

顶级业界专家揭密软件设计之美

架构之美

精选版



Diomidis Spinellis 等著

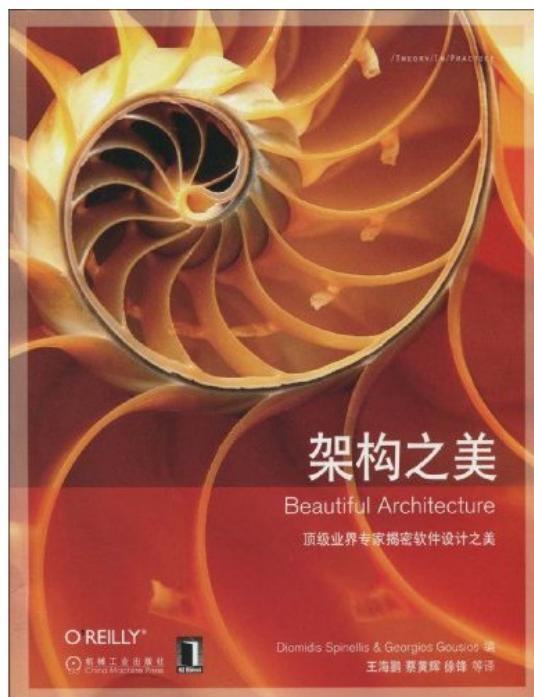
王海鹏 等译

InfoQ企业软件开发丛书

免费在线版本

(非印刷免费在线版)

[登录China-Pub网站购买此书完整版](#)



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品

InfoQ中文站
www.infoq.com/cn

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

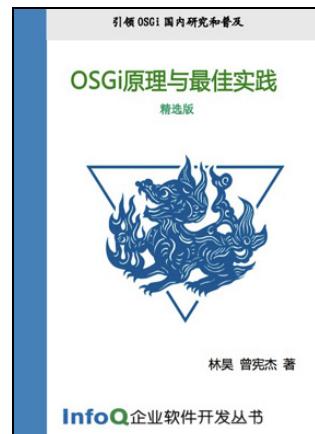
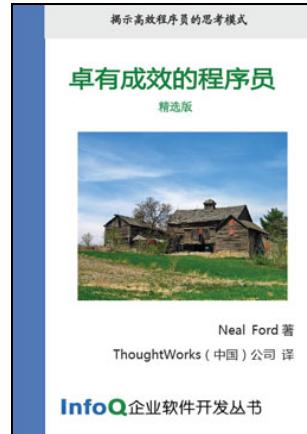
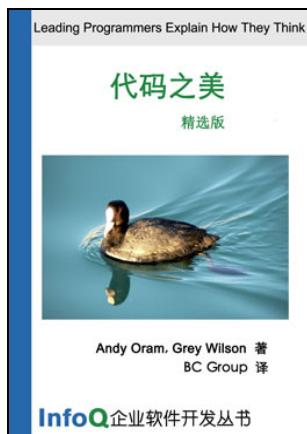
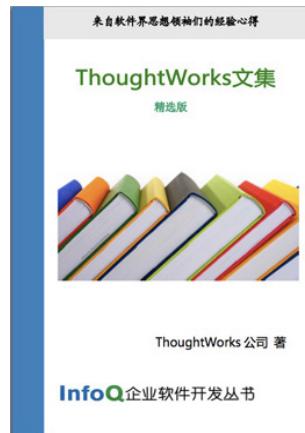
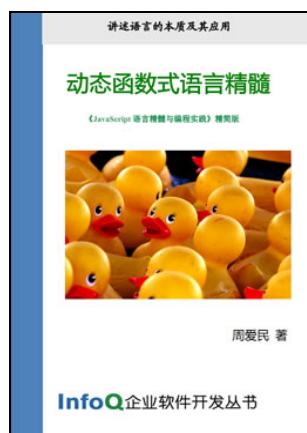
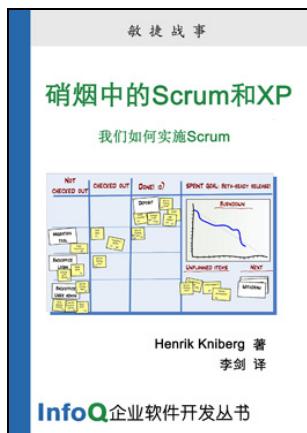
本书主页为

