_user, $input_users, $input_fields) { ... }
```

我们接下来要创建一个简单的Web服务，将通过HTTP的GET和POST输入转换成对内部栈的调用，以XML的格式输出结果。在Facebook平台中，目标方法的名称以及它的参数是在HTTP请求中传递的，还包括一些与调用应用相关的证书（称为“api key”），与用户-应用对相关的证书（称为“用户会话key”），与请求实例本身相关的证书（称为请求“签名”）。我们稍后将在6.2.2节中讨论会话key。要服务一个针对http://api.facebook.com的请求，其大致过程如下：

1. 检查传递的证书（第6.2.2节），验证调用应用程序的身份，用户当前在该应用中

的授权，以及请求的可信度。

2. 将进入的GET/POST请求解释为带有相应参数的方法调用。

3. 对内部方法进行单次调用，将结果保存为内存中的数据结构。

4. 将这些数据结构转换成已知的输出格式（如XML或JSON）并返回。

创建外部可使用的接口，难度主要在于第2步和第4步。为外部使用者提供这些数据接口的一致维护、同步和文档是很重要的，手工打造一个代码框架来确保这种一致性则是一项无人赞赏而又耗时的工作。另外，我们可能需要将这些数据提供给多种语言编写的内部服务来使用，或者以不同的Web协议将结果提供给外部开发者，如XML、JSON或SOAP。

那么这里的优美解决方案，就是利用元数据来封装数据类型和描述API的方法签名。Facebook的工程师创建开源的跨语言进程间通信（IPC）系统，名为Thrift（*http://developers.facebook.com/thrift*），干净利落地实现了这个目标。

深入一步，例6-7展示了一个针对1.0版API的“.thrift”文件的例子，在这个版本里，Thrift包实现了这个API的大部分机制。

例6-7：通过Thrift对Web服务定义

```
xsd_namespace http://api.facebook.com/1.0/
/***
* Definition of types available in api.facebook.com version 1.0
*/
typedef i32 uid
typedef string uid_list
typedef string field_list

struct location {
 1: string street xsd_optional,
 2: string city,
 3: string state,
 4: string country,
 5: string zip xsd_optional
}

struct user {
 1: uid uid,
 2: string name,
 3: string books,
 4: string pics,
 5: location current_location
}

service FacebookApi10 {

 list<uid> friends_get()
  throws (1:FacebookApiException error_response),
      ;
```

```
 list<user> users_getInfo(1:uid_list uids, 2:field_list fields)
  throws (1:FacebookApiException error_response),
}
```

这个例子中的类型是原生类型（string）、结构（location、user）或泛型方式的集合（list<uid>）。因为每个方法描述都有精心设计类型的方法签名，定义复用的类型的代码就可以直接在任何语言中生成。例6-8展示了针对PHP的部分生成结果。

例6-8：Thrift生成的服务代码

```
class api10_user {
  public $uid = null;
  public $name = null;
  public $books = null;
  public $pic = null;
  public $current_location = null;


  public function __construct($vals=null) {
    if (is_array($vals)) {
      if (isset($vals['uid'])) {
        $this->uid = $vals['uid'];
      }
      if (isset($vals['name'])) {
        $this->name = $vals['name'];
      }
      if (isset($vals['books'])) {
        $this->books = $vals['books'];
      }
      if (isset($vals['pic'])) {
        $this->pic = $vals['pic'];
      }
      if (isset($vals['current_location'])) {
        $this->current_location = $vals['current_location'];
      }
     // ...
    }
   // ...
  }
```

返回user类型的所有内部方法都会创建全部需要的字段，结束的语句类似例6-9的样子。

例6-9：一致地使用生成的类型

```
return new api_10_user($field_vals);
```

例如，如果current_location（当前位置）出现在这个用户对象中，那么$field_vals['current_location']就会在例6-9的代码执行之前，被赋值为new api_10_user(...)。

字段的名称和类型本身实际上会生成XML输出所需的schema，以及相应的XML Schema文档（XSD）。例6-10展示了整过RPC过程实际输出的XML。

例6-10：Web服务调用的XML输出

```
<users_getInfo_response list="true">
 <users type="list">
  <user>
   <name>Dave Fetterman</name>
   <books>Zen and the Art, The Brothers K, Roald Dahl</books>
   <pic></pic>
   <current_location>
    <city>San Francisco</city>
    <state>CA</state>
    <zip>94110</zip>
   </current_location>
  </user>
 </users>
</users_getInfo_response>
```

Thrift生成类似的代码来声明RPC函数调用、序列化成已知的输出格式，并将内部的异常转化成外部错误代码。其他像XML-RPC或SOAP这样的工具集也提供这样一些好处，但可能需要更多的CPU和带宽开销。

使用像Thrift这样的漂亮工具有以下好处：

自动化类型同步

在user类型中添加“favorite_records”，或将uid转换成i64需要在所有使用或生成这些类型的方法中进行。

自动化绑定生成

所有读写类型的麻烦工作都不需要了，转换函数调用生成XML的RPC方法要求函数声明、类型检查和错误处理，这些都由Thrift自动完成。

自动化文档

Thrift生成公开的XML Schema文档，它将作为外界看到的无二义的文档，通常比在“手册”上看到的文档要好得多。这种文档也可以直接在一些外部工具中使用，生成客户端的绑定。

跨语言同步

这个服务可以由外部的XML客户端或JSON客户端调用，内部是通过各种语言（PHP、Java、C++、Python、Ruby、C#等）写的服务程序通过套接口来通信的。这要求基于元数据的代码生成，这样服务的设计者就不必在每次小改动时花时间更新这些代码。

我们已经有了社会关系网站服务的数据组件。接下来我们将弄清楚如何建立这些会话键，在所有Facebook扩展上强制实现用户期望的隐私模型。

6.2.2 简单的Web服务认证握手

一个简单的认证策略让我们能够在尊重Facebook用户的隐私观点的前提下访问这些数据。用户对Facebook系统的数据有某种特定的视图，这取决于用户是谁、用户的隐私设定，以及与用户有关系的人的隐私设定。用户可以授权单个应用来继承这一视图。用户通过某个应用可以看到的信息，是用户通过Facebook可以看到的信息中有意义的一部分（但不会超出通过Facebook可以看到的信息）。

在独立应用站点的架构中（图6-1），用户认证通常采用浏览器发送cookie的方式，这些cookie是该站点在最初执行过认证动作之后生成的。但是在图6-2中，通常作为Facebook用法一部分的cookie不再提供了——外部应用需要在没有用户浏览器的帮助下从Facebook平台请求信息。为了修正这一点，我们在会话键映射的基础上设计Facebook，如例6-11所示。

例6-11：会话键映射

```
get_session: {user_id, application_id} -> session_key
```

Web服务的客户端只要在每次请求时发送session_key，让Web服务知道这代表的是哪个用户的请求执行。如果用户（或Facebook）禁用了这个应用，或者他从未用过这个应用，安全检查就会通不过，会返回一个错误。否则，外部应用站点会把这个会话键记入它自己的用户记录，或者放到该用户的cookie中。

但在最开始如何得到这个会话键呢？在*http://fettermansbooks.com*应用代码中的establish_facebook_session是一个占位符，为这个过程保留的。每个应用都有它自己特有的“应用键”（也称为api_key），开始应用认证流程（图6-3）：

1. 用户通过一个已知的api_key重定向到Facebook登录界面。
2. 用户在Facebook上输入口令，对这个应用授权。
3. 用户带着会话键和用户ID重定向到已知的应用。
4. 应用现在获得了授权，可以代表用户调用API方法（除非会话超时或被删除）。

要帮助用户发起这个流程，可以使用下面包含应用键（即“abc123”）的链接或按钮：

```
<a href="http://www.facebook.com/login.php?api_key=abc123">
```

如果用户通过Facebook上口令输入同意授权给这个应用（注意，口令是Facebook最需要保护的数据），用户就被重定向回这个应用站点，带着有效的会话键和Facebook用户ID。这个会话键是非常私密的，所以对于将来的验证，应用的所有调用都会带有从这个共享秘密生成的散列值。

假定开发者隐藏了他的api_key和应用私密数据，establish_facebook_session可

以很简单地按图6-3中的流程来编写。尽管这种类型的系统握手的细节可以不同，但重要的是只有当用户在Facebook上的关键步骤中输入了他的口令，才会产生授权。很有趣的是，一些早期的应用只是使用了这种认证握手来作为它们的口令系统，而根本没有使用其他的Facebook数据。

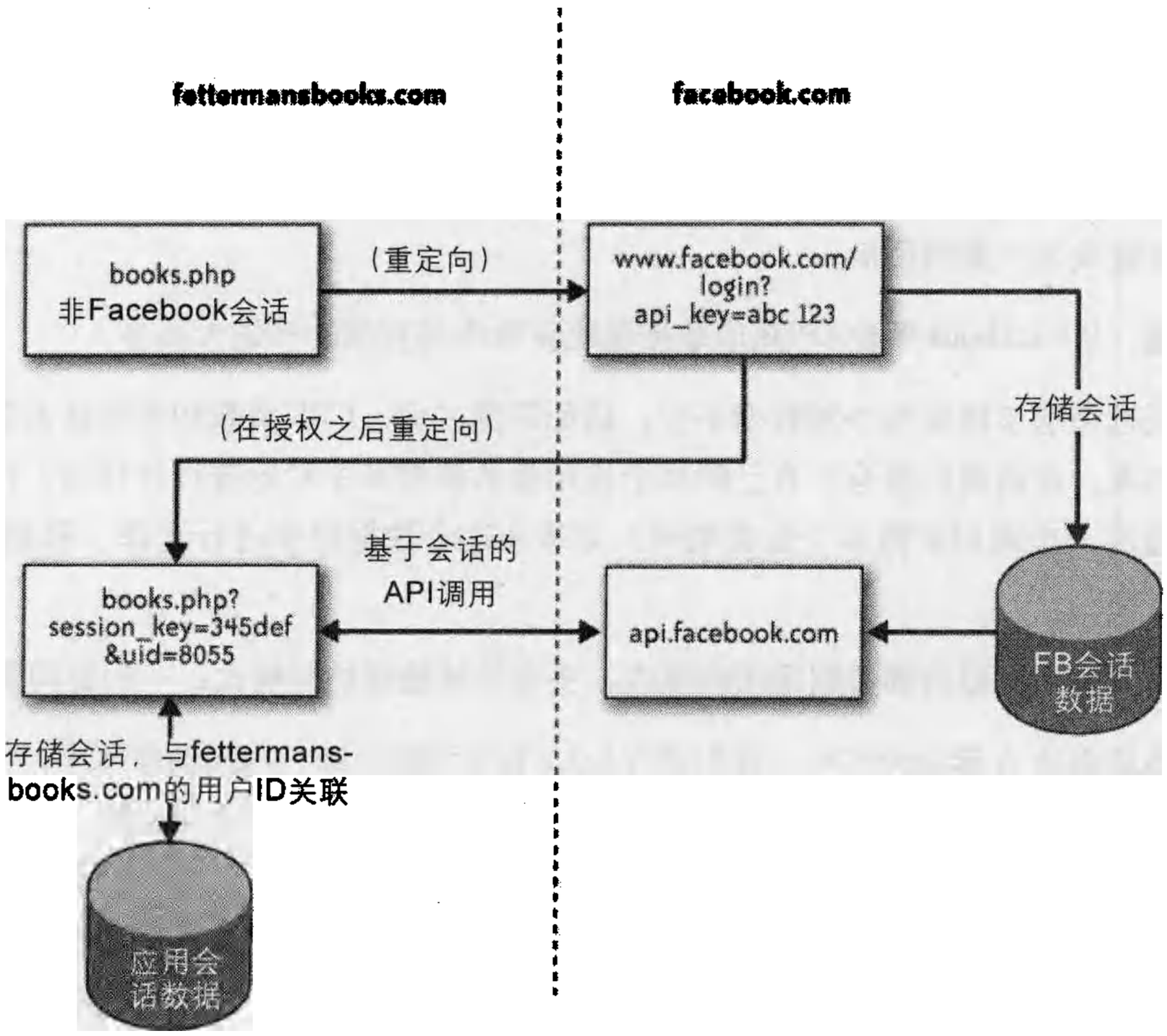


图6-3：对Facebook平台API的认证访问

但是，某些应用不容易适应这种第二步“重定向”的方式。“桌面”风格的应用、基于设备的应用（如手机应用），或浏览器内建的应用有时候也相当有用。在这种情况下，我们采用一种稍微不同的机制来使用第二次认证令牌。令牌是应用通过API请求得到的，在第一次登录时传递给Facebook，然后在现场用户认证之后，应用换到一个会话键和会话专有的一些私密信息。

6.3 创建社会关系数据查询服务

通过一个带有用户控制的认证握手的Web服务，我们已经将我们的内部库扩展到外部世界。通过这个简单的改变，Facebook的社会关系数据现在可以驱动其用户决定认证的任何其他应用程序，通过普遍关注的社会关系上下文，在应用的数据中创建新的关系。

随着用户渐渐了解这些数据交换的无缝性，使用这些平台API的开发者知道这些数据集是很独特的。开发者访问自己的数据的模式与访问Facebook数据的模式有着很大的不同。例如，Facebook的数据位于HTTP请求的另一端，通过许多HTTP连接来调用这些方法增加了开发者自己页面的延迟和开销。他自己的数据库也提供了更大粒度的访问，优于Facebook平台API中的几十个方法。使用他自己的数据和SQL这样熟悉的查询语言，他可以选择一个表的特定字段，对结果集排序或进行限制，匹配其他的指标，或进行嵌套查询。如果平台的API不能够让开发者在平台的服务器上进行智能的处理，开发者就必须经常获取相关数据的超集，收到数据后再在他自己的服务器上进行这些标准的逻辑转换。这可能成为严重的负担。

实际问题：从Facebook平台API获取数据要比获取内部数据的开销大很多。

随着应用越来越多地使用外部数据平台，诸如带宽占用、CPU负载和请求延迟等因素很快累积起来。难道我们没有在自己的单个应用栈的数据层中对此进行优化吗？没有技术让我们通过一次调用取得多个数据集吗？如果在这个数据层中进行选择、限制和排序，结果会怎样？

数据解决方案：类似内部数据采用的模式，实现外部数据访问模式：一种查询服务。

Facebook的解决方案称为FQL，我们将在6.3.2节中详细介绍。FQL很像SQL，但它将平台数据转换成字段和表，而不是简单松散地定义为XML schema中的对象。这让开发者能够在Facebook的数据上使用标准的数据查询语义，这种方式可能与他们取得自己数据的方式一样。同时，将计算推到平台一端的好处与将操作通过SQL推到数据层的好处是相似的。在这两种情况下，开发者有意识地避免了在应用逻辑中进行这种处理的代价。

FQL代表了基于Facebook的内部数据的另一项数据架构改进，是标准的黑盒Web服务的进步。但是首先，我们先来看一种容易而明显的方法，它让开发者能够消除多次数据请求的来回开销，同时我们也要说明为什么这是不够的。

6.3.1 批量方法调用

对于负载问题最简单的解决方案，就是类似于Facebook的`batch.run`API方法。这消除了多次通过HTTP栈对*http://api.facebook.com*进行调用的来回开销，一批接受多个方法调用的输入，一次返回输出的多棵XML树。在客户端，这个过程转变成类似例6-12中的代码。

例6-12：批量方法调用

```
$facebook->api_client->begin_batch();
$friends = &$facebook->api_client->friends_get();
$notifications = &$facebook->api_client->notifications_get();
$facebook->api_client->end_batch();
```

在Facebook平台的PHP5客户端库中，end_batch实际上是向平台服务器发起请求，取得所有结果，并针对每个结果更新引用的变量。这里我们从一次用户会话中批量获取了用户数据。通常，人们用批量查询机制将许多设置操作归为一组，如大量的Facebook个人描述更新，或大量突发的用户通知。

这些批量操作很有效，但这也揭示了这种批量操作的主要问题。问题是，每次调用必须与其他调用的结果无关。对多个不同用户的批量操作通常具备这种特点，但有一种常见的情况仍然不能处理，即使用一次调用的结果作为下次调用的输入。例6-13展示了不能利用批量机制的一种常见情况。

例6-13：批量机制的不正确用法

```
$fields = array('uid, 'name', 'books', 'pic', 'current_location');
$facebook->api_client->begin_batch();
$friends = &$facebook->api_client->friends_get();
$user_info = &$facebook->api_client->users_getInfo($friends, $fields); // NO!
$facebook->api_client->end_batch();
```

当客户端发出users_getInfo请求时，$friends的内容显然还不存在。FQL模型优雅地解决了这个问题和其他问题。

6.3.2 FQL

FQL是一种简单的查询语言，它包装了Facebook的内部数据。输出的格式通常与Facebook平台API的输出格式一样，但输入超出了简单的RPC库的模型，变成了SQL的查询模型：命名的表和字段，包含已知的关系。像SQL一样，这种技术添加了选择实例或范围的能力，从数据行中选择字段子集的能力，并通过嵌套查询将更多的工作推到数据服务器端，避免了通过RPC栈进行多次调用。

举个例子，如果期望的输出是所有用户中我朋友的“uid”、“name”、“book”、“pic”和“current_location”字段，在我们的纯API模型中，我们会使用例6-14中的过程。

例6-14：在客户端串联方法调用

```
$fields = array('uid', 'name', 'books', 'pic', 'current_location');
$friend_uids = $facebook->api_client->friends_get();
$user_infos = users_getInfo($friend_uids, $fields);
```

这导致了对数据服务器的多次调用（这里是2次），更大的延迟，更大的失败可能性。相反，对于查看用户编号8055（实际上是你的），我们在例6-15中写出这样的FQL语法并进行一次调用。

例6-15：利用FQL在服务器端串联方法调用

```
$fql = "SELECT uid, name, books, pic, current_location FROM profile
        WHERE uid IN (SELECT uid2 from friends where uid1 = 8055)";
$user_infos = $facebook->api_client->fql_query($fql);
```

我们在概念上将users_getInfo引用到的数据视为一个表，它基于一个索引（uid），包含一些可选择的字段。如果正确地扩展，这种新的语法可以支持一些新的数据访问能力：

- 限定范围查询（例如根据事件发生的时间）。
- 嵌套查询（SELECT fields_1 FROM table WHERE field IN (SELECT fields_2 FROM ....））。
- 结果集大小限制和排序 。

FQL的架构

开发者通过fql_query API来调用FQL。问题的要点是在FQL的命名“表”和“字段”中，统一外部API的命名“对象”和“属性”。我们仍然继承了标准API的流程：通过内部方法取得数据，应用跟这个方法的API调用相关的规则，然后根据第6.2.1节介绍的Thrift系统，转换到输出。对于每个数据读取API方法，在FQL中都有一个对应的“表”，代表了这次查询背后的数据抽象。例如，API方法users_getInfo，它提供给定用户ID的姓名、照片、书籍和当前位置等字段，在FQL中它就表现为用户表和对应的字段。fql_query的输出实际上也符合标准API的输出（如果修改XSD来允许省略对象小的字段），所以在用户表上调用fql_query返回的输出与相应的users_getInfo调用是等价的。事实上，像user_getInfo这样的调用在Facebook的服务器端通常是实现为FQL调用的！

注意：在编写本章时，FQL只支持SELECT，不支持INSERT、UPDATE、REPLACE、DELETE和其他操作，所以只有读取方法可以通过FQL来实现。大多数操作这类数据的Facebook平台API方法现在是只读的。

我们从这个用户表开始，以它为例，创建FQL系统来支持对它的查询。在平台的各个数据抽象层之下（内部调用、users_getInfo外部API调用，以及新的FQL的用户表），想象Facebook在自己的数据库中有一个名为“user”的表（例6-16）。

例6-16：Facebook数据表示例

```
> describe user;
+--------------+--------------+-----+
| Field        | Type         | Key |
+--------------+--------------+-----+
| uid          | bigint(20)   | PRI |
| name         | varchar(255) |     |
| pic          | varchar(255) |     |
| books        | varchar(255) |     |
| loc_city     | varchar(255) |     |
| loc_state    | varchar(255) |     |
| loc_country  | varchar(255) |     |
| loc_zip      | int(5)       |     |
+--------------+--------------+-----+
```

在Facebook的程序栈中，支持我们访问这个表的方法是：

```
function user_get_info($uid)
```

它在我们选择的语言（PHP）中返回一个对象，通常此后再应用隐私逻辑，并展现在*http://facebook.com*上。我们的Web服务实现做的事情相当类似，将Web请求的GET/POST内容转给这样一个调用，得到类似的栈对象，应用隐私逻辑，然后通过Thrift将它变成一个XML响应（图6-2）。

我们可以在FQL中将user_get_info包装起来，实际实现这个模型，将表、字段、内部函数和隐私组织成一个逻辑上的、可重复的形式。

下面是例6-15中的FQL调用创建的一些关键对象，以及描述它们的关系的方法。讨论所有的字符串解析、语法实现、可选索引、交集查询和实现许多不同的组合表达式（比较、“in”语句、交集、非交集）超出了本章的范围。这里我们只是关注面向数据的部分：FQL中数据的对应字段和表对象的高层规范，并将查询输入语句转换每个字段的can_see和evaluate函数（例6-17）。

例6-17：FQL字段和表示例

```
class FQLField {
  // e.g. table="user", name="current_location"
  public function __construct($user, $app_id, $table, $name) { ... }

  // mapping: "index" id -> {0,1} (visible or invisible)
  public function can_see($id) { ... }

  // mapping: "index" id -> Thrift-compatible data object
  public function evaluate($id) { ... }
}

class FQLTable {
  // a static list of contained fields:
  // mapping: () -> ('books' => 'FQLUserBooks', 'pic' ->'FQLUserPic', ...)
  public function get_fields() { ... }
}
```

FQLField和FQLTable对象构成了这个访问数据的新方法。FQLField包含了针对数据的逻辑，将“行”（如用户ID）和查看者的信息（用户和app_id）转换成我们内部的栈数据调用。在此之上，我们确保隐私评估利用要求的can_see方法得以正确实现。在我们处理一个请求时，我们可以在内存中为每个命名的表格（“user”）创建这样一个FQLTable对象，为每个命名的字段创建一个FQLField对象（为“books”创建一个，为“pic”创建一个，等等）。对应到一个FQLTable中的每个FQLField对象一般会使用底层相同的数据访问程序（在下面的例子里，是user_get_info），虽然不一定是这样——这只是一个方便的接口。例6-18展示了用户表中典型的字符串字段的例子。

例6-18：将核心数据库映射到FQL字段定义

```
// base object for any simple FQL field in the user table.
class FQLStringUserField extends FQLField {

  public function __construct($user, $app_id, $table, $name) { ... }

  public function evaluate($id) {
    // call into internal function
    $info = user_get_info($id);
    if ($info && isset($info[$this->name])) {
      return $info[$this->name];
    }
    return null;
  }

  public function can_see($id) {
    // call into internal function
    return can_see($id, $user, $table, $name);
  }
}

// simple string data field
class FQLUserBooks extends FQLStringUserField { }

// simple string data field
class FQLUserPic extends FQLStringUserField { }
```

FQLUserPic和FQLUserBooks的区别仅限于它们的内部属性$this->name，这是由它们的构造方法在处理过程中设置的。请注意，在底层，我们针对表达式中需要的每次求值调用user_get_info；只有系统将这些结果缓存在内存中，才能取得较好的性能。Facebook的实现就是这样做的，整个查询执行的时间与标准平台API调用的时间是同一量级的。

下面是一个更复杂的字段，表示current_location，它采用的是同样的输入，展示了同样的使用模式，但输出了一个我们前面曾看到过的结构类型对象（例6-19）。

例6-19：更复杂的FQL字段映射

```
// complex object data field
class FQLUserCurrentLocation extends FQLStringUserField {
  public function evaluate($id) {
    $info = user_get_info($id);
    if ($info && isset($info['current_location'])) {
      $location = new api10_location($info['current_location']);
    } else {
      $location = new api10_location();
    }
    return $location;
  }
}
```

像api10_location这样的对象是6.2.1小节中所说的生成的类型，Thrift和Facebook数据服务知道如何将它返回为良好类型的XML。现在我们知道，为什么就算是新的输入形式，FQL的输出也不会与Facebook API产生不兼容的情况。

在下面的例子中，FQLStatement的主要求值循环告诉了我们FQL实现的大致思想。在这段代码中我们引用了FQLExpression，但在简单的查询中，我们更有可能提到的是FQLFieldExpression，它包装了对FQLField自己的求值和can_see方法的内部调用，如例6-20所示。

例6-20：一个简单的FQL表达式类

```
class FQLFieldExpression {

  // instantiated with an FQLField in the "field" property
  public function evaluate($id) {
    if ($this->field->can_see($id))
      return $this->field->evaluate($id);
    else
      return new FQLCantSee(); // becomes an error message or omitted field
  }

  public function get_name() {
    return $this->field_name;
  }
}
```

要发起整个流程，类似SQL的字符串输入通过lex和yacc转换成主要FQLStatement的$select表达式数组和$where表达式。FQLStatemenet的evaluate()函数将返回我们请求的对象。例6-21中的主语句求值循环包括了以下步骤，说明了简单的大致顺序：

1. 取得我们希望返回的行在索引上的约束。例如，如果在用户表上选取，这就是我们想查询的那些UID。如果我们在一个按时间索引的事件表上查询，这就是时间边界。
2. 将这些转换成表的规范ID。用户表也可以按字段名查询，如果FQL表达式使用了字段名称，这个函数就会使用内部的user_name到user_id的查找函数。
3. 针对每个候选ID，看看它是否满足RHS表达式子句（布尔逻辑、比较、“IN”操作等）。如果不满足，就抛弃它。
4. 对每个表达式求值（在我们的例子里，就是SELECT子句中的字段），然后创建<COL_NAME>COL_VALUE</COL_NAME>格式的XML元素，其中COL_NAME是FQLTable中的字段名称，COL_VALUE是字段通过它对应的FQLField的求值函数进行求值的结果。

例6-21：FQL的主求值流程

```
class FQLStatement {

  // contains the following members:
```

```
// $select: array of FQLExpressions from the SELECT clause of the query
//  corresponding to, say, "books", "pic", and "name"
// $from: FQLTable object for the source table
// $where: FQLExpression containing the constraints for the query.
// $user, $app_id: calling user and app_id

public function __construct($select, $from, $where, $user, $app_id) {... }

// A listing of all known tables in the FQL system.
public static $tables = array(
  'user'          => 'FQLUserTable',
  'friend'        => 'FQLFriendTable',
);

// returns XML elements to be translated to service output
public function evaluate() {

  // based on the WHERE clause, we first get a set of query expressions that
  // represent the constraints on values for the indexable columns contained
  // in the WHERE clause

  // Get all "right hand side" (RHS) constants matching ids (e.g. X, in 'uid = X')
  $queries = $this->where->get_queries();

  // Match to the row's index. If we were using 'name' as an alternative index
  // to the user table, we would transform it here to the uid.
  $index_ids = $this->from_table->get_ids_for_queries($queries);

  // filter the set of ids by the WHERE clause and LIMIT params
  $result_ids = array();

  foreach ($ids as $id) {
    $where_result = $this->where->evaluate($id);

    // see if this row passes the 'WHERE' constraints
    // is not restricted by privacy
    if ($where_result && !($where_result instanceof FQLCantSee))
      $result_ids []= $id;
}

$result = array();
$row_name = $this->from_table->get_name(); // e.g. "user"

// fill in the result array with the requested data
foreach ($result_ids as $id) {
  foreach ($this->select as $str => $expression) { // e.g. "books" or "pic"
    $name = $expression->get_name();
    $col = $expression->evaluate($id); // returns the value
    if ($col instanceof FQLCantSee)
      $col = null;

    $row->value[] = new xml_element($name, $col);
  }
```

```
        $result[] = $row;
      }
      return $result;
}
```

FQL还有其他一些精妙之处，但这个总体流程说明了已有的内部数据访问和隐私规则实现与全新的查询模型的结合。这让开发者能够更快地处理它的请求，能够以比API更好的粒度来访问数据，同时又保持了SQL类似的语法。

由于我们的许多API在内部包装了对应的FQL方法，我们的整体架构演变为图6-4所示的状况。

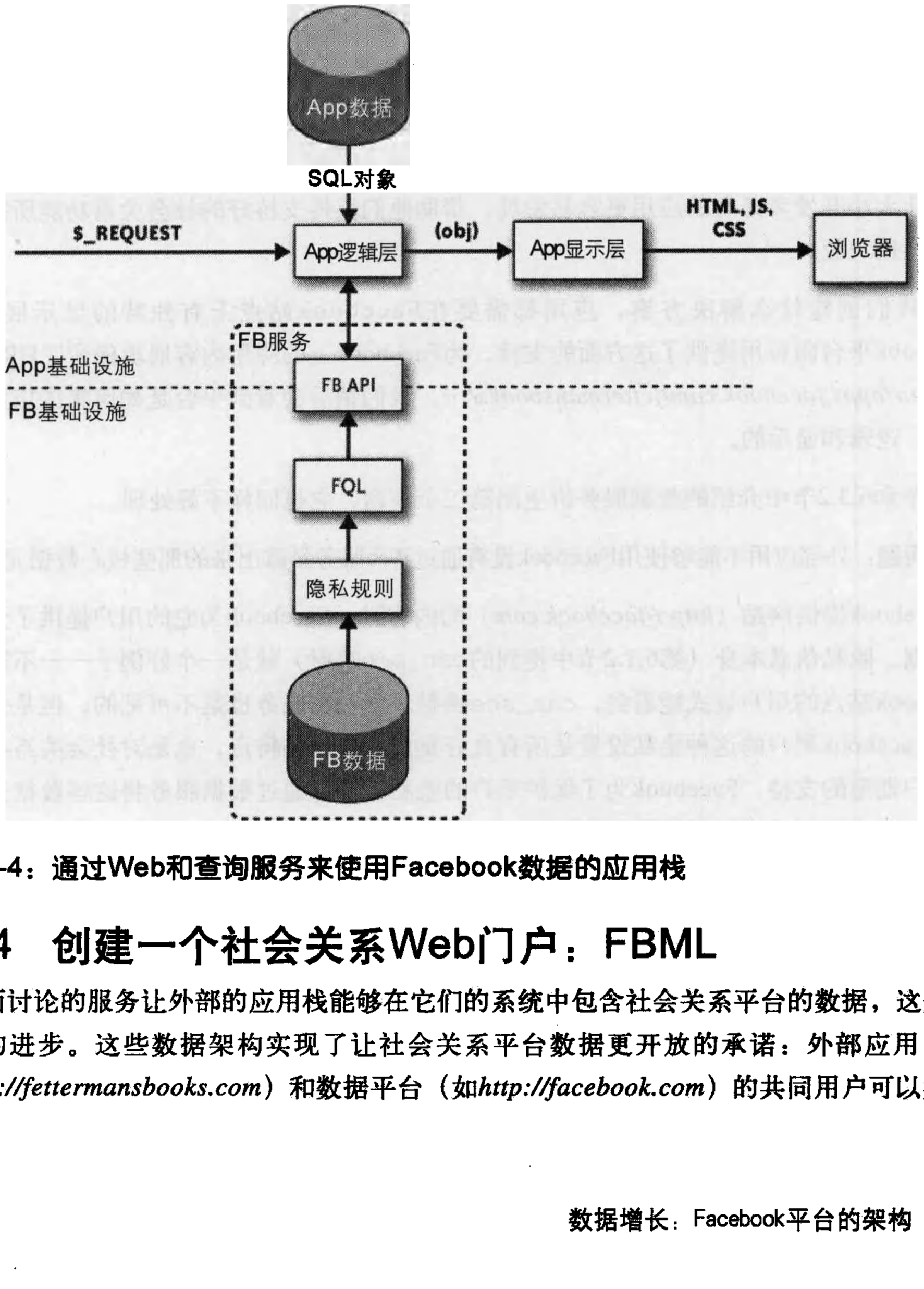


图6-4：通过Web和查询服务来使用Facebook数据的应用栈

6.4 创建一个社会关系Web门户：FBML

前面讨论的服务让外部的应用栈能够在它们的系统中包含社会关系平台的数据，这是很大的进步。这些数据架构实现了让社会关系平台数据更开放的承诺：外部应用（如*http://fettermansbooks.com*）和数据平台（如*http://facebook.com*）的共同用户可以共享

信息，每个新的社会关系应用就不需要一个新的社会关系网络。但是，即使有了这些新的能力，这些应用还是不能享受Facebook这样的社会关系网站的全部强大功能。应用还需要让许多用户发现，才会变得有价值。而且，并不是所有支持社会关系平台的内部数据都可以提供给这些外部的应用栈。平台的创建者需要解决这些问题，我们将依次讨论。

实际问题：对于社会关系应用来说，要获得引人注目的关键性用户数，支持它的社会关系网络上的用户必须要能注意到其他用户在利用这些应用进行交互。这意味着应用与社会关系网站更深层次上的集成。

这个问题在早期的软件中就存在了：我们难以让数据、产品或系统得到广泛使用。缺少用户成为Web 2.0空间中特别值得一提的困难，因为如果没用户使用并且（特别是）生成内容，我们的系统什么时候才有用呢？

Facebook支持大量的用户，他们对在社会联系之间共享信息感兴趣，而且Facebook的特点就是把应用的内容和它自己的内容等同视之。让外部的应用出现在Facebook站点上，就会让大小开发者开发的应用更容易发现，帮助他们获得支持好的社会关系功能所需的关键性用户数。

不管我们创建什么解决方案，应用都需要在Facebook站点上有独特的显示展现。Facebook平台向应用提供了这方面的支持，为Facebook上的应用内容展现保留了URL路径*http://apps.facebook.com/fettermansbooks/…*。我们稍后会看到平台是如何集成应用的数据、逻辑和显示的。

6.2.1节和6.3.2节中介绍的数据服务衍生出第二个问题，它也同样不好处理。

实际问题：外部应用不能够使用Facebook没有通过Web服务暴露出来的那些核心数据元素。

在Facebook提供网站（*http://facebook.com*）的内容时，Facebook为它的用户提供了大量的数据。隐私信息本身（第6.1.2节中提到的can_see映射）就是一个好例子——不能被Facebook站点的用户显式地看到，can_see映射对于数据服务也是不可见的。但是强制实现Facebook用户的这种隐私设置是所有良好集成的应用的特点，也是对社会关系系统上用户期望的支持。Facebook为了保护用户的隐私，不能通过数据服务将这些数据开放出来。开发者怎样才能利用这些数据呢？

对这些问题的最优雅解决方案就是结合Facebook的数据和外部应用的数据、逻辑和显示，同时让用户在一个受信任的环境下操作。

数据解决方案：开发者通过一种数据驱动的标记语言，在社会关系站点上创建应用执行和显示的内容，与Facebook交互。

只使用第6.2.1节和第6.3.2节中介绍的Facebook平台元素的应用，在Facebook之外创造了

一种社会关系体验，因Facebook的社会关系数据而变得更强大。利用本节介绍的数据和Web架构，应用本身也变成一种数据服务，支持针对Facebook的内容显示在*http:// apps.facebook.com*之下。像*http://apps.facebook.com/fettermansbooks/...* 这样的URL不再映射到Facebook生成的数据、逻辑和显示，而是会查询*http://fettermansbooks.com*的服务，生成应用的内容。

我们必须同时记得我们的资产和约束。一方面，我们有一个访问频率很高的社会关系系统，让用户能发现外部的内容，并有大量的社会关系数据来增强这种社会关系应用。另一方面，请求需要从社会关系站点（Facebook）上发起，将应用作为服务来使用，然后将内容渲染成HTML、JavaScript和CSS，并且不违反Facebook用户的隐私或期望。

首先，我们来看一些不正确的尝试。

6.4.1 Facebook上的应用：直接渲染HTML、CSS和JS

假定一个外部应用的配置现在包含两个字段，名为application_name和callback_url。通过输入“fettermansbooks"这样的名字和*http://fettermansbooks.com/fbapp/*这样的URL，*http://fettermanbooks.com*声明它将在自己的服务器上为用户提供服务，对*http://apps.facebook.com/fettermansbooks/PATH?QUERY_STRING*的请求将转向*http://fettermansbooks.com/fbapp/PATH?QUERY_STRING*。

对http://apps.facebook.com/fettermansbooks/…的请求于是简单地取出应用服务器上的HTML、JS和CSS等内容，并在Facebook上的页面主内容区域进行显示。这基本上是将外部站点作为一个HTML Web服务来渲染的。

这对应用的*n*层模型进行了重要改变。以前，应用栈会通过数据服务来使用Facebook的内容，这个数据服务是直接服务于对*http://fettermansbooks.com*的请求的。现在，应用在它的Web根下维护了一个树型结构，它自己提供HTML服务。Facebook通过在线请求这个新应用服务（该服务又可能用到Facebook的数据服务）而取得内容，将它包装成一般的Facebook站点导航元素，显示给用户。

但是，如果Facebook直接在它的页面中渲染一个应用的HTML、JavaScript或CSS，这就会允许应用完全违反用户对*http://facebook.com*上受控体验的期望，让站点和用户暴露在各种安全攻击之下。允许外部用户直接订制标记语言和脚本几乎从来都不是好主意。实际上，代码或脚本注入通常是攻击者的目标，所以这并不是一个很好的特征。

而且，没有新数据！尽管这为应用栈的改变奠定了基础，但这个解决方案没有完全解决前面的两个实际问题。

6.4.2 Facebook上的应用：iframe

还有一种更安全的显示应用内容的方法，可以显示另一个站点的可视化上下文和界面流转，这种方法已包含在浏览器中，即iframe。

为了复用前一节中提到的映射，对 *http://apps.facebook. com/ fettermans books/ PATH?QUERY_STRING*的请求将导致输出这样的HTML：

```
<iframe src="http://fettermansbooks.com/fbapp/PATH?GET_STRING"></iframe>
```

这个URL的内容将显示在Facebook页面的一个帧中，在它自己的沙盒环境中可以包含任何类型的Web技术：HTML、JS、AJAX、Flash等。

这实际上是让浏览器成为请求代理者，而不是由Facebook作为请求代理者。这比前一节中的模型有改进，浏览器也维护所得页面中其他元素的安全性，所以开发者可以在这个帧中随意创建他们想要的用户体验。

对于某些应用，如果开发者希望花最小的代价将他们的代码从他们的站点移到平台上，那么iframe的方式也是有意义的。实际上，Facebook继续支持完整页面生成的iframe模型。虽然这解决了第一个实际问题，将应用纳入到社会关系站点，但第二个实际问题仍未解决。虽然基于iframe的请求流程可以确保安全，但除了API服务暴露出来的数据之外，这些开发者并不能利用其他的新数据。

6.4.3 Facebook上的应用：FBML是数据驱动的执行标记语言

前两节中提到的解决方案尝试都有其优点。HTML的解决方案采用了直观的方法，将应用本身变成Web服务，将触点带回到Facebook上显示。iframe方式的好处在于将开发者的应用内容放在一个独立的（安全的）执行沙盒中。最佳解决方案将保留“应用即服务”的模型和iframe的安全和可信，同时又让开发者能够使用更多的社会关系数据。

问题是，为了让社会关系应用提供独特的使用体验，开发者必须通过他们自己的应用栈来提供数据、逻辑和展现。但是，生成这些输出必须用到那些不能离开Facebook的用户数据。

解决方案是什么？不是发回HTML，而是一种特定的标记语言，其中定义了足够的标记来表现应用的逻辑和显示，也包含对受保护数据的请求，完全让Facebook在受信任的服务器环境中渲染它！这就是FBML的前提 （图6-5）。

在这个流程中，对*http://apps.facebook.com*的请求同样被转换成对应用的请求，应用栈会使用Facebook的数据服务。但是，开发者不会让应用返回HTML，而是重写应用，返回FBML。FBML中包含了许多HTML元素，而且添加了Facebook特别定义的标签。当这个请求返回其内容时，Facebook的FBML解释器将这段标记语言转换成它自己的数据、

执行和显示实例，生成应用页面。用户就会收到一个页面，其中包含了Facebook页面的一般Web元素，而且也包含了应用的数据、逻辑和观感。不论FBML返回什么，它都能在技术上确保Facebook强制实现其隐私理念和良好的用户体验元素。

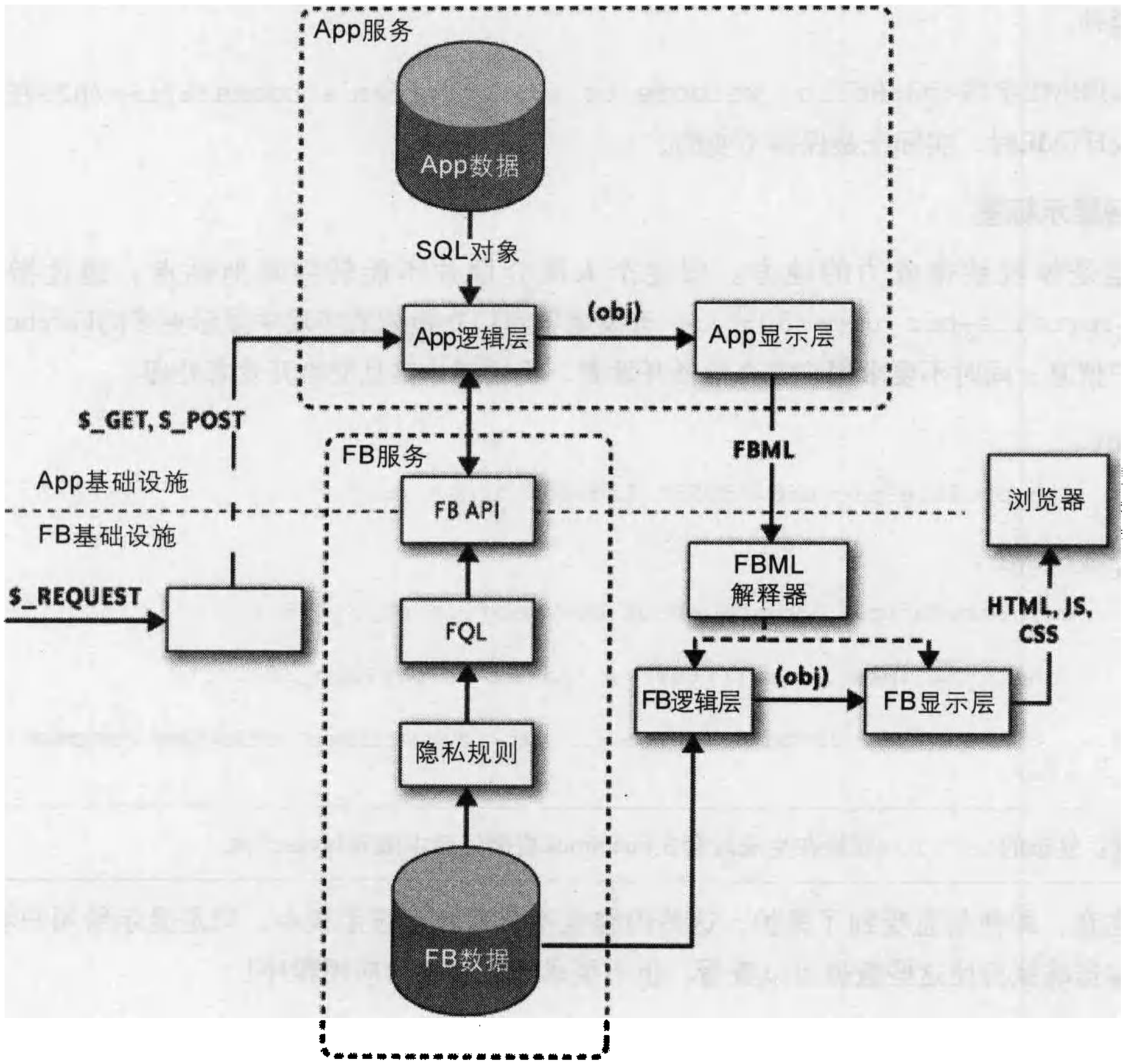


图6-5：应用即FBML服务

FBML是XML的一个特例，它包含了许多熟悉的HTML标签，增加了在Facebook上显示的平台专有的标签。FBML同样体现了FQL的高级模式：修改已知的标准（HTML，对FQL来说就是SQL），将执行和决定延迟到Facebook平台服务器上进行。如图6-5所示，FBML解释器让开发者通过FBML数据，自己能够控制在Facebook服务器上执行的逻辑和显示。这是数据处于执行中心的绝妙例子：FBML只是声明式的执行，而不是必须服从的控制流（如在C、PHP等语言中）。

现在来看具体细节。FBML是一个XML实例，所以它由标签、属性和内容组成。标签可以在概念上分成以下几大类。

直接的HTML标签

如果FBML服务返回标签<p/>，Facebook将在输出页上直接渲染为<p/>。作为Web展现的基石，大多数HTML标签都是支持的，少数违反Facebook层面的信任或设计期望的标签除外。

所以FBML字串<h2>Hello, welcome to Fetterman's books!</h2>在渲染成HTML时，实际上是保持不变的。

数据显示标签

这里是体现数据威力的地方。假定个人简介照片不能转到其他站点。通过指定<fb:profile-pic uid="8055">，开发者就可以在他们的应用中显示更多的Facebook用户信息，同时不要求用户完全信任开发者，将这部分信息交给开发者处理。

例如：

```
<fb:profile-pic uid="8055" linked="true" />
```

翻译成FBML：

```
<a href="http://www.facebook.com/profile.php?id=8055"
 onclick="(new Image()).src = '/ajax/ct.php?app_id=...">
  <img uid="8055" linked="true" src="..." alt="Dave Fetterman" title="Dave Fetterman" />
</a>
```

注意：复杂的onclick属性在生成时会在Facebook页面显示中限制Javascript。

请注意，即使信息受到了保护，这些内容也不会返回到应用栈中，只是显示给用户看。在容器端执行使这些数据可以查看，但不要求将它们交给应用程序！

数据执行标签

作为使用隐藏数据的一个更好的例子，用户的隐私限制只能通过内部的can_see方法来访问，它是应用体验的一个重要部分，但不能通过数据服务从外面进行访问。利用<fb:-if-can-see>标签和其他类似的标签，应用可以通过属性来指定一个目标用户，这样只有当查看者能够看到目标用户的特定内容时，那些子元素才会渲染出来。因此，隐私数据本身不会暴露给应用，同时应用又能满足强制实现的隐私设置。

从这个角度来说，FBML是一个受信任的声明式执行环境，与C或PHP这样的必须服从的执行环境不同。严格来说，FBML不像这些语言那样是“图灵完备”的（例如，没有提供循环结构）。像HTML一样，除了树状遍历所隐含的状态外，在执行时不保存任何状态；例如，<fb:tab-item>只在<fb:tabs>之内有意义。但是，通过让受信任系统中的用户获得数据，FBML提供了大量功能，这些功能正是大多数开发者希望提供给他们的用户的。

FBML实际上有助于定义执行应用的逻辑和显示，同时又让应用可以在应用服务器上显示独特的内容。

只面向设计的标签

Facebook因其设计标准而受到称赞，许多开发者都选择以某种方式复用Facebook的设计元素，保持Facebook的观感。通常，他们是通过利用*http://facebook.com*的JavaScript和CSS来实现的，但FBML提供了类似“设计宏”的库，以更可控的方式来满足这种需求。

例如，Facebook应用已知的CSS类来，将输入<fb:tabs>…/fb:tabs>渲染成特定的tab标签结构，位于开发者页面的顶部。这些设计元素也可以包含执行语义，如<fb:narrow>…</fb:narrow>只有在这次执行显示用户简介框中的少数列时，会在FBML中渲染它的子内容。

例6-22展示了一些使用只面向设计的标签的FBML。

例6-22：只面向设计的FBML的例子

```
<fb:tabs>
<fb:tab-item href="http://apps.facebook.com/fettermansbooks/mybooks.php"
 title='My Books' selected='true'/>
<fb:tab-item href="http://apps.facebook.com/fettermansbooks/recent.php"
 title='Recent Reviews' />
</fb:tabs>
```

这将渲染为一组可视的tab标签，链接到对应用的内容，并使用了Facebook自己的HTML、CSS和Javascript包。

替代HTML标签

HTML造成了一些信任风险，但没有暴露数据，所以FBML中的替代标签只是修改或限制一组参数，如Flash自动播放。这不是所有显示平台都严格要求的，它们只是强制应用满足容器站点的默认显示行为。但是，随着多个应用发展成为一个生态系统，它们都反映容器站点的观感，这种修改就变得很重要了。

请看这个FBML例子：

```
<fb:flv src=http://fettermansbooks.com/newtitles.flv height="400"
  width="400" title="New Releases">
```

这会翻译成一段相当长的JavaScript，渲染一个视频播放组件，这个元素由Facebook控制，特意禁止了自动播放这样的行为。

“功能包”标签

某些Facebook FBML标签包含了整套的常见Facebook应用功能。<fb:friend-selector>创建了类型前置的朋友选择器软件包，常见于许多Facebook页面，包括

Facebook数据（朋友、主要网络）、CSS样式和针对键盘动作的JavaScript。像这样的标签让容器站点可以推广某些设计模式和应用间的公用元素，也让开发者能够快速实现他们想要的功能。

FBML：一个小例子

请回忆一下我们在创建假想的外部网站时，通过引入`friends.get`和`users.getInfo` API对原来的*http://fettermansbooks.com*代码实现改进。接下来我们将展示一个例子，看看FBML如何能够结合社会关系、私有业务逻辑和完全集成的应用的感觉。如果我们能够通过数据库调用`book_get_all_reviews($isbn)`获得一本书的全部书评，那么我们就可以将朋友数据、私有业务逻辑和“墙式”风格结合起来，利用FBML在容器站点上显示书评，代码如例6-23所示。

例6-23：利用FBML创建一个应用

```
// Wall-style social book reviews on Facebook
// FBML Tags used: <fb:profile-pic>, <fb:name>, <fb:if-can-see>,
<fb:wall>

// from section 1.3
$facebook_friend_uids = $facebook_client->api_client->friends_get();
foreach($facebook_friend_uids as $facebook_friend) {
  if ($books_user_id = books_user_id_from_facebook_id($facebook_friend))
    $book_site_friends[] = $books_user_id;
}

// a hypothesized mapping, returning
// books_uid -> book_review object
$all_reviewers = get_all_book_reviews($isbn);

$friend_reviewers = array_intersect($book_site_friends, array_keys($all_reviewers));

echo 'Friends' reviews:<br/>';
echo '<fb:wall>';

// put friends up top.
foreach ($friend_reviewers as $book_uid => $review) {
  echo '<fb:wallpost uid="'.$book_uid.'">';
  echo '(' . $review['score'] . ')' . $review['commentary'];
  echo '</fb:wallpost>';
  unset($all_reviewers[$book_uid]); // don't include in nonfriends below.
}

echo 'Other reviews:<br/>';

// only nonfriends remain.
foreach ($all_reviewers as $book_uid => $review) {
  echo '<fb:if-can-see uid="'.$book_uid.'">'; // defaults to 'search' visibility
  echo '<fb:wallpost uid="'.$book_uid.'">';
  echo '(' . $review['score'] . ')' . $review['commentary'];
```

```
    echo '</fb:wallpost>';
    echo '</fb:if-can-see>';
}

echo '</fb:wall>';
```

虽然这采用的是输出FBML的服务的形式，而不是输出HTML的Web调用，但一般流程是不变的。这里，Facebook数据让应用能够在无关的书评之前，显示更多的相关书评（朋友的书评），并且使用了FBML来显示结果，采用了Facebook上相应的隐私逻辑和设计元素。

6.4.4 FBML架构

将开发者提供的FBML翻译成显示在*http://facebook.com*上的HTML，需要一些技术和概念综合作用：将输入字符串解析成一棵句法树，将这棵树中的标签转换成内部方法调用，应用FBML语法规则，保持容器站点的约束。像FQL一样，这里我们将关注点主要放在FBML与平台数据的交互上，对其他的技术则不作详细探讨。FBML处理了一个复杂的问题，FBML的全部实现细节是相当多的——我们省略的内容包括FBML的错误日志、为后来的渲染事先缓存内容的能力、表单提交结果的安全性签名等。

首先，看看解析FBML的低层问题。在继承了浏览器的某些角色的同时，Facebook也继承了它的一些问题。为了方便开发者，我们不要求提供的输入可以通过schema验证，甚至不要求是结构良好的XML——不封闭的HTML标签，如<p>（与XHTML不同，即<p/>）打破了输入必须作为真正的XML进行解析的假定。因为这一点，我们需要一种方法将输入的FBML字符串先转换成结构良好的句法树，包含标签、属性和内容。

为了做到这一点，我们采用了采用了一个开放源代码浏览器的一些代码。本章将这部分处理视为一个黑盒，所以我们现在假定，在接收到FBML并经过这样的处理流程后，我们得到了名为FBMLNode的树状结构，它让我们能够查询生成的句法树中任何节点的标签、属性键值对和原始内容，并能够递归查询子元素。

从最高的层面上看，我们可以注意到FBML出现在Facebook站点的所有地方：应用“画布”页面、新闻信号源的故事内容、个人简介框的内容等。每种上下文中或每种“风味”的FBML都定义了对输入的约束，例如，画布允许使用iframe，而个人简介框则不允许。很自然，因为FBML维护数据隐私的方式与API类似，所以执行上下文中必须包含查看用户的ID和生成该内容的应用ID。

所以，在我们真正开始有效使用FBML之前，先要看看环境的规则，它由FBMLFlavor类来封装，如例6-24所示。

例6-24：FBMLFlavor类

```
abstract class FBMLFlavor {

// constructor takes array containing user and application_id
  public function FBMLFlavor ($environment_array) { ... }
  public function check($category) {
    $method_name = 'allows_' . $category;
    if (method_exists($this,$method_name)) {
      $category_allowed = $this->$method_name();
    } else {
      $category_allowed = $this->_default();
    }
    if (!$category_allowed))
        throw new FBMLException('Forbidden tag category '.$category.' in
this flavor.');
  }
  protected abstract function _default();
}
```

下面是这个抽象类的一个子类，它对应于渲染FBML的页面或元素。例6-25是一个例子。

例6-25：FBMLFlavor类的一个子类

```
class ProfileBoxFBMLFlavor extends FBMLFlavor {
  protected function _default() { return true; }
  public function allows_redirect() { return false; }
  public function allows_iframes() { return false; }
  public allows_visible_to() { return $this->_default(); }
  // ...
}
```

这种风味类的设计很简单：它包含了隐私上下文（用户和应用），实现了检查方法，为稍后将展示的FBMLImplementation类中包含的丰富逻辑建立了规则。与平台API的实现层很像，这个实现类为服务提供了实际的逻辑的数据访问，其他的代码为这些方法提供了访问入口。每个Facebook特有的标签，如<fb:TAG-NAME>，将有一个对应的实现方法fb_TAG_NAME（例如，类方法fb_profile_pic将实现<fb:profile-pic>标签的逻辑）。每个标准的HTML标签也都有一个对应的处理方法，名为tag_TAG_NAME。这些HTML处理方法通常让数据无变化地通过，但是即便是对一些“普通”的HTML元素，FBML常常也需要进行检查和转换。

让我们来看看某些标签的实现，然后将它们结合起来讨论。每个实现方法都接收一个来自FBML解析器的FBMLNode，以字符串的方式返回输出的HTML。下面是一些直接的HTML标签、数据显示标签和数据执行标签的实现示例。请注意，这些程序清单用到了一些功能，在这里没有完整而详细地列出。

在FBML中实现直接的HTML标签

例6-26包含了<img>标签的内部FBML实现。图像标签的实现包含更多的逻辑，有时候

需要将图像源的URL重写到Facebook服务器上图像缓存的URL。这体现了FBML的强大：应用栈可以返回与HTML非常相似的标记语言，支持它自己的站点，而Facebook可以通过纯技术的手段强制实现平台所要求的行为。

例6-26：fb:img标签的实现

```
class FBMLImplementation {
  public function __construct($flavor) {... }


  // <img>: example of direct HTML tag (section 4.3.1)
  public function tag_img($node) {

    // images are not allowed in some FBML contexts -
    // for example, the titles of feed stories
    $this->_flavor->check('images');

    // strip of transform attribute key-value pairs according to
    // rules in FBML
    $safe_attrs = $this->_html_rewriter->node_get_safe_attrs($node);
    if (isset($safe_attrs['src'])) {
      // may here rewrite image source to one on a Facebook CDN
      $safe_attrs['src'] = $this->safe_image_url($safe_attrs['src']);
    }
    return $this->_html_rewriter->render_html_singleton_tag($node->
    get_tag_name(), $safe_attrs);
  }
}
```

在FBML中实现数据显示标签

例6-27展示了通过FBML使用Facebook数据的例子。<fb:profile-pic>用到了uid、size和title属性，将它们结合起来，根据内部数据产生HTML输出，并符合Facebook的标准。在这个例子中，输出是指定用户名的个人简单照片，链接到用户的个人简介页面，只在当查看者能看到这部分内容时才显示。这个功能也存在于FBMLImplementation类中。

例6-27：fb:profile-pic标签的实现

```
// <fb:profile-pic>: example of data-display tag
public function fb_profile_pic($node) {
  // profile-pic is certainly disallowed if images are disallowed
  $this->check('images');
  $viewing_user = $this->get_env('user');
  $uid = $node->attr_int('uid', 0, true);
  if (!is_user_id($uid))
    throw new FBMLRenderException('Invalid uid for fb:profile_pic ('.$uid .')');

  $size = $node->attr('size', "thumb");
  $size = $this->validate_image_size($size);

  if (can_see($viewing_user, $uid, 'user', 'pic')) {
```

```
    // this wraps user_get_info, which consumes the user's 'pic' data field
    $img_src = get_profile_image_src($uid, $size);
  } else {
    return '';
  }
  $attrs['src'] = $img_src;
  if (!isset($attrs['title'])) {
    // we can include the user name information here too.
    // again, this function would wrap internal user_get_info
    $attrs['title'] = id_get_name($id);
  }
  return $this->_html_renderer->render_html_singleton_tag('img', $attrs);
}
```

FBML中的数据执行标签

FBML解析的递归本质使得<fb:if-can-see>标签就像是标准的必须服从的控制流中的if语句一样，它是FBML实际控制执行的一个例子。这是FBML实现类中的另一个方法，例6-28列出了它的细节。

例6-28：fb:if-can-see标签的实现

```
// <fb:if-can-see>: example of data-execution tag
public function fb_if_can_see($node) {
  global $legal_what_values; // the legal attr values (profile, friends, wall, etc.)
  $uid = $node->attr_int('uid', 0, true);
  $what = $node->attr_raw('what', 'search'); // default is 'search' visibility
  if (!isset($legal_what_values[$what]))
    return ''; // unknown value? not visible

  $viewer = $this->get_env('user');
  $predicate = can_see($viewer, $uid, 'user', $what);
   return $this->render_if($node, $predicate); // handles the else case
for us
}

// helper for the fb_if family of functions
protected function render_if($node, $predicate) {
  if ($predicate) {
    return $this->render_children($node);
  } else {
    return $this->render_else($node);
  }
}

protected function render_else($node) {
  $html = '';
  foreach ($node->get_children() as $child) {
    if ($child->get_tag_name() == 'fb:else') {
      $html .= $child->render_children($this);
    }
  }
```

```
  return $html;
}

public function fb_else($ignored_node) { return ''; }
```

如果某对“观察者-目标”通过了can-see检查，引擎就会递归地渲染<fb:if-can-see>节点的子节点。否则，就会渲染可选标签<fb:else>子节点下的内容。请注意fb_if_can_see直接访问<fb:else>子节点的方式；如果<fb:else>出现在这样的一个“if风格”的FBML标签之外，标签和它的子标签就不会返回任何内容。所以，FBML不仅仅是一个简单的转换式例程，它会注意到文档的结构，因此可以包含条件控制流的元素。

结合在一起

前面讨论的每个功能，都需要注册为一个回调，在解析输入的FBML时使用。在Facebook（以及它的开放源代码平台实现中），这个“黑盒”解析器是用C写的PHP扩展，每个回调都存在于PHP树中。要完成这种高层控制流，我们必须向FBML解析引擎声明这些标签。和其他地方一样，出于简单性考虑，例6-29也是经过了大量编辑的。

例6-29：FBML主要求值流程

```
// As input to this flow:
// $fbml_impl - the implementation instantiated above
// $fbml_from_callback - the raw FBML string created by the external
application

// a list of "Direct HTML" tags
$html_special = $fbml_impl->get_special_html_tags();

 // a list of FBML-specific tags (<fb:FOO>)
$fbml_tags = $fbml_impl->get_all_fb_tag_names();

// attributes of all tags to rewrite specially
$rewrite_attrs = array('onfocus', 'onclick', /* ... */);

// this defines the tag groups passed to flavor's check() function
// (e.g. 'images', 'bold', 'flash', 'forms', etc.)
$fbml_schema = schema_get_schema();

// Send the constraints and callback method names along
// to the internal C FBML parser.
fbml_complex_expand_tag_list_11($fbml_tags, $fbml_attrs,
  $html_special,$rewrite_attrs, $fbml_schema);

$parse_tree = fbml_parse_opaque_11($fbml_from_callback);
$fbml_tree = new FBMLNode($parse_tree['root']);

$html = $fbml_tree->render_html($fbml_impl);
```

FBML利用回调扩展了浏览器的解析技术，包装了由Facebook创建和管理的数据、执行和展现宏。这个简单的思想实现了应用的完全集成，支持使用通过API暴露出来的内部数据，

同时保持安全性方面的用户体验。FBML本身几乎就是一种编程语言，它也是充分发展后的数据：外部提供的声明式执行，安全地控制了Facebook上的数据、执行和显示。

6.5 系统的支持功能

现在，开发者创建的软件运行在Facebook的服务之上，不仅是结合了界面组件，而是全部的应用。在这个过程中，我们创造了一个社会关系网络应用的完全不同的概念。我们从一个典型的Web应用的独立数据、逻辑和显示的标准设置开始，不考虑所有社会关系数据，只是让用户可以确信能够作出贡献。现在，我们取得了充分的进展，应用使用了Facebook的社会关系数据服务，同时它自己又成为一个FBML服务，完全集成到容器站点之中。

Facebook数据也获得了长足的发展，不再仅仅是本章第一节讨论的内部库。但是，仍有一些重要的、常见的Web使用场景和技术，目前平台还未能支持。通过将应用变成一个返回FBML的服务，而不是直接由浏览器解读的HTML/CSS/JS，我们接触到了关于现代Web应用的一些重要假定。让我们来看看Facebook平台如何修正这样一些问题。

6.5.1 平台cookie

应用的新Web架构排除了浏览器内建的一些技术，许多Web应用栈可能依赖于这些技术。可能其中最重要的一点是，过去浏览器用于保存用户与应用栈交互信息的cookie不再可以得到了，因为应用的目标消费者不再是浏览器，而是Facebook平台。

初看上去，伴随对应用栈的请求发送一些cookie似乎是一个不错的解决方案。但是，这些cookie的作用域现在是“*http://facebook.com*”，而实际上，cookie信息属于该应用领域所提供的用户体验。

解决方案是什么？让Facebook具有浏览器的职责，在Facebook自己的存储库中复制这种cookie功能。如果应用的FBML服务送回请求头，试图设置浏览器cookie，Facebook就保存这个cookie信息，以（user，application_id）对为主键。Facebook然后“重新创建”这些cookie，就像用户向这个应用栈发出后续请求时浏览器所做的一样。

这个解决方案很简单，在开发者从HTML栈方式转向FBML服务方式转变时，只需要很少的改变。请注意，当用户决定在这个应用提供的HTML栈上导航时，这种信息是不能使用的。另一方面，它可以有效地分离用户在Facebook上的应用体验和在应用的HTML站点上的应用体验。

6.5.2 FBJS

当应用栈作为一个FBML服务被使用，而不是直接由用户的浏览器来使用，Facebook就

没有机会执行浏览器端的脚本。直接返回未修改过的开发者提供的内容（一个不充分的解决方案，这在FBML小节的一开始就讨论过）可以解决这个问题，但它违反了Facebook在显示体验上所加的约束。例如，当加载用户的简介页面时，Facebook不希望在加载事件上触发一个弹出窗口。但是，限制所有的JavaScript会排除许多有用的功能，如AJAX或在不重新加载的情况下动态操作页面的内容。

相反，FBML在解释开发者提供的<script>树和其他页面元素的内容时会考虑到这些约束。在此之上，Facebook提供了一些JavaScript库，让这些场景容易实现，同时又受到控制。这些修改共同构成了Facebook的平台JavaScript仿真套件，称为FBJS，它通过以下几点，让应用既动态又安全：

- 重写FBML属性，确保实现虚拟文档范围。
- 延迟激活脚本内容，直到用户在页面或元素上发起动作时。
- 提供一些Facebook库，以受控的方式来实现常见的脚本使用场景。

很清楚，不是所有的实现自有平台的容器站点都需要这些修改，但FBJS向我们展示了几种解决方案，这样的新Web架构需要这些解决方案来绕过一些困难。我们在这里只展示了这些解决方案的一般思想，FBJS的许多部分还需要不断改进，与FBML和可扩展的专有JavaScript库进行融合。

首先，JavaScript通常可以访问包含它的文档的整个文档对象模型（DOM）树。但是在平台画布页面中，Facebook包含了许多它自己的元素，开发者不允许对它们进行修改。解决方案是什么？在用户提供的HTML元素和JavaScipt符号之前加上前缀，即应用的ID（如app1234567）。通过这种方式，在开发者的JavaScript中如果试图调用不允许调用的alert()函数，就会调用未定义的函数app1234567_alert，并且只有开发者自己提供的那部分文档的HTML可以被document.getElementById这样的JavaScript代码访问。

作为FBJS需要对提供的FBML（包括<script>元素）进行这种转换的一个例子，我们创建了一个简单的FBML页面，实现了AJAX功能，如例6-30所示。

例6-30：一个使用FBJS的FBML页面

```
These links demonstrate the Ajax object:
<br /><a href="#" onclick="do_ajax(Ajax.RAW); return false;">AJAX
Time!</a><br />
<div>
<span id="ajax1"></span>
</div>

<script>
function do_ajax(type) {
  var ajax = new Ajax(); // FBJS Ajax library.
  ajax.responseType = type;
```

```
  switch (type) {
   <!-- note FBJS's Ajax object also implements AJAX.JSON and AJAX.FBML,
omitted
  for brevity -->
    case Ajax.RAW: ajax.ondone = function(data) {
      document.getElementById('ajax1').setTextValue(data);
    };
    break;
  };

 ajax.post('http://www.fettermansbooks.com/testajax.php?t='+type);
}
</script>
```

FBML和我们的FBJS修改动作将这些输入转变成了例6-31中的HTML。这个例子中的NOTE注释指出了每种需要的转换，不是实际输出的一部分。

例6-31：HTML和JavaScript输出的例子

```
<!-- NOTE 1-->
<script type="text/javascript" src="http://static.ak.fbcdn.net/
.../js/fbml.js"></script>

<!-- Application's HTML -->
These links demonstrate the Ajax object:
<br>
<!-- NOTE 2 -->
<a href="#" onclick="fbjs_sandbox.instances.a1234567.bootstrap();
 return fbjs_dom.eventHandler.call(
[fbjs_dom.get_instance(this,1234567),function(a1234567_event) {
a1234567_do_ajax(a1234567_Ajax.RAW);
return false;
}
,1234567],new fbjs_event(event));return true">
AJAX Time!</a>
<br>

<div>

<span id="app1234567_ajax1" fbcontext="b7f9b437d9f7"></span><!-- NOTE 3
-->
</div>

<!-- Facebook-generated FBJS bootstrapping -->
<script type="text/javascript">
var app=new fbjs_sandbox(1234567);
app.validation_vars={ <!-- Omitted for clarity -->};
app.context='b7f9b437d9f7';
app.contextd=<!-- Omitted for clarity -->;
app.data={"user":8055,"installed":false,"loggedin":true};
app.bootstrap();
</script>
```

```
<!-- Application's script -->

<script type="text/javascript">
function a1234567_do_ajax(a1234567_type) { <!-- NOTE 3 -->

var a1234567_ajax = new a1234567_Ajax();<!-- NOTE 3 -->
 a1234567_ajax.responseType = a1234567_type;
 switch (a1234567_type) {
  case a1234567_Ajax.RAW:
a1234567_ajax.ondone = function(a1234567_data) {
a1234567_document.getElementById('ajax1').setTextValue(a1234567_data);
};
break;
};

