序

大规模企业级系统的设计与实现难度很大，要构建高效的 Java 企业级系统就更加困难了。这些难题对我来说已经司空见惯。在为企业级项目做咨询的时候，我常常会遇到开发者面临的这类现实问题。在 TheServerSide.com（企业级 Java 社区网站）上，我也经常看到对此类问题的探讨、它们引起的困扰以及相应的解决方案。当开发者面临着 J2EE 这个新领域时，许多问题随之而来，TheServerSide.com 正是针对着开发者的需求而发展壮大起来的。它是我们的交流场所，我们能够就使用的解决方案进行探讨，它也同时见证了企业级 Java 设计模式的发展历程。

与构建小型、单独使用的应用程序相比，企业级系统的开发非常不同。我们不得不去考虑那些以前确实可以忽略的问题。一旦我们要在多个用户中共享数据，就迈向了企业级系统之路。问题也随之而来：对这些数据进行并发访问的最佳策略是什么？要在多大程度上保证数据的一致性与正确性？我们能够从 2 个客户扩展到 5 个，甚至 1000 个客户吗？这些都是典型问题，我觉得普通开发者在面对这些问题的时候，并不能得到足够的帮助。当然，也许我们并不应该仅仅关注于问题的答案。我们还需要学习与问题相关的各种因素，以及遇到不同问题时，能够提供帮助的各种技术。有了 Ted Neward 的这本书，我们就拥有了这些知识，当遇到特定的问题，我们就能在解决方案中做出正确的权衡与取舍。

还没有哪本书能像《Effective Enterprise Java》这样专门针对这类问题。本书最重要的部分是，它将真正教会你两件事情。

你将理解企业级计算技术中的常规问题。

这些企业级问题并不新鲜。Ted 一直专注于这个领域，并且他理解问题的核心所在。出于这个原因，即使是非 Java 开发者也能从本书中获益。只要你还在开发企业级系统，那么在这里学到的知识就会一直陪伴着你。语言和 API 也许会发生变化，但是你将会理解：构建良好架构所要考虑的问题；有哪些通信方式可供选择；如何选择状态存储的位置；各式各样的

安全问题等等。

你将能够使用企业级 Java 平台技术来处理这些问题。

本书不仅为常规的企业级问题提供了真知灼见，而且还可以教会你采用当今的企业级 Java 技术来解决问题。当你把各种企业级 Java 技术组合在一起来考虑时，你的理解将更进一步。何时使用 Web 服务？消息通信能起什么作用？EJB 适合做什么？本书提供了对这些问题的解答。

对于这些常见问题，能够有现成的解答，那真是太棒了。本书的编排，采用一系列“高效项”的风格来呈献，正好符合这一点。让我们全神贯注，体会本书的乐趣吧！

Dion Almaer
TheServerSide.com 主编

前言

忘记过去的人，必将重蹈覆辙。

——George Santayana

对 Java 程序员来说，现在是大好时机。尽管 Java 作为可用的商业产品还不到十年，但在几乎所有主流计算平台上，它已经成为企业级系统的构建语言之一。那些要解决挑战性问题的公司和个人，正日益拥护 Java 语言及其平台。对于那些不使用 Java 的人来说，现在面临的问题不在于是否采用 Java 技术，而是准备何时开始采用。超越并且包含了 Java 语言本身的 Java 2 企业级平台（J2EE）规范，涵盖了大量规范和程序库。这使得在不牺牲性能，或者不用从头实现常用算法及数据结构的情况下，也可以编写出丰富、复杂的系统。Java 语言和虚拟机还在变得更加强大。针对 Java 开发者的工具和环境也在变得更加丰富和可靠。在许多应用领域，已经出现了大量商业程序库，这就降低了需要编写的代码数量。

大约十年以前，Scott Meyers 在他的《More Effective C++》[Meyers97]一书中写下了类似的开场白。稍加改动，那一段文字就非常适合作为本书的开场白。其实，我故意

模仿那一段写下了以上文字。从许多方面来看，我们会发现自己正处于 Java 的黄金时代，它所覆盖的领域是如此宽广，以至于 Java 似乎显得无所不能。正如 C++在 1996 年居主导地位一样，Java 在 2004 年占据了统治地位。

进行这种对比的主要目的是帮助我们认清形势。就在 Scott 写下那段话之后不到两年，C++就被异军突起的 Java 赶下了宝座。就像 C++开发者在后期开始能够“无所不能”一样，Java 这门新语言及环境一出现就备受欢迎，并且几乎是一夜之间，就取而代之。不过，Java 现在面临着微软公司.NET 平台的激烈竞争。一个自然而然的想法，是希望历史不要重演。要做到这一点，Java 开发者就务必要使自己开发出来的系统，能够达到、甚至超过原来对系统的期望。而要达到这个目的，他们就需要知道如何才能充分利用自己所使用的语言和平台。

有许多人，在众多场合都曾经说过，要“领悟”一项技术，找到它的最佳使用方式，大约需要五年时间。C++正是如此：1990 年的时候，我们只不过把 C++看成一门新的面向对象语言，因而对它的使用也就沿袭了从 Smalltalk 得来的一些比较好的实践经验。到了 1995 年，我们就已经超越了这个程度，开始探究 C++提供的独特功能（比如模板和 STL）。当然，HTTP 也是如此：1995 年浏览器初次亮相的时候，我们把 HTTP 当作 HTML 的传输方式。如今，我们则把 HTTP 看成一种通用传输机制，用它来传输各种数据。

所以，从时机来说，Java 是幸运的。它于 1995 年正式推出；但实际情况是，Java 真正为普通开发者所重视是在 1997 年，或者说，直到那时 Java 才好到足以战胜那些批评者和怀疑者。这十年以来，我们已经在大多数情况下采用 Java 编写应用程序，我们已经开始认识到许多实践经验和模式，来帮助（不过还不能确保）我们进行成功的开发。作为一个群体，我们才刚刚步入正轨。

很多情形与 C++时代很相似，我们又面临着十年前在 C++身上出现的类似问题（Scott 已经做出了回答），只不过现在换成了 Java。随着语言和环境的成熟，以及我们使用 Java 经验的增加，我们所需要的信息也在不断变化。1996 年，人们想知道什么是 Java。“不管它是什么，总之和互联网有关”是一种常见的解释。刚开始，开发者着眼于使用 applet，编写丰富的浏览器客户端程序，以及利用 Java 的跨平台移植能力。到了 1998 年，他们则希望知道如何使之工作：“我如何才能访问关系数据库？如何进行国际化？如何才能跨越物

理机器的边界？”如今，Java 程序员又提出了更高层次的问题：“应该怎样设计企业级系统，使之能适应未来的需求？如何才能提高代码的效率，而不用牺牲代码的正确性或者易用性？如何才能实现那些不被语言或平台直接支持的高级功能？”

就好像这些还不够似的，在整个企业级系统领域中，又诞生了一个新势力，它就是 Web 服务。尽管 Java 开发者还正在挑战那些更高层次的 Java 难题，他们又面临 Web 服务的学习周期：什么是 Web 服务？它如何工作？还有，也许最重要的是，它和 Java 有什么关系？

在本书中，我将回答这类问题。

关于项

在进行深入讨论之前，我觉得有必要指出（读者也许已经注意到了），与其它书籍相比，比如《Effective Java》[Bloch]和《Effective C++》[Mayers95]，本书中的项有很大不同。特别是本书中项的范围要比其他同类书中的要大得多：在这里我们没有把太多的注意力放在语言或 API 上，而是集中于设计层的元素和使用模式上。

这其实是有意为之的；我认为，对应于范围更大的企业级应用系统，这种编排方式能够在总体上与之相一致。当然，毫无疑问，所有《Effective Java》中的项同样可以应用于企业级应用的构建，但仅仅停留在这个层次上就不能把握住重点，企业系统有更多的内容需要考虑，而这些内容很多在语言或 API 范围之外。

例如，不成功的 EJB 应用通常并不是源于对某个方法或接口的误用，而是由于客户直接调用的实体 bean 设计不佳。设计上的问题远比实现上的问题严重，要解决它，则需要在更“高层”的角度来考量实体 bean 大体上要提供何种功能。（关于实体 bean 及其细节部分的讨论请参阅第 40 项。）

为此，本书中的项试图帮助开发者在系统和架构的层次，而不是在语言层次上认识效率。许多项对某些人来说比较熟悉，还有一些只是对读者早已熟知的内容加以提炼。这样很好：对

于某些读者来说是习以为常的内容，对于其他读者来说可能是全新的，反之亦然。

另外，我已经小心避免讨论常见问题。已经有很多讨论最佳实践和高效使用虚拟机和语言的书了；请查看参考书目列表。因此，本书中不会重复那些其它书中已经讨论过的内容，除非它们在企业领域中特别适用或有某种特殊应用。

出于这个目的，我将频繁引用那些在有关企业级 Java 的书籍中已经建立的模式；特别地，与其他书(见我的参考书目)相比，我会更加倾向于 Fowler 的《Patterns of Enterprise Application Architecture》(Addison-Wesley, 2002)，Hohpe 和 Woolf 的《Enterprise Application Integration 》(Addison-Wesley, 2004)，和 Alur, Crupi, 和 Malks 的《Core J2EE Patterns》第二版. (Addison-Wesley, 2003)

在那些用名字来引用模式的地方，我使用标准的 Gang-of-Four 模式引用格式，将页号放在引用名后面的圆括号中；但是，因为这些模式的来源不同，所以我也把作者的名字（比如在《Design Patterns》中用“GOF”来表示 Gang-of-Four）作为页引用的一部分。所以对 Fowler 的《Patterns of Enterprise Application Architecture》中的“数据传输对象”（Data Transfer Object）模式的引用就写成“Data Transfer Object（[Fowler，401]）”。

致谢

作者都愿意花很多时间来致谢；这是事出有因的。

首先，我想感谢我在工业界的同仁，我在 DevelopMentor 的许多同事。Kevin Jones，Brian Maso，Stu Halloway，Simon Horrell，Dan Weston，和 Bob Beauchemin，他们在我酝酿本书的 30 个月中为本书的主题提供了多方面的宝贵意见。Tim Ewald，Don Box，Fritz Onion，Keith Brown，Mike Woodring，Ingo Rammer，和 Peter Drayton，他们通过纠正我在讨论时所作的与 Java 无关的偏见和假设，使我深入到 Java 平台中。在 DevelopMentor 之外，我的同事，NoFluffJustStuff 研讨会的演讲人 Dion

Alamer，Bruce Tate，Mike Clark，Erik Hatcher，Glenn Vandenburg，Dave Thomas，Jason Hunter，James Duncan Davidson，和 Ron Bodkin，以及其他人，他们质疑我的结论，迫使我去证明我的断言，他们还提供了使本书更出色的建议和技巧。感谢 Jay Zimmerman 首先邀请我参与 NoFluffJustStuff。会上很多演讲者（太多了，在此不能一一列举）或多或少都扮演了类似的角色，让我时刻自省。

第二，我要向 Addison-Wesley 的工作人员致敬，本书所花的时间是预计的两倍，如果没有启动这个项目的编辑 Mike Hendrickson，和接手这个项目的编辑 Ann Sellers，这本书永远不会完成，他们总是非常礼貌地询问本书的进展，他们的耐心远胜于我，令人印象深刻。复审人员，不管是对我放在 blog 上的内容还是后来的手稿的审查中，都做了非常出色的工作，阅读所有材料并提供了很多修正、建议、完善和思路。感谢 Matt Anderson，Kevin Bentley，Dave Cooke，Mary Dageforde，Kevin Davis，Matthew P. Johnson，和 Bruce Scharlau 的帮助。

写作本书既是一次热情的释放，也是一次可怕的恐怖经历。我总是在寻找这样的一个项目来表达我对企业级 Java 开发的想法，很少有书像以前的 Effective 系列一样树立榜样。《Effective C++》中，Scott Meyers 为我（和其他上百万新生的 C++程序员）提供了开始使用 C++所需的帮助，而不仅仅是旁敲侧击。然后 Joshua Bloch 写了《Effective Java》，把漂亮的项格式引入了 Java 平台。Elliote Rusty Harold 进一步在《Effective XML》中沿袭了这种风格。如果有作者正在寻找一些可追随的榜样，以上三位就是最好的人选。很幸运，我从 Scott Meyers 处得到了帮助，他花了几乎和我写书一样的时间来复审并提出批评（以积极的口吻）。他的评论和见解帮助我把一些很好的建议融入到现在你面前的这本书中。Scott，我欠你一个很大的人情，即是为了过去一年中你对我的帮助，也因为 10 年前当我挣扎于理解 C++时你给我的指导。能和你一起工作是一种特权同时也是我的荣幸，谢谢你。

最后，当然，我必须感谢我的家人和朋友，那些时常出现的干扰和把我拖出去玩的可爱的人们，kicking 尖叫，把我带回到他们称之为现实世界的地方：聚会，度假，甚至连续一两个晚上玩 Nintendo64 或 Xbox。尽管我不会大声承认，但在过去的两年中，他们让我能以健康的面貌面对写这本书的压力。

报告 bug，提出建议，以及得到本书的最新资料

我已经试图让本书尽可能地准确、易读、易用，不过我知道依然有提高的空间。（提高的空间总是存在的）如果你找到了任何一个错误：技术上的、语法上的、排版上的、精神上的，不管是什么，请告诉我。我会尽力保证在后序版本中纠正这些错误，如果是你首先报告了某个错误，我会很高兴把你的名字加到本书的致谢中。同样，如果你对提高本书的后续版本质量有任何建议或想法，我也洗耳恭听。

我将继续收集那些高效的 Java 企业编程指南。如果你对新的指南有什么想法，并愿意与我分享，我将感到非常荣幸。你可以通过一些公开的 Java 编程邮件列表找到我，最好是 DISCUSS.DEVELOP.COM 上的 ADVANCED-JAVA 列表，或者你可以通过下面的地址与我联系：

```
Ted Neward
c/o Addison-Wesley Professional/Prentice Hall PTR
Pearson Technology Group
75 Arlington St., Suite 300
Boston, MA 02116
```

你也可以给我的邮箱 ted@neward.net 发邮件

从第一次印刷起，我就在本书的 blog 上（http://www.neward.net/ted/EEJ/index.jsp），维护着本书的修改列表（包括修订、澄清、注释和技术上的更新）。要是你愿意与其他读者分享，请把意见和勘误表贴上去。

差不多了，好戏上演了！

缩略语索引

ACID	原子性、一致性、隔离性、持久性	atomic, consistent, isolated, and durable
AWT	抽象窗口工具包	Abstract Windowing Toolkit
BLOB	二进制大型对象	Binary Large Object
BMP	Bean 管理的持久性	Bean-Managed Persistence
CMP	容器管理的持久性	Container-Managed Persistence
COM	构件对象模型	Component Object Model
CORBA	公共对象请求代理结构	Common Object Request Broker Architecture
CSS	层叠样式表	Cascading Style Sheets
DCOM	分布式 COM	Distributed COM
DHTML	动态 HTML	Dynamic HTML
DMZ	非军事区	demilitarized zone
DNS	域名系统	Domain Name System
DOM	文档对象模型	Document Object Model
DTCs	分布式事务控制器	distributed transaction controllers
EJB	企业级 Java Bean	Enterprise Java Bean
HTML	超文本标记语言	Hyper-Text Markup Language
HTTP	超文本传输协议	Hyper-Text Transmission Protocol
IDL	接口定义语言	Interface Description Language
IIOP	基于因特网的 ORB 间协议	Internet Inter-Orb Protocol
IPC	进程间通信	interprocess communication
J2EE	Java 2 企业版	Java 2 Enterprise Edition
J2SE	Java 2 标准版	Java 2 Standard Edition
JAAS	Java 认证与授权服务	Java Authentication and Authorization Service
JAXB	Java XML 绑定 API	Java API for XML Binding
JAXM	Java XML 消息 API	Java API for XML Messaging
JAXP	Java XML 解析 API	Java API for XML Parsing
JAX-RPC	Java XML 远程过程调用 API	Java API for XML RPC
JCA	Java 连接器 API	Java Connector API
JDBC	Java 数据库连接	Java DataBase Connectivity
JDK	Java 开发工具包	Java Development Kit
JDO	Java 数据对象	Java Data Objects
JESS	Java 专家系统 Shell	Java Expert System Shell
JIT	即时	just-in-time
JITA	即时激活	just-in-time activation
JMS	Java 消息服务	Java Message Service
JMX	Java 管理扩展	Java Management Extensions
JNDI	Java 命名和目录接口	Java Naming and Directory Interface
JNI	Java 本地接口	Java Native Interface
JNLP	Java 网络启动协议	Java Network Launch Protocol

JRE	Java 运行时环境	Java Runtime Environment
JSP	Java 服务端页面	Java Server Pages
JSR	Java 规范提案	Java Specification Request
JSSE	Java 安全套接字扩展	Java Secure Sockets Extension
JTA	Java 事务 API	Java Transaction API
JVM	Java 虚拟机	Java Virtual Machine
JVMDI	JVM 调试环境	Java Virtual Machine Debug Interface
JVMPI	JVM 性能测量接口	Java Virtual Machine Profiler Interface
JVMTI	JVM 工具接口	Java Virtual Machine Tools Interface
LAN	局域网	local area network
MDBs	消息驱动 Bean	Message-Driven Beans
MIB	消息信息块	Message Information Block
MVC	模型－视图－控制器	Model-View-Controller
NAT	网络地址转换	Network Address Translation
NFS	网络文件系统	Network File System
OODBMS	面向对象数据库管理系统	object-oriented database management system
ORB	对象请求代理	Object Request Broker
OSI	开放系统互联	Open System Interconnection
OWASP	开放 Web 应用安全项目	Open Web Application Security Project
POJOs	平凡 Java 对象	plain old Java objects
POP3	邮局协议 3	Post Office Protocol v 3
RDBMS	关系型数据库管理系统	relational database management system
RMI	远程方法调用	Remote Method Invocation
RMI/IIOP	采用 IIOP 的 RMI	RMI over IIOP
RMI/JRMP	采用 JRMP 的 RMI	RMI over Java Remote Method Protocol
RPC	远程过程调用	Remote Procedure Call
SAX	XML 流解析 API	Streaming API for XML
SMTP	简单邮件传输协议	Simple Mail Transport Protocol
SNMP	简单网络管理协议	Simple Network Management Protocol
SOAP	简单对象访问协议	Simple Object Access Protocol
SSL	安全套接字层	Secure Sockets Layer
STL	标准模版库	Standard Template Library
SWT	标准窗口部件工具包	Standard Widget Toolkit
TLS	传输层安全性	Transport Layer Security
TPC	两阶段提交	two-phase commit
TTL	生存周期时间值	time-to-live value
URI	统一资源标识符	Universal Resource Identifier
URL	统一资源定位符	Universal Resource Locator
URN	统一资源名称	Universal Resource Name
VM	虚拟机	virtual machine
W3C	万维网联盟	World Wide Web Consortium
WSDL	Web 服务定义语言	Web Services Definition Language
WS-I	Web 服务协同工作能力	Web Services-Interoperability

XML	扩展标记语言	Extensible Markup Language
XSD	XML 模型定义	XML Schema Definition
XSLT	“XSL 传输，一般也写成 XSL:T”	"XSL:Transformation, commonly also written as XSL:T"

第一章 绪论

在我来这儿之前，我对这个主题非常困惑。当我听完您的讲座之后，我仍然很困惑，但这是在更高层次上的困惑。

—— *Enrico Fermi*

这本书展示了如何去设计和实现更加*有效的*企业规模的 Java 软件系统：它们更有可能正确地运行，面对异常时更加健壮，更加有效率，更加高性能，更加可扩展，更不易被错误地使用。一句话，这些软件就是更*优良*。

然而，为了做到这一点，我需要对本书应该覆盖的内容与不应该覆盖的内容加以重要地区别。特别是，本书不是在对如何使用语言本身的那些有效的技巧进行改头换面的重新包装——那些内容属于 Joshua Bloch 的优秀著作 *Effective Java* 所讨论的领域，那本书应该被看作是所有 Java 程序员的必备读物。而本书则瞄向了更高的层次，即为企业系统编写 Java 软件的各种技巧，因此，本书具名为*《高效的企业 Java（Effective Enterprise Java）》*。

这样，至少对我们的四个目标来说，精确地定义什么是一个“企业 Java”系统显得非常重要。对许多开发者来说，在这里，有关关系型数据库、商业规则、事务功能以及可扩展性的讨论是处于支配地位的。任何运用了在 J2EE 规范中被定义的大多数规范的系统自然都是候选者。然而，我却倾向于从另一个稍显不同的角度来考虑问题的答案。

一个企业系统是具备下列性质的系统：

- *共享某些或全部在应用中被使用的资源*：这里普遍存在的例子就是所有的应用数据驻留的关系型数据库。共享这些资源会增加额外的隐含复杂性：数据被共享是因为它需要同时对多个用户可用。因此，系统必须支持安全地且快捷地并发用户访问。
- *规划成为内部使用*：这里的“内部”指的是“大量生产的卖给最终用户的软件的对立物”。当系统确实可以在公司与商业伙伴之间共享时，它可以用公司的特有知识、商业惯例和特殊需求来编写。
- *必须在现有的架构内运行*：在极特殊的情况下，公司才有可能已经拥有了一套适当的、系统必须能够与之进行互操作的硬件和软件。特别地，这意味着应用必须适应现有的数据库模式（而不是其它的方式）。一个企业系统必须能够适应它所存活的异构系统。

- *将由内部 IT 员工部署并提供支持*：对大多数公司来说，实际的“产品”生产都超出了开发者的职责范围。这是一件好事情——大多数开发者都会对因为他们开发的应用的故障，在凌晨时分被唤醒而感到不痛快。但是这同时也意味着系统的部署将要由他们之外的人手去完成，并且这还意味着负责数据中心的员工必须有某种途径在未经历编写代码环节的情况下，去监视、诊断和订正问题。
- *需要更强的健壮性，对于异常处理和可扩展性都是如此*：企业系统，特别是通过互联网可访问的系统（马上就能想到经典的 e-commerce 系统），象征着某个公司的一项巨额投资。系统宕机的每一分钟都意味着成千上万，也可能是成百万上千万美元收入的流失。每一个厌恶公司网站的用户（更糟的是，强制用户紧盯浏览器去等待登录请求的完成）都会导致公司的诚信度和潜在销售额的损失。
- *只能适度地失败*：在像词处理器这样的应用中，某个意外情形可以通过抛出一个“唉呀”对话框，保存用户正在进行的工作，并请求他或她重新启动程序来处理。一个企业系统却不能这么做——如果它崩溃了，成百万上千万美元就会因丧失了生产率、销售额、客户感知等等而付之东流。一个企业系统要为“五个九的可用性”而奋斗：系统每年全部的正常运行时间要大于 99.999%。这使得系统的停工期，不论是预先安排的还是突发的，只剩下每年 0.001%，或者粗略为少于 5 分钟。
- *必须合理地处理系统随时间推移而发生的演化*：企业系统具有长久的生命周期，Y2K（2000 年）问题证明了这一点。因此，一个系统必须能够适应在公司内随时间推移而发生的不可避免的变化：合并、销售额增长、政策调整、法人变更、征购等等。

很明显，这是一个相当大的领域。企业软件在尺寸和范围上都是横跨整个领域的，从个人的电子表格软件到数个 TB 的关系型数据库。随着无线设备在大型公司内的使用的迅猛增长，你甚至还会主张为 PalmOS 设备或移动电话编写代码也属于企业开发。然而，本书在很大程度上聚焦在了企业计算的传统领域内，即 PC 连接到一台或多台服务器上。

虽然上面的定义覆盖了一个相当大的范围。企业应用可以被内部或外部使用，可以运行完整的关键内容：某些可以在本质上纯粹是管理性的，例如人力资源部的休假报告系统，某些可以是公司的核心收入流，例如像 Amazon.com 这样的在线零售系统。商业伙伴可以使用企业系统来下订单、托运出货或者递交发票。咨询公司可以为顾客放置供其读取的技术内容。

所以，编写满足这些需求的软件可能会很艰难，并且很费时间。由此，J2EE 应运而生。

J2EE 的目标

按照老话的说法，知道我们在哪里（以及*为什么*我们在那里），将有助于我们去了解下一步该*如何*走。我想解释的是 J2EE 的“为什么”和“如何”，以确保对某些概念（例如就像 lookup，这对第 16 项（Item 16）来说很重要）的清晰认识。

纵观计算科学的历史，任何语言、工具或库的终极目标主要都是要提高抽象级别，使我们从那些会让我们无法专注于手头实际工作的细节中抽身而出。例如，思考一下经典的 OSI 七层网络协议栈。当你要“打开一个套接字（socket）”时，并不是要实际打开什么能够直接连接到其它机器的东西，这个动作实际上是在软件（以及硬件，只要你涉及物理层）的四个层次或五个层次之上的抽象，其中每一层都提供了确定数量的支持，以使这些资源运转起来。

在企业系统的早期发展阶段，分层的缺乏到了令人头痛的程度——所有数据访问都是直接通过由定长记录组成的文件而完成的，在那些记录上发生的任何事都属于你的业务，并且属于你独有的业务。之所以呈现出没有分层机制的局面，是因为在那些日子里我们运行的系统数量相对于 CPU 周期或内存空间来说并不算多。每件事物都必须尽其所能地使其受到关注。

随着硬件容量的增长和对复杂处理的需求的增长，我们发现必需而且很渴望使某些行为得到保障。所以一个新的软件层被置于了传统的平面文件集合之上，我们称之为事务处理系统。它管理对数据的并发访问，确保数据遵从我们通过自己编写的代码而对其施加的逻辑约束。随着时间的推移，这层软件通过引入了一个强大的查询语法而被更加形式化了，而且由此诞生了现代关系型数据库和 SQL。

此后，我们开始希望让终端用户使用存储在数据库中数据，而不是给数据处理员成沓的包含了输入数据的纸张。这绝不仅仅只是意味着大学生们丢掉了在暑期勤工俭学的一条出路，更重要的是客户/服务器架构由此而诞生。在客户机上执行的程序，要负责数据的获取和呈现，并将这些操作转换成工作语句以执行到数据库系统的操作。通常情况，这类程序都具有各种

不同的图形化用户接口，它们以为此目的而专门构建的某种高级语言编写，并需要为正在为公司开发的特定系统进行定制。

然而，随着这些客户/服务器系统的客户数量的增长，我们开始陷入了局限：由于有那些伴随客户动作的内部处理，到数据库的物理网络连接（以及相关的软件开销）就有了一个确定的上限，这样，就能够给可以同时使用系统的用户数扣上了一个任意大的帽子了。我们可以说 *n* 个客户端是系统用户数的上限，其中 *n* 是最大连接数，只要第 *n+1* 个客户想要登入系统，我们就需要为他或她创建新的数据库。

即使是在《财富》杂志的排行榜上最大的 50 家公司，在短期内也可以接受这一状况，因为一个企业系统的最大用户数通常不会超过四位数。不管在这么做时，花费了多大的代价在其中，通常还是有可能，尽管这并不是人们希望的，会将一个新的安装推给上千个内部客户端。然而，我们一开始采用 Web 作为企业系统的公共界面，情况就突然间从根本上发生了变化——Web 所涉及的东西都是在扩展公司“疆域”，事实上可以说，这意味着用户可以从世界上任意的地方访问公司。过去我们拥有上千个客户端的地方，现在使用 Web 就意味着拥有了上百万个客户端。

可能的并发客户端数量的几何级数般的跃升意味着旧有的“一个客户端，一个连接”的架构已经再也不可能了。因此急需新一代的软件架构，它能够使通过 Web 将系统带给最终用户的思想具有可以运作的机会。

计算机科学有一句经典格言叙述到：“没有任何问题是不能通过增加额外的附加层而得到解决的。”在这里，因为客户端程序通常并不是在 100%的时间内都在使用他们所持有的到服务器的连接的，所以被引入的间接层是位于客户端和服务器之间的软件层。（注意这个精细的术语定义，详情可参阅第 3 项（Item3）。）这个资源管理软件层，在为其起一个好名字而争论了数年之后，有了一个众所周知的名称：中间件（middleware）。

中间件和 J2EE

Bernstein（在[Gray/Reuter]中被引用）将术语中间件定义为一项分布式系统服务，它包含标准的编程接口和协议。他进一步声明到，中间件服务在操作系统和网络层之上，在工业相关的应用之下，提供了一个支持层。中间层，像在以前出现的人们熟知的事务处理监视器（Transaction Processing Monitor，缩写为 TP Monitor），是“为了要集成其它的系统构件和管理资源…它与许多不同的软件部件之间都有接口，它的主要目的是要以某种特殊的方式使这些软件部件协同工作，这种方式最终形成了著名的面向事务的处理（Transaction-oriented Processing）”[Gray/Reuter，240，特别强调]。

J2EE 很明显是事务处理监视器/中间件遗产的一个继承者。J2EE 规范自身就是将一打的其它 Java 规范糅合到一起组成的一个一致的、可定义的整体，它并没有描述多少新增的或附加的内容到这些规范中去，以此为利用所有这些规范来构建的系统提供一个稳定的和内部一致的基础。Servlets、JavaServerPages、JDBC、RMI（Remote Method Invocation，远程方法调用）、JMS（Java Message Service，Java 消息服务）、JTA（Java Transaction API，Java 事务 API）、JCA（Java Connector API，Java 连接器 API），当然还有 EJB——这些规范以及其它规范都被集合起来，形成了在 J2EE 规范的遮护之下的一个几近和谐的事物。

如果不是全部，那至少也是被 J2EE 统成一体的规范都是用来处理资源管理的。例如，JDBC 描述了如何与关系型数据库系统进行交互并进行操作；JMS 覆盖了集成面向消息的中间件（这里又使用了这个词）系统的内容；RMI 是有关远程过程调用的；JTA 是有关事务管理的；JCA 是有关“遗留（legacy）”系统、其它的通信系统以及面向记录的数据系统的；还有其它等等。

有关中间件的思想在许多方面都来源与对集成的渴望。非常频繁的情况是，一个企业发现它自身拥有大量的未互联的“烟囱”系统，这些系统都被用来完成某项特殊目的的任务。（烟囱系统（stovepipe system）这个名字来源于应用面很窄、专注于某一点的应用系统的思想：一个数据库被一个单一程序访问，当我们用图形法观察这个程序时，它看起来就像是一根烟囱。）这些系统通常都是在一个单独的科室或部门的控制下而被开发出来的，它们处理该特殊科室或部门的独一无二的要求。这里列出部分这样的系统，包括财会、存货管理、人力资

源、客户关系管理、订单登记等等。单独地看这些系统，通常每一个都是十分成功的——它们满足某个部门的需求，正是这些需求驱使该部门渴望首先构建这些系统。

遗憾的是，企业整体的需求并不止于各个部门自己的简单需求。一个次要的、中间的需求集贯穿整个企业，它们几乎是在“互联网革命”的冲击甫一出现之时，就得到了实现。当公司开始审视使用互联网作为一种触及客户的新方式时，一旁便是它们传统的实体零售店的渠道，它们所开发的企业系统随时间推移突然变得很不适合了。公司希望将每一件事情都推到线上：订单登记、订单跟踪、供应链管理等等。他们很快发现，为那些通过电话登记订单的销售商而建立的系统，在置于 HTML 表格之后时就不能运行了——当订单登记突然间成为了顾客的职责时，例如，比起由训练有素的公司职员来实现这一任务，这时需要更多的验证。新的渠道也在不断地被设计和探索，现在，公司希望通过互联网把它们的货物和服务不仅兜售给顾客，同时也兜售给供应商和商业伙伴（普适的“B2B”渠道）。移动设备正在迅速地吸引着人们的兴趣，这使得让商业企业通过无线 PDA、蜂窝电话和其它传统的 HTML 不能涉及的工具向客户销售成为了可能。

似乎互联网以及电子商务的繁荣还是不能满足需要的，公司发现这些烟囱系统只有在部门保持相对稳定的情况下才可以运行。然而，由于重组已经成为公司管理中一个非常普遍的工具，而且在每次重组中部门的责任和义务都会发生改变，支持部门的系统也需要相应地进行改变。在通常由销售部门掌握注册登记的公司中，销售部和市场部突然间合为一体，于是支持这两个部门的系统就需要对合同登记和客户关系管理进行整合。或者，在某些情况下，部门发生分立，这就意味着此系统现在由两个不同的部门依照不同的议程加以使用。这些系统应该能够延伸到其它系统中，并能够访问或者操纵数据，提供在系统需求中最初并未提出的处理程序，并迅速实现之——尽管该系统通常是用 C++语言编写的，并且需要访问它的系统是用 Java 编写的。

这些就足以使你想要放弃，去从事一些像商业空中交通控制，有组织犯罪或者自动贷款偿还之类的轻松事了。

这就是中间件试图涉足的问题。通过提供一个所有这些不同系统均可与之对话的共同基线“胶水”，使IT开发人员可以排除干扰，更加专注于系统的领域相关的部分。如果我们可以通

过某种方式找到系统中使构建企业系统程序员“头痛”的部分的话，我们就可以把这些功能性的部分置于一个领域依赖的代码可以访问的层中。系统中的这些完全独立于领域的部分通常被称为横切关注点（crosscutting concern）。

关注点（concern）是软件系统中一个有趣的特殊领域。许多情况下，大多数软件架构和设计幕后的驱动目标就是将相关的关注点捕获到模型化良好的结构中去。例如，在某个面向对象系统中，其要旨就是捕捉所有和在系统中成为一个“人”相关的事宜，并置于封装良好的、整合良好的Person类中。于是我们就可以扩展Person的关注点为包括学习的人——我们将表示他们的软件类型称之为Student，以及施教的人——称他们的类行为Instructor，等等。这种做法的更正式的名称称之为关注点分离（separation of concern）。

遗憾的是，关注点并不那么易于被设计成一层。这一大堆的问题并不能被轻松地再整合到那些我们可以继承的基类或是我们可以直接地组合或聚合的支持类中。例如，考虑一个不管怎样看都相对较简单的需求：我们想在每次有方法进入以及随后退出的时候各发送一个消息到诊断日志。（如果这个例子看起来太不起眼了，我们可以考虑在方法进入时启动一个分布式事务，并且在方法退出时将其提交；这个实现几乎是一样的。）这就是一个横切关注点，因为这个关注点横向地贯穿了经典的面向对象系统中的静态继承树，横切在继承线路上，这正是它的名字的由来。

这里的问题在于，如果不加控制，横切关注点要将在其他方面都整合良好的代码转换成为像一砣面条一样繁杂的代码和逻辑。考虑我们那个简单的例子，在正常的 Java 语言规则之下，没有任何途径可以自动提供这项行为给感兴趣的客户。如果一个类想让它的方法对诊断消息作日志，那么 Java 开发者必须手动完成：在每一个方法开始处，处理“方法进入”消息，并且在每一个方法退出处（事实上，准确地讲，是在方法中每一个可能的返回点结束之处），处理“方法退出”消息。这是一个“选择接受（opt-in）”系统：如果一个开发者忘记了将这段代码置于某个特定的方法中，那么该方法就不会记录日志。这使得代码在变得异常地长之前，就已经索然无味且易于出错了。

横切关注点在很大程度上，使得软件开发变得更加艰难了。例如，考虑下面这个并不完全，但是令人震慑的关注点列表，它们对每一个企业开发者来说，无论如何都是必须要面对并要

解决的。

- *状态管理*：状态管理是由两个在同一个名称下相关的但是不同的元素合并而成——瞬时状态和持久状态。对于瞬时状态来说，其管理是由厚客户端或胖客户端应用隐式地完成的，在瘦客户端系统中，它会变得更加复杂。由系统来处理部分管理可以降低应用程序员面临的问题复杂性。相类似地，对持久性状态来说，数据必须被存储在永久性的数据存储中，并且以后可以将其读取出来。通常使用的是关系型数据库，因为对象和关系型数据库不是完全相一致的，这使得将一个复杂的对象模型存储到一个关系型模型中成为了一项极度令人受挫的工作。有时我们称这种情况为*对象-关系阻抗失配（object-relational impedance mismatch）*。
- *处理*：尽管这看起来好像本来应该是领域自身要建模的部分，但是我们该如何执行系统的处理其自身就可以被认为是一个横切关注点。例如，处理应用级别的故障（failure）可能会是一项很艰巨的任务——如果系统可以以更系统化的方式来处理故障，那么它可以变得容易得多，特别是在处理诸如关系型数据库或消息代理这样的外部资源时。或者，像这里列举的第二个例子，如果系统可以为我们作某些处理，按照某套标准来评估数据的状态，例如依次标识哪些代码要被执行，那么它也可以变得更容易。
- *同步*：企业系统本质上是多任务的，这意味着同时可以有超过一个的客户端请求来访问某项共享资源。因为并不是所有资源都可以处理这种应用场景，所以开发者必须显式地编码去阻止对这种资源的同时访问（要么是在 Java 代码级别上，要么是在共享资源级别上）。
- *远程和通信*：由构件构建而成的系统彼此之间必须以某种方式进行通信，并且这种通信根据系统或构件的部署环境的不同而要采用不同的形式。例如，某些通信需要以面向消息的形式实现，而另外一些需要用到请求-响应方式。
- *查找*：当进程间的通信很容易就可以到达时，另一个问题就变得越来越突出了，这就是我们希望怎样与之进行通信的进程、机器或对象。通常情况下，首选的方法是直接将目标硬编码到代码基中，这很难让人接受，而且，其结果是必须引入一个新的间接层，用来实现从对开发者来说友好的名称到资源名之间的映射。
- *对象生命周期管理*：在面向对象的企业系统中，对象生命周期成为了一个特殊的关注点。例如，在一个单一对象表示数据库表中的一行记录的系统中，如何创建对象，何时创建对象，以及该对象在内存中将保持多长时间，都将成为重要的关注点。具有太多的对象会创建出占用内存比实际需求要大得多的系统，而具有太少的对象意味着系统将把时间

花费在频繁地将对象换进换出内存上。

- *资源管理*：线程、数据库连接、套接字、文件，所有这些资源比起堆内存来说都要更难于管理。它们的生命周期存活于 Java 虚拟机之外，并且需要以一种对并发使用来说友好的方式来被获取和被释放。更重要的是，某些资源可能要求有某种缓存工具以使其能够被最有效地使用，而不是每一次都使用开销昂贵的获取算法。

就像你能够看到的，这已远不是一个琐碎的关注点集合了，如果在我们每次启动一个企业项目时，都让程序员去解决这些问题，那么这就好比让木匠在每次踏上一个新的建筑工地时，都去重新铸造他们的工具。诚然，这有可能会构建出精确满足该项工作需要的工具集，而且甚至可能会构建出许多只对该项特定工作适合的工具，但是这要付出多大的代价呢？更糟的是，这要求木匠同时也得是铁匠，在今天，这种一专多能的木匠真是凤毛麟角。

因此，我们的目标是寻找一种能够捕获这些与问题域无关的横切关注点，构建成第一流的结构的方法，就像一种面向对象语言或系统的目标是寻找能够捕获相关联的状态和行为，构建成第一流的结构（典型的就是一个类）的方法一样。在中间件系统中，我们指望那些在系统不同的节点上运行的软件处理来提供对这些横切关注点的支持。

J2EE 实现

在过去的几年中，人们已经提出了许多技术用以处理横切关注点，其中包括一项你应该早已非常熟悉：面向对象的软件开发。对象（或者是更特别的继承技术）被视作是一种提取共用代码（横切关注点），将其置于一个单一位置的途径，在这种情况下，即一个基类。而且，尽管对象极好地提供了这种能力，但是我们后来还是发现简单的继承（或者更准确一点地说，是继承的实现）并不足以解决问题。例如，我无法在我的导出类中继承一个能够提供追踪行为的基类，因为在导出类中基类方法通常要被重载，以提供该导出类需要提供的功能说明。

众多用以处理横切关注点的最为有趣的方法中，最精彩的就是通过扩展语言（或者直接发明一种新语言）来更好地以第一流的水准处理横切关注点。这正是像 Eclipse 的 AspectJ（面向方面编程）和 IBM 的 Hyper/J（面向主体编程）这样两种最流行的语言幕后的目的和动机。

另外，还有包括 OpenJava 和 Javassist（二者都是元对象协议系统）的面向研究的方式。因此，这些方法不仅有趣，而且还是那些完全位于“正常”的 Java 谱系之外的方法的典型代表。由此我们可以得知，它们的确不适合放入像本书这样的一本书之内。我强烈建议你把它们中的每一个都看看，还有同一领域中的其它方法。因为像这样的方法很容易就会成为语言中“下一个大事件”。

处理横切关注点的另一个办法就是取用一个类，然后使用 Java 开放的类加载器机制（请参阅第 70 项），事实上是在加载的时候修改编译过的字节码，注入一些额外的代码，通常这样就可以提供横切关注点行为。例如，在当前的 Java 数据对象（JDO）规范中，JDO 开发者就是通过运行一个“增强”工具来修改编译过的类，以在类中注入持久性行为的；运行时刻的字节码增强工具/系统在程序运行过程中于容器内完成这一任务，而不是强制开发人员在编译过程中完成。许多 J2EE 容器对这一概念都不以为然，但是其它的开源库，如 Nanning 和 AspectWorks，却正在致力于运用这一概念来在它们企业系统中开发的实用的应用。

J2EE 运用了“拦截模式”（Interception pattern）[POSA2,109]这一经典的技术，来将横切关注点捕获到优良的结构中。在一个基于拦截的应用场景中，来自于客户端的请求会被第三者（即不是客户也不是程序员编写的服务器逻辑）有效地“截取”，以提供某种行为。例如，在 EJB 中，当一个构件想要参与到某个事务中，从而为它要处理的客户端请求提供原子性（Atomic）、一致性（Consistent）、隔离性（Isolated）和持久性（Durable）语义时，拦截器，一个在部署时刻生成的代理，将截取该客户端调用，启动事务，然后将这个调用（以及隐式地将它所启动的事务）传递给该调用想要调用的 EJB bean。当这个 bean 返回调用结果时，假设这个 bean 不显式地放弃事务的话，该事务将被提交，而 bean 的操作的结果在事务提交成功时，将被永久性地保存到数据库中，如果事务提交失败，将会移除该结果并抛出异常。

事务执行只是 J2EE 基于拦截的技巧包的一个实例。例如，在 EJB 实体 Bean 中，EJB 容器不仅会拦截到 bean 的调用，以启动一个事务，而且可能会利用此机会从数据库中将该 bean 重载到内存中（以保证其拥有来自于数据库的最新且最大的数据副本）。这就意味着拦截器需要知道怎样从底层的数据库中加载（并存储）bean，典型的是关系型数据库。为了做到这一点，EJB 运用了另一种被称为代码生成（code generation）的重要技术。同样是在部署时刻，它将使用包含在 bean 的部署描述符中信息来构建代理（Proxy）。尽管你对部署描述符

的熟知可能仅限于它们是 XML 文件，但在技术上，这些都是有关构件中的 bean 的声明式属性（declarative attribute）；这些对 bean 的事实声明如果用 Java 代码来表示就不会这么容易了。通过使用这些信息，容器将构建一个代理，以使其能够处理很多操作，例如去了解（例如）需要哪种 SQL 用来保证能够处理 bean 到数据库的所有可能的持久性操作。

拦截并不只是为 EJB 而保留的。Servlet 容器也拦截进入的 HTTP 请求，并检查该请求以决定该请求应该被发送到哪个 Web 应用，而且还确保该 Web 应用所需要的所有资源均已加载，并已准备好处理该请求。作为一个新增加的特性，在 Servlet2.3 以及更新的容器中，servlet 容器将查找所有程序员自定义的且绑定到特定请求的拦截器（即过滤器，filter），同时执行它们。因此过滤器本质上是一个使用 URL 模式为描述标准的，位于 Web 应用的部署描述符声明要其去拦截的所有资源之前的，由程序员自定义的拦截器。Servlet 程序员将构建过滤器，他可以有选择地用定制的版本去取代标准的 HttpServletRequest 和 HttpServletResponse 对象，以提供各种希望得到的行为，例如加密、压缩或是有可能对返回数据进行的直接替代。（顺便说一下，拦截技术并不仅局限于中间件容器之中；请参阅第 6 项）。

因此，J2EE 规范在其他各种技术中选择使用拦截技术，从而在企业系统中为横切关注点提供支持。例如，JDBC 捕获横切关注点来处理到关系型数据库的连接、认证以及 SQL 查询的传递。我们作为 JDBC 的客户端，仅仅编写了我们所熟悉的 connection . createStatement(…) 代码，而 JDBC 驱动则处理由此而生的细节问题。更经典的中间件应用场景是 servlet 容器和 RMI 查询：二者均要处理线程和连接管理，接收来自客户端的请求并将其传送给服务器端的处理代码，稍后再将响应发送回客户端。新发布的 JCA 也是要处理相同的事务，它为那些不是特别适合 RMI 及 JDBC 模型的系统，提供了通用客户端接口 API（Common Client Interface APIs），这些系统不仅包括遗留的消息系统和一流的事务处理系统，而且还包括像 SAP 这样的大型系统。

然而，很可能最普通的中间件应用场景仍然是 EJB 容器。该容器不但要处理与 servlet 容器及 RMI 查询所处理的类型相同的资源管理，即接收客户端的请求，而且还要进行事务处理。它处理从事务管理器处获取的分布式事务，自动地征募资源管理器（如相关的数据库）到这些分布式事务中，并自动地尝试在方法调用完成时提交该事务，这样，我们就不必再为这类事情而操心了。

遗憾的是，J2EE 容器本质上并不具备足够的知识来构建拦截器，以有效地处理我们程序员有可能创建的所有应用场景。例如，对于一个给定的 EJB 会话 Bean 方法，EJB 容器无法预先知道它应当被加以什么样的事务语义：这个方法是否应当在事务中运行？它是否应当借用调用者的事务？等等诸如此类的问题。在某些情况下，拦截器可以用于容器的整体（正如在 servlet 容器内部所发生的一样），但是拦截器仍旧需要额外的信息来了解怎样处理请求，例如调用哪一个 servlet 和/或过滤器来处理请求。

这恰恰是开发人员需要对 J2EE 系统给以帮助的地方，而且我们也正是通过以声明式属性向其提供必要的信息来做到这一点的。在 J2EE1.4 版本和更早版本中，这些工作都是通过部署描述符来完成的。在随后的 J2EE 版本中(JDK 1.5 以后)，这些工作极可能通过被直接嵌入到 Java 源代码中的定制属性声明来实现。另外一种方式，是 J2EE 部署工具会基于该信息去构建或者在内部配置它的拦截器，在理论上，这可以给予我们所期待的东西。

作为一名 J2EE 开发人员，你应当认识到，在许多层次上，中间件众多经典属性之一就是它的被动性。中间件系统是由客户驱动的，它要等到接收到客户的需求之后才能采取相应的行动。中间件系统是大型系统的一个组成部分，为我们去管理资源（请参阅第 73 项）。人们指望中间件系统去实现与其他系统的桥接，这会导致额外的性能和可扩展性的开销，所以千万小心不要由于失误而不得不接受那些开销（请参阅第 32 项）。中间件系统要将多个系统联系在一起，已使得它们可以被多用户以多渠道来使用。因此，你不要去对你和那些你正在处理的且并不会长期有效的数据之间的关系作出任何假设（请参阅第 45 项）。在你的头脑中一定要记住中间件系统的演化方式，特别是有关代码分离的部分，并且不要盲从于那些已经不再有效的建议了。

最重要的是，要认识到中间件让人感觉早于对象整整十年。在 J2EE 中，我们用对象来构建中间件的解决方案，但中间件的解决方法内部并不是对象的解决方法。当然，J2EE 已经尽其全力将对象的优点引入中间件环境——例如，我们采用了 servlet、会话 bean 和消息驱动 bean 的原理——但在其本质上，中间件是对象不可知的，而且这会对 J2EE 开发者强加以一种全新的思维模式，而这对他们来说可能并不熟悉和友好。当你认识到 J2EE 是 Java 的一个中间件时，你就会陷入到已经用许多方法解决过许多次的若干个问题之中。你千万不要因为

忽视过去系统中的教训而做无用功——通过理解中间件为什么要走他自己选择的路，可以加深你对 J2EE 演化方式以及最终如何构建成功可用的 J2EE 系统的理解。

企业计算的十大谬误

遗憾的是，在 J2EE 提出并解决了大量问题的同时，它也引入了许多它自己的问题。

构建一个分布对象系统是很困难的。事实上，构建一个分布对象系统，如果不考虑其对象特性，总是相当艰难的。Peter Deutsch，一位Sun的研究人员，将其总结为他在长期的工作中认识到的“分布式计算的七大谬误（7 Fallacies of Distributed Computing）”。James Gosling在其基础上又添加了一条，使其成为了八条[1]。我又冒昧地（并且可能有些自大地）多填加了两条。

就像 Deutsch 在他最初总结的七条的概述中讲到的，“基本上每个人，在第一次构建一个分布式应用时，都会作出下面七个（十个）假设。但是在长期的运行过程中，它们都会被证明是错误的，而且都会引发大麻烦和令人痛苦的学习经历”。

1. *网络是可靠的*。在面对网络时，它很难称得上我们所谓的可靠——对于当今的互联网来说，丢失报文、服务器宕机、路由器被黑掉而被用于不合符伦理道德的目的，等等这些依然是现实存在的。尽管你的网络可能真的是永远不会崩溃，但是你的网络并不是你唯一需要考虑的——所有那些你不能作出保证的网络也要保持处于激活状态。这就是为什么需要你的系统在面临故障的时候是健壮的，同时还需要你去定义系统所需的性能和可扩展性的类型，以使你可以确保你所拥有的基础设施可以处理故障。
2. *响应时间是零*。网络不是免费的——它会把时间花到通过网线（以及许多构成互联网的中间设备）传输数据上。在开发过程中，网络的响应时间经常是非常接近于零，如果开发部门与公司其他部门的网络是物理隔离的，那就更是如此，这在大型的公司中非常常见。事实上，在很多情况下，响应时间会尽其可能地接近于零，这是因为开发者的习惯是为了方便起见，在同一台机器上运行所有的层——应用服务器、浏览器、数据库和其它任何必需的东西。但是不要忘了，特别是在 Web 应用的情况下，你的客户端可能不

[1] 请查阅http://today.java.net/jaq/Fallacies.html.

是在 100-或 1000-兆的网络上，有些可能仍然是在拨号网络上，这取决于客户端的选择。正是因为这个原因，所以要确保作为表示层部分（请参阅第 52 项）或通信连接的主体部分（请参阅第 23 项）而被传输的数据量要最小化。

3. *带宽是无限的*。与我们想要的正好相反，网络在同一时刻可以发送的信息量具有有限的容量。特别是当企业系统暴露给公众，而某些我们潜在的用户仍然运行在慢速的拨号连接上或饱和的 DSL 连接上时（“什么，你是说我不能在使用你的应用的同时，观看视频流？”），我们将具有与 LAN 同等的带宽的想法是可笑的。这就是为什么有时尽可能多地将任务从线上移除，而采用厚客户端去访问传统的基于 HTML 的系统的原因。
4. *网络是安全的*。近年来，这已经一再被证明是一条谬误，即使是在公司内部的 Intranet 内也是如此。不仅黑客可以相当容易地穿透我们的防火墙（例如，防火墙不能防范 SQL 注入攻击），而且在许多情况下，你也不能相信在网络内部的人们——最近的研究表明，公司损失的 70%左右都归咎于雇员的偷窃和诈骗行为。
5. *拓扑结构不会发生变化*。对任何 IT 商店来说都会时常发生的一件事情就是变化。服务器增加了、服务器减少了、机器需要被替换了、硬件需要升级了等等。我们竭尽全力来跟上变化的形势，但是如果我们写的软件能够自动调整以适应变化（请参阅第 16 项），那将会很有益处。
6. *只有一个管理员*。合并、收购和资产分派都只是应用场景的一部分——你还使得进出公司的人达到了警戒水平，包括你的系统管理员。即使你今天有一个应用场景，到了明天就会有一个完全不一样的应用场景。正因为如此，你需要在构建企业系统时牢记管理（请参阅第 13 项）、部署（请参阅第 14 项）和监视（请参阅第 12 项）。而且，随着 PC 数量的增长，以及自己动手开发的程序的普及分发（其中像 Microsoft Access），不用提及像章鱼触手一样延伸的伙伴关系和联盟关系在商业中变得相当地常见，就是任何一个团队、部门、甚至公司拥有一个程序的假设都在发生变化。现在问题的关键是，你构建的任何应用都可能转变为另一个完全不同的公司所依赖的某些事物，而你却并不了解该公司。所以你不能假设你拥有自己的数据库（请参阅第 45 项），或是有随意改变构件接口的特权（请参阅第 2 项）。
7. *传输的代价是零*。对象通过网线传输并不容易。事实上，在 Java 中，除了原始类型之外，其他对象通过网线传输的在线表示都是同构的。这意味着这要超出将实际的二进制位通过网线推送出去所需的开销，你还需要合并一大堆的数据位，以打包和解包数据本身。正是因为这个原因，请关注你要产生的在网络上来回的数量（请参阅第 17 项），使

用的技术包括成批传递数据（请参阅第 23 项）以使得每一次在网络上的来回显得更有价值。

8. *网络是同构的*。当你停下来去思考.NET 迅猛地普及势头，与大量现有的用 Python、Perl、C++以及其它语言编写的系统并存的现象时，会很容易地认识到在系统级别上的同构性比硬件级别的同构性更不可能实现。因此，你需要仔细考虑在你的架构中的提供商的中立性（请参阅第 11 项），并且无论如何都要实现在当前去构建它，你可能在以后会无法拥有它的某些部分（请参阅第 45 项）。
9. *系统是整体式的*。尽管可能在早期的系统中确实是这样，但是关键是一个企业系统经常以某种方式与其它系统集成在一起，即使它们是访问相同的数据库。特别是在今天，系统的不同部分是在不同的时间被修订的（表示层变化了而业务逻辑却保持未变，反之亦然），这使得认识到系统不同部分需要被部署，以及进行版本控制，而且在很多情况下，它们被彼此独立地开发的这一事实，比以往任何时候都重要。因此，你希望支持基于构件的设计（请参阅第 1 项），并指望构建彼此间松散耦合的构件（请参阅第 2 项）。
10. *系统是可以结束的*。企业经常转换业务，并且经常改变环境。只有当你认为你已经完成了某件事时，商务专家会跑过来提出一些新需求或是一些对你刚刚做完的事情的变更。这正是为什么会产生拓扑结构变更以及为什么系统不能保证长期地同构的原因。

上面的这十条准则就是本书要构建的主要内容。

第二章 架构

理论与实践之间，从理论的角度来看没有差异；然而，从实践的角度来看，差异确实存在。

——Jan L.A. van de Snepscheut

架构——这是一个位于设计之上的层次，在此我们能够瞥见系统的最终轮廓、它将如何被构建，以及整个应用和系统的总体视图。

当你为企业级应用和系统规划基本流程和设计的时候，牢记以下项中包含的建议，你就能向优雅的目标迈进：建立一个能够对高性能、高可扩展性的企业级系统提供支撑的架构。

第 1 项：优先采用构件作为开发、部署和重用的核心元素

许多 Java 开发者在首次尝试构建基于 J2EE 的项目时，面临的诸多困难之一，就是 J2EE 应用的构建方式不同于传统的 Java 应用：J2EE 不是构建应用，而是委托现有的应用来生成要插入的构件的结构，这个现有的应用即 J2EE 容器自身。

乍一看似乎区别不大，但从两个不同的方面来看，其实隐含了巨大区别。首先，这表明开发者不再对构建对象本身感兴趣，而是专注于创建严密封装的构件，在 J2EE 环境中，构件由构成其的互相紧耦合的对象组成。第二，这表明我们必须遵守一系列严苛的规则，即使是作为构件的开发者，我们也不能违反这一点。

学习面向对象技术的开发者，从很早的时候起，就不断地被告诫，要提倡封装和数据隐藏。我们都还记得大师们教导的金玉良言。例如，“绝不使用公有域（public field）；而是编写读取器（accessor）和修改器（mutator）方法（用 JavaBean 术语表示，就是 getter 和 setter 方法）”，或是“对客户的视图隐藏你的实现”。因此，对曾经写过的每个类，我们都忠实地为每个私有字段加上 get/set 方法对，以及编写缺省构造器等等。

遗憾的是，这样就几乎偏离了面向对象提倡者们原来的想法。仅仅强迫客户使用 get 方法，

以得到对某个内部持有的数据结构的引用，并不能称为封装。这个问题很久之前就已经在《Effective C++》[Meyers95] 一书里指出了，更近一点的，《Effective Java》[Bloch, 第 24 项]也同样指出了这个问题。更严重的是，这会误导开发者在过于小的规模下（类和对象）考虑有效重用问题，而不是像 JavaBean 规范最初的意图那样，在较大的规模下去考虑重用。

还不相信？让我们考虑一个常见的集合类，比如 ArrayList。我们可以把 ArrayList 作为单独对象使用。不过要是想遍历集合的内容，如果不是明显地必需的话，至少也是希望能够用另一个对象，也就是 Iterator 实现类的对象。在类/对象的层次考虑重用，意味着我们要着重于考虑对“独立于 ArrayList 绑定的 Iterator 实现类”进行重用，这显然没有意义。由于这两个类本来就应该成对使用，所以这意味着我们应该在更高层次去考虑重用，这也就是我们称为构件（component）的层次。在实际环境下，它通常是一个单独的.jar 文件，在本例中它里面包含了两个类：ArrayList 及其内部的 Iterator 子类型。在使用 Collection 接口的情况下，还要包括像 Iterator、Collection 和 List 这样的相关接口，因为它们作为一个整体共同定义了客户能够信任的契约。

这把我们带入了对耦合的讨论。紧耦合的定义是，如果一个给定的“事物”必须做出变化，以适应另一“事物”的变化，我们就称前者紧耦合于后者。换句话说，考虑 ArrayList，如果我改变了 Collection 接口的定义，ArrayList 需要改变吗？当然要，因为 ArrayList 通过 List 也实现了 Collection 接口。如果我改变 ArrayList 的内部实现，那么其内部的 Iterator 实现需要改变吗？当然要。不过，客户端代码需要改变吗？不久之前客户端代码还把 Iterator 当作通用类型的 Iterator（并且没有向下转型）。尽管 ArrayList 与其内部的 Iterator 紧密耦合（反之亦然），不过只要客户端代码严格遵循接口编程，就能与 ArrayList 保持松耦合关系。（松耦合将在第 2 项中进行详细讨论，不过这里我要提前引用一下。）

还没看到联系吗？我们考虑一个基于servlet的Web应用程序。如果我们遵循传统的模型－视图－控制器模式（MVC）[1]，那么至少要编写两个类，它们或多或少是紧耦合关系：控制器servlet用来对进入的HTTP请求进行处理，然后把请求转发到视图JSP。控制器需要知道视图

[1] 这里其实有点用词不当：符合题意的恰当模式是表示－抽象－控制器模式（Presentation-Abstraction-Controller）[POSA1, 145]，因为MVC用来处理多个视图，并且视图能同步更新，这种方式在Excel或Word这样的GUI应用中很常见。

所依赖的数据元素，视图需要知道控制器已经做了哪些处理，以免重复工作。二者必须对视图所依赖的数据元素被绑定的名称达成一致。（HttpSession的“名称－值”属性提供了延迟绑定，而不是松耦合；详细讨论请参阅第 2 项）此外，控制器和视图都紧耦合于模型类，因为它们需要知道组成模型所使用的数据元素。问问你自己：单单重用控制器而不管相应的视图类/模型类，真的可行吗？视图能够在不先进入控制器servlet的情况下被成功执行吗？

对这个问题的回答一般总是“不行”。这表明，对于给定构件里的类，本例中就是你的表示层构件，相互之间暗含着紧耦合。这就产生了这样一种现实状况：对于紧耦合已经存在的地方，我们可以对传统的“全面封装”规则稍微放松一些，尽管我们曾经如此盲目地遵守这个规则。当然我并不是在建议，为了方便对字段进行访问立刻删除所有 get/set 方法，我的建议是，你应该对真正要保护的东西深思熟虑。例如，如果你的模型对象并不在表示层（注意这个术语；请参阅第 3 项）之外使用，尤其是当这些模型对象只是对数据的简单包装时，比如“数据传输对象”（Data Transfer Object）[Fowler, 401]，还要为每个字段加上 get/set 方法，真的有意义吗？请记住，封装是用来在具体实现发生变化的时候，为了保护客户端代码而设计的，其目的并不是在构件自身发生改变时，来保护构件。（开发者几乎不能从对单个的类进行封装中得到益处，因此只要把构件想象成更大、粒度更粗的类，你就离目标不远了。）

假定我们有几个类，它们相互之间紧密协作，以完成一些有用的工作。这些类要么被一起部署，要么就都不被部署。我们在寻找某种“大于对象”的东西，作为部署的核心单元。换句话说，我们要寻求的正是构件。

Servlet 规范已经抛弃了把单个 servlet 部署到 servlet 容器中的作法，而是采用 Web 应用的思想。Web 应用是一系列资源，诸如 servlet、JSP、模型类、工具类、以及静态资源（HTML、图像、声音文件等），它们共同地相互协作以提供所需功能。Web 应用采用单一的.war 文件部署，所以不可能出现：（a）只部署了部分应用；或（b）应用的各个部件之间的版本不匹配。实际上，.war 文件和 Java 的.jar 文件目的相同，都是为了使应用的部署更加容易；如果你曾经使用过 Java 1.0，那么你一定还记得“把.class 文件解压缩到 CLASSPATH 指定的路径下”这种部署风格，同时你也会认同，那时候的 Java 应用的部署方式不那么优雅。.war 文件提供的原子性部署意味着，只有那些被认为是 Web 应用的组成部分的资源才会真正出现在部署之后的应用里。

也就是说，只有你故意强行拷贝单独的文件到部署后的应用目录中，才会出现局部的基于“替换文件”的部署。

这么做并不好，原因很多。一个重要原因是可能会引入版本不匹配的问题。“只拷贝发生变化的文件到部署目录”看起来很有诱惑力，但是人们太容易忘记什么发生了改变，更重要的是，servlet 容器并不能以我们的观点去看待应用发生了什么样的变化。

考虑一下 ClassLoader。还记得吗？它由 servlet 容器创建，用来把你的 Web 应用从硬盘加载到 Java 虚拟机。每当 Web 应用发生“改变”，通常是“硬盘部署目录的内容发生变化”的时候，servlet 容器需要创建一个新的 ClassLoader。但你要是仔细阅读 Servlet 规范就会发现，当新的 ClassLoader 被启动时，它加载的是整个 Web 应用，而不是某个单独的 servlet。因为容器把整个 Web 应用当作单个构件，所以组成 Web 应用的所有 class 都要由同一个 ClassLoader 加载。因此我们完全可以想象：每当有一个文件被拷贝到部署目录时，就会产生一个新的 ClassLoader 实例，从而要执行加载整个 Web 程序的所有额外工作，这并不会带来任何实际效益。

顺便提一下，要确保理解类加载策略所产生的切实效果，请阅读第 70 项。

这些问题的关键是，Servlet（或 J2EE 中的其它）规范希望你使用构件来构造 Web 应用，而不是使用单独的类。JSP 规范则更进一步：通过鼓励“可重用标记库”的概念，促进了较小规模构件在 Web 应用内部以及跨越 Web 应用中的开发和使用。规范并不关心你创建和使用的具体对象，只要求这些返回给容器的对象本身要遵循某些契约（通过接口表现），以及规范所定义的必须遵守的约束。

不过，仅仅实现已有接口还不够。要构建成为一个构件的部分原因是因为你没有编写 main 方法，也没有必要知道代码将被执行的环境。一个典型错误是，servlet 开发者在更换容器之后，只是简单地假设“当前目录”不会改变，这个假设会让他们狠狠地摔一跤。对于某些 servlet 容器，这个目录是容器可执行文件所在的目录（比如 tomcat/bin 目录）；而对于另一些容器，它们会设定“工作”目录，并在此目录下安装和调用整个 Web 应用。最终结果就

是，如果你用../webapps/myapp/data.xml 作为参数创建一个 FileInputStream，希望从 Web 应用的部署目录里加载一个文本文件，这个方法可能会在某个系统上成功，而在别的系统上失败。基于这个原因，Servlet 规范建议使用 ServletContext.getResource 或者 ServletContext.getResourceAsStream 方法，这两个方法也都可以被装载 Web 应用的 ClassLoader 使用。

事实上，“上下文（context）”这个概念在基于构件的环境里具有重要意义。比如 servlet 应用程序中的 ServletContext，或者 EJB 中的企业 bean 上下文（SessionContext，EntityContext，或 MessageDrivenContext），它们是构件“通往外部世界的官方途径”，对外部世界的任何访问都应该通过上下文进行。如果容器要在底层作某种技术处理，又不愿意影响客户端代码，这就给了容器机会去截获应用程序的请求，并将其重定向到合适的位置。例如，当你从 servlet 转发某个请求到 JSP 页面时，就要通过 ServletContext 得到 RequestDispatcher，以进行真正的转发操作，这是因为集群容器可能会把 JSP 页面放在与执行 servlet 不同的机器上。因此，要是你试图直接访问 Java 虚拟机内的 servlet 实例，比如通过以前常用的 getServlet 方法，这个调用将会失败。

从某些方面来看，这也是为什么 Java 命名与目录接口（JNDI）被提出的原因——提供一套通用的 API 来查找资源，而不使用特定于每一个规范的 API，例如 RMI Naming 类提供的 API。所以，JNDI 的起始点被称为 InitialContext 并不是偶然的。

现在你可能已经很清楚，编写构件与你所经历过的应用开发很不相同。实际上，在编写构件的时候，你根本不是在进行应用开发；而是在编写供已有程序调用的程序库。之所以开发的是构件而不是应用，部分原因在于你的代码必须具有与那些使开发程序库（与开发应用相反）变得相当有趣的特性相同类型的特性。例如，为了尽可能提高单个构件的灵活性，通常最好是把定义的接口暴露给程序库的客户端，而不是具体的对象[Bloch，第 16 项]。特别是，这使得构件能够提供一个有趣的“挂钩点（hook point）”（请参阅第 6 项），以备将来使用。当然，在很大程度上，如果是编写直接供 J2EE 容器访问的构件，例如，servlet (还记得 javax.servlet.Servlet 吗 ?) 和 EJB(javax.ejb.SessionBean ， javax.ejb.EntityBean ， 和 javax.ejb.MessageDrivenBean)，那么这一点就已经具备了。但是，出于同样的原因，对于你自己的域类，也可以具备这一点，例如，你的绑定于 HttpSession 的模型对象。朝着这个方

向，你还得非常小心客户如何构造你的域对象[Bloch，第 1 项]，你是否允许别人从你的域对象继承[Bloch，第 15 项]，以及你的构件所返回的对象类型[Bloch，第 34 项]。

这导致的一个重要情形是，J2EE 构件，除了很少的例外，其它全部都是被动实体。换句话说，J2EE 构件要完成任何有意义的工作，必须从容器获取控制的逻辑线程。这个“控制的逻辑线程”的概念，通常被表现为一个真正的线程（换句话说，容器在响应调用链上的某个终端用户的请求时，将创建一个线程，并使用这个线程调用你的代码），它通常被称为活动（activity）或起因（causality），意思是你编写的构件不应该在“借来”的线程里做过于复杂的运算（例如，不要试图计算 pi，并精确到几百位），因为容器希望在某个时间点上取回这个“控制的逻辑线程”。如果不能取回的话，容器就很可能认为你的构件已经中止了，从而会决定把构件实例完全卸载。

这就在 J2EE 规范内部造成了某种窘境。因为除了让程序员控制线程以外，高频率的任务一般不能以任何合理的方式完成。一个典型的例子就是每隔 n 秒获取某个外部资源，以进行某种维护操作，或是每晚午夜进行的夜间操作。这个功能已经以 Timer 服务的形式被加入到了 EJB 2.1 规范中，但是在只支持早期规范的容器中，并没有标准的 J2EE 解决方案。除非你编写一个单独的应用程序，通过 HTTP 请求、EJB 会话 bean 调用，或者 JMS 消息队列传递等方法来调用到容器内部，从而将该“控制的逻辑线程”给予容器。

最后再次强调，J2EE 的关键特性是以构件为核心的属性，作为一个 J2EE 的开发者，你不得不去自己适应这个模型。做不到这一点，就违背了 J2EE 容器建立的规则，往往事倍功半。在很多情况下，还会导致许多与容器制定的策略相违背的代码；尽管你也许可以侥幸绕过当前版本容器的策略，但你的代码要是在以后的容器中崩溃，你也别感到惊讶。你应该接受构件的概念以顺应潮流，如果出于某种原因确实需要绕过容器，可以编写一个单独的后台进程或程序。

第 2 条：跨越构件边界优先采用松耦合

与许多术语类似，我们对“松耦合”这个词感到既熟悉又陌生：熟悉是因为，我们知道这是

我们的目标，应该为之奋斗；陌生是因为，我们并不确定松耦合代码究竟是什么样。许多程序员把采用“反射”（使用 java.lang.reflect 库）技术编写的代码称作“松耦合”代码，这其实意义不大。在讨论继续之前，我们有必要先搞清楚“紧耦合”、“延迟绑定”以及“松耦合”代码之间的区别。

重复一下第 1 项中“紧耦合”的定义：如果一个给定的“事物”必须做出变化，以适应另一个“事物”的变化，我们称前者紧耦合于后者。

紧耦合代码是这样的代码：它紧密依赖于正在使用的“事物”，而不论使用的方式是通过方法调用，还是某种更严密的协议。例如，考虑如下代码：

```
Person p = new Person();
int age = p.getAge();
```

这段代码紧耦合于 Person 类型的定义，如果 Person 发生了变化（比如修改 getAge 的签名），那么客户端代码将失效，并需要做出相应修改和重新编译。与之类似，Servlet API 紧耦合于 HTTP 协议规范：如果 HTTP 改变其规则，则 Servlet API（至少是 javax.servlet.http 包）也需要做出相应修改。

然而情况并非总是如此简单， API 的出现（诸如此类的方法调用）并非必然地表明是否存在紧耦合。有人认为以下代码算是松耦合代码，其理由是 Person 类及其 getAge 方法没有直接出现在代码中。这样，如果 Person 类的 getAge 方法发生了变化，编译器也不会报错。

```
Class c = Person.class;
Object o = c.newInstance();
Method m = c.getMethod("getAge", new Class[] { });
int age =
   ((Integer)(m.invoke(o, new Object[] { })).getIntValue();
```

然而，尽管我们在语法上没有绑定于 Person 类，但实际存在语义上的绑定。如果这个无参

数的 getAge 方法并不存在，或者通过 c 引用的 Class 对象不可访问，这段代码还是要失败。更糟的是，错误直到运行时刻（而不是编译时刻），才会被发现，要是你不具有真正优良的单元/集成测试包以捕捉这些错误的话，你就可能在最糟糕的时刻自找麻烦。这并不是松耦合代码，实际上，这是松绑定代码。意思是，方法调用（更技术化的词汇是，对方法调用的绑定）直到运行时刻才会被决定。（这就是为什么 Java 动态方法调用机制有时也被称作延迟绑定，直到运行时刻，也就是你确切知道具体对象的时刻，才能决定调用的是继承体系中的哪一个方法。）

考虑松耦合的一种方式是，问问你自己，是否真的不知道“别人”在做什么。比如 JNDI 通过访问从构造器传入的 Hashtable 或 Properties 对象（换句话说，仅仅是一个“名称－值”的集合），使用一种通用方法向 InitialContext（如果它不能从系统属性中推断的话）传递初始化设置。然后，InitialContext 将在这个“名称－值二元组的集合”中查找键和值，以决定下一步的操作。所以，作为一个程序员，你不必为有关“维护”JNDI 提供者可能需要的所有配置对象而担心。JDBC 的工作方式也很相似，它使用一个任意字符串来包含一个“JDBC URL”，来传递所有信息。

不过，在这种特殊情况下，我们并不能真正从 JNDI 层解耦合。如果你没有传入完全正确的字符串作为 Hashtable 的一部分，JNDI 顶多会抛出一个异常，或者是默默地接受这个字符串，然后在你以后试图进行查询调用的时候发生失败，抛出 NamingException 异常。传入一个对象，其值要与 JNDI 提供者期望的值相匹配，这项职责完全归你。要确保一切正确，你也别指望从编译器得到任何帮助。再一次，我们又回到了松绑定而不是松耦合的场景；要是提供者改变其特殊设定，我们的代码就会崩溃。

先回想一下，准确地说，紧耦合到底有什么不好呢？

就其本身来说，没有。紧耦合为代码的开发和执行带来了高效的方式，这一点松耦合代码无法与之相比。考虑上面基于反射的例子，在执行的时候，许多因素都可能导致故障。实际上，上面的代码甚至不能通过编译，因为这四行代码里大约有六个“被检查的异常”（checked exception）需要被处理，为了使代码范例的长度小于一页，我很轻率地忽略了它们。而在第一段代码中根本不需要处理这些异常，因为调用代码紧耦合于目标对象，编译器能确保在代

码执行以前一切正常。因此，即时编译器（JIT）能进行大量优化工作，使得对 getAge 方法的调用尽可能高效。

实际上，某种程度的紧耦合不仅值得，而且绝对必要，比如构件边界之内的紧耦合。在不知道 Collection 实现细节的情况下，编写一个能遍历 Collection 的 Iterator 是一项十分困难的工作，也是在浪费你我的时间。其实我只关心 Iterator 接口，而不是实现细节。你只要根据由 Iterator 接口所定义的隐式契约去实现你的 Iterator，我就可以使用它，而不用操心它如何工作。换句话说，我只要遵循由接口定义的契约，我就能和你的实现（实际上，包括 Iterator 和 Collection）保持松耦合关系。

弄清楚这一点很困难吗？反射并不是必需的，就能比基于反射的代码更能适应变化。

企业系统由于其复杂的本质，尤其需要松耦合：根据定义，企业级系统需要访问其它系统，也要被其它系统访问，包括公司内部或外部的系统。在第一次发布的时候，也许你能控制系统内部的所有选择，比如客户的浏览器；但是在即将进行第二次发布的时候，你会发现你失去了过去拥有的控制权。构件趋向于相互之间版本独立，这也是我们的意图。因此，事实上试图在一方发生变化，而另一方不能变化时，还要保持构件之间的紧耦合关系，就像是给早期心脏病患者下猛药一样，让我们遇到极大的麻烦。

我们看看具体情况。假设你编写了一个域逻辑层，不论是否通过 EJB、WSDL，或 POJO （普通 Java 对象（plain old Java object），Martin Fowler 及其同事发明了这个词）暴露都可以。你要向 paySalary 方法传入三个参数：Person，Account 和 comment。在 Java 接口定义中，可能像这样：

```
public interface MyBusinessLogicInterface
{
  public void paySalary(Person p, Account a, String comment);
}
```

在第一次发布的时候，一切正常。其它部门的人听说你的系统很好，就编写客户端代码进行

调用，来为员工发放薪水。太棒了！

可是，最初的高兴过后，你发现需要作一些改动。特别是，你需要添加一个参数，即需要支付给员工的总钱数，因为某些员工（承包人，或其按小时计薪的员工）每个计薪日的工资是不同的。你要如何修改才不会破坏原有客户端代码呢？

有人建议，可以编写新接口，比如 MyBusinessLogicInterface2 或 MyBusinessLogicInterfaceEx。从老的接口继承，使用方法重载，然后提供一个新的 paySalary 方法即可。需要新接口的客户端可以显示地查找这个接口，只需原有功能的客户端可以继续保持不变。当然，你如果非常小心，就可以避免在两个方法中重复代码（如果它们在不同的 bean 实现的话）。但是三四次修改之后，你就会面临没有名字可用的局面：MyBusinessLogicInterfaceEx3 不能算是直观的接口名称。考虑一下，任何希望使用该系统的新员工，可能会面临五个接口，而且他们看起来也差不多：最近一个和前面四个到底有什么区别呢？

你也不要建议进行重构。重构所隐含的假定是，你在调用端和容器端拥有完全的控制权。你不能随意强迫客户端修改它们的代码，这么作只会使你成为孤家寡人。在一个良好封装的构件内部进行重构，完全是你自己的责任。除非所有人都同意，否则在跨越构件边界的范围进行重构，绝对是个糟糕的想法。

人们可能会采用的一种方法（许多遗留系统已经这么做了）是：通过传递额外的参数，去按照你的意图改变 comment 域。实际上，你甚至可能想把它作为一个微型的“挂接点”：在 comment 字段里传入附加的数据，通过对此数据进行解析，进行必要的处理。比如，“如果 comment 以 amount 开头，就解析余下所有数字，直到遇到分号...”。要是你觉得这种方法就可以帮助你编写可维护、可伸缩的代码，那么请合上本书，去最近的心理健康机构咨询一下吧。

这里的问题是，RPC（远程过程调用）风格的接口，尽管写起来简单方便（这里又遇到效率问题），其实是非常脆弱的契约，它很难承受系统中的太大变化。客户和服务之间契约的任何改变，都需要在两端重新部署，这种连锁反应甚至会波及整个系统。如果你的新代码破坏了太多客户端系统的话（或者太明显以至于高层管理/外部世界都意识到了这个问题），也许

它们根本就不会允许被部署。

要避免远程构件的紧耦合特性，一种办法是，优先使用数据驱动的接口，而不是行为驱动的接口（本质上就是传递消息和 RPC 的区别），第 19 项解释了这一点。批评者会争辩，这将破坏构件的面向对象性质，然而，正如第 5 项所述，这未必是件坏事。对通讯保持上下文完整（context-complete）（请参阅第 18 项），也对促进松耦合有帮助。

对于本地构件，本地方法调用的成本要远远低于远程方法调用的成本，所以是否“数据驱动”或者“上下文完整”就不再是个问题。例如，根据 Servlet 规范，在 Web 应用和容器之间能安全地创建 Context 对象，作为 Web 应用和容器之间保持一定距离的方式；例如，如果我能向 Context 请求有关构件的文件资源（getResource 和 getResourceAsStream），就不必对容器的部署模型作任何猜测。

已故美国最高法院法官Potter Stewart曾经说过，他无法为色情下定义，“不过我看到的时候，我就知道了。”[2]从很多方面来说，松耦合同样如此。它并不能通过一系列精确的规则来进行区分；而更像某种哲学上或风格上的形式。一般来说，你很容易发现紧耦合的系统，它们是那种经过一段时间以后，没人愿意在上面继续工作，并且所有人都同意重写的系统。松耦合的目标很明确：让以前不相干的系统，只需在任意一个系统或程序上作最小的技术调整，就能够进行互操作。如果其它项目（部门/公司）可以开始使用你的构件，而且不用担心你是如何实现的，这时，你就达到了松耦合构件的要求。或者，从相反的角度，对于松耦合的构件，你能完全改变自己的实现（也许是从Java移植到.NET，或者反过来），而不会让客户觉察到这种变化。

第 3 条：区分逻辑层（layer）和物理层（tier）

还有人记得客户/服务器系统吗？在过去十年中，它是企业级应用系统的主流架构风格。多年来，它在开发者心目中占据的位置仅次于面向对象技术。提供商提供成套工具，以简化基

[2] “Jacobellis v. Ohio, 378 U.S. 184, 197 (1964)”案例，引用自 http://caselaw.lp.findlaw.com/scripts/getcase.pl?court=US&vol=378&invol=184。

于窗体（后来是基于 GUI）的程序开发，在这些方便的编程工具背后，封装了大量访问关系型数据库的烦琐细节，一切看起来如此美好。

不过，随着时间的推移，我们发现了客户/服务器系统的诸多问题。

第一个问题有关资源管理。我们发现客户/服务器系统的架构具有先天的缺陷，服务器能同时处理的最大连接数目限制了系统的可扩展能力。考虑到绝大多数客户使用它们单独的连接的时间，不会超过连接保持打开状态的总时间的（平均）5%，那么提出某种方法，把连接从一个客户那里“借”给另一个客户使用，就显得很有意义了。

第二个问题来自开发者。我们很快就发现，企业级系统决不仅仅是对话框和数据，还要加上限制数据的规则（比如验证规则）。现在我们将其称为域逻辑，常常在客户/服务器系统的客户端进行编码。这么作的危险在于，更新这些规则需要对客户端程序进行重新部署，任何好的 J2EE 书籍都会这么说。随着客户端数量的增加，尤其是它们需要同时对数据库进行修改的时候，这一点很难平稳地做到。我们发现，要是把域逻辑放在客户系统之外，就能大大减少对客户端重新部署的次数。

但是这样的话，客户/服务器系统中能放置域逻辑的就只剩下服务器端了，这就意味着要编写某些依赖于提供商的代码。如果“服务器”是 Tuxedo 这种事务处理系统，那么就要编写与 Tuxedo 程序库一起编译和链接的代码；如果是客户端直接访问关系型数据库，那么域逻辑将被编码成依赖于数据库的触发器和存储过程。从那时开始，直到今天我们一直在担心的问题，就是对提供商的依赖（尽管事实上很少有公司真正遇到过这个问题）。因此，要把你的代码移植到别的 RDBMS 或中间件提供商的产品中，就需要真正做点事情，而不仅仅是一次重新部署。（这种避免依赖提供商的驱动力代表着工业界里的神话之一，当然与其它神话一样，要以无损的方式达到这一点完全不可能；请参阅第 11 项）

起初，通过在客户和服务器之间加入一个中间机器，这两个问题似乎都可以得到解决。我们可以在中间机器上，以标准格式来编写域逻辑，这样客户端就只要调用中间机器上的域逻辑，而不用自己进行管理。新的域逻辑只需部署到中间层，而不是每一台客户机器，这就降低了部署成本，我们常称之为“零部署应用场景（Zero Deployment Scenario）”。这也能使代码基

（codebase）更具模块化，因为域逻辑的改变，不仅使“生成显示所需要的代码”与域逻辑仍旧保持清晰得隔离，而且确保了域逻辑集中在代码基内部，这就使维护工作更简单，还能保持域逻辑的应用一致性。

同时，如果把域逻辑部署在中间层，我们还能以更简单的方式进行资源管理。例如，中间层可以自己管理与数据库的交互，这就能共享数据库连接，以支持更多的客户请求。这补偿了多客户连接的成本，并能减少 n 个客户同时访问系统造成的总体冲击，提高系统能够支持的最大并发客户数量也就变得可行了。（实际上，这时中间层到底能支持多少客户就成了新问题，如果我们有多个中间层机器去访问服务器，那么能够支持的客户总数就会大大增加，这也是我们常常讨论的普适的“集群场景（clustering scenario）”。）

这些其实你都知道；那么现在我为什么要提这些呢？因为中间层（J2EE 称之为应用服务器层）的存在有两个原因，而且你要认识到，这些原因中至少有一个是没有很特殊的需要去要求有一个适当的中间机器。

Java 首次在开发社区中引起轰动的特性之一，就是其内在的可移植性（移植到任何有 JVM 的地方。）回忆一下，我们对 Java 的试验始于 applet 和移动代码领域，自那以后，我们就考虑通过 Web 浏览器传输 applet。然而，其目的是一样的：通过修改存储在服务器上的代码，我们能够在任何后续请求中，把代码悄悄地发布到客户端。毫无疑问，applet 自身存在缺陷，但是其与生俱来的可移植性并不是缺点。简而言之，不论是否使用像 applet、Java 网络加载协议（JNLP，Network Launch Protocol）、URLClassLoader，或是别的什么技术，Java 代码都是以非常透明的方式跨越网络的，这就为无需人工干涉就把变更“推送”到客户端提供了解决方案。（细节请参阅第 51 项）

当然，在资源管理和数据库连接池方面，我们不得不需要某种合适的中间层。我们需要一个“集合点”，使用单一通道拒绝或接受客户请求。但是请考虑一下传统的“浏览器－servlet 容器－数据库”流程，再数一下所需机器的数目。尽管我们通常不这么考虑，其实从很多方面来看，servlet 容器就是这个中间层机器，它接受客户请求（这里通过 HTTP）发给数据库。当与本身支持数据库连接池的 JDBC 驱动程序组合在一起时，我们已经巧妙地达到了资源管理的目标，而此时并不需要 EJB 服务器，这能够帮助我们只在事务处理中去考虑 EJB（请

参阅第 9 项）。

无论如何，我们不能把所有的域逻辑都放进中间层。许多域驱动的服务（比如验证），需要在客户端或服务端完成。考虑一个 Web 应用，它要把所有（注意我是指所有的）的域逻辑集中在中间层。这表明，每一次表单提交都不得不在客户和服务器之间来回传递（如果域逻辑用 EJB 实现的话，还要从服务器传到 EJB 容器），以验证输入的电话号码是否正确，这就增加了一次（或两次）来回，并带来相应的性能负担（请参阅第 17 项）。大多数人可能会嘲笑，这里显然可以使用 JavaScript 来进行相同的验证，而不用增加网络负担。不过请考虑：要想正确存储数据，就需要遵守关系型数据库的完整性约束，所以向关系型数据库存储数据时，还会发生另一种形式的验证。数据库模式即使是在表示层使用验证规则进行了强制约束，它含有的那部分域逻辑，还是与你手工编写的代码实现一样多。

关键在于，不同逻辑层（layer）之间具有真正的区别，软件代码基的不同部分各自承担着一项关键职责。物理层（tier），则是网络拓扑中的物理机器，我们不能把二者混淆。确实，通常逻辑层都会映射到给定的物理层，比如数据层（layer）和数据库层（tier），不过直接假设这种映射是一对一的话，就会消除考虑其它架构的可能性。这些架构可能具有强大的性能和可扩展性，在避免网络栈上任何两台（或者更多）机器之间通信的过多开销方面，更引人注目。（请参阅第 17 项）

考虑一下进行这种明确区分的好处。例如，考虑常见的订单跟踪系统。公司的销售员使用该系统为不同的客户下订单，订单被传递给发货部门，当订单准备完毕后，向客户发送产品。现在考虑 Joe，一个希望在便携电脑上运行系统的销售员，如果这是一个传统的基于 HTML 的三层应用，Joe 要下一份订单就需要网络连接。如果 Joe 正在高尔夫球场，和一位准备购买产品的 CEO 在一起，Joe 就会感到不满，因为他要回到办公室才能下订单。如果这是一笔价值百万美元的订单，Joe 和管理层都会感到不满；销售产品的第一原则是：决不给客户改变主意的机会。（这就是为什么有人讨论有关“灵巧客户（smart client）”前端的原因；请参阅第 51 项。）

另一方面，如果我们这样设计系统：把表示层、域逻辑层以及部分数据管理层都放置在客户层，数据管理层的其它部分从客户层、中间层（通过某种第 3 类型的 JDBC 网络驱动程序），

一直延伸到服务器层。Joe 在网络连接断开的时候还能运行应用程序，数据被缓存在本地机器，当网络连接恢复的时候，就能够用其来更新中央数据库（请参阅第 44 项）。当我们把消息中介放置在客户层时，就可以把消息存储在本地，等网络连接恢复的时候再传递出去，这时这种方法尤其有用。

考虑应用程序架构的时候，为了在集中化管理和通信成本之间找到最好的“平衡点”，要确保描绘出系统的各个逻辑层次（表示层、域逻辑层和数据管理层）和网络拓扑中的物理层次（客户端、中间层和服务器层）。更重要的是，千万别错误地以为表示层就在客户端，域逻辑层总在中间层，数据管理层总是在服务器上。这么作也许是最好的架构，不过请有意识地做出决定，而不是靠习惯做出选择。

第 4 条：数据和处理程序要尽可能靠近

这个想法本身很简单，但常常以各种形式反覆出现：当你要和某种东西一起工作时，把它放在手边，这样你就不用满世界去找它。所以，CPU 会有一个片载缓存，显示卡也有自己的存储器，性能专家会建议你在应用中进行缓存，以加快程序处理速度。

从许多方面来看，这不过是第 17 项的另一种表达方式，因为把数据存放在永久位置（在 J2EE 应用中，通常就是远程数据库），并不会带来真正的问题，除非我们每次需要使用数据的时候，都去通过网络获取数据的成本显得过高。如果没有数据来回的开销，我们就没有必要缓存数据。但情况正好相反，所以我们要对数据进行缓存。我们还要确保缓存不会变得过大，以至于造成任何 OutOfMemoryError 异常（请参阅第 74 项）。顺便提一句，本地和进程内数据库（请参阅第 44 项）都是缓存数据的好地方。

不过，有时数据不能到达你的处理程序。一个常见的原因是，这需要对数据库加锁，而且数据被锁定的时间远远大于实际需要。因为实体 bean 位于容器内部，所以即使用会话外观模式（Session Façade）[Alur/Crupi/Malks, 341]来访问实体 bean，也会出现这样的问题。 正如第 23 项详细讨论的那样，把数据从客户端移动到 EJB 层的时候，使用整块传递的方式开销相对较低，但是数据仍旧需要在数据库层和 EJB 容器之间来回传递。

所以，当数据不能到达处理程序的时候，就得把处理程序靠近数据。

如果你考虑到了这一点，那么这就是把中间件技术当作首选的原因：通过在服务器（而不是客户端）上放置业务逻辑，中间件系统可以使数据尽量靠近相关的处理程序。当然，这些理由最初由 TP Monitor 提出，不过目前许多 EJB 产品的文档也隐含了这一点："充分利用本产品的缓存功能"等等。不过，正如我们先前讨论的，既然数据最终还是要存储到位于 EJB 容器之后的关系型数据库中，我们还是要承担额外的来回开销，当然，这些开销是要付出代价的。(注意这样的情况：如果 EJB 容器支持的话，开发者可以通过一个依赖于提供商的"专有"标志来告诉 EJB 系统，它是数据库的唯一用户，这样 EJB 系统就能在本地缓存数据，并按自己的需要持有数据，这就避免了一些开销。但这么做，你就失去了提供商中立性（请参阅第 11 项），不过有时候还是值得的。）

使数据和处理程序靠近，还有许多其它的方法；其中之一就是充分利用 SQL 语言的所有功能。例如，要计算系统中姓氏为"Nelson"的所有人的平均年龄，你有两种有效的办法：（1）使用会话 bean 或自己编写的客户端代码，把数据库中姓氏为"Nelson"的所有 Person 对象读出，然后自己进行计算，并返回结果；（2）让关系型数据库帮你进行计算，使用 SQL-92 的 AVG 函数，执行 SELECT AVG(age) FROM person WHERE last-name='Nelson'。这样就可以返回一行只有一列的记录，它的值是数据库里所有姓氏为"Nelson"的人的平均年龄。换句话说，所有的计算都在服务器上完成，而且在数据库中检索符合要求的数据速度很快，也不需要额外的数据传递开销。

另一个例子，它也经常在新闻组和邮件列表里作为常见问题出现，"如何得到 ResultSet 里的行数？"，简短的回答是，"无法得到，ResultSet 里没有 getRowCount 这样的方法。"有时还有另一种回应，建议你使用 JDBC 2.x 驱动程序，它支持可滚动的 ResultSet 对象（请参阅第 49 项），"你可以滚动到 ResultSet 的末尾，看看你在哪一行。"不过这样做的副作用很大，要么是所有数据都在网络上传输一遍，要么是强迫在数据库端导航到 ResultSet 的末尾，并把那一段窗口数据传递给你。一个较好的办法是，在执行真正的"SELECT"语句获取数据之前，先在 WHERE 谓词之前使用 SELECT COUNT()，以得到行数：

```
Statement s = . . .; // get the Statement from the usual
                     // places
ResultSet rs_count =
  s.executeQuery("SELECT COUNT(*) FROM person " +
                 "WHERE last_name='Nelson'");
if (rs_count.next() == false)
{
  // Something went horribly wrong
}
int count = rs_count.getInt(1);

ResultSet rs =
  s.executeQuery("SELECT * FROM person
                  WHERE last_name='Nelson'");
for (int i=0; i<count; i++)
{
  // . . .
}
```

这里会出现一个小小的竞争条件，除非使用事务加以保护，否则就可能出现另一个客户修改了 Person 表里的内容，使计数发生错误的情况。要么接受这个小小的瑕疵，要么使用事务把这两个语句包装起来；这两种方法，尽管可能出现事务锁，但是比起“仅仅为了计算行数就读取所有的数据”的方法，其可扩展性要好得多，尤其在你除了第一个集合之外不需要其它任何数据的时候更是如此，例如统计搜索结果。

诸如 COUNT、SUM 这样的函数，有时被称为聚集函数，因为它们对表中数据集合进行操作，并根据这个集合得到聚集结果。此外，由于有了 SQL 闭包所具备的强大功能（请参阅第 41 项），这些聚集函数可以在任何关系型结果上进行计算，例如视图（view）或嵌套 SELECT 语句。SQL-92 及其后继版本，SQL-99 或 SQL 3，提供了许多简单的聚集函数，比如 COUNT、SUM、AVG、MIN、和 MAX，所有这些都会按你的预期去工作。许多数据库产品还具有提

供商特定的挂钩（hook），赋予你自己定义聚集函数的能力。有些数据库提供商甚至走得更远；例如，微软 SQL Server 有这样的功能，在定义表的时候，表中的某一列可以通过用户自定义的函数计算得到：

```
-- SQL Server syntax; other databases provide something
-- similar

--
-- SQL Server syntax for creating a user-defined function
--

CREATE FUNCTION ConcatName(@LHS nchar(80), @RHS nchar(80) )
RETURNS nchar(161)
AS
BEGIN
   RETURN ( @LHS + ' ' + @RHS )
END

--
-- Create a table that defines a column that
-- uses the above function
--

CREATE TABLE Person
(
  ID int PRIMARY KEY,
  first_name nchar(80),
  last_name nchar(80),
  age int,
  full_name AS
```

```
    (
      dbo.ConcatName(first_name, last_name)
    )
  )
```

当使用 SELECT 语句从表中获取数据的时候，SQL Server 将调用 ConcatName 函数，为 full_name 字段计算列数据；这说明我们只要执行以下代码，就能得到数据库中姓氏为“Nelson”的所有人的姓名：

```
SELECT full_name FROM person WHERE last_name='Nelson';
--
-- Assuming we have rows:
-- Row 1: 1, "Steve", "Nelson", 40, (ConcatName())
-- Row 2: 2, "Lori", "Nelson", 25, (ConcatName())
-- Row 3: 3, "Kirsten", "Nelson", 12, (ConcatName())
-- Row 4: 4, "Marnie", "Nelson", 7, (ConcatName())
--
-- Then the query above returns
-- "Steve Nelson","Lori Nelson","Kirsten Nelson","Marnie
-- Nelson"
```

同样，所有的处理工作都在服务器端完成，所以客户端没必要执行额外的工作。这种方法与“将相关逻辑放在会话 bean 中”形成了对比，再加上会话 bean 在实体 bean 上执行字符串连接操作将隐含着从数据库读取 first_name 和 last_name 字段这种开销的事实，你会发现这种方法的好处。当然，这里也有缺点，它严重依赖于提供商，如果你对不同数据库产品间的可移植性格外在乎，这就会给你带来很大的麻烦。（请参阅第 11 项）

实际上，第 4 项最终要表达的意思是，使用数据库的存储过程来处理数据。数据存放在数据库中，而处理程序往往要比聚集函数或提供商扩展所能表达的更复杂。存储过程提供了从数据库中返回结果（或者修改数据）之前，对数据进行完全过程化的、逐步处理的能力。不过，

存储过程存在两个主要缺陷：它们不是用 Java 语言（你不得不学习新语言）编写的；它们也不能被移植到别的数据库系统中。幸运的是，很快就有解决办法了：作为 SQL/J 规范的一部分，许多数据库提供商希望，对 Java 语言在存储过程中的使用进行标准化。一些著名的数据库提供商，比如 Oracle 和 IBM，已经这么做了。也许这将解决这两个问题，不过实用化可能还要过一段时间。（另一方面，根据存储过程的复杂程度，把它们移植到别的数据库产品中也许并没有想象中那么费力。）

最后，无论你是在处理器附近缓存数据，还是在数据附近进行处理，都要在架构中把二者尽可能地绑定到一起，这样才能最小化进行处理时必须产生的通信，也能避免跨越网络移动数据的开销。（请参阅第 44 项）。

第 5 条：牢记标识引起的竞争

大多数程序员都同意，对象是状态（数据）和行为（方法）的统一体。不过，任何基于对象或面向对象的语言，都存在着第三元素，从许多方面看，正是这个神秘的第三元素使对象世界得以运转。请看以下代码：

```
public class Person
{
  private int age = 18;
  private String name;

  public Person(String n, int a) { name = n; age = a; }
  public void happyBirthday() { ++age; }
}

Person p1 = new Person("Cathi", 29);
Person p2 = new Person("Alan", 35);
p2.happyBirthday();
```

代码运行之后，Alan 的岁数是多大呢？或者，更准确地说，p2 所引用的对象的岁数是多大呢？当然，Alan 的岁数是 36。那么 Java 如何区分名为“Cathi”和名为“Alan”的对象有何不同，从而作出正确操作呢？

这个问题并没有什么技巧，即使是最没有经验的 Java 程序员，通常也能作出正确回答。这里的技巧在于，每个对象都隐含一个 this 参数，由它唯一引用当前对象。以前的 C++程序员甚至会这么说，实际上这是指向内存中对象的指针，因为两个对象不可能占用内存的同一位置，所以对象的地址是唯一的。（当然，Java 程序员知道，尽管语言中提供了 NullPointerException 异常类，但并有指针这样的东西，希望 C++程序员就不要不懂装懂了）

事实上，对象由三部分构成：状态、行为和标识。正是标识使得 Java 语言平台能区分前面的代码和以下代码，这里 p3 和 p4 都引用同一个对象：

```
Person p3 = new Person("Stephanie", 10);
Person p4 = p3;

p3.happyBirthday();
  // How old is the Stephanie object at this point?
```

这些似乎过于简单了，不过请坚持一下。

通过标识可以很容易地访问对象，但是标识也存在于标准的企业系统中的其它地方。看看下列语句，标识在哪呢？

```
INSERT INTO person ("Cathi", 29);
INSERT INTO person ("Alan", 35);
INSERT INTO person ("Stephanie", 10);
```

即使不知道数据库模型，我们也能判断关系数据库应该具有某种标识。因为即使在这些表里

没有定义主键，还是能区分不同的行。数据库使用某些依赖于实现的模型来保存每行的标识。（例如 Oracle 里，称为 ROWID，它是隐含的列）无论采用何种机制，它们的目的都与 C++ 或 Java 里的 this 指针基本相同，即提供某种标识。这使得我们能这样写 SQL 语句（注意现在 Cathi 的年龄是 30）：

```
UPDATE person SET (age = age+1) WHERE name="Cathi";
SELECT name, age FROM person;
```

那么，为什么我们关心这些呢？对象是有标识，行也有，那又怎么样？

传统面向对象设计方法的一个基本理论是，抽象问题域的实体应该映射到单一对象。换句话说，如果我们用堆里的一个对象表示 Cathi，那么，所有对 Cathi 的访问都应该通过该引用进行，并且这些引用都要指向这个单独的对象。要是不小心弄错了，就会造成故障：

```
Person p1 = new Person("Cathi", 29);
Person p2 = new Person("Cathi", 29);
p2.happyBirthday();
  // So how old is Cathi?
```

从许多方面看，这也是为什么要区分==操作符和 equals 方法的原因。前者对标识是否相等进行测试，而后者对内容是否相等进行测试。这种区分对 Java 编程非常关键，请看代码：

```
if (p1 == p2)
  System.out.println("It's the same object!");
if (p1.equals(p2))
  System.out.println("It's equivalent, but not identical.");
```

目前为止，一切还算正常。

事实上，在我们设计系统所使用的概念中，标识非常重要，所以我们希望在分布式对象系统

中对其进行模拟；幸运的是，使用 RMI 机制（或者 CORBA，也一样）可以使其变得非常简单：

```
PersonManager pm = (PersonManager)
  Naming.lookup("rmi://localhost/PersonManager");
Person p1 = pm.findPerson("Cathi");
Person p2 = pm.findPerson("Cathi");
p1.happyBirthday();
  // Thanks to distributed object identity,
  // we have logical identity
```

当 RMI 查询发生时，我们先取得某种“查询”管理器来管理 Person 对象，在 EJB 环境中就是 Home 接口，后续的查询将返回远程对象的存根（stub），它们引用在服务器上的相同对象。所以通过引用 p1 增加 Cathi 的年龄，隐含了对 p2 进行相同操作，因为逻辑上它们引用的对象也相同。（从技术上讲，引用 p1 和 p2 是不同的代理对象，但是因为代理对象（存根对象）指向服务器上的相同对象，所以逻辑上它们是同一个对象。这就是为什么远程存根被编写得非常仔细，使得在你调用 p1.equals(p2)的时候，返回 true）

换句话说，分布式系统中的标识并不仅仅是内存地址，或是行为和状态的逻辑映射；在分布式系统中，标识还包含了对象生存的硬件环境。而且，如果去对 Robert Frost 的旧诗进行释义，那么世界将会完全不同。

再关注一下远程的 PersonManager/Person 代码，这次我们着重于把标识看作是对象所存活的机器的要素之一。我们假设 PersonManager 只能在本地构造对象，那么，每个 Person 对象存在于什么地方呢？当然，是在与 PersonManager 相同的机器上，这表明要将系统扩展到包含大量 Person 实例的时候，我们就会被“单一机器所能支持的最大数量”这个内在约束所限制。无论我们在集群里加入多少机器，我们得到的 Person 对象也不可能比这个限制更多。

更糟的是，我们面临这样的问题：我们需要内置某种同步机制，以确保两个访问器方法不会因为同时修改数据，而导致数据的损坏。毕竟，如果对象有标识，就可能出现多个客户需要

同时访问对象的情况。加锁意味着客户的同时访问变得不可能了（这也正是加锁的意图），这把我们带到了对“锁窗口”的讨论（请参阅第 29 项）。

结论很简单：对单一共享对象的可扩展的访问是不可能的。即使对象实现的内部竞争不会给你造成麻烦，并且我们也没有达到底层服务器能够处理的工作量的限制，那么“到对象的调用需要进行往返”这个首要条件（请参阅第 17 项），也会给你带来大麻烦。现在再看一下 Martin Fowler 对分布式对象系统的描述就很清楚了：“像把你吸入麻烦的黑洞。”[Fowler, 87]

EJB 的支持者已经准备好了反驳辞：“但是使用 EJB，我们有很多增强功能来处理这个问题，例如，钝化（passivation）。”是的，但前提是此刻客户已经不再使用某些对象了。钝化的工作方式与操作系统级别的虚拟内存非常类似，以至于有人建议抛弃钝化，就让操作系统来干这活儿，同时也去忍受与虚拟内存相似的缺陷。如果你有 100 个客户，每人使用 100 个活动对象，那么逻辑上，这 10000 个对象应该保持活动，不能钝化。但如果服务器的启发式算法认为，只应该使 5000 个对象保持活动，服务器就会钝化另外 5000 个对象，然后服务器就会有大量时间消耗在激活/钝化的系统颠簸之中，这与物理内存不足时，操作系统所表现的情况类似。这种强制出现的大量页面失效（或者说是激活/钝化 EJB 对象）是真正的性能杀手。

EJB 的支持者还在继续：“使用集群怎么样？我们能把这些对象分散到网络上，而不是集中在一个单一系统中。”但这需要几个前提。举例来说，这要求客户比较均衡地访问这些对象；如果某个对象被客户过多地访问，那么我们就又回到了原来的问题，即系统所能支持的最大对象数量这个约束。（顺便提一句，这正是远程单件（Remote Singleton）的危险之处，无论你对同步实现代码进行过多么深入的优化，希望避免对锁的不必要占用（请参阅第 29 项），但你总是被底层的远程机制或网络通道所束缚。）

那么，在 EJB 环境中，标识究竟适合做什么呢？

从许多方面来说，试图使用 EJB 来创建基于标识的系统将会令你碰壁。EJB 对标识的理解依赖于标识的外部形式（比如，客户或数据库中数据的外部表示），而不是依赖于我们在 Java 中习以为常的标识机制。这就是为什么，作为一个 EJB 程序员，你不能假设某个客户请求最终被内存中的同一个对象实例处理完毕。为了保护客户层次或数据层次上的标识，EJB 明

确抛弃了对象层次上的标识。这就是为什么 EJB 容器能如此自由地维护对象实例池。（这也是为什么用 EJB 编写一个单件如此困难，因为单件依赖于对象标识，而 EJB 抛弃了它。）

无状态会话 bean 隐含地就没有标识。实际上，无状态会话 bean 也许应该被称为“匿名会话 bean”，尽管在某些情况下，它们也能维护状态。但是你不能确定你现在调用方法的对象，就是你上次调用的那个；所以，它们没有标识。这种隐含地没有标识，可以允许最大的可扩展性，所以许多专家都把无状态会话 bean 称为 EJB 中最好用的技术。（顺便提一下，这未必全对。在《Bitter EJB》[Tate/Clark/Lee/Lindskey]一书中，Mike Clark 进行的一些测试证明了这一点。）

有状态会话 bean 的标识（最初）只有创建它们的客户知道。注意，有状态会话 bean 并不是由客户独占的，如果你把引用传递给别的客户，那么他也能调用同一个 bean 实例的方法。不过，要是只有一个客户访问 bean，标识就不是问题，因为这时不会出现对 bean 的竞争。有趣的是，EJB 规范要求 EJB 容器实现同步机制，以防止多个客户对整个 bean 实例的并发访问；当然除非客户使用多线程访问实例才可能发生这种情况，不过这并不常见。本质上，对有状态会话 bean 的同步访问将大大增加调用的延迟；这是因为规范要求，当存在对有状态会话 bean 的并发访问时（EJB 规范，7.12.10 节），容器将向调用者抛出异常，所以容器至少要检查一下是否已经存在对 bean 的调用。

《事务型 COM+（Transactional COM+）》[Ewald]一书，指出了“每个客户一个对象”模型的两个有趣属性。首先，如果访问系统的每个客户都有一个对象，那么已有对象的数量就反映了当前访问系统的客户数量；要是对象由客户共享，这种信息就很难得到。第二，维护每个客户的状态更简单，因为每个对象本身就是对客户的标识，而不需要依赖于某些外部机制。所以一般来说，如果有状态会话 bean 不能由客户共享，就比较理想。（这种“每个实例服务于一个客户”的方式，进一步消除了对会话 bean 进行同步的需求。）

对于实体bean，我们确实遇到了麻烦。因为实体bean不仅有标识，而且这个标识在整个系统内众所周知，这样每个实体bean都成了一个远程单件。这个设计是必要的，要知道，实体bean隐含地具有标识，它希望代表的是系统内的一个实体，即使不是物理实体的话，也可能是数

据库里的一行[3]。这表明如果多客户需要访问单一实体，那么它们都在针对同一个逻辑对象操作。这还隐含了在任何特定的点，都需要维护两个标识：实体bean对象的标识，和数据库里行的标识。

实体 bean（或其它对象优先的持久化机制）可以采用下面两种基本方法来避免标识问题。

1. 保持实体 bean 的标识，然后不断进行缓存。不过在这种情况下，是与 EJB 规范相抵触的，因为规范明确提倡访问实体 bean 应该使用事务语义。记住规范的要求：EJB 服务器可以崩溃，但是实体的状态必须得到保存。如果服务器通过把数据缓存进内存，试图节约几次数据来回传递的开销，那么就存在“缓存刷新完成之前，进程可能死亡”的风险。（顺便提一下，这种方法对本地实体 bean 不起作用，因为本地实体必须在同一个 JVM 内被访问；要是本地 bean 本身就是一个远程实体 bean 的远程代理，我们就完全丧失了本地 bean 的好处，不是吗？）

2. 破坏实体 bean 本身的标识，根据类型来定义标识，而不是对象，这样也不必依赖于底层的数据库来持有标识。这实质上是把实体 bean 变成了匿名对象，任何维护状态的逻辑都交给数据库去完成。（这实际上也是 Ewald 推荐的方法，虽然他针对的是 COM+。）这样服务器就可以尽可能地在集群中创建需要的实体 bean，把最经常被访问的 bean 的负载均衡到整个集群之上。不过，这时进行缓存就不太可能了，因为缓存本身也要具有某种标识，这意味着所有其它匿名对象都必须在这个单一位置进行交换更新，这样我们就又回到了标识瓶颈的问题。更有趣的是，这把实体 bean 变成了一个带有某些（在使用 CMP 的时候）为其专门生成的 SQL 的无状态会话 bean。