<http://infoq.com/cn/minibooks/beautiful-architecture>

注：封面图片选自 <http://www.flickr.com/photos/ladymolly/444785397/>

InfoQ企业软件开发丛书

欢迎免费下载



商务合作: sales@cn.infoq.com

读者反馈/内容提供: editors@cn.infoq.com

目录

推荐序	i
译者序	v
作译者简介.....	vii
第 1 章 架构概述.....	1
1.1 简介	1
1.2 创建软件架构.....	7
1.3 架构结构.....	11
1.4 好的架构.....	15
1.5 美丽的架构.....	16
致谢.....	18
参考文献.....	19
第 2 章 两个系统的故事：现代软件神话.....	21
2.1 混乱大都市.....	22
2.2 设计之城.....	28
2.3 说明什么问题.....	35
2.4 轮到你了.....	36
参考文献.....	36
第 3 章 伸缩性架构设计.....	37
3.1 简介	37
3.2 背景.....	38
3.3 架构.....	42
3.4 关于架构的思考.....	47
第 4 章 数据增长：Facebook 平台的架构	52
4.1 简介.....	52
4.2 创建一个社会关系 Web 服务.....	57
4.3 创建社区关系数据查询服务.....	64
4.4 创建一个社会关系 Web 门户：FBML.....	72
4.5 系统的支持功能.....	85
4.6 总结.....	90

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感兴趣，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亚度过，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，也是澳大利亚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，休闲活动占据了她的大多数时间，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，他曾是朗讯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。

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，约束开发者的工作。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法（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够得到一个有用的、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。

接下来，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 $2n$ （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." *OOPSLA '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。

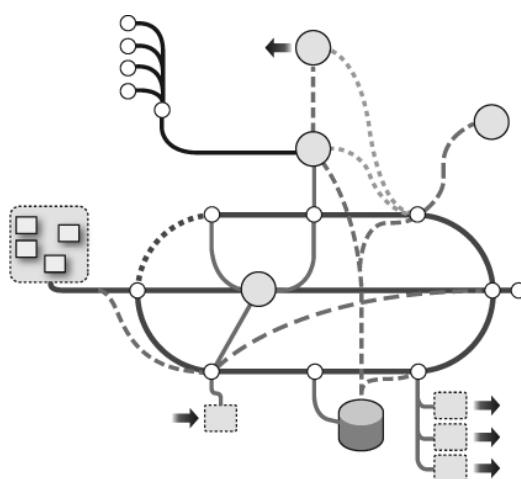


图2-1：“混乱大都市”的“架构”

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。地铁变成了云霄飞车，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，但我们都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

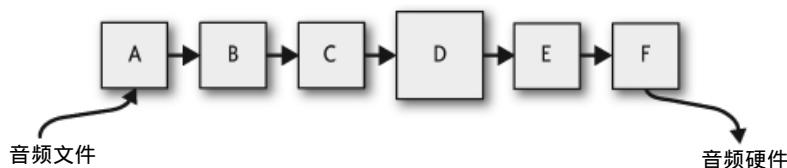


图2-3：“设计之城”的音频管道

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。”

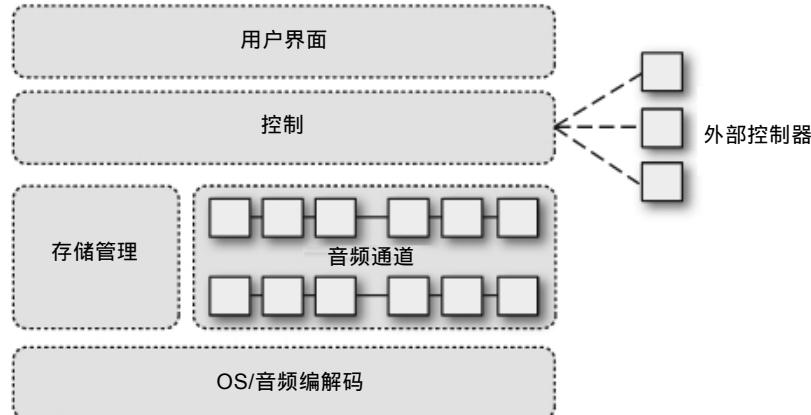


图2-4：“设计之城”的最终架构

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.

