J2EE Design Patterns Applied

实用J2EE设计模式编程指南

Craig A. Berry
(美) John Carnell 等著
Matjaz B. Juric

[illegible] 等译

目 录

第1章 J2EE设计模式

即使利用最先进的软件平台Java 2 Platform Enterprise Edition（J2EE），开发企业应用程序仍然是个难题。J2EE通过API提供技术与服务的高层抽象，使企业开发得到简化。但是，仅仅知道J2EE API是不够的。要设计良好的体系结构，得到高质量的应用程序，就要知道何时如何正确使用J2EE API。本书介绍如何正确使用J2EE API、技术与服务。

由于新技术方面的经验缺乏，我们通常要自己猜测如何正确使用。通常很难第一步就走对，而是要不断试验，直到找出最佳方法。最佳方法显然是从实践中得到的，不是发明出来的，而是发现和改善而成的。

工程学科中的一大原则是总结经验和利用实践证明有效的方案，企业应用程序也是这样。经验有助于更快更顺利地建立良好方案，从而节省成本，提高质量。惟一的问题是需要获得经验，但这个经验可以是别人的间接经验，而不一定非要是自己的直接经验。本书帮助读者节省自己的时间，介绍如何根据许多开发人员的间接经验进行合理设计。

但是，别人的经验如何描述呢？多年来，模式已经成为收集、规范和分析某些情境中常见问题的有效方法。本书主要介绍J2EE设计模式，不仅介绍J2EE企业开发中的常见问题及其解决方案，而且介绍其如何运用到实际项目中。

本章概述模式，并特别介绍设计模式与J2EE，同时介绍一些准则，如怎样有效利用设计模式。本章介绍的内容包括：

- 模式的演变
- 何谓设计模式
- 标识模式
- 设计模式如何解决设计问题
- 选择设计模式
- 使用设计模式
- 反模式
- J2EE与设计模式
- J2EE特定模式与J2EE情境中的模式
- J2EE模式的问题域

模式的演变

设计模式是从建筑与民用工程开始的。20世纪70年代，建筑学是需要大量经验的学科。Chris-topher Alexander等的三本著作震惊了整个行业，使不需要太多专门知识与经验的人也可以使用建筑学。他们标识有效结构之间的相似性，确定共同原则，作为常见设计问题的方案，并将其命名为建筑学中的方案“模式”。

但是，发布253个模式之后，Christopher Alexander仍然没能使建筑学成为机械化过程。那么，这些建筑模式有什么重要性呢？模式的目标并不是使建筑学成为机械化过程，而是为了普及常见建筑问题的解决方案，使经验较少的建筑师能更高效地进行设计，这是建筑模式的主要目标。

建筑模式对其他工程学科产生了巨大的影响。显然，每个成熟的工程学科都应建立常见问题的解决方案，软件工程也不例外，可以用模式成功地描述常见软件问题的解决方案。模式不仅提供良好方案，而且还可以帮助人们进行沟通，帮助他们分析工作和寻找原因。

软件工程中的模式

软件工程中的模式是什么呢？没有一个公认的定义。

简单地说，模式就是情境中一个问题经过证实的一个方案。

但是，这个简单定义可能造成对模式的误解。Christopher Alexander写道：

“每个模式描述一个在我们的环境中不断重复出现的问题，然后描述这个问题的方案核心，使你可以多次使用同一方案，而不必进行重复工作。”

近年来关于软件工程中的模式的最有影响的出版物是《Design Patterns: Elements of Reusable Object-Oriented Software》，作者Erich Gamma、Richard Helm、Ralph Johnson与John Vlissides称为四人帮；Addison-Wesley出版公司出版（ISBN: 0-201-63361-2）。其定义模式如下：

“模式是三段值，表示特写情境、问题与方案之间的关系。”

根据这个定义，模式是多种情形中可以使用的问题解。

Richard Gabriel提供了一个有趣的定义：

“每个模式是三段值，表示情境，这个情境中重复出现的问题及解决这些问题的一定软件配置之间的关系。”

模式是从经验中总结出来的，基于经过证实的方案。模式只有在实际系统中多次得到验证之后才能成为模式。因此，模式一方面促进复用，一方面又防止重复劳动，最终使我们能更快更高效地工作。

模式还增加表达能力，改进建筑师与设计人员之间的通信，使我们可以按前所未有的方式考虑常见结构性方案。模式鼓励通过结合解决大问题。

由此可见，模式没有什么新东西。有经验的设计人员可以阅读模式和发现过去的处理方法。众所周知，专家并不从低级结构考虑问题，而是建立高级抽象。但模式则鼓励人们标识与记录这些高级抽象。标识模式的动机包括：

- 寻找模式靠经验而不是靠想像。想像会引入风险，因为新的方法与技术要经过验证与测试之后才能证明有用。实际中，成功通常比想像更重要。因此，新的模式要经过试验才能评估其价值。好的模式是从实践中总结出来的。

- 模式能改进开发人员之间的通信，使他们可以更快地学习，从而开发更好的软件。为了达到高效通信，我们要用某种方式表示模式，最好使用标准格式。稍后将介绍表示模式的格式。
- 模式可以按照质和量验证知识，从而评估其价值。这样可以表示与理解模式及所建体系结构的利弊。

模式几乎无所不在。多年来，出现了多种不同的软件模式，包括：

- 设计模式，包括软件设计，可能是最重要的模式。通常是面向对象的，包括体系结构（系统设计）、设计（组件交互）和编程（特定语言技术）。本书主要介绍设计模式。
- 分析模式，描述可复用分析模型，在域分析中非常有用，涉及各种域，包括交易、度量、会计和组织关系。要了解分析模式的更多信息，见《Analysis Patterns: Reusable Object Models》一书，Addison-Wesley出版（ISBN 0-201-89542-0）。
- 过程模式，描述软件过程设计。具体地说，它们描述开发软件成功而经过证实的方法与活动。详见剑桥大学出版社的《Process Patterns: Building Large-Scale Systems Using Object Technology》（ISBN 0-521-64568-9）与《More Process Patterns: Delivering Large-Scale Systems Using Object Technology》（ISBN 0-521-65262-6）。
- 组织模式，描述组织与项目的结构和实践。详见http://i44pc48.info.uni-karlsruhe.de/cgi-bin/OrgPatterns。
- 实现模式，描述实现概念。详见《Essential Java Style: Patterns for Implementation》一书，Prentice Hall出版（ISBN 0-13-085086-1）。
- 其他域特定模式。