现在很容易想到，我们把标识从对象移到数据库，到底有什么好处呢？很简单：我们获得了把逻辑移出数据库，放入一个中间层的能力。这个中间层（如果希望或者必要的话）可以运行在数据库或客户之外的物理层次。它并不能消除数据库内部存在瓶颈的可能性（这很难预测），但是它能在你遇到瓶颈之前，给你带来更大的空间。为了达到吞吐量要求，只需移除足够的系统瓶颈。这个过程就是可扩展系统设计的本质。

[3] 这里我应该更加注意我的用词，因为从技术上讲，实体bean并不需要存储到关系型数据库。（早在 1998 年的一次会议上，我听说几家OODBMS提供商公开反对EJB 1.0 规范中的一句话，“一个实体bean代表了数据库里的一行”）然而，99.5%的实体bean都关联到了RDBMS，所以我将继续使用这个假设。

最后，我们不能从系统中完全消除标识。即使能的话，也会适得其反。毕竟，与我们一起工作的数据元素必须是可标识的，否则我们就无法加以区分。这里的技巧是，我们可以接受什么形式的标识，在哪里使用，使用标识是否需要加锁，是否可以避免使用。做到这一点，你就朝着高可扩展性的架构又前进了一步。

第 6 条：使用“挂钩点”来注入优化、定制或新功能

几乎所有系统，无论其是否采用 J2EE 技术，都提供了许多“挂钩点”（hook point），这样开发者就能“拦截”标准的处理过程，并对余下的操作施加影响。例如，在 servlet 栈中，过滤器就是最明显的“挂钩点”，servlet 开发者可以通过它来阻塞、重定向或修改任何进入的 HTTP 请求或者返回的 HTTP 响应。你也可以使用过滤器实现对 HTTP 体的压缩，对消息内容，或者任何使用 HTTP 消息交换机制的认证/授权请求，把它们全部或部分加密。

RMI 栈中也有这种“挂钩点”，可以把自定义的 socket 工厂插入 RMI 客户/服务器中。只要把标准的 socket 工厂对象替换成能强制使用 SSL/TLS socket，而不是标准 TCP/IP socket 的自定义对象，就能实现安全的 RMI 信息交换。例如，在全部采用 Windows 的网络上，RMI 甚至可以通过使用自定义 socket 工厂，采用更快的自定义 Windows 命名通道，而不是完全采用 TCP/IP 来交换数据。

在这些例子中，“挂钩点”用来添加新的行为，而不需要客户代码做任何改变；附加的处理过程将被添加到标准处理链的某个点上。这种处理可以有多种形式：可以提供某种机会来传递高优先级的（out-of-band）数据，比如在某层创建并传递到下一层的安全上下文；也可以通过查询某个外部资源以加快处理速度（比如对不可变数据的预先缓存，或者是用来优化 SQL 语句的优化器）等等。举个例子，Java 数据对象规范明确提供了 Caller，它具有把特定于提供商的查询传递给 JDO PersistenceManager 的能力，这样客户就能享有提供商附加功能带来的好处（这样作的好处请参阅第 11 项）。

提供“挂钩点”的典型方式是使用拦截模式（Interception pattern） [POSA2, 109]。在刚听

说 J2EE 容器是如何使用拦截模式在你的代码中添加行为之后，你也许会觉得拦截模式非常深奥、神秘、只有实现容器的程序员才能理解这种高深技术，其实这完全是误解。并不是只有 J2EE 才能使用拦截模式。从 JDK 1.3 发布开始，Java 就提供了动态代理，它允许程序员编写实现某个接口的匿名对象，这个对象对实现了相同接口的另一个对象（被代理对象）进行包装，这样就能接收到对被代理对象的调用，从而提供附加行为。

简单地说，只要代理对象和被代理对象都通过相同的接口被引用的，动态代理就允许你编写拦截器来拦截对普通 Java 对象的调用。

例如，我们希望为代码加入记录诊断日志功能，但不希望对所有代码基进行大规模的改动。除了可以使用奇异的代码编织（code-weaving）技术以外，我们还可以在任何对象之前设置一个通过一个 Java 接口暴露其自身的动态代理，用这种技术来构建第一流的概念：

```
public LoggingProxy
  implements InvocationHandler
{
  public static Object
    newLoggingProxyAround(Logger logger, Object obj)
  {
    return Proxy.newProxyInstance(
      obj.getClass().getClassLoader(),
      obj.getClass().getInterfaces(),
      new LoggingProxy(logger, obj));
  }

  private Object proxiedObject;
  private Logger logger;

  private LoggingProxy(Logger l, Object obj)
  {
```

```
    logger = l;
    proxiedObject = obj;
  }

  public Object invoke(Object proxy,
                       Method m, Object[] args)
    throws Throwable
  {
    Object result;
    try
    {
      l.info("Entering method " + m.getName());
      result = m.invoke(proxiedObject, args);
    }
    catch (Exception x)
    {
      l.warn("Unexpected exception " + x + " invoking "
          + m.getName());
      throw x;
    }
    finally
    {
      l.info("Exiting method " + m.getName());
    }
    return result;
  }
}
```

LoggingProxy 可以对任何在其被构造时传入的对象进行包装（使用 newLoggingProxyAround 工厂方法调用），并可以对每一个通过此接口进行的方法调用，记录诊断消息。它的用法如

下：

```
public interface Person
{
  public String getFirstName();
  public void setFirstName(String value);

  public String getLastName();
  public void setLastName(String value);

  public int getAge();
  public void setAge(int value);
}

public class PersonImpl
  implements Person
{
  // other details omitted for brevity

  private String fname;
  private String lname;
  private int age;

  public String getFirstName() { return fname; }
  public void setFirstName(String value) { fname = value; }

  public String getLastName() { return lname; }
  public void setLastName(String value) { lname = value; }

  public int getAge() { return age; }
```

```
  public void setAge(int value) { age = value; }
}

public class PersonManager
{
  public Logger personLogger = ...; // some Logger instance
  public boolean loggingEnabled = false;
    // Set to "true" to turn on logging on objects

  public static Person
    getPerson(String firstName, String lastName)
  {
    Person p = new PersonImpl();
      // get Person from someplace; in a real system, we
      // would obviously do a database lookup or something
      // similar here
      //

    if (loggingEnabled)
      p = (Person)
        LoggingProxy.newLoggingProxyAround(personLogger, p);

    return p;
  }
}
```

解释一下程序的流程。当静态方法PersonManager.getPerson被调用的时候，我们将从数据库（或任何存储位置）中查找到合适的person数据，然后使用这些数据构造一个代表Person实例的域对象。不过我们是通过静态工厂方法[Bloch, Item 1]，而不是构造器来构造该对象的，因为我们希望在客户构造对象的时候，能够加入一个额外的间接步骤。

记住，这里的关键在于，我们希望得到“捆绑于”当前对象内部的诊断日志行为。因为我们不能（或者说不希望）修改真正的字节码，而只是希望在调用开始和结束的时刻，得到诊断日志消息。所以我们对真正的 PersonImpl 对象使用拦截对象进行包装，拦截对象将保持对 PersonImpl 对象的引用，并在记录诊断日志消息之后，把接收到的调用转发给 PersonImpl 对象。

这种方法的优雅之处在于，无论 LoggingProxy 要代理的对象是什么类型，对方法调用的拦截始终能够工作。其关键是，代理要求对被代理对象（这里就是 PersonImpl）的任何调用，都要通过接口，而不是对某个具体类来进行。这就是为什么 PersonManager 返回类型是 Person 而不是 PersonImpl 的原因。更妙的是，Person 的实现与记录日志的行为之间，在编译期间不存在结构上的关系。而且，只要改变 loggingEnabled 标志（它也许是可以即时进行配置的选项，请参阅第 13 项）的值，就能随意打开或关闭日志记录功能。

通过创建构件而不是对象（请参阅第 1 项），我们可以随心所欲地使用拦截器，而不需要改变任何依赖于领域的代码或客户面对的代码。使用接口来定义对客户面对的构件功能；使用公有静态方法提供构建设施，并且把返回的真正对象隐藏在一个（或多个）动态代理对象内部，由后者提供期望的附加功能。这样的额外好处是，因为内部实现可以在不通知客户（他们也不关心）的情况下改变，我们也顺便实现了松耦合（请参阅第 2 项）。

不过，J2EE规范中的“挂钩点”并非都是拦截器[POSA2, 109]来实现。例如，Serialization 规范（请参阅第 71 条）就代表了J2EE系统中的另一类挂钩机制，因为J2EE本身在许多地方都要使用序列化，包括对RMI方法调用参数的解析（marshaling）；对有状态EJB对象（有状态会话bean和实体bean）的钝化机制（虽然规范没有强制要求，但这种情况很常见）。实际上，对于大多数普通的数据传输对象（Data Transfer Object）[Fowler, 401]，它们基本没有较深的继承层次或较多的对象引用，所以缺省的序列化机制显得过于烦琐。因此，对于经常使用的数据传输对象，可以编写一些速度较快的序列化代码来对其进行自定义，以取代标准的序列化行为。不过你应该先进行性能测量，并确定通信栈的瓶颈确实位于解析数据传输对象以后，再这么做（请参阅第 10 项）。

尽管系统中的“挂钩点”并非都被定义成了拦截器，但确实有很多“挂钩点”被定义成了拦截器，这主要是由 J2EE 系统的被动的面向客户性质所决定的（请参阅第 1 项）。例如 Java API for XML RPC (JAX-RPC)规范，它是与 WSDL 相关的 Web 服务的 Java 环境端规范，它将拦截器定义为处理器实现类（Handler-implementing class），用来对在进入的请求和外出的响应之间来回传递的 SOAP 包进行侦测。这样开发者就可以编写出提供商中立的扩展，以处理新的 SOAP 报头（例如 WS-Security 或 WS-Routing 定义的报头），而不用修改客户代码。在我们把 EJB bean 作为 Web 服务端点使用的时候，这一点非常重要，因为你的具体的 J2EE/EJB 容器未必能够支持你希望使用的（更新更好的）WS-规范。

然而，J2EE 规范族中最广为人知的一个，即 EJB 规范，没有提供独立于提供商的拦截机制。这在很大程度上造成了至少一家提供商产品的广泛流行，因为该产品以供应商相关的方式提供了丰富而强大的面向拦截的栈。希望这一点在 EJB 规范的未来版本中能得到改正。

第 7 条：面对故障时要健壮

“出错”是每个开发者必须直面的残酷事实。不仅代码可能存在缺陷，而且数据库也可能耗尽磁盘空间，路由器也可能宕机，电力供应也可能中断（UPS 可能在电力恢复前被耗尽），服务器也可能被攻击，“绝对安全”的操作系统补丁也可能绝对不安全。请注意这里的用词：不是“要”发生故障，而是“可能”发生故障。

在某些情况下，我们会想到防范性编程（defensive programming）这一术语，其思想是：认为调用者不会正确调用你写的方法，而是会破坏你的代码，所以要对每个参数进行断言，并验证每个返回值。我本人并不完全同意这种思想。我认为，多数情况下，构件内部的代码可以认为是“遵守游戏规则”的代码，不过任何从外部对构件的调用（包括使用 servlet/JSP 时的用户输入），在传递参数和进行处理之前，绝对需要进行全面验证。而且只对每个参数进行断言远远不够。

在微观的层次上，这说明你在编写代码时，需要考虑所有可能发生故障的场景：对 EJB 容器的调用抛出了 RemoteException 异常，会发生什么？是 RMI 处理管道因为某种原因不能

满足请求，还是数据库抛出了 SQLException 异常，亦或是传递给会话 bean 的参数超出了许可的边界？这不仅仅是捕获异常，然后给用户“出错了”这类信息这样的问题，而是要有某种合理的故障恢复策略。例如，如果数据库抛出了 SQLException 异常，是因为 SQL 语句格式的问题，还是数据库根本没有响应？如果是前者，那么告诉用户出错了，请重试一遍，也许就够了；如果是后者，那么最好把系统设置成某种紧急模式（panic mode），直到数据库的连接恢复。至少，要通知系统管理员，数据库已经宕机了。

考虑代码中的故障，部分工作就是要正确处理 Java 异常。

第一个问题：众所周知，使用 Java RMI 对导出的远程对象进行远程方法调用，可能会抛出 java.rmi.RemoteException 异常；这也是多年来 RMI 开发者指责 RMI 的原因之一。令人遗憾的是，因为 RemoteException 里面包含大量信息，所以你要是写出如下代码，大部分信息将被完全忽略：

```
try
{
  remoteObject.someRemoteMethod();
}
catch (RemoteException remEx)
{
  System.err.println("Error in calling RMI method!");
    // Beats the heck out of me what went wrong, but that's
    // OK, I logged it to the console, right? Besides, this
    // is just to keep the compiler happy, it's not like a
    // remote call will ever fail or anything...
}
```

当RemoteException异常被抛出时，异常中所包含的所有诊断信息将被完全忽略。问题发生在客户端、服务端、还是两者中间？是因为你试图调用的对象忽然不可访问了吗？是因为存根(stub)和主框架(skeleton)不同步，而导致解析出错吗？还是因为TCP/IP协议栈故障，导致

查询中指定的服务器不存在或不可访问？

例如，ConnectException 异常表明，客户因为某种原因，无法找到服务器上的服务对象（假设底层 TCP/IP 协议栈是好的，即你可以 ping 到服务器），这通常表明服务对象因为崩溃而不可用。请不要把这种情况和 NoSuchObjectException 异常混淆，它通常是因为尽管客户持有存根对象，但是它引用的服务对象已经不存在了。

通过捕获恰当的 RMI 异常，可以向系统管理员或支持人员（顺便提一下，通常就是你）提供丰富的诊断信息，以确定出错的位置。在你下次进行马拉松式的编码过程中，你应该腾出一点时间，仔细查看一下各种 java.rmi.*包里定义的 RMI 异常，然后编写捕获处理程序来正确处理各种 RMI 异常。（记住，如果你使用厂商提供的协议栈，比如 BEA 的 T3 协议，还可能会抛出不同的异常，你也同样需要查看一下这些异常的定义。）

不仅 RMI 在忍受程序员这种“捕获基类异常”的懒散综合症的影响，JDBC 也存在类似问题。通常，开发者仅仅捕获 SQLException，然后在异常处理代码中使用非常一般的处理代码（比如写入日志文件），这就忽略了真正的异常。同 RMI 规范一样，JDBC 规范并没有对所有可能发生的异常进行详细分类，不过它确实提到，鼓励提供商自己定义异常，事实上许多提供商也是这么做的。或者说，SQLException 异常类型为提供商相关的产品错误代码定义了一个处理位置，在这里可以对造成问题的原因得到更详细的信息。此外，SQLException 类型定义了一个“下一个异常（next exception）”属性，这样 SQLException 实例就能在允许的情况下，把异常相互链接起来。你还记得上次把此类信息除了记录到日志文件之外，还报告给其他人的时间吗？考虑到许多系统管理员也对数据库有所了解，那么针对那些“以提供商产品的错误代码所表示的”通用问题，提供专门的错误处理逻辑，也许是个好主意。

对了，顺便问一下，你还记得上次在 Connection、Statement 或 ResultSet 上检查 SQLWarning 实例的时间吗？还是你与其它 99.9%的 JDBC 程序员一样，直接忽略了它们的存在呢？

在 servlet 和 JSP 内部，异常处理策略显得尤为重要，因为当 JSP 发生意外错误时，你一定不希望用户看到堆栈跟踪信息。（这不仅是出于公共关系的原因，而且有安全的因素。一条简单的堆栈跟踪信息就可能泄漏系统架构和总体结构的秘密，攻击者将有效地利用这些信

息。）这意味着，你的每一个 JSP 页面都要定义 errorPage 标识，要么指向一个专门处理此类问题的页面；要么指向一个通用页面，它能提示类似消息：“我们不确定错误发生的原因，但是我们已经记录下来了，并给支持人员发出了电子邮件，我们已经在您的名下自动加入了 5 美元的抵用卷，请原谅。再试一次，好吗？”这也表明，你的每一个顶层 servlet（例如，直接被用户调用的 servlet，与之相对的是相互之间链接起来的 servlet）和过滤器，必须被包装进 try/catch 块，以处理所有可能的异常情况。顺便说一下，应该是捕获 Throwable，而不仅仅是 ServletException 异常。（J2EE 1.4，包括 Servlet 2.4 和 JSP 2.0，提供了某些容器管理的错误处理设施，它们能缓解这类问题；尽可能地去使用它们。）

使用 EJB 也需要仔细考虑异常处理问题。不过这时要考虑的，不是 bean 内部到底出了什么问题，而是客户如何应对 bean 抛出的异常。例如，我们知道如果一个标记为 Required 事务语义的 bean 方法抛出了异常，该事务将被隐式地回滚，但这时 bean 本身发生了什么呢？bean 还可以使用吗？我们还能够在 bean 上调用别的方法，以确定上次调用失败的原因吗？

EJB 规范对应用异常和系统异常进行了明确区分，前者指与领域相关的异常，它们不从 RuntimeException 或 RemoteException 继承；后者代表那些位于应用领域层次之下的错误，例如连接数据库发生的底层错误。规范还定义了 EJBException 这个新的异常类型，它继承自 RuntimeException，作为某种对系统异常的包装而出现。

当异常离开了由 EJB 拦截器产生的调用时，EJB 容器的行为会根据抛出异常的种类而表现出巨大的差异。当一个事务性的方法抛出应用异常时，容器会认为，客户需要知道哪一个域不变式被破坏了，并将异常再抛给用户。在这种情况下，事务自身被孤立了，从而使客户有机会从错误场景中恢复。

系统异常发生的时候，容器不会这么慷慨。因为 bean 本身的状态已变得不可确定（不管怎样，意外的 NullPointerException 异常的确会降低代码的稳定性），容器将事务回滚并将 bean 实例标记为坏的，这就迫使容器将事务完全抛弃。现在，根据调用者是否是“最终调用者”（root caller），客户端将要么看到返回一个 TransactionRolledBackException 或 TransactionRolledBackLocalException 异常，要么是一个 RemoteException 或 EJBException 异常。当我们考虑 EJB 2.1 中表述的 Web Service 端点行为时，情况就更加有趣了，因为现

在我们甚至不能通过通讯电缆返回一个异常对象，客户端必须处理标准 SOAP 错误代码，而这种代码是不确定的，除非你能够知道将返回什么，并有针对性地编写客户端代码。

（顺便说一下，一些 J2EE 书籍提到，你需要强制你的应用异常类型实现 java.io.Serializable 接口，因为这些异常将会在网络中传递；这样做你就不用担心网络传播问题了。Throwable 基类已经实现了 Serializable 接口，一般情况下，它是任何一个可抛出类的祖先。）

但这里不仅要考虑代码，相比其他类型的软件（当然，像飞机和核反应堆上的嵌入式软件控制系统除外），企业应用对正常运行时间有更严格的要求。在架构层次上，我们需要想想在更宏观的层次上出现失败的情况：如果数据库服务器整个瘫痪会发生什么？如果 EJB 容器整个瘫痪了呢？

实际上，J2EE 供应商将会告诉你哪些是你不需要担心的，这是因为他们的“功能丰富的”、“昂贵但却物有所值的”容器将会为你提供各种容错和故障转移能力。我比任何人都更想买下它，但残酷的事实是：供应商没有，也不可能提供一个完整的、照顾到各个方面的错误恢复策略。

拿一个简单的情况来说吧。在一个 Web 应用的 servlet 控制器中（请参阅第 53 项），servlet 需要经常访问 HTTP 请求所提供的参数来决定执行什么操作，甚至是输出该转到哪个 JSP 这样的处理。如果得不到这些参数会发生什么？如果你用会话状态来保存用户的瞬时状态，当用户中途退出并在第二天继续工作的时候会发生什么？

同时，我们也要面对这样的事实：不管我们多么努力，不管我们花了多少时间来试图预测并修复每一种失败情况，我们还是会不断地遇到不可预测的失败。系统缺陷、用例丢失或不合理的用户操作导致不可预测的行为的可能性是存在的。因为我们不能排除这种情况，我们在头脑中要有一个应对它们的计划：如何修复错误数据，如何在产品环境中部署修复工具（理想情况是容器和服务器不需要重启），甚至是如何在你的 servlet 或 EJB 容器中使用供应商的补丁来修复发现的系统缺陷。

然而，一旦你读过第 60 项，你就会意识到发生故障时不仅要健壮，而且要安全。也就是说，

不要偶尔地发出能够被攻击者用来进入系统的信息（像基于 servlet 前端的完全堆栈踪迹）。要确保任何失败都会被报告，并且反复的失败会通过某种方式引起管理员的注意；重复的失败往往是攻击正在发生的信号。

所有（任何）这些问题都可能在你身上发生。“把脑袋埋进沙子里”自欺欺人，或者哪怕有一点“这是系统管理员的麻烦，不关我的事”的想法，都将导致你在慢慢长夜里，为如何处理这些前面提到过的问题而苦战，而且根本没有时间进行试验。除非你真的喜欢没事找事，否则就不要与这种滥用行为扯上关系。

第 8 条：定义性能和可扩展性目标

古人云，千里之行，始于足下。这并不完全准确。千里之行，有起点还要有终点，否则，千里就可能是两千里或三千里，旅程也就成了漫无终点的跋涉。

当需要与客户和分析员交流，以确定给定应用的确切特性和功能时，开发者常常会感到不满。对字段的验证、用例、类结构图，所有这些都要仔细整理，辛勤工作，最后形成一份长达几百页的文档。但是不会有人停下来想一想，“这个功能究竟需要运行多快？”，而且，对于你千辛万苦开发出来的应用，用户最典型的抱怨往往是：“太慢了”。这种抱怨立刻会得到回应，“哦，去买更快的设备。”

这种交流很快会不欢而散。

问题在于，作为开发者要想得到任何程度的成功，我们需要确切知道努力的目标。正如我们要知道功能的具体形式，页面的工作流程，以及 VIP 客户在节假日下一份$150 订单的时候，购物车应该如何处理一样，我们要达到客户的期望，就必须知道程序究竟需要运行多快。

然而，这一点说起来容易做起来难。用户的语言简单，直白：“要更快一点。太慢了。怎么等那么久啊。”尽管我们也会认同这种反应（我们都经历过这种情况，急躁地坐在 Web 浏览器前面，等待 Amazon.com 或者别的网站能有响应。），但对我们却没有多少帮助。我们需要

的是可以定量的标准，而不仅仅是某种直觉。况且，这种直觉常常不能和用户真正的需要相吻合。

糟糕的是，我们要刻画和定量的指标，众所周知难以定义。诸如性能、可扩展性、容量这样的术语，在争夺市场份额的驱动下被混杂在一起，看起来像是诱人的美餐，但你很难识别其本来面目。一个常见的误解是：性能和可扩展性这两项指标，即使意思不完全相同，也相当接近，改善其中一个自然会对另一个又好处。对性能有好处必然会对可扩展性有帮助，果真如此吗？

性能和可扩展性不仅代表着不同的指标，而且对其中一个的改进还有可能会损害到另一个。其它书籍可能会使用不同的术语，但是出于本书的目的，我们按以下方式定义这些术语。

- *性能（performance）*：简单地说，就是系统对“特定用户执行特定逻辑操作”的响应速度。当用户在线订购产品时，如果应用载入页面的时间过长，他/她会感到不满，就可能会到竞争对手的网站上去购买。显然，这时应用的性能不能令人满意。高性能架构的目标就是要降低延迟（系统响应用户请求所需要的时间），以保证可用性，并把用户至上作为应用的目标。
- *可扩展性（scalability）*：如果说性能作为一种“垂直度量”，衡量了系统在单用户场景下的响应速度，那么可扩展性则恰恰相反：它用来衡量系统在用户不断增加时的并发响应。可扩展架构的目标是，随着客户需求量的增加，仅仅通过添置硬件设备而不是重新设计，就能够得到较高的吞吐量（在给定时间段内，系统处理的逻辑操作数。你也可以认为是：操作数/每秒）。
- *响应时间（response time）*：用来度量系统处理一个用户请求需要消耗的时间。也常常用于衡量一个用户界面动作（按一下按钮，选择一个菜单项，打开窗口，等等）的响应时间。响应时间还能以更小的粒度使用，比如度量某个 API 调用或系统动作（比如处理 SQL 调用）所消耗的时间。响应时间由三部分组成：延迟、等待时间和服务时间。
- *延迟（latency）*：到达可以执行业务动作的执行点之前，花在处理开销上的时间量。这个开销与整个系统相关联。延迟高的系统常常失败，因为大量时间用在处理开销上，而没有足够的时间进行真正的工作。
- *等待时间（wait time）*：用于等候服务器的时间，或者，当服务器执行的时候，用于等待资源所开销的时间。

- *服务时间（service time）*：不包括任何等待时间的系统处理请求所需要的时间。
- *吞吐量（throughout）*：这个术语用来衡量给定时间段内完成的工作量，例如每秒处理的事务数量，或每秒处理的字节量。衡量吞吐量通常是业务领域的行为，因为不同系统对企业事务的理解极为不同；更多情况下，我们以比较抽象的方式来使用，比如“通过将竞争最小化，你可以得到更高的吞吐量。”
- *负载（load）*：系统当前的工作量，也就是负载，可以使用多种单位来度量，可以是较粗的粒度（当前使用系统的用户数量），也可以是较细的粒度（使用内存的字节数，每秒使用的 CPU 周期，等等）。通常，负载并不单独使用，它一般用作基准来度量其它统计数据，比如吞吐量或响应时间：“当服务器 CUP 负载不小于 80％，响应时间大约 2 到 3 秒；当负载超过 80％，响应时间大大增加，90％负载的时候，响应时间大约 30 到 40 秒。”
- *并发负载（concurrent load）*：它用来描述任何给定时刻的负载。例如，负载可以是给定系统能够支持的用户总数，这时，并发负载就是该系统在给定时刻能够支持的用户总数。例如，一个 servlet 引擎，只要内存足够，就能够维护 10000 个用户会话（负载），而不至于崩溃，但是它最多只能接受 2000 个并发网络连接（并发负载）。
- *容量（capacity）*：对于给定的机器或系统，在达到能处理的“最大数量”之前，所能处理的负载/并发负载总数。

在这些指标中，对其中一个的改善往往会对别的指标有好处。例如，减少对资源的竞争，不仅能提升应用的伸缩性，而且会降低延迟（这时系统用于等待锁被释放的时间减少了）。不过也可能出现，以降低系统可扩展性为代价，换取系统性能提升的情况，反之亦然。

例如，某个开发者在评价当前系统性能的时候，认为系统运行过慢。据他判断，原因是系统访问数据库的次数过于频繁，解决方法是在应用服务器中缓存数据，这样就能减少网络流量。（他犯的第一个错误是没有先测量性能，就做出了判断；请参阅第 10 项）所以他决定编写一个通用的缓存机制，缓存算法在经过一个月的精心调试之后，开始在应用服务器上缓存数据。画外音：我们都可以高兴的下班了。真的吗？

遗憾的是，这样不行。问题并非如此简单。因为许多客户会突然访问系统，要让数据能够被所有用户使用，缓存要具有某种“全局”性质。并且如果数据发生了变化，在我们使用新数

据更新缓存的时候，就必须让所有用户等待。通过加入缓存机制，开发者也许可以提高性能，但也同时引入了新的竞争点，即“访问缓存以获取数据”，这就损害了可扩展性。在某些极端的例子中（依赖于缓存锁的使用率和同步策略），减少网络流量赢得的时间，还抵不上对缓存加锁造成的损失，这种基于缓存实现的扩展性能不佳；如果系统以后在集群应用服务器上运行的话，缓存就必须复制到集群中的所有节点，这样，系统不仅要承担额外的网络流量（要检查全局缓存）开销造成的延迟，还要承受对全局缓存加锁造成的资源竞争。

即使缓存具有非常高效的锁定机制，比如对缓存进行检查的成本为零（当然，这不可能），还是会影响到可扩展性：缓存会占用机器的内存，这样就会减少机器能够处理的并发客户总量。缓存越大，我们能够支持的客户量就越小；缓存越小，缓存的效果就会变得越差。（似乎缓存大小应该可以实时配置才是合理的；请参阅第 13 项。）

老实说，企业系统中的许多性能问题，其实并不真是性能问题，它们实际上是可扩展性问题。系统表现不佳，是因为它在等待访问某个别人也需要访问的共享资源。如果系统在应用级别记录统计日志，以跟踪系统中用户的动作，并且在向统计日志表里添加记录的时候，开发者直接使用缺省的隔离级别（细节请参阅第 35 项）。这时数据库正在对该表添加读/写锁，这表明，其它“写请求”被隔离等待，直到前面的写操作结束之后，后面的才能开始。如果我们能去除不必要的锁定，数据库引擎向表里添加记录的速度就要快得多，从而可以降低应用的总体延迟。

前面缓存例子中的缺陷就是以性能换取可扩展性。如果系统的目标是，即使在可能会导致系统容量降低的情况下，也要让每个用户获得最短的响应时间，这种缓存策略就很成功。不过，如果系统希望能够支持尽可能多的并发用户，而不考虑对某个特定用户的响应时间[4]，前面的策略就很糟糕，即使达到了目标，也会损害系统的总体能力。所以，最后再盘算一下，这真是成功的优化吗？所有答案都取决于系统的目标，如果这些目标不能完全确定，我们开发者也难以做出决定。

在实践级别上，这意味着几件事：首先，找出那 20％的操作和/或用例（请参阅第 10 项），它们占用了总执行时间的 80％，以确保进行的是最佳优化。寻找一些途径，尽可能地使用

[4] 与很多情况类似，大多数企业系统处于这两种极端之间。

户对计算机执行时间的感觉是很短。请注意最后一句的短语；“用户对计算机执行时间的感觉”是关键，可以使用一些技术或技巧，以减少用户执行两次任务之间的不可利用的时间间隔。比如多线程、直接数据库访问、使用任何能够合理地降低用户对系统性能不舒适感的方法、寻找那些可以在需要的时候提供必要优化的挂钩点（请参阅第 6 项）。或者，在某些情况下，直接构建一个隔离层，把用户隔离到物理上被阻塞的调用之外。例如，开启一个线程，或者向 JMS Queue/Topic 发送一条消息，并在以后进行异步处理。通常，这些方法也许并不能真正提高系统性能，但是只要用户觉得够快，那它们就足够称得上好了。

在任何企业系统中，性能和可扩展性的重要性都显而易见。要像对待其它客户需求那样对待它们：对于“快速”和“可接受的性能”这样的目标，把它们的含义精确记录下来，可以使用数字，或者（更可能）使用某些开发者和用户都能接受的参照物：“系统的用户界面至少要和我们将要替代的系统一样快。”在关心性能的同时，用具体的术语来明确系统的负载和并发负载；24 小时间隔内，预期有多少用户使用系统？1 小时间隔内有多少？等等。在项目开始的时候知道这些细节，你就可以在面临性能和可扩展性之间作取舍时，做出更好的判断。更重要的，这些明确的目标可以告诉你，何时可以停止性能调整，这一点的重要性不亚于知道何时开始性能调整。

第 9 条：只在事务性处理中使用 EJB

四年多以来，几乎整个 Java 和 J2EE 社区都在建议这种方式，但事实上，EJB 并不是 J2EE 的核心技术。尽管有大量介绍 EJB 的文章、书籍和会议，但它在 J2EE 社区中甚至不能算是“主流技术中的最重要的一项”。EJB 只不过是用来构建后端基础设施的又一项技术，应该与其它技术同等对待：即按需使用。

几年来，EJB 已经成为 J2EE 领域里任何问题事实上的解决方案。这要归功于市场宣传，以及它自身确有的一些听起来很“酷”的特点，EJB 已经成为那些“过分夸大且无法兑现”的最新技术之一，比如客户/服务器技术、分布式对象技术、“推送”（push）通讯技术，以及 UML 建模工具。Java 程序员把在项目中使用 EJB 看成是某种炫耀，某种可以在简历里夸耀的经历，或者能得到其他程序员尊敬的资本。正如多年以前人们认同“没有人会因为购买

IBM 的产品而被解雇”，现在也没有人会因为建议使用 EJB 而遭到解雇。毕竟，要是项目的性能不能令人满意，应该去责怪 EJB 的提供商，你也可以去换一个提供商的产品，以得到更好的实现（一般好货可不便宜），以解决所有性能或可扩展性问题。这样就行了，对吗？

问题在于，EJB 的许诺根本不可能兑现。它既不能奇迹般地阻止开发者犯愚蠢的错误，也不能帮你解决诸如 ClassLoader 代理树、额外的网络流量，以及对锁的竞争这样的底层问题。认为 EJB 容器能够奇迹般地提供所有这些功能的想法，只能是痴心妄想。

让我们回过头来重新考虑一下，EJB 应该具有什么功能。具体地说，EJB 应该使分布式系统的构建更简单也更容易，通过创建一系列常用服务，然后通过构件（按照严格的规则和限制编写）来调用这些服务，这就使得开发者可以专注于手头问题的“业务核心”。容器应该解决那些在所有企业软件项目中都会遇到的问题：远程、并发、序列化、对象生存期管理、支持事务等等。再加上提供商许诺这些都会良好地运行：可以按你所需进行扩展；提供容错和故障转移，这样开发者就能够知道系统的一切均在控制之中；你还可以根据期望的性能价格比（有时称为“最佳组合”的购买策略），把任何提供商的容器“即插即用”到你的后端，并且你对它们寄予了厚望（它们能够召之即来，挥之即去吗？！）

要说 EJB 完全没有达到这些目标，也有些夸张。但在某几个方面，它确实没有达到。

在 EJB 规范中申明的目标有这样一条：使分布式系统的创建更简单；这一点并没有做到。如果说有点什么的话，就是它带来了一整套新的复杂性，这个复杂性超过了目前为止 Java 社区所带来的任何一种复杂性。复杂的问题解决起来仍然复杂：想一下使用容器管理的持久化（CMP），在两个实体 bean 之间建立一对多关系是多么“容易”吧。或者看一下取得这些 bean 的实例是多么“容易”：你要使用 EJBQL 语言，与它想替代的 SQL 语言相比，它们非常相似，但是功能却要差很多。

不过，还有更糟糕的，EJB 把简单的事情也搞复杂了。考虑企业系统中最简单的任务之一：查询一个只读的表。不使用 EJB，解决方法是写一个简单的类，然后在初始化的时候从数据库里读取数据，在类的内部持有它们，这样就不必再访问数据库。但要使用 EJB，就没有简单的解决办法。如果我们编写 CMP 实体 bean 来持有只读数据，容器会产生谬误的且不必要

的调用，从数据库取回这些并没有改变的数据（使用事务也一样）。如果我们编写 BMP 实体 bean，容器仍旧会产生谬误的且不必要的对 bean 的 ejbLoad 和 ejbStore 方法的调用，除非我们能捕获这些调用，并且忽略它们。但这样你又回到了写 SQL 的方式。要避免多次存储/载入调用，以及访问数据库的开销，我们可以使用无状态会话 bean，在它的 ejbCreat 方法里把数据读入，在被要求时，才把数据写回。不过，这会导致持有数据的多个副本，每个客户一份（因为会话 bean 绑定于客户标识。请参阅第 5 项）。 因此，J2EE“最佳实践”建议我们，对于只读（或者主要是进行读取的）数据，要么直接访问数据库，要么使用 BMP 实体 bean；使用这两种方式，都会使你回到手工编写 SQL 的问题，而这恰恰是实体 bean 想要避免的。（请参阅第 40 项）

还有更糟的。因为 EJB 构件需要在容器内执行（就像 servlet），对其进行测试就很困难。而一个普通 Java 对象至少可以使用三种方法进行测试：通过 JUnit 这样的单元测试框架；通过一个被嵌入 main 方法；或者通过一个简单的测试程序对构件的基本功能进行演练。但是 EJB 构件只能放在容器里，然后从外部进行测试。在写作本书的时候，还没有任何可以把 EJB 构件放进去测试的“模拟容器”，但 servlet 已经有类似的容器（例如 Cactus）了。这表明，要编写健壮的 EJB 代码，即使不困难，也很麻烦。

造成这些困难部分是因为 EJB 部署本身非常复杂，比其它为 Java 开发的技术要复杂得多。部署的时候，EJB 容器要做大量工作，作为部署者，这其中相当一部分工作就要靠你来完成。这就意味着产品部署（请参阅第 14 项）将更复杂，往常的“编译－测试”周期也变得更长，现在变成了“编译－部署－测试”周期。如果部署过程中还需要人为干预的话，开发者要想真正测试他们的工作就会变得更加困难了。“嗨，如果编译通过了，那就一定没问题，对吧？而且，我讨厌部署这些东西……”

EJB 并没有使企业应用的开发者梦想成真；不过，从更现实的角度来看，它确实是一项有用的技术。EJB 规范的威力在于，它与事务处理的概念高度集成，而这个角色以前是由 Tuxedo 这样的事务处理系统来扮演的。所以，对于事务处理，EJB 非常适合；尤其是，EJB 容器与分布式事务控制器（DTC）的紧密关系，使得在与多个事务资源管理器协同工作时，例如多个数据库、JMS 消息系统，或连接器资源适配器，EJB 成为了目前的最佳选择。

随着 2.0 版本的发布，通过加入消息驱动 bean，EJB 使得异步 JMS 消息处理变得更加容易。尽管编写 JMS 处理主机并不困难（实际上，并不比编写 RMI 对象主机困难），但是使用 EJB 的消息驱动 bean 来处理，尤其是这些 bean 想要在事务语义之下执行的时候，就显得很有意义了。更重要的是，编写消息驱动 bean 可能是 EJB 规范中最简单的部分：编写一个类，实现 MessageDrivenBean 接口，不必考虑“remote”还是“local”接口，仅仅要在部署描述符里配置这个 bean 要监听的 JMS 目标，以及 onMessage 方法（这是你唯一要实现的非 EJB 要求的方法）的事务关联（transactional affinity）。EJB 容器内只部署消息驱动 bean，这与消息机制提供的内在灵活性一起，显得非常有意义，并且能够加强可扩展性。

此外，许多 EJB 提供商为其实现提供了扩展功能，使得宿主 RMI 或 CORBA 对象变得更简单，所以当你打算放弃会话 bean、消息驱动 bean 或实体 bean 时，先别急着把提供商的 CD 扔出窗外。

再次强调，这里的关键在于，EJB 提供了对特定问题的特定解决方案：提供对事务敏感的中间件能力。更准确地说，EJB 提供了“分布式事务敏感中间件”，如果你并不是在处理“多个事务敏感的资源管理器”，使用 EJB 就很难体现其真正价值。

第 10 条：先测量性能，再进行优化

80/20 规则，正式的名称是 Pareto 规则，它是计算机科学领域里最知名（也是最具普遍意义）的规律之一，其最常见的形式有：20％的代码使用了 80％的系统资源，20％的代码占用了 80％的运行时间，20％的代码占用了 80％的内存，等等。这个规则可以推广到软件开发生命周期的其它方面：比如，80%的需求文档只覆盖了 20％的必要工作。不过我们在这里要关注的，是把 80/20 规则应用到软件执行和软件工程领域。80/20 规则已经无数次被机器、操作系统、应用程序、企业系统反复验证；这决对不是偶然的巧合。在系统性能方面，80/20 规则不仅被广泛接受，而且具有坚固的经验基础。

请注意，这里的具体数字比较灵活，它依赖于你所处的环境：对于你的特定系统来说，可能遵循的是 90/10 或 70/30 的划分，而不是 80/20 的划分。关键不在于具体的数字，而是系统

的总体性能，往往取决于比你开始想象的要少得多的那部分代码。

作为企业系统的开发者，你要想竭尽全力提升系统的性能和可扩展性，80/20 规则是一把双刃剑。一方面，事情将变得简单，你可以忽略大多数“提升性能”的工作：展开循环，使用数组而不是 Collection 类，手工内嵌对方法调用的代码；所有这些优化产生的性能提升可能会微不足道，它们至少有 80％的可能不属于那 20％的核心代码。持有这种观点可能让你觉得奇怪：不进行优化可能会让你的代码更清晰，因为那些难以阅读的代码都是在“提升性能”而不是其它名义下编写出来的。（毕竟，我们不会故意去写那些含混、难以阅读的代码。）

有人认为，把这些优化措施交给虚拟机来执行，要比交给开发者好得多。因为 JVM 有能力进行方法内嵌（method inlining），或者根据使用模式产生完全不同的机器指令。但这就完全依赖于 JVM，如果提供商中立性对你或你所在的组织（请参阅第 11 项）很重要，那就不能依赖于 JVM。

不过，80/20 规则的另一面是，当出现性能问题的时候，我们得去找到那 20％代码。在你考虑优化问题之前，你得去把“80％时间调用的那 20％代码”隔离出来，进行诊断。这种情况有什么问题吗？绝大多数开发者依赖于直觉（也称为“经验”，或者“我上次看的一篇文章”）告诉自己系统瓶颈和低效代码的位置。一个又一个开发者，当他们面临性能（和扩展性）问题的时候，只是漫无目的地在屏幕上翻动代码，然后转过身来以严肃的表情宣布，导致问题的原因是：网络延迟、JVM、Java 编译器缺少更为积极的优化措施、操作系统，或者是某个笨蛋经理决定不在项目中使用 EJB。通常，建议的解决方案不外乎是购买更快的硬件，升级到最新的 JDK，或者完全重写所有代码。

这些言论的荒谬之处在于，绝大多数开发者的直觉是错的。“有经验”的开发者的直觉与盲目的胡乱猜测所导致的结果并无不同。但是，很遗憾，开发者与底层硬件之间的交互，要通过多个复杂的层次，这表明出问题的地方，往往并不是开发者的直觉引导他们相信的地方。那么，要是我们不能相信直觉，我们还能信任什么？

当然是信任系统本身；“让系统告诉你。”就是说，“运行工具对代码性能进行测量。”

然而，在 J2EE 环境里，说起来容易做起来难。Java 性能测量工具还不够成熟，直到 2004 年才真正具有可用性。更糟的是，基于 J2EE 的性能测量非常困难，特别是因为这里要牵涉到多台机器。当然并不是完全不可能，只是有难度：可以在 servlet 容器上运行一个 Java 性能测量工具，在应用服务器（如果有的话）或者其它基于 Java 的中间件上运行另一个，再加上数据库自带的性能测量工具和网络分析工具，把所有的输出综合起来分析，就能够找到瓶颈的位置。随着时间的推移，J2EE 性能测量工具的市场将继续扩大，集成工具的出现将会简化性能测量工作。

同时，别的工具也能提供类似数据，这些工具现在已经可以得到了。

首先，也是可得到的工具中最底层的一个，就是操作系统提供的性能监控工具，比如 Windows 下的性能监视器（PerfMon），或者 Linux 下的/proc 文件系统。当然它们不能告诉你创建的 Java 对象的准确数字，也不能告诉你 JVM 在启动时候强制执行的类加载数量，但是它们能告诉你同样重要的统计信息，比如内存使用量（footprint）；在进程使用虚拟内存机制的时候，发生的页面交换次数；和进程派生出的线程数目。

总是先检查性能测量报告

一个 VisualWorks（基于 Smalltalk）客户非常关心他们的系统的性能。他们注意到，有关“Smalltalk 太慢而不能用于实用系统”的传言可能是真的。他们系统中的一个基本模块执行任何操作时都要好几分钟，而他们希望几秒内就能完成。但是无论他们如何努力，都无法找出原因。

在试图挽救项目的最后一次努力中，他们请来 Dave Leibs 进行为期一天的咨询，他是 ParcPlace Systems 公司的 Smalltalk 性能专家。客户方很快对他们要做的事情进行了概括的介绍。对 Dave 来说，没有任何事可以让 Smalltalk 感到束手无策。于是，他问道：

“性能测量表明，大部分时间都花在哪呢？”
“什么是性能测量？”

于是在接下来的 30 分钟里，Dave 解释了什么是性能测量，并帮助安装了工具。

然后，Dave 针对有问题的代码，运行了性能测量工具。他发现，超过 98％的时间用在了 Date>>printOn:上面。Dave 考虑了一下造成 Date>>printOn:如此慢的原因，但是觉得这未必能引起这么大的问题。于是，他继续向性能书的更高层进发，检查了包含该语句的方法。

然后他就注意到了在循环内部的表达式“`Date today printString`”。于是他试着把该表达式提取出循环，用一个临时变量存储这个值，然后在循环里访问这个临时变量。在进行首先性能测量的十分钟后以后，他再次运行代码。此时，这段客户系统中的关键代码的运行时间不超过 1 秒。

Dave然后对他的客户说，“好了，今天余下的时间我们干什么呢？”[5]

[5] 引用自RoleModel Software公司的Ken Auer和Three Rivers Consulting公司的Kent Beck合著的文章；参考 http://www.rolemodelsoft.com/patterns/lazyopt.htm。

例如使用 PerfMon，通过创建计数器，你不仅能跟踪进程派生的线程数目，还能知道每个线程的上下文切换次数、用户数量、优先级（内核），以及处理器时间。如果系统出于某些原因调用本地代码，而你需要跟踪调用时出现的线程阻塞，这些信息就非常有用。通过 PerfMon 的 Process 计数器，你可以跟踪句柄（句柄基本上相当于 UNIX 中的文件描述符）数量，这样你就能够知道应用服务器打开了多少文件和其它内核对象。这是跟踪任何不能被及时中止的“泄漏”资源的好办法。Process 计数器还能跟踪工作组以及工作组的峰值，这使你能够从操作系统的角度观察进程消耗的内存数量。当你为虚拟机选择启动参数（-Xms 和–Xmx，细节请参阅第 68 项）的时候，这些信息非常有用。Process 计数器里的“页面故障/秒”这个统计量也很值得注意，因为过多的页面故障，说明存在大量虚拟内存交换，这可是性能杀手。（判断是否有“过多”的页面故障总是相对而言，所有在你作出任何有关“过多”还是“正常”的判断之前，你要慎重地收集大量统计数据。）

PerfMon 和其它操作系统级别的性能测量工具，还允许你在很长的时间段内跟踪这些统计信息，可以每隔 x 秒、x 分或 x 小时对系统进行一次快照，这样你就能在很长的运行区间内检查数据。至少，你要在应用开始常规测试的时候，在 QA 或开发服务器上设置这些计数器，这样当应用准备进行产品发布的时候，你就可以清楚地知道什么是应用的“正常”情况。此外，大多数性能测量工具还允许你设置报警器，这样当应用达到或超过某个门槛值时，系统管理员能够得到针对潜在问题的某种提示。例如，你可以配置 PerfMon，当应用服务器进程或 servlet 容器进程开启的线程数目达到某个你预先设定的值时，令 PerfMon 通过网络发送一条消息给系统管理员的机器。这种情况通常表明，要么是线程池配置不对；要么是遭到了“拒绝服务”攻击，应该立刻处理。这些工具最大的好处是，如果你要使用它们，并不需要任何额外的工作。（请参阅第 12 项）

也不要忽略了其它特定于操作系统的工具。尽管 Java 和 JVM 试图把底层操作系统抽象出来，但有时要是能够对 JVM 底层进行的工作有清楚的了解，也是一项强大的调试和诊断工具。例如在邮件列表里经常出现的典型问题，就是“当我从命令行把 servlet 容器作为一个标准进程运行时，一切工作良好，但是当我试图把它作为 NT 服务运行时，就失败了。为什么呢？”这里的问题在于，大多数服务在安装以后，都以较低的权限运行，而不是你所认为的那样（这样很好，请参阅第 60 项），这样它们就不能访问网络共享文件，或者没有足够权限对文件系

统的某些部分进行访问了。Linux 和其它基于 UNIX 的系统中也有类似工具，它们来自于提供商，或者是开源社区，请在准备使用之前，先熟悉它们。

操作系统级的性能测量工具并不是你的唯一选择。要知道，大多数企业级 IT 应用中，另一个主要角色是数据库，搞清楚数据库发生了什么动作，与对操作系统进程的观察相比，同样重要。主流数据库产品都带有某种查询分析工具或类似的性能测量工具，使用它们来观察数据库的执行行为，绝对可以更好地保护自己的利益。当然，我要再次强调“在你的项目依赖于这些工具之前，要掌握工具的使用。”当你的产品服务器因为突发的负载而崩溃时，如果之前已经进行了定期的性能测量，你就能很快找出产生问题的瓶颈所在，并加以处理。

数据库性能测量工具的优点是：它们只关注数据库，并且不对正在运行的任何客户端工具进行假设。对于数据库内正在运行的操作，它们能够提供强大而有用的视图：包括 SQL 注入攻击（请参阅第 61 项），糟糕的 CMP 实体 bean 或 JDO 查询（请参阅第 50 项）都可以被观察到，还包括执行这些语句需要的时间。当你评价应用的执行性能时，它还可以作为一个重要的统计量。例如，如果一个用户请求的执行需要 60 秒用户时间（从单击鼠标到显示结果），那么知道在数据库上花了 5 秒还是 55 秒，将会对你的优化方案产生巨大影响。更重要的是，你能确切知道用户请求在数据库上的执行过程。毕竟也有可能发生这样的情况：任何一个查询都不会造成瓶颈，但是你正在运行 15 个不同的“快速”查询，它们都来自于不同的代码，这就造成了性能降低。如果你还不知道什么是“全表扫描”（full table scan），去问问团队里的其他人；否则，你就该去钻研一本优秀的数据库性能/优化书籍了。

当然，要是能观察到 JVM 内部情况就更好了，不过由于缺少能够应用于不同 JVM，以提供性能测量信息的标准 API，这历来都是个难题。JDK 1.2.2 引入的 Java Virtual Machine Profiler Interface（JVMPI），从来就没有被正式应用于试验以外的情况，不过有两个因素使这个问题得以缓解。首先，JSR 174（Java 监控与管理规范）提供了大量 JMX MBean 和 SNMP MIB 技术来对 JVM 的一些重要特征进行监控和管理，比如线程信息、JIT 编译时间、内存统计等等。第二，尽管目前缺少标准 API，不过已经有了很多商业化性能测量产品，它们基本能够提供类似的功能。不过，对这些商业产品你也要当心，它们有些需要把你的代码放到它们自己“Sun 授权修改”的 JVM 中去执行，这样得出的结果可能就和在生产服务器上得到的大相径庭。例如，Sun 1.4.x JVM 系列中，1.4.1 就比 1.4.0 有一些明显不同和增强。还有一

些性能测量工具通过修改你的代码来得到统计信息（例如，在编译后的类里插入额外的字节码，以得到希望的性能信息），在某些情况下，这就可能掩盖真正的问题。JIT 通常会把只有一行的方法嵌入，但要是方法里被嵌入了别的（测量性能的）代码，JIT 的启发式算法可能就不会对这个方法进行嵌入处理，这就极大地改变了此方法真正的性能表现。

注意，作为 JDK 1.5 的一部分，Sun 引入了 JVMPI 和 JVMDI 的替代品，称为 Java 虚拟机工具接口（JVMTI），并期望它成为性能测量和调试的标准 API。不过，JDK 1.5 计划到 2004 年末才发布，即使到了那时候，如果它遵循的还是像其前继所采用的模式的话，那么它要得到广泛接受可能还得有几年。

Sun 还提供了一个相当底层的性能测量工具，称为 hprof，它可以收集 JVM 整个生命周期中的“调用－执行”统计信息，并记录到文本文件中。虽说总比没有好，但 hprof 的输出相对于 J2EE 应用而言过于庞大，要想从混杂了应用服务器执行信息的输出中（假设应用服务器是用 Java 写的），找出你的应用的执行路径，实在是太困难了。不过，这样还是比“基于经验的推测”或“凭直觉作出的决定”要好。

如果你熟悉 C++和操作系统上的共享对象库，你还可以直接使用 JVMPI API，自己编写“微型性能测量工具”。JVMPI API 本质上使用“回调驱动”的方式，你可以在启动的时候注册某些事件，比如对象的创建或销毁、类的载入或释放、线程的启动、停止和等待，等等。（不过对于心理脆弱的人，不推荐这么做，因为这需要投入大量的时间。不过这使得你可以对感兴趣的数据，得到尽可能准确和详尽的信息。再次提醒，这对时间临界的测量可能会成为一个问题，所以对走这条路要格外小心。）

无论使用何种性能测量工具，都要保证它们能正确地告诉你系统内部的情况。也就是说，要确保工具不会被“系统吞吐的数据”所误导。当进行测量的时候，使系统在正常的负载条件下运行是非常关键的。如果有可能的话，甚至可以人为引入这些条件。比如使用工具来模仿客户的 Web 浏览器会话。如果应用使用的是“厚客户端”（请参阅第 51 项），可以通过实例化直接连接并调用中间件的客户端代理，创建到中间件的模拟会话。记住，工具不能告诉你程序通常如何运行，它只能告诉你这几次程序运行的行为。要确保性能测量工具看到的数据具有代表性，这样它就能告诉你哪一部分执行了 80％的时间，然后你就可以专注于这部分

代码的优化了。一旦你确定了需要优化的代码，就可以使用挂钩点（请参阅第 6 项），以绕过正常的处理程序，并加入必要的优化代码。

第 11 条：认清“提供商中立”的成本

与 Java 把可移植性（还记得“一次编写，到处运行”吗？）作为高优先级对待相似，J2EE 把“提供商中立”，即编译后的 J2EE 应用可以放在任何提供商的 J2EE 容器里运行，作为高优先级来处理。例如，EJB 规范开头提出的 8 个目标中，就有 3 条以“提供商中立”为中心。Servlet 规范，JSP 规范，JDBC，JMS 等都明确提出，把“编译后的应用可以不用修改就在任何兼容的提供商提供的容器内运行”的能力作为设计目标。

问题是，你真的在乎吗？

当然，J2EE 应用通常是作为第三方构件出售，即使是单独的应用，为了最大化潜在的客户市场，也要尽可能做到容易移植和“提供商中立”。例如，要达到独立于数据库的特性，应用程序可以只使用 SQL-92 语法，从而避免使用 Oracle 的专门扩展功能，这时候如果还要限制应用只使用 Oracle 数据库就显然没有意义了。这么做就意味着，当你向客户推销时，没使用 Oracle 的客户不必专门去购买 Oracle 数据库。

不过，问题依然存在。很多情况下，J2EE 应用是作为一个内部系统而被开发的，专门供单个公司（和子公司，以及一些业务伙伴）使用。目标平台：硬件、操作系统、，数据库等，通常在编写第一行代码之前就已经预先确定了。所以，系统通常没有进行跨平台/容器/数据库移植的需求。

更重要的是，标准只是作为一条基线而存在，用来描绘任何平台都应具有的功能。如果标准是唯一要考虑的因素，提供商们如何才能卖出功能几乎相同的产品呢？那你就得靠掷骰子来选择提供商了。然而实际情况是，提供商当然希望你的钱进入他们的口袋，而不是竞争对手的腰包，所以他们会提出购买他们产品（而不是别人的）的理由。在商场上，这个理由就是附加值，它在可移植问题上扮演了重要角色，因为 Sun 正指望这种附加值使 J2EE 能够在太

平洋西北地区的大型软件公司面前继续保持活力。

其想法是：你按照规范编写系统，然后，在部署的时候，你希望根据“价格、可扩展性、集群能力等”来选择提供商的产品。每个提供商都会加入一大堆功能来证明其产品物有所值。你根据自己的需要，选择最符合需求的提供商的产品，从此大家都很开心。因为你根据规范来编写代码，提供商可以在底层进行优化，令你的系统更快、更好、更经济。

不过，提供商不会满足于仅仅在底层进行优化。例如，考虑一下第 17 项：当涉及 EJB 实体 bean 的事务开始的时候，实体 bean 存在于 EJB 容器里，并且要访问数据存储层。此时，数据必须在持有 bean 实例的 EJB 容器，和在磁盘保持 bean 持久状态的数据存储层之间进行同步。这很必要，因为在缺省情况下，规范假定容器不能对真正的数据存储层（数据库）拥有独占式访问。所以，就有必要在事务开始的时候，通过在实体 bean 上调用 ejbLoad 来“刷新”容器对数据的视图。但这个操作是否会导致真正的数据库访问，完全取决于 bean：CMP 会访问数据库；BMP 的行为则不一定，取决于它的实现。

不过，绝大多数 EJB 容器提供商，通过在他们的实现里提供某种附加功能的方式，在这里发现了商机。比如提供某种“独占”标志，这样，程序员就可以在没有其它系统并发访问数据库的时候，跳过不必要的操作。这样 EJB 容器就可以假定，它是唯一通往数据库的通道，然后容器就可以进行各种缓存优化。比如，不用在事务开始的时候，强制刷新实体 bean 的数据。这带来了性能上的提升，许多程序员都用到了这个标志。

不过，这么做的话，立刻就会使 bean 变得不可移植。这与其说是因为 bean 不能通过编译，或者不能在别的容器内运行，不如说是因为 bean 的执行语义环境发生了变化。bean 在容器环境的约束下编码、开发、测试，构成环境的规则指出了编写 bean 要进行的任务，如果规则忽然发生了变化，就可能造成代码冗余，更糟糕的是，可能会引入缺陷。

对于许多开发者来说，支持可移植性是因为，他们觉得“管理层总是一会儿一个想法”，项目开始时使用的工具/平台可能会忽然发生变化。这种情况的原因很多：合并、收购、战略伙伴的变动，甚至可能是提供商销售部门组织的免费高尔夫球活动。真正的原因往往却不那么引人注意：如果你以前利用了平台提供的特殊功能进行编码的话，当你突然被要求更换一

个完全不同的平台时，你将会面临重写代码的问题。

其他开发者可能认为这是在转移话题：在给定的系统内，这种方向性转变永远不可能预先计划，也不可能完全解决。这就好比某公司可能决定把战略伙伴从 Oracle 换成 IBM，它只要选择 Microsoft 的产品，而不是 Oracle 的产品就可以了，但这隐含了从 Java 到.NET 的转换。在这种情况下，任何 J2EE 容器或规范都不能适应这种变化，所以编写符合规范的 servlet 和 EJB 根本没有好处。

除了重写系统，这种平台转换还要做更多的工作。事实上，与其它需要的工作相比，重写代码通常只是最小的工作：移植数据库模型，迁移数据库的数据，对支持人员重新培训，把新平台集成到全面支持的结构之中等等。对于已有的应用，特别是已经使用了一段时间的系统，这些非编码成本很容易高达数百万美元，更不用说购买新平台或硬件的成本了（如果需要转换硬件或操作系统的话）。

如果我们能够从历史教训中学到点什么的话，那就是标准最终不会像现在这么重要。原因很简单：市场将逐渐被三到四家主要提供商占领，它们都会声称符合标准，但又都会和标准有一点点“不同”，并且提供大量的附加功能，这些功能会慢慢成为事实上的标准。

我们可以想到的一个例子就是 ANSI SQL 标准（SQL-89，SQL-92，和 SQL-99）。在标准已经存在，并不断演化的同时，主要数据库（一般是关系型数据库）提供商也在不断地提供对 SQL 的专门扩展，这些扩展不能移植到其它厂商的产品中。每家的产品都支持 SQL 标准的主要功能，但没有一家与 SQL-92 规范完全兼容。不过迄今为止，这对大多数 J2EE 应用来说，还没有造成麻烦。例如，对于 SQL 中的 variant，在 Oracle 和 Microsoft 的产品中有何不同，这个问题已经广为人知，所以，在两个产品之间进行移植的时候，问题就小得多。

最后，决定可移植性重要，还是提供商附加功能重要，完全是一种个人的判断和价值取舍。不过请有意识地做出判断，而不是靠习惯做出选择。

对于需要最大可移植性的系统来说，在部署之前，要避免使用任何提供商附加功能。确保在多种容器下进行测试，也包括参考实现，以发现任何“你没有意识到使用了提供商附件功能”

的地方。例如，提供商实现远程存根对象的产生方式，可能与标准提倡的不同，这样你在客户代码里就只需要一个简单的类型转换，而不必使用规范要求的形式：PortableRemoteObject.narrow。

对于其他人来说，与性能和可扩展性相比，可移植性通常处于次要地位。请尽早决定你的容器提供商，并尽可能使用其提供的附加功能。比如“独占”标志。使用厂商提供的分布式协议，而不是 RMI/IIOP。在数据库内部运行容器，这样实体 bean 就可以直接通过表访问数据，而不需要调用 SQL。或者，如果厂商提供的话，编写实体 bean 直接调用存储过程，而不是让容器生成 SQL 语句。（这又为你提供了“挂钩点”，来对数据访问层进行优化）为容器配置所有的选项，使用本地事务而不是分布式事务，来访问数据库。一句话，充分利用厂商提供的所有优化功能。

第 12 条：内置监控功能

现在项目已经部署完毕，开始工作了，你如何才能知道它工作正常呢？

当然，我也知道，你的代码没有缺陷，也决不会停止工作，你已经处理了每一个可能的异常，覆盖了所有可能的错误条件，你的代码如此强壮，即使是断电也不能令它停止工作。你的 SQL 查询语句是三重冗余的（triply redundant），这样，意外的数据库模型改变或更新将得到平滑地处理和解决。你的 EJB 调用总是放在 try/catch 语句块里面，并有恰当的代码来处理网络故障。你已经阅读并牢记第 7 项，对每个可能出错的情形，代码能正确处理和恢复。你的代码绝对不可能出故障。

然而，你自己并没有编写所有代码，那个大学实习生可不像你那么关注代码质量。

通常，我们知道产品出问题的唯一方式就是接电话，要么是用户自己打来的（要是应用程序在企业内部网使用的话），要么是管理层打来的，他们接到支持人员的电话，支持人员接到用户的电话，而用户正在下一个几百万美元的订单，忽然从浏览器收到了“404”错误，现在不能进行任何工作。看来无论我们能够如何快速地理清问题，或者多么快速地去修复故障，

我们都已经处于被动了。于是，我们总是处于不利位置，因为只有等到故障发生后我们才能解决它。

然而，故障一旦发生，就很难在系统中指出确切的位置。除非是显而易见的原因，比如整个EJB 容器瘫痪了，否则我们或多或少总要检查一下用户屏幕上显示的堆栈踪迹（stack trace），以判断异常发生的地点和原因。我们要感谢 Java 的异常踪迹，但是，如果你的程序里有一段这样的代码：

```
try
{
  // Do work here
}
catch (Throwable t)

{ /* FIX LATER */ }
```

突然间，你根本得不到堆栈跟踪了。还有更糟的情况，因为这个异常的抛出被推迟了，如果其他地方又恰巧发生了错误，那么你可能得到了错误的堆栈跟踪。这一切都要归咎于第一个未得到处理的异常。（当然，我们都知道你决不会这样写代码，正如第 7 项指出的，但要记得，角落里的那个实习生还没有经历过这样的惨痛教训。）

与其总是从不利的地位开始工作，我们不如开始就多花些功夫，使得我们能跟踪系统内部发生的情况。与其被动地依赖最终用户和管理层来告诉我们发生了故障，我们不如在系统内部嵌入一些挂钩（hook），使得从现在开始，我们可以知道系统正在做什么，这样就有望能在任何人之前知道故障的发生。

对于初学者，应该从 TCP/IP 的使用者那里受到启发，设计一个简单测试来确定每个部件都在按照它应该处理的、它在其上处理的和它超越其处理的任何种类的业务处理的方式运行。在 TCP/IP 的环境下，这个工具就是 ping。在诊断网络问题的时候，它是一件利器，因为 ping 可以告诉你两件事情：（1）目标服务器是否还在工作（要是没有响应，就假设它宕机了。）；

（2）从你的机器到服务器，来回要多少时间（这将帮助你判断，也许某些中继网络发生了故障）。

在企业系统里的这个 ping，有时也称为“happy bit”，通常以 JSP 页面或 servlet 的形式出现，当收到请求时，它简单地检查所有需要被访问的服务器及其附属部分是否还在工作。Axis 开源 Web 服务的程序包把这个 JSP 页面命名为 happyaxis.jsp，这个页面的工作是通过试着载入一些类来确认 Axis 管道是否安装了，是否能正确执行等。这样，当基于 Axis 的 Web 服务端出现错误时，可以首先测试 happyaxis.jsp，以确保至少 Axis 工作正常。

在你自己的系统里，可以在每个可行的层次放置必要的“happy bit”。编写一个happysystem.jsp（或类似的）页面，它仅仅验证必要的库能否被载入；不时地执行Class.forName以确保库都可以访问。编写一个“PingBean”——一个无状态会话bean，它只要提供一个方法，ping()，在你的happysystem.jsp页面里面，执行一次JNDI查询，调用PingBean的方法，以得知servlet容器和EJB容器之间的通讯是否正常。在数据库里加入只有一行的表，名为PING_TBL（或者按照其它命名习惯进行命名），这一行里只放置一行众所周知的数据，比如“PONG”[6]，这样你的PingBean和happysystem.jsp都能试着访问数据库，以确定一切是否正常，等等。

这里的关键是，当问题发生的时候，工具箱里有工具可用，你可以用它来开始解决潜在的问题。例如，要是 happysystem.jsp 页面根本不能响应，你就可以判断问题发生在 servlet 容器或者更靠前的部分，也许服务器宕机了，也许是服务器与外部世界的网络连接发生了问题，这些问题都可以使用 ping 请求来发现。要是 happysystem.jsp 返回了，但是显示不能执行针对 "happy table" 的 SQL 查询，那就应该检查数据库了。如果 happysystem.jsp 页面里的每一项都成功能成功运行，那就要仔细检查你的源代码了。

不过，“happy bit”测试仍旧是被动的：它等待你去测试它，并检查是否出错，而不能在出错的时候主动通知你，你必须自己去做。我建议让系统管理员每 15 分钟执行一次“happy bit”测试，以确保系统运行正常，不过这样有两个问题：（1）你的侦错周期现在长达 15 分钟，这就意味着在你发现故障时，他可能已经出现了 14 分 59 秒了；（2）如果你强迫系统管理员

[6] 对“PING”请求的响应，最典型的就是送回“PONG”。

每隔 15 分钟检查一次，他们会对你非常不满。（你自己试一试：不管事情多么简单，每隔 15 分钟一次实在让人厌烦，尤其在你同时还有别的工作的时候。）

不过这种重复性工作由计算机来做就比较理想，所以把这项工作交给计算机，并让它来通知你。建立一个简单的进程，每隔 5 分钟（或者其它你可以接受的时间）向 happysystem.jsp 页面发送一次 HTTP 请求，使用某种解析 HTML 的 API 检查结果，以确保一切正常运行。（开源项目 HttpUnit 非常适合做这项工作。）当这个自动的“心跳”程序发现问题的时候，它可以把“发生错误”的消息通过某种方式通知某人：给邮件列表发一封邮件，发消息给数字寻呼机，打开系统管理员/开发者的 IRC（或者 MSN，AIM，Jabber）连接，在 Win32 事件日志或 UNIX 系统日志里留下记录，触发 SNMP 错误，等等。不管是什么，只要它是某种能够保证有人在监听的通道就行。

目前为止，一切都好。我们的工具可以告诉我们系统底层是否存在问题，并在发生问题时发出警报。不过我们只是热了一下身。我们不仅要在错误发生之前知道，还希望在代码内部放置某种窗口，这样当事情变糟的时候，我们可以进行查看，并对原因作出合理的猜测。

要在代码里得到这样的“窗口”，我们首先需要做一些工作，不过这些工作并不多：我们要创建某种诊断日志，这样就可以向里面写入调试和诊断信息了。Servlet规范提供了一个与ServletContext关联的日志，它独立与J2EE规范，最值得注意的是，EJB容器一般没有这种设备，除非它非常需要。使用servlet日志机制来记录servlet消息，并使用其它机制来记录EJB消息是可能的，但这样我们的消息就在servlet和EJB的来回之中失去了连续性。[7]

已经有许多日志机制的实现可供使用（JDK 1.4 java.util.logging API 和 Log4j 开源项目可能最流行）；具体用哪一个，只是个人喜好问题而不会有任何技术障碍。他们都能够根据记录级别和你要记录日志的子系统来对输出消息进行“分类”。不过，这样的结果是，你需要在代码里使用日志来跟踪系统里发生的事件：

[7] 我们只准备在“servlet容器和EJB容器运行在同一个进程”的时候得到这种连续性，不过因为很大部分的J2EE容器都以这种方式安装，所以作这样的假设很安全。然而，要是servlet容器和EJB容器运行在不同的进程中，（几乎）不可能在没有大量时间开销的情况下，从两个环境中得到单一的诊断信息。

```
// Using JDK 1.4 logging, just by way of example
//
public class LoginServlet extends HttpServlet
{
  static Logger logger =
    Logger.getLogger(LoginServlet.class.getName());

  public void doPost(HttpServletRequest req,
                     HttpServletResponse resp)
    throws IOException
  {
    // Note that we've entered doPost()
    //
    logger.entering("LoginServlet", "doPost");

    // Check our input data
    //
    if (request.getParameter("username") == null ||
        request.getParameter("username").equals(""))
    {
      logger.fatal("Cannot log in without username");
      throw new ServletException("Login failed");
    }

    // and so on

    logger.exiting("LoginServlet", "doPost");
  }
}
```

日志消息粒度的大小可以自己掌握，不过一般的经验是，在不运行调试器的情况下，要调试 servlet 或 EJB（不管是什么）时，日志信息越多越好。

起先，这似乎有些过分，毕竟，这些日志的调用也有开销，不过当你首次不得不在产品服务器上调试错误，而你却不能在你的机器（要么是开发服务器，要么是 QA 服务器）上重现这个问题时，得到所有这些信息的原因就很明显了。相比根据几条简单的信息来猜测故障（猜测的结果往往是不正确的），有了日志里记录的这些信息，要发现它们就要容易得多了。

如果你担心写入所有这些日志消息会降低系统的性能（老实说，你应该这样想），请看看下面的建议。

- *不要做过多的字符串连接*。绝大多数日志实现在判断消息是否需要写入日志（根据某些配置属性）的时候，速度非常快，但是对发生于日志方法被调用之前的字符串连接，它们无能为力。记住，与多数程序语言类似，在 Java 里参数必须在方法调用之前赋值，所以请让日志消息里的参数保持简单；如果这些都不能解决问题，如果你必须将一长串的信息变成字符串来记录到一条日志中，请检测这是否可行（比如，通过 JDK 1.4 API 中的 isLoggable 方法）。因此，如果你使用 toString 来显示一个对象的状态信息，请确保它不会启动一个多方法调用链来走访每个域；要么将 toString 写成对日志记录器友好的，要么创建一个新的对日志记录器友好的方法（比如 toLogString）。
- *将日志写入一些比较快速的存储设备*。很多时候，诊断日志缺省地写入到了文件系统或进程控制台 的 STDOUT 流 或 STDERR 流。JDK 1.4 和 Log4J 的日志实现方法都很灵活，允许客户插入“日志接受器”，所以，如果对你来说标准的文件日志的实现方式花费太多的时间在写磁盘上，那么写一个你自己的实现来接受日志信息并交给一个单独的线程来完成写操作。（这样做时，你必须意识到如果 java 虚拟机不正常关闭的话，你可能会丢失日志信息）。这里的一个技巧就是写一段处理代码在常见的端口上来监听 socket 链接，当日志消息到来时把它们写入每一个相连的客户端，如果没有客户端连接，就把消息丢弃。（Log4J 把这称为 TelnetAppender）
- *将日志写入文件以便于获取*。即便你不是在追踪某个问题，也要把日志写入文件，这样万一突然出现了问题，你也可以回去查找文件中的日志信息来试着找出原因。再者，如果有关文件写入的代码没有你期望的那么快，你可以编写你自己的客户处理代码，在一

个单独的线程中完成文件写入（不要忘了前面的建议）。或者，你可以关闭记录日志到文件的功能直到出现问题，但必须要注意到，问题经常是零星的，无法用合理的方法来预测的，这也就是为什么一开始就用文件记录所有日志是如此有用的原因。

- *让日志记录级别成为一个实时配置的数据项*。换句话说，不要使系统必须重起才能使日志级别的改变生效；否则，在你试图追踪一个问题时，升高和降低日志级别会给你带来不必要的麻烦，这会大大降低它成为诊断工具的机会。实时配置数据的细节可以参阅第13项。
- *保持低级别的日志冗余，除非你在追踪问题*。日志实现通过使用冗余级别来允许你为某条日志消息标记更高的优先级；比如，在早期的 servlet 代码中，实际情况是即便我们进入（然后退出）一个无足轻重的方法，如果在需要时用户名没有被传入 servlet，将同样导致一个致命的错误，因为没有它，servlet 就无法继续运行。一般情况下，一个商业系统希望保持诊断日志的冗余级别在合适的高度，使它恰好得到足够的信息来捕捉错误（有时是警告），这样既能节省处理时间，如果日志存在磁盘上的话，还能同时节省磁盘空间。只有当出现问题的时候才将冗余级别升高，一旦问题解决后立即把级别降回去。

现在，当问题出现时，你可以从日志中查看可能出现的错误和纠正问题的途径。（这也意味着，当异常处理没有其他事情可做时，它只需完成最简单的事情——把异常记录到诊断日志中。就像第 7 项中提到的那样。）

理想的监视状况的另一方面是要持续追踪系统的运行过程，这不仅使我们可以事先找到潜在的错误，而且让我们能够更好地理解性能瓶颈或其他局限可能会引起系统哪些地方的崩溃。然而，构建这样的挂钩显然超出了本书的范围，幸运的是，JCP（Java Community Process）再次为我们带来了福音，这次是 JMX，它是一组应用程序接口，供那些希望被外部管理工具管理的对象来调用。一些供应商，包括部分开源项目，像 JBoss 和 Tomcat，已经内置了对 JMX 的支持。而“J2EE Management” JSR 正式地将 JMX 放入 J2EE 1.4 环境中。（JMX 的更多细节，请参考 Java and JMX [Kreger/Harold/Williamson]）

暂且不论 JMX 技术的细节，JMX APIs 允许你定义被管理的对象，称为 MBean，它可以追踪计数器和 MBean 发现某项性能统计结果超出可接受的范围时所发出的通知。比如，一个 servlet 可能使用“并行访问”MBean 来追踪在 doPost 方法中所执行的并行请求的数目。当

这个数目超出合理值（比如 5，10，20）时，发送一条通知，因为调用在 servlet 中的执行速度并不像预期的那么快。例如，如果 servlet 对数据库的调用占用的时间大大超出预期，可能说明数据库出现了问题（磁盘需要被填写，而数据库的 CPU 来不及响应，等等）。

JMX 最好的地方在于它允许系统管理员不仅可以关注你的代码内的数据统计和性能，而且可以把对上级管理层有用的统计数据集合起来，比如，对比记录失败的、不完全的，和没有正确填写的事务，创建一个 MBean 来追踪系统中所有成功事务的数目就太琐碎了。（经理们就靠这个养活自己）另一个值得追踪的是登录失败的次数，建议你看一下第 58 项；当单个用户或整个系统的登录失败次数超过合理的值时，你就应该发出一个事件通知。

顺便提一下，为了你和你的系统管理员的身心健康，请确保你所使用的监视工具（“happy bit”测试、“心跳”进程、诊断日志，和 JMX）是可以远程访问的，这样你或你友善的系统管理员就不需要站在机器旁监视它。大部分 JMX 供应商为了提供对监视资源的远程访问已经做出了很大的努力；典型的“happy bit”测试是在系统内部完成的，而它已经内置了远程能力（至少有 HTTP）；而诊断日志能够写入 socket 或文件，这两者在近些年已经有了些远程支持。所以，一般来说，使得这些工具可以被远程访问并不困难。这里的关键是避免把诊断日志写入 System.out 和 System.err，因为想远程访问它们可不容易。

如果你对这里所讨论的工作毫无兴趣，请记住我们并不是在讨论如何用统计数据使经理高兴或通过让检测产品问题变得更容易来使系统管理员高兴，尽管这些都是高尚的目标和有价值的工作。把前面所讲的东西都安排妥当还可以使你自己的生活更轻松，因为现在当问题出现时，你将有一整套的工具来帮你检测和诊断它。

事实上，只要你停下来想一想，就会发现这儿的意义比你先前所相信的更大。如果你的系统的后台数据库突然宕机，或因为某种原因而不能正常工作了，当用户面前的网站突然开始到处抛异常，你该责怪谁呢？除非你有工具帮你诊断出问题的根源并证明错不在你，否则，你将难辞其咎。如果你可以将这些工具交给系统管理员，使他们能够代替你做出诊断，那你就高枕无忧了。朋友，这可比前面的所有理由都更有说服力，还等什么？快去准备你的工具吧。

第 13 条：内置管理支持

与大多数用“塑料包装”，就直接出售的商业软件不同，企业应用的一个常见要求就是，它通常需要对其行为或功能进行某种管理控制。比如，管理员常常需要执行各种任务，从常见的“从授权使用系统的用户列表中添加或删除用户”到特定于应用的任务像“在某个用户的工作流程收件箱中取得支出报告，并转发给某人”，到简单系统配置像“左上角应该显示什么图像？”

所有这些问题都已经有潜在的解决方案：系统管理员可以直接访问数据库，做出修改，或者我们可以把已有的图像文件加以替换，并保持文件名不变。问题是，这种管理方式要求系统管理员非常熟悉应用系统的架构，并且要具有足够的技能。例如，他们需要知道足够的 SQL 知识，才能替换或者修改表里的数据。我们假设他们工作优秀，并且决不会在操作这些原始的实时数据时犯错。

事实是，如果应用程序想要在产品环境中长期工作，我们需要给管理员提供一种管理控制台，使他们在做出必要更新时，不需要将它们直接写入数据存储层。具体来说，我们需要考虑在配置和控制这两个方面给管理员某种高级别的控制台。

这里的配置是指某些“我们不希望硬编码的”值，这些值会周期性地在应用里出现。一个典型的例子就是某人的电子邮件地址，当系统故障或者别的什么事件发生的时候，你希望以电子邮件通知他（请参阅第 7 项）。通常，我们不希望这个地址被硬编码到应用里；即使你创建了一个电子邮件的别名（比如，systempanic@yourcompany.com），但是要是公司被收购，域名忽然改变了呢？配置数据应该是不会经常受整个应用之外的东西干扰而产生变化的。（很重要的一点是必须将它和那些随用户的动作而产生变化的数据区分开，否则你就很难区别配置数据和系统中的任何一种其他数据。）

然而，Java 在传统上其配置功能就不太强。从 JDK1.0 开始，Java 应用程序的配置就出现了多种风格，它们各有优劣。

第一种机制就是常见的“属性文件”，它是一个文本文件，里面有供你的 Java 代码读取的名

称－值对，通常使用 java.util.Properties 类进行读取。这种机制的好处是，从程序员的角度非常易于理解，因为“数据存储”的格式只不过是简单的文本文件，它也非常易于修改。使用属性也有一些缺点，最值得注意的是文本数据很容易出错，错误只要来源于错误的键盘输入。此外，因为数据作为文件存储在文件系统，你的代码就需要访问文件系统以获取数据（这在 EJB 里是不允许的）。攻击者要是能够通过文件路径的规范缺陷来发现如何查看文件或者文件能够通过 HTTP 请求访问的话，他们就可能设法从文件里得到有用的信息。一些“高级用户”如果能够得到文件的话，也可能过来“修正”一些“明显”错误的问题。在加上以下一些事实：配置数据的键和值只能使用字符串；从一台生产服务器上远程访问文件可能并不像你想象的那么简单；缺少一个通知机制来保证值改变的时候应用能够重新读入它；以及键是一个简单的标志命名空间，如果你想用层次结构来组织它，你需要自己来管理（连接或解析），但你又突然发现你的机制远不能适应 J2EE 产品应用。哦，我必须承认，这样我们也能蒙混过关，但是我们可以做得更好。

广泛使用的第二种选择是把所有的数据放进数据库。这样可以解决属性机制的不少问题，但是我们又回到了基本的引导问题：类似 JDBC URL，用户标识和密码这样的数据库配置数据，我们要放到哪里才能连接到数据库呢？（理想情况下，你无论如何也不会使用单一用户标识来连接到数据库，但是这确实是有点掩耳盗铃的习惯用法）我们可以把数据库配置数据放进属性文件，然后把其它数据放进数据库，但是我们现在要通过两个不同的位置来存储配置数据，这给我们带来了额外的问题。并且，即使把配置数据存入数据库，我们还是会遇到其它问题，比如，键值的层次关系的组织（关系模型中很难处理），以及更新配置数据（如果系统管理员不熟悉 SQL，或者你对里面的内容不熟悉，就会很难处理）。

与使用关系型数据库类似，我们可以使用一个 JNDI 访问层来存储数据，比如 LDAP 服务器。当然，LDAP 解决了把键以层次结构组织的存储问题，但是我们还是不能解决关系型数据库中遇到的“引导”问题。因为 JNDI 需要某种初始化配置参数以得到 LDAP 服务器的位置，这个数据本身也是我们不愿意硬编码的，所以我们又回到了相同的情况，我们还是需要某种方式来引导对 JNDI 的访问。

有许多 J2EE 规范已经提供了自己对这个问题的解决方案，在部署描述符里以 XML 的形式存储配置选项，这就使得开发者能够通过各种形式的 API 进行访问。比如，Servlet 规范允

许在 web.xml 文件里放置 init-param 标记，这个参数可以在每个 servlet 和每个 ServletContext 上调用 getInitParameter 方法得到。部署以后，许多 servlet 容器可以提供某种管理用的 GUI 在运行的 Web 应用中改变这些参数。然而，这种机制也有自身的问题。重新部署新版本的应用将使用部署描述符里的数据覆盖现有的配置数据，这就迫使系统管理员必须回过头来手工把数值改成所希望的。并且，我们还是要面对键的层次组织的问题，以及部署描述符里只能存储字符串的事实。

使用 XML 的方式，我们能够使用 XML 文件而不是属性文件来存储配置数据。尽管属性文件还存在一些问题，XML 确实已经解决了不少。因为 XML 天生具有层次关系，就不再存在键的层次组织问题了。XPath 使我们不需要手动遍历整棵树就能够得到数据。然而，这依然不够完美。对于那些以其他方式对属性文件产生破坏的问题，我们依然无法解决：我们需要文件系统中的一个文件并访问文件系统，系统管理员将需要编辑 XML 文件本身来做出修改（这就引入了因键盘输入错误而产生错误格式文件的可能性），等等。但我们离目标已经很近了。

从 JDK 1.4 开始，Sun公司引入了新的API，它能够为这个配置的困境提供一个长期的解决方案： java.util.prefs.Preferences（及其相关的）类。尽管它没有正式成为J2EE规范[8] 的一部分，Preferences API除了提供许多其它机制里已有的一些优点外，还有一些新特征：

以最简单方式使用 Preferences API 是非常直观的：

```
// all Preferences classes come from java.util.prefs
//
// Read a configuration value
//
Preferences prefs =
  Preferences.systemNodeForPackage(this.getClass());
```

[8] 有人会指出，J2EE含有对 Java 2 标准版（J2SE）的依赖，所以会接受它所有的相关API，比如 Preferences API。不过，问题在于 Preferences API需要一个特殊的安全许可，即授予RuntimePermission许可，才能访问Preferences数据，而这个许可并没有在 J2EE 1.3 规范里出现。

```
String url = prefs.get("databaseURL", "");
```

这里会发生很多动作。首先，Preferences API 会区分系统数据和用户数据，系统配置数据在整个机器范围内，而用户数据则针对每个用户，并随当前用户的不同而改变。对于多数运行在服务器上的企业应用，我们需要使用系统数据，因为 servlet/EJB 一般不与单个用户关联。其次，preferences 以层次节点的方式组织，类似于 XML Infoset 元素。通过把节点放入不同层次的包中，配置元素可以被划分成不同的可管理块，以便于进行管理。如果想要使工作更简化， Preferences API 具有一些工具方法（类似于上面演示的），可以遍历 Preferences 树以查找与类的包名匹配的 Preferences 节点。一旦我们发现了正确的 Preferences 节点，我们就能够得到与键“databaseURL”关联的值；如果没有发现这个键，就会返回缺省值，这个值将作为方法调用的第二个参数被返回。

这里，我们自然要问 Preferences API 在底层到底使用了什么存储机制，因为不管怎么说，考虑到部署应用时所要做的事，了解存储机制显得非常重要。简单地说，这依赖于具体情况。比如，Preference API 允许不同的存储层次来存储原始数据，称为辅助存储器。为 Preference 节点提供的缺省的后备存储（backing store）依赖于运行代码的系统；拿 Sun Win32 JVM 来说，缺省的后备存储是 Windows 注册表。（不要小看这种用 Windows 注册表来保存重要配置数据的想法。在听说了很多关于注册表的恐怖故事后，我可以负责地说，大部分的问题来源于 COM 程序员，他们在很多地方都滥用注册表。如果你仅用注册表来做它应该做的事——提供一个集中的配置信息仓库，那它将工作得很好。）在 UNIX 系统中，大部分 Java 虚拟机依然使用文本文件，存储在用户的本地目录或整个机器的/etc 或/user/etc 目录，这取决于该系统的 Java 虚拟机和 UNIX 文件系统的具体规范。

除了能够存储字符串之外，Preferences API 还能够存储字节数组，这对于存储可序列化的对象非常方便（关于 java 对象的序列化，请参阅第 71 项）。更重要的是，Preferences API 还允许我们注册事件回调实例（实现 PreferencesChangeListener 接口的类），当发生修改 Preferences 辅助存储器数据的事件时，这些实例将被调用。这就是说如果系统管理员修改了数据，我们能够发现并重新扫描到修改后的值。这就使得当配置数据发生改变时不需要重启服务器来使修改生效，即给与我们实时配置应用程序的能力。

因为 Preferences API 使用不同的辅助存储器，如果你真的不想把数据存储在 Windows 注册表或本地的文件系统中（原因前面已经提到了），那么，无论你喜欢什么样的存储媒介，包括关系型数据库或 LDAP 服务器，你可以总是把它们写入你自己的 Preferences 类中，但这个类必须继承自 AbstractPreferences。

当你非常希望能够用一种简便的方式浏览你的配置数据时，Preferences API 可以将数据导出成 XML 格式（在 Preferences 文档中给出了它的文档类型定义），也可以从 XML 或其他类似的结构中导入数据。这使得在机器间传递这些数据变得十分方便。

Preferences API 还是有一个缺陷，因为作为一个 API，我们没有为它定义非编程式（non-programmatic）的界面（比如一些系统管理员前台）。不通过工具来取得原始数据的话，系统管理员将无法访问它们，我们现在采用的一些东西离理想状况还有一定距离。作为项目开发人员，你需要自己建立用户界面。

乍一看，这根本就是雪上加霜，不是一个非常好的方法。但不管怎样，这样做带来了一些好处。首先，这使得管理员不再需要走访原始数据来更改配置，而这的确减少了我们的很多烦恼，因为现在系统管理员不可能误删关键数据了。（嘿，他们其实不比我们高明多少）然而，更重要的是提供了一个用户界面使我们有机会对输入的数据做一些正确性检查。比如，在填写一个 JDBC URL 时，我们可以试着打开一个与这个 URL 的链接（就像系统出错通知所需的空白 e-mail 地址），甚至可以在用户界面中添加一些帮助，使我们不需要理会那些忘记了“缓冲池最小化”这个词的含义的管理员的电话。

如果这听起来依然很繁重的话，请记住：（1）你依然需要为控制操作编写管理用户界面。（2）一般来说不需要建立太多的配置数据（我们更需要的是清晰和易理解的用户界面）（3）如果这个系统像许多企业系统那样需要长期运行，先前做的一些额外的工作会在系统的整个生命周期中得到多次回报。

就像前面所提到的，除了配置，管理的另一方面需求是控制应用程序或整个系统。比如，在标准的在线电子商务售货应用中，在系统中丢失订单是常有的事。一般来说，当这种情况发生时，数据并没有丢失，只是状态没有设置正确。无论是源自代码中的问题、需求中的问题，

还是什么时候对系统的不恰当修改，结果并没有什么不同。事实是系统管理员正收到一个愤怒的客户打来的电话，因为“系统吃掉了订单”。猜猜下一个倒霉的将会是谁？

系统管理员们管理具体的系统（他们也许与那些负责服务器和网络的人有不同的职责），他们将需要一种非标准的方式来进入系统并操作数据。理想状态下，这种方式只用来查找错误，纠正偶尔损坏的数据等等。但是我们不能忽视出于业务目的，它有时会被用来违反业务规则的事实，因为“CEO 说我们需要这么做，仅此一次，下不为例。”

其他一些可行的方法（一般是那些涉及直接进入数据库或其他数据中来修改原始数据集的方法），也同样会陷入相同的不利状况中。更重要的是，在操作实时数据时，我们需要比操作原始数据更小心，因为一个系统管理员不能像对待代码一样精确地处理数据。仔细想想，你最近一次打开 SQL*Plus（Oracle 数据库控制台）、MySQL 终端、iSQL 或其它数据库控制台去浏览系统并在事务中完成所有工作是什么时候？最近一次执行以 BEGIN TRANSACTION 作为开头，COMMIT 或 ROLLBACK 作为结尾的操作是什么时候？而这是一个很重要的技巧，但很少有数据库管理员、系统管理员，或开发人员在浏览或修复原始表单时用到它。所以，现在如果我们的系统管理员突然发现为修复最近的错误我们需要运行三条 SQL 语句，他们在事务中完成这类操作，并与同时运行的其它事务隔离开的可能性有多大？在负载很重的系统中，直接浏览和操作表单很快就会造成数据的破坏，而这显然不是人们所期望的，即便破坏的“只是”参数选择/配置信息。

这又把我们带回到了配置选项中的那个结论：我们需要想一想系统管理员需要什么样的管理控制特性，并通过某种用户界面控制台来提供这些特性（当然，要由严格的授权检测进行适当的保护。请参阅第 63 项）。这个控制台不在于功能有多强大，越简单越好，也不必花费大量的时间为它制作友好的界面。事实上，有人会说因为我们不想让一般用户在偶然进入控制台后就有能力使用它，所以它根本就不应该被设计成易于使用的。但是为系统管理员提供足够的控制能力便于他们修正可预见的问题，或出现的系统缺陷是非常必要的。

遗憾的是，与配置数据不同，确定系统管理员需要的控制选项要困难得多。几乎所有系统都不同程度地需要用户管理功能——管理员需要一些方法来添加、删除用户，更改用户选项（密码，角色等等）及面向基于角色的授权系统的角色管理（请参阅第 63 项）。我们可以创建一

个简单的 SQL 数据控制台使系统管理员能够对系统使用的数据库执行 SQL 语句，这实质上只是另一扇通向数据库中原始表单结构的后门，这可不是一个好主意。或者，我们可以建立一个高层模型，通过它来提取表单项，并完成操作，然后通过严格控制的用户界面把结果放回去（就像保护适当的事务边界）。管理员用这种方法可以对实时数据做必要的修改而不用担心事务性的泄露。

另一项一般归入管理控制的条目是系统后台数据库的运行报告——除非完成一个或多个报告被要求作为终端用户业务过程的一部分，否则运行周报告或月报告的任务就落到了操作人员的肩上。有些报告的格式是固定不变的，我们可以把对它们操作的 SQL 语句硬编码到代码中，不过，在大多数情况下，一份报告需要一些动态的数据变量或标准选择。你有两个选择：让管理员编写他们自己的特别的 SQL 查询来取得数据，或是将 SQL 隐藏到一个功能足够强大的用户接口中，使得管理员不需要懂 SQL 就可以运行报告。（时常出现的情况是你两者都要，因为无论你的用户接口多强大，总是有人想知道如何基于你的用户接口所没有提供的标准来运行报告。）

顺便说一下，如果你想知道你是否可以不提供这种功能或用户界面的话，请记住：如果你没有这些用户界面，那么修正应用域的问题，比如移除一个被解除雇佣关系的雇员的用户证书和配置，就需要访问原始数据，而这很可能需要你自己来完成这项工作，而不是系统管理员。除此之外，你无论如何都得要写些东西，使得你能够在系统被交付后不久就能够提供该用户界面，因此，你可能要预先去开发它，并要完成开发，并且在此过程中，你可能会听到你的操作人员（我们怎么敢说是高管人员呢？）说“哦”“啊”之类的话。

记住，就像部署和监测一样，注意细节不仅能够改善你和你的系统管理员之间的关系，而且会在今后给你省去许多麻烦。

第 14 条：部署要尽可能简单

在项目生命周期的某个时间点，你必须要交付产品。

尽管这似乎是陈腔滥调，然而有大量包括使用 Java 或别的技术的开发企业级软件的项目团

队，从来都没有真正考虑过把软件安装到生产环境中的过程。如果真有人在这上面花了点时间，通常也不过是“哦，我们只要把文件拷贝到服务器上，然后重启服务器，差不多就这些”。

J2EE 项目，与大多数企业级软件项目类似，部署过程要复杂得多。即使在最简单的情况下，你不仅要把编写的软件的可执行代码部署到相应的容器（servlet 和/或 EJB）中，还得考虑数据库的模型。如果数据库已经以某种形式存在了，而你又不得不对数据库模型做出少许修改（希望是添加，而不是修改或删除），那么事情就变得更复杂了。所有这些都要在代码被添加到产品中的同时，部署到生产环境中。如果你还要在生产机器上安装任何附加的软件，比如 servlet 或 EJB 容器，还得在防火墙上配置通道使得这些机器能够互相通信，在 DMZ 内部重新配置防火墙，以使 HTTP 服务器的新端口能够和另一端的数据库进行通信，事情就这样变得复杂了。

猜猜这些工作由谁来做？

在多数大型企业环境中，这些工作并非由开发人员完成；实际上，在大多数大型数据中心里，开发人员通常不允许接触生产服务器，更不用说安装软件或者改变环境配置了。不过在你觉得不用为处理软件安装这种复杂而琐碎的事情，并准备松一口气之前，请为处理这些事情的人（系统管理员）考虑一下。

先考虑一下这里面的含义。你，作为一个精通J2EE的Java开发者，知道并且理解J2EE容器的功能，以及它的必要性是什么，更重要的是，知道那些令人迷惑的安装和配置选项对你的程序意味着什么。不过，系统管理员，通常没有接受过这种训练（或者是兴趣的原因），也不会明白诸如“无状态会话bean实例池的最多数量”，“数据库连接池的初始数量”是什么意思，更不用说该如何设置它们的值了。除非你非常非常幸运，否则你觉得他们会注意这些问题吗？最可能的是，他们仅仅接受给出的缺省值（这就可能引入潜在的安全漏洞；细节请参阅第 60 项），然后进行下一步操作，以尽快完成任务。[9]

我的观点很简单：除非你真的希望：（a）应用程序根本不能工作；或者（b）因为配置的问

[9] 这里决不是说系统管理员懒惰，或者没有责任心，实际上绝对不是这个意思。绝大多数系统管理员属于工作努力，薪水偏低的员工，他们事务太多，以至于没有足够的时间（或者资源）来完成这项工作。这有点像开发人员，想想吧…

题，程序的表现非常糟糕；或者（c）管理员对你非常不满，因为他们要花费大量的时间，并且查阅几百页的文档，才能把安装工作正确完成。否则，你就要开始认真考虑一下如何部署你的系统，越早越好。

不过，J2EE 规范在这方面没有提供很多帮助；部署被作为提供商自己定义的任务之一，提供商能够在这方面相互竞争，以获取开发者的青睐。结果是，除了对 Web 应用和企业应用存储的直接可以部署的文件格式进行了标准化之外，Servlet 和 EJB 规范（还要其它规范）并没有讨论如何进行部署。比如 Tomcat 开源 servlet 容器至少有三种不同的部署 Web 应用的方法：（1）你可以手工把文件接压缩到 webapps 目录（或者任何目录，如果你配置了 Tomcat 的主配置文件，并指向这个目录的话）；（2）你可以只是把.war 文件放入 webapps 目录，Tomcat 会在下次启动的时候，自己进行解压缩和安装；（3）你可以通过 Tomcat 的“管理器”，这个基于 HTML 的 Web 应用程序，它提供了图形化的用户界面。J2EE 参考实现的 EJB 容器支持两种方法，一个是命令行工具，一个是 GUI 工具。Orion 应用服务器只用一种方法，不过这种方法非常方便：它会创建一个“部署目录”，任何拷贝到此目录的.war 或者.ear 文件都会立刻被部署到容器，而不用服务器重启。

停下来考虑一下，把构件部署到容器内部的动作本身。更确切地说，尤其是在部署 EJB 的时候，请考虑一下部署者在部署步骤中面临的大量选择。J2EE 规范中在有关生产的活动中定义了两个角色，应用部署者和系统管理员，这么做是有原因的（J2EE 的部署可不是一项简单的工作）然而至于其它地方，并非每家公司里都有专门进行 J2EE 应用部署的员工，所以这项工作又被踢回到系统管理员那里。（或者，更糟糕的，由应用开发团队完成。）

为了不使情况更糟，请认清这一点：J2EE 所需要的部署行为只能覆盖 J2EE 本身所覆盖的环境——比如，当部署行为涉及数据库或配置 JMS 管理对象（队列、主题、链接工厂等），或在 servlet 或 EJB 容器中配置 JDBC 数据源实例时，J2EE 规范并不能提供什么帮助。这些行为完全依靠部署和管理人员来完成。

事实是系统管理员的工作是保持产品服务器正常工作越长越好，再想想前面的内容，你就会开始对这一切有一种不祥的感觉了。（想象一下你就是那个可怜的系统管理员，开发主管自己出去庆祝了而希望你来完成所有这些事，你的心就更凉了。）

为了从失败的边缘拯救你的应用，也为了挽救你和系统管理员的关系，一旦你的应用到达了可以考虑测试的阶段，应该立即开始编写部署脚本来自动完成把应用（及其他相关的支持）安装到产品服务器上的任务。

现在，结合第 7 项，想想在部署中失败的可能性——如果部署过程进行到一半时失败了会发生什么？或者更糟，在完成部署后，你意识到你的部门和QA[10]要对一个重大的系统缺陷负责，这时又该怎么办？

在很多情况下，我们可以把部署过程本身看成事务（事务的细节请参阅第四章）。在这种情况下，它不是在数据库中单独执行的事务，而是针对 J2EE 容器和相关系统（比如数据库）的事务。它符合一个 ACID 事务的所有特征：我们希望部署过程是原子的、一致的和持久的，以及隔离的，这意味着（理想情况）在通过提交来完成部署之前，没有人可以看到应用程序的改变。

然而，因为供应商目前还没有为他们的部署 API 提供 JTA 支持，我们不能把 J2EE 应用的部署过程简单地作为标准 JTA 事务来对待。也就是说，我们不能依靠事务管理器为我们处理回调；而必须准备好运行补偿事务（compensating transaction）（它是另一个事务，在现在讨论的情况中，是另一个部署过程）来撤销刚才的工作，以恢复初始状态。换句话说，对于每一个部署脚本，都应该有一个“撤销”部署脚本负责把事情恢复到初始状态。

如果你觉得这么做工作量很大，那么实际情况也确实如此。但是在你开始产生迟疑之前，请记住在你的应用到达产品服务器之前，它将经历一系列的部署过程。当然，我所说的是我们需要部署应用到 QA 服务器，这样 QA 测试人员能够在应用被部署到产品服务器之前测试它——你可以使用各种 QA 发布，你需要把你的部署脚本提交给测试小组进行试验。不管怎么说，你总是希望使这个过程自动化，因为如果你不这样做的话，QA 小组和支持他们的系统管理员会产生厌烦情绪，因为对于每一个要发布的候选应用，他们都要进行部署。你为实现部署过程自动化而为系统管理员所做的任何一件事都会得到回报，不仅能够节省你和 QA 小

[10] 绝不要因为在产品中发现缺陷而责怪QA小组——你同样需要承担责任。

组的大量时间，而且能够改善你和系统管理小组的关系。

第三章 通信

任何时候，如果有人对我说，要是采用他们的信条，就可以给予我与他们相同类型的生活，那么我就会告诉他们这正是我所害怕的。

—— *H.W.Kenton*

进程间通信（IPC）是任何企业系统的基础部分之一。我们已经远离这样的日子：每件事物——用户接口、处理以及数据存储——都存放一台机器，但仍能有效地运转。代码应该具有与在其它机器上运行着的进程进行通信的能力。IPC 就是一种使分布式系统能够很好地被分布的技术。

在整个通信范畴中，各种各样的技术堆积如山。例如，一个数据库如果要访问远程的数据库就需要一个通信层，在该层之上发送 SQL 命令和接收结果。同样，一个分布式事务控制器（例如 EJB 容器或其它的符合 JTA 规范的软件处理）也需要网络通信。即使简单的 HTTP 本身也是一种两端点间的通信方式。不过，对于本章，*通信*是一个软件层，能够让我的客户端代码调用你的服务器端代码，用以执行我们中的一个人或另外一个人发起的任意某种处理——订购一本书、检测一个订购的状态、填写企业的花费报告表，等等。

第 15 项：理解你所做的通信选择

Java 通信 API 可以沿着三个兴趣轴进行划分：传输、格式和通信模式。

每个通信 API 必须要穿过某种通信层，我们称之为*传输层*。而事实上，它们最终都穿越 TCP/IP（或者它的伙伴——比较不经常用到的无连接的 UDP/IP），其中许多都是利用建立在 TCP/IP 之上的高层协议；这种例子之一就是大家都很喜欢的传输协议 HTTP。每种传输方式都有它自己独一无二的方面，因此，仍有必要区分使用原始的 TCP/IP 传输通信 API 和使用 HTTP 传输通信 API 之间的不同——比如，能够查看 HTTP 流量的防火墙产品更喜欢使用 HTTP 传输通信 API 协同工作而非原始的 TCP/IP。

如果有必要，可以通过 JNI 或者 java.nio 之“新 I/O”通道来扩展 Java，从而可以利用其它

类型的传输层。比如，有时利用其它一些与操作系统有关的 IPC 应用编程接口，像管道、命名管道及共享内存，会比 TCP/IP 快一些。而一些信道会提供另外一些行为功能，比如为传统的信道增添加密技术——举例来说，基于 SSL 的 HTTP 会提供一种“安全的”HTTP 信道，我们也很自然地称之为 HTTPS。这种跨越不同信道的能力在通信库的 API 层次上常常会作为挂钩点（请参阅第 6 项）显露出来。

为了在一次传输中传送数据，数据必须以某些传输友好（wire-friendly）的格式包装起来。从技术上，这意味着我们要么保证只有基本类型可以传送（这样就可以很容易地转换成传输友好的格式），否则，必须把一个精致的对象引用网中我们认为是完全成型的对象，转化成某种类型的平面（flat）字节数组，该数组可以反方向重新构建成网络中的对象引用。这就是著名的*编组（marshaling）*，并且，通常是（尽管并非总是）通信管道负责来为你编组数据。数据一旦被编组，通常就作为有效负荷（payload）而被引用，并且它通常包括程序员传送的数据和通信管道所需的额外信息（有时也作为分割数据（framing data）而被引用）。

在 Java 中有两种流行的编组格式：Java 对象序列化和 XML。序列化广受欢迎，因为它提供了将任意可序列化对象转化成 ObjectStream 而不需要以任何方式修改对象所需的全部行为——你需要做的只是实现 Serializable 这个接口，并且很简单。更为重要的是，序列化完全是一种无损处理。能够确保从对象到序列化格式，然后再返回到对象的整个往返过程中数据无损。当编组成 XML 时，就可能产生多种格式，但是越来越多的 XML 编组都是通过简单对象访问协议（SOAP）来实现的，或者用更近一些的 XML Schema，通过各种 schema 类型来定义被编组的数据应该是什么样子而实现的。其它编组形式的使用更是遍及整个计算机业，例如 CORBA 的 IIOP（Inter-Orb Protocol），或者是 RELAX/NG XML 规范，以及一些保留所有权而不公开的格式。

在 TCP 层，所有通过 socket 发送的数据被分割成能够在 IP 网络协议上传送的数据包，并且在目的地能够再重新组合起来变回最初的数据流。不过，随时间的流失，我们已经开始依赖几种抽象——网络通信的若干模式——来帮助我们远离网络通信令人生厌的现实。

出现过两种基本的网络通信方式。一个是现在大家比较熟悉的远程过程调用（RPC）模式，在这种模式中，程序员看起来就像是在执行一个本地的方法或过程调用，让通信管道编组参

数，通过传输将这些参数发送出去，然后阻塞等待，直到接收到一个响应，然后将该响应反编组（unmarshal），接着向调用者返回反编组过的数据（或者抛出反编组过的异常，如果异常就是执行结果的话）。（这就是为什么大多数RPC格式的工具包需要一个后编译步骤，比如RMI的rmic，它可以产生本地类——通常叫做代理（proxy）或存根（stub）——来完成这些工作。）更通用的称呼是*请求—响应通信*，PRC在程序设计人员中广受欢迎是由于对它的概念的熟悉："我只要调用这个方法，剩下的工作都很美妙，一直到它获得服务器上的方法实现。"

不过，基本上 RPC 请求——响应模型只是几种低层次通信模式中的一种，这些通信模式都建立在"发送消息"这个概念之上：在请求——响应模式中，发送者向接受者发送一条消息（请求包含有编组过的参数），然后阻塞等待直到接收者发送回所期望的消息（响应包括编组过的返回值或者错误代码）。其它形式（非 RPC 形式）的请求——响应包括 HTTP 协议本身、SQL 以及 Telnet。

然而，如果以这种方式来看，通信可能不仅仅是为了"发送一条消息，阻塞等待，收到一条消息。"例如，我可能发送一条消息而不阻塞等待，发送一条消息并期望零条或多条消息返回，发送零条或多条消息而不期望任何回应，等等。从本质上说，我们只是在描述发送一条消息的不同方式。

发送消息的概念以及伴随其的内在的灵活性是由消息通信API提供的。尽管它有点难以使用，因为你这部分工作通常需要更多的支持代码，但是消息传送提供了一些基于RPC的请求——响应模型无法提供的性能。有三种这样的额外的通信"模式"：询问——通报（solicit-notify），在这种模式中，一方请求另一方发送通告（例如电子邮件列表的工作原理）；触发及遗忘（fire-and-forget），也称作单向（*one-way*）或异步调用，在这种模式中，一方发送消息而不需要等待一个响应；并且在异步响应中，一方发送请求消息会期望一个回应，但是不会完全阻塞下来等待这个响应，这个响应可以稍后到来。另外，一些消息系统也支持广播消息，一个消息可以抛给多个接收者。

尽管消息本身是一个基础的、低层次的网络概念，但是消息这种思想以及交互的灵活性已经足以证明值得将这种方法提升到网络通信中，达到与 RPC 同样的抽象层次上。因此，我们

可以讨论那些只不过是在更高的抽象层次上提供相同功能的消息系统，或者面向消息的中间件——例如，Java 消息服务，它是一个规范，定义了与这种系统工作的标准 Java API，并且还定义了怎样发送这样的消息：其有效负载是一个可序列化的 Java 对象、简单的字符数组、字符串、一个 Java 映射表（Map）实现对象等等。

因为已经定义了这两种通信（RPC和消息传送）的基本方式，所以我们就可以到更高一级的层次上去开始划分网络通信的不同架构风格了。例如，*客户/服务器* 架构，它有一个进程——制定的服务器——可用来处理一个表示要发起与客户端通信的处理请求。不过，在*peer-to-peer*构架中，通常没有指定的服务器，并且相互之间可以自由地交互通信，每个端点都可以按照所期望的那样发起通信（请参阅第 16 项以获得和发现更多关于*peer-to-peer*构架的知识）。注意，无论是客户/服务器构架，还是*peer-to-peer*构架，从本质上讲，都并非是基于RPC的或是基于消息的；因为这两种架构都可以很容易地使用上述两种方法中的任意一个。

为了展示通信API是怎样通过Java沿着三个兴趣轴：传输、格式和通信模式进行分解以获得支持的——让我们依次研究分析每个API：

- *远程方法调用（RMI）*：遵循 RPC 方法，RMI 使用 Java 对象序列化层来编组数据，缺省情况下，通过原始的 TCP/IP 套接字来传送数据。不过，稍后会介绍 JDK1.2 中 RMI 的一个变体，叫做 RMI/IIOP（读作“IIOP 之上的 RMI”），用来提供与 CORBA 服务器的互操作性，而 CORBA 服务器是使用 IIOP 作为它们的编组层。从本质上讲，它只是带有使用 IIOP（特定于 CORBA 的二进制格式）而不是 Java 对象序列化的编组层的 RMI。对于较新的版本，RMI 开放了它的传输层，允许程序员编写自定义的套接字工厂，更重要的是，如果需要的话，允许 RMI 程序员提供他们自己定义的传输层。此功能最典型的用法是为 RMI 创建 SSL 传输，因此能够给予 RMI 一种加密方法，这也是其它方式所欠缺的。与 EJB2.0 或更早的系统进行通信时，几乎都是通过 RMI，尤其是 RMI/IIOP；另外，一些商家会提供他们自己的 RMI 形式（比如，BEA 提供基于 T3 协议版本的 RMI），但是使用它们会超越 J2EE 规范的边界，因此难于被其它供应商的系统所理解。（然而，注意作为 EJB2.1 规范草案的一部分，没有任何类型的以供应商中立的方式来执行到 EJB 服务器的 SSL 加密形式的 RMI，因此，要是考虑防火墙内的安全问题，你就必须求助于厂商所有权的扩展。）

