

“真希望在我刚开始进行企业Java开发时这本书就出现了。这本书告诉了我们使用J2EE的好处和缺陷，以及我们可以在多大程度上避免这些缺陷。”

——Andrew J. Smith, Java设计师

“Rod所拥有的经验的深度和广度令人印象深刻！通过阅读这本书，J2EE开发人员可以避免Rod所经历的惨痛教训。”

——Todd Laxinger, 软件结构设计师, Best Buy, Inc.



J2EE

设计开发编程指南

(美) Rod Johnson 著

程海萍 子晓菲 毛选 等译



Wrox
PROGRAMMER TO
PROGRAMMER™

电子工业出版社
<http://www.phei.com.cn>



J2EE设计开发编程指南

在实践中采用J2EE的结果通常是令人失望的——应用不但慢而且十分复杂，还要花很长时间去开发。问题不在J2EE本身，而是它总被误用。许多J2EE的支持者已经开始认识到，虽然在理论上很好，但J2EE在实践中却很失败，或者说没有提供真正的业务价值。

在本书中，笔者提供了一个真实环境的使用指南，使读者能够让J2EE在实践中起作用。其中的经验不但来自笔者成功设计的大型J2EE应用，还来自其挽救失败项目的过程，更包含了笔者使用J2EE规范的心得。

读者不但可以用J2EE解决一般问题，还可以避免J2EE项目中常见的代价高昂的错误。越过J2EE服务和API的复杂性，本书将带领读者建立不论是时间上还是预算上都最经济的简化方案。本书内容涉及：

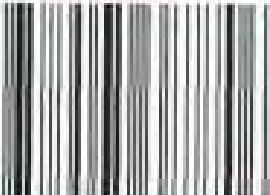
- 在何种情况下使用分布式体系结构
- 何时以及如何使用EJB
- 如何开发有效的数据存取策略
- 如何设计简洁并且可维护性高的Web接口
- 如何设计高性能的J2EE应用程序等

作者简介：

Rod Johnson是一位企业Java设计师，尤其擅长可编程Web应用。自其发布开始，他就一直使用Java和J2EE，而且他还见定义了Service 2.4规范的JSR 154专家组的成员。

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 7-5053-8770-7



9 787505 387706 >



责任编辑：李丽和 敬



ISBN 7-5053-8770-7 TP · 5086 定价：64.00元

Expert one-on-one J2EE Design and Development

J2EE设计开发 编程指南

〔美〕 Rod Johnson 著

魏海萍 于晓菲 毛选 等译

电子工业出版社

Publishing House of Electronics Industry

北京 ·

内 容 提 要

J2EE是当今可用于企业软件开发的最佳平台。本书的目标是让读者能够轻松自如地制定J2EE开发的体系结构决策与实现决策。内容涉及：在何种情况下使用分布式体系结构；如何高效地使用EJB；开发有效的数据存取策略；设计简洁并且可维护性高的Web接口；设计高性能的J2EE应用程序等。本书的观点是完全独立的，由它问题而作规范，并以作者在生产实践中使用J2EE的实际经验为基础。阅读完本书之后，熟悉J2EE的基本概念但可能还没有任何J2EE使用经验的开发人员，将能够自信地尝试J2EE项目。经验丰富的设计师或开发人员将能够从本书以实用角度为出发点的J2EE体系结构与实现的讨论中受益，因而本书适用于Java设计师、具有J2EE经验的开发人员以及拥有J2EE基础知识并希望从事J2EE项目的Java开发人员。



Copyright©2003 Wrox Press. All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

本书英文版由Wrox公司出版，Wrox公司已将中文版独家版权授予电子工业出版社及北京美迪亚电子信息有限公司。未经许可，不得以任何形式和手段复制或抄袭本书内容。

版权贸易合同登记号：01-2003-0742

图书在版编目（CIP）数据

J2EE设计开发编程指南 / (美) 约翰逊 (Johnson, R.) 著；魏海萍等译. —北京：电子工业出版社，
2003.7

ISBN 7-5053-8770-7

I. J... II. ①约... ②魏... III. JAVA语言—程序设计 IV. TP312

中国版本图书馆CIP数据核字（2003）第041721号

责任编辑：春丽和敬

印 刷：北京天竺新华印刷厂

出版发行：电子工业出版社 <http://www.phei.com.cn>

北京市海淀区万寿路173信箱 邮编：100036

北京吉海淀区翠微东里甲2号 邮编：100036

经 销：各地新华书店

开 本：787×1092 1/16 印张：39.375 字数：1000 千字

版 次：2003年7月第1版 2003年7月第1次印刷

定 价：64.00元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换，若书店售缺，请与本社发行部联系。联系电话：(010) 68279077

作者简介

Rod Johnson是一名专攻可缩放Web应用的企业Java设计师。Rod花费两年时间为FT.com——欧洲最大的商业门户网站设计并实现了一个J2EE解决方案，后来经过长途跋涉到了Everest Base Camp，做了一个孩子的父亲，并撰写了本书。Rod希望感谢Tristan给他提供了难度最大的挑战。

Rod在悉尼大学通过主修音乐与计算机科学获得了文学士学位，在回到软件开发之后他获得了音乐学博士学位。由于有C和C++方面的基础，Rod自从Java与J2EE发布以来一直使用它们。他目前是JSR 154 Expert Group的成员，这是一个定义Servlet 2.4规范的专家组。

Rod已经给包括“Professional Java Server Programming (J2EE and J2EE 1.3 edition)”和“Professional Java Server Pages (2nd edition)”在内的其他Wrox出版物贡献过几章内容，并且是Wrox出版社的一名审稿专家。他出席过包括Times Java (2001于Mumbai)在内的国际会议，而且他的作品已被登载在java.sun.com站点上。

Rod Johnson是澳大利亚人，但他目前生活和工作在伦敦。

他的联系地址是expert@interface21.com。

简 介

笔者相信，J2EE是当今可用于企业软件开发的最佳平台。它结合了Java编程语言的各种优点和过去10多年中企业软件开发的种种教训。

然而，这一承诺未必完全得到实现。许多J2EE工程项目中的投资回报是令人失望的。传输系统的速度常常太慢，结构常常过于复杂。开发时间常常和业务需求的复杂性不成比例。

原因出在什么地方呢？与其说是由于J2EE中的缺陷所致，倒不如说是因为J2EE常常没有得到正确使用的缘故所致。而不正确地使用通常是由忽视现实问题的体系结构和开发的方法所引起的。起着主要作用的一个因素是许多J2EE出版物中过分强调各种J2EE规范，而忽略了人们使用这些规范来解决的各种现实问题。现实应用中常常出现的许多问题受到了忽视。

当阅读J2EE讨论论坛时，笔者强烈地感觉到，许多开发人员几乎没有找到自己的行动准则和方向，结果浪费了大量时间和精力。在许多情况中，这些开发人员具有多年的IT经历，但仍觉得掌握J2EE很困难。

问题不是缺少关于J2EE组件的信息。许多图书和万维网站点在描述小服务程序、企业Java组件（Enterprise JavaBean，简称EJB）等方面都做得非常出色，而且对JNDI、RMI和JMS之类的技术也讨论得很充分。

问题出在如何达到下一个水平——怎样获得这些构造材料并使用它们以便在一段合理的期限内构造出满足现实业务需求的应用。在这方面，笔者觉得现有的大部分J2EE作品起着一种阻碍作用，而不是帮助作用。J2EE图书的世界与企业软件项目的世界之间存在一道鸿沟，也就是说脱了节。

本书的目标就是解决这个脱节问题，并提供如何在实践中有效地使用J2EE的明确方向和行动准则。笔者将帮助读者解决J2EE的常见使用问题，以及避免在J2EE项目中常犯的高代价错误。笔者将向读者揭示各J2EE服务和API的复杂性，以便读者能够按时间和预算要求构造出尽可能简单的可能解决方案。笔者将采用一种注重实际经验和实际效果的方法，进而对J2EE正统方法在实践中未能实现正确结果的地方提出质疑，并推荐已得到实践证明的有效方法。

笔者觉得，没有任何一本现有图书实现了这一目标。最接近这一目标的图书或许是Prentice Hall所出版的“Core J2EE Patterns”一书（ISBN：0-13-064884-1），这本书的出版曾经引起不少人的激动，因为终于有了一本论述如何使用J2EE组件的图书。“Core J2EE Patterns”确实是一本好书，也是J2EE设计师和开发人员的一个宝贵资源。尤其是，它所使用的方法已经得到广泛接受，但它是一个Sun出版物，而且无法帮助反映“各方的策略”。

这本书也只关注各种J2EE标准，而极少关注使用真正服务器时所遇到的问题。它没有提供明确的行动准则；只是经常不偏不倚地给出各种非常不同的可替换“设计模式（Design Pattern）”，不做任何具体的分析。已经能够在这些“设计模式”之间自信地做出选择的读者从本书中几乎得不到什么收获。

笔者见过的现有出版物、示例应用和讨论论坛越多，就越坚信J2EE需要一剂注重实际效果的健康良药。J2EE是一个伟大的平台。遗憾的是，针对它而提倡的许多体系结构没有帮助解决许多常见的问题。许多J2EE示例应用（比如Sun公司的Java Pet Store）十分令人失望。它们没有面对现实问题。它们的性能非常差，而且它们的代码常常没有考虑现实问题，因而提供了没有太大价值的模型。

笔者还深深感觉到，J2EE的新手与已经使用J2EE构造企业系统的开发人员之间在见解方面存在着差距。笔者以前的一名同事使用了一个引人遐想的绝妙词汇“多瘤的”来形容已经掌握了一项技术的实际使用技巧同时又留下了许多疤痕的开发人员。虽然J2EE的新手看起来像是遵守清规戒律的J2EE传教士，但“多瘤的”开发人员有所不同。他们不得不抛弃一些意识形态上的包袱来实现必要的功能度或达到足够的性能。像笔者的同事和笔者本人一样，他们已经发现，现实粗暴地改变了最初的想像。

在本书中，笔者将利用自己的亲身经历和行业知识帮助读者设计并开发出切实可行的解决方案，同时又不需要读者经历一个痛苦的过程来发现J2EE理论与现实之间的差距。

J2EE的神秘性

笔者以为，导致人们对J2EE失望的原因通常可以追溯到几个常见的神秘之处，而这几个神秘之处又证实了开发项目中的许多明显和隐含的假设。

- J2EE在应用服务器和数据库之间具有可移植性。
- J2EE是所有企业软件开发问题的最佳答案。如果使用非J2EE技术（比如RDBMS存储过程）能够解决的问题也能利用J2EE技术来解决，那么使用“纯”J2EE方法总是最好的。
- J2EE服务器负责性能和可缩放性，从而让开发人员能够集中精力实现业务逻辑。开发人员可以在很大程度上忽略J2EE“模式”的性能含义，并依赖产品中的可接受性能。
- J2EE能够让开发人员忘了数据存取和多线程化之类的低级问题，因为这些问题将由应用服务器透明地处理。
- 所有J2EE应用都应该使用企业Java组件（EJB），因为EJB是开发企业级应用的最基本J2EE技术。
- J2EE方面的任何问题不久将由更先进的J2EE应用服务器来解决。

下面，让我们来看一看上述每一个神秘之处。

可移植性是J2EE平台的一大馈赠。正如我们将要看到的，可移植性可以在实际应用中得以实现，但这不是J2EE的本质所在。绝大部分工程项目的需求是构造这样一个应用：它很好地解决一个目标平台上的某一个特定问题。在一个平台上运行很差的应用将永远不会被移植到其他平台（该应用可以被移植到运行在硬件威力更强的计算机上的另一个操作系统来获得足够的性能，但这不是专业开发人员所期望的那种可移植性）。

J2EE正统观念认为，应用在J2EE应用服务器之间应该是可移植的，并且必须能够处理不同的数据库。这两大目标之间的区别是非常重要的，有时也被忽视。应用服务器之间的可

移植性可以实现商业价值，因此常常也是一个现实的目标。数据库之间的可移植性比较容易实现，因此常常没有太大的商业价值。

可移植性通常被误认为“代码可移植性”：获得应用程序并能够无修改地在另一个平台上运行它的能力。笔者以为，这是一个需要付出极高代价的误解。对总代码可移植性的幼稚强调往往会在生产效率损失和不令人满意的交付方面导致严重的后果。尽管“编写一次，到处运行（WORA）”是Java本身所关注的一个现实目标，但同样也是一个适合于企业软件开发的危险口号，视资源的范围而定。

笔者不是在谈论开发“缩小包装式（shrink-wrapped）”构件（通常指EJB）的少数工程项目。这个颇具吸引力的概念仍需要到市场中得到验证。而且，笔者还见过一个以这两者为目标的重要构件：应用服务器可移植性（在这种情况下有意义）和数据库可移植性（几乎肯定会比它所值得的更麻烦）。

笔者更喜欢“设计一次，在任何地方重新实现几个接口（DORAFIA）”的口号。笔者承认，这句口号不那么有吸引力，而且让Java开发人员遵守它也是不可能的。但是，这种比较注重实效的方法在窗口化系统之类的其他领域内得到了广泛应用。

可移植性的神秘已经导致人们广泛地认为J2EE不能使用当今关系数据库的能力，而只能使用它们作为转储存储器。这种观点在现实生活中造成了极坏的影响。

这并不是说，笔者不相信J2EE应用能够或者应该是可移植的。笔者只是在说明一种更注重实效、更实际的可移植性观点。我们可以把J2EE应用设计成能够被轻松地移植，我们不能用诸如.NET之类的专有技术做同样的事情。

想像到J2EE是企业体系结构发展的最后阶段是十分令人愉快的；对象技术和Java语言的应用终于解决了10多年来困扰着业界的问题。遗憾的是，这不是现实，尽管在J2EE开发的许多方法中暗示了这一点。J2EE建立在早于它之前出现的许多技术之上。它只是向前进了一步，但不是最后一步，而且没有解决企业软件开发的所有问题。

对可移植性的过分强调以及这种以J2EE为中心的态度，已经导致这样一种假设：如果在标准J2EE中不能做某一件事情，那么做这件事情将是一个设计错误。这种假设甚至正在蔓延到随着EJB QL的引进而出现的EJB规范中（EJB QL是一种可移植但还不太成熟的查询语言，并且与可用于绝大多数J2EE应用的熟悉、成熟而又标准的SQL相比显得更复杂，但威力却更小）。

笔者把J2EE服务器看做一组企业资源（比如数据库）的指挥者。一个好的指挥对任何表演来说都是不可或缺的。但是，指挥不应该试图演奏各个乐器，而是应该把这项工作留给技巧熟练的专业人员。

最危险的神秘性或许是这样一种观点：J2EE是能够产生好的性能和可缩放性的一条方便的途径，同时其效率是一个比已得到认可的J2EE“设计模式”更无需担心的问题。这种观点会导致幼稚和效率不高的设计。这是非常遗憾的，因为Java业界之外的人对Java一直有这样一种恐惧心理：Java的性能极差。现在，事实表明Java语言提供了很好的性能，但有些流行的J2EE“设计模式”仍提供非常差的性能。

我们无法假设应用服务器能够负责性能和可缩放性。事实上，J2EE给我们提供了捆绑

J2EE应用服务器和数据库所需要的全部绳索。由于最佳性能一直是软件开发的主要追求目标，所以我们一直在用C和汇编语言编写Web应用。但是，性能对现实应用的商业价值是至关重要的。我们不能依赖Moore规则来让我们利用更快的硬件去解决性能问题。无论硬件威力有多么强大，出现性能问题的可能性始终是存在的。

J2EE服务器应该透明地处理数据存取之类的低级细节这一想法是非常有吸引力的。有时，这是可达到的，但会十分危险。另外，让我们来看一看关系数据库的例子。处于领先地位的企业级关系数据库管理系统（RDBMS）Oracle使用了一种完全不同于其他任何RDBMS产品的方式来处理加锁。使用粗糙还是精细事务意味着其性能在不同数据库之间有很大的差别。这就是说，“可移植性”会是虚假的，因为相同的代码在不同的RDBMS中可能会表现出很大差异。

Oracle和其他领先产品的价格都十分高昂，而且拥有令人印象深刻的能力。我们经常希望（或者需要）直接利用这些能力。J2EE在诸如事务管理和连接共享之类的基础结构性服务方面提供了十分有价值的标准，但在任何时候，我们都将不放弃各种厚厚的RDBMS产品说明手册。

“J2EE = EJB”神秘性会导致代价高昂的错误。EJB是一种复杂的技术，虽然很好地解决了一些问题，但在许多情况下也增添了比其商业价值更大的复杂性。笔者觉得，大多数图书都忽视了EJB的非常现实的缺点，而鼓励读者自动使用EJB。在本书中，笔者将冷静地分析EJB的长处和弱点，并提供关于何时使用EJB的明确准则。

允许所使用的技术（J2EE和其他任何一种技术）来确定一个业务问题的解决方法往往会导致很差的结果。此类错误的例子包括决定业务逻辑应该始终用EJB来实现，即决定实体组件是实现数据存取的单一方法。事实上，只有J2EE组件的一个很小子集（笔者认为还应该包括服务器小程序和无状态会话EJB），才是大多数J2EE应用的核心。其他J2EE组件的价值在很大程度上取决于待解决的问题。

笔者主张一种问题驱动而非技术驱动的方法（Sun公司的“J2EE Blueprints”通过建议一种J2EE技术驱动的方法，可能已实际造成了不良的影响）。尽管我们应该努力避免重复发明已有的东西，但是盲目地遵从我们绝不应该亲自实现服务器能够实现（尽管是无效率地实现）的东西这一正统观念将会付出极高的代价。用来处理事务管理等的核心J2EE基础结构是一个天赐之物，但这并不是说各种J2EE规范中所描述的所有服务都是天赐之物。

有些人将会争辩说所有这些问题不久将会得到解决，因为J2EE应用服务器变得越来越先进。例如，Container-Managed Persistence（容器管理式持久性，简称MCP）实体组件（Entity bean）的超高效实现将证明它比使用原始SQL的RDBMS存取具有更快的速度。这是幼稚的，并具有不可接受的风险。在IT行业中，几乎没有为信心留有什么地方。制定决策必须依据已得到证明的确凿证据，而且信心可能会用错地方。

现在，人们仍在激烈地争论着这样一个问题：J2EE的某些特性（比如实体组件）在许多情况中永远都无法像某些替代方法那样管用。而且，“理想中的乐土”仍然很遥远。例如，实体组件在1999年被首次引进到EJB规范中时，不久就提供了卓越的性能。然而，接下来的两年暴露了该原始实体组件模型中的严重缺陷。现在，EJB 2.0规范中的各种彻底变革仍有待证明，而且EJB 2.1规范已正在设法解决EJB 2.0规范中的遗留问题。

本书的不同之处

首先，本书的观点是独立的，以笔者和同事在生产实践中使用J2EE的经验为基础。笔者不打算布道。笔者主张使用J2EE，但警告读者谨防J2EE正统方法。

其次，本书的焦点是注重实际经验。笔者想帮助读者使用J2EE实现高性价比的应用。本书的目标是揭开J2EE开发的神秘性。它解释如何使用J2EE技术来减少而不是增加复杂性。尽管笔者并不把焦点集中在任何一个单独的应用服务器上，但将讨论读者使用实际产品时可能会遇到的一些问题。本书将不回避各种J2EE规范没有自然解决的现实问题。例如，我们在EJB层中怎样使用Singleton设计模式？在EJB层中应该怎样做日志记录？

本书不打算包括J2EE的全部情况，只打算解释解决常见问题的有效方法。例如，它将重点介绍如何与关系数据库一起使用J2EE，因为大多数J2EE开发人员都会面对O/R映射问题。一般说来，本书只打算在解决最常见的问题方面提供大量的帮助。

在全书中，我们将只着眼于一个统一的示例应用。在讨论每个问题时，我们不是使用一个不切实际的抽象示例，而是着眼于一个规模较大而又较注重实际的完整示例的一小部分。这个示例应用就是一个联机售票应用。它的设计目的不是为了举例说明特定的J2EE技术（像许多示例应用那样），而是为了说明J2EE设计师和开发人员所面对的常见问题。

本书注重的是质量、可维护性和生产效率。

这是笔者在处理自己的第一个J2EE工程项目时希望自己能够备有的图书。这样，它将会节省笔者大量的努力，也将会节省笔者雇主大量的金钱。

笔者采用的方法

本书是面向问题的，而不是面向规范的。和许多论述J2EE的其他图书不同，本书不打算全面介绍所有的服务和API，而是承认并非J2EE的所有部分都是同样有用的，或者说对所有开发人员都是有意义的，而且将讨论的重点放在构造典型解决方案中所用到的那些部分上。

软件设计与其说是科学不如说是一门艺术。J2EE的丰富性意味着寻找出同一个问题的多个有效解决方案（和许多坏的解决方案）常常是有可能的。尽管笔者尽了最大努力来解释自己的观点（或偏见），但本书自然也反映了笔者使用J2EE的经验和对待J2EE使用的看法。笔者介绍了自己觉得非常管用的一种方法。但是，这并不意味着它是惟一有效的方法。

本书大体上反映了笔者对待软件开发的如下看法：

- 笔者努力避免宗教式的立场。笔者永远理解不了有那么多的开发人员花费那么多的经历和热情致力于Usenet上的激烈争论。这实在没有什么好处可言。
- 笔者是一名实用主义者。笔者关心结果胜于意识形态。当处理一个工程项目时，笔者所追求的主要目标是按时和按预算交付一个高质量的结果。笔者所使用的技术是实现这个目标的一种工具，而不是结果本身。
- 笔者认为OO原理应该加强J2EE开发。
- 笔者认为可维护性对任何一个已交付项目的价值是至关重要的。

为了保持与这一实用主义方法相一致，笔者将经常引用Pareto Principle（巴累托定律），该定律说少量的原因（20%）担负结果的绝大部分（80%）。巴累托定律（最初起源于经济学）非常适用于实际的软件工程，而且我们在动手处理J2EE工程项目时将会不断地遇到它。例如，它可以提醒我们，设法解决某一特定领域中的所有问题会比只解决大多数实际应用中的各种要緊问题困难得多（而且性价比低得多）。

笔者的方法反映了Extreme Programming（极限程序设计，简称XP）的一些教训。笔者是一名方法学怀疑论者，不打算大肆宣传XP。这不是一本论述XP的图书，但笔者觉得XP给J2EE理论提供了一个宝贵的补充。尤其是，我们将会看到下列定律的价值：

- 简单性。XP专业人员主张做“可能管用的最简单事情”。
- 避免浪费精力。XP专业人员不添加他们可能永远不需要的功能。这种方法用首字母缩写词YAGNI（You Aren't Going to Need It）来表示。
- 重点放在整个开发过程中的测试上。

读者对象

本书的阅读对象是Java设计师、已经使用过J2EE的开发人员以及拥有J2EE基础知识而又希望从事J2EE项目的Java开发人员。

本书不是EJB、小服务程序、JSP和J2EE的入门读物。不熟悉这些领域的读者在阅读本书时可能需要参考一本综合性参考书（参见下文的推荐读物）。

本书目标

本书的目标是让读者能够轻松自如地制定J2EE开发的体系结构决策与实现决策。

在阅读完本书之后，熟悉J2EE的基本概念但可能还没有任何J2EE使用经验的开发人员，将能够自信地尝试J2EE项目。经验丰富的设计师或开发人员，将能够从本书以实用角度为出发点的J2EE体系结构与实现的讨论中受益。论述项目选择、测试和工具的各章节对正试图了解采用J2EE技术有什么影响的经理们特别有用。

本书内容

本书所涉及的内容包括：

- 如何做出关键性的J2EE体系结构选择，比如是否使用EJB以及在何处实现业务逻辑。
- J2EE Web技术，以及Model-View-Controller（模型 - 视图 - 控制器，简称MVC）体系结构模式在Web应用中的有效使用。
- 如何有效地使用EJB 2.0，其中包括引进含有本地接口的EJB所造成的影响。
- 如何选择应用服务器。
- 对J2EE开发至关重要的OO开发原理。
- 在J2EE应用中使用Java组件（JavaBean），以及这种使用如何帮助开发可维护与可移

植的应用。

- 在J2EE应用中使用XML和XSLT。
- J2EE事务管理。
- 如何在J2EE应用中有效地访问关系数据库。
- 如何使用通用的基础结构代码来解决常见问题，并保证应用代码只关注问题区内的业务逻辑。
- 如何测试J2EE应用，特别是如何测试Web应用。
- 如何设计具有满意性能的J2EE应用，以及如何改善现有应用的性能。
- 包装和部署J2EE应用。
- 主流应用服务器（比如BEA WebLogic和Oracle 9 Application Server）的特征，而这些特征可能会影响我们设计J2EE应用的方式。
- 了解可缩放性的基础性设计选择的各种影响。

本书还包括和讨论了能够在读者的应用中使用的大量通用基础代码。

阅读本书的前提知识

阅读本书的前提知识包括：

- 足够的Java语言技能；
- OO原理的牢固掌握；
- 对经典OO设计模式的熟悉；
- 对JSP的一定熟悉；
- 对Web和分布式对象协议（如HTTP和IIOP）的熟悉；
- 对J2EE的基础（如RMI、JDBC和JNDI）的了解。

再具有下列知识是最理想的：

- 关系数据库基础知识；
- 对事务概念（ACID属性和隔离层次）的了解；
- 对XML和XSLT的基本了解；
- 对UML，尤其是对类和序列图表的基本了解；
- 对基本计算机科学概念（如数据结构）的熟悉。

推荐读物

本书参考了由Addison Wesley出版的经典图书“Design Patterns: Elements of Reusable Object-Oriented Software”一书（ISBN 0-201-63361-2）中所讨论的23个设计模式。虽然这本经典著作中所介绍的许多模式只不过编纂了任何一名经验丰富的OO专业人员的设计习惯，但为设计师提供了一种公用语言，并且是任何一名需要提高技能的开发人员的必读之物。本书假设读者已经阅读过并理解了这本经典著作。

本书还使用了由Prentice Hall出版的“Core J2EE Patterns: Best Practices and Design

Strategies”一书（ISBN 0-13-064884-1）中所介绍的设计模型名称，因为这些模式名称已经得到广泛的接受。虽然不必参考“Core J2EE Patterns”一书也能阅读本书，但建议读者读一读这本书。不过需要注意的是，笔者在几个方面推荐了一种不同的方法。另外，“Core J2EE Patterns”一书的某些章节，特别是和实体组件（Entity bean）有关的那些章节在EJB 2.0规范发布之后已经过时。

另外，本书的一些J2EE设计模式参考了由Wiley出版的“EJB Design Patterns”一书（ISBN 0-471-20831-0）。同样，建议读者读一读这本书，尽管它不是必读的背景读物。

阅读一本论述J2EE 1.3平台的好参考书也是很有必要的。笔者推荐由Wrox Press出版的“Professional Java Server Programming J2EE 1.3 Edition”一书（ISBN 1-861005-37-7，中文版《J2EE编程指南（1.3版）》已由电子工业出版社出版）。由Ed Roman编写和Wiley出版的“Mastering Enterprise JavaBeans (Second Edition)”（ISBN 0-471-41711-4）也是一本论述EJB的好书。

读者可以从http://java.sun.com/j2ee/sdk_1.3/techdocs/api/index.html站点上联机地获得针对J2EE 1.3平台的完整Javadoc。

最后，所有专业的J2EE设计师和开发人员都应该参考定义J2EE 1.3平台的各种规范，这些规范可以从<http://java.sun.com/j2ee/download.html>站点上获得。在本书中，笔者将参考下列规范的相关部分（比如EJB 17.4.1节）：

- J2EE 1.3
- EJB 2.0
- Servlet 2.3
- JSP 1.2

在相关的地方，笔者还将参考现在仍处于公开草案阶段并且也可以从Sun站点上获得的各种J2EE 1.4规范版本：EJB 2.1、Servlet 2.4和JSP 2.0。

使用本书的软硬件需求

为了运行本书中的示例，读者将需要：

- Java 2 Platform, Standard Edition SDK v1.3或以上版本。我们使用了Sun SDK 1.3.1_02来运行所有样本代码。
- 一个支持J2EE 1.3的应用服务器。我们为本书中的示例应用使用了JBoss 3.0.0。
- 一个RDBMS。我们为示例应用使用了Oracle 8.1.7i。读者可以从Oracle站点上获得“Personal Edition”的一个免费评估拷贝。要想使用一个不同于Oracle 8.1.7或以上版本的数据库，读者将需要对示例应用做一些修改（文中已明确标出）。

要想运行示例应用，读者将需要下列第三方库：

- Apache Log4j 1.2
- JSP Standard Tag Library (JSTL) 1.0的一个实现

要想使用第3章中所讨论的那些测试策略，读者将需要如下产品：

- JUnit 3.7或以上版本

- Apache Cactus J2EE测试框架

要想运行第13章中的所有Web内容生成示例，读者将需要如下产品：

- Apache Velocity 1.3
- Lutris XMLE 2.1
- iText PDF生成库
- Domify XML生成库

关于如何安装和配置最后这4个产品的信息，请参见附录A。

上面所讨论的所有第三方产品和库都是免费的，并且是开放源代码的。

要想构造源代码，读者将需要Apache Ant 1.4.1或较新版本，其中包括可选的任务支持。

本书中所有示例的完整源代码可以从我们的Web站点<http://www.wrox.com/>上通过下载来得到。下载有两个版本：一个版本含有上面讨论的所有库，而另一个版本是一个小得多的捆绑，且只含有本书中所讨论的源代码和编译代码。如果读者已经拥有或者希望下载上面所列举的那些第三方产品，只需下载那个较小的版本即可。

本书中的约定

为了帮助读者从正文中获得最大的信息量和跟踪正在发生的事情，我们在整本书中使用了许多约定。

这含有不应该忘记的重要信息，而且这些信息与周围的正文直接相关。

这种背景样式用于当前讨论的旁注。

至于正文中的其他样式，本书使用了如下约定。

- 当介绍重要单词时，给出了它们的原文。
- 使用后面的样式表示键盘上的键击：**Ctrl-K**。
- 使用后面的样式表示正文中的文件名、代码以及用户界面上的文字和URL：**persistence.properties**。

本书使用两种方式表示代码：

In our code examples, the code foreground style shows new, important, pertinent, or Code.

While code background shows code that's less important in the present context, or has been seen before.

客户支持

我们始终十分重视听取读者的反馈意见，而且希望知道读者对本书的看法：喜欢什么，不喜欢什么，希望我们下次在什么方面做得更好。读者可以把自己的意见发给我们，给**feedback@wrox.com**发送电子邮件。请务必在邮件中注明书名。

如何下载本书的示例代码

当访问Wrox站点<http://www.wrox.com>时，通过我们的Search工具，或者通过使用书名列表之一，简单地找出本书的书名。然后，单击Code栏中的Download，或者单击本书的详细信息页上的Download Code。

可以从我们的站点中下载的文件已经使用WinZip进行过归档。当读者已经把那些附件保存到自己的硬盘驱动器上的一个文件夹中时，需要使用WinZip之类的解压缩程序抽取那些文件。当抽取那些文件时，代码通常被抽取到章文件夹中。当开始抽取过程时，读者需要确保自己的软件（如WinZip）被设置成使用文件夹名。

勘误表

我们已经尽了一切力量保证正文或代码中没有任何错误。但是，人无完人，出错是在所难免的。如果读者在我们的某本书中发现了错误之处，比如拼写错误或故障代码，我们将非常感激读者提供反馈信息。通过发送勘误表，你可以节省其他读者受挫的时间，当然也可帮助我们提供更高质量的信息。请直接把这些信息以电子邮件的形式发送给support@wrox.com，你的信息将接受检查，如果正确，这些信息将被张贴到用于本书的勘误表页上，或用在本书的后续版本中。

要想查找Web站点上的勘误表，请转到<http://www.wrox.com>，并通过我们的Advanced Search工具或书名列表，简单地找出书名。然后，单击Book Errata链接，该链接在本书的详细信息页上位于封面图形的下面。

p2p.wrox.com

要想与作者和同行讨论，请加入P2P邮件表。除了我们的一对一电子邮件支持系统之外，我们的特有系统还提供关于邮件表、论坛和新闻组的programmer to programmer™联系人。如果张贴一个查询到P2P上，读者可以确信你的问题正被许多Wrox作者和当时正在我们的邮件表上的其他业界专家所检查。不仅在阅读本书期间，而且在开发自己的应用期间，读者都将会在p2p.wrox.com上找到许多对自己有帮助的不同列表。

要想订阅一个邮件表，只需按如下步骤进行操作即可：

1. 转到<http://p2p.wrox.com/>。
2. 从左菜单栏上选择合适的类别。
3. 单击希望加入的邮件表。
4. 按照说明来订阅和填写你的电子邮件地址和密码。
5. 回复你所接收到的确认电子邮件。
6. 使用订阅管理器来加入更多的列表，并设置你的电子邮件参数选择。

本系统为什么提供最佳支持

读者可以选择加入那些邮件表，也可以选择将它们作为一个周文摘来接收。如果没有时间或便利工具来接收邮件表，读者可以搜索我们的存档文件。宣传和垃圾邮件已被删除，而且你自己的电子邮件地址将由特有的Lyris系统来保护。关于加入或离开邮件表的查询以及关于邮件表的其他任何一般性查询，都应该被发送给listsupport@wrox.com。

目 录

第1章 J2EE体系结构	1
企业级体系结构的目标	1
决定是否使用分布式体系结构	3
J2EE设计中的新考虑	4
何时使用EJB	5
数据存取	9
状态管理	11
J2EE体系结构	12
Web层设计	20
设计可移植的应用	22
小结	24
第2章 J2EE项目的选择与风险	26
依据规范版本开发一个策略	26
选择应用服务器	27
“纯技术”陷阱	36
何时使用替代技术来补充J2EE	37
可移植性问题	38
中间整备环境与发布管理	40
建立开发团队	41
选择开发工具	44
识别和降低风险	47
小结	50
第3章 J2EE应用的测试	52
测试能达到什么目的	53
定义	53
正确性的测试	54
性能与可缩放性的测试	83
测试的自动化	85
测试的补充方法	86
小结	87

第4章 J2EE项目的设计技术与编程标准	89
J2EE应用的OO设计推荐标准	89
编程标准	116
为什么（以及怎样）不从头开始	139
小结	144
第5章 示例应用的需求	151
概述	151
用户总数	152
前提	153
范围限制	154
交货进度表	154
Internet用户界面	154
售票处用户界面	166
非功能性需求	167
硬件与软件需求	168
小结	169
第6章 应用J2EE技术	170
分布式体系结构何时合适	170
决定何时使用EJB	176
决定怎样使用EJB	187
决定何时使用带有JMS的异步调用	192
身份鉴别和授权	199
决定何时使用XML	201
高速缓存来改进性能	208
小结	211
第7章 J2EE应用中的数据存取	213
数据存取目标	213
业务逻辑与持久性逻辑	214
对象驱动与数据库驱动建模：一场哲学辩论	214
O/R映射与“阻抗不匹配”	216
Data Access Object (DAO) 模式	218
使用关系数据库	219
可移植性与性能的比较	224
分布式应用中的数据交换	226
常见的数据存取问题	228

何处执行数据存取	233
小结	238
示例应用中的数据建模	239
第8章 使用实体组件进行数据存取	244
实体组件概念	244
CMP与BMP的比较	250
EJB 2.0中的实体组件	252
实体组件的高速缓存	258
实体组件的性能	262
实体组件的工具支持	263
小结	263
第9章 实际的数据存取	266
数据存取技术选择	266
JDBC的细微之处	273
一个通用的JDBC抽象框架	277
在示例应用中实现DAO模式	304
小结	310
第10章 会话组件	312
使用无状态会话组件	312
使用有状态会话组件	314
可应用于会话组件的J2EE设计模式	322
会话组件实现问题	327
示例应用中的会话组件	336
小结	336
第11章 基础结构与应用实现	338
基础结构	338
实现业务逻辑	369
小结	377
第12章 Web层的MVC设计	380
Web开发的挑战	380
Java Web开发中的教训	381
Web层的设计目标	385
MVC概念与Front Controller J2EE设计模式	386

Web应用框架	392
Web应用框架到总体应用体系结构中的集成	400
示例应用中使用的Web应用框架	401
Web层会话管理	420
处理用户输入	423
实现示例应用中的Web层	435
小结	440
 第13章 Web层中的视图	 442
控制器与视图的分离	443
构造预订页面的视图	445
JSP视图	451
专用模板语言	467
XSLT	474
置标生成的替代方法	483
二进制内容的生成	492
视图合成和页面布局	495
小结	499
 第14章 应用的包装与部署	 501
包装	501
应用部署：常见概念	513
在JBoss 3.0上部署示例应用	517
小结	523
 第15章 应用的性能测试与调整	 525
策略问题与定义	525
性能与吞吐量的测试工具	528
查找性能或可缩放性问题	535
解决性能或可缩放性问题	542
案例研究：示例应用中的“Display Show”页面	553
分布式应用中的性能	562
Web层性能问题	572
J2EE应用中性能与可缩放性问题的主要原因	580
小结	581
 第16章 结论：让J2EE为我所用	 582
 附录A 实现视图技术	 587

第1章 J2EE体系结构

J2EE提供了许多体系结构选择。J2EE还提供了许多构件类型（component type，比如服务器小程序、EJB、JSP页和服务器小程序筛选器），而J2EE应用服务器提供了许多附加服务。虽然这组选择能够让我们为每个问题设计出最佳的解决方案，但也带来了危险。J2EE开发人员会被J2EE所提供的这些选择所淹没，或者说会禁不住诱惑去使用不适合待解决问题的基础结构，原因很简单：因为它是现成的。

在本书中，笔者的目标是帮助专业的J2EE开发人员和设计师做出正确的选择来按时和按预算交付高质量的解决方案。笔者将把重点放在J2EE的此类特性上：这些特性已经证明对解决企业软件开发中的各种最常见问题是最有用的。

在本章中，我们将讨论J2EE体系结构开发中的各种高层次选择，以及怎样决定使用J2EE的哪些部分来解决现实问题。本章将要探讨下列内容：

- 分布式和非分布式应用，以及怎样选择合适的模型；
- EJB 2.0规范中的各种修改和Web服务的出现对J2EE设计的影响；
- 何时使用EJB；
- J2EE应用的数据存取策略；
- 4个J2EE体系结构，以及怎样在它们之间进行选择；
- Web层设计；
- 可移植性问题。

本书反映了笔者的经验和笔者跟其他企业设计师的讨论。笔者将尽力在本书的其余篇幅中阐明本章中所提出的各种主张。不过，存在大量看法是必然的。

特别是，笔者试图想让读者了解的要点将是我们应该运用J2EE来实现OO设计，而不是让J2EE技术支配对象设计。

本章假设读者已经熟悉了J2EE构件。我们将在后续章节中仔细地看一看容器服务，但如果读者还不熟悉本章所讨论的这些概念，请参考一本论述J2EE的入门图书。

企业级体系结构的目标

在开始仔细分析J2EE体系结构中的具体问题之前，先来看一看我们正试图实现的东西。

设计良好的J2EE应用应该满足下列这些目标。需要注意的是，尽管我们正在关注基于J2EE的解决方案，但这些目标适用于所有企业级应用。

- **是坚固的**

企业级软件对一个组织机构来说是十分重要的。它的用户期望它是可靠的和无缺陷的。因此，我们必须了解和利用J2EE中能够帮助我们构造坚固解决方案的各部分，

并必须确保我们编写高质量的代码。

• 是可工作的和可缩放的

企业级应用必须满足用户的性能期望值。它们还必须显示出足够的可缩放性——一个应用在给定相应硬件的条件下支持增加的负载的潜力。可缩放性对Internet应用来说是一个特别重要的考虑因素，因为预测此类应用的用户数量与行为是十分困难的。了解J2EE基础结构是满足这些目标所必不可少的。可缩放性一般会要求在一个聚类（cluster）中部署多个服务器实例。聚类是一个需要高级应用服务器功能度的复杂问题。我们必须确保把应用设计成让聚类中的操作是高效率的。

• 利用OO设计原理

OO设计原理为复杂系统提供了已经过证明的好处。好的OO设计原理提倡使用已经过证明的设计模式（Design Pattern）——常见问题的重复解决方案。设计模式的概念在OO软件开发中的普及应归功于Addison Wesley出版的经典图书“*Design Patterns: Elements of Reusable Object-Oriented Software*”（ISBN 0-201-63361-2），这本经典著作描述了具有广泛适用性的23个设计模式。这些模式不是技术特有或语言特有的。

至关重要的是，我们使用J2EE来实现OO设计，而不是让我们的J2EE来支配对象设计。当今，存在一个完整的“J2EE模式”行业。虽然许多“J2EE模式”是颇有价值的，但经典（非技术特有）的设计模式更有价值，而且仍与J2EE高度相关。

• 避免不必要的复杂性

Extreme Programming（XP）的专业人员主张做“可能管用的最简单事情”。我们应该提防有可能导致一个应用体系结构不适用的极端复杂性。由于现有构件的范围很广，开发人员可能会禁不住诱惑而过分设计J2EE解决方案，进而为与业务需求不相关的能力而采纳更大的复杂性。复杂性增加了整个软件周期内的成本，因此会成为一个严重的问题。另一方面，全面的分析必须保证我们没有盲目而又过分简单地看待需求。

• 是可维护的和可扩展的

迄今为止，维护是软件周期中代价最高的阶段。设计J2EE解决方案时考虑可维护性是特别重要的，因为采用J2EE是策略上的一个选择。J2EE应用很有可能是多年内一家组织的软件混合解决方案的一个关键部分，同时还必须能够接纳新的业务需求。可维护性和可扩展性在很大程度上取决于明确的设计。我们不仅需要保证该应用的每个构件都有一个明确的责任，而且还需要保证紧密耦合的构件不会妨碍维护性。

• 按时交付

生产效率是一个至关重要的考虑因素，而且该因素在动手处理J2EE时经常遭到人们忽视。

• 测试起来容易

测试在整个软件周期内是一项必不可少的工作。我们应该针对测试的简易性来考虑设计决策的影响。

- 提倡重用

企业级软件必须符合一家组织的长期策略。因此，鼓励重用是十分重要的，目的是为了让代码重复得到最小化（在项目内和整个项目期间）和投资得到充分利用。代码重用通常源自好的OO设计原理，而我们也应该一贯地使用应用服务器所提供的宝贵基础结构，因为这么做将会简化应用代码。

视应用的业务需求而定，我们还可能需要满足下列这些目标：

- 对多种客户类型的支持

有一个隐含的假设：J2EE应用始终需要支持多种J2EE技术客户类型，比如Web应用、使用Swing或其他窗口化系统或Java小程序的独立Java GUI等。但是，这种支持常常是不必要的，因为“轻型”Web界面正越来越广泛地得到使用，甚至用于为组织内部使用而设计的应用（部署的简易性就是针对这种使用的主要原因之一）。

- 可移植性

可移植性在J2EE应用所使用的各种资源（比如数据库）之间究竟有多么重要呢？可移植性在应用服务器之间究竟有多么重要呢？可移植性不是J2EE应用的一个自动目标。它只是J2EE帮助我们实现的某些应用的一个业务需求。

最后两个目标的重要性只是一个业务需求问题，而不是一个信不信J2EE的问题。如果我们只打算实现相关的目标，就会得到一个将在整个项目周期内提高质量和降低成本的简单性好处。

决定是否使用分布式体系结构

J2EE为实现分布式（distributed）体系结构提供了出色的支持。同一个分布式J2EE应用的构件可以被分布给运行在一台或多台物理服务器上的多个JVM。分布式J2EE应用以使用具有远程接口的EJB作为基础，而远程接口能够让应用服务器隐藏掉分布式构件的访问和管理的大部分复杂性。

但是，J2EE对分布式应用的出色支持已经导致这样一种误解：J2EE必定是一个分布式模型。

这是一个至关重要的地方，因为分布式应用很复杂，并导致显著的运行时开销，而且要求设计工作保证令人满意的性能。

人们常常有这样一种想法：分布式模型是实现坚固而又可缩放的应用的惟一方法。这种想法是站不住脚的。聚类一个应用以便把它的所有构件都集中布置（collocate）在同一个JVM中也是可能的。

分布式体系结构提供了下列好处：

- 支持许多需要一个共享式业务对象“中间层”的客户（可能具有不同类型）的能力。这个考虑因素不适用于Web应用，因为Web容器已提供了一个中间层。
- 部署任一应用构件到任一物理服务器上的能力。在某些应用中，这对负载均衡来说是非常重要的（请想像一下当一个Web接口做少量工作而业务对象做密集型计算时的

一种场景。如果使用一个J2EE分布式模型，我们就可以把该Web接口运行在一台或多台计算机上，而让许多服务器运行各种做计算的EJB。虽然每个调用的性能会有所下降，因为远程请求的系统开销将会降低该调用的速度，但是每个硬件的总吞吐量可以通过消除瓶颈来得到改善）。

但是，分布式体系结构也引发了许多麻烦的问题，尤其是如下问题：

- **性能问题**

远程请求比本地请求慢许多倍。

- **复杂性**

分布式应用的开发、调试、部署和维护都很困难。

- **实行OO设计方面的限制**

这是一个十分重要的方面，我们不久将进一步讨论这方面的内容。

分布式应用带来了许多有趣的挑战。由于分布式应用的复杂性，本书（和一般J2EE图书）使用了大量篇幅来专门讨论分布式J2EE应用。但是，倘若有选择的余地，最好是通过选择一个非分布式解决方案来避开分布式应用的各种复杂性。

根据笔者的经验，分布式应用的部署灵活性好处被过分夸大了。分布不是实现坚固、可缩放应用的惟一方法。使用了远程接口的大多数J2EE体系结构往往和所有构件一起被部署在相同的服务器上，以避免真正远程调用的性能开销。这意味着一个分布式应用的复杂性是没有必要的，因为它没有产生任何真正的好处。

J2EE设计中的新考虑

J2EE 1.2规范提供了简单的选择。EJB具有远程接口，并且只能用在分布式应用中。Remote Method Invocation（远程方法调用，简称RMI）（在JRMP或IIOP上面）是支持远程客户的惟一选择。

后来，有两项开发（一项在J2EE内部，而另一项在J2EE外部）对J2EE设计产生了意义深远的影响：

- EJB 2.0规范除了允许EJB有远程接口之外，还允许EJB有代替远程接口的本地接口。EJB可以由一个运行在同一个JVM内的集成性J2EE应用中的构件通过这些EJB的本地接口来调用，比如一个Web应用中的构件。
- 出现了基于XML的Simple Object Access Protocol（简单对象访问协议，简称SOAP），该协议已是一个得到广泛认可的、平台独立的、针对RMI的标准，并且广泛地支持Web服务。

引进EJB 2.0本地接口的主要目的是为了解决EJB 1.1实体组件(entity bean)的严重性能问题。在规范委员会对引进“独立对象”来改进实体组件性能的问题没能取得一致意见之后，EJB 2.0本地接口成为了最后时刻的一个补充物。不过，本地接口的意义已经远远超出了实体组件的范畴。现在，我们拥有了在不采用RMI语义的情况下是否使用EJB的选择。

虽然针对Web服务的某些较大胆声称（比如通过注册表的自动服务发现）仍有待证明在商业上是切实可行的，但SOAP已经证明了它对远程过程调用的价值。SOAP支持已经被内建

到了Microsoft的.NET（J2EE的主要竞争对手）中，而且可能会取代平台特有的远程化协议。Web服务的出现向分布式应用的传统J2EE假设提出了挑战。

在各种J2EE规范的下一个版本中，最重要的增强之一将是与标准Web服务支持的集成。但是，有几个出色而又易于使用的Java工具集能使J2EE 1.3应用实现和访问Web服务。例如，请参见Sun公司的Java Web Services Developer Pack (<http://java.sun.com/webservices/webservicespack.html>) 和Apache Axis SOAP实现 (<http://xml.apache.org/axis/index.html>)。

由于有了EJB本地接口和Web服务，我们现在不使用RMI也可以使用EJB，而且不使用EJB也可以支持远程客户。这使得我们在设计J2EE应用时有了更大的自由。

何时使用EJB

在设计J2EE应用时，最重要的设计决策之一是用不用EJB。EJB常常被理解成J2EE的核心。这是一个误解：EJB只不过是J2EE提供的选择之一。从理论上说，它适合解决某些问题，但是在许多应用中只增添很小的价值。

当业务需求明确指出需要一个分布式体系结构，而且RMI/IOP是必然的远程化协议时，EJB给了我们一个标准的实现。我们可以把自己的业务对象编写成具有远程接口的EJB，并可以使用EJB容器来管理这些EJB的生存周期和处理远程引用。这比使用了RMI的定制解决方案高级得多，因为RMI要求我们管理服务器端对象的生存周期。

如果业务需求没有明确指出需要一个分布式体系结构，或者RMI/IOP不是必然的远程化协议，是否使用EJB的决策将会困难得多。

EJB是J2EE中最复杂的技术，也是最时髦的J2EE词汇。这会导致开发人员因如下这个错误的原因而使用EJB：因为从简历上看起来EJB经历非常好，因为存在使用EJB是一种最佳习惯的广泛信念，因为EJB被看做是编写可缩放Java应用的唯一方法，或者只因为存在EJB。

EJB是一种高端技术。它解决某些问题非常好，但不应该没有理由地使用它。在本节中，我们将冷静而客观地看一看使用EJB的影响，以及对是否使用EJB的决策将会产生影响的重要考虑因素。

使用EJB的影响

EJB规范的关键目标之一是简化应用代码。EJB 2.0规范（§ 2.1）规定，“EJB体系结构将使编写应用变得容易：应用开发人员将不必了解低级事务和状态管理细节、多线程化、连接建池以及其他复杂的低级API”。

从理论上说，通过把所有低级问题都推给EJB容器，开发人员能够自由地把他们的精力都用于业务逻辑。令人遗憾的是，经验表明这种愿望在实践中常常是无法实现的。使用EJB给应用增加的复杂性与消除的复杂性至少是一样大的。此外，让开发人员“不必了解”他们的应用所面对的各种企业软件问题可能是十分危险的。

引进EJB技术有如下这些实际影响，而且我们应该十分仔细地权衡这些影响。

- **使用EJB使应用变得更难测试**

分布式应用测试起来始终比运行在同一个JVM中的应用更困难。EJB应用——无论它们使用远程还是本地接口——测试起来都很困难，因为它们严重依赖于容器服务。

- **使用EJB使应用变得更难部署**

使用EJB引入了许多部署问题。例如：

- 复杂的类装入器。包含了EJB JAR文件和Web应用的一个企业级应用将包含许多类装入器。具体细节随着服务器而有所不同，但是避免类装入问题（比如没有查找类的能力或不兼容的类版本）是一个很重要的问题，而且还要求对应用服务器设计的有所了解。
- 复杂的部署描述符。虽然EJB部署描述符的部分复杂性降低了EJB代码中的复杂性（例如，在事务管理方面），但其余复杂性是没有必要的。在这方面，工具可以帮忙，但避免复杂性而不是依赖工具来管理复杂性是最理想的。
- 更慢的开发 - 部署 - 测试周期。部署EJB通常比部署J2EE Web应用慢。因此，使用EJB会降低开发人员的工作效率。

使用J2EE的大多数实际挫折与EJB有关。这决不是无足轻重的事情；如果EJB没有提供补偿的好处，这将会付出时间和金钱上的代价。

- **使用具有远程接口的EJB可能会妨碍OO设计的实施**

这是一个严重的问题。使用EJB（一种实际上应该是一个实现选择的技术）来驱动总体设计是危险的。在Wiley所出版的“EJB Design Patterns”一书（ISBN 0-471-20831-0）中，6个“EJB层体系结构模式”中有4个不是真正的设计模式，而是针对使用含有远程接口的EJB所引入的问题的迂回解决方案（Session Facade（会话界面）设计模式试图最小化网络往返行程的数量，结果是一个具有粗粒度接口的会话组件（session bean）。接口粒度实际上应该由正常的对象设计因素来支配。EJB Command（EJB命令）设计模式试图最小化EJB远程引用中的网络往返行程数量，虽然它的结果是比较良性的。Data Transfer Object Factory（数据传输对象工厂）设计模式解决从EJB层中转移数据到一个远程客户的问题，Generic Attribute Access（通用属性存取）设计模式试图降低处理实体组件的系统开销）。

不必要地使用具有远程接口的EJB所导致的致命结果包括如下这些：

- 由最小化远程方法调用数量的要求所决定的接口粒度和方法签名。如果业务对象本身是细粒度的（情况也常常如此），这将导致不自然的设计。
- 对串行化（决定将通过RMI进行通信的对象的设计）的需要。例如，我们必须决定应该随每个可串行化对象一起返回多少数据，即应该遍历联合体吗？如果应该，遍历到什么深度？我们还必须编写另外的代码从任何不可串行化的对象中提取远程客户所需要的数据。
- 远程引用时应用业务对象中的不连续性。

这些缺陷在我们真正需要分布式语义时不适用。当需要分布式语义时，EJB不是问题的诱因，而是分布式应用的一个优秀基础结构。但是，当不需要分布式语义时，如果使用EJB会使一个应用变成分布式的，那么使用EJB会有一个致命的结果。正如前面已经讨论过的，

分布式应用比运行在单个服务器中的应用复杂得多。EJB还增加了我们可能希望避开的一些额外问题。

- **使用EJB可能会使简单的事情变得很困难**

一些简单事情在EJB（具有远程或本地接口）中令人惊奇的困难。例如，实现Singleton（单元集）设计模式和高速缓存只读数据是十分困难的。EJB是一种重量级技术，而且使一些简单问题变成繁重的工作。

- **应用服务器选择的减少**

Web容器的数量比EJB容器多，而且Web容器往往比EJB容器更容易使用。因此，与EJB应用相比，Web应用可以运行在范围较广泛的服务器上或相同服务器的较便宜版本上，并且只需要较简单的配置和部署（但是，如果我们有一个针对某个集成式J2EE服务器的许可，那么成本不是要关心的事情，而且EJB容器可能已通过在其他项目中的运用为我们所熟悉）。

这些是需要考虑的重要因素。大多数书籍忽略了这些重要因素，因而只关注理论，而不太关注实际经验。

下面让来回顾一下在J2EE应用中使用EJB的部分理由——好的和坏的。

使用EJB的可疑理由

下面是使用EJB的几个不太令人信服的理由：

- **为了通过把业务对象暴露为EJB来保证干净的体系结构**

EJB有助于好的设计习惯，因为好的设计习惯会产生一个清晰的业务对象（会话EJB）层。但是，相同的结果也可以利用普通Java对象来实现。如果使用具有远程接口的EJB，我们将被迫使用业务对象的粗粒度访问来最小化远程方法调用的数量，而该数量会迫使我们在业务对象接口中做出不自然的设计选择。

- **为了允许给数据存取使用实体组件**

笔者将这个理由看做使用EJB的一个可怜理由。虽然实体组件已经产生了较大的利益，但它们有一个可怜的过去。我们以后将较详细地讨论J2EE应用的数据存取选择。

- **为了开发坚固、可缩放的应用**

设计完备的EJB应用具有很好的缩放性——但Web应用也具有很好的缩放性。而且，EJB使产生误解的潜在可能性变得更大：与不使用EJB相比，使用EJB，缺乏经验的设计师更有可能开发出一个速度慢且不可缩放的系统。只有当一个远程EJB接口基于无状态会话EJB时，分布式EJB系统才有可能以更大的运行时系统开销为代价，提供比Web应用更大的可缩放性（在这种方法中，业务对象可以根据需要被运行在任意多个的服务器上）。

使用EJB的充分路由

下面是强烈建议使用EJB的几个理由：

- **为了允许应用构件的远程访问**

如果通过RMI/IOP的远程访问是必需的，这就是一个充分理由。但是，如果Web服务式的远程访问是必需的，就根本没有必要使用EJB。

- 为了允许应用构件被分布到多台物理服务器上

EJB提供了对分布式应用的出色支持。如果我们正在构造一个分布式应用，而不是给一个其内部未必是分布式的应用添加Web服务，那么EJB是一个显而易见的选择。

- 为了支持多种Java或CORBA客户类型

如果我们需要开发一个Java GUI客户软件（使用Swing或其他窗口化技术），EJB是一个非常不错的解决方案。EJB与CORBA的IIOP是可互操作的，因此是服务CORBA客户的一个上佳解决方案。由于这样的应用中根本不存在Web层，所以EJB层提供所需要的中间层。否则，我们就回到了客户-服务器应用和有限可缩放性的年代，因为没有能力代表多个客户把数据库连接之类的资源合并起来。

- 为了在异步模型适用时实现消息消费者

消息驱动式组件生成特别简单的JMS消息消费者。这是一种很少见的情况，而且在这种情况下EJB是“可能管用的最简单东西”。

按情形考虑EJB使用的理由

使用EJB的下列理由应该根据具体情形加以考虑：

- 为了使应用开发人员免于编写复杂的多线程代码

EJB把同步的负担从应用开发人员那里转移给了EJB容器（EJB代码被编写得好像是单线程的）。这是一个非常充分的理由，但证明使用EJB是否充分合理取决于个别应用。

围绕使用EJB来解决线程化问题的这个必要条件有许多的FUD（Fear（担心）、Uncertainty（不确定性）以及Doubt（怀疑））。编写线程安全的代码超出了一名专业企业开发人员的职责范围。无论我们是否使用了EJB，都必须在服务器小程序（servlets）和其他Web层类中编写线程安全的代码。此外，EJB不是简化并发编程的惟一方法。我们没有必要从零开始实现自己的线程化解决方案，可以使用一个标准包，比如Doug Lea的util.concurrent。关于该包的综述，请参见<http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>站点，这个包提供了许多常见并发问题的解决方案。

EJB的多线程代码的简化是使用EJB的一个有力但非决定性的理由。

- 为了使用EJB容器的透明事务管理

EJB可以使用Container-Managed Transaction（容器管理式事务，简称CMT）。这使得事务管理能够被大部分地从Java代码中转移出来，并在EJB部署描述符中被说明性地处理。如果应用代码需要回退一个事务，它只需关心事务管理就行了。实际的回退可以仅用单个方法调用来完成。

CMT是使用EJB的主要好处之一。由于企业级应用几乎总是事务性的，如果没有EJB CMT，我们通常需要使用Java Transaction API（Java事务API，简称JTA）。JTA是一个不太复杂的API，因此避免直接使用它是明智的（但不是必需的）。和使用线程化一样，使用助手类（helper class）来简化JTA编程和降低出错概率是可行的。

需要注意的是，J2EE事务管理基础结构（比如，协调分布于不同企业资源上的事务的能力）是运行在一个J2EE服务器内的所有代码都可以使用的，而不是仅EJB可以使用的；问题是我们在用来控制该基础结构的那个API。

借助CMT的说明性事务管理的可用性是使用EJB的最充分理由。

- **为了基于角色的安全而使用EJB声明性支持**

J2EE既提供了程序性（Programmatic）安全，又提供了声明性（Declarative）安全。虽然运行在一个J2EE服务器中的任何代码都能找到该用户的安全角色，并据此来限制访问权限，但EJB提供了声明性的将访问权限向下限制到个别业务模型上的能力（在部署描述符中）。因此，访问权限可以在部署的时候再处理，进而避免了修改EJB代码的需要。如果我们没有使用EJB，只有程序性方法是可以使用的。

- **EJB基础结构是熟悉的**

如果使用EJB的替代方法是开发各种EJB能力的一个实质子集，EJB的使用更可取，即使我们自己的解决方案显得更简单。例如，任何有能力的J2EE开发人员都通晓多线程化的EJB方法，但却不一定知道企业内部自主开发的复杂方法，进而意味着维护成本可能会比较高。另外，从策略上说，让J2EE服务器供应商去维护复杂的基础设施代码总比内部地维护它强。

EJB是分布式应用和复杂事务管理问题的一个上佳解决方案。但是，许多应用没有遇到这些问题。EJB在这样的应用中增加了不必要的复杂性。一个EJB解决方案可以比喻为一辆卡车，而一个Web应用可以比喻为一辆小轿车。当我们需要执行一些像搬运大型对象之类的工作时，一辆卡车将比一辆小轿车高效得多。但是，当一辆卡车和一辆小轿车能够做相同的工作时，小轿车行使起来将更快、更便宜，而且驾驶起来将更容易操作、更有趣。

数据存取

在决定是否使用EJB时，数据存取技术的选择是一个需要考虑的主要因素，因为各种选择之（实体组件）只有在使用EJB时才是可以使用的。数据存取策略常常决定企业系统的性能，进而使它成为了一个至关重要的设计问题。

需要注意的是，数据源连接池的容器级支持不仅在EJB容器中是可获得的，而且在Web容器中也是可获得的。

J2EE数据存取的形式

就数据存取而言，许多J2EE开发人员显得不够灵活。下面这些假设对设计有着深远的影响，这一点是毋庸置疑的：

- 数据库之间的可移植性始终是必不可少的。
- Object/Relational（对象/关系，简称O/R）映射始终是处理关系数据库时的最佳解决方案。

笔者认为，这些问题应该根据具体情况分别加以考虑。数据库可移植性不是免费的，而且可能会导致不必要的复杂性和性能的牺牲。

在某些情况中（尤其是数据能够在映射层中被高速缓存的情况下），O/R映射是一个优秀的解决方案，但前提常常是必须把一个“域对象模型”硬塞到一个关系数据库上，同时不顾效率。在这种情况下，引进一个O/R映射层几乎没有实际价值，而且还会对性能造成伤害。从积极的一面来说，如果O/R映射解决方案在某个特定应用中非常合适，那么它们可以使开发人员免于编写数据库存取代码，进而潜在地提高工作效率。

无论我们使用什么数据存取策略，通过一个抽象层将业务逻辑与数据存取的细节分离开是最好不过的了。

人们常常以为，实体组件是在数据存取与业务逻辑之间实现这种干净分离的惟一方法，这是一个谬误。如果我们希望仍保持对实现的不同选择，数据存取与系统的其他任何一个部分没有根本性的区别。只需遵守编程到接口而不是编程到具体类的优良OO设计原理，我们就可以把数据存取细节与应用的其余部分隔离开。这种方法比使用实体组件更灵活，因为我们只受限于一个Java接口（可以用任何一项技术来实现），而不是受限于单一实现技术。

实体组件

实体组件（entity bean）是一个有效设计原理的一个有疑问实现。分离出数据存取代码是一个很好的设计原理。遗憾的是，实体组件是实现这种分离的一种重量级方法，具有很高的运行时开销。实体组件并没有将我们约束于一种特定类型的数据库，而是将我们约束于EJB容器和一种特定的O/R映射技术。

实体组件的理论基础和实际价值存在着严重的疑问。将数据存取束缚于EJB容器限制了结构上的灵活性，并使应用的测试变得更困难。关于EJB的其他优缺点，我们没有选择的余地。一旦放弃了具有远程接口的实体组件这一构思（因为它实际上是在EJB 2.0中），就没有什么理由将数据对象建模为EJB。

尽管EJB 2.0中有了增强，但实体组件仍处于设计之中。这就使得用它们来解决许多常见问题变得很困难（实体组件是一个非常基本的O/R映射标准）。它们常常会导致关系数据库的无效使用，进而产生极差的性能。

在第8章中，我们将仔细分析围绕实体组件的各种争论。

Java数据对象（JDO）

JDO是Java Community Process领导下的一个新规范，描述了一种适用于从Java对象到任一持久性存储形式的机制。JDO常常用做一种O/R映射，但不限于RDBMS。例如，JDO可能会成为RDBMS的Java访问的标准API。JDO提供了一个比实体组件更轻便的模型。大多数普通Java对象都能被持久地保存，只要它们的持久性状态被存放在它们的实例数据中。和实体组件不同，使用JDO持久地保存的对象不需要实现任何特殊接口。JDO还定义了一种查询语言，用于运行对持久性数据的查询。它还提供了各色各样的高速缓存方法，进而把选择权留给了JDO供应者。

JDO目前还不是J2EE的一部分。但是，和JDBC和JNDI一样，它似乎有可能最终成为一个必需的API。

JDO提供了实体组件的主要积极面，同时消除了大部分消极面。它与J2EE服务器事务管理集成得很充分，但又不受限于EJB甚至J2EE。缺点是JDO实现仍相当不成熟，而且由于JDO实现不提供大多数J2EE应用服务器，所以我们需要从第三方供应商那里获得J2EE应用服务器（并必须与该第三方供应商维持一种关系）。

其他O/R映射解决方案

诸如TopLink和CocoBase之类的主流O/R映射产品比JDO更成熟。这些产品可以用在J2EE应用内的任何地方，并提供先进的高性能O/R映射，但代价是依赖于第三方供应商，并要花费相当于J2EE应用服务器的许可费用。在凡是存在自然O/R映射的地方，这些解决方案都可能是十分有效的。

JDBC

J2EE正统观念或多或少地认为，JDBC和SQL（即便不是RDBMS本身）都是有害的，而且J2EE应该尽可能少地利用它们。笔者认为这是误导。RDBMS在这里有存在的必要，而且这不是一件那么坏的事情。

尽管JDBC API是低级的，使用起来有点麻烦，但是较高级的库（比如我们为本书的示例应用所使用的那些库）使用起来就轻松得多。JDBC最好是在没有任何自然O/R映射或我们需要使用像存储过程那样的高级RDBMS特性时使用。如果使用得当，JDBC会提供极佳的性能。当数据能够在一个O/R映射层中得到自然的高速缓存时，JDBC不适用。

状态管理

对于J2EE设计师来说，另一个至关重要的决策是如何维护服务器端状态。这将决定一个应用在服务器聚类中如何表现（聚类是可缩放性的关键），以及我们应该使用什么J2EE构件类型。

决定一个应用是否需要服务器端状态是很重要的。当一个应用运行在单个服务器上时，维护服务器端状态不是问题，但是当一个应用必须依靠运行在一个聚类中来才能缩放时，服务器端状态必须在该聚类中得到复制，以允许故障切换和避免服务器亲和力（Server Affinity）的问题（在服务器亲和力中，一个客户变成只连接到一个特定服务器）。好的应用服务器提供先进的复制服务，但这必然会影响性能和可缩放性。

如果确实需要服务器端状态，应该最小化我们所持有的状态数量。

没有维护服务器端状态的应用比维护了服务器端状态的应用更可缩放，并且在一个聚类环境中部署起来更简单。

如果一个应用需要维护服务器端状态，我们需要选择将它保存在何处。这部分取决于我们必须保存哪种状态：用户接口状态（比如Web应用中用户会话的状态）、业务对象状态

或者这两种状态。分布式EJB解决方案利用无状态会话组件（Stateless Session Bean）产生最大的可缩放性，而与Web层中保存的状态无关。

J2EE为Web应用中的状态管理提供了两个标准选项：由Web容器管理的HTTP会话对象，以及有状态会话EJB。如果独立客户需要集中的状态管理，它们必须依靠有状态会话（Stateful Session Bean），这也是它们最好由EJB体系结构来支持的另一个原因。令人惊奇的是，有状态会话EJB未必是这两个选项中最可靠的（我们将在第10章中讨论这方面的问题），而且对状态管理的需要未必预示着EJB的使用。

J2EE体系结构

在讨论了J2EE设计中的一些高层次问题之后，现在该来看一看J2EE应用的几个可选体系结构。

常见概念

首先，让我们来看一看所有J2EE体系结构都共有的几个概念。

J2EE应用中的体系结构层

下面要讨论的每个体系结构都含有3个主要层，尽管有些体系结构在中间层内引入了另外的划分。

经验已经证明了将企业级系统明确地划分成多个层的价值。这确保了责任的明确划分。

J2EE的3层体系结构是各类系统中的经验结晶。具有3个或3个以上层的系统已经证明比其内没有中间层的客户-服务器系统具有更大的可缩放和灵活性。

在一个设计完备的多层系统中，每一层应该只依赖于它下面的那一层。例如，对数据库的更改不应该要求对Web接口的更改。

每一层所特有的东西应该向其他层隐藏起来。例如，Web应用中的Web层只应该依赖于服务器小程序API，而中间层只应该依赖于JDBC之类的企业资源API。这两个原则确保了应用修改起来容易，同时修改又不级联到其他层。

下面依次来看一看典型J2EE体系结构的每一层。

企业信息系统（EIS）层

这一层有时也叫做综合层（Integration Tier），由J2EE应用完成其工作所必须访问的企业资源所组成。这些资源包括数据库管理系统（DBMS）和遗留的主机应用。EIS层资源通常是事务性的，EIS层位于J2EE服务器的控制之外，尽管该服务器的确以一种标准方式管理事务和连接池。

J2EE设计师对EIS层的设计与部署的控制将是变化的，视该项目的性质（现有服务的绿色场或集成度）而定。如果该项目包含现有服务的集成，EIS层资源可能会影响中间层的实现。

J2EE为与EIS层资源的接口提供了强有力的能力，比如访问关系数据库的JDBC API、

访问目录服务器的JNDI以及允许连接其他EIS系统的Java Connector Architecture (Java连接器体系结构，简称JCA)。J2EE服务器负责建立连往EIS资源的连接池、横跨资源上的事务管理以及保证J2EE应用不危及EIS系统的安全。

中间层

这一层含有应用的业务对象，并调停对EIS层资源的访问。中间层构件主要从事管理和连接建池之类的J2EE容器服务中受益。中间层构件独立于选定的用户接口。如果使用了EJB，我们把中间层分离成两层：EJB以及使用这些EJB来支持该接口的对象。但是，这种分离不是保证一个干净中间层所必需的。

用户接口（UI）层

这一层将中间层业务对象暴露给用户。在Web应用中，UI层由服务器小程序、服务器小程序所使用的助手类以及诸如JSP页之类的视图构件所组成。为了清楚起见，我们在讨论Web应用时将把UI层称做“Web层”。

业务接口的重要性

许多人将EJB看做J2EE应用的核心。从J2EE的EJB中心论角度看，会话EJB将暴露应用的业务逻辑，而其他对象（比如Business Delegate J2EE设计模式中的Web层“业务委托”对象）将由它们与EJB的关系来确定。但是，这种假设将一种技术（EJB）抬高到了OO设计考虑之上。

EJB不是在J2EE应用中实现中间层的惟一技术。

正式业务接口层的概念体现了一种好的习惯，不管是使用了EJB，我们都应该使用这个概念。在下面将要讨论的所有体系结构中，业务接口层都由客户（比如UI层）直接使用的中间层接口所组成。业务接口层为普通Java接口中的中间层定义了联系人；因此，EJB就是一个实现策略。如果我们没有使用EJB，业务接口的实现将是运行在J2EE Web容器中的普通Java对象。当使用了EJB时，业务接口的实现将隐藏掉与EJB层的交互。

一定要设计到Java接口，而不要设计到具体类，也不要设计到技术。

下面来看一看满足不同需求的4种J2EE体系结构。

非分布式体系结构

下面的这些体系结构适合Web应用。它们可以把所有应用构件只运行在单个JVM中。这使它们变得简单而有效，但限制了部署的灵活性。

具有业务构件接口的Web应用

在大多数情况下，J2EE用来构造Web应用。因此，同一个J2EE Web容器可以提供许多应用所需要的整个基础结构。

和EJB一样，J2EE Web应用实际上享有对企业API的相同访问权。它们受益于J2EE服务器的事务管理和连接池能力，并可以使用JMS、JDBC和Java Connector API之类的企业服务。除实体组件之外的所有数据存取技术都是可以使用的。

Web应用的Web层和中间层运行在同一个JVM中。但是，在逻辑上使它们保持不同是极其重要的。Web应用中的主要设计风险是UI构件与业务逻辑构件之间的责任模糊不清。

业务接口层将由普通Java类所实现的Java接口来组成。这是一个简单而又可缩放的体系结构，并且能满足大多数应用的需要。

图1.1举例说明了这一设计。水平虚线表示该应用的3个层之间的划分。

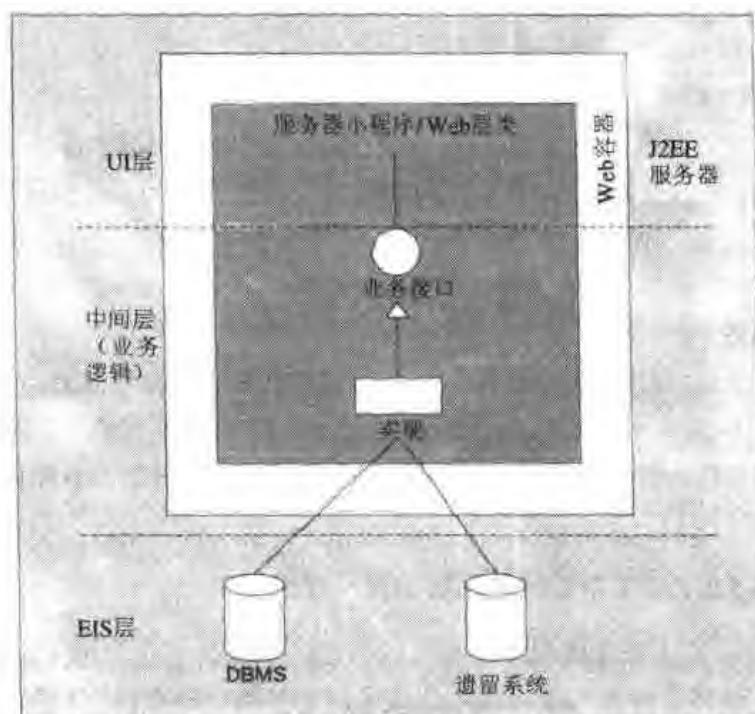


图1.1

长处

这种体系结构具有下列优点：

- 简单性。这通常是Web应用的最简单结构。但是，如果事务管理或线程化问题要求开发过分复杂的代码，使用EJB可能将更简单。
- 速度。这样的体系结构遇到了来自J2EE服务器的最小系统开销。
- OO设计不会被J2EE构件问题（比如调用EJB的影响）所妨碍。
- 容易测试。如果设计合理，无需Web层就能够对业务接口进行测试。
- 我们可以发挥服务器的事务支持。
- 缩放性很好。如果Web接口是无状态的，则根本不需要来自容器的聚类支持。但是，Web应用可以通过使用服务器支持会话状态复制来分布。

弱点

应该注意下列这些缺点：

- 这种体系结构只支持一个Web接口。例如，它不能支持独立的GUI客户（中间层和这个Web接口在同一个JVM中）。但是，正如我们稍后将会看到的，可以增加一个Web服务层。
- 整个应用仅运行在单个JVM中。虽然这提高了性能，但我们无法将构件自由地分配给不同的物理服务器。
- 这种体系结构不能使用EJB容器事务支持。我们将需要在应用代码中创建和管理事务。
- 服务器没有提供对并发编程的支持。我们必须亲自处理线程化问题，或使用一个解决常见问题的类库，比如util.concurrent。
- 将实体组件用于数据存取是不可能的，但可以证明的是，这根本不是什么损失。

访问本地EJB的Web应用

Servlet 2.3规范(SRV.9.11)可从<http://java.sun.com/products/servlet/download.html>站点上获得。如果一个应用被部署在一个集成的J2EE应用服务器中且该服务器运行在单个JVM中，该规范通过本地接口来保证EJB的Web层对象访问。这使我们既能从一个EJB容器中得到好处，又不至于招致过度的复杂性或把我们的应用变成分布式的。

在这种体系结构中，Web层与刚讨论过的Web应用体系结构的Web层相同。业务接口也是相同的，这两种体系结构的不同之处从它们的实现（面对EJB层）开始。因此，中间层被划分成了两部分（运行在Web容器中的业务接口和EJB），但这两部分运行在同一个JVM中。

有两种方法可以用来实现业务接口：

- 代理方法。在这种方法中，一个本地EJB直接实现业务接口，而Web容器代码被赋予一个对该EJB的本地接口的引用，同时无需处理必不可少的JNDI查找。
- 业务委托方法。在这种方法中，业务接口的Web容器实现明确地托付给相应的EJB。这具有允许高速缓存和允许故障操作在适当地点被重试的优点。

我们无需担心上述任一情况中的java.rmi.RemoteException捕获。传输错误不会出现。

在这种体系结构中，和通过EJB来暴露一个远程接口的体系结构不同，EJB的使用仅仅是这种体系结构的一个实现选择而已，而不是一个基本特征。不用改变总体设计，也不用EJB，就可以实现任何一个业务接口。

这是一个有效的折衷体系结构，可以由EJB 2.0规范中的各种增强来构成，见图1.2所示。

长处

这种体系结构具有如下这些优点：

- 它没有分布式EJB应用那么复杂。
- EJB使用不更改应用的基本设计。在这种体系结构中，只使这样一些对象成为EJB：它们需要一个EJB容器的那些服务。
- EJB使用只强加相当小的性能开销，因为没有远程方法调用或串行化。
- 它提供EJB容器事务与线程管理的各种好处。
- 如果需要，它允许我们使用实体组件。

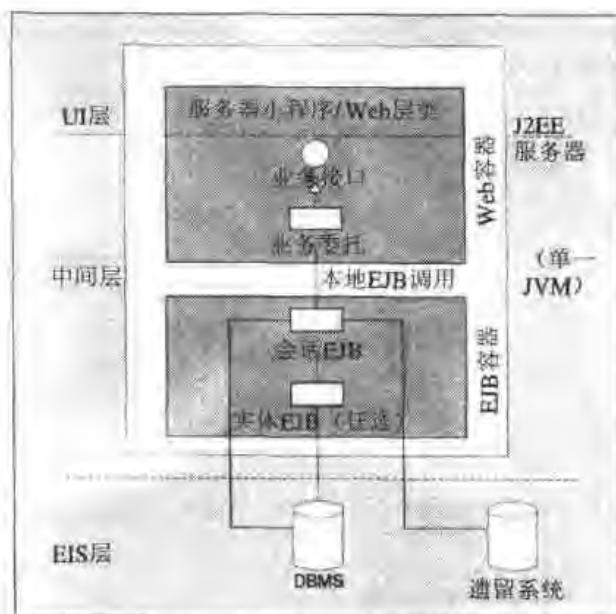


图1.2

弱点

这种体系结构的缺点有如下这些：

- 它比纯Web应用更复杂。例如，它遇到EJB部署和类装入复杂性。
- 它仍不能支持除一个Web接口之外的客户，除非我们添加一个Web服务层。
- 整个应用仍运行在单个JVM中，这意味着所有构件都必须运行在同一台物理服务器上。
- 具有本地接口的EJB测试起来很困难。我们需要在J2EE服务器内运行测试案例（比如用服务器小程序）。
- 作为使用EJB的结果，仍存在一些调整对象设计的诱惑。即使含有本地接口，EJB调用仍慢于普通的方法调用，而且这可能会诱惑我们修改业务对象的自然粒度。

有时，我们可能会决定把EJB引进到一个没有使用它的体系结构中。这可能是由“做可能管用的最简单事情”的XP方法所造成的。例如，最初的需求可能没有证明由EJB引入的复杂性是值得的，但后来增加的需求可能会提出使用EJB。

如果采用上面描述的业务构件接口方法，引进具有本地接口的EJB将不会引起问题。可以简单地选择应该被实现成具有本地接口的代理EJB的那些业务构件接口。

引进具有远程接口的EJB可能有较大疑问，因为这不仅仅是一个引进EJB的问题，而且也是一个从根本上改变了应用的性质的问题。例如，可能需要使业务接口粒度变得更粗，以避免“罗嗦的”调用和实现足够的性能。我们还可能需要把所有业务逻辑实现转移到EJB容器内部。

分布式体系结构

下面这两种体系结构除了支持Web应用之外，还支持远程客户。

具有远程EJB的分布式应用

这种体系结构被广泛地看做“经典的”J2EE体系结构。它提供了这样一种能力：通过给EJB及使用EJB的构件（比如Web构件）使用不同的JVM来物理和逻辑地划分中间层。这是一个复杂的体系结构，并具有显著的性能开销（参见图1.3）。

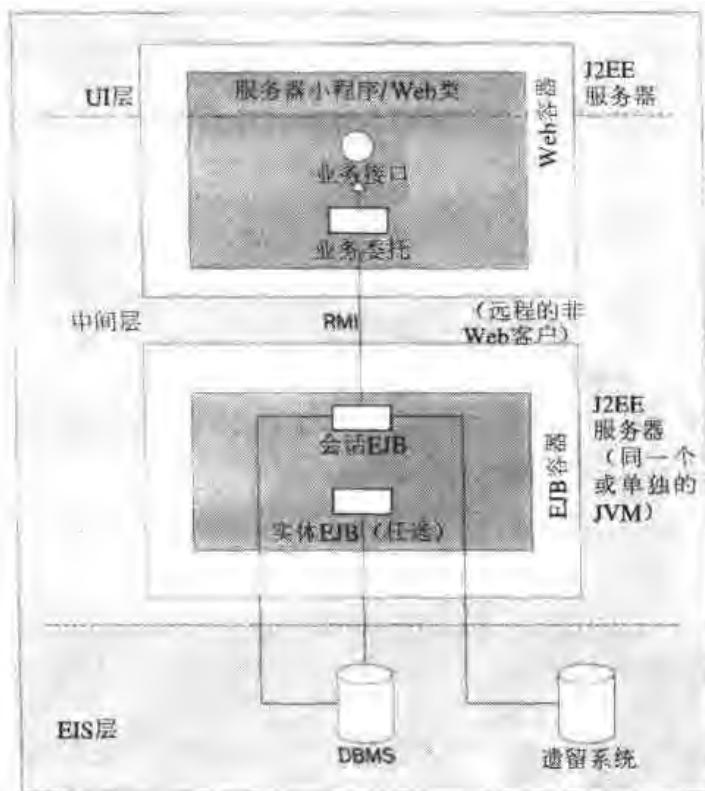


图1.3

虽然图1.3描述了一个Web应用，但该体系结构可以支持任一J2EE客户类型。它特别符合独立客户应用的需要。

该体系结构在UI层（或者说其他远程客户）与业务对象之间使用RMI，而这些业务对象被暴露为EJB（RMI通信的细节由EJB容器来隐藏，但我们仍需要处理使用它所带来的影响）。这使远程调用变成了一个主要的性能决定要素和一个核心的设计考虑因素。我们必须尽量最大限度地减少远程调用的数量（避免“罗嗦的”调用）。在EJB与EJB客户层之间传递的所有对象都必须是可串行化的，而且我们必须处理更复杂的错误处理需求。

该体系结构中的Web层和上面所讨论的那些结构中的Web层相同。但是，业务接口的实现将处理对（可能是远程）EJB容器中的EJB的远程访问。在已讨论过的用于本地EJB的两种连通性方法（代理和业务委托）中，只有业务委托在这里是有用的，因为EJB远程接口上的所有方法都抛出javax.rmi.RemoteException。这是一个已检查异常。除非我们使用一个业务委托来联系EJB，并把RMI异常包装为致命的运行时异常或应用异常，否则RemoteException将需要在UI层代码中被捕获。这把它不正确地束缚到了一个EJB实现上。

EJB层将单独负责与EIS层资源的通信，而且应该含有应用的业务逻辑。

长处

这种通信结构具有如下这些特有的优点：

- 它可以通过提供一个共享的中间层来支持所有J2EE客户类型。
- 它允许应用构件在不同物理服务器上的分布。如果EJB层是无状态的，这个特点特别管用，进而允许使用无状态的会话EJB。含有有状态UI层和无状态中间层的应用将会从这种部署选择中获得最大的好处，而且将会给J2EE应用实现尽可能大的可缩放性。

弱点

这种体系结构的弱点有如下这些：

- 这是我们已讨论过的最复杂的方法，如果这种复杂性确定是业务需求的合理要求，很可能会导致整个项目周期内的资源浪费，并为故障提供一个滋生地。
- 它影响性能。远程方法调用会比使用引用的本地调用慢数百倍，总体性能方面的影响结果取决于必需的远程调用数量。
- 分布式应用的测试和调试变得很困难。
- 所有业务构件都必须运行在EJB容器中。虽然这为远程客户提供了一个综合性接口，但如果EJB不能用来解决业务需求所引起的每个问题，这是有问题的。例如，如果Singleton设计模式完全适用，用EJB满意地实现起来将会很困难。
- OO设计被RMI的集中使用所严重阻碍。
- 异常处理在分布式系统中变得更复杂。我们除了必须考虑应用故障外，还必须兼顾传输故障。

当使用这种体系结构时，千万不要破坏它。Sun Java Center的“Fast Lane Reader”J2EE模式 (http://java.sun.com/blueprints/patterns/j2ee_patterns/fast_lane_reader/) 主张从Web层中执行只读JDBC访问，以便最小化通过EJB层进行调用的系统开销。这违背了每个层只应该跟直接位于它下面的那些层进行通信的原则，也降低了分布式体系结构的一个重要优点：部署灵活性。现在，运行Web层的服务器必须能够访问数据库，而这会使特殊的防火墙规则成为必需之物。

即使我们使用了远程接口，如果EJB及使用EJB的构件被放在了一起，那么大多数J2EE服务器仍能优化远程调用并替换按引用的调用。这可以极大地减少使用具有远程接口的EJB所造成的性能影响，但无法消除远程语义所引入的有害影响。这种配置更改了应用的语义。要想让这种配置得到使用，关键是保证EJB支持本地调用（按引用）和远程调用（按值）。否则，按引用的调用者可能会修改要传递给其他调用者的对象，进而产生严重的后果。

不要因为使用了具有远程接口的EJB导致一个应用变成分布式的，除非业务需求明确指出需要一个分布式体系结构。

暴露Web服务接口的Web应用

Web服务标准（比如SOAP）的出现意味着J2EE应用不再束缚于使用RMI和EJB来支持远程客户。图1.4中的这种体系结构可以支持像Microsoft应用那样的非J2EE客户。

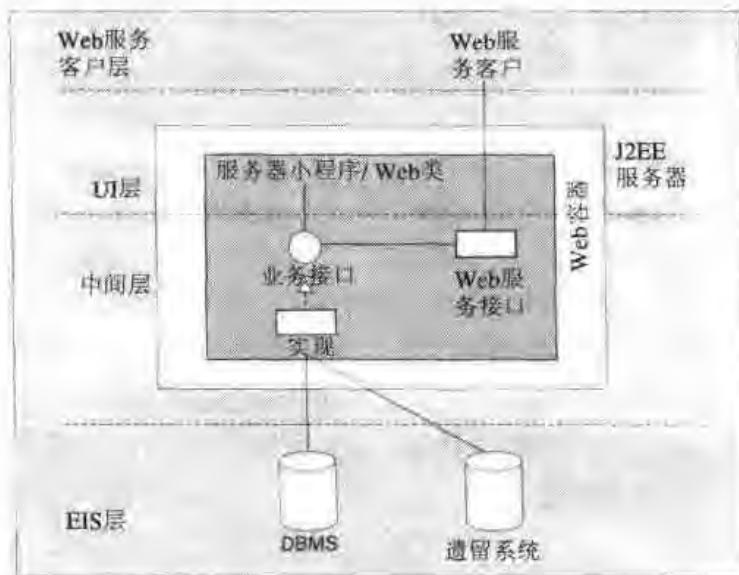


图1.4

该体系结构给任一J2EE应用增加了一个暴露Web服务的对象层和一个实现必要协议的传输层。Web服务组或者作为一个传统Web接口运行在同一个Web容器里，或者说Web服务接口可能是该应用所暴露的惟一接口。

对象层可能只是该应用的业务接口。传输层的细节将是变化的（J2EE目前没有标准化对Web服务的支持），但像WebLogic那样的J2EE服务器使实现变得很容易。像Apache Axis那样的第三方产品在任何J2EE服务器上都提供简单的SOAP Web服务。例如，Axis提供一个能被添加到任何Web应用上并能够出版任一应用类（包括生成WSDL）的服务器小程序作为一个Web服务。这是一个非常简单的操作。

该体系结构不同于刚刚描述的分布式EJB体系结构，这种差别不仅反映在远程访问协议方面，而且在如下方面也有体现：远程接口一般被添加到一个现有应用上，而不是被内建到该应用的结构中。

图1.4的相反部分描述了一个没有EJB的Web应用正暴露出的Web服务。我们可以使用这种方法给刚描述过的3种体系结构中的任何一种添加Web服务（尤其是第一种或第二种体系结构）。Web服务远程访问的使用消除了使用具有远程接口的EJB的一大理由。

像SOAP那样的Web服务协议最终将会支持像RMI那样的平台专用协议是可能的。这倒有点像是Microsoft的宗旨，因为这背离了它的专有DCOM远程访问技术。

长处

这种体系结构具有如下优点：

- SOAP比RMI/IOP更开放。这种体系结构可以支持除J2EE客户之外的客户，比如VB应用。
- 对于企业来说，暴露Web服务接口可能比暴露RMI/IOP接口具有更大的益处。
- Web服务传输协议运行在HTTP上面，并且比RMI更防火墙友好、更易于人类理解。
- 远程访问的传递是一个不要求总体体系结构的外接式附件。例如，可以根据实现一个

应用的最佳方法来选择是否使用EJB，而不是根据该应用将被如何访问来选择。

弱点

这种体系结构具有如下弱点：

- 性能。通过一个像SOAP那样的XML协议来传递对象的系统开销可能会高于RMI的系统开销。
- 如果所有客户类型都使用J2EE技术，这种体系结构就劣于分布式EJB体系结构。在这种情况下，几乎没有理由使用一个平台独立的远程协议。
- 复杂对象的编组（Marshaling）和反编组（Unmarshaling）可能需要自定义编程。对象将以XML格式通过通信线路来传递。我们可能必须在Java对象与XML之间做转换。
- 即使SOAP现在已得到广泛认可，但目前仍没有可以与EJB规范相比的Java Web服务支持的标准化。

EJB应用也可以暴露Web服务。WebLog和其他一些服务器允许EJB作为Web服务的直接暴露。

Web层设计

虽然J2EE考虑到了不同的客户类型，但Web应用实际上是最重要的客户类型。当今，甚至连许多内联网应用都使用了Web接口。

前面讨论的4种体系结构在它们的Web接口设计方面没有差别，但在它们实现和访问业务逻辑方面存在差别。Web接口设计的下列讨论适用于所有这4种体系结构。

Web层负责把用户能理解的表示和显示转换成应用的业务对象能理解的操作，或者进行相反的转换。

重要的是，Web层是一个位于中间层业务接口之上的独立层。这保证了Web层能在不更改业务对象的情况下被修改，以及业务对象能在不引用Web层的情况下被测试。

在Web应用中，Web层有可能是最常遭到修改的部分。许多因素（比如商标更改、高级主管人员的反复无常、用户反馈信息或业务策略方面的变化）都可能会驱使一个Web站点的外观和感觉方面的显著改变。这使Web层设计变成了一个重大挑战。

保证Web层易对变化做出反应的关键，是保证在业务对象的表示与逻辑控制和访问之间有一个明确的分离。这意味着确保每个构件要么只关注置标生成，要么只关注用户动作的处理和跟业务对象的交互。

Model View Controller (MVC) 结构模式

要想在表示与逻辑之间实现分离，一种已得到证明的方法是给Web应用使用MVC体系结构模式。MVC体系结构模式首先是为Smalltalk用户接口而正式公布的，并且一直是最成功的OO结构模式之一。它也是Java的Swing接口包的基础。MVC把构造一个用户接口所需要的构件划分成3种对象，进而确保我们所关注的明确分离：

- 模型数据或应用对象——不含有与表示相关的任何代码。

- 视图对象执行模式数据的屏幕表示。
- 控制器对象对用户输入做出反应，并据此更新该模型。

我们将在第12章中详细讨论Web层的设计，但现在先来看一看怎样在J2EE中实现MVC模式的另一种形式。

在J2EE应用中，Web层将建立在服务器小程序之上。JSP页和其他表示技术（如XSLT）将用来再现内容。

一种典型的实现包括使用一个标准的控制器服务器小程序作为进入整个应用或部分应用的统一入口点。这个入口点从应用特有的多个请求控制器中选择一个来处理请求（映射将以配置形式在Java代码外部进行定义）。这个控制器服务器小程序实际上是控制器的一个控制器。目前还没有适用于笔者所谓的“请求控制器”的标准术语——Struts Web应用框架把这样的委托称做动作（Action）。请求控制器或者说动作对应于MVC三元组中的控制器。它本身不产生任何输出，但处理请求，启动业务操作，最佳地操纵会话和应用状态，以及重定向请求给适当的视图（在该视图中，它暴露一个从它的处理中产生的模型）。

模型、视图以及控制器对象按如下所述映射到J2EE构件上：

- 模型就是一个暴露数据的Java组件。模型对象不应该知道如何从业务对象中检索数据，但应该暴露能让其状态由控制器来初始化的组件属性。因此，视图的再现将永远不会因检索数据故障之类的原因而失败。这极大地简化了表示层中的错误处理（使用XML文档作为模型也是可能的）。
- 视图就是一个用来显示模型中数据的构件。视图应该从不执行业务逻辑，或者获取模型中它可获得的那些数据之外的数据。J2EE系统中的视图构件常常是JSP页。这种结构模式中的每个视图类似于一个简单联系人的一种实现（“显示对象的某一个集合”）。因此，每个视图都能用另一个不同地显示同一个模型的视图来替换，同时又不需要更改该应用的行为方式。
- 控制器就是一个处理进入请求，与业务对象进行交互，建立模型，然后把每个请求转发给相应视图的Java类。请求控制器不实现业务逻辑（这将会侵犯中间层的责任）。

每种构件类型都有各自的长处：Java类（请求控制器）处理逻辑，不处理表示，而JSP页只关心生成置标。

图1.5举例说明了控制流程。控制器服务器小程序将是一个标准类，由Struts之类的框架提供（几乎没有必要亲自实现一个MVC框架，因为有许多作为开放源的实现可供我们使用）。请求控制器属于应用的UI层，并使用业务接口（该接口在上述4种体系结构的每一种中属于中间层）。视图可以是JSP页。

这个序列图表的背面描述了Command设计模式，该模式把针对一个子系统的请求封装成对象。在J2EE Web应用中，Command设计模式有利于Web层与中间层之间的干净连通性（通过非Web特有的命令对象）。在本例中，命令对象被创建用来封装HTTP请求中所包含的信息，但这些相同的对象也可以和其他用户接口一起使用，同时对业务对象又不产生任何影响。

阅读过“Core J2EE Patterns”的读者将会注意到，笔者已经描述过Service-to-Worker表示层模式。笔者不推荐Dispatcher View模式，因为该模式允许数据检索由视图来启动。我们将在第12章中讨论这种方法的缺点。

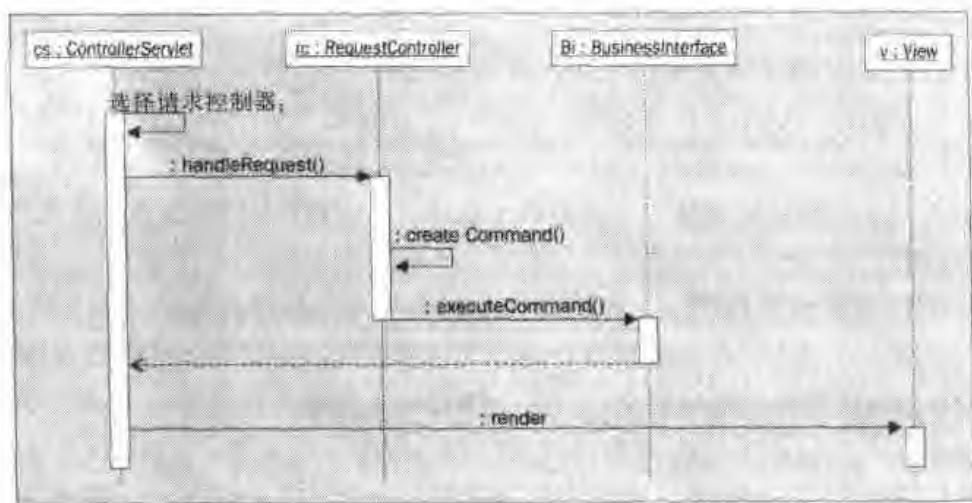


图1.5

使用MVC方法建立起来的系统具有较好的灵活性和扩展性。由于表示层与逻辑层是分开的，所以对应用程序的表示层进行重大改变而又不影响其功能也就成为了可能。我们甚至可以从一种视图技术转向另一种（例如，从JSP转向XSLT）而不必对应用的表示层进行大的改变。

在Web层中，应该使用MVC体系结构模式。应该使用一个像Struts那样的Web应用框架来提供该MVC模式的一个标准实现，以便最大限度地减少你将需要编写的应用代码量。

Web层与业务对象之间的连通性

将Web层与业务对象干净利落地分离开是至关重要的。虽然MVC模式在Web层控制器与表示对象（保存格式化信息）之间产生了一个干净的分离，但是将Web层控制器（仍束缚于Servlet API）与业务对象（是接口独立的）分离开决非不重要。

在以后的章节中，我们将讨论支持这种分离的基础结构。现在，让我们来看一看下面这些关键目标：

- Web层构件决不应该实现业务逻辑本身。如果这个目标得到满足，在没有Web层的情况下测试业务逻辑是可能的（进而使测试变得更容易）。满足这个目标也保证了业务逻辑不被Web层上发生的修改所破坏。
- 一个标准基础结构应该使Web层能轻易地访问业务对象，并仍允许业务对象在没有Web层的情况下被容易地创建和测试。

在一个设计完备的J2EE Web应用中，Web层将非常细薄。它将只含有调用中间层业务接口和显示调用结果所必需的代码。

设计可移植的应用

在用一种可移植的方式来标准化中间件概念方面，J2EE做了一项非常出色的工作。Java

本身运行在企业应用中所使用的几乎所有操作系统上。然而，仅有J2EE规范不会保证基础结构将帮助解决所有的现实问题（不过，J2EE 1.3填补了这个空白）。

和.NET的拥护者一样认为J2EE可移植性毫无意义是错误的。不过，确实需要考虑保证我们能够保持尽可能大的可移植性，即便我们决定或者被迫利用初始目标平台的各种特定能力。这可以用3种方式来实现：

- 编写用J2EE规范编译的代码。
- 熟悉标准J2EE基础结构的各种能力，并在存在满意的标准替代方案时避免使用平台特有的解决方案。
- 利用松耦合来隔离出平台特有特性的使用，以保证该应用的设计（即使不是它的全部代码）保持可移植。

通过使用一个抽象层（Abstraction layer），可以利用松耦合把该应用的其余部分与必须用一种平台特有方式来实现的任何部分隔离开。抽象层是指一个（或一组）其本身就独立于平台的接口。这些接口可以针对目标平台来实现，以利用目标平台的特有能力；如果这些接口移植到另一种平台是不可避免的，还可以在不影响应用其余部分的情况下不同地实现它们。

我们必须区分开实现可移植性（“代码在任一应用服务器上无需修改即可运行”）与设计可移植性（“应用在任一应用服务器上都能正确而有效地工作，只要少量被明确标识出的接口得到了重新实现”）。总的实现可移植性可以用J2EE来实现，但可能不是一个有现实意义的结果，甚至不是一个非常值得做的结果。设计可移植性是可实现的，而且贡献了可移植性的大部分商业价值。即使业务需求没有明确指出要求可移植性，设计可移植性仍应该是遵守好的OO设计原理的自然结果。

在现实生活中，可能需要使用我们的应用服务器所提供的供应商特有的扩展，或使用数据库之类的资源所提供的非可移植扩展。有时，根本不存在实现必需功能的其他方法。有时，性能需求明确指出要求一个平台特有的解决方案。此类情况要求一种注重实效的方法。J2EE设计师和开发人员毕竟是被雇来实现满足或超额满足业务需求的高性价比解决方案的，而不是被雇来编写“纯”J2EE应用的。

下面来考虑几个可能会迫使我们使用供应商特有扩展的典型问题：

- **EJB QL的限制**

如果使用具有CMP的EJB 2.0实体组件，我们可能就会遇到EJB QL的限制。例如，它不提供合计函数或ORDER BY子句。在这样的情况下，我们将需要考虑使用数据库特有的功能度或供应商特有的增强（例如，WebLogic提供EJB QL的扩展）。

- **对EIS层构件的专有特性的访问**

可以十分肯定地说，J2EE规范没有试图去解决要求不同数据源的问题。但是，有些时候，我们必须使用这类资源的不可移植特性。例如，当底层数据存储是一个关系数据库时，根本就不存在一种使用CMP实体组件来执行批量更新或调用存储过程的标准方法。然而，由于性能的缘故，这些操作可能又是必不可少的。

在这类情况下，保住可移植性的最佳方法是使用一个可移植的抽象层和一个针对目标

平台的具体实现。在特殊情况下，比如当出现了EJB QL的各种限制时，我们可以在有移植该应用的任何需要之前，先赌一赌已有的标准支持，即使不成功，其他的容器供应商也将会提供类似的专有扩展。

不要拒绝供应商特有的增强或EIS资源特有的功能度，它们可能会有明显的好处。还要保证在不危及设计可移植性的情况下使用它们。

应用可移植性是优良设计的自然结果。如果我们只局限于供应商特有的功能度，并保证通过与供应商无关的接口访问它，则可以既充分利用每个目标平台的宝贵特性，又不危及总设计的可移植性。

小结

本章从一个较高的层面上探讨了J2EE体系结构。我们已经讨论了如下内容：

- 采用分布式体系结构的优缺点。与所有构件都被集中布置在同一处的应用相比，分布式应用更复杂，更难以实现、测试和维护。分布式应用也会显出令人失望的性能，除非经过了仔细设计。不过，分布式应用在某些情况下会更坚固、更具可缩放性，而且分布式体系结构可能是满足某些业务需求的惟一方法。因此，决定是否使用分布式体系结构是一项应该在项目周期的早期做出的重要决策。最好是避免使用分布式体系结构，除非它提供真正的好处。
- EJB 2.0规范中的各种增强和Web服务的出现对J2EE体系结构的影响。EJB 2.0允许我们使用按引用调用，并通过本地接口来访问运行在同一个JVM中的EJB。因此，EJB 2.0允许我们使用EJB，同时又不逼迫我们采用一种分布式体系结构。Web服务协议使我们不必使用EJB也能支持远程客户，因为基于RMI的远程访问协议不再是针对J2EE应用的惟一选择。在远程访问的这些方法中，哪种方法最理想取决于该应用必须支持的任何远程客户的需求。
- 决定何时使用EJB。EJB是一种能够很好地解决某些问题的复杂而强有力的技术，但在许多应用中是不适用的。尤其是，我们不应该让使用EJB的要求使一个应用变成分布式的（在分布式不是必需的时候）。
- J2EE应用中的一些主要数据存取问题，特别是下列这些问题：
 - 数据库可移植性。虽然J2EE能够实现目标数据库之间的可移植性，但这未必总能产生商业价值。因此，数据库可移植性可能证明不了性能或效率方面的牺牲是合理的。
 - 主要数据存取策略。本章讨论了如何在O/R映射策略（如实体组件和JDO）与面向SQL且基于JDBC的持久性之间做出选择。给应用选择正确的数据存取策略会对性能和实现的容易程度产生巨大的影响。
 - 通过一个抽象层从业务对象中隐藏掉数据存取细节的重要性。如果这个抽象层将由普通Java接口组成，我们可以任意使用任何一种数据存取策略，并能够利用任何目标数据库的专有能力，同时又不危及总体设计的可移植性。

· 分层体系结构的重要性，在分层体系结构中，每个体系结构层只依赖于它紧下面的那一层。这保证了关注点的干净分离，也保证了对一层的更改不级联到其他层。

· 4种J2EE体系结构及其优缺点：

· **具有业务构件接口的Web应用**

这是一种简单、可行的体系结构，并能满足许多项目的需求。虽然它不使用EJB，但仍在业务对象与Web层构件之间提供了一种干净的分离。一个业务构件接口层把该应用的所有业务逻辑都暴露给Web层构件。

· **访问本地EJB的Web应用**

这是一个稍微复杂一点的体系结构，并允许我们使用EJB容器服务来提供线程和事务管理，同时又不会使应用变成分布式的或使远程方法调用的性能开销遭受损失。

· **具有远程EJB的分布式应用**

这是一个明显复杂的体系结构，完全能满足远程J2EE技术客户的需求。该体系结构在某些情况下比其他体系结构更具有可缩放性、更坚固。但是，它的实现、维护和测试比我们已经讨论过的各种较简单体系结构更困难，而且与满足相同需要的非分布式体系结构相比，它通常提供较差的性能。

· **暴露Web服务接口的Web应用**

该体系结构允许我们通过给一个本质上不是分布式的应用添加一个Web服务层来支持非J2EE技术的客户。该体系结构一般是上面所讨论的第一种或第二种体系结构的另一种形式。

· Web层的设计问题以及使用MVC体系结构模式的重要性。在Web层内，表示与逻辑之间的分离对实现能够响应可变表示需求的可维护Web应用来说是至关重要的。也很重要的另外两点是：Web层应该是J2EE应用的业务接口上面的一个薄层，以及应用中应该有尽可能少的部分依赖于Servlet API。

优良的OO设计原理是有效体系结构的基础。J2EE技术应该在追求一个有效的对象模型时才使用，而且不应该强迫使用对象模型本身。

第2章 J2EE项目的选择与风险

企业级软件项目通常很大，成本也很高。它们面对的挑战可能会包括：

- 集成一个组织内的资源。企业应用软件可能需要使用多种技术来访问多个数据源和遗留系统。多种技术的集成本身就是有风险的。
- 处理像应用服务器和数据库之类的复杂产品。
- 满足对服务、性能和可缩放性质量的苛刻期望值。
- 开发和维护一个大型代码库。
- 给一个组织引进新技术——例如当给一个遗留应用软件赋予Web能力时。有时，这些技术本身是新的，因而造成了它们可能是不熟悉或不成熟的风险。
- 具有不同技能集的开发团队的成功运作。
- 在一个竞争环境中实现快速的时间到市场。

J2EE平台帮助我们解决了企业软件开发的许多问题。但是，选择J2EE仅仅是许多选择中的第一个。

在开发周期的早期（编写任何代码之前）所做出的选择，对项目的成功与否以及投资的支配方式都有至关重要的影响。在项目的开始阶段所采取的决策将决定项目的开展方式。

对J2EE项目中的风险进行管理也是十分重要的，尤其是在涉及到集成的地方更是如此。

在本章中，我们除了讨论J2EE项目中的软件体系结构之外，还讨论部分最重要的选择。在这些选择中，有许多选择涉及到将随每个应用软件的需要而变化的折衷方案。通常情况下，根本没有单一的“正确”答案。我们将尽量涵盖一些主要的问题，并讨论决策制定过程，但具体的选择将留给读者自己来做出。

在有些项目中，这些选择中的许多选择将已成为一个现有J2EE策略的一部分。但是，参与此类项目的设计师和开发人员应该熟悉本章中所讨论的问题。

依据规范版本开发一个策略

要做出的第一个决策是该组织对待规范和语言版本的态度。使用最新的J2EE和J2SE特性是重要的吗？这是增强特性集与成熟功能度之间的一个折衷选择。

决定性的因素将包括：

· 该应用的性质

一小部分应用可能会从最新J2SE中获得如此之大的利益，以致没有可替代它们的方案。

· 该项目的期限

对于大型和期限长的项目来说，使策略依据新发布的特性并把应用中需要新特性的那些部分推迟到服务器支持更成熟之后再实现是完全可行的。对于短期项目来说，

快速交付一个使用成熟技术的解决方案是最重要的。需要注意的是，使一个期限长的项目依据非定型J2EE规范中的新技术是非常危险的，因此是不妥当的，即使在存在“预览”实现时。例如，EJB 2.0规范在后续的公开草案中发生了根本性的变化。

• 该组织的性质

一种“等到它工作为止”的方法可能适合对新技术非常看重的组织，但可能不适合金融机构或对可靠性要求很高的其他组织。该组织在新技术方面的技能基础在这里是另一个重要的考虑因素。

• 应用服务器之间的可移植性

越靠后采用规范版本，可移植性可能会越成问题，因为实现相同规范级别的可替代服务器的选择余地将会受到限制。

版本选择将既影响应用服务器选择，又受到应用服务器选择的影响。

在大型项目开发期间，有新的服务器或规范版本发布是可能的。当出现这种情况时，决定是否升级也很重要。如果新版本是一个故障纠正或识别版本，它可能值得做并且不太贵。如果它是一个重要升级，在采用之前应该仔细考虑，因为可能存在许多隐情，比如不同的工具需求、新的服务器故障以及不同的支持需求。跟上一个不断移动的目标是很困难的，而且J2EE仍然在快速地发展。

要想最大限度地降低风险，应该避开未经过证明的新技术，即便它们好像很有吸引力。一种安全的策略是在项目初期把J2SE和J2EE规范定为目标，因为它们得到了主流应用服务器开发商的支持，尽管有些项目可能需要考虑未来的服务器发布时间表。

选择应用服务器

选择一个J2EE应用服务器是一个组织将要做出的、最重要的IT决策之一。采用J2EE是一个战略性决策，因此怎样制定该决策将有一个长期的影响。许多应用服务器是很昂贵的，因而对大型安装来说，需要很大数目的许可费用。不过，服务器的成本只是一部分。即使在选择一个像JBoss那样的免费服务器时，我们仍承担了很大的责任。超出许可费用或许可费用之外的开支可能包括：

- 培训成本。服务器供应商通常提供产品特有的培训，而许多应用服务器复杂得足以保证为关键职员购买培训是值得的。不过，培训常常是很贵的，除非培训被提供为购买许可的一个激励品。
- 咨询费用。在熟悉应用服务器时，从服务器供应商或第三方那里购买咨询服务可能是必不可少的。
- 任何支持成本。对于某些供应商，支持需要另外一份合同，并加上许可成本。
- 熟悉新服务器时生产力的损失，以及培训或指导期间开发人员时间的损失。

了解服务器市场定位的好处是不同的服务器适合不同的需求。产品性能和价格的比较很快就会变得过时，并容易让人产生误解，而且无法以图书的形式保持最新。因此，我们在这里将只讨论做选择时需要加以考虑的问题，而不推荐或比较产品。在本节中，我们将了解如下内容：

- 何时选择应用服务器
- 如何选择应用服务器
- 谁应该选择应用服务器

需要注意的是，最终目的是整个组织在一个统一的应用服务器上的标准化。在整个软件周期内维护多个应用服务器将会是十分昂贵的，而且应该尽可能地避免。通常情况下，一个组织内使用几个不同的应用服务器反映了过去的失误或缺少一个连贯的策略。

若一个组织内运行了多个应用服务器，少数几个正当的技术原因之一是，几个应用依赖于某一个特定的特性，而这个特性在通常使用的该应用服务器中是不可获得的：例如，一个应用需要来自J2EE最新版本中的特性，因此它必须运行在一个最近发布的应用服务器上时，而大多数应用却运行在一个已证明的应用服务器上。在这种情况下，长期目标通常将是恢复到单个服务器的使用。

在许多情况下，应用服务器在项目开始以前将已被选定。例如，可能有一个关于应用服务器选择的组织级策略。在这种情况下，要尽量使用选定的服务器。不要引进另一个基于你或你的团队所喜爱的服务器，以免使组织的技术混合变得复杂。如果选定的产品确实很次，并且放弃它更有现实意义，要通过游说来改变该选择。

何时选择应用服务器

最明智的做法是在项目周期的早期选择应用服务器。这么做有下列好处：

- 不会把精力浪费在使用错误的服务器上。熟悉应用服务器（即使是同一个产品的一个新版本）需要花费时间和精力。在项目期间做这件事情所需要的次数越多，我们能够专门用于设计和开发应用的精力就越少。
- 整个项目周期都可以用来为满足组织需要的选定服务器制定一个完善的开发、部署与管理策略。J2EE应用服务器是复杂的，而且实现这方面的效率是至关重要的。
- 有更多的时间来建立与服务器供应商的牢固关系。

在服务器之间移植J2EE应用的可能性并不意味着就可以从容地选择一个J2EE服务器，明白这一点是非常重要的。上述这些考虑事项也适用于像数据库之类的其他重要企业产品。

一个选择被推迟的时间越久，浪费在熟悉上的时间就越长，利用最终选择的有价值特性的可能性就越小。

后期选择服务器的惟一好处是：该选择可能是在你的团队已经获得了J2EE经验之后做出的。如果所有团队成员都不熟悉J2EE，而J2EE本身又是一个无法接受的风险，这个好处则是一个需要加以考虑的因素。应当在项目的早期通过雇佣一些J2EE专业人员来解决这个问题。先做一个小的典型试验项目通常是让团队同时熟悉J2EE和候选服务器的一个好主意。实现已规划应用的一个“垂直程序片”也是降低许多其他风险（比如性能不足）的一种好方法。

应当尽可能早地选定应用服务器和其他重要软件产品（比如数据库）。由于“J2EE应用是可移植的”而试图推迟一个选择是一个浪费时间和金钱的秘诀。

虽然我们应该选择尽早选定应用服务器，但也应该尽量保证代码在应用服务器之间可

能的地方是可移植的。在项目生命周期的早期仅使用单个服务器，意味着我们很容易编写违反J2EE规范的代码，但又没有意识到这一点。例如，许多EJB容器允许对EJB规范的违反。但是，无法发现的违反是导致可移植性问题的一个主要原因。绝对的代码可移植性不一定能实现的原因有许多，但疏忽和粗心不应在这些原因之内。

要想避免发生对J2EE规范的此类非故意违反，一种策略是在另一个服务器上定期地部署和测试该应用（两个服务器都不能检测出违反的可能性是很小的）。但是，该策略要求资源致力于配置另一个服务器并掌握它的部署过程，而不仅仅测试代码的符合性。J2EE Reference Implementation服务器有时用于这个用途。但是，由于Reference Implementation服务器启动起来很慢，而且又是故障成灾的，因此，如果采用该策略，开放源JBoss服务器可能是一个更佳的选择。JBoss 3.0做严密的符合性检查，并提供特别简单的应用部署。

一个更佳的策略是使用J2EE Reference Implementation所携带的另外一个工具verifier（可从<http://java.sun.com/j2ee/download.html>上获得）来验证所有J2EE部署单元——WAR、EJB JAR和EAR文件的符合性。该工具提供出色而有用的输出。我们可以像下面这样使用命令来调用它：

```
cd J2EE_SDK_HOME/bin  
setenv  
Verifier -v /path/to/ejb-jar-file.jar
```

-v标志产生所有测试的详细输出。输出结果要么指出符合性（无失败的测试），要么列举出那些失败的测试。一个日志文件将被生成。下面的例子举例说明了一个失败测试上的日志输出结果：

```
Test Name: tests.ejb.session.ejbcreatemethod.EjbCreateMethodFinal  
Test Assertion: Each session Bean must have at least one non-final ejbCreate method test  
Detailed Messages:  
  
For EJB Class [ facade.ejb.JdbcReportEJB ] method [ ejbCreate ]  
Error: A final [ ejbCreate ] method was found, but [ ejbCreate ] cannot be declared as final.
```

验证可以而且应该被集成到应用生成过程中，例如使用Jakarta Ant生成工具（下文讨论）。

应当使用J2EE 1.3 Reference Implementation verifier工具定期地测试J2EE部署单元的符合性。应当把这种验证集成到应用生成过程中。

确定需求

服务器选择的过程应该是已规划好的和透明的，这一点很重要。下面来看一看其中的一些步骤和决定性的标准。

第一个目标是定义应用服务器的需求。遗憾的是，只有很少的组织做这件事。许多组织允许他们的选择过程受供应商介绍和他们对业界宣传的感觉来驱动。重要的是不受市场营销宣传的干扰而把组织的当前和可能的未来需要作为决策的依据。

关键标准包括：

- 系统的性能和可缩放性需求是什么？事实上，许多应用将永远不会成为下一个Amazon，因此可能不需要最有能力、最可缩放的服务器。
 - 该组织更喜欢一个商业产品还是一个开放源产品？
 - 可用于应用服务器选择和装备的预算是多少？
- 下面，我们将较详细地讨论其中的一些关键标准。

评估标准

我们先来看一看应该确定所用服务器的几个因素。

得到支持的规范

规范版本方面的策略（上文已讨论过）将决定支持最新J2EE规范的需求的重要程度如何（但是请记住，J2EE规范通常是向后兼容的——就J2EE 1.3来说，完全向后兼容）：

- 服务器将实现J2EE规范的什么级别（EJB、Servlet、JSP、J2EE）？
- 供应商对规范版本采取什么策略？如果供应商完全不打算跟上步伐，这可能表明该产品不是有前途的。但是，第一流产品有不同的策略。BEA保证，在每个规范版本出现时，WebLogic都支持它。另一方面，IBM传统地为WebSphere采用了一个更保守的策略，因而选择了等到每个规范版本尘埃落定时再支持它（但是，从版本4.0起，WebSphere似乎跟上了步伐）。哪种方法更好取决于业务需要。
- 服务器需要Java的哪个版本？这可能没有必要和J2EE版本相同。Java 1.4提供了显著的语言和库改进，因此，如果服务器在JDK 1.4上被支持，Java 1.4就是一个明显的选择。

Sun资源

作为J2EE规范的管理人，Sun主持着两个应用服务器评估计划：Compatibility Test Suite和ECPPerf。虽然这两个计划的存在是重要的，但正如我们将要看到的，我们不能依赖它们来指导我们的服务器选择。

下面来依次看一看这两个计划的目标和实际价值。

Compatibility Test Suite (CTS)

Sun发布了一个J2EE CTS，并在<http://java.sun.com/j2ee/compatibility.html>网页上维护一个满足其需求的产品列表。这个列表上的产品在市场上可能会作为“J2EE Compatible (J2EE兼容产品)”来销售。J2EE 1.3 CTS包括了涵盖J2EE API中所有类和模块的15 000多个测试以及J2EE服务之间的交互。

CTS是一个受欢迎的开发。在J2EE的早期年代，根本没有这样的保证，而且服务器常常不能提供J2EE核心特性的可工作实现。CTS的存在激发了开发商一丝不苟地实现J2EE规范。它增强了我们在如下方面的自信心：在服务器之间移植符合标准的应用的能力。

但是，CTS在服务器选择中的帮助是有限的，并且不能用做一个决定性的指导，原因如下：

- 不是所有的第一流竞争者都出现在了这个列表上，因为不是所有的开发商都已决定要通过认证。例如，Orion服务器就没有被包括进来（从2002年8月起），虽然它在生产中的使用比几个已通过认证的服务器更广泛。认证过程是很昂贵的，而且直到最近仍歧视开放源产品。由于这个原因，JBoss一直没能进展到认证，尽管它的目标是遵守J2EE规范（而且根据笔者的经验，它是成功的）。

需要提醒读者的是，CTS混合了良好愿望、政治策略和市场营销。关于JBoss认证问题的讨论，请参见http://www.serverworldmagazine.com/opinionw/2002/03/14_j2ee.shtml网页，对该问题做出的最后结论是：认证过程“只有资金充足的大公司才能获得”，因此是“神经错乱的”。

- 这些测试没有涵盖服务器性能和稳定性。
- 这些测试没有揭示服务器的所有构件是否可用在生产中。例如，许多已通过认证的产品从Sun的J2EE Reference Implementation中获取它们的某些功能度。该代码中的大部分决不是为生产中的使用而考虑的。
- 仅有J2EE规范不会满足一家企业的需要。服务器的增值特性（比如一个可执行的和可自定义的CMP实现）对某些应用来说是必不可少的。随着越来越多的服务器符合那些规范，这些增值特性很可能会成为选择应用服务器的主要决定因素。仅有规范符合性不是应用服务器选择的一个充分基础。
- 那些J2EE规范的一些最苛刻方面（比如CMP实体组件）对真正的应用来说具有令人怀疑的价值。因此，不是完全符合的服务器可能会提供J2EE的最有用特性的有效而又符合规范的实现。
- 这些测试没有涵盖服务器安装和应用在服务器上部署的简易性。这是一个重要问题，因为某些服务器配置起来非常困难，而其他服务器可以在几分钟内被安装完。

ECPerf

性能和可缩放性问题由ECPerf来解决，而它是来自Sun的、在JCP领导之下开发的另一个首创产品。ECPerf是一个基准应用程序，它试图复制现实问题，而性能结果的认证就是单独针对这些问题进行的。ECPerf应用程序以一个关系数据库为基础，并使用了CMP实体组件。主要的ECPerf主页是<http://java.sun.com/j2ee/ecperf/>。ECPerf结果宿主在<http://ecperf.theserverside.com/ecperf/>上。ECPerf应用程序可以通过从ECPerf主页中下载来获得。

但是，同CTS的情形一样，许多服务器没有提供ECPerf结果。自2002年8月起，只有5个服务器——WebLogic、WebSphere、Oracle、Sybase和Pramati声称提供ECPerf结果。ECPerf只对一点有用，但它适合演示某个使用了一个指定RDBMS的特定硬件配置上的性能。

例如，可以把WebLogic在一个使用了Oracle 8.1.7的HP服务器聚类上的性能与WebSphere在一个使用了DB2的IBM服务器聚类上的性能做比较。但是，如果我们需要预测WebLogic和WebSphere在使用了相同数据库的相同硬件上的性能，这种比较没有任何帮助（一般说来，硬件可用性主要被事先确定，而RDBMS在我们选择一个J2EE应用服务器之前已被选定，因此这实际上才是我们确实需要知道的东西）。另外，ECPerf结果具有有限的相关性，如果我们使用不同的应用体系结构，比如具有JDBC或JDO而不是具有实体组件的会话组件(session bean)。

澳大利亚的Commonwealth Scientific and Industrial Research Organization (CSIRO) (联邦科学与工业研究组织) 提供某些第一流应用服务器的商业性能比较, 可以从<http://www.cmis.csiro.au/adsat/j2eev2.htm>中获得。和ECPef不同的是, 这确定在相同硬件上比较了应用服务器。

除了在应用服务器比较方面使用它们之外, ECPef结果还是一个用于性能调整信息的宝贵资源。ECPef结果必须把任何修改完全文档化成默认的服务器配置, 比如JVM选项、EJB容器数量和操作系统设置。

成本

J2EE应用服务器在许可费用方面的变化是很大的——从为JBoss支付的零美元到为一流商业产品的一个大型安装支付的几十万美元, 而且在如何计算许可费用方面(比如按CPU还是按物理服务器)的变化也是很大的。

在20世纪80年代的英国电视戏剧“*Yes, Prime Minister*”中有这样一段情节, 诡计多端的公务员Humphrey Appleby爵士试图说服首相Jim Hacker不顾成本(和战略需要)而购买一种特殊类型的核导弹, 理由很简单: 因为这种核导弹是最好的。Humphrey爵士的理由最终得出吓人的短语“它是Harrods将要出售的核导弹”。在繁荣年代, 比如20世纪90年代后期的.COM狂潮, 这样的理由也极大地影响了高级管理人员。

作为一名CTO或CIO(许多公司中的一个常见角色)的Humphrey爵士将会争辩说, XYZ服务器是最昂贵的, 所以也是最好的, 而且公司不购买它将表明公司缺乏自信。他可能还会更有理有据地补充说, 市场期望他的公司在技术产品方面投入至少和竞争对手一样多的资金。董事会将可能会更怀疑一个开放源解决方案。这种基于经验的(确实具有讽刺意味的)分析被Gartner Group那样的业界分析家所证实, 他们估计, 到2001年晚些时候, 一些公司已经在他们不真正需要的高端应用服务器技术上浪费了10亿多美元(参见<http://www.nwfusion.com/news/2001/0821gartnerapp.html>)。

在2001年2月的惨淡时期, 一些公司不愿意把钱花费在软件产品上, 甚至不愿意做一项战略性投资。有这样一种趋势: 最初花费尽可能少的钱, 但任人迁移到一个更昂贵的平台, 只要这个平台是必需的。可以证明的是, 这种做法比提前多花费更昂贵。应用服务器的成本将只占项目总预算的一小部分; 开发团队的运作成本占得更多。勉强度日在顾客心目中以及在职员时间与士气上损失的代价, 再加上一个不充分的解决方案, 不久就可能会在价值上超过任何初期节省额。另外, 一个可能迁移的不可靠性也是造成挫折和瘫痪的一个秘诀。

成本不应该用做应用服务器的价值衡量标准。真正的检验标准是: 它是否满足本组织的需要。请记住, 要考虑总的所有权成本。应用服务器上的支出应该只是总成本的一小部分。

供应商关系

束缚于一个特定应用服务器意味着束缚于一种跟服务器供应商的连续关系。售前建议的质量可能表明供应商售前专家的质量, 但该组织向其顾客承诺的级别是十分重要的。这里

需要考虑的一些重点包括：

- 除了支付产品的价格外，你的组织还爽快地支付支持费用吗？
- 产品的其他用户是怎样找到支持经验的？

开放源产品的支持不同于商业产品，混合了积极和消极两个方面。从积极的方面看：

- 蓬勃发展的开放源产品提供范围广泛的支持选择——从免费的邮件列表到商业咨询。**Red Hat**的成功就说明了一个以某个开放源产品的附属商业出售品为基础的模型是如何生存的。
- 开放源产品的开发人员散布在世界各地，并经常在周末工作，因而意味着你可以不分昼夜地随时得到一个答复。

另一方面，没有公司会被迫负责纠正问题。开放源开发人员常常为他们已经贡献过的产品而自豪，通常也非常乐意帮助纠正错误和解决用户的问题。如果你张贴问题的帖子导致你被拒绝，而没有得到帮助，你将没有抱怨的对象。**JBoss**邮件列表的存档文件同时显示了积极和消极这两个方面。

开放源支持往往也比商业产品所提供的支持更民主。张贴有可能会得到相同的关注，无论它们来自单个开发人员，一个小型机构，还是一个大型组织。另一方面，商业服务器供应商可能会更关注他们与关键购买者的关系，从而意味着已经购买了多个或企业级许可的大型组织可能会获得优先关注。这也是大型企业很少选择开放源产品的原因之一。

供应商生存能力

自应用服务器市场于1999/2000年达到顶峰以来，由于过多的玩家和困难的交易环境，这个市场已经经历了轻度的衰退。一些较小的玩家已经退出了应用服务器市场，以便把精力集中在他们特别擅长的领域，而且这种趋势似乎还在继续。

因此，做选择时考虑应用服务器的生存能力是很重要的。在最糟糕的情况下，如果该产品不再被支持，会发生什么情况呢？如果你一直严重依赖于该服务器的某个特定能力，这是特别危险的。

从积极的方面看，小公司可以证明是反应更快的，并更关心个别顾客的需要，特别是个别小顾客的需要。

开发与部署

选择服务器的另一个重要标准是证明开发和部署应用到服务器上将多么容易。从理论上讲，当开发和部署应用时，服务器不应该强加麻烦的专有需求，并且启动起来应该快速。

能在没有较大影响的情况下发布软件的新版本是最基本的要求。从理论上讲，在不关闭服务器的情况下做发布应该是可能的。

开发与部署考虑事项包括：

- 服务器启动的速度、应用部署的速度和服务器内存需求。这些是开发期间的重要问题，但在生产中不是太重要，因此我们在这里可能需要接受一个折衷方案。
- 开发和部署应用中所需要的任何额外的专有步骤，比如编写复杂的专有部署描述符的需要和编译以前“编译”EJB JAR文件的任何需要。

- “热部署”应用（即不关闭服务器而部署或重新部署应用）的能力。
- 服务器所提供的管理工具的完善度（和可靠性）。
- 与Simple Network Management Protocol (SNMP, 简单网络管理协议) 和Java Management Extensions (JMX, Java管理扩展) 之类的管理标准的集成度。
- 运行时的错误报告的质量。该质量将影响服务器的可用性。能够生成（至少在调试时能够生成）关于异常的栈跟踪并在服务器日志中如实显示它们是很重要的。不同的服务器在这方面有很大区别。
- 部署时的错误报告的质量。该质量的范围可以从“出了毛病的东西”到能使问题被快速查出的有用消息。

增值特性

随着越来越多的服务器符合J2EE，可靠性、可缩放性和增值特性越来越明显地区分开应用服务器。由于J2EE规范没有满足所有常见需求，所以这是不可避免的。证明是重要的增值特性可能包括：

- 聚类支持的质量，包括故障消息送达的级别。
- 服务器所携带的JMS解决方案的质量。虽然任何符合J2EE的服务器都必须提供一个JMS实现，但有些服务器携带Sun Reference Implementation，这意味着在对JMS进行关键使用之前必须先购买一个第三方消息传递解决方案。
- 性能，应用性能将受到应用服务器性能的限制。
- 对Web服务（J2EE规范还没有涵盖的一个重要发展领域）的支持。但是，一个第三方产品也可以提供Web服务支持。Apache Axis SOAP包特别易于使用，并能给任何服务器增加强有力地Web服务能力。
- 对具有CMP的实体组件的完善支持，从而提供像开放式加锁那样的特性，而这种特性是J2EE规范没有要求的。此类特性是让CMP实体组件变得可用所必不可少的，因为该规范疏忽了一些重要的问题。

文档的质量

大多数供应商提供优秀的免费联机文档。这是服务器评估期间的一个宝贵资源。考虑图书和其他与该服务器相关的独创资源的可获得性也是一个不错的主意。这个考虑往往有利于那些在市场上领先的服务器。

技能的可利用性

保证项目团队的一个重要子集具有选定服务器（而不仅是J2EE）的实际经验是十分重要的，而且保证可以雇用到更多具有该服务器使用经验的开发人员也是很重要的。通过从作业站点中检查其他公司的需求，检查该服务器供应商所运行的任一认证程序的流行程度，以及寻找致力于该服务器的本地用户群来评估服务器特有技能的可用性是可能的（不过，对J2EE有深入了解的开发人员很快就会熟悉任何服务，而且最终将胜过只了解某个特定服务器和对J2EE概念只有浅显掌握的开发人员）。

同样，这个考虑往往有利于在市场上领先的那些服务器。需要注意的是，由于代理商和老板往往更喜欢寻找产品（比如WebLogic），而不喜欢寻找技术（如EJB），所以许多开发人员对参与使用了他们认为没有光明未来的不著名产品的项目持谨慎态度。

用户经验

跟使用了你所评估的每个服务器的组织内的职员交谈是至关重要的。供应商应该能够提供参考站点。如果可能，最好是不让销售人员在场，并进行不拘礼节的交谈。跟个别开发人员进行交谈可能比组织正式的会议访问更有启迪作用。令人沮丧的问题很可能会在即席谈话时被提及。

大多数服务器供应商都有新闻组或邮件列表。这些东西通常是可免费访问的，并提供了产品使用经验方面的宝贵见解。

- 供应商通常不节制新闻组，所以新闻组会是该产品中故障级别的一个有用暗示。人们正在争论极限应用的优点，还是发现了产品中的基本问题？什么东西妨碍了它们投入使用？（忽略这样一些用户的抱怨：他们似乎完全不懂J2EE和该产品。）
- 新闻组也会是安装和部署应用到该产品上有多困难的一个有用暗示。
- 新闻组有多活跃？不够活跃可能暗示一个产品已证明是不能用的，或者说在市场上是失败的。
- 人们获得答复有多快？未得到答复的合理问题的数量越多，业界对该产品了解不充分的可能性就越大。
- 该产品的任何一名开发人员经常向新闻组张贴东西吗？它们是有帮助的吗？这种情况在商业和开放源领域中都会发生。例如，许多高级WebLogic开发人员进行定期的张贴，而JBoss开发人员也是如此。

下面这些指向新闻组的链接可能是一个有用的起始点：

- **WebLogic**: BEA宿主了许多活跃的WebLogic新闻组。其索引在<http://newsgroup.bea.com/cgi-bin/dnewsweb>站点上。
- **Orion**: 从Orion服务器主页 (<http://www.orionserver.com>) 中跟随“Mailing List”链接。
- **JBoss**: 存档文件在<http://lists.sourceforge.net/lists/listinfo/boss-user>站点上。另外，<http://main.jboss.org>站点上也有许多论坛。

选择过程

笔者建议，为选择应用服务器使用一个半正式的过程，以保证该过程是透明的，并用可获得的证据加以证明。太多地讲究形式将会导致无休止的委员会会议，以及脱离现实的高层文档的编写。太少地讲究形式将会导致一个特定的决策和不充分的文档编写，进而导致该决策将来会受到怀疑（未来的商讨可能会不经意地涵盖相同的范围）。

开发团队应该最大限度地参与最后的选择。笔者建议把下列这些内容考虑进去：

- 第三方报告，比如ECPPerf和CSIRO应用服务器的比较。
- 委托一家咨询机构或具有丰富J2EE经验且无既得利益的外部专家所制作的报告。该

报告应该论及具体的项目和企业需求，而不是成为一个标准报告。

- 与来自竞争供应商的售前专家所进行的技术问题的讨论。
- 每个服务器的安装基础的证据。
- 与使用过已被列入候选名单的服务器的其他用户进行的讨论。候选名单应该在评估过程的中途被草拟出来，因为较后的那些步骤将要求把大量时间专用于正被评估的每个服务器。
- 支持的质量。
- 成本。
- 在每个被列入候选名单的服务器上所做的一次概念、部署和测试的内部验证。这会引出许多问题，并且比运行一个“标准”应用（如Sun的Java Pet Store）有用得多。

选择应用服务器时最主要错误

根据笔者的经验，下面是选择应用服务器中最常见（而且代价最高）的一些错误：

- 最大的错觉。把资源和金钱花费在你没有的一个问题的一个复杂解决方案上，即使它是一个好解决方案。这方面的一种特殊情况是可缩放性错觉。虽然Web应用的利用率可以迅速提高，但能够支持一个巨大的用户总数未必总是至关重要的，而且对许多组织来说，巨大的用户总数将永远达不到。这个错误常常会导致金钱被浪费在多余的昂贵许可上。在给定实际业务需求的条件下，一个伤害更大的结果会是整个项目生命周期内成本的增加，因为一个多余的先进产品管理起来必然很复杂。
- 没有考虑总所有权成本，而许可成本只是该成本的一部分。
- 没有内行经验就选择一个应用服务器。第三方报告、基准测试报告和市场营销广告再多也代替不了将在该服务器上构造应用的那些开发人员所拥有的内行经验。
- 高级管理人员在完全没有咨询开发人员和设计师的情况下就宣布的决策。令人感到痛心的是，这个错误十分普遍，通常也反映了一种将以其他方式使项目瘫痪的令人担心的文化。
- 购买服务器只考虑应用的眼前需求，即使可以适当地预见到不久后该应用的更苛刻的需求。这个错误可能会导致早期就实施迁移。

“纯技术”陷阱

所有好的开发人员不仅对他们所使用的技术感到兴奋，而且对其他第一流的技术也感到兴奋。杰出的开发人员在这种情绪可能会使其雇主损失金钱时，会控制这种情绪。

就使用新的和有趣的技术而论，开发人员自然会遇到一个利益冲突问题。首先，存在智力挑战。然后，存在个人履历要求。由于代理商和雇主通常首先扫描一个候选者的履历来查找时髦词句，所以收集新的时髦词句（而不是深入的新技能）是一件优先做的事情。

对任何一项新技术的引进都需要进行认真考虑，尤其是在打算引进这项技术来解决单个问题的时候。一项新技术一旦被引进就可能需要被维持很多年，进而使该组织的技术组合变得更复杂。它对支持来说将会有什么样的意义呢？它将需要系统中的其他任何地方都不用

的开发技能吗？它将有特殊的防火墙或其他需求吗？

要避免为了技术本身的缘故而使用技术。增加到项目中的每项新技术都会使项目的维护变得更困难，除非可替代的方法明显不适合。增加新技术是一个战略性决策，如果只是为了一个特定问题，不应该随随便便地采用它们。

一个与本题目相关的真实案例：一个大型.COM为业务逻辑使用了一个EJB服务器，其中一个表示层使用了小服务程序和JSP。我们叫做Joe的一名开发人员被要求开发一个内部管理工具来执行正在进行的数据迁移。Joe的经理正在喜马拉雅山脉中做艰苦的长途旅行，所有高级技术职员正穷于应付一个重要发布，因此这是Joe设计和实现一个解决方案的大好机会。

Joe花费一些时间考虑了该问题，并最终认为EJB满足不了他的需要，而且Web接口也无法提供这种功能度（这两个假设稍后证明是错误的）。数周后，他的解决方案由于一个笨重的Swing小程序接口而包括了RMI。除了J2EE服务器使用了RMI之外，无论RMI还是Swing都没有用在该系统中其他地方。

该接口在Joe的计算机上看起来很优美，但该管理工具由于下列原因在3个月内就遭到遗弃：

- RMI的使用意味着需要使一个RMI服务器保持运行。这对站点管理来说是一项不受欢迎的附加工作。RMI的使用也需要附加的防火墙规则。
- 该管理工具的用户原来使用低规格的便携电脑。Swing的使用意味着该应用会以一种令人痛苦的慢速度运行在这些系统上。它也要求安装Java Plug-In，进而要求开发团队与终端用户保持联络。
- 把精力都投人到这个迷人的界面中意味着忽略了故障纠正。
- Joe让这家公司使用他所喜欢的技术。由于该应用的体系结构与系统中其他应用的体系结构是如此不同，并且没有使用其他开发人员所熟悉的框架，所以很难确定需要做些什么工作来解决那些故障。结果是丢弃整个项目并重新再来反而更便宜。

至少在本案例中，本可以使一个Swing/RMI解决方案变得管用；尽管这些技术在本案例中被误用，但它们是已经过证明的技术。不过，“纯技术”时常也有不适用的时候，从而使它们变成了天生就有风险的技术。

何时使用替代技术来补充J2EE

不要因为你一直束缚于作为一个战略平台的J2EE，就意味着J2EE是解决每个问题的最佳方法。例如，在上面描述的案例中，Joe正面对一个数据库迁移问题。他从一开始就通盘考虑以数据库中心的解决方案可能会做得更好。通常能用于具体任务的其他技术是PL/SQL之类的数据语言和XSLT。

这里有一个痛苦的折衷方案。一个应用中结合的非J2EE产品越多，它维护起来就越困难。

除了J2EE之外，我们还应该看到的一些迹象包括：

- 当使用一个不同的工具会导致显著的简化时。这种情形对Joe的问题可能也很适用。
- 当利用现有投资是可能的时。例如，以现有的数据库存储过程为基础来解决某个特定问题或许是可能的。
- 当该选择利用组织中的现有技能（比如DBA和Perl技能）时。

许多J2EE专业人员对企业软件采取了一种过度以J2EE为中心的观点。虽然J2EE是一个很杰出的平台，但它不能而且也不应该试图包含一家企业的所有软件需求。

我们应该记住，J2EE是一个杰出的集成平台。因此，我们不应该觉得需要使用J2EE技术来试图实现每件东西，而是应该准备在必要时使用J2EE来使不同的资源能协同工作。

可移植性问题

Java的“编写一次，到处运行（WORA）”的基本前提，保证了J2EE应用比使用任何同类平台所编写的应用更可移植。我们可以确信，Java对象在不同环境中将行为一致，而Sun J2EE Compatibility程序将帮助保证不同的J2EE服务器满足J2EE规范的要求。由于J2EE服务器往往是使用Java编写的，所以服务器本身在多种操作系统上通常被支持。

遗憾的是，CTS没有展现这幅完整的画卷。在J2EE规范的范围内编写的应用能够在服务器之间进行移植虽是一件不错的事情，但是，如果一个现实的应用需要超出那些规范的范围，怎么办呢？

下面让我们来仔细看一看可移植性在J2EE上下文中究竟有什么含意。

可移植性的含意

围绕J2EE应用的可移植性，有几个问题：

• 应用在应用服务器之间的可移植性

这或许是在J2EE上下文中考虑可移植性时首先想到的第一个问题。应用在应用服务器之间的可移植性通常将取决于下列因素，其中的大部分因素都在我们的控制之下：

- 目标服务器实现那些J2EE规范的忠实度有多大？
- 应用的代码符合那些J2EE规范有多大程度？当目标服务器实施了J2EE要求的一个不同子集时，未加注意的违反是移植应用时的一大障碍。
- 应用利用原服务器的专有特性有多大程度？

Sun CTS帮助消除了上述第一个问题。第二个问题可以通过对J2EE的彻底了解和应用开发期间的纪律来避免。剩下的最后一个即第三个问题是较难的问题（当然，我们只应该在专有服务器特性提供真正好处时才使用它们）。下面，我们将讨论保证专有扩展的使用不破坏可移植性的策略。

• 应用在服务器操作系统之间的可移植性

J2EE在这里非常适用。Java作为一种语言其可移植性是极强的。所有服务器操作系统上的虚拟机都是可靠的。主流应用服务器在多种平台上被进行过测试，因此一个应用能在不需要任何开发工作的情况下被移植（比如从Windows 2000移植到Solaris）

是有可能的。那些J2EE规范不鼓励服务器特有的实现特性，比如业务对象中文件系统的使用。

- **更改J2EE应用所依赖的重要资源的能力**

这里的明显例子是切换数据库。事实上，这在一个系统被部署之后极少发生。有这样一种说法：“数据往往逗留在它登陆的地方”。实际上，J2EE应用将被根本地更改的可能性大于一个激活数据库将被迁移的可能性。在这种情况下，有两种可能的情景：

- 对一个同类竞争性资源的更改。假设我们需要从一个使用Cloudscape的概念试验模型迁移到Oracle上的一个部署。尽管修改SQL语言通常将是不可避免的，但JDBC在这里提供了一个基本层次的抽象。
- 对资源类型的更改。这涉及一个更根本性的改变，例如从一个RDBMS到一个ODBMS。在这种情况下实现可移植性将涉及一个更高层次的抽象，比如实体组件。但是，这种更改的可能性更小，而且这么高层次的抽象可能会减低效率和使应用代码变复杂。

区分这些问题是非常重要的。笔者觉得，已有过多的注意力被放在了数据库可移植性上。这反映了一种以J2EE为中心的不切实际的企业IT观点。组织在数据库上投入了大量金钱。他们通常不偷偷离船，并认为这项投资会得到充分利用。因此，忽视专有数据库特性在简化应用代码和改进性能方面的潜力是不明智的。另一方面，应用服务器之间的可移植性更容易达到，并且可能会提供更大的商务价值。

需要重点注意的是，即使我们从不选择迁移我们的应用到一个不同平台，仍可以从J2EE的可移植性中享有一些好处。由于J2EE支持范围这么广的多种实现，所以有大量对各种平台有使用经验的J2EE开发人员。这有利于J2EE团队中含有具备各种技能的成员。每当笔者与一家只使用Microsoft产品的组织打交道时，都会想起这一点。

总之，J2EE可移植性消息是好的，但是完全的可移植性既是不可能的，又是不值得向往的。可移植性的限制主要体现了那些J2EE规范的性能必要性和局限性，以及使用平台特有技术的最终要求。正如我们将要看到的，有多种可以用来解决这些问题的技巧。

可移植性的一种实用方法

正如前面已经讨论的，拒绝使用任何供应商特有功能度是十分危险的。设计师和开发人员是被雇来为真正的组织提供工作解决方案的，而不是被雇来开发“纯”J2EE应用的。

但是，这并不意味着为可移植性而设计是不可能的。这可以用3种方式来实现：

- 编写遵守那些J2EE规范的代码。为编写符合J2EE规范的代码而采用组织级的标准，并保证所有开发人员都知道和遵守它们。定期地使用J2EE Reference Implementation verifier工具来检查符合性。
- 熟悉标准J2EE体系结构的能力，并在存在一个满意的标准替代方案时避免使用平台特有的解决方案。但是，不要自动拒绝专有解决方案。
- 使用松耦合来隔离出平台特有的使用，以保证该应用的设计（即使不是它的全部代码）保持可移植性。

中间整备环境与发布管理

无论一个应用被编写得多么完善，运行它的服务器有多么高的质量，如果没有一个有效的发布管理过程，以及相配的开发与中间整备环境，其稳定性将会是令人失望的。在该应用到达生产阶段之前，提前投入时间来建立将被所有参与人员都熟悉的、可靠而又可重复的过程是很重要的。

一个典型的Web项目可能需要3个完全不同的环境：

- **开发环境**

这应该精密地镜像“测试”和“生产”环境。开发人员应该能相当自由地做修改。

- **测试或中间整备环境**

这应该尽可能精密地镜像“生产”环境。尤其是，它应该含有和“生产”环境相同的数据量，以便性能测试将是有意义的。成本可能不允许使用和生产中完全相同的硬件。不过，如果部署在一个“生产”聚类上，“测试”也应该被聚类，因为聚类可以引发可能只在生产中才会出现的许多问题。由于负载测试通常将在“测试”中进行，所以这也是它应该尽可能紧密地镜像“生产”硬件配置的另一个原因。“测试”的发布应该需要一个正式的发布过程。“测试”使用和“生产”完全相同的操作系统是至关重要的。

- **生成环境**

真实站点在里面运行的环境。开发人员应该没有直接访问“生产”环境的权限，而且在“测试”环境中的测试完毕之后，到“生产”环境中的部署应该是一个正式发布过程的最高点。

在许多站点中，所有开发人员在各自的工作站上都有各自的“开发前”沙箱模型。这通常是一个Windows工作站，而开发和整备服务器通常将运行一个UNIX类版本。虽然不同操作系统的使用引进了一些风险，因为那些JVM是无法一致的，所以它在生产效率方面的好处可能会使它成为一种值得采用的方法，因为开发人员可能需要在他们的本地计算机上执行生产操作系统不能最好地提供服务的其他任务。

发布管理指这样一项任务：保证一个应用发布从“开发”、“测试”到“生产”的发展是可逆和可重复的。在应用开发开始之前，每家组织都先开发一个让所有团队成员都接受的发布管理策略是十分重要的。由一个发布管理策略解决的问题将包括：

- 给一个版本控制系统中的所有源代码“加标签”，以便能轻松地抽取每个发布中所包含的那些人工制品的一个快照。因此，一个发布管理策略本质上将基于使用中的版本控制系统。
- 到“测试”和“生产”服务器上的部署。这可能包括到一个服务器聚类的转出。因此，使用中的应用服务器（以及它所提供的部署工具）是实际发布管理中的另一个关键变量。
- 一个回退策略，以便保证如果一个发布在“测试”或“生产”环境中引起了意想不到的问题，回退到该应用的前一个版本是可能的。

- 任何必需的数据库表的创建，以及必要时数据的供应。
- 对发布之间可能会变化的二进制相关性的管理（比如Velocity之类第三方产品所携带的JAR文件）。
- 因每个发布而需要的文件编制。
- 因每个发布而需要的结束过程。

成功的J2EE项目不仅取决于好的应用和应用服务器软件，而且也取决于强有力的发布管理习惯和开发人员的本地计算机与真实服务器之间的适当中间整备环境。在开发这些方面的一个健全而又可工作的策略时，投入足够时间并包括所有项目人员是至关重要的。

建立开发团队

J2EE项目需要各种各样的技能和好的交流。J2EE开发人员不仅必须拥有扎实的Java技能和J2EE知识，而且还必须能够成功地与他们所使用的软件方面的专家（比如DBA和主机系统）以及表示层开发人员保持联系。

由于企业（不仅仅J2EE）项目的复杂性，试图在用人上省钱很可能会导致相反的结果。从项目一开始至少拥有一批经验丰富、技能高超的核心人员是十分重要的。这也兼顾了指导资历较浅的开发人员来补充培训计划方面的投资。

当内部地获得第一流技能是不可能的时，考虑在项目生命周期的早期从具有专门技能的组织或个体那里获得短期咨询可能是一件值得做的事情，比如要求他们提供一个体系结构文档或一个概念试验模型。这么做的代价是相当高的，但可能是一项明智的投入，如果能帮助防止未来的问题的话。

同样，项目有一个良好的开端也是十分重要的。在项目的开始和加工阶段可以利用的人员质量将是特别重要的。这两个阶段要么会建立将使项目平稳地到达成功完成的合理体系结构和准则，要么会导致通过增加越来越多的人员也无法解决的前进中的问题，因为它一开始就错过了管理重点。

团队结构

假设你的组织中已经有了具备各种所需技能的职员，剩下的问题就是如何把他们组织成一个团队。同样，根本不存在惟一的正确答案，因为情况是变化的。但是，有几个值得考虑的常见问题。

谁拥有体系结构

最重要的决策之一是安不安排一名将对设计决策负最终责任的“总设计师”，或者更民主地做事。

安排一名惟一的总设计师有几个好处：

- 思路清晰；
- 快速制定决策；

- 只有一个人能够在必要时代表项目团队（当与高级管理人员的联系必不可少时，这在墨守陈规的大型组织中可能是非常重要的）。

这个策略的成功将主要取决于该设计师的人格和技能，他必须对J2EE有非常深入的全面了解，并应该能够赢得所有项目人员的尊重。

但是，必须考虑下列这些风险：

- 对单个人的过分依赖。此人可能会离开公司（这将意味着其他人需要被牵连到此人的工作中）。
- 单方面决策的可能性。这种方法不以多数人的意见为基础。
- 该设计师可能会脱离工作现场的可能性。一名设计师必须充分地亲临现场，以了解开发人员所遇到的所有实际问题。作为项目交流的一个关键渠道，该设计师必须能够在该组织中上传下达。

另一种方法是安排一个委员会来总负责设计。这种方法更官僚，并且可能会降低决策制定的速度，同时也只提供极少的好处。它的成功很可能依靠一名有力的主席（不一定是最高级的技术人员）。

Extreme Programming (XP)（一种民主程序设计模型）的专业人员反对“总设计师”的提法（有些人反对软件体系结构本身的提法）。XP使用“集体代码所有权”，在这里，任一程序员都能修改项目中的任何一行代码。这种方法在小项目中可能会管用，但不太适合大项目，因为控制权掌握在不计后果的开发人员手中。根据笔者的经验，大项目需要一个更高层次的形式。

无论谁“拥有”系统体系结构，重要的是必须清楚地表达它，并且整个团队都接受它。把一个设计强加给开发人员是毫无用处的。

纵向或横向责任

另一个决策涉及给开发人员的责任分配。这里有两种常见模型。

纵向方法

这种方法把实现整个使用案例（从置标生成到跟数据库和遗留系统的交互）的责任分配给个别开发人员或小团队。这种方法要求所有开发人员都拥有整个J2EE技术组的专门技能。

这种方法的一个优点是专家组之间根本没有交流费用。这种纵向方法也非常适用于一个受XP影响的使用集体代码所有权的开发过程。

不过，这种纵向方法可能会导致效率的损失和不稳定的代码质量。与这方面的专家相比，甚至连一个好的开发人员都有可能浪费时间来处理他们不熟悉的问题。经验的缺乏可能导致拙劣的解决方案，以及时间和精力的浪费。

横向方法

这种方法把开发人员分配给具体的专门技术领域，比如JSP或EJB。

这种横向方法是一种更传统而且可能是更保守的软件开发方法。它可以提高效率，因为开发人员把他们所有的时间都用来磨砺他们在一个小领域方面的技能。另一方面，它使良好交流在处理不同技术的开发人员之间变得非常必要，因为几个开发人员或小团队将被要求协同实现每个使用案例。不过，迫使存在于开发人员头脑中的理解被表达出来（而更好的是被写出来）可能是一件好事。使用这种横向方法不要求一个团队（比如JSP）中的开发人员必须等待另一个团队（比如Business Objects）完成工作：接口应该在初期就被商定好，进而意味着在真正的集成变得可能之前可以使用简单的模拟实现。

这种横向方法中的责任划分可能包括：

- **置标开发人员**

此项开发将包括HTML/XHTML开发，可能还包括JavaScript。

- **表示层开发人员**

负责较技术性的表示层工作，而不是负责设计外观好看的HTML。在给定所需置标的条件下，他们将提交JSP页、XSLT格式表、Velocity模板和其他Web模板。在一个MVC应用中，JSP页将含有极少的Java代码，所以可能属于置标开发人员的范围内。

在像Swing应用那样的GUI应用中，表示层开发人员将是专长于窗口化库的真正Java开发人员。

- **Web层Java开发人员**

负责MVC框架动作类、JSP标志处理程序和Web层助手类。

- **业务对象开发人员**

负责实现应用业务逻辑。业务对象开发人员将在适当的地方使用EJB，并且应该对EJB有扎实的了解。Web层开发人员和业务对象开发人员必须在项目生命周期的初期合作定义能使他们的类相互通信的接口，以保证他们能够平行而自由地工作。

- **数据存取开发人员**

Java开发人员负责有效的DBMS存取。通常情况下，这个角色将由业务对象开发人员来承担。专家任务可能包括配置和处理O/R映射产品（如TopLink或JDO实现），以及利用一个EJB容器的实体组件CMP实现。此类开发人员常常是Java与RDBMS世界之间的一座桥梁。

- **DBA**

基础数据库方面的专家。DBA能够在关键方面给J2EE开发人员提供有价值的帮助，比如保证好的性能和正确的加锁行为。

团队越小，每个开发人员或开发人员组有可能会担当的这些角色就越多。

组合这两种团队结构模型是可能的。下列这些准则会是有帮助的：

- 一个好的设计使运用任一方法变得容易，因为它形成技术之间的明确划分。
- 一些开发人员最乐意处理J2EE技术范围，其他开发人员不喜欢离开他们喜爱的领域。因此，哪个策略合适可能取决于有价值团队成员的个性。
- 从中间件编程中分离出置标创作（或GUI表示）是使用横向方法几乎总有意义的那些领域之一。Java开发人员很少能成为好的HTML编程人员，而许多好的HTML编程人员又缺少对系统体系结构的基本了解。设计一个Web应用的关键之处是把表示与业务

逻辑分离开；团队结构不应该颠倒这个成果。

- 团队越大和开发过程越正规，横向方法可能就越有吸引力。
- 在横向方法中，设立一个由摸清了整个产品的高级开发人员所组成的组将总是必不可少的。这些开发人员将明白整个体系结构，并将能指导其他开发人员。同样，在纵向方法中，拥有具备专门技能并能在特殊技术方面指导其他开发人员的开发人员也是必不可少的。
- 在纵向方法中，开发人员可能会在不同的技术之间被轮换，同时保证每个开发人员组仍有很高的自信心。

选择开发工具

J2EE开发的某些方面需要工具。用于J2EE开发的工具正变得越来越好，但许多工具仍不及它们的Microsoft对应物。不过，Java开发（以及较小部分的J2EE开发）通常不需要严重地依赖工具。

像一个好的XML编辑器之类的通用工具能够简化部署描述符的编辑和降低错误的概率。

如果使用EJB（特别是CMP实体组件，如果使用了它们），J2EE特有的工具就变成必不可少的，而不只是有帮助的。实现每个EJB所需要的多个文件的自动生成与同步是有帮助的。如果你只在使用会话EJB和Web构件，手工处理Java文件和编写部署描述符是可能的。不过，EJB 2.0 CMP实体组件部署描述符和某些服务器特有的部署描述符（同样，通常是在CMP实体组件不被关注的地方）用手工编辑起来太复杂。

请注意，工具不等于IDE。许多基于脚本的低端技术工具在J2EE开发中常常效率很高。

现在让我们来看一看能用于J2EE开发的一些主要工具类型。本讨论只反映笔者自己的经验和见解；这是一个有许多可替代方法的领域。

可视化建模工具

位于工具谱最智能端的是可视化建模工具，比如Together，它们具有生成J2EE代码的能力。这些产品通常比较昂贵。就个人而言，笔者不使用它们，尽管这是一个爱好问题，而且笔者知道，一些优秀的J2EE设计师和开发人员也不使用它们。笔者有时把可视化建模工具用于它们的UML支持（包括从一个可视化模型中生成代码），而不愿使用J2EE特有的扩展。由于UML建模工具与J2EE没有直接关系，所以它们在这里不是密切相关的。

Web层构件和会话EJB与普通Java类没有太大的不同，因此不需要特殊的工具。笔者往往不使用实体组件。当真地使用它们时，笔者也不使用对象建模工具来生成代码。正如我们在第7章中将要讨论的，就关系数据库而言，从一个对象模型中驱动实体组件设计会产生灾难性的结果（当今最常见的）。集成的可视化建模与J2EE代码生成工具使这种设计方法容易得十分危险。它们的智能导致了开发人员对这种基本方法不太可能产生怀疑的危险。经验不丰富的开发人员可以轻松地使用这样的工具来快速地生成不需要真正明白的代码——未来问题的一个滋生地。

IDE

虽然许多组织的标准化只建立在单个IDE上，但笔者认为，应该让各个开发人员自己去选择他们所熟悉的那些IDE。IDE对于项目配置是一个糟糕的贮藏库，而且开发人员通常具有应该得到尊重的强烈爱好和憎恶。有些开发人员非常执着地坚持使用他们所喜爱的文本编辑器，尽管当今的Java IDE是如此之好，以致很难再为他们的这种做法找到借口（笔者几年前就放弃了这种做法，那时笔者跟vi和TextPad说了再见）。

在GUI开发中，一个特定的IDE自然会变得对开发过程非常重要：例如，如果它提供特别强大的“窗体（Form）”建立功能度。对于服务器端开发来说，往往只有较少的Java代码，而且手工生成该代码是令人乏味的，因而没有必要去讨论IDE标准化问题。

笔者建议，在选择一个IDE时要考虑再制造（refactoring）支持。这节省了大量时间，并有助于养成好的习惯（我们将在第4章中讨论再制造）。Eclipse (<http://www.eclipse.org>) 是一个免费开放源产品，并且在这方面做得特别好。Eclipse提供如下自动支持：

- 重命名包、类、方法和变量（包括更新所有相关性）
- 支持“提取方法”之类的常见再制造，并把一个方法提升到一个超类
- 搜索整个项目中指向某一个类或方法的引用

生成工具

IDE的一个主要问题是，它们的使用无法被脚本化。因此，动作不能被置于版本控制之下，也不能被可预知地重复。检查项目文件或许是可能的，但项目文件是IDE特有的，并且可能使用了二进制格式。另外，有些任务不能用IDE轻松地执行，如果一个特定IDE是不可或缺的项目贮藏库，这就产生了一个严重的问题。

如果没有一种生成所有项目人工制品的有效可重复方法，在J2EE项目中获得成功是不可能的。基本任务包括：

- 编译所有源代码；
- 编译测试代码；
- 创建J2EE部署单元——WAR、EJB JAR和EAR；
- 生成整个应用的Javadoc；
- 在单个操作中自动检测测试用例并运行测试。

较高级的任务包括：

- 在版本控制系统之外检查代码；
- 使用指定的生成标识符来标注版本控制系统中的所有文件；
- 在一个测试运行和清理一个数据库之前以及在一个测试运行之后，把该数据库置于所需要的状态中；
- 部署到一个应用服务器上。

所有这些任务都能使用Another Neat Tool (Ant) 来完成，它是针对Java生成脚本的事实标准机制。Ant是一个来自Apache项目的开放源工具，可以从<http://jakarta.apache.org/ant/index.html>站点中获得。

尽管Ant是基于XML的并使用Java，但它与make而不是与外壳脚本共享许多概念。一项Ant任务对应于一个make目标，但Ant更直观，并提供更好的交叉平台支持和更好的面向Java功能度。Ant的好处包括：

- Ant能和读者所喜爱的IDE联合使用（在这种情况下，让Ant把类编译到一个不同于IDE把类编译到的地方，以保证IDE不会变得分不清楚目标是一个不错的主意）。大多数开发人员将在一个IDE中处理代码（并在第一个实例中编译它），但使用Ant来生成发布。流行的IDE提供Ant集成，进而允许Ant任务从IDE内被运行。
- Ant提供了针对许多常见需求的标准“任务”，比如执行系统命令，生成Javadoc，复制和转移文件，验证XML文档有效性，从远程服务器中下载文件，以及运行针对数据库的SQL脚本，而且还提供了Java源代码的编译（这项任务通常被认为是Ant的主要角色）。
- 许多产品提供Ant任务来简化它们的使用。例如，WebLogic 6.1及以上版本就把Ant用于某些任务。许多第三方Ant任务可用来执行范围广泛的功能度，其中包括包装J2EE应用并把它部署在几个服务器上。
- Ant提供了一种有效方法用来脚本化到一个应用服务器上的部署。
- 像make一样，大多数Ant任务检查相关性。因此，Ant将自动地只重编译已经发生了变化的源文件，并跳过其编译结果仍是最新的任务。
- Ant是可参数化的。Ant“属性”可以被设置成保存在操作系统和各计算机之间变化的值（比如一个类路径的绝对根位置），以保证每个生成脚本的其余部分是完全可移植的。Ant还能根据标准（比如操作系统或某些类的可用性）来执行不同的任务集。
- Ant是可扩展的。用Java定义定制的Ant“任务”是相当容易的。但是，Ant带有如此多执行常见操作的任务（其中的许多任务与J2EE相关）和如此多可用的第三方任务，以致很少有开发人员将需要实现他们自己的Ant任务。

Ant在商业和开放源项目中运用得非常广泛，因此任何专业Java开发人员都应该了解它。

Ant可以用于除简单地生成源代码之外的许多任务。可选任务（可以作为一个附加下载从主要下载站点中获得）支持WAR、EJB和EAR部署单元的生成。

如果存在笔者将再次运行一条命令的任何可能性，笔者将绝不键入类路径：笔者为每条面向Java的命令创建一个Ant build.xml文件，或者给一个现有生成文件添加一个新任务，无论这个任务是多么小。这不仅意味着笔者能立即使某件东西工作（如果笔者以后返回到它），而且也意味着笔者能够注解自己需要做的任何特殊事情，因此笔者将来不会浪费时间（笔者曾经使用Ant备份过构成本书的源代码和文档）。

如果读者还不熟悉Ant，请务必学习和使用它。继续使用你所喜爱的IDE，但要保证每个项目动作能够通过一个Ant目标来完成。提前花费少许时间编写Ant生成文件，以后会得到回报。有关有效使用Ant的指南，请参见http://jakarta.apache.org/ant/ant-in_anger.html站点。

代码生成器

对于普通Java对象和Web层类来说，几乎不存在自动生成代码的需要。但是，EJB开发

中所需要的许多人工制品已经使代码生成工具变得十分有吸引力，尤其是在涉及实体组件的地方。

与IDE相比，EJB代码生成器的技术性较低，但是对EJB开发却是十分有效的。

正如前文已经讨论过的，如果我们正在使用CMP实体组件，手工地制作和维护所有必需的部署描述符（标准和供应商特有的）是不可能的。下列免费工具在EJB组件实现类中使用特殊Javadoc标记来强行生成用于几个服务器的其他所需的Java源文件（本地及构件接口）和部署描述符。和IDE“EJB Wizard”不同，这是一种可脚本化的方法，并且兼容于任何IDE或编辑器。

- **EJBGen**

<http://www.beust.com/cedric/ejbgen/>

该工具由BEA开发人员Cedric Beust编写，并被捆绑在WebLogic 7.0中。

- **XDoclet**

<http://xdoclet.sourceforge.net/>

这个相似但更雄心勃勃的工具由Rickard Oberg编写，能以开放源形式来获得，并能用来执行EJB生成以及其他任务。

一种替代的EJB代码生成方法是在一个XML文档中定义必要的数据，通过启用XSLT的使用来生成必需的多个输出文件。同样，这仅对处理CMP实体组件的复杂性是必不可少的。几个此类产品之一是来自Tall Software公司（<http://www.tallsoftware.com/lowroad/index.html>）的LowRoad代码生成器。

版本控制

拥有一个好的版本控制系统是非常重要的；除了一个像Ant那样的好生成工具之外，一个版本控制系统也是每个成功的发布管理策略的首要因素。CVS在开放源界得到了广泛使用，并提供了可靠的基本功能度。有几个简单的免费GUI与CVS集成在了一起（尽管还有一些平台独立的Java GUI客户软件，但笔者见过的最佳GUI客户软件是WinCvs，可以从www.wincvs.org站点中获得它）。像Forte和Eclipse那样的流行IDE也提供CVS集成。任一专业组织在承担一个像J2EE企业解决方案那样的复杂开发项目之前，都应该已拥有了一个就绪的版本控制系统。

识别和降低风险

J2EE是一项相当新的技术。企业应用涉及到混合使用像J2EE、RDBMS和主机之类的技术，因而使互操作性成为了一个挑战。由于这些及其他原因，及早识别出风险是非常重要的。

当Java还不是很成熟时，笔者曾经参加过一家开发主机软件的软件机构的一个项目。笔者的角色是主持开发一个关键主机产品的一个Java Web接口。随着该产品逐渐展开，笔者感到印象深刻的那些主机开发人员的职业化。他们是一个既定的团队，又是各自技术方面的专家，而且他们使用这些技术已有很多年的时间。很明显，他们认为“事情总是像文档所描述的那样工作”。

该项目涉及Swing小应用程序，并需要在IE和Netscape中都能运行。我们遇到了严重的再现问题，而且为某些较严重的问题寻找解决方法花费了数天时间。最初，笔者的像“这被已知在IE中不工作”之类的解释引起了怀疑。后来，笔者想起了自己作为一名C程序员的第一次经历，以及使用早期C++实现感到的震惊。C工作了。如果某件东西不工作，那是程序员的过错。然而，早期的C++实现（不是编译器，而是C++到C转换器）偶而会从正确的语句中产生语义废话。

Java从那时起已经发展了一段很长的时间。但是，J2EE的早年出现了好像被J2EE编写者忽略的许多问题。例如，Web应用中的类装入就有严重的问题，而且这些问题需要早在2001年的几个主流产品中的激烈解决方法。

笔者所阅读过的关于J2EE的大多数图书和文章使用了太美好的颜色描绘了J2EE开发的一幅画卷。它们没能传达许多开发人员所经历过的痛苦和困难。需要重点注意的是，这类问题并不只是折磨J2EE技术。当在Microsoft车间中工作的时候，就他们的Web应用而言，笔者曾经遇到过许多不满和“已知问题”（Microsoft产品不再有“隐错（bug）”）。在刚过去的两年中，J2EE的某些东西有了很大的改进，但仍有一些路有走。

仅讨论特定产品中的隐错是没有什么用处的，因为这样的讨论在本书出版后可能已经过时。不过，承认如下事实是很重要的：将来可能有问题而且避开它们将占用一些开发时间。J2EE规范是复杂的，而实现是相当新的。问题可能会出自下列方面：

- 服务器隐错（除了在运行时行为中，也在部署和管理中）；
- J2EE规范不够完全的领域（类装入就是此类问题的一个丰富来源，下文讨论）；
- 规范中了解不充分的领域；
- 与其他企业软件应用的交互。

在最糟糕的情况下，此类问题可能会需要一个设计解决方法。例如，使用带CMP的EJB 2.0实体组件的决定可能会揭示，已选定的应用服务器的EJB QL实现无法处理一些查询的复杂性，或者EJB QL本身无法有效地满足需要。这些风险可以通过开发项目中的一个早期概念试验模型来加以防止，因为概念试验模型可以提示一个决策来避开EJB QL或选择一个不同的应用服务器。

在不太严重的情况下，比如服务器管理工具方面的一个问题，此类问题可能会涉及一个减慢开发过程的服务器隐错。

成功的风险管理取决于风险的及早发现，从而在资源已经严重地束缚于一个给定方法之前能够先采取行动。下列这些通用原则在J2EE项目风险管理中是十分有价值的：

- 尽可能早地开始处理风险。这是Unified Software Development Process（统一软件开发过程）的要点之一。我们可以采用近似的方法，而不必采用整套方法。
- 保证设计是灵活的。例如，如果我们把自己的应用设计成能让我们用另一种持久性策略来替代CMP实体组件，同时又不必重写大量业务逻辑，那么EJB QL方面的问题将会有较轻的影响。
- 在项目计划中预留出意外事件时间来处理意想不到的问题。
- 在问题变得明显时吸收更多的开发人员。这鼓励横向思考，但代价是总的开发人员工时数更大。

- 与应用服务器供应商建立一种良好的关系。一旦你确信出了一个问题，立即报告它。一种纠正方法可能正在研究之中。其他用户也可能遇到了这个问题，而且该供应商或许能够建议一个好的避开方法，如果无纠正方法可用的话。
- 学会区分由你的过错所导致的问题与不是由你的过错所导致的问题。不要低估这一点的重要性，因为它是任一重要项目需要至少一名真正J2EE专家的诸多原因之一。走上任一歧途都会显著增加查找问题所需要的时间。
- 使用Internet。Internet上有大量的联机知识。读者可以使用像Yahoo!和Google那样的正规搜索引擎来搜索它，并受益于它。无论你的问题是多么无名，某个人已经在某处的一个新闻组中报告了相似问题的可能性是非常大的。

表2.1给出了在J2EE项目中所遇到的一些重要风险，以及适合每个风险的风险降低策略。虽然我们还没有讨论这些问题中的某些问题背后所隐含的概念，但它们应该提供了有用、实际的风险管理例证。

表2.1

风 难	风险降低策略
你的开发团队缺少J2EE技能，进而有导致在项目生命周期的初期做出拙劣选择的危险，并使预计项目时间量变得不可能	购买J2EE咨询服务来启动项目。 长期雇用一名高水平的J2EE专家为项目做贡献和指导其他开发人员。 派遣关键的开发人员去参加培训
你的应用依赖于应用服务器的一个专有特性	如果该特性填补了那些J2EE规范中的一个空白，其他应用服务器将提供一个类似特性是有可能的。因此，把这个专有功能度分离到一个平台独立的抽象层中，进而保证你只需要重新实现一个或多个接口即可将该应用迁移到一个不同的服务器
你的应用服务器可能不再被支持，进而不得不迁移到另一个应用服务器	像前文所描述的，使用一个抽象层将你的应用与服务器的专有特性隔离开。 选择应用服务器时考虑服务器供应商的生存能力，并定期了解市场情况。 像前文所描述的，定期检查你的应用对J2EE规范的符合性
你的应用服务器可能没有满足你的可缩放性或可靠性需求	利用服务器供应商的帮助来构造一个能做负载测试的简单概念试验模型，以防止变换到一个不同的服务器时太迟
你的应用可能没有满足你的性能或可缩放性目标	在开发周期的早期建立应用的一个“纵向程序片”来测试它的性能
你的应用可能不能按需要进行缩放，因为它在单个服务器上虽然工作正确，但在一个聚类中表现出不正确的行为	如果聚类是一种可能性，需要考虑在所有设计决策中隐含会话管理和会话复制。 在你的应用从一个聚类的环境中被发布之前，要在一个聚类的环境中长时间地测试你的应用。 从你的服务器供应商那里寻求帮助；他们（以及他们的技术文档）将会提供与他们的聚类支持有关的重要信息，你将需要了解这些信息来达到好的结果

(续表)

风 难	风险降低策略
<p>一个服务器的故障使你的应用所需要的一个J2EE特性变得不能工作</p>	<p>尽可能早地实现应用的一个纵向程序片来检查关键技术的实现。</p> <p>将问题报告给服务器供应商，并希望得到帮助或一个补丁程序。</p> <p>修改应用设计来避免有问题的技术。</p> <p>如果还可能，转换到一个出众的应用服务器</p>
<p>你的应用需要第三方库（比如一个特定的XML库），而这些库可能会与你的应用服务器所携带的库发生冲突</p>	<p>这个风险必须尽可能早地通过一个纵向程序片的实现来解决。寻求服务器供应商（以及他们的技术文档）的帮助来保证兼容性（例如，配置类装入来避免该冲突或许是可能的）</p>
<p>一个使用EJB和Web模块的集成式J2EE应用遇到了使生产效率下降的类装入问题。当同一个类由两个类装入器装入时，如果被比较，这两个副本被认为是不同的类；当一个类依赖于已由一个类装入器装入的其他类，并且这个类的类装入器对后面这个类装入器又是不可见的时候，<code>ClassNotFoundExceptions</code>可能会被遇到。例如，当一个Web应用中所使用的、但实际上由EJB类装入器装入的一个类试图装入由WAR类装入器装入的类时（在大多数服务器中，EJB类装入器无法看见WAR类装入器），就可能会发生这种异常。</p> <p>类装入在第14章中将较详细地讨论</p>	<p>了解Java类装入分级结构（在<code>java.lang.ClassLoader</code> Javadoc中有详细定义）和你的目标应用服务器的类装入体系结构。令人遗憾的是，类装入策略随着服务器的不同而有所不同，进而意味着这是可移植性在其中失败的一个方面。</p> <p>小心封装部署单元来保证类被正确的类装入器（比如WAR或EJB类装入器）装入。这要求仔细部署生成脚本来保证类被包含在正确的部署单元中，而不是被包含在所有部署单元中。</p> <p>特别仔细地考虑哪个类装入器将装入按名装入其他类的类。编程到接口，不要编程到具体的类。这么做将使得把实现类的组保持在正确的类装入器中变得更容易。</p> <p>尽可能早地实现一个纵向程序片来核实类装入没有引起任何风险。</p> <p>如果发生堆处理的问题，考虑EJB的使用是否确实必要。类装入问题在Web应用中比较简单。</p> <p>作为最后一种手段，考虑把整个应用中的必需类添加到服务器的全局类路径上。这违反了J2EE规范，但可以节省大量时间</p>
<p>应用部署引起了不必要的故障停机</p>	<p>掌握你选定的应用服务器上的部署过程。</p> <p>制定一个满足你的需要的发布管理策略</p>

小结

在本章中，我们考虑了J2EE开发项目中要做出的最重要的选择的一部分，它们不同于我们在第1章中所讨论的体系结构决策。我们讨论了如下内容：

- 如何选择应用服务器。J2EE平台的优点之一是它允许从J2EE规范的竞争实现中选择，其中每个实现都具有不同的优缺点。选择正确的应用服务器将对一个项目的结果有重要的影响。我们讨论了选择应用服务器的一些主要标准，进而强调了考虑具体需求而非市场宣传的重要性。我们探讨了在项目周期的早期选择应用服务器的重要性，

目的是为了避免把资源浪费在熟悉多个服务器上。我们考虑了总所有权成本的问题，而许可成本只是其中的一部分。

- 管理一家企业中不同技术的组合。虽然不同技术的不必要激增常常会使维护得成本更高，但是J2EE不是企业软件开发中所有问题的最佳解决方案，明白这一点十分重要。我们应该准备使用其他技术来补充J2EE技术。
- 围绕J2EE可移植性的实际问题。我们知道了如何通过定期地运行随Sun的J2EE Reference Implementation所提供的有效性验证工具来保证不会无意识地违反J2EE规范，以及如何保证应用设计保持可移植性，即使我们有很好的理由使用目标服务器的专有特性。
- 发布管理规则。我们知道了拥有不同的“开发”、“测试”和“生产”环境的重要性，以及拥有一个得到好评的发布管理策略的重要性。
- 为J2EE项目组建和管理团队的问题。我们对照一种更民主的方法分析了使用一名“总设计师”的意义，并考虑了两种常见的团队结构：“纵向”结构——使用多面手来实现整个用例，以及“横向”结构——让开发人员重点关注个别专门技术领域。我们考虑了“横向”团队结构中一种可能的角色划分。
- 开发工具。我们简要地审视了J2EE开发人员可以利用的各种工具。我们重点强调了Ant生成工具的重要性，它现在是Java开发的一个事实上标准。
- 风险管理。我们已经看到，成功的风险管理基于及早识别和动手处理风险。我们讨论了几个总的风险管理策略，并考虑了J2EE项目的几个实际风险，以及管理它们的策略。

由于本书是一本以实用为重点的图书，所以笔者没有讨论选择一种开发方法论，或决定何时需要一种开发方法论。不过，这是另外一个重要的选择。我们已经知道及早发现风险的重要性。笔者建议为J2EE开发使用一种着重这种开发的方法论。Rational Unified Process和Extreme Programming（XP）都满足这一要求。就我个人而言，笔者更喜欢“轻型”或“灵巧”方法论（请参见`http://agilealliance.org/principles.html`），尽管形式的正确程度往往会使项目变得更大。笔者为不熟悉这些方法论的读者推荐如下资源作为起点：由Addison Wesley出版的“The Unified Software Development Process”（ISBN 0-201-57169-2），以及`http://www.extremeprogramming.org`（“Extreme Programming: A Gentle Introduction”）。

在下一章中，我们将讨论J2EE应用的测试。测试是整个软件开发周期中的一个重要问题，而且这两种方法论中都对测试给予了特别的重视。

第3章 J2EE应用的测试

从第2章中可以看到，在项目生命周期的初期所做的决策会决定一个项目的成功或失败。测试则是我们必须从项目一开始时就制定一个策略和设立良好准则的另一个关键领域。

测试常常被看做在大部分开发完成之后就可以着手进行的一项令人厌倦的活动。没有人真地相信这是一种好方法，但是，如果从项目开始时就没有连贯的测试策略，这是很平常的结局。大多数开发人员都清楚这种难应付的测试所带来的诸多问题；例如，错误出现的时间越长，纠正它们的代价就会越来越迅速地增大。

在本章中，我们考虑一种积极的测试方法。我们将会看到，测试是我们应该做的一件事情，不仅仅是为了不担心不做测试的后果，也是因为测试可以用来改进我们开发代码的方式。如果我们把测试看做开发过程的一个不可分部分，我们不仅能够提高应用的质量和使它们更易于维护，而且还能提高工作效率。

测试应该在整个开发周期内多次进行。测试决不应该是一件事后再做的事情。把测试集成到开发过程中会有许多好处。

测试企业应用会带来许多挑战：

- 企业应用通常要依靠数据库之类的资源，而这些资源应该在任何测试策略中都需要考虑到。
- 测试Web应用会很困难。Web应用不会暴露我们能够测试的简单Java接口，而且单元测试会被Web容器上的Web层构件的相关性搞得很复杂。
- 测试分布式应用很困难。这种测试可能需要许多计算机，而且模拟一些故障原因可能也很困难。
- J2EE构件（尤其是EJB）严重依赖于服务器基础结构。
- J2EE应用可能涉及许多体系结构层。我们必须测试每一层都工作正常，而且还必须从总体上做应用的接受性测试。

在本章中，我们将讨论上述这些挑战和满足它们的方法。我们将了解：

- 测试目标和概念。
- 测试的Extreme Programming (XP) 方法，这种方法基于测试优先开发 (test-first development)。XP把测试抬高成开发过程的中心部分。测试被看做必不可少的应用交付物。测试在代码之前先编写，并总是保持最新。无论我们是否考虑完全采用XP，这总是一种非常有效的方法。虽然所有好程序员都经常测试他们的代码，但从一种特定方法延续到一种较正规方法确实有好处，而在后一种方法中，测试定义明确，并且是可轻松地重复的。
- JUnit测试框架，它为测试策略提供一个好的基础。JUnit是一个简单但非常有效的工具，它非常易于学会，并能使测试的编写毫不费劲。

- Cactus J2EE测试框架，它基于JUnit来使EJB之类的J2EE构件能在应用服务器内被测试。
- 测试Web接口的技巧。
- 使测试自动化以便应用的所有测试都能用单个操作来运行的重要性。我们将会看到Ant生成工具怎样才能用来自动化JUnit测试。
- 测试的补充方法（比如断言），我们可以把这些方法用做一个集成QA测试的一部分。

测试能达到什么目的

测试不可能保证一个程序是正确的。但是，测试可以增强我们确信一个程序做我们期望它做的事情的自信度。“隐错（bug）”常常反映了程序员对代码应该真正做什么的疏忽。当我们对一个程序应该做什么的了解逐渐深入时，我们就能够编写出更符合程序要求的测试。

知道测试的局限性是很重要的——测试不一定会使并发问题显露出来。在这里，1盎司的预防确实值1磅的治疗（例如，测试有可能无法发现与多个线程正并发地修改的一个小服务程序中的实例数据有关的问题）。但是，这样的代码在生产中肯定会出现故障，并且能力强的J2EE开发人员决不应该编写出这样的代码。

一个隐错出现的时间越长，纠正它的代价就会越高。有一项研究表明，最后纠正一个隐错的代价10倍于在该隐错被发现前经历的每一个阶段（需求、设计、实现或最后发布）纠正它所花费的代价。测试绝代替不了编程前的仔细思考；测试绝捕捉不了所有隐错。

虽然本章的焦点是测试代码，但需要特别注意的一点是，从需求分析开始就有一个健全的QA策略是非常重要的。

定义

下面让我们来简要地定义一下本章中将要讨论的概念：

- **单元测试（Unit test）**

单元测试只检查功能度的单个单元。在Java中，这是常常是一个单独的类。单元测试是测试粒度的最精细级别，并且应该测试一个类中的每个方法都满足它的已签名契约。

- **测试覆盖度（Test coverage）**

这是指应用代码被测试（通常由单元测试来完成）的比例。例如，我们可能打算检查代码的每一行至少被一个测试所执行，或者代码中的每个逻辑分支都被测试了。

- **黑箱测试（Black-box testing）**

这种测试只检查被测试类的公用接口。它不关心实现细节。

- **白箱测试（White-box testing）**

这种测试只检查被测试类的内部细节。在一个Java上下文中，白箱测试只检查私用和

受保护的数据和方法。它不仅检查该类是否做我们需要它做的事情，而且还检查它怎样做。笔者不建议白箱测试（稍后详细讨论这方面的内容）。白箱测试有时叫做“玻璃箱测试”。

- **回归测试（Regression test）**

这种测试核实代码在修改和添加发生之后仍做它以前所做的事情。如果给定足够的覆盖度，单元测试能够用做回归测试。

- **边界值测试（Boundary-value test）**

这种测试检查被测试代码应该能够处理不寻常或极端情况（比如一个方法的意外空值参数）。

- **接受度测试（Acceptance test）（有时叫做功能测试（Functional test））**

这种测试是以顾客角度为出发点的测试。接受度测试所关心的是该应用怎样满足业务需求。虽然单元测试检查一个应用的每个部分怎样做它的工作，但接受度测试忽略实现细节，只使用对用户（或XP术语中的顾客）有意义的概念来检查最后的功能度。

- **负载测试（Load test）**

这种测试检查一个应用在负载逐渐增加（比如模拟越来越多的用户）时的行为。负载测试的目标是：证实该应用能够应付它在生产中预计会遇到的负载，并确定它能够支持的最大负载。负载测试通常将被运行一段很长的时间以检查稳定性。负载测试可能会使并发问题暴露出来。吞吐量目标是一个应用的非功能需求的一个重要部分，并且应该被定义为业务需求的一部分。

- **压力测试（Stress test）**

这种测试超出负载测试，把应用上的负载增加到超过计划的限制。目的不是模拟预期的负载，而是致使应用失败或显示出无法接受的响应时间，因而从吞吐量和稳定性的角度证实它的薄弱环节。这可以提供设计或代码方面的改进，并确定使该应用过载能否导致许多错误的行为，比如数据丢失或崩溃。

正确性的测试

现在来仔细看一看与测试应用的正确性（即检查应用满足其功能需求）有关的问题和技巧。

测试的XP方法

第2章中曾经提过Extreme Programming（XP），这是一种强调经常性集成和全面单元测试的方法论。XP中与测试有关的重要原则和惯例如下所示：

- 在编写代码之前先编写测试。
- 所有代码都必须有单元测试，单元测试可以用单个操作来自动运行。
- 当报告一个隐错时，在尝试纠正该隐错之前，先创建测试来重现该隐错。

XP的先驱们没有发明测试优先开发 (test-first development)。但是，他们一直推广它，并在大家都理解后把它与XP联系在了一起。在其他方法论当中，Unified Software Development Process也强调整个项目生命周期内的测试。

要从上述这些思想中受益，不必完整地采用XP。下面让我们来看一看它们的好处和含义。

在编写代码之前先编写测试案例（即测试优先开发）具有许多好处：

- 测试案例等于一个规范，并提供附加的文档。一个使用中的规范（能够被日常或更经常地检查其符合性）比一本厚厚的需求文档中一个无人阅读或更新的规范更有价值。
- 它鼓励对需求的理解。它将在任何时间已被浪费之前，暴露出与该类或构件的功能度有关的不可靠性，并要求解决不可靠性。实施后的修改对其他构件将不产生影响。如果没有了解一个构件应该做什么，编写一个测试案例是不可能的；如果缺乏了解，则有可能会把大量编程时间浪费在该构件上。

一个常见的例子是传给方法的空值参数。如果不考虑这种可能性，编写一个方法是很容易的，但结果是一个带空值的调用会产生意想不到的结果。一个合适的测试集将包括带空值参数的测试案例，以保证仅当该方法在空值上的行为得到了确定并被归档之后才被编写。

- 在整个项目生命周期内，测试案例很有可能被看做是至关重要的，并被时常更新。
- 为已有代码编写测试案例要比编写代码之前和期间编写测试案例困难得多。实现应用代码的开发人员应该对它应该做什么有完全的了解（因此对如何测试它也应该有完全的了解）；事后编写测试案例将始终像演警察抓小偷。因此，测试优先开发是最大化测试覆盖度的最佳方式之一。

一个测试优先方法并不意味着在编写一个类之前，开发人员就必须先花费整天时间为这个类编写所有可能的测试。测试案例和代码一般是在同一时期编写的，但必须按照先测试案例后代码的次序编写。例如，在完整地实现一个具体方法之前，我们可能先为这个方法编写测试案例。在这个方法被实现完毕和这些测试成功之后，我们继续到另一个方法。

当我们在编写应用代码之前编写测试时，应该确认这些测试在实现所需功能度之前失败。这使我们能够检查测试案例和验证测试覆盖度。例如，我们可能会为一个方法编写测试，然后编写这个方法的一个返回空值的次要实现。现在，我们可以运行该测试案例，并见到它失败（如果它不失败，就是我们的测试集有毛病）。

如果我们在代码之前编写测试，第二条规则（所有代码都应该有单元测试）将自动得到兑现。测试所有类有许多好处：

- 在单个操作中使测试自动化并验证所有代码正像我们所期望的那样工作是可能的。这和工作完美不是同一回事情。随着一个项目的进展，我们将会较详细地了解到我们怎么让代码工作，以及怎样相应地添加测试案例。
- 我们能够放心地添加新的功能度，因为我们拥有将指出我们是否已破坏了任何现有功能度的回归测试。因此，我们能够快速而轻松地运行所有测试是很重要的。
- 再加工 (Refactoring) 对开发人员的压力不太大，对总体功能度的威胁也较小。这保

证了应用代码的质量在整个项目生命周期内都保持很高（例如，根本没有必要担心有几分管用的吓人的类，仅仅因为有那么多其他类依赖于它）。同样，如果必要，放心地优化一个类或子系统也是可能的。我们有一种方法可用来证明优化后的代码做它以前所做的事情。如果使用全面的单元测试覆盖度，开发周期的后面阶段有可能会变得压力更小。

如果我们有一个全面的单元测试集，单元测试将只为再加工和隐错纠正提供一个安全的基础。一种不认真的单元测试方法将提供有限的好处。

我们也可以给隐错纠正应用一种测试优先方法。每当一个隐错被报告（或通过不同于测试案例的其他方式变得明显）时，在编写任何代码来纠正该隐错之前，应该编写一个缺陷测试案例（由该隐错引起的缺陷）。结果将验证该隐错既已经得到纠正，同时又没有影响以前测试所覆盖的功能度，而且放心地测出该隐错将不再出现。

在编写代码之前先编写单元测试，在整个项目生命周期内更新它们。隐错报告和新的功能度应该首先提示人们编写和执行缺陷测试，以展现该应用做什么与它应该做什么之间的不匹配。

测试优先开发是保证全面测试覆盖度的最佳方法，因为给已有代码编写全面测试更困难。

请记住，要使用测试缺陷来改进错误消息与处理。如果一个测试失败，并且什么东西有毛病马上又看不出来，应该试着首先使该问题变得明显（通过改进后的错误处理和消息），然后纠正它。

所有这些规则把测试的大部分责任转移到了开发团队身上。在从事软件开发的大型传统组织中，有一个专业化的测试团队负责测试，而开发人员生产要被测试的代码。有一个用于QA专家的地方；开发人员并不总是最擅长编写测试案例（尽管他们能够学会）。但是，开发与技术测试之间的区别是人为的。另一方面，接受度测试可能需要在开发团队之外进行，至少是部分地在开发团队之外进行。

开发与测试角色之间不应该有人为的划分。应该鼓励开发人员把编写好的测试案例看做一个重要的技能。

编写测试案例

为了享有全面单元测试的各种好处，我们需要知道如何编写有效的测试。下面让我们来考虑几个重要的问题，并了解一下帮助简化和自动化测试创作的几个Java工具。

什么产生一个好的测试案例

编写好的测试案例需要实践。我们对实现（或可能的实现，如果我们首先开发测试）的了解可能会暗示潜在的问题领域；不过，我们还必须培养从开发人员角色外面进行思考的能力。尤其是，把缺陷测试的编写看做一项成果而不是一个问题是很重要的。测试的常见主题将包括：

- 测试最常用的执行路径（这些路径从应用使用案例中看应该是很明显的）。
- 测试意外参数上发生什么。
- 测试当被测构件遇到起源于它们所使用的构件中的错误时发生什么。

我们将从实用的角度出发，简单地看一看测试案例的编写。

把测试案例创作和维护当做一项核心任务

编写所有测试案例确实需要时间；而且，由于它们对系统的文档非常重要，所以它们必须被仔细编写。测试集在整个项目生命周期内连贯地反映本应用的需求是至关重要的。和大多数代码一样，随着时间的推移，测试集往往堆积了许多垃圾。这会非常危险。例如，不再适用的旧测试会使测试代码和应用代码都变得复杂（而应用代码仍将被要求通过这些测试）。测试代码（和应用代码一样）处于版本控制之下。对测试代码的更改和对应用代码的更改一样重要，因此可能需要服从一个正式过程。虽然习惯测试优先开发需要一些时间，但在整个项目生命周期内，好处会渐渐增多。

单元测试

我们已经相信单元测试是重要的。那么，我们在J2EE项目中应该怎样着手进行单元测试呢？

main()方法

在Java中，单元测试的传统方法是在每一个被测类中都编写一个main()方法。不过，这不必要地增加了源文件的长度，增添了编译后的字节代码，而且常常引入对其他类的多余依赖性，比如该类的代码中所引用的那些接口的实现。

一种更好的方法是使用另外一个类，并且这个类具有一个像XXXXMain或XXXXTest这样的名称，并仅含有main()方法和它所需要的任何支持代码，比如从语法上分析命令行参数的代码。现在，我们甚至能把单元测试类放入一个并行源树中。

但是，使用main()方法来运行测试仍是一种特定的方法。正常情况下，可执行类将产生控制台输出，而开发人员必须通过阅读这个输出来确定该类成功了还是失败了。这是非常费时的，通常意味着脚本化main()方法测试和一次检查几个结果是不可能的。

使用JUnit

有一种比main()方法测试更好的方法，这种方法允许自动化。JUnit是一个简单的开放源工具，现在是单元测试Java应用的事实标准。JUnit的使用和安装非常容易，实际上根本不存在学习困难。JUnit是由Erich Gamma（Gang of Four的成员之一）和Kent Beck（XP的先驱）编写的。读者可以从<http://www.junit.org/index.htm>站点中下载JUnit。这个站点还含有许多用于JUnit的附加软件和关于使用JUnit的帮助性文章。

JUnit被设计成按一种一致的方式报告成功或失败，不需要翻译结果。JUnit对照一个测试附属工具（即一个被测对象集合）执行测试案例（个别测试）。JUnit提供了初始化和（如果必要）发布测试附属工具的便利方法。

JUnit框架是可自定义的，但创建JUnit测试通常只涉及下列这些简单步骤：

1. 创建junit.framework.TestCase的一个子类。
2. 实现一个接受一个串参数并用这个串来调用超类构造器的公用构造器。如果必要，这个构造器随后也可以加载被使用的测试数据。当测试还没有改变测试附属工具的状态时，它也可以执行应该为整个测试集只执行一次的初始化。这在附属工具的创建速度缓慢时是非常方便的。
3. 这是一个可选步骤。覆盖setUp()方法来初始化所有测试类所使用的那些对象和变量（即附属工具）。并不是所有测试案例都将需要这么做。个别测试可能会创建和破坏它们自己的附属工具。请注意，setUp()方法在每个测试案例的前面和tearDown()方法的后面被调用。
4. 这是一个可选步骤。覆盖tearDown()方法来释放setUp()中所需要的资源或把测试数据反转成一个干净状态。如果测试案例可能更新持久数据，这一步将是必需的。
5. 给该类添加测试方法。请注意，我们不必实现一个接口，因为JUnit使用反射，并自动检测测试方法。测试方法通过它们的签名来识别，而签名必须具有public test <Description>()形式。测试方法可能会抛出任何检查到或未能检查到的异常。

JUnit的价值和优雅在于它允许我们把多个测试案例组合成一个测试集（Test suite）的方式。例如，junit.framework.TestCase类的一个对象可以被构造为带有多个测试方法作为一个参数的类。它将自动识别测试方法，并把它们添加到该测试集上。这举例说明了反射的一个有效使用，以保证代码使其本身保持最新。我们将在第4章中详细讨论反射的用法。当我们编写新的测试或删除旧的测试时，不必修改任何核心测试列表——避免了潜在的错误。TestSuite类提供了一个API，该API能使我们轻松地把另外的测试添加到一个测试集上，以便多个测试能被组合起来。

我们可以使用JUnit所提供的、执行测试和显示测试结果的许多测试运行器（Test Runner）。最常用的两个测试运行器是文本运行器和Swing运行器，而后者显示一个简单的GUI。笔者推荐从Ant中运行JUnit测试（我们将在下文中讨论这方面的内容），也就是说，使用文本接口。Swing测试运行器提供了所有测试都通过时的著名绿色条码，而文本输出提供了一个更棒的审核记录。

测试方法在被测对象上调用操作，并含有基于比较预期结果与实际结果的断言。断言应该含有解释什么发生故障的消息，以方便故障情况下的调试。JUnit框架提供了几个可供测试案例使用的便利断言方法，它们具有如下所示的签名：

```
public void assertEquals(java.lang.String message, boolean condition)
public void assertSame(message, Object expected, Object actual)
```

不成功的断言等于测试失败，测试方法所遇到的未捕获异常也是如此。最后这个特性是非常方便的。我们不希望在测试案例中实施大量错误处理，因为try/catch块会迅速生产大量理解起来可能会很困难的代码。如果一个异常只是反映某个东西有毛病，而不是反映该API在一个给定输入下的意外行为，那么较简单的方法不是捕获它，而是让它引起一个测试失败。

请看一看下面这个JUnit使用例子，本例也举例说明了正在运转中的测试优先开发。我

们需要一个StringUtil类中的下列方法，该方法获取一个像“dog,cat,rabbit”那样的逗号分隔(CSV)列表，并输出一个由各串作为元素的数组，比如对于前面的输入，将输出“dog”、“cat”、“rabbit”。

```
Public static String [] commaDelimitedListToStringArray(String s)
```

不难看出，我们需要检查下列条件：

- 普通输入——用逗号分隔的单词和字符。
- 包含其他标点符号的输入，以保证它们不被作为分隔符来对待。
- 一个空值串。该方法应该在空值输入上返回空数组。
- 单个串(没有任何逗号)。在这种情况下，返回值应该是一个只含有单个串的数组，其中该串等于输入串。

使用一种测试优先开发方法，第一步是实现一个JUnit测试案例。这将展开junit.framework.TestCase。由于我们正在测试的这个方法是静态的，所以没有必要通过覆盖setUp()方法来初始化一个测试夹具。

我们声明这个类，并提供必需的构造器，如下所示：

```
public class StringUtilTestSuite extends TestCase {  
    public StringUtilTestSuite(String name) {  
        super(name);  
    }
```

现在，我们为上述4个案例的每一个添加一个测试方法。整个测试类如下所示，但首先来看一看最简单的测试：检查一个空值输入串上行为的方法。使用短测试方法名没有什么好处，因此我们将使用test<Method to be tested><Description of test>形式的名称：

```
public void testCommaDelimitedListToStringArrayNullProducesEmptyArray() {  
    String[] sa = StringUtil.commaDelimitedListToStringArray(null);  
    assertTrue("String array isn't null with null input", sa != null);  
    assertEquals("String array length == 0 with null input", sa.length == 0);  
}
```

请注意多个断言的使用(它们将提供失败情况下的尽可能多的信息)。事实上，第一个断言不是必需的，因为若第一个断言失败，那么第二个断言将总是不能计算(具有一个NullPointerException，而该异常会引起一个测试失败)。但是，分开它们将提供失败情况下的更丰富信息。

首先编写了我们的测试，然后实现返回null的commaDelimitedListToStringArray()方法。

接下来，我们运行JUnit。下面，我们将看一看如何运行JUnit。正如预计的那样，所有测试都失败了。

现在，我们以最简单、最明显的方式实现该方法：使用核心的Java类java.util.StringTokenizer。由于它不需要费多大劲，所以我们实现了一个更通用的delimitedListToStringArray()方法，并把逗号看做一个特殊情况：

```

public static String[] delimitedListToStringArray(
    String s, String delimiter) {

    if (s == null) {
        return new String[0];
    }

    if (delimiter == null) {
        return new String[] { s };
    }

    StringTokenizer st = new StringTokenizer(s, delimiter);
    String[] tokens = new String[st.countTokens()];
    System.out.println("length is " + tokens.length);

    for (int i = 0; i < tokens.length; i++) {
        tokens[i] = st.nextToken();
    }
    return tokens;
}

public static String[] commaDelimitedListToStringArray(String s) {
    return delimitedListToStringArray(s, ",");
}

```

我们的所有测试都通过了，而且我们认为自己已经充分地定义和测试了所需的行为。

以后的某个时候会出现这样一种情况：对于像“a,,b”这样的输入串，这个方法没有像预计的那样工作。我们想让这个方法产生一个长度为3且含有串“a”、空串和串“b”的串数组。这是一个隐含错误，所以我们需要编写一个展现该隐错并在现有代码上失败的新测试方法：

```

public void testCommaDelimitedListToStringArrayEmptyStrings() {

    String[] ss = StringUtils.commaDelimitedListToStringArray("a,,b");
    assertTrue("a,,b produces array length 3, not "
        + ss.length, ss.length == 3);
    assertTrue("components are correct",
        ss[0].equals("a") && ss[1].equals("") && ss[2].equals("b"));
    // Further tests omitted
}

```

通过查看`delimitedListToStringArray()`方法的实现，我们可以明显地看出，`StringTokenizer`库类没有提供我们需要的行为。因此，我们重新实现该方法，进而通过亲自做标记解析（Tokenizing）来提供预期的结果。在两次测试运行中，我们最终得到了`delimitedListToStringArray()`方法的下列版本：

```

public static String[] delimitedListToStringArray(
    String s, String delimiter) {

    if (s == null) {
        return new String[0];
    }

    if (delimiter == null) {
        return new String[] { s };
    }

    ...
}

```

```

List l = new LinkedList();
int delimCount = 0;
int pos = 0;
int delpos = 0;

while ((delpos = s.indexOf(delimiter, pos)) != -1) {
    l.add(s.substring(pos, delpos));
    pos = delpos + delimiter.length();
}

if (pos <= s.length()) {
    // Add remainder of String
    l.add(s.substring(pos));
}

return (String[]) l.toArray(new String[l.size()]);
}

```

虽然Java具有相当差的串处理能力，而串操纵又是引起隐错的一个常见原因，但我们可以放心地做这种再加工（refactoring），因为我们有回归测试来验证新代码表现得和原版本一样（而且还确保了新测试展示了该隐错）。

下面是这些测试案例的完整代码。需要注意的是，这个类包含了一个私用方法testCommaDelimitedListToStringArrayLegalMatch(String[] components)，该方法从传递给它的串数组中产生一个CSV格式的串，并验证commaDelimitedListToStringArray()方法对这个串的输出匹配于输入数组。大多数公用测试方法都使用这个方法，结果它们变得更简单了（尽管这个方法名用test开头，但由于它获取一个参数，所以它将不被JUnit直接调用）。在测试类中做这种基础结构方面的投入通常是值得的。

```

public class StringUtilsTestSuite extends TestCase {

    public StringUtilsTestSuite(String name) {
        super(name);
    }

    public void testCommaDelimitedListToStringArrayNullProducesEmptyArray() {
        String[] sa = StringUtils.commaDelimitedListToStringArray(null);
        assertTrue("String array isn't null with null input", sa != null);
        assertEquals("String array length == 0 with null input", sa.length == 0);
    }

    private void testCommaDelimitedListToStringArrayLegalMatch(
        String[] components) {

        StringBuffer sbuf = new StringBuffer();
        // Build String array
        for (int i = 0; i < components.length; i++) {
            if (i != 0) {
                sbuf.append(",");
                sbuf.append(components[i]);
            }
        }
        System.out.println("STRING IS " + sbuf);

        String[] sa =
            StringUtils.commaDelimitedListToStringArray(sbuf.toString());
        assertTrue("String array isn't null with legal match", sa != null);
        assertEquals("String array length is correct with legal match: returned " +
                    sa.length + " when expecting " + components.length + " with String [" +
                    sbuf.toString() + "]", sa.length == components.length);
        assertEquals("Output equals input", Arrays.equals(sa, components));
    }
}

```

```

public void testCommaDelimitedListToStringArrayMatchWords() {
    // Could read these from files
    String[] sa = new String[] { "foo", "bar", "big" };
    testCommaDelimitedListToStringArrayLegalMatch(sa);

    sa = new String[] { "a", "b", "c" };
    testCommaDelimitedListToStringArrayLegalMatch(sa);

    // Test same words
    sa = new String[] { "AA", "AA", "AA", "AA", "AA" };
    testCommaDelimitedListToStringArrayLegalMatch(sa);
}

public void testCommaDelimitedListToStringArraySingleString() {
    String s = "woeirqupoiewuropqiewuorpqiwueopriquwopeiurqopwieur";
    String [] sa = StringUtils.commaDelimitedListToStringArray(s);
    assertTrue("Found one String with no delimiters", sa.length == 1);
    assertTrue("Single array entry matches input String with no delimiters",
        sa[0].equals(s));
}

public void testCommaDelimitedListToStringArrayWithOtherPunctuation() {
    String[] sa = new String[] { "xcvwert4456346&.", "///", ".!", ".," ";" };
    testCommaDelimitedListToStringArrayLegalMatch(sa);
}

/** We expect to see the empty Strings in the output */
public void testCommaDelimitedListToStringArrayEmptyStrings() {
    String[] ss = StringUtils.commaDelimitedListToStringArray("a,,b");
    assertTrue("a,,b produces array length 3, not " + ss.length,
        ss.length == 3);
    assertTrue("components are correct",
        ss[0].equals("a") && ss[1].equals("") && ss[2].equals("b"));
    String[] sa = new String[] { "", "", "a", "" };
    testCommaDelimitedListToStringArrayLegalMatch(sa);
}

public static void main(String[] args) {
    junit.textui.TestRunner.run(new TestSuite (StringUtilsTestSuite.class));
}
}

```

请注意main()方法，该方法构造一个新的TestSuite（如果给定当前类），并使用junit.textui.TextRunner类运行它。让每个JUnit测试案例都提供一个main()方法是很方便的，尽管不是必需的（由于这样的main()方法调用JUnit本身，所以它们也可以使用Swing测试运行器）。

JUnit不需要任何特殊配置。我们只需保证在测试时junit.jar（含有所有JUnit库）都在类路径上即可。

我们有几个运行JUnit测试集的选择。JUnit被设计成允许我们实现多个脱离实际测试的“测试运行器”（我们将在后面看到几个允许测试集在J2EE服务器内执行的特殊测试运行器）。一般说来，我们将使用下列方法之一来运行JUnit测试：

- 从命令行上用一个main()方法来运行测试类。
- 通过一个提供JUnit集成的IDE来运行测试。和通过一个main()方法的调用一样，这通常只允许我们一次运行一个测试类。

- 作为应用生成过程的一部分来运行多个测试。通常，这是使用Ant来实现的。这是生成过程的一个基本部分，我们将在本章快结束处的“测试的自动化”一节中讨论这方面的内容。

虽然使用Ant的自动化是集成测试到应用生成过程中的关键，但是与一个IDE的集成会更方便，因为我们在个别类上工作。图3.1显示了怎么才能从Eclipse IDE中调用上面所讨论的JUnit测试集。

单击工具栏上的Run（运行）按钮后，我们从Run With子菜单上的启动器（Launcher）列表中选择JUnit。



图3.1

Eclipse打开一个对话框来显示测试的进度和结果。一个绿色条表示成功；一个红色条表示失败。任何错误或失败都被列举出来，其中它们的栈跟踪显示在Failure Trace窗格中（见图3.2所示）。