模式还可以根据包括的问题分为：

- 创建性模式
- 结构性模式
- 行为性模式

可以看出，模式有几个抽象层，可以按不同方式分类。本书主要介绍J2EE设计模式。首先介绍设计模式。

何谓设计模式

设计模式是情境中标准设计问题的重复性解决方案。设计问题必须进一步调查之后才能解决。问题通常发生在一定的环境或情形中，称为情境。解决方案是这些问题的答案，帮助我们在一定情境中解决这些问题。

例如，假设有一个小房间，问题是把门放在哪里更方便。经过不同测试之后，我们可能发现应把门放在房间的东南角。这样就找到了特定问题的解。同样，软件设计中也可以找到特定问题的解。

设计问题的解是否都是设计模式呢？不一定。设计模式只是适用于不同情境的解，即可以重复采用，在不同情境中解决同一问题。

回到房门的问题，这个解还不能作为模式，因为它只限于这个特定情形。但如果能对所有小房间找到一般解，则可以称为模式。Alexander就是这么干的。他发现，房门不能放在墙上任何地方。此外，房间的成功很大程度上取决于门的位置。因此，他建议除了很大的房间之外，房门应尽量靠近墙角。他把这个模式称为角门（Corner Doors）模式。

虽然这是一个简单概念，但定义设计模式并不容易。因此，人们提出了五花八门的定义。为了深入了解设计模式，下面再进一步解释。

设计模式针对软件设计，系统化地命名、解释和求值重要软件设计。设计模式使成功地经过证明的设计与体系结构更容易复用，使新系统开发人员能更方便地采用设计模式，特别是经验较少的开发人员。设计模式能帮助我们选择不同设计。本书主要介绍设计模式，因此此后“模式”一词特指设计模式。

假设要开发Customer实体Bean，并把客户数据提供给客户机。自然方法是用细粒方法定义远程接口（如get/setName()、get/setAddres()），让远程客户机直接访问这个实体Bean。要确定这个解不可伸缩，因此不适合实际应用，需要实现组件与客户机，进行部署和测试，需要进行大量工作。记住，实际应用程序中通常要面对多个组件，而不只是Customer。

知道Value Object与Session Façade之类的设计模式之后，就很容易看出EJB的远程接口应是粗粒的。Value Object（数值对象）模式显示了如何使接口变成粗粒，同时以巧妙的方式传输所有必要的数据，避免多次远程方法调用。Session Façade模式显示不能直接访问实体bean，而应开发会话bean，作为门户。如果能对实体bean增加一个本地接口，在同一容器中部署两个EJB，则可以进一步提高性能。

如果你不熟悉Value Object与Session Façade模式，则不容易理解上段的内容。别担心，这些模式都将在本书稍后详细介绍。但可以注意，这两个模式都适用于某种情境。所有模式都适用于某种情境，是一组矛盾的平衡。描述模式时，我们要明确定义所有这些项目。根据前面提到的四位专家对设计模式的描述，模式具有四大要素：

- 模式名，标识模式，增加表达性。
- 问题，描述何时应用这个模式，解释问题与情境。
- 解，不描述具体设计与实现，而是描述模式模板，可以在不同情境中使用。
- 结果，是采用一个模式的结果与取舍。结果对评估不同方法和进行决策至关重要。

设计模式描述如何在特定情境中解决一般设计问题。

设计模式对常见设计结构的关键方面进行命名、抽象和标识。标识方案参与者，他们的角色与责任及合作方式。大多数情况下，这些参与者是对象与组件。每个设计模式针对特定问题。每个模式还描述其结果和取舍。

相关模式交织在一起，构成模式语言。模式语言不是正式语言，而是一组相关模式的集合。模式和模式语言有助于更好、更快、更有效地学习、通信和解决问题。本书主要介绍J2EE的模式语言，但介绍J2EE设计模式及其应用之前，先要介绍如何标识模式。

标识模式

标识模式要靠经验。每个有经验的建筑师和设计人员能够看到，在大部分项目中会遇到类似问题，这些问题的解也是相似的。当然，实际解不一定是相同的。但是，所有解具有相同的基本概念。从抽象角度看，可以说所有这些解与问题表示具有共同抽象解的共同抽象问题。

了解这一点是标识模式的基础。我们应回顾过去的项目，标识处理过的问题。对每个问题，应标识其特征，还应记录这些问题的解。然后收集类似问题与解，确定其相似性。

类似解很可能表示一个模式。前面曾介绍过Alexander的角门模式，Alexander通过大量房间中门的布置位置找到这个模式。所有把门放在角上的房间都采用这种模式。前面介绍的Value Object与Session Façade模式也是这样，也是通过分析多个应用程序而得到的。

但是，一组解只有经过验证之后才能成为模式。由于Alexander验证把门放在角上是有用的，我们也要验证所找到的模式。在软件工程中验证复杂模式可能比验证角门模式之类的简单建筑模式更复杂。

因此，我们不是直接标识模式，而是标识模式候选者。模式候选者是独立方案，与外部的连接越少越好。记住，模式是为了复用。我们要验证模式候选者，通常使用三规则，即每个模式候选者至少要在三个不同系统中证明之后才能成为真正的模式。这个规则不太精确，因此本书作者建议模式候选者应尽量多次验证，使模式更有实用价值。

没有充分验证的模式可能带来大量危害。为什么呢？因为大多数建筑师、设计人员和开发人员并不标识模式，可能没有时间，也可能没有这方面的经验。但他们通常都学习模式和用其改进解决方案。无法达到目标的模式可能对大量应用程序造成伤害，从而给模式带来不好的名声。

另一个重要问题是模式应采用哪一级抽象，包括多少实现细节。J2EE模式社区中普遍接受的概念是模式应采用高层抽象，但每个模式应包括解的细节。这些解的细节通常比模式本身采用更低层抽象，称为策略。显然，模式与策略之间没有明显的分界。

由于本书介绍模式，因此会通过这些策略显示不同情境中如何采用模式。

在Prentice Hall PTR出版的《Core J2EE Patterns》一书（ISBN 0-13-064884-1）中，作者定义了模式与策略的下列差别：

- 模式应采用更高层抽象，因此应提出最佳实现策略。
- 开发人员可以通过策略扩展使用模式，寻找实现模式的新方法。
- 模式与策略名称改进通信。

标识模式之后，要表示模式。下节介绍如何表示设计模式。

表示设计模式

表示模式和定义模式一样难，仅仅靠统一建模语言（UML）之类的简单图形表示方法是不够的。为了使模式真正促进复用，就要表示意图、动机、决策、后果，等等。要成功运用

模式，就应提供具体例子。

近年来，人们找到了多种模式。标识这些模式的作者将其发表到模式类别中。也许最重要的模式类别是前面提到的四位专家在设计模式中提供的23种模式。对J2EE开发人员，模式类别是个社区过程，见Java Developer Connection站点http://java.sun.com/。J2EE开发人员的另一重要模式类别见http://www.TheServerSide.com/。