- *公用对象请求代理程序架构（CORBA）*：CORBA 是 20 世纪 90 年代后期紧盯微软分布式 COM 构架的另外一项技术，Sun 在 JDK1.2 中通过将创建 CORBA 对象请求代理（ORB）作为它的一部分，而使得 CORBA 在 Java 中得以广泛应用。CORBA 像 RMI 一样，也是 RPC 方法的一种，使用 IIOP 作为它的编组层，使用 TCP/IP 作为它的传输层。CORBA 支持其它编组加传输的结合，不过自从引入了 IIOP，所有这些就都没有任何优势了。
- *Servlet*：Servlet 是围绕请求——响应模式而建立的，在这种模式中 HTTP 是最受欢迎的（而且可论证是唯一的）协议。我们在本章所讨论的系统并不像大多数其它的通信系统那样，Servlet 要求程序员通过编写一个 HTTP 请求体的请求数据，并且经由 HttpServletResponse 的 println 方法编写响应数据，去手工编组数据。当然，将 Servlet 和 XML 有效负载格式结合起来，会让你直接进入 Web Service 领域。
- *Java 消息服务（JMS）*：相当明显，JMS API 属于我们的三元分类范畴中：它全都是关于消息的，尽管有数个 JMS 提供商现在提供了使用其它传输和编组层的能力，但是通常它都使用 TCP/IP 传输，以及 Java 对象序列化格式作为编组层，同样地，将 JMS 和 XML 消息有效负载结合起来会将你带入 Web Service。
- *Jini*：尽管不是 J2EE 成员之一，但是它向企业级的 Java 项目提供了一些强大而诱人的功能，最为显著的思想是自我修复网络和服务发现，我们将会在第 16 项中更详细地介绍它们。Jini 以基于发现的查找能力建模，所以它其实只不过是直接在 TCP/IP 之上，以二进制编组为基础的 RPC 通信的一种运用。（很多有价值的服务，像 JavaSpaces，就是建立在 Jini 之上的，但是它只是一种实现细节——例如，IBM 的 TSpaces，和 JavaSpaces 一样，也是另外一种基于空间的工具包，但是 TSpaces 就不需要任何方式的 Jini。）
- *JXTA*：也不是 J2EE 的成员之一，JXTA 项目聚焦于通过操作 TCP/IP 和/或 HTTP 信道，并使用 XML 作为线路传输格式，提供跨平台的 peer-to-peer 功能。每个对等者可以向它发现的对等集合中其它任何一个对等者发送消息——假定没有任何请求——响应格式。稍后的 JXTA 版本，在 JXTA 管道顶端，创建了一个类似套接字的抽象。
- *用于XML的Java API（JAX-RPC）*：JAX-RPC的分类也相当明显：使用XML编组的RPC通信，并且通常使用HTTP传输。注意：因为JAX-RPC和EJB绑定都使用WSDL文档描述它们的端点，正如CORBA使用它自己的接口描述语言（IDL）描述对象接口、RMI 使用Java接口那样，因此作为EJB2.1 规范一部分而提出的Web服务绑定几乎直接属于这个范畴。

- *用于XML消息传递的Java API（JAXM）*：JAXM是一种基于消息的方法，它或多或少地直接建立在SOAP之上，使用XML作为它的编组层，并且通常使用HTTP传输。SOAP和JAXM规范都公开宣布使用简单邮件传输协议（SMTP）作为另一种传输方法，但是它将提供非请求——响应（也就是说，异步的）选择。（顺便提一下，JAXM看起来在为成为一种规范而作最后一搏——来自 Sun 的多个引人注目的声音通过公共论坛和新闻组都评论到 JAXM 看起来并没有服务于任何实际的目的，更重要的是，不论它是否出现在较早版本的草稿中，JAXM 都不是 J2EE1.4 规范的一部分。）

正如你所见到的那样，Java 企业程序员（也就是你）拥有一个在网络中的计算机之间传送数据的广泛的选择集。当然，这会引发一个问题：你到底应该怎样做出决策呢？

毕竟，它们中任何一个都能最终完成这项工作——将数据从机器 A 转移到机器 B——很显然，决策必须根植于某种事物，而不仅仅是“它能够运转吗？”任何一种应该都能够运转。更大的问题是，“哪一种能够更好地运转以满足我的需要？”

很多有关决策制定的处理都在于识别，对你自己来说，就是识别你的通信想要发生的特定上下文。请考虑下面的问题：

- *你需要跨越防火墙的通信吗？*在你明确回答这个问题之前，要记住这并不表示要超越公司网络的范围——许多公司正开始隔离他们的内部网络，实践“深层防御”策略，并在其内部搭建防火墙。因为HTTP是一种清晰明白的、防火墙可以审查和清洗的格式，所以，防火墙倾向于使用HTTP作为主要的传输信道。防火墙实际上把服务器端返回的任何客户驱动的RPC都弄得一团糟（许多观察者风格[GOF, 293]的回调通告方案常常依赖于它），因为防火墙它们不允许在任意TCP/IP端口上有进入的流量，而这正是RMI和CORBA都要使用的。
- *你需要同步通信吗？*如果你的通信模式大多是请求——响应模式，RPC 肯定是首选的通信模式——基于消息的风格允许更高的灵活性，不过通常也需要你做更多的工作。
- *你需要能够和任何平台进行通信吗？包括那些你并不很了解的平台？*Web 服务（也就是在像 HTTP 这样的标准信道之上基于 XML 的有效负载）可以在所有平台之间创建一个“中间地带”，感谢 XML 的无处不在。但是，正如在第 43 项中描述的那样，XML

自己也要花费一定的代价，这个代价可能比你愿意接受的只是两个平台（.NET 和 Java）之间的互操作所招致的代价要大的多。更重要的是，当你在基于 Web 服务的网络模型中创建自己的通信栈时，你必须要确定考虑所有其它的平台。

当然，我们还要讨论其它一些问题。我们甚至可能要建立一个关于通信技术的庞大的决策制定流程图，不过，这样做会侵犯到每个架构和系统设计者想要做出的不同价值判断。重要的是，在确定使用任何技术之前，要仔细考虑你的通信需求。（并且请记住，假定你已经在你的系统中很好地使用了构件，由第 1 项可知，你通常可以修改和/或添加新的通信策略而不会引发太大的麻烦。）

第 16 项：仔细考虑你的查找

小测验时间：术语*位置透明*在它最初的上下文中的精确含义是什么？

在网络出现的早期，很少有这种场景：即便假定你是网络的一部分，你的程序也必须在脱机时运行以利用某些资源。实际上，如果你的组织机构中有不止一台计算机，那么更常见的情况是，只是建立传统的 SneakerNet 系统与设法辛苦努力地实现所有技术细节以创建一个更正规的网络相比，并不显得划算。

但是随着计算机的更加普遍，以及网络的更加流行，网络和操作系统供应商开始意识到：某种*透明*的概念是有必要的——换句话说，我们作为用户希望系统隐藏某些过程和资源在物理上被分布到了多台计算机上这一事实；其实，透明是想在网络之上添加一个抽象层；但是，在接受它之前，必须强制用户知道他们想要的文件在一个名为FILESERVER的机器上的被冠名为PUBSRV03 的共享节点上，现在用户只想知道“它在M：驱动盘上，”其中，基于Windows网络中的M：驱动盘被“映射”到刚才提到的文件服务器的共享节点上。UNIX网络文件系统（NFS）则更进一步，它以某种形式扩展贯穿网络的单根域名，在这种形式中，大多数用户甚至不能说出文件/usr/home/~neward/book.pdf位于什么机器上——或者至少是，他们不能说出在哪台机器上，除非这台机器不知何故停止运转。（分布式系统一个广受欢迎的定义，是由Leslie Lamport所给出的，这个定义是“你知道有一台计算机崩溃了，但是你从来都不

会听到这将会使你的某些工作不能完成。”[Tanenbaum, 7]）

在分布式系统中有多种形式的透明；例如Andrew S. Tanenbaum描述了 8 种形式的透明：访问（隐藏数据表示的不同，以及一个资源如何被访问）、位置（隐藏一个资源被定位在哪里）、迁移（隐藏一个资源可以被移动到另一位置）、重定位（隐藏一个资源可以在运行时移动到另一位置）、复制（隐藏一个资源的复制）、并发（隐藏一个资源可以被多个竞争用户分享）、失败（隐藏一个资源的失败和修复）以及持久性（隐藏一个软件资源是在内存中还是在硬盘上）[Tanenbaum]。在J2EE领域内，所有这些都被隐藏在一个单一的API之后，它旨在通过提供必需的间接层来努力实现这种程度的透明，此API即JNDI。

当第一次走进 J2EE 时，对于一个 Java 新手来说首先会疑问 JNDI 的必要性——这是一个样板，你“只是总需要这样做”，以便持有你想与之一起工作的事物。对大多数 J2EE 从业者来说，这也就是 JNDI 的终点：一旦样板代码显得不适合了，你就会通过在返回的 DataSource、EJBHome 或任何其它东西上调用方法，将这不合时宜的样板抛弃到一边，转移到更好的东西上。

暂且打住。

J2EE 中的 JNDI 有着一个非常实用但被严重低估的目的，并且如果照此直接忽视它，就真正是在冒险——以后会陷入严重的问题。

思考一下，当我们编写如下代码时会发生什么：

```
URL u = new URL("http://www.neward.net/ted/weblog");
u.openConnection();
```

当打开这个 URL 连接时，Java 网络类库进行 DNS 查找搜寻我的主页的 IP 地址，然后打开一个标准的 TCP/IP 连接，通过网络连接到这个 IP 的 80 端口。

问题：为什么不将IP地址直接插入URL中？如果并不是必须的，那为什么还要进行查找呢（可能它被缓存在本地机器上甚至是本地进程中的某处）？为什么不直接命中这个：http://168.150.253.23/ted/weblog（或者是在这本书出货时，我所拥有的任何IP地址）？

可以证明，你一点都不会关心这个 IP 地址，你关心的仅是人们可以读懂的表示。然而对于分布式系统来说，当谈论起你那些普遍是“没头脑的用户”时，这种说法还说得通，但是对于那些大概“没头脑的用户”都不是那些编写跨机器调用的人的分布式系统来说，这就显得缺乏成效了。不，我们不辞辛劳地在每个 TCP/IP 连接上面如此这样般地进行 DNS 查找，主要原因是为了合并几个透明的概念：

- *位置透明*：你不会特别在意资源宿主在哪台机器上。想想看——你只是想得到我的 blog，你不会特别在意我的 blog 是宿主在我房子里的机器上还是在某处的某个 ISP 的架式安装的机器上。然而，我所关心的是很荣幸你访问了我的 blog，因此我是想确定你总能找到它，而不管它是在哪台机器上运行。这把我们引入了下一个话题。
- *移植和重定位透明*：URL 的作用是它是一个重要的抽象层；如果我判断明天有太多的人点击我家里面的这台可怜的服务器，那么我应该“提升”到一台更快的机器或一个更宽的传送管道；通过直接改变我的域的 DNS 项，将其指向在不同地区运行着的另一台机器，我就可以很容易地做到这一点，并且你永远不会知道。实际上，如果负载变得过重，比如说，如果我突然在一天挨了好几次批评（当然，肯定会的！），我会尝试下一种思想……
- *复制透明*：我可以利用一个普通的窍门，使用DNS项作为几台机器之一的“前端”，而不是将它绑定到一台所期望的物理机器上。（你不会真的认为在Google.com 或 Yahoo.com后面只有一台机器，是吧？）而此时，由DNS提供的间接层以及在这种情况下，通过固定的路由器硬件和软件会再次给人留下只有一台机器的印象，而实际上是两台、三台或者数十台之多。

思考一下，如果你缓存了指向我的服务器的 IP 地址，会发生什么情况呢？然后我决定升级（或者降级——也许 Slashdot 们决定他们不再喜欢我了，并且这对于维护 ISP 上昂贵的群集 Web 场地也不再划算）；突然间，你的代码就开始四处崩溃，你不得不修改它们以适应变化。

JNDI 是中间件拼图中重要的一块，那就是查找：*为了避免资源配置和/或复制因修改而导致被意外的损坏，我们在用户和资源之间搭建一个抽象层。*

最典型的场景是数据库。在 JNDI 之前，开发者不得不创建事先知道的 JDBC *connection* 对象，将他们想连接的数据库 URL 传送到 *Driver Manager* 来构建一个 *Connection* 对象。当然，这意味着我们不得不将 URL 传入 *DriverManager.getConnection* 调用，而这个调用会导致我们陷入不舒服的、无法接受的硬编码的场景——我们的代码一旦需要连接另一数据库（例如，执行一个到 QA 环境中的部署，需要与 QA 数据库而不是部署数据库连接），我们不得不返回并修改代码。因此，通过使用 JNDI，我们可以创建一个对程序员友好的 JNDI 名字（"*jdbc/MyDataSource*"）作为间接指针，可以让系统管理员通过应用服务器的管理接口按照他们的希望或需要来修改 JDBC 的 URL。事实上，如果由于某些原因，数据库管理员需要将产品数据库暂时关闭，而且想把所有的数据库流量重新定位到另一数据库实例，那么管理员只要简单地修改一下 JNDI 名字另一端的 URL，那么我们的代码将会很愉快地开始向备份数据库而不是产品数据库发送请求。

你瞧，我们只是向着产生更好的正常运行时间的统计表迈出了重要的一步。但是，只有当你按照规则执行，继续通过抽象层去查找正在被讨论的资源时，这个抽象层才能运转。许多已经写成的 J2EE 书籍都建议你最好缓存你的 JNDI 查找，以便避免通过网络导向资源主机（例如，在查找一个 EJB 的这种情况下，就是 EJB 服务器）的远程往返开销。明白这里某些事情非常重要：如果你将查找结果缓存起来，那么如果想“转变”，至少需要重启服务器。请记住，容错是怎样成为我们之所以首选 J2EE 的最主要因素的？努力想想：像这样无力地默默处理转换与对潜在的、跨越网络的往返成本的节省相比，是否值得。特别是，在很多情况下，应用服务器可以执行一些它自己的智能缓存，从而不需要在网络上往返。

顺便说一下，在 JNDI API 中被奉为圣明的传统的客户机/服务器风格的查找并不是我们唯一可用的查找形式：20 世纪最后十年的前半段时间对被称为“peer-to-peer”或简写为“P2P”的这种新的资源共享形式讨论不断。Napster、 Kazaa 和 GNUtella 都纷纷提出了共享（如果你想直接下载下来的话就是剽窃）像 MP3 和视频这类资源的革命性方式。

但是如果你“啪”的一声打开用于这类系统的代码，然后往下查看，你就会发现一些很有趣的事情：几乎在所有的情形中，系统的“peer-to-peer”本质纯粹在于查找方面。换句话说，实际上 Napster 所做的就是在告诉你：靠近你的就是可以访问的，并且在那里你的客户端和其它方（扮演服务器角色）进行传统的客户/服务器交互，以便发现它们有哪些歌曲，反过来出于同样的目的，你也可以作为它们的服务器。实际上 Napster 所做的就是告诉你：你们两个在一起，可以进行交互了。简言之，Napster 允许两个互相未察觉的不同进程去互相发现：这只不过是查找的另一个名字而已。

（更有趣的是，既然 JNDI API 提供 JavaBean 事件风格的通告，那么用于这种动态发现的 JNDI 提供者并不难于编写；在其它东西进入这个命名上下文的时候，一个潜在的客户端通过使用它，就可以向它的本地 JNDI 提供者的实现请求一个回调——假定是响应一个新服务器的广播式公告——这样也就使 JNDI 成为一个动态的 peer-to-peer 系统。但是至今没有任何提供商将 JNDI 提升到支持这种功能这一点上。）

作为一种查找机制的发现在企业应用里面有很大的发展空间；比如，考虑一下这种思想：一个客户端可以 “注册”一项发现服务，当这个客户端感兴趣的服务器可以被访问时，这个客户端就可以得到一个通告。有两种来自于 Sun 的技术提供这种功能：Jini（使用一种对 RMI 友好的模型）和 JTXA——它构建了一个全新的 API，基于 XML 数据交换来实现（因此提供了一种更加访问透明的与非 Java 资源进行交互的模式）。遗憾的是，在 J2EE 层次上，对于这个问题，对这些技术或者对“发现”这个概念本身的研究都不是很多；因此，要想利用好发现，可能需要走出 J2EE 的范畴。

进行发现的一种最简单办法就是使用 UDP/IP，它同等于 TCP/IP，不过是面向非连接的。一个“客户端”在局域网内将 UDP/IP 请求广播出去，局域网内的所有机器会在给定的 UDP/IP 端口一直进行监听，然后响应该请求，通常会返回一些关于它们自身的详细信息，以赋予客户端能够通过传统的 TCP/IP 连接回来的能力。（顺便说一下，这正是 Jini 的 DiscoveryService 的工作原理。）

因此，一种获得一定数量的集群的廉价方法就是在两台或者更多机器上设置应用服务，然后在每台机器上面设置 UDP/IP 监听器。到了需要中间件层去发现一台服务器以执行处理的时

候，就发送 UDP/IP 广播，捕获响应的第一台服务器。捕获第一台响应机器也能够提供一定的负载均衡能力，因为正在提供服务的服务器在大多数情况下要需要更长的时间才能响应。

有趣的是，将发现作为一种查找机制的思想实际上是一种网络自我修复思想。多少年来，我们一直在与网络并不总是很可靠这个问题做斗争——路由器关闭了、电源没了、网络中继线被街道建设工人切断了[1]——以及其它灾难的发生。通常地，当一个客户端不能与服务器通信时，它只能放弃和中止，因为大多数客户端在没有服务器的情况下不能进行处理任何，大多数客户端应用只在应用开始时，查看一下服务器端是否启动，并且从来不会再次中断检查，而是假设服务器一直都在那里。

但是，在自我修复的网络环境中，客户端要明确地编写代码去处理服务器可能会由于任何原因而“死掉”的思想。因此，在厚客户端（rich-client）应用（请参阅第 51 项）中，当服务器无论由于什么原因而“离线”时，客户端仍然能够显示某种“未连接”图标，以告诉用户出现了问题。如果该应用编写得只是使用本地数据库（请参阅第 44 项），那么客户端只要使用发现就能够确定服务器在什么时候能够恢复正常（要么像 JTXA 那样周期性地检测，要么像 Jini 那样注册一个发现服务），并且立即重新连接。瞧：应用“修复”或“自我修复”所造成的网络中断期可能会中止其它一些应用。但是用户除了成功的操作之外，不会看到任何东西（总之，某种程度上是这样的）。

无论你是否探究过作为一种查找机制的发现，你都不要只是将中间件代码中的查找部分仅仅看作是你必须要回避以获得更加途径的东西。查找是任何中间件系统的成功基础，因为它使位置透明。顺便说一下，注意：位置透明仅仅是想隐藏资源所在的位置，并不是说资源位于远处——有时位置透明会在某些场合被引用，而在这些场合，位置透明并不是应该使用的正确类型的透明（像最常见的错误：“我不管对象在什么地方”）；详见第 18 项。

第 17 项：识别网络访问的代价

对开发者来说，当他们停下来思考一下这个问题的时候，尽管通常不会带来什么惊讶，但是

[1] 这是真实的故事——几年前，San Jose建设人员不小心将一个光缆主干剪掉，导致大多数通向硅谷的互联网连接断开大概半天的时间。

通过网络传输数据确实要花费相当多的时间和精力（按 CPU 周期来测量）。他们通常没有意识到到底多花费了多少，如果不是更多的话，通常大概是多了三个数量级（也就是 1000 倍）的样子。

你自己证明吧。让我们假定来设计一个简单的 API 接口，将会以三种方式来实现它，一种情况是内置（例如：将代码直接置入调用者，而不是进行一个函数调用，来测试 JVM 在进行方法调用方面的效率），一种情况是作为标准的内存对象，另外一种情况是作为 RMI 导出对象。我们将会在同一进程中宿主注册表以保持简洁，甚至在同一机器上运行所有的情况，从而将线路传输时间降低到零——换句话说，我们可以为远程对象创建最佳环境。

这里先列出这个测试后台的驱动；IApi 是我们的简单接口，ApiImpl 是我们的 RMI 导出的实现，以及 Driver 正如它名字暗示的那样，是这个测试后台的驱动。

```
// IApi interface

//
public interface IApi extends java.rmi.Remote {
  public int function(int k, int i)
    throws java.rmi.RemoteException;
}
//
// ApiImpl not shown here for brevity
//
// Driver code
//
public class Driver
{
    public static final int J_LOOP = 7;
    public static final int I_LOOP = 5000000;
    public static void main(String[] args)
    {
        init();
        noFunctionCall();
        functionCall();
        noFunctionCall();
        functionCall();
        rmiCallOnLocalHost();
        System.exit(0);
```

```
}
private static void init()
{
    int k = 0;
    // Warm everything up (ClassLoading, JIT, etc.)
    //
    System.currentTimeMillis();
    for (int i = 0; i < I_LOOP / 1000; ++i)
    {
        for (int j = 0; j < J_LOOP; ++j)
        {
            // Do something to avoid removal
            if (j < i)
            {
                ++k;
            }
            else
            {
                --k;
            }
        }
    }
    System.currentTimeMillis();
    ApiImpl.startServer();
    try
    {
        IApi api = (IApi) java.rmi.Naming.lookup("API");
        for (int i = 0; i < I_LOOP / 100000; ++i)
        {
            api.function(k, i);
        }
    }
    catch (Exception e)
    {
        e.printStackTrace();
        System.exit(1);
    }
}
public static void noFunctionCall()
{
    int k = 0;
    // Now do real timings. This is used to remove all
    // computing time from other tests
    long start = System.currentTimeMillis();
```

```
    for (int i = 0; i < I_LOOP; ++i)
    {
        for (int j = 0; j < J_LOOP; ++j)
        {
            // Do something to avoid removal
            if (j < i)
            {
                ++k;
            }
            else
            {
                --k;
            }
        }
    }
    long end = System.currentTimeMillis();
    displayResults("No Function Call", k, start, end);
}
private static void rmiCallOnLocalHost()
{
    int k = 0;
    try
    {
        IApi api = (IApi) java.rmi.Naming.lookup("API");
        long start = System.currentTimeMillis();
        for (int i = 0; i < I_LOOP; ++i)
        {
            k = api.function(k, i);
        }
        long end = System.currentTimeMillis();
        displayResults("API Call", k, start, end);
    }
    catch (Exception e)
    {
        e.printStackTrace();
        System.exit(1);
    }
}
public static void functionCall()
{
    int k = 0;
    long start = System.currentTimeMillis();
    for (int i = 0; i < I_LOOP; ++i)
    {
```

```
            k = function(k, i);
        }
        long end = System.currentTimeMillis();
        displayResults("Function Call", k, start, end);
    }
    private static int function(int k, int i)
    {
        for (int j = 0; j < J_LOOP; ++j)
        {
            // Do something to avoid removal
            if (j < i)
            {
                ++k;
            }
            else
            {
                --k;
            }
        }
        return k;
    }
    private static void displayResults(String desc, int k,
                                long start, long end)
    {
        System.out.println("k = " + k + ", " + desc + ": " +
                           (end - start) + "ms");
    }
}
```

在我的笔记本电脑上运行时，这些代码返回如下结果：

```
C:\Prg\Projects\Publications\Books\EEJ\code>java Driver
k = 34999944, No Function Call: 340ms
k = 34999944, Function Call: 271ms
k = 34999944, No Function Call: 190ms
k = 34999944, Function Call: 351ms
k = 34999944, API Call: 437559ms
```

正如你看到的那样，No Function Call 和 Function Call 与远程 API Call 之间存在很明显的差异。更糟的是，这个实验甚至不涉及任何形式的线路。我们只是测量了编组以及将其移动到 TCP/IP 栈中的回转通道（loopback）适配器的代价。

（注意，具有讽刺意味的是：第二次执行 Function Call 所花费的时间要更长一点，而内置的 No Function Call 却更短一点，这可能是由于 JIT hot-spot 内置的缘故吧。说实话，我无法解释为什么 Function Call 会更差一点；可能对函数调用代码进行 JIT 编译的害处多于好处吧，可能是由于解释成本地语言的转换边界的缘故吧，幸运地是，多达 3500 万的调用次数才产生 0.08 秒的差别，因此可能也不值得担心。）

思考一下，将这种情况放置到这样一个场景中：你很饿，想要三明治。你走到冰箱旁，发现完全没有制作三明治的任何材料——没有面包、没有芥末、没有火腿、没有西红柿，没有任何东西。因此，你打算前往食品杂货店（这恰巧就是我们的基线“本地方法调用”）获取材料。花费 20 分钟找到钥匙，拿到现金，赶到商店，停好车，进去，购物，出来返回，打开车子，开车回家，打开材料，铺放到桌子上，准备做。

现在设想不需要沿街去找食品杂货店，你听说有一个相当好的熟食店，你朋友经常去（“远程方法调用”）。如果你花费 20 分钟到本地的熟食店，而且一个远程调用会花费三个数量级（或更多）倍数的时间去执行，那么你在运送中所花费的时间就和带你到冥王星旅游的时间差不多相同啦。（那家最好是非常非常好的熟食店。）如果熟食店一次只卖一项物品给我们，或者我们忘记了配料单，此时我们必须不断地返回，一次一个地获得每一种配料（请参阅第 18 项），那么愿上帝保佑我们吧。

所有的开销来自何方？一系列事情，正如你所经历的那样——将参数从内存中的表示形式（实际上，因为我们跨越的边界全是像 int 这样的同构类型，所以不需要任何处理就可以发送，如果这样的话，这种表示形式确实是最小化的了。）编组成线路友好的形式，接着将编组过的数据压入 TCP/IP 栈成为 TCP/IP 适配器的本地主机回转循环的一部分（接着，该适配器马上将数据回退出栈），而且对于另一边的响应方来说，反之亦然。在这里，损失的时间不是很明显——事实上在群聚网络中，网络传送数据花费的时间极其可怕。

连同“身份标识引起竞争”（identity breeds contention）问题（请参阅第 5 项）一起，它们可能要毁掉大多数分布式对象系统：好的面向对象设计鼓励微小的、聚焦在做好一件事情的原子对象，而且通过方法调用（在分布式对象系统中通常是指网络访问）把其它任何请求推送给其它对象。思考一下，代码的执行性能会发生什么情况，如果你结合Visitor模式[GOF, 331]

和远程方法调用——即沿传输线路传送的对象间的每个方法调用，以及每次往返至少需要的两个或更多方法调用，哦！你将会看到只是远程对象间来来回回调用而用到时间量就已经是多么的惊人。

顺便说一下，这不再仅仅是 RPC 的事了——任何涉及到跨机器（或者甚至仅仅是跨进程）传送数据的技术都必须要经历 RPC 工具包所做的相同类型的问题。JDBC、JMS、HTTP，它们全部要花费一定时间将对象转换成 1 和 0，并且这些 1 和 0 仍受下面通道速度的限制（或缺乏速度）。由于这些原因，使每次到冥王星的旅行都变得有价值了。只要你能够，你就要确保批量传送数据（请参阅第 23 项），考虑考虑将数据移动到更接近处理器的地方，或者反之亦然（请参阅第 4 项），甚至考虑花费时间编写更灵巧的 RPC 代理（请参阅第 24 项）。你可以做任何事来最小化线路上的时间开销，只要它大体上不影响中间件的总体目标，将来就会以更好的性能来回报你所做的一切。

条目 18：优选上下文完整的（context-complete）通信风格

上下文完整的（context-complete）通信能够提供上下文不完整（context-incomplete）的调用所没有的很多优点。上下文（context）这个概念被证明是整个分布式设计中是一个主要组成部分，因此有必要先介绍一下通信中的上下文这个概念。

想想一下你和一个朋友通过电话谈话的场景[2]。

Stu: 嗨，今天一起吃午饭吧？

Justin: 好的。什么时间？

Stu: 正中午怎么样？

Justin: 我可能不行，11 点怎么样？

Stu: 好的，那什么地方？

Justin: 市区的皮萨店吧？

Stu: 好的，回见。

[2] 即兴举的这个例子是一个和*松耦合（Loosely Coupled）*中的例子相似： *Web服务缺少的部分（The Missing Pieces of Web Services）* [Kaye, 119–120]。

当这样看时，很容易理解这个场景，因为谈话的内容一目了然。

不过考虑一下，如果你现在从谈话中间开始，大概第五行吧。你知道他们在市区的皮萨店见面，但是不知道是什么时间，也不知道是哪一天。你可能对此次午餐或晚餐聚会只是有个模糊概念，并且，他们可能只是选择一个远足旅行的集合地点。简而言之，谈话上下文建立了关于谈话结果的重要部分，并且，除非你在一开始就在这里，否则你是无法看到谈话结果。更糟的是，万一两个朋友有午餐聚会的惯例（假设 Stu 和 Justin 有这种惯例），那么谈话可能变得更加神秘难懂：

Stu：嗨，今天一块吃午餐吧？

Justin: 好啊，什么时间？

Stu: 老时间怎么样？

Justin: 我可能不行，11 点怎么样？

Stu: 好的，那什么地方？

Justin: 昨天老地方怎么样？

Stu: 好的，回见。

现在，这个上下文延展到了更长的时间窗口，并且除非你刚好知道他们的吃饭习惯，否则完全没有办法明白他们的谈话内容。

很多分布式对象设计都采用了同样的方法，通常都显得没有什么意义：

```
Context rootContext = new InitialContext();
CartoonCharacter cc =
  (CartoonCharacter)rootContext.lookup(PATH);
cc.setFirstName("Fred");
cc.setLastName("Flintstone");
cc.setHometown("Bedrock");
String catchphrase = cc.utterCatchPhrase();
  // Returns this character's catch phrase as a string
```

当我们开始调用utterCatchPhrase时，上下文已经由先前的一系列调用建立起来了，也就是，通过设置对象实例本身的状态隐含地建立起调用的上下文。

如果这些上下文丢失或者是弄乱了，将会发生什么事情呢？

很多分布式系统构架犯的一个主要错误就是没有考虑到远程对象甚至在很短的时间期间内都有可能失败的各个方面。让我们展开一个简单的场景看看：调用setLastName抛出RemoteException异常，我们将它捕获。现在怎么样了呢？是否整个对象都不再正常了？那个调用是唯一失败的事物吗？我们还能否再次调用它的setLastName并能继续下去呢？

然而更糟的是，想想那个坐在角落里的大学实习生编写的代码；尽管我们都知道你从来不会编写出下面这样的代码，但是又有什么能够阻止他这么做呢：

```
Context rootContext = new InitialContext();
CartoonCharacter cc =
  (CartoonCharacter)rootContext.lookup(PATH);
try { cc.setFirstName("Fred"); } catch (Exception x) { }
try { cc.setLastName("Flintstone"); } catch (Exception x) { }
try { cc.setHometown("Bedrock"); } catch (Exception x) { }
String catchphrase = cc.utterCatchPhrase();
  // Returns this character's catch phrase as a string
```

我知道，你很难想象有人真的会这样写代码并且将它发布到产品系统中，但这还是发生了，而且CartoonCharacter实现的作者需要防御性地编码以便强壮地面对错误（请参阅第 7 项）。

那么，这也表明utterCatchPhrase方法可能应该进行检测，以确保在继续进行之前，它的内部状态良好：

```
public class CartoonCharacterImpl extends UnicastRemoteObject
  implements CartoonCharacter
{
  private String firstName = null;
  private String lastName = null;
  private String hometown = null;
  private boolean goodness = false;
  public String utterCatchPhrase()
  {
    if (this.goodness)
    {
```

```
      // Do database lookup to get character's catch phrase;
      // only do this if we have good firstName, lastName,
      // and hometown values
      //
    }
    else
      throw new IllegalArgumentException(
        "Invalid arguments!");
  }
}
```

现在我们进入一个有趣的难题中：我们怎样设置或者取消这个goodness标识呢？

很明显，最简单的方法是测试所有三个私有变量都不是null；也就是说，在每个修改方法（mutator）结尾处放置一个测试，如下所示：

```
if (firstName != null && lastName != null &&
    hometown != null)
  goodness = true;
else
  goodness = false;
```

注意，else 语句是此测试不可缺少的一部分；如果没有这条语句，像下面的简单代码就有可能很容易地搞乱 goodness 的状态：

```
cc.setFirstName("Fred");
cc.setLastName("Flintstone");
cc.setHometown("Bedrock");
  // At this point, goodness turns 'true'
cc.setFirstName(null);
  // Without else clause, goodness remains 'true'
```

实际上，事情远比这复杂，即使有 else 语句——当下面的代码运行时会发生什么？

```
cc.setFirstName("Fred");
cc.setLastName("Flintstone");
cc.setHometown("Bedrock");
  // Goodness turns 'true'
cc.setFirstName("Thundarr");
```

```
// Is it still good?
```

当我们调用utterCatchPhrase时会发生什么？显然，“Thundarr Flintstone”并不是一个完全有效的组合，不过由于初始化是以分段的形式进行的，所以对语言来说就没办法捕获和强制这一点了。

当仅仅需要状态的一部分而且剩余部分保持是可选的时候，事情真的会变得纷繁复杂。那些岁数比较大，能够回忆起卡通漫画的人一定会记得 Thundarr 并不像 Flintstones，他没有姓氏，也没有家乡；并且他的名字也足够标识他，因此下面的代码是充分的：

```
cc.setFirstName("Thundarr");
String catchphrase = cc.utterCatchPhrase();
```

同时，He-Man 没有姓，但是他确实有一个籍贯（他来自于 Eternia 星球，噢，确实很远啊，但是，围绕卡通人物创建的企业级 Java 系统也是这样，因此请容忍我）：

```
cc.setFirstName("He-Man");
cc.setHometown("Eternia");
String catchphrase = cc.utterCatchPhrase();
```

考虑一下，你应该怎样根据这些需求来设置goodness的状态，然后想想在这个时间序列场景中会发生什么：

```
cc.setFirstName("Fred");
cc.setLastName("Flintstone");
cc.setHometown("Bedrock");
String catchphrase1 = cc.utterCatchPhrase();
cc.setFirstName("He-Man");
cc.setHometown("Eternia");
String catchphrase2 = cc.utterCatchPhrase();
cc.setFirstName("Thundarr");
String catchphrase3 = cc.utterCatchPhrase();
```

在执行第三个调用时，我们会获得前两个调用剩余的初始化数据，因为程序员没有重新完成初始化以清除原有的数据——很容易就会犯这样的错误，尤其是在CartoonCharacter习惯于只需要名字，而姓和籍贯则在稍后再添加的时候。

按照正规术语来说，每个对象的创建需要两个步骤：实例化，在这期间，对象本身首次从无到有；初始化，在这期间给对象赋予所需的初始状态以便开始实现有意义的工作。通常地，是通过定义需要某些参数的构造器而不提供缺省构造器来实现。如果传送的参数非法，我们可以抛出异常并有效地使对象本身无效。不过，在远程对象的情况下，实例化完成的时间通常与初始化完成的时间完全不同，因此，我们也就失去在实例化期间验证初始化参数有效性的机会。

在经典的面向对象中，我们从来都不会想去使用一个还没有为使用做好准备的对象（例如，一个还未正确初始化的对象），因此，编程语言通常通过构造器来竭尽全力地确保初始化与实例化同时发生。如果参数无效，构造器可以抛出异常然后有效地终止对象。如果所需的参数根本没有传递进去，编译器就会抱怨。简而言之，我们有语言的支持来确保：传递给对象它所需要的参数以使对象履行职责。

然而，远程对象常常会破坏这一点。因为远程对象的实例化通常要发生在客户可以给初始化传递参数之前，因此远程对象不得不将初始化和实例化分离，在这里我们开始遇到麻烦。为了与经典的Java设计模式保持一致，我们常常在属性级别上单个属性地进行这种初始化调用，致使我们要进行多个调用，以便在对象中创建上下文（状态），实现某种有用行为。这致使我们在先前确定的问题中绕圈子：如果这些建立上下文的方法中有一个调用失败了，将会发生什么？没有任何编程语言能很好地处理这些问题。

让我们不予考虑上下文，或者可能更准确的说法是，让我们通过将上下文纳入到所有往返通信中作为其一部分，来排除上下文迷失的可能性，让我们再次从午餐时间的谈话开始：

Stu: 嗨，一起吃午餐吧？

Justin: 好的，一起吃午餐是个好主意，什么时间？

Stu: 让我们在通常的时间——正中午去吃午餐吧

Justin: 正中午吃午餐对我来说不行，11 点怎么样？

Stu: 好的，11 点吃午餐很好，什么地点？

Justin: 让我们 11 点在市区的皮萨店吃午餐吧。

Stu: 非常好，我会在 11 点在市区的皮萨店和你会面吃午餐。

从来都没有人会这样谈话。太笨拙了，冗长累赘，无论怎样，这在两个有任何形式的短期记忆的人之间都完全没有必要。但是，请记住，这正是关键之处：我们并不希望两个健谈的人，在此情况下即对象，必须要记得所有事情——这使得系统能够更好地扩展，因为现在任何对象在任何时间都可以参与到这个对话中：两个主要成员之间每次来来回回的交流都是上下文完整的。

将这种思想运用到代码中则意味着：不是将建立上下文作为已经从调用分离出来的 API 的一部分，而是将它作为每个“做某件事情”的调用的一部分。

```
CartoonCharacter cc =
  (CartoonCharacter)rootContext.lookup(...);
String catchphrase =
  cc.utterCatchPhrase("Fred", "Flintstone", "Bedrock");
```

在这种情况下，我们不是通过三个分离的API调用在允许调用utterCatchPhrase之前传送上下文，而是作为调用本身的一部分来实现。这也使得utterCatchPhrase方法可以更加容易地忽略先前调用的任何状态。

```
CartoonCharacter cc =
  (CartoonCharacter)rootContext.lookup(...);
String catchPhrase1 =
  cc.utterCatchPhrase("Fred", "Flintstone", "Bedrock");
String catchPhrase2 =
  cc.utterCatchPhrase("He-Man", null, "Eternia");
String catchPhrase3 =
  cc.utterCatchPhrase("Thundarr", null, null);
```

这样能够运转是因为上下文通常是存放在局部变量和/或方法参数中的：

```
public class CartoonCharacterImpl extends UnicastRemoteObject
  implements CartoonCharacter
{
  public String utterCatchPhrase(String first,
```

```
                                String last, String ht)
    {
      // Do our lookup here
    }
  }
```

实际上，这看起来很像为我们效力的无状态会话 Bean：因为无状态会话 Bean 在方法调用之间不持有任何状态，这暗含着需要上下文完整的通信。这也是为什么上下文完整的通信更可取的部分原因：因为集群中的任何对象都可以响应请求。之所以没有任何隐含的标识，是因为上下文可以在任何特定对象上建立（有状态会话 Bean 和实体 Bean 都属于这种情况）。

上下文完整调用的另一个有用好处是：如果调用在某个时刻不能实现，那么你也拥有稍后重建该调用所需的一切。这在调用处理失败的情况下很有帮助：将上下文打散存储到某种中间存储介质，稍后取回再重新尝试。实际上，如果那个中间存储介质是消息层，那么只要我们需要，我们就可以自由地获得异步通信。因此我们现在可以获得上下文，将它置入Message中，然后将Message插入队列用以后面的处理，这大多发生在负载降低或者断电恢复的时候。

再讨论一下事务的考虑因素——在上下文完整的方式下，什么时间什么地点开始和结束事务都变得相当清楚。在获取/设置方法（getter/setter）中，就不那么清楚了。我们是否应该在第一个调用set时开始事务，并且仅当调用utterCatchPhrase时才结束事务？这会导致事务在方法调用之间是悬而未决的，完全违反了第 30 项。如果你想避免留下悬而未决的事务，那么每个调用就都需要在它自己的事务下进行操纵，不过，这会给我们留下语义混淆，因为其它客户可能对同一对象调用别的set方法，这些调用与上述调用序列会交织在一起。在那种情况下，事情就变得更糟了。

更有趣的是，这种模式很适合Web服务文档-字符（document-literal）通信（与RPC编码对立）的主张，因为文档-字符通信需要整套参数作为信息包的一部分呈现出来，Servlet也可以从中受益；如果在HttpSession中没有存储任何隐含的上下文依赖，那么通过Web应用（甚至是通过多个Web应用）重用表单会变得更加容易。

尽量避免上下文，你就可以构建出扩展性好的系统。在实现过程中，几乎是偶然的，你将会

趋向于构建需要较少的跨网络往返的远程 API（这样可以遵从第 17 项），因为在调用自身被执行之前，不需要调用任何“初始化方法”来设置上下文。

第 19 项：优选数据驱动的通信而不是行为驱动的通信

经过这么多年，IPC 已经逐渐从“将数据从一个处理传输到另一个处理”进化到“在其它处理中产生某个函数（或方法）调用”。尽管我们可以把这种不同仅仅解释成细节和封装的不同（“噢，进行远程过程调用时，仅仅需要发送一个请求消息，然后等待响应消息，这没有什么不同”），事实上，按照通信风格的目的，驱动产生了两种不同种类的通信交互，一个是行为驱动，一种是数据驱动。通常情况下，你可能会倾向于数据驱动的通信，尤其是在通信要跨越多个构件边界的时候。尽管它主要应用于 Web 服务，在那里被称为面向文档的风格，不过，这种风格也适用于直接以 Java 为中心的通信层。

对大多数开发者来说，行为驱动和数据驱动这两个概念还是一个新的话题。不同的效果在于你——开发者，打算在什么时间进行网络通信。在像 RMI 和 CORBA 这样的标准 RPC 对象工具库中，你是在调用一个方法——也就是说，你是在命令一个对象实例去执行具体的方法，然后返回。你传递参数，等待在远程机器上执行直至结束，然后获得返回的结果。这相当简单直接，不是吗？

但是，在数据驱动的方式中，你本质上从没有“调用方法”，相反，你是向远程资源发送一个数据包，它会做该做的事情。并没有任何隐含的假设表明一个响应将会返回——实际上，在很多方面，如果没有一个响应，反而会好很多，因为这样可以让你继续处理本地事务而不必等待通过网络返回给你数据（请参阅第 20 项）。

更具体一点，考虑一下在线订货处理系统，就是很像Amazon.com所用的那种。用户已经填写好购物车转到校验页，指明订购已经准备好。现在该你负责了，接收这个订单及其里面的内容，然后对它进行常规的处理：验证信用卡、收取费用，等等。

一种方法是构建一个远程对象集合，从客户层可以调用它们（在这种情况下，就是一个

Servlet，但是 EJB 的会话 Bean 看起来和它几乎一样），并且按照订单所指示的一次一个地调用它们：

```
public void doPost(HttpServletRequest req,
                   HttpServletResponse resp)
  throws ServletException, IOException
{
  // . . . Put in verification and input-validity-checking
  // code here; see Item 61 for details
  //
  HttpSession session = req.getSession(false);
  OrderModel order =
    (OrderModel)session.getAttribute("order");
  Context ctx = new InitialContext();
    // Always do lookup, in order to permit failover;
    // see Item 16
  // Verify credit card
  //
  CreditCardProcessor ccp =
    (CreditCardProcessor)ctx.lookup(...);
  if (ccp.verify(order.getCreditCard().getName(),
                 order.getCreditCard().getNumber(),
                 order.getCreditCard().getExpirationDate()))
  {
    if (ccp.charge(order.getCreditCard().getNumber(),
                   order.getAmount()))
    {
      for (OrderItem oi : order.getItems())
      {
        // Process each item, take it out of stock, whatever
      }
    }
  }
}
```

此外，不管这些代码是出现在 Servlet 中、会话 Bean 中或者是与两者之间实在不相关的事物中——此时的重点是每一步都是通过到 Bean（无论是远程的还是本地的）上的行为性动作而被仔细度量的，而这些 Bean 是以模块方式实现了每一步。

然而，在数据驱动的方式中，数据的发送者没有假定一定要发生什么；实际上，客户端会因

为它相对来说只需无知而欣喜若狂。客户端不用发出任何行为命令，而仅仅包装数据然后发送到中立层（通常是某种存储层），在这里其它的感兴趣方会读取数据以进行处理。这种方式的经典实现方法是通过 JMS：

```
public void doPost(HttpServletRequest req,
                   HttpServletResponse resp)
  throws ServletException, IOException
{
  // . . . Put in verification and input-validity-checking
  // code here; see Item 61 for details
  //
  HttpSession session = req.getSession(false);
  OrderModel order =
    (OrderModel)session.getAttribute("order");
  Context ctx = new InitialContext();
    // Always do lookup, in order to permit failover;
    // see Item 16
  // Remain ignorant, just send it on for further processing
  //
  Connection conn = ...; // Where we get this isn't important
                         // now
  try
  {
    Session session =
      conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Destination dest = ctx.lookup("jms/NewOrders");
      // Note that there's no hint of what will happen in
      // the queue name
    MessageProducer producer = session.createProducer(dest);
    ObjectMessage msg = session.createObjectMessage();
    msg.setObject(order);
      // JMS ObjectMessage requires Serializable objects; if
      // OrderModel is a JavaBean, it must be Serializable
      //
    producer.send(msg);
  }
  finally
  {
    session.close(); // Aggressively release resources;
                     // see Item 67
  }
  // And we're out, happy and carefree; generate some kind of
  // "Thank you" response page back to the user, probably by
```

```
    // forwarding to a JSP
    //
  }
```

注意，在数据驱动的示例中，客户端完全没有给出任何关于这个订单将会发生什么的暗示——我们仅仅是让某些人（我们甚至不知道是哪些人）知道已收到一个新订单。从那以后将会发生什么不再是我们所关心的。

将这两种方法分别直接归类为 RPC 和消息机制可能会比较吸引人，因为使用像 RMI 和 CORBA 这类基于 RPC 的工具库很容易模拟行为驱动的系统，而数据驱动的系统也很适应面向消息的系统（例如 JMS）。如果这样会有所帮助，那么就自由地使用这种分类吧，不过，紧记这样会过度简化——因为使用 JMS 构建行为驱动的系统或使用 RMI 构建数据驱动的系统通常也是可行的，它们各有各的优点和缺点。这只是由于方式的风格，而不是构建所基于的技术，产生了这种不同。

就算上面这些都是事实，那还有什么呢？对开发者来说，行为驱动的方法可能更容易理解和使用，尤其是因为它很容易和创建 J2EE 应用的标准方式兼容——那么到底为什么每个人都在注意去避免它，尤其是避免使用那些不会很快地和我们用传统思维所想到的方法凝合为一体的东西？

三个原因：演化、中介以及可重用操作。

存在这样一个事实：一个企业系统会有一个“延展”，超越构建它的开发小组以及为其构建的团体——请记住第一章讨论过的企业系统中 10 大谬误，尤其是谬误 9：“系统是整体式的。”变动是无法避免的，而一个 API 一旦向世人公布，就不能再变动了。或者，更准确地说，只有该 API 的所有客户端都赞同修改时，它才可以被修改，这或多或少都是在说同一件事——让企业伙伴、部门以及系统管理员都赞同这个新的修改，正如保守党和自由党也要达成一致一样。

遗憾的是，变动必然会发生，并且数据驱动的方法比行为驱动的方法更容易处理这些。例如，21 世纪的信用卡支付通常不仅需要标准的 16 位数字的信用卡卡号，还需要以印刷形式打印

输出到卡背面的 3 位数字的“验证编码”。遗憾的是，在我们的 CreditCardProcessor 类中设计的 API 不接受 19 位数字，只接受 16 位数字，因此我们需要改变 API 类以便考虑到第四个参数，即验证编码。

在 EJB，或任何基于方法调用的分布式工具库中，对这种事进行的处理，意味着要改变接口和重建被传送到客户端的远程存根。幸运的是，一定数量的改变可以悄悄地被处理掉——例如，一个接口的方法通常可以默默地添加到远程实现中，而不会断裂先前的存根（RMI 支持这个特性）——不过，这样很快会耗尽远程工具库的容忍度。例如删除一个参数，或完全删除一个方法将会很快地断裂现有的客户端，快到你可能还没来得及说“嗷!”

但是，如果我所发送的是一个后面没有隐含行为的数据结构，那么，向数据中添加几个额外位就像在 CreditCard 类中添加额外域那样简单，CreditCard 类的对象实例存储在 Order 类中。如果这是一个序列化对象，当这些断裂变化到来时，我可以使用序列化（请参阅第 71 项）工具尽可能地处理这种数据结构的演变。

或者，事实上我可以直接创建一个新的数据结构，可能称它为Order2或OrderEx，或是任何一个普通的名字，消息处理器（那些“拉”消息并进行处理的程序）可以在运行时检验实际类型以确定下一步要做什么。实际上，没有人说过从JMS的Queue中拉消息的处理必须要单独运行。不过有时仅让一个处理器处理这些新订单会方便一点，当有重大改变的情形发生时（可能现在某些订单需要加密，而先前并不需要），有时创建一个全新的处理器可能会更简单一些，我们甚至可以将原来那部分仍保留下来处理先前的客户可能仍在给我们发送的消息集合。JMS并不关心这些——它的核心就是它所看到的一切都是字节数组。

如果你不想和序列化打交道，你通常可以选择一个对类型要求更为宽松的方式，这种发式不发送实际的对象而是发送包含了基本类型的 Collection 对象。Map 接口很适合这种情况，因为 Map 中的每个键/值对可以大体对应一般类中的域。是的，你正在牺牲编译时的安全来获得灵活性，因此要以防御性的策略编码，并不断地测试以避免任何产品部署的尴尬局面发生。

当然，最后但是却很重要的是，我们必须考虑极端灵活的数据格式：XML。发送某种基于 XML 格式的消息可以获得大量的数据演化的灵活性，使用或不使用 XML Schema 都可以。

并且，在数据驱动的设计中使用 XML 可实现更深度的互操作性，这种深度的互操作性在基于 JMS 的方式中并没有呈现出来的，尽管将其考虑进去并不是很难。（然而，要确保你的 XML 没有对将要不断接收 XML 的平台做出不适当的假设，这意味着除了少数非常简单的实现之外，没有任何对象到 XML 的自动转换 API；请参阅第 43 和 22 项以查看更多细节。）

演变并不是优选数据驱动设计的主要原因，尽管它的的确确是一个有利因素。数据驱动的信道可能包括大量的中间介质、软件或硬件处理器，通过将喜欢使用不同传送机制的有用的横切关注进行分层，以某种有用的但通常不可见的方式构成了完整的通信处理（例如，SMTP/POP3 甚或是像 Jabber 这样的即时报送协议）。

实际上，中间介质这个概念可以令人信服地被泛化为具有更广泛的意义，那就是在客户端仍不清楚信息的最终接受者时所能获得的灵活性。对于发起者来说，在一个消息驱动的环境中，消息层会一直持有这些消息，甚至当最终的接受者处于离线状态时也是如此。换句话说，客户端永远不必去处理那些由处理消息的代码而引发的中断——只要消息层仍处于活动状态，我们就可以按照我们所希望的频率去演化和修改Queue后面的代码而不会影响客户端（在这种情况下，就是接收订单的网站），因为消息会直接累积起来直到我们准备将处理器恢复到在线状态。（当然，我们假设客户端在等待某种响应时一直没有被阻塞，这也意味着你正在以某种行为驱动的方式使用JMS。）

更重要的是，因为发送者和最终接受者之间是断开的，我们可以对这些消息做很多有趣的处理，而这在行为驱动的通信中是不切合实际的。Hohpe 和 Woolf [EAI] 提供了大量有关有用模式的材料，例如消息路由器（“将单个的处理步骤之间进行解耦合，这样消息就可以根据一个条件集而被传递到不同的过滤层”78），基于内容的路由器（“处理这种情形：某个单一逻辑功能的实现被分散到了多个物理系统上”230），以及前面已经间接提到过的范化器（Normalizer）（“处理那些语义等价但是以不同格式到达的消息”352）。从本质上说，我们为系统的挂钩点（请参阅第 6 项）创造了各种各样的机会。

但是等一下——现在行动，我们将会随意地运用上下文完整（请参阅第 18 项）和避免过多的网络“对话”的倾向（请参阅第 17 项）！因为数据驱动的方式倾向于使用简单而完备的数据集，而不是域模型[Fowler,116]对象，所以很少会构建这样一个系统：该系统将创建出大

量随机的跨网络的往返。（当然，确保这一点的最简单方法是保证在网络上传输的事物都没有继承自java.rmi.Remote。）

遗憾的是，数据驱动的通信是一门很微妙的学科；最佳情况下，它是一个光滑的斜坡；最坏情况下，它是一个令人费解的学究式的傻瓜。比如，让我们先返回到电子商务网站。发送一条“下订单”消息和发送一条“添加订单”消息在语义上的差别很小。它们的主要差异在于：对现有的订单进行修改或变更时，所发生的情况会有所同。在数据驱动的构架中，订单只不过是被放回到同一队列中，并且它被假定为后端处理器会对其进行恰当的处理，而在行为驱动的情形下，我们将不得不创建新的方法和代码来处理“修改”订单，而不是放置一个新的订单。

当你需要转向开放集成模式（在这种模式中，正在进入的客户端可能来自于完全不同的平台）时，使用数据驱动的设计的长处才会真正显现出来。因为你现在的 API 是根据极其简单的数据结构来定义的，而不是复杂的对象模型，所以它可以使转向文档/字符的 Web 服务变得更容易，并且，对于非面向对象语言的客户端来说，也更容易适应。

顺便说一下，万一你准备抛弃这里介绍的整个思想，返回到你所喜爱的行为驱动的设计方法中去，记着：如果你遵循J2EE中的惯用思想，并使用数据传送对象[Fowler, 401]，以阻止直接从客户层访问实体Bean，那么你使用数据驱动方法（当然，依赖于你的会话Bean是怎样实现的）会更有效。尽管从表面上看它是如此极端的方法，但是它并不是如此，没有任何书面约束声称DTO必须严格地映射到它对应的实体Bean。这样就给予你将DTO从实体（或实体们）分离的机会，从而获得前面提到的一些好处。

要想开始运用数据驱动的通信，那么使用 DTO 是一种相对比较容易的方法，但是，不管你是采取基于 XML 的方式、基于集合的方法还是使用 DTO 的方式，关键是要把重点放在你想交换的数据上，而不是你想执行的行为上。在松耦合的构件构架中（请参阅第 1 项和第 2 项），这种做法可以使每个构件各自独立地演化，而不会触发大规模的重新编译以及更糟的新代码运行时所产生的大量错误报告记录。

第 20 项：避免为远程服务请求去等待响应

请求-响应通信模型的缺点之一根植于它的根本意图：我们必须等待响应。这样说好像极为愚蠢，但是等待响应被证明是通信中的主要障碍。

响应时间这个概念有一个令人不快的倾向，那就是它在整个系统中会累积。先从最终用户的角度来看。用户点击浏览器 Web 表单上的按钮。浏览器向 HTTP 服务器（穿越其它任何位于代理服务器和网关之间的底层设施）发送 HTTP 请求。HTTP 服务器将请求提交给内部的处理代理——可能是某种形式的 Servlet，它会对 POST 过来的数据执行某些处理（希望除了处理这些事物的页面脚本之外，还包括对输入的验证——请分别参阅第 61 和 56 项），然后向存储数据的数据库发出调用。这无疑意味着我们实际上必须将 Servlet 设置为暂停，一直到数据库请求被解析、部署、执行以及将数据返回为止。或者我们可能向 EJB 层发送请求，EJB 层会在再次前往数据库读取数据之前执行一些处理。基于这些被读取数据，我们可能要执行更多的处理步骤，然后执行另一个数据库调用来更新数据，返回到 Servlet，它接着将请求转送到 JSP 页面……

期间，我们的用户已经老死啦！

在很多情况下，基于 HTTP 系统的长久的反应时间并不能归咎于系统的任何特定构件。错不在 RMI 、HTTP 或者你碰巧正在使用的数据库自带的连线协议。系统不会被任何特定的硬件元件或网络速度所延迟。当然了，它们中的任何一个都会产生瓶颈，但是不使用它们也不能确保缩短反应时间。相反，长久的暂停时间应归咎于每一步处理所引起的反应时间的一点点累积，还有就是我们在继续处理之前必须等待每一个处理都被执行完成。

例如，再次考虑一下经典的电子商务网站。我们已经知道前端的 Servlet/JSP 层需要和后端的多个诸如数据库之类的代理进行通信，以显示网站的某些部分。（希望本书自身的分类条目还算固定，因此，能够指导我们对预生成的内容进行优化，就像第 55 项所讲述的那样。）我们知道在我们继续实现用户的需求和指示之前，有很大一部分事情要做，但是在对用户动作的直接响应中，具体需要做多少呢？

回想一下最后的检出阶段，比如：我们知道将来我们会想要处理用户的信用卡号，但是在“谢谢您的订单”这样的响应显示之前，它需要作为处理的一部分而发生吗？执行信用卡服务需要很长时间——想起上次你在结帐处的等待了吗？——而且，信用卡服务对系统其余部分的负载是相当敏感的。你真的想让你的用户只是为了所显示的“谢谢您的订单”而承受这一切吗？尤其是如果用户认为什么都没有发生过，并且具有第二次点击按钮的恼人倾向时，也是如此吗？

无论何时，都应该寻找方法将这种同步模型打散到多个异步通信步骤中，以避免强制客户端等待。注意我在这里特别说的是“客户端”而不是“用户”，因为你的客户端如果是另一个程序的话，也不会有问题。

例如，在电子商务网站的检出阶段的情况下，我们不需要将处理信用卡作为最终处理阶段的一部分。哦，不要误解我，我不是建议我们不需要处理信用卡支付，我只是建议在用户等待的时候不需要做这个。

这可能会让你吃惊：如果用户的信用卡失败，你怎么办？我们不能只是“吃掉”订单，不再作进一步的处理，放任用户认为交易是完全成功的。所以当信用卡失败时会发生什么呢？

首先，请考虑实际发生的可能性。由你的客户所决定的是信用卡交易成功比交易失败更具可能性。毕竟大多数人知道他们的卡何时是荷包鼓鼓的，如果他们知道会失败的话就不会再使用它了。（某些信用咨询服务代表可能不同意这个评价，但是大量的证据表明与他们的想法正好相反。）因此，不要计较这些，即使 25%的信用卡交易失败，也意味着有 75%的时间你的用户被强制等待从来都不会发生的事情——“你的信用卡失败”这则消息。四分之三的用户必须忍受这不必要的延迟。

其次，我们在这里做一个重要的假设——前端所依赖的所有服务实际上都被启动了并正在运转。回想一下，“堵塞发生了”，而且我们不认为这是一个完美的世界。由于一些良性的（可能我们还没有向代理支付每个月用于处理信用卡的服务所需的费用，如果我们没有直接去处理信用卡的话）或灾难性的（上帝不允许它发生，但是地震、洪水以及龙卷风非比寻常，而且 911 还将恐怖行动列入到了我们不得不考虑的危险评估表中）的原因，信用卡处理服务完

全有可能处于停止状态。如果处理是按照异步模式进行的，那么在很多情况下，用户的知觉依然完全不会意识到服务的中止。如果处理是按照同步模式进行的，那么只要服务中止发生在适当的位置，我们的系统就会完全崩溃。

这里面牵涉到很多你无法想到的内涵。如果我们创建了 5 个同步远程服务，而且每个服务一年只停止运转一天（这是一个相当好的错误记录，当你考察它时，会发现每年大概只有 0.33% 的停工时间），那么我们的整个系统一年就将停工 5 天，再加上任何时刻由于我们自己的软件升级、维护、错误以及其它任何使系统宕机的事物而造成的停工。从本质上讲，我们的停工时间不仅是我们养成了坏习惯的因素之一，而且也是每个我们所依赖的服务包含坏习惯的因素之一。

第三，并不是我们没有机制去实现在发生错误的时候，异步地通知用户——最为流行的互联网技术——电子邮件应运而生。（E-mail 已经在无数的地方被引用，因为它比 HTTP 和 Web 本身被更为广泛地使用。）如果 HTTP 是互联网上的请求-响应协议之王，那么 SMTP 和 POP3 或 IMAP4 就是互联网上的消息协议之王。好好利用他们吧。如果信用卡未能处理成功，向用户发送一封 e-mail 并让他/她在愿意的时候做出回应——一些用户甚至不想费事地去做出回应，这样就完全地降低了系统的负荷。

最后，异步执行可以有助于避免“Slashdot 效应。”我使用这个术语来描述一种作用于服务器的效应，当从事极度非法买卖的龌龊网站（Slashdot）抛出一条关于互联网上某个有趣事物的消息时，就会产生这种效应。每秒钟都会有成千上万条请求开始涌入，一个可以很轻松地处理平常流量负荷的网站在激增的流量负荷下痛苦不堪。如果你的可扩展性有任何弱点，你将会看到这些弱点首次爆发的情况——当然，它通常还会导致站点在巨大压力下崩溃，你的公共关系也会处于一种极为难堪的境况。

在异步方式下，如果电子商务前端执行尽可能多的处理，那么即使我们的后端服务不能跟上突发的流量负荷，我们的网络仍旧是安全的：后端处理将会尽可能地处理，即使消息堆积如山。大概到某个程度，峰值开始逐渐减弱时，追赶的游戏就开始啦。随着时间的推移，外来的消息都会被处理掉，队列被清空，系统经受住了这个可测量的梦魇般经历。或者，在最坏情况下，如果处理器无法及时处理进入的消息，那么系统管理员（当然是那个通过你所构建

的监视支持系统的在紧密地关注系统状况的人，就像第 12 项所述的一样。）可以在其它机器上发起另外一个或两个后端处理元素，直到清空所有的请求为止。

异步处理可以按照多种方式实现。有一种方式是完全异步的模式，正如上面讲述的那样，通过使用像 JMS 或 e-mail（大多通过 JavaMail）这种消息传递和面向消息的技术。EJB 的消息驱动 Bean（MDB）实现它则显得很轻而易举——在电子商务的应用场景中，当用户购物结束时，不是产生一个调用 EJB 和其它远程服务的集合，而是直接将 JMS 消息放入一个 Queue 或 Topic 中，然后立即返回。监听 Queue 或 Topic 的 MDB 将会发现这条消息，然后被唤醒并处理它，并执行所需的任何处理。如果其间出现问题，例如信用卡验证未通过，我们会向最终用户发送一封 e-mail，要求他或她联系客户服务代表以解决错误。

另一种方式是在一个单独的线程上执行远程请求，因此当我们等待远程服务响应的时候，“主”线程仍能继续处理。这种方式与 MDB（或者其它基于消息的解决方案）相比，显得不太可取，原因有二：（1）我们仍旧受限于这样的事实：即我们仍期望某种响应，因此直到远程服务请求返回之前，都无法完成我们的处理。（2）它使得我们的服务对拒绝服务攻击来说是门户大开——攻击者可能利用 HTTP 请求集涌入 HTTP 层，并且请求中的每一个都可能派生出五个线程，被分派到 HTTP 层之后的后端的不同部分……..

在某种程度上，我们不能完全消除请求-响应通信——它如此有用如此必需以至于无法完全放弃。例如，用异步通信方法，通常不能对验证请求进行最优化，因为在我们确定用户身份之前，不可能知道要向用户显示些什么。不过，在有些情况下，通过将请求验证的用户界面中的元素放置到独立的 HTML 框架中，我们也许能够借助某些技巧，因此，在我们完成验证动作的这段时间内，可以将页面中其它部分产生并显示出来，以提供给用户某些能够看到的东西。记着，在很多情况下，并不是真正的反应时间而是感觉上的反应时间，将网站区分为“快而可用”和“慢得像蜗牛爬一样”。

第 21 项：考虑构件的划分以避免任何一台机器负载过重

几乎从最开始，集群和故障转移以及其它这样的词汇就已经成为了 J2EE 程序员的专用词汇。

实际上，它们在 J2EE 诞生开始，就已经成为 J2EE 相当重要的一部分以至于丧失了它们原来的所有含义。对于许多 J2EE 开发者来说，开发商和 J2EE 布道者对集群、故障转移、划分和分布式的呼声如此之高，以至于在这呼声中迷失。

企业级系统，由于它们的多用户本质，所以大多都是某种分布的形式。这就无法避免一些灾祸。分布不但使得多用户和/或应用之间的数据分享变得更加容易（通过集中数据），通过确保整个系统即使是在系统的某个部分由于计划内或计划外的原因而导致当前不可用的情况下，仍然能够保持运转，分布还能够产生更好的操作。最重要的是，一个恰当的分布式系统可以提供大量的优化，主要是在通过保持所需数据本地化来避免网络流量这个领域内（请参阅第 17 项）。

在你自言自语道 “哦，集群，每个人都可以得到”，然后继续工作之前，先深深地吸一口气，和我一起把心悬起来。集群只是几种分布系统的方法之一，并且从多个角度来看，它甚至不是利用构件划分和跨多机负载的最为有效的方法。让我们回退一下，重新从更高级别考虑一下现在的情形。

根据中间件系统的本质，它趋向于集中化——毕竟，中间件的整个重点就是将全异构的系统整合为某种统一的整体。使这成为可能的部分原因是中间件创建了“胶水”，它能够将各个系统焊接在一起的，并且出于维护目的，如果没有其它的东西，那么在给定的构件在网上只有一个实例的情况下，它就是最好的选择。

不过，集中化也有它的问题，最明显的是，当我们消耗掉某台给定机器的可用容量时——也就是，我们此时陷入的情形是，需要支持的用户和/或资源数量，多于集中化服务器所能提供的支持数量——直接将一台机器添加到混合集群中会骤然产生一大堆的问题。并发尤其关心这种示人以丑的事情，但是瓶颈和其它扩展性问题也会突然出现。

划分的方法之一就是分布系统。本质上就是，我们将一个给定构件分割得更小，分割成“次原子”片，从战略上将这些分割片分布到整个系统上。

考虑一下 DNS。如果有一种服务，基于它可以构建互联网的完整性，那么它就是 DNS。没

有 DNS，www.neward.net 就成为无意义的 ASCII 字符的集合，那么你就必须记下实际的 IP 地址，就是对大多数人来说都显得无意义的四段数字。不过，好像这还不够，记着 DNS 提供一种位置独立性的量度，正如在第 16 项中所叙述的那样，没有了 DNS，这都将不可能（或者，至少很难构建）。

但是，请考虑 DNS 是如何工作的。每当你打开一个连接到 LAN 或 WAN 的任何地方的某台远程服务器的套接字，你就进行了一次 DNS 查找以获得人们易读懂的域名的 IP 地址。在互联网上和你做同样事的人们会有许多，你会突然意识到如果有一个可扩展并且可靠的系统该多好呀，那就是 DNS。想象一下，如果所有这些请求被直接导向到你的服务器——你认为把它直接放到地板上等待它自己融化需要多长时间？

事实是，DNS不是按照集中式服务或者数据知识库的形式来实现的，主要原因是：地球上没有任何一台机器强大到能够处理数以百万的、DNS服务器必须承受住的并发请求。相反，DNS通过互联网以高度分割的形式分布到世界各处，而且是分成互不相交的层次结构。当解析一个DNS名字时，就像是穿过直接包含在域名本身中的服务器集合。简单地举例来说，在域名www.neward.net中，我们从上层域名*net*开始，在它之下我们查找第二级域名*neward*，到达我的服务器，识别第三级名称*www*。我选择创建另外一个域名解析我的局域网内的不同服务器，我将它公布到我的DNS服务器上，互联网不需要知道其它任何变化——ftp.neward.net完全遵循同样的路径，并且我的DNS服务器一如它为先前的请求应答*www*那样应答新的*ftp*前缀。（不过，这种设置有一个问题，它要求客户在域名全部被解析之前，进行几次网络跳跃，因此DNS通过包含客户端缓存的域名而不断地进化。我们只需瞬间就可以检查它的分支。）

作为一个有趣的边注，日期能够对分布式数据库和我们所谓的“远程数据访问系统” 清晰地加以区别：“在远程数据访问系统中，用户可以操纵在远程位置上的数据，甚至是同时在几个远程位置上的数据，不过，‘缝隙表明’用户肯定或多或少意识到数据是在远端，因此必须要做相应的处理。在一个真正的分布式数据库系统中，缝隙被隐藏掉了” [Date, 652]。既然J2EE整个重点是使这些缝隙再次消失，那么这种区别就不是我们的目的所必需的，不过看看这些术语是怎样被定义的，会对我们有所帮助。

数据库专业人士有一段时间已经认识到了这种分割。数据库系统概论（*Introduction to Database Systems*）[Date]第 8 版在 21 章进行了详细介绍，着重讨论了分布式数据库。其中，Date将*数据切割（Data Fragmentation）*定义为出于物理存储目的而将数据分割成割片或切片的能力；例如，我们可能将一个EMPLOYEE表分成两个表，一个用于公司的纽约支部，而另一个用于伦敦支部。

```
FRAGMENT EMPLOYEE AS
  N_EMP AT SITE 'New York' WHERE dept="Sales" OR
    dept="Admin",
      L_EMP AT SITE 'London' WHERE dept="TechSupport";
```

换句话说，任何EMPLOYEE表中满足第一条件（例如，员工可以在Sales和Admin两个部门工作）的任何数据将会物理存储在纽约服务器上名叫N_EMP的表中。并且任何符合第二条件的数据将存放在伦敦。在真正分布式的系统中，“缝隙不会显示”；像″SELECT * FROM employee WHERE last_name='Date’ 这样的查询可能会在伦敦和纽约的数据库上操作。

实际上，我们可以讨论两种不同类型的数据切割：水平分割，我们将数据按照数据本身中的元素划分（正如上面我们做的那样），和垂直分割，我们在表的中间进行分割，可能将员工的FIRST_NAME、 LAST_NAME和 EMPLOYEE_ID列放到纽约数据库中，而DEPT 和SALARY列放到伦敦站点。（根据实际需要，DNS选择使用水平分割方案；实际上，它是递归水平分割方案，因为这些服务器是嵌套在其它服务器的分割区间内的。）

我们可以将同样的规则在代码级别和数据级别上运用到 J2EE 系统。我们可以水平分割数据库，将位置敏感型数据存放到按照地理位置集中分布的数据库服务器上（例如和人力资源有关的数据存放到本地部门的数据库中）；或者我们可以垂直分割数据，在多个服务器上将表和/或表的一部分划分开来，所有这些都依赖于将来打算怎样使用系统。在同样的风格中，我们可以将代码按照同样的方式进行分割——将与人力资源表一起运转的代码放置到靠近这些表的服务器上，等等。虽然代码很少水平分割，但这种分割仍是可取的。例如，你可以将近乎相同的会话 Bean 分布到两台服务器上，这两个 Bean 的不同之处在于它们所期望的输出或输入的不同——“总费用超过$1,000 的支出报告被导向服务器 A，而少于$1,000 的被

导向服务器 B。”

当我们将JMS 的Topic和Queue实例分散到网络上，以确保消费那些消息的处理器运行在宿主Topic或Queue的同一台机器上时，可以使用另一种分布系统的方式。每个Topic和Queue自身也可以遵从一定数量的分布——如果它发现一个给定的Queue超负荷工作（因此会变成瓶颈），我们只要将Queue水平划分成两个独立部分，例如，第一个用于以A–M开头的名字，另一个用于N–Z开头的名字。

水平分割的关键问题是，在某些地方有关水平分割的“准则”必须要遵守，而且重要的是，必须要实时遵守。例如，如果我们按照员工的姓（所有员工中，姓以 A–M 开头的存放在服务器 1 上，以 N–Z 开头的存放在服务器 2 上）水平分割员工数据库，我们需要某种规则来反映这一点——并且如果这一点难于硬编码成代码，那么三个服务器的重新分割就需要重新编码实现。垂直分割与此类似。

分割的第二种方法是复制系统。这是一种通用的“集群”方法，如果不是全部的话，那么至少是许多 J2EE 系统都提供这种方法，我们之所以要复制系统的拷贝并将这些拷贝存放到多个节点上，是因为如果一个节点失败了，其它任何一个节点都可以易如反掌地接受请求。复制提供了一系列的优点。如果数据是在本地复制的，那么这意味着执行任何一个请求所需的网络流量都要更小，同时还能获得更好的可用性——如果集合中的任何一个节点（服务器）出了问题，都不会有任何中断期被记录，因为任何节点都有能力响应请求，而不像前面讨论的分布式那样，如果用于 www.neward.net 的 DNS 服务器关闭，别无其它的服务器能够替代它，因此会拒绝该域的 DNS 服务直到该服务器重新启动起来。

我们会在第 37 项中将再次讨论复制，不过有几个和通信有关的问题值得在这里提一下。

正如在第 37 项中所叙述的那样，复制带来的问题很简单：更新传播。假定一下：我们将数据存储在整个企业中三个不同的数据库实例中。只要我们所做的都是从这三个数据库中读取数据，我们不用考虑就可以读取——每个实例都有一个精确的数据备份，因此三个中的任一个都可以充满信心地响应我们的请求。一旦我们更新其中的一个，事情就变得难于处理——我们现在拥有那个数据的多个备份，因此修改其中的一个意味着我们将无法和其它的数据保

持一致，直到相应地将它们都更新完毕。并且如果恰巧同时有两个更新……..

我们考虑一下 DNS 中的这个问题；例如，因为客户将 DNS 项复制到本地存储器中（他们要缓冲这些项），在一个 DNS 变更被渗透到整个互联网之前，它经常要被缓存数天。不过，在此期间，如果“旧的”服务器由于某些原因关闭或者下线，那么客户由于服务中断而离开，而这个中断并不是由服务器再次启动之前的这段时间所造成的，而是因为客户认为这个再次启动的服务器已经不再是原来的那台服务器了。在 DNS 中这种相应延迟是允许的，虽然更新域名/IP 地址之间的关系不会经常发生，但是对于一个更新可能每秒或者不到一秒就会发生的系统来说，将不会这样长久地正常运转。（具有讽刺意味的是，由于某些原因，使用信用卡时我们一直会发现这种响应延迟，即使在你已经完成收据的签名之前，找零已经物化到了卡上，仍旧需要很多时间才能完成支付。）

此时，你可能会试图返回到某种全局同步的方案，像用一个远程单例对象作为更新管理器；但是如果你想使你的系统可扩展，就不能这样做了。正如第 5 项中所叙述的那样，你会引入瓶颈问题，因为现在，对你数据库的更新将会被更新管理器处理请求的速度所阻塞。你还需要一个远程服务在你继续执行之前做出响应，这使你陷入了第 20 项所介绍的问题中。当然，你可以总是分割你的更新管理器，但是现在你又回到前面分布或复制的问题了。你可能认为你可以使用时间戳，或者有关乐观并发的每个讨论（请参阅第 33 项），但是很快你会发现又陷入了另外一个问题，已经折磨计算机科学多达数年之久：几乎不可能让两台计算机的时钟完全一样，在那种你最好需要毫秒级解决方案的高可扩展系统中，更是如此。

将直接更新所有备份作为更新操作的一部分似乎是个不错的办法；遗憾的是，这充其量只是一个天真的建议而已。记着，我们不能假设系统的每部分都是可以访问的（请参阅第 7 项），并且如果一个更新操作必须阻塞等待直到服务器完成重启操作，那么仅仅是一台服务器需要重启，也将会延迟整个系统，因此只有所有的服务器全部恢复在线，那么这个更新才能彻底完成。哦！

关于事务处理，情况则变得更糟——去除在某个服务器上的某个数据项上的锁，从道理上讲，应该去除在其它服务器上的同一数据锁，以避免同一数据在不同地方被更新成不同的值这种意外情况的发生。但是，我们会遇到一个现实的问题，两个并发更新中的每一个都要求在同

一数据项上加锁，并且由于线路中发送请求的响应延迟的缘故，发现那个数据项已经上锁——那么谁应该首先放弃呢？（顺便说一下，这种问题一般是博士的书面论题。并非是一个容易解决的问题。）

如果你已经阅读过第 5 项，你可能会突然意识到复制数据也会带来许多在这里讨论的、同样的一致性问题——你是对的。并且，遗憾的是，也没有真正的银弹解决方案能够解决这里的一致性问题。如果你能够接受数据更新传播中一定的响应延迟，那就不是那么困难了，但是遗憾的是，对于某些系统和/或表，不一致性是完全禁止的；对于这样的系统或表，复制并不是要追寻的光明大道。

注意，将从数据库返回的数据（或者任何其它数据，但似乎大多缓存处理的是数据库数据。）缓存起来仅仅是复制的一个子集，根据兴趣的不同，客户端会选择复制数据，而不是服务器。同样地，我们使用复制时也有相同的一致性和更新传播的问题，复制意味着如果你打算缓冲数据，那么在某人通过不是你所期望的信道来更新数据库的时候，你脑子中最好对此问题有个答案——记着，你不拥有数据库，正如在第 45 项所讨论的那样。

注意，我们也可以将上面讨论的 “数据”替换成“状态”来讨论有状态会话 Bean 和/或实体 Bean——集群中多台机器上的状态复制（或缓冲）特别难于实现。尤其是在有状态会话 Bean 中，询问你的供应商状态缓冲是怎样实现的——如果它们将状态推到“状态服务器”（正如几个供应商实现的那样），那么由于从状态服务器获取状态的开销（请参阅第 17 项），任何正被缓冲的事物很可能会丢失，除非以某种方式复制状态服务器，否则，你就又回到了集群本来就是要解决的单一故障点（状态服务器）的境地。

基本上，没有任何一个划分方案可以解决所有的问题。如果你要做分布，那么你应该将负载平均地分配到整个系统上（总之，应该像水平或垂直规则允许的那样平均），但是你会丧失复制方案能够提供的冗余。当然，如果你复制，你会陷入更新传播的问题，这很难解决。有时两种方案的结合被视为是最好的方法，因为分布数据不能承受任何不一致性，而复制数据则可以。

第 22 项：为了开放集成而考虑使用 Web 服务

企业级系统在本质上往往是异构的。它们可能在开始时并不是这样，但是在 IT 工业这种快速更新的动态环境中，很少有企业能够围绕它转，到最后就无法避免地形成了异构。即使你的上司完全赞同保持全 Java 环境，但是他或她的老板可能并不同意。或者，你所在的公司不使用 Java 开发，现在由你决定将两者整合起来。或者业务合作伙伴正需要访问一个你编写的系统，但是他们使用的是.NET。事实是，很少有企业完全是平台结盟的（使用相同的平台）。

有三种方式来融和外部技术：可移植性、迁移和集成。

如果某项技术是可移植的，那么我们只需在我们的环境中直接使用并运行它即可。在 J2EE 中，这意味着外部技术必须是一个 J2EE 能够兼容的构件，它可以插入到我们的容器中并且可以很容易地被重用，尽管在像 Python 代码可以在 Jython 的 Python-on-Java 环境下运行的这类情况中，它可能明确地不涉及到.NET 代码。

这里我们不能移植代码，但是我们能够迁移它——也就是说，用当前环境的技术完全重写这些代码。尽管在某些方面比较吸引人，但是，期望企业合作伙伴很乐意地把他们.NET 的功能代码重写成 Java 代码，相当不切合实际；事实上，他们可能反过来让你做迁移而不是反过来让他们做迁移。

这意味着我们只剩下了集成这条路了：设法只对实现做少许的修改，就能够将两个还像原来那样运行的构件整合在一起运转。嘿，这会难到什么程度？毕竟，我们数年前就已经可以使 Windows 系统通过 HTTP 与 UNIX 机器进行无缝的通信，并且反之亦然，对吧？

正如被证明的那样，对于任意两个平台——Java、.NET、COM/DCOM、 Perl、Python、Ruby、C/C++以及任何其它的平台——要获得它们之间的可互操作性实际上并不是如此困难。市面上存在众多的 Java 到.NET 的二进制的 RPC 互操作工具库。例如，Jython 处理 Java-Python 的互操作，JRuby 处理 Java-Ruby 的互操作，以及一些开源工具库使得 Java-COM 或 Java-C++的集成变得相当容易。

是到了需要我们去将这些变得纷繁复杂的事物全部实现互操作的时候了。

为了使跨越所有这些平台的互操作变得比较容易，我们需要一种世界语，一种每一个平台都能（或多或少地）按照原来方式进行交流的通用语言。于是 Web 服务进入了人们的视野，它利用 XML 和 HTTP 的普适性，以一种彼此都能理解的通用格式实现了这一目标，因此，不必为每个平台指数级别的连接器而感到担心，我们只需考虑和其中的一个进行连接（XML/Web 服务），并且每个平台自身要关心的只是数据怎么封装进 XML 和怎么从 XML 读取出来。

再者，Web 服务社区是由像万维网联盟（World Wide Web Consortium）、OASIS 和最新形成的 Web 服务互操作性组（WS-Interoperability group）这些标准组织所倡导的，它构建了一个有关在 XML 之上使得构建和使用 Web 服务更加容易的协议和标准的集合。除了这些以外，J2EE1.4 也采取了措施，通过将 JAX-RPC 整合为 EJB 的一部分，并将其它 JAX*规范整合到 J2EE 集合中，使得 Java 开发者可以很容易地集成 Web 服务。简言之，大致看来，好像你的可互操作性需求已经被考虑到了。

不过，实现这个特性需要众多的努力，事物表面之下并不像从上面看到的那样平静。对于发起者，我们有关于对象与 XML 信息集表示之间在对象层次上阻抗不匹配这个基本问题，关于 XML 的信息集表示在第 43 项有详细的介绍，这个问题使得对象转换成 XML 文本以及 XML 文本转换成对象都变得相当棘手。事物变得复杂了，J2EE1.4 假定从 J2EE 容器导出的 Web 服务端点应该是“RPC 编码（rpc-encoded）”格式的服务，而.NET，Web 服务集成的主要倡导者，则假定从暗箱里面出来的是“文档－字符（ doc-literal）”格式。尽管这并不是个无法攻破的难题，但仍需要更多的有关 Web 服务栈的知识，最为重要的是 WSDL，比大多数开发者可能对其的偏爱还显得重要。

遗憾的是，更大的危险潜伏在基本的前提假设下：所有的 Web 服务提供商都提供这样的能力，你可以从你自己平台相关的代码开始，翻转一下神奇的转换开关，突然间 Web 服务就为可以提供服务，等待你的享用了。考虑下面的会话 Bean 接口：

```
public interface TellerBean extends EJBObject
{
  public void
    generateBouncedCheckMessages(Set checks, Account acct)
    throws RemoteException
}
```

很明显，从第一个参数开始，这个 Bean 就在期望一个包含独一无二元素的集合——如果同一元素不知何故地插入到这个集合中两次，那么将会产生异常。就是这个问题：到底用于这个方法的 WSDL 定义应该是什么样的呢？

对于第二个参数，我们可以很容易地处理，因为 Account 仅仅是（期望是）对账号的简单封装，但是 Set 会引发某些问题。.NET 没有与 Set 等价的类型，并且即使有，也请记着：Set 仅仅是一个接口，它的实现可以有多种——可以是一个 java.util.TreeSet，可以是一个 HashSet，也可以是你自己定制的一个 Set 实现，它们不同实现方式是经由具体的行为，而不是通过接口表示出来的。期望产生 WSDL 文档的代码可能会将那个 Set 参数转换成一个简单的集合，这样会丢失 API 中一个重要的隐含部分，即集合的独一无二性。.NET 客户没有任何机会去避免意外地传送了 Bean 的副本，并且如果 Bean 被编写得是不会没有任何副本，那么我们可就要大费周折地调试一番了。

对于最好的互操作，Web服务应该是上下文完整的、数据驱动的通信端点（请分别参阅第 18 和 19 项），不过大多数EJB会话Bean接口没有按照这种方式设计。更糟的是，Java程序员在期望向会话外观[Alur/Crupi/Malks, 341]大批量地传送数据时（请参阅第 23 项），会选择数据传送对象[Fowler, 401]或集合，像Map或List。对对象层次结构数据进行映射时的限制会对数据传送对象产生影响，并且在XML中没有任何好的Map或List的等价物，这使得它们所包含的内容被转换之后会少很多。

简言之，如果你正打算让你的代码能够通过 Web 服务端点而被访问到，那么你就不能承担你的通信端点是以平台为中心的这样的假设所造成的影响。相反，你应该从某些平台不可知的事物开始——通常是数据类型的 XML Schema 定义，端点定义的 WSDL 文档，并且它们

两个都要在 Java 代码的第一行之前产生——正如 CORBA 对象要求首先定义 IDL，EJB 要求业务/远程接口在可以被编程实现之前要先被定义一样。这同时也意味着，一旦公布出来，这些接口就不能改变——比在 J2EE 环境中要求更高，Web 服务是让公众使用的，你不能假设如果你改变了某些 URL 或者这些 URL 预期的消息类型的定义，客户也要能够随着这个变化而变化。

在很多方面，Web服务是中间件软件的塔顶。记着，中间件应该成为连接彼此独立设计，并且符合Web服务定义的那些系统的“粘合剂”。所有这些因更高阶的Web服务协议而出现的规范都是中间件的焦点所在： WS-ReliableMessaging（Web服务可靠消息传递）、WS-Transaction（Web服务事务）、WS-Security（Web服务安全）以及WS-Addressing（Web服务寻址）反映了大多数中间件工具库在最近 20 年中所提供的功能。我们只是在它们两边加上尖括号（< 与>）。

像大多数企业中的其它事物一样，Web 服务提供了一些强有力的功能，在这里即和外部异构平台的互操作能力，但是，你不可能不付出代价就得到这些好处。如果你想使用 Web 服务，你就要确定它们物有所值——它们被设计来实现开放互操作性的，也就是说，同时与任意数量的平台进行互操作。如果你并不想提供与任何平台的互操作，那么选择针对你的特定平台的技术可能会更容易一点。

第 23 项：大批量地传送数据

考虑一下，如果你愿意，那么你的表示一个人的某些详细信息的普通实体 Bean（容器管理的或者 Bean 管理的，其实没有什么实质上的不同），可能来自一个美国本地化的地址薄应用。当从客户端使用它时，实体 Bean 看起来就象下面这样：

```
Person p = personHome.create(new SSNPK("555-12-9876"));
String fullName = p.getFirstName() + " " + p.getLastName();
String streetAddr = p.getStreet() + "\n" +
  p.getCity() + " " + p.getState() + " " + p.getZip();
```

```
. . . // Get some user input
p.setFirstName(firstName);
p.setLastName(lastName);
p.setStreet(street);
p.setCity(city);
p.setState(state);
p.setZip(zip);
```

很多熟悉 EJB 的读者，可以发现这些代码中的一个巨大错误——它太“饶舌”了，使得对这个实体 Bean 要进行多个方法调用才能够设置这个 Bean 实例所必需的数据。并且所有这些调用都是远程调用，这意味着你的每一个 get 和/或 set 调用都需要一个往返，因此它就严重地违反了第 17 项，这不仅把响应延迟引入到了应用中，而且还导致难于扩展，因为这个实体 Bean 正被要求去执行许多无法判断其往返开销的“小”操作。

这种逐个域的方法的开销不仅仅是在网络层，从概念上讲，六个 set 调用是作为一个单一业务操作的一部分而发生的，因此，应该有一个事务来保证在 set 调用之间不能进行修改。然而，实际上所有对实体 Bean 的调用都由它们自己单独的分布式事务保护，因此不是仅仅在单一事务中进行 6 次调用，而是每个 set 调用都创建了一个事务，征募数据源到事务中，设置数据，通过两阶段提交协议运行提交事务，然后将它消灭。我们必须经受 6 次这种并不简单的经历。

更糟的是，我们仍不能完全避免数据被污染；因为由容器管理的事务基于每个方法的，这里有 5 个窗口，在每两个 set 调用之间各有一个，另外一个客户就可以经由这些窗口进入去修改 Bean 里的数据。因此，想象一下像下面这样的场景会发生什么并不是是一件难事：

```
Client 1 wants to set bean to:
Ted Neward, 1 Artesia Way, Davis, CA 95616
Client 2 wants to set bean to:
Ted Neward, 1 Microsoft Way, Redmond, WA, 55512
Client 1 calls setStreet()
```

```
Client 2 calls setStreet()
Client 2 calls setCity()
Client 1 calls setCity()
Client 1 calls setState()
Client 2 calls setState()
Client 2 calls setZip()
Client 1 calls setZip()
Bean is now set to:
Ted Neward, 1 Microsoft Way, Davis, WA, 95616
```

显然这不是个好的事态。即使使用事务，即使使用全同步，我们仍然得到了语义上被污染的数据——该数据虽然按照语法上是合法的但仍然是不正确的。即使在华盛顿有一个名叫“Davis”的城市，它也不会有 95616 这个邮政编码——这个编码被保留给加利福尼亚州名叫 Davis 的城市所使用，并且，我还可以保证在 Davis 市没有像 “微软路 1 号”这样的地址。

面对这些，你的第一反应可能是通过去除客户端的事务来解决这个问题，但是正如第 29 项所展示的那样，客户端的事务是一个灾难式的决定，可能很快变成一件不幸的事情，没有任何有自尊心的 J2EE 程序员会愿意声称对此负责。我们只是想要执行这 6 个 set 调用，并且不需要补偿式的开销就可以去除事务。

通常的解决方法就是传送的块状数据要足够大以适合远程调用的开销。简言之，不要一次传送一个，而是大批量地传送：以完整的对象块，或者对象集的形式。（这正是 IBM/BEA 的服务数据对象（Service Data Object）解决方案的基本思想，并且可以论证这只比过程优先（procedural-first）的持久性层差一点，就像第 42 项所叙述的一样。）对于熟悉编组技术的人来说，我们想按值传送，而不是实体 Bean 和/或其它分布式技术缺省使用的按引用传送的方式。

办法之一就是创建数据传送对象（Data Transfer Object）[Fowler, 401]，这种对象的数据表示就是实体Bean或者持久性对象所表示的数据，或者至少是和它们接近的事物：

```
public class PersonDO
  implements java.io.Serializable
{
  public String firstName;
  public String lastName;
  public String street;
  public String city;
  public String state;
  public String zip;
}
public interface PersonTransferBean implements SessionBean
{
  // mandatory EJB methods left out for simplicity
  public PersonDO getData();
  public void setData(PersonDO data);
}
```

请注意PersonDO是怎样实现Serializable接口的——它可以确保在线路上发送PersonDO时，是按值编组而不是按引用编组的。这意味着我们不是发送一种存根给接收者，而是发送对象的完整备份，因此所有数据仍保留在目标JVM的本地。现在，是更新PersonBean实例上数据的时候了，我们可以一次传送所有的数据，而不是每次只取出其中的一小片。因为我们会失去系统的某些“面向对象性”，因为我们要放弃传统的、在一开始就已经成为Java一部分的获取/设置属性的惯例，但是通过这种方式我们至少能够避免严重的性能问题。更棒的是，我们可以将setData调用置于事务之下去执行，通过这种手段，可以确保六个调用遵循事务的ACID语义。

如果你不喜欢为每一个实体Bean类都编写一个序列化版本（而且我不能因为你没有这样做而责备你），那么请记住，现在的目的仅仅是在一个网络批量调用中传送所有的数据，而不是以任何有意义的形式来分割数据。因此任何按值传送批量数据的方式都是可行的，包括在

Java集合类的实现中传送数据，这些类的实现全部都是可序列化的，像HashMap或ArrayList，或者两者的综合——一个元素为HashMap对象的ArrayList，但这不是一种常用的方法。

这样传送数据的缺点是缺少类型安全。通过使用一个实体Bean一个传送对象类的方式，在这种方式中每个域都是强类型的（正如在任何Java对象定义中所描述的那样），编译器可以捕捉悄悄混入的输入错误，并且如果你试图访问lsat-Name域而不是lastName时发出警告。如果这个域名是Map的一个键，编译器不会验证你的代码（以避免输入错误），并且直到运行时你才能发现这个错误（希望验证代码是单元测试的一部分，或者是发生在QA处理期间）。

可供选择的是，你可以使用一个RowSet来传送数据，因为RowSet对象本身是序列化对象。RowSet接口实现中吸引人的属性之一是它们通常以非连接的方式操作。与ResultSet不同，RowSet不是通过持有一个激活的到数据库的JDBC Connection来大批量地获取数据；相反，它是将ResultSet中所有数据被拷贝到RowSet，因此RowSet就可以在本地拥有这些数据，因此一旦将RowSet序列化，所有的数据也就会和它一起被序列化。再者，JSR114 和J2SE1.5定义了一个标准化的RowSet接口实现，它具有很多有趣的特性，包括持有多条元组集（结果集）作为RowSet一部分的能力。

在此，有可能会问：关于可序列化的格式是否有什么特别的东西，答案当然是“并非如此”——它只是一种众人皆知的二进制格式。因此，正如第 15 项提到的那样，另一种可接受的可序列化格式是XML，因为其它平台，尤其是.NET，处理XML格式比处理它们原本并不熟悉的二进制格式要更容易。向着这个目标，Sun创建了WebRowSet。尽管没有正式成为J2EE的一员，但是从 2000 年开始，已经可以在Java开发者联盟（Java Developer Connection）的Beta版中获得它了，现在正在和其它几种RowSet接口实现一起，处于被JSR114 批准认可阶段。WebRowSet是对RowSet的一个实现，它继承自CachedRowSet并且添加了两种新方法——toXml和fromXml——每个方法的功能都如它们的名字所暗示的那样——：分别是将RowSet转换为XML信息集实例，和反向的从XML信息集到RowSet的转换。在某种程度上，这是一种专有的XML格式，但是至少还属于XML范畴，在和其它平台互操作时（请参阅第 22 项以查看更多关于互操作方面的内容），XML仍旧比某种专有的二进制格式要好。

顺便说一下，你要知道，当使用实体 Bean 的 CMP 实现时，仍旧完全有可能在容器和数据

库间进行多次往返——例如，根据事务关系被标识到实体 Bean“前端的”会话 Bean 上的方式的不同，你可能要分开调用，因为容器需要对每个调用都发起和终止事务（请参阅第 31 项）。在某些情况下，EJB 容器无法选择，只能为标识为 CMP 的实体 Bean 产生某些实在是死脑筋的代码，例如，对每个实体 Bean 的 get 调用都产生一个 SELECT，以及对每个实体 Bean 的 set 调用都产生一个 UPDATE，由于缺少提供给这个 Bean 的容器的任何标准提示，例如，可以被手动检测和使用的一个只读标志或者“脏”位。供应商常常提供这样的优化，但是利用这些长处会使你的代码与供应商相关（请参阅第 11 项）。正是由于这个原因，运行一个“能够监查的”JDBC 驱动或者使用数据库的监视工具来查看 SQL（请参阅第 10 项），对于准确知道 CMP 实体正产生多少次往返来说就显得至关重要。

批量地传送数据也有一些局限性——不断地在网络上传送大量数据并不比多次往返好多少，因为你现在正在吸干网络带宽，并强制通信层花费相当大的力气来编组、发送、接收、以及反编组数据。你还要基于客户端将怎样使用（或不使用）某一个特大的数据项，来判断它是据按引用传送还是按值传送更好；尤其要注意集合和可序列化对象，因为有了对象序列化规则，这使得一个对象的引用可能会招来一大堆你可能从来都没有想过要将其序列化的其它对象，而这种情况并不算少见（请参阅第 71 项）。请记着，我们的目的是避免过多的时间花费在网络上，而不要对某种方式或另一种方式教条化。

第 24 项：考虑定制你自己的通信代理

温习一下，像RMI这样的基于RPC的通信系统的优点之一在于：对开发者来说，很容易使用——它看起来像一个本地对象，闻起来也像一个本地对象，并且其作用还是像一个本地对象。所有令人生厌的通信数据都被本地对象——远程对象的代理[GOF, 207]所屏蔽。这种模式相当诱人，尽管与远程对象相关的问题有很多（详细细节请参阅第 5 项和第 17 项），它仍旧被广泛应用。

那么为什么要放弃它呢？

一旦我们认识到这种模式的诱惑来自于本地对象代理，就会很容易地意识到没有任何事物能

够阻止我们定制自己的代理，使我们有机会去提供我们所期望的任何优化和/或不同的实现。想使用 HTTP 作为通信的骨架？编写一个代理去序列化参数，然后将它们作为 HTTP 请求报文的报体运送到将其解包并执行该调用的 Servlet。想避免对象优先的持久性方案所带来的某些问题吗？编写一个代理，它采用的数据载入机制是积极载入数据，惰性载入数据，或者甚至一部分是积极载入数据，余下是惰性载入数据。

正如在第 23 项中所述，针对杂乱无章的分布式持久对象（也就是实体Bean）问题，给出的一种建议方法是大批量传送数据，因此对底层数据存储的全部更新可能刹那间发生，这样就适合远程调用的费用。大批量传送数据的问题是它通常会去除系统的“对象性（object-ness）”，也就意味着客户必须从逻辑上去操纵数据传送对象[Fowler, 401]和用于持久化与修复的API之间的不同。如果你需要一个更“对象化”的方法来多保存一点系统域模型的对象性，一种可供选择的方法就是使用半对象（Half-Object）增强协议[AJP, 189]，并创建“智能数据代理”，它能够将数据缓存在本地直至客户端做好一次性提交全部数据的准备。

你很可能会问为什么这种优化仍没有就绪，尤其是关于 EJB 实体 Bean——如果实体 Bean 仅仅是一种以对象优先的方式来对数据建模（请参阅第 40 项），那么像实体 Bean 存根应该以某种方式优化数据到服务器的传输这样的说法，很明显是一种保守的说法。在重申一遍，这里的问题是 EJB 规范的措辞问题——因为访问实体 Bean 的每一个属性都必须要在一个事务的支持下进行，这个数据必须压送到底层的数据存储。（请记着，在服务器万一发生崩溃时，对实体 Bean 所做的修改必须保存。）因为当前的 EJB 规范不允许“本地缓存”的概念，所以任何提供这种优化的 EJB 容器都被官方认为是不兼容的。更不必说一个没有什么用处的优化——请确保你阅读了第 11 项，并且在决定使用它之前，判断供应商的中立性对你是否很重要。（注意如果你是将实体 Bean 作为 Bean 管理的（BMP）来编写的，你就可以适时地使用这种优化，不过现在你基本上要再次编写整个代理了。）

实际上，很多供应商和开源的项目正在着手这样做；例如很多 JDO 供应商现在能够通过一个 JDO 调用来返回对象的“一部分”，这样就消除了我针对对象优先的持久层所提出的主要批判（请参阅第 40 项）。这个迟到的新生事物很受欢迎，因为这意味着你不必自己动手去创建这些灵巧的数据代理，正如在这里建议的那样。然而，如果你的特定供应商和/或工具库不能实现你所想要的，这就变为低效运行了。

数据访问只是一种可能性。

大多数EJB容器都利用RMI进行EJB客户端和EJB容器本身之间的通信连接。RMI是一种很有用的对象RPC系统，但是这个由Sun提供的参考实现有一个特别难以处理的缺点，那就是它与J2EE环境中我们用意正好相反：由RMI服务器返回的RMI存根被隐式地“固定”到了一个特定服务器上。换句话说，你获得的这个存根仅仅知道怎样和返回它的那个服务器对话，并且如果那个服务器由于某种原因已经停止接收进入的RMI调用（可能服务器已经崩溃了），那么你的存根现在就变得没有什么价值了。即使在容器分布到集群中两台或更多台机器上的情形下，你接收到的RMI存根也不能和集群中的其它机器对话；你必须通过从Home对象请求一个新客户端来与新存根重新交谈。因此要将它封装到一个代理中。

建议使用智能代理的其它可能的情形包括创建一个总是进行JNDI查找的Home，来保持位置独立性，或者缓存一次JNDI的Home查找的结果以避免网络上到容器的往返，或者 24 小时后让被缓存的JNDI查找超时等诸如此类的思想。

你不应该想着要对你系统中所有可能代理的地方都做这样处理，很显然——这意味着大量的工作。只有在存根或代理返回给你的缺省行为并非你之所想，并且创建智能代理的开销与所节省的网络通信量相比是合算的，或者具有更好的错误转移行为，以及具有其它一些切实可测量的好处的情形下，才应该使用智能代理。

第四章 处理

你必须放下两方的所有偏见，既不相信也不排斥任何事情，因为其他任何人或者这些人的描述，都已经排斥或相信它了。你自己的理由是上天给你的唯一神谕，对此你负有责任，不是为了抉择的公正，而是为了抉择的合理。

——Thomas Jefferson

不要去找精灵商议，因为他们既说是又说否。

——Gandalf 对 Frodo 说, 指环王

处理是在企业系统中中间件不能为我们解决的那一部分，是我们应用的“血肉”。它有过：业务规则、业务逻辑、领域逻辑、应用代码，等等无数多的称呼。正如它的本质所言，处理是业务首先要关注的东西：对没有在其它任何地方处理过的数据进行处理。

企业系统中相当大一部分都是在处理并发——考虑到最大的吞吐量，我们需要假定两件事情可以在同一时间发生。无论硬件速度多么快，如果我们试图运行一个系统，该系统对于系统中所有用户来说，每件事情都是在同一时刻同步发生的，那么这个系统很快将会陷入困境。因此作为程序员，我们需要处理多线程执行这个事实，无论是在逻辑上还是在物理上，现实的确如此。遗憾的是，这说起来容易做起来难。

在企业 Java 系统中，并发同时涉及到两个主题：Java 自己的基于监视器的对象同步系统和构建于大多数企业资源管理器内部的基于事务的同步机制。尽管如果我们直接忽略其中一个，而倾向另一个就会很容易处理，但是在此时是不可能的——Java 语言（和/或虚拟机）没有在其内部构建事务性的能力，并且试图“隐藏”资源的事务本质的做法已经多次被证实会产生瓶颈。因此，我们需要意识到并发的这两个方面。

想要更多地了解有关Java基于监视器的对象并发机制的细节，我强烈推荐《Java并发编程（Concurrent Programming in Java）》[Lea]这本书。另外，我假定你已经阅读过《高效Java（Effective Java）》[Bloch]的第 48 项和第 53 项；例如，由于在Bloch的书中第 49 项所叙述的原因（“避免过多的同步”），你永远都不会将Servlet的doGet或doPost方法标识成同步的。

你会很快注意到本章的许多项都是涉及到了事务性的处理；其它所有的处理（像基于规则的处理或者工作流）都明确地处于次要位置。尽管这显得有一点过分强调了，但是企业级IT系统的大部分都隐式地或显式地是某种形式的事务处理系统，并且作为一个企业级Java开发者来说，这些项也直接与你的日常生活有关。即使你当前没有使用EJB，也请你继续读下去——不但你将来会用到它们（你的系统如果成功的话，那么很可能会和多个数据库同时通信，因此依据第 9 项，这也就为EJB创造了成熟的使用环境），而且无论是通过企业Bean，还是直接通过Connector或供应商专有的 API访问企业资源，这些相同的概念都会得以应用。

第 25 项：保持简洁

KISS（Keep It Simple, Stupid，保持简洁易用）。要做那些能够让系统运转的最简洁的事情，这就是简洁性规则。它有过几个称呼，但是它们全都归于同一思想：在你的代码中倾向于简洁而不是复杂。

这一项并不是必须在这里讲述，但令人惊奇的是，架构师和技术带头人是如此经常地着迷于他们的聪明才智（我也不愿意承认这一点：肯定也包括你的架构师和技术带头人。）“我们要做的所有工作就是将数据传送到这个复杂的处理对象系列中，接着它们会全部联合起来产生我们期待的结果。它就是如此的酷。它是经典的 Blackboard 模式与 Visitor 和 Observer 模式的组合，还有一些作为额外增添之物而被采用的 Composite 和 Transaction Script 模式……”遗憾的是，他们忘记了这样的复杂性毁掉了所有的东西。我的一个朋友有一次这样对我说：“每个项目都有一个复杂性预算：只要在你的预算中考虑到了所有的复杂性，那么那就是复杂性预算。也就是说，如果你试图超出这样的预算，项目就会失败。如果限制在这个范围内，项目就会成功。”记着，上述复杂性的某些部分需要被应用到领域问题本身，因此如果你将全部复杂性都花费到了复杂的技术上，以至于超越费用底线，那么该项目将会崩溃。我现在意识到在我的开发经历中不止一个项目完全是因为这个原因而失败的，并且我想任何曾经构架或者领导过不止一个项目的人都可以这样说；当然我所知道的在同一位置的每个人都已经这样说过很多了。

如果你不能解释一个给定的由一条单一语句，或者至多两条语句构成的处理代码片段做了些什么，那么它就可能过于复杂了。

复杂解决方案存在的问题数也数不清，但是基本上都可以归咎于下面几个基本直接原因：

- *复杂解决方案很难被模块化，因此难以被重用*。一个复杂的解决方案通常试图在它自己的边界内来解决每件事情，没有留下和代码中其它部分组合的余地。换句话说，当必须考虑计算的复杂级数时，充分利用在代码中构建的挂钩（如果有的话；请参阅第 6 项）就要困难得多。
- *复杂解决方案难于调试*。这是一条众所周知的法则，可以追溯到C和UNIX的早期："规则 3. 奇异算法比简单算法会有更多的bug，并且更难实现。请使用简单的算法与简单的数据结构" [Raymond, 12, 引自Rob Pike的《C程序设计笔记（Notes on Programming in C）》]。如果没有其它因素，那么复杂解决方案要考虑更多行数的代码，要理解更多的类，你的头脑中要记着更多的对象间的交互。而通常的法规是，人类能够清醒记住 3 到 7 件事情——多于这个，我们就开始记不清细节了。
- *复杂解决方案难于被优化*。如果没有先进行性能分析（请参阅第 10 项），你可能永远不能进行优化，一旦你已确定在复杂纠结的方法调用、对象创建以及双重调度逻辑中的某个地方存在一个瓶颈，你就需要决定该怎样去优化它。对于简单的算法和/或类，很容易发现应该在什么地方优化，或者它是否可能被优化。遗憾的是，在进行优化的过程中，简单往往会变成复杂，因此你要确信劳有所获。
- *复杂解决方案更难于维护*。我们已经全在这儿啦——在一个类或模块中有一个 bug，现在到了该指派一个工程师处理它的时候了，每个人都会抱怨并挥手："我决不想被牵涉进去，那里的东西乱作一团。"如果代码几乎绝望地需要被重建，那么 10 次中有 9 次都是因为它过于复杂了。

肯定还有其它一些原因一时没有想到；这些原因只是最常被流行文献所引用的原因。

复杂性问题几乎都是因处理而产生的，而且我们编写的处理代码大多数是与领域相关的。灵活多变地使用 J2EE 规范很诱惑人——例如，发现 2.2 版本的 Servlet 容器有一个 Request Dispatcher 功能，于是，禁不住诱惑多次使用 forward 方法或者多次在其它 Servlet 上调用

include，而对于涉及到整个过程的处理，每个“subservlet”只实现其中的一部分。一定要抵制住，因为当你这么做时，会把一系列相当大的复杂性引入到系统中，例如错误处理和输出产生。

复杂性通常是在处理“特殊情况”的逻辑时就会表现出其丑陋的一面——例如，“如果我们的客户有多于 100 条未处理的订单，那么在订单生效之前，我们要让超级用户快速地把它们处理掉，”“将提交这一行到订单表之前，要确保信用卡支付是通过检验的，因为我们可从来都不想把没有付款的订单项交付客户，”或者甚至像“如果客户不在数据库中，可以继续进行，并使用‘缺省’的客户项，因为我们不需要客户为了购买东西，而必须在我们的数据库中”这样的情形。

直接在代码中处理这些情况的建议比较吸引人。Fowler将使用特定类的子类这种思想归档为特殊情况模式（Special Case pattern） [Fowler, 496]，在这里，类Customer以及特殊情况子类MissingCustomer 和 UnknownCustomer，都是领域相关的，并且都相当简单；不过子类这种方法在前面提到的以处理为中心的例子中运转得并不是很好。

对于客户有 100 个未处理订单的例子来说，基于规则的引擎通过将你从强制式的程序设计模式中解脱出来，然后让你以更加声明的形式去运转（细节请参阅第 26 项），以此来简化你的处理。另外，第一个例子也暗示着不仅仅是一方（人类或其它事物）在处理这个特殊请求时需要被处理——在企业领域内通常称其为工作流（workflow），而且尽管昂贵的或开源的工作流引擎都可以获得，但是你通常可以通过采用一种基于消息的处理模式来轻松地处理工作流。对于在提交订单之前信用卡支付的例子，寻思使用事务处理来确保以原子的和相当透明的方式（细节请参阅第 27 项）去处理错误场景。对于最后使用“缺省”客户的例子，特殊情况的一种[Fowler, 496]变形也许可行，返回的Customer是数据库中一个真实的客户，它除了存储的数据没有什么意义或者是作为数据库驱动的Null对象[PLOPD3, 5]之外，和其它的Customer没有什么不同。