<!-- NOTE 4 -->
a1234567_ajax.post('http://www.fettermansbooks.com/testajax.php?t='+a1234
567_type);
}
</script>
```

下面是这段代码中的NOTE的解释：

NOTE 1

Facebook需要包含它自己的特殊JavaScript，包括fbjs_sandbox的定义，目的是渲染开发者的脚本。

NOTE 2

还记得前面FBML初始化流程中的$rewrite_attrs元素吗？FBML会重写这个列表中的属性，变成Facebook特有的功能，这实际上是FBJS的一部分。所以这里的onclick会激活这个页面的其他元素，这些元素在用户执行这个动作之前是非激活的。

NOTE 3

请注意在HTML和脚本中的元素如何加上了该应用的应用ID作为前缀。这意味着开发者对alert()的调用将变成对app1234567_alert()的调用。如果Facebook的后台JavaScript在这个上下文中允许这个方法，它将最终转向执行alert()。如果不允许，这将是未定义的调用。类似地，这种加前缀的方式实际上为DOM树提供了命名空间，所以对该文档某些部分的改变只限于开发者定义的那些部分。类似的沙盒技术也允许开发者提供限制范围的CSS。

NOTE 4

Facebook提供了一些专门的JavaScript对象，如Ajax和Dialog，目的是支持（并且常常改进了）常见的使用场景。例如，通过Ajax()对象发出的请求实际上能获得FBML作为结果，所以它们被重定向到Facebook域的一个代理上，在这里Facebook完成在线的FBML到HTML的转换。

支持FBJS需要对FBML进行改动、专门的JavaScript和AJAX代理这样的服务器端组件，才能够绕过应用Web架构的一些限制，但结果是很强大的。开发者因此可以享受绝大多数的JavaScript功能（甚至改进了这些功能，如支持FBML的AJAX），而且平台确保了应用内容提供了用户在Facebook上期望的受控体验，这完全是通过技术手段来实现的。

6.5.3 服务改进小结

解决了新的*n*层社会关系应用的概念带来的剩下一些问题之后，我们又改进了服务架构，添加了COOKIE和FBJS等项，如图6-6所示。

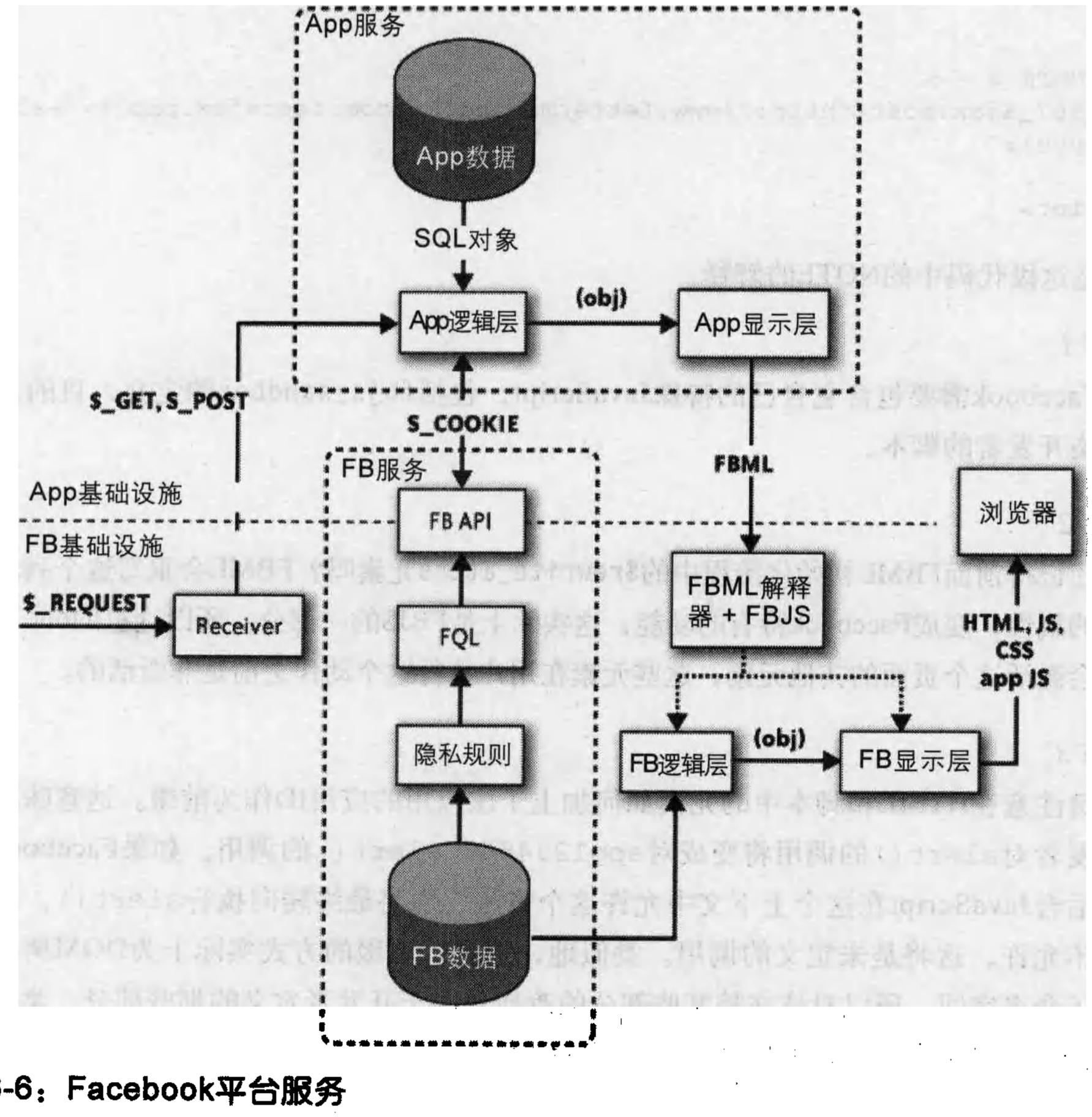


图6-6：Facebook平台服务

随着开发者的社会关系应用越来越成为Facebook使用的一项集成服务，而不是由浏览器使用的外部站点，我们已经重新创建或重新设计了浏览器的某些功能（通过平台cookie、FBJS等）。在试图改变或重建“应用”的概念时，这是必需的两个重要修改的例子。

Facebook平台包括类似的其他一些架构上的巧妙设计，这里没有详细介绍，其中包括数据存储API和浏览器端Web服务客户端。

6.6 总结

Facebook的用户贡献的社会关系有效地提高了*http://facebook.com*上几乎所有页面的效用。而且，这种数据非常通用，所以当它与外部开发者的应用栈结合在一起时，它的最佳使用就出现了，这都是通过Facebook平台的Web服务、数据查询服务和FBML等技术来实现的。从取得用户的朋友或简介信息的简单内部API开始，我们在本章中详细介绍的全部改进展示了如何协调不断扩展的数据访问方法和容器网站的预期，特别是对数据隐私和站点体验集成方面的要求。每次对数据架构的新改动都发现了Web架构的一些新问题，我们又通过对数据访问模式的更强改进来解决这些问题。

虽然我们将关注重点完全放在那些使用Facebook的社会关系数据平台的应用的潜力和约束上，但像这样的新型数据服务不一定局限于社会关系信息。随着用户贡献和使用的信息越来越多，这些信息在许多容器站点上都很有用（如内容收集、评论、位置信息、个人计划、协作等数据），各式各样的平台提供者可以应用Facebook平台特有的数据和Web架构背后的这些思想，并从中获益。

[redacted]

第三部分

系统架构

	原则与特性		结构
√	功能多样性	√	模块
	概念完整性	√	依赖关系
√	修改独立性		进程
	自动传播		数据访问
	可构建性		
√	增长适应性		
	熵增抵抗力		

第7章

Xen和虚拟化之美

Derek Murray
Keir Fraser

7.1 简介

Xen是一个虚拟化平台，它来自于一项学术研究成果，现已发展成一个重要的开源项目。它容许用户在单个物理机器上运行几个操作系统，并特别强调性能、隔离性和安全性。

Xen项目在多个领域造成了很大的影响：从软件到硬件，从学术研究到商业开发。它的成功很大一部分应归于采用了GPL（GNU General Public License）的开源发布方式。然而，开发人员并非某一天突发奇想决定编写一个开源的系统管理程序。它最初是一个大的（甚至野心勃勃的）研究项目Xenoservers的一部分。这个项目催生了开发Xen的动机，所以，在这儿我们将用它来解释虚拟化的必要性。

把Xen开源，这不仅使大量的用户可以使用它，也使它能够和其他开源项目享有共生关系。Xen的独特之处在于，当它第一次发布时，使用了*半虚拟化*（Paravirtualization）来运行像Linux这样的常规操作系统。半虚拟化需要修改运行在Xen之上的操作系统，既提升性能又简化了Xen自身。然而，半虚拟化只能做到这样，而且只有在处理器供应商的硬件支持下，Xen才能运行未修改的操作系统，如Microsoft Windows。处理器发展的前沿之一是增加新特性来支持虚拟机并消除一些性能开销。

随着新特性的增加及新的硬件变得可用，Xen的架构正在慢慢地发展。然而，从最初的原型到现在的版本，它的基本结构一直保持不变。在这一章中，我们介绍Xen的架构如何从早期的一个研究项目经过三个主要的版本，一直演变到目前的情况。

7.2 Xenoservers

与Xen相关的工作于2002年4月在剑桥大学开始。它最初作为Xenoservers项目的一部分，而Xenoservers的目标是创建一个“全球的分布式计算结构”。

几乎在同一时期，网格计算（grid computing）发展为利用分布于全球的计算资源的最好方式。最初的网格方案把计算机时间塑造成一种能力，就像电一样，这种能力可以通过协作计算机的网络或网格来获得。然而，后续的实现集中到了虚拟组织（virtual organization）上：可能建立了复杂信任关系的公司或机构组，这通过采用重量级的公钥加密来认证和授权而得以加强。

Xenoservers从相反的方向来解决这个问题。顾客通过一个代理（众所周知的XenoCorp）选择公开市场上的一个资源，而不是和服务提供商形成信任关系。 XenoCorp保存一系列的xenoservers（由第三方提供出租的计算机）把顾客与服务器相对应，根据使用情况收取费用并支付报酬。至关重要的是，在顾客和提供商之间是相互不信任的：顾客不能损坏提供商的机器，提供商不能篡改顾客的工作。

信任

不信任是一个有用的架构特征，这听起来似乎违反常识。然而，在这个环境中，安全的主要目标是防止其他人访问或破坏敏感数据。因此，可信的系统是一个允许访问数据的系统。当不信任被构建入架构时，可信组件的数量会减到最少，并由此默认地提供了安全性。

下面开始讨论虚拟化。提供商为顾客提供一个新的虚拟机任其使用，而不是为顾客提供服务器上的一个账号。然后，顾客可以运行任何操作系统和任何应用程序（参见图7-1）。

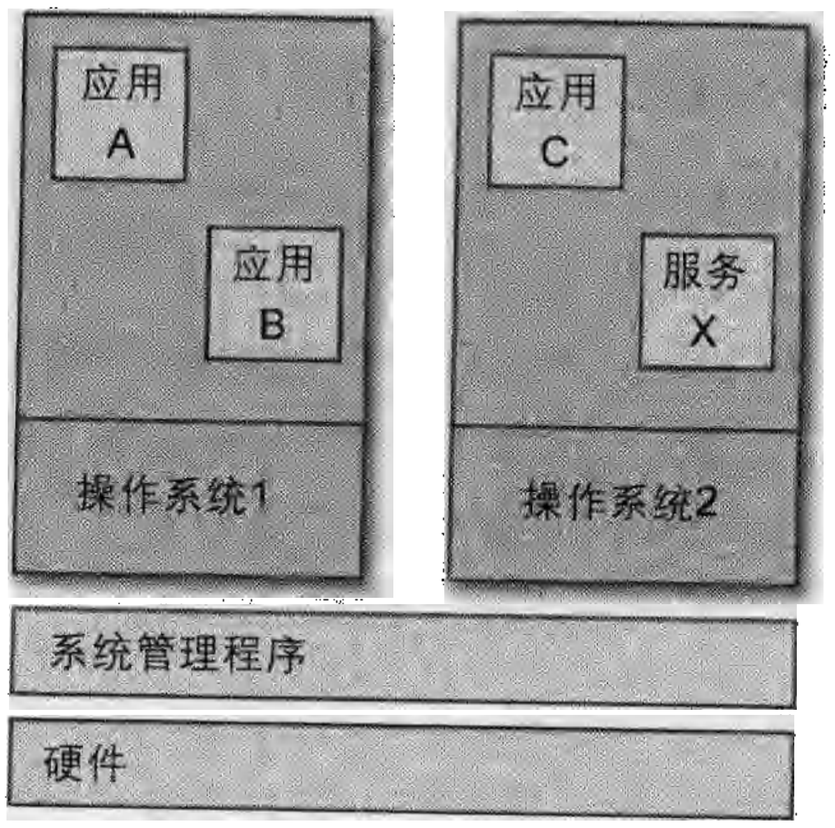


图7-1：虚拟机架构

虚拟软件确保这些与机器的其他部分隔离开，其他部分可能会租赁给更多的顾客。运行在虚拟机上面的系统管理程序（hypervisor）包含两个主要部分：一个引用监控器和一个调度程序，引用监控器确保一个虚拟机不能访问另一个虚拟机的资源（尤其是虚拟机的数据），调度程序确保每个虚拟机平等共享机器的CPU。

你不能只使用一个操作系统？

自20世纪60年代以来，分时操作系统就已经存在了，它能够使几个相互不信任的用户同时运行进程。比如说，在基于UNIX的机器上给每个用户一个账号还不够吗？

这确实让用户共享了计算的资源。然而，这不能令人满意，因为用户不具有灵活性和性能隔离。

在灵活性方面，用户只能运行与这台机器的操作系统兼容的软件，没有办法运行一个不同的操作系统，或改变操作系统。确实，（没有管理员的支持）用户不可能安装需要root权限的软件。

至于性能隔离，对于一个操作系统内核（这是一个极端复杂的软件块）来说，计算某一用户使用的所有资源比较困难。fork炸弹（fork bomb）就是一个例子，在这里一个用户的进程数呈指数级增长。这会迅速地消耗掉处理器的所有资源并导致对其他用户拒绝服务。因此，多用户系统通常需要用户之间有一定的信任和规范，以便不会出现这样的攻击。

在图7-1中，我们看到两个虚拟机运行在一个系统管理程序上。第一个虚拟机运行操作系统1（例如Microsoft Windows）和两个应用程序；第二个虚拟机运行操作系统2（例如Linux）、一个应用程序和一个服务。

这表明虚拟化对于其他事情也有用。例如，在许多数据中心，每个应用程序都有一个专门的服务器可用，如一个数据库服务器或一个Web服务器，但是，每个服务器只使用它的一小部分处理器。然而，仅仅在同一个操作系统上运行那些应用程序会引起糟糕的结果。当各种各样的应用程序一起运行时，可能会导致不可预知的差的性能。更糟糕的是，存在关联失败的风险，当一个应用程序崩溃时，也会引起其他应用程序崩溃。通过把每个应用程序单独放在一个虚拟机，然后在一个系统管理程序之上运行它们，系统管理程序可以保护这些应用程序并确保每个应用程序平等共享服务器的资源。

用虚拟化来进行效用计算（utility computing）的观点在最近几年开始流行。最知名的效用计算服务之一是Amazon的EC2，它允许顾客在Amazon数据中心的服务器上创建虚拟机。然后，顾客为他的虚拟机所使用的处理器时间和网络带宽付费。实际上，这些服务器运行Xen作为它们的虚拟化软件，使得它更接近于Xenoservers（虽然它只容许单个服务提供商）。

虚拟化对网格计算也有影响。网格计算事实的标准中间件Globus，现在支持虚拟工作空间（virtual workspace），它把虚拟机和现有的网格安全及资源管理协议结合在了一起。这样做的一个额外的好处是，如果条件改变，一个虚拟工作空间（就像任意的虚拟机）可以迁移到另一个物理位置。

采用Xenoservers模式的虚拟机的关键优势是它们可以用来运行流行的操作系统和现有的应用程序。实际上，这意味着它运行在主流的x86架构之上，这给系统管理程序的开发人员提出了一些挑战。

7.3 虚拟化的挑战

从高层次看来，操作系统虚拟化用于把几个虚拟机复合到单台物理机器上。虚拟机可以运行操作系统；物理机器也可以运行操作系统。那么，在虚拟机和物理机器之间有什么区别呢？

硬件是最显而易见的区别。在一台物理机器上，操作系统直接控制所有相连的硬件：网卡、硬盘驱动器、显卡、鼠标和键盘。然而，虚拟机不能直接访问这些硬件，否则它们将破坏每个虚拟机之间的隔离性。例如，某个虚拟机可能不想让其他的虚拟机查看它在二级存储器中存储的内容，或去读取它的网络信息包。此外，确保调度中的公平使用也会很困难。你可以为每个虚拟机配置某种类型的设备，但是这会给虚拟化的开销和能力带来负面影响。解决办法是给每个虚拟机一组虚拟硬件，这组虚拟硬件提供和真实硬件一样的功能，但是，这组虚拟硬件会复用物理设备。

当操作系统在虚拟机中运行时，一个更微妙的区别出现了。一般来说，操作系统内核是运行在计算机上的最有特权的软件，这允许它执行用户程序不能执行的某些指令。在虚拟化的情况下，系统管理程序是最有特权的，而操作系统内核在相对较低的权限下运行。如果操作系统现在试图执行这些指令，那它们就会失败，但是它们失败的方式是至关重要的。如果它们引起系统管理程序会捕获的一个错误，系统管理程序可以恰当地模拟这个指令并继续控制虚拟机。然而，在x86上存在一些在较低权限下表现不同的指令——例如，默默地失败而不触发系统管理程序。对于虚拟化来说这是坏消息，因为这妨碍了操作系统在虚拟机上适当地工作。显然，必须改变这些指令，普遍的技术（至少在Xen之前）是在运行时扫描操作系统代码，寻找特定的指令并用直接调用系统管理程序的代码替换它们。

实际上，在Xen之前，大多数虚拟化软件打算使虚拟硬件看起来完全像物理硬件。所以，虚拟设备就像物理设备一样运转，模拟相同的协议，当重写代码时确保操作系统不做修改就能运行。虽然这具有理想的兼容性，但在性能上有很大的开销。当发布Xen时，它表明，通过放弃理想的兼容性，性能会显著地提高。

7.4 半虚拟化

半虚拟化的观点是移除一个架构（例如x86）中难以虚拟化或如果进行虚拟化开销很大的所有特性，用与虚拟化层直接通信的半虚拟化操作替换它们。Denali中第一次使用了这种技术，Denali是移植了专门编写的Ilwaco客户操作系统的虚拟机监视器。Xen通过运行商业操作系统的半虚拟化版本而更进了一步。（注1）

一个操作系统的半虚拟化包括重写与半虚拟化架构不兼容的所有代码。性能改进了，因为开发人员已经预先进行了转换，而不是在运行时。为了展示半虚拟化的能力，Xen团队需要一个他们可以改变的操作系统。幸运的是，可以使用Linux，它是开源的，而且广泛使用。对Linux内核只修改或增加2995行代码，就可以使它能够在Xen上运行：这连x86 Linux代码的2%都不到。通过半虚拟化（就像通过虚拟化一样），所有现有的用户应用程序都可以继续使用而不作任何修改，所以，总的修改不是非常扩散的。

为了实现半虚拟化，你必须自己写操作系统（Denali的方式），或修改一个现有的开源操作系统（例如Linux或BSD），或使得操作系统开发人员确信半虚拟化代码是值得的。最初的Xen研究版本在修改过的Linux上运行时在许多方面获得了令人印象非常深刻的接近本地的性能。性能，还有Xen作为开源软件进行发布的事实，使得操作系统开发人员开始半虚拟化部分的代码，以便他们的操作系统可以在Xen的系统管理软件上更高效地运行。更鼓舞人心的是，为Xen和其他系统管理软件开发的半虚拟化操作已经标准化到最新版的Linux内核中。通过在标准内核中支持Xen（和其他系统管理软件），虚拟化的推进变得容易得多。

半虚拟化是如何工作的呢？全部的细节太过复杂以致不能在这里进行阐述，但本章末尾的7.8节列出了深入讨论这一技术的论文。这里，我们将探究两个半虚拟化的例子：虚拟内存和虚拟设备。

半虚拟化一个操作系统的第一步是意识到它不是运行在这台计算机上最有权限的软件；那个级别的权限给予了系统管理软件。大部分的处理器至少拥有两个模式：超级用户（supervisor）和用户（user）模式。通常，操作系统内核会以超级用户的模式运行，但这已经留给了Xen，所以它必须修改为以用户模式运行。（注2）然而，当以用户模式运行时，有几个操作是不合法的。在一个常规的操作系统中，进程之间必须相互保护，这

注1：当然，关于VM/370（20世纪60年代IBM的操作系统，虚拟化的起源）是第一个半虚拟化操作系统的说法可能存在争议。然而，自从IBM设计了指令集、操作系统和虚拟机监视器，这种方式应对了不同的挑战，直到现代的半虚拟化。

注2：x86架构有四个权限级别（或称为ring），0表示最大权限，3表示最小权限。在它的32位版本中，Xen以ring 0运行，半虚拟化内核以ring 1运行，用户应用程序和往常一样，以ring 3运行。然而，在64位版本中，由于内存分段硬件的差异，半虚拟化内核以ring 3运行。

至关重要。因此，内核必须使用一种名为超级调用（hypercall）的机制来要求系统管理程序代表它执行这些操作。超级调用类似于系统调用（从用户进程调用内核），只不过它用于在内核和系统管理程序之间通信，而且它通常实现较低级别的操作。

虚拟内存用来确保进程不能和其他进程的数据或代码进行交互。每个进程都给定虚拟地址空间，这确保那个进程只能存取分配给它的内存。内核通过维护页表（页表把虚拟地址映射到确定内存芯片中数据的实际位置的物理地址）来负责创建虚拟地址空间。当它在一个虚拟机中运行时，内核不能自由地管理这些表，因为它可以映射到属于另一个虚拟机的内存。因此，Xen必须验证页表的所有更新，而当内核想要修改任何页表时必须通知系统管理程序。如果系统管理程序涉及每个页表更新，那么效率会非常低（例如，当启动一个新的进程并最初构建它的页表时）。然而，结果表明这种情况相对较少，而且，Xen可以通过批处理这些更新请求或在页表更新时“松开”页表来分摊转给系统管理程序的开销。

请看图7-2。所有的虚拟机分享了整个物理内存。（注3）然而，这些内存可能不是相邻的，而且，在大多数情况下也不会从物理地址0开始。因此，每个虚拟机内核都处理两种类型的地址：物理地址（或机器地址）和伪物理地址。物理地址相应于内存芯片中数据的实际位置，而伪物理地址向虚拟机提供了从0开始的连续物理地址空间的假象。伪物理地址对于依赖这个假定的、否则必须半虚拟化的某些算法或子系统是有用的。

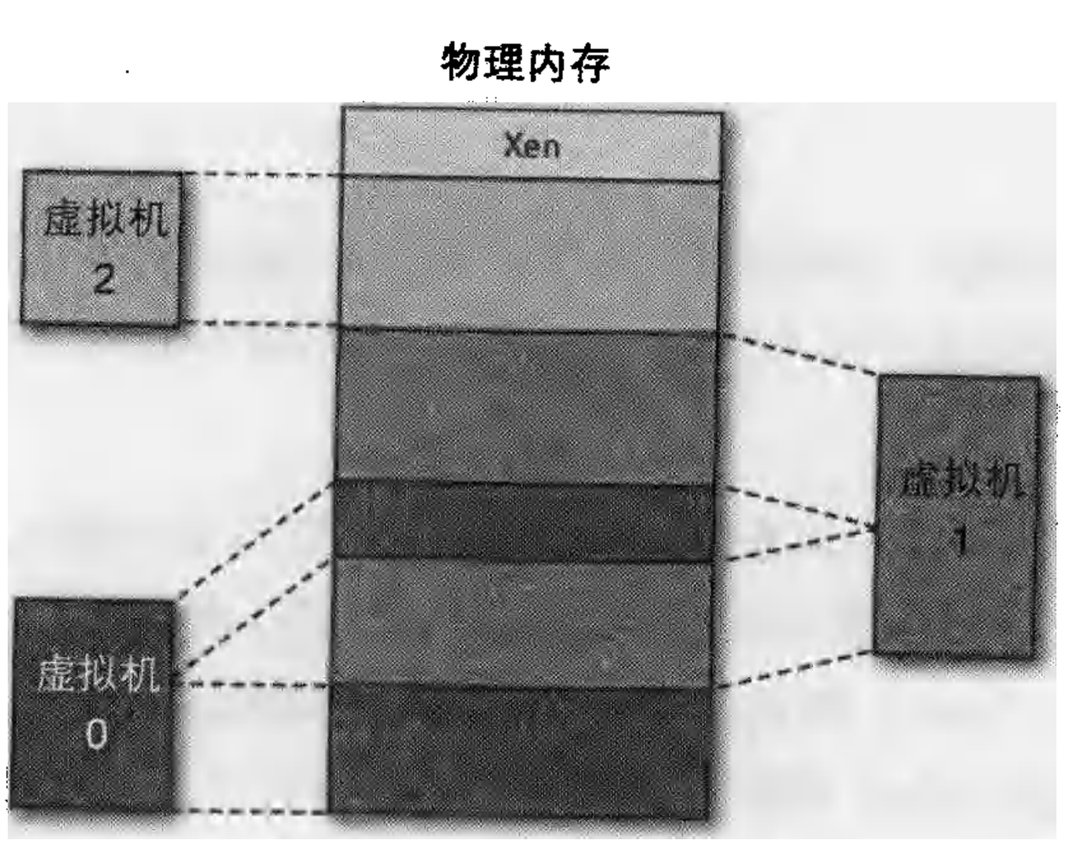


图7-2：虚拟机内存配置

注3：　请注意，Xen不支持过量使用物理内存，所以没有虚拟机的交换技术。然而，可以利用一个名为ballooning的进程改变虚拟机的内存覆盖区。

注4：　你可能认为对于交互来说鼠标、键盘和视频输出是必需的，但是，这些可以由像VNC这样的远程桌面客户端提供。不过，最新版本的Xen已经包括了对这些虚拟设备的支持。

在任何实际的使用中，都必须能够和虚拟机进行交互。虚拟机至少需要一个磁盘（严格来说，称之为块设备）和一块网卡。（注4）因为大多数操作系统至少支持一个块设备和一块网卡，让系统管理程序模拟这些设备以便可以使用最初的驱动程序，这似乎很诱人。然而，软件的实现会努力模拟实际设备的性能，模拟设备的模式可能不得不绕弯路（例如实现硬件协议），用软件实现硬件的功能是不必要和低效的。

还有其他实现方式吗？

虚拟化虚拟内存（当你不能改变操作系统时）的标准方式是使用影子页表。通过它们，访客会处理伪物理地址（例如，相邻的和从0开始的地址）而不是物理地址。

访客靠着这个地址空间维护他自己的页表。然而，它们不能由硬件使用，因为它们并不对应真实的物理地址。因此，系统管理程序更新这些访客页表，并用它们构建一张会把虚拟地址转化为物理地址的影子页表。

这个方法无疑会有一些开销，但是，当你不能修改操作系统时这是必需的。

Xen对硬件虚拟化的访客使用了这个方法的一个变体，正如本章后面所述。

因为Xen不必一定支持未修改的操作系统，所以它可以自由地引入虚拟块和网卡驱动。两者以相似的方式运作：它们包含访客虚拟机中的前端驱动和虚拟软件中的后端驱动。这两个设备利用环形缓冲进行通信，环形缓冲是用来在虚拟机之间传输大容量数据的一种高性能机制。这产生一个柔性层架构（如图7-3所示）：前端实现了操作系统的网络或块设备接口以便它对于操作系统就像是一个常规的硬件设备，后端把这个虚拟设备和真实的硬件连接起来。虚拟块设备可能会连接到一个包含磁盘映像的文件或一个实际的磁盘分区；虚拟网卡设备可能会连在一个软件网桥上，而它本身连在一块真实的网卡上。环形缓冲这个抽象概念确保前端和后端完全分离。一个后端可以支持来自Linux、BSD或Windows的前端，反之，相同的前端可以和各种后端一起使用，所以，像写时复制（copy-on-write）、加密和压缩这样的特性可以显而易见地加给访客。就像互联网协议，Xen的分设备模式可以操作大批硬件，而且它支持多数较高级别的客户端，如图7-3所示。

半虚拟化包括的内容远远超过这些例子。例如，当一个操作系统可以从CPU换出时，时间的意义就改变了，而Xen引入了虚拟时间的概念来确保操作系统仍然像预期的那样运行。还有许多其他的虚拟设备，内存的半虚拟化利用其他的优化来确保高效的性能。如果想要了解更多关于这方面的资料，请阅读本章末尾的7.8节。

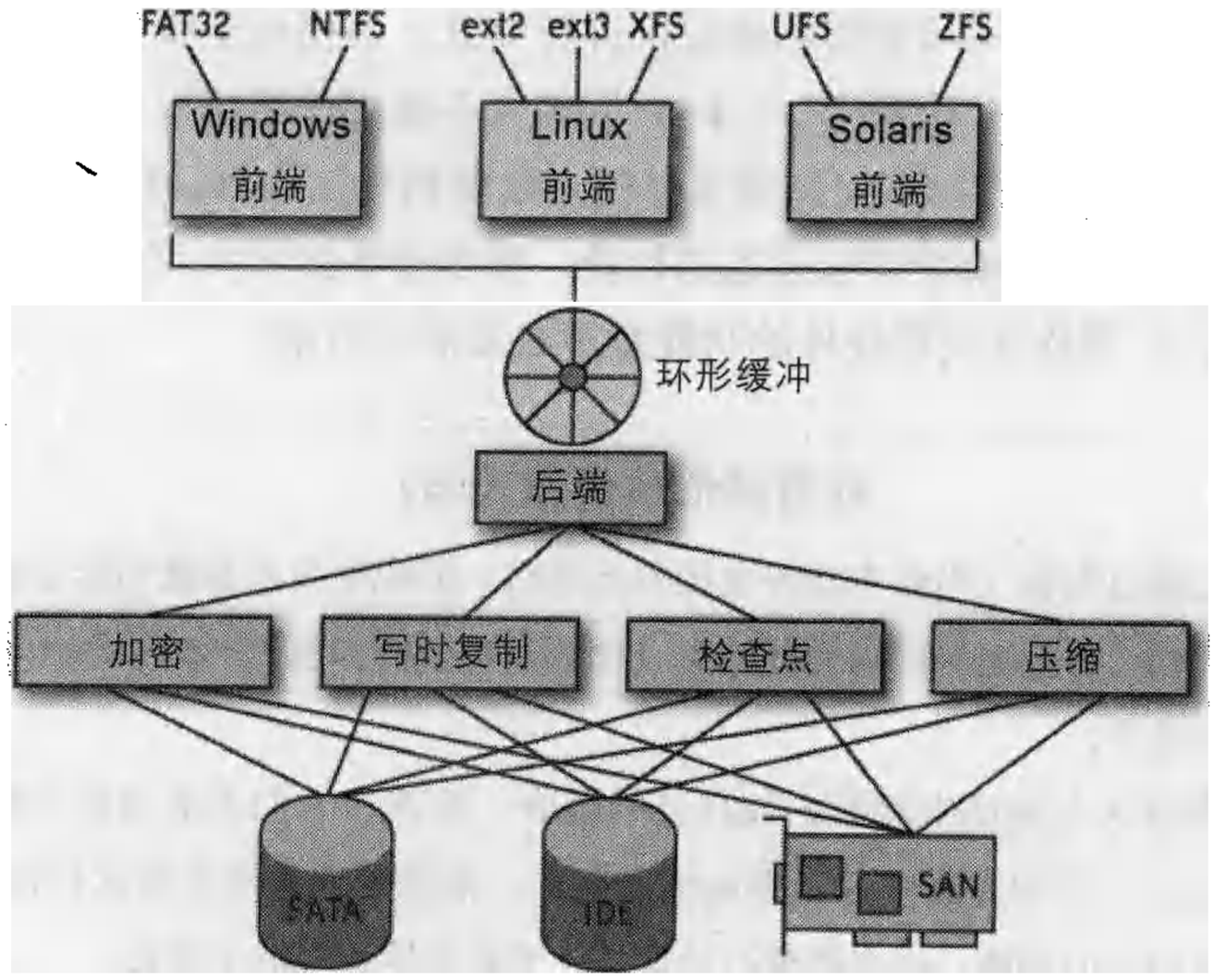


图7-3：分块设备的沙漏型架构

7.5 Xen的变换形式

一个以Xen为基础的系统的传统描述表明几个虚拟机（Xen中称为域（domain））设置在系统管理程序之上，而系统管理程序本身直接设立在硬件上（如图7-4所示）。当系统启动时，系统管理程序启动一个特殊的域，即众所周知的零域（domain zero）。零域拥有可以让它管理系统其余部分的特殊权限，它类似于常规操作系统中的一个根或管理员进程。图7-4显示了一个基于Xen的典型的系统，其中，零域和几个客户域（Xen术语中称为DomU）运行在Xen系统管理程序之上。

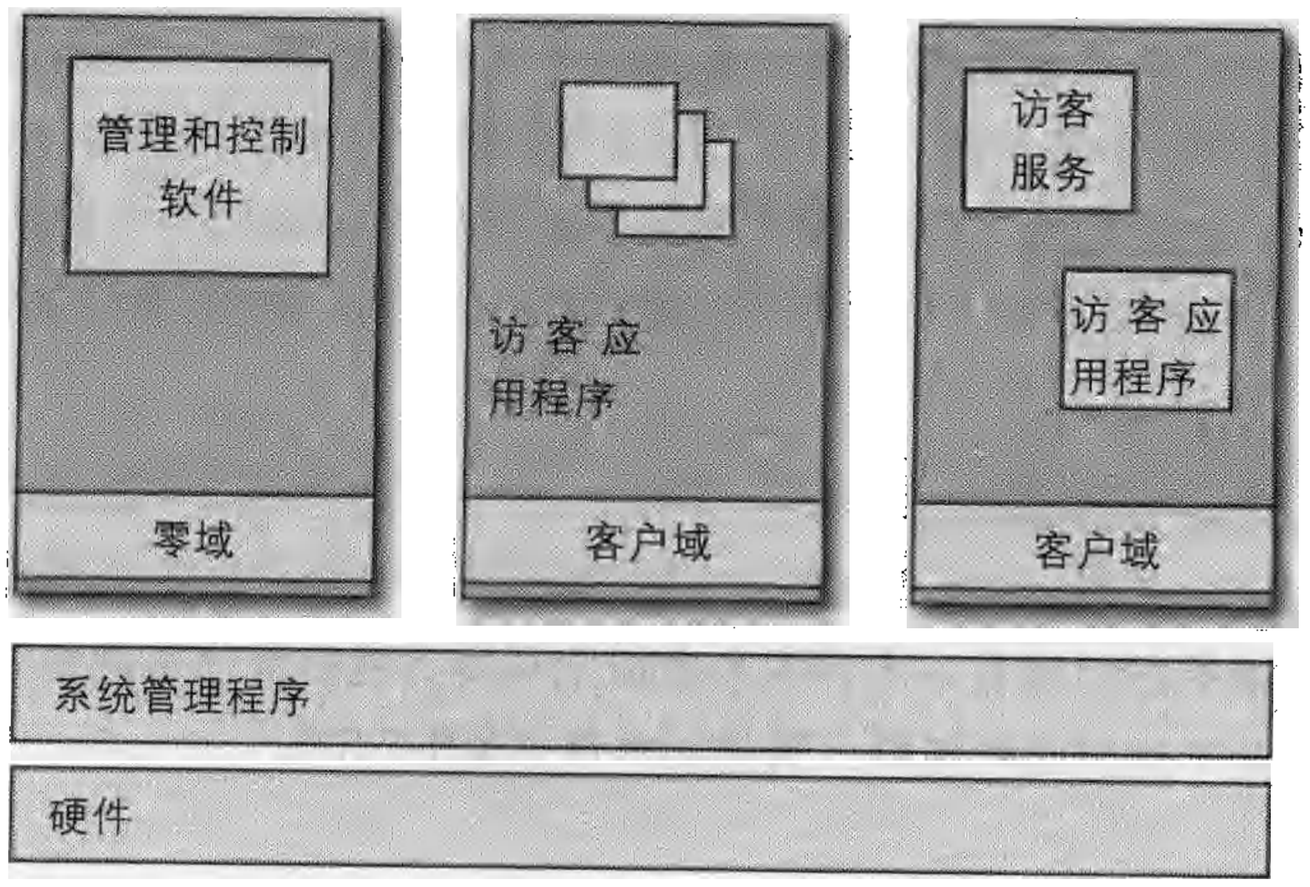


图7-4：Xen系统架构

宿主式虚拟化

Xen是原生式虚拟化（也被称为第一类虚拟化）的一个例子。另一种方式是在宿主操作系统上运行一个系统管理程序。在这种情况下，每个虚拟机实际上成了宿主操作系统的一个进程。宿主操作系统负责零域在Xen上执行的管理职能。宿主式的系统管理程序和管理软件就像一个常规的应用程序，设置（或联结）在一个常规操作系统之上，如图7-5所示。

宿主式系统管理程序通常用于其他虚拟化产品的“工作站”版本中，例如VMWare工作站、Parallels工作站和Microsoft的虚拟PC。这种方式的主要优点在于：安装一个宿主式的系统管理程序就像安装一个新应用程序一样简单，反之，安装一个原生的系统管理程序（例如Xen）就像安装一个新的操作系统。因此，宿主式虚拟化更适合于非专家的用户。

另一方面，原生式系统管理程序的优点在于它可以获得更好的性能，因为本地系统管理程序比组合了宿主式操作系统和系统管理程序的软件层薄很多。宿主式虚拟机受宿主操作系统支配，如果有其他应用程序和系统管理程序正一起运行，就会导致性能降低。相反，因为零域像一个常规虚拟机一样调度，在那里运行的应用程序对其他虚拟机的性能没有影响。

宿主式虚拟机通常用于桌面虚拟化：例如，允许运行Mac OS X的用户在计算机的一个窗口中运行Linux。这对于要运行宿主操作系统不支持的应用程序有用，而且，当使用交互式应用程序时，性能影响不太容易注意到。原生式虚拟化更适合于服务器设施，在那里，初始的性能和可预测性至关重要。

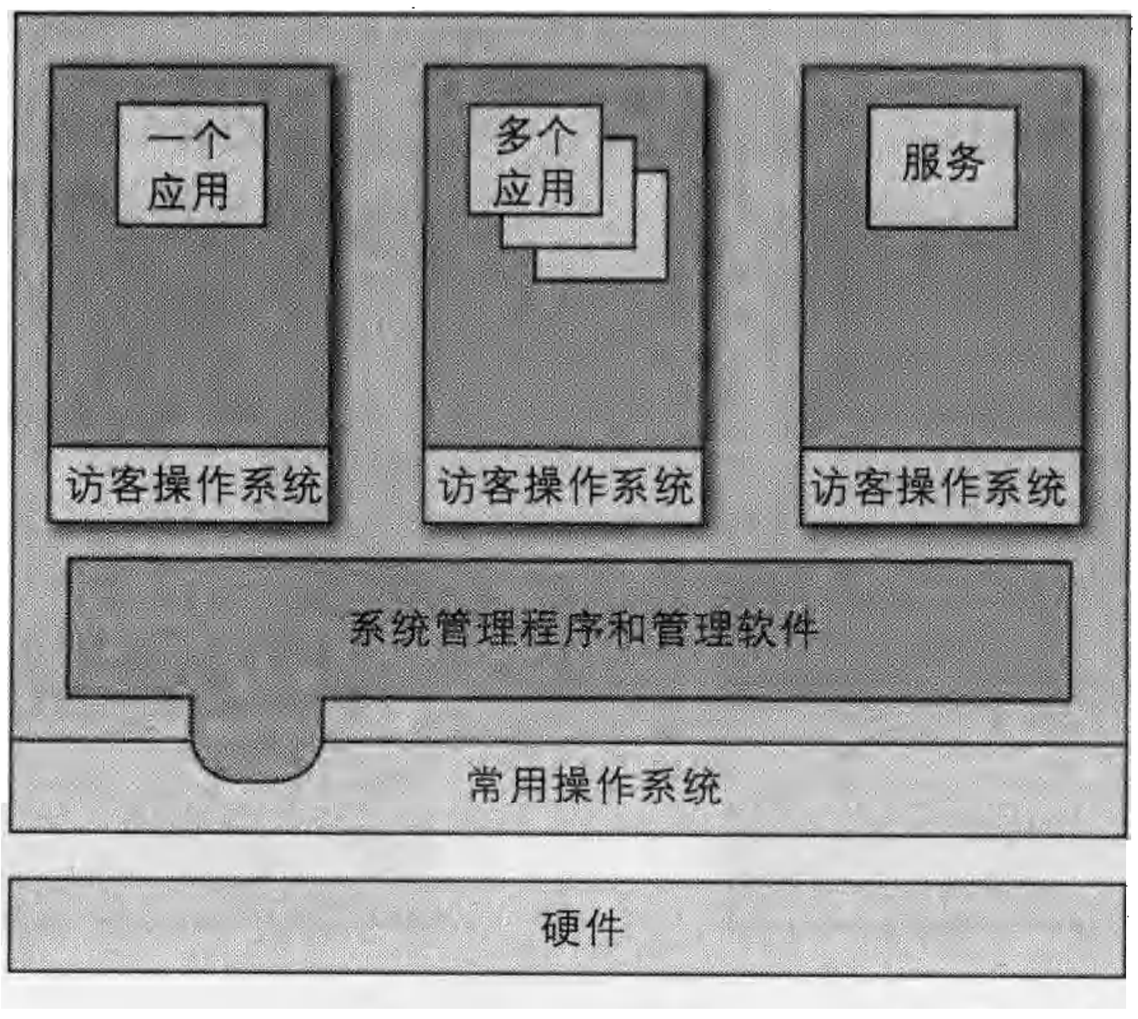


图7-5：宿主式虚拟化系统的架构

当设计Xen架构的时候，首要关注的是只要有可能，就把策略和机制分开来。系统管理程序设计成用来管理低级硬件的薄层，担当引用监控器和调度程序，复用对硬件设备的访问。然而，因为系统管理程序在最高级别运行（这里的一个缺陷会危及整个系统），较高级别的管理委派给了零域。

例如，当创建一个新的虚拟机的时候，大部分工作是在零域中完成的。从系统管理程序的角度来看，它分配了新的域，连同一部分物理内存，还映射了其中的部分内存（为了载入操作系统），并且这个域是未中止的。零域负责许可权限控制，设置虚拟设备并为新域构建存储映像。这种分离在发展过程中非常有用，在零域中调试管理软件比在系统管理程序中调试容易很多。而且，它支持把不同的操作系统加入到零域而不是系统管理程序中，后者所带来的额外复杂度通常是难以想象的。

先前我们提到Xen如何受益于可以使用一个开源操作系统，这个开源系统为半虚拟化提供了测试台。使用Linux的第二个好处是它支持非常多的各不相同的硬件设备。Xen几乎能够支持存在Linux驱动的任何设备，因为它重用了Linux的驱动代码。Xen总是重用Linux驱动以支持多种硬件。然而，在1.0和2.0版本之间，重用的特征发生了明显的改变。

在Xen 1.0中，所有的虚拟机（包括零域）都通过虚拟设备访问硬件，正如先前所述。系统管理程序负责把这些访问复合到真实的硬件，因此，它包含Linux硬件驱动的前端版本和虚拟驱动后端。虽然这简化了虚拟机，但是增加了系统管理程序的复杂度并把支持新驱动的责任交给了Xen开发团队。

图7-6演示了Xen在1.0版本和2.0版本之间设备架构的改变。在1.0版本中，虚拟后端在系统管理程序中实现。所有的域，包括零域，都通过这些设备访问硬件。在2.0版本中，系统管理程序进行了简化，零域通过本地驱动访问硬件。因此，后端驱动移动到了零域。

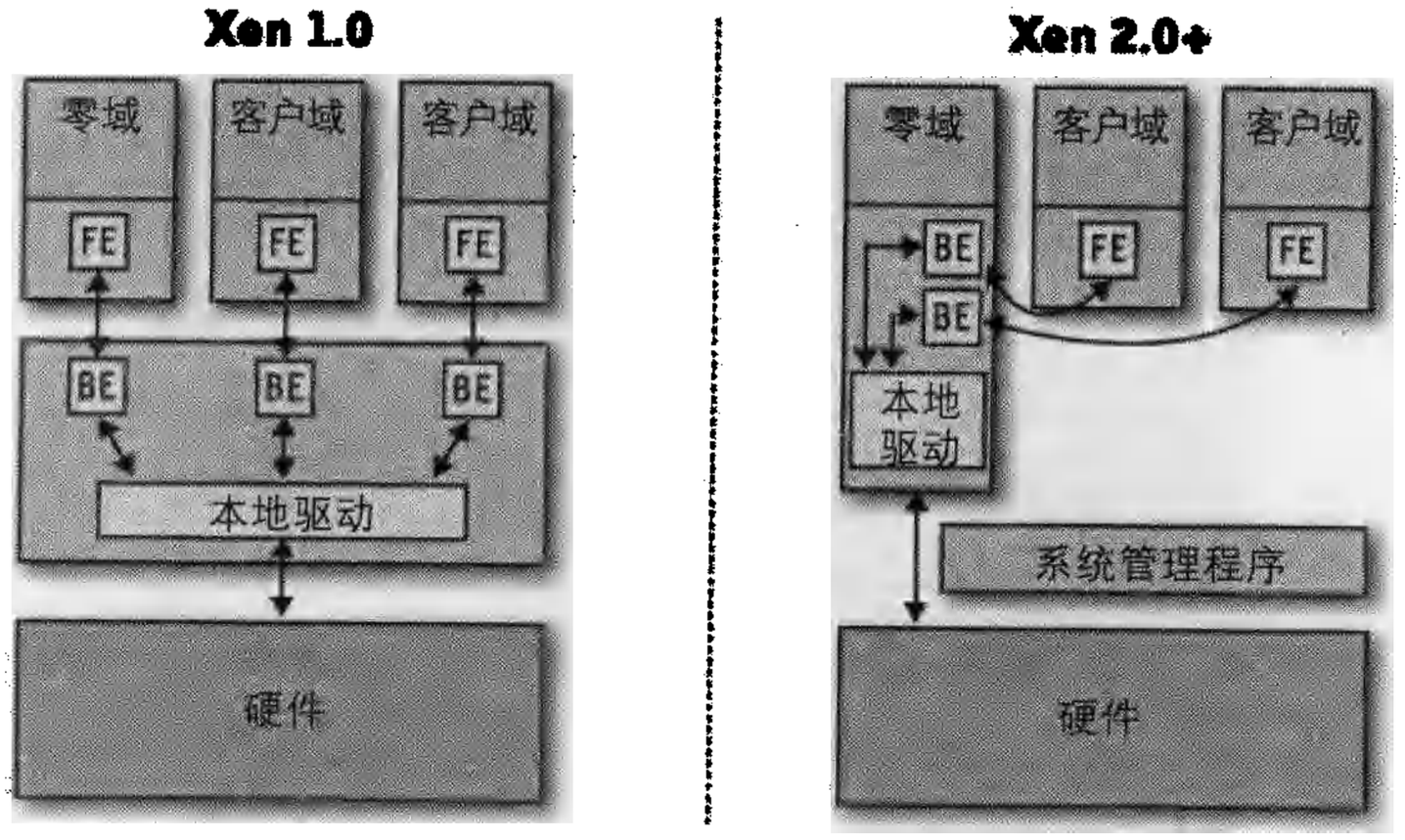


图7-6：Xen设备架构在1.0版本和2.0版本之间的改变

在Xen 2.0的开发中，设备架构全部进行了重新设计：本地设备驱动和虚拟后端移出了系统管理程序，移入了零域。（注5）现在，前端驱动和后端驱动利用设备通道（device channel）进行通信，它使得域之间能够进行有效和安全的通信。通过使用设备通道，Xen的虚拟设备达到了接近本地的性能。它们的性能依赖两个原则：无复制传递和异步通知。

请看图7-7，这个图演示了一个共享设备如何使用。用户向前端驱动提供了一页内存，其中包含了要写的或读入的数据（1）。前端驱动在共享环形缓冲下一个可用的空档中放置一个请求，其中包含了到提供页的引用（2）。前端驱动告诉系统管理程序通知驱动域有一个请求在等待（3）。后端醒过来并把提供页映射入它的地址空间（4）以便硬件可以与它使用的直接内存存取（DMA）进行交互（5）。最后，后端通知前端这个请求已经处理完成（6），前端通知用户应用程序（7）。

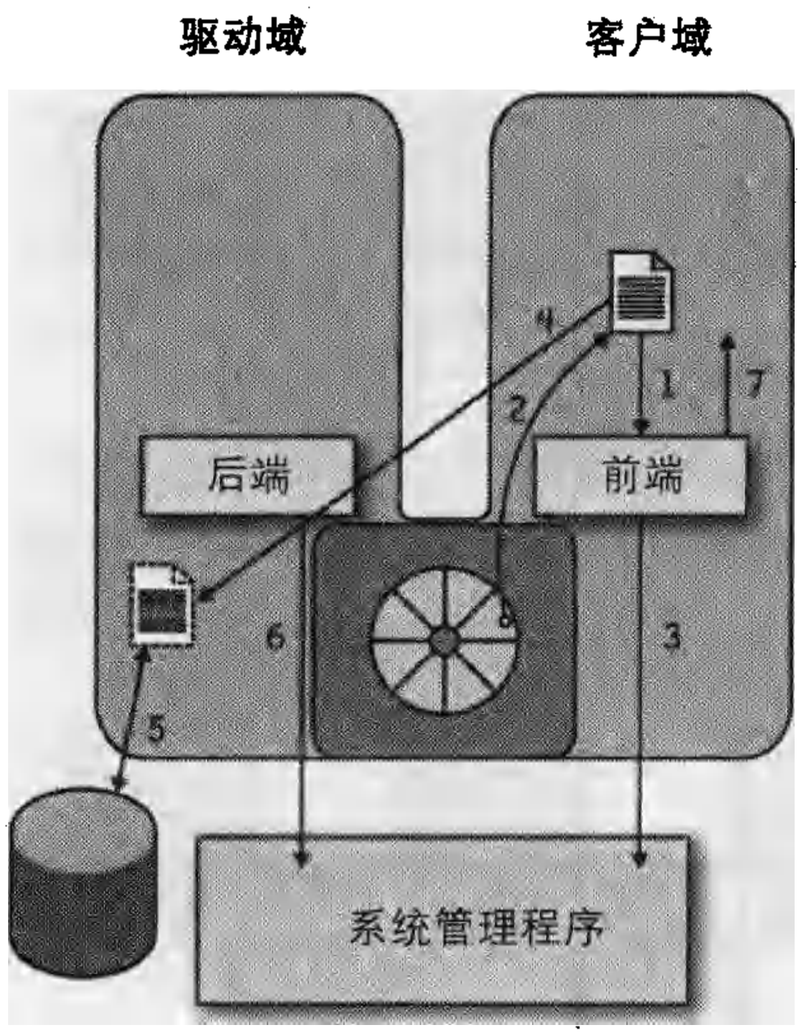


图7-7：一个共享设备的分析

用CPU复制数据的开销是昂贵的，这就是为什么使用像直接内存存取（Direct Memory Access，DMA）这样的技术，以便在设备和内存之间直接传输数据而不需要CPU参与的原因。然而，当数据必须在地址空间之间移动时，Xen必须专门处理以避免复制。Xen支持一种称为授权表（grant table）的共享内存机制，由此每个虚拟机维护一张定义它的哪些页能够由其他虚拟机访问的表。这张表中的索引称为授权索引（grant reference），当给与另一个虚拟机时，它代表一种能力。系统管理程序确保只有预期的接收者可以映

注5：　实际上，这个架构允许任何授权的虚拟机访问硬件，并因此作为驱动域（driver domain）。

射这个授权引用，这反过来维持了内存的隔离性。设备通道本身用于发送授权引用，然后用于映射为了发送或接收数据的缓存。

当作出新的请求或响应的时候，发送者必须通知接收者。这通常使用同步通知——类似于一个函数调用——因此，发送者必须等待，直到它知道通知已经被接收。如图7-8所示，这种操作模式导致糟糕的性能，尤其当只有单个处理器可以使用的时候。Xen实际上使用事件通道（event channel）来发送异步的通知。事件通道执行虚拟中断，但是，只有当目标域下一次调度时虚拟中断才会服务。因此，在目标域调度来执行它们之前，请求者可以产生多个请求，每次都唤醒事件通道。然后，当调度目标域时，它又通过异步的方式处理多个请求并发送响应。

请看图7-8。对于同步通知，前端必须等待后端完成它的工作之后才能进行下一个请求。这意味着等待后端域调度，然后等待前端域调度。比较而言，对于异步通知，前端在被调度的时候可以发送尽可能多的请求，而后端可以发送尽可能多的响应。这增加了吞吐量。

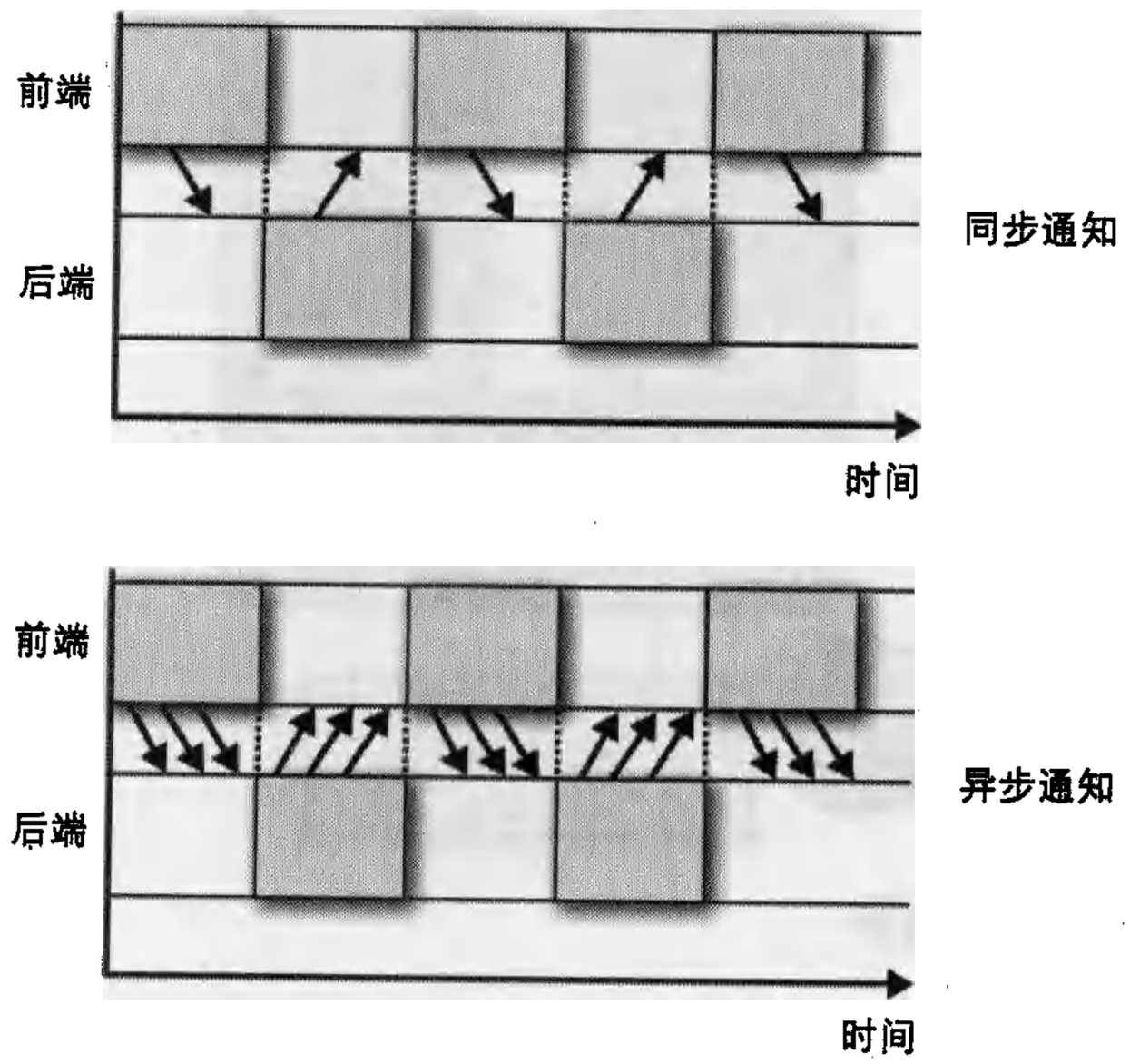


图7-8：异步通知的优点

当然，如果你把过多的功能移到零域，它会变为一个单个故障点。这尤其适用于设备故障，它可以使整个操作系统（由此引起整个虚拟系统）崩溃。因此，Xen考虑到驱动域，零域可以把一个或多个设备的控制委托给它。通过把后端驱动放到驱动域并把一些I/O权限授予这个域就可以简单地实现这些功能。然后，如果一个驱动失败，这个故障会隔离在这个驱动域，它可以重新启动而不损害系统或客户域。

这种模式已经应用于零域的其他部分。Xen的最新版本包括了存根域（stub domain），它为“虚拟硬件”域提供设备支持（在接下来的章节中介绍）。把这些代码移到单独的域中是考虑为了获得更好的性能隔离，提高健壮性并改善初始性能（这有点令人惊讶）。随着开发的延续，越来越多的特性可能会移出零域，特别是当这样做可以改善安全性时。

7.6 改变的硬件，改变的Xen

直到目前，我们的讨论集中在半虚拟化。然而，在Xen 2.0版本和3.0版本之间，Intel和AMD在处理器中引入了有区别但类似的硬件虚拟机支持。在虚拟机中运行未修改的操作系统（包括Microsoft Windows或Linux）已经成为可能。这是否会招致半虚拟化的终结？

首先，我们看一下硬件虚拟机是如何实现的。Intel和AMD都引入了一个新的模式（Intel中称为nonroot模式，AMD中称为guest模式），试图在这个模式中执行一个特权操作，甚至在最高的（虚拟）权限级别，以生成一个通知系统管理程序的异常。因此，不必再扫描代码并替换这些指令（在运行时或通过半虚拟化预先处理）。系统管理程序可以利用影子页表（shadow page table）向虚拟机提供连续内存的假象，接着它可以触发I/O操作以模拟物理设备。

Xen在3.0版本中增加了对硬件虚拟机的支持。开源开发为这次改变提供了很大的帮助。因为Xen是一个开源项目，所以，来自Intel和AMD的开发人员可以贡献支持新处理器的底层代码。此外，由于它的GPL身份，Xen可以合并其他开源项目的代码。例如，新的硬件虚拟机需要一个模拟的BIOS和模拟的硬件设备；实现其中的任何一个都需要巨大的开发工作。幸运的是，Xen可以使用来自Bochs项目的开源BIOS和来自QEMU的模拟设备。

现在你可能想知道有这么多优点的半虚拟化到底怎样了。模拟设备、影子页表、额外的异常等一定会导致糟糕的性能吗？一个草率的硬件虚拟化操作系统通常比半虚拟化操作系统运行得更糟糕，但有两个缓和因素。

首先，处理器厂商正在不断地开发优化虚拟化的新特性。就像存储器管理单元（memory management unit，MMU）使程序员处理虚拟地址而不是物理地址一样，IOMMU对于输入和输出设备来说同样如此。IOMMU可以用来使虚拟机（无论是虚拟化硬件的或半虚拟化硬件的虚拟机）安全、直接地访问一个硬件（参见图7-9）。虚拟机直接访问硬件的常见问题是许多设备可以操作DMA，因此，如果没有IOMMU，它可以读取或重写其他虚拟机的内存。IOMMU可以用来确保：在控制某一虚拟机时，DMA只可以使用属于那台虚拟机的内存。

模拟与虚拟化的对比

最新版本的Xen包含了Bochs和QEMU的代码，而它们都是模拟程序。在模拟和虚拟化之间有什么区别，两者怎样才可以合并呢？

Bochs用软件提供了x86处理器家族以及所支持硬件的开源实现。QEMU模拟了好几个架构，包括x86。两者都可以用于运行未修改的x86操作系统和应用程序。而且，因为它们都包括了硬件的完整实现（包括CPU），所以，它们可以运行在使用不兼容指令集的硬件上。

虚拟系统和模拟系统的区别在于两者的指令执行方式不同。在虚拟系统中，应用程序和大部分操作系统都直接在处理器上运行。而在模拟系统中，模拟程序必须模拟或转换每个指令以执行它。因此，对于相同的平台，模拟程序比虚拟机监控器引入更多的开销。（注6）

然而，即使它们使用了部分的Bochs和QEMU，Xen的硬件虚拟机仍然是虚拟化的，而不是模拟的。Bochs的代码提供了支持启动进程的BIOS，而QEMU提供了许多常规设备的模拟驱动程序。然而，只有在启动和试图进行I/O操作时才调用这些代码。其他大部分指令直接在CPU上运行。

图7-9演示了一个来自一个虚拟机（DomIO）的利用IOMMU的简化的DMA请求。当硬件驱动程序和设备通信时使用伪物理（特定虚拟机的）地址（1）。这个设备使用这些地址进行DMA请求（2），IOMMU（利用由系统管理程序配置的I/O页表）把这些地址转变为使用物理地址（3）。IOMMU也会终止虚拟机对任何不属于它的内存地址的访问。

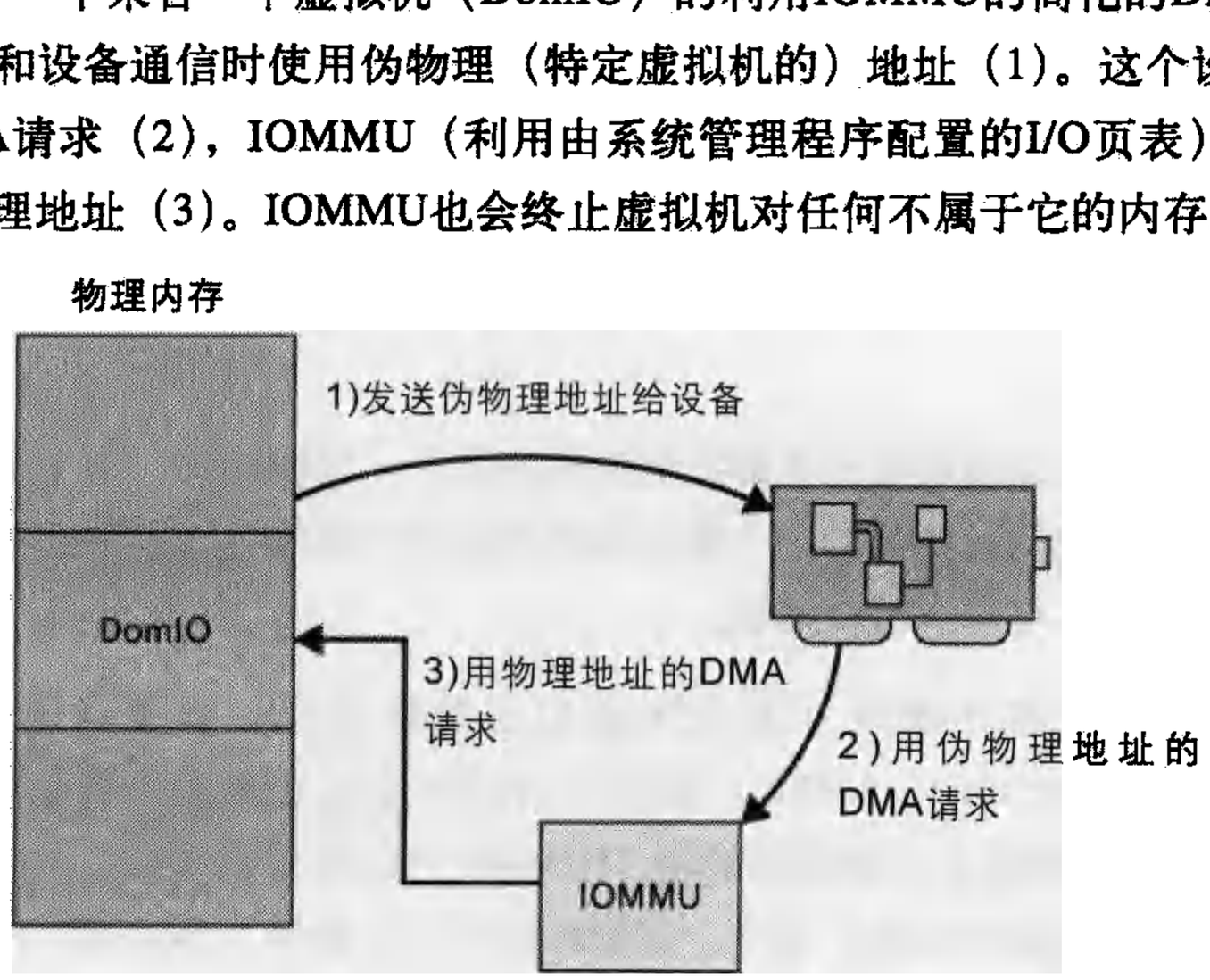


图7-9：利用IOMMU直接访问设备

注6：KQEMU是容许用户模式的代码（及一些内核模式的代码）直接在CPU上运行的Linux内核模块。在宿主和目标平台相同的地方，这个模块可以提供巨大的加速。结果是模拟和虚拟的混合。

增强内存管理硬件也可以避免使用影子页表。（注7）AMD和Intel都拥有在伪物理地址和物理地址之间进行转换的技术，它们分别称为快速虚拟化索引（Rapid Virtualization Indexing）和加强页表（Enhanced Page Table）。因此无需系统管理程序创建影子页表，因为整个转换都由硬件进行处理。

当然，一个廉价得多的解决方案是吸取半虚拟化的经验并把它们应用到未修改的客户操作系统。虽然不太可能改变操作系统的核心部分，但是我们可以增加设备驱动程序。此外，Xen可以修改操作系统运行的虚拟硬件。最后，模拟硬件提供一个Xen平台设备（Xen platform device），它看起来像是未修改客户操作系统的一个PCI设备并提供对虚拟平台的访问。然后就可以为未修改的操作系统编写前端设备，它会像半虚拟化操作系统中的前端一样运行。这样我们可以在硬件虚拟机中获得与半虚拟化情形相当的I/O性能。

当这一章前面介绍半虚拟化的时候，我们说过，使一个常用操作系统像半虚拟化客户操作系统一样运行的最好方式是我们自己做或使操作系统开发人员相信他们应该这样做。作为半虚拟化成功的一个证明，Microsoft在Windows Server 2008中已经包含了一些启蒙技术（enlightenment），当在虚拟机中运行时会提高内存管理的性能。这些启蒙技术相当于半虚拟化操作，因为它们依赖超级调用来通知当前运转的系统管理程序。

7.7 经验教训

回顾一下，我们可以从Xen中获得两个主要的经验：半虚拟化的重要性和开源开发的优点。

7.7.1 半虚拟化

首先是半虚拟化的成功。一句名言提醒我们：

> 计算机科学中的任何问题都可以用另外的间接层解决，但是这通常会引发另一个问题。
>
> ——David Wheeler

虚拟化仅仅是一种间接的形式，即使现代的计算机从硬件上支持虚拟化，盲目地依赖这种支持会导致糟糕的性能。当你盲目地利用任何类型的虚拟化时都会出现同样的问题。

例如，虚拟内存使用一块硬盘来产生有很多内存可用的假象。然而，假如你编写了一个试图使用所有内存（就像它是真正的物理内存一样）的程序，那性能会非常糟糕。在这种情况下，你可以想象把那个程序“半虚拟化”以使它意识到这个物理限制，结合虚拟内存系统来改变用来使它高效运行的算法和数据结构。

在操作系统的环境中，Xen已经表明半虚拟化（无论是它增加虚拟驱动程序，还是大规

注7：需要提到的是，Xen的影子页表实现方式非常优秀，其性能很有竞争力，但是，当和半虚拟页表进行比较时仍然显得开销略大。

模地改变操作系统，还是在选定区域中有意识地提升性能）是在虚拟环境中运行时提升性能的一项重要技术。

7.7.2 开源开发

在Xen发展过程中采取的最大胆的决策也许是当其他系统管理程序还只作为私有软件时把Xen作为开源软件。

这个决策绝对使Xen受益，因为这样，Xen可以利用非常多的软件：从Linux内核和QEMU机器模拟器，到启动时绘制Xen标识的小程序。（注8）没有这些软件，Xen项目将需要非常大工作量的重复实现。通过包括这些来自其他项目的软件，当它们更新时，Xen也会受益，而其他项目也受益于Xen开发人员提交的补丁。

Xen最初是剑桥大学一位研究生的兼职项目，现在已经成长为包括来自全球的超过100位贡献者。一些大的贡献来自Intel和AMD，他们提供了许多代码来支持硬件虚拟机。这使得Xen能够成为领先的系统管理程序之一来支持这些处理器扩展。

除此之外，因为可以自由获得Xen，一些项目已经采用它。主流的Linux版本，例如Debian、Red Hat、SUSE和Ubuntu，都包含了Xen包并给这个项目反馈代码，还有用于使用Xen的有用的工具。一些贡献者努力把Xen移植到其他架构，甚至移植到其他操作系统上，使操作系统直接运行在这个系统管理程序上。Xen已经用于运行半虚拟化的OpenSolaris、FreeBSD和NetBSD等。Xen现在也涉及Itanium架构，作用是把它移植到ARM处理器。后者非常令人激动，因为它使得Xen能够运行“非传统”的设备，例如移动电话。

当我们着眼于未来时，一些最有趣的Xen用途在于研究团体。Xen出现在2003年的操作系统原理座谈会（Symposium on Operating Systems Principles，SOSP）上并成为一系列研究（在它最初的研究组织内外都有）的基础。关于Xen的最早的论文之一来自Clarkson大学，一组研究人员在其中重复了SOSP论文中的结果。作者们强调开源软件改善了计算机科学，因为它使重复研究成为可能，并且反过来巩固任何关于性能或其他特性的主张。最近更多的研究工作直接导致Xen中的新特性变得令人感兴趣。一个特殊的例子是动态迁移（live migration），它使得一台虚拟机用几乎可以忽略的时间在物理计算机之间迁移。这在2005年的论文中进行了详细描述，并增加到了Xen的2.0版本中。

7.8 延伸阅读

本章只能够简单地介绍一下Xen项目，相关的研究论文是详细资料的最好来源。

注8 Figlet：*http://www.figlet.org*.

下面两篇论文分别描述了Xen 1.0和2.0的架构：

Barham, Paul, et al. "Xen and the art of virtualization," *Proceedings of the 19th ACM Symposium on Operating System Principles*, October, 2003.

Fraser, Keir, et al. "Safe hardware access with the Xen virtual machine monitor," *Proceedings of the 1st OASIS Workshop*, October, 2004.

下面的论文讲述了一些已经开发用来帮助虚拟化的新的芯片集及处理器技术：

Ben-Yehuda, Muli, et al. "Using IOMMUs for virtualization in Linux and Xen," *Proceedings of the 2006 Ottawa Linux Symposium*, July, 2006.

Dong, Yaozu, et al. "Extending Xen with Intel virtualization technology," *Intel ® Technology Journal*, August, 2006.

最后，Xen正在积极地发展并不断地演变。保持与新开发平行的最好方式是下载源代码并加入邮件列表。它们都可以在http://www.xen.org/找到。

原则与特性	结构
√ 功能多样性	√ 模块
√ 概念完整性	√ 依赖关系
√ 修改独立性	√ 进程
√ 自动传播	√ 数据访问
可构建性	
√ 增长适应性	
√ 熵增抵抗力	

第 8 章

Guardian：一个容错操作系统环境

Greg Lehey

架构不是什么新东西。真实的建筑架构已经存在了几千年，一些最漂亮的建筑架构也已经存在了几千年。当然，计算机还没有存在这么久，但是，过去也有许多漂亮架构的例子。正如建筑一样，风格并不总是保持不变，本章将讲述这样一个架构并探讨它为什么几乎没有影响力。

Guardian是Tandem的容错“NonStop”系列计算机的操作系统。它设计成与硬件并行以便用最小的开销提供容错性。

本章将讲述最初的Tandem机器，它设计于1974年—1976年，发行于1976年—1982年。它最初称为“Tandem/16”，但是在“NonStop II”成功之后，它重命名为“NonStop I”。Tandem经常把术语“T/16”使用在系统和随后的架构上。

我从1977年到1991年一直专门从事与Tandem的硬件相关的工作。与Tandem机器打交道既令人愉快又不寻常。在本章中，我很乐意地回顾程序员对这种机器的一些感觉。T/16是一种容错的机器，但是，这不是它唯一的特性，在本章的讨论中，我会提到许多与容错没有直接关系的方面——实际上，会降低容错性！所以，请准备回到过去，回到1980年，从Tandem的一个市场口号开始。

8.1 Tandem/16，将来所有的计算机都会像这样构建

Tandem把计算机描述成有多个处理器的单独的计算机，但是，从21世纪的观点来看，它们更像是与单个计算机一样运行的计算机网络。尤其是每个处理器都几乎完全不依赖其他处理器而独立地工作，而系统可以从任何单个组件（包括处理器）的故障中恢复。和传统联网的处理器最大的区别是整个系统都从单个内核映像开始运行。

8.2 硬件

Tandem的硬件设计成不可能存在“单个故障点”（single point of failure）：系统、硬件或软件的任何一个组件都可以出故障而不引起整个系统出故障。它设计成优雅降级（graceful degradation）。在大多数情况下，尽管系统有多个故障，但总体上还能继续运行，虽然这很大程度上依赖个别故障的种类。

这种架构的首要含义是每种组件都必须至少有两个，以防一个发生故障。这意味着系统至少需要两个CPU。

但是，CPU之间应该怎样连接呢？一贯的传统方法是CPU通过共享内存进行通信。在Tandem中，我们称之为紧耦合多处理器（tightly coupled multiprocessor）。但是，如果处理器共享内存，那么，内存就成了单个故障点。

从理论上来说，双重内存是可能的（实际上，随后的Tandem架构做到了这一点），但是，这样做非常昂贵，而且，这样做造成了重大的同步问题。Tandem改为在硬件级别选择一对高速并行总线——处理器间总线（interprocessor bus，IPB），有时候这也称为Dynabus，它在分开的CPU之间传输数据。有时候这种架构称为松耦合多处理器（loosely coupled multiprocessor）。

当然，一台计算机不仅仅有CPU。I/O系统和数据存储也非常重要。在这里，基本的方式也是进行硬件备份，后面将深入讲解。

最终的架构看起来有点像图8-1，这就是所谓的Mackie图，是以Tandem的副总裁Dave Mackie命名的。

这很容易导致一个系统的开销加倍（甚至更大），照“热备份”系统现在的样子，一个组件的存在只是为了等待它的伙伴出故障。Tandem对比较昂贵的组件（例如CPU）采用不同的方式。在T/16中，每个CPU都是活跃的，而不是为操作系统进程提供热备份功能。

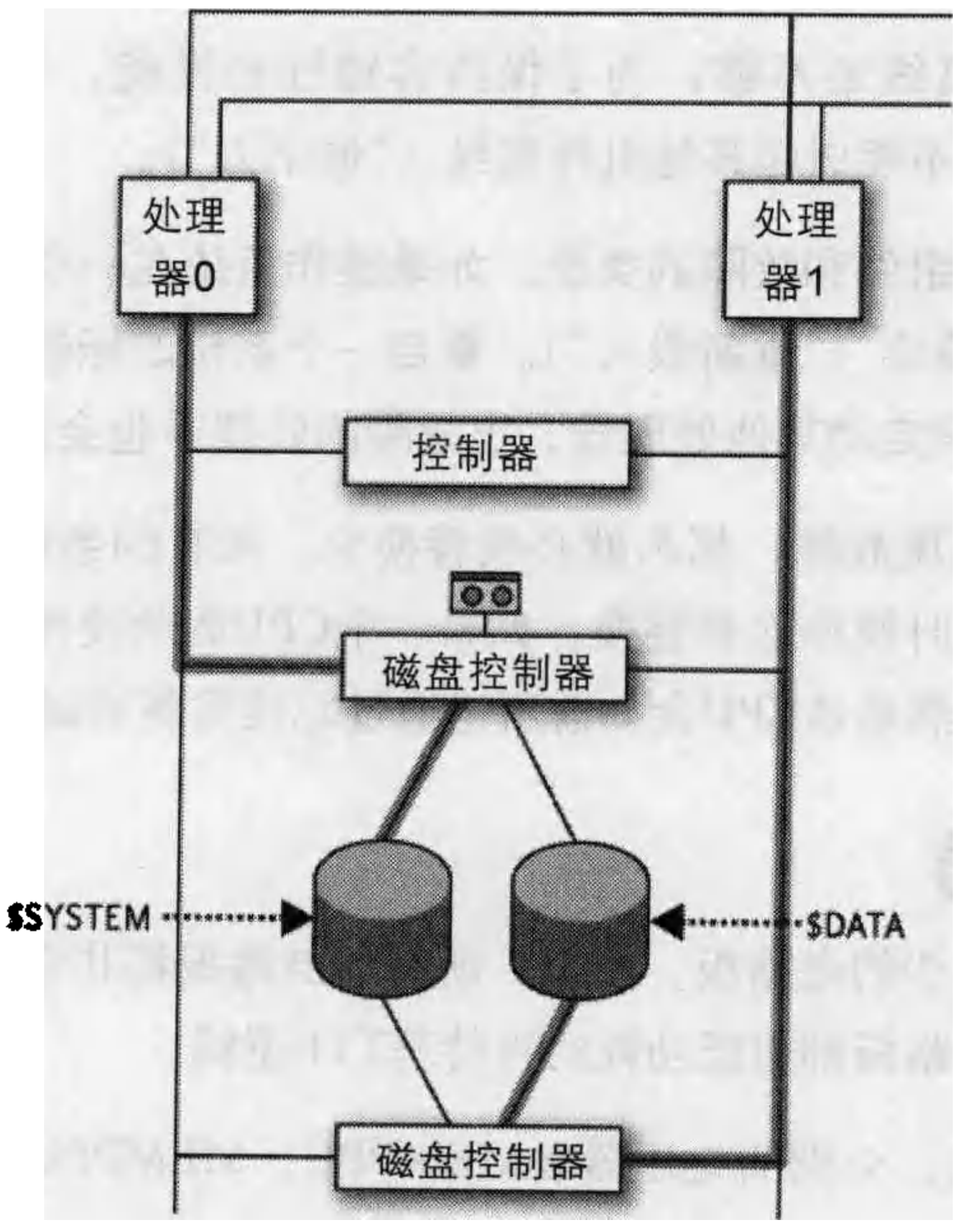


图8-1：Mackie图

8.2.1 诊断

当一个组件出故障的时候操作系统必须能够察觉。在许多情况下，这勿庸置疑：如果系统出现灾难性的故障，它会完全停止响应。但是，在另一些情况下，一个出故障的组件会继续运行，只是会产生不正确的结果。

Tandem对这个问题的解决方案既不优雅也不高效。这个软件设计成多疑的，最初的建议是：如果有些东西发生了故障，那么操作系统就停止这个CPU——由另一个CPU来接管工作。如果磁盘控制器返回一个无效的状态，那么它会被中止——另一个会不间断地继续处理。但是，如果故障很微小，它就可以继续运行而不被察觉，在极少的情况下会导致数据异常。

这样判断CPU出故障当然还不够；其他CPU必须发觉它已经出故障了。这里的解决方案是Watchdog。每个CPU每隔1.2秒在双方的总线上广播一个信息，即所谓的“我还活着”的信息。如果一个CPU连续两次没有收到另一个CPU的“我还活着”的信息，它会假定那个CPU出故障了。如果这两个CPU共享了资源（进程或I/O），那么，探测到这个故障的CPU会接管这些资源。

8.2.2 修复

让一个有缺陷的组件离线还不够；为了保持容错性和性能，它必须尽可能快地恢复（“起来”），而且，当然不能引起其他组件离线（“倒下去”）。

如何实现这一点取决于组件和故障的类型。如果操作系统在一个CPU中崩溃了（可能是故意的），它可以在线重启（“重新载入”）。重启一个系统的标准方式是先从磁盘启动一个处理器，然后通过IPB启动其他处理器。出故障的处理器也会通过IPB重新启动。

另一方面，如果硬件出现故障，那么就必须替换它。所有的系统组件都是可热插拔的：它们可以在系统运行的时候移除并替换。如果一个CPU因为硬件问题而出故障，那就会换一块合适的电路板，然后该CPU会如前所述通过总线重新启动。

8.3 物理布局

系统设计成包含尽可能少的电路板，所以，所有的电路板都非常大，大概是边长为50厘米的正方形。所有的电路板都用低功耗的肖特基TTL逻辑。

CPU由两块电路板组成，分别为处理器和MEMPPU。MEMPPU包含了与内存（包括虚拟内存逻辑）的接口和与I/O总线的接口。T/16最多可以配置512KW（1MB）的半导体内存或256KW的核心内存。内存板有三个尺寸：32kW核心内存、96KW半导体内存及192KW半导体内存。这意味着用完全组装的板没有办法得到精确的1MB半导体内存。核心内存有奇偶校验码保护，而半导体内存有ECC保护，它可以纠正一位的错误和发现两位的错误。

处理器机柜大概有6英尺高，可以容纳4个带半导体内存的CPU或4个带核心内存的CPU。这些处理器放置在机柜的顶部，I/O控制器放在下面相邻的第二个机架上。在那下面是风扇，而在机柜底部有电池，在电源出故障时可以保持内存中的内容。

大多数结构还有另一个用来放置磁带机的机柜。磁带机是独立式的14英寸的部件。还有一个系统控制台，一个DEC LA-36的打印终端。

8.4 处理器架构

CPU采用的是一个与Hewlett-Packard 3000很相似的定制的TTL设计。它拥有一个2KB页面大小的虚拟内存、一个基于栈的指令集和宽度固定为16位的指令。原始的处理器速度大约为每个处理器0.8MIPS，一个配置满16个处理器的系统的速度达到13MIPS。

8.4.1 内存寻址

T/16是16位的机器，它的地址空间的宽度限制为16位。即使在20世纪70年代后期，这也开始成为一个问题，Tandem通过在任何时候都提供4个地址空间来解决这个问题：

用户代码

地址空间包含了可执行的代码。它是只读的，而且可以在所有使用它的进程之间共享。由于这个架构（为每个CPU分隔内存），这个代码只能分配给特定的某个CPU。

用户数据

用户进程的数据空间。

系统代码

内核的代码。

系统数据

内核的数据空间。

有一个例外：在任何时候只有一个数据空间和一个代码空间可以访问。它们在环境寄存器（Environment Register）中指定，环境寄存器包含了许多描述CPU当前状态的标记，如图8-2所示。

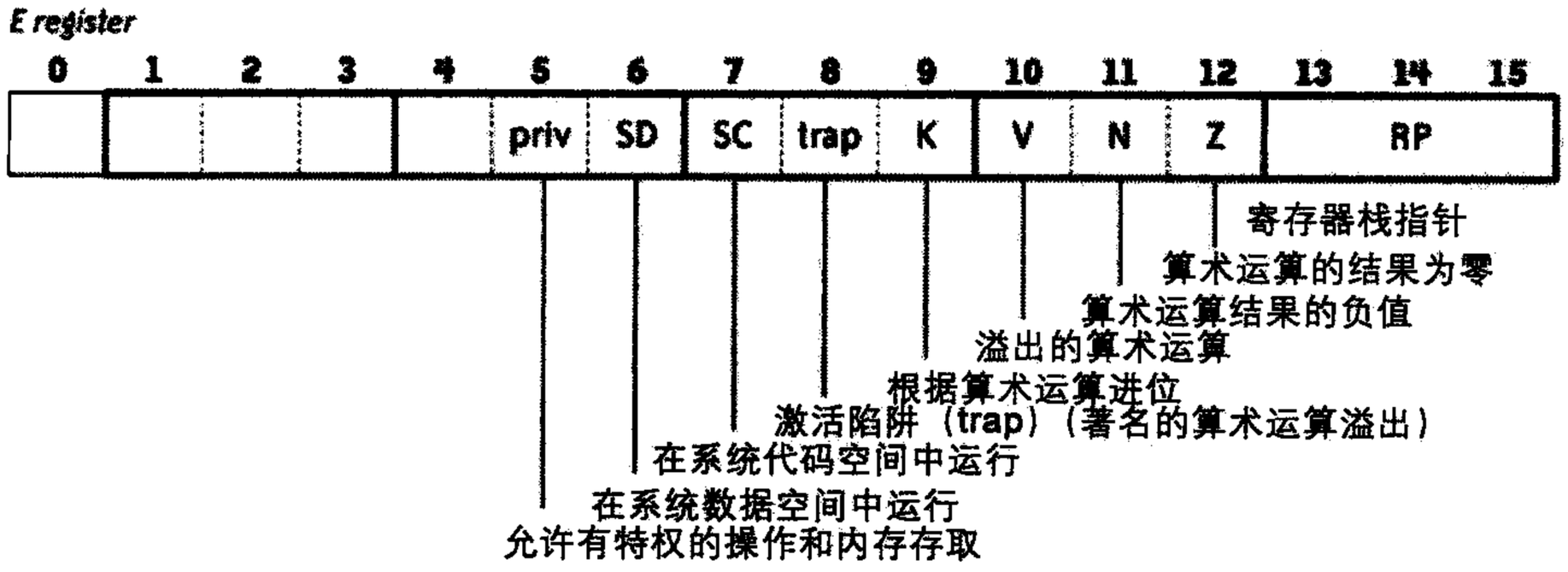


图8-2：环境寄存器

SD位确定数据空间，SC位确定代码空间。SG相关的寻址模式是这个规则（总是在系统数据中寻址）的一个例外。

为了保持可靠性和数据完整性，环境寄存器中的陷阱（trap）位和其他的位一起能够捕获运算溢出。这样就有了不设置条件代码的运算指令的“逻辑”等价物。

CPU拥有由两个寄存器（S寄存器或栈指针，以及L寄存器）寻址的一个硬件栈。L寄存器指向当前的栈帧（stack frame）。L寄存器是一个相对较新的概念，它指向当前帧的底

部。与S寄存器不同，在一个程序的执行过程中它不会改变。（注1）出于寻址的考虑，这个栈限制为当前数据空间最初的32KB，而且与其他的机器不同，它会向上增加。（注2）

除硬件栈之外，还有一个具有8个16位字的寄存器栈。这些寄存器的号码从R0编到R7，但是，指令集把它们用作一个循环栈，其中栈的顶部由环境寄存器的RP位定义。在下面这个例子中，RP设置为3，这就意味着R3是栈的顶部，视为A寄存器（如图8-3所示）。

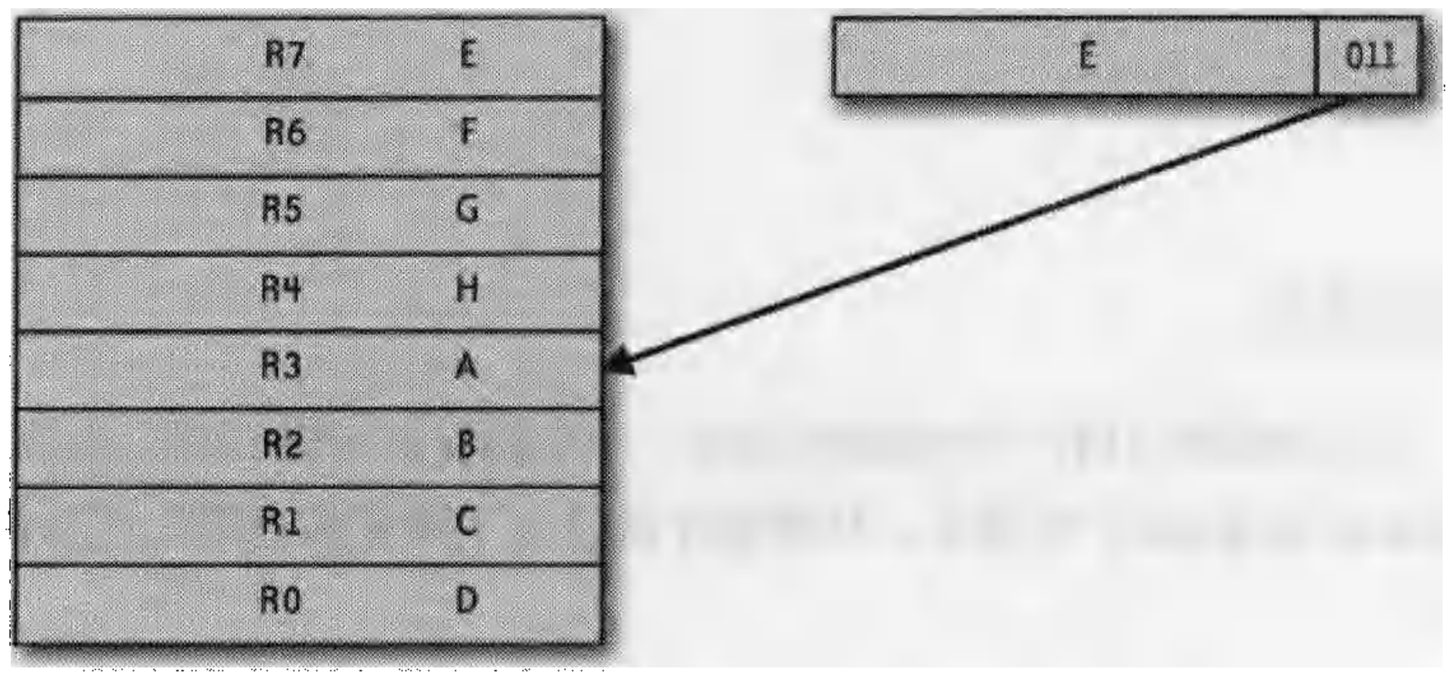


图8-3：寄存器栈

假设最初这个寄存器栈是“空”的，一个典型的指令序列如下所示：