模式可以用正式与非正式方式表示。如今大多数模式类别用非正式方式表示模式。但问题是他们不用相同格式，而采用稍有不同的格式。模式表示的主要差别源于不同类型模式解决的设计问题并不相似。例如，前面提到的四位专家的模式解决面向对象设计问题，而核心J2EE模式解决建立J2EE应用程序时遇到的问题。

非正式方式表示模式时通常使用几段，有些是大多数表示中都有的，有些是不同的。大多数非正式方式表示基于前面提到的四位专家的著作中使用的表示。前面提到的四位专家的著作中使用的表示采用下列模板：

- **Pattern name and classification（模式名与类别）**
  每个模式应有惟一名称，使模式便于口头表达。
- **Intent（意图）**
  简要描述模式的作用、理由与意图、解决的设计问题。
- **Also Known As（别名）**
  提供模式的别名名单。
- **Motivation（动机）**
  描述问题内容及如何用这个模式解决这个问题。
- **Applicability（适用性）**
  描述设计模式的适用情形，不好的设计例子及如何识别。
- **Structure（结构）**
  提供图形表示，可以使用UML图形。
- **Participants（参与者）**
  显示参与这个设计模式的类、对象和组件，及各处的责任。
- **Collaborations（协作）**
  描述参与者如何协作。
- **Consequences（后果）**
  列出模式如何达到目标，使用模式的结果与取舍。
- **Implementation（实现）**
  实现模式的提示、技术与策略，以及特定语言问题和注意事项。
- **Sample code（样本代码）**
  显示如何实现模式样本代码。
- **Known uses（已知用途）**
  实际系统中如何使用这个模式。
- **Related patterns（相关模式）**
  列出相关模式及主要差别。

介绍本书提到的模式之前，先要简单介绍设计模式的好处及其如何帮助解决问题。

设计模式如何帮助解决问题

了解模式之后，下面看看设计模式如何帮助解决问题。

设计模式可以帮助进行软件设计和更好地设计软件。

一般来说，设计模式可以帮助我们解决应用程序设计阶段遇到的大多数常见问题，包括：

- 标识组件、组件内部结构及组件之间的关系
- 确定组件粒度及适当交互
- 定义组件接口

J2EE平台设计模式解决使用J2EE服务与技术的常见设计问题，包括：

- 视图管理
- 请求处理
- 服务定位与激活
- 远程通信与层间通信
- 组件选择
- 持久状态、事务与安全性管理
- EIS集成

设计模式还可以帮助我们进行设计：

- 更适合于复用
- 更壮实，便于今后修改

J2EE应用程序是由组件构成的，放在客户层、Web组件层和业务逻辑（EJB）层。Java语言开发的组件采用面向对象方法。设计这类应用程序时，建筑师要面对不同问题：如何标识组件、选择组件类型和标识组件的内部结构（如作为组件建筑块的对象）。确定正确的抽象层、粒度与灵活性是很困难的，需要有足够的经验。

这时软件开发方法不一定能帮上忙。尽管可以用不同方法标识与分析模型对象，但通常不是显式针对设计模型组件与对象的。实际开发还显示，应用程序中的组件不一定直接模拟实际对象，但分析模型中通常根据实际对象定义对象。

因此，要选择J2EE提供的适当组件类型并不容易，特别是在选择不明显时，有时还要考虑性能与安全要求。Web组件也是这样，可能要选择JSP（Java Server Pages）或小服务，以及业务逻辑层组件，首先要选择不同类型的EJB（无状态会话、状态会话、实体和消息驱动bean）以及选择CORBA与RMI-IIOP分布式对象和JMS。这个决策会产生长远的影响。

采用设计模式时，可以得到这些决策的准则。下面是设计模式的另一些好处：

- 帮助我们进行适当抽象
- 帮助我们进行适当一般化
- 帮助我们确定支持复用的适当粒度
- 帮助我们提高设计灵活性，更好地适应将来的改变

这里，粒度起着重要作用。选择组件的适当粒度可以规定所需的通信量，这在分布式应用程序中特别重要（J2EE应用程序是分布式应用程序）。选择组件的适当粒度可以使应用程序性能更好。

粒度选择反映在组件接口中。J2EE是个分布式平台，严格遵照接口与实现分开的概念，减少客户机与组件之间的依赖性，从而提高灵活性。客户机只依赖于接口，因此不知道组件实现的改变。这样就可以改变组件实现而不影响客户机。这个概念对分布和集成非常重要。

但是，要得到这些好处，就不能改变接口。这就要求尽力设计接口，保证近期内不会发生改变。设计模式可以定义具有适当粒度与签名的接口。

设计模式还可以提供视图管理体系结构，帮助设计灵活的表示层组件，适应将来的修改和集中执行特定工作，如用户验证、授权和个性化，还可以分开请求处理与视图生成，从而进一步增加灵活性。

设计模式还可以促进复用。软件开发中的复用已经有较长的历史，从简单源代码复用进化到表示问题抽象的结构复用，以及表示问题解决思路的结构复用。模式就是复用思想，已被证明是成功的方法，是思想复用的催化剂。设计模式还可以帮助我们使设计更适合低级复用，复用组件与对象。从这个意义上说，模式也许是最成功的复用形式之一。

最后，设计模式还可以提高设计灵活性，更好地适应将来的改变。软件开发中的需求是迅速变化的。好的软件设计可以预期这些改变，但要求根据领域中不易发生改变的项目进行设计，要求从开始就预见将来可能的改变。这样可以大大减少重新设计。

一般设计模式和J2EE设计模式可以避免这些情形。如果使用设计模式，则可以使体系结构更加灵活。这是因为，每种设计模式定义了一般方案，可以独立改变某些方面。

要实现设计模式的这些好处，就要选择适当的设计模式。因此，下节介绍如何选择适当的设计模式。

选择适当的设计模式

近年来，设计模式不断增加，因此选择适当的模式并不容易，特别是类别中列出不熟悉的设计模式时更是如此。选择适当的模式非常重要，因为如果不针对问题选择模式，就得不到设计模式应有的好处。因此，使用模式类别之前，要先仔细研究和了解所有模式。

了解所有模式之后，仍然可能在特定设计问题中找到多种选择。例如，要隐藏业务逻辑层细节和分离EJB与客户机之间的通信，J2EE模式类别中至少列出了三个相似的模式：Session Façade、Message Façade和EJB Command。然后我们可以采用几种不同方法寻找与选择适当的模式。下面介绍其中一些方法：