编写测试的原则

在见过了运转中的JUnit之后，下面让我们退后一点，并看几个编写测试的有效原则。虽然我们将借助JUnit来讨论这些原则的履行，但这些原则适用于我们可能选用的任何测试工具。

编写测试到接口

在任何可能的地方，都应该把测试编写到接口，而不应该编写到类。编程到接口而非类是好的OO设计习惯，而且测试应该反映出这一点。不同的测试集可以容易地被创建成对照一个接口的实现运行那些相同的测试（参见后面的“继承性与测试”一节）。



图3.2

不要自寻烦恼地测试JavaBean属性

测试属性获取器和设置器通常是没有必要的。开发这样的测试通常是浪费时间。而且，使用没有用处的代码来增大测试案例会使阅读和维护变得更困难。

最大化测试覆盖度

测试优先开发是保证我们最大化测试覆盖度的最佳策略。但有些时候，工具可以帮助验证我们已经满足了我们的测试覆盖度目标。例如，一个像Strake的JProbe Profiler（第15章中讨论）那样的概貌工具可以用来检查一个被测应用的执行路径，并确定什么代码已得到（和未得到）执行。

像JProbe Coverage（也是JProbe Suite的一部分）那样的专业化工具使这项工作变得更容易。JProbe Coverage可以分析一个或多个测试运行以及应用代码库来产生未得到执行的方法列表与源代码行。

当给还没有测试集的代码开发一个测试集时，在这样一个工具中的适当投入可能是值得的。

不要依赖测试案例的排列顺序

当使用反射来识别要执行的测试方法时，JUnit不保证它运行测试的顺序。因此，测试不应该依赖于先前已得到过执行的其他测试。如果顺序是很重要的，通过编程把测试添加到一个TestSuite对象是可能的。它们将按照添加它们的顺序得到执行。不过，最好是适当地使用setUp()方法来避免顺序问题。

避免副效应

由于同样的原因，测试时避免副效应也很重要。当一个测试以一种可能会影响后续测试的方式修改被测系统的状态时，就会出现一个副效应。对数据库中持久数据的修改也是潜在的副效应。

从类路径中读取测试数据，不要从文件系统中读取

测试易于运行是非常重要的。需要做的配置应该是最少的。当运行一个测试集时，引起问题的一个常见原因是让测试从文件系统中读取它们的配置。使用绝对文件路径在代码被寄存到一个不同位置时将会引起问题；不同的文件位置和路径标准（比如\home\rodj\tests\foo.dat，或C:\Documents和Settings\rodj\foo.dat）会把测试束缚于一个特定的操作系统。这些问题可以通过使用Class.getResource()或Class.getResourceAsStream()方法从类路径中装入测试数据来加以避免。一般说来，最好是把必要的资源和使用它们的那些测试类放在同一个目录中。

避免测试案例中的代码重复

测试案例是应用的一个重要部分。和应用代码一样，测试案例含有的代码重复越多，它们含有错误的可能性就越大。测试案例含有的代码越多，它们要编写的零碎工作就越多，并且它们将被编写的可能性就越小。要通过在测试基础结构中做一些工作来避免这个问题。我们已经通过几个测试案例见识过一个私用方法的使用，这个私用方法极大地简化了使用它的那些测试方法。

何时应该编写“桩基”类

有些时候，我们希望测试的类依赖于测试时提供起来不容易的其他类。如果我们遵守了好的编程原则，任何这样的依赖性将依附于接口，而不依附于类。

在J2EE应用中，这样的依赖性将时常依附于应用服务器所提供的实现类。但是，我们经常希望能够在服务器的外面测试代码。例如，为用做EJB层中的一个Data Access Object (DAO) 而设计的一个类可能需要一个javax.sql.DataSource对象来提供指向一个RDBMS的连接，但是可能根本不具有依附于一个EJB容器的其他依赖性。我们可能需要在一个J2EE服务器外面测试这个类。

在这样的情况下，我们可以编写被测类所需要的接口的简单桩基实现。例如，我们可以实现一个不太重要的javax.sql.DataSource，并且它总是返回一个指向一个测试数据库的连接（我们不必实现我们自己的连接池）。特别有用的桩基实现（比如一个测试DataSource）是通用的，并且能够用在多个测试案例中，从而使编写和运行测试变得更容易。我们也可以使用目前还不可用，或者说还没有被编写出来的应用对象的桩基实现（例如，以便使Web层上的开发能够与EJB层的开发并行前进）。

本书下载中的/framework/test目录包含了几个有用的通用测试类，其中就包括能让我们不用J2EE服务器也能测试DAO的jdbc.TestDataSource类。

在实现没有包括太多工作的桩基化对象时，这个策略提供真正的价值。最好是避免编

写过度复杂的桩基实现。如果桩基化对象开始有对于其他桩基化对象的依赖性，我们应该考虑可替代的测试策略。

继承性与测试

我们需要考虑被测类的继承性分级结构的含义。一个类应该传递与它的超类相关的所有测试和它所实现的接口，这是“**Liskov Substitution Principle (Liskov替换定律)**”的一个必然结果，我们将会在第4章中遇到该定律。

当使用JUnit时，我们可以方便地使用继承性。当一个JUnit测试案例扩展另一个JUnit测试案例（而不是直接扩展junit.framework.TestSuite）时，除了被添加在该子类中的测试被执行之外，该超类中的所有测试也都被执行。这意味着JUnit测试案例能使用一个与那些被测类的具体继承性分级结构相平行的继承性分级结构。

在继承性以另一种方式用于测试案例之间时，当相对于一个接口编写一个测试案例时，我们可以使该测试案例变得抽象，并在具体的子类中测试个别实现。这个抽象超类可以声明一个抽象的保护方法，并让它返回要被测试的那个对象，进而迫使子类去实现它。

子类化一个更通用的JUnit测试案例以便为某个对象的一个子类或某个接口的一个特定实现添加新的测试是切实可行的。

现在让我们来看一个例子，本例来自本书示例应用中所使用的代码。这个代码将在第11章中详细讨论。现在，不要管它做什么；在这里，我们只对如何测试属于一个继承性分级结构的类和接口感兴趣。这个支持代码中的核心接口之一是BeanFactory接口，它提供了返回它所管理的对象的方法：

```
Object getAsSingleton(String name) throws BeansException;
```

一个常用的子接口是ListableBeanFactory，它添加附加的方法来查询所有被管理对象的名称，比如：

```
String[] getBeanDefinitionNames();
```

有几个类实现ListableBeanFactory接口，比如XmlBeanFactory（它从一个XML文档中获取Bean定义），全部的实现类除了传递应用于BeanFactory根接口的所有测试之外，还传递针对ListableBeanFactory接口的所有测试。图3.3举例说明了这些应用接口和类之间的继承性分级结构。

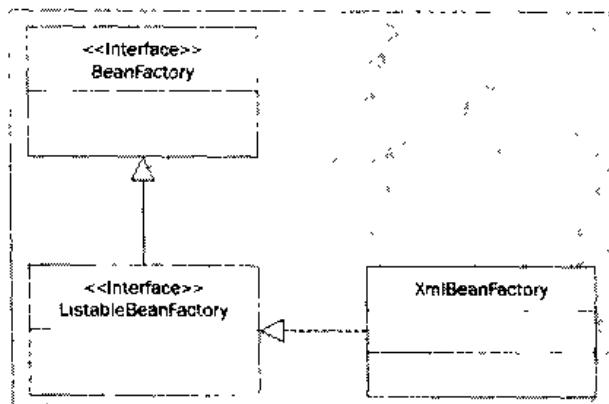


图3.3

在那些相关的测试案例中镜像这个继承性分级结构是很自然的。该JUnit测试案例分级结构的根将是一个抽象的BeanFactoryTests类。这个类将包含针对BeanFactory接口的测试，并定义子类必须实现用来返回实际BeanFactory的一个抽象保护方法getBeanFactory()。BeanFactoryTests类中的各测试方法将调用这个方法来获得运行测试所对照的附属工具对象。一个子类ListableBeanFactoryTests将包含针对ListableBeanFactory接口中所增加的功能度的附加测试，并保证getBeanFactory()方法所返回的BeanFactory具有ListableBeanFactory子接口。

由于这两个测试类都含有针对接口的测试，所以它们都将是抽象的。由于JUnit基于具体的继承性，所以一个测试案例将完全是具体的。测试接口几乎没有价值。

这两个抽象类的任何一个可以由具体测试类（比如XmlBeanFactoryTests）扩展。具体测试类将实例化和配置要被测试的BeanFactory或ListableBeanFactory实现，并（随意地）添加这个类所特有的新测试（通常没有必要添加新的类特有测试；这么做的目的只是创建运行超类测试可以对照的一个附属工具对象）。所有超类中所定义的所有测试案例都将被继承，并由JUnit自动运行。图3.4举例说明了测试案例分级结构。

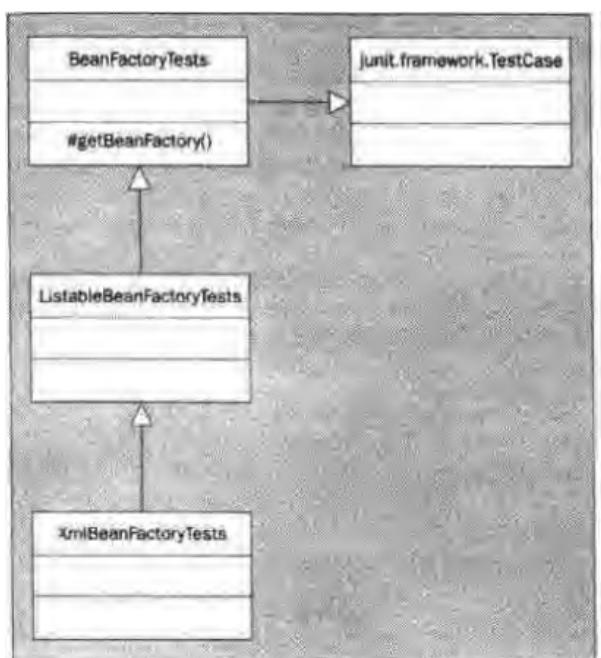


图3.4

摘自BeanFactoryTests抽象基测试类中的下列摘录说明了它怎样扩展junit.framework.TestCase并实现所需的构造器：

```

public abstract class BeanFactoryTests
    extends junit.framework.TestCase {

    public BeanFactoryTests(String name) {
        super(name);
    }
}

```

下面是由具体子类必须实现的抽象保护方法的定义：

```
protected abstract BeanFactory getBeanFactory();
```

来自**BeanFactoryTests**类的下列测试方法举例说明了这个方法的用法：

```
public void testNotThere() throws Exception {
    try {
        Object o = getBeanFactory().getBean("Mr Squiggle");
        fail("Can't find missing bean");
    } catch (NoSuchBeanDefinitionException ex) {
        // Correct behavior
        // Test should fail on any other exception
    }
}
```

ListableBeanFactoryTests类仅仅增加了更多的测试方法。它并没有实现抽象保护方法。

来自**XmlBeanFactoryTests**类（测试**ListableBeanFactory**接口的某个实现的一个具体测试集）的下列代码说明了抽象的**getBeanFactory()**方法是如何基于**setUp()**方法中所初始化的一个示例变量来实现的：

```
public class XmlBeanFactoryTests
    extends ListableBeanFactoryTests {

    private XmlBeanFactory factory;

    public XmlBeanFactoryTests (String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        InputStream is = getClass().getResourceAsStream("test.xml");
        this.factory = new XmlBeanFactory(is);
    }

    protected BeanFactory getBeanFactory() {
        return factory;
    }

    // XmlBeanFactory specific tests...
}
```

当这个测试类由JUnit执行时，它里面所定义的那些测试方法以及它的两个超类都将被执行，进而保证**XmlBeanFactory**类除了正确地实现只应用于它的任何特殊需求之外，还正确地实现了**BeanFactory**和**ListableBeanFactory**接口的契约。

测试案例应该放在何处

把测试放在一个独立于待测代码的源树中。我们不必为类的用户生成测试案例的Javadoc，而且在没有测试类的情况下，把应用类封装成JAR应该是很容易的。如果测试和应用代码处于同一个源树中，这两项任务都是较困难的。

不过，保证测试随每个应用生成一起被编译是很重要的。如果测试没有编译，它们就没有与代码同步，因此也没有价值。使用Ant，我们可以在单个操作中生成代码，而与代码位于什么地方无关。

在为待测类使用一个平行封装结构时，笔者遵循一个一般惯例。这一惯例的意思是说，

用于com.mycompany.beans包的测试也将在com.mycompany.beans包中，尽管它们在各自的源树中。这使得对保护方法和包保护方法的访问成为了可能（有时有用），但更重要的是，使查找任何一个类的那些测试案例变得很容易。

测试策略应该影响编写代码的方式吗

测试在开发过程中是一个如此重要的部分，以致我们所使用的测试策略（有保留地）影响了我们编写应用代码的方式也是正当的。

首先来看一看这些保留：笔者不喜欢白箱测试（White-box testing），而且不主张增大方法和变量的可见度来方便测试。我们前面所讨论的“平行”源树结构使测试案例能够访问保护和包保护的方法与变量，但这通常不是必需的。正如我们已经见过的，全面测试的存在促进了再加工——能够运行现有测试提供了再加工没有破坏任何东西的再保证。白箱测试降低了这个重要好处的价值。如果测试案例依赖于一个类的实现细节，那么再加工这个类有可能会同时破坏类和测试案例——事情的一种危险状态。如果维护测试变成一项太繁琐的工作，那么这些测试将得不到维护，而且我们的测试策略将会失灵。

那么，一个严格的单元测试策略对编程风格可能会有什么样的含义呢？

- 它促使我们保证类没有太大的责任，而太大的责任会使测试变得过分复杂。笔者总是使用粒度相当精细的对象，因此这往往不会影响我的编程风格。但是，许多开发人员报告说，采用测试优先开发在这方面改变了他们的风格。
- 它提醒我们保证类实例变量只能通过方法调用来修改（否则，对实例变量的外部修改会使测试变得毫无意义；如果一个类的状态能够使用不同于通过方法调用的方式来修改，测试就无法证明很多东西）。这再次体现了一种好的设计习惯：公用实例变量妨碍封装。
- 就继承性而言，它促使我们实施更严格的封装。读-写式实例保护变量的使用允许子类破坏一个超类的状态，就像允许覆盖具体方法那样。在下一章中，我们将从OO设计的角度讨论这些问题。
- 它有时促使我们添加纯粹为方便测试而设计的方法。例如，为了方便测试，添加一个包保护方法来暴露与一个类的状态有关的信息完全是合理的。请考虑这样一个类：它允许监听者通过一个公用方法被注册，但不含有暴露已注册监听者（因为其他应用代码对这一信息毫无兴趣）的方法。

添加一个包保护方法并让该方法返回已注册监听者的一个集合（或最方便的任一类型）将不会使该类的公用接口变复杂，或使该类的状态被破坏，但对同一个包中的一个测试类来说将是非常有用的。例如，一个测试类可以轻松地注册许多监听者，然后调用这个包保护方法来确认只有这些监听者被注册；该测试类也可以出版一个事件，并检查所有已注册监听者都已被通知了这个事件。

到目前为止，拥有全面单元测试对编程风格的最大影响是流量影响：再加工保证。这要求我们把那些测试看做本应用的一个核心部分。

我们已经讨论过这将如何允许我们在必要时执行优化。实现J2EE可移植性还有明显的含义。请考虑这样一个会话EJB：我们已经为该会话EJB定义了远程和本地接口。我们的测

试策略规定，我们应该有针对公用（构件）接口的全面测试（该容器隐藏了Bean实现类）。从客户软件的角度看，这些测试就等于EJB的功能度的一个保证。

现在，假设我们的当前需求是针对一个使用Oracle数据库的系统，我们可以编写一个会话组件（session bean），并让它使用一个助手类来运行Oracle专有SQL。如果我们在未来的某个时候需要迁移到另一个数据库，则可以重新实现该会话组件的实现类，进而使该构件接口保持独立。那些测试案例将帮助保证该系统表现得和以前一样。这种方法不是“纯”J2EE，但实际上很有效，并允许我们在任何时候使用最简单、最有效的实现。

当然，我们应该设法在Bean实现类之间的尽可能多的地方（或许在一个抽象超类中）共享代码。如果这是不可能的，或者实现它所需的努力将是得不偿失的，那么测试案例提供了那些实现类应该做什么的一个工作规范，并且将会使提供不同的实现（如果必要）变得更容易。

集成性和接受度测试

接受度测试是从顾客角度出发的测试。这将不可避免地包含一些动手测试，而且在这种测试中，测试者扮演用户角色，并执行测试场景。不过，我们也可以自动化接受度测试的某些方面。

集成性测试是级别较低的测试，并测试应用的类或构件如何合作。单元测试与集成性测试之间的区别实际上是模糊不清的；我们时常可以把同一个工具（比如JUnit）用于这两种测试。集成性测试只涉及使用其他类（单元测试涉及它们）来进行工作的较高级别的类。

业务对象的测试

如果我们遵照第1章的那些设计建议，应用的业务逻辑将通过一个业务接口层被暴露出来。针对这些接口而编写的测试将成为应用集成性测试的核心。测试应用接口层（比如一个Web接口）将比较简单，因为我们只需检查该接口层是否正确地调用了那些业务接口即可，也就是说，如果被正确调用，那些业务接口的实现就工作正确。

一般说来，我们可以了解一个应用的使用案例，并为每个使用案例编写许多测试案例。通常，一个业务接口上的一个方法将对应于一个单独的使用案例。

视我们在第1章中所讨论的体系结构选择而定，业务对象可能被实现为运行在Web容器中的普通Java类（没有使用EJB，但能够访问J2EE的大多数容器服务）或EJB。下面让我们依次来看一看测试每种业务对象方面的问题。

不使用EJB测试业务对象

测试普通Java类相对容易。我们可以简单地使用JUnit。惟一明显的问题是可能涉及这种类所需要的配置，以及对数据库之类的外部资源和JNDI之类的容器服务的访问。

容器服务通常可以用测试对象来模拟；例如，我们可以使用一个通用测试JNDI实现，并让它在业务对象从一个应用服务器外面被实例化时能使这些业务对象执行JNDI查找（这方面的内容将在下文中加以讨论）。

就应业务对象来说，我们将总是编写对业务接口的测试。

有些业务对象依赖于其他应用对象——虽然这样的依赖性应该依附于接口，不应该依附于类。要解决这个问题，我们有3种主要的选择：

- 使用相关接口的测试实现（它们返回测试数据）来替换那些必需的对象。只要这些接口实现起来不复杂，这种方法非常管用。
- 实现能够在应用服务器内运行的测试，其中应用已被配置成和生产中一样。我们将在下文中讨论这种方法，因为它通常是测试EJB的惟一选择。
- 设法把应用基础结构设计成使应用配置不依赖于J2EE服务器。第11章中将要讨论的基于JavaBean的应用基础结构使这种设计变得更容易，进而使相同的应用特有配置文件能够和运行时一样被一个测试附属工具读取。基于JavaBean的应用基础结构保证，（除了涉及到EJB的地方）许多业务对象能够在EJB容器外面被测试。依赖于容器服务的业务接口实现能够被测试实现所取代，以允许应用服务器上没有部署也能进行集成性测试。

测试EJB

测试EJB比测试普通Java类要难得多，因为EJB依赖于EJB容器服务。

一般说来，我们将只关注会话组件的测试，而不太关注实体组件的测试，即使我们选择使用实体组件。实体组件通常不含有业务逻辑；它们对持久性数据的影响应该用会话组件测试来检查。

我们不能像对待普通Java对象那样简单地实例化EJB，并测试它们。EJB是被管理的对象；EJB容器管理EJB在运行时的生命周期，并且EJB依赖于诸如连接池之类的容器服务。此外，EJB容器还控制着EJB的功能度的访问，而且容器监听所添加的行为（比如事务管理和安全约束）也是待测试的应用本身和应用需求的一部分。

解决这个问题有几种方法：

- 编写一个测试，并让它成为EJB容器的一个远程客户。这通常是测试具有远程接口的EJB的最佳方法。
- 编写并部署一个在应用服务器内执行的测试。这是测试具有本地接口的EJB的一种好方法。它将需要附加的基础结构以补充JUnit。
- 用桩基对象取代容器对象来进行测试。这种方法一般只在EJB对EJB容器有简单的要求时才管用。

最明显的方法是远程客户方法。这种方法简单、直观。我们可以编写用来连接EJB服务器的普通JUnit测试案例。这些测试案例从EJB容器中的一个单独JVM中运行。我们可以像调用任何JUnit测试一样调用它们，从而只需要注意我们提供了适当的JNDI属性来允许与EJB容器的连接，并供给了必要的EJB客户二进制文件。从消极的方面看，通过一个远程客户进行测试不能让我们通过它们的本地接口来测试EJB。我们不能测试本地调用语义的结果。甚至在我们希望测试具有远程接口的EJB时，这可能都是一个问题，因为当我们在同一个服务器实例中运行EJB和Web应用时，可能会希望允许容器优化。

我们可以通过编写在应用服务器内运行的测试来克服这些问题。一般说来，我们把测试封装成Web应用，因而让它们能够访问运行在同一个JVM内的EJB（这可能会允许本地调

用，但目前没有得到J2EE规范的保证）。不过，这种方法实现起来较困难，需要用于JUnit的附加基础结构，而且使应用部署变得更复杂。

最后，我们可以模拟我们自己的EJB容器来供应EJB在运行时所期望的服务。不过，由于EJB基础结构的复杂性，这种方法通常是不切实际的。EJB不仅能够访问容器提供的接口，比如一个`javax.ejb.SessionContext`，而且还能以不同于直接通过API的方式访问容器服务（例如，访问它们的命名上下文的能力）。安全与事务管理服务复制起来也困难。

本书的下载在`/framework/test`目录中包含了能与这种方法一起使用的一些有用的通用类，比如哑EJB上下文对象，以及一个测试JNDI实现（它允许所需对象在一个模拟命名上下文中的绑定，目的是允许EJB执行JNDI查找，就好像这些EJB正运行在一个服务器内）。可是，这种方法只有当EJB对该容器有简单的需求时才管用。当使用这种方法时，我们还必须保证在创建一个测试夹具时，调用诸如`setSessionContext()`之类的EJB生命周期方法。

表3.1总结了这3种方法的优缺点。

表3.1

方法	优点	缺点
用一个远程客户做测试	编写和运行测试容易。 能够使用标准JUnit基础结构。 将保证我们的EJB支持真正的远程语义。 由分布式应用中的EJB层所暴露出的那些远程接口通常也暴露出应用的业务逻辑，所以这是很自然要测试的一个地方	我们不能测试本地接口。 应用可能在生产中使用了按引用调用，甚至使用了带远程接口的按引用调用
在应用服务器内（或在EJB容器或Web容器内）做测试	就Web引用而言，这可能意味着测试将拥有和使用EJB层的应用代码完全相同的EJB层访问权限。	需要附加的测试框架。 测试的更复杂实现、部署和调用
使用桩基对象取代容器对象做测试	我们不用EJB容器也能运行测试。 我们或许能够在多个应用中重用标准的基础结构组件	我们最终可能需要编写许多模拟容器行为的类。 我们还没有测试我们将要在应用服务器中部署的那个应用

如果我们使用远程接口来测试EJB，根本就不需要除JUnit本身之外的任何特殊工具。如果我们在EJB容器内做测试，则需要一个能使测试案例被封装到一个J2EE应用中的工具。

Cactus

笔者所知道的、用来在应用服务器内进行测试的、最成熟的免费工具是Cactus（可以在<http://jakarta.apache.org/cactus/index.html>站点中获得）。它是一个基于JUnit的开放源框架，允许EJB、小服务程序、JSP页和小服务程序筛选器在目标应用服务器中被测试。

在客户端上，Cactus提供如同普通JUnit测试一样的测试调用和结果报告；Cactus负责建

立与服务器的连接，而该服务器是测试案例在里面实践运行的地方。客户JVM中的测试运行器建立与服务器JVM中的一个Cactus“重定向器”的连接。虽然每个测试类在服务器和客户中都被实例化，但那些测试在一个运行于服务器内的Web应用中被执行。一般说来，这个Web应用将是含有该应用的Web接口的同一个Web应用。

Cactus是一个复杂的框架，而且安装起来相当复杂。但是，它是测试EJB的一种好方法，而且在这种情况下，复杂性是不可避免的。

安装Cactus包括下列这些步骤：

1. 确保Cactus类路径得到了正确设置。这个区域是使用Cactus时导致错误发生的最常见根源，所以请仔细阅读Cactus分发品中所包含的关于“安装Cactus类路径”的技术说明。大多数Cactus二进制文件必须被包括在一个WAR分发品中的WEB-INF/lib目录下面。

如果有多个应用可能会使用Cactus，笔者建议把那些Cactus二进制文件包含在服务器级上，以便它们能让所有应用使用。在JBoss中，这只是意味着把那些JAR文件复制到你将把应用部署到的那个JBoss服务器的/lib目录中。当使用这种方法时，没有必要把Cactus JAR文件包括在每个WAR的WEB-INF/lib目录中。当使用Cactus来测试EJB时，要确保无任何一个Cactus小服务程序测试案例被包括在EJB JAR中。这将会导致类装入问题，比如产生神秘的“类未被找到”错误。

2. 编辑该Web应用的web.xml部署描述符来定义Cactus“小服务程序重定向器”小服务程序，该小服务程序将把来自远程测试的问题发送给服务器端测试实例。这个定义看起来应该像下面这样：

```
<servlet>
    <servlet-name>ServletRedirector</servlet-name>
    <servlet-class>
        org.apache.cactus.server.ServletTestRedirector
    </servlet-class>
</servlet>
```

我们还需要提供一个映射到这个小服务程序的URL（请注意，对于像Jetty那样的一些Web容器，需要把Cactus技术文档内的那个示例中所包含的尾部字符“/”删去，就像我们在本例中所做的那样）。

```
<servlet-mapping>
    <servlet-name>ServletRedirector</servlet-name>
    <url-pattern>/ServletRedirector</url-pattern>
</servlet-mapping>
```

3. 把测试类包括在WAR中。Cactus测试类必须从一个处理重定向的Cactus超类中派生——我们将在下文中讨论。
4. 配置Cactus客户。我们将需要保证所有Cactus二进制文件（服务器端上所需的那些文件和附加的客户端库）对该客户都是可使用的。我们还必须提供一个cactus.properties

文件，以便把服务器URL和端口告诉给Cactus，并指定该Web应用的上下文路径。为了测试笔者本地计算机上的示例应用，`cactus.properties`文件如下所示。请注意，`servletRedirectorName`属性应该匹配于我们在`web.xml`中所创建的URL映射。

```
cactus.contextURL = http://localhost:8080/ticket
cactus.servletRedirectorName = ServletRedirector
cactus.jspRedirectorName = JspRedirector
```

一旦我们完成了这些步骤，就可以像调用普通JUnit测试案例一样在客户端上立即调用测试案例。

现在让我们来看一看编写一个Cactus测试案例都涉及些什么。原则和针对任何JUnit测试案例的原则一样，但我们必须扩展Cactus的`org.apache.cactus.ServletTestCase`类，而不是直接扩展`junit.framework.TestCase`类。`org.apache.cactus.ServletTestCase`超类提供了在客户中调用测试和执行报告的能力，而测试实际上是在服务器中运行的。

下面来看一个实际的例子。我们首先扩展`org.apache.cactus.ServletTestCase`类。

```
Public class CactusTest extends ServletTestCase {
```

这个类的其余部分使用标准的JUnit概念。我们安装一个符合正常JUnit约定的测试附属工具，并像往常那样实现测试方法：

```
private BoxOffice boxOffice;
public CactusTest(String arg0) {
    super(arg0);
}
public static Test suite() {
    return new TestSuite(CactusTest.class);
}
```

当我们在如下所示的`setUp()`方法中，或者在各测试方法中创建一个测试附属工具时，可以访问服务器的JNDI上下文来查找EJB：

```
public void setUp() throws Exception {
    Context ctx = new InitialContext();
    BoxOfficeHome home =
        (BoxOfficeHome) ctx.lookup("java:comp/env/ejb/BoxOffice");
    this.boxOffice = home.create();
}

public void testCounts() throws Exception {
    int all = boxOffice.getSeatCount(1);
    int free = boxOffice.getFreeSeatCount(1);
    assertTrue("all > 0", all > 0);
    assertTrue("all > free", all >= free);
}
```

`org.apache.cactus.ServletTestCase`类使许多“隐式对象”可成为子类测试方法的实例变量，其中包括我们可以从中获得该Web应用的全局`ServletContext`的`ServletConfig`对象，这在我们需要访问Web应用属性时是非常有用的。

Cactus还提供了在客户端上提供测试输入的能力：通过与每个测试案例所关联的附加方法（这最适用于测试小服务程序，而不适用于EJB）。关于这类高级特性的信息，请参考详细的Cactus技术资料。

这是保证我们享有JUnit所提供的各种好处（比如自动执行多个测试集的能力），同时又在服务器中实际运行测试的一种强有力机制。可是，它使应用部署变得更复杂。一般说来，我们将需要不同的生成目标来生成包含测试案例的应用，用于测试的Cactus配置与Cactus二进制文件，以及不含有生产测试支持的应用。

JUnitEE (<http://junitee.sourceforge.net/>) 是一个比Cactus更简单的框架，但也基于在J2EE服务器内运行测试。和Cactus一样，JUnitEE把测试封装在WAR中。JUnitEE提供了一个小服务程序，这个小服务程序在Web容器内运行普通JUnit测试案例。JUnitEE不是给客户和服务器上所保存的测试案例使用一种重定向机制，而是提供一个允许在服务器上选择测试案例和生成输出结果的小服务程序。

使用JUnitEE来实现测试是非常容易的，因为测试案例就是JUnit测试案例。JUnitEE基础结构所做的只是提供一种运行测试案例的有J2EE感知力的手段。测试案例将被简单地实现成带有这样的认识：它们将运行在运行该应用的J2EE服务器内。最重要的含义是它们能够访问JNDI，这也是它们用来查找待测试的应用EJB的手段。

这种更简单的方法有一个缺点：测试只能从一个浏览器中被调用，进而意味着自动化测试过程是不可能的。

过去，Cactus不支持这种更简单、更直观的方法，但Cactus 1.4提供了一个“Servlet测试运行器”，并且该运行器能够使Cactus使用和JUnitEE相同的方法。笔者建议使用Cactus，不要使用JUnitEE，因为能够把测试过程自动化为生成过程的一部分是非常重要的。

要使用Cactus 1.4小服务程序测试运行器，需要完成下面这些步骤：

1. 确保所有Cactus二进制文件（不仅仅服务器端二进制文件）都被分布在应用WAR中（或被放置到服务器类路径上，以便所有应用都能够使用它们）。
2. 编辑web.xml，以便为Cactus ServletTestRunner小服务程序创建一个小服务程序定义和URL映射。该小服务程序定义如下所示：

```
<servlet>
  <servlet-name>ServletTestRunner</servlet-name>
  <servlet-class>
    org.apache.cactus.server.runner.ServletTestRunner
  </servlet-class>
</servlet>
```

URL映射看起来应该像下面这样：

```
<servlet-mapping>
  <servlet-name>ServletTestRunner</servlet-name>
  <url-pattern>/ServletTestRunner</url-pattern>
</servlet-mapping>
```

3. 确保测试案例被包含在WAR中。需要注意的是，当使用这种方法时，我们不必非要

扩展org.apache.cactus.ServletTestCase类；只要愿意，我们可以使用普通JUnit测试案例（虽然这些测试案例不支持Cactus重定向，如果我们需要自动化测试过程）。

使用这种方法，我们不必担心客户端配置，因为我们可以通过一个浏览器来运行测试。我们所需要做的只是请求一个像下面这样的URL：

<http://localhost:8080/mywebapp/ServletTestRunner?suite=com.mycompany.MyTest&xsl=junit-noframes.xsl>

小服务程序测试运行器默认地将结果返回为一个XML文档；上面例子中的xsl参数指定一个能够用来把XML文档转换成HTML文档并在一个浏览器中显示它们的格式表（该格式表与Cactus分发品被包含一起，但必须被包括在使用该小服务程序测试运行器的每个应用WAR中）。

测试结果将被显示成和图3.5中的下列例子一样（本例摘自Cactus技术资料）。

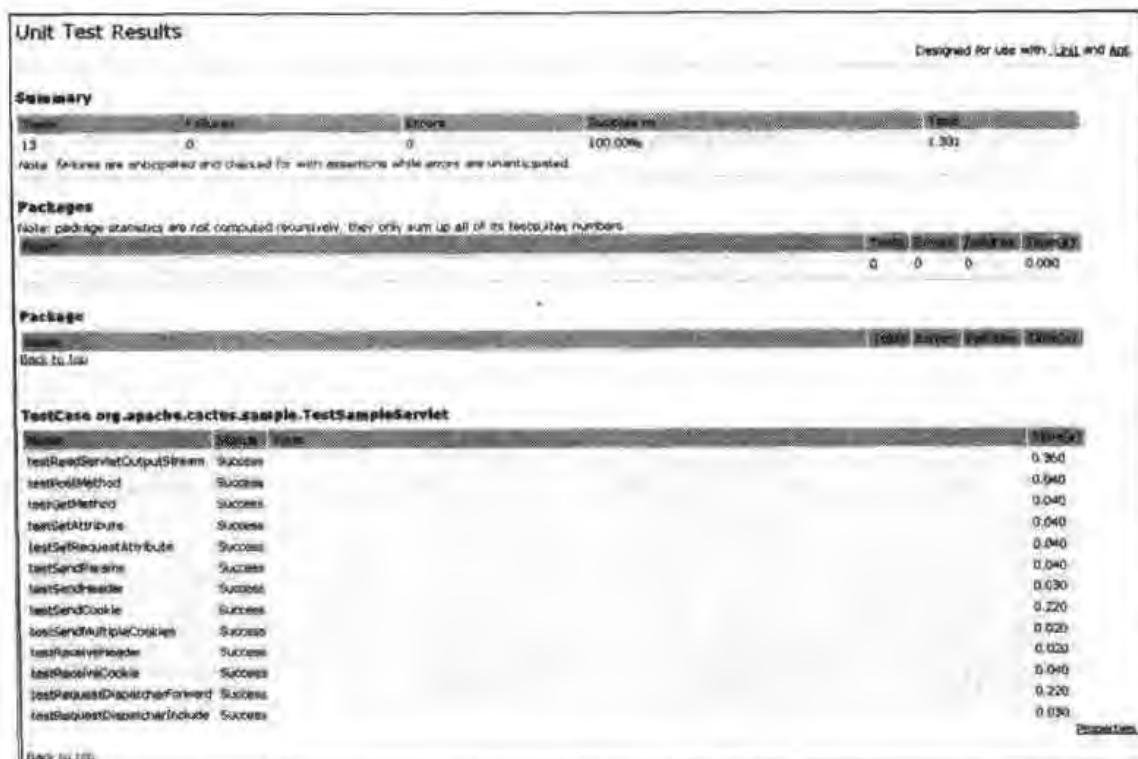


图3.5

这种方法具有在服务器内做测试的JUnitEE方法的各种优缺点。它配置起来相当简单，但不是可脚本化的，因此不能轻松地被集成到应用生成过程中。

当我们使用带远程接口的EJB时，可以编写从一个远程JVM中测试这些EJB的普通JUnit测试案例。

当我们使用带本地接口的EJB时，通常将需要在目标应用服务器内测试这些EJB。

在应用服务器内做测试的缺点是，应用部署变得更复杂，并且配置和执行花费比测试普通Java类更长的时间。

测试数据库交互性

业务对象（无论它们是EJB还是普通Java对象）将肯定会与一个数据库进行交互（尽管这种交互没有必要是直接的），而且将依赖J2EE数据源。因此，我们将必须考虑我们的测试对该数据库中的数据和它们所需要的数据有怎样的影响。这里有几种对策。

最基本的方法是在测试时摆脱该数据库，并使用模拟对象来代替实际的JDBC类（关于这种方法的详细信息，请参见http://www.mockobjects.com/papers/jdbc_testfirst.html站点）。这种方法避免了对持久性数据修改的任何需要或与持久性数据修改相关的问题。不过，这种方法将不帮助我们测试复杂的查询或更新（而我们确实想让目标数据库运行行为应用代码的一部分），并且集成EJB容器也很困难。

因此，我们通常将需要访问一个测试数据库。这意味着我们一般将需要编写SQL脚本，并使它们在每个测试运行之前执行，以便把该数据库设置成所需要的状态。这些SQL脚本是那些测试的一个不可分部分。使用Ant，在我们运行测试之前使数据库脚本的执行自动化是可能的：Ant允许我们执行一个生成文件或一个单独脚本中所包含的SQL语句。

当测试JDBC助手类时，编写一个回退任何修改的测试案例或许是可能的，进而意味着事后清理该数据库是没有必要的。不过，当我们测试运行在一个EJB容器中的代码时，这是不可能的，因为该EJB容器应该创建并提交或回退本事务。

对持久性数据的修改是大部分EJB代码的功能度的核心。因此，测试案例必须具有建立与数据库的连接并在EJB代码执行之前和之后检查数据的能力。我们还必须检查回退只有在它被业务逻辑要求或一个错误被遇到时才发生。

为了举例说明这一点，请考虑在一個EJB的远程接口上测试下列方法：

```
InvoiceVO placeOrder(int customerId, InvoiceItem[] items)
throws NoSuchCustomerException, RemoteException, SpendingLimitViolation;
```

在这里，我们需要多个测试案例：一个用于有效定单；一个用于不存在的顾客所放置的定单，以便确认正确的异常被抛出；以及一个用于量值过大的定单，以便检查SpendingLimitViolation被抛出。我们的测试案例应该包括代码，以便为随机顾客和随机产品生成定单。

这一级别的测试要求测试案例应该能访问基础数据。为了实现这一点，我们使用一个助手对象，并让该对象能连接到和EJB服务器相同的数据库，以便运行SQL函数和查询来验证该EJB的行为。我们也可以使用一个助手对象来装入来自数据库中的数据，以便提供我们可以用来生成随机定单的一组顾客号和产品号。我们将在第9章中讨论一些适用的JDBC助手类。

请看一看下面这个测试方法，它检查出一个过大的定单导致一个SpendingLimitViolation异常正被抛出。保证该事务在这种情况下被回退以及不存在对该数据库的任何持久性修改也是EJB的责任。我们也应该检查这一点。这个测试方法需要数据库中存在两个Products（发票项目）和一个其主键为1的Customer。一个测试脚本应该保证在该测试案例运行以前这些数据是存在的。

```

public void testPlaceUnauthorizedOrder() throws Exception {
    int invoicesPre = helper.runSQLFunction("SELECT COUNT(ID) FROM INVOICE");
    int itemsPre = helper.runSQLFunction("SELECT COUNT(*) FROM ITEM");
    InvoiceItem[] items = new InvoiceItem[2];

    // Constructor takes item id and quantity
    // We specify a ridiculously large quantity to ensure failure
    items[0] = new InvoiceItemImpl(1, 10000);
    items[1] = new InvoiceItemImpl(2, 13000);

    try {
        InvoiceVO inv = sales.placeOrder(1, items);
        int id = inv.getId();
        fail ("Shouldn't have created new invoice for excessive amount");
    } catch (SpendingLimitViolation ex) {
        System.out.println("CORRECT: spending limit violation " + ex);
    }

    int invoicesPost = helper.runSQLFunction("SELECT COUNT(ID) FROM INVOICE");
    int itemsPost = helper.runSQLFunction("SELECT COUNT(*) FROM ITEM");
    assertTrue("Must have same number of invoices after rollback",
               invoicesPost == invoicesPre);
    assertTrue("Must have same number of items after rollback",
               itemsPost == itemsPre);
}

```

因此，我们需要在基础结构中花一些时间，以支持测试案例。

Web接口的测试

测试Web接口比测试普通Java类困难，甚至比测试EJB困难。Web应用不提供简洁、可容易验证的响应：我们需要测试的动态内容就像一个花哨的置标海洋中的岛屿。Web应用的外观和感觉经常变化，因此我们需要能够设计在这种变化每次发生时都不必被重写的测试。

有许多与Web具体相关的问题，其中有些问题在自动化的测试中再现起来是很困难的。例如：

- 一个窗体的重新提交（例如，如果用户重新提交一个购置窗体，而服务器仍正在处理第一个提交，怎么办？）
- 使用后退按钮的含义
- 安全问题，比如对服务攻击拒绝的抵抗
- 用户打开多个窗口时的问题（例如，我们可能需要同步对有状态会话组件的Web层访问）
- 取消的请求的含义
- 浏览器（可能还有ISP）高速缓存的含义
- GET和POST请求是否都应该得到支持

和EJB一样，Web层构件依赖于容器服务，进而使单元测试变得很困难。

JSP页的单元测试是特别困难的。JSP页在被部署到一个Web容器中之前，并不作为Java类而存在。它们依赖于Servlet API，并且不提供一个可容易测试的接口。这也是JSP决不应该实现业务逻辑的一个原因，因为业务逻辑必须始终被测试。JSP页通常被作为该应用的完整Web接口的一部分来测试。

其他一些视图技术（view technology），比如Velocity模板和XSLT格式表的单元测试比较容易，因为它们不依赖于Servlet API。但是，一般说来，没有必要孤立地测试视图，因此这不是一项重要的考虑。

通常，我们将只关注测试Web接口的两种方法：Web层Java类的单元测试和整个Web应用的接受度测试。下面让我们来依次讨论每种方法。

Web层构件的单元测试

通过提供仿真服务器的桩基对象，我们可以使用标准JUnit功能度从小服务程序容器外面测试Web层Java类。ServletUnit项目（<http://sourceforge.net/projects/servletunit/>）提供了能够用来从一个容器外面调用小服务程序和其他Servlet API特有类的对象，比如ServletContext、HttpServletRequest和HttpServletResponse实现。这使我们能直接调用任何请求处理方法，并对响应做出断言（比如，该响应含有正确属性）。这种方法对简单的Web层类非常有效。但是，如果对象需要较复杂的初始化（例如，加载一个WAR的WEB-INF目录内所含有的数据），这种方法就不太有用。

虽然ServletUnit包是一个出色的主意，但它是一个过于简单的实现，没有实现我们将需要使用的某些Servlet API方法（比如那些状态码方法）。在本书的配书下载中，/framework/test/servletapi目录含有更合用的测试对象，这些测试对象最初基于ServletUnit实现，但提供了更完善的功能度。

使用这种方法是非常简单的。测试对象不仅实现相应的Servlet API接口，而且也提供能让我们把数据供给那些被测类的方法。最常见的需求是添加请求参数。下面这个例子使用一个单独的“name”参数为URL“test.html”创建一个GET请求：

```
TestHttpRequest request = new TestHttpRequest(null, "GET", "test.html");
request.addParameter("name", name);
```

由于使用一个MVC框架实现Web应用是一个好习惯，所以我们通常不需要直接测试小服务程序（MVC框架通常提供一个单独的通用控制器小服务程序，并且这个小服务程序不是我们的应用的一部分）。我们一般将使用Servlet API测试对象来个别地检查各请求控制器（我们可以假设该控制器框架已经被测试过，并且我们的请求控制器将在运行时得到正确调用）。

例如，我们的示例应用（参见第12章）中所使用的MVC Web应用框架就需要请求控制器来实现下列接口：

```
ModelAndView handleRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException;
```

请求控制器实际上不生成响应内容（这是视图的任务），但选择一个应该把响应结果再现出来的视图，并提供该视图应该显示的模型数据。控制器和视图的这种分离不仅是一个好的设计习惯，而且也极大地简化了单元测试。我们可以忽略置标生成，并简单地检查控制器选择了正确的视图和暴露了必要的模型数据。

现在让我们来看一个简单的控制器实现，顺便看一个能够用来测试该实现的JUnit测试

类。该控制器根据一个name请求参数的存在性和有效性，返回3个不同名称中的一个名称。如果提供了一个参数，它构造被传递给一个视图的模型。

```
public class DemoController implements Controller {
    public static final String ENTER_NAME_VIEW = "enterNameView";
    public static final String INVALID_NAME_VIEW = "invalidNameView";
    public static final String VALID_NAME_VIEW = "validNameView";

    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException {
        String name = request.getParameter("name");
        if (name == null || "".equals(name)) {
            return new ModelAndView(ENTER_NAME_VIEW);
        } else if (name.indexOf("-") != -1) {
            return new ModelAndView(INVALID_NAME_VIEW, "name", name);
        } else {
            return new ModelAndView(VALID_NAME_VIEW, "name", name);
        }
    }
}
```

下面的JUnit测试案例将检查3种情况——名称参数未被提供、名称参数有效以及名称参数无效。

```
package com.interface21.web.servlet.mvc;

import javax.servlet.http.HttpServletResponse;
import com.interface21.web.servlet.ModelAndView;
import junit.framework.TestCase;
import servletapi.TestHttpRequest;
import servletapi.TestHttpResponse;

public class DemoControllerTestSuite extends TestCase {
    private Controller testController;
    public DemoControllerTestSuite(String arg0) {
        super(arg0);
    }
}
```

在setUp()方法中，我们将初始化该控制器。对于真正的控制器，我们将需要通过设置Bean属性（比如对应用业务对象的引用）来配置控制器，但在我们所介绍的这个简单示例中，这种配置不是必需的。任何必需的对象通常都能由返回哑数据的测试实现来取代。我们不是在测试业务逻辑的实现，而是在测试它是否得到了Web层构件的正确调用。

```
public void setUp() {
    testController = new DemoController();
}
```

每个测试方法都将创建一个测试请求与响应对象，并核实该控制器选择了正确的视图名称，以及返回了所需要的任何模型数据。

```

public void testNoName() throws Exception {
    TestHttpRequest request =
        new TestHttpRequest(null, "GET", "test.html");
    HttpServletResponse response = new TestHttpResponse();
    ModelAndView mv =
        this.testController.handleRequest(request, response);
    assertTrue("View is correct",
               mv.getViewname().equals(DemoController.ENTER_NAME_VIEW));
    assertTrue("no name parameter", request.getParameter("name") == null);
}

public void testValidName() throws Exception {
    String name = "Tony";
    TestHttpRequest request = new TestHttpRequest(null, "GET", "test.html");
    request.addParameter("name", name);
    HttpServletResponse response = new TestHttpResponse();
    ModelAndView mv = this.testController.handleRequest(request, response);
    assertTrue("View is correct",
               mv.getViewname().equals(DemoController.VALID_NAME_VIEW));
    assertTrue("name parameter matches",
               request.getParameter("name").equals(name));
}

public void testInvalidName() throws Exception {
    String name = "Tony--";
    TestHttpRequest request = new TestHttpRequest(null, "GET", "test.html");
    request.addParameter("name", name);
    HttpServletResponse response = new TestHttpResponse();
    ModelAndView mv = this.testController.handleRequest(request, response);
    assertTrue("View is correct: expected '" +
               DemoController.INVALID_NAME_VIEW + "' not '" + mv.getViewname() + "'",
               mv.getViewname().equals(DemoController.INVALID_NAME_VIEW));
    assertTrue("name parameter matches",
               request.getParameter("name").equals(name));
}
}

```

有用于一些常见MVC Web框架的测试包，比如Struts，而且这些测试包允许我们使用它们编写用于应用的测试案例。对于使用第12章中将要讨论的框架来测试请求控制器，不需要任何特殊的支撑，因为这些请求控制器在运行时不依赖于控制器小服务程序。对于像Struts那样的框架，需要特定的支持，因为在这些框架中，应用请求处理程序依赖于框架的控制器小服务程序。

如果遵照第1章中所概述的那些设计原则，我们就不必编写许多这样的测试类。Web接口将是一个如此之薄的层，以致我们可以依靠Web层接受度测试来找出它如何调用业务逻辑方面的任何问题。

我们也可以使用一个像Cactus那样的工具，在Web容器内对Web层构件进行单元测试。在这种情况下，我们不必提供Servlet API测试对象。缺点是测试部署和创作比较复杂。在容器外面执行测试更容易、更快速，因此笔者建议读者采用这种方法。

Web接口的接受度测试

这两种方法等同于单元测试，而且单元测试在Web层中只具有有限的重要性。由于一个Web接口反映了系统在用户眼中的样子，所以我们需要能够实现接受度测试。HttpUnit项目

(<http://httpunit.sourceforge.net/>) 能使我们编写在服务器外面运行的测试案例 (**HttpUnit**二进制文件也和Cactus被包含在一起)。

HttpUnit是一个类集，其中的类供JUnit测试案例中使用，进而使我们能够自动化HTTP请求和做出对响应结果的断言。**HttpUnit**不跟小服务程序打交道；它不关心服务器端实现是什么，所以它同样适用于JSP页和XML生成的内容。**HttpUnit**允许对已生成文档的容易访问，例如，它的**WebResponse**对象暴露可在该页上找到的窗体、链接和cookie的一个数组。**HttpUnit**在屏幕碎片周围提供了一个优雅的封装。

当我们由于其他某种原因需要做HTML屏幕碎片时，**HttpUnit**也是一个方便的库。

这种**HttpUnit**方法本质上是白箱式接受度测试。它具有我们可以精确地测试应用的优点，因为它将被部署在生产中，进而运行在生产服务器上。编写测试案例也是轻松而直观的。缺点是屏幕碎片会容易被应用外观与感觉方面的变化所破坏，而这些变化又没有反映功能度方面的变化。

作为**HttpUnit**测试类的一个例子，请看一看下列程序清单。需要注意的是，它来自一个普通JUnit测试案例；**HttpUnit**是一个库，不是一个框架。突出显示的代码核实检查该页面不含有任何窗体，以及它含有非零个链接。我们可以在我们的测试案例中建立与数据库的连接，并验证那些链接反映了该数据库中的数据，如果该页面是数据驱动的。

```
public void testGenresPage() throws Exception {  
    WebConversation conversation = new WebConversation();  
    WebRequest request = new  
        GetMethodWebRequest("http://localhost/ticket/genres.html");  
  
    WebResponse response = conversation.getResponse(request);  
    WebForm forms[] = response.getForms();  
    assertEquals(0, forms.length);  
  
    WebLink[] links = response.getLinks();  
    int genreCount = 0;  
    for (int i = 0; i < links.length; i++) {  
        if (links[i].getURLString().indexOf("genre.html") > 0){  
            genreCount++;  
        }  
    }  
    assertTrue(" There are multiple genres", genreCount > 0);  
}
```

根据笔者的经验，就Web接口而言，接受度测试比单元测试重要，因为一个Web接口应该是位于本应用的业务接口上面的一个薄层。本应用的使用案例的实现应该对照那些业务接口的实现被测试。

HttpUnit类库提供了使用JUnit来实现Web层接受度测试的一种出色而又直观的方法。

设计的本质

概括地说，我们可以测试EJB和Web层构件，但比测试普通Java类要困难得多。我们需要掌握刚刚讨论过的用于J2EE构件的那些测试技巧，但也需要学习这样一门课：如果我们把

自己的应用设计成能使我们在不需要测试J2EE构件的同时尽可能深入地测试它们，那么我们的应用将更容易测试。我们在第11章中将要讨论的应用基础结构就被设计成了使上述这种设计变得更容易。

在EJB层中，通过使EJB成为代表普通Java类中所实现的业务逻辑的一个外观，我们或许能够实现这种设计。不过，仍将有诸如容器管理事务定界之类的问题。但是，如果没有EJB容器，测试关键功能度通常是不可能的（要想了解按这种方式来测试EJB的一种面向XP的方法，以及对测试EJB的各种困难的一般性讨论，请参见<http://www.xp2001.org/xp2001/conference/papers/Chapter24-Peeters.pdf>）。

这种方法不能被运用于实体组件，因为它们的功能度与EJB容器服务连接在一起，而且这种连接是解不开的。这也是笔者不喜欢实体组件的原因之一。

在Web层中，最大限度地减少测试小服务程序的需求是相当容易的。这可以通过保证Web特有的控制器（小服务程序和助手类）访问一个定义明确且不是Web特有的业务接口来实现。这个接口可以使用普通的JUnit测试案例来测试。

在可能的地方，应该把应用设计成它们的核心功能度能够对照普通Java类来测试，而不是对照像EJB、小服务程序或JSP页面之类的J2EE构件来测试。

性能与可缩放性的测试

一个应用（尤其是一个Web应用）可能会通过单元和集成性测试，但在生产中却执行不正确或失败得很惨。我们迄今为止已经考虑的大部分测试策略都是单线程的。使用JUnit执行多线程测试是可能的，但这不是最初设计它的目的。

我们通常需要知道一个应用如何执行，尤其是需要知道它如何处理递增的负载。负载和压力测试是J2EE开发生命周期的一个重要部分。负载测试和单元测试一样是重要的应用人工制品。

我们将在第15章中了解负载测试的一些实际示例。在接下来的这一节中，我们将了解一些主要问题。

负载（和压力）测试不应该留到一个应用实质上已完成之后再进行。我们已经考虑过及早处理风险的重要性。不充足的性能与可缩放性是企业应用，尤其是Web应用中的一个主要风险。

在项目生命周期的早期实现应用功能度的一个概念模型或“纵向程序片”并对它运行负载测试是一个好的习惯。因此，负载测试应该在整个项目生命周期内连续进行。在整个项目生命周期内，应用性能调整与负载测试密切相关。

负载测试EJB和其他业务对象

普通Java类的负载测试在J2EE中不总是占有一个突出的地位，因为J2EE基础结构的存在部分是为了解决像并发性那样的在负载之下可能会有麻烦的问题。对EJB和Web接口进行负载测试通常是最重要的。

但是，直接对高速缓存器之类的个别Java类进行负载测试可能是必要的。使用了同步的

任何应用代码都是孤立地进行负载测试的一个候选者，以保证它不会引起死锁或严重地限制总吞吐量。

业务对象（包括EJB）的直接负载测试是对Web接口的负载测试的一个宝贵补充。把业务对象的负载测试结果与整个应用（包括接口）的负载测试结果进行比较，将可以看出该应用的哪些层正占用更长的时间和生成大部分的系统负载——任何必要的优化的一个关键输入值。由于使用EJB可能会增加显著的系统开销——尤其是在分布式系统中，所以对EJB进行负载测试也将可以看出EJB是否正得到正确使用。

由于EJB和其他业务对象的工作经常涉及使用数据库之类的非Java资源，所以把那些资源所耗用的时间与业务对象所耗用的时间分开是很重要的。例如，如果出现了数据库访问时间是一个问题的情况，那么对EJB层的其他修改可能不会帮助改进性能。相反，我们可能需要对我们的查询和更新做负载测试，以便我们能微调它们，微调数据库，或者重新考虑我们的数据存取策略。考虑数据库上并发操作的含义（例如，是一个可能的死锁，给定该应用所发布的查询）可能也是很重要的。这样的问题可以通过在Oracle SQL*Plus之类的数据数据库工具中使用并发会话来特别地进行探测，但最好是通过编写一个自动化测试来正规化。

和功能性测试一样，就EJB而言，我们必须决定测试应该在何处运行。同样，在一个不同的虚拟机（也可能是一台不同的物理机）中运行压力测试并使用对EJB层的远程访问是最简单的形式。这防止了把负载测试客户所生成的系统负载与EJB容器本身的系统负载混淆起来，但引入了网络和串行化系统开销（我们也可以运行多个客户）。不需要任何特殊的基础结构——我们可以使用任一负载测试框架并编写连接到该EJB服务器的代码。如果应用使用了本地接口，或者将不被部署成带有远程调用，我们将需要在相同的企业应用中运行那些负载测试。不过，这将需要额外的基础结构。

任何一个基于Java的负载测试工具至少应该允许我们做下列事情：

- 在多个线程中执行用户定义操作（即普通Java对象或EJB的调用）；
- 在操作之间使用随机睡眠时间；
- 从配置文件中读取数据，以允许容易的参数化；
- 按需要重复操作；
- 制作易读报告。

在第15章中，我们将了解几个满足这些需求的工具，并见识这方面的一个实际示例。

Web接口的负载测试

Web接口为负载和压力测试提供了一个理想（而又非常容易）的选择。如果对一个Web接口（或应用生命周期的早期所建立的纵向程序片）进行负载测试的结果是令人满意的，我们可能就没有必要把资源用在对EJB和其他应用内部对象进行负载测试上。

目前有许多可用于Web应用的免费负载测试工具。笔者喜爱的工具是Microsoft的Web Application Stress Tool（可以从<http://webtool.rte.microsoft.com/>站点中免费获取），该工具的使用特别容易。

无论读者使用哪个Web应用压力工具，你都应该计划模拟一个真实的用户总数。编写一个模拟真实用户活动的脚本。视具体应用而定，这个脚本可能包括用户会话的创建和维护。和所有负载测试一样，压力测试必须能够模拟负载方面的尖峰值。

测试的自动化

一个应用测试集的价值依赖于它易于运行的程度。将有许多单独的测试类，而且在单个操作中运行所有测试类必须是有可能的。虽然每个测试运行都应该报告成功（即所有测试的成功）或失败（即一个或多个测试的失败），但它也应该创建一个详细说明任何失败的日志文件，以便为分析和调试提供一个基础。

幸运的是，如果我们把所有应用测试实现为JUnit测试案例，就有一种很容易的方法可用来实现这一级别的测试自动化，以及把测试运行集成到应用生成过程中。Ant生成工具与JUnit平滑地集成在了一起。使用Ant运行一组测试，记录测试结果，以及提供一个简单报告指出所有测试是否已被提供是有可能的。例如，下列Ant任务看上去在一个完整的源树下面，并作为一个测试来运行任何一个其名称匹配于XXXXTestSuite.java的测试类。成功或失败将被显示到控制台上，而一个详细的日志将被创建。

```
<target name="fwtests" depends="build-fwtests">
    <tstamp/>

    <mkdir dir="${junit.reports.dir}" />
    <mkdir dir="${junit.reports.dir}/framework-${DSTAMP}-${TSTAMP}" />

    <junit printsummary="yes" haltonfailure="yes">
        <classpath refid="master-classpath" />
        <formatter type="plain" />

        <!-- Convention is that our JUnit test classes have names such as
            XXXXTestSuite.java
        -->
        <batchtest
            fork="yes"
            todir="${junit.reports.dir}/framework-${DSTAMP}-
${TSTAMP}">
            <fileset dir="${framework-test.dir}">
                <include name="**/*TestSuite.java" />
            </fileset>
        </batchtest>
    </junit>
</target>
```

要想了解较详细的信息，请参考Ant技术资料。但是，本例中最重要的Ant任务是<batchtest>任务，该任务运行许多JUnit测试案例。在本例中，<fileset>子元素指定一个通配符（**/*TestSuite.java），用来选择我们希望运行的那些测试类的名称。<batchtest>任务的todir属性指定一个日志应该被写到的一个目录。

使用这样的Ant目标值来运行一个测试，意味着我们可以轻松地利用单个命令为一组包或整个应用执行所有测试。我们可以在每个修改或隐错纠正之后或者经常地运行所有测试。

要想确保测试定期运行，测试应该执行得快速是必不可少的。一般说来，所有测试都应该在10分钟内执行完：如果可能，越快越好。如果测试不能在这个时间级内执行完，就可

能需要再制作该测试集来改进它的性能。如果有些测试将不可避免地占用较长的时间，应该考虑把该测试集分解成提供立即反馈的测试和可以被整夜运行的测试。

测试的补充方法

虽然测试始终是必不可少的，但有一些补充方法可以用做总QA策略的一部分。

J2SE 1.4把断言（Assertion）引进了Java语言中。一个断言就是一条包含一个布尔表达式并且程序员认为该表达式在本代码内的此处必须是真的语句。断言机制在一个失败的断言被遇到时异常结束程序执行，因为这表明了一个非法状态。断言的目标是通过在运行时尽可能快地标识出错误状态来最大限度地降低一个程序含有错误的可能性。关于Java 1.4断言机制的描述，请参见<http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>站点。`assert`关键字用在断言的前面，如下所示：

```
assert booleanConditionalThatMustBeTrue;
```

断言的概念在Java出现以前就已经存在了很久。例如，David Gries所编著的“*The Science of Programming*”（1981春季出版）一书中就讨论过断言，该书建议应该设法证明程序是正确的，而不是证明调试是正确的。

和测试案例不同，断言不一定非要与代码做什么有关。断言可以基于低级的实现细节，比如专用变量的值，而这些实现细节没有被相关对象暴露出来。这也是断言补充而不是抵触单元测试的使用的一个原因。就笔者的观点而言，断言的正确用法是检查代码如何工作，而不是检查代码做什么。它们是替代白箱测试的一种杰出方法。断言对代码的用户应该是不可见的；它们与相关类的契约没有任何关系。

断言经常涉及到不变量（invariant）。一个不变量就是一个始终为真的谓词。常见的例子是类不变量（class invariant，以一个类的成员数据为基础的不变量），以及循环不变量（loop invariant，在一个循环的每次迭代之前和之后都为真的变量）。如果一个不变量没有保持不变，相关的对象或方法就含有一个隐错或者被不正确地使用。使用不变量的一种常见方法是让每个对类的数据做修改的方法在刚返回之前断言一个类不变量。

假设我们只允许通过类的方法对类数据进行访问——好设计的一个必要部分，我们可以确信不变量在每个方法的执行刚开始时保持不变。因此，一个方法结束时的一个断言失败说明这个方法中有一个隐错，而不是说明在这个方法被调用之前该对象正处于一个非法状态——一个跟踪起来可能会很困难的问题。多个类不变量可以被收集到一个专用布尔方法中，进而能使它们在必要时被作为一个单独的单元来更新，并像下面这样被调用：

```
assert classInvariants();
```

基于实例数据来断言不变量的一个方法看起来可能会像下面这样：

```
private boolean classInvariants() {  
    return this.name != null &&  
        this.count >= 0 &&  
        this.helper != null;  
}
```

把类不变量标识出来通常是一件值得努力做的事情。这不仅能使我们编写较强大的断言，而且还有助于增强对类本身的了解。Gries主张“利用产生这种方法的那些试验思想，开发一个并进的程序及其试验模型！”不变量的正确使用还可能会挖掘出单元测试所遗漏的隐错；有时，对一个单独的方法所进行的单元测试可能会工作正确，但疏忽了这样一个事实：该方法已经破坏了定义它的那个对象的状态。

和测试案例不同，断言被集成到应用代码中。通过引进新的assert关键字，Java 1.4保证断言的自由使用应该不存在任何性能损失：直至单个的类级别，在运行时启用或禁用断言都是有可能的。在Java中使用断言一直是可能的，只需使用如下这样的代码即可：

```
if (orderCount < 0)
    throw new RuntimeException("Assertion Failed: Illegal order count of " +
        + i + " when about to process orders");
```

有时，一个静态的Assertions类用来提供方便的断言方法，就像JUnit测试框架所做的那样。不过，这是相当罗嗦的，不是一个广泛使用的Java风格，而且具有断言条件必须总是被计算的缺点。在上述简单的示例中，性能代价是可以忽略不计的，但更复杂的断言条件计算起来可能会很慢。可以进行这样一项检查：断言在每个断言得到计算之前已经被启用。但是，这增加了进一步的复杂性，并且是笨拙的。因此，Java 1.4中的语言级断言机制是一大进步。

但需要注意的是，在断言关系到一个类的使用的地方，断言机制是不适用的：这个类应该简单地抛出一个把不正确使用报告给错误调用者的异常。这是这个类的一个API问题，而不只是它的一个内部一致性问题，所以不应该依赖于断言机制（能够被关掉）。

在Java的所有版本中，未经检查异常（unchecked exception）（Java编译程序不强迫我们捕捉的运行时异常）将具有和断言失败相同的效果。这使它们成为了处理“不应该发生”的不可恢复错误的一种方便而又正确的方式。我们将在第4章中较详细地谈一谈已检查和未经检查异常。

J2SE 1.4引进了许多增强，这个新的断言机制就是最重要的增强之一。如果使用Java 1.4，就可以利用断言——它们现在被集成到了该语言中，并提供了一种防止隐错的性能中立方法。要学会有效地使用断言，这是一项需要少许实践才能掌握的技能。

补充应用代码测试的其他习惯包括代码审查（code review）和XP的双编程（pair programming）（这是一种争议较大的方法）。这样的技巧帮助保证我们编写优质代码，而且这样的代码含有隐错的可能性较小，而且更易于测试和调试。在下一章中，我们将了解可以用来保证我们编写优质代码的设计和编程技巧。

小结

测试应该在整个软件生命周期内进行。测试应该是软件开发的一项核心活动。

通常，应该在编写代码之前编写测试案例。不应该只将应用代码看做唯一的移交品，测试案例、技术资料（Javadoc及此类文档）和应用代码应该被保持同步，其中测试案例首

先被编写。如果读者不能为某一代码编写一个测试，说明你对编写该代码本身还了解得不够充分。一个不是最新的测试是没有用处的。

除非测试易于运行，否则它们将不被运行。单元测试能够被自动运行是至关重要的，而且无需翻译结果也是十分重要的——应该有一个指出成功或失败的单个指示。**Ant**能够用来自动化用**JUnit**编写的测试。在项目生命周期的早期，投入少量时间来设立测试标准和简化测试过程，以便所有开发人员都能有效地编写和执行测试是很重要的。

尤其是在**Web**应用的情况下，压力测试是至关重要的。令人不满意的性能在企业软件开发中是一个严重的风险，应该及早地通过一个概念试验模型并在修改应用设计的代价还不十分高时解决掉。

测试**J2EE**应用涉及许多不同的测试。我们已经考虑过了下列这些测试策略：

- 使用**JUnit**对类进行单元测试（从所有体系结构层中）。正如我们已经见过的，简单、易使用的**JUnit**测试框架能够用做我们的测试策略的基础。单元测试通常是最主要的测试类型。
- 使用运行在一个客户**JVM**中的**JUnit**测试集来测试带有远程接口的**EJB**。
- 使用**Cactus**（一个建立在**JUnit**之上的、允许测试被运行在应用服务器内的开放源框架）来测试带有本地接口的**EJB**。
- 使用**JUnit**和**Servlet API**接口的测试实现（比如**HttpServletRequest**和**HttpServletResponse**）对**Web**层类进行单元测试。这使**Web**层类能够在应用服务器外面被测试，进而使测试变得更方便。
- 通过诸如**HttpUnit**之类的“屏幕碎片”**Web**工具对**Web**接口进行接受度测试。
- 数据库操作的负载和压力测试。
- **EJB**构件接口的负载和压力测试。
- **Web**接口的负载和压力测试。

我们还讨论了将应用设计成能够让核心功能度对照普通Java类而不是对照更难测试的**J2EE**构件来测试的愿望。

仅有测试不足以保证代码是无隐错的。其他有用的测试包括：在应用代码中使用断言，以及引进常规代码审查的原则。健全的编程和设计原则可以极大地减小隐错的概率。

虽然本章中所讨论的这些概念适用于我们所使用的任何工具，但笔者将焦点集中在了简单的免费工具。这些工具是普遍存在的，并且得到了广泛了解，但是还有一些更先进的商业测试工具，尤其是可用于**Web**应用的商业测试工具，而这些工具已超出了本章的讨论范围。

第4章 J2EE项目的设计技术与编程标准

由于J2EE应用往往大而复杂，所以遵守正确的OO设计原则，采用一致的编程标准，以及利用现有投入（无论是我们自己的，还是第三方的）是非常重要的。在本章中，我们将依次了解这些重要的方面。

前两个方面关系到对象设计级和代码级的代码质量。我们正设法实现什么？优质代码是什么？下面这些都是优质代码的几个特征：

- 优质代码无需大修改也是可扩展的。也就是说，不用把代码搞得一团糟即可容易地给它添加特性。
- 优质代码是易阅读和易维护的。
- 优质代码有齐备的文字资料。
- 优质代码使编写围绕它的劣质代码变得很困难。例如，对象会暴露鼓励有效使用的易懂、易使用接口。优质代码和劣质代码都会繁殖。
- 优质代码是易测试的。
- 优质代码是易调试的。请记住，即使一段代码工作完美，如果它没有接受过调试，它仍可能有问题。如果开发人员正在设法跟踪不完美代码中的一个错误，并且栈跟踪进入完美但不明确的代码中就失踪了，怎么办？
- 优质代码不含有代码重复。
- 优质代码被重用。

编写实现这些目标的代码是很困难的，尽管Java给笔者的帮助比其他任何流行语言所给的帮助都大。

自从1996年开始使用该语言以来，笔者已经编写和调试过大量Java代码（以及在此之前大量的C和C++代码），但笔者现在仍在学习。笔者并不想假装本章含有所有的答案，并且有许多见解，但本章有希望提供一些指导和有用的精神食粮。

我们不仅必须保证我们编写代码正确，而且还必须保证我们编写正确的代码，进而在任何适当的地方利用已有的解决方案。这意味着开发团队必须紧密地工作来避免努力的重复，也意味着设计师和主要开发人员必须保持对第三方解决方案（比如开放源产品）的最新了解。

和本书一样，本章的焦点也将放在J2EE 1.3上。不过，本章中的相关地方也将讨论J2SE 1.4中的语言和API改进，因为J2SE 1.4已经可用，甚至能和一些J2EE 1.3应用服务器一起使用。

J2EE应用的OO设计推荐标准

发生这样的事情是有可能的：将一个J2EE应用设计得如此之差，以致它即便在单个对象级上都含有编写优美的Java代码，但仍将被认为是一个失败。一个具有优秀总体设计但拥

有极差实现代码的J2EE应用将是一个同样惨的失败。令人遗憾的是，许多开发人员花费太多的时间来应付那些J2EE API，而花费太少的时间来保证他们坚持好的编程习惯。Sun的所有J2EE示例应用似乎都体现了这一点。

根据笔者的经验，坚持遵守好的OO原理不是迂腐行为：这确实会带来真正的好处。

OO设计比任何一项特定的实现技术（比如J2EE甚至Java）都重要。好的编程习惯和健全的OO设计使好的J2EE应用具有坚实的基础。劣质的Java代码就是劣质的J2EE代码。

有些“编程标准”问题——尤其是和OO设计有关的那些编程标准问题位于设计与实现之间的界线上，比如设计模式的使用。

下一节将探讨笔者曾经见到的在大型代码库中引起麻烦的一些问题，尤其是笔者还见到过的其他地方曾经谈及过的问题。这是一个巨大领域，所以本节决不是完整的。有些问题是见解问题，尽管笔者将设法使读者相信笔者的见解。

请抓住一切机会向贵组织内外的其他人所编写的优质（和劣质）代码学习。公共领域内的有用来源包括成功的开放源项目和核心Java库中的代码。如果许可允许，反编译商业产品的有趣部分是可能的。与亲自漫无目的地寻找某个特定问题的解决方案相比，专业的程序员或设计师更重视学习和发现最佳解决方案。

利用接口来实现松耦合

经典的Gang of Four设计模式图书（下文简称“GoF”）所提倡的“可重用面向对象设计的第一条原则”是：“编程到一个接口，不要编程到一个实现”。令人高兴的是，Java使遵守这条原则变得非常容易（而且自然）。

要编程到接口，不要编程到类。这分离了接口与它们的实现。在对象之间使用松耦合有助于灵活性。为了获得最大的灵活性，将实例变量和方法参数声明成具有所需的最小具体类型。

由于J2EE应用的规模，使用基于接口的体系结构在J2EE应用中特别重要。编程到接口而非具体类增加了少许复杂性，但回报远远超过了投入。通过一个接口调用一个对象有一点点性能损失，但这在实践中很少是一个问题。

一个基于接口的方法有许多优点，其中的几个优点如下所述：

- 修改任一应用对象的实现类同时又不影响调用代码的能力。这使我们能够在不破坏其他构件的同时参数化一个应用的任一部分。
- 实现接口方面的总自由度。不必束缚于一个继承性分级结构。但是，通过在接口实现中使用具体的继承性来实现代码重用仍是有可能的。
- 必要时提供应用接口的简单测试实现与桩基实现的能力，进而使其他类的测试变得更方便，而且使多个团队能够在他们对接口取得了一致意见之后并行地工作。

采用基于接口的体系结构也是保证一个J2EE应用是可移植的最佳手段，但仍能利用供应商特有的优化和增强。

基于接口的体系结构可以与配置反射（reflection for configuration）的使用有效地组合起来（参见下文）。

首选对象合成而非具体继承性

Gang of Four设计模式图书中所强调的第二条基本原则是：“首选对象合成而非类继承性”。很少有开发人员欣赏这个明智的忠告。

和C++之类的许多老式语言不同，Java在语言级上区分具体继承性（concrete inheritance，来自一个超类的方法实现和成员变量的继承性）和接口继承性（interface inheritance，接口的实现）。Java只允许来自同一个超类的具体继承性，但一个Java类可能会实现任意个接口（当然包括它在一个类分级结构中的祖先所实现的那些接口）。虽然多具体继承性（就像C++中所允许的那样）是最佳设计方法的情况不多见，但Java更好地避免了可能由允许这些少见的合理使用而产生的复杂性。

具体继承性被新学OO的大多数开发人员狂热地采用，但它有许多缺点。类继承性是严格的。对类实现的一部分做修改是不可能的。相反，如果该部分被封装在一个接口中（使用我们在下文中将讨论的委托和Strategy设计模式），这个问题可以被避免。

对象合成（object composition，其中新的功能通过装配或合成对象来获得）比具体继承性更灵活，而且Java接口使委托变得很自然。对象合成允许一个对象的行为在运行时被更改（通过把对象行为的一部分委托给一个接口，并允许调用者设置接口的实现）。Strategy和State设计模式依赖于这种方法。

为了澄清这种区别，让我们来看一看我们想用继承性达到什么目的。

抽象的继承性允许多态性（Polymorphism）：运行时用相同接口替代对象的一种可替换性。这提供了面向对象设计的主要价值。

具体继承性既允许多态性，又允许便利的实现。代码能够从一个超类中被继承。因此，具体继承性是一个实现问题，而不只是一个单纯的设计问题。具体继承性是任何OO语言的一个有用特性；但是，它很容易被过度使用。具体继承性方面的常见错误如下所述：

- 当我们可能需要一个简单接口的实现时，却强迫用户扩展一个抽象或具体类。这意味着我们使用户代码丧失了它的继承性分级结构的权利。如果正常情况下一个用户将需要它自己的定制超类是毫无道理的，我们可以为子类化而提供方法的一个便利抽象实现。因此，这种接口方法不抵触便利超类的规定。
- 通过子类调用超类中的助手方法，使用具体继承性来提供助手功能度。如果继承性分级结构外部的类需要助手功能度，怎么办呢？使用对象合成，以便助手是一个独立的对象，并且能够被共享。
- 使用抽象类来代替接口。抽象类在得到正确使用时是非常有用的。Template Method设计模式（下文讨论）通常被实现成带有一个抽象类。但是，一个抽象类不是一个接口的替代物。它通常是一个接口的实现中的一个便利步骤。不要使用抽象类来定义一个类型。这是导致陷入Java缺少多具体继承性问题的一大原因。令人遗憾的是，那些核心Java库就是这方面的劣质例子，因而常常在最适用接口的地方却使用抽象类。

接口在保持简单时是最有价值的。一个接口越复杂，把它建模成一个接口就越没有价值，因为开发人员将被迫扩展一个抽象或具体实现来避免编写过多的代码。这是正确的接口粒度显得至关重要的--种情况；接口分级结构可以与类分级结构分开，以便一个特定的类只需要实现它所需要的准确接口。

接口继承性（即接口的实现，而不是指从具体类中的功能度继承）比具体继承性灵活。

这意味着具体继承性是一个不好的东西吗？决不是；具体继承性是OO语言中实现代码重用的一种强有力方法。不过，最好是把它看做一种实现方法，而不要把它当做一种高级设计方法。它是我们应该选用的东西，而不是一个应用的总体设计所强行要求的东西。

Template Method设计模式

具体继承性的一种有效使用是实现Template Method设计模式。

Template Method设计模式（GoF）解决一个常见问题：我们知道一个算法的步骤和这些步骤应该被执行的顺序，但不知道如何执行所有这些步骤。Template Method模式解决方案是指把我们不知道如何执行的各步骤封装成抽象方法，并提供一个按正确顺序调用它们的抽象超类。这个抽象超类的具体子类实现执行各步骤的那些抽象方法。关键概念是，控制工作流程的是该抽象基类。公用的超类方法通常是最终的：顺从子类的抽象方法是受保护的。这有助于减小错误的概率：子类被要求做的只是履行一个明确的契约。

工作流程逻辑到抽象超类的集中化就是控制反转（Inversion of control）的一个例子。和传统类库中不同（在传统类库中，用户代码引用库代码），在这种方法中，超类中的框架代码引用用户代码。这种方法也叫做好莱坞规则（Hollywood principle）：“不要给我打电话，我会给你去电话”。控制反转对框架（Framework）是必要的，因为框架往往频繁地使用Template Method模式。

例如，请考虑一个简单的定单处理系统。该业务关系到根据单个物品的价格计算购买价钱，进而核实该顾客是否被允许花这笔钱数，并在必要时运用一个折扣。像RDBMS那样的某个持久性存储器必须被更新以体现一次成功的购买，并且必须被查询以获得价钱信息。但是，最理想的做法是把这个操作与业务逻辑的步骤分开。

AbstractOrderEJB超类实现业务逻辑，而该业务逻辑包括核实该顾客并不打算超过他们的花钱限制，以及给大定单运用一个折扣。公用的placeOrder()方法是最终的，以便这个工作流程不能被子类修改（或破坏）：

```

public final Invoice placeOrder(int customerId, InvoiceItem[] items)
    throws NoSuchCustomerException, SpendingLimitViolation {

    int total = 0;
    for (int i = 0; i < items.length; i++) {
        total += getItemPrice(items[i]) * items[i].getQuantity();
    }

    if (total > getSpendingLimit(customerId)) {
        getSessionContext().setRollbackOnly();
        throw new SpendingLimitViolation(total, limit);
    }
    else if (total > DISCOUNT_THRESHOLD) {
        // Apply discount to total...
    }

    int invoiceId = placeOrder(customerId, total, items);
    return new InvoiceImpl(iid, total);
}

```

笔者突出显示了这个方法中的3行代码，这些代码调用必须由子类来实现的受保护抽象“模板方法”。这些将在AbstractOrderEJB超类中被定义，如下所示：

```
protected abstract int getItemPrice(InvoiceItem item);  
  
protected abstract int getSpendingLimit(customerId)  
throws NoSuchCustomerException;  
  
protected abstract int placeOrder(int customerId, int total,  
InvoiceItem[] items);
```

AbstractOrderEJB类的子类只需要实现这3个方法。它们不必操心业务逻辑。例如，一个子类可能会使用JDBC实现这3个方法，而另一个子类可能会使用SQLJ或JDO实现这3个方法。

Template Method模式的这种使用提供了关系的有效分离。在这里，超类集中于业务逻辑；子类集中于实现基本操作（比如使用一个像JDBC那样的低级API）。由于那些模板方法都是受保护的，而不是公用的，所以调用者不用操心类实现的细节。

由于在接口中而不是在类中定义类型通常会更好，所以Template Method模式常常用做实现接口的一种策略。

抽象超类常常也用来实现一个接口的部分（而不是全部）方法。那些剩余的方法（即在具体实现之间有所不同的方法）留着未实现。这不同于Template Method模式，因为抽象超类不处理工作流程。

要使用Template Method设计模式来捕获一个抽象超类中的一个算法，但使各步骤的实现延迟到子类中。这具有防止隐错的潜在能力：通过一次彻底搞清楚棘手操作并简化用户代码。当实现Template Method模式时，抽象超类必须甄别出可能会在子类之间变化的那些方法，并保证那些方法签名在实现中提供足够的灵活性。

要始终使抽象父类去实现一个接口。Template Method设计模式在框架设计中是特别有价值的（本章最后部分将讨论）。

Template Method设计模式在J2EE应用中是非常有用的，它可以帮助我们在应用服务器间和数据库间实现尽可能大的可移植性，同时又仍利用专有特性。在前面，我们已经见过有时怎样才能把业务逻辑与数据库操作分离开。我们同样可以使用这个模式来提供对具体数据库的足够支持。例如，我们可以有一个OracleOrderEJB和一个DB2OrderEJB，并且它们在各自的数据库中有效地实现抽象模板方法，同时业务逻辑仍是无专有代码的。

Strategy设计模式

Template Method设计模式的一个替代物是Strategy设计模式，后者将把不变行为集中到一个接口中。因此，知道该算法的类并不是一个抽象基类，而是一个使用了助手的具体类，并且该助手实现一个定义各步骤的接口。Strategy设计模式的实现工作量要大于Template Method设计模型，但它更灵活。Strategy设计模式的优点是，它不需要涉及具体继承性。实现各步骤的类没有被迫从一个抽象的模板超类中继承。

下面让我们来看一看如何才能在上例中使用Strategy设计模式。第一步是把那些模板方法转移到一个接口中，该接口看起来将像下面这样：

```
public interface DataHelper {
    int getItemPrice(InvoiceItem item);
    int getSpendingLimit(customerId) throws NoSuchCustomerException;
    int placeOrder(int customerId, int total, InvoiceItem[] items);
}
```

这个接口的实现不必子类化任何一个特定的类，我们有最大可能的自由度。

现在，我们可以编写一个依赖于这个接口的一个实例变量的具体OrderEJB类。我们还必须提供一种设置这个助手的手段：或者在构造器中，或者通过一个bean属性。在目前的例子中，笔者选择一个bean属性：

```
private DataHelper dataHelper;
public void setDataHelper(DataHelper newDataHelper) {
    this.dataHelper = newDataHelper;
}
```

placeOrder()方法的实现与使用Template Method设计模式的那个版本几乎一致，所不同的只是它在下列突出显示的行中，调用了它不知道在该助手接口的那个实例上如何执行的那些操作：

```
public final Invoice placeOrder(int customerId, InvoiceItem[] items)
throws NoSuchCustomerException, SpendingLimitViolation {
    int total = 0;
    for (int i = 0; i < items.length; i++) {
        total += this.dataHelper.getItemPrice(items[i]) *
            items[i].getQuantity();
    }

    if (total > this.dataHelper.getSpendingLimit(customerId)) {
        getSessionContext().setRollbackOnly();
        throw new SpendingLimitViolation(total, limit);
    } else if (total > DISCOUNT_THRESHOLD) {
        // Apply discount to total...
    }

    int invoiceId = this.dataHelper.placeOrder(customerId, total, items);
    return new InvoiceImpl(iid, total);
}
```

这个版本实现起来比使用具体继承性和Template Method设计模式的版本稍微复杂一点，但更灵活。这是一个在具体继承性与接口委托之间进行折衷的典型例子。

笔者在下列情况下使用Strategy设计模式优先于使用Template Method设计模式：

- 当所有步骤都变化（不只是某些步骤）时。
- 当实现那些步骤的类需要一个独立的继承性分级结构时。
- 当那些步骤的实现可能与其他类相关时（这常常是带有J2EE数据存取的情况）。
- 当那些步骤的实现可能需要在运行时变化时。具体继承性不能接纳折衷情况；委托能。

- 当存在那些步骤的许多不同实现时，或者说预计实现数量将继续增加时。在这种情况下，Strategy设计模式的更大灵活性将几乎肯定是有益的，因为它允许那些实现的最大自由度。

使用回调来实现可扩展性

现在，让我们来考虑“反转控制”的另一种用法：参数化单个操作，同时把控制和错误处理转移到一个框架中。严格地说，这是Strategy设计模式的一种特殊情况：它显得不同是因为涉及到的那些接口是如此简单。

这个模式基于一个或多个回调方法的使用，并且这些方法将由一个执行工作流程的方法类调用。

笔者发现这个模式在处理像JDBC那样的低级API时是有用的。下面这个例子是一个JDBC实用程序类JdbcTemplate的一种分解形式，本书的示例应用中用到了它，而且我们将在第9章中进一步讨论它。

JdbcTemplate实现了一个query()方法，该方法接受一个SQL查询串和一个回调接口的一个实现作为参数，其中该回调接口将针对该查询所生成的结果集的每一行被调用一次。这个回调接口如下所示：

```
Public interface RowCallbackHandler {  
    void processRow(ResultSet rs) throws SQLException;  
}
```

JdbcTemplate.query()方法向调用代码隐藏起获取一条JDBC连接、创建并使用一条语句以及即便在错误的情况下正确释放资源的细节，如下所示：

```
public void query(String sql, RowCallbackHandler callbackHandler)  
throws JdbcSqlException {  
  
    Connection con = null;  
    PreparedStatement ps = null;  
    ResultSet rs = null;  
    try {  
        con = <code to get connection>  
        ps = con.prepareStatement(sql);  
        rs = ps.executeQuery();  
  
        while (rs.next()) {  
            callbackHandler.processRow(rs);  
        }  
  
        rs.close();  
        ps.close();  
    } catch (SQLException ex) {  
        throw new JdbcSqlException("Couldn't run query [" + sql + "]", ex);  
    }  
    finally {  
        DataSourceUtils.closeConnectionIfNecessary(this.dataSource, con);  
    }  
}
```

DataSourceUtils类含有一个助手方法，并且该助手方法能够用来关闭连接，进而高速缓存并记录已遇到的任何SQLExceptions。

在本例中，`JdbcSQLException`扩展了`java.lang.RuntimeException`，而后者意味着调用代码可能会选择捕捉它，但不是被迫的。这在目前状况下是有意义的。例如，如果一个回调处理程序试图获取一个列的值，而这个列在`ResultSet`中不存在，那么调用代码捕捉它将是没有好处的，这明显是一个程序设计错误，而且`JdbcTemplate`记录该异常并抛出一个运行时异常的行为是合乎逻辑的（参见后面关于“异常处理——已检查或未检查异常”的讨论）。

在这种情况下，笔者已把`RowCallbackHandler`接口建模为`JdbcTemplate`类的一个内部类。这个接口只与`JdbcTemplate`类相关，因此这是合乎逻辑的。需要注意的是，`RowCallbackHandler`接口的实现可能是内部类（在无足轻重的情况下，匿名内部类是恰当的），它们也可能是标准的可重用类或标准便利类的子类。

请考虑`RowCallbackHandler`接口的下列实现：执行一个JDBC查询。请注意，该实现没有被迫捕捉在从结果集中提取列值时可能会被抛出的SQLExceptions：

```
class StringHandler implements JdbcTemplate.RowCallbackHandler {
    private List l = new LinkedList();

    public void processRow(ResultSet rs) throws SQLException {
        l.add(rs.getString(1));
    }

    public String[] getStrings() {
        return (String[]) l.toArray(new String[l.size()]);
    }
}
```

这个类可以按照如下所示来使用：

```
StringHandler sh = new StringHandler();
jdbcTemplate.query("SELECT FORENAME FROM CUSTMR", sh);
String[] forenames = sh.getStrings();
```

这3行说明了使用`JdbcTemplate`的代码怎么才能把焦点集中在业务问题上，同时又不用操心JDBC API。被抛出的任何SQLExceptions将由`JdbcTemplate`来处理。

这个模式不应该被过度使用，但会非常有用。下列优缺点暗示了相关的折衷方案。

优点：

- 这个框架类可以执行错误处理和资源的获得与释放。这意味着棘手的错误处理（就像使用JDBC所要求的那样）可以只被编写一次，而且调用代码更简单。涉及的错误处理和清理越复杂，这种方法就越有吸引力。
- 调用代码不需要处理低级API（比如JDBC）的细节。这是最理想的，因为处理细节的代码是易隐含错误的和罗嗦的，进而遮盖了应用代码应该重点关注的业务问题。
- 一个控制流函数（本例中的`JdbcTemplate.query()`）可以与各种各样的回调处理程序一起使用，以执行不同的任务。这是实现对使用了低级API的代码进行重用的一种好方法。

缺点：

- 这种习惯语法的直观程度比不上让调用代码来处理执行流程本身。因此，如果没有一种适当的替代语法，代码可能会更难以理解和维护。
- 我们需要为回调处理程序创建一个对象。
- 在少数情况下，性能可能会被通过一个接口来调用该回调处理程序的要求所削弱。与那些JDBC操作本身所占用的时间相比，上述例子的系统开销可以忽略不计。

这个模式在回调接口非常简单时是最有价值的。在本例中，由于RowCallbackHandler接口只含有一个方法，所以它实现起来非常容易，进而意味着像匿名内部类那样的实现选择可以用来简化调用代码。

Observer设计模式

和接口的使用一样，Observer设计模式可以用来无修改地分离构件和实现可扩展性（参见Open Closed Principle）。它也有助于实现关系分离（separation of concerns）。

例如，请考虑一个处理用户登录的对象。一个用户的登录尝试可能会产生几种结果：成功的登录，由于一个不正确的密码而导致的失败登录，由于一个不正确的用户名和密码而导致的失败登录，由于未能连接到保存登录信息的数据库而导致的系统错误。

让我们设想这样一种情景：我们有一个正工作在生产中的登录实现，但进一步的需求意味着如果发生了给定数量的系统错误，该应用应当给一名管理员发电子邮件，并且应当维护一个由输入不正确的密码连同用于那些相关用户的正确密码所构成的列表，以便帮助制定信息来协助用户避免常见错误；我们还想知道用户登录活动的那些高峰期（与Web站点上的一般性活动相对）。

这个功能度可以完整地被添加到实现登录的那个对象上。我们应该有单元测试，并让这些测试证实这一添加没有破坏当前功能度，但这种方法不提供好的关系分离（处理登录的那个对象为什么应该需要知道或获得管理员的电子邮件地址，或者说知道如何发送一个电子邮件呢？）。随着越来越多的特性（或形态）被添加，登录流程本身的实现——这个构件的核心职责，将会被淹没在处理这些特性的大量代码之中。

我们可以使用Observer设计模式更优雅地解决这个问题。应用事件（application events）可以被通知给观察者（Observer）（或监听者）。该应用必须提供一个事件出版者（event publisher）（或者使用一个提供事件出版者的框架）。监听者可以注册，以便事件被通知给它：工作流程代码必须做的就是出版可能有关的事件。事件出版类似于生成日志消息，因为它不影响应用代码的工作。在上述例子中，事件将会包括：

- 已尝试过的登录，其中含有用户名和密码
- 系统错误，其中包括令人不愉快的异常
- 登录结果（成功和失败以及原因）

事件通常包括时间戳。

现在，我们可以通过使用不同的监听者给管理员发送关于系统错误的电子邮件来实现明确的关系分离；对一个失败的登录做出反应（把它添加到一个列表上）；以及收集关于登录活动的性能信息。

Observer设计模式用在核心Java库中。例如，Java组件（JavaBean）可以出版属性修改事件。在我们自己的应用中，我们将在一个很高的级别上使用Observer设计模式。相关事件可能与应用级操作有关，与设置一个Bean属性之类的低级操作无关。

另外，请考虑这样一个需求：收集关于一个Web应用的信息。我们可以把复杂的性能监视建立到Web应用框架的代码中（例如，任何控制器小服务程序），但这将需要修改那些类，如果我们将来需要不同的性能统计数字。比较好的做法是出版诸如“请求已收到”和“请求已完成”之类的事件（后者包括成功或失败状态），并把性能监视的实现留给惟一关心它的监听者。这是Observer设计模式怎样才能用来实现有效关系分离的一个例子。这相当于Aspect-Oriented Programming（面向形态的程序设计），我们将在后面的“使用反射”一节中简要讨论这种程序设计。

不要过分追求Observer设计模式：只有当松耦合监听者将真正有可能需要知道一个工作流程时，Observer设计模式才是必需的。如果我们在每个地方都使用Observer设计模式，我们的业务逻辑将会被淹没在事件出版代码的沼泽中，并且性能将会被显著降低。只有重要的工作流程（比如上述例子的登录过程）才应该生成事件。

使用Observer设计模式时的一个警告：监听者迅速返回是至关重要的。流氓型监听者会锁定一个应用。尽管事件出版系统用一个不同的线程调用观察者是可能的，但这对将迅速返回的大多数监听者来说是不经济的。大多数情况下，一种较好的选择是让监听者来承担迅速返回或把长时间运行任务甩出到各自线程中这项责任。观察者还应当避免共享应用对象上的同步，因为这可能会导致堵塞。监听者必须是线程安全的。

Observer设计模式在单服务器的部署中比在多服务器的聚类部署中有用，因为它只允许我们在单个服务器上出版事件。例如，使用Observer设计模式来更新一个数据高速缓存器是不安全的，因为这样一个更新将只适用于单个服务器。但是，Observer设计模式在一个聚类中仍会十分有用。例如，上面讨论的那些应用在一个聚类环境中都将是有效的。JMS可以用于聚类级事件出版，但代价是更大的API复杂性和一个更大的性能开销。

根据笔者的经验，Observer设计模式在Web层中比在EJB层中有用。例如，在EJB层中创建线程是不可能的（同样，JMS是替代方案）。

在第11章中，我们将了解如何在一个应用框架中实现Observer设计模式。用在本书示例应用中的应用框架基础结构提供了一种事件出版机制，进而使这里所描述的那些方法能够在不需要一个应用实现任何“探测”的条件下被实现。

考虑合并方法参数

有时候，把一个方法的参数封装到单个对象中是一个不错的主意。这或许能增强易读性，并简化调用代码。请考虑一个像下面这样的方法签名：

```
public void setOptions(Font f, int lineSpacing, int linesPerPage,  
                      int tabSize);
```

通过把这些多个参数组合为单个对象，我们可以简化这个签名，就像下面这样：

```
public void setOptions(Options options);
```

主要优点是灵活性。我们不必破坏签名来增加另外的参数：可以给该参数对象添加另外的属性。这意味着我们不必破坏已有调用者中对被添加特性不感兴趣的代码。

和C++不同，由于Java不提供默认参数值，所以这是一种能使客户软件简化代码的好方法。让我们假设所有（或大部分）的参数有默认值。在C++中，我们可以在方法签名中编码这些默认值，从而使调用者能够忽略它们中的某些，就像下面这样：

```
void SomeClass::setOptions(Font f, int lineSpacing = 1, int linesPerPage = 25,
                           int tabSize = 4);
```

这在Java中是不可能的，但我们可以用默认值填充该对象，从而使子类能够使用像下面这样的语法：

```
Options o = new Options();
o.setLineSpacing(2);
configurable.setOptions(o);
```

在这里，`Options`对象的构造器将所有字段都设置成默认值，所以我们只需要修改不同于默认值的那些字段。如果必要，我们甚至可以使该参数对象成为一个接口，以允许更灵活的实现。

这种方法对构造器特别适用。当一个类有许多构造器，并且子类可能会只为保持超类构造器排列而应付过多的工作时，意味着需要这种方法。子类可以替代地使用超类构造器的参数对象的一个子类。

`Command`设计模式就使用这种方法：一条命令实际上是一个统一的参数集合，其中参数的协同处理比个别处理更容易。

参数统一的缺点是许多对象的潜在创建，因而增加了内存使用量和无用数据收集的要求。对象耗用大量空间，而基本元素不耗用大量空间。这是否要紧取决于该方法将被调用的频繁程度。

如果方法调用有可能是远程的（即对一个EJB的远程接口所进行的一次调用），把多个参数统一到单个对象中有时会在J2EE应用中引起性能下降，因为编组或取消编组几个基本参数将总是快于编组和取消编组一个对象。但是，这不是问题，除非该方法被特别频繁地调用（这可能暗示了极差的应用分区——若能避免，我们也不希望做频繁的方法调用）。

异常处理——已检查或未检查异常

Java区分两种类型的异常。已检查（checked）异常扩展`java.lang.Exception`，编译程序要求这类异常被捕捉或者被明确地重新抛出。未检查（unchecked）或运行时（runtime）异常扩展`java.lang.RuntimeException`，并且不需要被捕捉（尽管可以按照和已检查异常相同的方式被捕捉和增殖调用栈）。Java是支持已检查异常的惟一主流语言。例如，所有C++和C#异常都相当于Java的未检查异常。

首先，让我们来看一看Java中关于异常处理的公认见解。这方面内容在Java Tutorial (<http://java.sun.com/docs/bokks/tutorial/essential/exceptions/runtime.html>) 内关于异常处理的

…节中有详细描述，该指南中建议在应用代码中使用已检查异常。

由于Java语言不要求方法捕捉或指定运行时异常，所以程序员会情不自禁地编写抛出运行时异常的代码，或者使他们的所有异常子类都从RuntimeException中继承。这两条编程捷径都允许程序员在编写Java代码时不用操心来自编译程序对错误的喋喋不休，也不用操心指定或捕捉任何异常。虽然这看起来好像对程序员是方便的，但回避了Java的捕捉或指定需求的意图，并且会给使用你的类的程序员带来问题。

已检查异常提供与这样一个请求的操作有关的有用信息：调用者合法地指定了这个请求，但调用者可能已完全失去了对这个请求的控制，并且调用者需要得到关于这个请求的通知。例如，文件系统现在已满，或远程端已经关闭了连接，或访问权限不允许这个动作。

如果读者抛出一个RuntimeException，或者仅仅因为不想涉及到指定一个运行时异常而创建RuntimeException的一个子类，怎么办呢？很简单，你完全可以在不用指定自己这么做的情况下抛出一个异常。换句话说，避开指定一个方法能够抛出的那些异常是一条途径。这何时合适呢？避开指定一个方法的行为永远合适吗？答案是“几乎从不”。

概括地说，Java正统观点认为：已检查异常应该是标准用法，运行时异常表明编程错误。

笔者曾经同意过这个观点。但是，在编写和处理了数千个捕捉模块之后，笔者最终得出这样一个结论：这一吸引人的理论在实践中未必总是管用。笔者不是独一无二的。自从得出了关于这个题目的个人观点以来，笔者已经注意到经典书籍“Thinking Java”的作者Bruce Eckel也已经改变了他的想法。Eckel现在主张使用运行时异常作为标准用法，并在考虑，作为一个失败的试验，已检查异常是否应该从Java中删去（<http://www.mindview.net/Etc/Discussions/CheckedExceptions>）。

Eckel引用了这样一个观察结果：当一个人看少量代码时，已检查异常似乎是一个才气十足的构思，并且有助于避免许多隐错。但是，经验往往表明，对大量代码来说情况正好相反。有关已检查异常问题的另外讨论，请参见<http://www.octopull.demon.co.uk/java/ExceptionalJava.html>站点上由Alan Griffiths所写的“Exceptional Java”。

使用已检查异常会特有地引起几个问题：

• 太多的代码

开发人员将会因为不得不捕捉他们无法合理地处理的已检查异常（属于“某个东西出了可怕错误”种类）并编写忽略（忍受）它们的代码而感到沮丧。公认的观点：这是无辩护余地的编程原则，但经验表明它发生得没有我们想像的那样频繁。甚至连好的程序员都可能会偶尔忘了正确地“嵌套”异常（关于这方面的详细讨论，请参见下文），进而意味着全栈跟踪被丢失，异常中所包含的信息的价值降低了。

• 难以读懂的代码

捕捉不能被正确地处理的异常并重新抛出它们（被封装在一种不同的异常类型中）没有执行一点有用的功能，反而会使查找实际做某件事的代码变得更困难。正统的观点是这仅使懒散的程序员感到麻烦，并且我们应该忽略这个问题。不过，这也忽略了现实。例如，这个问题明显被核心Java库的设计者考虑过。请设想一下这样一种

恶梦般的情形：必须处理像java.util.Iterator那样的集合接口，如果它们抛出已检查而不是未检查异常。JDO API是使用未检查异常的Sun API的另一个例子。相反，直接处理使用已检查异常的JDBC是十分不方便的。

• 异常的无休止封装

一个已检查异常要么必须被捕获，要么必须在一个遇到它的那个方法的抛出子句中被声明。这留下了这样一个选择：要么重新抛出数量不断增长的异常，或者说捕捉低级异常，要么重新抛出被封装在一个较高级的新异常中的它们。这是我们所希望的，如果我们通过这么做来添加有用的信息。但是，如果较低级异常是不可恢复的，封装它将达不到什么目的。我们不是将有调用栈的一个自动展开，就像未检查异常的情况那样，而是将有调用栈的一个等效手工展开：通过在涉及到的每个类中使用几行附加的无意义代码。大体上说，正是这个问题提醒了笔者重新思考对待异常处理的态度。

• 易毁坏的方法签名

一旦许多调用者使用一个方法，添加一个额外的已检查异常到该接口上将需要许多代码被修改。

• 已检查异常对接口不一定管用

就以正充满了Java Tutorial的那个文件系统为例。这似乎没有问题，如果我们正在谈论我们知道的一个处理文件系统的类。如果我们正在处理一个接口，并且这个接口只预示把数据存储在某个地方（或许存储在一个数据库中），怎么办呢？我们不希望把对Java I/O API的依赖性硬编码到一个可能有不同实现的接口中。因此，如果我们想使用已检查异常，就必须为该接口创建一个存储器独立的新异常类型，并把文件系统异常封装到它里面。这是否合适同样取决于该异常是不是可恢复的。如果它不是可恢复的，我们就做了多余的工作。

在这些问题中，许多问题可归因于代码处理不了的代码捕捉异常的问题，以及被迫重新抛出已封装的异常。这是十分不方便的，易出错误的（易失去栈跟踪），并且没有任何有益的用处。这此类情况下，较好的做法是使用一个未检查异常。这种做法将自动展开调用栈，并且对“某个东西有可怕错误”种类的异常来说，这才是正确的行为。

笔者采取了一种比Eckel稍正统的观点，因为笔者认为已检查异常有一定的用武之地。在一个异常相当于来自方法的一个可替代返回值的地方，这个异常无疑应该被检查，并且该语言能帮助实施这一点就再好不过了。但是，笔者觉得传统Java方法过分强调了已检查异常。

已检查异常要比错误返回码（许多老式语言中使用）好得多。迟早（或许是不久），人们将不能检查一个错误返回值；使用编译程序来实施正确的错误处理是一件好事。同参数与返回值一样，这样的已检查异常对一个对象的API来说是整体的一个不可分部分。

但是，笔者不建议使用已检查异常，除非调用者有能力处理它们的能力。特别是，已检查异常不应该用来表示“某个东西出了可怕错误”，因为预计调用者处理不了。

如果调用代码能够对该异常做些切合实际的事情，请使用一个已检查异常。如果该异常是致命的，或者调用者捕捉它将没有益处，请使用一个未检查异常。请记住，可以依赖一个J2EE容器（比如一个Web容器）来捕捉并记录未检查异常。

对于已检查和未检查异常之间的选择，笔者建议表4.1的指导方针。

表4.1

问题	示例	答案为是时的建议
所有调用者都应该处理这个问题吗？该异常本质上是该方法的第二个返回值吗？	在一个processInvoice()方法中使用过度的限制	定义和使用一个已检查异常，并利用Java的编译时支持
将只有少数调用者需要处理这个问题吗？	JDO异常	扩展RuntimeException。这给调用者留下了捕捉该异常的选择，但不强迫所有调用者都捕捉它
某个东西出现了可怕错误吗？	一个业务方法失败，因为它不能连接到应用数据库	扩展RuntimeException。我们知道调用者除了把该错误通知给用户外，无法做任何有用的事情
仍不清楚吗？		扩展RuntimeException。用文字详细说明可能被抛出并让调用者决定它们希望捕捉的那些异常

在包级决定每个包将怎样使用已检查或未检查异常。用文字详细说明使用未检查异常的决定，因为许多开发人员将预料不到该决定。

使用未检查异常的惟一危险是那些异常可能没有被充分地用文字加以详细说明。当使用未检查异常时，请务必用文字详细说明可能会从每个方法中被抛出的所有异常，进而使调用代码能够选择捕捉你预计将要致命的异常。理想情况下，编译程序应该对所有异常（已检查和未检查异常）进行Javadoc化。

如果分配在所有环境下都必须被释放的资源，比如JDBC连接，不要忘了使用一个终结模块来保证清理，无论你需不需要捕捉已检查异常。请记住，一个终结模块甚至能在没有一个捕捉块的情况下使用。

有时，提前避免运行时异常的一个理由是，一个未捕捉到的运行时异常将会删除当前的执行进程。这在某些情况下是一个站得住脚的理由，但在J2EE应用中常常不是问题，因为我们很少控制线程，而是把控制权留给应用服务器去取舍。应用服务器将捕捉并处理应用代码中未捕捉到的运行时异常，而不是把它们推给JVM去取舍。EJB容器内的一个未捕捉到的运行时异常将会致使该容器删除当前的EJB实例。不过，如果该错误是致命的，这通常才有意义。

最后，使用已检查还是未检查异常是一个见解问题。因此，不仅用文字详细说明被采取的这种方法是至关重要的，而且尊重其他人的习惯也是至关重要的。虽然笔者一般

更喜欢使用未检查异常，但在维护或增强只喜欢使用已检查异常的其他人所编写的代码时，笔者遵守他们的风格。

好的异常处理原则

无论我们使用了已检查异常，还是使用了未检查异常，我们仍将需要解决“嵌套”异常的问题。一般说来，这种情况只会发生在这样的时候：我们被迫捕捉我们无法处理，但又需要重新抛出以便尊重当前方法的接口的一个已检查异常。这意味着我们必须把原“嵌套”异常封装在一个新异常内。

像javax.servlet.ServletException那样的标准库异常提供了这样的封装功能。但是，对于我们自己的应用异常，我们将需要定义（或者使用现有的）接受一个“根起因”异常为一个构造器参数的自定义异常超类，把它暴露给需要它的代码，并覆盖printStackTrace()方法来显示全部栈跟踪，其中包括根起因的栈跟踪。一般说来，我们需两个这样的基异常：一个用于已检查异常，另一个用于未检查异常。

这在Java 1.4中不再是必需的，因为它支持所有异常的异常嵌套。我们将在下文中讨论这个重要的增强。

在与本书的示例应用所配的通用基础结构代码中，这两个类分别是com.interface21.core.NestedCheckedException和com.interface21.core.NestedRuntimeException。除了分别从java.lang.Exception和java.lang.RuntimeException中派生而来之外，这两个类几乎是相同的。这两个异常都是抽象类；只有子类型对应用才有意义。下面是NestedRuntimeException的一个完整程序清单。

```
package com.interface21.core;

import java.io.InputStream;
import java.io.PrintWriter;

public abstract class NestedRuntimeException extends RuntimeException {

    private Throwable rootCause;

    public NestedRuntimeException(String s) {
        super(s);
    }

    public NestedRuntimeException(String s, Throwable ex) {
        super(s);
        rootCause = ex;
    }

    public Throwable getRootCause() {
        return rootCause;
    }

    public String getMessage() {
        if (rootCause == null) {
            return super.getMessage();
        }
    }
}
```

```

    } else {
        return super.getMessage() + "; nested exception is: \n\t" +
            rootCause.toString();
    }
}

public void printStackTrace(PrintStream ps) {
    if (rootCause == null) {
        super.printStackTrace(ps);
    } else {
        ps.println(this);
        rootCause.printStackTrace(ps);
    }
}

public void printStackTrace(PrintWriter pw) {
    if (rootCause == null) {
        super.printStackTrace(pw);
    } else {
        pw.println(this);
        rootCause.printStackTrace(pw);
    }
}

public void printStackTrace() {
    printStackTrace(System.err);
}
}

```

Java 1.4在异常处理方面引进了受欢迎的改进。现在不再需要编写可链接的异常，尽管现有的基础结构类（比如上面所示的那些类）将继续毫无问题地工作。新的构造器被添加到java.lang.Throwable和java.lang.Exception上，以便使一个根起因能够被指定在异常构造后面。这个方法可能只被调用一次，而且只有当构造器中没有提供无嵌套异常时，才被调用。

有Java 1.4意识的异常应该实现这样一个构造器：它接受一个可抛出的嵌套异常，并调用新的Exception构造器。这意味着我们始终可以用单行代码来创建和抛出它们，如下所示：

```

catch (RootCauseException ex) {
    throw new MyJava14Exception("Detailed message", ex);
}

```

如果一个异常没有提供一个这样的构造器（比如，因为它是为Java 1.4之前的一个环境所编写的），我们被保证能够使用稍微多一点的代码设置一个嵌套异常，如下所示：

```

catch (RootCauseException ex) {
    MyJava13Exception mex = new MyJava13Exception("Detailed message");
    mex.initCause(ex);
    throw mex;
}

```

当使用嵌套异常解决方案时，比如上面讨论的NestedRuntimeException，请遵守它们自己的标准，而不要遵守Java 1.4标准，以便保证正确的行为。

J2EE中的异常

在J2EE中，有几个需要考虑的特殊问题。

分布式应用将会遇到许多已检查异常。这部分是由于Sun在Java的早期为了使远程调用变成显式调用而做出的决策。由于包括EJB远程接口调用在内的所有RMI调用都抛出java.rmi.RemoteException，所以本地远程透明性是不可能的。这个决策可能是有道理的，因为本地远程透明性是危险的，尤其是对性能来说。但是，它意味着我们必须经常编写代码来处理相当于“某个东西出了可怕错误，而且重试可能是不值得做的”的已检查异常。

保护接口代码（比如小服务程序和JSP页中的代码）使之免受诸如java.rmi.RemoteException之类的J2EE“系统级”异常影响是很重要的。许多开发人员没有认识到这个问题，因而产生了令人遗憾的后果，比如在体系结构层之间创建不必要的依赖性，以及阻止重试本可能已被重试过的操作，如果这些操作在一个足够低的级别上被捕捉。在确实认识到这个问题的开发人员当中，笔者曾经见过两种方法：

- 允许接口构件忽略上述这样的异常，例如，通过编写代码在一个像超类那样的高级别上捕捉它们，其中这个超类的所有子类将处理Web进入请求，并且这个超类允许子类从一个抽象保护方法中抛出某一范围的异常。
- 使用一个客户端外形，其中该外形隐藏起与远程系统的通信，并抛出由业务需求而非远程方法调用所要求的异常（已检查和未检查异常）。这意味着该客户端外形不应该模仿远程构件的接口，因为那些远程构件都将抛出java.rmi.RemoteException。这种方法称做Business Delegate（业务委托）J2EE模式（参见“Core J2EE Patterns”一书）。

笔者认为，上述两种方法中的第二种方法是出色的。它提供了体系结构层的一个明确分离，允许已检查或未检查异常的选择，并且不允许EJB和远程调用的使用太深入地侵入到应用设计中。我们将在第11章中较详细地讨论这种方法。

使异常能够提供信息

确保异常对代码和开发、维护及管理一个应用的人都有用是至关重要的。

请考虑这样一种情况：反映不同问题的同一个类的、但只有通过其消息串来区分的异常。这些异常对捕捉它们的Java代码是没有帮助的。异常消息串具有有限的价值：当这些消息串出现在日志文件中时，它们对解释问题可能是有帮助的，但它们将无法使调用代码正确地做出反应（如果不同的反应是必需的），并且不能依靠它们本身来把它们显示给用户。当不同的问题可能需要不同的动作时，相应的异常应该被建模为一个公用超类的独立子类。有时，该超类应该是抽象的。现在，调用代码将可自由地在相关的细节级别上捕捉异常。

第二个问题（即显示给用户）应该通过在异常中包括错误代码来处理。错误代码可以是数字的，也可以是串的（串代码的优点是它们能够对阅读者有意义的），而且串代码能够驱动被保存在异常外面的显示消息的运行时查找。除非我们能为一个应用中的所有异常使用一个公用基类（如果我们混用已检查和未检查异常，这是一件不可能的事情），否则将需要让我们的异常实现一个ErrorCoded接口或一个像下面这样定义一个方法并具有相似名称的接口：

```
String getErrorCode();
```

在第11章将要讨论的基础结构代码中，`com.interface21.core.ErrorCoded`接口含有这个方法。使用这种方法，我们能够把为终端用户而计划的错误消息与为开发人员而计划的错误消息区分开。异常内的消息（由`getMessage()`方法返回）应该用于记录日志，并且是面向开发人员的。

通过把一个错误代码和异常包括在一起，把要显示给用户的错误消息从异常代码中分离出来。当该是显示异常的时候，这个错误代码会被解析出来：例如，从一个属性文件中。

如果该异常不是针对用户的，而是针对管理员的，我们将需要操心消息格式化或国际化的可能性较小（不过，国际化在某些情况下可能仍是一个问题：例如，如果我们正在开发一个可能由非英语开发人员使用的框架）。

正如我们已经讨论过的，捕捉一个异常并再抛出一个新异常没有一点用处，除非我们增加价值。但是，制作尽可能好的错误消息的要求有时对捕捉并封装来说是一个很好的理由。

例如，下列这条错误消息几乎没有含有有用的信息：

```
WebApplicationContext failed to load config
```

像这样的异常消息一般反映了开发人员在编写消息时偷懒，或者（更糟）只使用单个捕捉模块来捕捉种类广泛的异常（进而意味着捕捉了该异常的代码和该消息的不幸阅读者一样几乎不知道什么东西出了毛病）。

比较好的做法是除了保护栈跟踪之外，还包含与失败的那个操作有关的详细信息。例如，下列这条消息就是一个改进：

```
WebApplicationContext failed to load config: cannot instantiate class com.foo.bar.Magic
```

更好的情况是，一条消息给出关于该进程在失败时正试图做什么的准确信息，以及关于纠正这个问题可能需要做些什么的信息：

```
WebApplicationContext failed to load config from file '/WEB-INF/applicationContext.xml': cannot
instantiate class 'com.foo.bar.Magic' attempting to load bean element with name 'foo' - check that
this class has a public no arg constructor
```

把尽可能多的上下文信息和异常包括在一起。如果一个异常可能从一个程序设计错误中产生，设法包括关于如何改正这个问题的信息。

使用反射

Java Reflection API能使Java代码不仅发现与运行时装入的类有关的信息，而且实例化和操纵对象。本章中所讨论的许多编程技巧依赖于反射，而本节将考虑反射的一些正反两方面的理由。

许多设计模式都能利用反射来得到最充分的表达。例如，如果类是Java组件，没有必要把这些类的名称硬编码到一个Factory（工厂）中；而且，这些类可以通过反射来实例化和配置。只有类的名称——比如一个接口的不同实现才需要在配置数据中进行提供。

在反射的使用上，Java开发人员的意见似乎各不相同。这是一件憾事，因为反射是核心API的一个重要部分，并且构成了许多技术的基础，比如Java组件、对象串行化（对J2EE至关重要）和JSP。许多J2EE服务器，比如JBoss和Orion都通过消除对容器生成的桩基和骨架的需要，使用反射（通过Java 1.3动态代理）来简化J2EE开发。这意味着对一个EJB的每次调用都可能会涉及到反射，而无论我们知不知道反射。反射是开发通用解决方案的一个强有力工具。

如果得到正确使用，反射能够使我们编写较少的代码。使用了反射的代码也能够通过使它自身保持最新来最小化维护工作量。例如，请考虑核心Java库中对象串行化的实现。由于它使用了反射，所以在字段被添加或从对象中被删去时，没有必要更新串行化和反串行化代码。这种方法以很少的效率为代价，极大地减轻了使用串行化的开发人员身上的工作负荷，而且还消除了许多编程错误。

有两个误解都集中在关于反射的保留上：

- 使用反射的代码是慢的；
- 使用反射的代码是过分复杂的。

上述两个误解每个都基于一种实际情况，但也都过度简单化了。让我们来依次看一看这两个误解。

使用了反射的代码通常比使用正常的Java对象创建和方法调用的代码慢。但是，这在实践中几乎无关紧要，而且这一差距随着JVM的每一代正逐渐缩小。性能差别是微乎其微的，并且反射的系统开销通常远远低于被调用方法实际执行的那些操作所占用的时间。

反射的大多数最佳使用根本没有性能影响。例如，系统启动时实例化和配置对象花费多长时间多半是无关紧要的。正如我们在第15章中将要看到的，大部分优化是不必要的。妨碍我们选择上佳设计选择的多余优化是绝对有害的。同样，处理一个Web请求时使用反射来填充一个Java组件（Struts和其他大多数Web应用框架所采取的方式）所增加的系统开销将是不可检测的。

如果不考虑性能在某一种特定情况中是否重要，反射距离对性能造成灾难性影响的可能也比许多开发人员想象中的要小得多，就像我们在第15章中将要看到的那样。事实上，在某些情况下，比如使用它来代替一个长的IF/ELSE语句链，反射实际上将改进性能。

Reflection API的直接使用是相当困难的。异常处理尤其不方便。不过，类似的限制也适用于许多重要的Java API，比如JDBC。解决方法是避免直接使用那些API（通过在适当的抽象级别上使用一个助手类层），但不避开它们所提供的功能度。如果我们通过一个适当的抽象层来使用反射，使用反射实际上将简化应用代码。

如果使用得当，反射不会降低性能。正确地使用反射实际上应该改进代码可维护性。反射的直接使用应该仅局限于基础结构类，不应该被分散到应用对象中。

反射的习惯用法

下列这些习惯用法举例说明了反射的正确使用。

反射与开关语句

IF/ELSE语句链和大开关语句应该使任何致力于OO设计原理的开发人员感到警觉。反射提供了避开它们的两种好方式：

- 使用条件来确定一个类名，然后使用反射来实例化该类并使用它（假设该类实现一个已知接口）。
- 使用条件来确定一个方法名，然后使用反射来调用它。

下面位我们来实际看一看第二种方法。

请考虑从java.beans.VetoableChangeListener接口中摘录的下列代码段。一个被接收到的PropertyChangeEvent含有相关属性的名称。很显然，这个实现将执行一个IF/ELSE语句链来确认该验证方法在该类中进行调用（vetoableChange()方法将变得巨大，如果所有验证规则都被联机包含）：

```
public void vetoableChange(PropertyChangeEvent e) throws PropertyVetoException {
    if (e.getPropertyName().equals("email")) {
        String email = (String) e.getNewValue();
        validateEmail(email, e);
    }
    ...
} else if (e.getPropertyName().equals("age")) {
    int age = ((Integer) e.getNewValue()).intValue();
    validateAge(age, e);
}
else if (e.getPropertyName().equals("surname")) {
    String surname = (String) e.getNewValue();
    validateForename(surname, e);
}
else if (e.getPropertyName().equals("forename")) {
    String forename = (String) e.getNewValue();
    validateForename(forename, e);
}
```

按照每个bean属性4行代码的速度，添加另外10个bean属性将增加40行代码到这个方法上。这个IF/ELSE链将在我们每次添加或删除bean属性时都需要更新。

请考虑下列替代方案。单个验证器现在扩展AbstractVetoableChangeListener——一个提供vetoableChange()方法的一种最终实现的抽象超类。AbstractVetoableChangeListener的构造器检查由符合一个验证签名的子类所添加的方法：

```
void validate<bean property name>(<new value>, PropertyChangeEvent)
    throws PropertyVetoException
```

该构造器是代码的最复杂部分。它查看符合该验证签名的那个类中所声明的所有方法。当它找到一个有效的验证器方法时，就把这个方法放入一个由该属性名为键的散列表validationMethodHash——就像该验证器方法的名称所暗示的那样。

```

public AbstractVetoableChangeListener() throws SecurityException {
    Method[] methods = getClass().getMethods();
    for (int i = 0; i < methods.length; i++) {
        if (methods[i].getName().startsWith(VALIDATE_METHOD_PREFIX) &&
            methods[i].getParameterTypes().length == 2 &&
            PropertyChangeEvent.class.isAssignableFrom(methods[i].
                getParameterTypes()[1])) {
            // We've found a potential validator
            Class[] exceptions = methods[i].getExceptionTypes();

            // We don't care about the return type, but we must ensure that
            // the method throws only one checked exception, PropertyVetoException
            if (exceptions.length == 1 &&
                PropertyVetoException.class.isAssignableFrom(exceptions[0])) {

                // We have a valid validator method
                // Ensure it's accessible (for example, it might be a method on an
                // inner class)
                methods[i].setAccessible(true);
                String propertyName = Introspector.decapitalize(methods[i].getName().
                    substring(VALIDATE_METHOD_PREFIX.length()));

                validationMethodHash.put(propertyName, methods[i]);
                System.out.println(methods[i] + " is validator for property " +
                    propertyName);
            }
        }
    }
}

```

vetoableChange()方法的实现为每个被修改的属性都做一次散列表查找，以找出相关的验证器方法，然后在找到一个验证器方法时就调用它。

```

public final void vetoableChange(PropertyChangeEvent e)
    throws PropertyVetoException {
    Method m = (Method) validationMethodHash.get(e.getPropertyName());
    if (m != null) {
        try {
            Object val = e.getNewValue();
            m.invoke(this, new Object[] { val, e });
        } catch (IllegalAccessException ex) {
            System.out.println("WARNING: can't validate. " +
                "Validation method '" + m + "' isn't accessible");
        } catch (InvocationTargetException ex) {
            // We don't need to catch runtime exceptions
            if (ex.getTargetException() instanceof RuntimeException)
                throw (RuntimeException) ex.getTargetException();
            // Must be a PropertyVetoException if it's a checked exception
            PropertyVetoException pex = (PropertyVetoException)
                ex.getTargetException();
            throw pex;
        }
    }
}

```

要想了解这个类的完整程序清单，或者实际使用它，请去配书下载的/framework/src目录，并参见该目录下的com.interface21.bean.AbstractVetoableChangeListener类。

现在，子类只需要和第一个例子中一样使用相同名称来实现验证器方法即可。不同的是，一个子类的逻辑在一个验证方法被添加或删除时将自动得到更新。另外请注意，我们已经使用反射来自动转换参数类型到验证方法。很显然，这是一个程序设计错误，比如，如果 validateAge() 方法期望一个 String 而非一个 int。这将在运行时在一个栈跟踪中被显示出来。明显的错误造成不了多大的危险。大多数严重问题源自于细微的错误，这些问题不会在应用每次运行时都出现，也不会导致明显的栈跟踪。

有趣的是，反射方法平均起来实际上比 if/else 方法快，如果有许多 bean 属性。串比较的速度慢，而反射方法使用一个单散列表查找来寻找要调用的验证方法。

可以肯定的是，AbstractVetoableChangeListener 类在概念上比 if/else 块复杂。但是，这是框架代码。它将被调试一次，并由一个综合性的测试案例集来验证。重要的是，应用代码——各验证器类由于反射的使用而变得更简单。另外，AbstractVetoableChangeListener 类对任何已扎实地掌握了 Java 反射的人来说仍是容易阅读的。笔者所使用的这个类的完整版本——包括完整的 Javadoc 与实现注释以及记录语句共计 136 行。

反射是 Java 的一个核心特性，任何一名认真的 J2EE 开发人员都应该扎实地掌握 Reflection API。尽管反射的习惯用法（比如三元算子）乍看起来似乎令人莫名其妙，但它们同样是该语言的设计的一部分，并且能够轻松地阅读和理解它们是十分重要的。

反射与 Factory 设计模式

笔者很少以其最简单的形式使用 Factory 设计模式，因为这种形式需要工厂所创建的所有类对该工厂的实现是已知的。这严重地限制了可扩展性：工厂对象不能创建对象（即使是实现一个已知接口的对象），除非该工厂对象知道它们的具体类。

下列这个方法（第 11 章中将讨论的那种“bean 工厂”方法的一个简化版本）说明了一种更灵活的方法，这种方法无需任何代码修改即是可扩展的。它的基础是使用反射来按名称实例化类。类名称可以来自任何一个配置源：

```
public Object getObject(String classname, Class requiredType)
    throws FactoryException {

    try {
        Class clazz = Class.forName(classname);
        Object o = clazz.newInstance();
        if (!requiredType.isAssignableFrom(clazz))
            throw new FactoryException("Class '" + classname +
                "' not of required type " + requiredType);
        // Configure the object...
        return o;
    } catch (ClassNotFoundException ex) {
        throw new FactoryException("Couldn't load class '" + classname + "'", ex);
    } catch (IllegalAccessException ex) {
        throw new FactoryException("Couldn't construct class '" + classname +
            "' : is the no arg constructor public?", ex);
    }
}
```

```

    } catch (InstantiationException ex) {
        throw new FactoryException("Couldn't construct class '" + classname +
            "': does it have a no arg constructor?", ex);
    }
}

```

- 这个方法可以像下面这样来调用：

```

MyInterface mo = (MyInterface)
    beanFactory.getBean("com.mycompany.mypackage.MyImplementation",
    MyInterface.class);

```

和其他反射例子一样，这种方法把复杂性隐藏在一个框架类中。确实，无法保证这个代码工作：类名称可能是错误的，或者该类可能没有一个无参数构造器，进而妨碍了它被实例化。但是，这样的失败在运行时将会毫不困难地显示出来，尤其是在`getBean()`方法产生易懂的错误消息时（当使用反射来实现低级操作时，要非常当心以生成有帮助的错误消息）。将操作推送到运行时之后确实涉及到折衷（比如对类型转换的需要），但好处可能才是实质性的东西。

反射的这种使用可以与Java组件的使用充分结合起来。如果要被实例化的那些对象暴露了Java组件属性，将初始化信息保存在Java代码外面是很容易的。

这是一种非常有用的习惯用法。性能是不受影响的，因为初始化通常只用在应用启动的时候；装入与初始化之间的差别，比如说反射10个对象与使用`new`运算符创建这10个对象并直接初始化它们之间的差别是不可检测的。另一方面，从真正灵活的设计观点来看，好处可能是许多的。一旦我们拥有了那些对象，就可以在不用进一步使用反射的情况下调用它们。

在使用反射来按名称装入类并在Java代码外部设置它们的属性与声明性配置的J2EE原理之间，存在一个特别大的协同作用。例如，小服务程序、筛选器和Web应用监听器从`web.xml`部署描述符内所指定的完全限定类名中被实例化。虽然它们不是`bean`属性，但`ServletConfig`初始化参数在同一个部署描述符内的XML段中被设置，进而允许小服务程序在运行时的行为能在不必修改它们的代码的情况下被更改。

使用反射是参数化Java代码的最佳方式之一。使用反射来选择动态地初始化和配置对象能使我们发挥出使用接口的松耦合的全部威力。反射的这种使用与声明性配置的J2EE原理是相容的。

Java 1.3动态代理

Java 1.3引进了动态代理（dynamic proxy）：能够在运行时实现接口但在编译时不用声明其实现接口的特殊类。

动态代理不能用来代理一个类（但能用来代理一个接口）。不过，这在我们使用基于接口的设计时不是问题。动态代理由许多应用服务器内部地使用，一般是为了避免对生成和编译桩基和骨架的需要。

动态代理通常用来截获对一个实际实现特有接口的委托的调用。这样的截获对处理资

源的获得与释放、添加附加的记录以及收集性能信息（尤其是关于分布式J2EE应用中远程调用的信息）会非常有用。当然，将有一定的性能开销，但开销的影响将随着委托实际做什么而变化。动态代理的一种有效使用是抽象引用EJB的复杂性。我们将会在第11章中见到这方面的-一个例子。

包含在本书示例应用的基础结构代码中的com.interface21.beans.DynamicProxy类就是一个通用的动态代理，这个动态代理处理相关接口的一个真实实现，其中该接口被设计成由添加自定义行为的动态代理来子类化。

动态代理可以用来实现标准Java中的Aspect Oriented Programming（AOP，面向形态的程序设计）概念。AOP是一个以系统的横切形态为基础的新出现范例，其中系统的横切基于关系的分离。例如，上面刚刚提过的记录能力的增加就是一个解决某一中心位置中的记录问题的横切。AOP是否将产生像OOP的好处那样的任何东西仍需拭目以待，但它将至少发展成补充OOP是有可能的。

要想了解关于AOP的详细信息，请参见下列站点：

- <http://aosd.net/>。AOP主页。
- <http://aspectj.org/>。AspectJ主页——AspectJ是一个支持AOP且扩充Java的扩展。

要想了解关于动态代理的详细信息，请参见你的JDK所携带的反射指南。

警告：在进行了反射的一种灵巧使用之后，笔者有一种不安全的感觉。过分灵巧降低了可维护性。虽然笔者坚定地认为如果使用得当，反射是非常有益的，但如果一种更简单的方法可能同样管用，就不要使用反射。

使用Java组件来实现灵活性

在可能的地方，应用对象（粒度非常精细的对象除外）应该是Java组件。这最大限度地提高了配置灵活性（正如我们在前面已经见过的），因为bean允许属性在运行时易发现和易操纵。使用Java组件的趋势有一点下降，因为没有必要实现一个特殊接口来使一个对象成为一个bean。

当使用bean时，请考虑下列标准bean工具能否用来实现功能度：

- PropertyEditor
- PropertyChangeListener
- VetoableChangeListener
- Introspector

将对象设计成Java组件（JavaBean）有许多好处。最重要的是，它使开发人员能够使用Java外面的配置数据来轻松地实例化和配置对象。

感谢笔者在FT.com的同事Gary Waston，谢谢他使笔者相信了Java组件的许多好处。

使用应用注册表来避免单元素集的激增

Singleton设计模式有非常广泛的用处，但很明显，该实现可能很危险。实现一个单元素

集的明显方式是，Java要使用一个含有该单元素集实例的静态实例变量，一个返回该单元素集实例的公用静态方法，并提供一个防止实例化的私用构造器。

```
public class MySingleton {  
    /** Singleton instance */  
    private static MySingleton instance;  
  
    // Static block to instantiate the singleton in a threadsafe way  
    static {  
        instance = new MySingleton();  
    } // static initializer  
  
    /** Enforces singleton method. Returns the instance of this object.  
     * @throws DataImportationException if there was an internal error  
     * creating the singleton  
     * @return the singleton instance of this class  
     */  
    public static MySingleton getInstance() {  
        return instance;  
    }  
  
    /** Private constructor to enforce singleton design pattern.  
     */  
    private MySingleton() {  
        ...  
    }  
  
    // Business methods on instance
```

请注意在该类被装入时，一个初始化单元素集实例的静态初始化器的使用。这防止了可能的竞争现象，如果单元素集在getInstance()方法中被初始化，并且是空值（导致错误的一个常见原因）。让静态初始化器捕捉由单元素集的构造器所抛出的任何异常也是有可能的，并且它们能够在getInstance()方法中被重新抛出。

但是，这种常见用法会产生几个问题：

- 对单元素集类的依赖性被硬编码到许多其他类中。
- 单元素集必须处理它自己的配置。由于其他类被锁定在了单元素集的初始化过程外面，所以单元素集将负责任何必要的属性读取。
- 复杂的应用可以有许多单元素集。每个单元素集都可能会不同地处理它的配置装入，进而意味着根本没有用于配置的中心贮藏库。
- 单元素集是接口不友好的。这是一件非常糟糕的事情。使一个单元素集实现一个接口没有一点意义，因为根本没有办法防止该接口的其他实现的存在。一个单元素集的普通实现定义一个类而非一个接口中的一个类型。
- 单元素集不服从继承性，因为我们需要编程到一个具体类，而且Java不允许超越像getInstance()方法那样的静态方法。
- 在运行时一致地更新单元素集的状态是不可能的。任何更新都可能被随意地在各单元素集或工厂类中执行。没有办法刷新一个应用中的所有单元素集的状态。

一种稍微复杂的方法是使用一个工厂，该工厂可能会为该单元素集使用不同的实现类。但是，这只能解决上述部分问题。

大体说来，笔者不太喜欢静态变量。它们通过引进对一个具体类的依赖性破坏了OO。Singleton设计模式的正常实现就显示出了这个问题。

笔者看来，一种较好的解决方案是拥有一个能用来查找其他对象的对象。笔者把这个对象称做应用上下文（application context）对象，尽管笔者也曾经见到过把它叫做“注册表”或“应用工具箱”。应用中的任何一个对象只需要获得指向该上下文对象的单个实例的一个引用，即可检索任一应用对象的单个实例。对象通常按名称来检索。该上下文对象甚至不必是一个单元素集。例如，使用Servlet API把上下文对象放到一个Web应用的ServletContext中是有可能的；我们也可以把上下文对象绑定到JNDI中，并使用标准应用服务器功能来访问它。这几种方法不要求对上下文对象本身做代码修改，只要求对引导代码做一点点修改。

上下文对象本身将是通用的框架代码，在多个应用之间是可重用的。

这种方法的优点包括：

- 它对接口非常管用。需要那些“单元素集”的对象从不需要知道它们的实现类。
- 所有对象都是普通Java对象，并且可以正常使用继承性。没有静态变量。
- 配置在那些相关的类外面被处理，并且被框架代码整个地处理。上下文对象负责实例化和配置各单元素集。这意味着Java代码外面的配置（比如在一个XML文档甚至RDBMS表中）可以用做配置数据的源。个别对象的配置可以使用Java组件属性来进行。这样的配置可以包括以应用上下文所管理的对象为背景的对象图表的创建，同时那些相关对象又无需做除了暴露bean属性之外的任何事情。
- 上下文对象将实现一个接口。这个接口允许不同的实现从不同的源中获取配置，同时又不需要对被管理的应用对象中的代码做任何修改。
- 支持“单元素集”的动态状态修改是可能的。上下文可以被刷新，进而修改它所管理的那些对象的状态（当然，尽管存在需要考虑的线程安全问题）。
- 使用一个上下文对象带来了其他可能性。例如，上下文可以提供其他服务，比如实现Prototype设计模式来作为独立对象实例的一个工厂。由于许多应用对象有权使用它，所以上下文对象可以用做Observer设计模式中的一个事件出版者。
- 虽然Singleton设计模式不灵活，但我们可以选择拥有多个应用上下文对象，如果这有用的话（第11章中将要讨论的基础结构支持分级结构的上下文对象）。

下列这些代码段举例说明了这种方法的使用。

上下文对象本身将负责装入配置。上下文对象可能会注册它自己（例如向一个Web应用的ServletContext，或者向JNDI），或者一个独立的引导类可能会处理这项工作。需要使用“单元素集”的对象必须在上下文对象中查找。例如：

```
ApplicationContext application = (ApplicationContext) ...;
servletContext.getAttribute("com.mycompany.context.ApplicationContext");
```

ApplicationContext实例可以用来获取任何“单元素集”：

```
MySingleton mySingleton = (MySingleton) ...
applicationContext.getSingleInstance("mysingleton");
```

在第11章中，我们将了解如何实现这个比Singleton设计模式更好的替代方案。需要注意的是，它不仅限于管理“单元素集”：这是基础结构的宝贵部分，能够以许多种方式被使用。

为什么不使用JNDI（一个标准J2EE服务），反而使用附加的基础结构来达到这个结果？每个“单元素集”可以被捆绑到JNDI上下文中，进而允许运行在应用服务器中的其他构件来查找它们。

使用JNDI增加了复杂性（JNDI查找是冗长的），并且与上面所描述的应用上下文机制相比，其威力明显弱得多。例如，每个“单元素集”将依靠它自己来处理它的配置，因为JNDI只提供一种查找机制，没有提供使配置外部化的手段。另一个严重的缺陷是，这种方法将完全依赖于应用服务器服务，进而使应用服务器外部的测试变成不必要的困难。最后，把对象绑定到JNDI中将需要某些种类的引导服务，进而意味着我们可能需要实现这种应用上下文方法中的大部分代码，无论使用什么方法。通过使用一个应用上下文，我们可以选择把个别对象和JNDI绑定在一起，如果它证明是有用的。

应该避免单元素集的急剧增加，其中每个单元素集带有一个静态getInstance()方法。使用一个工厂来返回每个单元素集更好，但仍欠灵活。相反，要使用一个返回每个类的一个单独实例的单独“应用上下文”对象或注册表。这种通用的应用上下文实现通常（但不是必定）基于反射的使用，而且应该负责配置它所管理的那些对象。该实现的优点是应用对象只需暴露用于配置的Bean属性，并且从不需要查找属性文件之类的配置源。

再加工

根据Martin Fowler在由Addison Wesley出版的“Refactoring: Improving the Design of Existing Code”一书（ISBN 0-201485-6-72）中所说，再加工（Refactoring）是“以这样一种方式修改一个软件系统的过程：这种方式不改变代码的外部行为，而是改进它的内部结构。它是整理这样一类代码的一种严格的方法：这类代码最大限度地减少引入隐含错误的机会”。要想详细了解关于再加工的详细信息和资源，请参见<http://www.refactoring.com>站点。

Fowler所描述的大多数再加工技巧对好的开发人员来说都是根深蒂固的习惯。不过，该讨论是有用的，而且Fowler的命名得到了广泛的采用（例如，Eclipse IDE就在菜单上使用了这些名称）。

应该准备再加工来消除代码重复并保证一个系统每时每刻都将得到充分实现。

使用一个支持再加工的IDE是有帮助的。Eclipse在这方面特别有帮助。

笔者认为，再加工可以被扩展到功能性代码之外。例如，我们应该不断地在下列这些领域寻求改进：

- **错误消息**

一个带有一条令人迷惑的错误消息的失败暗示一次改进该错误消息的机会。

- **日志记录**

在代码维护期间，我们可以细化日志记录来帮助调试。我们将在下文讨论日志记录。

- **文档编制**

如果一个隐错是由于对一个特定对象或方法做什么的不了解而导致的，文档编制应该被改进。

编程标准

J2EE项目往往都是大项目。大项目就需要联合作业，而且联合作业依赖于一致的程序设计惯例。我们知道，花费在软件维护上的工作量比花费在最初开发上的要大，因此保证应用的持续工作是至关重要的。这使好的Java编程标准——以及可靠的OO设计原理的实践——在整个J2EE项目期间变得至关重要。如果我们选择使用XP，编程标准就特别重要。集体的代码所有权只有当所有代码都被编写达到相同的标准时才能管用，而且一个团队在风格方面没有明显的差异。

为什么关于Java编程标准的一节（尽管其中强调J2EE）会被安排在一本关于J2EE的书中呢？因为存在这样一种危险：在J2EE技术的细节方面迷失方向并忽略好的程序设计习惯。这种危险已由许多J2EE示例应用所证明，因为这些应用含有草率的代码。

Sun在这方面是严重的犯规者。例如，Smart Ticket Demo版本1.1实际上根本没有包含注释，使用了诸如u、p、zc和cc之类的无意义方法参数名，以及含有诸如在出现异常时始终没有正确地关闭JDBC连接之类的严重编程错误。如果代码不是好得足以加入到生产应用中，它肯定也不是好得足以用做一个例子。

也许，这类应用的创作者认为，省略这样的“细化”澄清了他们试图要阐明的那些体系结构模式。这是一个错误。J2EE通常用于将被草率的习惯摧毁的大项目。此外，使代码达到生产标准可能会暴露出其原幼稚实现中的不足。

和设计原理一样，这是一个巨大的领域，因此下面的讨论远远不够全面。不过，该讨论试图解决笔者已经发现的、在实践中具有特殊重要性的问题。同样，必然存在大量的不同见解，而该讨论则以笔者的见解和实际经验为基础。

从标准开始

千万不要发明你自己的编程标准，或者引入你曾经使用过的其他语言中的那些编程标准。Java是一种相当简单的语言，因而只提供了一种做许多事情的方式。相反，Java的前辈C++通常提供几种方式。部分由于这个原因，开发人员用Java编写程序的方式方面还有一个较大幅度的标准化，而且这种方式应该得到尊重。

例如，你可能熟悉“Hungarian Notation（匈牙利表示法）”或Smalltalk命名标准。但是，Hungarian Notation存在是为了解决Java中不存在的问题（Windows API中的类型激增）。越来越多的Java开发人员没有使用过其他语言，而且会被引入命名标准的代码所难倒。

要首先从Sun的Java编程标准开始（可以从<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>站点中获取）。如果愿意，可以引进改进和变体，但不要偏离常用的Java习惯太远。如果你的组织已经有编程标准，在这些标准的范围内工作，除非它们是十分不标准的或有疑问的。在这种情况下，不要忽视它们；发动关于如何改进它们的大讨论。

下面是值得一看的其他一些编程标准：

- <http://g.oswego.edu/dl/html/javaCodingStd.html>

由Doug Lea开发的Java编程标准，他是“Concurrent Programming in Java”一书（现在有点过时）的作者。

- <http://www.chimu.com/publications/javaStandards/part0003.html#E11E4>
Chimu Inc编程标准（部分基于Lea的编程标准）。
- <http://www.ambysoft.com/javaCodingStandards.html>
Scott Ambler的编程约定。笔者曾经见过的一个冗长文档，其中有一些最佳的讨论。
Ambler撰写过许多关于Java和OO设计的图书，致力于对编程标准系列中的设计方向的讨论，并且他在这方面的讨论比那些Sun约定要深入得多。

但是，由坚持标准Java习惯所产生的一个常见问题值得一提。这个问题关系到使用实例变量名作为一个参数和解决使用this的多义性的约定。这个关键字通常用在属性设置器中。例如：

```
private String name;  
  
public void setName(String name) {  
    this.name = name;  
}
```

从积极的方面看，this是一个常用的Java惯用语，因此它得到广泛理解。从消极的方面看，忘记使用this来区分两个同名变量是十分容易的（该参数将屏蔽实例变量）。这个方法的下列形式将会编译通过：

```
public void setName(String name) {  
    name = name;  
}
```

下列形式也将会编译通过（方法参数的名称中含有一个打字错误）：

```
public void setName(String nme) {  
    name = name;  
}
```

在这两种情况中（假设实例变量名以空值开始），神秘的空值指针异常将会在运行时出现。在第一个错误的版本中，我们给该方法参数分配它自己，从而什么目的也没有达到。在第二个错误的版本中，我们给该实例变量分配了它自己，从而使它为空值。

笔者不提倡使用以m_来前缀实例或成员变量（比如m_name）的C++约定，因为该约定很糟糕，并且与其他Java约定不一致（下划线通常只用在Java内的常数中）。但是，笔者建议以下列3种习惯来避免我们刚才所见过的那两个错误：

- 考虑给参数赋予一个有区别的名字，如果可能会出现多义性问题。在上述情况中，那个参数可以叫做newName。这正确地反映了该参数的用途，并且避免了我们刚见过的那个问题。
- 在访问实例变量时总是使用this，无论解决多义性是不是必不可少的。这具有使每个方法对实例数据的依赖性变得明显的优点。例如，这在考虑并发问题时是非常有用的。
- 遵守局部变量名应该相当短的约定，而实例变量名是冗长的。例如，i应该是一个局部变量，userInfo应该是一个实例变量。通常，实例变量名应该是以一个小写字母开

头的接口或类名（比如SystemUserInfo systemUserInfo），而局部变量名应该在当前上下文中传达它们的含义（比如SystemUserInfo newUser）。

要想了解此领域内反对标准Java约定的理由，请参见<http://www.beust.com/cedric/naming/index.html>，这些反对理由来自Cedric Beust—WebLogic EJB容器的领先开发者。

一致的文件组织很重要，因为它能使参与一个项目的所有开发人员快速领会一个类的结构。笔者使用下列约定，它们补充了Sun的约定：

- 按照功能，而不是按照可访问性来组织方法。例如，不是把公用方法放在私用方法的前面，而是把一个私用方法和使用它的那些公用方法放在一个类的同一个段内。
- 界定代码段。例如，笔者界定下列各代码段（按次序）：
 - 任何静态变量和方法。请注意，main()方法不应该有问题，因为一个做任何事情的类不应该包括一个main()方法，而且代码应该使用JUnit被测试过。
 - 实例变量。有些开发人员喜欢把每个bean属性持有者与相关的获得器和设置器方法分为一个组，但笔者认为使所有实例变量在一起最理想。
 - 构造器。
 - 接口的实现（每个接口都有它自己的段），连同支持它们的私用实现方法。
 - 由该类暴露出来但不属于任一已实现接口的公用方法。
 - 抽象保护方法。
 - 打算供子类使用的保护方法。
 - 与前面的任何一组都无关的实现方法。

笔者像下面这样使用段界定符：

```
//-----  
// Implementation of interface MyInterface  
//-----
```

要想了解这里所描述的布局和约定的使用示例，请转到本书的配书下载，并参考framework/src目录中的类。一个很好的例子就是其中的com.interface21.beans.factory.support.AbstractBeanFactory类。

如果读者需要使自己相信确实需要编程标准，并且又有一定的闲暇时间，请访问<http://www.mindprod.com/unmain.html>站点。

责任的分配

每个类都应该有一个明确的责任。不合适的代码应该被再加工，通常被再加工成一个助手类（内部类常常是一种在Java中做这项工作的好方法）。如果某一个不同概念级别的代码将被相关对象重用，它可能需要被提升成一个超类。但是，正如我们已经见过的，针对一个助手的委托对具体继承性常常是最理想的。

应用这项规则一般防止了类大小爆裂。包括量大的Javadoc和内部注释在内，任何一个长于500行代码的类都是一个再加工的候选者，因为它可能有太大的责任。这种再加工也提高了灵活性。如果该助手类的功能度在不同情况下需要被不同地实现，可以用一个接口把原类与助手分开（在Strategy设计模式中）。

也应该给方法运用同样的原理：

一个方法只应该有一项明确的责任，而且所有操作都应该在同一个抽象级别上。

如果情况不是这样，应该再加工该方法。在实践中，笔者发现这防止了方法变得太长。

笔者没有为方法长度使用任何严格的规则。笔者的舒适阈值主要由笔者可在屏幕上立刻看到的代码长度来确定（假如笔者平常专门使用屏幕的一部分来查看代码，而且有时在便携电脑上工作）。这个阈值往往是30到40行（包括内部实现注释，但不包括Javadoc方法注释）。笔者认为，超过这个阈值的方法通常就需要再加工。即使一个方法内由几项任务构成的一个单元只被调用一次，把这几项任务提取到一个私用方法仍是一个不错的主意。通过给这样的方法赋予适当的名字（短方法名就不值得了），代码就会变得更容易阅读，而且是意义自明的。

避免代码重复

这似乎是非常明显的一点，但代码重复是致命的。

摘自Java Pet Store 1.3中的一个简单例子充分说明了这一点。有一个EJB实现含有下列两个方法：

```
public void ejbCreate() {  
  
    try {  
        dao = CatalogDAOFactory.getDAO();  
    } catch (CatalogDAOSysException se) {  
        Debug.println("Exception getting dao " + se);  
        throw new EJBException(se.getMessage());  
    }  
}
```

和

```
public void ejbActivate() {  
  
    try {  
        dao = CatalogDAOFactory.getDAO();  
    } catch (CatalogDAOSysException se) {  
        throw new EJBException(se.getMessage());  
    }  
}
```

这似乎是微不足道的，但这样的代码重复会导致严重的问题，比如：

- 太多的代码。在这种情况下，再加工只节省一行，但在许多情况下，节省量将会比较大。
- 令阅读者对意图感到迷惑。由于代码重复是不合逻辑和容易避免的，所以阅读者可能会对该开发人员产生怀疑，并认为这两个代码段是不一致的，因而浪费时间比较它们。
- 不一致的实现。即使在这个微不足道的例子中，一个方法记录该异常，而另一个方法则没有。

- 持续的为更新两段代码来修改单个操作究竟是什么的需要。

下列再加上更简单，并且更容易维护：

```
public void ejbCreate() {
    initializeDAO();
}

public void ejbActivate() {
    initializeDAO();
}

private void initializeDAO() {

    try {
        dao = CatalogDAOFactory.getDAO();
    } catch (CatalogDAOSysException se) {
        Debug.println("Exception getting dao " + se);
        throw new EJBException(se.getMessage());
    }
}
```

请注意，我们已经合并了该代码；我们可以做一个单行修改来改进它，以便使用EJB 2.0中的新EJBException构造器，该构造器接受一条消息，连同一个嵌套异常。我们还将包括关于我们正试图做什么的信息：

```
: throw new EJBException("Error loading data access object: " + se.getMessage(), se);
```

EJB 1.1允许EJBException含有嵌套异常，但把一个EJBException构造成同时带有一条消息和一个嵌套异常是不可能的，从而迫使我们选择是包括嵌套异常，还是包括一条关于该EJB在捕捉到该异常时正在做什么的有意义的消息。

避免字面常数

除了众所周知的著名值0之外，空值（Null）和“”（空串）在Java类内不使用字面常数。

请考虑下列例子。一个类含有下列代码作为处理一个订单的一部分：

```
if (balance > 10000) {
    throw new SpendingLimitExceededExeption(balance, 10000);
}
```

令人遗憾的是，我们经常看到这种代码。但是，它会导致许多问题：

- 该代码不是意义自明的。阅读者被迫阅读该代码来猜测10000的含义。
- 该代码容易出错。阅读者将被迫比较不同的字面值来确保它们是相同的，并且错误地键入多个字面值之一是很容易的。
- 修改那个逻辑常数将需要多处代码修改。

比较好的做法是使用一个常数。在Java中，这意味着一个静态的终结实例变量。例如：

```
private static final int SPENDING_LIMIT = 10000;

if (balance > SPENDING_LIMIT) {
    throw new SpendingLimitExceededExeption(balance, SPENDING_LIMIT);
}
```

这个版本更容易让人读懂，并且不容易出错。在许多情况下，它已足够好。但是，在某些情况下，它仍是有疑问的。如果花钱限制不总是相同，怎么办呢？今天的常数可能是明天的变量。下列替代版本能使我们有更多的控制权：

```
private static final int DEFAULT_SPENDING_LIMIT = 10000;

protected int spendingLimit() {
    return DEFAULT_SPENDING_LIMIT;
}

if (balance > spendingLimit()) {
    throw new SpendingLimitExceededExeception(balance, spendingLimit());
}
```

以稍多一点的代码为代价，我们现在可以在运行时计算花钱限制（如果必要）。另外，一个子类可以覆盖保护方法spendingLimit()。相反，覆盖一个静态变量是不可能的。一个子类甚至可能会暴露一个bean属性，从而使花钱限制能够通过一个配置管理类在Java代码外部被设置（请参见较前面的“使用应用注册表来避免单元素集的激增”一节）。spendingLimit()方法应不应该使用的是另外一个问题。除非知道其他类需要使用这个方法，否则使它保持为受保护方法可能会更好。

笔者推荐使用表4.2中的标准来确定如何设计一个常数。

表4.2

需求	示例	建议
实际上是应用代码一部分的串常数	只被使用一次并且将不随数据库而变化的简单SQL SELECT语句。 只被使用一次的JDO查询	这是在使用一个命名常数或方法值来取代一个字面值串没有一点好处时的总规则的一个罕见例外。在这种情况下，让串显示在应用内使用它的地方会更有意义，因为它实际上 是应用代码的一部分
绝不会变化的常数	在所有应用服务器中都相同的JNDI名——比如一个EJB的名字	使用一个静态的终结变量。共享常数可以在一个接口中被声明，而接口可以由多个类来实现，以简化语法
在编译时可能会变化的常数	在应用服务器之间可能需要变化的JNDI名——比如 TransactionManager的名字	使用子类可以覆盖或者可以返回一个bean属性以便允许外部配置的一个保护方法
在运行时可能会变化的常数	花钱限制	使用一个受保护方法
易受国际化影响的常数	在不同地区可能需要有所不同的错误消息或其他串	使用一个受保护方法或一个ResourceBundle查找。需要注意的是，一个受保护方法可以返回这样一个值：这个值是从一个ResourceBundle查找中获得的（可能在该类外部）

可见度和作用范围

实例变量和方法的可见度是编程标准与OO设计原理之间分界线方面的重要问题之一。由于字段和方法可见度会对可维护性产生很大的影响，所以在在这方面应用一致的标准是非常重要的。

笔者推荐下面这条通用规则：

变量和方法应该拥有尽可能小的可见度（包括私用的、封装的、保护的和公用的变量和方法）。变量应该被尽可能局部地声明。

让我们来依次看一看部分关键问题。

公用实例变量

公用实例变量的使用是不可避免的，除了极少数特殊情况之外。这通常反映了不良的设计或程序员的懒散。如果任何一个调用者不使用对象的方法都能操纵该对象的状态，那么封装会受到致命的伤害。我们决不能维持关于对象状态的任何不变量。

作为Value Object（值对象）J2EE模式中的一个可接受策略，“Core J2EE Patterns”一书建议使用公用实例变量（值对象是含有数据而非行为的可串行化参数，这些数据将在远程方法调用中在JVM之间被交换）。笔者认为，这个策略只有当那些变量被变成最终的时才是可接受的（以防止它们的值在对象构造之后被修改，以及避免调用者可能会操纵对象状态目录）。但是，对于公用实例变量在值对象中的任何使用来说，有许多应该加以考虑的严重缺点，笔者认为应该消除这些缺点。例如：

- 如果变量不被变成最终的，值对象中的数据就无法受到保护以免于修改。请考虑这样一种常见情况：值对象一旦在一个远程调用中被检索出来，就被高速缓存在客户端上。修改值对象状态的一个构件会影响使用这个相同值对象的所有构件。Java给我们提供了避免这类场景的工具（比如伴随获得器方法的私用变量）；我们应该使用它们。
- 如果变量被变成最终的，所有值对象都必须在值对象构造器中被提供，而这可能会使值对象更难以创建。
- 公用实例变量的使用不够灵活。一旦调用者依赖于公用实例变量，它们就依赖于值对象的数据结构，而不只依赖于一个公用接口。例如，我们不能使用第15章中将要讨论的用于优化值对象串行化的某些技巧，因为它们依赖于在不改变公用方法签名的情况下转换到多种有效的存储类型。虽然我们在必要时不影响调用者就能自由地修改公用方法的实现，但对值对象实现的修改将要求使用实例变量的所有调用者首先移植成使用访问器方法，而有证据表明这种移植可能是非常耗时的。
- 公用实例变量的使用使我们束缚于编程到具体类，而不是编程到接口。
- 实例变量访问不能被截听。我们没有办法知道什么数据正被访问。

使用公用实例变量的值对象实际上是一个struct（结构）的一种特殊情况，所谓结构指的是不具有任何行为的一个变量组。和C++（C的一个超集）不同，Java没有struct类型。但

是，在Java中定义结构是很容易的，它们就像只含有公用实例变量的对象。由于结构欠缺灵活性，所以它们只适合局部使用，比如作为私用和保护内部类。结构可能会用来从方法中返回多个值。例如，假定Java不支持基本类型的按引用调用。

笔者并不把这种隐蔽的结构看做对OO原理的严重违背。但是，结构通常需要构造器，进而使它们更接近于真正的对象。由于IDE使生成实例变量的获得器和设置器方法变得很容易，所以在开发期间使用公用实例变量会节省少量的时间。在现代JVM中，任何性能收益都将是极微小的，除了一些极少见情况之外。笔者发现，结构通常通过再加工被提升为真正的对象，因而首先就避免使用它们才是更明智的举动。

非常合法地使用公用实例变量（少见）的优点是如此少，误用公用实例变量的后果又是如此严重，因此笔者建议完全取缔公用实例变量的使用。

受保护和封装受保护的实例变量

实例变量应该是私用的，只有极少数情况例外。如果必要，通过受保护的访问器方法来暴露这样的变量，以便支持子类。

笔者对主张使实例变量成为受保护变量，以便最大限度地增加子类的自由度的编程标准（比如Doug Lea的编程标准）持坚决反对的态度。这对具体继承性来说是一种有疑问的方法，并意味着超类的完整性和不变性会被含有隐错的子类代码伤害。在实践中，笔者发现子类化工作得完全像一个“黑箱”操作。

同为了让子类操纵而暴露实例变量相比，允许类行为被修改有许多更好的方式，比如使用Template Method和Strategy设计模式（上文已讨论过），以及必要时提供受保护方法来允许子类状态的受控操纵。允许子类访问受保护实例变量将会产生一个继承性分级结构中的类之间的紧耦合，从而使修改它里面的类的实现得很困难。

Scott Ambler强烈主张，所有实例变量都应该是私用的，而且更进一步主张，“只有获准直接处理一个字段的那些成员函数才是访问器成员函数本身（也就是说，甚至连该声明类内部的方法都应该使用获得器和设置器方法，而不是直接访问私用实例变量）”。

笔者觉得，一个受保护实例变量只有当它是最终的时才是可接受的（比如子类不用初始化或修改即可使用的一个记录器）。这种方法的优点是避免了一个方法调用，因而提供了稍微简单一点的语法。但是，即使在这种情况下，仍有一些缺点。在不同的环境中返回不同的对象是不可能，而且子类不能覆盖一个变量，因为它们可以调用一个方法。

笔者很少见到有人为实例变量合法地使用Java的封装（默认）可见度。这有点像C++的friend机制：懒散程序员的酒肉朋友。

要避开受保护实例变量。它们通常体现了不良的设计：几乎总是有一个更好的解决方案。惟一的例外是一个实例变量能被变成最终实例变量的罕见情况。

方法可见度

虽然方法调用决不能造成和实例变量的直接操纵一样的危险，但尽可能大地降低方法可见度有许多好处。这也是降低类间耦合性的另一种方法。把使用一个类的那些类（甚至子

类)的需求与这个类的内部需求区分开是很重要的。这既可以防止这个类的内部状态遭到意外破坏，也可以简化开发人员处理这个类的任務(通过只给他们提供他们所需要的那些选择)。

尽可能地隐藏起方法。公用的、受封装保护的或受保护的方法越少，一个类就越简洁，它测试、使用、子类化和再加工起来也将会更容易。通常，只有一个类暴露出来的那些公用方法才将是它所实现的那个接口的方法和暴露Java组件属性的方法。

使一个类的实现方法变成受保护的，而不是变成私用的，以便允许它们被子类使用是一种常见的做法。这种做法是不妥当的。根据笔者的经验，继承性最好被当做-一个黑箱操作来处理，而不要当做-一个白箱操作来处理。如果类Dog扩展Animal，这应该意味着一个Dog能够用在一个Animal能够使用的地方，而不是意味着Dog类需要知道Animal的实现细节。

protected修饰符最好用于抽象方法(就像Template Method设计模式中那样)，或者用于子类所需要的只读助手方法。在这两种情况中，使方法变成受保护的而不是公用的确实有好处。

笔者发现自己很少需要使用受封装保护的(默认可见度)方法，尽管它们的缺陷没有受保护实例变量的那么严重。有时，暴露类状态的受封装保护的方法对测试案例是有帮助的。受封装保护的类常常更有用，进而使整个类能够隐藏在一个包内。

变量作用范围

变量应该尽可能近地被声明到它们得到使用的地方。范围内的变量越少，代码的阅读和调试就会越容易。在一个自动方法变量和(或)附加方法参数能够使用的地方使用一个实例变量是一个严重的错误。要使用C++/Java局部声明(在这类声明中，变量就在它们正要得到使用前被声明)，而不是使用C风格的声明(变量在方法的开始处被声明)。

内部类和接口

在Java中，内部类和接口可以用来避免名称空间污染(namespace pollution)。内部类常常是助手，并且可以用来确保外部类有一个一致的责任。

要了解静态与非静态内部类之间的差别。不用创建一个封闭类型的对象，静态内部类就可以被实例化；非静态内部类被链接到一个封闭类型的实例。对接口来说，没有任何差别，因为它们总是静态的。

当一个类需要一个在具体类中可能会变化但在类型中不变化的助手，并且这个助手对其他类没有任何好处可言时，一般才会使用内部接口(我们已经见过这方面的一个例子)。

匿名内部类(anonymous inner class)提供简单接口的便利实现，或提供只增加少量新行为的覆盖。它们的最常见使用是用于Swing GUI中的动作处理器，这种使用与J2EE应用的关系是有限的。不过，它们可以用在实现回调方法的时候(我们已经在前面讨论过)。

例如，我们可以利用一个匿名内部类实现一个JDBC回调接口，如下所示：

```

public void anonClass() {
    JdbcTemplate template = new JdbcTemplate(null);
    template.update(new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement(
            Connection conn) throws SQLException {
            PreparedStatement ps =
                conn.prepareStatement("DELETE FROM TAB WHERE ID=?");
            ps.setInt(1, 1);
            return ps;
        }
    });
}

```

匿名内部类的缺点是它们不鼓励代码重用，不能拥有接受参数的构造器，以及只在单个方法调用中才是可访问的。在上面的例子中，这些局限性都不是问题，因为那个匿名内部类不需要构造器参数，也不必返回数据。任何一个内部类（包括匿名内部类）都能访问超类实例变量，而超类实例变量提供了一种从封闭类中读取信息和更新封闭类的方法，以便克服这些局限性。就个人而论，笔者很少使用匿名内部类，除了在使用Swing的时候，因为笔者发现它们几乎总是被再加工成命名的内部类。

高级内部类（可由所有方法和其他可能的对象使用）与匿名内部类之间的一个中间点是定义在方法内的一个命名内部类。这不仅避免了污染这个类的名称空间，而且又允许使用一个普通构造器。不过，和匿名内部类一样，局部类可能会导致代码重复。定义在方法内的命名类具有两个优点：它们能够实现接受参数的构造器，以及能够被调用多次。在下面这个例子中，这个命名内部类不仅实现一个回调接口，而且还增加一个新的公用方法，我们将在这个公用方法的工作完成之后使用它来获得数据：

```

public void methodClass() {
    JdbcTemplate template = new JdbcTemplate(dataSource);
    class Counter implements RowCallbackHandler {
        private int count = 0;
        public void processRow(ResultSet rs) throws SQLException {
            count++;
        }
        public int getCount() {
            return count;
        }
    }
    Counter counter = new Counter();
    template.query("SELECT ID FROM MYTABLE", counter);
    int count = counter.getCount();
}

```

如果没有（适当地）利用封闭类中的一个实例变量来保存计数值，使用一个匿名内部类来实现上面这个例子是不可能的。

使用final关键字

final关键字可以用在几种情形中来得到好的结果。

方法覆盖和最终方法

有一个常见的误解：使方法变成最终的会减轻一个类的责任，因为这个误解过分约束了子类的实现。事实上，覆盖具体方法是实现扩展性的一种拙劣方法。

笔者建议使公用的和受保护的非抽象方法变成最终的。这可以帮助消除导致隐错的一个常见原因：子类污染其超类的状态。覆盖方法天生就是危险的。请考虑下列问题和疑问：

- 该子类应该调用该方法的超类版本吗？如果应该，该调用应该发生在什么点上？在该子类方法的开始处，还是在结束处？是否调用超类的方法只能通过阅读代码或依靠超类中的文档说明来确定。编译程序无法帮忙。这排除了黑箱继承性。如果该方法的超类形式没有被调用，或者在子类方法内的错误地点上被调用，超类的状态可能会遭到污染。
- 该超类为什么正在实现它还没有了解得足以代表所有子类来实现的一个方法？如果它能够提供一个有效的部分实现，它就应该让该操作中它还不知道的那些部分听从Template Method设计模式中的抽象保护方法；如果它的实现可能会被某些子类完全覆盖，最好是卸开继承性树，以便为那些共享相同行为的子类提供一个附加的超类（其中该方法是最终的）。
- 如果一个方法的一个子类覆盖实现做一件不同于超类实现所做的事情，该子类可能违反了Liskov Substitution Principle（Liskov替代原理）。Liskov替代原理由Barbara Liskov于1988（“Data Abstraction and Hierarchy”，SIGPLAN Notices, 23 May, 1988）规定，这项原理规定：一个子类应该始终可用而又不影响调用者，而不是代替它的超类。这项原理保护具体继承性的概念。例如，一个Dog对象在必须使用一个Animal对象的任何地方都应该是可用的。违反Liskov替代原理的子类对单元测试也是不友好的。一个不带有具体方法覆盖的类应该通过其超类的所有单元测试。

另一项OO原理——Open Closed Principle（开放式封闭原理）规定：一个对象对扩展应该是开放的，但对修改应该是封闭的。通过覆盖具体方法，我们实际上修改了一个对象，并且可以不再保证它的完整性。在给一个应用增加新功能时，遵守开放式封闭原理有助于减小隐错的可能性，因为新功能度在新代码中被增加，而不是通过修改现有代码，进而潜在地违反了该原理。

尤其在类将被许多不同子类覆盖的情况下，当方法不能是私用的时（例如，如果它们实现一个接口，由此而必须是公用的），使超类方法变成最终的将简化开发子类实现的程序员的工作。例如，大多数程序员将使用提供代码助手的IDE来创建子类；如果这些IDE展现出一个列表来列举出能够——或在抽象方法的情况下必须——被覆盖的那些非最终方法，这种做法是完全可行的。

使方法成为最终的将产生细微的性能收益，尽管这一点点性能收益可能太微不足道，以至于在大多数情况下不能成为一个考虑要素。

需要注意的是，与通过覆盖具体方法相比，扩展一个对象有许多更好的方法。例如，Strategy设计模式（前面已经讨论过）可以用来参数化该对象的部分行为：通过委托给一个接口。该接口的不同实现可以在运行时被提供，以便更改该对象的行为（但不损害完整性）。笔者曾经像这里所建议的那样在几个大项目中使用过最终方法，结果是实际消除了与超类状

态的污染有关的隐错，同时又没有对类可重用性造成任何不利影响。

最终方法常常和抽象保护方法联合使用。这方面的一种常见用法是笔者所谓的“链式初始化器（chaining initializers）”。请考虑一个假设的小服务程序超类AbstractServlet。假设这个便利超类的用途之一是初始化子类所需要的许多助手类。AbstractServlet超类在其Servlet API init()方法的实现中初始化这些助手类。

为了保护该超类的完整性，这个方法应该被变成最终的（否则，不用调用这个方法的AbstractServlet超类的实现，一个子类就可以覆盖init()方法，因而意味着超类状态将会得不到正确的初始化）。但是，子类可能需要实现它们自己的初始化（有别于超类的初始化）。答案是让该超类在init()的一个最终实现中调用一个链式方法，如下所示：

```
public final void init() {
    // init helpers
    // ...
    onInit();
}

protected abstract void onInit();
```

onInit()方法有时叫做一个挂钩方法（hook method）。这种情况下的一种变型是提供onInit()方法的一个空实现，而不是使它成为抽象的。这防止了不需要其自身初始化的子类被迫实现这个方法，但缺点是一个简单的打字错误就会导致该子类提供一个永远得不到调用的方法，比如通过把它称做oninit()。

这项技巧可以用在许多情况中，不仅仅用在初始化中。根据笔者的经验，它在框架中就特别重要，因为框架的类将经常被子类化，而且子类的开发人员应该没有理由操纵（或仔细检查）超类行为。

笔者建议，通常情况下应该把公用的或受保护的非抽象方法变成最终的，除非下列条件之一适用：

- 该方法的一个子类形式将不需要调用该方法的超类形式。这通常出现在这样的情况下：如果超类提供一个方法的一个简单的默认或空实现，以避免所有子类被迫提供只对少数子类有用的一个抽象方法的一个实现（和上述变型中一样）。
- 调用该方法的超类形式作为其子类形式的一部分是合乎逻辑的。覆盖一个Java对象的toString()方法是这方面的最常见例子。
- 挂钩方法的数量可能会在其他方面迂回曲折地失去控制。在这种情况下，我们必须利用实践经验来调整设计的严密性。超类的文档编写必须小心翼翼地说明子类方法应该在什么地方调用超类方法。

笔者在这方面的观点有点争议。不过，几个大项目中的经验已经使笔者相信编写帮助最小化出错可能性的代码有多大价值。这一立场已由著名计算机科学家（和快速分类的发明者）C.A.R. Hoare做了概述，如下所示：

“我最终相信需要设计编程表示法，以便最大限度地减少不能犯的错误的数量，如果犯了，就最大限度地减少能够在编译时可靠地检测出来的错误的数量”（1980 Turing Award Lecture）。

最终类

最终类的使用不如最终方法的使用那么频繁，因为它们是减少对象修改的一种更激烈方法。

“UML Reference Manual”一书（由Addison Wesley出版，ISBN 0-20130-998-X）竟然建议，只有抽象类才应该被子类化（由于我们在考虑最终方法时已讨论的那些原因）。不过，笔者觉得，如果最终方法使用得当，几乎就没有必要使类变成最终的来保存对象的完整性。

笔者往往只为必须被保证永远不变的对象使用最终类，比如含有从一个保险报价单中产生的数据的对象。

最终实例变量

笔者已经提过最终的受保护实例变量的使用。一个最终实例变量只可能被初始化一次，要么在它的声明中，要么在一个构造器中。最终实例变量是Java中定义常数的惟一方法，这也是最终实例变量的普通用法。不过，它们有时可以用来允许超类暴露受保护实例变量，同时又不允许子类操纵它们，也可以用来允许任何一个类暴露不能被操纵的公用实例变量。

Java语言行家还将会注意到，最终实例变量能够在一个类初始化器中被初始化：所谓类初始化器指的是出现在一个类中但在一个方法主体外面，并在一个对象被初始化时才得到计算的一个代码块。类初始化器的使用没有静态初始化器那么频繁，因为构造器通常是更可取的。

实现对诊断有用的toString()方法

让类去实现概述其状态的toString()方法是一个良好的习惯。这在生成大消息时会非常有用（我们将下文中讨论这方面的内容）。

例如，请考虑下列代码，它可能会用在表示用户的值对象中，并提供对象状态的一个简洁、易读的内像，而该内像在调试中是非常有用的。

```
public String toString() {
    StringBuffer sb = new StringBuffer(getClass().getName() + ": ");
    sb.append("pk=" + id + "; ");
    sb.append("surname='" + getSurname() + "' ");
    sb.append("forename='" + getForename() + "' ");
    sb.append("systemHashCode=" + System.identityHashCode());
    return sb.toString();
}
```

请注意，这段代码使用了一个StringBuffer——比并置串与+运算符更有效率。另外请注意，串的名字（forename）和姓氏（surname）值被括在了单引号中，这将使可能会引起意外行为的任何空格变得易于检测。另外还需要注意，状态串包括该对象的散列码，这对验证对象在运行时是否有区别会非常有用。本例使用System.identityHashCode()取代了对象的hashCode()方法，因为System.identityHashCode()方法返回默认的Object散列码，而大多数JVM将基于一个对象在内存中的位置，而不是基于该对象可能会实现的、这个方法的任一超越。

toString()值的另一种重要使用是显示一个接口的一种实现的类型与配置。

防守型的编程原则

NullPointerException是导致隐错的一个常见原因。由于NullPointerException没有携带帮助性的消息，所以它们引起的问题很难追踪。下面，让我们来看一看可以用来减小它们在运行时出现的可能性的几个编程标准。

正确处理空值

考虑一个对象是空值时将会发生什么是特别重要的。笔者为处理空值的可能性推荐下列指导原则：

- 用文字详细说明方法在空值参数上的行为。检查参数是否为空值常常是一个不错的主意。如果空值参数被认为暗示了错误的调用代码，并且一个方法可能会合法地抛出一个NullPointerException，用文字详细说明该行为是很重要的。
- 编写使用空值参数调用方法的测试案例，以验证用文字所说明的行为（它可能是任何行为）。
- 如果没有充分的证据，不要假设一个对象在某个特殊时刻决不可能是空值。这种假设会引起许多问题。

考虑对象比较的次序

下列两行代码在正常操作中将产生相同的结果：

```
if (myStringVariable.equals(MY_STRING_CONSTANT))  
if (MY_STRING_CONSTANT.equals(myStringVariable));
```

但是，第二种形式更可靠。如果myStringVariable是空值，结果会怎么样呢？如果不出现错，第二个条件将计算出假值，而第一行代码将抛出一个NullPointerException。在不太可能是空值的对象上调用equals()方法来执行对象比较通常是一个好习惯。如果另一个对象为空值是一个错误，则执行一次显式的空值检查，并抛出相应的异常（但不是NullPointerException）。

使用短路计算

有时，我们可以依靠Java的布尔表达式的短路计算来避免潜在的错误，比如含有空值对象。请考虑下列代码段：

```
if ((o != null) && (o.getValue() < 0))
```

这是安全的，即便对象o是空值。在这种情况下，第二个测试将得不到执行，因为该条件已经计算出假值。当然，这种惯用法只有当它体现了代码的意图时才能使用。如果o对象是空值，则可能需要做另外一件不同的事情（除了计算这个条件得出假值外）。但是，可以确信的是，我们将不需要一个NullPointerException。

另外一种可选方法是仅在一条外部if语句已经确定了该对象是非空值之后，在一条内部

if语句中执行第二个检查。但是，笔者不赞成这种方法，除非嵌套if语句有其他一些很好的理由（不过，常常有一些很充分的理由），因为语句嵌套会增加复杂性。

区分调试语句和错误消息中的空格

请考虑下列情景。一个Web应用因下列错误而失败：

```
Error in com.foo.bar.MagicServlet: Cannot load class com.foo.bar.Magic
```

该开发人员检查并确定com.foo.bar.Magic类正如所预计的那样在该Web应用的类路径中，即在WEB-INF/lib目录内的一个JAR中。这个问题的意思表达得不清楚：它是一个J2EE类装入问题吗？该开发人员编写了一个成功地按名装入这个类的JSP，并被搞得更糊涂。

现在，请看一看下面这条替代的错误消息：

```
Error in com.foo.bar.MagicServlet: Cannot load class 'com.foo.bar.Magic'
```

现在，这个问题变得显而易见：com.foo.bar.MagicServlet正在试图按名字装入类com.foo.bar.Magic，而且不知什么原因，一个结尾空格进入了该类名中。这个故事的寓意是，空格在调试语句和错误消息中是很重要的。串字面值应该用界定符括起来，以便明确地表示什么是该串的一部分和什么不是该串的一部分。在变量自身内的任何地方，都不应该使用界定符。

在公用方法签名中宁选数组而不愿选集合

Java缺少通用类型意味着每当我们使用一个集合时，都必须进行强制类型转换来访问该集合的元素，即使在我们知道它的所有元素都具有相同类型的时候（通常，我们知道）。这个为时甚久的问题在Java 1.5中可能会得到解决：借助于引进C++模板机制的一个较简单的相似机制。强制类型转换速度慢，会使代码变得更复杂，也可能很脆弱。

在一个类的实现中使用集合很少会引起严重的问题。不过，当集合用做一个类的公用接口中的参数时，可能会引起比较严重的问题，因为存在这样一种危险：外部调用者提供的集合可能会含有类型不正确的元素。因此，返回一个集合的公用接口方法将会要求调用者进行强制类型转换。

当定义公用方法的签名时，要尽可能地在指向一个集合的引用中使用一个类型化的数组。

宁选数组而不愿选集合更明确地体现了方法的用途和用法，而且还可能消除了执行强制类型转换的需要，而这种转换需要花费很大的性能代价。

不要呆板地运用这个建议，也就是说，应该根据情况灵活地运用这个建议。需要提醒的是，在下列这几种情况下集合是正确的选择：

- 当检索数据可能只是为了响应用户对集合的遍历时（这常常是JDO或CMP实体组件返回集合的情况）。
- 当元素可能不具有相同类型时。在这种情况下，可以使用一个Object型集合来建立这些数据的模型。

- 当把一个集合转换成一个数组可能效率更差时。
- 当该对象确实是关键字到值的一个映射时。
- 当该集合由一个超类返回，并且该超类可能不知道子类所处理的那些元素的类型时。

请注意，如果我们知道所有元素都具有必需的类型，用单行代码把一个集合转换成一个数组是有可能的。例如，如果我们知道集合c由Product对象构成，就可以使用下列代码：

```
Product[] products = (Product[]) c.toArray(new Product[c.size()]);
```

代码的文档编制

任何语言中缺少充分的代码文档编制都是没有道理的。在帮助开发人员编制代码文档方面，Java比大多数语言前进了一步：利用Javadoc来标准化文档编制规范。

没有充分地编制文档的代码是不完整的，可能也是无用的。

请记住，文档编制应该能够用来：

- 提供用于对象和方法的一种约定。用于一个对象的测试案例也是有价值的规范，并且文档编制和测试案例应该保持同步。
- 省去开发人员在使用代码之前需要阅读代码的麻烦。应该没有必要仔细检查一个类的代码来确定它做什么或它是否工作。Javadoc的存在就是为了确定它做什么，而单元测试应该确定它像文档编制所规定的那样工作。
- 解释该实现的不明显特性。确定什么不明显是一个棘手的问题。如果你知道自己的阅读者都是出色的Java和J2EE开发人员，那么可以不编制文档来说明语言特性，甚至不用说明还没有被普遍了解的那些语言特性，比如三元运算符。如果你不知道，则需要编制文档来说明语言特性。例如，如果你正在为一个新的部署编写一个演示应用。Java是一种小而简单的语言，所以开发人员不熟悉它的特性和常见惯用法是没有道理的。

笔者推荐下列文档编制准则：

- 学会使用Javadoc的那些特性（比如@param和@throws）。要想了解Javadoc的详细信息，请参考你的JDK版本所携带的技术资料。
- 使用关于所有方法的Javadoc注释，其中包括私用方法。使用一个使这项工作变得容易的IDE。手工生成注释是笨拙的，而且容易出错，但是IDE（比如Forte和Eclipse）能够生成桩基式Javadoc注释，进而让开发人员填充那些空白。在Javadoc注释中添加有意义的信息。要特别注意方法处理空值的方式。
- 始终编制文档来说明一个方法可能会抛出的运行时异常（如果这些异常实际是API的一部分）。或许，保证做到这一点的最佳方式是在该方法的抛出子句中声明这些异常（这么做是合法的，但不是编译程序所强制的）。例如，一个NullPointerException或许暗示一个程序设计错误，并且不应该被编制到文档中，但如果您的API（比如JDO）选择使用运行时异常，而不是已检查异常，那么指出什么东西可能会出错，以及在什么情况下调用者应该选择捕捉未检查异常是非常重要的。

- 关于方法和类的Javadoc注释通常应该指出该方法或类做什么。指出怎样实现一个类通常也是必不可少的。因此，在方法或类的主体内，要使用普通的//或/*注释。
- 为长于3行的实现注释使用/*样式注释。为较短的注释使用//注释。
- 使用关于所有实例变量的Javadoc注释。
- 当一个类实现一个接口时，不要重复关于接口约定的注释（它们不给该实现添加任何东西，而且将会配合不一致）。类中的注释应该把焦点放在该特定实现上；类中的Javadoc方法注释应该使用@see标志来参考用于该方法的接口文档编制（Eclipse为实现类自动生成这样的注释）。
- 始终编制文档来说明一个Map（映射）中关键字和值的类型，以及该Map的用途。笔者发现这对理解使用Map的类是一个巨大的帮助。
- 同样，编制文档来说明一个Collection（集合）中允许的那些元素类型。
- 确保所有注释都是有用的。Java之类的高级语言实质上是意义自明的。在你确信自己无法从代码自身中使某个东西变得明显之后，再注释它。例如，“loop through the array elements（循环各数组元素）”这样的注释就不会有什么用。
- 虽然没有必要给显而易见的东西编制文档，但是给不明显的东西编制文档是必不可少的。如果你由于某一原因需要使用一种错综复杂的迂回方法，一定要编制文档来说明它。否则，他人将来可能会转换到“自然的”方法上去，并遇到你以前设法避开的那个问题。这样的文档编制通常应该是在实现注释中，而不是在Javadoc注释中。
- 抓住一切机会改进文档编制。对如何使用一个方法感到困惑并不得不查看该方法的实现吗？一旦你知道它如何工作，就抓住这个机会改进该方法的文档编制。注意到代码中的一个不明显特性了吗？如果你不得不弄清楚它（并意识到它必不可少的），就添加一个解释它的注释。当然，这决代替不了最初编写完整的文档。
- 在每个包中包含一个package.html文件。这个文件将由Javadoc来提取（详细情况请参见Javadoc技术资料）。
- 及早编制文档，并总是使文档编制保持最新。从不计划“在编程完成之后”再添加文档编制。即便你立即就开始编写文档，仍可能已忘了某些至关重要的细节。像编写测试案例一样，编写文档有助于加强你对自己的代码和设计的了解。要考虑先编写方法文档，然后编写用于方法的测试案例，最后编写方法。使所有这3项工作保持一致。
- 不要使用“行尾”（或“拖尾”）注释。行尾注释是左对齐的，并且同它们所涉及的那条语句出现在同一行上。行尾注释往往会导致长行，而且在工作过程中需要花费时间去格式化代码，以保持注释对齐。行尾注释有时可能会用于一个方法内的变量。
- 不要把一个关于修改的日志包含在类文档编制中。把一个关于修改的日志（比如来自CVS）包含在一个Javadoc类注释是一种常见的做法。这些信息可以从源控制系统中轻松获得。修改日志将变得很长，而且无人会阅读它（他们可能也不会阅读真正的注释）。但是，把修订ID和上一个提交者包含在类注释中是一个不错的想法。如何做将随着源控制系统而有不同。

- 除非组织中的管理人员坚持，否则不要在含有公司任务陈述、冗长许可条款以及此类内容的文件的开始处使用大块注释（必要时简单地提供一个URL）。当某人打开一个文件时，如果不滚动就无法看到任何代码，这会是一件非常令人沮丧的事情。
- 不要自找麻烦地包含同版本控制系统所报告的文件路径相同的文件路径。Java的封装结构意味着我们总是知道从类路径的根到任一文件的路径（而且这个路径就是我们应该知道的一切）。
- 经常生成完整的Javadoc注释，并使它们在你的内联网上是可利用的。使用Ant或你喜欢的生成工具把Javadoc注释的生成集成到生成过程中。这不仅为开发人员提供了基本的最新信息，而且还帮助及早发现像无格式化结束标志之类的打字错误，并且还能用来使没有给他们的代码充分编写文档的开发人员感到羞愧。Javadoc还将报告诸如错误标志那样的问题，而且这样的问题应该得到纠正。

最后，如果你还没有学会编写文档，现在就从学习按指法打字开始。如果你能流畅地打字，编写注释会更容易。学会按指法打字是十分容易的。

记录日志

给代码装备工具——添加帮助跟踪应用执行的日志记录能力——是很重要的。充分的工具装备是如此重要，以致它应该是一个必不可少的编程标准。

记录日志有许多用处，但最重要的用处或许是方便调试。虽然这不是一个时髦的观点，但笔者认为调试工具被估计过高。不过，好在不是笔者一人持有这样的观点：程序设计专家 Brian Kernighan 和 Rob Pike 在由 Addison Wesley 出版的 “The Practice of Programming” 一书 (ISBN 0-201-61586-X) 中也主张这个观点。笔者发现自己用 Java 工作时很少使用调试程序。

编写代码来省略日志消息是一个低技术性但更持久的解决方案。请考虑下列问题：

- 调试会话是短暂的。它们帮助跟踪今天的错误，但明天不会使调试变得更容易。版本控制中没有今天的调试会话的记录。
- 当按步遍历代码变得必不可少时，调试是十分费时的。搜索一个日志文件中的一个指定模式可能会快得多。
- 记录日志鼓励人们思考一个程序的结构和活动情况，而不管错误是否得到报告。
- 调试程序在分布式应用中未必总是工作得很好（尽管有些IDE能够与J2EE应用服务器集成起来使调试分布式应用变得更容易）。

一个好的日志记录框架能够提供关于程序流程的详细信息。Java 1.4 和 Log4j 的日志记录包都提供描述生成日志输出的那个类、方法和行号的设置。

同一般配置一样，最好是在 Java 类外部配置日志输出。通过启用日志记录能力，在 Java 类本身内看到“冗长的”标志和类似的东西是常见的事情。这是一个不良习惯。这使重新编译类来重新配置日志记录成为了必须完成的一项工作。尤其在使用 EJB 时，这会意味着调试过程中的多个部署。如果日志记录选项被保存在 Java 代码外部，不必修改代码本身，就可以修改这些选项。

一个生产级日志记录包的需求应该包括：

- 可以让应用代码使用的一个简单API。
- 在Java代码外部配置日志记录的能力。例如，不用修改源代码就应该能为一个或多个包或类打开或关闭日志记录。
- 日志消息分成几种优先级的划分（比如调试、信息、错误等优先级）以及决定哪种优先级将是用于显示的阈值的能力。
- 程序化地查询具有给定优先级的消息是否将得到显示的能力。
- 配置消息格式化和报告消息的方式（比如报告给文件系统、为XML文档和给Windows事件日志）的能力。理论上说，这也应该以声明方式处理，并从API中被分离出来。
- 缓冲输出结果来最小化高代价的I/O操作（比如文件写入或数据库插入）的能力。

决不要把system.out用于日志记录。控制台输出无法被配置。例如，我们无法为一个特定类关闭日志记录，也无法选择显示一个消息子集。当运行在某些服务器中时，控制台输出也可能会严重损害性能。

被认为“已完成”或无隐错的代码甚至应该能够生成日志输出。但是要记住，终究有存在隐错的可能性，隐错可能会通过修改引入，或者说在一个可信模块中打开日志记录来查看开发中的其他类有什么问题可能是必不可少的。由于这一原因，所有应用服务器都能够生成详细的日志消息，只需把它们配置成这么做即可。这不仅对服务器的开发人员是有用的，而且还能帮助跟踪正运行在它们上面的应用中的问题。

请记住，单元测试的价值在于指出一个对象可能有什么问题，但不一定指出该问题出现在哪里。日志记录在这方面能够提供有用的帮助。

工具装备在性能调整方面也是十分重要的。通过了解一个应用正在做什么事情以及如何做这件事情，我们能够更容易地确定哪些操作不合理地慢。

除非代码能够生成日志消息，并且它的日志输出能够被容易地配置，否则它不可用于生产。

日志消息应该分成不同的优先级，并且调试消息应该指出一个构件的整个工作流程。调试日志消息应该经常显示对象状态（通常通过调用toString()方法）。

- 在重要的代码段中频繁地使用日志记录。
- 修改和改进维护期间的日志记录陈述（例如，如果日志输出显得不明确）。
- 为日志消息选择优先级时仔细考虑。如果所有日志消息都有相同优先级，能够配置日志输出是没有价值的。具有相同优先级的日志消息应该揭示一致级别的细节。

选择一个日志记录API

在Java 1.4发布之前，Java一直没有标准的日志记录功能度。像Servlet API那样的一些API提供了基本的日志记录功能度，但开发人员不得不依靠像Apache Log4j那样的第三方日志记录产品来实现应用级的日志记录解决方案。这样的产品增加了依赖性，因为应用代码直接引用它们，并且在EJB层中可能是有问题的。

Java 1.4日志记录和1.4以前的仿真包

Java 1.4引进了一个新包java.util.logging，这个包提供一个标准日志记录API，并且该API满足我们已经讨论过的标准。由于本书是关于J2EE 1.3的，所以下面的讨论假设Java 1.4是不可用的——如果它可用，就简单地使用标准的Java 1.4日志记录功能度。

幸运的是，从Java 1.4所引进的这个标准API中受益是有可能的，即便在运行Java 1.3的时候。这种方法避免了对专有日志记录API的依赖性，并且使向Java 1.4日志记录的最终迁移变得无足轻重。它还排除了学习第三方API的需要。

Java 1.4日志记录只是核心Java类库的一个添加物，而不是一个像Java 1.4断言支持那样的语言修改。因此，提供一个模拟Java 1.4 API的API并在Java 1.2和1.3应用中使用该API是有可能的。然后，应用代码就可以使用Java 1.4 API。虽然完整的Java 1.4日志记录基础结构将是不可用的，但实际的日志输出可以由另一个日志记录包来生成，比如Log4j（Log4j是最强有力和使用最广泛的Java 1.4以前的日志记录解决方案）。因此，Java 1.4仿真包是一个相当简单的封装器，并强加可以忽略不计的运行时开销。

惟一的收获是Java 1.4在一个新的java.util.logging包中定义了那些日志记录类。Java下的包保留给了Sun。因此，我们必须导入一个不同命名的仿真包（笔者选择了java14.java.util.logging）来取代Java 1.4 java.util.logging包。这一导入可以在代码迁移到Java 1.4时修改。

附录A将讨论本书的基础结构代码和示例应用中所使用的Java 1.4日志记录仿真包的实现。

可以证明的是，Log4j比Java 1.4日志记录更有威力。那么，为什么不直接使用Log4j呢？使用Log4j在有些应用服务器中可能会有问题；使用一个标准Java API的优点是显而易见的，而且在使用Java 1.4 API的同时使用Log4j的强有力输出特性也是有可能的（差别十分小）。但是，当使用一个已经使用了Log4j的第三方产品（比如许多开放源产品）时，直接使用Log4j可能是一个不错的选择。

对于Web应用中的日志记录，我们还有另一种选择。Servlet API提供了可供任一Web构件使用的日志记录方法，只要该Web构件有权访问该应用的ServletContext即可。由Servlet API所供给的javax.servlet.GenericServlet小服务程序超类提供了对相同日志记录功能度的方便访问。不要使用Servlet API日志记录。一个应用的大部分工作应该在普通Java类中完成，不用访问Servlet API对象。最后不要记录到不同日志的构件。要为所有日志记录都使用同一个解决方案，其中包括来自小服务程序的日志记录。

Java 1.4日志记录惯用法

一旦我们导入了Java 1.4 API，就可以使用该仿真包。要想了解详细信息，请参见Java 1.4 Javadoc。

最重要的类是java.util.logging.Logger类，这个类即用来获得一个日志记录器，又用来写日志输出。最重要的方法是：

```
Logger.getLogger(String name)
```

这个方法获得一个与给定构件关联的日志记录器对象。约定是：代表一个构件的名称应该是类名。例如：

```
Logger logger = Logger.getLogger(getClass().getName());
```

日志记录是线程安全的，因此获得并高速缓存一个将要在整个类生成周期内使用的日志记录器显然更有效，而且会产生更简单的代码。笔者通常使用下列实例变量定义：

```
protected final Logger logger = Logger.getLogger(getClass().getName());
```

抽象超类常常包含这个定义，以便子类不用导入任何日志记录类或日志记录器就能执行日志记录。需要注意的是，根据前面所讨论的那些可见度准则，该保护实例变量是最终的。日志记录调用看起来将会像下面这样：

```
logger.fine("Found error number element <" +  
    ERROR_NUMBER_ELEMENT + ">; checking numeric value");
```

Java 1.4 日志记录在 `java.util.logging.Level` 类中定义了下列日志级别常数：

- **SEVERE**: 表示一个严重失败。常常将有一个伴随的 `Throwable`。
- **CONFIG**: 为应用配置期间所生成的消息而设计的。
- **INFO**: 中等优先级。表示一个构件正在做什么（例如，监视执行一项任务时的进度）而不是打算帮助调试该构件。
- **FINE**: 跟踪信息。该级别和较低优先级的级别应该用来帮助调试特有类，而不应该用来总体地说明该应用的工作方式。
- **FINER**: 详细的跟踪信息。
- **FINEST**: 非常详细的跟踪信息。

每一级别都有一个对应的便利方法，比如 `severe()` 和 `fine()`。通用方法允许把一个级别分配给一条消息，并记录一个异常。

每条消息都必须被分配这些日志级别中的一种级别，以保证日志记录的粒度能够在运行时被轻松地加以控制。

日志记录与性能

一个日志记录框架的正确使用对性能应该只有微不足道的影响，因为一个日志记录框架应该只占用极少的资源。通常，应用应该被配置成只记录生产中的错误，以避免过度的系统开销和过大日志的生成。

保证日志消息的生成不减慢应用的速度是十分重要的，即便这些消息从未显示。这方面的一个常见故障原因是把 `toString()` 方法用在访问许多方法并生成大串的复杂对象上。

如果一条日志消息可能会生成得很慢，检查它是否将在生成之前被显示是很重要的。一个日志记录框架必须提供速度快的方法，以指出具有一个给定日志优先级的消息是否将在运行时被显示。Java 1.4 提供了执行检查的能力，比如：

```
if (logger.isLoggable(Level.FINE)) {  
    logger.fine("The state of my complex object is " + complexObject);  
}
```

如果针对这个给定类禁用了FINE日志输出，这段代码将执行得非常快，因为**toString()**方法在**complexObject**上将得不到调用。串操作的执行代价高得惊人，所以这是一个非常重要的优化。

另外需要注意的是，只要确保日志记录语句调用**toString()**方法将要针对的对象不会是空值，这些语句就不会引起失败。

与日志记录有关的一个同等重要的性能问题涉及到日志输出。Java 1.4日志记录和Log4j都提供了相应的设置，用来显示生成日志输出的那个类、方法和行号。该设置在生产中应该被关闭，因为生成这类信息的代价非常高（这类信息只能通过生成一个新的异常并语法分析它的栈跟踪串——正如它的**printStackTrace()**方法之一所生成的一样——来生成）。但是，这类信息在开发期间是非常有用的。Java 1.4日志记录允许程序员通过日志记录API提供类名和方法名。以使日志记录消息的编写变得更困难和阅读变得稍微麻烦一点为代价，这保证了这类信息将可高效地获得，即使一个JIT已经使得从一个栈跟踪中查找足够细节变成了不可能。

对性能有明显影响的其他日志记录系统配置选项包括：

- 日志消息的目的地。写日志消息到控制台或到一个数据库可能会比写到一个文件慢。
- 最大文件大小和文件翻转配置。当现有日志文件达到某一大小时，所有日志记录包都应该允许到一个新日志文件的自动翻转。允许一个太大的最大文件大小可能会明显减慢日志记录的速度，因为到文件的每次写可能会涉及真正的系统开销。调整翻转之后仍保住的日志文件数量通常是十分必要的，因为日志记录会以其他方式耗用大量磁盘空间，而这可能会引起服务器和应用发生故障。

EJB层中的日志记录

和许多其他方面一样，在日志记录中，EJB层引起特殊问题。

- EJB程序设计限制不允许配置从文件系统中被装入，也不允许写（任何日志文件）到文件系统。
- 大多数的日志记录框架从技术上违反了由EJB规范施加在应用代码上的程序设计限制（§ 24.1.2）。例如，几个核心的Log4j类使用同步。
- 可能会使用远程调用被传递给EJB层或从EJB层中传递过来的对象怎么才能处理日志记录，因为它们的执行横跨了截然不同的虚拟机？

下面让我们依次讨论上述每一个问题。

日志记录配置不是一个主要问题。我们可以从类路径中装入日志记录配置，而不是从文件系统中，因而允许它被包含在EJB JAR文件中。

利用日志输出做什么是一个比较严肃的问题。有时提议的两个解决方案是使用不允许EJB使用的企业资源（如数据库）写日志输出；或者，使用JMS来出版日志消息，进而希望一个JMS消息消费者将能够利用它们做一些合法的事情。

这两个解决方案没有一个是具有吸引力的。使用一个数据库将会引起日志记录对性能产生严重的影响，而这种影响将要求所讨论的日志记录的生存能力。数据库决不是寻找日志消息的一个理想场所。使用JMS只是把该问题推到了别的某个地方，而且从技术上讲杀伤力也

过强（JMS也可能有一个显著的系统开销）。

反对把数据库之类的企业资源以及JMS题目或队列用于日志记录的另一个有力证据是，我们将需要把一个失败记录到正用来产生日志输出的企业资源中的实际可能性。假设我们需要记录该应用服务器的失败来访问它的数据库。如果我们尝试写一条日志消息到同一个数据库，就会产生另一个失败，并且不能生成一条日志消息。

对待EJB程序设计限制不能太教条是很重要的。请记住，EJB应该用来帮助我们实现我们的目标，我们不应该让采用它而使我们的生活变得很困难。日志消息的目的地最好是放在日志记录系统配置中处理，而不要放在Java代码中处理。据笔者看来，最好是忽略这些限制，并记录到一个文件中，除非你的EJB容器反对（请记住，EJB容器必须内部地执行日志记录；例如，JBoss使用Log4j）。我们可以修改日志记录配置，如果使用一个数据库或其他输出目的地（这可能是必需的，如果该EJB容器不一定位于一个文件系统上；例如，如果它被实现在一个数据库上）。

笔者觉得，该同步问题需要我们结合实际来考虑对EJB规范的严格解释做一些类似的变通。避免使用在EJB中利用了同步的库是不切实际的（例如，这将会排除使用Java 1.2以前的所有集合，比如java.util.Vector；虽然当今使用这些遗留类没有太好的理由，但是大量已有代码已经使用了这些遗留类，而且把它们排除在EJB世界之外是不可能的）。在第6章中，我们将比较详细地讨论EJB程序设计限制。

最后，就分布式应用使用EJB而言，我们必须考虑远程方法调用这个问题。Java 1.4日志记录器不是可串行化的。因此，当我们在将要在体系结构层之间传递的对象（比如在EJB容器中创建，随后在一个远程客户JVM中访问的值对象）中使用日志记录时，需要特别小心。有3种似乎可能的可替代方法：

- 不在这样的类中使用日志记录。有这么一种强烈的主张：这样的对象基本上是参数，不应该有足够的智能来要求日志输出。
- 利用每条日志记录语句来获得一个日志记录器，从而保证该对象将始终获得一个合法的日志记录器，无论它运行在何种JVM中。
- 通过实现一个私用方法getLogger()来获得一个日志记录器，其中每条日志记录语句都使用这个方法替代一个实例变量，以获得一个日志记录器。

上述第三种方法允许高速缓存，并且将提供最佳性能，尽管复杂性通常不是十分合理。下列代码段举例说明了这种方法。需要注意的是，logger实例变量是过渡变量。当一个这样的对象被作为一个远程参数传递时，这个值将被保留为空值，因而提示getLogger()方法为新的JVM高速缓存该日志记录器。

```
| private transient Logger logger;  
  
private Logger getLogger() {  
    if (this.logger == null) {  
        // Need to get logger  
        this.logger = Logger.getLogger(getClass().getName());  
    }  
    return this.logger;  
}
```

在突出显示的那一行上，出现一种竞争状态是有可能的。但是，这不是问题，因为对象引用（比如logger实例变量）是有原子性的。会出现的较糟糕情况是，频繁的并发访问可能会导致多个线程对Logger.getLogger()方法做不必要的调用。该对象的状态不会被破坏，所以没有理由同步这个调用（当该对象用在EJB容器中时，同步将是不符合要求的）。

为什么（以及怎样）不从头开始

到目前为止，我们已经考虑了帮助我们编写高质量、可维护代码的设计与编程标准。专业的企业设计师和开发人员不仅要编写优质的代码，而且还要避免编写他们不必编写的代码。

许多常见问题（超出J2EE应用服务器解决范围的那些问题）已经由开放源或商业包及框架很好地解决。在这类情况下，设计并实现一个专有解决方案可能是浪费精力。通过采用一个现有解决方案，我们能够自由地把所有精力都用于满足业务需求。

在本节中，我们将看一看使用第三方框架来重用现有投入的问题。

救命！API过载

对于J2EE中的大多数问题，当今有许多API和技术选择。

现在，甚至连Sun似乎都处于这样一个关头：把一切都收集在一起已将事情搞得特别复杂，以致我们正看到明显的重复劳动。例如，JDO和带CMP的EJB 2.0实体组件似乎就有明显的重叠。

最后，我们将会为重复劳动付出代价：工作量越来越多，而质量越来越低。至少，我们可以在组织内竭力控制重复劳动。笔者以为，代码重用是有可能的，而且我们应该尽力实现代码重用。

避免重复劳动和利用现有投入有许多种方法。笔者建议以下列准则作为一个起始点：

- 尽可能采用现有框架。例如，为Web应用使用一个标准日志记录框架和一个现有框架。但是，不要强迫开发人员使用组织级的标准框架，如果他们似乎还没有充分胜任手边的问题。在存在多个可选框架的情况下，通盘考虑各种选择。不要自以为你所见到的第一个产品或最流行的产品将最满足你的要求。
- 决不容忍代码重复。这表明需要通用化：尽力避免最初的代码重复，但是一旦出现了代码重复，立即尽力再加工它。
- 确保开发人员之间的充分交流。例如，让开发人员介绍他们最近刚完成的模块，以便其他开发人员知道正出现或已满足了什么样的公共要求。鼓励开发人员促使其他开发人员使用他们已经实现的基础结构构件。
- 开发和维护这样一些简单的基础结构包：这些包实现了得到广泛使用的功能度。编写文档充分说明这些包，并保证开发人员都了解它们。
- 采用标准的体系结构模式，即使在共享代码的可能性不大的地方。当处理类似的模式时，避免重复劳动是比较容易的。
- 使用代码审查。这不仅帮助提高质量，而且还会激励团队内的交流。

使用框架

要想利用现有构件，无论第三方构件还是自产构件，一种特别有用的方法是在一个框架内构建。一个框架就是一个通用的体系结构，它构成了用于某一行业或技术领域内具体应用的基础。

框架不同于类库，因为选用一个框架就确定了一个应用的体系结构。使用了一个类库的用户代码直接处理控制流本身，而框架通过使用类库对象作为助手来负责控制流程，因而调用用户代码（我们已经谈论过控制反转和好莱坞规则——“不要给我打电话，我会给你去电话”）。这采取了和Template Method设计模式相同的方法，但把它运用在了一个更大的规模上。

框架在如下方面有别于设计模式：

- 框架是具体的，不是抽象的。虽然设计模式是概念性的，但你可以采用一个现有框架，并利用它和通过添加额外代码来构造一个应用。这通常会采取实现框架接口或子类化框架类之类形式。
- 框架比设计模式的层次高。一个框架可能会使用几个设计模式。
- 框架通常是领域特有的或技术特有的，而设计模式可以应用于许多问题。例如，一个框架可能处理保险报价单，或者提供Web应用的逻辑与表示的明确分离。大多数设计模式可以用在几乎任何应用中。

采用一个适应性很强的好框架可以大大缩短一个项目的开发时间。最难处理的设计问题可能已基于公认的最佳实践而得到了解决。项目的大部分实现将是专门填充那些空白，而这些空白不应该涉及太多困难的设计决策。

另一方面，试图强行使一个项目使用一个适应性很差的框架将会引起严重的问题。这些问题将比选择一个不合适的类库糟糕得多。在这种情况下，可以忽略类库：应用开发人员将不得不开发他们自己的、更合适的类功能度。一个适应性很差的框架将会给应用代码施加一个不自然的结构。

产生的应用的性能和可靠性也决不会比框架的更大。通常，这不是什么问题，因为一个现有框架很可能已在早期的项目中得到广泛使用，并且它的可靠性和性能特征是已知的，但就所有情况而论，在做决定以前应该对一个框架进行一次全面的质量检查。

成为一个好框架的条件

好框架使用起来简单，并且又有威力。

框架设计的Scylla和Charybdis是过分的灵活性和极端的刻板。

在希腊神话中，Scylla是一个六头海怪，她和海怪Charybdis各居住在Messia海峡的一边。船员必须绘制这两个妖怪之间的一幅航线图。

过分的灵活性意味着该框架含有可能永远都得不到使用、并且处理起来可能会令人迷惑的代码（该代码测试起来也将更困难，因为要涉及的可能性太多）。但是，如果一个框架没有灵活得足以满足一个指定需求，开发人员将会高高兴兴地实现他们自己的做事方式，以致该框架在实践中没有提供一点好处。

好框架代码与好应用代码的差别并不大。一个好框架可能会含有复杂的代码；这是合理的，只要它已向使用它的代码隐藏起了这种复杂性。一个好框架能够简化应用代码。

使用现有框架的好处

一般说来，比较好的做法是避免内部地构造任何简单的框架。在过去的这几年中，开放源已经有了长足的发展，特别是在Java方面，而且有许多现有的框架。开发好框架比开发应用难得多。

采用一个现有框架的主要好处和采用J2EE本身的好处一样：它能使一家组织的开发团队把主要精力集中在开发必需的产品上，而不是集中在操心底层的基础结构上。如果一个第三方框架是流行的，使用这个框架也有一个潜在的好处：技能的可利用性。

同往常一样，也有一个折衷问题：采用该框架的学习难度和对该框架的持续依赖性。

项目越复杂，证明初始投入和后续依赖性的合理性就越容易。

评估现有框架

采用一个框架是一个非常重要的决策。在某些情况下，这可以决定一个项目是成功还是失败；在许多情况下，这将决定开发人员的工作效率。同选择一个应用服务器一样，在做出一个决策之前执行一次全面的评估是很重要的。需要记住的是，即便选择一个框架没有涉及到任何许可成本（在开放源框架的情况），仍有许多需要考虑的其他成本，比如学习难度对开发人员工作效率的影响和处理该框架中的任何错误的潜在代价。

笔者运用下列标准来评估现有框架。按照这一次序运用它们往往可以限制用来评估不适合产品的时间量：

- 项目文档编制的质量是什么标准？
- 项目的性质是什么？
- 设计健全吗？
- 代码的质量是什么标准？
- 发布包括测试案例吗？

下面让我们依次看一看上述每个标准

项目文档编制的质量是什么标准

有一个解释该框架的基本原理和设计的相关（和有说服力的）综述性文档吗？有用于所有类的那些Javadoc吗？它们含有有意义的信息吗？

项目的性质是什么

如果该产品是商业产品，主要考虑事项将包括供应商的状态、该项目在供应商的策略中的地位以及许可策略。采用一个商业的、封闭源的、供应商将会倒闭或放弃以致用户得不到支持的产品是十分危险的。很显然，这种事情在大型供应商身上不太可能发生。