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。

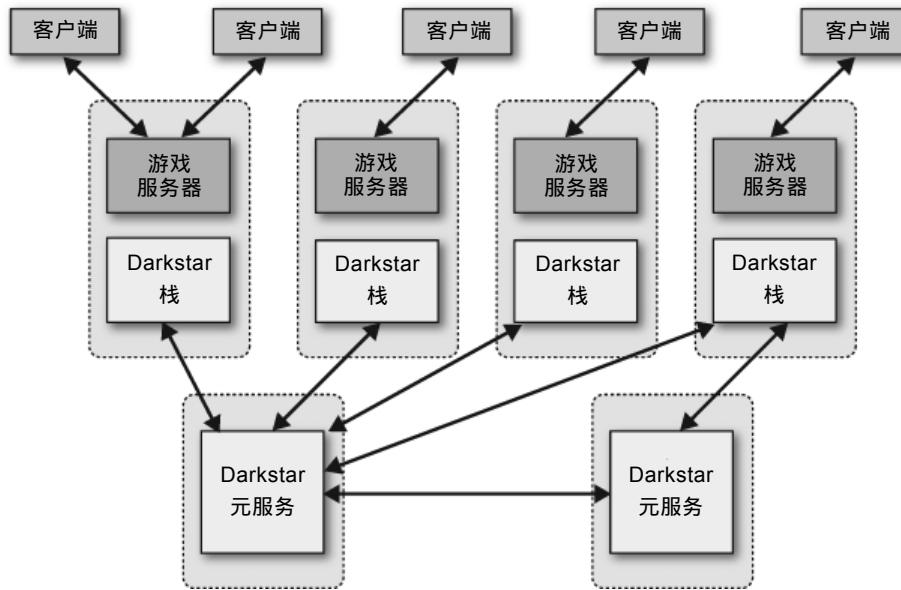


图3-1 : Darkstar项目的高层架构

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也不了解。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个架的主要不同之处，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

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)

4.1 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。

4.1.1 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4-1中这样的核心映射关系。

例4-1：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4-2中的简单“查找”。

例4-2：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。

4.1.2 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 4-3 所示。

例 4-3：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。

4.1.3 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 4-1 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。