- 模式类别通常根据用途组织模式，不难找到适当的组。然后可以分析这个组中的模式，选择最适合的设计模式。
- 对每个设计模式，找到这个模式解决的设计问题，从而缩小具体问题可用的设计模式。
- 阅读每个模式描述的问题和解部分。问题部分有助于标识适合的模式，然后可以根据解部分进行最后选择。

- 确定模式之间的相关性。可以用关系选择可用模式组。
- 一个有用的方法是考虑设计中的可变参数。应确定可以改变什么而不必重新设计。“包装变化概念”在许多模式中定义。了解之后，就可以根据这个条件选择适当的模式。
- 相反的方法是标识重新设计的原因。我们要标识造成重新设计的原因，然后找到避免这些原因的模式。

活动框图1.1显示了上述步骤。

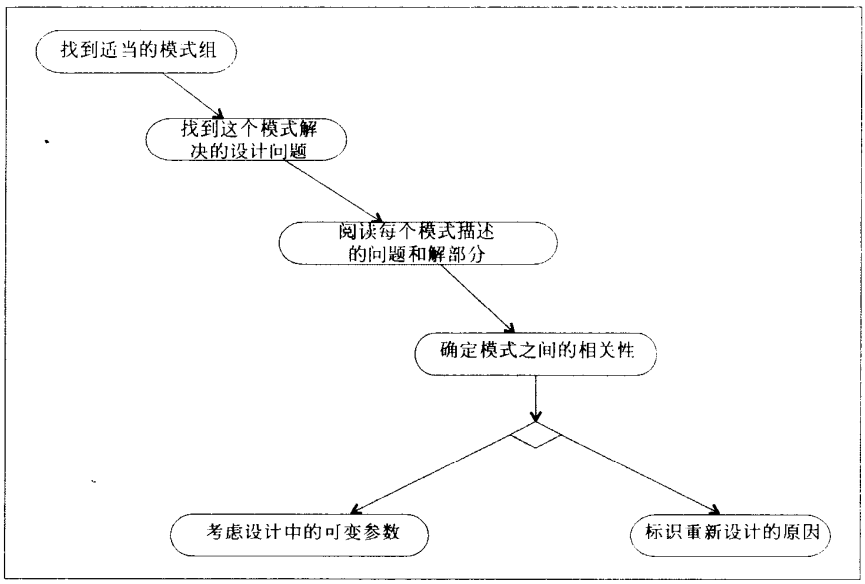

图1.1

初看起来，这些步骤好象很复杂，但本书提供了丰富的准则，可以帮你进行模式选择。对模式越熟悉，就越容易进行模式选择。

下面看看进行模式选择之后如何使用设计模式。

使用设计模式

要得到设计模式的好处，一定要进行正确的模式选择，更重要的是正确应用选择的设计模式，否则就无法得到设计模式的好处。

要有效利用设计模式，应遵循下列准则：

- 使用模式之前，应完全了解设计模式。这就要求分析解部分，了解参与模式的所有组件、类与对象及其相互关系，还要熟悉策略部分，了解不同情形中采用模式的准则。
- 再次检查是否选择了适当的模式。应认真分析模式描述，特别是后果与相关模式部分。
- 应检查采用模式部分，认真分析例子。如果已经熟悉这个模式，则只要看看策略部分的样本代码，目的是学习如何实现这个模式。

- 应当选择对应用程序有意义的参与者名称。模式类别中的名称通常太抽象，不能直接使用。选择适当的名称可以简化进一步开发工作，使设计（和代码）更容易理解。
- 下一步是定义接口、组件、类和关系。特别地，设计中与其他类的关系非常重要。从关系中可以看到模式影响的其他类。通常，我们要将这些类稍作修改，最后避免类之间的任何耦合。
- 还应选择模式中的适当操作名。这时模式类别中的名称通常也太抽象，还要注意遵循命名规则。
- 最后，要实现模式。这里可以参考采用模式部分，其中列出了正确实现每个模式的具体例子。

图1.2显示了上述步骤。

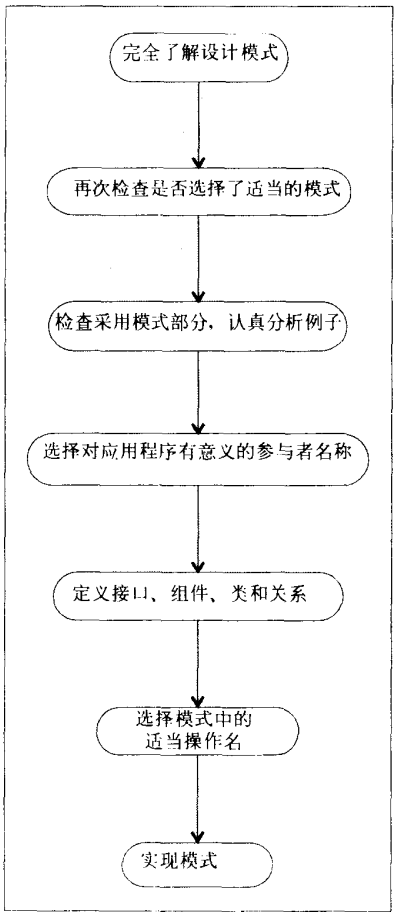

图1.2

注意模式在设计中引入了一定的灵活性，通过增加了另一个抽象层，可能使设计更复杂，影响性能。因此，一定要知道为何使用模式，能够确定实际需要，还要考虑是否需要模式提供的灵活性。不能为使用模式而使用模式，还要考虑（和了解）每个模式的后果。

下节介绍何时不适合使用模式，以及因素改变的问题。

因素改变

前面曾介绍过，设计模式对常见设计问题提供了有用的解决方案。但是，仅靠学习许多设计模式并不能成为好的设计人员。我们要了解模式和评估何时使用这些设计模式，采用某个设计模式有什么好处。

我们还要注意，使用太多模式和使用方式不对可能造成过量工程。过量工程就是使体系结构变得太复杂。过量工程的主要原因是想生成灵活的体系结构，适应将来所有可能的改变。但关键是不要让代码成为过量工程，否则会浪费时间和内存。

另一方面，还要注意不足量工程。不足量工程就是只考虑尽快在系统中增加功能，而不改进设计，从而引入定时炸弹。使用这种方法时，第一版能很快开发完成，但后面迟早会遇到麻烦。

在这两种情形中，都可以使用因素改变。

因素改变是一种“行为保留变换”。Martin Fowler指出，它是“对软件内部结构的改变，便于理解和修改，不需要改变其外部行为”。

如果连续采用因素改变，则可以一方面改进应用程序设计，一方面防止过量工程和不足量工程。因素改变时，通常可以利用模式。实践经验表明，我们通常在因素改变时标识模式，称为模式的因素改变，对应于Martin Fowler的说法是：“模式是目标，而因素改变是到达这个目标的途径。”关于因素改变的详细信息见《Refactoring: Improving the Design of Existing Code》一书，Addison-Wesley出版（ISBN 0-201-48567-2）。