注意，在所有这些情况中，“在代码中自行处理”这种措辞从来没有出现过，直接在你的处理逻辑中实现特殊情况的代码是产生复杂情形的“最佳方式”；这似乎有点违反直觉，不过最好的代码行确实是那些你不需要编写的代码行。

通常，企业级项目倾向于复杂而不是简单：复杂系统比简单系统更具“男子气概”这种想法是不对的。可能是某种无意识地渴望在系统中构建一点对这份工作的安全感，或者是构架师的强制要求以表明了他或她自己是舞台上的大玩家。但无论出于什么原因，我们都应该不断地重复强调：保持简洁，让工具（在这种情况下，就是J2EE和它的相关规范）去完成尽可能多的工作。

第 26 项：优先采用规则引擎去处理复杂状态的评估和执行

业务或领域逻辑的复杂性经常会达到难于用Java这样的命令式语言表示的程度，如果不是完全不可能表示的话。纯化论者将会毫无疑义地嘲讽那种“存在某种万能对象都无法解决的事情”的思想，但基本的事实是业务规则并不总是落在面向对象的范畴内，我们最终是通过用“验证”或“处理”方法创建Blob对象来覆盖这些规则的——从本质上说，由于缺少更好的方法，我们只能回过头去求助于过程式的某些习惯。

请考虑一下，你桌子上的计算机以及不同的生产商为这台机器模型而提供的所有可能的不同硬件组合——内存、硬盘、显示器，更不用说其它外围设备——以及它们之间不可避免的不兼容性。（“如果他们定购DVD，那么我们需要确保他们得到的不是KorSplatt 5900显卡，因为这种显卡不能和此DVD模型一起工作，当然除非他们想更换为CD-RW/DVD。噢，KorSplatt 5900 不能在一台少于 512MB RAM的机器上运转，除非它是SuperReallyFastRAM……”）。由于所有可能的“公司希望在基本不兼容的限制之上仍然能够运行”的促销策略的因素，再加上所有这些东西一个月（如果不是一个星期的话！）就会变动一次，以及突然冒出来的要试图为在线PC生产商创建“计算机配置器”的想法，都使得大多数坚毅的IT行家望而却步。

这里的问题是这种复杂性评估很难用Java这种命令式语言去执行，这种语言关注的是CPU一步一步的实现。从本质上说，我们告诉机器的是怎样做这项工作，因此这意味着我们必须对各种需要被评估的条件以及这些条件需要被考虑的顺序非常明了。这会导致产生像下面这样复杂和难以维护的代码：

```
if (currentPC.drives().contains("DVD"))
{
  if (currentPC.videoCard().equals("KorSplatt 5900") &&
      !(currentPC.drives().get("DVD").equals("CD-RW/DVD")))
  {
    warn("DVD incompatible with KorSplatt 5900");
  }
}
else if (currentPC.videoCard().equals("KorSplatt 5900") &&
         currentPC.memory() < 512)
{
  // . . .
}
```

我们再次可以讨论，尽量将高阶不兼容抽象出来并将它们重构到更为一般的面向基类的代码中，然而真实的情况时，在许多场合，业务创建了以产品为目标的规则，这些规则对于工程师来说没有什么意义，但是能够帮助业务在一个季度内增长 5%。

这种类型的处理是一种特殊的编程模式，称作基于规则的程序设计，并且幸运的是我们能够得到这些规则引擎，这种软件可以检验数据集，查看在引擎中声明的规则列表，然后推断应该引发那条规则以响应数据的当前状态。更重要的是，规则引擎接着可以在必要时重新应用这些规则到发生了变更的数据上，直到数据到达一种稳定状态不再触发更多的规则为止。

面向规则的实现已经出现很多年了，并且在消费者的使用中变得更加广泛。普遍存在的用于处理 e-mail 的发送邮件程序就是使用隐藏的规则集来决定怎样根据每条消息转发邮件的。垃圾信件过滤使用规则评估一个收到的邮件信息是来自于你妈妈的新年祝福还是来自于匿名的浪费你的感情的 e-mail 地址。

顺便说一下，为了避免你会认为这是一种可疑的应用，我们给一个最先出现的并且也是最著名的基于规则的应用实例——数字装备公司（Digital Equipment Corporation）的XCON系统，它完全实现了这一点：帮助DEC销售顾问配置DEC大型计算机订单，以确保订单已经有了所需的全部零件和附件[1]。在 1989 年，XCON涵盖了 17000 多条规则，通晓 31000 种硬件需求，随着精确度的渐增，DEC每年估计节约了 4 千万美元，因为他们降低了测试花销，提高了客

[1] 有关DEC XCON系统的讨论基于来自Jess in Action[Friedman-Hill, 10]的信息。

户满意度。如果你不是在销售电脑的行业里工作，那么基于规则的系统对其它各式各样面向领域的任务也很有用，包括基于先前购买历史的“客户推荐”，这是无论哪里的市场部门都开始寻求的一种面向大众的购买系统。实际上，稍做分析，你会发现几乎没有任何IT系统，其内部没有任何种类的基于规则的处理。

规则引擎通常服务于两个目的：（1）以最好的方式捕获业务规则（2）允许修改这些规则而不需要重新编码Java代码本身。第一个目的通常是最大的收获，因为在一大堆的if/then/else语句中去尽力追踪业务规则不仅难于实现而且易于发生错误。然而，如果你的用户足够老练，那么教会他们规则引擎能够理解的“规则语言”还会有一个附加的好处，即给予了他们修改应用的业务逻辑的核心部分的能力，从而有效地将程序员从业务逻辑改变的循环中跳出，因为业务逻辑的改变可能需要另外一个完整的开发周期（开发、测试、QA、发布、配置等等）。

有几种不同的实现：一种是开源的实现，叫做drools，可以从http://www.codehaus.org上获得；BEA也包含了一种实现，并将其作为WebLogic平台的一部分，ILOG也在销售一种用来整合其它服务器的第三方的引擎。JSR 94 定义了标准的API（在javax.rules包内）用于连接规则引擎；它的参考实现叫做Java专家系统外壳（JESS, Java Expert System Shell），可以免费用于非盈利目的。解释说明整个JESS语言以及API远远超出了我们在这里所能涵盖的范围；要想查看更多的细节，请参考JESS in Action [Friedman-Hill]。

使用符合JSR 94 的规则引擎要像任何其它的J2EE资源那样遵循一些同样的基本原则，那就是开始时要通过JNDI查找获得一个RuleServiceProvider实例。初始化规则引擎需求创建RuleExecutionSet，一个准备执行的载入规则集。使用RuleAdministrator实例注册到RuleExecutionSet，这样你就已经做好准备去使用它了：

```
RuleServiceProvider provider = // . . .
  // Either use a JNDI lookup, or else use
  // RuleServiceProviderManager, similar in concept
  // to the JDBC DriverManager
RuleAdministrator admin = provider.getRuleAdministrator();
RuleExecutionSet ruleSet = null;
// Load the rules from a local file using
// LocalRuleExecutionSetProvider
```

```
//
FileReader reader = new FileReader("rules.xml");
try
{
  HashMap props = new HashMap();
  props.put("name", "Configuration Rules");
  props.put("description",
          "Rulebase for company PC configuration");
  LocalRuleExecutionSetProvider lresp =
    admin.getLocalRuleExecutionSetProvider(props);
  ruleSet = lresp.createRuleExecutionSet(reader, props);
}
finally
{
  reader.close();
}
admin.registerRuleExecutionSet("rules", ruleSet, props);
```

rules.xml 文件正如它的名字所暗示的那样，包含一些用面向 XML 的格式表示的规则。JSR-94 规范没对规则语言本身做出任何强制规定；XML 格式完全是与基于 JESS 的参考实现相关的。其它规则引擎也可能会提供一种类似 XML 的格式或使用其它完全不同的事物。

一旦引擎的初始化完成，在运行时使用它就显得相当简单直接了。JSR-94 定义了两种RuleSession类型，有状态的和无状态的，大致与EJB的会话Bean的有状态和无状态的差异相对应：有状态的RuleSession在多个调用之间保留有它自己的工作内存，而无状态的RuleSession则没有。这意味着在无状态的RuleSession中，那些（为了评估规则）被添加到工作内存的对象，在规则被评估之后，即消失了。

```
//Create the RuleSession instance
//
RuleRuntime runtime = rep.getRuleRuntime();
StatefulRuleSession srs =
  (StatefulRuleSession)runtime.createRuleSession("rules",
      props,
      RuleRuntime.STATEFUL_SESSION_TYPE);
// Populate the RuleSession's working memory with the data
// to evaluate against
//
srs.addObject(new Integer(12));
```

```
srs.addObject("Hello, world");
srs.addObject(new CustomDataObject());
// Execute the rules
//
srs.executeRules();
// Examine the entire contents of the working
// memory—objects may have been modified by
// executing rules
//
List results = srs.getObjects();
// Release the RuleSession
//
srs.release();
```

依赖于规则本身的内容，结果List将会包含由在rules.xml文件中的规则体所创建和修改的对象。

规则引擎，当正确使用时，可能会是一种不需要一大堆if/then/else语句来表示业务逻辑的强大方法；不过它仍旧是另一种要学习的语言，并且在你的业务分析师或者终端用户不会并且将来也不会学习这种语言的情形下，那么任务就落在了你的肩上。与SQL和 XSLT很相似，大多数规则语言本质上是命令式的，意思是你不必像指定*什么时间去做*和*应该做什么*那样指出*怎样去做*——也就是，你指定某种谓词条件，去说明触发的规则和在规则触发时应采取的动作。例如，在构成完备的类似XML的规则语言中，一条用于PC配置器的规则可能像这里展示的代码那样，在这里，条件元素是某种类似XPath的查询语法，行为元素像某种脚本语言。

```
<rule>
  <condition>(drives[@type="DVD" and @type!="CD-RW/DVD"] > 1) and
           (video[@mfr="KorSplatt"] and
           video[@version="5900"])
  </condition>
  <action>
    put("KorSplatt 5900 is incompatible with regular DVD")
  </action>
</rule>
```

尽管不需要将对象替代成我们用来构建企业系统的主要的通用目的的语言，但是当处理复杂

的条件逻辑时，规则引擎的确能使你的生活更加轻松。尽管 JSR-94 不是 J2EE 规范集合中的正式部分，但它最初仍是为 J2EE 应用而设计的，并且现有很多规则提供商都是专家组（Expert Group）的成员。至少，这仍是你应该关注的事情。

第 27 项：优先为隐含的非原子性错误场景采用事务性处理

作为一个程序员，我们习惯于那种使用隐含地是原子性的语言去实现各种操作的模式：也就是说，被调用的函数要么返回正确的值，要么返回某种错误代码或异常。更重要的是，我们假定它返回的要么是完全完成了的结果，要么是什么都没有——例如，不会从Math.sqrt方法返回任何像“部分”答案这样的东西。它要么是一个正确的平方根，要么是抛出某种异常（比如，我们求-1 的平方根）。

遗憾的是，这种原子性操作完全是一种假象。在我们编程的高层几乎没有任何事物是真正原子性的。例如，当我们调用Math.sqrt方法时，我们所用的执行线程依次开始执行在Math.sqrt定义内的代码，将这些代码再依次分解成单独的CPU指令，在CPU上依次连续地执行这些指令，而CPU肯定周期性地请求总线从RAM获取某些额外的数据，等等。相反，很多工作是“在遮盖下”完成的以呈现出这种视觉上的原子性——实际上，就是这语言和/或编译器的大部分工作，极大地掩盖这种复杂性，因此就不会“妨碍”到我们真正想做的，在这里就是计算出平方根。

幸运的是，只要我们所做的任何事情都是只读的，那么表示这种程度的原子性就是一项相当简单的操作；无论是什么事情出问题了，我们可以只是抛出异常并将其整个操作忘却即可。然而，如果被执行的方法作了某些事情以改变调用它的对象的内部状态（如果是静态状态则是类的内部状态），那么事情就不是那么简单了。面临错误时，我们是否应该将对象/类的状态返回到它的初始值呢？（可能吧。）我们依次调用的方法又该怎样处理呢？它们也要这样做吗？

考虑一下*Effective Java* [Bloch]中的第 46 项：“争取实现错误原子性。”适用于简单Java编程的东西会更加适合Java企业级编程。实际上，在企业级编程中更难实现原子性，因为在这里

考虑的不仅仅是Java代码的问题——我们必须也要考虑JVM之外的一些资源（例如数据库表、消息队列、文件系统中的文件，等等）。

那么就请进入事务性处理吧。

从编程人员的角度来看，事务的基本模型很简单：你依据资源管理器创建一个事务，其中资源管理器就是你想进行处理的事物。事务性资源管理器当然就是关系型数据库，但是其它资源管理器当然也是有可能的，包括但不仅限于JMS消息代理、遗留的大型机系统或其它用Connector访问的系统，甚至可能是底层的文件系统，如果文件系统支持的话。在某些情形下，雄心勃勃和对错误洞若观火的程序员甚至可能编写他们自己的对象以实现Java事务API的资源管理器部分，因此这使得你使用的对象在本质上具有了事务性。

一旦事务被开启，你就可以把你想实现的工作作为事务的一部分来编写。这是一种与资源相关的工作，例如执行一条 SQL 语句、获取消息、向遗留系统发布一个请求，或者是其它的什么工作。当所有这些工作完成时，你就可以选择*提交*事务，这样就可以将你所提议的修改“持久化”，或者回滚事务，有效地抛弃掉从事务被启动时开始对资源所作的任何修改。（如果支持的话，你可以选择抛弃所做工作的一部分——细节请参阅第 36 项。）

换句话说，你所做的修改要么全部成功，要么全部失败。你就不需要处理“部分失败”或“部分成功”问题了。如果第三或第四步突然由于某个原因失败了，那么你并不需要编写代码明确地撤销在方法前面的步骤中所作的动作——以实现要么做完每件事情，要么什么都不做。并且你这部分工作并不需要任何其它额外的工作，以使这种原子性错误场景成为现实；这些就是事务处理模型的全部。

考虑下面的将这些都应用到了实际场景中的代码：

```
public class Person
{ . . . }
public class Minister
{
  public void marryPeople(Person spouse1, Person spouse2)
```

```
    {
      System.out.println("Do you, " + spouse1.getName() +
                         " take " + spouse2.getName() +
                         " to be your spouse?");
      if (spouse1.isOKToMarry(spouse2))
      {
        System.out.println("How about you, " +
                           spouse2.getName());
        if (spouse2.isOKToMarry(spouse1))
        {
          System.out.println("If any here know why these two " +
                             "should not be married, " +
                             "let them speak now " +
                             "or forever hold their peace.");
          if (Crowd.isOKWithMarriage(spouse1, spouse2))
          {
            System.out.println("Kiss, " + spouse1.getName() +
                               " and " +
                               spouse2.getName() +
                               ", I now pronounce you
                               married.");
            spouse1.setSpouse(spouse2);
            spouse2.setSpouse(spouse1);
            System.out.println("Let's party!");
          }
        }
      }
    }
```

从表面上看，这是一个为系统中两个Person对象举行传统美国婚礼庆典的颇为简化的版本。我们通常采用“询问每个人是否同意这门婚事”这种典礼，先问其中一个人，接着再问另一个，如果他们无论怎样都喜欢他们法定的婚姻，那么就一起转向人群。如果每个人都同意，我们就宣布这对新人结为夫妇，晚间宴会，然后将这对幸福的新人送到夏威夷或百慕大或迪斯尼世界，开始他们的蜜月之旅。

（实际上，传统婚礼是分布式事务管理中分布式两阶段提交协议一个极佳的例子：阶段 1 是“投票”阶段，在这期间所有和事务有关的资源——也就是这对新人，观众也是——逐个被要求提交事务，阶段 2 宣布投票结果。如果人群中任何一个说“不，”那么处理就结束了；

否则幸福的新人就会前往百慕大，详见第 32 项。）

这里有多少个可能的错误场景呢？在继续之前先考虑一下这个问题。

准备好了吗？我信手拈来地列举了下面几个：

- 我们关注一下对两个配偶的询问和稍后在代码中“翻转结婚状态位”的操作之间的并发性。举例来说，如果相同的配偶被用于另一个典礼，问题完全有可能就接踵而来，因为线程往往是并发进行的。这是一个简单的Java并发问题，但是解决起来并不是那么简单——我们不能仅仅将方法本身标识为同步的，因为这样仅仅同步了Minister对象，而不会对正在被直接使用的Person上锁。并且在Person类上设置同步方法也不能解决这个问题，因为交错的调用是问题所在，而不是同一对象被两个方法同时调用。
- 由于上面的问题，如果从spouse2.setSpouse抛出AlreadyMarriedException异常，那么将会发生什么事情呢？我们实际上从没有将spouse1重新设置为单身或者是将作为spouse1的配偶spouse2 移除，留下可怜的spouse1和一个从没考虑过他或她要结婚的人结婚。这种情形可能是编写肥皂剧或人类戏剧的很好素材，但对于计算机系统来说它们通常是有害的。
- 一个微小的变体：如果从同一位置抛出OutOfMemoryError（或其它的error或RuntimeException）异常，将会发生什么？同样，也没有将这对配偶中的任何一个重新设置为原来的初始状态。
- 当然，我们甚至没有考虑对DJ、花匠、教堂或主持婚礼的人劳动成果进行支付的问题，即使婚礼失败——尽管新郎和新娘突然从婚礼上逃跑，这些人毕竟仍期望他们的劳动有所回报。我们需要通过在isOK...方法中返回false来处理一个配偶、两个配偶甚至是人群阻止婚礼进行的各种情形。尽管在这种情况下这并不是什么问题，但是理论上我们要处理半打可能的错误场景（考虑所有的spouse1、 spouse2 以及crowd决定的变动）。

我敢确定这里有更多的可能性——你只需要稍稍想象一下。

现在考虑一下我们应该怎么在某种事务性系统中编写实现这一点。（这里我做了一下主观臆断，因为 Java 不支持 box 之外的事务性处理，但是其基本思想被证明是相同的。）

```
public class Person { . . . }
public class Minister
{
  public void marryPeople(Person spouse1, Person spouse2)
  {
    Transaction txn = TransactionManager.getTransaction();
    txn.begin();
    // Enlist all the players who get a say in the results
    //
    txn.enlistResource(spouse1);
    txn.enlistResource(spouse2);
    txn.enlistResource(Crowd.getCrowd());
    try
    {
      if (spouse1.isOKToMarry(spouse2) &&
          spouse2.isOKToMarry(spouse1)&&
          Crowd.isOKWithMarriage(spouse1, spouse2))
      {
        spouse1.setSpouse(spouse2);
        spouse2.setSpouse(spouse1);
        goOnHoneymoon(spouse1, spouse2);
        // Pay for the wedding services
      }
    }
    catch (Exception x)
    {
      // Nothing to do here—spouse1 and spouse2 are returned
      // back to their original state automatically
    }
  }
```

注意，万一出现了错误——任何类型的错误，也不需要任何其它工作将系统恢复到它的初始状态，因为对象对事务是有意识的，所以它们可以重新设置它们自己。（如果你没有做好准备去接受这种对象对事务是有意识的思想，那么就请把它想象成存储过程而不是 Java 代码——最后的结果是一样的。）

按照这种方式来看，很难不为编写事务性代码而感到兴奋，尤其是那些比上面场景更复杂的代码。当你考虑这一点时，它就像是从来不必说“对不起”在 Java 中的等价物一样：为了那些被处理所触及的所有事情，资源管理器可以很神奇地以某种方式把系统的状态恢复到你

启动时的状态。

遗憾的是，事务处理也遭到了一些责难。一方面，它被认为是“旧有的模式”，因为它是许多更老式的大型机系统的核心原则（“事务处理”是“事务处理监视器”和“事务处理系统”中的“事务处理”）。另外一方面，它被认为是只有数据库管理员才能在某个规则基础之上进行处理的事情，或者是充满神秘的规范和难辨认的实现的“让人头疼”的事情。事实上，这种复杂性的大部分都被隐藏在了事务性表象的后面——对大多开发者来说，事务性系统比任何可能的替代物（手动编写其代码）都更容易使用。

如果你需要处理作为这个工作单元一部分的其它资源——例如，你不但需要改变用户账户，而且还要将所做的改变映射到一个独立的数据库中审核日志的一部分，那么事情就变得尴尬了。这正是分布式事务开始起作用的地方，并且尽管它们使得你可以对到多个资源管理器的操作（也就是说，不止一个数据库）的处理仍旧是原子性地成功或失败，但是它们会产生它们自己的开销，就像第 32 项所述一样。

如果你要马上进行试验并开始寻找方法，以使你的对象在本质上成为事务性的，正如在前面的 Java 例子中假设的那样，那么请记住，没有完全免费的东西，事务也毫无疑义地会带来它那部分的开销。首要的是，事务性处理不仅仅只是被用于产生原子性的错误，而且也是一种并发模型，意味着任何时刻在某个资源上开始的事务，必须要给该资源上锁以确保系统中其它处理不能同时对它进行更新。实际上，事务性系统通常提供 ACID 性质。这些锁，如果没有被正确使用（请参阅第 29 项和第 30 项），将产生一个只能被描述成是一场可扩展性灾难的系统。

不过，从长远来看，一旦考虑到可扩展性，事务性处理为编写以简洁的、一致的、连贯的方式自动处理错误场景的代码提供了强大的机制。并且最后要提到的一点是，它也是保持简洁的有力工具（请参阅第 25 项）。

第 28 项：区分用户事务和系统事务

一个用户正在使用在线银行系统。她只是想进行简单的余额过户，从她的现金账号过户到她的支票账户（可能包括她今天稍早时候签过的大额支票）。她选择菜单中余额过户（Balance Transfer）选项，从显示的账户列表中选择她的支票账户，输入她想过户的钱数，从显示的第二个账户列表中选择她的现金账户，然后点击执行（Go）。

这里到底用到了多少个事务呢？

问题并不像看起来的那样简单。对于用户以及任何一个会计学的学生来说，这似乎好像是只有一个事务，但是对于系统来说，可能是有两个数据库的事务，尤其是如果我们正在和多个数据库或其它资源通信：例如，为了帮助避免数据库上的锁被长时间地打开（请参阅第 29 项），我们可能先要向源账户发送一个查询来确保我们的用户选择过户的钱数没有比账户实际存在的钱多，发送另外一个查询以确保目标账户仍然存在（例如，银行可能由于某些资金严重不足问题而将它关闭）。最后，是一个将源账户中的钱取走存入目标账户中的更新操作。

在某些情况下，我们甚至可能将更新语句也分解成多个步骤，尤其是在用户动作依赖于她开始时刻数据库中数据状态的场景中。例如，我们的用户可能基于此刻所示的她的账户列表中哪个账户上的钱更多来决定从哪个账户过户。

让我们考虑另一种情形，可能更容易识别，在这种情况下，简单事务模型或多或少都是失败的。假定我们正在讨论一个旅行社应用。我想从加利福尼亚的萨克拉门托旅行到德国的慕尼黑。[2] 因为没有直接从萨克拉门托到慕尼黑的航班。我必须乘坐两个航班才能到达那里——一个到航空“中心”，另一个到慕尼黑。事实上，完全有可能我会坐上第三个航班——萨克拉门托到美国航空中心，再到欧洲航空中心，最后到慕尼黑。不过，作为一个经验丰富的旅行者，对于怎样旅行我有我的偏爱。明确地说，我想最少化“航程段”的数量，并且我不介意在航程段之间过夜，只要我不需要在旅行中乘坐火车或打的等就行。

旅行代理人使用为了能够让他有权访问主要的航空公司的旅行系统而构建的事务系统，打算

[2] 这个例子来自于《事务处理（Transaction Processing）》[Gray/Reuter, 4.2.4 部分]

取出所有这些信息并且将其插入到它们的数据库中，来尝试着规划一条合理的旅行路线。他将我的启程地点插入到他的系统中，该系统向他展示可能的目的地列表。他或多或少能猜到下一步将到哪里，因此，让我们假设他开始查找从萨克拉门托到芝加哥的奥海尔机场的航班，因为奥海尔是一个主要的国际性机场，那么它可能比罗切斯特有更好的下一段航程的选择。他登记下那个航班，然后查找从这里到伦敦希斯罗机场的航班。遗憾的是，此时没有从希斯罗到慕尼黑的直接航班，但是他可以让我到法兰克福，然后打的到慕尼黑，这不是我想要的，因为我憎恨出租车。现在我们该重头开始。

那么，还是一样，我们要考虑的有多少个事务呢？

请记住，可扩展系统的目标就是最少化锁的数量（请参阅第 29 项），并且航空公司肯定有兴趣创建可能最具可扩展性的系统。因此让我们考虑旅行代理人使用他的系统来尝试登记这次飞行之旅时，可能使用的两种方式：

1. 他可能使用分布式两阶段提交的事务，通过将第一个航空公司的数据库作为事务的一部分征募进来，发起这个事务。他登记从萨克拉门托到芝加哥的航班。然后转移到另一个航空公司的数据库（也将该数据库作为事务的一部分征募进去），并登记第二个航班。不过，现在是登记第三个航程的时候了，他会发现这整个旅程不会被我接受。因此，他只要向整个分布式事务发布ROLLBACK命令，那么每件事情就都会返回到它先前的状态。
2. 他可以作为单个事务处理这些操作：一个事务处理第一个航空公司的数据库，第二段航程的处理也有它自己的事务，并且当第三段航程的处理开始时，可以直接放弃第三段航程（甚至从不试图去处理它）。

虽然许多程序员都会选择第一种模型，因为在很多方面它是两个中“最干净的”，但是第一个方法还有一些固有的问题。

回滚分布式事务本质上就是强迫旅行社重新开始。即使从萨克拉门托到芝加哥的航程完全可以接受并且还可以探索到更多的选择，但是回滚整个事情强迫旅行社重新登记这些航班。这当然是在假定这些航班仍可订到的条件下——一些其他寻找从萨克拉门托到芝加哥的航班

的旅游者完全有可能在回滚完成和第二个事务开始之间的时机，预定了最后剩余的座位，这使得我们的旅行代理人可获得的工作资源比他想象的要少。

不过，更重要的是，航空公司不允许旅行代理人对开放的座位持有那么长时间的锁，完全是由于同一原因：一些其他的旅行者可能也对你仍在“考虑”的座位感兴趣，但是因为支持“可能的”售卖，所以“确定的”售卖被拒绝了。这并不是保持高收入的方法。相反，航空公司通常是按照乐观的并行模型（请参阅第 33 项）运行，这种模型只有在我真正购买时，才查找检测这个座位是否还仍然存在——是的，我正关注的座位很可能在我的眼皮下突然消失了，但是他们相信：如果允许人们不需要付款就可以对一些座位“上锁”，那么上述情况发生的可能性远比没有销售掉的座位数量要少得多。

因此，综合考虑整个系统，第二种方法更好，尽管它会给编程人员添加额外的工作量。机敏的读者已经发现这个分析的一个主要缺点了：如果我们需要回滚先前已经完成了的事务中的某一个，将会发生什么？如果我们已经将从萨克拉门托到芝加哥的航班提交了，并且我们想要收回我们的要求，那么我们不可能仅仅只是发出ROLLBACK命令就可以撤销已经提交的工作——这将违反ACID事务的持久性部分。

这也就是事务性社区所讨论的补偿性事务，一种知道怎样撤销前期已提交事务的作用效果的事务。那么，在我们的旅行计划的例子中，代码应该知道怎样为萨克拉门托–芝加哥和芝加哥–法兰克福航程撤销已提交的事务，以便允许旅行代理人撤销前面已经完成的工作。（假设航空公司允许旅行代理人在交易中收回已提交事务，以使得数据库上的锁保持开放的时间更短，并且实际上，对于具有高可扩展性需求的系统来说，这通常都是允许的。）不用说，任何其它期望订购从萨克拉门托到芝加哥的航班的旅行代理人都无法意识到我最初持有的座位现在是闲置着的，直到补偿性事务开始运转——换句话说，我们也违背事务的原子性方面。

更糟的是，你可能已经开始考虑编写补偿性事务的实际现实。这也是你的脸色开始变得惨白的时刻：考虑一下，一个基于补偿性事务的系统所要求的全部工作。“但是如果只使用分布式事务，它就要简单的多！”，这句话可能会成为一个战斗口号，而且对于我，也是一名程序员，并不会表示不赞同。但是，有很多事情，对于程序员来说比较简单，对用户来说却并不总是最佳。在这种情况下，折中的办法也相当直接：用编程模型的简单性来对付连接多个数

据源的系统的更大的可扩展性。

在你将这本书抗议地归还书店之前，让我再给你一些告诫，希望能缓和你的恐惧感，并在手中保存好这本书。

首先最重要的是，当处理多个数据库时，这种场景并不是唯一真实的应用。是的，尽管我们总是想控制单个数据库上的事务的锁粒度，但是其实还有其它的方式能够实现，包括信任底层的数据库管道以尽可能合理地保持锁的细粒度。数据库供应商花费了开发生命周期中的大量时间去寻找最大化由他们的系统所引导的事务吞吐量的方式，这些时间比你我可能（甚或是能够）花费的时间要多得多，因此在某种程度上，对他们的努力进行事后的批评是没有意义的。不过，倒是有几种我们可以提供帮助的方式，并且本章剩余的大部分都是讨论它们的。

其次，这种补偿性事务模型假定在两个数据库上执行的动作必须保持本质上的原子性，但是它会欣然地牺牲掉隔离性以获取可扩展性。这也是我们在决定降低事务隔离性的时候所作的同样选择（请参阅第 35 项），不同之处只是这里是在多数据库级别上实现的。如果无法接受这个被降低了的隔离性，我们就不能使用前面的补偿性事务模型。对于很多事情，这种决定必须和用户（和/或他们的业务分析代表）及编程人员一起制定。

最后，然而我们已经有了一种创建好了的模式，从编程观点来看，通过它我们可以大大地简化我们的工作：即命令模式（Command Pattern）[GOF, 233]，把通过一个通用接口而被执行的对象的代码包装起来，封装成“命令”。在Java中，这个接口通常是Runnable接口（它有一个好处就是：可以很容易地将一个Command对象插入到一个单独执行的Thread线程中。）。不过，命令的一个很少被察觉的好处就是能够将命令的“undo”操作与“do”操作紧密地捆绑在一起：

```
public interface DatabaseCommand extends Runnable
{
  // We inherit public void run(); from Runnable
  //
  public void undo();
}
public class DatabaseWorker
```

```
{
  ArrayList commandList = new ArrayList();
  public void execute(DatabaseCommand dc)
  {
    commandList.add(dc);
    dc.run();
  }
  public void rollback()
  {
    ListIterator li =
      commandList.listIterator(commandList.size() + 1);
    while (li.hasPrevious())
      li.undo();
  }
}
```

现在，创建一个DatabaseCommand对象的工作很是微不足道，其中，DatabaseCommand对象不仅实现了提交工作而且也实现了回滚工作，并将它们一个一个地传送到DatabaseWorker实例中，该实例实际上现在就代表我们的用户事务。

在这里我们最后到达了问题的关键所在：正是由于前面提到的一些原因，我们需要将用户的事务角度，完全独立于系统的事务角度。在很多情况下，尤其是当使用单一数据库时，二者就会相当容易地、相当自然地融为一体，当这种情形发生时，可以完全放心地直接求助于标准的事务机制而不需要有任何犹豫。只是要意识到事务锁，并且当需要获得更好的可扩展性时，要愿意放弃事务模型所带来的安逸。

第 29 项：最小化锁窗口

Effective Java[Bloch]的读者肯定会记得来自于第 49 项的建议：“作为一条规则，在被同步的区域内，你应该执行尽可能少的操作。”对于企业级系统，这个规则也有效，在企业级系统中，共享资源中的锁将引发竞争，从而限制了系统最终的可扩展性。

首先，即使是在没有任何竞争的情况下，获得一个对象监视器，并在以后释放它的动作也并不是很简单的事。事实上，我们通常说“你在 Java 中能做的开销最大的事情就是创建一个

对象”（这导致很多种类的缓冲池机制现在大多都显得没有必要了——请参阅第 72 项），但是有了最近 JVM 所给出的有关创建对象的性能提高之后，现在应该说“在 Java 中你能做的开销最大的事情是获得一个对象的监视器。”

考虑这个繁琐的例子，其中一个线程调用一个非同步方法三千五百万次，在这之后调用另一个等同的同步方法三千五百万次，然后比较作为结果的计时时间。我们打算为这个有点琐碎的代码写出三个不同的版本——一个完全不使用任何同步，一个在方法调用本身上使用一个被同步的修改器，另一个使用粒度更粗的被同步的方法。执行分成两部分——第一部分将从一个单一线程中执行一个方法三千五百万次，然后我们新增加一个没有任何其它影响的由命令行传入的线程数，并让这些线程每一个都执行三千五百万次。首先，是非同步版本：

```
// Driver.java
//
// UnsyncDriver: just let 'em rip
//
public class UnsyncDriver
{
  private static int callCount = 0;
  public static void call()
  {
    // Do something to avoid dead code removal optimizations
    //
    callCount++;
  }
  static class CallRunner implements Runnable
  {
    public void run()
    {
      long start = System.currentTimeMillis();
      for (int i=0; i<35000000; i++)
        call();
      long end = System.currentTimeMillis();
      System.out.println(Thread.currentThread().getName() +
        " Call() Start: " + start + "ms " +
        "End: " + end + "ms Diff = " +
        (end - start) + "ms");
    }
  }
  public static void main(String[] args)
```

```
  {
    // Warm up the JIT
    for (int i=0; i<100000; i++)
    {
      call();
    }
    long start = System.currentTimeMillis();
    for (int i=0; i<35000000; i++)
      call();
    long end = System.currentTimeMillis();
    System.out.println("Call() Start: " + start + "ms " +
                       "End: " + end + " ms " +
                       "Diff = " + (end - start) + " ms");
    if (args.length > 0)
    {
      int threads = Integer.parseInt(args[0]);
      System.out.println("----------- Executing " + threads +
                         " threads. -----------");
      Thread[] threadArray = new Thread[threads];
      for (int i=0; i<threads; i++)
        threadArray[i] =
          new Thread(new CallRunner(), "Thread " + i);
      for (int i=0; i<threads; i++)
        threadArray[i].start();
    }
  }
}
```

除了将call方法装饰上synchronized关键字之外，同步版本几乎与上面的版本一样，并且粗粒度版本还使用了一个私有的Object锁对方法体本身上锁：

```
// FineSyncDriver.java
//
// Just like UnsyncDriver, except for the following
public class FineSyncDriver
{
  public static synchronized void syncCall()
  {
    // Do something to avoid dead code removal optimizations
    //
    syncCallCount++;
  }
```

```
}
// CoarseSyncDriver.java
//
// Just like UnsyncDriver, except for the following
public class CoarseSyncDriver
{
 private static int callCount = 0;
 private static Object lock = new Object();
 public static void call()
 {
  // Do something so as to avoid dead code removal
  // optimizations
  //
  syncronized(lock)
  {
   callCount++;
  }
 }
}
```

用 10 个线程运行这三个版本产生了一些有趣的内部现象，如下面输出所示：

```
C:\Projects\EEJ\code> java UnsyncDriver 10
Call() Start: 1076330441441ms End: 1076330441651 ms
                              Diff = 210 ms
public class CoarseSyncDriver
{
----------- Executing 10 threads. -----------
Thread 1 Call() Start: 1076330441721ms
              End: 1076330442402ms Diff = 681ms
Thread 3 Call() Start: 1076330441841ms
              End: 1076330442512ms Diff = 671ms
Thread 0 Call() Start: 1076330441661ms
              End: 1076330442572ms Diff = 911ms
Thread 2 Call() Start: 1076330441781ms
              End: 1076330442572ms Diff = 791ms
Thread 4 Call() Start: 1076330442142ms
              End: 1076330443043ms Diff = 901ms
Thread 5 Call() Start: 1076330442202ms
              End: 1076330443343ms Diff = 1141ms
Thread 7 Call() Start: 1076330442803ms
              End: 1076330443403ms Diff = 600ms
Thread 8 Call() Start: 1076330442753ms
```

```
             End: 1076330443403ms Diff = 650ms
Thread 9 Call() Start: 1076330442702ms
             End: 1076330443413ms Diff = 711ms
Thread 6 Call() Start: 1076330442262ms
             End: 1076330443413ms Diff = 1151ms
C:\Projects\EEJ\code> java FineSyncDriver 10
FineCall() Start: 1076330565970ms
         End: 1076330566731 ms Diff = 761 ms
------------ Executing 10 threads. ------------
Thread 0 FineSyncCall() Start: 1076330566731ms
                   End: 1076331238256ms Diff = 671525ms
Thread 8 FineSyncCall() Start: 1076330566791ms
                   End: 1076331310000ms Diff = 743209ms
Thread 4 FineSyncCall() Start: 1076330566791ms
                  End: 1076331310010ms Diff = 743219ms
Thread 9 FineSyncCall() Start: 1076330566791ms
                   End: 1076331310010ms Diff = 743219ms
Thread 2 FineSyncCall() Start: 1076330566791ms
                   End: 1076331310010ms Diff = 743219ms
Thread 7 FineSyncCall() Start: 1076330566791ms
                   End: 1076331310010ms Diff = 743219ms
Thread 1 FineSyncCall() Start: 1076330566791ms
                   End: 1076331310010ms Diff = 743219ms
Thread 3 FineSyncCall() Start: 1076330566791ms
                   End: 1076331310010ms Diff = 743219ms
Thread 6 FineSyncCall() Start: 1076330566791ms
                   End: 1076331310010ms Diff = 743219ms
Thread 5 FineSyncCall() Start: 1076330566791ms
                   End: 1076331310010ms Diff = 743219ms
C:\Projects\EEJ\code> java CoarseSyncDriver 10
CoarseCall() Start: 1076331255291ms
          End: 1076331255361 ms Diff = 70 ms
------------ Executing 10 threads. ------------
Thread 0 Call() Start: 1076331255371ms
             End: 1076331255641ms Diff = 270ms
Thread 1 Call() Start: 1076331255431ms
             End: 1076331255942ms Diff = 511ms
Thread 2 Call() Start: 1076331255431ms
             End: 1076331256252ms Diff = 821ms
Thread 3 Call() Start: 1076331255431ms
             End: 1076331256553ms Diff = 1122ms
Thread 4 Call() Start: 1076331255431ms
             End: 1076331256863ms Diff = 1432ms
Thread 5 Call() Start: 1076331255431ms
```

```
                End: 1076331257164ms Diff = 1733ms
Thread 6 Call() Start: 1076331255431ms
                End: 1076331257474ms Diff = 2043ms
Thread 7 Call() Start: 1076331255431ms
                End: 1076331257774ms Diff = 2343ms
Thread 8 Call() Start: 1076331255431ms
                End: 1076331258075ms Diff = 2644ms
Thread 9 Call() Start: 1076331255491ms
                End: 1076331258265ms Diff = 2774ms
C:\Projects\EEJ\code>
```

首先，我们可以很清楚地看到非同步版本是最快的，无论在何处，执行时间都从 600 到 1200 毫秒不等；差异可能是由于线程调度器以及哪个线程碰巧被首先发起而造成的（因此这就是在不得不让出 CPU 之前，所获得的最长运行时间。）。顺便说一句，小心注意：执行非同步方法的 10 个线程可能很容易把自己的值完全搞错，此时我完全都没有予以考虑。而你并没有那么奢侈，因此不要期望仅仅通过删除同步锁，你就可以提高系统的性能和扩展性。你的代码可以会运行得更快，只是深夜调试产品服务器就不值得了。

第二，我们还可以很清楚地看到粗粒度的同步版本比细粒度的同步版本花费时间要少很多，不过对于细粒度同步版本，有一件奇怪的事情：执行花费的时间好像与线程结束的时间成线性比例。实际上，这正是很精确地对应于所发生的事情——第一个调度线程抢占锁，并在整个三千五百次迭代循环中持有该锁，期间从没有放弃过该锁，因此其它的线程必须等待直到释放锁，才能去为得到它而竞争。到最后一个线程获得锁的时候，这个线程已经等待了一段时间。

第三，在细粒度的机制中，在每次调用上都会获得和释放监视器，因此产生了最平均的数字，不过也最差：大概是 740 秒，或者说每个线程执行完毕需要整整 12 分钟。并且这只是 10 个线程；想象一下如果需要 20、50 或 100 个线程，又会怎么样呢？

不应该从一次单独运行中做出太多的结论，但是很明显，同步有它自身的开销，很早以前我们就考虑过在被同步区域内所花费的时间。

简而言之，竞争是可扩展性的敌人。系统中存在越多的竞争，我们就需要等待越长的时间，

反过来又会加重系统的整体的响应延迟。我们不能扩展一个存在竞争问题的系统，因为请记住，可扩展性的要旨就是能够在出现问题时，只需增加更多的硬件，就可以向系统添加用户——如果响应延迟是由于人人都必须等待共享某个单一锁而引起的，那么世界上所有的硬件放到一起都无法给我们提供帮助。

最后，紧记两点。首先，当你有别的办法时，就请避免使用锁——即使没有任何其他人最终和你竞争锁，但它仍要把花费宝贵的时间花在只是获得监视器和将它释放上。这意味着不要通过将servlet中全部的doGet或 doPost方法同步化来“获得安全”。相反，应该花费时间来分析代码以判断哪些地方可能是热点，并且仅仅将这些热点同步化。

第二，尽可能地减少花费在同步区域内的时间；我会留一个练习给不相信的读者，将Thread.sleep调用放到call方法中来模拟对锁占有一段时间，变换线程数量再进行实验。足以证明，调用窗口时间越长，并且线程数量越多，所花费时间的情况就会（可能指数地）变得越糟。（如果你打算运行测试一下，那么你自己最好：将迭代的次数从三千五百万次降到一个更合理的数字，否则你将花费很长的时间去等待测试结束。）

当然，还有其它一些弥补措施。一种方法就是优选不需要同步的不变资源（对象、数据或任何其它事物），因为它们不会改变，正如第 38 项所述。另外一种就是将资源处理卸载到队列中（通过第 20 项），在此队列中可以按照异步的序列化方式进行处理，很像批处理那样。关键也是同样的：尽可能短的上锁时间。

第 30 项 ： 当持有锁时不要让步给在构件之外的控制

Effective Java [Bloch]的读者无庸置疑地会记得这一项的标题来自第 49 项的建议，几乎是以完全相同的方式表达的。幸运的是，多亏了EJB规范以及一些已经构建了事务死锁避免代码的大多数关系型数据库都已经构建了内置的安全措施，所以死锁就不是J2EE系统要特别关注的东西了。尽管如此，由于它部分地涉及到了前面的要保持锁窗口较小（请参阅第 29 项），所以，在持有的锁处于开放期间，调用到“外面的”构件仍旧是一个坏想法。

作为一个快速提示，请观察下面将会确切地发生什么？

```
public class Example
{
  private Object lock = new Object();
  public void doSomething()
  {
    synchronized(lock)
    {
      // Do something interesting
    }
  }
}
```

精确一点，执行doSomething方法的线程试图获得被lock所引用的对象实例的对象监视器的所有权。假定没有其它线程拥有这个监视器，那么这个线程很快就可以得到了该监视器。

更进一步：当一个线程调用递归到同步区域时，又会发生什么呢？

```
public class Example
{
  private Object lock = new Object();
  public void methodA()
  {
    synchronized(lock)
    {
      methodB();
    }
  }
  public void methodB()
  {
    synchronized(lock)
    {
      // Do something interesting
    }
  }
}
```

JVM记得调用methodA的线程拥有对lock对象的监视器，因此它在methodB中会掠过被同步

的区域；在Java中一个线程是不可能被其自身死锁的。

这是 Java 101 的修订版；为什么要在这里引入呢？从不同方面的考虑一下，在它后面添加一些 RMI 语义。尤其要记住，RMI 规范描述到，对一个导出对象的远程方法调用（也就是说，经由 RMI 栈来自于 JVM 外面的方法调用）可以来自于任意线程。因为 JVM 只记得拥有一个监视器的本地线程，所以在 JVM 之外进行调用，然后再返回到 JVM 的线程内，就有可能使一个线程将自身死锁。

```
// Deadlock.java
//
import java.rmi.*;
import java.rmi.server.*;
public class Deadlock
{
  public static interface RemoteFoo extends Remote {
    public void doTheCall() throws RemoteException;
  }
  public static class RemoteFooImpl
    extends UnicastRemoteObject
    implements RemoteFoo
  {
    public RemoteFooImpl() throws RemoteException { super(); }
    private Object lock = new Object();
    public void doTheCall() throws RemoteException
    {
      try {
        System.out.println("Entered doTheCall()");
        synchronized(lock) {
          System.out.println("Entered synchronized region");
          RemoteFoo rf = (RemoteFoo)
            Naming.lookup("rmi://localhost/RemoteFoo");
          rf.doTheCall();
        }
      }
      catch (NotBoundException nbex) {
        System.out.println("Not bound?"); }
      catch (java.net.MalformedURLException malEx) {
        System.out.println("Malformed URL"); }
    }
  }
```

```
  public static void main(String[] args) throws Exception
  {
    RemoteFooImpl rfi = new RemoteFooImpl();
    Naming.bind("RemoteFoo", rfi);
    RemoteFoo rf =
      (RemoteFoo)Naming.lookup("rmi://localhost/RemoteFoo");
    rf.doTheCall();
  }
}
```

这个程序的输出会让任何分布式系统程序员凉透脊背：

```
C:\Projects\EEJ\code\RMIdeadlock>java Deadlock
Entered doTheCall()
Entered synchronized region
Entered doTheCall()
     Window hangs
```

毫无疑问，如果我们在 Windows JVM 中使用 Ctrl-Break 技巧，就会得到一个完整的线程堆，我们可以看到这个死锁：

```
Full thread dump Java HotSpot(TM) Client VM (. . .)
other thread dumps removed for clarity
"RMI TCP Connection(3)-192.168.1.102" daemon prio=5
       tid=0x02edaf30 nid=0x7d8
       waiting for monitor entry [32cf000..32cfd8c]
       at Deadlock$RemoteFooImpl.doTheCall(Deadlock.java:29)
       - waiting to lock <0x1050d3d8> (a java.lang.Object)
       at sun.reflect.NativeMethodAccessorImpl.invoke0(
              Native Method)
       at sun.reflect.NativeMethodAccessorImpl.invoke(
              NativeMethodAccessorImpl.java:39)
       at sun.reflect.DelegatingMethodAccessorImpl.invoke(
              DelegatingMethodAccessorImpl.java:25)
       at java.lang.reflect.Method.invoke(Method.java:324)
       at sun.rmi.server.UnicastServerRef.dispatch(
              UnicastServerRef.java:261)
       at sun.rmi.transport.Transport$1.run(
              Transport.java:148)
       at java.security.AccessController.doPrivileged(
              Native Method)
```

```
        at sun.rmi.transport.Transport.serviceCall(
               Transport.java:144)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(
               TCPTransport.java:460)
        at
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(
               TCPTransport.java:701)
        at java.lang.Thread.run(Thread.java:534)
"RMI TCP Connection(2)-192.168.1.102" daemon prio=5
        tid=0x02ed9238 nid=0x91c
        runnable [324f000..324fd8c]
        at java.net.SocketInputStream.socketRead0(Native
          Method)
        at java.net.SocketInputStream.read(
               SocketInputStream.java:129)
        at java.io.BufferedInputStream.fill(
               BufferedInputStream.java:183)
        at java.io.BufferedInputStream.read(
               BufferedInputStream.java:201)
        - locked <0x1006a110> (a java.io.BufferedInputStream)
        at java.io.DataInputStream.readByte(
               DataInputStream.java:331)
        at sun.rmi.transport.StreamRemoteCall.executeCall(
               StreamRemoteCall.java:189)
        at sun.rmi.server.UnicastRef.invoke(
               UnicastRef.java:133)
        at Deadlock$RemoteFooImpl_Stub.doTheCall(
               Unknown Source)
        at Deadlock$RemoteFooImpl.doTheCall(Deadlock.java:31)
        - locked <0x1050d3d8> (a java.lang.Object)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(
               Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(
               NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(
               DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at sun.rmi.server.UnicastServerRef.dispatch(
               UnicastServerRef.java:261)
        at sun.rmi.transport.Transport$1.run(
               Transport.java:148)
        at java.security.AccessController.doPrivileged(
               Native Method)
        at sun.rmi.transport.Transport.serviceCall(
```

```
        Transport.java:144)
at sun.rmi.transport.tcp.TCPTransport.handleMessages(
        TCPTransport.java:460)
at sun.rmi.transport.tcp.TCPTransport$Connection
        Handler.run(TCPTransport.java:701)
at java.lang.Thread.run(Thread.java:534)
```

如你所见两个线程——一个调用者，另一个线程通过 RMI 被分派来响应到来的请求——被死锁了，因为调用者在等待被调者返回而阻塞，而被调者为了要在调用者上过锁的对象上上锁而等待。

这是一桩烦人的事；意味着RMI规范没有任何能力来发现一个符合逻辑的调用序列，使得JVM的执行仍能绕回到作为该处理的一部分的同一个JVM中，这正是事务性COM＋[Ewald]所指的因果律（causality），以及EJB规范所指的回转通道（*loopback*）。没有这项功能，就没有办法去探测逻辑上由自我创建的死锁，更不用说由两个客户创建的死锁。如果存在任何产生这样的循环调用的可能性，那么就需要由你来处理了。

注意EJB并不能严密地帮助我们去解决这里的一大堆问题——万一当前这种回调发生在一个有状态会话Bean上，容器就会抛出异常（如果是远程调用，则抛出RemoteException，如果是本地调用，则抛出EJBException），因此，就可以有效地回滚整个工作链。这整个行为是仔细设计过的，正如在规范(7.12.10 部分)中清楚陈述的那样：“这条规则隐含的意思是，一个应用不能回转调用一个会话Bean实例。”，这样做避免了死锁，但也会使所做的操作作废，并且只能在运行时导入。

如果这是唯一要关注的，那么我们可以不予考虑——毕竟，这种情形发生的最大可能性是当设计者试图通过使用Observer模式[GOF, 217]，在有状态会话Bean上构建某种通告机制的时候。这就像Java事件处理通常所作的那样，既然我们是使用这种方式来识别问题的，那么我们就应该求助于基于消息的方法来避免该问题。由于这个问题相对很少发生，所以就不值得在这里讨论了。

事实并不是这样，当你持有锁时，如果在构件之外进行调用，那么就会产生更大的问题，你正在将“外部”代码希望能够执行的操作有效地囊括为你的锁时间窗口的一部分（请参阅第

29 项）。这意味着，“外部”代码，即被定义为不在你控制之下的代码，完全有可能去执行诸如像计算 pi 到小数点后第 100 位这样令人头疼的事情，这使得远程服务调用或者更糟的行为都可能在你持有锁的时候发生。

这里隐含的东西远比你最初想到的要多得多。对于事务的启动者，当选择 EJB 上的事务关系时，你需要极度小心，尤其是在容器管理的事务（CMT, Container-managed transaction）下被执行的 EJB 操作，因为 CMT 事务窗口是方法调用的整个方法体，因此，在开放事务并有可能去掉锁之前，你没有任何机会进行非事务处理。只有当 EJB 在本质上是完全原子性的时候（也就是说，它不产生除对数据的调用之外的任何对其它构件的调用，希望到数据库的调用的处理是很容易理解的），CMT 才是一种可行的选择。

你还需要意识到这个事实，经由MessageListener被异步调用的JMS消费者（包括消息驱动 Bean）可以被任意线程调用，这意味着我们又回到另一个线程也需要我们所持有的对象监视器的危险中。尤其是在使用消息传递来打破请求-响应循环的时候（请参阅第 20 项），要意识到消息的传送可能立即就会将异步处理机制踢到一边去了，并且在你传送消息时所持有的任何锁，都有可能会产生竞争，或者在更糟糕的场景中，会形成死锁，例如，当消息处理器在试图获得你的线程当前所持有的锁之前，就像去除在你锁住的某些事物上的锁。并且因为它是如此地依赖于时机的选择，所以只有在负载严重的情况下或者在为潜在的一掷千金的客户做产品展示期间，才会使用它。

为了你自己，在执行任何构件外的通信之前，请确保所有的锁是释放的；这样做了，你就会成为更加愉快和无忧无虑的程序员。

第 31 项：理解 EJB 的事务关联

它是 EJB 规范中那些很多 EJB 程序员要么从不给予考虑，要么即使他们考虑了，也只是一直将它搁置到一边直到后来用到时才会想到的众多方面之一。本质上，它确实和编程没有任何关系，因为它从来没有以代码的形式示人。相反，大多数程序员只是依赖于他们使用的工具或者 IDE 来获取任何“感觉是正确的”值，然后对它不作任何修改。遗憾的是，在你的

EJB Bean 上选择事务关联，和其它任何事物一样对 Bean 的最终性能和可扩展性有着重要的作用，或许还会显得更多。

考虑一下经典的会话外观[Alur/Crupi/Malks, 341]——作为网络往返调用的优化，经由一个会话Bean把实体Bean的所有访问封装起来，因此可以强制客户批量地访问实体Bean而不是通过单个方法调用来访问。例如，有个实体Bean有五个属性，不是通过在客户端单独地设置实体Bean的每个属性，而是经由会话Bean的单个方法调用传送全部 5 个属性。会话Bean接着对实体Bean调用set方法，因此，去掉了到实体Bean往返调用的一段历程。相当直接，是吧？

那么我们就进入事务关联吧。

此时，我假设你或多或少对EJB中 6 种不同的容器管理事务关联设置有所了解：Mandatory、Requires、RequiresNew、Supports、NotSupported和Never。然而，你不熟悉的可能是这些关联设置之间的交互——对于一个从一个Bean到另一个Bean的调用，容器是如何反应的。例如，一个正在事务下运行的Bean调用另外一个标识为Requires的Bean的时候，来自于第一个Bean的事务就被传播到第二个Bean中，因此这两个调用是在一个单独的事务下进行操作的。表 4-1 详细展示了交互的完备集，包括容器管理的事务和Bean管理的事务。

表 4-1. 事务关联设置的效果

	在事务范围内？	事务范围的根部？	共享调用者的范围？
CMT: NotSupported	从不	从不是	从不共享
CMT: Never	从不	从不是	从不共享
CMT: Supports	如果调用者支持	从不是	如果调用者有一个事务，则共享
CMT: Requires	总是	只有调用者没有任何事务时才是	如果调用者有一个事务，则共享
CMT: RequiresNew	总是	总是	从不共享

CMT: Mandatory	总是	从不是	总是共享
BMT: 用户启动事务	总是	总是	从不共享
BMT: 没有启动任何事务	从不	从不是	从不共享

认识这个表的关键是：Bean 之间的事务关联交互会定义 Bean 将会如何执行的大部分内容。

假定实体Bean的set方法，不管由于何种原因，全部被标识成了RequiresNew类型的事务关联。这意味着每个set调用必须在它自己的事务下运行，这也意味着Bean的整个状态必须在事务开始的时候载入，并且在调用结束的时候存储到数据库中。这样，它不仅要求每个调用都要对数据库进行额外的往返调用，而且也不再适应完全的两阶段分布式事务语义（除非你的容器支持使用本地事务，这种情况请看第 32 项）。

实际中，实体Bean很少将属性方法上的事务关联设置为RequiresNew。更多情况是将这些方法设置为Requires，表明它会借用调用者的事务，使得实体Bean能够更灵活地适应会话外观的使用方法。但是如果会话外观 [Alur/Crupi/Malks, 341] 内的会话Bean被设置成了Supports、NotSupported或Never，我们就又回到了前面所述的情形中——对实体Bean方法的每次调用都会产生一个到数据库的新事务，意味着需要多个往返。

考虑一下常见的 EJB 容器，它们没有运行任何特殊的对数据库的独占式的访问（这些访问无论如何也不可能在容器之间进行移植，请参阅第 11 项以了解更多的细节）。实体 Bean 在容器内持有同时也是数据库持有的数据。因为这两个进程（容器进程和数据库进程）很少在同一台机器上进行，更不会在同一个进程空间内运行，因此完全有可能，EJB 容器持有的在内存中的数据已经被数据库改动过。

现在想象一下，正被排序的实体Bean要将一个属性值改为某一新值（不管是来自会话外观 [Alur/Crupi/Malks, 341]，还是直接来自客户端，都没有什么关系）。任何对实体Bean的访问都要在容器管理事务语义下完成（EJB 2.1 规范的 17.6.1），因此一个事务被启动了。然而，因为数据库中的数据有可能被EJB容器所能感知的范围之外的某个进程给修改掉，那么EJB容器在继续处理调用之前，必须重新载入整个Bean。接着调用set，提交事务，并刷新数据

库中的数据。

这表明对实体Bean的任何访问（再次提醒，没有使用供应商的排他的增值功能），都意味着EJB容器至少要进行从EJB容器到数据库的两个往返：一个是取出数据，另一个是刷新修改过的数据。幸运的是，只有在事务的边界处才需要这样做，因此通过被正确标识的能够感知事务的会话外观 [Alur/Crupi/Malks, 341]，正在被修改的实体Bean除了在事务开始和事务提交的时刻，其它时刻都不需要这样做。

然而，要指出的是，这对额外的往返是唯一必需的，因为在两个不同的地方同时持有该数据——在关系型数据库和 EJB 容器中。如果实体 Bean 完全被对关系型数据库的直接 SQL 访问而旁路（细节请参阅第 41 项），那么，数据就只会宿主在关系型数据库中，并且不需要任何额外的获取-刷新循环。

认识到对一个给定方法所作的事务关联选择能够造成的影响是不够的。另外，为了最大化对事务的使用（避免产生的事务比所需的要多）和最小化它的生命周期（请参阅第 29 项），在 EJB 容器内被调用的方法上的事务关联的联合也必须予以考虑，因为你正在有效地进行构件外调用，正如第 30 项所述那样。那么，这暗示着对于任何给定的在你的 EJB 容器内定义的 Bean，称之为 Bean X，你必须不仅考虑 X 上每个方法的事务关联，而且还要考虑调用 X 的任何 Bean 上的事务关联，以及被 X 调用的任何 Bean 上的事务关联。如果疏忽了这一点，就会产生比绝对必需要更长的事务。

第 32 项：优先使用本地事务而不是分布式事务

在给定资源上创建事务通常是一种与技术相关的动作——例如，在关系型数据库中，通过使用SQL语句BEGIN TRANSACTION创建事务。在这之后，所有的工作是在某种临时空间内完成的，在这个空间内，执行指令（SQL）的结果不会被感觉到，直到事务本身要么完成（使用SQL COMMIT语义）要么放弃（使用SQL ROLLBACK）为止。

不过，仍然会有问题——最特别的是，如果我们有两个需要作为某个单一处理的一部分而被

操作的资源，那么会发生什么呢？

规范场景是操作对两个不同的数据库[3]，不过在J2EE下，这还可以是一个数据库和一个JMS提供者，或者是一个Connector提供者。通常，当只有一项资源被使用时，我们可以依赖提供者来处理保证事务不被侵犯的问题。然而，如果我们要跨提供商使用资源，那么事情就变得更复杂了。

为了直接处理这些问题（在 J2EE 出现之前就已经有很久的历史了），大量的数据库开发商聚集到一起定义了分布式事务协议，称作两阶段提交协议（TPC, two-phase commit protocol）。在 TPC 中，我们正式地定义了作为每个事务的一部分的三个参与方：客户端；资源管理器（RM, Resource Manager），它提供了我们试图共享访问的共享资源；和事务管理器（TM, Transaction Manager），它被用来创建分布式事务并处理客户和 RM 之间的交互。

TPC 协议，如果剥离出它的本质，那么它看上去就像下面列出的那样：

1. 客户端从 TM 获得分布式事务。
2. 客户端征募所需的、作为事务一部分的 RM。这是通过将事务呈现给 RM 来实现的，因此 RM 能感知到分布式事务并且知道要去期待协议的剩余部分。
3. 客户端通常处理获取的 RM。RM 知道这是分布式事务的一部分，所以它会确保在 TM 接收到之前，不会提交或回滚所做的工作。（保持数据是 RM 的职责，其中的原因在稍后就会弄清楚。）
4. 当客户端已经做好结束的准备时，客户端向 TM 发出信号进行提交。TM 此时接收这些信息。
5. 阶段 1：TM 首先向事务上的每个 RM 发出信号，大体上是询问“你准备好提交了吗？”这是 RM 从事务中退出的唯一机会，可以是由于任何原因——缺少磁盘空间、关系完整性约束等等。因此，RM 通常将数据写成“将要提交的”状态；它知道要对系统中的其它事务隐藏该数据。如果事务上的任意一个 RM 表明存在某种故障，那么整个事务必须回滚，并且 TM 立即命令事务上所有的 RM 都要放弃，即使它们先前表明愿意提交。
6. 阶段 2：如果事务上的所有方（RM）都发信号表示愿意提交，那么 TM 再次向四处发送

[3] 通常来自于不同的供应商——对于同一个供应商的两个不同的数据库实例（例如，Oracle和Oracle），供应商提供挂钩使每件事情都保持井井有条。当我们在操作Oracle 和DB/2, 或者DB/2 和Sybase的时候，问题就会蔓延开来。

一个信号，主要是告诉它们“继续进行并提交。”这里没有任何表决——RM 必须提交数据，并且没有任何中途退出或失败的借口。如果所有的 RM 都发出了提交成功的信号，那么 TM 就向客户端发出成功提交的信号，事务也就处理完毕了。

如你所见，TPC 协议是整部机器中相对复杂的一个部件。幸运的是，它也是比较可靠的一个部件，因为它用极高的一致性等级，已经成功地为数据库访问提供强大动力几十年了。实际上，对于本地事务，大多数关系型数据库都使用本地化的 TPC 版本来提供相同类型的一致性和可靠性——在这些情形下，TM 和 RM 是同一个进程，因此它变成了一种简单得多的处理。

不过，没有免费的午餐，TPC 也有它自身的开销。尤其是，因为 TPC 需要分布式通信，花费到执行分布式事务上的时间就比本地事务要多出若干个数量级。即使是在只有一个资源被事务征募的场景中，TM 和 RM 之间的交互也需要进程间（如果不是机器间的话）通信，正如第 17 项所叙述的那样，这会是一个惊人的延时。这意味着 RM（提供必要的 ACID 性质）内部持有的锁，对于分布式事务来说可能持有时间会更长，这是一种不受欢迎的特性（就像第 29 项所述一样）。

那么你应该在什么地方规避它呢？由于很明显的原因，除非你绝对必须使用分布式事务，否则你就应该避免使用它，因为分布式事务意味着你必须在多个资源上（数据库、JMS提供者、JCA的Connector提供者等等）具有ACID属性。千万要注意这里是怎样表达的：你需要分布式事务，只有在你具有多个数据源，并且在处理这些多个数据源时，你必须具有事务性的语义的情况下——例如，当且仅当数据库的INSERT操作成功时，必须发送一条消息到一个JMS的Queue。

尽管似乎要经常提出这种需求，但是事实证明它并不像你想象的那么流行。例如，对于很多系统来说，即使其它的数据库可能只是系统收集到的数据的最后接收者，而不是对这些数据库进行直接的存取，开发者也会处理他们自己的数据库，并且让后端进程去获得期望的数据，并将它们运送给那些其它的数据库。或者，你的系统可能需要实时地读取另外一个数据库，但是不会做任何更新，因此其它数据库就不需要任何事务（细节请参阅第 35 项）。

不过，最糟糕的部分接踵而来：在缺省情况下，由你所喜欢的 EJB 容器所取出的每个事务都会被分布为 TPC 事务，即使在任何时候都只用到一个资源管理器也是如此。EJB 规范（17.1.1 节）包含了一个如下所述的信息段（用斜体字区分）：

很多应用都会包含一个或几个企业级 Bean，这些 Bean 都只使用一个单一的资源管理器（通常是关系型数据库管理系统）。EJB 容器可以利用资源管理器的本地事务作为并不是必需使用分布式事务的企业级 Bean 的优化技术……容器为实体 Bean 使用本地事务作为优化技术，这些 Bean 要么是由容器管理事务边界的划分，要么是 Bean 管理事务边界的划分，不过这种划分对企业级 Bean 来说是不可见的。

或者，换句话说，你不能保证容器将会选择使用本地事务，你也不能向容器表明你的企业级Bean应该使用本地事务。这个领域是EJB规范中被明确表示为具有“你作为一个程序员知道的越少就越好”这样的属性。并且如果容器规定，你对数据库和JMS Queue的访问必须在分布式事务下发生，那么现在不管你想还是不想，你都运行在分布式事务下。并且，由于自动征募的壮举——在EJB容器内部显得非常神奇，这些EJB容器能够自动征募像数据库这样的资源，使得这些资源一旦被JNDI Context检索到，就立即成为了事务的一部分——在同一个方法中直接引用一个JMS Queue和一个DataSource将会把它们置于同一个事务中，即使你不想它们被这样处理也毫无办法。（一旦事务开放，那么无论使用的是容器管理的事务还是Bean管理的事务，这一点都将成为事实。因此你避免这种情形的唯一希望就是——如果你想或者需要避免——编写Bean管理的事务，并且不要开放事务，直到你获得不应该在这个EJB事务之中的资源之后。棘手啊。）

最终结果？如果你想使你的事务窗口尽可能的短（请参阅第 29 项），由于 TPC 协议增加的通信需求，你将会在所有可能的地方使用本地事务。这样做可能需要用其它途径来实现你的企业级逻辑，因为 EJB 不允许强制使用本地事务，并且它优先采用的是取而代之的运行分布式事务，而不管你对此事有何意见。结合 EJB 并发模式（请参阅第 31 项），它反过来加强了这个观念：（a）EJB 真的只是被用于事务处理，并且（b）尤其是，EJB 真的是被用于分布式事务处理，正如第 9 项所述。

第 33 项：为了更好的可扩展性而考虑使用乐观的并发机制

考虑一下两个用户试图处理同一数据这个经典问题：每个用户都取出数据库中数据的一个工作备份，然后对它进行更新（因此为每个用户创建瞬时状态，如第五章所介绍的）。现在每个用户都想将改变保存回数据存储中。我们可以使用几种机制——情形之一是，在第一个用户读取数据时，我们可能通过发起一个事务，来依赖底层的数据库并发工具，因而也就取得了在该数据上的一个锁以确保没有其他人能够篡改这个数据，直到我们的第一个用户提交过所做的修改为止。遗憾的是，在持有一个开放事务时，明确地将控制让与用户（请参阅第 30 项），这种方法会产生巨大的锁窗口（违反了第 29 项的建议）。由于可扩展性被损坏，所以你不可能做出比这更糟糕的事了。

接下来的另一个方法可能是让读取成为一个事务，而让更新成为另一个分离的事务。在我们运行整个场景的时候，就能很浅显易懂地明白我们在这里遇到的问题——我们的两个用户执行读取，当更新的时候，两个更新一个接一个地进行，第二个更新很轻松地就将第一个更新给覆盖掉了。然而更糟的是，底层的资源管理器——数据库——在这种情况下，不能提供任何暗示或信号表明数据已经被覆盖掉了。

另一可供选择的方法可能是降低每个事务上的隔离级别（细节请参阅第 35 项），这样就允许进行同时的读取。最终效果可能也是一样的，然而：数据的丢失是因为每个用户进行更新数据时没有意识到他或她可能正在覆盖另一个用户所做的修改。

乐观的并发模型，也称作乐观离线锁（Optimistic Offline Lock）[Fowler, 416]或者乐观锁（Optimistic Lock）[Nock, 395]，提供了一种方法，可以兼顾两方，并双收其利——给你增加了一小部分的额外工作，同时你还可以将实际取出的锁的数量保持到最小。从本质上讲，你正在为从数据存储检索到的每个数据集添加某种“数据修改标记”，在进行更新的时候，检查数据库中瞬时状态的标记，如果它们不同，就采取“恰当的动作”来解决这个差异。“恰当的动作”可能随系统的不同而不同，但是也有一些包含试图自动合并修改的可能性，使得可以向用户展示修改过的数据，并给予她或他进行合并的机会，他或她可以直接覆盖（我们起先试图要避开的问题，有时可能仍然是要做的事情），或者向用户显示消息“嘿，有人修改过你的数据，我下一步应该怎么办？”

实际上乐观并发可以按照不同的形式实现；版本号（Version Number）[Marinescu, 70]就是这样的一种方法，在这种方法中，每个表包含一个增添的版本号列；其它方法喜欢使用最新修改的时间戳。使用版本号会给你一个提示，以告诉你自从你上次查看这个数据开始，它已经被修改了多少次（如果你持有的是版本 1，而当前版本是 20，说明这个数据已经被修改过多次，那么你可能会选择将它丢弃然后重新开始），它不是以简单的时间戳方式来呈现的。另一方面，时间戳方法可以让你在方法执行过程中获得数据库的临时要素，这对于审计来说极为有用，对于审计我们会在后面稍微讨论一下。

无论你选择了什么方法，在乐观并发模式下，访问数据的基本代码都如下所示：

```
SELECT primary_key, data elements of interest, last_modified
  FROM Table
  WHERE primary_key=?
Cache off last_modified; when it comes time to update, do:
UPDATE Table SET (data elements of interest = new values,
                last_modified = system-generated timestamp)
  WHERE primary_key=? AND last_modified=old last_modified value
```

密切注视UPDATE语句的行数至关重要；如果返回 0 行，意味着更新谓词（WHERE语句）失败了。除非你有数据污染问题，否则返回 0 行就意味着主键标识的行没有一个last_modified列和你认为应该有的last_modified数据相匹配——这是有问题的行的last_modified时间戳已经修改过的征兆，它可以告诉我们整个这行可能被修改过。

此时，下面将会发生什么完全由你决定。尤其是，在表中存在的问题是应该怎样处理数据，这些数据现在已经不能够和在数据库中存储的数据保持同步了。

- *放弃*：直接将修改丢弃，用最近被修改过的数据再重新开始。在某些情形下，这是最好的方法，尤其是对于修改幅度很大，或者如果使用者不是真的有能力去决定哪些值应该被合并的情况更是如此。它不要求你必须要告诉用户你做了什么，不过——放弃而不告知用户是相当不友好的做法。
- *覆盖*：有时最好的方法就是让最后的更新胜出。不过如果是这种情形，那么，首先就真

的没有进行乐观加锁的必要了——盲目的UPDATE也能很好地运转。

- *合并*：获得该用户的数据，与原来的数据进行比较，然后合并所做的修改。不过，这是一种充满技巧的方法，因为你要艰难地决定应该怎样进行合并。为了知道你的用户修改了哪些列，你怎么才能知道源数据集中哪些数据被修改过，万一用户数据和当前数据与源数据都不一致，那么哪一个该获胜？在某些情形中，最好的方法就是向用户询问部分或所有的合并。
- *询问*：这是与用户交谈的确定对话，“数据已经被改变了，你想放弃、覆盖还是合并？”正如已经描述过的那样，合并最多只能算是一种艰难的指望，因此通常“询问”方法直接寻求用当前的新数据重载该数据，并让用户再次去编辑那些需要改变的数据，这是一种混杂的放弃/询问方法。

没有一种方法可以适合所有的系统——在很多情况下，你可能会使用不止一种解决方案，甚至在代码的同一模块或片断内也是如此。这些都依赖于业务和领域逻辑的需要。

在某些情形下，你可能甚至想往数据中添加临时轴；通过添加start和end时间戳列，表示数据曾经活跃的“活动”期，然后创建包含最初的主键列以及start和end列的组合主键，这样你就创建了该表中每一行的历史记录：

```
CREATE TABLE person
  ( primary_key INTEGER,
    first_name VARCHAR(120),
    last_name VARCHAR(120),
    other interesting data elements,
    start TIMESTAMP,
    end TIMESTAMP ),
      PRIMARY KEY (primary_key, start, end);
```

为给定的primary_key[4]读取出“最新”的行也就意味着要去查询那些end列被设置为0或NULL（如果你的数据库支持主键可以是NULL的话），或者一些无意义的值的记录：

```
SELECT first_name, last_name, ... FROM person
```

[4] 记住，既然start和end列是主键的一部分，我们就可以获得多行primary_key相同的数据。这样做是经过深思熟虑的——我们想获得表中每个“逻辑”行的多个版本。

```
  WHERE primary_key=? AND end=0
```

对表的更新看似很简单，因为你只是UPDATE现有的行，通过填充end列来关闭记录的生存时间，以及INSERT一个新行（用新的数据），其中end列的值为NULL：

```
Date now = new java.util.Date();
String sql = "UPDATE person SET (end=?) " +
   "WHERE primary_key=? AND end=NULL";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setTimestamp(1, now);
ps.setInteger(2, primaryKeyValue);
int rows = ps.executeUpdate();
if (rows != 1)
  throw new InconsistentDatabaseException(
    "Something's rotten...");
sql =
   "INSERT INTO person(" +
   "primary_key, first_name, last_name,..., start)" +
   "VALUES (?, ?, ?, ..., ?)";
ps = conn.prepareStatement(sql);
ps.setInteger(1, primaryKeyValue);
ps.setString(2, firstName);
ps.setString(3, lastName);
// ... Fill in any other data for the person table as well
ps.setTimestamp(10, new java.util.Date(now.getTime()+1));
  // In other words, 1 millisecond after the
  // end of the previous record
rows = ps.executeUpdate();
if (rows != 1)
  throw new InconsistentDatabaseException(
    "Something's rotten...");
```

记住，这些所有语句都应该作为事务的一部分来实现，以消除UPDATE 和INSERT语句之间的竞争状态。删除也同样简单：你只是通过UPDATE行来填充end列，并且不要插入一条具有给定的start和空的end列的新记录。

不过，这意味着从来没有真正将数据从数据库中删除，也意味着通过在你的每个审计日志语句上都包含时间戳，你就可以重新创建任何特定时刻的数据库状态——这是一种强大的审计工具。它可以帮助简化你的编程逻辑，因为你实际上从来都没有通过UPDATE现有的行来修

改数据（即使你必须在某个地方，通过UPDATE来填充end列）。相反，你只是INSERT一个新行，在该行插入新数据，并且在start列带有当前的时间戳。实际上，只有在你想要实现任何特殊案例的逻辑时，才是第一个逻辑行被录入的时刻（换句话说，就是要插入新的被主键标识的行的时候），因为，没有任何相应的UPDATE来填充“先前”版本的end列。

将讨论返回到乐观并发，客户端以及从数据库读取的、和它们一起运转的数据集，通过检测查看它们的start列是否匹配最新的行的start列，来验证它们所持有的数据的连续真实性。如果二者不匹配，显然有人已经修改过数据，我们返回到前面提到的四种决策：放弃、覆盖、合并和询问。

乐观并发提供了一系列好处，首要好处是：在修改先前读取的数据期间，不需要取出任何本地数据库上的锁。换句话说，客户端读取数据的代码不必“预先”决定是否要取出锁。相反，我们是直接读取作为数据一部分的乐观标识（版本号和时间戳），稍后，当你想将数据更新到数据库中时，你将检测查看数据是否修改过。然后，只有在实际的修改中（UPDATE语句），你才需要某种类型的事务锁，而这些通常是由数据库和/或JDBC驱动来为你处理的（如果它处于自动提交模式）。

乐观并发方法带来的另外一个好处，就是提高了诊断能力。万一其他用户修改了数据，你的代码就有机会告知你的用户发生了什么，以及应该怎样处理。如果使用了本地数据库锁，那么就会没有任何关于为什么会抛出SQLException异常的信息可以获得，除非数据库本身选择要显示它。正如Nock指出的那样，“你可以剪裁出包含了任何最适合用户诊断的信息的通告。例如，告知你的用户谁最近更新过数据以及他是在什么时间修改的，可能会很有帮助。这种类型的信息可以让用户更加意识到并发问题，在长期中断返回之后，提醒他们刷新工作备份[Nock, 398–399]”。

遗憾的是，乐观并发模型并不完美。尤其是，你依赖于应用代码来保持并发模型井井有条，这意味着，如果一些其他程序决定访问你的数据库（请参阅第 45 项），你就要在代码和它的编程实现人员的支配下去获得正确的并发模型。正因为如此，你甚至还要受小组其他编程人员的支配来获得正确的模型，因为只有该模型才是将并发模型细节隐藏在某种封装层后面的良好案例（请参阅第 42 项）。

并且，你会发现乐观并发模型给予你的控制并不像你想要的那样能够控制每一样事物该如何一起运转，那么。例如，你可能想在系统中创建“用于读”的锁或者“用于写”的锁，该系统的一个用户指明想要具有在一段时间内排他地访问一行或一个行集的权限。

然而，你仍然要不情愿地退回到使用本地数据库锁的状态，一方面是由于可扩展性的原因，另一方面是由于可能你想要的对加锁的控制比数据库本身允许的粒度要稍微细一点。在这种情形下，你应该考虑构建某种悲观并发模型（请参阅第 34 项），它可以和乐观并发很好地和平共处，只要你不在同一数据上同时使用它们两个。

很多面向关系的数据访问工具库（JDO 实现以及它的类似物）可以“在 box 之外”为你提供乐观并发机制。如果你使用的工具库能够提供这种机制，那么除非你有坚定的理由不去利用它（请参阅第 11 项），否则你就应该利用它，特别是，既然通常它需要你做的最多不过是在配置文件或持久性描述符中的编辑一行，那么为什么不利用它呢？

通常，乐观并发模型纯粹被用作为一种可扩展性的措施，通过减少获取的本地数据库锁的实际数量以及它们持有的时间长度，来最小化锁窗口（请参阅第 29 项）。通常，对大多数企业级 Java 系统来说，乐观并发模型可能应该是缺省的并发模型——依赖于底层数据库来提供在短期内让所有的并发都能够运行的能力，但是最终当你扩展系统的时候，仍将会引发竞争。

第 34 项：为了显式的并发控制而考虑使用悲观的并发机制

再次考虑一下在第 33 项中引入的两个用户和一个数据集的问题。每个用户读取数据集，进行本地修改，然后将这些修改更新回数据存储中。如果没有某种类型的控制机制，我们将得到一个“最后一个获胜”的场景——最后的用户可以有效地将他或她的修改写到数据存储中，并覆盖掉任何其他用户在前面所做的任何修改。对大多数应用来说，大多数用户都会认为这样丢失一个或更多的更新是一种非常不好的想法。

乐观锁定（请参阅第 33 项）通过将每个数据集标识上某种类型的“标识”，也只能部分地解

决这个问题，这种标识会指示数据在什么时间被其它某一方更新过，因此当数据更新时，我们就可以意识到这个修改。遗憾的是，乐观锁定假设：在更新可以被接受的时候要接收到通告——也就是说，直到我们要真正提交回我们的瞬时状态之前，我们都不知道对底层数据所作的修改。对于某些情形，像无处不在的电子商务购物车，让它们放弃瞬时状态是很普通的，这是可以接受的——因为，很多用户的购物车实际上从来都不会成为真正的订单，因此对数据库的更新从来都不会发生。

不过，在其它情形中，在一个更新将要引发一个问题之前，乐观并发模型不会发送通告。考虑一个 Web 应用，要求用户通过复杂的 12 屏的处理才能更新一个订单。如果你由于一个错误消息“对不起，别人已经更新过这个订单”而返回，并且强制用户从第一个页面重新开始，那么你就要小心，因为很快你就要开始找寻另一份新的工作了。比这更微妙的是，——即使其他用户已经“锁住”了某项数据，但是有时系统仍然不得不允许对数据库进行更新。一些用户可能是“超级用户”，系统管理员或者超级用户能够指定某些用户不再具有从这个电子商务网站购买东西的能力（可能是因为信用级别太低或者是有太多的银行退票）。或者，有可能，我们需要对数据库中所有的货物项进行一次全球性的价格修改（从 12 月 26 号开始半价销售），并且不管有多少用户已经打开了购物车，这个修改都要奏效。

在这种情形下，底层资源管理器的乐观并发模型和本地锁定模型对我们来说统统不行，并且也没有其它任何资源了，只有靠我们自行解决。这种技术称作悲观并发模型（Pessimistic Concurrency Model）（也称作悲观脱机锁（Pessimistic Offline Lock））[Fowler, 426]，在有些地方称作悲观锁（Pessimistic Lock）[Nock, 405]）。尽管它也有若干缺点，但是它提供给你的所有好处就是：让你，编程人员能够对要执行的事物进行更多的控制。（之所以称之为“悲观”，是因为它假定其他方会试图更新你当前正在使用的同一数据——我们很“悲观”，因为我们使用了质朴的并发控制。）在这种模型中，应用的编程人员假定对所有的并发语义都可以进行完全的控制。

实现悲观并发方案相当简单易懂：在获得任何数据之前，你的代码检测通用“锁”表来看看那行是否是未加锁的：

```
SELECT userid, lock_start, comment FROM lock_tbl
```

```
  WHERE lock_table=? AND lock_pk=?
```

这里，第一个参数是你想取得数据的表，第二个参数是你想获得的那行数据的主键值。如果结果集返回为空，那么在那行数据上就不存在任何锁，在获取你感兴趣的数据之前，通过在锁表中执行INSERT命令，你将首次创建一个锁。当你处理完数据的时候，要保证在那行数据上执行DELETE命令，顺便说一下，否则这项数据会永远被锁定，因此对系统的其它部分来说就无法访问了。

在某些情况下，你可能在表定义中添加lock_user列和/或 lock_timestamp列，以此来包含用户的ID和/或时间戳，用于指示谁锁定了这一行以及是在什么时间锁定的，从而将锁信息置于每行之中。两种方式相比，第一个方式对数据表侵入得比较少，而第二个方式则更简单。

如果锁查询返回的是非零行集，那么就表明数据就被锁定了，你可以要么向用户显示一个消息，告诉他们数据当前正在使用中（例如，“用户‘FRED’当前正在处理你所请求的数据——如果你要求访问该数据，请联系该用户”），并抛弃这个请求，要么静静等待指定的一段时间然后再试。

第一眼看去，就觉得这似乎需要做很多工作。是的，对这一点没有什么争论。更糟的是，要求做的很多工作并不是那么显而易见。例如，为了避免产生竞争，即两个客户端查询锁表，发现锁当前并没有被取出，于是同时取出一把锁加到同一行上（因此，第二个用户的锁就会覆盖掉第一个用户的锁），这种锁-查询/获得-锁-更新循环必须在本地事务语义下完成。因此修改后的算法如下：

```
BEGIN TRANSACTION
SELECT userid, lock_start, comment FROM lock_tbl
  WHERE lock_table=? AND lock_pk=?
(assuming no rows returned, continue with:)
INSERT INTO lock_tbl (lock_table, lock_pk, userid,
                  lock_start, comment)
  VALUES (?, ?, ?, ?, ?)
COMMIT
(having locked the data, now you can continue with:)
SELECT * FROM table WHERE . . .
```

在失败情况下，你仍需要小心翼翼地释放锁；这意味着如果抛出 Java 异常或者发生其它的某些失败，你的处理代码中的异常处理操作将会从锁表中删除这个锁。对每一个你想运行的 SQL 语句来说，都有很多工作要做，最好是将其隐藏到过程优先的持久性框架之后（请参阅第 42 项）。

悲观模型的回报带来了这个事实：你对发生的并发有了大得多的控制权；例如，为了“旁路”一个悲观锁，你不必先去检测锁表，相反而是依赖底层的数据库事务框架来防止任何数据污染。一般而言，只有在最迫切的情况下才会这么做，因为如果这些情况能够被正确处理，那么客户端可能会在某些非常奇怪的情形中结束——例如，如果刚好在“你确定提交这些订单吗？”显示之后，对价格的修改进行了更新，那么填写购物车的用户可能会突然发现，从他或她的信用卡上收取的费用和前面的视窗中所显示的总费用不同。

可供选择地，你可能会需要这样的情形：超级用户能够“中断”给定的数据行或数据块上的锁。可能系统已经进入了某种死锁状态，或者超级用户需要将数据重新分配给不同的用户。（“JOE？Fred 度假去了，并且留下了许多没有完成的订单项。你能将它们分配给 George 吗？这样我们就能够最终处理掉那些 Fred 发起的未完成的订单了。”）

实际上你可以在你的并发模型中获得你想要的精致和技巧。也许两个用户有可能同时持有一个特定行上的锁，只要他们是在同一个团队内工作，即隐含地假设他们可以通过在卧室走廊里进行的令人生厌的交谈，就可以解决他们之间的任何数据并发问题。或者某些用户不管数据是否被锁定，都可以获得对所有数据的只读访问权限，但是不能进行任何更新。还有很多种可能性。

使用悲观锁定方案与依赖于本地并发相比较，其诊断也要更简单一些，而且提供的信息也更多一些。当用户请求的数据已经被另一个用户使用时，诊断消息可以表明谁持有这个锁，以及已经持有了多长时间，并且对于总是在线的系统来说，你甚至可以更进一步，向正在持有锁的用户弹出某种消息，询问她或他是否可以释放了。你可能还编写了某种守护进程来定期地浏览锁表，找出比 X 秒/分钟/小时/天/月/年或任何时间要更长的锁，在它可能成为一个死锁或应用故障时，自动地将它们释放（并向持有锁的用户发送适当的通知）。

在某些情况下，悲观并发是你能够得到的唯一形式。例如，当处理文件系统中的文件时，因为大多数文件系统仍旧不能向你提供任何类型的 X/Open 事务接口（请参阅第 32 项），为了确保你能够排他地访问一个特殊的文件，你通常会适当地需要某些形式的“外部”并发。这仍是前面讨论的锁表问题，或是很多老系统所使用的锁文件的概念，锁文件是一种与锁表同样意图的文件：创建锁文件（一般通过调用某种形式的“开放的”操作系统 API 来实现，对于该 API，如果文件已经存在，则返回错误），执行处理工作，接着当工作完成时，删除文件。请注意，你可以使用锁文件本身来持有那些通常存储在锁表中的数据——用户 ID 和注释这一类的事情——文件创建时的时间戳可以作为获取锁的实际时间。这种方法还具有一个好处，就是释放锁特别简单：只需删除锁文件。（怀疑整个这种方式能否运转？如果你使用 CVS 作为你的源文件控制系统，那么你时时刻刻都在使用锁文件——当你在向源文件池中提交源文件的时候，CVS 就是使用它们来确保排他性访问的。）

如果你特别雄心勃勃，那么你甚至可以试图在整个悲观并发方案中，创建 JTA 兼容的资源管理器，但是这么做通常会产生比大多数程序员愿意承担的工作量要多得多的工作。保持事务处理模型编程实现的简单性的同时（请参阅第 27 项），提供悲观并发模型还有一个独特的好处，那就是如果你在同时处理多个资源管理器，并且需要在一个资源上使用悲观并发时，这就显得极其有用了。

在系统的并发能力方面，悲观并发方案提供给你巨大的力量；不过，记着，正如漫画书中的英雄所领悟到的艰辛之路：“强大的力量伴随着强大的责任。”悲观并发给予你极大的灵活性，但是这取决于你去确保你所选择的并发决策是可靠的；否则，数据污染就是最终结果。

第 35 项：考虑使用较低的隔离级别以获得更大的事务吞吐量

考虑一下一个 ACID 事务的隔离属性。它在事务之间筑起了一道“墙”，在事务要么完成要么回滚之前，把对数据集所做的任何修改滤掉。它这样做有很好的原因：想象一下，你是否愿意在这样一个世界里，在那里事务不被隔离，并且可能将它们的结果“泄漏”给其它当前正在执行的事务。两方人员，让我们叫他们 Bob 和 Jane，在同一时间都要访问数据库；让

我们假设他们是已婚夫妇，每个人都试图在不同地点对他们的银行账户进行处理。根据每个人试图要做的事情和在什么时间去做，会发生几种结果：

- *丢失更新*：Bob 在银行从支票账户读取他们的支票账户行。一旦 Bob 读取完成，Jane 接着也读取同样的行。Jane 的下一个动作是，一个将$100 添加到她所持有的总额上的更新（一次存款），并将它写回到数据库中。现在，Bob 也开始向他持有的总额添加$100，然后写回到数据库。接着，每个事务都提交了。注意 Jane 存的款现在已经丢失——如果开始时账户上有$500，现在读出的则是$600。
- *脏读*：Bob 从支票账户表中读取他们的支票账户行，然后向总额添加$100，并写回。接着，Jane 读取那行（可能她想对账户余额进行更新）。遗憾的是，由于 Bob 出了点错，所以他现在要回滚他的工作。但是 Jane 会相信账户上比实际金额多了$100，因为她已经看到了未提交的处理。
- *不可重复读*：Jane 读取账户结余。Bob 也读取账户结余。Jane 接着又修改数据，现在向账户结余行存入$100。Bob 打算谨慎行事，重读数据，对他来说，他之前几秒刚刚读取过，但是他突然又得到了和之前不同的值。
- *幻影读*：从 Bob 和 Jane 的角度比较难于理解这个概念，因为我们现在处理的是多行数据。从本质上讲，假定 Bob 不仅查看支票账户，还查看他们两个之间的全部资产。Jane，在 Bob 请求完结之后，突然决定在银行开设一个新的退休账户。当 Bob 决定以完全相同的视角进行刷新时，他突然发现了几秒前并不存在的退休账户。

如果没有出现上述这些结果，那么，当事务“完全使能”时，其缺省场景就是串行化访问（serializable access），其本质是在模仿这样一个环境：一个事务可能会认为它是整个数据库中的唯一开放事务——似乎每一个事务都在排队等候（串行化），一个接一个地执行。这是执行中最安全的模式，但是也需要最多的锁，因此也表示最激烈的竞争冲突。

通常，大多数开发者会仔细地查看上面的列表或可能的问题，并会不禁疑问到为什么我会引起它们——毕竟，像给定的数据库查询可能不会返回“正确的”数据这样的暗示通常都足以使大多数开发者远远地规避这种想法。但是，我们不要那么快地完全丢弃掉它。

想象一下，我有一个当前生活在加利福尼亚的萨克拉门托市所有人的数据库，这个城市的人

口超过 25 万——这是一个相当大的数据库。我想运行一个查询，去查找职业是计算机程序员的那些人的平均年龄。这个查询执行起来不会很快，在此期间，数据将会被修改——有人被录用了，有人被解雇了等等。假定我一直不允许更新数据，直到查询结束为止——一旦查询返回，该查询仍是良好的吗？对数据所作的新修改不大可能引起平均年龄产生很大的变化；实际上，完全有可能没有任何人注意到这个不同。这在统计学上被称为给定数据集范围内的“误差幅度（margin of error）”。

现在考虑一下在查询期间持有数据上的锁所需的开销：在查询执行期间，每个访问该数据的人都被锁给关在了外边，这样就增加了我们的应用的响应延迟，并降低了应用的可扩展性。一旦查询完成，该查询有相当大的可能性会一去不返，那么为了这样一个偶尔才会执行的查询而这么做，值得吗？

当然，并不是所有的数据事务都适合这种松散的模型：例如，对于包含存款、取款和转账的财政信息系统，如果一个来自于某个账户的取款不小心被结算成给另一个不同账户的存款，那么该系统就不会长久地成功。锁代价对于这样的系统当然是值得的。但是在应用开发期间，很多情况属于这种范畴，即系统允许数据的微小不同步，尤其是如果这意味着，我们可以减少整个系统的竞争冲突。

因此，大多数据库系统支持降低事务隔离级别这种思想，要么通过一个直接的 SQL 调用，要么通过在 Java 中被称为 JDBC 的调用级接口来实现。下面标准的隔离级别以及作为结果所产生的影响都可以在标准的 SQL 中获得：

- `SERIALIZABLE`：不会产生上面讲到的任何影响。
- `REPEATABLE READ`：此隔离级别确保对用户维护相同的数据库视图——在那里的数据仍然在那里。它防止脏读和不可重复读发生，但是允许幻影读发生。
- `READ COMMITTED`：这个级别允许客户端事务查看其它已提交事务的动作，因此可以防止脏读，但是允许不可重复读和幻影读。
- `READ UNCOMMITTED`：这个级别能够防止混乱，它允许用户查看其它还未提交（或可能无法提交）的事务的动作。它允许脏读、不可重复读以及幻读；简言之，除了直接污染数据，它允许很多事情。

注意，事务的其它三个属性仍旧是完全有效的：隔离级别为 READ UNCOMMITTED 的事务仍旧是原子性的，仍旧是一致性的，仍旧是持久性的——它要么全部成功，要么全部失败，它总是在数据库上产生相同的效果，并且当事务完成的时候，它的动作就会被存储起来。这里唯一被改变的事情是事务对系统其它部分的可见性，反之亦然。

记住，这里的好处就是被降低的隔离级别可以降低竞争冲突。当对数据库直接执行更新逻辑时，我们可能需要运行SERIALIZABLE隔离级别上的更新——用这种方式，这个更新肯定能在“正确的”数据集上运行，并且不会意外地被一个对底层数据的修改所湮没。不过，对于查询、选取列表之类的操作，降低到REPEATABLE READ隔离级别可以帮助减少系统中锁的数量。对于需要在某个预定时刻从数据库中取出大量数据的处理（从一个数据库取出数据，然后将其压入另一个不同的数据库，也可能是数据仓库中的批处理）来说，在READ COMMITTED级别上运行可以使竞争冲突更低，因为它允许用户甚至是在批处理正在运行的时刻，也能够继续处理数据库。对于涉及到很多数据库，但是精度要求不是很高的一页的行政总结报告（“我们在这个季度花费了多少钱？”）来说，应该使用READ UNCOMMITTED隔离级别。实际上，大多数报告可以运行在READ UNCOMMITTED级别上，而不会引起特别的注意。（试一下——不用这个级别运行一次，然后再用这个级别运行一次，看看你是否能够发现有变化。）

在 SQL 级别上设置隔离级别要使用 SQL 语法：

```
SET TRANSACTION ISOLATION LEVEL <level>
<level> ::=
  READ UNCOMMITTED |
  READ COMMITTED |
  REPEATABLE READ |
  SERIALIZABLE
```

然而，如果要是在事务外面（也就是在执行BEGIN TRANSACTION语句之前）这么做，就要小心。如果你确实这样做了，那这就像是大多数数据库提供商承诺的“未定义的行为”，更委婉的说法是“在很大程度上它将会崩溃，可能正好是在重要的演示期间当着你老板的面发生。”

在JDBC级别上做同样的事情，要使用Connection的setTransactionIsolation方法，像这样：

```
import java.sql.*;
public void run(Connection conn)
  throws SQLException
{
  conn.setTransactionIsolation(
    Connection.TRANSACTION_READ_UNCOMMITTED);
  // or TRANSACTION_READ_COMMITTED,
  // or TRANSACTION_REPEATABLE_READ,
  // or the default, TRANSACTION_SERIALIZABLE
  Statement stmt = conn.createStatement();
  // This statement will run at the isolation level
  // established in the lines above
```

注意事务隔离性是一个Connection级别上的属性，这意味着它是为整个数据库的Connection而设置的，而不仅仅是一个Statement。并且，正如上面使用的SQL语义那样，在事务的中间调用setTransaction Isolation是“由实现定义的”，要是引用JDBC文档，那么就又是一个委婉的说法：“几乎可以保证在进行重要演示时你和/或你的公司会倍感尴尬”。

遗憾的是，现在已经让你信服于降低事务隔离性所带来的好处了，但是我们也带来了坏消息：EJB 规范，在很大程度上，对整个在 EJB 容器中支持事务隔离的思想只是虚晃了一枪。EJB2.1 规范的 17.3.2 节是这样陈述的：

> 用于管理隔离级别的 API 是与资源管理器相关的。（因此，EJB 构架没有定义用于管理隔离级别的 API。）……
>
> 对于由 Bean 管理事务边界划分的会话 Bean 和消息驱动 Bean 来说，通过使用与资源管理器相关的 API，Bean 提供者可以在企业级 Bean 的方法中以编程方式指定所期望的事务隔离级别。
>
> 对于由容器管理持久性的实体 Bean 来说，事务隔离是由数据访问类管理的，这些类是由容器提供商的工具所创建的。这些工具必须保证由数据访问类实现的隔

离级别的管理不会导致对在某个事务内的资源管理器的隔离级别请求发生冲突。

宽松地解释一下，可以将它们归结为简单的几点：

首先，如果你想利用降低隔离级别所带来的可扩展性方面的好处，你就必须使用 Bean 管理的事务。这也意味着，你不能在实体 Bean 时期使用事务隔离，就是这样，因为 EJB 规范很清楚地陈述到，无论你现在正在使用的是容器管理的还是 Bean 管理的持久性，总是要在容器管理的事务下运行（17.3.1 节）。如果你的容器没有提供本地事务优化（请参阅第 32 项），那么你就得考虑在每个单独的实体 Bean 访问上执行完整的两阶段提交。你可能试图通过将实体 Bean 的事务声明设置为 NotSupport、Supports 或 Never 而四处使用它；然而，这样做很快就会使你的 Bean 无法移植。17.4.1 节添加了重点声明，“容器可以可选地支持将容器管理持久性的实体 Bean 方法设置为 NotSupport、 Supports 和 Never 事务属性的用户。但是，使用这些事务属性的容器管理持久性的实体 Bean 将变得不可移植。”哦，这是另一个决定可移植性对你是否重要的原因（请参阅第 11 项）。

第二，如果你想利用较低的隔离级别，你需要到资源管理器API层次上去实现；对于关系型数据库，这意味着你必须得到Connection对象，然后调用Connection本身的setIsolationLevel方法去降低它的隔离级别。这并不很难；但是，要确保在你打开Bean管理的事务之前去这么做，因为正如我们已经注意到的那样，在事务内改变隔离级别是一种产生长时间的调试期的快速方式。

最后，如果你确实要在 Bean 管理事务的 Bean 中着手降低隔离级别，那么你要非常非常小心，因为规范的最后一句就像是一个杀手——从本质上讲，容器必须确保不同的隔离级别不会出现在一个单一的事务中。然而，对于怎样执行这条规则，规范没有提供任何线索，因此，如果具有较低隔离级别的 Bean 管理事务的会话 Bean 调用实体 Bean 时，会发生什么呢？不为你自己的特定容器尝试一下，你是永远没办法知道结果的。危险就在于你的容器会麻痹你，让你认为你是安全的，因为你的容器将会默默地为你处理这个问题。但是，其它容器可能不会，并且当然不会处理这个问题，所以，到你将应用移植到另一个容器的时候，异常就会大量地抛出。

第 36 项：面临回滚时使用保存点来保留部分工作

遗憾的是，事务的原子性并不总是一件好事情。

尤其是，原子性的事务引起的问题之一就是：当在事务的工作区内某事物出现问题时，那么每件事情就都有问题。一旦数据库语句在事务内失败，那么作为该事务一部分的每一点工作行为现在都被视为是无效的，因此必须放弃。

通常情况下，我们认为这种行为是积极的。毕竟，我们需要原子性事务的原因在于：万一发生错误时，我们不必编写回滚或者补偿事务的代码（请参阅第 28 项）来撤销我们已经完成的工作。尽管这种行为是我们在大多数时间内都想拥有的行为，但是有时它也会不符合我们的想法。

正如我们在旅行计划场景（请再次参阅第 28 项）中看到的那样，我们并不总是想放弃到目前为止被执行的工作的整个范围。在简单场景中，这种要么全有要么全无的模式能够很好地运转，但是某些动作沿途构建了较多的上下文，并且万一失败时，并非所有这些上下文都应该被视为无效——不能仅仅因为我们没有订到最后一段旅程从法兰克福到慕尼黑的航班，就意味着旅行中开始的两段旅程应该被丢弃。

将这个问题更系统化、更具体化地阐明，考虑一个计算所有存款账户的利息的银行系统。一种实现方法是在EJB容器内设置消息驱动Bean，并且设置新的EJB定时器服务，使得能够在特定时间发送消息。当收到这条消息时，InterestCalculatingBean执行相当直接的SQL UPDATE语句按照当前利率来增加所有存储账户的结余。很自然地，我们想确保这个计算是完整的；更准确地讲，我们需要确保我们的账户持有者的所有账户都被更新，因此我们将它置于一个单一事务中以确保这个动作是原子性的。

遗憾的是，我们的银行用户超于一千万，那么要花费相当长的时间在一千万行上执行UPDATE语句（尤其是，如果SQL语句很复杂。例如只更新那些多于最小结余的存款账户和那些VIP所持有的账户以及那些在 2 月 15 号和 4 月 15 号之间由会员开户的账户——你相信了吧）。即使在重型装备上，这种操作也可能会很轻易地就花费掉几个小时，获取是通宵达

旦，才能运行成功。

在第 9,947,831 行，我们遇到问题了，那么UPDATE失败，并且整个事务必须放弃。猜想一下，我们得在下个月再试一次，对吗？毕竟，利息可能不再是那么多钱了……

这只是标准的事务语义不能很好地为我们服务的情形之一——真的不应该在失败之前放弃已完成的所有工作，因为我们真正需要做的只是标识失败行，然后继续向前。

正是由于这些原因，JDBC3.0 规范在 API 中引入了一个新的概念，称作保存点（savepoint）。这个想法相当简单：它有效地树立了一个标志点，告诉通过这个点的数据库，在此时（在此刻保存点被创建），每件事看上去都运转良好，可以被安全地提交。注意这句话中很谨慎的一个字眼：“可以被提交，”而不是“继续并提交”——如果需要的话，我们可以回滚被保存点标识的工作。

当出现错误时，保存点的好处就显现出来了。如果事务的一部分突然失败，我们可以回滚事务到最近的保存点，因此仅仅放弃由最近的保存点作为边界的已完成工作，然后重新开始；本质上讲，保存点机制允许放弃我们旅行日程中不幸的第三段旅程航班，而保留前两段旅程航班作为已建立事务的一部分。

使用JDBC 的Savepoint相当浅显直接。当使用Connection时，在事务执行期间的任何时刻，用一个String命名Savepoint或者不使用任何参数，而让系统为你选择一个名字，来调用setSavepoint，然后会返回一个Savepoint对象：

```
Connection conn = ...; // Get this from someplace
Statement stmt = conn.createStatement();
stmt.executeUpdate(someSQLHere);
Savepoint svpt = conn.setSavepoint();
stmt.executeUpdate(someOtherSQL);
Savepoint svpt2 = conn.setSavepoint();
ResultSet rs = stmt.executeQuery(SQLToTestWorkSoFar);
if (rs.next())
{
  // Whoops! There's not supposed to be anything in here yet!
```

```
    // Time to abort part of the work
    //
    conn.rollback(svpt);
      // Note that we can roll back to any Savepoint, not just
      // the last one marked
    stmt.executeUpdate(someCorrectiveSQL);
  }
  stmt.executeUpdate(someMoreSQL);
  conn.commit();
```

保存点提供了一个机会来保存已经在一个事务中完成的工作，因此，我们可以选择事务工作是原子性的还是非原子性的。这是好消息；坏消息是保存点模式是JDBC3.0 规范中的新成员，到写这本书时为止，只有少数的数据库驱动真正支持它。如果你想了解自己现在正在使用JDBC驱动，那就要确保去查实它是否支持保存点（细节请参阅第 49 项，并使用DatabaseMetaData. supportsSavepoints去查实）。

第 37 项：当有可能避免锁定区域时就复制数据源

想象你现在是一个幼儿园老师。20 个孩子，差不多都是 5 岁的样子，似乎永远都在你的照料之下，而实际上只是每天 4 个小时。当休息时，你带他们到操场上活动筋骨，释放能量。小 Johnny 抓住绳子开始跳。很自然地，小孩子有他们的天性，另外 19 个孩子也会一起大叫到“不公平！”他们会在同一时间都想玩这唯一的一条绳子。你首先会告诉他们每个人轮流玩。毕竟，这是你作为一名教师应该做的：教导孩子们成为世界上行为端正的公民，开始教导怎样去分享。

任何提出这种建议的人，永远都不可能让这 19 个 5 岁的孩子站成一排去等待超过 2 分钟。

问题是：假定每个孩子可以拿绳子跳一分钟，之后必须将绳子交给队列中的下一个孩子，然后回到队列后面。这意味着，每个孩子在 19 分钟内完全不做任何事情只是等待（假定，每个孩子在他或她的轮次结束的时候，很乐意将绳子交出，完全不像个 5 岁孩子）。也就是，每 20 分钟才能活动一分钟，那么也就没有怎么释放能量。

解决方案是什么？一种方法是让每个孩子跳得更快，每个轮次是 30 秒，但是这样做加速了队列的前进速度——不过，它的比率实际上仍是每 20 分钟活动一分钟。相反，你可能会通过跳跃的次数来计算孩子们的轮次，但是这样会惩罚那些跳得比较快的孩子，速度慢的孩子会因此而更长时间地获得绳子，甚至会让跳得快的跳绳者活动得更少。

如果询问其他幼儿园老师应该怎样解决这个问题，他们都会告诉你：买更多的跳绳。

单一资源的关键问题是：这些资源通常会创造出竞争点，以至于必须使用同步结构来管理（要么是 Java 对象监视器，要么是数据库锁，或者其它任何事物）。一个单一竞争点会引起系统无法扩展，因为添加的硬件只会引入更多的客户去竞争这个单一竞争点；换句话说，当我们试图将幼儿园教室从 20 个学生扩展到 200 或者 2000，这种只有一条跳绳的情况会变得更糟。

因此，当我们想起第 21 项时，我们开始寻找方法去划分资源，这次是通过消除竞争点来减少系统的负载。

在这个最简单的情况下，我们要做的只是选择创建复制的Java对象，而不是试图强制所有的处理通过一个单一的对象；例如，假设有一个必须处理某种日期格式的servlet/JSP。将你的对象缓冲起来可能会比较吸引人（它本身就是一种天生不好想法，正如第 72 项所述），仅仅创建一个SimpleDateFormat实例让所有对Servlet进行调用的调用者去使用，但是你很快就会发现SimpleDateFormat不是线程安全的，并且必须被同步。

在这里暂停一会儿，回顾一下。在这种情形下，只有单一资源——SimpleDateFormat实例——的开销，会远远地超过获得和释放对象监视器的开销。通过在到来的每个HTTP请求上创建SimpleDateFormat，而将它复制，这样就会消除竞争点并让线程继续按部就班地运转。尤其是，这样做很容易获益，因为SimpleDateFormat本身并没有任何标识（请参阅第 5 项），从本质上讲，就是只需在构建的时候赋予它特定的状态，以便指示它应该是那种格式。

当我们开始考虑复制标识绑定的资源时，最典型的就是数据库（或者是它的延伸，像实体Bean或者JDO中的持久对象），如果使用复制的资源，那么我们就会开始陷入复杂的情形中。我们再次地陷入到了更新传播问题中，在第 21 项中介绍过——如果我们更新机器A上Person

的备份记录，我们需要确保在集群中其它每个机器上的同一条Person记录的每个备份都要更新，否则，读取同一条Person记录的客户端毫无疑问地会获得不同的值，因此也就破坏了一致性。我们的问题是，不仅需要运行在此JVM上的所有线程，还需要运行在其它机器上的所有JVM中的所有线程都同步这些更新传播，因为这个锁必须遍及整个集群。这就像整个学校只有唯一的一根绳子，而不是每个教室一根。

先返回到前面的技巧，再次考虑 DNS 问题。我们不想必须到单一服务器上去查找每一条我们要查找的 DNS 记录，因此 DNS 客户端通常是将他们检索到的 DNS 设置在本地备份缓冲起来。这里很显然存在着标识绑定的问题——如果我修改了 neward.net 的 IP 地址，那么就只有一个“真正的”记录持有真实的地址，也是我的 DNS 服务器。如果客户端将这些记录缓冲起来，他们怎样才能知道什么时间才能检索到新值？

在这种特殊情况下，DNS有力地陈述道：它可以承担特定的延时，并为每个DNS记录分派生存时间值(TTL, time-to-live)。这个值表明了确保DNS记录中所描述的数据处于良好状态的时间，并且对这些值进行追踪以及“返回到源点去刷新”是客户端集体的责任，“返回到源点去刷新”的时机取决于客户端是否能够确保它们所使用的是最近最好的值。在这种情形下，延时就可以接受的——我可能让旧的服务器和新的服务器并行运行（假定这两个服务器在内容和功能上完全一样），直到我知道这些DNS记录已经被传遍到世界上所有的DNS缓冲集合为止。

具有讽刺意味的是，这里给予我们关注的是同步和锁窗口问题，因为 DNS 数据通常是只读的或者以读为主的，我们可能实际上就是将它看作是永恒不变的（请参阅第 38 项），这再次为它的复制提供了便利。实际上，对于各种只读或者以读为主的数据，没有任何理由不让其在网络中复制，因为不需要任何更新，所以也就不存在更新传播的问题，因此也没有任何原因不去复制。