但是，像IBM那样的大公司启动了许多项目，而且这些项目不一定都适合他们的长期策略（请看一看IBM Alphaworks站点上的许多项目）。供应商的生存能力决不意味着他们将

继续给任何个别项目提供资源和支持。最后，尤其需要注意的是，如果该产品是商业的，但目前是免费的，许可协议中的限制性附属细则暗指该供应商随时开始收取许可费用吗？你的组织准备接受这个细则吗？

如果该产品是开放源产品，则有不同的考虑。该项目有多强的生命力？有多少开发人员正在从事它？上一次发布是在什么时候，以及发布的频率如何？项目文档编制引用了参考站点吗？如果引用了，它们给人的印象有多深刻？该项目的邮件列表有多活跃？寻求支持有地方可去吗？该项目的开发人员愿意提供帮助吗？也就是说，既要有愿意答复新闻组问题的开发人员，又要有付费的咨询。

像SourceForge (<http://www.sourceforge.net>) 那样的站点有关于项目活动性的统计数字。其他的指标表示方法是活动的邮件列表并使用你所喜爱的搜索引擎搜索关于该产品的资料。

许多经理对采用开放源产品持怀疑态度。虽然项目的质量差别很大，但这样的怀疑态度正变得越来越没有道理。毕竟，Apache现在是得到最广泛采用的Web服务器，而且已经证明是非常可靠的。有几个开放源Java产品也使用得非常广泛，比如Xerces XML语法分析器和Log4j。我们也正看到像IBM那样的主要商业公司在开放源方面的兴趣。例如，Xalan和Eclipse就是最初在IBM开发出来的两个著名开放源项目。

设计健全吗

项目的文档编制应该描述所使用的设计（例如，设计模式和体系结构方法）。这个设计满足你的需要吗？例如，一个完全以具体继承性为基础的框架（比如Struts）可能会证明是欠灵活的。这不仅可能会为你的代码带来问题，而且还可能会需要框架本身的彻底修改，以便将来增加新的功能度。如果你的类被迫扩展框架类，这可能会要求你的组织将来为转换为别的产品而付出大量的精力。

代码的质量是什么标准

如果源代码是现成的，提高代码质量可能很耗时，但也是非常重要的。假设该产品已经满足前面的标准，那么时间的投入是完全值得的。

花费半天时间来浏览代码。给它应用你给组织内的自编代码所应用的相同标准，并查看部分核心类来评估该实现的洁净度、效率和正确性。作为一个意外的收获，你的团队最终将会了解到该技术的大量信息，如果该框架被编写得十分完美，可能还会了解到一些有用的设计和编程技巧。

发布包括测试案例吗

利用一群在地理上很分散并通过电子邮件和新闻组进行交流的开发人员来开发可靠的软件有一些困难。保证质量的方式之一是开发一个测试集。像JBoss那样的成功开放源产品都有大型的测试集。如果一个开放源产品没有一个测试集，这就是一个令人担心的信号。如果坚持使用这样一个开放源产品，你可能会发现，由于缺乏回归测试，你的应用随着每次的新发布而突变一次。

实现自定义框架

开发内部框架的第一条原则是：不开发。一般说来，采用现有解决方案比较好。

但是，有些情况下，我们会有与众不同的要求，或现有框架不能满足我们的要求。在这种情况下，开发一个简单的框架比使用一个不合适的现有产品或随意地编写不利用任何框架的代码要好得多。

即便在这种情况下，过早地投入也不是一个好主意。只有在你了解了问题之后，才应该尝试设计一个框架，然后尽力设计尽可能简单的框架。不要指望你的第一个设计将是完美的：在做一次重大的提交之前，让这个设计有一段发展的时间。

向现有框架学习

由于编写框架很难，所以成功的框架是现实设计的最有价值的例子。要密切关注你所在领域和其他领域内的成功框架，它们所使用的设计模式，以及它们使应用代码能够扩展它们的方式。

实现一个框架

当实现一个框架时，事先有明确的目标是至关重要的。预见该框架未来的每个需求是不可能的，但是，除非你对自己需要实现什么已经胸有成竹，否则你将会对结果感到失望。

确定框架作用范围的最重要教训是用最小复杂性实现最大价值。我们经常发现这样一种情况：该框架能够相当容易地解决某一领域内的大部分而非全部的问题，但提供一个完整解决方案是很困难的。在这种情况下，对一个简单解决方案能够解决90%的问题感到满足或许更可取，不要试图强行通用化剩余的10%。

如果设计一个框架，就应用Pareto Principle（巴累托定律）。如果某个特定功能似乎特别难实现，问一问自己它是不是必不可少的，或者该框架能不能实现其价值的大部分（如果不能完全解决这个问题）。

编写框架在以下几个方面有别于编写应用代码：

- “编写能够管用的最简单东西”的XP忠告未必总是适用的

如果不破坏使用一个框架的代码，而只是大大降低它的有用性，再加工该框架所暴露的那些接口则是不可能的。即使在一家组织内部，对一个框架做不兼容修改的代价也是非常大的（另一方面，再加工一个框架的内部是有可能的）。因此，必须在前面就把该框架设计成满足先前就预见到的要求。不过，添加不需要的灵活性会增加复杂性。这种权衡要求很好的判断力。

- 提供不同级别的复杂性

成功的框架在几种层次上提供接口。让开发人员不用过分学习就能有效地处理它们是很容易的。然而，让具有更复杂需求的开发人员能够使用更多特性（如果他们愿意）也是有可能的。目标是开发人员应该能够处理决不比子头任务所要求的复杂性更高的复杂性。

- 区别对待框架内部与外部

外部应该是简单的。内部可能是较复杂的，但应该被封装起来。

- 拥有一个全面的测试集尤其重要

框架隐错的代价通常比应用隐错的代价高得多，因为一个框架隐错可能会引起许多流动性隐错，并必须找出高代价的克服方法。

伊利诺斯大学的Brian Foote和Joseph Yoder所撰写的一篇题为“*The Selfish Class*”的杰出文章，使用了一个生物学类比来刻画产生代码重用的成功软件制品。它与框架设计特别有关（请参见<http://www.joeyoder.com/papers/patterns/selfish/selfish.html>）。关于从XP角度的讨论，请参见<http://c2.com/cgi/wiki?CriticalSuccessFactorsOfObjectOrientedFrameworks>。

小结

J2EE项目往往都是复杂的，这使好的程序设计习惯变得至关重要。

在本章中，我们已经了解了好的OO习惯怎样加强好的J2EE应用。

我们还了解了一致应用健全编程标准的重要性，以便允许有效的团队合作并帮助保证应用易于维护。

最后，我们讨论了如何避免编写代码：通过利用现有框架，以及（作为最后一种手段）实现我们自己的框架。

表4.2总结了我们已经讨论过的OO设计原理。

在下一章中，我们将从理论转向实践，了解我们将要在本书其余篇幅中讨论的示例应用的各种需求。

表4.2

技巧	优点	缺点	相关设计模式	性能影响
编程到接口，不要编程具体类。应用构件之间的关系用接口来表示，不要用类来表示	促进设计灵活性。 当接口通过Java组件实现并通过它们的Bean属性配置时，工作得很好。 不排除具体继承性的使用。 实现可以有一个平行但不同于接口的继承性分级结构	实现起来比使用具体继承稍复杂一点。 许多设计模式都基于接口继承性	Strategy (GoF)	可以忽略不计
对象合成比具体继承性更可取	促进设计灵活性。 避免由于Java缺少多具体继承性带来的问题。 使类行为能够在运行时被修改	可能会导致类数量的增加。 对简单需求来说可能会是小题大做	无	无
当知道如何实现一个工作流程但不知道所有单个步骤该如何实现时，使用Template Method设计模式	保证工作流程能被实现和测试一次。 对解决J2EE中的可移植性问题非常理想。	有时委托是一个更好的模型，而且Strategy模式更可取	Template Method (GoF)	无
当希望得到委托而非具体继承性的灵活性时，使用Strategy设计模式作为Template Method模式的一种替代方案	当实现接口时，有比使用具体继承性更大的自由度。 实现可以在运行时变化。 实现可以与其他类共享	实现起来比Template Method模式稍微复杂一点，而Template Method模式常常作为一个替代方案	Strategy (GoF)	无

(续表)

技巧	优点	缺点	相关设计模式	性能影响
在集中化工作流程的语句时使用回调方法来实现可扩展性	当其他方式不能实现代码重用时能够实现代码重用。允许错误处理代码的集中化。通过从应用代码中把复杂性转移到框架中，降低错误的可能性。	概念上复杂，尽管使用它的代码一般比使用其他方式的代码更简单。	Strategy设计模式的一种特殊情况 (GoF)	如果回调接口被经常调用，会有微小的性能下降。
使用Observer设计模式	通过把监听者与生成事件的业务逻辑的执行分开，促进关系的分离。无需修改现有代码就能够提供可扩展性。	引入未必总是必要的复杂性。需要一个事件出版基础结构和事件类。	Observer (GoF)	拥有太多的观察者（监听者）会减慢一个系统的速度。
把多个方法参数组合成一个单独的对象	允许Command设计模式的使用。利用断开式接口来使扩展功能变得更容易。	增加一个系统中的对象数量	Command (GoF) EJB Command (EJB Design Patterns)	有助于“对象搅拌”。在EJB调用之类的相当频繁的调用中，必要的对象创建的代价可以忽略不计。在一个嵌套循环中，代价可能会很惊人。

(续表)

技巧	优点	缺点	相关设计模式	性能影响
把未检查异常用于不可恢复的错误; 当调用代码有可能能够处理该问题时使用已检查异常	较少的代码。 更易读的代码; 业务逻辑不会由于捕捉不能处理的异常而变得模糊不清。 增强的工作效率。 不必捕捉、包装和重新抛出异常; 失去栈跟踪的可能性较小。	许多Java开发人员习惯于几乎只使用已检查异常。 当使用未检查异常时, 请务必记住编写文档来说明可能被抛出且编译程序无法协助的那些异常。	Factory (GoF)	依赖于调用的频率。 通常没有显著作用。
使用反射	参数化Java代码的一种强有力方法。 优于实现Factory设计模式。 当与Java组件结合起来时非常强有力。 帮助解决J2EE中的可移植性问题。	反射会得到过度使用。 有时, 一个较简单解决方案同样有效。		通常可以忽略不计
把应用构件实现为Java组件		使以声明方式配置系统变得更容易, 从而与J2EE部署方法一致。允许输入验证之类的问题通过标准JavaBean API的使用来解决。	全部	

(续表)

技巧	优点	缺点	相关设计模式	性能影响
通过使用一个应用上下文或注册表来避免单元元素的激增	<p>促进设计灵活性。 使我们能够把“单元元素”实现为普通Java组件；它们将通过它们的Bean属性被配置。</p> <p>在Web应用中，我们可以把上下文放到ServletContext中，从而避免在注册表上需要一个getInstance()方法。</p> <p>在一个J2EE服务器内的任何地方，我们可以把注册表绑定在JNDI中。我们或许能够使用JMX，支持“单元元素”的重新装入是有可能的。</p> <p>应用上下文可以提供其他服务，比如事件出版。</p>	<p>注册表将需要Java外部的配置，比如一个XML文档。这对复杂应用来说是一种杰出来方法，但对非常简单的应用来说是不必要的。</p>	<p>Singleton (GoF) Factory (GoF) Prototype (GoF)</p>	无

(续表)

技巧	优点	缺点	性能影响
从JavaSoft的编程约定开始	使新来的开发人员阅读你的代码变得更容易。熟悉Sun的约定使你读其他人的代码变得更容易。	无	无
对象和方法应该拥有明确的责任	使代码变成意义自明的。	局部化修改的影响范围	无
避免代码中的字面值参数	使阅读和维护代码变得更容易，降低打字错误引起微妙问题的可能性。	无	无
只使用私用实例变量。必要时提供获得器和设置器方法	促进黑箱类重用和松耦合。公用实例变量允许对象状态被其他任一对象破坏，保护实例变量允许类状态被子类或同一个包中的类破坏	使用私用而非保护实例变量削弱了子类修改超类行为的能力。不过，这通常是一件好事情	使用方法时性能开销可以忽略不计，使用直接变量访问时，则不然
使一个类的公用接口保持为一个最小程度	帮助实现类之间的松耦合。	使类更容易使用	无
适当地使用最终方法	最终方法可以用来防止子类通过覆盖方法不正确地修改超类行为	限制子类的作用范围来定制超类行为。不过，覆盖具体方法是实现可扩展性的一种粗劣方法	稍有改善，因为JVM知道这个方法定义在哪个类中
实现所有类都有toString()方法	如果所有类都有toString()方法，调试就会容易得多，尤其是在和一个合理的日志记录策略结合起来时	无	toString()方法调用起来会花费很高代价，所以保证它不被多余地调用是很重要的（例如，通过生成将不被输出的日志消息）

(续表)

技巧	优点	缺点	性能影响
消除代码重复	代码重复对维护来说是灾难性的，通常也体现了开发中的时间浪费。要坚持不懈地消除代码重复	无	无
不在可以使用一个数组的地方公开地暴露无类型集合	帮助代码变成意义自明的，并消除不正确使用的可能性。	有时，把数据转换到一种数据类型是笨拙的和缓慢的，或者我们需要一个集合（例如，为实现一个懒成熟化）	不确定。如果把一个集合转换到成一个数组比较缓慢，使用这种方法可能就不是一个好主意
彻底编写文档来说明代码	避免昂贵的、易错的类型转换	无	无
给代码装备日志记录输出	没有用文档充分说明的代码是没有完成的，可能也是没有用处的。标准Javadoc工具是我们的文档编写策略的基础	在调试和维护期间有巨大的帮助。 对管理一个运行应用的职员会有帮助	日志记录的草率实现或一个日志记录系统的错误配置可能会降低性能。不过，这可以通过仅当我们被显示时才生成日志消息来加以避免

第5章 示例应用的需求

在本章中，我们将指定一个非常现实的应用，足以作为J2EE体系结构决策的讨论和一个有意义的实现练习提供基础。这个应用的设计以业务角度（由一位业务分析员朋友慷慨提供）为出发点，而不是以J2EE技术为出发点。为了避免引入不相关的细节，这个应用比一个实际应用更简单，但是为了举例说明一组预想的J2EE技术，这么做是值得原谅的。

为了加强本书剩余篇幅内的技术讨论和一个测试策略的开发，以及当我们着手开始实现时，我们将再返回来参考本章。

请注意，实际的应用软件需求比本章中将介绍的那些需求更正式。例如，笔者没有使用任何特定的方法学（比如Rational Unified Process）来描述需求。这里的目标不是提供一个如何形式化业务需求的例子，而是介绍将为本书的剩余篇幅要建立的J2EE解决方案提供一个讨论基础的易懂而又简明的需求。

概述

这项任务是为X Center开发一个联机座位预订系统，X Center是一家艺术与娱乐中心，上演多种流派的节目，比如歌剧、音乐会、戏剧和喜剧。X Center目前有3个演出大厅。每个大厅都有一个用于大多数演出的默认座位计划，但有时，一个特殊节目可能会稍微改变这个计划。对于Mahler的第八交响乐，座位计划可能需要从销售中扣除20%的座位，以便容纳合唱队。当Zanetti的马戏团演出时，舞台与大厅后部之间的每一排中必须有6个座位从销售中被撤回，以便腾出空间为狮子进场安排一个带栅栏的通道。这也影响被考虑为邻近的那些座位。

X Center由Z集团拥有，这家集团是一家还拥有许多其他剧院和娱乐场所的国际娱乐公司。Z集团把X Center的联机订票系统看做一个典型试验，并希望把它用做他们的其他娱乐场所的一个系统基础。因此，从最初发布开始，这个系统必须兼顾到品牌的重贴。在不改变基本功能度的情况下修改站点的外观必须是可能的，以便该集团内的另一家剧院能够让顾客利用贴有该剧院自身品牌的标识来订票。

X集团的其他一些娱乐场所都位于非英语国家，因此，尽管没有立即的国际化需求，但是，如果该系统是成功的，国际化将是必不可少的。有些场所位于像瑞士那样的多语种国家，因此，如果该系统在这些场所得到采用，它必须允许用户选择他们自己的语言。

X Center目前没有任何联机售票系统。它的5到10名售票处雇员（高峰期间雇用更多的临时雇员）目前使用一个基于Oracle数据库的客户-服务器系统。这个系统中可能没有任何一个部分（除了数据库模式的某些部分之外）能够重新用于该Web界面。

下列这些业务需求并不打算描述一个“现实”订票系统的每个功能性方面。

用户总数

这个应用必须支持3个用户群：公共Internet用户（顾客）、售票处用户（给已经选择不联机购票的顾客提供帮助的X Center雇员）和管理员（负责张贴和更新数据的X Center雇员）。

公共Internet用户

这些用户是顾客，即公众的成员，他们通过一个Web浏览器访问该服务。大多数人使用一台调制解调器按最大56K的速度进行连接，因此该系统在一台调制解调器连接上必须有一个可接受的响应时间。顾客希望：

- 发现正在出售什么节目的入场券，并访问关于节目和演出的信息。
- 发现还有什么演出日期的入场券可供购买。
- 使用信用卡联机地预订座位。

将来可能会给各别的“注册用户”团体提供不同的奖励性服务（比如在单个事务中预订大量座位的能力）是有可能的。但是，在最初的系统中，注册只提供为一个便利设施，以省略用户重新输入他们的付款地址。这个优先考虑的事情是保证购买入场券的途径尽可能直截了当，因此用户将不会被迫注册，强迫注册可能会有使用户疏远的危险。

为了达到这些需求，该系统应该提供一个简单、有用和快速的Web界面。这个界面能够在尽可能多的浏览器上工作是很重要的。浏览器特有的功能度应该避免，而且所要求的浏览器级别应该在不牺牲功能度的情况下尽可能低。小程序、Flash动画和此类东西被认为可能会吓住用户，并且不应该使用。

售票处用户

这些用户是工作在X Center自身内部的X Center雇员，并且是在局域网上。有些雇员可能会在家里工作，在这种情况下，他们将有权使用一条宽带Internet连接。售票处用户执行下列活动：

- 答复由公众成员打来的电话和现场的询价，主要是关于座位的可获得情况。为了支持这项任务，售票处用户必须被授予公共Internet用户不能使用的强有力查询功能度，以便使他们能够快速地响应诸如“我最早可以有25个邻近Carmen的B Reserve座位是什么日子？”之类的问题。
- 把入场券卖给希望看节目的公众成员。
- 把入场券供应给已经联机订了票但已决定从售票处取票而不愿意让邮局邮寄它们的顾客。这对没有时间邮寄入场券时的后期订票也是必不可少的。
- 有时，按照电话请求来取消预订。

这些用户主要关心的是服务可靠性和可用程度。他们连续不断地接收到期望当场得到服务的客户打来的询价电话。因此，公共Internet界面不应该降低售票处界面的可靠性或性能。

由于售票处用户会有机会接受关于如何使用该系统的培训，所以要优先考虑的事情应该是功能度，而不是易使用性。必须给售票处用户提供一个强有力的系统，并且该系统提供一种执行常见任务的快速方式。由于售票处界面仅供内部使用，所以决不应该过分强调它的品牌和装饰性。

X Center管理团队希望最大化他们的IT投资，所以认为相同的联机订票系统应该为公共Internet用户和售票处职员服务。同顾客一样，售票处用户将使用Web界面。这个界面将通过基于角色的安全来阻止Internet用户，并且将有一个不同于公共Internet应用的进入点。该系统将根据用户名和密码自动授予适当的权限。

管理员

这些用户是X Center的雇员，管理X Center中运转着的售票系统。管理员用户必须在X Center自身内工作，以便访问本地网。他们执行各种各样的行政、市场营销和管理功能：

- 添加新的节目和演出。
- 运行管理信息和财务报表。
- 配置售票系统的设置，比如同一个事务中能被购买的最大入场券数量，以及付款前入场券能被保留的期限。

X Center的管理层希望在他们的系统内使用一个一致的技术集，以便最大限度地减少他们的IT部分所需要的技能数量，因此他们希望同客户与售票处界面一样，也为管理员界面使用同一种Web技术，而不是使用像Swing或VB之类的另一种技术。管理层也希望管理员界面应该不需要安装；已被授权的职员应该能够在X Center办公楼内的任何一台机器上访问管理员界面，并且管理员界面应该不需要转出过程和前进中的更新。

管理员界面将提供运行在一个端口上的额外安全，并且这个端口只有在X Center的防火墙内才是可访问的。这意味着管理员界面必须运行在一个不同的Web应用中（尽管可能仍在同一台物理服务器上）。管理员界面将把中心放在Oracle数据库上，而该数据库将是它与其他界面进行通信的主要手段。由于X Center已经拥有一些内部的Oracle技能，所以管理员界面的开发可以被推迟到第二阶段（Phase 2）。

前提

现在，让我们来考虑一下以下前提：

- Internet用户将不被要求接受cookies，尽管预计大多数用户将会接受。
- 每个节目的座位将分成一种或多种座位类型，比如Premium Reserve。座位安排计划和座位类型的划分将与节目关联起来，不与个别演出关联起来。
- 同一场演出的同一种座位类别中所有座位将是同一种价格。但是，同一个节目的不同演出场次可能会有不同的价格结构。例如，白天举行的音乐会可能会比夜间的演出便宜，以吸引家庭观众。
- 用户将为一场指定演出请求一种选定类型的给定座位数量，并且将被禁止请求具体座位。用户将被禁止为座位分配提供“暗示”，比如“靠近A Reserve的前面并且在

右边而非左边”。

- 请求座位的用户将被假设成需要相邻座位，尽管他们将被提供现有的任何空座位，如果不能被分配足够的相邻座位。如果座位被一个过道隔开，它们不被看做相邻的。
- 一旦付款已经得到处理，Internet顾客将不能取消预订。他们必须给售票处打电话，然后该售票处将运用X Center关于取消和退款的政策，并在得到许可时使用售票处界面来取消预订。

范围限制

由于这个示例应用是作为一个例子，而不是作为一个商业应用，所以某些领域已经被忽略。不过，下面这些实质上是精致的改进，并且不影响系统体系结构：

- 将不为信用卡购买实现任何安全（在一个现实应用中会使用HTTP，但这方面的设置细节在J2EE服务器之间是不同的）。
- 一个现实系统会连接到一个外部付款处理系统来处理信用卡付款。在这个示例应用中，这由一个虚设构件代替，并且该构件模拟一个变化的期限，并随机地确定批准还是拒绝事务。
- 将不考虑特殊的座位要求（比如轮椅通道），尽管这在大多数售票系统中是一个重要的考虑事项。

交货进度表

X Center的管理层已经决定：拥有一个真正的联机售票解决方案是一项需要优先考虑的事情。因此，他们已经决定了一个交货进度表来保证该公共系统尽可能快地投入联机使用。这个进度表要求该应用分3个阶段交货：

- 第一阶段（Phase 1）：核心的Internet用户界面（和下一节中所描述的那样），以及售票处界面。
- 第二阶段（Phase 2）：管理员界面。在第一阶段中，将没有GUI可供管理员用户使用，他们将需要使用Oracle工具来处理数据库。在第二阶段开发期间的这些任务方面，开发资源将可用来支持管理员用户。
- 第三阶段（Phase 3）：更复杂的Internet界面。这将提供注册用户奖励访问，并添加国际化支持。

这个示例应用将只延伸到第一阶段。完整的核心Internet用户界面将被实现，并且是售票处界面（在本章中将得不到完整定义）的一部分。不过，该示例应用必须定义一个体系结构，以便为后面的阶段提供一个基础，因此它们的需求必须被同时考虑。

Internet用户界面

现在，让我们来看一看公共Internet用户界面。这个界面举例说明了构造J2EE Web应用方面的许多常见问题。

基本工作流程

虽然笔者在这里使用的是UML状态图表，但每个框所代表的不是一个状态，而是一个显示给用户的屏幕，并且该用户成功地选择了一场演出，并购买了许多张入场券。图5.1描述了一个成功地预订了座位的Internet用户。

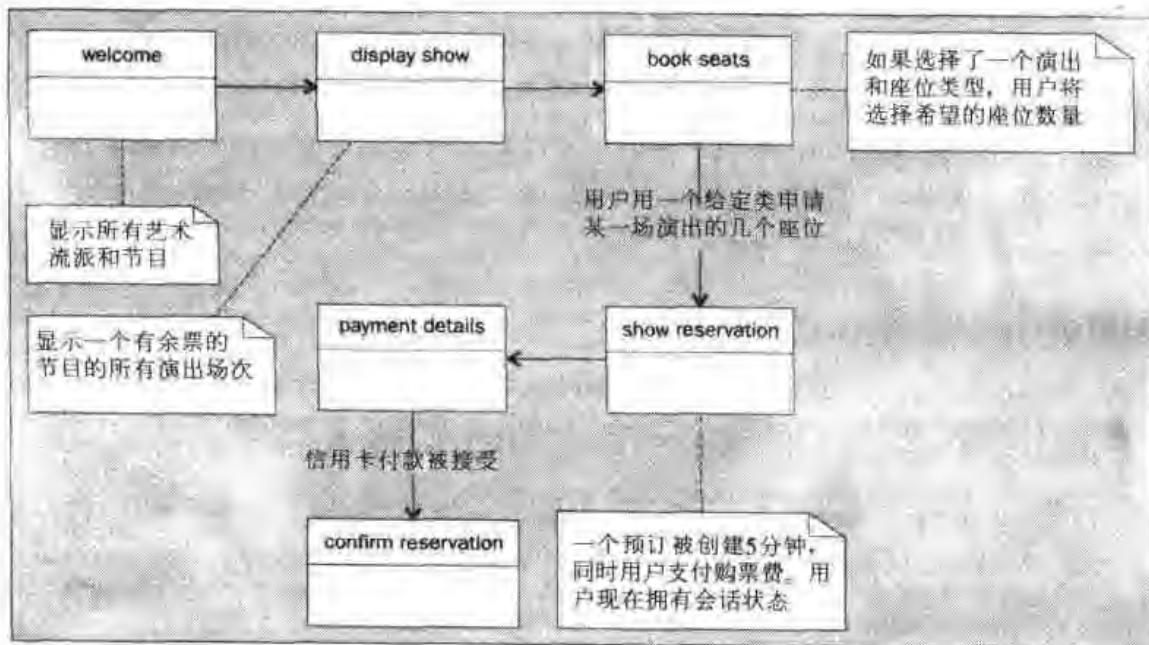


图5.1

用户一开始见到的是welcome屏幕，即一个含有X Center上演的全部流派（比如歌剧）和节目（比如Wagner的Tristan und Isolde）的清单（关于屏幕图，请参见下面的“应用屏幕”一节）。然后，用户可以继续到display show屏幕，该屏幕显示某一给定节目的所有演出日期，连同出售座位的类型与价格以及每种座位的可获得性。这个屏幕应该只显示每种座位是可获得的，还是已售出的，而不是显示剩余座位的数量。这既满足了大多数用户的要求，同时又避免了透露敏感的业务信息——例如，某一特定节目的入场券可能正销售很差。用户可以申请某一场特定演出的一种给定座位类型的座位数量，在这种情况下，这些座位被保留，以便为联机的信用卡付款预留一段时间。

该系统区分订票过程中的预订与确认/购买的概念。用户可以在一个时间期限（一般为5分钟）内预订一场演出的座位，并且该时间期限将被保存在系统配置中。预订防止这些座位在其他用户面前被显示为可获得。确认发生在用户提供了一个有效信用卡号的时候，此时一个预订变成一个实际的购买。预订被保存在数据库和该用户的会话状态中。预订必须：

- 发展到结果为确认的订票。
- 期满，如果该用户在给定时间内未能发展到购买。然后，这些座位将可供其他用户购买。但是，进行了预订的这个用户仍可以购买这些座位，而且只要这些座位在他（她）提交付款单时仍是空着的，就不会见到一条错误消息。

- 被清除，最终结果是除继续进行到购买之外的其他用户活动。例如，如果一名用户完成了预订，但后来又退回到display show屏幕，那么该系统假设他（她）不想继续购买这些座位，并且它们应该被恢复到完全可出售状态。

错误处理

如果出现了一个系统错误（比如J2EE服务器未能连接到数据库），应该给用户显示一个与当前品牌一致的internal error屏幕。这个屏幕应该建议用户稍后再试。如果出现了严重问题，应该通过电子邮件和日志文件记录通知支持人员。用户决不应该见到一个含有栈跟踪的屏幕，即“500 Internal Error”或其他技术或服务器特有的错误消息。请求一个无效URL的用户应该见到这样一个屏幕：它与当前品牌一致，并且告知被请求页面未找到或者含有一个指向welcome页面的链接。

应用屏幕

现在，让我们比较仔细地看一看公共Internet界面中的工作流程。图5.2是前一个图表的一个较详细版本，并描述了可供Internet用户使用的所有屏幕和这些屏幕之间的过渡过程。

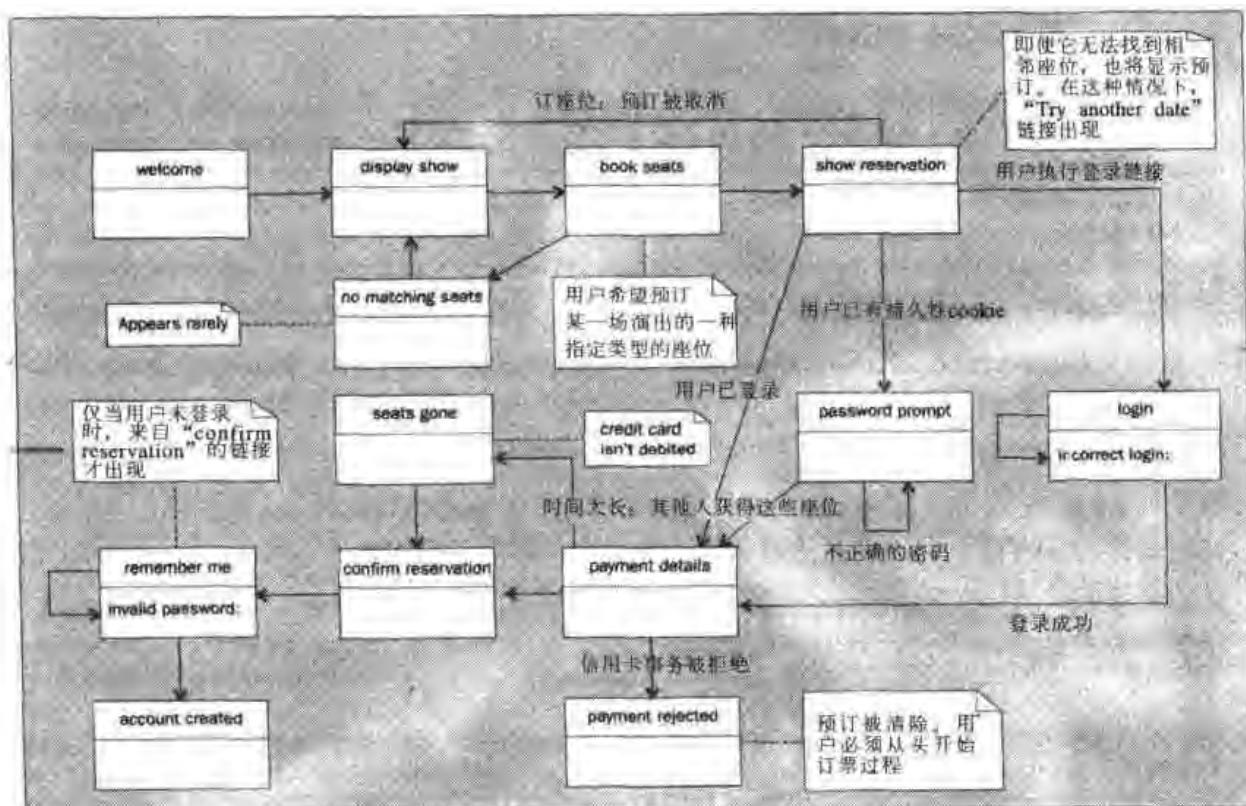


图5.2

附加的细节涉及到无法完成一次订票和可选的用户注册过程。用户在下列情况下将不能进展到预订或确认：

- 用户请求了display show屏幕，并在跟随那些链接之一到达book seats屏幕之前引起了一个时间很长的延迟。到他们前进到这个屏幕时，更多的座位已经售出，所申请类型的座位不再有售。
- 能供应给用户的座位不是相邻的。该用户不愿意接受这种座位，并顺着try another date链接回到display show屏幕。不必要的预订应该被清除，该用户必须重新开始订票过程。
- 用户做了预订，但未能在该预订的生存期内提交一个信用卡号。通常，这不成问题，除非那些座位已由另一个用户预订或购买。在这种情况下，向用户解释这个问题，以便让用户放心他们的信用卡没有被扣款，并提供一个指向display show屏幕的链接。
- 用户以一种有效格式输入一个信用卡号，但该事务被信用卡处理系统拒绝。在这种情况下，用户将被告知此次信用卡付款遭拒绝，并且该预订已经丢失。该预订被清除，用户必须重新开始预订过程。

该预订过程是任选的。它允许用户输入一个密码来创建一个账户，并且这个账户在该用户以后每次购票时将省去用户输入他们的付账地址和电子邮件地址（将来，其他奖励服务也可能被提供给注册用户）。用户不应该被迫通过注册来购买入场券，因为管理层觉得这可能会降低销售量。

预订将在一个还没有登录并且所提供的电子邮件地址没有与一个注册用户关联起来的用户已经进行了订票之后被提供。该用户的用户名将是他在订票过程中输入的电子邮件地址。当一个用户注册并且该应用检测到该用户的浏览器被配置成接受cookies时，该用户的用户名将被存储在一个持久性cookie中，并且该cookie能使该应用在用户以后访问时识别出他们的注册状态，并自动提示一个密码。这一提示将通过reservation和payment details屏幕之间的一个不同屏幕被引入到注册过程中。下列规则将应用：

- 如果该用户进入系统，付账地址和电子邮件地址（但不包括信用卡细节，该细节决不被保留）将在付款细节表单上得到预填充。
- 如果一个含有该用户的电子邮件地址的持久性cookie被查找到，但该用户没有注册过，则该用户将被提醒输入一个密码，然后再继续到付款细节表单。该用户可以选择跳过密码输入，在这种情况下，该持久性cookie将被删除，并且该用户将会看到一个空的付款细节表单。

如果该用户没有登录，并且没有一个持久性cookie，那么show reservation页面上应该有一个链接，以便该用户能够在见到payment details屏幕以前，通过输入用户名和密码这两者来尝试注册。如果注册成功，一个持久性cookie将得到创建。

跟在表后面的那些屏幕图给出了每个页面上所需要的所有控件和信息。但是，它们并没有完全定义该应用的表示。例如，这些屏幕可能会用做内容，其中被添加了一个自定义标题和副栏。可能会有一个以上这样的品牌标记：技术体系结构高性价比地允许这么做是一个业务需求。

对每个屏幕的讨论最终将得出一个总结，其中含有必须利用Web接口处理的一些常见问题，并且总结的格式如表5.1所示。

表5.1

屏幕名称	本章中所讨论和上述图中所给出的相同屏幕名称。该表描述一个请求的结果，这个请求成功时将会导致这个屏幕
用途	本页面做什么（比如“显示用户配置文件”）
替代屏幕	从一个请求中可能产生的若干替代页面，正常情况下该请求将产生本页面（比如从无效表单输入中产生的行为）
URL	本页面在应用内的URL，比如/welcome.html。需要注意的是，虽然选定扩展名是html，但所有页面都是动态的
显示的数据	显示在本页面上的数据，以及数据来自何处（数据库、用户会话等）
需要现有会话状态	是/否
对现有会话状态的影响	在一个针对本页面的请求之后，对现有会话状态发生什么（比如一次座位预订）？
刷新/重提交的结果	如果用户刷新本页面，会发生什么？
后退按钮的结果	如果用户单击后退按钮，会发生什么？
必需的参数	列举出所需参数的名称、类型和用途
无效参数的结果	如果期望的请求参数缺少或无效，会发生什么？

作为测试该Web界面的一个基础，这个总结将是非常有价值的。

Welcome屏幕

本页面（如图5.3所示）提供一个索引，其中含有目前现有的所有节目和演出。它还可能包含关于X Center的静态信息和链接。目标是最大限度地减少用户执行常见任务所必须遍历的页面数量，因此本页面允许直接前进到关于每个节目的信息，无需用户浏览中间页面。

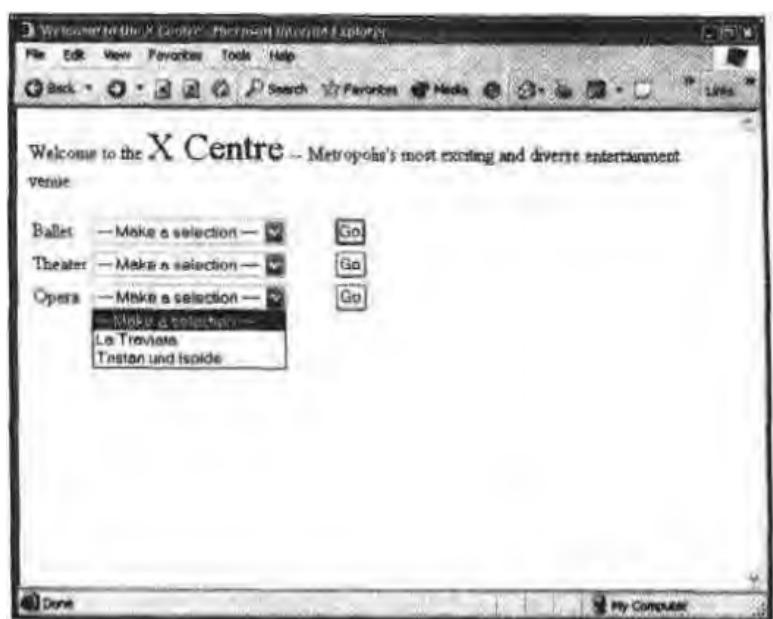


图5.3

本页面必须是动态的：节目种类和节目决不应该硬编码到HTML中。节目种类的数量在不久的将来会非常大，以致为每个种类都安排各自的下拉菜单可能会变得不切实际。但是，系统体系结构必须支持其他界面，比如通过为每个种类都安排各自的页面来拆分本页面，如果节目种类和节目的数量有明显增加。用户对系统的第一印象是至关重要的，因此本页面必须装入很快，如表5.2所示。

表5.2

屏幕名称	Welcome
用途	显示所有流派和节目。如果某一流派中目前没有节目，该流派就不应该显示出来
替代屏幕	无
URL	/welcome.html。这应该是默认页面，如果用户没有指定该应用内的一个URL
显示的数据	从数据库中列举出的动态节目种类和节目；静态信息由HTML表示（比如X Center的一张照片）决定
需要现有会话状态	否
对现有会话状态的影响	无
刷新/重提交的结果	页面应该被标记为可以由浏览器高速缓存1分钟。由Web服务器接收到的请求应该导致一个页面的生成
后退按钮的结果	不相关。用户没有该应用的会话状态
必需的参数	无
无效参数的结果	传递给本页面的参数将被忽视

Display Show屏幕

这个屏幕（如图5.4所示）将允许用户首先选择他们感兴趣的某一节目的某一场演出，然后进行订票。它还将提供关于该节目的信息，比如将演出的作品的历史和角色信息。预计有些不愿意联机使用其信用卡的用户将只使用该Internet界面到这种程度，然后再打电话到售票处购买某一给定日期的入场券；这也是在本页面上提供综合信息的另外一个原因。

在第一阶段中，没有必要提供编码页面支持（尽管以后这是一个潜在的改进）；用户可能需要通过滚动来观看所有演出信息。对于每场演出，通常有一个从每种座位到订票页面的超级链接。为了指出给定演出的给定等级座位已经售完，用一段说明文字来代替一个超级链接。

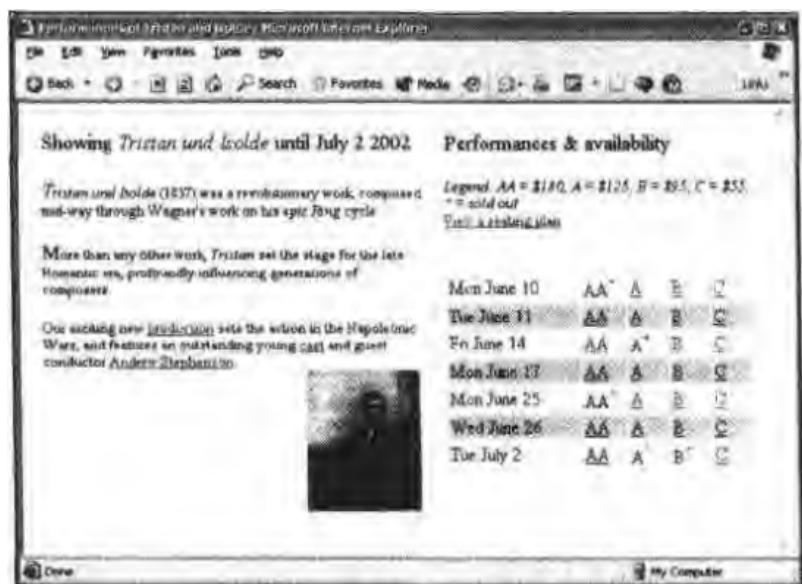


图5.4

表5.3

屏幕名称	Display Show
用途	显示关于该节目的信息。显示一个含有该节目的所有演出的列表，连同每种座位类型的价格
可选屏幕	无
URL	/show.html
显示的数据	动态的演出与座位余票信息，来自数据库。 与用户感兴趣的节目有关的HTML格式的信息。这必须是内容合法的HTML文档，并且可能含有内部和外部超级链接。这个文档将由拥有HTML技能的非程序员来维护。这个文档在Web应用中的URL应该与数据库中的节目关联起来。如果数据库中没有指定的URL，应该显示演出的座位安排计划来代替节目信息。
需要现有会话状态	否
对现有会话状态的影响	任何已有的预订将被清除，并且不通知用户。但是，任何已有的用户会话将不会被破坏
对刷新/重提交的影响	页面应该由服务器重新生成
后退按钮的结果	不相关。用户没有该应用的会话状态
必需的参数	id（整数）：节目的唯一ID
无效参数的结果	除节目ID之外的其他参数应该被忽视。如果没有提供节目ID，或者节目ID是非数字的或超出范围的，将显示一个通用的invalid request页面，其中含有一个返回到欢迎屏幕的链接。这在正常操作中不会发生，因为作为单击欢迎页面上的那些Go按钮之一的结果，用户只应该到达本页面

Book Seats屏幕

要想到达这个屏幕（如图5.5所示），用户需要选择一个演出日期和入场券类型：通过跟随一个座位类型链接——在图5.4中，是右表中的AA、A、B或C链接之一。



图5.5

表5.4

屏幕名称	Book Seats
用途	提供一个表单，该表单能让用户为他们所选定的演出申请某一给定座位类型的座位数量
替代屏幕	如果没有这个演出的余票可供出售，一个简单的屏幕应该说明这种类型的人场券已经售完，并提供一个返回到display show屏幕的链接
URL	/bookseats.html
显示的数据	<p>该节目的名称和这场演出的日期。 入场券类型的名称（比如Premium Reserve）及其单价。 一个含有选定入场券的默认数量的下拉菜单。默认值可以由管理员来设置，并且应该在该应用的factory settings中被配置成6个座位。最初显示在该下拉菜单中的值应该是默认值和被申请座位类型所剩余的人场券数量中的较小者（例如，如果默认值已经被设置成8，而且只剩余7个座位，选定值将是7）。下拉菜单中显示的值范围应该是1到剩余入场券数量和管理员设置的最大默认值中的较小者。同一个事务中购买的最大默认座位数量的典型值可能是12，目的是为了防止票贩子</p>
需要现有会话状态	否
对现有会话状态的影响	任何预订都将被清除。但是，用户会话将不被破坏

(续表)

刷新/重提交的影响	页面应该由服务器重新生成。重要的是，本页面总是提供关于座位可供应性的最新信息（具体数字决不应该出现本页面中，而应显示在下拉菜单内的那些值中），所以高速缓存是绝对不允许的
后退按钮的结果	不相关。用户没有该应用的会话状态
必需的参数	<code>id</code> （整数）：该节目的惟一ID <code>type</code> （整数）：座位类型的惟一ID
无效参数的影响	除节目ID和座位类型之外的其他参数应该被忽视。如果这两个参数中的任何一个参数是缺少的，非数字的，或者超出范围的，将显示一个通用的invalid request页面，其中含有一个返回到welcome页面的链接

Show Reservation屏幕

这个屏幕（如图5.6所示）是由被申请座位数量的成功预订产生出来的。这是导致会话状态创建的第一个操作。

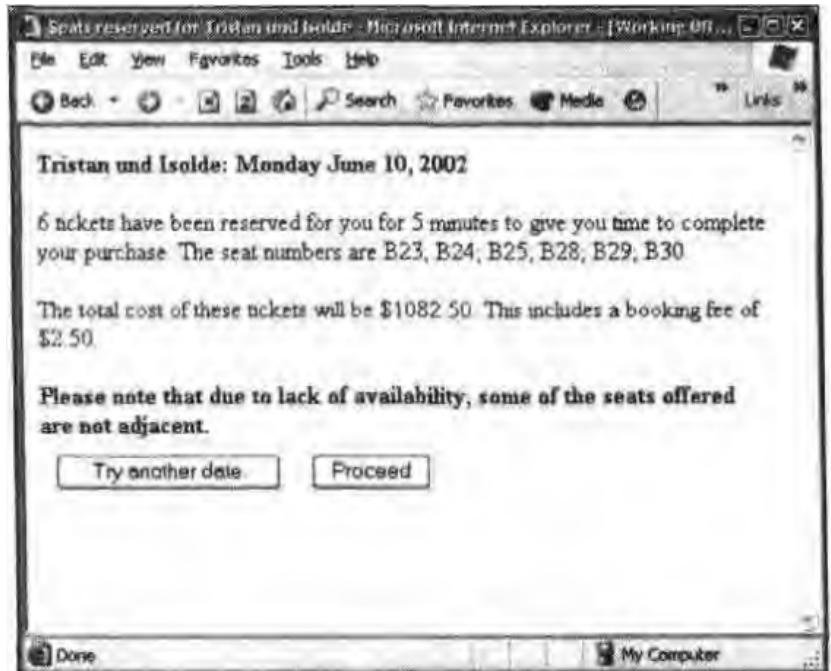


图5.6

表5.5

屏幕名称	Show Reservation
用途	通知用户他（她）已经成功地预订了许多座位
可选屏幕	如果无座位是可供应的（由于该用户在提交表单时耽搁了，并且这场演出已经客满），显示一个屏幕来解释这个问题以及提醒用户试一试另一个日期。在这种情况下，没有任何预订将得到创建，而且用户将没有会话状态

(续表)

URL	/reservation.html
显示的数据	<p>该节目的名称和选定演出的日期。</p> <p>申请的人场券的数量和名称。</p> <p>入场券的总价格。</p> <p>那些座位是否相邻。如果那些座位是不相邻的，应该向用户做出解释，并提供一个链接来允许用户试一试另一个演出日期。座位安排计划将使该用户能够计算出这些座位挨得有多近（例如，B25到B28可能是一个可接受的间距；A13到Y40可能不是）。</p> <p>座位安排计划（包括所有座位等级）使该用户能够计算出那些座位的位置。这个座位安排计划在第一阶段中将不是动态的：没有必要突出显示预留座位的位置。</p> <p>如果该用户没有一个持久性cookie，并且没有注册过，显示Remember me链接。这个链接将使该用户能够在进展到payment屏幕之前进行注册。</p>
需要现有会话状态	否。本页面的生成将创建会话状态，如果会话状态不存在
对现有会话状态的影响	如果该用户有一个已存在的预订，该预订在处理新座位请求之前被清除。让一个用户在他（她）的会话状态或数据库中拥有一个以上的同时预订应该是不可能的。
刷新/重提交的影响	在重新提交时，该应用应该留心该用户的会话中所保存的预订匹配于申请。应该返回一个相同屏幕（除了应该显示该预订的期满期限而非预订时间长度之外），同时不在数据库中做任何修改。该预订的生存期将不被延长。
后退按钮的结果	
必需的参数	<p><code>id</code>（整数）：该节目的唯一ID。</p> <p><code>type</code>（整数）：我们希望预订的座位类型。</p> <p><code>count</code>（整数）：申请的座位数量</p>
无效参数的影响	除上述参数之外的其他参数应该被忽视。将显示一个通用的invalid request页面，其中包含一个返回到welcome页面的链接，如果所需参数的任何一个缺少或无效，因为在正常操作中这不应该发生。

Payment Details屏幕

请注意，密码提示或注册页面可能会在本页面（如图5.7所示）之前出现，并且可能会为已注册用户预先填写好一些地址和电子邮件字段。信用卡细节将总是空着的，因为它们在任何情况下都不会被保留。

表5.6

屏幕名称	Payment Details
用途	允许一个已经成功地预订了座位的用户付票款，以及提供一个邮寄入场券的通信地址

(续表)

替代屏幕	没有注册过但有一个持久性cookie的用户将被引导到密码提示页面。 没有注册过但已从show reservation页面中选择了“注册”链接的用户将被引导到注册页面
URL	/payment.html
显示的数据	节目和演出的名称。 被预订座位的数量。 座位的价格。 用户配置文件数据，如果该用户已经注册过，否则所有字段将是空着的
需要现有会话状态	是。本页面只有当该用户在其会话中有一个预订时才应该显示出来。无会话或无预订地请求本页面应该产生一个通用的invalid request页面
对现有会话状态的影响	无
刷新/重提交的影响	页面应该由服务器来重新生成。如果该用户的会话中所保存的预订已经期满，应该给该用户显示一个页面来解释这个问题，以及提供一个返回到display show页面的链接
后退按钮的结果	
必需的参数	无。本页面依赖于会话状态，不依赖于请求参数
无效参数的影响	所有参数都将被忽视

图5.7

当一个无效提交发生时，本页面将被重新显示出来，其中含有突出显示的缺少或无效字段值，以及针对每个字段的一条解释问题的错误信息。突出显示错误的确切意思是指一个以后再决定的细节。但是，系统应该支持不同风格的突出显示。

除了2行地址之外，所有字段都是必须填写的。在提交本页面上的此表单时，下列验证规则将被应用：

- 信用卡是16位数字。
- 信用卡截止日期是4位数字。
- 邮政编码是一个有效的英式邮政编码格式（例如SE10 9AH）。但是，将不进行任何尝试来核实这是一个真实邮政编码（看起来像英式邮政编码的提交在语法上可能是正确的，但在语义上可能是无意义的）。正确输入其地址是用户的责任。在后续发布中，其他国家也必须得到支持，而且邮政编码验证必须基于从一个下拉菜单中选取的国家来加以实现。
- 电子邮件地址必须通过简单的验证检查（即它看起来像一个电子邮件地址，其中正确的部分含有@符号和句点）。但是，该事务仍能成功，即便该电子邮件地址没有证明是工作的；入场券购买在一个测试电子邮件发送期间不应该受阻。

将不进行任何尝试来验证信用卡号的检验和。这是外部信用卡处理系统要做的一件事情。

Confirm Reservation屏幕

这个屏幕（如图5.8所示）确认该用户的入场券购买，并提供一个电子收据。

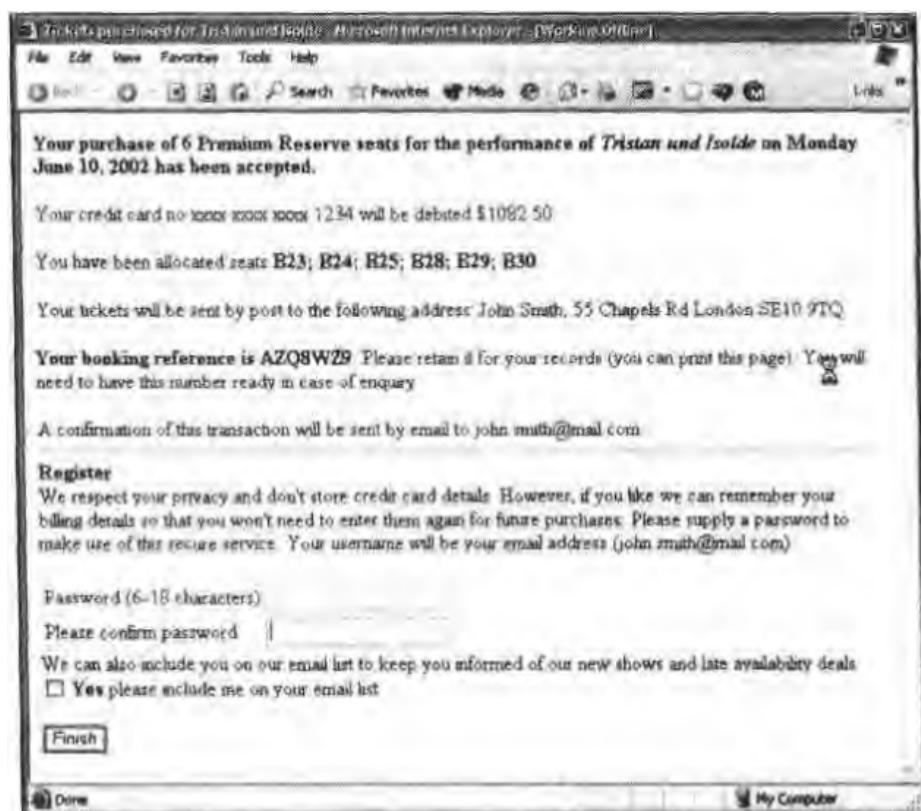


图5.8

表5.7

屏幕名称	Confirm Reservation
用途	确认该用户的信用卡付款。提供一个参考号码。允许该用户注册，如果该用户还没有注册（一个已注册用户将看不到直线下方的内容）
可选屏幕	<p>如果信用卡号的格式是无效的，或者必须填写的字段缺少或无效，付款屏幕重新显示出来，并填有那些遭拒绝的值。</p> <p>如果信用卡号的格式是有效的，但付款被信用卡处理系统拒绝，该用户应该被引导到payment rejected屏幕，并且该预订被清除。这被作为预订过程的结束来对待。使用一个可疑信用卡号的企图应该被明确地记录到日志中，供管理员查看。</p> <p>如果该用户的预订已经期满，但那些入场券仍是可供的，付款将像平常一样得到处理。如果入场券已经售给另外的用户，该用户将被引导到seats gone页面，并且这个期满的预订将从用户会话中被清除。该用户的信用卡将不被扣款</p>
URL	/confirmation.html
显示的数据	已预订的座位，订票参考，应支付的价格，用户的电子邮件地址（该电子邮件地址将不允许修改，如果该用户已经注册）。只有信用卡号的最后4位数字应该显示出来。为了安全起见，信用卡号决不应该被保存在用户会话或数据库中
需要现有会话状态	是。一个座位预订必须能在会话中找到，而且价格必须等于已提供的付款
对现有会话状态的影响	<p>该预订将从会话对象中被消除，因为它将已被转换成数据库中的一个已确认的订票。用户配置文件对象将仍被保留在会话中，并且该会话将不被破坏。一个已注册用户将保持已注册。</p> <p>如果已注册用户修改了他们的配置文件（两个地址之一），应该用新信息自动更新数据库</p>
刷新/重提交的影响	重新提交必须被阻止，以避免从用户的信用卡中支取两次款的任何风险
后退按钮的结果	这个表单的重新提交将不可能的，所以后退按钮应该决不会引起问题
必需的参数	讨论付款屏幕时所描述的参数
无效参数的影响	参见付款屏幕的讨论

售票处用户界面

售票处界面应该能够提供如下所示的查询和操作：

- 对于一个给定节目，显示某一给定日期的每种座位类型的可供应的总座位数量，以及最长5个的相邻座位（例如，这将使一个售票处用户能够告诉一位访客他们无法有18个相邻座位，但可以有16个和2个为一组或者10个和8个为一组的相邻座位）。
- 对于一个给定节目，显示某一给定座位类型的某一给定数量的可供应座位（相邻或不相邻）的第一个日期。

- 取消一个订票，假设给定订票参考（售票处用户将首先被允许查看订票数据）。

售票处界面的欢迎屏幕（在强制性注册之后）应该提供对所有常见查询的立即访问。

售票处界面中的每个屏幕应该突出显示它被再现的时间，而且售票处界面不必实现座位可供应性的高速缓存：售票处职员必须始终有权访问最新信息。售票处界面中与某一界面或演出有关的每个屏幕必须显示相关的座位安排计划。对于这个示例应用，只有售票处接口的上述那些特性才需要实现。

非功能性需求

平常活动是相当普通的。但是，在高峰期（比如在由一名大牌明星出演的一个新演出列入节目单时），该站点可能会经历用户活动的最高峰期。这样的活动通常将集中在一个单独的节目上，因而意味着该节目页面的性能需求几乎和欢迎页面一样。表5.8给出了公共Internet界面中性能最关键的那些页面的最小性能需求。

表5.8

页面	每秒钟的点击次数	再现页面的时间（秒） (忽略网络传输)	通过56K调制解调器返回页面的时间 (秒)
Welcome	2	1	3
Display Show	2	2	5
Book Seats	2	3	5
Show Reservation	1	3	5
Payment Details	1	3	5
Process payment details (性能将部分依赖于外部信用卡处理系统的速度)	1	30	30

这些只是满足中等业务需求的最小指标。最终目标应该是单独使用一台Intel型服务器运行该应用服务器来远远超过这些指标，进而展示该系统应付Z集团内的较广泛使用的潜力（数据库将单独运行在一台计算机上）。

需要特别注意的是并发访问下的性能特征，尤其需要注意下列情景：

- 对于一个节目甚至一场演出等繁重活动。这是一个现实的可能性——例如，如果一位电影明星在放映期间的某一个晚上出现在放映现场。
- 在一场演出几乎客满时支持对座位的并发申请。这可能会导致多个用户试图预订相同的小座位组。这种情景不严重损害总体性能是至关重要的，而且预订数据库不会遭到破坏也是十分重要的。
- 公共Internet界面上的严重负载对售票处界面的影响。售票处职员能够不受缓慢的响应时间影响而专心工作是十分重要的。

如果实现足够性能是必不可少的，下列折衷是可接受的：

- 由管理员添加的一个新节目或演出可能会占用多达1分钟时间才显示在Web界面上。不过，该系统绝不可以要求一次重启。
- 某一指定座位等级的可供应性或不可供应性（和display show页面上显示给公共Internet用户的一样）可能会占用多达1分钟时间才显示完，尽管应该尽可能地避免任何时间延迟。

由于取消是十分少见的，所以一个用户因为所需要的等级已被标记为售完而未能预订到座位，但在该用户请求display show屏幕的1分钟时间内可供应性事实上又重新出现的这种极小可能性为了性能的利益被认为是可接受的。这场演出在数据得到高速缓存之后可能已经售完的这种较大可能性不被看做一个严重的问题：不过要知道，该用户在查看某一给定演出日期或座位类型的链接之前可能已经打过电话或与朋友讨论过那些演出日期。这样的延迟通常将长于1分钟。

硬件与软件需求

目标应用服务器将是JBoss 3.0.0，并且与Jetty小服务程序引擎集成在一起。管理层渴望最大限度地降低许可成本，尤其是因为该应用可能被部署在许多站点上。将来，该应用可能需要被移植到另一个应用服务器，所以设计必须不依赖于JBoss。如果这样的移植是必需的，那么将需要给任何必需的代码修改和新平台上的测试安排少量的预算。如果可能，该应用应该是可验证为符合J2EE的，以便让管理层相信可移植性是可实现的。

X Center目前没有它自己的面向Web的硬件。根据性能分析，将需要购买适当的硬件；管理层希望通过最大限度地发挥每一硬件成本的性能来最大限度地降低成本。

该应用应该支持聚类，如果聚类随着要求的增加而变得必不可少，但管理层预计已得到描述的那些性能指标在单个服务器上会被超过。任一服务器聚类中的所有机器都将在同一个局域网内。

数据库将是Oracle 8.1.7i，因为X Center内部已经拥有一个Oracle许可和Oracle专门技能。在可预见到的未来时间内，不存在迁移到另一个数据库的可能性。Oracle数据库已经运行在一台Sun服务器上，并且这种情况将不会改变，无论为Web服务器选定的硬件是什么。数据库服务器和应用服务器将运行在同一个局域网上，所以可以认为有快速连通性。

不存在与要使用的技术有关的其他限制。管理层已经做出了一个采用J2EE的战略性决策，因为他们认为J2EE提供与Z集团的其他场所正使用的遗留订票系统成功地集成的最佳机会，而对应该怎样使用J2EE又没有任何偏见。是否使用像EJB、JSP和XSLT那样的技术是一个技术决策。由于该项目将涉及到建立一个团队，所以没有与现有J2EE专门技能（比如“丰富的小服务程序经验，无EJB经验”）有关的约束需要加以考虑。

X Center的许多职员都拥有HTML技能，但都不是程序员。因此，重要的是HTML内容和站点表示会得到尽可能大的控制，同时又没有深入了解J2EE技术的需要。

小结

在本章中，我们已经了解了示例应用的需求。这是最常见的一种J2EE应用——使用关系数据库的Web应用。

本书剩余篇幅中的讨论将利用实现这个应用的过程来说明已讨论的那些J2EE设计与实现概念。

在下一章中，我们将了解可用在J2EE应用中的数据存取选择，并为示例应用定义一个高层的体系结构。

在第7章到第9章中，我们将了解一个通用的应用基础结构怎么才能用来解决许多常见问题，以及它怎样简化应用特有的代码。

在第12章和第13章中，我们将了解如何实现J2EE应用中的Web界面，进而使用来自示例应用的预订使用案例来说明如何使用MVC体系结构模式。

在第14章中，我们将了解J2EE应用部署，进而描述怎样封装示例应用，以及怎样把它部署在JBoss 3.0.0上。

在第15章中，我们将了解怎样实现上面所标识出的那些性能指标。

第6章 应用J2EE技术

本章将较深入地讨论开发J2EE体系结构所涉及到一些关键决策，尤其是第1章中所讨论的那些决策。我们将讨论的内容如下所示：

- **在集中式与分布式体系结构之间选择**

这是一个对性能、易实现性、可缩放性、坚固性以及一个应用能够支持的客户类型都有影响的特别重要的决策。这个决策应该在设计过程中的早期做出，因为它对许多其他设计决策都有影响。

- **选择是否以及怎样使用EJB**

在第1章中已经见过，EJB是一项强有力的技术，能帮助我们解决一些复杂的问题，但这项技术也可能会引入不必要的复杂性，如果我们不需要它的全部能力。在本章中，我们将比较仔细地看一看何时以及怎样使用EJB来享有它的好处而又避开它的缺陷。

- **选择何时使用消息传递**

有时，业务需求要求异步应用行为或使用消息传递来与其他应用进行通信。在其他情况下，我们可能会选择在一个应用内使用消息传递。我们将考虑把EJB容器服务用做JMS消息传递，以及怎样决定何时使用异步消息传递作为一种实现策略。

- **身份验证与授权问题**

尤其是这两个问题与Web应用有怎样的关系。

- **在J2EE应用中使用XML**

XML是一项核心J2EE技术，而且XML在Web应用和企业中正变得越来越重要。我们将了解XML怎么才能J2EE应用中以及在XML的使用可能会证明起相反作用的领域内得到有效使用。

- **高速缓存问题**

大多数应用，尤其是Web应用，都需要高速缓存一些数据来最大限度地降低查询持久性数据源的代价。我们将考虑实现高速缓存的潜在好处和挑战。

在讨论了上述每个问题之后，我们将为本书的示例应用做出相关的决策。虽然示例应用只是一个相当简单的J2EE应用，但这个应用能够用来突出许多常见问题，并演示现实项目中所要求的那些折衷方案。

在本章中，我们不打算谈论数据存取。这个重要的题目将在第7章至第9章中讨论。我们也几乎不关注Web层的实现。这个题目将在第12章和第13章中详细讨论。

分布式体系结构何时合适

或许，最重要的体系结构决策是此应用该不该是分布式的，或者所有应用构件该不该被布置到运行此应用的每个服务器上。

在有些应用中，业务需求明确要求一个分布式体系结构。但是，我们时常有选择余地（比如在许多Web应用中）。

笔者相信，J2EE开发人员常常主张使用分布式体系结构，从不对这个重要的选择进行充分的考虑。这是代价可能会很高的一个失误，因为我们不应该没有充分理由就贸然采用分布式体系结构。在第1章中，我们已经考虑过分布式应用在复杂性和性能方面的缺陷。

现在，让我们来快速回顾一下我们为什么可能需要实现分布式体系结构的主要理由：

- 为了支持像Swing应用那样的J2EE技术客户软件，以便给它们提供一个中间层。一个J2EE Web容器是为了基于浏览器的客户软件才提供一个中间层；因此，这个理由不适用于仅有一个Web界面的应用。
- 当此应用必须与分布于整个企业内的资源集成时。这样一个应用可能会涉及到几个EJB容器，其中每个容器都访问一个企业资源，比如一个数据库或在网络上的其他地方不可获得的遗留应用。
- 为了获得对每个应用构件被部署在何处的控制权，以便改进可缩放性和可靠性。

在极少的情况下，我们也可能会选择让一个应用变为分布式的，以便在Web容器与业务对象之间引进一个额外的防火墙。

如果第一个或第二个要求确实存在，一个以具有远程接口的EJB为基础的分布式体系结构是最理想（而又最简单）的解决方案。

和第三个要求相关的各种问题要复杂得多，而且很少是轮廓清晰的。在后面的一节中，我们将比较仔细地看一看它们。本讨论假设我们正在考虑Web应用（实践中最常见的应用类型），并且RMI访问不是一个决定性的考虑因素。

分布式应用与可缩放性

分布式体系结构就一定比其内所有构件都被布置在每个服务器上的体系结构更具有可缩放性吗？J2EE设计师和开发人员往往认为分布式应用提供无敌的可缩放性。但是，这种想法是有问题的。

单JVM解决方案由于没有远程调用开销而比分布式应用提供更高的性能，从而意味着可以在同等硬件上实现更高的性能。另外，单JVM解决方案还可以被成功地聚类。企业质量的J2EE Web容器也提供聚类功能度；这不是EJB容器的专有领地。

作为一种替代方法，进入的HTTP请求可以被路由给一个聚类中的一个服务器；通过提供给J2EE服务器的负载平衡基础结构，或者通过像Cisco Load Balancer那样的硬件设备（这看起来像是一个单独的IP地址，但事实上是把请求路由给它后面的任何一个HTTP服务器；它为客户软件将需要返回到相同服务器以保留会话状态的情况既提供循环复用，又提供“粘性”路由）。使用硬件负载平衡的优点是它的工作方式和任何J2EE服务器的都相同。

通过添加一个远程业务对象层（比如一个EJB层），我们未必就使Web应用变得更可缩放。不需要服务器端状态的Web应用可以线性地（而且几乎无限地）缩放，而不管它是否使用远程调用。当服务器端状态是必需的时，可缩放性将是有限的，而不管我们是否采用了一个分布式模型。

如果我们把状态保存在Web层中（即HttpSession对象中），Web容器的聚类支持的质量

(状态复制与路由) 将决定可缩放性。如果我们把状态保存在EJB容器中(即有状态会话组件中), EJB容器将会有一项同样困难的任务。EJB不是一颗能让状态复制的那些问题消失得无影无踪的魔术子弹。再加上我们通常也需要把状态保存在Web层中, 除非我们采用把有状态会话组件柄串行化成一个cookie或隐式窗体字段的这种不常用手段。

只有当下列条件中的一个或多个有效时, 分布式J2EE应用(其内的业务对象将被实现为带远程接口的EJB) 才会内在地比单JVM解决方案更有可缩放性:

- **被远程访问的业务对象是无状态的, 无论Web层是否维持状态**

在J2EE中, 这意味着所有业务逻辑都将被无状态会话组件暴露出来。幸运的是, 这种情况常常能像我们在第10章中将要看到的那样被调整。下面, 我们将进一步讨论这种类型体系结构。

- **在Web层与业务对象之间的负载方面存在明显的不一致**

除非这个条件有效, 否则我们只能运行更综合性的Web容器和业务对象容器。除非该应用的某一特定层是一个瓶颈, 否则只在这个层而不是在整个应用中添加新硬件是没有意义的。

Web层与业务对象之间存在负载明显不一致的情景包括下列这些:

- **部分或全部业务对象都消费无法被分配给Web层中每个服务器的资源**

在这种不常见的情况下, 布置Web和业务对象意味着我们遇到了客户-服务器应用的可缩放性的限制。如果该应用在Web层上有大量负载, 而在业务对象上有相当少的负载, 这种情况可能就会出现。在这种情况下, 允许每个Web容器分配它自己的数据库连接池可能会导致数据库需要支持过多的连接。

- **业务对象比Web层做更多的工作**

在这种情况下, 我们将需要运行更多的EJB容器。

具有无状态业务对象的模型是高可缩放的, 因为我们可以根据需要添加尽可能多的EJB容器, 同时又不会增加Web容器的数量, 或增加Web层中可能需要的任何状态复制的系统开销。但是, 只有在Web层与业务对象之间的负载方面存在不一致的情况下, 这种方法才会比其内所有构件都被布置在每个服务器上的Web应用具有更高的可缩放性。

分布式应用未必就比其内所有构件都被布置在每个服务器上的Web应用更具有可缩放性。只有含有一个有状态Web层和无状态业务对象的分布式体系结构才会比其内所有构件都被布置在每个服务器上的Web应用更具有可缩放性。

分布式应用与可靠性

分布式体系结构一定就比其内所有构件都被布置在一个单独服务器上的Web应用更坚固吗? 同样, J2EE设计师和开发人员往往认为是这样的。但是, 这种想法也是有问题的。任何一个有状态应用都会遇到服务器类同性(server affinity)的问题, 在这个问题中, 用户被与某个特定服务器关联起来——对坚固性的一种特殊威胁。把一个应用分解成分布单元不会使这个问题消失。

如果我们有一个无状态Web层, 硬件路由会提供杰出的正常工作时间。像Cisco Load

Balancer那样的设备能够检测到它们身后的服务器何时性能下降，并停止发送请求给它们。应用的用户将会受到保护，使之免于遭受这样的失败。

如果我们维持服务器端状态，它将需要被Web层引用，然后Web层将变成有状态的。如果我们有一个有状态的Web层，该应用无论是分布式的还是集中式的，都只能同Web层聚类技术一样可靠。如果一个用户被与一个Web层服务器关联起来，并且该服务器又性能下降，那么另一个服务器上的业务对象是否仍在运行都无关紧要。该用户将会遇到问题，除非会话数据已经成功地得到复制。通过使用有状态会话组件，我们只是把这个问题转移给了EJB容器，这样常常会把EJB层复制的问题增加到Web层复制的问题上，从而引入了服务器类同性的潜在问题（正如我们在第10章中将要看到的，有状态会话组件在处理会话状态复制方面可能不如 HttpSession 对象那样坚固，或者说可能不比 HttpSession 对象更坚固）。

如果我们只有一个有状态Web层，而没有无状态业务对象（即J2EE应用中的无状态会话EJB），分布式体系结构可能会证明比集中式Web应用体系结构更坚固（并且更可缩放）。如果业务对象会潜在地引起严重的失败，把它们“排除在过程之外”可能会提供更大的弹性，尤其是当J2EE服务器提供无状态会话组件的复杂路由的时候。例如，WebLogic能够对运行同一个EJB的另一个服务器重新尝试无状态会话组件上的失败调用，如果该失败的方法被标记为幂等的（并且不引起更新）。

分布式体系结构未必就比集中式体系结构更坚固。使用无状态业务对象的体系结构是保证分布式体系结构比集中式体系结构更坚固（并且更可缩放）的惟一手段。

可缩放而又坚固的体系结构

一种体系结构的可缩放性和可靠性的关键决定因素是它需不需要服务器端状态，而不是它是不是分布式的。下列3种体系结构可证明是十分坚固而又可缩放的（以降序排列）：

- **无状态体系结构，无论是分布式的还是集中式的**

这种体系结构提供无敌的可缩放性、性能和可靠性。可缩放性可以通过硬件路由设备来实现，并且不依赖于复杂的应用服务器功能度。新的服务器可以被随便添加，同时又不会有额外的性能开销。J2EE应用以外的资源，比如数据库通常将会限制可缩放性。在Web应用中，有相当大的一部分应用能够被变成无状态的，特别是在有可能使用cookies的时候。

- **具有无状态业务对象的分布式体系结构**

如果Web容器不必维持会话状态，添加一个无状态的、过程外的业务对象层（比如远程EJB层）通常不会提高可缩放性，不管是损害性能并使设计和部署变复杂。例外是业务对象大量地拥有幂等方法并容易使它们运行在其上的那个服务器发生崩溃的这种有点人为的情况：正如我们已经见过的，至少WebLogic能够最大限度地提高可靠性，如果我们使用无状态会话组件。

但是，如果Web容器需要维持会话状态，那就是另外一回事了。通过从运行Web层的服务器中删除业务对象，我们或许能够改进Web吞吐量，同时又使会话状态复制的开销保持最小。通过添加运行无状态会话组件业务对象的更多EJB容器，我们可以提高

总吞吐量。增加规模时将会有一些开销，因为应用服务器将需要管理聚类中的路由，但是，由于不需要会话状态复制，所以这样的开销不会太大。

· 所有构件都被布置在每个服务器上的Web应用

设计完善的Web应用提供很高的性能和杰出的可缩放性与可靠性。只有在意外情况下，我们才需要根据可缩放性与可靠性来关注带无状态业务对象的分布式体系结构；正常情况下，集中式体系结构就能满足需要。

与带无状态业务对象的分布式体系结构相比，带有状态业务对象（比如有状态会话组件）的分布式体系结构可证明不如它们中的任何一种那样可缩放，部署起来更复杂，而且可操作性更差。

当我们需要维持会话状态时，带无状态业务对象的分布式体系结构在J2EE可缩放性与可靠性方面会提供最基本的保证，但代价是一些性能开销以及一定程度的、在大多数应用中显得没有必要的复杂性。

图6.1表举例说明了这样一种可缩放而又坚固的分布式体系结构。由于业务逻辑是使用带远程接口的无状态会话组件（SLSB）来实现的，所以运行比Web容器多的EJB容器是可能的。在这种体系结构中，添加一个EJB容器所增加的开销会少于添加一个Web容器所增加的开销，因为Web容器之间的会话状态复制是必需的。

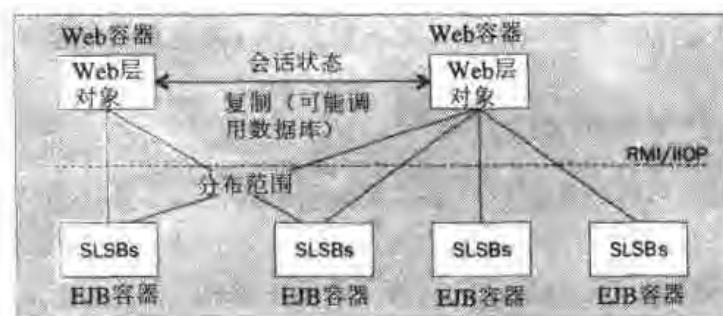


图6.1

需要重点注意的是，给一个聚类添加一个新Web容器的开销（在涉及到状态复制的地方）将会在服务器之间有所不同，而且将会随着所使用的复制策略而变化。笔者正假设使用内存中复制（in-memory replication），在这种复制方式中，被复制的会话数据在聚类中的服务器之间被交换，但从不被存储在一个数据库中（涉及到一个后备存储器的复制从理论上讲将几乎线性地增大，但将会强加一个很大的性能开销，而且可能会遇到问题，如果在会话数据已经得到存储之前，一个后续的复制平衡请求被接收到——一种实际的可能性，因为写一个持久性存储器可能会是一个速度很慢的操作；因此，它通常是不切实际的）。

在WebLogic之类的服务器中（这类服务器给每个请求使用一个单独的、随机选择的从属服务器（second server）作为会话状态被复制到的目的地），添加更多服务器方面的开销可能不会太大（代价是主从服务器都同时失败的可能性）。在Tomcat之类的服务器中（这类服务器中的会话状态被复制到一个聚类中的所有服务器上），对一个聚类中能够有效运转的服务器数量将有一个实际的限制，以便复制不会淹没局域网，并占用那些服务器的大部分工

作负荷（好处是总体弹性方面的潜在增加）。

根据笔者的经验，下列原则将帮助实现Web应用中的最高可缩放性与可靠性：

- **如果有可能，避免保持服务器端会话状态**

- **如果会话状态必须被保持，最大限度地减少被保持的最**

这将改进状态复制的性能。例如，共享数据不作为用户会话数据的一部分来保持。

- **如果会话状态必须被保持，把它保持在Web层中，不要保持在EJB层中**

正如我们在第10章中将要看到的，Web层中的HttpSession复制通常更有效，而且可能比有状态会话组件复制更可靠。

- **使用几个精细的会话对象，不要只使用一个单一会话对象**

宁愿保持多个较小的会话对象，而不要只保持单个会话对象，以便帮助Web容器有效地执行状态复制。然而，一个有效的Web容器应该只复制会话状态中的修改，因而意味着串行化和复制的开销将会得到极大降低。

示例应用的高层体系结构

示例应用的需求不要求业务对象的RMI访问。如果将来需要远程访问，更有可能要求一种基于XML的Web服务方式。因此，我们有关于是否使用一种分布式体系结构，以及是否需要带远程接口的EJB的选择。

示例应用确实需要服务器端会话状态。服务器端会话状态，比如该用户所保持的一个座位预订，对示例应用的工作流程是十分重要的，因此使用隐式窗体字段和此类东西来设法回避它将会证明是极其复杂的，而且是隐错的一个丰富来源。我们可以尝试把该用户的会话状态保持在数据库中，但这种做法将会影响性能，即使我们不是运行在一个聚类中（示例应用最初将运行在一个单独的服务器上），而且将会因为需要把一个会话密钥保持在一个cookie或隐式窗体字段中而使示例应用变得复杂。在示例应用中，不要求Internet用户接受cookie是一个业务需求（情况常常不是这样；如果cookie是允许的，就可能有服务器端会话状态的可替代物）。

我们应该保证服务器端状态的数量被保持为最小值，并保证放置在该用户的HttpSession中的任何对象是可有效串行化的。我们还应该避开把所有会话数据都保持在单个大对象中的诱惑，而是把个别对象保持在HttpSession中。例如，座位预订和用户环境配置对象能够被分别保持，以便在只有一个对象发生变化时，Web容器不必复制这两者的数据。我们将需要留心，保证不把共享数据包括在个别用户会话中。

例如，我们将包括一个用户已经预订的一场演出的数字主键，而不是包括表示这场演出本身的共享参考数据对象。由于演出对象是一个大对象图表的一部分，而这个图表又需要被全面地串行化，因此，如果示例应用始终运行在一个聚类中，这是一个可能很大的节约。用一个查找键来代替一个对象的这种替换是用来降低必须为每个用户所维持的会话状态量的一种常见而又十分有效的技巧。

在示例应用中，业务对象将证明它不大可能是一个瓶颈。我们可以简单地添加运行整个应用的新服务器（包括Web层）来增强。因此，使用一种分布式体系结构没有任何好处，即使该体系结构含有无状态业务对象。

因此，我们采用上面讨论的第三种体系结构：所有构件被布置在每个服务器上的Web应用。这种体系结构将不仅证明其可缩放和可靠得足以满足我们的需求，而且它还将最大限度地提供性能和最大限度地降低开销（那些业务需求所强调的是，性能应该按每一单元硬件被最大化）。

决定何时使用EJB

最重要的体系结构决策之一（对部署、测试、工具需求和支持都有影响）是用不用EJB。在本节中，我们将仔细分析一些决定性的考虑事项。

使用EJB实现一个分布式体系结构

EJB的使用可能由分布式与集中式体系结构之间的选择来决定。

如果我们需要一个基于RMI/HOP的分布式应用，EJB是帮助我们实现这种应用的理想J2EE技术。

通过使用带远程接口的EJB，我们让J2EE服务器去处理远程访问的细节。编写我们自己的RMI基础结构的可替代方案是十分差的：实现起来既非标准，又不必要的麻烦，而且维护起来可能十分困难。

事务管理

即使我们不需要EJB来实现一个分布式体系结构，但仍可能有使用EJB的理由。在第1章中，笔者已经解释过，Container-Managed Transactions（CMT）或许是使用EJB的最强大的理由。

现在，让我们大体上分析一下J2EE事务管理，以及使用EJB怎样才能帮助简化事务性应用。

J2EE应用中的事务管理

一个J2EE服务器必须提供下列事务服务：

- 它必须能够协调EIS层资源之间的事务。J2EE服务器中负责这项工作的部分叫做事务管理器。
- 它必须给EJB提供声明性事务管理的选择，从而免除了编写代码来明确地处理事务的要求。开发人员在这个模型中需要做出的唯一选择是回不回退一个事务。
- 它必须使标准J2EE Java Transaction API（JTA）和UserTransaction接口可以让希望管理事务本身的对象所使用。这些对象可能既包括EJB，又包括运行在一个Web容器中的对象。JTA是一个相当简单的事务管理API。

一个J2EE服务器必须能够处理由J2EE应用使用的事务性资源管理器能够理解的低级事务协议，比如X/Open XA。资源管理器（resource manager）就是一个EIS层资源，比如数据库。资源管理器控制着持久性状态，而且常常是事务性的。

J2EE服务器的事务管理器负责J2EE构件之间以及J2EE服务器到事务性资源管理器之间的事务传播（transaction propagation），无论这些构件被放置在一处，还是在各自的JVM中。例如，如果某个EJB上的一个方法被一个Web层对象调用，任何一个事务上下文都必须被传播。事务上下文被与一个JVM内的线程关联起来。一个容器必须能够使用IIOP的事务性能力来处理分布式应用的那些事务需求：通过在事务管理器之间传播事务上下文。由于IIOP是独立于供应商的，所以这意味着事务传播在全部J2EE服务器上都工作，即便在这些服务器由不同供应商供应的时候。

J2EE承诺的一部分是这种低级细节在开发人员面前被完全隐藏起来。开发人员可以假设他们的应用服务器所支持的任一事务性资源都将会得到正确管理，如果他们使用J2EE平台的标准特性：声明性事务管理（在EJB中有效）或使用了JTA的程序性事务管理（在所有服务器端J2EE构件中有效）。这是一个极大的好处，因为保证全部不同资源间的事务完整性是很复杂的。

例如，服务器应该支持两段提交（two-phase commit）来为涉及到多个资源管理器或多个EJB容器的事务实现正确的行为。两段提交将保证，即使在一个分布式场景中，支持同一个事务的所有资源要么提交，要么回退。这是通过分两个阶段处理一个分布式事务来实现的。第一个阶段把一个准备（prepare）请求发布给所有参与的资源管理器。

只有当每个资源管理器都已表示它准备好提交该组合操作中它的部分时，一个提交（commit）请求才将被发送给所有资源管理器。两段提交在某些情况下会失败。例如，如果一个资源管理器表示它准备好提交，但后来在一个提交请求发生时未能提交，那么完整性可能会受到损害。但是，两段提交被看做是分布式事务的一种非常可靠的策略。J2EE规范目前没有要求一个应用服务器提供完整的两段提交支持。但是，它做了未来要求这种支持的暗示（要想了解应用服务器所提供的两段提交支持的程度，请参考随应用服务器一起提供的技术文档）。

运行在一个J2EE服务器内但在EJB容器外部的任何一个类，都能执行程序性事务管理。这要求从JNDI获得UserTransaction接口或直接使用JTA。

任何一个J2EE构件自身都可能会管理一个事务性资源（比如RDBMS）上的事务，而这个事务性资源又使用了一个该资源特有的API（就RDBMS资源来说，该API是JDBC API或JDO API），而不是使用已集成的JTA。但是，这是一个很拙劣的选择，除非J2EE没有为上述资源提供事务性支持。这样的事务称做局部事务（local transaction）。它们之所以是“局部的”，那是因为它们从J2EE平台的完整事务管理中被分离了出来。

J2EE体系结构（它对局部事务毫无所知）无法自动回退局部事务，如果一个构件仅给一个J2EE事务做了回退标记。开发人员将需要在代码中处理回退和提交，从而导致过分复杂的代码，而且未能利用应用服务器的一个宝贵服务。局部事务必须使用资源特有的API来创建、提交和回退，从而使应用在它们处理事务的方法方面不一致。

在J2EE应用中，不要使用局部事务（比如JDBC事务），除非这个相关的事务性资源不被该服务器的完整事务管理所支持。要依靠EJB容器所提供的事务管理，或者使用JTA来管理事务。通过使用局部事务，你使该容器丧失了把事务管理集成到全部事务性资源当中的能力。

事务管理与EJB

EJB是配备用来管理事务的最佳J2EE构件。EJB从声明性和程序性事务管理中都能受益。在声明性模型中，开发人员不必编写任何JTA代码，从而保证了业务逻辑不会被低级事务管理操作搞得很复杂。一般说来，会话EJB用来管理事务，其中一个会话组件上的每个方法分隔一个事务。

对于选用Container-Managed Transactions (CMT) 的bean，容器在方法级别上声明性地设置事务性边界，从而消除了在用户应用中编写事务管理代码的需要。由于应用开发人员决定每个方法做什么，所以这巧妙地把事务管理集成到了Java语言中。

选用Bean-Managed Transactions (BMT)，开发人员在EJB本身内程序性地编写代码来开始、提交或回退事务。

Bean-Managed Transactions (BMT)

只有会话组件和消息驱动组件才可能会使用BMT。实体组件必须使用CMT。使用BMT是一个孤注一掷的选择。BMT还是CMT的选择在Bean级上做出，不是在方法级上做出；因此，如果一个bean中只有一个方法具有与众不同的事务性需求，并且这些需求似乎需要BMT的使用，那么这个bean中的所有方法都将被迫管理它们自己的事务（再加上这个设计来分解这个bean可能是更恰当的）。

BMT对无状态和有状态会话组件有一个不同的含义。

就无状态会话组件而言，容器必须检测这样一种情况：一个无状态bean方法在开始一个事务之后没有提交或回退它就已返回。由于一个无状态会话组件无法在方法调用之间为一个特定客户软件保持状态，所以每个方法调用都必须完成一套工作。因此，上述这种情况明显是一个程序设计错误。在检测到这种情况时，容器必须做出反应，采用的反应方式和它对待可从EJB方法中抛出的一个未捕捉异常的反应方式相同；然后，回退这个事务，并删除这个不愉快的EJB实例。由于相似的原因，这个相同的规则也适用于消息驱动的bean，这种bean在onMessage()方法返回之前必须结束任何一个打开的事务。

但是，就有状态会话组件而言，容器无权查看一个让一个事务作为一个程序设计错误打开着的方法。该无状态bean的几个方法调用构成一个事务，并需要另外的方法调用来自完成这套工作是有可能的。不过，这种情况的危险是显而易见的。客户软件可能会让用户在调用同一个事务中的那些方法之间等待。这将浪费宝贵的事务性资源，也可能会封锁资源管理器（比如数据库）的其他用户。更糟糕的是，客户软件可能会永不调用结束本事务的那个方法。

使用BMT的EJB没有必要在javax.ejb.EJBContext接口上使用setRollbackOnly()方法。这个方法的设计意图是能够为使用CMT的Bean提供容易的事务管理。

如果一个EJB正使用着BMT，它从EJB容器中获得的好处是极少的。这一相同的事务管理方法不用EJB也管用，使用EJB方面的惟一优点是任一事务在万一发生了一个未捕捉抛出时被自动回退。不过，一个终结块将在编写清楚的代码中处理这种情况；作为一种替代方法，一个框架类也可以用来处理万一出现错误时的清理。

Container-Managed Transactions (CMT)

使用CMT比使用BMT容易得多。我们不必在自己的EJB中编写任何事务管理代码。我们只需要：

- 定义方法责任，以便方法边界方面的事务界定是正确的。EJB方法——尤其是远程接口上的EJB方法常常实现一个完整的使用案例，所以这经常自然发生。
- 为相关ejb-jar.xml部署描述符中的每个方法指定正确的事务属性——EJB规范承认的事务属性有6个（我们将在第10章中详细讨论EJB事务属性）。
- 调用一个EJB API方法给一个事务做上回退标记，如果业务逻辑需要。

使用CMT的EJB通常也可以使用事务分离级别（transaction isolation level）的声明性控制，尽管EJB规范没有规定这应该被怎样实现。事务分离指的是并发事务间分离的级别。我们将在下一章中讨论这方面的内容。J2EE规范没有规定事务分离级别被怎样设置。一般说来，一个容器将使用一个附加的专有部署描述符来提供这些信息，比如WebLogic的weblogic-ejb-jar.xml EJB部署描述符。

使用CMT的EJB提供了一个简单的事务管理模型，并且这个模型无需编写任何事务管理代码就能满足大多数需求。使用CMT的声明性事务管理是使用EJB的一个重要理由。

使用BMT的EJB导致了EJB的复杂性与编写自定义事务管理代码的要求这两者的一种低劣价值的组合。只在极少数情况中才使用带有BMT的EJB：当CMT不能用来实现相同结果的时候。

示例应用中的事务管理

示例应用有几分事务性。因此，使用一个带有一个本地接口的EJB适合订票过程。这将使我们能够使用CMT并避免直接使用JTA。

EJB与授权

大多数J2EE构件都会对用户的安全主管（security principal）进行程序性访问，以确定该用户正处于哪些角色（Role）中。在Web容器中，这可以从javax.servlet.http.HttpServletRequest.getUserPrincipal()方法中获得；在EJB容器中，这可以从javax.ejb.EJBContext.getCallerPrincipal()方法中获得。javax.ejb.EJBContext.isCallerInRole()方法提供了一种便利方法用来检查一个用户是否处于一个指定角色中。只通过JMS接收到的异步调用不会与主管信息关联起来。

不过，EJB拥有用户主管的声明性限制的替代品。我们可以使用EJB部署描述符（ejb-jar.xml）来指定哪些用户角色无需编写任何代码就可以访问哪些方法。不用修改应用代码，只通过重新部署EJB来修改这些限制也是可能的。

Web容器提供了应用URL的声明性限制，但这种限制只提供了不太精细的控制权。它也只保护业务对象免于通过Web接口的未授权访问。在一个集中式Web应用体系结构中，这种限制可能就是业务对象所需要的全部保护，但对分布式应用来说是不够的。

即使我们使用EJB授权，通常也将需要实现Web层授权——我们不想让用户能够尝试他们未经授权的操作。

声明性授权是一个宝贵的服务，该服务能够指出EJB的使用。但是，它在集中式Web应用中不是太重要，因为这种应用中根本不存在对EJB进行未授权访问的危险。

EJB与多线程化

EJB被编写得好像它们是单线程的。EJB容器在后台执行必需的同步。有时，这可以极大地简化应用代码。但是，正如我们在第1章中所讨论过的，它可能会被过高地估计为使用EJB的一个理由。EJB通常解决不了我们的所有线程化问题。即使在大型应用中，也根本不存在编写真正复杂的并发代码的需要。更不用说，EJB也不是解决多线程化问题的惟一有效的解决方案。

示例应用举例说明了这一点。我们不必编写EJB来简化线程化。简单地通过避免实例数据，我们就可以保证小服务程序和其他表示层类是线程安全的。相同的方法可以很容易地被扩展到我们的无状态业务对象。只有在示例应用中涉及到数据高速缓存的地方，才可能会遇到线程化问题。

由于那些EJB程序设计限制（本章的后面将加以讨论）的缘故，以及我们确实想在Web层而非EJB层中高速缓存数据来最大限度地提高性能，所以在线程化问题上，EJB将帮不了我们的忙。因此，我们将使用第三方的非EJB库（比如Doug Lea的util.concurrent）来解决我们可能会遇到的任何线程化问题。这意味着我们不必从头开始编写我们自己的线程安全的高速缓存器，但又不必依赖于EJB来进行并发访问。

当然，在我们已经决定使用EJB的地方（在订票过程中），我们还将利用EJB的并发控制。

声明性配置管理

使用EJB的另一个潜在理由是它提供了用来通过标准ejb-jar.xml部署描述符管理环境变量和应用“连线”的机制，从而使EJB能够在不必修改Java代码的情况下被重新配置。

配置的这种外部化是一个好的习惯。但是，这种外部化仪在某种程度上对所有J2EE构件有效（例如，在有名的DataSource对象的规范中），而且将配置从遍及J2EE应用的代码中分离出来还有更简单的方法，比如使用普通Java组件（JavaBean）（EJB部署描述符是冗长而又复杂的，而且只允许基于XML的定义）。因此，这不是使用EJB的一个重要理由。

EJB的缺点

在决定使用EJB之前，我们除了必须考虑它的优点，还必须考虑它的缺点。

太多的基础结构

除了消息驱动的组件之外，所有EJB都要求开发人员实现至少3个Java类：

- 本地或远程主接口。这个接口允许客户软件创建和定位该组件的实例。EJB容器必须在部署时提供这个接口的一个实现。
- 本地或远程接口。这个接口定义该组件所支持的那些业务方法，也叫做该组件的构件接口。EJB容器必须在部署时以bean实现类为基础提供这个接口的一个实现。
- bean实现类。这个bean类实现定义了那些容器回调的相应接口——javax.ejb.SessionBean或javax.ejb.EntityBean，并提供该bean所支持的那些业务方法的实现。这个bean

类不应该实现该bean的远程或本地接口，尽管这是合法的。但是，实现与每个业务方法的签名相匹配的方法是必需的。我们将在第10章中讨论“Business Methods Interface” EJB技巧，该技巧可以用来规范化这种关系，以避免程序设计错误。

部署在一个EJB JAR部署单元中的每组EJB将需要一个或多个可能很复杂的XML部署描述符。在部署时，EJB容器负责生成EJB拼板玩具的缺少部分：处理远程调用的桩基和骨架，以及本地或远程主接口或本地或远程接口的实现。

使用EJB的代码也比使用普通Java对象的代码复杂。调用者必须执行JNDI查找来获得对EJB组件接口的引用，并且必须处理几种类型的异常。

为什么这一切是必不可少的呢？

要想提供像透明事务管理那样的服务，容器在运行时必须拥有对EJB实例的完整控制权。容器生成类的使用使得容器能够解释对EJB的调用来管理实例生成周期、线程化、安全、事务定界和聚类。当EJB得到正确使用时，这是一个很好的折衷，从而使得容器能够处理用户代码中将需要以其他方式得到解决的问题。在这种情况下，“简单的”非EJB体系结构可能会陷入这么一种困境中：使用DIY解决方案来解决已经由EJB容器很好地解决的问题。

但是，如果由EJB重定向所带来的好处还不够大，我们最终将会得到我们确实不想要的大量基础结构。正统的J2EE观点认为这无关紧要：工具将会同步所有这些类。但是，这是不切实际的。避开复杂性总比依赖于工具来管理它要好。

可以证明的是，Java 1.3动态代理（几个EJB容器现在用它们来取代容器生成类）的引进使得那些EJB API看起来已经过时。通过使用动态代理，用较少特殊需求的应用代码实现相同的容器服务将是可能的。

笔者将会给读者提供那些EJB生存周期图表。不过，重要的是要理解它们，如果读者不熟悉它们，建议读者参考EJB 2.0规范（<http://java.sun.com/products/ejb/docs.html>）的相应部分。

适用于EJB的程序设计限制

在第1章中，我们已经讨论过EJB会产生的那些实际问题：从本质上讲，就是开发、部署和测试中的更大复杂性。使用EJB之前，还有另外一个必须加以考虑的重要问题领域。

EJB 2.0规范的24.1.2节（第494页）中隐瞒了描述“一个Bean提供者要保证企业bean是可移植的并且能被部署在任一符合EJB 2.0的容器中所必须遵守的程序设计限制”的一节。

该规范的这一节对使用EJB的所有开发人员来说是必读的。对那些限制的了解不仅能使我们避免违犯它们，而且也有助于理解该EJB模型。在本节中，我们将讨论那些限制的一些最重要部分和它们背后所隐藏着的基本原理。

那些限制可以被认为是为了3个主要目的：

• 聚类

EJB必须能够运行在一个聚类中，而且EJB容器必须能够自由地管理遍及一个聚类的bean实例。这意味着我们不能依赖于在某个JVM内为真的假设——比如静态字段的行为和同步的影响。

· 线程管理

EJB容器必须能够管理线程，而且EJB绝不能通过创建它们自己的线程或者影响容器管理线程的执行来夺取这种管理。

· 安全

EJB容器必须能够管理EJB的安全环境。如果一个EJB以一种失控的方式工作，声明性或程序性安全保证是毫无价值的。

表6.1总结了最重要的限制及其对EJB开发人员的含义。

表6.1

限制	限制类别	注释	对J2EE开发人员的含义
EJB不应该使用读/写型静态字段	聚类	可能会引起一个聚类中的不正确行为。不保证该聚类中的不同JVM对该静态数据拥有相同值	这个限制使得在EJB层中实现Singleton设计模式变得很困难。我们在后面将讨论这个问题。这个限制不影响常数
一个EJB不得使用线程同步	线程管理聚类	执行线程管理是容器的责任，不是bean开发人员的责任。同步引入了死锁的危险。同步不能被保证像预期的那样工作，如果EJB容器一个特定EJB的实例分布到一个聚类的多个JVM上	这个限制在正确使用EJB的应用中只引起极少的问题，并且不试图解决EJB方面的每个问题。使用EJB的主要目的之一是避免需要实现多线程的代码。同步的使用不是必需的，因为容器将管理对EJB层的并发访问
一个EJB不得操纵线程。这包括启动、停止、悬挂或重新执行一个线程，或修改一个线程的优先级或名称	线程管理	管理线程是容器的责任。如果EJB实例管理线程，容器就无法控制关键的可靠性和性能问题，比如服务器实例中的总线程数量。允许EJB实例创建线程也将使保证一个EJB方法的工作在某一特定事务上下文中完成变得不可能	异步活动可以使用JMS消息传递来实现。容器能够保证使用JMS启动的异步活动的有效负载平衡。 如果JMS是实现所需异步功能度的一种重量级方法，EJB的使用就有可能是不恰当的
一个EJB不得试图输出信息到一个GUI显示上，或接受键盘输入	其他	EJB不是表示组件，所以根本没有理由违反这个限制。一个EJB容器可以运行在一个没有显示器的服务器（在“无头”模式中）	无
一个EJB不得访问本地文件系统（例如，使用java.io包）	聚类	如果EJB依赖于文件系统，这些EJB在这样一个聚类中可能会表现得不可预见：该聚类中的不同服务器可能存有具有不同内容的相同文件。	由于EJB通常是事务性组件，所以把数据写到像文件系统那样的非事务性存储器上是没有意义的。如果像属性文件那样的文件是只读访问所需要的，它们可以被包含在一个EJB

(续表)

限制	限制类别	注释	对J2EE开发人员的含义
		不保证一个EJB容器甚至将有权访问所有服务器中的一个文件系统	JAR文件中，并借助类装入器来装入。作为一种替代方法，读数据可以被放在该bean的JNDI环境中。EJB主要是打算用来处理像数据库那样的企业数据存储器的
一个EJB不得接受一个套接字上的连接，或把一个套接字用于组播	其他	EJB打算服务于本地或远程客户软件，或消息消息。没有理由让一个EJB成为一个网络服务器。	无
一个EJB不应该试图使用反射来破坏Java可见度限制	安全	例如，一个EJB不应该试图访问一个对它不可见的类或字段，因为它是私用的或包可见的	对反射的使用没有任何限制，除非它破坏安全。要避开像java.lang.class.getDeclaredMethods()和java.lang.reflect.setAccessible()这样的方法，但使用java.lang.class.getMethod()方法（它返回公用方法）没有任何问题。没有授予java.lang.reflect.ReflectPermission的任何允许的东西都是可接受的
一个EJB不得试图创建一个类装入器，或获得当前类装入器	安全		这个限制防止J2SE 1.3动态代理在EJB层应用代码中的使用（与EJB容器实现相对立），因为一个动态代理不能被构造为不具有访问当前类装入器的权限。但是，我们几乎没有什么理由要在一个EJB中获得当前类装入器
一个EJB不得试图装入一个本机库	安全	如果一个EJB能够装入一个本机库，该沙盒是无意义的，因为EJB容器无法把安全限制应用到本机代码上。允许EJB调用本机代码也可能会危及EJB容器的稳定性	当访问本机代码是必需的时，EJB不是正确的使用方法

(续表)

限制	限制类别	注释	对J2EE开发人员的含义
一个EJB不得试图修改安全配置对象（Policy、Security、Provider、Signer和Identity）	安全		当我们选择使用EJB时，就选择了把安全的处理留给EJB容器，因此，如果我们正在正确地使用EJB，这个限制不是问题
一个EJB不得试图使用Java Serialization协议的子类和对象替换特性	安全		不要影响串行化的正常使用。只有需要授予java.io.SerializablePermission的那些操作才会遭禁止
一个EJB不得试图把this作为一个参数或方法结果传递给它的构件接口	其他	返回this可防止容器截取EJB方法调用，从而消除了把该对象建模为一个EJB的企图	实现预期的结果有一种简单的迂回方法。从该bean的EJBContext对象中获得一个对该EJB的远程或本地构件接口的引用

这些限制中的大多数在实践中是没有问题的。需要记住的是，如果EJB帮助我们避免编写低级代码的需要，比如处理并发读/写访问的代码，就选择使用EJB：先选择使用EJB，然后再与自我强加的限制做斗争是没有什么意义的。不过，这些限制确实使我们在EJB层中简单地解决一些常见问题变得十分困难。让我们考虑一下这些限制的含义。

我们应该给同步采取多么严格的EJB限制呢？EJB规范明确规定，EJB不应该在它们自己的业务方法中使用同步，而且（正如我们已经见过的）这么做有充分的理由。不过，该规范没有明确规定：如果一个EJB使用一个使用了同步的助手类，这是否违反该规范。

笔者认为，我们在这里需要现实一点。在依赖于同步（而不是EJB容器线程管理）的EJB中尝试业务操作反映出了在EJB根本不适合的地方使用EJB，或者缺乏对EJB容器服务的了解。但是，我们希望用做助手的一些类可能会执行一些同步——或者我们可能没有源代码，而且无法证实它们没有使用同步。日志记录包和Java 1.2以前的集合类（比如java.util.Vector和java.util.HashTable）就是使用了同步的包的常见例子。如果我们对此十分担心，用在关注违犯该规范上的时间有可能会比用在实现业务逻辑上的时间多。

我们应该清楚地知道，这样的类不会用一种将会破坏聚类的方式使用同步，而且它们的同步的使用不会与EJB容器线程管理发生冲突（如果同步被频繁地使用，可能就会出现这种情况）。

EJB中的Singleton问题

开发人员时常发现，他们需要在EJB层中使用Singleton设计模式——或一种实现相同结果的更加面向对象的替代设计模式，比如我们在第4章中已经讨论过的那些设计模式。一个常见原因是高速缓存数据，尤其是当这些数据不直接来自一个持久性数据存储器，并且实体组件又不适合时。遗憾的是，单元素集功能度不适合EJB模型。在本节中，我们将考虑这方面的一些困难和可能的克服方法。

请注意，笔者认为单元素集的推荐用法是合乎逻辑的，即便是在一个聚类中。如果单元素集不适合的原因是由于聚类的缘故，那么EJB不会施加任何不必要的限制。

Java单元素集

普通Java单元素集（在这种单元素集中，Singleton设计模式由一个私用构造器和保存在一个静态成员变量中的单元素集实例来实施）可以用在EJB层中，但受严格规范单元素集使用的限制所支配。由于EJB规范禁止读/写型静态变量和同步的使用，因此以许多典型方式使用单元素集是不可能的。这是同时违反EJB规范的精神和语义的同步使用的一个真实例子，因为它涉及到应用业务逻辑。

就像下列使静态成员变成最终成员的例子中那样，通过在一个静态初始化器中初始化单元素集实例来避免违反读/写型静态数据上的限制是有可能的：

```
public class MySingleton {  
    static final MySingleton instance;  
  
    static {  
        instance = new MySingleton();  
    }  
  
    public static MySingleton getInstance() {  
        return instance;  
    }  
  
    private MySingleton() {  
    }  
}
```

在EJB层外部使用一个静态块来初始化单元素集也是一种不错的做法。仅当实例在静态getInstance()方法中是空值时才实例化它的常见做法容易受竞争条件的影响。需要注意的是，该静态初始化器不应该试图使用JNDI，因为没有任何保证EJB容器将何时实例化这个类。在上述例子中，单元素集实例会在instance变量被声明时得到创建。但是，如果实例化单元素集会导致一个异常，使用一个静态初始化器就是惟一的选择。该选择允许我们捕捉任一异常并在getInstance()方法中重新抛出它。

但是，我们仍可能会遇到问题，视这个单元素集做什么而定。最有可能遇到的问题是竞争条件。由于我们不能在单元素集的业务方法中使用同步，而且运行于容器中的所有EJB都将共享同一个单元素集实例，所以我们没有办法限制并发访问。这就限制了我们为数据高速缓存器使用单元素集：仅当竞争条件是可接受的时，我们才能在一个单元素集中高速缓存数据。这可能会使高速缓存仅局限于只读数据。

仅在下列条件下，我们才在EJB容器中使用普通Java单元素集：

- 单元素集实例能够在一个静态块中（或在声明时）被初始化，以避免竞争条件。
- 单元素集的初始化代价不是非常高。
- 单元素集不要求JNDI访问权来初始化。
- 单元素集的工作不要求同步。

RMI对象

倘若给定这些限制，使单元素集脱离EJB沙盒似乎是一个好办法。这么做的一种常见推荐方法是使单元素集成为远程对象，并从EJB代码中通过RMI来访问这些远程对象。但是，作为RMI对象来实现并访问单元素集是复杂的，易出错的，而且笔者不建议这么做。单元素集不是由EJB容器管理的，而且将会成为一个单故障点。如果RMI服务器发生故障，EJB容器将没有办法处理那些跟着发生的问题。EJB与RMI对象之间的远程方法调用也可能会限制性能。而且，我们肯定不希望由于这个原因而引进RMI，如果我们正只使用带有远程接口的EJB。

无状态会话组件的伪高速缓存器

在某些东西实现起来特别困难的时候，退后一步来看一看我们正试图实现什么是十分值得的。如果采用一种不同的实现策略，我们正试图使用单元素集实现的东西或许是能实现的。

无状态会话组件经常由EJB创建和破坏。每个实例在它的生存周期内通常都将服务于许多客户软件。因此，维护在一个无状态会话组件中的助手对象可以用来提供一个单元素集的一些高速缓存服务：保存在它里面（并在一个应用运行时装配）的数据通过普通Java方法调用是可获得的。

这种方法的优点是，它不违背那些EJB程序设计限制，并且实现起来很容易。同步的问题将不会出现，因为每个SLSB实例将表现得好像它是单线程的。这种方法的缺点是，我们最终将会得到这样的结果：每个会话组件实例仅一个“单元素集”实例。就无状态会话组件而言，最多将有10个实例，因为无状态会话组件生存周期独立于任何单个客户软件。但是，对象重复将是一个问题，如果所有会话组件实例都持有相同的已高速缓存数据（这或许是使用单元素集模式的意义）。

这种方法在使用有状态会话组件时几乎没有价值，而在使用实体组件时根本没有价值，因为这些对象可能经常改变身份。但是，对于一个实体组件将会需要使用一个单元素集，笔者几乎想不出几个合理的理由。

只读实体组件

如果你的EJB容器支持“只读”实体组件，那么使用它们作为单元素集或许是有可能的。但是，这种方法不是可移植的，因为EJB规范目前没有定义只读实体。性能影响可能也不清楚，因为用于只读实体的加锁策略是变化的。笔者不推荐这种方法。

JMX或启动类

另一种方法是使用专有应用服务器功能度（比如WebLogic的“启动类”）或JMX把单

元素集绑定在JNDI中。JBoss提供一种特别精致的方法把JMX MBean绑定到一个bean的环境上。然后，EJB就可以在运行时查找它。但可以证明的是，这些方法只是在外观上而不是从实质上使单元素集脱离了EJB沙盒。例如，同步仍将违反那些EJB程序设计限制。

Singleton设计模式是EJB中的一个问题领域。我们已经考虑了许多的克服方法，其中没有一种方法是理想的。

当使用一个集中式体系结构时，单元素集的最佳解决方案通常是回避EJB单元素集雷区，而在Web层中实现这样的功能度。这与一种在EJB可提供价值的地方使用EJB的策略是一致的；在EJB使事情变得更困难的地方，它明显不提供价值。

当使用一个分布式体系结构且该体系结构带有暴露核心业务逻辑的远程接口时，支持该业务逻辑的单元素集功能度将需要留在EJB容器内。在这种情况下，我们可能需要在当前应用的上下文中选择上述不幸中的较轻者。

定时器功能度

在EJB容器内实现起来有问题的、通常又必不可少的另一部分功能度是定时器功能度，或者说时间安排功能度。假设我们想让EJB层中的一个对象被定期地提示以执行某一个操作。在EJB层外部，我们可以简单地创建一个后台线程来做这件事情，或者使用J2SE 1.3中所引进的java.util.Timer工具类。但是，这两种方法违背EJB规范关于创建线程的那些限制。EJB 2.0中这个为时甚久的问题根本没有令人满意的解决方案，尽管EJB 2.1引进了它。我们通常将需求求助于服务器特有的解决方案。JBoss和其他服务器产品提供解决这个问题的构件。

示例应用中的EJB

示例应用的预订和订票（确认）过程是事务性的。因此，通过使用带有CMT的EJB，以及消除直接使用JTA的需要，我们可以简化应用代码。我们不需要EJB线程管理服务；尽管我们可能将需要在Web层中实现某些多线程代码（参见本章后面的“高速缓存来改进性能”一节），但是很明显，根本不需要真正复杂的多线程代码。

在这里选择不使用EJB是十分正当的，因为示例应用不需要EJB来支持远程客户软件——不使用EJB就无法被合法地实现的那种服务。虽然直接使用JTA可能是更易出错的（尽管不过分复杂），但使用一个没有EJB开销就能提供声明性事务管理的框架将是一种合理的实现选择。

决定怎样使用EJB

假设我们决定使用EJB来利用前面已经讨论过的那些EJB容器服务，那么在我们怎样使用EJB方面仍有许多要讨论问题。

EJB应该做什么

我们应该怎样使用EJB主要取决于我们正在实现一个分布式还是集中式应用。如果该应

用是分布式的，我们将需要使用会话EJB，并按照远程客户软件的要求来暴露该应用的业务逻辑的必需部分。当处理一个真正的分布式应用时，我们需要考虑接口粒度；远程EJB接口不得迫使客户软件做“好唠叨叨叨的”调用，而且应该保证使被交换的数据保持最少，否则远程访问的性能开销将会变得非常大。

我们可以非常有效地使用带有本地接口的会话EJB。尽管它们与普通Java类相比是重量级的对象，但它们所施加的开销不是非常大的（可证明的是，就调用速度而言，它们与使用远程调用时的数百甚至数千倍相比，只比普通Java类慢几倍，而且它们甚至能提供性能好处，只要我们正确地使用容器管理线程化机制）。这意味着当使用本地接口时，我们不必扭曲我们的对象模型来保证EJB暴露粗粒度的接口。

带有远程接口的EJB和带有本地接口的EJB是无法直接比较的，这不只是一个把不同接口暴露给同一种对象的问题。在设计带有远程接口的EJB时，性能考虑比OO设计考虑更重要：它们必须是粗粒度的，那些被交换的对象必须是可串行化的，而且调用者不得被迫做好唠叨叨叨的调用。带有本地接口的EJB看起来更像真正的对象，它们的接口将由正常的OO设计考虑来确定。

何时使用本地或远程接口

本地接口被引进到EJB 2.0规范中的主要目的是为了让EJB彼此调用起来变得更有效。这使Session Facade模式变得更切实可行，而且降低与实体组件相关的开销。Session Facade模式涉及到在远程客户软件与运行在同一个EJB容器中的本地实体组件之间做调停的会话组件。在这个模型中，本地接口是EJB容器内的一种优化，对Web构件和在一个远程界面背后使用EJB的其他对象是不可见的。

笔者认为，实体组件的这种使用（可能与其他实体组件有关系）是让EJB彼此调用变得有意义的唯一情况。最好的做法是避免让EJB彼此调用，即便是通过本地接口。当被调用的这个EJB是一个会话组件时，一种比较好的选择是使用一个普通Java助手类，原因如下：

· 性能

通过本地接口调用EJB比真正的远程调用快得多，但仍比调用普通Java对象慢（请记住，远程调用的开销只是调用一个EJB的开销的一部分：方法截取是截然不同的）。

· 复杂性

使用带有本地接口的EJB比使用普通Java对象复杂。同所有带有构件接口的EJB一样，为每个bean我们至少有3个源文件。和远程组接口一样，本地主接口必须在JNDI中被查找。bean实例无法被直接获得，必须通过JNDI被获得。当然，部署更复杂。

我们将需要提供一些基础结构来使配置EJB容器内的助手类变得更容易。第11章中将讨论一种方法。

笔者认为，在EJB 2.0中，本地接口的最佳用法是允许我们不用远程接口且不用采用一个分布式体系结构就能使用EJB。在这种方法中，EJB层外部的集中式对象（比如运行在Web容器中的业务对象）通过它们的本地接口调用EJB。这使得我们能够享有EJB提供的许多好处，同时又没有太大的复杂性。

除非你的应用天生就是分布式的，笔者建议不要使用带有远程接口的EJB。本地接口在EJB 2.0规范中的引入就证明了我们重新考虑使用EJB的方式是正确的，而且这一引入也使本地接口变成了EJB的默认选择，而使远程接口成为了一个特例。

让一个bean同时拥有本地和远程接口有意义吗

EJB规范暗示，该规范将让一个EJB要么有一个本地接口，要么有一个远程接口。但是，同时提供这两者是合法的。在这种情况下，这两个接口是不相关的。它们无法扩展一个常见超类，因为远程接口上的每个方法必须被声明成抛出这个异常。

虽然这两个接口是不相关的，但bean实现类中的方法可能会使用来自本地接口和远程接口这两者的带有相同名称和参数的一个方法，惟一区别是远程接口签名在抛出子句中包括了java.rmi.RemoteException。但是，bean实现可能会把该方法声明成抛出子句中没有java.rmi.RemoteException，以便同时满足这两个约定。这是合法的Java。但是，这种图省事的方法会十分危险。使用本地接口的调用者将使用按引用调用，而使用远程接口的调用者将使用按值调用。让同一个方法以不同的语义被调用是自找麻烦。

当然，我们不必这样折叠方法实现。给本地和远程方法赋予有区别的名称以便bean实现容易理解可能是比较明智的做法。

在给一个EJB同时提供一个本地接口和一个远程接口之前，应该考虑下列问题：

- 让同一个对象同时支持按引用调用和按值调用可能会令人迷惑，即便借助这两个相同的方法的情况不会发生。
- 远程和本地接口在不同的抽象和粒度级别上可能会不同，因而违背了让一个对象的所有方法都处于一个一致抽象级别上的良好OO习惯。远程接口应该是粗粒度的；本地接口可以是较细粒度的。
- 如果这个bean确实是一个远程对象，而且又提供一个本地接口来支持本地调用者，比较好的做法可能是把本地调用者所使用的功能度再加上一个助手类。让一个EJB调用另一个EJB意味着设计有问题。

给同一个EJB同时提供一个远程和一个本地接口的一种替代方法是，仅给暴露底层业务逻辑的这个EJB提供一个本地接口，然后添加另外一个EJB，让它使用一个远程接口作为一个门面来服务于远程调用者。这种方法具有使本地与远程明确分离的优点。但是，我们必须知道，本地接口仍使用按引用调用，而远程接口仍使用按值调用。

假远程接口

最后，还有大多数现有EJB部署最终所凭借的这种EJB接口方法。在这种方法中，设计师和开发人员都回避该应用该不该是分布式的和RMI是不是真的适合这两个艰难决策。他们使用远程接口实现该应用。然而，他们把Web层和EJB层放在同一个JVM中，并让该服务器凭想像把远程调用“优化”成按引用调用。

EJB容器或多或少（但不完全）表现得就像它在对待本地接口时那样。它仍截取EJB调用，以便使它们能够执行事务管理或其他EJB服务。但是，错误处理必须遵照发生致命错误时的远程语义（因而返回java.rmi.RemoteException和子类，而不是javax.ejb.EJBException）。

严格地说，这推翻了EJB规范。但事实上，远程调用的开销对性能有极大的伤害，因此大多数服务器把这作为它们在EJB和客户代码被部署到同一个JVM中时的默认行为。

按引用调用优化在EJB 1.1时代是一种受欢迎的优化；这种优化使EJB能够用在许多应用内的、它以其他方式执行得不够充分的地方。不过，笔者认为这种优化在当今是相当拙劣的解决方案，原因有以下这些：

- 笔者觉得不满意的是系统打算运行在与它们的声明语义不相符的配置中。这一类应用常常无法按照它们的声明语义所暗指的配置来运行：如果它们使用RMI来部署，将执行得不够充分。
- 为获得足够性能而依赖于这样优化的应用不保证是可移植的，因为这种优化不是J2EE规范的一部分。尽管笔者反对过分强调100%可移植性，但是让一个应用的执行基于一个单独的非标准扩展是一件修改起来不会十分轻松的事情。
- 客户代码被迫考虑可能永远都不会发生的异常。客户代码仍将会遇到java.rmi.RemoteException，但这将意味着EJB内发生了故障，而不是无法连接它。
- 由于编写成RMI语义的代码在被按引用调用时发生故障而导致隐错查找起来困难的弱点（例如，调用者修改了按引用调用传递过来的对象，而这些对象后来又被传递给了其他调用者）。
- 对OO设计的负面影响。远程对象接口由于性能原因常常被设计成粗粒度的。如果那些接口将从不被远程地调用，这样设计将是一件非常糟糕的事情。
- 这种优化不再是必不可少的。EJB 2.0给开发人员赋予了以一种标准方式来指定所需调用的权利。

惟一的好处是我们不用排除其他部署选择就能享有按引用调用的性能。这可是一个重要的考虑因素，但值得我们认真而仔细考虑的另一个因素是，还有没有其他任何部署选择。通常情况下，将不会有。惟一可能会使设计变复杂的是XPers所谓的“You Don't Go to Need It”的一个典型（也可能是高代价的）例子，而且要是扭曲OO设计来保证所有业务对象都是粗粒度接口，我们将只有部署选择的一种选择。

即使一个应用天生就不是分布式的，仍有一个对使用带有远程接口的EJB十分有利的理由：带有远程接口的EJB比带有本地接口的EJB更容易测试。通过远程接口进行访问意味着测试案例不必运行在J2EE服务器中，因而可以使用像JUnit那样的简单工具。但是，远程测试将只在所有被交换的对象都是可串行化的，并且能从J2EE服务器容器中“被断开”时才管用。在有些情况中，比如在CMP实体组件容器管理集的情况下，这是不可能实现的。

EJB接口总结

实体组件决不应该被提供远程接口。同一个JVM内的业务对象（比如Web容器内的其他EJB和构件）应该通过本地接口来访问实体。目前有反对远程地访问实体组件的这类压倒性的性能与设计理由（第8章中讨论）：EJB规范是否应该将它提供为一种设计选择是有疑问的。通过会话组件来调停实体组件访问的Session Facade设计模式，是适合EJB部署提供使用实体组件的远程接口的正确解决方案。

只有当该应用必须是分布式的，并且（或者）远程客户软件将使用RMI来访问EJB时，

会话组件才应该被供给远程接口。是远程对象的EJB通常不应该有本地接口，因为它们不应该由同一个EJB容器内的其他对象来调用。但是，有一种特殊情况：一个基本上是局部性的应用（比如一个Web应用）还需要使用RMI来支持远程调用者。在这种情况下，我们可以给该应用的部分EJB添加远程接口，或者实现调用它们的门面EJB。但是，我们将需要考虑同时支持按引用调用和按值调用的影响。

如果一个应用不是天生分布式的，EJB应该被提供本地接口。当带有远程接口的EJB与调用它们的代码被布置在一起时，笔者不赞成依赖于按引用调用优化。

本地接口应该是EJB的默认选择。远程接口是一个特例，以便在必需远程接口的地方提供额外的功能度（代价是性能、复杂性和OO设计方面的一点损失）。

让一个EJB同时暴露一个本地和一个远程接口是可能的。但是，这两个接口需要被仔细考虑，因为按引用调用与按值调用语义之间存在着差别。

在示例应用中使用EJB

我们已经决定示例应用将不是分布式的。因此，我们将只使用带有本地接口的EJB。没有必要暴露远程接口，尽管我们将来在RMI访问变得必不可少时有可能会暴露远程接口。这意味着我们不必保证所有EJB调用都是粗粒度的。

由于所有应用构件都将被布置在运行该应用的每个服务器中，所以没有必要让所有业务对象都在EJB容器中得到实现。这意味着我们可以挑选是否使用EJB：在我们需要事务管理、线程管理和其他EJB好处的地方，但不在那些EJB程序设计限制可能会引起任何困难的地方。订票过程是EJB在示例应用中惟一明显的应用。

图6.2（以第1章中所讨论的第二种体系结构的图表为基础）举例说明了该体系结构。

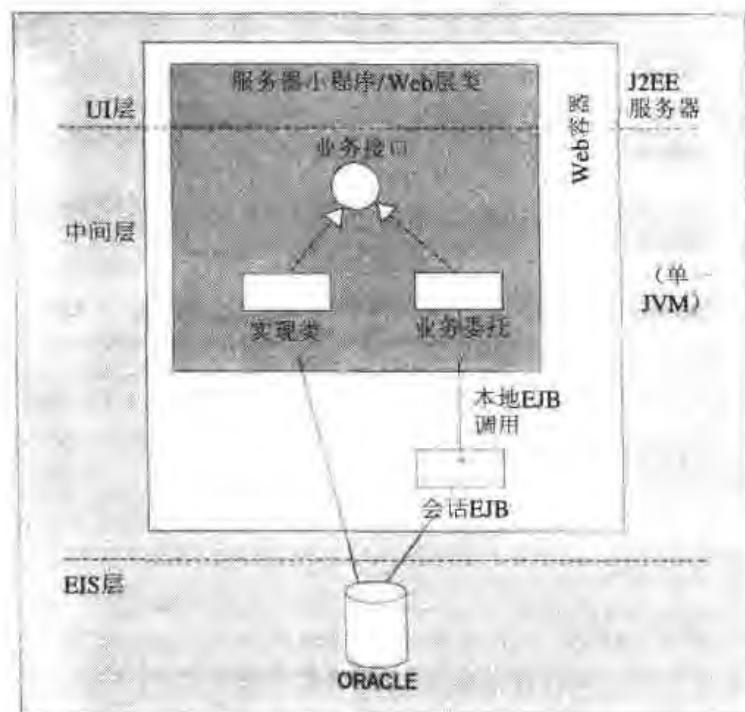


图6.2

大多数业务接口将用运行在Web容器内的普通Java类来实现，而订票对象的接口将用一个带有一个本地接口的无状态会话EJB来实现。Web容器中的一个业务委托将面对这个EJB，并保证运行在Web容器中的对象不必处理EJB特有的细节（如JNDI查找）就能访问它。因此，EJB的使用是一个实现选择，而不是总体体系结构的关键。

决定何时使用带有JMS的异步调用

到目前为止，我们只考虑了同步功能度，而且只考虑了使用RMI/HOP的远程通信。J2EE还允许我们实现异步功能度，以及使用消息传递作为与其他应用进行通信的一种可选形式。

我们可能需要使用消息传递来满足某些业务需求，或者在我们认为一个异步模型是最佳实现选择时，也可能会选择使用消息传递。本节将考虑一些与消息传递有关的结构性与实现性选择。

面向消息的中间件（MOM）和JMS

MOM是支持消息传递的基础结构，所谓消息传递就是指构件间或应用间的松耦合、基于标准而且常常又是异步的通信。消息传递涉及到座落在消息制造者（message producer）（相当于同步模型中的方法调用者）与消息消费者（message consumer）（监听并处理消息的对象）之间的一个消息代理者（message broker）。可能有多个消息消费者。一个消息制造者常常可以在出版了一条消息之后继续进行它的工作；和同步方法调用不同，出版一条消息不中断。这有时也叫做点火并忘掉（fire-and-forget）。

直到目前为止，MOM在J2EE中还没有得到广泛使用。但是，它的价值已在其他系统中得到证实。

传统上，不同的消息传递系统都有专有的API。Java Message Service（JMS）就是针对消息传递的一个标准J2EE API，它位于一个能为不同消息传递系统而实现的Service Provider Interface（SPI）上面。和JDBC相同的是，JMS抽象了使用专有约定的低级访问的细节。使用JMS的应用代码不必关心底层消息传递系统或它所使用的消息传输机制。JMS既支持出版与订阅（Publish and Subscribe，Pub/Sub）消息传递，又支持点对点（Point-to-Point，PTP）消息传递。

JMS API是可供EJB、小服务程序和应用客户软件使用的。J2EE 1.3把JMS集成到了EJB模型中，从而使EJB容器能够处理某些低级的JMS API问题。消息驱动bean为JMS消息消费者提供特殊支持。

我们可以在一个J2EE服务器所管理的任一构件中充分利用JMS。但是，EJB是专门配备用来既制造又消费消息的。

制造消息

运行在J2EE服务器内的任一对象都能制造JMS消息。

就JMS消息出版来说，EJB从特殊的JMS特有服务中决得不到任何益处。和运行于J2EE服务器内的其他对象一样，它们在出版消息之前必须先使用JNDI和JMS API来获得JMS连接和会话。

但是，和资源管理一样，消息制造可以从EJB层的CMT中获得益处。容器管理式事务可以为已事务化的JMS会话提供透明事务管理。

JMS支持事务概念。不过，区分开JMS事务与JTS（即普通J2EE）事务是十分重要的。事务化JMS会话（transacted JMS session）能使我们批量处理消息。当创建一个JMS会话时（我们在发送一个消息制造者中的消息之前需要做的事情），我们必须指定一个会话是否应该被事务化，如果一个会话被事务化，该会话的commit()和rollback()方法应该被调用，以便让JMS知道利用任何已缓冲的消息做些什么。如果是一个提示，这些消息将被作为一个单独的单元来投递（所有消息都被原子地投递）；如果是一个回退，这些消息将被全部删除。

如果有些消息必须被事务化，而其他消息不必被事务化，使用不同的JMS服务对象才是有可能的。像数据库那样的企业资源不被列入JMS事务。资源征召是普通JTS事务的工作。JMS事务是具有一种特殊意义的局部事务。让JTS事务征召JMS资源是有可能的，因此让消息制造者去处理JTS事务（当然，JTS事务也可以由EJB层中的容器来管理）通常比直接使用JMS事务更可取。

如果制造不是事务性的（例如，如果每个操作只有一条消息被出版，或者如果每条消息必须被个别地处理），就没有理由使用EJB作为消息制造者。

消费消息

只有在消费消息方面EJB才真正闪光。

无需使用EJB就消费消息

运行在J2EE服务器所管理的一个环境中的任一构件都可以消费JMS消息。但是，实现一个消息消费者涉及到使用JNDI来查找一个JMS目的地并向它注册一个监听者；相关的JNDI和JMS代码相当复杂。虽然我们可以使用通用基础结构代码从应用代码中消除这些任务，但是，如果可能的话，较好的做法是使用EJB容器所提供的标准基础结构。

我们将在第11章中讨论简化JMS使用的通用基础结构代码。

使用消费驱动bean (MDB) 消费消息

使用MDB消费消息容易得多，因为我们在ejb-jar.xml部署描述符中以声明方式指定消息目的地和筛选器，并把注意力集中在处理消息上，而不是集中在编写低级JMS或JNDI代码上。下面让我们来看一看MDB的部分能力和一些重要的实现考虑元素。

MDB概念

一个MDB就是一个有权访问EJB容器服务的异步消息。一个MDB对客户软件是不可见的。和会话或实体组件不同，MDB没有主或构件接口；MDB实例在运行时由一个EJB容器在一接收到一条JMS消息时调用。根据EJB部署描述符中的一个声明映射，EJB容器知道在一接收到一条消息时调用哪个MDB。

MDB能够消费来自题目（在Pub/Sub消息传递）或队列（在PTP消息传递中）的消息。

EJB容器（不是bean开发人员）为MDB处理JMS消息确认；开发人员不得使用JMS API来应答消息。

由于EJB容器给MDB提供有价值的服务，所以当在一个J2EE应用中使用JMS时，MDB通常是编写消息消费者的最佳选择。

接口

一个MDB必须实现两个接口：javax.ejb.MessageDrivenBean和javax.jms.MessageListener。MessageDrivenBean接口类似于SessionBean和EntityBean接口，而且与这两个接口共享EnterpriseBean超级接口。它定义EJB容器管理一个EJB实例的生存周期所使用的那些回调方法。MessageListener接口是JMS API的一部分，而不是EJB API的一部分，并且只含有一个方法onMessage()，这个方法在EJB容器接收到一条与该MDB实例有关的消息时被调用。

MessageDriven接口为什么不简单地扩展MessageListener接口呢？通过使这两个接口保持分离，EJB规范使消息处理基础结构与被处理消息的性质之间的区别变得十分明确。这保留了EJB规范的未来版本的选择余地，以便它使用相同方法来处理来自不同源的消息——例如电子邮件消息或Java API for XML-based Messaging (JAXM) 消息。

onMessage()方法负责一个MDB中的所有业务操作。和会话组件业务方法不同，这个方法是弱类型化的。消息可能是MDB不知道怎样处理的一种类型，尽管声明性筛选应该减少多余消息的数量。

和无状态会话组件一样，MDB必须实现一个统一而又无参数的ejbCreate()方法。请注意，和无状态会话组件一样，这个方法不是相关接口所必需的，因此未能提供它的故障将只在部署时才是显现出来。在第11章中，我们将讨论可以用来让MDB帮助满足EJB规范的那些要求的一个方便而又通用的超类。

保证一个MDB能够处理它所“打开”的那些消息是这个MDB的责任。应该做运行时测试，以避免运行时的类强制类型转换异常。

适用于MDB的程序设计限制

那些正常的EJB程序设计限制也适用于MDB。例如，MDB不能创建线程或使用同步。

MDB大多数时候具有无状态会话组件。它们不能保存对话状态。它们可以在应用服务器中被组建成池，因为同一个MDB的所有实例都是等效的。

另外还有MDB所特有的几个附加限制。

MDB的异常处理与其他EJB类型的异常处理不同，因为MDB根本没有客户软件。EJB规范的15.4.10节规定，MDB不得抛出应用异常或java.rmi.RemoteException。一个运行时异常（比如一个EJBException或一个未捕捉的可抛出异常）仅当该bean遇到一个意外的致命异常时才应该被抛出。此类异常将提示EJB容器试着重新投递消息，因而可能会导致同一个问题重新出现。引起这种情况的消息叫做毒消息(poison message)，尽管这个问题隐藏在MDB的实现中的可能性大于在消息本身中的可能性。

由于MDB与消息发送者（或客户软件）是分离的，所以它无法访问在运行时通过EJB的上下文在getCallerPrincipal()和isCallerInRole()方法中可获得的安全信息。这限制了消息传递在实践中使用，因为这个限制有时是不可接受的（例如，业务规则可能会不同，取决于试图执行某一操作的用户的身份和权限）。

事务与MDB

和所有EJB一样，MDB可以使用EJB容器的那些JTS事务管理服务。我们已经讨论过JMS事务，它们是一个截然不同的问题。但是，与其他类型的EJB相比，事务对待MDB有很大差别。

MDB不能参与客户软件的事务。

和会话组件一样，MDB可以和CMP或BMT一起使用。

当使用CMT时，要在ejb-jar.xml部署描述符中指定的合法事务属性只有Required和NotSupported。如果事务属性是Required，EJB容器就启动一个新的事务，并调用该MDB在它里面的onMessage()方法。这个事务在onMessage()方法返回时被提交。消息接收是由EJB容器所创建的这个事务的一部分。如果这个事务异常终止（由于一个运行时异常或对MessageDrivenContext的setRollbackOnly()方法的一次调用），消息仍留在JMS目的地中，并将被重新投递。

这是一个紧要关头；它意味着如果一个使用CMT和Required事务属性的MDB由于一个错误而回退或异常终止一个事务，该毒消息的后续投递可能会引起相同的问题。请仔细想一想这样一种草率代码的情况：该代码在未能检查出一条消息的类型时试图把这条消息强制转换成一条TextMessage。这将会导致一个运行时类强制类型转换异常——在EJB容器重新把这条消息投递给同一个MDB（尽管不是同一个实例）时反复不断地出现的一个错误。通过特别注意MDB在运行时不抛出运行时异常，我们可以避免这个问题。

MDB不应该通过抛出一个异常或回退当前事务来表明一个应用错误。这将会直接引起该消息被重新投递，在这种情况下，该故障将会重复出现。一种比较好的替代方法是出版一条错误消息给一个JMS错误目的地。

只有在发生一个致命（但可能短暂）的系统级故障时，才应该抛出一个运行时异常，或者当前事务才应该被回退，进而意味着该消息将被重新投递。后一种行为可能是恰当的，例如，在一个故障源自于一个数据库时。该数据库可能会被临时停机，在这种情况下，当根本问题可能已得到了解决时，再次尝试着处理该消息才会有意义。

当CMT和NotSupported事务属性一起使用时，EJB容器在调用该MDB的onMessage()方法之前将不会创建一个事务。在这种情况下，消息确认仍将由该容器自动处理。

当使用BMT时，事务边界必须由bean开发人员在onMessage()方法内设置。这意味着消息接收不是事务的一部分。如果bean管理式事务回退，消息将不被自动投递。但是，确认仍由EJB容器来处理，而不是由MDB开发人员来处理。bean开发人员应该在acknowledge-mode ejb-jar.xml部署描述符元素中指出JMS_AUTO_ACKNOWLEDGE语义或DUPS_OK_ACKNOWLEDGE语义是否应该适用（JMS AUTO_ACKNOWLEDGE是默认值）。DUPS_OK_ACKNOWLEDGE将产生较小JMS开销。

订阅耐久性

要和某个题目一起使用的MDB（为了Pub/Sub消息传递）可以在ejb-jar.xml部署描述符的选项<subscription-durability>元素中被标记为使用一个耐久性订阅。EJB容器将透明地处理耐久性订阅的细节（比如JMS服务器的订阅者识别）。如果这个元素没有得到提供，订阅是非持久性的。

何时使用异步调用

既然我们已经知道了关于JMS和MDB的主要问题，现在可以来看一看何时使用消息传递。

需要使用消息传递的暗示

异步调用只能在调用者不需要一个立即响应时使用。因此，有返回值的方法不是异步调用的候选者。可能会抛出应用异常的方法也不是候选者。已检查异常提供Java方法的可选返回值，并迫使客户软件采取行动。这对异步调用来说是不可能发生的事情。

假设这些基本条件得到满足，下列暗示将建议使用JMS。这些暗示中有越多的暗示适用，消息传递的使用可能就越恰当：

- **为了改进客户软件对应用响应的感知力**

如果利用“点火并忘掉”机制能够给一个客户软件提供一个立即响应，该系统似乎会比在同一个动作上受阻的系统速度更快。

- **当业务逻辑要求把结果用除了直接方法响应之外的其他方式投递给客户软件时**
发送一个电子邮件就是这方面的一个完美例子，而且常常是JMS的一种好用法。

- **为了执行速度慢的操作**

不产生Java返回值并且不抛出异常的慢速操作是JMS的理想候选者。例如，请考虑一个既涉及基于Java的计算，又包含数据库查询的报告制作操作。在电子邮件中返回这样一个报告可能更恰当，进而做到了报告在后台被生成，同时又立即返回到客户软件。

- **当处理任务的顺序无关紧要时**

虽然我们可以保证同步RMI调用的顺序，但无法保证消息处理的顺序。消息处理的顺序和消息出版的顺序可能会不一样。这就排除了某些情况中的消息传递。

- **为了实现尽可能平滑的负载平衡**

无状态会话组件提供卓越的负载平衡，但MDB仍然更好。虽然EJB容器决定哪个无状态会话组件实例应该处理一个客户请求，但知道自己有足够的多余能力的EJB容器实际上要求消息得到处理。消息排队也随着时间的推移允许负载平衡。

- **为了实现构件之间的松耦合**

需要注意的是，JMS不是J2EE应用中实现松耦合的惟一方法：通常，我们可以简单地使用Java接口来分离对象。

- **为了把暴露基于消息的接口的不同应用集成起来**

在某些情况中，JMS可能是集成遗留系统或实现与非J2EE应用的互操作性的最佳方法。

- **当创建线程不是一个选项时作为创建线程的一种替代方案**

请记住，在EJB层中创建线程是不合法的。JMS和MDB提供一种替代方案。如果一个问题可以通过创建一个后台线程并等待它做完某件事情来不同地解决，消息传递将是非常适合的，因为这不需要一个立即返回值。

- **如果并行一个慢速操作的不相关部分是可能的**

这是上一个暗示的一种特殊情况。

- **为了脱离EJB沙盒**

有时，我们可能需要配置除MDB之外的其他JMS消息消费者，以便使一个应用能够执行在EJB容器内不合法的操作，比如调用本机代码。但是，消息传递不是实现这个目的的惟一方法——我们也可以使用一个运行在一个独立进程中的RMI对象。

使用消息传递的缺点

使用JMS可能会带来下列缺点：

- 虽然消息传递可能给客户软件赋予了对较快速响应的感知力，但使用消息传递实际上增加了应用服务器要完成的总工作量（尽管使用消息传递在调度工作方面能够提供更大的自由度）。若使用耐久性订阅，工作量将会显著增加。
- 消息传递使得在代表客户软件做工作时考虑客户软件的安全身份变得不可能。
- 了解和调试异步系统比同步系统更困难。

因此，不要随随便便决定采用JMS的决定（即使当时的情况暗示使用消息传递是合适的）。

JMS与性能

消息传递得到提倡常常是由于性能的缘故。但是，如果消息传递要达到和RMI相同程度的可靠性（RMI至少保证客户软件知道服务器上某件东西出了毛病），就不可能提供任何性能益处。

JMS的开销变化很大，取决于它被怎样使用：确认模式、订阅耐久性等。有保证的投递（对某些操作是必不可少的）可能会速度很慢。请设想这样一个系统：它将请求进行排队来创建用户账户，因而异步地执行实际的数据库操作。为了变得足够可靠，它将必须使用有保证的投递，因而意味着应用服务器将需要把消息存储在一个数据库或其他持久性存储器中，以确保那些消息得到投递。因此，EJB容器最终可能会在消息被用任何方式出版时做数据库插入操作，因而意味着眼前根本不会有性能益处，只有增加的总开销。

只有当我们无需有保证的投递就能使用消息传递，或者异步操作极端缓慢时，消息传递才有可能提供性能益处。

不要毫无理由地使用消息传递。使用消息传递不当将会使一个系统变得更复杂、更容易出错，根本不增加价值，而且还可能会降低性能。

JMS消息传递的替代方案

在某些情况下，或许有可替代JMS的更简单、更轻便的方案：使用线程创建（允许在EJB

外部进行)或没有JMS的Observer设计模式(也就是说,使用普通Java实现而非JMS来通知监听者——可能是在调用线程或新线程中执行操作的人)。这样实现的Observer设计模式将会比JMS生成更少的开销。但是,它在聚类环境中是否管用取决于监听者必须做什么。服务器提供对JMS的聚类级支持;很显然,当JMS已经提供这种支持时,我们就不必建立自己的有聚类能力的Observer实现。Observer设计模式的简单实现是JMS的一个可行替代方案的情况包括:

- 当该应用没有运行在一个聚类中,并且决没有可能运行在一个聚类中时。
- 当所有事件处理是否都发生在生成事件的服务器上无关紧要时。例如,如果该事件应该导致一个电子邮件在后台的发送,这种情形就适用。我们只想让一个电子邮件被发送。另一方面,如果一个数据高速缓存器需要被变得无效以响应一个事件,这种情形通常将需要出现在运行该高速缓存器的每台机器上:单服务器的事件处理是不够的。

示例应用中的JMS

示例应用无任何一个部分的功能度是异步的。所有用户操作都必须导致同步的方法调用。

但是,使用JMS来试图改进性能似乎是有希望的:通过使已得到高速缓存的数据变得无效来响应系统中某件东西发生变化时所生成的事件。例如,如果一个用户预定了某一场具体演出的座位,那么这场演出的已高速缓存的可供应性细节可能会在整个服务器聚类中被更新。这种方法把关于数据更新的信息“推压”给高速缓存器,而不是让它们在某一给定时间过去之后“拉取”新信息。关于其他事件的消息也可能被发送,其中包括某一预定的取消或一场新演出或一个新节目的输入。这将使用JMS作为Observer设计模式的一个有聚类能力的实现。

这是否会改进性能将取决于JMS开销有多重要,以及一个同步替代方案(比如在超时之后一经请求时就刷新已高速缓存的数据)将是否满足要求。这种异步方法的缺点是使应用变得更难部署和测试。

这样的问题(具有明显的结构性影响)应该尽可能早地解决:通常的方法是在项目生存周期的最初实现应用功能度的一个纵向程序片。在这种情况下,一个纵向程序片将使我们能够运行基准测试来确定一个简单的高速缓存器是否将满足我们的性能需求(最好是首先测试较简单的方法,因为如果这种方法是令人满意的,我们将没有必要实现更复杂的方法)。这种基准测试将在第15章中讨论。

最后的结论是,在涉及到座位可供应性数据的地方,一个高速缓存器可以产生非常好的性能结果,即便数据的超时设置为10秒钟。因此,每当发生一个预定时都导致JMS消息传递的开销是不必要的。如果用户不接着购买,预定可以在数据库中被超时的事实对可供应性更新的一种JMS方法来说也是有问题的。

至于艺术流派、节日和演出的参考数据则是另外一个讨论题目。这些数据从数据库中检索起来花费更长的时间,并且极少会变化。因此,出版一条JMS消息是一种非常好的方法。我们将在示例应用中使用JMS,目的只是为了广播参考数据更新。

身份鉴别和授权

J2EE为身份鉴别和授权提供了声明和程序两种方式的支持，类似于它为事务管理所提供的声明和程序两种方式的支持。

身份鉴别（Authentication）指的是查明用户身份的任务。用户提供凭证（credential）（通常是一个密码）给服务器，以证明他们的身份。授权（Authorization）指的是确定一个特定身份应该被允许访问本应用内的什么资源的问题。

J2EE把身份鉴别和授权集成到所有体系结构层和构件中。一般情况下，用户将向应用接口（比如一个Web接口或一个Swing GUI）标识他们自己。J2EE服务器负责把此标识传播给由该用户的操作而导致被调用的所有应用构件。这将通过使一个用户身份关联到一个执行的线程上来实现。

J2EE服务器的透明授权管理使开发人员免除了把更多资源专用于实现应用功能度。

标准的安全基础结构

J2EE安全是基于角色的。系统的每个用户都能有多个角色。不同的访问权限可以与不同的角色关联起来。

一个J2EE构件的每个同步调用都携带下列安全信息（这些信息通过构件API来获得）：

- 一个java.security.Principal对象。这可以用来查找已鉴别的用户名。
- 该用户是否处于一个特定角色中。这可以用于用户权限的程序性限制。

如果从安全主管（security principal）中获得的用户名是该用户的持久性数据的主键，标准安全基础结构则可以用来避免这样的需要：给已鉴别用户的方法调用传递主键参数。这是服务器提供的标准基础结构怎样才能用来简化程序设计的一个上佳例子。

对受防护资源的访问可能会以声明方式限于某些角色中的用户。这种声明在web.xml和ejb-jar.xml部署描述符中进行。正如我们将要见到的，用户怎样被映射到这些角色上取决于服务器。EJB规范（§ 21.2.4）建议使用声明性安全，而不是使用程序性安全：“Bean提供者既不应该实现安全机制，也不应该把安全策略硬编码在企业组件的业务方法中”。

如果一个用户由于声明性限制而被禁止访问一个受保护资源，结果是多种多样的，取决于请求的类型。一个Web请求将被赋予一个表示未授权访问的标准HTTP返回码，或者将被赋予由该Web应用为这种情况而定义的一个显示页面。如果客户软件是远程的，一个EJB上的一个方法调用将会抛出一个java.rmi.RemoteException；如果客户软件是本地的，将会抛出一个javax.ejb.EJBException。

Web容器很有可能处理用户的安全凭证。小服务程序规范要求符合J2EE的Web容器支持HTTP基本鉴别和基于窗体的鉴别。基本鉴别是HTTP规范中定义的，这种鉴别基于一个用户名和密码来鉴别一个区域（realm，一个安全保护空间）中的一个用户。基本鉴别是不安全的，而且在浏览器中自定义它的外观是不可能的，因此它不应该用在企业应用中。

基于窗体的鉴别比较有用，而且小服务程序规范描述了管理它的一种标准机制。一个

Web容器必须使用有区别的伪URL `j_security_check` 来实现一个POST动作，其中该伪URL使用定义名称 `j_username` 和 `j_password` 来接受用户名和密码参数。该容器必须实现该动作来建立该用户的身份和角色。为调用该安全检查动作而提交的登录窗体是Web应用的一部分。这允许它被赋予任何所需的标记。服务器应该允许登录窗体使用HTTPS来提交，以保证用户凭证在传输中不被破坏。

容器将通过只在必要时（例如，在一个未经授权的用户试图访问一个受保护资源时）才提交登录窗体来迫使用户进行鉴别。这种鉴别称做惰性鉴别（*lazy authentication*），而且意味着从不试图访问受保护资源的用户将永远不会被迫进行鉴别。一旦通过了鉴别，在一个会话期间，用户将再不会遇到盘问。用户身份将会被维持到调用该应用的其他构件的时候。因此，根本没有必要让处于受保护资源背后的用户代码去核实鉴别已经进行过。不修改基础代码而修改受保护资源的集合也是有可能发生的事情——或许为了反映变化的业务规则。

基本鉴别与窗体鉴别之间的选择在Web应用的部署描述符中以声明方式做出。因此，没有必要修改应用代码。

目前还没有任何类似的标准方法可用来鉴别应用客户（即使用EJB的非Web远程GUI）的用户。

声明性和程序性授权技巧使用起来都很简单。可是，声明性模型更可取，因为它简化了程序设计模型，并提高了部署灵活性。

服务器实现

虽然J2EE定义了一个J2EE服务器应该怎样激发用户输入他们的凭证，以及该服务器必须怎样在一个应用内使用调用来传播安全身份，但J2EE没有规定一个服务器应该怎样实现身份鉴别。例如，如果一个用户由于请求一个受限制的URL而遭到盘问时，向一个应用提供了一个用户名和密码，那么服务器容器应该怎样确定那些凭证是有效的，并且该用户处于什么角色中？J2EE没有规定用户信息应该被怎样存储，以及这些信息应该怎样被检查。

目前还没有让用户不用提交该应用的登录窗体就通过身份鉴别的标准API——一个有疑问的疏忽。我们可能需要这种功能度，比如在我们创建一个新的用户账户时。很显然，我们不希望让该用户使用他（她）刚提供的用户名和密码进行登录。我们需要求助专有服务器扩展来实现这个目的。

传统上，身份鉴别一直都是服务器特有的。一般说来，每个服务器都将定义一个安全管理接口，并且这个接口必须被实现成对照一个数据库、目录服务器或用户信息的其他存储器进行自定义的身份鉴别。例如，WebLogic使用了实现 `weblogic.security.acl.BasicRealm` 接口的可插入安全区域。

保证自定义安全管理器有效是很重要的。容器可能会频繁地调用它们来检查用户权限，即使是在一个用户会话已被建立之后（WebLogic提供一个高速缓存区域来降低任何底层安全管理器的工作负荷）。

令人遗憾的是，自定义安全管理器通常将没有权力访问用户应用内的构件，比如暴露用户数据的实体组件或JDO。这是因为安全检查可能是在服务器级上进行的，而不是在应用

级上进行（尽管这在不同的应用服务器之间是不同的）。因此，存在一个潜在的问题：数据访问逻辑的重复，而且也可能存在用户管理器和应用构件对底层数据存储器进行更改的问题。

当解决身份鉴别需求时，需要仔细参考具体应用服务器的资料。

检查提供的凭证的一种标准替代接口可能是Java Authentication and Authorization API (JAAS, Java鉴别与授权API)，它被定义在javax.security.auth包和子包中。JAAS规定了可插入鉴别与授权实现。与J2EE服务器集成的容易程度随着服务器的不同而有所变化。关于JAAS的进一步信息，请参见下列资源：

- <http://www.pramati.com/docstore/1270002/>
来自Pramati的关于集成JAAS与J2EE服务器的资料。
- <http://www.theserverside.com/resources/article.jsp?l=JAAS>
关于集成JAAS鉴别与Struts或其他任何MVC Web框架的文章，其中还含有许多有用的参考。
- <http://java.sun.com/products/jaas>
JAAS主页。
- 应用服务器携带的安全文档。

使用标准J2EE安全基础结构的难点是，编写自定义代码来检查用户凭证并确定用户权限可能将是必不可少的。但是，作为任何一个自产安全系统的一部分，类似代码将是不可或缺的。应用服务器为便于用户管理提供了备有说明文档的接口。这些接口可以针对一个应用被部署到的每个应用服务器来实现，同时又不会影响应用代码库，因而保证了设计仍是可移植的。应用服务器或许还提供支持（比如用于实现自定义用户管理器的抽象超类）。

JAAS为身份鉴别提供了一种较新的标准替代方法，尽管自定义配置仍将是必要的。关于JAAS如何与服务器集成，请参考服务器所携带的文档。

决定何时使用XML

XML是一项核心的J2EE技术。由于它用于大多数服务器所需要的标准J2EE部署描述符(application.xml、web.xml和ejb-jar.xml)和专有部署描述符，所以让J2EE开发人员避免使用XML是不可能的。XML技术也可以选择用来给基于Java的J2EE技术提供有价值的补充。在J2EE 1.3中，最重要的增强之一是它保证JAXP 1.1 API能够供J2EE应用使用（JAXP朝着J2SE 1.4中的那些核心Java库迈进了一大步）。JAXP为J2EE应用中的XML语法分析和XSL转换提供支持。

J2EE应用可能需要生成并语法分析XML文档来实现与非J2EE应用的互操作性（但是，Web服务风格的互操作性极大地抑制了XML的使用；XML也可能会用来支持客户设备和用户代理）。

在J2EE应用中使用XSLT

使用XML技术的另一个重要原因（即便在J2EE应用内也适用）是能让开发人员使用

XSLT把XML数据转换成多种多样的输出格式。这在Web生成HTML、XHTML或WML时是非常有用的，因而意味着XSLT可能是Web层视图技术（如JSP）的一种替代技术。

XSLT专家Michael Kay（Wrox Press出版的优秀图书“*XSLT Programmer's Reference*”的作者）写道：“当首次见到XSL转换语言，即XSLT时，我就意识到这种语言将会成为万维网的SQL——将把XML从仅仅是一种存储和数据传输格式转换成能够用一种灵活、声明性方式来查询和操纵的一种活动信息源的高级数据操纵语言”。

如果数据是用XML文档的形式保存的，或者能够被轻松地转换成一种XML表示，那么XSLT在下列方面提供优于Java技术（比如JSP）的强有力功能度：

- 转换树结构数据。XSLT提供了使用XPath表达式来导航并选择树节点的强有力功能度。
- 分类和筛选数据元素。
- XML和XSLT技巧不是Java特有的。例如，许多Microsoft技术项目都把XSLT用于表示层，并且使用XSLT专家。这可能意味着我们能够使用宝贵的领域特有技能，而不是J2EE特有的技能。
- XML/XSLT范例明确区分开了数据模型（XML文档）与表示（XSLT格式表）。我们可以使用JSP来实现这个目的，但忽视对这种区分的要求是一个极大的诱惑。

但是，XSLT在下列方面弱于Java技术（比如JSP）：

- XSLT在API级而非规范级上与J2EE集成在一起。使用XML和XSLT需要一些自定义编程，而且可能需要使用第三方库或框架。
- 性能。XSLT转换通常比使用JSP再现输出结果明显慢，尤其是当数据必须先从Java对象转换成XML文档时。可是，这一性能开销在许多应用中可能是无关紧要的，并且可以由使用XML和XSLT所带来的其他好处所补偿。
- 串操纵。XSLT不是一种真正的程序设计语言，而且使用它的串操纵是可怕的和不直观的。
- 工具支持。XSLT是非常强有力的，但如果没工具，编辑起来比JSP更困难。XSLT工具仍是相当令人失望的。如果有强有力的XSLT技能可供使用，这可能没有问题。不过，JSP解决方案对许多组织来说实现起来更简单。

XML的“深入”使用

要想使用XSLT，我们需要XML文档来转换。这些文档不必作为串存在，或存在于文件系统上：它们可以在转换前作为W3C org.w3c.dom.Node对象存在。

当数据自然作为XML存在时没有任何问题：例如，如果它来自一个XML数据库、XML内容管理系统或J2EE应用使用XML与之通信的外部系统。虽然XML数据库不常使用，但这些情景中的最后两种在实践中是可以经常见到的。

可是，如果数据还没有作为XML存在，但我们希望把XSLT用于表示，则需要决定我们将在J2EE体系结构中的什么地点把数据转换成XML形式。我们必须在XML的“深入”使用（其中数据以XML而非Java对象的形式在整个应用内传递）与XML的表面使用（其中XML文档只在系统的边界处创建；例如，就在执行一个XSLT转换之前）之间做出选择。

Manning出版的“*J2EE and XML Development*”一书（ISBN 1-930110-30-8）提倡

XML的“深入”使用。在这种方法中，XML文档取代J2EE系统内的Java值对象，进而使边界处的XSLT使用变得更轻松。

通常，这些数据来自一个RDBMS，所以我们可以设法把查询结果转换成XML形式。Ronald Bourret维护着关于这个问题的一个优秀站点（<http://www.rpbourret.com/xml/XMLDBLinks.htm>）。Bourret还出版了关于这个题目的一篇文章（<http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html?page=1>），这篇文章是一个有用的起点。

还存在一个选择：在J2EE服务器里面用Java代码做映射工作，还是在数据库中完成映射工作。RDBMS供应商正匆匆忙忙地提供XML支持，并且有些关系数据库（比如Oracle9i）甚至允许我们把XML直接存储在该数据库中。我们或许还能够获得XML形式的查询结果。这种方法是不可移植的，因为还根本没有相关的标准。

令人遗憾的是，凡是在我们做转换的地方，这将是一项并非不重要的任务。在接下来的几章中，我们将讨论RDBMS模式与Java对象模型之间的“阻抗不匹配”。RDBMS模式与XML文档之间存在一个相当的阻抗不匹配，而且在设法桥接它时存在类似程度的复杂性。某些图书（如“J2EE and XML Development”）中所给出的RDBMS结果集到XML映射的次要例子对实际应用来说是不充分的：查询会变得极其复杂（而且运行起来代价很高），如果我们试图从一个或多个关系查询中建立一个很深的XML分级结构。

数据的关系模式不是分级结构的，并且使用它建立分级结构是很困难的。我们可以运行多个RDBMS查询来生成独立的浅显XML文档，进而希望使用XSLT在客户端上“连接”数据（XSLT允许我们用一种类似于RDBMS JOIN的方法按ID来查找节点——在同一个或另一个文档中）。但是，这用XSLT实现起来是相当复杂的，而且忽略了RDBMS自身的那些能力。这在只涉及到少量参考数据的地方是不适合的。例如，这在涉及到一个国家表的地方可能是一种有效的方法，但在涉及到一个发票表的地方是不适合的。

把关系数据直接提取到XML文档中的困难只是XML“深入”使用方面的几个主要问题之一，笔者总体上认为应该排除它的使用：

- 我们不应该修改总体应用体系结构来支持一种特殊视图策略。如果我们需要用XSLT不会帮助我们生成的一种形式来描述数据，怎么办？例如，XSLT不擅长于生成二进制格式。
- 如果该应用是分布式的，并且使用带有远程接口的EJB，从EJB层中传递XML文档给客户端可能会比传递Java对象速度慢，而且我们可能会遇到串行化困难。W3C Node对象不一定是可串行化的；这取决于实现。
- 该应用内Java构件将会发现处理XML文档比处理Java对象更困难。一个例外情况是树结构数据，对于这种数据，使用XPath API可能会有效。处理XML API远比处理Java对象麻烦得多。
- XML不支持面向对象原理，比如封装、多态性和继承性。如果我们使用更复杂的XML Schema，甚至会完全地失去Java的强有力类型化——在XML使用DTD的情况下部分地失去。
- 处理XML可能会比处理Java对象更慢。“深入”利用XML的应用通常速度很慢，而且可能会在XML和串处理中浪费服务器资源。
- 应用测试起来可能会更困难。测试应用所返回的Java对象比测试XML文档容易。

笔者不提倡XML在J2EE应用内的“深入”使用。虽然以使用XSLT的内部通信为基础的吸引力不强的应用——尤其是如果我们希望把XSLT用于表示——可能比以Java对象和OO设计原理为基础的应用有更慢的速度，而且理解、维护和测试起来比以Java对象和OO设计原理为基础的应用更困难。

Java组件与XML之间的转换

当我们需要把数据转换成XML时，在更接近系统边界的地方进行该转换通常是一种更可取的做法。如果我们在Web层自身内把该转换安排在XSL转换的紧前面，就可以实现视图技术的可交换性，进而使我们能在使用XSLT、JSP和其他解决方案之间进行选择，同时又修改整个体系结构（我们将在第12章和第13章中谈论如何实现这个重要的设计目标）。

在J2EE应用中，数据在再现给一个客户软件（比如一个Web浏览器）之前通常作为Java组件存在。如果我们使用基于Java的技术，比如JSP，那么可以直接处理Java组件。如果我们需要使用XML来暴露数据（例如，使用XSLT来转换数据），则需要把Java组件转换成一种XML表示。令人高兴的是，从Java组件到XML文档有一个相当自然的映射，因而使这种转换远比转换关系数据到XML简单得多。

从一个Java对象图表中生成XML有几种方法：

- 在我们需要表示为XML的每个类中编写一个toElement()方法

这种明显而又幼稚的方法有严重缺点。它给一个对象图表中的每个类都增加了复杂性和了解XML的要求。它把XML文档结构和元素与属性名称硬编码到了每个类中。

如果我们需要生成一种类型稍有不同的文档，怎么办呢？如果我们确实对从一个有多种实现且这些实现不共享同一个超类的接口中生成XML，怎么办呢？

- 使用GoF Visitor设计模式来方便XML的生成

在这种方法中，一个对象图表中的每个类实现一个Visitable接口。一个XML访问者负责遍历该图表并生成一个XML文档。这是一种把元素生成硬编码到应用对象中的出色方法。XML知识被浓缩在访问者实现中。生成不同类型的XML文档是很容易的。这种方法对接口非常管用。使图表中的每个对象都实现Visitabel接口可能对其他任务非常有用；Visitor设计模式是非常强有力的。

- 编写一个知道某个特定对象图表并能为它生成XML的自定义XML生成器

这种方法类似于Visitor方法，但不太常用。它也有把XML生成本地化（localizing）于一个或多个专门化的类中，而不是把XML生成乱丢在整个应用的对象模型中。

- 从使用反射的Java对象中生成XML节点，无需应用特有的XML生成代码

这种方法使用通用的XML生成基础结构来从应用对象中生成XML，而应用对象不需要对XML有任何了解。XML生成最后可能会出现为对象图表的一个完整遍历，或者对象可能会在XPath表达式要求时被动态地转换成节点。这种方法将需要一个第三方库。它将比使用自定义的XML生成稍微慢一点，但性能开销通常没有转换已生成XML的开销大。笔者将这种方法称做本地化（Domification）：XML DOM节点从Java对象中的生成。

第二种和第四种方法通常是最佳的。笔者在几个项目中已经成功地使用了后3种方法。

第四种方法——使用反射的“本地化”是最具吸引力的，因为它最大限度地降减少了应用开发人员编写代码的量和复杂性。虽然反射的使用将会招致少量的性能开销，但该开销或许比转换XML的开销小。

使用反射的一大好处是将保证XML生成代码始终是最新的；当对象发生变化时，将没有必要修改膨胀的XML生成类。我们也将不必编写冗长的XML生成代码；本地化库将会隐藏XML文档创建的细节。如果本地化被动态地完成，只有一个格式表实际需要的那些对象属性会在运行时被引用，从而意味着部分开销可以被避免。

有几件事情是我们在依靠本地化之前需要加以考虑的：

- 虽然不同的库有不同的能力，但那些被暴露的对象可能需要成为Java组件。**bean**属性获得器不需要参数；普通方法常常需要参数，因此不能被自动调用。虽然JSP和诸如WebMacro (<http://www.webmacro.org>) 那样的模板语言提供了调用方法及获得bean属性值的手段，但是大多数表示技术在暴露Java组件时都最管用。由于使模型成为Java组件是一个好的设计习惯（在JSP规范中得到鼓励），所以这不应该是一个主要问题。
- 我们可能需要为某些类型自定义XML生成，尽管我们可以依靠本地化库来处理基本类型。一个完善的本地化库会为个别对象类型考虑可插入的处理。
- 循环引用可能会引起一个问题。这些引用在Java对象图表中是合法的，但在XML树中是没有意义的。
- 意外错误可能会导致某个**bean**属性获得器抛出异常。一个本地化库有效地处理这类错误是不可能的。由于所有数据检索都应该在模型被传递给视图之前完成，所以这不应该是一个问题。

有几个用来把Java对象转换成XML的已出版库。笔者比较喜爱的是Domify开放源项目 (<http://domify.sourceforge.net/>)。Domify最初是Maverick MVC Web框架的一部分，但在2001年12月从这个Web框架中被分离了出来，并成为了一个单独的项目。它是微型库（仅有8个类），并且是非常容易使用的。其他更高级但也更复杂的产品包括Castor (<http://castor.exolab.org/xml-framework.html>)，该产品“能够把几乎任何一个具有‘bean特征’的Java对象编组成XML并编组来自XML的此类Java对象”。要想了解由几个XML-Java转换产品组成的一个目录，请参见<http://www.rpbourret.com/xml/XMLDataBinding.htm>。

Domify使用XML节点创建的动态方法和惰性装入机制。逻辑DOM树中从未被XSLT或其他用户代码访问过的节点从不被创建。一旦被创建，节点就被高速缓存，所以后续访问将更快。

要想使用Domify，必须先创建org.infohazard.domify.DOMAdapter类的一个对象。一旦一个DOMAdapter对象已被建成，它的adapt(Object, String)方法就可以用来本地化对象。一个DOMAdapter对象是线程安全的，因此能够被反复使用（不过，DOMAdapter对象实例化的代价不高，所以创建许多对象没有任何问题）。整个过程看起来像下面这样：

```
DOMAdapter domAdapter = new DOMAdapter();
Node node = domAdapter.adapt(javaBean, "nameOfRootElement");
```

如果转换失败，adapt()方法抛出一个未检查异常：最有可能的原因是一个被调用的获得器方法抛出了一个异常。由于这是不可恢复的，而且在遍历一个Web层模型时不应该发生，

所以这是一种合理的方法（正如第4章中所讨论的那样）。

Domify不检查循环引用，所以我们必须保证要被本地化的bean没有任何循环引用（同样，对Web应用中的Java组件模型来说，这通常不是一个问题）。

为了举例说明Domify如何工作，让我们使用一个简单的bean，并看一看Domify怎样从这个bean中生成XML文档。下列这个简单的bean有4个分别具有String、int、Collection和Map类型（这些类型在下面被突出显示）的属性，连同允许添加数据的方法：

```
public class Person {  
    private int age;  
    private String name;  
    private List hobbies = new LinkedList();  
    private Map family = new HashMap();  
  
    public Person(String name, int age) {  
        this.age = age;  
        this.name = name;  
    }  
  
    public void addHobby(String name) {  
        hobbies.add(name);  
    }  
  
    public void addFamilyMember(String relation, String name) {  
        family.put(relation, name);  
    }  
  
    public int getAge() {  
        return age;  
    }  
    public String getName() {  
        return name;  
    }  
    public Collection getHobbies() {  
        return hobbies;  
    }  
    public Map getFamily() {  
        return family;  
    }  
}
```

为了简洁起见，笔者省略了无参数构造器和属性设置器。现在，让我们构造一个简单的bean：

```
Person p = new Person("Kerry", 35);  
p.addHobby("skiing");  
p.addHobby("cycling");  
p.addFamilyMember("husband", "Rod");  
p.addFamilyMember("son", "Tristan");
```

Domify将自动暴露这4个bean属性。每个节点只在请求时才被创建，之后被高速缓存，以防它在同一-一个XML操作中被再次要求。下列代码举例说明了这个完整的XML文档。请注意对Collection和Map类型属性（突出显示部分）的处理：

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
    <name>Kerry</name>
    <age>35</age>
    <hobbies>
        <item type="java.lang.String">skiing</item>
        <item type="java.lang.String">cycling</item>
    </hobbies>
    <family>
        <item key="son" type="java.lang.String">Tristan</item>
        <item key="husband" type="java.lang.String">Rod</item>
    </family>
</person>
```

编写一个XSLT格式表把这些数据格式化成需要的样子是很容易的。这种方法非常简单，但非常强有力。Maverick MVC Web应用框架证明了它的有用性。

反方向的转换——从XML表示到Java组件也是相当直截了当的，而且是从Java代码中获取应用配置的一种使用广泛的方法。

许多应用和框架都使用XML文档来给Java组件提供长期持久性，比如JBoss的jboss.jcml XML配置文件，该文件使用XML来配置JMX MBean。每个bean属性通常都被表示为一个XML元素，其中属性值保存属性名称。我们在本书中将要描述和用于示例应用的那个通用框架也使用这种方法。大多数应用对象将是Java组件，它们的属性和关系被保存在Java代码外部的XML文档中。这方面的内容将第11章中讨论。

J2SE 1.4引进“Long Term JavaBeans Persistence”来标准化这样的功能度，尽管预言这种标准化将会得到多么广泛的接受还为时尚早，但在Java 1.4中，java.beans API被扩展成了把一个bean读和写为其属性值的一个XML表示。由于本书只关心基于J2SE 1.3的J2EE 1.3平台，所以假设这个新的API是不可用的。但是，它的引进暗示了Java组件到XML映射的重要性。

未来的J2EE与XML

J2EE还会与XML集成得越来越紧密——甚至除了Web服务支持之外。重要的未来增强包括Java Architecture for XML Binding (JAXB) ——通过提供从XML DTD或模式中生成Java类的工具来自动化Java对象与XML文档之间的映射。那些被生成的类可以有效地处理XML语法分析与格式化、简单使用它们的代码，以及提供可能比使用传统XML语法分析或Java-XML转换器（比如Domify）可获得的性能高得多的性能。JAXB主页的URL是<http://java.sun.com/xml/jaxb/index.html>。JAXB规范目前正在起草之中，并且应该在2002年底前完成。

XML是一项核心的J2EE技术。作为J2EE应用中的一项视图技术，如果XSLT简化了表示逻辑，而且在数据已经存在为XML或能被有效地转换成XML形式时，XSLT是一项可替代JSP的可行技术。

笔者不赞成XML在J2EE体系结构内的“深入”使用。XML文档是宽松类型化的，并且在J2EE应用中访问和操纵起来是相当不方便的。

示例应用中的XML

没有理由在示例应用的体系结构中深入使用XML。可是，考虑选择将XSLT用做一种视图技术是有道理的。一个关键的业务需求是，在不用修改工作流程的情况下修改表示来给示例应用“重新贴商标”。XML和XSLT提供了做这件事情的一种杰出方法，尽管我们也可以使用JSP和其他视图技术来达到这个目的，只要我们在Web层中使用MVC方法。

我们可能仍需要保留使用XML和XSLT来生成Web内容的选择，但没有理由把我们自己束缚于使用XML。这意味着我们应该将我们的数据模型创建为Java组件，并使用一个像Domify那样的包将它们转换成XML文档，如果必要的话。要被转换和格式化的用户特有数据（比如预订对象中）的量是适度的，所以这种方法的性能开销不应该成为一个问题。在涉及到参考数据（比如演出日期）的地方，我们可能需要高速缓存转换后的文档，因为越来越多的数据被涉及，而且越来越多的页面被涉及。

判断在第一阶段使用XML和XSLT是正确的将是很困难的（除了用做一个策略性选择之外），除非实际的考虑已暗示这么做是正确的（比如目前只有XSLT技能是可获得的，而根本没有JSP技能可供利用）。JSP将证明它生成最初需求中所描述的那些屏幕是比较简单和快速的。但是，XSLT将来有可能会盛行起来（例如，它将非常适合分类Web层中的参考数据）。在第13章中，我们将详细讨论怎样将XSLT用做一项视图技术，同时用一个例子说明怎么才能在示例应用中不修改示例应用的总体体系结构就使用它。

高速缓存来改进性能

数据高速缓存是许多J2EE应用中的一个重要的体系结构问题，特别是由于Web应用往往读取数据比它们更新数据频繁得多。

高速缓存在分布式应用中尤其重要，因为没有它，远程调用的开销很可能会成为一个大问题。但是，即便在像示例应用那样的集中式Web应用中，高速缓存也是非常有价值的，尤其是在我们需要从持久性存储器中访问数据的时候。

因此，高速缓存不只是优化；它对使一个体系结构有效工作也是非常重要的。

高速缓存可以通过降低服务器负荷来提供快速的性能好处和显著地改善吞吐量。这将会使整个应用都受益，而不只是那些使用缓冲数据的用例。

但是，高速缓存也会导致并发代码和聚类同步的复杂问题。如果没有需要它提供足够性能的充分理由，千万不要实现高速缓存（尤其是在这样的实现非同小可的时候）；通过实现根本不提供实际业务价值的高速缓存，浪费开发资源和创建不必要的复杂性是很容易的事情。

高速缓存选择

示例应用需要中等程度地频繁访问参考数据。我们将肯定需要高速缓存这些数据，而不是在每次使用其中的部分数据时都运行新的Oracle查询。下面来看一看可以给这种类型的应用使用的高速缓存选择。

高速缓存器距离用户越近，高速缓存就会提供越大的性能好处。在示例应用中，我们有下列选择（顺序依次为从RDBMS到客户软件）：

- **依靠RDBMS高速缓存**

如果得到正确配置，Oracle之类的数据库可以高速缓存数据，因此能够非常快速地响应对参考数据的查询。但是，这将不会产生所期望的性能或服务器负载方面的降低。

J2EE服务器与数据库之间仍存在一段网络路程；仍存在通过一条数据库连接与数据库对话的开销；应用服务器必须分配一条已建池的连接，并且我们必须穿过该J2EE应用的所有层。

- **依靠由J2EE服务器所实现的实体组件高速缓存**

这一选择要求我们使用实体组件来实现数据存取，以及我们选择通过EJB容器来访问参考数据。它也把数据高速缓存在距离应用调用路径相当远的地方：我们仍将需要深入到EJB服务器中（涉及到系统开销，即便我们没有使用RMI），去把已缓冲的数据取回到Web层。另外，实体组件高速缓存的效率在不同的J2EE服务器之间将是不同的。大多数服务器实现某种类型的只读支持，但这并没有得到EJB规范的保证。

- **使用能够高速缓存只读数据的另外一个O/R映射解决方案，比如JDO**

这一选择没有把我们束缚于EJB层中的数据存取，并提供了一个比实体组件简单的程序设计模型。可是，如果我们选择采用数据存取的一种O/R映射方法，这只是一个选择而已。

- **以Java对象的形式在Web层中高速缓存数据**

这是一个高性能的选择，因为不用深入J2EE栈，它就能使请求得到满足。但是，它需要的开发工作比已经讨论的那些策略多。最大的挑战是需要容纳并发访问，如果数据被变得无效和更新过。目前，没有任何标准的Java或J2EE基础结构可用来帮助高速缓存器实现，尽管我们可以使用简化相关并发问题的包，比如Doug Lea的util.concurrent。JSR 107 (JCache - Java Temporary Caching API) 可能会提供这方面的标准支持（请参见<http://www.jcp.org/jsr/detail/107.jsp>）。

- **使用JSP高速缓存标志**

我们可以使用这样一个JSP标志库：它提供了我们可以用来高速缓存JSP页面的部分片断的高速缓存标志。有几个这样的库是可免费获得的；我们将在第13章中讨论几个这样的库。这是一个高性能的选择，因为许多请求将能够用J2EE服务器内的最少活动来满足。但是，这种方法预先假设我们使用JSP表示数据。如果我们依赖高速缓存标志的使用来获得可接受的性能，我们就是在断言我们的体系结构依靠一项视图技术的生存能力，因而推翻了我们的MVC方法。

- **使用一个高速缓存筛选器**

另一个高性能选择是使用一个高速缓存式的Servlet 2.3筛选器在请求到达该应用的Web层构件之前截取这些请求。同样，有多种多样的免费实现可供选择。如果已知一个页面完全由参考数据组成，该筛选器就可以返回一个高速缓存的版本（这种方法可以用于示例应用的前两个屏幕——得到最频繁访问的屏幕，因为它们可以被高速缓存多种不同长度的时间）。基于筛选器的高速缓存提供不如JSP标志那么精细的高速缓

存；我们只能缓存整个页面，除非我们重新构造应用的视图层来使一个筛选器能够合成来自多个已缓存构件的筛选。如果该页面只有一小部分才是真正动态的，筛选通常是不可行的。

• 使用HTTP头部来最大限度地减少对无更改页面的请求

我们不能只依靠这种方法，因为我们无法控制代理高速缓存器和浏览器高速缓存器的配置。但是，这种方法会明显降低Web应用上的负荷，并且我们应该把它和自己所选择的上述方法组合起来使用。

JSP标志和高速缓存筛选器具有高速缓存将只使一个Web接口受益的缺点。这两种类型的高速缓存器都不“了解”它们所缓存的数据，只知道它们可能保存了答复一个申请内容的进入请求所必需的信息。这对示例应用的初时需求来说不成问题，因为不需要其他任何接口。但是，另一种高速缓存策略（比如Web层中的已缓存Java对象）可能是必要的，如果我们需要暴露一个Web服务接口。从积极的方面看，“前置高速缓存”不关心它所缓存的数据起源于何地，而且将会使所有视图技术都受益。例如，数据通过使用XML/XSLT、JSP还是Velocity来生存都是无关紧要的。这种方法对二进制和PDF那样的二进制数据也将适用。

无论我们使用什么高速缓存策略，重要的是我们能够禁用高速缓存来验证我们没有掩盖令人震惊的低效率。借助高速缓存的结构以便掩盖严重的瓶颈问题有可能引起其他的麻烦。

一个用于示例应用的高速缓存策略

要想在我们的示例应用中实现一个成功的高速缓存策略，需要区分开参考数据（不变化或极少变化）与动态数据（必须始终最新）。在示例应用中，参考数据（极少变化但一分钟后就该过时）包括：

- 艺术流派；
- 节目；
- 演出日期；
- 关于座位类型、座位名称和每个节目价格的信息；
- 每场演出的座位安排计划：哪些座位是相邻的，总共有多少个座位等。

被最频繁地请求的动态数据将是某一节目的每场演出的每种座位类型的座位可供应性。这些数据显示在“Display Show”屏幕上。业务需求要求这些可供应性信息一分钟就该过时，因而意味着我们只能短暂地高速缓存这些信息。但是，如果我们根本不高速缓存这些信息，“Display Show”屏幕将需要许多数据库查询，进而导致系统上的严重负荷和极差的性能。因此，我们需要实现一个能够适应频繁更新的高速缓存器。

我们可以通过使用“前置高速缓存”来整体地处理高速缓存。“Welcome”和“Display Show”屏幕可以由一个已设置成一分钟超时的高速缓存筛选器来保护。这种方法实现起来简单（我们不必编写任何代码），并且不管我们使用何种视图策略（JSP还是XSLT）都将管用。可是，这种方法有严重的缺陷：

- 虽然业务需求规定数据应该在一分钟内不过时，但只要可能就保证它为最新还是有业务价值的。如果我们使用一种筛选器方法，就无法做到这一点：所有数据都将需要被立刻更新，不管已经发生了什么变化。如果我们使用一种较精细的高速缓存策

略，或许就能够保证：如果我们保存较新的数据作为这个页面的一部分（例如，如果某一订票尝试显示某一场演出已经客满），这些信息就会立即显示在“Display Show”屏幕上。

- 一个Web服务接口将来是一个可能的实际需求。筛选器方法将不会使它受益。这些问题可以通过在Web层的Java对象中实现高速缓存来解决。

在最后做出一个决策之前，还有另外两个需要加以考虑的问题：来自其他应用的更新和一个聚类中的行为。我们从业务需求中知道，没有任何进程能够修改数据库（惟一例外是当管理员直接更新数据库的时候；我们可以提供一个特殊的内部URL，管理员在做了这样一个修改之后必须请求该URL使应用高速缓存器变得无效）。因此，我们可以假设，除非我们的应用创建了一次订票，否则它所保存的任何座位可供应性数据都是有效的。因此，如果我们正把示例应用运行在单个服务器上，并把数据高速缓存在Java对象中，那么我们将从不需要重新查询已得到缓存的数据：我们只需在一次订票被创建时刷新某一场演出的已缓存数据即可。

如果示例应用运行在一个聚类中，这种优化是不可能的，除非该高速缓存器是聚类范围的；创建这次订票的服务器将立即反映出这一变化，但其他服务器在它们的缓存数据过时之前将看不到这一变化。我们已经在前面探讨过这样一种构思：在一次订票被创建时，发送一条将由所有服务器来处理的JMS消息，以便避开这个问题。但是，我们已经判断出，带有一个超时设置的高速缓存提供了足够好的性能，而证明JMS消息出版的系统开销是合理的却很困难。

服务器聚类中的这种数据同步问题是常见问题（它们适用于许多数据存取技术，比如实体组件和JDO）。理论上说，我们应该设法把我们的示例应用设计成既能够在单服务器上实现最佳性能（如果它足够快，则根本不存在它将运行在一个聚类中的需要），在一个聚类中正确工作又是可配置的。因此，我们应该同时支持“从不重新查询”和“超时时重新查询”这两种选择，并且使它们在部署时无需Java代码修改就能够被指定。由于一分钟超时设置值可能改变，所以我们也应该参数化这个设置。

小结

现在该是结束本章的时候了。最后，再来看看一下我们在本章中所涉及到的部分关键问题和为示例应用所采取的决策。

我们已经判断出使示例应用变成分布式的没有任何确凿的理由。业务需求只是表明需要一个Web应用，因此我们不必启用通过RMI的远程访问。如果本应用确实需要支持远程访问，最有可能的是包含基于XML的Web服务。集中式Web应用体系结构可以提供优秀的Web服务支持，因此这也证明这种体系结构不会增加任何问题。使Web层构件和EJB能够运行在各自的服务器上不会使示例应用变得更坚固，因为这种坚固性依赖于Web层状态。另外，这种做法也会使事情变得更加复杂。

由于订票过程是事务性的，所以我们已经决定使用EJB来实现它。由于我们不需要一个分布式模型，所以我们将使用带有本地接口的EJB，因而意味着我们不必担心远程调用的性能开销，或对粗粒度EJB接口的需要。

使用本地接口意味着我们不必把所有业务逻辑（或必须把所有数据存取）都放到EJB容器中。由于处理EJB的复杂性，我们将不使用EJB，除非它提供明确的价值；因此，我们将不把EJB用于示例应用的非事务性部分。这使得我们能够避开EJB程序设计限制方面的问题（比如，与单元素集功能度有关的问题）。

不考虑在EJB容器内实现业务逻辑并不意味着我们把业务逻辑与Web接口密切联系起来。Web接口是变化多端的，并且把我们能够暴露为Web服务的业务接口保持在一个单独的层中是很重要的。这些将是普通Java接口，被实现在Web层中，或被EJB本地访问。

示例应用的功能度无任何一部分是异步的，所以我们将只在这个项目的第一阶段中使用JMS，以便出版参考数据更新，从而使保存艺术流派、节目和演出数据的高速缓存器只能在已知一个修改已经发生时才重新查询数据库。如果应用将只运行在单个服务器上，Observer设计模式可以提供一个简单的解决方案。但是，使用JMS将保证应用将继续工作正确，即便它被部署在一个服务器聚类中。

没有理由在示例应用的内部使用XML（例如，保存来自数据库查询的数据，并在整个应用内传递它，直到它可以由XSLT显示出来时为止）。但是，如果把XSLT用做一项Web层视图技术，XSLT可能是一个有效的选择。

这是我们在第1章中所讨论的4种体系结构的第二种（访问本地EJB的Web应用）。

第7章 J2EE应用中的数据存取

数据存取（或者说数据访问）对企业应用来说是至关重要的。通常，数据源的性能和用来访问该数据源的那些策略将明确规定一个J2EE应用的性能和可缩放性。J2EE设计师的关键任务之一是在J2EE业务对象与数据源之间实现一个简洁而有效的接口。

正如我们将要看到的，把业务逻辑构件与数据存取的细节分开是十分重要的。EJB规范提供了实现必要抽象的实体组件。但是，实体组件模型未必总是合适的，而且不存在我们可以用来达到这一目标的其他技巧。

本章将从一个很高的角度来看一看J2EE设计师和开发人员将要在实践中面对的数据建模和数据存取的选择，因而对使用关系数据库时的常见问题Object-Relational (O/R) 映射特别关注。本章最后将为我们的示例应用制定一个数据访问策略。

在接下来的两章中，我们将较仔细地看一看具体的数据访问策略。这些策略也是J2EE开发人员激烈争论的题目。我们将采取一种实用主义的方法，进而看一看怎样在实际的应用中实现最佳结果。

一项否认声明：笔者无法把读者了解J2EE应用中的数据存取所需要的一切都告诉读者。这要求读者具有所用底层数据源的专门知识。例如，目标是有效而可靠地访问一个ODBMS或一个RDBMS吗？访问Oracle、DB2还是Sybase？根本不存在令人满意的“以一概全”式解决方案，除非数据访问要求是无足轻重的。

到本章结束时，读者应该确信J2EE开发人员需要对数据库技术有扎实的了解，而且应该准备在必要时请教专家的意见。

数据存取目标

笔者为数据存取假设了下列目标：

- 它应该是有效的。
- 它应该保证数据完整性。
- 它应该保证访问和操纵相同数据的并发尝试有正确的行为。这通常意味着防止并发更新危及数据完整性，同时最大限度地减少必要的加锁对应用吞吐量造成的影响。
- 数据存取策略不应该决定我们怎样实现业务逻辑。业务逻辑应该被实现在运行于J2EE服务器内的面向对象的Java代码中，不论我们选择了怎样的数据存取方式。
- 不用重写一个应用的业务逻辑就应该能修改它的持久性策略：使用我们应该能够修改一个应用的用户接口但不影响业务逻辑的同一种方式。通常，这意味着在业务对象与持久性存储器之间使用一个像普通Java接口或实体EJB那样的抽象层。
- 数据存取代码应该是可维护的。如果不小心，数据存取会是导致一个应用的大部分复杂性的原因之一。本章和接下来的两章将探讨最大限度地降低数据存取复杂性的策

略，其中包括使用O/R映射框架，使用一个抽象库来简化JDBC的使用，以及隐藏目标数据库内的一些数据存取操作。

业务逻辑与持久性逻辑

上述第二个和第三个目标意味着需要区分开业务逻辑与数据存取，阐明这一点是十分重要的。

业务逻辑关系到应用的核心工作流程。它独立于应用的用户接口，并且与持久性数据被存储在哪里无关。

与业务逻辑相反，持久性逻辑关系到应用对持久性数据的访问和操作。持久性逻辑通常具有下列特征：

- 它不需要业务规则的运用。和上述示例中一样，持久性逻辑处理一项已决定任务的细节。如果删除一个用户只涉及到某些环境中的级联删除，一个业务逻辑构件就会用来做这个决定，并在必要时请求操作。
- 它在更新业务规则时不可能改变。
- 它不必处理安全问题。
- 它可能会涉及到保护数据完整性。
- 它需要知道目标持久性存储器来实现（尽管不定义）操作。

持久性逻辑不仅不同于业务逻辑，从业务逻辑构件中完整地移走它也是一个不错的主意。如果持久性管理的具体细节被转移到助手类中，从Java业务对象中收集的业务规则将会被表达得更清楚，更易于维护。

虽然业务逻辑应该由Java对象（通常但未必总是会话EJB）来处理，但在J2EE应用中实现持久性逻辑有许多的选择。本章和接下来的两章中将要考虑的重要选择包括：

- JDBC和SQL（用于访问关系数据库）
- 实体组件
- Java Data Objects (JDO)
- 第三方持久性框架，比如TopLink
- 存储过程，有时能够适当用来把持久性逻辑转移到一个数据库内

对象驱动与数据库驱动建模：一场哲学辩论

为一个项目使用关系数据库（RDBMS）还是对象数据库（ODBMS）已经超出了本书的范围。下列讨论更多地面向关系数据库，因为大多数的实际J2EE系统都使用这些数据库。

J2EE应用中的数据建模有两种基本方法：

- 对象驱动建模，其中数据建模由一个对象层（通常是一个实体组件层）驱动，并且该层体现了J2EE应用的持久性域对象的概念

一个例子就是使用一个像Rational Rose那样的建模工具从一个UML模型中生成实体组件，然后从这些实体组件中生成RDBMS表。有几个应用服务器（包括Orion和

WebLogic) 提供了从一个实体组件集中创建数据库表本身的选择。

• 数据库驱动建模，其中数据建模由数据库模式驱动

这种方法基于目标数据库的特征来创建一个数据库模式，用于有效地描述要被持久保存的数据。这个模式的设计在很大程度上将与一个J2EE应用将要使用这个模式的事实无关。例如，一个关系模式将基于规范化这样的关系概念来构造数据。一个例子是设计一个与要被持久保存的数据相适应的RDBMS模式（可能会使用一个像Erwin那样的数据建模工具），然后编写访问和更新该模式的Java代码。

这两种方法可能会产生差别很大的结果。

对象驱动建模有很大的吸引力（“我们知道对象是好东西，所以让我们到处使用它们”），并且常常被关于EJB的图书所推荐。它的优点包括：

- 更大的自动化潜力。或许，我们将能够使用一个建模工具生成Java代码，而不是手工地生成数据存取对象。我们还将免于在应用代码中编写SQL或使用其他数据库特有技术。在开发中，自动化的使用会明显提高工作效率。
- 更大的可移植性。由于对象模型与数据库之间的一个抽象层是必不可少的，所以不用修改对象模型就能使该应用以不同数据库为目标应该是可能的。

当使用一个ODBMS时，自然想到的将会是对象驱动建模。相反，当使用关系数据库时，数据库驱动建模的各种理由就会占上风：

- 对象驱动建模通常不是一个要考虑的选择。J2EE的关键前提之一是表示比其背后的数据结构变化得更频繁。这是实现一个有效数据库表示的理由，而不是实现一个满足某一特定J2EE应用的各种需要的表示的理由。
- 对象驱动建模通常意味着丢弃数据库的许多能力，比如存储过程和有效地自定义查询的潜力。大型公司可能已为一个具有先进功能度的RDBMS花费了许多钱。和实体组件不同，关系数据库已在实践中经过检验。
- 对象驱动建模降低了体系结构的灵活性。它紧密耦合Java代码和RDBMS模式。或许，这种模式能在不同的数据库中被创建，如果该应用被移植。但是，如果有修改数据库模式的其他业务原因（实践中较可能发生的一种情形），怎么办？
- 数据库驱动建模通常会提供更好的性能。这是常常是一个决定性的问题。我们从本章的其余篇幅中将会看到，当访问一个RDBMS时，对象驱动建模在许多方面使得实现满意的性能变得很困难。
- 数据库之间的可移植性可能不像看起来那样是个优势，而且在实践中可能是不可实现的。我们将在下文中讨论这个问题。
- 交流可能会有某种程度的削弱。对象驱动建模忽视基础数据库技术专家的意见。一名好的数据库管理员拥有开发企业应用方面的宝贵经验。数据库管理员将需要在生产中维护和调整系统。
- 通过使用符合数据库产品而不是对象模型的习惯语法创建一个数据库，读者或许能够利用现有的报告制作系统。例如，Oracle Forms和Microsoft Access可以用来最大限度减少实现一个简单的内部管理与报告制作系统所需要的工作量。

J2EE专家在支持对象驱动建模还是数据库驱动建模方面意见不一致。例如，Richard Monson-Haefel——撰写过EJB和J2EE方面的几本书籍的作者，强烈主张数据库驱动建模

(http://java.oreilly.com/news/ejbtips_0500.html)，而Ed Roman——“Mastering Enterprise JavaBeans”一书（Wiley出版，ISBN 0-471-41711-4）的作者，推荐使用EJB对象模型来驱动数据模型。

笔者推荐数据库驱动建模。虽然对象驱动建模很吸引人，但有实际的危险。它的根本问题是它忽略如下事实：关系数据库（a）是非常复杂的应用，以及（b）已经证明工作得非常好。认为J2EE经过几年的发展已经提供了一个优于数十年数据库理论和实践的解决方案是幼稚而又盲目自大的。

笔者发现，相对于一个企业系统的其他构件技术来说，把J2EE理解为起一个乐队指挥的作用是很有帮助的。如果没有这个指挥来指导这个乐队，这个乐队只是一群个体，并且不能有效地运转，但这并不意味着这个指挥试图演奏各个乐器。

数据库驱动建模可能需要我们编写更多的Java代码，但总体上将可能有更少的代码。可是，它给了我们数据和J2EE实现这两方面的更大控制权。它提供了处理任何性能问题的更大空间。如果使用得当，它不会降低可移植性。

O/R映射与“阻抗不匹配”

大多数J2EE应用从关系数据库中访问数据这一事实，对J2EE设计师和开发人员来说既有积极的含义，又有消极的含义。

从积极的方面看，RDBMS背后有着20多年的经验，而且最好的RDBMS已经证明工作得非常棒。从消极的方面看，对象模型与RDBMS模式之间的映射很困难。人们已经把大部分努力投入到了Java和OO语言方面的Object-Relational（O/R）映射中，最后得到的是混合结果。

O/R映射指的是尝试着将Java对象的状态映射到RDBMS中的数据上，以便提供透明的持久性。

关系数据库和面向对象模型有很大的差别。关系数据库基于排序和检索数据的数学概念。关系数据库设计的目标是范式化（normalize）数据（消除数据冗余度）。OO设计的目标是通过把一个业务过程分解成具有标识、状态和行为的对象来建模一个业务过程。关系数据库不支持诸如类、继承性、封装或多态性这样的对象概念。现代RDBMS不只是一个数据存储桶，而且也能够保存保证数据完整性的规则和作用于数据的操作。可是，这并不等于行为作为对象定义的一部分的OO包含。

这些不同模型向O/R映射提出的难题常常集体地叫做对象-关系阻抗不匹配（Object-Relational impedance mismatch）。其中的部分关键问题是：

- 我们怎样在一个SQL查询结果中的列值与Java对象之间进行转换？
- 我们怎样在已映射Java对象的状态发生变化时有效地发布SQL更新？
- 我们怎样为对象关系建立模型？
- 我们怎样为已映射到数据库上的Java对象中的继承性建立模型？
- 我们怎样为其数据横跨RDBMS中多个表的Java对象建立模型？

- 我们应该在对象层中使用什么高速缓存策略来设法降低对RDBMS的调用数量？
- 我们怎样执行集合函数？

只有少数解决方案能满足这些难题中的全部难题——或大部分难题。**O/R映射**解决方案一般把每个对象映射到单个数据行上，这一行通常来自一个表，但有时由一个连接操作产生（如果RDBMS支持可更新视图，使用一个视图来简化映射或许是可能的。一般说来，**O/R映射**解决方案允许这种映射不用自定义代码就能进行，从而向程序员隐藏掉低级数据存取细节。映射通常被保存在那些已映射类外部的元数据中）。

O/R映射在某些情况下工作得很好，但有时可能被吹嘘得太过分。**O/R映射**是访问关系数据库的所有**J2EE**应用的解决方案的假设没有引起太大的争论。笔者认为，这种假设是有疑问的。**O/R映射**既有优点，也有缺点，也就是说我们在使用它之前应该仔细考虑。

O/R映射的重要用处是它消除了开发人员编写低级数据访问代码的需要（这在某些应用中能够极大地提高工作效率），保证应用代码专心地处理对象，以及能够导致创建一个可以支持多个用例的域对象模型（domain object model）。

但是，存在这样一种危险：**O/R映射**降低总复杂性的程度不像把它用到别处所降低的程度那么大。结果可能是复杂的部署描述符，比如实体组件CMP所必需的那些部署描述符，而且透明数据访问的代价是对这种访问的控制力度变弱。

效率也是有疑问的。**O/R映射**解决方案一般假设RDBMS打算在单行和单列上操作。这是一个谬误：RDBMS在元组集上操作最佳。例如，我们用单个SQL操作更新许多行比分别地更新每一行要快得多。如果把数据高速缓存在对象层中是可行的，**O/R映射**解决方案就会提供杰出的性能；如果这是不可能的，或者在聚集更新被需要时，**O/R映射**通常会增加显著的系统开销。

真正高级的**O/R映射**解决方案能使我们享受**O/R映射**好处，同时又没有这些缺点中的部分。

不要假设O/R映射是所有数据存取问题的最佳解决方案。它在某些情况中工作得很好，但有时没有一点好处。

下面是一个**O/R映射**解决方案没有起到帮助作用的标志：

- 在对象驱动建模的情况下，它导致一个不自然的RDBMS模式，其中该模式限制了性能，并且对其他过程是无用的。一个不自然RDBMS模式的标志包括常见数据操作中需要复杂的连接，RDBMS不能实施参考完整性，以及在一个较好的模式已经允许使用一个集合操作的地方却需要发布许多个别更新。
- 在数据库驱动建模的情况下，它产生一个对象层，其中对象与RDBMS中的表有一个一对一关系。除非这些表从对象模型中产生，否则这些表可能不是真正的对象，而且处理它们可能是不自然的和低效率的。如果RDBMS模式发生变化，处理那些对象的所有代码也将需要变化吗？
- 它导致低效率的查询或更新（作为使用任一**O/R映射**层的结果，检查运行在数据库中的那些查询是一个不错的主意）。
- 在数据库内可以使用关系操作轻松而有效地执行的一些任务在**J2EE**服务器中可能需要实质的Java编程来完成，或者可能导致许多Java对象的多余创建。

正如我们将要见到的，在这样的情况中，存在O/R映射的合法替代方案。

O/R映射解决方案通常是OLTP (On-Line Transaction Processing, 联机事务处理) 系统中的一个上佳选择，因为这些系统的用户一般在小数据集上执行操作，而且这些操作常常基于简单的查询。但是，O/R映射解决方案在有OLAP (On-Line Analytic Processing, 联机分析处理) 或数据仓库化需求的地方很少是一个上佳选择。OLAP涉及到非常大数据集的操纵和复杂查询的执行。这些最好使用关系操作来处理。

Data Access Object (DAO) 模式

在访问RDBMS方面没有太大前途是可能的，而且不试图把对象直接映射到数据上——起码不设计需要这种映射的业务对象也是可能的。当我们区分业务逻辑与持久性逻辑时，可以通过定义一个能由业务逻辑构件使用的持久性接口从持久性存储器中分离出业务逻辑。Data Access Object (DAO) 模式（在第9章中详细介绍）在业务逻辑与持久性逻辑构件之间使用普通Java接口的一个抽象层。这些接口的实现处理持久性逻辑。DAO模式在“Core J2EE Patterns”一书中得到过描述，尽管它确实只是Strategy GoF模式（参见第4章）的一个特例。

在Sun Java Center的示例中，DAO模式通常与使用Bean-Managed Persistence (BMP) 的实体组件关联在一起。但是，它能用在一个J2EE应用内的任何地方。实体组件在业务逻辑与持久性存储器之间提供一个类似的抽象，但这些实体组件使我们束缚于一种访问数据的方法和一个特定的O/R映射解决方案：DAO模式给我们留下了一个选择。例如，我们可以使用JDBC、JDO或实体组件来实现DAO接口。DAO模式也具有在一个EJB容器内部或外部工作的重要优点：通常，我们可以在任一地方使用相同的DAO实现，而这增加了体系结构的灵活性。

DAO模式有别于实体组件或O/R映射方法，因为：

- 它不必以O/R映射为基础。虽然O/R映射涉及到名词，比如Customer或Invoice，但持久性门面也能够处理动词，比如“合计某一给定顾客的所有发票的金额”。然而，一个DAO实现可能会使用一种O/R映射方法。DAO接口将按照方法参数和返回值的需要使用域对象（比如Customer或Invoice），以允许源自于业务对象的数据库无关调用。
- 它是以OO设计原理为基础的一种轻便方法。它使用普通Java接口，而不是使用特殊的基础结构，比如与实体组件相关联的基础结构。
- 它的目标不是处理事务或安全，而把这项任务留给业务逻辑构件。
- 它使用任何基础持久性技术。

若使用DAO模式，业务逻辑决定数据存取接口。若使用O/R映射，存在这样一种危险：数据层将要求业务逻辑层也工作。这也是笔者怀疑对象驱动建模的原因之一。笔者经常看到这样的状况：每个RDBMS表的一个实体组件都要求会话组件代码的性质，并导致过分的复杂性。这是本末倒置：一个应用的实质是表达业务规则，不是处理一个预先确定的并且可能是人为的数据结构。

图7.1举例说明了DAO模式怎么才能用来向一个业务对象隐藏起差别极大的数据存取策

略。这两个图表都举例说明了怎样使用一个实现MyDAO接口的DAO。左图（情景1）举例说明了这个接口的一个实现是使用JDBC实现的。右图（情景2）举例说明了一个实现使用一个实体组件层来抽象对RDBMS的访问。虽然这两个体系结构看起来差别很大，但由于该业务对象通过一个确定的接口访问这个DAO，所以没有必要修改业务逻辑来适应这两种数据存取策略。

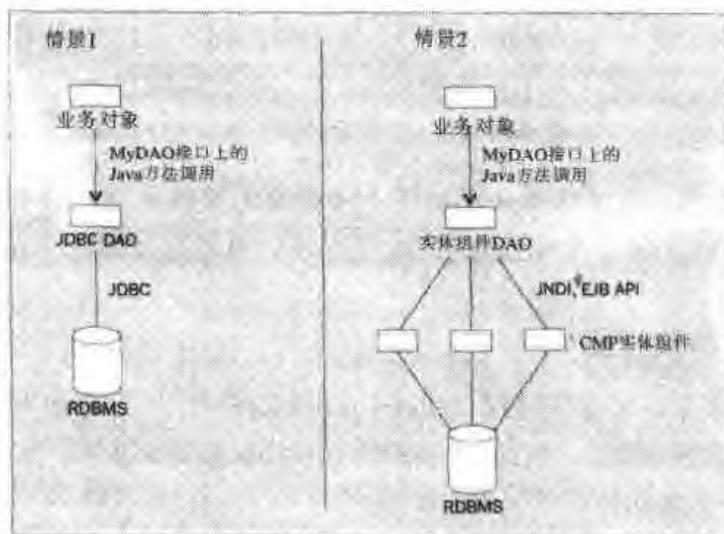


图7.1

和业务对象一样，DAO属于一个应用的中间层。

在第9章中，我们将了解如何实现DAO模式来干净地分离业务逻辑与持久性逻辑。我们还将在示例应用中使用这个模式。

需要重点知道的是，试图分开业务逻辑与持久性逻辑有时是行不通的和没有好处的，而无论我们使用什么抽象和无论多么需要这种分离。例如，我们有时无法做到既分开这两者而又不致命地损及性能；强迫每个实现都使用某一种O/R映射或某些值对象有时是不正确的。

在这样的情况下，我们应该运用安全可靠的OO设计原理——对数据存取来说也不例外。我们应该设法最大限度地减少需要混合业务逻辑与持久性逻辑的代码量，并通过把数据存取隔离到Java接口背后来分开它与其他业务对象。通过把单元测试编写到那些接口，而不是编写到具体类，我们将能够从全面的单元测试开始，如果我们确实需要把该应用移植到另一个持久性存储器。

使用关系数据库

由于大多数J2EE应用中都使用关系数据库，所以下面来看一看一些重要的RDBMS特征和能力。

引用完整性

所有RDBMS都提供了强制引用完整性的复杂机制，比如约束（constraint）（防止添加不尊重现有关系的数据）和级联删除（cascade delete）（当“父”行被删除时相关数据被自动删除）。

根据笔者的经验，一个J2EE应用在一个企业系统中独自享有一个数据库是不常见的。由于数据库是不同应用之间进行通信的一种有效手段，而不是J2EE应用的扩展，所以引用完整性机制是有意义的。因此，基于RDBMS的引用完整性机制在大多数企业应用中是最基本的，而且我们不应该依靠企业应用的Java代码作为数据完整性的惟一保护者。

EJB 2.0为使用CMP的实体组件提供了一种引用完整性机制，许多其他O/R映射也是如此。这是有用的，但只是对RDBMS所实施的引用完整性的一个补充。

存储过程、触发器和视图

大多数RDBMS都提供存储过程（stored procedure）：运行在数据库内的作用于已存储数据的操作。令人遗憾的是，尽管存储过程语言往往是面向SQL的，比如Oracle的PL/SQL，但这些语言在数据库之间是有差别的。

许多RDBMS都提供触发器（trigger）：与某个特定表相关联并且在插入之类的一个事件发生时被自动调用（不必由应用代码调用）的存储过程。

很显然，存储过程和触发器会被误用。虽然一种显而易见的用法是保证引用完整性，但使用一个触发器来实施一个业务逻辑约束会怎么样呢？例如，如果一个触发器将基于如下业务规则来禁止给一个定单表添加一行的尝试：仅当该顾客以前已做过（并且已支付了）至少3个50美元以上的定单并且来自Albania时，700美元以上的定单才能被接受，怎么办呢？这是业务逻辑，不应该放在J2EE应用的数据库中。这类业务规则应该由业务对象来执行。

视图（view）（即虚拟表，通常基于一个在该视图被访问时才得到执行的查询）可以用来简化查询和使O/R映射能够连接。但是，支持程度取决于底层数据库（例如，在Oracle 7.3或以后版本中，连接视图是可以部分更新的，但在Cloudscape 3.6中是不允许的）。通常，视图只适合支持只读对象。

只要下列这些标准得到满足，这些实现特有的RDBMS特性在J2EE应用中就会占有一席之地：

- 它们产生实际的好处：降低代码的量和复杂性，或者产生相当大的性能增益。
- 它们把业务逻辑保留在Java业务对象中。
- 它们被隐藏在一个可移植抽象层后面。

存储过程特别重要，值得较详细地讨论。

J2EE开发人员（和Java开发人员）往往憎恨存储过程。他们之所以如此，有几个正当理由：

- 存储过程不是面向对象的。熟悉Oracle的读者将会反驳说，Oracle 8.1.5引进了Java存储过程。不过，它们解决O/R阻抗不匹配的程度没有把它转移到数据库内那么大，而且它们并没有真正促进Java的面向对象使用。

- 存储过程不是可移植的。对存储过程的支持程度在不同RDBMS之间的差别比SQL方言的差别大得多。不过，当从一个RDBMS迁移到另一个RDBMS时，损失在一组存储过程中的全部投入是很少见的。
- 如果存储过程变得越来越复杂，它们可能会降低一个应用的可维护性。

其他一些常见的反对理由不是太充分：

- “使用存储过程把业务逻辑放在了错误的地方”

如果我们要区分持久性逻辑与业务逻辑，把持久性逻辑放在一个关系数据库中的主意是完全有道理的。

- “使用存储过程意味着J2EE安全可能会遭到危及”

安全是一个业务逻辑问题，不是一个持久性逻辑问题：如果我们把业务逻辑保持在J2EE中，就没有必要限制对数据的访问。

- “数据库将变成一个性能瓶颈”

尤其是，如果仅有一个数据库实例正服务于一个J2EE服务器聚类，存储过程的处理可能会限制总体性能。但是，存在要考虑的折衷方案：

- 根据笔者的经验，应用服务器与数据库之间的网络性能限制一个J2EE应用的总体性能比限制一个设计完善的数据库的总体性能要常见得多。
- 没有理由非要在一个存储过程中而不是在一个J2EE构件中执行一个操作，除非该操作在数据库服务器内比在Java中能被更自然、更有效地完成。因此，如果我们已经在RDBMS内有效地实现了一个操作，并且它更耗用服务器的CPU，这可能暗示该RDBMS被调整地很糟糕，或者需要运行在更好的硬件上。在应用服务器中不太有效地执行这个被频繁请求的相同操作可能会导致一个更严重的问题，并需要添加更多的硬件。

从J2EE应用中使用存储过程是我们应该重实效并避免固执己见的一个方面。笔者觉得，许多J2EE开发人员对存储过程的一概拒绝是错误的。在某些情况中使用存储过程来实现持久性逻辑有明显的好处：

- 存储过程可以处理横跨多个数据库表的更新。这种更新对O/R映射是有问题的。
- （第一点的一种更一般形式）存储过程可以用来向Java代码隐藏起RDBMS模式的细节。通常，Java业务对象没有理由应该知道数据库的结构。
- J2EE服务器与数据库之间的往返行程可能会很慢。使用存储过程可以合并它们：使用我们在分布式J2EE应用中努力合并远程调用来避免网络和调用协议开销的相同方式。
- 存储过程允许RDBMS结构的有效使用。在某些情况下，这将会导致性能的明显提高和RDBMS上的负载下降。
- 许多数据管理问题使用PL/SQL之类的数据库语言解决起来会比从Java中发布数据库命令要容易得多。关键是为这项工作选择正确的工具。笔者不会考虑舍弃Java而使用Perl来构造一个大型应用；如果使用Perl就能不太费劲地编写一个文本操纵实用工具，笔者也不会考虑舍弃Perl而使用Java来浪费自己的时间和雇主的金钱。
- 在能够被利用的现有存储过程中可能有一项投入。

- 存储过程从Java代码中调用起来很容易，所以使用它们往往会降低而不是增加J2EE应用的复杂性。
- 极少数拥有现有IT部门的企业已经把他们的所有应用都移植到了J2EE，或者不久就可能会移植到J2EE，因此，持久性逻辑在RDBMS中可能比在J2EE服务器中更有用，如果它能被其他非J2EE应用（比如自定义的报告制作应用或自主开发的VB客户软件）使用。

使用存储过程的危险是使用它们实现业务逻辑的诱惑。这有许多负面的后果，例如：

- 没有实现该应用的业务逻辑的单独体系结构层。业务规则的更新可能会涉及到同时修改Java代码和数据库代码。
- 该应用的可移植性将随着存储过程变得越来越复杂而降低。
- 两个不相干的团队（J2EE和DBA）将共同负责业务逻辑，从而引起交流问题的可能性。

如果我们区分开持久性逻辑与业务逻辑，使用存储过程将不会破坏我们的体系结构。如果使用存储过程满足下列标准，它就是一个好选择：

- 这项任务无法简单地通过使用SQL来完成（如果没有存储过程）。调用一个使用JDBC的存储过程比运行普通SQL有更高的开销，而且数据库中有更大的复杂性。
- 这个存储过程可以看做是一个简单Java接口的一个数据库特有实现。
- 使用存储过程只涉及到持久性逻辑，而不涉及到业务逻辑，并且不含有经常变化的业务规则。
- 使用存储过程产生一定的性能好处。
- 存储过程的代码不过于复杂。如果一个存储过程是适中的，回报的一部分将是一个更简单的实现（与可以在运行于J2EE服务器内的一个Java对象中已被完成的实现相比）。特别是，在一家拥有DBA资源的组织中，10行PL/SQL代码维护起来将证明比100行Java代码容易得多，因为这样一个大小差异已经证明PL/SQL是这项工作的正确工具。

不要使用存储过程来实现业务逻辑。业务逻辑应该放在Java业务对象中实现。但是，存储过程是实现一个DAO的部分功能度的一个合法选择。刚开始设计时，没有理由拒绝存储过程的使用。

同上述题目有关的一个案例：在2001年的晚些时候，Microsoft发布了Sun的Java Pet Store的一个.NET版本，他们声称该版本比Sun Java Pet Store快28倍。这一性能增益看来主要归因于Microsoft的数据存取方法，这种方法使用SQL Server存储过程取代了实体组件。

笔者发现J2EE业界中对Microsoft声明的反应是失望和担心（请参见http://www.theserverside.com/discussion/thread.jsp?thread_id=9797）。J2EE纯化论者对Microsoft使用存储过程的反应是极度的恐惧，因而辩解道Java Pet Store体现了一个优秀的设计，拥有“一个面向对象的域模型”和数据库之间的可移植性这两大好处。尤其是，纯化论者担心这个Microsoft基准可能会对不应该允许确定一个应用应该运行得有多快的经理们产生不良影响。

正如读者在读完第4章之后所知道的，笔者是OO设计原理的一名热心支持者，但是笔者读到这样的反应时也不大相信。设计只是达到某一目的的一个工具。现实中的应用必须满足性能需求，而阻碍这一点的设计就是拙劣的设计。

另外，恐慌和否认也是多余的。这个基准并没有证明J2EE天生就比.NET性能差。Microsoft所使用的体系结构方法在J2EE中同样可以实现（事实上，比Sun原来的Per Shop示例更容易），但这并没有证明J2EE正统观念会有危险。

存储过程的使用正在退回到两层化应用的悲惨时代吗？回答是不。两层化解决方案只是用在数据库完成业务逻辑。存储过程应仅用在一个J2EE系统中以执行将总是频繁地使用该数据库的操作，无论这些操作是被实现在数据库中，还是被实现在与该数据库交换大量数据的Java代码中。

RDBMS性能问题

一个J2EE应用使用一个RDBMS越频繁，保证数据库模式有效和数据库得到精确调整就越重要。

RDBMS性能调整

和J2EE应用服务器一样，RDBMS是很复杂的软件，比绝大多数用户应用复杂得多。和J2EE服务器的情况一样，需要专门知识的大量调整选项会对应用性能产生极大的影响。应该聘用一名DBA。

很显然，性能调整将是一场必败之战，如果：

- 数据库模式不能有效地支持常见的访问需求。
- 所使用的查询是低效率的。
- 这些查询或运行它们的方式在数据库中引起过多的加锁。令人遗憾的是，在一个RDBMS中有效的查询在另一个RDBMS中可能会产生争用：无论从语法上还是从语义上看，SQL都不像Java那么可移植。

惟一可能获胜的战斗是索引(index)的创建。顾名思义，索引能使RDBMS基于一个或多个列中的值非常快速地查找到一行。在大多数数据库中，这些索引被自动创建在主键上，但可能需要被创建成支持一些用例，不管我们在Java代码中使用了何种数据存取策略。例如，如果我们在自己的用户表中有数百万个用户，每个用户的数据包括一个数字主键（被默认地索引）、一个电子邮件地址和密码。如果电子邮件地址是该用户的登录名，在一个用户试图登录时，我们将需要按电子邮件地址和密码快速查找行。如果这些列上没有一个索引，这将需要一个全表扫描(full table scan)。在这样大小的一个表上，这将会花费许多时间和大量磁盘存取。如果在电子邮件和密码上有一个索引，查找一个用户的行将几乎是即时的，而且RDBMS上的负荷最小。

降低范式化

有些时候，某些查询一直很慢，而与查询优化和性能调整无关，这通常是因为它们涉及到多表连接。在这种情况下，有一种最后凭借的手段：降低范式化(denormalization)，或者说，由于性能原因而在数据库中保存冗余数据。一般说来，这会极大地降低所需连接的

复杂性。笔者假设读者已经熟悉范式化（Normalization）的关系概念。这是任何J2EE开发人员必须掌握的基本知识，因此，如果必要，请参考一本关系数据库入门读物。

降低范式化具有严重的风险。它增大数据库的大小，而数据库大小的增大在数据量很大时可能会有问题。最重要的是，降低范式化会损害数据完整性。只要有一个东西被存储一次以上时，就存在那些副本失去同步的可能性。

有时候，在Java中而不是在数据库中降低范式化是可能的。这会产生非永久性冗余度，因此在原理上是可取的。通常，只有当我们可以在一个部分只读的数据结构中高速缓存适当数量的数据时，这才是一个选择项：与在数据库中相比，Java代码或许能够更有效地导航这个结构。

有时，尝试降低范式化只是为了使J2EE应用能够更轻松地使用数据库。由于涉及到的风险，降低范式化极少是一个好主意。

不要仅仅为了支持一个J2EE应用就降低一个关系数据库的范式化。数据库模式有可能生存得比J2EE应用更久，因此在这种情况下，降低范式化的代价是很大的。

可移植性与性能的比较

慎重考虑是否有更换数据库技术的任何现实可能性是很重要的。这和更换数据库供应商不是同一回事情。例如，从一个Microsoft SQL Server数据库到一个Oracle数据库的更换比从一个关系数据库到一个对象数据库的更换所产生的影响应该小得多。这和更换同一个数据库内的一个数据库模式也不是同一回事情。如果发生更换数据库模式的事情，我们应该能够足以应付它。

组织在数据库上进行了大量投入，主要花费在许可成本和职员培训这两方面。然后，他们给这些数据库填充了宝贵的数据。他们不打算简单地放弃这项投入。在实践中，J2EE应用在根本不同的数据库之间被移植是极其少见的。事实上，一家组织从一个应用服务器转移到另一个应用服务器（甚至从一个J2EE之类的应用服务器平台转移另一个.NET之类的应用服务器平台或反之）的可能性要大于从一个RDBMS转移一个ODBMS的可能性。由于涉及到的投入问题，甚至连RDBMS供应商之间的更换在实践中都难得一见。

这个问题为什么重要呢？因为它对我们处理持久性的方式有影响。数据库可移植性最重要这一假设所带来的各种令人遗憾的结果是：(a) 对一个应用的目标数据库不太感兴趣，以及(b) 浪费精力来实现对大多数应用来说并非目标的东西：数据库之间的完整可移植性。组织会把J2EE看做是其数据存取策略的核心这一假设造成了对J2EE中的持久性过分注重的缺陷。

请考虑一个现实情景：一家公司已经在一个Oracle安装上花费了数万美元，由于这家公司的软件集中除了J2EE应用之外的其他几个应用也使用这个数据库，所以它雇用了几名Oracle数据库管理员（DBA）。J2EE团队与那些DBA没有取得联系，而且坚决主张开发没有利用任何Oracle特性的“可移植”应用。很显然，就这家公司在Oracle和实际业务需要方面所做的战略投资来说，这是一个极差的策略。

如果读者的组织明显倾心于一项特定的数据库技术（而且也很可能倾心于一个特定的

供应商），那么下一个问题是：是否利用供应商特有的功能度。

如果该功能度能够提供真正的好处，比如改进的性能，那么回答是“是的”。我们决不应该把这种可能性拒之门外，因为我们的目标是实现一个具有100%代码可移植性的应用；我们应该保证有一个把任何不可移植特性隔离到Java接口背后的可移植设计（请记住，好的J2EE习惯基于好的OO习惯）。代码的总体可移植性通常意味着一个无法针对任何平台而被优化的设计。

我们需要抽象来实现可移植性。问题是我们应该在什么层次上实现这种抽象。

下面来看一个例子，在本例中，两种可选实现方法演示了不同的抽象层次。

• 使用了一个DAO的抽象

通过决定“AccountManager会话组件将使用一个数据访问接口的一个实现，而这个实现能够为余额超过某一指定数额的所有账户和上月总和超过某一指定数额的所有事务返回值对象”，我们可以分开业务逻辑与数据存取代码，并实现一个可移植的设计。我们已经推迟了实现对该DAO的数据访问，以便不影响我们施加在该实现应该怎样着手处理数据访问方面的任何约束。

• 使用了CMP实体组件的抽象

假设尝试100%的代码可移植性。Account实体组件将使用CMP。它的本地主接口将让一个findByAccountBalanceAndTransactionTotal()方法返回满足这些标准的实体。这个方法将依赖于一个ejbSelectByAccountBalance()方法，而这个方法将返回满足余额标准的实体，余额标准将由一个不是RDBMS特有的EJB QL查询来支持。findByAccountBalanceAndTransactionTotal()方法将循环由ejbSelectByAccountBalance()方法返回的那个实体集，进而为每个实体都遍历一次相关的Transaction实体集以累加它们的值。

这种迂回方法是必不可少的，因为EJB QL（从EJB 2.0起）不支持集合函数：或许是因为这些集合函数不被看做一个关系概念。笔者认为自己已经十分了解那个算法，但笔者肯定将会编写单元测试来检查该实现（当然，这将是相当困难的，因为该代码将只运行在一个EJB容器内）！

假设我们正在开发所述应用来使用一个RDBMS。

这两种方法的第一种可以通过使用该数据库的能力来实现。那个数据访问接口提供一个高层次的抽象。它的实现将很可能会使用JDBC，而且那个查询的逻辑可以在单个SQL查询中有效地实现。将该应用移植到另一个数据库最多只涉及到使用另一个持久性API重新实现同一个简单的DAO接口（事实上，该SQL查询在RDBMS之间有可能是可移植的）。

第二种方法增加显著的系统开销，因为它迫使我们在一个太低的层次上执行抽象。结果，我们无法有效地使用RDBMS。我们必须使用威力小于SQL查询的一个EJB QL查询，并且被迫从数据库中取出大量数据，以及在J2EE服务器内用Java执行数据操作。结果是更大的复杂性和更差的性能。

在笔者所描述过的那种业务状况下，第二种方法提供总代码可移植性这一事实根本没有任何好处。第一种方法现在给了我们一个最佳解决方案，并且我们已经干净地隔离出了需要重新实现的少量代码，如果后端曾经发生了明显更改。

无需经历这种痛苦，我们就可以使用EJB QL轻松地做许多事情，所以本例从它的最差方面展现了EJB QL抽象。本例阐明了这样一个事实：不屈不挠地追求代码可移植性会导致极其低效的数据访问。

如果一个应用利用有价值数据库特有功能度是不可能的，该应用就拥有一个拙劣的体系结构。

分布式应用中的数据交换

我们已经在前面特别提过，与集中式应用相比，分布式应用是多么复杂。这种复杂性也影响数据访问。无论我们在一个分布式应用中怎样实现持久性的方方面面，都将需要把数据发回给远程客户软件。这些数据需要与后台数据存储器断开。当然，我们可以给该客户软件发送对实体组件的远程引用，但这么做保证会导致可怕的性能和可缩放性。

Value Object J2EE模式

一个值对象就是一个可串行化Java对象，并且该对象能够在网络上被有效地传递，以及含有当前用例的上下文中所必需的数据。值对象通常是服务器在其上调用设置器方法和客户获得器方法的Java组件。不过，一个客户软件可能会修改一个值对象的状态，并把它传回给服务器以更新持久性数据。存在许多可能的细化；例如，客户软件可能会接收到一个拥有一个不可变接口的对象，而服务器实现会增加获得器方法。

值对象有时也叫做Data Transfer Object (DTO, 数据传输对象)。

Value Object J2EE模式描述使用值对象来避免冗长远程调用的用法。在EJB 1.1中，这通常用在会话组件与实体组件之间。EJB 1.1实体组件需要方法来执行块数据检索与更新，以便会话组件能够用单个远程调用更新数据，而不是通过调用个别设置器方法更新数据。由于EJB 2.0实体应该被赋予本地接口，所以不再需要让实体组件来实现处理值对象的方法。

这是一件好事，因为值对象是一个业务过程的人工制品，不是数据存储器。事实上，同一个实体在不同情况中可能需要多个值对象（虽然下列讨论使用一个实体组件作为例子，但这些问题许多持久性策略所共有的）。

请考虑一个含有下列数据的Car实体（不是对建模的实际尝试）：制造年份、型号名称、制造商（一对多关系）、颜色、商标牌和该轿车的拥有者（一对多关系）。在我们的简单模型中，每个拥有者可能只拥有一辆轿车，并且必须有一个相关联的保险单对象。每个保险单对象与一家保险公司关联起来。

图7.2是一个UML类图表，阐明了持久性对象之间的这些关系。

没有任何一个单独的Car值对象将是通用的：它将含有太少或太多的数据。例如，如果我们只取Car实体中直接保存的信息，则只能得到年份、型号名称、颜色和商标牌。这在某些情况下或许是有用的，但如果我们要对该轿车的所有权历史感兴趣，怎么办呢？如果我们决定单个值对象应该横跨关系，就会遇到在哪单停止的问题，或者说最终将会执行太多的查找和发回太多的数据，进而严重地降低了性能（例如，我们应该停在一个Car的相关Owner对象上，还是应该继续物化该传递封闭包，其中包括InsurancePolicy和Insurer对象？）

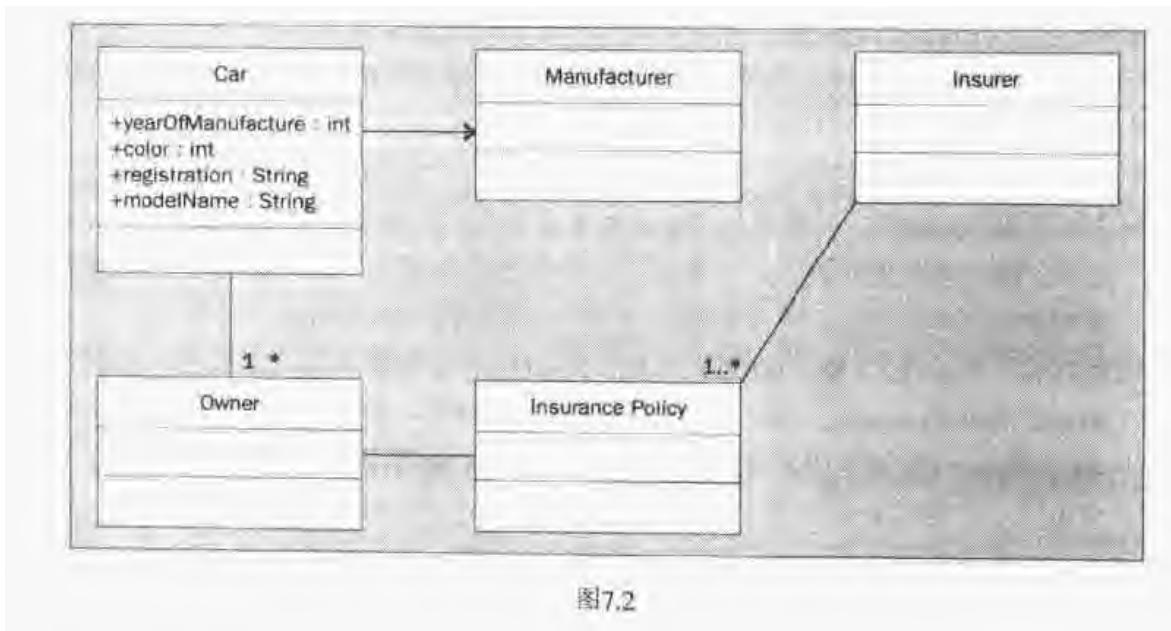


图7.2

最佳但相当复杂的解决方案是使用多个值对象：例如，我们可能有一个CarOwnership值对象——含有轿车的主键以及一个由保存关于轿车拥有者的信息的值对象所组成的数组。集中式应用通常不会遇到这个问题，因为没有必要“断开”数据。

创建值对象来响应业务需要的一个组件将实现Value Object Assembler模式（“Core J2EE Patterns”）。值对象装配器可能是会话组件，也可能是持久性门面。

按照需要创建值对象来支持用例。不要每个实体组件或RDBMS表都自动使用一个值对象（尽管有时这是正确的）。

“通用” 值对象

Value Object模式需要花费大量精力来实现。我们可能需要给每个持久性对象都拥有多个值对象，而且将需要编写自定义代码来创建和返回值对象。为一个实体组件或其他已映射对象自动生成一个值对象常常是可能的，但这将会产生一个“以一概全”的值对象，但正如我们已经见过的，这种值对象将满足不了所有需求。

特别当许多值对象对同一个持久性对象必不可少时，一个HashMap可以被看做一种发送数据给客户软件的替代方案。这不具有强大的类型化的优点，但可能是值得一试的，如果它消除值对象中的过量代码。

另一种非类型化方法是使用XML文档作为通用值对象。但是，我们在第6章中得出的结论是：这种方法具有在某些情况下会变得不适合的严重缺点。

使用JDBC行集“断开” 数据存取

Value Object模式的另一种替代方案是使用标准的JDBC基础结构。

Javax.sql.RowSet接口是java.sql.ResultSet接口的一个子接口，并能够与数据断开连接。一个结果集（Resultset）也正是一个值对象集。EJB Design Patterns建议按照Data Transfer Rowset J2EE模式使用行集（Rowset）来编组要发回给客户软件的只读数据。

这种方法是有效而又通用的，并且可能会节省自定义值对象中的代码。但是，笔者觉得它不应该被看做是一个模式，而且不推荐使用它。从体系结构的角度看，这样使用JDBC行集存在如下严重缺陷：

- 它牺牲类型安全性。
- 它不是面向对象的。虽然我们可能把对象存储在一个杂凑图（hashmap）中（必然迫使客户软件做类型转换），但数据以一个行集中的关系形式（而不是以对象形式）而存在。
- 所有的行集getXXXX()方法都是从ResultSet接口中继承来的，因而意味着它们抛出java.sql.SQLException。客户软件被迫捕捉这些异常；事实上，它们不必操心可能出现的问题，就应该能依靠由EJB层从数据库中成功地提取出的数据。这体现了这样一个事实：行集对象在一个太低的抽象层次上被客户软件操纵。
- 来自数据库的列名称和结构将被建到客户代码中。如果数据库模式发生变化，这将产生严重问题。或许，这是最大的缺陷。
- 即使从一个RDBMS移植到一个ODBMS是不太可能的，把数据库技术暴露给UI层也是毫无道理的。中间层的用途之一是提供一个抽象来隐藏起这样的低级细节。
- RowSet接口含有许多与客户软件不相干的方法。
- 假定J2EE应用性能中的一个主要因素是应用层之间的网络带宽，从数据库中直接发送大块未处理数据给客户软件是不明智的。在中间层中处理这些数据将会揭示UI层只需要它们的一个子集是非常有可能的。

如果一个分布式应用中的值对象数量正变得有问题，把杂凑图发回给客户软件对使用行集来说更可取，而且实现起来不会太困难。但是，在我们面对遍历关联到什么程度这一复杂问题时，这种杂凑图方法一点帮助也没有。例如，我们可以有条件地在一个杂凑图中包含一个对象的关联字段，但如果只需要遍历这些字段的关联直到某一深度，就会发现问题。在这种情况下，我们通常需要借助于编写自定义的值对象，而且最终有可能需要实现并维护大量的值对象。

常见的数据存取问题

不管我们使用什么数据存取策略——和什么类型的数据库，都将会遇到一些常见问题。下面要讨论的这些问题对可缩放性和性能将会有重要的影响。

事务隔离

事务隔离级别（transaction isolation level）决定数据库行为，但由J2EE服务器来管理，因为J2EE服务器负责全面的事务协调。事务隔离级别控制着访问同一个数据库的并发事务之间的隔离程度。SQL92定义了JDBC API中也支持的4种事务隔离级别。按照隔离的降序排列，这些事务隔离级别分别是：

- TRANSACTION_SERIALIZABLE
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_READ_COMMITTED

- TRANSACTION_READ_UNCOMMITTED

用于一个事务的隔离级别越高，这个事务的工作将不被其他并发事务破坏的保障就越大，但是这个事务对该资源的总吞吐量的影响也将越大。就关系数据库的情况来说，事务隔离级别将控制行或表的加锁（要想了解SQL92事务隔离级别的准确而充分的概述，请参见<http://www.onjava.com/pub/a/onjava/2001/05/23/j2ee.html?page=2>）。

事务隔离级别对数据完整性和性能会有极大的影响。可是，它们的准确含义在数据库之间是不同的。这也是透明持久性难以捉摸的一个原因。

大多数数据库不支持所有4种事务隔离级别，因而意味着一个J2EE服务器可能无法保证预期的行为。例如，Oracle不支持TRANSACTION_READ_UNCOMMITTED隔离级别。不过，Oracle保证非阻塞读，而非堵塞读保持了该READ_UNCOMMITTED隔离级别的意图。

在EJB容器的外部，我们将需要使用API特有的程序性事务隔离控制，比如java.sql.Connection接口上的setTransactionIsolation()方法。

虽然EJB规范没有规定EJB容器应该怎样控制事务的方式，但我们应该能够为使用CMT的EJB而声明性地设置它。这是使用带有CMT的EJB来协调数据存取的另外一个优点。我们可以使用标准ejb-jar.xml部署描述符来告诉任何容器何时开始以及怎样传播事务，但没有告诉该容器一个事务究竟表示什么意思的标准方法。

通常，这在一个容器特有的部署描述符中被指定。例如，在WebLogic中，weblogic-ejb-jar.xml文件中有一个看起来像下面这样的元素：