前面介绍了模式文档记录经过证明的良好设计方法，但也有一些不好的设计方法。记录不好的设计方法有助于避免使用这些方法，这就是反模式。下节介绍反模式。

反模式

设计模式文档记录经过证明的良好设计方法，其中的模式能够达到目标。也有一些不好的设计方法，不能够达到目标。人们很少提到这些应用程序的设计与体系结构。这些应用程序的建筑师错在哪里呢？记录这些错误决策有助于防止在自己的应用程序中犯同样的错误。

例如，第一个J2EE应用程序中常见的错误决策是用细粒接口建模实体Bean和直接访问。这样就会增加大量远程方法调用和事务管理开销，使应用程序的性能与伸缩性很差。

这类错误记录在反模式中。反模式描述不良方案，不良方案会造成不理想和无法预见的不利结果。了解不良方案及其不利结果有助于今后避免重走老路。反模式可以避免使用不良方案和恢复与纠正这些错误，因为反模式提供了成功开发的因素改变方案。

因此，可以把反模式看成与模式相对。模式标识可行的方案，而反模式标识不可行的方案或用到错误情境中的方案。尽快标识错误是减少风险的关键，因此，反模式对每个设计人员、建筑师和开发人员都很重要。

反模式可以在软件开发的各个阶段标识，因此可以像模式一样进行分类。像模式一样，我们要标识反模式，然后通过因素改变改进设计。本书不介绍反模式，但下列章节会举一些错误决策例子，将其归类为反模式。关于反模式的详细信息，见下列著作：

- 《AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis》John Wiley & Sons出版（ISBN: 0-471-19713-0）
- 《AntiPatterns In Project Management》John Wiley & Sons出版（ISBN: 0-471-36366-9）
- 《Java AntiPatterns》John Wiley & Sons出版（ISBN 0-471-14615-3）

熟悉设计模式的一般知识和因素改变与反模式之后，下面介绍Java 2 Platform Enterprise Edition（J2EE）中的设计模式。

J2EE与设计模式

J2EE提供了大量开发企业应用程序的技术。J2EE v1.3支持下列技术：

- **Enterprise JavaBeans（EJB）version 2.0**提供EJB层开发、部署与管理业务逻辑层组件的服务。
- **JavaServer Pages（JSP）version 1.2**可以开发动态Web用户界面。
- **Java Servlets version 2.3**提供扩展Web服务器访问业务系统功能的机制。
- **JDBC API version 3.0**提供与关系型数据库兼容的服务。
- **Java Message Service（JMS）version 1.0.2**是访问面向消息中间件（MOM，Message-Oriented Middleware）的标准化API，支持点对点模型与发表/预订模型。
- **Java Remote Method Invocation, RMI-IIOP**是Java 2 SDK version 1.3, Standard Edition的一部分，提供ORB服务，支持分布式对象与组件之间的透明远程方法调用。ORB是与协议独立的，目前支持RMI自然协议（JRMP）和CORBA兼容IIOP协议。
- **Java Interface Definition Language（IDL）**，是Java 2 SDK version 1.3, Standard Edition的一部分，是CORBA兼容ORB，实现使用IIOP协议的外部CORBA分布式对象之间的相互操作性。
- **Java Transaction API（JTA）version 1.0.1与Java Transaction Service（JTS）version 1.1**支持事务并提供应用程序层事务分界的接口。
- **Java Authentication and Authorization Service（JAAS）version 1.0**提供安全性服务，特别是验证与授权，提供验证用户的可插入验证模块（PAM，Pluggable Authentication Module）框架实现。
- **Java Naming and Directory Interface（JNDI）version 1.2**是Java 2 SDK version 1.3, Standard Edition的一部分，是访问名称与目录服务的标准化API。
- **Java API for XML Parsing（JAXP）version 1.1**支持处理XML格式数据，提供文档对象模型（DOM，Document Object Model）、XML的简单API（SAX，Simple API for XML）和XML样式表单转换语言（XSLT，XML Style Sheet Language for Transformations）转换引擎。

- **J2EE Connector Architecture version 1.0**是个服务提供者接口，可以开发资源适配器，访问企业信息系统。它定义J2EE兼容服务器与资源适配器之间的系统级合约。
- **JavaMail version 1.2**提供管理E-mail的API，要求有JavaBeans Activation Framework（JAF）。

除了每个J2EE 1.3兼容应用程序服务器都有的技术之外，还有其他JAX（Java API for XML）接口：

- **JAXM - Java API for XML Messaging**
  提供的API可以用ebXML.org、Oasis、W3C与IETF定义的线路协议包装与传输业务事务。
- **JAX/RPC - Java API for XML-based Remote Procedure Calls**
  支持基于XML的RPC标准。
- **JAXR - Java API for XML Registries**
  提供一组分布式注册服务的API，可以用ebXML.org定义的协议实现业务企业之间的企业对企业（B2B）集成。
- **JAXB - Java API for XML Binding**
  可以将XML Schema编译成一个或几个Java类，可以分析、产生和验证符合Schema的文档。
- **JWSDL - Java Web Service Definition Language**
  提供一组标准API，表示和操纵Web服务描述语言（WSDL，Web Services Description Language）文档描述的服务。这些API定义了构造与操纵服务描述的模型。

JAXP的新版本（1.2）正在开发之中。为了便于安装和使用这些API，Sun公司在Java XML Pack（JAX Pack）中收集了下列技术：JAXM、JAXP 1.2、JAXR与JAX-RPC。Sun公司还发布了Java Web服务开发包（WSDP，Java Web Services Developer Pack），包括Java XML Pack、JSP Standard Tag Library（JSTL）、Ant Build Tool、Java WSDP Registry Server、Web Application Deployment Tool和Apache Tomcat dev容器。必须先熟悉这些API之后再开发Web服务和采用相关模式。

除此之外，还有一些API原先是可选补充的，现已加进J2SE 1.4中，成为J2EE 1.4的一部分。这些API包括：

- Java安全套接扩展（JSSE，Java Secure Socket Extension）可以用SSL与TLS协议进行安全通信。JSSE包括加密、验证和验证消息完整性的功能。
- Java加密法扩展（JCE，Java Cryptography Extension）提供加密、密钥生成、密钥协议和消息验证码（MAC，Message Authentication Code）算法的框架与实现。
- Java一般性安全服务（JGSS，Java Generic Security Services）是一般性验证与安全消息接口的API，支持可插安全机制。
- Java管理扩展（JMX，Java Management Extensions）是管理和监视的API。