```
LOAD    var^a          -- 把变量a压入栈（R0）
LOAD    var^b          -- 把变量b压入栈（R1）
ADD                    -- 把A和B （R1和R0）加起来，把结果保存在R0 （A）中
STOR    var^c          -- 把A的值赋给变量c
```

指令的宽度都是16位，这没有为一个地址字段剩下许多空间：只有9位。为了解决这个问题，Tandem基于一系列寄存器的偏移量进行寻址（如图8-4所示）。

只有下列的内存区域可以直接定位（换句话说，不用间接寻址）：

- 当前数据空间的前256个字，称为“G”（global）模式。这些经常用于间接指针。
- L寄存器正偏移的前128个字，称为L+。这些是当前程序调用的全局变量，在C中，这些称为自变量。
- 系统数据的前64个字（“SG+”模式）。系统调用运行在用户数据空间，所以，CPU需要一些方法让授权的过程访问系统数据。对于没有授权的过程来说，它们是不能访问的，甚至是只读的。

注1：这与21世纪大多数的处理器使用的通用指针寄存器相同。

注2：在20世纪70年代，栈是相当新的概念。如同它的前身HP 3000，Tandem对栈的支持明显超越了其他系统（例如，DEC的PDP-11，它是那时候最有影响的基于栈的另一台机器）对栈的支持。

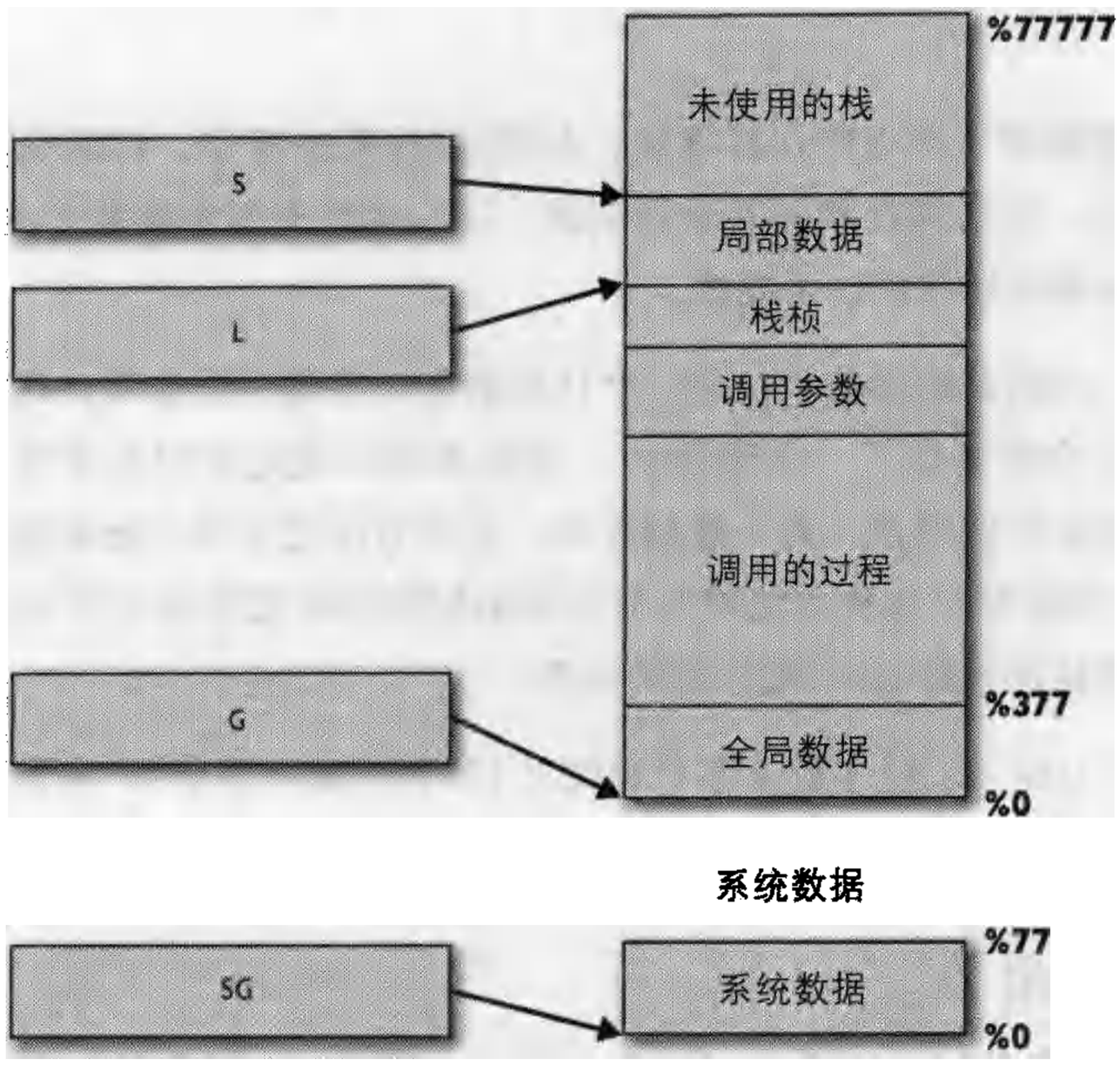


图8-4：内存寻址

- L寄存器当前值下方的前32个字。这包括调用者栈帧（3个字）和传给过程的最多29个参数字。
- 在这个栈顶部下方的前32个字（S-寻址）。它们用于子过程（在其他过程中定义的过程），子过程调用不会剩下一个栈帧。因此，这个地址既处理子过程的本地变量也处理子过程的参数。

这些地址模式都编码在指令地址字段的最初几位中，如图8-5所示。

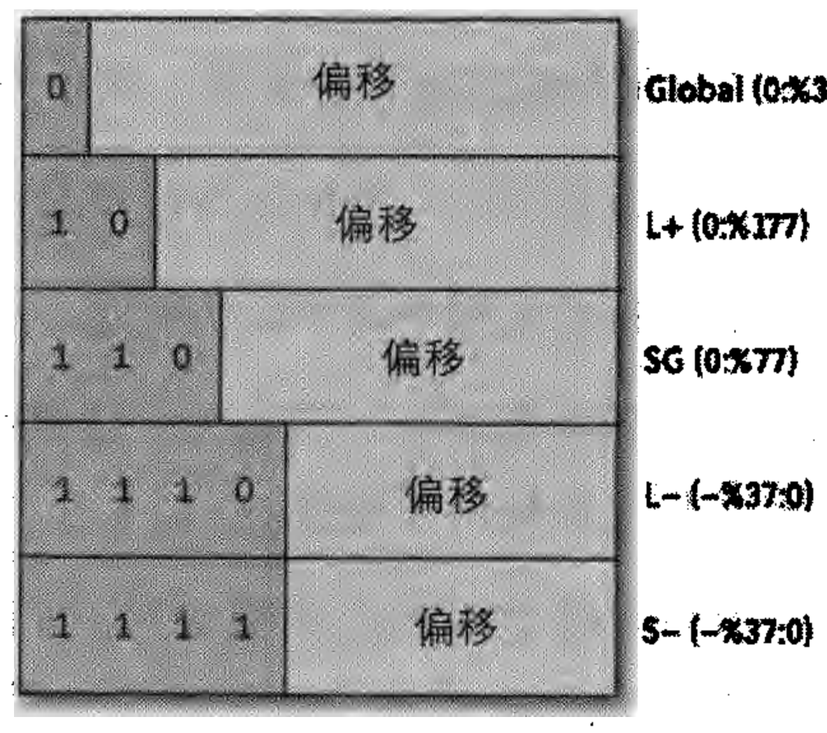


图8-5：地址形式

%符号代表八进制数，如同%377等于十进制的255，或十六进制的7F。Tandem没有使用十六进制。

这个指令形式也规定了单级的间接寻址：如果I位设置进指令，找回的数据字会作为最终操作数的地址，它必须在相同的地址空间。这个地址空间和数据字的宽度都是16位，所以，不存在多级间接寻址的可能性。

这种实现的一个问题是数据的单位是一个16位的字，不是一个字节。这个指令集也通过一个不同的寻址方法提供了“字节指令”：地址的低位指定字中的字节，而地址的其余部分是这个字地址的低15位。对于数据存取，这种方法把字节寻址限制到数据空间的前32KB；对于代码存取，这种方法把字节存取限制到当前指令地址空间的一半长。这导致一个过程不能越过代码空间中的32KB边界。

还有两个指令：LWP（从程序载入字）和LBP（从程序载入字节），它们可以访问当前代码空间中的数据。

8.4.2 过程调用

Tandem的编程模式受Algol和Pascal的很大影响，所以，它为返回一个值的函数提供了保留字function，并为不返回值的函数提供了保留字procedure。有两个指令用于调用一个过程：PCAL用于当前代码空间中的过程，SCAL用于系统代码空间中的过程。SCAL实现了其他架构中的系统调用的功能。

所有调用都是间接通过一个过程入口点表（Procedure Entry Point Table，PEP），它占据了每个代码空间的前512个字。PCAL或SCAL指令的最后9位是在这个表中的一个索引。

这种方式有危险也有好处：内核和用户代码使用完全相同的函数调用方法，这简化了编码规则并容许代码在内核和用户空间之间相互移动。另一方面，至少从理论上来说，SCAL指令使任何用户程序都能够调用任何内核函数。

系统基于环境寄存器中的priv位来保护敏感过程的使用。它分为三种过程：

- 非特权过程：任何过程都可以访问它，无论这些过程是否有权限。
- 特权过程：只有从其他有特权的过程中才可以访问它。
- 可调用过程：可以从任何过程中调用它，但是，一旦调用后就会设置priv位。它们在特权过程和非特权过程之间提供了链接。

特权过程、非特权过程和可调用过程之间的区别取决于它们在PEP中的位置。因此，在系统PEP（有时候称为SEP）中也可能存在非特权库的过程。这张表的结构如图8-6所示。

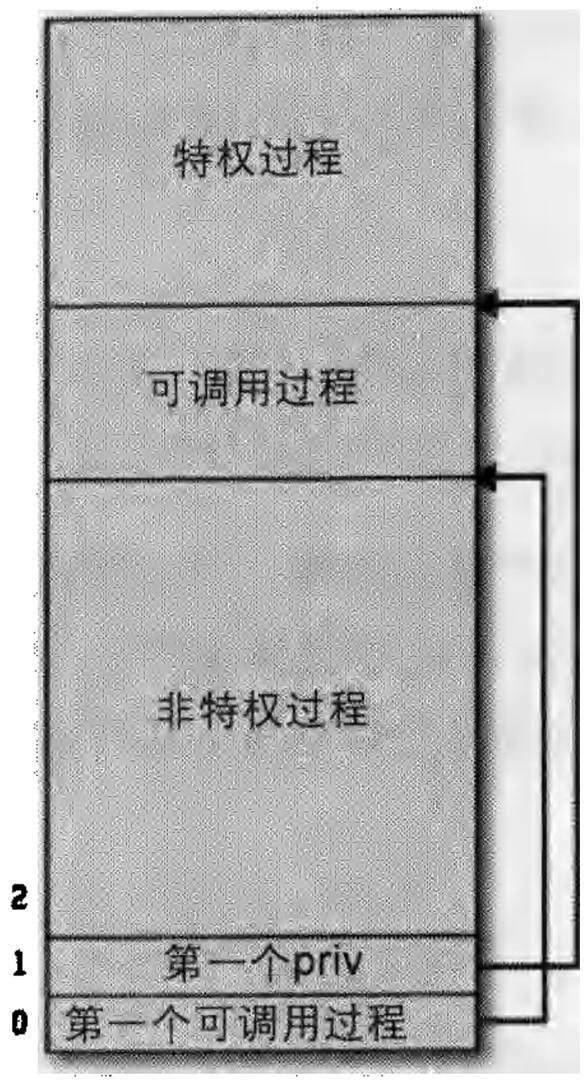


图8-6：过程入口点表

8.4.3 PCAL和SCAL指令的行为

PCAL指令完成下列行为：

- 如果环境寄存器中的priv位没有设置（意味着这个调用过程是非特权的），检查“第一个priv”的值（代码空间的字为1）。如果指令中的偏移量更大或相等，这个过程正在设法调用一个特权过程。生成一个保护阀。
- 如果环境寄存器中的priv位没有设置，检查“第一个可调用过程”的值（代码空间的字为0）。如果指令中的偏移量更大或相等，设置环境寄存器中的priv位。
- 把P寄存器（程序计数器）中当前的值压入栈。
- 把环境寄存器中原来的值压入栈。
- 把L寄存器当前的值压入栈。
- 把S寄存器（栈指针）复制到L寄存器。
- 把环境寄存器的RP字段设置为7（空）。
- 把这个指令定位的PEP字的内容载入P寄存器。

SCAL指令的工作方式完全相同，除此之外，它还设置环境寄存器中的SC位，因而确保在内核空间中继续执行。数据空间不会改变。

PCAL和SCAL指令非常相似，程序员通常不必区分它们。由系统在执行的时候区分它们。因此，库过程可以在用户代码和系统代码之间相互移动而不必重新编译。

8.5 处理器间总线

CPU之前的所有通信都通过处理器间总线（IPB）进行。实际上有两条总线，称为X和Y（如图8-1所示），以防一条出故障。与其他的组件不同，当两条总线正常时会并行使用。

数据以长度固定为16字的包的形式通过总线。总线的速度可以快到使两个CPU的内存满负荷，所以，客户端CPU利用SEND指令在调度程序中同步地执行它。目的（服务器）CPU在启动一个传输的时候为它预留缓存空间。在传输完成时，目的CPU收到一个总线接收中断信号并处理这个包。

8.6 输入/输出

每个处理器都有一个最多有32个控制器的单I/O总线。所有的控制器都采用双通道并连接到两个不同的CPU。任何时候，只有一个CPU可以访问任意特定的控制器。在CPU和控制器之间的关系称为拥有关系：控制的CPU“拥有”控制器。备份的通道不会使用，直到主通道出故障或系统操作员手动切换到它（所谓的主切换）。

磁盘问题是一个特别敏感的问题，因为许多组件都可能出故障。出故障的可能是磁盘本身、磁盘的物理连接（线）、磁盘控制器、I/O总线或磁盘连接的CPU。结果，除双通道的控制器之外，每个磁盘都有物理的备份（至少在理论上），而且它也是双通道并连接到两个不同的控制器，这两个控制器都连接到相同的两个CPU。在任何时候只有一个CPU可能访问其中某个控制器，这一约束保留了下来，但是，两个CPU分别拥有这两个控制器中的一个是有可能的。从性能的观点来看这是不适当的。

图8-1展示了一个典型的配置：正如灰色高亮的线路所示，系统磁盘$SYSTEM的I/O过程通过CPU 0和第一个磁盘控制器访问$SYSTEM，而另一个连接到相同两个控制器的磁盘$DATA的I/O过程通过CPU 1和第二个磁盘控制器访问$DATA。CPU 0“拥有”第一个控制器，而CPU 1“拥有”第二个控制器。如果CPU 0出故障，CPU 1中$SYSTEM的备份的I/O过程会接管并夺走这个控制器的拥有权，然后继续处理。如果第二个磁盘控制器出故障，$DATA的I/O过程不能使用第一个磁盘控制器，因为它属于CPU 0，所以，这个I/O过程首先会进行主切换，在切换之后，主I/O过程会在CPU 0中运行。它然后通过和$SYSTEM相同的路径访问$DATA。

总之，理论上来说就是如此。实际上，磁盘和驱动器是很贵的，许多人以降级模式运行他们的一些磁盘，不准备双重的驱动器硬件。这也会如你预料的那样运行，但是，当然也不存在任何容错能力：事实上，其中的一些磁盘已经出故障了。

8.7 进程结构

Guardian是一个微核系统：除了低级别的中断处理器（一个单独的过程，IOINTERRUPT）和一些非常低级别的代码之外，所有的系统服务都由运行在系统代码和数据空间中的系统进程处理。

比较重要的进程有：

- 系统监控程序，每个CPU中的PID 0，负责启动和停止其他进程，还负责其他像返回状态信息、生成硬件错误信息和维护时间这样的各种各样的任务。
- 内存管理器，每个CPU中的PID 1，负责虚拟内存系统的I/O。
- I/O进程，负责控制I/O设备。系统中无论什么地方对I/O设备的访问都通过它专用的I/O进程。I/O控制器连接到两个CPU，所以，每个设备都控制于运行在这些CPU中的一对I/O进程：完成工作的主进程和一个备份进程，备份进程跟踪主进程状态并等待主进程出故障或等待主进程主动把控制权移交给它（“主切换”）。

选择主CPU中的主要问题是CPU负载，CPU负载必须手动平衡。例如，假设你在CPU 2和CPU 3之间连接着6个设备，你可能会把其中的3个主进程放在CPU 2中，另外3个主进程放在CPU 3中。

进程对

进程对的概念不限于I/O进程。它是这种容错方式的基础之一。要理解它们的工作方式，我们必须理解消息传入系统的方式。

8.8 消息系统

正如我们看到的那样，T/16和其他传统计算机之间最大的区别在于没有一个组件是必需的。系统的任何部分都可以出故障而不会引起系统崩溃。这使得它比传统的共享存储器多处理器机器更像一个网络。

这对于操作系统设计来说具有深远的意义。一个磁盘可以任意连接到2～16个CPU。其他的CPU如何访问它？现代的网络使用NFS或CIFS等运行在网络协议之上的文件系统来处理这种特殊的情况。但是，在T/16上这不是特殊情况，这是标准。

不仅文件系统需要这种通信方式：所有进程间的通信也都需要它。

Tandem针对这个问题的解决方法是在操作系统中以非常低级别运行消息系统。用户程序不能直接访问它。

消息系统在进程之间传输数据，在许多方面它类似于后面的TCP或UDP。消息的发起者称为请求者，而目标则称为服务器。(注3)

进程之间所有的通信都通过消息系统，即使在相同的CPU中。下列的数据结构实现了通信功能：

- 每个消息都联合了两个链接控制块（Link Control Block，LCB），一个用于请求者，一个用于服务器。这些小的数据对象设计成适合单个IPB包。如果需要能比LCB容纳更多数据，那就会附上一个单独的缓冲。
- 为了发起一次传输，请求者调用过程link。这个程序把消息发给服务器进程并让这个LCB在它的消息队列中排队。此时，服务器进程还没有参与进来，但发送器会用一个链接请求（LREQ，link request）事件唤醒这个过程。

在请求者端，对link的调用会立即返回信息以识别这个请求；请求者不必等待服务器处理这个请求。

- 经过若干时间，服务器过程发现了这个LREQ事件并调用监听listen，监听从消息队列中取走第一个LCB。
- 如果有一个缓冲与这个LCB关联了并含有传给这个服务器的数据，服务器会调用readlink来读入数据。
- 服务器然后执行任何必需的进程，接着调用writelink响应这个消息，仍然有可能包含一个数据缓冲。这会通过LDONE来唤醒请求者。
- 请求者发现这个LDONE事件，检查结果，然后通过调用breaklink来结束这次交互，断开链接会释放相关的资源。

只不过内核的其他部分直接使用消息系统；文件系统用它和I/O设备及其他进程进行通信。进程间通信的处理方式几乎和I/O完全相同，而且，它也用于支持容错进程对。

这种方式本质上是异步的和多线程的：在调用link后，请求者继续它的操作。许多请求者可以把请求发给同一个服务器，即使当服务器不处理请求的时候。服务器不必立即响应链接请求。当服务器答复时，请求者不必立即确认这个答复。相反，每次当进程可用时会由一个它可以处理的事件唤醒。

8.8.1 回顾一下进程对

容错的要求之一是单个故障不能导致系统崩溃。我们已经看到I/O进程通过使用进程对

注3：　在功能方面，这些名字非常接近于现在的术语：客户端和服务器。

解决了这个问题，而且很明显，这也是处理CPU故障的常规方式。因此Guardian支持用户级进程对的创建。

所有进程对都以一个主进程和一个备份进程的方式运行。主进程进行处理，同时备份进程处于“热备份”状态。主进程时常通过一个名为checkpointing的进程更新备份进程的内存映像。如果主进程出故障或主动放弃控制，备份进程会从最后的检查点开始继续。许多程序实现了checkpointing，它是由消息系统执行的：

- 备份进程调用检查监视器（checkmonitor）以等待主进程的检查点消息。它一直调用检查监视器，直到主进程出故障或放弃控制。在这段时期，这个CPU的唯一用处是和消息系统通信以更新它的数据空间，以及打开和关闭消息系统以更新文件信息。
- 主进程调用检查点把它的部分数据空间和文件信息复制给备份进程。程序员可以决定哪些数据和文件在什么时候应该复制到检查点。
- 主进程对关于文件打开的检查点信息调用checkopen。这实际上产生一个来自备份进程的调用来打开。I/O进程认出这是一个来自备份进程的打开请求并把它和主进程的打开请求同等处理。
- 主进程对关于文件关闭的检查点信息调用checkclose。这实际上产生一个来自备份请求的调用来关闭。
- 主进程可能会调用checkswitch来主动释放对进程对的控制。当发生这种情况时，主进程和备份进程互换角色。

当备份进程从检查监视器返回时，它已经变为新的主进程。它回到原主进程最后调用checkpoint的位置，而不是回到它被调用的位置。然后，它从这个位置开始继续处理。

一般而言，一个进程对的生命周期如表8-1所示。

表8-1：一个进程对的生命周期

主进程	备份进程
进行初始化	
调用newprocess来创建备份进程	
	进行初始化
	调用checkmonitor来接收检查点数据
调用checkpoint	在checkmonitor中等待
调用checkopen	从checkmonitor中调用open
处理	在checkmonitor中等待
调用checkpoint	在checkmonitor中等待

（续）

主进程	备份进程
处理	在checkmonitor中等待
主动切换：调用checkswitch	接管
调用checkmonitor来接收checkpoint数据	处理
在checkmonitor中等待	调用checkpoint
在checkmonitor中等待	处理
在checkmonitor中等待	调用checkpoint
在checkmonitor中等待	CPU出故障
接管	（没用了）
处理	

8.8.2 同步

这种方式是非常可靠的，它可以提供比纯粹的锁步（lockstep）方式更好的可靠性。在一些程序错误中，特别在竞态条件（race condition）下，以锁步方式运行的进程会遇到完全相同的程序错误，还会崩溃。采用连接比较松散的方式通常可以避免完全相同的情形并使程序继续执行。

有两个问题不是非常明显：

- checkpointing是CPU密集（CPU-intensive）的。一个进程的检查点设置应该多频繁？在检查点设置应该处理哪些数据？这些决定留给程序员。如果他做错了或忘记在检查点处理重要的数据，或做的时间不恰当，那么备份进程中的内存映像将会不协调，备份进程就可能出故障。
- 如果主进程执行了外部可见的操作，例如在记入检查点之后但在出故障之前执行了I/O操作，备份进程在接管之后会重复这些操作。这可能会导致数据异常。

实际上，检查点选取异常的问题还没有证实是一个问题，但是，重复I/O的确是一个问题。系统通过给每个I/O请求关联一个序号（称为同步ID）来解决这个问题。I/O进程记录了这些请求，如果它收到一个重复的请求，就简单地返回这个请求第一次调用的完成状态。

8.8.3 网络：EXPAND和FOX

T/16的消息系统实际上是一个独立的网络。通过把消息系统扩展到全世界，Guardian就可能很好地提供大范围的网络。这种实现称为EXPAND。

从程序员的观点来看，EXPAND几乎是完全无缝的。它可以连接多达255个系统。

1. 系统名字

每个系统都有一个以反斜线符号开头的名字，例如\ESSG或\FOXII，加上一个节点号。节点号不像现在的IP地址那么易懂：从程序员的角度来看，仅仅为了给文件名编码才必须需要它们，随后我们将看到它们。

EXPAND是消息系统的一个扩展，所以，大部分细节对程序员都是隐藏的。唯一的问题在于速度和存取需求之间的不同。

2. FOX

完全由于实际的约束，构建一个拥有多于16个CPU的系统是困难的；特别是硬件的约束把处理器间总线的长度限制到只有几米，所以，实际的限制是16个CPU。除此之外，Tandem提供一种把多达14个系统连在一起就像本地集群一样的快速光纤连接能力。在许多方面，这是一个更高速度的EXPAND版本。

8.9 文件系统

Tandem使用术语“文件系统”来表示这种可以提供数据（“读”）或接受数据（“写”）的对系统资源的存取方式。除磁盘文件之外，文件系统也处理设备，例如终端、打印机、磁带单元，还处理进程（进程间通信）。

8.9.1 文件命名

设备、磁盘文件和进程有一个通用的命名规则，但是很遗憾，由于有许多例外，这个规则有点复杂。进程可以拥有名字，但只有I/O进程和进程对才必须拥有名字。不管怎样，文件“名字”的长度为24个字符，由分别为8字节的3个部分组成。只有第一部分是必需的；其他两个部分只用于磁盘文件和指定的进程。

没有命名的进程只使用名字的前8个字节。未配对的系统进程（例如监视器或内存管理器）的命名格式如图8-7所示。

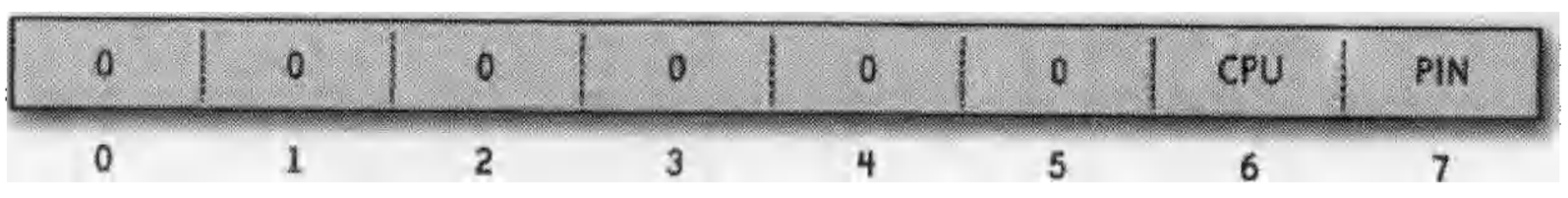


图8-7：未配对系统进程的命名格式

未配对用户进程的命名格式如图8-7所示。

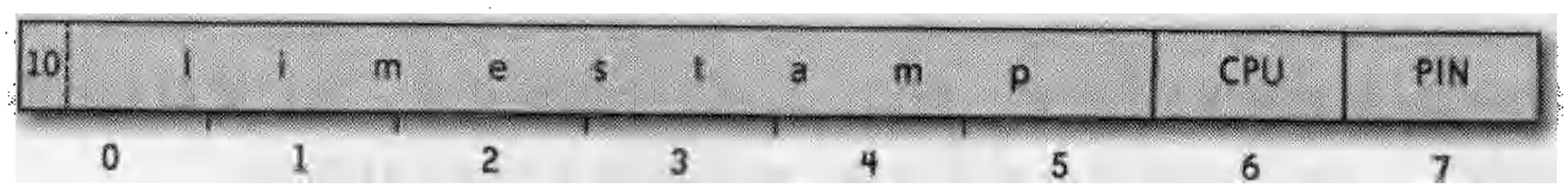


图8-8：未配对用户进程的命名格式

CPU和PIN组合在一起形成了进程ID或PID。PIN是CPU内部的进程标识号。这就限制了每个CPU最多拥有256个进程。

真正的名字以一个“$”符号开始。设备只使用前8个字节，而磁盘文件使用全部的三个部分。单个部分看起来像是磁盘名、目录和文件，虽然实际上每个磁盘只有一个目录。进程可以把其他两个部分用来相互传递信息。

表8-2中列出了一些典型的文件名字。

表8-2：典型的文件名字

$TAPE	磁带驱动
$LP	打印机
$SPLS	打印缓冲进程
$TERM15	终端设备
$SYSTEM	系统磁盘
$SYSTEM SYSTEM LOGFILE	$SYSTEM磁盘上的系统日志文件
$SPLS #DEFAULT	默认的打印缓冲队列
$RECEIVE	进来的消息队列，用于进程间通信

如果名字的某一部分的长度少于8个字节，就会用ASCII的空格填补。从外表来看，名字表示为有句点的ASCII字符，例如$SYSTEM.SYSTEM.LOGFILE和$SPLS.#DEFAULT。

在命名规则中还有较多的约定。进程的子名称必须以井号（#）开始，而用户进程的名字（不包括设备的名字，它是实际的I/O进程的名字）在第一部分的最后有PID（如图8-9所示）。

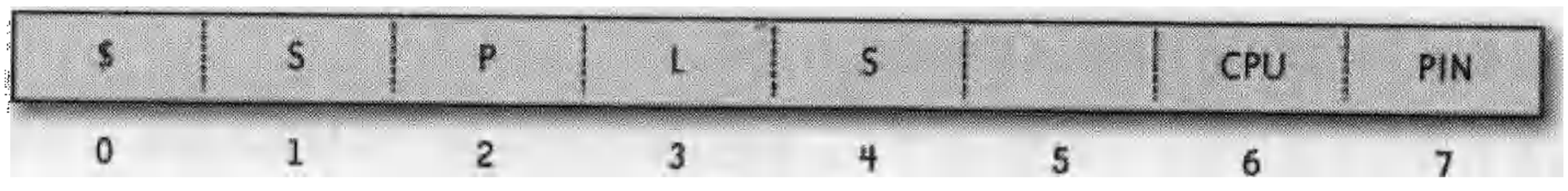


图8-9：指定用户进程的命名格式

这个例子中的PID是主进程的PID。这样就把用户进程名的长度限制到6个字符，包括起始的$。

这样似乎还不够，对于远程系统上指派的进程、磁盘文件或设备还有一组单独的命名。在这种情况下，起始的“$”符号替换为“\”符号，名字的第二个字节是系统号码，把名字的其余部分向右移一个字节。这样，如果一个进程是网络可见的，那么，该进程的名字的长度将限制为最多5个字符。所以，从另一个系统来看，我们先前看到的打印缓冲进程将拥有\ESSG.$SPLS这个外部名称，内部的格式如图8-10所示。

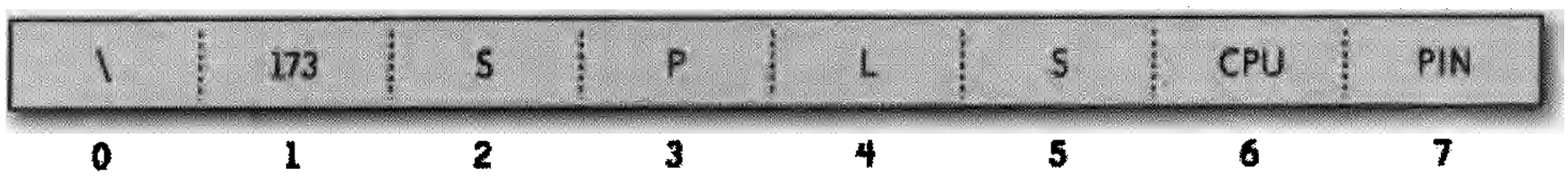


图8-10：网络可见进程的命名格式

数字173是系统\ESSG的节点数。

8.9.2 异步I/O

文件系统接口的重要特性之一是强调异步I/O。我们已经知道消息系统本质上是异步的，所以，这实现起来相对简单。

进程在打开一个文件的时候可以选择同步的或异步（“不等待”）的I/O。当以不等待的方式打开一个文件时，这个I/O请求会立即反馈，只有一些显而易见的错误会报告出来，例如，如果这个文件描述符是不公开的。随后，用户调用awaitio来检查该请求的状态。这引起这样一种编程方式：一个进程发起许多不等待的请求，然后进入调用awaitio的中心循环并处理完成的请求，典型的是发起一个新的请求。

8.9.3 进程间通信

在文件系统级别，进程间通信是到消息系统的一个相对直接的接口。这引起一个问题：消息系统是不对等的。请求者发送一个消息并可能接收一个响应。响应文件系统的一个read命令是没有意义的。在服务器端，服务器读取一个消息并响应它；响应一个write命令也没有意义。

文件系统提供读和写的过程，但是读的过程只对I/O进程起作用，把它们映射到消息系统的请求。在进程间通信级别，读过程没有用，而且，实际上，写过程也用得不多。请求者改为调用一个名为writeread的过程先写一个消息传给服务器，然后从服务器获得一个响应。消息或响应都可以为空（长度为0）。

这些消息到达服务器的消息队列。在文件服务器级别，消息队列是一个名为$RECEIVE的伪文件。服务器打开$RECEIVE并通常使用readupdate过程来读取一个消息。随后，它可以用reply过程进行响应。

8.9.4 系统消息

系统使用$RECEIVE把消息传给进程。最重要的消息之一是startup消息，它把参数传给新启动的进程。下面的例子是用TAL编写的，TAL是Tandem的低级别系统编程语言（虽然这个名字表示“Tandem Application Language”）。TAL源自于HP的SPL，而且它类似于Pascal和Algol。它的最不寻常的特征之一是在标识符中使用脱字符号（^）；下划

线字符（_）是不允许使用的。这个例子应该和C非常相似，并很容易理解。它展示了这样一个进程：启动一个子服务器进程，然后和它通信。

第一部分代码展示了父进程（请求者）：

```
call newprocess (program^file^name,,,,,, process^name); -- start the server process
call open (process^name, process^fd);                   -- open process
call writeread (process^fd, startup^message, 66); -- write startup message
while 1 do
  begin
  read data from terminal
  call writeread (process^fd,
                  data, data^length,        -- write data
                  reply, max^reply,         -- read data back
                  @reply^length);           -- return real reply length
  if reply^length > 0
    write data back to terminal
  end;
```

接下来的代码展示了子进程（服务器）：

```
call open (receive, receive^fd);
do
  call read (receive^fd, startup^message, 66);
until startup^message = -1;                  -- first word of startup message is -1.
while 1 do
  begin
  call readupdate (receive^fd, message, read^count, count^read);
  process message received, replacing buffer contents
  call reply (message, reply^length);
  end;
```

子进程接收到最初的消息是系统消息：子进程的父进程open给子进程发送了一个open消息，然后，最初的writeread调用发送了启动的消息。子进程处理这些消息并响应它们。它可以使用这个open消息来记录请求者或接收这个文件名最后的16个字节所传过来的信息。只有这时候这个进程才从它的父进程接收正常的消息。这时候，其他进程也可以和这个子进程通信。同样地，当一个请求者关闭服务器时，服务器收到一个close的系统消息。

8.9.5 设备I/O

记住这很重要：设备I/O（包括磁盘文件I/O）都由I/O进程处理，所以，“打开一个设备”实际上是打开了I/O进程。还要记住，到设备的I/O和到文件的I/O的实现稍微有些不同，虽然文件系统的过程都一样。特别地，典型的用于访问文件的程序是比较常规的读和写，而通常的磁盘I/O不是“不等待”的。

8.9.6 安全

为了使时间一致，T/16不是一个绝对安全的系统。实际上，这没有引起任何严重的问题，

但是，有一个问题值得提一下：从非特权程序到特权程序的转换是基于程序入口点在PEP表中的位置及环境寄存器中priv位的值。在早期，这种利用就十分明显。如果你可以让一个特权程序通过一个指针返回一个值并让这个程序以priv位的值被设置的方式重写栈上保存的环境寄存器，这个进程可以依赖那个程序的返回值保持特权。这是可调用程序的职责：检查它们的指针参数以确保它们没有任何寻址异常，而且它们返回的值仅适用于用户环境。程序setlooptimer（设置watchdog定时器并任意地返回原值）中一个缺陷使变为SUPER.SUPER(root用户，拥有255、255或-1的ID)成为可能。

```
proc make^me^super main;
begin
int .TOS = 'S';                          -- top of stack address

call setlooptimer (%2017);               -- set a timer value
call setlooptimer (0, @TOS [4]);         -- reset, return old value to saved E reg
pcb [mypid.<8:15>].pcbprocaid := -1; -- dick in my PCB and make me super
end;
```

对setlooptimer的第二次调用把原值%2017返回给栈中保存的环境寄存器的内容，尤其是设置priv位，这使这个进程处于特权状态。理论上，这个值可能会缩减到%2016，但是，这没有任何区别（这是保存的RP区域，不能恢复）。这个程序然后在它自己的进程控制块（PCB）中使用相对SG的寻址来修改用户信息。mypid是返回当前进程的PID的一个函数，而最后的8位（<8:15>）是PIN，用作为PCB表中的索引。

当然，这个缺陷很快就修复了，但是它显示了这种方式的一个缺点：它依赖程序员去检查传给可调用程序的参数。在这个架构的整个生命周期中，又重新出现了这样的问题。

8.9.7 文件访问安全

Tandem文件访问安全的方式类似于UNIX的方式，但是，用户可以只属于单个组，那是用户名的一部分。因此，我的用户名SUPPORT.GREG（用数字写成20，102）表示我只属于SUPPORT组（20），而在那个组内我的用户ID是102。这些字段的每一个都是8位长，所以，完整的用户ID正好一个字。如果我要成为另一个组的成员，我将需要另一个用户ID，可能会有一个不同的数字——例如，用户ID为255，17的SUPER.GREG。

每个文件都有许多位描述了拥有者（owner）、组（group）或所有用户（all）对这个文件的访问权限。然而，不像UNIX，这些位以不同的方式进行组织：四个权限是read、write、execute和purge。purge是Tandem对delete的称呼，这是必需的，因为目录不具有它们自身的安全设置。

对于每一种访问模式，都可以选择允许谁使用它们：

- Owner意味着只有这个文件的拥有者。
- Group意味着同一组中的任何人。
- All意味着所有人。

所有这些都只涉及文件所在的同一个系统。对于网络，还引入了另一组模式来控制其他系统的用户的访问：

- User意味着只有具有与该文件拥有者相同的用户名和组编号的用户。
- Class意味着具有和该文件拥有者相同的组编号的任何人。
- Network意味着任何人，任何地方。

对于设备没有任何安全性，用户进程必须自己进行控制。前者在网络环境中尤其是一个缺点。在1989年初的一个安全性讨论会上，仅仅通过在这个系统控制台上放置一个假的提示符，我能够演示偷到一个位于美国库珀蒂诺管理区域中部的\TSII系统上的SUPER.SUPER（root）的口令。那时候我在德国的杜塞尔多夫。

8.10 轶闻趣事

回到21世纪早期，很容易忘记用计算机进行工作的乐趣。Tandem是一个有趣的公司，而且这个公司关心它的职员。在1974年末的一个星期五，在这个系统的开发初期，几个创始人最终使这个软件运行在硬件上了；在这之前，这个软件一直在模拟器上进行开发。你可以想象那个激动的场面。故事开始了，一个副总裁出去并带回来一箱啤酒，然后，所有人都围坐在啤酒箱周围，庆祝并讨论将来。他们讨论的一件事是应该每周一次喝一箱啤酒，就这样，Tandem的啤酒会诞生了。它真的延续到了20世纪90年代，然后它变得逐渐不适宜，最后取消了。

Tandem提出了许多口号和文字游戏，当然，“Tandem”这个名字本身就是其中的一个。那时候，我们拥有印着像“So nice, so nice, we do it twice（非常好，非常好，我们做了两遍）”、“There's no stopping us（没有什么可以使我们停下来）”和“Tandem users do it with mirrors（Tandem用户用镜像完成了任务）”这样口号的T恤衫。当然，当有好奇心特别强烈的人问起时，标准答案是“这是为了防止其中一个出现故障”。

最后的口号不仅仅是俏皮话。它深深地影响我们的思考进程。在1977年5月，从Tandem最初的五个星期培训回来后，我难过地发现我的猫逃走了。在确定它不会回来后，我出去并又领了两只新猫。不久之后我意识到这是成功洗脑的结果。即使在今天，我仍然讨厌重启计算机，除非绝对不能避免。

8.11 弊端

就预期的目标而言，T/16是一款非常成功的机器（美国的ATM一度有超过80%由Tandem系统控制），但是当然也存在缺点。有些缺点（例如，比常规系统更高的成本）是不可避免的。另一些缺点对于设计者来说不是那么显而易见的。

8.11.1 性能

当增加硬件时性能获得接近线性的提高，Tandem有理由对此感到自豪。引用了随后的TXP系统的Hors和Chou（1985）展示了一个FOX集群如何能够从2个处理器线性地扩展到32个处理器。

Bartlett（1982）展示了它的弊端：消息系统的性能限制了系统的速度（即使是小系统）。没有附加数据的单个消息需要花2毫秒来传输，在每个方向上都有2000个字节数据的消息需要花4.6毫秒（相同的CPU）到7.0毫秒（不同的CPU）。这是单个I/O操作的开销，即使在当时，这也显得很慢。对一个磁盘文件的连续I/O请求之间的延迟如此长，以致它们直到数据传输到了磁盘头才会发生，这意味着磁盘每转一周才能满足一个请求。一个从磁盘中连续读取2KB数据并进行处理的程序（例如，类似于grep）将只能获得120KB/s的吞吐量。较小的I/O能力（例如512字节）可能使吞吐量受制于软盘的速度。

8.11.2 硬件限制

正如名字“Tandem/16”所示，设计者有一个16位的头脑。在20世纪70年代中期，这相当典型，但是，事实是“真的”计算机拥有32位的字。随着时间的推移，后续的机器会解决其中的很多问题。在1981年，Tandem通过一个向上兼容的指令集及少量硬件限制引入了NonStop II系统。在接下来的10年里，许多兼容但更快的机器引入了进来。没有一个特别快，但它们对于在线交易处理已经足够快了。另外，操作系统也进行了重写以解决比较紧迫的问题，而且，随着时间的推移还进行了额外的改进。这些改进包括：

- 引入了一个31位的寻址模式以给用户进程“无限的”存储空间。这个模式使用字节地址，但是，它没有移除栈大小和代码跨越32KB边界的限制，因为老的指令格式仍然保留着。
- 增加了硬件虚拟内存映像的数量。T/16只有4个，用于代码空间和数据空间。TNS/II，正如它的名字一样，总共拥有16个内存映像，这意味着处理器可以直接寻址到2MB而不必使用内存管理器。这些映像之一用作为类似于转换旁视缓冲以处理31位扩展地址。
- Guardian II，与TNS/II一起发布的Guardian新版本，也展示了系统库和用户库空间，它们把进程可用的总空间增加到了384KB。尽管如此，随后，通过段切换

(segment switching)，库空间的数量从2个（系统和用户）增加到多达62个（系统和用户各31个）。在任何时候只有一个用户库和系统库映射可以是活动的。

- 消息队列的大小被证实是一个问题。监视器进程每隔一定间隔把状态信息发送给需要它们的每个进程。如果这些进程不读取信息，非常多的资源（LCB和消息缓冲）会用于复写消息。为了解决这个问题，Guardian II引入了这样一个进程：保持这些状态信息的单个副本并当其收到提醒时把消息发送给一个进程。

8.11.3 错过的机会

T/16是一台革命性的机器，除此之外，它还提供了当时其他机器很少拥有的一个环境。不过说到底，妨碍的正是这些小事情。例如，设备无关性是操作系统最持久的目标之一，Tandem为实现这个目标提供了很大的帮助。不过，由于命名问题和几乎不必要的不相容性，它们失去了全部的潜力。为什么进程间通信不可能使用读取的方式？为什么进程名和设备名在格式上必须区分？为什么在第九个字节需要一个#符号？

8.11.4 分裂的大脑

一个比较严重的问题在于探测错误的基本方式。当只有一个组件出故障时这种方式工作得很好，即使两个组件出故障，这种方式通常也工作得相当好。但是，如果两个处理器间总线都出故障，那会发生什么呢？每个CPU都假定另一个已经出故障并接管I/O设备——不是一次，而是会持续不断。这样的情况确实出现过，但很幸运，非常少，它们通常导致在两个CPU之间共享的磁盘数据完全损坏。

8.12 后继者

从1990年开始，许多因素导致Tandem的销售出现滑坡：

- 计算机硬件总体正变得更可靠，这减小了Tandem的优势。
- 计算机硬件的速度提高了许多，从而突出了架构的一些基本性能极限。

在20世纪90年代，T/16处理器架构被基于MIPS的解决方案所替代，虽然剩下的许多架构仍然在使用。另一方面，性能上的差异仍然很大，以致即使到了2000年，Tandem仍然一直使用MIPS处理器来模拟T/16指令。其中的理由之一是Tandem的大多数系统级软件仍然用与T/16架构紧密相连的TAL语言编写。把源代码移植到C的提议由于高昂的费用而被否决了。

对于这样一个革命性的系统，Tandem/16对行业和现代计算机的设计只产生了极小的影响。很多功能现在很容易用到——镜像磁盘、网络文件系统、客户端-服务器模式或热

插拔硬件。但是，很难看到其中存在由Tandem引领的功能。这可能是因为T/16与大部分系统差异很大，当然，还因为把它开发出来的纯商业环境的改变。

8.13 延伸阅读

在Hewlett-Packard的网站上有许多论文；从位于http://www.hpl.hp. com/techreports/tandem/ 的Tandem技术报告开始看起。尤其是：

Bartlett, Joel. "A NonStop Kernel," June 1981. *http://www.hpl. hp.com/techreports /tandem/TR-81.4.html?jumpid=reg_R1002_USEN.* (关于操作系统环境的更多信息。)

Bartlett, Joel, et al. "Fault tolerance in Tandem computer systems," May 1990. *http://www.hpl. hp.com/techreports/tandem/TR-90.5.html.* (描述硬件的细节。)

Gray, Jim. "The cost of messages," March 1988. http://www. hpl.hp.com/ techreports/tandem/TR-88.4.html. (从理论视角描述性能问题。)

Horst, Robert, and Tim Chou. "The hardware architecture and linear expansion of Tandem nonstop systems," April 1985. *http://www.hpl.hp.com/techreports/tandem/TR-85.3.html.*

	原则与特性		结构
	功能多样性	√	模块
	概念完整性		依赖关系
	修改独立性	√	进程
√	自动传播		数据访问
	可构建性		
√	增长适应性		
√	熵增抵抗力		

第9章

JPC：一个纯Java的x86 PC模拟程序

Rhys Newman

Christopher Dennis

“模拟程序很慢，Java也很慢；因此，两者的结合只能意味着以蜗牛一样的速度进行计算。”正如这个传统智者所说，第一个JPC原型比真的机器慢10 000倍。

不过，纯Java x86 PC模拟程序是一个引人注意的想法——想象一下，在一个安全的Java沙箱内启动Linux和Windows时速度仍然保持快到足于实际使用。这个任务可能不容易，因为它需要复制人类已经创造的最复杂的机器之一的内核。仿制物理x86 PC的设计、部署在Java虚拟机之上，然后在Java Applet沙箱的安全限制之内装配这些成果，完成这个任务将是一个困难重重的探索之旅。

我们在这个过程中体验了现代软件工程师很少遇到的运算的挑战，但是这也对那些通常认为理所当然的基础提出了适时的提醒。现在，我们拥有了一个架构，它表明了纯Java的x86 硬件模拟程序是可行的且速度最终能够快到足于实际使用。

9.1 简介

随着处理器速度的加快以及家用计算机用户使用的网络性能的提高，越来越多在几年前还认为是不切实际的事情正变得平常。十年前，当一家名为VMWare的技术小公司在加利福尼亚成立时，把一台完全虚拟的计算机作为软件运行在一台物理计算机内，这种观点还被认为相当难以理解。毕竟，如果你拥有一台计算机，为什么仅仅为了要运行你已经在运行的系统而增加一个虚拟层，从而使计算机变慢呢？你使用的软件需要硬件全力来运行，如果需要，可以直接买更多的机器来做更多的工作，为什么这样做呢？

十年后我们都看到了虚拟机的好处。硬件的速度如此快，以致现代的机器可以运行许多虚拟机而对整体的性能没有大的影响，软件服务的重要性如此显而易见，以致在虚拟机中完全隔离软件服务的安全性和可靠性的优点非常明显。

然而，纯虚拟化也有它的问题，因为纯虚拟化在一定程度上依赖硬件的支持（注1）来运行，因此，它也受到与物理机器如此紧密地连接而导致的不稳定性的影响。与此相比较，模拟程序是完全用软件构建的虚拟计算机，因此，对底层的硬件没有特别的要求。这意味着模拟的计算机和实际的硬件完全分开。与通常的应用程序软件相比，它在系统上的存在既不遭受也不引起额外的不稳定性。正如最初运行应用程序软件是计算机存在的原因一样，模拟程序将始终是创建虚拟机的最稳定和最安全的方式。

正如十年前对虚拟化一样，目前对模拟程序的批评集中在它引起的速度损失上，这通常很明显。然而，历史证明，速度问题会随着技术的进步而得到解决，即使是现代日益复杂的硬件和软件栈，甚至在硬件、操作系统和应用程序软件之间微妙交互而引起的更困难的问题。在非常低的层次以非常强壮和安全的方式分开系统，与此同时仍然共享物理资源，这已经变得日益必要，但也日益困难。模拟程序提供了必需的强壮性、安全性和灵活性，因此，它会成为一个日益令人注意的选择。

目前有许多模拟程序可以使用，Bochs和QEMU是模拟x86 PC最著名的例子。两者都开发了很多年，都能够准确地启动现在的操作系统并运行应用程序软件。然而，两者都是用本机码（C/C++）编写的，如果它们要运行在新的底层硬件架构/OS栈上（例如，一个新型的宿主系统），就需要重新编译。此外，对于安全级别非常高的应用程序，总是有这样的担忧：模拟程序已经被恶意篡改了，或让访客的代码做恶意的事情，或包含一个容许这些问题的缺陷。例如，QEMU使用动态二进制转换来达到可接受的速度，如果这个进程被损坏或被发现有一个缺陷，那么，这个软件就会变得不可靠或破坏安全保护措施。

如果用户能够接受模拟程序在安全可以绝对保证的时候的速度损失，那么，为什么不在部署最广泛和最安全的Java虚拟机（Java VM，JVM）上建造一个虚拟程序呢？JVM作为运行代码的一个安全方式已经测试了十多年，用户通常愿意让从因特网下载下来的未认证代码在Applet沙箱（JVM提供的安全容器）中执行。这是因为JVM提防基础的编程错误，例如内存溢出和读取未分派内存中的数据。此外，安装了安全管理器以加强沙箱的JVM可以禁止访客软件尝试的任何敏感操作。

JPC完全是用Java编写的一个x86 PC模拟程序。在JPC中没有本机码；它模拟了一个x86

注1：　最起码你需要和“虚拟化的硬件”相同的硬件。因此，像Xen和VMWare这样的产品使虚拟的x86 PC只能创建在x86硬件上。

PC的所有标准硬件组件且完全在Java Applet沙箱内。因此，只要不越出这些安全约束，在JPC内运行的x86操作系统和软件就完全和底层的硬件隔离，甚至连硬件也不必是一台x86 PC。（注2）从宿主的角度来看，JPC代表的仅仅是一个Java应用程序/Applet，因此，宿主可以确信运行的代码（无论它是什么）是安全的。JPC可以启动DOS和现代的Linux，让访客的软件和操作系统完全无拘无束地访问虚拟机中的所有硬件，包括root/admin的权限，而所有的这些都在Java的沙箱内。

为了攻破JPC，攻击者必须在JPC的代码中发现与JVM中缺陷相符的缺陷，这可以使一个敏感的动作在一台也在运行这个JVM的用户的权限之内的宿主计算机上运行。这意味着打破了3个完全独立的安全层。通常每一层都是由完全不同的公司构建的，而且因为在如此多不同的环境中每一层都是必需的，它们的安全一直在不停地进行独立的测试和检查。还要注意在破坏这些层中的一致性要求。在JPC中发现一个缺陷后接着去攻破JVM是行不通的；黑客必须在JPC发现一个直接（且已经）连接到正使用的JVM内的一个相配安全缺陷的缺陷。由于JPC是开源的，检查代码并构建一个“干净”的版本是一个相对简单的任务，可以把一个安全性增强的JVM用于要求高安全性的应用程序。因此，JPC成为有恶意的x86代码无法攻破的屏障，而且真正提供了最可靠、便捷和安全的方式来检查运行中的恶意x86代码。

和虚拟化一样，当硬件的速度加快时，模拟的应用程序可以从安全领域延伸到需要保证健壮性的负责重要任务的系统。无论模拟的机器如何崩溃（或如何损坏），宿主都可以不受影响并能够继续运行其他模拟的实例。这个技术可以避免在现代的硬件支持虚拟化的有用特征中即使经过最仔细的思考还可能存在的问题。（注3）

在这一章的其余部分，我们将讨论用于构建原型和开发JPC的进程，我们还将说明如何对设计进行许多逐步改进才形成今天这个可用的JPC。然后，我们再略述一些应用程序和JPC呈现的独特技术的含义。

9.2 概念验证

x86 PC已经存在了三十多年，并从许多不同代的硬件中发展而来。在每一个时期，它都一直保持向后兼容，所以，即使在今天，最初的8086程序也可以运行在一台新的PC上。虽然这具有毋庸置疑的优点且导致这个平台空前的成功，但是，这也意味着它为了避免破坏现有的代码，在合并新技术时变得非常复杂。如果现在重新构建一台PC，硬件的许多方面都会截然不同，而且几乎肯定会简单得多。

注2：JPC已经适合在J2ME环境中运行并能够在一个基于ARM11的移动电话上启动最初的和未更改的DOS！

注3：例如，硬件支持的x86 CPU虚拟化（Intel VT、AMD Pacifica）由于多核芯片的L1/L2高速缓冲寄存器共享而具有安全弱点。

然而，x86平台到处存在，当今世界有超过10亿台，并且每年生产超过2亿台。因此，最有用的模拟程序将是以x86 PC架构为目标的模拟程序。

然而，这不是一项容易的工作。必须用软件模拟的硬件就包括x86处理器、磁盘（和它的控制器）、键盘和鼠标、VGA图形显卡、DMA控制器、PCI总线、PCI主桥、内部定时器、实时时钟、中断控制器和PCI ISA桥。每个设备都有它自己的规范说明书，必须阅读它们并翻译成软件。x86处理器手册有1500多页，技术手册总共大概有2000页。x86指令集非常大，一个程序代码中可能用到多至65 000条指令，在内存页上设置有4个不同的保护级别，4个不同的处理器“模式”，在每个模式中，指令都可能（而且确实）有不同的效果。

所以，在着手这项巨大的任务之前，评估预期的成果是否值得，这很重要。Bochs和QEMU项目都完成了这个壮举，通过检查他们解决这些问题的途径，我们可以获得一些信心。然而，这两个项目都是C/C++的程序，因此，只能获得有限的帮助，因为JPC必须运行在有额外设计约束和性能考虑的纯Java环境中。

通过选择模拟一组简单的硬件进行一些简化是有可能的。因为处理器模拟将是系统中的主要瓶颈，模拟一个复杂但快速的硬盘控制器（或驱动器）没有多大用处。一个简单并可靠的硬盘模拟程序就可以容易地满足，这对于其他的硬件组件同样适用。即使在考虑处理器的时候，可以选择奔腾II作为目标，那就没有必要模拟最新芯片中出现的扩展指令集。因为现在的软件，包括操作系统，即使没有这些指令保证向后的兼容性，但也能运行得相当好，这个决定没有明显的局限性。

然而，模拟程序的速度仍然是主要的挑战。获得最好性能的显而易见的方式是使用某种形式的动态二进制转换从不流畅的按部就班的软件模拟程序变成一个编译模式，在编译模式中，底层硬件的自身速度可以更有效地使用。这种技术使用在许多不同的外表下；现代的x86处理器把首次使用的x86指令分解为更小的微码，然后缓存下来以便快速地重复执行。即时编辑的Java环境同样把字节码编译成本机码以提高重复许多次的代码的执行速度。这种技术的效率是基于这个事实：在几乎所有的软件中，10%的代码使用了90%的执行时间。（注4）找到提高重复这10%“热”代码的速度的简单的方法可以显著地提高整体的速度。还要注意，应用这种技术并不意味着编译所有的代码，选择性的优化可以导致巨大的性能提升而不会引起无法接受的延迟，如果所有代码都进行优化，就可能发生这种情况。

注4：在DOS的启动次序中和在玩许多的DOS游戏时，利用JPC，这些单凭经验的说法实际上已经得到了证实。可以整体控制一个模拟程序后，编译出这些与程序执行有关的统计很容易。也可以参考Donald E. Knuth的“An empirical study of FORTRAN programs.” (Software Practice and Experience, 1: 105-133. Wiley, 1971.)

几乎所有的软件（无论是从C/C++或Java字节码编译到本机的x86）在执行前都由现代的计算机进行或多或少的动态二进制转换，这个事实显示了这种技术的能力和效果。因此，当着手处理创建JPC的这个任务时，如果可以使用相似的技术，获得合理的速度是有希望的。

通过从现有的模拟程序获得这项工作是可行的信心，通过选择一个简单的PC架构作为最初的目标，我们把最初的范围降低到一个可完成的级别。然后，我们需要评估动态二进制转换提高速度并确保达到一个（即使在大体上）实际可用的性能的可能性。如果JPC在应用了所有的编程技巧后的最优结果是以1%的速度运行，那么，这个项目将是不值得做的。

潜在的处理器性能测试

为了估计使用各种模拟策略后能获得的潜在性能级别，作为一个简单的模型，一个“Toy”处理器发明了出来。这个Toy处理器有13个指令、2个寄存器和128字节的RAM。用Toy的汇编语言编写了一个相当于下面这段C代码的简单程序以进行速度测试：

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 50; j++)
    memory[51 + j] += 4;
```

存储器最初都是零，这个程序的输出是最终的存储器状态（存储器的最初50个字节留给这个程序代码使用）。

Toy架构的一个Java模拟程序构建了出来，在Sun HotSpot VM上连续运行这个程序十万次花了8000毫秒。在相同的硬件上，一个用GCC编译的C程序花了86毫秒（用所有静态优化进行编译）。毫不奇怪，本机模拟引起了两个数量级的性能损耗。但是，这个简单的模拟程序确实是非常简单，每次都从寄存器中读取下一个汇编指令，通过一个switch语句选择要执行的操作，而且一直循环直到完成。

一个较好的版本将会除去这个查找-分派过程。这表现了内联的普通诀窍；C编译器在编译的时候内联经常用到的代码，而HotSpot VM在运行时进行内联以响应真正的执行遥测。做到这一点后，模拟程序花了800毫秒，速度提升了10倍。

当前，当使用查找-分派时，指令指针必须在每个指针之后更新以便处理器知道从哪儿获取下一个指针。然而，用内联代码，只有在整个内联块执行完成时指令指针才必须更新。移除这些交叉存取的增量也可以帮助HotSpot优化代码；它可以集中在关键操作上而不必担心这个指令指针寄存器必须始终与进度保持一致。由于把指令指针的更新移到内联程序部分的末端，执行的时间减少到250毫秒，又额外提高了3.2倍。

然而，由一个过分简化的自动算法翻译的汇编代码导致在内存和寄存器之间许多不必要

的数据移动。在好几个地方，一个结果刚存储到内存的字节数组就立即再次被读出。通过一个简单的流控制分析就可以自动发现这些无效率的地方并移除，这导致执行的时间减少到80毫秒。这和优化的本机程序一样快！

因此，通过适当的运行时编译（这并不是非常复杂），一个Java模拟程序可以以和本地硬件相同的速度运行这个Toy本机代码。很明显，在分析x86 PC的许多更复杂的硬件时期望获得100%的速度是不实际的，但是，这些测试确实表明实际可用的速度是可达到的。

所以，不论最初对纯Java x86模拟程序的整个概念存什么怀疑，现在都有充分的证据表明可以构建这样的技术。接下来的章节详述了如何利用x86 PC硬件设计的架构观点及如何映射到JVM以编写一个高效的模拟程序。还会概要地讲述许多完全基于JVM如何运转的非常重要的软件设计决策在什么地方拥有显著的性能优点。

9.3 PC架构

现代的PC是一个非常复杂的怪兽。它的硬件已经优化和迭代了许多次以形成一个高效和普遍的计算平台。然而，它也保留了设计来维持它向后兼容的遗留组件和功能。在某些方面这是好事。IA-32机器的基本架构从它自1985年的386中首次出现以来没有改变过。实际上，在系统架构方面，386本身也称不上背离它的x86前身。

虽然图9-1在某些方面非常简单，有一些不同的文字和重复的方框，但是，这可以作为JPC本身的架构图来容易地传递信息。

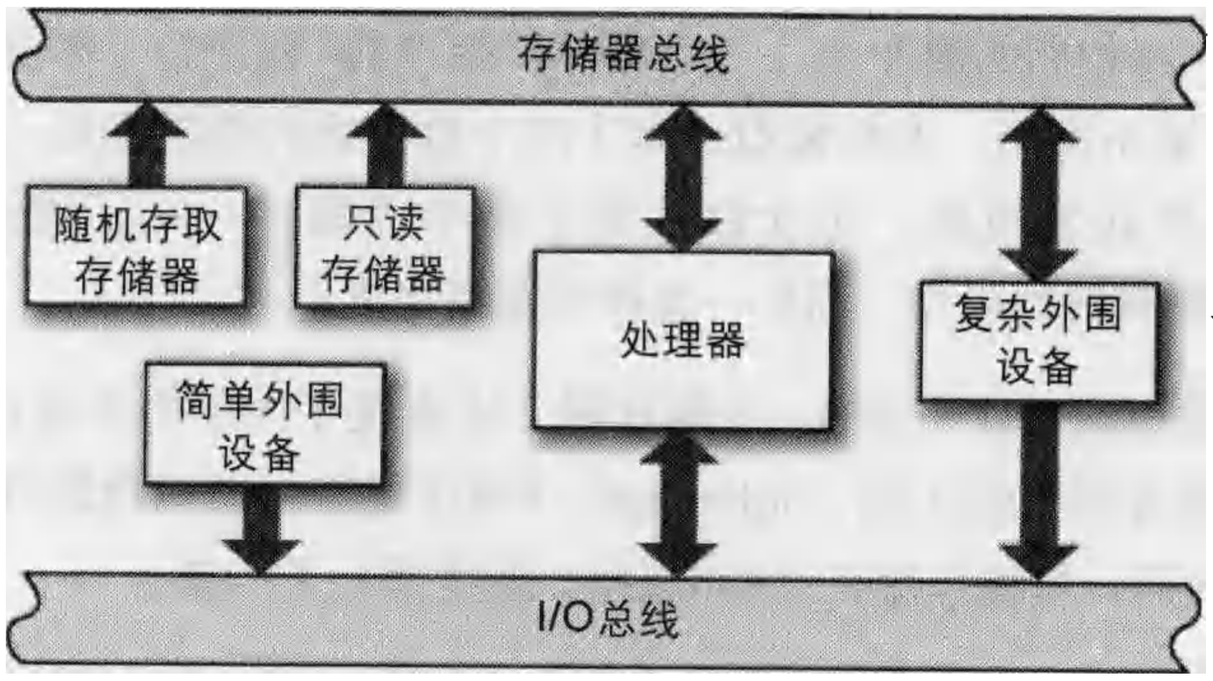


图9-1：现代PC的基础架构

JPC的大部分设计是相对简单的系统分析，对于这个模拟程序来说，把初始的系统映射到JPC也几乎是在硬件规范和Java类之间进行一一对应。例如，JPC中的串行端口由单个SerialPort类表示，它实现了HardwareComponent和IOPortCapable。这种简化的方式导致了一种易于理解和掌握的设计，基本上，这个架构内的对象彼此都是松散耦合

的。这使JPC具有了非常灵活的优点，就像在实际的机器中，虚拟设备可以插入PCI总线，组件可以相互交换来构建符合很多规范的虚拟机。

违背这种方式的唯一理由是设计和模块化的清晰度直接和性能相抵触。这种情况在JPC中出现在两个关键的地方：一次在计算的中心（处理器），一次在带宽的中心（存储器系统）。这两个热点对这个项目有两个影响：

- 代码基变得不平衡。像IDE接口这样简单的硬件设备仅仅是规范文档的翻译，对应于两个类：IDEChannel和PIIX3IDEInterface。处理器，一个复杂得多的设备，相对拥有很多相关的代码。总体上，它由8个不同的包和50多个类来代表。
- 作为开发人员，我们发现必须变得和代码基一样分裂。坦率地讲，当你处理存储器系统或处理器系统之外的硬件模拟程序时，你的目标是最终代码的清晰和模块化设计。当你在处理器系统或存储器系统内工作时，你的目标是最终的性能。

在处理架构的敏感部分时对独创性必须有悲观的心理准备。为了获得尽可能高的性能，我们必须不断地准备试验。即使对源代码的小改动也必须持怀疑态度去审视，直到能证明它们至少对性能没有负面的影响。

在模拟程序的瓶颈处要求最高性能，这使JPC成为一个令人愿意去做和愿意与其合作的有趣项目。这一章的其余部分集中在我们如何达到我们坚信的可维护和合理设计，且不牺牲系统的性能。

9.4 Java性能技巧

优化的第一准则：不要优化。

> 优化的第二准则（仅限于专家）：还是不要优化。
>
> ——Michael A. Jackson

和所有的性能技巧一样，下面的技巧是指导方针，不是准则。良好设计和清晰编码的代码几乎总是优于“优化的”代码。只有看到对设计有积极的影响或对性能真的有必要时再参考这些指导方针。

技巧#1：创建对象不好

过多的对象实例（尤其是短期的对象）会导致不好的性能。这是因为大量对象引起频繁的年轻代（young generation）垃圾收集，而年轻代垃圾收集算法几乎是“全部停止”类型的。

技巧#2：静态好

如果一个方法可以定义成静态的，那就定义成静态的。静态方法不是虚的，所以，不会动态分派。与实例方法相比，高级的VM更容易也更愿意内联静态的方法。

技巧#3：表切换好，查找切换不好

标签集适度紧凑的switch语句比标签集的值更分散的switch语句要快。这是因为Java对switch有两种字节码：tableswitch和lookupswitch。表切换是利用一个间接调用执行的，用switch的值把偏移量提供到一个函数表中。查找切换要慢得多，因为它们按对应关系进行查找以发现一个匹配的值：函数对。

技巧#4：方法越小越好

小块的代码好，因为即时的环境通常按一个方法的粒度去理解代码。包含一个“热”区域的大方法将会整个编译。由此产生的大块本机代码会引起更多的缓存代码未能利用，这会影响性能。

技巧#5：Exception用于异常的情况

Exception应该用于异常条件，不要用于错误。把异常用作为非正常环境中的流控制会提示VM优化正常的路径，从而带来更好的性能。

技巧#6：小心使用装饰（decorator）模式

从设计的角度来看，装饰模式是好的，但是，它的额外间接性开销很大。请记住，移除和增加装饰模式都是允许的。这种移除会被认为是一次“异常事件”并可以通过抛出一个专门的异常来实现。

技巧#7：对类进行instanceof的操作更快

对一个类进行instanceof的操作比用接口来操作它快得多。Java的单继承模式意味着对于一个类，instanceof仅仅是一次减法和一次数组查找；对于一个接口，那是一次数组搜索。

技巧#8：最低程度地使用同步

把同步的块减到最小；它们会引起不必要的开销。如果可能，则考虑把它们替换为原子的或volatile引用。

技巧#9：小心外部的库

避免使用超出你的目的的外部库。如果任务简单而且重要，那就认真地考虑在内部编码实现它；定制的解决方案可能更适合这个任务，导致更好的性能和更少的外部依赖。

9.5 把4GB放入4GB：这不起作用

在开发和设计JPC时我们碰到的很多问题都是与开销有关的问题。当设法在一个运算环境内部模拟一个完全的运算环境时，妨碍你获得100%本机性能的唯一的东西就是模拟

程序的开销。其中有些开销与时间有关，例如在处理器模拟程序中，而另外一些与空间相关。

由于空间的开销而导致问题的最明显的地方是在地址空间：虚拟计算机的4GB内存空间（32位地址）并不会放入实际（宿主）硬件中可用的4GB（或更小）内存空间中。即使拥有很大的宿主内存，我们也不能直接声明byte[] memory = new byte[4 * 1024 * 1024 * 1024];。我们必须以某种方式缩小模拟地址空间以放入宿主上的单个进程中，理论上有许多空间可以节省！

为了节省空间，我们首先观察这4GB地址空间是否总是不会装满。典型的机器不会超过2GB的物理RAM，而且，在大多数情况下都要少很多。所以，通过观察到不是所有的虚拟内存都会被物理RAM占用，我们可以非常快地把我们的4GB压下来。

设计我们的模拟物理地址空间中的第一步就有对未来的考虑。如果我们看一下它的发展道路，我们将看到IA-32存储器管理单元的特征之一有助于指导我们的地址空间的架构。在保护模式中，这个CPU的存储器管理单元把地址空间切成不可分的4KB块（众所周知的页）。所以，显而易见要做的事是按相同的比例分割内存。

把地址空间分成4KB的块意味着地址空间不再直接存储数据。数据改为存储在原子存储器单元（描绘成Memory的各种子类）中。地址空间然后保持这些对象的引用。最终的结构和内存存取方式如图9-2所示。

注意：为了优化instanceof的查找方式，我们为Memory对象设计了继承链而没有使用接口。

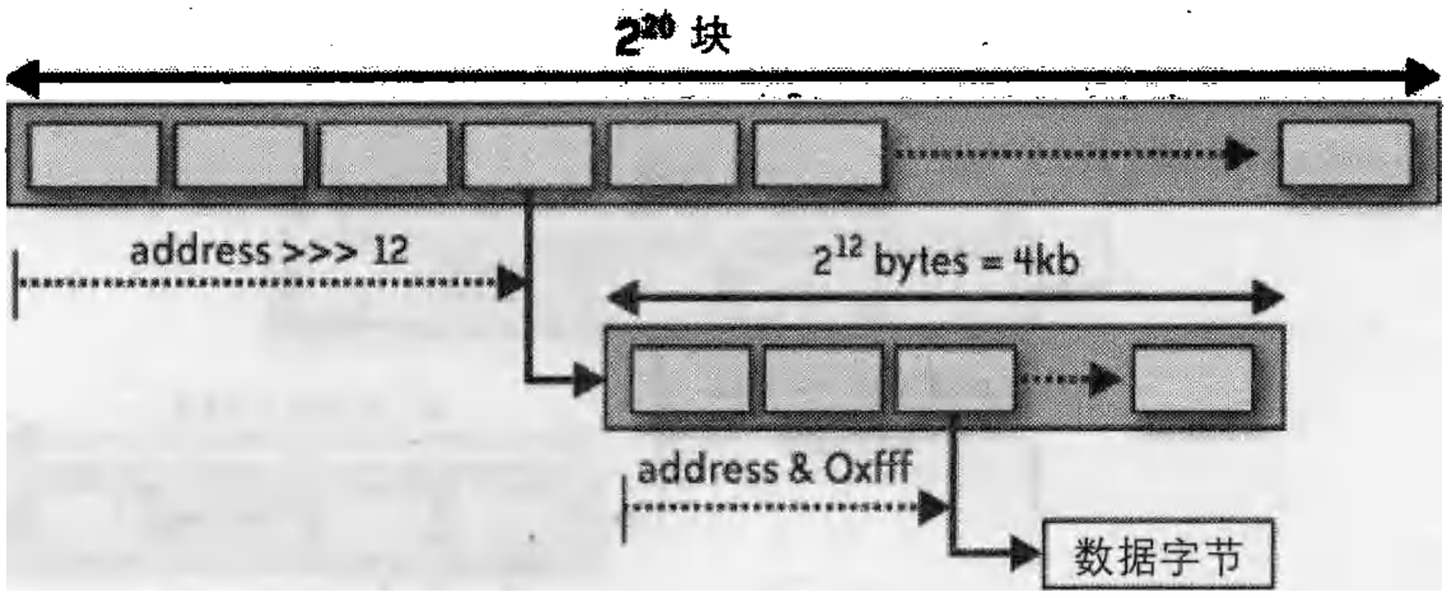


图9-2：物理地址空间块结构

这个结构有2^{20}个块，每个块都需要一个32位的引用来持有它。如果我们把这些引用放在一个数组中（最显而易见的选择），我们将要花掉4MB的内存，这在大多数情况下没有很大的影响。

技巧 #7 对类进行instanceof的操作更快

对一个类进行instanceof的操作比用接口来操作它快得多。Java的单继承模式意味着对于一个类，instanceof仅仅是一次减法和一次数组查找；对于一个接口，那是一次数组搜索。

在这点开销也成问题的地方，我们可以进行进一步优化。请注意，物理地址空间中的存储器分为三个不同的种类：

随机存取存储器（RAM）

物理的随机存取存储器从零地址向上映射。它被频繁地存取且等待时间很短。

只读存储器（ROM）

只读存储器芯片可以存在于任何地址。它们很少读取且等待时间很短。

I/O

I/O映射的存储器可以存在于任何地址。它被相当频繁地存取，但等待时间通常比随机存取存储器更长。

为了对那些落入实际机器的RAM中的信息进行寻址，我们使用一阶段查找。这确保对RAM存取的等待时间尽可能地低。为了存取其他地址（ROM芯片和I/O映射存储器占据的地址），我们使用两阶段查找，如图9-3所示。

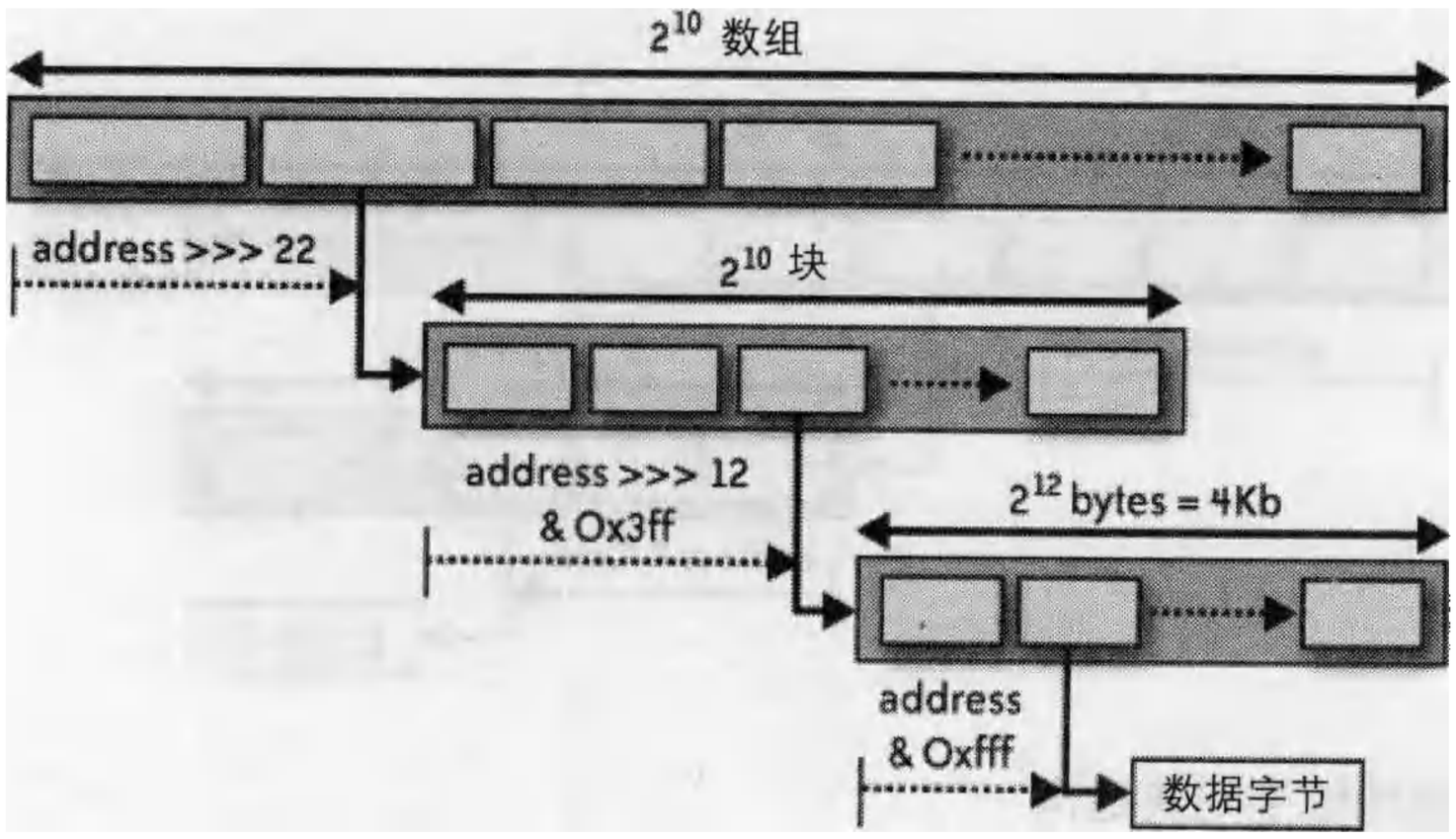


图9-3：两阶段查找的物理地址空间

现在，从RAM“get”一段内存有3个阶段：

```
return addressSpace.get(address);
    return blocks[i >>> 12].get(address & 0xfff);
        return memory[address];
```

从更高地址“get”有4个阶段：

```
return addressSpace.get(address);
    return blocks[i >>> 22][(i >>> 12) & 0x3ff].get(address & 0xfff);
        return memory[address];
```

这个两层优化节省了内存，同时避免在每次RAM存储器存取时形成瓶颈。在一个存储器“get”中的每次调用和间接放置都执行一个函数。确实应该使用的间接方式——不是为了交互，而是为了在性能和复杂度之间达到最好的平衡。

JPC在有可能用不到存储的任何地方还使用了延迟初始化。因此，一个新的JPC实例拥有一个映射到不占据空间的Memory对象的物理地址空间。当RAM的一个4KB区域第一次读或写时，这个对象将完全初始化，如例9-1所示。

例9-1：延迟初始化

```
public byte getByte(int offset)
{
    try {
        return buffer[offset];
    } catch (NullPointerException e) {
        buffer = new byte[size];
        return buffer[offset];
    }
}
```

9.6 保护模式的危险

保护模式的到来带来了一个完整的存储器管理系统，同时在物理地址空间之上又增加了一个复杂的层。在保护模式中，内存分页可以是激活的，这容许物理地址空间的4KB块进行重新排列。这种重新排列由保存在存储器中的、可以由运行在这机器上的代码动态修改的一系列表进行控制。图9-4演示了一个完整的分页转换所经过的路线。

大体上，机器上的每次内存存取都需要这样一个完整的查找次序：通过内存分页结构找到特定线性地址映射到的物理地址。由于这个过程太复杂且开销很大，实际的机器会把这些查找的结果缓存在转换后备缓冲（TLB）中。除了这些增加的间接层之外，内存分页还有额外的保护特征。每个映射的分页都可以给定一个用户或管理员和读或读/写的状态。没有足够的权限而试图存取分页的代码，或设法存取不存在分页的代码会引起一个在某种意义上类似于一个软件中断的处理器异常。

为了优化这样的结构，我们的首要战略应该是采用实际处理器的途径，这非常清楚；换句话说，我们必须拥有某种形式的查找缓存。为了设计它，我们做出了两个关键的选择：

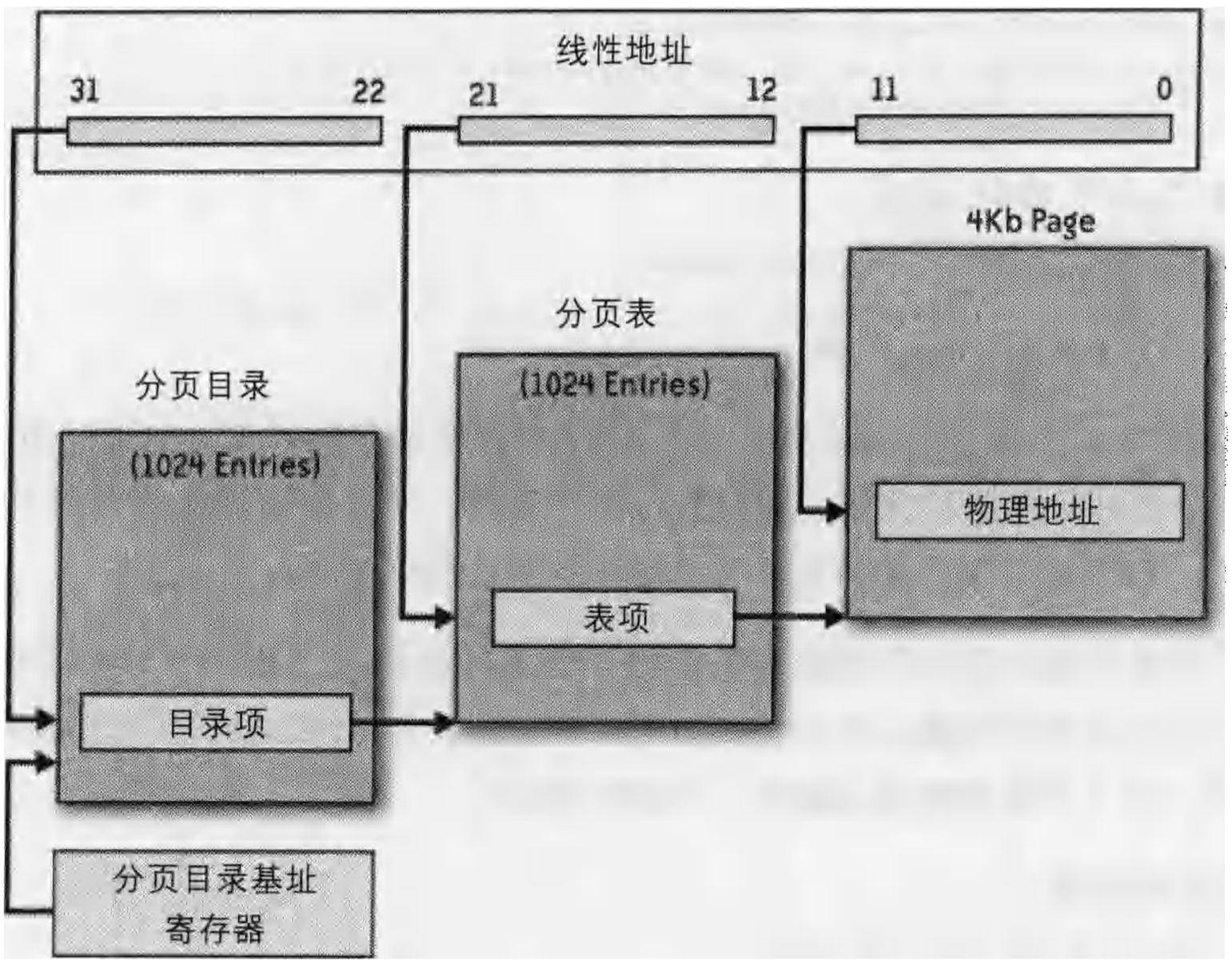


图9-4：IA32架构的分页机制（仅仅4KB页）

- 分页的重新映射以4KB的粒度进行。不是巧合，而是为了方便，这也是我们为物理地址空间选的粒度。
- 当一个保护模式进程存取内存时，它只看这些4KB块的重新映射图（虽然其中的一些块引起处理器异常）。物理地址空间仅仅是最初的Memory对象的一种可能的顺序（在那里，所有的对象都同时按地址顺序排列），与其他的任何顺序相比，没有特别的意义。

根据这两个选择，我们明白缓存（例如，TLB）的最自然的形式是复制物理地址空间结构。内存分页依照lookup分页表确定的新的顺序进行映射。第一次请求一个给定表内的一个地址时，会完整遍历这个表的结构以找到匹配的物理地址空间内存对象。然后，对这个对象的一个引用会放入线性地址空间内的正确位置，从那时起它就一直缓存着，直到我们选择清除它。

为了解决读/写和用户/管理员的问题，我们做了一个战术性的决定，为了速度而牺牲一些内存开销。我们为每个组合（读-用户、读-管理员、写-用户和写-管理员）制造一个线性地址空间。内存“get”使用读标记，“set”使用写标记。因此，在用户模式和管理员模式之间的转换只需要改变内存系统中的两个引用，如图9-5所示。

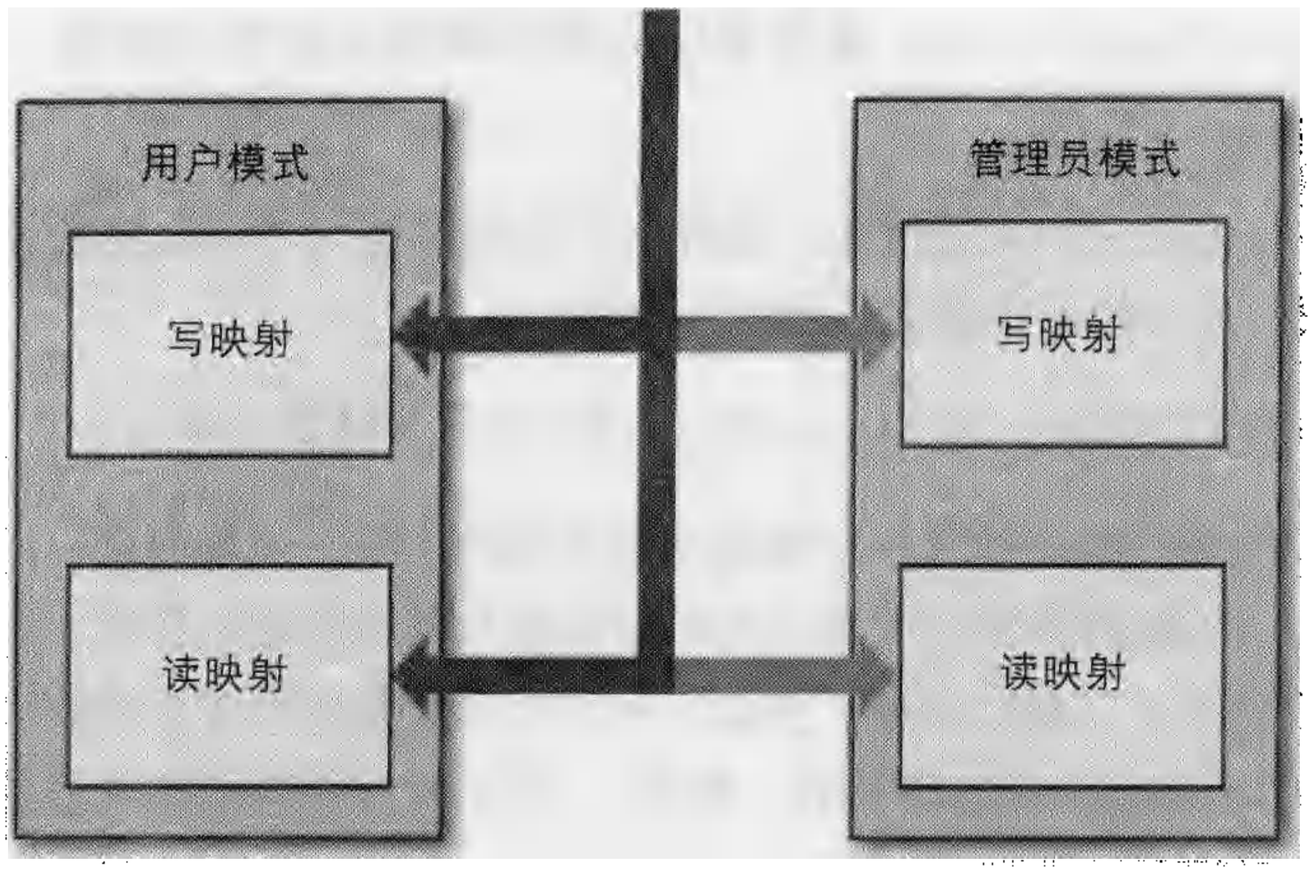


图9-5：线性地址空间中的保护级别切换

注意：这非常适合Linux内核的分页方法。在Linux中，每个用户模式进程都映射到它的线性地址空间的内核分页。在硬件中，这防止了日常维护操作或系统调用在转移到内核代码时的上下文切换。我们到内核空间的切换仅仅是数组引用的翻转，这也是一个低开销的进程。

这还给我们留下页错误和保护系统侵犯的问题。我们处理这些情形的态度在某些方面就是我们判断是否是异常的指示，既在低级别的处理器方面也在Java语言中。这个态度可以总结如下：

Exception是异常，Error是错误。

详细地说：Exception应该代表极少或例外的情形，但不一定是毁灭性的，Error应该代表毁灭性的或相当严重的不期望的情形。

就Java语言中异常处理实现的程度而言，许多程序员不愿意使用对他们有利的异常处理机制，这有些奇怪。对软件来说，异常代表了处理罕见但也可以纠正的情形的最有效和简捷的方式。抛出一个异常会由于有效地优化常规路线而使执行开销有偏向，同时会增加处理不可避免的异常情形的开销。这样说来，页错误和保护系统侵犯是一个异常的但应该映射到一个Exception实例的正常事情，这是显而易见的。

技巧 #5：Exception用于异常的情况

Exception应该用于异常条件，不要用于错误。把异常用作为非正常环境中的流控制会提示VM优化正常的路径，给你带来更好的性能。

在处理页错误和异常时，对异常的使用，我们有两个比较常用的惯例：

- ProcessorException，是代表Intel处理器所有异常类型的一个类，继承了RuntimeException。
- 页错误，轻度的保护系统侵犯，是异常的情形，但为了控制牺牲性能，它们非常普通，我们使用静态的异常实例抛出它们。

这两个决定都有很好的理由，但很大一部分与JPC项目的特性无关。

决定选择继承RuntimeException主要是考虑代码的美观。一般而言，运行时异常用于不能捕获或不应该处理的异常。大部分材料推荐说如果抛出这类异常，它们允许传播并引起抛异常的线程终止。我们知道，在运行时异常和检查的异常之间的区别是类似于泛型、自动封装或for-each循环的代码“糖果”。倒不是我们想要诋毁这些结构，而是发现异常的分类对编译后的代码没有影响，我们可以安全地选择最方便的类型而不会引起任何性能瓶颈。在我们的项目中，ProcessorException是一个运行时异常以避免必须在很多方法中声明throws ProcessorException。

为页错误和系统保护侵犯抛出静态已初始化的异常实例（高效单例异常），这在异常处理链的两端都产生好处。首先，我们自己节省了在每次抛出异常时初始化一个新对象的开销。这是避免了重新构建调用线程的栈跟踪的开销（完全不像在现在的JVM中那么微不足道）。其次，我们自己节省了判断thrown类型的开销，由于有了有限的一组静态异常来判断thrown类型，我们只需进行一系列引用比较。

9.7 从事一项毫无成功希望的斗争

IA-32架构中没有什么比指令集更流行。经过这么多年，曾经在8080时代基于累加器的简单架构已经成为一个非常巨大和复杂的指令组。IA-32已经成为一个拥有无数固定扩展和许多寻址模式的类似于RISC的芯片。

当作为Java开发人员看到这样的前景展望时，回复到打字和开始编写类，这非常诱人，就好像编的结构越多，问题就越简单。如果我们正在开发一个分拆器，那这种方式即使不完美，也会很好。在这种创建非常多的对象的系统中，不可避免也会有许多的垃圾收集。

这导致双重的速度瓶颈。我们不仅承受大量的对象分配的开销，还承受频繁的垃圾收集。在一个现代的按代区分的垃圾收集的环境中（Sun JVM就是这样的），小的、短生命周期的对象都创建于初生代，而几乎所有的初生代收集算法都是阻塞一切的。所以，一个有大量对象紊流的译码器不仅承受不必要的对象分配的可怜性能，还承受收集器清除所有对象紊流时很多非常短暂的GC暂停。

因为这个原因，降低推动解析执行的译码器内的对象紊流显得非常重要。在真实模式的译码器中，这种最低限度的方式导致一个6500行代码的类只有42个“new”关键字实例：

- 在类载入时间有4个new Boolean[] (static final)。
- 用于一个旋转缓冲的3个new Operation()。
- 用于Operation中扩展缓冲的2个new int[]。
- 用于异常情形的33个new IllegalStateException()。

一旦创建这个译码器的一个实例，唯一必须构建的对象是用于Operation中int[]缓冲的扩展。一旦这些缓冲扩展完成，就不再有对象构建和垃圾收集，因此，那是较少暂停的译码。

译码器的设计阐述了我们考虑的编程（在这个情形中，是Java的）重要原则之一：

仅仅因为你可以这样做，并不意味着你应该这样做。

在这个情形中，仅仅因为JVM可以进行自动垃圾收集，并不意味着将迫使你去运用它。在代码的关键性能部分，慎重地使用对象实例。小心处理像Iterator、String和varargs调用这样的类的不活跃对象实例。

技巧 #1：创建对象不好

过多的对象实例（尤其是短期的对象）会导致不好的性能。这是因为大量对象引起频繁的年轻代（young generation）垃圾收集，而初生代垃圾收集算法几乎是“全部停止”类型的。

微码：少即是多或多即是少

我们已经有了一个用于解析IA-32指令流的GC较少的译码器，但是，我们还没有讨论这样的一个译码器应该如何译码。IA-32架构不是一个固定长度的指令系统，指令长度的范围从单个字节到最多15个字节。指令集的许多复杂性源自于可以用于一个指令的任意特定操作数的内存-寻址模式过多。

初始最高级别的分解把每个运算分为4个阶段：

输入操作数

从寄存器或内存中载入运算的数据。

运算

对输入操作数进行数据处理。

输出操作数

把运算的结果输出到寄存器或内存。

标记运算

调整标记寄存器的位来表示运算的结果。

对操作数和运算的这种分解使我们能够把运算的简单性和它的操作数的复杂性分离开。像add eax,[es:ecx*4+ebx+8]这样的一个运算最初分解为5个运算：

```
load eax
load [es:ecx*4+ebx+8]
add
store eax
updateflags
```

立即就可以清楚地知道Load [es:ecx*4+ebx+8]完全不是一个简单的运算，它可以容易地分解到很多更小的元素。实际上，仅对于这个寻址格式就有：

- 6个可能的内存段
- 8个可能的索引寄存器
- 8个可能的基址寄存器

仅对于这个寻址模式就产生了384种可能的组合。显然，对这些地址计算还需要更多的分解。因此，把这种类型的内存存取进一步分解，直到我们得到：

```
load eax
memoryreset
load segment es
inc address ecx*4
inc address ebx
inc address imm 8
load [segment:address]
add
store eax
updateflags
```

为了生成可执行的模拟指令集，我们必须平衡这两方面的优先级：

- 首先，我们必须平衡译码时间和执行时间以优化整体的执行速度。我们必须记住，在解析处理器中，我们主要的目标是起始执行的响应时间短。常规执行的代码应该在随后的优化阶段处理。在这里，我们最初的目标是使代码编译出来而不要阻塞整个模拟器。随后的优化可以异步地进行，这里花的时间延迟了一切。所以，我们一直在寻找一个可以快速译码的相对简单的指令集。

- 其次，我们必须平衡指令集的大小和“编译后”代码的长度。小的指令集自然会产生冗长的代码，而较大的指令集应该会使代码更紧凑。我们说“应该”是因为如果选择不好，大的指令集仍然需要较长的区域。一个较小指令集的解析器在代码和记录日志上会更小，因此，执行它的每个运算的速度会快很多，但是，相对地，它有更多的指令集要执行。所以，我们一直在指令集大小和代码长度之间寻找一个合理的平衡以使解析器达到接近最优的性能。

在寻找这两个方面的最佳平衡点时，牢记Hoare的格言（注5）非常重要：

> 过早的优化是所有不幸的根源。
>
> ——C. A. R. Hoare

这些优化的精确平衡是由系统决定的。在一个Java环境中，系统不仅包括物理硬件，还包括JVM。加上在这个环境中的Java组件总是即时编译这个因素，小范围的性能基准众所周知是不可靠的。说到底，我们不要过度使用这样的基准来指导编码选择，只有当性能出现大的改变时依赖基本原理和信任的基准。在一个即时编译的环境中，最好的情况下，在这些微小基准中的小改变在一个单独的系统中不可重复。在最坏的情况下，它们如此依赖基准场景以致在同一个系统上都不一定可重复。

注意：微代码集的一个重要特征是为它设置的整数值常量是连续的。这个解析器的核心是对这个常量集的一个switch语句，我们必须确保这个switch语句运行得尽可能快。

牢记了这些因素，在几次试验后，我们推断对于全部整数或浮点的模拟程序而言，750行左右的代码集表现出很好的平衡效果。这给我们从x86运算到微代码一个近似10的转换因数。虽然这个集合似乎有点大，但是，它解析的速度很快，而且运算也相当小。这使得它们可以成为供给随后的优化阶段的很好的候选。

技巧 #3：表切换好，查找切换不好

标签集适度紧凑的switch语句比标签集的值更分散的switch语句要快。这是因为Java对switch有两个字节码：`tableswitch`和`lookupswitch`。表切换是利用一个间接调用执行的，switch的值把偏移量提供到一个函数表中。查找切换要慢得多，因为它们按对应关系进行查找以发现一个匹配的值：函数对。

注5： *http://en.wikiquote.org/wiki/C._A._R._Hoare.*

9.8 劫持JVM

“Java很慢”的说法至今仍然困扰着Java开发人员。很多这样的评论来自于非Java开发人员对20世纪90年代中后期的JVM的体验。自那以后，使用Java的人们知道它已经获得了大幅的改进。在这改进背后的驱动力也是加速JPC的关键：常规Java进程的环境可以非常简单地划分为程序区域和数据区域。

在图9-6中，我们可以看到数据区域可以进一步划分为静态数据（在编译期间就可以知道）和动态数据（在编译期间不能知道）。Java环境中大多数的静态数据是从类路径载入的类字节。虽然这些类字节作为数据载入，但它们实际上代表的是代码，这非常清楚，它们会在运行时由JVM进行解析。所以，很明显我们愿意把这些类字节移动到这张图的另一边。