```
<transaction-isolation>
  <isolation-level>TRANSACTION_SERIALIZABLE</isolation-level>
  <method>
    <ejb-name>Booking</ejb-name>
    <method-name>*</method-name>
  </method>
</transaction-isolation>
```

如果读者的EJB容器允许，应该总是为EJB方法指定事务隔离级别。否则，你正在依赖目标服务器的默认事务隔离级别，而且已经不必要地损害了可移植性。

保守式与开放式加锁

处理数据库并发性有两种基本策略：保守式和开放式加锁。保守式加锁采取这样一种“保守”的观点：用户很可能会破坏彼此的数据，并且唯一安全的选择是串行化数据存取，以便某一时刻最多只有一个用户拥有任一数据的控制权。这保证了数据完整性，但会严重地降低系统能够支持的并发活动量。

开放式加锁采取这样一种“乐观”的观点：这类数据冲突将极少发生，所以允许并发存取比锁定并发更新更重要。关键是我们不能允许用户破坏彼此的数据，因此，如果出现并发更新的尝试，我们就会有问题。我们必须能够检测到竞争性更新，并使某些更新失败，以便保护数据完整性。一个更新必须被拒绝，如果要被更新的数据自预期更新者读取它以来已经更改。这意味着我们需要一种策略来检测这样的更改；通常，使用一个开放式加锁属性(optimistic locking attribute)：对象或行中的、每当对象或行被修改时就递增一次的一个

版本号。

事务隔离级别可以用来实施保守式加锁。可是，EJB规范根本就没有提供开放式加锁支持，尽管实体组件CMP的特殊实现可以提供这种支持，而且许多O/R映射产品也提供这种支持。要想实现开放式加锁，我们一般需要编写代码来检查开放式加锁属性，并在检测到不一致时需要回退事务。

针对某一个特定情景，选择保守式还是开放式加锁取决于业务需求以及底层数据库。

和事务隔离级别一样，加锁在不同的数据库中也有非常大的差别。笔者强烈建议读者阅读一个关于所用数据库的好参考书，例如，“**Expert One-on-One Oracle**（Wrox出版，ISBN：1-86100-4-826）”一书清楚地解释了与Oracle有关的这些问题。

主键生成策略

持久性存储器要求对象拥有惟一性关键字（即RDBMS术语中的主键）。在每次插入数据时，我们可能需要生成一个新的主键。

下面的讨论使用代用键（surrogate key）：没有业务含义的主键。如果一个主键是一个具有业务含义的字段，比如电子邮件或具有业务含义的一些字段的一种组合，这个主键就称做智能键（intelligent key）或自然键（natural key），并且没有必要生成新的键——它们被创建为新数据的一部分。

尽管对每种类型的键有争论，但代用键在企业系统中是十分常见的。代用键更容易处理，并且可以产生更好的性能，以及不受业务规则变化的影响（例如，假设电子邮件被用做了一个User表的自然键，如果用户被允许共享电子邮件地址，则该表将需要一个新键）。代用键绝不能暴露给操作；如果它们拥有了业务含义，它们就拥有了自然键的那些缺点。关于智能键和代用键正反两方面的讨论，请参见<http://www.bcarter.com/intsurrl.htm>。

代用键生成的问题不是J2EE特有的。它通常通过自动增值列或其他数据库特有方法来解决。这些键不是可移植的，因此遭到许多J2EE开发人员的怀疑。它们从实体组件或JDBC中使用起来也会很困难。

EJB容器常常提供一种为CMP实体组件生成主键的方法；虽然EJB规范中根本没有定义标准方法（一个令人失望的疏忽）。当有一个这样的机制可供使用时，如果正在使用实体组件，最好是使用该机制。请查阅目标服务器的文档，并了解实体组件CMP的描述。

当和BMP一起使用实体组件时，或者当使用JDBC或另一个低级API实现持久性时，会遇到键生成问题。实体组件BMP的约定要求ejbCreate()方法在实体创建时返回新的主键。这意味着我们不能依赖于自动增值键列之类的数据库功能度。问题是根本没有办法知道刚插入了哪一行。我们可以试着根据刚插入的数据值做一个SELECT操作，但这是很笨拙的，而且不保证管用。为什么逻辑上不同的行中除了主键之外会存在重复数据呢？这可能就是一个站得住脚的原因。惟一的替代方案是在插入之前就生成一个惟一性主键，并在该插入中包括这个新主键。

需要注意的是，未必总是会遇到主键生成问题，除非我们正在使用实体组件。未必总是有必要知道一个新插入行的主键。

在TheServerSide.com之类的论坛上，有关于怎样使用一种在数据库之间实现可移植性的方法来生成主键的大量讨论。笔者觉得，这容易让人产生这样的误解：可移植性的价值大于简单性的价值。

不要试图用一种在数据库之间具有可移植性的方法生成惟一性主键来增加复杂性，而是应该使用目标数据库的相关特性，然后隔离实现特有的特性，以便使设计保持可移植。

如果一个应用被移植到另外一个数据库，面对一项简单的重新实现任务总比增加额外的复杂性来保证总代码可移植性强得多。第一种替代方案意味着始终有一个简单、易懂的代码库。

用于主键生成的推荐策略通常有3种：

- 使用一个实体组件表示来自数据库的一个序列
- 从Java内生成一个惟一性ID
- 使用一个生成主键的存储过程或其他数据库特有手段

前两种解决方法是可移植的。下面依次看一看上述每种解决方法。

序列实体组件

这种方法使用数字键。每当一个客户软件请求一个新键时，一个实体组件就把一个单独的数据存储在数据库中，并递增它。例如，我们可能有如下所示的RDBMS表：

KEYS	
NAME	CURRENT_VALUE
USER	109

实体组件实例可以封装每个实体组件的主键。每当一个键被请求时，CURRENT_VALUE列中的值就被更新一次。

这种方法举例说明了认为一个可移植解决方案就是一个好解决方案的危险。只有通过插入一个定期从该组件中申请一批键的会话组件，这种方法才能充分发挥作用。如果每当一个新键被请求时该实体就被使用一次，键生成将会产生一个争用点，因为生成一个键要求CURRENT_VALUE列拥有更新和串行化KEYS的访问权。这意味着我们需要部署两个EJB来纯粹地处理主键生成：它们对实现该应用的业务逻辑不起一点作用。这种方法是行不通的。

Java中的惟一性ID生成

在这种方法中，代用键生成完全是在数据库外部进行的。我们使用这样一个算法：它保证每当一个键被生成时，这个键必须是惟一的。这样的算法通常基于一个随机数、系统时间和服务器的IP地址的一种组合。这里的难题是：我们可能正运行在一个聚类中，所以必须保证各服务器决不会生成相同的键；以及Java只能使我们查找系统时间到毫秒级（由于这一原因，Java不是一种用来生成惟一性ID的合适语言）。

惟一性ID生成涉及到一个算法，而不是涉及到状态，所以不会遇到在EJB层中使用元素集的问题，因此我们不必使用额外的EJB来实现它。

Java中的惟一性ID生成是快速的和可移植的。但是，它有下列缺点：

- 它生成很长的键，无论该表将保存多少行；
- 键是字符串，不是数，因而可能会使数据库中的索引变得复杂；
- 键不是顺序的。

这种方法忽略RDBMS能力，并硬要在Java能力弱的地方使用它。

数据库特有的ID生成

依据笔者的看法，生成主键的最佳方法是使用数据库特有的功能度。由于主键创建是一个根本性的数据库问题，所以每个数据库都提供了一种在多个用户创建行时最大限度地减少争用的快速方法。通常，这将需要通过调用一个存储过程，而不是通过运行一个SQL INSERT来执行更新。需要记住的是，由于我们需要知道新插入行的ID，所以在INSERT运行时，我们不能只依赖于一个自动增值列或一个插入触发器来创建一个新主键。

该存储过程必须接受用于新行的所有数据值的输入参数，以及一个将使它能够把生成后的主键返回给JDBC调用者的输出参数。它必须插入这些新数据，然后返回新主键。视数据库而定，在INSERT发生时主键可能会自动得到创建（例如，如果主键列是一个自动增值列）；主键也可能会在插入之前先得到创建。这种方法将给我们可移植的JDBC代码，但将需要为该存储过程编写数据库特有的代码。

下列例子使用Oracle说明了这种方法（简单地把该代码键入到SQL*Plus中试一试它）。

下面来看一看当我们给下列这个简单的表添加行时如何生成键，其中这个表除了主键之外仅有一个单独的数据值：

```
CREATE TABLE person (
    id NUMERIC PRIMARY KEY,
    name VARCHAR(32)
);
```

Oracle使用序列代替自动增值列，所以我们需要定义一个能够用于这个表的序列：

```
CREATE SEQUENCE person_seq
    start WITH 1
    increment BY 1
    nomaxvalue;
```

现在，我们需要编写一个存储过程来做INSERT；在Oracle中，笔者使用了PL/SQL。需要注意的是，这个存储过程有一个输出参数，我们将使用它把新的id返回给调用的JDBC代码：

```
CREATE OR REPLACE
PROCEDURE person_add(p_name in varchar, p_id out number)
AS
BEGIN
    SELECT person_seq.nextval INTO p_id FROM dual;
    INSERT INTO person(id, name) VALUES(p_id, p_name);
END;
/
```

我们已经在数据库内处理了Oracle特有的代码，但我们用来调用这个存储过程的JDBC代码是可移植的。无论我们使用了什么RDBMS，需要做的只是保证我们有一个名为

person_add且具有相同参数的存储过程。在现实生活中，将会有许多输入参数，但始终只有一个输出参数。

首先，我们必须调用该存储过程。这与调用一个PreparedStatement十分类似：

```
Connection con = cf.getConnection();
CallableStatement call = con.prepareCall("{call person_add(?, ?)}");
call.setString(1, "Frodo");
call.registerOutParameter(2, java.sql.Types.INTEGER);
call.execute();
```

现在，我们可以抽取出输出参数的值：

```
int pk = call.getInt(2);
System.out.println("The primary key for the new row was " + pk);
call.close();
```

这种方法给了我们简单、可移植的Java代码，但将要求我们实现一个新的存储过程，如果我们在某个时候变换了数据库。但是，由于该存储过程是如此简单，所以不太可能会有问题。该存储过程实际上是我们Java代码中所定义的一个接口的实现。

JDBC 3.0

当使用一个支持JDBC 3.0的数据库驱动程序时，还有另外一个选项可用。JDBC 3.0 Statement接口允许我们获得由一个插入操作所创建的那些键。下面是使用这个功能的一个简单例子：

```
Statement stmt = connection.createStatement();
stmt.executeUpdate("INSERT INTO USERS (FIRST_NAME, LAST_NAME) " +
    "VALUES ('Rod', 'Johnson')", Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();

if (rs.next()) {
    int key = rs.getInt(1);
}
```

这解决了主键生成问题，如果该数据库有一个创建新主键的触发器，或者主键是一个自动增值列。可是，JDBC 3.0支持至今还不十分普及，所以这个策略通常不是一个可选项。要想在一个J2EE应用中使用这个策略，数据库供应商和应用服务器都将需要支持JDBC 3.0，因为应用代码由于底层数据库连接的缘故可能被赋予了服务器特有的封装器连接。

JDBC 3.0在J2EE 1.4中是一个必需的API，所以使用JDBC的J2EE 1.4应用将被保证能够知道新插入行的主键，因而使主键生成能够被隐藏在RDBMS的内部。

何处执行数据存取

由于我们已经了解了一些主要的数据存取问题，现在让我们来看一看在J2EE应用内的什么地方执行数据存取这一关键性的体系结构决定。

首先，让我们来纵览一下在J2EE中访问数据源的各种可选项。我们将在以后的几章中较详细地讨论各种推荐的可选项。

EJB层中的数据存取

大多数关于J2EE的书籍都推荐从EJB中执行数据存取。在分布式J2EE应用中，这是合理的建议；在这种情况下，分布式Web容器中的代码应该只依赖于直接位于这些容器（那些远程EJB）下面的体系结构层，并且不独立地与数据源进行通信。可是，在集中式应用中，有一种从EJB中执行数据存取的不太有说服力的情况。

从EJB中执行数据存取的优点是：

- 我们有使用实体组件的选择余地。
- 我们可以从CMT中获得好处。事务管理对大多数数据存取来说是不可或缺的，所以这是一个重要的优点。
- 所有数据存取将通过一个单独的层来进行。

从EJB中执行数据存取的缺点是：我们在前文中已经讨论过的EJB所具有的所有缺点（比如复杂性），再加上数据存取代码测试起来可能会很困难（因为该代码依赖于EJB容器）等重要考虑因素。下面让我们来看一看EJB层中数据存取的各种选择。

实体EJB

一个实体组件就是一个用于支持访问持久性数据的标准J2EE构件。一个实体组件层使分离业务逻辑与持久性逻辑的良好设计习惯变成规范化。它独立于底层数据源（至少在理论上）。无论我们使用什么数据源，一个实体组件层都使它具体化。在这种模型中，实体组件将暴露本地接口，而业务对象通常将是会话组件。不过，布置在同一个JVM中的任何对象都能通过它们的本地接口使用那些实体组件。

实体组件是一个非常基本的O/R映射解决方案。虽然EJB规范保证它们是可以使用的，但它们忽视了我们在前面所列举的大多数O/R映射问题。对象继承性是禁用的。数据只能来自单个表。如果我们不编写自定义代码，实体组件不允许我们访问数据库中的集合函数，这实际上破坏了它们之所以存在而提供的抽象。可是，EJB 2.0 CMP确实不支持对象关系（EJB 1.1不支持）。

EJB 1.1中的实体组件具有有限的功能度，而且与使用EJB 2.0 CMP约定所编写的实体组件相比，具有较差的性能。由于CMP 2.0约定不同于CMP 1.0约定，所以CMP实体要求修改代码后才能从EJB 1.1迁移到EJB 2.0。当使用不支持EJB 2.0的EJB容器时，不要使用实体组件。

虽然EJB规范对于数据存取是特殊的帮助，但实体组件或许是J2EE中最有争议的技术。我们将在下一章中较详细地讨论使用实体组件的正反两方面，但下列这些缺点排除了使用它们来解决许多J2EE数据存取问题（这些缺点中的部分缺点适用于任一简单的O/R映射）的可能：

- 使用实体组件通过分离业务逻辑与数据存取反映了合理的设计原理，但也使我们受困于这种方法的一个特定实现，因而将影响业务对象中的代码，处理任何类型的EJB要比处理普通Java对象复杂得多。
- 它们是EJB惟一可以利用的持久性策略，所以我们被迫使用EJB，无论它在其他方面是否合适。

- BMP实体组件具有严重而又很难解决的性能问题，这些问题将在下一章中讨论，并且妨碍了它们在许多情况中的使用。
- 实体组件生存周期是很死板的。这使得实现粗糙实体组件变得很困难。
- 使用实体组件可能会使我们束缚于一个RDBMS模式。如果我们使用从实体组件到RDBMS表的一对一映射，并且RDBMS模式在某个时候又发生了变化，我们将会有大量工作要做。
- 使用精细实体组件使得有效使用关系数据库变得很困难；例如，用同一条语句更新多个行将会很困难，因为实体组件把我们推向了映射到个别行的方向。
- 与已得到公认的O/R映射解决方案相比，实体组件（即便使用了EJB 2.0 CMP）只提供非常基本的功能度。
- 大多数EJB容器中都有与实体组件的性能相关的严重问题。
- 实体组件解决方案的可移植性优点是有争议的。加锁和并发性行为在主要数据库产品之间的差别很大。假设EJB容器一般把这些问题留给底层数据库来提供更大的吞吐量，那么实体组件所提供的抽象实际上可能会令人产生误解，如果我们移植它的话。
- 与使用会话组件的回报（透明线程、安全及有限部署复杂性的事务管理）相比，采用实体组件会提供一个更有疑问的回报，因为部署复杂性明显更大。

在把实体组件用于数据存取时，实质性矛盾是EJB（由于存取它们的复杂性和它们较高重量级的特性）最好被看做构件（component），而不要被看做普通Java对象。数据存取通常需要比EJB能够实际提供的粒度还要精细的粒度。

实体组件在有些J2EE体系结构中会很有效，但它们不是业务逻辑与物理数据存取之间的理想抽象。如果会话组件直接访问实体组件，会话组件就会有束缚于某一特定RDBMS模式的危险。如果性能有问题，使用另外一种方法来替换实体组件层也会很困难。

会话EJB与助手类

即使我们选择使用EJB并在EJB中实现业务逻辑，也不会被迫为数据存取使用实体组件。会话EJB是业务逻辑构件。这意味着它们不应该处理数据存取的细节。

但是，这保留了使用会话组件来控制数据存取和使用DAO模式来抽象数据存取细节的选择。这是一种非常灵活的方法。和使用实体组件（使我们束缚于一种方法）不同，我们仍有如何处理持久性的控制权。DAO可以使用任何持久性API，比如JDO、JDBC、专有映射框架或实体组件。这种灵活性（以及同实体组件基础结构相比降低的系统开销）常常产生比实体组件更好的性能。

从会话组件中与从其他类型的业务对象中使用DAO之间的惟一区别是声明性事务管理的可得到性。通常，相同的DAO在一个EJB容器的内部或外部都工作。

为了与实体组件CMP和BMP相对应，这种方法有时叫做Session Managed Persistence（SMP，会话管理式持久性）。笔者不太喜欢这个术语，因为它暗指持久性代码被实际包含在会话组件中——一种拙劣的选择。另外，各种DAO方法不束缚于EJB的使用。

不使用EJB时中间层中的数据存取

正如我们已经见过的，J2EE Web应用可以拥有另外一个由未使用EJB的业务对象所构成的中间层（在具有远程客户软件的分布式应用中，EJB是用于中间层的唯一选择）。这意味着我们可以拥有运行于一个J2EE Web容器内，但不是Web特有的中间层业务对象（事实上，它们也可以通过Web服务来支持远程访问）。这样的对象可以利用JTA来访问程序性的全局事务管理，并且可以访问连接池（和EJB服务器一样，J2EE Web容器提供连接池——用户代码可以利用JNDI查找来访问它们。这意味着我们不必使用EJB容器来访问企业数据）。

这样的业务对象能够使用J2EE可以利用的任一数据存取策略。一般说来，它们将使用一个像JDO或TopLink那样的O/R映射框架或一个像JDBC那样的资源特有API。它们没有声明性事务管理的选择，但这种方法提供好的性能（因为它避免了借助JNDI或O/R映射框架来访问一个JDBC数据源的能力）。这样的需求可以通过测试工具来轻松地得到满足，这与EJB容器服务不同。

在下列情形中，这种未使用EJB的数据存取对集中式应用来说是一个很好的选择：

- 我们不想使用EJB。正如我们已经见过的，EJB的数据存取好处没有实际证明在EJB不被看好的地方使用EJB是正确的。
- 数据存取是非事务性的。在这种情况下，EJB CMP不提供一点好处。
- 和数据存取有关的一些功能度（比如数据高速缓存需求）与应用于EJB的那些程序设计限制相冲突。在这种情况下，把数据存取实现和高速缓存功能度结合起来可能会比把数据存取放在EJB层中并把数据高速缓存在Web容器中更简单。

在集中式应用中，没有必要在同一个地方执行所有数据存取。例如，从EJB中（为了利用CMT）执行事务性数据存取和从Web容器内的对象中（因为更简单和更快速）执行非事务性数据存取都是合法的。我们将在本书的示例应用中采用这种混合方法。

Web层中的数据存取

相反，Web层（涉及到表示一个Web接口的逻辑体系结构层）中的数据存取是一个拙劣的设计选择。运行于Web容器中的业务对象应该有别于将依赖于Servlet API的Web层构件。

小服务程序与Web特有类

从Web特有的类中执行数据存取没有任何正当的依据。数据存取对象应该束缚于Servlet API也没有任何理由。这将会妨碍它们在一个EJB容器中的使用（如果必要）。UI代码也应该避开数据源实现问题。例如，修改一个RDBMS中的一个表或列名绝不应该破坏一个Web接口。从Web特有的类中执行数据存取是没有必要的，因为应该存在一个由它们可以采用的业务对象所构成的层。

从JSP页面中进行的数据存取

但是，还有一个更糟糕的数据存取选择。J2EE系统中最没有吸引力的数据存取选择是从JSP页面中执行数据存取。该选择往往会导致错误处理方面的问题，以及失去业务逻辑。

数据存取与表示这三者之间的区别。但是，这是十分普遍的，因此我们不能忽视。

在JSP的早期，JSP“Model 1”系统十分常见。许多开发人员试图用JSP页面来完全取代小服务程序，部分是由于增强后的创作便利性。JSP页面常常含有数据存取代码，连同该应用的大部分其余代码。结果是非常可怕的。

现在，MVC概念在Java Web应用中的价值得到了广泛认可（我们将在第12章和第13章中详细讨论这方面的内容）。不过，随着自定义标记在JSP 1.1中的引进，由于它们在其他方面受到欢迎，所以从JSP页面中执行数据存取再一次抬起它丑陋的头颅。

从自定义标记中执行JDBC存取表面上很吸引人，因为它是有效和方便的。请看一看摘自JSP Standard Tag Library规范中的下列JSP代码段，该代码段使用两个SQL更新从一个账户中转移一笔钱到另一个账户。我们将在第13章中讨论JSP STL Expression Language。\${}语法用来访问页面上已定义的变量：

```
<sql:transaction dataSource="${dataSource}">
  <sql:update>
    UPDATE account
    SET Balance = Balance - ?
    WHERE accountNo = ?
    <sql:param value="${transferAmount}" />
    <sql:param value="${accountFrom}" />
  </sql:update>
  <sql:update>
    UPDATE account
    SET Balance = Balance + ?
    WHERE accountNo = ?
    <sql:param value="${transferAmount}" />
    <sql:param value="${accountTo}" />
  </sql:update>
</sql:transaction>
```

现在来看一看一个JSP所违反的一些设计原理和它可能产生的各种问题：

- JSP源未能反映它将生成的那个动态页面的结构。上面所示的那16行代码无疑是包含它们的一个JSP的最重要部分，但它们根本不生成任何内容。
- （仅分布式应用才有）降低的部署灵活性。由于Web层依赖于数据库，所以它不仅需要能够与应用的EJB层进行通信，而且还需要能够与数据库进行通信。
- 崩溃的错误处理。直到我们遇到一个错误（比如与数据库进行通信发生故障）时，我们才知道实现一个特定的视图。最好的结果是，我们最终将会得到一个通用错误页面；最坏的结果是，在我们遇到该错误之前，缓冲区已被清洗掉，并且我们将会得到一个已坏了的页面。
- 需要在一个JSP中执行事务管理，以保证更新要么全部发生，要么根本不发生。事务管理应该是中间层对象的责任。
- 对业务逻辑应该放在中间层中这一原理的破坏。根本不存在中间层对象的支持层。根本没有把本页面中所包含的业务逻辑暴露给非Web客户端软件或Web服务客户端软件的方法。
- 执行单元测试的无能为力，因为JSP根本不暴露业务接口。
- 页面生成与数据结构直接的紧密耦合。如果一个应用使用这种方法，并且数据库模

式发生变化，那么许多JSP页面很可能需要更新。

- 内容表示的紊乱。如果我们要暴露本页面用PDF（JSP不能生成的一种二进制格式）描述的数据，怎么办？如果我们需要转换数据到XML，并用XSLT格式表变换它，怎么办？我们将需要重复数据存取代码。封装在数据库更新中的业务功能度被捆绑到JSP——一种特殊的视图策略。

如果存在从JSP页面中使用标志库进行数据存取的任何一个地方，那么这个地方必然是在无足轻重的系统或原型中（JSP标准标志库的那些创作者都同意这个观点）。

永远不要从JSP页面中执行数据存取。JSP页面是视图构件。

小结

在本章中，我们已经了解了J2EE系统中数据存取方面的一些关键问题。我们已经讨论了如下问题：

- 业务逻辑与持久性逻辑之间的区别。虽然业务逻辑应该由Java业务对象来处理，但持久性逻辑可以在多种多样的J2EE构件甚至数据库中合法地实现。
- 对象驱动与数据库驱动数据模型之间的选择，以及数据库驱动建模为什么常常更可取。
- 使用关系数据库的各种难题。
- O/R映射概念。
- 使用Data Access Object (DAO)（即普通Java接口）来提供由业务对象使用的一个数据存取抽象。一种DAO方法不同于一种O/R映射方法，因为它由动词（“禁用所有智利用户的账户”）构成，而不是由名称（“这是一个User对象；如果我设置数据库将被透明地更新的一个属性”）构成。但是，它不妨碍O/R映射的使用。
- 分布式应用中的数据交换。我们讨论了Value Object J2EE模式，该模式把多个数据值合并成一个单一的可串行化对象，以最大限度地减少必需的高代价远程调用的数量。我们考虑了多个值对象满足不同用例的各种需求的可能需要，并考虑了类型化值对象的、远程调用者有多种多样的需求时可能会适用的通用替代方案。
- 生成主键的策略。
- 在J2EE系统内的什么地方实现数据存取。我们得出的结论是：数据存取应该在EJB或中间层业务对象中进行，以及实体组件不是唯一方法。虽然中间层业务对象可能会实际运行在一个Web容器中，但我们看到，从小服务程序和JSP页面之类的Web特有构件中进行数据存取不是切实可行的。

笔者已经列举理由证明可移植性在数据存取中常常被列入最优先地位。设计的可移植性有很大的重要性；试图实现代码的可移植性常常是有害的。一个有效、简单并且在数据库发生变化时只需要重新实现少量持久性代码的解决方案，可以比一个低效率、不自然但100%可移植的解决方案创造更高的业务价值。XP的教训之一是：今天试图解决明天的问题常常是一个错误，如果这么做会增加第一个实例中的复杂性。

示例应用中的数据建模

在上述讨论之后，让我们来看一看示例应用中的数据存取。

Unicorn集团已经使用了Oracle 8.1.7i。其他报告制作工具很可能会使用该数据库，并且第一阶段中的有些管理任务很可能也会利用数据库特有工具来完成。因此，数据库驱动（而不是对象驱动）建模是适用的（现有售票处应用的部分数据模式可能还是可重用的）。

本书的讨论主题不是数据库设计，并且笔者也没有自称是一名专家，因此我们将快速浏览一遍数据模式。在实际的项目中，DBA在开发中将会起到非常重要的作用。示例应用中的数据模式将反映下列数据需求：

- 将有许多艺术流派，比如Musical、Opera、Ballet、Circus等。
- 每种流派中将有许多节目。给每个节目都关联一个HTML文档必须是可能的，进而让该文档含有与要演出的作品、角色等有关的信息。
- 每个节目都有一个座位安排计划。一个座位安排计划描述固定数量的待售座位，其中座位分成一种或多种座位类型，每种类型都关联一个能够被显示给顾客的名称（比如Premium Reserve）和代码（比如AA）。
- 每个节目都有多场演出。每场演出都将有一个价格结构，该价格结构将给每种类型的座位分配一个价格。
- 虽然每个节目都有各自的座位安排计划和每场演出都有各自的价格结构是可能的，但有些节目将针对同一个演出大厅使用默认的座位安排计划和一个节目的所有演出都将使用同一个价格结构是可能的。
- 用户可以创建占据某一场演出的许多座位的预订。这些预订可以进展到有效信用卡细节提交时的确认（座位购买）。

首先，我们必须决定在数据库中保存什么内容。该数据库应该是中心数据贮藏库，但它不是存储HTML内容的好地方。HTML内容是没有事务性需求的参考数据，所以能被看做该Web应用的一部分，并保持在其目录结构的外面。然后，该内容无需访问和修改数据库就能由HTML编码器来修改。当再现Web接口时，我们可以用相关记录的主键从数据库中轻松地查找到相关资源（座位安排计划图和节目信息）。例如，对于主键1的座位安排计划可能被保存在该Web应用内的/images/seatingplans/1.jpg处。

在这种情况下，实体EJB之类的O/R映射方法不会有什么好处。O/R建模方法通常是为读取-修改-写入情况而设计的。在示例应用中，我们有一些参考数据（比如艺术流派和节目数据），并且这些数据从不通过Internet User或Box Office User接口来修改。这样的只读参考数据可以使用JDBC来轻松而有效地获得；O/R方法可能会增加不必要的系统开销。除了访问参考数据之外，该应用还需要创建订票记录来表示用户的座位预订，以及在用户确认他们的预订时创建购买记录。

这种动态数据也不太适合O/R建模，因为高速缓存这种动态数据没有什么价值。例如，一条订票记录的细节将只在用户完成订票过程时显示一次。这种数据被再次需要的可能性几乎不存在，除了作为一个定期报告制作过程的一部分之外，而这个过程可能是打印和邮寄入场券。

由于我们知道该组织已经决定使用Oracle，所以需要利用任何有用的Oracle特性。例如，我们可以使用Oracle Index Organized Tables (IOT) 来改进性能；可以使用PL/SQL存储过程；以及可以使用Oracle数据类型，比如Oracle date类型——在Java中处理起来很容易的一种日期/时间组合值（标准SQL和大多数其他数据库使用分开的日期和时间对象）。

这两个考虑因素都意味着DAO模式的使用，其中JDBC作为第一个实现选择（我们将在第8章中讨论如何不降低可维护性来使用JDBC）。JDBC在涉及只读数据以及O/R映射层中的高速缓存将不产生任何好处的情况下会产生卓越的性能。使用JDBC还将使我们能够利用专有Oracle特性，同时又不使我们的设计束缚于Oracle。DAO可以使用一种替代策略来实现，如果示例应用在某个时候需要使用另一个数据库。

图7.3是一个E-R图，描述了一个适用的数据模式。

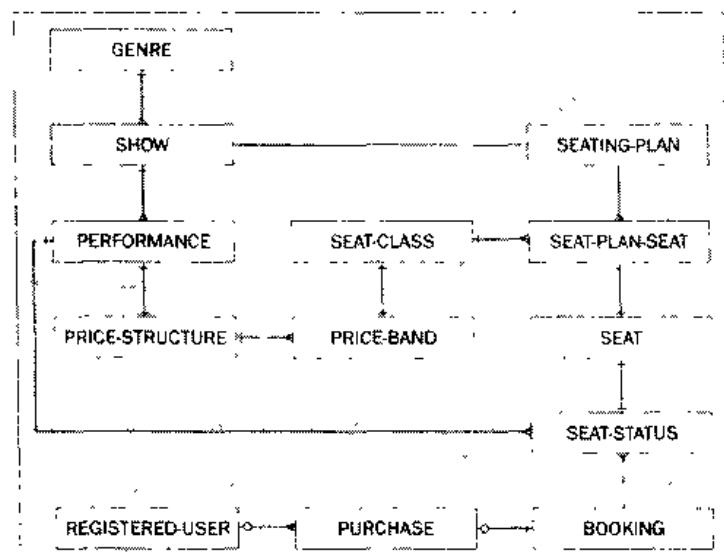


图7.3

该DDL文件（create_ticket.dll）被包含在配书下载的/db目录中。在接下来的讨论中，如果需要，请参考这个文件。

这些表可以分成参考数据和动态数据。除了SEAT_STATUS、BOOKING、PURCHASE和REGISTERED_USER表之外的所有表本质上都是参考数据，只能由Admin角色功能度来更新。这个模式中的大部分复杂性将不会直接影响该Web应用。每个节目关联了一个座位安排计划，该计划可能是一个针对相应大厅的一个标准座位安排计划，也可能是一个自定义座位安排计划。SEAT_PLAN_SEAT表把一个座位计划与该表所含有的那些座位关联了起来。不同的座位安排计划可能包含了一些相同的座位；例如，一个座位安排计划可能删除了许多座位或更改了被认为相邻的座位。座位安排计划信息可以只被装入一次，并被高速缓存在Java代码中。然后，将没有必要运行另外的查询来确定哪些座位相邻等。

在动态数据当中，BOOKING表中的行可能代表一个座位预订（将生存一段固定时间），也可能代表一个座位购买（在这种情况下，它有一个指向PURCHASE表的参考）。

SEAT_STATUS表是最有趣的，反映了该数据模型的一个微小范式化降低。如果我们为每个被预订或购买的座位都只创建了一条新的座位预订记录，则可以通过查询来确定哪些座

位仍是空闲的（基于本场演出的座位——通过相应的座位安排计划来获得），但是这将需要一个复杂而且可能还很慢的查询。然而，SEAT_STATUS表可以被预先填入：每场演出中的每个座位占用该表的一行。每一行都有一个指向BOOKING表的可空值参考；这将在一个预订或订票被执行时得到设置。SEAT_STATUS表的填入将被隐藏在数据库内；一个触发器（这里没有给出）用在PERFORMANCE表中被添加或删除行时添加或删除行。

SEAT_STATUS表的定义如下所示：

```
CREATE TABLE seat_status (
    performance_id NUMERIC NOT NULL REFERENCES performance,
    seat_id NUMERIC NOT NULL REFERENCES seat,
    price_band_id NUMERIC NOT NULL REFERENCES price_band,
    booking_id NUMERIC REFERENCES booking,
    PRIMARY KEY(performance_id, seat_id)
)
organization index;
```

其中，price_band_id也是座位类型的id。需要注意一个Oracle IOT的使用，它被指定在最后的organization index子句中。

降低范式化的合理性基于以下理由：

- 在数据库中实现起来容易，而又简化了查询和存储过程。
- 通过避免复杂连接改进了性能。
- 结果数据重复在这种情况下不是一个严重问题。重复的程度预先是知道的。正被重复的数据是不能变的，所以不会失去同步。
- 将避免SEAT_STATUS表中的插入和删除，进而用更新取代了这些操作。插入和删除可能会比更新有更高的代价，所以这将改进性能。
- 使将来添加可能必需的功能度变得更容易。例如，通过在SEAT_STATUS表中添加一个新列从销售中删去一些座位将是很容易的。

除了检查SEAT_STATUS表之外，检查BOOKING表来确认一个座位是不是可获得的是必不可少的，但没有必要遍历参考数据表。没有一个订票参考的一个SEAT_STATUS行始终表明一个可获得的座位，但带有一个订票参考的一个SEAT_STATUS行可能也表明一个可获得的座位，如果该订票没有得到确认而已经期满。我们需要执行一个与BOOKING表的外连接(outer join)来确定这一点，即执行这样一个查询：它包括了指向BOOKING表的外键是空值的行以及BOOKING表中的相关行指出一个期满预订的行。

Java代码（即便在DAO中）没有理由应该知道这个模式的所有细节。笔者已经做了几个决定，以便向Java代码藏起该模式的部分复杂性，以及把数据管理的一部分隐藏在数据库内。例如：

- 笔者使用了一个序列和一个存储过程来处理预订（我们在本章的前面已经讨论过的一种方法）。这插入到BOOKING表中，更新SEAT_STATUS表，并把用于新增订票对象的主键作为一个输出参数返回。使用该存储过程的Java代码不必知道进行一个预订将涉及到更新两个表。
- 笔者使用了一个触发器来设置PURCHASE表中的purchase_date列为系统日期，以便插入到这个表中的Java代码不必设置日期。这保证了数据完整性，并潜在地简化了Java代码。

- 笔者使用了一个视图来暴露座位安排可获得性并隐藏掉需要与BOOKING表进行的外连接。这个视图不必是可更新的；我们只需把它作为一个存储查询对待即可（不过，仅做查询的Java代码不必区分视图与表）。虽然该视图中的那些行仅来自SEAT_STATUS表，但不可得到的座位将被排斥在外。Oracle视图定义如下所示：

```

CREATE OR REPLACE
VIEW available_seats AS
SELECT seat_status.seat_id, seat_status.performance_id,
seat_status.price_band_id
FROM seat_status, booking
WHERE
    booking.authorization_code is NULL
    AND (booking.reserved_until is NULL or
        booking.reserved_until < sysdate)
    AND seat_status.booking_id = booking.id(+);

```

使用这个视图，我们可以非常简单地查询某一给定类型的空座位：

```

SELECT seat_id
FROM available_seats
WHERE performance_id = ? AND price_band_id = ?

```

这种方法的优点是：Oracle特有的外连接语法对Java代码来说是不可见的（我们可以在另一个数据库中使用不同的语法实现相同的视图），Java代码更简单，以及持久性逻辑由数据库来处理。Java代码没有必要知道订票是怎样表示的。虽然该数据库模式一旦含有真实的用户数据时将被更改是不可能的，但使用这种方法，它可以被更改，同时Java代码又不受影响。

Oracle 9i也支持用于外连接的标准SQL语法。但是，要求是该应用使用Oracle 8.1.7i。

在所有这些情况中，该数据库只含有持久性逻辑。对业务规则的更改不会影响该数据库中所含有的代码。数据库擅长使用触发器、存储过程、视图和此类东西来处理持久性逻辑，所以这将会产生一个更简单的应用。从本质上讲，我们有两份把业务对象与数据库分离开的契约：Java代码中的DAO接口，以及存储过程签名和DAO所使用的那些表和视图。这些相当于该数据库暴露给该J2EE应用的公共接口。

在接着实现该应用的其余部分之外，测试这个模式的性能（比如常见查询将运行得多快）以及并发使用下的行为是十分重要的。由于这是数据库特有的，所以笔者在这里将不做这方面讨论。但是，它是整个应用的综合测试策略的一部分。

最后，我们需要考虑我们希望应用的加锁策略——保守式还是开放式加锁。当用户试图预订同一场演出的相同类型的座位时，加锁将会成为一个问题。座位的实际分配（将涉及到查找合适相邻座位的算法）是一个业务逻辑问题，所以我们将需要在Java代码中处理它。这意味着我们将需要查询一场演出和座位类型的AVAILABLE_SEATS视图。然后，Java代码（将已高速缓存并分析了相应座位安排计划参考数据）将检查空座位id，并选择若干要预订的座位。然后，它将调用reserve_seats存储过程来预订具有相应id的座位。

所有这一切都将发生在同一个事务中。事务将由J2EE服务器来管理，不是由数据库来管理。保守式加锁将意味着迫使所有试图预订相同演出和座位类型的用户等待到该事务结束。保守式加锁可以通过给上述AVAILABLE_SEATS视图中的SELECT添加FOR UPDATE来轻松地进行实施。然后，下一个排队等候的用户将被授权并加锁，直到其事务完成了仍空着的那些座位id。

开放式加锁能够通过消除堵塞来改进性能，但可能会引起多个用户试图预订相同座位的危险。在这种情况下，我们必须确认那些选定的座位id所关联的SEAT_STATUS行还没有被一个并发事务更改，如果被更改，则需要使该预订失败（但试图做该预订的Java构件不用向该用户报告此开放式加锁故障就能重试该预订请求）。因此，使用开放式加锁可能会改进性能，但会使应用代码变复杂。使用保守式加锁将把这项工作传递给数据库，并保证数据完整性。

如果在数据库中进行座位分配，就不会遇到相同的加锁问题。在Oracle中，我们甚至可以在Java存储过程中做这件事情。不过，这将会降低可维护性，并使实现一个真正的OO解决方案变得十分困难。根据把业务逻辑保持在运行于J2EE服务器内的Java代码中并保证设计仍可移植这一目标，我们应该避免这种方法，除非这种方法证明它是确保满意性能的唯一方法。

加锁策略将被隐藏在一个DAO接口中，所以必要时我们不用修改业务对象即可更改加锁策略。保守式加锁在Oracle中工作得非常好，因为不带FOR UPDATE子句的查询在锁定的数据上将从不发生堵塞。这意味着使用保守式加锁对查询计算空座位数量将不会有影响（需要再现Display Performance屏幕）。在其他数据库中，这样的查询困难会发生锁定——充分说明了同一个数据库存取代码在不同数据库中将表现不同的危险。

因此，我们将决定尽可能使用保守式加锁。不过，由于存在不用破坏应用设计即可更改加锁策略的选择余地，所以若性能测试预示了一个支持并发使用的问题，或者需要使用另外一个RDBMS，我们可以实现开放式加锁。

最后是在哪里执行数据存储的问题。在本章中，我们决定只使用EJB来处理事务性订票过程。这意味着订票过程的数据存取将在EJB层中执行；其他（非事务性）数据存取将在运行于Web容器内的业务对象中执行。

第8章 使用实体组件进行数据存取

实体组件是EJB规范中描述的数据存取构件。虽然它们在实践中有一个令人失望的历史记录（导致了实体组件在EJB 2.0规范中的一次彻底修改），但它们在J2EE核心中的特殊地位意味着我们必须了解它们，即便我们决定不使用它们。

本章将讨论如下内容：

- 实体组件的目标是实现什么，以及在实践中使用它们的经验。
- 实体组件模型的正反两方面，尤其是在实体组件和关系数据库一起使用的时候。
- 决定何时使用实体组件，以及怎样有效地使用它们。
- 如何在使用容器管理式持久性（CMP）与组件管理式持久性（BMP）的实体之间进行抉择。
- EJB 2.0实体组件模型中的重要增强，以及它们对使用实体组件的影响。
- 主流应用服务器产品中的实体组件加锁和高速缓存支持。
- 实体组件的性能。

坦率地说，笔者不太喜欢实体组件。笔者认为它们不应该被看做J2EE应用中数据存取的默认选择。

如果读者决定使用实体组件，希望本章将会帮助读者避免许多常见的陷阱。但是，笔者将会推荐大多数应用中数据存取的替代方法。在下一章中，我们将了解实际的替代方案，并看一看如何实现DAO模式。这一模式在分离业务逻辑与数据存取实现方面通常比实体组件更有效。

实体组件概念

实体组件的用处是让会话组件免于处理持久性数据的低级任务，进而规范化好的设计习惯。它们在版本1.1中成为了EJB规范的一个核心部分；版本2.0引进了重要的实体组件增强。EJB 2.1提供了更一步的增强，笔者将在它们可能会影响未来策略时再讨论这些增强，尽管它们在J2EE 1.3开发中是不可利用的。

实体组件提供一个非常有吸引力的程序设计模型，进而让使用对象概念来存取关系数据库成为了可能。虽然实体组件被设计用来处理任何数据存储，但到目前为止，这是现实生活中最常见的情况，而且本章将集中讨论这种情况。实体组件获得成功的希望是数据存取的具体细节还由容器来透明地处理，进而让应用开发人员把精力都放在实现业务逻辑上。依照这一见解，预计容器提供者会提供高度有效的数据存取实现。

令人遗憾的是，现实情况却有点不同。实体组件是重型对象，并常常表现得不够。O/R映射是一个复杂问题，而且实体组件（即便在EJB 2.0中也）未能解决它的许多问题。愉快地使用对象概念（比如与实体组件关联的遍历）可能会产生灾难性的性能。实体组件没

有消除数据存取的复杂性；它们只是降低这种复杂性，但把这种复杂性大量地转移到了另外一层中。实体组件部署描述符（标准J2EE和容器特有的）是十分复杂的，而且我们简直无法忽视底层数据存储的许多问题。

实体组件的整个概念都存在严重的问题，而且到目前为止还没有通过试验得到可靠解决。最重要的问题包括：

- 当EJB的一个主要目标是把业务逻辑收集到会话组件中时，实体组件为什么需要远程接口？虽然EJB 2.0允许本地访问实体组件，但实体组件模型和相当麻烦的获得实体组件参考的方法反映了实体组件作为远程对象的遗留性。
- 如果按引用访问实体组件，为什么需要使用JNDI来查找它们？
- 实体组件为什么需要基础结构来处理事务界定和安全？这些不是能够由会话组件完全解决的业务逻辑问题吗？
- 实体组件允许我们自然而有效地处理关系数据库吗？实体组件模型往往实施对RDBMS表的两级（而不是面向集合的）访问。这不是关系数据库被设计用来做的事情，而且可能会是低效率的。
- 由于EJB的开销很大，所以它们最好是用做构件（component），而不要用做细粒度的对象。这使得它们极不适合建立细粒度数据对象的模型，而且可以证明的是，这不是使用实体组件的惟一高性价比方法（我们不久将会详细讨论实体组件粒度）。
- 当数据库以不同的方式工作时，实体组件是否获得或值得期望的吗？认为实体组件允许我们忘了加锁之类的基本持久性问题是十分危险的。

像JDO之类的替代方案避免了实体组件所引进的许多问题和大部分复杂性。

需要重点注意的是，实体组件仅仅是J2EE应用中数据存取的选择之一，应用设计不应该依赖于实体组件的使用。

实体组件只是EJB层中的实现选择之一。实体组件不应该被暴露给客户软件。Web层和其他EJB客户软件绝不应该直接访问实体组件。它们只应该与由实现应用业务逻辑的会话组件所构成的一个会话组件层打交道。这不仅保持了应用设计与实现中的灵活性，而且常常还改进了性能。

这一重要见解（加强了Session Facade设计模式）得到了普通赞同：笔者已记不得上一次看到有人主张使用实体组件的远程访问是什么时候的事情。不过，笔者觉得添加一个抽象层对分离会话组件与实体组件来说是一件值得做的事情。这是因为实体组件欠灵活；它们从持久性存储器中提供一个抽象，但使利用了该持久性存储器的代码依赖于这个有点可怕的抽象。

会话组件最好是只通过普通Java数据存取接口的一个持久性门面来访问实体组件。虽然实体组件实行了一种特殊的数据处理方法，但标准Java接口却没有。这种方法不仅保持了灵活性，而且还预见性地检验了一个应用。笔者对实体组件的未来产生了极大的怀疑，因为凡是在实体组件适用的任何地方，JDO都能提供一个更简单、更通用、性能更高的解决方案。通过使用DAO，我们仍有换用JDO或其他任一持久性策略的能力，即便在一开始就使用实体组件实现了一个应用之后。

在下一章中，我们将看一看这种方法的几个例子。

由于EJB 2.0中引进的实体组件有了明显的变化，所以正如我们将要看到的，关于EJB 1.1时期使用实体组件的大部分建议已经过时。

定义

实体组件是一个需要小心对待的题目，所以让我们先来看几个定义，以及实践中对实体组件的反映。

EJB 2.0规范将实体组件定义为：“表示实体的一种面向对象观点的构件，其中这些实体是一个诸如数据库之类的持久性存储器中所存储的或由一个现有企业应用所实现的实体”。该定义将实体组件的目标表达为“具体化”持久性数据。但是，它没有解释这为什么必须由EJB来实现，而不是由普通Java对象来实现。

“Core J2EE Patterns”一书将实体组件描述为：“一个分布式的、共享的、事务性的、持久性的对象”。这个定义解释了一个实体组件为什么需要是一个EJB，但EJB 2.0规范对本地接口的强调移动了门槛，并致使“分布式”特征过时。

所有定义都同意实体组件是数据存取构件，并且与业务逻辑根本无关。

实体组件的另一种重要目标是独立于持久性存储器。实体组件抽象可以处理任何持久性对象或服务，比如一个RDBMS、一个ODBMS或一个遗留系统。

笔者觉得这种持久性存储器独立性在实践中被估计过高：

- 首先，该抽象可能会证明它是高代价的。就现在的情况来说，实体组件抽象是相当不灵活的，并且规定我们应该怎样执行数据存取，所以实体组件与任一持久性存储器打交道的效率最终也可能会很低。
- 其次，笔者不敢保证为不同持久性存储器使用同等重量级的抽象会增加实际的业务价值。
- 第三，大多数企业使用关系数据库，并且这些状况很快发生改变是不可能的（事实上，至今仍没有任何明显的迹象表明这种状况应该改变）。

在实践中，实体组件通常相当于O/R映射的一种基本形式（在使用对象数据库时，几乎不存在对实体组件所提供的这种基本O/R映射的需求）。在现实生活中，实体组件的实现往往把一个关系数据库表的一行建模为一个视图。

实体组件通常是一个薄层，用于具体化一个不是基于对象的数据存储器。如果使用一个像ODBMS之类的面向对象数据存储器，这个薄层不是必需的，因为可以使用助手类从会话组件中访问这种数据存储器。

EJB规范描述了两种类型的实体组件：使用Container Managed Persistence (CMP，容器管理式持久性) 的实体组件和使用Bean Managed Persistence (BMP，组件管理式持久性) 的实体组件。EJB容器处理使用CMP的实体的持久性，因而只要求开发人员实现任何逻辑并且定义要被保留的Bean属性。就使用BMP的实体而言，开发人员负责处理持久性：通过实现由容器调用的回调方法。

应该怎样使用实体组件

令人惊讶的是，如果实体组件是EJB规范的一个关键部分，怎么会有关于怎样使用实体组件以及它们应该建模什么的那么大争论。这种情况同样出现在了EJB规范的第三版中，进而暗示了实体组件的使用几乎没有解决那些基本问题。没有任何一种使用实体组件的方法在实际的应用系统中卓越超群。

争论的焦点主要有两个：实体组件的粒度，以及实体组件是否应该实现业务逻辑。

粒度的争论

实体组件应该建模的对象粒度主要有两种选择：精细（fine-grained）和粗糙（coarse-grained）实体组件。如果我们正在使用一个RDBMS，一个精细实体可能会映射到单个表中的一个数据行。一个粗糙实体可能会建模一条逻辑记录，而这条记录可能散布于多个表中，比如一个User和关联的若干Invoice项。

EJB 2.0 CMP通过增加对容器管理式关系的支持和引进实体主方法，使处理精细实体变得更容易；而实体主方法使多个精细实体上的操作变得更方便。本地接口的引进也降低了精细实体的系统开销。这些优化中，没有一种优化可在EJB 1.1中得到，因而意味着粗糙实体通常是提供充足性能的惟一选择。“EJB Design Patterns”一书的作者Floyd Marinescu认为，EJB 2.0规范证明了反对粗糙实体方法是正确的。

粗糙的复合实体（Composite Entity）指的是这样的实体组件：它们提供一个统一的入口点来进入一个由相关的从属对象（dependent object）所构成的一个网络。从属对象也是持久性对象，但不能离开复合实体而单独存在，并且复合实体控制着从属对象的生存周期。在上述例子中，一个User可能会被建模为一个复合实体，其中Invoice和Address被建模为从属对象。该User复合实体将按照需要创建Invoice和Address对象，并使用它所管理的数据装入操作的结果填充它们。与精细实体模型相反，从属对象不是EJB，而是普通Java对象。

可以证明的是，粗糙实体比精细实体更面向对象。它们不必奴隶般地紧跟RDBMS模式，因而意味着它们不强迫代码使用它们来处理RDBMS概念，而是处理对象。它们降低了使用实体组件的开销，因为不是所有持久性对象都被建模为EJB。

Composite Entity设计模式的主要动机是消除远程地访问精细实体的系统开销。这个问题的消除主要依靠了引进本地接口。除了该远程访问理由（已不再充分）之外，赞同Composite Entity设计模式的关键理由包括：

- 更大的可管理性。使用精细实体组件可以产生大量可以与一个应用的用例没有太大关系的类和接口。我们将为每个表最少拥有3个类（本地或远程接口、主接口和bean类），也可能将有4到5个类（增加一个业务方法接口和一个主键类）。部署描述符的复杂性也将有显著的增加。
- 避免数据模式依赖性。精细实体组件具有代码耦合的危险：它们的使用与底层数据库太密切。

这两条理由也是反对使用精细实体的强有力理由，即便对EJB 2.0来说也是如此。

有几个资源详细讨论了复合实体（例如，“Core J2EE Patterns”一书中对Composite

Entity设计模式的讨论）。可是，Craig Larman提供了笔者已经见过的关于怎样建模粗糙实体（他称做Aggregate Entity（集合实体））的最中肯的讨论。请参见<http://www.craiglarman.com/articles/Aggregate%20Entity%20Bean%20Pattern.htm>。Larman推荐了区分实体组件与从属对象的下列标准：

- 多个客户软件将直接引用该对象。
- 该对象有一个不受另一个对象管理的独立生存周期。
- 该对象需要一个唯一性身份。

这些标准的前两条对性能有重要影响。从属对象与其他实体毫无关系是最基本的。否则，EJB容器的加锁策略可能会伤害到并发存取。除非第三条标准得到满足，否则使用一个无状态会话组件比使用一个实体更可取；无状态会话组件能够提供数据存取方面的更大灵活性。

使用Composite Entity模式的致命缺陷是，实现粗糙实体通常要求使用BMP。这不仅意味着开发人员要做更多的工作，而且BMP实体组件约定也有严重的问题（下文将讨论它们）。我们不是在谈论简单的BMP代码——我们还必须面对一些棘手的问题：

- 每当一个粗糙实体被访问时物化它里面的所有数据是非常高代价的。这意味着我们必须实现一个惰性装入（lazy loading）策略，即数据只在被需要时才被检索。如果我们正在使用BMP，最终将会编写许多代码。
- ejbStore()方法的实现需要非常巧妙，才能避开发布使该对象的完整状态持久留存所需要的所有更新，除非所有持久性对象中的数据都已发生了变化。

“Core J2EE Patterns”一书深入而详细地讨论了解决这些问题的“Lazy Loading Strategy”、“Store Optimization (Dirty Maker) Strategy”和“Composite Value Object Strategy”，然后举例说明了Composite Entity模式所产生的实现复杂性。首先涉及的复杂性是着手为每个Composite Entity编写一个O/R映射框架。

Composite Entity模式体现了可靠的设计原理，但实体组件BMP的限制使得它不能有效地工作。实质上，Composite Entity模式使用一个粗糙实体作为一个持久性门面来处理持久性逻辑，而会话组件处理业务逻辑。如果该持久性门面是一个实现了普通接口的助手类，而不是一个实体组件，结果会更好。

在2000年中后期发布的早期草案中，EJB 2.0规范似乎正朝着粗糙实体的方向发展，因而使“从属对象”的使用正规化。但是，从属对象在规范委员会上依然是有争议的，而后来引入本地接口则说明方向完全发生了变化。这看起来已解决了实体粒度争论。

不要使用Composite Entity模式。在EJB 2.0中，最好是通过使用CMP把实体组件用于相当精细的对象。

Composite Entity模式只能通过使用BMP或者通过给CMP Bean添加明显的硬编码持久性逻辑来实现。这两种方法都会降低可维护性。如果预期的Composite Entity根本没有自然主键，用一个助手类从一个会话组件中处理持久性要好于通过建模一个实体来处理持久性。

业务逻辑的争论

还有一个与实体组件是否应该含有业务逻辑有关的争论。这是致使许多EJB 1.1建议已经变得过时的另一个方面，甚至更有害，导致了EJB 2.0中的实体组件的彻底修改。

得到普遍认可的一点是：实体组件的用途之一是分离业务逻辑与对持久性存储器的访问。但是在EJB 1.1中，远程调用的系统开销就意味着从会话组件中访问实体组件的性能是极差的。以前，避免这一开销的一种方法是把业务逻辑放到实体组件中。现在，这不再是必需的。

把以下两种类型的行为放到实体组件中有一些争论：

- 输入数据的验证；
- 保证数据完整性的处理。

就个人而言，笔者觉得验证代码不应该进入实体组件。我们将在第12章中较详细地验证。验证常常需要业务逻辑，而且（在分布式应用中）可能需要运行在客户端上来降低往返于客户软件与EJB容器之间的网络通信量。

保证数据完整性是一个棘手问题，而且有时需要在实体组件中做一些工作。类型转换就是一个常见需求。例如，一个实体组件可以通过把数据库中的一个字符列暴露为来自某一常数集的一个值来增加价值。虽然用户的注册状态可以在数据库中被表示为字符值I、A或P之一，但一个实体组件可以保证客户软件看到这个数据，并把它设置为常数Status.INACTIVE、Status.ACTIVE或Status.PENDING之一。但是，如果其他进程将更新这个数据，这种低级数据完整性检查还必须在数据库中进行。

一般说来，如果我们区分开业务逻辑与持久性逻辑，确定具体行为是否应该被放到实体组件中就会容易得多。实体组件只是实现持久性逻辑的方法之一，而且不应该有实现业务逻辑的任何特殊权力。

在实体组件中只实现持久性逻辑，不要实现业务逻辑。

作为调停者的会话实体

EJB层的客户不应该直接与实体组件打交道，但应该专门与一个会话组件层打交道，对于这一点几乎不存在争论。与其说这是一个实体组件设计问题，倒不如说是一个体系结构问题，所以我们在下一章中分析这个问题。

使用会话组件来调停对实体组件的访问有许多理由，其中之一是允许会话组件处理事务管理，与其说这是一个业务逻辑问题，倒不如说是一个持久性问题。即便对本地调用来说，如果每个实体组件获得器和设置器方法运行它自己的事务中，数据完整性就可能会遭到损害，性能将会被降低（由于建立和完成一个事务的系统开销）。

需要注意的是，实体组件必须使用CMT。要想保证容器之间的可移植性，使用EJB 2.0 CMP的实体组件应该只使用Required、RequiresNew或Mandatory事务属性。

在ejb-jar.xml部署描述符中将实体组件业务方法上的事务属性设置成Mandatory是一个好习惯。这帮助保证实体组件得到正确使用：通过在一个事务上下文没有因为一个javax.transaction.TransactionRequiredException而失败的情况下引发调用。事务上下文应该由会话组件提供。

CMP与BMP的比较

EJB容器处理CMP实体的持久性，进而只要求开发人员实现任何逻辑，并且定义要被保留的bean属性。在EJB 2.0中，EJB容器还能管理关系和发现者（用部署描述符中所使用的一种特殊查询语言——EJB QL来指定）。开发人员只需要编写定义持久性属性和关系的抽象方法，并在部署描述符中提供允许容器生成实现代码所必需的信息即可。

开发人员不必使用JDBC之类的API来编写与数据存储器相关的任何代码。从消极的方面看，开发人员通常无法控制持久性代码的生成。在效率方面，容器所生成的SQL查询可能不如开发人员所编写的SQL查询（尽管有些容器允许生成的SQL查询被调整）。

下列讨论将使用一个关系数据库作为例子。但是，关于必须怎样装入数据的那些要点适用于所有类型的持久性存储器。

就BMP实体而言，开发人员完全负责处理持久性，通常是通过实现ejbLoad()和ejbStore()回调方法来装入状态和写状态到持久性存储器上。除了ejbCreate()和ejbRemove()方法之外，开发人员还必须实现所有发现者方法来返回匹配实体的一个主键对象集（Collection）。这是一项工作量很大的任务，但使开发人员拥有了对持久性被怎样管理的更大控制权。由于没有容器能够提供全部可接受数据源的CMP实现，所以当有异乎寻常的持久性需求时，BMP可能就是实体主键的惟一选择。

CMP与BMP的比较问题是J2EE界中争论较激烈的另一个焦点。许多开发人员认为BMP将会比CMP提供更好的性能，因为它允诺了更大的控制权。但是在实践中，结果通常正好相反。

BMP实体组件生存周期（其间数据要么必须用ejbLoad()方法装入并用ejbStore()方法更新，要么用个别属性获得器装入并用个别属性设置器更新）让生成有效地满足应用的数据使用模式的SQL语句变得十分困难。例如，如果我们需要实现惰性装入，或者需要作为一个组来检索并更新实体组件的一个持久性字段子集以反映使用模式，那么我们将需要付出大量的努力。另一方面，一个EJB容器的CMP实现可以轻松地生成支持这种优化所必需的代码（例如，WebLogic同时支持这两者）。与实现BMP实体组件时相比，实现由会话组件或普通Java对象所使用的一个DAO时编写有效的SQL要容易得多。

BMP所允诺的“控制权”在一个关键方面完全是虚假的。开发人员可以选择怎样在持久性存储器中提取和写入数据，但不能选择何时做这些操作。结果是一个非常严重的性能问题：**n+1**查询发现者问题。这个问题的出现归因于BMP实体的约定要求开发人员实现发现者来返回实体组件主键，而不是返回实体。

请看一看下列例子，本例以取自一个主流的英国Web站点的真实情景为基础。一个User实体运行在如下所示的一个表（含有300万个用户）上：

USERS		
PK	NAME	MORE COLUMNS...
1	Road	...
2	Gary	...
3	Portia	...

这个实体既在用户访问他们的账户时（即在每次有一个实体被装入时）被使用，又被该站点帮助台上的工作人员使用。帮助台用户经常需要访问多个用户账户（例如，在查找忘了的密码时）。有时，他们需要执行返回大型结果集的查询。例如，查询具有某些邮政编码（比如North London's N1）的所有用户将会返回数千个实体，因而导致发现者方法超时。

下面让我们来看一看为什么会出现这种情况。由User实体的开发人员所实现的发现者方法将从下列合情合理的SQL查询中返回5000个主键：

```
SELECT PK FROM USERS WHERE POSTCODE LIKE 'N1%'
```

即使POSTCODE列上没有索引（由于此类搜索还没有频繁到足以证明建立索引是正当的地步），但该查询在Oracle数据库中也不会花费太长的运行时间。问题在于接下来发生什么。EJB容器创建或重用5000个User实体，然后根据每个主键来自5000个独立查询的数据填充这些User实体：

```
SELECT PK, NAME, <other required columns> FROM USERS WHERE PK = <first match>
...
SELECT PK, NAME, <other required columns> FROM USERS WHERE PK = <5000th match>
```

这就意味着总共有n+1条SELECT语句，其中n是一个发现者所返回的实体数量。在本例中，n是5000（一个巨大的数量）。在该站点的这一部分投入使用之前，开发团队就已意识到BMP实体组件将解决不了这个问题。

很显然，这是效率低得可怕的SQL。实际上，我们也是迫不得已才使用它来演示BMP实际给了我们的控制权的局限性。另一方面，任何正当的CMP实现都会通过使用如下所示的一个有效查询来提供预装入那些行的选项：

```
SELECT PK, NAME, <other required columns> FROM USERS WHERE POSTCODE LIKE 'N1%'
```

如果我们只需要前几行，这仍是非常过分的，但运行速度比上述BMP例子要快得多。例如，在WebLogic的BMP实现中，预装入操作默认地发生，并且发现者将在一段不太长的时间内执行完毕。

尽管CMP性能对大型结果集将会好得多，但实体组件在这样的情景中是一个很差的选择，因为创建并填充这么多的实体组件将会有很大的系统开销。

BMP实体中的n+1问题没有令人满意的解决方法。使用粗糙实体也避不开这个问题，因为一个粗糙实体的实例个数未必就比一个精细实体的少。粗糙实体只用做一个人口，用于进入将以其他方式被建模为实体的那些相关对象。上述应用使用了与User实体相关联的精细实体，比如Address和SavedSearch，但在这种情况下，使User实体变成精细实体将不产生任何性能收益。

人们已经建议使用所谓的“Fat Key”模式来避开这个问题，这种方法的原理是将实体组件的数据保存在主键对象中。这使得发现者能够执行一个正常的SELECT（用所有实体数据填充那些“肥”对象），而实体组件实现的ejbLoad()方法只需从“肥”键中获得数据即可。这种策略确实管用，并且不违背实体组件约定，但本质上是一个窃用。任何一项需要这样—

种迂回方法来提供足够性能的技术都有问题。关于“Fat Key”模式的讨论，请参见http://www.theserverside.com/patterns/thread.jsp?thread_id=4540。

当返回主键而非实体导致这个问题时，BMP约定为什么强迫发现者返回主键而不返回实体呢？该规范要求这个约定允许容器来实现实体组件高速缓存器。在从持久性存储器中装入所有数据之前，容器可以选择查找它的高速缓存器来检查它是否已拥有具有给定主键的这个实体组件的一个最新实例。稍后，我们将讨论高速缓存。不过，允许容器执行高速缓存在我们刚描述过的那种大型结果集情景中绝没有什么安慰作用。对于一个人口稠密的London邮政编码来说，在这样一个搜索之后缓存代表所有用户的实体简直就是浪费服务器资源，因为这些实体中的任何一个在从高速缓存器中被逐出之前绝不会被访问。

赞同使用BMP的少数几个站得住脚的理由之一是：BMP实体比CMP实体是更可移植的；对容器有较小的依赖性，因此预计在不同的应用服务器之间行为和性能会十分类似。只有在需要运行在多个服务器上的极少数应用中，这才是一个需要考虑的因素。

BMP实体通常不如CMP实体那么容易维护。虽然在由会话组件所使用的一个助手类中使用JDBC编写有效而又可维护的数据存取代码是可能的，但BMP约定的刻板可能会使数据存取代码的可维护性变得较差。

和关系数据库一起使用BMP有几个站得住脚的理由。如果说BMP实体组件有任何合法用处的话，这就是处理遗留数据存储器。相对一个关系数据库来使用BMP使批处理功能度的使用变得不可能，而这一功能度恰好是人们设计关系数据库的目的。

不要在实体组件中使用BMP，而要从无状态会话组件中使用持久性。这方面的内容将在下一章中讨论。和从一个DAO层中执行数据存取相比，使用BMP实体组件不会增加什么价值，只会增加复杂性。

EJB 2.0中的实体组件

EJB 2.0规范（于2001年9月发布）引进了与实体组件，尤其是与那些使用CMP的实体组件有关的重要增强。由于这些增强迫使我们重新评估以前为EJB 1.1实体组件所确定的策略，所以仔细研究这些增强是十分重要的。

本地接口

为EJB引进本地接口（参见第6章）极大地降低了从同一个JVM内的会话组件或其他对象中使用实体组件的系统开销（但是，实体组件将始终比普通Java对象具有更大的系统开销，因为EJB容器在所有涉及到EJB的调用上都执行方法拦截）。

本地接口的引入使实体组件变得更管用，但却抛出了一个与实体组件有关的基本假设（它们应该是远程对象），并致使关于使用实体组件的大多数建议变得过时。值得商榷的是，EJB 2.0实体不再拥有一个哲学上的基础，或者说，成为EJB规范的一部分缺少正当的理由。如果一个对象只被给予了一个本地接口，使它成为一个EJB的根据就被极大地削弱，因而舍

弃了作为将对象建模为实体组件的惟一理由：实体组件所提供的各种数据存取能力，比如CMP；而CMP能力却可以与JDO之类的替代品相媲美。

在EJB 2.0应用中，绝不要让实体组件有远程接口。这样可以保证远程客户软件通过一个由实现该应用的用例的会话组件所构成的层来访问实体，最大限度地减少实体组件的性能开销，而且意味着我们不必使用一个值对象来获得和设置实体上的属性。

本地接口业务方法

另一个重要的EJB 2.0增强是添加了实体组件主接口上的业务方法：其工作不与某一个单一实体实例特别相关的方法。和本地接口的引进一样，主方法的引进使CMP和BMP实体组件都受益。

主接口业务方法是除了一个实体的本地或远程主接口上所定义的发现者、创建或删除方法之外的其他方法。主业务方法在容器所选择的任一实体实例上执行，无需访问一个主键，因为一个主方法的工作不局限于任何一个实体。主方法实现具有和发现者一样的运行时上下文。一个主接口的实现可以执行JNDI访问，找出该调用者的角色，访问资源管理器和其他实体组件，或者标记当前事务为回退。

主接口方法签名上的唯一限制是：为了避免混淆，方法名称不得以create、find或remove开头。例如，本地接口上的一个EJB主接口方法看起来可能会像下面这样：

```
int getNumberOfAccountsWithBalanceOver(double balance);
```

和创建方法必须具有以ejbCreate()开头的名称一样，bean实现类上的对应方法必须具有一个以ejbHome开头的名称，例如：

```
public int ejbHomeGetNumberOfAccountsWithBalanceOver(double balance);
```

主接口方法只不过做了比实体组件的历史中多一点的事情来允许对关系数据库的有效访问。它们提供了一种方法来摆脱精细实体所强行实施的面向表行的建模方式，进而允许使用RDBMS集合操作对多个实体进行有效的操作。

就CMP实体而言，主接口常常由一个bean实现类中所定义的另一种新方法来支持：一个ejbSelect()方法。一个ejbSelect()方法是一个查询方法。但是，它和一个发现者不同，因为它不通过该实体组件的主或构件接口被暴露给客户软件。和EJB 2.0 CMP中的发现者相同的是，ejbSelect()方法返回ejb-jar.xml部署描述符中所定义的EJB QL查询的结果。一个ejbSelect()方法必须是抽象的。在一个实体组件实现类中实现一个ejbSelect()方法并避免使用一个EJB QL查询是不可能的。和发现者不同的是，ejbSelect()方法不必返回实体组件，它们可能返回具有容器管理式持久性的实体组件或字段。和发现者不同的是，ejbSelect()方法能够在—个处于池状态的实体（没有身份标识）或一个处于准备状态的实体（有身份标识）上被调用。

主业务方法可以调用ejbSelect()方法来返回与多个实体相关的数据。如果单个实体上的业务方法需要获得或操作多个实体，它们也可以调用ejbSelect()方法。

在许多情况下，主接口方法的添加允许了实体组件的有效使用，而这在EJB 1.1约定下已经证明是不可能的。问题是EJB QL——可移植EJB查询语言（我们将在下文中讨论）还没有成熟得足以提供许多实体主接口方法所需要的威力。要想使用有效的RDBMS操作，我们必须使用JDBC或其他低级API来编写自定义的持久性代码。如果必要，主接口方法甚至可以用来调用存储过程。

请注意，与持久性逻辑相对的业务逻辑放在会话组件仍比放在主接口方法中更可取。

EJB 2.0 CMP

在EJB 2.0的实体组件增强中，人们谈论最多的是对实体组件之间的容器管理式关系的支持的增加，而该增加以本地接口的引进为基础。

EJB 1.1中对CMP实体的支持是初步的，并且只能满足简单需求。虽然EJB 2.0规范要求容器尊重CMP组件的EJB 1.1约定，但EJB 2.0规范为CMP实体引进了一个差别很大的新约定。

基本概念

在实践中，EJB 1.1 CMP只限于用做这样一种手段：映射一个Java对象的实例变量到一个单独数据库表中的列。它只支持基本类型和具有一种对应SQL类型的简单对象（比如日期）。该约定是粗糙的；具有容器管理式持久性的实体组件字段需要是公用的。一个实体组件是一个具体类，而且包含像下面这样的字段（将按照容器被映射到数据库）：

```
public String firstName;
public String lastName;
```

由于EJB 1.1 CMP处于严格的规定之下，所以使用了它的应用变得严重依赖于目标服务器的CMP实现，因而严重地损害了实体组件应提供的可移植性。例如，由于CMP发现者方法不是由开发人员编写的，而是由容器生成的，所以每个容器在部署描述符中都使用各自的自定义查询语言。

EJB 2.0是一大进步，尽管它实质上仍以映射对象字段到单个数据库表中的列为基础。CMP的EJB 2.0约定基于抽象方法，而不是基于公用实例变量。CMP实体是抽象类，而容器负责实现持久性属性的设置和检索。简单的持久性属性叫做CMP字段（CMP field）。定义firstName和lastName CMP字段的EJB 2.0方式如下所示：

```
public abstract String getFirstName();
public abstract void setFirstName(String fname);
public abstract String getLastname();
public abstract void setLastname(String lname);
```

和EJB 1.1 CMP中一样，映射在部署描述符内的Java代码外部被定义。EJB 2.0 CMP引进了许多另外的元素来处理它的更复杂能力。ejb-jar.xml描述了那些持久性属性和CMP实体之间的关系。另外的专有部署描述符，比如WebLogic的weblogic-cmp-rdbms-jar.xml定义了到一个实际数据源的映射。

抽象方法的使用是一种比公用实例变量的使用更优秀的方法（例如，它允许容器辨别字段何时已被修改过，进而使优化变得更容易）。惟一的缺点是：由于具体实体类由容器生成，所以一个不完整（抽象）的CMP实体类将成功地编译，但部署会失败。

容器管理式关系 (CMR)

EJB 2.0 CMP提供了属性的更多持久性。它引进了Container-Managed Relationship (CMR, 容器管理式关系) (即运行于同一个EJB容器内的实体组件之间的关系) 的概念。这使得精细实体能够用来建模一个RDBMS中的个别表。

关系涉及到本地或远程接口。使用了一个远程接口的一个实体组件可能具有关系，但这些关系不能通过它的远程接口被暴露出来。EJB 2.0支持一对一、一对多和多对多关系（多对多关系将需要由RDBMS中的一个连接表（join table）来支持。这个关系对实体组件的用户来说是隐藏的）。CMR可以是单向性的（只能沿着单个方向导航的），也可以是双向性的（能沿着两个方向导航的）。

和CMP字段相同的是，CMR由抽象方法表达在实体组件的本地接口中。在一一对关系中，CMR将被表达为一个属性，并且这个属性的一个值就是该相关实体的本地接口：

```
AddressLocal getAddress();
void setAddress(AddressLocal p);
```

至于一对多或多对多关系，CMR将被表达为一个集合（Collection）：

```
Collection getInvoices();
void setInvoices(Collection c);
```

实体组件的本地接口的用户可能会操纵被暴露的Collections，可常常遭受到某些限制（例如，一个Collection绝不能被设置成null：这个空的Collection必须用来指出没有任何对象正处于指定角色中）。EJB 2.0规范要求容器保护引用完整性——例如通过支持级联删除。

虽然本地接口中的抽象方法决定调用者如何使用CMR关系，但部署描述符用来告诉EJB容器如何映射关系。标准ejb-jar.xml文件含有描述关系和可导航性的元素。映射到一个数据库的映射细节（比如连接表的使用）将是容器特有的。例如，WebLogic在weblogic-cmp-rdbms-jar.xml文件中定义了几个元素来配置关系。在JBoss 3.0中，jbosscmp-jdbc.xml文件扮演了同样的角色。

不要依靠EJB 2.0 CMP来保证数据的引用完整性，除非读者可以肯定绝对没有其他进程将访问该数据库。应该使用数据库约束。

使用EJB 2.0中“从属对象”的粗糙实体概念是可能的。EJB 2.0规范（§ 10.3.3）把它们称做从属值类（dependent value class）。从属对象只不过是通过抽象的获得器和设置器方法所定义的CMP字段，只是这些字段具有无对应SQL类型的Java对象类型。从属对象必须是可串行化的具体类，而且通常被作为一个二进制对象保存到底层数据存储器中。

使用从属值对象通常不是一个好主意。问题是它把底层数据源看做一个转储存储装置。或许，数据库将理解不了已串行化的Java对象。因此，该数据将只对创建了它的J2EE应用有用。例如，在该数据上运行报表将是不可能的。集合操作将无法使用该数据，如果数据存储

器是一个RDBMS。从属对象串行化和取消串行化将是高代价的。根据笔者的经验，已串行化对象的长久持久性对版本化问题是一个缺点，如果这个已串行化对象发生变化。EJB规范建议只把从属对象用于保留遗留数据。

EJB QL

EJB 2.0规范引进了一种新的可移植查询语言，供CMP实体使用。这是实体组件的可移植性允诺的一个关键元素，目的是为了让开发人员免于需要使用数据库特有的查询语言，比如SQL或EJB 1.1 CMP中所使用的专有查询语言。

笔者对EJB QL存有极大的疑虑。笔者不认为它设法实现的结果——CMP实体组件的总代码可移植性——会证明发明（和学习）一种新查询语言是正确的。重新发明已有的东西同样不是一个好的主意，无论是由规范委员会，应用服务器供应商，还是应用开发人员来重新发明。

笔者注意到关于EJB QL的下列概念性问题（我们不久将讨论一些现实问题）：

- 它引进了在绝大多数情况中绝非必要，并且使有效地完成某些任务变得困难的一种相当低级的抽象，
- 它使用起来不是特别容易。相反，SQL得到了广泛的理解。EJB QL将需要变得更复杂才能满足现实需求。
- 它纯粹是一种查询语言。使用它执行更新是不可能的。惟一选择是获得由一个ejbSelect()方法所产生的多个实体并个别地修改它们。这浪费了J2EE服务器与RDBMS之间的带宽，需要遍历一个Collection（以及必要的转换）和发布许多各别的更新。这保持了实体组件背后的基于对象的概念，但在许多情况下可能是无效的。它比使用SQL在一个RDBMS中执行同样的更新要复杂和慢得多。
- 它不支持子查询，而子查询在SQL中可以用做构造复杂查询的一种直观手段。
- 它不支持动态查询。查询必须在部署时编写到部署描述符中。
- 它束缚于CMP实体组件。相反，JDO提供了一种能用在任一类型对象中的查询语言。
- EJB QL测试起来很困难。我们只有通过测试运行于一个EJB容器内的实体的行为，才能确定一个EJB QL查询没有像预计的那样工作。我们可能只有通过查看EJB容器正在生成的SQL语句，才能确定一个EJB QL查询没有工作。修改EJB QL并重新测试将涉及到重新部署EJB（这是多么大的工作量，并且在应用服务器之间将会有差别）。相反，通过在SQL*Plus（当使用Oracle时）那样的一个数据库工具中发布SQL命令或运行脚本，没有任何J2EE也能测试SQL。
- EJB QL没有ORDER BY子句，因而意味着排序必须在数据被检索出来之后才能进行。
- EJB QL似乎沿着两个方向发展，在任一方向上它都无法获得成功。坦率地说，如果希望它被翻译成SQL（在实践中似乎更有现实意义），这是多余的，因为SQL已是众所周知的，并且更具有威力。如果它脱离RDBMS概念而单独存在——例如允许遗留主机数据源上的实现，它被认为只提供数据操作的一个最低共同点，并且不足以解决现实问题。

为了解决上述这些问题当中的一部分，像WebLogic那样的EJB容器实现了EJB QL的扩展。可是，如果EJB QL的整个依据就是它的可移植性，那么对专有扩展的需要将严重地降

低它的价值（但SQL方言不同，可在大多数RDBMS之间使用的SQL子集比SQL QL强有力得多）。

通过引进对AVG、MAX和SUM之类的集合函数的支持和引进一条ORDER BY子句，EJB 2.1解决了EJB QL的部分问题。但是，它仍不支持更新，而且永远不可能支持更新。子查询和动态查询之类的其他重要特性仍被推迟到EJB规范的未来发布。

使用EJB 2.0实体进行O/R建模的限制

尽管引进了各种重要增强，但根据指定，CMP实体组件依然是O/R映射的一个基本形式。EJB规范忽略了O/R映射方面最难办的部分问题，并且使利用关系数据库的部分能力变得不可能。例如：

- 不支持开放式加锁。
- 对批量更新的支持极差（EJB 2.0主方法至少使它们有可能，但容器——和EJB QL——在实现它们方面没有提供任何帮助）。
- 从一个对象到单个表的映射概念是有限的，而且EJB 2.0规范没有建议EJB容器应该如何解决这个问题。
- 不支持被映射对象中的继承性。像WebSphere那样的有些EJB容器把这实现为一个专有扩展。请参见http://www.transarc.ibm.com/Library/documentation/websphere/appserv/atswfg/atswfg12.htm#HDREJB_ENTITY_BEANS。

混合了CMP/BMP的自定义实体行为

笔者在前面曾经提过使用自定义代码来实现用CMP、CMR和EJB QL无法实现的持久性操作。

这就产生了CMP/BMP混合体。这些混合体也是实体，只是它们的生存周期由EJB容器的CMP实现来管理，并使用CMP来保留它们的字段和简单关系，但使用数据库特有的BMP代码来处理更复杂的查询和更新。

一般说来，主接口方法是从这类BMP代码中受益的最可能的候选者。当生成的EJB QL速度慢和效率低时（由于容器不允许调整从EJB QL中生成的SQL），主接口方法也可以使用JDBC来实现。

和ejbSelect()方法和CMP实体上的发现者方法不同，实体组件开发人员（不是EJB容器）实现主接口业务方法。如果ejbSelect()方法不能提供必要的持久性操作，开发人员可以自由地获得数据库存取的控制权。一个带有CMP的实体组件不受限于执行资源管理器存取；它只是决定把大多数持久性操作留给容器。为了一个带有CMP的实体组件，ejb-jar.xml部署描述符中将需要把一个数据源变成可供使用的。数据源不自动暴露给带有CMP的实体组件。

编写数据装入和存储的自定义扩展也可能的，因为EJB容器在带有CMP的实体上调用ejbLoad()和ejbStore()方法。EJB 2.0规范的10.3.9节描述了这些方法的约定。

CMP/BMP混合组件是不精致的，但有时它们在EJB QL的当前给定条件下是必不可少的。

CMP/BMP混合组件带有的惟一严重困难是对EJB容器高速缓存实体组件的能力的潜在影响，如果自定义代码更新数据库。EJB容器没有办法知道自定义代码正在对底层数据源做什么，所以它必须把这样的修改当做各别进程所做的修改来对待。这是不是将会影响性能将取决于使用中的加锁策略（请参见后文关于加锁和高速缓存的讨论）。有些容器（比如WebLogic）允许用户清洗掉其底层数据由于集合操作而发生了变化的缓存实体。

当使用实体组件时，如果一个CMP实体组件未能接纳那些必要操作的一个子集，那么给这个CMP实体组件添加自定义数据存取代码通常比转换到BMP好。CMP/BMP混合组件是粗糙的。但是，它们有时是有效地使用实体组件的惟一方法。

当使用CMP/BMP混合组件时，请注意下列事项：

- 更新数据可能会破坏实体组件高速缓存。要保证你了解自己的容器中任何高速缓存如何工作，以及自定义数据存取代码所执行的任何更新的影响。
- 这种组件的可移植性可能会随着EJB规范的成熟而改进——如果BMP代码将查询，而不是更新。例如，由于EJB 2.0没有提供集合函数，所以需要用BMP来实现的EJB主方法或许能用EJB 2.1中的EJB QL来实现。
- 如果可能的话，把数据库特有代码隔离到一个实现数据库无关接口的助手类中。

实体组件的高速缓存

实体组件性能主要取决于EJB容器的实体组件高速缓存策略，而高速缓存又取决于该容器所运用的加锁策略。

依笔者看，实体组件的价值取决于有效的高速缓存。令人遗憾的是，这在应用情景与不同EJB容器之间有着非常大的差别。

如果获得很高的高速缓存器命中率是可能的，那么使用只读实体组件或者由于你的容器有一个高效的高速缓存器，实体组件是一个不错的选择，而且将会工作得很好。

实体组件加锁策略

实体组件的加锁策略主要有两个，而这两个策略在EJB规范（§ 10.5.9和§ 10.5.10）中都得到了预示。用来描述它们的术语在容器之间有所不同，但笔者选择了使用WebLogic术语，因为它易懂而准确。

在使用实体组件开发应用之前，了解你的EJB容器所实现的加锁策略是十分必要的。实体组件不允许我们忽视基本的持久性问题。

独占性加锁

独占性加锁是WebLogic 5.1和WebLogic容器的更早期产品所使用的默认策略。许多其他EJB容器至少最初也使用了这种策略。独占性加锁在EJB规范（§ 10.5.9）中被描述为“Commit Option A”，而JBoss 3.0资料也给独占性加锁使用了这个名称。

如果使用这种加锁策略，容器将只把实体的单个实例保持在使用状态。实体的状态通常将在事务之间被高速缓存起来，因而可以最大限度地减少对底层数据库的调用。问题（以及把这称做“独占性”加锁的原因）是：容器必须串行化对实体的访问，因而锁定等待使用该实体的用户。

独占性加锁具有下列优点：

- 并发访问在不同的底层数据存储器之间将以相同的方式被处理。我们将不依赖于数据存储器的行为。
- 对单个实体的真正串行访问（当连续访问时，或许由同一个用户的动作所引起，不被封锁）将会运转得很好。这种状况在实践中确实会出现。例如，如果实体只涉及个别用户，并且只被这些相关用户访问。
- 如果我们不是正运行在一个聚类中，并且无其他进程正更新数据库，通过保持实体组件在事务间的状态来高速缓存数据是很容易的。容器可以跳过对ejbLoad()方法的调用，如果它知道实体状态是最新的。

独占性加锁具有下列缺点：

- 吞吐量将受到限制，如果多个用户需要使用相同的数据。
- 独占性加锁是多余的，如果多个用户只需要读取相同的数据，而不更新这个数据。

数据库加锁

如果使用数据库加锁策略，解决并发问题的责任就留给了数据库。如果多个客户软件访问同一个逻辑实体，EJB容器只实例化具有同一个主键的多个实体对象。这种加锁策略由数据库决定，而且将由实体组件方法上的事务隔离级别来决定。数据库加锁在EJB规范（§10.5.10）中被描述为“Commit Option B and C”，而JBoss 3.0资料采用了这个术语。

数据库加锁具有下列优点：

- 它能支持更大的并发性，如果多个用户访问同一个实体。并发控制会更智能。数据库或许能够识别出哪些用户正在读取，哪些用户正在更新。
- 加锁基础结构没有重复。大多数供应商已经在他们的加锁策略上花费了十几年时间或做了大量的工作，而且已取得了不错的成果。
- 数据库提供工具来帮助检测死锁的可能性大于EJB容器供应商。
- 数据库能够保护数据完整性，即使除J2EE服务器之外的其他进程正在访问和操纵数据。
- 我们能够有机会在实体组件代码中实现开放式加锁。独占性加锁是由EJB容器实施的保守式加锁。

数据库加锁具有下列缺点：

- 数据库之间的可移植性无法得到保证。并发访问的处理在不同的数据库之间有非常大差别，即便在发布相同SQL的时候。虽然笔者怀疑可移植性在不同数据库之间的可实现性，但这是实体组件的主要承诺之一。能够对不同数据库运行但行为有所不同的代码比需要明确移植的代码更危险，更糟糕。
- ejbLoad()方法必须在一个事务开始时始终被调用。一个实体的状态不能在事务之间被高速缓存。与独占性加锁相比，这会降低性能。

- 我们有两种选择：一个非常聪明的高速缓存器，以及根本没有高速缓存器，无论我们是否正运行在一个聚类中。

WebLogic 6.0和较后版本既支持独占性加锁，又支持数据库加锁，但默认地使用数据库加锁。支持数据库加锁的其他服务器包括JBoss、Sybase EAServer和Inprise Application Server。

WebLogic 7.0增加了一种“Optimistic Concurrency”策略，在该策略中，没有锁被保持在EJB容器或数据库中，但EJB容器在提交一个事务之前先进行一个针对争用更新的检查。我们在第7章中已经讨论过开放式加锁的优缺点。

只读与“常读”实体

数据被怎样访问将影响我们应该使用的加锁策略。因此，有些容器提供了针对只读数据的特殊加锁策略。同样，接下来的讨论体现了WebLogic术语，尽管这些概念不是WebLogic所特有的。

WebLogic 6.0及以上版本提供了一个叫做只读加锁(*read-only locking*)的特殊加锁策略。一个只读实体组件从不被一个客户软件更新，但可以定期地被更新(例如，为了响应底层数据库中的修改)。WebLogic从不调用一个具有只读加锁的实体组件的ejbStore()方法。不过，它按照部署描述符中所设置的一个正规间隔时间调用ejbLoad()方法。部署描述符可区分正常(读/写)和只读实体。JBoss 3.0提供了类似的功能度，因而把这称做“Commit Option D”。

WebLogic通过使容器生成式主接口实现去实现一个特殊的CachingHome接口，让用户能够控制高速缓存器。这个接口提供了无效化个别实体或所有实体的能力(一个只读实体组件的主接口可以被强制转换成WebLogic的专有CachingHome子接口)。在WebLogic 6.1及以上版本中，无效化在一个聚类中也管用。

如果我们知道数据将不被客户软件修改，只读实体组件会提供上佳的性能。它们也让实现一个“常读”模式成为了可能。这是通过映射一个只读和一个正常读-写实体组件到同一个数据上来实现的。这两个实体组件将有不同的JNDI名称。读取是通过只读组件来执行的，而更新则使用读-写组件。更新也可以使用CachingHome来无效化只读组件。

Dmitri Rakitine建议了“Seppuku”模式，该模式以更加可移植的方式实现相同的东西。Seppuku仅需要只读组件(而不是需要专有的无效化支持)来工作。如果一个非应用异常被遇到，该模式通过依靠容器来强制删除一个组件实例来无效化只读组件(我们将在第10章中讨论这种机制)。问题之一是EJB容器也被迫记录该错误，因而意味着服务器日志很快将会充满由“正常”活动所产生的错误消息。和Fat Key模式一样，Seppuku模式是一项充满灵感的发明，但也暗示了寻找整个问题的一种回避方法更可取。详细信息请参见<http://dima.dhs.org/misc/readOnlyUpdates.html>。

名称Seppuku是由BEA的Cedric Beust建议的，并参考了日本的切腹自杀习俗。该名称无疑比“Service-to-Worker”之类的乏味名称更令人难忘。

BEA的Tyler Jewell拥戴常读实体为EJB性能的救世主（请参见Tyler Jewell在<http://www.onjava.com/lpt/a/onjava/2001/12/19/eejbs.html>网页上的为实体组件辩护的文章）。他认为实体组件的一个“开发一次，部署n次”模型对释放它们的“真正威力”是必不可少的，而且建议了应该怎样根据使用模式来部署实体组件的判断标准。他主张每个使用模式的每个实体都有各自的部署。

与传统实体组件部署相比，这种多部署方法具有提供明显性能改进的潜力。但是，这种方法也有许多缺点：

- 依赖于只读组件来提供足够的性能不是可移植的（即使在EJB 2.1中，带有CMP的只读实体也仅被列为EJB规范未来发布中的可能增加，因而意味着至少到2004年以前它们将非标准的）。
- 存在把内存浪费在一个实体的多个副本上的可能性。
- 部署和使用多个实体需要开发人员的干预。实体的用户负责为他们的使用模式提供正确的JNDI名称（一个会话界面可以对EJB客户软件隐藏起这一点），进而部分地消除了这个缺点。
- 实体组件部署已经变得十分复杂；添加同一个实体组件的多个部署进一步使部署描述符和会话组件代码变得更复杂，而且被工具支持是不可能的。在涉及到容器管理式关系的地方，部署描述符的大小和复杂性将迅速增加。还有一些麻烦的设计问题。例如，在部署描述符中，一个只读组件应该链接到一个相关实体组件的几个部署中的哪一个部署上？

多部署的性能好处只有当数据被经常读取和被偶尔更新时才能获得。在涉及到静态参考数据的地方，高速缓存器离用户越近（比如在Web层中）将越好。多部署在我们需要集合操作，以及EJB CMP所提供的简单O/R映射不充分的情形中将没有帮助。

即使撇开这些问题不谈，如果在其他任何方面实现这种多部署方法的目标是不可能的，那么该方法也只演示了实体组件的“真正威力”。事实上，实体组件不是提供这种多高速缓存器的惟一方法。JDO和其他O/R映射解决方案也能使我们维持几个高速缓存区来支持不同的使用模式。

事务性实体高速缓存

使用只读实体组件和多部署是一种不方便的高速缓存形式，该形式要求开发人员花费大量精力来配置。这种形式之所以不能令人满意，是由于它不是真正可移植的，并且需要开发人员采取曲折技巧。如果实体组件高速缓存好得足以不用开发人员帮助就能工作，情况会怎么样呢？

Persistence PowerTier (<http://www.persistence.com/products/powertier/index.php>) 是一个带有一个事务性和分布式实体高速缓存器的正式产品。Persistence围绕着它的C++高速缓存解决方案，而不是给一个J2EE容器添加高速缓存支持来建立了它的J2EE服务器。

PowerTier对实体组件的支持与大多数其他供应商的这种支持有极大的差别。PowerTier实际上创建一个内存中的对象数据库来减轻底层数据库上的负荷。PowerTier使用一个共享式事务性高速缓存器，该高速缓存器允许内存中的数据存取和关系导航（关系被作为指针缓存在内存中，进而避免每当关系被遍历时运行SQL连接的需要）。

每个事务也被分配它自己的专用高速缓存器。对缓存数据的已提交修改被复制到共享式高速缓存器，并被透明地同步到底层数据库，以保持数据完整性。Persistence声称：对于偏向于支持读取的应用（比如许多Web应用），这可以把应用性能提高多达50倍。PowerTier的性能优化包括支持开放式加锁。Persistence提出了一个精细实体组件模型，并提供了从RDBMS表中生成实体（包括发现者方法）的工具。PowerTier还支持从一个实体组件模型中生成RDBMS表。

第三方EJB 2.0持久性提供商（比如TopLink）也声称实现了分布式高速缓存（请注意，TopLink通过它的专有O/R映射API提供类似的高速缓存服务，同时又不必使用实体组件）。

笔者还没有在实践中使用过这些产品中的任何一个，因此无法证实他们的销售团队的这些声称。不过，Persistence鼓吹了一些非常高容量的、任务关键的J2EE安装，比如Reuters Instinet联机交易系统和FedEx的后勤系统。

一个真正的好实体组件高速缓存器将极大地改进实体组件的性能。但是请记住，实体组件不是提供高速缓存的惟一方法。JDO体系结构允许JDO持久性管理器提供至少和任一实体组件高速缓存器同样精密的高速缓存。

实体组件的性能

使用实体组件可能会产生比主流的持久性替代方法更糟糕的性能，除非你的应用服务器有一个有效的分布式和事务性实体组件高速缓存器，否则开发人员需要把大量的精力投入到多部署上。在后一种情况中，性能将由应用的性质来决定；对数据主要进行只读访问的应用将会运转得很好，而对数据主要进行写访问的应用从高速缓存中将得不到什么好处。

这为什么重要呢？因为没有高效的高速缓存，实体组件性能可能会非常差。

来自实体组件模型的高效性能取决于下列条件：

- 数据在被访问时可能会被修改。除了只读实体的专有支持之外，实体组件采用一个读-修改-写模型。
- 修改发生在个别的被映射对象级别上，而不是作为一个集合操作（也就是说，更新可以利用Java中的个别对象来有效地完成，而不是对一个RDBMS中的多个元组进行更新）。

实体组件在许多情况下为什么有性能问题呢？

- 实体组件使用一种“以一概全”的方法。正如我们使用RDBMS时所见过的，实体组件抽象可能会使有效地访问持久性数据变得不可能。
- 实体组件约定是严格的，进而使编写有效的BMP代码变得不可能。
- 调整实体组件数据存取是很困难的，无论我们使用BMP还是CMP。
- 实体组件具有相当大的运行时开销，即便使用本地而不是远程接口（如果在一个EJB的部署描述符元素中没有为它定义安全角色或容器事务，许多应用服务器在该实体组件的实例于运行中被调用时可能会跳过事务和安全检查。当和本地调用结合起来时，这可删除实体组件的大部分开销。但是，这种情况不保证在所有服务器中都发生，而且一个实体组件将始终比普通Java类有一个大得多的开销）。

- 在实践中，实体组件性能常常会下降到O/R映射性能，而且不保证一个J2EE应用服务器供应商在这个领域内拥有很强的专门技能。

实体组件运行效率特别差，而且由于大型结果集的缘故而耗用过多的资源，尤其在结果集（比如搜索结果）没有被用户修改的时候。实体组件对总是在个别记录级别上被修改的数据有最佳的运行效率。

实体组件的工具支持

实体组件通常不含有业务逻辑，因此它们对自动生成来说是一个上佳的候选者。这是无关紧要的，因为实体组件往往含有许多代码——例如在获得器或设置器方法中。EJB 2.0 CMP实体的部署描述符也十分复杂，以至于不能可靠地手工创作。

如果使用实体组件，好的工具支持对工作效率是至关重要的。目前，有几种类型的工具是可以利用的，这些工具来自第三方供应商和应用服务器供应商。例如：

- 像Rational Rose和Together那样的对象建模工具。这些工具使用我们在第7章中已讨论过的对象驱动建模方法，进而使我们能够使用UML来形象地设计实体，并生成RDBMS表。这是方便的，但对象驱动建模会产生问题。
- 从RDBMS中生成实体组件的工具。例如，Persistence PowerTier就支持这种类型的建模。
- 从一个较简单的、较容易创作的表示中生成实体组件人工制品的工具。例如，EJBGen和XDoclet都能从一个实体组件实现类里的特殊Javadoc标记中生成本地和主接口、J2EE标准与应用服务器特有的部署描述符。此类简单工具是强有力的，可扩展的，并非常适合手工编码。

存在实体组件绝不应该被手工创作的一种特殊情况。赞同使用实体组件的一大理由是实体组件创作的工具支持已有相当高的水平。

小结

实际上，实体组件提供了EJB规范中所描述的一种基本O/R映射。这种映射具有标准化的优点。但是，它目前还没有达到主流O/R映射产品的那种威力。在使用RDBMS时，O/R映射未必是最佳解决方案。

实体组件在EJB 1.0规范中就有了预示，并且自EJB 1.1以来已经成为EJB规范的一个核心部分。EJB 2.0规范在实体组件支持方面引进了重要的增强，同时还引进了更高级的容器管理式持久性和本地接口。EJB 2.1规范进行了递增增强。

EJB 2.0规范帮助解决了与怎样使用实体组件有关的部分未定论的问题。例如，关于实体组件粒度的争论似乎由于赞同精细实体的缘故而得到了平息。这样的实体可以被赋予本地接口，以便使会话组件能够有效处理它们。EJB 2.0 CMP支持精细实体间关系的导航。EJB 2.0实体也可以使用其主接口上的方法来执行影响多个实体的持久性逻辑。

但是，实体组件在实践中有一个多变的历史记录。特别是，它们的性能常常是令人失

望的。

作为一项技术，实体组件的未来可能取决于可使用的CMP实现的质量。EJB 2.0规范要求应用服务器携带CMP实现；我们也正看到在O/R映射解决方案方面有雄厚背景的公司推出了第三方实现。这样的实现可能会超过该规范的要求而包括像高性能高速缓存、开放式加锁和EJB QL增强之类的特性。

实体组件在J2EE解决方案中会非常有价值。但是，最好的做法是把实体组件的使用看做一个实现选择，而不是看做应用体系结构中的一个关键成分。这可以通过在会话组件与实体组件之间使用一个由普通Java接口构成的抽象层来实现。

笔者觉得，一种独占性地使用实体组件执行数据存取的策略是行不通的，主要是由于性能的缘故。相反，一种从会话组件中（使用助手类）执行数据存取的策略是行得通的，而且可能会工作得更好。

至关重要的一点是：EJB层外部的构件不应该直接与实体组件打交道，而应该与调停实体组件访问的会话组件打交道。这一设计原理保证数据与客户端构件之间的正确分离，并最大限度地增加灵活性。EJB 2.0只允许我们给实体组件赋予本地接口，以保证这一设计原理得到遵守。

在会话组件自身中避免直接实体组件访问还有另外一个重大的理由：这避免把应用体系结构束缚于实体组件，因而提供了灵活性，尤其是在需要解决性能问题或利用当前数据源的各种能力时。

如果使用实体组件，笔者推荐下列总的指导原则：

- **如果你的EJB容器仅支持EJB 1.1，不要使用实体组件**

正如EJB 1.1规范中所规定的，实体组件不足以满足大多数现实中的需求。

- **使用CMP，不要使用BMP**

BMP提供了对持久性管理的更大控制权在很大程度上是虚假的。BMP实体组件开发和维护起来比CMP实体组件更困难，而且常常提供更糟糕的性能。

- **使用ejbHome()方法来完成你的数据上所需要的任何集合操作**

ejbHome()方法（能够作用于多个实体）帮助避开由实体组件模型所强加的行级访问，而行级访问会在其他方面妨碍有效的RDBMS使用。

- **使用精细实体组件**

“Composite Entity”模式（通常被推荐给EJB 1.1开发）在EJB 2.0中是过时的。实现粗糙实体需要做大量的工作，而且可能会提供一个很差的投资回报率。如果使用EJB 2.0风格的精细实体组件满足不了你的需要，很可能是实体组件的使用不得当。

- **不要把业务逻辑放在实体组件中**

实体组件应该只含有持久性逻辑。当使用EJB时，业务逻辑通常应该放在会话组件中。

- **仔细研究你的EJB容器用于实体组件的加锁和高速缓存选项**

实体组件是不是一个可靠的选择取决于你的EJB容器对它们的支持完善程度。你如何构建自己的实体组件部署对应用的性能可能会产生很大的影响。

下列指导原则主要适用于分布式应用：

- **绝不要允许远程客户软件直接访问实体组件；使用Session Facade模式，通过会话组件来调停实体组件访问**

如果远程客户软件直接访问实体组件，结果常常是过多的网络通信量和无法容忍的性能。当EJB层外部的任何构件直接访问实体组件时（即便在同一个JVM内），它们必然会变得与使用中的数据存取策略更密切。

- **给实体组件赋予本地接口，不要赋予远程接口**

通过远程接口访问实体组件已经证明速度太慢，以至于不太切实可行。远程客户软件应该使用一个会话门面。

- **在会话组件中创建值对象，不要在实体组件中创建值对象**

正如我们在第7章中已经讨论过的，值对象通常与用例有关，因此也与业务逻辑有关。

业务逻辑应该放在会话组件中，而不应该放在实体组件中。

个人见解：当实体组件首次被描述为EJB 1.0的一个可选特性时，笔者就对实体组件的构思非常热心。大约在2000年6月EJB 2.0的第一个草案被发布的时候，笔者就希望自己和其他设计师在使用EJB 1.1中的实体组件时所遇到的那些限制将得到解决，而且实体组件将成为EJB的一个强有力特性。但是，笔者的幻想逐渐破灭。

实体组件性能已经证明它在笔者已经见过的使用实体组件的大多数系统中都是一个问题。笔者逐渐相信，对实体组件的远程访问和用于实体组件的事务与安全管理基础结构从体系结构上看是没有必要的，并且是一项多余的系统开销。这些是会话组件要处理的问题。本地接口的引进仍把实体组件保留为重量级的构件。实体组件仍未能解决O/R映射的各种重大难题。

笔者的感觉是，JDO将代替实体组件成为J2EE中基于标准的持久性技术。笔者认为，把实体组件的地位降成EJB规范的一个可选部分有一个很充分的依据。对实体组件的支持大大超过了EJB 2.0规范的三分之一（与稍微超过少得多的EJB 1.1规范的五分之一相比），而且占有EJB容器的大部分复杂性。消除对实现实体组件的要求将会激励应用服务器市场中的竞争和创新，而且将会帮助JDO成为访问持久性数据的一个统一而又强有力的J2EE标准。但是，这仅仅是笔者的个人见解！

笔者更喜欢使用一个由组成一个持久性门面的DAO所构成的抽象层从会话组件中管理持久性。这种方法使业务逻辑代码独立于任一特定的持久性模型。

正如下一章中将要介绍的，我们将在本书的示例应用中采用这种方法。

第9章 实际的数据存取

在本章中，我们将详细了解几项领先的数据存取技术，并选择一两项技术用在本书的示例应用中。

本章中将讨论的这些数据存取技术可以用在一个J2EE应用中的任何地方。和实体组件不同，它们不局限于用在一个EJB容器中。这种做法的明显优点是测试和体系结构的灵活性。不过，当我们在EJB容器内实现数据存取时，仍可以充分利用会话EJB CMP。

本章的焦点是访问关系数据库，因为我们已经在前面的章节中说明过，大多数J2EE应用都使用它们。

我们将考虑除了实体组件之外的基于SQL的技术和O/R映射技术。我们将了解Java Data Object (JDO) 的潜在重要性，这是用于持久Java对象的一个新增API——可能会标准化Java 中O/R映射的使用。

我们将详细了解JDBC API，因此将描述如何避免常见的JDBC错误，并讨论经常被疏忽的一些细小之处。

由于JDBC API是相当低级的，而且使用它很容易出错，并需要大量的代码，所以让应用代码使用建立在它上面的那些高级API是十分重要的。我们将了解一个JDBC抽象框架的设计、实现和使用，而这个框架将极大地简化JDBC的使用。这个框架将用在本书的示例应用中，并且可以用在任何使用JDBC的应用中。

正如我们在第7章中已经讨论过的，最好是把数据存取与业务逻辑分离开来。我们可以通过把数据存取的细节隐藏在一个O/R映射框架或Data Access Objects (DAO) 模式之类的一个抽象层背后来实现这种分离。

最后，当我们使用DAO模式和本章中所讨论的JDBC抽象框架来实现示例应用的一些核心数据存取代码时，我们将会看到正处于工作中的DAO模式。

数据存取技术选择

让我们首先来回顾一下J2EE应用可以采用的一些主要数据存取技术。这些技术可以分成两大类：使用关系概念且基于SQL的数据存取，以及基于O/R映射的数据存取。

基于SQL的技术

下列技术被完全定位成处理关系数据库。因此，它们使用SQL作为检索和操纵数据的手段。虽然这要求Java代码处理关系（而不是对象）概念，但是却允许了RDBMS结构的有效使用。

需要注意的是：在数据存取代码中使用RDBMS概念，并不意味着业务逻辑将依赖于SQL 和RDBMS。我们将使用DAO模式（参见第7章）来分离业务逻辑与数据存取实现。

JDBC

与关系数据库的大多数通信，无论由一个EJB容器、一个第三方O/R映射产品还是一名应用开发人员来处理，可能都是基于JDBC的。实体组件——还有O/R映射框架的大部分魅力都是基于这样一个假设：使用JDBC是容易出错的，并且对应用开发人员来说太复杂。事实上，这是一个可能有问题的争论，只要我们使用适当的助手类。

JDBC基于SQL，而这绝不是一件坏事情。SQL不是一项神秘的技术，而是一种已经过实践检验的、简化了许多数据操作的语言。RDBMS与Java应用之间可能存在一个“阻抗不匹配”，但SQL是查询和操纵数据的一种有效语言。许多数据操作若利用SQL来完成，需要使用的代码行数远远少于处理被映射对象的Java类中的代码行数。专业的J2EE开发人员需要拥有SQL的全面知识，并且绝对不能不知道RDBMS基础知识。

当我们需要调用存储过程、执行特殊自定义查询或使用专有RDBMS特性时，JDBC也是一项理想的技术。

关键是我们怎样使用JDBC。一种幼稚的方法（即在应用代码中到处乱放JDBC代码和SQL语句）是导致灾难的一大根源。这种方法使整个应用束缚于某个特定的持久性策略，在数据库模式发生变化时准时出问题，并且让应用代码含有抽象级别的不正确混用。不过，只要我们遵守几项不错的原则，JDBC就可以有效地用在应用代码中：

- 在凡是可能的地方都分离开业务逻辑与JDBC存取。通常，JDBC代码只应该在数据存取对象中才能发现。
- 避免直接处理JDBC API的原始JDBC代码。JDBC错误处理十分麻烦，所以会严重降低工作效率（例如，需要一个finally块来实现某种东西）。像错误处理那样的低级细节最好是留给暴露一个高级API给应用代码的助手类。这在不牺牲对被执行SQL的控制权情况下是有可能的。

“让容器处理持久性总是比编写SQL来处理持久性好”的J2EE正统观念是有问题的。例如，虽然调整容器生成的语句会很困难或者是不可能的，但是在像SQL*Plus那样的工具中测试并调整SQL查询是有可能的，这样可以在不同会话访问相同数据时检查性能和行为。只有在重要的对象高速缓存在一个O/R映射层中切实可行的地方，使用一种O/R映射来编写代码才有可能等于或超过JDBC的性能。

使用JDBC来管理持久性绝不会出现任何问题。在许多情况中，如果我们知道自己正在处理一个RDBMS，那么只有通过JDBC才能利用该RDBMS的全部能力。

但是，不要直接从会话EJB或DAO之类的业务对象中使用JDBC。要使用一个抽象层把你的业务构件与低级JDBC API隔离开。如果可能的话，要使这个抽象层的API变成非JDBC特有的（例如，设法避免暴露SQL）。我们将在本章的较后面考虑这种抽象层的实现和使用。

SQLJ

SQLJ是JDBC的一个代用品，提供Java代码与RDBMS之间的更紧密集成。它提供和JDBC相同的基本模式，但简化了应用代码，并且可能会改进运行效率。它是由包括Oracle、IBM、Informix、Sun和Sybase在内的一组公司所开发的。SQLJ由3部分组成：

- **Part 0—Embedded SQL (嵌入式SQL)**

这一部分提供了通过转义SQL来嵌入静态SQL语句到Java代码中的能力。对比一下， JDBC使用动态SQL，并基于API调用，而不是基于一种转义机制。SQLJ Part 0已经被正式接纳为一个ANSI标准。

- **Part 0—SQL Routines (SQL例程)**

这一部分定义了一种把Java静态方法作为存储过程来调用的机制。

- **Part 0—SQL Types (SQL类型)**

这一部分包括了使用Java类作为SQL用户定义数据类型的各种规范。

SQLJ Part 0即Embedded SQL在功能度上可以与JDBC相媲美。其语法能使SQL语句表达得比使用JDBC表达得更精确，并且方便了从数据库中获取Java变量值和写Java变量值到数据库上。SQLJ预编译器把SQLJ语法（带有嵌入式SQL的Java代码）翻译成正式的Java源代码。嵌入式SQL的概念绝不是新东西：Oracle的Pro*C和其他产品采取了接近于C和C++的相同方法，甚至有用于COBOL的类似解决方案。

SQLJ提供了SQL中的方便转义和Java变量到SQL常数的绑定。下列例子演示了这两个特性：#sql转义语法的使用，以及通过在嵌入式SQL中使用:语语法来绑定Java变量。本例插入一个新的行到一个表中，进而获取Java变量中的值；然后，把新数据选回到那些变量中。

```

int age = 32;
String forename = "Rod";
String surname = "Rod";
String email = "rod.johnson@interface21.com";
#sql {
INSERT INTO
    people (forename, surname, email)
VALUES
    (:forename, :surname, :email)
};

#sql {
SELECT
    forename, surname, email
INTO
    :forename, :surname, :email
FROM
    people
WHERE
    email='rod.johnson@interface21.com'
};

```

需要注意的是，笔者省略了错误处理：像在JDBC中一样，我们必须在SQLJ中捕捉java.sql.SQLExceptions，并且必须保证我们在发生错误的情况下关闭连接。不过，我们不必处理JDBC Statement和PreparedStatement对象，进而使这段代码不像JDBC等效代码那么冗长。SQLJ Part 1和Part 2试图标准化数据库特有的功能度，比如存储过程。

SQLJ可以用和JDBC相同的方式被用在J2EE体系结构中，比如在数据存取接口的实现中。因此，SQLJ与JDBC之间的选择是一个在应用内部只有局部意义的问题。SQLJ胜过JDBC的优点包括：

- SQLJ提供了SQL的编译时而不是运行时检查。若使用JDBC，我们只有在运行时才会知道一条SQL语句是无意义的，但SQLJ将在编译时挑出这条语句。
- 性能可能会通过静态而不是动态SQL的使用得到提高。

- SQLJ代码的编写和理解比JDBC代码更简单。

SQLJ的缺点包括：

- 开发 - 部署周期比较复杂，因而需要SQLJ预编译器的使用。
- SQLJ没有简化JDBC笨拙的错误处理需求。
- SQLJ语法不如Java语法那么精致。另一方面，SQLJ实现一些任务所使用的代码行数可能会比使用JDBC的Java代码少得多。
- 虽然SQLJ是一个标准，但一个SQLJ解决方案不如一个JDBC解决方案那么可移植，因为不是所有的RDBMS供应商都支持SQLJ（如果SQLJ用来实现一个以某个特定数据库为目标的DAO，这未必是一个缺点）。

尽管它很有潜力，并且已有几年的发展历史，但SQLJ似乎还没有得到广泛使用，而且仍被看做和Oracle是一体的。

要想了解关于SQLJ的较详细信息，请参见www.sqlj.org，以及O'Reilly所出版的“Java Programming with Oracle SQLJ”一书（ISBN 0-596-00087-1）——<http://www.oreilly.com/catalog/orasqlj/>。

O/R映射技术

O/R映射框架提供一个与JDBC和SQLJ之类的API完全不同的程序设计模型。它们也可以用来在一个J2EE应用内的任何地方实现数据存取。有些产品提供一个针对EJB 2.0的可插入CMP持久性管理器，而且这个管理器提供比应用服务器所携带的实现更丰富的功能度。

在现实经验方面，实体组件远远落后于那些最好的O/R映射框架。不过，O/R映射框架一直是很昂贵的产品（尽管这种状况在2002年似乎正在发生改变），而且一直束缚于专有API。据笔者所知，目前还没有任何开放源产品提供一个企业强度的O/R映射解决方案（尽管笔者非常乐意听到自己是错的）。

一个新登场的新来者是Sun的JDO规范。和J2EE一样，这只是一个规范——实质上是一个API，而不是一个产品。不过，许多公司已经支持或者正在实现这个JDO规范。重要的是，一些主流的已有O/R映射框架现在通过使用它们的现有O/R映射基础结构来支持JDO。有些ODBMS供应商也正在支持JDO。JDO可能会把Java中的O/R映射使用标准化到JDBC为Java中的ODBMS存取所提供的标准化程度。

正式的商业产品

首先，让我们来看一看商业O/R映射产品，因为它们已经有了很长的发展历史，并且已在实践中得到过检验。这个领域中有许多相互竞争的产品，但我们将只讨论得到最广泛使用的两个产品：TopLink和CocoBase。关于这两个产品的许多论点适用于所有O/R映射产品。

TopLink

TopLink或许是市场领头羊，并且提前实现了EJB规范。它经历了所有权的几次转变，现在是Oracle 9i Application Server的一部分，尽管它可以被单独下载，并能和其他应用服务器一起使用。它提供一个可插入的EJB 2.0 CMP解决方案，尽管它的文档暗示TopLink的

开发人员不赞同实体组件方法。TopLink允许实体组件和轻量级的持久性Java对象一起混用。

TopLink的特性列表说明了实体组件必须超过标准EJB 2.0 CMP多远，才能成为一个成熟的O/R映射框架。下面是在EJB规范中见不到的一些TopLink特性：

- 映射一个Java对象到多个表的能力。
- 对Java对象继承性的支持。
- 对开放式加锁的支持。
- 对嵌套事务（TopLink说法中的“工作单元”）的支持。不仅EJB不支持这个概念，而且J2EE也不支持这个概念。
- 数据在数据库与对象之间的可自定义变换。除了简单地映射一个列值到一个对象属性上，TopLink还允许程序员自定义映射。
- 自定义SQL来取代依赖于已生成的SQL的能力。
- 对存储过程的支持。
- 减少访问底层数据库次数的高速缓存。高速缓存在节点之间被同步。

和大多数O/R映射框架一样，TopLink可以和数据库驱动或对象驱动建模一起使用。如果存在一个数据库，TopLink的映射工具就可以用来生成使用该数据库的Java对象。作为一个选择，TopLink也可以用来从已映射的Java对象中生成一个数据库模式。映射被定义在TopLink的GUI Mapping Workbench中。已映射的对象一般不必含有TopLink特有的代码（和CMP实体组件不同，因为它们必须实现一个指定API）；不过，使用已映射对象的代码严重依赖于专有API。

TopLink提供了它自己的基于对象的查询语言。下列例子摘自TopLink 4.6演示，使用一个Expression对象来筛选一个给定类的对象，其中该给定类的Budget属性大于或等于一个给定值：

```
Expression exp = new ExpressionBuilder().get("budget").greaterThanEqual(4000);
Vector projects = session.readAllObjects(LargeProject.class, exp);
```

另外，TopLink 4.6支持JDO作为一个查询API。

TopLink一直是很昂贵的（价格比得上领先的J2EE应用服务器）。在Oracle于2002年6月收购它之后，它可免费供开发使用，并且它的产品定价可能更吸引人。

TopLink的长处是：它是一个已经过检验的产品，并且具有强有力而又可配置的O/R映射能力。它的主要弱点是：使用它的代码依赖于专有库。但是，它对JDO的支持可能会抵消这个缺点。

关于TopLink的进一步信息，请参见<http://otn.oracle.com/products/ias/toplink/content.html>。

CocoBase

产自Thought Inc的CocoBase是另一个得到公认的O/R映射产品。和TopLink一样，CocoBase包括一个EJB 2.0 CMP持久性管理器，并提供一个分布式高速缓存器。它还提供透明持久性：要被保留的对象不必实现特殊的接口。映射被保存在要被保留的类外面。

CocoBase支持高级的O/R映射，比如映射一个Java对象到多个表的能力。和TopLink一样，调整生成的SQL是有可能的。CocoBase与底层数据库集成的紧密程度高于许多O/R映射解决方案：调用存储过程很容易，并且查询是用SQL而不是用一种专有查询语言写成的。

关于CocoBase的进一步信息，请参见<http://www.thoughtinc.com/>。

Java Data Objects (JDO)

JDO是在Java Community Process领导之下开发的一个新规范（1.0版于2002年3月发布），描述了Java对象的“持久性存储器中立”持久性。虽然详细讨论JDO超出了本书的范围，但笔者觉得JDO对J2EE开发人员来说是一个十分重要的API，并希望接下来的一节将足以满足读者的需要。

JDO最有可能用做一个O/R映射，但它不束缚于RDBMS。例如，JDO可能会成为访问ODBMS的标准API。

与实体组件模型相反，JDO不要求将被保留的对象去实现一个特殊的接口。大多数普通Java对象都能被保留，只要它们的持久性状态被保存在它们的实例数据中。虽然用在JDO持久性管理器（persistence manager）中的对象必须实现javax.jdo.PersistenceCapable接口，但开发人员通常不编写实现这个接口的代码。PersistenceCapable接口所需要的附加方法，通常由JDO供应商所提供的一个字节码增强器（byte code Enhancer）添加到已编译的类文件上，而字节码增强器是指这样一个工具：它能够获取普通类文件，并“增强”它们供一个JDO持久性引擎使用。

JDO比实体组件的量级更轻，部分原因是正被保留的对象更接近普通Java类。JDO在创建新对象方面可能有低得多的系统开销：查询返回大型结果集时的一个重要因素。

应用代码与一个JDO PersistenceManager实例打交道，而该实例一次只处理一个事务。PersistenceManager实例必须从JDO供应商所提供的一个PersistenceManagerFactory中获得。这允许了多种多样的高速缓存方法，进而把选择权留给了JDO供应商。

高速缓存除了发生在事务内，还可以发生在PersistenceManagerFactory级别上。有几个JDO实现支持聚类环境中的高速缓存。JDO还允许对高速缓存的程序性应用控制。实体组件没有提供类似的控制。JDO还明确支持开放式加锁和持久Java对象的继承性；同样，这也可程序性地控制。

JDO映射被定义在XML文档中，而这种文档理解起来比CMP部署描述符更简单，更容易。不过，映射在JDO实现之间不完全是可移植的，因为它们依赖于供应商特有的扩展。

JDO目前还不是J2EE的一部分。可是，JDO最终将成为一个和JDBC与JNDI一样的必需API似乎是有可能的。

尽管Sun不肯承认，但JDO与实体组件之间存在明显的重叠。它们都相当于一个面向对象的映射，并定义了一种用于操作持久性数据的查询语言。我们很难看出在J2EE内拥有两个完全不同的O/R映射标准，尤其是拥有两种不同的查询语言有什么正当的理由。不过，JDO补充J2EE和EJB（与实体组件相反），而不是与它们竞争。

JDO查询语言叫做JDOQL。一个JDO查询就是一个可重用、线程安全并且定义一个结果

类和一个JDO筛选器的对象。JDOQL筛选器表达式就是Java表达式，尽管它们被保存在串字面值或变量中，并且不能在编译时对照持久性存储器被验证。和EJB QL不同，JDOQL用在Java代码中，因而意味着查询是动态的，而不是静态的。下列例子说明了一个查询的构造，其中该查询寻找具有一个给定email地址和password（这两者都是实例变量）的Customer对象：

```
Query loginQuery = persistenceManager.newQuery(Customer.class);
loginQuery.setFilter("*email == pEmail && password == pPassword");
loginQuery.declareParameters("String pEmail, String pPassword");
loginQuery.compile();
```

这个查询可以按如下所示来执行：

```
Collection c = (Collection) loginQuery().execute("rod.johnson@interface21.com",
    "rj");
Customer cust = (Customer) c.iterator().next();
```

对由查询所产生的对象（比如上述Customer对象）做修改通常会导致后援数据存储器被透明地更新。但是，JDO允许对象被明确地与数据存储器断开联系（如果需要）。一旦JDO被检索，透明地导航关联（RDBMS中的连接）是有可能的。

在JDO Query接口上的几个execute()方法中，最重要的是接受一个Object数组作为一个参数；上述例子使用了接受两个串并调用那个通用方法的便利方法。

JDO由于使用串查询筛选器，而不是一个基于对象的查询API（比如TopLink的专有API）而一直遭到批评。编译时检查的丢失是令人遗憾的，但这种方法管用，而且使用方便。例如，使用Java语言来构造OR和AND的复杂组合就很容易，而且这种组合对Java开发人员来说是很容易理解的。

和JDBC一样，JDO也提供一个无需全局事务基础结构就能使用的事务管理API，但是这在J2EE中是不适合的。JDO实现能够检测并在JTA事务内工作，无论这些事务是由用户代码创建的，还是由EJB CMT产生的。

JDO规范的第16章详细描述了怎样才能从EJB中使用JDO来处理持久性。与会话组件的集成特别有吸引力，因为JDO操作可以使用由会话组件所提供的声明性事务界定。虽然这种集成有时会得到拥护，但弄明白从带有BMP的实体组件中使用JDO的意义是很困难的。JDO已经提供了数据库的一个对象观点，进而致使由实体组件构成的那个附加抽象层变得多余，甚至变得有害（结果很可能是JDO和EJB这两者所支持的功能度的一个最小公分母）。

由于所有JDO异常都是运行时异常，所以使用JDO的任一代码都必须捕捉它们（尽管它可以在异常可能是可重获的地方）。与使用像JDBC那样使用已检查异常的API相比，这使代码量明显减少并明显改进了阅读性。

关于JDO的进一步信息，请参考下列资源：

- <http://access1.sun.com/jdo/>
由JDO规范主要起草者Craig Russell维护的索引页面。链接到其他有用的资源。
- <http://www.jdocentral.com/index.html>
JDO Portal。这些论坛是特别有趣的，它们的特色是讨论JDO开发中的许多实际问题。

- <http://www.onjava.com/pub/a/onjava/2002/02/06/jdo1.html>
对JDO的简单介绍，包括一个完整的例子。
- http://theserverside.com/discussion/thread.jsp?thread_id=771#23527
比较JOD与实体组件的ServerSide.com讨论。
- <http://www.ogilviepartners.com/JdoFaq.html>
由一家私人咨询组织维护的JDO FAQ。
- <http://www.solarmetric.com/>
Kodo（首批JDO实现之一）的主页。Kodo使用简单，并支持聚类中的高速缓存。
Kodo的文档被编写得十分完善，并且它是认真学习JDO的一个好选择。
- <http://www.prismtechnologies.com/English/Products/JDO/index.html>
OpenFusion（来自Prism Tech的一个JDO实现）的主页。

对J2EE来说，JDO可能是一个非常重要的技术。但是，在撰写本书的时候（2002年的年中），JDO尚待证明它已成熟得足以用在任务关键的企业应用中。

当JDO成熟时，笔者怀疑它可能会致使实体组件变得过时。JDO保留了实体组件的那些好的品质（比如向应用代码隐藏数据存取细节，O/R映射解决方案，在由会话组件界定的事务内的使用，以及用高速缓存对象来减少对数据库进行访问的潜力），并消除了它们的许多问题（比如不能使用普通Java类作为持久性对象、对数据对象的无故远程访问以及重量级的运行时基础结构）。

为示例应用选择一种数据存取策略

我们已经在第7章中提过，示例应用从一个O/R映射层中不会得到什么好处。例如，根本没有高速缓存的机会。我们也见到过，存储过程的使用可以有效地消除RDBMS内的部分数据管理复杂性。这就意味着JDBC是显而易见的数据存取策略。

由于保证我们的设计保持可移植是非常重要的，所以我们将使用第7章中讨论过的DAO模式，把JDBC的使用隐藏在一个由数据存取接口所构成的抽象层后面。这些接口将是数据库无关的。它们不依赖于JDBC和RDBMS概念，进而允许有各种不同的可能实现。

在本章的剩余篇幅中，我们将：

- 比较仔细地看一看JDBC，进而了解我们为编写坚固、有效的代码而必须弄清楚的一些细微之处。
- 介绍一个由通用基础结构类所构成的抽象层的实现和使用，其中这些类可以简化JDBC的使用，并最小化JDBC特有错误的出现概率。
- 了解示例应用的部分数据存取需求的实现：使用JDBC和我们的通用抽象层。

JDBC的细微之处

由于我们将在示例应用中使用JDBC，所以先来了解一下实际使用JDBC的主要挑战，并分析一下经常被忽视的JDBC API的一些细微之处。这是一个重要的题目，据笔者看，J2EE应用经常含有草率的JDBC代码，这类代码会引起很大的问题。

正确的异常处理

编写坚固JDBC代码的主要挑战是保证正确的异常处理。大多数JDBC API方法都会抛出已检查的java.sql.SQLException。我们不仅要保证捕捉这个异常，而且还要保证万一一个异常被抛出时执行相应的清除操作。

下列代码清单（执行一个无关紧要的SELECT操作）演示了一个常见的JDBC错误，该错误会产生严重的后果。

```
public String getPassword(String forename) throws ApplicationException {
    String sql = "SELECT password FROM customer WHERE forename='"
        + forename + "'";
    String password = null;
    Connection con = null;
    Statement s = null;
    ResultSet rs = null;
    try {
        con = <get connection from DataSource>;
        s = con.createStatement();
        rs = s.executeQuery(sql);

        while (rs.next()) {
            password = rs.getString(1);
        }
        rs.close();
        s.close();
        con.close();
    } catch (SQLException ex) {
        throw new ApplicationException("Couldn't run query [" + sql + "]", ex);
    }
    return password;
}
```

此类代码会使整个应用崩溃，甚至会危及到数据库。问题是那些突出显示的行每个都会抛出一个SQLException。如果发生这种情况，我们将无法关闭那个Connection，因为只有当我们到达那个try块的最后一行时，该连接才被关闭。结果将是一个“漏掉的”连接，即从应用服务器的连接池中偷偷溜掉的连接。虽然应用服务器最终可能会检测到这个问题，并释放这个溜掉的连接，但对该连接池的其他用户所造成的影响可能是很严重的。每个连接都花费RDBMS资源来维持，并且合并成池的连接是一个很宝贵的资源。最终，池中的所有连接都可能会变得不可利用，进而使应用服务器暂停工作；而且，应用服务器可能无法增加连接池的大小，因为数据库不再能支持并发连接。

这个方法的下列正确版本增加了一个finally块（突出显示的部分），以保证连接总是得到关闭。请注意：和第一个版本中一样，在这个版本中，我们需要抛出一个封装任何遇到的SQLException的应用特有异常（ApplicationException）：

```
public String getPassword(String forename) throws ApplicationException {
    String sql = "SELECT password FROM customer WHERE forename='"
        + forename + "'";
    String password = null;
    Connection con = null;
    Statement s = null;
    ResultSet rs = null;
    try {
        con = <get connection from DataSource>;
        s = con.createStatement();
        rs = s.executeQuery(sql);

        while (rs.next()) {
            password = rs.getString(1);
        }
    } finally {
        if (rs != null)
            rs.close();
        if (s != null)
            s.close();
        if (con != null)
            con.close();
    }
    return password;
}
```

```
String password = null;
Connection con = null;
Statement s = null;
ResultSet rs = null;
try {
    con = <get connection from DataSource>;
    s = con.createStatement();
    rs = s.executeQuery(sql);

    while (rs.next()) {
        password = rs.getString(1);
    }
    rs.close();
    s.close();
}
catch (SQLException ex) {
    throw new ApplicationException("Couldn't run query [" + sql + "]", ex);
}
finally {
    try {
        if (con != null) {
            con.close();
        }
    }
    catch (SQLException ex) {
        throw new ApplicationException("Failed to close connection", ex);
    }
}
return password;
}
```

这个版本是比较坚固的，但也是比较冗长的。编写39行代码来完成几乎最简单的JDBC操作是荒唐可笑的。显而易见，我们需要一个比直接使用JDBC更好的解决方案。

当直接使用JDBC时，必须使用一个finally**块来保证连接得到关闭，即便在出现了异常的情况下。**

从SQLException中提取信息

JDBC异常处理的另一个重要方面是从SQLExceptions中提取尽可能多的信息。令人遗憾的是，JDBC使这变得相当麻烦。

java.sql.SQLExceptions可以链接起来。SQLException上的getNextException()方法返回此故障所关联的下一个SQLException（或null，如果没有进一步的信息）。检查深一层异常是否必要好像取决于RDBMS和驱动程序（例如，笔者发现这在Cloudscape 3.6中是非常有用的，但在Oracle中常常是没有必要的）。

虽然JDBC为几乎所有问题都使用同一个异常（SQLException），但从一个SQLException中提取有用的细节是有可能的。SQLException还包括了两个可能有用的错误代码：供应商代码和SQLState代码。

供应商代码是由getErrorCode()方法所返回的一个int。顾名思义，供应商代码在供应商之间可能会不同，即便对死锁之类的常见错误条件来说也是如此。因此，依赖于它的代码将不是可移植的。例如，就Oracle而言，供应商代码将是Oracle在它的所有错误消息中所报告

的数字码（不仅仅报告给JDBC），比如“ORA-00060: deadlock detected while waiting for resource”。有时，这个不可移植的功能度是至关重要的，因为以其他方式精确地确定一条SQL语句有何错误是不可能的。

我们甚至可以为自定义的应用异常使用我们自己的一个错误代码。下面来看一个包含了下面这一行的Oracle存储过程：

```
raise_application_error(-20100, 'My special error condition');
```

这个错误代码将像一个标准错误代码一样显示给JDBC：因该SQLException而显示的消息将是“ORA-20100: My special error condition”，并且供应商代码将是20100。

SQLException.getSQLState()方法返回一个5字符串，并且这个串至少在理论上是一个可移植的错误代码。前两个字符含有一个类代码中的高级信息；接下来的3个字符含有更具体（常常供应商特有）的信息。那些SQLState代码在JDBC API中没有被正式说明。笔者能够找到的惟一免费的联机文档是http://developer.mimer.se/documentation/html_91/Mimer_SQL_Mobile_DocSet/App_return-status2.html#1110406。

令人遗憾的是，那些SQLState代码不被保证将得到所有数据库的支持，而且可能没有包含足够确定一个问题的性质的信息。大部分数据库具有比SQLState代码多出许多的供应商代码和描述它们的较完善文档。有时，根本没有表示一个问题的标准SQLState代码，比如Oracle的8177（“can't serialize access for this transaction”），该代码可能包含了对一个J2EE应用来说非常重要，但对某个特定RDBMS的实现来说又是惟一的信息。

除了SQLException之外，我们还必须了解java.sql.SQLWarning异常。和SQLException一样，这是一个已检查异常（它通常是SQLException的一个子类），但它实际上从不被JDBC API抛出。诸如读取时的数据截取之类的非致命性错误可以被报告为SQL警告，它们在不愉快的语句执行完毕之后被附加到ResultSets、Statements或Connections上。这些JDBC API对象每个都有一个getWarnings()方法，如果根本不存在警告，这个方法返回null，否则返回一个SQLWarnings链（类似于一个SQLException链）中的第一个警告。当采取一个新操作时，警告被复位。

直接使用JDBC不是一个可行选择的另一个原因是警告。在一个实现用来简化JDBC的框架中，在出现警告时抛出一个异常（如果愿意）或把警告保存起来供将来参考是一件容易的事情。在直接使用JDBC API的代码中，检测警告的过程会增加很大的代码量；这是如此麻烦，以至于在实践中根本行不通，除非我们借助于将引起许多其他问题的块状剪切和粘贴操作。

PreparedStatement问题

了解java.sql.PreparedStatement接口与java.sql.Statement接口之间的区别是十分重要的。如果使用一条准备语句（prepared statement），我们就提供含有?占位符（代表赋值变量（bind variable））的必要SQL，并设置代表每个占位符的参数。如果使用一条普通语句，我们就提供整个SQL串，其中包括每个参数值的一个串表示。

一般说来，应该首选准备语句。

首先，它们极大地简化了串和其他对象的处理。如果不使用准备语句，我们将需要转

义串中的非法字符。例如，下列查询将会失败：

```
SELECT id FROM customers WHERE surname='d'Artagnan'
```

姓氏d'Artagnan含有一个非法的"字符，数据库将会把该字符看做一个串字面值的结束符。在Oracle中，我们将会得到错误ORA-01756和消息quoted string not properly terminated。我们将需要按照如下所示来修改该查询，以便使它工作：

```
SELECT id FROM customers WHERE surname='d\'Artagnan'
```

我们也将会遇到DATE和其他标准对象类型的问题，对于上述问题，我们将需要使用一种RDBMS特有的串格式。如果使用准备语句，我们就不必担心转义或对象转换。如果我们调用那些正确的setXXXX()方法，或者知道对应于该RDBMS类型的适当JDBC类型（被定义在java.sql.Types类中的参数之一），JDBC驱动程序将会透明地负责转换和任何转义。我们的JDBC代码也将是更可移植的，因为该驱动程序将会知道如何在标准Java对象与适合目标数据库的SQL类型之间做转换。

其次，准备语句也可能比串语句更有效率，尽管任何性能增益的程度在平台之间将是不同的。准备语句比普通串语句的创建代价高，但如果它们被重用，有可能会减少JDBC驱动程序，甚至数据库必须完成的工作量。当一条准备语句被首次执行时，它会被数据库语法分析和编译，并且可能会被放在一个语句高速缓存器中。于是，数据库能够存储该语句的执行路径，而这么做可能会降低它将来需要执行该语句时所做的工作量。相反，正规的SQL串语句不能被这样高速缓存，如果它们有变量参数，因为含有不同字面参数值的语句对数据库来说是不同的。这不仅妨碍了有用的高速缓存，而且还因为阻碍了其他语句的高速缓存而显著地增加了数据库上的总负荷。数据库的语句高速缓存器可能会充满了明显不同的语句，从而因没有空间用于真正可高速缓存的语句而影响同一个应用或其他应用。

令人遗憾的是，无法确信这种行为对所有数据库、JDBC驱动程序和应用服务器都适用，因为JDBC 2.0 API（被保证与J2EE 1.3相适应的版本）和J2EE规范没有要求准备语句的高速缓存。我们自己也无法保证高速缓存；如果我们需要关闭自己获得的连接（在J2EE环境中通常需要这么做），那么在创建了准备语句后，我们就必须关闭它们。这意味着为了准备语句的任何重用，我们需要依靠应用服务器的一个JDBC连接池而不是我们的代码。令人高兴的是，大多数现代应用服务器似乎都把准备语句合并成池。这么做所产生的性能好处将取决于底层JDBC驱动程序和数据库。

JDBC 3.0引入准备语句高速缓存作为标准。在很大程度上，这对开发人员是透明的：假设一个连接池实现可能会提供对准备语句高速缓存的支持。因此，准备语句将来会变得越来越有吸引力。

宁选PreparedStatements，而不选Statements。它们减小了类型错误的出现概率，并且可能会改进运行效率。

一个通用的JDBC抽象框架

仅了解JDBC API和正确使用它方面的问题是远远不够的。直接使用JDBC简直太费劲，而且太容易出错。我们不仅需要了解JDBC，而且还需要了解使JDBC的使用变得更容易的一

个代码框架。

在本节中，我们就来了解一个这样的解决方案：一个通用的JDBC抽象框架，我们将在示例应用中使用这个框架，并且它可以显著地简化应用代码和最小化出错的概率。

因直接使用它太低级，以致JDBC API不能成为一个可靠的选择。当使用JDBC时，应始终使用助手类来简化应用代码。不过，不要试图编写你自己的O/R映射，要使用一个现有的解决方案。

在本节中，我们将介绍示例应用中所使用的JDBC抽象框架的实现和使用，该框架被包含在了本书的配书下载内的框架代码中。这是一个通用的框架，读者可以把它用在自己的应用中，以简化JDBC的使用和减少JDBC特有错误的出现概率。

动机

编写低级JDBC代码是一件痛苦的事情。问题不在于SQL，因为它使用和理解起来通常是相当容易的，而且只占整个代码量的极少部分，而是在于使用JDBC所附带的事情。JDBC API不是特别精致，而且用在应用代码中显得太低级。

正如我们已经见过的，使用JDBC API执行非常简单的任务都是相当复杂的，而且容易出错。现在来看一下下列例子，它查询我们的订票系统中的空座位的ID：

```
public List getAvailableSeatIds(DataSource ds, int performanceId,
    int seatType) throws ApplicationException {

    String sql = "SELECT seat_id AS id FROM available_seats "
        + "WHERE performance_id = ? AND price_band_id = ?";
    List seatIds = new LinkedList();

    Connection con = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        con = ds.getConnection();
        ps = con.prepareStatement(sql);
        ps.setInt(1, performanceId);
        ps.setInt(2, seatType);
        rs = ps.executeQuery();

        while (rs.next()) {
            int seatId = rs.getInt(1);
            seatIds.add(new Integer(seatId));
        }
        rs.close();
        ps.close();
    }
    catch (SQLException ex) {
        throw new ApplicationException("Couldn't run query [" + sql + "]", ex);
    }
    finally {
        try {
            if (con != null)
                con.close();
        }
        catch (SQLException ex) {
            // Log and ignore
        }
    }
    return seatIds;
}
```

虽然SQL是无足轻重的（由于使用数据库中的一个视图隐藏了必需的外连接），但执行这个查询需要30余行JDBC代码，实际上其中只有少数代码才有作用（笔者已经突出显示了这些有作用的代码行）。这种不提供功能度的“探测（plumbing）”代码占优势就意味着一种极差的实现方法。使十分简单的事情变得如此困难的一种方法是行不通的。如果每次需要做一个查询时都使用这种方法，我们最终将会拥有大量的代码，并且很容易犯许多错误。像关闭连接（数据检索操作的附带操作）那样的重要操作应该在整个代码中占有重要地位。需要注意的是，关闭连接需要外层try...catch块的finally块内有第二个try...catch块。每当见到一个嵌套的try...catch块时，笔者都有把该代码再加工成一个框架类的强烈欲望。

笔者没有包含获得DataSource的代码，因为这将需要在一个J2EE应用中使用JNDI。笔者假设这已在其他某个地方使用一个合适的助手类得到了实现，以避免JNDI访问进一步增大该代码。

一个简单的抽象框架可以使这个查询变得更简单，同时又仍隐含地使用了JDBC API。下列代码定义了一个可重用、线程安全并且扩展一个框架类的查询对象（下文详细介绍这个框架类）。

```
class AvailabilityQuery extends
    com.interface21.jdbc.object.ManualExtractionSqlQuery {

    public AvailabilityQuery(DataSource ds) {
        super(ds, "SELECT seat_id AS id FROM available_seats WHERE " +
            "performance_id = ? AND price_band_id = ?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    protected Object extract(ResultSet rs, int rounum) throws SQLException {
        return new Integer(rs.getInt("id"));
    }
}
```

这段代码只有第一个版本的三分之一，而且每一行都有某种作用。难对付的错误处理留给了超类。我们只需要编写提供SQL查询、提供赋值参数（在指定了它们的JDBC类型之后）和提取结果的代码。换句话说，我们只需要实现应用功能度，不需要实现探测。框架超类提供许多执行方法，我们可以按如下所示来运行其中的某一方法：

```
AvailabilityQuery availabilityQuery = new AvailabilityQuery(ds);
List l = availabilityQuery.execute(1, 1);
```

一旦被构建好，我们就可以重用这个查询对象。该execute()方法抛出一个运行时异常，而不是一个已检查异常，因而意味着我们只需要捕捉它，如果它是可重获得的。与我们刚才讨论的JDBC Query接口相类似是故意的。JDBC API容易使用，而且已被越来越多的开发人员所熟悉。因此，给JDBC应用相同的原理是有意义的。当然，关键差别是我们的JDBC查询不返回映射后的对象；对返回对象的修改对数据库将毫无影响。

在本节的剩余篇幅中，我们将来看一看如何实现这个抽象框架，以及它怎样才能用来简化应用代码。

目标

还记得Pareto Principle（巴累托定律）（80:20规则）吗？利用JDBC抽象可以实现最佳结果，只是不要试图实现得太多就行。

下列方面将会占用直接使用JDBC的开发人员的大部分精力，并且是最容易出错的。我们的抽象应该设法解决各个方面：

- 太多的代码——例如，查询必须拥有一个Connection、一个Statement和一个ResultSet并循环该ResultSet。
- 万一发生错误时的正确清除（保证连接和语句得到关闭，就像我们上面所讨论的那样）。大多数有能力的Java开发人员在见到了打开连接的数量在服务器端应用中逐渐增加之后，就已经清除了JDBC代码内的许多已坏了的try...catch块。
- 处理SQLExceptions——这些是已检查异常，即应用代码不应该被迫捕捉的异常，因为捕捉它们要求对JDBC有了解。
- 无需查看SQLState和供应商代码，就能让用户代码找出异常的原因变得更容易。

令人惊讶的是，似乎没有太多解决这些问题的库或框架。大多数库或框架都是比较有抱负的（比如，以某种形式的O/R映射为目标），从而意味着它们的实现和使用会变得复杂。因此，笔者为示例应用开发了一个框架（我们马上就谈论该框架），并且它可以用在使用JDBC的任何应用中（实际上，这些类在笔者为各种客户所开发的一系列这种类中是最新的）。

这不是解决使用JDBC的各种挑战的惟一方法，只是这种方法比较简单和有效。

另一个关键问题——把SQL散布在应用中的危险——最好是利用DAO模式来解决，我们已经讨论过这个模式。在本节中，我们将只谈一谈如何实现数据存取对象，不打算谈论它们在整个应用体系结构中的位置。

异常处理

异常处理是一个非常重要的问题，所以在开始适当地实现我们的框架之前，我们应该先为异常处理制定一个一致的策略。

JDBC API是一门讲解如何不使用异常的对象课程。除了保证我们即便遇到错误也不释放资源之外，我们的抽象层还应该为应用代码提供一个比JDBC更好的异常处理模型。

JDBC为除了数据截取之外的所有问题都使用一个统一的异常类——java.lang.SQLException。捕捉一个SQLException实质上并没有提供比“某个东西出了毛病”更多的信息。正如我们已经见过的，只有通过检查SQL异常中可能包含的供应商特有代码，才有可能弄清楚问题。

JDBC实际上可以提供更高级的错误处理，同时又保持数据库之间的可移植性。下面是几个在任一RDBMS中都有意义的错误：

- 使用JDBC发布的SQL语句中的语法错误。
- 对数据完整性约束的违反。
- 试图把一个SQL赋值变量的值设置成一个不正确类型。

这样的标准问题可以——而且应该已被建模成java.lang.SQLException类的各个子类。在我们的抽象框架中，我们将提供一个使用了各个异常类的一个较丰富的异常分级结构。

我们还将解决另外两个与JDBC异常处理有关的问题。

- 我们不希望把使用我们的抽象层的代码束缚于JDBC。我们的抽象层主要是为了实现Data-Access Object (DAO) 模式而设计的。如果使用DAO的代码需要捕捉资源特有的异常，比如SQL异常，业务逻辑与数据存取实现的分离（DAO模式存在的目的就是为了提供这种分离）就会丢失（被抛出的异常是方法签名的一个不可分割部分）。因此，尽管JDBC API（相当充分地）使用了JDBC特有的异常，但我们将不会把自己的异常束缚于JDBC。
- 根据我们在第4章中对已检查异常和运行时异常的讨论，我们将通过使所有异常变成运行时异常而非已检查异常，使我们的JDBC API变得容易使用。JDO采用这种方法具有很好的结果。使用运行时异常，调用者只需要捕捉它们能够从中恢复的那些异常（如果有的话）。这很好地建模了JDBC（和其他数据存取）的使用：JDBC异常通常是不可恢复的。例如，如果一个SQL查询含有一个无效的列名，捕捉它并重试就毫无用处。我们希望致命错误被记录下来，但需要纠正出错的应用代码。
- 如果我们正从一个EJB中运行查询，就可以简单地让EJB容器捕捉运行时异常，并回退当前事务。如果捕捉到了EJB实现中的一个已检查异常，我们则需要自己回退当前事务，或者抛出一个让容器做这件事情的运行时异常。在本例中，使用一个已检查异常给应用代码提供了引起不正确行为的更大余地。同样，无法连接一个数据库可能是致命的。相反，我们可能需要捕捉的一个可恢复错误将是对一个开放式加锁冲突。

一个通用的数据存取异常分级结构

通过创建一个通用的数据存取异常分级结构，我们可以满足自己的所有异常处理目标。这个异常分级结构不局限于和JDBC一起使用，尽管一些JDBC特有异常将派生自它里面的部分普通异常。这个异常分级结构将使利用了DAO的代码能够用一种数据库无关方式处理异常。

所有数据存取异常的根是抽象的DataAccessException类，而这个类扩展了第4章中已讨论过的NestedRuntimeException类。使用NestedRuntimeException超类能使我们保存我们可能需要封装的任何异常（比如SQL异常）的栈跟踪。

和SQLException不同，DataAccessException（数据存取异常）有几个指出具体数据存取问题的子类。这些直接子类是：

- **DataAccessResourceFailureException**

访问所指资源的彻底失败。在一个JDBC实现中，这将由从一个数据源中获得一条连接的失败所引起。不过，这种错误对任一持久性策略来说都是有意义的。

- **CleanupFailureDataAccessException**

成功完成一个操作之后清除工作（比如关闭一个JDBC Connection）的失败。在有些情况下，我们可能需要把这个作为一个可恢复错误来对待，进而防止当前事务被回退（比如，我们知道某一更新已经成功）。

- **DataIntegrityViolationException**

当一个更新由于会破坏数据完整性而遭到数据库拒绝时被抛出。

- **InvalidDataAccessApiUsageException**

这个异常并不暗示底层资源出了问题（比如一个SQL异常），而是暗示数据存取API的不正确使用。下面要讨论的JDBC抽象层在被不正确使用时将会抛出这个异常。

- **InvalidDataAccessResourceUsageException**

这个异常暗示数据存取资源（比如一个RDBMS）被使用得不正确。例如，我们的JDBC抽象层在尝试执行一个无效SQL时将抛出这个异常的一个子类。

- **OptimisticLockingViolationException**

这个异常暗示一个开放式加锁冲突，并在一个竞争更新被检测到时被抛出。这个异常通常将由一个Data Access Object抛出，而不是由像JDBC这样的底层API抛出。

- **DeadlockLoserDataAccessException**

暗示当前操作是一个死锁失败者，进而导致它遭到数据库拒绝。

- **UncategorizedDataAccessException**

不符合上述这些正式类别中的一个异常。这个异常是抽象的，所以资源特有的异常将扩展它来包含附加的信息。我们的JDBC抽象层将在捕捉到其无法归类的一个SQL异常时抛出这个异常。

所有这些概念对任一持久性策略都是有意义的，而不只对JDBC才有意义。普通异常将包含嵌套的根由，比如SQL异常，进而保证没有任何信息会丢失，并保证所有可得到的信息都被记录下来。

图9.1一个UML类图表，举例说明了我们的异常分级结构——被包含在com.interface21.dao包中。请注意com.interface21.jdbc.core包中的一些JDBC特有异常（被显示在水平线以下）是怎样扩展普通异常的，目的是为了让关于一个问题的尽可能多的信息能够包含进来，同时又使调用代码不依赖于JDBC。调用代码通常将捕捉普通异常，尽管JDBC特有的DAO实现能够捕捉它们所理解的JDBC特有子类，而让致命错误传播给使用它们的业务对象。

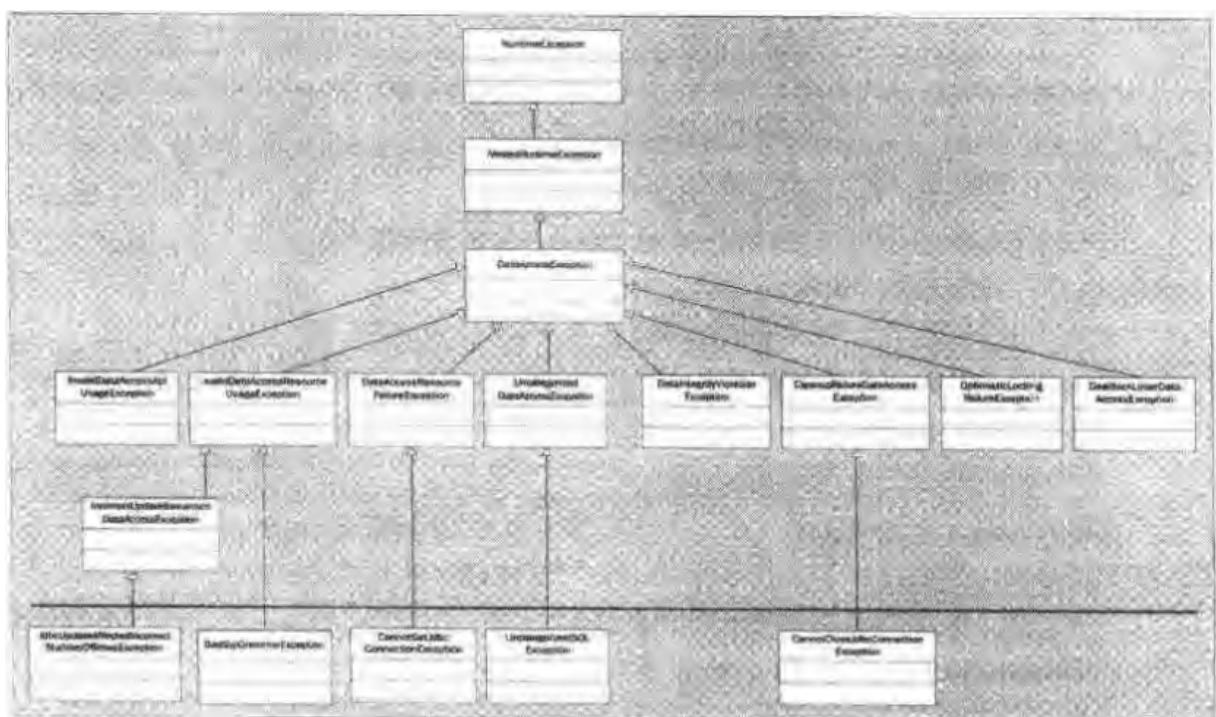


图9.1

虽然这个异常分级结构看起来可能很复杂，但它有助于极大地简化应用代码。所有这些异常类将由我们的框架抛出；应用代码只需要捕捉它认为可恢复的那些异常即可。大多数应用代码将不捕捉这些异常中的任何一个。

这个分级结构自动给我们提供了比一个SQL异常有用得多的信息，而且使需要处理异常的应用代码变得更容易阅读。虽然我们可以选择捕捉基类DataAccessException，如果我们需要知道是否有东西出了问题（用我们捕捉一个SQL异常的相同方法），但在有些情况中只捕捉某个特定子类将会更有用。

下面来看一个似乎合理的需求：执行一个更新操作，并且在该操作万一导致一个数据完整性冲突时，应用一个恢复策略。在一个RDBMS上下文中，该操作可能已试图把一个不可空值列的值设置成null（空值）。使用我们刚才分析过的这个异常分级结构，我们所需要做的只是捕捉com.interface21.dao.DataIntegrityViolationException。所有其他异常都可以被忽略，因为它们是未检查异常。下列代码明确地传达了刚刚描述过的那个恢复需求。作为一个意外收获，它不依赖于JDBC：

```
try {
    // do data operation using JDBC abstraction layer
}
catch (DataIntegrityViolationException ex) {
    // Apply recovery strategy
}
```

现在来看一看如果不使用一个抽象层，而是使用JDBC，我们将需要怎样做这件事情。由于SQL异常是已检查异常，所以我们不能让我们不感兴趣的异常通过。确定我们是否对该异常感兴趣的唯一方法是检查它的SQLState串或供应商代码。在下列例子中，我们检查SQLState串。由于这个串可能是null，所以我们需要在检查它之前先核实它是非空值的。

```
try {
    // do data operation using JDBC
}
catch (SQLException ex) {
    boolean recovered = false;
    String sqlstate = sqlex.getSQLState();
    if (sqlstate != null) {
        String classCode = sqlstate.substring(0, 2);
        if ("23".equals(classCode) ||
            "27".equals(classCode) ||
            "44".equals(classCode)) {
            // Apply recovery strategy
            recovered = true;
        }
    }
    if (!recovered)
        throw new ApplicationSpecificException("Other SQL exception", ex);
}
// Finally block omitted
```

哪个是更简单、更易读、更可维护的方法一目了然。第二种方法不仅复杂得多，而且也使调用代码紧密地依赖于JDBC API的细节。

转换JDBC异常到普通异常

在上面所示的图9.1中，水平线以上的所有类都是通用的。这些都是调用将要处理的异常。

水平线以下的异常都是通用数据存取异常的JDBC特有子类。大多数调用者将不直接使用这些异常（因为这样的使用将会使它们束缚于JDBC），但它们提供更具体的信息（如果需要的话）。例如，`BadSqlGrammarException`扩展通用的`InvalidDataAccessResourceUsageException`，并且在指定SQL是无效时被抛出。

到目前为止，我们还没有考虑过如何在SQL异常与我们的普通异常之间做转换。为了做这种转换，我们将需要检查`SQLState`串或供应商代码。但是，由于`SQLState`串不足以诊断所有问题，所以我们必须保留RDBMS特有实现的选择余地。

为了做转换，我们创建一个接口来定义所需的转换功能度，以便能够在不影响我们的抽象框架的工作方式的情况下使用任何实现（同往常一样，基于接口的设计是实现可移植性的最佳方式）。我们的JDBC抽象层将使用`com.interface21.jdbc.core.SQLExceptionTranslator`接口的一个实现，因为该接口知道如何把JDBC操作期间遇到的SQL异常转换成我们的通用分级结构中的异常：

```
public interface SQLExceptionTranslator {
    DataAccessException translate(String task,
                                  String sql,
                                  SQLException sqlex);
}
```

默认实现是`com.interface21.jdbc.core.SQLStateSQLExceptionTranslator`，它使用`SQLState`代码。这个实现通过构造`SQLState`类代码的一个静态数据结构，避免了对`if/else`语句链的需要。下面是一个部分清单：

```
public class SQLStateSQLExceptionTranslator
    implements SQLExceptionTranslator {

    private static Set BAD_SQL_CODES = new HashSet();
    private static Set INTEGRITY_VIOLATION_CODES = new HashSet();

    static {
        BAD_SQL_CODES.add("42");
        BAD_SQL_CODES.add("65");

        INTEGRITY_VIOLATION_CODES.add("23");
        INTEGRITY_VIOLATION_CODES.add("27");
        INTEGRITY_VIOLATION_CODES.add("44");
    }

    public DataAccessException translate(String task, String sql,
                                         SQLException sqlex) {
        String sqlstate = sqlex.getSQLState();
        if (sqlstate != null) {
            String classCode = sqlstate.substring(0, 2);
            if (BAD_SQL_CODES.contains(classCode))
                throw new BadSqlGrammarException("(" + task +
                    "): SQL grammatical error '" + sql + "'", sql, sqlex);
        }
    }
}
```

```

        if (INTEGRITY_VIOLATION_CODES.contains(classCode))
            throw new DataIntegrityViolationException("(" + task +
                "): data integrity violated by SQL '" + sql + "'", sqlex);
    }

    // We couldn't categorize the problem
    return new UncategorizedSQLException("(" + task +
        "): encountered SQLException [" + sqlex.getMessage() +
        "]", sql, sqlex);
}
}

```

下面的这个部分清单举例说明了一个Oracle特有的实现——com.interface21.jdbc.core.oracle.OracleSQLExceptionTranslator，该实现使用供应商代码来标识同样范围的问题。请注意，供应商代码比SQLState代码有高得多的代价。

```

public class OracleSQLExceptionTranslator
    implements SQLExceptionTranslator {

    public DataAccessException translate(String task, String sql,
        SQLException sqlex) {

        switch (sqlex.getErrorCode()) {
            case 1 :
                // Unique constraint violated
                return new DataIntegrityViolationException(
                    task + ":" + sqlex.getMessage(), sqlex);

            case 1400:
                // Can't insert null into non-nullable column
                return new DataIntegrityViolationException(
                    task + ":" + sqlex.getMessage(), sqlex);

            case 936 :
                // Missing expression
                return new BadSqlGrammarException(task, sql, sqlex);

            case 942 :
                // Table or view does not exist
                return new BadSqlGrammarException(task, sql, sqlex);
        }

        // We couldn't identify the problem more precisely
        return new UncategorizedSQLException("(" + task +
            "): encountered SQLException [" +
            sqlex.getMessage() + "]", sql, sqlex);
    }
}

```

两级抽象

既然我们已经有了~一种强有力的、数据库无关的异常处理方法，现在就让我们来看一看如何实现一个将使JDBC使用起来容易得多的抽象框架。

用在示例应用中的这个JDBC框架包括两级抽象（在第4章中，我们已经讨论过通常怎样把框架分成层，以便开发人员能够根据手头的任务使用适当的层来与适当级别的抽象打交道）。

较低级别的抽象——在com.interface21.jdbc.core包中（我们已经见过该包的使用），负责JDBC工作流程和异常处理。它使用一种回调方法来实现这一抽象，因而要求应用代码实现简单的回调接口。

较高级别的抽象——在com.interface21.jdbc.object包中（我们已经见过该包的使用），依靠这一抽象来提供一个更面向对象的、有JDO特征的、把RDBMS操作建模为Java对象的接口。这对启动这些操作的应用代码彻底隐藏了JDBC的细节。

下面让我们依次来看一看每一级别的抽象。

一个控制JDBC工作流程和错误处理的框架

较低级别的抽象处理JDBC查询和更新的发布，进而负责资源清理，并使用前面讨论过的SQLExceptionTranslater把SQL异常转换成普通异常。

“控制反向”的再讨论

还记得我们在第4章中所做的Strategy设计模式和“控制反向”的讨论吗？我们已经见过，保证复杂的错误处理由普通基础结构代码来隐藏的惟一方法是让基础结构代码调用应用代码（一种叫做“控制反向”的方法），而不是让应用代码把基础结构代码作为一个传统类库来调用。我们还知道，应用这种方法的基础结构包通常叫做框架（framework），而不是叫做类库（class library）。

JDBC错误处理的复杂性要求这种方法。我们需要一个框架类使用JDBC API来执行查询和更新，并处理任何错误，而我们提供SQL和任何参数值。为此，该框架类必须调用我们的代码，而不是我们的代码调用它。

com.interface21.jdbc.core包

构成较低级别抽象的所有JDBC特有类都在com.interface21.jdbc.core包中。

在com.interface21.jdbc.core包中，最重要的类是JdbcTemplate，该类实现核心工作流程，并按照需要调用应用代码。JdbcTemplate类中的方法运行查询，同时把准备语句的创建和从JDBC ResultSets中的结果提取委托给两个回调接口：PreparedStatementCreator接口和RowCallbackHandler接口。应用开发人员将需要提供这两个接口的实现，而不是直接执行JDBC语句或实现JDBC异常处理。

图9.2是一个UML类图，举例说明了JdbcTemplate类与它所使用的助手类和使它能够被参数化的那些应用特有类的关系。

下面让我们依次来看一看每个类和接口。水平线以上的类属于该框架，水平线以下的类指出应用特有类怎样实现框架接口。

PreparedStatementCreator接口及相关类

PreparedStatementCreator接口必须由应用特有的类来实现，以便在给定一个java.sql.Connection的条件下创建一个java.sql.PreparedStatement。这意味着为一个应用特有的查询或更新指定SQL语句并设置赋值变量，而该查询或更新将由JdbcTemplate类来运行。需要注意

的是，这个接口的一个实现不必操心获得一条连接或捕捉可能由它的工作所产生的任何异常。它所创建的PreparedStatement将由JdbcTemplate类来执行。这个接口如下所示：

```
public interface PreparedStatementCreator {
    PreparedStatement createPreparedStatement(Connection conn)
        throws SQLException;
}
```

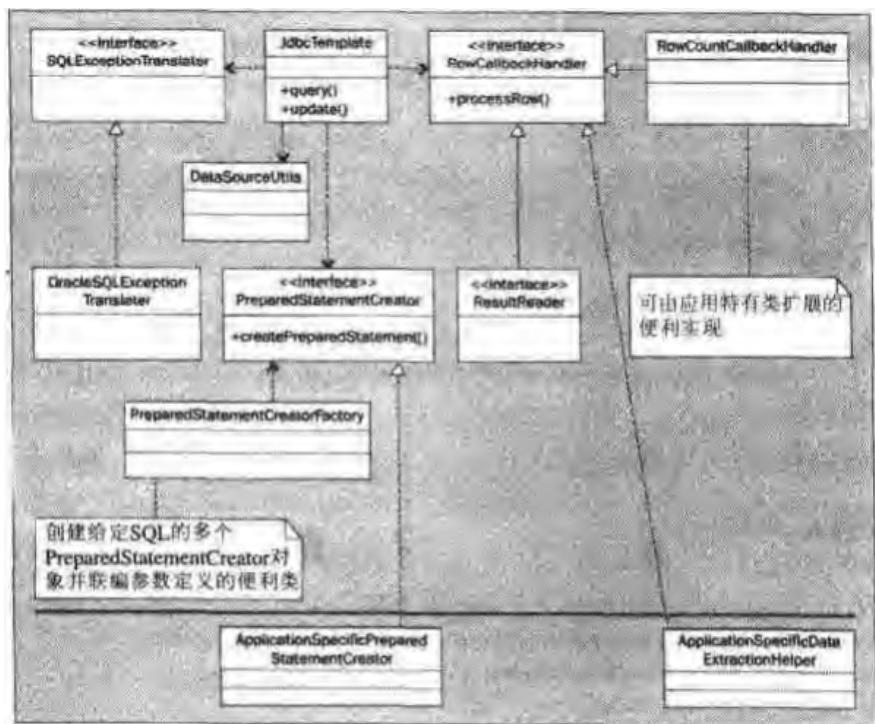


图9.2

下面给出了一个典型的实现，该实现使用标准JDBC方法来构造一条具有指定的SQL的PreparedStatement并设置联编参数值。和此类简单接口的许多实现一样，这是一个匿名的内部类：

```
PreparedStatementCreator psc = new PreparedStatementCreator() {
    public PreparedStatement createPreparedStatement(Connection conn)
        throws SQLException {
        PreparedStatement ps = conn.prepareStatement(
            "SELECT seat_id AS id FROM available_seats WHERE " +
            "performance_id = ? AND price_band_id = ?");
        ps.setInt(1, performanceId);
        ps.setInt(2, seatType);
        return ps;
    }
};
```

`PreparedStatementCreatorFactory`类是一个能够被重复使用以根据相同的语句和赋值变量值创建具有不同参数值的`PreparedStatementCreator`对象的通用助手类。这个类主要由下面将描述的较高级的对象抽象框架所使用；应用代码通常将像上面所示的那样定义`PreparedStatementCreator`类。

RowCallbackHandler接口及相关类

RowCallbackHandler接口必须由应用特有的类来实现，以便从一个查询所返回的结果集的每一行中提取列值。**JdbcTemplate**类处理**ResultSet**上的循环。这个接口的实现也可能会放任SQL异常不去捕捉：**JdbcTemplate**将会处理它们。这个接口如下所示：

```
public interface RowCallbackHandler {
    void processRow(ResultSet rs) throws SQLException;
}
```

实现应该知道**ResultSet**中预计的列和数据类型的数量。下面给出了一个典型实现，该实现从每一行中提取一个int值，并把它添加到那个内层方法中所定义的一个列表上：

```
RowCallbackHandler rch = new RowCallbackHandler() {
    public void processRow(ResultSet rs) throws SQLException {
        int seatId = rs.getInt(1);
        list.add(new Integer(seatId));
    }
};
```

RowCountCallbackHandler类是这个接口的一个便利框架实现，并且该实现存储有关列名和类型的元数据，并计算结果集中的行数。虽然它是一个具体类，但它的主要用处是作为应用特有类的一个超类。

ResultReader接口扩展**RowCountCallbackHandler**类来把检索出的结果保存在一个**java.util.List**中：

```
public interface ResultReader extends RowCallbackHandler {
    List getResults();
}
```

ResultSet的每一行到一个对象的转换可能会在实现之间有所不同，因此根本不存在这个接口的标准实现。

其他类

我们已经分析了**SQLExceptionTranslater**接口和两个实现。**JdbcTemplate**类使用一个**SQLExceptionTranslater**类型的对象把SQL异常转换成我们的普通异常分级结构，以保证**JdbcTemplate**类的行为的这一重要部分能够被参数化。

DataSourceUtils类含有一个从某个**javax.sql.DataSource**中获得一条**Connection**的静态方法，进而把任一SQL异常转换到来自我们的普通异常分级结构中的一个异常，而且含有一个关闭一条**Connection**的方法（同样具有适当的异常处理）和一个从JNDI中获得一个**DataSource**的方法。**JdbcTemplate**类使用**DataSourceUtils**类来简化连接的获得和关闭。

JdbcTemplateClass: 核心工作流程

现在，让我们来看一看**JdbcTemplate**类本身的实现。这是运行JDBC语句并捕捉SQL异常的惟一类。它含有能够做任何查询或更新的方法：被执行的JDBC语句由静态SQL或一个被作为一个方法变元来提供的**PreparedStatementCreator**实现来参数化。

下面是**JdbcTemplate**类的一个完整清单：

```

package com.interface21.jdbc.core;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.Statement;
import java14.java.util.logging.Level;
import java14.java.util.logging.Logger;

import javax.sql.DataSource;
import com.interface21.dao.DataAccessException;

public class JdbcTemplate {

```

实例数据由一个Logger对象（用来记录关于SQL操作的信息消息）、一个DataSource对象、一个指出SQL警告应该被忽视还是被看做错误的boolean和SQLExceptionTranslater助手类所组成。所有这些实例变量在JdbcTemplate类得到配置之后都是只读的，因而意味着JdbcTemplate对象都是线程安全的：

```

protected final Logger logger = Logger.getLogger(getClass().getName());
private DataSource dataSource;
private boolean ignoreWarnings = true;
private SQLExceptionTranslater exceptionTranslater;

```

构造器接受一个DataSource，这个数据源将用在JdbcTemplate的整个生存周期内，并实例化一个默认的SQLExceptionTranslater对象：

```

public JdbcTemplate(DataSource dataSource) {
    this.dataSource = dataSource;
    this.exceptionTranslater = new SQLStateSQLExceptionTranslater();
}

```

在使用JdbcTemplate之前，下列这些方法可以任选地用来修改默认配置。它们能使警告和异常转换的行为被参数化：

```

public void setIgnoreWarnings(boolean ignoreWarnings) {
    this.ignoreWarnings = ignoreWarnings;
}

public boolean getIgnoreWarnings() {
    return ignoreWarnings;
}

public void setExceptionTranslater(
    SQLExceptionTranslater exceptionTranslater) {
    this.exceptionTranslater = exceptionTranslater;
}

public DataSource getDataSource() {
    return dataSource;
}

```

JdbcTemplate类的剩余部分由使用上述这些回调方法来执行JDBC工作流程的方法所组成。

最简单的query()方法（接受一个静态SQL串和一个从查询ResultSet中提取结果的Row-

`CallbackHandler`) 举例说明了回调的使用怎样把控制流程和错误处理集中在`JdbcTemplate`类中。需要注意的是，这个方法抛出我们的普通`com.interface21.dao.DataAccessException`，以便使调用者不必使用JDBC就能查找任一错误的根源。

```

public void query(String sql, RowCallbackHandler callbackHandler)
    throws DataAccessException {

    Connection con = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        con = DataSourceUtils.getConnection(this.dataSource);
        ps = con.prepareStatement(sql);
        rs = ps.executeQuery();
        if (logger.isLoggable(Level.INFO))
            logger.info("Executing static SQL query '" + sql + "'");

        while (rs.next()) {
            callbackHandler.processRow(rs);
        }

        SQLWarning warning = ps.getWarnings();
        rs.close();
        ps.close();

        throwExceptionOnWarningIfNotIgnoringWarnings(warning);
    }
    catch (SQLException ex) {
        throw this.exceptionTranslator.translate("JdbcTemplate.query(sql)",
            sql, ex);
    }
    finally {
        DataSourceUtils.closeConnectionIfNecessary(this.dataSource, con);
    }
} // query

```

`throwExceptionOnWarningIfNotIgnoringWarnings()`方法是一个用在几个公共工作流程方法中的私用助手。如果传递给它的该警告是非空值，并且`JdbcTemplate`类被配置成不忽略异常，它就抛出一个异常。如果遇到一个警告，但`JdbcTemplate`类被配置成忽略警告，它就记录该警告：

```

private void throwExceptionOnWarningIfNotIgnoringWarnings(
    SQLWarning warning) throws SQLWarningException {

    if (warning != null) {
        if (this.ignoreWarnings) {
            logger.warning("SQLWarning ignored: " + warning);
        } else {
            throw new SQLWarningException("Warning not ignored", warning);
        }
    }
}

```

一个更普通的`query()`方法发布一条`PreparedStatement`，该语句必须由一个`PreparedStatementCreator`实现来创建。笔者已经突出显示了与上述`query()`方法的不同之处：

```

public void query(PreparedStatementCreator psc,
    RowCallbackHandler callbackHandler) throws DataAccessException {
    Connection con = null;
    Statement s = null;
    ResultSet rs = null;
    try {
        con = DataSourceUtils.getConnection(this.dataSource);
        PreparedStatement ps = psc.createPreparedStatement(con);
        if (logger.isLoggable(Level.INFO))
            logger.info("Executing SQL query using PreparedStatement: [" +
                + psc + "]");
        rs = ps.executeQuery();

        while (rs.next()) {
            if (logger.isLoggable(Level.FINEST))
                logger.finest("Processing row of ResultSet");
            callbackHandler.processRow(rs);
        }

        SQLWarning warning = ps.getWarnings();
        rs.close();
        ps.close();
        throwExceptionOnWarningIfNotIgnoringWarnings(warning);
    }
    catch (SQLException ex) {
        throw this.exceptionTranslator.translate(
            "JdbcTemplate.query(psc) with PreparedStatementCreator [" +
            psc + "]", null, ex);
    }
    finally {
        DataSourceUtils.closeConnectionIfNecessary(this.dataSource, con);
    }
}

```

我们给更新应用相同的手段。下列这个方法允许使用--条统一的JDBC Connection执行更新。一个PreparedStatementCreator对象数组提供要使用的SQL和赋值参数。这个方法返回一个含有每个更新所影响到的行数的数组：

```

public int[] update(PreparedStatementCreator[] pscs)
    throws DataAccessException {

    Connection con = null;
    Statement s = null;
    int index = 0;
    try {
        con = DataSourceUtils.getConnection(this.dataSource);
        int[] retvals = new int[psc.length];
        for (index = 0; index < retvals.length; index++) {
            PreparedStatement ps = pscs[index].createPreparedStatement(con);
            retvals[index] = ps.executeUpdate();
            if (logger.isLoggable(Level.INFO))
                logger.info("JDBCTemplate: update affected " + retvals[index] +
                    " rows");
            ps.close();
        }
        return retvals;
    }

```

```

        catch (SQLException ex) {
            throw this.exceptionTranslator.translate("processing update " +
                (index + 1) + " of " + pscs.length + "; update was [" +
                pscs[index] + "]");
        }
    } finally {
        DataSourceUtils.closeConnectionIfNecessary(this.dataSource, con);
    }
}

```

下列这些便利方法使用上述方法来执行一个单独的更新，如果给定静态SQL或一个单独的PreparedStatementCreator参数：

```

public int update(final String sql) throws DataAccessException {
    if (logger.isLoggable(Level.INFO))
        logger.info("Running SQL update '" + sql + "'");
    return update(PreparedStatementCreatorFactory.
        getPreparedStatementCreator(sql));
}

public int update(PreparedStatementCreator psc)
    throws DataAccessException {
    return update(new PreparedStatementCreator[] { psc })[0];
}
}

```

需要注意的是，`JdbcTemplate`对象维护一个`javax.sql.DataSource`实例变量，该对象将从这个实例变量中获得用于每个操作的连接。重要的是，API应该与`DataSource`对象打交道，而不是与`java.sql.Connection`对象打交道，因为：

- 我们必须从别处获得那些连接，因而意味着应用代码有更大的复杂性，并需要捕捉SQL异常（如果使用`DataSource.getConnection()`方法）。
- `JdbcTemplate`关闭它处理的连接是很重要的，因为关闭一条连接会导致一个异常，并且我们需要自己的框架负责所有JDBC异常处理。从别处获得那些连接并在`JdbcTemplate`类中关闭它们将没有意义，而且将会引入企图使用已关闭连接的危险。

使用一个`DataSource`并不限制`JdbcTemplate`类只在一个J2EE容器中使用：`DataSource`接口是相当简单的，因而为了测试目的或在独立使用的应用中实现它是相当容易的。

使用`JdbcTemplate`类

在了解了`JdbcTemplate`类的实现之后，现在该来看一看运转中的`com.interface21.jdbc.core`包。

执行查询

使用`JdbcTemplate`类，我们可以按照如下所示来完成座位ID的JDBC查询（参见前面的“动机”一节）：使用匿名的内部类来实现`RowCallbackHandler`和`PreparedStatementCreator`接口。`RowCallbackHandler`的这个实现把数据保存在那个内层方法中所定义的一个`Integers`型列表中：

```

public List getAvailableSeatIdsWithJdbcTemplate(
    DataSource ds, final int performanceId, final int seatType)
    throws DataAccessException {
    JdbcTemplate t = new JdbcTemplate(ds);
    final List l = new LinkedList();

    PreparedStatementCreator psc = new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
            PreparedStatement ps = conn.prepareStatement(
                "SELECT seat_id AS id FROM available_seats WHERE " +
                "performance_id = ? AND price_band_id = ?");
            ps.setInt(1, performanceId);
            ps.setInt(2, seatType);
            return ps;
        }
    };
    RowCallbackHandler rch = new RowCallbackHandler() {
        public void processRow(ResultSet rs) throws SQLException {
            int seatId = rs.getInt(1);
            l.add(new Integer(seatId));
        }
    };
    t.query(psc, rch);
    return l;
}

```

这把所需代码减少了一半，并且解决了我们在直接使用JDBC API时所遇到的大多数问题。我们可以使用最少量的不相干代码来处理PreparedStatement和ResultSet。最重要的是，使用JdbcTemplate类消除了主要的错误根源。不再存在连接将得不到关闭的危险：JdbcTemplate类保证这一点。应用代码不必捕捉不提供信息的已检查SQL异常；我们的普通数据存取异常分级结构提供了一个较丰富的异常分类，但是，由于数据存取异常是未检查的，所以应用代码一般可以不管要由应用服务器来处理的异常。

执行更新

如果我们直接使用JDBC API，更新需要使用与查询类似的代码量。同样，错误处理占优势地位，而我们需要执行的SQL处于劣势地位。同样，JdbcTemplate类可以提供真正的好处。JdbcTemplate update()方法能够使用我们已经见过的PreparedStatementCreator回调接口来运行更新。下列例子（不是示例应用的一部分）将删除某一给定演出的一种特定座位类型的所有座位预定与订票。本例假设我们已经有一个范围内的JdbcTemplate对象。由于JdbcTemplate对象是线程安全的，所以我们通常将一个JdbcTemplate对象保持为使用了JDBC的DAO实现中的一个实例变量。

```

class PerformanceCleanerPSC implements PreparedStatementCreator {
    private int pid;
    private int pb;

    public PerformanceCleanerPSC(int pid, int pb) {
        this.pid = pid;
        this.pb = pb;
    }
}

```

```

public PreparedStatement createPreparedStatement(Connection conn)
    throws SQLException {
    PreparedStatement ps = conn.prepareStatement("UPDATE seat_status " +
        "SET booking_id = null WHERE performance_id = ? AND " +
        "price_band_id = ?");
    ps.setInt(1, pid);
    ps.setInt(2, pb);
    return ps;
}
};

PreparedStatementCreator psc = new PerformanceCleanerPSC (1, 1);
int rowsAffected = jdbcTemplate.update(psc);

```

使用静态SQL的更新更容易。下列例子将把售票应用中的所有座位标记为空座位，同时我们又不必实现回调接口：