要使用J2EE，就要学习所有标准J2EE技术，最好还要学习上述补充选项。这并不简单，但仅仅学习这些技术还不足以在J2EE中设计良好的应用程序，因为这些技术只回答了如何使用的问题，而没有回答何时使用和为何使用。学习这些技术也不同于学习在J2EE中设计应用

程序。设计应用程序时，需要确定技术和体系结构。

学习J2EE技术是学习如何用J2EE设计应用程序的基础。

如果要成为成功的设计人员，就要知道更多的知识，包括：

- 哪个问题用哪个技术
- 如何最佳利用，即每种技术的最佳用法
- 常见问题及何时不能用某个技术
- 常见问题的最佳方案
- 如何改进不良方案

每个J2EE设计人员、建筑师和开发人员都要在使用J2EE技术时面对这些问题。小服务、JSP、EJB、JMS、JNDI、JCA和其他J2EE技术能够很好地为开发人员抽象技术细节，但我们仍然需要知道其如何使用。要知道其如何使用，就要对基础概念进行深入研究，即J2EE如何抽象和隐藏基础细节。

过去，大量应用程序以错误的方式使用J2EE技术，使应用程序性能不好，无法满足目标与要求。说实话，这些问题不仅在J2EE中有，每种技术都可能因使用不当而建立无用的应用程序。

尽管J2EE从高层抽象基础技术，但我们仍然要建模和设计方案，利用这些技术。J2EE设计人员、建筑师和开发人员要寻找上述问题的答案，实际中可能更具体，例如：

- 特定任务最适合哪种企业Bean
- 如何设计EJB以达到性能和伸缩性要求
- 如何充分利用容器管理事务、持久性和安全性
- 是否每次查找组件或存储引用
- 把会话数据存放在哪里
- 用JSP还是用小服务
- 是否分别处理内容与表示
- 谁负责数据验证
- 如何定义实体Bean之间的有效关系
- 如何减少远程方法调用
- 层之间要发送多少数据
- 如何集成现有应用程序

这些问题是每天都要回答的。J2EE设计模式开始回答这些问题。如今，J2EE模式有两大来源，一个在TheServerSide.com模式仓库中，可以从http://www.theserverside.com/patterns/index.jsp联机访问，其中最重要的模式见《EJB Design Patterns: Advanced Patterns, Processes, and Idioms》一书，John Wiley & Sons出版（ISBN 0-471-20831-0）。

另一个重要J2EE模式源是Sun Java Center，定义了15个J2EE模式，可以从http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns联机访问，见《Core J2EE Patterns》书，Prentice Hall出版（ISBN 0-13-064884-1）。

J2EE中为何使用模式

J2EE中使用模式具有使用一般模式的所有好处和针对J2EE开发的好处。因此，这些模式针对J2EE，不像一般模式那么抽象，从而简化了特定系统和应用程序中的标识、使用与适配。

正确使用J2EE模式可以改进应用程序设计。前面曾介绍过，我们要先了解模式之后才能使用模式。尽管模式初看起来很简单，但通常不那么容易采用，因此本书花大量篇幅演示如何以最佳方式采用模式。

J2EE模式建档设计与开发J2EE应用程序时所遇到的常见问题的最佳解决方案。这些方案是经过实践证明的，已经多次使用，能够在不同项目中成功地解决类似问题。

J2EE模式定义了J2EE开发人员的共同词汇，使开发人员之间可以更好地交流。使用模式的每个开发人员能够很快在与别的开发人员交流时使用模式名。

由此可见模式命名是非常重要的。尽管还没有J2EE模式的命名标准，但情况是不错的，只有几个几乎一致的模式采用不同名称。同一模式采用不同名称容易造成混乱，但这种情形很快会得到解决。

使用模式还有另一个重要好处，即限制解的空间。模式定义了采用解决方案的边界，因此向开发人员提出了边界。

J2EE特定模式与J2EE情境中的模式

我们已经介绍J2EE设计模式的概念，那么，J2EE模式与通用设计模式有什么关系呢？大多数通用设计模式也适用于J2EE平台，称为J2EE情境中的模式。有些模式是J2EE特定模式，解决J2EE特定问题，称为J2EE特定模式。

将模式进行分类时，我们发现设计模式由一般到具体。根据其一般性，可以将模式进行如下分类：

- 基础设计模式，不与特定域、平台或编程语言相关联，如前面提到的四位专家定义的Façade是个基础设计模式。
- 设计模式，与特定平台或编程语言相关联，是用特定平台或编程语言部分或完全实现的（如Java、J2EE）。例如Session and Message Façade、EJB Command、Generic Attribute Access，等等。稍后将概要介绍J2EE模式。
- 设计模式，与特定域相关联，表示有限情境中问题的解，例如与事务、持久性、EIS集成等相关的模式。域模式可以直接从基础设计模式派生，也可以定义为特定平台模式的规范。

要确定模式是否与特定平台或编程语言相关联，需要了解这个平台或编程语言。前面曾介绍过，模式是用特定平台（J2EE）或编程语言实现的成功方案，然后进行一般化。重要的问题是一般化能否不用特定结构的平台或编程语言表示这个模式。

另一个标准是模式是否应用特定程序域，越不用程序特定域，模式使用范围就越广。

另一个条件是模式是否是原子的与基本的，或是由另外几个模式建成的。尽管这些分析很有趣，但对模式应用并不重要，只是有助于更好地理解模式。

回到J2EE模式，可以看到J2EE模式中有针对J2EE与Java的模式，也有J2EE平台中采用的通用模式。有些J2EE模式是特定域，而有些则可以应用到不同域中。后面将会介绍，本书根据技术域将J2EE模式进行分类，从而简化选择和应用。

图1.3显示了模式层次。

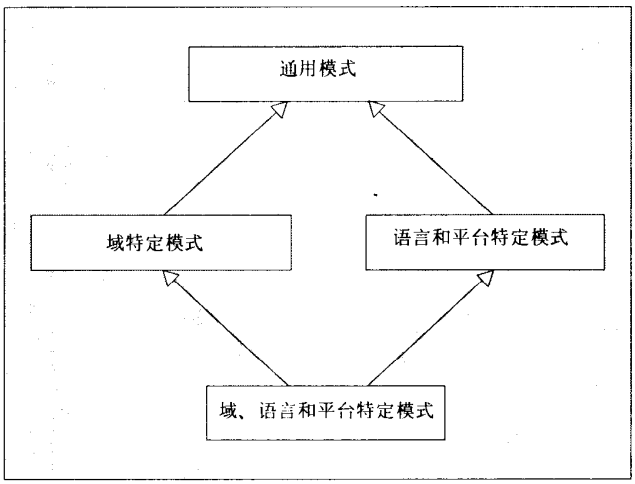

图1.3

J2EE模式体现了Java设计人员、建筑师和开发人员的经验与智慧。下节简要介绍最重要的J2EE模式。

最重要的J2EE模式

前面曾介绍过，J2EE有两大类重要J2EE模式，一类由Sun Java Center管理，定义15种模式，在《Core J2EE Patterns》书中发表，另一类是TheServerSide.com，这种类发表了大量模式，最重要的见《EJB Design Patterns: Advanced Patterns, Processes and Idioms》。下面介绍这两类中的最重要模式。