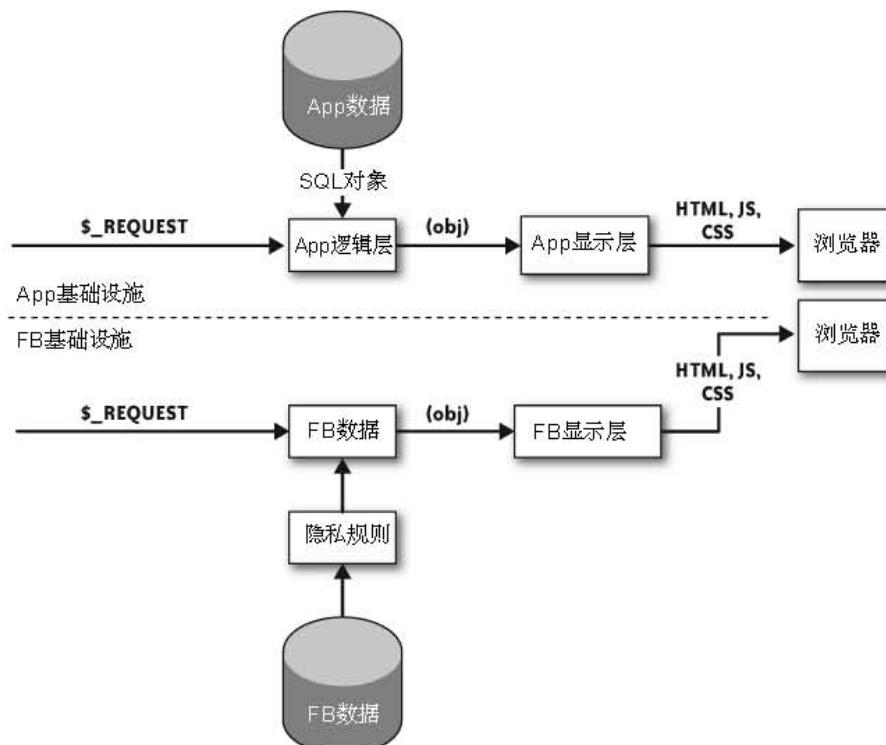


图4.1：分离的Facebook和n层应用栈

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。

4.2 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4-2）。

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4-4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4.4：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);
```

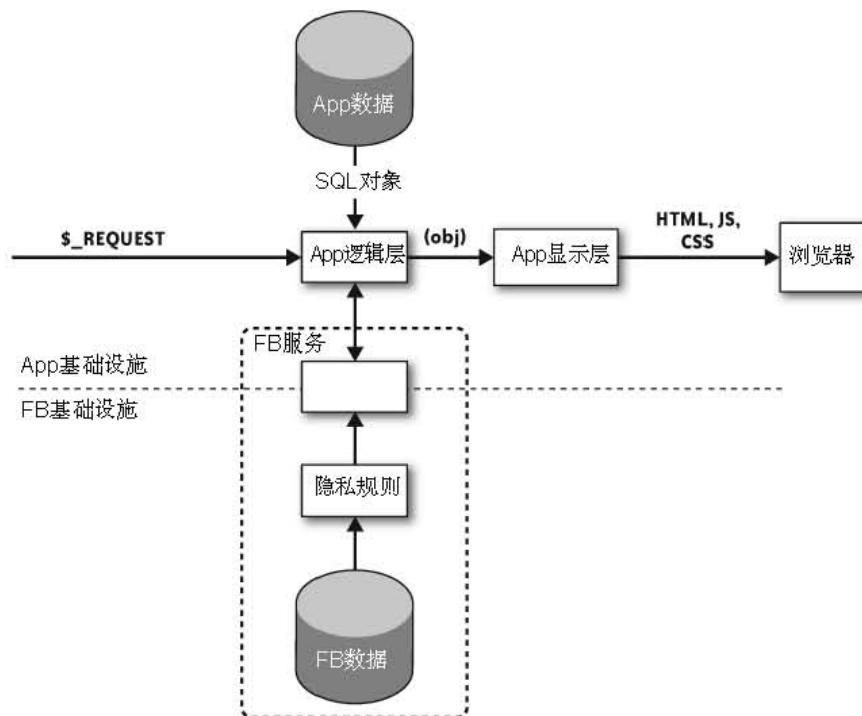


图 4-2：应用栈通过Web服务使用Facebook数据

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4-5所示。

例 4-5：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。

4.2.1 数据：创建一个XML Web服务

为了能够在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4-6展示了这些例子。

例4-6：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4.2.2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4.2.2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4-7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4-7：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 4-8 展示了针对 PHP 的部分生成结果。

例 4-8: 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 4-9 的样子。

例 4-9: 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 4-9 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 4-10 展示了整过 RPC 过程实际输出的 XML。

例 4.10: 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。

4.2.2 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4-1），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4-2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4-11所示。

例4-11：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4-3）：

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 4-3 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。

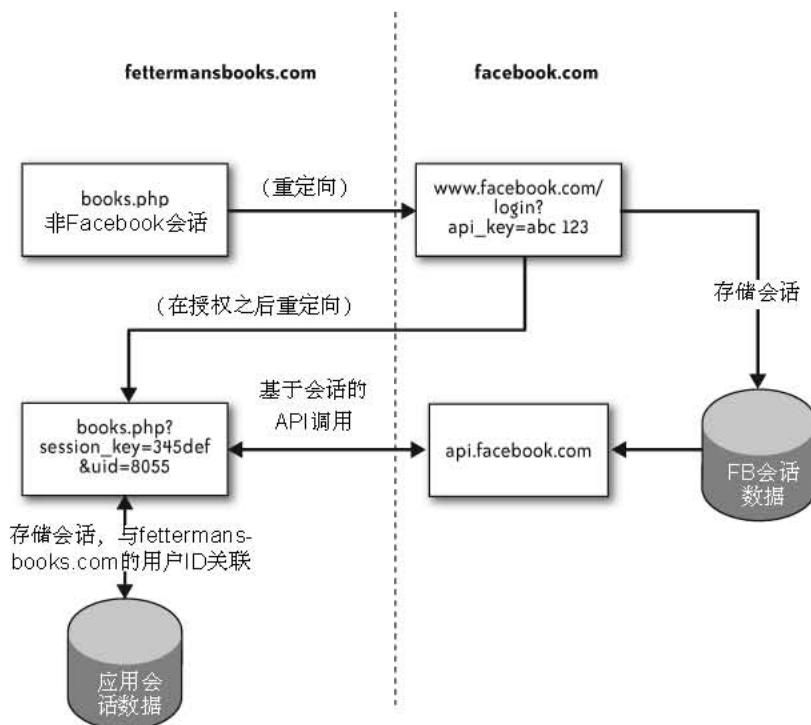


图 4.3：对 Facebook 平台 API 的认证访问

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。

4.3 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4.3.2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。