```
template.update('UPDATE SEAT_STATUS SET BOOKING_ID <- NULL');
```

较高级别抽象：把RDBMS操作建模为Java对象

使用JdbcTemplate类解决了我们已经见过的、与原始JDBC的使用有关的大部分问题，但可以证明的是，这种方法仍太低级。使用了JdbcTemplate类的应用代码仍需要JDBC Statement和ResultSet的知识。回调接口的使用（尽管对转移工作流程到框架内是必不可少的）从概念上讲是复杂的（尽管回调接口的实现通常是非常简单的）。

com.interface21.jdbc.core包解决了使用JDBC时的各种最复杂的问题，但我们希望给应用代码提供更简单的解决方案。我们需要一个以com.interface21.jdbc.core包所提供的功能度为基础的较高级别抽象。

com.interface21.jdbc.object包的实现

com.interface21.jdbc.object包隐藏了com.interface21.jdbc.core包（它被建立在这个包上），从而提供了一个较高级别的、有JDO特征的对象抽象。应用代码不必实现回调方法，而是每个查询或其他RDBMS操作被建模为一个可重用的、线程安全的对象。一旦得到配置，每个对象就可以用不同的参数被重复地运行，其中参数被映射到赋值变量上，并在每次运行时被提供。

与JDO之间的一个重要差别是：因为我们的RDBMS操作对象是类，不是接口，所以让应用查询子类化它们是可能的（使用JDO的应用代码必须从一个PersistenceManager中获得Query接口的实例）。通过为每个RDBMS操作使用各自的子类，我们完全可以向调用代码隐藏掉SQL。表示RDBMS操作的对象可以实现不束缚于关系数据库的接口。

这种方法可以一致地用于查询、更新和存储过程。图9.3是一个UML类图，举例说明了框架类之间的继承性分级结构，以及一个虚构的ApplicationSpecificQuery类在它里面的位置。它还描述了这些类怎样和上面讨论的那些低级JDBC抽象类相关联。

现在让我们来详细地看一看这个继承性分级结构中的每一个类。由于涉及到的类数量太大，所以在此提供一个完整的清单是不可能的。在阅读下面的描述时，请根据需要参考配书下载的/framework/src目录中的com.interface21.jdbc.object包。

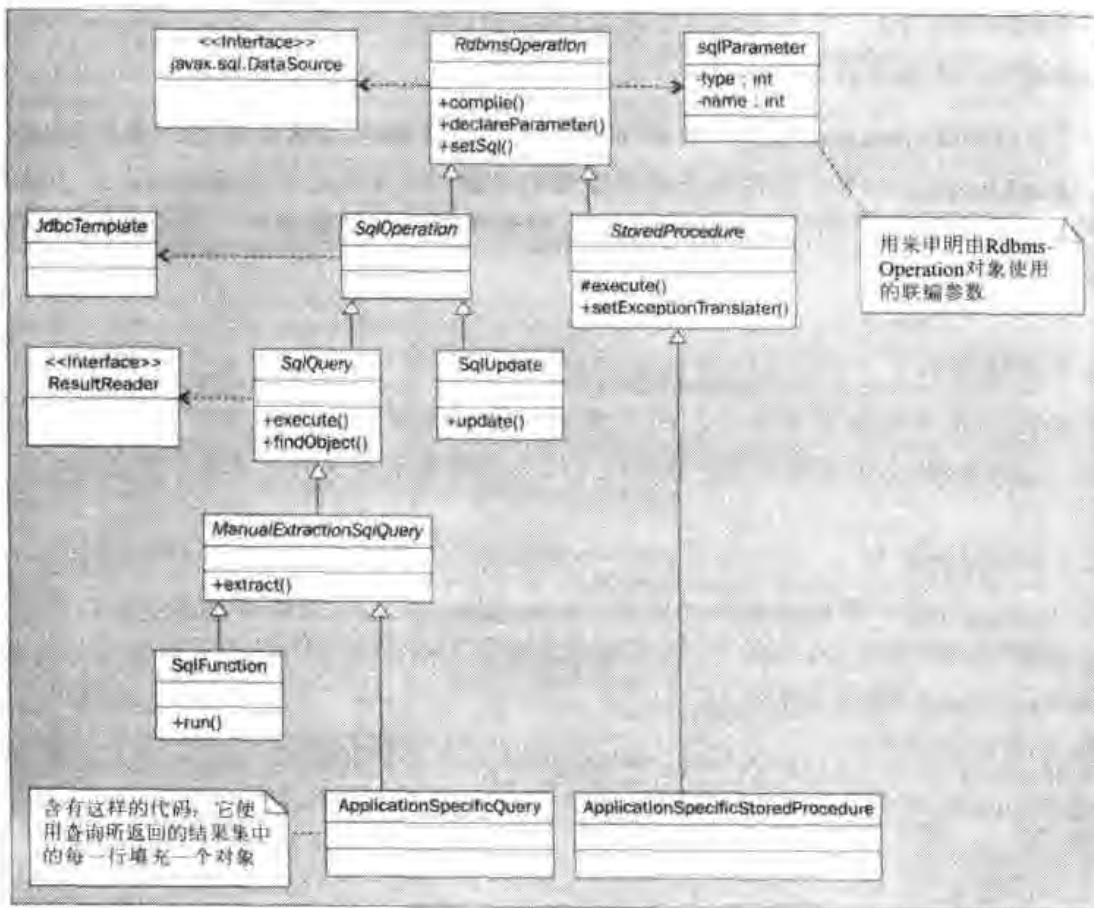


图9.3

RdbmsOperation基类

RdbmsOperation是一个抽象类，并且是这个类分级结构的根。它含有作为实例变量的一个javax.sql.DataSource和一个SQL串，并允许赋值变量被声明（查询、更新和存储过程共享赋值变量的概念）。一旦得到了配置，一个RdbmsOperation就必须被“编译”——从JDO Query接口那里借用的另一个构思。编译的含意将在子类之间有所不同，但一个DataSource和SQL已被供给将至少是有效的。当编译之后，就不再能添加参数，但该操作可以被重复地执行。

RDBMS操作就是Java组件（JavaBean）。要执行的SQL和要使用的DataSource被暴露为组件属性（bean property），而子类通常将把它们自己的配置属性暴露为组件属性。

参数使用declareParameter (SQLParameter)方法来声明。SQLParameter类被定义在com.interface21.jdbc.core包中，但主要由这个包使用。参数声明的主要目的是指定每个参数的JDBC类型，就像java.sql.Types类中所枚举的那样。这保证了框架代码能够使用JDBC PreparedStatement set<Type>()方法而不是setObject()方法来设置赋值变量，而赋值变量对保证null值的正确处理是必不可少的，并且可能会更有效。传递给查询和更新的参数不需要名称（只需要正确的顺序和类型）；存储过程除了支持输入参数之外，还支持输出参数，于是需要参数名。

虽然RdbmsOperation类为查询、更新和存储过程确保了一致的概念，但它不知道怎样执行数据库操作。子类必须添加方法来执行这些操作。

一个RdbmsOperation就是一个表示SQL查询、更新或存储过程的可重用Java对象。RdbmsOperations可能有对应于赋值变量的参数（在使用它们之前声明）。一旦得到了配置和“编译”，一个RdbmsOperation对象就可以被重复执行，每次执行时使用不同的参数。

这里所描述的类分层结构是正确使用具体继承性的一个上佳示例。在这种情况下，我们除了想使多态性成为可能，还希望继承实例变量和实现。由于参数化主要由com.interface21.jdbc.core包来处理，所以没有必要使用基于接口的设计来考虑更大的灵活性。

SqlOperation类

SqlOperation是一个抽象类，并且扩展RdbmsOperation类来用做基于SQL的查询和更新的一个超类（与存储过程相对）。它的编译实现核查SQL语句中赋值变量的预计个数（即？字符的个数）匹配于已声明的SqlParameters的个数，并配置一个能够为配置时所指定的SQL和参数有效地创建一个PreparedStatementCreator对象的PreparedStatementCreatorFactory。SqlOperation类创建一个其子类可以用来执行数据库操作的JdbcTemplate。

SqlQuery类

这是所有查询对象的超类，并使用从SqlOperation中继承来的JdbcTemplate实例变量来执行查询（给定SQL和赋值变量）。

SqlQuery类是抽象的，因而使用Template Method设计模式把结果的提取推迟到子类中，而这些子类是实现下列抽象保护方法所必需的：

```
protected abstract ResultReader newResultReader(
    int rowsExpected, Object[] parameters);
```

我们已经在前面讨论过ResultReader接口：这个接口被定义在com.interface21.jdbc.core包中，并把来自RowCallbackHandler回调的结果保存在一个对象列表中，其中每个对象代表来自一个行的数据。

和JDO Query接口一样，SqlQuery类提供了带有不同变元的许多execute()便利方法，比如execute(int)和execute(String)。和JDO Query中一样，execute(Object[])方法（由所有其他execute()方法调用）接受一个对象数组作为一个参数。传递给execute()方法的那些参数代表SqlQuery对象中所声明的那些赋值变量的动态值。所有execute()方法都返回一个结果列表——由SqlQuery的ResultReader实现所产生。

SqlQuery类还暴露许多findObject()便利方法。这些方法类似于单对象实体组件发现者方法，比如findByPrimaryKey()；如果有一个以上的对象相匹配，它们会引起一个错误。

子类可以依靠继承来的execute()或findObject()方法，也可以实现它们自己的具有有意义名称的查询方法。例如，一个需要4个参数的子类方法可能接受一个对象变元，并隐藏掉调用普通超类方法的工作：

```

public List purchasesByAddress(Address a) {
    return execute(new Object[] {
        a.getStreet(), a.getLine2(), a.getCity(), a.getPostCode()
    });
}

```

一个返回了单个Seat对象的子类发现者方法可能按如下所示来实现，进而向使用它的调用代码隐藏掉必需的强制类型转换：

```

public Seat findSeat(int seatId) {
    return (Seat) super.findObject(seatId);
}

```

一个子类方法还可以把由一个继承的execute()方法所返回的一个列表（List）转换成一个类型化的数组，因为该子类将会知道什么应用对象表示查询结果的每一行。

ManualExtractionSqlQuery类

应用查询通常不直接扩展SqlQuery类，而是子类化SqlQuery类的ManualExtractionSqlQuery抽象子类。ManualExtractionSqlQuery类使用Template Method设计模式来迫使子类实现下面这个抽象方法，而这个方法将针对当前查询所产生的结果集的每一行被调用一次。不过，让子类实现这个方法比实现SqlQuery的新ResultReader()方法简单得多。

```

protected abstract Object extract(ResultSet rs, int rownum)
    throws SQLException;

```

该实现看起来像我们在上面已见过的RowCallbackHandler接口的实现。ManualExtractionSqlQuery类负责从每个返回对象中构造一个List，Extract()方法的下列实现将从一个被返回的结果集的每一行中创建一个虚构的、应用特有的Customer对象：

```

protected Object extract(ResultSet rs, int rownum) throws SQLException {
    Customer cust = new Customer();
    cust.setForename(rs.getString("forename"));
    cust.setId(rs.getInt("id"));
    return cust;
}

```

需要注意的是，RdbmsOperation的子类不必捕捉SQL异常。被抛出的任何SQL异常将由JdbcTemplate类来处理（参见上文），进而使一个普通的数据存取异常被抛出。还需要注意的是，子类代码可以基于单个列值轻松地设置多个属性值。我们将在谈论如何使用这个框架时，看一看ManualExtractionSqlQuery类的几个完整子类。

读者可能正觉得奇怪，笔者为什么不采取主动来摆脱从结果集的每一行中提取一个对象的“手工提取”方法。实现一个ReflectionExtractionSqlQuery并让它使用反射从每个结果集行的列值中创建一个JavaBean和设置组件属性是可能的。这种方法是非常吸引人的，因为它进一步减少了必需的应用代码量。

笔者平常热衷于使用反射，并且已经几次实现了这种方法。不过，笔者不敢确信这种方法会增加真正的价值。这种方法有下列缺点：

- 这种方法明显使框架变得更复杂。
- 这种方法只是令人怀疑地转移了复杂性，而不是消除了复杂性。例如，使用映射控制从RDBMS列值到JavaBean属性的转换将是必不可少的。即便映射被保存在Java代码外部，它们仍将需要被创建及维护。
- 这种方法没有兼顾到计算属性，比如以几个列的值为基础的属性，否则就要在框架中引入极大的复杂性。
- 这种方法不允许把不同的对象类型用于不同的行（例如，如果不时地表示需要一个子类）。

这一抉择说明了巴累托定律的作用。仅仅节省了少数几行从单个结果集行中提取值的代码（它们不再非常复杂，因为没有必要捕捉SQL异常）并没有增加足以证明引入该复杂性有道理的价值。

SqlFunction类

SQL函数可以被看做SQL查询的特例，它只返回一行。因此，我们可以轻松地应用相同的方法。`com.interface21.jdbc.object.SqlFunction`类是一个简单的具体类，它扩展`ManualExtractionSqlQuery`来使其结果可被保存在一个Java int中的查询能够简单地通过提供SQL和声明参数被运行。一个`SqlFunction`对象可以按照如下所示被构造：

```
SqlFunction freeSeatsFunction = new SqlFunction(dataSource,
    "SELECT count(seat_id) FROM available_seats WHERE performance_id = ?");
freeSeatsFunction.declareParameter(new SqlParameter(Types.NUMERIC));
freeSeatsFunction.compile();
```

然后，`SqlFunction`对象就可以像下面这样使用：

```
freeSeatsFunction.run(performanceId);
```

和`SqlQuery`类一样，`SqlFunction`提供了许多带有不同变元的`run()`便利方法，以及一个接受一个`Object`型数组的普通形式。

SqlUpdate类

更新与查询共用了许多概念，比如SQL、赋值变量声明，而且使用一个`JdbcTemplate`类来帮助实现。因此，`SqlUpdate`类扩展了`SqlOperation`类，进而继承了`JdbcTemplate`助手以及核查赋值变量声明与供给的SQL相一致的验证逻辑。

`SqlUpdate`类是具体类，因为不存在要提取的结果，并且没有必要让子类实现自定义提取。它提供了许多`update()`方法，其中每个方法都返回受到当前更新影响的行数。和查询方法一样，所有`update()`方法都调用一个接受一个`Object`参数值数组的普通更新方法：

```
public int update(Object[] args)
    throws InvalidDataAccessApiUsageException
```

StoreProcedure类

我们的建模除了支持普通SQL查询和更新，自然也支持存储过程。**StoreProcedure**抽象类扩展**RdbmsOperation**类。因为它不需要一个**JdbcTemplate**助手，而且由于供给的SQL只是该存储过程的名称，所以对照赋值变量声明来验证它是不可能的（只有当该存储过程在运行期间被调用时，不正确的参数个数才将会引起一个故障）。

直接使用JDBC调用一个存储过程涉及到创建**java.sql.CallableStatement**接口的一个对象，以及提供一个调用串（call string）。调用串包含和用于JDBC准备语句的占位符相类似的占位符，而且看起来像下面的例子：

```
{call reserve_seats(?, ?, ?, ?)}
```

一旦创建了一个**CallableStatement**对象，调用存储过程就要求与查询和更新相类似的错误处理。存储过程可以返回**ResultSet**，但我们多半使用输出参数（output parameter）（让存储过程返回结果集的机制在RDBMS之间是不同的。它在Oracle中是十分复杂的）。

StoreProcedure类必须由应用特有的类来子类化。每个子类实际上都成为该存储过程的一个Java代理。我们已经见过，存储过程与查询和更新的唯一差别是：一个存储过程除了可以有输入参数之外，还可以有输入/输出参数。**StoreProcedure**类自动建立调用串，并隐藏一个**CallableStatement**的使用和必需的错误处理。输入参数被供给，输出参数在**java.util.Map**对象中被返回。

使用JDBC对象抽象

在学习了这个基于对象的JDBC抽象的实现之后，下面让我们来看一看如何使用它完成常见的任务。

执行查询

JDBC在执行查询时效率非常高，尽管我们不能透明地更新由JDBC查询所创建的对象。但是，我们能够执行带有许多参数并包含无法使用O/R映射来轻易完成的复杂连接的查询。JDBC查询产生的对象是理想的值对象，因为它们与数据库断开了，并且不具有依赖于持久性API的相关性。

下列3个代码例子举例说明了可以和**RdbmsOperation**和子类一起使用的3个惯用法。

应用特有查询通常将子类化**ManualExtractionSqlQuery**。我们至少将需要实现抽象的**extract()**受保护方法。下面匿名的内部类就实现这个方法。需要注意的是，该查询使用从**RdbmsOperation**中继承来的那些配置方法进行配置：

```
SqlQuery customerQuery = new ManualExtractionSqlQuery() {
    protected Object extract(ResultSet rs, int rownum) throws SQLException {
        Customer cust = new Customer();
        cust.setForename(rs.getString("forename"));
        cust.setId(rs.getInt("id"));
        return cust;
    }
};
customerQuery.setDataSource(ds);
```

```

customerQuery.setSql("SELECT id AS id, forename AS forename FROM " +
    "customer WHERE id=?");
customerQuery.declareParameter(new SqlParameter(Types.NUMERIC));
customerQuery.compile();

```

这个查询可以使用来自SqlQuery的那些execute()便利方法之一来执行，如下所示：

```
List l = customerQuery.execute(1);
```

下列查询实现稍微复杂一点。它把SQL、参数声明和编译过程隐藏到了构造器内，进而提供了较好的封装。extract()方法的实现和我们的第一个例子相同：

```

private class CustomerQuery extends ManualExtractionSqlQuery {

    public CustomerQuery (DataSource ds) {
        super(ds, "SELECT forename, id FROM customer WHERE id=?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    protected Object extract(ResultSet rs, int rownum) throws SQLException {
        Customer cust = new Customer();
        cust.setForename(rs.getString("forename"));
        cust.setId(rs.getInt("id"));
        return cust;
    }
}

```

创建这种类型的对象更简单，但我们必须仍依赖于继承来的execute()方法。这个查询可以按照如下所示来使用：

```

SqlQuery customerQuery = new CustomerQuery(dataSource);
List customers = customerQuery.execute(6);

```

由于RdbmsOperation对象是线程安全的，所以我们一般将构造这样的对象一次，并把它们作为实例变量保存在Data Access Objects中。

下面的版本是一个更复杂的查询，此查询不仅把SQL、参数声明和编译过程隐藏在它的构造器内，而且还实现一个新的查询方法；这个新的查询方法能够使该查询接受一种参数组合，而SqlQuery类中根本没有execute()便利方法能够接受这样一种参数组合。

```

class CustomerQuery extends ManualExtractionSqlQuery {

    public CustomerQuery(DataSource ds) {
        super(ds, "SELECT id AS id, forename AS forename FROM customer " +
            "WHERE mystring=? AND myint=? AND string3=?");
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }

    protected Object extract(ResultSet rs, int rownum) throws SQLException {
        Customer cust = new Customer();
        cust.setForename(rs.getString("forename"));
        cust.setId(rs.getInt("id"));
        return cust;
    }
}

```

这个新的查询方法能够接受被严格类型化的参数，并且可以被赋予一个有意义的名字：

```
public List findWithMeaningfulName(
    String myString, int id, String string3) {
    return execute(new Object[] {
        myString, new Integer(id), string3 });
}
```

我们可以按照如下所示来使用这个方法。需要注意的是，下列这个代码是不言自明的：

```
CustomerQuery customerQuery = new CustomerQuery(ds);
List l = customerQuery.findWithMeaningfulName("foo", 1, "bar");
```

实际上，通过在查询对象之间使用继承性，避免代码重复是很容易的。例如，extract()方法的实现可以在一个基类中得到提供，而子类提供不同的SQL和变量声明。当仅有SQL WHERE子句不同时，同一个类的多个查询对象可以被创建，其中每个查询对象被配置成执行不同的SQL。

执行更新

使用一种基于SQL的JDBC方法，我们无法拥有真正的O/R映射，真正的O/R映射一般会在对象属性发生改变时导致透明的更新。不过，我们可以在O/R映射不适用的地方完成有效的更新，比如影响多行的更新和使用存储过程的更新。

下列更新对象执行和上面所显示的JdbcTemplate更新相同的更新，但把所用到的SQL和参数声明隐藏在了它的构造器内：

```
class PerformanceCleaner extends com.interface21.jdbc.object.SqlUpdate {

    public PerformanceCleaner(DataSource dataSource) {
        setSql("UPDATE seat_status SET booking_id = null " +
               "WHERE performance_id = ? AND price_band_id = ?");
        setDataSource(dataSource);
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    public int clearBookings(int performanceId, int type) {
        return update(new Object[] {
            new Integer(performanceId), new Integer(type) });
    }
}
```

clearBookings()方法调用超类update(Object[])方法，以便用给定参数进行执行，进而为调用者简化了API——我们已在前面见过的、用于查询execute()方法的同一种方式。

这个更新对象可以按照如下所示来使用：

```
PerformanceCleaner pc = new PerformanceCleaner(dataSource);
pc.clearBookings(1, 1);
```

调用存储过程

由于本书示例应用中的存储过程调用相当复杂，并且使用专有的Oracle特性，所以让我们先来看一个调用存储过程的例子，这个例子不是示例应用的一部分，并且具有两个数字输入参数和一个输出参数。

构造器与我们已经见过的那些查询和更新的构造器十分相似，因而声明参数和调用compile()方法。需要注意的是，那个SQL串是该存储过程的名称：

```
class AddInvoice extends com.interface21.jdbc.object.StoredProcedure {
    public AddInvoice(DataSource ds) {
        setDataSource(ds);
        setSql("add_invoice");
        declareParameter(new SqlParameter("amount", Types.INTEGER));
        declareParameter(new SqlParameter("custid", Types.INTEGER));
        declareParameter(new OutputParameter("newid", Types.INTEGER));
        compile();
    }
}
```

我们必须实现一个方法来执行该存储过程（虽然笔者已经使用了名字execute()，但我们可以给这个方法赋予任何一个名字）。突出显示的这一行是一个对StoredProcedu类的execute()受保护方法的调用。我们使用了从那个新方法的变元中所构造出的一个输入参数映射(Map)来调用这个方法。我们从execute()方法所返回的一个输出参数映射中构造出了一个返回值。这意味着使用了我们的存储过程对象的代码能够使用强有力的类型化。至于较复杂的存储过程，变元和返回值可以具有应用对象类型。

```
public int execute(int amount, int custid) {
    Map in = new HashMap();
    in.put("amount", new Integer(amount));
    in.put("custid", new Integer(custid));
    Map out = execute(in);
    Number Id = (Number) out.get("newid");
    return Id.intValue();
}
```

仅有20行Java代码，而且我们拥有了一个能够实现一个不是存储过程或JDBC特有接口的对象。

JDBC 3.0中的一项改进，使得给CallableStatement接口使用命名参数而不是索引参数成为了可能。这把我们已经介绍过的StoredProcedu类的那些特性之一引入到了JDBC API中。不过，使用一种较高级别的抽象仍比JDBC API有意义，因为使用JDBC API仍存在错误处理问题。

JDBC抽象小结

我们刚刚描述过的抽象并不是让JDBC的处理变得更容易的惟一方式。不过，它们比直接使用JDBC更可取，并且可以显著地减少应用中的代码中和错误的出现概率。

最大的收获与错误处理有关。我们已经见过一个普通的数据存取异常分级结构可以给

应用代码赋予对具体问题作出反应的能力，同时又允许应用代码忽略绝大部分不可恢复的问题。这也保证了业务对象不需要对该应用所使用的持久性策略（比如JDBC）有任何了解。

我们已经见过两个抽象的级别。`com.interface21.jdbc.core`包（使用回调接口来使JDBC工作流程和错误处理能够由该框架来管理）使得JDBC更容易使用，但让应用代码去应付JDBC `PreparedStatement`和`ResultSet`。

`com.interface21.jdbc.object`包依靠`com.interface21.jdbc.core`包来提供一个更高的抽象级别，在该抽象级别上，RDBMS操作（查询、更新和存储过程）被建模为可重用对象。这通常是一种更好的方法，因为它使SQL操作局部化在了`RdbmsOperation`对象内，并使得使用它们的代码变得简单和意义自明。

和大多数O/R映射方法不同，这些抽象没有对RDBMS使用的任何控制权。我们可以执行我们所喜欢的任何SQL，能够轻松地执行存储过程。我们已经简单地使JDBC变得容易使用。

机警的读者可能已经发现，笔者疏忽了自己在第4章中曾经提过的关于外部化串的忠告。上面介绍的那些类包括了SQL串字面值——但不是常数。使SQL独立于Java代码的愿望是有问题的。虽然我们应该总是外部化配置串，但SQL并不是真正的配置数据，而是代码。如果SQL代码发生改变，使用了它的Java代码的行为也可能会改变，因此外部化SQL串可能是不明智的。例如，假设我们有同一个查询的两个版本，其中一个版本后缀了FOR UPDATE。这两个版本是不同的查询——它们将表现得有极大差异。由于不想修改Java代码而想使修改SQL串变得很容易会很危险，所以SQL串应放在DAO中（相反，不想修改代码而想让修改配置变得更容易是完全有益的）。

需要注意的是，虽然我们可能不希望把SQL与Java分开，但我们肯定希望使所有SQL都局部化在DAO实现中。这么做不仅使修改SQL（如果必要）变得更容易，而且也向应用的其余部分隐藏掉了SQL。

使用了这些JDBC抽象包的应用代码容易测试。由于所有框架类都从一个数据源中获得连接，所以我们可以轻松地提供一个测试数据源，进而使数据存取代码能够在没有应用服务器的情况下进行测试。由于运行在一个J2EE服务器内的任何代码都能从JNDI中获得一个得到管理的DataSource，所以我们可以在一个EJB容器的内部或外部使用这样的DAO，进而提高了体系结构上的灵活性。

框架代码本身是相当简单的。它并不试图解决真正复杂的问题；它只是消除了使处理JDBC变得可怕和容易出错的那些障碍。因此，学习它没有什么太多的困难。例如，这种方法不提供对加锁或并发的任何特殊处理。实质上，它允许我们用最小的努力来使用SQL。它向我们保证了我们的目标RDBMS行为正确。

同样，那个JDBC抽象接口使得我们能够在事务管理方面不必花费任何精力。应该使用“全局” JDBC API或EJB CMT来管理事务。如果我们使用JDBC API来管理事务，我们就丧失了让J2EE服务器来征募同一个事务中的所有资源并自动在必要时回退所有操作的能力。

Floyd Marinescu在“EJB Design Patterns”一书中描述了“Data Access Command Bean”设计模式，进而给出了JDBC的示例和常用超类。这种方法和此处所描述的那种方法具有相类似的目标，但这两种方法的区别在于：数据存取对象是命令（为单独使

用而设计），API基于javax.sql.RowSet接口，并且应用代码在从结果集中提取每个列值时必须捕捉异常。当处理许多数据库列时，这显然是极其罗嗦的（根本没有异常处理应该独立于数据存储的概念：应用代码被要求直接使用JDBC错误处理）。这种“Data Access Command Bean”方法比直接使用JDBC更可取，但笔者认为本章中所描述的那种方法更出色。

一个RdbmsOperation不是一条命令。命令一般是按每个用例来创建的，但一个RdbmsOperation只被创建一次，然后被重用。不过，RdbmsOperation模型兼容于“Data Access Command Bean”设计模式。一旦得到了创建和配置，一个RdbmsOperation就能重复地执行命令。

在示例应用中实现DAO模式

在武装了我们的普通JDBC基础结构之后，让我们来看一看如何实现本书示例应用中的数据存取。

让我们将焦点放在订票过程上。要想使用DAO模式，我们需要使持久性逻辑操作独立于业务逻辑操作。第一步是设法创建一个技术无关的DAO接口，然后我们可以使用自己所选择的任何一种持久性技术来实现该接口。

我们将在下一章中探讨业务逻辑的实现。现在，让我们来看一看BoxOffice接口上最重要的方法——入场券预订系统的公用接口：

```
public interface BoxOffice {
    int getSeatCount(int performanceId);
    int getFreeSeatCount(int performanceId);
    SeatQuote allocateSeats(SeatQuoteRequest request)
        throws NotEnoughSeatsException;
    // Additional methods omitted
}
```

现在，我们可以忽略SeatQuote和SeatQuoteRequest参数类的细节。PriceBand类是一个只读对象，我们将在下文中分析它。

我们需要把座位分配算法保持在Java中。在这种情况下，把那些必需的持久性操作隔离到一个DAO接口中是一个简单的练习（通常，这种隔离是很难实现的；有时，这是不可能的）。我们必须设法保证该接口（虽然没有暗示一个持久性策略）兼顾到有效的实现。一个适用的DAO接口看起来将像下面这样：

```
public interface BoxOfficeDAO {
    /**
     * @return collection of Seat objects
     */
    List getAllSeats(int performanceId) throws DataAccessException;
```

```
/*
 * @return list of PriceBand objects for this performance
 */
List getPriceBands(int performanceId) throws DataAccessException;

/**
 * @return list of Integer ids of free seats
 * @param lock guarantees that these seats will be available
 * to the current transaction
 */
List getFreeSeats(int performanceId, int classId, boolean lock)
throws DataAccessException;

int getFreeSeatCount(int performanceId) throws DataAccessException;

String reserve(SeatQuote quote);

// Purchase operation omitted
}
```

这个接口中没有什么东西限定我们必须使用JDBC、Oracle甚至一个RDBMS。这些方法每个都抛出我们的普通com.interface21.dao.DataAccessException，从而保证即便出现了错误，使用它的业务对象仍不必使用RDBMS概念。

传递给getFreeSeats()方法的第三个参数lock允许我们程序性地选择是否锁定被返回的那些座位。如果这个参数是true，该DAO必须保证：那些具有返回ID的座位得到了锁定，以防其他用户预订，并且这些座位保证能够在当前事务中被预订。如果这个参数是false，我们需要在不影响其他用户的情况下查询当前的空座位，以便显示信息，而不是显示为预订过程的一部分。

这个接口没有包含事务管理，事务管理应该使用JTA来实现。我们计划使用一个含有CMT的无状态会话EJB来声明性地处理事务管理，但不必操心实现并测试该DAO的事务管理。

至此，我们已经定义了一个DAO接口，我们现在应该编写针对它的测试。涉及到持久性数据的测试往往是很繁琐的，所以这里不打算展示那些测试类，但这一步是非常重要的。一旦我们有一套测试用具，就可以使用这套用具测试任何DAO实现；如果我们在某个时候需要迁移到另一项持久性技术或数据库，这将明显是十分宝贵的。为了创建有效的测试，我们必须：

- 保证在每个测试运行之前先创建测试数据。我们绝不能在一个生产数据库上运行这样的测试。我们可以使用一个JUnit测试案例的setUp()方法来创建测试数据，或者在测试案例运行之前使用Ant的SQL能力。
- 编写确认数据库正在工作的测试方法。对于每个测试，我们将编写代码来检查所述操作的结果，进而直接建立与数据库的连接。使用我们的JDBC抽象API（我们可以假设该API已经过了测试），我们不用编写太多的代码即可发布查询。
- 保证我们把数据库回复到它的旧状态。我们在测试期间将提交许多事务，所以我们无法回退，因此这一步可能是十分复杂的。我们可以使用一个JUnit测试案例的tearDown()方法，或者使用Ant或一个类似工具。

在尝试DAO接口的一个真正实现之前，我们可以通过使用所有测试在里面都应该失败的虚构实现来编写和运行测试（该虚构实现的方法应该什么事情都不做，只返回null，或者抛出java.lang.UnsupportedOperationException来检查该套测试工具。任何一个IDE都能帮助我们不用手工编程即可创建一个接口的这种虚构实现）。

在我们拥有了一套完整的测试工具之后，让我们来看一看如何使用JDBC来实现我们的DAO接口（我们可以使用SQL，但使用我们的JDBC抽象层，它几乎是不必要的）。关于这个实现的一个完整清单，请参考com.wrox.expertj2ee.ticket.boxoffice.support.jdbc.OracleJdbcSeatingPlanDAO类。

使用上面所描述的基于对象的JDBC抽象，我们将为每个必需的查询、更新和存储过程调用都创建一个RdbmsOperation对象。所有用到的RdbmsOperation对象都将被建模为内部类，因为它们只应该在OracleJdbcSeatingPlanDAO类中才是可见的。这也具有这个内层类的DataSource成员变量ds是可见的优点，进而简化了对象的构建。

首先，让我们来看一看PriceBand查询的实现，该实现返回轻量级的只读对象（相对JDBC来说的一种理想应用）：

```
private class PriceBandQuery extends ManualExtractionSqlQuery {
    public PriceBandQuery() {
        super(ds, "SELECT id AS class_id, price AS price " +
            "FROM price_band WHERE price_band.price_structure_id = " +
            "(SELECT price_structure_id FROM performance WHERE id = ?) " +
            "ORDER BY price DESC");
        declareParameter(new SqlParameter("price_structure_id",
            Types.NUMERIC));
        compile();
    }

    protected Object extract(ResultSet rs, int rownum) throws SQLException {
        int id = rs.getInt("class_id");
        double price = rs.getDouble("price");
        return new PriceBand(id, price);
    }
}
```

这演示了比我们迄今为止所见过的查询稍微复杂一点的一个查询。SQL代码在该对象的代码中占大部分这一事实表明，我们拥有一个精确的Java API！getAllSeats()方法后面的那个查询将是类似的。

我们在OracleJdbcSeatingPlanDAO类中把一个PriceBandQuery对象声明为一个实例变量，并在构造器中按如下所示初始化它：

```
private PriceBandQuery priceBandQuery;
this.priceBandQuery = new PriceBandQuery();
```

使用PriceBandQuery实例，来自BoxOfficeDAO接口中的getPriceBands()方法的实现是微不足道的：

```
public List getPriceBands(int performanceId) {
    return (List) priceBandQuery.execute(performanceId);
}
```

现在，让我们来看一看查找空座位的查询。还记得吗，数据库模式在两个重要的方面简化了我们的任务：该数据库模式提供了AVAILABLE_SEATS视图，以阻止我们需要执行一个外连接来查找空座位，并且还提供了reserve_seats存储过程来隐藏为BOOKING表生成一个新主键的过程，以及相关的更新和插入。这使查询变得直截了当。要想获得某一场演出的某一给定类型的空座位的id，我们可以对我们的视图运行一个像下面这样的查询：

```
SELECT seat_id AS id FROM available_seats  
WHERE performance_id = ? AND price_band_id = ?
```

为了保证在lock参数是true时我们尊重getFreeSeats()方法的约定，我们需要一个查询来把FOR UPDATE附加到上面所示的那个选择上。这两个查询是截然不同的，所以笔者已经把它们建模为独立的对象。

这里是需要考虑的两点：Oracle只允许一个事务中有SELECT...FOR UPDATE。我们必须记住这一点，并保证我们在测试这个代码时有一个事务；而且，对一个视图所使用的FOR UPDATE子句将正确地锁定那些底层表。

下面这两个查询共用相同的参数和提取逻辑，并且它们能被收入一个公用基类中。一个抽象超类实现每行数据的提取，而子类则多样化该查询的WHERE子句，并且参数的使用方法是非常强有力的；这只是一个微不足道的例子。这两个查询及其超类看起来将会像下面这样：

```
private static final String FREE_SEATS_IN_CLASS_QUERY_SQL =  
    "SELECT seat_id AS id FROM available_seats " +  
    "WHERE performance_id = ? AND price_band_id = ?";  
  
private abstract class AbstractFreeSeatsInPerformanceOfTypeQuery  
    extends ManualExtractionSqlQuery {  
  
    public AbstractFreeSeatsInPerformanceOfTypeQuery(String sql) {  
        super(ds, sql);  
        declareParameter(new SqlParameter("performance_id", Types.NUMERIC));  
        declareParameter(new SqlParameter("price_band_id", Types.NUMERIC));  
        compile();  
    }  
  
    protected Object extract(ResultSet rs, int rownum) throws SQLException {  
        return new Integer(rs.getInt("id"));  
    }  
}  
  
private class FreeSeatsInPerformanceOfTypeQuery  
    extends AbstractFreeSeatsInPerformanceOfTypeQuery {  
  
    public FreeSeatsInPerformanceOfTypeQuery() {  
        super(FREE_SEATS_IN_CLASS_QUERY_SQL);  
    }  
}  
  
private class LockingFreeSeatsInPerformanceOfTypeQuery  
    extends AbstractFreeSeatsInPerformanceOfTypeQuery {  
  
    public LockingFreeSeatsInPerformanceOfTypeQuery() {  
        super(FREE_SEATS_IN_CLASS_QUERY_SQL + " for update");  
    }  
}
```

OracleJdbcSeatingPlanDAO构造器（在一个DataSource可用之后）能够像下面这样创建这两个具体类的新对象：

```
freeSeatsQuery = new FreeSeatsInPerformanceOfTypeQuery();
freeSeatsLockingQuery = new LockingFreeSeatsInPerformanceOfTypeQuery();
```

现在，通过调用来自SqlQuery超类中的execute(int, int)方法，我们可以利用这两个查询中的任何一个来查询空座位。我们使用要传递给来自SeatingPlanDAO接口中的getFreeSeats()方法的lock参数来选择要执行哪个查询。这个方法的完整实现是：

```
public List getFreeSeats(int performanceId, int classId, boolean lock) {
    if (lock) {
        return freeSeatsLockingQuery.execute(performanceId, classId);
    } else {
        return freeSeatsQuery.execute(performanceId, classId);
    }
}
```

调用该存储过程来完成一个预订要稍微复杂一点。虽然我们已经通过依靠我们对Oracle的加锁行为的了解，隐含地编码到Oracle，但迄今为止，我们还没有做任何专有的事情。由于PL/SQL存储过程接受表参数，所以要想在一个单独的数据库调用中能够传递要预订的那些座位的ID，我们需要依据一些Oracle特有的规定来行事。

首先，让我们来看一看我们需要在数据库中做些什么。我们需要定义下面这些自定义类型：

```
CREATE OR REPLACE TYPE seatobj AS object(id NUMERIC);
/
CREATE OR REPLACE TYPE seat_range AS table OF seatobj;
/
```

现在，我们可以使用seat_range类型作为要传递给下列PL/SQL存储过程的一个表参数：

```
CREATE OR REPLACE
PROCEDURE reserve_seats(
    perf_id IN NUMERIC,
    seats IN seat_range,
    hold_till DATE,
    new_booking_id OUT NUMBER)
AS
BEGIN
    -- Get a new pk for the booking table
    SELECT booking_seq.nextval INTO new_booking_id FROM dual;

    -- Create a new booking
    INSERT INTO booking(id, date_made, reserved_until)
        VALUES (new_booking_id, sysdate, hold_till);

    -- Associate each seat with the booking
    FOR i IN 1..seats.count LOOP
        UPDATE seat_status
            SET booking_id = new_booking_id
            WHERE seat_id = seats(i).id
            AND performance_id = perf_id;
    END LOOP;
END;
/
```

为了把一个Java数组作为表参数传递给该存储过程，我们还需要在我们的JDBC中执行一些Oracle特有的操作。

.StoredProcedure超类不仅允许我们在一个Map中到达execute()方法，而且还允许我们进入到这样一个回调接口中：它在给定一条Connection的情况下创建一个参数映像：

```
protected interface ParameterMapper {  
    Map createMap(Connection con) throws SQLException;  
}
```

这在我们需要该Connection来构造适当的参数时是必不可少的，就像我们在本例中所做的那样。在执行该存储过程之前，reserve_seats存储过程对象需要使数据库类型seat和seat_range能让JDBC使用。这两个类型只有使用一条Oracle连接才能变得可用。除此之外，创建一个输入参数Map（映像）并提取输出参数的过程和我们在前面见过的那个简单的.StoredProcedure例子中的相同：

```
private class SeatReserver extends StoredProcedure {  
  
    public SeatReserver(DataSource ds) {  
        super(ds, "reserve_seats");  
        declareParameter(new SqlParameter("perf_id", Types.INTEGER));  
        declareParameter(new SqlParameter("seats", Types.ARRAY));  
        declareParameter(new SqlParameter("hold_till", Types.TIMESTAMP));  
        declareParameter(new OutputParameter("new_booking_id", Types.INTEGER));  
        compile();  
    }  
  
    public int execute(final int performanceId, final int[] seats) {  
        Map out = execute(new StoredProcedure.ParameterMapper() {  
            public Map createMap(Connection con) throws SQLException {  
  
                con = getOracleConnection(con);  
  
                // Types MUST be upper case  
                StructDescriptor sd = StructDescriptor.createDescriptor(  
                    "SEATOBJ", con);  
                ArrayDescriptor ad = ArrayDescriptor.createDescriptor(  
                    "SEAT_RANGE", con);  
  
                Object[] arrayObj = new Object[seats.length];  
                for (int i = 0; i < arrayObj.length; i++) {  
                    arrayObj[i] = new Object[] { new Integer(seats[i]) };  
                    //System.out.println("Will reserve seat with id " +  
                    // new Integer(seats[i]));  
                }  
  
                // Need Con to create object (association with Map)  
                ARRAY seatIds = new ARRAY(ad, con, arrayObj);  
  
                Map in = new HashMap();  
                in.put("perf_id", new Integer(performanceId));  
                in.put("seats", seatIds);  
                Timestamp holdTill = new Timestamp(System.currentTimeMillis()  
                    + millisToHold);  
                in.put("hold_till", holdTill);  
                return in;  
            }  
        });  
    }  
}
```

```

        }
    }:
    Number Id = (Number) out.get("new_booking_id");
    return Id.intValue();
}
}

```

处理数据库中的一个自定义类型增加了JDBC代码的复杂性，无论我们使用了何种抽象层（类型描述符可以用来利用Oracle和其他数据库中的对象-关系特性，尽管还存在更简单的方法，比如Oracle的JPublisher，在较复杂的情形中，我们应该考虑使用这些方法）。由于本书不是一本论述高级JDBC和Oracle JDBC的书籍，所以我们不打算细说这方面的具体内容，此处的宗旨是说明我们怎么才能既使用专有的RDBMS特性，同时又不对我们的体系结构产生不良影响，因为它们被收入了实现一个公用接口的一个统一类中。

需要注意的是，我们使用一个java.sql.Timestamp来保存Oracle DATE类型。如果使用一个java.sql.Date，我们将会失去精确性。

突出显示的行说明了这个`StoredProcedur`e要求能够从它的数据源所提供的连接中获得一条Oracle特有连接。在大多数应用服务器中，数据源将会返回已封装的连接，而不是一条RDBMS供应商特有的连接（连接封装用来实现连接池和事务控制）。但是，如果必要，我们始终可以获得底层连接。我们需要覆盖`OracleJdbcSeatingPlanDAO getOracleConnection()`方法，该方法接受一个池化的`Connection`，并为每个应用服务器都返回底层的Oracle特有连接。下列覆盖（在JBoss30OracleJdbcSeatingPlanDAO类中）将为JBoss 3.0工作：

```

protected Connection getOracleConnection(Connection con) {
    org.jboss.resource.adapter.jdbc.local.ConnectionInPool cp =
        (org.jboss.resource.adapter.jdbc.local.ConnectionInPool) con;
    con = cp.getUnderlyingConnection();
    return con;
}

```

关于`OracleJdbcSeatingPlanDAO`和`JBoss30OracleJdbcSeatingPlanDAO`的完整清单，请参见本书的配书下载。

通过使用DAO设计模式，我们已经能够既使用Oracle特有特性，又不影响我们的体系结构的可移植性。我们拥有了一个简单的接口，并且能够为其他任何一个关系数据库实现它。通过使用一个JDBC抽象库，我们已经极大地简化了应用代码，并且使它变得更易阅读，更不易出错。

小结

在本章中，我们探讨了适用于J2EE应用的一些最有效的数据存取方法。我们已经考虑过：

- 基于SQL的方法，比如JDBC和SQLJ。当我们正在使用关系数据库时，并且当O/R映射不适用时，这些方法可以用最小的开销提供非常高的性能（我们在第7章中讨论了决定是否使用O/R映射的标准）。由于大多数J2EE应用都使用关系数据库，所以基于SQL的方法在实践中是十分重要的。

- 基于O/R映射的方法，比如专有O/R映射框架和JDO。JDO特别值得关注，因为它可能会成为存取持久性数据的标准API，无论在RDBMS中还是在ODBMS中。

本章中所讨论的所有这些方法都能用在一个J2EE应用服务器的EJB容器或Web容器中。它们还可以在一个J2EE应用服务器的外部被轻易地测试：一个重要的考虑因素，就像我们在第3章中已经见过的那样。我们可以在含有BMP的实体组件中使用这些策略的任何一个，但正如我们已经见过的，BMP在许多情况中是不合用的，并且将强加一个几乎不或完全不提供任何好处的、复杂的、规定性模型。

我们最后得出的结论是：一种使用了JDBC的基于SQL的方法最适用于我们的示例应用，因为它无法从一个O/R映射层内的高速缓存中获得明显好处，而且我们已在第7章中看到我们可以充分利用存储过程把它的部分持久性逻辑转移到RDBMS中。但是，由于我们不打算让示例应用的设计依赖于一个RDBMS的使用，所以我们选择了使用DAO设计模式把JDBC的使用隐藏在由持久性存储无关性接口所构成的一个抽象层后面。

由于示例应用将使用JDBC，所以我们比较仔细地了解了JDBC API。我们看过了正确的错误处理的重要性（和难度），如何从一个javax.sql.SQLException中提取与某一问题的根源有关的信息，以及JDBC PreparedStatements和Statements的正反两方面。

我们最后得出的结论是：直接使用JDBC API是一种没有前途的选择，因为它要求编写太多的代码来完成数据库中的每项任务。因此，我们需要一个比JDBC API所提供的更高级别的抽象，即便我们不打算使用一个O/R映射框架。

我们分析了一个普通JDBC抽象框架的实现，这个框架就提供了这种级别的抽象。我们在本书的示例应用中使用了这个框架，这个框架也可以用在其他任何一个使用JDBC的应用中。这个框架提供两个级别的抽象：

- com.interface21.jdbc.core包，它使用回调方法使JdbcTemplate框架类能够处理JDBC查询和更新的执行以及JDBC出错处理，从而消除了在使用JDBC时应用代码出错的最常见根源。
- com.interface21.jdbc.object包，它依靠com.interface21.jdbc.core包把每个RDBMS查询、更新或存储过程调用建模为一个可重用的、线程安全的Java对象，从而在应用代码面前彻底隐藏掉了JDBC和SQL使用的细节。使用这个包，我们可以极大地降低使用JDBC的应用代码的复杂性。

我们的抽象框架的另一个重要好处是：它使利用了它的代码能够处理一个由数据存取异常，而不是JDBC SQL异常所构成的普通分级结构。这保证业务对象绝不依赖于JDBC特有的概念。由于所有数据存取异常都是运行时异常，而不是已检查异常，所以通过能够忽略不可恢复异常，应用代码得到了极大的简化（JDO就充分说明了这种方法的效力）。

我们了解了示例应用如何使用我们的抽象框架来实现DAO设计模式。使用该抽象框架能使我们编写比直接使用JDBC所要求的简单得多的代码。使用DAO设计模式使我们既能使用目标数据库Oracle的专有特性，同时又使示例应用的总设计不依赖于Oracle，甚至不依赖于关系概念。在这种情况下，专有特性的使用是有正当理由的，因为这使我们能够使用有效的RDBMS结构，以及使用PL/SQL存储过程来处理用Java代码实现起来困难得多的持久性逻辑。

最后，我们分析了J2EE应用中的数据存取。

在下一章中，我们将了解会话EJB的有效使用。

第10章 会话组件

在第1章和第6章的J2EE应用总体设计的讨论中，我们已经分析了如何决定什么时候使用EJB，以及EJB容器所提供的关键服务，比如容器管理式事务。在第8章中，我们分析了实体组件，并得出了这样的结论：它们在大多数情况中是数据存取的一种有疑问选择；因此，只有会话组件才提供了EJB的大部分价值。

在本章中，我们将较深入地了解用会话组件实现业务逻辑所关系的问题。

首先，我们将讨论无状态和有状态会话组件之间的选择。正如我们将要看到的，有状态会话组件具有明显的缺点（主要关系到聚类化环境中的可靠性和可缩放性），这就是说，正常情况下应该首选无状态会话组件。

接着，我们将了解与会话组件使用有关的两个重要的J2EE设计模式：**Session Facade**和**EJB Command**模式。

Session Facade模式使一个会话组件（通常是一个无状态会话组件）成为进入EJB层的入口点。这个模式在分布式应用中是极其重要的，因为会话组件为这类应用提供了远程接口。像实体组件或Data Access Object之类的实现细节被隐藏在会话门面（Session Facade）的背后。会话组件也可能用来实现一个Web服务接口。有些应用服务器已经支持这一点，而且EJB 2.1增加了对Web服务端点（Endpoint）的支持。在集中式应用中，会话组件用来实现业务接口，而且在这种接口中，EJB容器服务可以简化应用代码。

相反，**EJB Command**设计模式的基础是把代码（不仅仅数据）从客户端转移到EJB容器上。虽然这个模式与**Session Facade**模式在会话组件的较常见用法方面有着明显的不同，但它也是使用无状态会话组件实现的。

然后，我们将详细地了解影响会话组件的容器服务，比如EJB容器在EJB抛出异常时的响应方式和实现会话组件的后果，以及声明性事务属性怎么才能用来保证CMT提供必要的行为方式。

最后，我们将分析编写会话组件时的一些良好实现习惯。

在分布式J2EE应用中，应该使用会话组件来实现应用的用例。在集中式J2EE应用中，应该使用会话组件来实现能够从EJB容器中获益的业务逻辑。如果需要选择的话，一般应该优先使用无状态会话组件。

使用无状态会话组件

无状态会话组件是EJB系统的主要成分。它们以性能和部署复杂性方面的最小开销提供J2EE的关键好处——容器管理式生成周期、远程访问、自动线程安全以及透明事务管理。

无状态会话组件的好处

无状态会话组件（简称SLSB）对分布式应用来说是最具可缩放性和可靠性的J2EE构件。

正如我们在第6章中所见过的，只有使用了像SLSB之类的无状态业务对象的分布式应用才有可能比集中式应用更有可缩放性，更坚固。在集中式应用中，SLSB提供了优于普通Java类的较少好处。

SLSB特别适合运行在聚类环境中，因为从客户端的角度看，所有实例都是一致的。这就意味着进入调用总是能被定向给性能最佳的服务器。相反，有状态会话组件很可能会遇到服务器仿射性（Server Affinity）问题：此时，客户束缚于某一个特定的服务器，而不管对该服务器还有无其他要求。因此，无状态会话组件部署具有极高的可靠性，即便有一个或多个服务器实例发生了故障。WebLogic在这方面做得极为出色，即使在一个方法调用期间有一个无状态会话组件实例发生了故障，它也能够恢复。这样的恢复由客户端EJBObject占位程序（Stub）来完成，如果该方法在一个WebLogic部署描述符中被标识为幂等的（Idempotent），即如果对该方法的反复调用不会导致重复的更新。

由于所有实例都是一致的，所以一个EJB容器能够管理一个无状态会话组件池（Pool），保证对每个会话组件的业务方法来说，线程安全都能够得到保障，同时又不妨碍客户端对无状态会话组件的访问。这种具有管理式生成周期的无状态组件的可缩放性在除了J2EE之外的其他平台上已经得到证实，比如在EJB之前出现的Microsoft Transaction Server（MTS）。

使用无状态会话组件的缺点是：从客户端角度看，它们不是真正的对象（它们缺少同一性）。这就给OO设计提出了难题，但令人遗憾的是，可缩放企业级体系结构的现实要求一些牺牲。由于无状态会话组件常常只是门面（Facade）对象，所以它们能够位于EJB层内的一个面向对象体系结构的上面。

无状态会话组件与内部状态

有一个常见的误解：无状态会话组件不能保存状态。无状态会话组件之所以是无状态的，是因为它们在方法调用之间不为任一单独的客户保存状态。它们能够随意维持内部状态——例如，为它们所服务的所有客户软件高速缓存用到的资源和数据。

这个区别是十分重要的，因为它能使我们通过执行某些慢速操作（比如只读数据的检索）一次并高速缓存这些操作的结果，优化无状态会话组件。这样的数据能够在一个组件实例被创建和该容器调用它的ejbcreate()方法时，或者在它被一个业务方法首次需要时被高速缓存。EJB容器允许我们控制该池中的无状态会话组件的个数，这意味着我们能够在实践中保证会话组件实例会得到重用。一个无状态会话池的大小应该总是小于该服务器的配置所允许的最大线程个数。维护一个大于最大线程个数的SLSB实例池将会浪费服务器资源，因为所有实例将同时得到使用的情景是不可能出现的（仅当只存在一个使用全部SLSB实例的执行线程时，这种情景才会出现）。建池的意义是为了最小化各SLSB实例的争用；维护超出最大个数之外的、能够被同时使用的实例没有任何用处。

无状态会话组件建池的影响

正是通过SLSB建池，EJB容器才允许我们编写SLSB代码，就好像它是单线程的。这样的建池实际上是在J2EE应用服务器中实现的（尽管普通Java对象是不是更可取将视具体情况

而定)，而且我们通常不必为它操心。

但是，在给SLSB分配操作时，考虑使用模式和SLSB实例建池的影响有时是很重要的。例如，假设按用例分组意味着一个CustomerServices无状态EJB除了提供一个performWeirdAndWonderful()方法之外，还提供一个login()方法。login()方法被频繁使用，并且不要求状态在该无状态会话组件中被保持。另一方面，performWeirdAndWonderful()方法很少被调用，但要求一些状态在该无状态会话组件中被保持（即内部状态——不暴露给任一个别客户软件的状态）。在这种情况下，比较好的做法是把CustomerServices组件分裂成两个组件。

一个组件含有经常使用的login()方法。由于它只占用极少的内存，所以会有一个含有这些组件的大池。另一个组件含有performWeirdAndWonderful()方法。为了支持该组件的使用模式，可能只需要一个单独的实例，或者说一个非常小的池，从而节省了内存。

弄清楚EJB容器在运行时怎样为EJB的一个特定类执行池管理常常是十分重要的。一些像WebLogic那样的服务器提供了GUI工具，以便我们能监视每个已部署EJB的池实例个数。不过，笔者发现下列策略也是有帮助作用的：

- 在ejbCreate()方法中使用记录日志消息来显示实例是何时被创建和初始化的。
- 在EJB实现类中维护一个静态实例变量作为一个计数器，每调用一次ejbCreate()方法递增一次该计数器。需要注意的是，这个策略违背了EJB只保持读/写实例数据的约束，并且在聚类环境中不管用。它只对显示一个单服务器部署中的建池行为来说才是一个有用的技巧。

不用说，这两个策略只应该在部署阶段使用。

使用有状态会话组件

有状态会话组件（简称SFSB）有它们自己的用处，但与它们的无状态同伴相比，距离成为EJB的主要组成还很远。它们是否应该受到它们在EJB规范中被给予的重视程度仍有疑问，该规范暗示有状态会话组件是标准，而无状态会话组件是一个特例。

与无状态会话组件不同的是，有状态会话组件增加了显著的内存和性能开销。有状态会话组件在客户软件之间不被共享，因而意味着服务器必须为每个客户软件管理一个SFSB实例。相反，一些SLSB实例可以服务于许多客户软件的请求。

应用常常可以被设计成使用无状态方法来完成它们的工作。在可以实现这一点的地方，通常将会产生杰出的可缩放性，即使在必须维持服务器端状态时，有状态会话组件也不是唯一而且也可能不是最佳的选择。

为什么不使用有状态会话组件

避免使用有状态会话组件存在一些十分强有力的理由。下面让我们来看一看其中最重要的理由。

性能与可缩放性问题

一个服务器上所维持的SFSB实例个数与某时刻的不同活动客户的个数成正比。由于SFSB是比较重型的对象，所以这就限制了一个应用在某一给定部署配置中能够同时服务的客户软

件的个数。这也给在完成时要释放服务器资源的客户软件增加了一项对无状态组件或实体组件的客户软件不适合的工作。

如果使用了SFSB，应该保证当这些SFSB不再被需要时，客户代码通过调用这个会话组件的构件或主接口上的remove()方法删除它们。

为了帮助管理维持有状态会话组件实例所强加的开销，EJB规范允许容器使用它们所选择的一个策略来钝化和重新激活有状态会话组件实例。这意味着最近最少使用的实例可能会从内存中被交换到一个持久性存储器上（存储器的选择留给了容器）。钝化和重新激活通常是使用到一个数据库Binary Large Object (BLOB) 或磁盘文件的Java语言串行化来实现的，尽管EJB规范允许EJB容器使用其他策略。

EJB 2.0规范的7.4节中描述了有状态会话组件的、幸免于钝化和重新激活的、转换状态的子集。除了保留像EJB本地和远程主接口、EJB本地和远程接口、SessionContext参考、JNDI上下文以及资源管理器连接工厂参考之类的企业资源之外，该子集还遵守用于Java语言串行化的规则。实现ejbPassivate()方法（容器调用它来通知一个实例阻止钝化）来保证实例方法持有合法值是会话组件提供者的责任。

当编写有状态会话组件时，要保证会话组件能够根据EJB规范的7.4节所概述的那些规则来钝化，并且当在钝化后被重新激活时将具有正确的行为。这意味着应该使用ejbPassivate()方法来保证实例变量在组件钝化发生之前就有合法的值。

例如，任何JDBC连接都必须被关闭，同时那些参考实例变量被设置成null。ejbActivate()方法必须保证不能得到保留的任何必要资源都被复位成有效值。

下列代码示例显示了一个假想的有状态会话组件的部分正确实现，该会话组件使用了一个助手实例变量MyHelper，并且该变量不是可串行化的，因此也不是一个实例变量在钝化时的合法值：

```
public class MySessionBean implements javax.ejb.SessionBean {  
    private MyHelper myHelper;
```

ejbCreate()方法在该SFSB首次被使用时初始化这个实例变量。需要注意的是，ejbActivate()方法仅当一个组件在钝化之后又被重新激活时才会得到调用；它不属于容器所做的该组件的初始化（请参见EJB规范的7.6节中的生成周期图表）：

```
public void ejbCreate() {  
    this.myHelper = new MyHelper();  
}
```

ejbPassivate()方法的实现把该实例变量设置成null，该值是一个可接受的值：

```
public void ejbPassivate() {  
    this.myHelper = null;  
}
```

ejbActivate()方法的实现重新初始化该助手，如果该组件变量被重新激活的话：

```
public void ejbActivate() throws CreateException {
    this.myHelper = new MyHelper();
}
```

请注意，`myHelper`实例变量可以被标记为瞬态，而在这种情况下，在`ejbPassivate()`方法中将它设置成`null`是没有必要的，但在`ejbActivate()`方法中重新初始化它是必要的。

还记得第4章中关于Java 1.4断言的讨论吗？`ejbPassivate()`方法把实例数据保持在一个合法状态的检查就是断言的一个上佳运用。

另外还需要记住，容器能够使已钝化的有状态会话实例超时。这意味着它们不再能被客户软件使用，客户软件在试图访问它们时将会接收到一个`java.remote.NoSuchObjectException` (`java.remote.RemoteException` 的一个子类)。

另一方面，由于降低了需要随同每个方法调用一起传递给服务器的上下文个数，所以使用有状态会话组件可能会改进分布式环境中的性能（与使用无状态会话组件相比）：一个有状态会话组件为它的客户端保持着状态，因此通常只需要被告知做什么，不必被告知对什么数据做什么。

可靠性问题

假定EJB的设计目标是帮助建立稳固、可缩放的应用，那么令人惊奇的是有状态会话组件对上述任一方面都没有太大帮助。中心问题是：有状态会话组件给状态在聚类化服务器之间的有效复制（Replication）造成了困难——有效复制是一个最根本的要求，如果服务器证明它们随着应用部署规模的扩大是可靠的。

EJB规范对有状态会话组件的故障切换未做任何保证。这就意味着利用一个有状态会话组件来实现一个联机商店以保持每个用户的购物车（就像EJB规范中所建议的那样）是危险的。可能的情况是：如果存储任一特定购物车实例的服务器因故障而停机，该用户（连同许多其他用户一起）将会失去他们的购物车。为了得到真正可靠的EJB状态管理，惟一的办法是使用数据库持久性——该持久性利用另一个持久性策略并通过实体组件或会话组件来管理。

为了弄清楚有状态会话故障切换的各种限制，让我们来看一看有状态会话组件在WebLogic 7.0内的一个聚类中是怎样表现的（WebLogic 7.0可能是目前市场上最领先的服务器）。WebLogic从版本6.0以后一直只支持有状态会话组件的对话状态的复制，并且这种复制由于它对性能的潜在影响而受到了限制。每个有状态会话组件在聚类中都有一个主（Primary）和副（Secondary）WebLogic服务器实例。

请注意，故障切换支持及必需的复制与一个聚类中有状态会话组件创建的负载平衡不是同一回事情，WebLogic一直都支持后者。

在正常情况下，EJBObject占位程序（即由WebLogic所提供的、该会话组件的远程接口的实现）把所有调用都定向给一个指定组件在主服务器上的实例。每当一个事务在该有状态会话组件上被提交时，WebLogic就利用内存中复制（In-memory replication）把该组件的状态复制到副服务器。如果主服务器实例发生故障，后续的客户方法调用将被定向给副服务器。

上的EJB实例，然后副服务器将成为新的主服务器。一个新的副服务器将被分配，以便恢复故障切换能力。这种方法具有下列结果：

- 在正常状况中，有状态会话组件束缚于单个服务器，从而导致服务器亲合性。
- 我们无法保证在万一发生故障时状态将能得到保留，因为复制发生在内存中，而不是发生在一个后备持久性存储器上。由于复制只涉及到两个服务器，所以这两个服务器都有可能发生故障，进而彻底丢失该组件的状态。
- 我们无法保证客户将始终见到有状态会话组件所更新的持久性资源的已提交状态。虽然这种情景是不可能出现的，但如下情况是可能的：一个事务相对于主服务器上的有状态会话组件实例来说已被成功提交，但主服务器在这个有状态会话组件的状态被复制到副服务器之前却发生了故障。当客户相对于这个有状态会话组件调用一个方法时，该调用将会转到旧的副服务器（现在的主服务器），而副服务器将拥有过时的对话状态，这可能会与任何已提交的持久性更新发生冲突。
- 在一个客户正等待来自一个有状态会话组件的一个调用的返回结果时，从主服务器上的故障中完整地恢复过来是不可能的。这样的恢复对无状态会话组件来说或许是不可能的。
- 甚至连内存中的复制对性能都有影响。相反，在一个聚类中的无状态会话组件实例之间，根本没有复制状态的需要。
- 无论一个容器采取了怎样的复制策略，在万一发生事务回退时，正确地把一个有状态会话组件上的对话状态恢复原状完全是会话组件提供者的责任。即使有状态会话组件实现了SessionSynchronization接口（下文讨论），以便使它能收到关于事务边界的通知，但仍必须编写可能十分复杂的代码来处理回退。

相比之下，设置在Servlet API HttpSession中的数据处理通常稳固得多。WebLogic和其他产品还提供了使用一个数据库来备份HttpSession状态持久性的选择，而不是只提供内存中复制机制（当然，数据库备份式持久性有很高的性能开销）。

就可靠性和可缩放性来说，最大问题或许是一个SFSB的状态在它的串行化表示中可能被作为一个整体来复制，而与它自上次被复制以来的变化程度无关。EJB规范没有提供任何办法来标识一个SFSB的状态的哪个部分在每个方法调用之后已发生了变化，而让容器在运行时解决这个问题又是十分困难的。

要求复制每个SFSB的整个状态所产生的开销通常比为HttpSession对象所需要的开销大得多，因为只有当重新绑定后续变化时，才会给HttpSession对象绑定和复制多个精细属性。Oracle 9iAS提供了高度可配置的状态管理，并提供了一种非移植方法，以便用户能够使用专有的StatefulSessionContext来标记一个SFSB的状态的哪些部分已经发生了变化，并需要被复制，进而模仿HttpSession行为方式（详细信息，请参见http://otn.oracle.com/tech/java/oc4j/doc_library/902/ejb/cluster.htm#1005738）。一个专有类的运用也暗示了在EJB规范内实现精细的SFSB复制是多么困难。

有状态会话组件的使用常常导致状态被保持在两个地方。客户代码只能从一个有状态会话组件中受益，如果它持有指向该组件的一个参考或句柄。例如，在一个Web应用中，一个HttpSession对象可能会持有一个参考指向该用户的无状态会话组件。这就意味着我们必须

为这一个用户管理两个会话状态对象，而不是一个会话状态对象。如果我们无论如何都需要一个HttpSession对象，那么通过把所有必需的会话状态都保持在该HttpSession对象中，而不是通过在EJB层中拥有另外一个有状态会话组件，我们或许能够避免上述情形。

回避这个问题的一种可能的方法是把该会话对象的句柄的一个表示保持在一个Cookie或隐藏表单字段中。但是，这种方法比使用一个HttpSession对象更复杂，并且不保证在所有服务器中都管用（我们不知道一个句柄可能有多大）。

有状态会话组件的拥护者争辩说，这个关于重复的问题不适合没有Web前端的客户。不过，这样的客户很可能比Web客户更复杂、更具交互性，而且很可能希望保持它们自己的状态——例如，一个独立的Swing应用就根本没有让它的状态在服务器上被管理的需要。

主张使用有状态会话组件作为会话组件的标准暗示了J2EE的一种不切实际的观点：认为J2EE应该工作的观点。事实上，SFSB可能不像开发人员所期望的那样工作，即便是在目前市场上最先进的应用服务器中。这严重地降低了它们的价值。

何时使用有状态会话组件

笔者认为，应该使用SFSB的情形是十分少见的。例如：

- **当一个无状态接口过分复杂并要求在整个网络中传输过量的用户数据时**

有时，试图利用无状态接口来建模EJB层会导致过度的复杂性。例如，要求过量的会话状态随同每个调用一起被传递的SLSB方法。

- **当多种客户类型要求状态在同一个地方被处理，甚至被共享时**

例如，一个SFSB可能会妥当地为一个小应用程序和一个服务器小程序应用保持状态（不过，这样的共享通常发生在持久性存储器中）。

- **为了处理经常连接和断开连接的客户**

这是一个典型的情形，在这种情形中，如果状态的可靠保护十分重要，状态必须被保存在数据库中。不过，如果一个SFSB提供了足够的可靠性，客户或许就能够存储一个SFSB句柄，并有效而随意地重新连接到服务器。

- **当可缩放性不是一个需要优先考虑的问题时**

有时，我们没有必要考虑利用有状态会话组件进行聚类的性能影响——例如，如果我们确信一个应用将从不运行在一个以上的服务器上。在这种情况下，就没有必要通过考虑状态复制的影响来使设计变得复杂（不过，在一个Web应用中，一个HttpSession对象有可能是更简单的选择方案）。

这4个标准的前3个只适用于分布式应用。

请注意，在证明使用SFSB有理由的上述标准当中，没有任何一个标准适用于Web应用——J2EE的最常见运用。

会话同步

有状态会话组件有一个非常特别的特性：一个有状态会话组件能够实现javax.ejb.Session-Synchronization接口，以接收与这样一些事务的进展情况有关的回调；这些回调被加进到这

些事务中，但不控制这些事务（请参见EJB规范的7.5.3节和17.4.1节）。这个接口含有如下3个方法：

```
public void afterBegin();
public void beforeCompletion();
public void afterCompletion(boolean success);
```

最重要的方法是`afterCompletion()`。如果该事务得到提交，`success`参数为`true`；如果它被回退，`success`参数为`false`。

`SessionSynchronization`接口允许SFSB重新设置关于事务回退的对话状态，通过一次参数值为`false`的`afterCompletion()`回调来指出。我们也可以使用这个接口来控制数据高速缓存：当该会话组件的状态指示我们仍处在一个事务中时，我们可以把得到高速缓存的数据作为最新数据来对待。我们也可能会选择高速缓存像数据库连接那样的资源。

一个有状态会话组件可能只实现`javax.ejb.SessionSynchronization`接口，如果它使用了CMT（让一个使用了BMT的组件去实现这个接口没有意义，因为它处理事务定界本身，并且没有必要被告知这一点）。只允许3个事务属性：`Required`、`RequiresNew`和`Mandatory`。实现`SessionSynchronization`接口的有状态会话组件是CMT的被动用户，因而被告知在其他地方定界的事务的进展情况（我们已经在前面讨论过如何使用CMT定界事务）。

会话同步提出了需要认真考虑的设计问题。如果有状态会话组件没有控制它自己的事务，那么谁来控制呢？无状态会话组件通常不调用有状态会话组件。很显然，我们也不希望实体组件调用有状态会话组件。因此，那些事务将可能在EJB层的外部被管理。

允许事务由远程客户来管理是非常危险的。例如，我们必须相信远程调用者在事务超时之前就结束该事务——如果它没有结束，有状态组件使之保持打开的任何资源可能会被锁定，直到容器最终做清理时才能得以释放。例如，假设远程客户在开始一个事务之后遇到了一个连接故障，那么该事务将被搁置起来直至超时，从而可能会锁定宝贵的资源不让其他构件访问。

最后这个危险不适用于一个集成应用部署中的同一个服务器实例的Web层中的对象所管理的事务。不过，这预示了一个可能的设计错误：如果我们正在使用EJB，那么我们为什么使用JTA来管理事务，而不是使用EJB CMT？可以证明的是，后者是EJB所提供的最大好处。

不要为有状态会话组件使用`javax.ejb.SessionSynchronization`接口。这会助长拙劣的应用设计，因为定界事务的是EJB层的远程客户，或者因为在EJB CMT是一个更好的选择时却为事务管理使用了JTA。

保护有状态会话组件，使之免于并发调用

对有状态会话组件做并发调用是非法的。当使用SFSB时，我们需要考虑保证这种情况不会发生。

这种情况在实践中很容易出现。例如，一个用户已经在一个Web站点上打开了两个浏览器窗口，并正在这两个窗口中同时浏览。在这种情况下，如果用户状态被保持在一个SFSB中，那么它可能会遇到非法并发调用。结果将会是某些调用将失败。

这个问题不是SFSB所特有的。不管我们在Web应用中如何处理用户状态，我们都将需要解决它。不过，这表明即使在我们使用SFSB时，我们仍将需要亲自解决一些棘手的问题（在本例中，或许通过Web层代码中的同步来解决）。

使用SLSB实现有状态功能度的模式

由于SLSB提供更好的可缩放性，并且在聚类中更坚固，所以调整设计来尽可能地使用SLSB取代SFSB是一个很不错的主意。下面就让我们来看一看几种调整方法。

对象参数

模仿有状态行为的常见方法是根据需要随同每个方法调用一起从客户中把状态传递给SLSB。通常，我们不是传递许多参数，而是只传递一个单独的大对象。在Web应用中，状态将被保持在Web层的HttpSession对象中。随同每个SLSB调用一起传递状态对象，在集中式应用中通常不是问题，但在分布式应用中可能会影响性能。

如果状态的容量非常大，我们可能需要随同每个方法调用一起传递大量状态。在这样的情况下，我们可能需要重新检查设计。通常情况下，保持容量过大的用户状态预示着很差的设计，而很差的设计应该被质疑。正如我们在第6章中已经见过的，无论我们怎样着手实现我们的应用，为每个用户都保持大容量的状态将会限制该应用的可缩放性。

每个用户都需要大容量的状态可能意味着我们需要使用一个SFSB。不过，如果我们可以在一个单独的SLSB方法调用中实现每个用例，那么系统仍将可能是比较可缩放的，如果我们随同每个方法调用一起传递状态的话。通常情况下，状态是有限的（例如，驱动一次数据库查找的主键）。

使用一个“必需的工作流程异常”来模仿一个SFSB状态机

只需要使用一点点创造力，我们有时就能让无状态会话组件干有状态会话组件的工作。下列示例说明了如何使用无状态会话组件来管理客户端工作流程。

笔者曾经不得不为一个Web门户设计一个强行要求新的用户名、密码和配置文件的注册系统。大多数现有用户可以直接在这个新系统上登录，并被标记为活动的。在这个新系统上登录为非法的其他用户必须先执行一个或一个以上的步骤来更新他们的账户，然后才能成功登录。例如，有些用户可能会被要求在登录之前先修改无效的用户名，并在分开的操作中提供附加的配置文件信息。这个迁移过程需要在EJB层中得到管理，因为其他客户类型有可能被添加到初始Web应用。

笔者的初始设计使用了一个有状态会话组件（实质上是一个状态机）来描述登录过程。每个用户的状态主要含有他们在此过程中已到达的点。令人遗憾的是，该应用必须运行在一个地理分布很散的聚类中，并且SFSB复制不是一个很有前途的选择。

笔者被迫使用无状态会话组件设计了一个系统。通过使用异常作为备选的返回值，笔者成功地从EJB层中控制了客户端工作流程。该Web应用或EJB层中根本不需要状态，而且该应用已经证明具有非常好的缩放性。

登录工作流程的起点是调用UserManager SLSB上的下列方法的那个客户（一个通过业务委派来访问UserManager SLSB的Web层控制器）：

```
UserProperties login(String login, String password)
    throws NoSuchUserException, RequiredWorkflowException, RemoteException;
```

一个成功的返回意味着登录已经完成，并且该客户拥有了有效的用户数据。该客户必须捕捉那两个应用异常。NoSuchUserException意味着登录过程结束，该用户的凭证已被拒绝。如果遇到了一个RequiredWorkflowException，那么该客户已检查了该异常中的一个代码。例如，在重复登录的情况下，该代码是常数值MUST_IDENTIFY_FURTHER。在这种情况下，控制器给用户转发一个表单，提醒他们输入他们的电子邮件地址。当这个表单被提交时，客户发布另一个登录请求给无状态UserManager上的下列方法：

```
void identifyUserFromEmail(String login, String password, String email),
    throws NoSuchUserException, RequiredWorkflowException, RemoteException;
```

这个方法也抛出了一个NoSuchUserException（如果该用户不能被识别出来的话），或一个带有这样一段代码的RequiredWorkflowException：该代码指出用户是否应该必须修改他们的用户名和密码。一个或另一个修改始终是必不可少的，但由UserManager来实现决定哪个修改必须被进行的规则。

这个方法成功地把账户迁移的细节隐藏在了EJB层内的一个无状态会话组件中。状态被有效地保持在客户的浏览器中——每…表单提交的URL和附加的数据促使了正确EJB方法的调用。

使用一个有状态会话组件作为控制器

普遍接受的一个观点是，Model View Controller (MVC) 体系结构模式最适用于Web接口和许多其他GUI。在典型的Web应用体系结构中，控制器构件是一个服务器小程序，或者是一个MVC框架特有的类。

对于使用了EJB的Web或其他GUI应用（由Sun JavaCenter提倡）来说，一个似乎合理的替代实现是把控制器移到EJB容器内作为一个有状态会话组件。这把控制器代码收集在EJB层中（尽管部分代码将不可避免地仍留在UI层中）。这意味着同一个控制器可以供不同客户端类型使用，如果该SFSB暴露了一个远程接口。

笔者认为，这种方法举例说明了大量J2EE文献所希望的思想。它应该管用。如果它的确管用，世界将会变成一个更美好的地方，但由于它的性能影响，笔者还没有听说有任何一个采用了这种方法的真正且频繁使用的应用（采用了这种方法的Java Pet Store表现得非常差）。

性能问题是J2EE设计权衡中的一个重要考虑因素。为了给每个客户请求提供服务，最大限度地降低方法调用深入J2EE体系结构的层深度是十分重要的。如果我们迫使每个请求都深入到EJB层中，性能和可缩放性都将会受到严重影响。

只有在考虑了合理的备选方案之后，才应当使用有状态会话组件。通常情况下，暗示使用有状态会话组件的征兆有：(a) 试图解决无状态接口方面的问题会导致过度的复杂性，以及(b) 必需的状态不能被保持在Web层内的一个HttpSession对象中（也可能是因为该应用根本就没有Web层）。

有状态会话组件(SFSB)的价值在EJB专家当中争论得非常激烈。许多专家和笔者一样对有状态会话组件持怀疑态度。例如，要想了解由Tyler Jewell(一位Principal Technology Evangelist for BEA Systems)所撰写的一篇批评有状态会话组件的辩论文章，请参见<http://www.onjava.com/pub/a/onjava/2001/10/02/ejb.html>站点。相反，Sun Java Center设计师们赞成使用有状态会话组件。

可应用于会话组件的J2EE设计模式

现在，让我们来看一看两个可应用于会话组件的J2EE设计模式：Session Facade和EJB Command。Session Facade设计模式对使用了实体组件的应用来说是一个已经过实践验证的主要成分，而EJB Command设计模式提供了一个有趣的、有时适用的替代方案。

这两个设计模式都与分布式应用有莫大的关系。它们一般都是使用无状态会话组件来实现。

分布式J2EE应用中的Session Facade模式

在分布式J2EE应用中，远程客户应该通过会话组件排他地与EJB层进行通信，而不管应用中是否使用了实体组件。在一个分布式应用中，会话组件将实现该应用的用例，处理事务管理，以及调解对实体组件、其他数据存取构件、助手对象等低级构件的访问。这种方法就叫做Session Facade设计模式，它与直接让客户使用实体组件相比，能够明显改善性能。

使用会话门面产生的性能增益源自于实现一个给定用例所需要的高代价网络往返旅程次数的降低。这样的网络往返旅程由远程方法调用和客户端JNDI查找所引起。因此，即使在客户确实需要与实体组件之类的基础构件打交道时，增加一个额外的会话门面层也能改善性能。这些会话组件能够在EJB容器内与低级构件有效地进行通信。

下面来看一个将要访问3个实体组件的用例实现。如果客户打算实现这个用例，它将可能因远程对象而需要3个JNDI查找，并将需要用3个远程调用中最小的一个针对必要的实体做调用。该客户还需要执行事务管理，以保证所有3个调用都在同一个事务中进行（通常是预先设计好的行为）。另一方面，使用一个会话门面，一个客户将只需要一次远程JNDI查找和一次远程调用。这将会产生一个很大的性能增益。

远程方法调用常常比本地调用慢数百倍。按照定义，一个客户必须至少进行一次远程调用来访问任一EJB方法（通常必须进行两次远程调用，因为在调用一个业务方法之前，调用EJB主接口来获得一个EJB参考将是必不可少的）。我们利用每个远程调用能够完成的事情越多，我们的分布式应用的执行效率就会越高，即使这意味着一个远程调用中所交换的数据量也越大。通过援用一个会话门面来实现我们的用例，我们一般可以避免这个权衡，因为业务逻辑所需要的大部分数据从不离开服务器。例如，方法工作期间被创建的对象可能从不被暴露给客户，并且从不通过通信线路来传递。

最后，如果远程客户能够直接援用像实体组件那样的低级构件，它们就与EJB层的体系结构紧密耦合在一起。这使得EJB层内的再加工变得不必要的困难。例如，用另一个持久性策略替换实体组件是不可能的。

在集中式应用中，这些性能考虑因素不适用。接口粒度不再是设计EJB接口时的一个决定性因素，但对是否向EJB容器外部的代码隐藏了EJB层的实现仍存在着争论——例如它是否使用实体组件。

EJB Command设计模式

Session Facade设计模式的一个替代品是EJB Command设计模式——GoF Command设计模式的一个特例。这个设计模式在IBM的San Francisco业务框架（先于EJB）中得到频繁使用。它特别适合用在分布式应用中，但在集中式应用也很有用。

实现EJB Command设计模式

这个设计模式是回调方法的一种面向对象形式（我们已在第4章中讨论过回调方法，并在第9章内所讨论的JDBC抽象框架中使用了它们）。

应用功能度被封装在可串行化的command（命令）对象中。在分布式应用中，命令由远程客户创建，并通过通信线路被传递给EJB容器。一般说来，命令是带有输入和输出属性的Java组件。所有命令实现一个共用接口，其中包括一个execute()方法。例如：

```
public interface Command extends java.io.Serializable {  
    void execute() throws CommandException;  
}
```

命令在客户上被构造，客户设置命令的输入属性（例如，HTTP请求参数可能会被绑定到命令的组件属性上）。然后，命令被传递给EJB服务器；在那里，命令的execute()方法由一个普通的无状态会话组件来调用。因此，execute()方法中的代码能够通过JNDI环境访问EJB的运行时环境（比如访问EJB或资源管理器）。execute()方法要么抛出一个异常（该异常将被返回给客户），要么设置命令对象的输出属性。然后，命令对象被返回给客户（客户将使用那些输出属性的值）。

需要注意的是，由于Command execute()方法的签名是固定不变的，所以命令实现可能只抛出一个已检查的命令异常（CommandException），以约束它们的方法签名。一般情况下，它们将抛出一个其内嵌套了另一个异常的普通命令异常，或者抛出CommandException的一个自定义子类。

下面，我们重点来看一看如何将一个SLSB用做命令执行器，但是这个命令模式不局限于EJB的使用。首先必须有一个CommandExecutor接口，它可以由SFSB来实现。典型的CommandExecutor接口可能只有一个方法：

```
public interface CommandExecutor {  
    Command executeCommand(Command command)  
        throws RemoteException, CommandException;  
}
```

SFSB远程接口可能会扩展这个方法（executeCommand()方法必须抛出java.rmi.RemoteException来允许这么做）。

下面是一个简单命令执行器SLSB的完整实现：

```

import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class CommandExecutorEJB implements SessionBean, CommandExecutor {
    private SessionContext sessionContext;

    public void setSessionContext(SessionContext sessionContext)
        throws EJBException, RemoteException {
        this.sessionContext = sessionContext;
    }

    public void ejbRemove() throws EJBException, RemoteException {
    }

    public void ejbActivate() throws EJBException, RemoteException {
    }

    public void ejbPassivate() throws EJBException, RemoteException {
    }

    public void ejbCreate() {
    }
}

```

`executeCommand()`方法的实现是完全通用的，因为该命令本身含有全部应用特有的代码。需要注意的是，如果执行该命令时遇到一个异常，`CommandExecutorEJB`保证当前事务被回退。这个命令执行器会话组件使用了CMT，所以`executeCommand()`方法运行在它自己的事务中：

```

public Command executeCommand(Command command) throws CommandException {
    try {
        command.execute();
    } catch (CommandException ex) {
        sessionContext.setRollbackOnly();
        throw ex;
    }
    return command;
}

```

一个典型的命令对象看起来可能如下所示：

```

public class CustomerDataCommand implements Command {

    private int custid;
    private String name;
    private int invoices;

    public void setCustomerId(int id) {
        this.custid = id;
    }

    public void execute() throws CommandException {
        try {
            Context ctx = new InitialContext();
            SalesHome home = (SalesHome) ctx.lookup("java:comp/env/ejb/sales");
            Sales sales = home.create();
            Customer cust = sales.findCustomer(this.custid);
            this.name = cust.getForename();
        }
    }
}

```

```

        this.invoices = <code to count invoices omitted>
    } catch (Exception ex) {
        throw new CommandException("Failed to execute command in EJB container",
                                   ex);
    }
}

public String getName() {
    return name;
}

public int getInvoices() {
    return invoices;
}
}

```

该命令拥有一个输入属性（CustomerId）和两个输出属性（name和invoices）。突出显示的execute()方法运行在EJB容器内，它在那里能够连接到另一个EJB来查找这些数据。因此，一个完整的用例（查找客户数据）在单一的网络往返旅程和单一的事务中执行。

请注意，一个真正的命令可能会使用Service Locator设计模式（在下一章中讨论）来减少查找它所参考的那些EJB所需要的代码量。

下列代码段描述了我们怎么才能利用一个指向某个CommandExecutor EJB的参考从客户端执行这个命令。需要注意的是，tc变量（保存要发送给EJB层的命令）被设置成了在该命令执行成功时从该EJB中返回的命令。这保证了客户代码能够访问命令的输出属性，比如name和invoices属性：

```

TestCommand tc = new TestCommand();
tc.setCustomerId(customerId);
try {
    tc = (TestCommand) commandExecutor.executeCommand(tc);
    System.out.println("Customer name is " + tc.getName());
    System.out.println("Customer has " + tc.getInvoices() + " invoices");
} catch (CommandException ex) {
} catch (RemoteException ex) {
}

```

这段代码可以被保存在一个客户端命令执行器里，并且该执行器向其他客户端代码隐藏了EJB访问的细节。

EJB Command设计模式的优缺点

EJB Command设计模式具有以下优点：

- 它提供GoF Command设计模式的常见好处，允许命令的排队和记日志。
- 它使一个完整用例能够在单一往返旅程中执行，因而最小化网络开销。
- 它使EJB层变得可扩展（无需修改EJB）。
- 它允许从EJB层返回多个返回值。
- 它具有相当强的对抗分布式故障的坚固性。
- 它不束缚于一个EJB实现。客户与命令对象打交道，而不是与EJB打交道。在上述示

例如，CommandExecutor接口可以在不使用EJB的情况下实现。

EJB Command设计模式的缺点不是那么明显：

- 我们需要把全部命令类包含在EJB容器中，所以每当我们增加一个新命令时，都需要重新部署该命令执行器EJB。让EJB容器把类“输出”给客户是可能的，但是由于安全和其他原因，把另外的类“输入”到EJB容器中是不可能的。
- 它使得客户依赖于EJB层的库。例如，如果有些命令使用了JDO，客户就需要访问JDO库。
- 有时，效率可能会要求去往EJB服务器的东西不应该再返回来。使用像上面所述设计模式一样的命令设计模式的一个典型实现，输入数据随同响应结果一起被多余地返回给客户。
- 笨拙的错误处理。所有异常都必须包装在一个通用的CommandException中。客户代码必须根据需要抽取进一步的细节。
- 虽然会话组件通常能高速缓存资源和数据，但命令却不能，因为新的命令经常被创建，而且所有命令数据都必须被串行化和反串行化两次。例如，我们将需要随同每个execute()调用一起做一个JNDI查找，以获得已经得到高速缓存的那些EJB参考（如果该操作被实现在一个正常SLSB中）。不过，JNDI查找在EJB容器中应该速度很快。
- 命令只能通过抛出一个CommandException来回退当前事务，因为命令无权访问命令执行器EJB的SessionContext。虽然把SessionContext作为一个参数传递给execute()方法是可能的，但是这会增大客户对一个EJB实现的依赖性。
- EJB Command设计模式引进了代码重复的危险。由于使用EJB Command模式往往会影响团队的工作方式，不必再加工共用代码，开发人员就能更轻松地实现相同的功能度。虽然封闭的团队工作可以帮助最大限度地降低这种风险，但是也存在EJB Command模式使共享共用代码变得困难的严重危险。

由于这些缺点，笔者往往不使用EJB Command设计模式。不过，它是一个有发展前途的选择，并且值得讨论，因为它展示了一种保证一个完整用例只在分布式应用内的单个远程调用中执行的方法。

在不采用EJB Command设计模式的情况下使用命令

把一个针对EJB层（或任一业务接口）的请求封装为一个单独的对象而不是个别参数的构思，可以在不必把可执行代码从客户传输到服务器的情况下使用。

我们可以简单地在我们的业务接口上提供一个方法来处理每个命令（这些接口可以使用EJB来实现，也可以不使用EJB来实现）。在这种方法中，命令不含有必需的业务逻辑，只含有一个针对该应用的请求。在这种方法中，我们不总是需要返回命令（我们可以返回一个截然不同的输出对象），并且这样的业务方法可以抛出我们所选择的异常。

这种方法（我们将在示例应用中使用它）通过形式化Web层代码的任务，即把用户示意动作转换成命令，并把命令响应结果转换成要显示给用户的视图，提出了Web应用中的良好习惯。这种方法也使测试变得更方便。许多Web应用都使用这种方法，而且大多数MVC Web

应用框架的设计都鼓励这种方法。

下面的示例摘自本书的示例应用，它举例说明了上面所描述的这种方法。`com.wrox.expertj2ee.ticket.boxoffice.BoxOffice`接口上的部分方法采取了对象（命令）变元，而不是采取了多个基元或对象参数。例如，启动订票过程的`allocateSeats()`方法有下列签名：

```
public interface BoxOffice {  
    // Other methods omitted  
  
    Reservation allocateSeats(ReservationRequest request)  
        throws NotEnoughSeatsException, NoSuchPerformanceException,  
              InvalidSeatingRequestException;  
}
```

`ReservationRequest`参数是一个命令。它的属性由`com.wrox.expertj2ee.ticket.web.TicketController`（示例应用的Web层控制器类）所调用的基础结构代码从`HttpSession`参数值中自动填入。只使用单个对象而不是多个参数的用法，意味着`BoxOffice`接口不需要修改（如果需要随同每个预订请求一起传输更多的信息），并且在命令被发布或处理时排队和记录命令或者出版事件（使用`Observer`设计模式）将很容易。

`BoxOffice`接口（不是`ReservationRequest`命令对象）的这个实现含有处理该命令所必需的业务逻辑。

会话组件实现问题

在详细考虑了需要使用哪种类型的会话组件和经常与会话组件一起使用的一些正确习惯之后，现在让我们来看一看与会话组件有关的几个重要实现问题。在本节中，我们将考虑EJB错误处理模型和该模型对会话组件开发人员有怎样的影响，使用EJB CMT的事务传播和事务传播对实现会话组件的影响，以及一个EJB实现模式（它能够帮助我们避免引发部署错误的常见原因）。

EJB中的错误处理

由于EJB是管理式对象，所以EJB容器插手帮助处理EJB所抛出的部分类型的异常。本节将讨论EJB规范中所定义的规则，以及开发人员怎样才能使用这些规则来受益。当我们在会话组件中实现业务逻辑和定义会话组件的本地与远程接口时，这些考虑因素是十分重要的。

发生异常时EJB容器的行为

EJB规范处理EJB所抛出的异常的方法是简单而精致的。此规范区分了应用异常和其他所有异常（统称为系统异常）。

应用异常是指被定义在一个EJB的本地或远程主接口或本地或远程接口的一个方法的`throws`子句中的已检查异常，而不是指`java.remote.RemoteException`（请记住，Java RMI要求所有远程方法都必须在它们的`throws`子句中声明这个异常）。

系统异常是指由一个EJB实现类方法在运行时所抛出的未检查异常或可抛出异常，或由

该应用内的另一个EJB上的一个调用所产生的一个未捕捉到的java.remote.RemoteException（决定一个会话组件类中的一个方法是否捕捉由调用其他EJB所产生的远程异常决诸于会话组件开发人员，只有当该错误是可恢复的或给嵌套异常增添上下文信息时，这么做通常才有意义）。

我们假定EJB客户认识应用异常，并且知道怎样恢复这些异常。决定某个特定异常是不可恢复的决诸于客户。因此，在一个应用异常被抛出时，容器不插手处理，只是让EJB客户去捕捉这同一个应用异常。通常，容器将不记录应用异常，并且当前事务的状态将不会受到影响。

系统异常的处理有非常大的差别。容器假设客户没有预期到一个这样的异常，而且这种异常对当前用例有可能是致命的。这是一个合理的假设，与应用异常相反（它们是已检查的，并且客户必须知道它们，因为Java编译器要求客户去捕捉它们），系统异常对客户可能是毫无意义的。例如，就拿一个JDO持久性管理器所产生的一个运行时异常来说，客户不应该知道EJB层的持久性机制，不能指望客户纠正它所不知道的问题。

因此，容器采取猛烈的动作。它不可逆地将当前事务标记为回退。它废弃该EJB实例，以便它不能给更深的调用提供服务。EJB实例将被废弃的事实，意味着开发人员不需要花费任何精力来清理该会话组件内所维护的任何对话或内部状态。这减轻了开发人员的工作负担，并消除了隐错的一个潜在发源地。EJB容器必须记录这个令人不愉快的系统异常。如果客户是远程的，EJB容器就给该客户抛出一个java.remote.RemoteException（或一个子类）。如果客户是本地的，EJB容器就给该客户抛出一个javax.ejb.EJBException。

如果一个EJB遇到了一个不可恢复的已检查异常，并且又无法重新抛出这个已检查异常，它应该抛出一个包装了这个已检查异常的javax.ejb.EJBException或子类。这个异常将被容器当做一个意外异常来对待。出现意外异常时的这种容器行为不应该用来代替明确地回退使用了setRollBackOnly()方法的事务。这种容器行为是容器在一个组件实例遇到一个不可恢复错误时所提供的一个便利清理工具。由于这种容器行为会导致组件实例被废弃，因此，如果这种情况在运行时经常发生，它将会降低性能。

下面，让我们来看一看在设计和实现会话组件时的一些影响。

- 在一个正常操作中，我们绝不能允许一个EJB方法抛出一个运行时异常

一个未捕捉到的可抛出异常不仅会使调用栈像往常一样反绕，而且也会使容器回退当前事务。

- 我们应该考虑是否捕捉我们在EJB方法中使用了其远程接口的EJB所抛出的RemoteExceptions

我们的确拥有声明EJB实现来抛出RemoteException的较简单选择，但是这将被作为致命错误来对待，而且组件实例将被废弃。如果该异常是可恢复的，它必须被捕捉。

- 如果一个运行时异常是致命的，我们不必捕捉它，但可以让EJB容器去处理它

这方面的一个上佳例子是由一个JDO持久性管理器所产生的一个运行时异常。除非我们相信自己能够重试这个操作，否则捕捉这样的异常毫无用处。

- 如果一个EJB方法捕捉到了它不能纠正的一个已检查异常，它应该抛出一个包装了原异常的javax.ejb.EJBException

- EJB本地或远程接口上的应用异常必须是已检查异常

容器把所有运行时异常都作为致命的系统异常来对待。这就意味着我们在EJB方法的签名上还是在可能由EJB实现的任何接口上使用未检查异常没有选择权。这可能也是错误处理的这种EJB方法的惟一不幸的结果，因为客户代码对只处理不寻常情形中的运行时异常没有选择的余地——第4章中已讨论过的、能够证明非常有用的一种方法。

保证EJB容器将插手帮助处理未检查异常和确保事务回退的承诺，使得把可能由EJB使用的助手类所抛出的致命异常定义成未检查异常特别有吸引力。例如，我们在第9章中所讨论的那个JDBC抽象框架就抛出未检查的数据存取异常（DataAccessException），而使用了该抽象框架的EJB在遇到一个不可恢复的java.sql.SQLException时，可以简单地忽略这个数据存取异常。即便一个EJB需要截取这个异常，它仍可以实现一个catch块。

了解EJB API异常

java.ejb包定义了11个异常。J2EE开发人员最感兴趣的是下面介绍的这些异常。

- **javax.ejb.EJBException**

这是为包装不可恢复的已检查异常而设计的一个便利的运行时包装器异常。我们可能会使用它包装像JDBC异常那样的、必须捕捉的致命异常。不过，存在这样一种重要的情况：我们自己定义的、必须包装在一个EJBException中且重新抛出的一个异常，应该只是一个未检查（运行时）异常。这将简化EJB代码。由于抛出EJBException的catch块不必执行任何清理（EJB容器将回退该事务并废弃该组件实例），因此捕捉和重新抛出该异常没有增添任何价值。

下面是标准的EJB应用异常。它们被报告给客户。

- **javax.ejb.CreateException和子类**

这种异常应该从一个ejbCreate()方法中被抛出来，如果创建该组件的失败应该被视做一个应用错误。例如，一个有状态会话组件ejbCreate()方法可能会抛出这个异常，如果一个客户给它传递了无效变元。如果创建一个组件的失败反映了一个系统异常，就应该使用一个运行时异常（必要时使用EJBException包装器异常）。例如，告诉远程客户一个表在关系数据库中不存在是没有用处的。远程客户不应该知道EJB层用户使用了关系数据库。

- **javax.ejb.RemoveException和子类**

类似于CreateException。

- **javax.ejb.FinderException和子类**（只对实体组件有效，但时常被会话组件实现所遇到）

发现者异常必须被包含在实体组件的主接口上的每个发现者方法的throws子句中。在未能查找到一个被请求的对象时，或在遇到来自持久性存储器的不一致结果时（例如，如果同一个对象发现者查找到多个匹配的记录），一个发现者应该抛出这个异常。需要注意的是，从一个多对象发现者中返回的一个空Collection并不预示着一个错误。就带有CMP的实体组件来说，发现者方法将由EJB容器来生成。

用于使用CMT的EJB的事务属性

EJB通常都是事务性构件。正如我们在第6章中已经见过的，对使用会话EJB来实现业务逻辑来说，最激烈的争论之一就是使用声明性CMT（容器管理式事务）的选择。我们在EJB部署描述符中指定CMT行为，进而使EJB容器根据需要为EJB方法调用创建事务，免除开发人员直接实现JTA的需要。这是一个重要的工作效率增益。不过，彻底了解CMT究竟怎样工作是很重要的。

创建一个事务关系到一定的开销，所以我们不希望多余地创建事务。然而，对与事务性资源的多个更新有关的业务逻辑的完整性来说，事务是至关重要的。

为了集中精力考虑带有CMT的会话组件，现在让我们来看一看部署描述符所允许的事务实现，以及它们对代码的行为有怎样的影响（正如我们在第6章中已经见过的，如果我们可以选择的话，应该总是首选CMT，而不是BMT）。EJB规范认可了6个事务属性，这些属性都与ejb-jar.xml部署描述符中的EJB方法有关联。表10.1总结了每个属性的商定行为。

最安全的EJB事务属性是Required（用于会话组件）和Mandatory（用于实体组件）。

这两个属性保证预期的事务行为，而不管调用者的事务上下文。**NotSupported**事务属性可以用来改善只读方法的性能，因为对这些方法来说，事务不是必需的。这种情况时常出现，因为Web应用经常使用只读数据。

EJB规范建议，与EJB开发人员截然不同，“应用装配人员”可能会设置事务属性。这可能是危险的。正如我们已经见过的，更改一个EJB方法的事务属性会改变该方法的语义。只有当组件开发人员定义了事务属性的时候，他们才能开发出对EJB有意义的单元测试。虽然EJB事务属性的指定从Java代码中消失正是人们所希望看到的，但是把它们从组件开发人员的控制中除去却是误入歧途。

虽然事务管理的细节最好留给EJB容器，但是决定一个事务该不该被回退通常是一个业务逻辑问题（请记住，如果有一个运行时异常，EJB容器将自动确保回退，无论一个组件使用BMT还是CMT）。因此，在一个使用了CMT的组件中确保回退是很容易的。

所有EJB都可以使用EJB上下文接口的一个实例（由容器在运行时供给）。javax.ejb.EJBContext有3个子接口，而这些子接口可以在运行时让3种类型的EJB使用：javax.ejb.SessionContext、javax.ejb.EntityContext和javax.ejb.MessageDrivenContext。EJBContext超级接口提供了一个setRollbackOnly()方法，而该方法允许CMT组件将当前事务不可逆地标记为回退。

业务方法接口“模式”

由于EJB容器（而非组件开发人员）提供EJB的构件接口（本地或远程接口）的实现，所以我们怎么保证构件接口和组件实现类仍然同步呢？缺少这种同步是引起部署时问题的一个常见原因，因此这是一个重要的问题，除非我们准备依靠为EJB开发而提供的工具支持。

我们可以选择让组件实现类来实现构件接口，但这具有下列缺点：

- 由于构件接口必须扩展javax.ejb.EJBObject或javax.ejb.EJBLocalObject，所以组件实现必须为来自这些接口的、而且从不被容器调用的方法提供空的实现。读者对此可能混淆不清。

表10.1

部署描述符中所指定的事务属性	对标记方法（称做方法m）和其他资源m调用的含意	当调用者已有一个事务时的结果	说明	推荐的用法
Required	该方法被保证运行在一个JTA事务中。如果调用者已有一个事务，它将被使用。如果没有，一个新的事务将被创建	方法m将运行在调用者的事务中	如果调用者的事务被回退，方法m的工作将被自动回退。这是最灵活的选择。它允许调用者使用方法m作为一套较大的可逆工作的一部分。但是，方法m仍可以独立使用。例如，它能够支持从EJB层外部的客户所做的调用	这应该是会话组件方法的默认事务属性，而这些方法定义了EJB层的公共接口。
RequiresNew	该方法被保证运行在一个新的事务中，无论调用者是否已有一个事务	调用者的事务在该方法的执行期间被悬挂起来	这会是危险的。如果方法m成功并提交了更改结果，但调用者的事务被回退了，那么方法m所做的那些更改将不被翻转，这意味着调用者对业务操作没有全部控制权。	仅在这样一种特殊情况下有用：方法m从实现工作中必须提交的一个不同部分，而不管进行中的任何一个高级操作是否成功。这方面的例子可能是一个审计动作，其中一条持久性记录必须舍弃。有一个事实：一个用户尝试了这样一个动作，无论这个动作是否成功

(续表)

部署描述符中所指定的事务属性	对标记方法（称做方法m）和其他资源m调用的含意	当调用者已有一个事务时的结果	说明	推荐的用法
NotSupported	该方法被认为对JTA事务没有兴趣，并且将在没有事务上下文的情况下执行。如果该方法调用其他的EJB或资源管理器，该事务上下文被传递。	调用者的事务在该方法的执行期间被悬挂起来。该属性对实体组件无效。	调用者的事务的回退不影响方法m所做的任何更改。该属性对实体组件无效。	如果该方法的实现是作事务性的，适用。 如果该方法更新持久性数据，不要使用。 对只读方法将有用。例如，Oracle SELECT不必运行在一个事务中，如果不存在事务，Oracle将不创建事务。因此，就NotSupported事务属性来说，查询方法将是有效的。
Supports	如果调用者有一个事务，它将被传送给该方法，并且执行向Required情况下的执行一样。如果调用者没有事务，容器将不创建事务，并且该方法将在没有事务的情况下执行。[i]Not-Supported情况中的执行一样	和Required情况下一样，方法m将运行在调用者的事务上。	有潜在的危险。方法m的语义将是变化的，视客户是否有事务上下文而定。 该属性对实体组件无效。	不要把这个事务属性用于更新。 在只读操作的情况下，该属性可以被安全地使用，并且执行将向Not-Supported情况中的执行一样。

(续表)

部署描述符中所指定的事务属性	对标记方法（称做方法m）和其他资源m调用的含意	当调用者已有一个事务时的结果	说明	推荐的用法
Mandatory	该方法需要一个JTA事务，而该事务必须由客户提供。如果客户在调用该方法时没有事务上下文，容器将抛出一个javax.transaction.TransactionRequiredException	这是惟一有效的情况	通常是Requires的一个替代属性。限制该方法可能会被调用的各种情况。不过，这可能是件好事。	当使用一个会话界面时，对实体组件方法有用。在这种情况下，我们希望客户通过该界面进行工作，而不是操纵实体组件那样的低级构件，因为操纵低级构件可能会深入到业务逻辑的监控区，并破坏系统完整性。
Never	Mandatory的对立属性；该方法不能被只有事务上下文的客户调用。如果它被这样的客户调用，容器就抛出一个java.rmi.RemoteException。在该方法内，执行将同NotSupported情况下的执行一样	禁止使用	NotSupported的一个替代属性，如果认为阻止具有事务上下文的客户调用该方法是必不可少的。	该属性对实体组件无效

- 返回this给远程或本地客户来代替构件接口的容器生成实现将是可能的。这是EJB容器可能检测不到而且在运行时可能会引起微妙问题的一个严重的程序设计错误。

幸运的是，有一个简单、有效而又得到广泛认可的解决方法：让构件接口和组件实现类都实现一个共用的“业务方法”接口。这个业务方法接口不扩展任何EJB基础结构接口，尽管它可以扩展其他应用业务接口。

如果我们正在处理一个EJB，并且它使用了一个远程接口，业务方法接口中的每个方法都必须被声明成抛出javax.remote.RemoteException。对使用了远程接口的EJB来说，业务方法接口平常是一个EJB层实现细节：客户将与该组件的远程接口打交道，而不是与业务方法接口打交道。

对于使用了本地接口的EJB来说，业务方法接口的方法可能不抛出RemoteException，这意味着业务方法接口不必是EJB特有的。通常，业务方法接口（而非EJB的本地接口）对EJB的客户是最有意义的。例如，在示例应用中，com.wrox.expertj2ee.ticket.boxoffice.BoxOffice接口就是一个普通Java接口，该接口在不使用EJB的情况下也是可以实现的。客户调用这个接口上的方法，而不是调用扩展它的EJB本地接口上的方法。

“业务方法接口（Business Methods Interface）”的使用其实并不是设计模式——它经常是EJB规范的一个粗糙特性的一种迂回解决方法。但是，由于人们经常把它称做一个模式，所以笔者将遵循公认的术语。

下面让我们来看一看怎样把这个模式用于示例应用的BoxOffice EJB，以及要使用的命名约定。为该EJB使用这个模式涉及到下列4个类：

- **BoxOffice**

定义该EJB上的那些业务方法的接口，该接口由构件接口来扩展，并由组件实现来实现。在示例应用中，这个接口与EJB实现被分开定义，并由客户使用。即使客户将只与该EJB的本地或远程接口打交道，一个这样的接口也完全可以作为一个EJB层实现选择被引进，以同步实现该组件所需要的那些类。

- **BoxOfficeLocal**

EJB构件接口。该构件接口应该是空的。它将既扩展业务方法接口，又扩展javax.ejb.EJBObject（在一个远程接口的情况下）或EJBLocalObject（在一个本地接口的情况下）中的一个或另一个对象，继承它的所有方法。在示例应用中，我们使用一个本地接口。需要注意的是，笔者已经在业务接口名称的后面后缀了Local；当然，给远程接口应该使用一个Remote后缀。

- **BoxOfficeHome**

该EJB的主接口。除非一个EJB既有本地接口，又有远程接口（一种相当少见的情况），否则笔者认为没有必要区分BoxOfficeLocalHome与BoxOfficeRemote Home。

- **BoxOfficeEJB**

组件实现类。这将既实现javax.ejb.SessionBean，又实现业务方法接口。有些源推荐BoxOfficeBean形式的名称，但笔者避免这么做，因为笔者为了与EJB形成对比，往往为一个特定接口的、是普通JavaBean的实现使用“Bean”后缀。如果组件实现束缚于一项特定的技术，名称中就会暗示这一点，比如JdbcBoxOfficeEJB。同一个接口的

实现一样，名称应该暗示该实现有什么特色，而不是只尽可能相近地重复接口名称。

图10.1所示的UML类图描绘了这些类与接口之间的关系，以及EJB规范所要求的那些类和接口。类图中用圆环圈起了业务方法接口。需要注意的是，组件实现类BoxOfficeEJB扩展了一个通用超类com.interface21.ejb.support.AbstractStatelessSessionBean，以帮助满足EJB规范的要求。我们将在下一章中介绍应用基础结构的时候再讨论这方面的内容。

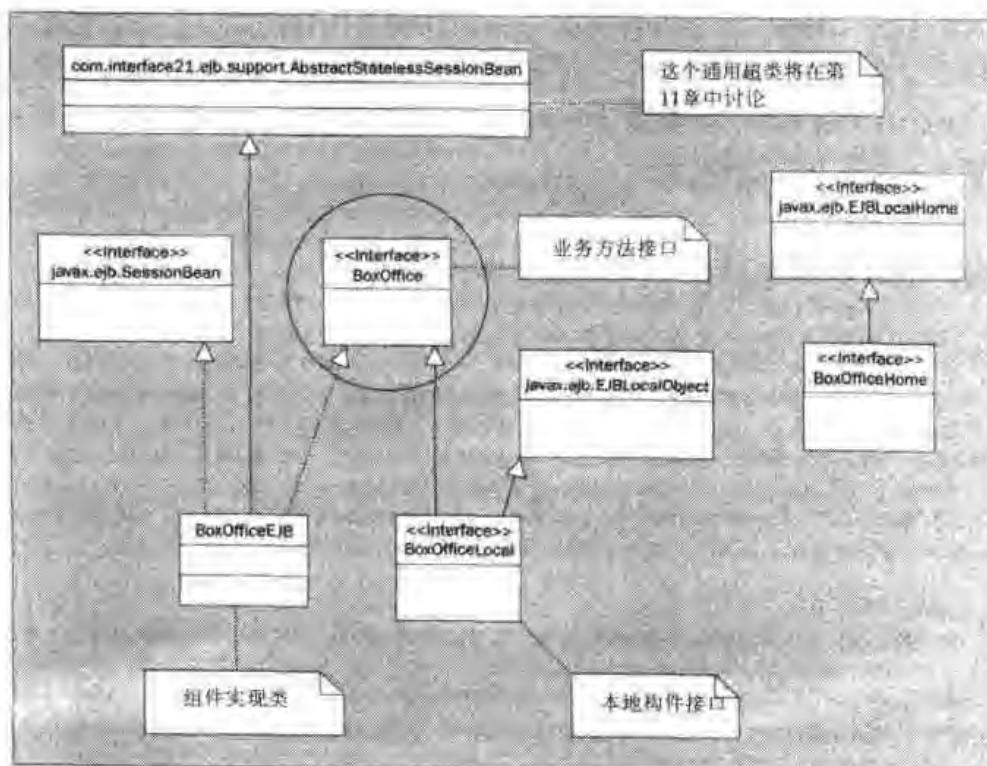


图10.1

下面是BoxOffice EJB的本地接口的完整清单，该接口没有定义任何新的方法：

```

public interface BoxOfficeLocal extends javax.ejb.EJBObject,
    com.wrox专家组.ticket.boxoffice.BoxOffice {
}
  
```

下列清除了展现了EJB规范所要求的`javax.ejb.SessionBean`接口，还展现了该组件的实现类是怎样实现业务接口的：

```

public class BoxOfficeEJB extends AbstractStatelessSessionBean
    implements SessionBean, BoxOffice {
    // Implementation omitted
}
  
```

虽然EJB Business Methods Interface得到了广泛的拥护，但不是每个人都赞同命名约定。有些源主张为业务方法接口的名称使用一个BusinessMethods后缀。无论决定采用何种命名约定，为了避免混淆不清，请在所有EJB实现中一致地使用选择的命名约定。

Business Methods Interface模式是一项能够节约大量时间的简单技术。不过，它无法发现EJB的主接口与实现类之间的同步问题。例如，主接口方法create()必须对应于ejbCreate()方法。主接口与组件实现类之间的同步问题可能会在有些服务器中产生神秘的“抽象方法错误”。下一章中所讨论的那些通用超类可以帮助避免此类错误，其中涉及到会话组件和消息驱动式组件。

要使用**Business Methods Interface**模式来帮助避免构件接口与业务方法之间的同步问题。错误在编译时就会被挑出来，而不是在那些组件被部署给一个EJB容器的时候。虽然使用一个IDE来同步这些类是可能的，但业务方法接口使用编译程序，因此可以保证在任何开发环境中都工作。

示例应用中的会话组件

示例应用要求我们为执行某些用例的用户保持服务器端状态。入场券预订情况在订票过程中必须得到保持；用户配置情况在订票过程中可能需要得到保持。不过，订票过程以外的页面不要求会话状态得到保持。

我们可以把这种会话状态保持在SFSB中。但是，优先选择Web容器状态管理有几个很好的理由。

- 使用SFSB将会使我们束缚于使用EJB。我们没有其他理由让我们的体系结构依赖于使用EJB；我们只是选择了在EJB简化事务管理的地方才使用它。
- 这是一个Web应用。没有必要为远程RMI客户保持会话状态。
- Web容器状态管理可能胜过SFSB状态管理。我们将不必调用SFSB来访问用户状态，而且在聚类中可能有一个降低了的状态复制开销。
- 将状态保持在Web容器中则更简单。我们可能需要使用HttpSession对象，不管我们是否使用了SFSB。如果使用SFSB，增加的JNDI查找复杂性将不会产生任何好处。

因此，我们使用HttpSession对象把示例应用的用户状态保持在Web容器中，并使用SLSB。由于所包含的会话状态量不是太大（我们曾经保证参考数据在用户间共享），以及我们正通过一个本地接口来调用我们的唯一EJB，所以随同每个方法调用一起把用户状态传递给EJB容器不会有任何问题。

我们将使用CMT，因为我们选择使用一个EJB来实现订票过程的主要原因是简化事务管理。

小结

本章介绍了会话组件设计和实现的一些主要问题。我们已经了解了如下内容：

- 在我们选择使用EJB时，无状态会话组件通常是实现业务逻辑的最佳方法。
- 有状态会话组件只应该在EJB层中没有有状态模型的替代方案时使用，因为它们比SLSB耗用EJB容器中的更多资源，而且不提供真正稳固的状态管理（特别是，应用服务器可能发现，在聚类中复制修改结果到SFSB比复制修改结果到HttpSession状态更困难）。我们已经见过了一些可能允许我们使用无状态会话组件来代替有状态会

话组件的技巧。

- Session Facade设计模式可以用来改善性能，简化实现，以及最大限度地提高设计灵活性，尤其是在分布式应用中。让远程客户访问EJB层通常是最佳方法。
- EJB Command设计模式是Session Facade设计模式的一个替代方案，其原理是把代码从客户移到服务器并在服务器上执行它。

我们还通过学习下列内容加强了对EJB容器服务的理解：

- EJB容器在EJB抛出异常时的行为。同普通Java方法调用一样，应用异常（Application exception）（不同于java.rmi.RemoteException的已检查异常）被抛给EJB客户，无需EJB容器的任何干预，而系统异常（System exception）（java.rmi. RemoteException和未检查异常）却致使EJB容器把当前事务标记为回退，并废弃引起麻烦的EJB实例。这种行为可以用来简化错误处理，意味着我们不必捕捉致命的运行时异常，可以完全让EJB容器去处理它们。这也意味着如果希望当前事务在万一发生了应用异常时得到回退，我们必须在应用代码中使用setRollbackOnly()方法，因为EJB容器将不采取任何动作。
- 用于EJB CMT的6个事务属性（Transaction attribute）的含义，以及它们对会话组件方法的行为有怎样的影响。

我们还探讨了“Business Methods Interface（业务方法接口）”设计模式——一种可以通过保证EJB的构件接口和实现类总是保持同步来防止错误的简单编程技巧。这涉及到让构件接口和组件实现类都实现一个“业务方法接口”，进而使编译程序能够核实方法签名是匹配的。

我们还讨论了下列良好的会话组件实现习惯：

- 使用会话组件来定界事务。
- 在分布式应用中，尽可能在远程接口上的每个方法中执行一个完整的用例。
- 当实现会话组件时，高速缓存JNDI查找的结果，比如EJB主参考和DataSource对象。
当实现SLSB时，为所有客户都高速缓存资源和数据。

下列建议适用于有状态会话组件：

- 如果在聚类环境中部署使用了SFSB的应用，请阅读应用服务器资料，特别关注与聚类中的有状态会话组件行为密切相关的部分。不同的服务器有不同的行为，该行为会影响性能，甚至影响使用SFSB的可行性（例如，在事务或每个方法调用结束之后，会话状态得到复制了吗？）
- 使事务保持简短，因为SFSB对话状态在事务期间可能是最脆弱的（尽管这将视EJB容器而定）。
- 不要使用javax.ejb.SessionSynchronization接口。要尽力在个别方法中管理事务，进而依靠EJB CMT。
- 不要忘了使实例数据在ejbPassivate()方法之后保留在一个合法状态中，并在ejbActivate()方法之后保留在一个可用状态中。
- 如果必要，同步客户代码来保护SFSB使之免于非法的并发调用。

在下一章中，我们将转而讨论总体的应用基础结构，包括一些通用的EJB超类，这些超类在实现会话组件和消息驱动式组件时能够简化开发和降低出错的可能性。

第11章 基础结构与应用实现

在本章中，我们将了解怎样在J2EE应用中实现业务逻辑。但是首先，我们将来看一看能使我们把应用代码限定于实现业务逻辑的基础结构。这是J2EE规范的一个关键目标，但令人遗憾的是，J2EE API的复杂性意味着这个目标实现起来将很困难。

在本章中，我们将看一看附加的基础结构怎么才能用来解决常见问题和简化J2EE的使用。我们将讨论下列内容：

- 一个强有力的应用基础结构应该实现的目标，以及应用应该使用一个基础结构为什么很重要。
- 随同示例应用一起包括的一些通用包的背后的动机和实现，这些包分成两类：
 - 一类包提供一种通过JavaBean实施应用配置的一致方法并使基于接口的设计更容易（参见第4章中的讨论）；
 - 一类包简化复杂的J2EE API（如EJB和JMS）的使用。

最后，我们将了解如何使用本章中所描述的通用基础结构来实现示例应用的一些关键功能度，进而阐明该基础结构是如何简化应用代码的，以及它是怎样使工作效率和可维护性都受益的。

基础结构

让一个应用拥有一个坚固的基础结构是非常重要的，该基础结构应该能使应用代码把精力都集中在解决业务问题上，不分心去处理“管道工程”。这就是为什么选择使用J2EE的原因，为什么在另外需要编写过分复杂的代码时可能会选择使用EJB的原因，以及充分的基础结构类为什么和应用服务器几乎同等重要的原因。

我们通常需要亲自构造部分基础结构，但是在有现成基础结构可供选用的地方，一般最好是使用现成的基础结构。

令人遗憾的是，目前还没有与能够简化J2EE使用的基础结构相关的标准。因此，不同的项目将会有不同的选择。基础结构的可能来源包括：

- 现有的自产基础结构。大多数的大型组织机构将会开发许多常用的包，尽管它们的质量会稍有不同。通常情况下，这样的基础结构没有一个连贯的策略，而且也失去了项目间重用的可能性。这是一个花费了很高代价但却又丢失的机会；在本章中，我们将会看到代码可以得到重用的许多方面，以及在项目间采用的许多一致习惯，这么做有明显的好处。
- MVC Web应用框架。有些产品（如Struts）除了提供一个MVC实现之外，还能够提供一个用于Web应用的框架。不过，MVC Web框架只打算（自然也足够）供Web应用中使用；因此，使一个应用的总体基础结构依赖于一个MVC Web框架可能会使测试变得很困难，并减低重用的可能性。

- 开放源应用框架。在这个领域中，还没有任何一个产品能够像领先的MVC Web应用框架一样有名或得到广泛采用，而且许多产品是非常复杂的。因此，笔者在此无法推荐一个这样的产品。
- 商业产品，比如随同应用服务器一起提供的框架，或承诺简化J2EE开发的框架。同样，这个领域中还没有一个产品已得到广泛使用。来自应用服务器供应商的赠品一般把用户限定在相关的应用服务器内，进而取消了应用服务器之间的可移植性。
- 本章中所描述的、解决了许多常见问题的基础结构代码。这个基础结构相当简单，并且广泛利用JavaBean来最大限度地消除应用代码对它的依赖性，以便尽可能地避免牵连太深。

在本章中，我们将主要通过分析示例应用中所使用的那些支持包来了解基础结构的问题。这个基础结构代码（基于笔者在几个实际项目中的成功经验）举例说明了一些主要问题，并提供了一个简单的解决方法来解决笔者所知在其他地方尚未得到很好解决的问题。在本章中得到讨论的所有类和那些支持类的源代码被包含在了本书的下载中，因此读者可以把这个源代码用做你的应用的一个基础。

对支持性基础结构的强调是笔者在本书中所用方法与大多数J2EE文献中所用方法之间的一个主要区别。大多数J2EE书籍更关注J2EE API，极少关注使用它们的现实性。因此，它们往往介绍简单而又不太干净的方法（例如，兼顾正确的错误处理），或者忽略生产效率和可维护性考虑因素。

在本书中，笔者正力图通过举例来阐明J2EE满足业务需求的用法，笔者的目标是使用J2EE满足业务需求，而不是使用J2EE本身。这就意味着作为一个意外收获，本书的配书下载包含了读者可以在自己的应用中直接使用的大量代码，或用做读者自己的基础代码的大量代码。

本章中所讨论的这种方法只是构造J2EE应用的诸多方法之一。不过，它背后所隐含的那些原理在几个实际项目中已经提供了高效率和高质量的解决方法。这个基础结构所解决和设法说明的问题与所有J2EE应用都有关系。

基础结构的目标

使用一个坚固的标准基础结构可以提供更好、速度更快的应用。一个坚固的基础结构通过达到下列目标使这一点成为了可能：

- 允许应用代码在尽可能的情况下集中精力实现业务逻辑和其他应用功能度。这通过减轻开发努力缩短了上市时间，并通过使应用代码更易维护降低了整个项目生命周期内的成本。这是一个最终目标，而且下面的许多目标都帮助我们达到这个最终目标。
- 把配置从Java代码中分离出来。
- 通过消除对常见折衷的需要，使OO设计的使用变得更容易。
- 通过只解决每个问题一次，删除代码重复。一旦我们有了一个问题（比如一个复杂的API）的有效解决方法，就应该始终使用这个解决方法，无论在遇到这个问题的构件中，还是在遇到这个问题的类中。

- 隐藏J2EE API的复杂性。我们已经在讨论JDBC时谈过这个目标；用于较高级别抽象的其他候选API包括JNDI和EJB访问。
- 保证正确的错误处理。我们在第9章中讨论JDBC时谈过这个目标的重要性。
- 如果必要，使国际化变得更便利。
- 在不损及体系结构原理的情况下改进生产效率。如果基础结构不够充分，势必要急功近利地采用将引起后续问题的简单而又粗糙的解决方法。充分的基础结构应该促进和方便正确设计原理的应用。
- 在一个机构内的应用之间应获得一致性。如果应用使用相同的基础结构，以及相同的应用服务器和底层技术，那么生产率将被最大化，团队工作将更有效率，而且风险将降低。
- 保证应用容易测试。在凡是可能的地方，一个框架应该使应用代码在无需部署在应用服务器上的情况下就能被测试。

我们将基本上使用标准基础结构来帮助实现第4章中所确定的那些目标。

基础结构不应该以Web为中心，即便我们正在开发一个Web应用，这一点是非常重要的。这会使应用的测试变得不必要的困难，并且会使应用依赖于某个特定的用户接口。因此，基础结构不应该受限于Struts之类的某个Web框架（尽管这样的Web框架是有价值的，如果使用正确的话）。只是从表示层产品（如JSP页面）中分离出业务逻辑是不够的：业务逻辑根本不应该在Web特有的类中。例如，一个Struts Action类束缚于Servlet API。这使Struts动作变成了业务逻辑的一个极差场所，业务逻辑不应该依赖于Servlet API。因此，尽管使用一个Web应用框架很重要，但Web层应该依然是位于一个不同业务逻辑层上面的一个薄层。

基础结构将经常采取框架的形式，而不是采取类库的形式。通常，控制反向对从应用代码中消除掉复杂性是非常重要的。我们已经在第9章中讨论JDBC时见过这种方法的好处。

一个框架应该是可扩展的。虽然一个设计完善的框架将会满足大多数需求，但除非它能够被扩展以满足未预计到的需求，否则它最终将会失败。在框架内使用接口将帮助实现这个目标。

一个框架应该是容易使用的。过分复杂的框架最终将会被开发人员所忽视，而且将会引发它们自身的许多问题。不相关或极少使用的能力是一个关键的危险。巴累托定律（Pareto Principle）¹⁹与框架设计特别有关。一般说来，一个解决大多数问题的简单解决方法，要比一个全面但使许多任务变得更困难的复杂解决方法更令人满意（可以证明的是，J2EE本身就属于后面这一类）。

现在，让我们来看一看示例应用中所使用的核心基础结构。同大多数成功的基础结构一样，这里所描述的这个基础结构是分层的。当使用它时，我们可以从不同的抽象级别中进行选择，其中每个级别建立在下一级的构件上面。

使用框架来配置应用构件

现在，让我们首先从解决外部化配置和简单化OO设计的使用这两个目标开始。

问题

在应用配置中采用一致的风格是十分重要的。虽然外部化配置（从Java代码中分离出它）

对参数化应用是必不可少的，但是在没有支持性基础结构的情况下，这种外部化实现起来是非常困难的。

如果没有这样的基础结构，配置管理通常是无规则的。J2EE应用常常以多种多样的格式保存配置，比如属性文件、XML文档（一个逐渐流行的选择）、EJB JAR与WAR开发描述符以及数据库表。在大型应用中，在怎样管理配置方面常常没有一致性。例如，属性命名约定（如果有的话）在不同的开发人员之间可能是不同的，对于不同的类可能也是不同的。

更严重的是，应用类将会因为与它们的业务任务不相干的配置管理代码而变得很大。许多应用对象可能需要读取属性文件或语法分析XML文档。即便它们使用助手类简化了底层API的使用，它们的业务目的仍是模糊不清的，并且它们依然束缚于最初设定的配置格式。

同样严重的是，如果没有一个用于配置管理的中心基础结构，应用代码很可能会使用多种查找应用对象位置的方法。Singleton设计模式常常被过度使用，进而造成我们下面将要讨论的那些问题。作为一种选择，有些应用对象可能会被束缚在JNDI上，或被附加到全局的ServletContext上（在Web应用中），这两种方法都可能会使JNDI服务器外的测试变复杂。每个应用对象通常被独立地配置。

通用而又可重用的基础结构应该用来集中化配置管理和以一种一致方式“捆绑”应用。

在本节中，我们将看一看示例应用所使用的基础结构怎样使用基于JavaBean的配置，确保整个应用中的一致性。

在本节中，我们将分析一个既集中化了配置管理又解决了上述那些问题的应用框架。

使用Java组件

我们已经在第4章中讨论过Java组件的各种好处。例如，基于Java组件的操作非常适用于应用被保持在Java代码外部的配置数据，也适用于做“数据构造”，比如用HTTP请求构造出Java对象状态。

如果把所有应用构件都变成JavaBean，我们就最大限度地提高了自己从应用代码中分离出配置数据的能力。我们也可以保证应用构件能够以一种一致方式被配置，无论配置数据被保存在什么地方。即使我们不知道一个应用在运行时的类（与它所实现的那些接口相对比），我们仍知道如何配置它，只要它是一个Java组件。

一个对象不必实现任何特殊的接口，或把一个特殊的超类扩展成为一个简单的Java组件，以便暴露组件属性（Bean properties）而又不支持组件事件（Bean events）。所必需的是一个无变元的构造器和若干遵守一个简单命名约定的属性方法。和EJB不同的是，Java组件不施加任何运行时开销。不过，处理Java组件比较复杂，而且需要使用反射来实例化对象并按名调用方法。核心的Java组件API未提供在Java组件上执行一些有用操作的能力，比如：

- 在一个单独的操作中设置多个属性，在一个或多个属性设置器失败时抛出一个统一的组合式异常。
- 获得或设置嵌套属性（Nested properties）（即属性的属性），比如getSpouse().getAge()。Java 1.4通过新的java.beans.Statement和java.beans.Expression类引进了对这种属性的支持。

- 在不使Java组件的实现变复杂的情况下支持组件事件传播。只有使用了标准Java组件API，Java组件类才会需要通过调用像java.beans.PropertyChangeSupport那样的API类来做它们自己的“管道工程”。
- 提供一个标准方法，以便在所有组件属性都得到设置之后在一个Java组件上执行初始化。

由于按名调用方法并实例化Java组件意味着处理来自核心反射包的异常，所以错误处理也会变得很笨拙，除非我们使用一个更高的抽象级别。

示例应用中所用框架的最低层是com.interface21.beans包。这提供了轻松处理Java组件的能力并增加了上述增强功能度，同时又尊重和避免了重复标准的Java组件基础结构。

这个包的核心是BeanWrapper接口和默认实现BeanWrapperImpl。一个BeanWrapper对象可以在给定任一Java组件的条件下被创建，而且提供个别或成批地处理属性和增加监听器的能力，其中这些监听器将在使用BeanWrapper处理组件时得到通知。BeanWrapper接口中，最重要的那些方法有如下这些：

```
Object getPropertyValue(String propertyName) throws BeansException;
```

这个方法在给定一个遵守Java组件命名约定的串名称时返回一个属性值。串“age”将返回一个通过getAge()访问器方法来暴露的属性。不过，所有BeanWrapper方法也支持任意深度的嵌套属性：串“spouse.age”将试图从spouse属性的对象值上获得age属性。即使spouse属性的被声明类型没有暴露一个age属性，但只要当前值暴露了一个age属性，这个串仍将工作。

setPropertyValues()方法是一个具有下列签名的并行设置器方法：

```
void setPropertyValues(String propertyName, Object value)
    throws PropertyVetoException, BeansException;
```

这个方法抛出java.beans.PropertyVetoException，以使事件监听器的使用能够像JavaBean规范中所描述的那样“否决”属性修改。如果需要，那个新增的值参数可以使用标准的API PropertyEditor支持（下文讨论）从一个串中被转换而来。另一个设置器方法具有下列签名：

```
void setPropertyValues(PropertyValue pv)
    throws PropertyVetoException, BeansException;
```

这个方法类似于前面的setPropertyValues()方法，但接受一个保存了属性名和值的简单对象。这是允许下列方法所执行的组合式属性更新所必需的：

```
void setPropertyValues(PropertyValues pvs) throws BeansException;
```

这个方法在一个单独的批更新中设置若干个属性。PropertyValues参数含有多个PropertyValue对象。当遇到试图设置个别属性的异常时，属性设置继续进行。任何异常都被保存在一个统一的异常PropertyVetoException中，然后这个异常在整个操作结束时被立即抛出。设置成功的属性依然得到更新。批属性更新在许多情况下都是十分有用的，比如当填充来自HTTP请求参数的Java组件时。

BeanWrapper接口包括了查找属性是否存在的便利方法，例如：

```
boolean isReadableProperty(String propertyName);
```

使用下列方法来获得由一个BeanWrapper包装的对象是可能的：

```
Object getWrappedInstance();
```

这个方法能使我们在组件处理一结束时——比如在初始化之后，就立即转到普通Java调用。

使用像下面这样的方法，可以增加和删除实现标准Java组件接口的事件监听器：

```
void addPropertyChangeListener(PropertyChangeListener l);
void addVetoableChangeListener(VetoableChangeListener l);
```

这意味着如果一个BeanWrapper用来处理它的组件属性，属性更改事件将被激发，即使正被处理的Java组件没有实现属性更改支持。

笔者采取了这个重要的设计决策，以便使Java组件包异常分级结构的根com.interface21.beans.BeanException变成未检查的。由于我们在初始化代码中常常使用Java组件处理，所以这是正确的；未能处理一个命名属性的故障通常将是致命的。不过，让应用代码捕捉BeanException（如果我们选择这么做）仍是可能的。BeanException的几个子类在com.interface21.beans包中定义，其中包括：

- TypeMismatchException，在转换一个属性值到所需类型是不可能时被抛出。
- NotWritablePropertyException，在试图设置一个不可写属性时被抛出。
- MethodInvocationException，在一个属性获得器或设置器方法抛出一个异常时被抛出。

现在，让我们来看一个使用BeanWrapper接口的简单例子。TestBean是一个实现ITestBean接口的具体类，而ITestBean接口含有3个属性：age(int)、name(String)和spouse(Person)，并且它们是使用标准Java组件命名约定来定义的，如下所示：

```
public interface ITestBean {
    int getAge();
    void setAge(int age);
    String getName();
    void setName(String name);
    ITestBean getSpouse();
    void setSpouse(ITestBean spouse);
}
```

通过创建一个新的BeanWrapper对象，我们可以轻松地设置和获得属性值：

```
TestBean rod = new TestBean();
BeanWrapper bw = new BeanWrapperImpl(rod);
bw.setPropertyValue("age", new Integer(32));
bw.setPropertyValue("name", "rod");
bw.setPropertyValue("spouse", new TestBean ());
bw.setPropertyValue("spouse.name", "kerry");
bw.setPropertyValue("spouse.spouse", rod);

Integer age = (Integer) bw.getPropertyValue("age");
String name = (String) bw.getPropertyValue("name");
String spousesName = (String) bw.getPropertyValue("spouse.name");
```

通过反射所获得的信息因效率原因而被缓存了起来，所以使用一个BeanWrapper只强加了十分少的开销。

有时，要被设置的值必须被提供为串（比如HTTP请求参数的值）。BeanWrapperImpl类可以自动把一个串表示转换成任一基元类型。不过，有时必须用一个串表示来创建一个custom对象。com.interface21.beans包支持解决这个问题的标准Java组件API手段。我们可以为该类注册一个java.beans.PropertyEditor实现，以便实现setAsText()方法，而且BeanWrapper类和使用它的所有类都将自动获得从串中创建该类的能力。

Java组件API提供了java.beans.PropertyEditorSupport便利类，我们可以扩展这个便利类来非常轻松地实现属性编辑器，请看一看下面这个类，它能够从<name>_<age>形式的串中创建TestBean对象：

```
class TestBeanEditor extends PropertyEditorSupport {
    public void setAsText(String text) {
        TestBean tb = new TestBean();
        StringTokenizer st = new StringTokenizer(text, "_");
        tb.setName(st.nextToken());
        tb.setAge(Integer.parseInt(st.nextToken()));
        setValue(tb);
    }
}
```

我们忽略了java.beans.PropertyEditor接口的面向GUI的方法，继承了PropertyEditorSupport类所提供的默认实现。上述清单中突出显示的setValue()方法设置了由PropertyEditor为给定输入串所返回的对象值。

我们可以使用标准Java组件API手段来注册这个PropertyEditor，如下所示：

```
PropertyEditorManager.registerEditor(ITestBean.class, TestBeanEditor.class);
```

现在，我们可以设置ITestBean类型的属性了，如下所示：

```
TestBean rod = new TestBean();
BeanWrapper bw = new BeanWrapperImpl(rod);
bw.setPropertyValue("spouse", "Kerry_34");
```

BeanWrapperImpl类默认地注册了若干个属性编辑器（它们在com.interface21.beans.propertyeditors包中定义）。这些属性编辑器包含了对java.util.Property对象（用java.util.Property类所支持的格式指定）和串数组（用CSV格式指定）的支持。

使用“组件工厂（Bean Factory）”

com.interface21.beans包提供了强有力的功能度，但对大多数情况中的直接使用来说，它太低级。这个包实质上是一个构件块，我们可以在它上面建立一个隐藏那些低级细节的更高抽象级别。

无论我们做基于Java组件的“后期装配”有多么容易，通常都需要在应用代码中使用普通的Java调用。这种“后期装配”较简单，也较快速（尽管反射的性能开销常常被夸大），而且除非我们需要做不寻常的事情，否则好的设计通常将意味着我们知道正在与哪些接口打交道（但不知道正在与哪些具体类打交道）。

可是，在应用启动阶段，它又是另外一回事。复杂性可以由框架代码来管理；性能开销可以忽略不计，因为初始化操作将只占总体应用活动的一个极小比例；另外，最重要的是，在这种情形中使用后期装配有非常充分的理由。

`com.interface21.beans.factory`包定义了一种方法，这种方法使用一个“组件工厂”，按名称从一个中心配置贮藏库中获得Java组件。组件工厂的目标是消除需要各Java对象都读取配置属性或实例化对象的要求。配置而是由一个中心构件即组件工厂来读取，而且每个应用对象所暴露的组件属性被设置成反映配置数据。然后，组件工厂制作组件的可用实例，其中每个实例都具有一个惟一名称。复杂的对象图可以在组件工厂内被创建，因为被管理的组件可能会引用同一个组件工厂内的其他组件。

我们对组件工厂的实现使用了`com.interface21.beans`包，从应用代码中隐藏掉了它的低级概念。

为了举例说明这种方法的威力，现在来看一看示例应用中的业务对象。我们的大多数业务对象需要有访问两个基础业务对象的权力：`Calendar`对象（返回关于艺术流派、节日和场次的信息）和`BoxOffice`对象（实现订票过程）。无论`Calendar`对象还是`BoxOffice`对象，它的一个单独实例都应该在整个应用中被共享。

一种显而易见的方法是把`Calendar`和`BoxOffice`都变成单元集，并使用它们的`getInstance()`方法为需要它们的业务对象获得参考。这样的方法在许多应用中得到了使用，但具有下列缺点：

- 单元集的使用使得与接口打交道很困难。在本例中，我们在整个示例应用中已经有了对具体类`Calendar`和`BoxOffice`的硬编码依赖性。因此，不可能改换到一个不同的实现。
- 每个单元集都需要查找它自己的配置数据——或许通过使用属性、JNDI或JDBC。有可能出现这种情形：配置管理不一致，而且单元集代码中乱糟糟地充满了与它的业务目的不相干的配置管理代码。
- `Singleton`设计模式所固有的问题，比如欠灵活性。如果我们需要多个`BoxOffice`对象，怎么办呢？

这些问题可以通过从对象实现中分离出对象创建（用一个传统的工厂）来得到稍许减少，但主要的障碍依然存在。

这里所描述的“组件工厂”不同于传统的工厂，因为组件工厂是通用的。它能够基于运行时被提供的配置信息来创建任一类型的对象。根据笔者的经验，类型安全（只在应用初始化时出现且主要隐藏在基础结构内的一个问题）的损失足以用灵活性方面的巨大收获来弥补。

使用下面所描述的组件工厂方法，`Calendar`和`BoxOffice`将是接口，而不是具体类。这两个接口每个的实现都将暴露Java组件属性，以使一个组件工厂能够配置这些属性。组件属性可能是基元，也可能是对象，由指向组件工厂中的其他组件的参考来决定。配置工作将从应用代码中完全转移到基础结构中。

需要这两个对象的每个业务对象也是Java组件，并且将暴露`Calendar`和`BoxOffice`属性，以使组件工厂能够在初始化时“装配”该应用。应用类将不含有不相干的配置管理代码。配置管理将是一致的，而且应用代码将完全被编写到接口，而非具体类。这保证应用是完全“可插入的”：任一接口的实现都可以在配置中进行改换，不用修改一行Java代码。

如果Calendar和BoxOffice对象是EJB，客户代码可以通过JNDI来查找它们，这也将避免对某一个具体类的依赖性。但是，EJB容器是一个大重量级的“组件工厂”（如果它再不提供什么其他价值的话），而且客户代码会被EJB访问代码搞得相当复杂。

单元集的各种严重缺点已是众所周知的。可是，似乎没有什么更好的替代方案得到普遍使用。Java.beans.beancontext核心包使用了一种同这里所讨论的相类似的方法，尽管这种方法是复杂的，面向GUI的，而且似乎极少得到使用。Apache Avalon服务器框架也有点类似，但比较复杂，而且比较严格地要求应用代码实现框架接口（请参见<http://jakarta.apache.org/avalon/index.html>）。JMX提供了一种得到广泛采用的、基于标准的可能解决方法，但这种方法更复杂，它的重量级更大。

或许，与我们所设定的目标最接近的也是Digester，它从前是Struts的一部分，现在是Apache Commons中的一个不同的开放源项目。这个包使用一种基于规则的方法从一个XML表示中配置Java对象（通常是Java组件，但未必总是），该XML表示也依赖于反射的使用。对基于XML的映射来说，Digester包比本章中所描述的XML组件工厂实现更灵活，但束缚于XML，并且不能支持其他表示中所保存的映射定义。可以证明的是，Digester包也欠直观性，而且一般要求应用开发人员编写更多的代码（进一步信息，请参见<http://jakarta.apache.org/commons/digester.html>）。

com.interface21.beans.factory.BeanFactory接口只含有两个方法：

```
public interface BeanFactory {
    Object getBean(String name) throws BeansException;
    Object getBean(String name, Class requiredType) throws BeansException;
}
```

第一个方法使组件能够按名来获得，如下所示：

```
MyInterface mi = (MyInterface) getBean("myInterface");
```

第二个方法是一个便利方法，如果被找到的对象不具有必需的类型，这个方法就抛出一个异常。下面这行代码和上面那行代码相同，只是这行代码增加了检查项：

```
MyInterface mi = (MyInterface) getBean("myInterface", MyInterface.class);
```

视实现而定，getBean()方法可能会返回一个单元集（由当前组件工厂中具有相同name属性的所有getBean()调用所返回的一个共享实例），也可能会返回一个原型（一个以组件工厂所管理的配置为基础的独立实例）。这种共享实例方法是一种比较有价值的方法，而且也经常被使用。通过维护单一的组件工厂，应用可以避免要求各应用类必须被实现为单元集：“单元集”功能由组件工厂来实施。万一需要发生了变化，我们可以选择自己在该应用中需要使用多少个组件工厂。

一旦分开了应用代码（在应用特有的Java组件中）与配置管理（由一个框架组件工厂实现来处理），我们就可以轻松地使用自己所选择的格式来保存配置。

示例应用中使用得最多的组件定义格式是一个XML文档，这个文档和BeanFactory接口的com.interface21.beans.factory.xml.XmlBeanFactory实现所使用的XML文档一样。

现在来看一看我们在上文中所讨论的相同简单组件类（TestBean）。使用XML格式，我们不用编写Java代码就能获得相同的对象图，如下所示。XML格式多半是自描述的，使用了标识属性名的name属性和含有属性值之串表示的元素值。请注意指向组件工厂内其他组件的那些参考，笔者已对它们做了突出处理。

```
<bean name="rod"
    singleton="true"
    class="com.interface21.beans.TestBean">
    <property name="name">Rod</property>
    <property name="age">31</property>
    <property name="spouse" beanRef="true">kerry</property>
</bean>

<bean name="kerry" class="com.interface21.beans.TestBean">
    <property name="name">Kerry</property>
    <property name="age">34</property>
    <property name="spouse" beanRef="true">rod</property>
</bean>
```

有了这个定义，我们就可以获得一个指向第一个组件的共享实例的参考，如下所示：

```
TestBean rod = (TestBean) beanFactory.getBean("rod");
```

BeanFactory接口的另一个实现com.interface21.beans.factory.support.ListableBeanFactoryImpl可以从属性文件中读取定义。这是一种比XML文档更简单的组件定义方法，因为XML文档与java.util.ResourceBundle类所提供的Java国际化核心支持有联系。上述那些组件可以按照如下所示使用属性语法来定义，属性语法指定每个组件的类，并同样把属性值设置成串：

```
rod.class=com.interface21.beans.TestBean
rod.name=Rod
rod.age=32
rod.spouse(ref)=kerry

kerry.class=com.interface21.beans.TestBean
kerry.name=Kerry
kerry.age=35
kerry.spouse(ref)=rod
```

请注意上述代码中的特殊后缀（ref），它用来指出哪些属性是指向同一个组件工厂中所定义的其他组件的参考。

示例应用中所使用的另一个框架实现com.interface21.jndi.JndiBeanFactory从EJB环境变量中取得组件定义，而这些环境变量在ejb-jar.xml文档中定义。这是一种更冗长的格式（定义环境变量必须使用的方法），所以笔者将只给出上述定义的开始部分：

```
<env-entry>
    <env-entry-name>beans.rod.class</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>com.interface21.beans.TestBean</env-entry-value>
</env-entry>
<env-entry>
    <env-entry-name>beans.rod.name</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Rod</env-entry-value>
</env-entry>
...
```

`com.interface21.jndi.JndiBeanFactory`实现使EJB能够使用和其他Java类相同的配置管理，并避免了为了从应用EJB中检索环境变量而编写冗长的JNDI查找代码的需要。

对`BeanFactory`接口的各种可能实现没有任何限制。例如，它们可以从一个数据库中读取数据，也可以反出行化持久性对象，或在完全不使用反射的情况下创建新的对象。

`BeanFactory`的一个有用于接口`ListableBeanFactory`增加了查询所有定义的能力（一个`BeanFactory`实现可能只在请求时才装入组件定义，而一个`ListableBeanFactory`必须知道所有可能的组件定义）。我们的`xmlBeanFactory`实际上就是一个`ListableBeanFactory`，因为它把XML输入解析为一个DOM文档。`ListableBeanFactory`子接口按如下所示来定义：

```
public interface ListableBeanFactory extends BeanFactory {

    String[] getBeanDefinitionNames();
    int getBeanDefinitionCount();
    String[] getBeanDefinitionNames(Class type);

}
```

查找一个给定类或接口的所有组件定义的能力在初始化期间是非常有用的。例如，它能使下一章中将描述的Web应用框架获得指向能处理HTTP请求的所有类的参考。

通常情况下，一个组件需要知道它的所有属性是什么时候得到设置的，以便它能确认其状态的正确性和（或）执行初始化。我们的组件工厂为此定义了一种方法。如果一个组件工厂所创建的一个组件实现了`com.interface21.beans.factory.InitializingBean`接口，那么它在所有属性都已经得到了设置之后将会接收到一个回调。`InitializingBean`接口只含有一个方法：

```
public interface InitializingBean {
    void afterPropertiesSet() throws Exception;
}
```

这个方法的一个典型实现可能会在执行初始化之前，先核实一个或多个属性已经得到了设置，以确认值的正确性。下面这个例子在试图执行一个JNDI查找之前先核实一个`jndiName`属性已经得到了设置：

```
public final void afterPropertiesSet() throws Exception {
    if (this.jndiName == null || this.jndiName.equals(""))
        throw new Exception("Property 'jndiName' must be set on " +
            getClass().getName());
    Object o = lookup(jndiName);
    initialize(o);
}
```

由于所有应用对象都由一个组件工厂来创建，所以我们可以在一个没有它自己的生成周期管理解决方法的应用类中使用这个组件工厂。

有时，在返回一个组件实例之前，我们可能需要先创建一个组件定义来执行自定义的处理。例如，我们可能需要一个组件定义来注册这个被返回的组件为一个JMS监听器，创建一个指定类的一个对象并返回一个包装这个对象的动态代理，或者返回一个从另一个对象上的一个方法调用中所产生的组件。组件工厂概念支持“自定义的组件定义”，在这个定义中，

一个自定义的组件类可以在返回一个组件之前执行任一动作。

一个自定义的组件定义按如下流程进行工作。

- 组件工厂装入自定义的定义类（它必须被指定）。这个自定义的定义类必须是一个Java组件。
- 组件工厂设置这个自定义的定义类上的属性（必须用一种特殊语法区分这个自定义的定义类）。
- 配置后的自定义组件定义类返回一个包装了被返回组件的BeanWrapper。在返回一个BeanWrapper之前，这个自定义的定义类可以执行它所喜欢的处理。
- 组件工厂设置被返回的BeanWrapper上的属性，就好像它是一个正常的组件定义。

下面这个自定义的组件定义举例说明上述工作流程：

```
<bean name="referenceDataListener"
      definitionClass="com.interface21.jms.JmsListenerBeanDefinition"
>
  <customDefinition property="listenerBean">
    com.wrox.expertj2ee.ticket.referencedata.support.DataUpdateJmsListener
  </customDefinition>

  <customDefinition property="topicConnectionFactoryName">
    jms/TopicFactory
  </customDefinition>

  <customDefinition property="topicName">
    jms/topic/referencedata
  </customDefinition>

  <property name="cachingCalendar" beanRef="true">calendar</property>
</bean>
```

<bean>元素的definitionClass属性指出这是一个自定义的组件定义，并提供这个自定义的定义类。

<customDefinition>元素设置这个自定义定义上的属性。属性的含义将随着自定义的定义类而有所不同。在本例中，listenerBean属性指定那个被返回的实际组件的类。在上面的清单中，那个自定义的定义将创建DataUpdateJmsListener的一个实例，并把这个实例注册为一个JMS监听器。

<property>元素设置那个被返回的实际组件的cachingCalendar属性。这个机制使得该组件工厂接口可以无限地扩展。它对隐藏J2EE API使用的复杂性是特别有用的。

应用上下文

组件工厂概念解决了许多问题，比如由Singleton设计模式的过度使用所产生的那些问题。但是，我们可以转到一个更高的抽象级别来帮助聚拢应用。

应用上下文的目标

以组件工厂为基础，一个应用上下文可以为组成一个应用或子系统的那些Java组件提供一个名称空间，以及运行时共享工作对象的能力。`com.interface21.context.Application Context`接口扩展`ListableBeanFactory`接口，进而增加了如下能力：

- 使用Observer设计模式来出版事件。正如我们在第4章中所讨论过的，Observer设计模式提供了去耦应用构件并实现关系的干净分离的能力。
- 参加一个应用上下文分级结构。应用上下文可以拥有父上下文，进而使开发人员能够选择子系统之间所共享的配置量。
- 运行时在应用构件之间共享工作对象。
- 按照串名称查找消息，提供了对国际化的支持。
- 使测试更方便。除了使用一个组件工厂之外，如果再使用一个应用上下文，那么常常可能提供一个使应用代码能够在应用服务器外部被测试的测试实现。
- 在不同类型的应用中提供一致的配置管理。应用上下文在不同类型的应用（比如Swing GUI应用或Web应用）中和在J2EE应用服务器的内外部都表现相同。

在ApplicationContext接口中，增加的最重要方法包括如下这些：

```
ApplicationContext getParent();
void publishEvent(ApplicationEvent e);
void shareObject(String key, Object o);
```

应用上下文就是含有协作组件的目录，这些组件提供一些附加的服务。示例应用使用了com.interface21.web.context.support.XmlWebApplicationContext ApplicationContext实现，这个实现与Web容器所提供的一个javax.servlet.ServletContext对象关联在一起，并使用我们已经见过的XML组件定义格式。不过，其他实现不需要Web容器，并允许在没有Web接口的情况下做测试（或建立应用）。

应用上下文中的每个组件都被赋予了访问该应用上下文的权限，只要这个组件需要。每个组件只需要实现可选的com.interface21.framework.context.ApplicationContextAware回调接口，它就能访问其他组件，出版事件，或者查找消息。这个接口只含有两个方法：

```
public interface ApplicationContextAware {
    void setApplicationContext(ApplicationContext ctx)
        throws ApplicationContextException;
    ApplicationContext getApplicationContext();
}
```

但是，这种能力不应该在没有充分理由的情况下使用。这里所描述的这个框架的优点之一是：应用代码能够从这个框架中受益，但又不依赖于它。这最大限度地降低了对这个框架的依赖性，并保留了不同地配置应用的选择余地。

组件要想在一个应用上下文中工作，需要通过实现一个可选接口来进行注册，以通知这个应用上下文，但是对用在应用上下文的组件工厂中的组件却没有任何特殊的要求。

每个应用上下文都有一个父上下文。如果父上下文是null（空值），此应用上下文就是应用上下文分级结构的根。如果父上下文是非空值，当前上下文将把它未能找到的组件和它未能解答的消息传递给父上下文。这使得我们能够精确地判断出有多少应用状态在不同组件之间得到了共享。

一个ApplicationContext所提供的Observer支持只在当前服务器的范围内有效；在一个聚类中广播消息将没有任何作用（应该为此使用JMS Pub/Sub消息传递）。不过，只在当前服务器的范围内有效的一个轻量级事件出版仍是非常有用的。通常情况下，消息没有必要被更广泛地广播。例如，假定有这样一条消息：它应该导致一个电子邮件的发送，那么通过使用Observer设计模式，我们既可以添加执行这类任务的监听器，又不把生成这类事件的业务对象的代码变复杂。

Web应用上下文

一个com.interface21.web.context.WebApplicationContext类型的应用必然集成了javax.servlet.ServletContext。

在Web应用中，应用上下文分级结构如下进行工作：

- /WEB-INF/applicationContext.xml文件定义应用的根上下文。这里的定义可供Web ApplicationContext所管理的所有对象使用。根上下文作为一个属性被添加到ServletContext上，使它可供所有Web层对象使用，其中包括服务器小程序筛选器和自定义标志，而这些对象通常都不属于应用配置管理的范围。
- 使用了该框架的每个服务器小程序都有它自己的应用上下文，即根上下文的一个子女（它由一个具有如下文件名格式的文件所定义：/WEB-INF/<servletName>-servlet.xml）。服务器小程序名是Web应用的web.xml文件中相应<servlet>元素的<servlet-name>子元素的值。这使得我们能够有多个控制器服务器小程序，其中每个服务器小程序都具有一个不同的名称空间，但都能访问全局对象。

示例应用只使用了一个具有ticket名称的服务器小程序，并且不需要任何全局配置。因此，所有组件定义都被放置在了/WEB-INF/ticket-servlet.xml文件中。

对测试的影响

笔者在这里已描述过的基础结构是测试友好的：一个重要的考虑因素。

虽然把应用上下文与Servlet API集成在一起并在一个EJB中使用一个组件工厂是可以做到的，但那些核心接口不依赖于J2EE API。因此，完全可以创建一个测试应用上下文，在不需要J2EE服务器的情况下对应用对象进行测试。例如，一个JUnit测试案例可以从Web应用的应用上下文定义文档中创建一个XmlApplicationContext，然后使用它按名获得组件，并测试这些组件的行为。

应用配置基础结构的小结

图11.1的UML图表描绘了我们在上文中所描述的框架类与接口之间的关系。

该图表中省略了一些实现类，这么做的目的是为了说明那些公用接口。需要注意的是，应用代码不必使用这些类，编写一个由应用上下文类“装配”但又不依赖于任何框架类的应用是完全可以的。

示例应用中所用框架的版本不支持动态重配置。笔者的结论是：这种功能度的价值不足以弥补它将引入的复杂性。不过，如果必要，可以增加这种功能度；这种方法比使用单元集和配置的其他“硬编码”方法要灵活得多。

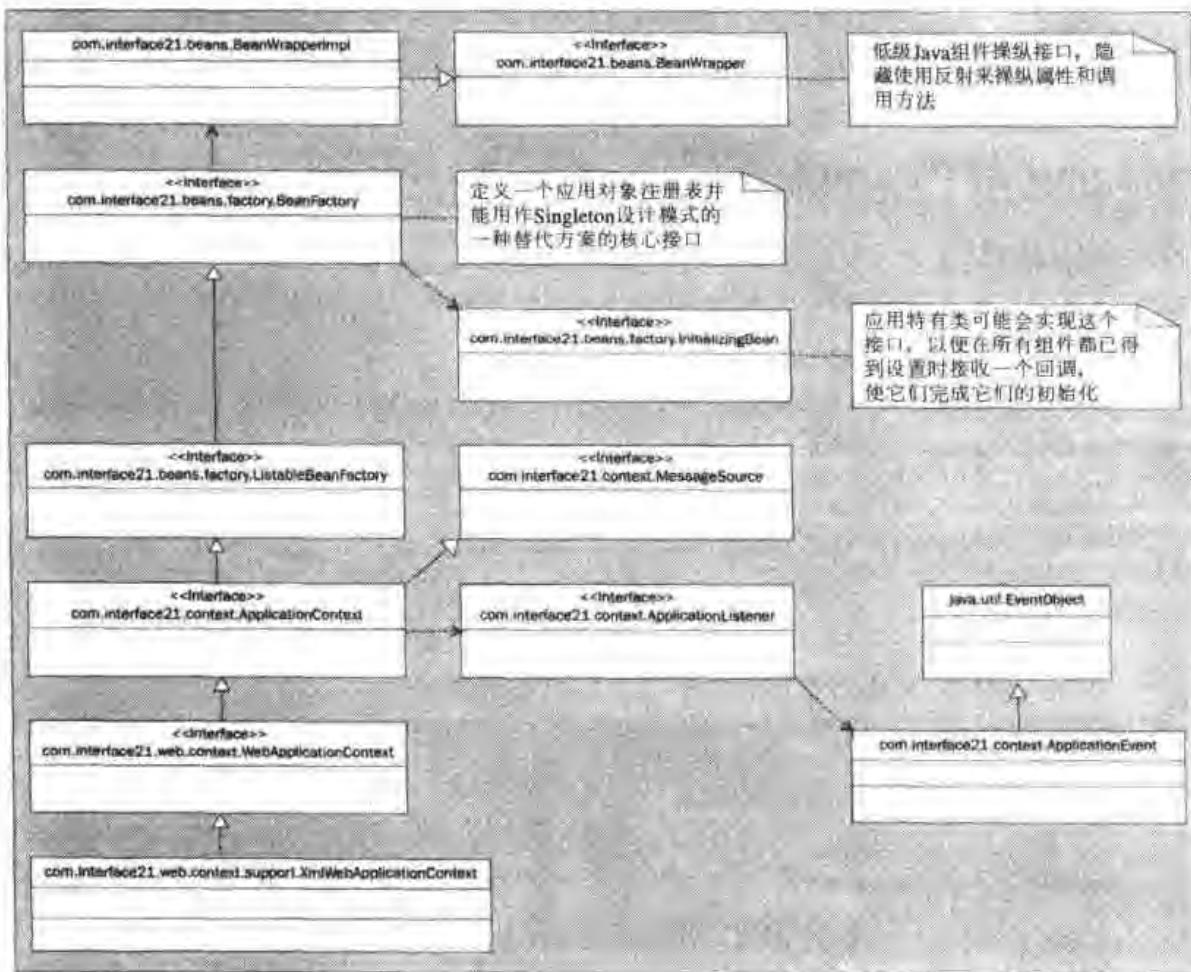


图11.1

在本章的较后面，我们将比较详细地讨论示例应用的代码是如何通过这个框架的使用来简化的。

上面所描述的基础结构能让我们编程到接口，但绝编程不到具体类，因而需要在初始化时依靠一个中心配置管理器（一个“应用上下文”）来“装配”应用。这从应用对象中彻底消除了配置数据和膨胀代码，应用对象只需要暴露Java组件属性。这也具有应用代码不依赖于支持性基础结构的优点，而这种依赖性是一个对许多框架都有影响的问题。

管理API复杂性

既然我们已经有了一个能把我们的应用凝聚在一起的坚固基础结构，现在就让我们来看一看怎样才能使用基础结构来解决J2EE API的复杂性问题。在下一节中，我们将讨论几个关键的J2EE API怎样才能变得更容易使用，以及怎样才能使它们的使用更不易出错。

实现EJB

只要我们避免使用与EJB规范相抵触的基础结构构件（比如通过获得当前类装入器，因

为它排除了动态代理解决方案），对EJB代码就没有什么可担心的特殊事情了。不过，通过从通用超类中继承应用EJB实现类，我们可以极大地简化EJB代码。

用于EJB的抽象超类

我们应该从满足下列目标的常用超类中获得我们的EJB：

- 实现了与组件实现类无关的、在生成周期内有效的方法。
- 在凡是可能的地方，都迫使EJB实现来自自主接口的、不是EJB必须实现的那些接口所必需的方法。
- 提供一致的日志记录解决方案。
- 避免EJB必须使用JNDI来查找环境变量的要求，从而简化了配置管理。

图11.2所示的UML图显示了我们的通用基础结构所提供的那些EJB超类的分级结构，以及那些超类与EJB规范所要求的接口有怎样的关系。所有基础结构类都在com.interface21.ejb.support包中。我们将在下文中讨论每个类的完整代码清单。

需要注意的是，我们没有打算为实体组件提供超类。实体组件应该使用CMP，通常不应该含有业务逻辑，以免给容器增加提供基础结构代码的负担。

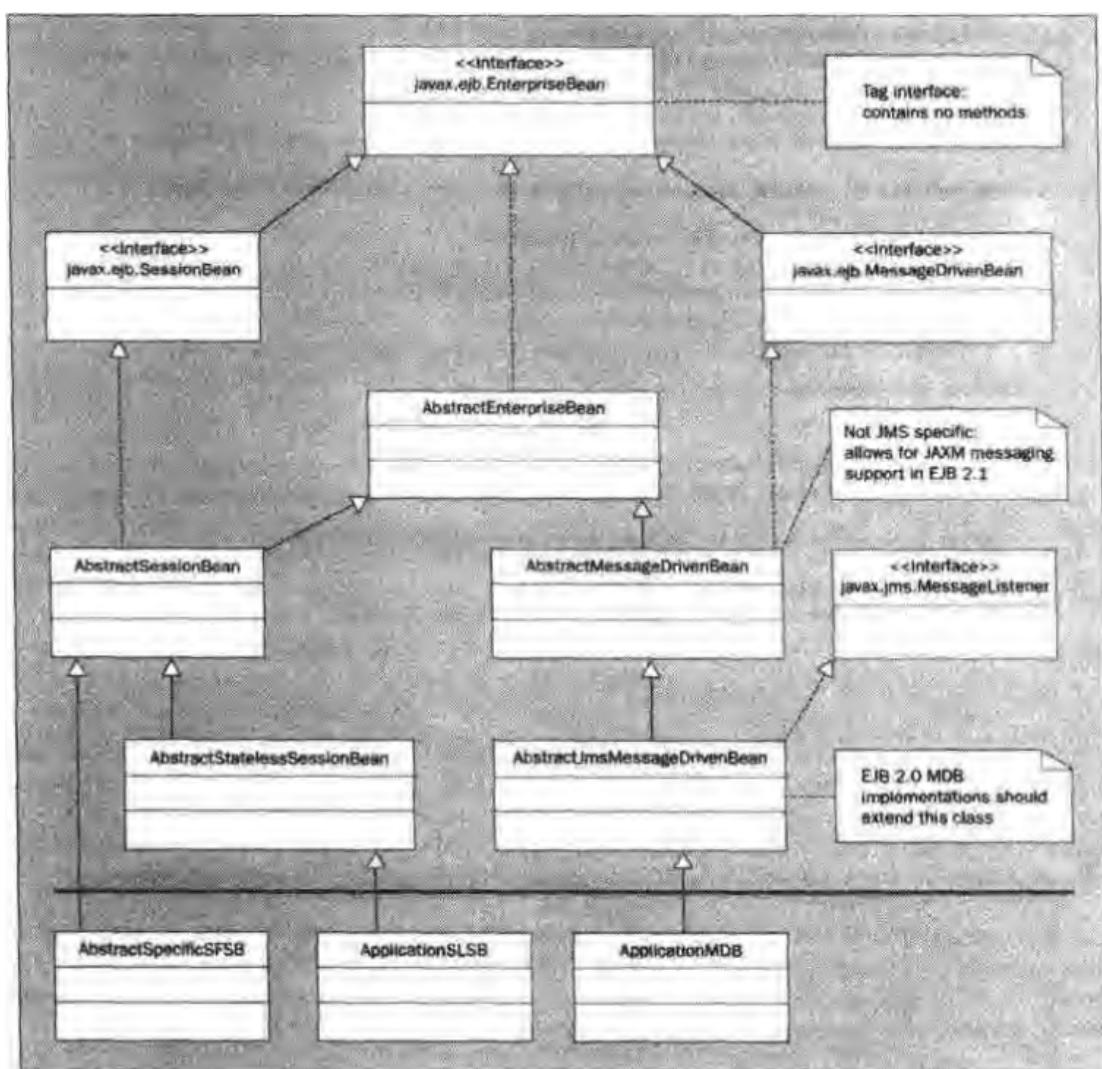


图11.2

虽然这里有相当多的类，但每个类都很简单，而且继承性分级结构能使我们既没有代码重复，又支持无状态与有状态会话组件和消息驱动式组件。水平线以上的类都属于框架，而水平线以下的类显示了应用EJB实现类怎样才能扩展水平线以上的框架类。

继承性分级结构的根（不被应用EJB直接用做超类）是AbstractEnterpriseBean类。这个类实现javax.ejb.EntityBean标志接口，并提供标准的日志记录与配置管理。下面是一个完整的清单：

```

package com.interface21.ejb.support;

import java.util.logging.Logger;
import javax.ejb.EJBException;
import javax.ejb.EnterpriseBean;
import com.interface21.beans.BeansException;
import com.interface21.beans.factory.ListableBeanFactory;
import com.interface21.jndi.JndiBeanFactory;

public abstract class AbstractEnterpriseBean implements EnterpriseBean {
    protected final Logger logger = Logger.getLogger(getClass().getName());
    private ListableBeanFactory beanFactory;
    protected final ListableBeanFactory getBeanFactory() {
        if (this.beanFactory == null) {
            loadBeanFactory();
        }
        return this.beanFactory;
    }
    private void loadBeanFactory() {
        logger.info("Loading bean factory");
        try {
            this.beanFactory = new JndiBeanFactory("java:comp/env");
        }
        catch (BeansException ex) {
            throw new EJBException("Cannot create bean factory", ex);
        }
    }
}

```

logger保护实例变量使用第4章中所讨论过的Java 1.4日志记录仿真包，或标准的Java 1.4日志记录（如果可用）。getBeanFactory()方法迟钝地从环境变量中装入组件工厂。这是非常有用的，因为它避免了使用JNDI来查找环境变量的要求，然而又允许了EJB部署时的参数化。我们可用实例化并配置DAO之类的助手对象，并把配置变量放在一个简单的组件中。

我们已经把一个低级API换成了一个相同的高级API，后者也是我们的整个应用中所使用的API。通过把工厂制造功能度简化成助手类（而非EJB代码），我们发扬了良好的习惯，并改进了可测试性。通常，我们可以在不用EJB容器的情况下测试助手类。例如，下列环境变量为示例应用的BoxOffice EJB定义了一个DAO：

```

<env-entry>
    <env-entry-name>beans.dao.class</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>
        com.wrox.expertj2ee.ticket.boxoffice.support.jdbc.
        JBoss30OracleJdbcBoxOfficeDao
    </env-entry-value>
</env-entry>

```

这个特别的DAO不需要任何属性，但这些属性可以使用我们的组件工厂约定来设置。只需通过编辑ejb-jar.xml部署描述符，就可以把该DAO助手的类修改成其他任何一个实现com.wrox.expertj2ee.ticket.boxoffice.support.BoxOfficeDao接口的对象，无需修改EJB代码。

EJB规范要求所有会话组件都实现javax.ejb.SessionBean接口，如下所示。这包括setSessionContext()方法和其他生成周期方法。

```
package javax.ejb;

import java.rmi.RemoteException;

public interface SessionBean extends EnterpriseBean {
    void ejbActivate() throws EJBException, RemoteException;
    void ejbPassivate() throws EJBException, RemoteException;
    void ejbRemove() throws EJBException, RemoteException;
    void setSessionContext(SessionContext ctx)
        throws EJBException, RemoteException;
}
```

因此，所有的会话组件都可以从我们的AbstractEnterpriseBean基类的一个简单子类中派生出来，不过，这个子类需要实现setSessionContext()方法来将会话上下文存储在一个实例变量中，并通过一个getSessionContext()保护方法来暴露已保存的这个上下文。我们还增加了必需的ejbRemove()生成周期方法的一个空实现，以便需要时能替换这个空实现。

```
package com.interface21.ejb.support;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public abstract class AbstractSessionBean extends AbstractEnterpriseBean
    implements SessionBean {

    private SessionContext sessionContext;

    protected final SessionContext getSessionContext() {
        return sessionContext;
    }

    public void setSessionContext(SessionContext sessionContext) {
        logger.info("setSessionContext");
        this.sessionContext = sessionContext;
    }

    public void ejbRemove() {
        logger.info("AbstractSessionBean NOP ejbRemove");
    }
}
```

有状态会话组件可以直接子类化这个超类。有状态会话组件常常不必实现ejbRemove()方法，但它们始终需要保证ejbPassivate()和ejbActivate()生成周期方法的行为正确。我们已在第10章中讨论过如何正确地实现这两个方法。因此，我们在AbstractSessionBean中没有实现这两个方法。

相反，让EJB容器在无状态会话组件上调用ejbPassivate()和ejbActivate()生成周期方法是非法的，尽管无状态会话组件实现类是实现它们所必需的（不合乎逻辑）。因此，无状态会

话组件可以从**AbstractSessionBean**的一个简单子类中派生出来，该子类为抛出异常而实现这两个生成周期方法。让任一子类替换这个实现都毫无理由，尽管我们无法使这两个方法成为终结方法，因为它与**EJB**规范相抵触。

我们用于无状态会话组件的超类也迫使子类实现一个无变元的**ejbCreate()**方法——与主接口上的必需方法**create()**相配，而子类是通过把这个方法定义为一个抽象方法来做到这一点的。这就把只在**EJB**部署时才进行的检查转移到了编译的时候，因为这个方法不是**EJB API**所必需的。下面是**AbstractStatelessSessionBean**类的完整清单，这个类应该被SLSB实现类用做一个超类。

```
package com.interface21.ejb.support;

import javax.ejb.CreateException;
import javax.ejb.EJBException;

public abstract class AbstractStatelessSessionBean
    extends AbstractSessionBean {

    public abstract void ejbCreate() throws CreateException;

    public void ejbActivate() throws EJBException {
        throw new IllegalStateException(
            "ejbActivate must not be invoked on a stateless session bean");
    }

    public void ejbPassivate() throws EJBException {
        throw new IllegalStateException(
            "ejbPassivate must not be invoked on a stateless session bean");
    }
}
```

这使SLSB子类实现一个**ejbCreate()**方法及它们的业务方法。需要记住的是，应用**EJB**通常实现一个“业务方法”接口，保证这些业务方法匹配于它们的构件接口上所定义的那些方法。在极少数的情况下，一个SLSB可能会替换**AbstractSessionBean**中所定义的**ejbRemove()**方法。**ejbCreate()**的实现可能会使用可从**AbstractEnterpriseBean**超类中获得的组件工厂来创建像**DAO**之类的助手对象。这些超类可以由带有本地或远程接口或同时带有这两种接口的**EJB**来使用。

下面让我们来看一个例子，了解“业务方法接口”设计模式和我们的通用超类如何联合起来保证编译程序（而不仅仅部署工具）强行使SLSB实现类是有效的。下面这个业务方法接口定义了虚构**EJB**的那些远程方法。

```
import java.rmi.RemoteException;

public interface Calculator {
    int getUncacheableValue() throws RemoteException;
    int getCacheableValue() throws RemoteException;
}
```

按照上一章中所描述的那个业务方法设计模式，**EJB**的远程接口将不含有它自己的任何方法，但将同时扩展这个接口和**javax.ejb.EJBObject**：

```
import javax.ejb.EJBObject;

public interface CalculatorRemote extends EJBObject, Calculator { }
```

这个组件实现类将实现业务方法接口，并扩展AbstractStatelessSessionBean。这意味着编译程序将迫使我们实现为一个有效EJB所需要的所有方法。笔者已经在下列清单中，突出了表示ejbCreate()方法怎样才能用来从AbstractEnterpriseBean基类所暴露的组件工厂中装入一个助手对象的那些行：

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import com.interface21.ejb.support.AbstractStatelessSessionBean;

public class CalculatorEJB extends AbstractStatelessSessionBean
    implements Calculator {

    private MyHelper myHelper;

    public void ejbCreate() {
        this.myHelper = (MyHelper) getBeanFactory().getBean("myHelper");
    }

    public int getUncacheableValue() {
        return 0;
    }

    public int getCacheableValue() {
        return 0;
    }
}
```

我们的超类已经提供了实际价值。这个EJB实现类不含有任何与其业务目的不相干的代码，而且可以轻松地装入其配置被全部保存在Java代码外部的助手对象。

最后，让我们来看一看消息驱动式组件（MDB）。MDB实现类还必须满足除了实现javax.ejb.MessageDrivenBean接口之外的一些要求：它们还必须实现一个不带变元的ejbCreate()方法，而且它们必须实现javax.jms.MessageListener接口来处理JMS消息。我们可以通过扩展AbstractEnterpriseBean来省掉MessageDrivenContext，并迫使子类实现一个ejbCreate()方法，如下所示：

```
package com.interface21.ejb.support;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;

public abstract class AbstractMessageDrivenBean extends AbstractEnterpriseBean
    implements MessageDrivenBean {
    private MessageDrivenContext messageDrivenContext;

    protected final MessageDrivenContext getMessageDrivenContext() {
        return messageDrivenContext;
    }

    public void setMessageDrivenContext(
        MessageDrivenContext messageDrivenContext) {
        logger.fine("setMessageDrivenContext");
        this.messageDrivenContext = messageDrivenContext;
    }

    public abstract void ejbCreate();
```

```

    public void ejbRemove() {
        logger.info("ejbRemove");
    }
}

```

虽然EJB 2.0只支持JMS消息传递，但EJB 2.1另外还支持JAXM消息传递，这意味着javax.jms.MessageListener仅仅是一个EJB 2.1 MDB可能实现的3个可选消息传递接口之一。因此，我们不是把AbstractMessageDrivenBean类束缚于JMS的使用上，而是通过提出在JMS特有子类中必须实现javax.jms.MessageListener的要求来保证与EJB 2.1的向前兼容性。

```

package com.interface21.ejb.support;

import javax.jms.MessageListener;

public abstract class AbstractJmsMessageDrivenBean
    extends AbstractMessageDrivenBean implements MessageListener {

    // Empty

}

```

子类化这个类的应用MDB被保证满足EJB 2.0 JMS MDB的各项要求。由IDE自动做了占位程序处理的次要子类看起来像下面这样：

```

public class SimpleMDB extends AbstractJmsMessageDrivenBean {

    public void ejbCreate() {
    }

    public void onMessage(Message message) {
    }
}

```

同会话组件一样，ejbCreate()方法可以访问由AbstractEnterpriseBean超类所提供的组件工厂。

当然，我们在实现EJB时不是必须使用这些（或相似）超类，但由于它们简化应用代码，并减少导致开发 - 部署周期被无谓浪费的错误，所以它们提供了非常大的价值。使用让应用EJB失去了它们自己的继承分层结构的具体继承性很少会出现问题。由于EJB编程限制的缘故，通过子类化一个对象来使该对象变成一个EJB，从而实现必需的生成周期方法不是一个太好的主意。不过，我们的解决方法让一个新的EJB使用现有代码变得更容易：通过使用现有类作为经由组件工厂实例化过的助手对象。自定义的应用EJB超类可以通过简单地子类化上述框架超类之一来增加应用特有的附加行为。

访问EJB

在EJB与使用这些EJB的代码之间实现松耦合是十分重要的。EJB客户代码直接访问EJB有许多坏处：

- 客户代码必须在多个地方处理JNDI查找。获得并使用EJB参考是十分笨拙的。我们需要对EJB主接口做一个JNDI查找，并调用EJB主接口上的一个创建方法。

- EJB JNDI名称被保存在多个地方，这使修改客户代码变得很困难。
- 客户与EJB层耦合得太紧密，这使修改它所暴露的那些接口变得很困难。
- 为EJB层或诸如命名上下文和EJB主接口参考之类的服务器资源所返回的应用数据引入一个一致的高速缓存策略是不可能的。尤其是在分布式应用中，高速缓存调用EJB的结果对满足性能需求来说可能是至关重要的。
- 客户代码必须处理来自EJB层的低级已检查异常。对于本地和远程EJB，都存在JNDI NamingException和EJBCreateException。对于远程EJB，起点或EJB上的每个调用都可能会抛出java.rmi.RemoteException。除了远程异常（我们将在下文中讨论）之外，这是极少要实现的异常处理。在一个本地应用中，查找一个EJB起点或使用一个无变元的create()方法创建一个无状态会话组件的失败，几乎肯定预示着严重的应用故障。我们很可能需要记录下这个故障，并愉快地通知用户此时由于一个内部错误而无法处理他们的动作。

最重要的是，如果允许代码直接与EJB访问API打交道，我们不一定把代码束缚在一个特定的实现策略上。

要抽象化与Java接口的内层通信，不要依赖于像EJB这样的J2EE技术。技术不是实现的特性。

两个众所周知的J2EE设计模式（Service Locator和Business Delegate）解决了这些问题。在本节中，我们将讨论它们的优点，以及怎样实现它们。同实现EJB一样，可以使用通用的基础结构类简化应用代码。我们可以把EJB访问与我们的总体应用基础结构集成在一起：通过使服务定位器和业务委托变成Java组件——由一个应用组件工厂来配置。

本地与远程访问

通过本地与远程接口访问EJB之间存在极大的差别。通过本地接口调用一个EJB与调用一个普通Java对象没有太大的差别。EJB容器截获每个方法调用，但调用者和EJB两者都正运行在同一个JVM中，这意味着根本不可能出现分布式故障和串行化问题。

远程调用是一个完全不同的问题。我们可能会遇到分布式故障，而且必须考虑效率：“饶舌的”调用，或者说交换过量的参数数据对性能是灾难性的。

我们可以在针对远程调用的两个基本策略之间进行选择：可以设法实现本地—远程透明度，进而向调用者隐藏远程调用的问题；也可以使用一个客户端门面来为EJB调用提供一个可本地访问的联系点。

笔者不主张本地—远程透明度。由于Java RMI的设计者们没有做两者中的任何一个，所以这种透明度在调用EJB时实现起来相当困难。其次，这种透明度也鼓励严重低效率。

最好是提供一个本地门面，以便所有与EJB层的特性都通过这个门面传送。这是对Business Delegate设计模式的出色的运用（我们将在下文中讨论）。

Service Locator与Business Delegate J2EE设计模式

从一个EJB实现中分离出调用代码的第一步，是保证所有EJB参考都通过一个公用工厂来获得。这避免了JNDI查找代码的重复，并且能高速缓存JNDI查找的结果，以提高性能（尽管系统开销在不同的服务器之间有所不同，但获得一个命名上下文和查找一个主接口都可能是高代价的操作）。这样的对象就叫做服务定位器（Service Locator）（请参见“Core J2EE Patterns”一书）。

缺点是客户仍将需要直接与EJB打交道。它们必须仍处理分布式应用中的远程异常，并且可能需要捕捉创建EJB时被抛出的异常。不过，一个服务定位器可以返回指向SLSB EJB实例的参考，而不是只返回指向主接口的参考（服务定位器可以在必要时调用无参数create()方法）。这就使客户不再需要处理可能由于试图从一个主接口中创建一个EJB所引起的那些已检查异常。

当可能时，客户不必知道EJB正用来实现业务接口。这明显是我们所希望的；除非我们使用EJB来实现一个分布式模型，否则就没有理由把EJB看做一个对业务接口的特定实现有帮助的框架。Service Locator设计模式对本地EJB访问非常管用。

下面让我们来看一看在涉及SLSB的地方使用访问定位器的两种方法。

使用Typed Service Locator访问EJB

在第一种方法中，我们为每个SLSB使用一个专用的、线程安全的访问定位器。每个访问定位器都实现一个不是EJB特定的工厂接口：一个可以被实现成把相关业务接口的任一实现返回给调用者的方法。当EJB用于实现工厂接口时，被返回的对象是一个EJB参考，而且EJB的构件接口将扩展业务接口。

在这种方法的笔者所喜欢的实现中，工厂方法抛出一个运行时异常，而不是抛出一个已检查异常，因为创建业务对象的故障可能会是致命的。如果重试是可能的，服务定位器就可以在抛出一个异常给调用者之前重试。

每个服务定位器实现将需要执行一个JNDI查找来为相关的EJB缓存主接口。这意味着使用通用的基础结构类来隐藏JNDI的使用。我们的框架包括了对这方面的便利支持：应用服务定位器只需要子类化com.interface21.ejb.access.AbstractLocalStatelessSessionServiceLocator类，并实现抽象的setEjbHome()方法及相关的工厂接口。

AbstractLocalStatelessSessionServiceLocator类扩展com.interface21.jndi.AbstractJndiLocator类，而后者可以用来执行任意JNDI查找并缓存结果。这使得Service Locator模式不仅能用于EJB访问，而且还能用来查找和缓存通过JNDI获得的任何资源。AbstractJndiLocator类是为用做我们的应用框架中的一个Java组件而设计的。下面是一个完整的清单：

```
package com.interface21.jndi;

import java14.java.util.logging.Logger;
import javax.naming.NamingException;

import com.interface21.beans.factory.InitializingBean;

public abstract class AbstractJndiLocator implements InitializingBean {
```

```

protected final Logger logger = Logger.getLogger(getClass().getName());

private static String PREFIX = "java:comp/env/";

private String jndiName;

public AbstractJndiLocator() {
}

```

JndiName属性设置器能使EJB或其他资源的JNDI名称被保存在Java代码的外面：

```

public final void setJndiName(String jndiName) {
    if (!jndiName.startsWith(PREFIX))
        jndiName = PREFIX + jndiName;
    this.jndiName = jndiName;
}

public final String getJndiName() {
    return jndiName;
}

```

afterPropertiesSet()方法（由组件工厂在所有属性都得到了设置之后调用）的下列实现执行JNDI查找。如果对象被成功地找到，afterPropertiesSet()方法就用这个对象去调用抽象的模板方法located()。子类必须实现这个方法来根据对象的类型执行它们自己的初始化。如果查找失败，作为结果的javax.Naming.NamingException就被抛出，并由应用框架来处理（这个异常被看做一个致命的初始化异常）。

```

public final void afterPropertiesSet() throws Exception {
    if (this.jndiName == null || this.jndiName.equals(""))
        throw new Exception("Property 'jndiName' must be set on " +
                            getClass().getName());
    Object o = lookup(jndiName);
    located(o);
}

protected abstract void located(Object o);

private Object lookup(String jndiName) throws NamingException {
    logger.info("Looking up object with jndiName '" + jndiName + "'");
    // This helper will close JNDI context
    Object o = new JndiServices().lookup(jndiName);

    logger.fine("Looked up object with jndiName '" + jndiName +
               "' OK: [" + o + "]");
    return o;
}

```

AbstractLocalStatelessSessionServiceLocator子类实现located()模板方法来检查对象实际上是一个EJB本地起点，并调用一个抽象的setEjbHome()方法（知道这种应用特有类型的EJB起点的子类必须实现这个方法，以便这些子类能够从这个方法中获得EJB参考）。

```

package com.interface21.ejb.access;

import javax.ejb.EJBLocalHome;

import com.interface21.beans.FatalBeanException;
import com.interface21.jndi.AbstractJndiLocator;

public abstract class AbstractLocalStatelessSessionServiceLocator
    extends AbstractJndiLocator {

    public AbstractLocalStatelessSessionServiceLocator() {
    }

    public AbstractLocalStatelessSessionServiceLocator(String jndiName) {
        super(jndiName);
    }

    protected abstract void setEjbHome(EJBLocalHome home);

    protected final void located(Object o) {
        if (!(o instanceof EJBLocalHome))
            throw new FatalBeanException("Located object with JNDI name '" +
                getJndiName() + "' must be an EJBLocalHome object", null);
        setEjbHome((EJBLocalHome) o);
    }
}

```

缓存指向本地EJB主接口的参考始终都是安全的。虽然EJB规范（§ 6.2.1）暗示缓存远程主接口是安全的，但被缓存的远程主接口在有些服务器中可能会变得“陈旧”的微小可能性是存在（例如，如果宿主EJB的远程服务器在客户的生存期内被重新启动过）。

图11.3所示的UML类图描述了用于本地和远程SLSB服务定位器的超类，以及一个LocalSLSBBoxOfficeFactory类怎样用来按需要创建示例应用中的BoxOffice对象（实际上是EJB）。

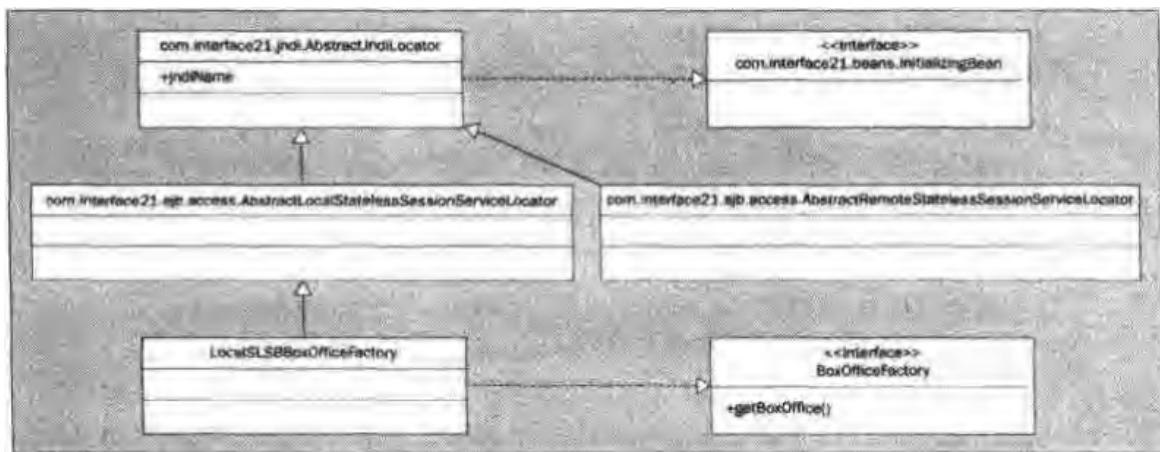


图11.3

下面让我们来看一看这些超类怎么用于简化一个真实服务定位器的实现。`BoxOfficeFactory`接口是我们的应用特有类型化服务定位器的公共接口。

```
public interface BoxOfficeFactory {
    BoxOffice getBoxOffice() throws FatalException;
}
```

扩展com.interface21.ejb.access.AbstractLocalStatelessSessionServiceLocator超类使得实现这个接口变得微不足道：

```
public class LocalSLSBBoxOfficeFactory
    extends AbstractLocalStatelessSessionServiceLocator
    implements BoxOfficeFactory
```

`setEjbHome()`抽象保护方法的下列实现在把主接口强制转换成适当类型之后，缓存了主接口。这个方法将只在服务定位器被初始化时得到一次调用。

```
private BoxOfficeHome home;
protected void setEjbHome(EJBLocalHome home) {
    this.home = (BoxOfficeHome) home;
}
```

来自BoxOfficeFactory接口的`getBoxOffice()`方法（将由使用EJB的代码调用）的下列实现根据需要创建一个新的SLSB参考。EJBCreate异常被捕获，并且一个未检查异常FatalException被重新抛出。

```
public BoxOffice getBoxOffice() throws FatalException {
    try {
        return home.create();
    }
    catch (CreateException ex) {
        throw new EjbConnectionFailure(
            "Cannot create BoxOffice EJB from home", ex);
    }
}
```

下列组件定义显示了怎么才能在示例应用的组件工厂中定义一个这样的服务定位器。必须被设置的惟一属性是jndiName——从AbstractJndiLocator超类中继承而来。

```
<bean name="boxOfficeFactory"
    class="com.wrox.expertj2ee.ticket.framework.ejb.LocalSLSBBoxOfficeFactory" >
    <property name="jndiName">ejb/BoxOffice</property>
</bean>
```

这是一种对本地EJB非常管用的简单方法。这种方法向使用业务接口的代码隐藏了EJB访问。缺点之一是它为每个EJB都要求一个工厂接口和一个实现类。通常情况下，这不是一个主要问题，因为经过精心设计，只有少数系统才会需要大量EJB。稍微严重的问题是，使用了EJB的所有代码都必须与工厂对象打交道：持续保存指向业务接口的参考是不可能的。

透明的动态代理：用于本地EJB访问的类型化服务定位器的一个替代方案

Service Locator设计模式的、实现相同目标的一个替代方案，是把组件工厂隐藏在一个

自定义的组件定义中，就像我们在前面讨论“组件工厂”时所描述的那样。自定义的组件定义查找并缓存组件的主接口，然后再返回所需接口的一个对象，这个对象实际上是一个包装了SLSB实例的动态代理。在每个方法调用开始之前，动态代理（它是线程安全的）透明地创建SLSB的一个新实例。虽然自定义的组件定义不知道EJB的类型，但却知道SLSB本地接口必须暴露一个无变元的create()方法，然后可以使用一个BeanWrapper对象来按名调用这个方法。基准测试显示这种方法只有很少的性能开销。

如此使用动态代理和反射性方法调用好像很复杂，但这种复杂性被隐藏在框架代码内，而且我们能够编写较少的应用代码。应用只需如下所示定义一个使用自定义组件定义的组件即可。请注意JNDI名称和业务接口的指定。

```
<bean name="boxOffice"
      definitionClass=
      "com.interface21.ejb.access.LocalStatelessSessionProxyBeanDefinition">

    <customDefinition property="businessInterface">
        com.wrox.expertj2ee.ticket.boxoffice.BoxOffice
    </customDefinition>
    <customDefinition property="jndiName">
        ejb/BoxOffice
    </customDefinition>

</bean>
```

至此，调用代码就可以像下面这样简单地获得一个BoxOffice对象，并缓存它。

```
BoxOffice boxOffice = (BoxOffice) beanFactory.getBean("boxOffice");
```

这多半不是必需的：需要使用BoxOffice接口的对象可以简单地暴露一个组件属性，并让应用上下文在运行时把它设置成BoxOffice接口。

这种方法的优点是不要求自定义代码访问EJB。我们只需要使用自定义的组件定义来定义一个组件。

这也是示例应用中所使用的方法，尽管换到Typed Service Locator方法将是很简单的（本书的配书下载中包含了这种方法的代码）。

要想了解这种方法的实现，请参见com.interface21.ejb.access.LocalStatelessSessionProxy BeanDefinition类中的代码。

对远程EJB访问使用Business Delegate模式

Business Delegate模式实现了一种更高级别的分离，因为在这个模式中，客户端对象暴露由对EJB的调用所实现的业务方法。这是一种比代理方法更好的方法，因为业务委托可以改变方法签名，比如修改异常类型，组合方法调用，甚至调用多个EJB。

由于Business Delegate模式比Service Locator模式需要更多的实现工作量，所以笔者将把重点放在它和远程接口的使用上，Business Delegate模式必须和远程接口一起使用才提供实际价值（若和本地EJB访问一起使用，该模式的价值将会降低）。业务委托方法的优点包括如下这些：

- 业务委托或许能重试失败的事务（如果这一点已得到预示），又无需把调用代码搞得复杂。

- 以一种一致的方式处理来自服务器的异常是可能的，每个客户又不必捕捉与J2EE基础结构相关的异常，比如java.rmi.RemoteException和javax.naming.NamingException。业务委托可以捕捉这种低级异常，记录它们，并抛出一个对客户代码更有意义的应用特有异常。把EJB层异常包装成未检查异常以简化客户代码可能也是恰当的。这么做在异常为致命异常时有意义；例如，如果我们已经确定在遇到一个妨碍用例执行的远程异常以后重试没有用处。
- 可能实现较高抽象级别的操作。例如，如果某个特定用例需要多个EJB调用，这些调用或许能够在业务委托中进行。
- 可能通过提供业务委托接口的缓存性实现引入高速缓存。这将使所有客户都受益。

使用业务委托模式的缺点是EJB层在客户端上所暴露的业务接口的可能重复。不过，使用业务委托却让我们有机会提供一个最满足客户的各种需要的接口。例如，通过省略不相干的操作，我们可以简化EJB层所暴露的接口；通过只抛出已检查异常（如果它们是API的一部分），我们可以简化客户代码，并且可以期待客户来处理这些异常。

表面上看，让业务委托实现会话组件的业务方法接口很有吸引力。不过，这不能分离开EJB层与客户层，而且不能提供我们已经确认的一些好处，比如隐藏远程异常及简化客户API。因此，笔者不建议复制业务委托中的方法签名（“**EJB Design Patterns**”一书中所推荐的方式）。如果这么做是正确的，Service Locator设计模式更简单，并且应该被首选。

我们的框架提供对实现业务委托的轻松支持。我们为类型化服务定位器所使用的同一个超类AbstractLocalStatelessSessionServiceLocator可以用做一个业务委托的超类。现在，子类不仅按要求返回EJB的一个实例，还将实现业务操作，从而在必要时调用EJB。

由于示例应用使用带有本地接口的EJB，所以没有必要使用Business Delegate设计模式，因为该模式比服务定位器模式需要更多的实现工作量。

下面这个例子说明使用业务委托来访问远程EJB的用法。它使用了一个虚构的EJB，该EJB使用了一个暴露两个int值的远程接口：一个可以被高速缓存的值和一个不能被高速缓存的值。我们在前文的“实现EJB”一节中已经见过这个EJB的业务方法接口和构件接口。其中的远程接口暴露下列两个业务方法：

```
int getUncacheableValue() throws RemoteException;
int getCacheableValue() throws RemoteException;
```

由于使用了Business Delegate设计模式，所以通过隐藏远程异常并重新抛出已检查的致命异常，我们可以简化调用代码，并且可以通过把可缓存的数据值缓存一个指定期间来改善性能。

当使用Business Delegate设计模式时，定义一个让应用代码与之打交道的业务委托接口是个好习惯。这使得一个缓存性实现能够在不影响客户代码的情况下被实例化。这个接口对EJB来说可能会像下面这样：

```
public interface MyBusinessDelegate
{
    int getUncacheableValue() throws FatalException;
    int getCacheableValue() throws FatalException;
}
```

下列实现扩展AbstractRemoteStatelessSessionServiceLocator框架超类，该超类执行必要的JNDI查找，并给这个实现提供一个EJB起点，这个实现可以在setEjbHome()方法中高速缓存这个EJB起点。

```
public class MyBusinessDelegateImpl
    extends AbstractRemoteStatelessSessionServiceLocator
    implements MyBusinessDelegate {

    private static long MAX_DATA_AGE = 10000L;
    private CalculatorHome home;
    private long lastUpdate = 0L;

    protected void setEjbHome(EJBHome home) {
        this.home = (MyHome) home;
    }

    private CalculatorRemote create() throws FatalException {
        try {
            return home.create();
        }
        catch (CreateException ex) {
            throw new EjbConnectionFailure(
                "Cannot create BoxOffice EJB from home", ex);
        }
        catch (CreateException ex) {
            throw new EjbConnectionFailure(
                "Cannot create BoxOffice EJB from home", ex);
        }
    }

    public int getUncacheableValue() throws FatalException {
        try {
            return create().getUncacheableValue();
        }
        catch (RemoteException ex) {
            throw new EjbConnectionException("Can't connect to EJB", ex);
        }
    }

    public synchronized int getCacheableValue() throws FatalException {
        if (System.currentTimeMillis() - lastUpdate > MAX_DATA_AGE) {
            try {
                this.cachedValue = create().getCacheableValue();
                lastUpdate = System.currentTimeMillis();
            }
            catch (RemoteException ex) {
                throw new EjbConnectionException("Can't connect to EJB", ex);
            }
        }
        return this.cachedValue;
    }
}
```

抽象化对EJB的访问是很重要的。如果EJB使用了本地接口，应使用Service Locator设计模式；如果调用代码不应该与EJB实例打交道有充分的理由，则使用Business Delegate设计模式。如果EJB使用了远程接口，应使用Business Delegate设计模式，并向客户隐藏远程访问细节。

使用JMS

和JDBC一样，JMS也是一个复杂的API。直接与它打交道会使应用代码的意图变得难懂。因此，使用一个抽象层是很必要的。`com.interface21.jms.JmsTemplate`类和我们在第9章中已分析过的`JdbcTemplate`使用同一种方法。我们应该设法把必需的各种JNDI查找的复杂性和处理多个JMS API对象的要求隐藏在框架代码中。

下面让我们来看一看`com.interface21.jms.JmsTemplate`类如何支持Pub/Sub消息传递。一开始，`JmsTemplate`实例为`TopicConnectionFactory`做一个JNDI查找。JNDI名称被传递到构造器中，因为它在不同的应用服务器之间可能会有所不同。下面的方法使用被缓存的`TopicConnectionFactory`来调用一个回调接口，以便根据一个给定的`TopicSession`创建一条消息，然后出版这条消息。

```
private static String PREFIX = "java:comp/env/";

public void publish(String topicName, PubSubMessageCreator
    pubSubMessageCreator) throws JmsException {

    if (!topicName.startsWith(PREFIX))
        topicName = PREFIX + topicName;

    TopicConnection topicConnection = null;

    try {
        topicConnection = topicConnectionFactory.createTopicConnection();
        TopicSession session = topicConnection.createTopicSession(
            false, Session.AUTO_ACKNOWLEDGE);
        Topic topic = (Topic) jndiServices.lookup(topicName);
        TopicPublisher publisher = session.createPublisher(topic);

        Message message = pubSubMessageCreator.createMessage(session);
        logger.info("Message created was [" + message + "]");
        publisher.publish(message);

        logger.info("Message published OK");
    }
    catch (NamingException ex) {
        throw new JmsException("Couldn't get topic name [" + topicName + "]", ex);
    }
    catch (JMSEException ex) {
        throw new JmsException("Couldn't get connection factory or topic", ex);
    }
    finally {
        if (topicConnection != null) {
            try {
                topicConnection.close();
            }
            catch (JMSEException ex) {
                logger.logp(Level.WARNING, "com.interface21.jms.JmsTemplate",
                    "publish", "Failed to close topic", ex);
            }
        }
    }
}
```

`com.interface21.jms.JmsException`是一个已检查异常，这意味着我们需要选择是否捕捉这个异常。调用代码可以在不必深入了解JNDI的情况下实现`JmsTemplate`的`PubSub MessageCreator`内部接口（如下所示）：

```

public interface PubSubMessageCreator {
    Message createMessage(TopicSession topicSession) throws JMSException;
}

```

因此，可以使用下列代码出版一条消息：

```

JmsTemplate jmsTemplate = new JmsTemplate("jms/TopicFactory");
jmsTemplate.publish("jms/topic/referencedata",
    new JmsTemplate.PubSubMessageCreator() {
        public Message createMessage(TopicSession session) throws JMSException {
            TextMessage msg = session.createTextMessage();
            msg.setText("This is a test message");
            return msg;
        }
    });

```

消息消费者也可以从一个简化的基础结构中受益。需要注意的是，使用MDB作为消息消费者通常是最佳的，因为EJB容器为JMS消息消费提供了宝贵的基础结构；可是，我们不一定总能使用EJB。我们始终需要实现MessageListener接口。不过，我们不希望必须处理给题目注册监听器的复杂性，因为一个题目也会涉及多个JNDI查找。JmsTemplate类为此提供了一个便利方法，该方法具有签名：

```

public TopicConnection subscribeToTopicNonDurable(String topicName,
    MessageListener listener, String messageSelector)
    throws JmsException;

```

com.interface21.jms.JmsListenerBeanDefinition类的一个自定义组件定义可以自动把一个JMS监听器注册给一个指定的题目。下面这个例子摘自本书的示例应用，其中的com.wrox.expertj2ee.ticket.referencedata.support.DataUpdateJmsListener JMS监听器把数据更新通知给缓存性对象。

```

<bean name="referenceDataListener"
    definitionClass="com.interface21.jms.JmsListenerBeanDefinition">

    <customDefinition property="listenerBean">
        <ref to="com.wrox.expertj2ee.ticket.referencedata.support.DataUpdateJmsListener" />
    </customDefinition>

    <customDefinition property="topicConnectionFactoryName">
        jms/TopicFactory
    </customDefinition>

    <customDefinition property="topicName">
        jms/topic/referencedata
    </customDefinition>

    <property name="cachingCalendar" beanRef="true">
        calendar
    </property>
</bean>

```

这使得我们不用编写低级JMS代码就能实现一个简单的JMS监听器，让它响应事件和刷新另一个应用构件中所缓存的数据。就像下列清单中所显示的那样：

```

public class DataUpdateJmsListener implements MessageListener {

    protected final Logger logger = Logger.getLogger(getClass().getName());
    private CachingCalendar calendar;

```

```
public void setCachingCalendar(CachingCalendar calendar) {
    this.calendar = calendar;
}

public void onMessage(Message m) {
    try {
        logger.info("Refresing calendar in response to JMS message [" +
                    + m + "]");
        calendar.refresh();
    }
    catch (Exception ex) {
        logger.log(Level.SEVERE, getClass().getName(), "onMessage",
                   "Failed to refresh calendar", ex);
    }
}
}
```

这种简单的抽象没有照顾到所有的JMS需求——JMS API包含了诸如事务化会话之类的、给抽象化造成了较多问题的许多特性。不过，它提供了一个满足最常见需要的简单基础结构。

实现业务逻辑

作为一名专业开发人员，读者知道如何用Java编程。但是，在这里要讨论的是怎样装配应用，以及如何把我们的编程技巧集中运用到应用的范围内。因此，关于示例应用的下列讨论不是解决低级细节，比如相邻座位安排算法的实现，而是主要解决怎样配置示例应用的构件，以及它们如何合作。

我们既讨论示例应用如何使用标准J2EE基础结构，又讨论它怎样使用我们在本章中迄今为止所讨论的基础结构。

示例应用的代码可以在本书的配书下载中获得。在下面讨论期间，请根据需要参考该代码，因为提供所有涉及类的一个完整清单是不现实的。

实现示例应用

现在，让我们来看一看示例应用的一些关键的业务逻辑。重点放在两个关键问题上：怎样获得关于艺术流派、节目和场次的参考数据，以及怎样获得关于座位可供性的信息并实现订票过程。

这些用例在第5章中已进行过详细描述，所以笔者在这里只提供一个简明的需求摘要，连同我们在涉及这些需求的前几章中所做出的高级设计决策。

艺术流派、节目和场次数据极少发生变化，而且可以在示例应用的所有用户之间被共享。这些数据的检索不必是事务性的。

订票过程需要事务性的数据存取，而且涉及用户会话状态的保存。我们已经选择把用户会话状态保存在Web容器的一个HttpSession对象中。

定义业务界面

第一步是定义各种必需的业务接口。这些接口不是Web特有的，但它们运行在集中式应

用的Web容器中。

这些接口应该能被实现成带有或不带有EJB，同时又不影响使用了它们的代码。在像示例应用这样的集中式应用中，我们将使用上文中所讨论的Service Locator设计模式来彻底隐藏EJB访问的细节。

保留使用EJB的选择只在一个方面对我们的设计有影响：异常处理。正如我们在第10章中所见过的，由于EJB把未检查异常作为致命的系统异常来处理，所以我们不能在可能由EJB实现的业务接口中使用未检查异常。有时，这需要妥协。

请考虑如下情景：一个请求要求查找某一场次的空座位数，但它的场次ID却与任何场次都不匹配。这样的请求在我们的应用中不可能出现，因为用户只能从提供有效场次ID的页面中使用链接来申请空座位。因此，通过捕捉一个已检查的NoSuchPerformanceException来使应用代码变复杂是毫无意义的。

但是，假如我们希望能够使用EJB来实现我们的BoxOffice接口，就必须选择如下两害中的较小者：使我们的所有应用异常（包括NoSuchPerformanceException）都变成已检查的，并解决在捕捉已检查异常没有意义的地方捕捉它们所引起的系统开销；或者失去在EJB与其他业务对象实现之间透明地切换的能力。在目前的应用中，我们选择使用已检查异常。我们可以使用Business Delegate设计模式，以捕捉来自EJB的已检查异常并重新抛出未检查异常。不过，这需要更多的实现工作量。

在示例应用中，我们需要两个分开的接口来处理我们这里正讨论的功能度：一个接口用来处理参考数据，而另一个接口用来处理座位可供性和订票过程。下面让我们来依次看一看每个接口。

com.wrox.experj2ee.ticket.referenceData.Calendar接口处理参考数据需求：

```
public interface Calendar {
    List getCurrentGenres();
    Show getShow(int id) throws ReferenceDataException;
    Performance getPerformance(int id) throws ReferenceDataException;
}
```

Genre、Show和Performance对象是轻量级的值对象，已与数据源断开了连接。它们暴露一个可向下导航的树结构：从一个艺术流派，我们可以查找到节目，而从每个节目，我们可以得到一个演出场次列表。整个树可以高速缓存长达1分钟时间，或者只要我们知道参考数据没有变化（个别地缓存对象的复杂性是无保证的，因为这种缓存方式几乎不会提供什么好处）。

com.wrox.experj2ee.ticket.boxoffice.BoxOffice接口暴露关于座位可供性的信息，并暴露订票过程。它包括了下列这些方法：

```
public interface BoxOffice {
    int getSeatCount(int performanceId) throws NoSuchPerformanceException;
    int getFreeSeatCount(int performanceId)
        throws NoSuchPerformanceException;
}
```

```
Reservation allocateSeats(ReservationRequest request)
throws NotEnoughSeatsException, NoSuchPerformanceException,
InvalidSeatingRequestException;

Booking confirmReservation(PurchaseRequest purchaseRequest)
throws ExpiredReservationTakenException,
CreditCardAuthorizationException,
InvalidSeatingRequestException,
BoxOfficeInternalException;
}
```

前两个方法暴露座位可供性。`allocateSeats()`方法创建一个预订，而`confirmReservation()`方法把一个现有预订转变成一个购买。这里，几乎没有高速缓存的机会，而且这两个订票方法是事务性的。

`ReservationRequest`对象（用做一个要传递给`allocateSeats()`方法的参数）是一个命令，要传递给`confirmReservation()`方法的`PurchaseRequest`参数同样也是一个命令。使用命令对象有下列优点，其中多个单独参数在其他方面是必需的：

- 可能在不破坏方法签名的情况下增加参数数据。
- 通过使用JMS或一个简单的Observer实现，命令为事件出版（如果需要）提供了极好的基础。例如，当用户成功地进行了一个预订时，我们可以轻松地出版一个事件并在它上面附加一条能使电子邮件被发送的命令。
- 可能把HTTP请求参数值附加到一条命令上。大多数MVC Web应用框架都能自动用HTTP请求参数填充命令对象的JavaBean属性。这可以极大地简化应用请求处理代码。

从`ReservationRequest`命令的下列代码清单中可以看出，该命令只是简单的数据持有者。这个对象是可串行化的，不允许远程调用，但允许远程调用被存储在一个`HttpSession`中，而这个`HttpSession`在聚类中可能需要被复制，或被编写到一个持久性存储器（如果被交换出应用服务器RAM）。

```
public class ReservationRequest implements Serializable {

    private static final long MILLIS_PER_MINUTE = 1000L * 60L;

    private int performanceID;
    private int seatsRequested;
    private double bookingFee;
    private int classID;
    private boolean reserve;
    private Date holdTill;
    private boolean mustBeAdjacent;

    public ReservationRequest() {
    }

    public ReservationRequest(int performanceID, int classID,
        int seatsRequested, boolean reserve, double bookingFee,
        int minutesToHold)
        throws InvalidSeatingRequestException {
```

```
    this.performanceID = performanceID;
    this.classID = classID;
    this.seatsRequested = seatsRequested;
    this.reserve = reserve;
    this.bookingFee = bookingFee;
    holdFor(minutesToHold);
}

public int getPerformanceID() {
    return performanceID;
}

public void setPerformanceID(int performanceID) {
    this.performanceID = performanceID;
}

public int getSeatsRequested() {
    return seatsRequested;
}

public void setSeatsRequested(int seatsRequested) {
    this.seatsRequested = seatsRequested;
}

public boolean getSeatsMustBeAdjacent() {
    return mustBeAdjacent;
}

public void setSeatsMustBeAdjacent(boolean mustBeAdjacent) {
    this.mustBeAdjacent = mustBeAdjacent;
}

public double getBookingFee() {
    return bookingFee;
}

public void setBookingFee(double bookingFee) {
    this.bookingFee = bookingFee;
}

public void holdFor(int minutes)
    throws InvalidSeatingRequestException {

    if (holdTill != null)
        throw new InvalidSeatingRequestException(
            "holdFor is immutable: cannot reset");
    holdTill = new Date(System.currentTimeMillis() +
        minutes * MILLIS_PER_MINUTE);
}

public Date getHoldTill() {
    return holdTill;
}

public int getClassID() {
    return classID;
}

public void setClassID(int classID) {
    this.classID = classID;
}
```

```

public boolean isReserve() {
    return reserve;
}

public void setReserve(boolean reserve) {
    this.reserve = reserve;
}
}

```

业务接口返回值（比如Reservation对象）被编码到接口。这使得实现能够返回一个不可更改的实现，以防止由调用代码操纵它——无意的或恶意的。

确定实现策略

Calendar接口涉及非事务性的数据存取。因此，没有必要使用一个EJB来实现它。我们可以在Web容器中使用DAO。

BoxOffice接口涉及事务性的数据操纵。这样，具有本地接口的无状态会话组件不是明显的实现选择，因为使用CMT可以简化应用代码。

这些决定不影响使用那些对象的代码。因此，如果将来我们希望增加或减少EJB的使用，或使用一种不同的数据存取策略，那么对代码库将只存在有限的影响。

首先来看一看Calendar接口的实现。`com.wrox.expertj2ee.ticket.referencedata.jdbc.JdbcCalendar`类使用我们在第9章中所讨论过的那些JDBC抽象包提供一个线程安全的实现。源代码类似于第9章中所显示的代码。`JdbcCalendar`类实现`com.interface21.beans.factory.InitializingBean`接口，使它能够在它的`afterPropertiesSet()`方法的初始化中装入数据。