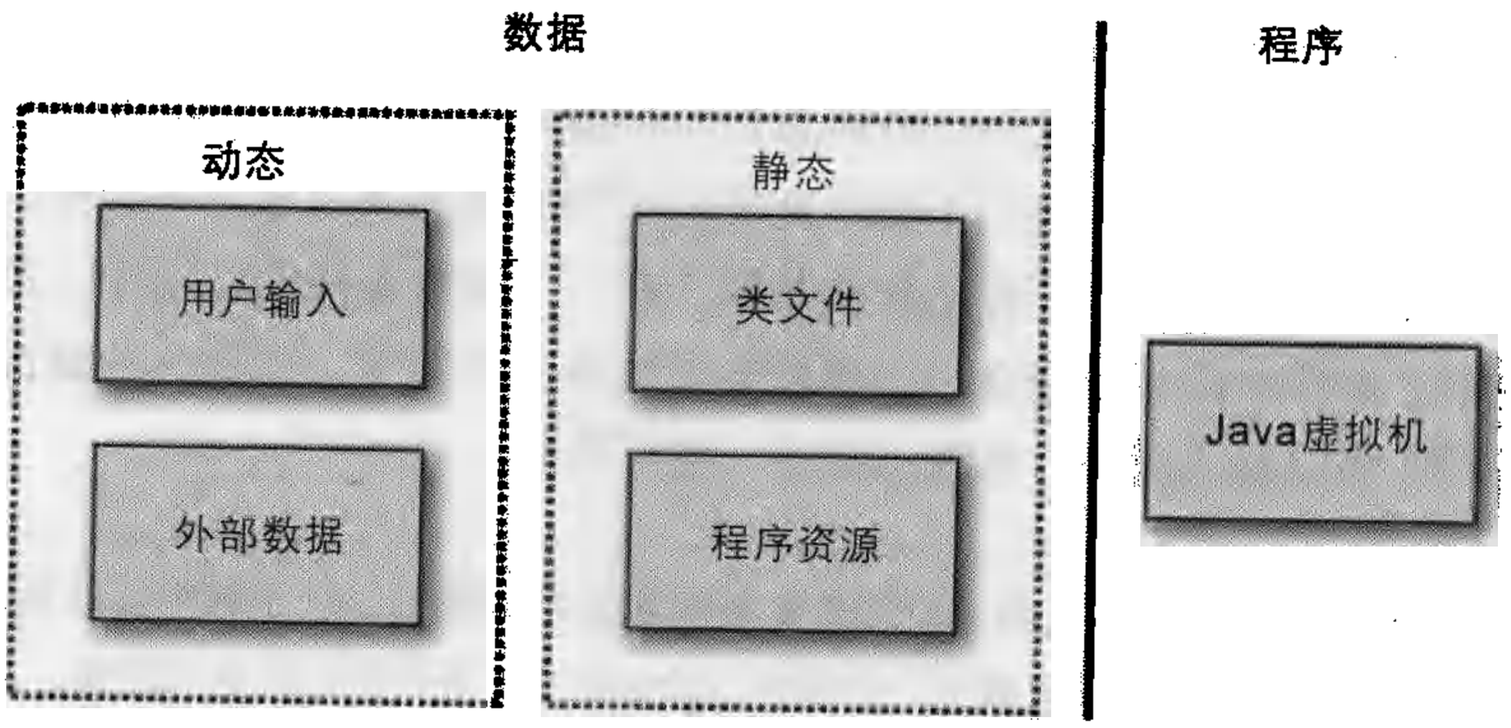


图9-6：一个Java进程中的程序区域和数据区域

在一个即时编译的环境中，例如Sun HotSpot，常用的字节码区域分解析或动态编译到宿主的本地指令集。这样就把类字节从数据区域移入了代码区域。然后，这些类就像本地代码一样执行，并把程序加速到本地的速度（参见图9-7）。

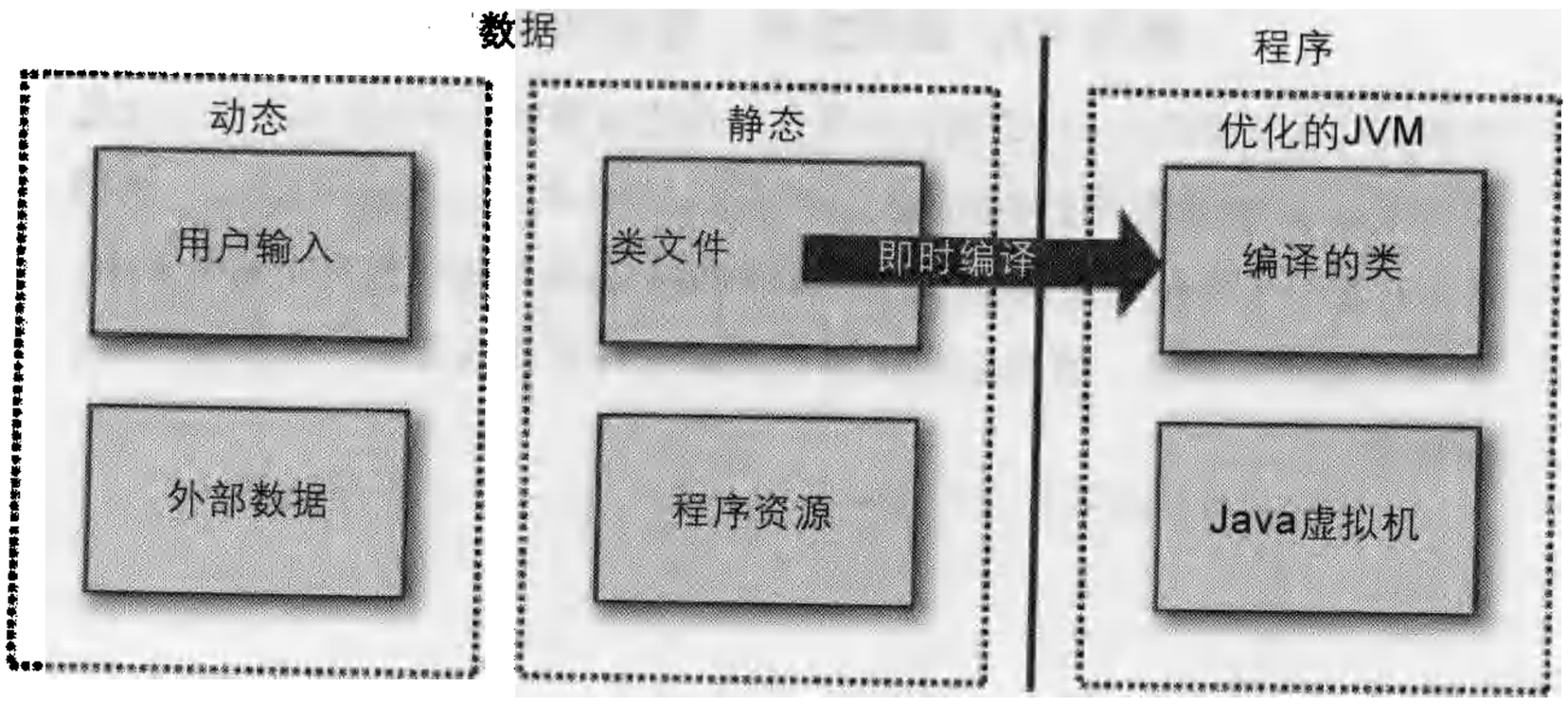


图9-7：Java环境中的即时编译

在JPC中，我们利用这个事实：在JVM启动的时候不必知道所有的类数据。实际上，用Java的说法，“静态数据”作为“最终数据”来引用比较好。当一个类加载时，它的类字节将固定并不能修改（为了方便，让我们忽略JVM TI（注6））。这允许我们在运行时定义新的类，那些使用插件架构、Applet或J2EE Web容器的人会立即熟悉这个概念。

接下来我们重复一下JVM执行的即时编译技巧，但仅仅在JPC的级别。对于JPC，在Java运行时环境的范围内我们有两层的程序-数据信息划分。现在，我们的编译有两个阶段：

1. 在JPC内，IA-32机器代码在需要时会编译成字节码。这些x86块由此成为了可以由JVM加载的有效的Java类文件。

2. 类由JVM编译成本地代码。由于JVM并不区分由手写代码组成的原始的“静态”类和自动构建的动态类，两种类型都尽可能地获得最佳的本地性能。

在这两个编译阶段后，我们最初的IA-32机器代码已经翻译成宿主的架构（参见图9-8）。幸运的是，新指令的数量与最初的数量相比没有明显增多，这意味着性能不会比本机慢很多。

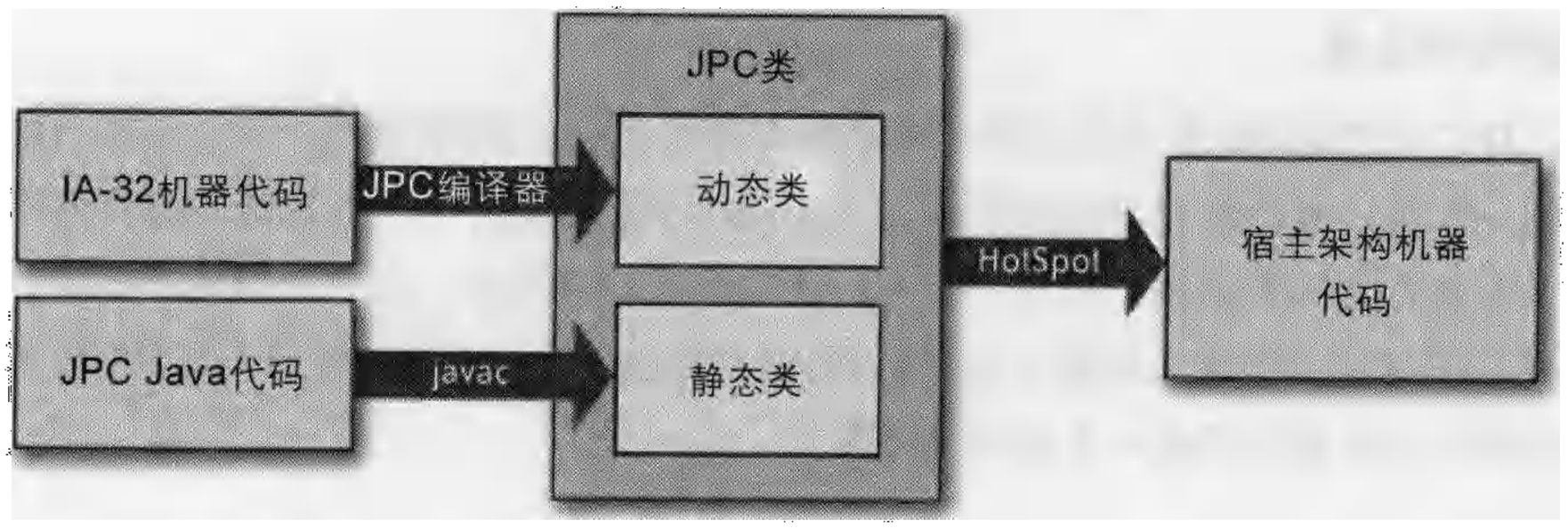


图9-8：JPC架构中的三个编译器

目前我已经知道如何使我们的模拟程序速度更快，但是，我们稍微掩盖了一些详细信息。这些详细信息（我们的新问题）现在落入两个不同的区域：我们如何进行这样的编译，以及我们应该如何载入最终的类？

9.8.1 编译：如何多此一举

我们这里讲述的编译器不是最理想的例子，但是，我们所处的情况也略有些不平常。javac和JPC编译器都仅是第一阶段的编译器，这意味着它们仅仅把它们的输出提供给第二阶段，一个字节码解析程序或一个即时编译器。我们知道javac几乎不优化它的输出；javac生成的字节码只是输入的Java代码的一个翻译（参见例9-2）。

注6：对于那些喜欢摆弄JVM顽皮代码的人来说，位于*http://java. sun.com/ javase/6 /docs/technotes/guides/jvmti/*的*Tool Interface*可以用来做许多非常有趣的事，包括重新定义类文件。

例9-2：Javac的未优化的编译

```
public int function()
{
    boolean a = true;
    if (a)
        return 1;
    else
        return 0;
}
```

```
public int function();
   0:   iconst_1
   1:   istore_1
   2:   iload_1
   3:   ifeq    8
   6:   iconst_1
   7:   ireturn
   8:   iconst_0
   9:   ireturn
```

很容易证明进行最小限度的优化是恰当的，因为大部分的优化工作都留给编译的后续阶段来进行。在JPC中，我们不仅有理由，也有额外的时间压力：

- 我们想要使编译的开销最小化以避免占用其他模拟程序线程的CPU时间。
- 我们想要使编译的延迟时间最短以便解析的类可以尽快地替换。一个高延迟的编译器将发现当编译完成一个类时它的代码已经不再需要了。

简单的代码生成

现在，JPC中的编译任务就是把单个解析的基础块的微代码转换到一组Java字节码这么简单的一件事。我们最有可能会假设这个基础块不能抛出异常。这意味着这样的基础块是一个严格定义的基础块，只有一个入口和一个出口。现在，由一个特定基础块改进的每个变量都可以由包含一组输入寄存器和内存状态的一个函数来表示。在JPC内部，我们共同把这些函数表示成一个有向无环图。

图的源点（source）

源点表示以寄存器值的形式或直接以指令的形式的输入数据。

图的汇点（sink）

汇点表示以寄存器值和内存的形式的输出数据或写I/O端口的输出数据。对于受到块影响的每个状态变量，都有一个汇点。

图的边界（edge）

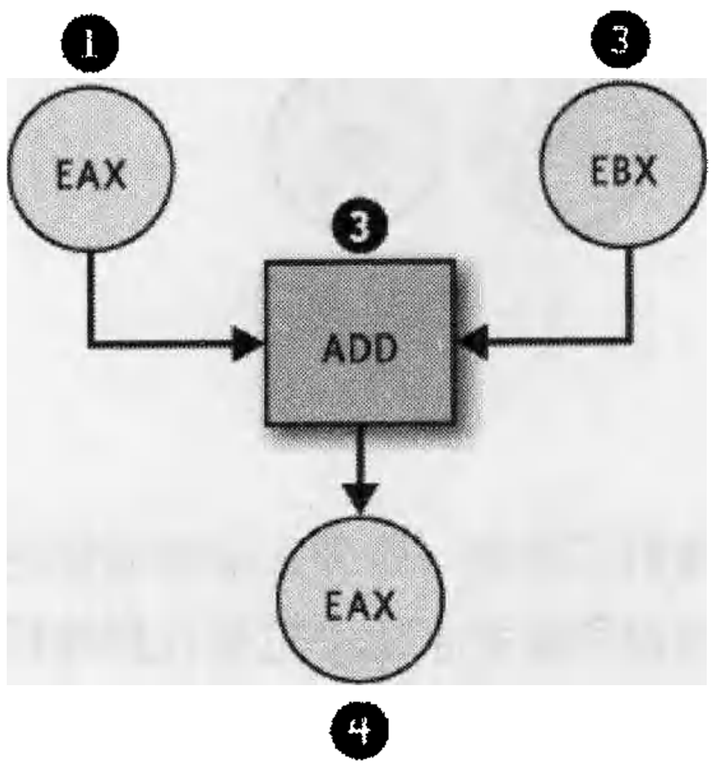
边界表示变量值在图的内部传播。

图的节点（node）

节点表示对输入边界的运算，其结果会沿着输出边界传播。在JPC中，这些运算是对已解析微代码的单态修正的组成部分。因此，如果微代码影响多个变量，它就会映射到多个图节点。

把已解析基础块的这种图形表示法转换到Java字节码，只是图形上每个汇点依次简单的最浅层的移动。图中的每个节点把栈中最顶端的元素作为输入，然后把它的结果放在最顶端，准备由任意子节点进行处理(参见图9-9)。

注意： 图的经典的最佳移动顺序需要每个节点上的优先权都应该按深度的顺序进行计算。距离最远源的路线最长的节点应该最先计算，路线最短的节点最后计算。在一个基于寄存器的目标上，这会生成最短的代码，因为它消除了大多数的寄存器修改。在一个基于栈的机器上，例如JVM，同样的决策过程会引起编码的栈深度最小。在JPC中，我们忽略这些细节并依靠JVM消除其中的差异，仅追踪所有节点的深度这个额外的复杂情况就不值得这么做。



	代码	栈
1	load eax	eax
2	load eax load ebx	ebx eax
3	load eax load ebx add	ebx+eax
4	load eax load ebx add store eax	<empty>

图9-9：把x86运算表示成一个有向无环图

这种汇点接着汇点、最浅层分析的方式引起图的自然优化，如图9-10所示。对应于无用代码的图的孤立部分不能通过汇点访问，所以会自动删除，因为解析永远都不会达到那儿。在图的解析中，重用的代码区域会发现自身计算了多次。我们可以把它们的结果缓存到一个本地变量中，在对这个节点进行后续访问时载入这个结果，由此避免了所有的代码重复。

与一棵树中的节点相关联的代码表示成由源编译成的部分JPC代码的单个静态函数。那样，我们生成的代码只是对处理器变量的一系列推动并直接压入栈，紧接着是对每个节点的一系列静态调用，最后是取出成一系列的处理器对象。

技巧 #2：静态好

如果一个方法可以定义成静态的，那就定义成静态的。静态方法不是虚的，所以，不会动态分派。与实例方法相比，高级的VM更容易也更愿意内联静态的方法。

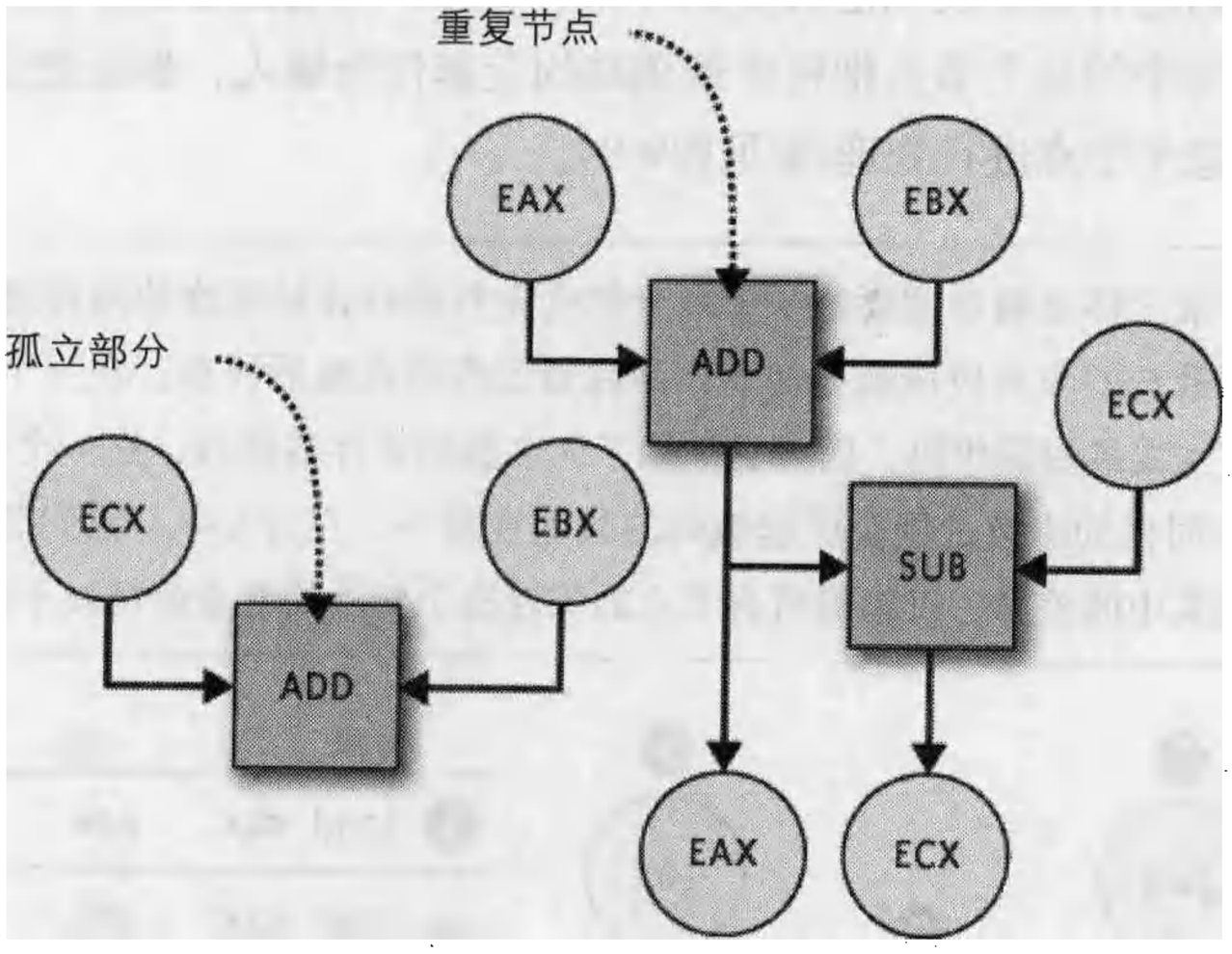


图9-10：有向无环图的特性

处理异常

既然我们可以处理基础块的编译，那么，我们必须把一些异常情况恢复正常。正如我们先前讨论的那样，异常并不总是错误。页错误和保护系统侵犯很自然地任意抛出。当异常抛出时，处理器的状态必须和最后成功执行的那个操作一致。这对于编译器来说显然有相当重要的含义。把IA-32中的异常映射到Java异常，我们知道这个问题的唯一可用的解决办法是在异常处理器中捕获这个异常，然后确保处理器的状态与最后成功的操作一致。

任意特定基础块内的异常行为路线就像基础块本身的路线。它有一个入口和一个出口，和主路线唯一的区别是出口。因为异常的路线和基础块本身的路线的区别不是很大，描绘它的最自然的方式是在它自己的有向非循环图内。异常路线的图将和基础块共享相同的一组节点，但是，它另外拥有一组映射到它自己不同的出口的汇点。

然后，异常处理器的代码通过遍历选定的路线图而生成了。当异常抛出时，与主路线共享的节点重新设置回主路线遍历它们时的原来状态。这意味着从主路线来的所有缓存和计算值都可以在异常处理器中重用，因此避免了重复任何工作。

字节码处理

已经有许多已建立和设计良好的解决方案来把我们已编译的字节码转变成可载入的类。Apache的字节码工程库（Byte Code Engineering Library，BCEL）自身定位于便利地

“分析、创建和处理（二进制的）Java类文件”。（注7）ASM是一个“完全致力于Java字节码处理和分析的框架”。（注8）

遗憾的是，我们要做的只是修改单个框架类中的单个方法（总是修改同一个方法）。我们只能生成一小部分可能的字节码序列，不必提供任何分析工具。这样看来，BCEL和ASM对我们的要求来说有些重量级了。我们改为开发一个只有非常有限能力的完全符合我们需要的定制的字节码处理库。例如，我们的栈深度算法调整为快速确定方法的最大栈深度（以便它们可以通过验证）。虽然这个算法不适用于常规的类编译，但是，对于我们的用途来说，它足够了，而且更有效。

技巧 #9：小心外部的库

避免使用超出你的目的的外部库。如果任务简单而且重要，那就认真地考虑在内部编码实现它；定制的解决方案可能更适合这个任务，导致更好的性能和更少的外部依赖。

9.8.2 大规模的类加载和卸载

既然有了类，那么，它们必须进行加载。第一个显而易见的问题是“有多少？”图9-11给了我们关于这个问题的一个说明。可以证实把100 000个类加载到单个JVM中只是一个微不足道的挑战。

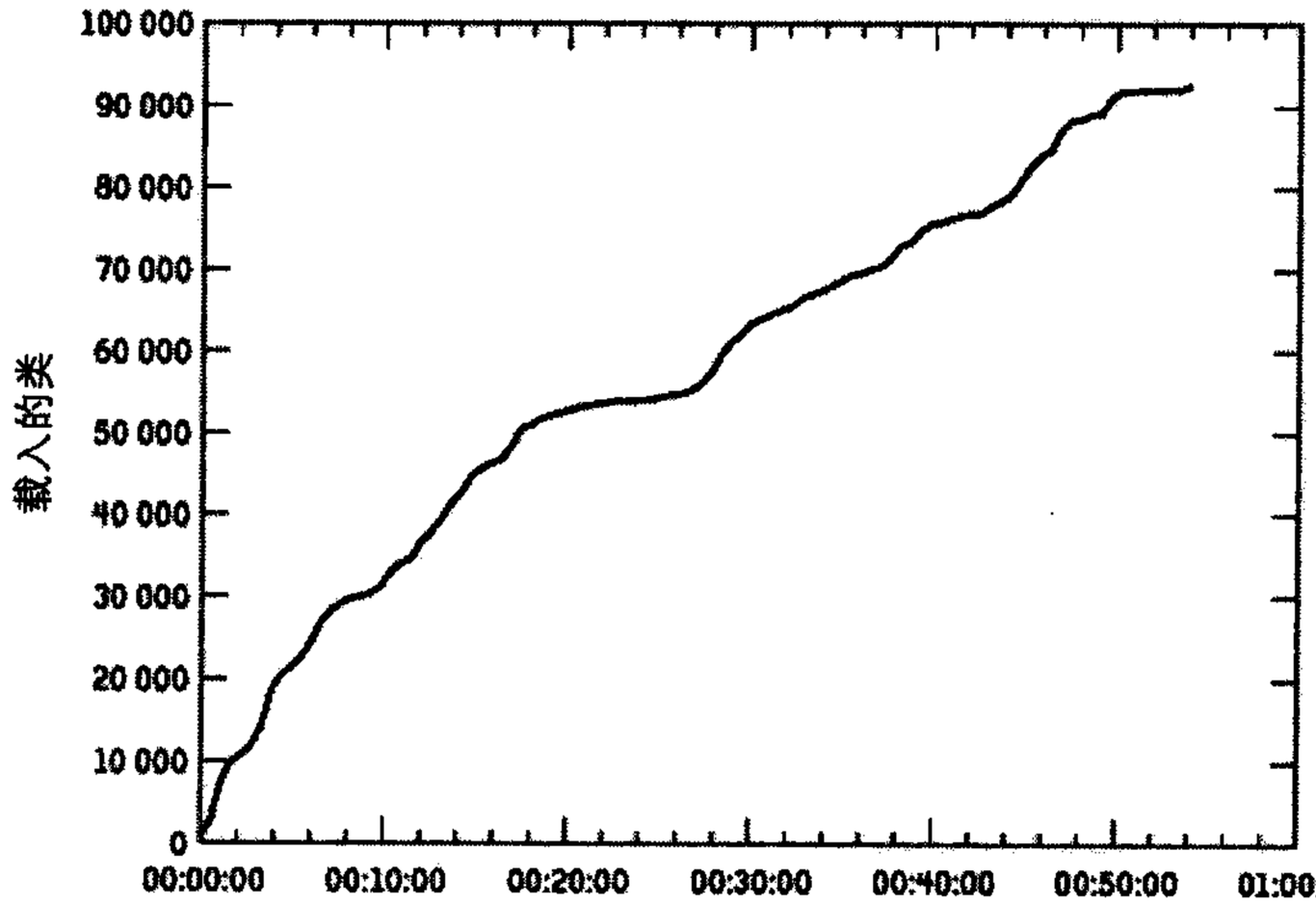


图9-11：一个现代的GNU/Linux启动期间的类的数量

注7：*http://jakarta.apache.org/bcel.*

注8：*http://asm.objectweb.org/.*

因此，我们必须为所有的这些类寻找空间。类文件存储在JVM中一个位于正常对象堆（即众所周知的“永久区域（Permanent Generation）”）之外的特定的内存区域。在典型的Sun JVM中，永久区域初始有16MB，最多可以增加到64MB。很显然，100 000个类不能全部载入64MB的堆。我们有两个方法可以解决这个问题。

第一个办法是用命令*java -XX:MaxPermSize=128m*。虽然这个办法比较笨拙，但它确实有助于解决这个问题。遗憾的是，这不是根本的解决办法，因为我们所做的只是延缓问题的发生。最终，我们会加载足够的类来填满新的空间，我们不能一直增加空间。

后一种解决办法涉及减少加载类的数量。实际上，垃圾收集器不应该清除所有不用的类吗？对于垃圾收集器来说，类和堆的对象没有实质性的区别。只要一个类没有活动的引用指向它就可以垃圾收集掉（卸载）。当然，类的每个实例都对它这个类型的Class对象保持了一个强引用。所以，对于一个要收集的类，首先必须不存在这个类的活动实例；其次，没有这个类的其他的引用。因此，如果我们定义如例9-3所示的定制类加载器，那或许就可以解决这个问题。

例9-3：简单的不保持引用的ClassLoader

```
public class CustomClassLoader extends Classloader
{
    public Class createClass(String name, byte[] classBytes)
    {
        return defineClass(name, classBytes, 0, classBytes.length);
    }

    @Override
    protected Class findClass(String name) throws ClassNotFoundException
    {
        throw new ClassNotFoundException(name);
    }
}
```

例9-3中的ClassLoader不保持它定义的类的引用。每个类都是没有联系的单例，所以，findClass可以安全地抛出ClassNotFoundException而其他都保持正常。一旦某个类的单例成为GC的候选，那这个单例和这个类本身都可以收集掉。这运行得很好，所有的类都加载了。然而，某个地方出错了。由于未知的原因，永远没有类卸载掉。似乎在某处有一个对象对我们的类一直保持着引用。

让我们看一下定义一个新类的调用清单：

```
java.lang.ClassLoader: defineClass(...)
java.lang.ClassLoader: defineClass1(...)
ClassLoader.c: Java_java_lang_ClassLoader_defineClass1(...)
vm/prims/jvm.cpp: JVM_DefineClassWithSource(...)
vm/prims/jvm.cpp: jvm_define_class_common(...)
vm/memory/systemDictionary.cpp: SystemDictionary::resolve_from_stream(...)
vm/memory/systemDictionary.cpp: SystemDictionary::define_instance_class(...)
```

当你不满足于“因为事实就是这样”的回答时，就会发生这种情况。我们深入到JVM的内部，找到了这一小段代码：

```
// Register class just loaded with class loader (placed in Vector)
// Note we do this before updating the dictionary, as this can
// fail with an OutOfMemoryError (if it does, we will *not* put this
// class in the dictionary and will not update the class hierarchy).
if (k->class_loader() != NULL) {
  methodHandle m(THREAD, Universe::loader_addClass_method());
  JavaValue result(T_VOID);
  JavaCallArguments args(class_loader_h);
  args.push_oop(Handle(THREAD, k->java_mirror()));
  JavaCalls::call(&result, m, &args, CHECK);
}
```

这是Java级的对正在处理加载的classloader实例的一个回调：

```
// The classes loaded by this class loader.  The only purpose of this table
// is to keep the classes from being GC'ed until the loader is GC'ed.
private Vector classes = new Vector();

// Invoked by the VM to record every loaded class with this loader.
void addClass(Class c) {
    classes.addElement(c);
}
```

现在我们知道是这个淘气的小对象一直保持着对所有类的引用，从而阻止了对它们的垃圾收集。遗憾的是，对于这一点，我们无能为力。好吧，在不违背基本原理的情况下我们可以做的就是在java.lang包中声明一个类。我们现在知道我们的超类将一直有用地为我们保持着引用。这对于类卸载来说意味着什么呢？

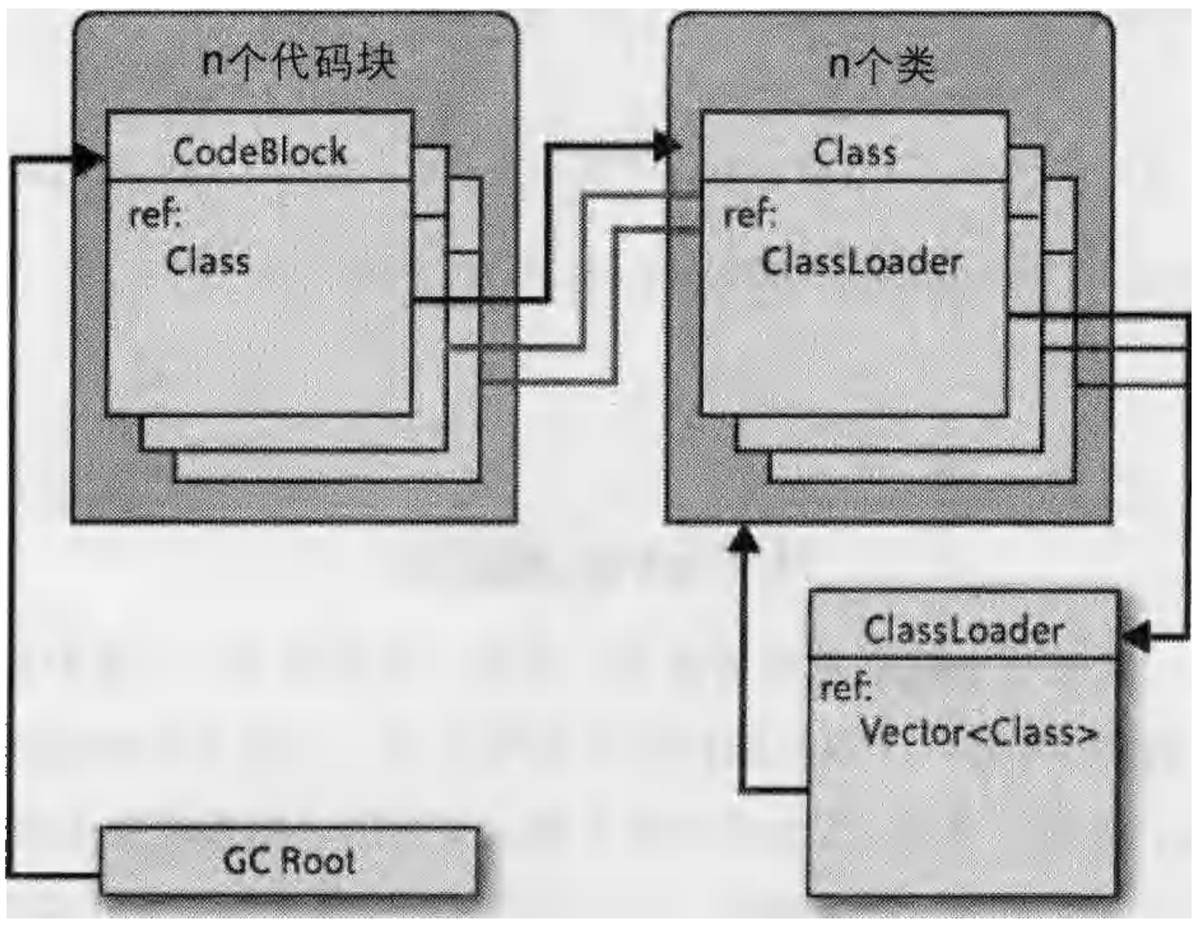


图9-12：类的GC根路径

在图9-12中的GC根路径中，我们可以看到，在由一个classloader加载的所有类的所有实例都成为GC候选之前，所有的类都会保持加载。所以，一个活动的代码块可以阻止*n*个类卸载。这一点都不好。

有一个简单的办法可以缓解这个问题。没有人说我们必须让*n*无限变大。如果我们限制任意加载器加载类的数量，那么，我们就可以减少类相互制约的机会。在JPC中，我们有一个默认只加载10个类的定制类加载器。加载第十个类会触发构建一个新加载器，它会用于下10个类，等等。这个方法意味着任意一个类都只能制约其他10个类（参见例9-4）。

例9-4：JPC中的ClassLoader实现

```
private static void newClassLoader()
{
    currentClassLoader = new CustomClassLoader();
}

private static class CustomClassLoader extends ClassLoader
{
    private int classesCount;

    public CustomClassLoader()
    {
        super(CustomClassLoader.class.getClassLoader());
    }

    public Class createClass(String name, byte[] b)
    {
        if (++classesCount == CLASSES_PER_LOADER)
            newClassLoader();

        return defineClass(name, b, 0, b.length);
    }

    protected Class findClass(String name) throws ClassNotFoundException
    {
        throw new ClassNotFoundException(name);
    }
}
```

HotSpot代码缓存

在JPC中，当运行加载大量类的复杂工作时，会遇到另一个内存限制。在Sun HotSpot JVM中，即时编译的代码保存在一个名为代码缓存的非堆（nonheap）区域。不仅JPC会生成许多类，其中的一些类是为HotSpot准备的，这也很不寻常。这意味着HotSpot的缓存会很快填满。这正好意味着增加永久保存区的大小也要增加代码缓存的大小。

CodeBlock替换

现在我们拥有一个已编译、已加载和已实例化的定制代码块实例。无论如何我们必须把这个块放置到需要的地方。如何做到这一点和如何执行这些块的最初安排也紧密相关(参见例9-5)。

例9-5：编译安排的装饰模式

```
public class CodeBlockDecorator implements CodeBlock
{
    private CodeBlock target;

    public int execute()
    {
        makeSchedulingDecision();
        target.execute();
    }

    public void replaceTarget(CodeBlock replacement)
    {
        target = replacement;
    }
}
```

例9-5演示了装饰代码块如何截取执行调用，然后可以对是否排队进行编译进行决策。除此之外，我们还有一个方法可以用一个不同的块实例来替换这个装饰的目标。一旦这个块编译完成，这个装饰就没用了，所以，理论上我们愿意替换它。这个替换实际上可以非常容易地实现。通过用如例9-6所示的块代替最初的解析块，我们可以传播新的块到了调用栈的通知。一旦这个异常达到了合适的标准，我们可以把对这个装饰的引用直接替换为对已编译块的直接引用。

例9-6：Block替换CodeBlock

```
public class CodeBlockReplacer implements CodeBlock
{
    private CodeBlock target;

    public int execute()
    {
        throw new CodeBlockReplacementException(target);
    }

    static class CodeBlockReplacementException extends RuntimeException
    {
        private CodeBlock replacement;

        public CodeBlockReplacementException(CodeBlock compiled)
        {
            replacement = compiled;
        }
```

```
            public CodeBlock getReplacementBlock()
            {
                return replacement;
            }
        }
    }
```

技巧 #6：小心使用装饰（decorator）模式

从设计的角度来看，装饰模式是好的，但是，它的额外间接性开销很大。请记住，移除和增加装饰模式都是允许的。这种移除会被认为是一次“异常事件”并可以通过抛出一个专门的异常来实现。

9.9 终极灵活性

使用了这些技巧，现在我们有了一个不必进行重要的架构调整就可以改进和扩展的高度优化的模拟系统。一个更好的编译器可以插入这个系统的后端，可以使用其他的组件和替换不同的实现来适应大范围的目标。例如：

- 组成虚拟硬盘的数据实际上可以由世界上任意地方的服务器提供（只要需要）。
- 模拟系统的用户交互（虚拟屏幕、键盘和鼠标）可以通过一个远程系统来进行。
- JPC可以在任意标准的Java 2虚拟机上运行x86软件，因此，底层的硬件选择与操作系统和软件的选择无关。另外，虚拟机的全部状态都可以保存，模拟的机器也可以及时“冻结”。可以在随后的某天在一台不同的物理机器上恢复，虚拟机上的任何宿主软件都不会感到任何的改变。

9.9.1 灵活的数据

有了JPC，可以通过一根存储棒来携带你的磁盘镜像，连同一份完整的JVM和JPC代码。然后你可以把它插到任意的计算机上并“启动”你的机器来处理所有私人邮件和其他工作，当你完成工作并拔出存储棒时，在宿主上不会留下任何东西。

另外，你的硬盘镜像也可以放在因特网上的一台服务器上，只要载入一个本地JPC并把它指向你的服务器，你就可以从世界的任何地方访问你自己的机器。加上适当的认证和传输安全，这成为移动办公的一个强大的工具。JPC对Java空间的自然适应性意味着几乎任何设备都可以用作为远程访问的终端入口，从嵌入Java的浏览器到移动设备。

对于高安全环境中的敏感工作，通过在一个其硬件驱动位于安全服务器上的本地JPC实例上工作，可以在最基础的硬件级别保证数据的安全性和完整性。每个工作人员都可以

完全控制他们正在使用的虚拟硬件，从而使他们能够高效地工作。然而，他们没有办法从系统中抽取数据，即使他们设法破坏掉他们前面的计算机的物理安全：本地机器实际上完全不知道运行在JVM内、JPC内、访客操作系统内等的应用程序。

即使当雇员的信任不是一个问题时，在虚拟机（尤其像JPC那样灵活的虚拟机）上工作意味着物理硬件完全可以随时替换。因此，对于灾后恢复和最终备份来说，机器的全部状态都备份了起来，而不仅仅是硬盘数据，像JPC这样的模拟程序在即时故障转移很重要的地方具有很明显的优势.(即使通过广域网)。

因为JPC的硬件不可知性，这些情形也可以完全适用于非x86的硬件，对于瘦客户端来说这是完美的解决方案，而不论什么情况，用户都可以得到他们喜欢的x86环境。

9.9.2 灵活的审计和支持

通过适当的授权系统，可以远程检查和接管一个运转的JPC实例的屏幕、键盘和鼠标。通过远程收集可疑活动的键盘输入和屏幕截图以作为证据，可以有效地监控欺诈行为。JPC使低级别的硬件访问成为可能，而那些试图从模拟操作系统中探测并移除监控软件的技术能人不可能破坏这个特性。即使赋予了（访客系统上）管理员权限的用户也不能够逃避监控，无论他多有见识。

一个审计的JPC系统可以只是记录活动、即时扫描并标记动作，或更进一步防止某些行为。例如，与适当的服务器软件合作，一个监控的实例可以扫描不恰当图像的视频输出并在虚拟显卡级别模糊（或用其他内容替换）它们。这样低级别的监控意味着用户不能通过安装其他的查看软件来破坏内容保护系统。

如果一个帮助桌面可以逐字逐句地完全看见整个屏幕在做什么并直接在虚拟硬件级别用键盘和鼠标进行交互，那么，远程辅助可以变得高效得多。即使当JPC在运行那些从来都没有实现过远程访问的操作系统（例如DOS，它的使用遍及世界的许多行业和国家）时，这都有可能。

9.9.3 灵活的计算

比依赖本地资源或必要时仅仅从远程资源中导入数据来运行主模拟程序更好，核心的模拟程序可以在一台集中的“JPC”服务器上执行。因为JPC只需要一个标准的JVM就能运行，这台集中的JPC服务器可以基于完全不同于常规的x86 PC的硬件。已经有一些候选可以做到这一点，JPC已经可以在一个96核心的Azul计算服务器（Azul compute appliance）上演示了。Sun基于Niagara的服务器和来自于移动电话技术的系统（JPC已经在Nokia N95这个基于ARM11的系统上启动了DOS）也有可能。

但是，为什么要集中一台服务器来运行所有这些JPC实例呢？大概因为因特网上的任何资源都可以运行一个代表其他人的JPC实例，而屏幕输出和用户输入通过网络传输到虚拟机的拥有者。按这种工作方式，整个世界都可以看成是多个用户和多台机器，在前端的用户和后端的机器之间都没有固定的硬件拥有关系了。如果有一台机器空闲了，任意一个用户都可以使用它，远程启动一个JPC实例来处理他们个人的磁盘镜像数据。如果这个空闲的机器突然有了其他的用途，这个JPC实例可以“冻结”，它的状态可以移动到另一台空闲的物理资源上。

虽然后面这种情形对于交互式用户来说也许难以想象，对于他们来说，通过因特网进行冻结和恢复会花太长的时间，从而会觉得不太方便，但对于那些需要同时运行许多并行虚拟机而没有很多交互的用户来说，这就很有意义。这是目前那些使用大量机器来运行大规模并行任务（例如渲染动画电影的画面、通过分子模拟搜索药物、优化工程设计问题和给复杂的金融工具定价）的用户的体验。

9.10 终极安全性

容许未检查过的代码在你的机器上运行，这充满了危险，而且这种危险正在加剧。因特网上的恶意软件“malware”（也称为“特洛伊”、“键盘记录器”、“劫持软件”、“间谍软件”和“病毒”）的清单正在快速增长。你可能会丢失数据、身份失窃和受骗，最坏的是，如果你在运行从不知名的或未认证的源下载的软件时没有小心地检查，那就可能会牵连到犯罪事件中。

对于流行的操作系统和因特网浏览器的制造商修补的每个安全漏洞，似乎在原来修复的地方又会新增长两个漏洞。意识到这一点后，你如何才能运行那些真正增强你的浏览体验或提供有用服务的代码呢？

运行在Java Applet沙箱中的Java代码在十多年前就已经提供这种安全级别了。增加由JPC呈现的额外的独立安全层，你就拥有了两层保险的沙箱来在其中运行未检查过的代码。JPC的Web站点（*http://www.jpc.physics.ox.ac.uk*）演示了JPC如何能够在一个标准的Applet中作为网页的一部分来启动DOS和运行许多经典的游戏；换句话说，他们演示了一段未检查过的x86（DOS）可执行代码在任意机器上的完全安全的容器中运行。

在Applet沙箱中运行JPC有一个主要的不利方面：安全限制不允许JPC创建类加载器，因此，对JPC的速度有很大提高的动态编译禁止了。一个好消息是通过利用JPC设计的内在灵活性，可以避开这个问题而不影响安全性。

在Applet沙箱中的Java代码可以按需从网络加载类，只要它们来自和Applet代码最初所在的相同的服务器。利用这一点，我们已经构建了这样一个远程编译器：对运行在Applet中的JPC实例所需要的类进行编译，当负责运行这些JPC实例的JVM请求时把编译

好的类发送回这些实例。本机JVM仅仅把这些类看作凑巧需要而不是JPC内其他类需要的静态资源，而实际上，这些类已经由这些JPC Applet实例按需编译了。

这样，即使在标准的Java Applet沙箱中，我们也获得了已编译类的速度，而用户可以放心，无论JPC正在运行什么，JVM都遵守在一个危险环境中使执行代码这个行为变得安全的基本限制。

远程编译的一个好处是随着时间的推移，许多JPC实例利用它，它构建了一个可以在JPC实例之间共享的已编译类的库。因此，编译服务器很快成为一个先前已编译的类的Web服务器。此外，通过计算每个已编译类的请求的次数，服务器知道哪些类最常使用，因此知道对哪些类花更多的时间进行优化。虽然后面这个特性还没有实现，但是，我们相信，这样集中的优化可以弥补由于在加载类时网络延迟所造成的JPC Applet客户端所遭受的执行速度瓶颈。因此，JPC Applet客户端可以像那些本地实现编译的JPC应用程序客户端一样好。(注9)

9.11 第二次做会更好

每个人都知道一件事在第二次做的时候可能会做得更好。

——Henry Ford，《我的生活和工作》

在学术的环境中开发伴随着它自己的挑战，这些挑战和商业背景中的挑战有些微小的不同。在学术的环境中，性能目标主要是自我要求，这既有好处也有坏处。对于开发人员的这一方面需要训练以保持项目的前进路线和防止项目的中心偏移。然而，自由的环境也容许想法获得快速的发展和测试以确认或反驳它们的好处。对于非常有创造力和雄心的项目，这种氛围对于最终的成功非常关键。

JPC的架构只有一小队的开发人员就进展得非常好，(注10) 这是由于对编码采取迭代的态度才获得的。如图9-13所示，在这个项目的发展过程中已经编写了500 000行代码。而至今，只有85 000行代码留了下来。这个模拟程序的每个部分都经历了许多重写，包括一次完全清除和重写。

周而复始的净化、重写和精炼是难以达到的，虽然在学术环境中比较容易。倘若你在情绪上没有非常依恋你的代码，代码删除的过程就像是泻药，这不仅适合于代码，也适合于开发人员对于它的态度。Henry Ford是对的：你的第二次尝试几乎总是会更好。在这一系列的迭代改进达到终止的条件时，一切都会变得很好。

注9：当然，当编译服务器和JPC Applet通过100M网络连接时，实际上不会察觉到由于网络问题造成的性能瓶颈。使编译在其他地方进行的好处是释放了本地的CPU资源，而这似乎平衡了网络的延迟。

注10：在为期30个月的主开发阶段，团队的平均人数为2.5个程序员，并不都是Java专家。

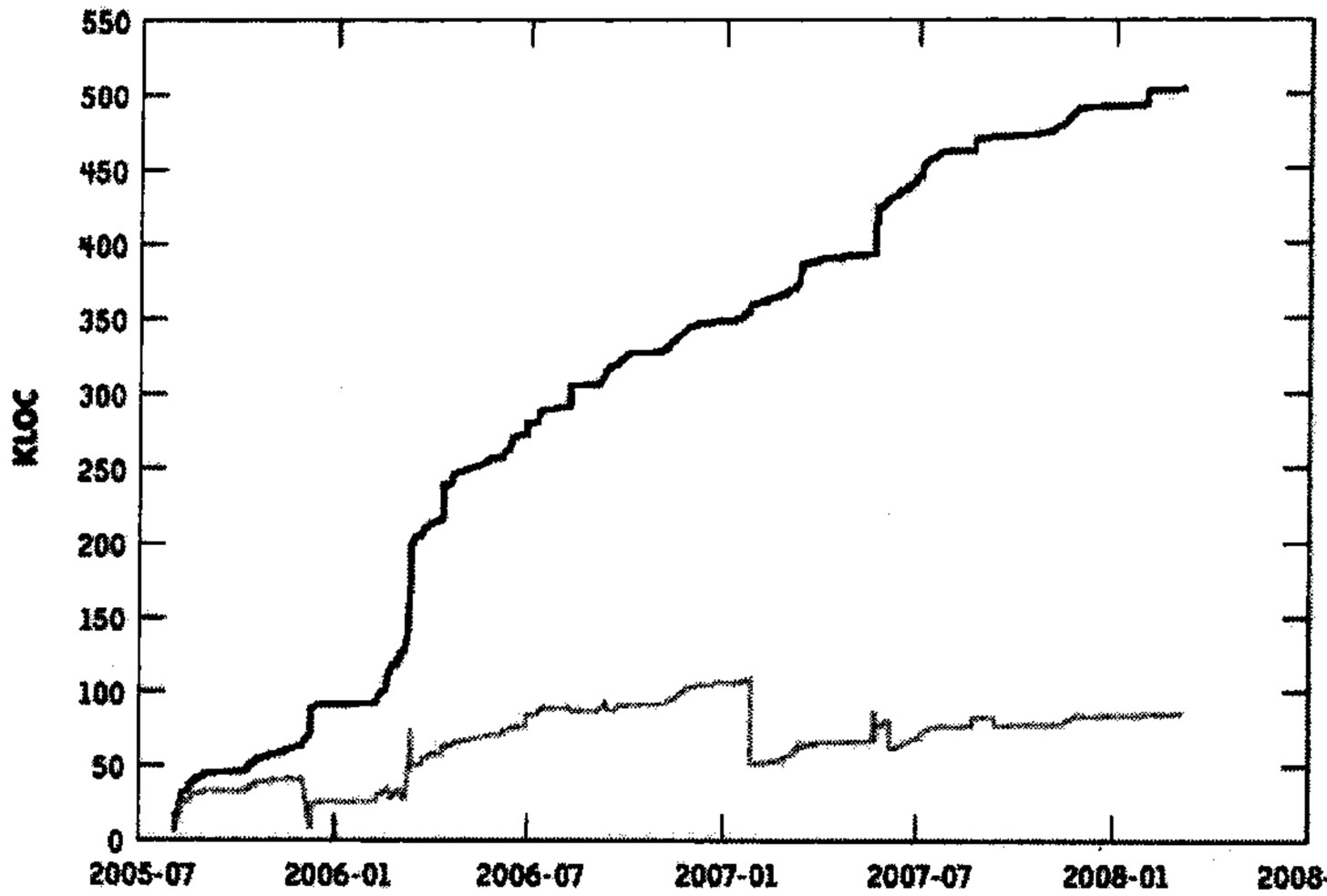


图9-13：JPC中积累的代码

所以，达到“美丽架构”的途径是一个分为4个步骤的计划：

1. 全面接受一个大而复杂的问题，找到一组比较简单的、能够使构建一个完善的端到端的原型系统成为可能的阶段。每个阶段代表一个比最终目标简单和具有较少功能的系统，但是，每个阶段都可以在它的设计限制内作为完整系统原型的一部分进行测试，而不是作为最终设计的小部分单独原型（例如，比较传统的单元测试）。

2. 在构建每个阶段的每一部分之前，清楚地知道正在开发哪个方面和为什么。理论上，每个阶段的瓶颈都可以容易地识别，对这些瓶颈的改进将是下一阶段或下面一些阶段的主要目标。原则上，在着手大块的工作之前要努力找到办法证明这个方法是否可行——即使对于每个阶段的每一部分。

3. 完成每个阶段的编码并对整个原型进行系统测试，抵制在阶段设计限制参数上快速推进的诱惑。一定要做到对每个阶段进行完整的系统测试才通知下一个阶段的测试。

4. 对设计进行迭代并返回到第二步，无论在什么情况下，你都不应该担心重写全部的组件。

在几乎不存在商业压力的学术环境中，容易在第四阶段修改代码。在商业背景中，面对必要的残酷是需要勇气的，但是我们认为在这种关键时刻缺少勇气是项目失败的一个潜在的和通常没有识别出来的主要原因，尤其对于最有创新和挑战的项目。

为了获得一个具有完全料想不到的优点的美丽架构，忠于你的信念，并无所畏惧。

	原则与特性		结构
	功能多样性	√	模块
	概念完整性		依赖关系
√	修改独立性	√	进程
	自动传播		数据访问
√	可构建性		
	增长适应性		
	熵增抵抗力		

第10章

元循环虚拟机的力量：Jikes RVM

Ian Rogers
Dave Grove

在一个托管运行时环境中运行代码是当今开发人员的普遍选择。实际上，大部分已开发代码都适合于托管运行时环境。然而，虽然运行时环境日趋流行，但是，多数代码都是用异于运行时环境支持的语言编写的。在Java虚拟机（Java应用程序的运行时环境）的情况下，一般用编程语言C和C++来实现这个运行时环境本身。

在这一章中，我们概要介绍一个名为Jikes RVM的成熟虚拟机，它采用Java语言编写来运行Java运行程序。不但运行时系统是用Java编写的，而且这个架构的所有其他组件都是用Java编写的。这些组件包括自适应和优化的编译系统、线程、异常处理和垃圾收集。我们在这儿概要介绍一下这些系统，还将解释为什么用单一的语言、运行时和实现所产生的系统更具有内在的吸引力和潜在的优势。

10.1 背景

如何开发一种新的编程语言是拥有像T型图（Aho 1986）形式的计算机科学的主要组成部分。图10-1演示了这样的一个T型图：利用一个C编译器（它运行并创建PowerPC机器码）来编译一个用C语言编写的、创建PowerPC机器码的Pascal编译器，生成一个运行PowerPC机器码的编译器，并创建PowerPC机器码。

与传统的编程语言（把代码编译到程序想要在上面运行的计算机的机器码）不同，大多数现代的语言都能够编译成架构中立的机器码。在Java中，这称为Java字节码。中立的机器码容许应用程序导出到任意运行时环境。所以，Java可以运行在现在的任意一台Java虚拟机上。

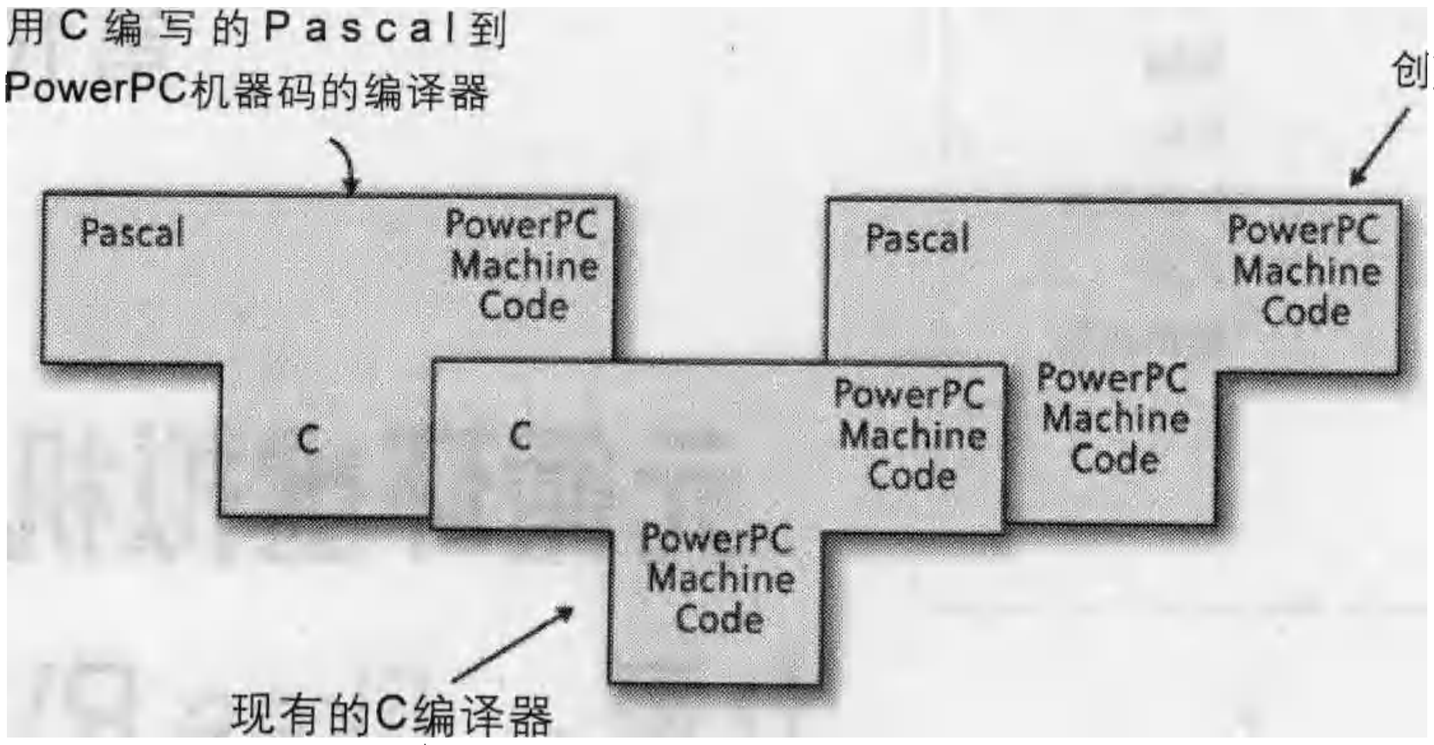


图10-1：演示了一个从Pascal到PowerPC机器码编译器的创建的T型图

现代语言希望通过解决潜在的编程语言缺陷来帮助开发人员。最普遍的特性是拥有内存安全性，通过限制开发人员如何使用数据类型和只允许自动的垃圾收集释放内存。另一个特性是抛出异常的能力。

自足执行（self-hosting）被看作是编程语言的一个重要原则。自足执行的意思是编程语言应该允许足够的表达式以使得该编程语言可以用它自身的编程语言编写。例如，用Pascal编写的Pascal编译器是自足执行的，而用C编写的Pascal编译器就不是。自足执行允许编程语言的开发人员为他们负责的内容使用该语言的特性。关键的是，自足运行形成了一个高效的循环，在这个循环中，语言实现者希望在这个语言实现的性能关键部分使用高级的和有表现力的语言特性，因此，经常发现创新的方式来有效地实现上述语言特性。

虽然使编译器自足执行很重要，但是，许多运行时环境不是由它们通常运行的语言编写的。例如，一个用C或C++编写的运行时可能会运行Java应用程序。如果这个运行时存在一个与内存安全相关的缺陷，它就会破坏Java应用程序，即使这个Java应用程序本身内存安全。消除缺陷是拥有一个自足执行的运行时的重要理由。

随着计算机系统更好地理解和发展，编程语言的要求也发生了改变。例如，编程语言C和C++没有利用多处理器处理线程的标准库（虽然像POSIX线程和OpenMP这样流行的扩展确实存在）。现代语言将使这些特性设计到语言和标准库中。允许运行时利用更好的库和抽象是拥有自足执行的运行时的另一个重要理由。

最后，无论一个运行时和它运行的应用程序在什么时候相互通信，都有用来通信的一个层。这个通信层的一项工作可能是调整对象，把一种编程语言的格式改变成另一种语言的格式。对于这些对象，通信层还必须记住不要回收在这个托管运行时之外使用的任何对象。这样的通信层不是必需的，或至少在运行时自足执行的许多情形下不是必需的。

我们希望已经提供了足够的理由来制造一个自足执行的运行时。在这章中，我们概要介绍一个这样的运行时，Jikes RVM，它由Java编写而成并运行Java应用程序。自足执行运行时环境被称为元循环（Abelson等 1985）。Jikes RVM不是唯一的元循环，实际上，它从类似于Lisp（McCarthy等 1962）和Smalltalk的Squeak虚拟机（Ingalls等 1997）这样的运行时系统中汲取了灵感。成为一个用Java编写的元循环虚拟机允许使用杰出的工具、开发环境和库。因为Java在某些团体中缺少可信性，我们将首先处理一些导致人们相信元循环Java运行时有内在缺陷的传言。

10.2 与运行时环境相关的传言

关于如何最好地创建适用于不同环境的应用程序，仍然有许多激烈的争论。在应用程序将运行的地方的可用资源、开发应用程序的开发人员的生产力的挑战和开发环境的成熟度等因素都会有影响。如果应用程序以相同的方式实现，性能和内存需求将是开发环境的一个特性。接下来，我们驳斥关于托管环境最普遍的传言。

10.2.1 因为运行时编译器必须快，所以它们必须简单

关于运行时环境的一个误解是它们对纯粹的即时（just-in-time，JIT）编译感兴趣。即时编译必须快速地创建代码，因为代码一准备好就会投入使用。虽然这个简单的执行模式用于许多早期的JVM和原型运行时环境，但是，大部分现代生产的虚拟机依赖选择性优化（selective optimization）的形式。在选择性优化中，在线剖析用于识别部分的执行方法来用一个激进优化的编译器进行优化；其他方法在执行之前立即由一个非常快速的非优化编译器进行解析或编译。正是选择性使得在运行时能够使用成熟的优化编译器。

10.2.2 静态编译器中的无限分析必然意味着更好的性能

因为运行时环境会由许多应用程序大量地使用，所以，优化它们以在所有的应用程序使用时获得更好的性能是有意义的。然而，如果运行时没有作为一个动态环境的一部分来创建，一些优化将不能实现：

在线剖析

运行时的情况在运行时会发生变化——例如，数据块的平均大小或基于不同设计模式的特殊的编码风格。在线剖析允许及时地使用这些信息来降低开销，例如分支预测，还允许更高级的优化，例如数值推测。Java的一个数值推测的例子是推测主要的流输出操作出现在`java.lang.System.out`文件流中。数值推测是局部估值的一个扩展，我们将在后面的“局部求值”一节（10.5.3节）进一步讨论局部估值。

底层系统中的差异

一个应用程序运行的系统差异正逐渐变得越来越大。不同处理器的能力、内存的数量、不同处理器的数量、电能要求和运行时在上面执行的系统的负载，这些对于了解运行时应该如何才能最好地适应都很重要。

过程内分析

过程内分析是一个优化编译器的重要工具，允许跨方法边界优化。虽然离线分析也没有限制，但通常会导致太多的数据以至编译器不能确定哪些数据很重要。因为运行时反馈更及时，它可以更好地指导程序内及其他编译器优化。

10.2.3 运行时分析使用许多资源

拥有一个运行时环境就存在这个环境所需要的内存开销。对于传统的应用程序所使用的标准库来说，类似的需要也同样存在。除此之外，运行时环境必须保存将来可以帮助它编译和执行的信息。这些内存需要是适度的，而且，通过及时和有效的内存取样，运行时环境可以用最小的开销获得最大的好处。

10.2.4 动态类加载抑制了性能

许多现代的运行时环境，包括Java，都有能力动态地扩展。这在用户想要把一个系统的不同部分插在一起的环境中是有用的；就Java而言，组件可以从因特网下载下来。这种技术虽然有用，但是，这意味着编译器不能确定关于一个应用程序的所有信息对于它来说是否都有用，反之则不然。例如，如果一个对象的一个方法正在调用，而且对于那个对象只有一个可能的类，避免进行动态的方法分派并直接调用这个方法会更好。在优化的运行时环境中有专门的优化来处理这种情况，我们将在10.5.3中讨论它们。

10.2.5 垃圾收集比显式内存管理更慢

自动的垃圾收集是计算机科学研究的一个高级领域。当内存不再需要时会被请求并回收。显式内存管理利用给运行时环境发送明确的命令来请求然后释放内存。虽然显式内存管理易于出错，但是，人们经常坚持这对于性能来说是必需的。然而，这忽略了由于显式内存管理所引起的许多复杂性。用了显式内存管理，就必须记录哪些内存块在使用并维护没有使用的内存块的清单。当这结合了许多线程并发地请求内存时，内存碎片的问题，以及把较小的区域合并起来可以形成较大的供分配的内存区域，显式内存管理器的工作会变得复杂。显式内存管理器还不能在内存中移动东西——例如，为了减少碎片。

内存管理器的需求是应用程序特有的，在一个元循环运行时的背景中，一个简单的即时编译器不必进行很多的内存分配，所以，显式内存管理或自动垃圾收集的方案都能很好

地工作。对于一个比较成熟的优化编译器而言，除了垃圾收集减少了缺陷的可能性之外，两者的区别不是很明显。对于更复杂的运行时系统的其他部分（稍后我们将在10.5.5小节中讨论），虽然我们不能反驳垃圾收集比显式内存管理更慢这种说法，但是，它明确改进了系统的“处理能力”。

10.2.6 小结

因为开发工具本身就是应用程序，我们最初关于如何最好地开发一个应用程序的问题成为了自指（self-referential）。托管语言消除了缺点并提高了开发人员的生产率。简化开发模型，探索更多的机会来优化应用程序和运行时，以元循环的方式来做这项工作，这会允许开发人员从它们引入的特性中获益而避免遇到在应用程序、运行时和系统的编译器之间的障碍。在下面的几节，我们将介绍Jikes RVM，一个把这些原理组合在一起的运行时环境。

10.3 Jikes RVM简史

Jikes RVM起源于IBM的Jalapeño项目。Jalapeño项目开始于1997年11月份，目标是开发一个灵活的研究架构来研究高性能虚拟机设计中的想法。在1998年初，一个最初的功能原型可以使用了，它能够运行小的Java程序。在1998年的春天，关于优化编译器的工作开始了，这个项目的规模快速地增长。在2000年初，项目成员已经发表了好几篇讲述Jalapeño方面的学术论文，大学研究人员开始表示有兴趣把这个系统用作为他们自己的研究成果的基础。

到2001年10月这个系统开源的时候，已经有16个大学在IBM的许可协议下使用Jikes RVM。这个团体快速地扩张，现在已经包括了好几百个研究人员及一百多个机构。Jikes RVM已经成为发表在学术刊物中的多达188遍论文的基础，而且它已经构成了至少36篇学术论文的基础。

Jikes RVM从第2版开始开源，而且已经支持Intel和PowerPC架构。已经有一些垃圾收集算法可用，包括引用计数（reference counting）、标记清除（mark-sweep）和半空间（semi-space）。一年后，2.2版本的Jikes RVM发布了。一个主要增强是全新实现了一个名为内存管理工具包（Memory Management Toolkit，MMTK）的内存管理子系统。内存管理工具包已经成为垃圾收集研究团体中一个使用非常广泛的框架，而且已经移植到Jikes RVM之外的其他运行时。我们将在10.5.9小节进一步讨论内存管理工具包和垃圾收集技术。优化编译器和自适应优化系统也有显著的提高，运行时的开发通过转到开源的GNU Classpath标准类库而简化了。在2003年4月，Jikes RVM 2.2.1成为能够运行Eclipse IDE重要部分的最早开源的Java运行时之一。

在2.2版本和2.4.6版本之间的近四年内，在功能和性能方面都有了许多明显的改进，但是，源的结构和架构大部分都没有改变。

2008年8月Jikes RVM 3.0发布了，它体现了团队近两年改进和使这个系统现代化的共同成果。代码中采用了Java 5.0的语言特性，构建的系统转为使用Apache Ant，而且还开发了一个改良很多的测试基础设施来增加系统稳定性和性能。另外，还进行了许多功能和性能的改进，导致许多程序的性能可以与使用现代的生产JVM所达到的性能（用传统的运行时语言实现的，如C/C++）相媲美。

Jikes RVM有太多的贡献者，以致在这儿不能一一提及，但是，我们要感谢Jikes RVM的开发团队。只有不多于100个人为Jikes RVM贡献了代码，有19个人作为Jikes RVM核心团队的成员。要了解全部的感谢名单，可以去Jikes RVM的站点。在2005年《IBM System Journal》（IBM系统杂志）的一篇文章（Alpern等 2005）中可以了解关于Jikes RVM的早期历史和它的开源团体的成长的更多信息。

10.4 一个自足执行的运行时自举

与一个传统编译器（图10-1）的自举相比，元循环运行时的自举涉及更多的技巧。图10-2演示了描述这个过程的一个T型图。

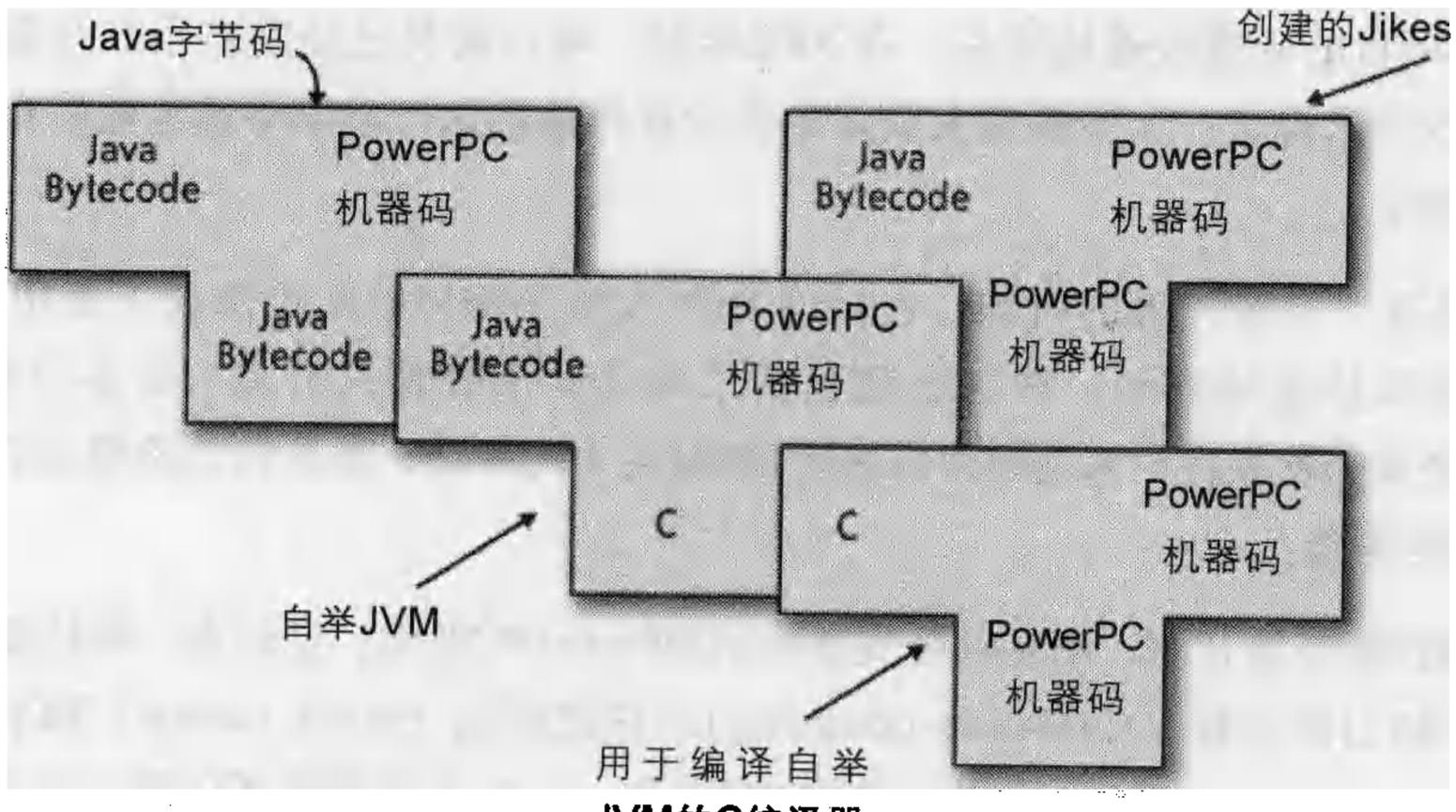


图10-2：演示了Jikes RVM在一个C编写的现有JVM上自举的T型图

这个启动映像包含了当这个系统自足执行时代表内存的好几个文件（图10-2中最右边的T）。启动映像的内容是代码和数据，类似于在一个常规编译器的对象文件中的东西。Jikes RVM的启动映像中额外包含了由垃圾收集器创建的根映射。我们将在后面的10.5.9节讲述根映射。启动映像的创建者是使用Jikes RVM的编译器创建这个启动映像文件的

一个程序，在一个自举JVM上执行。一个加载器负责把这个启动映像加载到恰当的内存区域，在Jikes RVM中，这个加载器是众所周知的启动映像管理器。

10.4.1 对象布局

启动映像记录器必须把对象布置在磁盘上，因为它们将用于运行的Jikes RVM中。Jikes RVM中的对象模型是可配置的，在保持系统的其他部分固定的同时允许可选设计进行评估。一个可选设计的例子是是否为每个对象提供更多的位来容纳对象散列，或为每个对象提供更多的位来实现同步的快速锁定。Jikes RVM对象布局的纵览如图10-3所示。

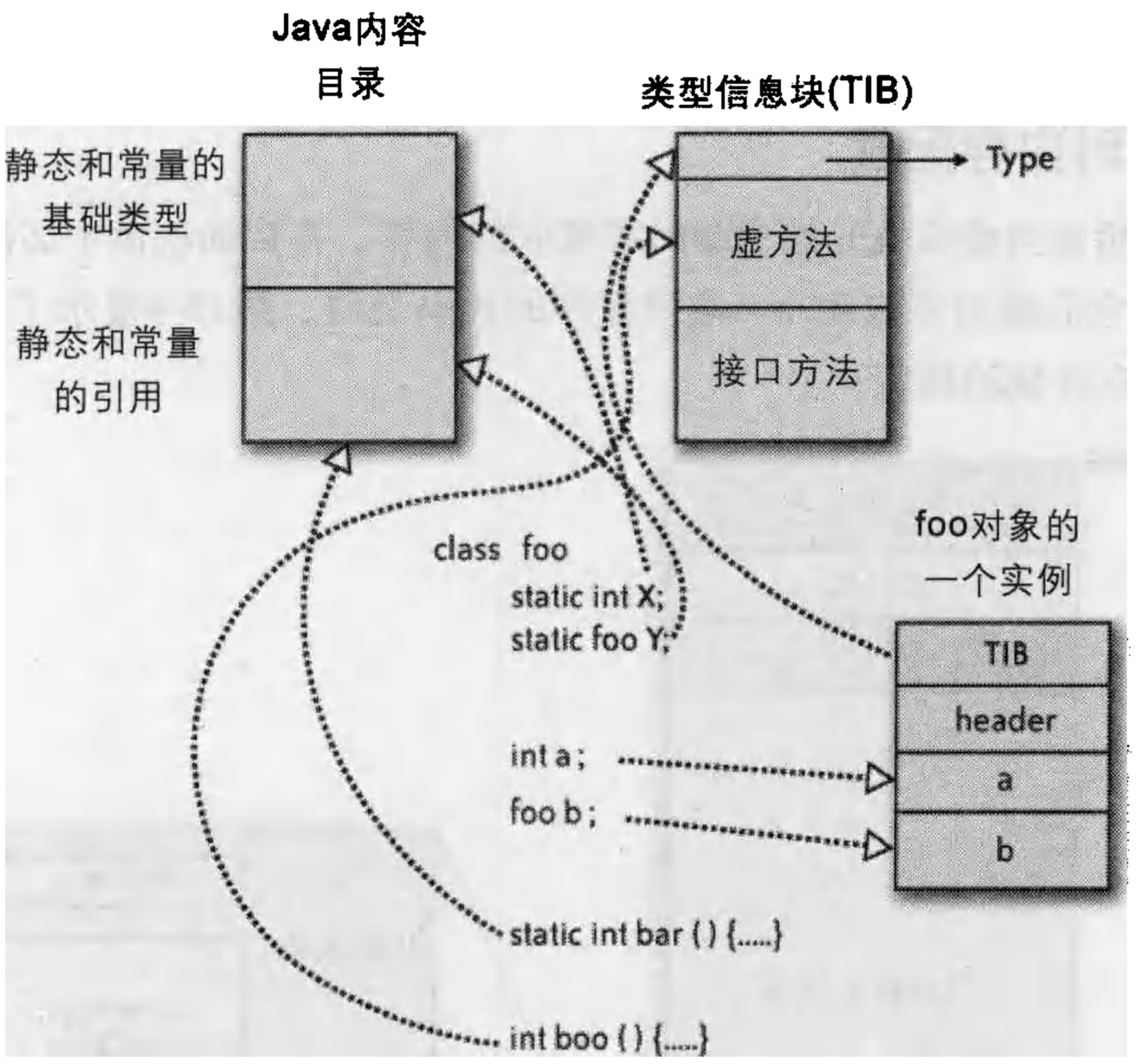


图10-3：Jikes RVM中对象的布局

Jikes RVM中默认的32位对象模式通常对于对象头使用两个字：第一个字引用类型信息块，第二个字保存对象锁定、散列和垃圾搜集的状态信息。位于对象头之后的是对象域。对于数组来说，第一个域是数组的长度，其余的是数组的元素。为了避免存取数组的移位，数组的大小就是对象头的大小和数组长度域，对对象的所有引用实际上引用这个对象内三个字的位置。这样就允许数组中的第一个元素在这个对象内的偏移为零，但是，这也意味着对象头总是在一个对象引用之后的三个字，而且，一个对象的第一个域始终位于这个对象的引用的负偏移位置。

类型信息块负责保存特定类型的每个对象的共同数据。这些数据主要用于虚方法和接口方法分派。方法分派是识别与特定对象相关并应该调用的方法的过程。类内部的方法在

类型信息块中分配位置，以便能够进行快速和有效的方法分派。类型信息块还保存那些容许快速确定运行时类型信息的值，这加速了Java的instanceof和checkcast的操作。它还保存了特别的方法以在垃圾收集期间处理一个对象。

和对象一样，属于类的静态数据也必须跟踪。静态数据保存在一个名为Java内容表（Java Table Of Contents，JTOC）的位置。Java内容表在两个方向进行组织：Java内容表中的正向偏移保存了包含引用和引用文字的静态域的内容。Java中文字的值是类似于一个字符串的东西，是字节码可以直接引用但没有域位置的东西。Java内容表中的负地址负责保存基础类型的值。按这种方式分离值，使垃圾收集器更容易地判断静态域的哪些引用应该防止一个对象被认为是垃圾。

10.4.2 运行时内存配置

启动映像记录器负责当虚拟机开始启动时布置它的内存。在启动映像中必需的所有对象都在那儿，因为它们都由将启动Java应用程序的代码引用。图10-4显示了当Jikes RVM执行时使用的内存区域的概览。

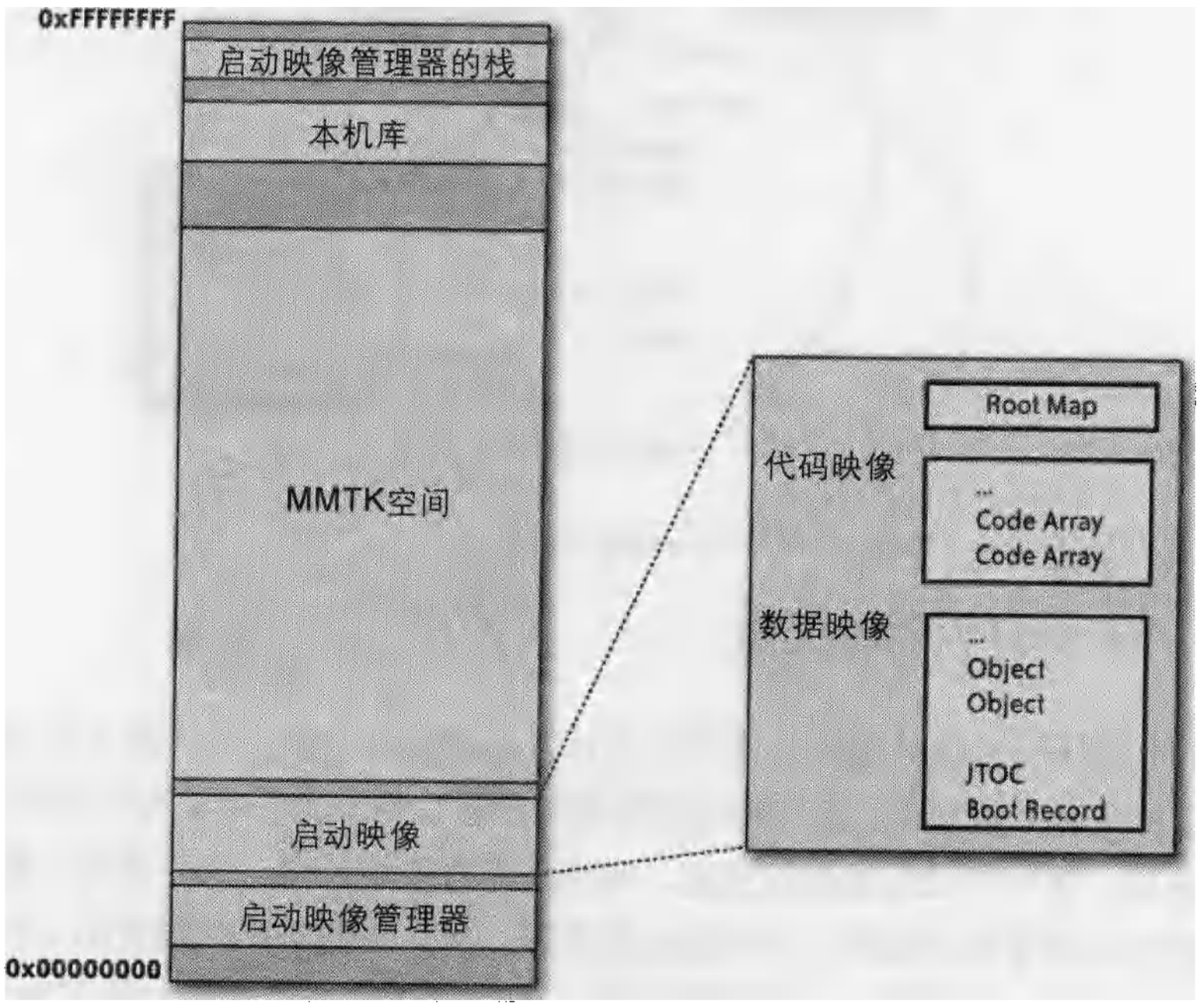


图10-4：Jikes RVM的运行时内存配置

在图10-4中可以看到许多关键项：

启动映像管理器

启动映像管理器和它的栈由负责载入启动映像的加载器组成。

本机库

类库使用的本机库所需要的内存。这将在10.5.7节进一步讲述。

MMTK空间

有不同的垃圾收集堆供MMTK使用来支持运行的应用程序。

根映射

垃圾收集器在启动映像中可获得的域的信息。在随后的10.5.9节将进一步讲述。

代码映像

分别从Java内容表和类型信息块中可以直接获得的静态方法和虚方法的可执行代码。代码写入了单独的启动映像区域以提供内存保护的支持。

数据映像

数据映像是通过先记录启动记录和Java内容表然后对Java内容表可获得的对象进行移动所创建的。利用Java的反射API，这些对象移入自举JVM中。对象移入的顺序会影响位置，由此影响性能，所以，这个移入机制是可配置的（Rogers等 2008）。

Java内容表

正如先前在10.4.1节中所述，Java内容表负责保存字和静态的域值。移动Java内容表产生启动映像。

启动记录

位于数据映像开头部分的包含在启动映像管理器和Jikes RVM之间共享数据的表。这些值在自举期间通常不能确定。

10.4.3 编译原生类并填充Java内容表

原生类是必须构建到启动映像中以让它运行的类的集合。最重要的原生类是负责启动虚拟机的`org.jikesrvm.VM`。如果某些类不是启动映像的一部分，那就不是原生类，因而就是引用的。当一个引用的对象在运行时被访问时，会导致类加载器加载并链接这个引用的类。

原生类的清单是在自举期间通过搜索目录和读取一个明确要编译的类的清单来生成的。这个明确的清单对数组类型特别重要。通过重复编译和增加构建到启动映像中的类，就可以能生成原生类的清单。但是，这会显著地增加构建Jikes RVM的时间。一个建议的替代方案是使用Java的annotation来标示哪些是原生类。

在移动对象图和记录启动映像之前，启动映像记录器会编译原生类。编译一个原生类涉及用Jikes RVM的类加载器加载它的类，这个类加载器会自动地在Java内容表和类型信息块中分配必需的空间，然后遍历所有的方法并用Jikes RVM的一个编译器编译它们。因为这全是纯Java代码，启动映像记录器利用Java的并发API来尽可能地并行执行这个任务。

一旦核心的原生类编译完成，主机JVM堆中的对象图就提供了充足的功能让Jikes RVM启动自身、分配额外的对象、开始载入和执行用户类。为了完成这个自举过程，核心对象图就利用主机JVM提供的Java的反射API的能力进行移动并记录到磁盘上Jikes RVM的对象模型中。

10.4.4 启动映像管理器和VM.boot

正如10.4节所述，启动映像管理器负责把编译后的映像载入内存。这种方式的具体细节根据操作系统而改变，但是，这些映像都设置成按请求分页载入内存。按请求分页的意思是启动映像中的页保持在磁盘上，直到需要它们。

一旦载入内存，启动映像管理器初始化启动记录，然后载入机器寄存器来把执行移交给Jikes RVM的方法`org.jikesrvm.VM.boot`（或简写的`VM.boot`）。Jikes RVM负责所有的内存布置，使高效的垃圾收集技术和有效处理Java异常的一个栈体制成为可能（参见后面的10.5.4节）。一旦进入VM.boot方法，就需要专门的封装器在启动映像管理器中的本机代码和C库之间进行传递（这些将在随后的10.5.7节进一步讲述）。

VM.boot的工作是确保这个VM处于执行程序的就绪状态。它通过初始化在记录启动映像时不能初始化的那些RVM组件来做到这一点。一些组件必须显式地启动——例如，垃圾收集器。其余的组件是那些由于在自举程序和Jikes RVM类文件之间有冲突而没有完全写入启动映像的小部分原生类。为了初始化这些类，必须运行这些类的静态初始化方法。

初始化线程系统是`VM.boot`方法的一个重要部分。它创建必须的垃圾收集线程、一个用来运行对象的finalizer方法的线程和负责自适应优化系统的线程。还会创建一个调试器线程，但只有当操作系统给Jikes RVM发送一个信号时这个线程才会安排执行。最后才创建并开始执行的线程是负责运行这个Java应用程序的主线程。