Sun Java Center J2EE模式类别

Sun Java Center把模式组成多层平台中的层，与J2EE相似。前面曾介绍过，每一层有自己的责任，与其他层分开。核心J2EE模式主要针对Web组件与业务逻辑（EJB）层，其组织如下：

- 表示层模式，用于Web组件层
- 业务层模式，用于业务逻辑（EJB）层
- 集成层，连接EIS层的系统

这个类别定义下列表示层模式：

- **Intercepting Filter（截获过滤）**

  提供请求预处理和后处理的方案，定义灵活的体系结构，可以声明对截获请求和响应进行过滤。小服务过滤是这个模式的实现方法。

- **Front Controller（前端控制器）**

  通过中央控制器提供请求管理和处理。控制器取代通常发生在表示层的请求，从而取代模型视图控制器（MVC，Model View Controller）模式的控制器部分。前端控制器管理内容读取、安全性、视图管理和导航。

- **View Helper（视图帮助器）**

  将负责表示（格式化输出——视图）的编程逻辑与其他（业务）逻辑分开。表示格式放在视图组件中，可能包括多个子组件，组成复杂视图。业务逻辑代码放在帮助器组件中。帮助器组件的典型功能是内容读取、验证与适配。帮助器组件可以用Business Delegate模式访问业务服务。

- **Composite View（复合视图）**

  是从原子子组件创建累计表示（视图）的灵活方案。表示的体系结构可以方便地组织基本视图组件，使表示很灵活，还可以进行其他工作，包括个性化与定制。

- **Service-to-Worker（服务/工人）**

  由Dispatcher组件与Front Controller和View Helper模式组合而成，先进行请求处理再进行视图处理，适合大型应用程序，类似于Dispatcher View模式。

- **Dispatcher View（派遣视图）**

  类似于Service to Worker模式，也是由Dispatcher组件与Front Controller和View Helper模式组合而成。但与Service-to-Worker模式不同的是，这个模式在进行视图处理的期间进行请求处理，因此更适合小型应用程序。

这个类别定义下列业务层模式：

- **Business Delegate（业务代理）**

  减少层之间的耦合，特别是表示层与业务逻辑层之间。它提供门户代理，可以作为业务逻辑层和EIS层服务的通用入口。代理还可以缓存远程方法调用，从而提高性能。这个模式可以和Service Locator模式组合。

- **Value Object（数值对象）**

  解决层之间交换数据及相关远程开销的问题。这个模式收集系列化对象中需要的所有数据，用单个远程方法调用在层之间传输。这个模式通常是层之间交换数据和进行通信的最佳方法，可以成功地减少远程方法调用，这在分布式系统中成本很高。

- **Session Façade**

  隐藏业务组件的细节和集中工作流程，提供客户机的粗粒接口，减少远程方法调用开销，也适合声明式事务和安全管理。这个模式通常和其他模式组合，如Service Locator、Value Object、Value Object Assembler、Value List Handler、Service Activator与Data Access Object。这是J2EE模式中最重要的一个。

- **Composite Entity（复合实体）**

  解决EJB 1.1实体Bean的远程接口开销，提供设计粗粒实体Bean的方法，将相依赖对象合并为一个实体Bean，用于EJB 1.1持久性模型，在EJB 2.0持久性模型、本地接口和管理关系中已经过时。

- **Value Object Assembler（数值对象汇编器）**

可以灵活地复合不同来源的Value Object，包括EJB、数据访问对象或Java对象。这个模式创建要传递到客户机的数据。这个模式与Value Object模式相联系。

- **Value List Handler（数值清单处理器）**

  提供查询执行和结果处理的良好方案，还可以缓存结果，进一步提高性能。这个模式基于前面提到的四位专家类别中的Iterator模式。

- **Service Locator（服务定位器）**

  可以查找、创建与定位服务工厂，包装其细节。多个客户机可以使用这个对象，从而减少复杂性，提供单个控制点，通过缓存提高性能。

还有两个集成层模式：

- **Data Access Object（数据访问对象）**

  可以灵活而透明地访问数据，抽象数据源和隐藏EIS表示层细节。优点是业务与EIS层之间得到松耦合。这个模式也用于实体Bean的Bean管理持久性。

- **Service Activator（服务激活器）**

  可以异步处理同步EJB组件，特别是会话Bean。这个模式提供类似于EJB 2.0规范中消息驱动Bean的功能。这个模式的优点也适用于EJB 1.1。

有些Sun Java Center J2EE模式（特别是业务层模式）已经过时，是针对EJB 1.1定义的。众所周知，EJB 2.0已经作了几处重要修改，包括消息驱动Bean、本地接口和新的持久性模型。

这些新特性使Composite Entity模式已经无效，因为EJB 2.0中的本地接口可以建立细粒实体Bean，定义其间的关系时不会造成性能影响。实际上，EJB 2.0规范希望采用后一种方法。消息驱动Bean要求另一种模式Message Façade。由于消息驱动Bean提供的功能，也使Service Activator过期了。其中一些新模式已经在TheServerSide.com的第二类模式中定义。

TheServerSide.com模式类别

TheServerSide.com模式类别有所不同，分为下面几类：

- EJB Layer Architectural Patterns（EJB层体系结构模式）
- Inter-Tier Data-Transfer Patterns（层间数据传输模式）
- Transaction and Persistence Patterns（事务与持久性模式）
- Client-Side EJB Interaction Patterns（客户端EJB交互模式）

重要的EJB层体系结构模式包括：

- **Session Façade**

  提供与Sun Java Center J2EE模式类别中同名模式相同的解决方案。

- **Message Façade**

  与Session Façade模式相似，但用消息驱动Bean而不用会话Bean，因此可以异步容错访问。

- **EJB Command（EJB命令）**

  与Session Façade模式相对，将业务逻辑包装在命令Bean中，分离客户机与业务逻辑层，减少远程方法调用。

- **Data Transfer Object Factory（数据传输对象工厂）**
  提供数据传输对象的工厂。
- **Generic Attribute Access（一般性属性访问）**
  提供通过HashMaps以动态方式设置和读取实体Bean属性的方法。
- **Business Interface（业务接口）**
  提供实现EJB接口编译时一致性检查的方案。为此定义业务接口，在本地/远程接口和企业Bean类中实现。