我们可以将 DNS 方法扩展成一种更泛化的思想，即从集中的数据库中租借（leasing）数据的思想。本质上讲，通过取出伴随数据本身的 TTL 值，并且在我们将数据复制到本地备份的时候，将它随数据一起存储起来，我们会可以“借”数据了。接着，当我们处理数据的时候，如果 TTL 的值和当前时间的比较表明数据已经到期限了，那么我们就返回到集中数据

库去进行一次刷新。再次强调一下，它并不如像银行账户结余之类必须绝对精确的数据那么有用，但是在很多系统中，相当多数量的数据都可以忍受这种程度的延时。

实际上，在某些情形中，你可以将本地数据库和乐观并发结合起来（请参阅第 33 项），以完全在本地数据库上运行你的应用，只是在众所周知的同步点上从数据库中压入和取出数据（每晚、每小时，或者每当网络被探测到的时候，等等）。不过，再次提醒，你需要确保应用可以处理不可避免的并发问题，例如，在星期一，每个条目的价格都已经被修改到了中央数据库中，因为我们仍没有进行同步，因此在星期二下的订单所使用的价格仍旧是星期六的。（顺便说一下，在这种情况下会发生什么事情是商业决策，而不是技术决策——不要对应该发生什么做任何假设，或者你的这份工作将会因一些非常不爽的用户和/或客户而告终结。）

复制不能解决所有的可扩展性问题，不过当我们谨慎使用时，它可以为单一资源缓解相当大的一部分压力。要想让复制成功的关键在于，要最终确保你复制的资源是没有标识的，因为一旦复制的是独一无二的数据，我们就会陷入一致性的问题，对于这个问题，仍没有任何很好的通用解决方案。

第 38 项：偏爱不可变的，因为它不需要任何锁

在通读过本章后，你可能开始失望了——毕竟，如果 Java、事务甚至 EJB 规范都不能阻止你去担心同步问题（或者更准确地说，如果解决方案产生的结果如此不能令人信服，以致于你必须开始主动地去担心它们了），那么那种幸福的什么都不用担心的时代将一去不返。你必须考虑同步锁、竞争冲突、死锁、活动锁以及其它所有相关的事情。

有一种方法可以将你从不得不担心的同步问题中解救出来，那就是通过创建状态永远不会改变的对象，来完全避免它——利用一个不可变对象是不会改变其状态这个事实，同步就完全没有必要了。这样就避免了对synchronized语句块和锁的需要，从而也就避免了竞争冲突。

使用这种方法也带来了几条限制。首先，不可变对象并不是很容易就能够建立的——要想构建一个对象，它没有给可获取其数据的用户以任何后门，着实是看起来容易做起来难。第二，

我们通常需要不可变对象去改变其状态，也就是说，我们想得到一个状态稍微不同的新的不可变对象。这需要构建一个全新的对象，而不是仅仅在现有对象中去梳理一些域，以获得我们所感兴趣的状态。

构建不可变对象比初看起来的时候要难一点；请参阅Effective Java [Bloch, 第 12 项]去查看涉及到的更多细节。额外补充一个例子，看看下面这个标准类，用来表示地球上以碳为基础的生命形式：

```
public class Person
{
  public Person(String firstName, String lastName, int age,
                Address homeAddress, Person spouse)
  {
    // Do what you'd expect here—copy parameters to fields
  }
  public String getFirstName() { return this.firstName; }
  public String getLastName() { return this.lastName; }
  public int getAge() { return age; }
  public Address getHomeAddress() {return this.homeAddress; }
  public Person getSpouse() { return spouse; }
}
```

不仔细看，Person类似乎是完全不可变的——在这个类的API中哪里都没有提供任何设置方法，因此不可能修改类成员，是这样吗？

快速提问：如果我们这样做，会发生什么？

```
Person p = new Person("Michael", "Neward", 10,
                      new Address(...), null);
p.getAddress().setStreet("100 White House Way");
```

可能不会很快就察觉到，Person类“遗漏”了一个被Person类中的homeAddress域所引用Address对象的句柄。这就给用户提供了“曲线迂回到”引用对象（在此情形下就是Person），并直接修改它内部的对象的能力。这就违反了Person的不可变性，会让它遭受到Address对象中的同步问题所造成的影响。

但是，当我们确实想修改Person状态的时候，我们必须创建一个全新的Person实例来实现。最简单的方法就是重用上面定义的构造器，如下所示：

```
Person p = new Person("Michael", "Neward", 10,
         new Address(...), null);
// Michael just had a birthday: increment his age
p = new Person(p.getFirstName(), p.getLastName(),
            p.getAge()+1, p.getAddress(), p.getSpouse());
```

这种方法有一个问题，当进行不相干的同步保持时，就会强迫分配众多的对象，给垃圾收集器带来了更大的工作量，至少从原理上讲是这样。幸运的是，正如第 72 项解释的那样，很多垃圾收集器在面临众多的短期存活的临时对象时，能够很好的运转，因此，这不是一个多大的问题。

如果你遵循一些EJB“最佳实践”书籍上推荐使用的数据传送对象[Fowler, 401]，在EJB客户端和需要将数据设置到数据库的实体之间进行数据传送，那么将数据传送对象设置为不可变对象会有很大的实际意义。相似地，如果你决定在系统的各层之间按值传送所有的数据（请参阅第 23 项），那么，使用不可变对象不仅能确保同步场景不会悄然而至，而且还能确保按值传送的对象不会被意外修改。

第五章 状态管理

对于寻求真理的人而言，有些准则是必须遵守的，真理并非教条或无知，而是通过推理、调查、检验、与探究得来的。无论其意图有多好，信仰都必须构建在事实而非幻想之上，幻想之上的信仰是最糟糕的虚假希望。

—— Thomas Edison

在企业级系统中，大部分工作都涉及数据处理。事实上，可以论证，企业级系统只做了一件事，那就是数据处理。

在两层架构的客户/服务器系统时代，这还不太明显，那时企业级程序员需要关心两种状态：瞬时状态（transient state）与持久状态（durable state）。瞬时状态并不算企业级数据所覆盖的正式部分，持久状态则是无论发生什么都需要被跟踪的部分。

瞬时状态是那些企业并不关心，也不会为之流泪的数据，因为在系统崩溃的时候，真正重要的东西决不会丢失。电子商务中的购物车是瞬时状态典型的例子。首先，我们决不希望系统崩溃，但是让我们暂时带上客观现实的眼镜：如果在顾客购物期间，服务器崩溃了，这会造成购物车中的内容丢失，但那并没有什么实际的损失（除了顾客花在填满购物车上的时间）。顾客也许会很生气，不过似乎也没有什么生意能够令顾客完全不会生气。

在厚客户端或富客户端的应用中，瞬时状态很容易处理：相当于客户端进程中存储在局部变量中的数据，它们没有被保存在持久的存储介质中。当客户端进程结束的时候，瞬时状态也随之消亡，无需为其生命周期的处理而多费心思。

不过在瘦客户端中，例如基于 HTML 浏览器的应用，瞬时状态则呈现出另一种尺度。因为 HTTP 本就是无状态的协议，自身并不具备保存每一个客户端状态的能力。所以要由程序员在底层协议之上自己实现瞬时状态机制。对大多数开发者而言，是通过 HttpSession 机制实现瞬时状态。HttpSession 是 Servlet 2.x 规范的一部分。然而，天下没有免费的午餐，使用 HttpSession 肯定会付出相应的代价，最引人注目的就是整个系统的可扩展性（scalability）。对于希望扩充为超过 5 个并发用户的系统而言，每个客户端会话状态的谨慎使用是至关重要的。

另一方面则是持久状态，在谈到它时人们自然就会想到“持久数据（persistent data）”，即需要长久保存的数据。正规的说法是，如果定义了某个持久状态，那么它就绝对会被保存下来，即使遇到 JVM 终止甚至崩溃的情况也是如此。不过，由于我们令人信服地将瞬时状态保存在数据库中以应对那种状况，所以我们可以直接更方便地说，持久状态就是我们关心的状态。

持久状态通常具有隐含的法律性或经济意义。假如你开发的系统，在顾客用信用卡为买书的订单付账之后，弄丢了订单中的条目，那么你的公司肯定会吃官司。或者完全掉过来，如果你把书给了顾客而账却没结，你可就是直接花掉了公司的钱，很快你就会发现自己需要一份新的简历了。

在讨论状态管理时，二者的区别至关重要。因为对瞬时状态有效的机制对持久状态不一定有效，反之亦然。不过我们也看到，某些情况下会有重叠的部分。例如为避免用户的瞬时状态信息丢失，我们可能会将用户的会话状态记录在关系型数据库中。也许它根本不是商业站点，只是一个人力资源处理过程，需要有一个长时间运行以让人们填写的表单集合；或者它是一次考试，我们可不希望学生只需关闭浏览器就可以“扔掉”这次考试，然后重新开始。这种情况下，瞬时状态与持久状态的区别开始变得模糊，但是在直觉上，通常还是可以相当容易地区分二者。可以论证，在学生考试与人力资源表单的例子中，我们最先想到是持久状态。在心中铭记二者的区别，而设计将有助于你为系统中的状态管理选择相应的处理机制。

第 39 项：节省地使用 HttpSession

在基于 HTML/HTTP 的应用中，为维护代表客户端的瞬时状态，servlet 容器提供了一种称为会话空间的设施，被表示为 HttpSession 接口。这个思想本身是简单而直接的，servlet 程序员可以将任意的可序列化对象（请参阅第 71 项）置于会话空间，而下一次，同一个用户对同一个 Web 应用的任何部分发出请求时，servlet 容器将确保同样的对象会处在 HttpSession 对象中。这使得 servlet 开发者可以为在服务器上的 Web 应用在多个 HTTP 请求之间维护每个客户端的状态信息。

遗憾的是，这种机制并非完全免费的。首先，在服务器端为每个客户端存储数据将会减少该服务器上的可用资源，这意味着服务器的最大负载能力会成比例地下降。这个算式很简单：在会话空间中保存越多的数据，机器能够处理的会话就越少。由此推导出，为了令给定的机器能够支持尽可能多的客户端，必须将会话的存储量保持在最小。实际上，对于真正具备可扩展性的系统而言，无论何时都应该避免使用会话。如果在服务器端可以不产生任何为每个客户端进行处理的开销，那么机器的负载能力(在理论上)可以到达无限，能够支持任意多连接到它的客户端。

避免使用会话的建议不单单是考虑到系统的可扩展性。对于在 Web 集群内运行的 servlet 容器而言，这也是必须的。会话是驻留内存的结构。因为内存是局限于特定的机器，除非 Web 集群有某种机制，能够令给定客户端的每一次请求都被传送给同一个服务器，否则对应先前的某个请求，其后续处理可能会找不到之前存储的会话对象。

顺便说一下，作为一种解决方案，将 HTTP 请求绑定到相同的机器，将使问题变得非常困难。如果网关使用客户端的远程地址作为客户端请求的指示器，那么就会在几个方面陷入问题。为客户端拨号上网提供 IP 地址、代理服务器和 NAT 的互联网服务提供商（ISP）将会为多个客户端提供相同的 IP 地址，这偶然地会产生将所有的客户端对应到同一个服务器的情况。如果同一个代理后面只有少量客户端，则问题还不大，但是如果这个代理服务器是 AOL，问题可就大了。

Servlet 2.2 规范提供了一种可能的解决方案，以应对集群内的会话问题。如果是支持 Servlet 2.2 规范的 servlet 容器，在部署描述符中可以用<distributable/>元素标记 Web 应用，然后 servlet 容器就能够自动地确保会话信息在集群节点间无缝地移动，一般是将会话序列化然后在网络上传送。（这也就是为什么 Servlet 规范要求会话中的对象必须可序列化的原因。）表面上，这似乎提供了一种解决方案，但结果是它使得问题更复杂了。

有一种可能的机制可以对此提供支持：在服务器集群中，指派一个单独的节点作为会话状态服务器。对每一个请求，无论哪个节点正在处理它，该节点都向会话状态服务器查询此客户

端的会话状态，然后将会话状态通过网络传给处理此请求的节点。然而，这种机制有两种副作用：(1)每个请求都增加了一次与会话状态服务器之间的往返访问，这增加了客户端请求的等待时间。但更重要的是，(2)所有的会话状态都被存储在集中的服务器上，这使得集群中产生了一个单一故障点(single point of failure)。而避免单一故障点正是我们采用集群最常见的原因，所以这很明显不是我们的理想解决方案。

另一种可能的机制是采用更 P2P（peer-to-peer）方式。当一个请求进入某节点时，此节点发出一个集群广播信号，询问其它节点是否拥有此客户端最近的会话状态。拥有此客户端最近状态的节点对此做出回答，并将该会话状态传递给当前正处理请求的节点。这避免了单一故障点问题，但由于会话状态在网络间的传递，我们仍要面对一些额外的往返访问。更糟的情况是，当某个客户端慢慢地在集群中转了一圈之后，集群中每个节点都存储了一份客户端会话状态的拷贝。这意味着，现在，整个集群能够支持的客户数就只是单个节点能够存储的最大客户数，这显然少于理想值。如果集群中的每个节点，在将客户端会话状态发送给其它节点之后就丢弃该会话状态，那么，对会话状态进行缓存，以避免在网络间往返传递会话状态的优点就丧失了。

所有这些努力带来的结果说明，要构建这样的功能可不是件简单的工作。没有几个 servlet 容器能够保证实现它。（有些 EJB 容器同时也是 servlet 容器，例如 WebLogic 和 WebSphere，它们支持分布式 Web 应用，不过那通常是建立在集群服务器对有状态会话 bean 的支持之上。不用说，集群化的有状态会话 bean 的状态也有相同的问题。）在确定某个 servlet 容器能否处理这些之前，应该仔细地询问其供应商是如何实现的，以了解其中涉及的开销。

当需要某种分布式会话状态机制，但是 servlet 容器不提供此机制，或者是提供的机制不能满足你的特定需要时，这也没什么损失。借助 Servlet 2.3 和过滤器(filter)的威力，你能够不太费力地创建自己的分布式会话机制。关键在于过滤器能够对那些在 servlet 处理管道中使用的 HttpServletRequest 和 HttpServletResponse 对象进行指代替换。

该思想很简单：创建一个过滤器，代替缺省的 HttpServletRequest 对象，覆写标准的 getSession 方法，返回一个自定义的 HttpSession 对象。逻辑上，它应该像下面这样：

```
import javax.servlet.*;
import javax.servlet.http.*;

public class DistributableSessionFilter
  implements Filter
{
  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
    throws ServletException
  {
    HttpServletRequest oldReq =
      (HttpServletRequest)request;
    HttpServletRequestWrapper newRequest =
      new HttpServletRequestWrapper(oldReq)
    {
      public HttpSession getSession(boolean create)
      {
        if (create)
          return new DistributedHttpSession(oldReq, response);
        else
        {
          // If user has created a distributed session already,
          // return it; otherwise return null
        }
      }
    };
    chain.doFilter(newRequest, response);
  }
}
```

在此处的代码中，DistributedHttpSession类实现了HttpSession接口，它的getAttribute/setAttribute（或其它）方法以传递进来的对象为参数，并将它们保存在某处“安全”地带，例如保存在RDBMS或共享的会话状态服务器中等。注意，由于你已经不再使用标准的HttpSession对象，所以将会话cookie嵌入到HTTP应答报头(response header)的过程，可以是包含在此过滤器中，也可以是包含在DistributedHttpSession中，这由你决定。通常servlet容器自己会处理这些，但由于我们不再使用它缺省的会话机制，也就不能像从前那样偷懒了。使用巨大的随机数作为会话的标识，以防止攻击者能够猜到会话的ID值。还要确保不使用标准的JSESSIONID报头，以避免与servlet容器自己

产生冲突。

此处虚构的 DistributedHttpSession 类的实际实现能够进行很宽泛地变化。一种简单的实现是将会话保存在即时复制（hot replication）被打开（以避免单一故障点的情形）的 RDBMS 中；另一种实现是使用外部的共享的会话状态服务器;第三是采取 P2P 的方法。无论采用什么实现，重要的是使用你自己的会话系统代替标准的会话系统，由你来控制系统确切的行为，使它在必要时能够在 Web 应用之间可调节。这正是第 6 项在工作中的实际应用。

与会话相关的另一个需要注意的问题是会话的意外使用。由开发者自己编写的servlet不会有此问题，因为只有在HttpServletRequest中明确地调用getSession，才能创建会话。然而，在JSP网页中就不是这样了。JSP的规范清楚地陈述道，对于给定的JSP网页，“打开”会话的指示(带有session属性的@page指示性标记)，缺省地被设置为true，这意味着下面的JSP网页将为其建立一个会话，即使该网页从未使用会话：

```
<%@ page import=”java.util.*”>

<html>
<body>
Hello, world, I’m a stateless JSP page.
It is now <%= new Date() %>.
</body>
</html>
```

更糟的是，在 Web 应用的任何角落，只要有一个这样的 JSP 页面，在该客户端使用此 Web 应用的整个过程中，该会话（即使没有在会话中保存对象，也会带来相关的系统开销。）将一直存在。所以，除非你需要使用会话，否则你应该确保你的 JSP 页面都关闭会话。我希望对于给定的 Web 应用，有某种方法能够令 session = false 成为缺省设置，然而到目前为止它并非如此。

希望你不要误解，我并不是鼓吹完全不使用会话，实际上，如果建议 HTTP 不提供任何合理的机制以提供每个用户的状态的话，将是很滑稽的。如果小心地使用，那么 HttpSession 就可以提供必要而强大的机制去提供在 Web 应用中的每个用户的状态，而这通常是关键而且必需的东西。（否则怎么才能确定用户是否成功地通过了身份认证，或者在应用中跟踪用户的进度呢？）危险在于，不是必需使用会话的时候，过度地使用或滥用此机制，那将给 servlet 容器带来额外的开销。所以，如非必要，尽量不要使用会话，如果非用不可，为了尽可能少地消耗运行 servlet 容器的机器上的资源，请保持会话精简而有意义。

第 40 项：使用对象优先的持久化来保存你的领域模型

为了设计一个优雅的对象模型来表现你的应用系统的业务领域对象，你一定花费了好几周，甚至几个月的时间，用以寻求继承、组合、聚集或其它的对象建模技巧的最佳搭配。这是项很耗时的工作，不过你最终将得到一个令整个开发团队都为之骄傲的对象模型。现在，是时候将领域数据存入数据库了，而且你也想让你的对象模型保持是适当的。说到底，如果你不得不绕回去，编写一堆丑陋的 SQL 语句将数据存入数据库，那么对象模型又有什么好的呢？

使用对象优先的持久化方式时，我们力求在持久化的过程中保持对象的视角。这意味着无需我们的任何提示，对象就知道如何默默地持久化自己，或者它们能够提供某种以对象为中心的(object-centric) API 进行持久化和读取操作。那么在理想世界中，编写像下面这样的代码将自动在数据库中创建一个包含了代表 25 岁的 Stu Halloway 的项：

```
Person p = new Person(“Stu”, “Halloway”, 25);
System.out.println(p);
// Prints “Stu Halloway, age 25”
```

而下面的代码将自动更新在第一段代码中所创建的行，将 Stu 的年龄从 25 改成 30：

```
Person p = Person.find(“Stu”, “Halloway”);
System.out.println(p);
// Prints “Stu Halloway, age 25”
p.setAge(30);
System.out.println(p);
// Prints “Stu Halloway, age 30”
```

注意到了吗，对象优先持久化方法的一个重要的优点：没有丑陋的 SQL 语句，不用为是该 INSERT 还是该 UPDATE 这样的问题而烦心。我们所能看到的只是对象，这正是我们喜欢的方式。

然而，在读取对象时，对象优先的方法往往很快就不起作用了。一般说来，对象优先的方法可能会采取以下两种方式：要么以纯面向对象的形式，通过创建包含查询规则的对象，进行对象查询；要么使用某种特定的“查询语言”进行对象查询。

在纯粹的对象优先的环境中，除了对象，我们不希望看到任何东西，所以我们创建了查询对象（Query Object）[Fowler, 316[1]]，它包含了我们所关心的约束查询的规则。遗憾的是，如果要创建一个复杂的查询，若它的查询规则不是对象的主键（有时称为对象标识符 object identifier，简称OID），从OODBMS的角度来看，这样做通常是复杂而且笨拙的：

```
QueryObject q = new QueryObject(Person.class);
q.add(Criteria.and(
        Criteria.greaterThan(“dependents”, 2)),

Criteria.lessThan(“income”, 80000)));
q.add(Criteria.and(
Criteria.greaterThan(“dependents”, 0)),
Criteria.lessThan(“income”, 60000)));
```

我们此处所做的等价于下面几行语句：

```
SELECT * FROM person p
WHERE ( (p.dependents > 2 AND p.income < 80000)
OR (p.dependents > 0 AND p.income < 60000) )
```

哪一种更易于阅读？如果我们开始在查询中执行深层嵌套的布尔逻辑，例如查询“收入少于 $80,000 而且有超过 2 个子女的人，或者收入少于$60,000 而且无子女的人”，此时事情将以指数级地恶化。事实上并不难发现，比起通用目的的查询语言，例如 SQL，纯对象优先

[1] 顺便说一句，如果你使用Fowler在其书中对查询对象的实现，那么请记住，其代码可能易受SQL注入攻击之害（请参阅第 6 项）。

的查询方法对于能查询什么，有着过于严格的限制。

这启发我们找寻第二条路，即创建某种“查询语言”，使它能够更简明地表达查询，而无需使用过于复杂的代码。Java 中所有的对象优先技术最终都回到了这一点：EJB 2.0 引入了 EJBQL，一种为实体 bean 编写查询方法（finder）的查询语言；JDO 引入了 JDOQL，它为 JDO 增强的持久类做相同的工作；而 OODBMS 回过头来采用 OQL(Object Query Language)对象查询语言。这些语言相互之间有着微妙的区别，但有着一个明确的共同点：它们都很像 SQL，而 SQL 正是我们起初试图摆脱的。（JDOQL 从技术上来说仅仅是一种用来描述过滤器的语言，尽管仍然在使用一种查询对象风格的 API，但是它实质上只是查询的谓词部分，即 WHERE 从句部分。）更糟糕的是，在 EJB 2.0 中定义的 EJBQL 缺乏很多关键功能，而正是这些关键的功能令 SQL 在执行查询时显得如此强大。2.1 版将注意到此缺陷，但 EJBQL 仍将缺少某些 SQL 的功能。

使用对象优先的方法还有另一个副作用，即不可视的往返访问。例如，当像下面这样使用实体 bean 时，引发了多少次数据库访问呢？

```
PersonHome ph =
  (PersonHome)ctx.lookup("java:comp/env/PersonHome");

// Start counting round-trips from here
//
Collection personCollection = ph.findByLastName("Halloway");
for (Iterator i = personCollection.iterator(); i.hasNext(); )
{
  Person p = (Person)i.hasNext();
  System.out.println("Found " + p.getFirstName() +
                     " " + p.getLastName());
}
```

虽然看起来似乎只访问了一次数据库（读取每个姓 Halloway 的 Person 对象，并将其组装到 PersonBean 池中的实体 bean 上），但实际上，这正是 EJB 中的 N+1 次查询问题，查找方法调用只查找符合查询条件的行的主键，然后用只知道主键的实体 bean 的存根组装 Collection 中，并在必要的时候才将数据惰性加载到（lazy-load）实体 bean 中。由于我们立刻就开始访问实体 bean 上的数据，这迫使实体 bean 从数据库中依次更新它自身，

而且由于遍历了 Collection 中的每一项，所以，在为得到主键进行了第一次数据库访问之后，还要再加上 N 次访问，N 等于集合中项的数量。

敏锐的开发者很快就会指出，特定的 EJB 实体 bean 的实现不一定非要这么做。例如，开发一个实体 bean 的实现，对于查询的结果，它不是简单地取回实体的 OID/主键，而是取得保存在实体中的整个数据集，这也是可行的（也许会导致不可移植，请参阅第 11 项）。实质上这就是采用积极加载（eager-load），而不是更为常用的惰性加载（请分别参阅第 47 项和第 46 项）。遗憾的是，这引发了一个相反的问题，现在我们抱怨的是取回的数据太多，而不是太少。

这里问题的关键是，在对象优先的持久化场景中，数据读取的原子单位是对象本身，从面向对象的观点来看，返回那些比对象小的东西根本没有意义，就像在 SQL 查询中，如果返回的是比行还小的单位，结果同样没有意义。所以，如果我们需要的是 Person 的名和姓，则必须读取整个 Person 对象才能得到它们。熟悉 OQL 的读者会站起来抗议了，他们会（正确地）指出，OQL 允许读取“部分”对象，但是这会引发进一步的问题。像这样的查询，其返回类型到底是什么呢？我可以写出某些类似于下面的语句：

```
SELECT p.FirstName, p.LastName
FROM Person p
WHERE p.LastName = ‘Halloway’;
```

可是，这到底返回了什么？通常而言，一次对象查询的返回值是一个已定义类型的对象（如上例中的 Person 实例）。可此处我们得到的是什么？对于只返回“部分”对象，并没有普遍都可接受的方式，所以典型的结果是采用像 ResultSet 或者 Java Map 一类的东西（或者是 Map 实例的一个 List）。

即使我们理清了这些问题，对象优先的查询还是有其它的问题：对象到对象的引用。在此种情况下，困难并不经常发生，因为我们还没有很好的建模技巧去管理在关系型数据库中的一对多、多对多、或多对一的关系（顺带一提，这并非微不足道）。但是问题仍然存在，当一个对象被读取的时候，问题就来了，是否应该将所有与它相关联的对象都读取出来呢。还有，我们应该如何解决这个问题：两次独立的查询通过该间接引用取回了两个相同的对象？

例如，考虑此场景，我们的系统中有四个 Person 对象：Stu Halloway 娶了 Joanna Halloway，他们有两个孩子 Hattie Halloway 和 Harper Halloway。从任何良好的对象观点来看，这意味着好的 Person 模型应该有一个配偶属性 spouse，它是 Person 类型的（或者更确切一点，是指向 Person 的引用），同样还有一个孩子属性 children，它是某种集合类型，包含指向 Person 的引用。

现在，如果我们执行前面的查询，以取得第一个对象（让我们假设是 Stu），那么通过网络取回 Stu 对象时是否也应该通过网络去取回 Joanna、Hattie 和 Harper 呢？问题又来了，我们此处是采用积极加载数据，还是惰性加载呢，记住，这些对象是被 Stu 对象实例的域所引用的。当我们从查询结果中取得下一个对象 Joanna 时，她也引用着 Stu，此时在客户端的进程空间中，我们是有一个 Stu 对象还是两个？如果我们做两次独立的查询，第一次只读取 Stu 对象，第二次读取 Joanna，会发生什么状况呢？对象同一性（identity）的概念很重要，因为在 Java 中，对象的同一性是通过 this 指针（对象的位置）来确立的，而在数据库中它是通过主键来表现的，令二者相匹配困难重重，特别是当我们将事务处理置于二者之间时。不过这也并非是不可解决的问题，同一性映射（Identity Map）[Fowler，195]就是一个典型的解决方案，但是作为一个对象程序员，你对此必须警惕，以防你采用的对象优先持久化机制没有考虑到此问题。

此处最终的结论是，如果你想采用某种对象优先的持久化方法，可不能仅仅因为“更容易使用”就选择它。在很多情况下，只使用对象，其性能与吸引力就已经足够弥补你别处的损失了，并且还具备很多优点。

第 41 项：使用关系优先的持久化来显示关系模型的威力

多年以来（如果不是几十年的话），数据存储层的国王毋庸置疑是关系型数据库。尽管 90 年代中期跑出了一个勇敢的OODBMS，关系型数据库仍然在企业级数据应用中保持着其牢不可破的统治地位，而且相关的提供商也不愿放弃其已有的地位。我们这些开发者想继续使用面向对象语言，因为我们发现面向对象技术最有助于解决我们的问题，但企业则希望延续他

们在关系型数据库上的投资。所以，很自然地，我们想把将对象数据持久化保存到关系型数据库中时所碰到的棘手细节“隐藏”起来。遗憾的是，这其中有个问题：对象和关系相处得并不好。在这两种技术之间取得良好的映射很困难，这种困难甚至有自己的名字：对象与关系的阻抗失配（impedance mismatch）。

面向对象语言与关系型数据访问技术（如JDBC）协同工作时的问题，很大一部分只是因为这两种技术的基础在看待世界时所采用的方式非常不同。面向对象语言希望使用对象，它具有属性（域）和行为（方法）。而关系技术将世界看成元组，即被群组为某种逻辑“事物”的数据项集合，Date称之为“关系”[Date]。实质上，关系模型处理的是关系的集合，我们通常称之为表；每一个关系即是一行，元组中每一个项即为一列。围绕着在这种关系格式下操作数据的思想，人们构建了一种完整的语言以提供数据访问能力。

尽管针对对象-关系映射层所固有的问题有大量的论文，且此问题也超出了本书的讨论范围，但是简要地看看其中的一个问题，将有助于我们理解为什么对象-关系映射问题在 J2EE 系统中如此普遍。请思考下面这个简单的领域对象模型：

```
public class Person
{
  private String firstName;
  private String lastName;
  private int age;

  // . . .
}

public class Employee extends Person
{
  private long employeeID;
  private float monthlySalary;
}
```

这可能是世界上最简单的领域模型了，但我们应该如何将它持久化到一个关系型数据库中呢？

一种方法是创建两个表：PERSON 和 EMPLOYEE。使用外键（foreign-key）关系将二者

的行彼此关联起来。每次我们想得到一个 Employee 时就需要对两个表做一次连接（join）操作，而每次查询和修改数据时，数据库还需做更多的工作。我们也可以将 Person 和 Employee 数据存入单一的 EMPLOYEE 表中，但如果我们又创建了 Student（继承自 Person），并想找到所有姓 Smith 的 Person 对象时，我们不得不搜索 STUDENT 和 EMPLOYEE 两个表，而二者在关系层面上并不相关。如果这种继承层次继续变得更深，则问题几乎呈指数级地混杂起来。

更何况，企业应用的开发人员通常没有对数据库模式（schema）的控制权，因为遗留系统或其它 J2EE 系统已经在使用它了，或者是由其他开发团队负责数据库模式。所以，即使我们想建立一个表的结构，使它优雅地映射到我们的对象模型，我们也不能随心所欲地改变数据库模式的定义。

从另一个完全不同的角度来看，也许有其它更为实际的原因致使我们放弃对象优先的方式。这个星球上可能还没有任何对象-关系映射产品能够处理你作为你项目的组成部分而继承来的关系型数据库模式。或者，你就是更喜欢采用关系模型和 SQL，而不喜欢对象模型（虽然这似乎暗指你的职业以前是数据库管理员，而后才变成 Java 程序员）。

由于这些（以及更多的）原因，我们通常更易于采用关系的观点来看待并操纵数据，而不是将访问关系的操作隐藏在其它某种封装技术之后，例如面向对象、面向过程、或面向层次结构。

如果要理解我对于采用关系优先方式的看法，我们需要后退一步，重新看看关系型方式到底是什么。Chris Date 与 E. F. Codd 一同被看作是关系模型之父，按 Chris Date 的说法，“关系系统建立在形式化的基础或理论之上，被称为数据的关系模型”[Date，38 强调指出]。对数学家而言，关系模型是建立在集合论和谓词逻辑的基础上。然而对我们而言，用最简单的话来说，数据的关系模型就是表而已。访问数据得到的只是表，而操作那些数据的运算符（SQL）也是由表再产生表。

也许这些讨论显得很多余，不过我们只用了三十秒来复习关系型数据库，弄清了它的核心就是表，表就是关系模型中“关系”，就是它使得关系模型如此强大。因为关系型数据访问（SQL

语句）的最终产品是表，而表又是诸如 JOIN，SELECT 等关系数据操作的源头，由此关系数据访问做到了 Date 所说的闭包性（closure）：一次访问的结果可以作为另一次访问的输入。这使得我们能够写出嵌套的表达式：表达式中的操作数由一般的表达式来代表，而不是直接使用表名。SQL 之所以那么强大，很大部分原因就是因为支持嵌套，虽然我们并不想过多地使用嵌套（这通常是因为我们倾向于采用对象优先持久化的方法，而 SQL 的嵌套表达式并不适合对象-关系映射层）。

为什么闭包性很重要？对于关系型数据库而言，SQL 是一种强大的数据访问语言，而 SQL 查询得到的结果就是表，这真是值得庆幸，因为这样我们只需要一个 API，就能取得任何一次查询的返回结果，无论结果数据是很多，还是很少。我们也没有第 40 项中“比对象小”的问题，因为取得的结果总是表，即使是只有一列的表。我们必须面对的问题是，关系模型经常不能与程序员使用的对象模型相匹配，不过关于这一点我们可说的有很多。还是让我们先来看看怎样令关系访问本身更容易吧。

在你一想到你余下的职业生涯都要跟讨厌的底层的 JDBC 访问打交道，因而在恐惧中萎缩之前，请先做个深呼吸，采用关系优先的方法并不意味着放弃任何比 JDBC 层次高的方法。实际上，远远不是这样。Java 允许我们可以采用多种更为简单的机制进行关系的存取访问，而不仅仅是原始的 JDBC（在许多情况下 JDBC 仍然是一种可选的方案，尽管它具有相对比较低层的特性）。

首先，JDBC 可不仅仅只是 Connection、Statement、和 ResultSet 对象。RowSet 和 Sun 公司独特的 CachedRowSet，通过将查询行为与获得的结果进行封装，令 JDBC 更便于使用。因此，假设你没有 JDBC DataSource，也可以很容易地像下面这样进行查询：

```
RowSet rs = new WebRowSet();
   // Or use another RowSet implementation

// Provide RowSet with enough information to obtain a
// Connection
rs.setUrl("jdbc:dburl://dbserver/PEOPLE");
rs.setUsername("user");
rs.setPassword("password");
```

```
rs.setCommand("SELECT first_name, last_name FROM person " +
              "WHERE last_name=?");
rs.setString(1, "Halloway");

rs.execute();

// rs now holds the results of the query
```

大多数对 RowSet 的调用都可以被隐藏在对象工厂接口之后（关于对象工厂的详细描述请参阅第 72 项），所以客户端代码还可以再少一些，如下所示：

```
RowSet rs = MyRowSetFactory.getRowSet();
rs.setCommand(. . .);
rs.setString(1, “Halloway”);
rs.execute();
```

这与直接使用 SQL 进行访问几乎一样简单。此处使用的工厂也并不难以想象：

```
public class MyRowSetFactory
{
  public static getRowSet()
  {
    RowSet rs = new WebRowSet(); // Or some other
                                 // implementation

    // This time, use JNDI DataSource
    // rather than url/username/password
    rs.setDataSourceName("java:comp/env/jdbc/PEOPLE_DS");

    return rs;
  }
}
```

RowSet API 提供了多个方法来控制查询执行的细节，包括 ResultSet 返回查询结果的数量（这应该被设得尽可能高，最好就是查询返回的确切行数，以避免当返回结果很多时，在网络上往返的开销），以及此次查询的事务隔离级别（请参阅第 35 项）。

然而，在每次你想使用RowSet API时，仍然需要你自己写出你感兴趣的SQL查询，每次你想取回一些数据时，就要写又丑又令人生厌的SELECT XXX FROM table WHERE baz=? 语句，特别是如果你使用PreparedStatement对象来避免堕入简单字符串连接的陷阱，

则又将自己暴露给了注入攻击（请参阅第 61 项）。因此你会尝试着努力去避免注入攻击的可能性；或者，也不是个好主意，你开始采用别的方式来获得解脱，像使用SELECT * FROM xxx取得表中的所有列，以此取代只取回你关心的数据，然而，这意味着你在浪费网络带宽。也许在开发环境中这不成问题，但是对最终的产品而言，如果 1000 个用户同时执行命令，很容易就会超出网络的最大负载[2]。如果不是你需要的数据，你就不应该通过网络请求它。那么，有自尊心的程序员会找寻什么正确的方式，以避免他们可怜的手患上腕管综合症呢？（如果你觉得这个例子没有说服力，那么我们换一种方式来讨论该问题：如果我们改变了表的定义会发生什么，是否意味着要修改散布于所有代码中的所有SQL语句？）

一种方法是保持数据库面向表的观点，然后通过表数据网关（Table Data Gateway）[Fowler，144]在你与数据访问技术之间构架更多的渠道。其实质上就是将每个表变成一个类，然后用这些类依次作为该表中任意行的访问点。

```
public class PersonGateway
{
  private PersonGateway() { /* Singleton—can't create */ }

  public static Person[] findAll()
  {
    ArrayList al = new ArrayList();

    RowSet rs = MyRowSetFactory.getRowSet();
    rs.setCommand("SELECT first_name,last_name,age "+
                  "FROM person");
    rs.execute();
    while (rs.next())
      al.add(new Person(rs.getString(1),
                        rs.getString(2),
                        rs.getInt(3));
    return (Person[])al.toArray(new Person[0]);
  }

  public static void update(Person p)
  {
    // And so on, and so on, and so on
  }
}
```

[2] 是的，它也许必须是一个宽得令人厌恶的表才会导致这个问题，不过我相信你明白我的意思。

使用这种方法时请注意，表数据网关看起来很像是采用过程优先（procedural-first）方式进行的持久化处理（请参阅第 42 项）；关键的区别在于，表数据网关是针对每个表，且只关注对表中数据的操作，而过程优先是以一种与表无关的方法（因此底层的数据存储甚至可以不是关系型的表），它为所有持久化逻辑提供了单一的访问点。

你也许会问，“这对单个表起作用，可是我要做的很多查询，可不是只针对单个表的，那怎么办呢？”感谢关系型数据访问的闭包性，事实上，表数据网关可以扩展出新的变体，叫做查询数据网关（Query Data Gateway），它不是对单个表进行包装，而是对一个查询进行包装。

```
public class ChildrenGateway
{
  private ChildrenGateway() { }
    // Singleton, can't create

  public static Person[] findKidsForPerson(Person p)
  {
    ArrayList al = new ArrayList();

    RowSet rs = MyRowSetFactory.getRowSet();
    rs.setCommand("SELECT first_name,last_name,age "+
                  "FROM person p, parent_link pp "+
                  "WHERE p.id = pp.child_id "+
                   "AND p.last_name=?");
    rs.setInt(1, p.getPersonID());
    rs.execute();
    while (rs.next())
      al.add(new Person(rs.getString(1),
                        rs.getString(2),
                        rs.getInt(3));

    return (Person[])al.toArray(new Person[0]);
  }
}
```

具有讽刺意味的是，早在我们成为 Java 程序员之前，这种方法就在关系的世界中被广泛使用了，且数据库一向对它提供支持：称为视图。数据库将查询装扮得看起来就像一个表，且其行为也像是一个表：

```
CREATE VIEW children AS
  SELECT first_name, last_name, age
  FROM person p, parent_link pp
  WHERE p.id = pp.child_id
```

这样就创建了一个被称为 children 的伪表，它包含了所有具有父母的 Person 对象。然后我们再通过对 last_name 的约束，进一步查询 children。

这种方法有什么优点？我们需要做的只是创建某种看上去像是表的东西，它不会令我们对数据库的性能感到恐慌吧？对一些老的数据库产品而言，恐怕会的，但是这个问题在很多年前就被纠正了。虽然对视图也有一些约束，例如更新视图时，一些数据库产品不允许全面更新，如果视图由多个表连接而成，则有的数据库只允许更新一个表等等，但视图仍然是一个功能强大的工具，它令查询更具可读性，某些情况下甚至更高效。

也许这些方法都不是你正在特别找寻的，因为在 Java 中使用它们似乎仍需要做很多工作，而且还是很难以使用。那么还有另一种方法，它利用近来才被标准化的 SQL/J，也称为“Java 的嵌入式 SQL”，该规范作为 SQL-99 规范的一部分才刚刚通过。同其它嵌入式 SQL 技术一样，SQL/J 允许程序员在 Java 代码中直接编写 SQL 语句，然后通过一个 SQL/J 预处理器进行预处理，转为规范的 Java 代码，使其可以通过 javac 正常的编译步骤。SQL/J 预处理器会处理 java 代码中的所有调用级接口逻辑，于是，当你想与数据库对话时你只需专心写 SQL，而当你想做别的事情时就只需关注于 Java。

使用 SQL/J 最简单的方法是在代码中直接嵌入静态的 SQL 语句，称为 SQL/J 子句，如下所示：

```
public static float averageAgeOfFamily(String lastName)
  throws Exception
{
  #sql iterator Ages (int individualAge);

  Ages rs;
  #sql rs =
    { SELECT age FROM person
      WHERE last_name = :lastName };
```

```
    int totalAge = 0;
    int numPersons = 0;
    while (rs.next())
    {
      numPersons++;
      totalAge += rs.individualAge();
    }

    return ((float)totalAge) / ((float)numPersons);
  }
```

除了#sql区块，这看起来就是标准的Java方法，#sql会引发预处理器处理代码，而#sql iterator子句引入的Ages类型则已经在方法体内了。在这里并不适合去讨论SQL/J语法的细节，可以从http://www.sqlj.org下载SQL/J的参考实现和相关文档。

这里我们需要认识到，SQL/J 实质上是从你眼前隐藏了数据访问的几乎所有步骤，只留下了最小的一个架子，令我们可以很容易地编写 SQL，它在很多方面都与对象优先技术（例如 JDO）是完全相反的，对象优先技术努力从程序员的视线中完全隐藏关系访问。到目前为止，在“正确地进行关系数据访问”的思想上，比起其它数据访问技术，SQL/J 做出的最主要的改进在于：如果数据库的模式在编译期可用，SQL/J 预处理器或翻译器就能够在编译期检查 Java 代码中的 SQL 是否符合数据库的模式，以捕捉输入错误和语法错误，这样就不必在运行期依赖 SQLException 实例来找到这些错误了。

然而 SQL/J 有一个重要的缺点，它相当缺乏 J2EE 社区的支持。虽然 J2EE 和 EJB 规范对于可能的数据访问技术层，提到了 SQL/J 的名字，但也仅此而已。Oracle 从当前版本才开始令 SQL/J 工具成为可用的，这也是唯一一个支持 SQL/J 的主流数据库供应商。从整个 Java 社区来看，它始终缺乏支持，这很令人遗憾，因为 SQL/J 在很多方面都是关系优先持久化方法的典范。

和对象优先持久化方法一样，关系优先的方法也有其长处和短处。它具有源自 SQL 的强大与优美，不过这是以（至少是在一定程度上）放弃对象模型在获取与更新数据方面的优美为代价的。这意味着必须编写代码，使用 JDBC 或 SQL/J 从关系的 API 中取出数据，再放入你的对象模型，但是你必须要能够控制所产生的确切的 SQL 语句，因为这些语句可能会被

大肆优化，而优化后的结果并非都与原来的 SQL 等价（请参阅第 50 项）。

第 42 项：使用过程优先的持久化来创建一个封装层

在面向对象接管编程语言社区之前，在 70 年代末 80 年代初数据存储的选择面临大扫荡之前，也就是关系型数据库普遍使用之前，另一种方法迅速地在 IT 界走出了自己的一条路。

为取代通过SQL直接访问数据存储层，中间件产品（TP Monitors或类似的东西）提供了著名的入口点例程（entry point routines），今天我们称之为过程（procedure）。过程可以接收输入（参数）并确保将数据存入系统，过程会返回某种结果以表示操作成功或失败。实际的存储模式和机制的细节被隐藏在过程入口点这堵不透明的墙之后，只要中间件层能够确保恰当地存储数据（并且能够取回，这一般是通过另一个过程进行的），我们就不必学习太深入的知识了。不久之后，过程被扩展用来提供某些处理，例如提供事务语义以确保整个过程的执行具有ACID性质。再后来，通过现在所谓的存储过程（stored procedure），这种处理也可以直接在数据库中使用了。

最初，我们的目标只是提供一个封装层，以避免程序员直接操作原始数据，最终却为事务处理创造了条件，更为后来的数据库管理系统打下了基础。SQL 的很多方面都是源自这种思想，SQL 作为一种声明式的语言，它只需要操作员简单地说出对什么数据感兴趣，而这做为 SQL 语句的一部份，表现为一系列的约束。这种方法虽然极为强大，但也对熟悉过程式（和后来的面向对象）语言的程序员造成了理解上的困难。

与其它声明式的语言（Prolog 和更现代的 XSLT）一样，SQL 只需要用户说明他们对什么数据感兴趣即可，而不用理会如何取得数据的方法。这与关系模型的“潮流”相适应，但是过程型程序员在第一次学习 SQL 时则会感到很笨拙，他们习惯于“从这个表的这里开始，然后取得那个数据元素，根据……”等等。声明式与过程式语言是非常不同的两种事物（正如对象与关系那样）。关系专家 Joe Celko 曾对此说过“学习 SQL 编程最大的挑战就是忘却过程式编程” [Henderson03，1]。

与随意地采用某种模型相比，更应该考查在不同情况下什么模型工作得最好，它应该能够直接将持久化的细节掩藏在软件的另一层之后，如此，即可将持久化操作无关紧要的细节从程序员的视线中隐藏起来，达到真正的封装。要做到这一点，可以像下面的代码这么简单：

```
public class DataManager
{
  private DataManager()
  { /* Prevent accidental creation */ }

  public Person findPersonByLastName(String lastName)
  {
    // Open a Connection, create a Statement, execute
    // SELECT * FROM person p WHERE
    // p.last_name = (lastName)
    // Extract data, put into new Person instance,
    // return that
  }

  public void updatePerson(Person p)
  {
    // Open a Connection, create a Statement, execute
    // UPDATE person p SET . . .
  }
}
```

注意，DataManager 是一个单件（Singleton），运行在客户端本地的虚拟机上，为避免成为远程单件（Remote Singleton）（请参阅第 5 项），它依赖于数据访问层（在这里是 JDBC）来防止并发问题。

虽然这会令 DataManager 成为一个相当庞大的类，但也有许多优点令它成为颇具吸引力的选择。

- 数据访问专家可以调整数据访问的逻辑（我们假设就是SQL），以获取最佳结果。
- 如果必须改变数据存储系统，也只需修改这一个类。如果我们想将关系型的后端换成 OODBMS，也无需修改客户端代码来执行持久化的工作（这不是必然的，取决于所使用的OODBMS和它底层的持久化语义）。

- 既然我们知道该请求的所有持久化逻辑都在方法体内由括号括起来了，所以客户无需担心事务语义或事务是否打开太久了，这对于最小化锁窗口很重要（请参阅第 29 项）。
- 我们可以就在这一个类中使用任何数据访问的API，令实现的方法获得最大利益，而不需要修改任何客户代码。如果我们先是使用JDBC，然后感到JDBC太底层而不便于使用（这并非是不常见的观点），那可以只在这一个类中转为使用SQL/J，而客户则可以完全忽略此变化。如果我们发现SQL/J虽然容易使用，却不能生成优化的JDBC访问，那我们还可以换回来。
- 为保持后端的可移植性，可以通过以DataManager为接口，创建DataManager的派生物，令它专门针对一个或多个不同的后端存储层做存储工作，比如一个OracleDataManager，一个HSQLDataManager（针对进程内的关系型数据存储），一个泛化的JDBCDataManger（针对泛化的未优化的JDBC-SQL访问）等等。这使得我们可以针对特定的后端进行调整，而无需牺牲整个系统的可移植性，代价是你需要为每一种选择都做很多工作，且当DataManager接口发生变化时还需要你来进行维护。

- 新的访问需要修改此中心类。这初看起来似乎并不好，毕竟谁会愿意修改已经编好的代码呢，但是如果有人的确需要一个新的查询或更新，那么此方法允许DataManager开发人员去修改数据访问的代码或实际存储的数据。我们能够使用这种方法，基于新的用途进行一些优化，例如在关系模型中创建新的索引。也可以作为合法性检查以防止使用无效规则查询数据（“嗯，既然大家都知道，只有男人才能成为程序员，所以我打算在查询程序员时加上一个约束规则，性别为男（gender=male），这应该能够加速查询，对吧？”）。

DataManager 类本质上是我们唯一的一个耦合点，将数据存储层与访问数据层的 Java 代码耦合了起来。

然而，过程式访问并不总是在 Java 语言级别上实现的。大多数商业化的数据库本身就提供了直接执行过程式代码的能力，在 DBMS 术语表中称为存储过程。虽然所有主流数据库供应商都有各自特殊的语言用来编写存储过程，但是其中一些也开始支持编写 Java 存储过程的想法（Java stored procedures，其缩写不幸与 JSP 重复），这意味着 Java 程序员不需要再学另一种语言了。然而，使用存储过程作为持久化模型最大的优点是，我们能够将存

储过程的实现交给负责数据库的同事，将整个数据模型都交给他们设计、实现，以及在必要时进行调整。只要存储过程的定义不变(否则我们就会得到运行期的SQLExecption异常)，以及它的实现确实如我们所希望的（存储或取回数据），我们就可以完全忽略底层的数据模型。如果你在关系的设计与执行方面与我一样“优秀”的话（即根本不擅长），这确实是件好事。

顺便说一句,如果你认为过程模型是个很糟糕的点子,而且你也绝不会考虑使用类似的东西，那么暂且先停下来，看一看会话外观[Alur/Crupi/Malks，341]、领域存储[Alur/Crupi/Malks，516]、以及其它的对象访问模式。为避免对数据库的往返访问，请记住，当前 EJB“最佳实践”的思想是采用客户端调用会话 bean，将数据传送对象[Fowler，401]或其它整批的数据（参见第 23 条）作为参数传给会话 bean 的方法，而为了存储到数据库中，再把它们提取出来并插入到本地实体 bean。这与前面提到的DataManager 没什么不同，只不过现在将 DataManager 换成了会话 bean。而这对于基于数据访问模型的 Web 服务来说也同样成立。

第 43 项：识别对象-层次结构的阻抗失配(impedance mismatch)

XML 无所不在，包括在你的持久化方案中。

一旦我们最终认识到 XML 的一切都是关于数据的，而不是一种像 HTML 那样进行标记的语言时，工业界的专家和作者们很自然地开始讨论用 XML 将对象以数据的形式表示出来。不久之后，引入了使用 XML 来编组在网路上传输的数据的想法，随后 SOAP 和与之相随的 Web 服务规范就诞生了。

问题在于 XML 在表示数据时使用的是固有的层次结构的方式，看看 XML Infoset 规范，它要求数据是格式良好的（well-formed），即 XML 文档中的元素必须形成一棵良好的元素树（每个元素都可以有子元素嵌套其中，每个元素都有一个唯一的父结点，唯一的例外是“根”结点，它囊括了整个文档）。这意味着 XML 善于表现层级结构的数据（所以作为这一项的题目），假设你的对象形成一个整洁的层次结构，则 XML 是表达该对象最自然的方式（因

此自然地假设了 XML 和对象是手牵着手）。

但是当对象没办法形成整洁、自然的树时会发生什么呢？

层次结构的数据模型并不是新事物，实际上它们相当有年头了。那时，关系数据模型正在试图找寻一种比那时候的数据库系统更容易使用的东西，而那时候的数据库系统与今天我们在 XML 中看到的层次模型，虽然形式不同，但概念相似。层次模型带来的问题是，在其中查找数据很困难。用户必须亲自在树的元素中搜寻，这迫使用户必须知道“如何查询”以获取数据，而不是仅仅关注“查询什么”感兴趣的数据即可。

随着 XML 的出现（以及不断增长的对“XML 数据库”的兴趣，尽管这个词具有二义性），似乎层次数据模型也再次流行起来。虽然对层次数据模型的全面讨论超出了本书的范围，但在此仍有必要弄清两件事：我们何时会在 J2EE 中使用层次数据模型，而它对 Java 程序员又有什么影响。

当前工业界还没有认识到，将对象映射到XML（今天最常见的层次结构存储模型）并不是件简单的事情，这使得我们想知道是否存在对象-层次结构的阻抗失配（impedance mismatch），换言之，形式自由的对象模型与XML Infoset严格的层次结构模型是否难以匹配[3]。实际上，即便考虑到现在已经有供应商为我们提供类库进行对象到XML的映射，以及最近的Java的XML绑定API（JAXB）标准能够帮助我们统一各种不同的映射实现，我们也应该公平的讲，将对象映射到XML或反向的映射并不像看到的那样简单。我们承认，简单的对象模型映射到XML相当容易，但是简单的对象模型映射到关系的表上也是相当容易的，然而我们都知道真正的对象-关系映射是如何的“简单”。

将对象映射到层次模型的问题，很大程度上与发生在将对象映射到关系模型时发生的问题一样：保证对象的同一性。为了说清楚我的意思，让我们重新看看前面曾经用过的 Person 对象：

```
public class Person
```

[3] 更不要考虑存储在关系型数据库中的对象转换成XML蕴含的意义了：对象-关系-层次结构阻抗失配的概念足以令最坚强的程序员落泪。

```
{
  // Fields public just for simplicity
  //
  public String firstName;
  public String lastName;
  public int age;

  public Person(String fn, String ln, int a)
  { firstName = fn; lastName = ln; age = a; }
}
```

同样地简单而直接，也不难想象该对象的 XML 表示会是什么样子：

```
<person>
  <firstName>Ron</firstName>
  <lastName>Reynolds</lastName>
  <age>30</age>
</person>
```

到目前为止一切都好。但是，现在让我们来添加一些东西，它们对面向对象模型而言绝对合理，却将层次结构完全打破了，这就是循环引用：

```
public class Person
{
  public String firstName;
  public String lastName;
  public int age;
  public Person spouse;

  public Person(String fn, String ln, int a)
  { firstName = fn; lastName = ln; age = a; }
}
```

你应该如何表现下面的对象集？

```
Person ron = new Person("Ron", "Reynolds", 31);
Person lisa = new Person("Lisa", "Reynolds", 25);
ron.spouse = lisa;
lisa.spouse = ron;
```

下面的将 ron 序列化输出到 XML 的方法并非不合理，它简单地遍历属性成员，递归处理每个的对象并依次遍历其属性成员，以此类推。然而这样做很快就会出问题，如下所示：

```
<person>
  <firstName>Ron</firstName>
  <lastName>Reynolds</lastName>
  <age>31</age>
  <spouse>
    <person>
      <firstName>Lisa</firstName>
      <lastName>Reynolds</lastName>
      <age>25</age>
      <spouse>
        <person>
          <firstName>Ron</firstName>
          <lastName>Reynolds</lastName>
          <age>31</age>
          <spouse>
            <!-- Uh, oh . . . -->
```

正如你看到的，因为两个对象循环引用对方，此处产生了无限递归。我们可以使用与Java 对象序列化所采用的相同的方法来处理这个问题（请参阅第 71 项），即对哪些项被序列化了而哪些项没有被序列化都保持跟踪，但是这样的话我们就陷入了更大的问题：即使在给定的XML层次结构中我们保持对对象同一性的跟踪，那么我们怎么在多个层次结构之间作到这一点呢？也就是说，如果我们在两个独立的流中序列化ron和lisa两个对象（可能是作为一个JAX-RPC方法调用的一部分），我们怎样令反序列化逻辑了解到这个事实：ron的spouse域所引用的数据与lisa的spouse域所引用的数据相同（原文即如此“the data referred to in the spouse field of ron is the same data referred to in the spouse field of lisa”）。

```
String param1 = ron.toXML(); // Serialize to XML
String param2 = lisa.toXML(); // Serialize to XML
sendXMLMessage("<parameters>" + param1 + param2 +
               "</parameters>");

/* Produces:
param1 =
<person >
  <firstName>Ron</firstName>
  <lastName>Reynolds</lastName>
  <age>31</age>
  <spouse>
```

```
    <person >
      <firstName>Lisa</firstName>
      <lastName>Reynolds</lastName>
      <age>25</age>
      <spouse><person href="id1" /></spouse>
    </person>
  </spouse>
</person>

param2 =
<person >
  <firstName>Lisa</firstName>
  <lastName>Reynolds</lastName>
  <age>25</age>
  <spouse>
    <person >
      <firstName>Ron</firstName>
      <lastName>Reynolds</lastName>
      <age>25</age>
      <spouse><person href="id1" /></spouse>
    </person>
  </spouse>
</person>
*/

// . . . On recipient's side, how will we get
// the spouses correct again?
```

（顺便说一句，使用 id 和 href 跟踪对象同一性的技巧不是新生事物。它在 SOAP 1.1 规范的第 5 节中被正式地描述，因此它通常被称为 SOAP 第 5 节编码，或简称为 SOAP 编码。）在每个独立的流中，我们尽量直接保存对象的引用，但是当我们将所有的流汇入一个大型文档后，这两个流到对方都毫无意识，整个对象-同一性模式就失效了。我们怎样才能解决这个问题呢？

简短而粗暴的回答是：不可能，如果不依赖 XML Infoset 规范以外的机制的话，这意味着 schema 和 DTD 合法性验证不会挑拣出任何畸形数据。实际上，通过 SOAP 第 5 节编码保证对象同一性的思想完全超出了 schema 和 DTD 检验器的能力，而且已经从最新的 SOAP 规范（1.2）中删掉了。循环引用在对象系统中实际上比你想象得要常见的多，它总是会破坏层次结构的数据格式，无一例外。

有些人会指出，我们可以通过向“捕获”两个独立对象的流中引入新的结构来解决这个问题，就像下面这样：

```
<marriage>
  <person>
    <!-- Ron goes here -->
  </person>
  <person>
    <!-- Lisa goes here -->
  </person>
</marriage>
```

但这就跑题了，这样做实质上是引入了新的数据元素，而元素本该是从对象模型中创建出来，但该元素在对象模型中并不存在。一个自动的对象-XML 序列化工具不应该能够做出这样的决定，而且如果没有开发者的某种协助的话，是肯定不能的。

那又怎么样呢？我们使用 XML 并不是为了数据存储，一般说来，我们使用关系型数据库来解决数据存储问题，而且对象-关系映射层会为我们处理所有的细节。为什么要一直在对象-层次映射这条路上充满烦恼地走下去呢？

如果你打算开发 Web 服务，你就会遇到对象-层次映射：请记住，SOAP 第 5 节编码正是为解决此问题而创造的，因为我们希望默默地而且透明地将对象转换成 XML，并反向为之，而且不需要我们作任何工作。然而，悲哀的事实是，如同对象-关系层永远也不可能默默地完全处理对象到关系的映射一样，对象-层次层，例如 JAXB 或 Exolab 的 Castor，也永远不可能完全处理对象到层次的映射。

而且不要认为这些限制只是单方面的。即使具有 schema 有效的 XML 文档， XML 到对象的映射也很困难，这与对象图到 XML 的映射同样困难。考虑下面的 schema：

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    xmlns:tns='http://example.org/product'
    targetNamespace='http://example.org/product' >
  <xsd:complexType name='Product' >
    <xsd:sequence>
```

```
        <xsd:choice>
          <xsd:element name='produce'
                       type='xsd:string'/>
          <xsd:element name='meat' type='xsd:string' />
        </xsd:choice>
        <xsd:sequence minOccurs='1'
                      maxOccurs='unbounded'>
          <xsd:element name='state'
                       type='xsd:string' />
          <xsd:element name='taxable'
                       type='xsd:boolean'/>
        </xsd:sequence>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:element name='Product' type='tns:Product' />
  </xsd:schema>
```

这有一个 schema 有效的相关文档：

```
  <groceryStore xmlns:p='http://example.org/product'>
    <p:Product>
      <produce>Lettuce</produce>
      <state>CA</state>
      <taxable>true</taxable>
      <state>MA</state>
      <taxable>true</taxable>
      <state>CO</state>
      <taxable>false</taxable>
    </p:Product>
    <p:Product>
      <meat>Prime rib</meat>
      <state>CA</state>
      <taxable>false</taxable>
      <state>MA</state>
      <taxable>true</taxable>
      <state>CO</state>
      <taxable>false</taxable>
    </p:Product>
  </groceryStore>
```

请问你自己一个问题：究竟 Java（或者任何传统的面向对象语言，例如 C++或 C#）是怎样表达这种成对的 state/taxable 元素似的重复序列的，或者是有区别的两种不同元素类型的 union，例如 produce 或 meat？最近似的答案可能是为 produce 和 meat 元素

粒子创建两个子类型，然后为 state/taxable 创建另一个新类型，将它们存储在 Product 类型的数组中。Schema 只定义了一个类型，而我们却定义了至少四个 Java 类型。

更不要说这个 schema 到 Java 的类型转换系统使用起来将有多么的困难。而且，如果我们开始讨论通过限制、发生约束（schema 的排版器中的 minOccurs 和 maxOccurs 刻画面）、以及其它条件进行推导的话，事情会变得更有趣。JAXB 和其它 Java-to-XML 工具能够尽量做到最好，但是它们永远不能与 schema 声明一对一地相匹配，正如 schema 与 XML 无法一对一地匹配对象一样。简言之，我们确实遇到了阻抗失配（impedance mismatch）。这给我们带来了什么呢？

对于初学者而言，能够认识到 XML 善于层次结构的数据建模，但是它无法有效处理任意的对象图。在一定的条件下，当对象模型呈整洁的层次结构时，两者之间可以平滑无缝地过渡，但问题的严重性在于，只要有一个引用不是指向其直接的子对象，它就会超出对象到 XML 的序列化工具的能力范围。幸运的是，字符串、日期和包装类通常是以相当透明的方式被处理的，无论其形式的对象状态如何，所以此时没有什么问题。但是对于其它的东西，你要就为面对某些从 schema 到 Java 的代码生成器所产生的怪异而令人困惑的结果做好准备了。

其次，让我们以更现实的观点来看看 XML 能为你做什么。XML 的普遍存在令它成为了颇为吸引人的数据存储格式，但实际上关系型数据库仍占据着统治地位，而且在可预见的未来我们一般还只是作为互操作技术而使用 XML。特别是随着越来越多的 RDBMS 供应商将 XML 作为一种描述数据的格式，那种在“XML 数据库”中将数据存储为 XML 的可能性也就更低了。相反地，我们是将 XML 视为一种在 Java 与其它类型系统，例如与.NET 或 C++之间的“数据粘合剂”。

在此给出了我脑海中的一些基本原则，但是我要提醒您，与任何好的原则一样，如果使用的环境发生变化，这其中的一些原则可能也就消亡了。

- *使用XML Schema定义你的数据类型。*如果没有在数据模型上定义关系约束，你不会实际地考虑开发一个企业项目，其数据被存储在关系型数据库中，与此类似，如果没有定义XML数据类型，你也不会实际地考虑开发一个企业项目，其数据被存储在XML中。虽说

如此，然而有时候更灵活的XML模型会更有用，例如允许用户扩展XML数据实例。通过明确使用any类型元素，去编写你的类型，以获得在类型声明中的扩展点，从而为这种扩展做好准备。虽然在企业级应用中这应该少出现，但在某些情况下，为了使有些XML数据更有用处，我们需要完全自由的形式，例如Ant脚本。在那种情况下，需要很有经验才能认识到，你不可能（或不希望）有已定义的schema类型，并且它们需要手动的验证与解析。

- *使用schema简单类型的子集。*XML Schema提供了丰富的简单类型（非常接近Java中建模的原始类型）集合，例如为时间提供了yearMonth和MonthDay类型，可是在Java中没有对应的等价物，一个从schema到Java的转换几乎两边都要建模，即使是简单的整数域。遗憾的是，这就意味着你可以向那个域中存储任何你希望的东西，因此也就丢失了schema文档最初进行类型定义的作用。为避免这种情况，最好只使用XSD schema中的类型，它紧紧围绕着Java（.NET以及其它你最终采用的语言）容易处理的类型进行建模。
- *使用XML Schema验证解析器去验证你要解析的schema类型的实例。*该解析器将标记schema不合法的对象，实质上它的作用就象是某种数据净化和输入检验层，而不需要你做任何工作。这将有助于你采用面向文档的观点对待你的XML类型，因为一旦偏离该模型，就会被验证器所标记。需要知道的是，带schema验证的解析器比不带schema验证的要慢许多。但更重要的是，带schema验证的解析器将只能够标记schema不合法的对象，如果你采用基于对象的方法去处理使用了带外（out-of-band）技术的XML类型（例如像SOAP编码所做的那样），schema验证器并不能识别出来，所以只有在某个对象通过了解析器，而在你的代码中出了问题时你才知道发生了问题。这种情况对于测试而言就难以处理了。
- *理解类型的定义是相对的。*你对于Person是什么的观念与我对Person的看法是不同的，而我们各自对于Person的定义（无论是对象类型的定义，XML Schema，或关系模式）因此也不相同。也许有人想找到关于Person定义的普遍真理，然而事实是根本不存在这样的真理。关于Person，对于你的组织而言重要的东西，与我的组织认为重要的东西是不同的，而且没有任何劝说能够令你为我改变想法，反之亦然。为避免继续在这条死胡同上走下去，我们且各自保留自己的意见，如果我们都要使用该schema描述的文档，那么就应该各自去建立相应的schema。换而言之，用schema去验证从一个系统传递给另一个系统的数据，而不是用它来定义每个人都接受的类型。

- *避免HOLDS-A关系。*在Person的例子中，持有指向其它实例的“引用”的实例导致了问题。应尽量避免它们。遗憾的是，这说起来容易做起来难。如果既要Person指明其配偶，又要保证对象的同一性，那还真是没有办法以面向文档的方式建立Person模型。反之，你必须认识到Person的配偶是其“拥有的”数据项，所以不是将其作为独立的Person进行建模，而只需获得足够的数据以能够独一无二地从其它文档中找出那个Person即可（就像关系表使用外键指向另一个表中的主键）。然而，又很遗憾，XML Schema不能反映出这些信息[4]，只能通过某种带外（out-of-band）机制来获得这些信息。对Schema的约束无法跨越文档的界限，至少当前版本不行。

最重要的是，确定你了解层次结构的数据模型，并且知道它与关系和对象模型的区别。将XML 作为一种对象优先的数据池使用是自找麻烦，所以别走向那条路。

第 44 项：使用进程内（in-process）或本地存储以避开网络

大多数时候，当 J2EE 开发人员开始设计或划分项目的架构层次时，数据存储问题已经形成结论了：将采用关系型数据库，运行在数据中心或操作中心某处的机器上。

为什么？

并没有什么神秘或神奇的答案，仅仅只是为了可访问性。我们希望该系统的任何潜在客户都能够访问到数据。

过去，在 n 层架构流行之前，客户端直接连接到服务器上，为了让所有客户端都能够访问到全部数据，数据需要直接存储在服务器端。那时网络还不普及，使网络更简单的无线访问技术还没出现。将机器连接到网络上是日常主要的杂务，结果是，P2P 通讯的基础概念总是在讨论最底层的网络堆栈问题（例如 IP 协议）。

[4] 如果该XML文档是Person实例的集合，而且保证配偶也在此集合内，那故事就不同了，但是也就改变了我们此处的模型，变成了完全不同的另一个问题。

我们渐渐认识到，将所有数据存放在一台中心服务器上有许多优点，即主要的负荷都置于该服务器端，而不是客户端。因为服务器是单独的机器，（所以我们相信）升级或替换它变得很划算，特别是比起替换所有连接到它的客户端来，更是如此。所以不久之后我们就建立起了数据库中心化的思想，我们开始将数据库放在能找到的最重的铁盒子中，用容量最大的 RAM 和巨大的千兆字节（后来是万亿字节）的驱动器装满其中。

这些关于历史的题外话的重点是，中心化的远程数据库服务器的存在只是为了提供单一的数据聚合点，而不是因为数据库“必须”运行在服务器上，那可是花了 5、6、或 7 位数的开销的。我们将数据存放到服务器上，因为(a)它是个很方便的地方；(b)易于将所有为客户端进行的处理放在一个集中的地方，且无需向客户端推出更新（零部署）；(c)这是一种令数据与处理靠近的方法（请参阅第 4 项）。

通过线路传送所有数据并不便宜，而且也有其内在固有的问题。在可扩展性与执行性能两方面都有所牺牲（因为每有一个 byte 的带宽用于在网络上传送数据时，就有一个 byte 的带宽不能用于其它目的），而重组来回传递的数据所消耗的时间也并非微不足道，正如第 17 项所描述的那样。考虑到我们将数据放在中心的数据库中是为了使其它客户也可以使用它们，而这样的数据传送开销又不低，所以除非必须，否则不要将数据放在远程数据库中。也就是说，不要将任何数据放到远程数据库中，除非确实需要与其它客户端共享它们。

在这种情况下，在与客户端应用相同的进程中运行关系型数据库（或其它数据存储技术，这里我们可以考虑使用对象数据库，甚至选择 XML 数据库），不仅可以令网络往返访问回合最少，而且可以将数据完全保存在本地机器上。虽然在我们的 servlet 容器内运行 Oracle 恐怕一段时间内还不可行，但是运行一个全 Java 的 RDBMS 实现却并非遥不可及。例如，Cloudscape 就提供了这项功能，以及 PointBase 和 HSQLDB（在 Sourceforge 上的一个开放源代码的实现），实质上是通过伪装成 JDBC 驱动程序而成为数据库的。或者，如果你喜欢基于对象的方法，另一个开源项目 Prevayler，它以传统的对象优先持久化方式存储任意的 Java 对象。如果你希望看到层次结构形式的数据，Xindice 是 Apache Group 下面的一个开源的 XML 数据库。

RowSet本身就是一个简单的进程内数据存储技术。因为RowSet是完全与数据库断开连接的，我们可以用查询的结果创建一个RowSet，并在客户进程的整个生命期内都保持着此RowSet，而无需担心对数据库可扩展性的影响。因为RowSet是可序列化的，所以我们可以将它存进任何OutputStream而不会对其造成改动，例如一个文件或一个Preferences节点。事实上，如果RowSet是围绕着配置数据而被包装的，那么与以某些方式将其存储在本地文件中相比（请参阅第 13 项），将它保存在Preferences节点中更有意义。它不会扩展到数千行，也不强制要求关系的完整性，但是如果你需要存储很多数据，或是将关系约束施加于本地数据，那么你就应该使用“真正的”数据库，例如HSQLDB或支持“嵌入”到Java进程中的商业产品。

然而这个故事还有另一面，一个显而易见的事实，远程数据库需要能够触及它的网络。尽管这似乎没有必要说出来，但是还是先思考一下这个问题，然后我们来看看大型制造业公司中普遍存在的连接到清单数据库的销售-订单应用。我们将此应用安装在销售员的便携式电脑上，然后此销售员去拜访一个客户公司的副总裁(VP)。在饮过红酒，吃过午餐，以及打完漂亮的私人高尔夫球场的第 18 洞之后，副总裁终于准备签署这个百万美元的订单了。销售员启动便携电脑，开始签单，令他恐惧的是发生了错误：“database not found.（数据库未找到）”坐在高尔夫俱乐部豪华的餐厅，我们勇敢的销售员忽然面色苍白，转向副总裁，说到：“嗯，呃，我们可以回您的办公室吗，让我借用一下您们的网络？”

假设副总裁确实让我们无畏的销售员使用了他的网络，通过 VPN 连接到销售员屋里的网络；假设销售员知道如何在他的便携电脑中设置 VPN；假设本公司的 IT 员工打开了公司网络的 VPN；无论 VPN 连接的速度怎样，假设应用确实在工作，此销售员的信誉仍然受到了沉重的打击。而在这之前，该副总裁有各种理由可以拒绝销售员使用其网络，毕竟，让外人的机器进入防火墙之后的网络是有安全风险的。而本公司的 IT 员工也有各种理由将数据库尽可能地与“外面的世界”隔离开，无论是 VPN 或不是 VPN。当销售员回到公司开始保存订单的时候（潦草地记录在豪华餐厅的餐巾上），对方的副总裁可能已经改变了主意，或者销售员忘记在餐巾上记下某些重要的细节了等等。（所以销售员被训练为一旦顾客同意就要尽量快地下单，这是有道理的。）

在某些圈子里，这称为旅行推销员问题（traveling salesman problem）（不要与最便宜的旅行路途 cheapest-way-to-travel 问题相混淆了，那通常是在人工智能的书中讨论的问题）。此问题的核心很简单：你不能假设网络总是可用的。很多情况下，你想设计一个应用，并确保它可以在无法访问网络的时候以非连接模式运行。这与当路由器或 hub 不通的时候，提供格式良好，易于处理的 SQLException 是不同的。这是将应用设计为在没有网络的地方，也可以使用独立的便携电脑处理销售员的订单。当为了这个目标而进行设计时，问问你自己，如果用户在 37,000 英尺的空中，他应该怎样使用该应用把一些小工具卖给飞机上刚认识的某个人。

要适应此场景，最简单的方法是运行一个本地的数据库，将中心数据库完整的数据转储在客户端机器上（顺便说一句，这可能是一个肥客户端应用，因为没有连接到 HTTP 服务器的网络连接，请参阅第 51 项）。当机器通过网络连接到远程数据库时，就通过中心数据库使用最新最完整的数据更新本地机器。你要知道，虽然没有必要是完整的数据库模式，但仍要保证在没有远程数据库的情况下有足够的数据保证运作。例如，在假想的销售应用中，只需要订单的清单、详细信息、和价格可能就足够了，而完整的销售、历史、以及运送信息的数据副本可能就不需要在本地存储。然后，当下了新订单之后，应用可以更新运行在同一台机器上的本地表。

有些人无疑会很憎恶整个建议。“不连接到远程数据库？不可想象！我们如何避免数据完整性错误？那正是起初我们将数据库中心化的原因！说到底，如果只剩下 100 个小工具，而 Susan 和 Steve 将这最后 100 个工具都通过他们的便携系统卖给了不同的客户，那么当我们将他们俩的数据与中心数据库同步的时候，就肯定会遇到并发问题。这是每个系统架构师都知道的！”

先做个深呼吸。然后问问自己，你会如何处理这个场景，因为无论它是发生在销售员下订单的时候，还是在订单进入数据库的时候，同样的问题总会发生。可以看看第 33 项，那提供了一种解决办法。

一般而言，在连接到服务器数据库的系统中，库存清单的检查发生在销售员下订单的时候，如果对小工具的需求数量库存不足，我们就需要使用某种红色标记去标注此订单，要么就不

签此订单，要么必须通知销售员，令他知晓此刻库存不足。在此应用可连接的版本中，这个红色标记一般是指消息对话框或者警告窗口，这与稍候再作有何不同呢，即销售员与中心数据库同步数据的时候才通知他？事实上，后一种方式可能会更好，因为警告窗口很可能立刻就毁了这场生意。如果对方的副总裁看到了这条消息，她可能会重新考虑订单的事情：“嗯，我打赌你的竞争者有现货，而我们立刻就需要这些工具，”即使“立刻”指的是“我一周只需 50 个。”于此相反，如果红色标记回到办公室才出现的话，在打电话给客户告诉他新的消息之前，销售员完全可以做些研究（如果要想坐在对方副总裁的办公室里做这些研究，那么即便不是不可能，也将会很困难）。（“好的，现在我们的仓库里只有 50 个工具，但是 Dayton 那里还有 50 个，而我会让他们快递给你，不另收费。”）

远程数据库对于企业系统有明显的优势，毕竟，我们对企业系统的定义，部分是因为它管理着对必须共享的资源的访问，所以才推出了中心化，但是对于某些类型的数据或特定的应用而言，远程数据库不是必然的最佳仓库（例如每个用户的设置、配置选择、或非共享数据），特别是那些需要在无网络连接方式下运行的应用。

第 45 项：不要假设拥有数据或数据库

请记住，（从第一章起）关于企业级软件我们有一个基本的假设，即企业级软件系统的部分或全部资源都是共享的。事实上，企业系统的核心—数据，是最频繁共享的资源。大多数时候，我们假设共享的范围是发生在企业系统各种不同的用户中。然而，它不止如此。许多企业数据库将其它企业系统也视作其用户的一部分，而且很遗憾，那些“用户”比普通用户有更加严格的需求。

将使用你的数据库的其它系统，有些并不是陌生者；这并非不常见，例如，设计或购买一个报表引擎，以使终端用户能够由数据库创建自己的报表。这是件好事，如果没有它的话，创建逻辑合理的且格式漂亮的输出的任务就落在开发人员的身上了。如果需要不断改变报表以跟上用户突然产生的念头，则你的余生可以一直在这个位置上作下去了，这也意味着你永远也不会去做项目中的其它东西了，而这可不是我所理解的有趣的工作。这些报表引擎通常是直接基于数据库模式而运作的，而这也是我们需要当心的。

如果多系统或代理共享一个资源，则这些系统或代理中的任何一个都不可能随心所欲地改变该资源的格式或模式，而不会引起其它系统的连锁反应。对模式的改动、过度的数据库锁操作，以及数据库实例的重定位，这些改变似乎可以只局限在你的代码中，但是必将引发其它系统的问题。事实上，正是这个原因令遗留系统一直存活着。对给定公司的资源或仓库（特别是大公司），追踪其每个客户端极为困难，远比弄清一个单一系统的一个给定用户的所有数据依赖关系要困难。因此，与其使用某种新系统来替换遗留系统，还不如为其创建适配器、翻译器和代理，然后将遗留系统保持在原处显得简单。

对 J2EE 开发者而言，它暗示的东西即使没有令你心寒也足够让你清醒的：你不拥有你的数据库，也不拥有其中的数据。即使在项目中你完全是从头开始建造数据库，即使该项目似乎只是塞在整个业务的一个小角落里，没有人对它有兴趣，即便如此你也不拥有此数据库。只是时间问题，总会有公司中的某个人听到了关于你的小系统，并希望获得对那些数据的访问，所以你开始设置后台数据源，直接连接等等。或者更糟的，你团队中的某人为他们建立了连接而没有告诉你。不久，你改动了某个 schema，然后你从没听说过也没见过的人，就咆哮着给你打电话，要求知道你为什么破坏了他们的应用。

现在结论很清楚了：一旦完成了你的数据库模式的设计，与你的代码相比，它就处于了危险得多的被锁定状态之中。这也是为什么用户与数据库（请参阅第 42 项中的讨论）之间的封装层变得如此重要的原因，它强制客户（任何客户，包括你自己的代码）必须通过某种封装的障碍才能获得对数据的访问，例如存储过程层或 Web 服务端点层(endpoint layer)，实质上，你是在为未来以及如下事实作计划：当你修改底层的数据库模式以及数据描述的定义时，其他人仍将需要访问这些数据。

这也意味着数据库模式不应该为 J2EE 应用进行优化：建立自己的模式时要当心，别将它与你的对象模型绑得太紧了，以二进制大型对象 BLOB（Binary Large Object）存储序列化的 Java 对象时更要极为当心，因为那样做意味着此数据不能在 SQL 查询中使用（除非你的数据库支持，在那种情况下的查询将执行得非常慢），而且不要在数据库中存储“魔幻值”。例如，虽然将 Java Date 对象存储为长整型（java.util.Date 中持有的底层的值，可以通过 getTime 得到，这是构造 java.util.Date 的方式中唯一未受抨击的）似乎很

方便，其它语言拿到那样的值却不知道该如何处理，而将它转成可接受的日期值对它们而言也很困难。举个例子，也许有些数据库试图隐式地将整型值转成日期，但是它们转换这种值的方式又将与别的数据库完全不同。尽管 Java 在这方面贡献良多，最好仍是将日期存储为基于字符串的格式，或者标准 ANSI 的 DATETIME。

事实上，因为你的数据库可能很快就会变成一个共享的数据库，并且因为 J2EE 并不是肯定能统治整个世界（.NET 和 PHP 程序员对认为它能够做到的人可能有话要说），所以你最好尽量将约束施加在数据库上，而不要依赖于 J2EE 代码来施加限制。举个例子，对于一个要存储的字符串，在存进数据库之前检查它的长度是否少于 40 个字符，可以直接依赖于数据库对这张表的完整性约束来检查字符串的长度，而不是在 J2EE 中作检查，并把列的长度设得比它大。既然数据库总会进行尺寸约束的检查，为什么要做两次呢？同样地，通过使用外键约束来确保约束的双方实际上都存在于数据库内，来直接在数据库内为该数据库中的表/实体之间的任何类型的关系进行建模。

乍一看，这似乎很容易，继续下去并在你的对象模型或领域逻辑中开始直接按此逻辑进行编码，Java 可能是你的首选语言，毕竟这是人类的自然倾向，希望使用我们用得最舒服的工具来解决问题。应该抵制这种倾向。数据库提供了大量功能，但在你的 J2EE 代码中并不容易（或普遍）使用。例如，大多数数据库都支持触发器（trigger）的概念，大段的数据库代码能够在基于行或基于表的任何类型的访问上执行，这令你能够在行被插入、更新、删除、或任何操作之后随即应用领域逻辑。试着在 J2EE 应用中构建此功能，它将在系统各处被普遍运用，这需要在你的代码中建立复杂的通告机制（时刻记住基于 RPC 的回调机制的缺陷，请参阅第 20 项中的描述）或者你直接手动对代表数据库实体的领域模型进行编码，然而随着时间推移，这将引领你走向一条梦魇般令人恐惧的维护之路。

作为 J2EE 开发者，在数据库层建立快捷方式可以令你更容易地使用数据库，这虽然看起来很省力而且貌似完全有理，但是，请抵制住这种诱惑。最终，如果你抵制不住诱惑，那将只会导致更多的问题。而当最不可避免的问题出现时，“嘿，我们注意到你有某些我们感兴趣并希望得到的数据，我们怎样才能得到它们呢？”最终将由你负责找到那些数据，而项目开发之前去处理这种问题，比在项目完成之后处理要容易得多。

第 46 项：惰性加载不频繁使用的数据

考虑到在网络上对远程数据库一次往返访问的开销，我们其实并不希望通过网络读取所有的数据，除非我们需要它。

这看起来是个很简单的思想，但它确实引人注目，在系统中经常使用，它将数据库对程序员隐藏起来，例如，幼稚的对象-关系映射机制经常以一对一的形式将对象映射为表，于是对于从数据库取回某个特定对象的请求，就意味着将一个特定行的每一列全部取出构造出要求的对象。

很多情况下，这显然是个糟糕的方式：考虑典型的深入检查（drill-down）场景，我希望列出数据项的清单，用户可以从中选择一个（或多个）进行更完整的检查。为了用户对选择有一个总体的映像必须列出大量的数据项，例如有 10,000 人符合初始查询条件，取回其中的每一个人意味着 10,000 x N 个字节的数据必须通过网络传输，其中 N 是表中单个行的完整尺寸。如果这些行有外键关系，它们也必须作为对象的一部分而被取回（Person 也许有 0..*个地址实例与之相连），那么最终的数量就很容易膨胀到无法管理的程度。

由于这些原因，许多对象-关系映射机制选择不为特定的行读取所有的数据，那是标准的读取操作的一部分，它们选择惰性加载数据，宁愿多做几次数据库的往返访问，以保持总的需求下降到可管理的级别。事实上，这种场景很常见，许多地方都对它引证，最引人注意的是惰性加载模式（Lazy Load Pattern）[Fowler，200]。

请注意：关键在于惰性加载的是不频繁使用的数据，而不是还没有被使用到的数据。

在这里，EJB 实体 bean 部分似乎要遇到灾难了，就像典型的 N+1 个查询问题。很多 EJB 实体 bean 的实现就决定了，对于不是立刻就需要使用的数据，我们希望避免全部都取回，通过基于 Home 的查找方法（finder method）取得一个实体 bean 时，实际从数据存储（关系型数据库）中取回的只是行的主键，它们被存储在 EJB 容器里的实体 bean 中。然后，当客户端访问实体 bean 的属性方法（get 或 set）时，就会发出单独的查询以取得或更新特定 bean 的特定列的值。

危险之处在于，如果你带着实体 bean 的帽子，却在客户端代码中写出如下的操作：

```
PersonHome personHome = getHomeFromSomewhere();
Person person = personHome.findByPrimaryKey(1234);

String firstName = person.getFirstName();
String lastName = person.getLastName();
int age = person.getAge();
```

使用这样的代码，你其实是提出了如下这些要求：

```
SELECT primary_key FROM Person
   WHERE primary_key = 1234;

SELECT first_name FROM Person
   WHERE primary_key = 1234;

SELECT last_name FROM Person
   WHERE primary_key = 1234;

SELECT age FROM Person WHERE primary_key = 1234;
```

这除了显著的浪费 CPU 周期（解析每次查询，规划查询计划，将返回值编组成关系元组等等），同样还令 cache 淹没于不相关的数据中，同时你也承担着巨大的风险，即在 bean 方法调用之间数据不能改变。请记住，每次调用代表着一个独立的 EJB 驱动的事务，然而你现在却打开了这种可能性，在方法调用之间不同客户能够改变相同的行（请参阅第 31 项）。基于这些（以及更多的）原因，产生了两个模式：会话外观（Session Façade）[Alur/Crupi/Malks，341]和数据传送对象（Data Transfer Object） [Fowler，401]。

然而，这个问题与惰性加载并没有必然联系。关键是要了解什么数据需要惰性加载。在容器管理的实体 bean 实现的例子中，与 SQL 相关的所有细节对于 EJB 开发者来说都被隐藏了，EJB 容器没有办法知道哪些数据元素比其它的元素更可能被使用，所以它采取了更严厉的假设，这些数据都不一定会被使用。现在回想起来，这可能是个糟糕的决定，仅仅比留给容器实现者可选择的另一种方法稍好一点，那就是假设所有的列都会被使用。对于实体 bean 实

现者最理想的环境是，允许你提供某种关于使用的提示，作为初始化 bean 状态的一部分，指出什么字段需要读取，而哪些字段最好是惰性加载。但这种事情是依赖于供应商的，会在相当大的程度上减小你的可移植性（如果你关心这个问题，请参阅第 11 项）。

如果你在编写自己的关系访问代码，例如编写 BMP 实体 bean 或者就是直接写 JDBC 或 SQL/J 代码，那么对于读取什么数据的决定权就在落在了你的身上。在这里，就像任何 SQL 访问一样，你应该精确地指明你想要的数据：一点也不多，一点也不少。这意味着需要避免如下这种 SQL 查询：

```
SELECT * FROM Person
    WHERE ...
```

虽然在你写这个查询的时候，它似乎没那么重要，但当代码执行的时候，你会取回 Person 表中的每一列。如果 Person 表一开始的定义正好与你想要的数据列一致，那么使用通配符 * 来取得所有列似乎更方便更简单，但是请记住，你并不拥有数据库（请参阅第 45 项）。在很多情况下，数据库的定义可能会改变，如果你在代码中使用通配符的话，就会出现问题。

- *增加额外的列。*嘿，需要更新了，除非你喜欢这份不讨好的工作，回头遍历你所写的每一行 SQL 代码，以确保能够处理额外增加的列，否则你将读取你不想要的数据。
- *通配的 SELECT 语句不指定列的顺序。*遗憾的是，SQL 标准并不能保证 SELECT * 这样的查询所返回的列的顺序，在某些情况下，甚至你所使用的数据库产品也不能保证，这就意味着你底层的 JDBC 代码如果依赖返回的列的隐含顺序，那么在试图将 first_name 列作为 int 读取的时候，就会受到 SQLException 实例的热情款待。更糟糕的是，这种问题不会在处于开发状态的数据库中出现，因为数据库管理员不会为了提高性能而乱调乱改数据库定义，它只会在最终的产品系统中显现出来，这将令你在迷惑中得好好挠挠头才能搞清楚问题出在哪儿。
- *隐含的文档提示丢失了。*老实讲，如果你列出你需要的列，以显式地指明你想读取什么，这将令事情更清楚。这也意味着你的 Java 代码只需维护更少的注释；无论何时，只要可能，最好的方式是让代码为自己作主。

不管是否需要多打一些字，最好还是显式地列出所需的列。幸运的是，通过代码生成工具可

以相当容易地生成这个列表，机会真的是来了。

然而，惰性加载并不只是限制从查询加载的列。还有其它理由使我们在更大的规模中应用相同的原则。例如，回到经典的深入检查场景。在多数瘦客户端应用中，当取得查询结果时，我们经常是向用户显示前 10、20、或 50 个结果，这与显示结果的窗口尺寸有关。我们不希望向用户显示完整的结果集，因为当用户提供特别模糊的约束时，可能会有上千个数据项。这么做的理由各种各样，但归结起来无非是这种思想，用户一般在前 10 个或 20 个数据项中就找到了他们需要的数据项，否则就使用更强的约束，重新再查一次。

如果将另外 990 行数据项也取回会发生什么呢？浪费空间，浪费 CPU 周期，浪费带宽。

有几种方法可以控制 SQL 查询读取多少数据。一种方法是在 JDBC 级别通过设置 setFetchSize，将需要读取的项的数量设为等于你呈现给用户的显示窗：如果你只显示读取的前 10 条数据，就在通过 next 读取第一个条之前，对 ResultSet 使用 setFetchSize(10)。使用这种方法，你可以确定只读取指定数量的行，而且你知道你是通过一次往返访问以获取这些你需要使用的数据的。另一种方法是，对于驱动/数据库的组合，你可以让驱动程序设置它认为最佳的获取数量，通过调用 setMaxRows 语句来限定返回行的绝对数量；任何超过了传给 setMaxRows 的数量的多余行都会被悄悄地丢掉，不会被传送回来。

有些数据库还支持另一种方法，使用 TOP 限定符作为查询的一部分，例如在 SELECT TOP 5 first_name, last_name FROM Person WHERE...中，只有符合该谓词表达式的前 5 个行才会从数据库中返回。虽然这是非标准的扩展，但它仍然很有用，而且许多数据库供应商都支持。TOP 是 SQL Server 的语法；其它数据库使用 FIRST（Informix）、LIMIT（MySQL）、或 SAMPLE（Oracle），原理都是相似的。

再说一次，该思想很简单：只取回当前需要的数据，避免取回不会用到的过多数量的数据。然而请当心，当你开始考虑惰性加载的时候，你就站在滑梯的斜坡上了，如果你不当心的话，很容易就会发现自己陷入了过多地通过网络惰性加载数据元素的境地，Fowler 称之为“波纹式载入（ripple-loading）”问题，或者更有名的 N+1 次查询问题[Fowler，270]。

在那种情形下，为避免网络拥塞有时最好是采用积极加载数据（请参阅第 47 项）。

第 47 项：积极载入频繁使用的数据

积极加载与惰性加载（请参阅第 46 项）正好相反：不再是通过增加的网络往返访问，在后来需要的时候才取回数据，因为考虑到我们终将会需要这些数据，所以我们决定现在就通过网络读取额外的数据，然后将它保存在本地数据库中。

此思想隐含了一些条件。首先，对于最终得到的结果，即实际访问的额外数据，必须证明它带来的开销的合理性，即对通过网络从服务器传送到客户端的数据进行编组的开销。如果这些数据永远也不会被使用，积极加载就是浪费网路带宽，并损坏了可扩展性。对于只返回一行的查询，即使它有许多列，其损失也是可接受的，特别是比起为其中每一列都进行额外的数据库访问带来的开销。然而如果有 10,000 行，则每个用不到的列都绝对是一个打击，特别是如果不止一个客户端在同时执行此代码时。

其次，我们也隐含假设了，在网络上移动数据的实际开销并不过分。例如，如果我们正在讨论取得一个巨大的 BLOB 列，请确定该列确实是必须的，因为大多数数据库对于获取与传送 BLOB 数据并不做特别的优化。

关于积极加载数据，你也许听到的不多，因为对大多数开发人员而言，它让人回忆起对早期的对象数据库的恐怖印象，以及当某些“根”对象被请求时，他们对积极加载对象的癖好。例如，如果一个 Company 包含 0 个或多个 Department 对象，每个 Department 包含 0 个或多个 Employee 对象，而每个 Employee 对象又包含 0 个或多个 PerformanceReview 对象……好了，想象一下，为得到数据库中当前存储的 Company 对象的列表，有多少对象会被读取。如果我们感兴趣的只是 Company 名字的列表，将所有的 Department、Employee、和 PerformanceReview 对象都通过网络进行传送，显然只是对时间和精力的巨大浪费。

事实上，积极加载和对象数据库没有关系（除了它们都因此受指责之外），我曾使用对象-关系层做过完全相同的事情，结果也完全相同。虽然“解决”方式不同，但这个问题与惰性加载一样：在获取对象状态时，底层并不知道什么数据需要被取回。所以，在采用积极加载的系统例子中，我们偏向于更少的网络往返，并将数据都取回来。

除了显然的带宽损耗，积极加载还是有许多惰性加载不具备的优点的。

首先，积极加载你所需要的完整的数据集将非常有助于并发的情形。请记住，在惰性加载的场景中，（如果没有使用会话外观[Alur/Crupi/Malks，341]）客户端有可能会看到语义上被损坏的数据，因为每次实体 bean 访问都会导致在独立的事务下的一次单独的 SQL 查询。这意味着，在事务与事务之间，在我们不知道的情况下，其它客户端就能够对行作出修改，这会导致所有已经取得的数据都变得无效。当积极加载数据时，作为一个单独的事务的一部分，我们取得完整的行的副本，这就排除了对数据库多次访问的需要，也消除了会破坏数据语义的可能性。因为不再需要会返回被改变了的数据的“第二次查询”。本质上，积极加载采用传值的方式，与惰性加载的传引用方法正相反。

这对于锁窗口（请参阅第 29 项）也有一些非常有用的隐含意义。在会话 bean 的整个事务期间，如果通过 EJB 进行访问，或是通过 JTA Transaction 或 JDBC Connection 进行显式的事务维护，那么容器不必持有对应行（或页、或表，这依赖于你的数据库的缺省的锁机制）上的锁。缩短锁窗口意味着更低的竞争，也即更好的可扩展性。

当然，积极加载所有数据也意味着，对于其它客户修改数据打开了一个机会的窗口，因为你不再持有其上的事务锁，但是这可以通过各种方法解决，包括显式的事务锁、悲观并发模型（pessimistic concurrency models，请参阅第 34 项）、和乐观并发模型（optimistic concurrency models，请参阅第 33 项）。

其次，正如已经推导出的，对于给定的数据集，如果那些数据可以安全地假设为是确实需要的，那么积极加载数据能够大大降低花在通过网络获取数据上的总时间。例如，考虑用户的首选配置：为了让用户能够以各种方式对应用的使用进行个性化配置，个别的用户可以定义这样的数据，例如主窗口的背景图片，是否使用“基本”或“高级”菜单等等。这些数据在

用户登录时不会立即就需要，但是可以打赌，在应用的某一点它们总会被用到。我们可以惰性加载这些数据，但是考虑到每个数据元素（配置项目）很可能是相互独立的，若是获取每一个单独的数据项而不是独立的行，则我们在这个例子中可以再一次见到 N+1 次查询问题。因此我们一次获取整个数据集合并将其保存在本地，当决定了如何输出着色、装饰窗体、或任何配置后，就通过代码使用它。

积极加载并不仅仅是为了获取一行中的所有列。与惰性加载一样，你可以将积极加载原则应用于超过单独的行的范围。例如，我们可以将积极加载原则应用于表和数据依赖，这样的话，如果一个用户请求了一个 Person 对象，我们将加载与之相关联的所有 Address，PerformanceReview，以及其它与此 Person 相关联的对象，尽管这可能意味着以某种批处理形式的单次往返访问执行了多个查询（请参阅第 48 项）。

当我们处理它时，没有任何理由可以解释，为什么当一个表含有只读的值时，就不应该被积极加载。例如，很多系统将国际化的文本（例如一周中的每一天，一年的月份等等）存入数据库的表中，以便于在客户端进行修改。由于人们改变一周中每天的名字的可能性相当低，因此为了以后的使用，可以一次读取整个表，将结果以某种进程内的集合类或 RowSet 的方式保存。这也许会使系统在启动的时候时间长一点，但是终端用户对于每次请求-响应则可以获得更快的访问。（在很多方面，这正是第 4 项的特色。）

最后，积极加载数据与惰性加载数据一样，都是可行且有益的优化方式，尽管其名声并不好。实际上，很多情况下它是一种比惰性加载的情形更可接受的折衷方案，比较起我们当前生存环境中的昂贵而缓慢的网络访问，它对内存的额外需求相对来说开销更低。和平常一样，在做惰性加载或积极加载的优化之前，应该先进行性能测量（请参阅第 10 项），不过如果积极加载能够为你节省很多网络访问，那么第一次访问时占用的额外带宽，以及积极加载的数据所占用的额外内存，通常都是值得的。

第 48 项：批处理 SQL 的工作以避免往返访问

考虑到在网络上移动的代价，我们必须将通过网络去连接其它机器的次数最小化。遗憾的是，

JDBC 驱动缺省的行为模式是有一个语句，就进行一种往返访问：每次运行 execute（或者是它的某种变体，如 executeUpdate 或 executeQuery），驱动就会编组请求并将其发送给数据库，在数据库中解析、执行并返回。在每一次往返访问中，这个冗长的过程都要消耗大量的 CPU 资源。如果批处理这些语句，在一次往返传递中就将多个 SQL 语句传递给数据库，那么就可以避免一些开销了。

请铭记于心，尽管这主要是一个状态管理问题，但是通过减少使用网络的次数，以提高系统性能，它也同样可以运用于事务处理。在某种情况下，我们很可能持有开启的事务锁，因此我们应该使得锁打开的时间最小化。（请参阅第 29 项）

然而还有其它的原因，有时，多个语句需要在一个群组里执行以获得合乎逻辑的结果。通常情况下，我们希望在一个事务里完成这些语句（因此，这需要你对正在使用的 Connection 设定 setAutoCommit(false)），但是这并不意味着驱动程序会将它们在一个往返调用中执行。驱动程序完全可能分别地传送每一条语句，并在此期间保持事务锁为打开。因此，既然这些都发生在一个单一的事务中，那么将它们作为一个群组来执行就要好得多。毕竟，事务本身就是一个要么全部都做，要么全部都不做的过程。

```
Connection conn = getConnectionFromSomeplace();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();

// Step 1: insert the person
stmt.addBatch("INSERT INTO person (id, f_name, l_name) " +
  "VALUES (" + id + ", '" + firstName + "', '" +
  lastName + "')");
// Step 2: insert the person's account
stmt.addBatch("INSERT INTO account (personID, type) " +
  "VALUES (" + id + ", 'SAVINGS')");

// Execute the batch
int[] results = stmt.executeBatch();
// Check the batched results
boolean completeSuccess = true;
for (int i=0; i<results.length; i++)
{
  if (results[i] >= 0 || results[i] ==
```

```
      Statement.SUCCESS_NO_INFO)
      /* Do nothing—statement was successful */ ;
    else
    {
      /* Something went wrong; alert user? */
      completeSuccess = false;
    }
  }

  if (completeSuccess)
    conn.commit();
  else
    conn.rollback();
```

在此处的代码中，我们创建了一个 Statement，并使用它对数据库执行了两个非 SELECT 的 SQL 语句。（批处理对 SELECT 语句不起作用。）在这个例子中，我们向数据库中添加了一个人以及其新开的储蓄帐户的信息。要注意的是，此处执行批量处理的关键是使用 Statement 上的 executeBatch 方法，它会告诉驱动程序将这些语句全部一起送往数据库。

execueBatch 方法返回一个整数数组，以此表示批处理方法是否成功。数组中的每个元素对应批处理中的一个语句；0 或者正数表示“更新计数”结果（此次方法调用影响到的行的数量），如果返回值为 SUCCESS_NO_INFO 则表示调用成功，但是没有更新计数。

在有一个语句失败的情况下，JDBC 允许驱动程序作出以下几种选择中的一种。它可以抛出异常并在当前点停止执行，或者可以继续执行，而在返回的整数数组中的对应位置设为 EXECUTE_FAILED。

注意，批处理的执行并不蕴含事务的边界，那两个语句已经执行了，但是由于 AutoCommit 被关闭，它们还没有被提交给数据库。因此，需要由我们完成提交或者回滚工作；在此例中，我们只在所有语句都成功的情况下才进行提交，这是一种合理的策略。如果你在调用 executeBatch 之前调用 commit，那么批处理的语句会被发送出去，就好像你是在 commit 之前调用了 executeBatch。如果你试图在批处理时将 AutoCommit 设为 true 的话，其结果在 JDBC 规范中没有定义，而这可以简单地理解为“很可能引发某种异常”。

然而，JDBC 的 executeBatch 方法并不是唯一的批处理执行方式。因为许多数据库都支持某种逻辑的行终止符，这就使我们有能力将多条 SQL 语句作为一个逻辑上的输入行来执行，于是就可以像下面这样调用 JDBC：

```
Connection conn = ...; // Obtain from someplace
Statement stmt = conn.createStatement();
stmt.execute("SELECT first_name, last_name FROM Person " +
    "WHERE primary_key = 1234; " +
    "SELECT ci.content, ci.type " +
    "FROM ContactInfo ci, PerConLookup pc " +
    "WHERE ci.primary_key = pc.contactInfo_key " +
    "AND pc.person_key = 1234");
ResultSet personRS = stmt.getResultSet();
    // The first SELECT

if (stmt.getMoreResults())
{
  // The other SELECT
  ResultSet contactInfoRS = stmt.getResultSet();
}
```

虽然使用起来更显笨拙[5]，但是如此批处理语句也有其优点，可以像使用INSERT、UPDATE和DELETE语句一样使用SELECT语句，这也使得我们向着基本目标更进了一步，即仅在一次单一的到数据库的请求-响应周期内运行多个SQL查询，以分摊多个SQL查询的开销。

第 49 项：了解你的 JDBC 供应商

尽管经过了这么多年的修改，JDBC规范仍然没有刻意去制订大量的细节。例如，当新建一个ResultSet时，它通常（这也不是必须的，仅仅是一个惯例）只持有前N行，N是JDBC供应商认为比较合适的某个值。对于一些主要的供应商，N的值就是 1，我可不是在开玩笑。幸运的是，这个值不仅可以看到，而且能够通过getFetchSize和setFetchSize这两个ResultSet的API进行配置。但是问题仍然存在，你知道它的缺省值是什么吗？如何判断

[5] 在产品代码中，你应该检查execute返回的布尔值，以确定第一个SELECT语句产生的结果，此处我是为了代码更清晰，你不应该像我一样忽略它。

它是否合适呢？一定要检查setFetchSize的布尔返回值，这样才能确保你所请求的抓取尺寸没有超出驱动程序或者数据库的支持范围。

JDBC 的许多特性都是不可用的，它们依赖于驱动程序的能力，同时这也会对你如何利用 JDBC 驱动进行编程有着很大影响。例如，当从 ResultSet 中获取数据时，可以通过 ResultSet 中的顺序位置或者通过列名来获取数据，这两种方法看起来完全等价。除了程序员个人的编程习惯之外，这两种方法难道真的就没有任何不同吗？

事实表明两者之间确实存在不同，这完全依赖于你所使用的 JDBC 驱动是怎么实现的。最早发布的 JDBC 规范（随同 JDK1.1 发布的版本）仅要求 JDBC 驱动提供对消防水龙式的游标（*firehose cursor*）的支持。也就是说，一旦某一行或一列通过游标被取出，就无法再次获得了。这意味着当你从 ResultSet 中读取数据时，如果你没有按照在 SQL 语句中指定的顺序去读取某一列数据时，你就会跳过前面所有列的数据，而这些数据将永远不能再被读取了：

```
ResultSet rs =
    stmt.executeQuery("SELECT id, first_name, last_name " +
                      "FROM person");
while (rs.next())
{
  String firstName = rs.getString("first_name");
  String lastName = rs.getString("last_name");
  int id = rs.getInt("id");
  // Error! This will throw a SQLException in a JDBC 1.0
  // driver
}
```

也许你会说你一直在使用 JDBC2.0 或者到目前为止更好的驱动程序。如果某人使用 JDBC-ODBC 驱动来部署你的代码（这个想法从一开始就错了，因为 JDBC-ODBC 驱动是一个未受支持，并且充满 bug 的 1.0 版驱动程序，它不仅速度慢得惊人而且据说会导致某些 ODBC 驱动配置的内存泄露），你的代码马上就会抛出很多原因不明的 SQLException。

现在，如果你关注怎样写出可移植的J2EE代码，你必须确保这种特殊情况不再出现，以彻底解决这种 1.0 版驱动程序的可能性。因此，你必须严格按其顺序来获取数据，最简单的

方法就是使用序号的形式来调用方法：

```
ResultSet rs =
   Stmt.executeQuery("SELECT id, first_name, last_name " +
                     "FROM person");
while (rs.next())
{
  int age = rs.getInt(1);
  String firstName = rs.getString(2);
  String lastName = rs.getString(3);
}
```

这样的代码虽然不是很难书写，但是它们确实存在一些缺点。第一，如果一个程序员（或者，可能更糟糕，一个并不熟悉 JDBC 1.0 API 的这些小怪癖的数据库管理员）曾经修改过 SQL 语句，循环中对应的数据检索的代码也要更新以反映列顺序的更改，否则 SQL 语句又将抛出异常。（顺便讲一下，用序号的形式调用方法确实比用列名的形式速度要快一些，但是应该避免仅仅因为性能原因就将代码改为使用序号的调用形式，因为它的优化能力不值一提，尤其是，那就失去了使用列名带来的文本固有的提示信息。如果你确实希望优化代码，还有许多其它优化方法也许能带来更好的结果，所以不要采用这种方式了。）

我们讲这个并不仅仅是为了弄清楚列的顺序（顺便提一下，关于 SQL 语句中使用*替代所有列名曾经存在过争论）。你需要知道你所使用的驱动是否支持批处理语句（请参阅第 48 项）和隔离级别（请参阅第 35 项），它可能支持哪个数量级的功能（scalar functions）等等。许多信息能够通过 DatabaseMetaData 类获得，而 DatabaseMetaData 的实例可以通过 Connection 取得；又或者可以通过 ResultSetMetaData 类获得稍少一些的信息，可以通过给定的 ResultSet 得到 ResultSetMetaData 的实例。SQL Performance Tuning [Gulutzan/Pelzer]提供了好几个图表，这些图表描述了八个供应商的数据库的返回结果(IBM, Informix, Ingres, InterBase, Microsoft SQL Server, MySQL, Oracle, 以及 Sybase)，可是你无论如何还是要进行一些测试，因为就算你使用的数据库就在前面的数据库名单中，你还是不能确定那些图表对于你的特定数据库是否适用，因为每一次新发布版本的数据库的性能都会有所不同。

JDBC 程序员感兴趣的另一个地方是线程安全：多线程调用驱动是否安全？在多线程中访问

Connection、Statement、或ResultSet对象时，需不需要控制同步？例如，JDBC-ODBC驱动不是线程安全的，这样就需要由你确保，在同一时刻驱动不会被多于一个线程进行访问，否则在 Java 代码中调用此 ODBC 驱动就有可能导致非常糟糕的事情发生。（这里又一次强调，在产品代码中使用 JDBC-ODBC 驱动绝对不是一个好主意。）

在许多 JDBC 和 J2EE 的书中，为了最佳的执行性能与可扩展性，它们提出了很多建议，其中一个思想是 PreparedStatement 比 Statement 好：应该使用 PreparedStatement 取代常规的 Statement。道理很简单：因为每次使用数据库都需要一定量的工作（解析 SQL，创建查询计划，进行优化等等），如果能够假设将会发生同样的调用的话，最好是能够分摊这种开销，于是可以将上一次的这些预备工作保持到下一次。一个 PreparedStatement 只“准备”一次 SQL 调用（除去你想要传进参数，因为还不知道这些参数是什么呢，所以无法准备那些参数），这样在后续的调用中你就能够获得更好的性能。

从安全的角度来看，使用 PreparedStatement 是必须的（请参阅第 61 项），但是从性能和（或）可扩展性的角度来看就不一定了。例如，JDBC 规范指出，当一个 Connection 被关闭时，它所对应的 Statement 对象也随之关闭。这样当你将从连接池中取出的 Connection 返回给连接池的时候，从该 Connection 中获得的 PreparedStatement 是否也随之关闭了呢？又或者底层的物理连接仍然打开，这样使得 PreparedStatement 依然处于活动状态准备接受其它的请求呢？从 PreparedStatement 继承而来的 CallableStatement 是否也遵从同样的约定呢？

遗憾的是，当我在挥手并声明“很明显，实际的驱动程序会在连接池中保存 Statement，只有傻子和笨蛋才不这样做”的时候，现实的情况是，有时你不得不使用傻子和笨蛋编写的软件，只有通过测试才能确信你的数据库或者数据库驱动是否属于这一类。

顺便说一下，某些数据库供应商（最显著的是 PostgreSQL）在一个事务结束的时候（COMMIT）会将 PreparedStatement 变成“无准备的”。当讨论这个问题的时候，关于 PreparedStatement 的讨论向左来了个急转弯。也就是说，下面的代码[Gulutzan/Pelzer，336]不会按我们预期的工作，尽管 JDBC 规范坚持它应该按我们预期的工作：