10.5 运行时组件

前面一节讲述了使得Jikes RVM准备好执行。在本节中，我们看一些Jikes RVM主要的运行时组件，从直接负责执行Java字节码的组件开始，然后看一些支持执行的其他虚拟机子系统。

10.5.1 基础的执行模型

Jikes RVM不包括解释器，所有的字节码都必须通过Jikes RVM的一个编译器转变为本机

机器码。这种编译的单位是方法，而方法的编译延迟到其第一次由程序调用时才进行。最初的编译由Jikes RVM的基线编译器进行，这是一个非常快速地生成低质量代码的简单未优化编译器。随着执行的继续，Jikes RVM的自适应系统监控程序的执行以发现程序的热点并用Jikes RVM的优化编译器选择性地重新编译它们。这是一个可以生成更高质量代码的成熟得多的编译器，但是，在编译时和编译内存使用上比基线编译器的开销大很多。

这种选择性优化的模式不是Jikes RVM所特有的。所有现代的生产JVM都依赖某种形式的选择性优化把编译资源优化成将产成最大益处的部分程序方法。正如先前所述，选择性优化是使成熟的优化编译器能够作为动态编译器进行部署的关键。

10.5.2 自适应优化系统

从架构而言，自适应优化系统实现为一组松散的同步实体。因为它是用Java实现的，我们能够利用内建的语言特性（例如线程和监视器）来构建代码。

随着程序执行，基于定时器的样例由运行的Java线程累积入取样线程。两种类型的概要数据会收集起来：当前执行的方法的样例（为了指导识别优化编译的候选方法）和调用栈的样例（为了识别重要的概要数据定向内联的调用图边界）。当取样缓冲充满时，低级别的概要代理给一个高级别的Organizer（用单独的Java线程实现的）发出一个信号来总结和记录原始的概要数据。一个Controller线程会周期性地分析当前的概要数据并利用一个分析模型来确定哪些方法（如果有）应该安排进行优化编译。这些决策利用来自“滑雪板租赁（ski rental）”（注1）问题在线算法的一个标准2-竞争解决方案所制定的。一个方法没有选定进行优化，直到优化它所预期的好处（将来调用中的加速）超过预期的成本（编译时间）。这些成本效益计算是通过结合在线概要数据（候选方法在当前的执行过程中取样的频率）和（离线的）描述预期的相对加速与优化编译器的每个优化级别的编译成本的经验驱动的常量来进行的（众所周知的编译器的DNA）。

10.5.3 优化编译

作为一个元循环运行时，Jikes RVM自己进行编译而不是依赖另一个编译器以确保良好的性能。元循环创建了一个有效力的循环：我们要在虚拟机的实现中编写清楚、雅观和高效的Java代码，这个强烈的愿望驱使我们发展创新的编译器优化和运行时实现技术。在这一节中，我们将介绍优化编译器，它由许多状态组成，这些状态归并为三个主要阶段：

注1： 查看*http://en.wikipedia.org/wiki/Ski_rental_problem*来了解更多的信息。

1. 高级中间表示（HIR）

2. 低级中间表示（LIR）

3. 机器级中间表示（MIR）

所有这些阶段都在一个由基础块组成的控制流程图上进行，这些基础块由如图10-5所示的一些指令组成。一条指令由几个操作数和一个运算符组成。因为运算符逐渐变得更加针对特定的机器，所以，运算符在这个指令的生命周期内会发生改变（变异）。操作数按定义的操作数和使用的操作数来进行组织。主要类型的操作数包括固定操作数和由寄存器编码的操作数。基础块是分支指令只出现在末端的一列指令。有专门的指令标识基础块的开始和结尾。控制流程图通过边界线连接着不同的代码区域。异常会进行特殊处理，这在后面会进行讨论。由于编译的三个主要阶段都使用相同的基础中间显示，许多优化都应用于不止一个阶段。接下来我们介绍这三个编译器阶段的主要任务。

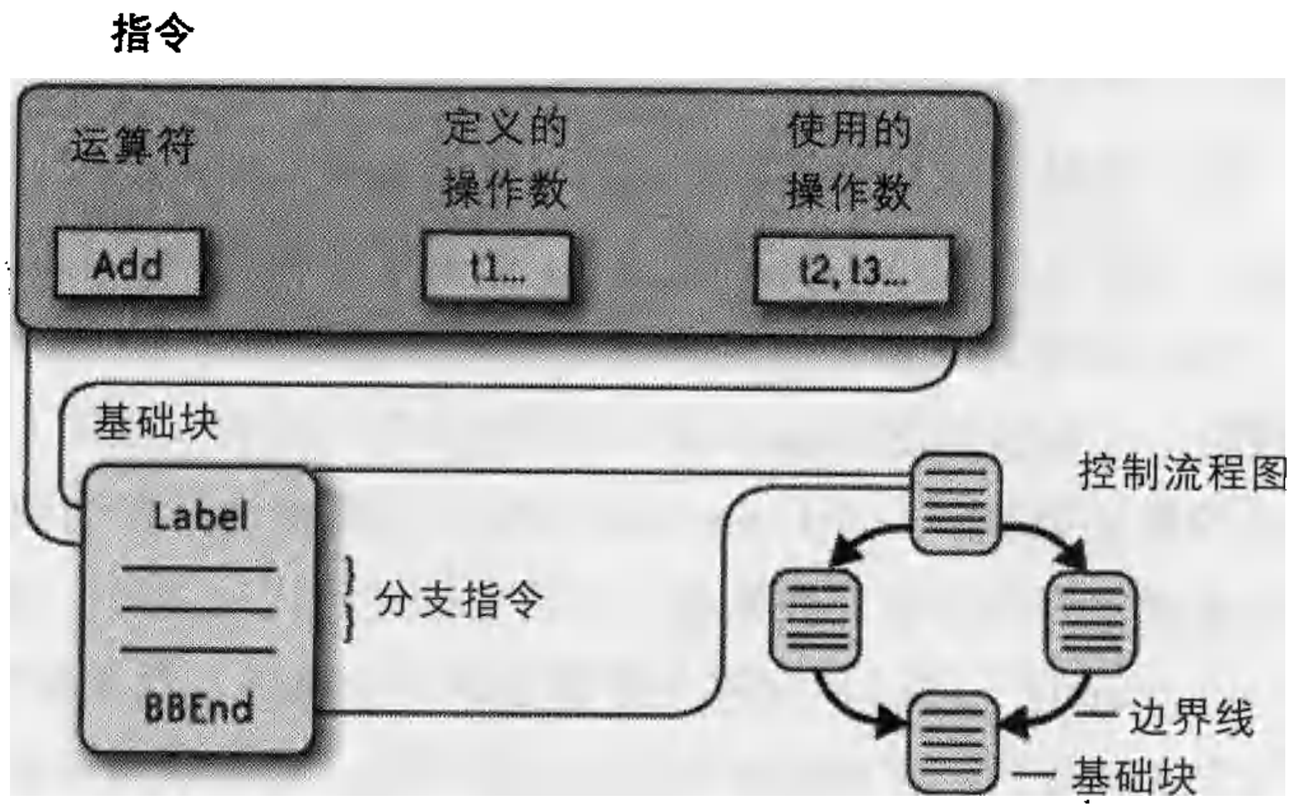


图10-5：优化编译器的中间表示的概览

高级中间表示

高级中间表示由一个名为BC2IR的编译器阶段生成，它获取字节码作为输入。这个初始阶段基于字节码操作数栈进行传播。这个栈是模拟的，字节码合在了一起来生成HIR指令，而不是生成分开的操作。除了用一个极大的符号寄存器来代替一个表达式栈之外，HIR级的运算符等同于字节码中执行的运算。

一旦指令形成，它们就转变为更简单的操作（如果更简单的操作存在）。在这之后，其余的调用指令都考虑内联。BC2IR阶段的主要部分进行递归地编写，所以，内联递归地使用BC2IR阶段。

本地优化在HIR阶段进行。本地优化考虑降低基础块的复杂性。考虑一个基础块的时候，

这个基础块内的依赖也会考虑。这简化了编译器阶段，不必考虑分支和循环的结果，这些情况会引入不同种类的数据依赖。本地优化包括：

常量传播

传播常量值以避免把它们放入寄存器中。

副本传播

利用最初的寄存器替换寄存器的副本。

简化

基于运算的操作数把运算改为更简单的运算。

表达式合并

把运算目录合并起来。例如，“x=y+1； z=x+1”可以把表达式“y+1”合并到“z=x+1”，生成“x=y+1; z = y+2”。

共用子表达式消除

查找对相同的操作数进行相同运算的指令，用前者结果的副本消除后面的运算。

闲置代码消除

如果一个寄存器在两次定义之间没有使用，那么，执行最初定义的指令是闲置代码，可以消除。

分支优化考虑改进控制流程图以便最有可能的路径按对编译器的目标来说最优的方式进行布置。其他的分支优化希望去除多余的测试和解开循环。

逃逸分析考虑对象是否可以只在编译的代码的上下文中存取，还考虑一个不只在局部上下文中存取的对象是否可以在线程之间共享。如果一个对象只是局部存取，那么，就可以排除为它分配内存的需求并把这个对象的域移入寄存器。如果一个对象只在一个线程中访问，那么，对这个对象的同步操作可以消除。

后面将介绍使用增加到这个中间形式的额外信息的许多优化。

低级中间表示

低级中间表示把先前类似于字节码的运算转换为更类似于机器码的运算。对域的运算转换为载入和存储运算；像new和checkcast这样的运算符扩充为调用提供这些运算的运行时服务并可能进行内联。适用于高级中间表示的大部分优化也适用于低级中间表示，但是，像逃逸分析这样的优化不行。由于支持的运算类型不同，因此对于不同的架构，在低级中间表示之间有一些微小的差异。对于低级中间表示指令而言，非常多的符号寄存器可以用于低级中间表示指令。

机器级中间表示

创建最终的机器级中间表示包括编译器后端的三个相互依存也相互竞争的转换。说这些转换相互竞争是因为它们的顺序可以决定生成的机器码的性能。在Jikes RVM中，第一个转换是指令选择。指令选择是类似于RISC（精简指令集计算机）的低级中间表示指令转换成已经存在于实际机器上的指令的过程。有时候，执行一个低级中间表示指令需要不止一个指令；例如，在32位的架构体系上，一个低级中间表示的长整型加法需要两个指令。在其他时候，树型模式匹配指令选择器（一个著名的自底向上重写的系统（Bottom Up Rewrite system，BURS）把几个指令合并到一个指令。图10-6演示了模式匹配的一个例子。两个模式都可以在IA32架构中找到：一个模式创建三条指令用了41条机器指令，另一个模式把加载和存储编成单个指令的内存操作，开销只有17条机器指令。最终选择了开销最小的模式。

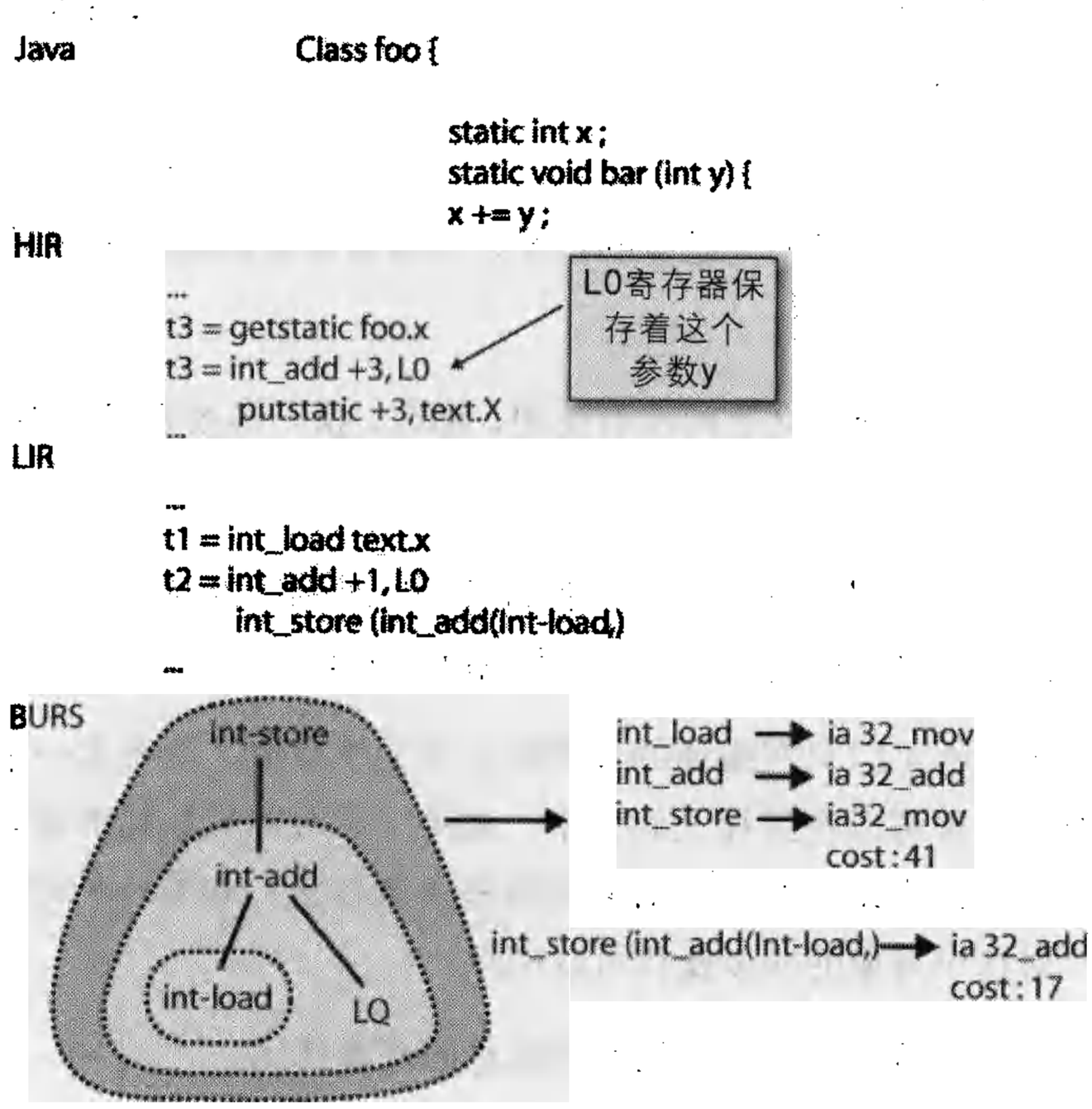


图10-6：自底向上重写系统指令选择

在指令选择后，无限的寄存器必须映射到一个真实机器所提供的有限寄存器。这个过程称为寄存器分配。当没有足够的寄存器时，可以依靠栈把值保存在内存中；依靠栈交换实际寄存器的值，这种技术称为溢流（spilling）和填充（filling）。寄存器分配器必须将溢流和填充减到最少，也要考虑任何架构的需要——例如，要求乘法和除法必须用固定的寄存器进行。

Jikes RVM有一个进行寄存器快速分配的线性扫描寄存器分配器，但是可能会产生额外的复制和内存操作。当机器上可用寄存器的数量很少时这是一个大问题。随着寄存器数量的增加和线性扫描算法的改进，对于某些代码来说，进行一次比较昂贵的寄存器分配的好处越来越不明显。

创建机器级中间表示指令的最后部分是指令调度。调度把指令分开以容许处理器充分使用指令级的平行性。

在机器级中间表示级别，只进行了很少的优化。由于与机器指令相关的很多副作用，说服并据此优化指令的行为很困难。机器级中间表示级的其他编译器阶段关注于确保遵守调用、异常和其他规定。

分解的控制流程图

如果一个空指针用于访问内存或数组的索引越界，Java程序会创建运行时异常。这些运行时异常控制了代码的流向并会因此结束基本程序块。这导致小的基础程序块具有较少的局部优化范围。为了增大基础程序块，运行时异常依赖的控制流在高级中间表示和低级中间表示级变为人工的数据依赖。这种依赖确保操作按异常的语义学恰当地排序，因此，可能在执行的中间过程中离开基础程序块来处理一个运行时异常。当这种中间形态在程序块的中间存在异常出口时，就可以说控制流程图被分解了（Choi等 1999）。

明确测试运行时异常的指令被创建并生成合适的保护结果。那些接下来需要排序的指令使用这些保护结果。可以生成一个异常的指令就是众所周知的潜在异常指令（Potentially Exceptioning Instruction，PEI）。所有的潜在异常指令会生成一个合适的保护结果，那些用来移出潜在异常指令（如果潜在异常指令是多余的）的指令也同样如此。例如，测试null的分支对多余的同样的值进行空指针测试。来自这个分支的保护结果会代替来自空指针测试的保护结果以确保指令不能重新排在这个分支之前。

图10-7演示了单个数组赋值：把它组成的运行时异常转变为确保代码按正确顺序执行的潜在异常指令和保护依赖。

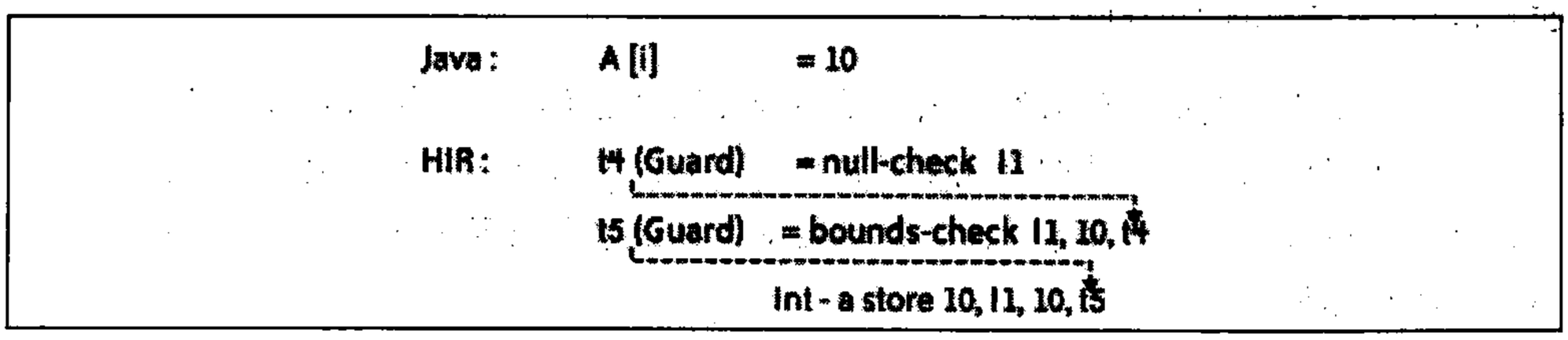


图10-7：分解控制流程图中指令的一个例子

分等级和扩展的数组SSA形式

静态单一赋值（Static Single Assignment，SSA）形式减少了编译器优化必须关心的依

赖。这种形式确保任何寄存器（更普通的叫法是变量；在Jikes RVM的高级中间表示阶段和低级中间表示阶段，所有的变量都保存在寄存器中）几乎立即写入。一次编译器转换必须处理三种依赖：

真依赖

在这种情况下，一个寄存器写入然后读出。

输出依赖

在这种情况下，一个寄存器写入，然后再次写入（第二次写入必须发生在第一次有了任何转换之后）。

反向依赖

在这种情况下，一个寄存器读出，然后写入（写入必须发生在读出有了任何转换之后）。

知道一个寄存器写入一次就意味着不会发生输出依赖和反向依赖。这个特性意味着我们先前的局部优化可以应用于全局。为了处理循环和其他分支，专门的Φ指令用于对控制流程图中不同地方的特殊值的合并进行编码。

数组静态单一赋值形式是静态单一赋值形式的一个扩展，在其中，加载和存储认为是定义一个特殊的名为堆的变量。按这种方式对内存存取建模使编译器推论如果两次读取都发生在相同的数组位置以及相同名字的一个堆，第二次读取可以替代为第一次读取的一个副本。多余的存储也以类似的方式处理。它还允许存取无关的堆来重新组织——例如，用浮点和整数运算。数组静态单一赋值形式最初设计用于FORTRAN；扩展的数组静态单一赋值形式增加了形式因素，例如Java的域不能相互同名（Fink等 2000）。

还是在静态单一赋值形式中，Jikes RVM构建了π指令，放在一个分支后面，使用一个操作数。这个π指令使用和分支一样的操作数并给它一个新名字以代替后续指令中使用的操作数。利用π指令，编译器可以推断：一个分支进行一个测试（例如一个null测试），那么，利用这个π指令的寄存器结果的任何null测试都是多余的。类似地，数组边界检查也可以取消（Bodik等 2000）。

高级中间表示的静态单一赋值形式中的循环倒置优化也消除了可能的异常。循环倒置把异常检查代码移出了循环并明确地测试异常是否会在进入循环之前发生。如果会出现异常，那就会执行一个产生异常代码的循环版本。如果不会出现异常，那就会执行一个没有异常的循环版本。

局部求值

高级中间表示优化（包括静态单一赋值优化）能够减少局部代码的复杂度，但是，这受限于可用的常量值。当一个值来自数组时通常不能确定它是否是常量，在Java中，数组

总是容许它的值发生改变。最常发生这种情况的是对于字符串。Jikes RVM引入了纯粹的annotation作为Java的扩展。利用反射，annotation能在编译期使一个有常量参数的方法求值。出于这个目的而利用注释和反射在元循环运行时是不复杂的。

局部求值有能力完全消除一些开销，例如运行时安全检查。安全检查通常遍历栈来确定哪个方法调用了一个受限方法，然后检查以了解那个方法是否有权限调用这个受限方法。因为方法在调用时进入栈的时间可以在编译器内确定，如果把进入栈作为内联的结果，精确的方法就简单地产生了。通过对安全检查代码求值，或如果这检查是纯粹的进行一次反射调用，那么安全检查的结果就可以是确定的或如果总是通过就可以取消。

当前栈替换

当前栈替换（On-Stack Replacement，OSR）是当代码在栈上执行时交换执行代码的过程。例如，如果一个运行时间长的循环正由JIT基线编译器创建的代码执行，当优化编译器创建优化代码后用优化代码交换这段代码是有好处的。它的用处的另一个例子是如果一个运行的方法有不安全的虚构的类层级属性（例如一个类没有子类，这容许在内联上进行改进），那就可以使这个方法无效。

当前栈替换通过引入保存一个方法的执行状态的新的字节码（在JIT基线编译器的情况）或新的指令（在优化编译器的情况）来工作。一旦执行状态保存下来，代码可以交换为新编译的代码，通过载入保存的状态来继续执行。

小结

本节概要介绍了Jikes RVM的优化编译器的许多高级组件。用Java编写这些组件已经给这些系统带来了许多内在的好处。线程是有用的，允许系统的所有组件都支持多线程。系统设计成线程安全，例如，容许多个编译器线程并行运行。另一个优点来自Java通用的集合，它提供了易于理解的库，简化了开发，并容许系统的组件集中在它们的角色上，而不是底层的数据结构管理。

10.5.4 异常模式

异常是例外情况吗？虽然许多程序员对这个问题都抱着肯定的观点，但是，虚拟机必须对基准测试中常见的用例进行优化。有问题的用例是读取一个输入流并当一个预期的模式出现时抛出一个异常。这种模式既出现在SPECjvm98（注2）jack基准测试中又出现在DaCapo 2006（注3）lusearch基准测试中。

首先让我们考虑一个异常需要什么。每个异常必须创建在抛出异常时当前栈的一个方法

注2：*http://www.spec.org/jvm98/.*

注3：*http://dacapobench.org/.*

清单（大家知道的栈跟踪）并把控制传给处理这个异常的catch块。通过允许出现内存保护失败，空指针异常（例如，读或写不存在的内存页）就有效地处理了。当产生一个错误时，错误指针的地址会提供给一个处理器。正是因为如此，栈跟踪和处理器信息必须确定下来。

用C语言编写的虚拟机会设法使用C的调用方式来调用Java的即时编译代码。其中的一个问题是C的调用方式不记录正在运行的代码。为了解决这个问题，C的虚拟机可以设法使用返回地址来计算正在运行的方法。必须搜索虚拟机内的每个方法来了解它对应的编译后代码是否包含特定的返回地址。对于当前栈中的每个方法都必须进行重复搜索。

由于Jikes RVM控制了它的内存配置，栈配置成包含额外的信息来快速地处理异常。Jikes RVM特别在栈上放置了一个标识符，以便不用提供方法信息就能进行简单地查找。为了确定字节码出现异常的地方和处理器的位置，需要利用返回地址或错误指针地址进行计算。这意味着处理一个异常的速度与栈的深度成比例，而不由虚拟机中方法的数量来决定。

和其他虚拟机一样，Jikes RVM尽可能优化地远离异常，最好把异常压缩到分支。它还把许多方法内联到一个方法中，因此，在栈上放置一个标识符的情况很少发生，只有当处理未内联的方法时才这样处理。

10.5.5 Magic、annotation，使事情流畅地进行

由编译器生成的代码有直接存取内存的指令。有时候需要直接存取内存——例如，为了实现垃圾收集程序或为了锁住对象而访问对象头。为了容许强类型地内存存取，Jikes RVM利用一个名为VM Magic的库，它已经作为一个标准接口并用于其他的JVM项目中。VM Magic定义了这两种类：编译器pragma annotation（扩展了Java语言）或未封装类型（描绘对底层内存系统的存取）。未封装类型的地址用于直接存取内存。

Jikes RVM的编译器内特别处理了VM Magic的未封装类型。对未封装类型的所有方法调用都作为直接操作一个宽度为一个字的值来对待。例如，“plus”这个方法会作为累加机器底层一个字的宽度来对待，从而直接影响通常是这个对象的指针的内容。由于操作的这个值不是一个对象的引用，编译器会标明，垃圾收集器对保存未封装类型的位置和保存基础字段（例如一个int）的内存地址同样感兴趣。

拥有强类型存取内存还不足于完全容许Jikes RVM完好地运行。还需要一些专门的方法向编译器指明内在的事情正在发生或它们应该简化部分的Java强类型语法。这些专门的方法存在于org.jikesrvm.runtime.Magic中。一个内在操作的例子是平方根方法，对于这个方法，Java没有提供字节码，但许多计算机架构都有直接获得结果的一条指令或一个程序。绕过Java的强语义的一个例子是在线程调度的过程中避免转换，因为在某些关键时刻运行时异常是不允许的。

所有的神奇操作都采用不同于常规方法的方式编译，而且它们不能作为单独的方法进行编译。方法确实在那儿，但如果一旦执行它们，就会抛出一个异常。编译器根据方法来确定哪个神奇运算正在执行，然后寻找编译器必须为它提供哪些运算。在启动映像的创建期间，提供了一些未封装的神奇类型和方法的替换。例如，对象在配置好之前在启动映像中的地址是未知的。启动映像的未封装类型知道标识符和它们映射的对象。在启动映像的创建期间，这些标识符链接到它们引用的对象。

虽然神奇运算和未封装类型允许Jikes RVM中的Java代码像C或C++中的指针一样有权访问内存，但是，指针是强类型，不能用来代替引用。强类型使程序员的错误在编译期就能发觉，优秀的集成IDE和对查找缺陷的工具的支持，所有这些都帮助了Jikes RVM的开发。

10.5.6 线程模式

Java有完整的线程支持。在1998年，操作系统对线程的支持还不能很好地适合许多Java应用程序。尤其是，典型的操作系统线程实现方式不适合支持单个进程中有好几百个线程，锁操作（实现Java的同步方法所必需的）的开销非常昂贵。为了避免这个开销，一些JVM自己实现线程，利用一个名为绿色线程（green threading）的操作模式。在绿色线程中，线程是通过让JVM自己管理这个过程（把多个Java线程复合到较少的操作系统线程上，通常是复合到一个操作系统线程，或在多处理器系统中每个处理器一个线程）来实现的。绿色线程的主要优点在于实现了Java语义中特有的锁和线程调度并使它非常容易扩展。主要缺点是操作系统不知道JVM正在做什么，这会导致许多性能反常，尤其当Java应用程序与本机代码频繁交互的时候（它自身对线程模式进行了假定）。然而，实现绿色线程的JVM已经为许多Java操作系统项目（能够避免绿色线程在标准操作系统上的性能反常）提供了基础。

随着处理器性能和操作系统实现的改进，在JVM中管理线程的优势已经不明显了——虽然在JVM内管理非竞争的锁操作仍然具有优势。

利用Java，一个清楚的线程API已经创建以允许Jikes RVM拥有不同的底层线程模式，就像由不同的操作系统提供的那些模式一样。将来，在语言和操作系统之间拥有一个灵活的接口可以容许Jikes RVM适合新的程序员行为（例如，支持数千个线程）。

10.5.7 本机接口

很不幸，Jikes RVM不能全部使用Java代码。访问操作系统程序必须分配内存页。Jikes RVM的类库把Java代码与现有的库（例如用于窗口化的库）结合起来。Jikes RVM提供了两种方式来访问本机码：

Java本机接口（Java Native Interface，JNI）

JNI是容许Java应用程序与通常用C编写的本机应用程序结合的一个标准。JNI的一个特性是可以处理Java对象，也可以调用Java方法。为了防止本机代码访问的对象被当作垃圾收集掉，这些对象必须记录下来。当使用JNI调用本机方法时，这会产生一些开销。

系统调用

在系统调用的声明中，它们与本机方法相似，只是它们有一个额外的annotation。由于限制了本机代码不能通过JNI接口回调入虚拟机，系统调用容许更有效地与本机代码进行传入和传出。Jikes RVM利用默认的操作系统调用惯例把系统调用实现为对一个本机方法的简单的进程调用。

10.5.8 类加载器和反射

类加载器负责把类载入Jikes RVM并把这个过程与对象模式连接起来。反射允许一个应用程序查询对象的类型，甚至对在静态编译期还不知道其类型的对象进行方法调用。反射是通过应用程序使用像`java.lang.Class`这样的对象或像`java.lang.reflect`包中那些对象（它们是在Jikes RVM的运行时执行的API封装器）来进行的。

由于一些优化依赖类的层次结构，重要的运行时钩子存在于类加载器中，如果先前保存的假定不再正确时它们可以触发重新编译。关于这些内容，我们已经在前面的10.5.3节详细地讨论过。

10.5.9 垃圾收集

内存管理工具包（Memory Management Toolkit，MMTk）是构建高性能内存管理实现的一个框架（Blackburn等，2004年5月）。经过5年时间，MMTk已经成为垃圾收集研究团体密集使用的核心基础。它的高便捷和可配置架构是它取得成功的关键。除用作Jikes RVM的内存管理系统之外，它也移植到了其他语言的运行时，例如Rotor。本节将略微讨论一些MMTk的关键设计概念。关于垃圾收集算法和概念的完整信息，最好的参考是《Garbage Collection: Algorithms for Automatic Dynamic Memory Management (Jones和Lins 1996)》。

MMTk的一个底层元法则完全用在了Jikes RVM中：通过依赖一个激进的优化编译器，可以以高层次、可扩展和面向对象的风格编写出高性能的代码。这个法则在MMTk中得到最大的发展，即使对性能而言非常关键的操作（例如快速的对象分配路径和写障碍）也完全表示成结构良好的、面向对象的Java代码。表面上，这种风格与高性能（分配单个对象，有许多代码级的虚拟调用和相当多复杂的数值计算）完全不相容。然而，当分

配序列由优化编译器递归内联和优化后，它会产生紧密内联的代码序列（类似于分配一个对象并初始化该对象头的10个机器指令），这些代码序列等同于那些通过在汇编代码中手写的快速路径序列或把它硬写入编译器而获得的代码序列。

MMTk把内存组织成空间（Space），这是（可能不连续的）虚拟内存区域。MMTk提供了空间的许多不同的实现，每个实现都体现了一个分配和回收由空间管理的虚拟内存块的特殊基础机制。例如，CopySpace利用碰（bump）指针分配并通过把所有活的对象复制到另一个空间来回收；MarkSweepSpace通过这种方式组织内存：利用空闲列表把空闲的内存块链在一起，通过“清扫”无记号的（死的）对象并把它们重新链接到适当的空闲列表来支持分配。

计划（Plan）是MMTk关于一个垃圾收集算法的典型构想的术语。计划是通过把一个或多个空间实例以不同的方式组合在一起来定义的。例如，MMTk的GenMS计划实现了相当标准的按代的标记来清除的算法，组合了一个CopySpace和一个MarkSweep空间来分别实现新生代和壮年代的空间。除了用于管理用户级Java堆的空间之外，所有的MMTk计划还包括好几个用于管理虚拟机级内存的空间。特别地，专门的空间用于分配即时生成的代码、Jikes RVM启动映像和低级别的虚拟机实现对象（例如类型信息块）。在MMTk 3.0版本中预定义和发布了15个计划（也就是15个不同的垃圾收集算法）。此外，许多其他的计划已经由MMTk研究团体开发出来并发布在学术文献中。

MMTk和它的主机运行时环境通过两个仔细定义的、说明MMTk暴露给主机虚拟机哪些API和MMTk希望主机提供哪些虚拟机服务的接口来进行交互。为了保持MMTk对于多个主机虚拟机的轻便性，构建过程通过依靠虚拟机接口的一个存根实现单独编译MMTk来严格控制这两个接口。或许这些接口最复杂的是用于允许MMTk和Jikes RVM协作识别用于垃圾收集的根的那部分。大部分MMTK的计划都代表轨迹收集器。在一个轨迹收集器中，垃圾收集器从一组根对象（通常是这个程序的全局变量和线程栈结构中的引用）开始并对它们的引用域求传递闭包来找到所有可获得的对象。可获得的对象认为是活的且不会收集掉。传递闭包中不可获得的任何对象都是死的；这些都会由收集器安全地回收并用于满足将来的内存分配请求。在Jikes RVM中，垃圾收集的根来自寄存器、栈、Java内容表和启动映像。启动映像中的引用根据根映射来确定，而根映射压缩了包含引用的启动映像中的所有偏移。要确定栈上和寄存器中的引用，这个方法和在它内部的位置都按异常模式中所用的相同方式来确定（参见前面的10.5.4节）。根据编译器确定的方法，它可以为MMTk提供一个处理寄存器和栈来返回这些引用的迭代器。

Jikes RVM集成

Jikes RVM在对象表示的初始创建期间和MMTk结合在一起，提供迭代器来处理引用、对象分配和障碍实现。对象分配和障碍会影响性能，因为MMTk是用Java编写的，所以，

关联的代码可以直接连接到高效的编译代码。因为多种原因，障碍在垃圾收集计划中是必需的。例如，读障碍对于捕获可能被一个并行垃圾收集器复制的对象的使用情况是必需的，而写障碍用于分代收集器中，在那儿，写一个旧的对象就意味着必须把这个旧的对象当作年轻代回收的根。

小结

MMTk提供了一组功能强大而且流行和精确的垃圾收集器。能够容易地把Jikes RVM和MMTk的不同模块链接起来，这使开发变得容易而且减少了开销，出于性能的考虑，还使Jikes RVM内联了部分MMTk。用Java编写垃圾收集算法使得垃圾收集器的实现人员不用考虑编译器内部发生的情况。Java固有的线程已经使得垃圾收集器并行且和运行时模式结合在一起。Java的库为收集器的接口提供了灵感，这意味着用MMTk编写新的垃圾收集器已经显著地简化了。

10.6 经验教训

Jikes RVM是一个成功的实验虚拟机，以一种灵活和易于扩展的方式提供近似于艺术级的性能。以运行时支持的语言来编写，这允许紧密的集成和组件重用。Java的使用使简单易懂的代码、好的模块性和高质量工具的使用（例如集成开发环境）成为可能。

Java语言和JVM的实现正在发展中，Jikes RVM已经而且还将不断地随着这些改变而发展。一个有趣的发展是X10编程语言，它通过提供线程安全的保证来处理程序员如何为许多核心系统开发应用程序的问题，正如Java中垃圾收集提供内存安全保证一样。Jikes RVM的代码已经为X10的开发提供了一个出色的测试环境。因为JVM的实现引起了新的优化，例如新的垃圾收集技术或对象内联，所以，把这些优化嵌入到一个框架（例如RVM）中意味着运行时、编译器和代码基也改进和展示了元循环的效率。

元循环环境的扩展性使Jikes RVM成为一个进行多语言虚拟机试验的出色的平台。这种扩展也允许Jikes RVM的概念以不同的编程语言实现。

Jikes RVM有许多其他有趣的扩展和相关的项目，包括通过二进制翻译支持像C和C++这样的语言，通过扩展在虚拟机内部提供面向方面的编程，使整个虚拟机进入一个操作系统以消除运行时优化的障碍。虽然最初存在争议，但是现在，垃圾收集、自适应优化和链接时优化是开发人员所期望的。Jikes RVM演示了如何通过一个帮助开发的共同的元循环环境来达到和适应这些特性。Jikes RVM支持一个大的研究团体并获得了高性能。通过拥有一个美丽的架构，Jikes RVM可以为将来的运行时环境连续地提供一个平台。

参考文献

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press.

Aho, Alfred, Ravi Sethi, and Jeffrey Ullman. 1986. *Compilers, Principles, Techniques, and Tools*. Boston, MA: Addison-Wesley.

Alpern, Bowen, et al. 2005. "The Jikes Research Virtual Machine project: Building an open-source research community." *IBM Systems Journal*, vol. 44, issue 2.

Blackburn, Steve, Perry Cheng, and Kathryn McKinley. 2004. *Oil and water? High performance garbage collection in Java with MMTk* (pp. 137-146). International Conference on Software Engineering, Edinburgh, Scotland. ACM, May '04.

Bodik, Rastislav, Rajiv Gupta, and Vivek Sarkar. 2000. *ABCD: eliminating array-bounds checks on demand*. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000), Vancouver, British Columbia, Canada. ACM '00.

Choi, Jong-Deok, et al. 1999. *Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs*. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99), Toulouse, France: ACM, Sept. '99.

Fink, Stephen, Kathleen Knobe, and Vivek Sarkar. 2000. *Unified Analysis of Array and Object References in Strongly Typed Languages*. Static Analysis Symposium (SAS 2000), Santa Barbara, CA: Springer Verlag.

Ingalls, Daniel, et al. 1997. "Back to the future: the story of Squeak, a practical Smalltalk written in itself." *ACM SIGPLAN Notices*, vol. 13, issue 10: 318-326.

Jones, Richard and Rafael Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Hoboken, NJ: John Wiley and Sons.

McCarthy, John, et al. 1962. *LISP 1.5 Programmer's Manual*. Cambridge, MA: MIT Press.

Piumarta, Ian, and Fabio Riccardi. 1998. "Optimizing direct threaded code by selective inlining." *ACM SIGPLAN Notices*, vol. 33, issue 5: 291-300.

Rogers, Ian, Jisheng Zhao, and Ian Watson. 2008. *Boot Image Layout For Jikes RVM*. Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '08), Paphos, Cyprus. July '08.

第四部分

最终用户应用架构

[redacted]

	原则与特性		结构
√	功能多样性	√	模块
	概念完整性		依赖关系
√	修改独立性		进程
	自动传播		数据访问
	可构建性		
√	增长适应性		
√	熵增抵抗力		

第11章

GNU Emacs：滋长的特性是其优势

Jim Blandy

我常使用Emacs，它是一种功能超强的文字处理程序。它的作者是Richard Stallman。它是用Lisp语言编写，这是唯一的一种优美的计算机编程语言。它很庞大，而且只能编辑纯ASCII的文本文件，也就是说，没有字体、不能加粗、无法添加下划线……如果你是一名专业的作者，即不需要考虑对文字设置格式、生成打印稿，那么Emacs会使其他编辑软件相形见绌，就像正午的太阳相对于星星一样。它不仅庞大、光芒四射；而且能使其他的东西化为乌有。

——Neal Stephenson

GNU Emacs文本编辑器和其恶名并不相配。它的支持者认为它是无可比拟的，甚至顽固地拒绝使用现在更流行的文本编辑器。而其反对者说它隐晦、复杂，而且和Visual Studio这样的主流开发环境相比，它显然有些过时了。甚至有些爱好者也抱怨，Emacs中复杂的组合键弄伤了自己的手腕。

Emacs所激起的争论和Emacs所提供的功能一样多。当前版本的Emacs拥有110万行代码，它是用Emacs自己的编程语言Emacs Lisp编写的。其中有一些代码是用来帮助你更好地使用C、Python以及其他语言完成编程任务的。同时还有一些代码是用来帮助你完成程序调试，在程序员之间进行协作，阅读电子邮件和电子新闻，浏览和搜索文件夹，以及解决符号代数问题。

继续探究其内幕，情况就有些奇怪了。Emacs Lisp并不支持对象，它的模块系统就是一种命名规范，所有基本的文本编辑操作使用了隐晦的全局变量，甚至局部变量也不是很

局部。Emacs对许多广泛接受的、有用的、有价值的软件工程原则都不屑一顾。其代码已经有24年的历史了，体系庞大，是由成百上千的不同开发人员共同完成的。总的来说，整个代码在不断膨胀中。

但它却工作得很好。其功能集不断地成长；用户界面中不断增加了许多令人着迷的新行为；而且整个项目有效避免了对基础架构的大幅修改、频繁的交互、领导者的冲突和分歧。其活力的源泉是什么？它有什么局限性？

最后，其他软件可以从Emacs中学到什么？如果有一个新架构需要实现与Emacs相同的目标，在度量是否成功时应问什么问题？综观全章，我们将分析这三个问题，相信这样就能够挖掘出Emacs架构中最有价值的特性。

11.1 使用中的Emacs

在探讨其架构之前，我们先简单看看Emacs是什么。它和你经常使用的其他文本编辑器类似。当你用Emacs打开一个文件时，将弹出一个窗口，并显示出该文件的内容。你可以对其内容进行修改，然后保存这些修改后退出。但是，Emacs并不一定在所有情况下都十分有效，如果它的功能对你不是全都有用，那么你可能会抱怨它的启动速度要比流行的文本编辑程序更慢。当遇到这种情况时，我不会使用Emacs。

Emacs的设计预想是你需要时只启动一次，然后一直运行着它。你可以在一个Emacs会话中对多个文件进行编辑，并保存修改。Emacs能将这些文件保存在内存中而不用显示它们，所以你看到的就是当前的文件，当你需要对其他文件进行编辑时也能够马上切换过去。有经验的Emacs用户只会在发现计算机内存不足时才关闭文件，因此运行很久的Emacs会话可能会打开成百上千个文件。图11-1所示的屏幕截图展示了打开了两个帧的一个Emacs会话。左边这个帧分成了三个窗口，分别显示了Emacs广告页、可浏览的目录列表以及展示Lisp交互界面的缓冲区（buffer）。右边的帧只有一个窗口，显示的是存储源代码的缓冲区。

这里涉及了三种最基本的对象：帧、窗口、缓冲区。

帧是Emacs对计算机图形界面中窗口的称呼。在前面的屏幕截图中显示了两个紧挨着的帧。如果以文本终端使用Emacs，例如通过telnet或ssh连接，那么这个终端也是Emacs中的帧。Emacs可以管理多个图形帧，也可以同时管理多个终端帧。

窗口是帧的一部分。（注1）要创建一个新的窗口，只能通过将原有窗口分成两个，当你

注1：绝大多数图形界面中都称为窗口，在Emacs中称为帧，因为Emacs将术语“窗口”用来描述更小的部分。这似乎是不合时宜的，不过Emacs的术语早在图形界面被广为接受之前就确定了，因此Emacs的维护人员通常不倾向对它进行修改。

关闭某个窗口时，将把空间给邻近的窗口；这样的结果是一个帧中的窗口始终会填满整个帧的内容。只有当前选中的窗口，才会对键盘命令进行响应。窗口是轻量级的，在使用时经常会频繁地创建和关闭。

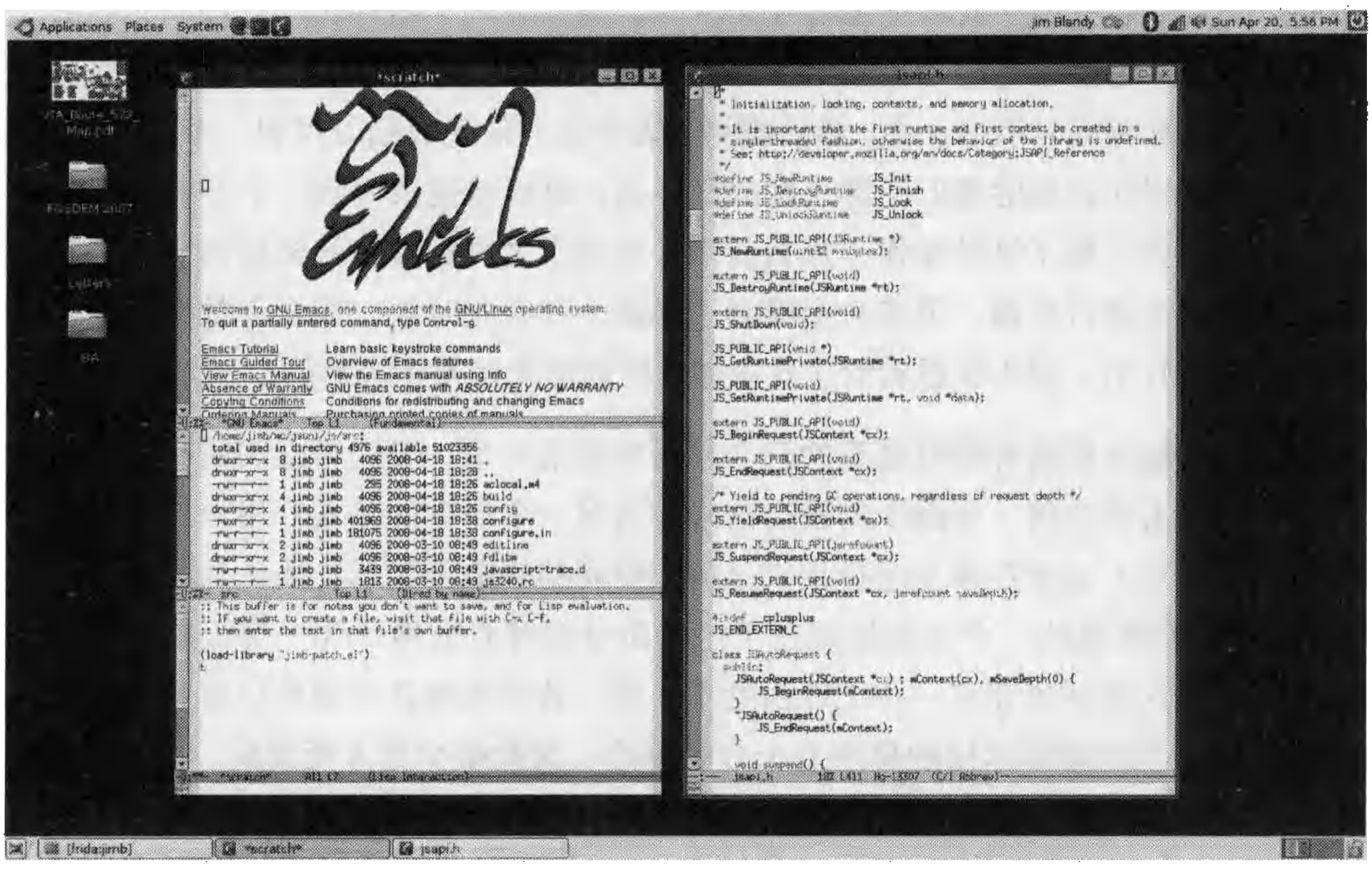


图11-1：使用中的Emacs

而缓冲区用来保存可编辑的文本内容。Emacs将会把打开的文件中的文本内容保存在缓冲区中，但缓冲区中的内容并不一定必须与某个文件关联：它可能包括搜索结果、在线文档，以及刚输入还没有保存到任何文件中的内容。每个窗口将显示某些缓冲区的内容，而一个缓冲区可能保存0个、1个或多个窗口。

还有一个很重要的地方，那就是除了每个窗口底部的模式行以及其他类似的东西，Emacs在向用户显示文本信息时首先会将其放在缓冲区中，然后将缓冲区的内容显示在某些窗口中。帮助信息、搜索结果、目录列表以及其他类似的内容，保存在缓冲区中的内容都有相应的名称。这看起来是一个拙劣的实现技巧，虽然它简化了Emacs的内核，但实际上它是相当有价值的，因为这样做可以使得这些不同的内容都被当作可编辑的文本内容来处理：你可以使用相同的命令来在这些内容中导航、搜索、组织、剪裁、排序，就像文本缓冲区中的其他内容那样。任何命令的输出都可以作为其他命令的输入。这与诸如Microsoft Visual Studio之类的编程环境形成了鲜明的对比，在这些环境中，搜索结果等只能在相应的工具中才有用。不过这也不是Microsoft Visual Studio独有的，绝大多数图形界面的编程环境都有这样的缺点。

例如，在上面的屏幕截图中，左边的帧中位于中间的窗口展示了一个目录列表。和大多数目录浏览器一样，该窗口为复制、删除、重命名、文件比较、通过匹配模式或正则表达式筛选一组文件、通过Emacs打开文件提供相应的快捷键命令。但与大多数目录浏览器不同，在Emacs中该列表本身也是纯文本的，保存在一个缓冲区中。Emacs中所有通用的搜索机制（包括优秀的增量搜索命令）都可以使用。只要你需要，都可将列表中的内容复制并粘贴到临时缓冲区中，然后删除列表中的其他元数据以得到一个文件名列表，接着使用正则表达式剔除我们不感兴趣的文件名，最后就能够得到一个传给新命令的文件名列表。当你习惯了这种操作，那么使用常见的目录浏览器时就会感到恼火：不能对显示出来的信息进行处理，很多命令受到了限制。在有些时候，甚至会觉得命令行中的组合命令也不好用，因为要想看到这些命令的中间结果是很不容易的。

这将使我们想起本章最开始时提到的三个问题中的第一个，我们可以问所有用户界面一个问题：如何方便地将一个命令的输出结果作为另一个命令的输入？用户界面提供的命令能否相互组合？或者在命令运行结束之前，中间结果能否显示？我认为许多程序员喜欢UNIX命令行环境的一个主要原因在于这些命令能够有效地作为一部分整合到复杂的脚本中，虽然这些环境存在一些无理由的不一致、数据模型过于简单以及其他的一些缺点。由于写一个能够脚本化的程序是十分简单的，因此很少走入死胡同。Emacs实现了相同的目标，尽管是以完全不同的方式实现的。

11.2 Emacs的架构

Emacs架构所采用的是在交互式应用程序中应用广泛的模型-视图-控制器模式，如图11-2所示。在该模式中，模型是程序所操作数据的底层描述；视图则是向用户展示数据的方法；而控制器则负责实现用户与视图的交互（按键、点击鼠标、选择菜单项等），并对模型进行相应的更新。在本章中，我谈及该模式时将使用模型、视图和控制器进行描述。在Emacs中，控制器基本上是完全用Emacs Lisp语言编写的。Lisp最初是用来操作缓冲区中的内容（模型）以及窗口布局的。负责重绘的代码（视图）将完成对显示内容的更新，它无须直接受Lisp代码控制。不管是缓冲区的操作还是负责重绘的代码都可以通过Lisp代码进行定制。

11.2.1 模型：缓冲区

Emacs是用来编辑文本文件的，因此Emacs的模型中最为核心的是用来保存文本信息的缓冲区。缓冲区就是简单的字符串，在每一行的行末是新行字符；它并不是一个行的列表，也不是像Web浏览器展示HTML文档时使用的文档对象模型那样的节点树。Emacs Lisp对于缓冲区的基本操作包括添加、删除文本，以字符串的形式析取缓冲区中文本的一部分，通过精确的字符串或正则表达式来搜索匹配的字符串等。缓冲区能够保存各种字符集中的字符，包括亚洲、欧洲可能使用到的一些字符。

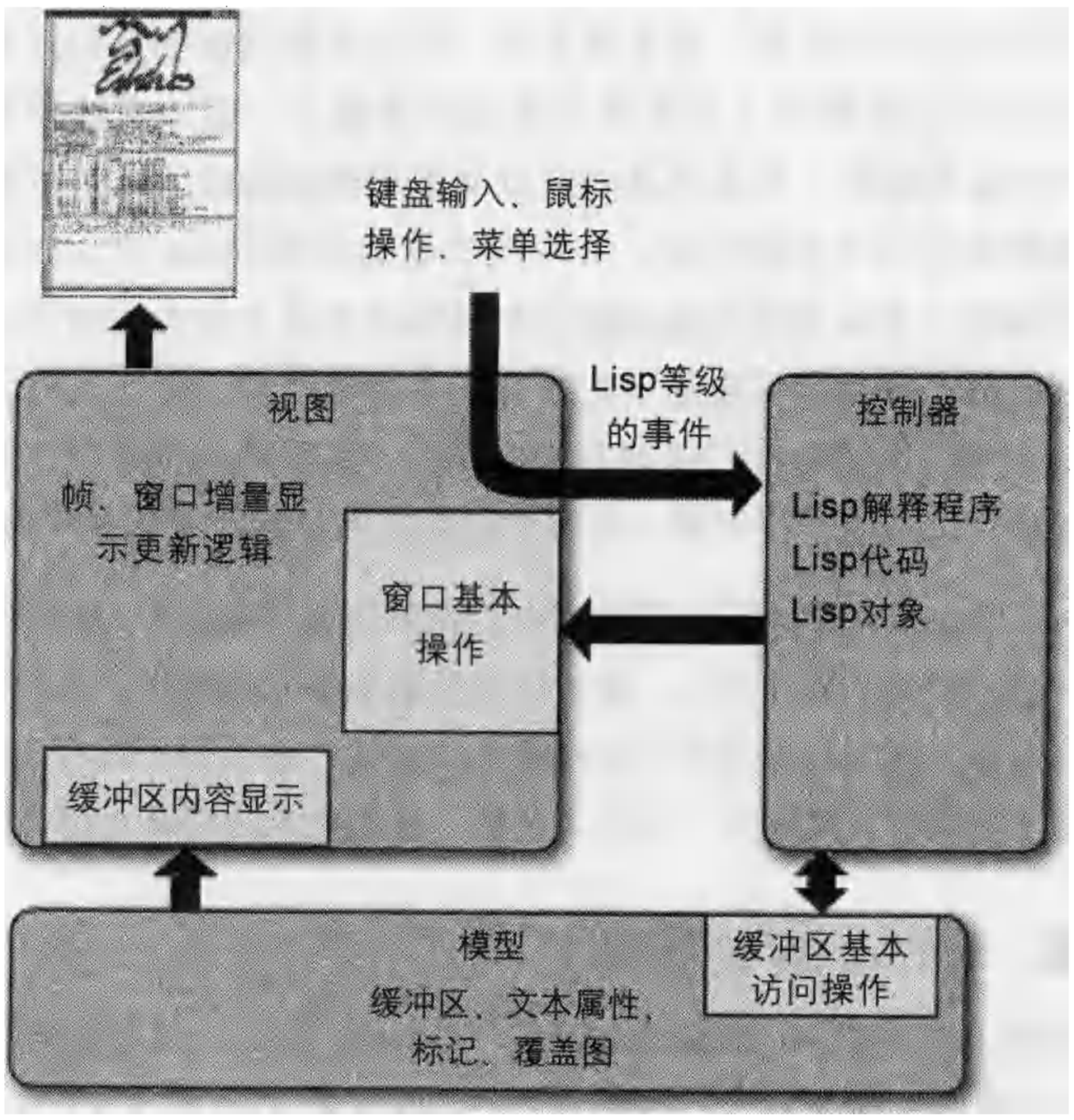


图11-2：Emacs中的模型—视图—控制器模式

每个缓冲区都有一种模式（mode），它用来指定针对特定类型文本进行编辑时的缓冲区行为。Emacs中提供了针对C语言程序、XML文本等大量类型的模式。在Lisp等级，模式将使用缓冲区局部键绑定来为用户提供特定于某种模式的命令，然后使用缓冲区的局部变量来维护缓冲区状态。通过它们的组合，这些功能使缓冲区的模式与对象的类十分类似：模式用来确定哪些命令是可用的，并根据这些命令所依赖的Lisp实现来提供相应的变量。

对于每个缓冲区，基本的文本编辑功能维护了一个“撤销”日志，它为撤销某些操作保存了足够的信息。这个“撤销”日志将记住各个用户命令之间的边界，因此每个用户命令都承载了用来撤销该操作的基本操作。

当Emacs Lisp代码对缓冲区进行操作时，可以通过整数在缓冲区文本中定位某个字符。这是一种简单的方法，但在对特定的文本块进行插入和删除操作时，必将导致表示位置的数字发生变化。要在缓冲区内容被修改时跟踪其位置，Lisp代码可以创建一个标记（marker）对象，它将随着文本的移动而变化。各种基本操作既可以接受以整数表示的位置参数，也能够接受标记对象的方法表示的位置参数。

Lisp代码可以为缓冲区中的字符添加文本属性。每个文本属性都将有一个名称和一个值，它们的内容可以是任何Lisp对象。从逻辑上看，每个字符属性都是相互独立的，不过对文本属性的表示法却在同样的文本属性中有效地存储了一组连贯的特性，并且Emacs Lisp还提供了一些基本操作，用来快速地定位文本属性被修改的字符位置，因此实际上文本属性还能够用来标记文本的范围。文本属性能够告诉Emacs该如何显示文本，文本如何应对鼠标的操作。文本属性还能够指定当鼠标放在某个文本上时可以提供什么样的键盘命令。缓冲区的“撤销”日志除了记录文本本身的改变之外，还将记录文本属性的改变。Emacs Lisp中的字符串也可以拥有文本属性：从缓冲区中析取一段文本并以字符串形式保存，并且可以将该字符串插入到缓冲区的任何位置，它将拥有相同的字符属性。

覆盖图（overlay）用来表示缓冲区中某些文字的相邻区域。覆盖图的起点和终点包含其覆盖的文本以及标记。和文字属性类似，覆盖图也会影响显示的结果，对于其包含的文本也能够响应鼠标的操作，但与文本属性相比也有不同之处，覆盖图不被视为缓冲区文本的一部分：字符串是不能拥有覆盖图的，而且“撤销”日志也无法记录对覆盖图终点的修改。

11.2.2 视图：Emacs重绘引擎

当用户编辑文本信息、在窗口上进行操作时，Emacs的重绘引擎将确保显示信息做出相应的更新。Emacs重绘引擎有两个重要的特性：

Emacs将自动更新显示

Lisp代码无法指定如何对显示结果进行更新。它可以自由地操纵缓冲区文本、属性、覆盖图，对窗口进行所需的配置，而不用关心显示结果最终将做出怎么样的改变。当进行这些操作时，Emacs的重绘代码将审视这些修改，然后确定一组有效的界面重绘操作，以便使得界面显示体现出模型的新状态。通过解除Lisp代码管理显示结果的职责，Emacs显著地简化了编写正确扩展插件的任务。

Emacs仅当等待用户输入时更新显示

一个命令将执行多个缓冲区基本操作，对缓冲区的任意部分进行修改，或者对多个不同的缓冲区进行修改。同样，用户可以调用一个事先录制好的、包含一系列命令的键盘宏。对于这种情况，Emacs不是快速地（看起来就像在闪烁）显示出缓冲区的中间状态，而是直到用户输入结束之后更新显示，只展示最后的效果。这样就避免了Lisp程序员尝试对缓冲区和窗口基本操作进行优化，以便展示出更平滑的显示效果。最简单的方法看起来也相当好。

通过更新显示，以自动、高效地体现编辑操作的效果，Emacs彻底地简化了Lisp创建者的工具。有些系统也借鉴了类似的行为模式，最有代表性的是JavaScript，嵌入在Web页面中的程序将对页面内容进行编辑，而浏览器将进行相应的显示更新工作。不过，很多

系统要求扩展插件认真处理扩展插件代码和显示更新代码之间的交互，这大大增加了扩展插件创建者的负担。

当Emacs最早出现时，它所针对的用户是那些通过串口线或调制解调器连接到计算机上的独立终端。这样的连接通常不是很稳定，因此文本编辑器是否能够找到一组合理的重绘操作来完成显示的更新操作，对于编辑器的可用性有很大的影响。Emacs在解决这方面的问题上花费了很多的心思，提供了一个更新策略体系，从“快速但受限”的策略到“工作更多但也很彻底”的策略一应俱全。后来还使用了诸如diff之类文件比较程序中常用的动态编程技术，以寻找一组最小的操作集，将原始的屏幕显示转化成新的屏幕显示。直到现在，虽然Emacs仍然使用这些算法来使更新工作最小化，但大部分努力都是多余的，随着CPU处理速度、网络速度越来越快，采用简单的算法也能够很好地完成这些工作。

11.2.3 控制器：Emacs Lisp

Emacs的核心是用自己独立的Lisp语言变体开发的。在Emacs实现的模型-视图-控制器模式中，Lisp代码扮演的是控制器的角色：你调用的所有命令，不管是通过键盘、菜单还是名称，几乎都是一个Lisp功能模块。Emacs Lisp是Emacs能够成功地随着其发展提供大量功能的核心基础。

Lisp的五分钟教程

第一次接触Lisp的开发人员总是会发现它是一种难以阅读的语言。其主要原因是Lisp提供的语法构造要比绝大多语言少得多，但基于这些语法构造的确能够获得更丰富的功能。为了让读者有些直观的感受，我们来看看将Python程序转成Lisp时的一些规则：

1. 程序控制结构的写法与功能调用类似。例如，`while x*y<z:q()`将写成`while(x*y<z,q())`。虽然表示方法修改了，但它仍然是一个while循环。
2. 中缀表达式的写法与功能调用类似，写出来的结果就像是一个名字古怪的功能函数。例如表达式`x*y<z`将写成`<(*(x,y),z)`。这里使用的圆括号只是用来分组的（而不是用来指定功能函数的参数），可以将多余的删除掉。
3. 现在所有的东西看起来都像是功能调用，包括程序控制结构和基本操作。将功能函数名称之后每个“调用”的左括号移到功能函数名称的左边，然后去除所有的逗号。例如，`f(x,y,z)`将写成`(f x y z)`。对于前一个例子，`<(*(x,y),z)`将写成`(< (* x y) z)`。

将这三个规则组合在一起，如果将用Python编写的代码`while`

x*y<z:q()用Lisp写出来，将变成(while (< (* x y) z) (q))。这就是正确的Emacs Lisp表达式，它和Python的表达式的含义是一样的。

当然Lisp的语法还有很多，不过这是它的本质。Lisp中的主要功能术语（诸如<和*）和控制结构（诸如while），在上面的例子中都有体现。

下面定义的是一组交互式Emacs命令，用来统计当前选择文本的字数。如果我告诉你“\\<”在Emacs中是用来匹配单词开始的正则表达式，你或许能够自己读懂这段代码的含义：

```
(defun count-region-words (start end)
"Count the words in the selected region of text."
(interactive "r")
(save-excursion
  (let ((count 0))
    (goto-char start)
    (while (re-search-forward "\\<" end t)
      (setq count (+ count 1))
      (forward-char 1))
    (message "Region has %d words." count))))
```

有一种观点认为Lisp为人类读者在阅读代码、理解代码方面提供的可视化线索太少，以致大家倾向于放弃这种乏味的语法风格，选择那些对基本操作、函数调用、程序结构控制予以更明显区别的语法风格。不过，Lisp中有一些十分重要的功能（我们稍后将会谈及）是依赖于这种一致性的；任何修改这种模式的努力都并不顺利。根据我的经验，许多Lisp程序员都感觉为让该语言能够如此强大、灵活，付出这样的代价也是合理的。

在Emacs中，命令就是程序员指定的负责与用户交互的Emacs Lisp功能函数（再加上定义之前的一些简要注释）。命令的名称，也就是用户在 Meta + x 之后输入的名称就是Lisp功能函数的名称。keymaps用来绑定执行该命令按键序列、鼠标点击操作以及菜单项。核心Emacs代码将会在keymaps中查询与用户按键和鼠标操作相一致的定义，然后将控制权分发给相应的命令。和Emacs的视图层所提供的自动显示管理机制类似，keymap分发操作意味着Lisp代码几乎不用负责对一个事件循环中的事件进行处理，这对于很多用户界面的开发人员而言大大减轻了工作负担。

Emacs及其Lisp语言有几个关键的特性：

- Emacs Lisp的机制是轻量级的。小型的自定义和扩展是很容易的：在你用户主目录下的.emacs文件中放上一行表达式，那么Emacs在启动时就会自动载入，可以载入原有的Lisp程序包，设置影响Emacs行为的变量，重新定义键盘序列等。你可以通过几行代码写出一个有用的命令，同时添加其在线文档。（要完成一个有用的Emacs命令的定义，请参见前面的“Lisp的五分钟教程”。）

- Emacs Lisp是交互式的。你可以在缓冲区中输入函数定义或表达式，Emacs将立即执行它。对这些定义进行修改，那么Emacs会马上重新执行它；你无需重新编译或重新启动Emacs。Emacs实际上就是Emacs Lisp程序的一种有效的集成开发环境。
- 你自己写的Emacs Lisp程序将是一等公民。Emacs中每个非凡的编辑功能都是用Lisp语言开发的，因此在你自己的Lisp程序代码中，同样可以使用到Emacs本身的基本功能函数和程序库。缓冲区、窗口和其他与编辑相关的对象，在Lisp代码中就是一个平常的值。它们可以作为功能函数的参数、返回值，还可以将它们存储到数据结构等。尽管Emacs的模型层和视图层组件都是硬编码的，Emacs称在控制器层没有为自己提供任何特权。
- Emacs Lisp是一种很完善的编程语言，很适合编写大型的程序（包含成千上万行代码的程序）。
- Emacs Lisp是很安全的。有很多人揭示了许多将导致错误产生的Bug成灾的Lisp代码，但它们并不影响Emacs。Bug引发的问题可以通过其他途径解决。例如，缓冲区中内建的日志机制可以让你撤销预期外的操作。这些Lisp开发给人留下更舒适、更积极的体验。
- Emacs Lisp代码很容易生成文档。在功能函数的定义中可以包含一个docstring（文档字符串），其中的文字用来说明该功能函数的意图和使用方法。Emacs提供的所有功能函数都有docstring，意味着对于指定设施的帮助不会比命令少。当Emacs显示一个功能函数的docstring时，还将提供一个指向功能函数源代码的超链接，使得Lisp代码更加容易阅读。（显然，docstring并不适用于大型的Lisp包，因此这些通常将放在更传统的结构化手册中。）
- Emacs Lisp不是一个模块化系统。取而代之的是通过内建的命名规范来实现在同一个Emacs会话中载入不相关的包时避免相互之间的干扰，因此用户可以相互共用Emacs Lisp代码包，不需要Emacs开发人员进行协调与确认。这也意味着在一个包中的所有功能函数对于其他包而言都是可见的。如果一个功能函数有足够的价值，就会有很多包使用它，并且依赖于其行为细节。

奇怪的是，这样的做法并没有产生人们所预期的大量问题。一种猜测是Emacs Lisp包之间独立性很强，不过在Emacs的标准版本中，大约包括1100个Lisp文件，其中有500个文件使用到了其他包里的功能函数。我的猜测是这些与Emacs一同发布的包的开发人员仔细地处理了与其相关包的兼容性，并且这些开发人员形成了一个紧密的小组，会对那些不兼容的修改进行相互的协商。那些不属于Emacs的包或许就无法做到这点，这些包的开发人员可能会收到加盟的邀请。

11.3 滋长的特性

Emacs不断滋长的特性是其架构的直接成因。下面是一个典型特性的生命周期：

1. 当你发现一个想拥有的特性时，尝试实现它是很容易的；Emacs的机制不官僚，从而使进入门槛很低。Emacs为Lisp开发提供了一个舒适的交互式环境。简单的缓冲区模型和显示自动更新机制，使得你可以只关注于手头的任务。

2. 当你定义了一个可以运行的命令之后，只需将其放在自己的.emacs文件中就可以使其永久有效。如果你经常使用这个功能，还可以用代码将其与一组快捷键绑定起来。

3. 最初你将从一个简单的命令开始，逐渐将演变成一组相互协作的命令集，这时你可能希望将它们收集在一起，封装到一个包中分享给朋友们。

4. 最后，有一些比较流行的包将纳入Emacs发行版本中，使其标准功能得到了扩展。

在内建的代码基线中，采用的处理过程是类似的。当拥有编写自定义Emacs命令的经历之后，就将更容易理解Emacs本身的代码了，因此当你发现现有命令中的潜在改进点时，则可以阅读该命令的源代码（前面说过，在帮助文本中就有相应的链接），然后尝试动手对其进行改进。当你在Emacs工作时重新定义Emacs Lisp功能函数，使得测试更加容易。当你修改完成之后，可以将修改版本放在自己的.emacs文件中以供自己使用，也可以为官方源代码提供一个补丁，以供大家使用。

当然，真正的创新想法是很少的。对于经验丰富的用户可言，对Emacs的改进想法会有类似的感觉，看看其提供的文档，可能会发现已经有人实现过了。由于有一群喜欢Lisp的Emacs用户，不断地对Emacs进行调整和修改，至今已经将近20年了，因此大部分人会发现自己所需要的各种功能，都已经有人做过相应的开发了。

不过，不管是通过添加新的包，还是打上用户贡献的补丁，Emacs的发展都是一个民众主导的过程，充分反应了用户的兴趣：存在的功能，不管是显然需要还是很少使用的，都是因为有人编写了它、有人认为它很有用。Emacs维护者的职责除了修正bug、认可补丁之外，还包括添加有用的新操作，换句话说，在社群中已经使用的包中选择最流行的、开发得不错的，将其添加到官方源代码中。

如果这就是全部，那么滋长的特性也就不成什么问题了。不过，它通常会带来两个方面的副作用：程序的用户界面将变得复杂、难以理解，程序本身变得难以维护。Emacs对第一个问题做了一些成功的处理，但对于后一个问题则彻底不管，相对而言更重视程序的功效。

11.3.1 滋长的特性和用户界面复杂性

在评价一个应用程序的用户界面复杂度时，有两个常见的维度：要维护的模型的复杂度，以及操作该模型的命令集的复杂度。

模型的复杂度

在用户自认为对应用程序的模型有所需的理解之前，需要花多长时间来学习？如果模型对用户而言是隐藏的，或者比较隐晦，会影响其应用吗？

微软的Word文档就拥有一个十分复杂的模型。例如，Word能够自动编译文档中一个小节或一个子小节的字数，自动对后续的文本片段进行编号，更新文本中对特定小节的引用。不过，如果需要这些特性按自己的预期执行，那么就需要对Word样式表有很好的理解。对于那些不会对文档内容产生可视影响的操作是容易出错的，例如如何避免对列表进行重编号（另外再举一个例子，你可以问一个文员，在Word 2003中如何“自动更新样式”）。

Emacs为了避免此类问题，采用了十分简单的方法：不支持样式表、自动编号小节、页眉/页脚、表格等在现代字处理软件中很常见的功能。它只是一个纯文本编辑器，Emacs的缓冲区只是一个字符串。在几乎所有情况下，缓冲区的所有状态只关心其外观。这样的做法，其带来的结果之一就是使人容易知道Emacs对其文件内容做了什么修改。

命令集的复杂度

在任何时候，发现相关的和有用的操作是否容易？发现一个还未使用过的功能是否容易？在这方面，Emacs的用户界面相当复杂。一个刚启动的不带任何定制的Emacs会话，提供了大约2400个命令和700个快捷键，并且随着使用过程，会话将载入更多的命令和快捷键。

幸运的是，新用户并不是突然面对这些东西，就像UNIX用户不需学习所有的shell命令一样。对于初次尝试Emacs的新用户而言，需要掌握的操作和使用带有图形用户界面的编辑器类似，用鼠标选择某段文本，通过键盘上的箭头键移动光标，通过菜单命令载入和保存文件。不熟悉这些命令，不会影响最基本的、可见的功能的使用。

不过，以这种方式使用Emacs得到的效果肯定不会比选择其他文本编辑器更好。成为专业的Emacs用户必须阅读操作手册和在线文档，以及学习如何有效地搜索这些资源。诸如grep、编译缓冲区中的内容、交互式调试工具、源代码索引等功能是Emacs与其他文本编辑器的主要区别。但是如果不使用它们，这些功能都是不可见的；同时如果你不知Emacs提供了这些功能，也就不会使用它们。

为了使这类探索更加容易，Emacs还提供了apropos命令集，你可以通过输入一个字符串或一个正则表达式来获得一组名字、与名称或文档字符串匹配的命令列表和自定义变量。

尽管它无法取代操作手册，但在你对要查询的命令有一些基本概念时，apropos命令是十分有用的。

在此类复杂性方面，Emacs用户界面具有命令行界面的很多共性：提供了大量可用的命令，用户并不需要全部操作它们（甚至很多都可以不掌握），并且为用户发现新功能付出了很多努力。

幸运的是，Emacs社区在命令的命名规范的建立方面做出了很大的努力，因此在不同包之间的一致性方面还是很好的。例如，几乎所有Emacs命令都是非模态的：对于移动、搜索、切换缓冲区、重新排列窗口等标准命令在任何时候总是可用的，因此你无需担心如何“退出”一个命令。再举一个例子，绝大多数的命令在要求用户输入参数时都采用了标准的Emacs设施，以确保不同包之间的提示行为是高度一致的。

11.3.2 滋长的特性和可维护性

显然，当你拥有越来越多的代码时，维护它就需要付出更多的努力。当一个开发人员的Lisp包被选入Emacs标准版本时，主维护人员将邀请这个包的作者一起参与Emacs的后续维护工作，因此随着包的不断增多，Emacs的维护团队也会相应扩大。如果某些开发人员放弃对包的维护职责，那么主维护人员必须再寻找一位志愿者，或者将这个包从Emacs标准版本中去除。

对于这个解决方案而言，关键在于Emacs是一个包的组合体，而不是统一定义的整体。从某种意义上说，从Emacs的动态维护效果上它更像是一个平台（例如操作系统），而不像是一个独立的应用程序。它不是由一个独立的设计团队来安排工作优先级、分配工作任务，而是由一个自主的开发人员组成的社区开发的，每个人都有自己的工作目标，然后通过一个选择和合并的过程，将他们的工作合并到发布版本中。最后，没有任何一个人能够承担整个系统的维护工作。

在这个过程中，Lisp语言扮演了重要的抽象边界的角色。和绝大多数流行的解释型语言一样，Emacs Lisp代码和Lisp解释程序细节、底层处理器架构是高度隔离的。同样，Lisp所提供的文本编辑类基本操作隐藏了缓冲区、文本属性及其他编辑所需对象的底层操作细节；Lisp中的可见特性绝大部分都取决于开发人员长期的贡献与支持。这为Emacs中核心的C程序代码的改进和扩展工作提供了很大的自由度，减少了破坏Lisp包中现有内容的兼容性。例如，Emacs的缓冲区在对文本属性、覆盖图、多字符集提供支持之前，对此前存在的特性提供了良好的兼容性支持。

11.4 另外两个架构

有很多应用程序都为用户提供了添加扩展的机制。从协作软件开发网站系统（如Trac插

件）到字处理软件（Open Office的Universal Network Objects），再到版本控制软件（Mercurial的扩展插件），都提供了扩展接口。在此我们将对Emacs和另外两个支持用户扩展的架构进行比较。

11.4.1 Eclipse

虽然很多人都知道Eclipse是一个流行的、开源的、针对Java和C++的集成开发环境，不过它实际上并没有提供任何功能。它只是为各种插件提供了一个框架，为支持各种特定开发功能（如编写Java代码、调试程序或者使用版本控制软件）的组件提供方便的通信机制。Eclipse的架构使开发人员能够将针对某个问题的有效解决方案集成到平台中，以得到一个统一的、功能齐备的开发环境。

作为一个开发环境，Eclipse提供了许多Emacs所欠缺的有价值功能。例如，插件Java Development Tools就对重构和代码分析提供了广泛的支持。与其比较，Emacs只是能够理解正在编辑的程序代码的语义结构，无法提供类似的支持。

如果Eclipse的架构不是处处预留端口，那么它将毫无价值，所有有效的功能都是由插件提供的。在此没有二等公民：最流行的插件也和其他一样，构建在同一个用户界面上。因为Eclipse为每个插件提供了对其输入和显示的底层控制，插件可以根据自己的目的自由地选择模型、视图、控制器层。

不过，这种方法也有许多缺点：

- Eclipse插件开发是不安全的，这是Emacs Lisp代码所具备的。一个充满bug的插件很容易导致Eclipse崩溃或死锁。在Emacs中，Lisp解释程序能够确保用户能中止失控的Lisp代码，在Lisp代码和模型实现之间的强边界可以避免用户数据被破坏。
- 由于Eclipse插件之间的接口相对复杂，因此编写一个Eclipse插件更像是为一个深奥精妙的应用程序添加一个模块，而不仅仅是写一段脚本。当然，这些接口正是实现Eclipse功能的基础，但插件的作者将不仅和大型复杂的项目打交道，还将涉及一些头脑简单的项目。
- 一个插件需要有足够的样板文件代码，在Eclipse中提供了一个帮助大家编写插件的插件。Eclipse Plug-in Development Environment（Eclipse插件开发环境）能够生成代码的骨架，生成默认的XML配置文件和manifest文件，同时还将创建测试环境中的回收（tear down）机制。此外还提供了一个自动生成样板文件代码的“向导”，它使插件开发人员更易于动手，但它无法降低底层接口的复杂性。

所有这一切使得Eclipse插件成为了一种不太优雅的扩展机制。插件并不是一个能够轻易制造完成的自动化任务，提供的用户界面体验也不太友好。

这让我想起在本章开始处提出的三个问题中的第二个，其中我认为值得考虑的就是关于插件机制的问题：在插件中可以使用哪类接口？它是否足够简单，只需通过脚本化语言就能够完成快速开发？插件开发人员能否在较高的抽象级别上开发，也就是更接近问题域？以及应用程序如何保护数据不被充满bug的代码破坏？

11.4.2 Firefox

在当前深奥精妙的Web应用（如Google Mail、Facebook等）中大量使用了诸如动态HTML、AJAX之类的技术，以提供更加流畅的用户体验。这些应用程序的网页中包含了在本地对用户输入进行应答的JavaScript代码，然后根据需要与底层的服务进行通信。JavaScript代码访问和修改网页内容时使用的是标准的接口，名为文档对象模型（Document Object Model，DOM），还有一些标准用来确定页面的显示效果。所有现代浏览器都在某种程度上实现了这些标准。

虽然Web浏览器不是一个文本编辑器而是一个浏览器，但它的架构和Emacs的架构却有着惊人的相似之处：

- 虽然Emacs Lisp和JavaScript在语法层面完全不同，但在语义层面却有许多本质的共性：例如Emacs Lisp和JavaScript都是解释型的、高度动态的、安全的。都是提供了垃圾回收机制的语言。
- 和Emacs Lisp类似，经常是先从一小段改变页面局部行为的JavaScript片段开始，然后逐渐发展成更加庞大、深奥精妙的应用。其进入门槛很低，但它也能够解决很多大型的问题。
- 和Emacs类似，页面显示的管理也是自动的。JavaScript代码只是对将在网页上展现的文本节点进行修改，浏览器将会根据需要及时地在页面完成相应的更新。
- 和Emacs类似，将输入事件分发给JavaScript代码的过程是由浏览器管理的。Firefox将负责确定事件是针对页面中的哪个元素，然后确定事件句柄并调用它。

不过，Firefox进一步应用了现代Web应用程序背后的思想：Firefox自己的用户界面是使用相同的底层代码实现的，它们负责显示网页、处理对其的交互。有个名为chrome的软件包集描述了界面的结构和样式，并通过JavaScript赋予其行为（注2）。该架构允许第三方开发人员编写一个对chrome包扩展的用户界面进行扩展的扩展（add-ons）。进一步采用相同的技术，开发人员能够替换Firefox chrome包所提供的标准界面，甚至彻底重新构建其用户界面，例如可以通过这样的方法使其适用于移动设备。

注2：显然在chrome中使用的JavaScript代码能够读写偏好选择文件、书签表以及原始的用户文件，从网页中下载的代码是不具备这些特权的。

与Eclipse插件类似，Firefox的chrome包中有许多元数据。与Eclipse Plug-in Development Environment（插件开发环境）插件类似，Firefox也提供了一个帮助大家开发Firefox插件的插件。因此在你能够扩展或修复Firefox之前，有大量重要的工作要做。不过，Firefox提供的自动页面显示管理和简化的事件处理机制使得Firefox插件的开发工作量要比开发Eclipse插件小得多。

Firefox开发人员更在改进其JavaScript实现的性能。这不仅对用户访问应用了大量JavaScript脚本的网站是很有意义的，同时也能够使Firefox开发人员将浏览器本身的更多功能从C++移植为JavaScript，它是一种使用更舒适、灵活性更高的语言。因此，Firefox的架构将更接近于Emacs，它自己的控制器层就都是用Lisp语言编写的。

这让我想起三个问题中的最后一个，我可以问各种使用过的插件开发语言一个问题：该插件开发语言是否是为该应用程序添加新功能的最好方法？如果不是，那么是什么原因导致了该限制？是语言本身的缺陷？是和模型层之间的接口过于麻烦？无论是哪种情况，相同的缺点或许会以相同的形式影响插件开发人员，使插件成为二等公民。（对于插件用户界面，也可以问同样的问题。）与Emacs类似，Firefox将其插件开发语言视为架构的核心，最主要的观点是该语言和应用程序之间的关系应该正确地设计。

作为一位热心的Emacs用户，我关心其未来。我对Firefox特别感兴趣的原因是它在很多方面看起来都与Emacs类似：一个实现自动页面显示管理的视图层，一个基于解释性的动态语言的控制器层，以及一个覆盖Emacs各种功能的模型层。如果有人乐于研究Emacs Lisp代码所积累的各种文献，那么可能会在chrome的基础上花一些时间开发出在架构上与Emacs很相似的文本编辑器，不过它将具有更丰富的模型，与当前技术发展结合更紧密。Emacs架构课题的最大价值在于教会我们学会不要遗忘。

	原则与特性		结构
	功能多样性	√	模块
√	概念完整性	√	依赖关系
√	修改独立性	√	进程
	自动传播		数据访问
√	可构建性		
√	增长适应性		
√	熵增抵抗力		

第12章

当集市开始构建教堂

KDE社区是如何发展ThreadWeaver和Akonadi项目的，以及它们是如何依次成形的

Till Adam

Mirko Boehm

12.1 简介

KDE项目是当今世界上最大的几个自由软件（注1）成果之一。在10多年的发展历程中，贡献者来自很多不同的人群，有学生、经验丰富的专业开发人员、沉溺于某种癖好的开发人员、公司人员、政府机构人员等，他们贡献了大量用来解决不同问题、完成不同任务的软件，不仅仅有完善的桌面环境（带有Web浏览器、群件、文件管理器、字处理软件、电子表格及演示工具），还有一些诸如天文学工具之类的专用应用程序。这些应用程序的基础是所有项目共同构建和维护的共享程序库。除了他们首选的KDE开发社区本身成员的东西之外，也使用了许多第三方开发人员的东西，包括商业的和非商业的，以创造出成千上万个附加的应用程序和组件。

虽然KDE项目最初的目标是为开源的UNIX操作系统（特别是GNU/Linux）提供一个集成桌面环境，但实际上KDE的范围是相当广泛的，它所包括的软件不仅仅是各种风格的UNIX中所拥有的，还包括许多在Microsoft Windows和Mac OS X甚至是嵌入式平台中所拥有的。这就意味着，在编写KDE程序库代码时将和许多不同的工具链打交道，应付不同平台的特性，以灵活、可扩展的方式整合系统服务，明智、谨慎地使用硬件资源。加上其程序库拥有大量的目标受众，因此其提供的API必须是易于理解、有用的，并适用于不同背景程序员。对于那些习惯于和Windows平台中的微软技术打交道的程序员，可

注1：　它通常也称为开源软件。

能会拥有不同的偏见和习惯，对于那些拥有Java背景的嵌入式开发人员、有经验的Mac开发人员也是如此。其目标是使所有程序员都能够舒适、高效地工作，使他们能够快速地解决问题，同时（从某种角度看也是最重要的）受益于他们的贡献，包括建议、改进和扩展。

这是一个很灵活、形式多样和相互竞争的生态系统，绝大多数贡献者都十分乐意通过协作来改进其软件，通过不断地相互评审来提升他们的技巧。大家可以自由地发表意见，争论可能十分激烈。做某件事总有更好的方法，代码总是会被很多聪明的人不断审查。在大学的课堂上，会有很多计算机专业的学生对其实现进行分析。许多公司会从中发现缺陷、发布安全建议。新的贡献者会通过对现有代码的改进来展示自己的技能。硬件厂商会将该桌面移植到手机上。人们对他们的工作以及他们完成工作的方法很感兴趣，并给它取了一个现在大家众所周知的称呼：集市。（注2）而且当遇到那些开发、维护大量程序库和应用程序的传统组织所面临的挑战时，它将显得更加自由和狂热。

有些挑战是来自技术方面的。软件将面对不断增长的数据量，而且数据将变得越来越复杂，个体和组织所需的工作流也是如此。当公司或政府管理者想实现产业转型时，还需要和他们使用的其他软件（开源的或私有的）实现整合，例如迁移到SOA（面向服务架构）时就需要做一些适应性修改。对于政府和公司的关键任务将会提出更高的安全要求，需要良好地自动部署大量的软件。新手、小孩、中老年人会有完全不同的需求，然而这些关注点都是有效的，并且是同等重要的。（注3）和其他产业相似，一般的开源软件和KDE的应用越来越普及，分布式处理的需求也不断提高。用户显然会要求更易于使用、更清晰、更优美的界面；和软件的交互响应更快、更好，而不是显得过于复杂；同时还希望系统高可靠、稳定，数据更加安全。开发人员希望扩展点对语言中立，API更好维护，提供兼容性更好的二进制程序和源代码，移植路径更清晰，和他们每天习惯使用的商业软件提供的功能更相近。