4.3.1 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.runAPI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4-12中的代码。

例4-12：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 4-13 展示了不能利用批量机制的一种常见情况。

例 4-13：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。

4.3.2 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 4-14 中的过程。

例 4-14：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 4-15 中写出这样的FQL语法并进行一次调用。

例 4-15：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 4.2.1 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4-16）。

例4.16：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4-2）。

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4-15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4-17）。

例4-17: 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（在下面的例子中，是`user_get_info`），虽然不一定是这样——这只是一个方便的接口。例4-18展示了用户表中典型的字符串字段的例子。

例4.18：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4-19）。

例4.19：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 4.2.1 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 4-20 所示。

例 4-20：一个简单的 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。`FQLStatement`的`evaluate()`函数将返回我们请求的对象。例 4-21 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 4-21：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4-4所示的状况。

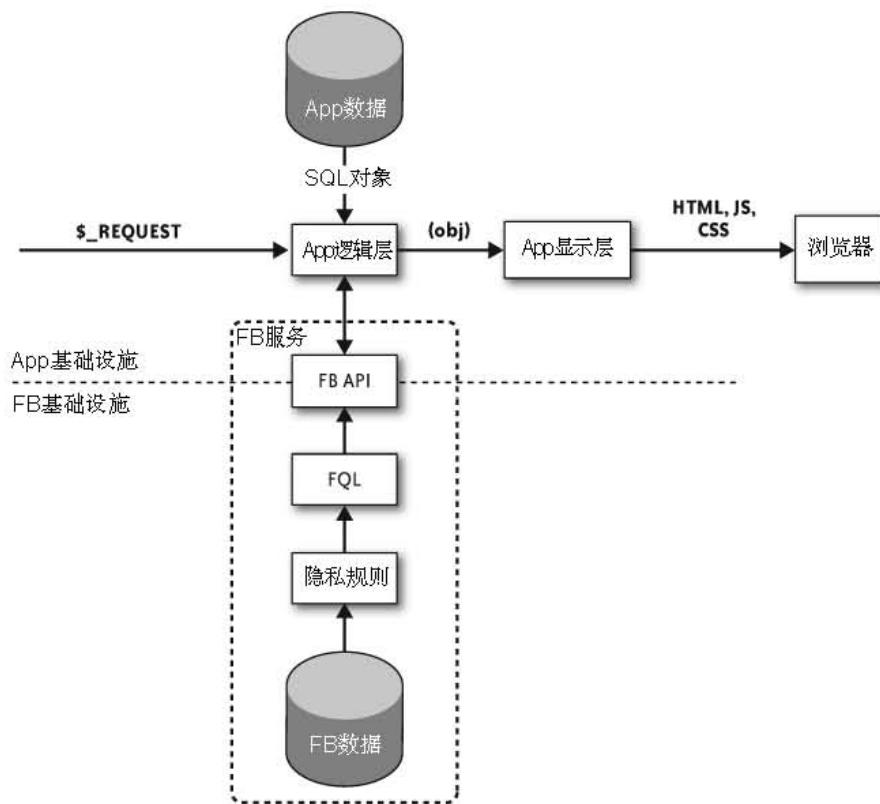


图4-4：通过Web和查询服务来使用Facebook数据的应用栈

4.4 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，要获得引入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。

4.2.1节和4.3.2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4.1.2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4.2.1节和第4.3.2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。