由于其他所有应用对象都被编写成使用Calendar接口，而不是使用JdbcCalendar实现，所以高速缓存可以由`com.wrox.expertj2ee.ticket.referencedata.support.CachingCalendar`类来处理，这个类代理底层的Calendar实现（比如JdbcCalendar），并在它的`refresh()`方法被调用时，使用写时复制（Copy-on-write）来构造一个新的Calendar实现。缓存性代理的这种使用对基于接口的设计来说是一种自然用法，而这种用法可以显著地改善性能，同时又保持高速缓存逻辑与应用实现的分离。

实现BoxOffice

BoxOfficeEJB使用两个助手对象：BoxOfficeDAO接口（它把数据存取的细节隐藏在抽象层的后面，我们已在第9章中见过），以及CreditCardProcessor接口（它提供处理信用卡支付的能力）。示例应用使用了接口的一个哑实现；在实际的应用中，一个实现可能会与一个外部的支付处理系统联系。

EJB构件接口`com.wrox.expertj2ee.boxoffice.ejb.BoxOfficeEJBLocal`除了扩展`javax.ejb.EJBLocalObject`接口（这是EJB规范所要求的），还扩展`BoxOffice`接口（它的“业务方法”接口）。

```

public interface BoxOfficeLocal extends BoxOffice, EJBLocalObject {
}

```

本地主接口`com.wrox.expertj2ee.boxoffice.ejb.BoxOfficeHome`指定所有无状态会话组件都必需的单一的创建方法：

```

public interface BoxOfficeHome extends EJBLocalHome {
    BoxOfficeLocal create() throws CreateException;
}

```

组件实现类将扩展我们的com.interface21.ejb.support.AbstractStatelessSessionBean框架类，并实现BoxOffice业务接口：

```

public class BoxOfficeEJB extends AbstractStatelessSessionBean
    implements BoxOffice {
}

```

扩展AbstractStatelessSessionBean类迫使我们实现一个具有ejbCreate()签名的方法，以匹配主接口上的create()方法，我们使用这个方法来获取组件工厂中所定义的助手对象，并把它们缓存在实例变量中：

```

public void ejbCreate() {
    logger.config("Trying to load data source bean");
    this.dao = (BoxOfficeDAO)
        getBeanFactory().getBean("dao");
    logger.config("Data source loaded OK: [" + this.dao + "]");
    this.creditCardProcessor = (CreditCardProcessor)
        getBeanFactory().getBean("creditCardProcessor");
    logger.config("Credit card processing loaded OK: [
        this.creditCardProcessor + "]");
}

```

ejb-jar.xml部署描述符中的相应组件定义使用本章前面所描述的JNDI组件工厂语法，如下所示：

```

<env-entry>
    <env-entry-name>beans.dao.class</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>
        com.wrox.expertj2ee.ticket.boxoffice.support.jdbc.
        JBoss30OracleJdbcBoxOfficeDao
    </env-entry-value>
</env-entry>

<env-entry>
    <env-entry-name>beans.creditCardProcessor.class</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>
        com.wrox.expertj2ee.ticket.boxoffice.support.DummyCreditCardProcessor
    </env-entry-value>
</env-entry>

```

有关这个EJB的完整代码清单，请参见本书的配书下载。但是，现在让我们来看一看confirmReservation()方法的实现，该实现也说明了CMT的使用。这个方法在部署描述符中被标记成运行在一个事务中。如果我们未能成功批准一个信用卡事务，就借助Session Context的setRollbackOnly()方法来使用程序性回退。在尝试信用卡批准以前，先尝试创建一个订票。这使得如下情形不可能出现：用户作为购票者已被记录，却未能获准订票。

这个方法将只抛出已检查异常，因为我们想把异常留给客户：

```

public Booking confirmReservation(PurchaseRequest purchase)
    throws ExpiredReservationTakenException,
           CreditCardAuthorizationException,
           BoxOfficeInternalException,
           InvalidSeatingRequestException {
}

```

在试着批准支付以前，我们先创建一个购买记录，这个记录在以后万一出现了预料之外的故障时创建一个永久性记录：

```
int purchaseId = dao.createPurchaseRecord(purchase, getReservation());  
String authorization = null;
```

如果信用卡事务遭拒绝，我们使用CMT来回退这一数据更新（下列代码中被做了突出处理的部分）。如果出现了支付被接受但我们未能利用批准代码更新购买记录这种不可能的情况（一个致命的内部错误），我们就创建一个日志记录，但抛出一个已检查异常。抛出一个运行时异常会导致事务被回退，回退将会丢失新的购买记录，而这个记录应该仍留在数据库中。

```
try {  
    authorization = creditCardProcessor.process(  
        purchase.getReservation().getTotalPrice(),  
        purchase.getCardNumber(), purchase.getExpiry());  
  
    // Complete data update for purchase  
    dao.setAuthorizationCode(purchaseId, authorization);  
  
    Booking booking = new ImmutableBooking(  
        authorization, new Date(), purchase.getReservation());  
    logger.info("Booking successful [" + booking + "]");  
    return booking;  
}  
catch (CreditCardAuthorizationException ex) {  
    logger.severe("Failed to authorize credit card number");  
    getSessionContext().setRollbackOnly();  
    throw ex;  
}  
catch (DataAccessException ex) {  
    // Failed to authorize: we know we've created pending purchase record  
    // DO NOT ROLLBACK: this will lose PENDING record  
    // This is unusual; of course the tx may not commit, so we want to log  
    // as much information as possible as well  
    String mesg = "***** Database problem: failed to set authorization code "  
        + authorization + " for purchase id " + purchaseId  
        + "; payment WAS accepted for reservation ["  
        + purchase.getReservation() + "] *****";  
    logger.severe(mesg);  
    // Must throw a checked exception to avoid automatic rollback  
    throw new BoxOfficeInternalException(mesg);  
}
```

通过使用带有CMT的EJB，我们简化了我们的应用代码：事务管理只增加了最少的复杂性。

请注意，我们可以捕捉第9章中所描述的com.interface21.dao.DataAccessException通用异常，而又不会依赖使用JDBC来实现BoxOfficeDAO数据存取接口。

使用JMS来传播数据更新

由于效率的缘故，我们选择缓存参考数据，直至我们知道它已被更新时为止。可是，这样的更新必须在聚类中的每个服务器上都进行，所以我们不能使用已内建在我们的应用框

架中的Observer设计模式——必须使用JMS。JMS监听器使得已缓存的数据在消息上被更新成一个数据更新题目。如果Admin接口是可用的，它将在参考数据发生修改时出版JMS消息；在Admin接口被完成之前，管理员将需要访问一个受到限制的URL，该URL在数据库更新完成之后出版一个参考数据更新事件。

请注意，由于已缓存的数据没有被保存在EJB容器中，所以我们不能使用一个MDB作为消息消费者。

我们在本章前面讨论JMS基础结构时，已经见过消费JMS消息的com.wrox.expertj2ee.ticket.referencedata.support.DataUpdateJmsListener类的实现，这个实现调用CachingCalendar对象的refresh()方法，进而引起所有参考数据都从数据库中被重新读取。

装配

由于我们已经拥有了自己的接口和实现，所以可以使用我们的应用框架把它们装配起来。下列代码摘自ticket-servlet.xml应用上下文XML定义文件，说明了我们所讨论过的那些应用构件是怎样被串成JavaBean的。

下列这两个组件定义，定义了一个利用JDBC的Calendar实现和一个在接收到更新事件时就使用realCalendar定义来创建新对象的“缓存性日历（Caching calendar）”。需要注意的是，realCalendar定义的singleton属性被设置成了false，所以每当CachingCalendar申请这个组件定义的一个实例时，就有一个新对象被返回。

```
<bean name="realCalendar"
    singleton="false"
    class="com.wrox.expertj2ee.ticket.referencedata.jdbc.JdbcCalendar" >
</bean>

<bean name="calendar"
    class="com.wrox.expertj2ee.ticket.referencedata.support.CachingCalendar" >
</bean>
```

下面这个自定义组件定义（前文做过讨论）注册一个JMS监听器，这个监听器在参考数据被更新时通知CachingCalendar对象。

```
<bean name="referenceDataListener"
    definitionClass="com.interface21.jms.JmsListenerBeanDefinition">

    <customDefinition property="listenerBean">
        com.wrox.expertj2ee.ticket.referencedata.support.DataUpdateJmsListener
    </customDefinition>

    <customDefinition property="topicConnectionFactory">
        jms/TopicFactory
    </customDefinition>

    <customDefinition property="topicName">
        jms/topic/referencedata
    </customDefinition>

    <property name="cachingCalendar" beanRef="true">
        calendar
    </property>

</bean>
```

BoxOffice组件定义使用前文中讨论过的动态代理组件定义来隐藏EJB访问的复杂性。我们指定了EJB所实现的业务接口和该组件的JNDI名称。

```
<bean name="boxOffice"
      definitionClass="
          com.interface21.ejb.access.LocalStatelessSessionProxyBeanDefinition">

    <customDefinition property="businessInterface">
        com.wrox.expertj2ee.ticket.boxoffice.BoxOffice
    </customDefinition>

    <customDefinition property="jndiName">
        ejb/BoxOffice
    </customDefinition>

</bean>
```

没有必要让应用对象都做JNDI查找或直接使用EJB API：它们可以简单地使用这个组件定义所创建的对象作为com.wrox.expertj2ee.ticket.boxoffice.BoxOffice业务接口的一个实现。

使用本章中所讨论的基础结构，利用了这些接口的应用构件只需要暴露能由框架适当地设置的组件属性：它们不必依赖于框架代码，也不必查找由框架所管理的对象。例如，本例应用的主Web层控制器暴露了calendar和boxoffice属性，框架根据下列组件定义自动设置这两个属性：

```
<bean name="ticketController"
      class="com.wrox.expertj2ee.ticket.web.TicketController" >

    // Other properties omitted

    <property name="calendar" beanRef="true">calendar</property>
    <property name="boxOffice" beanRef="true">boxOffice</property>

</bean>
```

该控制器不必做任何配置查找，但必须实现下列方法来保存指向实例变量的参考。

```
public void setBoxOffice(BoxOffice boxOffice) {
    this.boxOffice = boxOffice;
}

public void setCalendar(Calendar calendar) {
    this.calendar = calendar;
}
```

我们将在下一章中详细讨论Web层。正如本例所表明的，它与我们已经分析过的总体基础结构集成得非常紧密。

小结

在本章中，我们讨论了一个支持应用实现的、牢固而又通用的基础结构的重要性和优点。虽然J2EE应用服务器为J2EE应用提供强有力的标准基础结构，但我们需要使用附加的

基础结构构件来使J2EE API更容易使用，保证应用代码集中精力解决业务问题，以及解决各J2EE规范没有涉及的问题。

为了说明这些要点，并为读者自己的应用提供一个可用的基础，我们分析了示例应用所使用的一些关键基础结构包，以及这些包所要解决的问题。

我们讨论了允许集中式配置管理的基础结构的重要性，而且介绍了怎样把应用构件变成JavaBean使它们可被通用的应用框架来配置，从而消除了让应用代码来实现配置管理（比如属性文件查找）的需要。与基于接口的设计（在这种设计中，代码始终被编写到接口，而不是被编写到类）结合起来，使用JavaBean就成为了一种无需修改Java代码即可装配和参数化应用的强有力手段。

一旦配置管理被集中化，就可能无需对应用代码做任何修改即可从一个源（比如XML文档、属性文件或JNDI）中读取配置。通过配置JavaBean属性保证了整个应用内及体系结构层间的一致性。例如，EJB和普通Java类使用同一种配置管理方法。

我们讨论了能让我们实现集中式配置管理的几个基础结构层：

- 一个抽象了使用反射来操纵JavaBean的包。这是一个很有价值的基本基础结构，它不仅能够用在配置管理中，还能够用于其他任务，比如初始化HTTP请求中的JavaBean对象（就像下一章中所讨论的那样），从而简化Web应用中的请求处理。这个功能度非常有用，所以有几个像Apache Commons那样的开放源产品也提供了它们自己的实现；不过，笔者至今还未见过一个十分优质的产品，本章中所描述的com.interface21.beans包是笔者所知道的最先进的产品。
- 一个使用这个较低级功能度来维护应用对象注册表的“组件工厂”。当应用对象被定义为JavaBean时，它们不必依赖于这个基础结构，基础结构可以幕后地设置组件属性来配置对象，其中包括根据保存在Java代码外部的定义来设置对象之间的关系。如果将对基本属性的内部支持、设置受管理对象间关系的能力以及用标准的JavaBean PropertyEditor功能度把对象类型表示为串的可使用性等几个方面结合起来，简练的组件工厂语法则可以定义复杂的应用配置。我们已经见过以XML文档、属性文件和EJB JAR部署描述符中所定义的环节变量为基础的标准组件工厂实现。
- 一个以组件工厂概念为基础来实现Observer设计模式，提供一种查找消息的标准方法以满足国际化需求，以及允许应用构件在运行时根据需要进行通信的“应用上下文”。

标准的J2EE基础结构使一些简单的事情变得很困难。而这种困难在实践中已经引起了人们对J2EE的失望。这个问题可以通过使J2EE API和服务变得更容易处理的简化性结构来解决。我们已经见过基础结构怎么才能用来简化复杂J2EE API的处理，其中包括：

- 更容易实现EJB和降低常见错误发生率的支持类。
- 帮助实现Service Locator和Business Delegate J2EE设计模式的基础结构，使利用EJB的代码不再依赖于JNDI和EJB API。我们已经见过，Service Locator模式通常是最简单的，最适用于通过本地接口访问EJB，而Business Delegate模式是通过远程接口访问EJB的较好选择。在这种情况下，Business Delegate模式可以用于从客户代码中隐藏远程访问的细节。

- 使发送和消费JMS消息变得更容易的类，隐藏了必需的JNDI查找，简化了错误处理，并消除了与多个JNDI API对象打交道的需要。

无论读者是否选择使用本章中所描述的具体基础结构，所有这些问题都应该在应用代码外部被解决，而应用代码应该把精力集中到问题范围和业务逻辑实现上。

最后，我们了解了示例应用的一些核心实现，说明了如何通过使用适当的基础结构来简化它。

第12章 Web层的MVC设计

在本章中，我们将讨论Web层的设计，以及如何保证Web接口是一个以业务对象和总体应用基础结构为基础的、简单而又可维护的层。

这是一个非常重要的题目。设计不良的Web接口是导致不可维护应用和整个项目失败的捷径之一。经验证明设计不良的Web接口会导致在开发工作中要付出巨大的代价，而首先使笔者对J2EE设计产生强烈兴趣的是业务机会。

在全面考虑了我们在Web应用设计中要尽力避免的各种问题之后，我们将关注MVC体系结构模式，该模式已经证实它对编写成功的Web接口是至关重要的。

在研究了MVC理论及MVC模式的成功实现所共有的各种概念之后，我们将了解3个开放源MVC Web应用框架：Struts、Maverick和WebWork。然后，我们将了解一个与上一章所介绍的基础结构集成在一起的、简单但强有力的Web应用框架，而且我们将在示例应用中使用这个框架。和我们迄今为止所见过的基础结构构件一样，这个框架是为实际应用而设计的，而不仅仅用做一个演示。

笔者选择实现这个框架，而不是使用现成框架有以下几个原因：

- 为了演示把一个MVC Web应用框架集成到一个总体应用基础结构中的可能性，以及这么做所带来的各种好处。
- 为了形式化独立于Web接口的业务对象与Web层构件之间的关系。笔者认为，Web层构件与业务对象之间的关系在MVC应用中常常被疏忽，因而Web特有的应用代码太多。
- 因为笔者在这方面具有相当丰富的经验，而且以前已经实现了一些生产用框架。实现这个框架是一个自然的发展。
- 因为没有任何一个现成MVC Web应用框架满足我们将要讨论的所有设计目标。
- 因为这个框架的设计清楚地阐明了本章的各核心MVC概念，尤其阐明了视图与控制器的完全分离。当我们在下一章中讨论可选的视图技术时，将会看到这种完全分离的各种好处。

当然，这个框架不仅基于笔者的经验，而且也基于现有框架中所使用的各种概念。尤其是，它借用了Struts和Maverick中的一些概念。

我们还将讨论Web应用中的会话状态管理和数据确认的问题。

最后，我们将看一看如何使用本章中所讨论的MVC框架来实现示例应用的Web层，特别关注Web构件怎样访问业务对象。

Web开发的挑战

设计Web接口为什么会如此困难？出错的后果为什么如此可怕？以下是众多原因中的部分原因：

- **Web接口经常变化**

例如，更换商标可能会改变Web应用的视觉效果，但不一定改变工作流程。成功的应用可以容纳这样的修改，同时又无需修改业务对象甚至Web层控制代码。

- **Web接口牵涉到复杂的鼠标**

典型的Web站点含有嵌套的表和很长的JavaScript；页面甚至可能使用产生晦涩置标的GUI工具来设计。通常，只有内容的一小部分是动态的。复杂置标的生成不便生成的动态内容令人难懂是至关重要的。弄懂这种置标是专业人员的职责；Java开发人员不太擅长置标，而且他们的技能最好用在其他地方。因此，区分开置标开发人员与Java开发人员的任务是至关重要的。

- **Web接口使用了与Java等语言中的传统UI有极大差别的模式**

Web程序设计模型所强调的是请求和响应；和Java Swing等传统MVC方法中一样，在底层模型发生变化时，动态地更新GUI构件是不可能的。

- **HTTP请求只能携带串参数**

因此，我们时常需要在串值与Java对象之间来回地做转换。这会很困难，而且容易出错。

- **Web接口使确认用户输入变得困难，因为我们对客户浏览器只有有限的控制权**

我们必须准备拒绝无效的用户输入并让用户重新提交输入。相反，在Swing或VB接口中，我们可以使用自定义的控制来防止用户输入无效的数据，因为我们拥有对UI表示的完全控制权。

- **HTML只提供UI控制的有限选择余地**

同样，这与传统GUI应用正好相反，因为传统GUI应用提供了丰富的控制及实现自定义控制的能力。

- **保证Web站点看起来正确并在所有常见浏览器中都正确工作会很困难**

这明显会使置标变复杂，并加大了区分表示问题与Java程序设计问题的难度。

- **存在许多效率考虑因素**

通常，我们无法限制或预知使用一个Web应用的用户数量。性能和一致性在Web应用中可能都是特别重要的。

- **Web接口的测试相当困难**

这是我们应该保证尽可能少的应用功能度是Web特定的诸多原因之一。

Java Web开发中的教训

设计Web接口来实现可维护和可扩展的应用需要纪律。实现这种纪律的最佳推动力是了解未能实现纪律的各种后果。下面对Java Web开发历史的简要讨论足以说明我们必须避免的各种危险。

纯服务器小程序解决方法的缺陷

第一种Java Web技术是服务器小程序（Servlet）。Servlet API的第一个版本发布于1997年，是对当时盛行的技术——Common Gateway Interface（CGI）（公共网关接口）标准的

一项改进，CGI是一个针对服务器端脚本调用的标准。为了处理每个请求，用C或Perl等语言编写的脚本通常需要让系统创建一个新进程。和脚本不同，服务器小程序允许可重用的Java对象生成动态内容。服务器小程序可以充分利用Java语法的威力（包括JDBC和Java对象模型），因此把Web开发提高到真正的OO开发。

至少从理论上讲，Servlet规范向前迈进了一大步，但开发人员一建立实际的应用就会遇到问题。服务器小程序已证明非常擅长调用业务对象。可是，从Java代码中生成复杂的置标是令人可怕的。如果置标串被嵌入在服务器小程序和助手类中，当修改Web内容的表示时经常需要修改和重新编译Java代码；Java开发人员必须始终被牵涉进来。此外，查看一个完整的页而是如何用散落在整个Java对象内的置标装配起来的也会非常困难，进而使站点维护起来十分困难。服务器小程序已证明在生成二进制内容方面比较有效。不过，这是一个比较少见的需求。

从服务器小程序这样的Java对象中生成置标是非常笨拙的，而且会产生无法维护的Web应用。仅有Servlet规范还不足以满足大多数Web应用的各种挑战。

很明显，需要附加的基础结构。许多开发人员使用模板化（Templating）方法。在这种方法中，置标连同负责只生成动态内容的服务器小程序一起被保存在Java代码外部。可是，不存在已得到广泛采用的标准。

JSP：承诺与诱惑

因此，1998年发布的JSP 0.92规范引起了普遍的兴奋。JSP是一个标准的模板化解决方案，由Sun公司开发。尽管JSP页由Web容器透明地编译成服务器小程序，但JSP页使用了一种与服务器小程序有极大差别的方法。虽然服务器小程序是能够生成置标的Java对象，但JSP页却是容许“必要时放入Java”来启动处理的置标构件。

JSP不久就得到了广泛使用，过于热心的采用引起了大量的新问题。经验很快就证明，基于“必要时放入Java”来生成动态内容的模型是一个取代了旧有需求“从Java代码中生成置标”的新祸害。1999年发布的JSP 1.0是对JSP 0.92的一个明显改进，但没有改动基础模型或解决这些问题。

JSP Model 1体系结构

为了弄清楚这些问题，让我们来看一看适合J2EE Web应用的接口使用的、最简单而又最明显的体系结构。请设想我们正使用JSP页来实现整个Web接口。每个JSP都处理进入请求，用请求参数创建域对象，并使用它们调用业务对象。然后，每个JSP再现业务处理的结果。JSP基础结构通过使用组合了置标和嵌入式Java代码的JSP页，使这种方法的实现变得很轻松。标准JSP基础结构允许JSP页完成下列事情：

- 通过使用<jsp:useBean>标准动作，自动使用请求参数值填充它们所声明的JavaBean。
- 访问并操纵会话属性及应用级的ServletContext属性。
- 使用脚本小程序（Scriptlet）（嵌入在JSP页中的Java代码块）执行某种逻辑。

这种Web应用体系结构经常叫做JSP Model 1体系结构。“Core J2EE Patterns”一书使用了更有意义但相当夸张的称呼Physical Resource Mapping Strategy（物理资源映射策略）。本章中将使用较早的称呼JSP Model 1。

这种方法实现起来较简单。我们需要做的只是编写JSP页，并提供支持类。没有必要编写服务器小程序，或在web.xml部署描述符中定义URL到服务器小程序的映射。开发-部署周期常常很短，因为JSP页可以由Web容器在这些页被修改时自动重编译。

但是，这种方法在实践中多半会产生极差的结果。使用这种方法构造的应用（即使当今，也有太多这样的应用）会面临下列这些严重的问题：

- 每个JSP必须要知道如何处理由一个请求所产生的所有可能路径（例如，无效或缺少参数时的路径），结果导致复杂的条件；要么必须在需要时转到另一个JSP，从而使应用工作流程变得很难跟踪。当预计我们将需要显示什么内容还太早时，就束缚于一个JSP是一个错误。
- 削弱的错误处理。如果在JSP运行的中途，但在缓冲器已被清洗以致转到另一个JSP页太迟之后，我们遇到了一个异常，怎么办？尽管标准JSP错误页机制足以应付微不足道的Model 1系统，但它在这种似是而非的情景中将帮不了我们的忙。
- Web接口束缚于JSP。事实上，JSP只是一种可供J2EE应用利用的视图技术。JSP页不适合表示二进制数据，其他像XSLT那样的技术在解决某些问题方面十分出众。
- JSP脚本小程序中的Java代码测试起来很困难，而且不能被重用。这种方法不仅不是面向对象的，它甚至也不能提供我们在编写完善的过程性应用中所实现的代码重用。
- JSP页将含有混成一团的置标和Java代码，使得这些页无论由Java开发人员还是由置标专家来维护都很困难。
- JSP页与它们所生成的置标几乎没有相似之处。这就使得使用JSP页失去了意义，而JSP页实质上就是置标的组成部分。

JSP Model 1体系结构（JSP页处理进入请求）产生不可维护的应用。JSP Model 1绝不应该用在实际的应用中：使用它的惟一地方是在原型试验中。

JSP标准体系结构的诱惑

如果顺利的话，读者已经确信JSP页不是J2EE Web接口的最终答案。万一读者还没有确信，让我们来看一看JSP标准基础结构的一些缺陷，但这个基础结构让试验许多破坏优良设计的东西变得很容易，也使JSP Model 1体系结构给人的第一印象是似乎合理。

请求参数可以用两种方式被映射到JSP组件上。单个请求参数可以像下面这样，用一个<jsp:setProperty>动作被映射到一个组件属性上：

```
<jsp:setProperty name="bean" property="propertyName" param="paramName" />
```

作为一种选择，所有请求属性可以像下面这样一个单独的<jsp:setProperty>动作中被映射到一个组件上。

```
<jsp:setProperty name="bean" property="*" />
```

这种映射请求参数到组件属性上的JSP机制是数据联编（Data Binding）（稍后详细讨论）的一种不切实际的天真方法。问题如下：

- 根本不存在处理类型不匹配的机制。例如，如果对应一个int属性的某一参数的值是非数值的，映射将会默默地失败。组件上的这个int属性将被保持它的默认值。该组件不能持有用户在任一重提交页面上所提供的无效值。如果不直接检查该请求参数，我们就无法区分出是缺一个参数，还是类型不匹配。
- 确定一个参数是否被提供的唯一方法是把默认值设置成一个带外值。这种方法未必总能行得通。
- 不存在处理由组件的属性设置器所抛出的任何异常的机制。
- 没有办法指定哪些属性是必需的，哪些属性是任选的。
- 我们不能通过截获字段的处理来解决这些问题：我们对怎样执行映射没有控制权。

我们最终将需要在JSP页中编写自己的请求处理代码来解决这些缺陷，因为这些缺陷使属性映射基础结构失去了用处。

映射请求参数到JSP组件上的标准JSP基础结构是原始的，在实际的应用中几乎没有使用价值。

JSP标准基础结构的另一个诱人的特性是在JSP页中定义方法甚至类的能力。通常情况下，脚本小程序中的代码以JSP的`_jspService()`方法结束。通过使用`<%!`语法代替`<%`，我们可以保证嵌入的Java代码能从这个方法中移出来，进而在被生成的类本身中定义方法、变量和类。

JSP是十分有威力的，但它的威力所产生的危险和好处一样多。JSP允许我们做可在视图中正当地做的任何事情，以及最好不要轻易去尝试的许多事情。自从JSP第一次发布起，它一直不断地获得新的能力：最明显的是，增加了能使Java助手代码被隐藏在置标后面的自定义标志（Custom Tag）。但是，基本模型一直没有变化，JSP页擅长显示数据却不适合请求处理和业务逻辑启动的事实也没有变。

达成平衡

迄今为止所讨论的这两种方法——纯服务器小程序和JSP Model 1都具有严重的缺陷，它们不可能实现Java开发人员与置标开发人员角色之间的责任分离。这样的分离在复杂的Web站点上是十分重要的。

但是，通过同时使用这两种技术——用服务器小程序处理控制流程及用JSP页或另一种视图技术再现内容，我们就可以避开笔者所描述过的那些问题。

一个所谓的JSP Model 2体系结构引进一个控制器服务器小程序作为中心进入点，而不是使用单个JSP页作为中心进入点。该控制器负责请求处理，并选择一个JSP来再现与请求参数相关的内容和各种必要业务操作的结果。事实上，正如我们要看到的，这个模型有许多需要改进的地方：重要的是，要避免只拥有一个“上帝”服务器小程序，但保证该服务器小程序与许多子控制器打交道。

笔者不太喜欢把这个体系结构称做JSP Model 2体系结构，因为这个体系结构事实上不依赖于JSP：它使用了一种模板化技术。它的基础是服务器小程序——迄今为止J2EE Web技术当中的一种较重要的技术。笔者将使用“Core J2EE Patterns”一书所使用的术语Front Controller（前置控制器）设计模式。这是一个把MVC结构模式运用于Web应用的尝试（我们将在下文中进一步讨论这个尝试）。

这个设计模式（我们将在下文中详细讨论）是设计可维护J2EE Web应用的关键，也是本章的中心点。

服务器小程序和视图技术（比如JSP）都是建立可维护J2EE Web应用所必需的。服务器小程序和助手类应该用来处理控制流和启动业务操作；JSP只应该用来再现内容。

Web层的设计目标

一个好的Web接口应该具有什么样的特点呢？下面的两个目标是可维护性和可扩展性的首要目标。

干净的Web层

正如JSP Model 1经验所显示的，如果置标生成纠缠地混入了控制流和业务对象调用，Web应用注定要失败。

干净的Web层把控制流和业务对象调用（由Java对象处理）与表示（由JSP页之类的视图构件处理）分隔开。

Java类应该用来控制流程和启动业务逻辑。Java类不应该用来生产置标。应该可以不修改Java类就能修改表示置标。JSP页——或得到使用的模板将只含有置标和简单的表示逻辑。这样就可能使Java开发人员与置标创作者角色几乎彻底分离。

细薄Web层

但是，实现一个干净的Web层是不够的。保证Web层尽可能地细薄也是至关重要的。这意味着除了从用户动作中启动业务处理和显示结果所必需的Java代码之外，Web层不应该再含有任何Java代码。这意味着Web层只应该含有与Web特定相关的控制逻辑（比如选择数据应该被显示在里面的布局），不应该含有业务逻辑（比如数据检索）。

这为什么会如此重要呢？业务逻辑束缚于Web层的应用（甚至像服务器小程序而非JSP页那样的Java对象）不能满足我们在第1章中所确立的那些目标。这样的应用：

- 测试起来困难——一个非常重要的考虑因素。测试Web接口比测试普通Java对象困难得多。如果我们把业务对象暴露为实现定义明确的接口的JavaBean，就可以轻松地编写测试工具，该工具以孤立于用户接口的方式测试业务逻辑。在业务对象暴露定义明确的接口的应用中，测试Web接口是指这样一个比较简单的问题：核实该接口把用户动作正确地转换成正确的业务请求，并正确地显示结果。如果应用的Web层含有业务逻辑，我们就只能通过整体地测试Web接口来测试业务逻辑，而这可能是比较困难的。

- 降低重用业务逻辑代码的可能性，即便是在同一个应用内。例如，服务器小程序的service()方法中的代码即使对其他服务器小程序也是不可用的。
- 可能会证明易受Web接口工作流中的修改所危及（尽管未必易受表示中的改动所危及）。如果我们有一个不同的业务接口层，则可以轻松地随时运行回归测试。如果业务逻辑的一部分在Web层中，修改工作流将可能引入错误。针对Web接口的测试将需要被重新编写，以体现被修改的工作流，从而使测试新工作流变得有问题。
- 不是可插入的。如果我们暴露业务接口层中的业务操作，则可以在不影响调用代码的情况下轻松地修改一个或多个业务对象的实现类（假设我们有一个使基于接口的设计变容易的框架，比如笔者在上一章中所描述的那个框架）。如果业务操作束缚于Web层，就不可能有这样一种直截了当的修改实现的方法。
- 使暴露除Web GUI之外的接口变得很困难，比如Web服务接口。

在一个设计完善的J2EE应用中，Web层应该是一个以定义明确的业务接口为建立基础的细薄层。它应该只负责把用户动作转换成应用事件，以及把用户输入的结果处理成由Web接口显示的结果。

这一点值得强调。尽管一个细薄的Web层和一个干净的Web层一样重要，但Web层与业务对象之间的关系似乎不像保证干净Web层的MVC方法那么引人关注。

MVC概念与Front Controller J2EE设计模式

现在，让我们来较详细地看一看采用MVC方法对Web应用究竟意味着什么。在本节中，我们将看一看理论和一般原则。在下一节中，我们将了解几个真正的MVC Web框架，这些框架可以使用MVC方法简化应用的实现。

概念

让我们首先来看一看基本的MVC概念。

MVC三元组

MVC方法把构件分成3种类型的对象：模型化（Model）数据对象，视图化（View）对象，它在屏幕上显示模型数据，以及控制器化（Controller）对象，它对用户动作做出反应，从而适当地更新模型。

在一个真正的MVC体系结构中，比如在Java的Swing GUI库的MVC体系结构中，每个视图向一个模型进行注册，模型在其数据被更新时出版事件。现在来看一看Swing JList构件。这个简单的视图构件显示一个列表中的表项，并允许表项选择。列表数据被定义在一个实现ListModel接口的对象中，这个接口在数据发生变化时出版ListDataEvents事件。JList视图构件提供一个向ListModel进行注册并在事件被接收到时更新显示的ListDataListener实现。这是一个推送（Push）模型：这种模型把通知推送给若干监听者，而监听者一般（但不一定）是视图。

由于Web应用中的修改通常只在一个请求被接收到和一个新页面被生产时，才被显示给

用户，所以推送模型不适用。因此，Web应用视图将只再现一次。注册这种视图来接收前进中的通知是没有用的。这个问题的解决方法是使用一个回拉（Pull）模型：在这种模型中，视图可以访问再现动态页面所需要的那些模型，并在需要时从模型中拉回数据。由于一个Web页可能含有不同的控制和动态数据的分段，所以可能需要一个以上的模型来支持这个Web页。

无论我们使用推送模型还是回拉模型，MVC体系结构仍提供它的关键好处。每种构件类型都有一个明确的责任。模型不含有视图特定的代码。视图不含有控制代码或数据访问代码，只集中精力显示数据。控制器创建并更新模型；它们不依赖于特定的视图实现。

虽然MVC方法具有显著改善Web层代码的潜力，但设置起来却有点难度。因此，我们一般将使用一个通用的MVC Web应用框架，而不是我们亲自实现该模式。

控制器

由于控制器对Web应用中的MVC来说是中枢神经，所以让我们首先来看一看控制器。

Web应用控制器的主要责任如下：

- **检查和抽取请求参数**

这可能会导致命令域对象的创建，然后命令域对象将被传递给业务对象。MVC框架通常提供简化请求参数处理的基础结构，例如通过用参数值填充JavaBean。

- **调用业务对象，传递从请求中抽取出的数据**

控制器通常捕捉由业务方法调用所产生的应用异常，修改模型创建及视图选择（不过，有些致命异常可以简单地留给Web容器来处理，因为Web应用具有一个标准的“错误页”机制）。

- **创建模型，视图将基于业务方法调用和会话状态显示该模型**

- **在具有服务器端状态的应用中，创建并操纵会话状态**

- **选择一个视图并促使它再现，使模型数据可供它使用**

其他责任可能还包括生成Web层日志输出及实施安全限制（尽管安全也可以用J2EE来声明性处理，就像我们在第6章中已见过的那样）。

由于这些责任是应用功能度（比如调用应用特定的业务对象）和管道工程（比如把模型数据传递给一个命名视图，并让它再现内容）的一个混合体，所以在使用一个MVC Web应用框架时，它们通常在应用特定的与通用的框架类之间被分离开。

控制器对象可以轻松地访问应用业务对象这一点是十分重要的。无论模型对象还是视图对象都不应该需要这样的访问。

模型

模型含有由视图显示的数据。在J2EE Web应用中，模型通常由JavaBean组成。如果模型是JavaBean，包括JSP在内的许多视图技术都非常管用。Web应用中的模型（与真正的“传统”MVC相反）通常是表示一个完整业务操作的结果的唯一存储对象，比如值对象。一旦控制器完成了它的处理，并选择了一个要生成内容的视图，模型就应该含有所有要显示的数据。模型本身不应该执行进一步的数据访问，也不应该与业务对象联系。没有理由让模型依赖于Servlet API或一个Web应用框架；作为模型一部分的JavaBean不必是Web特定的。它们经常是可在Web层外部使用的域对象。

在JSP范例中，模型通过使用`request.setAttribute()`方法，被作为属性添加到请求上。但是，这不是把模型告知视图的惟一方法。不是所有视图都会发现请求属性是有用的。

视图

视图显示模型中所包含的信息，全部承担置标或其他内容生成的工作。视图不必了解控制器或底层业务对象的实现。

JSP是最常用于J2EE应用的视图技术，但正如我们在下一章中将要看到的，目前还有几个有效的替代方案。我们已经说过，JSP给我们提供了一种能力，做不适合在视图中做的一些事情。下列准则指出了什么样的操作在视图中是合适的：

- 视图应该全面负责在给定数据模型的情况下生成Web内容。
- 视图不应该直接处理请求参数。控制器应该截获进入请求，并给视图提供模型。
- 视图不应该执行数据检索（例如，通过运行SQL查询）。视图只应该显示由一个或多个模型所暴露的数据。“Core J2EE Patterns”一书中所描述的Dispatcher View J2EE模式（书中JSP视图在再现时使用一个助手检索数据），从根本上遭到违犯，因为它在再现期间易受意外错误的破坏。“Core J2EE Patterns”一书中把此处所描述的控制器管理式数据检索叫做Service-to-Worker模式（不，笔者也不理解他们怎么会想出这些名称）。
- 视图可能需要执行显示逻辑（Display logic）（与控制逻辑和业务逻辑相对照）。显示逻辑的例子包括以交替颜色显示某个表的各行，或在一个生成的页面上显示附加信息（如果模型含有任选数据）。这样的逻辑不应归入控制器中，因为它是-一个具体表示所特有的。
- 视图不应该必须处理数据检索故障。在实际中，这意味着模型所暴露的属性和方法不应该抛出异常；数据检索应该在视图开始再现之前就已完成。

最后，而又最重要的是：

在一个Web应用中，在不用修改模型或控制器代码的情况下使用另一个显示相同数据的视图替换任一视图应该是可能的。视图应该看做满足一个约定：“显示这种数据”。在不影响应用工作的情况下使用另一个视图替换任一视图应该是可能的。这对保证表示能在不影响工作流的情况下被更新是十分关键的。

在不影响控制器或模型代码的情况下使用另一个利用了一种不同视图技术的视图替换一个视图应该是可能的。例如，使用XSLT视图替换JSP视图应该是可能的。

许多人认为，视图替换在实践中是不可能的——或者过分困难的。笔者不这么认为，并将在本章和下一章中举例说明怎么才能实现视图替换，以及它所带来的各种好处。

当然，除非视图含有正确的链接或表单字段，否则继续使用该应用是不可能的。但是，这是一个显示现有信息的问题，而不是一个显示控制逻辑的问题。

在给定视图构件的条件下，这样有限制而又明确指定的责任，是保证区分开Java开发人员与置标（表示）开发人员角色的最佳方法，而这种区分是在大型网站上是必不可少的。

这种区分对测试也有极大的好处。如果我们知道视图只从控制器所提供的模型中拉回数据，则可以通过修改视图来改变表示，同时又不会有破坏应用运行行为的危险。测试可以

局限于只检查修改过的视图不含有语法错误并显示正确的内容。

单元和回归测试将针对控制器构件来编写，以检查它们启动正确的业务逻辑操作和暴露正确的模型。为Java类比为JSP页或其他再现技术编写单元测试和回归测试要容易得多，所以针对控制器构件编写单元和回归测试非常适用。另一方面，如果我们的“视图”执行数据检索，我们将需要在修改表示时执行回归测试。

控制流

为了说明工作中的这些概念，让我们来看一个简单MVC实现中的一个可能控制流（正如我们将要看到的，真正的MVC Web应用框架要稍微精细一点）。请设想下面这个请求将预订某个节目的某一场演出的某一种特定类型的几个座位。我们的示例应用为此预订使用一个更高级的工作流——下面是一个拆开的示例。

1. 控制器服务器小程序接收一个请求。在该应用的web.xml部署描述符中，有一个从请求URL到控制器服务器小程序的映射。
2. 控制器服务器小程序检查请求参数。performanceId、seatTypeId和count参数是必需的。如果这些参数中的任何一个缺少的或非数值的，则应该给用户发回欢迎视图welcome.jsp。
3. 如果来自请求参数的数据是有效的，服务器小程序就调用它在应用启动时所获得并存为实例变量的一个业务对象上的allocateSeats()方法。
4. 如果这个尝试的预订获得成功，服务器小程序就把返回的Seat对象数组（即模型）作为一个属性添加到HttpServletRequest请求上，并把该请求转发给视图displayReservation.jsp。如果没有足够的座位，服务器小程序就把含有这条信息的异常添加到HttpServletRequest请求上，并把该请求转发给视图retryBooking.jsp。

这个简化了的工作流程，简要地说明了一种类型的请求（通常是针对某个具体URL的请求）怎么才会引发多个视图，以及视图在数据检索完毕之后怎么只需再现内容。

图12.1说明了文中所描述的工作流程。

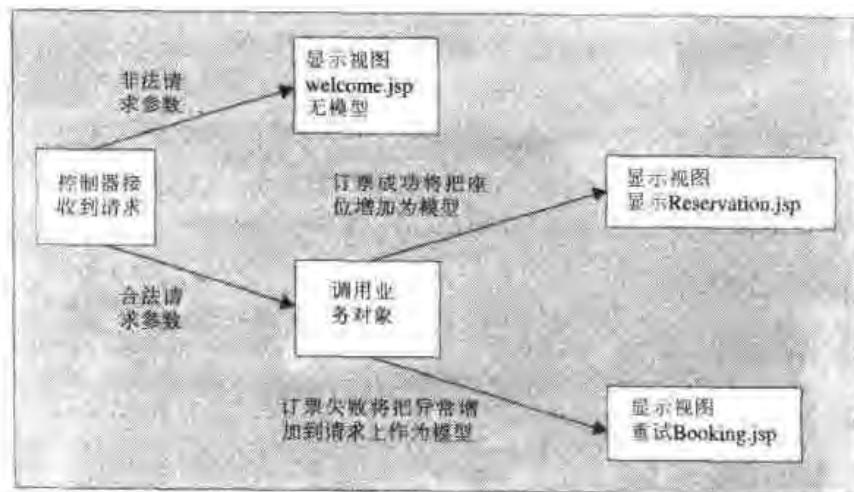


图12.1

模式变种

Front Controller（前置控制器）实现有许多变种。在开始讨论真正的MVC框架之前，让我们先来看一看其中的几个变种。

模板选择服务器小程序

最简单的前置控制器方法，是使用笔者所谓的模板选择服务器小程序（Template Select Servlet）来处理进入请求。每个模板选择服务器小程序处理请求参数，调用适当的业务对象，最后选择一个模板并使模型数据可让它利用（和上述简单预演中的工作流程差不多）。

为了说明这种方法，让我们来看一个流行的实现。我们将要在下一章中较详细地讨论的WebMacro模板解决方案，为使用WebMacro模板作为视图的模板选择服务器小程序提供了一个方便的通用超类org.webmacro.servlet.WMServlet。

WMServlet的应用特有子类需要实现下面的方法来处理请求，启动必要的业务操作，使模型数据可让WebMacro利用，以及返回选定的WebMacro Template来生成内容。请求与响应对象通过WebContext参数可获得。

```
public Template handle(WebContext context) throws HandlerException
```

为了使用这种方法，示例应用中的每个请求URL都必须被映射到web.xml中的一个单独模板服务器小程序上。从积极的方面看，这意味着没有必要用一种框架特有的方法把请求URL映射到处理器。从消极的方面看，web.xml中的映射是罗嗦的，而且我们需要为每个URL都定义一个新的模板服务器小程序实例。

请注意，通过把JSP用做模板化技术，我们可以轻松地运用这种方法；每个服务器小程序都可以简单使用一个RequestDispatcher对象来发送到适当的JSP。不需要WMServer类所提供的那种基础结构。

这是一种简单、有效的方法，也是在纯服务器小程序或JSP Model 1策略上的一大进步。但是，这种方法不能实现我们已确定的所有目标，笔者也不推荐这种方法。尤其是：

- 控制器（在这种方法中，被实现为模板选择服务器小程序）依赖于一种特定的视图技术。如果我们决定从WebMacro改换到XSLT或PDF视图，就会不走运。这可能会证明是一个严重的问题，例如，如果我们需要生成二进制输出。
- 我们必须直接处理请求参数。没有任何一个总体框架提供一个比Servlet API具有更高级别的工作流程。
- 没有能帮助我们实现一个细薄Web层的应用框架。

有多少个控制器服务器小程序

MVC拥护者中对一个应用应该拥有多少个服务器小程序存在争论。我们应该为每种请求类型都使用一个服务器小程序（和模板选择服务器小程序方法中的相同），还是使用单个的前置控制器来处理针对某一应用或子系统的所有请求？“Core J2EE Patterns”一书把后面这种方法叫做Multiplexed Resource Mapping Strategy（多路化资源映射策略）。如果我

们使用一个统一的前置控制器，它应该是通用的，并且应该把请求转发给若干个子控制器。

在反对使用一个统一的控制器服务器小程序的理由中，许多理由是站不住脚的——比如它会引进一个单故障点的想法（线程会死亡，但对象不会），或控制器将太大的想法（我们可以使用一个带有许多应用特有委派的通用控制器，而不是使用一个巨大、全知的控制器）。一个站得住脚的批评是需要重复URL到请求处理类的标准映射。虽然我们可以直接把URL映射到web.xml中的服务器小程序上，但是在我们使用单个控制器来应付许多URL时，我们通常将把所有的URL都映射到该控制器上，然后该控制器必须使用它自己的映射格式在它的子控制器当中定向URL。不过，这在实践中不是一个真正的问题。

使用统一控制器的优点包括它保证控制流的一致性——例如保证必要的资源可让所有子控制器利用，这也意味着每个子控制器不必是一个服务器小程序（非常重要的一点，因为我们可以允许子控制器实现一个不太复杂的接口）。

人们自然要问的一个问题是：采用通过一个统一的控制器来灌进请求所附加的系统开销会产生什么样的性能影响。一个典型的控制器服务器小程序实现需要基于每个请求的URL做散列表查找。有些框架使用反射（尽管一个有效率的框架应该高速缓存反射的结果）。令人高兴的是，性能影响是可以忽略不计的。每个请求都要做一次的事情对一个Web应用的总体性能将不会产生可察觉出的影响。在运行于MVC框架内的仿形应用中，笔者始终发现框架开销是可忽略不计的——甚至是无法觉察到的。

笔者知道的所有实际框架都为整个应用或一组相关用例使用一个统一的控制器服务器小程序。然后，控制器服务器小程序把工作委托给助手类，该助手类实现一个框架特有接口或扩展由框架提供的基类。

JSP还是服务器小程序控制器

另一个关键的问题是：控制器应该是一个服务器小程序，还是一个JSP。“Core J2EE Patterns”一书把这两者分别叫做Servlet Front和JSP Front策略。有些Sun示例示范了JSP作为控制器的用法。

JSP Front方法是幼稚的，误入歧途的，而且不值得被尊为一个“策略”。JSP页非常不适合用做控制器，原因有许多。例如，它们不是第一等的Java类（我们不能子类化一个JSP，所以也不能自定义一个JSP的行为）。它们也不鼓励代码重用。

对使用JSP控制器惟一有利的理由是它们实现起来稍微容易一点，因为我们回避了web.xml服务器小程序映射：一个不值得为其构造一个体系结构的微小节省。

服务器小程序在J2EE Web应用中是必不可少的；JSP对视图来说只是选择之一。企图使用JSP页来执行一项没有发挥它们的长处却暴露它们的许多短处的任务是不合逻辑的，更何况Java类又特别擅长于这种类型的控制流。

不要使用JSP页作为控制器。JSP的用途是生成置标，而不是处理控制流。

请求应该导致命令的创建吗

一种常用的方法是让一个请求导致一个命令对象（Command object）的创建；命令对

象就是一个含有与该请求相关联的数据但对Servlet API又不依赖的业务对象。这种方法具有下列优点：

- 我们可以享有Command设计模式的各种优点，比如排队、记日志和撤消命令的能力。
- 我们或许能够使用从请求到命令组件属性的透明数据联编。这可以使应用代码免去处理请求参数的需要。
- 命令不依赖于Servlet API，所以能被传递给业务对象。这有利于服务器小程序与业务对象之间的明确交互。
- 这种命令方法与Observer设计模式合作得很好；接收到命令时出版事情很容易。

缺点是这种命令方法在简单的情况下可能有点大材小用。例如，如果一个请求没有与之关联的参数，创建一个“空”命令来表示它就没有意义。

一条命令应该在一收到一个请求时就马上被生成，如果数据联编（稍后讨论）将证明是有用的（例如，如果有许多请求参数），或者使用Command设计模式有真正的业务价值。在后一种情况中，Command设计模式的使用应该由相关业务接口来驱动；正是业务对象决定了应用内部通信中将要使用的设计模式，而不是Web接口。

实现的目标

使用Web应用框架应该能简化应用代码，并有助于接口构件与业务对象之间的明确交互。

Web应用框架应该允许我们不必修改控制代码就能使用任一视图技术。例如，应该可能不修改Java代码就能使用XSLT格式表代替JSP页。

人们对MVC存在许多担心。特别是，开发人员常常认为MVC方法与JSP解决方案相比会增加复杂性，而且MVC解决方案运行速度慢。第一个担心是明显的错误，而第二个担心在实际的应用中是不相干的。

在不重要的应用中，如果使用MVC框架，结果应该是更容易编写与调试的比较简单的代码。一个好的MVC框架应该减少开发人员需要编写的代码量，并简化应用代码。即使开发人员编写了他们自己的框架（应该没有必要），MVC框架实现起来也不特别复杂。

MVC应用比JSP解决方案做的工作稍微多一点，但多出的工作量通常仅限于两个隐藏在框架内的散列表查找，并且这两个查找在应用性能方面不会产生可察觉到的变化。成功的MVC框架都将是高效率的。

使用MVC框架有可能会改善性能：通过让我们有机会使用在特定情况下可能比JSP具有更佳性能的视图技术。在第15章中，我们将会见到JSP不一定是再现内容的最快方法。

Web应用框架

由于MVC Front Controller设计模式几乎得到了普遍的拥护，所以目前有几个可用的实现。

虽然MVC是迄今为止最流行的产品，但它不是构造Web应用的惟一方法。有些产品，比如Tapestry框架（请参见<http://tapestry.sourceforge.net/doc/api/overview-summary.html>）采取了一种不同的方法来解决MVC模式所解决的各种问题。不过，还没有任何产品得到了普遍采用，所以笔者在这里将不讨论它们。

尽管实现控制器框架不是特别困难，但读者未必就有实现控制器框架的理由。现在，让我们从模式理论转向实践，首先去看一看真正的框架做些什么，然后去了解几个实现。

常见的概念

大多数MVC Web应用框架都使用以下概念：

- 用于应用的某一部分或整个应用的统一通用控制器。所有应用URL在标准的web.xml部署描述符中被映射到框架的控制器服务器小程序上。
- 使控制器服务器小程序能够选择一个委托来处理每个请求的专有映射。这样的映射通常也基于请求URL。委托将实现一个框架特有API。笔者在下文中把这样的委托叫做请求控制器（Request controller）：控制器服务器小程序实际是一个元控制器。
- 笔者所谓的命名视图（Named view）策略。视图由名称来标识，使请求控制器能够在选择一个视图时指定视图名称，而不是指定视图实现。框架基础结构将解析视图名称。这将把视图实现与控制器分离开。对于一个具有特定名称的视图，它的实现可以在不影响控制器代码的情况下进行修改。

许多框架把命令的角色（上文讨论过）添加到模型、视图和控制器的角色上，可是，这4个角色不一定由各自的对象来扮演。在Struts和用于示例应用的应用框架中，这4个角色通常是各自独立的，而控制器在整个应用生成周期内都能被重用。在WebWork和Maverick中，命令、控制器和模型通常是同一个对象，被创建用来处理一个单独的请求。

与专门使用一种MVC方法相比，选择一个Web应用框架是一个次要问题：甚至连一个拙劣的MVC实现，都将比一个聪明的JSP Model 1实现更擅长于把表示与业务逻辑分离开。现有框架共用了如此多的常见概念，所以只要学会了一个框架，使用其他框架就会相当容易。

需要记住的是，Web层应该很细薄，使用了框架的代码不应该复杂。如果大部分应用代码依赖于某一特定框架，该框架的设计无疑很糟糕，或使用得不正确。

现有框架

在本节中，我们将简要介绍几个领先的MVC框架。

Struts

采用最广泛的MVC框架是开放源的Apache Struts (<http://jakarta.apache.org/struts/>)。Struts最初由Craig McClanahan——Tomcat服务器小程序引擎的主要开发者所编写，并于2000年早期发布，因而使它成为了得到公认时间最长的开放源Web应用框架。部分由于它有相当长的历史，所以Struts已经成为了可以让开发人员购买的出色开放源产品，而且目前有围绕它的许多附件。在撰写本书的时候，Struts版本1.1正处于B版的后期。因此，我们将同时讨

论版本1.0和1.1的能力，但重点放在版本1.1上，因为版本1.1含有显著的改进，并应该用在使用Struts的新项目中。

目前有几本图书和许多篇在线文章专门谈论Struts，比如由Manning出版的“Java Web Development With Struts”一书（ISBN 1-930110-50-2），所以笔者不打算在这里详细描述它，而只分析它怎样处理我们已经讨论过的概念。

Struts为整个Web应用或一个Web应用的一个子集使用一个统一的控制器。Struts 1.0只允许每个应用一个控制器：一个严格的限制，因为它意味着这个统一控制器的配置将大得难以管理。Struts 1.1允许多个控制器，其中每个控制器都具有它自己的配置文件。一个Struts控制器具有具体框架类org.apache.struts.action.ActionServlet。这个标准控制器服务器小程序负责处理所有进入请求，并通过WEB-INF/struts-config.xml文件来配置（正如笔者在前文中所说的，许多框架把XML用于它们的配置）。

和大多数框架中一样，这个控制器服务器小程序是通用的。它不含有应用特有代码，子类化它通常不是必需的（至少在Struts 1.1中如此；子类化在Struts 1.0中时常是必不可少的）。该控制器使用它的映射信息来选择一个能处理每个请求的动作。一个Struts动作扩展org.apache.struts.action.Action类。最重要的方法是下列这个方法，它在每个请求上都被调用：

```
public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception;
```

这个方法在Struts 1.0中被命名为perform()；在Struts 1.1起，perform()遭到反对。

一个Struts动作必须是线程安全的：由于动作实质上是扩展这个控制器服务器小程序的功能度，所以线程化问题和编写服务器小程序中的线程化问题完全相同。

Struts资料建议，动作不应该包括业务逻辑，而“应该调用另一个对象（通常是一个JavaBean）来执行实际的业务逻辑。这允许动作把精力集中在错误处理和控制流程上，而不是集中在业务逻辑上”。但是，Struts没有提供管理业务对象和暴露它们给Web层动作的基础结构。

传递给动作execute()方法的两个Struts特定参数（ActionMapping和ActionForm参数），传达了核心的Struts功能度，所以让我们较仔细地看一看它们。

ActionMapping参数含有从请求到ActionForm实例的相关映射信息：最重要的信息是能使ActionForm在运行时返回一个命名视图的“转发”的定义。

ActionForm参数是一个Java组件，扩展org.apache.struts.action.ActionForm类来暴露对应于请求参数的组件属性。ActionForm组件由Struts用请求参数自动填充。Struts使用Apache Commons组件操纵包，这个包和我们在上一章中所讨论过的com.interface21.beans组件包具有类似的目标。不过，Apache Commons组件包不太复杂，尤其是缺少好的错误处理。

Struts ActionForm表单具有我们必须加以考虑的几个特色。由于所有ActionForm表单都必须扩展Struts超类，所以它们都依赖于Struts和Servlet API。这就意味着它们不能被传递给业务对象，因为业务对象不应该依赖于某一具体Web应用框架或Servlet API。其次，可能含有无效数据（比如用户可能为其输入了非数值数据的数值输入）的请求参数都必须用String

型的ActionForm属性来匹配。幕后的组件操纵代码尝试把串请求参数转换成适当的类型，但在企图设置属性方面的任何异常都会导致org.apache.struts.util.RequestUtils类通过抛出一个ServletException异常来使该请求失败。

因此，一个Struts ActionForm表单不是一个真正的域对象。它是一个保存用户数据的地方，用户数据能一直被保存在这里，直至它能通过确认并被传入到命令之类的域对象中为止。这种方法的优点是能轻松地显示无效数据；缺点是我们将需要在某个时候使这个数据进入到真正的域对象中，所以Struts只把我们带到了通往真正数据联编的半途。

从一个统一的超类中派生ApplicationForm表单的需要对笔者来说似乎始终是一个设计缺陷。这个缺陷不仅使命令束缚于Struts框架和Servlet API，而且通过用请求参数做数据联编来进行更新还不正确地暴露了继承来的框架属性。例如，添加一个具有串值的servlet参数将会破坏由Struts 1.0应用所生成的几乎任何页面（因为把一个ActionServlet型的属性设置成一个串会导致失败）。Struts 1.1为这个特定问题引进了一种回避方法，但这个问题的根源是整个ActionForm概念。

ActionForm类还定义了一个具有下列签名的validate()方法：

```
public ActionErrors validate(ActionMapping mapping,
                           HttpServletRequest request);
```

子类可能会超越这个方法，以确认一个动作表单在用请求参数做完填充之后的状态。Struts还提供了状态确认的替代方案。我们将在本章的稍后部分概括性地讨论状态确认，并详细地讨论状态确认的Struts方法。

在Struts模型中，一个标准的控制器服务器小程序（即Struts ActionServlet）代表若干个Action对象。每个Action对象作为一个参数被赋给一个ActionForm对象，而每个ActionForm对象含有用反射从请求中预填充的Java组件属性。由于ActionForm对象被束缚于Struts框架，所以它们不能用做一个应用中的命令。每个Action对象负责启动处理请求所必需的各种业务操作。在处理结束之后，每个动作都应该返回一个映射后的视图或“动作转发”。

Struts还带有几个JSP标志库来处理数据联编和其他操作。需要注意的是，这些标志库中的有些库已被新的JSP Standard Tag Library (JSTL)（在下一章中讨论）所替代。

重要的Struts标志库包括：

- **Bean**

执行如下任务的其他标志：暴露一个cookie值为一个请求变量，基于头部值、请求参数和组件属性来定义脚本变量，以及调用本应用内的另一个资源并把生成的输出保存到一个串中。

- **HTML**

帮助构造要提交给Struts动作的表单的标志。这些标志可以使用一个支持组件值来填充表单字段，进而简化JSP代码。这个库中的标志（如<html:text>）用来替代HTML输入标志；这些自定义标志将暴露所有必要的属性，并生成必需的HTML。虽然自定义标志对填充像下拉菜单和单选按钮之类的一些HTML控件是必不可少的，但使用

Struts控件替代标准HTML控件会使HTML理解起来更费劲，用GUI工具编辑起来更困难。笔者更喜欢一种其内的标志提供数据值但不生成置标的方法。

- **Logic**

不用脚本小程序就能做比较和迭代的条件标志。JSTL标志可能会取代这些标志中的某些标志。

- **Template**

建立一个能通过指定每节中的内容来参数化的公用布局的标志。

请注意，所有Struts标志都能访问嵌套属性。例如，传递给一个标志的属性路径spouse.age将导致getSpouse().getAge()求值。JSTL Expression Language（在下一章中讨论）消除了Web应用框架中对这种支持的需要。

目前，还有各色各样的附件式标志库，尽管不是所有的标志都依赖于Struts框架本身。各色各样相关标志库的可用性是Struts获得成功的一大原因。

不管Struts的声望有多大，但笔者绝不是一个Struts迷。Struts相当好，但距离一个理想的J2EE Web应用框架还有很大差距：

- ActionForm方法（Struts请求处理模型的核心）是很差的。组件联编是原始的，也就是说，只有串属性是真正有用的。与简单地查询请求参数相比，这几乎没有增加多大的价值。从动作表单中把属性复制给命令对象的构思是不完善的，而且根本没有对类型检查的支持。
- Struts太面向JSP，尽管和其他模板化技术一起使用Struts是可能的。
- Struts几乎完全基于具体类。这使得自定义Struts的行为很困难。
- 虽然有些东西在版本1.1中已经有了明显改进，但Struts代码库很差。伴随着Struts 1.1的诞生，一直有大量的责难。

Struts 1.1纠正了Struts 1.0的许多（但并非全部）缺陷，比如允许多个控制器服务器小程序在同一应用中使用（不过，用于这方面的机制不是十分精致——它明显是一个后添加的事物）。Struts 1.1中的其他改进包括org.apache.struts.actions.DispatchAction超类的引进，允许同一个类执行几个动作。这在许多请求类型要求简单处理的情况下是十分有用的；它避免了许多次要动作类的增殖。我们将在下文中进一步讨论这个概念。Struts 1.1还引进了声明性异常处理（Declarative exception handling）：稍后将要讨论的另一个概念。

Maverick

另外一个开放源框架是Maverick (<http://mav.sourceforge.net>)（该名称也隐含了MVC的意思）。这个框架致力于实现MVC工作流程，并且没有提供标志库之类的表示支持。两名主要Maverick开发者之一的Jeff Schnitzer，也是我们在第3章中所讨论的JUnitEE测试框架的创作者。现在，Maverick的最新版本是2.1，本节将要讨论的就是这个新版本。版本2.0是原始模型的一个重要修订。Maverick版本1.0于2001年早期发布。

和Struts和大多数其他框架一样，Maverick使用一个统一的控制器服务器小程序作为进入点。一个Maverick控制器服务器小程序具有org.infohazard.maverick.Dispatcher类。所有应用URL都被映射到web.xml中的这个服务器小程序上。控制器服务器小程序从Maverick特定

的WEB-INF/maverick.xml配置文件中获得它的配置。Maverick目前只允许每个应用有一个控制器服务器小程序——因此也只允许有一个配置文件。和Struts 1.0中一样，这会使大型应用中的配置难以管理。

提供Maverick的一个简短概述是很困难的，因为Maverick是高度可配置的（它的可配置程度大于Struts）。基本工作流程可以由用户自定义，而且Maverick提供了Web层应用能够对它们进行扩展以提供不同工作流程的几个标准类。下面的讨论描述了Maverick 1.0所提供的模型，而且从这个模型在“FriendBook”示例应用中的使用来看，它在Maverick 2.0中似乎仍得到提倡的模型。这个模型以“废品”控制器的使用为基础。

Maverick不同于Struts，因为Maverick请求控制器通常是JavaBean（Java组件）。在这个模型中，一个新的“废品”控制器实例被创建用来处理每个请求（但是，Maverick 2.x还支持Struts所使用的可重用控制器）。在这个模型中，Maverick不区分控制器（类似于一个Struts动作）与命令（类似于一个Struts ActionForm）。这具有这样一个优点：每个新控制器不必是线程安全的——它不必支持并发调用。但是，这会导致控制器实例的增殖。

请求的处理按照如下所述进行：

- 一个新的org.infohazard.maverick.flow.Controller对象通过使用反射得到创建（该控制器必须有一个无变元构造器）。通常，应用控制器扩展由Maverick框架所提供的org.infohazard.maverick.ctl.ThrowawayBean2超类。
- 控制器上的组件属性由org.infohazard.maverick.ctl.ThrowawayBean2超类通过使用反射从请求参数中进行设置。控制器组件本身可以确认已得到成功设置的属性；类型不匹配时将会产生一个致命的ServletException型异常（意味着我们需要像思于Struts一样忠于Maverick，以保证有机会确认所有输入），组件属性填充使用Struts所使用的同一个Apache Commons BeanUtil包来进行。尽管和可选组件填充策略一起使用Maverick是可能的。
- 由org.infohazard.maverick.ctl.ThrowawayBean2超类所定义的perform()保护方法被调用，而且在调用了业务对象之后，应该返回Maverick配置文件中所定义的视图的串名。每个控制器都可以访问org.infohazard.maverick.flow.ControllerContext对象，并且它可以从这个对象中获得服务器小程序请求和响应：ServletContext和ServletConfig。但是，在正常情况下，控制器的填充式组件属性应该含有控制器工作所需要的大部数据。

当控制器的处理完毕时，控制器必须通过调用ControllerContext对象上的setModel()方法，来设置视图所显示的模型对象。因此，该模型被限定在单个对象上。为了返回多个对象，定义一个包含所有必要数据的容器对象是必不可少的。org.infohazard.maverick.ctl.ThrowawayBean2超类把它自己返回为这个模型，把我们前面所讨论的4个对象角色的另外两个角色组合起来。

每个命名视图必须实现org.infohazard.maverick.flow.View接口，该接口提供以一致方式把模型数据传递给视图的、与视图技术无关的方法。视图分解由调用控制器的org.infohazard.maverick.flow.Command类来处理。View接口含有下面这个唯一的方法，必须实现这个方法

来再现给定模型数据的输出：

```
public void go(ViewContext vctx) throws ServletException, IOException;
```

ViewContext接口通过getModel()方法来暴露模型数据，而且还暴露请求和响应对象。getModel()方法返回经由控制器所暴露的模型对象。

随同Maverick一起提供的View接口的标准实现，包括一个在转发请求给JSP页之前先把模型作为属性添加到请求上的实现，以及一个把模型数据（被转换成串形式）传递给GET查询串中的一个外部URL的重定向视图。还有对WebMacro和Velocity模板引擎的支持。由于这个视图接口是如此简单，所以可以轻松地实现自定义视图。

Maverick的视图替换方法简单而又精细，而且笔者在下面要讨论的框架中借用了这个视图替换的概念。模型的明确分割有利于养成良好的习惯：它不鼓励请求属性的随意设置，以免提供一个杂乱的、纯JSP的模型。

最有趣的Maverick概念，可能是JavaBean数据到XML的透明“变换”或转换，目的是为了飞行地支持XSLT格式表的使用（Jeff Schnitzer是XSLT视图的坚定拥护者，对XSLT视图的拥护胜于JSP视图）。Maverick的变换库现在是一个单独的开放源产品。我们已经在第10章中讨论过变换的概念；令人惊讶的是，这种方法在实践中非常有效。Maverick还支持一个“可配置”的转换“管道”，这个管道和XSLT一起使用会很有用，但与其他视图技术不太相干。

另一个有趣的特性是基本调度者功能度的使用；在使用这个功能度时，不同的工作流程可以通过扩展随同该框架一起提供的多种超类来嫁接。这使得Maverick能够在不必子类化控制器服务器小程序的情况下支持各种各样的习惯用法。

从消极的方面看，“废品控制器”模型中命令与控制器之间缺少分离，是导致Struts动作表单所遇到的同一个问题的不同方法：在这里，命令依赖于Servlet与Maverick API。这就意味着在Web层与业务接口之间使用一种干净的分离是很困难的。例如，我们不依靠Maverick API就无法确认命令数据的有效性，使实现非Web特定的数据有效性确认变困难。为处理每个请求而新建一个控制器实例的做法也使得参数化控制器变得十分困难；单个可重用对象可以根据需要暴露任意多个只被设置一次的配置属性。

Maverick是Struts的一个合格替代品。Maverick一般用来创建一个新的控制器以处理每个请求（尽管Maverick还支持其他模型）。控制器是Java组件，框架在幕后透明地设置组件属性。Maverick使用起来相当容易（尽管Maverick 2.x模型比Maverick 1.0模型复杂得多），而且内部地支持XML节点从Java组件模型中的透明生成。

WebWork

一个较新的框架是WebWork——由著名J2EE开发者Richard Oberg所设计的另一个开放源产品。Richard对JBoss和其他项目也有过贡献。WebWork 1.0（本节要讨论的版本）于2002年早期发布（<http://opensymphony.com/webwork/>）。

无论WebWork使用什么样的名字，它不是一个纯粹的Web应用框架。它采用了一种最大限度地减少应用代码与Web概念的相关性的聪明方法。

WebWork以Command设计模式为基础。Struts动作是一个类似于服务器小程序的、可重用的、线程安全的对象，而WebWork动作却不同于Struts动作，它是一个命令，被创建用来处理一个特定的请求（和Maverick中相同的基本方法）。动作就是一个Java组件，因而允许它的属性被自动设置。WebWork动作必须实现webwork.action.Action接口，这个接口含有下列方法。

```
public String execute() throws Exception;
```

隐含输入是动作的组件属性的状态。返回值是一个表示结果的串——通常是应该用来再现输出结果的那个视图的名字。数据输出（即模型）就是动作组件属性在execute()方法被调用后的状态。因此，WebWork组合每个动作对象中的命令、控制器和模型的角色。

控制流程的工作过程如下：

- 进入请求由主调度者服务器小程序webwork.dispatcher.ServletDispatcher来处理（请注意，这不是调用WebWork动作的唯一方法：不同的控制流程是可能的）。
- 一个新的动作实例被创建。实现并设置一个自定义“动作工厂”来创建动作是可能的。默认情况下使用Class.newInstance()。
- 命令的组件属性被填充。在Web应用中，这种填充将从请求参数中发生；但是，基本方法应该是使用接口（做这件事情的方式因穿过一系列ActionFactory实现而变得相当复杂。webwork.action.factory.ParametersActionFactoryProxy处理组件属性串）。属性设置期间的类型不匹配会由WebWork记录下来，以便于重新显示；保存此类异常的这种能力是对Struts和Maverick的相当原始的组件属性填充的一项明显改进。
- 命令的“上下文”被设置。WebWork上下文（具有webwork.action.ActionContext类型）与当前线程相关联。不过，这种做法不被提倡，因为它意味着这样的动作只能用在Web接口。因此，WebWork动作不一定依赖于Servlet API（尽管ActionContext类依赖于Servlet API——最好是只让ActionContext类的Web特定子类依赖于Servlet API）。动作没有必要知道它的上下文，尽管它可以通过调用webwork.action.ActionContext.getContext()静态方法来查找它的上下文。在调用时供给上下文的方法类似于上下文对象向EJB实例的供给。
- 动作的execute()方法被调用。调用期间抛出的任何异常都将被作为一个系统异常来对待（结果导致一个ServletException异常），但webwork.action.ResultException异常除外，这种特殊异常似乎准备提供一个备用的返回值。

和Struts一样，WebWork带有它自己的JSP标志库。不过，WebWork依赖JSP不像Struts那么严重。WebWork也支持Velocity模板引擎（在下一章中讨论），而且提供了一个与Maverick的XML变换等效的功能度。

WebWork被设计和实现得比Struts更精致。它的工作流程更类似于Maverick的工作流程。ActionContext构思是独创的和聪明的。WebWork帮助最大限度地减少应用代码对Servlet API的依赖性，并提供更有利向优良设计方向发展的诱因。但是，它的缺点包括：

- 在没有大量的请求数据时，在每个请求上都创建一个动作可能有点浪费。和Maverick中的情形一样，配置许多新动作比仅配置单个可重用动作要困难得多。让新对象访问业务对象可能也比较困难。

- WebWork给每个用户交互都强加Command设计模式，无论是不是合适。
- 异常处理需要面对Command设计模式的典型问题之一：我们不知道某一特定命令可能会抛出什么类型的异常。因此，execute()方法被迫抛出Exception。
- 动作与Web层概念的分离可能不现实或值得。基于请求-响应Web范例来强行抽象过度地抑制了与传统GUI（如Swing）的更复杂交互。

WebWork框架以Command设计模式为基础，结果导致了一个与Maverick的模型更相似的基本模型（与Struts的模型相比）。WebWork企图把用户动作建模成接口无关的、不依赖于Servlet API的命令。

Web应用框架到总体应用体系结构中的集成

一个好的Web应用将帮助我们实现我们的第一个目标——干净的Web层。但是，这样的Web应用不保证我们实现第二个目标——薄薄的Web层。

我们在前文中所讨论的所有3个框架都能使我们彻底清理Web层，从而把表示与内容分割开。但是，在集成我们的Web层与设计完善的体系结构方面，没有一个框架能给我们更大的帮助。

WebWork帮助我们创建接口无关的命令；但是，它没有提供一个能使这些命令以标准方式调用业务对象的体系结构。

Struts能使自定义的“插件”在struts-config.xml文件中被定义为一种使Struts Web构件能够访问其他应用对象的手段。插件是用于用户特有应用构件的配置包装器，并且必须实现org.apache.struts.action.Plugin接口。Struts插件可以暴露简单的组件属性，而这些属性可以像摘自Struts 1.1示例应用的下列示例一样，用struts-config.xml文件中的元素来填充。

```
<plug-in
  className="org.apache.struts.webapp.example.memory.MemoryDatabasePlugIn"
>
<set-property
  property="pathname"
  value="/WEB-INF/database.xml"/>
</plug-in>
```

填充由Struts Digester来执行。正常情况下，插件把它们自身设置为全局ServletContext上的属性，以便让动作实现能够访问它们。但是，与我们在上一章中所讨论过的基于组件的配置方法相比，这种方法不那么有能力、那么通用。

- 构造一个图表来描述彼此依赖的应用对象是不可能的，因为只有基本属性得到了支持，而对象参考没有得到支持。
- 定义业务对象的唯一途径是在Struts XML配置文件中。
- 插件实现束缚于Struts API。
- 业务对象的配置依赖于Web框架——我们在需要实现一种不同类型的接口时会遇到的一个问题。

包含了它自己的Web应用框架的Java Pet Store示例应用（版本1.3），通过形式化事件

的创建和一个集中式事件处理器（必须实现com.sun.blueprints.waf.controller.web.WebClientController接口）的调用，进行了我们在实现薄Web层时已经见过的、最坚定的尝试之一。

这具有一个明显的优点：Web层应用代码只创建事件或代表用户动作的命令，并显示业务处理的结果。框架代码（不是Web层应用代码）调用业务对象来处理事件。令人遗憾的是，这种方法虽然在理论上十分有趣，但不适用于实际的应用。成功的Web应用框架不试图实施这样的工作流程有一个原因：它在许多情况中是欠灵活的，而且是不合适的。

因此，我们需要把一个Web应用框架和一个超一流的[应用基础结构](#)联合起来使用，而且这两者集成得越轻松，我们的境况就越好。

由于Web接口应该是细薄的，所以Web应用框架不应该为应用提供引力中心，以免影响总体结构。这是应用的业务接口应该扮演的角色。Web应用框架只应该让处理用户输入和显示结果变得容易起来，并提供与业务对象的容易（而又基于接口）的集成。

把我们在上一章中所讨论的基于组件的体系结构与任一Web应用框架集成起来是可能的。根对象com.interface21.web.context.WebApplicationContext（定义Java组件及其关系，并且不需要使用Singleton设计模式），由com.interface21.web.context.ContextLoaderServlet添加到Web应用的ServletContext上，于是任何可以访问该ServletContext的对象都能访问WebApplicationContext根对象。ServletContext把控制器包含在一个框架中。因此，在一个框架中使用Singleton业务对象（即简单的工厂对象）就有了一种杰出的替代方法。Web ApplicationContext根对象实现了与Web应用框架的紧密集成，其紧密程度和Struts插件与Web应用框架的集成紧密程度一样，而且还没有需要在Struts配置文件中定义Struts插件之类的顾虑。

也可以轻松地实现与Struts的紧密集成。方法是通过实现一个通用的Struts插件，让该插件创建一个ApplicationContext对象（比如基于一个串参数且这个参数指定一个XML组件定义文件的位置），并把该对象添加到ServletContext上。这将使一个Struts配置文件能够借助于<plug-in>元素来指定业务对象组件定义的位置，而且使Struts动作能够通过查找ServletContext中的ApplicationContext来访问业务对象。

但是，拥有一个威力强大的、基于Java组件的基础结构，不仅可能实现应用对象与Web应用构件的较紧密集成，而且也可能在总应用上下文中配置框架本身的部分。这样的基于Java组件的基础结构既简化了框架的实现，又简化了它的使用。

示例应用中使用的Web应用框架

因此，笔者为本书示例应用选用了我自己的Web应用框架——更准确地说，选用了笔者所编写的几个Web应用框架中最新的框架。

笔者曾经仔细考虑过是否使用一个现有的框架，比如Struts或Maverick。但是，笔者最后判定，利用任何一个现有的框架都无法实现笔者要演示的集成应用体系结构，而且没有一个现有的框架满足笔者要达到的所有目标。但是，开放源的好处就在于笔者可以为自己的框

架从这些产品中借用一些最好的概念。

笔者在开发Web应用框架方面走过一段很长的弯路。笔者编写自己的第一个生产用Web应用框架是在2000年的年初——在Struts或任何代用品上市之前。后来，这些产品细化了它们的概念。这给了笔者基于这段经历开发一个干净的实现同时又无需担心向下兼容性的自由，而这种自由是Struts开发人员在那时难以获得的。但是，应该说明的一点是，笔者的第一个框架在两年后的今天仍在几个大容量的Web站点上使用，而且已经证明支持可维护的、高质量的应用。

此处所描述的这个框架不只是为演示而设计的，也不只限于本书的示例应用，它可以在生产中。欢迎把它用在实际的应用中。

全部源代码都包含在本书配书下载的/framework目录中，其中的部分代码将在这里加以讨论。

设计目标

从我们在第11章中所讨论的基于Java组件的体系结构的基础开始，实现Web应用框架就变得惊人地容易。

笔者采用了下列设计目标：

- 框架应该基于Java组件，并与前面几章中所描述的体系结构紧密集成。
 - 控制器和视图之类的全部框架对象本身都应该是Java组件，以便允许容易而又一致的配置（包括与应用对象的关系的配置）。
 - 全部应用对象，无论Web处理构件还是业务对象都应该是Java组件，它们可以按同一方式由应用上下文来配置，并可以访问同一个应用上下文中的其他全部对象。这意味着框架定义能够用多种格式来保存，而不只是用XML格式。
- 这种方法使细薄Web层的实现变得更轻松，而且使业务对象在J2EE服务器外部测试起来更容易。
 - 框架应该明确区分开控制器、模型、视图及命令（在命令可应用的地方）的角色。
 - 使用框架所建立的应用代码对Servlet API的依赖性应该尽可能地小。例如，虽然控制器通常依赖于Servlet API，但模型不应该依赖于它。
 - 框架应该是可扩展的：通过在整个框架内基于接口而非具体或抽象类的使用实施松耦合，并一致地使用Java组件。例如：
 - 框架应该轻松自如地使用任一视图技术。它不应该只以JSP为中心。
 - 框架应该支持从请求到逻辑资源的任何映射策略（包括自定义映射）。这使得采用除了大多数框架所实施的基于URL的映射策略之外的其他映射策略成为了可能。
 - 框架应该把控制器与视图彻底分离开，以实现彻底的视图可替换性（在不影响任何应用代码的情况下交换一个视图与另一个视图的能力）。
 - 框架应该提供对JSP、XSLT、Velocity、WebMacro等模板化技术以及XMLC的标准视图实现（这些技术和其他视图技术将在下一章中加以讨论）。轻松地实现对任一视图技术的支持应该是可能的。
 - 框架应该有一个干净的、可维护的实现，以便满足代码质量和我们在第4章中所讨论

的OO设计标准。

- 代码库的大小应该通过避免不相干的能力来保持最小。例如，Struts提供了一个JDBC连接池，但由于连接池是J2EE应用服务器必须提供的，所以Struts的这个连接池是多余的。
- 框架应该提供一个简单的MVC实现，而且在这个实现上不同的工作流程能够被分层。
- 框架应该支持国际化。
- 框架应该允许HTTP高速缓存器控制头部在单个页面级的生成，以兼顾重要的性能优化。在我们已讨论过的框架中，没有框架考虑到了这一点。
- 框架应该提供自定义超类来简化典型使用模式的实现。

基本的MVC控制流

现在，让我们来看一看示例应用中所使用的这个框架怎样支持基本的MVC工作流程。

与我们已经讨论的其他框架相比，这个框架的控制流程更类似于Struts的控制流程，尽管它的视图管理方法更类似于Maverick的视图管理方法。进入请求由一个统一的通用控制器服务器小程序来处理，并且所有应用URL都在web.xml文件中被映射到这个服务器小程序上。这个控制器服务器小程序使用它自己的映射：从进入请求（不一定是基于URL，尽管模式映射是基于URL的）到应用上下文中所定义的请求控制器组件。拥有多个控制器服务器小程序且每个服务器小程序都拥有各自的配置文件是可能的。这些服务器小程序共用一个根上下文WebApplicationContext，但在其他方面是彼此独立的。每个应用特有的请求控制器组件都是一个像Struts动作一样的线程安全的可重用对象。

图12.2所示的时序图描绘了控制的流程（它实际上稍有简化，但还是描绘了各种基本概念）。笔者将在下文中较详细地讨论4种消息的每一种，而且我们稍后将详细看一看每个类和接口。

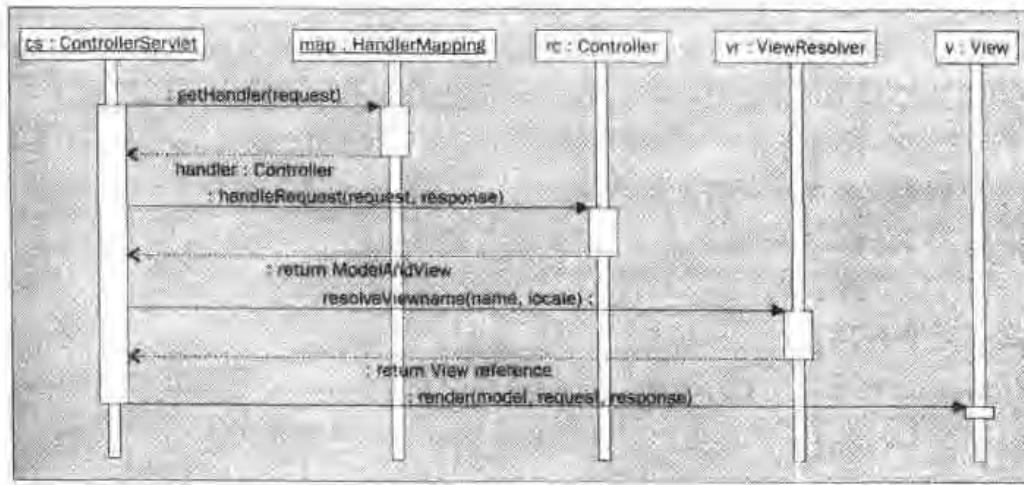


图12.2

1. 在接收到一个HTTP进入请求时，控制器服务器小程序向一个应用请求控制器要求com.interface21.web.servlet.HandlerMapping接口的实现来处理这个请求。Handler Mapping接口的默认实现根据请求URL选择一个控制器（实现了com.interface21.web.

`servlet.mvc.Controller`接口)；可是一个自定义的实现可以使用任何方法(实际上，控制器维护一个包含按一种固定顺序被应用的`HandlerMapping`对象的列表，直到有一个`HandlerMapping`对象匹配时为止，以便兼顾到比我们已讨论过的那些框架的任何一个都复杂的映射)。

2. 控制器服务器小程序调用请求控制器的`handleRequest()`方法。这个方法必须返回一个`com.interface21.web.servlet.ModelAndView`型的对象，该对象含有一个视图的名称和要显示的模型数据。
3. 控制器服务器小程序调用一个`com.interface21.web.servlet.ViewResolver`型的对象来获得一个指向这样一个View对象的参考：该对象具有请求控制器所返回的名称。一个`ViewResolver`型的对象应该能够支持国际化，于是可能返回一个视请求场所而定的不同视图实例。
4. 控制器服务器小程序调用视图的`render()`方法来生成内容。和Maverick中一样，根本不存在对任一特定视图技术的依赖性。

这个核心的框架工作流程没有涉及命令对象的创建。这经常但不始终是令人想要的功能度。把它包含在基本的MVC工作流程中将会不必要地使简单的交互变复杂，在这里创建命令有点大材小用。这样的工作流程细化可以由`com.interface21.web.servlet.mvc.Controller`接口的抽象实现来实现，而应用特有的控制器可以扩展这个Controller接口，同时又不给上述基本工作流程增加复杂性。这个框架携带了多种不同的超类。核心的工作流程舍去了相当多的选择：自定义控制器超类可以支持多种命令流程，包括和Struts中一样的命令对象使用。作为一种选择，可以实现适配器来允许不同工作流程在没有来自框架类的具体继承性的情况下使用。

图12.3所示的类图描述了那些类与相关接口之间的关系，以及这个Web应用框架与上一章中所讨论的那个应用框架是怎样集成的。它还描述了`ControllerServlet`是怎样从提供与`WebApplicationContext`根的集成的框架超类中继承而来的。

这幅类图一举容纳了许多内容。读者最好把它用做自己阅读下列讨论期间的一个参考。

图中圈出的`com.interface21.web.servlet.mvc.Controller`接口由应用特有的控制器来实现。在大多数应用中，没有必要实现图中所显示的其他任何接口，或扩展图中所显示的任何一个类，因为默认实现就满足了常见需求。

现在，让我们依次来看一看每个重要的类和接口。

控制器服务器小程序

使用了任一框架的MVC Web应用的进入点是一个通用的控制器服务器小程序。在Interface21框架中，控制器是`com.interface21.web.servlet.ControllerServlet`类的一个服务器小程序。这是一个具体类，没有必要子类化，因为它通过在其运行的应用上下文来参数化。

控制器服务器小程序本身不执行控制流程。它们是控制器的控制器：它们的角色是选择一个应用特有的请求控制器来处理每个请求，并把请求处理委托给这个请求控制器。

尽管示例应用使用了一个统一的控制器，但Web应用可能拥有多个控制器服务器小程序，并且每个控制器服务器小程序都有一个`WebApplicationContext`(一个与Web应用具体的、`ApplicationContext`型的扩展)。一个Web应用中的所有控制器服务器小程序都共享一

一个共用的父应用上下文，以便使它们能够按照它们的愿望共享任意多个或任意少个的配置。这使得大型应用能够为一组相关用例使用同一个控制器服务器小程序，进而使每个配置文件的大小保持容易管理（就像Struts 1.1所允许的那样），同时又为业务对象的共享提供一种形式机制（与借助于ServletContext的随意共享相反）。

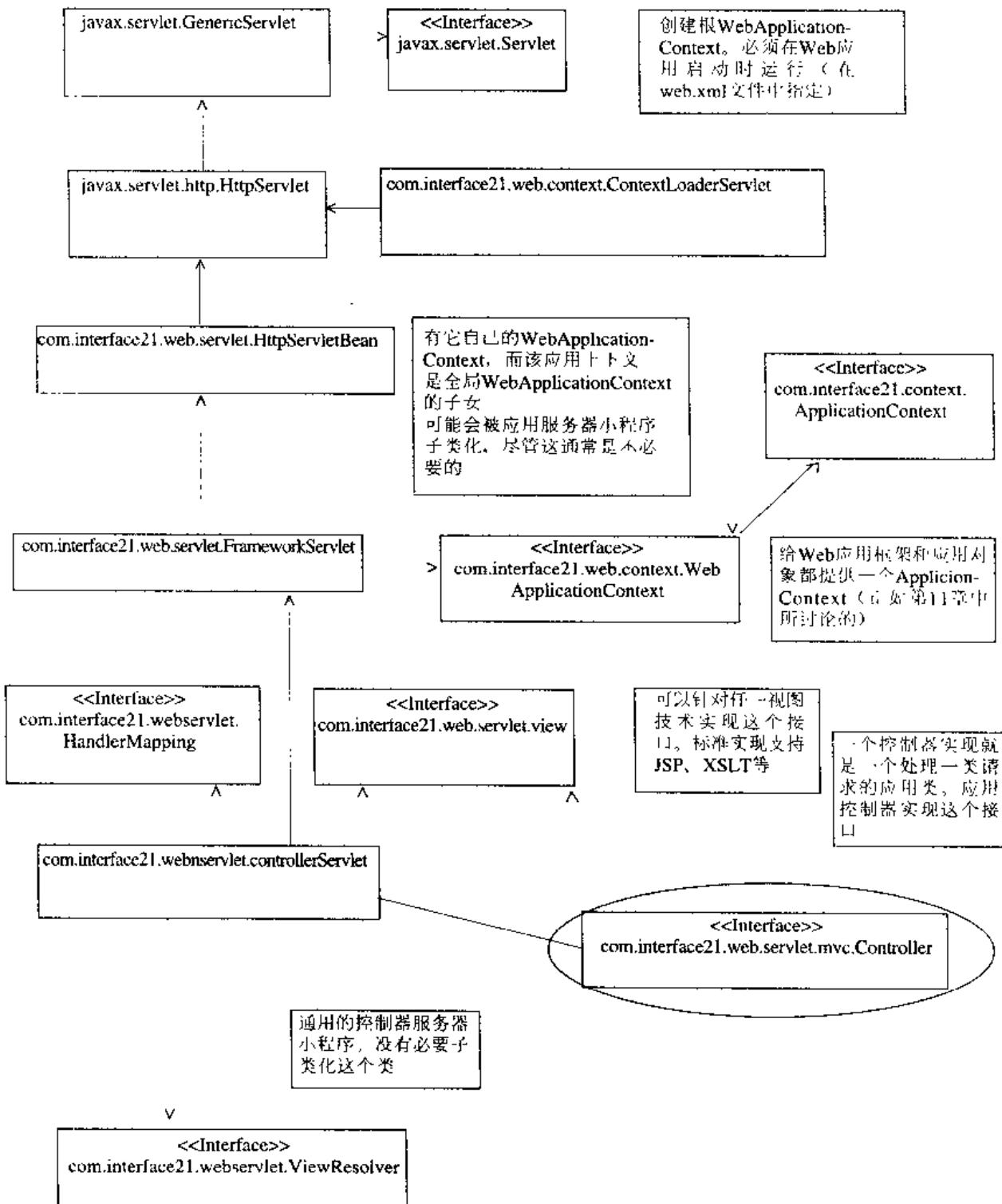


图12.3

服务器小程序基类com.interface21.web.servlet.HttpServletBean通过扩展超类javax.servlet.HttpServlet，来使服务器小程序组件属性能够在初始化时用web.xml服务器小程序<init-param>元素透明地设置。这个超类能够被任一服务器小程序来扩展，不管这单所描述的框架是否得到了使用——它不依赖于MVC实现。和我们的由EJB环境变量所配置的BeanFactory一样，这保证服务器小程序只需暴露要被配置的组件属性即可。没有必要在服务器小程序代码中查找服务器小程序配置参数，而且类型转换是自动的（必要时通过使用PropertyEditor）。

ControllerServlet的中间超类是com.interface21.web.servlet.FrameworkServlet，这个超类通过扩展HttpServletBean来获得根对象WebApplicationContext（这个根对象必须已被ContextLoaderServlet设置为ServletContext中的一个属性，我们不久将讨论这个ContextLoaderServlet），并创建它自己的独立子上下文，以便定义它自己的组件。它还可以访问全局父上下文中所定义的组件。

在默认情况下，每个框架服务器的子上下文将由一个XML文档利用WAR/WEB-INF/<servlet-name>-servlet.xml内的URL来定义（尽管一个组件属性可以用来设置一个自定义的上下文类名），其中服务器小程序名由web.xml中的<servlet-name>元素来设置。因此，就示例应用来说，用于统一控制器服务器小程序实例的上下文必须被放在/WEB-INF/ticket-servlet.xml文件中。

用于示例应用的控制器服务器小程序在web.xml中的配置如下：

```
<servlet>
    <servlet-name>ticket</servlet-name>
    <servlet-class>
        com.interface21.web.servlet.ControllerServlet
    </servlet-class>

    <init-param>
        <param-name>debug</param-name>
        <param-value>false</param-value>
    </init-param>

    <load-on-startup>2</load on-startup>
</servlet>
```

我们还需要在web.xml中定义一个<servlet-mapping>元素。在这种情况下，我们需要保证这个控制器服务器小程序处理针对.html资源的所有请求。与Struts和Maverick所使用的习惯相比，笔者更喜欢这种使用.do和.m作为各自默认映射扩展名的方式。当使用虚拟URL时，我们不必把自己所使用的技术暴露给用户。

```
<servlet-mapping>
    <servlet-name>ticket</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

需要注意的是，当我们把.html URL映射到控制器服务器小程序上时，请求处理中可能包含的静态HTML文件都必须有一个.htm扩展名或另一个不同于.html的扩展名，否则申请这些HTML文件的请求将被控制器拦截。

在使用了多个控制器服务器小程序的应用中，我们可能会把各URL映射到每个控制器上，除非每个服务器处理带有一个不同扩展名或虚拟目录名的请求是合乎逻辑的。要想进一步了解在web.xml部署描述符中定义映射的各种规则，请参见Servlet 2.3规范的第11.1节。

控制器服务器小程序的全部进一步配置被保存在WEB-INF/ticket-servlet.xml中所定义的组件中。这具有一个额外的优点：没有必要学习像Struts配置格式那样的复杂DTD，也就是说，为所有配置（Web特有的和其他类型的）都使用同一种属性配置语法，所以只需知道我们需要配置哪些类的哪些属性即可。

请求到控制器的映射（com.interface21.web.servlet.HandlerMapping）

控制器服务器小程序选择要调用哪个请求控制器来处理进入请求：通过使用当前组件工厂中所定义的com.interface21.web.servlet.HandlerMapping接口的实现，或试图把请求URL与某一个控制器组件名进行匹配（如果当前组件工厂中没有定义HandlerMapping实现）。

示例应用根据请求URL使用一种从请求到控制器的标准映射。框架类com.interface21.web.servlet.UrlHandlerMapping支持一种从进入URL到同一个组件定义内的控制器的名称的简单映射语法，如下所示：

```
<bean name="a.urlMap"
    class="com.interface21.web.servlet.UrlHandlerMapping">
    <property name="mappings">
        /welcome.html=ticketController
        /show.html=ticketController

        /foo/bar.html=otherControllerBeanName
        *=defaultControllerBeanName
    </property>
</bean>
```

上段代码中做了突出处理的那一行指定一个默认控制器（如果没有其他控制器匹配）。如果没有默认控制器被指定，URL不匹配将会导致控制器服务器小程序发送一个HTTP 404 Not Found响应代码。

需要注意的是，com.interface21.web.servlet.UrlHandlerMapping类的mappings属性，用java.util.Properties类所支持的一种格式被设置成一个串值。这之所以得以实现，是因为com.interface.beans.BeanWrapperImpl类（参见第11章）为java.util.Properties类注册了一个标准的属性编辑器；从String到Properties的转换由框架自动进行。

不知道处理器映射的工作方式也能使用框架。上面所给出的标准映射语法足以满足大多数用途，并且适用于许多框架。不过，Interface21框架允许我们定义自定义映射：只需通过实现简单的HandlerMapping接口即可。这个接口可能会根据当前请求的某一方面，比如URL、cookie、用户身份验证状态或请求参数来选择一个控制器。

为每个控制器服务器小程序定义所需个数的映射是可能的。控制器服务器小程序将在它的应用上下文中查找出所有实现了HandlerMapping接口的组件，并按照组件名称、以字母顺序来应用它们，直到有一个组件发现一个匹配，在没有发现匹配时抛出一个异常。因此，赋予映射的组件名有重要意义，尽管这些名称从不被应用代码检索。

这种将请求映射到控制器上的方法比笔者所知道的其他任何框架中所使用的方法更强有力，自定义起来更容易。

请求控制器（com.interface21.web.servlet.mvc.Controller）

com.interface21.web.servlet.mvc.Controller接口定义应用控制器必须扩展的约定。这个接口的实现是多线程的可重用对象，类似于Struts动作。

请求控制器实质上是控制器服务器小程序的一个功能度扩充。通过委托给若干个控制器对象之一，控制器服务器小程序依然是通用的，应用代码不再需要if...else语句链。

请求控制器通常把请求处理成一个单独的URL。请求控制器实现com.interface21.framework.web.servlet.mvc.Controller接口，该接口只含有一个方法：

```
ModelAndView handleRequest(HttpServletRequest request,
                           HttpServletResponse response)
    throws ServletException, IOException;
```

返回对象ModelAndView含有一个数据模型（一个单独的对象或一个java.util.Map）和一个视图（能够再现该数据模型）的名称。这不会引入对视图技术的依赖性：视图名称由ViewResolver对象来解析，从控制器的handleRequest()方法中返回一个null值，表明控制器自己通过写到HttpServletResponse生成了响应。

像服务器小程序和Struts动作一样，控制器是多线程化的构件。因此，任何实例数据通常都应该是只读的，以免它的状态遭到破坏，或保持访问同步的要求损及性能。

像所有应用（即这个框架的对象）一样，控制器是Java组件。这使得它们的属性能够在特有服务器小程序所关联的应用上下文定义中被设置。组件属性都在初始化时被设置，所以在运行时是只读的。

下面这个XML元素定义了示例应用中所使用的主控制器。需要注意的是，这个控制器暴露了这样一些组件属性：它们能让框架使业务对象可由该控制器使用（在突出显示的那些行中），此外还暴露了这样一些配置属性：它们能让该控制器的行为在不用修改Java代码的情况下被修改。

```
<bean name="ticketController"
      class="com.wrox.expertj2ee.ticket.web.TicketController" >
    <property name="calendar" beanRef="true">
        calendar
    </property>
    <property name="boxOffice" beanRef="true">
        boxOffice
    </property>
    <property name="availabilityCheck" beanRef="true">
        availabilityCheck
    </property>
    <property name="userValidator" beanRef="true">
        userValidator
    </property>
    <property name="bookingFee">3.50</property>
    <property name="minutesToHoldReservations">1</property>
</bean>
```

稍后，我们将了解控制器实现。虽然Controller接口非常简单，但该框架提供了几个抽象实现，这些实现提供不同的工作流程，并且应用控制器可以扩展这些工作流程。

该框架也可以处理其他控制器接口，只要为这个接口提供了一个“HandlerAdapter” SPI 实现。这超出了当前讨论的范围，但提供了极大的灵活性。

模型

控制器同时返回一个模型和一个视图名。`com.interface21.web.servlet.ModelAndView`类含有模型数据和应该再现该模型的那个视图的名称。与Maverick及WebWork中不同，模型数据不束缚于一个统一的对象，而是一个映像。用于 ModelAndView 类的便利构造器能返回一个单独的命名对象，如下所示：

```
ModelAndView mv = new ModelAndView("viewName", "modelName", modelObject);
```

这类似于Maverick单模型方法。

如果需要，名称参数可以用来把模型值添加到`HttpServletRequest`上。

为什么把模型数据返回为一个Map，而不是简单地把模型数据暴露为请求属性呢？通过将模型数据设置为请求属性，我们限制了视图实现。例如，一个XSLT视图将需要 JavaBean形式的模式数据，而不是需要请求参数形式的模型数据。转发给另一个系统的视图可能会试图把模型值再现为串参数。因此，在不依赖于Servlet API的情况下暴露模型有实际价值。

模型由一个或多个Java对象（通常是Java组件）所组成。每个模型对象有一个关联的名字；因此，完整的模型被返回为一个Map。通常，这个映像将只有一个单独的项。

视图

视图就是一个能再现模型的对象。`View`接口的用途是通过把视图技术细节隐藏在一个标准接口的后面来分开控制器代码与视图技术。因此，这个框架达到了我们在本章前面所讨论过的视图可替换性的目标。

视图不做任何请求处理或启动任何业务逻辑：它只获取供给它的模型数据，并把后者再现给响应。这形式化了前面所讨论过的“Service-to-Worker”早期数据检索策略的使用。

和控制器一样，视图必须是线程安全的，因为它们代表多个客户来执行。视图通常也是Java组件，因而可以使用我们的-致属性管理方法来定义它们。

视图必须实现`com.interface21.web.servlet.View`接口。最重要的方法是：

```
void render(Map model,
            HttpServletRequest request,
            HttpServletResponse response)
```

这个实现根据模型数据把输出结果写给响应对象。

视图可能会使用若干策略中的某一策略来实现这个方法。例如：

- 和标准`com.interface21.web.servlet.view.InternalResourceView`实现中一样，转发给一个像JSP那样的资源。

- 执行XSL转换。标准com.interface21.web.servlet.view.xslt.XsltView实现使用预编译的格式表执行XSL转换。
- 使用一个自定义输出生成库生成PDF或一种图形格式。

为了弄清楚这个实现如何工作，让我们来看一看com.interface21.web.servlet.view.InternalResourceView实现用来转发JSP页的实现。这个实现把JSP页在WAR内的路径作为一个组件属性来接受。在render()方法中，这个实现按如下过程工作：

- 为视图中所定义的每个“静态值”都创建一个请求属性。视图可以暴露在视图定义时得到配置并在运行时不变化的静态属性，尽管它们可以被动态模型属性替代。
- 为控制器所提供的动态模型中的每个值都创建一个请求属性。每种情况下，属性名都是映射键。
- 使用一个请求调度员转发给JSP视图。在这个模型中，JSP页将主要由模板数据组成，但如果需要，可以生成动态内容。

自定义视图可以轻松地实现View接口，或者扩展com.interface21.web.servlet.view.AbstractView便利超类。例如，笔者在最近的一个项目中实现了一个自定义视图，这个视图在一个被保证是XML的模型内从一个固定的XPath中抽取嵌入的HTML，并把它写给响应对象。在这种情形中，这是一种比使用JSP视图甚至XSLT视图更简单、更有效的方法。

提供使用几乎任何一种输出技术暴露模型数据的视图实现也很容易。例如，Interface21框架所携带的视图支持截然不同的输出技术，比如XMLC和PDF生成。

ViewResolver

控制器通常返回包含了视图名而不是视图参考的ModelAndView对象，因而分割开了控制器与视图技术。和Struts视图定义（它们可能是全局的，但通常与动作有关联）不同，所有视图定义都是全局。

依笔者看来，就视图范围的概念而言，它所增加的价值不及它所增加的复杂性。在一个具有许多视图的应用中，如果必要，视图名称可以包含像包路径那样的限定符来唯一地标识它们。

在适合返回一个匿名内部类或其他实例化视图，而不适宜返回一个供整个应用使用的共享视图对象的少数情况下，构造一个 ModelAndView 对象并让它含有一个实际的视图参考是可能的。控制器可以按照如下所示来做这件事：

```
View myView = new MySpecialView(params);
 ModelAndView mv = new ModelAndView(myView, "modelName", modelObject);
```

框架解析视图名的方式可以通过控制器服务器小程序的应用上下文来配置的。com.interface21.web.servlet.ViewResolver接口的实现可以在控制器服务器小程序的上下文中利用预定义组件名viewresolver来设置。如果没有指定，一个标准实现com.interface21.web.servlet.view.ResourceBundleViewResolver就在类路径上（即在/WEB-INF/classes下）查找一个具有默认名称view.properties的资源包，进而把视图的定义包含为组件。

通过使用标准Java ResoureceBundle功能度，这使得对国际化的透明支持成为了可能。例如，我们可以定义另一个文件views_fr.properties或views_fr_ca.properties来为French和French Canadian地区提供各自的视图定义（可能使用不同的JSP页）（利用标准实现支持的方便性是笔者决定为默认视图配置使用一种属性格式，而不是使用XML的主要原因）。

如果必要，编写一个自定义的视图解析器是很容易的（它可以按照自己所喜欢的任何一种方式自由地创建并返回视图对象），但是标准实现满足所有常见需求。通常，我们需要保证几个控制器服务器小程序中的每一个都为它的视图定义文件使用一个惟一性名称：通过按如下方式设置ResourceBundleViewResolver类的basename参数：

```
<bean name="viewResolver"
      class='com.interface21.web.servlet.view.ResourceBundleViewResolver">
    <property name="cache">true</property>
    <property name="basename">ticketServlet</property>
</bean>
```

ResourceBundleViewResolver类的视图定义遵守我们已经见过的属性组件定义语法。摘自示例应用的下列视图使用了一个默认视图类，并且这个类适合暴露WAR内的JSP页（这个文件可在/WEB-INF/classes/views.properties处的示例应用的WAR中找到）。

```
welcomeView.class=com.interface21.web.servlet.view.InternalResourceView
welcomeView.url=/welcome.jsp
```

以下视图使用了一个标准View实现，这个实现为一个XSLT格式表动态地提供XML数据（这是使用Maverick所使用的同一个Domify包来实现的）。设置root元素变换的名称的根属性应为模型建立：

```
xView.class=com.interface21.web.servlet.view.xslt.XsltView
xView.root=myDocument
xView.stylesheet=/xsl/default.xsl
```

由于ResourceBundle和属性文件一般是从类路径中装入的，所以通常可在/WEB-INF/classes下的一个WAR中找到它们。基于XML的配置文件由框架代码按照URL来装入，通常被放在/WEB-INF目录中，但这个目录本身没有类路径。这种情形适用于大多数Web应用框架。

ContextLoaderServlet

我们已经论及了这个框架的所有活动部分。可是，还有管道工程的一个重要部分有待考虑。

在一个控制器服务器小程序可以工作之前，一个根对象WebApplicationContext必须被附加到ServletContext上。这个对象含有只读数据，所以聚类中的每个服务器有它自己的实例将没有关系。

根对象WebApplicationContext将由com.interface21.web.context.ContextLoaderServlet创建，并被设置为一个ServletContext属性，但必须使用<load-on-startup>web.xml元素把ContextLoaderServlet设置成在其他服务器小程序运行以前的启动时刻装入。

需要注意的是，全局的WebApplicationContext对象不仅仅可以让我们的Web应用框架中的类使用它；还可以让服务器小程序筛选器、JSP自定义标志或使用其他Web框架实现的构件使用它，允许它们访问被暴露为Java组件的业务对象。

在示例应用中，web.xml中的下列元素定义了ContextLoaderServlet：

```
<servlet>
    <servlet-name>config</servlet-name>
    <servlet-class>
        com.interface21.web.context.ContextLoaderServlet
    </servlet-class>
    <init-param>
        <param-name>contextClass</param-name>
        <param-value>
            com.interface21.web.context.support.XmlWebApplicationContext
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

请注意，根本不存在URL到这个服务器小程序上的URL映射：它的用途是装入配置，并使它能供其他Web构件使用。笔者可以把这个对象建模为一个Servlet 2.3应用监听器。不过，仅仅为了这一点就破坏与Servlet 2.2的兼容性是毫无道理的。另外，如果ContextLoaderServlet需要提供Web接口——比如为了返回关于WebApplicationContext根对象的信息，那么把它建模为服务器小程序将会是一个成功的决定。

自定义标志

最后，存在许多自定义标志，它们可以用来执行数据联编，暴露具有国际化支持的消息，以及对模型做循环。这些都是附带的优点，不是核心功能度，因为JSP只打算成为这个框架所支持的视图技术之一。我们将在本章的较后面讨论这些自定义标志中的部分标志。

工作流细化

既然我们已经知道了基本MVC工作流程是如何实现的，下面该来看一看com.interface21.servlet.mvc.Controller的各种抽象实现，以便实现各种不同的工作流程。

图12.4所示的类图描述了这些实现之间的关系，以及应用相关的控制器怎么才能扩展它们。水平线以上的那些类是通用框架类，水平线以下的那些类是我们不久将要讨论的应用特有示例。

最有可能被应用控制器扩展的是下面这些类。所有这些类都使用Template Method设计模式。

- com.interface21.web.servlet.mvc.AbstractController

一个控制器的简单实现，该实现提供一个记录器对象和其他助手功能度。

- com.interface21.web.servlet.mvc.AbstractCommandController

一个命令控制器的实现，该实现自动创建一个命令类的新实例并设法用请求参数填充它的控制器，把命令和任何错误传递给将由子类实现的一个保护模板方法。这个实现类似于Struts Action约定，尽管它可以使用一个不依赖于该Web框架的域对象作为一个命令。

- com.interface21.web.servlet.mvc.FormController

显示束缚于某一Java对象的表单，并能在出现有效性错误的情况下自动重新显示该表单的控制器。我们将在下面的“处理用户输入”一节中讨论这个类。

- `com.interface21.web.servlet.mvc.multiaction.MultiActionController`

通过映射每种请求类型到一个方法而非一个对象上来允许子类处理多种请求类型的控制器。

我们将在下一节的示例里看到使用中的这些类。

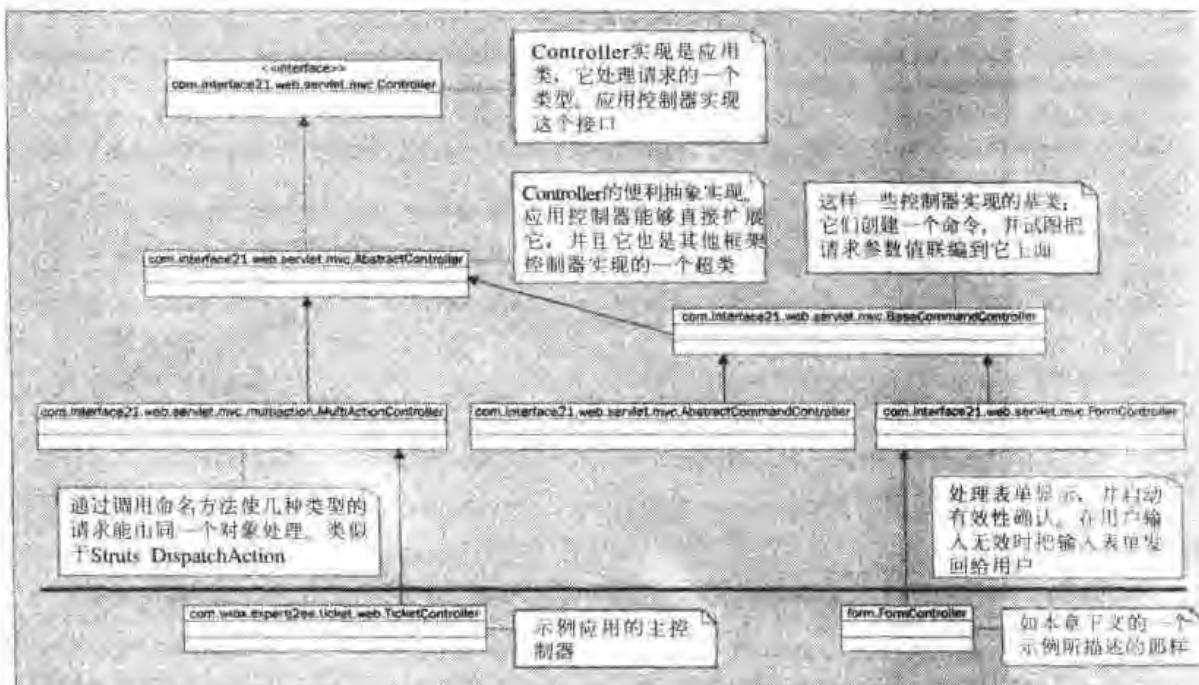


图12.4

示例

现在，让我们来看几个简单的代码示例。稍后，我们将看几个摘自示例应用的较实用的示例。

本节中所讨论的这些示例被包含在与示例应用分开的下载中，此下载叫做i21-web-demo，并含有一个Ant生成脚本。这个下载也可以做为使用了这个框架的Web应用的一个程序骨架。

一个基本的控制器实现

下面的代码清单完整地列出了Controller接口的一个简单实现，这个实现依据一个“名称”请求参数的出现与否和有效性来选择视图。如果该名称参数出现，这个实现就尝试着验证它的有效性（为了便于演示，有效性验证规则只检查该名称是否含有一个连字符，因为连字符被认为是非方法的）。这个控制器可以在下列3种情景中转发给3个视图的任何一个：

- 无名称被供给 (`enterNameView`)
- 名称出现但无效 (`invalidNameView`)
- 名称出现且有效 (`validNameView`)

```

package simple;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.interface21.web.servlet.ModelAndView;
import com.interface21.web.servlet.mvc.Controller;

public class SimpleController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException {

        String name = request.getParameter("name");
        if (name == null || "".equals(name)) {
            return new ModelAndView("enterNameView");
        } else if (name.indexOf("-") != -1) {
            return new ModelAndView("invalidNameView", "name", name);
        } else {
            return new ModelAndView("validNameView", "name", name);
        }
    }
}

```

通常，我们将需要扩展com.interface21.web.servlet.mvc.AbstractController类，这个类提供下列支持：

- 创建一个可供子类使用的Logger保护实例变量；
- 实现com.interface21.context.ApplicationContextAware接口来保证它被赋予了一个指向当前服务器小程序的WebApplicationContext对象的参考，以免把该对象暴露给子类。
- 暴露一个指定哪些请求方法被允许的组件属性。例如，这使得不允许GET请求成为了可能。
- 暴露一个指定该控制器的调用是否只有在该用户已具有会话状态时才会成功的组件属性。

这个AbstractController类使用Template Method设计模式，提供handleRequest()方法（实施请求方法和会话检查）的终结实现，并迫使子类实现下面这个方法（如果子类成功，该方法就被调用）：

```

protected abstract ModelAndView handleRequestInternal(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException;

```

用于这个方法的约定与用于Controller接口的handleRequest()方法的约定完全相同。

一个暴露组件属性的控制器

控制器非常易于参数化，因为它们都是Java组件（由控制器从应用组件工厂中获得）。为了能够形象化配置，我们所需要做的只是给一个控制器添加组件属性。需要注意的是，这些组件属性只在配置时被设置一次，所以在应用服务申请时是只读的，并且不会引起线程化问题。

在启动时配置控制器实例的这种能力，是使用多线程化的可重用控制器的一个主要优点，与Maverick和WebWork的每个请求模型一个新控制器的要求正好相反。也使用可重用控制器的Struts，还提供在它的配置文件中设置控制器属性（在运行中）的能力，这种设置通过<action>元素内所嵌套的<set-property>元素来实现。可是，Struts只支持简单的属性；它不能处理与其他框架对象的关系，限制了把动作对象“装配”成一个集成应用的一部分的能力。

现在，让我们来看一下下面这个简单的控制器。它使用了一个组件属性name（做过突出处理的部分），这个属性可以由控制器在Java代码外部被设置，以修改模型数据输出。笔者已经使这个控制器扩展AbstractController，以便它能替代init()生存周期方法来检查它的名称参数已经得到设置。

如果用户没有得到一个名称参数，这个控制器就把他们发送给enterNameView。如果一个名称参数被供给，用户就被发送给greetingView视图；这个视图的模型是一个greeting串，而这个串按姓名问候用户并显示该控制实例的name属性的值。

```
package simple;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.interface21.context.ApplicationContextException;
import com.interface21.web.servlet.ModelAndView;
import com.interface21.web.servlet.mvc.AbstractController;

public class HelloController extends AbstractController {

    private String name;

    public void setName(String name) {
        this.name = name;
    }

    protected ModelAndView handleRequestInternal(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String pname = request.getParameter("name");
        if (pname == null) {
            return new ModelAndView("enterNameView");
        } else {
            return new ModelAndView("greetingView", "greeting",
                "Hello " + pname + ", my name is " + this.name);
        }
    }

    protected void init() throws ApplicationContextException {
        if (this.name == null)
            throw new ApplicationContextException(
                "name property must be set on beans of class " +
                getClass().getName());
    }
}
```

我们可以按照如下所示来配置这个对象。

```
<bean name="helloController">
    class="simple.HelloController" >
    <property name="name">The Bean Controller</property>
</bean>
```

组件属性常常具有业务接口类型，因而使他们能被设置给同一个应用上下文中的其他组件。我们在示例应用中将会看到这个使用模式。

以一种标准方式参数化控制器如此容易的能力具有许多优点。例如：

- 我们绝对不必在Web层控制器中编写代码来查找配置。
- 我们绝对不必使用Singleton设计模式或调用工厂方法：我们可以简单地暴露业务接口类型的一个属性，并依靠外部配置来初始化它。
- 我们可以经常使用同一个控制器类的不同对象实例来满足不同的需求。
- 在Web容器外部测试控制器组件很容易。我们可以简单地设置它们的组件属性，并用测试请求与响应对象调用它们的handleRequest()方法，就像第3章中所讨论的那样。如果控制器需要在运行时执行复杂的查找（如JNDI查找），这样一种简单的测试策略将会不管用。

多动作控制器

虽然每个请求类型仅使用一个控制器通常是一个好的模型，并且具有诸如为所有控制器使用一个共用超类的能力之类的优点，但这种用法未必总是适用。有时，许多应用控制器最终是无价值的，而且存在微小类的胡乱增殖。有时，许多控制器共享许多配置属性，而且，尽管具体继承性可以用来允许常用属性被继承，但把控制器建模为不同对象是不合逻辑的。在这种情况下，让一个方法而不是让一个类去处理进入请求将会更自然。这是一种能改善工作效率但未必始终是最佳设计决策的实用方法。

Struts 1.1利用org.apache.struts.actions.DispatcherAction超类引入了对基于方法的映射的支持，而且笔者在当前框架的com.interface21.web.servlet.mvc.multiaction.MultiActionController超类中借用了这个概念。MultiActionController超类实现Controller接口，以便按名称为每个请求调用子类上的方法。笔者所知道的其他框架没有一个支持这个特性。多个请求URL可以被映射到一个DispatchAction或MultiActionController控制器上。

要实现基于方法的功能度，惟一方法是使用反射来按名调用方法。这将会有个可以忽略不计的性能开销（如果方法调用对每个进入请求都进行一次），而且很好地举例说明了由框架所隐藏的反射的各种好处。Struts DispatchAction和笔者的MultiActionController都高速缓存方法，这意味着将会最大限度地减少反射的开销。

面临的主要挑战是决定调用哪个命名方法。这个框架超类可以在启动时分析具体子类来找出并缓存请求处理方法。在目前的框架中，请求处理方法必须至少接受HttpServletRequest和HttpServletResponse对象作为参数，并必须始终返回一个ModelAndView对象。请求处理方法可以有任意名称，但必须是公开的，因为在其他方面反射性地使用它们更困难。用于一个请求处理方法的典型签名是：

```
public ModelAndView meaningfulMethodName{
    HttpServletRequest request, HttpServletResponse response);
```

Struts DispatchAction用一个请求参数的值来决定要使用的那个请求处理方法的名称，尽管抽象的LookupDispatchAction子类把这个选择延迟到具体子类。笔者的MultiActionController更灵活，因为它使用Strategy设计模式把这个选择纳入了下面这个简单的接口——能够在扩展MultiActionController为一个组件属性的任一控制器上被设置的实例。

```
public interface MethodNameResolver {
    String getHandlerMethodName(HttpServletRequest request)
        throws NoSuchRequestHandlingMethodException;
}
```

Struts LookupDispatchAction替代DispatchAction来查找一个与应用属性中的某一特殊请求参数相匹配的方法名称。Struts的设计者们似乎不相信接口，把不定的功能度放入一个接口中（Strategy设计模式）比子类化要灵活得多，因为它使得我们能够在无需通过改动超类来修改源代码的情况下，修改扩展了MultiActionController的应用类所使用的映射策略。这也是一个说明合成法优于具体继承性（参见第4章）的上佳例子。

MethodNameResolver的默认实现（com.interface21.web.servlet.mvc.multiaction.InternalPathMethodNameResolver）使方法名称基于请求URL。例如，/test.html被映射到一个名为test的方法上；/foo/bar.html被映射到一个名为foo_bar的方法上。不过，我们一般需要这种映射是较可配置的，因为我们不应该使控制器实现束缚在被映射的URL上。

com.interface21.web.servlet.mvc.multiaction.ParameterMethodNameResolver实现提供与Struts相类似的行为，基于请求所携带的一个action参数的值来选择一个方法。

com.interface21.web.servlet.mvc.multiaction.PropertiesMethodNameResolver实现是更加可配置的。这使得映射能够被保存在由控制器服务器小程序的XML配置文件所指定的属性中：我们的UrlHandlerMapping实现所使用的同一种机制。摘自示例应用的下列组件定义说明了这种方法：

```
<bean name="ticketControllerMethodNameResolver"
    class="com.interface21.web.servlet.mvc.multiaction.
    PropertiesMethodNameResolver">
    <property name="mappings">
        /welcome.html=displayGenresPage
        /show.html=displayShow
        /bookseats.html=displayBookSeatsForm
        /reservation.html=processSeatSelectionFormSubmission
        /payment.html=displayPaymentForm
        /confirmation.html=processPaymentFormSubmission
        /refresh.html=refreshReferenceData
    </property>
</bean>
```

在从MultiActionController中派生出来的应用特有控制器上，这个对象被设置为继承属性methodNameResolver的值：

```

<bean name="ticketController"
      class="com.wrox.expertj2ee.ticket.web.TicketController" >
    <property name="methodNameResolver"
              beanRef="true">ticketControllerMethodNameResolver
    </property>

</bean>

```

做为一种选择，应用可以轻松地提供`MethodNameResolver`接口的自定义实现，这个实现可以依据除请求URL之外的自定义标准来决定方法名，比如用户的会话状态。

`MultiActionController`类也比Struts `DispatchAction`更高级，因为它提供调用另一个委托对象而非一个子类上的方法的能力（委托对象也可以被设置为一个组件属性）。如果需要调用一个类中的命名方法，并且这个类在一个不同的继承性分级结构中，这种能力是十分有用的。委托对象不必实现特殊接口，只需暴露具有正确签名的公用方法即可。

为了举例说明“多动作”控制器方法在实践中如何工作，让我们来看一段从示例应用的主控制器中摘来的代码。`TicketController`类扩展`MultiActionController`类，意味着它不实现`handleRequest()`方法（这个方法被`MultiActionController`实现为了一个终结方法），但必须实现若干个被按名调用的请求处理方法。这个单独的类实现示例应用的全部控制器代码，合并次要方法（否则它们作为次要类要耗用更多的空间），并暴露与所有方法都相关的配置组件属性。

```

public class TicketController
    extends com.interface21.web.servlet.mvc.multiaction.MultiActionController {

```

笔者省略了配置组件属性。但是，实例数据对所有方法是可用的。

请求处理方法必须是公用的，并且可能会抛出任一类型的异常。否则，它们履行和`handleRequest()`方法相同的约定，返回一个`ModelAndView`对象。下面这3个方法对应于上述基于属性的映射中的前3个键。笔者在下面同时给出了这3个方法的各自实现。

```

public ModelAndView displayGenresPage(HttpServletRequest request,
    HttpServletResponse response) {
    List genres = calendar.getCurrentGenres();
    return new ModelAndView("welcomeView", "genres", genres);
}

public ModelAndView displayShow(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, NoSuchPerformanceException {
    // Implementation omitted
}

public ModelAndView displayBookSeatsForm (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, NoSuchPerformanceException {
    // Implementation omitted
}

```

`MultiActionController`类还提供了一种简单的数据联编能力。请求处理方法中的最后一个参数（请求和响应参数将始终是必需的）将被假定成指出该方法期望一个命令组件。命令的类型可以从方法签名中得到确定，而方法签名看起来将类似于摘自示例应用的下列示例：

```
public ModelAndView processSeatSelectionFormSubmission(  
    HttpServletRequest request,  
    HttpServletResponse response,  
    ReservationRequest reservationRequest);
```

一个新的命令实例将通过使用反射（命令必须是具有无变元构造器的Java组件）和捆绑到它的属性上的请求参数来自动创建。如果数据联编成功（即不存在类型不匹配），将用填充后的命令对象调用该方法。任何数据联编异常都被假定成是致命的。这种方法的威力不足以满足许多数据联编应用的需要，但也会十分有用。它类似于Struts ActionForm方法，如果我们实际选择使用无类型化的串参数（在这种情况下，类型不匹配是不可能的）。我们将在本章快结束的时候分析样本应用中的这种数据联编机制。

MultiActionController类还检测子类方法何时需要会话状态参数。在这种情况下，如果存在一个会话，用户的HttpSession将被作为一个参数传递。如果不存在会话， HttpSession将被看做一个工作流程违规和一个被激活的异常，同时不调用该子类方法。

MultiActionController类所提供的、Struts DispatchAction（或者说整个Struts框架）中所缺少的另一个特性是对“上次修改”HTTP头部的支持，其目的是为了支持动态内容的浏览器或代理高速缓存。这种支持也用在示例应用中，并将在第15章中加以讨论。

与“每个请求类型一个控制器”模型相比，这种“多动作”方法具有下列优点：

- 可能更容易维护，因为只有少数较大的类，而不是许多小类。
- 在有些情况中是一个更直观的模型。
- 配置几个较复杂的类比配置许多次要的类更容易，所以应用配置可能会更简单。

这种方法具有下列缺点：

- 一个需要学习的代用模型。“每个请求类型一个控制器”模型是我们比较熟悉的，因为它除了得到Struts 1.0等框架的认可之外，还得到Servlet规范的长期认可（但是，这种情形可能会随着Struts 1.1投入使用而发生改变）。
- 我们失去了编译时检查的优点。子类含有无效的方法定义是有可能的，而我们只有在应用启动和MultiActionController超类未能找到必要的请求处理方法时，才会知道这种情况。不过，MultiActionController实现可以提供与子类方法签名中的可能错误有关的充分错误信息。
- 通过使用个别方法，而不是使用对象，我们失去了在一个公用超类中捕捉异常及消除公用请求处理代码的能力。异常处理变得更有问题。当我们在下文中了解示例应用的实现时，将会看到MultiActionController类怎么才能使子类捕捉从请求处理方法中抛出的异常。

做出怎样的选择实质上归结于编译时与运行时之间的哲学选择。如果一种基于方法的模型使应用更易于维护，它就是一个好选择；如果使用它会构造出一个巨大的控制器，并且该控制器聚集了不相关和不依赖于相同配置属性的功能度，它就是一个差选择。

当然，混用这两种模型是可能的，比如使用单一类处理需要较复杂控制逻辑的动作，使用多动作类处理简单相关动作构成的组。

这绝不是Interface21框架能力的一个全面浏览。有关其他能力的例图和解释，请参见Interface21 web-demo下载。

Web层会话管理

不管我们使用哪个MVC框架，我们的会话管理方法都会对性能和可缩放性产生很大的影响。通过为用户保存服务器端状态，我们可以简化应用代码；通过高速缓存以后可能会用在某一用户会话中的数据，我们可能会改善性能。但是，有一个棘手的权衡问题，因为保存服务器端状态会降低可缩放性。

当我们需要维持会话状态时，基本上有两种选择：

- 使用标准J2EE基础结构让服务器透明地处理状态（通常是最佳选择）。
- 设法把状态保存在浏览器中，以避免对服务器端状态的需要。

下面让我们来依次考虑每种选择。

由J2EE服务器所管理的会话状态

Servlet API提供了一种把状态保存在javax.servlet.http.HttpSession对象中的简单方法。 HttpSession对象实质上就是名称-值对的一个映像，并具有与Map的get()和put()方法相类似的setAttribute()和getAttribute()方法。Web容器将隐藏会话对象查找的细节，在cookie或特殊请求参数中为会话保存一个惟一性键。

聚类和复制

在运行于单一服务器上的应用中，把会话状态保存在 HttpSession 对象中（由 Web 容器透明地管理）毫无问题；代码简化和性能方面的好处超过了附加代价。正如我们在第 10 章中已经讨论过的，这种方法通常比使用有状态会话 EJB 更可取。

但是，当我们有一个服务器聚类时，这样的会话状态将需要由服务器复制，以保证故障切换和最大限度地消除服务器亲合性（Affinity）。被复制的会话状态数据通常被存储在数据库中，以保证稳固的持久性：一个主要性能命中。这也意味着被放在 HttpSession 对象中的所有对象都必须是可串行化的，以便服务器能把它们保存在数据库或文件系统中。

简单优化

由于状态复制的潜在负荷，在使用服务器管理式状态时应用下列优化是十分重要的：

- 不创建会话状态，除非必要。
- 最大限度地减少被保存在服务器端会话状态中的数据量。
- 使用精细粒度的会话状态对象，而不是单一的大会话状态对象。
- 考虑优化会话数据的串行化。

这些优化没有一种会损害应用设计。下面让我们来依次看一看每一种优化。

不创建会话状态，除非必要

若无必要，应该避免创建会话状态（因此也避免了需要在聚类中复制它），这一点十分重要。例如，在示例应用中，只有在订票期间我们才需要保存用户的会话状态。用户活动的大部分将涉及浏览关于艺术流派、演出场次和节目的信息及可供应性，所以通过只在用户

试图订票时才创建会话状态，我们可以避免不必要的状态复制。

当使用JSP作为视图技术时，我们必须意识到这里有一个陷阱。在默认情况下，如果不存在HttpSession对象，每个JSP就创建一个新的HttpSession对象，以防存在这样一个JSP组件：它的会话范围被束缚在该页面上。我们可以使用下列JSP指令替代这个默认设置：

```
<%@ page session='false' %>
```

笔者推荐把这条指令用在每个JSP上。视图不应该访问或操纵会话状态；控制器构件应该利用请求范围使所有数据可供JSP使用。控制器可以在必要时把会话属性复制给数据模型，以使这些属性可供没有会话识别能力的JSP页使用。

应始终利用一条页面指令关闭默认的JSP会话创建。这会避免会话状态的无用创建，并消除JSP页访问和操纵会话状态的危险，会话状态对视图不适合。

避免JSP页访问会话对象的另一种原因是：这违反了视图只应该处理模型中所包含的数据这一重要原则。许多视图（比如XSLT视图）也许根本就不能访问会话状态。但是，如果必需的会话状态被包含在模型中，那么任一视图都能像访问其他模型数据一样来访问它，而且视图可替换性得到保护。

最大限度地减少被保存在服务器端会话状态中的数据量

最大限度地减少被保存在用户会话中的数据量也是很重要的。保存多余的数据会降低复制的速度，而且会浪费服务器上的资源。如果太多的会话组件被包含在模型中，服务器需要把会话数据交换到一个持久性存储器上，在钝化的会话被重新激活时导致一个很明显的性能降低。

能在用户之间共享的参考数据应该总是被保存在应用级别上。不太可能得到重用的数据或许只在需要时才再次被检索。例如，可能保存一个主键，而不是保存从数据库中检索出的大量数据；在为了获得可缩放性好处的少数情况下，这是一个不错的性能牺牲。

使用精细的会话状态对象，而不是单一的大会话状态对象

通常，最好把会话状态分解成若干小于单个大合成对象的小对象。这意味着只有那些已发生变化的对象才需要在聚类中复制。

由于没有什么容易的方法可用来确认一个Java对象的状态是否已经发生变化，所以有些服务器（比如WebLogic）仅当一个对象在一个 HttpSession 中被重新联编时才在聚类中复制会话状态，尽管 Servlet 2.3 规范没有规定什么行为在这里才是预期的（Servlet 2.4 规范可能为会话复制定义标准语义）。当会话中所保存的对象被频繁地更新时，这样的选择性复制可以提供一个很大的性能增益。因此，为了保证正确的复制行为，在会话数据被修改时，应该始终重新联编一个会话属性。这在任何服务器中都不会引起问题。

考虑优化会话数据的串行化

有时，通过替代默认的串行化行为，我们可以极大地改善性能，并减小被串行化对象的大小。这是一种极少需要的特殊优化。我们将在第15章中讨论串行化优化。

浏览器中保存的会话状态

有些会话状态必须始终被保存在用户的浏览器中。由于HTTP是一个无状态协议，如果一个J2EE服务器查找一个cookie，或者找含有用户会话惟一性键的特殊请求参数，那么它只能检索用户的会话对象。

通过把会话状态保存在浏览器中，我们失去了会话状态的透明服务器管理这一优点，但能实现更大的可缩放性和简化聚类。

使用cookie的会话状态管理

在这种方法中，我们使用cookie来保证会话状态。Servlet API使设置cookie很容易，尽管我们必须知道cookie值中被允许的字符有什么限制。

这种方法的优点是它允许我们把应用的Web层变成无状态的，而这可能会明显改善可缩放性，并简化聚类。

但是，有几个常常把这种方法排除在实践之外的缺点。

- 用户必须接受cookie。这未必总是行得通的，因为我们对客户浏览器没有控制权，对用户行为常常也没有多大控制权。
- 为了拒绝服务器状态管理，我们必须构造自己的基础结构把状态编码成cookie。一般来说，这涉及使用反射来抽取对象属性，并使用一个ASCII编码算法把该状态放到合法的cookie值中（cookie值中被允许的字符存在限制）。从理论上讲，这样的支持应该被添加到Web应用框架上，但笔者在实践中还未见到过有人这么做。
- 一个cookie中能够保存的数据量被限定为2KB，我们可以选择使用多个cookie，但这会增加这种方法的复杂性。
- 所有会话数据都必须随同每个请求被发送给服务器或从服务器中发送出来。不过，这个缺陷在实践中不是那么重要，因为若存在大量状态，这种方法就不再适用，而且典型网站上的价值远大于我们可能放在cookie中的任何东西。

有时，使用这种方法也会有好的结果。在某个实际的应用中，笔者成功地使用cookie取代了会话状态，其中用户信息由3个串值（没有一个长于100个字符）和3个布尔值组成。这种方法提供了商业价值，因为该应用需要运行在地理位置很分散的几个服务器上，进而排除了服务器管理式会话状态的复制。为了通过使用反射从会话对象中抽出属性值，并生成一个可接受的ASCII码cookie值，笔者使用了一个基础结构类。

使用隐藏表单字段的会话状态管理

另一种由来已久的会话状态管理方法（也不是J2EE特有的）涉及应用在页之间跟踪含有会话状态的隐藏表单字段。笔者不推荐这种方法。它有几个严重的缺点：

- 它实现起来相当困难。
- 它使得应用逻辑易受视图错误（比如JSP页面错误）的伤害。这会使保证Java开发人员与视图开发人员的角色区分变得更加困难。
- 和cookie状态管理一样，它要求会话数据被完全保存为串。
- 它使得为站点内的导航使用普通超级链接变得不可能。每个页面事务需要是一个表

单提交。这常常意味着我们必须使用JavaScript链接（为了提交含有隐藏字段的表单）来取代普通链接，进而使鼠标变得很复杂。

- 和使用cookie的情形一样，发送隐藏表单字段值给服务器和从服务器中发出这些值会占用带宽。不过，和cookie的情形一样，这个缺陷在实践中很少是一个主要问题。
- 用户可以轻松地通过检查页面源代码来知道会话状态。
- 会话状态可能会丢失，如果用户临时离开站点。这个问题不影响服务器状态管理或cookie的使用。

在某些应用中，会话状态可以被保存在cookie中。不过，当我们需要会话状态时，依靠Web容器来管理 HttpSession 对象通常更安全——而且更简单，保证我们能够集中精力来确保会话状态在聚类式环境中的有效复制。

处理用户输入

在实现Web接口时，最单调乏味的任务之一就是需要从表单提交中接受用户输入，并使用它填充域对象。

核心问题是，由HTTP表单提交所产生的所有值都是串。当域对象属性不是串时，我们需要转换值，在必要时检查参数类型。这就意味着我们需要同时验证提交参数的类型和这些参数的语义是否有效。例如，age的参数值是23x，则它是无效的，因为它不是数值；但是，一个-1的数字值也是无效的，因为年龄不能是负的。我们还可能需要检查年龄是不是在一个预先规定的范围内。

我们还需要检查强制性表单字段的存在性，而且可能需要设置域对象的嵌套属性。

Web应用框架在这里可以提供有价值的支持。通常我们希望对无效数据采取下列行动中的一个：

- 完全拒绝输入，把这个输入作为一个无效提交来对待。这种方法是恰当的，例如，在用户只能用该应用内的一个链接到达这个页面时。由于该应用而非用户控制着那些参数，所以无效参数值表示一个内部错误。
- 给用户发回表单，以便允许它们改正错误并重新提交。在这种情况下，用户期望看到他们所输入的实际数据，不管该输入是有效的还是无效的，即便具有正确的类型。需要第二个动作是最常见的，而且最需要从Web应用框架中支持第二个动作。

本节中的讨论都集中在单个输入表单的提交上，而不是集中在涉及几个页面的“向导式”提交上。在适用于单个表单提交的规则中，有许多规则也适用于向导式提交，尽管实现多页表单自然比实现单页表单更困难（向导式提交可以被建模为面向不同Java组件的多个表单提交，或被建模为更新同一个组件的不同属性的多个子提交）。

数据联编及显示输入错误以便重新提交

通常，我们不是做各请求参数的低级处理，而是希望透明地更新Java组件上的属性。更新HTTP表单提交上的对象属性的操作通常叫做数据联编（Data binding）。

在这种方法中，Java组件（一般是命令）将同名地把属性暴露为预期的请求参数。Web

应用框架代码将使用组件操作包，依据请求参数来填充这些属性。这是<jsp:useBean>动作所使用的同一种方法，尽管正如我们前面已经说过的，这不提供一个足够高级的实现。在理想情况下，这样的命令组件不依赖于Servlet API，因此，一旦设置完毕组件的属性，组件就能作为参数被传递给业务接口方法。

诸如Struts和WebWork之类的许多框架，都使用数据联编作为它们处理请求参数的主要方法。不过，数据联编未必是最好的方法。当有很少的参数时，创建一个对象可能不合适。我们可能需要调用不带有变元或带有一两个基本变元的业务方法。在这样的情况下，应用控制器本身就可以使用HttpServletRequest接口的getParameter(String name)方法来处理请求参数。在这种情况下，创建一个命令对象会浪费许多时间和内存，但更严重的是，使一个十分简单的操作变得似乎更加复杂。就一种纯粹的数据联编方法来说，如果我们像Struts中那样为每个组件都拥有一个单独的命令，我们最终可以轻松地得到几十个动作表单类。

另外，传递给一个请求的动态数据有时不被包含在请求参数中。例如，我们有时可能需要把动态信息包括在一个虚拟URL中。例如，一篇新闻文章URL可能会把一个ID包括在服务器小程序路径中，同时又不需要一个查询串，如下例中所示：

```
/articles/article304/.html
```

这种方法也许方便了内容的高速缓存，比如通过一个服务器小程序筛选器或在浏览器中。

需要注意的是，逐个处理请求参数会给应用控制器代码增加检查参数是否具有必需类型的负担；例如，在使用一个像Integer.parseInt(String s)之类的方法来做类型转换之前。

数据联编最能发挥作用的地方是有许多请求参数的地方，或无效输入时需要重新提交的地方。一旦用户数据被保存在一个Java组件中，使用该对象来填充一个重新提交表单马上会变得很容易。

MVC框架中的数据联编方法

Java Web应用框架中似乎存在两种基本的数据联编方法，这两种方法都基于Java组件。

- **把数据（无效或有效）保存在单个Java组件中，使一个重新提交表单的字段值能在必要时被轻松地填充**

这是Struts ActionForm方法。这种方法具有ActionForm组件中的数据未被类型化的缺点。由于ActionForm组件的所有属性是串，所以ActionForm组件一般不能是一个域对象，因为只有少数域对象保存串数据。这就意味着当我们已经确认该ActionForm组件所含有的数据是有效的（而且能被转换成目标类型）时，我们将需要执行另一个步骤，以便用它来填充一个域对象。

在这个模型中，有效性验证将发生在表单组件上，而不是发生在域对象上；我们不能尝试着用该表单组件来填充一个域对象，除非我们确信该表单组件的所有串属性都能被转换成必要的类型。需要注意的是，我们需要把错误代码之类的错误信息保存在某个地方，因为表单组件只能存储被拒绝的值。Struts把每个被拒绝对象的错误信息保存在该请求内的单独ActionErrors对象中。

- **把错误保存在一个独立对象中**

这种方法尝试着在不使用中间保存对象（如“动作表单”）的情况下填充域对象。

我们正试图联编的域对象将把它的字段更新成正确类型的输入，而类型不正确的任

何输入（不能被设置在域对象上）将可以从被添加到本请求上的一个独立错误对象中访问到。语义（而非语法）的有效性验证可以由域对象在填充完成之后执行。

WebWork使用了这种方法的一个变通方法。如果一个WebWork动作实现了webwork.action.IllegalArgumentExceptionAware接口，并且webwork.action.ActionSupport便利超类也实现了这个接口，那么当这个动作的属性正被设置时，该动作将会接收到任何类型不匹配的通知，使必要时存储非法值以便显示成为了可能。在这个模型中，类型不匹配由框架透明地处理，应用代码不必考虑它们。应用有效性验证可以处理域对象，但不能处理哑存储器对象。

由于WebWork结合了命令和控制器这两者的角色（即WebWork术语的“动作”），所以WebWork控制器不是真正的域对象，因为它依赖于WebWork API。不过，这是WebWork的总体设计所造成的后果，不是它的数据联编方式所造成的。

第二种方法在框架中实现起来更困难，但可以简化应用代码，因为它给框架增加了执行类型转换的责任。这样的类型转换会是十分复杂的，因为它可以使用标准的JavaBean PropertyEditor机制，使复杂对象能够用请求参数来创建。第二种方法用在了示例应用的框架中，尽管这个框架还支持一种类似动作表单的方法（和Struts的动作表单方法类似），在这种方法中，串属性用于最大限度地降低出错的可能性。

JSP自定义标志

我们怎么才能知道一个表单上要显示什么数据呢？通常，我们希望为新输入表单和重提交使用同一个模板（JSP和其他），因为检查含有数百行不一致置标的不同表单不是一件令人开心的事情。因此，表单必须能够被无内容的对象数据、来自一个现有对象（例如，从数据库中检索出的一个用户配置文件）的数据或表单提交遭拒绝后可能含有错误的数据填充。在这3种情况中，可能需要在用户之间能共享的附加模型数据；例如，填充下拉菜单等动态列表的参考数据。

在数据可能来自不同数据源（或直接无内容，如果根本不存在可从中获取数据的组件）的情况下，借助于标志处理器，JSP自定义标志可以用来把数据获得问题从模板中转移到幕后的助手代码中。许多框架都使用一种类似的方法，不管它们怎样存储表单数据。

如果我们需要执行数据联编，表单一般需要使用特殊标志来获得数据。示例应用所用框架中的标志在概念上类似于Struts或其他框架中的标志。无论表单是否已被提交，我们都使用自定义标志来获得值。

对于每个字段，我们可以使用自定义标志来获得由一个失败的数据联编尝试所导致的错误消息，连同遭到拒绝的那个属性值。这些标志协同应用上下文的国际化支持来自动显示用于正确地区的错误消息。例如，下面是一个JSP页的代码段，如果该页面上用户组件的电子邮件属性未能通过有效性验证，它就使用协作性自定义标志来显示一条错误消息，连同一个含有被拒绝值的预填写输入字段（如果这个被提交值是无效的）或当前值（如果被提交的电子邮件值中无错误）。需要注意的是，只有外层标志<i21:bind>是这个框架所特有的。它把数据暴露给它里面所嵌套的、使它能够使用JSP Standard Tag Library标志（如条件标志<c:if>和输出标志<c:out>）的标志。这意味着最复杂的操作（如条件判断）可以利用标准标

志来执行（我们将在下一章中详细讨论JSTL）。

```
<i21:bind value="user.email">
    <c:if test="${bind.error}">
        <font color="red"><b>
            <c:out value="${bind.errorMessage}" />
        </b></font><br>
    </c:if>

    <input type="text" length="2" size="30" name="email"
           value=<c:out value="${bind.value}" />> />
</i21:bind>
```

<i21:bind>标志使用一个value属性来标识我们感兴趣的组件和属性。组件前缀是必不可少的，因为这些标志和支持性基础结构可以在同一个表单上支持多个联编操作：一种特有的能力，据笔者所知。<i21:bind>标志定义一个bind脚本变量，这个变量能够用在其作用范围内的任何地方，并暴露关于这个属性的联编操作是成功还是失败的信息及操作结果的显示值。

这个标志（<i21:bind>）通过依次计算如下值到达要显示的串值：针对这个字段的、遭到拒绝的输入值；页面上具有所需名称的一个组件的这个属性的值（和<jsp:getProperty>标准动作相同的行为）；以及空串，如果没有组件或错误对象（无背景数据的新表单就是这一情形）。由<i21:bind>标志所创建的bind脚本变量，还暴露方法来指出该显示值来自这些数据源中的哪一个。

另一个便利的标志只在表单上有数据联编错误时才计算它的内容：

```
<i21:hasBindErrors>
    <font color="red" size="4">
        There were <%=count%> errors on this form
    </font>
</i21:hasBindErrors>
```

我们将在下一章中较详细地讨论自定义标志，但本例演示了自定义标志的一种非常好的用法。它们从JSP内容中彻底隐藏了数据联编的复杂性。

需要注意的是，不用JSP自定义标志而是用Struts框架或本章中所设计的框架来处理表单提交也是可能的；自定义标志只不过提供了一种简单、方便的方法而已。

数据确认

如果存在类型不匹配，比如一个非数字值对一个数字属性，那么应用代码应该不必执行有效性验证。但是，在决定是否让用户重新提交输入之前，我们可能需要运用高级的有效性验证规则。为了保证我们的显示方法适用，由应用代码有效性验证所引起的错误（比如18岁以下的年龄不能接受），必须和框架所引起的错误（比如类型不匹配）使用相同的报告系统。

数据有效性确认应该在何处进行

毫无疑问，数据有效性验证问题不可能在应用的任一体系结构层上都适用。我们有一组容易让人混淆的选择：

- 运行于浏览器内的JavaScript中的有效性确认。在这种方法中，有效性检查在表单提交以前进行，并用JavaScript警报提醒用户修改无效的值。
- Web层中的有效性确认。在这种方法中，一个Web层控制器或助手类将在表单提交之后验证数据的有效性，并在数据无效时把用户返回到该表单，同时不调用任何业务对象。
- 业务对象中的有效性确认。这些对象很可能是EJB。

做一个选择会很困难。这个问题的根源是“有效性验证是业务逻辑吗？”这个问题的答案在不同的情况下有所不同。

有效性验证问题一般属于语法或语义确认的范畴。语法确认包括像检查数据存在之类的、长度可接受的简单操作，或有效格式（比如一个数）。这一般不是业务逻辑。语义确认需要更高的技巧，并且涉及一些业务逻辑，甚至数据存取。

现举一个简单注册表单的例子。一个注册请求可能含有用户所喜欢的用户名、密码、电子邮件地址及邮政编码。语法确认可以用来保证所有必需字段都存在（假设它们都是必需的，并由一个业务规则来控制，而在这种情况下，语义就被涉及）。可是，每个字段的处理变得更复杂。如果我们不了解国家选择，就无法确认邮政编码字段的有效性，因为不同国家的邮政编码，比如UK邮政编码和US邮区编码有不同的格式。这是语义确认，并且近似业务逻辑。

更糟糕的是，用于确认UK邮政编码的规则太复杂，并且需要确认客户端的太多数据。即使我们同意接受看起来像一个UK邮政编码但无语义意义的输入（比如 Z10 8XX），JavaScript仍将证明它是没有实际意义的，如果我们打算支持多个国家。我们可以用JavaScript来确认电子邮件地址格式，并执行密码长度和字符检查。不过，有些字段将会引起一个更严重的问题。我们假定用户名必须是惟一性的。如果不访问一个能连接到数据库的业务对象，确认一个被申请的用户名是不可能的。

上述所有方法都有它们的优缺点。使用JavaScript将会降低服务器上的负荷，并改善响应时间。如果在表单提交之前必须做多个改正，那么负荷降低可能会很明显，并且用户可能会察觉出系统的响应速度大有提高。

另一方面，复杂的JavaScript很快就会变成一个可维护灾难，交叉浏览器问题可能会出现，而且页面重要性可能会被客户端脚本明显提高。根据笔者的经验，维护复杂的JavaScript可能会证明比使用Java维护等同功能度有高得多的代价。由于JavaScript是一种完全不同于Java的语言，所以这种方法也有这样一个缺点：Java开发人员编写业务逻辑，而JavaScript开发人员却必须编写确认规则。JavaScript确认对非Web客户（比如带有远程接口的EJB的远程客户或Web服务客户）也是无用的。

不要只依靠客户端JavaScript确认。用户（而不是服务器）控制着浏览器。让用户禁用客户端脚本功能是有可能的，意味着在服务器上执行相同的检查始终是必要的。

相反，Web层中的有效性确认具有这样一个严重缺点：使确认逻辑（有可能是业务逻辑）束缚于Servlet API，还可能束缚于一个Web应用框架。令人遗憾的是，Struts趋向于将有效性确认朝着Web层的方向推进，因为有效性确认必须在Struts ActionForm对象上进行，而这些对象依赖于Servlet API，于是不能被传入到一个EJB容器中，并且不应该被传给任何

业务对象。例如，有效性确认常常通过替代org.apache.struts.action.ActionForm确认方法来完成，如下所示：

```
public final class MyStrutsForm extends org.apache.struts.action.ActionForm {  
    ...  
  
    public ActionErrors validate(ActionMapping mapping,  
        HttpServletRequest request) {  
  
        ActionErrors errors = new ActionErrors();  
  
        if (email == null || "".equals(email)) {  
            errors.add("email", new ActionError("email.required"));  
        }  
  
        return errors;  
    }  
}
```

笔者认为这是一个重要的设计缺陷——而且ActionForm对象必须扩展一个依赖于Servlet API的Struts超类也是。

Struts 1.1还提供通过XML配置文件来控制的声明性确认。这种确认方式十分强有力，使用起来也很简单（它基于规则表达式），但无论确认规则是在Java代码中，还是在XML中，它们在Web层中常常是放错了地方。

有效性确认应该依靠业务对象，而不是依靠Web层。依靠业务对象的做法最大限度地提高了重用确认代码的可能性。

可是，存在这么一种情况：有效性确认代码没有必要和业务对象放在同一个位置：其中的业务对象都是EJB的体系结构中。每个进入EJB层的调用可能都是一个远程调用：我们不希望把网络往返行程浪费在交换无效数据上。

因此，执行有效性确认的最佳地点是在Web容器所位于的同一个JVM中。不过，有效性确认不必是Web接壤逻辑层的一部分。我们应该为有效性确认设立下列目标：

- 确认代码不应该被包含在Web层控制器或Web所特有的对象中。这使确认对象可以重用于其他客户类型。
- 为了容许国际化，从Java代码中分离出错误消息十分重要。资源包提供了做这件事的一种标准而又上佳的方法。
- 凡是在恰当的地方，我们都应该考虑到有效性确认的参数化，以免重新编译Java代码。例如，如果某个系统上的最小和最大密码长度分别是6和64个字符，这个长度就不应该被硬编码到Java类，甚至用常数的形式。这样的业务规则可以改变，而且在不重新编译Java代码的情况下做这样一个改动应该是可能的。

我们可以满足这些目标，如果确认者是不依赖于Web API的JavaBean，但可能会访问它们所需要任何业务对象。这意味着确认者必须作用于域对象，而不是作用于Servlet API特有的概念（比如HttpServletRequest）或框架特有的对象（比如Struts ActionForm）。

示例应用框架中的数据确认

现在，让我们来看一看这种数据确认方法在本章所描述的，也是示例应用中所使用的框架中是如何工作的。所有确认者只依赖于两个非Web特有的框架接口：`com.interface21.validation.Validator`和`com.interface21.validation.Errors`。该`Validator`接口要求实现类首先确定它们可以确认哪些类的有效性，然后实现一个确认方法：该方法接受一个要被确认的域对象，并把错误报告给一个`Errors`对象。然后，`Validator`实现必须把这个要被确认的对象强制成正确的类型，因为用一致方式调用具有类型化参数的确认者是不可能的。完整的`Validator`接口如下所示：

```
public interface Validator {  
    boolean supports(Class clazz);  
    void validate(Object obj, Errors errors);  
}
```

错误通过调用下列方法被添加到`Errors`接口上：

```
void rejectValue(String field, String code, String message);
```

`Errors`对象暴露错误信息，这些信息用来支持上述自定义标志所定义的显示。传递给应用确认者的同一个`Errors`对象还由框架用来说明类型转换错误，保证所有错误都能以相同方式得到显示。

下面是示例应用中所使用的`Validator`实现的部分代码清单，这段代码确认`RegisteredUser`对象中所保存的用户配置信息。`RegisteredUser`对象暴露电子邮件和其他属性，其中包括一个`Address`型关联对象上的邮政编码。

`DefaultUserValidator`类是一个应用特有的JavaBean，暴露分别确定电子邮件地址的最小和最大容许长度的`minEmail`和`maxEmail`属性：

```
package com.wrox.expertj2ee.ticket.customer;  
  
import com.interface21.validation.Errors;  
import com.interface21.validation.FieldError;  
import com.interface21.validation.Validator;  
  
public class DefaultUserValidator implements Validator {  
  
    public static final int DEFAULT_MIN_EMAIL = 6;  
  
    public static final int DEFAULT_MAX_EMAIL = 64;  
  
    private int minEmail = DEFAULT_MIN_EMAIL;  
    private int maxEmail = DEFAULT_MAX_EMAIL;  
  
    public void setMinEmail(int minEmail) {  
        this.minEmail = minEmail;  
    }  
  
    public void setMaxEmail(int maxEmail) {  
        this.maxEmail = maxEmail;  
    }  
}
```

Validator接口中的supports()方法的下列实现表明这个确认者只能处理RegisteredUser对象：

```
public boolean supports(Class clazz) {
    return clazz.equals(RegisteredUser.class);
}
```

Validator接口中的validate()方法的下列实现完成对于对象参数的若干个检查，这个对象参数可以被安全地强制成RegisteredUser类型：

```
public void validate(Object o, Errors errs) {
    RegisteredUser u = (RegisteredUser) o;
    validateEmail(u.getEmail(), errs);
    // More check method invocations omitted
}

private void validateEmail(String email, Errors errs) {
    if (email == null || "".equals(email)) {
        errs.rejectValue("email", "emailRequired",
                          "E-mail Address is required");
        return;
    }

    if (email.length() < this.minEmail || email.length() > this.maxEmail) {
        errs.rejectValue("email", "emailLengthInvalid",
                          "E-mail Address is invalid");
        return;
    }
    // Other checks omitted, including checks on min
}
}
```

该确认者的组件属性使用我们在上一章中讨论过的同一种文件格式来设置，以满足我们形象化业务规则的目标：

```
<bean name="userValidator"
      class="com.wrox.expertj2ee.ticket.customer.DefaultUserValidator" >
    <property name="minEmail">6</property>
    <property name="maxEmail">64</property>
    ...
</bean>
```

因此，我们可以确认RegisteredUser对象的有效性，不管我们使用什么接口（Web或其他）。接口无关性错误对象中的错误信息可被显示在我们所选择的接口中。

现在，让我们来看一看怎样才能使用我们的框架来调用这段确认代码。

com.interface21.web.servlet.mvc.FormController超类是一个框架Web控制器，被设计用来既显示以单个Java组件为基础的表单，又处理表单提交，在必需重新提交时，自动把用户返回到原表单。

子类只需指定该表单组件的类即可（这个类被传递给超类构造器）。“表单视图”和“成功视图”的名称应该在组件定义中被设置为组件属性（如下文中所示）。

下面这个简单的示例摘自我们的MVC演示，其功能是显示FormController的一个子类，这个例子能够显示以RegisteredUser对象为基础的表单，允许用户输入邮政编码（从关

联的Address对象中)、出生年份、电子邮件地址等属性。邮政编码和电子邮件地址是文本输入，而出生年份应该从系统所接收的一个出生年份下拉菜单中选取：

```
package form;

import java.util.HashMap;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import com.interface21.web.servlet.ModelAndView;
import com.interface21.web.servlet.mvc.FormController;
import com.wrox.expertj2ee.ticket.customer.RegisteredUser;

public class CustomerInput extends FormController {
```

在默认情况下，FormController将使用Class.newInstance()来创建表单组件的一个新实例，这个实例的属性将用来填充表单（表单组件必须有一个无变元构造器）。不过，通过替代下面这个方法，我们自己就可以创建一个对象。在目前这个示例中，笔者已经简单地用某些文本预填充了一个属性：通常，我们只有当会话中存在一个合适组件，或我们知道如何依靠一个数据库查找来检索一个组件时，我们才会替代这个方法。

```
protected Object formBackingObject(HttpServletRequest request) {
    RegisteredUser user = new RegisteredUser();
    user.setEmail("Enter your email");
    return user;
}
```

如果控制器正在处理一个表单而非表单提交的请求，将调用这个方法。

如果存在有效性确认错误——类型不匹配和（或）确认者对象所引起的错误，子类将不必做任何事情。FormController类将自动把用户返回到提交表单，使组件和错误信息能让视图利用。

如果对象通过了有效性确认，我们必须替代几个过载onSubmit()方法之一来采取必要的动作。这些方法中的每一个都被传递填充后的域对象。在我们的简单示例中，笔者刚刚使对象的属性之一可让视图利用；实际的控制器将会把填充后的命令传递给响应对象，并选择一个依赖于结果的视图。需要注意的是，这个方法不接受请求或响应对象。这些对象是多余的，除非我们需要操作用户的会话（这些请求和响应对象可以通过替代其他onSubmit()方法来获得）：那些请求参数已经被抽取出来，并被赋给命令属性，而通过返回一个模型映像，我们让视图做它需要对响应所做的事情。

```
protected ModelAndView onSubmit(Object command) {
    RegisteredUser user = (RegisteredUser) command;
    return new ModelAndView(getSuccessView(),
                           "email", user.getEmail());
}
```

我们可能希望替代isFormSubmit()方法来告诉FormController，它正在处理一个要显示该表单的请求，还是正在处理一个表单提交。默认实现（如下面所示）假设一个POST型的HTTP请求方法指出一个表单提交。情况可能未必总是这样；我们可能希望把映射到这个控制器的

两个URL区分开，或者检查一个特殊请求参数的存在性。

```
protected boolean isFormSubmission(HttpServletRequest request) {
    return "POST".equals(request.getMethod());
}
```

如果除了已联编的对象之外，表单还需要参考数据，那么这个数据可以从referenceData()方法中作为一个映像（比如我们的框架中的一个模型映像）被返回。在本例中，我们返回一组显示在表单上的静态出生年份，尽管参考数据通常将来自数据库。

```
private static final int[] BIRTH_YEARS = {
    1970, 1971, 1972, 1973, 1974 };

protected Map referenceData(HttpServletRequest request) {
    Map m = new HashMap();
    m.put("BIRTHYEARS", BIRTH_YEARS);
    return m;
}
```

这是所有应用特有的Java代码所必需的。这个控制器通过服务器小程序的XML配置文件中的下列组件定义来配置。请注意下列代码段中突出处理过的那一行，该行把从通用超类FromController中继承来的确认者属性设置成一个指向确认者组件的参考；另外，请注意继承来的beanName、formView和successView属性是如何设置的。

```
<bean name="customerController"
      class="form.CustomerInput" >
    <property name="validator" beanRef="true">customerValidator</property>
    <property name="beanName">user</property>
    <property name="formView">customerForm</property>
    <property name="successView">displayCustomerView</property>
</bean>
```

下面让我们来看一看整个表单的代码清单，其中用到了前文所描述的自定义标志。请注意，“user”的组件名称匹配于被设置为控制器上某一属性的组件名称。由于邮政编码是RegisteredUser对象的billingAddress属性的一个属性，所以请注意嵌套的属性语法。

```
<form method="POST">

<i21:hasBindErrors>
    There were <%=count%> errors
    <p>
</i21:hasBindErrors>

Postal code:
<br>
<i21:bind value="user.billingAddress.postcode">
    <c:if test="${bind.error}">
        <font color="red"><b>
            <c:out value="${bind.errorMessage}" />
        </b></font>
        <br>
    </c:if>

    <input type="text" length="16" name="billingAddress.postcode"
          value=">">
</i21:bind>
```

`birthYear`属性应该被设置成用`referenceData()`方法所返回的那个参考数据来填充的出生年份下拉列表中的一个选择。我们使用嵌套的JSTL条件标志从该列表中选择联编值（Bind value）所匹配的值。如果联编值是空白（因为当表单被首次显示时它将是空白），第一个值“Please select”将被选取。

```

Birth year:
<br>
<i21:bind value="user.birthYear">
    <c:if test="${bind.error}">
        <font color="red"><b>
            Birth year is required
        </b></font>
        <br>
    </c:if>

    <select size="1" name="birthYear">
        <option value="">Please select</option>
        <c:forEach var="birthYear" items="${BIRTHYEARS}">
            <option value=">">
                <c:if test="${bind.value == birthYear}">SELECTED</c:if>
                    > <c:out value="${birthYear}" />
            </option>
        </c:forEach>
    </select>
</i21:bind>
```

电子邮件地址字段需要与邮政编码文本字段相似的代码：

```

Email:
<br>
<i21:bind value="user.email">
    <c:if test="${bind.error}">
        <font color="red"><b>
            <c:out value="${bind.errorMessage}" />
        </b></font>
        <br>
    </c:if>

    <input type="text" length="2" size="30" name="email" value="

```

当用户请求要被映射到控制器上的`cust.html`时，表单将显示带有用`formBackingObject()`方法预填充的电子邮件地址字段。邮政编码字段将是空白，因为这个字段在新组件中被保持为空值，同时第一个（提示）值将在出生年份下拉列表中被选取（请参见图12.5）。

在无效提交时，用户将会看到以红色显示的错误消息，连同重新显示的实际用户输入（有效或无效的），如图12.6中所示。

如果我们需要对确认过程有更为精确的控制，我们的框架允许我们使用`FormController`对象背后的数据联编功能度（大多数框架不允许这种“手工”联编）。把同一个请求赋给几个对象甚至是可能的。摘自示例应用的下列代码展示了`com.interface21.web.bind.HttpServletRequestDataBinder`对象的这种用法。

```

RegisteredUser user = (RegisteredUser) session.getAttribute("user");
...
HttpServletRequestDataBinder binder = null;

try {
    binder = new HttpServletRequestDataBinder(user, "user");
    binder.bind(request);
    validator.validate(user, binder);

    PurchaseRequest purchaseRequest = new PurchaseRequest(reservation, user);
    binder.writeTarget(purchaseRequest, "purchase");
    binder.bind(request);

    // May throw exception
    binder.close();

    // INVOKE BUSINESS METHODS WITH VALID DOMAIN OBJECTS
}

catch (BindException ex) {
    // Bind failure
    return new ModelAndView("paymentForm", binder.getModel());
}

```



图12.5



图12.6

如果存在联编错误，无论是类型不匹配，还是由确认者所引起的错误，`HttpServletRequestDataBinder close()`方法都会抛出一个`com.interface21.validation.BindException`。由于这个异常，通过调用`getModel()`方法来获得一个模型映像是可能的。这个映像既含有所有被联编的对象（用户和购买），又含有错误对象。需要注意的是，这个API不是Web特有的。虽然`HttpServletRequestDataBinder`知道关于HTTP请求的情况，但`com.interface21.validation.DataBinder`的其他子类可以从数据源中获得属性值。当然，这个功能度完全基于`com.interface21.beans`组件操纵包——我们的基础结构的核心。

这里所介绍的有效性确认方法成功地填充域对象，而非填充Web特有的对象，如Struts `ActionForm`，而且使有效性确认与Web接口完全无关，使得有效性确认能够在不同的接口中被重用，而且能够在J2EE服务器外部被测试。

实现示例应用中的Web层

我们已经讨论了示例应用的Web层中所使用的许多概念。现在，让我们来简要地谈一谈它如何装配。

需要提醒的是，本节不打算讨论JSP视图，下一章将详细讨论视图技术。

概述

本节将讨论的Web接口基于我们迄今为止所考虑过的业务接口，比如`Calendar`和`BoxOffice`接口。

它使用单一控制器`com.wrox.expertj2ee.ticket.web.TicketController`，该控制器扩展前文讨论过的`MultiActionController`超类来处理所有应用URL。视图是JSP页，这些页面显示控制器所返回的模型组件，而且该控制器不是Web特有的，比如`Reservation`对象。这些JSP页不执行任何请求处理，并只含有少数脚本小程序（`Scriptlet`），因为控制器做关于视图选择的所有重要决定。

框架配置包括：

- 在`web.xml`文件中定义`ControllerServlet`（带有名称入场券）和`ContextLoaderServlet`（我们已经在前文中介绍方面的内容）。
- 创建XML应用上下文定义文件`ticket-servlet.xml`，其中定义了控制器服务器小程序所需要的各种组件——业务对象、Web应用框架配置对象和应用控制器。这包括：
 - 业务对象组件的定义（我们已在上一章中见过这些组件中的一部分）。
 - `TicketController` Web控制器组件的定义，同时设置它的组件属性。组件属性分成参数化组件行为（比如通过设置订票费来增加购买价）的属性和允许框架使应用业务对象可供它使用的属性。
 - 一个把所有应用URL都映射到控制器组件上的`HandlerMapping`对象。

下面让我们再来看一看`TicketController`组件的完整定义。和下文中的所有XML片断一样，这个定义摘自示例应用的`/WEB-INF/ticket-servlet.xml`文件。

```

<bean name="ticketController"
  class="com.wrox.expertj2ee.ticket.web.TicketController" >
  <property name="methodNameResolver" beanRef="true">
    ticketControllerMethodNameResolver
  </property>
  <property name="calendar" beanRef="true">
    calendar
  </property>
  <property name="boxOffice" beanRef="true">
    boxOffice
  </property>
  <property name="availabilityCheck" beanRef="true">
    availabilityCheck
  </property>
  <property name="userValidator" beanRef="true">
    userValidator
  </property>
  <property name="bookingFee">3.50</property>
  <property name="minutesToHoldReservations">1</property>
</bean>

```

从请求URL到控制器组件名的映射使用一个标准框架类来定义，如下所示。

```

<bean name="a.urlMap"
  class="com.interface21.web.servlet.UrlHandlerMapping">
  <property name="mappings">
    /welcome.html=ticketController
    /show.html=ticketController
    /bookseats.html=ticketController
    /reservation.html=ticketController
    /payment.html=ticketController
    /confirmation.html=ticketController
  </property>
</bean>

```

由于同一个控制器处理所有请求，因此所有映射都是到它的组件名上。

TicketController类的methodNameResolver属性是从MultiActionController中继承来的，而且定义请求URL到TicketController类中各方法的映射。被映射到TicketController上的每个URL都有一个对应的映射，如下所示。

```

<bean name="ticketControllerMethodNameResolver"
  class="com.interface21.web.servlet.mvc.multiaction.
  PropertiesMethodNameResolver">
  <property name="mappings">
    /welcome.html=displayGenresPage
    /show.html=displayShow
    /bookseats.html=displayBookSeatsForm
    /reservation.html=processSeatSelectionFormSubmission
    /payment.html=displayPaymentForm
    /confirmation.html=processPaymentFormSubmission
  </property>
</bean>

```

座位预订请求的处理

我们已经见过了TicketController对象的骨架。让我们来看一个完整的请求处理方法，这个方法试图为用户预订一个座位。

这个方法处理用于某特定演出场次中的某种特定座位类型的选择表单，如图12.7所示。



图12.7

当处理这个表单时，我们可以不理会无效字段或类型不匹配的可能性：参数中的两个是由该应用所写的隐藏表单字段（场次ID和座位类型ID），而且用户只能从下拉列表中输入若干个座位，这保证所有值都是数字的。表单提交的结果应该是如图12.8所示的屏幕，如果为该用户预订足够多的座位是可能的。

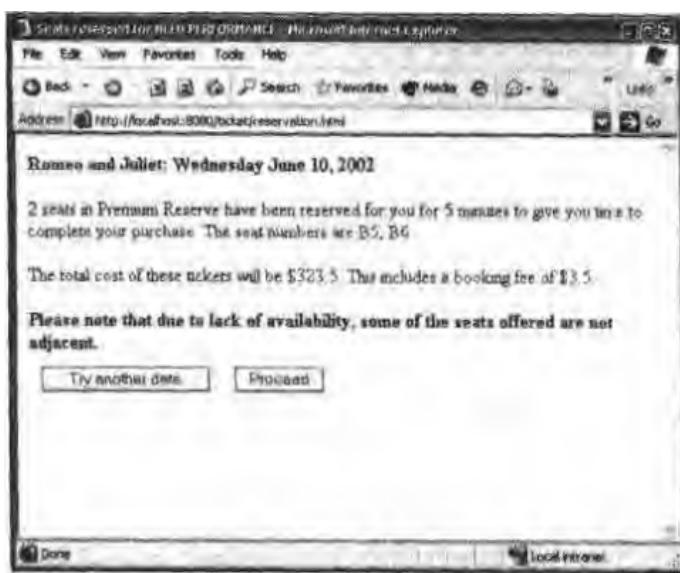


图12.8

如果没有足够的座位满足这个请求，用户将会得到试试另外一个日期的提示，并见到另一个的视图。

业务需求规定：如果订票获得成功，作为结果的Reservation对象应该被存储在该用户的会话中。在这个表单提交之前，用户可能还没有一个HttpSession对象。如果用户刷新或重提交这个表单，显示出来的预订结果应该是相同的。

这个方法的名称是reservation，匹配于我们的简单默认映射系统中的请求URL reservation.html。

由于无效输入时的错误处理不是问题，所以我们可以使用MultiActionController超类的自动数据联编能力。这将自动使用Class.newInstance()方法创建一个ReservationRequest型的新对象，并把请求参数联编到它上面。因此，这个方法的调用表明，我们有一个含有适当用户信息的ReservationRequest。

需要注意的是，我们只是把这个方法声明成抛出InvalidSeatingRequestException和NoSuchPerformanceException。这些异常都是不能出现在正常操作中的致命错误（事实上，它们只是已检查异常，因为我们不能放心地从EJB业务方法中抛出未检查异常）。MultiActionController超类将允许我们抛出我们所喜欢的任何异常，并且将抛出一个包装它的ServletException。

实际上，通过实现一个接受异常或其超类中的任一超类（相当于java.lang.Throwable）作为一个参数的异常处理器方法，我们有机会处理该异常。这样的处理器方法将在一个MultiActionController对象中的所有方法上被调用。不过，在这种情况下，几乎不需要这个功能度。

需要注意的是，声明请求处理方法只是为了抛出异常一般不是好主意；好的做法是必须决定哪些应用异常要捕捉及哪些异常要留给超类，而且在方法的约定中使这一点变得十分明确，就像我们在下面所做的那样：

```
public ModelAndView processSeatSelectionFormSubmission(
    HttpServletRequest request,
    HttpServletResponse response,
    ReservationRequest reservationRequest)
    throws ServletException,
    InvalidSeatingRequestException,
    NoSuchPerformanceException {
```

一旦这个方法被调用，ReservationRequest对象的属性就已成功地从请求参数中被填充完毕。这个方法的第一项任务是基于我们的当前配置把标准信息添加到用户在预订请求中提交的信息上——场次ID、座位类型ID以及被请求的座位数。TicketController类中的下列实例变量全部都是通过设置在XML组件定义元素中的组件属性来设置的：

```
reservationRequest.setBookingFee(this.bookingFee);
reservationRequest.setReserve(true);
reservationRequest.holdFor(this.minutesToHold);
reservationRequest.setSeatsMustBeAdjacent(true);
```

有了一个得到完全配置的ReservationRequest对象之后，我们就可以检查用户会话，看看是否已存在一个匹配于该请求的预订，指出相同表单数据的重新提交，这应该促使“Show Reservation”视图的重新显示。需要注意的是，我们使用一个fasle值调用了request.getSession()方法的、接受一个布尔参数的那种形式：如果没有会话存在，这种形式的调用不创建一个会话。另外还需要注意，笔者实现了一个预订是否满足Reservation类中某一请求的检查，而没有实现这个Web控制器。这使它能够在另一个接口中被重用，因为这样的检查不是Web特有的。

```

Reservation reservation = null;
HttpSession session = request.getSession(false);
if (session != null) {
    reservation = (Reservation) session.getAttribute(RESERVATION_KEY);
    if (reservation != null) {
        if (reservation.satisfiesRequest(reservationRequest)) {
            return new ModelAndView("showReservation", RESERVATION_KEY,
                reservation);
        } else {
            reservation = null;
            session.removeAttribute(RESERVATION_KEY);
        }
    }
}

```

如果这没有把用户直接送到“Show Reservation”视图，我们将需要调用BoxOffice业务对象来试着创建一个Reservation。TicketController的boxOffice实例变量在应用启动时用一个组件属性来设置，所以查找一个BoxOffice实现没有任何结果。BoxOffice对象的allocateSeats()方法要么返回一个Reservation，要么抛出一个NotEnoughSeatsException；我们将捕捉这个异常，而且它将导致一个不同视图的显示。一旦成功地创建了一个预订，我们就把Reservation对象放到用户的会话中，在必要时通过调用request.getSession(true)来创建一个新会话：

```

try {
    reservation = boxOffice.allocateSeats(reservationRequest);
    session = request.getSession(true);
    session.setAttribute(RESERVATION_KEY, reservation);
    return new ModelAndView("showReservation", RESERVATION_KEY, reservation);
}
catch (NotEnoughSeatsException ex) {
    return new ModelAndView("notEnoughSeats", "exception", ex);
}
}

```

这个请求处理方法未含有表示特有的代码。它只处理业务对象和模型数据：“Show Reservation”视图可以自由地使用任一视图技术，按照它所喜欢的任一方式显示预订确认。这个方法未含有业务逻辑：它所应用的各种规则（比如“在一接到一个重复表单请求时，就重新显示一个预订”）已被限定在用户接口上。所有业务处理都由BoxOffice接口来完成。

在下一章中，我们将了解如何使用包括JSP、XSLT和WebMacro在内的多种技术来再现这个屏幕的视图。

实现的回顾

那么，在满足我们自己为示例应用的Web层所设定的目标时，我们大体上是怎么做的呢？

我们实现了一个“干净的Web层”吗，也就是说，其中的控制流程由Java对象控制，与表示分开，而且用模板来处理？

Java代码中无任何置标，JSP页之类的视图只需要执行循环。正如我们将要在下一章中看到的，不用脚本小程序也能实现视图。

我们一直没有专门使用某一特定的视图技术，尽管我们使用JSP作为示例应用的视图技术，因为它始终是容易获得的，也不存在使用另一种视图技术的更好理由。

我们实现了一个“细薄Web层”吗，也就是说，它具有最少的代码量，Web层与业务逻辑分离得尽可能充分？

我们的Web层Java代码只由一个类组成，即com.wrox.j2eedd.ticket.web.TicketController类，它的长度不足500行。无其他类依赖于该Web框架或Servlet API，也无其他类可能只与一个Web接口相关。输入确认代码不依赖于Web应用框架或Servlet API。如果应用中的屏幕个数发生增长，这个控制器可以被分解成几个控制器。我们甚至拥有在大型应用中引入附加控制器服务器小程序的选择余地。不过，现在使用更多的类没有任何好处。

小结

在本章中，我们已经见识了在Web接口中把表示与控制逻辑分离开为什么是至关重要的，以及Web接口是建立在定义明确的业务接口上的一个薄层为什么也是同样重要的。

我们已经知道了初始Web应用体系结构的许多缺陷，比如“JSP Model 1”体系结构，它里面JSP页用来处理请求。我们已经知道Java对象是Web应用中处理控制流程的正确选择，以及JSP最好只用做一种视图技术。

我们已经了解了MVC体系结构模式（也叫“Model 2”或“Front Control”体系结构）能够帮助实现我们为Web应用所确立的设计目标。我们已经知道把控制器构件（处理Web层中的控制流程）与模型构件（含有要显示的数据）及视图构件（显示模型数据）分离开的重要性。这种分离有许多好处。例如，它使我们能够用域对象作为模型，不必非要创建Web特有的模型不可。它保证我们能够在不影响控制流程的情况下轻松地修改表示；在站点的表示发生改动时最大限度地减少引入功能性错误可能性的一种重要方式。它实现Java开发人员与表示开发人员角色之间的明确分离：大型Web应用中的关键分离。我们知道彻底的“视图可替换性”（在不修改控制器或模型代码的情况下用另一个显示相同数据模型的视图替换一个视图的能力）是分离开控制器与视图构件的理想级别。如果我们实现它（而且它是可实现的），就能在不影响控制流程的情况下更换视图技术。

我们分析了3个开放源MVC框架（Struts、Maverick和WebWork），并弄清楚真正的MVC实现怎样共享许多常见概念。我们研究了示例应用中所使用的MVC框架的设计，这个设计结合了这3个开放源MVC框架的部分最佳特点。这个框架的设计目标是用在实际的应用

中，而不仅仅作为一个演示。它以我们在上一章中所讨论的基于Java组件的基础结构为基础；该基础结构使得这个框架配置起来很容易，也使得Web层构件能够在不依靠其具体类的情况下轻松地访问应用业务对象。

我们讨论了Web层状态管理的问题。首先，我们探讨了保证使维护有状态Web层的应用仍可缩放的各种对策，然后考虑了把会话状态保存在客户浏览器和隐藏表单字段中的可选策略。

我们讨论了Web应用中的数据联编（请求参数值到Java对象属性的填充）问题，以及Web应用框架怎样解决表单提交和有效性确认问题。

最后，我们见识了示例应用中的Web层的设计与实现，描述了它怎样满足我们自己为Web层所确立的设计目标。

迄今为止，我们一直很少关注JSP或XSLT之类的视图技术。在下一章中，我们将较详细地了解一下Web层视图，以及我们在使用一种MVC方法时怎么才能显示模型数据。

第13章 Web层中的视图

在上一章中，我们把讨论的重点放在了Web应用应该怎样处理控制流程和访问业务对象上。在本章中，我们将较仔细地看一看用于J2EE Web应用的视图技术。

在很大程度上，视图技术的选择可以（而且应该）独立于Web工作流程。我们从了解分开视图与控制器构件的优点开始。然后，全面纵览可用于J2EE Web应用的一些领先视图技术。我们将讨论每种视图技术的优缺点，从而使读者能够为每个项目做出正确的选择。

虽然J2EE标准定义至少一种置标生成方法是必需的，但各种J2EE规范中对JSP所给予特殊重视却产生了一些令人遗憾的结果。尽管JSP是再现内容的一种重要方法，但也仅仅是许多有效可选方法中的一种。如果顺利的话，上一章已经做了一个令人心悦诚服的选择：服务器小程序和委托Java对象（“控制器”）应该用来控制Web层中的工作流程，而不是JSP页。通过把Java语言的大部分威力带进表示模板中，JSP也引入了和好处一样多的危险。当我们使用JSP页时，采用严格的标准来保证这些危险不产生严重的后续问题是至关重要的。

在本章中，我们将以一种开放式思路来探讨用于J2EE应用的JSP和一些领先的可选视图技术，以及如何判定哪种视图技术最适合解决给定问题。我们将介绍下列每种视图技术的优缺点，并去看一看它们在实际使用中的情况。

- **JSP。** 我们将介绍使用JSP时必须加以避免的陷阱，以及使用JSP页来保证利用了它们的Web应用保持干净和可维护的最佳方法。我们还将介绍新技术JSP Standard Template Library (JSTL)，该技术在帮助JSP页作为有效视图构件方面提供了重要的支持。
- **Velocity模板引擎。** 这是一个简单、易学而又受制于Servlet API的模板化解决方案。这是一个重要的意外收获，因为它可以使模板在应用服务器外部进行测试。
- **基于XML和XSLT的方法。** XSLT提供了非常强有力的内容再现能力以及使表示与内容充分分离的能力，付出的代价是一个可能复杂的创作模型和实质性的性能开销。
- **XMLEC。** 这是一种基于DOM的内容生成方法，首先在Enhydra应用服务器中引进，它能使动态页面从静态的HTML置标中生成。
- 允许二进制内容生成的解决方案。虽然这些不是上面所列举的面向置标视图技术的替代方案，但任何设计完善的Web应用都必须能在必要时生成非置标内容。我们将使用PDF生成——一种常见需求作为示例，说明由生成非可读内容与二进制内容所引起的各种挑战。

无论使用何种视图技术，我们都应该不必修改控制器中的代码。如果我们采用第12章中所讨论的MVC Web体系结构，控制器与视图之间是完全分离的。

为项目选择恰当的视图技术十分重要，因为它可以使满足表示需要变得十分容易。但是，这一选择在很大程度上可以（而且应该）独立于应用代码的其余部分。

决定性因素有可能包括如下这些：

- 给定的解决方案适合当前问题的程度。此处的指导标准是每个解决方案所必需的代码量和复杂性。
- 性能考虑。在本章中，我们将考虑几个主要的性能问题，而在第15章中，我们将介绍这里所讨论的一些视图技术的基准点。
- 项目团队中的技能。例如，如果团队拥有强大的JSP技能，但在可选视图策略方面没有太多经验，那么JSP是明显的选择。如果某些团队成员有过Microsoft ASP经历，那么他们很可能熟悉JSP。相反，在拥有XSLT经验的团队中，XSLT可能就是首选视图技术。如果内容开发人员当中没有JSP或XSLT经验，那么像Velocity那样的一种简单模板语言可能是最佳选择，因为它几乎没有学习困难。

需要记住的是，在几乎所有非常简单的Web应用中，页面布局将由HTML开发人员来维护。因此，在选择视图技术时，我们必须考虑到他们的技能和偏好。实际的J2EE开发涉及到多个开发团队，而不是只涉及Java开发团队。

在本章中，我们将使用示例应用中的视图之一作为示例。随着我们分析每种视图技术，我们将实现这个视图来提供一个实际的例子。

我们还将介绍视图合成（View composition）的重要概念：通过组合其他视图的输出结果或页面内容来建立复杂的页面。视图合成是建立实际Web站点上所需复杂页面的一个重要技巧。我们将利用实际的例子来分析视图合成的两种常用方法。

本章不打算做为使用此处所讨论的每种视图技术的一个指南，而是打算做为介绍如何在MVC Web应用中使用每种技术的一个综述和纵览每种视图技术的优缺点的一个高层面视点。每一节都含有关于相关技术的进一步信息的参考。

虽然大多数示例将使用第12章中所讨论的MVC Web应用框架，但我们所讨论的概念与所有Web应用，尤其是与使用了Struts和其他MVC框架的应用都有关系。本章将把重点放在读者在应用代码中需要做些什么，才能以最少的框架特有内容使用此处所讨论的每种视图技术。要想较详细地了解如何安装和配置此处所讨论的每种视图技术，以及该框架对每种视图技术的内部支持的实现，请参见附录A。

控制器与视图的分离

控制器与视图的分离对自由地实现本章所描述的视图是必不可少的。

控制器与视图的分离基于以下两条原则。

- 使用一个模型来包含由处理请求所产生的所有数据。这意味着视图对调用业务操作永远都不是必需的，只对显示一个完整的模型才是必需的。
- 使用上一章中所讨论的命名视图策略（Named view strategy）。这个间接层能使一个控制器按名选择一个视图，同时又无需知道该视图的任何实现情况。框架基础结构可以在运行时“分解”视图名来查找该名称所关联的共享实例。

控制器与视图的分离带来许多好处。例如：

- 它是保证分离Java开发人员与UI开发人员角色的最佳方法。这样的分离在几乎所有

小规模Web开发中都是十分重要的，因为这些角色的每一种都需要专家技能。

- 它强制执行运用MVC模式时的纪律，保证控制器、模型与视图的责任是明确定义的。
- 它保证能够在不影响站点控制流程或不破坏功能度的情况下修改站点的表示。
- 它允许视图合成。在视图合成中，多个视图的输出结果或页面内容被组合起来，同时又不影响Java代码。
- 它允许独立测试控制器和模型。
- 它能使我们支持任一视图技术，同时又不影响Java代码。

正确执行这样的分离绝不会增加任何复杂性。

MVC J2EE Web应用中的视图应该是“Java组件把什么XSLT建模到XML”。有了明确的责任划分之后，不同的视图可以轻松地用不同的格式表示相同的数据，同时视图可以由不懂Java程序设计的专家来编辑。

在上一章所讨论的框架内，com.interface21.web.servlet.View接口提供这种分离，提供了在控制器—返回模型数据时，控制器服务器小程序就可以调用的一个标准Java接口。这种视图接口概念不是这个框架所特有的。例如，Maverick框架就使用了一个类似的视图接口（笔者在设计当前框架时就受惠于这个视图接口）。

View接口中最重要的方法是下面这个方法——基于控制器所返回的模型数据建立响应对象：

```
void render(Map model, HttpServletRequest request,
           HttpServletResponse response)
           throws IOException, ServletException;
```

图13.1所示的有序图描述了控制器如何选择一个视图，把视图名和模型数据返回给控制器服务器小程序。接着，控制器服务器小程序使用一个ViewResolver对象来查找被映射到该视图名上的视图实例，然后利用控制器返回的数据模型来调用该对象的render()方法。

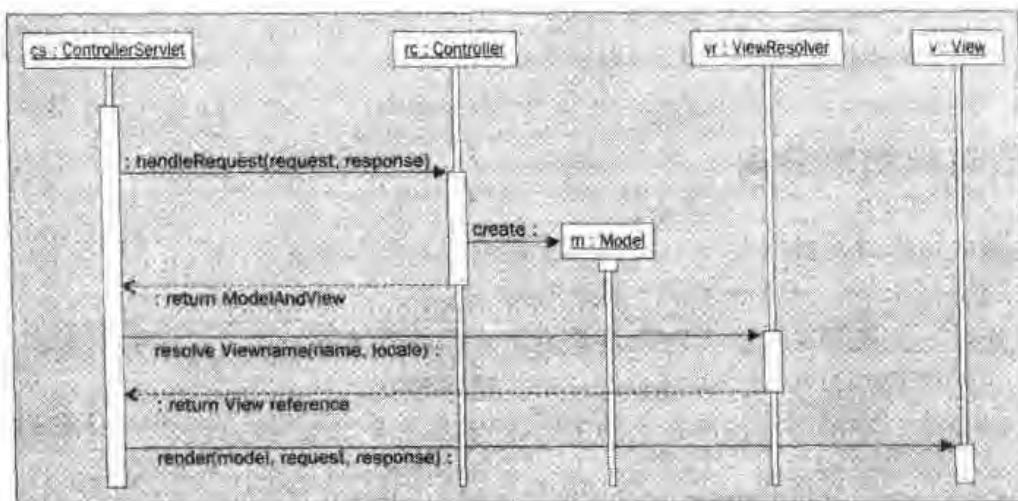


图13.1

视图接口的实现必须满足一些基本需求：

- 包装一个特定的页面结构，常常被保存在JSP或另一种模板语言的一个模板中。
- 接受控制器用非视图相关的格式所供给的模型数据，并把它暴露给已得到包装的视图技术。
- 使用Servlet API HttpServletRequest和HttpServletResponse对象来建立一个动态页面。

每个视图都有一个线程安全的共享视图定义。视图通常是Java组件，因而允许它们的配置被保存在Java代码外部。

除了控制器所提供的动态模型数据之外，在我们框架中的视图上设置静态属性（Static attribute）也是可能的。静态属性在初始化时被设置，并且是一个具体视图实例的定义的一部分。静态属性可以用来设置表示特定的、不随模型数据变化的值。静态属性的一个典型用法是在由多个构件组成的模板视图中定义页面结构（我们将在本章后面的“视图合成和页面布局”一节中讨论这种用法）。由于静态属性与视图相关，与模型或控制器无关，所以可以在不必修改Java代码的情况下添加或更改它们。

为了使框架能够分析视图名，每个视图名必须有一个视图定义。大多数MVC框架都提供一种把视图名映射到定义的方法。

当前框架在视图定义如何存储方面提供了极大的灵活性。这取决于正得到使用的ViewResolver接口的实现。在示例应用所使用的默认ViewResolver实现中，视图定义是WEB-INF/classes/views.properties文件中的Java组件定义。第11章中已经介绍过基于属性的组件定义语法。

在附录A中，读者将会了解到框架所包含的View接口的实现，这些实现支持本章中所讨论的所有视图技术。

构造预订页面的视图

由于我们要在本章中分析不同的视图技术，因此要使用一个简单的动态页面作为示例。在上一章中，我们分析了com.wrox.expertj2ee.ticket.web.TicketController类的processSeatSelectionFormSubmission()方法中的控制器代码。该方法处理表单的提交，这个表单允许用户申请预订某一特定场次的某一指定座位类型（比如“Premium Reserve”）的若干个座位。

表示的信息与必需的格式化

处理该请求可以产生两个视图：

- 一个视图显示刚建成的预订，并要求用户接着购买那些座位（“Show Reservation”视图）。
- 一个视图通知用户本场次没有足够的空余座位满足他们的请求（“Not Enough Seats”视图）。

在本章中，我们将使用“Show Reservation”视图作为例子。“Not Enough Seats”视图则截然不同。使用控制器来处理请求的部分意义在于：它使我们能够依据必要业务操作的结果来选择几个可能的视图之一。“Show Reservation”视图屏幕看起来如图13.2所示。

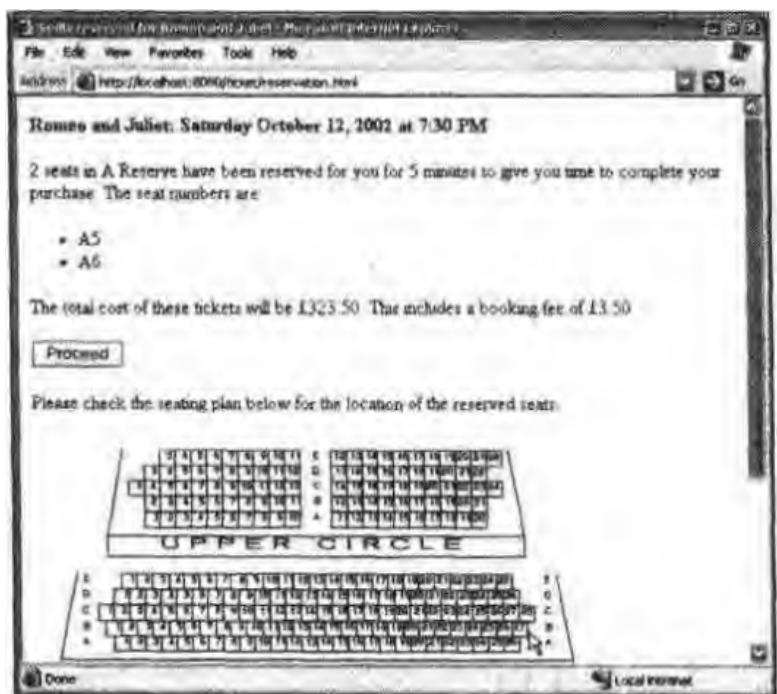


图13.2

该视图显示如下动态信息：

- 正在上演的节目的名字及本节目在一个座位安排计划图。
- 本场演出的日期。
- 预订的座位数，该预订的有效期（如果当前用户没有继续购买这些座位，其他用户多久之后可以预订它们）、座位的名称以及购买这些座位的价钱（包括订票费）。货币显示和日期/时间显示应该适合用户所在地区。

这个视图有一个变体，只有当有足够的座位满足用户的请求但它们不相邻时，才显示这个变体。在这种情况下，应该给予用户放弃这次预订和试订另一日期的机会。我们需要另加一个区域来突出这个可能的问题并提供一个试订另一场次的链接。无论使用何种视图技术，我们都必须能够处理这个简单的条件：这是表示上有一点较小差别的同一个视图，而不是一个像notEnoughSeats视图那样的不同视图。如图13.3所示。

这个视图背后的模型

控制器的processSeatSelectionFormSubmission()方法在它选择“Show Reservation”视图时所返回的模型中含有如下3个对象：

- **performance**

这个对象含有关于本场演出及其父节目的信息。它具有com.wrox.expertj2ee.ticket.referencedata.Performance类型。

- **priceband**

这个对象显示名称之类的信息（上面插图中的“Premium Reserve”）和被请求座位类型的价格。它具有com.wrox.expertj2ee.ticket.referencedata.PriceBand类型。

- **reservation**

这个对象含有关于用户预订的信息，比如预订的座位和这些座位是否相邻。它具有**com.wrox.expertj2ee.ticket.boxoffice.Reservation**类型。

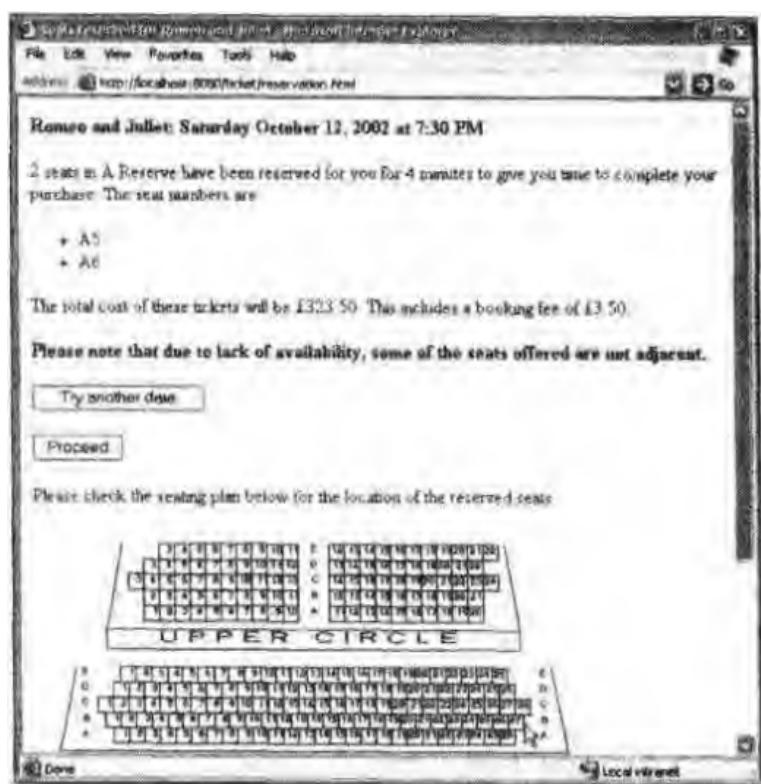


图13.3

performance和**priceband**对象是参考数据，在本应用的所有用户之间共享。**reservation**对象是当前用户所特有的。

这3个类型都是接口，不是类。返回对象将含有暴露方法的实现类，这些方法允许它们的一些状态被操纵。这些状态最好从视图中被隐藏掉，因此我们选择不把视图编码到这些易变类（视图应该把模型看做只读的）。使用基于接口的设计也提供不同实现的可能性。

下面依次来看一看上述每个接口的一些重要方法。关于每个接口的完整代码清单，请参考示例应用源代码。

performance和**priceband**接口都是一个抽象分级继承性结构的一部分，并且这个分级结构基于**com.wrox.expertj2ee.ticket.reference.ReferenceItem**接口，该接口暴露一个数字id和一个名称或代码，如下所示：

```
public interface ReferenceItem extends Serializable {
    int getId();
    String getName();
}
```

保证参考数据对象之间的这种基本通用性是很值得的，因为它节省了具体实现中的少量代码（这些具体实现使用一个并行的继承性分级结构），并使对象的一致性处理变得很容易。例如，参考项可以按id来索引，以便为快速查找创造条件。

图13.4所示的UML类图说明了这个继承性分级结构。

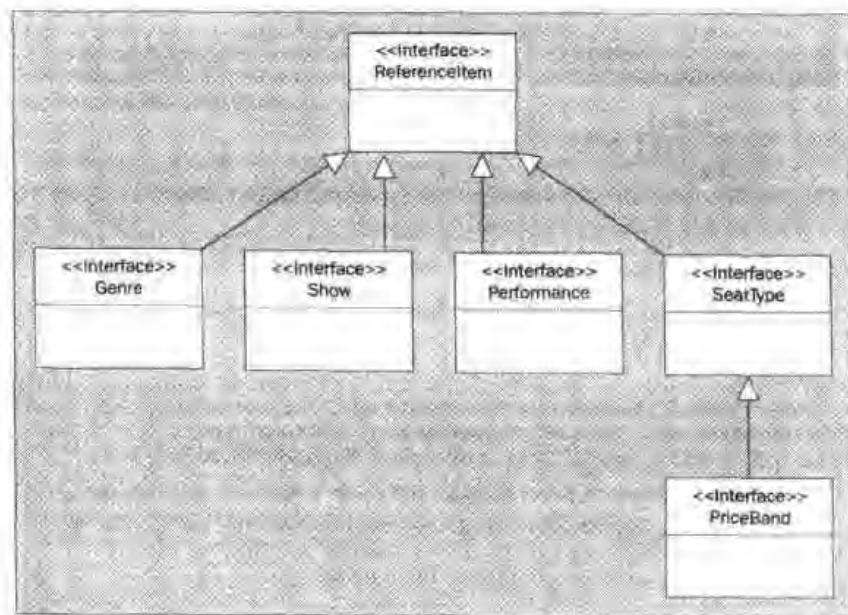


图13.4

图13.5描绘了上述类型的参考数据实例在运行时怎样被装配成一个树。每个对象都提供通过其子女向下导航的方法（为了简单起见，笔者只展开了一个Show和一个Performance）。

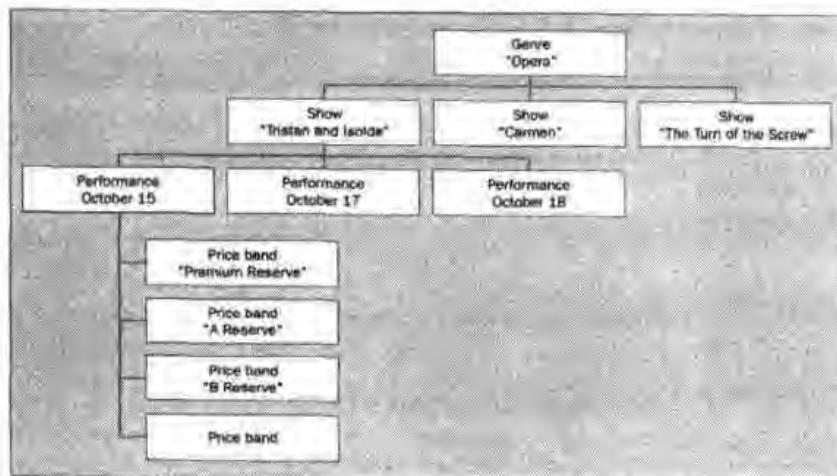


图13.5

Performance接口（“Show Reservation”视图模型的一部分）扩展ReferenceItem来暴露一个指向父对象Show的链接，此外还暴露本场次的日期/时间和由本场次的PriceBand对象构成的一个列表。

```

public interface Performance extends ReferenceItem {
    Show getShow();
    Date getWhen();
    List getPriceBands();
}
  
```

PriceBand对象把座位价格增加到由SeatType接口所暴露的、关于座位类型的信息上，比如名称与代码（如AA）以及描述（如Premium Reserve）。下面是这两个简单接口的一个完整代码清单：

```
public interface SeatType extends ReferenceItem {  
    int getSeatTypeId();  
    String getDescription();  
}  
  
public interface PriceBand extends SeatType {  
    public double getPrice();  
}
```

com.wrox.expertj2ee.ticket.boxoffice.Reservation接口暴露用户特定的信息。在为用户预订座位时，一个Reservation对象得到创建，并含有一个参考，用于指向ReservationRequest对象（由用户生成的一个命令），正是这个ReservationRequest对象才使该预订得以进行。一个Reservation对象不是参考数据，而是在用户成功地完成订票过程的第一步时产生的动态数据。需要注意的是，Reservation对象是可串行化的。我们将需要把Reservation对象放在一个用户会话中，需要保证这在聚类化环境中是可能的。

笔者在下面突出了与视图相关的方法。

```
public interface Reservation extends Serializable {
```

下面的方法返回一个数组，其中包含了预留给该用户的Seat对象。

```
Seat[] getSeats();
```

下面的方法指出这些座位是否相邻。正如我们已经见过的，表示逻辑可能依赖于这个信息，但确定一组座位是否相邻却不是表示逻辑，因为它包含对相关座位安排计划的了解。

```
boolean seatsAreAdjacent();  
  
int getMinutesReservationWillBeValid();  
  
double getTotalPrice();  
  
ReservationRequest getQuoteRequest();
```

下面的方法仅由控制器使用：

```
long getTimestamp();  
String getReference();  
boolean isReserved();  
boolean satisfiesRequest(ReservationRequest r);  
}
```

了解下面所描述的视图代码时，请参考这些代码清单。

这些模型对象没有一个是Web特定的。事实上，它们甚至不是UI特定的，这意味着我们可以轻松地编写检查预期结果的测试工具，并可以方便地使用这些对象作为Web服务接口的基础。

模型原理

这些模型的设计提出了与Web应用中可能用到的所有模型都相关的几个重要问题。

- 模型应该是Java组件，以保证它们给视图提供最大价值。需要注意的是，这些类为显示目的而暴露的所有方法都遵守Java组件命名模式：例如，**Reservation**接口借助于一个**getTotalPrice()**属性获得器，而不是借助于一个**TotalPrice()**方法暴露一个**TotalPrice**属性。如果我们暴露方法，而不是暴露组件属性，JSP和其他几种视图技术访问模型信息的能力可能就有限。
- 有些简单的计算是在模型中完成的，即使它们可以由视图来完成。例如，从一个**Reservation**对象所含有的**ReservationRequest**对象的**holdTill**属性和系统日期中，计算出这个**Reservation**对象期满前的秒数是可能的。因此，严格说来，**Reservation**对象的**minutesReservationWillBeValid**属性是多余的。

但是，并非所有视图技术都可能允许我们轻松地访问系统日期或所需计算的实现。如果这个模型对象有几个视图，那么每个视图都需要实现该计算的各自版本——可能会产生严重影响的冗余度。模型设计是一个与关系数据库设计有很大差别的问题：避免冗余没有任何奖励。但是，涉及到地区特定信息（比如数值格式化）的处理通常应该在视图中完成，因为这完全是一个表示上的问题。

- 同样，模型中的冗余信息没有经过精选。例如，严格说来，**PriceBand**对象也是多余的。让一个视图通过比较座位类型id与表单提交所产生的**seatTypeId**请求参数，来导航到**Performance**对象的所有**PriceBand**子女中的相关**PriceBand**对象是可能的。但是，在视图中提供这种“冗余”信息有充分的理由。访问请求参数会侵害控制逻辑（在视图中不合适）。此外，并非所有视图都能访问请求参数；例如，JSP视图能，而XSLT格式表则不能。视图应该不必检查当前请求或任何用户会话就能从数据模型中获得它们所需要的所有信息。由于模型中的**PriceBand**对象是一个参考，用于引用**Performance**对象所保存的一个**PriceBand**对象，所以我们只是把查找移到了控制器内，而不是浪费一个多余对象上的内存。

由于第12章中所介绍的Web应用框架允许我们返回一个Map而非一个单独的对象作为模型，所以我们不必创建单独的容器对象来保存在一个对象中的所有这些信息。这样的对象将是这个页面所特有的，所以凭借不必创建一个对象，我们避免了创建将是一个Web接口特定对象的东西。

Web接口中所用的模型对象不必（而且不应该）是Web特定的。例如，它们绝不应该暴露置标或其他格式化信息。

遵守下面这几个重要的规则将会保证模型对象给视图提供最大的价值，无论我们使用了何种视图技术。

- **模型对象应该是Java组件**

视图应该能够通过访问组件属性来获得它们所需的全部数据，并且应该不必在模型对象上调用方法。

- 模型对象应该很聪明，以便显示它们的视图可以很笨

这并不意味着模型对象应该执行任何数据检索；在模型被Web层控制器所调用的业务对象返回之前，这种检索应该已经完成。不过，这意味着模型应该多花费点心思把东西变得容易让视图显示；例如，通过执行重要的计算，并暴露计算结果，即使视图可以从模型的其他组件属性中获得该计算的那些输入值。

- 模型应该暴露视图所需要的所有信息

应该没有必要——因此也没有理由让视图访问`HttpServletRequest`或`HttpSession`对象。使用不完善的模型会把视图实现限定在那些使访问底层Servlet API对象很容易的视图技术上，比如JSP。

- 通常，不是模型而是视图应该处理地区特定的问题

模型对象不一定是显示特定的，因此应该不必处理本地化问题。

有时，这4条规则中的第二条最好能够通过创建一个适配器（Adapter）组件来得到遵守。所谓适配器是指这样一个值对象：它以一种使访问变简单的方式，高速缓存并暴露一个或多个视图所需要的全部模型数据。例如，适配器可能通过组件属性暴露了方法调用在几个具有复杂接口的非组件对象上所检索出的数据。在这样的情况下，适配器简化了视图可以利用的数据接口，并消除了使人分心的多余信息。示例应用不需要这种适配器方法，因为像`Reservation`对象那样的域对象自然适合用做模型，没有必要做任何设计折衷。

如果用了一个适配器组件，它应该由相关的控制器创建，而不应该由视图创建。虽然让JSP页创建适配器组件也很容易，而且这种方法有时又得到拥护，但这会破坏视图可替换性原则。

JSP视图

JSP已经逐渐成为用于J2EE Web应用的最流行视图技术，主要原因是Sun公司把它作为J2EE的一个核心部分来颂扬。

JSP视图提供了下列好处：

- 它们开发容易。请求式重编译（至少在开发期间）意味着快速的开发 - 部署周期。
- Java作为一种脚本语言的使用使访问模型等Java应用对象变得更容易。
- 优良的性能，因为JSP页被编译成服务器小程序。但是，正如读者在第15章中将要看到的，我们不应该假设这就意味着JSP将始终是最高性能的视图选择。
- JSP是J2EE标准之一。有论述JSP创作的大量文献及大量的工具支持。许多开发人员都熟悉JSP。
- JSP对自定义标志（下文讨论）的支持意味着第三方标志库可用于多种用途，以及能够以一种标准方式扩展JSP的各种能力。