但对于开源软件项目而言，更容易让人畏惧的是社会性和协调方面的工作更加巨大。充分交流是非常关键的，它可能会受到地区、文化的阻碍，或多或少地反映了某些人的爱好和偏见，也可能只是受到了时空距离的影响。通过站立会议只要15分钟就能结束的讨论，在跨越半球的成员之间通过邮件列表的讨论可能需要花费好几天的时间。达成意见统一的过程可能和做出决策一样重要，它需要将整个团队的意见进行融合。显然，要让大家达成共识，需要更多的耐心、理解和说服技巧。它将征得更多精通数学、物理知识

注2：*http://en.wikipedia.org/wiki/The_cathedral_and_the_Bazaar.*

注3：它和完全由市场生成的软件是根本不同的。开源软件能够迎合价值不大的需求，例如，它无须保证所有努力是满足市场销售意图的，也不用证明能够收回投资。这也是KDE为什么比商业公司开发的竞争产品应用更广泛的一个重要原因。

的勇气十足的人的意见，而不是简单的意见修改，因为他们不喜欢和诸如附庸者之类的非程序员交流。基于这样的想法，气氛友好的Akademy年会总是拥有一个良好的体验（参见下一小节）。我们从来没有看到过这么庞大的、由来自世界各地的老年人和年轻人组成的团队。他们来自相互竞争的国家、多样的民族，甚至是来自于不同的世界，不同肤色的人们激烈地但又不失友好地在争论着与C++细节相关的问题。

除了技术、人文方面的话题之外，第三个主流的有影响的话题是开源软件的组织结构。在某些方面，组织结构对于开源软件团队来说是必需的。特别是大型的、知名的软件项目需要拥有商标，接受来自各方的贡献，组织年会等。当这段时间可以从与配偶相处和周末旅行中挤出来时，通常也是该项目被公众认识的时候。从某种角度来看，组织结构（或缺乏组织结构）将决定了社区的命运。到底是团结但略显迟钝，还是疯狂但略显混乱的模式，这是一个很重要的抉择。答案并不重要，因此它涉及了多种竞争项需要选择：采用基金模式还是自由影响模式；稳定的开发过程还是吸引才华横溢的穴居者；（注4）可视性、想法共享还是聚焦于给人留下深刻印象的技术产物。组织结构是必需的，不过在管理层级上将有很多不同的选择。有些项目甚至没有董事会，而有些则拥有宣称自己是乐善好施的独裁者。在成功和不成功的社区中都有最终形成该结果的例子。

在我们详细说明两个具体示例项目的技术、社会和组织问题以了解KDE组织之前，先介绍一些KDE相关的背景和历史是十分有价值的。因此在接下来的小节中，我们将介绍KDE社区当前的情况。

12.2 KDE项目的历史和组织结构

KDE或称为K桌面环境，最初是从失望中孕育的。那时FVWM是最常见的桌面环境，Xeyes是屏幕上总会出现的东西，Motif是与XForms竞争的工具集，它们大量吞噬了开发人员的脑细胞，而且没有什么吸引力，KDE的创立是由一个革命性的目标引领的：将UNIX的原生动力和漂亮、养眼的用户体验结合起来。之所以该目标能够达成，是由于挪威的Trolltech公司发布了第一个版本的QT，它是针对C++的面向对象GUI工具集的基础。QT为GUI程序员提供了一种系统的、面向对象的、优雅的、易于学习的、文档齐备的、有效的工具。1996年，当Matthias Ettrich还是Tuebingen大学的一名学生时，第一次提出了其于QT开发完整的桌面环境的想法。该想法很快就得到了约30名开发人员的响应，从那之后，该团队就不断稳定地发展。

KDE 1.0版本是在1998年发布的。尽管其功能从现在看显得不够强大，不过应该以当时的竞争对手作评价标准：那时的Windows 3.1还没有提供内存保护机制，Apple还在努力

注4：使用这个术语带有很多感情因素。无法否定，我很多亲爱的朋友总的来说将贡献一些阳光的、健康的养料。

寻找新内核，Sun还在努力弥补CDE中的缺憾。这时Linux还未被大肆宣传，开源软件的力量还没有被软件产业深刻理解。

甚至在完成KDE 1.0的同时，开发人员已经开始重新设计2.0版本的工作。还缺少很多主要的元素：组件模型、网络抽象层、桌面通信协议、UI样式API等。KDE 2.0是第一个经过理性的架构、设计、实现过程的版本。它曾经考虑过选择CORBA作为组件模型，不过最终还是放弃了。同时也是第一次出现团队动荡。有趣的是，虽然一些早期的核心开发人员离开了，但活跃的KDE开发团队的规模仍然在缓慢且稳定地发展。KDE e.V.，作为KDE贡献者组织，成立于1996年，并在1998年成为KDE的主要提交者。从一开始，该组织就致力于为KDE贡献者提供支持，但不主导KDE的技术方向。他们希望开发过程是活跃的开发人员的工作结果，而不是充分管理的团队。该理念已经被证实是最重要的因素之一，如果不是这样，KDE很难成为开源软件项目中少数几个不受发起人深刻影响的一个。外部支持在KDE中不被视为问题，但很多项目过度依赖于项目主要发起人，当他们离开或失去兴趣时将导致项目停止。正因为如此，KDE没有重蹈覆辙，成为了开源软件世界大力宣扬的项目，拥有着不断持续的开发动力。

在2002年4月，KDE 3.0发布。由于KDE 2.0被认为是设计良好的，因此3.0版本的开发历时6年，经过了5次主要的发布，最终朝着尽善尽美的方向不断演化和完善。基于KDE 3.0技术开发的重要应用程序成为开源桌面环境的标准：磁盘管理程序K3B；通用的、漂亮的音乐播放器Amarok；完善的个人通信套件Kontact。最有意思的是，这些应用程序最初并不是将KDE作为唯一的目标桌面环境的，但都将其作为构建最终用户界面的平台。从3.0版本开始，KDE开始分成两个部分：桌面和环境（通常也称为平台）。但由于KDE仍然受到X11的限制，因此这种分解并不容易被用户认可。这是下一步要做的事情。

在2004年，KDE团队进入了其发展历史上的一段艰苦期。Trolltech打算发布Qt 4.0版本，它和之前的版本以及市场上的其他同类工具集相比，有很大的增强，也有很大的变化。由于该工具集发生了很大的变化，因此从Qt 3.0迁移到Qt 4.0所需的不是修改，而是移植。关键的问题是KDE 4.0是简单地在KDE 3.0的基础上从Qt 3.0移植到Qt 4.0还是重新设计以完成移植。两种方案都有很多支持者，不管采用什么方法都明显引入大量棘手的问题，有大量的工作需要完成。最终的决定是重新设计KDE。虽然现在大家都认为这是一个正确的选择，但它仍然是一种危险的选择，因为这意味着在完成巨大的移植工作之前，KDE 3.0将成为扩展周期的主线。

Qt 4.0中主要的新功能需要做些特别的强调。最初Trolltech是对X11版本的Qt采用了GPL和商业双版权模式，现在延伸到了Qt所支持的所有平台，特别是加入了Windows、Mac OS X和嵌入式平台。因此KDE 4.0从某种意义上已经超越了UNIX世界。尽管UNIX桌面仍然是其根基，但为KDE开发的应用程序现在将可以运行在Windows和Mac OS X平台

上。这可能会引起一些争议。其中一个主要的观点是，为这些专有的桌面环境开发提供漂亮的应用程序，势必会降低用户从这些桌面环境中迁移到开源软件上的动力。另一个主要的观点是“什么才是我们应该关心的？”更客气一些的说法是“我们为什么要将有限的开发时间投入到非开源的目标系统上呢？”而支持这样做的人则认为，使这些应用程序在各种平台都能使用可以使这些专有平台的用户更容易迁移到开源操作系统上，并逐渐替代其关键的应用程序。这些争论最后的发展完全符合KDE长期以来的魔咒“最终的决定是由大家的工作做出的”。大量贡献者对在新目标平台上的开发产生了浓厚的兴趣，因此最终的结果是，没有任何理由可以剥夺那些期待以久的用户的需求。

为了实现平台无关，KDE 4.0进行了重架构，将其分成了应用开发平台、基于它构建的桌面以及相关的应用程序三部分。当然此时的划分还是很模糊。

这样的历史让我们了解了诸如KDE之样的项目解决问题的方法。这决不是单个开发人员能够解决的，无论他是多么才华横溢。如何能够使知识和经验积累起来，并使其发挥最大的作用？从战略角度思考，如何使一个大型的、来源不同的团队做出困难的决定，如何朝着全部的目标前进，而不会破坏其中的乐趣？如何融合各种观点，使最终的决定符合每个参与者的意愿，以及使KDE像其他开源软件那样成功？换句话说，当大家各自拥有自己想法的时候，如何一起构建大教堂，架构师如何承担起架构的角色，而不至于带上滑稽的帽子。我们之所以从贡献者说起，是因为我们认为开源软件社区首要的是社会组织结构，而不是技术结构。人是最稀有的也是最有价值的资源。

每个项目都有不同的角色，在招募新成员时都没有权威。每个开源软件项目都需要大量的对其宣传的男孩、女孩，以及一批有动力、有技能、务实的黑客、艺术家、管理员、作家和翻译者。使他们像是一个共同的集体的主要原因是他们都需要自发的动力、技能和自我的约束与管理。项目能够吸引不同年龄段的贡献者，从高校学生到退休人士。他们加入项目的动力是成为创建未来技术团队的一员极具吸引力，大家对这些技术感兴趣，并希望推动这些技术，能做更多有意义的事情，而不是写一些最终被抛弃掉的学术论文。

功能齐备的开源软件社区是一个极具竞争力的环境。我所遇见过的绝大多数商业性的工程师团队都有很多管制，并对雇员所作贡献的投资做了些政策上的保护。对于开源软件开发人员则不是这样。要将你贡献的代码集成到软件中，唯一的标准是质量，以及有人继续维护它。虽然源代码中可能会暂时存在一些平庸的片段，但在发展过程中都将会被替换掉。代码是公开的，经常被人仔细检查。即使有些代码实现已经拥有很高的质量，但在某些人眼里还不够尽善尽美，那么就会有程序员对它进行改进。由于创新十分频繁，开源软件的开发人员知道自己的作品经常会被别人改掉。这也能够说明为什么开源软件项目的代码质量通常都比商业性项目的代码高，你无法寄希望于代码中的缺陷不被人发现。

令人惊讶的是，很少编码人员会以被用户或媒体认可作为动机。经常，在最终交付之前

他们仍会对某些工作感到厌恶。和马拉松选手一样，其满足感来自于内部动机，也就是凭借自己的知识到达了终点线。有时，由于他们不太愿意向公众宣传而使大家缺乏了解，他们大多对歌功颂德没有兴趣。由于它是个人价值的体现，因此在软件项目中是十分自由的，甚至能够对开发目标设置优先级。在选择和决定其个人工作优先级时，代码艺术始终是重要的动机。软件总能完成，是源于有挑战的东西总能引起大家的高度重视。同时，大家对目标和持续引领结果的兴趣，要小于对过程的兴趣。其选择的路径往往和管理良好的商业软件所采用的路径不同，因为商业软件通常要在质量上向最后期限做出妥协。如果认同妥协是一个基本原则，那么通常认为是开源软件项目正处于向商业软件或“不够开源”项目转变的信号。出现这种情况，往往是主要的软件开发成员成立了一个公司来运行该项目，基金合伙人声称将控制主要的贡献者，或者项目本身已经停止了创新，逐渐转成了维护模式。

开源软件项目所承载的各种类型的组织结构都会遇到阻力。贡献者加入KDE项目可能有多种原因，但享受项目的官僚作风肯定不是其中的一个。加入开源软件项目的人，通常都十分清楚使用和保护其自由选择权。同时，大家都认可需要一种组织形式，虽然可能是不太正式的。虽然几乎所有技术挑战对于良好的团队来说都不是问题，但正式的组织结构仍然可以提高效率。许多开源软件项目之所以解散，都是因为策略上的范围决定是由最有影响力的团队成员独断专行的。寻找一种正式的组织形式，是解决项目面临问题的有效方法，此时如何不对未来的技术开发产生影响，是KDE发展历史上的一个关键（或许是唯一的）步骤。实际的情况是，KDE社区寻找到了一种稳定、被认可的组织结构，这对社区的长期稳定发展作了重要的贡献。

在1996年，KDE创建了代表KDE贡献者的KDE e.V.。一个e.V.，或称为“eingetragener联盟”是德国传统的非赢利组织，绝大多数的KDE贡献者当时都聚集在该组织中。驱使成立该组织的主要动力是该项目必须成为一个加入Free Qt基金会的合法主体。Free Qt基金会是Qt制造商Trolltech公司和KDE项目之间的约定，用来确保KDE永远能够将Qt作为免费的工具集来使用。它确保了即使Trolltech停止开发和发布Qt的免费版本，那么最后发布的免费版本的使用也可以无须获得Trolltech授权。由于Qt免费版本最早并不是采用GPL协议，因此该协议是十分重要的。直到现在，该基金会巧妙地解决了版权、开发人员权益等相关问题。该协议之所以重要，是因为KDE贡献者在投身于一个存在商业化风险的项目时难免会犹豫不决，另外KDE作为一个项目也必须拥有合理的权力。在此后不久，KDE e.V.还注册并持有了KDE商标，同时也承担起了举办KDE年会的职责。

关于KDE的架构和设计，该组织始终拥有很大的影响力，虽然它并没有直接管理整个开发过程。由于它假定所有“已完成任务”的KDE贡献者都是其成员，因此KDE e.V.的意见将具有重大影响。同时，它还是Akademy年度的组织者，这也是绝大多数贡献者唯一聚会和讨论的机会。它同时还会吸纳投资，主要是发起成员的会员费和捐赠款，因此还

能够对诸如sprint（译注1）或targeted会议之类的开发活动提供资金资助（有时也不提供）。该组织每年的预算和其产生效果相比，总是令人惊讶的少。开发人员之间的会议经常是自发的，它的资金起到了杠杆效应。积极的贡献者小组总会尽力完成工作，而无需一个概述性的目标描述。

Akademy逐渐成了一个众所周知的协会，甚至是独立于KDE社区的。因为绝大多数的贡献者的确需要一个个人级的会议。由于无法向任何开发活动分配人力资源或资金，很多范围很广的讨论都需要通过这样的会议来进行。通过这样的年会收集决策所需的信息，并进行宽松的协调，这已经成了常态。这个年会通常都是在每年夏天召开，其他开源软件项目的贡献者也会利用这个机会和KDE社区进行交流与协商。2007年的年会上所做的决定之一就是将发布周期改成6个月，这个建议是Ubuntu项目的Marc Shuttleworth建议和拥护的。

Akademy是KDE组织的唯一的全球性会议。除了这场大型会议之外，还有许多由子项目组举行的小规模会议和一些sprint会议。这些会议通常举办更加频繁、区域性更强，同时也更有针对性，因此Akademy通常讨论的是架构性问题，而特定模块或应用程序的设计问题的讨论则是通过这些小会议完成的。有些子项目组通常会在每3个月聚会一次，如KOffice和Akonadi开发人员。

在这些迭代的开发过程中还有相应的高等级或中等级的评审，它们相当高效，同时也能够使开发人员对目标和下一阶段任务有清晰的了解。年会的绝大多数参与者都从中获得了更大的动力，也有效地提高了开发工作的效率。

KDE的组织和结构说明当前对于一个开源项目而言，无法通过一个核心执行组来完成整个项目的组织。其结果就是需要一个迭代的过程来尝试和发现符合核心非技术目标的组织结构，以确保其开源，确保项目基业长青，同时顺应社区的发展需要。对于这个领域而言，自由不仅仅意味着自由的开发、创造开源的软件，同时意味着大家不会受到第三方参与者的干扰和影响。诸如公司、政府组织等外部参与者也会定期参加年会，它们对项目中的发现、经验和贡献也很感兴趣。不过这些利益相关方（Stakeholder）必须避免通过投入足够的资源来获得社区的更多投票权。这看起来有点痴心妄想，不过在其他项目中也曾出现过，KDE社区也明白。因此开源软件项目的永久和健康与是否确保项目的自由度是直接相关的。主要的技术目标，是开发人员和提供资金、物资、组织及其他资源的贡献者的支撑因素，也是比较容易达成的。

其结果是形式一个建立了开发过程的、有生命力的、活跃的、充满活力的社区，贡献者

译注1：　sprint是Scrum开发过程中的一次迭代，通常称为“冲刺”。

的变动是稳定和健康的，其过程中还充满了乐趣。同时这反过来也对新贡献者产生了更强的吸引力，它也是社区最重要的资源之一。KDE成功地避免了创建越来越大的规范化组织的趋势，同时也没有造就独裁者，大家都是尽可能地乐善好施。KDE是开源社区的原型。

为了理解该社区的实际功能，如何实现架构决策，以及独特的开发过程对技术性输出如何产生影响，我们接下来就进一步分析两个实例：Akonadi，针对KDE 4.0的个人信息管理基础设施层；ThreadWeaver，一个用于高级并发管理的小型程序库。

12.3 Akonadi

KDE 4.0平台，既是一个开发平台，也是一个依赖许多“支撑部件”的执行集成应用程序的运行时环境。这些关键的基础设施提供了一些核心的服务，其他应用程序可以简单地在一个现代的桌面环境中自由访问。它提供了一个稳固的硬件交互层，负责向桌面环境提供硬件的相关信息，诸如USB存储设备变得可用、网络掉线等。Phonon提供了一个在程序中很容易使用的多媒体层，通过它可以完成各种媒体文件的播放和用户界面元素的控制。Plasma则提供了一个丰富的、动态的、可缩放的（基于SVG）的用户界面，它要比标准的办公应用更加丰富。

用户的个人信息，包括电子邮件、约会、任务、日志、博客、收藏夹、聊天记录、地址簿等，其不仅包括大量的基础信息（数据的内容）。它还交织着丰富的上下文关系，诸如用户的偏好、社会关系，以及可以学习的工作上下文，这些对于桌面应用程序提供交互体验而言都是很有价值的，有效地利用这些信息将可以提供易用的、深入的、可靠的界面机制。Akonadi框架的目标是提供一个访问和操作用户个人信息、关联的元数据以及这些数据之间关系的服务平台。它将从不同来源收集相关的信息，诸如从电子邮件和群件服务、Web和网格服务以及它所触及的本地应用程序和缓存中收集，并提供相应的访问机制。Akonadi将成为KDE 4.0桌面环境的一个支撑部件，并且我们将会看到，它的目标甚至是超越于此。

在接下来的小节中，我们将介绍这个大型的、功能强大的框架的历史，了解它曾经做过的社会、技术和组织方面的努力，以前作者未来的愿景。同时，我们将进一步审视其技术解决方案，并说明选择这些方案的理由。

12.3.1 背景

在KDE发起小组的开发人员的早期交流中，就开始讨论桌面环境应该为用户提供哪些应用程序才算完整，处理电子邮件，安排日程，管理工作任务、地址簿都被认为是用户显然需要的，而且也是十分重要的。因此KMail、Kaddressbook和KOrganizer（分别是用

来处理电子邮件、联系人和工作任务的应用程序）就成为第一批开发的项目，很快也就提供了第一个可用的版本。用户的需求比较简单，接收电子邮件的标准模式是通过本地电子邮件池或POP3服务。电子邮件的内容并不大，电子邮件的文件夹通常是以mbox格式存储（每个文件夹以一个连续的纯文本文件存储）。电子邮件中HTML格式的内容对于KDE所针对的用户群体而言通常是不需要的，各种类型的多媒体内容是很少见的，加密和数字签名也是相当少见的。沿着这一思考线索，在地址簿、日历中的自定义数据格式都是基于文本的，全部信息都将以易于管理的形式存储。要编写一个能够被其他KDE开发人员，以及KDE第一次发布的用户而言，快速引用、功能足够强大的基本的应用程序还是相对简单的。

个人信息管理（PIM）应用程序的早期和持续成功，被证明对于后续的开发而言是一把双刃剑。随着因特网和计算机应用的不断普及，PIM问题领域开始变得越来越复杂。诸如IMAP之类的访问电子邮件的新形式，诸如maildir之类的电子邮件存储格式，都开始被集成进来。很多工作小组开始通过群件服务来共享日程信息和地址簿，或者通过如vcal/ical或vcard之类的新格式存储在本地。存储在LDAP服务器上的公司和大学的姓名地址录逐渐发展到数万条之多。希望使用KDE应用程序的用户开始认识到这些变化，并很快接受了这些新功能。加上参与PIM应用程序开发的贡献者并不多，其带来的结果是其基础架构并没有进行重新思考，也没有随着新功能的增加来进行相应的清理和更新，代码的总体复杂度在不断提高。诸如对电子邮件存储层的访问必须是同步的并且不支持并发，将整个地址簿读到内存中是合理的，用户不会穿越不同时区之类的基本假设在此时仍然是被认同的，也是开发工作的基础，毕竟这些修改所带来的成本是超出时间和资源限制的。这对于电子邮件应用程序KMail而言是绝对的问题，它将会造成后续的演化版本变得令人不快、难以理解、难以扩展和维护，而且庞大和功能臃肿。甚至，这些工作都是由不同的开发人员贡献的，没有人敢对其内核做太大的修改，害怕引发基于他们工作的开发人员的愤怒，这将使他们无法正常地阅读和编写电子邮件。

随着PIM应用程序使用越来越广泛地在不同领域使用，基于这些应用程序的东西也越来越多，不仅仅涉及最初的忠实的开发人员，还吸引了许多新的贡献者。越来越多KDE其他领域甚至是其他开源项目的成员都希望加入这个社区，以便降低学习曲线、减少错误使用的风险，例如它们影响一些企业级的加密功能，即使充分了解也无法自己对其进行测试。另外（或许是部分原因）还有一些非技术的因素，它会使社交气氛（KMail犹为明显）变得不太友好。讨论经常变得激烈、不友好，从个人偏好出发的批评变得越来越常见，尝试加入该团队的贡献者开始发现气氛不太友好。不同应用程序的各自工作几乎是相互隔离的。在KMail的维护团队中甚至出现了一些不太友好的争论。这些对于KDE社区而言是十分罕见的，这也使得其在社区内的声誉无法跻身第一流。

在不断发展的个人应用程序之间，对集成与融合的需求越来越多，用户也越来越希望能够通过一个公共的外壳应用程序访问电子邮件和日程组件，基于这些组件的个人应用也将发生一些变化。他们就开始着手实现更多的交互功能，在接口上达成了一致，就计划和开发进度做了一些交流，开始考虑诸如通过Kontact群件提供的保护机制来标识自己的成果、实现各自负责领域之间的命名统一。与此同时，由于有许多外部的、商业性的利益相关方开始对KDEPIM应用程序和Kontact产生了兴趣，因此纷纷推动KDEPIM社区采用整体的方法实现一致的交流，因此能够更可靠、更专业化地实现相互之间的交互。这些变化有效推动了PIM小组的发展进程，使其逐渐发展成为KDE中最紧密、最友好、最高效的团队。个人之间的差异和过去的怨恨都被抛在一边，新加入的成员彻底地改变了大家的态度。现在，绝大多数的沟通和开发方面的交流都是通过一个邮件列表完成的（<kde-pim@kde.org>），开发人员还可以使用一个公共的IRC频道（#kontact），当你问及大家的工作时，得到的回答一定是“KDEPIM”，而不是“KMail”或“KAddressBook”。

在企业环境中，个人信息管理是个关键应用。随着KDE应用程序在很多大型组织中得到了广泛的应用，使得专业化服务、代码修改、扩展、打包等（分发工作的一部分，主要是针对个人应用和部署的场合）需求都在稳定增长，这反过来了也创造了一个提供以上服务的企业生态系统。这些公司自然会首选雇佣最有资格完成这些工作的人，也就是这些KDEPIM项目的开发人员。正因为如此，对于KDEPIM项目中最主要的、活跃的代码贡献者而言，KDEPIM的开发工作已经成为其公司日常工作的一部分。他们中也有一些人不是直接从事KDE项目的开发工作，而是全职的C++和Qt开发人员。他们仍然是志愿者，特别是那些新的代码贡献者，不过核心开发小组都是由专业人士组成的。（注5）

由于PIM基础设施对于绝大多数计算机用户而言都是十分重要的，不管是个人还是商业环境中都是这样，KDEPIM技术决策过程的指导原则逐渐变得很实用。如果一个想法无法在合理的时间周期内达到可靠的工作成果，它就会被放弃。对于所有的修改，都会仔细地分析它对核心功能的潜在影响，这些核心功能在任何时候都必须处于可运作的状态。对于实验性的、有风险的决策，项目并没有提供太多的空间。从这个角度而言，该项目和绝大多数商业化的和私有的产品团队十分类似，它和KDE中的其他部分以及其他常见的开源软件有些不同。正如前面所说的那样，这样的做法并不永远是积极的，因为它必将会阻碍创新性和创造性。

注5：这是令人惊讶的，因为在我们的直觉中，开源软件都是由拥有大把自由支配时间的学生开发的，但近几年来的事实证明，多半成功的开源软件项目实际上都是由专业人士开发和维护的。Karim Lakhani、Bob Wolf、Jeff Bates和Chris Dibona发表的Hacker Survey v0.73（2002年6月24日，Boston Consulting Group发布，*http:// freesoftware. mit.edu/ papers/ lakhaniwolf.pdf*）就是一个实证。

12.3.2 Akonadi的演化

KDEPIM社区的会议很有规律，有年会以及其他大型的开发人员聚会，也有一些由5～10名开发人员参与的小型sprint会议。他们会通过几天的时间、聚焦于某个特定问题的方式进行讨论和编码。这些会议为讨论大家关心的主要问题、做出大的决策、制定路线、排定优先级等工作提供了良好的机会。在这些会议中，实现了诸如Akonadi的第一次发布版本、后续的稳定版本的架构奠定了基础。本小节接下来的内容将回顾一下本项目中最重要的决策点，我们将从得出第一个基础性观点的会议开始。

2005年1月，大家在这个每年的冬季会议上碰面时，KDEPIM的基础设施中的有一些部分初步显露出了不胜负荷的迹象。支持联系人、日历信息的多种后端实现的抽象KResources、电子邮件应用程序KMail的存储层中当时所做的一些基本假设都开始变得不再有效。它的假设主要包括：

- 想载入地址簿或日历信息的应用程序是十分有限的：也就是说，KAddressbook和KOrganizer是使用这些信息的主要应用程序。同样，它假设只有KMail需要访问系统中存储的电子邮件信息。因此修改通知、并发访问方面的支持是不需要的，或者是很少见的，也就没有提供正确的锁定机制。
- 数据总量是十分有限的。毕竟，用户通常需要管理的联系人、约会、任务会有多少？它假设需要处理的数量级是“百”。
- 只有C++和Qt程序库、KDE应用程序需要访问这些信息。
- 会有多种不同的后端实现“在线”工作，它们可以访问存储在服务器上的数据，而无需将大量数据复制到本地。
- 对数据的读/写操作是同步的、足够快速的，锁定用户界面的时间几乎可以忽略不计。

出席2005年年会的人们存在一些意见的分歧，有些人看到的需求是基于当前用户的实际使用场景的，而有些则是从未来的使用场景角度看到的潜在需求，这些东西可能是当前三个主要的子系统的设计无法满足的。数据量在不断上升，源于多个客户端的并发访问变得越来越多，越来越多的复杂的错误场景给系统的健壮性、可靠性带来了更大的压力，使事务存储层与用户界面彻底分离的需求越来越明显。在移动设备上调用KDEPIM程序库时，数据传输将通过低带宽、高延迟的网络上进行，网络连接经常是不可靠的，这些用户数据的访问能力要求已经超出了传统的在应用程序内部的访问，而是在桌面环境中各种的应用都会需要访问它。需要支持的访问不再限于C++和Qt中，还会涉及脚本语言、进程间通信、Web和网格技术所带来的访问需求。

虽然这些高级别的问题和目标曾经是无可非议的，有些开发团队遇到的痛苦是十分具体的，但对每个开发人员而言强度不尽相同，因此大家对如何解决当前紧急的挑战也有着

千差万别的意见。其中一个问题是当需要在地址簿中查询一个联系人信息时，需要将整个地址簿载入内存中，因此如果地址簿比较庞大并且包含图片和其他附件的话，速度就会变得很慢，同时会消耗很多内存。由于它是通过一个程序库来访问的，每个进程每次初始化这个程序库时就会创建一个独立的地址簿，而通常在KDE桌面环境中会运行电子邮件、地址簿、日程安排等应用程序，有时还会有约会提醒进程，因此内存中就经常可能出现4个甚至更多个地址簿副本。

为了修正在内存中载入多个地址簿副本的问题，这些应用程序的维护人员提出采用一种基于客户端/服务器模式的方法。更简单的方法是仅由一个进程负责在内存中保存实际数据。当地址簿被载入到内存之后，所有对该数据的访问都需要通过IPC机制，现在最常用的是KDE提供的远程进程调用基础设施DCOP。这样的做法还将隔离访问同一个联系人数据后台服务（如群件服务）的通信，还有人更关心原来的构架。虽然经过了冗长的讨论，关于内存覆盖区的问题已经通过该方法解决了，但还存在一些主要的问题。最显著的是锁定机制、冲突确定还需要在服务层上实现。同时由于现在针对日程安排的API是形式多样的（因此是基于指针机制的），所以对数据进行序列化是十分困难的，这要通过DCOP传输就需要对数据进行序列化。通过IPC接口传输数据始终是十分缓慢的，因此也会引发一些令人关注的问题，特别是用来访问电子邮件的服务，因此它需要提供一个建议信息。

对于内存覆盖区的问题，该会议讨论出了一种更好的解决方案，那就是通过更为灵活的、针对磁盘缓存和内存信息的共享机制，或许可以考虑使用内存映射文件来解决。鉴于改成基于客户端/服务器架构方案的复杂性，大家认为挑战很大，而且无法证明其带来的利益能够抵消妨碍系统正常工作（虽然现在的效果不太令人满意）的风险。作为一种可行的备选方案，Evolution团队采用了Evolution数据服务（Evolution Data Server，EDS）方法被认为是有价值的探索，Evolution是GNOME项目中开发的与Akonadi竞争的PIM套件，它是基于glib和GTK程序库栈，用C语言开发的。

在会议上提出数据服务观点的建议者有些失望，在接下来的几个月中没有太多进展。后来对EDS的基础代码进行了一些尝试性修改，其目的是为KDE调用该程序库提供支持，但最终发现这将涉及C和C++世界的连接问题，特别是思维模式和API样式都存在不同，这样完成的解决方案最好的情况也是不够优雅的，最差的情况则是不可靠、不完整。EDS使用的是CORBA，KDE最初采用的也是CORBA，只不过后来出于种种原因改成了DCOP，因此它并不吸引人。拒绝采用EDS作为KDE新的PIM数据基础设施的关键原因还是技术上的判断，同时也是一些对C的偏见，大家不喜欢C语言实现，认为它的可维护性不好，还有一些“这样做没有创新”的感觉。

到了2005年底，年初讨论的问题开始变得越来越紧急。在诸如桌面搜索代理程序、语义

标签、链接框架等都难以访问电子邮件信息。实现电子邮件处理功能的应用程序，都拥有自己的用户界面，在此可以访问搜索出来的电子邮件里的附件，可以打开并对其进行编辑。这些类似的可用性和性能问题开始增加人们对原有基础设施的忧虑。

在印度班加罗尔召开的年会上，负责Evolution的开发团队中的主要成员都参加了，我们就拥有了一次和他们讨论这些问题的机会，同时也可以向他们询问EDS方面的经验。在这些集会中，很明显看出他们正面临着KDEPIM团队所发现的相似问题，他们也正考虑着相似的解决方案，但都不认为在EDS的基础上进行扩展使其支持电子邮件，或者将其移植到CORBA架构上是可行的做法。来自Evolution团队的结论是：如果KDEPIM开发团队针对PIM数据访问构建了新的基础设施，那么他们至少会在概念层面共享其成果，如果它们不是以独立的C++和KDE程序库的形式提供，那么将可能共享其具体实现。

未来可能在整个开源桌面软件中共享这一基础设施，另外如果能够正确地完成该工作，那么将对很多开发社区而言都是有用的，而不仅限于KDEPIM，它也将为客户端/服务器设计思想添上一个重要的砝码。它将以协议或IPC机制的形式提供一个集成点，而不是一个需要链接到程序中的程序库，因此对于其他工具集、语言和开发范型都是开放的。从这一个在KDE和GNOME间共享PIM服务器的场景，我们也可以看到DBUS栈越来越可能成为跨桌面、跨平台的IPC和桌面总线系统，现在它也正被所有桌面环境项目所采用，并通过*Freedesktop.org*网站实现共享。

另外还有一个跨项目传播的有趣实例，有些PostgreSQL数据库项目的开发人员也参与了那年的年会，并参与了该主题的一些讨论。他们感兴趣的是如何有效地管理、访问以及查询结构复杂、规模庞大的数据，诸如用户邮件、事件、联系人等。他们认为自己的经验和其开发的软件都将成为这样的系统中的重要部分。他们带来了许多诸如搜索效率、类型扩展性设计思想等方面的观点，同时还提出了许多可操作性角度的想法，诸如如何备份这样的系统、如何确保数据的完整性和健壮性等问题。

几个月之后，在德国Osnabrueck召开的年会上关于PIM数据服务器的概念再次提到前台，这次有了更多开发电子邮件应用方面的支持者。这些人都是一年前置疑关于可能会影响性能、增加复杂性（注6）观点的人。它所引入的视角超出了KDE项目的范围，他们采用的实际上就是一年前大家所建议的方法之一，只不过当时大家都没有觉悟甚至没有尝试过，这些压力日增的团队关起门来搭建了一个骨架，最终使反对者们重新审视其观点，并认真考虑这一分裂性的修改。

在前一年的这些会议上，通过开发人员们的讨论，使得一个非技术方面的问题得到了答案。团队开始清楚地意识到他们最大的问题是缺乏新的开发人员加盟项目，而其背后的

注6：*http://pim.kde.org/development/meetings/osnabrueck4/overview.php.*

原因就是他们为项目开发的是难以使用的、令人不够愉快的程序库或应用程序。更可怕的是，当前的开发人员不可能永远承担贡献者的责任，而且他们无法将其学习到的东西传给下一代以保持项目的活性。该解决方案让人的感觉是对尽可能融合当前KDEPIM开发人员的经验和知识予以了同样的关注和努力，它可以成为KDE项目下一代贡献者的工作基础，而不仅仅是在构建一个更好的PIM工具。其目标是创建使开发过程更加轻松愉快、文档更齐全、神秘性更低、更现代的开发平台。它希望能够吸引新的贡献者，使PIM领域的创新工作不必陷于令人不快和复杂难解而又不得不用的基础设施中。客户端/服务器方法似乎能够利于该目标的实现。

要注意的是，这一基础性决策将彻底地改写KDEPIM的数据存储基础设施，这也意味着需要重新编写系统中很大部分的代码。我们接受了这一事实，它必将使得聚焦于维护当前代码基线的资源大大减少，以某种特殊方式维护4.0版本，毕竟这么大的一次重构在完成之前会发现很多的变故。也就是说，在KDE 4.0中实际上是没有KDEPIM的，因此将替换上一个有些粗糙但又十分必要的过渡品。

当团队认同这一决策之后，就提出了如下所示的使命描述：

我们的目标是设计一个可扩展的、跨桌面环境的PIM数据及元数据的存储服务，提供并发的读、写和查询访问支持。它将提供唯一的桌面范围域的对象标识和查询服务。

12.3.3 Akonadi的架构

在这次会议中制定的第一份设计草案中，就提到了许多在该架构的后续迭代中仍然存在的关键点。最重要的决策之一就是不使用DBUS来传输实际的负载数据，这虽然是Akonadi中最显然可选的IPC机制。取而代之的是使用了一个名为IMAP的、能够处理成批传输的、独立的传输通道和协议。之所以选择它，主要考虑到它能够对数据传输过程进行控制，因此可以取消冗长的数据传输，这样数据管道就不会阻塞控制管道。基于IMAP的数据传输的负载要比通过IPC机制传输小得多，因为该协议就是面向快速和流式传送大量数据而设计的。而DBUS的传输性能正是放弃它的主要原因之一，在其文档中就已经明确地说明它不是针对这种应用场景设计的。这将可以复用现有的IMAP程序库，在实现该协议时可以减少很多工作，不管是对KDEPIM开发团队还是未来想集成Akonadi的第三方开发团队均是如此。它还可以保留通用的访问邮件内容的功能，不仅仅是Akonadi特定的工具，还可以通过诸如pine或mutt之类的命令行电子邮件应用程序。如果保管在一个系统中的数据将被锁定，无法通过其他手段来访问，那么会增加用户的担心。由于IMAP只识别电子邮件，因此需要对该协议进行扩展，以对其他mime类型提供支持，不过仍然可以保留基本的协议兼容性。在选择数据传输协议时，还曾经讨论过http/webdav，它可以复用诸如Apache之类的现有http服务器，但这种方法并未得到足够的支持。

Akonadi的一个核心观念是为系统中所有的PIM数据和相关的元数据建立一个集中的缓存。然而老框架假定对后端存储的访问通常是在线式的，而Akonadi引用了本地副本机制，这样当需要向用户显示数据时能够马上提供，例如它可以保留许多可能已经获得的数据，以便避免不必要的重新下载。应用程序希望能够直接从内存中获得当前所需显示的信息（例如，在电子邮件应用程序中，只有当前文件夹中的少量电子邮件的头信息是需要显示的），而无需自己维护磁盘缓存。这样就需要使缓存是共享的和保持一致的，减少应用程序对内存的访问，并采用惰性加载（lazy-loaded）等优化技术，不过这样做用户无法马上看到数据。由于缓存是一直可用的，虽然可能不完整，但在绝大多数情况下仍然是可离线使用的。这大大加强了基于不可靠、低带宽、高延迟网络时的健壮性。

为了支持非拥塞性的并发访问，该服务器设计成了为每个连接保存一个执行上下文（线程），所有层要考虑的并发上下文的数量可能很庞大。这意味着针对状态的操作将引入事务性语义，需要进行严格的锁定，能够检查可以导致状态不一致的交叉性操作等。它同时也是选择管理系统内容的磁盘存储技术时所需考虑的关键约束，它也必须能够支持高强度的并发读、写操作。由于状态可能会在任何时候被当前会话（连接）所修改，因此需要一种通知机制，可靠、完整、快速地通知所有当前修改所影响的连接终端。这也是将低延迟、低带宽但高相关性的控制信息和高带宽、高延迟（由于存在潜在的服务器往返通信）、时间性不强的批量数据分开的另一个理由，这样就可以避免这样的通知信息被排在了一个很大的电子邮件附件之后之类的问题。如果应用程序是并发的，而在数据传输的同时还将发送带外的通知信息时则更重要。虽然这对于应用Akonadi的潜在应用程序而言并非主流的场景，但有足够的理由认为未来的应用程序将会有更多的并发，而不仅仅是因为存在诸如ThreadWeaver之类的工具，我们在本章的下一个小节中将介绍它。在针对KDE的Akonadi访问程序库中的一部分高等级的辅助类中就已经用到了这一功能。

Akonadi中还有一个基础性的视角，它在其第一次迭代的设计中就已经存在，那就是用来访问特定类型存储后端（如群件服务器）的组件将以单独的进程运行。这样做有好几个好处。与该服务器潜在的错误、缓慢或不可靠的通信都不会影响整个系统的可靠性。代理（注7）可能会崩溃，但不会使整个服务器崩溃。如果和服务器之间的同步交互更方便，或者从其他后端获取数据只有这一种方法，那么它将被阻塞而不会阻塞其他与Akonadi的交互。它们可以通过链接第三方程序库的方法，以避免对核心系统的依赖。它们可以独立地授权。它对于需要访问开源软件之外的程序库时十分重要。它同时也将正确地隔离Akonadi服务器的地址空间，因此减少了潜在的安全问题。这样第三方编写它也更简单，也能够更容易地在已运行的系统上部署、测试，无论什么编程语言都能够

注7：与Akonadi服务器交互并读、写其数据的实体称为代理（agent）。数据挖掘代理就是一个很常见的例子。特定用来处理本地缓冲和远程服务器间数据同步的代理称为资源（resource）。

匹配，同时也提供了DBUS和IMAP的支持。当然它也存在缺点，那就是必须对Akonadi存储空间中的数据进行序列化，存储空间是跨进程边界的。不过这在实践中很少考虑，其原因有两个方面。首先，它是在后台发生的，从用户的视角看不会对UI层面有任何干扰。其次，不管是本地已有的数据，还是在缓存的数据，或者从网络套接字（Socket）中获取的数据，当用户请求时都将把数据传给未解析的Akonadi服务器套接字上，在很多情况下甚至无需复制它。如果用户请求的数据是其中的一部分，那么就将能够以最快速度呈现，避免了过长的等待时间。在多数情况下，代理和服务器存储之间的交互性能并不是关键的。

从并发的观点看，Akonadi有两层。在multiprocessing（多处理）层，每个Akonadi用户（也就是一个应用程序或一个代理）都有自己独立的地址空间和资源获取上下文（文件、网络套接字等）。这些进程可以打开一个或多个到服务器的连接，每个都将分别以一个线程表示。为了平衡线程和进程的好处，最终决定都使用：进程更健壮、资源更独立、实现了安全的隔离是最重要的，而线程采用的地址空间共享机制是为了提高性能的需要，同时其代码也可以通过Akonadi服务器实现本身对其进行控制（而它被假定很少会引发可靠性问题）。

通过合理的努力为系统添加新数据类型的支持，也是最初希望实现的属性之一。从理想的角度看，data management（数据管理）层应该是完全类型无关的，针对每种类型数据（最初的电子邮件、事件和联系人，以及后来的RSS文件、即时通讯聊天记录、书签等）的内容和标签相关的知识应该集中放在一处。虽然最初对于实现该目标是很清晰的，但如何实现却没有明确的构想。当该目标实现时，核心系统本身以及访问程序库API都经历了几次设计迭代。在本小节中，我们还将对其最终结果做进一步说明。

在最初关于架构的讨论中，团队以很传统的方法构建了一个大蓝图，通过自底向下的层分离了各种关注点。他们讨论中在白板上绘制出来的架构快照如图12-1所示，在最底下是一个存储（持久）层，上面是访问它的逻辑层，再上面就是传输层和访问协议层，最后就是应用程序域。

虽然关于应用程序应该采用什么样的API来访问存储层（反对使用代理或资源分发来访问）有许多争论，有些人认为只需要一种访问API，所有实体都通过它来访问存储层，无论它的目标是提供数据还从以用户视角来使用数据。大家很快发现这种选择是更好的，它使得这一切更加简单、也更加均衡。代理所执行的任何操作，都是由数据变化通知触发的，这些都将由负责监控系统其他部分的代理获取。无论资源是否对应用程序有额外的需求（例如，能够向存储层提交大量数据而不会引发提醒消息风暴），实现统一API的努力仍然是足够通用的，也是足够有用的。无论如何，不管是应用程序访问还是资源的需求都需要使API保持简单，这样才能够使第三方能够提供额外的群件服务器后台，也

能够使应用程序开发人员接受Akonadi。针对性能、错误校正等特定情况的处理都应该尽可能放在后台处理。要避免出现提醒消息风暴，通过可配制的提醒消息压缩和更新监视系统来有效地解决它，这样系统的用户可以按特定的时间周期来订阅。

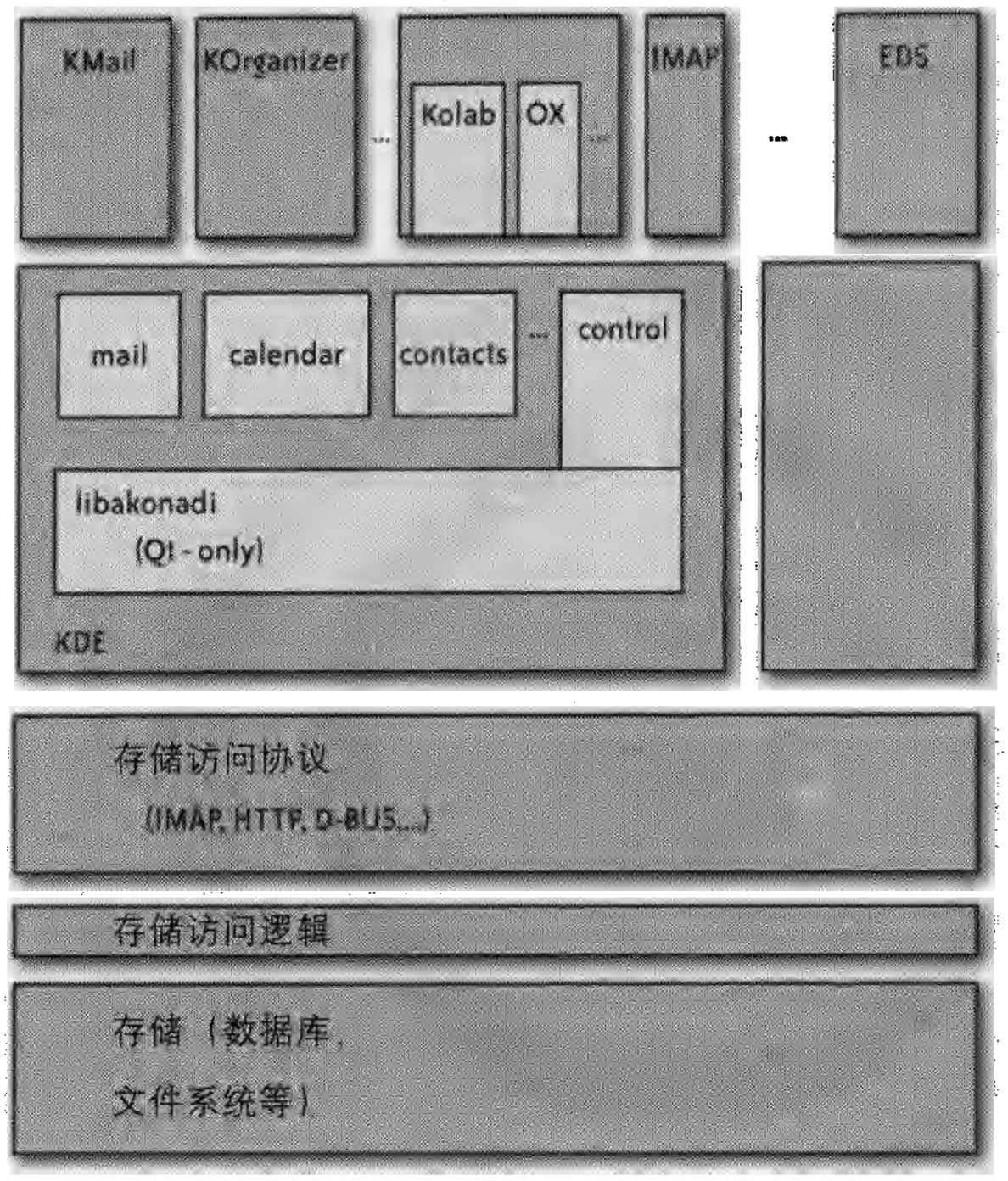


图12-1：Akonadi框架最初的层划分草案

图12-2所示的就是新版本的高层级架构图，它映射出来的想法是将系统的各个层描绘为同心圆或圆弧。

根据前面描述的需求，最底层（或最里层）使用关系型数据库构建显然是最简单的。至少针对实际PIM数据项（如获取时间、本地标签、每个文件夹的策略）的元数据而言，甚至没有考虑过其他解决方案，因为它们的名称是有限的、有类型的、结构化的，可以受益于快速、已索引的随机访问和高效的查询。而对于负载数据本身（如电子邮件、联系人等），它们都是存储在磁盘上的。这一决策并不是那么清晰明确。由于这方面信息的存储应该是类型无关的，如果需要在数据库表中存储的数据无法确定其结构类型，那么通常会被迫以BLOB字段形式存储。当涉及非结构化数据（从数据库的视角看）时，数据库的优势只能够发挥出一小部分。进行有效的索引基本上是不可能了，因为这需要能够对数据字段的内容进行解析。因而对这些字段进行的查询无法高效完成。而且预期的访问模式似乎也不是数据库所喜欢的：采用一种能够处理包含大量本地引用的连续数

据流是更可取的方法。对于某个数据项无法使用数据库，意味着对存储层的事务性操作必须手动实现。针对这个问题，大家提议的方法是对maildir标准定义的原则进行扩展，其本质是通过文件系统中原子性的重命名操作来实现无锁的ACID访问（注8）。对于最初的实现，决定采用数据库来存储所有的数据和元数据，并决定当以后能够更好地定义搜索需求时再对其进行优化。

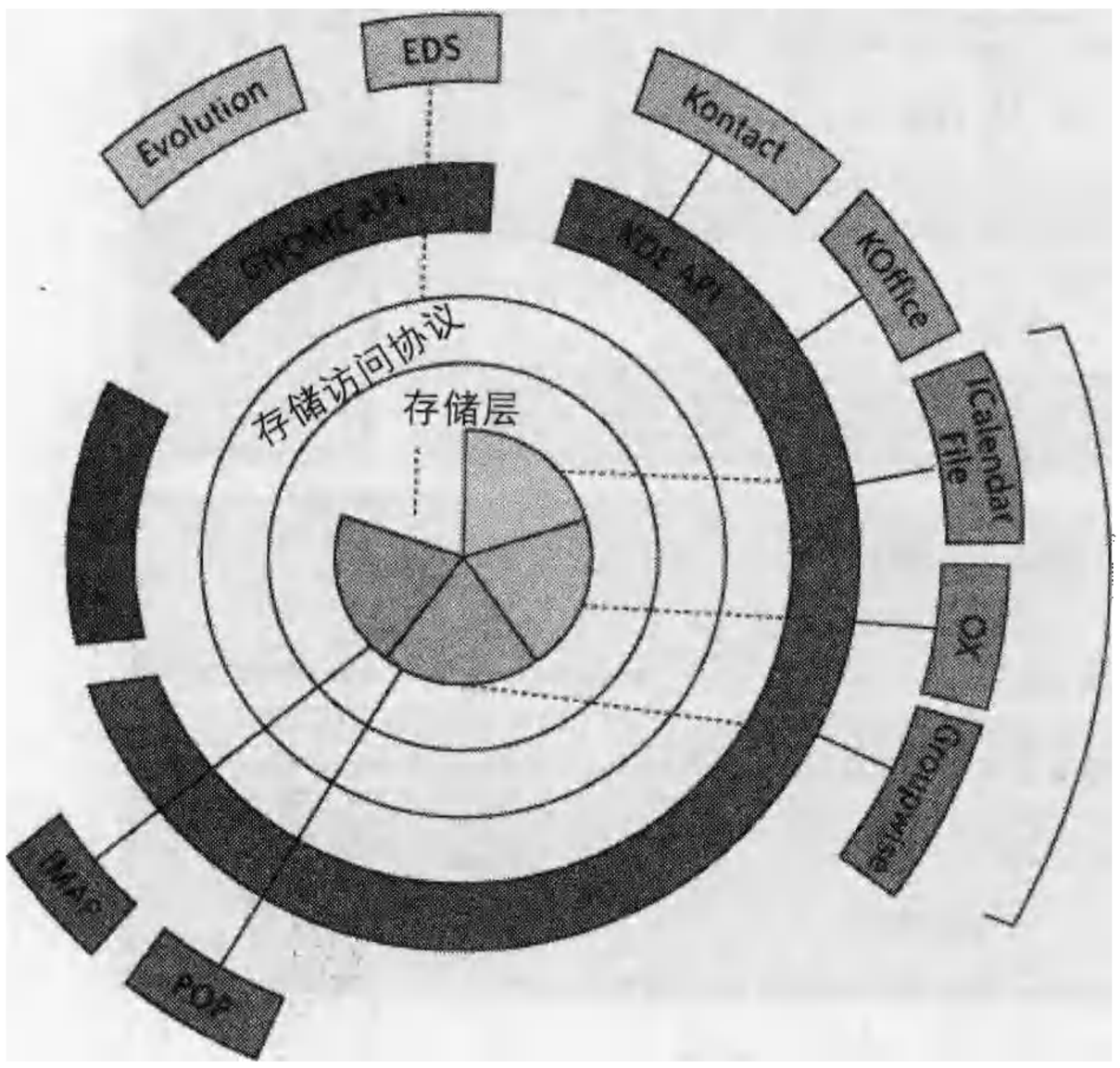


图12-2：从里向外，而非自底向上

将Akonadi存储服务变成存储用户个人信息的权威的缓存，它可以有效地提供和加强高级缓存生命周期管理。通过缓存策略的概念可以实现很精密的控制，包括应该保留哪些数据项、在多长时间内可以输出给用户，并提供了多种多样的使用模式。如果位于应用谱的末端（例如，在网络链接很差、存储很受限的嵌入式设备上），那么开发一个除了头信息之外其他都不存储的策略就很有意义，这样做是很轻量级的，而只等到需要显示时才下载完整的信息，不过对于已下载的数据项将仍然存储在RAM中，直到连接占用了过大的内存或者关机时清空了内存。对于笔记本电脑而言，经常会遇到没有网络连接或连接不可靠的情况，但通常会有很丰富的磁盘空间，那么就可以尽可能多地进行预先的缓存，以便在离线的情况下也能够使用，并且将每隔几天、几周甚至是几个月才清除一部分文件夹。而对于桌面工作站而言，通常都能够通过宽带Internet或本地网络连接到

注8：*http://pim.kde.org/development/meetings/osnabrueck4/icaldir.php.*

群件服务器上，可以快速地在线访问服务器中的数据。这就意味着除非用户出于备用或节省带宽的想法，希望在本地保留一个副本，否则就可以对缓存采用十分被动的策略，或许只保存已下载的附件以供本地索引和引用。这些缓存策略可以按文件夹、账户、后端进行设置，并且可以由服务器的一个线程中运行的线程来强制执行，对于数据库的低优先级的、常规的查询结果可以根据所有适用的策略进行清理。

在2007年的会议上，发现了架构中一个主要的缺失，那就是如何处理搜索和语义性链接。KDE 4.0平台已经通过Strigi和Nepomuk两个项目为扩散式索引（pervasive indexing）、富元数据管理和语义Web提供了一些强大的解决方案，将其集成到Akonadi是十分有趣的。不清楚的地方在于，是将负责向Strigi提供用来全面索引的数据的组件实现为一个代理，再通过一个独立的进程处理源于内核的消息提醒；或者为了获得更好的性能，将其集成到服务器应用程序中。由于至少全文索引信息会存储在Akonadi之外，那么就将引出一个相关的问题，那就是如何分解搜索查询，如何集成源于Strigi和Akonadi自身的查询结果，以及如何将查询传给能够提供在线查询的后端服务器系统，诸如LDAP服务器。同样，还讨论了应该使用什么样的策略来和Nepomuk划分职责边界，例如添加标签的工作是否可以完全委托给它。从某种程度上看，这些讨论仍旧在继续，其涉及的技术也随着Akonadi的发展而不断变化，而当前使用的方法仍然能够有效地应用到产品中。在本书编写时，有一些负责向Nepomuk和Strigi提供数据的代理，这些是独立的进程，它们和所有其他客户端、资源使用相同的API来访问存储层。进入的搜索查询将以XESAM或SPARQL表示，它们分别是Strigi和Nepomuk中的查询语言，在其他搜索引擎（例如Beagle）中也有相应的实现，然后将其通过DBUS转发出去。这一转发操作发生在Akonadi服务器进程中。其结果将通过DBUS返回，并以标识符列表的形式提供，Akonadi根据这一列表能够生成源于存储层的实际数据项，并将其返回给用户。此时数据层本身并不执行任何搜索和内容索引操作。

针对KDE的、C++访问程序库的API的发展成熟还需一段时间，最主要的原因是它还没有在程序库中清晰地解决服务器如何从类型无关的基础上输出多种类型信息。通过2007年4月的会议，它明确了对访问程序库进行扩展以支持新类型的方法，也就是被称为序列化器（serializer）的插件。这是运行时可载入的程序库，它能够将某种特定格式的数据（以mime类型标识）转化为二进制表示，以便以BLOB类型字段存储在服务器上，同时也能将二进制表示的数据转成特定格式的数据，还可以从序列化后的数据恢复成内存中的表示。它与添加新的存储层后台支持是正交的，例如需要添加一种新的数据格式，它是由一个新的资源进程（一个代理）实现的。这个资源的职责就是将服务器下发的内容转化为有类型的、内存中的表现形式，它知道如何处理这种表现形式。然后，再利用一个序列化插件，将它转化为二进制数据流，以便能够存入Akonadi存储库，并在访问该库时进行反向转换。这个插件还能够将数据分成多个部分，以便对局部访问提供支持

（例如只访问邮件正文或附件）。该程序库中的核心类是Akonadi::Item，它表示的是存储层中的一个数据项。它拥有唯一ID，它是一个全局标识符，用来标识在用户桌面中的实体，并作为语义链接的一部分，与其他实体关联（例如remote标识符）。它将映射到一个源存储位置、属性、一个数据负载，以及其他一些有用的基础设施，如标志位（flag）或修订计数器。属性和负载都是强类型的，用来设置和获取它们的方法都是模板化的。Akonadi::Item实例可以自己赋值，是易于复制的、轻量级的。数据项可以对负载和属性的类型进行参数化，而无需使用模板类本身。模板逻辑使其变得有些棘手，但得到的API是十分易用的。负载被假定是一个值类型，以避免赞成所有权语义的不清晰。当负载需要多态时就需要一个指针，或者当它已经是一个用来处理特定类型数据的、基于指针的程序库（例如libkcal程序库，它是KDE中用来实现事件和任务管理的），都将共享指针（如boost::shared_ptr），用来提供值的语义。为负载设置一个原始指针有助于检查模板的特化，还能够设置运行时断言。

下面的示例展示了在Akonadi中添加一种新的数据类型支持是多么简单，在通常情况下，已经有能够处理此种格式数据的程序库。它展示了联系人的序列化器插件的源代码，KDE库将联系人称为KABC::Addressee对象：

```
bool SerializerPluginAddressee::deserialize( Item& item,
                                             const QByteArray& label,
                                             QIODevice& data,
                                             int version )
{
    if ( label != Item::FullPayload || version != 1 )
        return false;

    KABC::Addressee a = m_converter.parseVCard( data.readAll() );
    if ( !a.isEmpty() ) {
        item.setPayload<KABC::Addressee>( a );
    } else {
        kWarning() << "Empty addressee object!";
    }
    return true;
}

void SerializerPluginAddressee::serialize( const Item& item,
                                           const QByteArray& label,
                                           QIODevice& data,
                                           int &version )
{

    if ( label != Item::FullPayload
      || !item.hasPayload<KABC::Addressee>() )
        return;
    const KABC::Addressee a = item.payload<KABC::Addressee>();
    data.write( m_converter.createVCard( a ) );
    version = 1;
}
```

通过设置负载类型的setPayload以及Item类中的hasPayload方法，开发人员能够直接、方便地使用其数据类型程序库中的原生类型。与存储层的交互通常表示为一个任务，这是command（命令）模式的一个应用。这些任务将跟踪一个操作的生命周期，提供取消操作的功能并能够访问错误上下文，并允许单步跟踪。通过Monitor类可以监控存储层中你感兴趣的范围内的修改，如某种mime类型、某个集合，或者甚至是某些特定数据项。下面这个源于电子邮件通知Applet程序的示例中展现了这些概念。在该例中，负载类型就是多态的，并封装在一个共享的指针中：

```
Monitor *monitor = new Monitor( this );
monitor->setMimeTypeMonitored( "message/rfc822" );
monitor->itemFetchScope().fetchPayloadPart( MessagePart::Envelope );
connect( monitor, SIGNAL(itemAdded(Akonadi::Item,Akonadi::Collection)),
         SLOT(itemAdded(Akonadi::Item)) );
connect( monitor, SIGNAL(itemChanged(Akonadi::Item,QSet<QByteArray>)),
         SLOT(itemChanged(Akonadi::Item)) );

// 开始下载最初的电子邮件，以显示第一封电子邮件
ItemFetchJob *fetch = new ItemFetchJob( Collection( myCollection ), this );
fetch->fetchScope().fetchPayloadPart( MessagePart::Envelope );
connect( fetch, SIGNAL(result(KJob*)), SLOT(fetchDone(KJob*)) );

....

typedef boost::shared_ptr<KMime::Message> MessagePtr;

void MyMessageModel::itemAdded(const Akonadi::Item & item)
{
    if ( !item.hasPayload<MessagePtr>() )
       return;
    MessagePtr msg = item.payload<MessagePtr>();
    doSomethingWith( msg->subject() );
    ...
}
```

12.3.4 第一次发布和未来

2008年1月，当开发小组再次聚集在寒冷、多雨的Osnabrueck时，就有开发人员展示了使用Akonadi的第一个应用程序，他们是受邀参加的。Mailody（KDE中默认电子邮件应用程序的一个竞争者）的作者在Akonadi能够帮他们开发出更好的应用程序之前就决定选择它，并且成为第一批尝试使用它的工具和API的团队。他们的反馈是十分有价值的，对于大家找出哪些过于复杂、哪里要进一步细化、哪些概念还没有良好的文档化或良好地实现都是十分有帮助的。Akonadi的另一个早期采用者Kevin Krammer也参加了会议，他曾经着手开始完成一些有意思的任务，希望能让KDE中针对PIM数据的遗留程序库进行修改，通过兼容的代理和资源来使其能够访问Akonadi。他所遇到的问题是如何在API中开些小口，并验证至少所有的功能能够在新工具中存在。

本次会议最值得注意的讨论结果是决定与IMAP在协议级实现向后兼容。它现在已经逐渐变得不再是最初的支撑Akonadi服务器能力的、仅限于电子邮件的标准，而是一个符合标准的IMAP服务器，承担起了电子邮件访问的功能，提供了更大的效益。IMAP协议是一个很好的起点，它的许多概念都在Akonadi访问协议中保留下来，但它不再有理由称为IMAP。或许在该服务器的后续版本中会回到该机制，或许会以兼容的代理服务器模式实现。

由于马上要发布KDE 4.1版本，因此在2008年3月大家都再次聚集在一起，以便在第一次公开发布之前再次对API做一次详细的评审，以便在可预知的未来能够保证其稳定性和二进制兼容。经过两天的讨论，出现了数量惊人的、大大小小的问题，有些文档缺少了，有些实现模式十分怪异，发现了一些不适当的命名模式，因此在接下来的一周内他们做了许多矫正工作。

在本书写作时，KDE 4.1版本马上就要发布，Akonadi团队兴奋地看到了KDE社区中许多应用程序和程序库的开发人员予以反应，这些都是其目标客户。有兴趣为不同的存储后面编写资源的人在逐渐增加，有人开始为Akonadi添加对Facebook地址薄、Delicious书签、MS Exchange电子邮件、基于OpenChange程序库的群件服务器、博客的RSS文件等的支持。现在很高兴地看到整个社区能够简单、深入、可靠地创建针对新数据格式的支持；能够高效地查询；能够对数据进行注释、实现数据的相互链接、为其创建语义和上下文；能够利用它使用户在其软件中完成更多功能，更好地使用它。

对于迄今仍未实现的优化工作，存在两种相关的观点。第一个是避免将负载数据以blob格式存储到数据库中，而是只在数据库表中存储指向文件系统的URL，而数据本身则直接存储在前面提及的文件系统中。采用这种方法之后，就可能避免将数据从文件系统中复制到内存中，避免通过socket通信将其分发到客户端（它涉及另一个处理器），从而导致虽然发布一份副本却在内存中创建了第二个副本。这样做之后，只需向应用系统传送一个文件句柄，以便其在内存中建立指向该文件的映射。无论如何，这样做不会影响健壮性、一致性、安全性以及API约束，整个架构的运转仍然和原来一样。另一种思路是使用MySQL中针对blob数据流的扩展插件，它承诺可以在保留通过关系型数据库API能获得的好处，同时实现与裸文件系统访问相似的性能。

虽然服务器和KDE客户端程序库会随着KDE 4.1同时发布，但其目标仍然是希望能够尽可能地实现开源软件领域的共享。最终，创建了一个放在Freedesktop.org上的新项目，在开发完成时服务器也就迁移到了这里。其DBUS接口的命名就已经采用了桌面环境无关的风格；该服务器唯一依赖的就是Qt程序库的4.0版本，它是Linux Standard Base（Linux标准基础）规范的一部分，它在Linux、Windows、OS X及嵌入式系统（包括Windows CE）中均采用了GNU GPL版权协议。下一个主要阶段将实现第二个访问程序

库，例如用Python编写，将会引入许多能够引入有价值效果的基础设施，也可能使用Java语言来编写。

12.4 ThreadWeaver

ThreadWeaver现在是KDE 4.0中的一个核心程序库。在此谈及它的主要原因是其起源在很多方面与Akonadi项目形成了鲜明的对照，可以提供一些有趣的比较。ThreadWeaver是负责并行操作调度的。该项目的提出时间就是大家觉得基于KDE所使用的程序库（也就是Qt）在技术上已经能够实现它的时候。看起来有许多开发人员具有这方面的需求，但只到Qt 4.0逐步成熟和逐渐变成主流时才出现。现在，在Koffice、Kdevelop等主要应用程序中都在使用它。它通常应用在高伸缩性、高复杂性的软件系统中，在这些系统中对并发性、带外（out-of-band）处理的需求越来越明显。

ThreadWeaver是一个并发作业调度程序。其目标是在多线程软件系统中管理和分配资源的使用。其第二个目标是为应用程序开发人员提供一个实现并行机制的工具，与那些在开发GUI应用程序时所使用的方法类似。这些目标都是高层次的，还有一些第二层次的、规模更小的目标：避免过强的同步，为协作的数据串行存取提供手段；充分利用现代C++程序库的功能，如线程安全、隐式共享以及线程之间的信号/槽连接（signal-slot-connections）等；将处理器元素和UI中相应展现的委托进行分离，以集成应用程序的图形用户界面；实现运行时的工作队列节流，以适应当前系统负载；使问题得以简化；等等。

ThreadWeaver程序库最初是针对那些开发事件驱动的GUI程序的开发人员而开发的，而后来变得越来越通用。由于GUI程序是由一个核心事件循环驱动的，因此无法在主线程中处理耗时的操作。如果这样做，那么在操作完成之前将会冻结用户界面。在某些窗口环境中，用户界面只能通过主线程绘制，或者窗口系统本身就是单线程的。因此要想实现反应迅速的、跨平台的GUI应用程序，最自然的方法就是在工作线程中执行所有操作，而在必要时通过主线程更新用户界面。令人惊讶的是，在用户界面上对并发性的要求远比直观感觉少得多，虽然在OS/2、Windows NT和Solaris操作系统都曾经强调过。多线程编程更加复杂，需要对编写的代码的实际运行过程有更深入的理解。多线程看起来是软件架构师、设计师已经良好理解的主题，但对软件维护人员和经验不足的程序员则并不容易理解。另外，有些开发人员认为对绝大多数操作而言，同步执行已经能够满足其执行速度需要，即使是那些从已挂载的文件系统中读取信息之类的操作，它要比CPU中进行处理的操作慢几个数量级。这些误解只是在特殊的环境下产生的，那时文件系统经常处于休眠状态以节约电力，但当文件系统位于网络中时一切都发生了改变。

在接下来的几个小节中，我们从基础概念开始，逐渐介绍该程序库的架构。在本章的末尾，还将说明它是如何找到进入KDE 4.0方法的。

12.4.1 ThreadWeaver简介：当其载入一个文件时有多复杂

要说服开发人员使用并发机制，就需要为其提供便利。以下是在GUI程序中执行一个操作的典型例子，将一个文件载入到内存缓冲区中，然后对其进行处理并显示相应的结果。在命令式程序中，每个操作步骤都是阻塞型的，其复杂较低：

1. 检查文件是否存在、可读。
2. 打开该文件读取。
3. 将文件内容读取到内存中。
4. 对它们进行处理。
5. 显示处理结果。

要提高用户友好性，可以在每一步骤之后在命令行中显示一条进度消息（如果用户选择了verbose模式）。

在GUI程序中，各种操作将以不同形式体现，因为在这些操作步骤中都需要能够更新屏幕，用户也希望提供一种取消操作的方法。尽管这听起来有点难以置信，甚至在最近的一些关于GUI工具箱的文档中还提出“减少事件检查”的方法。其观点是在执行上述步骤时周期性地检查事件，以完成必要的更新或取消操作。在这种场景下需要十分仔细，因为应用程序状态可能会出现意外的改变。例如，用户可能决定关闭程序，而不会注意到程序正在操作的调用栈中检查某个事件。简单来说，这种轮询方法从来没有良好地运行过，而且也是超常规的做法。

当然更好的方法是使用线程。但如何没有可用的框架来支持，通常会产生不可思议的实现。由于GUI程序是基于事件的，每一步骤都是由事件列出的，并且会有一个事件来说明该步骤已经完成。在C++中，通常会使用信号作为通知机制。有些程序会采用以下方法：

1. 用户发出载入某文件的请求，它将通过一个信号或事件来触发一个处理者方法。
2. 此时将执行打开并载入文件的操作，然后会链接上一个通知操作已完成的方法。
3. 在这个方法中将开始对数据进行处理，并链接上第三个方法。
4. 这最后一个方法将完成结果的显示工作。

这组层级式的句柄方法无法很好地跟踪操作的状态，并且通常会有错误的倾向。这同时也暴露了将操作和视图分离的缺点。不过，在许多GUI应用程序中仍然经常看到它的身影。

这种情况就是ThreadWeaver能够发挥作用的地方。使用了作业机制之后，其实现将采用如下所示的步骤：

1. 用户发出载入某文件的请求，它将通过一个信号或事件来触发一个处理者方法。

2. 在这个句柄方法中，用户将创建一个作业序列（该序列是一个作业容器，它将根据添加作业的顺序来执行各个作业）。他将添加一个载入文件的作业，以及一个处理文件内容的作业。序列对象本身也是一个作业，当其包含的作业完成时将会发出一个信号。直到此时，还没有开始执行任何操作；程序员只声明了需要按顺序执行哪些作业。当整个序列设置完成时，用户将其放到应用程序的全局作业队列（一个惰性初始化的单例对象）中。这时将通过工作线程自动执行整个序列。

3. 当收到序列发出的done()信号时，就表示数据做好了显示的准备。

这里揭示了两个方面的信息。首先，我们将一次性地声明所有要执行的每个步骤，然后再执行它。这将大大减轻GUI程序员的工作，因为它是一个开销不大的操作，可以很容易地通过一个事件处理者方法来执行。同步模式所带来的主要问题将在很大程度上被解决，而这是通常一个简单的约定实现的，那就是只有当队列化的线程准备好之后才会接触作业数据。由于工作线程都将访问作业数据，因此对数据的访问将是串行的，不过是以合作模式的。如果程序员想向用户展示进度信息，通过该序列在每个作业完成时产生的信号就可以解决（信号在Qt中可以跨线程发送）。执行过程中GUI仍然可以响应用户的其他操作，能够将作业从队列中删除，也可以取消所有操作。

由于采用这种方法实现I/O操作十分简单，因此程序员很快就采用了ThreadWeaver。它以一种很漂亮、便利的方法解决了这个问题。

12.4.2 核心概念与功能

在前一个示例中，我们提及了作业序列。接下来，我们来看看该程序库中提供的其他构造。

序列是作业集合的一种特定形式。作业集合是一个容器，其中排队了一组由原子性操作构成的作业，对于程序而言它们是一个整体。作业集合是复合体，也就是它本身也可以实现为一个作业类。在ThreadWeaver中只实现了一个队列操作，也就是获取某个作业的指针。复合的作业有助于使队列API最小化。

作业队列将可以确保其包含的作业以正确的顺序执行。如果声明两个作业之间有依赖关系，那么就意味着依赖作业只能在被依赖作业执行完成后才能执行。由于我们可以用m:n格式来声明依赖性，在这种声明格式中，可以建模相当多针对依赖操作所预期的控制流（它们都是有向图，无法定义作业循环）。只要执行路线是有向的，那么就可以完成作业甚至是作业队列的执行。呈现一个Web页面就是一个典型的实例，在处理HTML文档本身时，锚元素只会被识别一次。然后为获取和准备每个链接元素添加一个作业，而负责显示的最后一个作业将取决于这些准备性作业。此外，在此无需互斥机制。

这种依赖关系和诸如ThreadWeaver之类的调度系统相比，其区别在于它只是并行处理的一个工具而已。它有效地减轻了程序员关于如何最好地将每个子操作分配到线程中的压力。即使是采用如Future模式（译注2）之类的现代概念，程序员仍然需要决定操作执行的顺序。基于ThreadWeaver，工作线程会尽快地执行所有可能的作业，其中并不存在未确定的依赖关系。由于并发流的执行具有天生的不确定性，因此通过手动定义执行顺序，其灵活性是不可能很高的。在此可以应用更好的调度程序，计算机科学通常倾向于避免这一方式，而在经济学领域却经常使用随机系统分析来支持它。

你也可以通过优先级来控制执行顺序。它使用的优先级系统十分简单：队列中的每个作业都指定了一个整型的优先级，工作线程会首先执行优先级最高的那个。由于作业基类的实现应用了装饰器模式的，因此修改作业的优先级可以通过编写一个装饰器类来实现，这样既修改了其优先级又不用触及作业的实现。优先级和依赖关系的组合使用会产生有趣的结果，这在稍后就会提及。

ThreadWeaver并不直接依赖于队列行为的实现，而是使用了队列策略。当在执行一个特定的作业时，队列策略不会马上对其产生影响。相反，它影响的是作业的执行顺序，决定了工作线程会从队列中取出哪个作业。ThreadWeaver提供了两个标准实现。第一个就是前面提及的依赖关系。另一个就是资源约束。通过资源约束，可以声明所有已创建的作业中的某个特定的子集（例如，开销很大的本地文件系统I/O操作）在某一时间段内，只能执行一个特定的总量。没有这样的工具，很容易出现某些子系统出现过载现象。资源限制扮演的角色与传统线程机制中的信号机制类似，只不过它不会阻塞正在调用的线程，而只是简单地让某个作业不能执行。线程将检查某个作业是否可以执行，如果不行将会尝试执行另一个作业。

队列机制是为作业指定的，相同的策略对象可以指定给多个作业。同样，它们也是复合的，每个作业可能受限于多个可用策略的组合。从特定的策略驱动的作业基类中继承无法提供这样的灵活性。同样，采用这种方法的作业对象无需任何额外的策略，而且无法通过策略来影响其性能。

12.4.3 声明并发：略缩图阅读器示例

下面我们介绍另一个示例，用来展示这些不同的ThreadWeaver概念是如何整合在一起的。

译注2：Future在金融行业叫期权，市场上有看跌期权和看涨期权，你可以在现在（比如9月）购买年底（12月）的石油，假如你买的是看涨期权，那么如果石油真的涨了，你也可以在12月依照9月商定的价格购买。Future就是你可以获得未来的结果。对于多线程，如果线程A要等待线程B的结果，那么线程A没必要等待B，直到B有结果，可以先拿到一个未来的Future，等B有结果时再取真实的结果。其实这个模式用得很多，比如浏览器下载图片的时候，刚开始通过模糊的图片来代替最后的图片，等下载图片的线程下载完图片后再替换。

它将使用作业、作业复合、资源限制、优先级和依赖关系，完成在一个GUI程序中展示略缩图像。我们首先来看看实现该功能需要哪些操作，它们之间有什么依赖关系，用户希望如何展示其结果。该示例是ThreadWeaver源代码中的一部分。

在该示例中，假定载入某个数码相片的略缩图涉及三个操作：从磁盘上读取原始文件数据，将原始数据转成大小不变的图像，然后根据略缩图所需的尺寸进行缩放。可能有人会认为第二和第三步可以合并成一个，但它不是本练习的目标（不是流式载入图像数据），并且它将受限于该图像格式能够在载入时进行缩放操作。同时本例将假定这些图像文件存储在硬盘上。由于每个文件的处理都不会影响或依赖于其他文件的处理，所以所有文件的处理都是并行的。对于每个文件的三个处理步骤都是按顺序进行的。

但这并不是全部内容。由于这是一个图形用户界面的示例，还需要考虑用户的预期。在本示例中，我们假定用户对可视化反馈感兴趣，也就是希望提供可视的进度提示。当图像可用时，应该尽快地提供预览，以进度条的形式展示处理进度是一个不错的方法。用户还希望程序在执行时不会阻塞计算机，例如出现过度的I/O操作。

解决这个问题，可以使用不同的ThreadWeaver工具。首先，每个文件的处理可以实现为由三个作业组成的序列。这些作业都是很平常的，它可以作为应用程序中预先准备的作业类工具箱的一部分。作业包括（其类名与示例源代码是一一对应的）：

- FileLoaderJob，用来从文件系统中将一个文件载入内存中。
- QImageLoaderJob，用来将图像裸数据转换成Qt应用程序中的内存图像表示（使应用程序可以访问框架和在应用程序中注册的可用图像解码器）。
- ComputeThumbNailJob，用来执行图像缩放操作，使其大小符合预览的需要。

这些任务都将添加到JobSequence中，每个这样的序列将添加到JobCollection中。集合类的复合实现使其与最初的问题十分接近，因此程序员会感到十分自然和符合规范。

这解决了问题的一部分，也就是不同图像的并行处理，不过它很容易引发其他问题。采用这种方式声明的ThreadWeaver队列会存在问题，那就是无法避免当开始处理图像时尝试同时载入所有文件。尽管这是不太妙的，但我们没有告诉系统其他的处理方式。要确保只有这么多的文件载入操作同时开始，就需要使用资源约束。其代码类似于：

```
#include "ResourceRestrictionPolicy.h"

...

static QueuePolicy* resourceRestriction()
{
    static ResourceRestrictionPolicy policy( 4 );
    return &policy;
}
```

这些文件载入器只需在在其构造函数中应用该策略即可，如果使用的是泛化机制，那么则在创建时应用：

```
fileloader->assignQueuePolicy( resourceRestriction() );
```

但这样仍然没有正确地按预期安排好作业的执行顺序。该队列中可能最开始只立即启动4个文件载入器，但接下来将载入所有文件并计算生成预览（这再次将成为不太妙的行为）。在此需要更多的工具，需要更多的思考，要解决这一问题就将引出优先级。如果用ThreadWeaver行话来解释这个问题，那就是文件载入器作业优先级最低，但需要首先执行；图像载入作业优先于文件载入作业，但图像载入器启动前文件载入作业必须完成；最后，略缩图计算作业的优先级最高，虽然它依赖于其他两个处理阶段。由于这三个作业已经放在一个序列中，它可以确保针对每个图像的三个步骤会按顺序执行，将文件载入器的优先级设置为1，图像载入器的优先级设置为2，略缩图计算作业的优先级设置为3，这样就可以解决该问题了。这样，该队列会尽可能快地完成一个略缩图的显示，但在发现文件载入时间槽可用时仍然不会停止图像的载入作业。由于该问题主要是I/O的限制，因此意味着显示出所有图像略缩图的总时间只比从硬盘中载入这些文件的总时间多一点点（当然还有一些其他因素，如原始图像的分辨率极高）。在任何顺序式的解决方法中，其行为可能会带来更坏的结果。

大家可能会觉得以上关于该解决方案的描述比较复杂，因此提供一些源代码可能解释得更清楚。以下就是在用户选择了一组要处理的图像之后生成相应作业的过程：

```
m_weaver->suspend();
for (int index = 0; index < files.size(); ++index)
{
    SMIVItem *item = new SMIVItem ( m_weaver, files.at(index ), this );
    connect ( item,  SIGNAL( thumbReady(SMIVItem* ) ),
              SLOT ( slotThumbReady( SMIVItem* ) ) );
}
m_startTime.start();
m_weaver->resume();
```

为了提供正确的进度反馈，在任务添加完成之前将暂停具体的处理过程。只要一个作业序列处理完成，item对象就将发出一个信号以更新整个视图。对于每个选中的文件，将会创建一个特定的item对象，然后它将创建处理一个文件所需的job对象：

```
m_fileloader = new FileLoaderJob ( fi.absoluteFilePath(),  this );
m_fileloader->assignQueuePolicy( resourceRestriction() );
m_imageloader = new QImageLoaderJob ( m_fileloader,  this );
m_thumb = new ComputeThumbNailJob ( m_imageloader,  this );
m_sequence->addJob ( m_fileloader );
m_sequence->addJob ( m_imageloader );
m_sequence->addJob ( m_thumb );
weaver->enqueue ( m_sequence );
```

优先级将作为作业对象的虚属性进行设置。要注意这里设置的所有对象在添加到队列之

前是不会执行的，这一点很重要，在本案例中，只有当显式启动处理过程才会执行。因此创建这些队列和作业的整个操作过程只需花费很短的时间，程序控制权很快会交还给用户，一切都不会有什么问题。当预览图像可用时就会马上更新视图。

12.4.4 从并发到调度：如何系统地实现预期的行为

在前一个示例中，展示了如何分析问题以及如何实际地解决它（我希望这个部分不会令你感到惊讶）。要确保并发处理是用来编写出更好的程序，而不仅仅是提供一个填充线程的工具。它和调度是不一样的：调度需要告诉程序将执行什么操作，以及按什么顺序执行。该方法可以追溯到很早以前的PROLOG编程，有时需要采用相似的方法进行思考。一旦充分地理解了这一理念，其回报是十分有益的。

关于核心的Weaver类中的一个设计决策我们还没有谈及。Weaver类的API有两个互不相关的用户组。内部的Thread对象将访问它并从中获取要执行的任务，而程序员将使用它来管理其并行操作。为了确保公共API最小化，在此组合使用了装饰器模式和外观（facade）模式，以限制了向应用程序开发人员可能公开暴露的公共API的数量。并且通过PIMPL模式进一步实现了内部实现和API的解耦，它在KDE所有的API中都是十分常用的。

12.4.5 一个疯狂的想法

早在ThreadWeaver开始开发时就提出了该想法，但彻底实现该想法是不可能的。实际上最主要的障碍是框架的限制：在Qt程序库中使用了高级的隐性共享机制，也就是引用计数。由于这一隐性共享机制不是线程安全的，传输的每个简单POD（Plain Old Data）对象都是同步点。作者认为这对于用户而言是不切实际的，因此推荐在任何生产环境中使用Qt 3.0所提供的原型。KDEPIM套件的开发人员（也就是现在负责开发Akonadi的团队）真的认为ThreadWeaver是很不错的，并且马上将其初始版本引入到了KMail中，一直沿用到现在。由于已经遇到了ThreadWeaver承诺要解决的许多问题，KMail的开发者很迫切地采用了它，愿意承受其作者指出的一些缺点，甚至不顾他表达的一些愿望。

实际上，该程序库的一些未完成版就已经在KDE中获得了应用，当其发布了beta版本时就使大家纷纷快速地移植到Qt 4.0上来。因此它甚至在KDE 4.0开发周期之前就可用了，虽然它仍然是一个不属于KDElibs的次要模块。在两年的发展过程中，其作者为该程序库做了许多推介，提供了一些还在改进中的曾经简单或复杂的API。换句话说，它是一个在寻找问题的解决方案。多数KDE的开发人员需要时间来了解该程序库，这不仅仅是在理念上的了解，而且是促使他们回头做些投入甚至重新思考一些架构性结构问题，从而对其软件产生显著的改进。推进该程序库的发展进程的不是具体的开发人员需求；而是十分独立地不断发展，因为他相信相关的问题正在日益严重，同时为KDE 4.0平台提供一个好的解决方案是十分重要的。特别是在2005年的西班牙Malaga召开的Akademy年

会之后，越来越多的程序开始使用ThreadWeaver，包括KOffice、KDevelop等主流软件，这足以催使其成为KDE 4.0主程序库的一部分。

ThreadWeaver为问题提供了一种备选的方法，当程序库作者以及预期的用户社区开始在其自己的项目中采用它的时候，就逐渐进入了成熟期，很快也就被提议进入KDE 4.0的基础框架。此后社区成员的态度也就发生了改变，从原来适度的尝试转变成了赞赏其所带来的效果。这个示例展示了该社区在技术决策方面的有效性，以及将方法应用于实践时所采用的姿态。如果ThreadWeaver没有通过3～4年朝着KDE项目方向的努力，那么无疑不会比现在的结果更好。而这样的努力也包括KMail开发团队的过早采用。如果不是它最终获得了成功，那么难以想象基于KDE 4.0开发的应用程序能够更好地处理并发，并且为用户提供更好的体验。

ThreadWeaver后续的主要扩展是添加了能够可视化展示队列活动的GUI组件，并引入了更多预先定义的作业类。另一个思想是与操作系统的IPC机制相整合（实现全局资源限制等功能），不过这样的需求受限于跨平台的障碍。不同操作系统所采用的方法千差万别。KDE 4.0凭借着公认的可用性，获得了广泛的用户。由于ThreadWeaver并非为KDE定制的，因此它的下一步发展方向（Freedesktop.org？）还悬而未决。现在，其目标仍然是为应用程序和桌面环境开发人员提供可靠的并发调度功能。

第五部分

语言与架构

[redacted]

[redacted]

第13章

软件架构：面向对象与面向函数

Bertrand Meyer

支持函数式编程的主要观点是它能够更好地实现模块化设计。按照宣扬该方法的书籍中的阐述，特别是通过其针对金融契约的框架示例，我发现了它与面向对象设计相比的优点和不足。最终的结论是，面向对象的设计，特别是支持高级例程对象或“代理”（agent）的现代形式，包含了函数式编程，在保留其优点的同时，又提供了高层抽象，更好地支持扩展和复用。