4.4.1 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://fettermansbooks.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上的页面主要内容区域进行显示。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。

4.4.2 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。

4.4.3 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4-5）。

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。

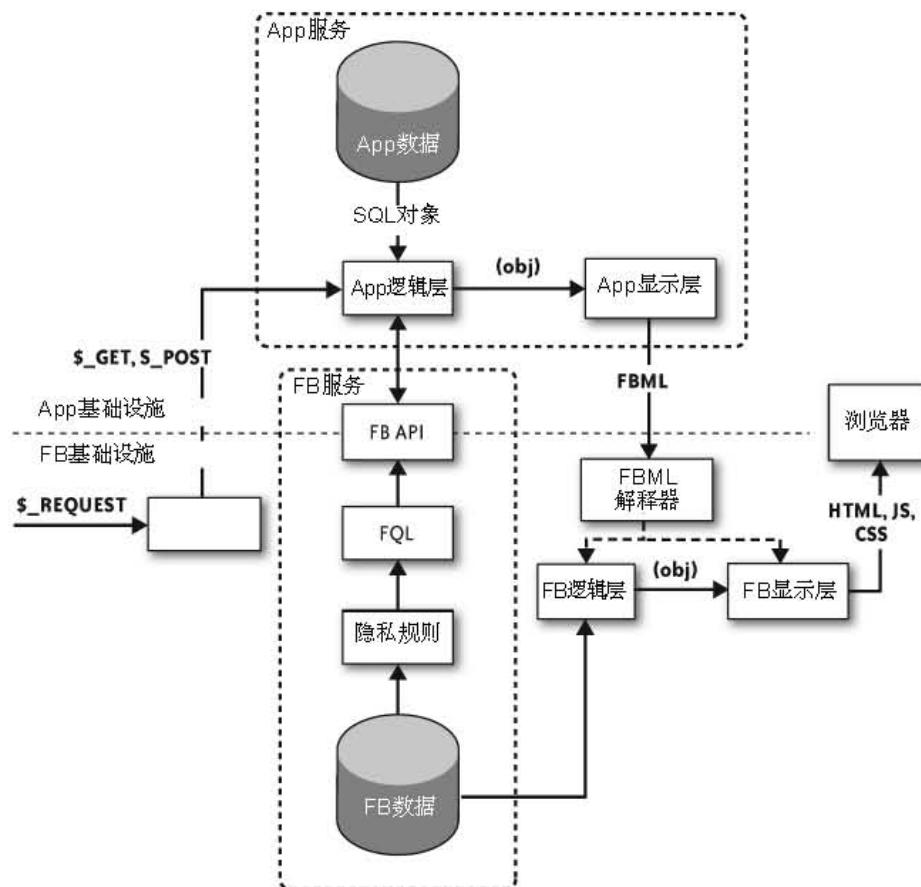


图4-5：应用即FBML服务

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4-5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字符串<h2>Hello, welcome to <i>Fetterman's books!</i></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=...'">
    
</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4-22展示了使用只面向设计的标签的FBML。

例4-22：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4-23所示。

例4-23：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。

4.4.4 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FBMLEflavor类来封装，如例4-24所示。

例 4-24：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 (! $category_allowed)
            throw new FBMLError('Forbidden tag category '.$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 4-25 是一个例子。

例 4-25：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 4-26 包含了标签的内部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4-26：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 off transform attribute key-value pairs according to
        // rules in FBML
        $safeAttrs = $this->_html_rewriter->node_get_safe_attrs($node);
        if (isset($safeAttrs['src'])) {
            // may here rewrite image source to one on a Facebook CDN
            $safeAttrs['src'] = $this->safe_image_url($safeAttrs['src']);
        }
        return $this->_html_rewriter->render_html_singleton_tag($node->
            get_tag_name(), $safeAttrs);
    }
}
```