标准化通常是一件好事，但也存在一种危险：一个有缺陷的解决方案仅仅因为它是一个标准而得到太广泛的使用。作为一种有问题的视图技术，JSP也有一些明显的缺陷，如下所述：

- JSP的起源早于MVC在Web应用中的使用，而且它也表露了这一点。JSP程序设计模

型应归功于Microsoft公司的Active Server Pages (ASP) 技术，而这项技术要追溯到1996年。ASP使用了类似的语法，以及在可执行脚本与静态置标之间转义的同一种模型。Microsoft公司最近放弃了该模型，因为经验表明ASP应用变得凌乱和难以维护。

- 脚本小程序（JSP页中的Java代码块）可以用来执行包括EJB层访问在内的复杂处理，而不是只支持视图再现。这不是一件绝对的事情。它太诱人以至于不能妄用这种威力，这也是反对JSP的主要理由之一。如果存在一个方便的工具，并且它又容易使用，肯定会有人去使用它，而不顾将来的后果。
- 存在使用JSP页处理控制逻辑的诱惑——同样是因为它容易使用。JSP页提供了用于请求处理的基础结构，其中包括请求参数到Java组件的映射；该基础结构在实践中害处大于用处。
- JSP语法是相当笨拙和易出错的。不过，该缺陷可以通过适当地使用JSP标志库和创作工具来加以弥补。
- JSP页中的语法错误会产生令人迷惑的错误输出。JSP页会同时遇到转换时与运行时错误。错误消息的质量随着Web容器的不同而有所不同。
- 由于JSP语法包括了在XML中非法的字符，所以不可能保证JSP文档将生成构造正确的标志。生成构造正确的置标可能是一个重要的考虑因素。JSP规范为JSP定义了一个XML语法，但它不是为手工创作而计划的。
- JSP页难以进行独立测试，因为它们不是正当的Java类，而且它们除了依赖于Servlet API之外，在转换时和运行时还依赖于Web容器。因此，我们通常只能把JSP页作为Web应用验收测试的一部分来进行。
- 在实践中，JSP所关联的各种缺陷毫无疑问是有害的。要想了解JSP页的各种缺陷，请参见关于这个题目的下列文章。
 - <http://www.servlets.com/soapbox/problems-jsp.html>。投给JSP讨论会的一篇早期而又被经常引用的稿件，其作者是Jason Hunter——他出版了几本关于服务器小程序的图书。
 - <http://jakarta.apache.org/velocity/ymtd/ymtd.html>。这基本上是Velocity模板系统的一篇广告（我们将在下文中讨论），但对JSP做了许多有根有据的批评。

在本节的其余篇幅内，我们将看一看如何避开JSP的各种缺点而利用它的各种优点。从本质上说，这就相当于最大限度地减少JSP页中的Java代码量。

下面是关于如何在可维护的Web应用中使用JSP基础结构的一段讨论，而不是关于JSP基础结构本身的一个使用指南。如果必要，请参考关于JSP的参考资料。

JSP页正得到正确使用的一个象征，是在JSP页不违背第12章中所讨论的视图可替换性的时候。例如，如果一个JSP页（不正确地）用来处理请求，在不破坏应用功能度的情况下，一个JSP视图就无法被一个XSLT视图替换。

需要避免什么

为了演示错误使用JSP的后果，让我们来看一个JSP页的例子，这个JSP不仅是一个视图，而且还使用Java的威力来处理进入请求（“Model 1”风格），并使用J2EE API。

由于“Show Reservation”视图背后的控制逻辑会使一个JSP Model 1版本变得太长，所以笔者使用一个简单的页面，这个页面允许我们按演出场次ID和座位类型查询座位空余情况。这个页面的一个较现实版本会给出下拉列表，允许用户输入这些值，但为了方便举例，笔者忽略了这个方面，连同对用户输入非数字值的关心。

这个JSP页（model1.jsp）既显示一个允许用户输入演出场次和座位类型的表单，又处理结果。如果请求是一个表单提交，空余座位的数量将被显示在表单下面；否则，仅显示表单。这个页面的输出看起来将会像图13.6所示。

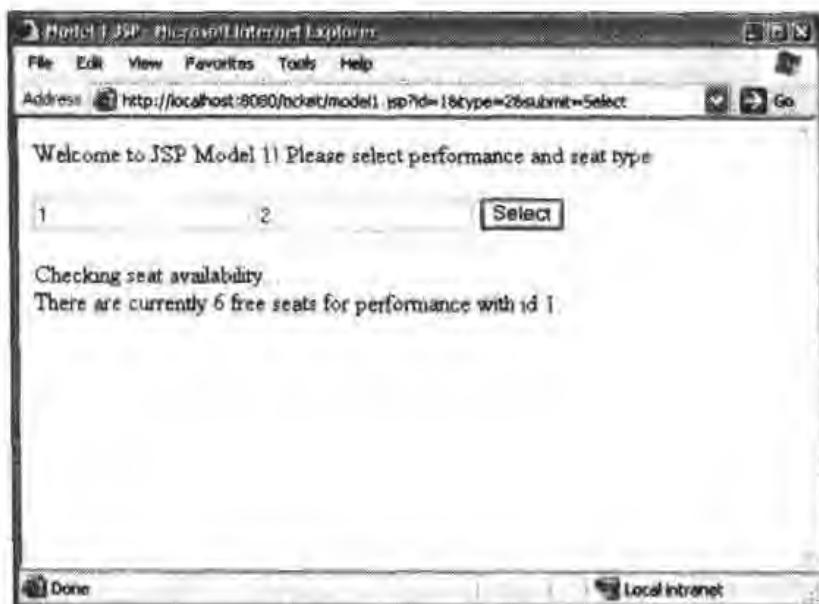


图13.6

该列表从导入被使用的Java对象（J2EE与应用特定的两种Java对象）开始。需要注意的是，我们必须使用JSP错误页机制（做了突出处理的那一行），以避免需要捕捉该页面上可能出现的每个异常。

```
<%@ page errorPage="jsp/debug/debug.jsp" %>
<%@ page import="javax.naming.InitialContext" %>
<%@ page import="com.wrox.expertj2ee.ticket.boxoffice.BoxOffice" %>
<%@ page import="com.wrox.expertj2ee.ticket.exceptions.NoSuchPerformanceException" %>
<%@ page import="com.wrox.expertj2ee.ticket.boxoffice.ejb.*" %>
```

接下来，我们声明一个简单的组件，它的属性匹配于预计的请求参数，因此我们不必与该请求直接打交道。Model1Bean类被包含在示例应用所携带的源代码中，它只暴露两个int属性：id和type。需要注意的是，我们使用<jsp:setProperty>标准动作从请求参数中填充这些属性。为了便于举例，我们不关心用户输入非数字输入。

```
<jsp:useBean id="queryBean"
    class="com.wrox.expertj2ee.ticket.web.Model1Bean"
    scope="session">
    <jsp:setProperty name="queryBean".property="" />
</jsp:useBean>
```

接下来，有少许模板数据：

```
<html>
<head>
<title>Model 1 JSP</title>
</head>
<body>
Welcome to JSP Model 1!
Please select performance and seat type
```

不管是不是处理一个表单提交，我们都显示表单。表单字段被预填上组件的属性值，在首次显示该页面时这些值都是0。

```
<form method="GET" >
<input type="text" name="id"
       value="

```

如果这个JSP页没有被表单提交调用，这就是我们所看到的全部内容。如果它正在处理一个表单提交（我们可以通过检查一个submit请求参数的出现与否来判断），脚本小程序在该页面的其余部分内生效。

脚本小程序执行一个对BoxOffice EJB的JNDI查找。一旦它获得这个EJB，它就查询这个组件，以寻找与这个组件中所保存的演出场次id和座位类型相对应的座位空余情况。我们不必费劲捕捉JNDI或EJB API错误，而是让JSP错误页机制处理这样的错误。增加另外的try/catch块是白找麻烦。

生成的输出是变化的，不管有没有一个具有指定id的演出场次。如果没有，BoxOffice getFreeSeatCount()方法将抛出一个NoSuchPerformanceException（由JSP捕捉），以提供一次生成不同输出的机会。笔者已经对生成输出的那些行做了突出处理，它们占了JSP页面源代码的不到一半。

```
<% if (request.getParameter("submit") != null) { %>

    Checking seat availability...<br>

<%
    int freeSeats = 0;
    InitialContext ic = new InitialContext();
    Object o = ic.lookup("java:comp/env/ejb/BoxOffice");
    BoxOfficeHome home = (BoxOfficeHome) o;

    BoxOffice boxOffice = home.create();

    try {
        freeSeats = boxOffice.getFreeSeatCount(
            queryBean.getId(), queryBean.getType());
    }
    There are currently <%=freeSeats%> free seats
    for performance with id <jsp:getProperty name="queryBean" property="id" />.

<% } catch (NoSuchPerformanceException ex) { %>
```

```
There's no performance with id <jsp:getProperty name="queryBean"
property="id" />.
<br>Please try another id.

<% %>
```

最后，无论如何都要显示一些结束代码的模板数据，如下所示：

```
</Body>
</HTML>
```

虽然这个JSP页没有完成实际的功能，但它的确展示了在现实中可以见到的问题。大多数经验丰富的J2EE开发人员已经见到过许多这样的JSP页——而必须清理由这样的JSP页引起的问题。

这个JSP页出了什么毛病，这样的JSP页在实际的应用中为什么总是变成一个可维护性灾难？原因有许多，主要包括如下这些：

- 这个JSP页有两个角色（显示一个输入表单和显示处理该表单的结果），使它阅读起来混淆不清。
- 模板数据与JSP之间的参数转义是混淆不清的。虽然JSP创作工具将帮助防止不匹配的<% 和 %>，但没有任何办法使本页面中的Java控制结构（if, try/catch）与置标生成的组合变得更易读。如果没有Java知识（比如Java异常处理），要了解这个JSP页将产生内容是不可能的。
- JSP页与一些业务对象的EJB实现结合在一起。如果我们选择弃用EJB，将需要修改所有这样的页面，即使我们希望保持同样的表示。甚至连修改EJB的JNDI名也需要对所有这样的JSP页做修改。
- JSP页的结构没有体现它将生成的内容。在它的每个角色中，输出结果将是不同的。
- 异常处理是原始的。当我们正在生成动态内容的时候，我们可能会遇到错误，而处理它们太迟了。错误将会导致产生错误页面：由于处理不及时，以致不能对特定类型的错误执行适当的控制流程。
- 增加诸如用户输入是数值之类的检查并相应地警告用户将是困难的。页面将会变得更难读懂。
- 服务器小程序中有太多的Java代码，以致观察将被生成的置标是很困难的。
- 由于使用了像这样的JSP页，所以修改表示可能会破坏业务逻辑。

简单地说，表示与工作流程之间根本没有分离：同一个JSP处理所有事情，使得它对可能需要修改置标外观的HTML开发人员来说是难以理解的。

怎样在JSP页面中使用Java组件

在见过了这个有益的反面示例之后，让我们来看一看如何编写可维护的JSP页。

JSP页中显示的数据，通常来自用<jsp:useBean>标准动作在该页面上所声明的一个或多个Java组件。在MVC体系结构中，这些组件将在视图被调用之前已被设置为请求参数。在有些框架中，控制器将把模型数据设置为请求属性。在我们的框架中，View接口的一个实现

将在转发给一个JSP之前，把模型属性暴露为请求参数。在MVC Web应用中正确使用<jsp:useBean>标准动作的语法如下所示：

```
<jsp:useBean  
    id='performance'  
    type="com.wrox.exportj2ee.ticket.referencedata.Performance"  
    scope="request"  
/>
```

id属性的值是组件对象的名称，可供表达式、脚本小程序和动作使用。

请注意scope属性，它决定该组件值是来自本地JSP PageContext、HttpServletRequest、HttpSession（如果存在的话）还是全局ServletContext的一个属性值来设置。有4个范围值：page、request、session和application，这些值中只有一个值（request）与JSP在MVC Web应用中的正确使用相兼容，因为视图不应该访问（而且可能会处理）会话或应用级状态。

一个page范围的组件是一个本地对象，用在当前JSP内。有些情况下，这是合适的，尽管笔者更喜欢在JSP页中完全避免对象创建。

读者可能已注意到笔者省略了可选属性class，该属性指定全限定组件实现类，而不是指定相关类型；类型可能是接口，就像笔者已用过的type参数所指定的那样。通过只指定类型和不指定组件类，我们已经使JSP丧失了实例化一个新组件的能力，如果在指定范围内没有发现组件。在这种情况下，JSP将抛出一个java.lang.InstantiationException。

这是一种不错的做法，因为除非该组件已是可供JSP用做模型的一部分，否则JSP不应该得到调用：我们不希望它在自动创建了一个未配置的新对象取代缺少的模型对象之后神秘地失败。通过只指定类型，我们依然保持控制器中的灵活性（该控制器可以供应一个实现该接口的、任一类型的模型对象），而且把JSP对组件的访问限定在最受限制的相关接口上；另一种不错做法，如果涉及到接口继承性，或该对象具有一个不仅实现接口还暴露与相关视图没有关系的其他属性的类。

千万不要允许<jsp:useBean>动作创建一个对象，如果在class属性所指定的范围内未发现对象。JSP应该失败，如果所需的模型数据未得到供给。只应该和<jsp:useBean>动作一起使用request范围。这意味着当默认类型是page时，我们必须始终明确指定request范围。

JSP页不应该访问会话或应用对象，只应该访问由控制器暴露出来的模型数据。即使有些模型数据也被约束在用户的会话中，或者与其他用户共享（或许被约束在Servlet-Context中），控制器仍应该把这些模型数据作为模型的一部分传递给视图。使用这种方法能使我们使用任一视图技术，以及独立地测试视图。

正如笔者在第12章中所说的，JSP请求属性到组件映射机制太原始，以至于不能用在大多数实际情况中。该机制对优良设计来说也是一大威胁。令人遗憾的是，JSP规范中支持组件的方式存在混乱。组件实际上应该是用于页面的模型，但请求参数到组件属性的映射却意味着组件也可以用做帮助JSP页处理请求的一种方法。这是一种JSP Model 1方法，与MVC体系结构不相容。在起草JSP规范的第一个版本时，这种方法看起来像是合理的；后来，经验表明这种方法是有致命缺陷的。

JSP自定义标志

JSP 1.1中最显著的改进是引进了自定义标志（custom tag），也叫做标志扩展（tag extension）。这些改进极大地增强了JSP的威力，但容易使用不当。

自定义标志提供了如下好处。

- 它们提供一种把其他情况下将在JSP页中的代码转移到标志处理器类中的灵巧方法。例如，它们可以从使用了它们的JSP页中隐藏迭代模型数据的复杂性。
- 自定义标志是可移植的。它们以一种标准方式被打包，并支持重用。
- 自定义标志库可以用来解决许多常见问题，无论开放源中的，还是商业产品中的。但是，自定义标志也带来了一些危险。
- 自定义标志进一步增加了把JSP页用于非视图用途的诱惑力。比把代码从JSP页移到自定义标志好得多的做法，是设计Web层使JSP页不必含有代码。
- 基于XML的自定义标志语法未必总是适用。
- 从JSP 1.2起，自定义标志用Java来实现。虽然这在有些情况下是正确的，但在自定义标志完全是表示时会有问题，因为使用JSP的意义完全是为了避免需要让Java代码处理内容。
- 每个自定义标志都带有许多附属品。实现一个自定义标志需要编写至少一个Java类，以及一个XML标志库描述符。
- 自定义标志的大量使用会稍微降低JSP页的性能。不过，这在实践中通常是一个次要问题。
- 由于自定义标志可以语法分析它们的主体内容，而不是让JSP引擎计算它，所以自定义标志可能产生意料之外的行为（例如，一个自定义标志可以为一种可以用做其主体内容的特有语言提供一个翻译器）。这又一次举例说明了自定义标志可以实现在JSP内最好被避免的功能度。

已经证明自定义标志是非常流行的，而且它们现在是关于如何使用JSP的中心话题。

在了解自定义标志的具体使用之前，先让我们来看一看它们在前文中描述的MVC Web应用体系结构内扮演何种角色。

如果JSP页要用做视图，自定义标志就是视图助手（View helper）。它们既不是模型，也不是控制器；它们主要用来帮助显示JSP视图可以利用的类型模型。

我们不应该使用自定义标志去做超出视图角色之外的事情。例如，从数据库中检索数据在标志处理器中实现起来可能会很容易，但这不是JSP视图的责任。把代码移到标志处理器类中不会改善已基本坏掉的错误处理。一般说来，企业级访问（比如EJB访问和JNDI访问）不应该在标志中进行。由于这一原因，笔者认为提供简单数据库存取及类似功能的标志库是危险的，而且不应该用在设计完善的Web应用中。

使用标志来输出置标通常不是个好主意。有时，这是唯一可选的方法，但是为什么最好避免这么做有几个原因：

- 如果一个自定义标志生成置标，它就限定了它能被使用的情况。使用标志来控制页面的外观正是JSP页的责任。

- 标志处理器是Java构件。经验表明，自服务器小程序出现以来，从Java代码中生成置标是凌乱的。相反，JSP页是天生的模板构件，非常适合生成置标。
令人高兴的是，自定义标志可以定义脚本变量，允许JSP页处理输出格式化。

Java标准标志库

自笔者开始写本书以来，JSP可用性方面已经有了一个重要的里程碑：JSP Standard Tag Library (JSTL 1.0) 的发布。笔者把该发布看得比JSP 1.2的发布更重要，因为JSP 1.2只在JSP发展历程中前进了一小步。

JSTL提供了如下内容：

- Expression Language (EL)（表达式语言），它简化对Java组件属性的访问，并提供嵌套属性支持。这种语言可以用在传递给库标志的属性中。
- “通用动作”标志，它们可以输出表达式的值，创建或删除脚本变量，以及执行错误处理。
- “条件动作”标志，它们类似于Java的if和switch语句或XSLT的<xsl:if>和<xsl:choose>。
- “迭代器动作”标志，它们为迭代Java数组、Collection、Iterator、Enumeration、Map和权标（如在CSV串中）提供了一致的语法。
- “URL相关动作”标志，它们隐藏了创建URL时的URL编码和“URL重写”，并允许从当前Web应用内部或外部的资源中轻松地“导入”内容。
- “国际化动作”标志，它们允许轻松地访问适当地区的消息，这些消息被保持在标准Java资源包中。
- “格式化动作”标志，它们允许轻松地、串行地格式化数据和数字。
- “SQL动作”标志，它们允许从JSP页中轻松地执行RDBMS操作。
- XML标志，它们支持XPath表达式、XML语法分析和XML数据的XSLT转换。

当然，这些标志中没有一个是原来的。JSTL是一个法典，编纂了自JSP 1.1发布以后许多开发人员（包括笔者自己）所编写的这类标志的许多变种，但它是令人兴奋的，原因如下：

- 它是一个标准，因此有可能成为JSP开发人员当中的一种混合语言；
- 它是精致的，得到了好评的，完全集成的，而且基于广泛的经验；
- 它是一个相当小的标志集，没有令人混淆的重复；
- JSTL Expression Language要被并入到JSP 2.0中。

笔者见过并实现过许多标志库，笔者的感觉是JSTL已经获得了成功：这些标志简单而又强有力，并且使用起来直观。

JSTL标志库中惟一有疑问的一个库是SQL标志库。正如JSTL的设计师们所意识到的，这个库不应该用在设计完善的应用中。它的使用最好被限定在原型或次要应用中。

在此，笔者不打算逐个讨论JSTL标志，因为笔者将会在本章的稍后部分演示一些最重要的标志在示例应用代码中的使用。Jakarta实现所携带的示例是值得仔细研究的，因为该实现展示了最常见用法。

示例应用使用JSTL的Apache实现，读者可以从<http://jakarta.apache.org/taglibs/>站点上获得这个实现。笔者发现它非常便于使用，尽管抛出NullPointerException的倾向性有时会使调

试变得更困难。

再怎么强调JSP Standard Tag Library的重要性也不为过。这实际上给JSP增加了一种新语言——JSTL Expression Language。由于使用Java脚本的老式方法存在许多问题，JSP标准动作又不足以解决许多问题，因此这是一个“非常好的东西”。

如果读者使用了JSP，应该学习和使用JSTL。使用高级的JSTL Expression Language取代有局限的<jsp:useBean>标准动作。使用JSTL迭代取代for和while循环。不过，不要使用SQL动作：RDBMS存取在视图代码中没有什么用处，而且是业务对象（但不是Web层控制器）的责任。

其他第三方标志库

Jakarta TagLibs站点是其他开放源标志库的一个不错来源，也是需要访问的一个好去处（如果读者正考虑实现自己的标志库）。这里提供的许多标志库和别处的一样与JSP视图的责任相抵触，但它们对我们仍会有极大的价值。

特别有趣的是高速缓存标志；通过高速缓存JSP页的某些部分，这些标志可以用来改善性能。Jakarta TagLibs也有这样的标志，但正处于发展之中，而OpenSymphony (<http://www.opensymphony.com/>) 则有一个更成熟的实现。

自定义标志库的实现

虽然实现应用特有的标志库是相当容易的，但不要仓促行动。由于JSTL是现成的，所以许多应用或站点特有的标志不再是必需的。例如，JSTL Expression Language使许多被设计用来帮助显示复杂对象的应用特有标志变得多余（使用了Expression Language的JSTL标志可以轻松地显示复杂数据）。

如果读者正考虑实现自己的标志，笔者建议下列指导准则：

- 不要重复JSTL标志或已有第三方标志的功能度。一个应用使用特有标志越多，将来需要维护该应用的开发人员学习它们就会花费越长的时间。
- 设计标志使要尽力从内容开发人员而不是Java开发人员的角度出发。精巧的程序设计未必会产生好处；简单、直观的使用模式会产生好处。
- 把标志设计得与JSTL标志相互合作，因而需要认识到各种JSTL标志提供了许多常见问题的一种有效而又标准的解决方法。例如，不要编写条件标志；要保证你的标志暴露了能用于JSTL标志的脚本变量。示例应用的框架代码所携带的i21标志库清楚地说明了新标志怎么才能被设计得与JSTL相互合作。
- 使标志变得尽可能地可配置。这可以通过标志属性和使用嵌套提供上下文来达到。可选属性可以用在能够供给默认值的地方。
- 要使用JSP 1.2声明性语法来定义脚本变量，而不要使用JSP 1.1 TagExtraInfo类机制；这种机制比较复杂，而且很少是必需的。
- 避免在标志处理器中生成HTML，除非绝对必要。标志通常可以被设计成允许使用它们的JSP页去控制置标生成。
- 当标志处理器必须生成HTML时，要保证生成后的HTML能够用在多种上下文中；尽力避免生成<html>、<form>和其他结构性标志。要考虑从一个属性文件中读取

HTML。

- 要使用标志来声明脚本变量，以帮助JSP页格式化置标。
- 要避免让自定义标志对请求和响应做意想不到的事情。仅仅因为标志处理器能够通过PageContext来访问这些对象，并不意味着这就是一个好主意。

如果没有充分的理由，不要实现自己的JSP自定义标志。尤其是在拥有了JSTL的威力之后，许多应用特有的自定义标志不再是必需的。自己的JSP方言与标准JSP和JSTL的差别越大，维护自己的应用将会越困难。

使用自定义标志库的指导准则

下面是使用自定义标志、应用特有的和标准的标志的一些指导准则：

- 要使用JSTL标志库。它几乎始终提供脚本小程序的一种杰出替代方法。JSTL不久将被JSP开发人员普遍了解，因而保证用了它的页面是可维护的。
- 如果JSTL能够用来实现相同的结果，就不要使用专有标志库（比如Struts逻辑标志）或实现自己的标志。
- 不要使用标志处理器来做视图不应该做的事情，比如检索数据。标志处理器是视图助手，不是控制构件。
- 不要使用标志处理器来执行涉及深层嵌套的复杂迭代。这意味着创建一种非标准的XML脚本语言，而且有可能使JSP页很难以理解。笔者就曾经见过使用了深层嵌套自定义标志的JSP页比使用了普通脚本小程序的JSP页更难维护的情况。
- 如果没有充分的理由，不要实现自己的自定义标志。

自定义标志应该用做JSP视图外的视图逻辑再加工。它不应该隐藏不适合在视图中完成的那些任务的实现。

使用JSP的指导准则

最后，下面让我们来看一看使用JSP的一些总体建议。下面这些指导准则似乎过分严格。不过，如果读者在这些准则范围内仍无法看出一个JSP页如何工作，那么这个JSP页可能正在做不恰当的事情，并且需要被再加工。

- 要把JSP页完全作为视图来使用。如果一个JSP页做了另一种视图技术无法完成的事情（由于与Java的集成性或涉及到的复杂脚本化），那么该应用的设计有毛病。应该考虑把这个代码重新加工成一个控制器或业务对象。
- 要把条件逻辑放在控制器中，而不是放在JSP视图中。如果一个JSP页的某个实质部分根据某个条件相应地发生变化，那么这个JSP页是重新加工的一个候选对象，用控制器把请求转发给两个独立的页面，每个页面都含有那些公共部分。相反，如果一个JSP页只有一小部分是变化的，那么把条件逻辑放在一个JSP内则是正确的。
- 要考虑把JSP页放在/WEB-INF/目录下，以便它们无法被客户直接申请。在一个RequestDispatcher是否转发给/WEB-INF/目录下的资源方面，Servlet 2.3规范是含糊不清的。这在所有J2EE 1.3 Web容器中可能不工作，尽管这一疑惑在Servlet 2.4规范（第9节）中得到了澄清。

- 要使用下列指令关闭掉自动会话创建：

```
<%@ page session="false" %>
```

这将改善性能，而且有利于养成好的设计习惯。视图不应该被赋予访问会话数据的权限，因为它们可能会修改它，从而破坏好的设计。决定用户在一个HttpSession对象中是否应该拥有服务器端状态是控制器的责任，而不是视图的责任。

- JSP页不应该直接访问请求参数。控制器应该完成请求处理，并把处理后的模型数据暴露给JSP页和其他视图。访问请求参数需要仔细的错误检查；这种检查在JSP页中有可能被忽视，而且将会使这些JSP页变复杂，如果它没有被忽视的话。
- JSP页所使用的所有组件都应该具有request范围。
- 要避免使用JSP继承性机制。JSP页可以子类化一个实现了javax.servlet.jsp.HttpJspPage接口的自定义超类，而不是像默认地发生的那样，子类化容器所提供的实现。继承性机制是JSP基础结构的另一个强有力但有疑问的部分，而且对于这个基础结构，笔者还没有见过一个合法的应用。使用继承性也可能会降低性能，因为容器通常提供HttpJspPage接口的优化实现。
- JSP页不应该使用像JNDI或JDBC那样的企业级API。
- 不要使用请求参数到JSP组件属性的标准映射机制。它还没有强有力到足以解决现实问题的程度，而且还会破坏好的设计习惯。
- 要避免JSP页中的代码重复。需要代码重复是设计不良的一个信号。要把重复的功能度再加工成一个内藏的JSP页、组件或自定义标志，以便使它变得可重用。
- 要避免使用out.println()来生成页面内容。使用这个语法鼓励把JSP页看做程序，而不是看做Web页，而且可能会使编辑JSP页的HTML设计师产生混淆。
- 要通过适当的再加工和JSTL的使用，努力避开脚本小程序。存在无法避开脚本小程序的情形；接受这些情形，但只在考虑了所有真正的替代方法之后。
- 不要使用<jsp:forward>标准动作。这个JSP动作等效于可怕的goto语句，而且会使页面行为难以理解。转发违背了JSP页是视图的原则；如果JSP页不知道如何响应被传递给它的请求，那么这个系统的设计中存在错误。选择视图是控制器的责任，不是JSP视图的责任。
- 一般说来，声明和脚本小程序不应该用来创建整个JSP页内都参考的变量。JSP页里面的状态应该被保持在组件内。不过，自定义标志可以合法声明脚本变量。
- 不要使用自定义标志来允许JSP页做诸如数据库访问之类的事情，这类事情适合视图。
- 应该使用隐含注释来防止注释膨胀HTML输出结果。不仅要用文字说明动态内容的处理，而且还要说明复杂的置标。这可能会比使用得当的表达式和自定义标志更容易使阅读者产生混淆。
- 要在JSP页中避免异常处理。在MVC Web应用中，应该没有必要在JSP页中使用try/catch块，甚至使用标准错误页面机制，因为JSP页在运行时不应该遇到可恢复性错误。
- 不要在JSP页中替代jspInit()和jspDestroy()方法。使用这两个方法会使一个JSP页更像一个服务器小程序（一个程序），而不太像一个Web页。获得和放弃资源是控制器的

责任，不是JSP视图的责任，因此JSP页中应该没有做清理的必要。

- 不要在JSP页中定义方法、参数或内部类。这是另外一个最好避开的JSP能力。

前瞻：JSP 2.0的实现

本书谈论的是读者此刻可以使用什么来建立真正的应用。但是，当重大变化预先显露出来时，对设计策略会有影响。

JSP 2.0（撰写本书时正处于最后的建议草案阶段）将给JSP创作带来重大变化。尤其是，它将把JSTL中引进的Expression Language集成到JSP核心中，简化自定义标志的创作，并增加一个基于JSP碎片（JSP fragment）的较复杂包含机制。

JSP 2.0将引起JSP历史上最重大的变化，尽管它将是向下兼容的。这些变化中许多都趋向于格式化大多数经验丰富的Java Web开发人员已经采取的远离Java脚本小程序的行动。JSP 2.0将把JSTL Expression Language集成到JSP核心中，允许Java组件属性的更容易而又更复杂的导航。它还将允许更简单的自定义标志定义（不需要Java程序设计）。

用于示例应用的一个JSP视图

作为示例应用的一部分，我们所需要编写的惟一代码是暴露模型的JSP页。但是，我们首先需要创建一个视图定义，以便框架的视图分解器能够把“Show Reservation”视图名解析给我们的JSP视图。

Com.interface21.web.servlet.view.InternalResourceView类提供View接口的一个标准框架实现，并且这个实现可以用于包装JSP页或静态Web应用内容的视图。这个类（将在附录A中讨论）把控制器所返回的模型Map中的所有项目都暴露为请求属性，进而使目标JSP能够把它们用做具有request范围的组件。每种情况中的请求属性名和Map键相同。一旦所有属性都得到了设置，InternalResourceView类就使用Servlet API RequestDispatcher转发组件属性所指定的JSP页。

我们需要为InternalResourceView实现所设置的惟一组件属性是url：该JSP页在WAR内的URL。/WEB-INF/classes/views.properties文件中的下列两行定义我们用做示例的、名为showReservation的视图。

```
showReservation.class=com.interface21.web.servlet.view.InternalResourceView  
showReservation.url=/showReservation.jsp
```

这彻底分开了控制器与JSP视图；当选择“Show Reservation”视图时，TicketController类不知道该视图由showReservation.jsp、/WEB-INF/jsp/booking/reservation.jsp、一个XSLT格式表，还是由一个生成了PDF的类来再现。

现在，开始讨论实际的JSP代码，并看一看如何为示例应用实现一个JSP视图。这里不必编写另外的Java代码；框架已经考虑到了把请求路由给新的JSP页。

我们将用两步完成这项工作：首先实现一个使用脚本小程序来表示动态内容的JSP页；然后实现一个使用JSTL来简化事情的JSP页。这两个JSP页可以在示例应用的、名称分别为showReservationNoStl.jsp和showReservation.jsp的WAR中。

在每种情况中，首先关闭掉自动会话创建。我们不希望一个JSP视图创建一个会话（如果不存在会话）——会话管理是控制器的责任：

```
<% page session="false" %>
```

接着，使该页面的其余部分可以使用这3个模型组件：

```
<jsp:useBean id="performance"
    type="com.wrox.expertj2ee.ticket.referencedata.Performance"
    scope="request"
/>

<jsp:useBean id="priceband"
    type="com.wrox.expertj2ee.ticket.referencedata.PriceBand"
    scope="request"
/>

<jsp:useBean id="reservation"
    type="com.wrox.expertj2ee.ticket.boxoffice.Reservation"
    scope="request"
/>
```

需要注意的是，正如上面所建议的，笔者为每个组件指定了类型，而不是指定了类，而且每个组件被赋予了request范围。

下面来看一看如何使用脚本小程序输出数据，以演示将发生的各种问题。

第一个难题是格式化用户所在地区的日期和货币。如果简单地输出演出场次的when属性，将会得到默认的toString()值，这不是一个用户友好的值，因此需要使用一个脚本小程序，以一种恰当的格式来格式化演出场次的日期和时间部分。我们使用java.text.SimpleDateFormat类来应用一个在所有地区都可理解的模式，以保证月和日文本用用户的语言显示出来，并避免US和British日期格式化约定（分别为mm/dd/yy和dd/mm/yy）的问题。我们将把格式化后的文本保存在一个脚本变量中，所以稍后不必中断内容再现。

```
<%
    java.text.SimpleDateFormat df = new java.text.SimpleDateFormat();
    df.applyPattern("EEEE MMMM dd, yyyy");
    String formattedDate = df.format(performance.getWhen());
    df.applyPattern("h:mm a");
    String formattedTime = df.format(performance.getWhen());
%>
```

我们需要一个类似的脚本小程序来处理这两个货币值：总价格和订票费。这将使用java.text.NumberFormat类来获得并使用一个货币格式器，这个格式器将预先考虑适当的货币符号：

```
<%
    java.text.NumberFormat cf = java.text.NumberFormat.getCurrencyInstance();
    String formattedTotalPrice = cf.format(reservation.getTotalPrice());
    String formattedBookingFee =
        cf.format(reservation.getQuoteRequest().getBookingFee());
%>
```

现在就可以输出头部了：

```

<b><%=performance.getShow().getName()%>: <%=formattedDate%>
at
<%=formattedTime%>
</b>
<br>
<p>

<%= reservation.getSeats().length %> seats in
<%=priceband.getDescription()%>
have been reserved
for you for
<jsp:getProperty name="reservation"
                  property="minutesReservationWillBeValid" />
minutes to give you time to complete your purchase.

```

由于<jsp:getProperty>标准动作无法处理嵌套属性，所以笔者不得不使用一个表达式来获得节目的名称，这涉及到来自Performance组件的两个路径步骤。笔者还必须使用一个表达式（而不是<jsp:getProperty>动作）来获得PriceBand对象的description属性，这个属性是从SeatType接口中继承而来的。JBoss/Jetty 3.0.0所使用的Jasper JSP引擎在理解继承属性方面有问题——JSP的这个实现中的一个严重错误（有几个Web容器使用了这个实现）。

现在，需要对Reservation对象所暴露的座位数组中的Seat对象进行迭代。

```

The seat numbers are:
<ul>
<% for (int i = 0; i < reservation.getSeats().length; i++) { %>
    <li><%= reservation.getSeats()[i].getName()%>
<% } %>
</ul>

```

这是十分罗嗦的，因为在输出每个座位的名称时，需要在指定一个数组索引之前为每个座位都声明一个脚本变量或调用getSeats()方法。

由于我们已在处理日期格式化的脚本小程序中声明了formattedTotalPrice和formattedBookingFee变量，所以使用表达式输出这些信息是很容易的。

```

The total cost of these tickets will be <%=formattedTotalPrice%>
This includes a booking fee of
<%=formattedBookingFee%>.

```

如果Reservation对象指出分配后的座位不相邻，用一个简单的脚本小程序能显示允许用户选择另一场演出的链接。这对易读性来说没有多大问题，但我们确实需要记住关闭掉用于条件逻辑的复合语句。

```

<% if (!reservation.getSeatsAreAdjacent()) { %>
    <b>Please note that due to lack of availability, some of the
    seats offered are not adjacent.</b>
    <form method="GET" action="payment.html">
        <input type="submit" value="Try another date"></input>
    </form>
<% } %>

```

我们可以用节目ID计算出WAR内的座位安排计划图URL。

/static/seatingplans 目录中为每个座位安排计划都含有一个图，图的文件名具有 <seatingplanId>.jpg 格式。需要注意的是，在视图中构造这个 URL 是合法的；我们需要显示一个座位安排计划图或应该怎样定位这个图文件是与模型无关的。无论使用了何种视图技术，我们都将直接包括来自这个模型的静态内容。

在使用 HTML 标志以前，可以使用一个脚本小程序将这个 URL 保存在一个变量中。这个两步过程使得本 JSP 页更易懂：

```
<% String seatingPlanImage = 'static/seatingplans/' +
   performance.getShow().getSeatingPlanId() + '.jpg'; %>
<img src=<%=seatingPlanImage%> />
```

本 JSP 页的这个版本（使用了脚本小程序）不是灾难性的。MVC 模式的使用已经保证本 JSP 页只有表示上的责任，而且是十分简单的。不过，整个页面不是很精致，而且对于不了解 Java 的鼠标创作者来说维护起来不是很轻松。

现在来看一看如何使用 JSTL 来改善某些东西。这个页面的下列版本是示例应用中实际使用的版本。

我们的主要目标将是简化日期和货币格式化，以及解决座位数组的冗长迭代问题。我们还将会从 JSTL 的 Expression Language 中受益，以使属性导航在涉及到嵌套属性的地方变得更直观。

我们从关闭自动会话创建的同一条页面指令和 3 个相同的 <jsp:useBean> 动作开始。可是，我们需要在导入核心 JSTL 标志库和格式化库之后，才能使用它们，如下所示：

```
<%@ taglib prefix='c' uri='http://java.sun.com/jstl/core' %>
<%@ taglib prefix='fmt' uri='http://java.sun.com/jstl/fmt' %>
```

笔者将在事情有所不同的地方开始代码清单。

可以使用 <c:out> 标志和 JSTL 表达式语言来直观地输出嵌套属性的值。需要注意的是，由于不必使用脚本小程序，所以在访问属性时不需要 get 前缀或圆括号，从而使东西显得更简洁，尤其是在涉及到嵌套属性的地方。

```
<b><c:out value='${performance.show.name}' />
```

正如上述代码段所展现的，JSTL 表达式语言只能用在 JSTL 标志的属性里，而且表达式语言值像如下这样被括起来：\${expression}。特殊环绕字符的使用允许表达式用做一个属性值的一部分，而且这个值还含有字符串内容。JSTL 表达式语言除了支持嵌套属性导航之外，还支持算术与逻辑运算符。上述示例说明了嵌套属性导航是如何使用运算符来实现的。

日期格式化也相当漂亮。我们使用 java.text.SimpleDateFormat 中所定义的相同模式，但通过使用格式标志库，也可以在不用任何 Java 代码的情况下实现格式化：

```
<fmt:formatDate value='${performance.when}' type='date' pattern='EEEE MMMM dd,
yyyy' />
at
<fmt:formatDate value='${performance.when}' type='time' pattern='h:mm a' />
</b>
<br />
<p>
```

JSTL不允许通过表达式语言进行的方法调用，因此笔者必须使用一个普通JSP表达式来显示预订对象中的座位数组的长度（数组的长度不是组件属性）。

```
<%= reservation.getSeats().length %>
seats in
<c:out value="${priceband.description}" />
have been reserved
for you for
<c:out value="${reservation.minutesReservationWillBeValid}" />
minutes to give you time to complete your purchase.
```

使用JSTL迭代来迭代座位数组就没有那么罗嗦，因为这种迭代允许我们为每个数组元素创建一个变量，我们把这个变量称做了seat。借助于简单的嵌套属性导航，代码也得到了简化。

```
<c:forEach var="seat" items="${reservation.seats}" >
  <li><c:out value="${seat.name}" />
</c:forEach>
```

由于使用JSTL格式化标志库，货币格式化也相当漂亮。同样，没有必要转入到Java中，嵌套属性访问简单而又直观。

```
The total cost of these tickets will be
<fmt:formatNumber value='${reservation.totalPrice}' type='currency' />.
This includes a booking fee of
<fmt:formatNumber value='${reservation.quoteRequest.bookingFee}' type='currency' />.
```

如果座位不相邻，为了一致性，我们为货币使用JSTL核心库的条件标志（而不是一个脚本小程序）来显示要被显示的附加内容。对条件逻辑来说，使用置标取代脚本小程序未必就更直观，但同样，JSTL表达式语言也简化了嵌套属性访问。

```
<c:if test="${!reservation.seatsAreAdjacent}" >
  <b>Please note that due to lack of availability, some of the
seats offered are not adjacent.</b>
  <form method="GET" action="payment.html">
    <input type="submit" value="Try another date"></input>
  </form>
</c:if>
```

最后，我们使用一个嵌套属性表达式来查找座位安排计划图的URL。需要注意的是，只有value属性的一部分才被计算为一个表达式——其余部分是字符串内容。

```
<img src=
  "<c:out value='static/seatingplans/${performance.show.seatingPlanId}.jpg'" />
```

这个页面的两个版本使用了名义上相同的技术（JSP），但它们是非常不同的。第二个版本维护起来比较简单和容易。本例既说明了JSTL是JSP开发人员武器库的一个重要部分，又展示了JSTL完全改变了JSP页显示数据的方式。这个页面的JSP 2.0版本将比首先给出的标准JSP 1.2版本更接近于第二个版本。

JSP小结

JSP是J2EE规范中定义的新技术。它提供了优良的性能和一个相当直观的脚本模板。不过，使用JSP必须小心。除非应用了严格的编程标准，否则J2EE的威力会证明它对可维护性是一个严重威胁。

JSP Standard Template Library（JSP标准模板库）凭借它的简单而又直观的表达式语言，以及通过支持许多常见需求（借助于自定义标志），可以使JSP页变得更容易、更精致。

不要只使用标准JSP。要把JSP和JSTL结合起来使用。正如我们的示例所展示的，使用JSTL可以极大地简化JSP页，并帮助我们避免需要使用脚本小程序。

专用模板语言

如果仅把JSP页用于视图，我们正在使用JSP页作为模型。如果以JSTL为基础使用JSP，我们正在放弃使用Java作为脚本语言的原JSP概念。

由于我们不希望使用JSP页的请求处理能力或JSP页所提供的Java脚本化威力，并且认为这些特性的可利用程度目前可能会造成不良后果，所以理所当然地怀疑我们目前为什么还应该完全使用JSP。

目前，在开放源领域内，有许多模板语言都可以用来替代JSP（XSLT就值得占用一节的篇幅，在稍后加以讨论）。和JSP不同，这些语言是完全被作为模板语言来设计的。每种语言都有各自的优缺点，但它们之间有很多相似之处。首先来做个全面的了解，然后看一看使用一个模板解决方案（而不是JSP）的得与失。

常见概念

本节中将要讨论的模板化解决方案共用下列基本概念：

- 一个模板是一个纯粹的视图。它只能含有简单的条件逻辑与迭代，不应该提供一种真正的程序设计语言所具有的威力。
- 由于这一原因，模板语言学习起来简单、轻松。
- 控制器必须使数据可供模板使用。和JSP页不同，模板自身没有办法获得数据，例如从请求对象或用户会话中获得数据。使模型可供模板使用的确切方式随着产品的不同而有所不同。
- 使用一个模板包括一个两步过程（该过程需要使用一个模板管理器特有的API）：从文件系统或类路径中获得一个模板实例，以及使用给定模型数据和模板来生成输出结果并把结果写到响应上。

我们为什么可能为Web层视图选择使用一种模板语言来代替JSP呢？原因如下：

- 使用模板语言有利于优良的设计。正如我们已经见过的，JSP使得使用一个不充分的“Model 1”体系结构或破坏MVC体系结构变得很容易和有诱惑力。通过使用模板语言，视图的角色变得很明确。
- 在匹配模板页的外观与置标的外观方面，大多数模板语言比JSP更胜任。这使得页面设计师的生活变得更轻松。

- 模板语言比JSP提供了更简单的语法。JSP脚本小程序语法令人眼花缭乱，而JSTL和其他自定义标志需要使用置标，使用置标对Java开发人员或置标创作者来说不是处理控制流程的最自然方式。
- 大多数模板语言不束缚于Servlet API。这意味着可以在其他场合使用它们（例如，生成静态页面）以充分发挥采用它们时所投入的资金，而且模板开发与测试可以在不运行J2EE服务器的情况下完成。

但是，使用专用的模板语言也有缺点。

- 与JSP和JSTL表达式语言相反，无任何一种模板语言是得到公认的标准。既然大多数模板语言都非常简单，又没有太大的学习难度，这就不应该像它所显露的那样是一块主要的绊脚石。此外，JSP页经常由不太完全了解Java和JSP语法的置标开发人员来编辑。然而，政治上的现实是：在许多项目中，由于存在一种“标准”替代技术，要想让人们接受J2EE标准之外的技术简直是不可能的，无论有没有应该首选这些技术的基础。

只要选择了正确的模板化技术，就不太会有束缚于一种没有生存前途的模板语言的危险（支持“标准”解决方案的经典理由）；像Velocity那样的主导模板语言具有生存前途并被普遍接受也没有多大疑问。

- 我们需要把运行时库作为应用的一部分来分布。JSP保证在任何兼容于J2EE的Web容器中都是可利用的，尽管分布JSTL的二进制文件目前还是必不可少的。不过，WAR给我们提供了一种包含库类的标准方法，因此这不是一个主要问题。

虽然JSP页（被编译成脚本小程序）预计可能会好于模板化解决方案，但在实践中，结果可能正好相反。我们将在第15章中讨论视图性能。

下面来看一些流行的模板化解决方案。

WebMacro

WebMacro是基于Java的最古老而又最有名的模板语言之一。

下列摘录出自WebMacro资料 (<http://germ.semiotek.com/README.html>)，并得到作者许可；它重点描述了WebMacro与JSP之间的差别，并提供了使用专用模板化解决方案代替JSP的案例。

WebMacro在许多方面类似于JSP/ASP，在许多方面又类似于其他项目，但根据我们的偏爱又不同它们：

- 我们认为把置标用于一种脚本语言是不恰当的；
- 我们认为把程序嵌在Web页面上是不恰当的；
- 我们认为Web脚本看起来像困难的程序设计是不恰当的；
- 我们觉得一个像WM那样的API应该使你能够轻松地完成你在设计和部署应用时需要完成的事情；
- 我们认为程序设计和图形页面设计是彼此独立的任务。

WebMacro使用一种不需要与脚本之间进行转义的简单语法，就像JSP页所做的那样。由于不是基于置标的，所以该语法比使用JSP自定义标志更直观。下列示例说明了怎么才能将WebMacro用于一条基于Java组件属性的条件语句。需要注意的是，这个组件脚本变量有

一个前缀\$:

```
#if ($bean.GoldService) {  
    <font size='4'>Welcome to our Gold Service!</font>  
} #else  
    <font size='4'>Welcome!</font>  
}
```

通过被放在一个WebMacro上下文（WebMacro context）中，Java组件可以让WebMacro模板使用（WebMacro上下文含有其用法与HttpServletRequest属性在JSP页中的用法相类似的对象）。

和JSTL一样，WebMacro支持嵌套属性路径，比如person.apouse.name。也可能创建新变量、执行迭代、做计算。

虽然这种语法稍有不同，但概念实际上和Velocity的概念是相同的（下文讨论），因此笔者在此不打算讨论WebMacro模板语言。进一步信息，请参考<http://webmacro.org/WebMacroBasics>。

WebMacro已在实际的Web站点上得到证明。自2001年1月以来，AltaVista（万维网上点击率最高的站点之一）一直在使用WebMacro作为它的主要生成技术。WebMacro可以从<http://webmacro.org>站点上获取，并且在Gnu GPL协议下是免费的。

示例应用所携带的框架代码包含了View接口的一个实现com.interface21.web.servlet.view.webmacro.WebMacroView，这个实现允许第12章中所讨论的MVC框架和WebMacro模板一起使用。如果使用这个类（不要忘了把WebMacro JAR文件包含在WAR的WEB-INF/lib目录中），把WebMacro模板和这个框架一起使用就不必修改任何控制器或模型代码。这个视图实现和下文将要讨论的Velocity实现十分相似，因此笔者在这里不再讨论它。

Velocity

Jakarta Velocity实质上是WebMacro的一个再实现，只是具有一个截然不同的代码库和一些新的特性。要想了解关于Velocity与WebMacro之关系的讨论，请参见<http://jakarta.apache.org/velocity/differences.html>站点。Velocity比WebMacro拥有更全面的资料，出版在一个不同的许可下，而且Velocity项目似乎更活跃。

本节将介绍Velocity的基本概念，并描述示例视图怎么才能被一个Velocity模板再现出来。

Velocity概念

Velocity纯粹而且仅仅是一个模板引擎。这使得Velocity学习起来很轻松，而且避免了一个框架的常见问题：对于其他产品已经充分处理的概念，该框架非要提供它自己的实现。Velocity模板引擎不依赖于Servlet API，意味着它可以用于除Web应用之外的其他应用，并且Velocity模板可以在服务器小程序容器外部被测试（优于JSP的一个重要优点）。

Velocity的核心是Velocity Template Language (VTL) —— 等效于WebMacro模板语言。

VTL看起来非常像WebMacro模板语言。没有必要转义表达式或“脚本小程序”。#字符用来前缀指令（控制语句），\$字符用来前缀变量。如果这两个字符是静态模板数据的一部分，它们都可以被转义。

Velocity具有一个稍微不同于WebMacro的原理，其目的在于使核心指令集保持最小，同时又允许用户通过定义Velocimacro（Velocity宏）来为他们的应用扩展Velocity的能力；Velocity宏是指能够像Velocity指令一样使用的可重用模板段。宏可以定义调用时要传递的任意个变元。下面这个宏示例（仅带有一个变元）输出一个用HTML格式化过的、从1到一个给定最大值的平方表，并演示了基本的Velocity语法：

```
#macro( squareList $upTo )
<ul>
#foreach ( $item in [1..$upTo] )
#set ($square = $item * $item)
<li>$square
#end
</ul>
#end
```

这个宏可以在一个Velocity模板内被调用，如下所示：

```
#squareList(6)
```

Velocity宏可以在Velocity模板中被“内嵌地”声明（尽管这种声明可以根据需要被禁用），或被收集在模板库（Template library）中，类似于JSP标志库。从某些方面来看，Velocity宏就是JSP自定义标志应该已拥有的东西：简单而且无需用Java进行程序设计就能定义（JSP 2.0将引进更接近于Velocity宏的、更简单的标志扩展定义）。

和WebMacro中一样，Velocity模板中所使用的模型对象必须能让Velocity上下文对象中的Velocity使用。模板可以轻松地为放在Velocity上下文中的对象，暴露包括嵌套属性在内的组件属性。

和JSP页不同但和WebMacro模板相同的是，Velocity模板并不使用一种类似于<jsp:useBean>的机制来声明它们的上下文中所需要的模型对象。上下文中无法解析的变量参考将显示为文字，使得纠正打字错误变得更容易（例如，如果没有叫做的nothing的模型对象，那么\$nothing将表现为模板输出中的一个串文字）。变量声明的缺少和打字检查对Java开发人员来说可能是令人不愉快的，但Web层表示模板不是为Java开发人员编辑而打算的。

Velocity带有一个JSP自定义标志，这个标志把它的主体内容翻译为VTL。当使用Velocity时，很少有理由使用JSP作为一个包装器，而且混用JSP和VTL会致使JSP的所有消极面沉渣泛起，再加上技术的混用会使维护变得更为困难。然而，有几种情况下Velocity标志库可能是有用的：

- 在整个页面必须由JSP来生成的地方。
- 在需要使用Velocity没有提供的、而JSP却拥有并且又是一个有效视图操作的某一特性的地方。例如，如果读者需要使用某一个特殊的标志库。

在此，笔者不打算详细描述VTL语法。Velocity发行包携带了一个不错的简明参考指南。做为补偿，让我们来看一看如何使用Velocity模板显示我们的样本视图。

用于示例的Velocity模板

我们的MVC Web框架提供对Velocity视图的标准支持。在WEB-INF/classes/views.properties中，通过使用com.interface21.web.servlet.view.velocity.VelocityView标准View实现，可以为样本页面定义一个Velocity视图组件。这个标准View实现在启动时将获得并高速缓存相关的Velocity模板。当被要求再现一个模型给响应时，它创建一个Velocity上下文，其中含有对应于模型中每一项的属性。这类似于InternalResourceView把模型项复制给请求属性的方式。然后，在已知模型和上下文数据之后，这个标准View实现使用Velocity把输出结果写给响应。

关于如何实现这个视图以及如何安装并配置Velocity的详细描述，请参见附录A。

在VelocityView类上要设置的最重要的组件属性是templateName属性，该属性标识本WAR内的一个Velocity模板。我们可以按照如下所示来定义样本视图：

```
showReservation.class=com.interface21.web.servlet.view.velocity.VelocityView
showReservation.templateName=showReservation.vm
showReservation.exposeDateFormatter=true
showReservation.exposeCurrencyFormatter=true
```

需要注意的是，这个组件定义把两个格式助手属性都设置成了真(true)，这会引起VelocityView类给Velocity上下文添加具有名称simpleDateFormat和currencyFormat的DataFormat和NumberFormat助手，然后模板可以使用它们来格式化日期和货币量。这个功能度（由VelocityView框架类提供）是必不可少的，因为Velocity 1.3（令人惊讶）根本就没有提供对日期和时间格式化的支持（该问题将在附录A中做进一步讨论）。

编写该Velocity模型（可在WEB-INF/classes/showReservation.vm处的样本WAR中找到）的第一项任务是设置由日期助手所使用的模式，并创建含有格式化后的日期和时间的变量。由于我们依靠这方面的Java支持，所以将要使用的代码类似于我们的JSP示例中所使用的代码：

```
<!--
$simpleDateFormat.applyPattern('EEE MMM dd, yyyy')
#set ($formattedDate = $simpleDateFormat.format($performance.when) )
$simpleDateFormat.applyPattern("h:mm a")
#set ($formattedTime = $simpleDateFormat.format($performance.when) )
-->
```

需要注意的是，笔者已把Velocity“脚本小程序”放在了一个HTML注释中。这个脚本小程序实际上不生成输出，但这么做也避免它扰乱HTML编辑工具。#set指令允许我们声明稍后要用在模板中的变量。

一旦这些不再是障碍，模板的其余部分编写起来就很容易，而且很直观。请注意对嵌套属性的简单访问。我们的模型对象一旦被放到Velocity上下文中，就能被按名访问。

```
<html>
<head>
<title>Seats reserved for $performance.show.name</title>
</head>
```

现在可以输出格式化后的日期和时间。

```
<body>
<b>$performance.show.name:
$formattedDate at $formattedTime
</b>
<br/>
<p>
```

尽管Velocity允许我们调用方法，以及获得属性值（通过使用圆括号，即使方法不带有变元），但不允许我们查找一个数组的长度。幸运的是，座位数量也可以从Reservation对象的quoteReservation属性中获得，因此我们可以采取一条迂回路线。

```
$reservation.quoteRequest.seatsRequested
seats in
$priceband.description
have been reserved
for you for
$reservation.minutesReservationWillBeValid
minutes to give you time to complete your purchase
```

Velocity最闪光的地方是迭代座位数组。和JSTL相同（但和JSP脚本小程序不同）的是，Velocity依次把每一项暴露为一个变量，使这个代码变得非常简单。Velocity的语法比基于置标的JSTL语法更简单，更直观。

```
The seat numbers are:
<ul>
#foreach ($seat in $reservation.seats)
    <li>$seat.name
#end
</ul>
```

Velocity也允许我们通过velocityCount预定义变量的值来寻找这个列表中的位置，尽管本例中不需要这个能力。

可以使用NumberFormat助手来格式化货币量，意味着不必提供货币符号。NumberFormat类将始终适合请求地区的货币符号。

```
The total cost of these tickets will be
$currencyFormat.format($reservation.totalPrice).
This includes a booking fee of
$currencyFormat.format($reservation.quoteRequest.bookingFee).
```

条件判断（比如我们关于座位是否相邻的检查）使用Velocity if指令。请注意结束用的#end指令。

```
#if (!$reservation.seatsAreAdjacent)
    <b>Please note that due to lack of availability, some of the
    seats offered are not adjacent.</b>
    <form method="GET" action="payment.html">
        <input type="submit" value="Try another date"></input>
    </form>
#end
```

座位安排计划图URL的显示涉及到另一个嵌套属性。请注意Velocity资料称为形式表示法 (formal notion, 用花括号括起来的变量名) 的使用方法, 这种表示法允许我们把字符串中是一个变量的部分与文字值文本区分开, 区分所使用的方式和我们使用JSTL \${}表达式定界符的方式相同。

```

```

通过比较这种表示法与JSTL方法, 可以看出JSTL与Velocity中访问组件属性之间的相似之处, 以及基于置标的脚本设计的各种缺点。JSTL版本包含嵌套的和<c:out>标志, 而输出只包含我们正在设法实际生成的标志。

```
<img  
    src=<c:out  
        value='static/seatingplans/${performance.show.seatingPlanId}.jpg' />  
/>
```

请注意这两种方法是怎样不像标准JSP方法那么罗嗦和易出错的。JSP方法使用困难, 即便是在被分解成一个脚本小程序和一个表达式的时候, 也是如此。

```
<% String seatingPlanImage = "static/seatingplans/" +  
    performance.getShow().getSeatingPlanId() + ".jpg"; %>  

```

Velocity小结

Velocity提供了一种高性能的、十分简单的、非常适合暴露Java组件属性的模板语言。Velocity不是Web特有的。

Velocity中的属性导航语法与JSTL中的差不多。简单的Velocity脚本设计语言使用特殊字符来标识指令, 而不是像JSP脚本小程序中那样使用与嵌入代码之间的转义, 或像JSP自定义置标中那样使用置标。因此, 本节示例的Velocity模板比使用了JSTL的JSP版本更简明、更容易读。

虽然J2EE标准支持JSP的理由很充分, 但笔者发现, 通过比较Velocity或WebMacro模板与表示相同信息的JSP页可以看出, JSP语法是多么罗嗦和不直观, 而且自定义标志 (和JSP 2.0的期望) 只是缩小了这一间隙。虽然Java开发人员往往不怀疑JSP页的易读性, 但根据笔者的经验, 不得不与它们打交道的HTML开发人员发现它们难于使用, 而且很怪异。

Velocity和WebMacro都是实施干净分离控制逻辑与视图模板的有效而又实用的解决方案。虽然笔者选择了使用JSP作为示例应用的视图技术, 但正如上述示例所显示的, 使用Velocity可能会产生更简单、更易读的视图模板。

FreeMarker

FreeMarker在概念上类似于WebMacro和Velocity, 因为通过使用供给的API来获得模板, 并用该模板来再现通过使用方法调用所构造的模型。FreeMarker模板语言拥有一种比WebMacro和Velocity更面向置标的语法, 但在威力上是类似的。

但是和WebMacro、Velocity及JSP不同，FreeMarker视图避免使用反射来获得模型值，大概是因为担心反射的性能。这意味着适配器必须被编写为应用代码的一部分，才能把数据暴露给FreeMarker（这些适配器类似于Swing模型）。虽然可能不必编写附加的Java代码就能把任何一个Java组件添加到一个WebMacro或Velocity上下文中并访问它的属性和方法，但FreeMarker需要明确编程。

WebMacro或Velocity上下文的FreeMarker等效物是一个“数据模型树”，而且在这个树中，树根必须实现freemarker.template.TemplateModelRoot接口，并且每个子节点必须实现freemarker.template.TemplateModel的另一个子接口。这就是说，当暴露一个对象图表，将需要为每个应用类都编写一个适配器类。通过使用freemarker.template.SimpleScalar便利类，可以较轻松地暴露串数据。随同FreeMarker一起提供的来宾册示例应用就展示了适配器在每个应用类中的使用。在这个次要的应用中，适配器类中有114行代码。

笔者还见过反射证明是这类情形中的一个性能问题，所以笔者认为，给应用开发人员强加这样的额外工作来避免在框架中使用反射是一种极差的设计折中方法。不过，不是人人都赞成这一观点：请参见<http://www.javaworld.com/jw-01-2001/jw-0119-freemarker.html>站点，这里就有一篇文章称赞FreeMarker的设计。

FreeMarker也是开放源的，并且是按照Gnu GPL协议来发布的。读者可以从<http://freemarker.sourceforge.net/>站点上获得它。

XSLT

XSLT（XML转换语言）比JSP更标准。和JSP不同，XSLT纯粹是为变换数据到另一种格式而设计的（例如，为了把数据显示给用户），不是为获取数据而准备的。因此，在Web应用中，XSLT格式表变换用于表示的内容，但不能用来处理用户请求。对于某些任务来说，JSP比其他任何一种可替代的视图技术更能胜任。它保证了数据与表示之间的一流分离。

但是，在我们一下子做出XSLT是再现Web内容的最终方法这一结论之前，必须先来考虑几个明显的消极因素。XSLT对外行人来说是复杂的。与我们已讨论过的其他再现技术相比，XSL变换速度较慢（有时非常慢）。第15章将讨论一些衡量性能问题的测试基准。在可以变换Java对象之前，我们可能需要先把它们转换到XML，因而引起额外的复杂性和性能开销。不过，通过只使用XSLT的简化应用代码的特性，也许能最大限度地减少这些缺点。

下一节不是XSLT或XML的入门读物。如果读者对这里所讨论的某些概念还不十分熟悉，请参考一本论述这方面内容的参考书，比如由Michael Kay所著、Wrox出版公司出版的优秀图书“XSLT Programmer's Reference”（ISBN 1-8-6100506-7）。

何时使用XSLT

从技术上和实践中看，采用XSLT作为视图技术有如下好处：

- XSLT自身就是一个标准。XSLT技能与J2EE或Java无关。例如，大量的Microsoft开发人员就广泛使用XML和XSLT。这意味着我们或许可以把具有本行业或站点类型经验的XSLT创作人员吸收进来，而不是把我们眼光只盯在JSP专家身上。XSLT很可能会成为Web创作的一个平台交叉标准。我们或许还可以发挥现有XSLT格式表的作用。

- XSLT比其他任何一种再现树结构内容的表示解决方案更优秀。它甚至可以用来执行与参考数据的SQL连接相等效的操作。
- XSLT具有杰出的分类能力。
- 由于XSLT格式表是XML文档，所以使用XSLT可以帮助保证生成的置标是构成完善的。例如，交叉标志在格式表中是非法的。这将帮助避免许多常见的浏览器错误。
- 使用XSLT要求我们使数据可以用做XML。这可能会获得超出Web表示之外的好处。
- XSLT不是Web特有的，只要它不是仅用在诸如JSP之类的其他视图技术中。
- 和JSP不同（但和我们已考虑过的其他所有模板化解决方案相同）的是，XSLT不是Web特有的。在开发XSLT技能中的投入可以在许多方面产生好处。

对照这些优点，我们需要权衡下列缺点：

- XSLT有一种复杂的语法，因为XSLT格式表是XML文档。正如我们讨论JSP自定义标志时已经见过的，置标对一种脚本设计语言来说不是一种理想的语法。虽然一旦理解了它，做复杂的任务就会很轻松，但对页面设计师（甚至对许多Java开发人员）来说，XSLT似乎没有VTL甚至JSP那样的简单模板语言直观。工具支持仍是有限的。
- XSLT性能尽管还没有差到使它不能用于最性能敏感型的应用，但也不太可能接近于JSP或更简单的模板化解决方案的性能。
- 即使有了好的XSLT创作工具（一种必需之物，因为XSLT格式表看上去与它们所生成的输出相差甚远），一个项目仍将需要至少一名对XSLT十分精通的开发人员。例如，格式表性能的变化范围将非常大，视格式表被如何构造而定。
- XSLT与J2EE集成得没有JSP那么平滑。不过，正如我们将要看到的，可以使用JSP自定义标志和一个MVC框架来提供必需的基础结构。
- 在可以使用XSLT来显示它们所含有的数据之前，通常需要执行额外的步骤把Java对象转换成XML元素。

与选择JSP还是Velocity相比，选择使用XSLT则是一个更具策略性的决定，前者实质上是选择如何显示相同对象的Java组件属性。XSLT模型明显不同于其他大多数表示数据的解决方案，采用XSLT可能会需要大量的培训。

如果数据已经以XML形式存在，使用XSLT几乎始终是最佳方法。但是，从基于Java的面向对象应用中获得XML未必就那么容易。第6章已经讨论过“Domification”的概念：飞行地把Java组件转换成XML文档。下列示例将使用Domify开放源包。虽然反射的使用将导致少量的性能开销，但不要忘了，像Velocity甚至JSP那样的其他模板技术也频繁利用反射。

希望从XSL中得到什么

XSLT建立在几种可拆分开的技术之上，记住这一点十分重要。XPath表达式语言（在XSLT格式表中用做一种子语言）对XSLT从XML模型数据节点内选择内容的强大能力来说十分关键。

即使没有采用XSLT作为我们的主要视图技术，我们仍可以享有XPath的各种好处。使用Domify或另一个库（比如Apache Commons JXPath，可从<http://jakarta.apache.org/commons/jxpath/>站点上获得），我们可以把XPath表达式应用于已经以XML形式存在的数据之

外，甚至还可以把XPath表达式应用于Java组件模型数据。另外，还可以选择仅使用XPath和XSLT来生成某一视图的一部分：例如，JSTL提供了允许XPath表达式的计算和允许我们把XSLT处理嵌入到JSP页中的标准标志。

怎样在视图中使用XSLT

假设我们已经有了XML形式的数据，那么怎样完成将这些数据再现在视图自身内的任务呢？

使用XSLT取代JSP

一种“纯”方法是利用XSLT来完成全部的再现和内容生成。纯XSLT方法的优点如下：

- 通过把全部模型数据转换到XML，我们已经创建了一个平台独立的表示层。
- 我们知道生成后的内容将始终是构成完备的。不匹配标志是HTML标志中的一个常见错误源，而且确信从JSP页中生成的内容在所有执行路径下都构成完备是不可能的。
- 我们在单一的视图技术上标准化。

纯XSLT方法有如下这些缺点：

- 我们被迫把所有模型数据都转换成XML。有些情况下这可能是很困难的。
- 导入数据的XSLT机制相当复杂，需要依靠使用了XSLT元素<xsl:import>和<xsl:include>的规则导入。不过，<xsl:import>机制是十分强有力的。包含进来的文件必须是构成完备的XML文档。相反，JSP和其他模板语言提供了任意内容的轻松包含。
- 我们可能要忍受由使用XSLT带来的最大性能命中率，返回给客户的每个数据字节都是一个XSL变换的结果。

从JSP自定义标志中使用XSLT

通过把XSLT类操作和XPath表达式嵌入到其他模板中，可以获得XSLT和XPath的部分好处。JSTL使这种做法变得特别容易，所以我们将把讨论的重点放在如何从JSP页中使用XSLT上。

这种方法的优点是：我们可以“挑选”何时使用XSLT。例如，如果某些模型数据是可以作为XML使用的，那么使用XPath和XSLT暴露它们会比使用Java脚本小程序更容易。这种方法的缺点是：

- 保证生成的置标构成完备是不可能的。
- 我们没有站点的一个完整XML描述。这也是采用XML的另一种重要好处；我们没有为特定格式表选择XML，但由于暴露XML允许我们使用任一格式表，所以我们可以暴露XML或我们所希望的其他基于文本的格式。
- 组合技术可能会增加复杂性。

下面来看一看如何使用JSTL来暴露可用做请求属性的模型对象，该模型对象已经是XML形式，作为org.w3c.dom.Node。

首先，必须从JSTL中导入XML标志库：

```
<%@ taglib uri="/x" prefix="x" %>
```

这个库提供了允许我们基于XPath表达式做迭代、计算和执行条件逻辑的XSLT类标志。例如，假设请求属性node是一个XML节点，那么我们可以像下面这样迭代子元素：

```
<x:forEach select="$node/reservation/seats/item">
    <x:out select="name"/>
</x:forEach>
```

自由路径的第一个元素(\$node)才是JSTL特有的，它使用JSTL表达式语言来标识一个含有XML数据的对象。该表达式的其余部分是一个与这个节点相关的XPath，所以这个标志以与<xsl:for-each>标志相同的方式对待它的内容。这是一个没有太大价值的例子；我们可以在这样的表达式中使用XPath的全部威力。

当模型数据以XML形式存在时，这是一种简单而又强有力的方法。但是，JSTL没有提供把Java组件数据暴露为XML的支持。在这种情况下，我们可以使用一个简单的结束用标志来“Domify”Java组件数据。下列示例使用了一个可在示例应用所携带的框架代码中使用的标志：该标志使用Domify库来暴露一个对象属性（通过一个JSP表达式来获得），并使它可用做一个名为node的脚本变量（关于这个标志的实现，请参见com.interface21.web.tags.DomifyTag的源代码）。这使得JSTL XML标志的任何一个都能在它的范围内使用。下列代码段可以用在上述标志库导入的后面，以便在showReservation.jsp视图上显示Reservation模型对象中的座位列表，无需做其他任何修改。

```
<%@ taglib uri="/x" prefix="x" %>
<%@ taglib uri="/i21" prefix="i21" %>
<i21:domify
    root='reservation' model="<%=$reservatuon%>" >
    <ul>
        <x:forEach select="$node/reservation/seats/item">
            <li><x:out select="name"/>
        </x:forEach>
    </ul>
</i21:domify>
```

这对示例应用的“Show Reservation”视图来说是简直是没有必要的：使用JSTL<forEach>标志执行这样的简单迭代更容易。不过，如果我们需要像XPath节点选择的全部威力之类的东西，这将是另外一回事。

JSTL还提供了允许使用XSLT格式表来生成一个JSP页局部的标志；同样，当数据以XML形式存在时，或当它的显示需要XSLT的威力时，这是一个很有用的功能度。

JSTL还提供了允许语法分析XML主体内容供其他XSLT标志使用的标志。这些标志应该谨慎使用：获取内容是Web层控制器和业务对象的责任，不是JSP视图的责任。出现语法分析错误的可能性在视图中也是有存在的。

JSTL引进了在JSP视图内使用XPath和XSLT的强大而又标准的支持。这通常是一个好的折衷方案，使我们能够在凡是XSLT适用的地方使用它的威力，而使用JSP作为我们的主要视图技术。

需要注意的是，JSP不是我们使用嵌入式XSLT的惟一选择（尽管在使用了JSTL时，这是一个非常不错的选择）：像Velocity那样的其他视图技术可能也允许我们在较大的组合视图内使用XPath和XSLT。

使用“纯”XSLT方法来实现示例。

由于J2EE与XML和XSLT集成得不如JSP那么平滑，所以最大的挑战是使用一种“纯”XML方法（在这种方法中，XSLT格式表生成整个视图）。幸运的是，我们的MVC框架提供了必要的基础结构代码。View接口的com.interface21.web.servlet.view.xslt.XsltView通用实现，在初始化时编译并高速缓存一个XSLT格式表。当它被要求把一个模型再现给响应对象时，它自动使用Domify包创建一个XML节点来代表该模型的数据。然后，它利用TrAX API并使用已得到缓存的格式表和模型的XML数据来执行一个XSLT变换，并把输出写给响应对象。

这个类的实现和本例中所使用的扩展函数将在附录A中讨论。

我们在应用中需要做的只是在WEB-INF/classes/views.properties中声明一个XsltView型的视图组件。我们必须始终设置组件属性来指定文档根（代表模型根元素的标志名称）和模板URL（格式表在WAR内的位置）。我们还可以任选地将excludedProperties属性设置成我们希望从飞行变换中排除掉的全限定属性的一个CSV串。这可以简化生成后的节点。在目前情况下，防止循环参考是必不可少的。示例视图的定义如下所示：

```
showReservation.class=com.interface21.web.servlet.view.xslt.XsltView
showReservation.root=reservationInfo
showReservation.stylesheet=/xsl/showReservation.xsl
showReservation.excludedProperties=
    com.wrox.expertj2ee.ticket.referencedata.support.ShowImpl.performances
```

需要注意的是，如果没有指定格式表URL，输出将会显示输入的XML文档。这在开发期间是一个非常有用的功能度。XsltView类还给模型添加一个名为<request-info>的特殊元素，该元素含有XsltView类从请求中抽取出的地区信息。这可以在XSLT格式表内支持国际化。

我们的框架也在com.interface21.web.servlet.view.xslt.FormatHelper类中提供了简化日期和货币的XSLT扩展函数。和Velocity一样，XSLT在这些方面目前缺少很好的支持。我们将在自己的格式表中使用这些扩展函数。

既然知道了必要的基础结构代码如何工作，以及如何配置它，现在该来看一看为了使用一个XSLT格式表来暴露“Show Reservation”屏幕，作为示例应用的一部分，我们需要开发些什么。

由于数据没有以XML形式存在，所以需要依靠Domify包。因此，非常有必要了解Domify从我们的模型中所生成的XML文档将会具有什么样的结构；这个XML文档将含有3模型对象（Reservation、PriceBand和Performance）和我们在XsltView中所添加的RequestInfo对象。需要注意的是，这个文档中的节点只在请求时才被生成；虽然这个DOM文档在我们编写XSLT格式表时是一个不可或缺的参考，但它没有给出该应用在运行时需要做什么的一个精确反映（只要我们的格式表是有效的，而且没有强制执行文档节点的穷举搜索）。

下面是生成的文档的完整的代码清单：

```
<?xml version="1.0" encoding="UTF-8"?>
```

根元素对应于我们的模型映像：

```
<reservationInfo>
```

对于每个模型对象，我们将会获得一个与之对应的<item>元素，其中“key”属性是该对象在模型中的名称（这是Domify在暴露java.util.Map对象时的默认行为）。需要注意的是，简单组件属性被暴露为相关<item>元素的子元素，其中它们的文本内容是它们的值的一个串表示。

```
<item key="priceband">
  type="com.wrox.expertj2ee.ticket.referencedata.support.PriceBandImpl">
  <name>AA</name>
  <id>1</id>
  <description>Premium Reserve</description>
  <price>102.5</price>
  <seatTypeId>1</seatTypeId>
</item>
```

模型对象将被Domify暴露的顺序没有任何保证。在这种情况下，上述4个模型对象的第二个是XsltView类为了暴露地区信息而添加的<request-info>对象。注意，这个对象含有标准语言和国家代码。

```
<item key="request-info">
  type="com.interface21.web.servlet.view.RequestInfo">
  <language>en</language>
  <country>GB</country>
</item>
```

其余元素暴露Reservation和Performance对象。请注意，Reservation对象含有嵌套对象，这些嵌套对象具有它们自己的组件属性。它们被暴露在一个分级的自然XML结构中。

```
<item key="reservation">
  type="com.wrox.expertj2ee.ticket.boxoffice.support.ReservationImpl">
  <timestamp>1027419612736</timestamp>
  <reference>AA</reference>
  <totalPrice>10.0</totalPrice>
  <quoteRequest>
    <performanceID>1</performanceID>
    <seatsRequested>2</seatsRequested>
    <seatsMustBeAdjacent>false</seatsMustBeAdjacent>
    <bookingFee>3.5</bookingFee>
    <holdTill>
      <date>23</date>
      <time>1027419912736</time>
      <year>102</year>
      <month>6</month>
      <hours>11</hours>
      <minutes>25</minutes>
      <seconds>12</seconds>
      <day>2</day>
      <timezoneOffset>-60</timezoneOffset>
    </holdTill>
    <classID>1</classID>
```

```

<reserve>true</reserve>
</quoteRequest>
<seats>
  <item type="com.wrox.expertj2ee.ticket.boxoffice.Seat">
    <name>A1</name>
    <id>1</id>
    <seatClassId>1</seatClassId>
    <left>0</left>
    <right>0</right>
  </item>
  <item type="com.wrox.expertj2ee.ticket.boxoffice.Seat">
    <name>A2</name>
    <id>2</id>
    <seatClassId>1</seatClassId>
    <left>0</left>
    <right>0</right>
  </item>
</seats>
<seatsAreAdjacent>true</seatsAreAdjacent>
<reserved>true</reserved>
<minutesReservationWillBeValid>4</minutesReservationWillBeValid>
</item>
<item key="performance"
      type="com.wrox.expertj2ee.ticket.referencedata.support.PerformanceImpl">
  <name>1</name>
  <id>1</id>
  <show>
    <name>Romeo and Juliet</name>
    <id>1</id>
    <seatingPlanId>1</seatingPlanId>
    <seatTypes/>
  </show>
  <when>
    <date>23</date>
    <time>1027419612746</time>
    <year>102</year>
    <month>6</month>
    <hours>11</hours>
    <minutes>20</minutes>
    <seconds>12</seconds>
    <day>2</day>
    <timezoneOffset>-60</timezoneOffset>
  </when>
  <priceStructureId>1</priceStructureId>
</item>
</reservationInfo>

```

下面来看一看用来显示这些信息的格式表。

XSLT创作模型与JSP和模板引擎（比如Velocity）的创作模型也是十分不同的。通常，XSLT模板不是顺序执行，而是基于被应用到XML输入中的元素上的规则。

首先，为FormatHelper类中的各种扩展函数声明一个名称空间。

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:format="com.interface21.web.servlet.view.xslt.FormatHelper"
>
```

由于我们将访问“request info”项目中的地区代码，所以声明一个全局变量来缩短访问路径，或许还改善性能：

```
<!-- Make this available globally -->
<xsl:variable name="reqInfo" select="/reservationInfo/item[@key='request-info']"/>
```

在这样的一个简单格式表中，我们可以使用许多种方法来显示内容。XSLT专家Michael Kay在他的著作“XSLT Programmer's Reference”中区分了4种常见类型的XSLT格式表。本例中，我们可以在他所称的“Fill-in-the-blanks（填空）”格式表和“Rule-based（基于规则）”格式表之间进行选择。

“Fill-in-the-blanks”格式表看起来像生成后的内容，使用XPath表达式和迭代来选择动态内容。这样的格式表沿着页面由上向下地执行，并使用一个类似于JSP或Velocity模板的结构。“Rule-based”格式表定义用于内容元素的规则，所以不反映输出文档的结构。这种格式乍看起来不太直观，但会证明是非常有威力的。

笔者选择了为示例使用一个简单的“Rule-based”格式表，因为它比“Fill-in-the-blanks”格式表更精确一些，并且演示了经典的XSLT模型。

首先为根元素定义一个规则。这将使该规则应用于XML输入内其key属性值为“performance”的所有`<item>`元素。在当前示例中，我们知道只有一个这样的`<item>`元素。需要注意的是，这实际上是在元素被遇到时禁止规则的自动应用。由于我们不知道输入元素的顺序，所以这在当前示例中不是非常有用，但在其他情形中会非常有威力。

```
<xsl:template match="/">
  <xsl:apply-templates select="item[@key='performance']"/>
<xsl:template>
```

对于其key值为“performance”的`<item>`元素，模板可以把这样的元素设想为它的上下文节点（Context node）。这意味着它能够使用与这样的元素相关的XPath。在下列代码段中，笔者已经对`dateTimeElement()`扩展函数的使用做了突出处理：

```
<xsl:template match="item[@key='performance']">
  <html>
    <head>
      <title>Seats reserved for <xsl:value-of select="show/name"/></title>
    </head>

    <body>
      <b><xsl:value-of select="show/name" />:</b>
      <xsl:apply-templates select="format:dateTimeElement(when/time,
        $reqInfo/language, $reqInfo/country)"/>
      </b>
      <br/>
      <br/>
```

这里惟一棘手的事情是我们怎样处理关于座位分配和座位价格的信息。这些信息来自`Reservation`对象，尽管该内容的其余部分来自`Performance`和`PriceBand`元素。我们通过调用`ReservationInfo`对象数据的相应模板来处理这些信息，提供一个来自文档根的，而不是来自上下文节点的XPath。

```
<xsl:apply-templates select="/reservationInfo/item[@key='reservation']" />

<form method="GET" action="payment.html">
  <input type="submit" value="Proceed"></input>
</form>
```

```

Please check the seating plan below for the location of the reserved seats.
<br/>
<br/>
<xsl:variable name="seatingPlanImage">static/seatingplans/<xsl:value-of
select="show/seatingPlanId"/>.jpg</xsl:variable>


</body>
</html>
</xsl:template> .

```

预订信息的模板十分直截了当，除了需要显示PriceBand对象的描述之外（从根元素中获得，即突出显示的部分）：

```

<xsl:template match="item[@key='reservation']">

<xsl:value-of select="quoteRequest/seatsRequested" />
seats in
<xsl:value-of select="/reservationInfo/item[@key='priceband']/description"
/>
have been reserved
for you for
<xsl:value-of select="minutesReservationWillBeValid" />
minutes to give you time to complete your purchase.
The seat numbers are:

```

XSLT轻松地迭代预留座位数组：

```

<ul>
<xsl:for-each select="seats/item">
    <li><xsl:value-of select="name" />
</xsl:for-each>
</ul>

<p>
The total cost of these tickets will be
<xsl:value-of select="format:currency(totalPrice, $reqInfo/language,
$reqInfo/country)" />.
This includes a booking fee of
<xsl:value-of select="format:currency(quoteRequest/bookingFee,
$reqInfo/language, $reqInfo/country)" />.

```

XSLT条件语法很像JSTL的语法：

```

<xsl:if test="not (seatsAreAdjacent='true')" >
    <b>Please note that due to lack of availability, some of the
    seats offered are not adjacent.</b>
    <form method="GET" action="payment.html">
        <input type="submit" value="Try another date" />
    </form>
</xsl:if>
</xsl:template>

```

最后，我们需要一个规则来显示由dateTimeElement()扩展函数所创建的<formatted-date>元素：

```

<xsl:template match="formatted-date">
    <xsl:value-of select="day-of-week"/><xsl:text> </xsl:text>
    <xsl:value-of select="month"/><xsl:text> </xsl:text>

```

```
<xsl:value-of select="day-of-month"/>,
<xsl:value-of select="year"/>
at
<xsl:value-of select="hours"/>;<xsl:value-of select="minutes"/>
<xsl:text> </xsl:text>
<xsl:value-of select="am-pm"/>
</xsl:template>

</xsl:stylesheet>
```

令人高兴的是，如果这个格式表计算成功，我们就知道了生成后的输出是构成完备的。

没有必要修改控制器代码，而且再现给浏览器的输出结果看起来将和JSP和Velocity所生成的输出结果完全一样。

只要服务器上提供了XML文档和格式表这两样东西，在客户浏览器中进行XSLT变换也是可能的。不过，这种方法（具有把处理负荷从服务器转移到客户上的可能性）在现实生活中是有问题的，因为不是所有的浏览器都提供了标准或可靠的XSLT支持。在我们的MVC框架中，即使不修改控制器或模型代码，一个特殊的View实现也可以增加这种XSLT支持。

可以论证的是，就我们的简单视图来说，它的XSLT格式表比我们迄今为止所见过的那些方法更复杂、更难理解——尽管在掌握了XSLT的情况下该格式表是简单的。证明使用XSLT作为示例应用的视图技术正是示例应用的业务需求所要求的将是一件困难的事情。

不过，XSLT是一种非常强大的语言，而且正是因为能够经常轻松地处理较复杂的表示要求而获得了人们的赞赏，例如，如果示例应用的欢迎页面需要显示艺术流派、节目和演出场次的一个树结构（而不是像目前这样只显示艺术流派和节目），并且分类或筛选又是必不可少的，那么XSLT是一个非常好的选择，而且所需的XSLT格式表几乎肯定比生成相同输出结果的一个JSP页更简单。

幸运的是，我们不必非要依靠“纯”XML方法才能享有XSLT的各种好处；我们可以利用JSTL把XPath表达式和XSLT格式表嵌入在JSP页内。

XSLT和XPath最好是在数据已经以XML形式存在的时候使用，但正如本节的示例中所显示的，把Java组件模型转换到XML也是相当容易的。

置标生成的替代方法

我们迄今为止已考虑的各种方法都是模板化解决方案，在这些解决方案中，一种模板语言再现出Java视图适配器使之变得可供它使用的内容。这并不是用于视图的惟一有效方法。现在让我们来看几种使用Java代码生成视图的方法。在我们的框架中，这意味着针对将实际生成内容的每个页面，都将有com.interface21.web.servlet.View接口的一个不同实现。

HTML生成库

一种方法是使用笔者称做HTML生成库（HTML generation library）的一种东西——允

许我们使用Java代码来构造HTML文档的一个Java类集。在这个模型中，不是使用一个像Velocity模板或JSP页那样的视图模板，而是在把生成的置标写给响应对象之前，先使用Java代码把一个HTML文档构造为一个由表示文本的对象、格式化、表单字段等所组成的对象。HTML生成库保证生成的置标是构成完备的，而且隐藏最终输出的复杂性。这个概念十分古老，而且最早的实现先于JSP，尽管没有一个十分流行。

大多数生成库都是HTML特有的。但是，有些概念可以在不同的输出格式之间共享。例如，iText库（下文讨论）就允许HTML与PDF生成之间的一些通用性。

对象生成库有如下这些优点：

- 它是一种比使用JSP页或模板语言更面向对象的方法。
- 它完成任务可能很出色。
- 它可以帮助避免像打字错误之类的问题，因为复杂的输出是生成的，而不是由开发人员编码的。
- 它可以支持多种输出格式，把它们的细节隐藏在一个抽象层的后面。不过，根据笔者的经验，这个优点的理论意义大于现实意义，因为不同的输出格式共享概念的程度还不足以使这样的通用化值得花时间去做。

对象输出库的缺点如下：

- 谁修改表示？每个修改都需要Java开发人员吗？如果始终都需要Java资源，这种修改将是一个严重的问题吗，即使Java开发人员无需处理细微的置标细节？
- 有问题的创作模型。HTML实物模型（常常在新接口的设计阶段制作）怎么才能被转变成生成它的Java代码？有些解决方案（比如XMLEC）解决了这个问题。
- 许多生成库与HTML结合得太紧密。例如，如果我们需要生成WML，怎么办？如果我们正在使用的生成库不支持WML，这将是一个致命的问题；即使它支持WML，这也可能要求重大的代码修改。
- 我们正在把HTML的中心问题引入到我们的Java代码中：HTML不明确区分内容与表示。
- HTML是一种人类易读的格式。如果我们需要生成PDF之类的复杂格式，使用生成库就没有意义。不过，就HTML而论，点缀了几条动态语句的置标（就像在编写完善的JSP页和模板中那样）将会比调用了类库的Java代码更容易阅读。

或许，最完善的对象生成库是BEA的htmlKona。关于其能力的描述，请参见http://www.weblogic.com/docs51/classdocs/API_html.html。这是一个封闭源的专有产品，与WebLogic服务器捆绑在一起。在WebLogic版本6中，htmlKona遭到轻视，赞成JSP——HTML生成库没有证明是一个巨大成功的象征。Jetty服务器小程序引擎也包含了一个可单独使用的HTML生成库。请参见<http://jetty.mortbay.com/jetty/index.html#HTML>。

稍后，我们将看一个如何使用iText来生成PDF的例子。这个示例演示了类似于HTML生成库的创作模型。

笔者不喜欢HTML生成库，也无法想出我将选择使用它们的理由。不过，它们的使用与好的MVC设计是一致的，并且像其他视图技术一样能被我们的Web应用框架支持。

XMLC

要解决HTML生成库所具有的最严重问题：Java开发人员进行表示方面的修改的需要以及在把(X)HTML实物模型转变成生成它的Java代码方面的困难，惟一方法是自动生成要创建和操纵预定义输出格式的Java代码。

完成这个目标的一种技术是XMLC。它最初由Lutris开发作为Enhydra应用服务器的一部分，但现在正处于开放源之中。XMLC是一种与我们迄今为止已见过的方法都有很大差别的视图生成方法。XMLC保留了HTML生成库的部分优点，同时又几乎消除了用Java代码创建固定模板结构的基本问题。

XMLC用置标来驱动生成方法。页面设计师首先用静态HTML创建一个实物模型站点。XMLC将“编译”那些仿制的HTML页，产生Java源代码和(或)类。这些类(即使用了W3C DOM的“XMLC对象”)用Java描述HTML内容，并在该内容被输出之前允许程序性地处理HTML。

如果来自静态实物模型的内容位于具有id属性的一个标志内，它可以被修改。例如，一个页标题可以被赋予一个id属性，以便允许它被修改。当需要修改置标的较大片段时，可以引进一个HTML 或<div>标志来围住该片段(这些标准的结构性标志被定义在HTML 4.0规范中)。该XMLC对象和它所表示的HTML内容可以通过标准DOM接口(允许元素的检索和修改)或使用XMLC所提供的便利方法来操纵。

因此，Java开发人员用动态内容填写那些空白，以及添加或删除静态模板上的内容。如果那些空白不再改动，设计师和Java开发人员就可以独立地进行下一步工作，修订后的实物模型导致新Java对象的生成。HTML实物模型实际上是一个约定，定义了视图的动态内容。

因此在设计时，XMLC方法包括以下步骤：

1. 创建HTML内容，里面含有代表动态数据的占位符。
2. 使用XMLC把HTML内容“编译”成XMLC对象(Java类)。
3. 在输出之前编写Java代码来处理XMLC对象。

在运行时，XMLC方法包括以下步骤：

1. 构造代表本视图的“XMLC对象”的一个新实例。
2. 处理该对象的状态，比如通过增加元素、输出元素或设置动态元素内容。
3. 使用XMLC把该对象的最终状态输出给HttpServletResponse对象。

XMLC提供了如下优点：

- XMLC除了擅长生成HTML之外，还擅长生成XML。
- XMLC不束缚于Servlet API。
- 建立原型很容易。模板是纯粹的HTML；它们不含有使浏览器混淆不清的东西。

XMLC允许设计师插入原型数据，并且该数据可以从编译后的类中被排除掉。这就是说，与使用了迄今为止已讨论过的其他视图技术的模板相比，XMLC模板看上去常常较复杂——当在浏览器中被查看的时候。例如，表可以包括多行在运行时将被真实数据所取代的模型数据。这是一个特有的特性，此特性使得XMLC在某些情况下非常有吸引力。但是，存在两个问题：

- 依据条件逻辑显示的输出数据是有问题的。我们可以在生成的XMLC对象中增加或删除内容，但内容需要在某个地方被放样。通常的解决方法是让置标模板包括全部有条件地输出的数据。在运行期间，数据将在那些条件得到计算时被删除，这不会引起技术上的问题，但却意味着对设计师和企业用户来说，模板看上去可能没有意义。
- 静态包含在设计时不显示出来，除非那些模板被放在了一个理解服务器端包含的Web服务器上。
- XMLC将比其他技术更适合处理由许多流行Web创作工具所生成的理解起来困难的HTML。但是，设计师必须能够给动态内容赋予id属性，而且能够在必要时创建和

元素。XMLC是我们已见过的、置标是不是能被人看懂都无关紧要的惟一技术。

XMLC的缺点包括：

- 使用DOM API来处理由XMLC生成的类非常麻烦。不过，XMLC生成额外的便利方法来访问元素对解决该问题有点用处。
- XMLC模型与众不同，而且不共享其他视图机制所共享的许多概念。这可能会使企业在引进它变得困难，因为它需要企业采用大家可能都不熟悉的一个模型。
- 由页面设计师所维护的HTML模板实际上握有Java代码的关键。对它们的任何改动都需要相关XMLC对象的生成。通常，这没有问题，但如果粗心的改动删除或破坏了XMLC所需要的id，那么相关的XMLC对象将不再工作。因此，在维护纯HTML模板时需要小心。

用于示例的一个XMLC模板

这个模板是标准的HTML，不含有任何XMLC特有的标志。事实上，笔者首先从保存Velocity视图的动态输出开始处理这个模板。

惟一特殊的要求是具有动态内容的元素必须被赋予一个id属性，如果必要，就使用一个结束用的或

标志。需要注意的是，当该模板在浏览器中查看时，插入到该模板中的那些值将是可见的。

标题被赋予一个id属性，使我们能把演出场次名称附加到它的模板文本的初始内容上。在这里，我们还可以使用java.text.MessageFormat：

```
<html>
<head>
<title id="title">Seats reserved for </title>
</head>
```

对于演出的名称、日期和时间，我们必须引进元素，因为没有我们可以处理的置标元素，比如开始代码段中的元素：

```
<body>
<b><span id="performanceName">Hamlet</span>:
<span id="date">January 1, 1983</span>
at
<span id="time">7:30 pm</span>
```

请注意，该模板中的元素的内容将充当占位符，以便使该模板在浏览器中显示有意义。我们给预订信息采用相同的方法：

```
<span id="seatsRequested">2</span>
seats in
<span id="seatType">Seat Type</span>
have been reserved
for you for
<span id="minutesHeld">5</span>
minutes to give you time to complete your purchase.
```

显示座位id列表包括创建一个模板列表id（- 标志可以被赋予一个id属性，所以我们不需要一个或

标志来结束它）。需要注意的是，笔者还增加了两行模型数据，它们用一个“mockup”类来标识。这些元素将帮助使该模板的外观在浏览器中看起来更逼真，但可以由XMLC在该XMLC对象生成之前删除：

```
The seat numbers are:
<ul>
  <li id="seat">Z1</li>
  <li class="mockup">Z2</li>
  <li class="mockup">Z3</li>
</ul>
```

显示价格信息只包括使用标志来标识可能动态的内容。请注意，我们不必修改这些值：在某些情况下，我们或许对默认值感到满意，只偶尔修改它们：

```
The total cost of these tickets will be
<span id="totalPrice">totalPrice</span>.
This includes a booking fee of
<span id="bookingFee">bookingFee</span>.
```

由于在预订的座位万一不相邻时才显示的内容包含了一个嵌套的元素，所以我们必须使用一个

元素，而不是一个元素来结束它：

```
<div id="nonAdjacentWarning">
<b>Please note that due to lack of availability, some of the
seats offered are not adjacent.</b>
<form method="GET" action="otherDate.html">
  <input type="submit" value="Try another date">
</form>
</div>
```

剩下的惟一动态数据是座位安排图的URL。我们给该元素赋予一个id属性以便能处理它，并给它赋予一个将使该模板能在浏览器中显示的相对URL：

```

```

这肯定分离开了模板与Java代码。整个过程的最得意之作是HTML模板看起来和动态生成的页面一模一样，而与我们已经见过的模板都不一样。座位名称列表被填上虚构项，而其他动态内容含有占位符值。惟一障碍是不相邻座位警告的包含，该警告在大多数页面上不会出现（我们必须在运行时有计划地删去不想要的分支）。图13.7所示的屏幕图在浏览器中显示了模板。

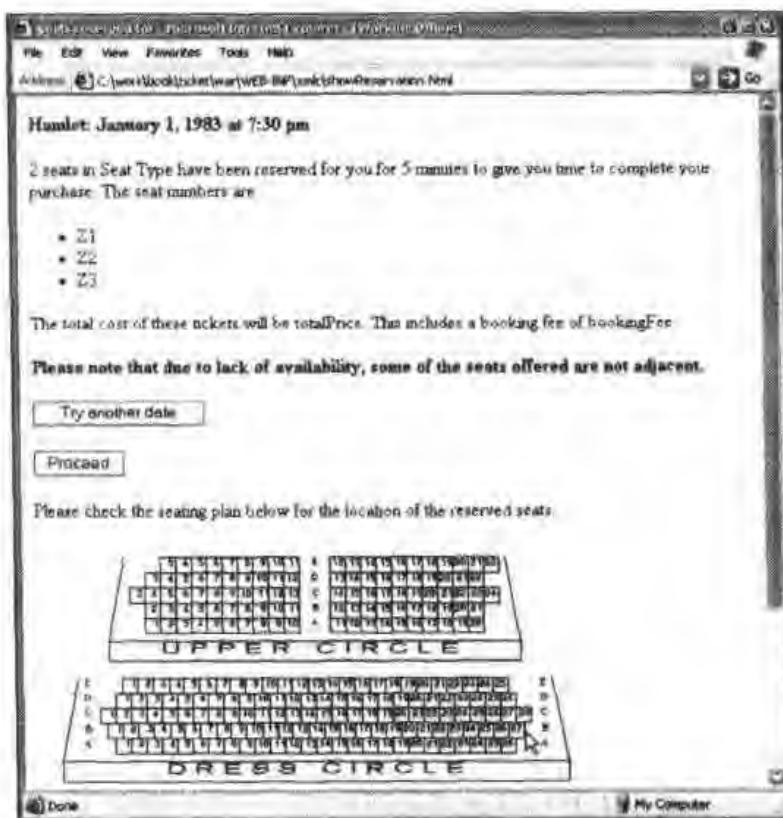


图13.7

模板的编译

在编写Java代码来实现“Show Reservation”视图之前，需要生成一个XMLC对象。可以运行XMLC所携带的xmlc命令，但笔者选择了使用Ant来使该任务可重复。由于示例应用中只有一个HTML文件使用XMLC来编译，所以笔者硬编码了它的名称。不过，使该Ant目标更完善是很容易的：

```

<target name="xmlc">
  <java
    classname="org.enhydra.xml.xmlc.commands.xmlc.XMLC"
    fork="yes">
    <classpath>
      <fileset dir="${lib.dir}">
        <include name="runtime/xmlc/*.jar"/>
      </fileset>
    </classpath>

    <arg value="-keep"/>
    <arg value="-nocompile"/>
    <arg value="-dump"/>
    <arg value="-ssi"/>
    <arg line="-sourceout src"/>
    <arg line="-delete-class mockup"/>
    <arg line="-class
      com.wrox专家组.j2ee.ticket.web.xmlc.generated.ShowReservationXmlcObject"/>
    <arg value="war/WEB-INF/xmlc/showReservation.html"/>
  </java>
</target>
```

最有趣的内容是在指示XMLC的标记符中：

- -keep标记符告诉XMLC，不要删除Java源代码（它默认成只保留一个.class文件）。
- -nocompile标记符告诉XMLC，不要编译生成的源文件。我们选择使这个源文件成为我们的连编过程的一部分，而不是XMLC编译过程的一部分。
- -dump标记符告诉XMLC，显示由它对HTML输入所做的分析所揭示出来的结构。如果XMLC没有生成我们所期望的那些便利设置器方法，该显示是非常有用的。
- -ssi标记符告诉XMLC处理服务器端包含（它不默认地处理这些包含）。
- -sourceout标记符告诉XMLC把生成的Java类放在何处。我们选择了把它和我们自己的类一起放在/src目录中。
- -delete-class标记符告诉XMLC删除具有mockup类的元素，这将删除我们包含在模板中的虚拟列表数据，以便当该模板在浏览器中被查看时显示得更真实。
- -class标记符为Java类指定一个全限定名（默认设置是在默认包中生成一个和模板同名的类）。
- 最后一个值是到达该模板的路径。

在编译完毕时，我们应该有一个名为com.wrox.expertj2ee.ticket.web.xmlc.generated.ShowReservationXmlcObject的类，这个类就是该模板HTML的Java表示。我们不打算编辑这个类，但将会使用它的方法来处理它的状态。如果生成的XMLC对象和普通源代码一起被放在一个IDE能够找到它们的地方，这个IDE就应该能够提供关于它们的方法的上下文帮助，而且这个帮助很可能会证明是非常有用的。

处理从模板中生成的XMLC对象

现在来看一看如何为XMLC实现com.interface21.web.servlet.View接口。

在XMLC对象中，和代码生成库的情形一样，我们为每个视图都需要一个不同的Java对象。在运行时，HTML模板不再是必需的，但我们为XMLC所生成的每个视图都需要一个视图实现。

我们不必从头开始。我们的MVC框架为XMLC视图提供了一个便利超类——com.interface21.web.servlet.view.xmlc.AbstractXmlcView，这个超类使用Template Method设计模式从子类中隐藏掉必要的管道工程，并只给它们留下操纵相关XMLC对象的任务。子类只需要实现下列保护方法即可：

```
protected abstract XMLObject createXMLObject(
    Map model,
    HttpServletRequest request,
    HttpServletResponse response,
    XMLCContext context)
    throws ServletException;
```

和大多数XMLC视图子类一样，我们用在示例应用中的具体实现不暴露组件属性。因此，/WEB-INF/classes/views.properties中的整个组件定义如下：

```
showReservation.class=
    com.wrox.expertj2ee.ticket.web.xmlcviews.ShowreservationView
```

关于安装和配置XMLC的进一步信息和com.interface21.web.servlet.view.xmlc.AbstractXmlcView框架类的描述，请参见附录A。

现在来看一看我们的具体示例com.wrox.expertj2ee.ticket.web.xmlcviews.ShowReservationView。首先从扩展AbstractXmlcView开始：

```
public class ShowReservationView extends AbstractXmlcView {
```

没有组件属性，而且不必提供一个构造器。我们实现必需的保护抽象方法，如下所示：

```
protected XMLObject createXMLObject(
    Map model,
    HttpServletRequest request,
    HttpServletResponse response,
    XMLCContext context)
    throws ServletException {
```

首先构造一个新的XMLC对象。我们可以利用一个无变元构造器来构造该XMLC对象，但使用XMLC上下文变元来提供一个对象会更有效：

```
ShowReservationXmlcObject showReservationXmlcObject =
(ShowReservationXmlcObject) context.getXMLCFactory().create(
    ShowReservationXmlcObject.class);
```

现在，从映像中抽取我们的模型对象，所以仅执行一次类型强制：

```
Reservation reservation = (Reservation)
    model.get(TicketController.RESERVATION_KEY);
Performance performance = (Performance)
    model.get(TicketController.PERFORMANCE_KEY);
PriceBand priceBand = (PriceBand)
    model.get(TicketController.PRICE_BAND_KEY);
```

接着，使用我们在前面已见过的类似代码，以便利用标准Java国际化支持并基于请求地区来格式化日期和货币量：

```
SimpleDateFormat df = (SimpleDateFormat)
    DateFormat.getDateInstance(DateFormat.SHORT, request.getLocale());
df.applyPattern("EEEE MMMM dd, yyyy");
String formattedDate = df.format(performance.getWhen());
df.applyPattern("h:mm a");
String formattedTime = df.format(performance.getWhen());
NumberFormat cf = NumberFormat.getCurrencyInstance(request.getLocale());
String formattedTotalPrice = cf.format(reservation.getTotalPrice());
String formattedBookingFee =
    cf.format(reservation.getQuoteRequest().getBookingFee());
```

现在，可以开始修改含有id属性的元素中的动态内容。XMLC使我们省去了使用DOM做这件事的麻烦，因为它生成便利的处理方法。设置动态文本是非常容易的：

```
showReservationXmlcObject.setTextTitle(
    showReservationXmlcObject.getElementTitle().getText() + " " +
    performance.getShow().getName());
showReservationXmlcObject.setTextPerformanceName(
    performance.getShow().getName());
showReservationXmlcObject.setTextSeatsRequested(" " +
    reservation.getSeats().length);
```

```

showReservationXmlcObject.setTextSeatType(priceBand.getDescription());
showReservationXmlcObject.setTextMinutesHeld("+" +
    reservation.getMinutesReservationWillBeValid());
showReservationXmlcObject.setTextDate(formattedDate);
showReservationXmlcObject.setTextTime(formattedTime);
showReservationXmlcObject.setTextTotalPrice(formattedTotalPrice);
showReservationXmlcObject.setTextBookingFee(formattedBookingFee);

```

为了建立座位名称列表，我们需要获得原型列表项，为模型中的每行数据克隆它，然后删除原型行本身。这是一种和我们已见过的方法都有极大差别的方法。它不是很困难，但也不特别直观：

```

// Get template node for seat list
HTML.Element seatEle = showReservationXmlcObject.getElementSeat();
for (int i = 0; i < reservation.getSeats().length; i++) {
    showReservationXmlcObject.setTextSeat(
        reservation.getSeats()[i].getName());
    seatEle.getParentNode().insertBefore(seatEle.cloneNode(true), seatEle);
}
// Remove template node
seatEle.getParentNode().removeChild(seatEle);

```

不过，需要记住的是，XMLC创作过程的这一部分由Java开发人员来完成，而不是置标开发人员（他们已经提供了实物模型HTML）。因此，采取必要的程序设计技能是安全可靠的。需要注意的是，那些“实物模型”列表数据元素在XMLC从HTML模板中生成一个Java对象时被忽略。

我们通过删除这个元素（如果它不再被需要）来处理相邻座位警告的有条件显示：

```

if (reservation.getSeatsAreAdjacent()) {
    Element adjacentWarning =
        showReservationXmlcObject.getElementNonAdjacentWarning();
    adjacentWarning.getParentNode().removeChild(adjacentWarning);
}

```

最后，在返回已配置好的XMLC对象供超类用来建立响应对象之前，我们获得显示座位安排计划图和设置其源代码URL的元素：

```

HTMLImageElement graphicEle = (HTMLImageElement)
    showReservationXmlcObject.getElementSeatingPlanImg();
graphicEle.setSrc("static/seatingplans/" +
    performance.getShow().getSeatingPlanId() + ".jpg");

return showReservationXmlcObject;
}

```

XMLC提供了一种与大多数视图技术非常不同的分离表示模板与动态代码的方法。它的优点是：它所使用的标准HTML模板可以在浏览器中显示得和它们在运行时将出现的完全一样，并被填充上动态数据；以及它可以处理人类难以读懂的HTML。但是，它在开发阶段需要一个额外的“编译”步骤，并且需要比学习Velocity之类的较简单技术花费更多的学习精力。

关于XMLC的进一步读物

要了解关于XMLC的进一步信息，请参见下列资源：

- <http://xmlc.enhydra.org/software/documentation/xmlcSlides/xmlcSlides.html>: XMLC的一个简单介绍。
- <http://staff.plugged.net.au/dwood/xmlc-tutorial/index.html>: XMLC Tutorial。
- <http://xmlc.enhydra.org/software/documentation/doc/xmlc/user-manual/index.html>: XMLC用户手册。
- <http://www.java-zone.com/free/articles/Rogers01/Rogers01-1.asp>: XMLC的综述以及与JSP的比较。

二进制内容的生成

迄今为止，我们已经见证了置标的生成。如果我们需要生成二进制内容，怎么办呢？

我们已讨论过的各种模板技术不适合生成二进制内容。例如，它们中没有一个能够让我们对生成的空白空间进行足够的控制。

不过，我们或许可以使用一种XML方法。XSL-FO（XSL Formatting Objects）是XSL规范的另一半（不同于XSLT），定义了一种描述布局的XML格式。将来，XSL-FO可能会被浏览器和其他GUI所理解。目前，XSL-FO必须被转换成另一种布局格式才能显示，比如PDF。关于XSL-FO的进一步信息，请参见<http://www.ibiblio.org/xml/books/bible2/chapters/ch18.html>站点。关于Apache FOP——少数几个能够把XSL-FO转换成可显示格式的现有产品之一，请参见<http://xml.apache.org/fop/>站点。尤其是，Apache FOP还支持PDF和SVG。

有时，我们可能会与HttpServletReservation对象直接打交道。例如，我们可以实现自己的View接口来从响应对象中获得ServletOutputStream，并输出二进制数据。二进制数据可能被包含在控制器所提供的模型中。

可是，我们常常可以使用一个助手类来为我们希望生成的二进制格式提供一个抽象：例如，如果该格式是众所周知的，并公开用文件说明过，比如图像格式或PDF。下面通过分析PDF生成来举例说明这种方法。

利用iText生成PDF

严格地说，PDF不是一种二进制格式。可是，它不是人类可读的，并且可以含有像图像数据那样的ASCII码二进制数据，所以我们必须按照和二进制格式相同的方式来对待它。

PDF被正式向外界公开。它是商业创作者Adobe公司所销售的PDF工具，但享有该规范。因此，就有几个可以用来生成PDF文档的开放源Java库。

笔者使用的是iText版本0.93——由Bruno Lowagie所编写并受Gnu GPL保护的一个PDF生成库。读者可以从<http://www.lowagie.com/iText/>站点上获得它，而且这个站点提供了优秀的资料和许多使用了iText的Java代码示例。iText还提供了一个用来生成(X)HTML、XML和RTF的类似模型，尽管它的主要焦点是PDF生成。

PDF生成是对笔者曾经拒绝用来生成HTML的“生成库”方法的一种完美应用。这是一种复杂的格式，没有任何一种模板语言能够用于它，而且没有一个高级别抽象，Java应用代码不必非要处理它。

使用iText生成PDF很简单，也相当直观。和XMLC的情形一样，我们将需要一个应用特有类来生成每个PDF视图。

同往常一样，首先在WEB-INF/classes/views.properties中创建一个视图定义。和XMLC的情形一样，没有必要组件属性，尽管我们可以指定属性来自定义页面大小和其他输出参数。完整的视图定义如下所示：

```
showreservation.class=
com.wrox.expertj2ee.ticket.web.pdfviews.ShowReservationView
```

和XMLC中的情形相同，我们的MVC框架提供一个便利超类——在这种情况下是com.interface21.web.servlet.view.pdf.AbstractPdfView抽象类，以便使用Template Method设计模式来简化应用特有的PDF生成类。子类必须实现下列这些保护抽象方法，以便把模型数据写给被作为第二个参数传递的iText PDF文档。通常用不到请求和响应对象，但我们包含了它们，以防视图需要地区信息或写一个Cookie。

```
protected abstract void buildPdfDocument(Map model,
    Document pdfDoc,
    HttpServletRequest request,
    HttpServletResponse response)
    throws DocumentException;
```

要想详细了解如何安装iText和com.interface21.web.servlet.view.pdf.AbstractPdfView框架超类的实现，请参见附录A。

用来显示预订的应用特有PDF视图首先从子类化AbstractPdfView超类开始：

```
public class ShowReservationView extends AbstractPdfView {
```

接着，定义我们在生成输出时使用的数据和字体。理想情况下，这个内容应该被保存在一个ResourceHandle中，以便该视图实现能够使用适合请求地区的 ResourceBundle。需要注意的是，由于我们不能使用一种模板语言，所以我们的Java代码将不得不至少处理含有输出串的变量：

```
private static final String MESSAGE1 =
    "{0} tickets have been reserved for you for " +
    "(1) minutes to give you time to " +
    "complete your purchase. The seat numbers are: ";
private static final String COSTING =
    "The total cost of these tickets will be {0}. " +
    "This includes a booking fee of {1}.";
private static final String ADJACENT_WARNING =
    "Please note that due of lack of availability, some " +
    "of the seats offered are not adjacent";
private static final Font HEADLINE_FONT =
    new Font(Font.TIMES_NEW_ROMAN, 15, Font.BOLD, Color.red);
private static final Font HEADING_FONT =
    new Font(Font.HELVETICA, 13, Font.ITALIC, Color.black);
private static final Font TEXT_FONT =
    new Font(Font.HELVETICA, 11, Font.BOLD, Color.black);
private static final Font WARNING_FONT =
    new Font(Font.HELVETICA, 12, Font.BOLD, Color.black);
```

我们必须实现buildPdfDocument()抽象保护方法来生成内容：

```
protected void buildPdfDocument(Map model, Document pdfDoc,
    HttpServletRequest request, HttpServletResponse response)
    throws DocumentException {
```

和XMLC视图中的情形一样，我们首先从映像中抽取出必需的模型对象。由于这些模型对象的控制器必须已使它们变得可使用，所以我们不必检查它们是不是非空值。我们可以简单地允许一个NullPointerException，因为这相当于一个断言故障（在Java 1.4中，每个控制器方法最后都可以得到一个断言：必需的模型关键字都是非空值）：

```
Reservation reservation = (Reservation)
    model.get(TicketController.RESERVATION_KEY);
Performance performance = (Performance)
    model.get(TicketController.PERFORMANCE_KEY);
PriceBand priceBand = (PriceBand)
    model.get(TicketController.PRICE_BAND_KEY);
```

接着，使用我们在前面已经见过的相同代码，根据请求地区格式化日期和货币量：

```
SimpleDateFormat df = (SimpleDateFormat)
    DateFormat.getDateInstance(DateFormat.SHORT, request.getLocale());
df.applyPattern("EEEE MMMM dd, yyyy");
String formattedDate = df.format(performance.getWhen());
df.applyPattern("h:mm a");
String formattedTime = df.format(performance.getWhen());
NumberFormat cf = NumberFormat.getCurrencyInstance();
String formattedTotalPrice = cf.format(reservation.getTotalPrice());
String formattedBookingFee =
    cf.format(reservation.getQuoteRequest().getBookingFee());
```

现在，可以开始生成动态内容。这采取把代表文档内容的新对象增加到Document对象上的形式：

```
String title = "Reservation for " + performance.getShow().getName();
pdfDoc.add(new Paragraph(title, HEADLINE_FONT));
String when = formattedDate + " at " + formattedTime;
pdfDoc.add(new Paragraph(when, HEADING_FONT));
pdfDoc.add(new Paragraph());
String note = MessageFormat.format(MESSAGE1, new String[] { "" +
    reservation.getSeats().length,
    "" + reservation.getMinutesReservationWillBeValid()});
pdfDoc.add(new Paragraph(note, TEXT_FONT));
```

在这个模型中，迭代由普通Java代码来处理，使迭代变得直截了当的：

```
List list = new List(false, 20);
list.setListSymbol(new Chunk("\u2022",
    new Font(Font.HELVETICA, 20, Font.BOLD)));
for (int i = 0; i < reservation.getSeats().length; i++) {
    list.addItem(reservation.getSeats()[i].getName());
}
pdfDoc.add(list);

pdfDoc.add(new Paragraph());
note = MessageFormat.format(COSTING, new String[] {
    formattedTotalPrice, formattedBookingFee });
pdfDoc.add(new Paragraph(note, TEXT_FONT));
pdfDoc.add(new Paragraph());
```

同样，有条件地包含关于座位不相邻的警告需要使用一条简单的Java if语句：

```
if (!reservation.getSeatsAreAdjacent()) {  
    pdfDoc.add(new Paragraph(ADJACENT_WARNING, WARNING_FONT));  
}  
}  
}
```

如果说PDF很复杂，这却是相当简单的，但也说明了这样的方法为什么不是生成HTML的一个好主意。让Java代码来处理文本内容是十分笨拙的，而且要想得到对完成后的文档看起来将是什么样子的感性认识也很困难。如果可选择一种基于模板的方法，该方法则要自然得多。

我们刚才创建的PDF文档看起来如图13.8中所示。

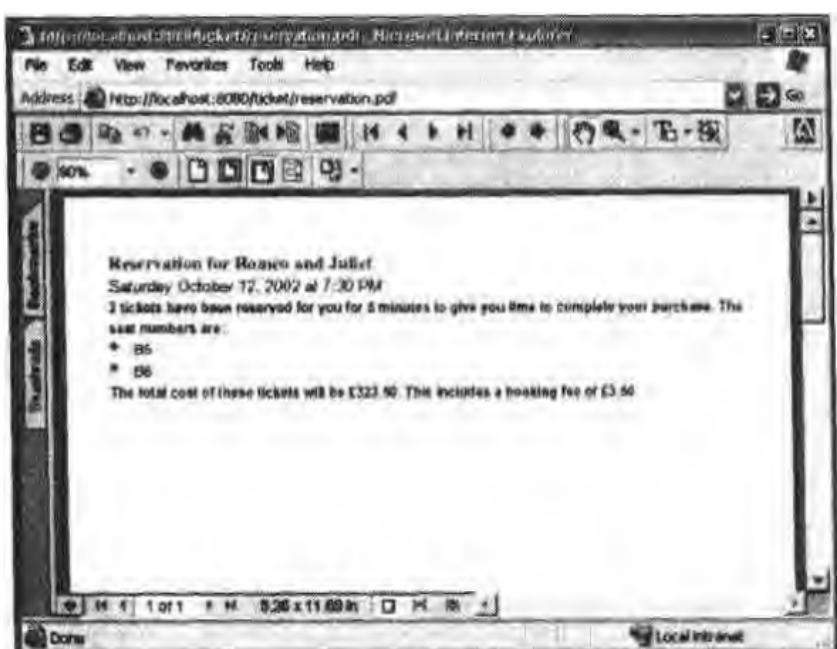


图13.8

iText支持给PDF文档增加图像，所以我们也可以显示座位安排计划图。要想了解进一步信息，请参考相关的iText资料。

视图合成和页面布局

读者很可能正在想，“这很不错，但前面的示例HTML却非常简单。我们的站点中还需要使用标题、页脚和导航条，不是只有一个内容页面”。

视图结构可以比我们在前面已见过的简单HTML文档复杂得多。但是，正如笔者将要设计演示的，这不会给MVC设计或我们的视图方法造成问题。

前面所讨论过的所有模板设计技术都允许我们用多个动态和静态块来合成视图。例如，在JSP中，视图合成通过使用包含指令和包含标准动作来实现，而WebMacro和Velocity都提

供了包含指令。

有两种基本方法，笔者将它们称做内容页面（Content page）包含和模板页面（Template page）包含。在内容页面包含中，每个内容页面包含其他元素：通常是一个标题和页脚。在模板页面包含中，一个布局或模板页面根据需要包含内容页面。同一个布局页面可以用于整个应用，包含不同上下文中的不同内容井、标题、页脚等。如果必要，可以轻易地定义多个模板。

模板页面包含比内容页面包含灵活得多。它允许一个模型里面的页片断都是构件，以后这些构件由模板页面负责装配起来。它还使页面结构的全局修改变得很容易：只需简单地通过修改模板页面即可。如果使用内容页面包含，我们将需要修改许多个别页面来使修改一致。

由于JSP 1.2纠正了JSP 1.1动态包含机制的各种严重问题，所以JSP已经提供对模板页面包含的充分支持。不过，需要一个更高的抽象级别。例如，Struts就提供了满足这一需要的“模板”标志库。

当前框架通过在视图上使用静态属性来解决模板页面包含。静态属性是可供视图使用的数据，并且这些数据不必来自控制器所提供的模型。因此，静态属性可以是表示特有的，也可以提供对模板页面的有用支持。静态属性将由同名的模型值来替代，因此也可以用来提供默认值。

下面通过介绍一个简单布局的JSP实现来举例说明静态属性的使用，其中这个实现包括一个标题和页脚。

内容页面包含意味着每个页面包括了一个标题和页脚，就像下面这样：

```
<%@ include file= "header.jsp" %>
Content well content for this page
<%@ include file= "footer.jsp" %>
```

请注意，由于我们知道每种情形下包含进来的页面的名称，所以可以使用JSP静态包含（通过一个指令而不是`<jsp:include>`标准动作）。这是一种简单而又直观的方法，但常常不适用于大而又复杂的应用。虽然对包含进来的标题和页脚JSP页的修改将自动影响内含了这些JSP页的所有页面，但如果我们需要在一个单独的内含JSP页中增加额外的模板内容，怎么办呢？这样的改动将需要对所有JSP页都进行修改。关键问题是，每个JSP页都含有相同而又多余的布局信息。

相反，模板页面包含仅需要一个模板，这个模板将包含标题与页脚，以及在运行时将随时变化的内容井。页面布局在该模板中将只定义一次。一个简单的模板看起来可能像下面这样：

```
<%@ include file= "header.jsp" %>
<jsp:include page= "<%=contentWell%>" />
<%@ include file= "footer.jsp" %>
```

`contentWell`变量将需要作为一个请求属性被提供给模板页面，或需要由其他某个方法提供给模板页面。需要注意的是，一个JSP动态包含是我们不知道自己需要包含什么页面时的

惟一选择。这个简单的示例为标题和页脚使用了静态包含，当然这些包含的任何一个也可以被参数化。

动态包含与静态包含问题在JSP 1.0和1.1中比在JSP 1.2中重要得多，因为<jsp:include>动作始终清洗掉缓冲区，而这未必总是受欢迎的行为。JSP 1.2消除了这个局限性。这就意味着现在使用动态（而不是静态）包含只有较小的速度影响。

每个内容井页面只含有与它的业务用途相关的输出。但是，在模板内，这个内容在模板中它出现的地方应是合法的HTML，因而给对使用了这种技巧的布局合成施加了一些限制。

现在来看一看如何实现模板页面包含的一个实际实现，其中我们利用静态属性来给模板提供必要的信息。需要预先说明的是，我们的所有视图都是从com.interface21.web.servlet.view.AbstractView类中派生出来的，并且这个类允许我们用一个CSV来设置串的静态属性。我们可以设置3个静态属性：header（标题在WAR内的URL）、contentWell（内容井的URL）和footer（页脚的URL）。无论动态模型是什么，这些静态属性值是所有模板都可以使用的，而且它们定义了合成视图的布局。

这个继承来的支持意味着我们可以像使用任一JSP页一样使用InternalResourceView。当我们使用模板页面包含时，把视图定义中的URL改换成模板页面的URL，但把原URL（/welcome.jsp）设置为contentWell静态属性的值。请注意，页脚是一个静态HTML资源，因为这种方法可以使用WAR内的任何资源：

```
welcomeView.attributesCSV= header=[/jsp/header.jsp],\
                           contentWell=[/welcome.jsp],\
                           footer=[/jsp/footer.htm]
```

每个属性值的第一个和最后一个字符都被忽略，以便“,”或“=”字符能在必要时被包含在属性值内。虽然当前示例使用了“[”和“]”字符来围住每个值，但也可以使用其他任何一对字符。请注意“\”字符的使用，正如java.util.Properties资料中所描述的，它的作用是分解一个属性值以便于阅读。

这3个静态属性值将出现在我们的模型中，好像它们是由控制器所提供的数据。这意味着当我们使用JSP时，它们将可以作为请求属性来使用；我们还可以声明String类型的JSP组件来访问它们。

示例应用中的/jsp/template.jsp文件就是模板文件。在禁用了自动会话创建（在任一JSP页中都应该是第一项任务）之后，它声明了这3个组件：

```
<%@ page session="false" %>

<jsp:useBean id="header"
              type="java.lang.String"
              scope="request"
            />

<jsp:useBean id="contentWell"
              type="java.lang.String"
              scope="request"
            />

<jsp:useBean id="footer"
              type="java.lang.String"
              scope="request"
            />
```

现在，使用这些值来构造输出并控制各组成部分的包含就微不足道了：

```
</html>
<head>
<title>Could also parameterize this</title>
</head>

<jsp:include page="<%#header%>" />

<hr>
<jsp:include page="<%#contentWell%>" />

<hr>
<font size="2">
<jsp:include page="<%#footer%>" />
</font>

</body>
</html>
```

现在，使用示例应用的欢迎页面来试试这个模板页面。/WEB-INF/classes/views.properties中的视图定义如下所示：

```
welcomeView.class=com.interface21.web.servlet.view.InternalResourceView  
welcomeView.url=/welcome.jsp
```

现在（如果对控制器或模型代码或welcome.jsp文件不再改动），我们的欢迎页面看起来将会如图13.9所显示。



图 13.9

在实践中，我们需要保证内含的页面构件不包含<html>或<body>标志，所以这是一个有点过分的简化。

这种方法（使用一个视图实现和若干“静态属性”）有一个明显胜过自定义标志的优点，因为它可以利用除了JSP之外的视图技术，比如WebMacro和Velocity。

使用Java代码而不是一种模板技术来组合多个视图也是可能的。例如，RequestDispatcher接口除了提供了我们已经见过的forward()方法之外，还提供了include()方法；这个方法可以被一个自定义的View实现用来包含多个JSP页的输出或一个WAR内的其他资源。作为一种选择，我们也可以提供View接口的一个CompositeView实现，并让这个实现能够把连续地再现几个视图（可能具有不同类型）的结果输出出来（笔者成功地使用JSP 1.1的视图合成方法避开了动态包含的那些局限性）。

当涉及到了视图合成时（比如在使用模板页面的场合下），应该把细节隐藏在视图代码中。这些细节不是控制器或模型所关心的事情。

无论选择了哪种J2EE视图技术，不要忘了遵守在置标层自身内把表示与内容分离开这一相同的原则。如果使用XML和XSLT，这不成问题。当生成HTML时，要确保HTML使用了CSS格式表来保证动态生成的内容与再现信息之间的分离。

小结

在贯穿于本章的讲解示例中（取自本书的配书示例应用中），我们已经见识了良好的MVC设计习惯怎样使既改换视图技术又无需修改任何控制器或模型代码的情形成为了可能。

虽然JSP是核心的J2EE规范之一，但它仅仅是许多可供选择的视图技术之一。在本章中，我们已经了解了JSP和其他几种可供选择的领先视图技术，分析了每种技术的优缺点，以及应该在什么时候选用它。另外，还分析了下列可供选择的视图技术，演示了在实践中怎么才能把它们综合到MVC Web应用中。

- WebMacro
- Velocity
- XML/XSLT方法
- XMLE
- 使用iText库的PDF生成，以及二进制内容的生成

这些视图技术没有一种是完美的，也没有一种完全适合所有项目。除了每种技术固有的优缺点之外，技能的可获得性将是开发每个项目时的一个重要考虑因素。

JSP的优点是它被包含在核心的J2EE标准中。它是一种优秀的视图技术，只要它的使用服从于严格的规范。我们已经见识了保证JSP页可维护的编码标准。严格的编码标准是不可或缺的，因为错误使用JSP的后果是严重的，而且这种后果在实践中可以经常看到。JSP Standard Template Library (JSTL) 于2002年的发布是JSP的一个巨大进步，JSTL应该被看做JSP的一个实质性部分，应该用在使用了JSP的每个应用中。JSP 2.0把这个库的表达式语言转移到JSP核心中，并引进更多的增强。这些增强将消除JSP模型对脚本小程序的不信任依赖。

Velocity模板语言是一种简单而有效的视图技术，是适合许多应用的一个好选择。它比JSP更简单，也更容易让HTML开发人员学会，只提供了实现简洁视图所需要的各种特性，没有提供至今仍困扰着JSP的各种诱惑。Velocity宏特别精巧，提供了一种无需Java编程即可采用公共内容的、简单而有效的方法。正如我们在第15章中将要看到的，Velocity提供了杰

出的性能。

XSLT和XMLC提供了两种与JSP非常不同的模型，以及诸如Velocity之类的模板语言，其中每种模型都提供表示与动态内容生成之间的有效分离。它们在项目中的适合程度将视总体创作策略而定。

正如我们已经见过的，在同一个应用中混用不同的视图技术是可能的。我们讨论了XSLT在JSP页内的使用。也根本不存在不能把Velocity用于某些视图，而不能把JSP页用于其他视图的任何理由。这或许能使我们从各种技术中获得解决具体问题的好处。不过，此类混用的缺陷是它使部署变得更复杂，并增加了开发和维护一个应用所需要的技能范围。当然，在一个设计完善的应用中，使用不同技术来生成几个视图应该没有任何困难——例如使某些内容可以按PDF文档的形式使用。

请记住，视图技术的选择不应该对应用设计造成意义深远的影响。有时，视图技术的选择可能会从应用的业务需求和体系结构中自然得出。例如，如果数据在应用内存在为XML，XSLT视图可能就是一个显而易见的选择。不过，在视图技术之间进行选择而又不改变应用体系结构一般是可能的。业务逻辑构件甚至Web层控制器应该不受视图技术变化的影响。

作者说明：在JSP Standard Tag Library发布以前，笔者想起了这样一个问题：JSP对大多数应用是不是一个明智的选择。JSTL打消了过去针对JSP提出的许多正当的批评，并为应用视图提供了一个坚强的基础。虽然基于XML的脚本设计（就像包括JSTL标志在内的所有JSP自定义标志所提供的）仍会是笨拙的，但带有了JSTL的JSP却为大多数应用提供了一个强有力而又相当简单的解决方案。

无论选择了何种视图技术，不要忘了用文字详尽地说明Web层控制器所暴露的模型对象。这相当于控制器与视图之间的一份合约。

我们还分析了视图合成：用多个构件块构造复杂视图。这些构件块可以是其他视图的输出或内含式JSP页之类的页面构件。我们讨论了这是如何使用我们的MVC框架来实现的，而不管我们使用了何种视图技术。

在下一章中，我们将讨论部署问题。在第15章中，我们将谈一谈性能调整与测试的重要题目，包括在本项目中所讨论的部分视图技术的测试基准及HTTP高速缓存问题。

第14章 应用的包装与部署

本章将讨论如何打包J2EE应用，以及如何把它们部署到应用服务器上。这是我们所需要应用服务器特有知识的一个方面，也是应用服务器间可移植性目前受到限制的一个方面。

由于应用服务器使用不同的类装入方法，所以我们可能需要针对不同的应用服务器来不同地打包应用。甚至连EAR之类的标准部署单元也可能不是完全可移植的。我们还将需要遵守服务器特有的约定来定义应用所必需的资源，比如JDBC DataSources。正如我们要看到的，标准J2EE部署描述符没有提供所有必需的信息，这意味着我们一般需要另外的专用部署描述符。虽然各种J2EE规范描述了标准部署单元的结构，但任何一个标准部署过程（至今）都不是统治一切的。

由于应用部署的细节随着应用服务器的不同而有所不同，所以笔者在这里的目标是提供一个入门，重点给出读者研究自己的目标服务器的行为而需要知道的一些领域。在读完本章之后，读者应该对不同服务器上的应用部署之间的共性，以及打包和部署一个典型J2EE应用到一个服务器上所必需的步骤有一个感性认识。

我们通过讲解读者需要做些什么才能让示例应用在JBoss 3.0中运转起来，来举例说明所涉及到的各种概念。有关部署示例应用到其他服务器上的信息，请查看本书的配书下载。

部署选择对性能会产生重要的影响，所以至关重要的是必须贯穿于整个应用生成周期来探讨部署选择，不要把部署选择仅限于让应用在选择的服务器上运转起来这一项上。

包装

首先从了解如何打包J2EE应用开始。

部署单元

J2EE应用中最常用的两个部署单元是Web Archive (WAR) 和EJB JAR文件。这些都是JAR格式的文件，它们含有如下内容：

- 一个Web应用的各种实现类、二进制依赖内容、文档内容（比如JSP页、静态HTML和图像）以及部署描述符。如果我们没有使用EJB，一个WAR文件就可以包含一个J2EE Web应用所需要的全部代码和二进制文件。
- 一个EJB部署的各种实现类和部署描述符，而一个EJB部署可以包括多个EJB。

一般说来，这些部署单元每个都既包含应用服务器特有的、标准J2EE部署描述符，又包含应用服务器特有的专用部署描述符。

在应用使用了整个J2EE栈时，构成这个应用的WAR和EJB JAR部署单元可以被包含在一个叫做Enterprise Archive (EAR) 的统一J2EE部署单元中。EAR文件含有一个附加的部署描述符application.xml，该部署描述符标识构成该应用的各种J2EE模块。在本书中，我们

已经介绍了实践中最常使用的**WAR**和**EJB JAR**文件模块类型。也可能把Java应用客户和**J2EE Connector Architecture (JCA) Resource Adapter**包含在一个**EAR**中。

在涉及布置式应用的地方，**EAR**部署一般是最佳选择，它提供了方便的部署，并准确地反映了该应用的语义。因此，我们将给示例应用使用**EAR**部署。

但是，**EAR**部署可能不太适合分布式应用。如果每个服务器上的**Web**层构件总是使用同一个服务器上的**EJB**，采用分布式体系结构是毫无意义的。因此，在分布式应用中，**EAR**中的**EJB**客户构件（**WAR**）很可能会在运行时与运行在另一个服务器上的**EJB**进行通信，而不是与同一个服务器上的那些**EJB**实例进行通信。对采用分布式体系结构有利的主要理由之一，是把附加软件专用于已知瓶颈（比如执行费时处理的**EJB**）的可能性。在这样的情况下，所有构件在所有服务器上的**EAR**部署可能是浪费和误导的，因为目标不是让所有服务器都运行所有应用构件。

对于适合分布式应用的**EAR**部署，一种替代方法是把**Web**应用与**EJB**部署分开，因为我们知道调用**EJB**要通过**RMI**来进行。不过，分开模块中的部署可能会比较复杂。我们需要在**EJB**与**Web**部署中同时包含**EJB**客户视图（主接口与构件接口，但不包括**EJB**实现类和使用了它们的助手类）。

正如我们在第2章中曾经说过的，对一个**EAR**、**WAR**或**EJB JAR**部署单元运行**J2EE Reference Implementation**的**Verifier**工具来检查对各种**J2EE**规范的符合性是可能的。示例应用的**Ant**生成脚本包含了一个运行该检验器的目标。这样的检验应该定期进行以保证应用是符合规范的，而且因为它还提供了一个容易的顶部部署错误检查。该检验器工具以一种详细而又一致的方式报告诸如缺少类或无效部署描述符之类的问题。与一些**J2EE**服务器的输出结果相比，它可能提供关于部署故障原因的更明确信息。

扩充式部署单元

大多数服务器都允许部署单元以“扩充”或“分解”的形式被部署：也就是说，作为一个目录结构，而不是作为一个固定目录结构中的一个档案文件。一般说来，扩充式部署在开发中最有用；我们需要把个别部署单元转出到生产中。

在开发中，扩充式部署的优点是：它通常允许更新个别文件，不必全面重新部署。例如，在开发期间，仅修改一个**JSP**页就不得不重新部署整个应用是令人无法接受的。示例应用的**Ant**生成脚本包含了一个把该应用作为一个**EAR**（含有一个**EJB JAR**文件），而不是作为一个扩充式**WAR**来部署的目标。这使得修改**JSP**页和其他**Web**内容同时又不必重新部署该应用成为了可能。

了解**J2EE**类装入

在打包**J2EE**应用时，最难应付的问题可能与多模块应用中的类装入有关，这会影响到：

- 我们打包应用的方式，尤其是在我们包含由**EJB JAR**和**WAR**模块同时使用的类方面；
- 应用服务器之间的可移植性。不同应用服务器中的类装入行为之间存在差别，对可移植性来说，这种差别会是一个真正的问题，而且意味着在一个服务器上工作的**EAR**在另一个服务器中可能就不工作，即使它是在各种**J2EE**规范的范围内被编码的。

除非我们知道J2EE服务器类装入可能会工作并吸取适当的教训（我们只能说“可能”，因为它随着应用服务器的不同而有所不同），否则会有遇到神秘的**ClassNotFoundException**或**ClassCastException**的危险。

虽然有充分的理由相信类装入会正常工作，但令人遗憾的是，J2EE类装入的复杂性会影响应用开发人员，并降低工作效率。因此，了解涉及到的问题是很重要的，尽管这些问题很复杂。

以下的讨论将把重点放在用EAR打包应用（实践中最常用的方法）上，而不是放在分别打包EJB JAR和WAR模块上。

Java类装入概念

首先来看一看Java 2类装入如何工作。下面这两条基本原则将始终适用：

- 每个类都与装入它的装入器保持一种联系。`java.lang.Class`的`getClassLoader()`方法返回装入了这个类的类装入器，这个类在被装入之后就不能被修改。如果这个类试图按名装入其他类，被使用的正是该类装入器。
- 如果同一个类被两个类装入器装入，这两个类装入器所装入的类将不是类型兼容的（尽管串行化仍将工作）。

`java.lang.ClassLoader`类的技术资料为类装入器另外定义了下列行为：

- 类装入器是分级的。当一个类装入器被要求装入一个类时，它首先请求它的父类装入器来尝试着装入这个类。只有当父类装入器（及其祖先）都没有能装入这个类时，原来的类装入器才会尝试着装入这个类。类装入器分级结构的最顶端是内建在JVM中的引导装入器（Bootstrap Loader），它装入`java.lang.Object()`。
- 虽然一个类装入器能够看到它的父辈（们）所装入的类，但它看不到它的子女所装入的类。

由于实现自定义类装入器是可能的（大多数应用服务器都提供几个），所以也可能不符合上面所描述的分级结构行为。

J2EE中的类装入

J2EE服务器使用多个类装入器的主要原因是这允许动态应用重装入。很明显，我们不希望在重新部署一个应用时重装入应用服务器自己的所有类。这意味着应用服务器始终需要重新启动。因此，应用服务器为应用代码使用了不同的类装入器，比如为应用代码所使用的类装入器就不同于为应用服务器自己的标准库所使用的类装入器。一般说来，会为部署在服务器上每个应用都创建一个或多个新的类装入器。

但是，通常不为同一个应用中的不同应用特有类使用多个类装入器，除非我们使用了EJB（JSP页可能被分配一个单独的类装入器，但这通常不影响应用代码）。

正如笔者在前面所提过的，与纯Web应用的部署模型相比，使用EJB会使部署模型变得相当复杂。类装入的情形也是如此。在WAR中，我们可以简单地把所有二进制依赖内容包含在`/WEB-INF/lib`目录中。不过，当WAR访问EJB时，事情会变得更复杂。

为了弄清楚原因，现在来考虑一种在应用服务器中实现类装入的常用方法。在以WAR

形式部署为一个企业综合应用的应用中，EJB类装入器常常是WAR类装入器的双亲。例如，Orion和WebLogic都使用了这种方法。这是一种自然的实现方法，因为WAR一般将访问EJB（因此需要能够至少看见EJB客户类），而EJB不访问Web构件。

但是，它不是惟一有效的实现方法，因此下列讨论不适用于所有应用服务器。

最后得到的的类装入分级结构看起来将像图14.1所示。实际上，更多的类装入器可能会被牵涉进来，但该图中所显示的这些类装入器是对应用代码最有意义的3个类装入器。

在该图中，类装入器分级结构用一些封闭框表示。父 - 子关系用一个封闭框表示。



图14.1

由于采用标准J2EE分级类装入器行为，所以这样的分级结构将意味着任何一个类都可以访问围住其类装入器的框中的类。但是，与外层框相关的类不能装入内层框中的类。因此，Web应用类可以看见被部署在该应用的EJB中的类和系统类。但是，EJB类看不见Web应用类，而且被安装在服务器级上的类看不见应用特有的类。

企业综合应用中的类装入

当然，这种类装入只有J2EE应用服务器的实现者才会感兴趣。令人遗憾的是，应用开发人员也不能视而不见，而且也不能不知道J2EE类装入的实现。

假设采用上述类装入器分级结构，现在来考虑一个似乎合理的情景。请设想EJB和Web构件中同时使用的一个应用类或一个框架类试图装入WAR内的一个类。例如，请设想EJB容器中所使用的一个BeanFactory实现，也被一个WAR内的代码用来按名装入类并操纵这些类。即使这个基础结构类在该WAR中是可见的，但它也不能装入WAR类。由于它是由EJB类装入器所装入的，所以它看不见该WAR中的、由一个子孙类装入器所装入的类。

通过简单地把类定义同时保存在WAR和EJB JAR文件中，我们也未必始终能解决这个问题，因为这可能会引起类数据类型强制异常，只要这两个类装入器最终独立地装入了一个或多个类。

因此，在EJB和Web模块被同时包含在同一个企业应用内这种常见情况下，存在两个与J2EE类装入有关的基本问题：

- 我们把EJB和Web应用中都使用的类的定义保存在什么地方？
- 我们怎么保证两个类装入器最终将不保存同一个类的独立版本，以避免出现类数据类型强制异常？

Servlet 2.3规范的类装入建议

不仅类装入的实现是不同的，不同J2EE规范对待类装入也是不同的。

Servlet 2.3规范（§ 9.7.2）叙述道，“建议把应用类装入器实现成这样：包装在WAR中的类和资源比驻留在容器级库JAR中的类和资源优先得到装入”。

这显然与java.lang.ClassLoader类的Javadoc中所描述的标准J2SE类装入行为相抵触。由于WAR类装入器必须是一个动态类装入器，所以它必须是应用服务器所提供的另一个类装入器的子女。因此，Servlet 2.3规范的上述建议与标准Java 2类装入行为正好相反，因为后者明确规定，只有当类不能被祖先类装入器解决时，它们才会由子女类装入器来装入（在这种情况下是WAR类装入器）。

这个建议（注意它不是一项要求）对EJB适合放在这个建议的类装入模型中的什么地方是不明确的。EJB大概不会被看做是“驻留在容器级库JAR中的类和资源”，在这种情况下，该要求就不适用于这些EJB。

Servlet 2.3规范与标准Java 2类装入行为之间的这个矛盾由这样一个事实得到强调：Sun公司的J2EE 1.3 Compatibility Test Suite在默认实现Servlet 2.3型反向类装入的服务器上是无效的。由于这一缘故，许多服务器或者不实现该Servlet 2.3建议，或者只把它提供为一个配置选项。示例应用中所使用的JBoss/Jetty包默认成使用标准Java 2类装入行为，尽管它可以被配置成使用Servlet 2.3 WAR优先行为。Oracle iAS采取了相同的方法。

Servlet 2.3型类装入的主要好处是允许我们把一个应用所必需的任何补丁库做为该应用的一部分来发送，无需更改本服务器安装。例如，第13章中所讨论的XMLC 2.1 Web内容生成技术就需要随同某些应用服务器一起发送的XML库的补丁版本。由于Servlet 2.3类装入的缘故，我们可以把必需的补丁包含在WEB-INF/lib目录中，无需对总体服务器配置做任何修改，也没有与其他应用相抵触的危险。

J2EE中的Java 1.3扩展机制体系结构

另外，我们还需要重视J2SE类装入改进。J2SE 1.3中的改动使得JAR文件指定与其他JAR文件的相关性成为了可能：通过在相对文件路径的/META-INF/MANIFEST.MF文件的一个Class-Path头部内指定一个由空格分隔的相对文件路径列表。J2EE 1.3规范的第8.1.1.2节要求应用服务器支持EJB JAR文件的相对文件路径列表。下列示例摘自示例应用的ticket-ejb.xml文件的MANIFEST.MF文件，并举例说明了这种机制在示例应用中的使用：

```
Class-Path: log4j-1.2.jar;i21-core.jar;i21-ejbimpl.jar i21-jdbc.jar
```

这行代码声明ticket-ejb.xml文件中的应用特有类都依赖于4个基础结构JAR文件，意味着该EJB JAR文件不必包含任何第三方类。这些路径是相对的。所有这些JAR文件都随同该EJB

JAR文件一起直接包含在应用EAR的根中，正如应用EAR的下列内容清单所显示的：

```
META-INF/  
META-INF/MANIFEST.MF  
META-INF/application.xml  
i21-core.jar  
i21-ejbimpl.jar  
i21-jdbc.jar  
ticket-ejb.jar  
ticket.war  
log4j-1.2.jar
```

某些应用服务器支持WAR和EAR部署单元的载货清单类路径机制，但由于这些单元不被类装入器直接装入，所以这不是J2SE 1.3或J2EE 1.3规范所要求的。例如，参见<http://otn.oracle.com/tech/java/oc4j/htdocs/how-to-servlet-warmanifest.html>站点上的资料，就可以了解到如何启用Oracle 9iAS Release 2上的WAR载货清单类路径（这一支持在默认情况下被禁用）。WebSphere 4.0也支持WAR文件的载货清单类路径，并且IBM资料（参见http://www-3.ibm.com/software/webservers/appserv/doc/v40/aee/wasa_content/060401.html站点）建议在WAR和JAR文件引用相同的类时使用这一机制。Orion和Oracle也默认地装入EAR中的载货清单类路径。需要注意的是，JBoss/Jetty似乎没有尊重WAR中的载货清单类路径，所以笔者在打包示例应用时没有依靠这个不可移植的特性。J2EE Reference Implementation也忽略了WAR中的载货清单类路径。

Servlet 2.3规范要求Web容器尊重Web应用的WEB-INF/lib目录中所包含的库文件载货清单类路径。可是，这在集成EAR部署中是有问题的，因为在涉及到一个WAR的地方相对路径应该是什么不是十分明确。从一个档案文件内的一个嵌套目录中看，相对路径的含义是什么？例如，如果一个.war文件连同它所引用的那些EJB JAR文件一起被包含在一个EAR的根目录内，那么库JAR应该使用下列两个似乎合理的相对路径中的哪一个呢？

- *../../../../other-jar-file.jar*: 导航到WEB-INF目录再到本WAR的根，并假设那个（些）库JAR和本WAR的根在同一个目录中。
- *../../../../other-jar-file.jar*: 向上导航一个目录，进而假设该.war文件已被抽取到位于本EAR文件的根下面的它自己的目录中，这也是大多数服务器使用的一种方法。

上述任一替代方法在JBoss/Jetty中都不工作。因此，在一个WEB-INF/lib目录中的JAR文件中使用载货清单类路径不是可移植的。或许正是由于这个缘故，J2EE 1.3规范（§ 8.1.1.2）建议，把共享库包含在WEB-INF/lib目录中是必需的，即使它们被包含在了同一个EAR文件内的其他地方。

J2EE规范的第8.1.1.2节没有要求解决EAR文件外部的类，比如安装在服务器级上的库。这在某些服务器中可能适用，但不是标准的。如果一个应用依赖于外部二进制文件，比较好的做法通常是只使用你的服务器在服务器级上安装二进制文件的方法（这也是不可移植的，但更简单）。

撇开这些限制不说，J2SE Extension Mechanism Architecture对J2EE应用打包也有重要影响。它允许这样一种J2EE打包方法：使用该方法，我们使用多个JAR文件来避免把多个模

块中的相同类定义包含在同一个EAR文件中的需要。这在应用类依赖于自产或第三方库时是特别重要的。例如，我们可以使用包含由多个EJB所需要的库类的JAR文件，而EJB EAR文件只包含应用特有的EJB实现类。请参见<http://www.onjava.com/lpt/a/onjava/2001/06/26/ejb.html>站点，此处有一篇由BEA的Tyler Jewell所撰写的文章，该文章讨论了载货清单类路径的使用。

尤其在EJB JAR文件中，应该使用J2SE 1.3载货清单类路径来避免包含多个模块中的相同类定义。但是请记住，并非所有应用服务器都支持EAR和WAR部署单元中的载货清单类路径。另外还要记住，载货清单类路径只影响一个类定义被保存在什么地方，而不解决哪个类装入器首先装入一个类所引起的问题（例如，在许多应用服务器中由一个EJB类装入器所装入的类看不见WAR文件内的各个类的问题）。

线程上下文类装入器

设法通过使用Java Thread API去程序性地获得一个类装入器来解决类装入问题也是可能的。J2EE 1.3规范的第6.2.4.8节要求所有J2EE容器都支持getContextClassLoader()方法在java.util.Thread上的使用。

该J2EE规范对上下文类装入描述得不是十分清楚。不过，其意图好像是允许可移植类（比如值对象）把应用类装入到它们可能会运行的任何容器（比如EJB或Web容器）中。在实践中，上下文类装入器似乎是在当前容器的上下文中。为了弄清楚这一特性，让我们来看一看由一个助手类所做的下列两个调用的结果，其中这个助手类由EJB类装入器装入，但又同时用在了运行于Web容器上的EJB和类中：

- Class.forName(classname): 将使用助手类的类装入器，在这种情况下是EJB类装入器。这就是说，如果EJB类装入器是WAR类装入器的双亲，该助手将永远不能按名装入WAR中的类。
- Class.forName(classname, true, Thread.currentThread().getContextClassLoader()): 将使用当前容器的类装入器。这就是说，该助手在它正运行的地方将表现不同。如果EJB类装入器是WAR类装入器的双亲，那么当该助手用于EJB容器中时，它将只能装入EJB类和更高的类装入器所装入的类。如果该助手用于WAR中，它还将能装入WAR类。

许多框架（比如WebWork）都使用这种方法来避免由分级类装入所引起的问题。但是，应用代码中通常不需要这种方法，应用代码通常只应该通过使用一个应把上下文类装入器的使用隐藏掉的抽象层来按名装入类。

服务器检查表

由于根本没有两个服务器会以完全相同的方式实现类装入，而且类装入行为甚至会随着同一个服务器的逐次发布而有所变化，所以最后让我们来看一看为了了解自己的应用服务器中的类装入而应该查看的一个事物检查表。

- 要使用的服务器的运行时类分级结构是什么？例如，EJB类装入器是WAR类装入器的双亲吗？

- 服务器支持Web应用的Servlet 2.3型类装入（WAR优先类装入）吗？
- 服务器不仅为EJB JAR文件之类的JAR部署单元，而且还为EAR和WAR部署单元提供载货清单类路径支持吗？假如是这样的话，必须修改服务器配置来允许这种支持吗？
- 服务器的类装入行为是固定的吗？或者说配置类装入行为是可能的吗？例如，在有些服务器中，在标准Java 2与Servlet 2.3类装入行为之间进行选择是可能的。
- 部署在同一个服务器上的不同应用中的类之间的关系（如果有的话）是什么？例如，在使用了一个特殊的平面类装入器结构的JBoss 3.0中，把同一个类的两个不同版本部署在同一个服务器上需要特殊的配置（否则该类的这两个版本将会冲突）。在许多其他服务器中，不同应用将是完全独立的。

建议

虽然不同应用服务器的不同行为使得在涉及打包和类装入的地方执行一成不变的规则变得不可能，但笔者推荐下列指导性准则：

- **只把应用特有的类包含在EJB JAR文件和/WEB-INF/classes目录中**

依赖于可重用基础结构类的EJB JAR文件应该使用载货清单类路径来指明它们对本EAR内其他类的依赖性。

- **只有Web应用所需要的二进制文件应该被包含在/WEB-INF/lib目录中**

在纯Web应用中，简单地把所有必需的JAR文件都包含在这个目录中。

- **EJB和Web应用都使用的二进制文件应该被包含在应用EAR的JAR文件中**

如果应用服务器为WAR部署单元提供载货清单类路径支持，这些二进制文件可以用来声明一种对本EAR中那些JAR文件的依赖性。如果应用服务器不提供这种支持，那些JAR文件还将需要被包含在每个模块的/WEB-INF/lib目录中。这种重复是令人遗憾的，但J2EE 1.3规范暗示它对可移植应用是必需的。

- **仔细考虑按名装入类的各种影响**

要尽力保证按名装入其他类的类只被这些类将要用在的那个模块的类装入器所装入（例如，当一个WAR或EJB JAR部署单元中需要类似功能度时，使用同一个共用超类的不同子类是可能的）。做为一种选择，读者或许可以使用线程上下文类装入器来获得一个要用来装入类的类装入器。通常，这个问题应该被基础结构类所隐藏，因此一般不应该影响应用代码。

- **像JDBC驱动程序和JDO实现之类的服务器级类应该被安装在服务器级上，而不要安装在应用级上**

例如，在JBoss 3.0中，这样的JAR或Zip文件可以被放在当前服务器的/lib目录中；在Orion中，它们可以被放在Orion的/lib目录中。这种方法有时可以用来解决应用特有类装入方面的问题（例如，保证某些应用类在一个应用的所有模块中都是可利用的），但它也是一个粗糙的最后手段。这意味着应用服务器不能动态地重新装入这些类，而且部署单元不再是自主的（这一点与各种J2EE规范相抵触）。

- **如果必要，运行测试来确定应用服务器的类装入器分级结构**

和示例应用包含在一起的基础结构中的com.interface21.beans.ClassLoaderAnalyzer

类，含有显示一个给定类或ClassLoader的类装入器分级结构的方法。笔者发现，在跟踪类装入的问题时，这种诊断十分有用。

在复杂的应用中，设计在应用服务器之间可以移植的打包会十分困难。在某些情况下，可移植打包可能不会增加太大的业务价值，但实现起来可能非常费时间。

在打包应用时，要把重点放在目标应用服务器上。但需要记住上面所讨论的那些问题，尤其是使用了EJB的应用中的分级性类装入器可能产生的影响。

进一步信息

J2EE类装入是一个非常复杂的题目。请参见下列资源来获取进一步的信息：

- <http://kb.atlassian.com/content/atlassian/howto/classloaders.jsp>。Orion服务器的类装入行为的明确而又准确的描述，其中含有对其他资源的参考。
- <http://www.javageeks.com/Papers/ClassForName/index.html>。Java 2类装入行为的出色而又详尽的讨论，其中含有对J2EE的一些（现在已过时的）参考。
- <http://www.theserverside.com/resources/articles/ClassLoading/article.html>。由Brett Peterson所撰写关于应用服务器类装入的文章，其中讨论了WebLogic 6.1、WebSphere 4.0和HP-AS 8中的各种实现。
- 所用服务器的文件资料。这通常是最重要的资源。如果它不十分明确，请运行一个诊断工具，比如包含在配书下载中的com.interface21.beans.ClassLoaderAnalyzer类来显示该服务器的类装入器分级结构。

示例应用的包装

示例应用不是分布式的，而且含有一个Web应用和一个EJB。因此，我们需要创建WAR、EJB JAR和EAR部署单元，而且该应用一般被部署为一个EAR部署单元。我们使用Ant来建立这些部署单元的每一个。

首先，需要了解如何打包该应用所使用的各种通用基础结构类，它们也可能用在其他应用。由于第9章、第11章和第12章中所讨论的基础结构（尤其是各种组件工厂和JDBC抽象包）既用在EJB中，又用在Web构件中，而且需要按名装入类，所以，重要的是，该基础结构的打包不应使应用特有的类装入变复杂。

因此，本章中所讨论的各种框架类被打包在4个独立的JAR文件中，这些JAR文件可以包含在应用部署中，并可以通过使用J2SE 1.3载货清单路径来引用。这个基础结构的实现和打包一定要保证按名的类装入将工作，即便在使用了复杂的类装入器分级结构的应用服务器中。

这些框架类分成下列JAR部署单元，这些JAR部署单元由配书下载中的/framework/build.xml文件来建立，由配书下载的根目录中的示例应用的build.xml文件来调用。如果读者创建供多个应用使用的自定义库包，就应该采用一个类似的多JAR策略：

- i21-core.jar

包含了第11章中所讨论的com.interface21.beans包的核心框架包，第4章中所讨论的门志记录支持和嵌套异常，以及串、JNDI和其他实用工具类。这些类没有一个按名装入其他类，尽管它们中的某些类的子类将按名装入其他类。该JAR文件既用在EJB中，

又用在Web应用中。所有其他JAR文件依赖于这个JAR文件中的各种类。

- i21-web.jar

只有Web应用中所需要的框架包，这些包应该只被包含在WAR文件的/WEB-INF/lib目录中。这个JAR文件包含下列内容：

- 只在Web应用中所使用的组件工厂实现，比如com.interface21.beans.factory.support.XmlBeanFactory。由于这些实现由WAR类装入器装入，所以不保证它们能够看到本WAR文件内所包含的应用类。
- 第11章中所讨论的“应用上下文”基础结构，EJB不需要这个基础结构。
- 第12章中所讨论的MVC Web框架，包括自定义标志和有效性确认基础结构。
- EJB客户类，比如业务委托和服务定位器的超类。
- JMS抽象类。EJB中从不需要这些类，因为我们可以使用MDB并以一种标准方式解决同样的问题。

- i21-ejbimpl.jar

WAR文件不使用EJB超类和JNDI组件工厂实现，所以可以由EJB类装入器来装入。

- i21-jdbc.jar

第9章中所讨论的JDBC抽象层和通用数据存取异常包。这个JAR文件中的类不使用反射，并且都有可能被Web应用和EJB使用。

我们装配一个应用所需要做的只是保证所有必需的JAR文件都是EJB和WAR模块可利用的。编译应用特有类所需要的这些相同Interface21 JAR文件必须都是相关部署单元在运行时可利用的。

EAR文件将把除i21-web.jar之外的所有基础结构JAR文件都包含在根目录中，如下所示：

```
META-INF/
META-INF/MANIFEST.MF
META-INF/application.xml
i21-core.jar
i21-ejbimpl.jar
i21-jdbc.jar
ticket-ejb.jar
ticket.war
log4j-1.2.jar
```

该EJB JAR模块使用一个载货清单类路径，该类路径声明了该EJB JAR模块对我们前面已见过的i21-core.jar、i21-ejbimpl.jar和i21-jdbc.jar的依赖性：

```
Class-Path: log4j-1.2.jar i21-core.jar i21-ejbimpl.jar i21-jdbc.jar
```

正如该WAR文件的下列部分内容清单所显示的，它在自己的/WEB-INF/lib目录中包含了i21-core.jar、i21-web.jar和i21-jdbc.jar：

```
WEB-INF/lib/log4j-1.2.jar
WEB-INF/lib/jstl.jar
Other library classes omitted

WEB-INF/lib/i21-core.jar
WEB-INF/lib/i21-jdbc.jar
WEB-INF/lib/i21-web.jar
```

在Orion或WebLogic这样的服务器（它们里面的EJB类装入器是WAR类装入器的双亲）中，我们只需包含i21-web.jar文件即可。但是，在JBoss（它不使用这种分级性类装入结构）中，所有这些JAR文件都是必需的。如果知道了我们的服务器在WAR文件中支持载货清单类路径，则可以在该WAR文件中简单地声明一个载货清单类路径。

ticket-ejb.jar文件和该WAR文件的/WEB-INF/classes目录只包含应用特有的类。

既然我们已经知道了什么内容应该进入我们的部署单元中，现在就可以编写Ant脚本来建立它们。

Ant提供了建立WAR和EAR部署单元的标准任务。war任务是jar任务的一个扩展，允许我们在以WAR的WEB-INF目录为驻留地的部署描述符、以WEB-INF/lib目录为驻留地的JAR文件和以/WEB-INF/classes目录为驻留地的应用类之间进行轻松的选择。`<war>`元素的webxml属性选择标准部署描述符；`<webinf>`子元素选择要包含在WEB-INF目录中的其他文件，比如专用部署描述符；而`<lib>`和`<classes>`子元素分别选择二进制文件和类。

下面是示例应用中所使用的目录：

```
<target name="war" depends="build-war">

    <war warfile="${web-war.product}"
        webxml="${web-war.dir}/WEB-INF/web.xml">

        <fileset dir="${web-war.dir}" excludes="WEB-INF/**"/>

        <webinf dir="${web-war.dir}/WEB-INF">
            <exclude name="web.xml"/>
        </webinf>
```

下面，我们保证应用特有的类（被编译到由Ant属性`classes.dir`所指定的目录中）进入到/WEB-INF/classes目录中：

```
<classes dir="${classes.dir}">
    <include name="**/*.class"/>
</classes>
```

笔者使用了`<war>`元素的几个`<lib>`子元素来给第13章中所讨论的不同视图技术选择所需的运行时库，如下所示：

```
<lib dir="${lib.dir}/runtime/common" />
<lib dir="${lib.dir}/runtime/jsp-stl" />
<lib dir="${lib.dir}/runtime/velocity" />
<lib dir="${lib.dir}/runtime/xmle" />
<lib dir="${lib.dir}/runtime/itext-pdf" />
```

下面这个子元素把那些Interface21 JAR文件包含在WEB-INF/lib目录中。如果该应用服务器支持WAR文件的载货清单类路径，我们可以忽略这些JAR文件的前两个，并提供一个载货清单来引用WAR的根目录中的这些JAR文件：

```
<lib dir="${dist.dir}">
    <include name="i21-core.jar"/>
    <include name="i21-jdbc.jar"/>
    <include name="i21-web.jar"/>
</lib>
</war>
</target>
```

Ant EAR任务也很容易使用。我们只需简单地指定application.xml部署描述符的位置，以及使用<fileset>子元素指定要被包含的那些档案文件即可。在示例应用的目录布局中，部署描述符在/ear目录中，WAR和EJB JAR文件在/dist目录中。请注意，这项任务依赖于EJB JAR和WAR文件来保持最新：

```
<target name="ear" depends="ejb-jar,war">
    <ear earfile="${app-ear.product}" appxml="ear/application.xml">
        <fileset dir="${dist.dir}">
            <include name="ticket.war"/>
            <include name="ticket-ejb.jar"/>
```

我们把EJB JAR文件的依赖文件包含在WAR文件的根目录中：

```
<include name="i21-core.jar"/>
<include name="i21-ejbimpl.jar"/>
<include name="i21-jdbc.jar"/>
</fileset>
<fileset dir="lib/runtime/common">
    <include name="log4j*.jar"/>
</fileset>
</fileset>
</ear>
</target>
```

可选的Ant任务都没有太大用处，至少根据笔者的经验是如此。同WAR和EAR部署单元不一样，EJB JAR文件是普通JAR文件，其中部署描述符存放在/META-INF目录中。因此，笔者使用了标准jar任务来生成EJB JAR文件。这包括指定已生成JAR文件的/META-INF目录的各种内容（标准ejb-jar.xml和任何专用部署描述符），以及通过把basedir属性设置成含有已编译应用特有EJB类的那个目录的根来选取这些EJB类：

```
<target name="ejb-jar" depends="build-ejb">
    <jar jarfile="${ejb-jar.product}"
        basedir="${ejbclasses.dir}"
        manifest="${ejb-jar.dir}/manifest"
        >
        <metainf dir="${ejb-jar.dir}">
            <include name="*/**/*"/>
        </metainf>
    </jar>
</target>
```

请注意jar任务的manifest属性的使用，该属性指定要用做本JAR文件的载货清单的文件，能使我们指定上面所示的载货清单类路径。

同WAR目标一样，这个EJB目标接受来自部署描述符目录的所有文件，而不是只接受ejb-jar.xml。像jboss.xml或weblogic-ejb-jar.xml之类的专用部署描述符也必须被包含在该部署单元中。

要想在自己的应用中使用本书中所包含的各种库类，需要包含i21-core.jar以及i21-ejbimpl.jar或i21-web.jar，视正在实现EJB还是一个Web模块而定。请注意，i21-web.jar应该被包含在一个/WEB-INF/lib目录中：它不应该由一个EJB类装入器来装入。如果正在使用第9章中所描述的JDBC抽象（可以同时用在EJB和Web应用中），则需要包含i21-jdbc.jar。

所有这些JAR文件都可在配书下载的/dist目录中找到。上面讨论的完整Ant生成文件在配书下载的根目录中被命名为build.xml。

应用部署：常见概念

除了标准打包任务之外，在我们可以进行一个成功的部署之前，还需完成3项不同的任务。

- 需要准备应用服务器，通过配置应用所依赖的服务器级资源，比如连接池来完成。
- 需要编写针对目标应用服务器的专用部署描述符，并把这些描述符与应用打包在一起。
- 在某些服务器中，可能需要执行附加步骤来给目标服务器准备一个部署单元，比如生成并编译EJB主或构件接口实现类。

下面依次来看一看上述每一项任务。

配置服务器来运行应用

现在来讨论一下我们可能会遇到的几个最重要的服务器级配置问题。这些问题涉及到创建本应用在运行时要依赖的各种服务，以及保证满足依赖性。

创建连接池

在大多数J2EE应用中，我们需要执行RDBMS访问。这意味着我们必须配置一个javax.sql.DataSource对象，并保证它被绑定在该服务器的全局JNDI命名上下文中。这通常涉及到创建一个连接池（其细节将是服务器特有的），以及定义一个暴露该连接池的服务器特有DataSource对象。DataSource对象可以由运行于EJB或Web容器内的代码来使用。

在定义一个连接池时，需要设置的典型配置参数包括：

- JDBC驱动程序类。这一般是由RDBMS供应商所供应的一个类，比如oracle.jdbc.driver.OracleDriver。
- JDBC URL。该格式将取决于驱动程序。
- 用户名和密码，假设该容器（而不是该应用）将执行身份鉴别（J2EE应用中最常见的选项）。
- 连接池的最大大小。这对性能可能会产生明显的影响，因为线程在等待连接时将堵塞，如果所有连接在正在使用中。不过，过大的连接池大小可能会浪费宝贵的数据资源，尤其是当多个J2EE服务器访问同一个RDBMS服务器的时候。
- 连接池的初始大小。有时，最好是在启动时分配几条连接，如果我们知道一个应用在启动后将立即频繁地利用该连接。
- 连接池维护设置，比如到不活动连接被关闭时的时间。

我们还需要保证必需的数据库驱动程序在应用服务器级（而且不是在应用级）上是可利用的。

创建JMS目标

JMS目标是服务器级资源的另一个例子。这些资源并不局限于某一个单独的应用。事实上，它们可以用于应用间通信。

在JBoss 3.0中，\$JBOSS_HOME/deploy/jbossmq-destinations-service.xml文件定义默认的JMS题目和队列，设置它们的名称，在Orion中，相同的任务由/config/jms.xml文件来完成。

设置身份鉴别

不同的应用服务器有不同的身份鉴别方法，但我们需要提供对照一个持久性存储器来检查用户凭证的应用服务器方式。这一般涉及指定一个检查用户与角色信息的类，这个类可能由应用服务器提供，也可能是自定义的。大多数应用服务器定义一个必须由这样的类来实现并十分简单的接口。用户信息通常被保存在一个RDBMS中。

在Orion中，身份鉴别可以在应用级上配置，但在WebLogic和JBoss中，身份鉴别是服务器级的。

安装库

我们可能希望把多个应用所使用的二进制文件安装在服务器级上，而不是把它们分布为每个应用的一部分。如果必要，可以通过修改调用应用服务器的那条命令来实现，即通过添加额外的类路径元素，尽管遵守服务器约定会更妥当。在JBoss中，我们可以简单地把必需的.jar或.zip文件复制到相关服务器的/lib目录中。

需要注意的是，当在服务器级上安装库时，不重新启动应用服务器就不可能修改它们；应用服务器将无法用一个动态类装入器装入它们，因为它们可能正用于应用代码。

一般说来，比较好的做法是通过把所有必需的二进制库都放到J2EE部署单元中使该应用保持自给自足。不过，下列是我们可能选择一种服务器级方法的典型情形：

- 当相关的二进制库与服务器而不是与一个具体应用有关系时。数据库驱动程序就属于这个类别。它们通常不能被包含在应用中，因为服务器一般会在启动应用之前先启动连接池，因此在装入应用之前先需要这些驱动程序。
- 为了将被所有应用使用并且可能需要服务器级JNDI绑定的产品，比如JDO实现。
- 当我们需要换掉或修补一个服务器级库，而Servlet 2.3 WAR优先类装入（上文讨论过）又解决不了这个问题时。例如，JBoss 3.0.0捆绑有Log4j。随同一个应用一起包含Log4j的一个较新版本就不再会产生任何影响，因为随着服务器的类装入器一同被装入的这个版本将始终优先。因此，要更新Log4j的这个版本，唯一方法将是在服务器级上换掉它。

不要更新或修补应用服务器所使用的库，除非这么做是必不可少的，因为这么做会有影响应用服务器行为的危险。

有时，把一些类部署在服务器级上来避免WAR与EJB部署之间的类装入问题可能是必不可少的。不过，正如我们已经见过的，这常常体现了这些类的不良设计，而且是一个最后的手段，而不是一个积极的选择。

编写应用的专有部署描述符

除了服务器配置之外，我们还需要添加要包含在部署单元中的专用信息。

我们需要专用部署描述符来提供填补标准部署描述符所遗留下的空隙和配置实现特有的参数的附加信息。例如，专用部署描述符必须用于：

- 告诉服务器去哪里寻找各种标准部署描述符所标识的资源。例如，各种标准部署描述符需要应用代码中所使用的JNDI名称来访问诸如DataSource之类的资源，但不指定这些资源应该被怎样映射到服务器配置期间所创建的服务器级数据源上。
- 指定无状态会话EJB的建池应该怎么进行。无状态会话组件池的大小是一个重要的配置参数，通常在专用部署描述符中被指定。但是，由于它的实现是供应商特有的，所以ejb-jar.xml当然不会描述配置参数。

专用部署描述符的结构是多种多样的，但它们的内容之间存在许多共性。笔者在下面将要讨论的这些设置是笔者所使用过的所有服务器共有的。

使用CMP的实体组件通常需要复杂的专用部署描述符，这些描述符的格式在应用服务器之间是不同的。由于笔者不主张使用实体组件，所以不打算在这里介绍这方面的内容。如果正在使用CMP实体组件，请参考应用服务器的技术资料，了解专用部署描述符。应该计划使用工具来管理这类部署描述符的复杂性；手工编辑对必需的ejb-jar.xml元素和专用描述符来说不是可行的选择。

EJB特有的配置

通常，最复杂的参数都与EJB有关系。我们始终需要为ejb-jar.xml文件中的JNDI名称提供映射。但是，其他设置可能会包括：

- 无状态会话组件的建池选项。
- 有状态会话组件的复制选项。
- 与聚类相关的其他选项，比如一个聚类内的路由算法。
- 在使用具有远程接口的EJB的应用中，同一个JVM内的EJB调用是否应该被优化成使用引用调用（Call by Reference）。这在大多数服务器中是默认设置，并且能极大地改善性能，但也可能会引起意外结果，如果这些EJB被编写成具有按值调用异常。
- CMT的事务超时设置。
- 要给使用CMT的方法使用事务隔离级别。在一些服务器中，只能针对实体组件来设置这些隔离级别；在WebLogic中，只能针对任一类型的EJB来设置它们。

需要注意的是，我们不一定需要替换这些默认设置，这些设置可能会保证大多数应用的行为正确。

Web层配置

Web层配置通常比EJB层配置更简单，因为相关的容器基础结构更简单，web.xml中的空隙更少，以及使用一个MVC框架（比如Struts和我们在第12章中所讨论的框架）往往把Web层构件的更多配置转移到框架特有的部署描述符中。

一般说来，只需为web.xml内的资源引用中所声明的那些JNDI名提供映射即可。不过，我们或许可以设置像下列各项（在weblogic.xml中可配置）这样的选项。

- JSP符合性选项，比如要使用哪个编译器，特殊命令行参数，以及所有JSP页是否应该在应用启动时被预编译；
- 会话跟踪选项；
- 如何存储会话数据（到一个平面文件、RDBMS等）；
- 会话复制配置。

应用的部署

部署一个应用的步骤在服务器之间将是不同的。

除了让WAR文件可让服务器使用外，Web应用一般不需要任何特殊的步骤，因为一个Web容器只需要装入相关服务器小程序类，并使必需的资源可通过JNDI来利用。

但是，EJB部署可能会更复杂，因为作为应用的一部分来实现的EJB类不实现组件的EJBObject和主接口。服务器必须提供这些接口的实现，调用应用类来执行业务逻辑。

有些服务器使用动态代理来实现EJB构件与主接口，封装应用类。但是，许多服务器（比如WebLogic）需要一个编译步骤，在这个步骤中，容器生成并编译必要的类。容器生成式类的使用可能会通过最大限度地消除对反射的需要来产生稍微好一点的性能，尽管这样的增益在大多数应用中不太可能十分显著。直到J2SE 1.3引进动态代理之前，容器生成式类是惟一可用的部署策略。

就WebLogic的情形来说，专用的ejbc工具用来生成容器特有类，并把它们添加到EJB JAR文件上，然后把它们部署到服务器上。WebLogic可以在一个标准EJB JAR文件的部署时自动执行这一步，但是执行这一步可能会不必要地浪费服务器资源，并且可能会使跟踪部署错误变得更为困难，原因是它组合了建立WebLogic特有的EJB JAR文件和把它部署在一个运行服务器上这两项任务。

如果一个服务器特有的“编译”步骤是必需的，应该使用一个Ant目标来调用它，并保证它在部署之前进行。

示例应用的部署参数

至此，我们已经知道了关于部署应用的很多内容，足以列举出配置某个服务器来运行本书的示例应用和添加必要的部署描述符到该应用上所需要的各项关键任务。只是我们完成这些任务的方式在不同的服务器之间是有差别的。

- 为我们的Oracle数据库创建一个服务器级DataSource对象。
- 为引用数据更新消息创建服务器级JMS题目。
- 使该DataSource对象可让BoxOffice EJB通过一个专用部署描述符来利用。
- 使该DataSource对象和JMS题目可让运行在Web容器内的构件通过一个专用部署描述符来利用。
- 使BoxOffice EJB可让运行在Web容器内的代码利用。

在JBoss 3.0上部署示例应用

针对部署示例应用的下列说明从JBoss 3.0的一个“工厂”安装开始。该安装可以通过简单地把发行包`jboss-3.0.0.zip`抽取到一个适当的位置来创建，笔者把这个位置叫做`$JBOSS_HOME`。

这个ZIP文件是带有Jetty Web容器的JBoss发行包。JBoss也与Tomcat Web容器一起捆绑发行，Tomcat是针对J2EE Web技术的Reference Implementation（参考实现）。但是，Jetty似乎提供了优秀的性能（示例应用已利用JBoss/Tomcat进行过测试）。

了解JBoss目录结构

JBoss 3.0的最终发布趋向于WebLogic 6.0及以上版本所使用的多服务器概念。

JBoss安装的根目录含有一个`/server`目录，这个目录又为若干个独立的JBoss服务器配置的每一个含有一个目录。在下列示例中，笔者使用了`/default`服务器，它由默认启动脚本来启动。该服务器的启动目录可以在调用JBoss引导类时被设置为系统属性`jboss.server.name`的值。

由所有服务器使用的那些JBoss启动JAR文件可在`$JBOSS_HOME/lib`目录中找到。

每个服务器目录（比如`$JBOSS_HOME/server/default`）都含有与应用部署相关的下列子目录：

- `/conf`

含有用于本服务器的全局配置信息，比如身份鉴别配置和默认的EJB部署设置。

- `/lib`

含有本服务器所使用的各种二进制文件，包括JBoss和标准J2EE库。这个目录中的JAR和ZIP文件由本服务器自动装入。应用所需要的数据库驱动程序一般被放在这个目录中。

- `/log`

服务器和HTTP请求日志文件。

- `/deploy`

含有由JBoss自动部署的部署单元（EAR、WAR、EJB JAR文件和Resource Adapter），以及服务定义（Service definition）文件。

服务定义文件是XML文件，以JMX MBean定义的形式定义了JBoss服务。任何一个其名称以`-service.xml`结尾的文件（被放在服务器的`/deploy`目录中）都将被当做含有MBean定义。每个文件必须把若干个`<mbean>`元素包含在一个`<server>`元素中。但是，可应用于每个MBean的属性随着相关类而有所不同。MBean类和属性的设置方式类似于我们已在第11章中见过的XML组件工厂定义，只是比XML组件工厂定义更加复杂。同组件工厂方法的情形一样，JMX MBean定义提供了定义不同对象的一种一致方法，意味着学会基本概念来理解它怎么才能用于任一MBean实现是非常必要的。

由于几个MBean（涉及服务器配置的不同方面）可以被包含在同一个服务定义文件中，所以可能在一个单独的XML文件中指定一个应用所需要的所有配置，这使部署变得特别容易。我们在示例应用中使用这种方法。

JBoss没有要求我们定义随MBean一起发送的标准MBean。例如，在JBoss启动时，可以使用相同的机制来添加我们自己的自定义类或第三方类到全局JNDI上下文中。

配置JBoss服务器来运行示例应用

在上面所描述的那些步骤完成之后，我们接着就开始定义示例应用所使用的DataSource对象。

创建连接池

JBoss在\$JBOSS_HOME/docs/examples/jca目录中包含了用于几个数据库的示例定义，包括Oracle。从<http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/jboss/jbosscx/src/etc/example-config/>站点上的JBoss CVS贮藏库中，读者可以获得范围更广的示例。可是，由于这些文件正处于开发之中，所以寻找正确的版本会是一个挑战（请仔细检查那些CVS标志）。

笔者采用了示例目录中的oracle-service.xml文件作为一个基础来定义示例应用的Oracle连接池。

下面是完整的MBean定义。在这段代码中，笔者突出了设置笔者Oracle数据库的URL、驱动程序类、用户名和密码的那儿行数据库连接定义。请根据你的环境中的需要修改它们。

```
<server>
  <mbean
    code="org.jboss.resource.connectionmanager.LocalTxConnectionManager"
    name="jboss.jca:service=LocalTxCM,name=OracleDS">

    <depends optional-attribute-name="ManagedConnectionFactoryName">
      <mbean
        code="org.jboss.resource.connectionmanager.RARDeployment"
        name="jboss.jca:service=LocalTxDS,name=OracleDS">

        <attribute name="JndiName">OracleDS</attribute>

        <attribute name="ManagedConnectionFactoryProperties">
          <properties>
            <config-property name="ConnectionURL" type="java.lang.String">
              jdbc:oracle:thin:@127.0.0.1:1521:rj
            </config-property>
            <config-property name="DriverClass" type="java.lang.String">
              oracle.jdbc.driver.OracleDriver
            </config-property>
            <config-property name="UserName" type="java.lang.String">
              SYSTEM
            </config-property>
            <config-property name="Password" type="java.lang.String">
              MANAGER
            </config-property>
          </properties>
        </attribute>

        <depends optional-attribute-name="OldRarDeployment">
          jboss.jca:service=RARDeployment,
          name=JBoss LocalTransaction JDBC Wrapper
        </depends>
      </mbean>
    </depends>
  </mbean>
</depends>
```

```

<depends optional-attribute-name="ManagedConnectionPool">
    <mbean code="org.jboss.resource.connectionmanager.
        JBossManagedConnectionPool"
        name="jboss.jca:service=LocalTxPool,name=OracleDS">

        <attribute name="MinSize">1</attribute>
        <attribute name="MaxSize">20</attribute>
        <attribute name="BlockingTimeoutMillis">5000</attribute>
        <attribute name="IdleTimeoutMinutes">15</attribute>
        <attribute name="Criteria">ByContainer</attribute>
    </mbean>
</depends>
<depends optional-attribute-name="CachedConnectionManager">
    jboss.jca:service=CachedConnectionManager
</depends>
<attribute name="TransactionManager">java:/TransactionManager</attribute>
<depends>jboss.jca:service=RARDeployer</depends>
</mbean>

</server>

```

需要注意的是，这个示例使用默认的管理登录来登录Oracle。这个登录应该被改成一个具有与示例应用相适应的各种权限的用户账户。

在该连接池工作之前，我们必须通过复制和Oracle包含在一起的classes12.zip文件到JBoss服务器的/lib目录中，使Oracle驱动程序可以被JBoss使用。

创建JMS目标

接着，通过下列MBean定义，创建示例应用所使用的JMS题目。关于JMS MBean定义语法的示例，请参见默认JBoss安装中所包含的jbossmq-destination-service.xml文件。

```

<mbean code="org.jboss.mq.server.jmx.Topic"
    name="jboss.mq.destination:service=Topic,name=data-update">
    <depends optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
    </depends>
    <depends optional-attribute-name="SecurityManager">
        jboss.mq:service=SecurityManager</depends>
    <attribute name="SecurityConf">
        <security>
            <role name="guest" read="true" write="true"/>
            <role name="publisher" read="true" write="true" create="false"/>
            <role name="durpublisher" read="true" write="true" create="true"/>
        </security>
    </attribute>
</mbean>

```

为了获得JBoss启动时的一点点改进，笔者从默认服务器的/deploy目录中删除了jbossmq-destination-service.xml文件，该文件定义了JBoss测试套件和样本JMS应用所使用的JMS目标。

同DataSource的情形一样，JMS题目可以在/deploy目录内的任何一个其名称以-service.xml结尾的文件中。

安装服务定义文件

这两个MBean定义都需要被包含在一个服务定义文件中。笔者把上面所显示的这两个定义放入了同一个文件：ticket-service.xml。这个文件在配书下载的/deploy/jboss目录中，并且

必须在部署应用时被复制到默认服务器的`/deploy`目录中。示例应用的Ant生成脚本中的`jboss`任务自动完成这个复制。该脚本还保证Oracle驱动程序被复制到JBoss服务器的`/lib`目录中。JBoss服务器的位置和部署单元的名称被参数化为Ant属性。这个完整任务如下所示：

```
<target name="jboss" depends="ear">
    <copy todir="${jboss.home}/server/${jboss.server}/lib"
        file="lib/oracle/classes12.zip"/>

    <copy todir="${jboss.home}/server/${jboss.server}/deploy"
        file="deploy/jboss/ticket-service.xml"/>
    <copy todir="${jboss.home}/server/${jboss.server}/deploy"
        file="${app-ear.product}"/>
</target>
```

审查配置

JBoss 3.0使用JMX Reference Implementation的简单Web接口来暴露运行于该服务器上的JMS MBean。通过在端口8082上查看默认页面，如图14.2所示，我们可以看到应用可以使用的DataSource、JNDI题目以及其他全局资源。这在故障诊断阶段是十分宝贵的。



图14.2

JBoss 3.0.1引进了它自己的JMX HTML适配器。WebLogic控制台也是基于JMX的，并且提供一个更高级的接口。无须修改该JBoss服务器的其他任何一个配置即可运行示例应用。

编写示例应用的JBoss部署描述符

由于示例应用没有使用实体组件，只使用了一个会话EJB，所以EJB配置简单明了。

必需的配置被保存于`jboss.xml`专用部署描述符中，而这个专用描述符必须和`ejb-jar.xml`部署描述符一起被包含在EJB JAR文件的/META-INF目录中。`jboss.xml`文件允许我们填补标准部署描述符中的各种空隙，并提供另外的JBoss特有信息。

为了便于参考，下列给出了`ejb-jar.xml`中的完整`<session>`组件元素：

```
<session>
    <ejb-name>BoxOffice</ejb-name>

    <local-home>
        com.wrox.expertj2ee.ticket.boxoffice.ejb.BoxOfficeHome
    </local-home>
    <local>
        com.wrox.expertj2ee.ticket.boxoffice.ejb.BoxOfficeLocal
    </local>
    <ejb-class>
        com.wrox.expertj2ee.ticket.boxoffice.ejb.BoxOfficeEJB
    </ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>

    <env-entry>
        <env-entry-name>beans.dao.class</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>
            com.wrox.expertj2ee.ticket.boxoffice.support.
            jdbc.JBoss30OracleJdbcBoxOfficeDao
        </env-entry-value>
    </env-entry>

    <env-entry>
        <env-entry-name>beans.creditCardProcessor.class</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>
            com.wrox.expertj2ee.ticket.boxoffice.support.
            DummyCreditCardProcessor
        </env-entry-value>
    </env-entry>

    <resource-ref>
        <description/>
        <res-ref-name>jdbc/ticket-ds</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</session>
```

我们必须使用jboss.xml来提供从代码内所使用和ejb-jar.xml内所声明的DataSource JNDI名称（jdbc/ticket-ds）到我们在上面所创建的那个服务器级连接池的JNDI名称（java:/OracleDS）的一个映射。我们还指定该组件的JNDI名称，尽管这是绝对不必要的，因为它默认成<ejb-name>元素的值，而该元素指出这个配置元素适用于ejb-jar.xml内所定义的那个EJB。下面是用于示例应用的完整jboss.xml文件。

```
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>BoxOffice</ejb-name>
            <jndi-name>BoxOffice</jndi-name>

            <resource-ref>
                <res-ref-name>jdbc/ticket-ds</res-ref-name>
                <jndi-name>java:/OracleDS</jndi-name>
            </resource-ref>
        </session>
    </enterprise-beans>
</jboss>
```

`jboss.xml`部署描述符可以用来配置EJB行为的许多方面，其中包括拦截EJB上调用所使用的“拦截器”链。JBoss使用若干个标准拦截器之一来处理服务器管理式功能度的各个方面，比如线程和事务管理。`jboss.xml`部署描述符甚至允许我们指定我们自己的自定义拦截器来执行应用特有的行为，尽管这已远远超出了EJB规范的目前范围，并且不是大多数应用所必需的。

这种可配置性的缺陷是：替换像EJB池大小之类的简单配置参数在JBoss中比在许多服务器中更困难。大多数服务器没有提供这种自定义EJB调用的任一方面的同等能力，但却提供了指定常见设置的一种较简单方法。

每个JBoss服务器的默认EJB设置和其他配置设置都被定义在该服务器的`/conf/standard-jboss.xml`文件中，所以另一个选择是在该文件中以服务器级为条件替换它们。默认的无状态会话组件池大小最大值是100。虽然这个值远远超过了我们的单个组件所需要的大小，但因为它是最大值，而且服务器将没有必要在运行时为许多实例都创建这个最大值，所以替换它是没有必要的，无论在服务器级还是在组件级。

`jboss-web.xml`部署描述符与标准`web.xml`之间的关系和`jboss.xml`部署描述符与标准`web.xml`之间的相同。令人高兴的是，涉及到的配置却简单得多。我们为示例应用所需要做的，只是提供从运行于Web容器内的代码中所使用的JNDI名称（被声明在`web.xml`中）到我们在上述服务器配置步骤中所创建的服务器级对象的JNDI名称之间的同一种映射。那些资源引用被定义在`web.xml`中，如下所示：

```
<resource-ref>
    <res-ref-name>jms/TopicFactory</res-ref-name>
    <res-type>javax.jms.TopicConnectionFactory</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>

<resource-ref>
    <res-ref-name>jms/topic/data-update</res-ref-name>
    <res-type>javax.jms.Topic</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

下面是`jboss-web.xml`的一个完整清单，它显示了这些JNDI名称被怎样映射到我们为数据源和JMS题目所声明的那些JNDI名称上：

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
    <resource-ref>
        <res-ref-name>jdbc/ticket-ds</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <jndi-name>java:/OracleDS</jndi-name>
    </resource-ref>

    <resource-ref>
        <res-ref-name>jms/TopicFactory</res-ref-name>
        <res-type>javax.jms.TopicConnectionFactory</res-type>
        <jndi-name>java:ConnectionFactory</jndi-name>
    </resource-ref>

    <resource-ref>
```

```
<res-ref-name>jms/topic/data-update</res-ref-name>
<res-type>javax.jms.Topic</res-type>
<jndi-name>topic/data-update</jndi-name>
</resource-ref>

</jboss-web>
```

请注意这里用于JMS目标的一个JBoss特有命名约定。我们按照如下所示声明了topic:

```
<mbean code="org.jboss.mq.server.jmx.Topic"
       name="jboss.mq.destination:service=Topic,name=data-update">
```

由这个MBean定义所产生并在上述<jndi-name>元素中所指定的服务器级JNDI名称，是 /topic/data-update。

示例应用的部署

一旦服务器配置完毕，并且我们又拥有了一个打包好的部署单元，JBoss上的部署就非常简单了。我们只需要复制打包好的部署单元到相应JBoss服务器的/deploy目录中，并定义它所需要的服务。由于JBoss使用动态代理来实现EJB主接口与EJB对象接口，所以根本没有必要生成容器特有的类，并且部署非常快。

我们已经见过了示例应用的Ant生成脚本中的jboss目标，这个脚本在重新建立了EAR（如果必要）之后，就把EAR部署单元和必要的服务定义文件复制到JBoss部署目录。

最后，通过从JBoss根下的/bin目录中发布相关命令，我们就可以启动JBoss了：

run.sh

或

run.bat

当JBoss启动的时候，日志输出最初将显示在控制台上。示例应用将会得到自动部署。

小结

本章讨论了如何打包J2EE应用，以及部署应用到应用服务器上所涉及的各种问题。

我们已经知道，当使用EJB时，应用打包中的大多数复杂问题都与类装入有关。J2EE服务器使用多个类装入器。许多应用服务器为EJB类和Web应用类使用各自的类装入器，因而意味着我们可能会遇到EJB JAR和WAR部署描述符中同时使用的类所带来的问题。应用服务器之间在类装入行为方面的差别意味着部署单元不一定总是可移植的。

本章分析了J2SE和J2EE规范所定义的类装入行为，还考虑了几个应用服务器中的类装入行为。

本章讨论了适合应用打包的几种实际选择，最后得出的结论是：最佳方法通常是使用J2SE 1.3载货清单类路径机制来允许EJB JAR文件之类的JAR格式文件去声明与其他库JAR文件的依赖性。本章通过介绍示例应用的打包演示了这种方法，并说明了前几章中所讨论的各种基础结构类怎么才能被打包到应用部署单元所引用的4个JAR文件中。

本章还讨论了加强J2EE部署的基础所需要的服务器配置，比如RDBMS数据源的定义和JMS定义。

本章讨论了我们通常为什么需要服务器特有的专用部署描述符来映射标准部署描述符中所引用的资源到服务器配置中所定义的资源。

最后，本章通过在JBoss 3.0上部署样本应用的过程，讨论了实际的部署。我们学习了怎么才能用Ant生成工具自动化和简化应用打包与部署的所有方面。

J2EE Deployment API Specification (<http://java.sun.com/j2ee/tools/deployment/>) 可能会使应用服务器之间的应用部署变得更一致，但没有解决打包问题，也没有标准化DataSource等资源的定义。

第15章 应用的性能测试与调整

迄今为止，我们已经从理论层次上讨论了很多性能和可缩放性。在本章中，我们将更实际地讨论如何避免引起性能问题的常见原因，以及如何在性能问题对项目造成极大影响之前解决它们。

本章将讨论基本的性能和可缩放性概念，以及性能和可缩放性在整个项目生存周期为什么是决定性的考虑因素。如果直到应用在功能上全部完成之后才重视这些问题（一个常见的失误），我们正在招致危险。

本章将考虑一种满足性能需求的经验方法及其重要性，以及我们怎么才能通过载荷测试和仿形关键应用功能度来收集必要的证据。此外，还将介绍载荷测试Web应用和普通Java类的工具，以及JProbe仿形器（Profiler）。

本章将介绍解决通过测试和仿形所找出的性能问题的技巧。此外，还将考虑如何保证J2EE容器服务的有效使用、高速缓存常用数据的好处（和引起的潜在问题）以及代码级优化。

本章将通过分析和改进示例应用的关键用例之一的性能来举例说明这些技巧，以及本章中所讨论的各种性能测试工具的使用。此外，还将讨论为什么保证分布式应用中的满意性能会是一个特殊的挑战，以及我们怎么才能通过最大限度地降低远程访问方法调用的开销来克服该挑战。

本章还考虑与Web应用有关的一些性能问题。我们将看一看第13章中所讨论的部分可视图技术的测试基准。读者将会看到，我们怎么才能最大限度地减小Web应用上的载荷，即通过使用HTTP首部来允许代理和浏览器进行高速缓存。

最后，本章将讨论J2EE应用中引起性能与可缩放性问题的一些常见原因，这使我们能避免犯高代价的错误。

策略问题与定义

高级设计主要确定一个J2EE应用的效率。代码级优化可以帮助消除已知瓶颈，但很少会产生与努力程度相当的回报。J2EE提供了许多级别上的大量选择。也许，最重要的选择是用不用一个分布式体系结构。如果采用了分布式体系结构，应用划分（Application partitioning），即构件在何处运行的问题将主要决定性能和吞吐量。由于J2EE提供了范围广泛的构件类型，所以在建立应用模型时考虑各种性能影响也是十分重要的。例如，应该使用实体组件，还是使用JDO或JDBC来执行一个特定的数据操作？我们真需要使用提供很多服务但增加开销的EJB吗？

我们必须权衡性能考虑与其他问题。例如，应该使用初始目标数据库的执行性能高但不可移植的特性吗？如果我们的应用服务器没有提供高速缓存服务，应该通过添加我们自己的高速缓存服务，或者使用第三方高速缓存软件来使设计和实现变复杂吗？这样的决策可能

会涉及到权衡。通常，性能考虑元素对决策制定是最关键的，但是好的设计和可维护的代码库依然是最本质的。令人高兴的是，正如我们将要看到的，通常没有必要做出牺牲来实现好的性能：好的设计常常会产生好的性能，并提供实现高速缓存和其他优化所必不可少的灵活性。

需要特别注意的是，令人满意的性能是最主要的业务需求。如果一个应用没有实现足够的性能，无论它有多么精致或使用了多少个设计模式都是无济于事的。它是一个失败者，因为它没有满足业务的各种需求。太多的J2EE开发人员（和太多的J2EE文献）似乎忘了这一点，从而导致了对天生低效设计的不屈追求。

由于一个项目可能会有令人不满意的性能或吞吐量不足的危险，所以我们应该着眼于项目的整个生存周期来考虑性能影响。应该随着开发的进展准备进行性能和载荷测试。例如，我们可能需要在项目生存周期的初期实现一个“纵向程序片”来验证本设计将会产生令人满意的性能，以免受到太严重的困扰。

应该在项目生存周期内尽可能早地通过实现一个“纵向程序片”或“峰值答案”来测试性能和吞吐量。如果这样的测量被留到一个实现在功能上已经完成之后再进行，纠正任何问题可能都需要对设计和代码进行重大改动。

通常，纵向程序片将实现我们怀疑在应用中有最差性能的一些功能度，以及在生产中使用最频繁的一些功能度。但是，我们应该尽力使用证据来支持对可能的性能问题的怀疑。预计问题区域将在什么地方未必总是可能的。

性能与可缩放性

性能和可缩放性是截然不同的。

当谈论一个应用的性能时，我们通常谈的是在一个给定硬件配置上执行它的关键用例所花费的时间。例如，我们可能在这样一台服务器上对一个正运行于单个应用服务器实例中的应用进行的性能测量：这台服务器将被部署在生产中。一个运转正常的应用对它的用户来说应该显得运行速度很快。

但是，性能未必总是能够测度出该应用处理增长负荷的能力，或该应用利用一个更强硬硬件配置的能力。这是可缩放性的问题。可缩放性通常借助于该应用能够同时服务的并发用户数量或事务吞吐量来度量。

可缩放性强调一个给定硬件配置上的可缩放性（例如，一台给定服务器能够同时处理的最大并发用户数量），以及最大可缩放性（在任一硬件部署中能够实现的绝对最大吞吐量）。实现最大可缩放性将需要硬件配置随着负荷的增长而相应地跟着得到增强。通常，一个不断增长的服务器聚类将为可缩放性打下基础。最大可缩放性在实际中通常会受到限制，因为运行一个聚类的开销随着该聚类的大小而增长，这意味着从增加附加服务器中得到的回报将相应地减少。

可缩放性的一个特例与数据容量有关。例如，一个补给系统在它的数据库中可能会有效地支持含有500个作业的用户搜索，覆盖一个城市的一个行业。如果用户搜索含有10万个作业，覆盖几个国家的许多行业，它仍能有效地运转吗？

性能和可缩放性不只是不同的概念，它们在实际中有时是对立的。

性能很差的应用也可能有极差的可缩放性。

一个使服务器CPU满足不了5个用户的需要的应用几乎不可能满足200个用户的需要。

不过，应用在轻负荷下可能是高性能的，但可能会显示出极差的可缩放性。可缩放性是一个能够把应用中的许多缺陷暴露出来的质询者。潜在的问题包括堵塞当前线程的过度同步操作（在轻负荷下可能不十分明显的一个问题，此时该应用可能会显得响应非常快）、资源匮乏、过度内存使用、使数据库服务器发生超载的低效数据库访问以及在聚类中有效运行的无力。不能缩放以满足要求将会有严重的后果，比如不断变长的响应时间和不可靠性。

性能和可缩放性都可能是重要的业务需求。例如，就一个Web站点的情况来说，极差的性能可能会驱使用户不再访问这个站点。如果这个站点在吸引和保住用户方面确实获得成功，极差的可缩放性最终将会证明是致命的。

作为应用设计师和开发人员，采用J2EE不允许我们假定性能和可缩放性需求已由BEA或IBM的专家替我们考虑好了。如果做了这样假定，我们将势必受到惩罚。虽然J2EE基础结构（如果使用正确）能够帮助我们实现可缩放的和高性能的应用，但是它也增加了开销，而这些开销会严重影响被自然实现（没有使用J2EE基础结构）的应用的性能。为了保证我们的应用是高效率的和可缩放的，我们必须对J2EE应用服务器的开销和实现特征有相当的了解。

设立性能与可缩放性的明确目标

弄清楚我们需要实现什么才能保证好的性能和可缩放性是十分重要的。

可缩放性可能是最本质的。如果一个应用不能满足用户群的各种要求——或用户数量的增长，那么它将被认为是失败的。

但是，性能可能会更多地涉及到权衡利弊。性能可能需要对照可扩展性、可移植性、可能的维护成本等考虑因素来加以权衡。我们甚至可能需要在性能方面进行取舍。例如，有些用例对应用的外观可能是至关重要的，而其他用例是不太重要的。这可能会对应用的设计产生影响。

很明显，这样的取舍应该事先就做好，因为它们决定着一个应用应该试图满足其业务需求的方式。

掌握应用的具体性能与可缩放性需求，要比获得应用应该“运行速度快且应付许多用户”这种含糊不清的概念重要得多。非功能性需求应该含有针对性能和吞吐量的目标，并且这些目标应该明确得足以能让测试去证实它们在项目生存周期的任何时刻都能得到满足。

设计与代码优化的比较

J2EE应用中的主要设计问题对性能产生的影响可能会远远超过任何代码级的优化。例如，在分布式应用的一个时间苛刻式操作中，做两个远程调用而不是单个远程调用由于设计不良而对性能产生的影响，可能会远远大于做任何数量的代码级优化所产生的影响。

最大限度地消除对代码级优化的需要是十分重要的，原因如下：

- 优化很困难；
- 大多数优化无意义；
- 优化导致许多隐错；

- 不当的优化可能会永久地损及可维护性。

下面依次来考虑上述每一点，因为它们都适用于J2EE应用。

程序设计中很少有事情会比优化现有代码更困难。令人遗憾的是，这也是优化为什么满足任何程序员的利己主义的惟一原因。问题是专用于这种优化的资源很可能会被白白浪费掉。

大多数程序员都熟悉80/20定律（或类似定律），这个定律认为：执行时间的80%被花费在一个应用的20%的代码库上（这也叫做巴累托定律（Pareto Principle））。这个比例随着应用的不同而有所不同——90/10可能更常见，但这条定律是根据经验得出的。这意味着优化一个应用代码库的80%~90%错误代码是浪费时间和资源。

找出应用中的瓶颈并解决具体问题是可能的（最好使用工具，而不是凭感觉来寻找；我们将在下文中讨论那些必要的工具）。优化没有证明有问题的代码将会对好代码造成更大的危害。在J2EE上下文中，优化可能是特别没有意义的，因为瓶颈可能是在由于应用设计不良而导致被频繁地调用的应用服务器代码中。需要特别注意的是，J2EE应用运行时执行的许多代码是应用服务器的一部分，而不是应用的一部分。

优化是引起隐错的一个常见原因。传奇人物Donald Knuth（“The Art of Computer Programming”一书的作者）曾经说过，“我们应该忽视极小的效率，比如说在97%左右的时候：不成熟的优化是所有罪恶的根源”。如果优化使代码变得更难理解（而且许多优化都会如此），该优化就是一件令人十分烦恼的事。优化会直接造成隐错，此时有些曾经工作得很慢的东西只在某些时候才工作，但速度更快。它对质量所造成的影响是更阴险的：难以理解的代码容易出问题，并且将会在应用维护期间不断地耗用过度的资源。

性能与可维护性之间的权衡取舍在J2EE应用中是决定性的，因为企业级应用往往关系到企业的生死存亡，而且该权衡取舍也关系到企业的后续投入。设计性能与可维护性之间不存在冲突，但优化可能会更有问题。万一发生了冲突，牺牲性能来换取可维护性可能是正确的。我们选用Java而不是C或汇编语言编写软件的事实就是这方面的明证。由于可维护性在一个软件项目的成本中占有绝大部分，所以简单地购买速度更快的硬件可能会证明比从应用中挤压出少量的性能更便宜。如果这使维护变得更困难的话。当然，也存在性能重于其他考虑因素的情形。

应该通过利用优良的设计去防止问题，从而最大限度地消除对优化的需要。不可避免地优化和选择优化什么应该基于仿形器结果之类的有力证据。

一些论述优化的权威著作建议：首先编写一个简单的程序，然后优化它。这种方法通常只在J2EE中的构件内，并且构件接口已经得到建立之后才会令人满意地工作。先不必考虑性能就提供接口的一个快速实现，并在以后需要的时候再优化它通常是一种上佳的方法。如果试图把这种“以后优化”方法运用于整个J2EE开发，我们势必会很失望；在涉及到分布式J2EE应用的地方，我们正在自取灭亡。我们可以在J2EE应用中实现显著性能改善的惟一方面是高速缓存。可是，正如我们将看到的，高速缓存只有在应用的设计非常完善时才能被轻松地增加上。

性能与吞吐量的测试工具

为了在开发过程中的任何阶段测试应用的性能和吞吐量，我们需要使用适当的工具，并

有一个明确的测试计划。至关重要的是测试应是可重复的，测试结果应被归档以便将来参考。

通常，我们首先从载荷测试开始，载荷测试会以响应时间的形式给出每个并发测试的性能。可是，我们可能还需要仿形个别请求，以便能够优化或消除速度慢的操作。

基准测试的准备

基准测试是一种试验形式，所以采用一种有效的试验方法来保证基准测试尽可能地准确和可重复是至关重要的。例如，由于该应用将要用在生产中，它必须被配置成下面这样：

- 该应用应该正运行在生产用硬件上或最接近于生产用硬件的现有硬件上。
- 日志记录必须被配置成同生产中一样。罗嗦的日志记录（比如在Java 1.4 FINE级别上或在Log4j DEBUG级别上）会严重影响性能，尤其是当日志记录导致如下一类日志消息生成的时候：这些消息只有当它们被显示时才被生成。日志输出格式也是很重要的。例如，Log4j允许我们显示所有产生了输出的记录语句的行数。这对调试是极其有用的，但代价如此之高（由于需要生成异常和分析它们的栈跟踪），以致严重地扭曲了性能结果。
- 配置第三方产品以便得到最佳性能，因为它们将被部署在生产中。例如：
 - MVC Web框架可能有会降低性能的调试设置。要保证这些设置已被禁用。
 - Velocity模板引擎可以被配置成定期地检查模板中的修改。这在开发中是很方便的，但会降低性能。
 - 应用服务器应该被配置成生产用设置。
 - RDBMS应该被设置成生产用设置。
- 禁用在某个特定生产环境中不使用但可能会影响性能的那种应用特性。
- 使用实际数据。如果一个系统只被加载了用于数百个用户的数据，那么它的性能可能会和它被加载了生产中所使用的成千上万条记录时的性能有极大的差别。

保证不再存在可能会对测试的运行产生影响的讨厌因素也是十分重要的。

- 当在运行该J2EE应用的同一个服务器上运行载荷测试或仿形软件时，要检查它没有通过独占CPU时间来扭曲性能数字。如果可能，要从一个或多个独立的机器中载荷测试一个Web接口。
- 要保证运行该应用服务器的那台（那些）机器上根本不可能会减少该应用可以利用的资源的其他进程。甚至连top那样的无害监视工具也会占用惊人的CPU时间。病毒扫描和类似操作在费时很长的测试运行中会是灾难性的。
- 要保证运行载荷测试软件的应用服务器和机器上有足够的RAM可供使用。

需要特别注意的是，基准测试不是一门精确的科学。要努力消除令人讨厌的因素，但是请记住，不要给一个特别的数字加上太多的意思。根据笔者的经验，由于用在生产中的变量的个数，在对J2EE应用的连续测试之间见到20%至30%的偏差是常见的事情，尤其是在涉及到载荷测试的情况下。

Web测试工具

要想确定一个J2EE Web应用在轻负载下是否执行得令人满意并提供充足的吞吐量，最

容易的方式之一是载荷测试它的Web接口。由于用户对这个J2EE Web应用的体验将是通过该Web接口获得的，所以非功能性的需求应该提供所需性能和并发性的一个明确定义。

Microsoft Web Application Stress工具

有许多可以用于测试Web应用性能的工具。笔者首选的工具是免费的Microsoft Web Application Stress (WAS) 工具 (<http://webtool.rte.microsoft.com/>)。

作为一个平台独立的、基于Java的替代工具，可以考虑使用Jmeter工具（可以从<http://jakarta.apache.org/jmeter/index.html>站点上获得）。不过，这些工具不太直观，而且设置起来较困难。一般说来，最好是在一台独立于应用服务器的机器上运行载荷测试软件，因此寻找一台Windows机器来运行该Microsoft WAS工具通常不会有问题。

配置Microsoft WAS工具非常容易，只涉及创建一个或多个脚本。脚本也可以使用Internet Explorer来“记录”。脚本由一个或多个待载荷测试的应用URL所组成，包括GET或POST数据（如果必要）。WAS可以使用一组或一系列参数值。图15.1所示的屏幕图举例说明了如何配置WAS来请求示例应用中的“Show Reservation”页面。请记住，如果必要，修改每个URL的默认端口（默认设置是80）。在本例中，它是JBoss/Jetty默认端口8080。



图15.1

每个脚本都含有用于如下配置的全局设置：要使用的并发线程的数量，每个线程所发布的请求之间的延迟，以及像是否遵守重定向及是否模拟用户通过一条慢速调制解调器链接进行访问之类的选项。配置Cookie和会话行为也是可能的。每个脚本都通过图15.2所示屏幕进行配置。

一旦运行运行一个脚本，就可以借助于View菜单上的Reports选项来查看报告。报告被存储在一个数据库中，以便它们能被随时查看。报告包括每秒钟的请求数量、收发的数据量以及接收响应的第一个与最后一个字节的平均等待时间。如图15.3所示。

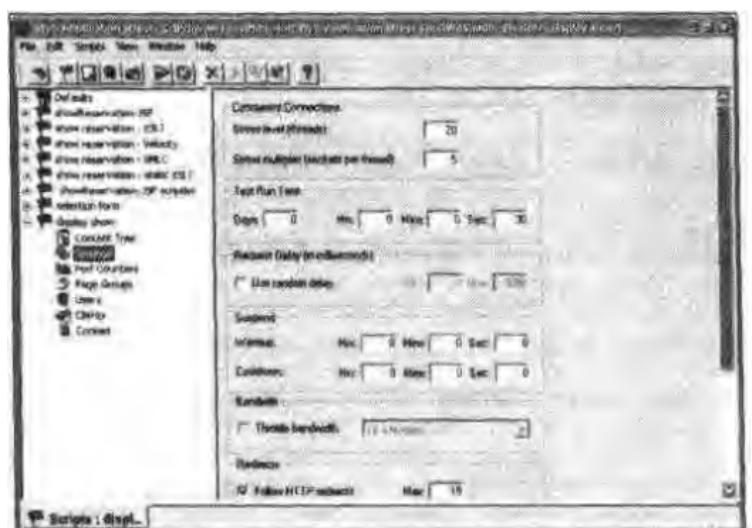


图15.2



图15.3

使用Web Application Stress工具或类似产品，我们可以快速地确立整个Web应用的性能和吞吐量，指出哪里可能需要进一步的、更详细的分析，以及是否需要全面的性能调整。

非Web测试工具

有时，通过Web接口的测试就是我们需要做的惟一测试。性能和载荷测试和单元测试不同；没有必要为每个类都做性能测试。如果我们能够轻松地确立整个系统的一个性能测试，并对测试结果感到满意，就没有必要花费更多的时间来编写性能或可缩放性测试。

但是，并非所有J2EE应用都有一个Web接口（即使在Web应用中，我们仍可能需要较详尽地细分应用将其大部分时间所花费在的那些体系结构层）。这意味着我们需要载荷测试和性能测试个别Java类的能力，而这两种测试又可能测试像数据库那样的应用资源。

有许多可用于这类测试的开放源工具，比如Apache Jmeter和Grinder（可以从`http://sourceforge.net/projects/grinder`站点上获得），后者是最初为一本关于WebLogic的Wrox图书所开发的一个可扩展载荷测试程序。

就个人而论，笔者发现这些工具中的大多数都复杂得有点多余。例如，为Grinder工具编写测试案例就不是一件容易的事情，该工具要求能够多点广播，以便支持载荷测试进程与其控制器之间的通信。和JUnit不同，Grinder工具有一定的学习难度。

笔者使用了下面这个简单的框架，这个框架最初是笔者在两年前为一个客户开发的，后来笔者发现这个框架几乎能毫不费力地满足编写载荷测试的所有需求。这个代码和示例应用下载包含在一起，位于`/framework/test`目录下。和Jmeter或Grinder不同，该工具没有提供一个GUI控制台。笔者曾经为该工具的一个早期版本编写过一个GUI控制台，但发现它不像文件报告那么有用，因为那些测试常常在没有显示器的服务器上执行。

和大多数载荷测试工具一样，`com.interface21.load`包中的上述测试框架基于下列概念：

- 它能使若干个测试线程（Test thread）作为一个测试序列（Test suite）的一部分并行地运行。在这个框架中，测试线程将实现`com.interface21.load.Test`接口，而测试序列通常是一个通用框架类。
- 每个测试线程都执行与其它线程的活动无关的若干遍扫描（Pass）。
- 每个测试线程在测试扫描之间都能使用一个最大长达若干毫秒的随机延迟。这是模拟运行时可能要经历的意外用户活动所不可或缺的。
- 每个测试线程都实现一个简单接口，并且该接口要求它为每遍扫描都执行一个应用特有的测试（测试扫描之间的随机延迟由框架测试序列来处理）。
- 所有测试线程都使用一个统一的测试夹具（fixture）来暴露要测试的应用对象（这类似于一个JUnit夹具）。

周期性的报告被送到控制台，而在一个测试运行完毕之后，一份报告可以被写到文件上。

图15.4所示的UML类图举例说明了涉及到的各框架类，以及一个应用特有的测试线程类（已用圆圈标出）怎么才能扩展`AbstractTest`便利类。该框架提供了一个测试序列实现，该实现提供了协调所有应用特有测试的一种标准方法。

实现一个载荷测试所需要的唯一代码是`AbstractTest`框架类的一个扩展，如下所示。这个扩展只涉及实现两个方法，因为`AbstractTest`类提供了`java.lang.Runnable`接口的一个final实现：