13.1 概述

“美”作为软件架构的口号，并不是由旁观者来判定的。其实早就存在一些明确的标准（Meyer，1997）：

可靠性

该架构能否帮助我们创建出正确、健壮的软件？

可扩展性

应对变化是否很容易？

复用性

该解决方案是否具有通用性？或者甚至可以将其作为一个组件直接插入到新的应用程序中，而无需做定制开发？

对象技术的成功在于它能够使最终开发出来的程序在可靠性、可扩展性、复用性等方面获得很大改进，当然前提是你正确地应用了该方法，而不仅仅是使用了一种面向对象编程语言。

函数式的编程方法比面向对象编程方法出现得更早些，最早可以追溯到约50年前就已经很流行的Lisp语言。近几年来，它有卷土重来之势，出现了一些诸如Scheme（基于Lisp）、Haskell、OCaml和F#之类的新语言，以及精密类型系统（sophisticated type system）和诸如monads之类的高级语言机制。现在看起来，函数式编程甚至正在变成改进面向对象编程的新技术之一。在本章中，我们将从之前所概括的软件架构评价标准入手，对两种方法进行比较。我们将从反方向来论证它们之间的关系：面向对象的架构融合了函数式编程的思想，特别是吸纳了诸如Eiffel中的代理（agents，有的语言称之为闭包或委托）之类的新技术之后，在保留了其优点的同时克服了其缺点。

要证明这些发现是否合格，需要同时注意这项研究的限制条件，以及为缓和这些限制条件而提出的一些论据。这些限制包括：

较少的数据点

以下的分析将从两个函数式设计的实例开始。这可以让本章内容更具通用性。

细节不足

该示例的来源是一篇文章（Peyton Jones等 2000）和一个PPT幻灯片（Eber等 2001），我们后面再次提到它们时，将称之为“该文章”和“该幻灯片”（对本小节有帮助的还包括函数式编程的一篇经典论文[Hughes 1989]）。如果和那些更加详细的文档相比，使用该幻灯片可能会忽略掉一些细节。

具体关注

我们只考虑模块化。函数式编程也依赖于其他标准，如声明方式的优雅性。

实验者偏见

本章的作者长期以来一直是面向对象技术的贡献者和支持者。

以下的论据，将减少一些可能的批评：

- 函数式编程的示例源于行业的实践，具体来说，是一家通过函数式编程技术实现的应用程序来对外提供商业服务的公司。主要的示例是一个金融工具，用来解决金融行业所面临的复杂问题。根据该文章作者（一位该领域的专家）的说法，这个工具并不能很好地解决当前的问题。这暗示它体现了当前的技术发展水平。（本章中的第一个示例是一个关于布丁的理论性项目，是一个用于教学的试验品。）
- 该文章的作者之一，S. Peyton-Jones是一位底层技术研究领域的知名作者，它是Haskell语言的主设计者，是函数式编程领域的著名人物，因此该文章具有很高的

可信度。在13.3节中用了一篇文章作为补充的例子，这篇文章极有影响，是函数式编程社群中另一位一流人士J. Hughes所写的。

- 虽然有些地方值得探讨，但该文章中描述的解决方案提供了一些清晰、有效的解决方案。
- 在这些示例中，我们将不考虑可改变状态的概念，而这正是面向对象编程语言必须支持的特性之一。

在此还将关注诸如代理之类的机制，它是实现完整面向对象解决方案的基本成分，它显然是从函数式编程的观点中获得启发的。因此可以得出一个结论，那就是我们无法忽略函数式编程学派的贡献，只不过从论据中能够看出面向对象风格更适用于可靠、可扩展、可复用软件架构的定义，而这样的架构通常是面向对象和函数式技术的融合体。

关于下面的讨论还有一些要注意的方面：

- 在此使用的是Eiffel所提供的对象技术。我们并不打算分析除了多重继承（Java和C#就没有提供）、泛型（在这些语言的早期版本中都没有提供）、契约（除了JML和Spec#之外未提供）以及代理风格设施（Java未提供），或添加如重载、静态函数之后还有什么，这些东西都会影响面向对象的简单性。
- 讨论将围绕架构和设计进行。我们不管它叫什么名字，这些任务都与函数式编程（与对象技术类似）是相关的，因为“编程”并不局限于实现。Eiffel通过无缝开发的概念，明确地引入了从规格说明到设计和实现的连续过程。对于其中那些面向实现的属性，在此将不会详细地涉及，虽然它们在实践中也是十分重要的。
- 与实现有关的还有与表示式和表示法相关的问题。之所以考虑它们，是因为它们对架构和设计的关键标准有一定影响。不过，这些讨论在很大程度上只讨论语义的本质，而不讨论语义的形式。

还有两个更基本的提示。首先是术语问题，“契约”在此默认是指金融契约，它是与文档中的应用程序域相关的，别和软件中的“契约”（Meyer 1997）混为一谈，软件中的契约是指规约的元素（前置条件、后置条件、不变式）。为了避免混淆，这里将分别采用术语金融契约和软件契约来描述。

其次，我需要表示一下歉意：当在后半部分讨论到面向对象问题时，我将从作者先前的文章和幻灯片中引用或复制更多内容，而没有提出更多自己的判断和观点。这是因为面向对象技术已经有了广泛的应用，已经不缺乏command-query separation（命令-查询分离原则，参见第13.3.5节）之类的精妙的关键原则（我们的观点），在此只需做一些简单的说明即可。要想了解更完整的内容，可以阅读在此提及的参考文献。

13.2 函数式示例

该文章和该幻灯片的核心目标就是为描述、处理金融契约提供一个便利的机制，特别是针对那些非常复杂的现代金融手段，正如幻灯片中所提到的一个例子那样（数值会在不同时间、不同的主要货币之间呈现不同的关系）：

- “针对在12月27日支付2.00美元的承诺（期权的价格），持有者有权在12月4日选择：
- 在12月29日接受1.95美元，或
- 有权在12月11日选择：
  - 在12月28日接受2.20欧元，或
  - 在12月18日选择：
    - 在12月30日接受1.20英镑，或
    - 马上多支付1.00欧元并在12月29日接受3.20欧元”

（在本小节中，所有加引号的内容都是从该幻灯片或该文章中节选的。没有加引号的内容则是我们对其的解释和说明。）

为了解释这个问题，该幻灯片首先介绍了一个教学用的虚拟示例：它用“布丁”替代了“金融契约”。要想精确地描述布丁，可能需要涉及“计算所需糖量”、“估计布丁所需的制作时间”，并获取“制作的方法”。下面就是一种解决该问题的“坏方法”：

- “列出所有布丁（trifle, lemon upside-down pudding, Dutch apple cake, Christmas pudding）（译注1）
- 对于每种布丁，写下它的糖量、制作时间、制作方法等信息。”

虽然在该幻灯片中没有说明这种方法为什么不好，但我们能够很容易地猜出其理由：这是一个由特定描述组成的集合，不具有可复用性，因为它没有充分发挥“不同布丁可能共用相同原料”的特点；它也不具有可扩展性，因为如果要对某种布丁的某个部分进行任何修改，都需要对依赖这一部分的所有布丁进行修改。

这里的“布丁”是对我们真正感兴趣的“金融契约”的一个隐喻，不过由于它十分易懂，也不需要你事先掌握特定问题域的知识，因此我们还是先继续讨论完这个例子。对于这个问题，以下就是可行的“好方法”之一：

- “定义一个小型的‘布丁合成方法’集。

译注1：布丁是一种以面粉、牛奶、鸡蛋等为基料的糊状甜食；括号中列出的是不同口味的布丁，依次是屈莱弗甜食、柠檬布丁、荷兰苹果蛋糕、圣诞布丁。

- 用这些合成方法定义所有布丁。
- 糖量的计算也从这些合成方法中得出。”

图13-1所示的树形结构（节选于该幻灯片）展示了这些合成逻辑的结构。

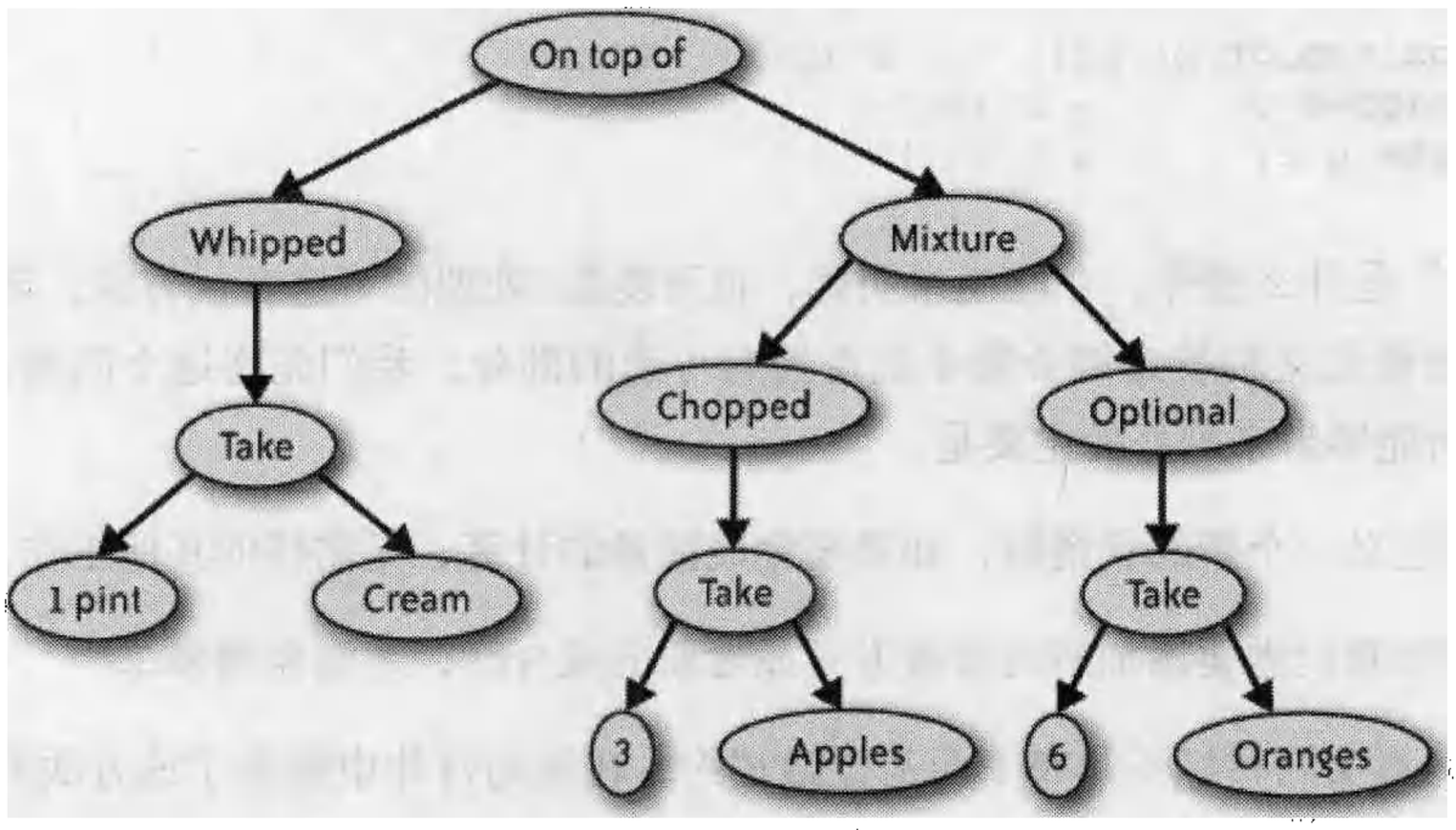


图13-1：描述布丁食谱的成分与合成方式

提示：我能够理解有些读者对于该示例中的甜点没有胃口，特别是因为它还出自于巴黎的作者。比较合理的解释是，该幻灯片是针对外国观众的，作者假定他们不熟悉度量系统（译注2）、不熟悉烹调习惯。而幻灯片中的讨论也是基于这样的假设：餐后甜点本身的味道不够好，并不会导致其语言和架构范型也随之不好。

在图13-1中，非叶子节点表示应用于其子树的合成方式。例如，“Take”就是一种合成方式，它需要两个参数，一个是布丁原料（左边的“Cream”，右边的“Oranges”），另一个是重量（“1 pint”和“6”）。最终这个树形节点在该应用程序的结果将成为一个布丁原料，或者由指定重量的指定原料制作而成的布丁。

它也可以通过一个微型的、“针对布丁描述”的“领域特定语言（DSL）来描述，得出如下所示的结构化文本（黑体字部分是为操作员添加的）：

```
"salad    = on_top_of topping main_part"      -- 为了保持一致，将其改成“OnTopOf"
"topping      = whipped (take pint cream)
main_part     = mixture apple_part orange_part
apple_part    = chopped (take 3 apple)
orange_part   = optional (take 6 oranges)"
```

这里使用了一个匿名的、但很典型的函数式编程表示法的变体（我们可以称之为“香草”），

译注2：　即米、千克等国际通用的度量单位体系。

在函数式的应用程序中通常写为函数变量（例如，plus a b在应用程序中表示将a和b相加），而圆括号只是用来完成分组而已。

有了这一基础，我们就可以定义合成方式（如案例分析中“糖量”）所需的操作了（它的定义与递归定义的数学函数一样，都将遵循相同的递归结构）：

```
"S (on_top_of p1 p2)     = S (p1) + S (p2)
S (whipped p)       = S (p)
S (take q i)      = q * S(i)
etc."
```

对于“etc.”是什么操作，在此尚未明确，也可能是S处理的可选合成方式；我们必须有一些方法来确定某种特定调合物中是否拥有可选的部分。我们先将这个问题放在一边，这种方法所能够带来的好处主要是：

- “当定义一个新的食谱时，如果要完成糖量的计算，无需任何其他工作。
- 只有当我们需要添加新的合成方式或者新的成分时，才需要增强S。”

当然，我们真实的目标不是布丁而是金融契约。在该幻灯片中提供了该方法的梗概，在其文章中则做了更加详细的介绍。它采用了相同的观点，应用到了更为有趣的元素、合成方式和操作的集合。

元素可以是金融契约、日期和可观测量（如特定日期的特定汇率）。在基本契约示例中，包含0个（可以在任何时候获取，没有权利与义务约束）或1个（c）针对货币c（立即支付给债务人一个单位的c）的契约。

针对这种契约的合成方式包括：or，如获取契约（or c1 c2），表示获取c1或者c2，只有它们都过期，契约才过期；anytime，如（anytime c），表示可以在c的过期之前的任意时间获取，只要c过期该契约就过期；truncate，如（truncate t c），表示除非c的过期时间比t早，否则将其过期时间提前到t；get，如（get c），表示在其过期日获取c。在该文章中，列出了一打这样的基本合成方式，还有一些可观测量和日期。通过它们可以定义出更高级的金融指令，如“欧式期权”可以表示为：

```
european t u    = get (truncate t (or u zero))
```

在一个操作中，包括契约的过期日期（这是所有建模努力所预期获得的最终的、重要的、具有实际意义的好处）和其值处理过程，以及一个按时间排列的预期值序列。与布丁中的糖量类似，在其案例分析中，该函数的定义是基于相同基本构造的。在以下的例子中，是一个包含前面的基本元素和合成逻辑的H操作，它用来表示过期日期函数，或者“范围”：

```
H (zero)    = ∞ -- 预期属性的一个特殊值
H (or c1 c2)    = max (H (c1), H (c2)
H (anytime c)    =  H (c)
H (truncate t c)    = min (t, H (c)
H (get c)    = H (c)
```

在值处理过程所遵循的规则中，采用了类似的结构，虽然右边的内容相对比较复杂难懂，涉及许多金融及数学计算。

13.3 函数式解决方案的模块性评价

在先前的介绍中，虽然忽略了该幻灯片和该文章中的许多观点，但对于讨论函数式方法的架构特性，以及将它们和OO视图进行比较，这些已经足够了。在后续的内容中，我们将会自由地使用“布丁”示例（它能使观点马上为大家所理解）和“金融契约”示例（真实应用程序的视角）。

13.3.1 可扩展性标准

正如前面的介绍中所说的那样，这种方法对于架构所产生的最直接的好处就是添加新的合成方式很简单：“当需要定义一个新的食谱时，无需更多的工作就能够完成糖量的计算”。不过，这只不过是使用函数式编程方法必然带来的效果之一。合成方式这一概念背后的本质是基于基本成分来创建布丁及布丁原料，或者金融契约，这些基本成分要么是原子性的，要么是应用合成方式所生成的成分。

该文章和幻灯片看起来像是针对金融契约问题的新思路，但实际上它也适用于其他问题。我们转到GUI设计领域，如果采用在该幻灯片之初否决的“坏方法”（列出所有布丁的类型，然后分别计算其糖量等），那么将需要以特定的方式，逐一生成能够执行显示、移动、大小调整、隐藏等操作的交互式应用程序界面。当然没有人会这样做；所有GUI设计环境都提供了一些原子性元素，如按钮、菜单项，而通过操作将它们递归式地组合成窗口、菜单以及其他容器元素，最终生成完整的用户界面。这就像布丁的合成方式定义了糖量、根据其成分计算卡路里值，金融契约的合成方式则按照复杂契约的要素定义了其范围和值序列一样，对组件的显示、移动、调整大小、隐藏等操作将针对组件中的所有成分递归执行。EiffelVision程序库（Eiffel 软件：EiffelVision文档）就以一种特殊的系统化途径应用了这些成分，不过这种方法十分独特。它当然并不仅限于函数式编程；任何带有子程序和递归机制的框架都能够实现它。

在与模块性相关的问题中，最有趣的并不是在将原有合成方式应用到原有类型的成分时出现的问题，而是当合成方式和成分类型发生改变时遇到的问题。该幻灯片明确指出，“只有当我们需要添加新的合成方式或新的成分时，才需要增强S”（针对合成方式sugar）。大家所感兴趣的问题是，接下来如何在架构中适应这些变化。

相关的变化实际上比其提到的更多：

- 不管是原子类型还是合成方式，我们都认为变化在操作中：为“布丁”应用程序添加卡路里计算函数，为“金融契约”应用程序添加“延期”操作，为图形化对象添加翻转操作。

- 除了添加各种操作之外，还要考虑到可能修改和删除操作。为了保持简单，我们只考虑添加新操作的情况。

13.3.2 函数式方法的评价

其提供的程序结构很简单，是由一组定义构成的，定义的格式要么形如（Arnout，2004）：

$$O\ (a) \ = b_a, O$$

要么形如（Arnout 和Meyer，2006）：

$$O\ (c\ (x,\ y,\ \ldots)\ \ = f_c, O\ (x,\ y,\ \ldots)$$

对于任何操作O，原子类型a以及合成方式c，右边都是由适当的常量b和函数f组成的。同样，为了保持简单，我们将诸如a之类的原子类型视为0-元合成方式，因此我们只需考虑第二种格式。对于合成方式t（on_top_of、hipped等）和操作f（糖量、卡路里等），我们需要定义t×f。

不管采用什么方法，t×f元素都必须提供。其架构性问题是如何将它们组织到模块中，以便对其进行扩展和复用。在该幻灯片和文章中都没有对这个问题进行讨论。当然，对于小型的t和f而言这并不是问题，我们可以将所有的定义都封装到一个独立的模块中。这是一种考虑可扩展性的简单方法：

- 添加一个合成逻辑c，并以上述格式在其中添加一个f定义，它是针对现有的所有操作的。
- 添加一个操作O，并在其中添加一个t定义，它是针对现有的所有合成方式的。

这种方法的可伸缩性不好，对于较大型的开发项目而言，必然需要将该系统分拆到不同模块中。现在的扩展性问题就将变成了如何使这样的修改影响的模块尽可能少。

即使t和f都很小，这种单模块的解决方案是无法实现复用的。如果其他程序只需要一部分操作和合成方式，那么就必将面临模块化技术中经常遇到的两难选择：

Charybdis（译注3）

复制-粘贴相关的部分，但这样做也有一些风险，当原始内容发生改变时（可能是修复一个缺陷），派生模块是无法做相应更新的。

Scylla（译注4）

引用整个模块，使用模块中一部分可用功能，最终担负起不必要的包袱，它会使更

译注3：西西里海岸附近墨西里海峡的一个漩涡，被拟人化为一个吞噬船只的海怪（希腊神话中的六头女妖）。

译注4：锡拉巨岩，正好位于著名大漩涡 Charybdis 对面。因此在英文中经常用“between Scylla and Charybdis”来表示进退维谷、进退两难、腹背受敌之意。

> 新工作更加麻烦，更容易产生冲突（例如，当派生模块定义了一个新的合成方式或函数，而原模块的最新版本也定义了与之重名的合成方式或函数）。

这些观测结果让我想起可复用性和可扩展性之间存在着紧密的联系。关于函数式编程语言OCaml（Steingold 2007）（注1）的在线评论中，有一个具体的例子：

> 在模块之外，很难对模块中的行为进行修改。假设使用了Time模块中定义的Time.date_of_string函数——用来解析ISO8601基本格式（“YYYYMMDD”），但如果还想能够识别ISO8601扩展格式（“YYYY-MM-DD”）。那么很不幸，只能让该模块的开发人员来修改这个函数，无法在自己的模块中重新定义该函数。

随着软件的发展和变化，另一方面的可复用性将变得十分关键，那就是对公共属性的复用。该文章在介绍“欧式期权”合成方式的同时还介绍了“美式期权”的合成方式，它们的函数签名是不同的（分别是Data→Contract→Contract和(Date,Date)→Contract→Contract），对于它们而言，所有的操作都必须分别定义。不过，这样做经常被置疑，因为这两种期权中有许多属性和操作是相同的，同样，不同的布丁也可以分成几种类型。这样的分类，对软件的模块化是有帮助的，同时还带来了额外的好处（如果相同的内容比较多），那就是定义中所需的内容将大大减少，要比t×f少得多。不过，这需要我们花些时间来研究问题域，不仅仅是从函数角度，而是从类型的角度。

这样的视图是位于更高抽象等级上的。在实践中，其中最容易引发争论的是函数及其签名的确定。根据该文章的说明，“美式期权要比欧式期权更加灵活。通常，美式期权授予的权利是在两个日期之间的任何时候行权，也可以不这样做。”这会让我们想起该定义的一个变体：要么美式期权是欧式期权的一种变体，要么它们都是某种通用期权的变体。不过，如果我们采用合成方式来定义它们，会马上让它们显得完全不同，因为在签名中将引入额外的日期值。同样，这也是一种基于具体实现来定义某一概念的方法，不管它采用的是数学计算实现还是计算机实现，这仍然意味着该解决方案缺乏抽象度和通用性。使用类型作为基本的模块化机制（正如面向对象设计中那样）将会提升抽象的等级。

13.3.3 模块性等级

将模块性作为评价函数式编程的标准之一是合理的，因为该方法所宣扬的主要优点就是提供了更好的模块性。我们在该幻灯片中看到了原作者对这个问题的说明，下面是来源于一篇针对函数式编程的、更为基础的论文（Hughes 1989）中所提到的更通用的描述，它对该方法的描述是：

注1：　此处引用做了些删节。并且这也不意味着作者认同该评论中的其他观点。

> [程序]可以通过新方法进行模块化和简化。函数式编程的主要价值就是大大改进了程序的模块化程度。采用函数式编程的程序员也必须努力确保程序使用更小、更简单、更通用的模块，并使用新的粘合剂（我们稍后就将介绍）将它们整合在一起。

在Hughes的论文中所提到的“新的粘合剂”，实际上就是我们在前面两个示例中所使用的方法之一，即无状态函数的系统化应用，包括高抽象层次的函数（合成生成），它们将对其他函数产生作用。此外，还有大量使用的列表、其他递归定义的类型，以及延迟计算（lazy evaluation，译注5）等概念。

虽然这很吸引人，不过这些技术仍主要致力于提供更为精细的模块化。Hughes用函数式编程方法开发了一个平方根计算程序，采用的是牛顿迭代法，它将计算数字N的平方根，公差为eps，初始近似值为a0：

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

在此除了使用了within、repeat、next等合成方式之外，作者还将该程序与使用了大量goto语句的FORTRAN程序版本进行了比较。即使我们不考虑这一价值不大的改进（在该论文最初发表时，FORTRAN语言已经略显过时，而且goto语句也已经被弃用了），大家也能理解为什么我们会喜欢这样的解决方案，它完全是基于一些通过合成因子粘合在一起的小型函数实现的，而不是使用循环结构。不过也有人更喜欢循环结构，但我们认为结构良好的程序要比大规模模块化更为重要，这比基础性架构问题更麻烦。在实践中，所展示问题的正确性实际上对于任何方法而言都是相同的。例如，在Hughes的论文中定义的函数里，序列中第一个元素和前一例子并不相同，它处理的是当值小于eps时的情况：

```
within eps ([a:b:rest])  =  if abs (a - b) <= eps then b
else within eps [b:rest]
```

似乎假定相邻元素之间的距离在减小，并且肯定假定这些差中有一个不大于eps（注2）。对这些特点的说明隐含了某种契约式编程的机制，将前置条件与函数联系在一起（在一般的函数式方式中没有这种机制）。确保eps终止的条件在本质上与命令行风格中对应循环终止的条件是一样的。

在这个例子和前面的例子中，该做法似乎对于大粒度模块化或软件架构没有什么贡献。在实际应用中，函数式这种无状态特性看起来对该问题没有什么（正面或负面）影响。

13.3.4 函数式的优势

在之前的例子中，还展示了函数式方法的四个重要优点。

译注5：采用此种方法的类将推迟计算工作，直到系统需要这些计算的结果。

注2：在引用Huges文章中的例子时，经他同意，我们使用了列表的新表示法（Haskell），如[a:b:rest]。这种表示法比最初的cons表示法可读性更好，以前需要写成cons a(cons b rest)。

第一个优点是表示法（notational）。毫无疑问，简洁的定义正是函数式编程语言吸引人之处，诸如前面示例中的定义就很简洁（正如大家所知道的那样，诸如Haskell之类的现代函数式语言提供了更加简洁的定义，例如cons a (cons b rest)可以写成[a:b:rest]）。这和常见的命令式编程语言中的例程声明相比，语法负担要小得多。不过，对于这一观测结论，有一些限制因素降低了其价值：

- 正如前面所说，从设计的角度看，表示法问题不是关键的。例如，有人可能会针对命令式语言使用函数式的设计。
- 诸如Haskell、OCaml等许多现代的函数式语言都是强类型的，这意味着表示法将会稍稍冗长；例如，除非设计师希望依赖于类型推断（这在设计阶段并非好主意），否则在合成方式within中将需要使用类型声明Double-->[Double]-->Double。
- 不是每个人都能习惯将多参数函数替换为返回函数的函数（就像医学文献中所说的“疯狂的嵌套综合症（RCS）”那样，也就是在那篇关于金融契约的文章中所使用的诸如(a→b→c)→Obs a→Obs b→Obs c之类的函数签名）。

不过，保持表示法的简洁在设计和架构层面上也是一种美德，而函数式编程语言也有一些关于其他设计表示法的经验教训。

第二个优点（Simon Peyton Jones和Diomidis Spinellis在对本章的早期版本提建议时强调了这一点）也与表示法有关，即定义对象的组合子表达式（combinator expression）的优雅性。在强制的面向对象语言中，与下面的组合表达式：

```
on_top_of topping main_part
```

对应的应该是一条创建指令：

```
create pudding .make_top(topping, main_part)
```

并带有一个创建过程（构造方法）make_top，通过指定的参数来初始化base和top属性。组合子的形式是描述式的，而不是强制式的。但在实践中，在面向对象编程中使用组合子的变种，利用“工厂方法”而不是显式地创建指令，是很容易也是很常见的。

另外两个优点则更为基础。第一个是能够让操作成为“一等公民”，这是一个很常用的说法，就像我们说“程序中的对象”或者“数据”是Lisp语言中的“一等公民”，也就是它是最能够被高效处理的部分。许多主流的语言都能够支持将一个例程作为另一个例程的参数，但这不能够被视为基础性的设计技术，而且实际上经常被视为自修改代码所带来的各种不确定性的根源。现代的函数式语言都将高级的函数视为常规的程序对象，并开发了相关的类型系统，以体现其强大。这也是函数式编程的一部分，它在采用主流编程方法的开发过程中展现了直接的效果；正如我们稍后将看到的那样，与代理相关的表示法是直接从这些函数式编程概念中派生的，这是在面向对象框架中很受欢迎的新机制。

函数式编程的第四个主要的优点是延迟计算（lazy evaluation）：它能够描述一个可能无限大的计算，当然该计算的任何具体执行过程都将是有限的。早先对within的定义就假定采用了延迟计算；这在repeat的定义中则更加清晰（在此使用的是Haskell的list表示法，而非Hughes原先采用的cons）：

```
repeat f a  = [a : repeat f (f a))]
```

谁负责处理（在普通的函数式应用程序表示法中）无限序列a,f(a),f(f(a))…（next N x的定义为(x+N/x)/2）呢，within的定义将会在处理了有限个元素之后，停止对该序列的处理。这是一个很不错的想法。它在软件设计中的通用应用程序会带来两个观测结论。

首先，这里存在正确性问题。这种可能出现无限循环的程序虽然容易编写，但它掩盖了确保其不会永远无法终止的困难。我们看到within中假定了一个前置条件，虽然没有直接描述出来，但该前置条件要求使该元素的值小于eps，在无限序列（它是半可判定的）上是无法执行有限计算的。这对于设计师而言是一种比较麻烦的技术，这也是为什么许多采用函数式编程方法的程序员想寻找新的“明灯”的原因。（这是很难预先知道的。当有任何采用函数式编程方法的程序员改变时，就可能尝试换一盏“灯”，如果失败了，则还会尝试另一盏。）

第二个也是最后一个问题，可能会在非函数式的设计环境中对无限结构采用延迟计算，不需要任何特殊的语言支持。而抽象数据类型方法（也就是众所周知的面向对象设计）则提供了合适的解决方案。在Eiffel程序库中，有限序列和列表都可以通过API来访问，这种访问是基于“光标（cursor）”这一表示法的。

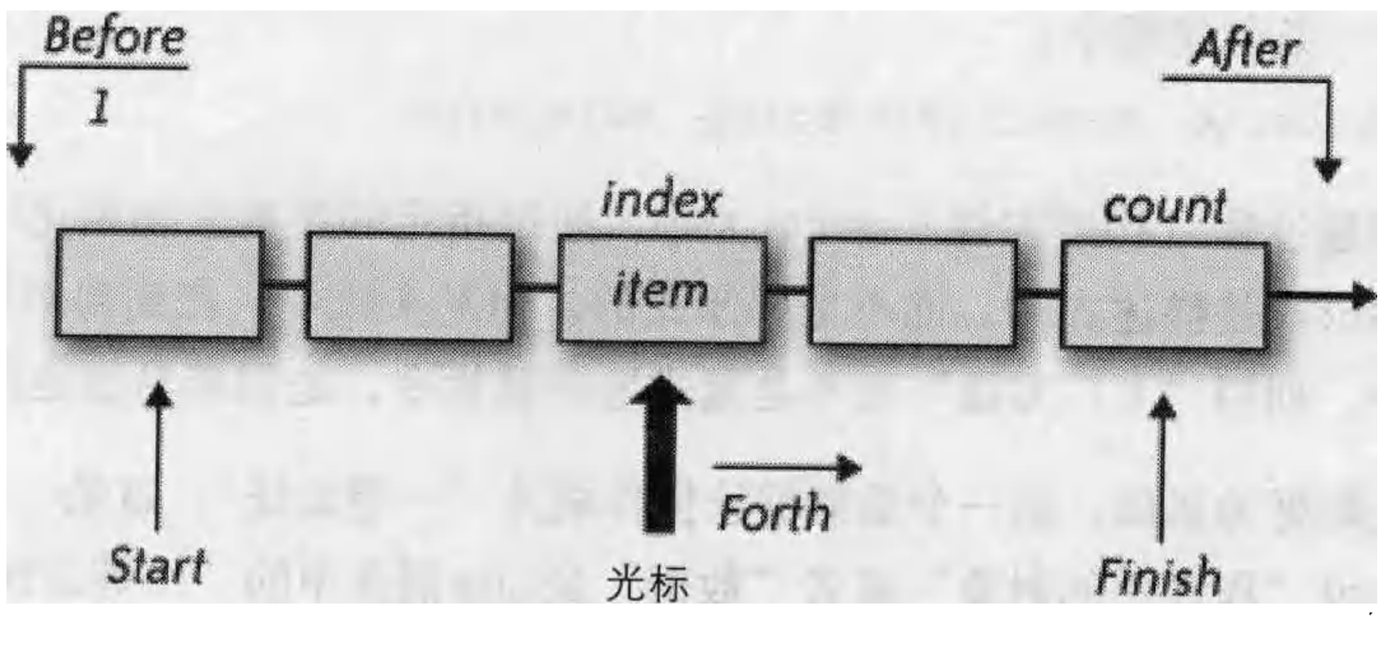


图13-2：Eiffel列表中的光标

移动光标的命令包括start（移动到第一项）、forth（移动到下一项）和finish。布尔型查询before和after用来确认当前光标是否在第一个元素之前，或者在最后一个元素之后。如果不是以上两种情况，那么item将返回当前光标位置的元素，而index将返回其索引值。

我们很容易将这种方式移植到无限序列中：只要去除finish和after命令即可（当然还有计算项目总数的count）。这也就是Eiffel程序库中的派生（抽象）类COUNTABLE的规约。它的派生类包括PRIMES、FIBONACCI和RANDOM；这些派生类都提供了自己的start、forth和item命令实现（在最后一个派生类中，还提供了plus方法，用来为随机数生成器设置一个种子）。要获得这些无限序列中的后续元素，可以先使用start命令，然后向前查询后续有限个元素即可。

任何无限序列结构都需要提供能够以该风格建模的有限计算。虽然这并未覆盖各种延迟计算的应用场景，但它的优点在于使无限结构更加清晰，也更易于构建正确的延迟计算。

13.3.5 状态调停

函数式方法旨在直接依赖于数学函数的属性，而且拒绝采用假定，并且隐含着命令式方法，其计算操作除了生成结果（数学函数常见的功能）之外，还能够修改计算的状态：可能是修改全局状态，也可能只是修改状态的一部分（在更模块化的方法中），例如某个特定对象的内容。

尽管在各种函数式编程的文章中都强调了这一点，但该属性在本例中却没有涉及，或许因为根据最初问题分析，已经将状态从函数式的结构中去除了。这也是可能的，例如，在一个用来计算金融契约的非函数模型表示法中，操作可能只是将状态转换为对值的更新，而不是基于一个序列（值处理序列）实现的函数。

它仍然可能是在函数式方法的这一基础性决策上做出的通用评价。在任何系统模型（无论是否为计算机系统）中，状态的概念总是难以避免的。甚至有观点认为它是计算的核心概念。（有人辩称[Peyton Jones 2007]无状态编程有助于解决并发编程的一些问题，但是还没有足够的证据能得出一般性结论。）世界不会因每次重要事件而克隆自己我们的计算机内存也是这样，而只是覆盖其单元格而已。我们总是可以对这样的状态改变进行建模，例如通过设置一个值序列的方法，不过这些都是人造的（作为前一个疑问的备选回答，采用函数式编程的程序员并没有换“灯泡”，而是买了一个带有新插槽、新电缆、新灯泡的“灯”）。

由于我们认识到作为这些操作的输入和输出的状态是不可能忽略的，加上之前的一些笨拙的尝试（Peyton Jones和Wadler 1993），因此在现代的函数式语言（特别是Haskell）中引入了monad的概念（Wadler 1995），它可以将最初的函数插入到高阶函数中，并且使用了更为复杂的签名；新添加的签名部分可以用来记录状态信息，以及其他一些额外的元素，诸如错误状态（用来对错误进行建模）或输入/输出结果。

使用monad整合状态处理的想法，与上一小节中通过将无限序列建模为一个抽象数据类型来实现延迟计算行为所采用技术的想法是相似的，只不过它是一种反向应用而已：在框架A中模拟一个框架B中隐含的技术T，然后在A中编程实现一个T的显式版本，或者

实现一个实现T的关键机制。对于第一个示例而言，T就是无限列表（关键机制就是对无限列表的有限计算）；对于第二个示例而言，T就是状态。

对于编程语言的语义描述（例如，用来简化标记理论）而言，monad的概念是十分漂亮和十分有用的。不过，有人可能不会关心它是否是程序员直接使用的合适的解决方案。在此，我们必须关注于正确的观点。monad也有明显的非直接相关的缺陷，那就是普通的程序员很难学会它；创新的想法，很难在最初引入时就融入到主流的教育体系中。（递归式编程和面向对象编程都曾经被视为超出了“Joe程序员”的范畴。）其中最重要的问题是这个麻烦的方法是否有价值。通过monad为函数式编程引入状态机制，就像在你确认大家信奉贞洁之后，告诉你的追求者，你有孩子。

真是需要首先把状态排除在外？以下两个观测结论就足以引发这一怀疑：

- 基本的状态改变操作，如简单的赋值操作，都有清晰的数据模型（在此是霍尔逻辑规则，它是基于“置换”的）。这降低了无状态编程所预期的主要好处：简化与程序相关的数学推理。
- 针对与确保设计或实现的正确性相关的更复杂方面而言，函数式方法的优势并不清晰。例如，校验有特定的属性和终止条件的一个递归定义（需要循环不变量和变量）。有效的函数式程序也未必能够不使用链接的数据结构，尽管其导致的问题（如别名混淆）都是与底层的编程模型无关的挑战。

如果对于确保程序正确这一任务而言，函数式编程未能显著简化，那么就将遗留一个主要的实际问题：引用透明。这就是等价替代性的概念：在数学领域中，f(a)始终意味着与指定的f和a相关的某种事情。这在纯函数式方法中也是正确的。但在函数可能带有副作用的编程语言中，f(a)在连续调用时可能会返回不同的结果。通过保留语义推理的常用模式来消除这种可能性，会使程序文本更易于理解；例如，我们都认为g+g和2×g具有相同的意义，但如果g是产生副作用的函数，那么这一结论就无法确保了。这对于自动化验证工具（能够检查函数是否会产生副作用）而言并不困难，实际上是对于人类读者而言有困难。

确保表达式的引用透明是个很合乎大家希望的目标。不过，这正说明将与状态相关的表示法从计算模型中分离出来是合理的。在此回顾一下Eiffel方法中定义的“命令-查询分离原则（Meyer 1997）”规则是很重要的。在这个方法中，类的特性（操作）可以清晰地分成两类：一类是命令，它能够修改目标对象以及它的状态；另一类是查询，它能够提供与对象相关的信息。命令不会返回结果；查询不会改变状态，换句话说，它们满足引用透明原则。在先前的列表示例中，start、forth和finish（针对有限列表）都是命令；item、index、count、before和after（针对有限列表）都是查询。该规则排除了很多通过调用函数获取一个结果并修改其状态的太过于常见的模式，我估计这就是

对命令式编程不满意的实际根源，它比先通过一个命令来修改状态，再通过一个（没有副作用的）查询来获取信息更令人烦恼。该原则也先可这样描述："提问时不修改答案。"例如，它意味着一个典型的输入操作应该写成：

```
io.read_character
Result:= io.last_character
```

在此read_character是一个命令，它将从输入中获得一个字符；而last_character则是一个查询，将返回最后读取的字符（它们都是基本I/O程序库提供的特性）。紧接着调用last_character将确保返回相同的结果。不管是哪里提到的，不管是出于理论还是实践的角度（Meyer 1997），命令-查询分离原则都是一种方法性规则，而非语言本身的特性，但所有慎重的Eiffel软件开发过程中都会遵从这一原则，以获得引用透明的优点。尽管其他面向对象编程流派未能应用它（继续采用C风格的函数调用，而没有专门完成修改的程序），但在我们看来这是面向对象方法中很关键的原则。它看起来也是在函数式编程中实现引用透明的可行方法，由于只涉及查询的表达式不会修改状态，因此可以被理解为传统的数学公式可函数式的语言。而在建模系统和计算中状态的概念，则通过命名表示法来完成。

13.4 面向对象视图

现在我们将考虑如何为该幻灯片和该文章中讨论的示例设计一个面对象的架构。

13.4.1 组合子很好，但类型更好

迄今为止，我们处理的都是操作和组合子。操作我们将保留，而关键的步骤是抛弃组合子，并将它们替换成类型（或者称为类，它们的区别是由泛型带来的）。它将明显地提升抽象等级：

- 一种组合子描述了基于原有的机制构建新机制的特定方法。组合是以很严格的方式定义的：与一个名为take的组合子（如take 3 apple）关联的是一个数量元素和一个食物元素。正如前面所说的，它和数学公式很像，是按照精确的实现过程定义的结构。
- 一个类定义了一个对象类型，它将列出所有可应用的特性（操作）。它提供了抽象数据类型意义上的抽象：它不仅定义了由什么内容构建，还定义了相应对象可应用的操作。我们可以从中领会数据抽象和面向对象设计的原则，采用这种方法意味着不仅知道对象是什么，还知道它有什么（它们公共的特性和相关的契约）。这同样提供了对各种类型继续进行分类的方法，或称之为继承，它一方面能控制模型的复杂度，同时还能充分发挥不同类型存在共性的优势。

从第一种方法改成第二种方法，我们并不会丢失任何东西，因为类可以很容易将组合子当作一种特殊的情况来考虑。它完全能够为指定要素提供所需的特性，并提供能够构建正确对象的相关创建程序（构造函数）。我们来看一个例子：

```
class REPETITION create
      make
feature
      base: FOOD
      quantity: REAL
      make (b: FOOD; q: REAL)
            --用quantity个base生成该food元素。
            ensure
                  base = b
                  quantity = q
            end
  ... Other features ...
end
```

这样我们可以通过创建apple_salad.make(6.0,apple)来获取一个该类型的对象，它和使用组合子的表示式是等价的。

13.4.2 使用软件契约和泛型

由于我们只关注设计，因此其效果实际上已经通过后置条件的形式表示了，但它对于引用实现语句（do base:=b; quantity :=q）而言，实际上并不是什么问题。它对于易于理解的面向对象设计而言是很重要的，它能够缩短实现与设计（以及规约）之间的距离。在所有这些方面，我们可以自由地使用会修改状态的赋值指令，并仍然（在此感谢读者的提问）拥有完整的功能。

不过，类和组合子不同，它的功能并不限于这些。例如，它可以拥有其他创建程序。一种很常见的现象就是将两个相同的东西混合在一起：

```
make (r1, r2: REPETITION)
      --通过合成r1和r2生成该food元素。
      require
            r1.base = r2.base
      ensure
            base = r1.base
            quantity = q
```

后置条件表达式说明了和相同基本食物类型混合的数量。这样的需求也可以通过该类型系统以静态的形式实现，使用泛型（在类型化的函数式语言中也实现了，不过其名称会令人感到好奇，即“参数化多态”）将使类的定义变成：

```
class REPETITION[FOOD] create
      ... 和之前一样...
feature
      make (r1, r2: REPETITION[FOOD])
```

```
            ... 在此不一定需要预言 ...
            ... 其他部分和之前一样 ...
end
```

类不仅可以有不同类型的创建程序，还可以拥有更多特性。具体来说，我们之前程序中的操作都可以变成相应类的特性。（读者可能会猜到，变量名t将变成类型，而f将变成特性。）对于pudding类（包括用来描述诸如REPETITION等食物类型的类）而言，将拥有诸如sugar、calorie_content等特性；而对于contract类而言，则将拥有如horizon、value等特性。在此有两点需要说明：

- 由于我们是从纯函数式模型开始的，因此迄今为止提到的所有特性要么是创建程序，要么是查询。虽然也可以在面向对象框架中保留这种函数式的风格，但在开发中也可能需要引入命令。例如，因响应特定事件（诸如重新协商）而对金融契约所做的修改。而是否需要维护状态的问题，实际上与模块化的讨论是不相关的。
- 最初，value函数是遵从于一个无限序列的。我们可以通过使用COUNTABLE类型来保留这一签名，并允许执行延迟计算；或者我们可以提供一个整型的参数，因此value (i)返回的值将是第i个值。

13.4.3 模块化策略

迄今为止所实现的模块化，阐述了面向对象技术的本质想法（至少是我们找到的本质想法之一）：将类型和模块的概念合并起来（Meyer，1997）。用最简单的话来说，那就是面向对象分析、设计和实现意味着我们基于的每个模块都是系统维护的一种类型的对象。它与其他方法提供的模块化设施相比，这是一个限制更多的原则：一个模块不再是一个软件元素（如操作、类型、变量）的简单结合体，设计师在组合这些时要基于一些合理的标准；它们是一种类型实例所能接受的属性和操作的集合。

类就是这种类型-模块合并的产物。在诸如Smalltalk、Eiffel和C#（不仅限于这些，例如还有C++或Java）等面向对象语言中，这种合并是双向的：不仅仅是类定义了一种类型（例如一种类型模板，或者使用了泛型），反过来任何类型（包括诸如整型之类的类型）也都被定义成了类。

如果想使类只保留类型的角色，并使其完全与模块化结构分离开，这也是可能的。诸如OCaml之类的函数式语言是一个特例，它不仅提供了传统的模块结构，也从面向对象编程中吸纳了类型机制。（Haskell也类似，只不过对类这一概念提供了更多的限制。）反过来，也可以去除所有类型都必须定义为类的限制，例如在C++和Java中，诸如整型（Ininter）之类的基本类型就不是类。从对象技术的视角上看，是假定它们完全合并了，另外理解更高层面的类分组机制（诸如Java和.NET中的包，Eiffel中cluster）也是必要的，但这些机制更多着眼于组织的便利，而非基础性的程序构造。

该方法意味着在定义软件架构时，类型优先于函数。其模块化标准是类型，每个操作（函数）将放在一个类中，反之则不亦然。不过，函数也通过抽象数据类型原则的应用反戈一击：当定义了一个类之后，要理解其他信息，可以通过一个抽象接口（API）列出可应用的操作，以及其定义的语义属性（契约：前置条件、后置条件，以及针对整个类的不变量）。

这种模块化策略的基本原理是遵从更好的模块性原则，包括可扩展性、可复用性以及可靠性（通过使用契约实现）。不过，我们必须检查这些承诺对于我们的示例而言是正确的。

13.4.4 继承

面向对象方法对于模块性目标的最本质贡献是继承。我们预期本书的读者对该技术很熟悉，在此将只回顾一些基本的观点，并勾画其在我们示例中可能的应用方法。

继承采用了“分类法”来组织类，大概可以表示为“is-a”关系，类之间的基本关系还有一种名为“客户”（client）的关系，它表示某类将通过其API（包括操作、签名、契约）来使用该类。继承通常是不遵从信息隐藏原则的，它是一种恰好相反的“is-a”视图。尽管有些作者只针对纯子类型使用继承，但实际上如果用它来实现标准的模块包含机制也是没有问题的。Eiffel实际上就提供了“非一致性继承”机制（Ecma International，2006），它失去了多态的支持，但保留了继承的其他特性。继承的这种双角色与类的双角色（类型和模块）特性是相吻合的。

不管是哪一种，继承都是用来捕获类之间的共性的。对于与布丁相关的元素进行分类，可以用如图13-3所示的继承图来表示。

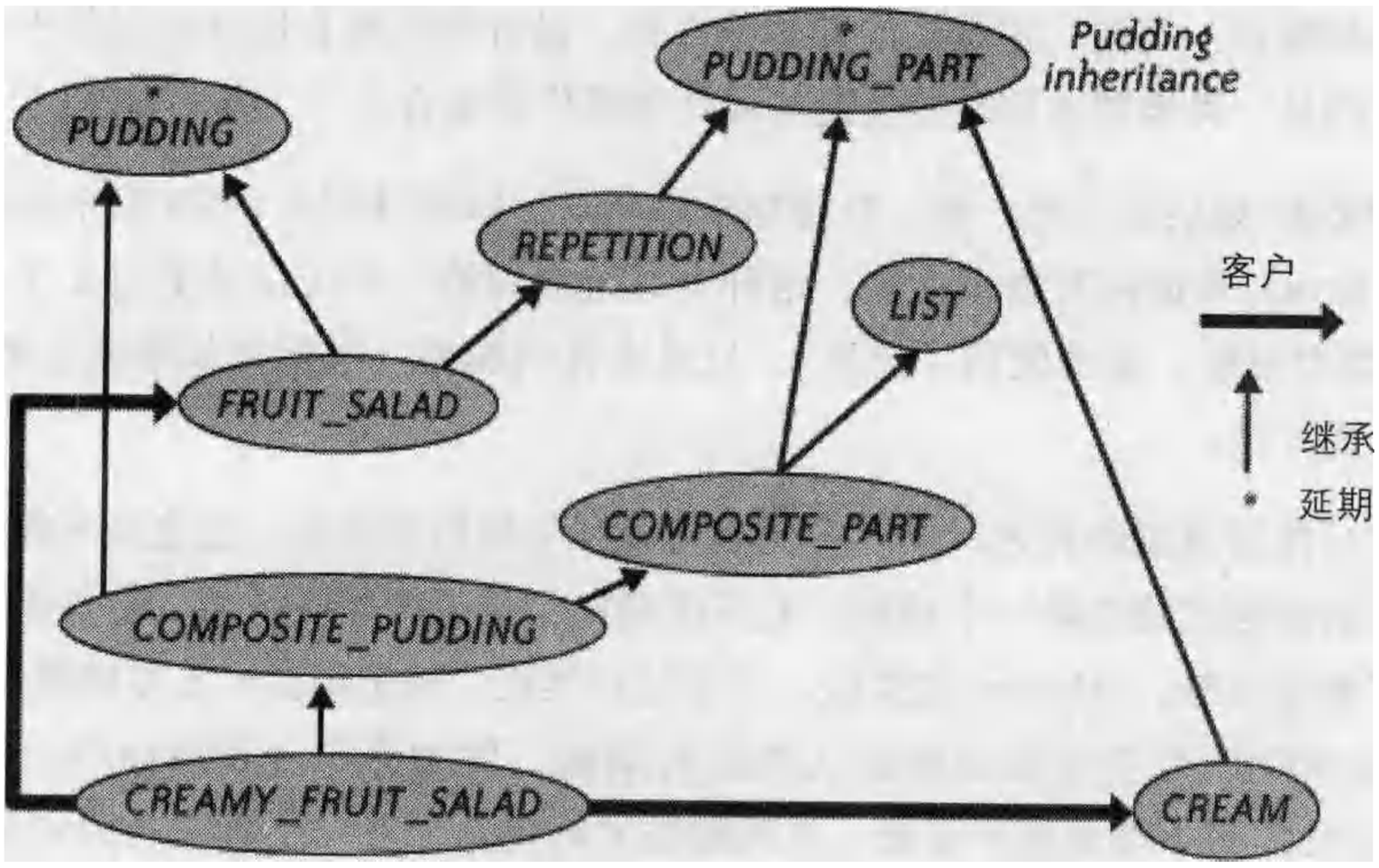


图13-3：展示布丁继承结构的类图

继承和客户（client）关系之间的区别是值得注意的。水果沙拉（fruit salad）是一种布丁，它同时也是一种复用品（我们忽略了通用参数）。一种复用品（repetition）并不是一种布丁，而是一种描述食物构成的“布丁成分”。有些“布丁成分”（诸如“合成布丁”）也是布丁，当然也不全是。水果沙拉（fruit salad）是一种布丁，同时也是一种复用品（repetition），它是一种布丁成分（pudding _part）。另一方面，一种奶油水果沙拉（creamy fruit salad）不是一种水果沙拉（fruit salad），即使我们使用这一表示法表示用水果制成的布丁。它拥有水果沙拉和奶油，因此它使用了客户关系。它是一种合成布丁（composite pudding），因为该表示法用来表示它是由几个部分组成的，它和更通用的表示法COMPOSITE_PART一样，也是一种布丁。对于这个成分而言，它和水果沙拉（fruit salad）和奶油（cream）之间是客户关系。

对于金融契约示例而言，我们可以使用相似的方法，可以基于契约的类型将其分成“零息票债券”、“期权”以及其他由问题域专家经过仔细分析所得出的类别。

多重继承对于这种形式的面向对象建模是至关重要的。特别注意组合部分（composite part）的定义，在此应用了一个通用的模式来描述这种组合结构（参见Meyer 1997，5.1节，“composite figures”）：

```
Class COMPOSITE_PART inherit PUDDING_PARTLIST[PUDDING_PART]
feature
 ...
end
```

其中方括号引入了泛型参数一个组成部分（composite part）既是布丁的成分，拥有所有可应用的属性和操作（sugar content等），也是一个布丁成分列表，同样带有一些可应用的列表操作：诸如start、forth之类的指针移动命令，诸如item和index之类的查询，以及插入、删除元素的命令。列表中的元素可以是任何一种布丁成分，包括（递归的）组合部分（composite part）。这样，多态和动态绑定技术也就有了用武之地，我们稍后将对其进行说明。泛型和继承都是很有用的机制。另外，多重继承使针对类的机制更加完整，它不仅仅局限于用接口（Java风格）实现，这此并没有应用到接口。

13.4.5 多态、多态容器和动态绑定

继承和泛化对可扩展性的贡献，以及多态和动态绑定技术所带来的可扩展性，我们可以从COMPOSITE_PART类中的sugar_content中体会到（参见图13-4）：

```
sugar_content: REAL
      do
            from start until after loop
                  Result := Result + item.sugar_content
                  forth
      end
end
```

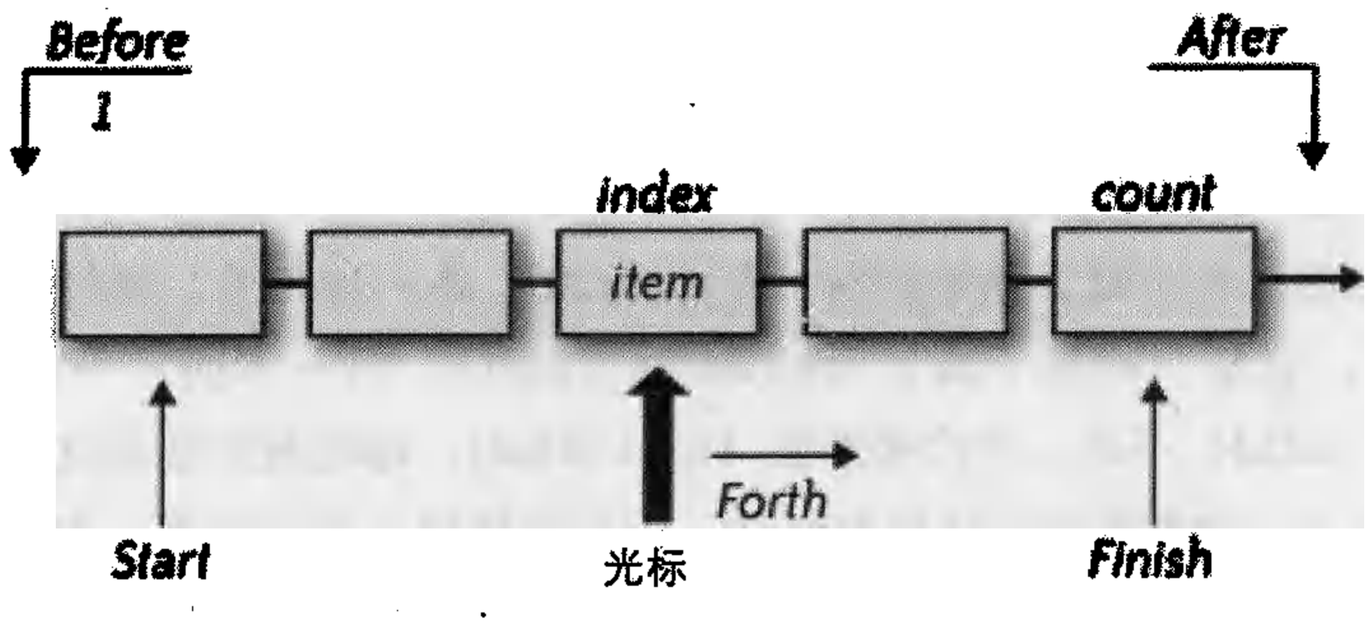


图13-4：带光标的多态列表

在此直接将LIST类中的操作应用到COMPOSITE_PART中了，这是因为COMPOSITE_PART就是从LIST类中继承的。item命令返回的结果可以是任何类型（包括子类型）的PUDDING（布丁）；由于它的结果是多种类型的对象，也就是众所周知的多态变量（在这个例子中正好是多态查询）。一个完整的COMPOSITE_PART结构，包括不同类型的项目，这也就是众所周知的多态容器。多态容器使多态的合成成为可能，它本身是继承和泛型的结果。（在此有两种完全不同的机制，在函数式编程中，针对泛型的“参数化多态”很容易与这里的多态混淆起来。）

item的多态，意味着后续调用item.sugar_content时可能会应用在不同类型的对象上；相应的类可能会有不同版本的sugar_content查询。动态绑定就是负责确保这样的调用能够应用合适的版本，它将基于item所属对象的实际类型来决定。如果某种成分本身就是组合部分，也就是前面所示的版本，它将递归地应用，而且可以是其他任何版本，例如针对CREAM的版本。

这是针对面向对象设计的最新方法，多态是由类型系统控制的。item值的类型是可变的，不过它肯定是PUDDING及其子孙类，它们是由COMPOSITE_PART的泛型参数指定的。这部分的内容不管对函数式编程的环境，还是面向对象编程的环境都有影响：使用日益复杂的类型系统（仍然是基于诸如继承、泛型和多态等少量概念），将与系统架构相关的、日益发展的部分整合到类型结构中。

13.4.6 延期类和特性

在前面的类结构图中，PUDDING和PUDDING_PART类都被标识为“延期”（deffered）（使用的是BON面向对象建模表示法[Walden和Nerson，1994]中所指定的星号）。这意味着，它们并没有完整实现；换一种说法，可以称之为“抽象类”。一个延期类通常拥有延期特性，拥有一个签名和（更重要的）契约，但没有实现。具体的实现是由非延期的（“已实现的类”）子孙类呈现的，适合的选择是由每个已实现类来实现延期类中定义的

通用概念。在该示例中，PUDDING和PUDDING_PART类都拥有延期特性sugar_content和calories；其子孙类将实现它。例如，在COMPOSITE_PART就将sugar_content定义为计算其各成分的糖量总和。在COMPOSITE_PUDDING中，它就从COMPOSITE_PART中继承了该版本，并从PUDDING类中继承了一些延期版本的特性，并给予实现。

注意：当继承了两个同名特性时就将导致名称冲突，这就需要通过重命名来解决，除非一个特性是延期的，另一个是已实现的。在这种情况下，只有一个特性可用，那就是那个已实现的。对于继承机制带来的名字重载会引发难以处理的复杂度，因此通常建议在面向对象语言中不提供这种机制。

延期类要比Java和.NET中的“接口”更为复杂，它们都可以指定一个未来必须实现的契约，而且它们都可以包含已实现的特性，这样它能够提供的就是一个完整的特性集，从全部特性都是延期的类（只描述了要实现的特性）到全部都已实现的类（提供了完整实现的类）。它也可以只实现了一部分特性，这也是在架构和设计中使用面向对象技术的本质。

在金融契约示例中，CONTRACT和OPTION就是纯延期类的候选者，尽管它们可能无需对所有特性的实现进行延期。

13.5 面向对象模块性的评价和改进

在前一个小节中，我们总结了面向对象架构设计在这两个例子中的应用。接下来我们将按照本章之前描述的模块化标准来检查这个初步应用的结果。虽然类型系统和契约对可靠性也有贡献，不过我们在此只专注于可复用性和可扩展性。

13.5.1 复用操作

使用继承的原则之一是将公共特性放在可应用的最高等级类上，这样在子孙类中就无需重复它们；也就是说继承关系就是“as is”。如果既需要保留该功能，又需要修改其实现，那么只需要重新定义（或“重载”）继承而来的版本即可。“保留该功能”意味着仍然应用原来的契约，但其内容需要重写，不管它是已实现还是仍然为延期的。这和动态绑定能够很好地结合：一个客户端可以使用高层级类中定义的操作，例如my_pudding.sugar_content或my_contract.value，而无需知道实际使用的是哪个类中的哪个版本，也不管它是该类定义的还是继承的。

由于继承捕获了类之间的公共信息，因此要定义的特性会明显小于最大的$t \times f$。任何减少代码的工作在此都是有价值的。代码重复始终是有害的，这是软件设计的通用规则，这是因为代码重复意味着在后续的配置管理、维护和调试时（当发现原代码存在问题时，必须修改所有的副本）会带来问题。正如David Parnas所说的那样，复制-粘贴是软件工程师的敌人。

实际上能够减少多少代码量，取决于继承结构的质量。我们发现抽象数据类型的原则在此仍然适用：由于在面向对象设计中，定义类型的关键在于分析其可应用的操作，在一个设计良好的继承结构中，最顶层的类中将收集在大量变体中可应用的特性。

该技术在函数式的模型中没有明确的等价物。在使用组合子时，对于每个组合子都需要定义不同的操作，肯定会存在一些代码重复。

13.5.2 可扩展性：添加类型

面向对象风格的架构对可扩展性的支持如何呢？系统最常见的扩展形式就是添加新的类型：一种新的布丁、布丁成分或金融契约。这种情况，是面向对象技术充分发挥自己作用的地方。你只需要在继承结构中找到最适合新类型的位置（也就是具有最多相同操作的类），然后编写一个新的类，让它继承一些原有的特性，根据自己的情况重新定义或修改原有的操作，并添加一些新的类可以使用的新特性和不变量。

动态绑定在此又起关键作用；面向对象方法的好处在于当客户类执行某个操作时，无需进行多路分支判断，诸如："如果是水果沙拉（fruit salad），那么就以这种方式计算，而如果它是flan，那么就用那种方式计算，否则……"，这样的判断必须在每种操作中不断重复，更坏的是当添加或修改某种类型时，我们需要分别针对每个客户、每个操作进行相应的更新。这样的结构，需要客户类维护各种复杂的知识，了解不同结构提供的不同概念以及它们依赖的基础，这是架构退化的根源，也是面向对象之前的技术逐渐过时的原因。动态绑定解决了这个问题；客户应用程序只需调用`my_contract.value`，内建的机制就将自动选择出相应的版本，客户无需知道它们之间的不同。

没有哪种软件架构技术能够提供如此优美的解决方案，这也是面向对象方法所能够提供的最佳解决方案。

13.5.3 可扩展性：添加操作

面向对象技术对于可扩展性的支持，有一部分（此外还有诸如信息隐藏、泛型以及契约等重要特性）源于这样的假设：系统中最最重要的变化是由于添加一个新的类型，它和原有类型共享了部分操作，并添加了一些新的操作。经验确实表明，这些是实际系统中最常见的变化，也是面向对象技术展示其优势的地方。但如果我们想在原有类型中添加一些操作呢？例如，有些要使用pudding类的客户应用程序，可能想确定生产不同种类布丁所需的成本，虽然pudding类中并未包含与成本相关的信息。

与添加类别相比，对于添加新操作而言，函数式编程表现得不好也不坏，不管添加1个到*f*个，还是添加*t*个。不过，在面向对象解决方案中是无法享受这种中性的优点。其基本解决方案是在继承层次结构中合适的等级上添加一个特性。但这样做有两个潜在的缺点：

- 由于继承是类间的强绑定（“is-a”），因此所有原有的子孙类都会受到影响。一般来说，在继承结构的较高层次的类中添加一个特性，经常会带来一些微妙的麻烦。
- 如果客户系统不允许修改原有类或不允许访问其上下文，那么该解决方案甚至可能是不可用的。这在实践中是很常见的场景，特别是当这些类被归入到一个程序库中，例如将其放在一个金融契约程序库中。这样就不可能允许使用该程序库的应用程序作者修改它。

对此，基本的面向对象技术（Meyer 1997）并不够用。针对该问题，广泛应用的标准的面向对象解决方案是访问者模式（Gamma 等 1994）。在下面只是对该模式的概略描述，并不是标准的介绍，但也足够对该观点做一总结。（它是Meyer的《Touch of class：An Introduction to Programming Well[2008]》一书中所作的总结，它说明了这些基础概念是如何出现的，这本书可以选作第一学期的编程教科书。）图13-5列出了该模式所涉及的角色。

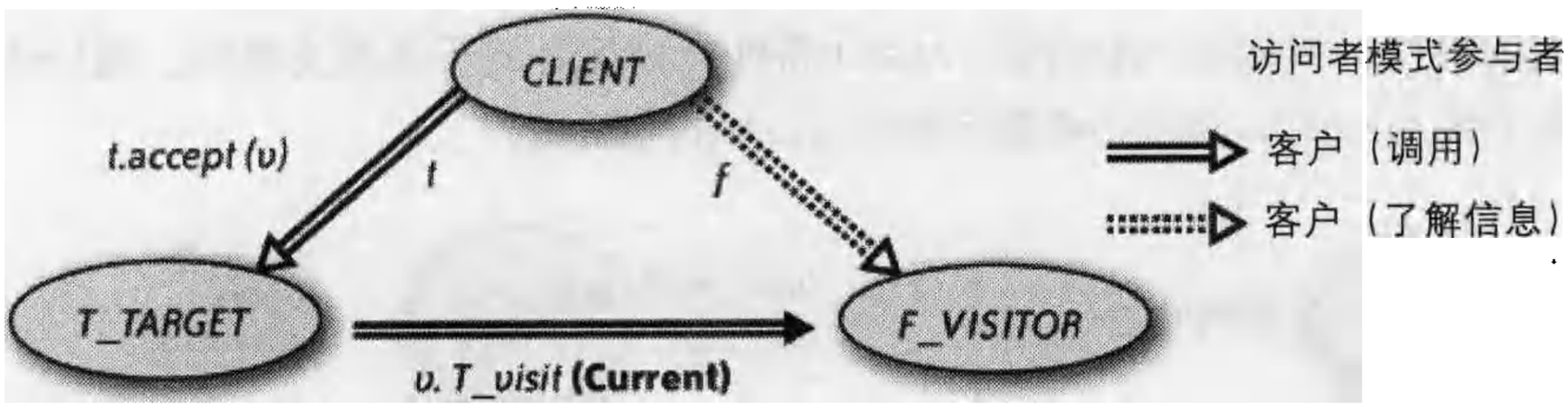


图13-5：访问者模式的参与者

该模式在应用程序（诸如CLIENT之类的类）和原有类型（诸如针对特定类型T的T_TARGET，在我们的示例中就是PUDDING或CONTRACT）跳“双人芭蕾舞”，并且通过为每个可应用的操作F引入一个访问者类F_VISITOR，使其演变成“三角家庭”。诸如CLIENT之类的应用程序类在调用目标类的操作时，将会把相应的访问者作为参数传入，例如：

```
my_fruit_salad.accept (cost_visitor)
```

accept (v:VISITOR)命令将通过调用其参数v（在本例中就是cost_visitor）来执行该操作，它是诸如FRUIT_SALAD_visit类的一个特性，该名称中体现了目标类型。该特性是类描述的一部分，在此是FRUIT_SALAD；它将应用于相应类型的一个对象上（在此是fruit salad对象），它将作为参数传给特性T_visit。Current是Eiffel中针对当前对象的表示法（类似于大家所知道的“this”或“self”）。调用的目标，也就是图中的v，表示该操作将使用相应访问者类型的一个对象，诸如COST_VISITOR。

当我们评价可扩展性时，对于软件架构最为关键的问题始终是“知识分发”(distribution of knowledge)。有一种方法能够实现可扩展性，那就是限制模块必须拥有

的与其他模块相关的知识总量（这样当添加或修改某个模块时，对原有结构的影响就将最少）。为了让大家理解访问者模式的精妙之处，看看每个参与者必须知道什么、不需要知道什么是十分有必要的：

- 目标类知道某种特定的类型，同时也知道类型层次结构中的上下文（例如，FRUIT_SALAD从COMPOSITE_PUDDING中继承，而COMPOSITE_PUDDING又从PUDDING中继承）。它无需知道外部所需要的新操作，诸如获取生产某种布丁所需的成本。
- 访问者类必须知道与特定操作相关的所有信息，诸如成本，并为一组相关的类型提供相应的变量，以及通过参数指示相应的对象：在本例中，我们将发现诸如fruit_salad_cost、flan_cost、tart_cost等例程。
- 客户类需要对特定类型的对象应用一个指定的操作，因此必须知道这些类型（只需知道它们存在，而不关心它们的其他属性）和操作（只要它们存在并且适用于指定类型，无需知道每个的特定算法）。

某些所需的操作，诸如accept和T_visit特性，必须是源于其祖先类的。图13-6展示了带继承关系（FRUIT_SALAD将简写为SALAD）的完整图。

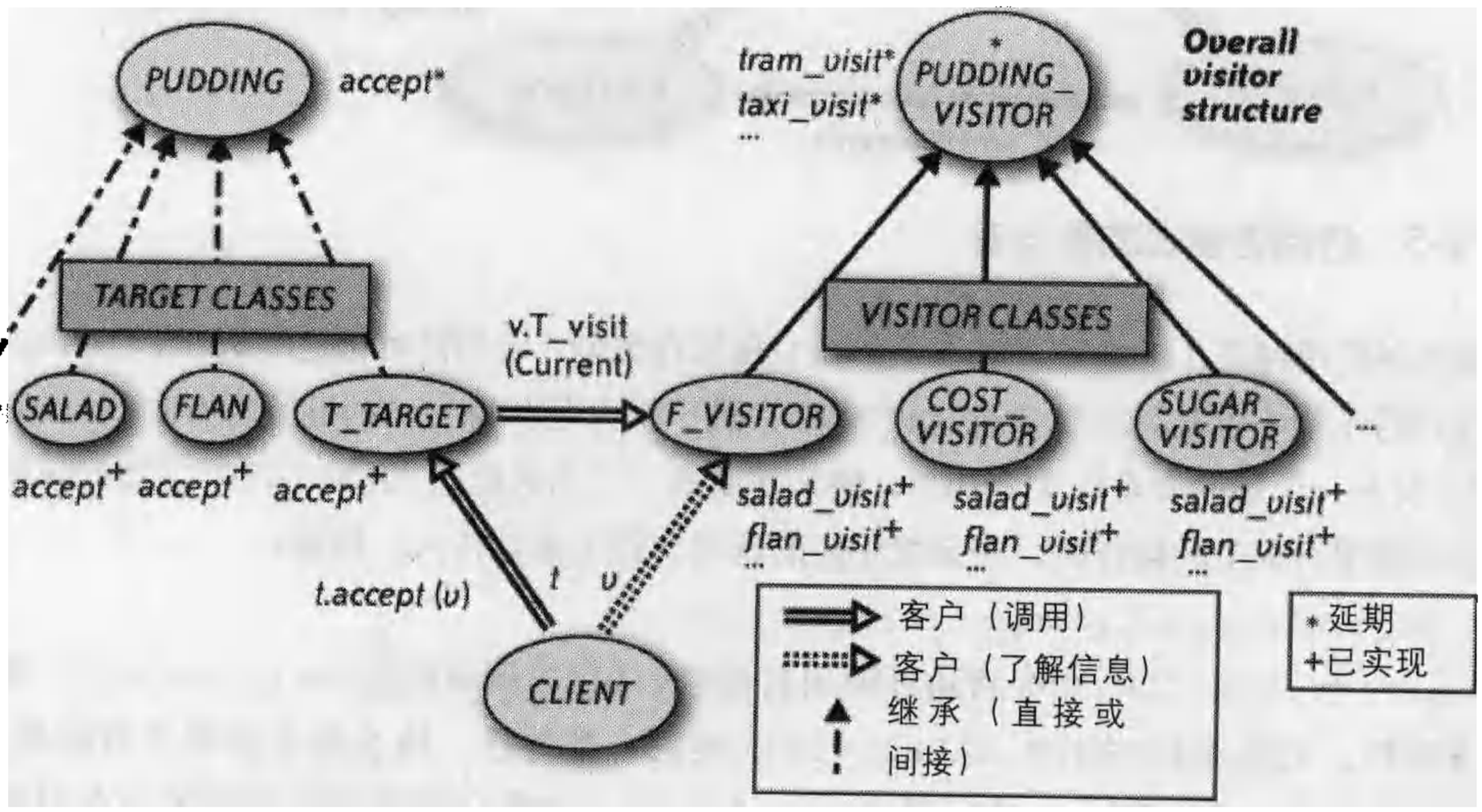


图13-6：整合在一起：构建布丁程序的架构

像这样的架构，经常用来解决在带有许多继承关系的原有结构中添加新操作的问题，它无需针对每个操作修改整个结构。语言处理程序（在集成开发环境中的编译及其他工具）就是存在这种情况的常见应用程序，它的底层结构是一个抽象语法树（AST）：每当新工具有需要时，更新AST类都是很麻烦的，根据其意图，需要在该树上执行一个遍历操

作，对每个节点应用该工具选择的操作。（这也就是“访问”节点，同时解释了“访问者”这一术语和T_visit这一特性名称的来源。）

对于所有这些操作，客户必须能够针对任何目标类型的任何t执行t.accept(v)操作。这也就假定所有目标类型都是从公共类（在此就是PUDDING）中派生出来的，能够接受该特性的地方将以延期实现的形式声明它。这一需求很精妙，因为整个示例的目标正好是避免对现有的目标类进行修改。设计师在使用访问者模式时，通常会认为该需求是可接受的，也就是意味着我们关心的类必须有相同的祖先类，这是很常见的场景，它们是用来表示相同概念的不同类型，诸如PUDDING或CONTRACT，我们只需要在其祖先类中添加一个延期特性accept即可。

访问者模式应用十分广泛。读者可能已经知道它“美”在什么地方。但在我们的视角中，这并不是最终定论。评价标准包括：

带有特定accept功能的领域特定类需要拥有共同的祖先类，不应该受到与应用领域无关的概念的影响，无论是布丁、金融契约还是其他东西。

更令人不安的是，随着表示每种特定知识（基于特定类型的特定操作）的、小型的F_VISITOR类的大量增加，最终将出现“类爆炸”现象。不过，对于整个软件架构而言，这只是一个小污点。

要清除这个污点，需要在基本的面向对象框架中添加一个新的概念：代理。

13.6 代理：将操作封装到对象中

代理（Eiffel语言在1997年就在其基本的面向对象框架中添加了代理；C#中的“委托”和它是等价的）的基本想法可以用函数式编程相关论文中类似的话表述：我们将操作（函数式编程中的函数，面向对象编程中的特性）视为“一等公民”。在面向对象上下文中，在运行时的一等公民应该是对象，它和静态结构中的类相对应。

13.6.1 代理机制

代理是一个对象，用来表示特定类的一个特性，以便被调用。一个名为x.f(u,⋯)的特性的完整定义包括特性名f、目标对象x以及参数u,……而代理表达式则将指定f，并指定一部分或全部目标对象和参数，也可以都不指定，全部指定的代理称为已封闭的。对于那些未提供完整信息的代理定义，则称为开放的。该表达式将指定一个对象；该对象指定了该特性，并且包括设置指定值的参数。对于代理对象而言，其中一个可执行的操作是call，用来表示对f的一次调用；如果该代理还有未指定的参数，那么还必须将相应值作为参数传给call（对于已封闭的参数，则将使用代理定义中指定的值）。

最简单的代理表达式是agent f。在该例子中，所有参数都是开放的，但目标对象是已封闭的。因此，如果该代理表示式是a（它是执行赋值语句a:=agent f的结果，或者在调用p(agent f)中，形式变量p的值是a），那么a.call([u,v])的效果与x.f(u,v)是等价的。当然，它们之间也是有区别的，f(u,v)直接命名了该特性（尽管动态绑定意味着可能会使用不同的已知特性），而采用代理的形式，a只是一个名称，它是从其他程序单元中获取的。因此，在程序的这个位置上，对于特性而言除了其签名（或者是契约）之外是一无所知的。由于call是通用的程序库例程，它需要的参数类型只有一种。其解决方案是使用tuple，在此是两元素的tuple[u,v]。当采用这种格式时，agent f的目标对象是封闭的（它就是当前对象），而所有的参数都是开放的。

在此还有一个变体，agent x.f，其参数是开放的，目标对象是封闭的：它的目标对象是x而非当前对象。如果想让目标对象是开放的，那么可以使用agent {T}.f，其中T是x的类型。然后调用时就需要使用三个参数的tuple:a.call([x,u,v])。为了使某些参数是开放的，你可以使用相同的表示法，诸如agent x.f({U},v)（其典型的调用是a.call([u])），但由于u的类型U是可以清晰地从上下文中获得的，因此无需显式指定；在此只需使用问号，诸如agent x.f(?,v)。这同时也说明最初表示所有参数均是开放的格式的表达式agent f和agent x.f，实际上就是表达式agent f(?,?)和agent x.f(?,?)的缩写。

call机制就应用了动态绑定：f版本也应用了动态绑定，正如非代理的调用一样，它取决于动态的目标类型。

如果f表示的是一个查询而非命令，那么你可以获取相应的代理，其结果是用item来代替调用，也就是a.item([x,u,v])（它将执行一个调用，并获得其执行结果）；你也可以先调用它然后再访问a.last_result，这样就遵从了命令-查询分离原则，它将返回相同的值，在连续调用时无需执行后续的调用。

对于更高级的应用，它不仅是基于一个针对原有特性f的代理，你也可以在同一行中使用代理，例如editor_window.set_mouse_enter_action (**agent do** text.hightlight end)，它展示了图形用户界面上的典型应用，这是EiffelVision程序库（Eiffel软件：EiffelVision文档）中事件驱动编程的基本风格。内联的代理提供了与函数式语言中lambda表达式相同的机制：编写操作，使它们在软件中像数值一样直接可用，就像其他“一等公民”一样。

总而言之，代理使面向对象框架能够定义更高等级的功能模块，就像在函数式语言中那样，并提供了功能相当的表达式。

13.6.2 代理的应用范围

代理现在已经成为了对基本面向对象机制的重要、自然的补充。在实践中，它们广泛应用于：

- 迭代：为一个容器结构中的所有元素应用一个可变的操作，因此使用代理是很自然的决策。
- 正如前面所提到的GUI编程领域。
- 数学计算，正如之前示例中那样对在特定时间间隔内的特定函数进行整合，表示为一个代理。
- 反射，一个不仅提供特性的属性（不仅能够通过call和item来调用），还能提供类的代理。

在我们的研究中，代理已经被证实可以用来把设计模式替换为可复用组件（Arnout 2004；Arnout和Meyer 2006；Meyer 2004；Meyer和Arnout 2006）。其原因在于，任何应用程序的设计师在使用模式之前必须详细地学习，包括架构和实现，然后从头开始构建整个应用程序，而可复用组件能够直接通过API来使用它。成功的案例包括观察者设计模式（Meyer 2004；Meyer 2008），没有人在看到基于代理的解决方案之后，还会继续尝试使用该模式，此外还有工厂模式（Arnout和Meyer 2006）和接下来还将讨论的访问者模式。

13.6.3 基于代理的程序库使访问者模式变成多余的

对于用访问者模式笨拙地解决的问题，代理机制提供了更好的解决方法：向现有的类型添加操作，并且不改变其支持的类。该解决方案在（Meyer和Arnout 2006）中详细地介绍过，你可以在ETH Chair of Software Engineering（苏黎士理工大学软件工程主席，网站为http://se.ethz.ch）下载网站的open source library（开源程序库）中找到它。

它可以使客户应用程序接口变得十分简单。对于目标类（诸如PUDDING、CONTRACT）无需做任何修改：它们没有其他要接受的特性。它可以复用需要的类，并接受其子孙类：它不会产生访问者类的大爆炸，只需要一个VISITOR程序库类，学习register和visit两个特性的基本用法即可。客户应用程序的设计师无需理解这些类的内部细节，也不用关心访问者模式的实现，只需要通过API来应用这个基本模式即可：

1. 声明一个表示访问者对象的变量，通过VISITOR的泛型参数指定最顶部的目标类型，然后创建相应的对象：

```
pudding_visitor: VISITOR [PUDDING]
create pudding_visitor
```

2. 针对在目标结构中特定类型的对象，为其中每个要执行的操作注册与访问者相应的代理：

```
pudding_visitor.register (agent fruit_salad_cost)
```

3. 要执行特定对象（通常是遍历操作的一部分）中的操作是，只需使用程序库类VISITOR的visit特性，如下所示：

```
pudding_visitor.visit (my_pudding)
```

这个接口的所有内容只包括：一个访问者对象、可应用操作的注册，以及一个独立的visit操作。我们简单地解释一下这三个属性：

- 对于将应用的操作（诸如fruit_salad_cost）都需要写出来，不管选择了什么架构。它们通常是以例程的形式存在的，这样就可以使用表示法agent fruit_salad_cost调用它；如果不是（特别是它只是一个很简单的操作时），客户应用程序可以使用内联的代理来引入一个例程。

- 当你最初看到只需一个VISITOR类、一个用来添加访问者的register例程就足够了的时候，可能会感到奇怪。在访问者模式解决方案中，当调用t.accept(v)时，t指出了目标类型（某种特定的布丁），但在此register例程却没有指定任何此类信息。那么它是如何找到需应用的正确的操作变体（是fruit salad的cost操作，还是plan的cost操作）呢？其答案是代理机制中反射属性所带来的结果：一个代理对象收集了与相关特性有关的所有信息，包括其签名。因此agent fruit_salad_cost所包含的信息就是fruit salad可应用的例程（从签名fruit_salad_cost(fs:FRUIT_SALAD)中也可以获得，对于内联代理而言，则可以从其文本中获得）。这样使得我们可以对VISITOR内部的数据结构进行组织，所以在调用visit时（诸如pudding_visitor.visit(my_pudding)），visit例程能够找到正确的例程，或者基于目标对象的动态类型的例程，在此针对特定布丁类型P的pudding_visitor:VISITOR[P]也能够匹配，它是由类型系统强制静态绑定的，对象的类型将动态和参数相关联，在此是多态的my_pudding。

- 该技术也享受到了继承和动态绑定在复用方面的好处：如果某个例程注册成通用的布丁类型（也就是COMPOSITE_PUDDING），而且没有任何注册到特定类型（例如，对于所有合成布丁而言，成本计算将采用相同的方法）的例程，那么visit将使用最佳匹配者。

这里描述的机制对传统的面向对象机制提供了一些补充。当添加的类型将提供与原有操作不同的操作时，继承和动态绑定能够提供很好的解决方案。对于和其成对的、想在不修改原有类型的基础上添加操作问题，这里介绍的解决方案也适用。

接下来，我们根据之前关于知识分发的模块性评价标准“谁必须知道什么？”来看看该方法：

- 目标类只需要知道体现相应类型的、诸如sugar_content的基本操作。

- 应用程序只需要知道它所使用的目标类的接口，以及register和visit这两个本质的特性，它们是由VISITOR程序库类实现的。如果目标类型需要添加新的操作（在目标类设计时是无法预见的），诸如我们示例中的cost（计算成本），只需要提供该操作变量，以及所需的目标类型，另外我们需要知道当缺乏对应的注册项时，将会针对更特定的类型使用更通用的操作。
- 程序库类VISITOR完全不需要知道特定的目标类型或特定的应用程序。

如果还想进一步减少系统不同部分所需信息的总量，看起来似乎是不可能的。在我们的观点中，只有一个问题还没有结论，那就是这种基础性机制如果通过程序库实现，或者不知何故屈从于特定的语言构造，那么它是否仍然是可用的。

13.6.4 评价

当引入代理这一概念时，最初大家关注的是它是否会带来引发混乱的冗余，因为它提供了一种也符合标准面向对象机制的备选方法。（这种关注在Eiffel中特别强烈，因为该语言在设计时遵循的原则是“一个好方法，做任何事情”。）这种问题并没有发生：代理在面向对象库中找到了适合自己的位置，设计师在决定何时应该使用时并不会觉得困难。

在实践中，代理的各种重要应用（特别是用来代替设计模式的应用）都同时依赖于泛型、继承、多态、动态绑定以及其他高级的面向对象机制。这也使大家强化了“该机制是成功的面向对象技术中必要的组件”的认识。

注意： 也有另一种不同的观点，你可以看看Sun公司白皮书中对于Java为什么不需要代理（或委托）之类的机制所做的解释（Sun Microsystems，1997）。它展示了如何通过Java中的“内部类”（inner classes）来模拟这一机制。虽然这很有趣、争论也很有价值、很成功，但我们的观点是，它实际证明了相反的观点。内部类能够完成这一任务，但你不难发现，当用这种方法替代访问者模式时仍然会带来类爆炸问题，与基于代理的解决方案相比，其简洁、优雅、模块性都更逊一筹。

正如之前所说的那样，代理使面向对象设计也能够实现在函数式编程中通过定义更高等级函数（操作可以作为操作的输入或输出，这样就可以递归使用相同的属性）这一机制所实现的威力。甚至在内联代理中还会发现lambda表达式的身影。这一机制对函数式编程产生了公开的影响，并且原则上吸引了其支持者的关注，虽然人们担心，从某种角度看这是罪行向美德所表示的敬意（La Rochefoucauld 1665）。（译注5）

译注5： La Rochefoucauld 的名言是Hypocrisy is the homage that vice pays to virtue（虚伪是恶德对美德所表示的崇敬），在此似乎表示的是“面向对象技术”不是美德，“函数式”是美德，因此面向对象技术通过“代理”技术来向“函数式”这一美德表示崇敬。

除了语法之外，最重要的区别在于代理不仅可以封装纯函数（查询），还能够封装命令。不过，绝对的纯粹（没有任何副作用）是不存在的，特别是对于架构的讨论而言更是如此，至少只要我们想遵从命令-查询分离原则，就能够保留该原则带来的好处（表达式的引用透明），而不需在无状态的模型中强行添加一个有状态的模型。

最终，大家认为代理对对象技术的模块性还是有贡献的，但它只是所有元素中的一个，这些在前面的讨论中有涉及，但只有一小部分。它是这些元素的组合体，超出了函数式方法所能够提供的函数，使面向对象设计成为了构建美妙架构的最有效方法。

致谢

我很感谢那些对本章的草稿提出重要意见的朋友，明显，致谢不是担保（同样明显，函数式编程的那些重要人物非常和善地分享了建设性的意见，同时恐怕又完全没有受我的观点的影响）。特别重要的是Simon Peyton Jones、Erik Meijer和Diomidis Spinellis的意见。对我关于John Hughes的经典文章所提的问题，他的回答详细而富于启发。本章最后部分提到的`Visitor`库是Karine Arnout(Karine Bezault)的作品。我感谢Gloria Muller，他在ETH硕士论文中进一步讨论了针对Eiffel实现像Haskell那样的支持库。我特别感谢这本书的编辑Diomidis Spinellis和Georgios Gousios，他们给我机会出版这些讨论，而且他们对我的延迟交稿也给予了极大的耐心。

参考资料

Arnout, Karine. 2004. "From patterns to components." Ph.D. thesis, ETH Zurich. 可以在*http://se.inf.ethz.ch/people/arnout/patterns/*找到。

Arnout, Karine, and Bertrand Meyer. 2006. "Pattern componentization: the Factory example." *Innovations in Systems and Software Technology (a NASA Journal). New York, NY: Springer-Verlag*. 可以在 *http://www.springerlink.com/content/am08351v30460827/*找到。

Eber, Jean-Marc, based on joint theoretical work with Simon Peyton-Jones and Pierre Weis. 2001. "Compositional description, valuation, and management of financial contracts: the MLFi language." 可以在*http://www.lexifi .com/ Downloads/ MLFiPresentation.ppt*找到。

Ecma International. 2006. *Eiffel: Analysis, Design and Programming Language*. ECMA-367. 可以在*http://www.ecma-international.org/publications/standards/Ecma-367.htm*找到。

Frankau, Simon, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard.2008.