```
PreparedStatement pstmt =
  connection.prepareStatement(“. . .”);
boolean autocommit =
  connection.getAutoCommit(); // Let’s assume it’s true
pstmt.executeUpdate();
// COMMIT happens automatically, since autocommit == true
pstmt.executeUpdate();
// FAILS! PreparedStatement pstmt is no longer prepared
```

如果这还不能使你对“你有必要去了解你所使用的 JDBC 驱动程序到底是怎么工作的”这一思想信服，那么我只能祝你好运了。

要知道你的驱动程序到底做了什么，最简单的方式就是使用数据库执行查询的性能测量工具，并且看一看在JDBC查询执行的过程中到底发生了什么。在某些圈子里，这被视为探险测试（exploration testing），Stu Halloway(http://www.relevancellc.com)对此有详细的介绍。简单地说，就是编写一系列基于JUnit的测试程序，并以此来验证你关于执行一系列调用时会发生什么的设想（例如前面所提到的PreparedStatement代码）。然后在多种不同的情形下执行这些测试程序，例如不同的JDBC驱动，不同厂家的数据库，或者同一厂家的数据库的不同版本。

第 50 项：调整你的 SQL 语句

小测验时间：什么时候两个逻辑上等价的 SQL 语句会不等价？让我们来看看下面的两个 SQL 语句：

```
SELECT * FROM Table WHERE column1=’A’ AND column2=’B’
SELECT * FROM Table WHERE column2=’B’ AND column1=’A’
```

你认为哪一个执行速度会快些[6]？

答案是：你不知道，大多数数据库的优化器也不知道。但是，如果你恰好知道对于特定数据库中的特定用户或者特定查询，column2 的值为 B 的数据要少得多，这样的话第二个语句

[6] 这个例子出现在SQL Performance Tuning[Gulutzan/Pelzer，24]一书中。

将会比第一个执行得快，尽管逻辑上它们是等价的（顺便提一下，对于 Oracle 数据库，当使用基于开销的优化器时，一定不要这样做，它肯定会出错）。

或者我们再看另一个例子，下面这两个语句将会怎样？

```
SELECT * FROM Table WHERE column1=5 AND
    NOT (column3=7 OR column1=column2)

SELECT * FROM Table WHERE column1=5 AND column3<>7 AND
    column2<>5
```

答案是：对于前面所提到的八个数据库中的五个来说，第二个语句执行得会快些[7]。同样的，在逻辑上它们是完全一样的语句，所以它们的执行结果将完全相同。只是数据库优化器对待第二条语句的方式不同于第一条，所以获得了更佳的响应时间。

当使用关系型数据库时，调整 SQL 语句仍然是你能够做到的最佳优化。不管 JDO 和实体 bean 的供应商在过去五年的努力中得到了什么，我们仍然处于 SQL 很重要的时代。是的，供应商确实使他们的对象优先持久化模型进步了很多，不像最初那么糟糕。但是，如果数据库本身不能区分这两种语句的不同之处，那么你的对象-关系层也就不可能区分得出来。这就是为什么对于一个对象-关系系统而言，它必须要向你提供某种类型的挂钩点（请参阅第 6 项）是如此重要，通过它你才能够向其中传入原始的 SQL 语句，并对那些最经常执行的查询进行调整和优化。

顺便说一下，你应该铭记于心，对某些数据库而言以下两个语句是完全不同的，因此需要重新解析、重新计划、并重新执行：

```
SELECT column1*4 FROM Table1 WHERE COLUMN1=COLUMN2 + 7
SELECT Column1 * 4 FROM Table1  WHERE column1=(column2 + 7)
```

这就是说，即使这两个语句的行为完全是一模一样，但由于它们所使用的大小写与空白符不同，数据库将它们当作独立而无关的语句分别处理。当使用工具产生你的查询语句时，我们希望它们能够使用统一风格，但是你能确信这一点吗？（理想情况下，在为此过多的担心之

[7] 这个例子出现在SQL Performance Tuning[Gulutzan/Pelzer，16]一书中。

前，你更需要了解此类事情会不会影响到你的数据库，不过这又意味着你对数据库供应商要有着更深入的认识。请参阅第 11 项。）

在许多情况下，你必须知道数据库中 SQL 语句实际是如何执行的，然后才去考虑如何调整它。对于某些对象-关系系统而言，如果它们不暴露出实际产生的或动态创建的 SQL 代码的话，这就很困难了。幸运的是，答案只有一步之遥了。

首先，大多数数据库产品都提供了某种基于数据库的、在给定的数据库实例上被执行的查询的视图，因此，能够相当容易地使用数据库性能检测工具，并且能够看到实际的 SQL 语句。（当你这么做时，首先假设你的数据库性能检测工具提供了查询分析，针对性能检测工具运行这几个 SQL 查询，以观察数据库是如何去执行这些特定查询的。请参阅第 10 项。）这将会告诉你，你所喜欢的对象-关系工具是如何产生 SQL 查询的，因此，使你能够清楚地了解到对 SQL 的优化是否适宜。

如果你使用的是没有提供这种工具的数据库（在我看来是时候换一个新的数据库了，但是有时你被迫只能使用手边的工具），假设你的对象-关系层仍然通过 JDBC 与数据库进行对话，你可以使用一个小技巧，给对象-关系层提供一个 JDBC 驱动，通过这个驱动能够“泄露”进入它的实际的查询语句。这就是 P6Spy 驱动程序，你可以从 http://www.p6spy.com 得到，它的操作非常简单：它暴露出这个星球上所有 JDBC 驱动都会提供的相同的接口，但是它实际上不做任何工作，而是将工作都委托给你提供了配置数据的 JDBC 驱动。P6Spy 驱动只是简单地显示对多个数据源的 SQL 查询，包括 Log4J 日志记录流。

在那些对象-关系层不进行 SQL 优化，以及对象-关系层不提供某种挂钩点以传入优化的 SQL 的情况下，你可以有以下几种选择：

- *转换工具*。有许多其它对象-关系的映射工具，商业的或开源的都有，你没必要因为其中一种没有提供你必须的优化挂钩点就感觉“不能动弹”了。
- *采用快车道（Fast Lane）模式*[Alur/Crupi/Malks，第一版.]，它提倡直接对数据库进行JDBC存取，因此要对被发送的SQL语句直接进行控制。在基于EJB访问的情况下，这意味着从你的会话beans直接进行JDBC调用或者像BMP beans那样重写你的

CMP 实体beans。遗憾的是，这种方式有许多直接的缺陷（这可能就是为什么第二版中不再提倡它的原因）。例如，如果对象-关系层正在进行任意类型的缓存管理，你将会绕开缓存管理和任何正在被维护的隐含的同一性映射表（Identity Map ）[Fowler，195]。也许你能够克服没有缓存的损失，但是，如果正在维护的同一性映射表（[Fowler，195]，是为了允许对象-关系层延迟向数据发送尚未提交的改变，绕开它将会给你造成一些难以应付的并发问题。

- *用你的对象-关系工具玩调包把戏，*这是由P6Spy驱动处所获得的提示（典型的装饰器（Decorator）[GOF，175]，如果曾经有一个的话）。写一个类去实现JDBC驱动的接口，使它看上去就像JDBC驱动。用它来监听那些与你想要进行优化的查询相匹配的查询，然后用手动调整和优化后的SQL语句替代此查询，在对此查询进行包装之后再交给真正的JDBC驱动。作为对此驱动程序的损害，你必须采用额外的方法调用层，但是假设你做完了所有的预备工作（请参阅第 10 项），调整后的SQL语句带来的好处将超过额外的方法调用的开销。

不管你选择哪种方式，在你将缓存 JNDI 的查询结果作为一种优化之前，从你的立场上好好看看将要执行的 SQL 语句，对 SQL 语句进行优化，往往比你在别的事情上花费时间精力能够获得更大的效益。即使你的 CMP 实体 bean 层有某种适应 SQL 生成的模式，你还是需要了解它，并确保你的数据库管理员所进行那部分工作不会在其上不能运转。

第六章 表示

一切不过是对话框和数据而已。

——Mike Cohn

表示：看起来真是太简单了。只要随便准备一些对话框和按钮，再加上几个列表框和滚动条，用户就能够很容易地明白这是什么意思了，对吗？只要获取用户的输入，进行一些处理，再把它发送回服务器，我们就算完成任务了。

对许多开发者来说，构建用户界面很可能算是工作中最有趣的部分。其中部分原因在于：与系统中的其它部分不同，用户界面是看得见，摸得着的，它是衡量项目进度的指示器。今天早晨，窗体还是一片空白，到了下午，就有了一个用户可以交互的窗体（尽管内部没有进行任何处理）。用户界面还是系统中用户能理解并欣赏的少数部分之一。很少有（就算有的话）用户会去查看你的事务处理代码，并说“哦，支持 ACID 的事务处理，你小子挺牛啊！”，而多数情况是：在项目进度会议上，向普通用户展示一个漂亮的用户界面，并演示 UI 窗体如何能够根据用户的输入而自动地显示有关数据。然后整个会议室就会充满对项目团队的赞誉。如果让你在“哦，这到底是什么！”的反应和“用户的赞誉和尊敬”之间进行选择的话，你还会对许多程序员喜欢编写用户界面感到奇怪吗？

遗憾的是，对于盲目乐观的 UI 程序员，这里潜藏着许多危险。

许多开发者发现（通常是项目即将发布之前才会注意到），对于系统应该做什么，用户有一个严格的概念模型，一旦用户界面不能反映这个概念模型，那么用户的赞誉马上就会变成指责。因为开发者常常会在最后关头做出突发奇想式的修改，以安抚用户，所以用户界面常常是应用中修改量最大的部分。

而且，许多用户逐渐认识到，程序员并非总是进行用户界面设计的最佳人选。Visual Basic 之父 Alan Cooper，他也是用户界面书籍《The Inmates Are Running the Asylum》 [Cooper99] 和《About Face 2.0》 [Cooper03]的作者，一生都在从事有关面向客户的用户界面设计方面的咨询工作。在给出的某些蹩脚的用户界面设计的例子中，他从一些以用户为中心的应用中（主要来自太平洋西北部的一家大型软件公司），引用了用户界面方面的研究成果。作为开

发者，在我们看来显而易见的内容，但是对于那些不以编写代码谋生的人来说，却非常晦涩和神秘。

如果这两大陷阱还不够的话，对于基于 Web 的应用，情况还要糟糕。虽然 Web 能够把成千上万的计算机用户互连起来，形成一个庞大的社区，但是用户进行通讯的主要门户，也就是 Web 浏览器，它把我们带回到了那种其最吸引人的 UI 风格就是“图形化按钮”的应用，这使得用户界面至少落后了 10 年。更糟糕的，基于 HTML 的应用完全依赖于服务器，这虽然表明不需要在客户端安装任何特殊的软件，以执行基于 HTML 的应用，但也同时表明，所有内容必须通过网络传输，这就显著地增加了对带宽的消耗。对于要求能够扩展到支持百万个并发用户的应用，过大的带宽消耗正是我们要极力避免的。

最终结论？构建表示决不仅仅是在 JSP 页面上添加一些 HTML 标记就行了的。

第 51 项：考虑富客户端 UI 技术

不，这一条并不是指富有的客户要求对应用做出修改，而是指与瘦客户端技术相比（基于 HTML 浏览器），能够呈现更为丰富、更加强大的用户界面的表示形式。

与传统上两层的、胖客户端的客户/服务器应用相比，基于 HTML 浏览器的瘦客户端应用具有明显的优势：较低的甚至压根就没有的部署成本。因为 HTML 文件位于服务器，所以部署新版本的应用只需在客户端进行零安装。只要客户端有 Web 浏览器，用户就可以使用该系统。准确地说，因为现在每个操作系统都把浏览器作为基本安装的一部分，所以只要把应用安装到服务器端，客户马上就可以开始使用该应用。更重要的是，因为不需要在客户端机器上进行更新，所以对应用的更新可以按实际需要频繁地进行。这种“零部署场景”十分有用，尤其是对于企业级应用系统。这是因为，企业级系统往往比那些放在货架上销售的软件更容易发生变化。用户甚至没有必要知道这种版本的变化，无论如何，他们总是通过访问服务器来使用系统的。

不过，HTML 也存在着不利的方面：

- *可移植性*：从一开始，HTML 就被设计成跨平台的技术，结果是，它或多或少地受到了这种可移植性的困扰。HTML 表单上的控件完全是按照主机（浏览器）的意愿进行绘制的，所以在不同的平台上，表单会具有不同的行为。所以，要想设计易于使用的表单，即使不是不可能的，也会是相当困难的，尤其是当表单相当复杂的时候。
- *状态管理*：因为 Web 浏览器不能持有客户端状态（cookie 除外，不过它的作用非常有限），所以基于 HTML 的瘦客户应用需要在服务器上持有所有的用户状态，这个状态通常由一个不透明的令牌所表示，该令牌一般存储于 cookie 中，或者作为后续请求的一部分进行传递（也称为 URL 重写）。不过，这就增加了服务器的负载，并降低了服务器的可扩展性。另一种办法是让应用成为完全无状态的，并且在浏览器和服务器之间传递所有数据，这些数据往往被放在表单的隐藏字段中，或者直接嵌入到请求的 URL 中。不过，这种方法也有问题。因为所有状态数据都要在每次请求中来回进行传递，这就带来了额外的带宽损耗。还有安全方面的问题。有一个流传甚广的（也许是虚构的）故事，有一个很聪明的客户，打算在线购买一台计算机。他注意到计算机的价格存储在表单的隐藏字段里。于是他模仿在线销售订单编写了一个本地 HTML 页面，然后把隐藏的价格字段的值设定为 1 美元。然后他提交了这个订单，并被系统接受。实际上，他只用 1 美元就购买了价值 5000 美元的计算机。（这个故事还有后话，计算机的制造商试图起诉这名客户，但是被驳回。因为这是在线销售系统自己设计上的问题。后来这个在线销售系统再也不把价格字段放在表单的隐藏字段中了。）
- *控制*：HTML 几乎没有提供多少办法来对如何呈现页面上的元素进行控制。事实上，HTML 最初的方式是设计尽可能高的层次，把如何呈现 HTML 页面上元素的工作交给主机上的浏览器去负责，这样显示出来的内容就能够尽可能地接近客户平台的风格。不过，随着时间的推移，HTML 页面设计者对于如何呈现浏览器里的表示元素，已经找到了更多的办法（所有优秀的用户界面设计人员都会这么做）。层叠样式表（CSS）提供了某种上述的控制能力，但尽管如此，也并非所有的浏览器都完全支持 CSS，CSS 也不能对所有的表示元素都提供完全的控制能力。例如，对于某个表示元素，当需要的字体在客户平台上并不存在的时候，页面设计者必须自行提供一些字体。
- *跨平台的差异性*：尽管万维网联盟（World Wide Web Consortium）做出了巨大努力，但 HTML 仍旧是互联网上最少被遵守的标准之一。两种主流的浏览器，微软公司的 Internet

Explorer 和网景公司的 Netscape Navigator (Mozilla)，都提供了显著的扩展功能。并且尽管他们都声称完全遵守 HTML 4.01 标准，但是在呈现某些合法的 HTML 4.01 标记的某些刻面时，二者似乎都存在一些问题。当考虑到客户端脚本语言的时候，问题就更严重了：Internet Explorer 同时支持 VBScript 和 Jscript，而 Navigator 则使用 JavaScript。它们都提供了自己的对象模型，用来与页面上的元素进行交互。并且它们都在新版本的浏览器里添加了增强功能，这表明，站点不仅要考虑浏览器的类型，甚至还要考虑浏览器的版本。这样的结果是，通常网站只是选择支持某个提供商的特定版本的浏览器（“请用 Internet Explorer 5.5 浏览本站点”），并只针对这种浏览器进行编码，其他被挡在门外的用户，或多或少地就不那么走运了。

- *带宽*：HTML 被设计成一种表述语言，它向浏览器提供一些标志和命令，来定义如何在客户端呈现用户界面。用户每进行一次动作，就需要新的界面，整个界面（包括数据和表示元素）都必须传送到浏览器。这表明每次请求都需要耗费额外的带宽，从而增加了网络（包括服务器）的负载。某些站点通过只向浏览器传送数据，然后在浏览器进行 XSLT 转换以形成用户界面，试图减少带宽的耗费，但只有 Internet Explorer（并且只在最新的几个版本里）才支持 XSLT。即便如此，浏览器也必须通过另一次 HTTP 请求，才能得到 XSL 脚本。
- *缺乏表示元素*：HTML 表单极其简单，只提供了六种可用的控件：按钮、单选按钮、复选框、编辑控件（单行、多行和密码输入）和下拉列表框。尽管你对此可能有异议，但是对任何应用，有这些“核心”控件就足够了，并且 HTML 还可以通过可点击的图像来提供额外功能。但事实是，与浏览器所在的操作系统能够提供的 UI 元素相比，HTML 的表现能力非常有限。虽然能够使用动态 HTML（DHTML）或客户端脚本语言来模拟菜单条和工具条，但是我们又会遇到跨平台支持的问题。

之所以还有如此多的应用开发者愿意在开发过程中去忍受这些问题，只不过是看上“零部署”的好处罢了。实际上，许多应用仅仅只支持 HTML 4.0，因此，应用的用户界面看起来就像是回到了 1980 年代的早期。

这里的问题在于，开发者已经忘记了自己采用基于 HTML 的界面的初衷：在部署新版本代码的时候，很少的甚至是为零的部署成本。对于某些应用，HTML 当然已经足够，不过考察一下其它 UI 技术，看看是否既能够得到零部署成本的好处，又能够解决这些问题，也很

不错。实际上，确实有几种富客户技术，既能够提供零部署成本的好处，又拥有更丰富的客户端功能。

富客户技术的一个主要好处在于状态管理：它不仅能够在本地持有状态信息，在需要的时候（比如当用户最终选择了 OK 或者 Apply 按钮的时候）发送到服务器进行处理，而且，富客户应用的生存期还可以反映出用户会话的生存期。当用户离开富客户应用的时候，能够进行必要的清理操作，以释放占用的资源。因为在客户端保持所有状态，服务器就免除了这种负载。这样我们就能够避免所有有关 cookies、服务器端会话状态（这会导致更加影响服务器可扩展性的问题）、会话过期等问题。仅凭这一点，就足以说服 servlet 开发者去考虑一下其它选择，而不仅仅是 HTML。

以下小节将对几种可用的富客户技术进行简要讨论。

动态 HTML

如果我们把客户端表示技术的发展看成一个数轴，传统的胖客户技术位于数轴的最右边，纯 HTML 4.0（或者其继任者，XHTML 1.0）位于最左边，DHTML 技术则处于 HTML 4.0 右边一点。它实质上就是标准的 JavaScript/JScript/VBScript 脚本代码，从服务器上作为 HTML 的一部分而下载，然后在客户浏览器里执行。两种主流浏览器都对 DHTML 进行了广泛支持，能够得到类似鼠标浮动效果这种非常有趣的功能，这些效果是纯粹的 HTML 无法实现。

不过，DHTML 的问题是，它完全缺少标准化。尽管已经进行了很多这方面的努力，但是每一家浏览器提供商都已经引入了自己特定的功能作为其产品的附加值，因此，你常常会在应用入口页面的角落里看到一个小图片，上面写着“请用 Internet Explorer 5.5 浏览本站点”或类似的文字。除此之外（如果这些麻烦还不够多的话），还要平滑支持一些较老的浏览器，这也是一项困难的工作。

当然，多年以来已经出现了许多针对服务器端的支持技术，使得 DHTML 使用起来更简单。例如，JSP 标记库就是其中之一。DHTML 的真正问题在于，许多用户基于某种原因（许多

组织因为担心其中的安全漏洞，非常讨厌这种技术），并不喜欢客户端脚本。所以，万维网上的许多浏览器都关闭了客户端脚本功能，这实质上也就关闭了你精心打造的 DHTML 用户界面。哎呀！

Macromedia Flash

Flash 也许是当今最流行的富客户技术。尽管它通常用于被动接收基于动画的多媒体片断，但是 Flash 也能发出 HTTP 请求，并对响应进行解析。这表明，使用 HTTP 与服务器（通常是一个 Java servlet）进行通信的基于 Flash 的客户端，不仅可行而且实用。要正常工作，用户机器上必须安装 Flash 播放器的浏览器插件，基于 Flash 的方法确实受到这个问题的影响，不过有鉴于 Flash 的流行，我们可以很安全地假定许多用户已经安装了相应软件。对于那些还没有安装 Flash 播放器的用户，Flash 的网站上提供了能够自动进行安装的下载链接，只要一次单击，就可以开始安装。

虽然有这许多优点，Flash 还是因被传为难于编程而名声不佳。许多机构已经对雇佣 Flash 程序员持保留态度，他们的出发点是因为，这种技能更倾向于为图形设计师而不是专业的 J2EE 程序员所具备，这样会造成团队在两种语言之间分化。Macromedia 公司已经通过加入对 XML 更好的支持，致力于把 Flash 扩展到企业领域，结果是一些“Flash 到 Web 服务”的后端应用已经开始出现。此外，Macromedia 公司正在推广 Flex 技术，它是一种更加适合开发者构造 Flash 应用的技术；至于这种技术是否能够成功，还有待长期观察。

Flash 编程困难这个问题还是可以解决的；然而，它还有其它与 DHTML 类似的问题。因为 Web 浏览器一开始并没有安装 Flash 播放器，用户在首次访问你的网站的时候（假设他们尚未安装），就得安装 Flash 插件，对于某些用户（例如，考虑一下你的祖父）来说，这并不简单。还有与前面类似的问题，许多大型机构处于安全问题的考虑，会关闭用户浏览器中安装的任何插件。尽管浏览器上通常已经安装了 Flash 插件，但如果你的目标是在互联网上获得尽可能广泛的用户群体的话，这就是一个值得考虑的问题。此外，你还要当心安装 Flash 插件对浏览器的最低要求，以确保能尽可能覆盖到你的用户群体。

Applets

Java 中的 applet 对前面提到的问题也提出了解决方案，但是它也带来了自己的问题。首先，applet 依赖于浏览器中的 JVM，Microsoft 和 Netscape 在这方面都不太好，它们的浏览器都不支持 JDK 1.1 以上的版本，并且在考虑工业品质的企业应用的时候，Java 中的 AWT UI 程序库过于简陋。作为一种解决办法，Sun 为两种浏览器都开发了 Java 插件，但问题是，除了插件技术在那些依赖于它的项目中名声不佳之外，其它的浏览器仍旧得不到支持。此外，applet 的安全模型过于严格，即使是执行写入本地文件系统这种最简单的操作，也需要数字签名才行。更糟糕的，用户每次访问页面的时候都要下载 applet，这就意味着大型 UI 应用要面临显著的下载负担。

不过，applet 的最大问题在于，如果所有浏览器都安装了 1.1 版本的 JDK，但是没有 Swing 的支持，那么使用 AWT 1.1 来开发用户界面绝对是令人难以置信的苦差事。理论上说，你能够从 rt.jar 里面提取出 Swing 的 class 文件，并把它们置于 applet 的.jar 文件中，不过你可能将面对几兆字节的编译后的代码，并且，这些都要在你每次访问页面的时候进行下载。当然，这里有个假设，就是 Swing 对 JDK 1.2，1.3，或者 1.4（或者即将发布的 1.5）中没有对其它类形成依赖。我们离 JDK 1.1 越远，这就越不可能会发生这种情况。

除了使用比较广泛以外，applet 基本上已是穷途末路的 UI 技术，我建议你在准备使用 applet 之前，先考虑一下 JNLP 和 Java Web Start (参考以下小节)。Applet 已经死亡，请让它安息吧。

URLClassLoader 类

除了以上选择之外，总还存在着“自己动手”的解决方案。java.net.URLClassLoader 类允许通过 HTTP（或者 FTP，或者任何 URLProtocolHandler 能够识别的协议）链接载入代码；因此，用户可以安装一个简单的 Java 客户程序，由它从服务器上取回“真正的”应用：

```
public class RemoteLauncher
{
```

```
  public static void main(String[] args)
    throws Exception
  {
    // Construct the ClassLoader to pull the code across
    //
    URL[] urls = new URL[]
    {
      new URL("http://intranetserver/DeployedApps/HRApp.jar")
    };
    URLClassLoader urlcl = new URLClassLoader(urls);

    // Pull the main class across and launch it
    //
    Class mainClass =
      Class.forName("com.javageeks.HRApp.Main", true, urlcl);
    Method mainMethod =
      mainClass.getMethod("main",
                          new Class[] { args.getClass() } );
    mainMethod.invoke(null, new Object[] { args } );
  }
}
```

我们还可以做许多事情，以使这段代码更加通用——允许把名称通过命令行参数传入，为 URLClassLoader 动态地构造 URL 对象数组等等。不过这段代码已经展示了主要的原理。

尽管很简单，但是这种方法的主要问题是，每次执行应用都必须从网络上下载所有的 class，这就意味着此方法只能适用于具有高速宽带连接的客户。在很大程度上，只有位于企业防火墙之后的在线客户才能达到这个要求。

JNLP 和 Java Web Start

JNLP 是一个 Java 规范，适用于通过 HTTP 请求向用户机器传送 Java 客户端代码。在许多方面，它实质上与 URLClassLoader 例子很类似，不过还有额外的好处：一旦传送完毕，代码就被存储于本地硬盘驱动器。从那以后，只有在更新的时候才需要从网络下载。JNLP 提供了在用户机器上下载并安装多版本 JDK 的能力，并且支持并行安装，即 JDK 1.3、1.3.1 和 1.4 可以共存而不会冲突。作为 J2SE 的一部分，Sun 公司提供了一个 JNLP 实现，称为 Java Web Start。与 applet 相比，JNLP 支持更精巧的安全模型，尽管在它的最初修订版中，只支持三种安全模型（能力依次增加）：applet、J2EE 客户端和应用。

典型的 JNLP 应用是基于 Swing 的客户程序，这样就可以使用 Swing API 提供的所有丰富且强大的功能，包括工具条、菜单条、主题支持、可切换外观等等。JNLP 还提供了许多访问底层操作系统的能力（比如访问剪贴板），而不需要数字签名。JNLP 还被考虑作为候选者，在将来集成到 J2EE 中，所以它并不像你可能认为的那样，与 J2EE 毫无关系。

与编写标准 Java 应用相比，编写基于 JNLP 的应用并没有太大区别；先以通常方式，编写你的用户界面，通常是一个基于 Swing 的应用。一旦完成了 Swing 用户界面，把.jar 文件复制到可以通过 HTTP 访问的位置，这样用户就可以从浏览器通过 HTTP 请求访问到它。

这时，这种开发工作基本上就完成了；最后一步就是编写一个.jnlp 文件，它是一个 XML 文件，里面包含了应用的有关信息，供 Java Web Start 使用：构成应用的.jar 文件的位置、应用需要何种安全机制等等。尽管这里不太适合讨论 JNLP 规范的繁琐细节，不过一个简单的 JNLP 描述文件就像这样：

```
<?xml version="2.0" encoding="utf-8"?>
<jnlp spec="1.0" version = "1.1"
  codebase="http://www.neward.net/ted/samples/"
  href="http://www.neward.net/ted/samples/Hello.jnlp">
  <information>
  <title>Hello World</title>
```

```
    <vendor>None</vendor>
    <offline-allowed/>
    </information>
    <resources>
    <j2se version="1.2+"/>
    <jar href="helloworld.jar"/>
    </resources>
    <application-desc main-class="Main">
      <argument>Hello</argument>
      <argument>World</argument>
    </application-desc>
  </jnlp>
```

把这个.jnlp 文件放到能够通过 HTTP 访问到的位置，这样用户就可以从浏览器直接访问到.jnlp 文件。

现在，当用户希望使用应用的时候，只要使用浏览器访问.jnlp 文件，如果用户机器上已经安装了 Java Web Start，那么 Web Start 浏览器插件（以 x-jnlp MIME 类型注册）就会弹出，它将对下载的.jnlp 文件进行解析，然后根据这些信息来下载.jar 文件、需要的 JDK 版本，以及支持库等等。

JNLP 规范提供了大量的 API，使得对客户机器的访问更加安全可靠，比如对于剪贴板或者少量的磁盘存储空间，就不需要对下载到客户机器的.jar 文件进行数字签名。详细信息请参考 JNLP 规范。

不过，JNLP并非完美的方案。（唉，金无足赤，人无完人。）与Flash类似，要使得一切工作正常，我们又要遇到在客户端进行安装的问题；这里，Java Web Start（或者等价的JNLP客户端）客户端需要进行安装，并与浏览器集成。这样它才能识别application/jnlp MIME类型，并开始必要的JNLP下载操作。然而，与Flash不同的是，Web Start并不流行，除非Sun能变出

某种魔术，与微软达成交易，把Web Start作为标准Windows初始安装的一部分，[1]否则它很难在短时间内流行起来。与Flash相同的是，对于已经略懂计算机的用户，Web Start客户端的安装并不困难，但是对于你的一般用户而言，他们可能刚刚才弄清楚浏览器和互联网的区别，可能就不会那么简单了。

全部采用富客户端技术？

再次重申一下，本项的关键，不是建议富客户技术适用于所有场合，而是建议对于某些以 Web 为中心的应用，富客户技术是恰当的。在许多方面，HTML 是 Web 中众多 UI 技术的“最小公分母”。但是对于某些应用，尤其是那些部署后供特定人员使用的应用（比如部署在企业内部网的应用），也没有理由受到 HTML 的限制。

实际上，JNLP 可能在公司内部的 intranet 环境中工作得最好，而 Flash 可能更适用于互联网范围的应用。URLClassLoader 方法也许最好作为“底层技术”，来快速替换更大的服务端（可能是集群）系统中的构件，一般的 J2EE 应用可能不需要这种技术，当然，Applet 技术已经过时，可以不予考虑。

然而，与此同时，一些商业产品也在涉及这个领域，考察一下它们是否符合你的特殊需求也很有价值。Droplets、Thinlets、Curl、Sash，以及其它产品已经面世，就在开发者还在考虑究竟应该过渡到何种 UI 技术的同时，肯定还会出现更多的技术。重要的是要搞清楚，这并不是“采用瘦客户还是胖客户”这样的二选一问题，而是找到最符合项目需求的结合方式。

第 52 项：使 HTML 短小精悍

与标准的 GUI 应用相比，Web 应用在很多方面都与之不同，最明显的事实是，所有东西都要从服务器传输到客户端。这不仅包括要显示的数据，还包括显示数据所需要的格式代码。这与旧式大型机上基于终端的应用很相似；从 HTTP 服务器返回的响应中，不仅包括了与用

[1]就目前来看，两家公司很可能已经解决了法律方面的分歧，但是这仍旧需要若干严肃的谈判。

户请求在逻辑上匹配的数据，还包括了用于表示数据的 HTML 元素，以及用来在浏览器内提供特定功能的某种客户端脚本。

然而，Web 的实际情况是，页面下载速度非常慢，即使用上了高速宽带或线缆调制解调器（Cable Modem）也常常如此。我们都遇到过这种情况：点击了一个超链接或者提交了一个表单，而其响应速度却非常非常的慢。与此同时，用户开始琢磨他今后是不是还应该来访问这个 Web 站点或 Web 应用。对于部署在企业内部使用的内联网应用来说，把用户羁绊于响应迟缓的 Web 应用很快就会使你在公司里声名狼藉。对于在全球使用的互联网应用来说，这会令你更快地确定，用户永远都不会回来了，这将导致公司的收入降低，进而导致开发者收入锐减。请记住，对 UI 技术的研究表明，用户在放弃等待，然后访问其它站点以前，平均会等待 5 秒钟。

那么，究竟是什么原因，使得精心设计，功能正确的应用慢得像蜗牛爬呢？很大一部分原因是由于返回的 HTML 页面。当 HTML 页面包含了大量对图像引用的时候，这些大大小小的图像分散在整个页面中，浏览器不得不一次又一次地访问服务器，以下载这些图像，这就使得整个页面的载入看起来非常缓慢。对于 Web 站点上的主菜单，真的需要那种鼠标移动到上面图像就会进行切换的图形按钮吗？页脚的版权申明真的需要用常春藤的叶子来环绕吗？或者公司标志真的需要在每个页面的左上角都出现吗？首先我必须承认，这使得页面看起来很漂亮，不过，用户看过这些内容之后，就会把它们抛之脑后。而糟糕的是，这些内容仍然被显示，这也就意味着需要重复下载。（优秀的浏览器会尽可能地进行缓存，但是对能够进行缓存的内容还是有限制的，尤其是图像本身是动态产生的时候。）即使不考虑这些，我们也要考虑一下图像本身大小的问题，如果图像尺寸稍微大一点，颜色稍微多一点，它们的大小就是以几百 K 字节计算的，所有这些内容都要跨越网络，从服务器传输到客户端。

不过，这比仅仅带来终端用户的延迟更糟糕。所有这些内容都要从服务器移动到客户端，这就意味着二者之间的管道被用来传输这些不必要的内容。这就导致了其它客户可用带宽的降低，也就降低了应用整体上的可扩展性。遗憾的是，这也属于不能“通过添加更多的服务器硬件来解决问题”的领域之一，进入建筑（或数据中心）的管道是瓶颈所在，要进行扩展将非常昂贵。

为了防止不佳的 HTML 破坏你的应用，请牢记一些处理 HTML 的技巧。

- *尽量少用“重量级”标记*。APPLET、OBJECT 和 IMG 标记都不能提供浏览器所需的所有内容，还需要和服务器进行另一次 HTTP 传输，这就意味着这段时间内用户界面将保持“停顿”，从这种角度看，它们都属于“重量级”标记。许多浏览器能够继续解析和处理页面的后续内容，但是会留下一大块丑陋的空白区域，用来放置图片、applet，或其它内容。这就是令用户“觉得”慢的部分原因。
- *使用框架（frame）来划分页面的不同部分*。如果 Web 应用有一个在上方显示的主菜单，可以把它放到一个单独的（无边）框架中，这样菜单就只需要获取一次。在页面下方的版权标志，也可以这样处理。
- *尽可能重用页面上的图像*。因为浏览器（包括代理服务器和防火墙这样的中继处理节点）倾向于对网站上下载的数据进行缓存，通过在不同页面中重用相同的图像，就提高了页面的载入性能。这使得浏览器能够使用已缓存的图像，而不是从 Web 站点重新下载。不过，对图像的引用必须完全匹配，所以，为了在不同页面中统一图像的 URL，你要确保所有图像都来自于服务器上的某个常见目录中。
- *使用 HTML 的特性，而不是图像*。HTML 标记提供了丰富的功能，要显示它们也无需额外的下载。例如，与其把超链接放进图像，来实现图像“按钮”的功能，还不如在标准文本上用不同的背景颜色，前景颜色，字体进行修饰，也能得到类似的效果。这样就节省了到服务器上取得图像的时间（请参阅第 17 项）。
- *遵循高雅、实用的用户界面设计原则，避免过多的页面导航*（或者换句话说，避免网络传输，请参阅第 17 项）。如果在基于 Web 的应用中采用向导风格的界面，那么假如向导中的每一步都间隔了 10 秒种，以等待服务器对下一步进行响应，这种方案很快就会崩溃。可以试着把几个步骤置于一个页面中，可能的话，就完全抛弃向导风格的界面。

针对基于 HTML 的应用，更多有效的设计建议可以参考《*The Design of Sites*》[Van Duyne/Landay/Hong]。

尽管我们在讨论 HTML 的问题，但是还是让我们先做点别的。现在举起你的右手，把你的左手放在本书的封面上，然后跟着我念：

“我，<这里接你的名字>，以戒咖啡所带来痛苦的名义，在此真诚宣誓，从我的 Web 应用所返回的所有 HTML 输出，都将与 XHTML 兼容，它们格式良好并符合标

准。我所有的标记都用小写，属性将使用完整的名称－值格式，值将被引用，并且所有的标记要么使用标准的结束标记，要么以短格式书写。我将不会使用没有被 XHTML 规范标准化的标记，并且我将永远不会认为使用 BLINK 标记是个好主意，无论这个想法在清晨四点经历了长时间的调试工作后看起来是多么的恰当。”

这并不困难，是吧？XHTML 1.0 只不过是把 HTML 4.01 标准以 XML 方式表示，这就从根本上确保了与 XHTML 兼容的页面，可以被 XML 解析器成功解析。为什么如此重要？因为如果 servlet/JSP 的输出可以被 XML 解析器接受，那么它也可以作为 XSLT 转换过程的输入，这样就可以在必要的时候，把 XHTML 转换成某种不兼容的格式，供使用较老的浏览器的用户使用。如果你的输出不是 XHTML（也就是说，不是格式正确的 XML），那么这种转换就不可以利用 XSLT 进行，你也就前景堪忧了，你将不得不使用 HTML 解析器来手工编写一个适配层代码。浪费的时间将无法想象。

HTML 以一种相对简单的方式，为描述表示层提供了一种独立于机器的方法，从这种角度看，它令人赞叹。不过请记住，每个页面将不得不通过网络下载给用户。根据定义，这是一次网络往返调用，所以要确保每一次跨越网络到服务器的访问合理且必要。这表明，在设计用户界面的时候要采用不同的策略，要抛弃传统的 GUI 方法，而是采用更加基于终端的方式。要确保你的 HTML 不是正在变成一个 GUI 应用，这通常是优秀的 HTML 应用正在走向失败的前兆，在考虑其它表示层技术的时候（请参阅第 51 项），这也是一个清楚的标志。

第 53 项：表示与处理相分离

考虑以下（简化过的）JSP 页面：

```
<%@ page language="Java" %>
<%@ page import="java.sql.*, javax.sql.*" %>

<html>
<head><title>Product Search Results</title></head>

<%
```

```
Connection conn = application.getAttribute("dbconn");
  // Assume JDBC Connection was stored in ServletContext
String SQL = "SELECT name, sku, stock FROM products " +
           "WHERE name = ? AND stock > 0";
PreparedStatement stmt = conn.prepareStatement(SQL);
stmt.setString(1, request.getParameter("name"));
ResultSet rs = stmt.executeQuery();
ArrayList results = new ArrayList();
while (rs.next())
{
  HashMap hm = new HashMap();

  hm.add("name", rs.getString(1);
  hm.add("sku", rs.getString(2);
  hm.add("stock", rs.getString(3);

  results.add(hm);
}
stmt.close();  // Aggressively release resources; see Item 67
%>

<body>
<h1>Product Search Results</h1>
<%
  if (results.size() == 0)
  {
%>
<b>No items found that match your query; try again?</b>
<%
  }
  else
  {
%>
<table>
  <tr>
    <th>Product Name</th>
    <th>Product SKU</th>
    <th>Stock Count</th>
  </tr>
<%
    for (Iterator iter = results.iterator(); iter.hasNext(); )
    {
      Map current = (Map)iter.next();
%>
```

```
    <tr>
      <td><%= current.get("name") %></td>
      <td><%= current.get("sku") %></td>
      <td><%= current.get("stock") %></td>
    </tr>
  <%
     }
  %>
  </table>
  <%
    }
  %>
  </body>

  </html>
```

看出问题了吗？没有的话，也别担心。许多 JSP 页面就是这么编写的，并且这种编程风格似乎很流行，尤其是在许多有关 JSP 的文章和书籍中。在某些情况下，这样确实可以简化范例。不过问题在于，当项目的交付日期很紧的时候，读者未必能认识到（或者关心）这个问题，这就导致了此种风格的代码混进产品代码中。

这里的危险之处很隐蔽，业务规则（此处是指，搜索请求只能针对库存里存在的产品）混进了 JSP 页面之中。这样一来，业务规则就会与“用于显示结果的代码”出现在同一层次之中，这种混和很危险，通常应该进行隔离。

当看到这类代码的时候，大多数 J2EE 架构师和专家的第一反应是建议，通过把搜索代码提取出来放进一个会话 bean，这样就把整个流程放入了业务层中。通常情况下，无状态会话 bean 是首选。但是当搜索结果很多，一个屏幕显示不完时，使用有状态会话 bean 也很方便。这时，与其再次访问数据库，取得下一块搜索结果，还不如就把结果放进一个有状态会话 bean，这样取得余下的搜索结果就只需要一次网络来回（请参阅第 17 项）。

不过，把这些代码放进会话 bean（有无状态都一样）会有一种严重的副作用。在 JSP 页面和真正的搜索代码之间加入了网络传输之后，会显著增加页面的延迟。更重要的是，如果使用了有状态会话 bean，我们不得不在每个用户状态内保持一个会话 bean 实例的引用，也就是要用到 HttpSession（请参阅第 39 项）。这样就会导致服务器负载的增加，即使 Web 应用

的其它部分并不需要的时候，也会需要使用会话（session）。此外，对于会话 bean 里 SQL 语句执行时刻的事务属性（请参阅第 31 项），我们也得做出明智的决定。一般情况下，因为不会经常修改数据，我们可以在较低的隔离级别下执行查询语句，不过请记住，EJB 不允许变更查询的隔离级别，所以，我们也许需要在一个事务下运行，从而也就会导致竞争（请参阅第 29 项）。

这时，通常的作法是，完全绕开 EJB 层，直接在数据库上执行查询，这样我们就能够控制事务的隔离级别。这种方式被称作快车道模式（Fast Lane pattern）。不过，这样就把我们带回到了原来的问题中：又要把业务规则混进代码中了。正如第 3 项所述，尽管我们希望绕过 EJB 层，以避免网络传输和减少竞争，但是我们也没有必要把业务规则代码直接置于表示层代码中。

现在你肯定会问，为什么这样做不好。毕竟，在许多情况下，把这种代码放进某种 Model 或者 Bean 类中，并不见得能够简化真正执行的 SQL 语句。这样只不过多加了一层方法调用，也就降低了页面整体性能，这样很好吗？一些人可能会说，这样对隐藏数据库模型的细节有帮助，它们应该位于页面之后。不过请现实一点，在应用的整个生命周期中，数据库模型本身会经常发生变化吗？

事实是，经常发生变化。所以，数据库模型常常需要改变，尤其是在项目从一个阶段到达下一阶段的时候。新的需求通常至少会使得对存储模型进行部分修改：扩充表以包含新的信息，拆分一个表以更好地进行组织，或者进行范化（或者为了提高性能，进行反范化）。在一些情况下，还会修改表/列的名称，以更恰当得表示存储的数据。

不过更重要的是，直接把业务规则嵌入表示层代码，通常会导致违反所谓的“一次且仅有一次”规则（Once- and-Only-Once Rule）：特定的功能代码在代码库中应仅出现一次。这时，要是进行其它类型的产品搜索（比如与有关客户结合起来进行搜索），就会违反这个规则。用 SQL 语句中的“stock > 0”这个谓词所表示的，“只针对库存里存在的产品进行搜索”这条业务规则，将不得不手工复制到搜索代码中。但是过了一段时间以后，要是决定改变业务规则，搜索包括库存中没有的产品，那么就必须在所有地方找到并修改这个 SQL 语句。当然，要是数据库模型发生了变化，比如把库存清单存储到了别的表中，也会发生同样的问题。

这时，所有实现这条业务规则的查询语句都需要更新，以反映新的数据库模型。

那么这样的结果就是，编写一个代码层（请参阅第 3 项），使得表示层代码与执行产品搜索的真正细节相隔离；当在 JSP 页面中使用的时候，这个层次通常是位于标记库里的一个标记处理程序，不过，通常仅仅使用静态方法来封装查询的 Java 类，也能够工作地很好：

```
public class Queries
{
  public static List productSearchQuery(String pname)
  {
    // Same JDBC logic as before
  }
}
```

从 JSP 页面里使用这个类，就像这样：

```
<% page language="Java" %>

<html>
<head><title>Product Search Results</title></head>

<%
  List results =
    Queries.productSearchQuery(request.getParameter("name");
%>

<body>
<h1>Product Search Results</h1>
<%
  if (results.size() == 0)
```

```
  {
%>
<b>No items found that match your query; try again?</b>
<%
  }
  else
  {
%>
<table>
  <tr>
    <th>Product Name</th>
    <th>Product SKU</th>
    <th>Stock Count</th>
  </tr>
<%
    for (Iterator iter = results.iterator(); iter.hasNext(); )
    {
      Map current = (Map)iter.next();
%>
  <tr>
    <td><%= current.get("name") %></td>
    <td><%= current.get("sku") %></td>
    <td><%= current.get("stock") %></td>
  </tr>
<%
    }
%>
</table>
<%
  }
```

```
%>
</body>

</html>
```

把表示层代码和处理逻辑相分离带来的额外好处，在于为并行开发开来了机会。编写一个隔离的层次，表示层设计者就不用关心查询的真正细节，这样他就能够使用可以传回模拟数据的模拟对象，对表示层进行测试。这样就能够帮助减轻项目进度的压力，用户可以尽可能快地看到表示层，他们就可以明确将得到的产品。这同时意味着，当 JSP 页面使用标记库来提供这种层次的时候，表示层设计者不需要懂得程序设计，而只需精通图形设计和布局，就能够确保你的 Web 应用不会成为一堆设计拙劣的 Web 页面。

对于前面第二个 JSP 范例，尽管对查询进行了抽象，还是不够“干净”，或者说抽象不佳。页面上有许多代码，例如判断是否有数据行返回的代码块，更不用提那些 JSP 脚本代码了，它们大多用来在循环里处理查询返回的 ResultSet 数据集。此外，查询本身也在 JSP 页面里进行处理，有人也会指出不应这么做，因为 JSP 的主要目标是使得非程序员（比如熟悉 HTML 或者简单脚本语言的页面设计者和绘图师）能够具有编写动态页面的能力。所以，应该通过传递参数，将查询置于一个 servlet 中，然后转发到此 JSP，以显示结果：这样，程序员来编写 servlet，页面设计者来编写 JSP，然后我们就可以把关注点进行较好的分离。

这个想法变得非常流行，Sun公司将它命名为Model 2 架构，并加入到JSP 0.92 规范中。（这个比较原始的名称来源于这样的事实：在servlets和JSP的三种推荐使用的惯用法中，它是第二种。）因为Model 2 这个词描述能力有限，所以它已经被订正为模型－视图－控制器架构，这在许多文献里都有详尽的描述。实际上，位于Jakarta Apache Web站点（http://jakarta.apache.org）的开源程序库Struts，就完全是在Model 2 架构上丰富和发展起来的，它已经成为一个框架。还有一些项目自己定义了基于模板的表示层语言，而不是使用JSP。不管怎样，其核心概念总是一样：把表示层（JSP）与处理逻辑（servlets）和领域相关的对象（beans）相分离。

第 54 项：内容与样式相分离

乍一看，这一条似乎不过是第 53 项的重复。尽管它们在思路上有些共同点，但内容还是有些不同。在第 53 项中，我们考虑把表示层逻辑和处理逻辑相隔离，这里我们则试图把表示的样式与内容区分开来。不过，在我们深入讨论之前，我们需要先定义二者的不同。

内容是指，从某个请求返回的实际数据。这些数据没有任何显示机制或相关标记。例如，当考虑在 Web 网站上进行搜索的时候，内容就是搜索所返回的各种数据项（或者就是“未找到项目”这样的文字）。

另一方面，样式则关系到一些审美元素，我们使用样式来修饰内容，以在视觉上吸引用户。字体尺寸、背景/前景颜色、布局等都属于样式。一条基本经验是，我们从页面中移除样式，而不会影响页面的可用性；但是，我们无法在不影响可用性的前提下移除内容。搜索结果即使不加任何修饰，仅以逗号风格，仍旧是有用的；但是即使格式再漂亮的页面，没有内容（甚至连“未找到项目”也没有）的话，也是无用的。

“我们已经决定改变整个网站的外观，所以你的页面也得做出修改，以保持一致。这不会花多少时间，是吧？”对于那些曾经在 Web 应用发布前夕听到如此恐怖言论的人们来说，本节的价值就显而易见了。如果把页面的样式嵌入到应用的输出中，那么 Web 应用中的每一个 JSP 页面就得进行修改，以反映出新的外观。对于程序员来说，没有比这更浪费时间的了。幸好，对于基于 HTML 的应用，至少有两种技术可以解决这个问题。

第一种是层叠样式表（Cascading Style Sheet），更为人知的是其缩写，CSS。它提供了对 HTML 页面中元素的样式和布局进行控制的能力，通常在一个单独文件中集中定义整个站点的样式。颜色、字体、尺寸、甚至是确切位置和布局都可以通过 CSS 元素进行控制。CSS 还支持样式级别。比如，它既可以为页面上的所有段落元素（P）定义样式，也可以对“通过带名称的样式属性来定义的”段落元素提供不同的样式。作为万维网联盟的标准，CSS 也获益匪浅，绝大多数 HTML 浏览器都对其提供广泛支持。

然而，不同的浏览器实现（甚至不同的版本），对 CSS 的支持也参差不齐；一个主要依赖

CSS 来定义样式的站点，可能在某个浏览器版本下工作正常，但是在别的或者较老的浏览器中，就可能完全不同。对于部署在企业内部网中的企业级应用来说，这里通常能集中管理浏览器的版本，这个问题还容易解决；但是对于部署在公共互联网上的应用来说，这里的用户可能还在使用 Internet Explorer 1.0 和 Netscape Navigator 2.0 这样的浏览器，这个问题就很难处理了。

XML 的 CSS 在逻辑上的继承者是 XSL:Transformations，或者简称 XSLT，它提供了另一种可能的解决方案。XSLT 几乎可以把任意 XML 数据源转换成任何输出，包括 HTML。这表明，如果某个 JSP 的输出是一个格式良好的 XML 数据集，我们就可以通过 XSLT 把它转换成任何可能的 HTML 页面。

使用 XSLT 的方式好处很多。首先，由于不同浏览器所支持的 HTML 花样繁多，要想编写出在所有的浏览器上都能够正常显示的 HTML，其实非常困难。所以还不如在所有 JSP 页面之前放置一个过滤器，用它来检查 HTTP 请求报头中的 User-Agent，以判断发出请求的浏览器类型，然后使用相应的 XSLT 页面，把 JSP 输出转换成特定于用户浏览器版本的 HTML 文件。对于避免直接在 JSP 页面中对这些差异进行编码，这么做会很有帮助。而且，被呈现的 HTML 的样式可以在 XSLT 文件中被直接捕获，这样就把所有有关样式的决定都放在了一个逻辑位置上。

无论你选择何种方式，目的都一样：把所有有关样式的决定局限于单一位置，也就是遵守“一次且仅有一次”规则（请参阅第 53 项）。这样，当市场部的副总裁来到你面前，告诉你要在整个企业范围内改变站点的外观时，这种对 Web 应用进行更新以反映新变化的过程，只需以天或小时计算，而不是以周或月来计算。实际上，如果站点本身使用了 CSS，你直接重用新的 CSS 文件就可以了。

除此以外，现在可以使得从 JSP（或者 servlet）中生成的真正输出保持最小，它们只需完全关注返回的内容，而把进行修饰的工作交给 CSS 或 XSLT 来完成。这表明，现在 JSP 或 servlet 发出响应将更加简单，调试和修改输出也更方便。并且，在使用 CSS（输出的 HTML 将在用户浏览器中绘制）的时候，发出响应所需的带宽也更小，这样就不仅降低了服务器和网络的并发负载，而且减少了页面载入的总体延迟。

除此以外，我们还可以得到意料之外的好处。通过把“对样式的决定”与“产生JSP/servlet输出的过程”相分离，我们能够有效地使开发者避免决定页面样式的窘境。程序员经常因为设计出蹩脚的用户界面而名声不佳[2]，这多数是因为，与普通计算机用户相比，程序员看待世界，并与之打交道的方式非常不同。通过把有关颜色、字体、背景、有时甚至是布局方面的决定，交给受到过更好训练的人来做，程序员就能集中精力于产生数据响应。

如果 servlet 或 JSP 的响应中只含有数据，并由这些数据产生页面的样式元素，这时使用 XSLT 来产生需要的 HTML，还可以得到额外的好处。如果当应用需要与其它非 Java 系统（比如.NET）互操作的时候，使系统过渡到支持 Web 服务就变得非常简单。现在，就不必编写两个单独的 servlet（一个用来产生 HTML 输出，一个用来产生 SOAP 响应），而是可以使用能够产生通用响应的 servlet，然后用不同的 XSLT 样式文件来转换成所需的 HTML 或 SOAP 输出。这种转换可以在过滤器中进行，建立不同的 URL 模式来指向同一个 servlet，这样就能够从两个不同的终端访问同一个 servlet。Web 应用的部署描述符看起来像这样：

```
<web-app>
  <servlet>
    <servlet-name>ProcessRequest</servlet-name>
    <servlet-class>
      com.javageeks.webapp.ProcessRequestServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ProcessRequest</servlet-name>
    <url-pattern>/ProcessRequest</url-pattern>
  </servlet-mapping>
```

[2] 当然，我这里有点偏颇。有些程序员确实意识到了要设计出优秀的用户界面。不过除非你经过专门训练，否则你最好还是把自己归于“蹩脚的用户界面设计者”那一类。因为在这方面，依赖直觉通常会导致蹩脚的设计。不信的话，有关此主题的更多资料请参考Alan Cooper的《The Inmates Are Running the Asylum》[Cooper99]

```
    <servlet-mapping>
      <servlet-name>ProcessRequest</servlet-name>
      <url-pattern>/WebService/ProcessRequest</url-pattern>
    </servlet-mapping>

    . . .

</web-app>
```

现在，要对产生的输出做任何修改，只需要进行一次。如果要做出结构上的修改（需要返回更多或者不同的数据），只要修改一个 servlet；如果要做出视觉方面的改变（改变修饰 HTML 的样式，或者适应较新版本的 SOAP 协议），可以修改过滤器和/或进行转换工作的 XSLT 文件。在这两种情况下，受到影响的代码都被局限于单独的位置，这就使维护起来更加方便。

不过请注意，这些都不是免费午餐，基于 XSLT 的方法也不例外。在这种情况下，我们增加了请求的延迟，因为执行转换过程需要时间。对于 CPU 负载较高的站点，可能更希望把不同的 JSP 页面（针对不同浏览器版本）都预先用 XSLT 进行处理（请参阅第 55 项），然后在控制器 servlet 进行转发的时候，进行适当的选择。不过，这将大大增加网站上 JSP 页面的数量，可能会为修改/调试带来潜在的麻烦。只有在延迟完全不可接受，并且其它性能提升技术都无法解决问题的时候，才能把这种方法作为最终手段。

第 55 项：预生成内容以最小化处理过程

尽管某些类型的应用需要完全动态生成，不过还是有一大类应用，并不需要完全在运行时刻产生输出。例如，考虑一下 Amazon.com，大体来说，整个 Web 站点在多数情况下以读取内容为主，对于网站上的给定商品，99%的用户只是不断浏览几乎相同的内容。对于 Amazon.com 库存中的任意商品，尽管我们能够编写一个单独的 ViewItem.jsp 页面来显示详细信息（出版社、价格、广告文案、试阅章节以及读者评论等），但这样会给服务器带来严重负担，因此就降低了站点的总体性能和可伸缩性。

尽管 Amazon.com 库存中的商品数量非常庞大，但是这些数据并不经常发生变化。当然，价格会浮动，读者评论也会添加或删除，也可能加入新的广告文案等内容，但是这种变化的间隔是以周为单位，而不是以分钟或秒来计算的。绝大多数读者并不会真的在乎 Amazon 的销售排行榜是否能够精确到秒（尽管某些书籍的作者可能会关心这个）。

不过，如果页面完全是动态产生的，每一次请求就至少需要一次对数据库的访问，以获取数据并动态插入到页面中（为了最小化延迟，我们可能会希望尽量减少这种访问的次数，请参阅第 17 项）。这样就不仅在页面等待数据返回的时候引入了延迟，而且，如果这些请求被盲目地以事务方式进行处理的话，就可能在数据库服务器上引起竞争（请参阅第 29 项）。更糟糕的是，因为数据并不经常发生变化，所有的事务开销以及数据库查询基本上毫无价值，每次返回的数据 99.9%与上次返回的完全相同。遗憾的是，数据库缓存在这里也不会有多大帮助；网站可能会同时支持几千个并发用户，所以除非数据库能够对几十万条查询进行缓存，否则缓存的命中率可以忽略不计。

对于每次请求，与其不断取回相同数据，还不如预生成一个静态页面，并把数据作为常量放进页面。实际的“预生成”动作可以隐藏在 Web 应用中，例如，可以用 servlet 在 Web 应用的目录里产生 JSP 页面。当需要修改的时候，用于编辑的 servlet 可以直接重新生成 JSP 页面或者外部资源来反映新变化。尽管这样可能会造成大量页面的产生，但是你通常可以把文件放进站点中的某个子目录下，以缓解这个问题。子目录可以使用商品的主键来标识（例如，在图书销售网站上，可以使用书籍的 ISBN 代码）。

请注意，这种方法只针对只读（或多数情况下只读）的表示层资源工作良好。例如，门户网站就属于这一类，因为大多数用户只会选择他们希望看到的那一部分，并且这种选择并不经常改变。所以，为每个用户预生成门户页面可以节省大量处理工作，当我们希望门户站点的主页能够尽可能快的时候，更是如此。

在某些情况下，我们可以创建一个缓存（通过一个能够访问内存缓冲区或其它临时存储设备的 servlet 过滤器）并把预生成的内容放进缓存。然后从缓存取得数据，而不是重新遍历整个页面的逻辑代码，来达到“预生成内容”和“每次都动态生成”二者之间的某种平衡。这

样就带来了我们所期望的好处：免去了为创建内容而进行的大量方法调用和必需的缓冲区（字符串）连接操作。不过这样也确实带来了一些问题，最明显的是，你必须确保缓存不会给系统总体的内存使用量带来太大负担，因为这将降低可扩展性。一般来说，如果你采用缓存方法，要确保只对站点上所有用户都能够用到的数据进行缓存；针对每个用户进行缓存的策略将严重导致对内存的大量需求。此外，当对已经生成的输出进行缓存的时候，你会遇到与第 4 项所描述的相同的更新传播问题，这是因为，现在你的缓存过滤器需要知道它所缓存的表示层的变化（或者说，需要发现这种变化，这就需要时间，并且会降低缓存的效率）。

假设你可以找到一种快速且简单的方法，来判断何时刷新缓存，并且假设这种方法能够被明智地应用于网站上的各种元素，对于静态的 GIF 或 JPG 图像，进行缓存并没有意义。一个缓存 servlet 过滤器也可以带来和“预生成内容”相同的好处。

预生成内容还有其它优点。如果我们能够预生成 HTML（而不是 JSP），它就变成了完全静态的资源。这时，从用户浏览器到 HTTP 服务器之间，所有能够进行缓存的节点都可以参与进来，以更进一步地降低请求的延迟。尽管未必所有页面都能够预生成为静态的 HTML 页面，你也要尽可能地预生成内容，尤其是针对 Web 应用的入口。这样就能够在用户初次使用程序的时候带来深刻的第一印象：“哇，真快。”

第 56 项：尽早验证，尽量验证

用户并非总能够提供你所需要的所有信息，这样假设合情合理。实际上，我敢保证，你的普通用户恰恰会在最关键的时刻忘记提供某些关键信息。对于电子商务应用，用户可能会忘记填写信用卡号码；对于门户网站，用户可能会忘记提供用户名称或者认证证书（密码或者是某种硬性编码的值）。更糟糕的是，虽然用户提供了数据，但却是错误的数据。比如拼写错误，对数据的曲解，或者仅仅是常见的“哎哟，我选错了。”

我们不得不验证用户提供的数据，这也是程序员日常生活中的现实情况，不过在基于 HTML 的应用中，我们就没有必要在验证发生失败的时候大惊小怪。

考虑一下，在阅读产品白皮书，下载产品试用版本，或者访问专门提供给公司目标客户的“内部”信息之前，通常需要用户进行注册。当提交注册表单的时候，需要发生什么呢？

首先，我们需要校验那些在用户以后进行访问的时候，能够帮助我们对用户进行识别的数据；表单上的某些字段为必填字段，这些字段不能为空，否则就会破坏我们对系统的基本假设。例如，我们可能需要用户的完整姓名（包括姓和名），以及登录名和密码，以供用户以后访问站点时使用。如果用户没有能够有效地提供这些数据，我们就需要请他/她重新输入。

第二，某些字段需要进行数据验证；例如，对于只针对北美地区客户的表单，电话号码一定是“NNN-NNN-NNNN”形式的，美国客户的邮政编码一定是美国邮政局的“5＋4”格式：NNNNN-NNNN。这种对数据的语法验证，能够确保用户提供的数据遵循特定的表达模式。以上两种字段（电话号码和邮政编码）如果格式不对，将被拒绝接受。

我们还可以要求提供的邮政编码必须与地址里出现的城市相匹配；甚至可以更进一步，对城市里是否存在这个街道进行验证（当然绝大多数站点不会进行如次深入的验证）；当然我们希望能够验证电子商务订单里提交的信用卡号码是否与客户的姓名相匹配；我们还可能希望利用信用卡公司提供的防止信用卡诈骗的某些最新措施：要求用户提供印刷在信用卡背面的“验证码”；或者我们希望能够验证用户选择的产品代码是否真的存在于产品目录中；或者用户针对某种产品的选择，对这种产品来说是否真的有意义（例如，根据某个人电脑制造商在线销售站点提供的情况，某种型号的个人电脑不能同时配置 DVD 和运动捕获卡）。这种针对数据语义的验证，当然和对数据语法的验证同样重要。

最困难的问题是：我们应该何时去验证所有这些内容呢？

答案是：越早越好，越全面越好。原因很简单，对于这种简单的数据项错误（数据的语法验证），只要能够及早地发现，就能够尽早地让用户进行更正。

你也看到了，也许会和我一样，发誓以后要避免犯这种错误：直到表单被提交到服务器之后，才真正对数据进行验证。单击一下“提交”按钮，然后必须等 5－10 秒钟，服务器才能收到数据，处理请求，并发出响应。然而映入眼帘的文字（通常以粗体红色表示），却是告诉你

忘记填写表单上某个特定字段。如果网站设计得很糟糕的话，原来添好的表单现在就消失了，单击“后退”按钮只会让你看到一个空表单，你不得不重新填写所有内容。更糟糕的是，原来的表单也没有告诉你哪一个字段是必填的，你就不得不：（a）填写表单上的所有字段，必要的话就胡乱拼凑一番；或者（b）不断进行试验，直到你碰巧输入了某种神秘组合，使你得以通过数据验证逻辑把守的大门。

更糟糕的，是那些在服务器上逐个字段进行验证的应用，一旦遇到了第一个错误，页面就会返回，这就迫使用户在更正错误以后，再次把页面发回到服务器，这样的结果可能仅仅是发现了下一个错误。

在绝大多数用户心目中，这些都是不可接受的。

主要问题在于，要想正确地对数据进行验证，其实既单调又笨拙，更不用说它有点违背程序员的本能了。例如，对于一个程序库的 API，某方法希望接受几个字符串作为参数，这些字符串既不能是 null 也不能是空字符串。这样编写代码是完全可以接受的：

```
public class NiftyLibrary
{
  public static String transmogrifyStrings(String[] source)
  {
    // Verify that none of the Strings are null
    for (int i=0; i<source.length; i++)
      if (source[i] == null)
        throw new IllegalArgumentException("No nulls!");

    // Verify that none of the Strings are empty
    for (int i=0; i<source.length; i++)
      if (source[i].equals(""))
        throw new IllegalArgumentException("No empties!");
```

```
      // Do the rest of the work
    }
  }
```

这里，通过在检测到错误的时候抛出异常，我们就能够支持“参数既不可以是 null 也不可以是空字符串”的前置条件。那么，为什么我们不可以在用户界面中支持类似的错误处理（数据验证）逻辑呢？

然而，这种作法不能类推到用户界面中。因为在UI层次上，尤其是对于HTML应用来说，每一个用户动作都需要一次携带请求信息的网络传输，而对于上面的程序库，这种动作完全位于JVM[3]内部。所以，根据第 17 项的建议，为了使得网络来回传输的次数最小化，我们需要充分挖掘每次网络传输的能力。

假设我们使用控制器 servlet（或者是 filter；或者是其它类似 MVC 中的代理对象，如第 53 项所述）来处理表单数据，那么所有的表单验证逻辑都将位于一个标准的 Java 类中；出于简化问题的考虑，我们假设使用 servlet。尽管很容易把 servlet 的验证代码写成与以上程序库类似的代码，但是在向用户返回错误信息之前，我们需要进行更全面的验证：

```
public class RegistrationServlet extends HttpServlet
{
  public void doPost(HttpServletRequest req,
                     HttpServletResponse resp)
    throws ServletException
  {
    // Capture all validation errors in one place
    //
    List validationErrors = new ArrayList();

    // Verify that required fields are non-null
```

[3] 不过，如果库函数在远程对象上调用的话，就又回到了通过网络传输来调用方法的问题上了，这时，基于同样的原因（避免网络来回的耗费），我们就可能希望对验证逻辑所处的位置作出调整。

```
//
if ( (request.getParameter("first_name") == null) ||
    (request.getParameter("first_name").equals("") )
  validationErrors.add("First Name must not be empty");

if ( (request.getParameter("last_name") == null) ||
    (request.getParameter("last_name").equals("") )
  validationErrors.add("Last Name must not be empty");

// . . . and so on . . .

// If there were errors, send the user back to the original
// form to reenter the data
//
ServletContext ctx = getServletContext();
if (validationErrors.size() > 0)
{
  request.putAttribute("validationErrorsList",
                  validationErrors);
  RequestDispatcher rd =
    ctx.getRequestDispatcher("/registration.jsp");
  rd.forward(request, response);
}
// Otherwise, pass them on to the goodies behind
// the registration page
//
else
{
  RequestDispatcher rd =
    ctx.getRequestDispatcher("/premium/index.jsp");
```

```
      rd.forward(request, response);
    }
  }
}
```

请注意，我们使用户回到原来那个包含表单的 JSP 页面；在此页面中，我们要确保验证错误信息位于表单上方，并确保把那些作为请求的一部分所提交的数据抽取出来，放回用户已经填写的字段中。（通常，浏览器自己可以缓存这些数据，不过我们为什么要冒这个险呢？）

```
<%@ page language="Java" %

<html> <!-- the usual head, title elements -->

<body>

<%
  List validationErrors =
    request.getAttribute("validationErrorsList");
  if (validationErrors != null)
  {
%>

<!-- Note that this is probably not the most human-friendly
     way to put this, depending on your target audience;
     consider getting UI advice before putting this into
     production.
  -->
<h2>There were validation errors; please correct the following
and submit the page again:</h2>

<ul>
<%
   for (Iterator iter = validationErrors.iterator();
       iter.hasNext(); )
   {
%>
<li><font color="red"><%= iter.next().toString() %></font></li>
<%
   }
%>
```

```
</ul>
<%
  }
%>

<%!--
  Probably should check for nulls in the request parameters,
  just to avoid the possibility of the literal string "null"
  showing up in the field values; this is left as an exercise
  --%>
<form method="POST" action="/servlet/RegistrationServlet">
First Name: <input name="first_name" type="text"
  value="<%= request.getParameter("first_name")%>" /> <br />
Last Name: <input name="last_name" type="text"
  value="<%= request.getParameter("last_name")%>" /> <br />

<!-- and so on -->

<input name="submit" type="submit" />
</form>
</body>

</html>
```

如你所见，提供这些功能并不是非常困难，只是有些单调乏味。尤其是当 JSP 页面设计者并非程序员的时候。幸好，这也是现在 JSP 支持标记库的原因：

```
<%@ page language="Java" %>
<%@ taglib uri="http://www.host.com/HTMLSupportLib"
          prefix="html" %>

<html> <!-- the usual head, title elements -->

<body>

<html:validationErrors>
<h2>There were validation errors; please correct the following and
submit the page again:</h2>
```

```
<ul>
<html:validationErrorList>
<li><html:validationErrorText /></li>
</html:validationErrorList>
</ul>
</html:validationErrors>

<form method="POST" action="/servlet/RegistrationServlet">
First Name: <html:textInput name="first_name" /> <br />
Last Name: <html:textInput name="last_name" /> <br />

<!-- and so on -->

<input name="submit" type="submit" />
</form>
</body>

</html>
```

validationErrors、validationErrorList 和 validationErrorText 都是自定义标记。它们依赖于之前的 servlet 把验证错误信息的列表命名为 validationErrors，并放进 HttpServletRequest。这与前面的例子相同，但是这样就简化了页面中的逻辑，使得 JSP 页面的设计者更容易处理。此外，textInput 标记能够对进入的 HttpServletRequest 进行检查，如果发现了与 name 属性相同的输入参数，它就会把此参数的值预先放置于相应的 HTML 字段中。这样一来，代码就没有那么乏味了。

但是我们还是要忍受之前相同的问题：我们对表单上的字段进行任何验证，都不得不回到服务器上才能进行。正如第 17 项说明的那样，如果要降低延迟并提高可扩展性，我们要使得消耗在网络传输上的时间最小。幸好，绝大多数 HTML 浏览器都提供了在服务器上进行验证之外的选择，即在客户端进行验证：脚本语言。Netscape 浏览器提供了 JavaScript（网景

公司对标准 ECMAScript 语言的实现），Internet Explorer 提供了 VBScript 和 Jscript（微软公司对标准 ECMAScript 语言的实现）。在客户端脚本语言环境中，尽管大多数对数据的语义检查是不可能的（有时仅仅是困难），不过绝大多数对数据的语法检查，在这些脚本语言的能力范围内还是可以全面进行的。不过，这里又将遇到同样的问题，把代码放置到每一个包含表单的页面中非常枯燥，同样，这也可以使用标记库来解决问题。例如，textInput 标记可以为其展开的<input>元素加入脚本支持，以确保在提交表单的时候（或者输入焦点离开字段的时候，或者在单击某个“验证”按钮的时候，或者其它可能的事件发生的时候），姓和名字段不为空。如果验证失败，就弹出一个对话框，并停止提交动作，这就使得在网络传输之前用户就能够改正错误。

我们将不得不以某种智能的方式来编写标记库，以把浏览器的版本作为一个因素来考虑，不过好在这种代码只需编写一次。更重要的是，正如第 51 项所述，我们不能假设总是可以得到脚本支持，所以我们仍旧需要在服务器端进行验证。客户端脚本可以使用户获益，而服务器端验证者则使程序员获益。可能的话，要尽量同时使用两种方式，尤其是可以把它们编码进JSP标记库中的时候。（位于http://jakarta.apache.org/taglibs的Jakarta Taglibs项目，已经有几种标记提供了这种功能；位于http://jakarta.apache.org/struts的Struts项目，也提供了类似的标记，它们与Struts的架构结合得更紧密。）

顺便提一句，不要依赖于把文字颜色设置为红色来表示发生了验证错误。要知道，世界上有许多人是红/绿色盲，他们并不能辨别文字是否为红色。如果没有某种支持信息的话，他们就可能对表单提交时发生的错误一筹莫展。并且，当我们讨论错误消息的时候，你要确保错误消息对于非开发人员也是易于理解的。你可以找一个非开发人员来检查一下，以确定错误信息是有意义的。

最后，请记住，进行客户端验证只有一个目的：在用户向系统输入数据的时候，使程序尽可能友好。你也不应该放弃在服务器端进行验证（无论如何，在阅读了第 61 项之后，都不应该放弃这么做），并且，为了避免客户机器和 servlet 容器之间在每一次数据项发生错误的时候都进行网络传输（请参阅第 17 项），要尽可能早地捕获到用户的错误。应该编写验证程序一次验证所有的用户输入，并且提供一个包含了所有需要改正的项目的列表，尽管这么做确实有点笨拙。不过请记住，在项目交付的时候，系统是供用户使用的，而不是你自己，所以

你最好确保它易于使用。用户友好的验证方式是实现这一目标的关键。

第七章 安全

君子安而不忘危，存而不忘亡，治而不忘乱。是以身安而国家可保也。

——孔子

它是站在屋子中心的大象，每个人都可以看到它，每个人都认识它，但每个人都拼命地试图对它视而不见，绕开它进行工作。我这里说的就是在程序员词典中“声名狼藉”的主题——安全。

在大部分 Java 程序员的理解中，安全的是非常神秘的，不可思议的，甚至是不可理解的。它的同义词就是部件管理、密码技术，和 FUD（fear, uncertainty, and doubt，恐惧、未知和疑惑）。你要是问 Java 程序员他们的应用是否安全，有人会自信并带点天真地说“绝对！”，而另一些人会耸耸肩膀，环顾四周，并试图使谈话内容回到更为保险的、更容易理解的主题上来，比如像脑外科手术般精细的任务、对象关系映射层，以及使用对象池带来的欢乐。

遗憾的是，这种态度无异于自欺欺人。企业级应用员也许比任何其他软件项目的开发人员更需要认同这一点，并意识到他们所面临的安全风险及如何降低它们。每天我们都听到很多关于公司信用卡号码从他们的数据库中被偷走，或是因为拒绝服务攻击损失上几百万，或是因为直接的诈骗和其他非法行为而损失更多的钱的事情。

使得整个状况更糟的是大部分开发人员对真正的应用安全是什么只有一个狭义的概念；在 Java 社区，提到“安全”这个词会引起两种反应：要么是“Java 安全性——那是 applet 沙盒该负责的事——不是吗？”，要么是“好，让我们先取两个非常大的质数……”

安全是一个庞大，复杂而令人兴奋的主题，它远不止 applet 砂盒和不对称密钥加密那么简单。即使整本书花上几百页来关注一个安全方面都无法涵盖它的所有内容，这里我们要感谢攻击者和安全专家之间不断升级的军备竞赛，今天那些被证明正确的东西将成为明天的宝贵遗产。关注于企业级 Java 开发的一章内容不可能对安全有太深的涉及。而我的目标是提供你关于安全主题的一些基础知识和资源，这样你就可以发现自己是否想要进行深入的学习。另外，我们也会对现存的 Java 平台中的安全机制做一个简介，希望使读者在 Java 平

台中编写安全的 Java 企业应用时更容易些。

然而，为了方便起见，在我们开始深入学习安全之前，先学习一些安全方面的术语。

- *认证*：这是一种校验行为，它检查系统中的一个实体（principal），不管是个人还是其他范围更大的实体（比如一个公司）是否真的与它声明的身份相符。授权的行为很简单：安全层要求（“challenges”）实体证明他的身份，实体通过提供某种形式的安全层能够辨认的认证证书来证明自己。如果认证证书是真实的，主体就成功通过认证；否则，安全层进行相应的动作，拒绝访问或通知其他的安全系统。认证有三种基本形式：根据你所知的、你所有的或你是什么来进行认证。很明显，从这一点上讲，认证检测是生活中的普遍情况，它每天都在发生——我每天早上向我的汽车认证自己，这是通过提供合适的信任证书（车钥匙）给安全层（车门的锁和防盗系统）而实现的。我向那个拿着我的信用卡的人认证我自己使得我可以用信用卡为午饭付账，这是通过提供信任证书（有时是我在信用卡上的签名，有时是我的驾驶执照）给安全层（那个拿着我的卡的人）而实现的。在美国社会中的我们甚至开始教我们的孩子有关认证的基础知识：很多小孩都被告知一种特殊“暗号”，只有那些被他们父母派去接他们的人才知道。当有人说：“你妈妈让我来接你”时，孩子通过询问暗号来进行认证检验。孩子被告知如果那个大人不知道暗号，就去报告一个可信任的权威人物（比如老师、警察等）并告诉他们发生的事。基本原则都是相同的。
- *授权*：授权行为确认系统中的一个主体能够做什么。在很多时候，实际的情形是成功的认证导致某种授权。比如，如果我成功地向我的车认证了我自己，我就被授权启动引擎并驾驶，并且在我试图进行超速行驶、急速转向等可能危害他人的行动时，我的授权被终止。不是因为汽车自己能够让我停下来，而是那个藏在广告牌后面的警察。警察在维护交通法规的时候，有权做任何必要的事情，包括撞击我的车迫使它停到路边。认证模式是相似的（你是否有钥匙？），但批准的权力就很不同了。在软件系统中，授权确定各种系统参与者可执行的行为（许可）——允许向购物车中加入一个条目，允许取消购物车，允许访问底层的表结构等等。通常，我们把这些许可分成不同的集合，称为角色，通过给用户赋予一个或多个角色使得对系统的管理更加方便（请参阅第 63 项）。
- *密码学*：密码学是将消息（称为明文）转化为其他人无法读懂的信息的科学，只有那些知道如何将加密后的消息（现在称为密文）转换为明文形式的人才能解读它。因为用来

加密消息的算法一般都是公开的（或很容易发现），所以大部分加密算法都依赖于一个密钥，它是一个数据，作为算法的一部分用来使它能够更好地抵御来自不受欢迎者的解密行为。如果交换消息的双方需要相同的密钥来解密和加密，我们称这种加密方法为对称密钥加密。如果每一边都使用一对密钥中的一个，我们称它为非对称密钥加密或公钥/私钥加密。

- *密码分析学*：它与密码学对应，它是研究如何在不知道加密消息所用的密钥的情况下把密文翻译成明文的学科。
- *信息隐藏*：这是研究如何将信息藏匿于其他东西中的学科。比如，一个密码学者试图将消息加密并将加密后的内容清楚地公布出来，而一个信息藏匿者则选择公布一份图形文件，以一种很隐蔽的方式将信息隐藏于整个图形中，同时又不影响图形的其他部分，只有那些知道图形不过是个掩护的人才能把信息提取出来。（也许每隔 14 个字节的图形信息后就是 1 个字节的消息信息）信息隐藏和密码学有时被结合起来使得明文更难被发现。它的理论基础在于如果你根本不知道信息在哪儿，你就不可能破译它。
- *Alice*、*Bob*、*Mallory*、*Eve*和*Trent*：不管你是否相信，这些名字可不是在密码学著作中随机抽取的。在一次安全协议讨论会上，Alice总是与Bob进行交谈。Mallory是那种试图打断、改变或转换话题的怀有恶意的人，而Eve是那种被动的偷听者，她只想听谈话的内容。Trent是第三方，Alice和Bob都认识并信任他。这些名字帮助我们明确在安全协议的讨论中我们正在和什么人打交道。在《应用密码学》中对这方面有详细的介绍［Schneier95］
- *机密性*：当我上六年级的时候，曾为我梦中情人疯狂，她叫 Tracey Latipow。她座位离我有两排，比我靠前三列，我认为她是我见过的最美妙的东西。然而，作为一名对社会充满好奇的六年级生，我不能简单地站出来承认我喜欢她，特别是如果她不喜欢我的话，我那脆弱的自尊一定会崩溃的。但我必须知道她对我的感觉。所以，我设计了一个简单的计划（这也许是历史上不断被使用的方法）——我向她的一个朋友写纸条询问。（如果现在我是六年级的话，我会发手机短信给她，但我 12 岁时那玩意还不存在。）问题是我的六年级老师，马丁先生，严令禁止上课时传纸条，如果他发现这种行为，他会对那个传纸条的六年级生做出最严厉的惩罚——他会向全班同学大声宣读纸条的内容。所以我不能在纸条上写直白的英语，我需要把信息隐藏起来以保证除了那个人外没有人能够读懂它。在正式的密码学术语中，这被称为数据机密性；因为纸条通过不可信任的媒介传输（教室里的其他孩子），任何一个人都可能在传递纸条时看到内容，所以

对消息的内容必须保密。这和信息隐藏不同，信息隐藏是藏匿消息的行为（当然我也做到了，通过尽量悄悄地传递纸条，通常是趁马丁先生背过身去的时候。）

- *完整性*：让我们先回到六年级，只是把信息向那些偶然的读者隐藏是不够的。设想如果有人发现了我那卓越的编码方式，并想和我开个玩笑，他假装是 Tracey 的朋友并错误地回答我的问题。那么如果那个拦截纸条的小子回答“是”而实际的答案是“不”，等待我的将是无止境的尴尬——结局是我可能愚蠢地向 Tracey 表白。用专业点的词来说，我所寻找的是数据完整性，它是一种保证，保证消息的确来自 Tracey 的朋友而不是中间的某个和我开玩笑的人。

这些基本术语是讨论企业级安全的基础。

第 57 项：安全是一个过程，而不是产品

“如果你以为技术可以解决你的安全问题，那你就根本没有理解这个问题，并且你也没有理解这项技术”[Schneier01, xii]。

对于那些偶然上网的冲浪者、各种贸易新闻发布者或各种安全性提供商的网站来说，安全看起来是一种后期开发的添加物——添加到系统中，就能使系统突然变得“安全”，可以免受攻击的侵害。只是购买一个产品，调用一些接口就可以了！使用这些方便的安全应用，你只需花费几分钟时间部署它。还有什么能比这更好呢？

开发者也用相同的方法看待密码学。为使应用安全，我们所要做的就是依据密码学算法中的数学机制以某种方式加密数据来防止它们被窃取。Bruce Schneier自己甚至都赞成这种观点，他写道：“仅用法律是不够的；我们需要用数学来保护自己” [Schneier95, xx]。

然而，这是对安全的错误态度。针对因特网的各种开发商的产品不能保证你的应用的安全性。没有哪种安全技术能够保证你的应用不受任何的侵害，甚至传输层安全性（Transport Layer Security——TLS），即所谓的安全套接字层（Secure Sockets Layer ——SSL）都不能提供这样的保障。

问题很简单：开发者使自己相信“密码学等价于安全”并且如果加密密钥够强，系统就将是安全的。遗憾的是，这是一个可怕的谬论，Schneier 自己在《秘密与谎言》的序言中也承认了这一点：

> 《应用密码学》中的错误是我没有谈论所有的（安全）内容。我谈到密码学，似乎它就是（解决安全性的）答案。我太天真了。
>
> 这样的结果并不理想。读者相信密码学就是神奇的安全万能药，他们可以在软件中到处使用它来获得安全性。他们能冒出那些神奇的字眼，像“128 位密钥”和“公共密钥结构”。一个同事曾对我说世界上到处都是由那些读了《应用密码学》的人设计的有安全问题的系统[Schneier01, xii]。

这不是一种过于自信的论调。如果有一个熟知因特网时代大部分密码学知识的人都突然觉得密码学并不是我们所寻找的解决方案，那么那些没有时间像 Schneier 那样对密码学进行深入研究的企业级开发人员，怎么可能使我们的系统安全呢？

问题不在于密码学本身的使用；而在于大部分开发人员相信密码学具有应对我们的所有安全需要的解决办法。想想那个标准的因特网电子商务应用：一个新的公司，试图在网上贩卖它的产品，为此构建了网上售货站点e-socks.com，它是世界上最早的网上皮毛制鞋类零售商。作为开发者，我们构建网站来提供各种典型的电子商务功能：购物车，客户结算等等。特别地，为了减轻客户对于在网上传递他们的信用卡号码的担忧，[1] 我们将信用卡号码通过HTTPS连接传递。所以我们是安全的，对吧？

然而，答案是——不。系统也许会以某种安全的形式来传递信用卡号以防止被不怀好意的人看到，但是老谋深算的黑客实在防不胜防。能够进入系统的方法有很多，下面列出了一些。

- *社会工程攻击法*：“社会工程”是我们给那些常被“骗子们”所采用的攻击方式的委婉称

[1] 具有讽刺意味的是，这些客户对于通过电话将他们的号码告诉不相识的客户服务代表或提交他们的信用卡来付晚饭的账单却没有丝毫顾虑，尽管这种方式同样将他们的卡号暴露给了其他人。

谓。 简单来说，一些有吸引力的、善于交际的、富有魅力的人说服系统中的一些人员来向他们提供信息。Kevin Mitnick 在《欺骗的艺术》一书中描述了儿子如何在与父亲的打赌中赢得 50 美元的故事。挑战内容是从一个录像带商店中得到父亲的信用卡号码。儿子只花 10 分钟，打了三个电话就搞到了。说服一个店员说出某个特定客户的信用卡号码到底有多难？去问 Mitnick 吧，他以此为生（现在也在这样做，只不过是通过其他途经）。

- *数据库攻击*：很多系统存储客户信息在公司的网站上作为客户档案的一部分，以使客户不需要每次交易时都输入信用卡号。大部分公司并不介意对这些号码进行加密，但事实上有的公司并不像其他公司那样重视数据库中的安全过程。（比如，大部分公司不会去假设他们的系统是不安全的。就像第 60 项中解释的那样）
- *在公司防火墙内的中间人攻击*：只要公司防火墙的任何一部分妥协，攻击者就可以自由控制系统内的任何一处资源。SSL 一般只在代理服务器或公司网络的防火墙上起作用，因为负载平衡和路由器如果要工作，就需要访问底层数据。所以黑客进入未受保护的区域（demilitarized zone——DMZ），设置一个网络探测器，并对被代理服务器解码后的包进行观察。

还存在其他的攻击方式，我确信我们所揭示的不过是冰山一角。请注意，他们都不会试图直接攻击 SSL 本身；而是攻击整个系统安全的其他方面。既然其他的多种攻击方式更容易，干嘛还要为攻击 SSL 和它的密钥交换协议操心，反正攻击的结果都是一样的。（比如，那个人可以用他那很不道德的拳头来获取你的信用卡号。）

安全不是那种我们可以简单地在系统实现的生命周期的某个时刻“打开”的特性。遗憾的是，这正是许多开发团队和项目经理对安全的态度：“好，当然，系统需要安全，但在系统实现并开始运行后再来考虑它也不迟。”虽然这种做法和态度可以用来优化系统（即使这点，也还存在争议），但它永远不能用来讨论整个系统的安全性。记住，安全必须贯穿于系统开发的每次迭代过程的分析、设计、实现和测试中，否则，安全漏洞就会出现。

比如，再次考虑 e-socks.com 电子商务应用。假设这是一个典型的 Model-Viewer-Controller 应用，我们应在哪些地方考虑安全性？安全要涉及哪些东西？下面的列表包含了其中的部分问题：

- 假设站点使用某种每个用户一个会话状态的机制（例如HttpSession），我们必须保证攻击者不能猜出某个正在使用的合法的JSESSIONID值，从而能够访问到其他用户的会话状态。对于e-socks.com，情况可能是一个攻击者使用我的信用卡来把丝袜运送到他的地址。对于一个处理财政或医学数据的网站来说，结果可能要糟糕得多。永远不要使用不能为JSESSIONID查询参数产生某种安全的随机值的servlet容器。
- 每个处理网页上输入的servlet必须保证输入在合法的范围之内。比如，当验证某个用户对数据库的登陆信息时（SELECT * FROM user WHERE...），保证用户名和密码没有隐藏SQL注入攻击。
- 每个servlet和JSP页面必须被小心地检测以保证无序的网页请求不会绕过关键的数据项信息。比如，不允许绕开“选择付款方法”网页来直接跳到“确认订单”页面。当然，理想状况是后台处理会验证订单在发出前已被支付，但作为程序员，你需要多频繁地重复检测那些“我知道已经被检测过了”的东西？
- 页面如何计算当前用户购物车中的总价？如果数值来自一个隐藏字段，攻击者总能够绕开整个浏览器并手动提交（通过 Telnet）一个HTTP请求，请求中包含了数千的购物车中的条目，而隐藏字段的值是“0.01”美元。即使这个值是根据每个请求计算出来的，购物车从哪里得到购物车中货物的单价？这又回到了前面的情况，如果它来自于一个HTML表格的字段，这个数据就可以被非常方便地修改。

本章中后面的项对这些问题进行了详细的讨论，但所有这些问题都只是对本项主题的支持，Schneier 反复地用到一句话：“安全是一个过程，而不是产品。”如果你希望建立一个能够抵御远程攻击的系统，那么在系统开发的每个阶段你都要认证地思考这句话。这需要改变你的思维模式：这需要你暂时戴上攻击者的黑帽子并思考你将如何攻击这个系统，然后再戴回白帽子，想想你怎样来阻止那次攻击。不仅仅要考虑架构和技术上的因素——在系统实现上的各个层次上的每个人，都必须时刻把安全性记在心里。“编写安全代码”应该是程序员的一条指导原则，就像“编写好的代码”，“编写优雅的代码”那样。

第 58 项：记住安全不仅仅是预防

让我们暂时把软件世界放到一边。如果我们正在寻找如何构建安全系统的优秀模式，那么至少在一定程度上，我们可以回到“现实世界”去寻找什么管用什么不管用的例子。

在现实世界中，在自动摄像机、电子围栏、动作探测器和区域警报司空见惯的时代，简单地防止攻击者进入系统是远远不够的。先回忆一下欧洲历史—在中世纪，防御的中坚是城堡：高三十多英尺厚一两码的石质城墙，被护城河围绕，要接近城墙的底端即使不是不可能也是极其困难的。有了这些保障，城堡似乎是坚不可摧的目标（至少在火药发明之前）。既然这样，那为什么每个城堡还要有卫兵？如果城堡是如此坚不可摧，为什么还要雇人在城墙上巡逻，除了观察入侵企图外什么都不做？类似地，在现代，想想汽车警报—是什么使我们相信一声刺耳的长鸣会赶走偷车贼？是因为声音响到足以对身体造成伤害吗？

当然不是。无人城堡最终会被占领—有人会发现一架梯子就能帮他达成目标。汽车在空旷的停车场中央发出警报，窃贼一旦发现没有人来阻止他，还是会把车偷走。任何安全系统的另外两个部分就是*检测*与*反应*。

再次回到城堡。当入侵者出现时，城墙上的守卫会立即通知城中的其他人有不速之客。城主会立即召集他的军队，他们将会用一种非常有说服力的手段阻止入侵者进城：长剑、弓箭、沸油等等东西。

当敌人进攻城堡时，守卫会反击，力图阻止敌人进入城堡，一般通过一些野蛮的行为，像在进攻者头上泼沸油，或向他们射箭，或推倒他们的梯子。守卫不会呆呆地站在那儿看着那些人爬上城墙放下吊桥—他们会用必要的武力进行反击以阻止攻击者进入。这就是为什么要付钱给他们，至少在理论上是这样的。

自动报警器的工作原理类似。我把房门和车门关上，防止一个攻击者能够很容易地进入我的房子或汽车，但还是有办法不通过门进入：窗户是一种简单的途径。（在圣地亚哥，一个勤奋的窃贼用切割器在墙上为自己开了一扇门。）所以我在窗上安装了感应器；如果他们被破坏，将触发一个警报，不但会通知我和我的家人，而且会通知我每个月都付钱给他们的警报

公司。一旦我或警报公司检测到有人试图闯入，就会立即报警，警察就会赶到现场，逮捕闯入者，如果需要的话还会使用武力。其他的房主可能采取不同的方法，使用床边的合法枪支——作用是相同的。

所以，如果简单的防御措施在现实生活中会不起作用，那我们凭什么相信在软件系统中它就一定管用呢？防火墙当然会阻止大部分对系统的攻击，但如果正好有人透过了它，它还能做什么呢？你怎么知道你被攻击了？你怎样回应？然而，这一领域又正是软件安全工业存在严重不足的地方。当然，入侵检系统可以从市场上购买，但它们都有一个严重的缺陷：误检测。软件包周期性地将正常的网络传输当成入侵企图并响起警报。事实是这并不是一次真正的入侵企图；结果是软件被重启，系统管理员又回去做日常工作。

为什么误检测如此糟糕？让我们回到前面的汽车警报例子：它基于这样的假设：当警报响起时，有人会听到并报警（检测）以阻止盗窃行为（反应）。你最后一次听到警报后报警是什么时候？以前，警报响起后一般都会有热心的人跑去现场，因为他们知道有人正在试图闯入别人的汽车。毕竟，某一天如果有人在偷我的车，我也希望有人去阻止，对吧？

但问题是很多报警器对汽车周围的情况太敏感了：人们知道火车或飞机掠过汽车会触发警报，也可能是什么东西偶然碰到了车门，或者是你三岁的孩子跑进了停车位的警戒区。作为人类，我们意识到误检测出现的几率比实际检测到入侵行为的几率要大得多得多。我们渐渐放弃了意味着真的有了麻烦这样的想法：其结果就是，警报响了，我们的反应要么是诅咒车主为什么还不出来把这该死的东西关掉，要么就假装听不见，继续作自己的事。

这个问题并不新鲜——叛军和游击队早就知道攻击军事基地的最好方法不是冲向电网和枪弹。有强大火力的守卫开着装备了机枪的吉普车或装备了大口径武器的装甲车到处巡逻，在他们面前，一小撮拿着步枪的自由战士可支持不了多久。因此，叛军采取潜入行动；用一种虚假的安全感来迷惑守卫。

叛军找来一只兔子或其他的小动物，把它丢过围墙，然后躲起来。警报响了，守卫冲了过来，在围墙边找到了一具烧焦的动物尸体，然后通知大家警报有误就回营地去了。一个小时后，叛军故伎重演。警卫冲出来，又发现一只动物，然后回营地。三四次以后，警卫对警报就麻

木了——他们甚至都不愿意派人去检查大门，这时，叛军剪断铁丝网（警报会再次响起，但这次要么没人理睬，要么是守卫去把这可恶的东西关掉），潜入，进行致命一击。

你认为这方法不会成功？80 年代，人民圣战者组织（mujahedin）反复对占领阿富汗的苏军使用它。90 年代，在前南斯拉夫，科索沃人用它来对付他们的塞尔维亚占领者。也许就是它使得那艘装满炸药的小船能够靠进美军的Cole号军舰（“嗯，这只是又一艘摩托艇，杰克——看起来它对我们不能构成什么威胁”）。你认为这种战术不能被用来对付你的公司？这取决于公司是否安装了某种入侵检测软件；很多公司都没有。即使是安装了这种软件的公司，也被每天成千上万的误检测吞没，系统管理员已经受够了，他要把警报声音关小或干脆把它关掉。

从编程人员的角度看这意味着什么？它意味着你的软件为了提供友善的管理(请参阅第 13 项)，必须在任何可能的地方提供检测和反应机制。当然，经典的策略是只允许有限的登录失败次数，一旦超出范围，用户账户就会被锁定，只有那个可信任的系统管理员才能解锁。如果你的 servlet 开始接收到奇怪的用户输入——它们本应该被客户端的验证代码捕获（比如，HTML 浏览器中的 Javascript），那么可能正有一个攻击者试图找出你系统的安全底线。请通知管理员，最好记下通讯的来源和尝试的次数等。在你的威胁模型的每个点上（请参阅第 59 项），插入代码检测入侵企图，并用日志记录它，用某种途径通知管理员。这时要小心，既要保证有人注意到你的通知，又不能因为每次都是误检测而使人们想要关掉它。（电子邮件是一个好的方法，手机短消息或网页有时也可以。）最终，软件不能仅依赖于防御——检测和反应也必须要有，防御甚至不是我们首先要考虑的。

第 59 项：建立威胁模型

所以你相信你需要认真地在你的应用中考虑安全性。现在做什么？开始阅读所有能弄到的关于安全性的资料（书籍、文章、网页等）会使开发人员沮丧——不管是加密部分，还是正在被讨论的安全协议，似乎总是有突破它们的办法。对于加强应用的安全性我们真的什么都不能做吗？

首先，记住安全性远不止防御（请参阅第 58 项），很多对于安全协议或加密算法的攻击都有一个前提：攻击者有无限的时间来攻击一个指定的系统。你插入的安全代码永远不能抵御有这种优势的攻击者，所以别指望你能做到。

第二，不管怎样，我们需要接受现实——构建“有完美安全性的系统”就像在追寻圣杯——它会榨取你大量的时间，而最终你不会得到什么好处。有句老话“不要捡了芝麻丢了西瓜。”所以我们现在需要进行冷静的分析，看看为了保护应用我们愿意付出什么样的代价。理想状态下，应该是合理的时间和金钱，但很多时候“我们需要花多少钱？”取决于“它会花掉我们多少？”，这个稍后会谈到。

一旦我们确定愿意为保护应用所花的美元和人力，我们就需要将注意力转向需要为应用提供什么样的安全。一个攻击者可以通过多种方式接触系统，要覆盖所有的入侵似乎是不现实的，而且会分散我们的精力，战线拉得太长会使每个部分都变得薄弱。我们需要准确地确定系统的哪些部分最重要且易受攻击，优先保护他们。有两种方法可以被用来进行处理。

第一种方式是大部分开发人员的首选，它直接依赖于开发人员的直觉来确定系统的哪一部份是在成功的攻击事件中最易受攻击或最关键的部分。但是，就像第 10 项中所指出的那样，大部分时候，开发者的直觉都很糟，不值得信任——既不能优化系统，也不能改善安全性

第二种方式是使用一种可度量的有计划的方法来处理。

在过去几十年的软件开发中，我们已经了解到如果没有一种引导模式，一种对系统整体结构的视图，就直接放任开发者自由发挥是一个非常糟糕的主意。这就是我们为什么要有“设计优先”的方法学，就像 Rational Unified Process，因为作为一个实现者，我们真的非常需要能看到整个宏图，这样才能在正确的时间正确地处理那些细节——否则，我们就要承担以后付出巨大代价来纠正错误的风险。

（注意，大部分敏捷方法的支持者，他们常被人误解为是反对设计的，但事实上他们建议一些必要的设计还是需要的——比如，在极限编程中的任务实现就需要设计，特别当需要重构时。极限编程人员反对的是“为明天设计”，取而代之的是“为今天设计”。）

类似地，为了安全性，我们需要某种针对我们的应用的安全性的宏图。我们需要知道哪个地方最容易受到攻击，以及如果它们被攻破会造成怎样的破坏。这些信息反过来可以帮助我们区分哪些弱点是需要关注的而哪些是可以忽略的。这些资源一般被称为一个威胁模型，顾名思义，它的目的与对象设计模型在很多方面是相似的。

许多安全方面的人士提倡开发安全模型的方法学。比如，Schneier，使用“攻击树”将攻击点按不同层次映射到很大的、像 UML 那样的图中，它在必要时还可以扩展或收缩[Schneier01，318-333]。它往往提供了进入系统的各种途径的一个宏观视点，层次结构的组织方式使我们能够很容易地估计攻击类型。

其他作者提倡一种更简单的模型，在这种模式中开发人员坐在会议桌旁，讨论各种可能的攻击方式，然后给讨论的内容列表中的每一项一个威胁值，它根据两个因素得到：威胁出现后产生的经济上的损失，和威胁出现的几率。（这种模式也被用于风险评估研究；因为安全常常关系到管理风险，在这个例子中是遭到入侵所带来的风险，其中使用到了很多相同的原则。）所以，比如如果我们确定一个攻击者成功获得 Web 服务器的最高管理权并破坏网站意味着需要两个工作时来恢复（我们有很好的备份）并且破坏带来了大概 10，000 美元的损失，那么这个受攻击点的权值大概是（假设一个好的系统管理员年薪 6 万，大概是 30 美元一小时）$10,000 + 2 小时，(每小时值 30 美元)，或 10，060 美元。现在，我们估算这种攻击成功的机率是 10%，因为我们非常小心地在低级权限账户下运行 web 服务器（这就迫使攻击者必须对其他地方进行某种引诱攻击才能获得最高权限；请参阅第 60 项）。这意味着总的安全风险在这里大概是 1，006 美元，这就是我们修复这个缺陷所应该花的钱——多了就不值了。

不是所有的安全评估都能够这样简单地计算；有些破坏造成的损失是非常难以估算的，像有害公开（当黑客攻入公司后，公司的客户数据被公开到黑客的网站上）所造成的损失就是如此。记住，这里我们所赋的值完全是为了建立威胁模型，只是作为评估哪种攻击需要优先处理的参考。如果能够使评估更简单，可以用抽象的“危害性”单位来计算损失，

这里的重点是如果没有威胁模型，想知道需要花多少时间和精力来防范一个指定的企业系统

可能面对的某个潜在的安全攻击是不可能的。比如，你是否信任你的系统管理员？是否信任你的用户？是否信任每一个在公司防火墙内的人？如果你对所有这些问题的答案都是“是”的话，请允许我提醒你存在的风险：像对现状不满的系统管理员、社会工程用户（socially engineered users）、访问你公司网站来教课的 Java 教师（或其他的访问者），他们进入网络“只是为了发邮件”，但事实上，他们可能已被你的竞争对手收买，在网络上到处打探希望偷听到会话并试图推测出机密的数据。并不是我的竞争对手曾这样对我干过，而是每次在课上提起他们，总会引起一些人的惊讶，并使他们开始思考这个观点到底在向我们建议什么。商业间谍活得很滋润。

如果你突然认为对其中一些问题的答案是“不”，下一步就是确定你打算怎样做，以及威胁模式到底试图告诉你什么——哪种威胁是需要重点抵御的，哪种威胁是你无能为力的？毕竟，如果全副武装的突击队员席卷数据中心，硬是将硬盘拔出服务器，你的软件并不能对他们做什么，不是吗？

第 60 项：作不安全假设

所以，你最终建立了庞大的 J2EE 系统，准备部署到产品环境中，每个人都非常兴奋。CEO 承诺付给每个参与人员一笔可观的奖金，负责开发的副总已经准备好接受来自身后的祝贺。到目前为止，这对每个人都是一次愉快的经历。系统已经成功通过了质量验证（有少量 bug，但没有什么能够阻止系统被部署到产品中），我们准备开始了。

系统管理员，准备安顿这个小宝贝，先安装数据库软件（如果还没有安装的话），J2EE 容器，以及让整个系统运作所需的第三方软件。他们弹出光盘，设置缺省值，几小时后，部署代码的准备工作就绪。他们浏览了部署脚本（请参阅第 14 项），运行了一个“happy bit”测试来保证所有的东西都运行良好并打开监控工具（请参阅第 12 项），宣布所有构件都已就位；系统已经正式被部署到产品中！人群沸腾了，每个人都去酒吧庆祝，公司的股价也因此而上涨了几点。

然而，就在你打开第二瓶香槟之前，注意那个你（和其他人）在走出房门时所做的危险的假

设。你假设安装的缺省值是可接受的。当然，从性能或正确性角度来说，缺省值的确能够工作，但是，更重要的是，你是否确定厂商提供的缺省值能够给系统带来安全？

稍微考虑一下 Oracle 数据库的安装。缺省情况是：如果没有采取一些步骤来更正，每个 Oracle 数据库会至少创建一个人人都知道的登录帐户：用户名 scott，密码 tiger，地球上每一本 Oracle 的书都用它作为例子，而且大部分 Oracle 数据库管理员和开发人员都未对它进行处理，但他们忘了地球上每个 Oracle 开发人员都知道这个帐户。类似的，SQL server 的缺省管理员帐户是 sa，没有密码，就是说“提示输入密码时只要按回车就行了”。其他的数据库也差不多。

这里给你一个有趣的测试：浏览你公司的所有 Oracle 和 SQL Server 数据库实例，尝试 scott/tiger 和 sa/<空>，看看有多少次你能成功登录。（当然，确保你已经向系统管理员报告了你所做的事，以防止万一你偶然触发了一个入侵检测系统或他们碰巧抓住你正在数据库中乱转；我可不想你因为我的建议而被炒鱿鱼。）如果你的公司和其他公司一样，那你很可能震惊地发现竟然有这么多次这些缺省帐户都被遗留在系统中了。

这还不是最坏的，很多时候这些帐户依然有缺省的权限，这意味着如果它是一个标准管理员帐户，你有权访问所有安装在服务器或集群上的数据库实例。如果你感到非常生气，那想象一下它所带来的混乱情况吧。[2] 即使缺省帐户的权限不是很大，你怎么阻止别人通过它用一些流行网站上的大量流媒体文件把磁盘塞满来进行拒绝服务攻击呢？

这是建立一个安全系统的特殊的要点：你必须假设每一样东西都是不安全的，包括你正在建立的那些部分，直到你确定系统可被入侵和攻击的安全疑点都被清除了。当然，如果当你的供应商对你说那是安全的时候你相信他们，那你就只管假设那是安全的好了，但在这之前，请停下来想一想，你真的相信如果系统是不安全的，供应商会告诉你吗？

逻辑上接下来的一步是证明系统是安全的。你可以试图通过建立一张一般攻击的清晰的检验

[2] 我不是建议你用它来报复你的老板或公司，因为他们轻视了你，不管这种轻视是真实存在的还是你想象的。但安全准备的一个要点就是向攻击者一样思考，理解他们将如何攻击你的系统。

清单来做到，特别是当你得到公开网络应用安全项目（OWSP）[3]十大网络应用攻击列表和其他文档的帮助时。你可以对历史上已知的成功攻击过供应商产品的攻击方式进行测试，你甚至可以测试一些你所联想到的新的攻击方式。遗憾的是：所有这些测试都仍旧不能证明系统是安全的。

问题是证明一个系统是安全的就像证明某人正在计划进行叛国活动那样：它是难以掌握的，模糊不清的，时刻都在变化的。比如，即使我们能够证明系统今天是安全的，明天黑客社区可能会发现对供应商的软件的新的攻击方式，突然之间，我们的系统有了一个我们甚至都不知道的弱点，至少在它被用来成功地攻击我们的系统之前我们是不知道它的存在的。

总的来说，有两条关于企业级安全性的定律。

1. 假设一个构件是不安全的，除非你能证明。
2. 一个构件永远不能被证明是安全的。

这到底告诉我们什么？这两条似乎将我们引向了一种不可能应对的局面。

欢迎来到令人恐慌的企业安全世界。

在你认为我是对于这个问题的妄想狂和失败主义者之前，我并不是唯一一个宣传这种悲观论调的人；Schneier 写道“在安全性方面进行的这么多年的工作中，我们还没有看到一个安全的完整系统。是的，我们分析过的每个系统都曾被这样或那样的方法攻破。总有一些构件是很优秀的，但人们总是以不安全的方式使用他们” [Schneier03, 1]。

这告诉我们一个很简单的道理：总是假设每件东西都是不安全的，假设人们会通过其中一些构件攻破系统。考虑到当前我们所面对的实际情况，你除了接受它之外的确别无选择。我意识到这药很难下咽——需要的话你可以花点时间鼓气勇气——但是，良药苦口啊。

[3] 一个开源项目，旨在帮助组织识别并理解安全的网络应用和网络服务的细节。参考 http://www.owasp.org.

并不是说这就是我们最终的宿命，虽然看起来像。如果我们再次回到现实世界，就会发现这是事物的一种普遍状态。许多安全设备在设计和建造时都被人们假设为最终是可破坏的。想想吧：如果银行没有假设劫匪能够破坏门锁，他们为什么还要把钱放到深藏在大楼中的金库里？如果当局没有假设敌人能够通过铁丝网，为什么还要派军队开着吉普车架着机枪在周围巡逻？如果汽车制造商没有假设偷车贼能撬开车门上的锁，为什么还要设计用钥匙来发动引擎？

本质上来说，所有这些现实世界中的安全系统都使用深度防御策略。而不是一股脑儿把所有安全措施都堆积在一起，它们假设在每个层次上，防御都会被攻破，因此需要另一种防御来以防万一。让我们再次回到银行的例子，一般银行都会建立一些安全层次防止有人从保险库里偷了钱后溜走。假设银行窃贼在晚上行动，我们已经锁上了门，我们放置了摄像头来监视门和窗户（用来对付窃贼，因为我们可以辨认并追捕他们），我们在每扇门窗上都安装了传感器，并将他们和地区警察局的报警器绑定，我们的保险库用将近一码厚的钢板建成，并安装了极其牢固的门锁。如果窃贼想要在白天袭击，我们在大厅里安排了守卫，有的还带了枪（更多的预防），出纳员通常站在一些物理屏障的后面，并可触发一种无声警报来通知警察，这样如果窃贼动作不是非常快的话，保险库就可以安然无恙。

我们如何将这种深度防御概念引入企业系统？首先，我们放弃使用防火墙能够保护所有东西，或是只要我们坚持更新操作系统补丁就不会有问题的想法。直接把 Web 服务器丢给 DMZ 是不够的——它只是一个步骤，但我们必须深入一点，就像拥有防火墙或 DMZ 只是一个步骤一样。

下面是一些部署一个深度防御策略需要考虑的东西（当然没有包括所有方面）。

- 页面如何计算当前用户购物车中的总价？如果数值来自一个隐藏字段，攻击者总能够绕开整个浏览器并手动提交（通过 Telnet）一个HTTP请求，请求中包含了数千的购物车中的条目，而隐藏字段的值是“0.01”美元。即使这个值是根据每个请求计算出来的，购物车从哪里得到购物车中货物的单价？这又回到了前面的情况，如果它来自于一个HTML表格的字段，这个数据可以被非常方便地修改。

- 在系统核心构件（HTTP服务器，数据库服务器等）周围建立防火墙。
- 在系统的转换部件（用户的客户端软件必须与它们进行交互，典型的是 HTTP 服务器）后面建立二级防火墙，从而建立一个 DMZ。配制二级防火墙使它只接受来自 HTTP 服务器的传输。
- 关闭 HTTP 服务器和数据库服务器上的所有系统服务；这不需要什么理由，比如，HTTP 服务器或数据库服务器需要回应 TCP/IP date 或 echo request。你可能只需要 ping 作为 TCP/IP 基本服务，但即使这点也还存在争议。比如，对于 Win32 操作系统，HTTP 服务器的确不需要 MSMQ、COM+系统应用或事件系统服务、红外线监视服务等等。所有这些都只会成为潜在的攻击点，如果服务器永远都不会用它们来做什么，为什么要打开它们？为 Solaris 或 Linux 服务器做一个相似的检查列表——你真的需要 X 运行在所有这些产品机上吗？当然，它们有时带来了方便，但是否值得为它们冒潜在的安全风险？如果假设系统管理员已经可以通过键盘和监视器来对机器进行管理，那甚至 Telnet 都可能需要关掉。（是的，我知道打开 Telnet 会更方便，但是我又要重申，你能证明机器的 Telnet 守护进程是安全的吗？）
- 永远不要使用和管理 HTTP 或数据库服务相同的登录账户。换句话说，确保在应用中使用的系统管理员和数据库管理员的用户名/密码是不同的；由此可以得出，如果可能的话，永远不要在远程连接中使用管理账户，也不要在防火墙外使用。
- 确保在安全边界上（比如，在用户访问的服务器和用户之间）的所有东西都有自己的认证检验——关掉缺省的，确保选择的密码不会太容易被攻破（如果可能的话——要是用户坚持使用“password”作为密码，你也没有办法。）等等。这对每个设备都有效，包括路由器、集线器、防火墙、代理服务器等。

就像你所见到的那样，当你这样思考安全性时很容易陷入妄想症。但这是一个好的开始。

记住，我们不能阻止一个攻击者，预防本身永远不能保证提供一个安全的系统（请参阅第 58 项）。目标是拖延攻击者，以使入侵检测系统和警报系统管理员可以检测到攻击并采取合适的反应措施，包括在必要的情况下关掉账户或系统（取决于被攻击系统的敏感度，这就是为什么你首先需要建立威胁模型——请参阅第 59 项）。

这一条的一个结论就是采用最低权限原则——只给与运行代码所必要的最低限度的权利。比

如，对于需要访问关系型数据库的代码，不要使用可以执行SELECT/INSERT/UPDATE/DELETE动作以外的动作的账户。就像第 61 项中所解释的那样，如果代码被一个命令注入攻击迷惑，如果账户有权操纵schema，你也许最终会发现你正在盯着一个没有任何表单的数据库实例。相似的，如果代码有权访问文件系统，而且代码再次被成功地欺骗了，你会发现大量的不可识别的文件（它们都以XXX开头，真够奇怪的）被存储到了你的服务器文件系统中。

顺便说一下，这种态度并不仅限于你的代码。一有机会就使用最低权限原则，包括在 Java 程序访问底层操作系统时。如果你安装了一个 J2EE 容器作为单独的不区分用户的应用（一个 UNIX 下的守护进程，一个 Win32 下的服务），确保进程运行在最低权限上，即使这将意味着需要为容器创建一个特殊的账户。

（还有，在NT/Win2K/XP下，用来运行进程的账户可以通过Services snap-in来配置——启动它，找到可疑的服务，进入这个服务的属性页面，一般通过点击右键从栏目中选择属性。属性页面中有一项被称为登录，通过它你可以配置执行进程的账户信息。一般来讲，对于一个商业系统，你不会使用LocalSystem 账户，因为这个账户被其他服务所使用，你希望将赋给账户的权限变得越低越好。对于Win32 安全的更多信息，包括所有的权限及它们代表的意义，我强烈推荐Programming Windows Security[Brown]。）

第 61 项：总是验证用户的输入

开放Web 应用安全性项目（http://www.owasp.org），在第 60 项中提到过，是一个开源合作项目，旨在帮助那些组织（在本书中特指在那些组织中的开发人员）识别并理解安全的网络应用和网络服务的细节。作为功能的一部分，OWASP提供了一张十大网络应用安全攻击文档，风格就像SANS/FBI二十大列表。

在 OWASP 列表中占首位的是不合法参数。事实上，十大中有三种都使用相同的方法直接处理用户输入：不合法参数（#1）、缓存溢出（#5）和命令注入缺陷（#6）。如果我们稍微扩展一下处理用户输入的概念，那我们还可以包括交叉站点脚本缺陷（#4）和错误处理问题

（#7）。显然，我们如何处理用户输入是建立一个安全系统的关键。如果你停下来思考一下，很容易发现其中的原因：任何一个系统接受的用户输入都是攻击者进入系统的通道。

当我们建立一个基于 HTML 的应用，验证用户输入绝对是构建安全应用的一个重要环节。然而，太多的开发人员依赖技术来帮他们搞定一切而使自己暴露在攻击者狡诈的策略之下。比如，就像第 56 项中所说的一样，很多网络应用太多地使用客户端的脚本来处理几乎所有浏览器中的验证工作。这很快速并且可以防止走弯路；当需要改变用户接口时我们可以直接操纵 UI 元素而不需要回到服务器中更新显示。

事实上，客户端的验证看起来是个不错的主意，为什么要花力气在服务器上再次验证输入？毕竟，当用户或攻击者浏览页面的时候脚本肯定会被执行，因为我们已经（推测）确保在登录页面中用户的浏览器已经开启了对脚本的支持，为什么要在服务器上重复这些验证？这只会浪费 CPU 的工作周期，而浪费 CPU 周期不仅会影响性能而且会影响可测量性。

一旦你停下来思考就会发现，问题在于攻击者不会被这些规则要弄——他们并不限于通过浏览器攻击你的应用。很多攻击者对 HTTP 非常了解，以致于他们仅使用 Telnet 就可以虚构一张表单并提交，因为大部分 Telnet 客户端还没有被扩展来支持 JavaScript，于是你所有的客户端验证逻辑都成了垃圾。

残酷的事实是作为一个开发者，你必须假设客户端验证逻辑没有被执行，对于每个用户输入的提交，都要严格地进行一系列的验证检验以保证用户输入中的所有基于输入的攻击都被过滤了。

参数验证错误有各种大小和形式。比如，作为测试你的 Web 应用的一部分，每次向用户要求输入的时候，甚至在输入数据就在那儿，根本不需要输入（HTTP 报头、cookie 值、等等）的时候，试试输入各种无效数据看看你的 Web 应用有什么反应。当你输入以下这些东西时发生了什么？

- Null？
- 不同字符集中的字符？Unicode 字符？不可显示的 ASCII 字符？
- 长度为 0 的参数?1K 长的参数？10K 长的参数？

- 在需要输入字母时输入数字，然后反过来试试？
- 重复数据（比如，同一张表单中填写两次相同的参数）？

你也许会认为这应该归入“质量评估的怪问题”中，因为显然没有人会在需要电话号码的时候输入一个姓名，但请记住，我们不是在讨论用户，我们在谈论攻击者，他们可不会遵守这种规矩。

比如，让我们以一个标准的用户登录页面开始。在很多（如果不是全部的话）系统中，用户认证的细节被储存在一张 RDBMS 表中，一般是一张简单的有两列的表，分别保存用户登录的用户名（uid，一个 20 个字符长的 VARCHAR）和用户信任认证信息，在本例中是一个密码（另一个 20 个字符长的 VARCHAR）。其他存在表中的用户细节，比如配置设定，个人数据，授权设定等等，在此不讨论。

一般的做法是建立一张登录 JSP 页面，核心部分看起来像这样：

```
<form action="/LoginProcessor" method="POST">

Your Username: <input type="text"

                    name="username"><br />

Your Password: <input type="password"

                    name="password"><br />

<input type="submit" value="Log in">

</form>
```

目前为止，一切都好，LoginProcessor 的URL指向servlet，servlet的核心部分需

要产生对数据库的SQL语句来检查uid/password对是否的确在表中存在。没有什么比这更简单了，对吗？

```
Connection conn = getConnectionFromSomewhere();

Statement stmt = conn.createStatement();

String SQL = "SELECT * FROM users " +

  "WHERE uid = '" + request.getParameter("username")+

  "' AND password = '" +

  request.getParameter("password") + "'";

ResultSet rs = stmt.executeQuery(SQL);

if (rs.next())

{

  // User/password pair was there; go ahead

  // and forward to the next JSP in the page flow

}

else

{
```

```
  // Whoops! User failed to authenticate safely;

  // keep track of the number of times this user

  // fails to authenticate (see Item 60). After

  // a few more tries, lock out the account in case

  // it's an attacker trying a brute-force attack.