在FBML中实现数据显示标签

例4-27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4-27：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4-28列出了它的细节。

例4-28：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4-29也是经过了大量编辑的。

例4-29：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。

4.5 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。

4.5.1 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。

4.5.2 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JavaScript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4-30所示。

例4-30：一个使用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.responseText = 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 4-31 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4-31：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.contextId=<!-- 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.onreadystatechange = function(a1234567_data) {
                a1234567_document.getElementById('ajax1').setTextValue(a1234567_data);
            };
            break;
    }

    <!-- NOTE 4 -->
    a1234567_ajax.post('http://www.fettermansbooks.com/testajax.php?t=' + a1234567_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\$rewriteAttrs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。

4.5.3 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4-6所示。

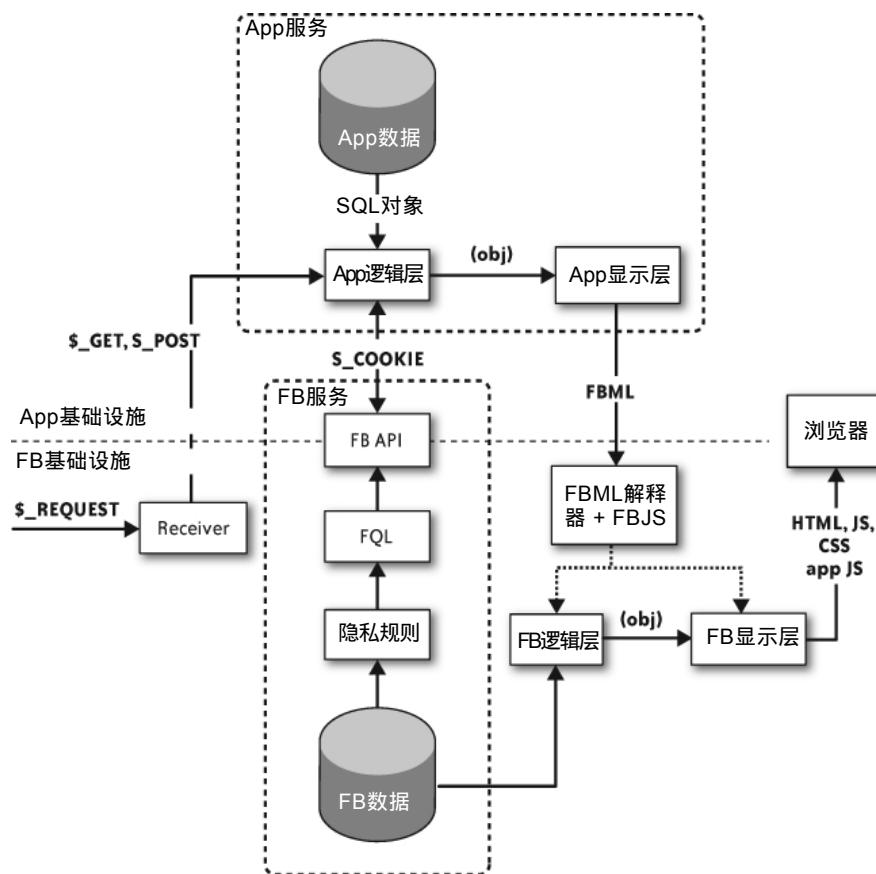


图4-6：Facebook平台服务

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。

4.6 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果。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。

架构师

www.infoq.com/cn/architect

每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

11月 ARCHITECT



Paul Hudak
谈Haskell
SOA宣言发布
如何进行平台型网站架构设计
HTML5来了
JS框架如何发展
每月技术综述

InfoQ中文站
www.infoq.com/cn

每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

10月 ARCHITECT



Stu谈云计算
一个100%的Ruby云方案
企业架构适用于小企业吗?
SOA与云计算有多大关联?
演进架构中的领域驱动设计
我来选架构

InfoQ中文站
www.infoq.com/cn

每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

9月 ARCHITECT



采访Lisp之父
John McCarthy
SQL SERVER的未来之路
软件开发7大浪费
TechLead三重人格
并发与不可变性

InfoQ中文站
www.infoq.com/cn

每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

8月 ARCHITECT



开发减速，
是为了赢利提速
“服务重用”是否被过度使用?
云计算虚拟研讨会
面向服务的经济学
架构师修炼之道

InfoQ中文站
www.infoq.com/cn

每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

7月 ARCHITECT



Dan Farino谈MySpace架构
SOAP基于Java消息服务
云汇聚天边
Java创新的未来
Google Wave加速HTML5发展?

InfoQ中文站
www.infoq.com/cn

每月8号出版