"Commerical uses: Going functional on exotic trades," *Journal of Functional Programming*, 19(1):2745, October.

Gamma, Erich, et al. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

Hughes, John. 1989. "Why functional programming matters." *Computer Journal*, vol. 32, no. 2: 98-107 (revision of a 1984 paper). 可以在*http://www.cs.chalmers.se/~rjmh/Papers/whyfp.pdf*找到。

La Rochefoucauld, François de. 1665. *Réflexions ou sentences et maximes morales.*

Meyer, Bertrand, and Karine Arnout. 2006. "Componentization: the Visitor example." Computer (IEEE), vol. 39, no. 7: 23-30. 可以在 *http://se.ethz.ch/~meyer/ publications/computer/visitor.pdf*找到。

Meyer, Bertrand. 1991. *Eiffel: The Language*. (second printing.)Upper Saddle River, NJ: Prentice Hall.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*, Second Edition. Upper Saddle River, NJ: Prentice Hall. 可以在*http://archive.eiffel.com/doc/oosc/*找到。

Meyer, Bertrand. 2004. "The power of abstraction, reuse and simplicity: An object-oriented library for event-driven design." From Object-Orientation to Formal Methods: *Essays in Memory of Ole-Johan Dahl*, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, *Lecture Notes in Computer Science* 2635, pp. 236-271. New York, NY: Springer-Verlag. 可以在 *http://se.ethz.ch/~meyer/publications/lncs/events.pdf*找到。

Meyer, Bertrand. 2008. *Touch of Class: An Introduction to Programming Well*. New York, NY: Springer-Verlag. 见*http://touch.ethz.ch.*

Peyton-Jones, Simon, Jean-Marc Eber, and Julian Seward. 2000. "Composing contracts: An adventure in financial engineering." Functional pearl, in *ACM SIGPLAN International Conference on Functional Programming* (ICFP '00), Montreal, Canada, September '00. ACM Press, pp. 280-292. 可以在*http://citeseer.ist.psu.edu/jones00composing.html*找到。

Peyton-Jones, Simon, and Philip Wadler. 1993. "Imperative functional programming." *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Charleston, South Carolina, pp. 71-84. 可以在 *http://citeseer.ist.psu.edu/peytonjones93imperative.html*找到。

Steingold, Sam. Online at *http://www.podval.org/~sds/ocaml-sucks.html.*

Sun Microsystems. 1997. "About Microsoft's 'Delegates.'" White paper by the Java Language Team at JavaSoft. 可以在 *http://java.sun.com/docs/white/delegates.html*找到。

Wadler, Philip. 1995. "Monads for functional programming." *Advanced Functional Programming*, Lecture Notes in Computer Science 925. Eds. J. Jeuring and E. Meijer. New York, NY: Springer-Verlag. 可以在 *http://homepages. inf.ed.ac.uk/ wadler/ papers/ marktoberdorf/baastad.pdf*找到。

Walden, Kim, and Jean-Marc Nerson. 1994. *Seamless Object-Oriented Software Architecture*. Upper Saddle River, NJ: Prentice Hall. 可以在*http://www.bon-method.com/index_normal.htm*找到。

第14章

重读经典

Panagiotis Louridas

似乎在所有的科学领域中，都有一些著作和人名是不能不提的。目前还活着的冠军可能是Norm Chomsky。根据1992年4月在MIT Tech Talk上的一篇文章，Chomsky是之前20年中出版的著作中引用最多的人名。艺术和人文学科引用索引的前10名是Marx、Lenin、Shakespeare、Aristotle、the Bible、Plato、Freud、Chomsky、Hegel和Cicero。在科学引用索引中，他在1972—1992年间，被引用了1619次。

在软件工程方面，勋章可能要颁给《设计模式：可复用面向对象软件的基础》（即“Gang of Four”书 [Gamma等 1994]，（编辑注））。在Google上精确查询该书的书名得到了173 000条结果（在2008年春天）。如果我们把注意力放在更为学术化的方面，查询ACM Digital Library得到了1527条结果。设计模式社区是过去20年中软件工程领域最有活力的社区之一。

本章再次增加了这两项的引用计数。

人们可以把设计模式的普及归功于《设计模式》一书。它不仅可以看作是设计模式运动的起始点，也可以看作是返回点：与设计模式相关的内容非常多，但关于这个主题的大多数讨论都是围绕这本书中列出的设计模式。我们并不是排除其他模式，但这本书中提到的23个模式肯定可以称之为经典。

但是这本书中迷人的部分不是那些设计模式本身，而是第1章简介，它提供了许多设计

编辑注：　此书由机械工业出版社引进出版，书号为978-7-111-21126-6。

模式背后的理由，也提供了将它们串起来的通用线索。从中我们可以发现可复用面向对象设计的原则。其中第二个原则就是更倾向使用对象组合（“包含”关系），而不是类继承（“是一种”关系）。（第一个原则是“针对接口编程，而不是针对实现编程”，对于过去40年中看到过有关封装的建议的人来说，这一点应该很清楚了。）

对于那些没有跟随面向对象编程在20世纪八九十年代来到中心舞台的程序员来说，这条规则可能似乎没有那么重要。但如果你还能回忆起那个年代，在面向对象编程中的一个决定性概念就是继承。以Bjarne Stroustrup在《The C++ Programming Language》(1985)中的描述为例。我们发现有这样的内容：

> C++是一种通用目的的编程语言，侧重于系统编程。它：
>
> - 是更好的C语言。
> - 支持数据抽象。
> - 支持面向对象编程。
> - 支持泛型编程。

如果我们想弄清楚“支持面向对象编程”究竟指的是什么，我们发现：

> 编程方式是：
>
> - 确定你想要的那些类。
> - 为每个类提供一组完整的操作。
> - 通过继承来明确那些共性。

现在对比一下另一本经典著作，Joshua Bloch的《Effective Java》(2008)。我们发现至少有三点劝诫是针对继承的：

- 倾向于组合而非继承。
- 设计继承并为之编写文档，否则不要用继承。
- 倾向于接口而非抽象类。

那么继承应该避免使用吗？这在学术上没有定论。在Microsoft Windows上编程可能会让人非常沮丧，即使是利用很舒服的书（Charles Petzold的《Programming Windows》[1999]）中介绍的方法。当第一批Windows编程框架推出时（源自Borland和Microsoft），它们就像是一股清新的空气。在那之前，创建一个简单的窗口都极其复杂：程序员们满怀兴趣地得知，要在Microsoft Windows中编程，他们必须面对所谓的window类，而这与C++类没有任何关系。在新的框架中，你只需创建一个子类，继承框架提供的类，事

情就完成了。我们很高兴能够突然摆脱所有的苦工（或者说几乎所有的苦工），也很高兴能够突然发现面向对象的这样一种简洁应用方式。

在Microsoft Windows上编程只是一个例子，对面向对象和继承的热情无所不在。很奇怪，我们现在知道我们一直搞错了，但也许没有错得那么离谱。继承在本质上可能并不坏。像所有技术一样，它可以用得好，也可以用得不好，不好的继承已经在许多地方都有讨论了（《设计模式》是一个不错的开始）。这里我们将介绍一个漂亮软件系统的例子，它以继承作为基础。这个系统就是Smalltalk。

Smalltalk是一种纯面向对象语言，尽管它从未成为主流语言，但却以多种方式影响了编程语言的发展。也许另一种对后来的计算机语言产生了这么多影响的编程语言就是Algol60，它的影响也超过了它的实际使用。

这不是对Smalltalk编程语言及其环境的介绍（这两方面实际上是在一起的），而是介绍其中基本的架构思想，以及这些思想如何为我们的编程工作提供指导。借用设计心理学的一个术语，这里讨论的是基本设计原则和它们提供给程序员的“可操作暗示”(affordance)。Donald Norman在《The Psychology of Everyday Things》(1988)中透彻地（并且有趣地）解释了可操作暗示的概念。简而言之，一个对象的出现允许我们（有时甚至是诱使我们）做某些事情。悬挂的绳子诱使我们走过去拉一下，水平的把手诱使我们推一下，门把手诱使我们走过去旋转它。同样，编程语言表现出来的样子诱使程序员利用它做某些事情。打造得很漂亮的语言拥有美丽的架构，这会在我们用它编写的程序中体现出来。

这种思想的另一个强烈表达方式是赛故二氏假说（shapir-whorf Hypothesis，SWH），它声称语言决定思想。这一综合症已经让语言学家和编程语言设计者们兴奋了好些年了。《The C++ Programming Language》第1版的前言就是从SWH开始的，K. E. Iversion的1980年图灵奖演讲谈的就是表示法对表达思想的重要性。SWH是有争议的，毕竟，每个人都曾经遇到过找不到一些词来表达自己的思想的情况，所以我们能思考的超过了我们能说的。但是在计算机代码中，语言和程序的关系是清楚的。我们知道一些计算机语言是图灵完备的，但我们也知道，对于某些事情来说，一些语言比另一些语言更合适。

但除了影响程序的架构之外，语言架构的本身也是很有趣的。我们来看看Smalltalk自身的架构，即它的实现选择、设计概念和模式。今天，我们会在最近的编程语言中看到其中的多个方面，那些现在不再见到的方面则让我们能够停下来，反思一下它们消失的原因。

我们在这里并不假定你已具备Smalltalk的相关知识，但到这章结束时，我们就已经介绍Smalltalk的主要部分。我们会突出设计原则，并通过小代码片段来展现这些原则。强大的设计原则有一个好处，就是要学的东西不多，而当你一旦掌握了这些原则，整个基础

架构就从这些原则中自然生成了。我们要参考的Smalltalk系统是Squeak(http://www.squeak.org)，它是一个开放源代码的实现。某些代码示例在第一次阅读时可能难以理解，因为我们会引入一些非常规的概念，但它们会在随后的示例中得到说明，所以最好是努力读到最后，然后再回来读那些不太能理解的部分。同时，我们没有低估读者的智商。

研究Smalltalk将发现一些特征，这些特征在你喜欢的语言里不一定有。这应该不是一个问题。在软件开发中有一条经过时间检验的准则，即你要使用的某一项特征在你所使用的语言中不一定直接支持，经过一些努力，你会在你所选择的语言中找到一种优雅的替代方式。根据Steve McConnell的《Code Complete(2004)》，这被称为“编程为一种语言”：

> 理解“用一种语言编程”与“编程为一种语言”之间的差异是很重要的……绝大多数重要的编程原则不取决于具体的语言，而是取决于你使用它们的方式。如果你的语言缺少你想使用的结构，或者易于出现其他类型的问题，请试着进行补偿。发明你自己的编码规范、标准、类库和其他扩展。

实际上，与SWH相对的是对程序员创造性（或执着）的推崇。笔者记得，当面向对象在20世纪90年代成为一种必需品时，本地技术书店到了一本书，主题是面向对象的汇编语言。最近，Randall Hide的“High Level Assembler(HLA)”将汇编与类、继承和其他技术结合在了一起。

我们对待一门编程语言的方法就像我们对待一本经典书籍一样。在我们急切开始之前，让我们先来看看Italo Calvino的文章“Why Read the Classics”(1986)中提出的一些定义：

> 经典作品是那些你经常听人家说“我正在重读……”而不是“我正在读……”的书。
>
> 经典作品是这样一些书，它们对读过并喜爱它们的人构成一种宝贵的经验；但是对那些保留这个机会，等到享受它们的最佳状态来临时才阅读它们的人，它们也仍然是一种丰富的经验。
>
> 经典作品是一些产生某种特殊影响的书，它们要么自己以遗忘的方式给我们的想象力打下印记，要么乔装成个人或集体的无意识隐藏在深层记忆中。
>
> 一部经典作品是一本每次重读都好像初读那样带来发现的书。
>
> 一部经典作品是一本从不会耗尽它要向读者说的一切东西的书。
>
> 经典作品是这样一些书，它们带着以前的解释的特殊气氛走向我们，背后拖着它们经过文化或多种文化（或只是多种语言和风俗习惯）时留下的足迹。
>
> 一部经典作品不一定要教一些我们不知道的东西；有时候我们在一部经典作品中发现我们已知道或总以为我们已知道的东西，却没有料到这个作者早就说了，或那个想法与那部经典作品有一种特殊联系。这种发现同时也是非常令人满足的意外，例如当我们弄清楚一个想法的来源，或它与某个文本的联系，或谁先说了，我们总会有这种感觉。

> 经典作品是这样一些书，我们越是道听途说，以为我们懂了，当我们实际读它们时，我们就越是觉得它们独特、意想不到和新颖。
>
> 一部经典作品是一部早于其他经典作品的作品，但是那些先读过其他经典作品的人，一下子就认出它在众多经典作品的系谱图中的位置。

当然，书籍不是计算机语言，但这些定义可能仍适合于我们。

14.1 所有东西都是对象

今天流行的面向对象计算机语言（C++、Java和C#）并不是纯面向对象的。并非所有东西都是对象。某些类型是原生类型。因此，举个例子说，我们不能为一个整数类型提供子类。运算以一般的方式在纯数字上执行，而不是调用对象的方法。这带来了性能上的好处，并且对于从过程式语言转向面向对象的人来说，这也比较容易理解。

但是，如果我们决定将所有东西都作为一个对象，那么情况就大为不同。在Smalltalk中，不超过31位长的整数是SmallInteger类的实例（实际上，存在一个抽象的Integer类及其子类SmallInteger、LargePositiveInteger和LargeNegativeInteger，系统会根据需要自动进行转换）。我们可以对它们执行普通的运算操作。SmallInteger类提供了至少670个方法（用Smalltalk的说法，是selector——选择器），我们可以很容易地发现下面的代码片段：

```
SmallInteger allSelectors size
```

考察这段代码如何工作是很有意义的。allSelectors是一个类选择器，它完成的功能正如其名。它返回一个类的所有选择器，放在一个Set中（实际上是一个IdentitySet，但在这里对我们并不重要）。这个Set本身又是一个一级对象，带有自己的选择器。其中的一个选择器名为size，告诉我们它包含的元素的个数。

在SmallInteger的选择器中，我们可以找到期望的运算操作。我们也可以找到三角和对数函数、计算阶乘的函数、计算最大公约数的函数和计算最小公倍数的函数。还有位操作的函数和许多其他函数。

我们在其他语言中遇到的整型原生类型在Smalltalk中实际上都是SmallInteger实例。这解释为什么

```
2 raisedTo: 5
```

能工作，更有趣的是，这也解释了为什么下面的代码也能工作：

```
(7 + 3) timesRepeat: [ Transcript show: 'Hello, World'; cr ]
```

带参数的选择器在每个参数前面有一个冒号(:)。算术运算和逻辑运算的选择器，如上例中的+，是这条规则的例外情况。Transcript是一个类，表示类似系统控制台这样的

东西。cr代表回车，分号(;)将信息连接在一起，所以cr也被发往Transcript。我们可以在一个解释窗口（在Smalltalk中通常称为工作空间，）中执行这段代码，直接看到结果。

当然，并不是所有670个SmallInteger的方法都定义在SmallInteger中。SmallInteger处于一个继承层次结构中，如图14-1所示，其中我们也可以看到SmallInteger的每个祖先的选择器数目。大多数选择器继承自Object，理解Object提供了什么功能就可以解释Smalltalk架构的许多方面（在Squeak中，继承层次结构真正的根是ProtoObject，但这只是一个不太重要的细节）。

Object的实例都有自己可以自由处理的比较选择器（包括相等，用=表示，也包括标识符相同，用==表示）；复制选择器（包括深复制，即调用deepCopy，也包括浅复制，即调用shallowCopy）；在流中打印的选择器、出错处理、调试、消息处理及其他选择器。这几百个对象方法中，只有少数是日常编程会用到的。Smalltalk中的方法被分成组，称为协议（protocol），查看协议描述让我们能够更容易地找到方法。

在Smalltalk中，方法本身也是一级对象。为了弄清楚这一点对于整体架构的意义，请看下面的代码：

```
aRectangle intersects: anotherRectangle
```

其中aRectangle和anotherRectangle都是Rectangle类的实例。当消息接收者aRectangle（在Smalltalk中接收消息的对象）收到intersects:消息时，Smalltalk解释器会做下面的事情（Conroy和Pelegri-Llopart 1983）：

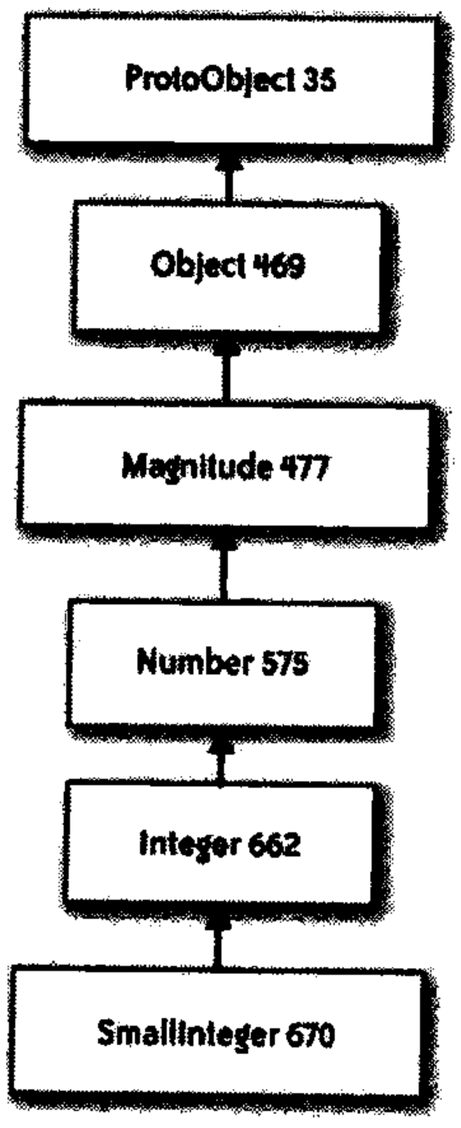


图14-1：SmallInteger继承层次结构

1. 确定消息接收者的类。

2. 在该类及其祖先类中查找消息选择器。

3. 取得与找到的类的消息选择器相关联的方法。

在Smalltalk中，不仅像数字这样的东西是对象，类也是对象。所以，SmallInteger、Object、Rectangle等都是对象。当解释器在一个类中查找消息选择器时（上面列表中的第2步），它会查找对应的类对象的内容。更准确地说，会在它的方法字典中查找。Dictionary类的实例将键和值关联在一起，方法字典将每个选择器与对应的CompiledMethod实例关联在一起。

顺便说一下，intersects:在Smalltalk中可以优雅地实现为：

```
(origin max: aRectangle origin) < (corner min: aRectangle corner)
```

要明白它的工作原理，你需要知道origin选择器返回矩形左上角的顶点（类Point的一个实例），corner选择器返回矩形右下角的顶点，max:返回消息接收者和参数唯一确定的矩形的右下角，min返回消息接收者和参数唯一确定的矩形的左上角。虽然这种实现很简明，但它不是最优化的解决方案，Squeak提供了另一种方法：

```
intersects: aRectangle
    "Answer whether aRectangle intersects the receiver anywhere."
    "Optimized; old code answered:
        (origin max: aRectangle origin) < (corner min: aRectangle corner)"

    | rOrigin rCorner |
    rOrigin := aRectangle origin.
    rCorner := aRectangle corner.
    rCorner x <= origin x ifTrue: [^ false].
    rCorner y <= origin y ifTrue: [^ false].
    rOrigin x >= corner x ifTrue: [^ false].
    rOrigin y >= corner y ifTrue: [^ false].
    ^ true
```

这种方法更快，但不那么漂亮。然而，这让我们有机会介绍一些Smalltalk的语法。方法局部使用的变量声明在| |中。赋值操作符是:=，^与C++和Java中的return等价，小数点(.)分隔语句。方括号([ ])之内的代码称为一个语句块，这是Smalltalk架构中的一个重要概念。语句块是一个闭包，也就是一段代码，可以访问围绕它的范围内定义的变量。Smalltalk中的语句块由BlockContext来表示。当语句块对象接收到消息值时，语句块的内容就得到执行，在大多数情况下（像这个例子中），消息是隐式发送的。Smalltalk中的注释放在双引号内，单引号用于表示字符串。

BlockContext可以使用消息接收者、参数和临时变量，也可以使用创建它的上下文中的消息发送者。有一个类似的类，名为MethodContext，表示了与方法（正如我们前面看到的，它由CompiledMethod表示，是一个字节码数组）执行相关的所有动态状态。

在一种面向对象的语言中，我们会想到，BlockContext和MethodContext都是ContextPart的子类。ContextPart在其超类InstructionStream的基础上添加了执行语义。InstrucionStream类的实例可以解释Smalltalk代码。Instruction Stream的超类是Object，继承关系到此为止。

除了value选择器之外，语句块还有fork选择器，它在语言中实现了并发。因为Smalltalk中的所有东西都是对象，所以进程就是Process类的实例。Delay类让我们可以将进程的执行挂起一段时间，当它收到wait消息时，Delay对象会将当前执行的进程挂起。将这些结合起来，我们可以利用下面的代码实现一个简单的时钟（Goldberg和Robson，1989）：

```
[[true] whileTrue:
    [Time now printString displayAt: 100@100.
    (Delay forSeconds: 1) wait]] fork
```

只要whileTrue:选择器的消息接收者语句块的值为true，它就会执行它的语句块参数。@字符是Number类的一个选择器，它将构造Point类的实例。

语句块也为我们提供了基本的错误处理功能，思路就是：当某些东西出错时，我们指定执行某些语句块。例如，在Collection对象（一个对象容器）中，remove:方法会从集合中删除指定的元素。方法remove:ifAbsent:会尝试从集合中删除指定的元素，如果该元素不存在，它就会执行作为ifAbsent:参数的语句块。在一个代码最少化的漂亮例子里，第一个方法是基于第二个方法来实现的：

```
remove: oldObject
    "Remove oldObject from the receiver's elements. Answer oldObject
    unless no element is equal to oldObject, in which case, raise an error.
    ArrayedCollections cannot respond to this message."

    ^ self remove: oldObject ifAbsent: [self errorNotFound: oldObject]

remove: oldObject ifAbsent: anExceptionBlock
    "Remove oldObject from the receiver's elements. If several of the
    elements are equal to oldObject, only one is removed. If no element is
    equal to oldObject, answer the result of evaluating anExceptionBlock.
    Otherwise, answer the argument, oldObject. ArrayedCollections cannot
    respond to this message."

    self subclassResponsibility
```

self是对当前对象的引用（与C++和Java中的this等价），它是一个保留的名称，是具有固定语义的伪变量。还有其他一些伪变量：super是对超类的引用（等价于Java中的super）；nil、true和false的意思很明白；最后还有thisContext，我们可以在subclassResponsibility方法中看到它。这个选择器定义在Object中，只是说明子类必须覆写它。

```
subclassResponsibility
    "This message sets up a framework for the behavior of the class's subclasses.
    Announce that the subclass should have implemented this message."

    self error: 'My subclass should have overridden ', thisContext
sender selector printString
```

thisContext伪变量是对当前执行上下文的引用，即当前执行的方法或语句块，所以它是MethodContext或BlockContext的当前执行实例的引用。sender选择器返回消息发送的上下文，selector给出了该方法的选择器。所有这些都是对象。

将代码当作对象来处理不是新思想，Lisp的强大之处就在于它对代码和数据一视同仁。它允许我们利用反射来编程，也就是说，进行元编程（metaprogramming）。

随着时间的推移，元编程这种思想已经变得越来越重要。C++中的模板元编程是一种相当不同的方式：我们利用了C++编译器会在编译时生成模板代码这一事实，在编译时执行计算（Abrahams和Gurtovoy 2005）。这种技术带来了令人兴奋的可能性，但却需要复杂的编程技能。在Java中元编程是通过反射来实现的，它是语言的一部分，虽然使用反射的Java代码可能会比较麻烦。

当我们需要在运行时刻构建一个菜单时，一个相关的问题就凸显出来了。菜单会将菜单项和处理代码关联起来，当用户选择相关的菜单项时，就会执行对应的处理代码。如果我们能够通过名称来引用这些处理代码，我们就能够使用下面这样的代码动态地构建一个菜单：

```
CustomMenu new addList: #(
    #('red' #redHandler)
    #('green' #greenHandler)
    #('blue' #blueHandler)); startUpWithCaption:'Colors'.
```

在Smalltalk中，分隔符#()代表了数组。我们通过一个包含多个标签-处理代码对的列表，创建了一个新的菜单及其菜单项。这些处理代码是选择器（在真实的代码中，我们需要给出它们的实现）。处理代码前面加上了#字符，这在Smalltalk中表示符号（symbol）。我们可以认为符号和字符串类似。符号通常用于代表类或方法的名称。

反射让我们能够以简洁的方式来实现抽象工厂（Abstract Factory）设计模式（Alpert等1998）。如果我们需要一个工厂类，它在运行时可以根据用户的指定来实例化类的对象，那么我们可以采用下面的方式：

```
makeCar: manufacturersName
    "manufacturersName is a Symbol, such as #Ford, #Toyota, or #Porsche."
    | carClass |
    carClass := Smalltalk
                    at: (manufacturersName, #Car) asSymbol
                    ifAbsent: [^ nil].
    ^ carClass new
```

当用户给出一个制造商的名称时，我们在它后面加上单词"Car"，创建类名。对于#Ford来说，类名将是#FordCar；对于#Toyota来说，类名将是#ToyotaCar，等等。字符串连接在Smalltalk中是用逗号(,)表示的，我们希望连接得到的字符串成为一个符号，所以我们调用它的asSymbol方法。在Smalltalk中，所有的类名都存在Smalltalk字典中，它是SystemDictionary类的唯一实例。当我们找到要求的类名时，我们就返回该类的一个实例。

我们已经看了几个Smalltalk的例子，但还没有看到类定义。在Smalltalk中，类的构建方式和其他东西的构建方式是一样的：向相应的消息接收者发送必要的消息。我们先在Smalltalk环境中，从完成下面的模板开始：

```
NameOfSuperclass subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Unknown'
```

我们用真实的名称替代NameOfSuperclass和NameOfSubclass。在instanceVariableNames中，我们列出实例变量，在classVariableNames中列出类变量，在category中标出类所属的分类（在Smalltalk中，类被归到一些分类中，类似于其他语言中的命名空间或包）。poolDictionaries中列出了我们与其他类共享的字典，这是在Smalltalk中共享变量的机制。当该模板的各项细节填入之后，它们就被传递给Class类的subclass选择器。

```
subclass: t instanceVariableNames: f classVariableNames: d
poolDictionaries: s category: cat
    "This is the standard initialization message for creating a new class as a
    subclass of an existing class (the receiver)."
    ^(ClassBuilder new)
    superclass: self
    subclass: t
    instanceVariableNames: f
    classVariableNames: d
    poolDictionaries: s
    category: cat
```

subclass选择器会创建一个ClassBuilder类的实例，它将创建新类或修改已有的类。我们向ClassBuilder实例发出必要的信息，这样就可以根据我们在类模板中输入的信息创建新的类。

在对象上做所有事情都是通过发送消息来完成，这使我们需要把握的概念不是很多。它也让我们限制了语言中语法结构的数量。让编程语言最小化有很长的历史。在关于Lisp的第一篇论文（McCarthy 1960）中，我们发现Lisp比较了两类表达式：一是S表达式（或句法表达式），它是从列表构建的表达式；二是M表达式（或元表达式），它是将S表

达式作为数据的表达式。最后，程序员们总是选择使用S表达式，所以Lisp就变成了我们今天所知道的样子：一种几乎没有语法的语言，因为所有的东西，包括程序和数据，都是列表。它取决于你对Lisp的态度，这证明了一个简单的思想就足够表达最复杂的结构（或者人们可以被强迫接受任何东西）。

但是，Smalltalk不局限于一种句法元素，Smalltalk程序由6种构建块组成：

1. 关键字、或伪变量，只有6个（self、super、nil、true、false和thisContext）。
2. 常量。
3. 变量声明。
4. 赋值。
5. 语句块。
6. 消息。

我们在这个列表中没看到的东西可能比看到的东西更有趣：我们没有看到任何表示控制流的元素，没有条件判断或循环。它们不需要，因为它们是通过消息、对象和语句块（都是对象）来实现的。下面的方法在Integer类中实现了阶乘函数。

```
factorial
    "Answer the factorial of the receiver."

    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
    self error: 'Not valid for negative integers'
```

操作符=和>是消息选择器，它返回抽象类Boolean的对象，Boolean有两个子类，True和False。如果选择器ifTrue:的消息接收者是True的一个实例，那么它的参数就会被执行。它的参数是[]界定的语句块。还有一个对称的选择器ifFalse:，它具有相反的语义。一般来说，使用循环要好于使用递归，所以下面是阶乘函数的循环实现：

```
factorial
    "Implement factorial function using a loop"

    | returnVal |
    returnVal := 1.
    self >= 0
        ifTrue: [2
                    to: self
                    do: [:n | returnVal := returnVal * n]]
        ifFalse: [self error: 'Not valid for negative integers'].
    ^ returnVal
```

大部分工作都在两个语句块内完成。第一个语句块针对从2到消息接收者的值为止的正数执行。每次迭代的值都被传入内层的语句块，内层语句块将计算结果。语句块参数之

前有冒号(:)，与语句块体之由短竖线(|)分隔。可以有多个语句块参数，如下面的阶乘定义（Black 等 2007）：

```
factorial := [:n | (1 to: n) inject: 1 into: [:product :each | product * each ] ].
```

to:选择器返回类Interval的一个实例，它实际上列出了从1～10的值。对于数值*n*，阶层语句块将执行下面的动作。首先，它将内层语句块的product参数设为1。然后它将针对从1～*n*的值，调用内部语句块，计算每次迭代数的乘积和当前的乘积，并将结果保存到product中。要计算10的阶乘，我们需要写下阶乘值：10。借用Herbert Simon在《Sciences of the Artificial》(1996)中对更早的荷兰医生Simon Stevin的引用：

> *Wonder, en is gheen wonder.* 这句话的意思是："漂亮，但并不难懂。"

14.2 类型是隐式定义的

尽管所有东西在Smalltalk中都是对象，甚至类也是，但类并不对应于C++和Java这类语言中的类型。类型是根据它们的行为隐式定义的，而不是根据它们的接口。这被称为潜在类型或鸭子类型。

潜在类型是Smalltalk中唯一的类型机制（在其他一些动态类型语言中也是如此），但这并不意味着它对于强类型语言来说不重要。例如，在C++中，潜在类型是通过模板的泛型编程的基础。看到它第一次出现在这种语言中是很有意义的。请看下面的C++模板的示例介绍（Vandervoorde和Josuttis，2002, 2.4）

```
// maximum of two int values
inline int const& max (int const& a, int const& b)
{
    return a < b ? b : a;
}

// maximum of two values of any type
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// maximum of three values of any type
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);
}

int main()
{
    ::max(7, 42, 68); // calls the template for three arguments
    ::max(7.0, 42.0); // calls max<double> (by argument deduction)
```

```
    ::max('a', 'b'); // calls max<char> (by argument deduction)
    ::max(7, 42); // calls the nontemplate for two ints
    ::max<>(7, 42); // calls max<int> (by argument deduction)
    ::max<double>(7, 42); // calls max<double> (no argument deduction)
    ::max('a', 42.7); // calls the nontemplate for two ints
}
```

我们在main()方法中可以看到，函数::max适用于实现了比较操作符的所有类型。在C++中，这些类型可以是原生类型，也可以是用户定义的类型。没有限制它一定要继承自某个特定的类。它可以是任何类型，只要它在比较方面满足基本需求。隐含的类型定义是：操作符<有意义的任何类型。

我们在学校中曾学到，有两种方法定义一个集合。一种是显式的，列出集合的元素，这被称为客观式定义。我们知道，自然数集是{1, 2, 3, …}。另一种定义集合的方法是描述集合中元素的共性，这被称为主观式定义。自然数集的主观式定义是“大于零的整数”。当我们在C++中声明一个对象时，我们通常使用客观式定义：我们说这个对象属于某个指定的类型。但当我们使用模板时，我们实际上使用了主观式定义：我们说这段代码适用于一个对象集，其中的对象具有某种属性（即它们提供了要求的操作）。

遗憾的是，在Java中情况比较混乱。Java提供了泛型编程，但只是在表面上和C++模板的语法相似。Bruce Eckel已经有说服力地解释了这个问题（参见*http://www. mindview. net/WebLog/log-0050*）。像Python这样的语言支持潜在类型，我们可以这样做：

```
class Dog:
    def talk(self): print "Arf!"
    def reproduce(self): pass

class Robot:
    def talk(self): print "Click!"
    def oilChange(self): pass

a = Dog()
b = Robot()
speak(a)
speak(b)
```

两次对speak()的调用都能工作，我们不必关心其参数的类型。我们可以在C++中做同样的事情：

```
class Dog {
public:
    void talk() { }
    void reproduce() { }
};

class Robot {
public:
    void talk() { }
```

```
    void oilChange() { }
};

template<class T> void speak(T speaker) {
    speaker.talk();
}

int main() {
    Dog d;
    Robot r;
    speak(d);
    speak(r);
}
```

我们不能在Java中做同样的事情，因为下面的代码不能通过编译：

```
public class Communicate {
    public <T> void speak(T speaker) {
        speaker.talk();
    }
}
```

令人困惑的是，下面的代码能够通过编译，因为在Java中，泛型类型被悄悄转换成了Object的实例（这叫做擦除）：

```
public class Communicate {
    public <T> void speak(T speaker) {
        speaker.toString(); // Object methods work!
    }
}
```

所以我们必须像下面这样做：

```
interface Speaks { void speak(); }

public class Communicate {
    public <T extends Speaks> void speak(T speaker) {
        speaker.speak();
    }
}
```

但是这在很大程度上减少了泛型的好处，因为我们要通过Speak接口来定义一个类型。缺乏一般性也体现在Java的原生类型不能使用泛型机制。作为一种变通方案，Java提供了包装类，即与原生类型对应的真正的对象类。在原生类型和包装类型之间的转换在Java编程中曾是令人不快的工作。在该语言最近的版本中情况有所好转，因为提供了自动装箱（auto-boxing）功能，在特定的环境下进行自动的转换。尽管如此，我们可以写List<Integer>，却不能写List<int>。

潜在类型最近很流行，因为在Ruby编程语言中大量采用。术语“鸭子定型”（duck typing）是一种对归纳推理的半开玩笑的说法，源自James Whitcomb Riley，原文是这样的：

如果它走路像鸭子，而且叫起来像鸭子，我就会称之为鸭子。

要了解鸭子类型的重要性，让我们以面向对象编程的一项基本特征——多态——为例。多态指的是在相同的上下文中使用不同的类型。实现多态的一种方法是通过继承。子类可以用在（准确地说是"应该能用在"，因为程序员可能不小心）所有可以使用超类的地方。鸭子类型提供了另一种实现多态的方法：只要一个类型提供的方法适用于某个上下文，它就可以用于该上下文。在前面用Python和C++写的宠物和机器人的例子中，Dog和Robot没有共同的超类。

当然，你可以通过编程在只支持继承类型的多态的语言中变通地实现鸭子类型。但是，如果程序员在解决面对的问题时可以使用更多的工具，那他就更富有。只要工具的流行程度不是问题，他就可以选择最合适当前情况的工具。关于这一点，Bjarne Stroustrup在《The Design and Evolution of C++》(1994)中表达得很漂亮：

> 我对计算机和编程语言的兴趣基本上是实用主义的。
>
> 我更习惯经验主义而不是理想主义…也就是说，我更喜欢亚里士多德而不是柏拉图，更喜欢休谟而不是笛卡儿，对帕斯卡只能难过地摇摇头。我发现一些完备的“系统”就像柏拉图和康德梦想的那样，但基本上不能让我满意，因为它们离我的日常经验极其遥远，也与个人的基本特点相去甚远。
>
> 我发现克尔凯郭尔对个人的狂热关注和敏锐的心理学洞见对我来说更有吸引力，远甚于黑格尔或马克思对人性的崇高计划和关注。尊重一个群体而不尊重群体中的个人，则根本不是尊重。许多C++的设计决定的根源都在于我不喜欢强制人们用某种特定的方式来做事情。从历史上看，许多最大的灾难都来自于理想主义者试图强迫人们“做对他们有好处的事情”。这种理想主义不仅会导致无辜的受害者遭受痛苦，而且会导致理想主义者应用强制力时的幻觉和腐败。我也发现理想主义者常常倾向于忽略经验和实验，这些经验和实验不巧恰好与教义或理论相抵触。当理想发生冲突时，有时甚至是权威们一致同意时，我倾向于提供支持，让程序员自行选择。

回到Smalltalk，请考虑遍历一个对象集合，对其中每个元素调用一个方法，然后收集结果。实现如下：

```
collect: aBlock
 "Evaluate aBlock with each of the receiver's elements as the argument.
Collect the resulting values into a collection like the receiver. Answer
 the new collection."

 | newCollection |
newCollection := self species new.
self do: [:each | newCollection add: (aBlock value: each)].
^ newCollection
```

要理解这个方法，只要知道species方法要么返回消息接收者的类，要么返回一个与之类似的类——这种区别很小，在这里对我们几乎没有区别。有趣的是，要构造新的集合，我们只需要元素具有名为value的选择器，我们将调用它。语句块确实有一个名为value的选择器，所以所有的语句块都可以使用。但我们只是碰巧提到语句块：任何实现了value选择器的类都可以。

所有返回一个值的东西在编程时具有足够的重要性，所以它们得到了一个名称，被称为函数对象，它是C++ STL算法的基本构成。从传统来看，它是函数式编程的主要成分，通常称为map函数。在Lisp中当然提供这个函数。在Python中也提供这个函数，让我们能做这样的事：

```
def negate(x): return -x map(negate, range(1, 10))
```

在Perl中这样写：

```
map { -$_ } (1..10)
```

C++ STL能做同样的事（Josuttis 1999, 9.6.2）：

```
vector<int> coll1;
list<int> coll2;

// initialize coll1

//negate all elements in coll1
transform(coll1.begin(), coll1.end(), //source range
          back_inserter(coll2),       //destination range
          negate<int>());             //operation
```

也许我们这个时代的悲剧就是许多C++程序员只会写对数组循环的代码。

鸭子定型可能会引起争论，它也确实引起了争论。在像C++这样的静态类型语言中，编译器会检查包含潜在类型的表达式中使用的对象是否确实提供了要求的接口。在像Smalltalk这样的动态类型语言中，当运行出错时才会被发现。

这一点不能不考虑。强类型的语言防止了程序员的疏忽，这在大的项目中特别有用，因为好的结构有助于维护。20世纪80年代，从传统UNIX C转到ANSI C的最重要变化就是引入了更强的类型系统：C有了规矩的原型，函数参数会在编译时进行检查。我们现在会对类型间的转换觉得不舒服。简而言之，我们用一些自由换取了一些纪律，或者说用一些混沌换取了一些有序。

在Smalltalk中，通常假定不会导致混沌，因为我们应该编写小的代码片段，同时对它们进行测试。测试在Smalltalk中很容易。因为没有明显的编译器和构建循环，我们可以在工作空间中写一些小的代码片段，直接观察代码的行为。单元测试也很容易，我们可以在写代码之前先编写单元测试，不必担心编译器会报错说有未声明的类型或方法。给我们带来JUnit的社区与Smalltalk社区分享了很多经验，这也许不是偶然的。我们可以通过

一些Java脚本语言实现差不多同样的东西，例如BeanShell或Groovy。实际上，强类型语言会带给我们安全的错觉：

> 如果程序是以强静态类型的语言编译而成的，那么这仅仅说明它通过了某些测试。这意味着语法保证是正确的……但是不能仅因为代码通过了编译就保证它是正确的。如果你的代码似乎能运行，那也不能保证其正确性。
>
> 正确性的唯一保证，不论你使用的语言是强类型的或弱类型的，就是它是否能通过保证程序正确性的所有测试。（*http://www.mindview.net/WebLog/log-0025*）

但是，有时候谨慎一些是有好处的。我们可以向某个集合添加一些对象，并期望这些对象满足特定的接口；我们不喜欢在运行时才看到其中的某些对象实际上并未满足特定的接口。我们可以利用Smalltalk的元编程机制来确保避免这样的问题。

元编程对潜在类型是一种补充，我们能够检查特定的接口是否真正提供，我们有机会在没有提供特定的接口时作出反应。例如，将未实现的调用代理给我们已知实现了该调用的对象，或只是优雅地处理失败，避免程序崩溃。所以在鹤嘴锄书（pickaxe book）中（Thomas等 2005），我们看到了下面的Ruby示例，试图向一个字符串中添加一些歌曲信息：

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.kind_of?(String)
    fail TypeError.new("String expected")
  end
  unless song.kind_of?(Song)
    fail TypeError.new("Song expected")
  end
  result << song.title << " (" << song.artist << ")"
end
```

这是我们在采用Java或C#编程风格时会做的事。在Ruby风格的鸭子类型中，可能就是：

```
def append_song(result, song)
  result << song.title << " (" << song.artist << ")"
end
```

这段代码适用于所有可以通过<<操作符连接的对象，对于不支持<<的对象，我们会得到一个异常。如果我们真希望采用防御式编程，我们可以检查该对象的能力，而不是检查它的类型：

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.respond_to?(:<<)
    fail TypeError.new("'result' needs `<<' capability")
  end
  unless song.respond_to?(:artist) && song.respond_to?(:title)
    fail TypeError.new("'song' needs 'artist' and 'title'")
```

```
    end
    result << song.title << " (" << song.artist << ")"
  end
```

Smalltalk提供了respondsTo方法，在Object中定义，我们可以用它在运行时判断特定的消息接收者是否具有特定的选择器。

```
respondsTo: aSymbol
    "Answer whether the method dictionary of the receiver's class contains
    aSymbol as a message selector."

    ^self class canUnderstand: aSymbol
```

实现很简单，将检查代理给Behavior中定义的canUnderstand选择器：

```
canUnderstand: selector
    "Answer whether the receiver can respond to the message whose selector
    is the argument. The selector can be in the method dictionary of the
    receiver's class or any of its superclasses."

    (self includesSelector: selector) ifTrue: [^true].
    superclass == nil ifTrue: [^false].
    ^superclass canUnderstand: selector
```

最后，includesSelector:也定义在Behavior中，它最终检查该类的方法字典：

```
includesSelector: aSymbol
    "Answer whether the message whose selector is the argument is in the
    method dictionary of the receiver's class."

    ^ self methodDict includesKey: aSymbol
```

当消息接收者收到一条它不理解的消息时，它的标准反应是向系统发出doesNotUnderstand:消息。如果我们愿意自己处理这种情况，只需要重载这条消息，做类似下面的事情：

```
doesNotUnderstand: aMessage
    "Handles messages not being understood by attempting to
    proxy to a target"
    target perform: aMessage selector withArguments: aMessage arguments].
```

我们假定target指向一个代理对象，我们希望它能够处理发错的消息。

潜在类型不是粗心编程的借口。我们模糊了类型，但并没有模糊责任。

14.3 问题

公有继承意味着“是一种”关系。程序员需要仔细思考，才能够得到符合这种模式的类继承关系。如果你有一个类，它包含某一个方法，而在子类中实现这个方法没有什么意义，那么这就超出了公有继承的范畴，应该是不良设计的标志。

在C++中，你可以创建一个无意义的方法，它要么返回一个错误，要么抛出一个异常，从而避免这种情况出现。典型的例子如下（Meyers 2005, item 32）：

```
class Bird {
public:
    virtual void fly(); // birds can fly
    // ...
};

class Penguin: public Bird { // penguins are birds
public:
    virtual void fly() { error("Attempt to make a penguin fly!"); }
    // ...
};
```

C++程序员也可以选择隐藏这个令人不快的方法：

```
class Base {
public:
    virtual void f()=0;
};

class Derived: public Base {
private:
    virtual void f() {
    }
};
```

这样，Derived不再是一个抽象类，但它仍然没有可供使用的函数f()。这样的诡计应该避免。

在Java中，可以通过返回一条错误或抛出一个异常来避免这一点，也可以在子类中创建一个无意义的方法抽象，这样就使得类层次从那一点开始抽象，向下直到你将该方法标识为具体为止。同样，这样的诡计应该避免。

要处理类层次结构中不合适的或不相关的方法，正确的做法是重新设计这个层次结构。为了更好地反映类鸟类世界的奇特性，类层次结构将为Bird类引入一个FlyingBird子类，让Penguin成为Bird的直接子类，而不是FlyingBird的子类。

在Squeak中，我们发现只有45个方法发出shouldNotImplement消息，该消息在某个继承自超类的方法在当前类中不适用的时候使用。这在Smalltalk的所有对象和方法之中只占了很少一部分，所以这门语言并没有充满设计不佳的类层次结构。然而，即使是shouldNotImplement消息实际上也是一种实现。这暗示了Smalltalk中的一个更深层次的问题，即我们没有真正的抽象类或方法。方法在惯例上是抽象的，根本不存在无实现的方法。

除了使用shouldNotImplement消息之外，我们也可以规定某个方法是子类的职责，这就是我们看到的subclassResponsibility消息的用途。发出subclass Responsi-

bility消息的类按惯例就是抽象类。例如，Collection类给出了一个通用的接口来添加和删除对象，但并没有提供实现，因为实现取决于我们所面对的子类（它可以是字典、数组、链表等）。方法add:将在子类中实现：

```
add: newObject
    "Include newObject as one of the receiver's elements. Answer newObject.
    ArrayedCollections cannot respond to this message."

    self subclassResponsibility
```

这个“抽象的”add:定义甚至允许我们在使用它的Collection方法中定义，如add: with Occurences:，是这样定义的：

```
add: newObject withOccurrences: anInteger
    "Add newObject anInteger times to the receiver. Answer newObject."

    anInteger timesRepeat: [self add: newObject].
    ^ newObject
```

我们甚至可以在根本没有定义add:时这样做，add:withOccurrences:还是像刚才那样定义，只要运行时接收该消息的对象定义了add:，Smalltalk就不会中止。（顺便说一句，add:withOccurences是Strategy模式的一小段漂亮实现。）同时，add:中的注释指出，Collection的某些子类，即以ArrayedCollection子类为根的那些子类，根本不应该实现该消息。同样，这只能在运行时使用shouldNotImplement强制实现：

```
add: newObject
    self shouldNotImplement
```

在编程时运用惯例不是什么根本性的错误，这门艺术的一部分就是要掌握惯例。但可能出现问题的是，只依赖集合来得到要求的结果。如果我们在ArrayedCollection中忘记了实现add:，Smalltalk不会给出警告。我们只会在运行时失败。

我们在前面曾看到，在Smalltalk中实现一个代理类有多么容易。但是实际情况却是，如果我们确实想要一个代理类，让它只作为少数方法的代理，那么事情只会变得更复杂。原因与缺少真正的抽象类有关。代理类可能是我们想代理的那个类的一个子类，在这种情况下，它会继承被代理类的所有方法，而不仅是我们想代理的那些方法。或者，我们可以利用潜在类型，并在代理类中只定义那些被代理的方法；但问题是，由于所有东西都是对象，代理类会继承Object类的所有方法。理想情况下，我们希望代理两个方法的类就只有两个方法，但我们不清楚如何才能做到这一点。这个类会带有从它的祖先那里继承来的所有方法。我们可以求助于一些技巧来减少继承的方法的数量，例如，在一些Smalltalk的方言实现中，可以让某个类成为nil的子类，而不是Object的子类。这样就不会继承任何东西，但我们需要从Object那里复制和粘贴一些必要的方法（Alpert等 1998）。在Squeak中，我们可以让它成为ProtoObject的子类，而不是Object的子类。

当我们开始寻找从哪里继承了什么时，事情就变得复杂了。因为在Smalltalk中所有东西都是对象，包括类也是对象，所以我们可能会想知道类是哪一种实例。我们发现是元类（metaclass）的实例，某个类的元类的名称就是该类的名称加上“class”。例如，Object的元类就是Object class。这是合情合理的惯例，但这并没有真正回答这个问题，因为我们会想知道元类是哪一种实例。我们发现元类是Metaclass类的实例（这里没有class后缀）。Metaclass是另一个类的实例，名为Metaclass class（包含class后缀），而它又必须是某个类的实例，所以它被作为Metaclass的实例。（如果读者认为正在阅读Terry Gilliam的“Twelve Monkeys”的手稿，那是很自然的想法，不必受到责怪。）如果我们同时考虑到继承关系，事情就会变得更复杂（到目前为止，我们还没有提到继承）。我们发现Object类是Class的子类，它又是Class类的一个实例，Class类是ClassDescription的子类，Class类是Behavior的子类，Behavior是Object的子类，在Squeak中，ProtoObject是继承层次结构的最顶端。图14-2展示了这种情形。请注意，我们没有包含Object之下的类。读者可以尝试合并图14-1和图14-2（不要忘记添加元类，从SmallInteger开始向上画）。可能是我们不能够拥有太多的好东西，让所有东西都是对象是很好的，但顺着这个公理走下去，不一定导致容易处理的结构。幸运的是，大多数Smalltalk程序员不需要关注这些事情。

“所有东西都是对象”和“所有事情都由消息完成”的规定还有其他一些后果，这些后果是程序员的日常工作需要关心的。所有稍具编程经验的程序员都会预期：

```
3 + 5 * 7 == 38
```

结果为真，但不巧的是，在Smalltalk中，下面的表达式为真：

```
3 + 5 * 7 == 56
```

原因是二元算术运算符只是数字类的选择器。所以它们像其他消息一样传递，没有什么规定一个二元消息一定要在另一个二元消息之前。确实，当你熟悉了这个游戏规则后，这样的结果是有意义的。但是这还是让人想不明白。在Smalltalk环境中，很容易在工作空间里检查算术运算，但也许最好是程序员永远无需考虑这样的问题。

Smalltalk环境本身是Smalltalk的一项了不起的发明。回到20世纪80年代，那时候图形显示器还很少，大多数编程都在单色文本终端上进行，Smalltalk为基于虚拟机上的一门语言实现了一个图形用户界面。这可能超越了它的时代。虚拟机要再等10年，才由Java带给主流编程社区。GUI赢得了胜利，但只是在硬件价格降下来之后。同时，具备这些要求的环境在当时被认为（也许被认为）不适合大多数的目的。但是还有一些微妙的东西在继续。

Smalltalk环境实际上是多个类构成的生态系统，你可以利用它来工作，也不得不利用它来工作。在Smalltalk中创建一个新类的唯一合理方式，就是利用合适的浏览器找到你想

要的类，并从它继承。你可以使用自己喜欢的编辑器和磨练得很好的技能，以及用各种语言创建的命令行工具，但Smalltalk不同。你可能会喜欢Smalltalk设计者精心提供的类库，但当你换个角度想想时，你可做的事情并不多。在它里面时你感觉很好，但你必须在它里面做所有事情。这可能不是程序员愿意接受的。有可能你在读了本章的前面几页之后，就会冲过去下载并安装Smalltalk，亲自体验一下，然后当你意识到所有东西都不一样、不熟悉时，就会举手投降。

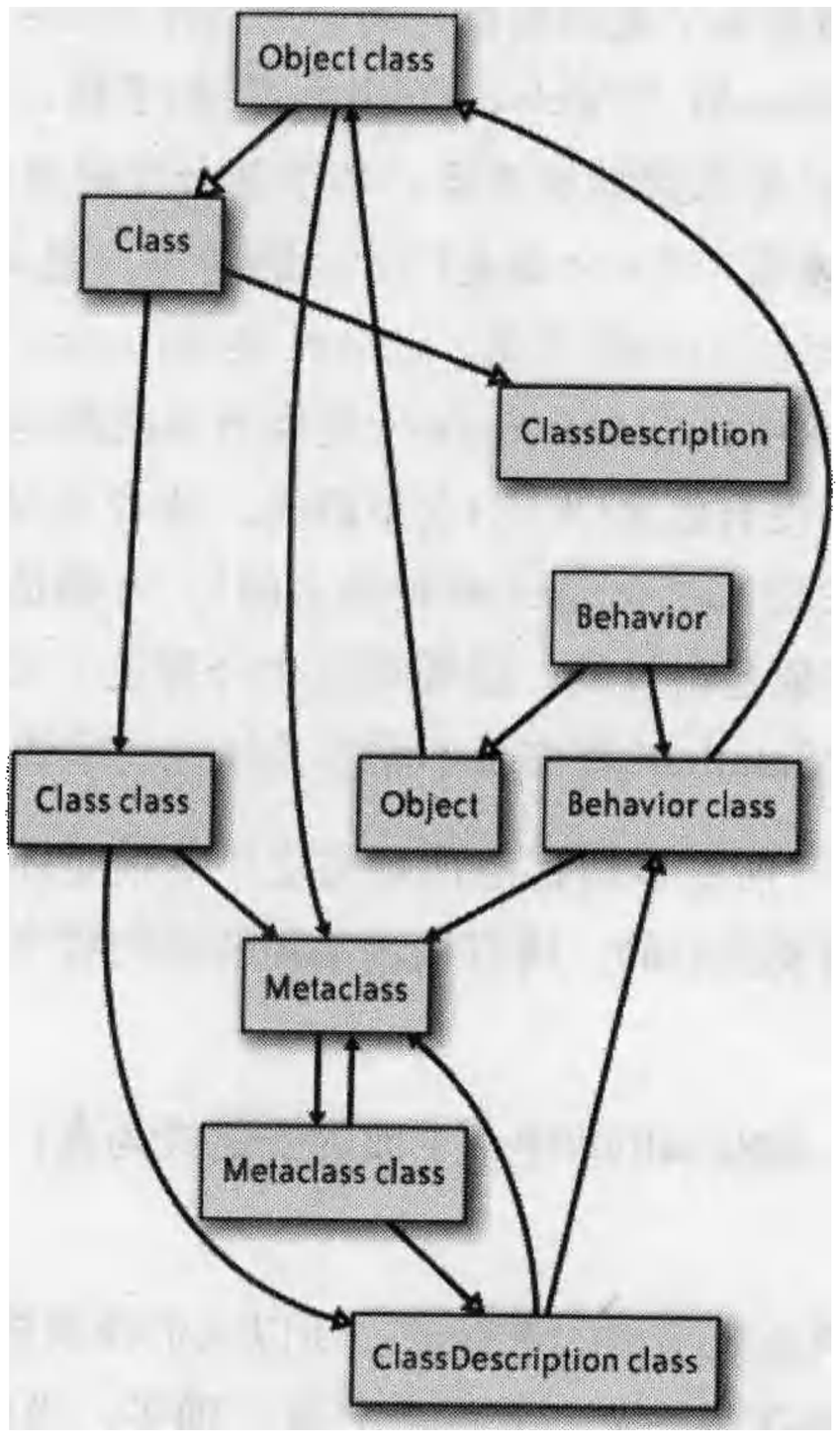


图14-2：Smalltalk中的“意大利面条”

这一点暗示了Smalltalk为什么从未变成主流语言。Smalltalk是一种坚定不移的语言，它不做折中。它定义了一种新的编程模型，它使用的概念后来被许多其他语言所采用，它在很多方面是一个范例，被大家模仿。这一点与建筑界区别不大。

14.4 砖块和灰浆建筑架构

在所有的建筑奇迹中，在美国最受推崇的房子一定是Fallingwater，由Frank Lloyd Wright在1935年设计，“可能是20世纪最频繁出现在举例说明中的房子”（Nuttgens, 1997）。Fallingwater建在名为Bear Run的深谷中的一处瀑布之上。这座房子是为Edgar Kaufmann, Sr.建造的，他是匹兹堡的一名百万富商。在1937年—1963年间，它被用作

Kaufmann一家周末度假的屋子，之后它被捐赠给了Western Pennsylvania Conservancy，在1964作为博物馆向公众开放。

即使是这所房子的照片（参见图14-3）也能让人感觉平静，分享房子与其周围环境的美。Wright努力追求的是让自然与建筑成为一个整体，让艺术和自然交相辉映。在Fallingwater的设计中，自然进入了建筑内部，建筑成为了自然的一部分。这是现代建筑的杰作，仍然吸引我们思考建筑师希望在他的作品中注入的含义。

但房子不只是用来看的，它需要让人居住。我们不能住在Fallingwater中（我们最多只能参观它），但也许我们可以想象住在里面是什么感觉。

很有可能我们是错的。在Steward Brands的《How Buildings Learn》(1995)中我们得知：

> Wright有生之年的最后一件杰作，即宾夕法尼亚的Fallingwater，被AIA投票推举为“有史以来最好的美国建筑”，饱受渗漏的烦恼。渗漏破坏了窗户、石墙，并使水泥结构退化。对于它最初的拥有者来说，Fallingwater被称为“很快发霉”和“七桶之屋”。它确实是一座漂亮而有影响力的房子，但无法居住。（引自Judith Donahue, “Fixing Fallingwater's Flaws,” Architecture, Nov. 1989）

这种评判可能过于苛刻。曾在Fallingwater中住过的Edgar Kaufmann, Jr.所说的话有些不同：

> 一些错误困扰着Fallingwater，但房子特殊的美和它因其建造环境而给居住者带来的喜悦也应该正确评价。住在当中确实有些缺点，需要花些力气来克服。（Kaufmann 1986）

这种评判仍然不能全信。Edgar Kaufmann, Jr.不是利益无关方。他在1934年加入了Wright的Taliesin Fellowship。（23名学徒在1932年来到Spring Green, Wisconsin，开始生活和学习，从此开始了“The Frank Lloyd Wright School of Architecture”，现在这所学校还在。）他向他的父母介绍这所房子，在这所房子设计和建造时，他通常是Wright和他父母之间的中间人。

另一个建筑奇迹可能是20世纪最有影响力的房子（影响力在于它的风格确定了这个世界的现代城市的形态结构），是巴黎之外位于Poissy的萨瓦别墅（Villa Savoye），它由瑞士建筑师Charles-édouard Jeanneret-Gris，即Le Corbusier设计（参见图14-4）。Villa Savoye和Fallingwater一样，设计时是作为周末度假小屋。它是在1928—1931年之间建造的。

Wright对于建筑的职责有着强列的观点，Le Corbusier也有很强烈的想法，但思想却相当不同。Wright探索艺术与自然的关系：

> Le Corbusier发明了一种按比例的系统，即“模度（Modulor)”，它结合了黄金分割、六英尺的人形和和协的比例，形成了一个精巧的科布森法则，通过设想重新考虑了机械化和“自然秩序”。（Curtis 1996）

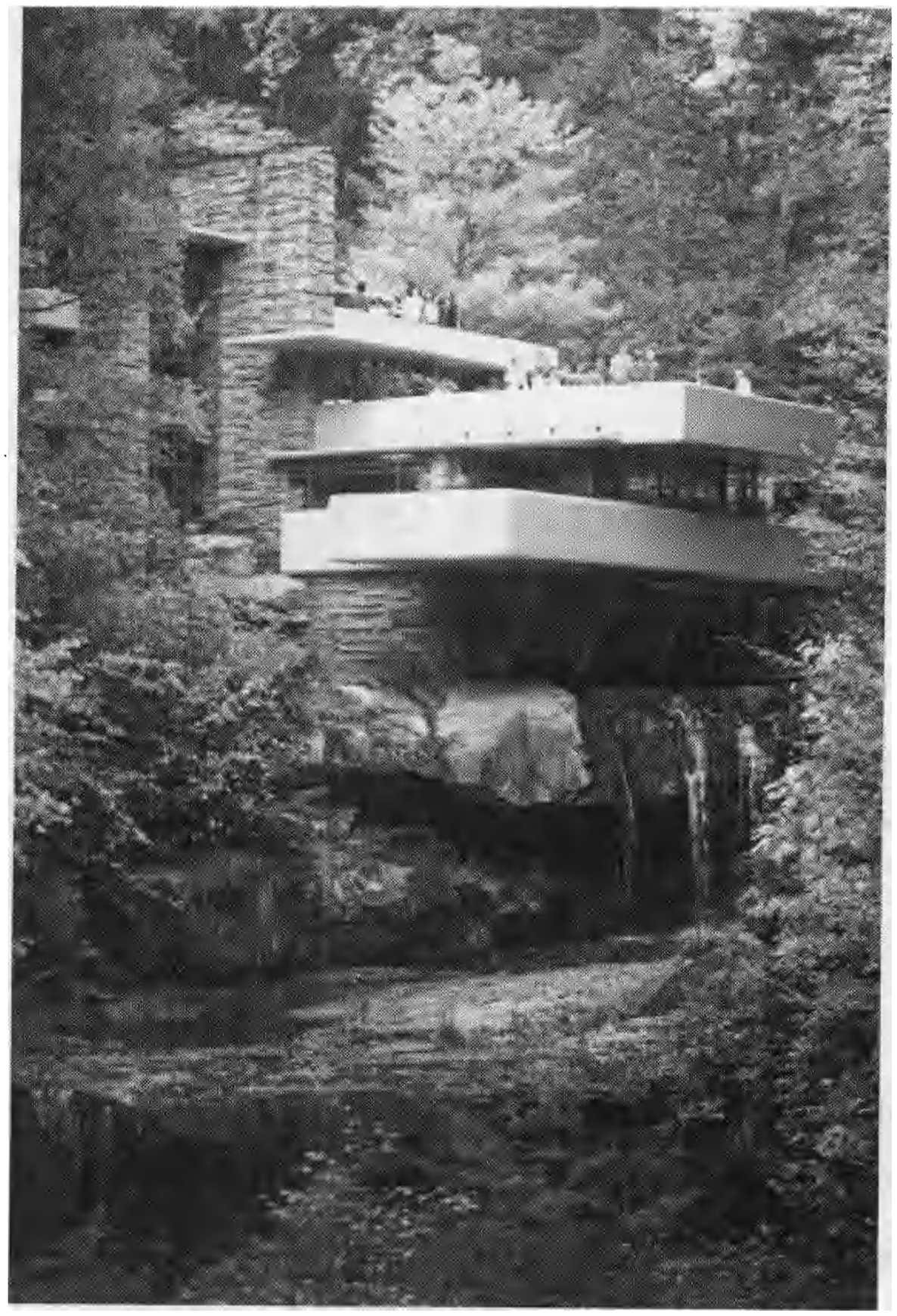


图14-3：Fallingwater

图14-4：萨瓦别墅

萨瓦别墅激起了人们对建筑师的敬畏：

> 像所有高度有序的工作一样，萨瓦别墅不容易归为哪个类别。它既简单又复杂、既睿智又感性。它充满了思想，同时又把这些思想直接通过形体、容积和空间的“特定关系”表达出来。这是现代建筑的一个“经典”时刻，同时又与过去的建筑有着密切的关系。这是Le Corbusier哲学的核心关注点，即对现代生活的观点通过建筑形式的持久价值而表达出来，在萨瓦别墅中人们会意识到它对古老经典主题的回应：静谧、均衡、通透以及简单的横梁结构。(Curtis 1996)

人们可能不同意这样的陈述，对于有些人来说，萨瓦别墅可能看起来就像一个白盒子，已准备吊走，毕竟，讨论品味是没有意义的。从另外一些不同的角度来考虑，除了它作为房子的实用价值之外，我们可能会传递更多明白的评判。下面是Le Corbusier的客户采取的相当不同的观点：

> 尽管萨瓦家一开始就不同意，Le Corbusier还是坚持平顶比斜顶更好（根据技术和经济上的理由来推测）。他向他的客户保证建造的费用更低，更容易维护，在夏天更凉快，而且萨瓦夫人可以在上面跳健身操，不会受到底楼的潮气困扰。但在一家人入住后刚一个星期，屋顶在Roger（萨瓦夫妇的儿子）卧室的位置就开始漏了，进了很多水，导致孩子患上了胸部感染，后来转成了肺炎，最后导致他花了一年的时间在Chamonix的疗养院进行康复。(De Botton 2006)

这听起来就像是建筑师与客户间的一个真实的笑话，但建筑师似乎还有一条窄路逃走：

> 在1936年，别墅正式完工的6年之后，萨瓦夫人在一封（雨水飞溅过的）信中表达了她对平顶的看法：客厅在下雨，弯道在下雨，车库的墙完全湿透了。更有甚者，我的卫生间仍在下雨，不好的天气里简直是洪水，水从顶灯上掉下来。Le Corbusier保证问题马上就会解决，然后寻找机会提醒他的客户，他的平顶设计在世界各地的建筑评论家那里受到多么热烈的追捧：“你应该在楼的大顶的桌子上放一个签名本，请你的客人在上签下名字和地址。你会发现收集到了许多的漂亮签名”。但这种签名收集的建议对萨瓦一家没带来什么安慰。“在我无数次的申请之后，你终于同意这所你在1929年建造的房子是不能住人的”，萨瓦女士在1937年秋天告诫说。“你的可信度正受危害，而我没有必要为此买单。请马上将它修复得可以居住。我真诚地希望我不必将此事诉诸法律。”只到第二次世界大战爆发，萨瓦一家后来飞离了巴黎，Le Corbusier才有幸不必在法庭上应诉。这一切只因他设计了基本上无法居住的、特别漂亮的“居住机器”(machine-for-living)(De Botton 2006)

另一名现代建筑的名誉领袖Ludwig Mies van der Rohe，使用了基于I型金属梁的极简主义规范系统。他“在书桌旁边保留了全尺寸I型金属梁的详细信息以获得恰好那样的比例。他认为“I型金属梁”是多利安式圆柱（Doric Column）的现代等价物”(Jencks 2006)。

更著名的是他对“少即是多”的运用，通过这个原则，他希望将建筑带回主要本质：没有装饰、装修或多余的元素，除非它们实现了特定的功能。I型金属梁是建筑的基本部分。

或者说看起来似乎是这样。Mies van der Rohe的一件主要作品是纽约的Seagram大厦，建成于1958（参见图14-5）。建筑师面临了一个难题：尽管他很想在结构中展示I型金属梁，但在美国的建筑法规中却不允许这样做，法规要求金属支撑必须包以防火材料，如混凝土。但很奇怪，仔细看Seagram大厦会看到它表面的I型金属梁。这些不是真正的支撑I型金属梁。Mies van der Rohe要求在表面添加假的I型金属梁，这样它们就能“揭示”内部的结构。而且，为了不影响大厦的视觉效果，窗户的形状只能处于三种状态：打开、关闭或打开一半。对于防护太阳光来说，这可能不是最佳的安排（Wolfe 1982）。

图14-5：Seagram大厦

一流的现代建筑师Louis Sullivan（摩天大楼的创造者之一，Wright的指导者，以及其他一些成就）写下了著名的一段话：

> 不论是翱翔的雄鹰，或盛开的苹果花、辛苦工作的马匹、无忧无虑的天鹅、枝叶繁茂的橡树、蜿蜒的溪流、飘浮的云朵、飞奔的太阳，形式总是服从于功能，这就是自然的法

则。功能不变，形式就不变。花岗岩、永恒沉寂的山峦，常年不变；一闪而过的生命，生长成形，转瞬即逝。

这是所有事物的法则，无论有机还是无机、实体的还是形而上的、关于人的还是关于超人的、有关大脑和心灵的所有现象，即生命可以通过它的表达方式来认识，形式总是服从于功能。这就是自然法则。(Sullivan 1896)

Mies van der Rohe接下来似乎在脑门上贴上了“形式服从于功能”的标签。也许可能是这样的格言更有煽动性，而不是描述真实发生的情况。Paul Rand可能是一流的美国图形设计师。他曾负责设计IBM、ABC和最初UPS的标志，他与Steve Jobs在NeXT计算机公司合作过，他写了一些有影响的书籍，介绍他的设计理论。他指出：

分离形式和功能，分离概念和执行，不太可能产生具有美学价值的东西，而这些美学价值是一直以来反复不断呈现的。类似地，如果某系统根本不考虑美学，将艺术家与他的产品分开来，将艺术家和他的产品割裂开来，将个人的工作分开来，或由集体创作，或者由一个创造过程得到“百果馅”，那么从长期来看，不仅产品会贬值，它的创作者也会贬值。(Rand 1985)

责难现代建筑是很容易的，特别是Mies van der Rohe和Le Corbusier的作品在世界范围内被拙劣地模仿，造成了许多丑陋的民居区、犯罪滋生的工人宿舍和没有灵魂的商业中心。更有趣的是深入研究大师们自己的评论。我们看到，Wright、Le Corbusier和Mies van der Rohe都受到了批评，因为他们不让步。他们因坚定不移而受批评，因不做折中而受批评。他们尖锐的观点给我们留下了美丽的建筑，但这些建筑没有给我们带来物质上的享受。

坚定不移不一定是缺点。在《Doctor Dobb's Journal》杂志1996年4月的访谈中，Donald Knuth被问到他对Edsger Dijkstra的看法。“他了不起的地方在于他不做折中。想到用C++编程就会让他身体不舒服。”他的不折中达到了这样一种程度，以至于多年未碰计算机，写了“关于谦恭程序员（Humble Programmer）的真正极好的文章来讨论这一点”。Dijkstra是最有影响力的计算机科学家之一，他的著作今天仍然充满了有价值的建议，当我们被编程领域最新的时尚或银弹吸引时，阅读这些著作是很有益的。对于那些必须在真实世界计算机上编程的程序员来说，他不让步的姿态使他的著作显得尤为珍贵。

这可能是理解Smalltalk的角色的关键，像在它之前的Algol一样，Smalltalk在我们的职业生涯中扮演了重要角色。有一些建筑师开辟了新的道路，为将来的几代人创建了纪念碑，可能这些建筑的本质比实际的房屋或办公楼更是一种宣言。没人会争论Fallingwater打动拜访者的力量以及它为年轻架构师带来的灵感，即使我们可能会争论它是否是合适的住房，或者我们可能争论“形式服从于功能”，并让装饰到处都是，直到后门。类似地，有一些软件系统的成功更多地在于它们对代码的影响，而不是用它们来编写代码。

如果我们想编写代码，为了业务或为了开心，我们就需要从美丽的架构中获取灵感，但我们也许不能直接利用它来工作。我们的工作必须以美丽的架构为榜样，但必须是实际可行的。最纯粹最美丽的智力体系是纯数学，我们可以从中学到很多，但我们不能用它来编程。我们必须开发出能工作的系统，这就是事情开始变得复杂的原因。有时候人们很容易迷失在设计方法学中，忘记了我们的目标是不同的。Christopher Alexander，这位作为设计模式之父的架构师这样说到：

> 整个学术领域都是围绕着“设计方法”的思想发展起来的——我也被看作是这些所谓的设计方法的拥护者之一。我对此非常报歉，我想公开声明，我完全拒绝将设计方法作为一个课题来研究，因为我认为将设计的研究和设计的实践分离开来是非常荒唐的。实际上，学习设计方法而不实践设计的人几乎都是失败的设计者，他们没有活力，他们丧失了创造事物的冲动，或者说从来就没有过这种冲动。(Alexander 1971)

作为程序员，我们必须创造能工作的系统，而不只是看起来漂亮的东西。这两者不一定不可兼得。图14-6展示了Robert Maillart设计的Salginatobel大桥，它峻工于1930年。Maillart是瑞士人，学的是工程，但他的作品，特别是他的桥，是建筑之美的典范。最重要的是，这些作品不只是漂亮。Maillart在竞标中赢得合同再建造大桥，以Salginatobel大桥为例，他击败了其他19份竞争设计，赢得了合同。建造大桥和公路当时只花了70万瑞士法郎，在今天接近400万。这座桥不是一个小工程。它的跨度达到90米，拱顶距Salgina河谷底达80米（Billington 2000）。结构的修长和轻巧是它低成本的原因。这座桥的低成本要归因于它的优雅。

Maillart的主要优点在于他的实用主义观点。他通过一种创造性的直觉得到了他的设计。他避开装饰和点缀，而且也没有模仿传统的建筑风格。他设计的结构不能由当时的计算工具分析（而且缺少计算机），所以不能证明设计是可靠的。他利用简化的图形分析来评估设计的可行性。如果Maillart必须等待其设计的严格证明，他的所有作品都不会建造（他于1940年去世）。Maillart“发现创新，尤其是桥梁设计的创新，不是来自于实验室的工作和数学理论，而是来自于设计室和建筑工地。数据在工程中扮演了十分重要的角色。但桥梁设计中的创新是可视的几何学想象的产物，而不是来自于抽象的数值分析，也不是从一般理论推导而来”(Billington 1997)。

编程像建筑一样，是一种实践。我们最好避免教条主义，而是将注意力放在能工作的东西上：

> 建筑是全能和无能的危险混合。表面上涉及的是“塑造”这个世界，因为他们要成为有影响力的建筑师，就要影响其他人——客户、个人或机构。因此，不连贯，或者更准确地说，随意性，是所有建筑师生涯的底层结构：他们面对的是随意的指令序列，以及他们没有确定的参数，在他们几乎不了解的国家，面对他们几乎没有意识到的问题，人们希望他们处理那些即使聪明得多的人都很难处理的问题。建筑在本质上就是混乱的冒险。(Koolhas 等 1998)

建筑是混乱的冒险，因为只有美丽的架构是不够的。不仅要美丽，而且要有用，这是建筑和编程等活动的法则。

图14-6：Salginatobel大桥

参考资料

Abrahams, David, and Aleskey Gurtovoy. 2005, *C++Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Boston, MA: Addison-Wesley.

Alexander, Christopher. 1971. *Notes on the Synthesis of Form*, Preface to the paperback edition. Cambridge, MA: Harvard University Press.

Alpert, Sherman R., Kyle Brown, and Bobby Woolf. 1998. *The Design Patterns Smalltalk Companion*. Boston, MA: Addison-Wesley.

Billington, David P. 2000. "The Revolutionary Bridges of Robert Maillart." *Scientific American*. July, pp. 85-91.

Billington, David P. 1997. *Robert Maillart: Builder, Designer, and Artist*. New York, NY: Cambridge University Press.

Black, Andrew P., et al. 2007. *Squeak By Example*. Square Bracket Publishing.

Bloch, Joshua. 2008. *Effective Java*, Second Edition. Boston, MA: Addison-Wesley.

Brand, Stewart. 1997. *How Buildings Learn: What Happens After They're Built*, Revised

Edition. London, UK: Phoenix Illustrated.

Calvino, Italo. 1986. "Why Read the Classics?" *The Uses of Literature*. Translated by Patrick Creagh. New York, NY: Harcourt Brace Jovanovich.

Conroy, Thomas J., and Eduardo Pelegri-Llopart. 1983. "An Assessment of Method-Lookup Caches for SmallTalk-80 Implementations." *Smalltalk-80: Bits of History, Words of Advice*. Ed. Glenn Krasner. Boston, MA: Addison-Wesley.

Curtis, William J. R. 1996. *Modern Architecture Since* 1900, Third Edition. New York, NY: Phaidon Press.

De Botton, Alain. 2006. *The Architecture of Happiness*. London, UK: Hamish Hamilton.

Gamma, Erich, et al. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

Goldberg, Adele, and David Robson.1989. *Smalltalk-80: The Language*. Boston, MA: Addison-Wesley.

Forman, Ira R., and Scott H. Danforth. 1999. *Putting Metaclasses to Work: A New Dimension in Object Oriented Programming*. Boston, MA: Addison-Wesley.

Jencks, Charles. 2006. *The New Paradigm in Architecture: The Language of Post-Modernism*. New Haven, CT: Yale University Press.

Josuttis, Nicolai M. 1999. *The C++ Library: A Tutorial and Reference*. Boston, MA: Addison-Wesley.

Kaufmann, Edgar Jr. 1986. *Fallingwater: A Frank Lloyd Wright Country House*. New York, NY: Abbeville Press.

Koolhas, Rem, et al. 1998. *S, M, L, XL*, Second Edition. New York, NY: The Monacelli Press.

McCarthy, John. 1960. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." *Communications of the ACM*, April 1960.

McConnell, Steve. 2004. Code Complete, Second Edition. Redmond, WA: Microsoft Press.

Meyers, Scott. 2005. *Effective C++: 55 Ways to Improve your Programs and Designs*. Boston, MA: Addison-Wesley.

Norman, Donald. 1988. *The Psychology of Everyday Things*. New York, NY: Basic Books.

Nuttgens, Patrick. 1997. *The Story of Architecture*, Second Edition. New York, NY: Phaidon Press.

Petzold, Charles. 1999. *Programming Windows*, Fifth Edition. Redmond, WA: Microsoft Press.

Rand, Paul. 1985. *A Designer's Art*. New Haven, CT: Yale University Press.

Simon, Herbert. 1996. *The Sciences of the Artificial*. Cambridge, MA: MIT Press.

Stroustrup, Bjarne. 1985. *The C++ Programming Language*. Boston, MA: Addison-Wesley.

Stroustrup, Bjarne. 1994. *The Design and Evolution of C++*. Boston, MA: Addison-Wesley.

Sullivan, Louis H. 1896. "The tall office building artistically considered." *Lippincott's Magazine*, March 1896.

Thomas, David, et al. 2005. *Programming Ruby: The Pragmatic Programmers Guide*, Second Edition. Raleigh, NC, and Dallas, TX: The Pragmatic Bookshelf.

Vandervoorde, David, and Nicolai M. Josuttis. 2002. *C++ Templates: The Complete Guide*. Boston, MA: Addison-Wesley.

Wolfe, Tom. 1982. *From Bauhaus to Our House*. London, UK: Jonathan Cape.

跋

漂亮地构建

William J. Mitchell

人们常常对软件系统和建筑工作进行宽松的类比。但与初看上去相比，这两种系统间结构上的类似程度实际上更大，可以更严格地具体化。

构成软件系统的代码由一维的符号字符串构成，按照精确的句法规则将一些定义良好的词汇放在一起，目标是在合适的机器上执行时能得到有用的结果。

建筑工作显然不是一维的，但在其他方面则与代码非常相似。它们是离散的物理组件的三维组装，按照严格的句法规格将定义良好的组件词汇放在一起，目标是得到有用的结果。（建筑师在实践中确实比程序员在词汇和句法方面拥有更大的活动余地。）

在两种情况下，我们都可以编写正式的语法来建立规则。总的来说，正式的语法告诉你如何把东西放在一起。更准确地说，根据语言学和计算机科学中的标准定义，正式的语法包括：一个有限的非终结符集合N，一个有限的终结符集合T，一个有限的替代规则集合R，和一个初始符号S。替代规则的左边有一组符号，中间是一个箭头，右边是另一组符号。它规定你可以用右边的那组符号代替左边的那组符号。完整、正确的结构化组件是通过递归地在初始符号S上应用替代规则R而得到的。由该语法确定的语言是所有可以通过这种方式导出的终结符构成的组件的集合。

正式的语法通常应用于单词词汇，告诉你如何将这些单词放在一起构成完整正确的句子，但并不限于此。语法可以应用于许多不同类型的东西，告诉你如何将它们放在一起，得到有用的组件。

在针对编程语言的语法中，放在一起的东西是可识别的符号，替代规则的左边和右边是一维的符号字符串，这些规则推导出这种语言的完整而正确的表达式。在两维的图形语言中，放在一起的东西是两维的形状，替代规则的左边和右边是两维的形状组件，这些规则源于完整而正确的图形设计。在三维的建筑语法中，要放在一起的非终结符是一些建筑结构，要放在一起的终结符是实际的建筑构件，替代规则推导出这些构件完整而正确的组合——换言之，即用该语法确定的语言所表达的设计。

例如，在几十年前，George Stiny和我出版了一本这种形式的正式语法，内容是关于伟大的意大利文艺复兴建筑师Andrea Palladio（注）的著名的漂亮别墅。它推导出了所有已知的Palladio设计的别墅，以及大量确认的伪Palladian别墅。（或者可以说，它推导出了所有Palladio可能设计的别墅，如果他活得更长、有更多客户的话。）而且，它对Palladio的别墅建筑背后的原则提供了让人信服的解释。从那时起，人们针对其他设计工作编写了无数建筑语法。

建筑语法最重要的功能之一就是记录下原则或模块性以及层次结构，它们总结了某些特定建筑风格的特征。例如，在精确定义和广泛使用的经典建筑语言中，圆柱有底座、柱身和顶部。顶部又进一步分解为一个组件层次结构（多立克式、伊奥尼亚式和科林斯式各有不同），等等。沿着子构件的层次结构向上，通常隔开一定距离的圆柱形成柱廊。圆柱、柱上楣构和人字墙形成了门廊。最后，所有的组件和子构建漂亮地结合在一起，构成了完整的、符合语法的经典组合。这些组合可以像句子一样被解析为一些有名称的部分。

从几何学或CAD系统的角度来看，建筑的组件和子构件是一些具体的形状，它们可以被变形和组装，得到更大的空间构造。从供应链和建造的角度来看，它们是一些物料元素，被制造（通常通过许多步骤构成的一个过程，每个步骤都提供附加值）、购买、运送至要求的地点，最后在一个构件中组装到位。从建筑维护的角度来看，它们是可以替换的部分。从功能的角度来看，它们是一些模块，在建筑中执行着可以识别的任务，可以与其他模块一起形成一个子系统，执行更高层次的任务。当模块以这种方式组合时，它们不仅有悦目的空间关系，而且也通过这些空间关系的优点，通过它们的接口传递了一些东西（如结构负荷）。

类似地，编程语言提供了一些方式将代码分解到模块中，层次化地组装模块，得到更高层次的模块，最终完成整个软件系统。所有程序员都知道，好代码不是一团乱麻，它的

注：George Stiny and William J. Mitchell, "The Palladian Grammar," Environment and Planning B, vol. 5, no. 1, 5-18 (1978).

模块和层次有着清晰的逻辑结构，这种结构是利用语言的抽象和有组织的构建来实现的。经典建筑的组织清晰性为此提供了一个极好的模型。

除了少数的例外情况，建筑工作也遵循其他一些内部秩序原则。例如，圆柱排成一列通常是等间距的。如果想编写代码生成柱廊的CAD模型，那么你不需要为每个圆柱指定位置。你可以利用迭代，按照增量步长来确定位置参数。换言之，你以更精确更优雅的方式表达了建筑的原则，这种方式为代码的阅读者提供了更多的深刻见解。

如果你想生成有规则圆柱栅格怎么办？你可以使用嵌套迭代。首先对一个圆柱进行迭代，生成间距相等的一列圆柱，然后再迭代这一列圆柱，次数由你决定，生成圆柱栅格。

如果你想让角上的圆柱与中间的圆柱不一样，怎么办？（根据建筑转角处存在的不同结构情况或其他情况，建筑师常常这样做。）你可以使用条件：如果是转角，那么换成另一种圆柱设计。如果想改变中间圆柱之间的间距，以突出中轴线的重要性，区分外部的圆柱和内部的圆柱，那么你只要引入更多的条件。

模块性、层次结构和规则地重复并不是建筑师经常使用的全部秩序原则。如果你仔细分析建筑组合，就会经常发现在维度和比例、对称性（和有意破坏的对称性）、类似分形那样的嵌套的自相似性，以及参数化变化的花纹等方面的规律。

但是有时候，似乎缺乏内部的规律。建筑师可能出于某种原因，随机安排圆柱。为什么？似乎最准确地描述这种配置的方法就是单独指定每个圆柱的位置。没有更简短、更经济的描述方式。

就像这些简单的例子所说明的，好的建筑师不是通过随意的、蛮力的方式来构造他们的设计，他们肯定会避免笨拙的做法。根据现场环境、气候、要容纳的活动、材料和组件的供应链、建造过程和预算，他们会设计出相当不同的复杂建筑。但他们试图带着一种概念上的优雅来完成设计——遵守经济性的原则，并严格运用他们自己的奥卡姆剃刀原理。在漂亮的建筑作品所表现出来的不同和复杂性背后，你通常会发现一些关于功能组织和规范秩序的简单而优雅的原则。发现这些原则需要思考，而思考正是建筑的快乐和经验的关键部分。

如果你能弄明白这些原则，就可以通过某种标准编程语言中同样优雅的几行代码构造出这些作品的模型，或者（在更现代的编程环境中）使用一些形状重写规则。你甚至可以一般化，编写代码来生成设计，遵循同样的一些规则，对不同的情况和要求做出适当的反应。但如果你弄不明白这些原则，那么肯定要编写更长的、不包含那么多深刻见解的代码。

建筑明显的复杂性来自于建筑师所面对的需求的复杂性，它是通过蛮力的、逐点的描述的长度来衡量的。建筑师得到设计所遵循的原则通常可以用一段简短得多的代码来表示，执行这段代码就能生成所有这些复杂性。根据相当不错的估计，简短描述与冗长描述之比越小，建筑就越美丽。

建筑师会仰慕那些应用了简单优雅原则来实现许多复杂性的架构之美。类似地，软件架构师和程序员也会仰慕那些清晰而准确执行了许多复杂任务的代码之美（不牺牲可读性和可维护性）。科学家会仰慕那些描述了各种不同现象的简单法则之美和它们的解释能力。