}
```

我们搞定了，对吧？我们检查表单中的下一项，继续处理下面的环节/任务/不管是什么。

然而，不，我们没有搞定。事实是，这段代码可以通过很多途径被命令注入攻击方式攻击（OWSP，攻击方式 #6）。当用户输入可以携带某种在处理过程中能够被系统中的某一层所执行的代码时，就发生了命令注入攻击；在这种情况下，它在概念上类似于缓冲溢出攻击，后者试图从堆栈上打开缺口。

来看看这种攻击的行为，让我们站在攻击者的角度来看。一种首先被采用的技巧是盲目地进行注入攻击并看看它是否在某种情况下会成功（“成功”在这里指任何可以显示攻击奏效的线索）。比如，攻击者可能提交一个名为'SELECT'的用户名，也就是一个单引号，SELECT 关键字，然后是另一个单引号，看看他能得到什么。在前面的代码中，在文本之前的单引号会使我们的 SQL 语句中的引号无效。从而使 SQL 解析器认为应该填入的字符串是空，然后它会看到 SELECT，这显然是 SQL 中的关键字，在错误的地点被成功解析，它会抛出 SQL 异常。

所以……看看我们到底是怎么处理 SQL 异常的？

这里有两种不同的场景。第一种，理想的场景，我们捕获 SQL 异常，不考虑错误，显示一个组织得很好的页面来告诉用户发生了错误，他们是否愿意重新输入？然而，这儿的问题是在登录失败时显示这条消息会把用户搞糊涂，所以我们需要一个不同的页面来告诉他们登录失败了，请重输。虽然我们对不合理的用户输入和失败的 SQL 调用进行了区分，但我们给了攻击者一个很重要的线索。事实上，这种区分正中攻击者下怀——他知道他的单引号输入搞乱了 SQL 语句，这就意味着 SQL 命令注入攻击可能会奏效。所以他继续攻击。

第二种场景中——很不幸，它是我们的缺省场景，我们只是捕获异常并将它作为 Servlet 异常或 IO 异常重新抛出（因为这是 servlet 的 doPost 方法声明的异常），这种方法认为，一旦抛出了 SQL 异常，就意味着程序员出错了，我们至少希望看到堆栈踪迹。这结果是很可怕的，因为现在，当攻击者尝试他的命令注入实验时，他会看到堆栈踪迹，他很可能会从中看到诸如"Malformed SQL statement: unexpected SELECT"或其他类似的异常信息。对于攻击者，这就等于告诉他“绿灯！继续前进！你就快达成目标了！”。因此那些老谋深算的攻击者下次精心选择注入命令。如果攻击者想作为一个普通用户获得访问，他会像这样提交登录表单：

```
username = boss'; SELECT pwd FROM users WHERE password = 'foo

pwd = foo
```

将这些插入到将会在 servlet 中动态建立的 SQl 语句中，我们开始预感到情况不妙了：

```
SELECT * FROM users

  WHERE username = '



boss';SELECT password FROM users WHERE password='foo
```

```
' AND password = '
```

```
foo
```

```
'
```

整理一下，我们得到：

```
SELECT * FROM users WHERE username = 'boss';
```

```
SELECT * FROM users WHERE password = 'foo' AND password = 'foo'
```

现在，因为分号在很多SQL语言中作为语句的分割符，当我们认为正在执行一条SQL语句的时候，由于攻击者的不正常输入，我们实际上在执行两条语句。当然，JDBC驱动如何反应取决于所采用的JDBC驱动类型，但大部分会支持这两个ResultSet实例作为“额外的结果”，这意味着访问第二个ResultSet我们需要调用getMoreResults。然而，在我们的servlet代码中，我们只是检查在第一个ResultSet中是否有结果,因为查询成功了（当然，要假设攻击者得到了正确的用户名，这不是什么难事），结果攻击者以老板的身份登录了系统，这也许不是什么好事。

假设我们的攻击者没有这么狡诈，然而：

```
username=boss'; DELETE FROM users WHERE password != '
```

```
password = <empty>
```

再次产生了可读的 SQL 语句，我们遇到了非常严峻的形势：

```
SELECT * FROM users WHERE username = 'boss';

DELETE FROM users WHERE password != '';
```

突然，我们的系统管理员不允许我们使用非空密码了，因为所有使用非空密码的用户的登录信息都被那条在关系数据库中运行的毫无益处的 SQL 语句删除了。突然间没有人可以登录了，你在早上三点接到 CTO 或副总裁的电话，命令你弄明白为什么你的系统突然不能让任何人进入了。这可不是你想要的吸引上层注意的方式。

更有趣的事还在后面；假设你的系统有某种基于角色的授权机制，而且你的“根”账户是用户表中的第一条纪录（一般情况都是这样的），攻击者就可以这样做：

```
username = boss' OR 1 = 1 --

password = <irrelevant>
```

整理一下就是：

```
SELECT * FROM users WHERE username='boss' OR 1=1 -- AND
   password=''
```

注意到在许多 SQL 风格中那两个横杠是行结束符。因此，检验密码的语句就无效了。因为 1=1 是永真的，整个用户表会从这个语句中返回，因为最高权限的管理员账户会第一个被返回，我们无畏的攻击者就会以此登录。天哪！

不仅仅是登录页面潜在着这种弱点，其他很多地方都是这样。到处都有 servlet 代码直接接收用户输入并以此创建 SQL 语句的情况。事实上，在任何 servlet 代码从到达的 HTTP 请求中接收数据，再据此创建 SQL 语句的地方，这种弱点都存在，不管是不是真的需要用户进行某种输入。典型的招数就是把数据放入隐藏表单中，并把它作为创建 SQL 语句的一部分。

```
String SQL = "INSERT INTO order_totals VALUES (" +

 // . . . Other data, like order ID and items,

 // go here . . .

 (totalPrice *

  Float.parseFloat(request.getParameter("discount"))) +

 ")";
```

这里通过 JavaScript 将“discount”设置成隐藏字段，所以，狡猾的攻击者在 HTML 页面里浏览一下源码（或直接用 Telnet 点击 URL），看到隐藏的“discout”字段，一下子，他就以原价 1%的价格买下了所有商品（如果他把 discount 值设为 0 的话，他就一分钱都不用花，但这可能会触发某种报警器。这样他就几乎不能得到什么好处）。

像这样的例子我们还可以举很多，但它们只是表现形式不同而已。重要的是要找到应对的办法。

在那个登录和订单处理的 servlet 例子中，第一步是停止直接从用户输入创建 SQL 语句。我们最先的反应可能是编写验证代码来发现并过滤输入字段中的 SQL 代码（在关于登录的 servlet 例子中），但这不能帮助处理订单的 servlet，特别是如果真的存在能够让某些人打 99%或 100%的折扣的情况。（比如，CEO 们每次都这么做。）

庆幸的是，存在另一个更简单的答案：永远不要使用基于用户的输入的SQL语句。这样做迫使你要处理字符转义问题，这就是为什么攻击者的“单引号”方法可以在前面成功的原因。如果你使用PreparedStatement，不仅可以提高性能（请参阅 49 项），但是PreparedStatement机制必须适当地处理作为输入参数而被传入的输入的转义问题，所以现在我们的登录servlet看起来像这样：

```
Connection conn = getConnectionFromSomewhere();

String SQL = "SELECT * FROM users " +

           "WHERE uid = ? and password = ?";

PreparedStatement prep = conn.prepareStatement(SQL);

prep.setString(1, request.getParameter("username");

prep.setString(2, request.getParameter("password");

ResultSet rs = prep.executeQuery();

// Rest as before
```

现在，假设驱动不允许注入攻击（与其它的相比，这是一个快速而有效的解决办法—确保你已经针对这种可能性对你的驱动进行了测试；请参阅第 49 项），那么一个 SQL 命令注入攻击将导致一个 SQL 异常。

然而，不仅仅是 SQL 命令有这个问题，在任何数据被传送到外部系统去执行的时候，比如，创建一个由外部 Shell 执行的命令字符串，你又要回去关心用户输入的验证问题了。幸好，JDK1.4 引入了对字符串正则表达式的支持，使得这种验证变得简单了。很容易用正则表达

式来保证一个给定的字符串不含有可能引起注入攻击的转义字符。

顺便说一下，如果你的站点允许用户使用HTML在页面上添加注解，你就有可能遭受交叉站点脚本攻击（OWASP 第 4 种），这种攻击是指一个攻击者能够将恶意的HTML输入到你的站点，在用户浏览你的站点时对用户的浏览器进行攻击。这可不是建立良好的客户关系的好方法。（*《编写安全代码》*给出了一种使用regexp来检查交叉站点脚本攻击的例子[Howard/LeBlanc, 430]。）

在某些方面，如果 Java 支持“脏”字符串（"tainted" string，比如，来自于外部 JVM 的字符串）的想法就好了，Perl 这样做了，但 Sun 和 Java Community Process 都没有显示出将把它加入到 Java 中的迹象。如果你的开发人员训练有素，你可以创建一个 TaintedString 类，就像这里显示的那样来处理各种数据检验，用它来屏蔽所有客户端输入：

```
public class TaintedString

{

  private final String endUserData;

  private boolean validated = false;



  public TaintedString(String input)

  {

    endUserData = input;
```

```
}

public void validateForXSS()
{
  validate(new XSSTaintValidator());
}
public void validateForSQL()
{
  validate(new SQLTaintValidator());
}
public void validate(TaintValidator v)
{
  try
  {
    v.validate(endUserData);
```

```
    validated = true;
  }
  catch (SecurityException secEx)
  {
    validated = false;
  }
}

public String getString()
{
  if (validated)
    return endUserData;
  else
    return null; // Or throw an Exception,
                 // whichever you prefer
```

```
}


public interface TaintValidator
{
  public void validate(String data);
}


public static class XSSTaintValidator
  implements TaintValidator
{
  public void validate(String data)
  {
    // Run through validation scenarios to
    // make sure data doesn't contain an XSS
    // attack; if it does,
```

```
    // throw a SecurityException
  }
}


public static class SQLTaintValidator
  implements TaintValidator
{
  public void validate(String data)
  {
    // Run through validation scenarios to make
    // sure data doesn't contain an SQL injection
    // attack; if it does,
    // throw a SecurityException
  }
}
```

```
}
```

这儿的想法是一旦 String 被提交给 TaintedString，它必须被验证，既可以通过已有的针对特定攻击的验证方法（比如交叉站点脚本和 SQL 注入就是两个例子），也可以通过实现 TaintValidator 接口的定制的策略对象来验证其他的恶意输入。假设验证成功了（比如，TaintValidator 的验证方法没有抛出安全异常），验证标志就被设为“true”，我们可以通过 getString 方法获得输入。如果验证失败了或根本没有被调用，getString 就会返回 null（或者，如果你喜欢，可以抛出一个异常）。

然而，适用这种方法的难点在于，开发人员必须在进一步使用用户的输入之前将它传给 TaintedString。因为不管是 Java 语言还是 JVM 本身都不支持 TaintedString，所以在代码评审和源码扫描之前我们并不能保证开发人员的确这么做了。但是，如果开发人员能够强制自己使用 TaintedString 来输入，他们就可以避免很多麻烦，在我的书中，这是一个很好的动机。

噢，对于那些牢记了第 51 项，并使用胖客户端作为前端以及某种可选择的通讯层与服务器进行数据交换的人，别以为你在这里就不需要注意了。Web 服务正在迅速转向允许胖客户端应用和后台中间件系统进行交流，对于老谋深算的攻击者，基于 HTML 的 Web 应用和基于 XML 的并没有多大差别。事实上，你必须对 Web 服务更加小心，因为清除出于安全入侵企图而到来的 Web 服务数据的行为，对于大部人 Web 服务开发人员来说依然是不可见的。不管是 HTML 格式的，XML 格式的，还是 RMI/JRMP 对象序列格式，输入仍旧是输入。如果它来自进程之外，那么就使用所有你可以想到的办法去验证它，否则，总有一天你会后悔的。（叮！“呃，嘿，CEO 先生，……会谈？和法律小组？五分钟？”（倒吸一口凉气）“当然，先生”）

不管你的前端是什么，请记住，如果每次你都是直接从用户处得到输入并将它用于数据库查询（或者其他与后台系统的交互；我确信 SQL 注入攻击是唯一一种最普遍的源码脚本注入攻击方法，但不是唯一的一种可能的方式），那么你就需要比攻击者更聪明地处理它，这是非常不切实际的假设。应该依赖你所使用的各种软件中的机制，比如 JDBC 驱动的转义能力，来保护你自己不受恶意用户输入的侵犯。在你阅读了第 60 项后，你当然会假设所有的用户

输入都在某种程度上怀有恶意。

第 62 项：打开平台安全机制

自 JDK 1.0 起，Java 的一个主要优势就是它在语言与环境中直接建立安全机制的思想。SecurityManager，及其在 JDK 1.2 之后的继任者 AccessController，是 Java 环境中会触发安全检查的运行期行为以及各种环境行为的基础部分。例如打开一个文件、通过套接字（socket）连接服务器、读系统属性等等。

而为了使这种状况更加完善，从 JDK 1.2 开始，经过扩展与公开 Java 平台内的安全机制，现在允许非 Sun 公司的开发人员编写自己的 Permission 类（代表敏感的行为）和自定义的 Policy 类（通过这个类可以确定并建立权限）。Java 代码不再受像 applet 沙盒那样笨拙的安全环境的约束了。感谢 JDK 附带的基于文件的 Policy 缺省实现，现在，开发者能够做非常多细粒度的控制，在 JVM 中对允许做什么事情进行精确地控制，然而，更令人遗憾的是，不仅大多数 J2EE 开发者不理会它，而且大部分 J2EE 规范与许多 J2EE 的容器实现也会忽视它。

对许多企业级Java开发者而言，Java安全管理器（Java Security Manager）总是令人感到害怕、厌恶，从而尽一切可能不去招惹它。大多数开发者是在RMI代码中第一次遇到SecurityManager的。但更糟糕的可能是，突然之间，到处都抛出SecurityException实例，这时你才意识到SecurityManager，而这通常发生在将应用演变为产品的阶段。于是，开发人员很快就会学到如何设置“全部放行”的策略文件：

```
grant

{

  permission java.security.AllPermission;
```

```
}
```

当然，这只是解决暂时的问题，如果安全策略说将 AllPermission 赋予所有事物，那么 SecurityManager 实际上就无法拦截任何东西，而现在我们就可以幸福地生活了，对吗？

我们并不能批评这些可怜的 Java 程序员，因为任何与“安全”这个词扯上关系的东西，都会引出非常多令人焦虑的事情。当情况危急时，CEO 就紧跟在你背后，要求昨天就应该交付的应用，此时再纠缠于 Java 策略文件的语法绝对不是你希望做的事情。

然而，问题是在这么宽大的许可策略下运行 Java 代码，会很有效地放任攻击者自由通过，而他（她）会设法利用你的代码来做坏事。关于最低权限原则（请参阅第 60 项）就讲到了这一点。由于 Java 安全管理器无法再强加限制，所以当你的 servlet 或 JSP 完成了对 JVM 中某个东西的访问后，攻击者可能会突然开始打开套接字，覆写本地文件系统中的文件，并恶意地造成其他损失。最令人丧气的是，这正是安全管理器被设计用来防范的事情。

举个例子，让我们先停下来看一看 servlet 容器。从企业的观点，每个部署在 servlet 容器中的 Web 应用都应该是孤立的、自足的（self-standing）应用，它应该完全不知道容器中是只有它一个应用，还是它与其他一打（或上百个）Web 应用部署在一起。从互联网服务提供商（ISP）的观点来看，这也绝对有必要，否则我的 Web 应用将能够调用你的 servlet 的方法，这可是一个公认的糟糕想法。

如果我写的servlet以一个采用相对路径的文件名作为输入，并将该文件作为响应的内容通过网络返回，会发生什么呢？虽然看起来没有什么毛病，但是如果我给它一个相对的文件名（例如像这样../../../../etc/passwd），该相对文件名将沿着目录树一直向上，直到抵达安装了所有Web应用的父目录，然后打印出目录列表，这将导致发生什么呢？或者我“向上并转向”，进入其他Web应用的部署目录，打印出其下的web.xml文件又将会导致什么发生呢？或者打印密码文件又会怎样呢？

如果servlet容器在过分宽泛的Java Policy实现之下运行，则容器根本无法阻止我。当然，如果我通过servlet API请求该文件，例如ServletContext的

getResourceAsStream方法，那么servlet容器就能够拦截该请求，知道我的请求超出了我的Web应用的范围，并拒绝它。servlet容器确实可以做到这一点，虽然规范并没有要求。然而，如果我使用前面的相对路径创建一个File对象，谁能阻止我呢？servlet容器并没有取代标准的Java运行期类库，而且由于Java安全管理器不经审查就批准了所有请求，因此我就有了一扇窗户，能够进入安装在系统上的其它Web应用。

打开安全管理器是件很简单的事情：当执行 Java 启动器（java.exe）启动 J2EE 容器时，通过-D 选项设置 java.security.manager 系统属性即可。无需再设置别的东西，只需在启动容器的脚本中添加 JVM 命令行参数-Djava.security.manager 即可。这样就能打开安全管理器，它将使用存储在 JRE 的 lib/security 目录中 java.policy 文件作为 JVM 的安全策略。如果你的容器正使用独立的 JRE 实例（请参阅第 69 项），那么就可以通过直接修改该文件来改变安全策略。

实际上，确实可能需要进行许多修改。一般说来，J2EE 环境总是假设自己运行在一个关闭了安全管理器的 JVM 中，因此打开安全管理器很容易产生 SecurityException 异常。例如，J2EE 假设容器可以在任何时候访问系统属性，然而，由于访问系统属性的特权是一种受约束权限，所以需要在策略文件中显式地授予容器该权限。同样的，如果因为某种原因你的代码需要使用网络（例如连接到数据库），就需要授予其套接字权限等。

考虑到你必须制定一个安全策略以精确地描述需要授予容器哪些权限（与你的构件代码正相反），而这将花费大量时间，如果连你的容器供应商都没有为你提供安全策略的话，为何你要为此烦恼呢？我们都有太多的事情要做，根本没有足够的时间来做这件事，为什么浪费时间搞什么安全策略呢？一定有某种合理的回报，对不对？

先做个假设，假设你的 Web 应用在你公司的 DMZ 上的 servlet 容器中运行，它要连接到一个数据库，该数据库运行在名为 dbserver.mycompany.com 的服务器上。你已经建立了一个安全策略，只允许到那台机器的连接，而不允许其他连接（permission java.net.SocketPermission "dbserver.mycompany. com", "connect"）。现在，如果一个攻击者试图攻击你的 servlet 以执行任意的 Java 代码，尽管他能得到线程的控制权，但当他试图打开其他数据库服务器的连接时，他将与安全管理器冲突，然后就会

抛出 SecurityException 异常；如果他试图打开一个新的套接字以某个随机的很大的数字作为端口进行连接，也同样会抛出异常；如果他试图向你部署的 Web 应用之外的目录写文件，也会抛出异常。

打开你的 J2EE 容器中的安全管理器，即便只是使用它也绝对是物超所值，特别是如果供应商没有为自己定义安全策略的话。（如果你的供应商也没有，那就考虑换一个供应商的产品，或者对供应商施加压力要求提供一个。）不过，当配合 JAAS 使用的时候，如同第 63 项中描述的那样，Java 平台安全模型的好处将变得更清楚。

第 63 项：使用基于角色的授权

认证很容易理解：就是某人试图向系统证明他（她）的身份的行为，一般是通过提供密码（一个“秘密”），该密码理论上只有这个人知道。登录表单是 Web 应用中有名的习惯用语，所以我们就不对此多唠叨了，不过稍候我们仍将回到这里。

而另一方面，授权则是一个模糊得多的概念。正式的说，授权就是搞清楚系统中什么行为可以执行。例如，在普通的Java安全管理器之下，授权是基于代码从何处被载入来判断是否同意它执行某些“敏感”行为的（例如启动线程，打开连接等等）。这在《Inside Java 2 Platform Security》[Gong]与《Java Security》[Oaks]两本书中都有很好的描述。然而，在标准的Java安全平台内，并没有关于基于用户的安全概念的讨论。

不过，为了满足自身的需要，大多数应用至少会有一个粗粒度的授权思想。几乎没有几个应用会将所有用户一致对待。一般至少会有 3 类用户：普通用户、游客、以及管理员。对许多应用而言，其授权模型极为复杂：系统中有几十个不同的可能行为，有成百上千的用户，而每个人对于那些行为都有不同的访问权限。如果我们亲自为此进行编码的话，那么表示层（servlet 或 JSP）将变得非常复杂。

安全代码会与系统中的所有东西都扯上关系，这就是横切关注点（crosscutting concern），而这些东西与系统的业务逻辑毫不相关。因此，J2EE 规范，特别是 servlet

和 EJB 规范，就在部署描述符中加入了配置选项，试图从开发人员的肩膀上卸掉此负担。

你想看看在现实世界中授权机制有多么复杂吗，只需想象一下有着数以万计用户，几百种不同访问权限（在中型应用中很常见）的系统。每向系统中添加一个用户，系统管理员就必须清楚地为该用户指定访问权限，当访问权限的数量上升时，这将耗费越来越多的时间。如果你采用后续措施，或者以新版本的方式向系统添加新的权限，愿上帝保佑你的系统管理员吧：现在，他们不得不搜寻整个用户数据库，清楚明白地为每个需要该新增权限的用户添加权限（因为你的缺省假设是他们没有权限。请参阅第 60 项）。很快，这将令你的系统管理员对你非常生气，完全无助于建立开发者与管理员之间的良好关系。

为避免管理员与开发人员因为安全问题而发疯，我们的一个办法是使用基于角色的授权系统，将权限（执行敏感行为的权利）分派给抽象的用户类别，称为角色。这种思想是在用户与授予他们的权限之间引入一个间接层。不是将权力直接分派给用户，而是分派给角色，然后为用户指定一个或多个角色。现在，权限检查变得简单多了，因为我们只需要考虑角色这个小得多的集合。用户管理也更容易了，因为用户只是简单地指派角色，而不是直接分派访问权限了。如果在稍后的阶段，或者新版本中引入了新的访问权限，可以直接将新增的权限分派给已经存在的角色（这意味着无需修改用户数据库），或者，如果必要的话就新建一个角色。

J2EE的servlet和EJB容器包含有安全部分，具体地说是授权与访问控制。一旦用户通过认证进入了系统（即通过了servlet容器内置的j_security_check行为或者在EJB bean的Home查询期间通过JNDI属性进行了认证），容器就能够通过供应商特定的用户数据库确定用户已配置的角色。因此，通过以下两种方法，程序员能够批准或限制对URL资源（servlet或JSP）与bean方法（在EJB上）的访问：（1）在J2EE构件部署描述符中清楚地列出这些限制；（2）使用两方面环境提供的可编程API，最特别的是isCallerInRole方法。使用部署描述符可以进行粗粒度的安全控制，例如，如果对JSP进行粗粒度的安全控制，意味着批准或拒绝用户对整个页面的访问，而使用可编程API则是一种更细粒度的安全控制方式。

不过，尽管 servlet 和 EJB 提供了简单灵活的方法，它们却忽视了授权控制的一个很重要

的方面：对 J2EE 势力范围之外的资源的访问控制。例如，打开底层文件系统中的一个文件，这需要获得恰当的 Java 平台安全级别的许可。然而，由于 J2EE 安全系统与 Java Security Manager API 完全正交，如果在打开文件的代码前不进行显式的检查，就毫无办法根据终端用户的角色授予其文件系统的访问权限：

```
// EJB bean

EJBContext context = ...;

if (context.isCallerInRole("admin"))

{

  // Do whatever admin-only processing is necessary

  //

}



// servlet

HttpServletRequest request = ...;

if (request.isUserInRole("admin"))

{

  FileOutputStream fos = new FileOutputStream(. . .);
```

```
  // Do whatever admin-only processing is necessary

  //

}
```

在很大程度上，这是多层结构的一个糟糕的典型。首先，如果攻击者能够以某种方式找到你代码中的漏洞，那么底层平台安全模型（而它就是明确用来阻止这种攻击的，且这里确实一一走访了所有调用源进行了完整的安全检查）将无法阻止攻击。然而，更重要的是，对于你团队中的开发人员来说，非常容易忘记在打开文件的代码之前进行 J2EE 安全检查，特别是在凌晨 3 点的时候，而你正打算明天就交付应用。或者，还有更糟糕的，“我知道我们应该在这里放一些保证安全的代码，但系统现在可以工作，所以让我们迟一点再修正它吧，攻击者并不一定知道这个漏洞，对吧？”

我们真正希望看到的是，J2EE 基于角色授权的安全模型与底层平台安全模型的结合体，那样我们才能从兼顾两方面的安全模型中获益，以保证我们的应用的需要。遗憾的是，不仅 J2EE 的基于角色的授权模型无法与平台安全模型挂钩，而且直到 JDK 1.3，平台安全模型仍没有基于用户授权的概念。

在JDK 1.3 与JDK 1.4 之间，Sun引入了一个新的API：Java JAAS。它是对这种异乎寻常的忽视的纠正。简言之，JAAS扩展了Java标准安全模型，使之包含基于身份的授权，还包含了源代码。总之，标准的平台安全模型现在知道了基于用户的授权，这意味着，将基于角色的授权整合进平台安全模型现在已经成为可能。也许你想知道这一条建议是否还值得读下去（“哼，J2EE安全机制对我而言已经足够了”），那么还有一个新的JSR，《Java Authorization Contract for Containers（JSR 115）》，它有效地将J2EE安全与底层的JAAS以及Java平台安全模型整合起来，所以你迟早还是要回到这里。幸运的是，一旦你克服了某些基本问题，JAAS的使用并不是很难的事。（在其它地方，例如《Inside Java

2 Platform Security》[Gong]深入讨论了JAAS，因此在这里我只讨论使用它时必需的一些高层的步骤。）

设置 JAAS 安全机制包含两个元素。第一，因为 JAAS 支持可插拔的认证模型，即它可以同时支持多种认证与授权系统，所以 JAAS 需要知道将使用哪一个可插拔模块。这称为 JAAS 配置，虽然可以通过编程进行配置（参考 javax.security.auth.login.Configuration 类），但通常都是使用缺省的 Configuration 来实现，它读取一个可以被 JVM 系统属性 java.security.auth.logon.config 识别的文本文件。该文本文件的内容看起来就像下面这样：

```
MySystem

{

  com.tagish.auth.RdbmsLoginModule Required

    driver="org.gjt.mm.mysql.Driver"

    url="jdbc:mysql://localhost/userdb?user=root";

}
```

（在这个特别的例子中，我们使用了 http://free.tagish.net/jaas 提供的开源的 LoginModule，它使用关系型数据库作为用户数据库。我们可以建立自己的 LoginModule 实现，不过那超出了这一节的范围。若希望了解其中的细节，可以参考《Java Security》[Oaks]，或者你可以通过 Google 搜索互联网上的文章和论文，它们将非常详细地告诉你应该如何操作。）

设置 JAAS 安全机制的第二部分就是将权限实际分派给用户角色，我们稍候会介绍这方面的

内容。

现在，配置好了 JAAS（记住在启动容器的命令行中加入系统属性），在用户进入系统之前，我们首先通过创建 LoginContext 对用户进行认证，它引用我们先前定义好的配置块：

```
// In the servlet acting as the recipient of the

// login.jsp page

//

final String username = request.getParameter("uid");

final String pwd = request.getParameter("pwd");

LoginContext loginCtx =

  new LoginContext("MySystem", new CallbackHandler()

    {

      public void handle(Callback[] callbacks)

        throws IOException,

              UnsupportedCallbackException

      {

        for (int i=0; i<callbacks.length; i++)
```

```
        {
          Callback cb = callbacks[i];
          if (cb instanceof NameCallback)
            ((NameCallback)cb).setName(username);
          else if (cb instanceof PasswordCallback)
            ((PasswordCallback)cb).setPassword(pwd);
          else
            throw new
              UnsupportedCallbackException(cb);
        }
      }
    });
loginCtx.login();
Subject userSubject = loginCtx.getSubject();
HttpSession session = request.getSession(true);
```

```
session.setAttribute("userSubject", userSubject);


// userSubject now represents the user
```

此处立刻发生了好几件事情。我们的目的是调用 LoginContext 的 login 方法，不过这也意味着，我们必须向作为 LoginContext 组成部分而定义的各种模块出示认证的证明（能够从前面定义的配置文件中识别出它们）。因为 JAAS 不必提前知道登录需要做些什么，所以它使用回调的方式收集信息。JAAS 不希望对它运行的环境做任何假设。一般而言，在交互式应用系统中，回调能够弹出某种提示给用户，以收集必要的信息，但是在 Web 应用中这显然行不通。所以，JAAS 需要一个实现了 CallbackHandler 接口的对象，将它传给 LoginContext 构造器，然后 JAAS 调用该对象来收集必要的信息。在此处的代码中，我们提供了一个匿名内部类的实现，它将这个 servlet 前面的 JSP 表单收集到的字符串简单地返回，并且一旦各种各样的认证系统要求用户名与密码之外的其它信息时，它就抛出异常。

成功构造了 LoginContext 对象之后，我们就可以直接调用 login，让 JAAS 依据配置文件中指明的登录模块来认证当前用户。假设成功了（如果不成功它会抛出 LoginException 异常），它就返回一个 Subject 实例，实质上代表了一个用户。缓存此 Subject 对象（基于某种原因该对象是可序列化的），将它存储在 HttpSession 中或者无论什么机制中都行，只要是你使用的是为每用户保持数据的机制即可。（这就是我们必须拥有某种每个用户有各自状态的一个例子，因此我们必须忽略在第 39 项中的争论，将它存储在 HttpSession 或等价的东西中。）或者，缓存 LoginContext 本身，因为你总是能够通过调用 getSubject 从 LoginContext 中取得 Subject，而且 LoginContext 还有另一个方法，logout，它可以关闭与认证用户相关联的所有资源。（事实上，Sun JDK 中带有的 LoginContext 实现没有终结器（finalizer），因此除非你调用 logout，否则任何分配给此 Subject 的外部资源都作为 login 的一部分，永远也不会被清除。你可以做一些性能测试，以确定释放最后一个指向此 LoginContext 的引用不会造成资源泄漏。）

现在，我们已经将 Subject 存入了 HttpSession，此后该用户每次访问 JSP 或者 servlet 时，我们就能够使那个 servlet 在整个 HTTP 请求处理过程中都扮演该用户：

```
// Inside the controller servlet or JSP

//

Subject userSubject =

   (Subject)session.getAttribute("userSubject");

Subject.doAsPrivileged(userSubject, new PrivilegedAction()

  {

    public Object run()

    {

      // Do the usual servlet processing logic;

      // inside here, it will be done under

      // userSubject's rights

      //

    }

  }, null);
```

```
// Out here, code will execute as the container

//
```

调用 Subject.doAsPrivileged 方法实质上是告诉正在运行的线程，要求其安全策略改变一下，以模拟那个用户（稍后我们会看到如何配置其可用的权限集）的安全策略，因此，run 方法体内的所有调用都将在 Java 平台安全语义下执行，而 run 方法会在这个扮演者被建立的时候由 Subject.doAsPrivileged 调用。这说明，如果 servlet 想打开文件系统中的某个文件，而该用户（或者更精确一点，是该用户所属的角色）没有这样的权限的话，将会抛出一个标准的 Java SecurityException。（在一个 JSP 页面中，与其使用直白的 Java 代码，不如写一个 JSP 自定义的标记处理器（tag handler）来扮演检查更好一些，或者如果有可用的就下载一个，不过那就是另一回事了。）

为一个指定的用户角色配置授权策略，需要标准的Java安全策略文件（可在$JRE/lib/security/java.policy找到），并以正规的Java平台安全的形式建立一些授权组块。注意，你可以直接编辑这个JRE之外的文件，因为你已经为此容器建立了一个独立的JRE，正如第 69 项建议推荐的那样。如果不是这样，你将需要创建一个自定义的策略文件，并通过另一个系统属性向它指明servlet容器。此策略文件看起来应该像这样：

```
// Java.policy file

//



// Permissions granted to all users
```

```
//

grant Principal * *

{

  permission java.security.BasicPermission

    "menuActions","logout";

  // Other permissions here

}



// Permissions granted to users in the "users" role

//

grant Principal com.tagish.auth.TypedPrincipal "user"

{

  permission java.security.BasicPermission "menuActions", "save";

  // Other permissions here

}
```

```
// Permissions granted to users in the "admin" role

//

grant Principal com.tagish.auth.TypedPrincipal "admin"

{

  permission java.security.BasicPermission "menuActions",

    "delete";

  // Other permissions here

}



// And so on
```

（如前所述，Principal类来自于前面提到的开源类库。）由于每个角色都分派了一个权限的组块（由Java Permission对象或其子类[4]表示），所以，现在我们可以在JSP和servlet中写代码测试用户是否具备了恰当的权限，而不用再担心用户是谁，他（她）的角色是什么，也不用为执行超过了实际所需的Permission对象的行为而担心：

```
<%!-- In a JSP page --%>
```

[4] 请查看*Inside Java 2 Platform Security* [Gong]或*Java Security* [Oaks]以了解如何创建Permission的子类，这很容易。

```
Menu options:

<% try {

    AccessController.checkPermission(

      new BasicPermission("menuActions","logout");

  %>

<a href="logout.jsp">Logout</a>

<% } catch (SecurityException secEx)

   { /* We ignore this; they don't have access,

        which is OK */ } %>



<%!-- The above is a bit tedious, so we would

      probably replace it with a JSP custom tag

      handler that would look like the following:

   --%>

<taglibrary:checkPermission
```

```
    type="java.security.BasicPermission"
    name="menuOptions" actions="save">
   <a href="save.jsp">Save</a>
</taglibrary:checkPermission>

<taglibrary:checkPermission
    type="java.security.BasicPermission"
    name="menuOptions" actions="delete">
   <a href="delete.jsp">Delete</a>
</taglibrary:checkPermission>

<%!-- And so on --%>
```

同样地，有关 JAAS 更详细的信息可以从网络以及前面提过的 Java 安全方面的书籍[Gong；Oaks]中找到。不过，作为关于 JAAS 的最后一句评语，请你注意，Sun 的 Win32 JDK 和 Solaris JDK 都包含了 LoginModule 对象，它使用底层操作系统的安全数据库进行认证，千万不要在你的 Web 应用中使用它们。在要求系统管理员对操作系统用户进行管理的情况下，每当一个新用户要使用此 Web 应用时，就需要通过 HTTP 链接，使用 LoginModule

将用户的操作系统密码发送给服务器，因此，应该锁住用户与 servlet 容器之间每一条可能的链接，以防止 Eve（窃听者）监听操作系统的密码。此外，如果攻击者猜到了一个系统（操作系统或者 Web 应用）的密码，那么他（她）就能够很容易地攻击另一个系统，因为它们未采用深度防御方式（请参阅第 60 项）。

第 64 项：使用 SignedObject 以保证序列化对象的完整性

通常而言，我们在内心深处并不信任那些通过媒介传送而来的可序列化的数据（顺便说一下，应该是对各种数据都不信任，请参阅第 60 项）。例如，如果某个系统无需提供正确的证书，那么可能会被攻击者骗取的认证或授权信息就是获准进入此系统的一种非常强有力的方式。在任何认证或授权对象被传递到或被存储到磁盘的情况下，我们都应该证实此对象来自恰当的服务器。实质上，我们需要确保这个令牌不能被伪造，只要对它有任何修改，无论是否善意的，都必须立即发现，这样才能丢弃该令牌。

换而言之，就是要求你的对象向你证明它们的可靠性。

如果你已经阅读过第 6 项建议，并且为了使用 Java 对象序列化规范（请参阅第 71 项）中定义的挂钩点（hook point），你仔细浏览过该规范的话，那么，你的第一个反应可能是通过实现 readObject/writeObject 来提供自定义的序列化行为。在对象第一次被序列化时，使用发行方的 PrivateKey（来自 javax.security API）对信息加密，并在反序列化时使用发行方的 PublicKey，它随序列化的对象一起被传送。你的思路正确，不过有两件事阻碍着你：（1）如此实现乏味且困难；（2）除非你是一个安全专家，否则在此过程中你不可避免地会留下漏洞。

幸运的是，通过 java.security 包中一个不太出名的类 SignedObject，Java 已经提供了解决的办法。这个类将一个 Signature 对象和一个序列化了的对象（实际上是原本的可序列化对象的一个副本，因此，对原序列化对象的修改不会反应到此 SignedObject 中的副本上）包装起来，并提供了简易的方法使用 PublicKey 来验证对象的可靠性。

让我们把它翻译成代码，想象一下，通过一个普通的 HTTP 链接，过程优先持久化引擎（请参阅第 42 项）正在客户端与服务器之间传递序列化的 RowSet 对象。数据不是机密的，但是当它来自服务器时我们需要验证它的可靠性，否则我们收到的回复很可能来自某个中间人，而他正试图伪装成服务器并影响最终结果。

为了消除这种可能，我们创建一个 PrivateKey 实例并嵌入到服务器端，序列化 RowSet 将其存入 SignedObject 中，然后使用此 PrivateKey 对它签名：

```
PrivateKey privateKey = getPrivateKey();



RowSet data = ...;



Signature signingEngine =

  Signature.getInstance("SHA1withDSA");



SignedObject signedObject =

  new SignedObject(data, privateKey, signingEngine);



// Serialize the signedObject and ship it
```

```
// over the wire
```

PrivateKey 通常以字节流的形式被嵌入到服务器软件资源的某处（做为私钥，它一般会以某种方式伪装起来，以避免被轻易识别出来）。在序列化的时候，signedObject 实例会包含该私钥的数字签名，而 signedObject 的可靠性可以使用 Signature 实例和对应的公钥相进行验证，此 Signature 实例需要与一开始对数据进行签名时采用的 Signature 相匹配，同样地，该公钥通常作为客户端软件的一部分被嵌入其中：

```
PublicKey publicKey = getPublicKey();



Signature verificationEngine =

   Signature.getInstance("SHA1withDSA");



SignedObject signedObject = ...;

   // Deserialize from server socket



if (signedObject.verify(publicKey, verificationEngine))

{

  RowSet data = (RowSet)signedObject.getObject();
```

```
    // We know data was issued under the private key, and
    // that it wasn't tampered with
}
```

作为实验，我们使用 java.util.HashMap 代替 RowSet，将 SignedObject 序列化到磁盘，然后通过十六进制编辑器修改此 SignedObject 的内容。（SignedObject 只提供了完整性，不保证机密性；机密性由 SealedObject 负责，请参阅第 65 项。）在验证修改过的数据时，签名将不匹配，证明这些数据在签名之后被篡改了。更重要的是，除非攻击者能够替换客户端软件的公钥，使用他（她）自己选择的公钥，否则我们总能确保从服务器收到的数据确实来自该服务器。

对不了解 SignedObject 目的或不了解需要在使用对象之前先做验证的程序员而言，直接使用 SignedObject 有时太过繁重了。为了向客户端隐藏那些细节，可以直接扩展 SignedObject。例如，建一个已经签名的授权上下文（authorization context），它可以通过网络传递，再创建一个 SignedObject 的子类，它包含一个 JAAS 的 Subject 作为其可序列化的内容，参见下面的代码：

```
public class AuthorizationToken extends SignedObject
{
  private static Signature engine =
    Signature.getInstance("SHA1withDSA");

  public AuthorizationToken(PrivateKey privKey,
```

```
                        Subject subject)
  {
    super(subject, privKey, engine);
  }

  public Subject getSubject(PublicKey pubKey)
  {
    if (verify(pubKey, engine))
      return (Subject)getObject();
    else
      throw new
        AuthenticationFailedException("Spoofed!");
  }
}
```

或者还有另一种方法，不需要继承 SignedObject，你可以简单地对它进行组合（将它做

为一个成员属性，把调用直接转给它)；以上两种方法都工作得相当好。许多开发人员喜欢基于继承的方式，因为它省掉了许多录入工作，不过在这个特定的例子中两者差不多。

顺便说一下，生成一对 PublicKey/PrivateKey 与使用 KeyPairGenerator 类一样简单，KeyPairGenerator 也是来自 java.security 包。下面的代码是一个实用工具，将成对的密钥写到.publickey 和.privatekey 文件：

```
import java.io.*;

import java.io.*;

import java.security.*;



public class genkeypair

{

  public static void main(String[] args)

    throws Exception

  {

      KeyPairGenerator kpg =

        KeyPairGenerator.getInstance("DSA");

      KeyPair kp = kpg.generateKeyPair();
```

```
PrivateKey privKey = kp.getPrivate();

PublicKey pubKey = kp.getPublic();



try

{

  ObjectOutputStream oos =

    new ObjectOutputStream(

        new FileOutputStream(".privatekey"));

  oos.writeObject(privKey);

  oos.close();



  oos = new ObjectOutputStream(

         new FileOutputStream(".publickey"));

  oos.writeObject(pubKey);

  oos.close();
```

```
      }

      catch (Exception ex)

      {

        ex.printStackTrace();

      }

  }

}
```

请注意，你绝对不应该将这些密钥以文件的形式随软件一同发布。否则，即便是最初级的攻击者也很容易就能用他自己的密钥替换掉客户端与服务器上的密钥文件。这里最主要关注的当然就是密钥本身。持有服务器私钥的攻击者可以伪装成你的服务器。如果攻击者能够换掉在客户端系统中的公钥，从而使用他（她）自己选择的公钥的话，这个攻击者就能拦截客户端与服务器之间的通信，将服务器的数据用他（她）自己的私钥重新签名，而已被攻击的客户端会认为这些数据将是可靠的。（这就是为什么数字签名通常伴随着任意的公钥，由于我们相信证书的发行方，因而证书可以作为验证公钥可靠性的一种方法。）

还要注意，如果需要多重签名的话，`SignedObject` 实例可以嵌套使用（将一个 `SignedObject` 作为另一个 `SignedObject` 的数据使用）。验证这种对象唯一棘手的地方是，怎样才能确保采用了与签名相逆的顺序来验证签名。（有报告指出，新版本的 `SignedObject` 将接受多重签名，不过暂时将它们堆在一起也不是太丑陋，因为如果需要的话，签名的顺序也可以作为数据的一部分一同传递。）

再一次提醒您，使用 `SignedObject` 的目的仅仅是为了验证那些通过传递、存储、序列化

的数据的可靠性，以确保它们没有被篡改，或确实来自你所期望的地方。无论如何，SignedObject 并没有为数据提供机密性，数据的机密性是 SealedObject 的责任，我们将在第 65 项中学习 SealedObject。

第 65 项：使用 SealedObject 以保证可序列化对象的机密性

我们不仅需要确保序列化了的数据在通过不可靠的媒介被传递时（请参阅第 60 项）不被篡改，而且，除了计划中的接受方以外，我们经常还需要确保没有人能够看到这些数据的内容。通常保证数据的机密性比保证数据的完整性更有必要。保证机密性是 javax.crypto.SealedObject 类的职责，其工作方式与第 64 项中讨论的 SignedObject 相似。

此时，不再是使用成对的PublicKey/PrivateKey了，SealedObject需要一个双方共享的SecretKey。SealedObject使用对称密钥对数据进行快速加密解密，因为非对称密钥的密码体系（公钥/私钥密码体系）执行起来相当耗时。以双方都安全的方式交换密钥以获得SecretKey，避免Eve(窃听者)或Mallory（恶意的中间人）搅乱密钥交换的过程，这个话题超出了本项的范围，因此，现在我们假设SecretKey已经建立好了。(不过请注意，可供你选择的一个方法是复制SSL机制，它在每个SSL连接的接收方之间建立一个新的会话密钥[5]。另一个方法就是附录F中的Java密码机制扩展(Java Cryptography Extension)参考指南，包括部分标准JDK文档，可以使用KeyAgreement等类，在双方之间为共享的秘密的会话密钥进行谈判。)

与 SignedObject 一样，任何 Searializable 的对象（在这里它将与通过 SecretKey 构造出的 Cipher 结合）都可用来构造 SealedObject 实例。这个 Searializable 对象以复制值的方式进入 SealedObject，并且被加密。现在，可以将 SealedObject 序列化了，此处采用的是本地文件系统中的一个文件，其中的数据可以躲开窥探的视线，除非它们碰巧也有相同的 SecretKey。

[5] 通常而言，重新实现已经存在的安全防卫设施绝对是个糟糕的点子，因为你非常可能不小心在新的实现中留下了漏洞，而这些漏洞会被攻击者利用。如果你想换掉自己的密钥交换协议，请确保你的威胁模型被全部覆盖了，而且在尝试这么做之前，作为对你自己的测试，请确定你能够描述出SSL是如何工作的。

```
String s = "Greetings";



KeyGenerator keyGen = KeyGenerator.getInstance("DES");

SecretKey key = keyGen.generateKey();

Cipher cipher = Cipher.getInstance("DES");

cipher.init(Cipher.ENCRYPT_MODE, key);



SealedObject sealedObj = new SealedObject(s, cipher);



ObjectOutputStream oos =

  new ObjectOutputStream(new FileOutputStream(".secret"));

oos.writeObject(sealedObj);

oos.close();
```

取得 SealedObject 内的数据很简单，只需调用 getObject 方法，并传入 SecretKey：

```
ObjectInputStream ois =
```

```
  new ObjectInputStream(new FileInputStream(".secret"));

SealedObject sealedObj = (SealedObject)ois.readObject();

ois.close();



String data = (String)sealedObj.getObject(key);
```

虽然将数据存储到文件系统中也是 SealedObject 的一种用途，但更常见的方式是在 JMS 消息队列中传递 SealedObject 实例，特别是，当数据非常敏感或者如果系统管理员可以看到队列中的消息时更是如此。实际上，对于保证 JMS 消息的机密性，这种方法的可扩展性更好，到目前为止，我们一直都没有依赖于消息中间件来加密所有传输的消息，那很可能太过分了。取而代之的是，我们只保护敏感的有效负荷（消息本身）。只需将 SealedObject 作为 ObjectMessage 的有效负荷进行传递即可。

与其表兄 SignedObject（请参阅第 64 项）和 GuardedObject（请参阅第 66 项）一样，为了使用户不必了解实际的安全机制，我们可以开发 SealedObject 的子类，引入更强类型的 API 给客户使用。

你也许想知道，既然其他机制为实际的加密解密过程提供了更好的控制，例如成对的 CipherInputStream/CipherOutputStream，为什么我们还要为 SealedObject 如此费心。简言之，因为，当安全被包括在内的时候，你对其实现的实际控制越少越好，否则很容易成为具有漏洞的手工制造的安全机制。而使用 SealedObject，你只需简单地关注于将可序列化类型的对象存进其中，其他的事情由 SealedObject 的实现自己处理。那些事情正是你不想亲自动手实现的东西。

不过，通常而言，仅采用加密的形式还不够。加密本身并不能确保对象不会被篡改，如果

Mallory 能够以某种手段插入数据传输流，他就可以轻易地使用他与 Bob 之间的另一个密钥取代 Alice 一开始采用的那个密钥。基于这个原因，SealedObject 实例经常与 SignedObject 组合起来传送：创建并签名此 SignedObject，然后再用它创建一个 SealedObject。

第 66 项：使用 GuardedObject 以保证对象的存取控制

相信你已经阅读了第 62 项，并且决定利用底层的平台安全模型为你的系统建立授权机制（也许你已经开始采用基于角色授权的 JAAS 了，如第 63 项描述的那样）。你已经创建了自定义的权限类型，也建立了自定义的 Policy 对象，或者写好了策略文件（java.policy）以反映恰当的权限分派。现在，你开始为各种权限的所有权编写一些必要的测试，然后才允许进行敏感操作。但是在这里你会遇到一个小小的障碍：你会得到一个对象，它包装了一个敏感操作，并且需要通过网络进行传递，然而接受方的安全上下文（security context）却可能与你的不同。

思考这个问题：在一个基于角色的安全环境中，一个 Person 对象可以通过请求主体（principal）（换而言之，代表正在执行的代码的用户）要求获得一定的权限，然后才放行对其的各种调用。例如，近来美国立法宣布，某些敏感的个人数据任何人都不能访问，例如社会安全号码。这正是最适合使用基于 JAAS 的权限检查的地方：

```
public class Person

{

  private String ssn;



  // . . .
```

```
  public String getSSN()
  {
    AccessController.checkPermission(
      new SensitiveDataAccessPermission("ssn",
                                        "read"));
    return ssn;
  }
}
```

在此特定场景中，只有这一个方法需要检查，所以走访所有调用源进行安全检查的开销还算合理。

但是，如果对对象本身的一次简单访问也需要许可的时候，会是什么状况呢？例如，在 JDK 类库中，任何种类的文件 I/O 都应该通过某种权限进行管理，这意味着，在理论上，对基于文件的对象（FileInputStream，FileOutputStream 等）的每一个方法的每一次调用都应该进行验证，验证在其安全上下文中，调用线程上的所有 ProtectionDomain 实例是否都具有适当的 FilePermission。如果按照前面讲过的指导原则，那么这些类中的每个方法都应该进行权限检查，以确保其具备适当权限。

遗憾的是，这将导致“许多问题”。

关键在于，走访所有调用源进行安全保障对性能必定有影响，当 JDK 尽力将此影响降到最小的时候，基本的事实却是：线程中建立的每个 ProtectionDomain 都需要进行检查，而 ProtectionDomain 的每一个权限也都需要进行检查，以判断它是否能推导出必需的 FilePermission 。这不可能是省事的操作，更何况每次调用 FileOutputStream.write 或 FileInputStream.read 都需要做此检查，这将毁灭我们对性能的任何要求。

JDK 可以解决这个问题，自从它认识到，如果每个操作都需要安全许可的话，那么只在创建对象的时候进行一次权限检查就足够了，在那之后，我们可以假设，由于调用者必须拥有创建该对象的权限，所以它们一定也具备所需的权限以调用该对象上所需的方法。如此，就只在对象构造器内做一次安全检查，以后就不用管其他方法了。

然而，在一个非常特别又非常普通的场景下，这种避免多重（通常是冗余的）安全检查的高效方法也会失败，那就是序列化操作。请记住，《Effective Java》[Bloch，第 54 项]中指出，一个对象的序列化形式，相当于使用另一个构造器表现出这个类，因为反序列化的过程意味着通过字节数组构建出一个对象，而不是通过正规的 Java 参数。

如果该对象，是在一个具备恰当权限的安全上下文中创建的对象，当它被序列化然后传送给另一个安全上下文（通过网络或者只是序列化到磁盘上然后再反序列回来），而后者不一定能够使用该对象，此时事情就变得很危险了。因为安全检查已经被放到了构造器中，而该对象再也不会调用其构造器了（事实上，Java 的法律也不允许），于是原本不应该能够使用该对象的代码就可以使用它了。

这似乎是考虑不足所致，因为它也有这样的基本前提，从对象被构造开始一直到它被析构为止，在调用者的访问控制上下文中，调用者将一直拥有同样的权限。在 JAAS 出现之前，当对象总是待在同一个 JVM 中时，情况可能的确如此，只要该对象不被另一个 ProtectionDomain 中的代码序列化和反序列化就行。但是现在，JAAS 是 JVM 的一个组成部分了，所以也就不能这样假设了，进行存储控制检查的代码必须要考虑这个问题，它们可能在类库本身的类中，或者在使用类库的客户端代码中。

基于这些原因，当你希望确认只有具备了适当访问权限的调用者才能使用某个对象时，可以使用 java.security.GuardedObject 提供访问控制权限。对于受保护的对象，它会强迫调用者通过方法调用取得访问权限，这能够保证访问控制至少会经历一次检查。

GuardedObject 的使用很像它的表兄 SignedObject(请参阅第 64 项)和 SealedObject(请参阅第 65 项)：创建一个受警卫的对象（它应该是可序列化的，但这并不是必须的，除非你想通过网络传递它），将它与一个“警卫”对象一起传递给 GuardedObject 类的构造器。该警卫对象本身必须是实现了 Guard 接口的类型，这要求它实现 checkGuard 方法。该方法将被用来验证调用者是否具有存储访问的权限。请注意，自定义的 Permission 对象也是可接受的“警卫”，因为依靠基础的 Permission 类实现 Guard 接口与 checkGuard 方法，可以获得 SecurityManager 并调用其上的 checkPermission 方法。

```
public class Person { . . . }



public class PersonPermission extends Permission

{ . . . }



void foo()

{

  Person p = new Person();
```

```
  GuardedObject go =

    new GuardedObject(p,

                  new PersonPermission("read"));

}
```

受保护的对象（这里的 Person）不必是可序列化的，当然，如果我们希望通过网络传递此 GuardedObject 的话，它就应该是可序列化的。

要访问敏感对象，可以通过调用 GuardedObject 上的 getObject 获得，它将依次调用警卫对象的 checkGuard 方法。假设警卫对象是一个 Permission 类型的对象，那么调用者的安全上下文将会受到检查，此时并不是假设创建对象的一方的安全上下文仍然存在。

```
GuardedObject go = getGuardedObject();

Person p = (Person)go.getObject();

  // At this point, if the caller of getObject()

  // doesn't have PersonPermission in its

  // ProtectionDomain, a SecurityException

  // is thrown and access will be denied
```

与 SignedObject 和 SealedObject 一样，为了令 GuardedObject 的使用变得更容易，

我们可以创建 GuardedObject 的子类以及自定义的 getObject 方法，令它包装 getObject 并且返回一个指向敏感对象的强类型的引用。

这使我们有能力确保首先进行对安全上下文的检查，然后 GuardedObject 的接收方才能访问敏感对象。再说一次，这对于消息驱动 Bean 或直接的 JMS 消费者的形式特别有用，因为现在，即使是通过网络访问对象的内容，我们也能够确保调用者具备恰当的安全上下文。例如，我们可以将调用者当前的 JAAS 安全上下文（已确定的 Subject）作为参数，将它作为消息的一部分一起传递：

```
Message msg = queueSession.createObjectMessage();



Person p = new Person(...);

GuardedObject guardedObj =

  new GuardedObject(p, new PersonPermission("read"));



Subject s = loginContext.getSubject();

  // Obtain the Subject out of the authenticated

  // LoginContext



msg.setObjectProperty("subject", s);
```

```
msg.setObject(p, guardedObj);



queueSender.send(msg);
```

然后，当收到消息时，我们可以提取出 Subject，获得 GuardedObject，然后从中抽取出 Person:

```
ObjectMessage objMsg =

  (ObjectMessage)queueReceiver.receive();



Subject subject =

  (Subject)objMsg.getObjectProperty("subject");

GuardedObject guardedObj =

  (GuardedObject)objMsg.getObject();



Subject.doAs(new PrivilegedExceptionAction()

  {
```

```
    public Object run()
      throws Exception
    {
      Person p = (Person)guardedObj.getObject();
        // An access control check is made here; if
        // the Subject doesn't have PersonPermission,
        // a SecurityException is thrown


      // Use p as appropriate; if you got here, you
      // obviously have the necessary permissions
    }
  });
```

当然，一旦取得了该对象并将它存入一个 Person 引用，再对它调用 Person 的方法时就不再进行安全检查了，这使得“只做一次安全检查”的优化成为可能，同时确保进行了安全检查。

还要注意，如果 Person 引用的持有者决定将它交给其他的调用者，而另一个调用者是在一个独立的安全上下文中运行的话，那么最初的问题仍然存在，如果不对 Java 语言与平台

做重大改变的话，基本上没有办法能阻止这样的事情发生。假设你遵循了第 1 项建议，构建了构件以代替对象，你就能够在暴露给客户的接口背后创建动态代理，在一个 GuardedObject 中存储实际的对象，每次都使用 getObject 转发调用。

当然，这样做大体上消除了“只做一次安全检查”优化的可能，而那正是我们一开始追求的。不过请记住，基于运行期掌握的因素，我们可以选择是否要这么做，例如我们可以知道对象是否将在同样的安全上下文中使用。（同时，如同第 71 项中所描述的，通过实现序列化的挂钩，序列化 GuardedObject 中的实际数据，你可以了解到一定的信息。只需小心以避免 GuardedObject 序列化时发生无限递归，它将依次序列化受保护的对象，也就是你最初想序列化的对象。）

第八章 系统

神话是生命之源，当它不觉之间一再重现时，生命就在神话永恒的模式与虔诚的规则之中流逝着。

—— Thomas Mann

J2EE 平台——Java 虚拟机加上类库——包含众多功能，J2EE 的实现依赖于这些功能。例如，servlet 引擎和 EJB 容器的热部署能力完全要归功于底层的 JVM 的类加载器（ClassLoader）架构；如果没有该架构，要加载新版本的代码就必须关闭再重新启动服务器。贯穿整个 JDK，很多地方都隐式地使用了 Java 对象序列化，因而它也成为 J2EE 的一块基石。当然，为了让容器能够一边走路一边嚼着口香糖（比喻词，即一心二用之意），J2EE 充分利用了 Java 线程模型，为客户请求提供并发执行的能力。

因此，理解 Java 平台的关键部分，对于任何想在 J2EE 平台上认真地做事情的 Java 开发者来说，都是绝对“必须的”。

第 67 项：主动释放资源

可以说，J2EE 应用就是访问 JVM 之外的资源，例如：关系型数据库（JDBC）、消息系统（JMS）、在互相分离的进程中运行的对象（RMI）、分布式事务协调器（JTA）、企业集成系统（JCA）等等。Java 虚拟机非常善于管理虚拟机内部的资源，特别是内存管理，但遗憾的是 JVM 的自动内存管理机制在管理 JVM 之外的资源时就不那么优秀了（比如数据库连接、结果集游标等）。因此，J2EE 程序员需要养成良好的习惯，要像对待资源分配一样显示地对待资源释放。简单的说就是，当我们不再使用资源对象的时候，就应该主动地关闭资源对象。

最初，这看起来可能并不是必需的，毕竟，Java 提供了终结器（finalizer）机制来在整个垃圾收集过程中清理对象。因此，假设所有这些资源对象（连接、结果集等）都具有终结器，能够关闭任何 JVM 外部的资源，那么我们就可以让垃圾收集来处理这一切了，对吗？

悲惨的是，这完全不对。就像第 74 项所解释的，终结器有一些很严重的问题。在这里，终结器完全是非确定性的，在对象已经被遗弃，并且已经可被收集之后的“某个时刻”，终结器可能会运行。 然而垃圾收集器并不知道这些资源对象所持有的资源远比对象本身所占的 N 个字节的内存要珍贵得多，因此，在收集器稍后有时间运行之前，是不会收集它们的。事实上，因为这些资源对象可以在数个分代垃圾收集（generational garbage collection）传递中存活下去（请参阅第 72 项），所以它们将迁移到分代式回收器中较老的一代中，这意味着直到这些对象完全年轻的一代被回收之前，它们都不会被考虑回收，因此也就不会有释放出来的资源被执行新的分配。在年轻一代能够被协调得很好的（请参阅第 68 项）、长期运行的系统中，这些对象在被终结之前的时间可能会是数分钟、甚至是数小时或数天。一个以每一秒一个的速率来分配 ResultSet，并且从来都不释放它们的系统，不管它底层的数据库服务器的硬件配置如何，都将会在另一边迅速地耗尽数据库资源。

这意味着除非作为程序员的你插手去帮助纠正这种状态，否则你将遭遇某种令人生厌的高度竞争和资源耗尽的场景。例如，在从 Statement 中获取 ResultSet 时，依据其事务隔离级别（请参阅第 35 项），数据库通常都要对表中 ResultSet 所表示的数据加锁，因为如果 ResultSet 是可以滚动的，那么直到 ResultSet 被正式关闭它也不知道哪些数据将会被访问和使用。（而一个只能向前转的“消防水龙带”式的 ResultSet 至少知道一旦数据被读取过，它就再也不能被访问了。这就是为什么许多 JDBC 性能指南都建议尽可能地使用它们的原因。）因此，对一张表打开过多的 ResultSet 对象会产生这样的情况：其它客户端要访问这张表的数据时，将被阻止，直至某些锁被释放为止。这种情况是一种竞争，它将损害可扩展性，而这正是我们应该避免的。

因此，你需要尽可能积极地释放任何由系统分配的资源对象，特别是 JDBC 的 Connection、Statement（带有一个值得注意的异常）以及 ResultSet 对象。在理想世界中，所有这些对象都由 WeakReference 实例持有，因此，一旦最后一个强引用被解除，对象自身就可以被清除了（请参阅第 74 项），但是这也不见得会立即发生，因此，你需要确保对于任何 JDBC 资源对象，一旦你完成了对其的使用，就应该在其上调用 close 方法。要注意的是，直到 JDBC 3.0 规范才明确提到，在一个 Connection 上调用 close 通常都会关闭由该 Connection 对象创建的所有 Statement 对象，而在一个 Statement 上调用 close 通常都会关闭由该 Statement 对象创建的所有 ResultSet 对象。

然而，在编写执行关闭的代码时要格外地仔细，你需要确保所有可能的代码路径都要覆盖到。例如，考虑下面的代码片断：

```
public String[] getFirstNames(String lastName)
{
  ArrayList results = new ArrayList();
  Connection con = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;

  try
  {
    con = getConnectionFromSomeplace();
    String prepSQL =
      "SELECT first_name FROM person " +
      "WHERE last_name = ?";
    stmt = con.prepareStatement(prepSQL);
      // See Item 49 for why we use PreparedStatement
    stmt.setString(1, lastName);
    rs = stmt.executeQuery();

    while (rs.next())
```

```
      results.add(rs.getString(1));
    }
    catch (SQLException sqlEx)
    {
      Logger l = getLoggerFromSomeplace();
      l.fatal("SQL statement failed: " + sqlEx);
      // By the way, don't forget to do something
      // more proactive to handle the error;
      // see Item 7
    }

    if (rs != null) rs.close();
    if (stmt != null) stmt.close();
    if (con != null) con.close();
      // Could also just call con.close if we know
      // that the JDBC driver does cascading closure
      // (which most do), but we'd have to know our
      // JDBC driver (see Item 49) to feel safe doing
      // that

    return (String[])results.toArray(new String[0]);
  }
```

只要我们处理完 ResultSet（这意味着我们也处理完了为其准备的 Statement 和 Connection），我们就在它们之上调用 close，并且就可以转移到我们要处理的任务列表中的下一项去了，对吗？

错了——这里有一个可怕的 bug，悄然等待着在以后猛咬我们一口。如果在方法的某处抛出了一个不是 SQLException，并且未经检查的异常，会发生什么呢？因为我们没有在 try/catch 语句块中捕获任何除 SQLException 之外的异常，所以我们将永远都不会执行在 Connection、Statement 或 ResultSet 上的 close 调用，而且，我们会高效地“泄露”资源，直至垃圾收集器抽身去终结它们。

因此，我们需要稍微改变一下上面的代码，利用 finally 语句块来执行关闭，而不是依靠命运的宠爱来确保那些 close 调用会被执行：

```
public String[] getFirstNames(String lastName)
{
  ArrayList results = new ArrayList();
  Connection con = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;

  try
  {
    con = getConnectionFromSomeplace();
    stmt = con.prepareStatement(
      "SELECT first_name FROM person " +
      "WHERE last_name = ?");
```

```
    // See Item 61 for why we use PreparedStatement
    // even though we'll lose the "preparation"
    // part of it when the Connection closes

  stmt.setString(1, lastName);
  rs = stmt.executeQuery();

  while (rs.next())
    results.add(rs.getString(1));

  return (String[])results.toArray(new String[0]);
}
catch (SQLException sqlEx)
{
  Logger l = getLoggerFromSomeplace();
    // See Item 12, as well as Item 7
  l.fatal("SQL statement failed: " + sqlEx);

  return new String[0];
    // See Effective Java [Bloch, Item 27]
}
```

```
  finally
  {
    if (rs != null) rs.close();
    if (stmt != null) stmt.close();
    if (con != null) con.close();
      // Could also just call con.close if we know
      // that the JDBC driver does cascading closure
      // (which most do), but we'd have to know our
      // JDBC driver (see Item 49) to feel safe doing
      // that
  }
}
```

现在，不管执行是怎样离开 try 语句块的——直接完成、一个返回语句或是一个在其内部抛出的异常——finally 语句块总是可以被调用，这样就可以确保 JDBC 资源对象总是可以被积极地释放。

乍一看，这么做有违你的直观感受。毕竟，建立连接并不是一项无开销的操作，因为数据库必须对传入的（用户名、密码）凭证进行验证，所以为什么要在我们建立它之后马上就又释放它呢？很明显，我们应该在某处缓存连接，要么是在 HttpSession 中，要么是在无状态会话 bean 中。

问题在于缓存连接以提高性能是以牺牲可扩展性为代价的。大多数时候，客户端程序在使用一个数据库连接或任何其它形式的外部资源时，都不会 100%地使用它的容量。请考虑典型的基于 Web 或基于 UI 的企业应用：我们呈现某些数据给用户，他们将花上漫长的时间（对 CPU 来说）用于考虑这些数据，以及可能会做出修改，然后提交这些修改，它们可以直接

也可以不直接写入数据库。在这段时间内，我们一直持有一个到数据库的连接，这个连接不能被其它任何客户端所使用。如果你考虑到了这一点，那么就会发现一个客户端可能利用它所获取的连接的时间大概也就是 5%。这又把我们带回到了一开始我们要转移到三层或 n 层系统时所面对的问题：共享资源。如果我们能够在 20 个客户端之间多路复用该连接，那么该连接的容量就被 100%地完全利用了，而此时只需要有一个物理连接——理论上是如此，然后，我们现在就可以用相同的硬件资源去支持 20 倍的并发客户端了。

但是，我们仍然有一个基础性的问题，就是连接管理——我们仍然要面对建立和关闭那些数据库连接的开销。在理想世界中，如果所有的客户端都是以某种方式通过相同的 JVM 来路由它们的数据库请求的，那么我们就可以池化连接，使它看起来就像一个中间件，建立了 20 个到数据库的连接，而实际上，这只是在同一个物理连接上多路复用的 20 个客户端。然后，假设每一个客户端都使用相同的凭证连接到数据库，获取连接的耗费将分摊到所有 20 个客户端上，这使得它成为了一种好的多方案。

即使是在池化连接不可能实现的时候，我们仍然倾向于主动地获取、使用和释放资源。即便这对性能是一个打击，但是当今编写的大多数企业系统对可扩展性都比对性能要感兴趣得多。乍一看，这像是一句难以置信的表述，但是请记住，在 1995 年之后开发的 IT 项目中，许多都被构建成通过互联网来服务于大众的，如果你的系统在经历 Slashdot effect 效应所产生的那种爆炸式负载之下崩溃，那将成为真正令人尴尬的公共关系方面的大笑话。大多数用户都不会注意到在线订书是否多花了几秒钟（时间通常可以通过减少你的网页上的琐碎的图形化元素来得到弥补——请参阅第 52 项），但是他们肯定会注意到服务在负荷之下何时崩溃了。

这种获取-使用-释放的方式称为*即时激活（just-in-time activation，JITA）*，顺便提一下，它或多或少地被 J2EE 规范中的多个部分用来直接建模，特别是在无状态会话 bean 中。记住，一个无状态会话 bean 不能在方法调用之间持有状态，这意味着 bean 需要的任何资源都必须在方法开始时获取，并在方法结束时释放。这是 JITA 策略的一项强制规定。将单个的用户交互会话（经常是 HTTP 请求）建模为单个会话 bean 的方法调用时，情况良好，但是当一个用户请求需要多个无状态会话 bean 调用时，它会引起“资源搅拌器（resource churn）”一般的效果，你将不断地“获取-使用-释放-获取-使用-释放-获取-使用-释放”。这就是为什么，

在谈到 EJB 的无状态会话 bean 时，许多作者都建议，你的无状态会话 bean 方法应该与你的系统用例（或多或少地）是一一对应的，因为在这种方式下，资源可以只被获取和释放一次。

当我们需要把从 ResultSet 中读取的数据保持比上面描述的场景更长的时间时，又会怎么样呢？例如，在一个搜索的结果页面中，我们想要在页面调用之间持有搜索结果——很明显，我们不想一次显示所有的搜索结果，但是，我们也不想对于结果页面中的每一个后续的“Next”请求都回到数据库中重复执行同一个查询。

为了不将数据存储在保持着连接的 ResultSet 中，我们将使用不保持连接的 RowSet，关闭 ResultSet，并利用 RowSet（它继承自 ResultSet，记住，因此他们的 API 是相同的）而不是最初的 ResultSet 来显示搜索页面的结果。遗憾的是，这么做意味着要在客户端进程中占据更多的内存，因为所有的结果集现在都被保存在 RowSet 实例所占据的内存中部了，而之前，这些数据是通过 ResultSet API 在要求时拉到客户端的。在这种情况下，你必须有意识地做出决定，支持 JVM 级别的可扩展性（此时内存成为了被争用的有限资源），或者数据库（事务锁）级别的可扩展性。虽然对大多数项目来说，为客户端增加更多的内存比改变数据库中的事务策略要容易得的多，但是你仍要做出自己的决定。

顺便提一下，请关注 JSR-114，这个有关新的 RowSet 实现的 JSR 将定义五种不同的 RowSet 对象，以满足各种 J2EE 应用中 RowSet 的标准化使用。（RowSet 接口现在已经成为了 J2EE 的一部分，但是它的任何实现都没有被标准化，Sun 发布过几个，其中 CachedRowSet 和 WebRowSet 是其中最流行的两个，但是它们仍旧被正式的 J2EE 规范排斥在外。）RowSet 不但提供了我们所喜欢的非连接的关系型数据存储，它还添加了若干新特性，例如优化的并发语义（请参阅 javax.sql.rowset.spi）、XML 输入/输出（请参阅 javax.sql.rowset.spi.XMLReader 和 javax.sql.rowset.spi.XMLWriter），以及在多个 RowSet 对象之间维护关系型关系的能力（请参阅 javax.sql.rowset.JoinRowSet）。RowSet 应该成为你的离线储存的首选——可以直接使用 JSR-114 已经提供的实现，或者根据你所需的附加语义实现自己的 RowSet。

无论你是将数据存储在 RowSet 中还是在由 Map 实例组成的 List 中，都应该尽可能快地拉出你需要的数据，主动地释放 JDBC 资源对象，从而缓解在 JDBC 管道上和数据库锁管理上的压力。请注意，尽管并不为大众所知，但是这条建议确实已经应用到了其它能够提供连接

数据输入的“外部”资源上，例如分布式事务和 Connector 资源对象（像 Record 对象一样）。

第 68 项：调整 JVM

随着 JDK 1.3 的发布，Sun 对 Java 虚拟机做出了一个重要的改进：特别是在他们发布 Hotspot 时，推出了一个优化的虚拟机，它承诺具有更好的垃圾收集、更好的对本地代码的 JIT 编译，以及大量的其它旨在使 Java 应用更快地运行的改进。事实上，有两个这样的虚拟机被创建了出来，一个专供基于客户端的应用使用，一个专供基于服务器的应用使用。客户端虚拟机通过向短期优化而不是长期优化进行倾斜，针对短期运行的应用（就像你常用的 Swing 应用）作了优化；而服务器虚拟机则针对相反方向的目标作了优化。

然而，令人疑惑的是，为什么大多数基于 Java 的 J2EE 容器的缺省安装（而不是本地代码的实现）并没有使用专门为长期运行的服务器操作而调整过的虚拟机？

要理解我的所指，我们就必须快速浏览一下 JVM 的调用代码。作为标准的 J2SDK 下载的一部分，如果你在安装过程中打开 Install Soureces 复选框，你就可以获取这些代码。当安装程序完成安装之后，一个大约 10MB 大小的名为 src.zip 的 zip 文件就显示在了 J2SDK 的根目录下。解开这个 zip 文件，除了别的目录之外，有一个“launcher”目录，它包含了 java.exe 启动器的源代码，我们特别感兴趣的核心是 java.c 源文件（另一个 C 源文件，java-md.c，它是机器相关的支持程序，依据你当前正在使用的 JDK，因平台不同而不同：Linux、Win32 或 Solaris）

我们对 Java 启动器所兴趣的方面在上述 C 文件中出现的相当早：

```
/*
 * Entry point.
 */
int
main(int argc, char ** argv)
{
  JavaVM *vm = 0;
  JNIEnv *env = 0;
  char *jarfile = 0;
```

```
char *classname = 0;
char *s = 0;
jclass mainClass;
jmethodID mainID;
jobjectArray mainArgs;
int ret;
InvocationFunctions ifn;
char *jvmtype = 0;
jlong start, end;
char jrepath[MAXPATHLEN], jvmpath[MAXPATHLEN];
char ** original_argv = argv;

/* ... Code elided for brevity ... */

/* Find out where the JRE is that we will be using. */
if (!GetJREPath(jrepath, sizeof(jrepath)))
{
  fprintf(stderr,
    "Error: could not find Java 2 Runtime Environment.\n");
  return 2;
}
```

```
  /* ... Code elided for brevity ... */


  /* Find the specified JVM type */
  if (ReadKnownVMs(jrepath) < 1)
  {
    fprintf(stderr,
      "Error: no known VMs. (check for corrupt jvm.cfg file)\n");
    exit(1);
  }
  jvmtype = CheckJvmType(&argc, &argv);


  jvmpath[0] = '\0';
  if (!GetJVMPath(jrepath, jvmtype, jvmpath, sizeof(jvmpath)))
  {
    fprintf(stderr,
      "Error: no '%s' JVM at '%s'.\n", jvmtype, jvmpath);
    return 4;
  }
/* If we got here, jvmpath has been correctly initialized. */


  /* ... Rest of main() elided for brevity ... */
}
```

对那些对 C 有点生疏的读者来说（在今天，谁不是如此呢？），main 本质上是要调用一个实用的方法，在这个文件内部也是如此，调用 ReadKnownVMs 来确定在 JDK 中有哪一个可用的虚拟机选项，并比较由 java.exe 命令行传入的参数，来发现用户想使用哪一个。一旦知道了存储在本地变量 jvmpath 中的虚拟机类型，启动器将使用标准的 JNI 调用 API 来创建 JVM，找到要加载的类，并调用它的 main 方法。

深入 ReadKnownVMs（这里没有展示）可以发现，JDK 已知虚拟机的列表仅仅是由一个文本文件控制，没什么复杂的技术——在这种特殊情况下，是一个名为 jvm.cfg 的文本文件，它存储在 Java 运行环境（Java Runtime Environment, JRE）的 lib 子目录下与 CPU 相关的目录中（例如，在 Win32 中该目录为 i386）。顺便提一下，这个子目录结构和文件名被硬编码在启动器代码中；如果不编写自己的启动器，你将无法对其进行修改。例如，在 JDK1.4.1 下，jvm.cfg 文件的内容如下：

```
#
# @(#)jvm.cfg  1.6 01/12/03
#
# Copyright 2002 Sun Microsystems, Inc. All rights reserved.
# SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
#
# List of JVMs that can be used as an option to java, javac, etc.
# Order is important – first in this list is the default JVM.
# NOTE that this both this file and its format are UNSUPPORTED and
# WILL GO AWAY in a future release.
#
# You may also select a JVM in an arbitrary location with the
# "-XXaltjvm=<jvm_dir>" option, but that too is unsupported
```

```
# and may not be available in a future release.
#
-client KNOWN
-server KNOWN
-hotspot ALIASED_TO -client
-classic WARN
-native ERROR
-green ERROR
```

如果文件中的注释没有泄露天机，你也要明白这里的关键是在列表中的第一个选项是缺省选项。这意味着除非你在 J2EE 服务器调用命令行中指明-server，否则你运行的将是客户端 Hotspot 虚拟机。（在 JVM 的早期发布版本中，Hotspot 是赋予被高度优化的 JVM 的产品名，它通常被推荐在客户端应用中使用；随着 JDK 1.3 的发布，Hotspot 有了两种可能的配置，服务器形式和经典形式，它们都被称为 Hotspot。）

如果你不十分确定到底哪一个 JVM 被调用了，你可以通过打开一个未公开的环境变量_JAVA_LAUNCHER_DEBUG，来查看 java.exe 启动器所使用的选项，这个变量将显示与 JVM 和创建它所使用的选项有关一个非常有趣的细节信息集合：

```
C:\Prg\Test>set _JAVA_LAUNCHER_DEBUG=1

C:\Prg\Test>java Hello
----_JAVA_LAUNCHER_DEBUG----
JRE path is C:\Prg\java\j2sdk1.4.1\jre
jvm.cfg[0] = ->-client<-
```

```
jvm.cfg[1] = ->-server<-

jvm.cfg[2] = ->-hotspot<-

jvm.cfg[3] = ->-classic<-

jvm.cfg[4] = ->-native<-

jvm.cfg[5] = ->-green<-

41768 micro seconds to parse jvm.cfg

JVM path is C:\Prg\java\j2sdk1.4.1\jre\bin\client\jvm.dll

9816 micro seconds to LoadJavaVM

JavaVM args:

   version 0x00010002, ignoreUnrecognized is JNI_FALSE,

                   nOptions is 2

   option[ 0] = '-Djava.class.path=.'

   option[ 1] = '-Dsun.java.command=Hello'

250409 micro seconds to InitializeJVM

Main-Class is 'Hello'

Apps' argc is 0

317641 micro seconds to load main class

----_JAVA_LAUNCHER_DEBUG----

Hello, world!


C:\Prg\Test>java -server Hello

----_JAVA_LAUNCHER_DEBUG----
```

```
JRE path is C:\Prg\java\j2sdk1.4.1\jre
jvm.cfg[0] = ->-client<-
jvm.cfg[1] = ->-server<-
jvm.cfg[2] = ->-hotspot<-
jvm.cfg[3] = ->-classic<-
jvm.cfg[4] = ->-native<-
jvm.cfg[5] = ->-green<-
31218 micro seconds to parse jvm.cfg
JVM path is C:\Prg\java\j2sdk1.4.1\jre\bin\server\jvm.dll
25056 micro seconds to LoadJavaVM
JavaVM args:
   version 0x00010002, ignoreUnrecognized is JNI_FALSE,
                   nOptions is 2
   option[ 0] = '-Djava.class.path=.'
   option[ 1] = '-Dsun.java.command=Hello'
288153 micro seconds to InitializeJVM
Main-Class is 'Hello'
Apps' argc is 0
398426 micro seconds to load main class
----_JAVA_LAUNCHER_DEBUG----
Hello, world!
```

```
C:\Prg\Test>
```

这里所揭示的内容都在“JVM path is…”这一行中。在用客户端虚拟机运行时，所用的 JVM 是被存储为 jre/bin/client/jvm.dll 的动态链接库，但是，在显式地用-server 标志来调用时，所用的 JVM 就变成了 jre/bin/server/jvm.dll。

可以固定使用这个 JVM。如果你能够获得用来调用 JVM 的命令行，那么就在命令行参数中添加-server 选项。如果这条命令行对你来说是屏蔽的，那么就请直接重新安排在 JDK 的 jvm.cfg 文件中的选项顺序，将-server 选项置于第一位。然而，请记住在 jvm.cfg 文件顶部的注释：这个文件格式在将来的某个时间会被修改，并且可能会以不同的机制被使用。（例如，JDK 1.3 机制没有使用这些关键词：KNOWN、ALIAS 等等。）将来的 J2SDK 发布也许会修改这种机制，所以请准备好在 J2SDK 1.5 发布时，要做一些新的学习与研究。

请注意，你还可以通过剔除文件中的无关选项（-classic、-native 和-green，以及顶部的大量注释），在启动时获得性能上的一次性的少许提高。这将把平均解析时间从 30,000 微秒减少到大约 9,000 微秒左右。这种减少看似不多，但是有时每一微秒都很有价值。

也可以通过使用-XXaltjvm 命令行参数来指定一个备用虚拟机的位置，但是除非备用 JVM 能够变得比编写本书时更为有用，否则这么做并不会有什么实际用处。这也是一种仅供 Sun 的 JVM 使用的特性（而且无证明文件），并且可能会从将来的发布版本中剔除，所以请慎用。

请注意，在这里 Hotspot JVM 并不是唯一的选择；特别是，BEA 获得了 JRockit，一个专为服务器操作而调整过的 JRE 的替代者。JRockit 本质上就是要替代 Sun 的 JRE，因此要想使用它，只需将你的 PATH 指到 JRockit 安装树中的 java 启动器，而不是 Sun 的 JRE 即可。

一旦正确的 JVM 得以执行，而且你也已经分析过你的代码，找出了所有明显的瓶颈和阻碍，在某种程度上，决定如何配置 JVM 自身，使其能够为你的系统更好地运行垃圾收集、线程策略等等，就变得相当有诱惑力。在某些情况下，这种知识会降低系统的可移植性，因为你的实现可能会依赖于底层的 JVM 的行为，但是这或许是，也或许不是一件坏事，完全取决于你对第 11 项的反应。

在系统启动时，JVM 花费了大量的时间在分配内存上，因为 JVM 缺省的启动内存（starting memory footprint）只是很小的 2MB。这对于像“Hello, world!”和“Goodbye, world!”这样的应用来说很合适，但是却不能满足企业应用运行一个 servlet 容器或 EJB 容器的需求。因此，应该基于实际的分析统计结果，给-Xms 传递一个合理的值：写一个简单的 servlet，返回从 Runtime.totalMemory 得到的值，把它部署到你的 servlet 容器中，先关闭容器，再重新启动，然后直接访问该 servlet。这样做可以获得 servlet 容器开销需求的一个粗略的估计值。

相似地，JVM 设置了它能够使用的堆的最大值。许多 Java 资料都描述到：这是 JVM 将要开始进行垃圾收集时，所达到的值，但是真实情况比这要稍微复杂一些。而真实情况就是 JVM 永远不会增长得超过这个值。缺省地，JVM 将建立 64MB 的上限。在对你的系统进行最大负荷测试做出合理近似的过程中，基于分析统计数据（通过使用商业的分析软件或上面提到的 servlet 而获取的），使用-Xms 选项来调节这个值，使它更合理。

在传递给-Xms 比较大的值时，请格外小心，因为随着堆的不断增长，扫描对象的困难度将呈几何级数增长。现实一些，如果你正在考虑大于 1GB 的 JVM 的最大堆尺寸参数，那么你应该尝试运行多个小尺寸的而不是一个大的 JVM 进程。如果你正在担心 JVM 的多次开销（一个进程一次），那么你应该庆幸这样的事实：（a）大多数操作系统都会默默地将多个进程中的代码片断映射到相同的物理内存上，这样就回避了在物理内存中对相同的代码加载两次的问题；（b）J2SE 1.5 发布版本正在把“代码共享”引入到 JVM 自身中来，以试图避免在其它事物中已经被加载过并被定义过的类的开销。再次提醒你，在将该方式应用到产品中之前，你需要仔细地进行分析（请参阅第 10 项）。

另一个选择是将启动堆尺寸设置为与最大堆尺寸具有相同的值，以避免在堆增长得超过启动尺寸时，要进行堆尺寸的调整。但是这种做法假设 JVM 在需求缩减时，也永远不会实际释放被分配的堆内存。尽管这种情况是 Sun 的 JVM 1.4 发布版本才有的，但是据推测，它迟早要被修正，而且，因为像 JVM 是唯一在机器上运行的应用的这种情况很少见，所以当你不再使用某些东西（像内存）的时候，将它们返回给其所有者（操作系统）会显得更优雅一些。

尽管我们在谈论释放某些事物的话题，但是开发者们还时不时地提出调用 System.gc 以强制终结那些有外部资源需要释放的对象。除了不可思议的浪费之外（这将触发一次完全的垃圾收集清扫），垃圾收集操作永远都不能保证可以找到已不被使用的对象并收集它们——就像第 72 项所解释的那样，垃圾收集算法事实上可能不会马上释放对象，不管该对象是否处于不可再取用的状态。相反，一种好得多的方式是：如果对象具有 close 或 dispose 方法，那么就通过这些方法显式地释放对象的资源。（如果该对象没有这样的方法，而且它也没有使用 Reference 对象或关闭挂钩来确保释放，那么此时你就像拿到了一个烫手山芋，你赶快给向你提供这个对象的供应商的支持部门发一封怒气冲冲的 e-mail，以泄你心中的怨气，然后去找其它的什么东西来替代它。）

另一个你可以使用的调整参数来自RMI管道。JRMP之上的RMI，由于它具有分布式垃圾收集行为，因此它强制进行周期性显式的垃圾收集，它受到分别针对客户端和服务器的JVM系统属性sun.rmi.dgc.client.gcInterval 和sun.rmi.dgc.server.gcInterval的控制，它们的缺省值都被设置为 1 分钟（60,000 微秒）。为了避免这种频繁的垃圾收集操作，可以将这些值设得高一些（要认识到这同样也使得不再被使用的RMI对象在被回收之前的存活期变长了）。是不是在想你没有用到RMI？再猜猜——记住，RMI是EJB在其上进行操作的首选RPC协议。（缺省情况，会使用RMI-IIOP，但是大多数服务器好像更喜欢使用RMI-JRMP，因为它对Java更加友好一些。）

另外，就像在第 72 项中更详细地描述到的一样，有大量的垃圾收集参数可以用来使垃圾收集器在你的代码上以更高效的方式运行；然而，使用这些参数只是一种辅助手段——只有在你分析过代码（或者看过verbose:gc的输出，那好像可以指出问题）并且你没打算第二天交好运时，才应该考虑使用它们。

总的来说，调整 JVM 是一种粗粒度的改善：你是在声明，你的代码在大模样上应该如何运行，并且你应该可以得到相近的结果。例如，选择客户端还是服务器 JVM，将有很大的不同，这可能会（够讽刺的是，依据你的特定应用也可能不会）使你的企业应用具有更好的性能。不要期望性能会有数量级的改善，但是当这种情况偶尔发生了一两次时，也不要感到惊讶。

第 69 项：为版本并存使用独立的 JRE

微软的.NET 平台大肆鼓吹的一个特性是它已经能够支持版本并存（side-by-side versioning）了——换句话说，后继版本的.NET 平台与以前版本可以安装在同一台机器上，而不会产生任何反作用。对一个应用发布的第一个版本来说，这看起来像是一个超值的选择。然而，一旦应用出现了第二个版本，或者另一个应用被部署到了应用服务器中，又或者是两个应用服务器需要并存，那么确保每样事物都正确地运行就变得绝对关键了。

例如，考虑一下，有一家公司，它在多年前编写了其第一个 EJB 应用，它被部署在与 JDK1.3 相绑定的应用服务器中。这个应用现在仍然在使用，而且当该公司被其它公司收购，或者合并或收购其它公司时，它在用户社区中都十分受欢迎。很自然地，该公司的首要愿望就是用第一个企业应用去巩固第二个公司的企业应用——如果第二个公司正在使用的是需要有 JDK1.4 支持的完全不同的应用服务器，那么这可就不是一项容易完成的任务了。理想情况是，我们可以从第一个应用服务器中获取.ear 文件，然后在第二个应用服务器中原封不动地运行它们。遗憾的是，事情并非如此简单。

对初学者而言，新组成的公司实体已经决定继续将第一个应用服务器提供商作为它的企业应用服务器伙伴——这意味着第二个应用服务器，既要依赖 JDK 1.4 的那一个，将不能继续使用了。遗憾的是，这将导致另一个问题。第二个企业应用使用了数个 JDK1.4 相关的特性，例如 Preference 和日志记录 API。

如果到目前为止，这个故事看起来还显得有一点难以置信，那么请考虑另一种变体：当前，你的公司在同一台机器上使用了一个单独的 servlet 容器来处理所有的 servlet/JSP 操作，以及一个单独的 EJB 容器来处理它的 EJB 操作（为了避免网络上的往返调用——请参阅第 17 项）。当这些服务器第一次被部署到那台机器上时，它们都要求有 JDK1.3 的支持；然而，在 servlet 容器中发现了一个安全漏洞，它需要被升级为需要有 JDK1.4 支持的最新版本。

无论你认为哪个故事更可信或是更可能，这里最终的结果都是那台单机现在要面临必须要同时支持 JDK1.3 和 JDK1.4 安装的问题。将来更有可能是 JDK1.4 和 JDK1.5，或者 JDK1.4 和 JDK2.0 等等。

正常情况下，当Java被安装到了一台机器上时，JDK就被部署成了一个共享目录，在基于UNIX的机器上通常是/usr/local/java。在Win32 的运行环境中，可以放置两个JRE安装：第一个位于用户在安装过程中指定的位置，第二个位于另外两个位置。JRE的大部分在C:\Program Files\Java\(JRE version)目录下，但是一些可执行文件还被拷贝到了C:\WINNT\SYSTEM32 目录下——这正是为什么“java”总是能够在任何命令提示符下被访问到的原因，因为C:\WINNT\SYSTEM32（或者在Win9X、XP或以后的安装上的C:\WINDOWS\SYSTEM32）目录缺省地是PATH固有的一部分。

然而，仅仅只是因为安装程序将Java置于了一个共享位置，并不能说明所有的应用都要求Java从该共享位置运行；实际上，应用真正需要的只是在运行时刻所需的文件集合，即JRE。正如该应用所表明的，在启动时刻所需的只是要知道JRE安装的根目录在哪里，从而使它能够找到启动JVM所必需的配置文件。

返回到我们在第 68 项中所研究过的启动器代码，再次审视一下main函数靠前的部分。在那里，启动器发现了执行JRE的路径。请记住，这是要调用java.exe的代码，所以从技术上讲，启动器是在试图发现到底应该从哪里启动它。该发现是从GetJREPath函数而得到的，这个函数没有在java.c源文件内部的任何地方被定义；反之，它是一个平台依赖的函数，它的实现在Solaris、Linux和Win32 结构的JDK之间会有所变化。因此，它是在java_md.c源文件中定义的，而该文件与java.c在相同的目录下；相关的代码片断（这里是摘自Win32 版本的片断）概括如下：

```
/*
 * Find path to JRE based on .exe's location or registry
 * settings.
 */
jboolean
GetJREPath(char *path, jint pathsize)
```

```
{
  char javadll[MAXPATHLEN];
  struct stat s;
  if (GetApplicationHome(path, pathsize))
  {
    /* Is JRE co-located with the application? */
    sprintf(javadll, "%s\\bin\\" JAVA_DLL, path);
    if (stat(javadll, &s) == 0)
    {
      goto found;
    }

    /* Does this app ship a private JRE in
       <apphome>\jre directory? */
    sprintf(javadll, "%s\\jre\\bin\\" JAVA_DLL, path);
    if (stat(javadll, &s) == 0)
    {
      strcat(path, "\\jre");
      goto found;
    }
  }
```

```
  /* Look for a public JRE on this machine. */
  if (GetPublicJREHome(path, pathsize))
  {
    goto found;
  }

  fprintf(stderr, "Error: could not find " JAVA_DLL "\n");
  return JNI_FALSE;

found:
  if (debug) printf("JRE path is %s\n", path);
  return JNI_TRUE;
}

/* ... Some code elided for brevity ... */

/*
 * If app is "c:\foo\bin\javac", then put "c:\foo" into buf.
 */
jboolean
GetApplicationHome(char *buf, jint bufsize)
{
```

```
    char *cp;
    GetModuleFileName(0, buf, bufsize);
    *strrchr(buf, '\\') = '\0'; /* Remove .exe file name */
    if ((cp = strrchr(buf, '\\')) == 0)
    {
      /* This happens if the application is in a drive root,
       * and there is no bin directory. */
      buf[0] = '\0';
      return JNI_FALSE;
    }
    *cp = '\0';  /* Remove the bin\ part */
    return JNI_TRUE;
  }
```

尽管我们可以深入GetApplicationHome和GetPublicJREHome的定义，但是其实并没有必要知道它们的底层细节。当启动器在探索以期发现到达JRE的路径时，它首先检查JRE是否与应用被定位在一起；也就是说，JRE的根目录是否与应用自身运行于其中的目录相同。其次，假设在那里并没有找到与JRE相似的结构，于是，它转而查看应用是否在运行一个“私有的”JRE——该JRE被安装在应用的主目录下的一个子目录中。最后，假设上面两项检查的结果都不为真，启动器执行一个Win32 注册表查找。它查询在HKEY_LOCAL_MACHINE下的Software/JavaSoft/Java Runtime Environment键。被命名为CurrentVersion的值持有最近安装的JVM的版本信息，它是该键的最后一部分；例如，假设CurrentVersion的值是 1.4.1，启动器查找Software/JavaSoft/Java Runtime Environment/1.4.1，并找到了三个值：JavaHome，JRE位于该目录；MicroVersion，它指示了该发布的小版本号（因为是JDK1.4.1，因此此时是 1）；以及RuntimeLib，即包含了JVM自身的本地部分的jvm.dll文件所在的位置。一旦所有这些都

被发现了，启动器就会动态地加载jvm.dll文件，通过JNI调用API来使用该文件创建一个JVM实例，然后通过调用由命令行参数所指定的类的main方法，将控制传递到JVM中。

关键的事情是要了解所有这些都是在说明查询JRE的顺序——特别是要注意到这个事实，即启动器首先在执行所发生的目录查找JRE，这意味着，每一个Java应用都可以拥有它们自己的JRE而不会产生任何冲突。事实上，这很简单，只需要从JDK中把整个JRE子目录（jre子目录在主JDK安装的根目录下）拷贝到容器安装目录之下的任何位置，并使用该java启动器来调用容器。例如，我总是在我的笔记本电脑上运行多个JDK和Tomcat servlet容器，因此我将一个JRE副本拷贝到Tomcat安装的根目录下，然后启动指向它的批处理脚本，而不是依赖PATH来识别使用哪一个java.exe启动器。

这提供了一种杜绝系统上存在意外的或未知的版本不兼容问题的手段。每一个 JRE 都完全独立于系统上的其它 JRE，这意味着无论机器上安装或卸载了什么样的 JDK，服务器或容器都可以继续运行。在某些情况下，这还可以简化部署，因为它不需要系统管理员必须了解要部署到哪一个正确版本的 JRE。只需为应用提供一个 tar 或 zip 文件形式的 JRE，并把它们解压缩到恰当的位置，而不需要做任何修改。（JNLP 为客户端应用透明地处理这些工作。）

顺便提一下，在本项中没有任何事物是与Win32 相关的；我只是列举了Win32 版本的代码，因为它更复杂（这要归因于它使用注册表来查找“公共的”JRE的位置）。UNIX版本的GetApplicationHome实际上要简单得多，并且为启动器引导JVM运行提供了所有必须的信息。在此处“必须要有一个 GetJREPath 函数”这句话就显得并不那么可信了。启动器使用GetApplicationHome函数在内存中的地址来查找与该函数所属的可执行文件（要么是一个可执行的应用程序，要么是一个共享的类库）相关的动态加载的信息，得到该可执行文件所属目录的信息，并用已知的JRE目录结构反向推导出JRE的根目录。（请参阅你的系统的程序员API手册，以了解dladdr系统调用的细节。）再次强调，关键是该查找完全是相对于启动器可执行文件的位置。

现在，让我们深入到产品中，应用服务器、servlet容器、支持Java的数据库，以及其它Java服务，它们每一个都可以运行在一个为其精确定义的JVM环境中。请注意，这种在多个容器的JVM之间的隔离性还提供了一个机会，可以更好地在容器之间隔离Java扩展（被置于

jre/lib/ext 扩展目录中的Java .jar 文件），并有助于消除某些关于JDK 1.4“认可标准目录（endorsed standards directory）”以及有关XML解析器和CORBA ORB的混淆。顺便说一下，本项并非仅限于企业服务器容器。我通常会为我的笔记本电脑上最新安装的Ant配置它自己的JRE，并且，为了简单起见，我安装了所有Ant所依赖的.jar 文件，例如JUnit，把它们都安装到了JRE的扩展目录下。每一个JVM都用它们自己的设置、自己的扩展、更重要的是都有自己的Hotspot配置文件（请参阅第 68 项）。改变其中一个并不会影响到其它的JVM，这正是我们在任何种类的产品环境中都想要的特性。

第 70 项：识别类加载器的边界

类加载器架构的职责是 JVM 最基础性的任务，那就是把代码加载到 JVM 中，使其能够用于执行。沿着这条路，它还定义了隔离边界，以使得包/类的命名相似的类不会彼此之间产生不协调。通过这样做，类加载器把显著的灵活性引入到了 Java 基于服务器的环境中，进而也就引入了显著的复杂性。处理类加载器之间的关系让开发者勃然大怒——就在我们以为已经解决了它的所有问题时，某些其它的东西又悄悄混了进来，把每件事情又再次置于悬而未决的境地。尽管我们希望集体采取鸵鸟政策去回避现实，并假装如果我们的愿望足够强烈的话，类加载器的问题就会自动消失，但是残酷的现实仍然存在，企业 Java 的开发者们必须理解他们所选择的服务器产品的类加载器之间的关系。

考虑一个最普通的场景，经典的 MVC/Model 2 架构：一个 servlet Web 应用在用户会话空间中存储 Bean（不是 EJB 类型的 Bean），作为该 Web 应用的一部分，以备后面处理过程中使用。最初，这些 Bean 被打包在一个单独的.jar 文件中，并被部署在服务器产品自身的 CLASSPATH 中——这看起来是一种较为简单的方式。遗憾的是，这样做意味着 Bean 偶尔会与在不同的 Web 应用中具有相同名称的其它 Bean（我们可以数出多少个 LoginBean 类？）发生冲突，因此这些 Bean 不得不被移入到 Web 应用的 WEB-INF/lib 目录中，即它们早先被认为应该处于的位置。

然而，现在无论何时，如果 Web 应用中的某个 servlet 被修改过了，并且容器施展了它那自动重新加载的魔法，那么奇怪的 ClassCastException 错误就悄然而至了。比方说你下面这样

的代码：