```
import com.interface21.load.AbstractTest;
public class MyTestThread extends AbstractTest {
    private MyFixture fixture;
```

该框架在`AbstractTest`类的子类上调用下列方法，以创建每个测试线程都可以使用的共享测试夹具——要测试的应用对象。不需要夹具的测试不必替换这个方法：

```
public void setFixture(Object fixture) {
    this.fixture = (MyFixture) fixture;
}
```

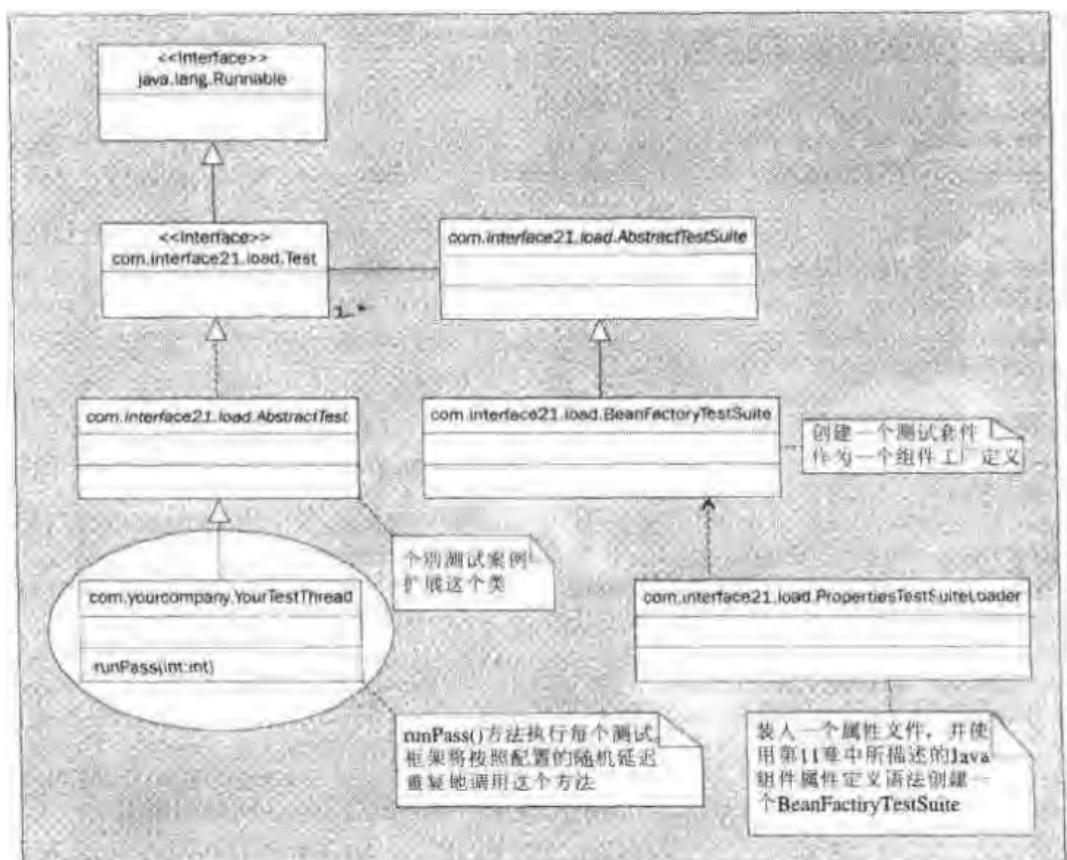


图15.4

下面这个抽象方法必须被实现，以运行每个测试。在必要情况下，测试扫描的索引作为一个变元来传递：

```

protected void runPass(int i) throws exception {
    // do something with fixture
}
  
```

一般说来，`runPass()`方法将被实现，以便选择由夹具使之变得可用的随机测试数据，并使用这些数据在正被测试的那个类上调用一个或多个方法。和JUnit测试案例的情形一样，我们只需要捕捉由正常执行情景所产生的异常：未捕捉异常将由测试序列记录为错误，并被包含在最终报告中（测试线程将继续运行另外的测试）。如果断言未得到满足，异常也可以被抛出，以指出发生了故障。

这个工具使用了第11章中所描述的，而且也是本书所讨论的应用框架中一贯使用的那种基于组件的配置方法。每个测试都使用各自的属性文件，使参数化可能轻松实现。这个文件由`PropertiesTestSuiteLoader`类读取，这个类将把此文件名作为一个变元来接受，并从该属性文件所包含的组件定义中创建和初始化一个`BeanFactoryTestSuite`型的测试序列对象。

下列这些定义配置该测试序列，其中包括它的报告格式，它在测试运行期间多久向控制台报告一次，以及它把它的报告文件写在什么地方。如果`reportFile`组件属性未得到设置，那么不存在文件输出。

```

suite.class=com.interface21.load.BeanFactoryTestSuite
suite.name=Availability check
suite.reportIntervalSeconds=10
suite.longReports=false
suite.doubleFormat=###.#
suite.reportFile=c:\\reports\\results1.txt

```

下列关键字控制有多少线程得到运行，每个线程运行多少遍扫描或多少个测试案例，以及每个测试线程中测试案例之间的最大延迟是多长时间（以毫秒为单位）：

```

suite.threads=50
suite.passes=40
suite.maxPause=23

```

下列这些属性展现了怎样处理一个应用特有的夹具使之可供测试序列使用，以及怎么才能通过它的Java组件属性来配置。测试序列将在每个测试线程上调用`setFixture()`方法，以使所有测试线程实例都能共享这个夹具：

```

suite.fixture(ref)=fixture
fixture.class=com.interface21.load.AvailabilityFixture
fixture.timeout=10
fixture.minDelay=60
fixture.maxDelay=120

```

最后，我们必须包括一个或多个测试线程的组件定义。这些定义的每一个在运行时都将是独立的。由于这个缘故，这个组件定义不能是一个单元集，因而我们在突出显示的行中替换组件工厂的默认行为：

```

availabilityTest.class=com.interface21.load.AvailabilityCheckTest
availabilityTest.singleton=false

```

默认行为是让每个线程都从测试序列中取出它的扫描遍数和最大暂停值，但可以针对每个测试线程来替换这个默认行为。并发地运行几个不同的测试线程也是可能的，其中每个线程具有一个不同的权重。

可以像下面这样使用一个Ant目标来运行测试序列：

```

<target name="load">
  <java
    classname="com.interface21.load.PropertiesTestSuiteLoader"
    fork="yes"
    dir="src">
    <classpath location="classpath"/>

    <arg file="path/mytest.properties"/>
  </java>
</target>

```

对突出显示的这一行按需求进行了改变，以保证在类路径上可获得`com.interface21.load`包和应用特有的测试夹具与测试线程。

报告将显示已完成的测试运行数量、错误个数、每个测试线程和全部测试线程每秒钟所达到的命中次数以及平均响应时间：

```
AvailabilityCheckTest-0 40/40 errs=0 125hps avg=8ms
AvailabilityCheckTest-1 40/40 errs=0 95hps avg=10ms
AvailabilityCheckTest-2 40/40 errs=0 90.7hps avg=11ms
AvailabilityCheckTest-3 40/40 errs=0 99.8hps avg=10ms
AvailabilityCheckTest-4 40/40 errs=0 110hps avg=9ms
***** Total hits=200
***** HPS=521.3
***** Average response=9
```

最重要的设置是测试线程的数量。通过增加这个数量，我们可以确立吞吐量在哪一点上开始变糟，该点通常是做本练习的要点。现代JMV能够应付非常大的并行线程数量；笔者曾经利用数百个并发线程做过成功的测试。不过，需要特别注意的是，如果我们运行太多的并发测试线程，在测试线程之间切换执行的操作量可能会变得非常大，足以扭曲最后结果。通过指定每个线程要执行的一个非常大的扫描遍数，使用这个工具确定应用如何处理长时间负载也是可能的。

通过提供一个请求Web资源的AbstractTest实现，这个工具也可以用于Web测试。Web Application Stress工具更容易使用，并且提供大多数情况所需要的所有灵活性。

查找性能或可缩放性问题

只有当确信存在一个性能问题时——由基准测试一个纵向程序片或整个应用而引起，我们才应该使用专门的资源来跟踪问题区域。

需要重点注意的是，寻找问题的过程是以经验作为基础的。虽然经验和知识可以帮助培养强烈的直觉，但没有什么东西能够比真凭实据更过硬。

下面来看两个基本技巧，我们可以利用这两个技巧来确定一个已知的性能或可缩放性问题处于J2EE栈内的什么地方。

- 尽量独立地测试每个体系结构层的性能，或者只测试较低的体系结构层。
- 使用一个仿形工具测试整个应用。

层中的测试

优良的设计应该允许我们独立地测试一个J2EE应用的每个体系结构层。优良的设计将保证该应用被划分成具有不同责任的不同层。这样的分层测试在有些情况下是不可能的，比如，假如JSP视图执行数据库访问。

分层测试允许我们执行下面这样的测试（从一个典型应用的最低层开始）：

- 使用数据库工具测试RDBMS查询与更新。这应该包括测试各事务中并发活动的行为。
- 通过利用一个测试数据源在一个J2EE应用服务器外部测试Data-Access Objects来测试JDBC操作。
- 测试业务对象性能。可以独立地测试业务对象（使用虚构对象来代替用于连接数据库等资源的构件），也可以利用真正的应用后端测试它们。
- 使用立即响应测试数据的虚构业务对象来测试Web层性能，使我们能够单独看到Web层性能。

在涉及到EJB的情况下，这样的测试会比较困难；可能需要在应用服务器内运行测试。

这种分层方法在涉及到载荷测试的情况下特别有用，因为它使得测试任何一个层在轻负载下如何表现变得非常容易。

仿形工具

我们也可以使用工具，而不是通过我们自己的努力来确定应用把它的时间都花费在了什么地方。虽然这种方法替代不了“分层测试”方法——尤其是在涉及到吞吐量的情况下，但它能够提供关于瓶颈的真凭实据来取代人的直觉，从而有可能避免浪费精力。仿形还可以通过显示调用栈来解释一个应用的运行时行为。

但是，使用仿形存在一些问题。

- 仿形会严重降低性能，有时会损及稳定性。因此，我们也许无法对运行在一个仿形器之下的应用执行载荷测试，而且可能只限于查看个别请求的性能。于是，仿形在跟踪锁定之类的可缩放性问题方面没有太大帮助（但其他专业化工具在这方面或许会有帮助）。
- 仿形可能需要将JVM设置改成不同于生产中所使用的那些设置，因为生产设置有时会产生不规则的结果。例如，我们可能需要禁用经过优化的HotSpot JVM。

JVM仿形选项

大多数JVM都提供了一个仿形选项。例如，利用下列变元启动一个Sun Java 1.3 JVM会导致它在退出时转储仿形输出结果：

```
-Xrunhprof:cpu=samples,thread=y
```

最有用的信息被包含在java.hprof.txt生成文件的最后一节中，这一节按照方法所花费的执行时间量为顺序来显示这些方法。它还显示每个方法的调用次数。

rank	self	accum	count	trace	method
1	26.67%	26.67%	12	57	java.net.SocketInputStream.socketRead
2	11.11%	37.78%	5	107	java.net.SocketInputStream.socketRead
3	4.44%	42.22%	2	83	oracle.jdbc.driver.OracleConnection.privateCreateStatement
4	4.44%	46.67%	2	32	java.io.Win32FileSystem.getBooleanAttributes
5	2.22%	48.89%	1	114	java.net.PlainSocketImpl.socketClose
6	2.22%	51.11%	1	130	java.lang.System.currentTimeMillis
7	2.22%	53.33%	1	43	java.net.InetAddressImpl.getLocalHostName
8	2.22%	55.56%	1	34	oracle.jdbc.ttc7.TTC7Protocol.connect
9	2.22%	57.78%	1	17	com.interface21.beans.BeanWrapperImpl.setPropertyValues
10	2.22%	60.00%	1	122	java.lang.ClassLoader.findBootstrapClass
11	2.22%	62.22%	1	87	java.lang.Class.forName0
12	2.22%	64.44%	1	79	oracle.security.o3logon.O3LoginClientHelper.<init>
13	2.22%	66.67%	1	47	java.lang.Throwable.fillInStackTrace
14	2.22%	68.89%	1	77	oracle.jdbc.ttc7.TTC7Protocol.logon

有几个免费工具可以用来查看仿形器输出文件，并抽取关于调用栈的信息。

过去，笔者曾经成功地使用过JVM仿形，但它是一个非常基本的解决方案。它缓慢地显示应用，并且会造成频繁的系统性事故。例如，当在Windows操作系统上启用了仿形选项之后，JDK 1.3.1_02似乎就有JBoss 3.0.0启动方面的问题，并且笔者以前让JDK 1.2仿形在Solaris操作系统上工作时也遇到过严重问题。

如果读者确实需要仿形，就可能需要某个更高级的东西。笔者没有听说过任何真正令人满意的免费或开放源产品；假如性能问题对项目很重要，花费一点投资购买一个商业产品是完全值得的。

JProbe仿形器

Sitraka的JProbe (<http://www.sitraka.com/software/jprobe/>) 是最著名的商业仿形器，而且主导市场达若干年时间之久。JProbe不是惟一选择，但它非常优，价格也不是很高。下载一个免费试用10天的评估拷贝是可能的，在这10天时间内足以确定它是不是项目的一笔明智的投入。JProbe套件含有几个产品。笔者在这里将谈一谈这个仿形器。

本节的焦点将放在JProbe 4.0上，但这里所讨论的大多数概念将适用于任一仿形器。

试用JProbe首先需要配置JProbe，以便利用必要的检测仪表运行应用。对于服务器端应用来说，JProbe眼中的“应用”是运行用户应用的J2EE应用服务器。当该应用正在运行时，JProbe可以拍摄“性能快照”用于立即或脱机分析。

集成JProbe与应用服务器

JProbe 4.0发行包带有用来把它与大多数商业应用服务器集成起来的设置。但是，笔者不得不手工地把它与JBoss 3.0集成在一起，这项工作花费了笔者不少时间。

JProbe安装过程通过“JProbe Application Server Integration”对话框提示用户把JProbe与应用服务器集成在一起。得到支持的任何服务器都可以从左边的下拉菜单中选取。要想与一个新型的服务器集成在一起，比如JBoss 3.0，按照以下步骤进行操作：

1. 从下拉菜单中选择 Other Server。

2. 单击Create按钮。

3. 自定义右窗格中的设置，其中包括设置Advanced Options。这包括把IntegrationID设置成一个有含义的名称（比如JBoss）。

右窗格中最重要的设置包括：

- 服务器目录。这通常是该服务器的主目录。
- 服务器类。含有该服务器的main()方法的Java类。
- 服务器类路径。JBoss使用一个最小的“引导”类路径，并从它的目录结构中进一步装入二进制库，所以这个设置配置起来非常简单。
- 要使用的JVM可执行文件。
- JVM变元。要想实现图15.5中所显示的那些“Java选项”，需要进行一些试验。那些Java选项基于随同JProbe一起提供的WebLogic集成参数。如果没有设置堆大小，启动过程会很慢，而且JBoss在启动时经常会中止。

上述大多数设置的正确设置值，可以在默认的JBoss启动脚本\$JBoss_HOME/bin/run.sh或\$JBoss_HOME/bin/run.bat中找到。究竟在哪个脚本中可以找到取决于读者所使用的操作系统。图15.5所示的屏幕图是笔者为JBoss 3.0所使用的各种JProbe集成设置。

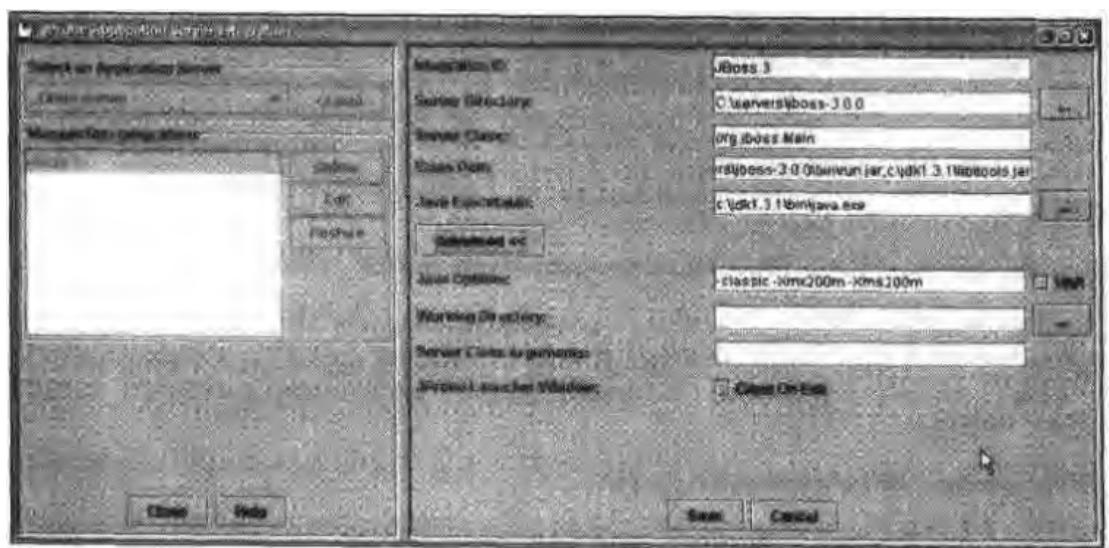


图15.5

同一个JProbe安装可以为多个J2EE服务器使用相同的集成设置。

设置筛选器

设置每个JProbe分析“会话”的配置也是不可或缺的。这决定使用哪个应用服务器集成定义，该仿形器将寻找什么，以及仿形器数据将被怎样收集和查看。可以借助于JProbe主窗口上的“Open Settings”选项来定义设置。可以定义多个设置，以便能够为某个具体的仿形运行选择最恰当的定义。

最重要的设置是要筛选的各种包，这些包显示在了对话框的中间部分。通常，我们不需要关于应用服务器的性能信息，也不需要标准Java或J2EE包的性能信息（尽管这些信息在我们怀疑某个性能问题位于我们自己的代码之外时可能是十分有用的）。因此，我们一般通过筛选把仿形信息限定在我们自己的类上。通过指定包前缀和从右列中为每个前缀选择一个Detail Level来激活筛选。需要注意的是，与其名称以给定串开头的所有包相关联的信息都将被收集起来：例如，像图15.6屏幕图一样指定com.wrox作为包前缀将会产生示例应用中的所有包的仿形数据集。

笔者发现撤选Program Start中的Record Performance选项会十分有用。通常，我们不希望因为仿形应用服务器而减慢了应用服务器的启动速度。图15.6的屏幕图显示了笔者为本章中的那些仿形运行所使用的会话定义。

在示例应用用例上运行JProbe

现在，随时可以启动一个仿形运行。首先，需要启动运行在JProbe之下的JBoss。为此，在JProbe Profiler主窗口上选择Program菜单的Run选项，如图15.7所示。

假设示例应用已被部署在JBoss上，那么当JBoss启动时，它将照常启动。需要注意的是，JBoss用来启动和部署示例应用的时间将会比平常要长得多，因此需要有一点耐心（或一台运行速度更快的机器）。要监视由JBoss与应用所生成的日志文件及CPU使用率来判断示例应用何时启动完毕。此刻正是煮咖啡或买咖啡的好时候。

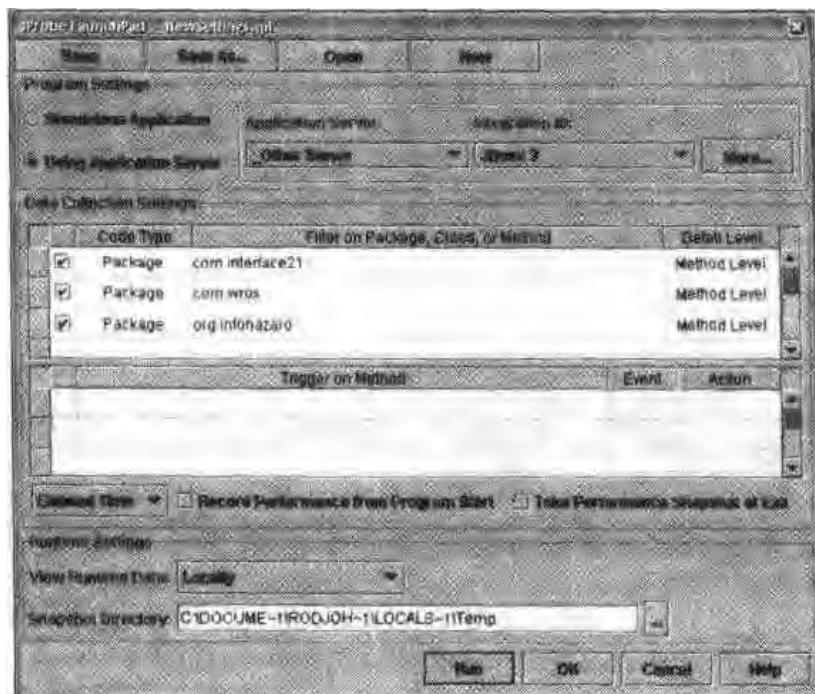


图15.6

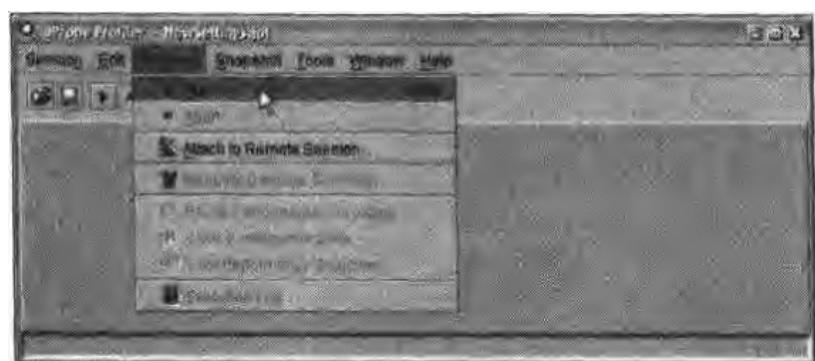


图15.7

现在，该是开始记录性能数据的时候了。为此，从Program菜单中选择Resume Performance Recording选项，如图15.8所示。

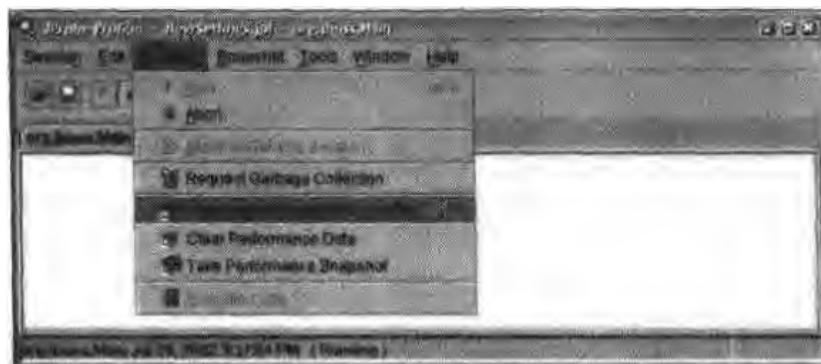


图15.8

现在，执行我们希望仿形的应用用例。同样，这将会花费比平常更长的时间。当该用例完成时，从Program菜单中选择Take Performance Snapshot选项。一个新的快照图标将会出现在JProbe主窗口中：默认名称是snapshot_1，但我们可以保存该快照时对它重命名。保存它可以供将来使用，与以后是否运行该仿形器来收集新数据无关。

右击任一快照，并从出现的弹出菜单中选择Open in Call Graph选项，将会看到一个如图15.9所示的窗口，其中显示了按累积执行时间（方法及其子树所占用的执行时间）进行降序排列的方法，如下图中所示。

图15-9

图15.9所示的屏幕图显示了相同的数据，只是按方法时间（花费在每个方法内的时间）进行了降序排列。

org.jboss.undeploy.400DeleteJobs.java LocalPrepareGatheringJob	2	02	228,847	22	125,351	499	{15,461}	496	{15,461}
java.lang.reflect.Method.invoke(Object, Object)	2	02	113,301	2	15,311	1,395	{15,311}	16	{15,311}
org.jboss.redeploy.AdaptorJobs.java LocalPrepareGatheringJob	2	02	14,743	2	16,751	251	{19,361}	252	{19,361}
java.lang.StringBuilder.append(String)	24	02	(1,791)	2	16,752	298	{12,792}	301	{12,792}
org.jboss.resources.ResourceJobs.java LocalPrepareGatheringJob	2	02	13,461	2	13,391	58	{15,391}	52	{15,391}
org.jboss.informix.InformixJobs.java LocalPrepareGatheringJob	2	02	(1,131)	2	21,135	93	{12,861}	92	{12,861}
java.lang.StringBuilder.append(String)	2	02	11,891	2	21,136	18	{10,561}	18	{10,561}
org.jboss.resource.adapter.jdbc.local.LocalPrepareGatheringJob	2	02	16,391	2	11,491	61	{11,531}	61	{11,531}
org.jboss.resource.adapter.jdbc.local.LocalPrepareGatheringJob	2	02	17,491	2	10,492	98	{12,591}	94	{12,591}
java.lang.StringBuilder.append(String)	158	02	11,251	2	11,251	52	{15,941}	52	{15,941}
org.jboss.proxy.JBossProxyLocalPrepareGatheringJob	1	02	11,391	2	12,391	80	{11,391}	46	{12,391}
java.lang.StringBuilder.append(String)	26	02	(0,91)	2	12,391	52	{12,391}	52	{12,391}
org.jboss.ResourceAdaptorJobs.java LocalPrepareGatheringJob	2	02	16,391	2	16,391	20	{15,791}	20	{15,791}
java.lang.StringBuilder.append(String)	60	02	16,391	2	16,391	47	{12,941}	46	{12,941}
org.jboss.ResourceAdapterLocalPrepareGatheringJob	2	02	16,391	2	16,391	10	{16,391}	10	{16,391}
java.lang.StringBuilder.append(String)	1	02	(0,10)	2	16,391	22	{16,391}	12	{16,391}
java.util.ListIterator.hasNext()	1	02	(0,91)	2	16,391	12	{16,391}	12	{16,391}
java.util.ListIterator.next()	1	02	(0,91)	2	16,391	10	{16,391}	10	{16,391}

图15.10

上半部窗格中的“调用图”在本例中不是十分有用，但对提供瓶颈的一个彩色显示和数据的轻松导航会很有用。

对于每个方法，我们可以快速地获得什么方法调用了它、它调用了什么方法以及它的执行时间怎样分解的一个显示画面。

例如，通过双击BoxOfficeEJB.allocateSeats（JProbe显示它花费了执行时间的45%），我们将会看到图15.11所示的信息（该allocateSeats()方法始终被Method.invoke()方法反射地调用，因为JBoss使用动态代理而不是生成类来实现EJB构件接口）。



图15.11

在这种情况下，这是非常有趣的。它显示OracleJdbcBoxOfficeDAO.reserve()方法所调用的存储过程比通过更新来检查空余座位的选择有更快的运行速度（可能是因为该选择对一个视图进行一个外层连接的缘故）。我们已经在第9章中讨论过这里运行的数据存取代码。

在依照相同的过程仿形另一个用例之前，从Program菜单中选择Clear Performance Data。

可以把快照保存起来供将来使用。这对分开数据收集与分析（两项差别极大的任务）是十分方便的，也使得我们可能比较借助于设计和代码修改来解决问题的工作进度。

稍后，我们将看一看如何使用JProbe来解决一个真正的性能问题。

像JProbe这样的优秀仿形工具是非常宝贵的，它使我们能够把设计建立在真凭实据之上，而不是依靠直觉。高级的仿形工具还可以快速揭示关于应用运行时调用栈的有用信息。

即使根本不存在很大的性能问题，在每个用例上都执行单独的仿形运行也是一个好习惯，这能保证没有任何浪费的活动：比如在多余的串处理方法中，它可以从仿形运行中被明显地看出。

但是，避免沉湎于仿形也是很重要的。使用重复的仿形器运行来获得越来越少的性能收益是一个诱惑。这是浪费时间，尤其是在主要问题都得到了解决之后。

解决性能或可缩放性问题

一旦知道了性能问题出在哪里，就可以考虑如何解决它们。它们被解决得越早，就越有可能不必修改主要设计并重加工现有代码就能消除它们。

有许多解决性能或可缩放性问题的技巧。在本节中，我们将考虑一些最有用的技巧。

服务器选择和服务器配置

在考虑修改设计或改动代码之前，应该保证问题的起因不在本应用的外部。

应用服务器的选择和服务器配置对应用性能有至关重要的影响。

调整应用服务器以满足它所运行的应用的各种需求是极其重要的。除非代码的质量特别差，否则这通常将会产生一个比优化代码更大的回报，而且又不损失可缩放性。

可利用的性能调整参数随着应用服务器而变化，但它们一般包括下列这些参数，其中的部分参数已经在前面的章节中得到过讨论。

- 线程池大小。这个参数将影响Web容器和EJB容器。太多的线程将会导致JVM遇到关于有多少个线程能够并发而又有效地运行的操作系统限制；太少的线程将会导致吞吐量的不必要的节制。
- 数据库连接池大小。
- JSP重编译支持。通常，这个参数会因为少量的性能收益而被禁用。
- 用于HTTP会话和有状态会话EJB的状态复制机制（如果有的话）。
- 所有EJB类型的池大小。
- 用于实体组件的加锁和高速缓存策略（仅当使用了实体组件时才相关）。
- 用于其他任何O/R映射层（比如一个JDO实现）的加锁和高速缓存策略。
- JMS选项，比如是否把消息保存到持久性存储器上。
- 要使用的事务管理器。例如，在我们仅使用了一个事务性资源这种常见的情况下，禁用两步提交支持可能会获得更快的运行速度。

调整JVM选项和操作系统配置是另外一个工作区域。通常，这种调整应该适合应用服务器，而不适合运行在该应用服务器上的具体应用，因此寻找这类信息的最佳地点是在具体应用服务器所携带的技术资料里面。重要的JVM选项包括初始与最大堆大小和无用数据收集参数。关于Sun 1.3 JVM的详细信息，请参见<http://java.sun.com/docs/hotspot/gc/>。

禁用从不使用的应用服务器服务，以使更多内存可供应用使用并消除多余后台线程活动或许也是可能的。

一般说来，应用服务器开发商往往会提供关于性能调整和J2EE性能的指导原则。下面给出的一些是比较好的资源：

- WebLogic
<http://edocs.bea.com/wis/docs70/perform/>
- IBM WebSphere
http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

- SunONE/iPlanet

http://developer.ipplanet.com/viewsource/char_tuningias/index.jsp

数据库配置也同样重要，而且需要专门技能。

免除多余的容器服务

虽然J2EE应用服务器提供了许多有用的服务，但这些服务中的大多数不是无代价的。不必要地使用它们可能会损及性能。有时，通过避免或废除多余容器服务的使用，我们可以改善性能。例如，我们可以：

- 避免EJB的不必要的使用。所有EJB调用——乃至本地调用都会连带着产生一个容器拦截开销。因此，把EJB用在未实现它真正价值的地方会损及性能。
- 避免JMS的不必要的使用。一般说来，应该在业务需求明确要求使用一个异步模型的时候使用JMS。试图使用JMS来改善性能通常会造成相反的结果。
- 避免把实体EJB用在它们不增值并且JDBC访问是一个更简单、更快速替代方案的地方。
- 避免容器服务内的重复。例如，如果我们正依靠Web容器来管理HttpSession对象，可能就没有必要再依靠EJB容器来管理有状态会话组件；必要时把会话状态传递给有状态会话组件或避免使用EJB常常会证明是更加可缩放的。

迄今为止，最重要的性能增益很可能出于避免多余远程调用方面；下文将会进一步讨论这方面的问题。

越简单的设计常常就是性能越好的设计，常常也不会导致损失可缩放性。

高速缓存

要想在J2EE应用中改善性能，最重要的技巧之一是高速缓存（Caching）：把检索代价很高的数据存储起来，使它不必再从原数据源中被检索就能被快速返回给客户。

高速缓存可以在一个J2EE体系结构中的许多地方实行，但只有在它使一个体系结构层能够消除一些针对下一层的调用时才是最有益的。

在分布式应用中高速缓存通过消除远程调用可以产生巨大的性能增益，在所有应用中，高速缓存都可以避免从应用服务器到数据库的调用，这种调用除了涉及JDBC的开销之外，还可能会涉及网络往返行程。一个成功的高速缓存策略，将会改善一个应用中这样一些部分的性能：它们不能从高速缓存数据中直接受益。一般说来，服务器响应时间将会由于工作负荷的降低而变得更佳，网络带宽将会得到解放。数据库将会有更少的工作要做，从而意味着它的响应速度更快，每个数据库实例或许能够支持一个规模更大的J2EE服务器聚类。

但是，高速缓存也提出重大的设计挑战，无论它是由J2EE服务器开发商、应用开发人员还是第三方实现的。因此，如果没有用测试结果或其他确凿证据证明使用它是正确的，就不应该使用它。

如果我们在应用代码中实现一个高速缓存器，这就意味着我们可能必须编写J2EE曾经承诺我们不必编写的那种复杂代码，比如线程管理。可能需要J2EE应用运行在一个聚类化环境中的事实有可能会显著增加复杂性，即使我们使用了一个第三方高速缓存解决方案。

何时进行高速缓存

在可以开始缓存数据之前，有一些问题是应该先弄清楚的。其中的大多数问题都与时效性、争用和聚类的中心问题有关。

- 在没有缓存数据的情况下获得该数据有多慢？引进一个高速缓存器将把应用的性能改善得足以证明增加的复杂性是值得的吗？通过高速缓存来避免网络往返行程很可能是值得的。可是，由于高速缓存常常增加复杂性，所以若非必要，不应该实现高速缓存。
- 我们对自己要高速缓存的数据集的使用与变动情况了解吗？如果该数据集是只读的，高速缓存是一个明显的优化。当我们只朝着只写这一个方向前进时，高速缓存就变得越来越没有吸引力。并发问题出现，而且避免读取的好处变小。
- 我们能容忍陈旧数据吗？如果能，容忍多大的陈旧程度？在有些情况下，这可能是几分钟，因而使高速缓存的回报变得更大。在其他情况下，数据必须始终是最新的，高速缓存是不可接受的。业务需求应该指出什么样的陈旧程度是容许的。
- 高速缓存器出现毛病的后果是什么？如果用户的信用卡被划走了一个不正确的量（例如，如果缓存的价格信息已经过时），这个错误明显是无法接受的。相反，如果一个媒体站点上的临时用户所阅读的标题新闻是30秒钟后就过时的，那么为了获得性能上的明显提高而付出这样的代价可能是完全可接受的。
- 我们能应付为支持高速缓存而需要增加的实现复杂性吗？如果我们使用了一个合适的通用高速缓存器实现，这种复杂性将会得到缓解，但必须认识到读-写高速缓存会引入明显的线程化问题。
- 我们需要高速缓存的数据量是可控制的吗？很明显，如果我们需要高速缓存的数据集含有数百万个实体，而且我们又无法预见用户将需要哪些实体，那么实现一个高速缓存器将只浪费内存。数据库非常擅长从一个大范围内抽取出少量记录，而我们的高速缓存器可能不太胜任这项工作。
- 我们的高速缓存器在聚类中适用吗？这对参考数据来说通常不是问题：如果每个服务器都有只读数据的各自副本，这不是问题，但维持已缓存的读-写数据在整个聚类中的完整性是很困难的。如果高速缓存器之间的复制看起来是必不可少的，那么非常明显的一点是，我们不应该把这样的基础结构实现为应用的一部分，而是应该在应用服务器中寻找支持，或者寻找一个第三方产品。
- 高速缓存器能适度地满足客户对数据所做的那一类查询吗？否则，我们可能会发现自己正试图重复发明一个数据库。在有些情况下，用XML文档比用已缓存的Java对象更容易满足对查询的需求。
- 我们能肯定应用服务器无法满足我们的高速缓存需求吗？例如，如果我们知道应用服务器提供了一个有效的实体组件高速缓存器，那么把数据高速缓存在客户上可能是不必要的。在这里，一个决定性的问题是客户距离EJB层有多远（指网络距离）？巴累托定律（即80/20规则）也适用于高速缓存。性能增益的大部分通常可以使用解决这些较困难的高速缓存问题所付出的努力的一小部分来实现。

数据缓存可以显著地改善J2EE应用的性能。可是，高速缓存也会增加复杂性，而且是引起隐错的一个常见原因。实现不同高速缓存解决方案的难度变化很大。应该欣然接受容易获得的成功，比如高速缓存只读数据。这增加少量的复杂性，而且可以产生不错的性能改善。当高速缓存是一个较难解决的问题时——比如当高速缓存涉及到读写一些数据时，应该更加仔细考虑可替代的方法。

不要依据我们将来会需要高速缓存的假设仓促实现它；应该把高速缓存策略建立在性能分析的基础之上。

体系结构层之间关系明确的上佳应用设计，通常会使增加必需的高速缓存很容易。基于接口的应用设计尤其如此；如果业务需求能得到满足，我们可以轻松地使用一个高速缓存实现替换掉一个接口。稍后，读者将会见到一个简单高速缓存器的例子。

在何处进行高速缓存

由于使用J2EE自然会产生一个分层的体系结构，所以有多个可能会出现高速缓存的位置。在这些类型的高速缓存中，有些是由J2EE服务器或底层数据库实现的，并且是开发人员可以通过配置而不是代码来访问的。其他类型的高速缓存必须由开发人员实现，并且会占总开发努力的一大部分。

下面来看一看每个位置的选择，首先从后端开始，如表15.1所示。

一般说来，我们越接近客户进行高速缓存，性能改善得就越显著，尤其是分布式应用中。较不重要的一个方面是，我们越接近客户进行高速缓存，从高速缓存器中受益的范围就越窄。例如，如果我们缓存整个应用的动态生成页面，这些页面的响应时间将会非常快（当然，这种特殊的优化只适合不含有用户特有信息的页面）。但是，这是一种“笨拙”的高速缓存形式；该高速缓存器可以有这个数据的一个明显关键字（可能是被请求的URL），但它无法理解它正存储的数据，因为该数据与表示置标混在一起。这样的高速缓存器对Swing客户端毫无用处，即使这些缓存页面的可变部分中的数据与Swing客户端相关。

J2EE标准基础结构确实只适合支持对实体EJB中的数据进行高速缓存。如果我们没有选择使用实体EJB（而且存在我们为什么可能不选择使用它们的许多理由），这个可选项则是无效的。它在分布式应用中的价值也是有限的，因为分布式应用在将数据从EJB容器转移到远程客户时所面对的问题和它们将数据从数据库转移到EJB容器时所面对的问题差不多。

因此，我们通常需要实现我们自己的高速缓存解决方案，或者借助于另一个第三方高速缓存解决方案。下面是笔者建议的几条指导原则：

- 除非高速缓存涉及到参考数据（在这种情况下，它实现起来很简单），或者性能明确规定要求它，否则避免高速缓存。一般说来，分布式应用比集中式应用更有可能需要实现数据高速缓存。
- 由于读/写式高速缓存器涉及到复杂的并发问题，所以要使用第三方库（下文讨论）来隐藏所需同步的复杂性。要使用最简单的可以提供令人满意的性能的方法来保证并发访问之下的完整性。

表15.1

高速缓存器的位置	由谁实现	可能的性能改善	实现的复杂度	说明
数据库	RDBMS开发商	显著。可是，RDBMS中所高速缓存的数据量与应用之间的距离还是很长的，尤其足一个分布式应用中	不需要任何J2EE工作。有些数据仓库配置可能是单一的，索引创建是简单的。我们没有能够使用更有效的表类型，视我们的目标数据库而定	RDBMS高速缓存常常遭J2EE开发人员忽视。大多数RDBMS都高速缓存常用语句的执行路径，并且可能会高速缓存查询结果。RDBMS索引相当于是提前的高速缓存，并且可以产生非常显著的性能改善。我们在第9章中见过PreparedStatement在应用代码中的使用，这可以保证RDBMS能够执行有效的高速缓存。
实体组件高速缓存器	EJB容器开发商	变化。如果该高速缓存器极大地降低了对底层数据库的调用数量，性能改善会非常显著。可是，这假定一个非常高效的实体组件实现：在一个聚类化环境中，该高速缓存器将需要是分布式的，从而引发了事务性完整性和复制的问题。	零或非常小	J2EE规范没有保让高速缓存，这意味着一个表现得令人满意的只使用了有效实体组件高速缓存的体系结构是不可移植的。不过，第三方持久性管理器可以与多个应用服务器一起使用，这是一个帮手。

(续表)

高速缓存器的位置	由谁实现	可能的性能改善	实现的复杂度	说明
数据访问层的数据缓存——即Web容器或EJB容器中除实体组件高速缓存，比如一个JDO实现或一个像TopLink那样第三方OR映射解决方案	第三方开发商	好处与实体组件高速缓存类似。也有类似的向题：在聚类化环境中将会减弱，除非它是分布式的。这意味着只有高档产品才适合可缩放部署	小	引入对另一个产品的依赖性，但EJB容器除外。不过，我们可能会由于其他缘故而选择一个第三方OR映射工具。
会话EJB	开发人员	取决于缓存数据的检索有多大代价。在分布式应用中，没有消除到EJB容器的网络往返行程	小到中等	在EJB层中使用Singleton设计模式很困难。因此，由于可能处于不同状态中的那些高速缓存器，缓存数据在无状态会话组件实例中可能是重复的。不过，缓存数据将使一个无状态会话组件的所有用户受益。 有状态会话组件只能为单个用户高速缓存数据。ejbCreate()方法是检索数据的自然地点。可是，使用惰性装入是很轻松的，因而只有在某个业务中第一次需要资源时才检索它们，因为EJB可以被实现得好象它们就是单线程的。因此，没有必要执行同步，或者操心竞争现象。ejbRemove()方法应该用于释放不能被简单地留给无用数据收集器处理的资源。我们已经在第10章中讨论过会话组件中的数据缓存

(续表)

高速缓存器的位置	由谁实现	可能的性能改善	实现的复杂度	说明
运行在Web容器中的业务对象	开发人员或第三方解决方案	非常显著。消除了分布式应用中EJB容器之间的网络往返行程，即使用EJB被集中在一个VM中，对EJB的调用仍将比本地方法调用的执行速度慢	中等到高	容易取得的成功（比如高速缓存参考数据）将会产生很大的回报。可是，在试图实现较有疑问的高速缓存解决方案之前，建议三思而行。J2EE基础结构无法帮助解决并发问题。不过，我们或许能使用第三方解决方案，其中的两个下文将会加以讨论
Web层	开发人员或第三方解决方案	非常显著	中等到高	这里有大量替代方案，比如高速缓存自定义标志，高速缓存服务器小程序，以及高速缓存筛选器。高速缓存筛选器特别有吸引力，因为它能使用高速缓存器设置在web.xml部署描述符中被声明性管理
位于J2EE应用服务器前面的高速缓存器	J2EE服务器开发人员	非常显著	小到中等	1. 一个类似，但的应用服务器开发商提供。 Web层缓存由WebSphere和iPlanet/SunONE提供 我们将往下文的“Web层性能问题”一节中讨论适合Web解决方案的“前端”高速缓存

- 要考虑多个高速缓存器协同工作的影响。这种合作导致用户看到比这些高速缓存器中的任意一个所能容忍的还要陈旧的数据了吗？或者，一个高速缓存器消除了对另一个高速缓存器的需要了吗？

供J2EE应用中使用的第三方高速缓存产品

现在来看几个可以用在J2EE应用中的第三方商业高速缓存产品。我们花钱购买商业解决方案的主要目的是为了实现可靠的复制式高速缓存功能度，以及避免需要内部地实现和维护复杂的高速缓存功能。

出自Tangosol公司的Coherence (<http://www.tangosol.com/products-clustering.jsp>) 是一个复制式高速缓存解决方案，该解决方案甚至声称支持由地理位置分散的服务器所组成的聚类。Coherence能与包括JBoss在内的大多数领先应用服务器集成在一起。Coherence高速缓存器基本上是标准Java映像实现（比如java.util.HashMap）的替代品，所以使用它们只要求Java核心接口的Coherence特有实现。

出自SpiritSoft公司的SpiritCache (http://www.spiritsoft.net/products/jms_jcache/overview.html) 也是一个复制式高速缓存解决方案，并声称提供一个“适用于Java平台的通用高速缓存框架”。SpiritCache API基于被提议的JCache标准API (JSR-107:<http://jcp.org/jsr/detail/107.jsp>)。由Oracle提议的JCache定义了一个用于高速缓存和检索对象的标准API，其中包括一个基于事件并允许应用代码注册高速缓存器事件的系统。

商业高速缓存产品可能会证明，它们对具有高级高速缓存需求的应用来说是一项非常有益的投资，比如在一个服务器聚类中进行高速缓存的需求。内部开发和维护复杂的高速缓存解决方案会花费非常高的代价。然而，即使我们使用第三方产品，运行一个聚类式高速缓存仍是十分复杂的应用部署，因为除了J2EE应用服务器之外，该高速缓存产品也需要正确地配置以适合我们的聚类环境。

代码优化

由于设计在很大程度决定了性能，因此除非应用代码编写得特别糟糕，否则代码优化在J2EE应用中几乎不值得去努力，除非该代码优化的目标是针对已知的问题区域。然而，所有的专业开发人员都应该熟悉代码级的性能问题，以避免犯最基本的错误。要想了解Java性能的一般性讨论，笔者向读者推荐由Jack Shirazi编著和O'Reilly公司出版的“Java Performance Tuning”一书 (ISBN 0-596-00015-4)，以及由Prentice Hall公司出版的“Java 2 Performance and Idiom Guide”一书 (ISBN 0-13-014260-3)。还有许多关于性能调整的上佳在线资源。Shirazi维护了一个性能调整网站 (<http://www.javaperformancetuning.com/>)，该网站含有一个目录，该目录详尽列出了来自许多渠道的代码调整技巧。

除非万不得已，否则应该避免会损及可维护性的代码优化。这样的“优化”不仅是一项一次性的工作，而且很可能是今后的一项成本和产生隐错的一大原因。

编程问题的级别越高，代码优化产生的潜在性能增益就会越大。因此，常常存在这样的可能性：通过重新排列一个算法的步骤三类的技巧，使代价大的任务只有在绝对必要时才

得到执行，从而实现更好的结果。和设计的情形一样，一盎司的预防将会值一磅的良药。虽然对性能的执着追求往往会产生相反结果，但好的程序员不会编写以后需要优化的效率非常低的代码。然而，先试一个简单的算法，然后只在必要时才修改实现来使用一个速度更快但更复杂的算法有时是有意义的。

像循环展开之类的真正低级技巧不太可能给J2EE系统带来任何好处。任何优化都应该是有目标的，并应该以仿形测量的结果为基础。当查看仿形输出结果时，要把重点放在最慢的5个方法上；把精力放在别处可能会是一种浪费。

表15.2列举了一些可能的代码优化（值得做的和造成相反结果的），以举例说明要考虑的性能与可维护性之间的某些权衡。

表15.2

技巧	性能改善	对可维护性的影响
通过创建对象池和“规范化”对象（以防止创建多个代表同一个值的对象）之类的技巧，最大限度地减少对象创建	变化。可能会减少无用数据收集的工作。就较新VM的高级无用数据收集而言，性能收益可能不是非常大	实现这样的算法可能是复杂的。代码可能会变得难懂。 这是编写代码时应该牢记在心的那种性能问题。如果没有充分的理由，又存在一个替代方案，比如使用基本类型，就不应该创建大量对象
使用正确的集合类型。例如，当不知道要预定多少个元素时，以及当一次将只添加一个元素时，使用java.util.LinkedList；当知道要预定多少个元素时，使用java.util.ArrayList。还记得Computer Science I中的所有数据结构模块吗？Sun公司已经把那些标准数据结构中的许多实现为核心库集合，意味着我们只需要选择最适合的数据结构即可	变化。可能是非常显著的，如果该列表不可预料地增长或需要分类	无。我们应该通过该集合的接口（比如java.util.List），而不是具体类（比如java.util.LinkedList）来访问它
使用异常而不是条件检查来结束循环	随虚拟机而变化	可能会使代码变得难懂。这也是我们应该尽可能避免的一种优化的例子
使用终结类和方法	小	使用终结类与方法常常是一种优良的风格，所以我们经常由于其他缘故而使用这种“优化”（参见第4章）
避免使用System.out	显著，如果涉及到大量输出	在任何情况下，企业级应用使用一个恰当的日志记录框架都是至关重要的（参见第4章）

(续表)

技巧	性能改善	对可维护性的影响
避免计算多余的条件。Java保证and和or的“短路”计算。因此，我们应该首先做最快的检查，因而可能会避免计算速度较慢的检查	会是显著的，如果某个代码段被频繁调用	无
避免对String的操作，应该优先使用StringBuffer。由于String是不变的，所以String操作可能是低效率的和浪费的，导致许多短命对象的创建	可能是显著的，视JVM而定	由于显著的性能收益，这应该是专业开发人员对阅读稍微冗长的StringBuffer语法早已习惯的一种情况。需要注意的是，Sun公司的JDK 1.3中的HotSpot JVM似乎自动进行这样的优化，不过，最好不要依赖这样的优化
避免多余的String或StringBuffer操作。甚至连StringBuffer操作都相当慢	显著	无。这是我们可以安全地获得快速成功的一个区域
最大限度地减少接口的使用，因为它们调用起来可能比类慢	非常小	这是有可能会破坏代码库的那种“优化”。少量的性能增益不值得这种方法所造成的伤害

String和**StringBuffer**操作对性能会产生很大的影响。正如使用一个像JProbe之类的仿形器所快速演示的，甚至连**StringBuffer**操作都是代价十分高昂的。请务必注意使用频繁的代码中的串操作，以便保证这些串操作是必不可少和尽可能有效的。

作为这方面的一个例子，请考虑示例应用中的日志记录。下面这条似乎无害的语句取自我们的TicketController Web控制器，并且只被执行一次；如果用户请求他们的会话中已保存的一个关于预订的信息，这条语句会占用高得令人惊讶的执行时间：总执行时间的5%。

```
logger.fine("Reservation request is [" + reservationRequest + "]");
```

问题不是出在这条记录语句自身上，而是出在执行一个串操作（HotSpot会把串操作优化成**StringBuffer**操作）并对**ReservationRequest**对象调用**toString()**方法（**ReservationRequest**对象又调用另外几个串操作）的语句上。在生产中，添加一个关于该日志消息是否将被显示的检查以避免在它不被显示时创建它，将会几乎完全消除这个代价，因为任何一个好的日志记录包都会提供效率非常高的日志配置查询机制。

```
if (logger.isLoggable(Level.FINE))
    logger.fine("Reservation request is [" + reservationRequest + "]");
```

当然，5%的性能节省在大多数情况下根本不是什么了不起的事情，但日志记录的这种草率运用在频繁得到调用的方法中更危险。这种有条件地进行的日志记录在得到频繁使用的代码中是不可或缺的。

通常情况下，生成日志输出对性能的影响不是太大。但是，不必要地构造日志消息，尤其是如果这种构造涉及到不必要的`toString()`调用，将会有高得令人惊讶的代价。

两个特别难处理的问题是同步（Synchronization）和反射（Reflection）。这两个问题可能是非常重要的，因为它们位于设计与实现之间的中途。现在，让我们依次来仔细讨论一下这两个问题。

同步的正确使用既是一个设计问题，又是一个编程问题。过度的同步会扼制性能，并且可能造成死锁。不充分的同步会引起状态残缺。同步问题常常在实现高速缓存的时候出现。关于Java线程设计的一本重要参考书是由Wesley公司出版的“Concurrent Programming in Java: Design Principles and Patterns”（ISBN 0-201-31009-0）。笔者衷心建议读者在实现复杂的多线程代码时参考这本书。不过，可能下面的提示也很有用：

- 不要假设同步将始终证明对性能是致命的。要根据经验来制定决策。特别是，如果处于同步控制之下执行的操作执行速度快，同步则可以保证数据完整性，同时对性能的影响又最小。本章的后面将举一个实用的例子来说明与同步有关的各种问题。
- 在凡是可能的地方，要使用自动变量来取代实例变量，以便同步不是必需的（这个建议特别适用于Web层控制器）。
- 要使用最小的同步，只要符合保护状态完整性的要求就行。
- 要同步尽可能少的代码部分。
- 要记住，像int（而不是long和double）之类的对象引用才是原子的（只用单个操作读或写），所以它们的状态无法遭到破坏。因此，两个线程连续地初始化同一个对象（就像把一个对象放入到一个高速缓存器中时那样）的竞争现象可能无害（只要初始化发生多次不是一个错误），而且是可接受的（为了求得减少同步）。
- 使用锁分裂（Lock splitting）来最大限度地消除同步所造成的性能影响。锁分裂是一种增加同步锁粒度的技巧，以便每个得到同步的块都只锁定对正被更新对象感兴趣的线程。如果可能，要使用一个像Doug Lea的util.concurrent那样的标准包，以利用像锁分裂这样的著名同步技巧。请记住，使用EJB来照顾并发问题不是编写自定义的低级多线程代码的惟一可选方案：util.concurrent是一个开放源包，能够用在一个Java应用内的任何地方。

反射因其速度缓慢而著称。反射对许多J2EE功能度是不可或缺的，并且是编写通用Java代码时的一个强大工具，因此值得比较仔细地看一看各种相关的性能问题。十分明显的一点是，围绕反射性能的大部分担心是毫无根据的。

为了举例说明这一点，笔者运行了一个简单的测试来记录4个基本反射操作的时间。

- 利用`Class.forName(String)`方法按名装入一个类。调用该方法的代价取决于这个被请求的类是否已被装入。任一操作（使用或未使用反射的），只要它需要一个类被第一次装入，都会比较慢。
- 通过调用`Class.newInstance()`方法来实例化一个已装入的类，进而使用该类的无变元构造器。
- 自省：使用`Class.getMethod()`查找一个类的方法。
- 使用`Method.invoke()`的方法调用，只要已高速缓存了指向一个方法的一个引用。

这个测试的源代码可以在示例应用下载中的/framework/test/reflection/Tests.java路径下找到。

下面这个方法就是通过反射调用的：

```
public String foo(int i) {  
    return 'This is a string with a number ' + i + ' in it';  
}
```

当在一台1GHz Pentium III计算机上的JDK 1.3.1_02下并行地运行这些测试时，有如下最重要的结果：

- 通过Method.invoke()调用这个方法10 000次花费480ms。
- 直接调用这个方法10 000次花费301ms（不到两倍那么快）。
- 使用两个超类和量相当大的实例数据创建一个对象10 000次花费21 371ms。
- 使用new操作创建同一个类的对象10 000次花费21 280ms。这意味着无论使用反射还是new操作符，都不会对创建一个大对象的代价产生任何影响。

笔者从这个测试和过去运行过的测试中得出的结论及笔者从开发实际的应用中获得的经验是：

- 使用反射来调用一个方法是非常快速的，只要有一个指向Method对象的引用是可使用的。当使用反射时，只要有可能，要设法高速缓存自省的结果。请记住，可以在声明类的任一对象上调用一个方法。无论该方法做什么工作，这个工作的代价都有可能会超过该反射调用的代价。
- 实例化任何平凡对象的代价都会超过在相关类上调用newInstance()方法的代价。当一个类有几个具有实例数据的实例变量和超类时，对象创建的代价比通过反射启动该对象创建的代价高数百倍。
- 反射操作的速度是如此之快，以致每个Web请求都做一次任何数量的反射对性能也不会产生可感觉到的影响。
- 诸如串操作之类的慢速度操作比使用反射调用方法慢得多。
- JDK 1.3.1和JDK 1.4中的反射操作一般都比JDK 1.3.0中的反射操作快——而且有些快很多。Sun公司已经认识到反射的重要性，而且已经投入了大量精力来改进每个新JVM的反射性能。

流行在许多Java开发人员当中的“反射很慢”的假设是一种误导，而且正变得日益与成熟的JVM不相符。避免反射是没有用处的，除非在不寻常的环境中——例如在一个深层嵌套的循环中。反射的正确使用有许多好处，它的性能开销在几乎任何地方都不足以证明避免它是正确的。当然，应用代码通常只通过基础结构代码所提供的一个抽象来使用反射。

案例研究：示例应用中的“Display Show”页面

下列测试上的所有基准程序都运行在一台具有512MB RAM、运行Windows XP操作系统的1GHz Pentium III计算机上。Microsoft Web Application Stress Tool、应用服务器

及数据库都运行在同一台计算机上。软件版本分别是JBoss 3.0.0、Oracle 8.1.7及Java JDK 1.3.1_02。日志记录被切换到生产级（仅错误和警告）。

现在来看一个案例研究，该案例解决一个用例在示例应用中的各种性能需求。首先来考虑以“Display Show”页面为目标的请求。这个页面显示关于某一特定节目的所有可预订演出场次的信息。“Welcome”页面直接链接到这个页面，所以大多数用户将在他们的第二个页面视图上到达这里，尽管他们可能对不同的节目感兴趣。正因为如此，这个页面能够处理频繁的用户活动，它再现快速，以及生成它不使系统有太大负荷都是至关重要的。

这个页面上所显示的部分信息是很少变化的参考数据，比如节目的名称和定价结构。其他信息经常变化。例如，我们必须显示每场演出的每种座位类型的座位可供应性（其中给一个节目显示10场演出，给每种座位类型显示4级，这意味着40个可供应性选择）。业务需求规定：高速缓存可能是可接受的，如果需要它提供足够的性能；但又规定：座位可供应性信息的实时性必须不能超过30秒钟。图15.12表示了这个页面。

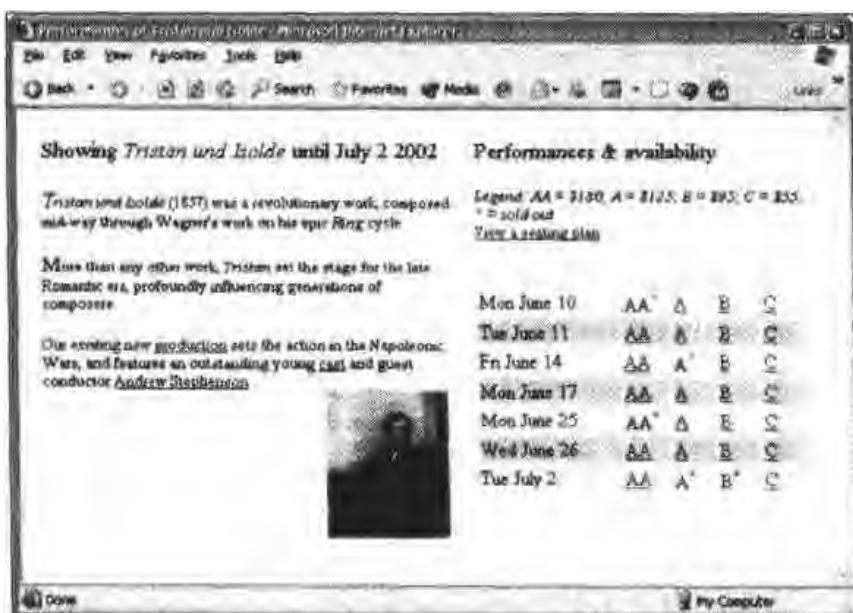


图15.12

我们首先在应用代码中没有高速缓存或其他优化的情况下运行载荷测试，来看看是否存在问题是。Microsoft Web Application Stress Tool显示：在有100个并发用户的情况下，这个页面每秒钟会获得14次点击，其中平均响应时间刚过6秒。该载荷测试显示JBoss使用了80%的CPU时间，而Oracle使用了几乎20%的CPU时间（在载荷测试期间利用操作系统的负载监视工具是十分重要的）。

虽然这超过了我们为并发访问所设定的适度性能目标，但没有满足我们为响应时间设定的需求。如果我们必须显示一个节目的3场以上演出（我们的测试数据是一个节目3场），或者Oracle是在一个远程服务器上，吞吐量和性能会急剧变糟，生产中的情况也将如此。当然，我们将会在一个实际的应用中测试这些情景的结果，但是在编写本书时，笔者可以支配的硬件和时间有限。因此，我们必须做必要的设计和代码修改来改进生成这个页面的性能。

从Task Manager的显示结果中可以明显看出，问题主要出在与数据库进行通信和在数据库内工作方面。但是，在开始改进我们的设计和修改代码之前，获得该应用将其时间都花费在了哪里的一些准确测量数据是一个不错的主意。因此，我们在JProbe仿形器中仿形了两个以这个页面为目标的请求。按累积方法时间排序后，仿形结果看起来如图15.13中所示。

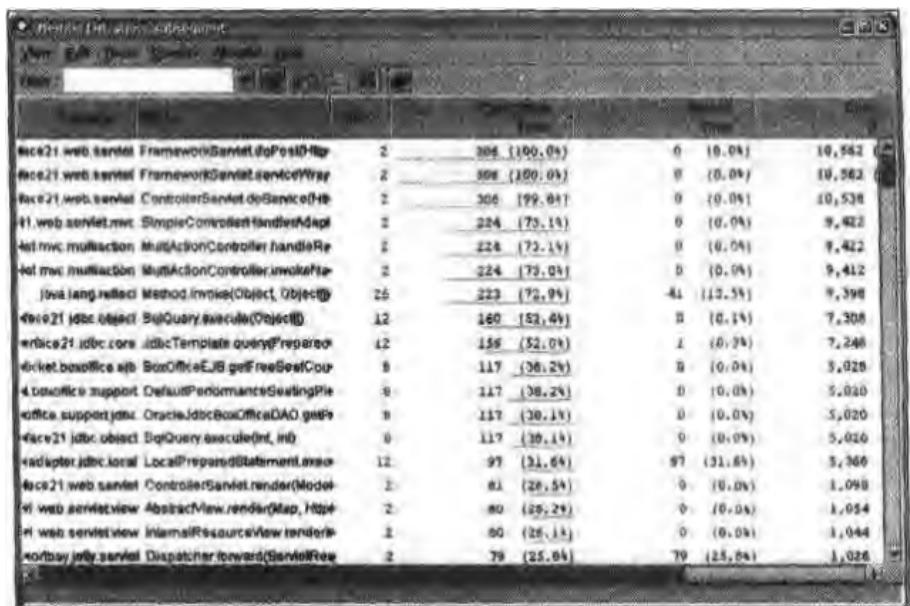


图15.13

这些结果表明：我们每个页面视图执行了6个SQL查询（由SqlQuery.execute()方法的12次调用指示），而且这些查询占用了总时间的52%。再现该JSP视图占用了26%的总执行时间，这个比例高得惊人。显然，数据库访问是性能的主要限制因素。借助于Method.invoke()方法，反射性地使用反射来调用方法所花费的13%执行时间表明每个页面视图使用了12个EJB访问。JBoss和第11章中所讨论的EJB代理基础结构都在EJB调用中使用反射。由于调用EJB方法的开销很大，所以每个页面12个EJB调用也是高得令人无法接受的，因而我们也将需要解决这个问题。

由于这些查询都是简单的选择操作，并且不涉及事务或加锁问题，所以我们可以排除数据库中或应用服务器内的加锁（也应该核实数据库已得到正确配置，数据库模型有效；在这里，我们假设这些都已得到核实）。由于我们无法提高简单的选择操作的效率，所以需要在业务对象中实现高速缓存，以便最大限度地减少对数据库的调用次数。由于业务需求允许这个屏幕上所呈现的数据长达30秒钟后才过时，所以我们还有回旋的余地。

由于com.wrox.expertj2ee.ticket.web.TicketController中的Web层代码被编写成使用com.wrox.expertj2ee.ticket.command.AvailabilityCheck接口来检索可供应性信息，而不是使用一个具体实现，所以我们可以轻松地换用一个不同的Java组件实现来实现高速缓存。

接口驱动式设计是优良设计习惯产生性能调整方面的最大自由度的一个领域。虽然通过一个接口来调用方法与在一个类上调用方法相比会有一点点开销，但是与既能够重新实现一个接口又不影响调用者的各种好处相比，这一点点开销又算得了什么。

在高级设计阶段，作为高速缓存的一个替代方案，我们还考虑了使用JMS来启动更新预订与购买的可能性，目的是为了只有在知道数据被修改时才使它无效。因为没有进一步的活动再通过Web层时，预订在数据库中会超时，所以这实现起来将是相当复杂的：我们将必须安排在预订期满时发出第二个JMS消息，以便任一高速缓存器都能够检查该预订是否期满或是否已被转变成购买。进一步的性能调查将会显示这一选择是不是必不可少的。

首先来看一看AvailabilityCheck接口的实现中用来返回演出与可供应性组合信息的表示代码。做过突出处理的那些行使用了BoxOffice EJB，这个EJB需要执行一个数据库查询。这个方法被调用了几次，以便构造每个节日的信息。请注意，JNDI查找的结果已在基础结构代码中得到了高速缓存。

```
public PerformanceWithAvailability getPerformanceWithAvailability(
    Performance p) throws NoSuchPerformanceException {
    int avail = boxOffice.getFreeSeatCount(p.getId());
    PerformanceWithAvailabilityImpl pai =
        new PerformanceWithAvailabilityImpl(p, avail);
    for (int i = 0; i < p.getPriceBands().size(); i++) {
        PriceBand pb = (PriceBand) p.getPriceBands().get(i);
        avail = boxOffice.getFreeSeatCount(p.getId(), pb.getId());
        PriceBandWithAvailability pba =
            new PriceBandWithAvailabilityImpl(pb, avail);
        pai.addPriceBand(pba);
    }
    return pai;
}
```

首先从尝试最简单的方法开始：按一个散列表中的关键字来高速缓存演出场次。由于这是十分简单的，所以有理由在应用代码中实现它，而不是引进一个第三方高速缓存解决方案。我们没有考虑使用同步——可能是实现高速缓存器时的最大难题，而是使用了一个java.util.HashTable来作为PerformanceWithAvailability对象的一个高速缓存器，这些对象用整数型演出ID做为关键字。

请记住，Java 2之前的旧集合在几乎每个方法上都使用同步，其中包括映像上的put和get，而像java.util.HashTable这样的较新集合让调用者来处理必要的同步。这意味着这些较新集合始终是适合只读数据的一个更佳选择。

没有必要给高速缓存器的最大大小设置一个限定值（实现高速缓存器时有时会遇到的另一个问题），因为永远不会有比我们可以保存在RAM中的节目和场次对象更多的这类对象。同样，没有必要担心聚类的影响（另一个潜在的高速缓存问题）；业务需求规定数据应该不旧于30秒钟，但没有规定它在任一聚类的所有服务器上都必须是完全相同的。

由于业务需求规定座位选择页面（它们的生成也使用AvailabilityCheck接口）始终需要最新数据，所以我们需要做一点再加工来给AvailabilityCheck接口中的方法添加一个新的boolean参数，以便高速缓存能被禁用，如果调用者选择这么做。

我们的高速缓存逻辑需要能够检查一个已得到缓存的PerformanceWithAvailability对象有多旧，所以我们使PerformanceWithAvailability接口扩展成了一个简单的接口TimeStamped，这个接口将暴露对象的年龄，如下所示：

```

package com.interface21.core;

public interface TimeStamped {
    long getTimestamp();
}

```

由于高速缓存数据的时间长度对性能可能是至关重要的，所以我们在CachedAvailabilityCheck类上暴露了一个“超时”Java组件属性：CachedAvailabilityCheck类是我们扩展Availability-Check接口所实现的新高速缓存实现，并使用了一个HashTable作为它的内部高速缓存器：

```

private Map performanceCache = new Hashtable();
private long timeout = 1000L;

public void setTimeout(int secs) {
    this.timeout = 1000L * secs;
}

```

现在，我们将getPerformanceWithAvailability()分成两个方法，将捕获的新数据放入reloadPerformanceWithAvailability()方法。我把决定是否为要求的ID使用性能数据的高速缓存副本的条件突出显示。注意，最快的检查，如超时组件属性是否设置为0，即高速缓存是否被有效禁用，首先被执行，这样我们不需要估算最慢的检查，此检查需要取得当前的系统时间（一个相当慢的操作），除非必要。

严格地说，关于超时属性是否为0的检查不是十分必要的，因为即使该属性为0，时间截比较仍照常进行。不过，由于这项检查实际上根本不占用时间，所以执行一项多余的检查有时比执行一项既多余而又昂贵的检查要好得多。

```

public PerformanceWithAvailability getPerformanceWithAvailability(
    Performance p, boolean acceptCached)
    throws NoSuchPerformanceException {

    Integer key = new Integer(p.getId());

    PerformanceWithAvailability pai =
        (PerformanceWithAvailability) performanceCache.get(key);

    if (pai == null ||
        this.timeout <= 0L ||
        !acceptCached ||
        System.currentTimeMillis() - pai.getTimestamp() > this.timeout) {

        pai = reloadPerformanceWithAvailability(p);
        this.performanceCache.put(key, pai);
    }
    return pai;
}

private PerformanceWithAvailability reloadPerformanceWithAvailability(
    Performance p) throws NoSuchPerformanceException {
    int avail = boxOffice.getFreeSeatCount(p.getId());
    PerformanceWithAvailabilityImpl pai =
        new PerformanceWithAvailabilityImpl(p, avail);

    for (int i = 0; i < p.getPriceBands().size(); i++) {
        PriceBand pb = (PriceBand) p.getPriceBands().get(i);
        avail = boxOffice.getFreeSeatCount(p.getId(), pb.getId());
        PriceBandWithAvailability pba =
            new PriceBandWithAvailabilityImpl(pb, avail);
        pai.addPriceBand(pba);
    }
    return pai;
}

```

由于使用一个同步散列表能自动保证数据完整性，所以我们没有必要亲自执行同步。存在这样一种可能性：在上述代码清单中加了底色的第一行处，我们从该散列表中检索出一个null值，但是在我们检索出数据并把它插入到该散列表中之前，另一个线程已打败了我们。不过，这不会引起任何数据完整性问题：偶尔、多余的数据库访问与更复杂、易出错的代码相比是一个更小的祸害。聪明的同步有时是必要的，但如果它不提供真正的价值，最好避开它。

除了这些修改外，我们还在ticket-servlet.xml文件的相关组件定义中把availabilityCheck组件的超时属性设置成了20秒，并重新运行Web Application Stress Tool。结果是吞吐量和性能方面的巨大改善：每秒51个页面，在没有高速缓存的情况下只达到14个页面。Task Manager指出Oracle现在实际上未做任何事情。这远远地满足了我们的业务需求。

不过，数据越新越好，所以我们试着降低超时设置。一个10秒的超时设置导致每秒平均49个页面，其中平均响应时间不足2秒，它指出这可能是值得做的。把超时降至1秒导致吞吐量降至每秒28个页面：这可能是一个太大的性能牺牲。

至此，笔者仍担心同步的效果。一种更高级的方法会最大限度地减少加锁并产生更好的结果吗？为了证实这一点，笔者编写了一个多线程的测试，该测试能使笔者使用以前讨论过的com.interface21.load包只测试CachingAvailabilityCheck类。工作者线程扩展了AbstractTest类，并只包括了从整个测试套件启动时被装入的那些节目当中的一个随机节目里检索出数据。

```
public class AvailabilityCheckTest extends AbstractTest {
    private AvailabilityFixture fixture;
    public void setFixture(Object fixture) {
        this.fixture = (AvailabilityFixture) fixture;
    }
    protected void runPass(int i) throws Exception {
        Show s = (Show) fixture.shows.get(randomIndex(fixture.shows.size()));
        fixture.availabilityCheck.getShowWithAvailability(s, true);
    }
}
```

每个线程都调用同一个AvailabilityCheck对象是必需的，所以我们创建一个由所有实例共享的“夹具”类。这创建并暴露一个CachingAvailabilityCheck对象。需要注意的是，在下面的代码清单中，笔者暴露了一个公用终结实例变量。通常情况下，这不是一个好主意，因为它不是Java组件友好的，也意味着我们不能在一个获得器方法中增加智能，但它在一个快速测试的情况下是可接受的。AvailabilityFixture类暴露3个能使测试参数化的组件属性：timeout（直接设置受测对象CachingAvailabilityCheck的超时），以及minDelay和maxDelay（下文讨论）。

```
public class AvailabilityFixture {
    public final CachingAvailabilityCheck availabilityCheck;
    public final List shows = new LinkedList();
    private long timeout;
    private JdbcCalendar calendar = null;
```

```

private long minDelay;
private long maxDelay;

public AvailabilityFixture() throws Exception {
    calendar = new JdbcCalendar(new TestDataSource());
    calendar.afterPropertiesSet();
    shows.add(calendar.getShow(1));
    availabilityCheck = new CachingAvailabilityCheck();
    availabilityCheck.setCalendar(calendar);
    availabilityCheck.setBoxOffice(new DummyBoxOffice());
}

public void setTimeout(int timeout) {
    availabilityCheck.setTimeout(timeout);
}

public void setMinDelay(long minDelay) {
    this.minDelay = minDelay;
}

public void setMaxDelay(long maxDelay) {
    this.maxDelay = maxDelay;
}

```

我们感兴趣的是高速缓存算法的性能，而不是底层数据库访问，所以笔者在一个内部类中使用了BoxOffice接口的一个简单伪实现（同样，基于接口的设计证明在测试期间是非常方便的）。这始终返回相同的数据（我们对那些值不感兴趣，只对检索它们花费了多长时间感兴趣），进而在minDelay与maxDelay组件属性的值之间延迟了任意毫秒数。与该测试相关的方法简单地抛出一个UnsupportedOperationException异常。这比返回null好，因为我们会立即看到这些方法是否做了意外的事情。

```

private class DummyBoxOffice implements BoxOffice {
    public Reservation allocateSeats(ReservationRequest request)
        throws NotEnoughSeatsException,
               NoSuchPerformanceException,
               InvalidSeatingRequestException {
        throw new UnsupportedOperationException("DummyBoxOffice.allocateSeats");
    }

    public Booking confirmReservation(PurchaseRequest purchase)
        throws ExpiredReservationTakenException,
               CreditCardAuthorizationException,
               InvalidSeatingRequestException,
               BoxOfficeInternalException {
        throw new UnsupportedOperationException(
            "DummyBoxOffice.confirmReservation");
    }

    public int getFreeSeatCount(int performanceId, int seatTypeId)
        throws NoSuchPerformanceException {
        AbstractTest.simulateDelay(minDelay, maxDelay);
        return 10;
    }

    public int getFreeSeatCount(int performanceId)
        throws NoSuchPerformanceException {
        AbstractTest.simulateDelay(minDelay, maxDelay);
        return 30;
    }

    public int getSeatCount(int performanceId)
        throws NoSuchPerformanceException {
        return 200;
    }
}

```

要想使用BoxOffice的真正EJB实现，我们需要在EJB容器中运行测试，或通过一个远程接口访问EJB，而这对扭曲测试结果。如果没有使用EJB，我们可以在测试套件中简单地读取ticket-servlet.xml中的XML组件定义。在决定使用EJB之前，应该考虑EJB的使用在整个软件生存周期内所增加的复杂性；如果决定使用EJB，使用EJB确实通过声明性事务管理提供了实际价值，所以我们可以接受其他方面的较大复杂性。

我们可以利用下列属性文件来配置测试套件，这个文件类似于我们在上面见过的那个例子。

```
suite.class=com.interface21.load.BeanFactoryTestSuite
suite.name=Availability check
suite.reportIntervalSeconds=1
suite.longReports=false
suite.doubleFormat=##.#
suite.reportFile=<local path to report file>
```

至关紧要的框架测试套件定义具有并发运行的线程数量、每个线程要运行的测试遍数以及每个线程的最大暂停值：

```
suite.threads=50
suite.passes=40
suite.maxPause=23
```

我们把fixture对象设置为该框架的通用BeanFactoryTestSuite的一个组件属性：

```
suite.fixture(ref)=fixture
```

fixture对象也是一个组件，所以我们可以配置CachingAvailability对象的timeout，以及 JDBC模拟方法中的延迟：

```
fixture.class=com.interface21.load.AvailabilityFixture
fixture.timeout=10
fixture.minDelay=60
fixture.maxDelay=120
```

最后，我们设置每个工作者线程的各种属性：

```
availabilityTest.class=com.interface21.load.AvailabilityCheckTest
availabilityTest.(singleton)=false
```

首先，笔者把fixture.timeout属性设置为0来禁用高速缓存。这导致了每秒大约140次命中，其平均响应时间大约为360毫秒。把线程超时设置成1秒产生了显著的改善，其中每秒大约7000次点击且平均响应时间为7毫秒。进一步加大超时产生了20%或不到20%的改善。

到目前为止，还没有什么惊人之处。不过，笔者通过调查同步的结果有点感到惊讶。笔者首先使用非同步的java.util.HashMap替换HashTable。除非这产生了实质性的改善，否则付出更多的努力来开发更聪明的同步是毫无意义的。这种改善在所有现实的载荷级别上最多是10%至15%。只有通过尝试数百个用户同时请求同一个节日的信息，其间数据库响应时间慢得与事实不符并且超时为1秒（一种不可能的情形，因为Web接口不能把这种载荷提供给业务对象），HashTable同步才开始显著地降低吞吐量。笔者还了解到，在中等到繁重的载荷之下，通过getPerformanceWithAvailability()方法内的同步来消除上面提过的潜在竞争现

象，将性能降低了40%左右，进而使这种同步失去了吸引力。

稍做思考，就可以很容易解释这些结果。虽然在同步所关联的JVM中有一个不可避免的锁管理负荷，但同步对吞吐量的影响最终取决于执行被同步的操作花费了多长时间。由于散列表get和put操作花费极少的时间，所以同步的影响是相当小的（这非常像我们在第11章中所讨论的Copy-On-Write（写时复制）方法：同步只被应用于更新一个引用，不被应用于查找新数据）。

因此，这种最简单的方法（上述高速缓存器，它使用了同步化的java.utilHashTable）产生了远远超过业务需求的性能。

最后，笔者在同一个用例上又运行了JProbe，其间启用了高速缓存以了解发生了什么。需要注意的是，这是一个单请求仿形，因此没有反映并发访问中的同步代价。

图15.14所示结果表明，94%的执行时间现在被用在了再现该JSP视图上。只有通过切换到一种性能更高的视图技术，我们才可能略微改善一点性能。对Java应用代码的进一步修改不会产生任何好处。通常情况下，这样的结果（表明我们已经遇到了那些底层J2EE技术的限制）是非常鼓舞人心的。不过，检查该JSP视图来确定它有效率是值得的。在本案例中，换用视图技术没有多大价值，所以根本没有改善的余地。

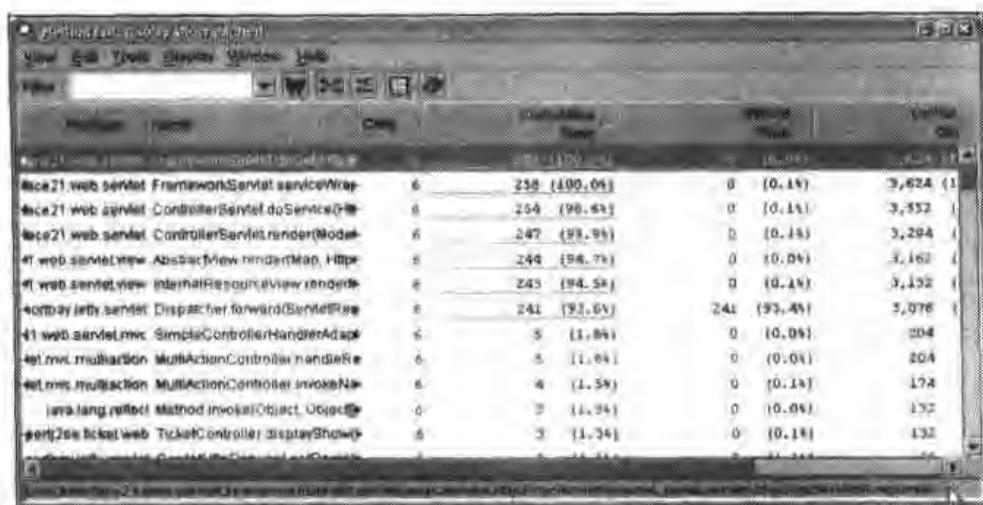


图15.14

该是停下来的时候了。我们已经超出了我们的性能目标，进一步的努力也不会产生任何值得的回报。

这个案例研究表明了一种以经验为基础的性能调整方法的价值，以及做“可能会管用的最简单事情”在性能调整中的价值。作为总体设计策略的一部分，我们把Web层控制器编写成了使用一个业务接口，而不是使用一个具体类，所以换用一个高速缓存实现十分容易。

通过使用Microsoft Web Application Tool，并利用一种以经验为基础的方法，我们证实了，在本案例中，最简单的高速缓存策略（通过同步来保证数据完整性）在所有条件（除了不大可能出现的高负载）下都比更高级的加锁策略管用。我们还证实了保证被显示的数据

不小于10秒没有任何困难，因而远远满足了业务对数据新鲜度的需求。通过使用JProbe仿形器，我们确认了最终版本在启用了高速缓存时的性能只被再现该JSP视图的工作所限制，进而暗示了根本没有进一步改善性能的余地。

当然，最简单的方法可能未必始终提供足够的性能。不过，本例表明不愿付出更大努力，以及只在必需时才付出更大努力是明智的。

分布式应用中的性能

分布式应用比所有构件都运行在同一个JVM中的应用复杂得多。性能在避免采用分布式体系结构的许多原因中是最重要的，除非它是满足业务需求的惟一方法。

使J2EE应用中的性能落空的最常见原因，是远程调用的多余使用——通常是以对EJB的远程访问的形式。一般说来，远程访问所强加的开销比J2EE应用中的其他任何操作所强加的开销都大得多。许多开发人员发觉J2EE是一个天生的分布式模型。事实上，这是一个误解。J2EE只是为必要时实现分布式体系结构提供了特别强大的支持。仅仅因为这个选择垂手可得，并不意味着我们应该总是做出这样的选择。

本节将讨论远程调用为什么代价如此之高，以及在我们必须实现一个分布式应用时，怎样最大限度地消除远程调用对性能的影响。

远程方法调用（RMI）的系统开销

普通Java类对同一个虚拟机中的对象按引用进行调用，但是对分布式应用中的EJB进行调用必须使用一个诸如IIOP之类的远程调用协议，按远程方式进行。客户不能直接引用EJB对象，并且必须使用JNDI来获得远程引用。EJB和EJB客户可能位于不同的虚拟机中，甚至可能位于不同的物理服务器中。这种重定向有时增强了可缩放性：因为一个应用服务器负责管理命名查找和远程方法调用，多个应用服务器实例能够协作地定向一个聚类中的通信，并提供故障切换支持。但是，如果应用设计不当，远程（而非本地）调用的性能代价必定极高。

EJB对远程客户的支持基于Java RMI（远程方法调用）。不过，适用于分布式调用的基础结构都有类似的开销。

Java RMI支持两种类型的对象：远程的和可串行化的。远程对象支持来自远程客户的方法调用；所谓远程客户是指运行在不同进程中，并被赋予了指向远程对象的远程引用（*Remote reference*）的客户。远程对象含有如下类：这些类实现java.rmi.Remote接口，而且这些类的所有远程方法除了被声明成抛出应用异常之外，还都被声明成抛出java.rmi.RemoteException。使用了远程接口的所有EJB都是远程对象。

可串行化对象实质上是指可以用在远程对象调用中的数据对象。可串行化对象必须含有实现java.io.Serializable标志接口并必须拥有可串行化字段（其他可串行化对象或原始类型）的类。可串行化对象被按值传递，意味着两份副本能被独立地修改，而且对象状态必须随同每个方法调用一起被串行化（被转换成一个流表示）和反串行化（从流表示中被重构出来）。可串行化对象用于分布式J2EE应用中的数据交换，用做参数，用于返回值，以及用于调用远程对象中的异常。

EJB对象或EJB主接口之类的远程对象上的方法调用始终需要一个网络往返行程，即从客户到服务器再从服务器返回到客户。因此，远程调用消耗网络带宽。不必要的远程调用消耗了应该为执行某些必需事情的操作（比如把数据传送到需要它的地方）而预留的带宽。

每个远程调用都会遇到编组（Marshal）和反编组（Unmarshal）可串行化参数的开销；编组和反编组是指如下过程：借助于这个过程，调用者把方法参数转换成一种能够在网络上传送的格式，而接收者装配对象参数。编组和反编组的系统开销都超过串行化和反串行化，它们的时间用来在网络上传送字节。编组和反编组的开销视所用的协议而定，该协议可以是IIOP，也可以是一个优化过的专有协议，比如WebLogic的T3或Orion的ORMI。J2EE 1.3应用服务器必须支持IIOP，但不必默认地使用它。

图15.15描述了远程方法调用中所涉及到的开销。

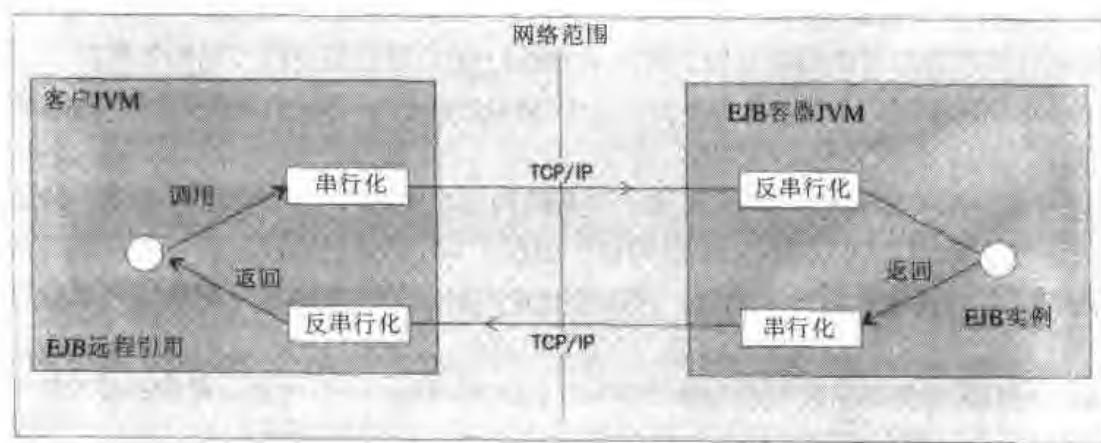


图15.15

这个开销意味着远程调用可能比本地调用慢1000多倍，即使那些相关的应用构件之间有一条快速的LAN连接。

远程调用的数量是分布式应用之性能的一个主要决定因素——也可能是最主要的因素，因为远程调用的开销十分大。

在下一节中，我们将看一看在设计分布式应用时怎么才能最大限度地降低远程调用的性能影响。

幸运的是，作为设计师和开发人员，我们有许多种选择。例如：

- 可以设法把我们的应用构造成最大限度地消除通过远程调用在体系结构层之间转移数据的需要。
- 可以设法把我们无法帮助转移的数据移到最少数量的远程调用中。
- 或许可以设法更有效地转移数据的单个块。
- 可以把构件放在同一个虚拟机中，以便内层调用不需要远程调用。
- 可以高速缓存来自远程资源的数据，以便最大限度地减少远程调用的次数。我们已经讨论过高速缓存，它在这种情形中将是特别有用的。

下面来逐个分析上述这些技巧。

最大限度地减少远程调用

性能增益的最大余地是在构造应用以便最大限度地减少将来所必需的远程调用个数方面。

应用划分

应用划分（Application partitioning）是指把一个分布式应用划分成主要体系结构层，并把每个构件分配给一个层的任务。在使用了EJB的J2EE Web应用中，这意味着把每个对象或功能构件分配给客户浏览器、Web层、EJB层或数据库当中的一个。一个“功能构件”不必是一个Java对象。例如，关系数据库中的一个存储过程可能就是应用的一个功能构件。

应用划分将决定该应用运行时所需要的网络往返行程的最大范围。网络往返行程的实际范围在有些部署配置中可能是较小的。分布式J2EE应用必须支持不同的部署配置，这意味着Web容器和EJB容器可能被集中在同一个JVM中，而这将会减少某些部署中的网络往返行程数量。

应用划分的主要目标，是保证每个体系结构层都有一个定义明确的责任。例如，我们应该保证分布式J2EE应用中的业务逻辑位于EJB层中，以便它能够在客户类型之间共享。不过，还有一个不可避免的性能问题：保证时间要求极高的频繁操作不用网络往返行程就能够被执行。正如我们已经从分析Java远程方法调用的代价中见过的，应用划分对性能会有极大的影响。糟糕的应用划分决定会导致“罗嗦”的远程调用——分布式应用中的最大性能敌人。

相对于应用划分来说，设计和性能的考虑因素往往要协调起来。过度的远程调用会使应用变复杂，并且容易出错，因此无论从设计角度还是从性能角度看，这都不是合乎需要的。不过，应用划分有时确实牵涉到权衡取舍。

正确的应用划分对性能会产生极大的影响。因此，考虑应用划分中各决策的性能影响是至关重要的。

最大的性能收获，将源自于最大限度地降低为满足进入请求而深入分布式J2EE栈的调用深度。

为给一个请求提供服务而需要进行的各调用深入分布式J2EE栈越深，最后的性能就会越差。尤其是，对于常见的请求类型，我们应该设法在距离客户尽可能近的地方为请求提供服务。当然，这要求权衡取舍：把这作为自己的主要目标，我们就很容易引起许多其他问题，比如复杂而易出错的高速缓存代码或过时的数据。

我们可以使用什么技巧来保证有效的应用划分呢？

最大的决定因素之一是我们要处理的数据来自哪里。首先，需要分析应用中的数据流程。数据可能从EIS层中的数据存储器流向用户，也可能从用户开始沿着应用的各层流动。

有3个策略对最大限度地减少往返行程次数是特别有用的：

- 把数据转移到我们操作它的地方。
- 把操作转移到数据所在的地方。Java RMI允许我们为此目的而转移代码及数据。我

们也可以转移EIS层资源内的某些操作，比如数据库，以便最大限度地减少网络通信量。

- 把具有很强亲和力（Affinity）的构件集中在一起。具有很强亲和力的对象经常彼此交互。

把数据转移到我们操作它的地方

最糟糕的情形是数据位于一个层中，而作用于它的操作却位于另一个层中。例如，如果Web层持有一个数据对象，并在它处理这个对象时对EJB层进行了许多调用，那么这种情形就会出现。一种较好的替代方法是通过把这个数据对象作为一个参数来传递，把它转移到EJB层，以便所有操作都在本地运行，只有一个必不可少的远程调用远程地运行。第10章中所讨论的EJB Command设计模式，就是这方面的一个例子。Value Object设计模式也在一个单独的远程调用中转移整个对象。

我们已经讨论过的高速缓存，是转移数据到我们操作它的地方的一个特例。在这种情况下，数据以相反方向被转移：从EIS层到客户。

把操作转移到数据所在的地方

该策略的例子是使用一个运行在关系数据库内的单独存储过程来实现一个操作，而不是执行EJB层与数据库之间的多个往返行程而用Java和SQL实现同样的逻辑。在某些情况下，这会极大地改善性能。存储过程的使用就是实际运用了一个受性能启发的应用划分决策，而且这种应用划分技巧不牵涉到折中，但可能有其他的缺点。它可能会降低应用在数据库之间的可移植性。不过，这种应用划分技巧除了适用于分布式J2EE应用之外，还适用于集中式J2EE应用。

另一个实际运用是做为一种确认用户输入有效性的可能方法。确认规则就是业务逻辑，因此自然应归入分布式应用的EJB层中，而不是归入Web层中。但是，每当有一个表单被提交时，就进行一次从Web容器到EJB容器的网络往返行程来确认用户输入是不经济的，尤其是在用户输出中的许多问题不必访问后端构件就能被识别出来的时候。

一种解决方法是让EJB层控制确认逻辑，并把确认代码转移到Web层内的一个实现了议定确认接口的可串行化对象中。这个确认者对象只需要在网络上被传递一次即可。由于Web层已有该确认者接口的类定义，所以这个实现类只需EJB层在运行时提供。然后，确认者就可以在Web层内被本地调用，而远程调用只对少数需要访问数据的确认操作来说才是必不可少的，比如检查一个用户名是唯一的。由于本地调用比远程调用快得多，所以该策略可能比调用EJB层来执行确认有更高的性能，即使EJB层需要再次执行确认（也是本地的）来确保无效输入绝不会导致数据更新。

现在来看一看该策略在实际运用中的一种情况。请设想有这样一个需求：确认一个用户对象含有电子邮件地址、密码、邮政编码、用户名等4个属性。在一个过于简单的实现中，Web层控制器可能在一个远程EJB上调用一个方法来依次确认这些属性的每一个，如图15.16所示。

毫无疑问，这种方法将使性能极差，因为需要过多的高代价远程调用来确认每个用户对象。

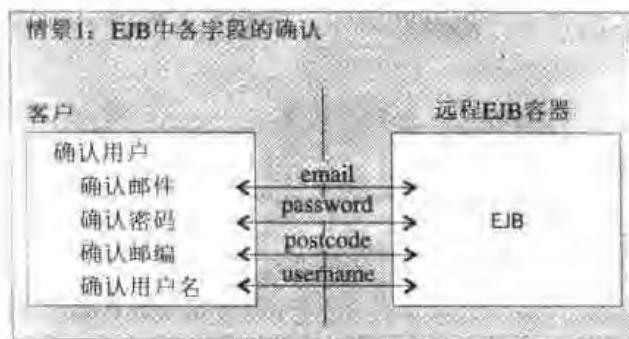


图15.16

一种更好的方法是将用户数据转移到我们处理它的地方（就像上面所描述的那样）：使用一个可串行化的值对象，以便用户数据能在一个单独的远程调用中被传送给EJB服务器，以及确认所有字段的结果能被返回。这种方法如图15.17所示。

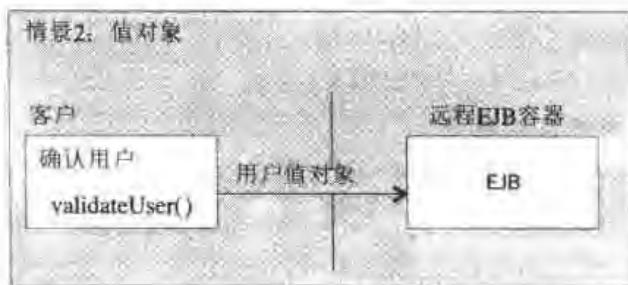


图15.17

这将实现一个巨大的性能改善，尤其是在有许多要确认的字段时。仅执行一个远程方法调用，即使该调用需要传递较多的数据，也将会比执行许多精细的远程方法调用快得多。

但是，让我们假设只有用户名字段的确认才需要数据库访问（以检查提交的用户名还没有被另一个用户占用），而且所有其他确认规则都能被应用在该客户上。在这种情况下，我们可以应用上述方法：借助于一个在该应用启动时从EJB层中获得的确认类将确认代码转移到该客户那里。在该应用运行时，客户端确认者实例不必调用EJB就可以确认大部分字段，比如电子邮件地址和邮政编码。确认每个用户对象将只需进行一次远程调用（以确认用户名值）。这种方法如图15.18所示。

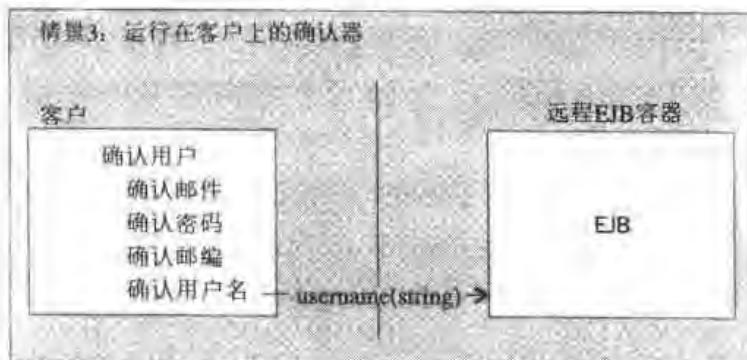


图15.18

由于在每个用户对象的确认期间只有一个串值需要被串行化和反串行化，所以这种方法将比值对象方法表现得更好，因为值对象方法需要从线路中传送一个较大的值对象。但是，这种方法依然允许EJB实现从客户代码中隐藏确认规则。

更彻底地执行“把确认代码转移到它所处理的数据处”这一原理，是把确认逻辑转移到运行在浏览器里的JavaScript中，以消除在拒绝某些无效提交之前与服务器进行通信的需要。但是，这种方法有其他一些缺点，这些缺点通常妨碍了它的使用。

在第12章中，我们已经利用实际示例讨论过对送到Web应用的输入进行确认的问题。

合并远程调用

由于远程调用的串行化和编组开销，最大限度地减少远程调用的数量常常是有意义的，即使这意味着必须随同每个调用一起传递更多的数据。

方法调用合并的一个经典方案是我们已在上面的例子中见过的Value Object设计模式。与其对一个远程EJB做多个精细调用来检索或更新数据，倒不如从一个单独的大块获得器中返回一个单独的可串行化“值对象”，并把它传递给一个单独的大块设置器方法，在一个调用中检索或更新所有字段。Value Object设计模式是“远程对象不应该含有迫使客户进入‘罗嗦’访问的接口”这个通用规则的一个特例。

最大限度地减少远程调用的数量将会改善性能，即使需要随同每个远程调用一起传递更多的数据。

对EJB的远程接口所执行的每个操作，都应该执行一个有效的操作。通常，它将实现一个单独的用例。深入到EJB容器中的罗嗦调用通常会导致创建许多事务上下文，这种创建也浪费资源。

有效转移数据

虽然最大限度地减少远程调用的数量通常需要全力关注的、最有利可图的区域，但有时，我们也可以减少在网络上传送交换数据的开销。

串行化的优化

由于远程调用中的所有参数都必须被串行化和反串行化，所以串行化的性能是至关重要的。虽然核心Java提供了透明的串行化支持，而且这种支持常常是令人满意的，但有时，通过把下列各种技巧运用于个别可串行化对象，我们可以投入更多的精力来保证串行化是尽可能高效的。

瞬态数据

在默认情况下，可串行化对象的所有字段（transient和static字段除外）都将被串行化。因此，通过把“短命”字段（比如能够用其他字段计算出的字段）标记为transient，我们可以减少必须被串行化与反串行化并通过网络来传递的数据量。

可串行化对象的“短命”字段都可以被标记为transient。但是，使用这种方法要特别小心。瞬态字段的值在反串行化之后将需要被复位：不这么做是造成微妙错误的一个潜在原因。

基元字段

我们还可以用一种更有效的方式表示所需要的数据。与串行化对象相比，串行化基元字段速度快得多，需要少得多的网络带宽。因此，任何能被表示为基元类型的对象都是可串行化对象中更有效表示的最佳候选者，进而节省了内存，还优化了串行化。经典例子是java.util.Date，它通常可以被一个以毫秒为单位的long系统时间取代。如果可以用可串行化对象把Date对象表示为long，我们通常就会看到串行化性能得到显著改善，串行化后对象大小得到减小。表15.3给出了以3种不同方法串行化10 000个日期中的数据之后的结果。

表15.3

10 000个串行化，被写到一个磁盘文件上	时间 (ms)	文件的大小 (KB)	未优化情况下 的时间百分比	未优化情况下的数 据大小百分比
存有一个java.util.Date对象	370	225	100%	100%
存有long系统时间的对象	110	137	30%	61%
long系统时间的数组	5	79	1%	35%

我们使用了下列接口的两个实现：

```
interface DateHolder extends java.io.Serializable {
    Date getDate();
}
```

第一个实现DateHolder把日期保存为一个java.util.Date对象。第二个实现LongDateHolder使用一个long系统时间表示日期。

第三行数据是废除那10 000个对象，而使用由含有相同信息的10 000个long系统时间所组成的一个数组所产生的结果。这种方法带来了性能上的巨大改善。

当然，这种优化可能会有意料之外的结果。如果需要与一种基元类型之间的重复转换，这种优化的开销在价值上可能会超过串行化好处。另外，存在一种使代码变得过分复杂的危险。大致说来，这种优化与优良的设计是相符的，并常常会证明它是一种值得做的优化。

基元类型的串行化与反串行化速度比对象要快得多。如果可能用一个基元表示取代一个对象，那么就可能非常显著地改善串行化性能。

串行化的无问题性

认识到使用标准串行化时不必担心什么也是很重要的。

开发人员有时很想知道，如果他们串行化一个含有许多方法的“人”对象，会发生什么。串行化过程不传送.class文件，所以可执行代码不会浪费带宽。类需要由接收者来装入，所以类可能需要通过网络被传送一次，如果它不是本地可获得的；这将有少量的一次性开销。

串行化一个类且这个类又是一个继承性分级结构的一部分（比如串行化一个类C，类C扩展类B，而类B又扩展类A），将会给串行化过程增加少量的开销（串行化过程基于反射，所以将需要遍历继承性分级结构中的所有类定义），但是不增加将被写出的数据量，除了那些超类的附加字段之外。

类似地，静态字段也不膨胀一个类的串行化表示的大小。这些字段将不被串行化。发送者和接收者将维护静态字段的各自副本。在每个JVM装入这个类时，静态初始化器将独立地运行。这意味着静态数据会在远程JVM之间失去同步——EJB规范中对静态数据的使用加以限制的原因之一。

默认串行化实现的一种重要优化涉及到被串行化到同一个流的同一个对象的多个副本。每个被写到这个流的对象都被赋予一个句柄（Handle），进而意味着对该对象的后续引用在输出结果中可以由这个句柄，而不是由对象数据来代表。当这些对象在客户上被实例化时，将构造这些引用的一个可靠副本。这在串行化和反串行化时可能会产生极大的好处，而且在对象图的对象之间有多条连接的情况下可能会节省网络带宽。

自定义串行化

在某些情况下，有可能比默认实现更有效地工作（与串行化与反串行化速度及串行化数据的大小有关）。但是，和我们刚才所讨论的选择不同（它们实现起来都相当简单），这不是一个可以轻松获得的选择。替代这个默认的串行化机制需要自定义编程来代替标准行为。自定义代码还需要随着对象（可能还包括继承性分级结构）的发展被保持最新。

这里有两个可能的技巧。

第一个技巧涉及到在可串行化类中实现两个方法。需要注意的是，这两个方法不是某个接口的一部分：它们对核心的Java串行化基础结构有特殊含义。这两个方法的签名分别是：

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;  
private void readObject(java.io.ObjectInputStream in)  
    throws IOException, ClassNotFoundException;
```

`writeObject()`方法保存对象自身内所存有的数据。该数据不包括超类状态，超类状态仍被自动处理。这个数据只包括相关对象，这些对象的引用被保存在对象自身中。

当对象从一个`ObjectInputStream`中被读取时，得到调用的是`readObject()`方法，而不是一个构造器。这个方法必须用从对象输入流中读取的数据来设置类字段的值。

这两个方法将直接使用Serialization API将对象写到串行化基础结构所提供的ObjectOutputStream，并从ObjectInputStream中恢复对象。这意味着需要更多的实现工作，但这个类控制它的数据在输出流中的表示格式。保证字段以正确的顺序被读写是十分重要的。

在某些情况下，替代readObject()和writeObject()方法会提供显著的性能增益，而在其他情况下，只提供极小的性能增益。它消除了标准串行化过程的反射开销（但可能不是特别人），而且在字段被持久保存时，可能允许我们使用一种更有效的表示。许多标准库类都实现了readObject()和writeObject()方法，其中包括java.util.Date、java.util.HashMap、java.util.LinkedList以及大多数AWT与Swing构件。

在第二个技巧中，如果一个对象实现了java.io.Externalizable接口，它可以获得对串行化与反串行化的完全控制权。这个接口是Serializable标志接口的一个子类，它含有两个方法：

```
public void writeExternal(ObjectOutput out) throws IOException  
public void readExternal(ObjectInput in) throws IOException,  
    ClassNotFoundException
```

与实现了readObject和writeObject的Serializable对象相反，Externalizable对象完全负责保存和恢复它的状态，包括超类状态。readExternal和writeExternal的实现将和readObject和writeObject的实现使用同一个API。

虽然在可串行化对象被实例化为标准反串行化的结果时避免了构造器调用，但这种情况不适用于可具体化对象。当一个可具体化类的实例用这个类的可串行化表示重新构造时，这个类的无变元构造器在readExternal()被调用之前先被调用。这意味着我们确实需要考虑无变元构造器的实现是否可能会证明它对可具体化对象是不经济的。这还将包括连锁式的无变元构造器，即运行时被隐含地调用的无变元构造器。

可具体化类可能会从自定义的串行化中提供最大的性能增益。笔者曾经在实践中见过50%左右的增益，这个增益对应用中交换最频繁的可串行化对象来说可能是值得的，如果证明串行化这些对象是一个性能问题。但是，可具体化类需要最大量的实现工作，并且会导致最大的后续维护要求。如果一个类有一个或多个超类，把该类变成可具体化的通常是不明智之举，因为对这些超类的修改可能会破坏串行化。

然而，这个考虑因素不适用于许多不是继承性分级结构一部分的值对象。还需要注意的一点是，如果一个可具体化类未能识别出指向同一个对象的多个引用，那么这个可具体化类可能会比一个普通可串行化类提供更糟糕的串行化性能：请记住，串行化的默认实现给每个被写到OutputStream的对象都分配一个以后能被引用的句柄。

JDK的技术资料含有关于高级串行化技巧的详细信息。笔者建议读者在尝试使用自定义串行化来改进分布式应用的性能之前，先仔细读一读这个资料。

对串行化过程的控制是应该仔细权衡的优化的一个示例。这种控制在大多数情况下会带来性能好处；如果我们正在处理简单对象，最初的实现难度将不会太大。但是，这种控制会降低代码的可维护性。例如，我们必须保证持久性字段以相同的顺序被写出

和读入。我们必须小心而又正确地处理对象关联。我们还必须考虑持久性类可能属于的类分层结构。例如，如果持久性类扩展一个抽象基类，并且一个新字段被增加到它上面，那么我们就需要修改自己的可具体化实现。串行化的标准处理允许我们忽略所有这些问题。

需要注意的是，给一个应用的设计或实现增加复杂性来实现优化不是一个一次性的操作。复杂性是永恒的，而且可能会引起无休止的维护成本。

其他数据转移策略

我们未必总是希望以串行化对象的形式转换数据。无论选择何种方法来转换大量数据，我们都会遇到转换开销和网络带宽的类似问题。

XML有时被推荐为数据交换的一种替代方案。在整个过程范围内使用串行式DOM文档来转换数据，可能比在发送者中把该文档转换一个串并在接收者中语法分析这个串要慢得多。串行式DOM文档可能会比同一个文档的串表示大得多。Bruce Martin在JavaWorld的一篇文章中讨论了这些问题（请参见<http://www.javaworld.com/javaworld/jw-02-2000/jw-02-ssj-xml.html>）。

虽然DOM文档可能是可串行化的，但也没有任何保证。例如，Xalan 2语法分析器（WebLogic 6所使用的语法分析器的一个变种）允许DOM文档的串行化，而Orion 1.5.2所使用的Crimson语法分析器的版本却不允许。`org.w3c.dom`接口自身不是可串行化的，尽管大多数实现是可串行化的。确实保证可串行化的一种选择是使用JDom API，而不是使用DOM API来表示文档：JDom文档是可串行化的（关于这个可选Java XML API的进一步信息，参见<http://www.jdom.org>站点）。

因此，笔者不建议在网络范围内使用XML来交换数据。正如笔者在第6章中所说的，笔者认为XML最好保持在J2EE应用的范围内。

另一种可能性是在通用Java对象中转移数据，比如`java.util.HashMap`或`javax.sql.RowSet`对象。在这种情况下，考虑串行化与反串行化这些对象的代价及它们的串行化形式的大小是很重要的。就实现`writeObject()`和`readObject()`方法的`java.util.HashMap`对象来说，该容器本身几乎不增加开销。简单的计时可以用来确定其他容器的开销。

但是，以一个通用容器（比如一个RowSet）的形式将数据从EJB层传给客户的想法，从设计角度来看是毫无吸引力的。EJB层应该为客户通信提供一个强类型的接口。此外，处理来自EIS层的原始数据（比如一个RowSet）可能会导致部分数据的废除，而原始数据却需要被完整地发送给客户。如果始终把所有数据都发送给客户，并让客户自己去处理这些数据，上述部分数据废除的情况就不会发生。最后，该客户变得依赖于它在其他方面不会使用的`java.sql`类。

把构件集中在同一个JVM中

有一种不必编写一行代码就能消除分布式应用的大部分（而不是全部）开销的方法：把各种构件都集中布置在同一个服务器上，以便它们运行在同一个JVM上。就Web应用的情况来说，这意味着把Web层和EJB层都集中布置在同一个J2EE服务器中。大多数J2EE服务器都检测集中式的布置，并能够使用本地调用来取代远程调用（在大多数服务器中，这种优化

被默认地启用）。这种优化避免了串行化的开销和远程调用协议。调用者和接收者都将使用一个对象的同一份副本，意味着串行化是多余的。不过，有可能证明比EJB通过本地接口所做的调用慢，因为容器需要规避调用语义。

这种方法不适用于Swing客户之类的分布式应用客户，这类客户不能作为一个服务器进程运行在同一个JVM中。但是，这种方法对使用了远程接口的EJB来说可能是最常用的部署选择。

笔者已经在第6章的“假远程接口”一节中讨论过这种方法。顾名思义，只有当我们知道自己需要一个分布式体系结构，而应用至少最初能够控制它的被集中布置在同一个JVM中的所有构件时，这种方法才会有效。有这样一个实际的危险：集中布置会导致开发人员只能开发带有RMI语义的应用，而RMI语义又依赖于引用调用，因此集中布置在真正的分布式环境中是无法做到的。

如果集中布置始终是惟一的部署可选项是明确无疑的，就要考虑不使用EJB（至少不使用EJB远程接口），并坚决地废除远程访问问题。灵活的部署是EJB所提供的一个关键部分；当不需要时，EJB的使用可能就不会有什么好处。

Web层性能问题

我们已经讨论过Web层会话管理的性能可缩放性的意义。现在来考虑我们迄今为止还未涉及过的几个Web层特有性能问题。

视图性能

在第13章中，我们见过如何使用几种不同的视图技术来生成同一个示例视图。我们考虑了创作意义和可维护性，但除了一般性考虑之外，没有考虑性能问题。

现在来看一看对我们已经讨论过的部分视图技术运行Microsoft Web Application Stress Tool的结果。

请注意，为了测试目的，笔者稍微修改了控制器代码，以避免需要一个预先存在的用户会话，并使用一个统一的Reservation对象来提供用于所有请求的模型，从而避免访问RDBMS的需要。这在很大程度上使得该测试变成了一个视图性能测试，消除了联系业务对象的需要；同时，这也避免了另外一个棘手的问题：保证数据库中有足够的空余座位可供应，以满足数百个并发请求。

被测试的5种视图技术分别是：

- JSP，使用了JSTL；
- Velocity 1.3；
- XMLC 2.1；
- XSLT，使用了Reservation共享对象的飞行“变换”；
- XSLT，使用了同一个格式表，但又使用了一个被高速缓存的org.w3c.dom.Document对象（一个人造测试，但也是一个能使反射型变换的负载与XSLT型变换的负载进行比较的测试）。

每个视图生成相同的HTML内容。WAS和应用服务器都运行在同一台计算机上。但是，操作系统的负载监视功能显示，WAS所消耗的资源不足以影响测试结果。XSLT引擎是Xalan。

图15.19给出了运行连续请求页面的100个并发客户所得出的结果。

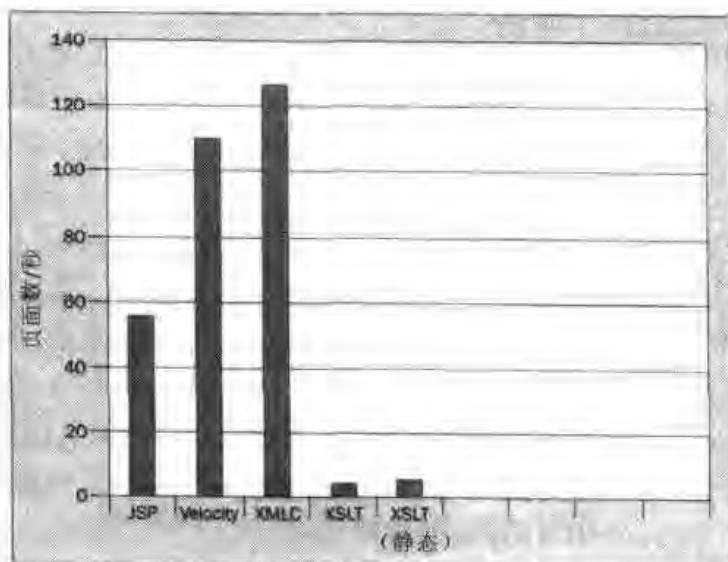


图15.19

虽然这些实际数字会随着不同的硬件而有所不同，并且不应该被当做一个生产环境中的性能指标，但是它们之间的差距是一种比较有意义的度量。

笔者对结果的范围之大感到惊讶，把那些测试运行了几次，才确信它们不是反常的。JSP达到了每秒54页左右；Velocity达到了每秒112页；XMLC显然是一个赢家，达到了每秒128页；而两种XSLT方法则慢了许多，分别是每秒6页和7页。

最后的结论是：

- 视图技术会支配着总体应用性能，尽管效果将会较小，如果需要由业务对象做更多的工作来满足每个请求。有趣的是，在同一个用例中，XSLT与其他视图技术之间的性能差距明显大于包括了到数据库的一次往返行程。虽然笔者已经预计到会证明XSLT是最慢的视图技术，但笔者和许多开发人员一样，往往自以为在一个用例中，访问一个数据库几乎始终是最慢的操作。因此，这个测试用示例说明了以实验为基础做性能决策的重要性。
- 不应该假设JSP在工作性能上比其他视图技术好（因为JSP页面由Web容器编译到服务器小程序），就认为它就是一个自然的选择。已证明XMLC比JSP好了两倍以上，但它束缚于一个非常不同的创作模型。在笔者看来，Velocity才是真正的赢家：JSP的感觉特性优点是赞成宁选JSP而不愿选一种像Velocity那样的简单技术的关键理由之一。
- 使用XSLT来生成动态内容的整个页面是非常慢的——以致在许多情况下不能成为一个可选项。XSLT转换的代价远远大于使用反射来飞行地“变换”Java组件数据的代价。格式表的效率对是否使用“变换”似乎有较大的影响。从“基于规则的”格式表变换到“填表式”格式表把性能提高了30%左右，尽管取消使用Java扩展函数似乎不会产生多大的改进。

笔者在这里所给出的数字可能会随着应用服务器而有所不同：笔者不情愿地得出了XMLC和Velocity将始终比JSP快几乎一倍的结论。Jetty所使用的Jasper JSP引擎可能是相当慢的。笔者曾经在Orion 1.5.3中见过表明所有3种技术有同等性能的结果，但无可否认的一点是使用了Velocity和XMLC的较旧版本。

JSP结果在同等硬件上与Orion非常相似，而新的Velocity和XMLC结果则快得多这一事实，意味着Velocity和XMLC在各自的版本1.3和2.1中已得到了明显的优化。但是，这些结果明确表明，我们不能简单地认为JSP是速度最快的选择。还需要注意的是，一个不同的XSLT实现可能会产生更好的结果，尽管有很充分的原因说明XSLT的性能为什么不太可能接近于其他技术的性能。

虽然充分了解使用不同产品可能造成的影响是十分重要的，但是我们不应该错误地认为，由于J2EE应用中有那么多的变数（应用服务器、API实现、第三方产品等），所以基准测试是毫无意义的。在任何一个项目的早期，我们都应该选择一个要使用的基础结构，并基于它的性能和其他特征来制定决策。为了追求J2EE可移植性，推迟做这样的决策可能会证明是高代价的。因此，就目前来说，对示例应用惟一有意义的度量标准是使用了Xalan的JBoss/Jetty的性能。

使用HTTP能力的Web高速缓存

我们已经讨论过在J2EE应用中高速缓存数据的各种选择。还有另外一个重要的高速缓存选择，这个选择基于HTTP的各种能力，能由客户浏览器来高速缓存内容，或将内容高速缓存在客户浏览器与应用服务器之间。

可以防止客户请求转到服务器，或者说可以减小后续请求对服务器的影响的Web高速缓存器有两种：浏览器高速缓存器和代理高速缓存器。

浏览器高速缓存器是独立的高速缓存器，由客户浏览器保存在用户的本地硬盘上。例如，在笔者的Windows XP计算机上查看C:\Documents and Settings\Rod Johnson\Local Settings\Temporary Internet Files目录将会显示数兆字节的临时文档和Cookie。这些文档中的一些文档含有GET参数。

代理高速缓存器是共享高速缓存器，由Internet服务供应商或机构代表数百或数千个用户来保存。

这些高速缓存器之一能否满足后续请求，取决于标准HTTP首部值（下文讨论）是否存在。如果用户的浏览器高速缓存器能够满足后续请求，响应将几乎是即时的。用户还可能会体验到更快的响应，如果内容能够由机构防火墙内的一个代理高速缓存器来返回，或由Internet服务供应商维护，尤其是当Web站点所运行在的那个（些）服务器距离用户很远时。这两种高速缓存形式可能会极大地减轻Web应用上的负载，并总体上减轻Internet基础结构上的负载。例如，请设想这样一个情景：如果来自某个站点（比如一个奥林匹克运动会或足球世界杯站点）的内容没有被代理，情况将会怎么样呢？

HTTP高速缓存在Java Web应用中常常遭到忽视。这是非常不幸的，因为Servlet API提供了为充分利用它而需要的全部支持。

高速缓存器控制HTTP头部

HTTP 1.0和1.1协议定义了几个高速缓存器控制（cache control）头部选项：最重要的是，HTTP 1.0 Expires头部和HTTP 1.1 Cache-Control头部。请求与响应头部位于文档内容的前面。请求头部必须在内容被输出给响应之前先被设置。

如果没有高速缓存器控制头部，大多数高速缓存器将不存储对象，这意味着后续请求将直接转到原点服务器（origin server）（在目前上下文中，指生成了该页面的那个J2EE Web容器）。但是，这没有得到HTTP协议的保证；如果没有明确地指定高速缓存行为，我们对处于原点服务器与客户浏览器之间的浏览器或代理高速缓存器是无能为力的。这两个高速缓存器的各自默认行为可能是不同的。

HTTP 1.0 Expires头部设置一个日期，该日期之前的已缓存响应应该被认为是有效的。这个日期使用GMT格式，而不是使用原点服务器或浏览器地区的格式。幸运的是，Servlet API使我们省去了设置这一信息的细节。该头部看起来将像下面这样：

```
Expires: Mon, 29 Jul 2002 17:35:42 GMT
```

如果高速缓存器接收到申请给定URL并包含GET参数数据的请求，则可以简单地用已缓存的数据来满足它们。不必再次与原点服务器联系，进而保证最快速地响应客户，并最大限度地减轻原点服务器上的负载。相反，要想保证内容不被高速缓存，则可以把Expires日期设置成过去的一个日期。

HTTP 1.1 Cache-Control头部是比较高级的。它包含了几个能够被单独或组合使用的指令（Directive）。最重要的指令如下所示：

- max-age指令以秒为单位指定响应应该被认为有效的未来时间，在这段时间内无需与原点服务器联系。这是一个和HTTP 1.0 Expires头部相同的基本功能度。
- no-cache指令阻止代理高速缓存器提供已缓存的数据，如果没有向原点服务器成功地重新生效。下文将讨论重新生效这个重要概念。
- must-revalidate指令迫使代理高速缓存器向原点服务器重新生效，如果内容已经期满（否则高速缓存器可以自由行事）。这个指令一般与no-cache指令结合使用。
- private指令指定该内容是为单个用户预定的，因而不应该由共享式代理高速缓存器来高速缓存。该内容仍可以由浏览器高速缓存器来高速缓存。

通过发布一个新请求去申请一个含有If-Modified-Since请求头部的内容，高速缓存器可以重新生效该内容。如果该资源已经发生了变化，服务器将会生成一个新页面。如果该资源自从本日期以来还没有发生过变化，服务器可以生成一个像下面这样的响应，该响应是这个已缓存内容的一个成功的重新生效。

```
HTTP/1.1 304 Not Modified
Date: Tue, 30 Jul 2002 11:46:28 GMT
Server: Jetty/4.0.1 (Windows 2000 5.1 x86)
Servlet-Engine: Jetty/1.1 (Servlet 2.3; JSP 1.2; java 1.3.1_02)
```

当然，任何效益收获都取决于服务器能够在比它生成该页面所花费的时间少得多的时间内做各种必要的重新生效检查。不过，正如我们在上面所看到的，仅仅再现视图的代价就如此之大，以至于这种效益收获几乎是确定无疑的。当一个资源很少发生变化时，使用重新

生效不仅可以促进网络通信量和原点服务器上负载的显著下降，尤其是当该页面有很多的内容时，而且又在很大程度上保证了用户能看到最新的内容。

例如，请设想一个繁忙新闻网站的主页。如果内容变化无规则，但一天只有几次，那么使用重新生效将保证客户能够看到最新信息，而又极大地减轻原点服务器上的负载。当然，如果内容变化可预测，我们可以简单地把高速缓存器期满日期设置成下一个更新日期，并跳过重新生效。下文将讨论Servlet API如何处理If-Modified-Since请求。

至少从理论上说，这些高速缓存器控制头部应该能控制Back按钮使用时的浏览器行为。但是，需要重点注意的是，代理高速缓存器（尤其是浏览器高速缓存器）的行为可以多种多样。高速缓存器控制头部在实践中未必始终得到尊重。不过，即使它们不是100%地有效，当内容很少发生变化时，它们仍能产生巨大的性能收获。

这是一个复杂但又重要的题目，所以笔者在这里只能提供一个基本介绍。关于Web高速缓存和如何使用HTTP头部的一个优秀教程，请参见http://www.mnot.net/cache_docs/。关于HTTP 1.1高速缓存头部的一个完整定义，请参见<http://www.w3.org/Protocols/rfc2616/rfc2616-section14.html>。

HTTP POST方法是Web高速缓存器的大敌。POST非常排斥高速缓存。最好是给可高速缓存的页面使用GET方法。

有时，HTML元标志（即HTML文档的<head>段内的标志）用于控制页面高速缓存。这不如使用HTTP头部那么有效。通常，它将只影响浏览器高速缓存器的行为。代理高速缓存器不一定会读取文档中的HTML标志。类似地，不能依靠一个Pragma:no-cache头部来防止高速缓存。

使用Servlet API控制高速缓存

要想设法保证所有浏览器与代理高速缓存器有正确的行为，一种不错的方法是同时设置这两种高速缓存器控制头部。下面这个阻止高速缓存的例子说明了我们如何使用Servlet API所提供的抽象来最小化做这件事的复杂性。请注意使用 setDateHeader()方法来隐藏日期格式化工作的用法。

```
response.setHeader("Pragma", "No-cache");
response.setHeader("Cache-Control", "no-cache");
response.setDateHeader("Expires", 1L);
```

如果知道内容将保持有效的时间长度，可以使用像下面这样的代码。seconds变量是该内容应该被认为有效的时间长度（以秒为单位）。

```
response.setHeader(
    "Cache-Control", "max-age=" + seconds);
response.setDateHeader(
    "Expires", System.currentTimeMillis() + seconds * 1000L);
```

要想看一看生成的那些头部，Telnet到一个Web应用就可以做到。例如，笔者可以使用下列命令让Telnet与我的JBoss/Jetty安装对话：

```
telnet localhost 8080
```

下列HTTP请求（后跟两个回车）将检索内容，该内容前面有那些头部：

```
GET /ticket/index.html
```

我们的第二个代码示例将产生下面这样的头部：

```
HTTP/1.1 200 OK
Date: Mon, 29 Jul 2002 17:30:42 GMT
Server: Jetty/4.0.1 (Windows 2000 5.1 x86)
Servlet-Engine: Jetty/1.1 (Servlet 2.3; JSP 1.2; java 1.3.1_02)
Content-Type: text/html;charset=ISO-8859-1
Set-Cookie: jsessionid=1027963842812;Path=/ticket
Set-Cookie2: jsessionid=1027963842812;Version=1;Path=/ticket;Discard
Cache-Control: max-age=300
Expires: Mon, 29 Jul 2002 17:35:42 GMT
```

Servlet API通过javax.servlet.http.HttpServlet类的受保护的getLastModified()方法集成了对重新生效的支持，HttpServlet类也是由大多数应用服务器小程序扩展的一个类。Servlet API Javadocs对getLastModified()方法如何工作解释得不是很清楚，因此下列解释可能是很帮助的。

当一个GET请求进入时（不存在用于POST请求的重新生效支持），javax.servlet.HttpServlet service()方法调用getLastModified()方法。该方法在HttpServlet中的默认实现返回-1——指出该服务器小程序不支持重新生效的判别值。但是，getLastModified()方法可以由子类替代。如果一个替代的getModified()方法返回一个有意义的值，服务器小程序就检查该请求是否含有一个If-Modified-Since头部，如果它含有，就比较日期。如果该请求的If-Modified-Since日期迟于服务器小程序的getLastModified()日期，HttpServlet就返回一个304响应代码（Not Modified）给客户。如果getLastModified()值是比较新近的，或者没有If-Modified-Since头部，就重新生成内容，并且Last-Modified头部被设置成最近的修改日期。

因此，要想允许重新生效支持，我们必须保证自己的服务器小程序替代getLastModified()方法。

对MVC Web应用的影响

设置高速缓存器期满信息应该是Web层控制器的责任，而不是模型或视图构件的责任。但是，如果别无选择，我们可以在JSP视图中设置头部。

使用MVC方法的几个缺点之一是：它使得使用重新生效更困难，因为独立控制器（不是控制器服务器小程序）知道上一次修改日期。包括Struts、Maverick和WebWork在内的大多数MVC框架，简单地忽略这个问题，使支持If-Modified-Since请求变得不可能。使用这些框架时的唯一选择是在服务器小程序前面使用一个Servlet筛选器来保护这样的资源。但是，这是一种笨方法，它要求重复应该由一个框架来提供的基础结构。这是真正的问题，因为我们有时可以通过允许重新生效来获得性能上的极大回报。

第12章中所讨论的框架允许对If-Modified-Since请求的有效响应。支持重新生效的控制器对象，都能够实现com.interface21.web.servlet.LastModified接口，该接口只含有一个方法：

```
long getLastModified(HttpServletRequest request);
```

因此，只有定期地处理更新内容的控制器才需要担心支持重新生效。这个方法的约定和HttpServlet getLastModified()方法的约定相同。返回值 -1 将保证内容始终被重新生效。

com.interface21.web.servlet.ControllerServlet进入点将使用发送请求所使用的同一个映射，把重新生效请求发送给适当的控制器。如果这个受映射的控制器没有实现LastModified接口，控制器服务器小程序将始终使该请求得到处理。

一个getLastModified()实现的特征必须是：

- 它运行得非常快。如果它的代价与生成内容的代价几乎相同，对getLastModified()方法的多余调用实际上可能会降低应用的速度。幸运的是，这在实践中通常不是一个问题。
- 它返回的值在内容被生成时不变化，但在它背后的数据发生变化时变化。这就是说，为一个没有发送If-Modified-Since请求的客户生成一个新页面不应该更新上一次修改日期。
- 没有相关的会话或身份鉴别。通过了鉴别的页面可以被特殊地标记为可高速缓存，尽管这不是平常令人想要的行为。如果读者需要这么做，请参见上面已引用过的W3C RFC。
- 如果一个long型的上一次修改日期被存储在控制器或控制器所访问的一个业务对象中，线程安全性得到了保持。long类型不是原子的，所以我们可能需要同步。可是，与生成多余的内容相比，同步一个long日期访问的代价远远不及一次性能命中。

对于我们示例应用的TicketController类中所扩展的com.interface21.web.servlet.mvc.multiaction.MultiActionController类，重新生效必须在方法级上，而不是在控制器上被支持（请记住，这个类的扩展可以处理多种请求类型）。因此，对于每个请求处理方法，我们都可以在选地实现一个上次修改方法，并把它命名成处理器方法名，后跟“LastModified”后缀。这个方法应该返回一个long日期，并接受一个HttpServletRequest对象作为变元。

示例应用中的欢迎页面

示例应用提供了使用重新生效的好机会，因而也提供了显著地改进性能的可能性。

“Welcome”页面上所显示的数据（艺术流派和节目）很少发生变化，但它的变化是不可预测的。修改由JMS消息来提示。要获得该数据上一次发生变化的日期，可以查询提供了它所显示的参考数据的com.wrox.expertj2ee.ticket.reference.data.Calendar对象。因此，我们有了一个实现getLastModified()方法的现成值。

com.wrox.expertj2ee.ticket.web.TicketController类中的欢迎页面请求处理器方法具有下列签名：

```
public ModeAndView displayGenresPage(
    HttpServletRequest request,
    HttpServletResponse response);
```

因此，按照上述给任选的重新生效方法签名的规则，我们需要给这个方法名附加上“getLastModified”后缀，以创建MultiActionController超类在If-Modified-Since请求上调用的方法。该方法的实现非常简单，如下所示：

```
public long displayGenresPageLastModified(
    HttpServletRequest request) {
    return calendar.getLastModified();
}
```

为了保证重新生效对每个请求都得到执行，我们设置了一个must-revalidate头部。在重新生效不可避免之前，我们允许高速缓存60秒，因为我们的业务需求规定这个数据可以有长达1分钟那么久。

```
public ModelAndView displayGenresPage(
    HttpServletRequest request,
    HttpServletResponse response) {
    List genres = this.calendar.getCurrentGenres();
    cacheForSeconds(response, 60, true);
    return new ModelAndView(WELCOME_VIEW_NAME, GENRE_KEY, genres);
}
```

为所有框架控制器所共有的com.interface21.web.servlet.mvc.WebContentGenerator超类中的cacheForSeconds方法如下实现，其中提供了一个简单抽象以处理HTTP 1.0和HTTP 1.1协议的细节：

```
protected final void cacheForSeconds(
    HttpServletResponse response, int seconds, boolean mustRevalidate) {
    String hval = "max-age=" + seconds;
    if (mustRevalidate)
        hval += ", must-revalidate";
    response.setHeader("Cache-Control", hval);
    response.setDateHeader("Expires",
        System.currentTimeMillis() + seconds * 1000L);
}
```

J2EE Web开发人员和MVC框架经常忽略支持由HTTP头部所控制的Web高速缓存，特别是忽略支持重新生效。这是十分不幸的，因为没能发挥这个标准能力不必要地增加了应用服务器的负荷，迫使用户不必要地等待，并浪费了Internet带宽。在考虑在应用服务器中使用像高速缓存筛选器之类的粗糙高速缓存解决方案之前，应该考虑选用HTTP头部，这个选择可能会提供一个更简单，甚至更快速的替代方案。

先进高速缓存与ESI

在J2EE应用服务器前面进行高速缓存的另一种选择是边缘式（Edge-side）高速缓存。边缘高速缓存器不同于共享式代理高速缓存器，因为它们一般在应用提供者或应用提供者所选择的主机控制之下，并且明显复杂得多。边缘高速缓存器不是依赖HTTP头部，而是知道一个文档的哪些部分已经发生了变化，因而最大限度地避免了生成动态内容的需要。

边缘高速缓存器产品包括出自Persistence Software的Akamai和Dynamai。这样的解决方案能够产生巨大的性能好处，但也超出了本J2EE图书的讨论范围。

随着Edge-Side Includes (ESI) 的出现，这样的高速缓存正被逐渐标准化。ESI是“一种用来为应用在Internet边缘的动态装配和交付定义Web页面构件的简单置标语言”。嵌入在生成文档内容中的ESI置标语言语句控制着应用服务器与客户之间的“边缘服务器”上的高速缓存。ESI开放标准规范正由Akamai、BEA Systems、IBM、Oracle以及其他公司合作编纂。实际上，ESI相当于一个高速缓存标志标准。有关这方面的进一步信息，请参见<http://www.esi.org>站点。

JSR 128 (<http://www.jcp.org/jsr/detail/128.jsp>) 在Java Community Process领导下正由同一个机构集团开发，并且涉及到通过JSP自定义标志提供对ESI的Java支持。这些JESI标志将生成ESI页面，这意味着J2EE开发人员可以使用熟悉的JSP标志，而不是学习新的ESI语法。

许多J2EE服务器自身都实现了Web层高速缓存（这也是Microsoft .NET体系结构的一个关键服务）。像iPlanet/SunONE和WebSphere这样的产品提供了服务器小程序和其他Web构件的输出结果的可配置高速缓存，因而消除了修改应用代码的需要。

J2EE应用中性能与可缩放性问题的主要原因

我们已经在前面讨论过所有这些问题。下面不分先后顺序简要总结笔者所见过的、在J2EE应用中引起性能和可缩放性问题的主要原因。

- 错误地配置了JVM、应用服务器或数据库。
- 在不恰当的时候使用了分布式体系结构。远程调用是应用可执行的代价最高的操作之一。
- 在分布式应用中不必要的进行EJB的“罗嗦”调用。
- 在集中式应用中没有充分理由就使用EJB。除非EJB的使用增加实际价值，否则它的使用只增加复杂性和显著的运行时开销。
- 在无状态会话EJB可以实现相同任务时，使用有状态会话EJB。
- 由于没有考虑状态管理的影响而导致聚类中的性能不良和的可缩放性下降。例如，不必要的为每个用户保存大量会话状态，未删除不活动的会话。
- 在必须从较精细的方法中返回大型结果集的地方，使用了含有BMP的实体EJB。
- 在O/R建模不适用而JDBC将证明更有效的场合使用实体EJB。正如笔者在第1章中所说的，J2EE开发人员往往对数据访问有点教条；效率似乎比“纯”J2EE体系结构的理论空想更重要。
- 使用含有远程接口的实体EJB。随着EJB 2.0中对本地接口的引进，这可以被看做只是一个设计失误。
- 没有挖掘目标数据库的各种能力。在这种常见的情况下，开发人员把主要精力集中在代码可移植性上，此时最好是把精力集中在设计可移植性上，把方言的有效使用隐藏在一个可移植接口的后面。例如，一概拒绝使用存储过程可能会不必要导致很慢的RDBMS访问。
- 盲目信任一些导致极差性能的“J2EE设计模式”。在第7章中，我们通过Sun公司的Java Pet Store示例应用讨论了性能争议，该示例应用就演示了这种危险。

- 没有高速缓存应该已得到缓存的资源和数据：例如，执行多余的JNDI查询或重复的数据库访问来检索引用数据。

小结

性能和可缩放性是至关重要的业务需求，可以决定一个项目的成败。因此，在项目一开始，就必须明确定义性能期望值，而且我们应该考虑设计决策在整个项目生存周期内的各种性能含义。由于这是一个如此重要的考虑因素，所以笔者讨论了整本书中所涉及的每个主要设计选择的性能与可缩放性含义。通常，通过优化来改进一个应用的性能存在一定的余地，即便是当该应用在功能上已经完成的时候，但由于高级设计决策对J2EE应用的性能有这样的影响，所以在这么迟的阶段实现真正明显的改进常常是不可能的。

我们已经知道基于确凿证据，而不是基于直觉进行性能驱动的设计选择的重要性。还讨论了使用载荷测试工具和仿形测试工具为我们收集关于性能和吞吐量的证据，并见过一些实际例子。我们已经知道这样的测试不应该留到项目生存周期结束之后再进行，以及应该及早实现应用功能度的“纵向程序片”，以检验性能目标可以得到满足。

学习了解决性能问题的一些技巧，其中包括：

- 保证应用服务器配置对应用的各种需求是最佳的。
- 省去增加开销而又不是支持功能度所必需的多余容器服务。
- 实现高速缓存，或使用一个第三方高速缓存解决方案。高速缓存在Web应用中是特别重要的。一定程度的高速缓存在分布式应用中通常是不可或缺的。

本章还讨论了代码级优化，尽管这种优化在J2EE应用中通常产生的收获有限，除非应用代码开始时是效率特别低的。

本章还讨论了分布式应用的各种特有性能问题。远程方法调用比本地方法调用慢许多倍，因此最大限度地减少分布式应用中的远程调用次数，以及最大限度地降低整个网络范围交换数据的总数量是至关重要的。

本章还讨论了一些Web层性能问题。我们通过比较第13章中所讨论的部分Web视图技术的性能，介绍了一些基准测试。另外，还分析了代理与浏览器高速缓存如何降低Web容器上的负荷和改进Web容器的性能，以及我们怎么才能使用HTTP头部来使这些变得更有可能。

最后，本章介绍了J2EE应用中引起性能问题的一些常见问题，以及我们应该从中汲取的教训。

如果读者手中正有几个J2EE项目，你将需要发展出一种相当可靠的直觉，以觉察出潜在的性能问题。但是，不管你的经验是何种层次，基于实验制定决策都是很重要的。

第16章 结论：让J2EE为我所用

在本书中，我们已经谈论了许多题目。虽然我们还没有尝试过从总体上探讨J2EE，但已经详细地讨论了对构造典型的应用最有用的那些特性和构件。虽然本书将讨论的重点放在了Web应用上，但书中所讨论的大多数问题与所有客户类型都有关系。

笔者发觉，太多的J2EE文献采用了一种与现实的企业级开发不相符的、高度理论化的方法。在本书中，笔者力图采取一种适合J2EE设计与开发的、实际而又实用的方法。特别是，笔者把重点放在了以下几个方面：

· 简单性

在很大程度上，采用一个像J2EE应用服务器这样的框架是为了简化应用代码。如果使用正确，J2EE能够帮助开发人员部分地免去编程任务的最大复杂性。

· 可维护性

软件成本的80%以上被花费在维护上，而不是花费在开发上，因此开发易于维护的应用是至关紧要的。

· 性能

效率在大多数实际的应用中是一个重要考虑因素，尤其是在Web应用中。J2EE应用服务器不能免去我们设计保证满意性能的应用的责任，而且我们需要有必要时进行设计权衡的思想准备。

· 生产效率

应用必须在合理的时间内被开发出来。生产效率是一个至关重要的考虑因素，我们在整个项目生存周期内都必须重视这个问题。

我们不仅分析了各种J2EE规范，而且还讨论了一些重要特性在领先服务器中是如何实现的。J2EE现在落后了数年时间，虽然许多特性已经证明了它们的价值，但其他特性则没有取得如此大的成功。我们必须把这个因素考虑进来。

本章将介绍我们已经汲取的部分教训。

一般原则

下面简要总结一下我们已经讨论过的部分最重要的一般原则：

· 使用J2EE：不要让它使用你

成功的软件项目把技术作为工具来对待。应该把J2EE用做一种最后的手段。J2EE范围内的诸多技术每种都只应该用在它提供了明显好处的地方。不要仅仅因为J2EE的特性存在（可能永远存在），就禁不住诱惑去使用它们；这可能会增加不必要的复杂性和后续成本。

· 不要使用EJB，除非它提供实际价值

这或许是上述原则的特殊事例。笔者发觉，有太多的J2EE开发人员认为EJB是J2EE的核心，而且所有企业级应用都应该使用EJB。这是错误的观点。EJB是一种复杂技术，

它能很好地解决某些问题，但使用不当时会增加多余复杂性。当EJB得到正确使用时，它们应该通过隐藏并发问题、事务管理、安全管理和远程调用的处理来简化应用代码。当EJB不能提供一种简单性好处时，决定使用它们可能是错误的。各种好处应是显而易见的，而不只是一个可缩放性更大的模糊承诺。

- **不要使用分布式体系结构，除非它是满足业务需求必不可少的**

分布式应用（其应用构件被分布在多个服务器当中）比集中式应用（其所有构件都运行在同一个JVM中）复杂得多。因此，开发和维护分布式应用的难度和成本，比开发和维护集中式应用的难度和成本大得多。除非万不得已，否则应该避免招致这种成本。第1章和第6章中讨论了如何决定一个分布式体系结构何时适合。需要注意的是，如果必要，集中式应用仍可以被聚类，其中该聚类中的每个服务器都运行所有应用构件。当在集中式应用中使用EJB时，应该使用本地接口，而不是远程接口。

- **不要对企业级软件开发抱有过分的J2EE中心论观点**

Java是一种适用于企业级开发的杰出语言，而J2EE是一个适用于企业级开发的杰出平台。但是，企业级应用需要不止一种语言和围绕这种语言的各种技术。优秀的J2EE开发人员具有认识J2EE应用与之交互的各种企业级技术的能力，并有必要时寻求专家帮助的思想准备。专业开发人员还应该知道和接受来自替代平台（如.NET）的概念。

- **J2EE不只是规范**

虽然公用规范描述J2EE应用服务器的功能度是一大好处，但从这些规范在领先应用服务器内的各种实现中汲取教训也是十分重要的。例如，开发人员必须知道领先实现现在有状态会话组件、实体组件等方面的各种局限性。开发人员必须把他们的决策建立在证据之上，而证据只能从实际的实现中，而不是从规范中提取。

- **仔细考虑数据库可移植性**

如果数据库可移植性增加复杂性或降低性能，就不要试图实现这种可移植性，除非将来确实有可能需要这种可移植性。通常，实现不同持久性存储器类型之间的抽象，比如RDBMS与ODBMS之间的抽象有一定的代价。不应该轻易招致这种代价，因为数据往往放在它所到达的地方。RDBMS模式很有可能比一个J2EE应用有更长的寿命。

- **JDBC（和SQL）是强有力的工具**

不要害怕使用它们，而要学会熟练地使用它们。对J2EE能够使开发人员不必关心关系数据库或其他企业级资源的处理细节的各种声称要持怀疑态度，比如EJB规范的那些声称。通常，这既不现实，又不值得想望。不要指望通过简单地忽略RDBMS如何工作来解决所谓的“对象-关系阻抗不匹配”；要设法构造能够有效使用RDBMS的OO应用。如果使用正确，JDBC会有很高的效率，并易于使用。我们在第9章中已经见过一个简单的框架，它能够用来简化JDBC的处理和帮助防止常见错误。

- **不要忽视目标数据库的各种能力**

不仅你已经为这些能力付出了代价，而且它们也是数十年研究与开发的成果，应该工作得非常好。例如，有些任务用存储过程来完成会比用Java代码有效得多，而且不必放弃好的设计原则即可获得性能好处常常是可能的。不要把RDBMS当做傻呼呼的

存储装置，也不要忽视DBA输入对“纯”J2EE解决方案的强烈请求。

- **如果可能，要设法实现应用服务器自己的可移植性**

要定期地使用Sun公司的J2EE Reference Implementation来检查你的应用对各种J2EE规范的符合性。有时，使用应用服务器的专有特性是必不可少的：例如，为了最大限度地提高某个特定的性能敏感操作的性能。但是，由于疏忽或懒惰而违反各种规范不会有好结果，并且在需要把应用移植到另一个应用服务器上时将付出极高代价。

- **一个好的应用也可能是最简单的应用**

做“可能管用的最简单事情”的XP忠告对J2EE特别中肯，因为J2EE提供了大量复杂的选择，其中的许多选择常常是必需的。对似乎会导致过度复杂性的设计决策要特别谨慎。例如，请考虑一个利用JMS来实现异步调用的决策。虽然JMS和消息驱动式组件在某些情况中十分有用，但使用它们通常将增加复杂度，并且与使用普通同步调用相比，将使一个应用变得更难调试和维护，除非这个增加的复杂性提供真正的好处，否则最好是避开它。

- **EJB的大多数好处可以通过无状态会话组件来实现**

使用有状态会话组件和实体组件所牵涉到的权衡取舍是比较复杂的，而且它们的好处不十分明确。无状态构件模型也已在许多中间件系统中通过了实践的检验，而不是只有EJB才通过了实践的检验。相反，有状态会话组件和实体组件在实践中的检验结果不是很好。有状态会话组件会限制可缩放性，如果可缩放性是一个关键需要，就不要轻易选用有状态会话组件。在使用关系数据库时，使用实体组件会使许多操作的有效执行变得困难。请记住，O/R映射正如实体组件所规定的那样未必总是适用的。另外需要记住，实体组件仅仅是EJB规范中碰巧定义的一种O/R映射方法而已；JDO和专有O/R映射产品（TopLink）常常更出众。

- **从项目一开始就考虑性能**

业务需求应该含有针对性能和并发性的明确而又可测的目标。明智的做法是在项目生存周期的早期开发一个试验模型（常常叫做峰值解决方案（Spike solution）或纵向程序片（Vertical slide）），以保证性能需求能够得到满足。这使得我们能够在必要时修改应用体系结构，从而避免浪费大量的精力。在项目生存早期的后期，想不做高代价的修改就能改进性能可能已经太迟；在大多数J2EE应用中，我们不能依赖于代码优化来实现显著的性能改进。在J2EE中，性能是一个需要重点关注的方面，因为它常常是一个关键的业务需求（例如在Web应用中），而且有些流行的J2EE“模式”性能特别差。在第15章中，我们已经见过了基于确凿证据做性能相关的决策的重要性，这些证据可以从载荷测试和仿形测试运行中获得。

- **考虑应用在一个服务器聚类中运行的可能性与效果，并将该考虑纳入设计中**

设计这样一个应用是完全有可能的：它在单个服务器上执行良好，但如果必须在聚类中运行，它将证明是效率极差的——例如，如果它保存了大量会话状态，而会话状态又将需要在被聚类的服务器之间进行复制。一定要警惕这样的失误，假如确实存在该应用将来需要被放大成一个聚类部署的可能性，即使它最初被部署在单个服务器上。

- **如果可缩放性是一个需要绝对优先考虑的因素，实际上是在高容量的Web应用中，要设法使用无状态构件来设计**

任何应用服务器和技术（J2EE或其他任何技术）都无法阻止服务器端状态的维护对可缩放性造成的损害。请考虑可缩放性的这样一个例子：使用了物理位置分得很散的服务器。这将证明服务器端状态的复制将有一个严重的问题。如果服务器端状态是必需的，好的设计应该最大限度地减少需要保存的状态量。

- **好的J2EE设计依赖于好的OO设计**

处理J2EE API与构件的细节很容易使我们无法专注于好的OO习惯；这是一个代价很高的失误。J2EE项目的规模和复杂性使得遵守好的OO习惯成为了重中之重，比如不是编程到具体类，而是编程到接口，以及在类里面实现一致的抽象级别。我们还见过使用Java组件如何帮助我们保证把配置保存在Java代码外部，以及基于接口的设计如何帮助我们保证应用服务器之间的可移植性。

- **要使用助手类来抽象低级API（如JDBC、JNDI和JMS）的使用，以使简化应用代码和实现一致性抽象级别**

应用代码只应该专注于问题区域与业务逻辑的实现，这一点是至关重要的，而且只能通过保证低级任务为通用基础结构代码所隐藏来实现，否则直接处理J2EE API会十分罗嗦。这并不意味着我们需要牺牲效率；这样的一个抽象层应该能使应用代码控制底层API的使用。在第9章和第11章中，我们已经见过了从应用代码中隐藏J2EE的部分复杂性的通用包。我们还讨论了这些包的使用如何显著地简化示例应用的代码。

- **要在Web应用中使用MVC结构模式**

在关于J2EE Web应用的高层设计忠告中，这可能是惟一最重要的忠告。MVC结构模式保证表示与控制流程和业务逻辑是分离的。经验已经证明这是所有真正Web应用的最基本目标。要使用MVC模式的一个通用实现，比如Struts框架或本书第12章中所讨论的那个框架。还要记住，Web层应该尽可能地细薄，而且只负责把用户动作转换成业务操作和显示结果。业务逻辑不应该与Web层具体相关。

项目

下面简要总结一下适用于J2EE项目的一些重要指导原则：

- **要基于目前的工作，而不要基于未来某个卓越功能度的承诺来制定决策**

把项目基于新服务器和新规范版本的某个不确定目标是导致失败的一个根源。J2EE规范形成过程的性质表明，规定的特性在现实世界中未必能用（规范形成过程虽然体现了业界的意愿，但导致规范的发布先于它们的产品级实现）。这也是J2EE开发涉及多个规范的另一个方面。

- **不要给无价值的示例加上太多的意思**

大多数J2EE示例应用（比如Sun公司的Java Pet Store）远远没有达到产品级，并且完全忽略了许多常见问题。因此，不要假定它们所使用的所有方法与你的现实问题都相关。

- **全面测试你的应用**

要把你的应用设计成可轻松测试的，并在多个级别上测试它。要把单元测试、功能测试和性能测试加入到开发过程的所有阶段中。在编写应用代码之前，先考虑编写测试，并使测试保持最新。

- **全面用文字说明你的应用，但保持说明文档与应用代码相关**

始终提供全面的Javadoc注释。必要时，使用图形模型，并设计文档。但是要记住，罗嗦而又形式主义的文档很可能会遭到开发人员的忽视，并且在整个软件生存周期内很难保持相关。

- **采用一种及早消除危险的方法**

J2EE应用有许多活动部件，而且必须与分散的资源打交道：许多资源都可能会出错。成功的项目应该能尽可能早地消除危险。

- **在项目的初期选择应用服务器，并用它积累经验**

了解它的能力，以便使开发、测试和发布周期变得顺畅而有效。虽然J2EE应用在应用服务器之间通常是可移植的，但每个应用服务器的使用有非常大的差别，并且都具有各自的优缺点。推迟应用服务器的选择无疑会招致无数的麻烦，给项目所涉及的工作增添难度。在选择应用服务器时，要把重点放在贵机构的需求上，而不是放在销售资料上。

- **在项目生存周期的初期自动化生成和测试过程**

虽然IDE提供非常有价值的功能度，但它们的功能度不是可脚本化的。使用IDE来提高工作效率，但是要保证有一个已就位的标准生成脚本。Ant目前是生成Java应用的事实标准，并且能够用来自动化许多其他基本任务，比如单元测试、更新资料和部署。

- **从项目一开始就聘请J2EE设计方面的一个核心专家组**

J2EE是复杂的，而且未必像它所说的那样工作。企图省钱而使用一个无经验团队最终会花费更大的代价。一个核心专家组能够在项目进行过程中指导不太有经验的开发人员，进而逐渐建立一支有较高专业技能的后备队。

- **编写尽可能少的应用代码**

尤其是，要通过了解和利用应用服务器的各种能力并使用现有框架来避免重复劳动。

- **采用一致的设计与编程标准，以保证好的习惯**

同一个项目上的所有开发人员都要了解和遵守公共标准。这有助于保证好的习惯，提高可维护性，以及帮助最大限度地减少重复劳动。

最重要的是，采用一种灵活而又实用的J2EE设计方法是至关重要的。最终，应用将由它们满足业务需求的程度以及开发与维护它们的性价比来衡量。

如果顺利的话，在读完本书之后，读者对制定设计决策及进行实现选择将会更自信。祝您好运！

附录A 实现视图技术

本附录的意图是作为配合第12章和第13章的参考。本附录讨论第12章中所介绍的MVC Web应用框架如何支持某一种视图技术，并且不需要修改控制器或模型代码。

本附录首先考虑这个框架如何分离控制器与视图，然后看一看这个框架怎样支持第13章中所讨论的那些视图技术。对于每种视图技术，我们将讨论：

- 使用这种技术的Web应用所需要的安装与配置；
- 我们的MVC框架如何支持这种技术，以及集成这种技术所需要的基本Java代码，无论我们使用了何种框架；
- 如何在使用了这个框架的应用中定义这种类型的视图。

最后，我们将看一看如何在这个框架内实现自定义的新视图实现，以支持其他视图技术或提供应用特有的高级行为。

虽然本附录主要与第12章中所介绍的MVC Web应用框架有关，但关于安装和配置视图技术的信息适用于所有Web应用。演示如何与每种视图技术集成的代码示例也可以用做读者自己的实现代码的基础。

本附录含有一些很长的代码清单。但是，篇幅有限意味着这些清单不是完整的。笔者已尽力给出实现每项任务所需要的核心代码（不过，要想了解这里所讨论的各种方法的完整实现，请参考示例应用及其基础结构包中的代码）。

使用视图接口来分离控制器与视图技术

首先来回顾一下第12章中所描述的MVC Web应用框架分离控制器与视图的方法。

这个框架使用命名视图策略（Named view strategy）来提供一个重定向层，以便一个控制器无需知道视图实现的任何情况就能按名选择一个视图。这意味着所有视图都必须实现一个公用Java接口。大多数视图技术把（X）HTML格式化之类的表示信息保存在JSP页之类的模板中，而不是保存在Java类中。但是，由于模板的细节在诸如JSP与XSLT之类的视图技术之间是不同的，所以这种分离取决于一个使用适当技术和模板来管理内容再现的Java视图接口的不同实现。

笔者使用术语“视图接口（View interface）”来称呼在控制器与视图之间提供这种分离的Java对象，使用术语“视图（View）”来称呼保存实际页面布局的资源，比如JSP页。在某些情况下，一个单独的Java类可以担当这两种角色。

视图接口的实现通常是多线程的，而且代表该应用的所有用户进行工作。一般说来，视图的一个实例包装一个特定的模板，进而实现一个单独的、无状态的、接受模型数据并建立一个动态页面的方法。在正常情况下，只有像JSP页这样的模板才需要作为应用的一部分来提供，因而必需的视图实现是由框架所提供的一个标准类。

在第12章中所描述的框架内，一个视图接口必须实现com.interface21.web.servlet.View接口，而该接口基于下面这个方法：

```
void render(Map model,
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException;
```

其中，model参数含有控制器所暴露的若干个模型对象。在某些情况下，它将是空的，如果一个视图不需要任何模型信息。

com.interface21.web.servlet.View接口还允许静态属性（Static attribute）被添加到底层视图技术所暴露的模型上。这使得附加的表示特有数据（比如传给模板页面的变元或默认的页面标题）能够在应用配置中进行添加，不会破坏控制器所返回的模型，因此该模型不应该被设计成适应任一特定的视图技术。

静态属性只有当它们被第一次创建时才能被添加到视图上。一旦一个视图处于使用之中，它的数据（包括它的静态属性）是永久不变的，以保证线程安全。本附录中所讨论的各种视图实现所使用的AbstractView基类允许静态属性按如下所示的CSV格式来定义。这个特性定义了3个静态属性：header、contentWell和footer。

```
welcomeView.attributesCSV=header=/jsp/header.jsp!,\n                           contentWell=/welcome.jsp!,\n                           footer=/jsp/footer.htm!
```

由于控制器所返回的模型数据和静态属性都是表示无关的，所以View接口的实现可以与任何一种视图技术一起工作。

我们将在下文中详细地讨论View接口的实现，但现在来较详细地看一看特定视图实现如何满足我们在第13章中所标识出的基本需求。

· 包装一个特定页面结构

如果一个模板被需要，它可以在该视图对象被配置时被查找和高速缓存。如果视图已经被编译和高速缓存，有些模板技术（比如XSLT）的工作性能会好得多，尽管没有必要高速缓存JSP视图。在某些情况下，比如在使用了XMLC的情况下，视图实现本身将保存页面结构，意味着每个应用屏幕都需要View接口的一个实现。可是，在大多数情况下，视图实现是一个适用于使用了一种给定技术的不同视图的通用框架包装器。例如，同一个应用中的每个Velocity模板可以通过接受模板名作为初始化属性的框架Velocity视图来访问。

· 暴露模型数据给底层视图技术

如果视图是一个JSP页，模型数据将被暴露为请求属性。至于像Velocity之类的模板引擎，模型数据将被放在该模板引擎所特有的一个“上下文”中。在某些情况，比如在XSLT的情况下，另外的预处理可能是必不可少的；例如，为了在模型数据还不是XML格式的地方把Java对象转换成XML节点。

· 构造动态页面

再现的手段差别极大。就同一个Web应用内的每个JSP页来说，javax.servlet.RequestDispatcher接口可以用来向该JSP页进行转发。从该Web容器中获得的一个Request-

Dispatcher，也可以用来向同一个Web应用内的一个输出服务器小程序或一个静态资源进行转发。像Velocity这样的模板引擎提供它们自己的API来翻译一个模板并输出数据。就XSLT来说，TrAX API可以用来执行一个XSL转换，并把结果输出给响应对象。

在我们的应用框架中，View对象通常是应用初始化时所配置的Java组件。这意味着配置可以被保存在Java代码外部。一般说来，每个视图模型只存在一个视图实例（比如一个JSP页）。

视图定义的存储方式取决于所使用的ViewResolver接口的实现。第12章中所讨论的com.interface21.web.servlet.ViewResolver接口定义了下列惟一方法，以返回一个对应于给定名称和Locale（从请求中取出）的视图实例。

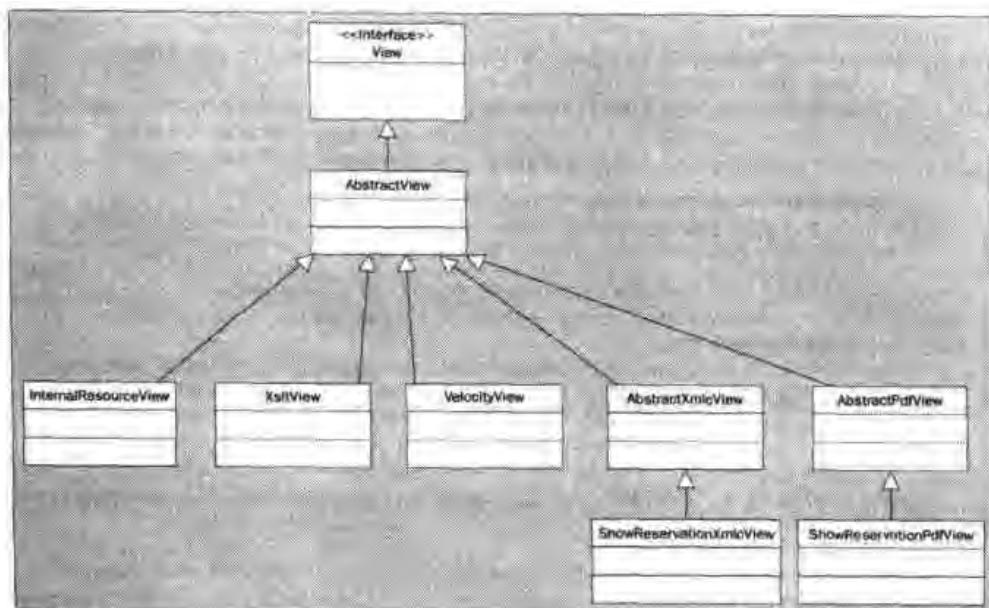
```
View resolveViewname(String viewName, Locale locale)
throws ServletException;
```

提供这个接口的一个自定义、应用特有的实现是很容易的，该实现可以用一种自定义格式保存视图定义，甚至可以完全用Java代码执行视图分解。

但是，我们通常可以依赖示例应用中所使用的默认ViewResolver实现。在这个实现中（我们将把它用于本附录的各个示例），视图定义是/WEB-INF/classes/views.properties文件中的Java组件定义，或者是一个具有相应本地化名称的资源包，比如/WEB-INF/classes/views_fr_ca.properties，这个资源包可以替代部分或全部的视图定义。当下文讨论每种视图技术时，我们将会讨论默认的ViewResolver实现所需要的组件定义。

视图实现

图A.1的UML类图描述了本附录中所讨论的视图实现，以及从AbstractView便利基类中派生出来的所有类。右面底部的两个类（在这种情况下，指从示例应用中派生出来的类），演示了应用特有的类如何扩展了用于XMLC与PDF视图的抽象框架超类。



图A.1

下面的这些框架视图实现是具体的，并且可以在不子类化的情况下使用。

- **com.interface21.web.servlet.view.InternalResourceView**

这个框架视图实现能够在当前Web应用内使用JSP、服务器小程序或其他内容暴露模型数据。

- **com.interface21.web.servlet.view.xslt.XsltView**

这个视图实现能够使用一个XSLT格式表暴露模型数据，在必要时把模型中的Java组件转换成XML形式。

- **com.interface21.web.servlet.view.velocity.VelocityView**

这个视图实现能够使用了一个缓存Velocity模板暴露模型数据。

下面这两个类是用于应用特有视图的抽象超类。涉及到的这两种视图技术要求为每个视图实现一个Java类。

- **com.interface21.web.servlet.view.xmlc.AbstractXmlcView**

这个抽象超类用于使用XMLC技术的视图。运行时要用到一个静态HTML模板，但这个应用特有的Java类保存运行时的页面布局。

- **com.interface21.web.servlet.view.pdf.AbstractPdfView**

这个抽象超类用于使用iText库生成PDF的视图。无任何模板被涉及到，因为所有内容都用Java代码来生成。

View接口从头实现起来很简单，但**com.interface21.web.servlet.view.AbstractView**超类为大多数视图实现提供了一个便利超类：通过使用Template Method设计模式来隐藏静态属性与控制器所提供的模型对象的合并（如果一个静态属性和一个模型属性共享同一个名称，该模型属性将优先）。**AbstractView**的子类需要实现下面保护方法，该方法和**View**接口中的**render()**方法有相同的约定，但它使用的是模型与静态属性的一个合并映像。

```
protected abstract void renderMergedOutputModel(
    Map model,
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException;
```

AbstractView按如下所示在**render()**方法的一个终结实现中调用这个方法。需要注意的是，动态（模型）属性值将替换同名的静态属性。

```
public final void render(Map pModel,
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    // Consolidate static and dynamic model attributes
    Map model = new HashMap(this.staticAttributes);
    model.putAll(pModel);

    renderMergedOutputModel(model, request, response);
}
```

AbstractView类暴露以下组件属性——由这里所讨论的所有视图实现所继承。

表A.1

名称	类型	需要	默认	用途
contentType	String	否	Text/html; charset=ISO- 8859-1	要写到响应对象的内容类型。若需要 了解进一步信息,请参见javax.servlet. ServletResponse的资料
name	String	否	null	这个视图的名称。纯粹用于诊断目的
static AttributesCSV	String	否	null	定义与这个视图相关联的静态属性的 CSV格式串。用在默认的属性视图定 义语法中。参见上面的例子
Static Attribute	java.util. Properties	否	n/a	定义与这个视图相关联的静态属性的 属性值。当在XML组件定义中定义视 图时使用它(当在属性文件中定义视 图时使用CSV格式,因为拥有一个其 自身已在属性语法中的属性值是有问 题的)

JSP

首先来讨论JSP技术——J2EE Web应用中最常用的视图技术。

配置JSTL

由于JSP 1.2是J2EE 1.3规范的一部分,所以能够保证我们的应用服务器肯定带有JSP 1.2。

但是,我们需要包含JSTL库。

示例应用使用JSTL的Apache Jakarta实现,该实现可从<http://jakarta.apache.org/taglibs>站点上获得。下面的JAR文件来自Jakarta JSTL发行包的/lib目录,并且是使用示例应用中所显示的各种标志所必需的。

- jstl.jar
- standard.jar
- sax.jar
- saxpath.jar
- jaxen-full.jar

前两个JAR文件定义JSTL API和标准标志的实现;后3个JAR文件是XML标志的实现所必需的。

这些文件被包含在了配书下载的/lib/runtime/jsp-stl目录中,并被自动复制到由示例应用的Ant生成脚本所构造的Web应用的WEB-INF/lib目录内。

我们不必把那些JSTL TLD包含在使用了它的WAR文件中。我们可以简单地使用JSTL库的API公布URL从java.sun.com站点上导入这些库,就像导入core库的下列示例:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

所有使用了JSP的应用都应该使用JSTL，这会极大地改进JSP创作模型。因此，各种JSTL二进制库都应被包括在所有基于JSP的应用上。

InternalResourceView视图实现

现在来看一看该框架为了用于JSP技术而实现的View接口的实现。这个框架实现的完整清单可以在示例应用所携带的框架代码内的com.interface21.web.servlet.view.InternalResourceView类中找到。这个类之所以被命名为InternalResourceView，而不是被命名为JspView，是因为它除了可以在该Web应用内用于JSP之外，还可以用于静态HTML和服务器小程序。

在使用一个Servlet API RequestDispatcher向当前Web应用内的指定资源进行转发之前，这个视图实现先把模型项复制到请求属性上。其中，这个资源由视图初始化时所提供的一个URL来标识。

通过扩展com.interface21.web.servlet.view.AbstractView超类，InternalResourceView类对只实现renderMergedOutputModel()抽象保护方法和暴露配置新实例所必需的组件属性是必不可少的。

InternalResourceView类的下列部分清单忽略了日志记录方法，以及把调试信息保护在该请求中的有用功能度（如果该ControllerServlet已经设置了一个指明它正处于调试模式中的请求属性）。

包装一个JSP模板的基本步骤非常简单。我们需要一个实例变量来保存url属性，并需要一个属性设置器来允许InternalResourceView组件被配置。

```
public class InternalResourceView extends AbstractView {
    private String url;

    public void setUrl(String url) {
        this.url = url;
    }
}
```

url组件属性必须被设置成当前Web应用内的一个JSP页的路径，比如/jsp/myJsp.jsp。

renderMergedOutputModel()方法的下列实现从Web容器中为已得到高速缓存的URL获得一个RequestDispatcher，并在把模型数据暴露为请求属性之后，把该请求转发给这个RequestDispatcher。

```
public void renderMergedOutputModel(Map model,
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException {

    exposeModelsAsRequestAttributes(model, request);

    try {
        request.getRequestDispatcher(getUrl()).forward(request, response);
    }
    catch (IOException ex) {
        throw new ServletException("Couldn't dispatch to JSP with url '" +
            getUrl() + "' in InternalResourceView with name '" +
            getName() + "'", ex);
    }
}
```

把模式属性设置为请求参数的代码被重新加工成为一个被保护方法，以防有子类希望重用它。

```
protected final void exposeModelsAsRequestAttributes(
    Map model, HttpServletRequest request) {

    if (model != null) {
        Set keys = model.keySet();
        Iterator itr = keys.iterator();
        while (itr.hasNext()) {
            String modelName = (String) itr.next();
            Object val = model.get(modelName);
            request.setAttribute(modelName, val);
        }
    }
}
```

这个标准视图实现可以用来包装同一个Web应用中的任何JSP页、服务器小程序或静态内容。通常，没有必要提供应用特有的子类。

定义在应用中使用的JSP视图

/WEB-INF/classes/views.properties中一个使用了InternalResourceView类的典型视图定义像下面这样：

```
showReservation.class=com.interface21.web.servlet.view.InternalResourceView
showReservation.url=/showReservation.jsp
```

除了从AbstractView中继承来的组件属性之外，InternalResourceView所暴露的惟一组件属性是url。正如上面所显示，这个属性必须被设置成本WAR内的一个资源URL。像下列示例中一样，这个URL可能位于本WAR的/WEB-INF目录之中。

```
showReservation.class=com.interface21.web.servlet.view.InternalResourceView
showReservation.url=/WEB-INF/jspa/protectedJsp.jsp
```

位于特定/WEB-INF目录之中的资源对RequestDispatcher仍是可访问的，但对直接用户请求是不可访问的。

Velocity

本节讨论如何安装和配置Velocity 1.3，以及框架对Velocity模板引擎的内部支持。

安装并配置Velocity

和JSP不同，Velocity不是J2EE的一部分，并且你的应用服务器可能也没有携带它。示例应用的/lib/runtime/velocity目录包含了Velocity 1.3所需要的两个JAR文件：velocity-1.3.jar和velocity-dep-1.3.jar。这两个文件必须被复制到使用了Velocity的Web应用的/WEB-INF/lib目录中。示例应用的Ant生成脚本处理这个复制。

在可以使用各模板之前，必须先配置Velocity。Velocity是高度可配置的，并暴露了许多

配置参数，但是只有我们必须设置的那些配置参数才告诉Velocity在运行时去哪里查找模板。Velocity提供了几种装入模板的策略，比如使用文件系统或类装入器。在示例应用中，笔者选择了类装入器，因为该策略在Web容器之间更有可移植性。

下列关键字取自示例应用的WEB-INF/velocity.properties文件，它们告诉Velocity从类路径中（即WAR文件中的WEB-INF/classes或WEB-INF/lib目录下）装入模板，而且它应该自动地重新装入被更改的模板，以使开发变得更轻松。

```
resource.loader=class
class.resource.loader.class =
org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
class.resource.loader.cache = false
class.resource.loader.modificationCheckInterval = 5
```

和JSP编译一样，在正常情况下，Velocity模板重装入在生产中应该被关闭，因为这种装入将会影响性能。

重要的可选设置包括在Velocity启动时被装入并允许供所有模板使用的Velocity宏库的路径。下列示例指定一个Velocity宏库的位置：

```
velocimacro.library = myMacros.vm
```

可以在一个CSV列表中包含多个库，并且应该像查找普通Velocity模板一样来查找它们。

在试图计算模板之前，需要给Velocity应用这些属性。完成这项任务的方法之一是使用一个具有<load-on-startup>子元素的服务器小程序，以保证它在Web应用接收到请求之前被装入。不过，我们在第11章中所讨论的那个基于组件的框架提供了一种更好的替代方法。由于我们的应用上下文中的一个“单元集”组件被保证得到实例化，即使用户代码不访问它，所以我们可以简单地定义一个在应用初始化期间配置Velocity的组件。由于这个缘故，该框架提供了com.interface21.web.servlet.velocity.VelocityConfigurer组件。下列组件定义取自示例应用的WEB-INF/ticket-servlet.xml文件，并展示了该组件的用法：

```
<bean name="velocityConfig"
      class="com.interface21.web.servlet.view.velocity.VelocityConfigurer">
    <property name="url">/WEB-INF/velocity.properties</property>
</bean>
```

请注意那个惟一的组件属性url，它是标准Velocity属性文件在WAR描述符内的位置。VelocityConfigurer组件装入Velocity属性（在这种情况下是从WAR描述符内的一个属性文件中），并使用它们初始化Velocity单元集，如下所示：

```
Velocity.init(properties);
```

Velocity 1.2引进了对同一个Web应用中多个Velocity独立实例的支持，所以Singleton设计模式不再是惟一选择。

实现用于Velocity的视图接口

现在来看一看本框架对com.interface21.web.servlet.View接口的内部实现，这个实现通过Velocity模板来暴露模型数据。

这个框架实现需要在它的render()方法中执行下列步骤：

- 创建一个新的Velocity上下文对象来建立这个响应。
- 使用我们在InternalResourceView类中把模型对象暴露为请求属性的同一种方式暴露模型对象为Velocity上下文对象。
- 获得相关的Velocity模板。由于每个VelocityView对象都将与一个单独的视图关联，所以该Velocity视图可以在启动时获得并高速缓存必需的Velocity模板，使它能够在框架初始化时核实该模板存在并且没有语法错误。
- 使用Velocity把模板与上下文数据“合并”起来写入HttpServletResponse对象。

把模型数据暴露给一个Velocity模板

要想了解完整实现，请查阅配书下载的/framework目录中的com.interface21.web.servlet.view.velocity.VelocityView类。笔者在下面描述各基本步骤。

和本附录中所讨论的其他视图实现一样，VelocityView类扩展AbstractView。renderMergedOutputModel()方法中的第一步是创建一个Velocity上下文来建立这个响应：

```
Context vContext = new VelocityContext();
```

模型对象在下面这个私用方法中被暴露给这个新建的请求特有Velocity上下文：

```
private void exposeModelsAsContextAttributes(Map model, Context vContext) {
    if (model != null) {
        Set keys = model.keySet();
        Iterator itr = keys.iterator();
        while (itr.hasNext()) {
            String modelname = (String) itr.next();
            Object val = model.get(modelname);
            vContext.put(modelname, val);
        }
    }
}
```

模板可以在该视图被初始化时从Velocity单元集实例中被获得和高速缓存，如下所示：

```
private void loadTemplate() throws ServletException {
    String mesg = "Velocity resource loader is: [" +
        Velocity.getProperty("class.resource.loader.class") + "]; ";
    try {
        this.velocityTemplate =
            RuntimeSingleton.getTemplate(this.templateName);
    }
    catch(ResourceNotFoundException ex) {
        mesg += "Can't load Velocity template '" + this.templateName +
            "': is it on the classpath, under /WEB-INF/classes?";
        logger.logp(Level.SEVERE, getClass().getName(), "getTemplate",
    }
```

```
        mesg, ex);
    throw new ServletException(mesg, ex);
}
catch(ParseErrorException ex) {
    mesg += "Error parsing Velocity template '" + this.templateName + "'";
    logger.logp(Level.SEVERE, getClass().getName(), "getTemplate",
               mesg, ex);
    throw new ServletException(mesg, ex);
}
catch(Exception ex) {
    mesg += "Unexpected error getting Velocity template '" +
            this.templateName + "'";
    logger.logp(Level.SEVERE, getClass().getName(), "getTemplate",
               mesg, ex);
    throw new ServletException(mesg, ex);
}
}
```

在使用模型数据填充了一个Velocity上下文之后，我们就可以把输出结果生成给HttpServletResponse对象，如下所示：

```
this.velocityTemplate.merge(vContext, response.getWriter());
```

VelocityView类的实际实现使用了一种性能更高但稍微复杂一点的方法，并基于和Velocity捆绑在一起的VelocityServlet。详细情况，请参见源代码。

提供日期与货币格式化支持

令人惊讶的是，Velocity 1.3缺少对日期和货币格式化的支持。Velocity Tools项目包括一些支持，但它似乎仍处于开发之中。因此，为了保证框架的Velocity视图不束缚于当前地区，我们需要在这方面帮助Velocity解决难题。

我们可以选择把格式化的日期和货币量暴露为每个模型中的串属性，但这会破坏优良的设计。通常情况下，在视图中处理本地化比在模型中处理好。

如果视图定义指出要包装的模板需要执行日期和货币格式化，VelocityView类就会给VelocityView上下文增加java.text.SimpleDateFormat和java.text.NumberFormat对象。Java对数字和货币格式化的可本地化支持很简单，也很熟悉，在java.text包Javadocs中有文字说明。

必须为每个请求都构造SimpleDateFormat和NumberFormat对象，因为Velocity模板代码可能会更改这些对象的配置——例如通过设置格式化模式。这些助手仅当VelocityView对象上的两个组件属性——exposeDateFormat和exposeCurrencyFormat被设置成true以指出一个特定View实例被关联到一个需要格式化日期和货币量的模板时才被暴露（把这些助手对象暴露给不需要它们的大多数模板将是不经济的）。这些助手通过使用HttpServletRequest的locale属性来初始化，如下所示：

```
public static final String DATE_FORMAT_KEY = "simpleDateFormat";
public static final String CURRENCY_FORMAT_KEY = "currencyFormat";
```

```

private void exposeHelpers(Context vContext, HttpServletRequest request) {
    if (this.exposeDateFormatter) {
        // Javadocs indicate that this cast will work in most locales
        SimpleDateFormat df = (SimpleDateFormat)
            DateFormat.getDateInstance(DateFormat.LONG,
                DateFormat.LONG, request.getLocale());
        vContext.put(DATE_FORMAT_KEY, df);
    }

    if (this.exposeCurrencyFormatter) {
        NumberFormat nf =
            NumberFormat.getCurrencyInstance(request.getLocale());
        vContext.put(CURRENCY_FORMAT_KEY, nf);
    }
}

```

如果必要，我们现在可以处理Velocity模板中的模式，并使用那些对象格式化模型数据，如下所示：

```

The date of the performance is
$simpleDateFormat.format($performance.when).

The total cost of these tickets will be
$currencyFormat.format($reservation.totalPrice).

```

定义在应用中使用的Velocity视图

取自WEB-INF/classes/views.properties文件的下面这个示例，为第13章中所讨论的示例页面定义一个Velocity视图组件。需要注意的是，它把那两个格式化助手属性设置成了true，因为这个页面需要显示演出日期和入场券价格计算。

```

showReservation.class=
    com.interface21.web.servlet.view.velocity.VelocityView
showReservation.templateName=showReservation.vm
showReservation.exposeDateFormat=true
showReservation.exposeCurrencyFormat=true

```

除了从AbstractView超类中继承来的组件属性之外，VelocityView类还暴露表A.2所示的组件属性。

表A.2

名称	类型	需要	默认值	用途
templateName	String	是	n/a	要使用的Velocity模板的名称。这个属性的含义将取决于把Velocity配置成怎样装入模板。模板通常位于WAR描述符内
exposeDateFormat	boolean	否	False	我们应该给Velocity上下文添加一个针对请求地区被初始化的SimpleDateFormat对象吗

(续表)

名称	类型	需要	默认值	用途
exposeCurrencyFormatter	boolean	否	False	我们应该给Velocity上下文添加一个针对请求地区被初始化的NumberFormat对象吗
poolSize	int	否	40	要用来处理这个页面的Velocity书写器数量。关于这种优化的进一步信息，请参见Velocity技术资料

XSLT

本节讨论MVC Web应用框架怎样支持XSLT视图。

安装Domify

由于JAXP是J2EE 1.3的一部分，所以能够保证那些支持库已处于适当的位置。但是，这个框架的内部XSLT支持依靠开放源Domify库（参见第6章和第13章）把Java组件暴露为XML节点。示例应用下载的/lib/runtime/common包中含有domify.jar——必需的小JAR文件。这个JAR文件必须被复制到使用XML变换的WAR描述符的/WEB-INF/lib目录中。同样，示例应用的Ant生成脚本处理这个复制。

实现用于XSLT的视图接口

和JSP和Velocity视图的情况一样，每个XSLT格式表都需要一个视图实例。但是，这个实现可以由一个框架来提供；没有必要让应用代码直接使用XSLT。

和我们已经见过的那些视图实现一样，com.interface21.web.servlet.view.xslt.XsltView标准实现扩展com.interface21.web.servlet.view.AbstractView便利超类。在使用JAXP标准API进行XSLT转换之前，它使用Domify把Java组件转换成XML形式（如果必要）。

基础XML语法分析器和XSLT转换引擎在服务器之间可能是不一样的，因为JAXP把使用了它的代码与实际的实现隔离开。对于JBoss/Jetty，要使用的产品分别是Apache Xerces和Apache Xalan——最常见的两个选择。

执行XSLT转换

XsltView类暴露一个组件属性，该属性能将格式表位置设置成在WAR内的一个URL。

```
public void setStylesheet(String url) {
    this.url = url;
}
```

在初始化时，XsltView类的每个实例都将获得javax.xml.transform.TransformerFactory接口的一个新实例。

```
this.transformerFactory = TransformerFactory.newInstance();
```

它可以使用该对象创建一个javax.xml.transform.Templates对象，这个Templates对象是格式表的一个已编译的、线程安全的表示；该格式表能够便宜地返回供该格式表使用的javax.xml.transform.Transformer对象。

```
private void cacheTemplates() {  
  
    if (url != null && !"".equals(url)) {  
        Source s = getStylesheetSource(url);  
        try {  
            this.templates = transformerFactory.newTemplates(s);  
        }  
        catch (TransformerConfigurationException ex) {  
            throw new ApplicationContextException(  
                "Can't load stylesheet at '" + url + "' in XsltView with name '" +  
                getName() + "'", ex);  
        }  
    }  
}
```

如果传递一个URL，XsltView类的getStylesheetSource()方法就返回一个javax.xml.transform.Source对象。默认的实现使用ServletContext.getRealPath()方法来查找该URL在WAR描述符内的文件系统位置。这在笔者所试验过的所有容器内都工作得很好，但不保证是可移植的（例如，请设想一个把WAR内容抽取到一个数据库而不是一个文件系统的容器），所以这个方法必须由一个自定义子类来替代，才能用在不允许这种文件访问的容器中。

```
protected Source getStylesheetSource(String url)  
throws ApplicationContextException {  
  
    String realpath = getWebApplicationContext().getServletContext().  
        getRealPath(url);  
    if (realpath == null)  
        throw new ApplicationContextException(  
            "Can't resolve real path for XSLT stylesheet at '" + url +  
            "'; probably results from container restriction: " +  
            "override XsltView.getStylesheetSource() to use an " +  
            "alternative approach to getRealPath()");  
  
    Source s = new StreamSource(new File(realpath));  
    return s;  
}
```

警告：ServletContext.getRealPath()方法可能是不可移植的。不过，它在实践中是如此有用，以至于许多Web应用框架和供Web应用中使用的包使用了它。

实际的实现在写结果到响应对象的doTransform()方法中完成。请注意，我们用已得到高速缓存的Templates对象创建一个javax.xml.transform.Transformer对象来执行转换。我们无法高速缓存一个Transformer对象，而能高速缓存一个Templates对象，因为Transformer对象不是线程安全的。因此，使用已得到高速缓存的Templates对象是一种基本的性能优化；我们不希望必须为每个转换都从头创建一个新的Transformer对象，因为这需要对XSLT格式表做语法分析，并分析该XSLT格式表的结构。

```

protected void doTransform(HttpServletRequest response, Node dom)
    throws IOException, ServletException {

    try {
        Transformer trans = this.templates.newTransformer() :
            trans.setOutputProperty(OutputKeys.INDENT, "yes");
        trans.setOutputProperty("(http://xml.apache.org/xslt)indent-amount",
            "2");
        trans.transform(new DOMSource(dom), new StreamResult(new
            BufferedOutputStream(response.getOutputStream())));
    }
    catch (TransformerConfigurationException ex) {
        throw new ServletException(
            "Couldn't create XSLT transformer for stylesheet '" + url +
            "' in XSLT view with name='" + getName() + "'", ex);
    }
    catch (TransformerException ex) {
        throw new ServletException(
            "Couldn't perform transform with stylesheet '" + url +
            "' in XSLT view with name='" + getName() + "'", ex);
    }
}
}

```

虽然笔者在这里设置了两个Xalan特有的属性，但这不会妨碍代码使用另一个XSLT引擎；它们将被简单地忽略掉。

`renderMergedOutputModel()`方法的下列实现将把我们的模型映像的内容暴露为一个单独的XML文档，进而在数据还不是XML格式时飞行地“变换”Java组件对象：

```

protected void renderMergedOutputModel(Map model,
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    if (model == null)
        throw new ServletException("Cannot do XSLT transform on null model");

    Node dom = null;
    String docRoot = null;
    Object singleModel = null;
}

```

如果该模型映像中只含有一个模型对象，则必须检查它是否已含有XML数据。如果它含有，就简单地把XML数据暴露给XML视图。

```

if (model.size() == 1) {
    docRoot = (String) model.keySet().iterator().next();
    singleModel = model.get(docRoot);
}

if (singleModel != null && (singleModel instanceof Node)) {
    dom = (Node) singleModel;
} else {
}

```

如果该模型映像中含有多个模型对象，或者只含有一个模型对象，但它不是XML格式，那么我们将需要飞行地“变换”整个模型映像。我们使用`docRoot`组件属性值为模型提供一个名称，如下所示：

```

try {
    addRequestInfoToModel(model, request);
    dom = this.domAdapter.adapt(model,
        (docRoot == null) ? this.root : docRoot);
}
catch (RuntimeException rex) {
    throw new ServletException("Error domifying model in XSLT view " +
        "with name=' " + getName() + "' ", rex);
}
}

```

既然dom对象中已经有了XML数据，现在就可以调用XML转换了，如下所示：

```

doTransform(response, dom);
}

```

这维护与JAXP API的交互。Java应用代码不必执行任何XSLT处理或XML处理，除非它与应用的业务需求相关。

日期和货币格式化支持

XsltView框架类还解决另外一个常见问题。XSLT没有提供对数据格式化的支持，所以同Velocity的情形一样，我们需要在这方面帮助XSLT解决难题（XSLT 2.0可能会增加一种日期格式化能力，并把它建模在Java的java.text.SimpleDateFormat上）。我们还应该能够给XSLT格式表提供关于用户地区的信息，以便该格式表能够以正确的格式初始化日期格式化。虽然用户的地区是可以从HttpServletRequest中获得的，但它不是我们的数据模型的一部分（它也不应该是）。

我们先给模型增加附加的信息，以后再变换这个信息来暴露它，如下所示：

```

model.put(REQUEST_INFO_KEY, new RequestInfo(request));
}

```

com.interface21.web.servlet.view.RequestInfo类（也可以用于其他视图技术）暴露下列这些属性，其中每个属性返回一个标准的两字母代码，比如GB或en：

```

public String getCountry();
public String getLanguage();
}

```

这些代码可以驱动XSLT中的查找，也可以用做给XSLT提供日期格式化功能度的Java XSLT扩展函数（Extension function）的变元。XSLT提供了一种简单的Java语言联编（Language binding），以便XSLT格式表能够调用使用了反射的Java方法。虽然Java扩展函数会被滥用（它们提供了通往XSLT滥用的一个小后门：例如，请设想一下执行JDBC操作的扩展函数可能对可维护性做些什么），但它们有时候是非常有用的，而且易于使用。目前的使用（为了在XSLT视图格式化表现很弱的方面增强这种格式化的各种能力）就是一个使用得当的上佳例子。

请注意，笔者从Taylor Cowan在Java World内所发表的一篇关于XSLT扩展函数的优秀文章中借用了这种方法背后所隐含的那些概念：请参见<http://www.javaworld.com/javaworld/jw-12-2001/jw-1221-xslt-p2.html>。

但是，使用扩展函数降低了格式表的可移植性，所以应该尽量减少使用扩展函数。

`com.interface21.web.servlet.view.xslt.FormatHelper`类暴露了两个静态方法，使我们能格式化日期和货币量，如下所示：

```
public static Node dateTimeElement(long date, String language,
String country);

public static String currency(double amount, String language,
String country);
```

`dateTimeElement()`方法返回一个使用`SimpleDateFormat`对象为指定地区所建的XML节点，而不是返回文字内容（在扩展函数中，我们可以使用JAXP API来创建节点）。这个新节点不被包括在XML输出文档内，但看起来会像下面这样：

```
<formatted-date>
<month>July</month>
<day-of-week>Tuesday</day-of-week>
<year>2002</year>
<day-of-month>23</day-of-month>
<hours>11</hours>
<minutes>53</minutes>
<am-pm>AM</am-pm>
</formatted-date>
```

关于`dateTimeElement()`方法的实现，请参见`FormatHelper`类的源代码。如果我们为`<formatted-date>`元素创建了一条格式表规则，现在就可以像我们希望的那样格式化信息了，如下所示：

```
<xsl:apply-templates select="format.dateTimeElement(when/time, 'en', 'GB') " />
```

`FormatHelper`类的`currency()`方法简单地返回一个串，该串中含有一个已格式化的货币量，比如 £ 3.50 或 \$3.50。我们可以像下面这样使用这个串：

```
<xsl:value-of select="format:currency(totalPrice, 'en', 'GB') " />.
```

为了在格式表中使用这些扩展方法，我们所需要做的只是在根元素中为它们声明一个名称空间，如下所示：

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:format="com.interface21.web.servlet.view.xslt.FormatHelper"
>
```

最后，我们需要能够自定义变换。Domify无法处理循环引用，比如我们的对象模型中的`Performance`与`Show`对象之间的引用：每个`Performance`对象都有一个父`Show`对象，而`Show`对象含有一个包括原`Performance`对象的`Performance`对象列表。因此，我们还需要能够从变换中排除属性，以保证避免由于循环引用所造成的无限循环。`XsltView`类的`excludeProperties`属性是一个其列表项具有`<concrete class>.<property name>`形式的CSV列表，这个列表告诉Domify哪些属性要排除在外。我们还可以使用这个列表来简化Domify所暴露的DOM文档。在下文中，我们将会见到这方面的一个例子。

定义在应用中使用的XSLT视图

我们在应用代码中需要做的只是在WEB-INF/classes/views.properties文件中声明XsltView类型的新视图组件，指定文档根的名称、模板URL（在WAR描述符内）以及我们希望排除掉的属性。用于示例视图的定义如下所示：

```
showReservation.class=com.interface21.web.servlet.view.xslt.XsltView
showReservation.root=reservationInfo
showReservation.stylesheet=/xsl/showReservation.xsl
showReservation.excludedProperties=
    com.wrox.expertj2ee.ticket.referencedata.support.ShowImpl.performances
```

需要注意的是，如果我们没有指定一个格式表URL，输出结果将显示XML输入文档。这是开发期间非常有用的功能度，为了简单起见，笔者从上面的代码清单中省略了这个功能度，如表A.3所示。

表A.3

名称	类型	需要变化	默认值	用途
root	String	变化	null	XML生成文档的根。是必需的，除非此模型仅含有一个元素（在这种情况下该属性名被使用）。如果这个唯一的元素是一个XML节点，它将无变化地用做该模型的基础，并且根属性的任何值都被忽略
stylesheet	String	否	默认行为是显示未修改过的XML输入文档	指定XSLT格式表在WAR描述符内的位置。如果没有为这个属性指定值，该XML文档将被写给响应对象
cache	boolean	否	true	XSLT格式表是否应该在视图初始化时以编译形式被高速缓存，或者该格式表是否应该在每个响应生成时被重新读取。true值在调试期间是很有用的，因为它能使修改后的格式表在不必重启应用时生效，但不应该用在生产中
Excluded Properties	CSV格式 String	否	n/a	应该从“变换”中排除掉的属性（或全限定类）集合。请参见上面的讨论和示例

虽然框架的默认XsltView实现使用Domify库把Java组件模型数据转换到XML形式，并使用TrAX来执行XSL变换，但一个替代实现可以使用替代库来实现这些。例如，模型特有的实现（与这里所描述的通用实现相反）可以使用XML数据联编来实现更高的性能。

XMLC

本节描述如何安装和配置XMLC 2.1。关于XMLC创作模型的进一步信息，请参见第13章。

安装并配置XMLC

和Velocity一样，XMLC不是J2EE应用服务器（Enhydra除外）的捆绑品。因此，我们需要自己把它随同我们的应用一起发行。

示例应用下载的/lib/runtime/xmlc目录含有用于XMLC 2.1的所有二进制库。XMLC不仅需要像GNU正则表达式库那样的支持库，而且还需要像Apache Xerces XML语法分析器那样的常见产品的补丁版本。

这个目录含有下列文件：

- **xmlc.jar:** XMLC二进制库
- **xerces-1.4.4-xmlc.jar:** Xerces语法分析器的一个补丁版本
- **jtidy-r7-xmlc.jar:** 各种JTidy实用程序的一个补丁版本
- **gnu-regexp-1.1.4.jar:** GNU正则表达式库

所有这些文件都需要被包含在使用XMLC的Web应用的/WEN-INF/lib目录中。同样，示例应用的Ant生成脚本处理这件事情。

需要常用库的补丁版本是否有引擎兼容性问题可能视你的应用服务器而定。如果该应用服务器提供了针对第14章中所讨论的“反向”类装入(WAR优先)的Servlet 2.3支持，就不太可能有问题。但是，如果该应用服务器遵守了标准的Java语言类装入行为(系统类优先)，那些必需的补丁可能就得不到装入，这意味着修补该应用服务器自己的库，或者放弃XMLC。如果运气好的话，这个问题随着XMLC和相关库的新发布很快就会得到解决。

和Velocity一样，在可以用在Web应用中之前，XMLC需要先被配置。在前面，我们可以使用应用上下文中的一个组件来配置Velocity，但配置XMLC却需要使用一个服务器小程序。XMLCContext对象只能用一个服务器小程序来配置；ServletContext不是很有效的。我们的MVC框架提供com.interface21.web.servlet.view.xmlc.XmlcConfigServlet来配置XMLC，必须设置XMLC在应用启动时装入。XmlcConfigServlet按如下所示配置XMLC：

```
XMLCContext xmlcc = XMLCContext.getContext(this);
```

当使用XMLC时，下列服务器小程序定义元素必须被包含在每个Web应用的web.xml描述符中，才能在启动时装入XmlcConfigServlet。

```
<servlet>
    <servlet-name>xmlcConfig</servlet-name>
    <servlet-class>
        com.interface21.web.servlet.view.xmlc.XmlcConfigServlet
    </servlet-class>
    <load-on-startup>3</load-on-startup>
</servlet>
```

请根据应用所需要的预初始化服务器小程序的数量来设置<load-on-startup>值。

实现用于XMLC的视图接口

现在来看一看MVC框架为XMLC所实现的com.interface21.web.servlet.View接口的抽象实现，应用特有的XML视图都应该扩展这个抽象实现。

对于XMLC，如同对于生成库方法，我们为每个视图使用不同的Java对象。在运行时不再需要HTML模板，但对每个XMLC生成视图我们需要一个视图实现。

我们的MVC Web应用框架为所有XMLC视图提供了一个公用超类——com.interface21.web.servlet.view.xmlc.AbstractXmlcView，并使用Template Method设计模式从子类中隐藏必要的管道工程，而只给子类留下了创建和处理相关XMLC对象的任务。AbstractView类中的renderMergedOutputModel()保护方法的实现使用了由XmlcConfigServlet所创建的org.enhydra.xml.xmlc.servlet.XMLCContext对象，而该对象可以创建新的模板。下面是一个部分代码清单：

```
public abstract class AbstractXmlcView extends AbstractView {
    private XMLCContext xmlcContext;
```

AbstractView类中的下列替代方法高速缓存XMLCContext对象，当XMLC未得到配置时抛出一个异常：

```
protected void onSetContext() throws ApplicationContextException {
    this.xmlcContext = (XMLCContext) getWebApplicationContext().
        getServletContext().getAttribute(XMLCContext.CONTEXT_ATTRIBUTE);
    if (this.xmlcContext == null)
        throw new ApplicationContextException(
            "No XMLCContext initied. Use XMLCConfigServlet with loadOnStartup");
}
```

renderMergedOutputModel()模板保护方法的下列实现调用一个模板保护方法来创建应用特有的XObject，然后使用XMLCContext把这个对象的数据输出给响应对象：

```
protected final void renderMergedOutputModel(Map model,
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    XMLHttpRequest xo = createXObject(model, request, response,
        this.xmlcContext);
    response.setContentType(getContentType());
    this.xmlcContext.writeDOM(request, response, xo);
}
```

子类必须实现下列保护方法来创建一个应用特有的XObject，并根据模型参数中的数据来处理它：

```
protected abstract XMLHttpRequest createXObject(
    Map model,
    HttpServletRequest request,
```

```
HttpServletResponse response,  
XMLCContext context)  
throws ServletException;
```

定义在应用中使用的XMLC视图

由于AbstractXmlcView类没有暴露除了从AbstractView中继承来的组件属性之外的其他组件属性，所以我们定义一个XMLC视图所需要做的只是指定一个应用特有的具体子类，如下所示：

```
showReservation.class=  
com.wrox.expertj2ee.ticket.web.xmlcviews.ShowReservationView
```

当然，子类可能会自己增加视图属性，然后这些属性以设置框架属性的相同方式被设置。基于Java组件的配置方法的优点之一是：它自动处理任何类，而不是只处理框架已知的类。

利用iText生成PDF

本节将讨论如何使用iText库来生成PDF。

安装iText

iText仅仅是一个类库，所以没有必要像配置Velocity或XMLC一样配置运行时设置。但是，我们需要把二进制库包含在使用了它的WAR描述符内。示例应用的/lib/runtime/itext-pdf目录包含了所需要的单一JAR文件——iText.jar。同样，示例应用的Ant生成脚本将把iText.jar复制到它所生成的WAR分布单元的/WEB-INF/lib目录中。

利用iText实现用于PDF生成的视图接口

同XMLC的情形一样，我们将需要View接口的一个实现，以便用于每个生成PDF的视图。

使用iText生成PDF涉及到的基本步骤包括获得一个com.lowagie.text.pdf.PdfWriter实例，给该PdfWriter添加文档内容对象，以及关闭该PdfWriter来输出内容：

```
PdfWriter.getInstance(document, response.getOutputStream());  
document.open();  
// Output model data  
document.close();
```

但是，这种简单方法对Internet Explorer不管用，因为该浏览器需要知道被生成PDF的内容长度。因此，在Web应用中，必须使用如下所示的较冗长代码来执行相同的基本步骤。我们还需要给被请求的URL赋予一个.pdf扩展名。虽然AbstractPdfView把内容类型设置成application/pdf（应该足够了），但Internet Explorer不一定遵守这个设置，除非我们改变了该扩展名。

使用.pdf扩展名意味着我们必须记住把相关的.pdf URL映射到我们的控制器服务器小程序上。未在web.xml部署描述符中定义所有必要的映射是导致MVC框架令人失望的一个常见原因。如果框架的控制器服务器小程序早先从未获得一个请求，问题出在哪里可能不很清楚。

同样，我们的MVC框架使用Template Method设计模式，给iText PDF视图提供一个便利超类。com.interface21.web.servlet.view.pdf.AbstractPdfView抽象超类负责创建一个iText文档，并用renderMergedOutputModel()方法的一个终结实现把整个文档写给响应对象。该实现从应用代码中隐藏了生成内容长度和处理Internet Explorer未遵守被返回内容类型的复杂性。请注意，我们还设置了页面大小。

```
protected final void renderMergedOutputModel(Map model,
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    Document document = new Document(PageSize.A4);
    try {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        PdfWriter writer = PdfWriter.getInstance(document, baos);

        writer.setViewerPreferences(
            PdfWriter.AllowPrinting |
            PdfWriter.PageLayoutSinglePage);

        document.open();
    }
}
```

在创建了一个新文档之后，我们就需要调用应用代码把内容写到该新文档上。下面这一行代码调用一个模板保护方法（该方法的定义在下文中介绍）：

```
buildPdfDocument(model, document, request, response);
```

最后，我们把数据输出给HttpServletResponse，兼顾到使用Servlet API来设置内容长度。

```
document.close();
response.setContentLength(baos.size());
ServletOutputStream out = response.getOutputStream();
baos.writeTo(out);
out.flush();
}
catch (DocumentException ex) {
    throw new ServletException("Error creating PDF document", ex);
}
}
```

AbstractPdfView类的子类必须实现下列抽象保护方法，以便把模型数据写给被作为第二个参数传递的iText PDF Document。通常，不使用请求与响应对象，但我们包含了它们，以防该视图需要地区信息，或需要写一个cookie。

```
protected abstract void buildPdfDocument(Map model,
    Document pdfDoc,
    HttpServletRequest request,
    HttpServletResponse response)
    throws DocumentException;
```

定义在应用中使用的PDF视图

要定义一个PDF视图，只需在WEB-INF/classes/views.properties文件中指定该应用特有的子类即可，如下所示：

```
showReservation.class=
com.wrox.expertj2ee.ticket.web.pdfviews.ShowReservationViewExample
```

补充视图

下列这些视图包含在示例应用所携带的框架中，但本节不打算讨论它们。

- com.interfaxe21.servlet.web.view.webmacro.WebMacroView

WebMacro视图实现，类似于前面讨论过的Velocity视图。

- com.interfaxe21.servlet.web.view.RedirectView

把响应重定向给一个外部URL，把模型数据暴露为串类型查询参数的视图实现。这种方法只适用于简单模型，但在与第三方站点集成时会十分有用。默认实现通过在模型中的每个元素上调用toString()方法建立一个查询串。子类可以自定义这个行为，即通过替代queryProperties(Map model)保护方法来返回一个Properties对象，该对象含有要用在查找串中的名称-值对。RedirectView类负责构造查询串（使用?和&字符）及必要的URL编码。

自定义视图

正如前面所讨论过的各种视图实现所展示的，实现View接口是很容易的，尤其是如果我们子类化com.interface21.web.servlet.view.AbstractView便利类。

自定义的视图实现可以与几乎任何一种视图技术集成。我们只需提供一个实现类即可，而且这个类可以是通用的或应用特有的，并使用标准Java组件语法提供一个视图定义。

这种基本方法的威力可由这样一个事实来证明：不必修改本应用中的控制器或模型代码就可以轻松地为下述各视图编写一个View实现。

- 暴露XML和格式表，并使XSLT转换在客户浏览器上（而不是在服务器上）进行的视图。
- 包含了其他视图的输出，甚至可能使用了不同视图技术的合成视图。
- 检查浏览器所返回的用户代理串，并相应地选择一个委托视图的转发性视图（例如，如果已经知道该浏览器能够执行XSLT变换，控制权就可以被传递给一个启动客户端转换的视图；否则，XSLT转换可以在服务器上进行，然后把生成的输出结果返回给客户）。
- 使用类库为数字模型数据生成Excel电子表格的自定义视图。

使用这里所描述的这种方法，我们可以为能够接受模型数据的任何东西实现一个视图，并生成输出结果。如果必要，我们可以把模型数据转换成除Java对象之外的另一种形式。