```
LoginBean lb = (LoginBean)session.getAttribute("loginBean");
if (lb == null)
{
  . . .
}
```

Servlet 容器不住地抱怨：从 session.getAttribute 返回的对象不是一个 LoginBean。同时，你在大把大把地揪你的头发，因为你知道事实上它就是一个 LoginBean；就在前一个页面，你把它放在了那里。更糟的是，当你通过在返回的对象上调用 getClass.getName 来验证它是一个 LoginBean 时，它揭示的结果是该对象确实是 LoginBean。更妙的是，如果你重新启动服务器，这个问题就完全消失了，直到你下一次修改 servlet。你静静地沉思是否该隐退了。

这里的问题是一个有关类加载器的问题，代码并没有问题。

特别的是，类加载器在 Java 环境中不仅作为一种加载机制使用，而且用来在不同的代码之间建立隔离边界——举例来说，这意味着我的 Web 应用不应该以任何方式与你的 Web 应用相冲突，尽管事实上我们在相同的 servlet 容器中运行。我可以使用与你的 Web 应用完全一样的名字来命名我的 servlet 和 Bean，而这两个应用应该可以秋毫无犯地在一起运行。

想理解类加载器为什么要提供这种隔离行为，我们就必须明确某些有关类加载器的基本规则：

1. *类加载器组成层次结构*。当一个类加载器被创建时，它总是缺省地指向某个“父”加载器。缺省地，JVM 开始于三个类加载器：一个是以本地代码编写的、用来加载运行时类库（rt.jar）的引导加载器；一个是指向扩展目录（通常是在你的 JRE 目录中的 jre/lib/ext）的、被称为扩展加载器的 URLClassLoader；另一个是指向由 java.class.path 系统属性所指示的元素的 URLClassLoader，该属性是通过 CLASSPATH 环境变量设置的。像 EJB

或 servlet 容器之类的容器通过将它们自己的类加载器添加到这个类加载器之树中来扩大它，通常它们被置于树的底部或叶子节点。在子类加载器加载代码之前，该层次结构将加载代码的任务委托给父类加载器，这样就在加载代码时，给了引导加载器首先加载的机会。如果父加载器已经加载了一个类，那么就不会再做任何加载该类的尝试了。

2. *类是被惰性加载的*。Java 虚拟机，像大多数可管理的环境一样，总是想最小化它在启动时所必需的工作量，并且若非必要不会加载一个类。这意味着在某个时刻，一个类可能突然调用了一个之前从未调用过的方法，而该方法引用了某个尚未加载的类。这将触发 JVM 去加载这个类，这也引出了类加载器的下一条规则。顺便说一句，这正是为什么旧式的非 JNDI 的 JDBC 代码需要“引导”驱动程序到 JVM 中的原因。如果不这样做，实际的驱动程序将永远不会被加载，因为你的 JDBC 代码按惯例是不会直接引用驱动程序相关的类的，而 JDBC 自身也不会这么做。
3. *类是由加载了请求类的类加载器加载的*。换句话说，如果一个 servlet 使用了 PersonBean 类，然后当 PersonBean 需要被加载时，JVM 将回到加载了该 servlet 的类加载器。当然，如果你有一个到某个类加载器的引用，那么你也可以显式地使用该类加载器实例来加载一个类，但是这只是一种例外，并不是一条规则。
4. *类在 JVM 中是由类名、包名和加载该类的类加载器结合在一起而唯一地标识的*。这条规则意味着一个指定的类可以在 VM 中被加载两次，只要该类是通过两个不同的类加载器加载的即可。这还意味着，当 JVM 检查一个 castclass 操作时（诸如前面提到的 LoginBean 的转型），它会检查两个对象，从类加载器的角度去查看它们是否共享了任何共同的祖先。如果没有，就会抛出一个 ClassCastException 异常。这同样还意味着，既然两个类被认为是唯一的，那么每一个都拥有其自己的静态数据副本。

在确定了这些规则之后，让我们来审视一下这些规则对企业 Java 开发者来说，实际意味着什么。

隔离

为了支持在 Web 应用之间进行隔离的概念，servlet 容器为每一个 Web 应用都创建了一个类加载器实例，从而有效地防止了从一个 Web 应用中把类“泄露”给了另一个应用。许多 servlet

容器还提供了一个“公共”目录，在其中放置了所有 Web 应用都可以看到的.jar 文件。因此，大多数 servlet 容器都有一个类加载器的层次结构，在最小情况下，它看起来就像图 8.1 所示的那个一样。

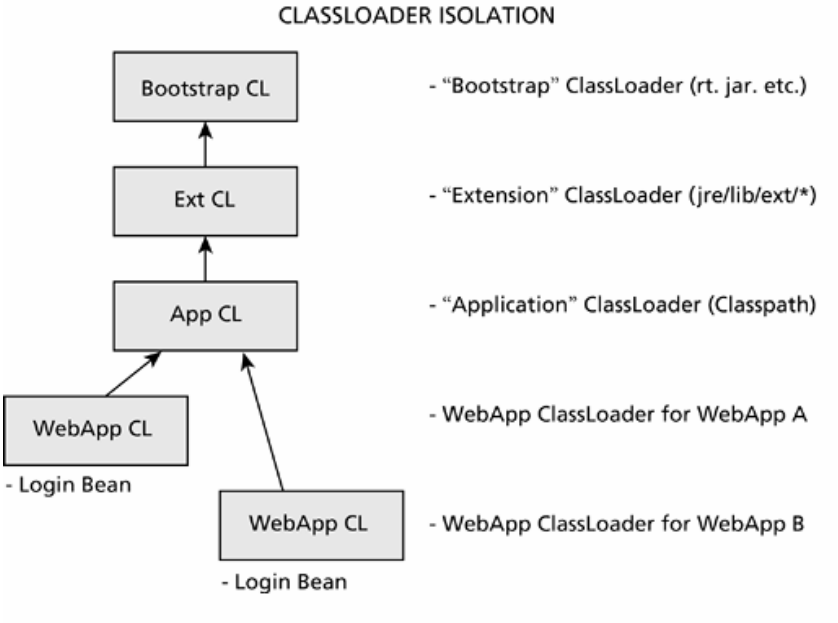

图 8.1 类加载器的隔离

请注意，这表明既被部署为 WebAppA.war 的一部分，又被部署为 WebAppB.war 的一部分的 LoginBean 类被两次加载到了 servlet 容器中：一次是通过 Web 应用 A，一次是通过 Web 应用 B。如果 LoginBean 有静态数据成员，那么会发生什么呢？

答案很简单，来源于前面提到的规则 4：每一个 LoginBean 都是由类名、包名和加载该类的类加载器结合在一起而唯一地标识的。每一个 Web 应用被单独的类加载器实例所加载，因此，这是两个完全不相关的类，它们维护着各自独立的静态数据。

这对 serlevt 开发者来说具有深刻的含义——例如，考虑一下无处不在的 ConnectionPool 类。典型地，这个类被编写用来维护一个静态数据成员，它持有连接池想要分发出去的 Connection 实例。如果我们修订图 8.1，将 ConnectionPool 置于 LoginBean 的位置，信任这种方法的开发者其实有了三个 ConnectionPool 实例在运行，而不是一个，尽管事实上连接池自身是被作为静态数据维护的。为了修正这一点，可以将 ConnectionPool 类置于一个 jar 文件中，或者是在类加载层次结构中层次较高的类加载器中。或者，最好是依赖于遵循 JDBC3.0 的驱动程序来完全处理 Connection 缓冲池机制（请参阅第 73 项）。

教训：单件并不能解决上述问题，除非你知道你在类加载器层次结构中所处的位置。

版本控制

为了支持 servlet 的热重载，典型的 servlet 容器将在 Web 应用每一次被修改时，创建一个新的类加载器——例如，当一个开发者重新编译一个 servlet，并将其置于 Web 应用的 WEB-INF/classes 目录下时，servlet 容器会注意到发生的修改，并将创建一个全新的类加载器实例。容器通过该类加载器重载 Web 应用的所有代码，并使用这些类来应答任何新到来的请求。因此，现在类加载器的格局就像是图 8.2 所示一样。

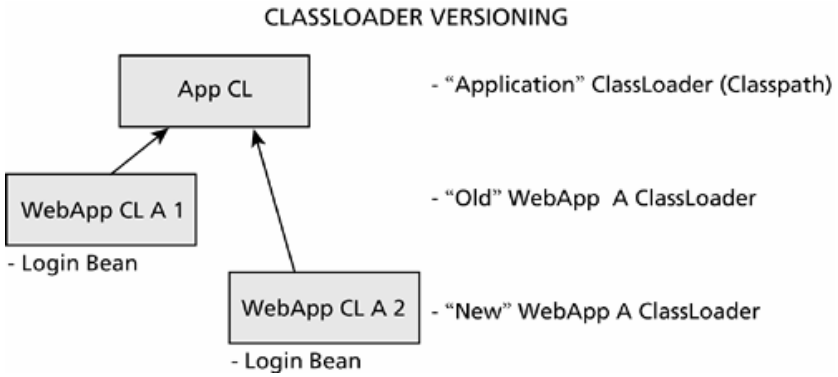

图 8.2 类加载器并行地运行以提供版本控制

让我们稍微使上面的格局复杂一些：假设 SampleWebApp 的版本 1 创建了一个 LoginBean，并将其存储到了会话空间。该 LoginBean 是被作为 SampleWebApp-v1 的一部分而创建的，因此与这个对象相关联的类的类型（由包名、类名和类加载器实例构成的唯一的元组）是*（未命名的包）/LoginBean/SampleWebApp-v1*。但目前为止，一切顺利。

现在，开发者接触到了一个 servlet（或者是热部署了 Web 应用的一个新版本），它强制 servlet 容器重载该 servlet，这将需要一个新的类加载器实例。你可以看到接下来会发生什么。当该 servlet 试图从会话空间取出 LoginBean 对象时，类的类型就不匹配了：该 LoginBean 实例是一种由类加载器 1 加载的类型，但是它却被要求转型为一种由类加载器 2 加载的类。即使这两个类是同一个（在两种情况下都是 LoginBean），但是它们是由两个不同的类加载器实例加载的这一事实意味着它们是完全不同的类，因此会抛出一个 ClassCastException 异常。

相同的类若由不同的类加载器加载，将被当作完全不同的两个类对待，这种做法初看有些专横。然而，对于 Java 作的大多数事情来说，都有一个相当好的理由。让我们考虑一下，如果事实上这些类确实是不同的，那会如何。假设 VM 允许上面的转型发生，但是事实上，新版本的 LoginBean 没有旧版本中的某个方法，或者没有实现旧版本实现的某个接口。既然我们允许这种转型，那么 VM 应该在代码调用那个旧方法时做些什么呢？

某些人曾经建议，VM 应该比较两个类的二进制结构，然后基于这两个类是否在事实上确实相同来决定是否允许转型。这意味着在系统中的每一个转型，更不用说每一个引用委派，将必须支持这种二进制比较，这将严重地损害性能。另外，还需要去开发规则，以确定何时两个类是“相同的”——如果我们添加了一个方法，这是否是一个可接受的修改呢？如果我们添加了一个域，情况又怎样呢？

不幸的现实是，类名与类加载器实例配对是最简单的确定类的唯一性的方式。然后，我们的目标就是怎样使用它，而使那些令人恼火的 ClassCastException 错误不会再蹦出来。

一种方式是完全忽略：经常地，当我们面对这个问题时，总是尝试着通过完全绕开它而解决问题。为了让代码可以工作，我们将 LoginBean 类置于类加载器层次结构中层次足够高的某处，以使其不会受重新加载的影响，通常要么将其置于容器的 CLASSPATH 中，要么将其置于容器的 JVM 的扩展目录中（请参阅第 69 项）。遗憾的是，这意味着如果 LoginBean 发生了变化，服务器不得不重启以重新加载它。这可能会产生某些严重的演化问题：Web 应用 A 和 B 依赖版本 1 的 LoginBean，但是 Web 应用 C 却需要并不能向后兼容的版本 2 的 LoginBean。如果 LoginBean 被部署在类加载器层次结构的较高层次上，那么 Web 应用 A 和 B 在 Web 应用 C 被部署后就会被突然地“中断”。这是一种检查你能够连续挺住多少个 24 小时不间断调试的好方法。

然而更糟的是，把 LoginBean 部署在类加载器层次结构中如此高的层次，意味着其它 Web 应用可能也能够看到 LoginBean，即使是它们并不应该看到。因此，对于根本就不使用 LoginBean 的 Web 应用 D，它仍然能够看到这个类，并且潜在地有可能作为一种攻击 Web 应用 A、B 或 C 的手段而使用它。如果你的代码宿主在你与其它人共享的服务器上（就像

ISP 的情形），那么这就相当危险。其它应用可以在你的 LoginBean 类上使用反射机制，并可能会发现你想要保密的许多东西。

不要失望——我们并没有丧失一切。有几个技巧仍然可用。

技巧一是定义一个接口，叫做 LoginBean，然后置于类加载器层次结构中的较高层次，使得它不会被加载 Web 应用的类加载器所加载。该接口的一个实现，LoginBeanImpl，驻留在 Web 应用中，并且任何想要使用 LoginBeanImpl 的代码，都是将它作为 LoginBean（接口）进行引用的。当 Web 应用重启动时，“旧的”LoginBeanImpl 转型为接口 LoginBean，而它不会被重新加载，因此这种转型会成功执行而不会抛出任何异常。这里的缺点也显而易见：每一个可会话对象都需要被分离成接口和实现。（这正是为什么 EJB 为每个 bean 强制这种特殊形态的部分原因：通过这种方式，EJB 可以移动类加载器实例的位置而不用担心 ClassCastException 错误。并非巧合的是，这也是为什么 EJB 实例被禁止含有静态数据成员，因为静态数据成员不能与接口很好地一起运作。）

技巧二是在会话空间中只存储来自 Java 基本运行时类库中的对象（例如，string 和 Date 对象），而不存储自定义的对象。Java 的 Collections 类，特别是 Map，在此处作为持有数据的“伪类”将非常有用。因为引导类加载器加载的是运行时类库，所以这些对象永远也不会出现热版本，因此也就不会遭遇相同的问题。然而，这里的危险在于，Java 的 Collections 类可以持有任何类型事物的实例，这意味着将所有的东西都附着到会话中的诱惑将很难抵挡（请参阅第 39 项）。

技巧三假设你希望或者必须拥有自己定制的对象，但是不能花时间去将其分解为接口和实现两部分。在这种情况下，将这个类标记为可序列化的，然后使用 Java 对象序列化来把对象的一个序列化副本作为一个字节数组存储在会话中。因为 Java 对象序列化或多或少地忽略了这个问题，并且因为字节数组其自身隐含的是可序列化的（因此满足 Servlet2.2 规范有关只有可序列化对象才能被存储进会话的要求/建议），你可以存储对象被序列化后的版本，而不是标准的对象类型，从而取代将会话对象转型回到 LoginBean 引用，而是反序列化它：

```
// Store in session
```

```
LoginBean lb = new LoginBean(...);
try
{
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  ObjectOutputStream oos = new ObjectOutputStream(baos);
  Oos.writeObject(lb);
  byte[] bytes = baos.toByteArray();
  session.setAttribute("loginBean", bytes);
}
catch (Exception ex)
{
  // Handle exception
}

// Somewhere else, retrieve LoginBean from session
LoginBean lb = null;
try
{
  byte[] bytes = (byte[])session.getAttribute("loginBean");
  ByteArrayInputStream bais = new
    ByteArrayInputStream(bytes);
  ObjectInputStream ois = new ObjectInputStream(bais);
```

```
    lb = (LoginBean)ois.readObject();
  }
  catch (Exception ex)
  {
    // Handle exception
  }
```

然而，这第三种方式要付出显著的代价：序列化和反序列化对象代价有些昂贵，即使是你遵循第 71 项时也是如此。幸运的是，你不会被迫去经常使用它。另一个问题与 servlet 容器有关，字节数组并不符合 JavaBean 规范，因此不能被当作标准的 JSP bean 标签（useBean、getProperty 和 setProperty）的目标而使用。

最终，这里的关键是要精确地知道你的Java环境是如何建立类加载器之间的关系的，然后利用它们而不是抵制它们。如果你的环境不能直接告诉你这些信息，那么通过调用getClass().getClassLoader()并穿越其层次结构，来进行一些明智的探查就相当有必要了。不这么做就意味着神秘的ClassCastException错误将会出现在你最不想看到它们的地方——产品中。

第 71 项：理解 Java 的对象序列化

Java的对象序列化是一项神奇的功能。它使得Java程序员可以通过让对象实现java.io.Serializable接口，并将其传递给ObjectOutputStream的writeObject方法，来获取该对象，并将其还原为一个字节流。从该字节流中重构该对象也是相类似地简单：只需在包装字节流的ObjectInputStream上调用readObject。作为一个展示序列化真正是多么神奇的例子，请看下面的代码：

```
import java.io.*;
```

```
import java.util.*;


class Person implements Serializable
{
  public String name;
  public Person spouse;
  public ArrayList children = new ArrayList();
}
public class Serial
{
  public Serial()
  { }


  public static void main(String[] args)
    throws Exception
  {
    if (args[0].equals("write"))
    {
      Person youssef = new Person();
      youssef.name= "Youssef";


      Person sheryl = new Person();
```

```
  sheryl.name = "Sheryl";


  youssef.spouse= sheryl;
  sheryl.spouse= youssef;


  Person child1 = new Person();
  child1.name = "Johnny";


  Person child2 = new Person();
  child2.name = "Mike";


  youssef.children.add(child1);
  youssef.children.add(child2);
  sheryl.children = youssef.children;




  FileOutputStream fos = new FileOutputStream("people.ser");
  ObjectOutputStream oos = new ObjectOutputStream(fos);
  oos.writeObject(youssef);
  fos.close();
}
else
```

```
    {
      FileInputStream fis = new FileInputStream("people.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      Person youssef = (Person)ois.readObject();

      System.out.println(youssef.name);
      System.out.println(youssef.spouse.name);
      System.out.println(((Person)youssef.children.get(0))
        name);
      System.out.println(((Person)youssef.children.get(1))
        name);
      System.out.println(((Person)youssef.spouse.children
        get(0)).name);
      System.out.println(((Person)youssef.spouse.children
        get(1)).name);
    }
  }
}
```

这里需要仔细注意的是writeObject上的一次调用满足下面完整的对象图：当youssef被序列化时，sheryl、johnny、和mike也被序列化了。序列化还可以确保每一个对象只会被反序列化一次，即使相同的对象可能会在整个对象图中被引用多次。因此，即使youssef引用了sheryl，而sheryl同时也引用了youssef，每一个对象仍然只会出现一次，正如它们在实际的对象图中的情况一样。你不需要编写任何支持该特性的代码就可以享用它。

尽管如此，你可能还是想知道，既然实体Bean或JDO被喻为拥有自动序列化的能力，那么为什么作为一个J2EE程序员，你还需要去操心序列化的方方面面呢？毕竟，你不是在将数据存储到文件中，你是在将数据存储到关系型数据库中。因此，为什么要劳烦地去使用序列化呢？

因为，不管你是否真的喜欢，你确实需要大量地使用它。

序列化是构件J2EE其他部分的一块基石。例如，JRMP之上的RMI使用序列化作为其创建远程调用的编组框架（请参阅第 3 章）。Servlet容器经常在容器关闭时，使用序列化来将会话状态存储到硬盘上，以便在重新启动之后可以恢复那些活动的会话，从而维持了容器从来都没有被关闭的假象。许多EJB容器使用序列化在远程（或本地）调用之间将bean实例钝化到硬盘上。JMS使用序列化来支持ObjectMessage。因此，开发者需要了解序列化规范中的细节，以避免被序列化对象数据的某些微妙行为所“惊吓”。

使用序列化作为 RMI 的编组框架也存在着副作用，例如，由于序列化是不关心机密性的，因此所有通过 RMI 传递的参数事实上都是以纯文本的形式被序列化的。如果机密性是某个基于 EJB 的系统所关心的事情，那么这就会成为一个问题。遗憾的是，对于以可移植的方式提供 RMI 的桩与 EJB 容器之间的机密性，EJB 规范并没有提供任何框架或规则；没有任何方式可以自定义被 RMI 管道用来进行客户端和服务器之间通信的通道（套接字）。然而，通过对如何序列化对象进行控制，可以修改个别的参数类型，以在 RMI 调用期间加密（或者至少是混淆）它们的敏感数据。

请注意，尽管许多 EJB 容器可能为钝化而使用了序列化，但是 EJB 规范并未对此进行强制要求，因此，某些容器可能利用了非标准化的机制来实现了钝化。理想情况下，这样的容器将用文档说明它所使用的这种机制，以及使用它的幕后原因，并将为修改它而提供相似类型的挂钩点（请参阅第 6 项）。序列化有大量你可以（而且在某些情况下是应该）利用的挂钩点。

SerialVerUID 域

序列化中对于每一个 Java 程序员都应该理解的第一个元素就是 SerialVerUID 域。当一个对象被序列化时，Java 对象序列化机制将基于该类全部的元数据——类中的域、域的访问范围、域的类型、类的方法、方法的参数、类的基类、所有实现了的接口等等，来计算整个类的哈希值（hash）。这将产生一个几乎唯一的值，它将在反序列化时被用来进行比较，以确保被反序列化出来的对象与被序列化对象具有相同的类型。

Java 类可以预先计算出这个哈希值，并将预先计算出来的值存储在一个私有的、静态的 long 类型的域 serialVerUID 中。在序列化过程中，如果这样的一个域存在，那么将会直接使用它的值而不是去计算其哈希值。（这里假设这个域的值是使用 JDK 的 serialver 工具预先计算出来的，而不是随机选取的值。）尽管对于基本的序列化来说这不是必须的，但是如果一个已经被序列化的对象进化了，那么计算这个值对于将“旧”对象反序列化为“新”类型来说，显得至关重要。

然而，即使是对进化不需要提供支持的类，你同样可以通过预先计算这个哈希值而获得小幅的性能提高，因为它节省了计算该值而必需的运行时刻的 CPU 周期。

自定义（writeObject 和 readObject）

对序列化提供支持和/或自定义的最简单的方式就是在类中编写私有的 writeObject 和 readObject 方法，其中每一个方法都接受一个流参数，分别是 ObjectOutputStream 和 ObjectInputStream。这些方法，如果有的话，将在序列化（writeObject）或反序列化（readObject）发生时被调用。在这些方法体内，开发者可以对类的内容的序列化进行完全地控制，或者是调用流的 defaultWriteObject 或 defaultReadObject 方法来执行缺省的序列化，然后再进行自己的操作。

因此，让我们再回到前面的 Person 的例子中，增加一个 totalWorth 域来跟踪个人的总资本净值。我们想把该值作为一种隐私，所以我们可以在序列化前混淆它，并在反序列化时重新计

算它：

```
import java.io.*;
import java.util.*;

class Person implements Serializable
{
  public String name;
  public Person spouse;
  public ArrayList children = new ArrayList();
  public double totalWorth;

  private void writeObject(ObjectOutputStream oos)
    throws IOException
  {
    totalWorth = obfuscateValue(totalWorth);
    oos.defaultWriteObject();
  }
  private void readObject(ObjectInputStream ois)
    throws IOException
  {
    ois.defaultReadObject();
    totalWorth = deobfuscateValue(totalWorth);
```

```
    }

    private static double obfuscateValue(double originalValue)
    { return originalValue * 2 – 1; } // Imagine your algorithm
                                      // here
    private static double deobfuscateValue(double hiddenValue)
    { return (hiddenValue - 1) / 2; } // Imagine your algorithm
                                      // here
}
```

很显然，在产品代码中，我们想要一种比上面所示要更强的算法，但是在这里，上面的算法可以服务于我们的目标。现在，在序列化时，一个 Person 实例的 totalWorth 将被扭曲以隐藏它的原始值，然后在反序列化时，再把它还原成其原始值。

替换（writeReplace 和 readResolve）

有时，直接修改给定类的被序列化的数据是不够的。例如，基于类的演化以及安全原因，有时要强制采用更严苛的措施，可能会为序列化和反序列化指定一个不同类型的类。这是通过在要被序列化的类上提供 writeReplace 和 readResolve 方法而得到处理的。

例如，考虑 CreditCard 类，它是电子商务在线系统的一部分。很自然地，我们想确保信用卡号码以加密的方式从客户端的 Web 浏览器发送到我们的接收 servlet，但是如果我们宣称信用卡号码确实是安全的，那么我们还需要确保在 servlet 和 EJB 容器之间也是被加密后通过网线传输的（千万不要信任网络，即使是在防火墙内部，请参阅第 60 项）。朝着这个目标，我们可以编写 CreditCard 类，为其指明一个加密过的替代者 EncrytedCreditCard，在网络上发送，并且，在接收端对 EncrytedCreditCard 进行反序列化时，可以将其转换成初始的

CreditCard 实例：

```
class CreditCard implements java.io.Serializable
{
  public CreditCard(Date expiration, String number)
  {
    System.out.println("CreditCard.<init>");

    this.expiration = expiration; this.number = number;
  }

  public Date expiration;
  public String number;

  private Object writeReplace()
    throws java.io.ObjectStreamException
  {
    System.out.println("CreditCard.writeReplace()");
    return new EncryptedCreditCard(expiration, number);
  }

  public String toString()
  {
```

```
    return "CreditCard: " + number + " (" + expiration + ")";
  }
}

class EncryptedCreditCard implements java.io.Serializable
{
  private Date expiration;
  private String encryptedNumber;

  public EncryptedCreditCard(Date exp, String number)
  {
    expiration = exp;
    encryptedNumber = encryptCreditCardNumber(number);
  }

  public Object readResolve()
    throws java.io.ObjectStreamException
  {
    return new CreditCard(expiration,
                decryptCreditCardNumber(encryptedNumber));
  }
```

```
    private String encryptCreditCardNumber(String num) { ... }

    private String decryptCreditCardNumber(String num) { ... }

}
```

这种方式的美妙之处在于，当使用CreditCard时，程序员可以忽略安全方面的需求，至少在开放的网络线路中发送CreditCard实例时是如此。事实上，你可以设计和实现完全使用“开放”的CreditCard实例的整个系统，然后在这些对象在序列化中的机密性成为系统要关注的问题时，再添加writeReplace/readResolve逻辑。（怎样加密数据以防止被很容易地观测完全是另外一回事，这超出了我们这里要讨论的范畴；请参阅第 65 项以了解其中的细节，或者研读一下*Java Security*[Oaks]以了解对这个主题更完备的讨论。）

更多的细节

你可以在标准的Java 2 SDK文档包，或者是《*Java平台的构件开发*》（*Component Development for the Java Platform*）[Halloway]以及《*基于服务器的Java编程*》（*Server-Based Java Programming*）[Neward]这些书中，找到更多的关于Java对象序列化的信息（包括对可序列化域API和其它序列化功能的讨论）。你还应该研读一下《*高效Java*》（*Effective Java*）[Bloch]，特别是看看第 54 项，作者在其中指出将一个类标记为可序列化，就意味着你悄悄地在这个类中引入了一个新的构造器，它接受一个字节数组作为其唯一的参数。如果你的类需要将维护数据约束作为其行为的一部分，那么你还要确保在从反序列化过程中构建对象时，要检查这些数据约束。

理解序列化所带来的好处超出了仅仅只是理解如何在 J2EE 中使用它；开发者经常要寻求各种方式以存储对象或是其它类型的数据，序列化就像是为此类事情量身定做的一样。对象可以被序列化并存储在表的 BLOB 列中，对象可以被序列化并作为一个 HTTP 请求或响应的内容体而被发送，等等。事实上，存储用户偏好的一种简便的方式（如果你不想或者不能使用 Preferences API 的话）就是将偏好置于一个 HashMap 中，并序列化这个 HashMap 和它的内容。关键之处在于，如果没有理解序列化自身是如何运作的，你就会做出最终会伤害到你

的决策——就像只为保护几个属性成员，而将所有的 RMI 调用都运行在 SSL 之上。

第 72 项：不要对抗垃圾收集器

我们都听过这句传统的至理名言——它被杂志上的文章、书籍、甚至是会议和巡回讲座大肆鼓吹：Java 对象分配是相当慢的，请最小化你所创建的对象的数量。遗憾的是，这些观点都是基于过时信息的，因此他们所做出的结论会导致 Java 程序员去做完全错误的事情。

在早期，即 Java 1.0 版本的那些日子里，JVM 垃圾收集器确实是很糟糕的。它是一个相当简单的、停顿所有处理的、标记-清除式的垃圾收集器，它会导致在执行过程中频繁可见的暂停。JVM 在达到其最大的内存极限之前，是不会进行任何垃圾收集行为的，在达到该极限之后，它将剖析整个 JVM 堆以找出所有值得收集的对象，并将它们标记为不可用（因此也就符合了收集的条件），然后再次遍历堆，一次性收集它们。因为 VM 需要保持每样事物都完全地正确并且同步，因此所有正在执行的线程——也就是那些正在解释 Java 字节码的线程，因为我们在那些日子里并没有任何 JIT 技术——就不得不在垃圾收集发生时暂时停止下来。在很大程度上，1.0 版本造就了 Java 是一种“为性能考虑而应避免的平台”的名声。

当 Java 从 1.1 版本提升到了 1.2 版本，开发者开始寻找规避和绕过垃圾收集器的方式——任何我们能够做的、可以阻止垃圾收集器去剖析堆并去执行它那令人生厌的职责的事情都是幸福的事。既然对象分配是垃圾收集器最频繁可能被触发的地方，那么尽可能地最小化调用 new 的数量就显得很有意义了。于是大量的想法迸发了出来，包括多用途的对象池（稍后我们将再次造访它）。

然而，当我们考虑那句传统的至理名言时，就会发现在这中间所发生的明显是在检讨过去：Java 垃圾收集算法和分配策略变得更好了。到了 Hotspot VM 发布的时候，特别是 J2SE1.4（以及 1.5）发布的时候，我们不再相信对象分配是昂贵的这一断言了。相似地，除了可终结对象（finalizable object）外，对象回收是昂贵的这一断言也不再为真了。实际上，在现代的 VM 中，对象分配和回收所需的开销，只是以前的 VM 所需开销的一小部分；要想理解这是为什么，就要对各种不同的垃圾收集算法作一个简要的浏览。

最流行的一种收集器是*复制收集器*（*copying collector*），之所以这样命名是因为它把堆分解为两个相等的一半，一个被标注为 Fromspace，另一个被标注为 Tospace。对象是从 Fromspace 被分配出来的，当一个垃圾收集过程被调用时，垃圾收集器执行其正常的工作，从引用集合的根开始，查找引用对象。然而，这一次不是直接标记对象，而是跟随其链接，并慢慢地将每一个被引用对象从 Fromspace 拷贝到 Tospace（这正是它们名字的由来），正如图 8.3 所示：

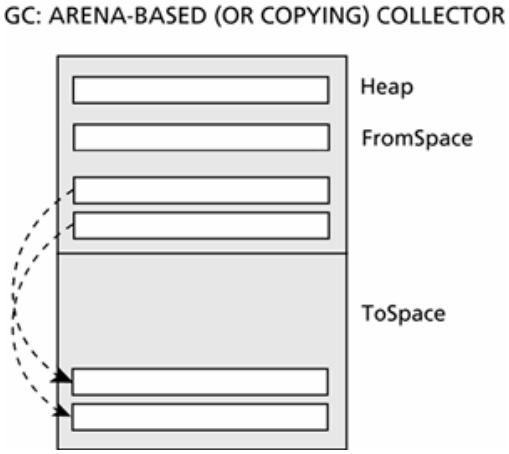

图 8.3 基于竞技场的垃圾收集

通过这样做，按照定义，任何留在 Fromspace 内的对象都是不可用的，因此也是符合回收条件的。因此垃圾收集器可以假设整个 Fromspace 都是符合回收条件的，并且可以将整个空间清除。很自然地，在 Fromspace 中的任何可终结对象必须在我们清除 Fromspace 之前被处理掉，这再次证明了对垃圾收集器来说，终结器是多么大的一个痛苦。然而，排除终结器场景，复制收集器的这种立即执行的行为意味着它释放被分配对象的速度非常地快，并且，作为一个附带的好处，它可以自动地使对象更紧凑地位于 Tospace 中。一旦这次过程结束，我们就翻转其名称，因此，当前的 Tospace 就变成了 Fromspace，然后我们将等待下一次垃圾收集过程。

然而，复制收集器并不完美，它最显著的缺点是相当容易被察觉的：堆需要两倍的内存，因为我们需要维护足够的空间来复制整个堆到 Tospace 中。这意味着如果我们有一个应用，它平均消耗 128MB 的对象，那么 JVM 仅为堆就需要至少分配 256MB 内存。正如你可能已经想到的那样，这显得有点昂贵，并不是特别地吸引人。

另一个通用的垃圾收集算法是*分代收集器*（*generational collector*），它所依靠的思想是：大

多数对象都是短期存活的，尽管一小部分对象会存活很长的一段时间。因此，对象按照它们的代（*generation*）而被分类，每一块内存包含了生存期大致相同的对象。基于这个假设，分代收集器将新近创建的对象置于*年轻的一代*（*young generation*）中，它是最近分配的所有对象的集合。当要求执行垃圾收集时，收集器只需要扫描最年轻的一代以查找已经不被使用的对象，因为那里是它们最有可能出现的地方。只有当没有从年轻的一代中收集到足够的空间时，收集器才会开始在分代栈中向上移动，去检查在紧挨着的较老的一代中可以被收集的对象。如果一个对象存活了相当长的一段时间（通常由成功地从垃圾收集过程中存活下来的次数来度量），它将被移动到相邻的较老的一代中。大多数分代收集器都只有几代，两代或者至多三代，尽管在理论上可以使用无限多的代。（请参阅图 8.4）

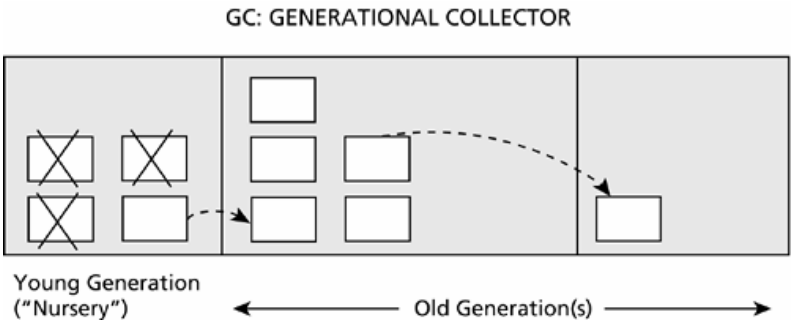

图 8.4 分代垃圾收集器

分代收集器最直接且最明显的好处就是在一次收集过程中不用扫描整个堆，这意味着垃圾收集过程可以更快地执行。然而，它同样也有缺点，在较老的一代中的对象在很长的一段时间之内都不会被收集，特别是如果年轻一代保持足够宽敞，总是能够满足任何新的分配请求时，就更是如此了。最终，由收集器所决定的，我们将承担必须进行两次而不是一次收集的风险——如果穿越年轻一代的一次收集不能满足分配需求，那么，就需要另一次，这一次就必须要穿越较老的一代了（这可能意味着还要再次穿越年轻的一代），这将大大地扩展垃圾收集的时间。

还有其它形式的垃圾收集算法，包括*基于竞技场的收集器*（*arena-based collector*）（从*竞技场*分配对象，竞技场是一块内存，对象从其中被分配出来，它非常像操作系统以页面来管理虚拟内存），*引用计数收集器*（*reference-counting collectors*）（它已经被弃用了，因为引用计数收集器不能处理循环引用场景），以及其它的收集器。要想更广泛地了解垃圾收集的更多信息，请参阅《*垃圾收集Garbage Collection*》一书[Jones/Lims]；要想更广泛地了解Sun的

收集器的更多信息，请参阅Sun 的Hotspot文档，可以从http://java.sun.com/docs/hotspot/gc/index.html获得。

请记住标记-清除、复制和分代收集器只是种类几乎无限的所有可能的垃圾收集算法中的三个，很显然，程序员的选择可能在有助于某种类型的垃圾收集器的同时，会损害另外一种垃圾收集器。特别是，程序员应该从对象池中分配对象以加速分配过程这句传统的至理名言，将完全依赖于运行在底层的垃圾收集器的类型。因此，如果（在已经考虑了第 11 项之后）你努力编写可移植的代码，那么你就不能假设你对底层的垃圾收集器的所有事情都了如指掌，因此，你也就不能确切知道是否要用池缓存对象。一句话，你必须相信垃圾收集器在做正确的事情。

然而，这并不意味着你完全没有了选择。根据你所使用的 JVM，某些数量的配置仍旧是可能的——从 J2SE1.3 以来，每一个相继版本都提供了对垃圾收集的行为和可视性的更多的控制。下面几个段落描述了 Sun 的 JVM 的某些细节，因此如果你想要保持提供商中立的立场（请参阅第 11 项），那么就请闭上眼，大声歌唱一会儿，然后直接跳到这几个段落结束的地方。

从J2SE 1.3 开始，Sun Hotspot JVM垃圾收集机制就是一个混合的机制。它用分代收集器将年轻的对象与年老的对象分离开，从而将堆分成了两个竞技场。对象首先被分配到了*托管所*（*nursery*），并且只有当它们在经历了一定次数的垃圾收集过程后仍然存活时，才将其迁移到年老的一代中。年轻的一代是通过使用一个分代垃圾收集器来管理的，它由一个*eden space*，和两个*survivor space*组成。*eden space*是对象的出生地，作为分代设计的一部分，对象将被复制到*survivor space*中。（这样，*eden space*就起到了一个永久性Fromspace的作用，而*survivor space*则在Fromspace和Tospace之间切换，这样就可以在两次垃圾收集过程之间的任意时刻，都保证有一个*survivor space*为空。）

然而，因为被分配到年老的一代中的对象只有极小的可能性是准备好可以被回收的，所以它改而使用标记-清除-压缩算法；除了当对象被标记为继续存活后要进行压缩以使年老一代中的“空洞”最小化之外，该标记-清除-压缩算法与标准的标记-清除算法的作用是一样的。标记-清除-压缩比复制收集器要慢许多，但是它有一个优点，就是能够利用整个堆，而不像复

制收集器那样需要将其划分为Fromspace和Tospace才能工作。

另外，年老一代的空间中有一部分空间被保留下来给VM使用，称为永久空间（*Permanent space*），因为JVM将在其中为自己分配必需要使用的Java对象，例如Reflection对象。

就像已经描述过的，分代收集器的一个主要好处就是不需要对整个堆完整的遍历——经常性地，一次单一的对年轻一代的遍历就可以返回足够的自由空间满足近期的内存请求。因此，在Sun JVM中的垃圾收集过程形成了两种风格，一个发生在年轻一代中的较小收集（*minor collection*）和一个要同时穿越年轻一代和年老一代的完全收集（*full collection*）。Sun JVM的目标是保持年轻一代中的自由空间足够多，使得如果不是全部的话，那么至少也是大部分分配请求可以在年轻一代中得到满足，因为在那里进行的垃圾收集过程比在年老一代中进行的垃圾收集过程的开销要小得多。

遗憾的是，我们不能去猜测垃圾收集器正在如何运行；通过使用JVM标准的-verbose:gc选项，我们可以在JVM运行到了垃圾收集阶段时去观察它，因为每一次收集，不论是较小收集还是完整收集，都会作为应用执行而被写到java的控制台窗口中。在 1.4.x的JVm下，这个详细的垃圾收集行为的输出看起来与下面的内容类似：

```
[GC 325407K->83000K(776768K), 0.3400514 secs]
[GC 325816K->83372K(776768K), 0.3054352 secs]
[Full GC 267628K->83769K(776768k), 1.9542351 secs]
```

在这种情况下，我们可以看到两个较小收集和一个完整收集。解析这个输出相当容易：箭头前面和后面的数字分别表明了在垃圾收集之前和之后所有存活对象联合在一起的尺寸，在括号中的第三个数字是 Java 堆的总计可用空间，它是堆减去年轻一代的两个 *survivor space* 之后的空间（因为我们需要这部分空间以使得在年轻一代中的复制收集器能够工作）。最后一个数字当然就是该次垃圾收集过程所花费的时间。

观察这些输出只是了解垃圾收集器在你的应用中如何运做的第一步。例如，如果你看到了比 GC 项要多许多的 Full GC 项，那么这就是一个极大的危险信号，告诉你年轻一代中的对象

已经饱和了，无法找到足够的空间满足对象分配请求了。出现这种情况的部分原因可以归咎于这样一个事实：在年轻一代中的对象，如果被年老一代中的对象所引用，那么它就不能被回收，这是对徒然地使用对象池的又一个争论。无论是什么原因，如果堆没有大到能够满足每一个分配请求，那么 JVM 将延伸它的操作系统进程的足迹（*footprint*），以此来重新调整堆的，而这个调整的上限是通过-Xmx 选项指定的 JVM 允许使用的内存最大值。

然而，如果年老一代有充足的空间，而仅仅只是年轻一代饱和了，那么我们可以通过使用某些未被列入文档的 Sun 相关的 JVM 调整标志来重新调整年轻一代的尺寸。缺省地，年轻一代被设置为占整个堆的尺寸的四分之一；如果你经常性地想改变这个比例，以扩展年轻一代的尺寸，那么你就可以使用-XX:NewRatio 标志，传递给该标志的值是年轻一代与年老一代之间的比例。例如，-XX:NewRatio=3 设置的比例是年轻一代比年老一代为 1:3，它也就是缺省的设置。

如果你在寻求比整个堆尺寸的简单倍数这种粒度要更细的控制方式，那么另外两个标志，NewSize 和 MaxNewSize，可以用来设置年轻一代的边界，就像-ms 和-mx 标志设置整个堆时所做的一样。尽管可以这样做，但是对这些标志的值还是要格外当心——当年轻一代大于整个堆尺寸的一半时，就意味着为年轻一代分配了许多空闲空间。在 Solaris 的 JVM 上，缺省的 NewSize 是 2172K；在 x86 的 JVM 上，是 640K。MaxNewSize 的缺省值在两个平台上都是 32M，这通常显得过小了，特别是如果你已经将-mx 选项设置得超过了 64M（缺省值）时更是如此。你可以用 OldSize 选项对年老的一代做同样的事情，其缺省值为 1408K，还可以使用 PermSize 来设置永久空间，其缺省值为 1M。

在一个对象必须被迁移到年老一代之前，究竟要经历多少次对年轻一代的垃圾收集过程呢？如果你对此很好奇，那么就可以用打开的 PrintTenuringDistribution 标志(-XX:+PrintTenuringDistribution)来运行程序。这种运用可不适合那些懵懂的人，它可能只对那些对此领域更为熟悉的人才显得有用。另一方面，对这些统计数据感到习惯是熟悉垃圾收集器的一种好方式。

顺便说一句，这种配置方式比试图以编程方式来运行要好得多。例如，有许多开发者，他们相信自己比垃圾收集器更清楚地知道何时是运行垃圾收集过程的好时机，这些人会在半正规

的基础上调用 System.gc，而这通常发生在他们释放了一个大型对象或一个对象集合之后。“我希望那些对象尽可能快地被清除掉，”他们争辩道，“因此我需要告诉垃圾收集器该去运行了，这样它就会立刻清除掉这些对象。”

这种争辩有几个主要的缺陷。请注意，文档中描述道：对 System.gc 的调用并不是一个让垃圾收集器去运行的命令，而只是一个请求。垃圾收集器如果喜欢的话完全可以忽略掉它。然而，在你义愤填膺地急于辩解“他们怎么敢假定这种行为？”之前，请先静下来考虑一下：如果一个对 System.gc 的调用要强制执行一次新的垃圾收集过程，那么如果在垃圾收集器已经处于一次收集过程中的时候，又会怎么样呢？它是否应该仅仅是因为你这样说了，就放弃掉所有已经完成的工作，去重头开始？这实际上将导致更慢的垃圾收集时间，并且与你所描述的目的完全背道而驰。更重要的是，收集器也许并不能够高效地收集你刚刚释放掉的对象，例如，在基于竞技场的垃圾收集器中，你可能会发现：你认为符合被回收条件的对象仍然位于包含可获得对象的竞技场中，并且收集器可能会决定不回收那些对象。你在这里的“帮助”并不是必需的，也不会有人感激。

事实上，我知道有一个基于Web的系统，编写它的程序员们只要他们认为恰当，就放置“强制”垃圾收集器去运行的调用，结果弄得这类代码到处都是。在一个老资格的开发者的建议下，他们试着运行了一个将这些调用全部注释掉的版本，结果发现它运行起来比以前的版本快了大约 20%，并且只占用更小的内存。即使是在Sun的文档中，也指出了在System.gc调用幕后那令人恐慌的事实：“这些调用强制进行全面的收集，从而抑制了大型系统的可扩展性。”[1] 正因为这个原因，Sun的JVM有一个完全禁止System.gc的选项（“不支持”选项-XX:+DisableExplicitGC），我强烈建议只有在你的许多狂热的Java开发者仍旧将System.gc放置到他们的代码中时，你再去考虑使用它。

作为另一个例子，请再次考虑有关对象池的传统的至理名言：“分配一个Java对象的开销昂贵得令人害怕。”实际上，已经证明，在Sun的Hotspot VM下，构建一个新对象的开销大约是 10 个本地CPU指令周期[2]加上在对象构造器中花掉的时间。这可以很轻松地与最好的C++编译器比肩，希望这能够平息“分配对象是昂贵的”这一想法——只要对象的构造器在进行

[1] 引自http://java.sun.com/docs/hotspot/gc/index.html

[2] 基于Y. Srinivas Ramakrishna的“在Java Hotspot虚拟机中的自动内存管理”（JavaOne, 2002）一文所表述的信息。

构造时不会太慢或者是太复杂，那么构造一个新对象的开销就远比从池中获取一个对象要高效得多。（请记住，即使对象已经从池中被获取了出来，它仍旧需要被初始化到一个良好的启动状态，而这正是构造器的典型行为，因此池真正能够避免的只是构造的开销——而我们已经看到了它是微不足道的。）

这并不是在说所有的对象池机制都是糟糕的，只是说它不是杂志上那些文章描述的万能药，那些文章更加注重于卖弄特殊的池机制实现而不是讨论何时使用池最好。例如，J2EE 竭尽全力地为昂贵的连接提供连接池机制（例如数据库连接，它是资源池机制的招牌产物），因为获取一个数据库连接的开销可能是非常昂贵的操作——不是因为创建对象所花费的时间，而是因为在底层执行的工作，包括在网络之间的往返、处理认证证书等等。

事实上，对象池机制——或者更准确地讲，是资源对象管理，因为它是我们要描述的一个非常宽泛的术语——可以分为下面几个主要的类型[3]：

- *对象工厂：*一个对象工厂管理着无限的、廉价的资源，其典型的特征就是拥有一个静态方法，用来简化客户端创建和构造对象的过程。例如，JAXP API 描述了一个创建 SAX 和 DOM XML 解析器实例的对象工厂。创建超过一个的解析器不会有任何内在的开销（当然除了内存），因此假设我们并不在乎其不修改代码就可以将 XML 解析器换进换出的能力，那么我们也可以回过头直接调用构造器。但是，对象工厂方式还可以使得客户端不知晓实际被创建的实例。本质上，这里并没有任何池机制。
- *对象银行。*在对象银行中的资源是无限的但是创建它是昂贵的，它允许客户端创建并存储以后将会用到的资源对象，这样就避免了对这些对象不必要的创建和销毁。对象银行用得并不多，因为大多数创建开销昂贵的资源都是有限的（因此，也就落入了下面的对象池类型中）。然而，一个可能的对象银行的例子是它被用来管理到某台服务器的套接字连接，因为套接字在极大程度上是无限的，然而它需要网络的协调才能创建，因此从某种意义上讲它是昂贵的。
- *有限对象管理器：*有时资源是有限的，但是其创建和销毁开销却是廉价的。在这些情况下，有限对象管理器选择创建和销毁对象，因为只是分配和回收它们比在池中缓存它们要快，但是有限对象管理器自身必需明了已被分配的资源，以保证客户端请求不会超过

[3] 感谢Brian Maso，他首先提出了这种分类，并且很客气地允许我“使用”它。

资源的上限。许可证管理器经常被划归到了这一类中，当对给定类型对象的使用超出某个确定的上界时，它会被阻止（当然，除非你购买了一个更大的许可证）。

- *对象池：*当一项有限资源其创建开销很昂贵时，不仅限制被分配的资源对象的数量很有意义，而且（当客户端不再使用它们时）紧紧地持有已经被创建的对象，使得以后感兴趣的客户端可以重用它们的做法同样很有意义。对象池机制的经典例子就是数据库连接池机制——大多数数据库现在都基于允许到数据库实例的并发连接数来收取发证费，并且每一个连接自身在被打开以供使用时，还需要一定数量的协商处理。

这项讨论中具有讽刺意味的部分是当我们跳出来看时，很明显，EJB 规范在某些问题上犯了错误，因为它过于强调对象池机制了——无状态会话 Bean 是一种创建开销很廉价的资源，因为它不为任何客户端持有任何状态，因此缓存无状态会话 Bean 实例到池中也许是不必要的。相反，容器也许应该更多地信任垃圾收集器，直接在需要时创建新的会话 Bean 实例，并且在要求清除时让旧的实例被垃圾收集器捕捉到。（为了安全考虑，用于创建无状态会话 Bean 实例的对象工厂可能应该有一个它能够创建的 Bean 实例数量的上边界，以避免攻击者同时创建过多数量的无状态调用以发起拒绝服务攻击；大多数 EJB 实现都提供了这个上界。）

某些读者，在看了上面的列表之后，毫无疑问，肯定会随意地将四种类型置之一旁，对整个“对象池机制”的概念进行挑剔；但是在这么做之前，请记住，它们每一种都在处理一种不同的场景，并且都根据其所要表示的资源相对的昂贵性和有限性，使用了不同的算法。（怎样构建高效的对象池的细节，请参阅第 74 项。）

那么，既然这样，现在怎么办呢？我们到底应该如何去编写高效使用内存的代码呢？

对初学者来说，如果你计划编写提供商中立的代码，你就必须信任垃圾收集器。它真的是相当的简单，而且它被证明在大多数情况下确实是最好的选择。应该以这种方式来看待它：编写 JVM 实现的开发者编写了垃圾收集器，这些垃圾收集器在“正常”情况下运行得最好，而在面对棘手的或者奇异的对象生命周期场景时，可能会碰上问题。不要有意地编码去尝试愚弄垃圾收集器，因为你只会搞乱它，并且产生更坏的结果。

然而，如果你已经了解了你正在使用的 JVM，那么全权委托它去把对象缓存到池中就并不

是必需的了（假设你的 JVM 推荐这样做，Sun Hotspot 开发组并没有为它的 JVM 做过这样的推荐）。

通常，只有当资源对象变为有限的、昂贵的、或者二者皆是的情况下，有意地编码管理它们依然是比较好的做法。在大多数 JVM 中的垃圾收集器都被优化为可以处理更为普通得多的情况，也就是对于外部资源没有任何特殊需求的对象。

这里全部的重点就是要努力避免那些你相信能够使你的垃圾收集更为高效的“技巧”，除非有任何过硬的科学证据能够证明这些技巧是必需的或者是值得采用的（请参阅第 10 项）。例如，你最近在什么时候对字符串连接进行过分析，以观察使用 StringBuffer 是否确实比简单地将两个 String 加到一起要更高效。

第 73 项：优选容器管理的资源管理

从 servlet/JSP 环境转移到 EJB 环境的 Java 开发者，经常会因为相比较之下，EJB 环境是如此受限而感到惊讶。企业 Bean 是不允许启动线程的；企业 Bean 是不允许执行基于文件的 I/O 的；企业 Bean 是不允许手动实现任何种类的同步行为的；企业 Bean 不允许建立服务器套接字，或者反射、或者加载本地库、或者……

它让你产生了疑惑，你到底能做些什么？

当然，这些限制是有理由的。为了维护系统使其尽可能可扩展，容器于是趁机接手资源管理作为它的一部分职责。这样做是因为，作为将不同种类的程序和企业系统统一化的力量，容器通常可以对全面的资源需求有一个更好的认识，而不像你的代码只能局限与自己的视图。

例如，考虑一下线程机制：在许多 servlet 书籍中，经常会看到这样的场景：一个 servlet 在启动之后，创建了一个守护线程，在后台去执行某种类型的处理。考虑一下这对 servlet 容器意味着什么。Servlet 容器自身需要管理线程以获取在 80 端口或 443 端口（或者是其它的端口）上进来的请求，并且用适当的过滤器链和 servlet 代码来处理它们。在直觉上，我们

知道我们想让容器去这么做——或者，更特殊地，我们想让 servlet 容器对正在执行的 servlet 的最大数量严加控制，因为不这样做就意味着 servlet 容器对拒绝服务攻击是开放的。

（如果这种联系对你来说并不明显，那么请考虑下面的场景。假设我有一个 servlet 框架，它为每一个进入的请求都创建一个新的线程。一个攻击者创建了一个无限循环的小程序，它对 Web 应用创建 HTTP 请求。该 servlet 框架继续不断地创建新线程，直到机器因同时执行太多的线程而崩溃。）

但是现在你又开始为从你的 servlet 中创建线程而感到恼火了。因为 Servlet 规范没有为 servlet 开发者提供任何方式来集成 servlet 容器的线程管理方案，根据定义，这些线程在 servlet 容器控制范围之外。因此如果系统管理员对在一个 servlet 容器中的线程池设置了上限 N，这个 N 是他在大量的调整和测试之后计算出来的，它的值对该特定容器和平台来说是最优化的数字，那么底层的虚拟机实际运行的就是 N+1 个线程——servlet 容器知道的 N 个线程，以及一个它不知道的线程。突然，你错过了最佳的使用点，并且开始在收益递减的道路上滑下去了。如果不是一个线程而是两个或者更多的线程，你可能很快就会陷入线程切换的时间比实际工作的时间还要长的境地。

如果这个捉弄人的线程是在 servlet 上下文被启动时创建的，但是程序员（由于有 bug、无知或者默然）没有在关闭 servlet 上下文时关闭线程，那么上面的故事就变得更糟糕了。这意味着在每一次 servlet 上下文重启动时（这可能因为各种不同的、由系统管理员所要求的原因而发生），都会创建出一个新的线程出来而同时并没有回收旧线程。垃圾收集在这里不会有所帮助——线程是极少数几个即使所有对其正式的引用都被移除之后仍然能够存活的资源之一。（直到线程自身消亡并且所有强引用都被移除之前，Thread 对象都不会被收集。）另外，来自该线程的任何对象引用将继续保持是强引用的，这意味着它们不能被当作垃圾收集，这也意味着 JVM 现在占有的内存比它应该占有的要大；还有，这些 Thread 对象表示在操作系统内的线程资源，它们比简单的基于内存的对象要重的多；还有……好，现在你理解了吧。

这才刚开始，我们甚至还没有开始谈论 servlet 是否应该从线程池中去创建线程；该主题在第 68 项中有更多的细节描述。然而，概括其意思就是你如何能在实际测试某特定虚拟机之

前，就能够知道在给定系统上使用线程池是否会更好一些？某些 JVM 使用的线程系统在使用线程池时工作得更好，而其它的 JVM 则掩盖缓存线程到池中，并且当 Java 代码自己来缓存线程到池中时，他们的实际效率就更差了。

J2EE 出于大量的理由详细说明了资源管理，主要是因为当在一个基于容器的环境中工作时，让容器管理这些资源并让构件在必要时借用它们会显得更高效和更具可扩展性。在某些情况下，就像你在 EJB 规范中看到的，将某些能力赋予开发者实际上是弊大于利。例如，直接管理线程并不是很简单的任务，并且会引入各种各样的同步问题，而这正是 EJB 规范试图让开发者避免的问题。（尽管 Servlet 规范对此持有一种更为悲观的观点，但是开发者仍被劝阻不要直接创建和操纵线程。）相类似地，无论容器希望使用什么样的生命周期策略，试图直接管理构件的生命周期通常都会损害这种策略。

我们可以理解为什么如此多的书籍和文章都建议从 Servlet 中创建一个线程；很长一段时间以来，J2EE 的空间中都缺失了一块，那是一种能力，一种给与 J2EE 构件某种“活动”状态的能力。所有“传统的”J2EE 构件（servlet、EJB、JMS）都需要一个来自客户端的控制逻辑线程——客户端通过调用它们来借用一个容器中的线程去执行某些动作。在 EJB2.1 规范之前，没有任何方式可以创建一个与某个 J2EE 构件绑定在一起的活动线程，而不使该线程成为该构件的某种类型的客户端。例如，你不能大约每十分钟就“唤醒”在 EJB 容器中的一个 bean，然后去检查数据库表中的新的数据项。

最接近的相似做法是创建一个客户进程，它运行在与 EJB 容器相同的机器上，该进程将大约每十分钟调用到 bean 上。这并不是最优雅的解决方案，但是你做了你必须做的。唯一的其它选择是去寻找对这些资源管理规则更懈怠的 EJB 容器，这样就可以允许它们不去理会规范对此的相关禁令而去创建一个线程（这样也就消除了任何可移植性），或者创建一个在前面描述过的守护或服务应用进程，它可以在希望的时刻调用到 EJB 容器中。

只有在现在，作为 EJB2.1 的一部分，程序员才拥有了某种以定时器服务（Timer Service）形式出现的解决方案，它允许你为一项活动注册一个由 EJB 容器创建的请求——你创建一个定时器，要么指定一个周期性发生的调用序列（例如，每 5 分钟），要么指定到某个特定的时间（例如，从现在开始的 5 分钟后），并指定你希望容器调用的 bean；然后容器在时间

流逝适当的量之后，就会必然地产生该调用。多亏有了它，对任何 J2EE 规范来说，都没有任何强制性的理由去让它们因为轮询或超时目的而产生自己的线程。

这种放任自由（laissez-faire[4]）的态度不仅仅局限于线程。网络连接是另一种放任自由的资源；除非你以某种方式与容器的连接管理机制实现了绑定，否则你都不要自己打开或者关闭套接字。除了你可能会打开容器自己以后想要监听的套接字端口这一表面原因之外，大多数JDK1.4 以后的容器都使用java.nio类库来高效地处理进入的连接，如果你在这里进行干预，那么只会自找麻烦。如果你想监听到外部的连接，那么要么使用一个在J2EE中描述的已指定的通信层（RMI、JMS或者任何一种确定的互联网协议，像servlet/HTTP或JavaMail/SMTP），要么就在容器外进行你的通信，并且桥接到RMI、JMS、servlet或JavaMail。

容器将要帮你管理的经典的“其它”资源是数据库连接（包括其它连接式的资源，诸如 Connector 连接和 JMS 连接）。通常，在 EJB 容器内部这没什么问题，但是当从一个直接处理 JMS 的 Queue 或 Topic 实例的 servlet 容器或应用中去操作资源时，EJB 容器并不会为你去完全执行神奇的数据库连接池机制。当从客户端的 Swing 应用中操作资源时，情况也相同，至少看起来是这样。

因为大家有一条并非没有根据但是并不总是正确（请参阅第 72 项）的普遍信条，即连接池机制是挺不错的，所以担心外部数据库连接的 servlet 开发者开始编写他们自己的数据库连接池系统或者是下载一个已有的连接池系统（例如，可以在 Jakarta 公共项目中就能找到一个非常流行的实现）。他们将连接池实现部署到他们的 Web 应用中，并且长舒一口气，因为他们知道现在所有的数据库连接都已经是可以循环利用的了，而且现在他们的系统具备了之前并不具备的可扩展性。

遗憾的是，事情并非像他们想象的那么生动逼真。首先，如果连接池被部署到了 Web 应用自身中，那么它很有可能只会为该特定 Web 应用缓存连接到池中——因为类加载器有边界（请参阅第 70 项以了解更多细节），所以在大多数情况下，每一个连接池都有它自己的静态连接集合。这意味着实际的 Connection 实例比希望存在的实例要多许多。其次，依赖于连接池是如何被编码的，连接完全有可能被丢失——如果池没有利用 Reference 对象（请参阅

[4] 法语中或多或少地类似于“放任自由（keep your hands off）”的词。

第 74 项以了解更多细节），任何没有明确地返回一个 Connection 对象到池中的代码都会“泄露”一个不能被循环使用或垃圾收集的 Connection。第三，当使用最新的 JDBC 驱动时，依赖于驱动的实现细节，你不需要使用外来的连接池机制的实现。JDBC3.0 规范建议驱动可以并应该直接缓存连接到池中，甚至不需要经过 JNDI 指明的 DataSource 就可以实现。

除了个别异常情况之外，开发者几乎没有什么理由去“手动”管理在 J2EE 容器中的资源——底层的规范或容器自身的实现通常将更好地完成这项工作，这要归功于它们具有更多的有关资源在内部如何被操作的知识。然而通过剥离池机制，你也可以潜在地降低垃圾收集器的工作量，因为在你的代码中只需跟踪更少的长期存活的对象。

第 74 项：使用 Reference 对象来扩展垃圾收集行为

那么，在阅读过第 72 项后，当你在面对只是因垃圾收集器还未开始工作而等待的情形时——你需要知道对象何时不再被引用，以积极地释放资源（请参阅第 67 项）——一个好的 Java 程序员应该做些什么呢？

你的第一反应可能是去翻看 Java 语言的规范，看看在该语言中有什么样的工具可用——毕竟，如果你曾经被要求初始化了某事物（这正是构造器要做的事），那么你肯定也必须要去清除它，对吗？而且毫无疑问，Java 提供了终结器（*finalizer*）的概念，它是一个在对象被清除时由垃圾收集器调用的方法。但是在你决定到处去编写终结器之前，请先深吸一口气，然后读下去。

就像 Joshua Bloch 在他优秀的《高效 *Java*（*Effective Java*）》一书中所讨论的，终结器通常是一个糟糕的想法。对于初学者来说，你不会得到关于何时终结器会被执行的保证——如果它会执行的话。例如，如果 JVM 正在关闭，对于那些正等待被终结的对象而言，完全被忽略了，从而使得终结器永远得不到调用；这都是冗余以及不必要的，因为 JVM 无论如何都要关闭了。其次，终结器为垃圾收集器增加了更多的工作；现在不是直接将已分配的对象放回到可用内存的缓存池中（当然，这依赖于垃圾收集算法是如何实现的，请参阅第 72 项），而是必须将对象置于一个对象队列中，这些对象都是要调用终结器的对象。这不仅降低了回

收的速度，而且也降低了分配的速度，因为这需要在对象被创建时进行标记。这还意味着那些需要终结的对象将会保存得更长，从而产生比实际需求要更大的内存堆。有这样一个故事，一个 Java 程序员在运行一个给定的程序时，总是陷入 OutOfMemoryError 的麻烦中。弄清楚了之后发现，过多的对象需要被终结，终结器线程完全跟不上——最终 JVM 为了兑现新的分配请求而耗尽了所有空闲堆。

更重要的是，从服务器端的角度来看，JVM 没有提供任何保障来确保终结器被调用的顺序或时机。例如，给出三个对象，A、B 和 C，其中 A 引用 B，B 引用 C，如果每一个对象都有一个终结器，那么表面看起来，JVM 调用终结器的顺序是 A、B 然后是 C 会比较合乎逻辑；然而事实并不是这样。事实上，完全有可能 C 的终结器将会被首先调用，然后是 B，最后是 A，这意味着如果 B 在 B 的终结器中调用了 C，我们只是在一个对象上调用了一个方法，而这将引发死锁。

好像所有这些也并不算太糟，但是在下面的情景中将会发生什么呢？

```
public class Resurrector
{
  private static ArrayList deadObjects = new ArrayList();

  protected void finalize()
  {
    try
    {
      deadObjects.add(this); // Arise, Lazarus!
    }
    finally
```

```
    {
      super.finalize();
    }
  }
}
```

因为在终结器中我们使对象再次可以从引用的一个根集合中获得，所以我们有效地重生了一个死对象。但是就像所有好的恐怖片一样，重生并非是对已消亡事物的完全重现；在这种情况下，这个对象已经被终结过一次了，重生并不能改变这一点——因此，现在你有了一个一只脚已经踏进坟墓中的对象，只是在等待着对它的最后一个引用被移除，而这一旦发生，对象就会立即被释放，并不会再次调用终结器，因此也就没有给它机会作任何种类的清除。天哪！

请参考《高效*Java*（*Effective Java*）》[Bloch]中有关如何正确地实现终结器的提示；在很大程度上，企业Java开发者希望完全避免终结器。那么，现在怎么办？

在某些情况下，各种各样的 J2EE 规范都提供了事件方法来告知你何时对象将被销毁掉。例如，servlet 有一个 init 方法，它只在 servlet 提交其第一个请求前被调用，而 destroy 方法只在servlet容器打算要将servlet实例移交给垃圾收集器去循环利用之前会被调用。从Servlet2.3规范开始，我们也可以通过创建一个 ServletContextListener 来了解整个 Web 应用自身的生命周期，容器承诺会在启动和关闭时调用这个监听器。EJB bean 通过 ejbCreate 和 ejbRemove 方法也提供了相似的支持，尽管它是基于每一种 Bean 的基础之上的。

遗憾的是，我们对其生命周期感兴趣的所有对象并非都是那些特殊的 J2EE 对象——例如，我们想创建一个数据缓存以加快处理速度（请参阅第 14 项），并且这个缓存是在 servlet 和 EJB 实例之间共享的。我们可能会在被缓存的对象中创建某种计数方案，但是我们不得不为每一个我们想在缓存中保存的单一对象都这样做，而如果我们不能保证每件事物都完全地正确，那么此计数方案可能很快就会崩溃。

从 JDK1.2（Java 2）版本发布开始，Sun 意识到 Java 程序员需要能够更好地在 JVM 中与垃圾收集器交互，并且该公司也通过在 java.lang.ref 包中声明的 Reference 对象类型实现了这一点。但是它们的用法看起来有点深奥，在许多情况下，它所提供的功能类型正是我们在苦苦寻找的。

Java 提供了三种 Reference 对象：SoftReference、WeakReference 和 PhantomReference。它们都是基类 Reference 的子类，每一个都有一个基本属性，表示它们所“包装”的其它对象，我们称之为“引用物（*referent*）”。你可以通过 Reference 对象上的 get 方法来访问这个引用物（有一个例外，PhantomReference 类型，我们将简略地介绍它）。

Reference 对象降低了对某个对象的引用的“强度”。正常情况下，当我们编写一些象下面这样的代码时，在堆栈上声明的引用，在这里就是 strongRef，表明在引用另外一端的对象仍然在被使用：

```
Person strongRef = new Person();
while (true)
{
  // Do some work with the Person object here
}
return;
```

因此，根据 Java 语言规范的规定，该对象不能被垃圾收集。只有当该对象不再强可获得时，引用物，假设它在其它任何地方也没有被强引用，就会是弱可获得、软可获得，或者是幻象可获得。其结果就是垃圾收集器现在可以自由地收集该引用物了，而这有点依赖于引用对象自身的语义。

首先，这好像并没有什么太大的好处；然而，故事到这里还没有完。除了将引用物标记为符合收集条件之外，Reference 对象还提供了通告机制，称为 ReferenceQueue。当一个 ReferenceQueue 被传递到 Reference 对象的构造器中时，垃圾收集器会确保将该 Reference

对象置于这个队列中（它将对引用进行排队），并且对此感兴趣的各方都可以将该 Reference 对象从 ReferenceQueue 中被拉出，以了解该 Reference 对象的引用物是否不再被我们所使用，以及不再使用的情况下，它将在何时被收集。

审视完 Reference 对象的基本内容后，我们将依次看看每一种具体类型。

SoftReference 对象

作为一个 Java 程序员，在你的生活中的某个时刻，某人（通常是大老板）路过你的办公室，开始谈论你做了多么伟大的事情，他或她对你为其公司工作感到多么的高兴，等等诸如此类的事情。而你此时开始猜想，他或她的甜言蜜语之后会是什么：应用程序太慢了，你应该提升其速度，快点。

特别是对那些基于 servlet 的应用，说到快速的响应，在我们脑子里马上就会浮现出一个词：缓存。对一个面临性能问题的开发者来说，迅速地决定（这是基于直觉做出的，但这并不好——请参阅第 10 项）需要一个缓存来加速处理过程的情况并非不常见。特别是，程序员的直觉告诉他系统正在把过多的时间花费在访问位于其它某处的数据上，例如硬盘或数据库，而这些数据并不是经常会发生变化的。因此，为避免慢速的硬盘 I/O 操作或数据库访问而在内存中缓存数据，将会有助于系统的提速。

在做出这个结论时必须非常小心——你的应用的性能可能与访问硬盘或数据库中的数据的速度压根儿没有关系，而可能是受到了由于争用某项共享资源而造成的瓶颈的影响。在此种情况下，缓存机制一点用处也没有。只有在你已经分析过你的系统，消除了系统瓶颈，并发现应用的响应时间仍旧是令人无法接受的时候——只有这样，可能缓存机制才会是解决问题的出路。然而，这是一种深思熟虑之后的权衡：你在用服务器端系统的可扩展性换取获得结果所需的更短的响应时间。

因此你开始缓存你能够缓存的一切事物：输出结果、生成的图像、生成的对象，以及任何你可以从一个请求保持到下一个请求的事物，这样你就不必再次重新创建重复的对象。在某些

情况下，程序员甚至会缓存 String 对象，尽管事实上 String 对象经常会被 JVM 所缓存。

你测试代码并发现系统现在可以比以前更快地处理 N 个客户端。而且在第 N+1 个客户端访问系统之前，该代码工作得都非常好；也就是说，在某一时刻，由于系统增加了越来越多的客户端，所以迟早你会耗尽所有内存，并且请求将会失败。被拒绝的客户端将被强制重新尝试，直到有一个现有客户端放弃其连接，从而释放包括被缓存数据在内的资源时为止，而这些资源都是被代表该客户端的服务器端的代码所使用的。“唉,”你告诉你的老板，“是到了要去购买更多的硬件来处理那些有 N+1 个客户端的罕见情况的时候了。它只是一个花费了数百万美元的大型演示系统，这确实挺丢人的，但是请记住，我们想让那些调用比以前更快，所以我们缓存了数据。这是在响应时间和可扩展性之间的一个权衡，老板——我们对此无能为力。”

这可不是什么好事情。缓存的目的是要减少响应时间，而不是削弱可扩展性。这里所发生的问题显而易见：系统上的每一个客户端突然间都在吸收更多的资源，这导致在给定的硬件节点上，你所能支持的客户端的数量在减少。即使缓存在本质上是全局的，在所有客户端之间共享的，基于服务器的应用也很少能够在用户之间缓存数据，甚至当只有共享对象被缓存时，它们仍然将占据一定量的、客户端处理再也不能使用的内存。因此，很遗憾，这意味着缓存，以及那些根据定义你可以在必要的时刻重新创建的数据，就成为了可扩展性的绊脚石。

在很多方面，在类似上面的情形中，程序员真正想要的是缓存能够在内存不足的条件下自我清空，因为我们总是可以在必要时返回，重新创建对象， 甚至重新计算数据。这正是 SoftReference 所做的：当我们对一个对象创建了一个 SoftReference 时，该对象就成为了软可获得的，这意味着如果没有任何其它对该对象的强（正常的）引用，那么在内存少的情况下，JVM 将释放被软引用的对象，希望以此来为对象分配腾出更多的空间。当这一切发生时，SoftReference 对象将不再持有一个对其对象的有效引用，并且在被询问其引用物时将返回 null。

SoftReference 的用法实际上非常简单：对任何我们希望被软引用的对象，我们可以对其创建一个 SoftReference，并持有该 SoftReference。例如，松散地基于 java.util.Map 接口的一个通用缓存的实现如下面代码所示：

```
public class Cache
{
  private Map cachedItems = new HashMap();

  public Cache()
  { }

  public void put(Object key, Object data)
  {
    cachedItems.put(key, new SoftReference(data));
  }
  public Object get(Object key)
  {
    SoftReference sr = (SoftReference)cachedItems.get(key);
    return sr.get();
  }
}
```

请注意，缓存将对象的强引用转交给了软引用的对象；若代码向缓存请求一个对象，代码返回时，我们并不希望对象从此就突然消失。返回的强引用会令软引用的对象一直存活，直到该强引用被丢弃。一旦该强引用被丢弃，那么当内存吃紧的时候，该对象就又一次符合被收集的条件。

在 Cache 类中，即在上面的代码中，请注意，如果一个软引用的对象被收集了，那么我们可

能会想同时移除该对象所对应的键（key）。这意味着我们需要以某种方式向 JVM 注册，当一个 SoftReference 被清除出去时要接收到相关的通告；幸运的是，使用 ReferenceQueue 可以使这种行为成为可能。

当我们创建 SoftReference（或者任何 Reference 对象）时，我们可以传递进去一个 Cache 已知的 ReferenceQueue 实例。当 SoftReference 被清除时，JVM 将把该 SoftReference 实例排队到 ReferenceQueue 中，这样我们就可以从 ReferenceQueue 中通过调用 remove（该方法将引起阻塞，直到可获得一个被排队的引用或超时为止）或 poll（该方法会立即返回一个 null 或者是一个被排队的 Reference 对象）来获取它。因此我们可以修正前面所示的 Cache 的实现，让它更加了解软引用的对象何时会被回收；在这种情况下，我们将在每一个 get 或 put 调用上对被排队的引用进行调查，因此，这种方法使我们不必对设置一个单独的 Thread 去阻塞 remove 调用而感到担心。

因此，现在我们就可以缓存我们所希望的那么多的对象了，因为我们知道如果 JVM 开始运行于内存吃紧的状态，那么它将开始从缓存中回收对象，直到要么缓存为空，要么对内存的需求得到满足为止。如果这一切发生了，那么我们总是可以在内存状况比较友好时，返回去并重新组装缓存，但是要小心——除非在堆上有更多的空间或者我们开始强制地鞭策垃圾收集器执行之前，都不应该重新组装缓存。在一个产品实现中，Cache 类应该在构造器中接受一个阈值参数——如果可用的堆空间小于这个阈值，那么它就不用为怎样去持有被缓存的项而感到烦恼，它只需丢弃这些引用（因为我们假设垃圾收集器无论如何都将会在数毫秒之后对 SoftReference 进行清除）。

WeakReference 对象

WeakReference 对象，就像其名称所表示的，将它们的引用物变成了弱可获得，这在本质上意味着其引用物在任何时刻都是符合垃圾收集条件的，其前提是这些引用物没有通过某个其它的引用而成为强可获得的。就像使用 PhantomReference 对象（下面将会讨论）一样，WeakReference 的威力更多的是体现在我们可以在垃圾收集器想要收集引用物时得到通告，而不是我们允许引用物被回收。为了理解这为什么很有用，我们必须先回到前面再谈谈对象

池。

让我们回忆一下第 72 项，如果一项资源是有限的，并且其创建开销是昂贵的，我们就会想对其创建一个对象池，以此来缓解分配和清除的开销，并且可以了解已经创建了多少对象。然而，这暗示着，你不仅要知道应该何时创建一个我们正在讨论的资源对象的实例，还要知道何时被取用的资源对象不再被使用。在传统的对象池实现中，这项职责被丢给了程序员，他要去调用在池上的某种返回或清除方法，并为重用将实例返回给池。

遗憾的是，你我都知道这种策略要求程序员的警惕性很高，并且要遵守规则，然而残酷的事实是：在紧迫的最终完工期限和不可能实现的需求规范的压力之下，警惕性和纪律性会被开发团队首先牺牲掉。这意味着我们确确实实是在冒险：对象可能不会被返回到池中，现在我们退一步，依赖于池中对象的终结器能够被触发，而这也是我们能够知道客户端已经完成了对缓存对象的处理的唯一手段。

幸运的是，有一个好方法，可能你现在已经猜到了，就是使用 WeakReference。因为 WeakReference 不会使对象保持存活状态，所以我们可以通过将强引用分发给池中缓存的资源，来构建一个对象池。当客户端完成操作时，它直接丢掉强引用，这使得该对象就成为了弱可获得的，在下一次垃圾收集过程中，该对象会被收集，并且（通过捕获 ReferenceQueue 通告）该资源会被返回到对象池中。

因而，这样做的关键之处就在于分发可以被自由回收的对象，并进而通告对象池；经典的实现这项工作的方式是让对象池自己保存池中对象的有限集合，并且不是分发对这些对象的引用，而是分发对代理的引用。当客户端丢弃掉对代理的引用时（该代理是被我们的池弱引用的），对该代理的 WeakReference 就会被排队。我们之所以能够探测到这种丢弃，要归功于在池中持有 ReferenceQueue，这样就可以将代理另一端的资源对象返回到池中。

因为客户端总是与代理（而不是我们的 WeakReference 自身）进行交互，所以对于由池管理的资源对象以及要实现的代理而言，这意味着我们需要一个公共接口（请参阅第 1 项）。

请注意，一个对象池还可以使用 PhantomReference 对象来实现，并且其实现代码（由

JavaWorld 的专栏作家 Vlad Roubtsov 慷慨地捐献）在本书的 Web 站点上可以看到。实际地讲，从客户端的角度来看，PhantomReference 对象与 WeakReference 对象没有任何功能性的差异；唯一真正的变化就是，PhantomReference 对象与 WeakReference 对象是在对象生命周期的不同时刻收到信号的。

PhantomReference 对象

根据 PhantomReference 对象的文档所述，“PhantomReference 绝大多数情况下被用来以比 Java 的终结机制更为灵活的方式调度亡前（*pre-mortem*） 清除动作。”公正地讲，在阅读过前面有关终结器的段落并意识到它是多么糟糕的一个概念之后，你可能会认为任何能够使对象清除变得更为容易的事物看起来都会是一个好的想法。

（我在这里真的应该做到公正，并应该指出终结机制并非那么糟糕，但是事实上它确实是完全非确定的、无序的，以及不可靠的。这可不是 Java 自身能够纠正的问题，任何自动回收系统都必须妥善处理它。对于那些在家自学的人来说，.NET 也有完全相同的问题。）

问题是，PhantomReference 对象最初看来并非有用；它在三种引用中显得很独特，因为它并没有实际地持有一个对其引用物的引用，这意味着在一个 PhantomReference 上调用 get 时，它将返回 null。或者，更具体一点，在 PhantomReference 被排队之前调用 get 将会返回 null——所有三种 Reference 类型在 Reference 被排队/发送通告信号之后，都将返回 null。文档中指出我们可以创建 PhantomReference 的子类，但是如果我们将一个我们想在其上调用 close 方法的对象的引用置于 PhantomReference 子类中，那么该引用就会成为一个强引用，PhantomReference 自身就永远都不会被排队。

PhantomReference 对象的使用更为精妙。下面是一个例子，展示了在对象不再强可获得时，我们怎样使用 PhantomReference 来执行清除。

```
// Example showing use of PhantomReference
//
```

```
import java.lang.ref.*;
import java.util.*;



class CleanThisUp
{
  // The resource we need to finalize; for simplicity's sake,
  // I'm not actually going to show the connection, but it's
  // pretty easy to see how this would work in practice
  //
  private java.sql.Connection conn =
    getConnectionFromSomeplace();



  public CleanThisUp()
  {
    System.out.println("CleanThisUp created: " + hashCode());

    // Create our PhantomReference to do the cleanup and
    // register it so the PhantomReference itself doesn't get
    // lost
    refList.add(new CTUPhantomRef(this, conn, cleanupQueue));
```

```
}



private static class CTUPhantomRef extends PhantomReference
{
  private java.sql.Connection connToClose;

  public CTUPhantomRef(CleanThisUp referent,
                       java.sql.Connection conn,
                       ReferenceQueue q)
  {
    super(referent, q);

    this.connToClose = conn;
  }
  public void clear()
  {
    try
    {
      super.clear();
    }
    finally
```

```
    {
      // Now do our own cleanup
      //
      try
      {
        if (connToClose != null)
          connToClose.close();
        System.out.println("I cleaned up a connection!");
      }
      catch (java.sql.SQLException sqlEx)
      {
        // Log this, ignore it, whatever—it's never exactly
        // clear what should be done in the event of an
        // exception on a close() call. Regardless, don't
        // just ignore it.
        //
        sqlEx.printStackTrace();
      }
    }
  }
}
```

```
private static ReferenceQueue cleanupQueue =
  new ReferenceQueue();
private static List refList =
  Collections.synchronizedList(new ArrayList());
private static Thread cleanupThread;
static
{
  cleanupThread = new Thread(new Runnable()
    {
      public void run()
      {
        try
        {
          Reference ref = null;
          while (true)
          {
            ref = cleanupQueue.remove();
            refList.remove(ref);
            ref.clear();
          }
        }
```

```
        catch (InterruptedException intEx)
        {
          return;
        }
      }
    });
   cleanupThread.setDaemon(true);
   cleanupThread.start();
  }
}



public class PhRefTest
{
  public static void main (String args[])
  {
   for (int i=0; i<100; i++)
   {
     CleanThisUp[] ctuArray = new CleanThisUp[10];
     for (int j=0; j<10; j++)
       ctuArray[j] = new CleanThisUp();
     ctuArray = null;
```

```
        }
      }
    }
```

这里发生了很多事。首先，我们有这样一个类，它请求某种类型的清除——因为此时，它持有一个数据库连接，而我们想确保能够以及时的方式关闭该连接，理想情况是在对象自身一被释放就关闭它。尽管垃圾收集器不能确保一旦最后一个对 CleanThisUp 实例的强引用被丢弃，它就能够有所反应，但是我们可以让垃圾收集器在它即将要使用 PhantomReference 清除该对象之前的一刻，通知我们。因此，在 CleanThisUp 的构造器中，我们用 CleanThisUp 类持有的静态成员 ReferenceQueue 创建了一个 PhantomReference 实例（实际上是 PhantomReference 的一个私有派生类型）。

然而，请记住，PhantomReference自身不能持有对CleanThisUp的引用，这就是为什么CTUPhantomRef类被声明为在CleanThisUp中的一个嵌套类的原因[5]——除非被声明成为静态的，否则，一个嵌套类实例将持有一个对其外围类实例的引用（按照Java的说法，就是外部类）。这足以使CleanThisUp实例保持强可获得，这意味着它永远都不会被垃圾收集器拉去排队，而我们所谓的比终结器更好的清除方案也将会惨淡地失败。

请注意，我们还一直在跟踪着在 CleanThisUp 静态域的 ArrayList 中持有的 CTUPhantomRef 实例（顺便说一句，该 ArrayList 必须被同步，因为它将会被多个线程轮番轰击）。我们需要让 PhantomReference 自身保持存活状态，但是指向 PhantomReference 的强引用对引用物（CleanThisUp 实例）的可获得性来说，没有任何影响。

下面是 PhRefTest 类，即本例中的驱动，我们循环 100 次，去创建一个由 10 个 CleanThisUp 实例构成的数组，然后释放对该数组的引用（因此也就释放了链接到该数组内部实例上的强引用）。这意味着那些实例只是虚幻地可获得的；请记住，我们仍旧持有 CTUPhantomRef 实例，因为在 CleanThisUp 中的 ArrayList 是静态的。此时，垃圾收集器会被鼓励去执行一

[5] 我们想对CleanThisUp的客户端屏蔽这种实现细节，因此我们使它成为嵌套类，并将其标记为私有的以防止对它的探查。

次完整的收集（尽管事实上强制垃圾收集器去这么做通常是个糟糕的想法——请参阅第 72 项），然后我们执行循环中的下一次迭代。

当垃圾收集器决定要收集我们遗弃的，并且是虚幻可获得的 CleanThisUp 实例时，它首先将 CTUPhantomRef 实例排队到我们构建它时所传递给它的 ReferenceQueue 中。对于垃圾收集器，那就是它所需要做的一切工作，但是我们有一个无限循环的守护线程，它阻塞在那里，直到在 ReferenceQueue 中有了一个可获得的引用时为止。我们将 Reference 从队列中拉出，从 ArrayList 中移除该引用，然后在其上调用 clear 方法。（我们必须这样么，因为如果不这么做，PhantomReference 实例将永远不会被清除，因此也就永远不会实际去收集它们的引用物。它是 PhantomReference 类的那些怪癖之一。）然后，通过重载 CTUPhantomRef 的 clear 方法，我们就可以在 CleanThisUp 实例被释放回可用内存的池中之前，执行我们的清除。

然而，我们仍必需在此仔细一些——如果一个 CTUPhantomRef 将 CleanThisUp 作为引用物持有着，我们就不能持有该 CleanThisUp 实例的引用，因为那将会使得引用物保持强可获得，然后我们就会再次回到“引用不会被排队”的状态。因此，为了清除的目的，我们将 CleanThisUp 资源自身传递给 CTUPhantomRef；在这种情况下，我们赋予 CTUPhantomRef 一个 CleanThisUp 所持有的 JDBC Connection 实例的副本，所以我们可以在 CTUPhantomRef 之上的被重载的 clear 调用中关闭它。

毫无疑问，如果你运行上面的程序，你将会看到大量的 CleanThisUp 实例被创建，然后在一段时间之后，对 CTUPhantomRef 的 clear 方法的调用开始混杂在屏幕显示中。千万要注意观察——如果你正在仔细地观察，那么你很快就会注意到并非所有被创建的对象都被清除了。这是因为我们的 Thread 是一个守护线程，所以即使这些对象事实上确实被排队到了 ReferenceQueue 中，由于主线程已经退出，我们的守护线程不足以让 JVM 仍旧保持存活，因此，我们在关闭的时候某些实例仍旧处于等待被清除的状态。如果你绝对地、肯定地需要那些对象被清除掉，那么就应该将 Thread 标记为非守护线程，并且在关闭 JVM 的时候，指出销毁它的方式。

这个样例的关键缺陷是 Thread 是在 CleanThisUp 类内部创建出来的；在一个 J2EE 环境中，任意地创建线程并非总是一件易事（请参阅第 73 项以了解为什么容器想要承担线程管理的

职责）。其它能够想到的可能性有：在 CleanThisUp 上创建一个静态方法，它获取一个 Thread 引用，供监听 ReferenceQueue 的 Runnable 使用，或者从一个静态方法中返回该 Runnable，将其交给容器去在一个 Thread 上执行。可供选择的另一种方法是，你可以周期性地截获客户端的线程，并在 ReferenceQueue 上使用 poll 方法来查看是否有任何要监听的 Reference 实例，但是这会使 CleanThisUp 类的复杂度变得更大。

然而，最终我们还是拥有了一种比使用终结器的形式要更好的执行清除的方式；当然，这需要作更多的工作，但是任何有意义的事情都并非轻而易举就可以实现的。

请仔细观察我们在上面三个例子中所做的事情：在每一种情况下，我们都向垃圾收集器提供了它在确定的情形和场景下应该采取何种行为的线索。在使用 PhantomReference 对象的例子中，在对象准备好被回收之后，我们请求某种发生在亡后的清除。因为实际的清除发生在我们自己的线程中，而不是终结器的线程中，所以我们可以对清除和线程死锁做出有意识的决策，而终结器线程由于它被深入编码到了 JVM 内部，所以并不能这么做。在使用 WeakReference 对象的例子中，我们让垃圾收集器在将某个对象的最后一个引用丢弃时发送一个通告，这样就可以赋予我们对 WeakReference 引用的对象执行某种回收的能力。在使用 SoftReference 对象的例子中，对于垃圾收集器来说，它本身就是一个信号，说明位于引用另一端的对象在内存吃紧的情况下不值得保持。在每一种情况下，我们都向垃圾收集器提供了比我们在早期版本的 Java 中所能提供的更多的信息；请恰当地使用它们。

第 75 项：不要担心在服务器上的 JNI 代码

经历多年以后，Java 开发者对 JVM 之外的代码已经有了几分喜爱和憎恨。很常见的是，我们需要得到这样的代码，但是从 JVM 中这么做，在最好情况下，会成为一种艺术形式，而在最坏情况下，会成为导致进程崩溃的快速途径。

当然，问题是访问 Java 环境范围之外的任何事物都需要使用 JNI，这通常意味着我们将退回到指针、不受管理的代码和 C/C++。尽管大多数 Java 程序员确实没有任何个人的原因去反对使用 C/C++编译器，但是我们仍旧有一个理由去使用 Java 编写代码：自动内存管理，它

是一个虚拟机，如果我们偶然地废弃了一个空指针，它几乎可以消除野指针（*wild pointer*）和进程崩溃等等诸如此类的事情。

然而，在 Java 程序员的生活中存在着一个令人头痛的事实，即无论 Sun 作出了多少努力，Java 环境并没有覆盖所有事物。我们时不时地需要从 Java 中通过一个基于 C/C++的 API 来访问某些事物，这使我们直接退回到了 JNI。当发生这种情况时，先做深呼吸，鼓足勇气，然后一头扎下去。

实际情况是，JNI 其实不像看起来那样糟糕；事实上，假设你已经基本掌握了 C/C++，关于 JNI 的困难部分就不是编写（以及调试）代码，而是努力去解决为什么 JVM 不从你的 J2EE 环境中加载你的共享类库，这通常会让 Java 开发者发疯的。

然而，还是先看看重要的事情。当我们面对编写 JNI 代码的任务时，基本上有两种方式来处理它：困难的、低层的方式，以及容易的、高层的方式。困难的、低层的方式是编写 JNI 规范所定义的 JNI 代码。编写你的拥有用 native 关键词定义的方法的 Java 类，使用 javah 工具生成 C 头文件的桩（然后将它剪切并粘贴下来，作为 C/C++实现的起点），然后编写一个包含了 native 方法实现的共享类库（Win32 底下的.dll，大多数 UNIX 底下的.so 或类似的结构），这些 native 方法使用 JNIEnv 结构获得函数指针，它允许在必要时回调进入 JVM。想分配一个 Java 字符串作为你的 native 方法的一部分？那么就回调进入 JVM，取得 C 字符串并将它转换成为一个 Java 字符串。想将传入的 Java 字符串转换为一个 C 风格的以 null 结尾的字符数组？那么就回调进入 JVM 之中去实现吧。想分配一个 Java 对象？你可能已经猜到了——还是通过 JNIEvn 回调进入 JVM 之中。哦，不要忘了每一步都要检查 Java 异常，以防 JVM 产生问题（例如一个 OutOfMemoryError 或一个 ClassNotFoundException）。做这些事很乏味，而且任何程序员发觉很乏味的事情很快就会出错。

然而，有一种更好的方式，它有各种各样的风格：让其他的某些人来为你进行低层的编码。互联网上到处存在着大量的工具包，无论是商业的还是开源的，都可以使你从JNI编码的繁重任务中解脱出来。一些开源的可供选择的工具包包括：*Sheng Liang*在他的《Java本地接口（*The Java Native Interface*）》[Liang]一书中提到的“共享桩”的代码，*Stu Hallway*在他的《Java平台的构件开发（*Component Development for the Java Platform*）》[Halloway]一书中提到的

Jawin（Java-Windows）类库，以及JACE，它是一个开源项目，提供了Java对象的C++包装器，以使得在本地代码中可以很容易地调用那些Java对象的方法。应该在任何可能的时候都充分利用这些工具包——它们大多数都具有灵活的许可证发放方案，甚至可以令最吝啬的经理和大多数顽固的律师感到满意，并且如果开源社区不能满足你的需求，还有大量的公司提供了商业化的可供选择的工具包。

无论你采用哪种方式编写你的 JNI 代码，现在最困难的部分出现了：到底应该把你的共享类库置于何处，以使得 JVM 甚至是在 J2EE 容器内就可以获得它？问题的答案分为两部分。首先，JVM 以操作系统通用的方式来查找共享类库，这意味着它使用的是底层操作系统的动态加载策略；例如，在 Win32 的运行环境中，它将使用 Win32 API 中的 LoadLibrary 来了解 Win32 加载器将在哪里查询 DLL 的细节信息（包括 PATH、WINNT 目录、WINNT\SYSTEM32 目录，以及当前目录）。然而，JVM 还可以用一个其它的、“可移植的”位置来扩充这个位置集合：在 JRE 的 lib 目录之中有一个 CPU 相关的目录，在该目录中可以防止共享类库。例如，在 Win32 的 JVM 中，lib 目录包含了一个 i386 目录。正常情况下，在该目录下唯一的一个文件是一个配置文件（请参阅第 68 项以了解其细节），但是如果你在此放置了一个本地库 DLL，那么在使用 System.loadLibrary 调用时，它会自动成为 JVM 查找本地类库时所使用路径的一部分——这也是使用独立的 JRE 实例的另一个原因（请参阅第 69 项）。

事实上，在 JVM 要搜索的路径中，有一部分是由一个的 JVM 系统属性所控制的，这个属性是 java.library.path，这部分路径包含由底层操作系统指定的那部分路径。可以在 Java 启动命令行中通过标准的-D 选项来改变这个属性，以替换 JVM 的缺省值（例如，JRE 的 bin 目录、当前目录，以及在 Win32 运行环境中的 PATH 值）。

然而，指出 J2EE 容器支持构件的热部署是相当重要的，这意味着我们无需关闭服务器，就可以插入、移除和升级部署在一个 J2EE 容器中的构件。如果容器已经在处理来自客户端的请求了，那么只要它们是由不同的类加载器实例加载的，就意味着该构件的两个版本可能会同时存在于 JVM 中。遗憾的是，类加载器提供的这种隔离性并没有扩展到本地库，并且大多数操作系统对一个动态库只会加载一次。加载相同的类库的请求实际上会是一个空操作，因为通常只会根据类库名而区分不同的类库。

对于 JNI 程序员来说，这意味着一旦某个本地类库被加载了，那么我们几乎就要忙一整天了——让一个 JVM 去卸载一个本地类库是一项让人感到极度受挫的任务。JNI 规范描述到：一个本地类库只有在与它所提供实现的类相关的类加载器被卸载时，才会被卸载，但是在 JVM 中试图去强制卸载一个类加载器实际上是办不到的。

因此，如果你在编写需要被热部署到某个 J2EE 容器（例如一个 servlet 容器）中的本地类库，那么请确保每一个类库都有一个基于版本号的、可区分的名字；这可以糊弄过大多数的操作系统，让它们相信这些是彼此独立的类库，从而在新版本的构件被热部署到容器中时允许加载新的类库。当然，其缺点是如果构件被热部署多次，那么很可能相同的本地类库会有多个副本被加载到了进程中，这将使 JVM 进程所占的内存大出许多。

很明显，Java“本地化”并不是最理想的情况。你的代码将因此失去了一定的可移植性（如果可移植性对你很重要的话；请参阅第 11 项）；削弱了 JVM 的稳定性，因为不受管理的代码（也就是说非 Java 代码）可能会意外地破坏部分进程的可能性；削弱了 Java 代码执行的安全环境，因为不受管理的代码没有 SecurityManager 和 Permissions 模型的保护，凡此种种。但是对于那些你绝对、肯定要超出 JVM 范围的场景来说，JNI（以及各色俱全的高层工具包）将赋予你可以这样做的能力。

参考书目

——如果我看得远些，那是因为我站在巨人的肩上而已。
——伊萨克 牛顿

书库

[Alur/Crupi/Malks] *Core J2EE Patterns*（第二版），Deepak Alur,，John Crupi， Dan Malks. Boston, MA： Addison-Wesley著 ；2003。它是一本很好的关于J2EE平台的术语集。第二版本校订了反射的惯用术语，以及其他一些事物。第一版于 2001 年出版发行，包含一些第二版未涵盖的一些模式。

[AJP] *Applied Java Patterns*，Stephen Stelting ， Olav Maassen，Palo Alto，CA： Sun Microsystems Press，2002。

[Bernstein/Newcomer] *Principles of Transaction Processing* ， Philip A，Bernstein ， Eric Newcomer.， San Francisco： Morgan Kaufman著； 1997。第一部用于理解什么是事务处理、为什么它对企业级系统如此重要的阅读书籍，虽然它的一些技术实例和描述有点过时，但其中的概念是永恒的。

[Bloch] *Effective Java* ，Joshua Bloch. Boston， MA:：Addison-Wesley，2002。Nuff说这本书是每个Java开发者的必读之作。

[Box]*Essential COM*， Don Box. Reading ，MA: Addison-Wesley著；1997。该书的第一章是技术著作中最佳的一章，很好地解释了为什么开发者更倾向于面向构件的设计。第 1 项的灵感来源于此。

[Brown] *AntiPatterns*， William J. Brown. New York:：Wiley著；1999。针对产生某些结果环境中的特定问题，一种模式描述一种解决方案；而反对者则尽力叙说怎样从负面情形中恢复。

[Brown] *Programming Windows Security*，Keith Brown. Boston,，MA: Addison-Wesley著；2000。理解Windows操作系统中的安全因素远比大多数Java开发者认为的那样更为重要，尤其是当运行的Java代码是作为Windows后台服务的时候。

[Celko97] *SQL Puzzles and Answers,*， Joe Celko.。San Francisco： Morgan Kaufman著；1997。知道SQL吧？请选择适合你的。

[Celko99] *Data and Databases*:：*Concepts in Practice*，Joe Celko。San Francisco:：Morgan Kaufman,著；1999。Joe Celko是迄今为止对关系技术最具洞察力的作者之一。这部著作中，他清晰而准确地解释了关系模型。

[Celko00] *SQL for Smarties: Advanced SQL Programming* (第二版)，Joe Celko. San Francisco:：Morgan Kaufmann,著，2000。阅读完*Data and Databases：Concepts in Practice* [Celko99]之后，阅读本书达到更高层次。

[Cooper99] *The Inmates Are Running the Asylum*，Alan Cooper. Indianapolis著， IN： SAMS 出版社，1999。一篇关于为什么程序员设计的用户接口往往仍不被世界上的其他人所理解的优秀文章。

[Cooper03] *About Face 2.0*，Alan Cooper. New York: Wiley,著；2003。关于构建基于最终用户的友好用户接口颠覆之作的第二版。

Date] *Introduction to Database Systems* (第 8 工程研制.)， C. J. Date. Boston, MA: Addison-Wesley著； 2004。是迄今为止对所编著的关系型数据库探讨的最好——也是最深刻的一本书。非常概念化，对关系型数据库系统原理的探讨比任何其他一种SQL语言或实现都详细得多。

[EAI] *Enterprise Integration Patterns*， Gregor Hohpe， Bobby Woolf. Boston， MA: Addison-Wesley著；2004。收集了关于消息系统的几十种模式，在企业级应用整合环境中。不要在意它的标题，阅读过它，你会对消息以及它的灵活性和效能有一个深入的理解。

[Ewald] *Transactional COM+*， Tim Ewald. Boston，MA:：Addison-Wesley,著；2001。尽管它着重于COM+，但仍对现流行的面向对象环境中的事务处理进行详尽讨论。他探讨的第 1 章都是精华，对本书第 5 项有所启发。不要在意代码实例，而是着重概念理解。

[Falkner/Jones] *Servlets and JavaServer Pages*，Jayson Falkner，Kevin Jones. Boston，MA:：Addison-Wesley著； 2004。极好地论述了Java程序员无一例外要用到的Servlet和JSP。

[Ferguson03] *Practical Cryptography*，Niels Ferguson ，Bruce Schneier. New York：Wiley 著；2003。[Schneier95] 的提取精炼版本，更紧凑也更易吸收。任何期望对密码有更多了解人员的必读之作（不过，密码并不是这里所说的“安全”的所有内容）。

[Fowler] *Patterns of Enterprise Application Architecture*，Martin Fowler. Boston，MA:：Addison-Wesley 著; 2002。该书对企业应用开发和其间用到的技术不可知模式进行细目分类。注意 Fowler 避开了 EJB 和其他“重量级”J2EE 技术，因此阅读本书是为了理解概念，而不是 J2EE 指南 (或者 .NET)。

[Friedman-Hill] JESS in Action， Ernest Friedman-Hill. Greenwich，CT： Manning Press，2003。是一本关于 JESS（Java Expert System Shell）以及一般基于规则系统的完美入门和参考书籍。

[GOF] *Design Patterns*， Erich Gamma,，Richard Helm， Ralph Johnson， John Vlissides. Reading， MA：Addison-Wesley 著；1995。模式书籍的黄金标准——每个使用面向对象语言进行设计软件的开发者必读初级读本。

[Gong] *Inside Java 2 Platform Security*，Li Gong， Gary Ellison， Mary Dageforde. Boston，MA：Addison-Wesley 著；2003。第二版甚至比第一版中的基线平台安全模型、GSSAPI Kerberos 、第 64、65 和 66 项资源的整合论述都好。

[Gray/Reuter] *Transaction Processing*，Jim Gray，Andreas Reuter. San Francisco：Morgan Kaufman，1993。如果你仍在寻找事务处理的禅宗，那么在阅读完[Bernstein/Newcomer]后阅读此书。如果没有什么别的好处，对回顾历史也是大有裨益的。

[Gulutzan/Pelzer] *SQL Performance Tuning*， Peter Gulutzan ，Trudy Pelzer. Boston， MA: Addison-Wesley 著，2003。迄今为止，调整 SQL 仍是企业级开发者创建高效系统的重要部分。不过关键是，这本书指出当前流行的 8 种数据库中每种调整方法的相对优点，论证了对于多种调整选择，一个是不能涵盖全部的道理。

[Halloway] *Component Development for the Java™ Platform* ， Stuart Dabbs Halloway. Boston，MA：Addison-Wesley 著； 2001。对类装载器、序列化、Java 本地始接口以及一般的构件设备进行了令人敬畏的介绍。

[Henderson] *The Guru's Guide to SQL Server Stored Procedures, XML, and HTML*，Ken Henderson. Boston，MA： Addison-Wesley 著； 2005。

[Howard/LeBlanc] *Writing Secure Code (*第二版*)*，Michael Howard ， David C. LeBlanc. Redmond，WA：Microsoft Press， 2003。尽管它本质上是以 Window 为中心，这是一座关于代码中所关心内容的金矿。例如，在 430 页使用正则表达式来避免跨站点的脚本攻击。

[Jones/Lims] *Garbage Collection*，Richard Jones ， Rafael Lims. New York:：Wiley 著；1996。是一本关于垃圾收集怎样运转的规范参考书。

[Kaye] *Loosely Coupled*：*The Missing Pieces of Web Services*， Doug Kaye. Kentfield， CA：RDS Press，2003。

[Kreger/Harold/Williamson] *Java and JMX*，Heather Kreger，Ward Harold， Leigh Williamson. Boston，MA：Addison-Wesley 著；2003。对 Java 管理扩展 API 和标准进行精彩而全面的讲解。

[Lea] *Concurrent Programming in Java* (第二版)， Douglas Lea. Boston， MA：Addison-Wesley, 著；2002。对 Java 平台的并发和同步进行高效讲解。如果你不清楚关键字“synchronized”在 Java 中意味着什么，那么强力推荐这本书。

[Liang] *The Java Native Interface*， Sheng Liang. Boston， MA：Addison-Wesley 著；1999。有点陈旧（在 JDK1.1 期间发布的），Sheng 的书仍是 JNI 的典型代表。它还包含共享存根类库（可从网上获得），是一个更简单的本地代码引发机制。

[Marinescu] *EJB Patterns*， Floyd Marinescu. New York:：Wiley 著，2001。EJB 术语的早期汇集。

[Meyers95] *Effective C++*，Scott Meyers. Reading，MA： Addison-Wesley 著；1995。 尽管它着重于 *C++*，但仍具有很多想法和概念资料。也是本书的灵感来源。

[Meyers97] *More Effective C++*，Scott Meyers. Reading， MA：Addison-Wesley 著；1997。紧紧跟随高水平的语言元素，例如一个类中的懒惰计算法和热情计算 。

[Mitnick] *The Art of Deception*， Kevin Mitnick. New York： Wiley 著； 2003。如果你或者同事具有这样的思想：安全比代码更重要，那么请阅读本书。世界上最安全的密码协议不能预防一个简单的骗子，Mitnick 向你介绍最无害的交谈是怎样揭示关键性的攻击隐秘。

[Neward] *Server-Based Java Programming*，Ted Neward. Greenwich，CT：Manning Press，2000。一本致力于介绍应用服务是怎样运转的企业级 Java 开发的早期版本。有点过时，不过各个部分仍相关。

[Nock] *Data Access Patterns*， Clifton Nock. Boston，MA： Addison-Wesley 著；2004。 可能是一本关于普通关系数据访问模式的最好书籍（例如，不局限于任何特定技术）。Java 中的例子仅仅是一个额外收获。第 33 和 34 项的灵感来自于此。

[Oaks] Java Security (2nd ed.)， Scott Oaks. Sebastopol，CA： O'Reilly， 2001。阐明 Java 平台安全模式对创建安全的 Java 系统极为关键；阅读本书是理解 Java 安全模式的第一步。.

[PLOPD2] *Pattern Languages of Program Design 2*， John Vlissides， James O. Coplien，Norman L. Kerth. Reading， MA：Addison-Wesley 编辑；1997。从同一名称的模式会议收集而来，汇集了一些对企业级系统很有用的模式。

[PLOPD3] *Pattern Languages of Program Design 3*，Robert Martin， Dirk Riehle，Frank Buschmann. Reading， MA： Addison-Wesley 编辑，1998。 像前面那个，很好汇集了各种系统中的模式，大多还是企业级系统，比起先前那一卷来说，更多的是以代码为中心。

[PLOPD4] *Pattern Languages of Program Design 4*， Neil Harrison， Brian Foote， Hans Rohnert. Boston，MA： Addison-Wesley 编辑； 2000。像这个系列的早期卷目那样，很好地收集了对企业级架构和开发很有用的模式，非常地以代码为中心。

[POSA1] *Pattern-Oriented Software Architecture* (卷 1)， Frank Buschmann，Regina Meunier，Hans Rohnert， Peter Sommerlad，Michael Stal. New York： Wiley， 1995。 最初的模式著作的一种，恰在[GOF]之前载入。 注意它的模式划分成不同层：架构，模式及术语。（注意：当第一次载入的时候，不是将它标识成卷 1，因为卷 2 直到 8 年后才载入。Wiley 在 2002 年将下面的卷 2 和此著作再版）。

[POSA2] *Patterns of Software Architecture* (卷 2)， Douglas Schmidt， Michael Stal， Hans Rohnert， Frank Buschmann. New York： Wiley 编辑， 2002。紧紧跟随卷 1，涵盖处理并发和通信以及网络方面的模式 。

[Raymond] *The Art of UNIX Programming*，Eric S. Raymond. Boston，MA： Addison-Wesley 著；2004。 像理解 Windows 平台对理解怎样创建最好的系统，然后在上面运行很重要那样，

理解 UNIX 平台也同等重要。不过这不是一本介绍 UNIX 内部源代码的书籍，它涉及一些更为重要的方面——UNIX 本身的哲学。

[Schneier95] *Applied Cryptography* (2nd ed.)， Bruce Schneier. New York: Wiley 著； 1995。关于密码算法和安全数据交换的最新命令。我建议在打开这本书之前，阅读初级读本的[Schneier03]。

[Schneier01] *Secrets and Lies*，Bruce Schneier. New York: Wiley 著；2001。这是一本关于系统安全、以及为什么从一开始就要重视它这些方面的书，是每个开发者、项目领导、架构师和经理的初级读本。第 7 章安全方面的大多内容都来自于此。

[Stevens] *UNIX Network Programming* (3rd ed., vols. 1, 2, and 3)， W. Richard Stevens. Boston，MA： Addison-Wesley 著；2004 以及即将出版的。Stevens 的书籍很久都是 TCP/IP 通信的定论，并且他的书籍正为流行开发而修订，尽管 Stevens 已经去世。

[Szyperski] *Component Software* (2nd ed.)，Clemens Szyperski. Boston， MA： Addison-Wesley, 著；2003。 面向构件设计的神圣教规。如果第 1 项仍旧不能使你明白，这本书正是为你而准备。

[Tanenbaum] *Distributed Systems: Principles and Paradigms*， Andrew S. Tanenbaum ，Maarten van Steen. Upper Saddle River，NJ：Prentice Hall/Pearson Education，2002。对一般分布式系统进行很好的概念详述。

[Tate] *Bitter Java*，Bruce Tate. Greenwich，CT： Manning Press，2002。反对 Java 模式和 J2EE。无疑是本书的很好伙伴。

[Tate/Clark/Lee/Lindskey] *Bitter EJB*， Bruce Tate，Mike Clark， Bob Lee，Patrick Lindskey. Greenwich，CT： Manning Press，2003。是一本追随[Tate]的 关于 EJB 的书；本书的另外一个好伙伴。

[Van Duyne/Landay/Hong] *The Design of Sites*， Douglas K. Van Duyne，James Landay， Jason I. Hong. Boston，MA： Addison-Wesley 著； 2002。一个基于 HTML 用户接口的构建比学科更具艺术性；这本书对可访问的网站设计进行了最佳讲解。

网站

[http://www.alphaworks.ibm.com/tech/hyperj] IBM's Hyper/J语言及框架的主页。

[http://aspectj.eclipse.org] AspectJ AOP编程语言的主页。

[http://aspectwerkz.codehaus.org/] AspectWerkz —— 一种AOP框架——的主页。

[http://www.enterpriseintegrationpatterns.com] [EAI]工厂和开发的网站。并不是所有的模式在书中都有所讲述，并且在它的wiki上积极讨论。

[http://nanning.codehaus.org] Nanning —— 一种AOP 框架的主页。

[http://www.neward.net/ted] 作者的网站。

[http://www.neward.net/ted/EEJ] 本书的附录网站；由于错误数据和更新正在调整中。

[http://www.owasp.org]开放的Web应用安全项目—— 一个安全工具、文档和讨论的开放源码收集的主页。

[http://www.theserverside.com] 卓越的J2EE社区和Internet上的讨论入口。

“通过这本书，Ted Neward 将帮助你实现从一个优秀的 Java 企业开发者向一个伟大的开发者的飞跃！”

—— John Croupi, Sun 卓著的工程师，Core J2EE Patterns（J2EE 核心模式）的合著者之一

高效企业 Java

如果你想构建更好的 Java 企业应用，并且运行起来更加高效，那么你不用再期待了。在本书中，你就可以找到一种易于理解的有关 Java 2 平台企业版（Java 2 Platform, Enterprise Edition, J2EE）开发微妙之处的指南。你将学会如何：

- 使用进程内或本地存储以避开网络，请参阅第 44 项
- 考虑使用较低的隔离级别以获得更大的事务吞吐量，请参阅第 35 项
- 为了开放集成而考虑使用 Web 服务，请参阅第 22 项
- 仔细考虑你的查找，请参阅第 16 项
- 预生成内容以最小化处理过程，请参阅第 55 项
- 使用基于角色的授权，请参阅第 63 项
- 面对故障时要健壮，请参阅第 7 项
- 为版本并存使用独立的 JRE，请参阅第 69 项

Ted Neward 向你提供了 75 项易于理解的提示，它们可以帮助你在系统和架构层次上驾驭 J2EE 开发。他对 J2EE 开发的优势、缺陷和弊端的全景式的看法将传递出你最迫切关心的问题：学习如何设计你的企业系统使其适应未来的需求；在无损于代码正确性的前提下提高你的代码的效率；发现如何实现语言或平台无法直接支持的复杂功能。在阅读过《高效企业 Java》之后，你将了解如何设计和实现更好的、更具可扩展性的、企业规模的 Java 软件系统。

Ted Newward 是一名软件架构师、顾问、作者和演讲家，他为诸如 Intuit and Pacific Bell, 和 UC Davis 这样的工作做咨询工作。他是《基于服务器的 Java 编程（Server-Based Java Programming）（Manning, 2000）》一书的作者，以及《C#速学（C# in a Nutshell）（O’Reilly,2002）》

和《SSCLI 精髓（SSCLI Essentials）（O’Reilly,2003）》的合著者。Ted 曾经是 JSR175 专家组的成员之一。他现在频繁地在巡回会议上对来自全世界的用户群演讲。他仍然在不断地开发和教授 Java 以及.NET 的课程。