重要的层间数据传输模式包括：

- **Data-Transfer Object（数据传输对象）**
  提供用系列化对象传输数据的方案。这个模式与Data Transfer Object Factory模式相联系，类似于Value Object模式。
- **Domain Data-Transfer Object（域数据传输对象）**
  是特定域的Data Transfer Object规范。
- **Custom Data-Transfer Objects（定制数据传输对象）**
  显示如何定制Data-Transfer Objects。
- **Data-Transfer HashMap（数据传输散列图）**
  提供通过HashMaps在客户机与EJB层之间调动任意数据集的方案。
- **Data-Transfer RowSet（数据传输行集）**
  显示如何用RowSets直接从EJB层的ResultSet向客户机层调动原始关系型数据。

重要的事务与持久性模式包括：

- **Version Number（版本号）**
  对跨事务的使用案例提供维护一致性和防止并发性问题的方案。
- **JDBC for Reading（读取JDBC）**
  显示如何用JDBC对关系型数据库进行列表操作。
- **Data-Access Command Beans（数据访问命令Bean）**
  用命令Bean进行数据访问，类似于EJB Command模式。
- **Dual Persistent Entity Bean（双持久实体Bean）**
  显示如何编写实体Bean，通过继承支持BMP与CMP。最后选择通过部署描述项完成。

重要的客户端EJB交互模式包括：

- **EJB Home Factory（EJB主工厂）**
  在查找EJB Home接口时改进性能，方法是使用EJB主工厂，通过实现缓存改进性能。
- **Business Delegate（业务代理）**
  类似于Sun Java Center模式类别中同名模式。

J2EE模式间的关系

Sun Java Center模式类别中的十五个模式与TheServerSide类别中的十七个模式是相联系的，有些模式是相似的，提供同一问题的相似解决方案。我们发现有下列相似性：

- Service to Worker与Dispatcher View

- Session Façade、Message Façade与EJB Command
- Service Locator与EJB Home Factory
- Value Object与Data-Transfer Object及其所有变形：Domain Data-Transfer Object、Custom Data-Transfer Object、Data-Transfer HashMap与Data-Transfer RowSet
- Value Object Assembler与Data-Transfer Object Factory
- Data-Access Object与Data-Access Command Bean

为了更好地了解这两个模式类别，我们用图1.4显示它们的关系和依赖性。灰色阴影框表示TheServerSide类别中的模式，而白框表示Sun Java Center模式类别中的模式。

图1.4

这些模式的组织、关系和依赖性如图1.5所示。

这两个类别中的几个J2EE模式对应于Java相关模式和通用设计模式，因此下节简要介绍其他最重要的设计模式。

其他最重要的设计模式

前面提到的四位专家的模式类别中发表了最流行和最有影响的通用设计模式，参见《Design Patterns: Elements of Reusable Object-Oriented Software by the GoF》一书，Addison-Wesley出版（ISBN: 0-201-63361-2）。其中包括下列设计模式。

- 创建性模式：
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- 结构性模式：
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- 行为性模式：
  - Chain of Responsibility

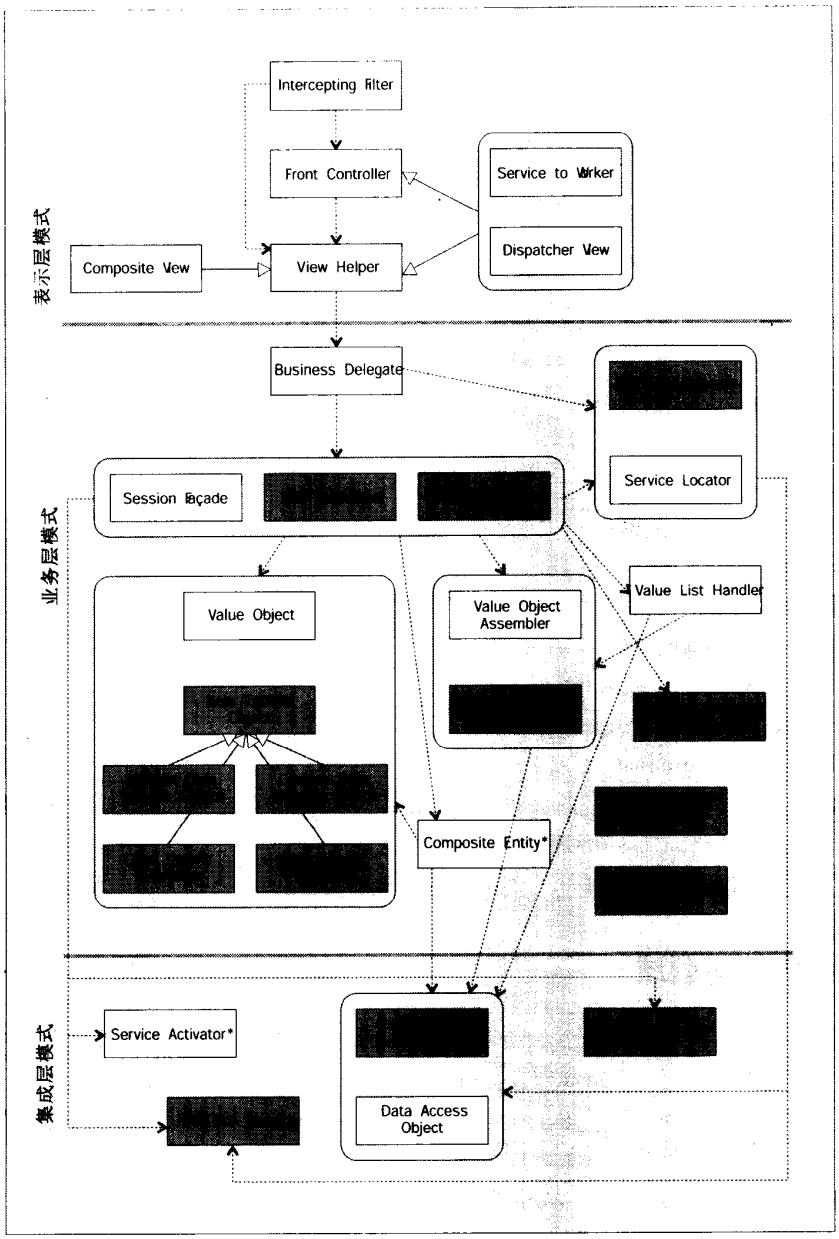

图1.5

- Command
- Interpreter
- Iterator
- Mediator
- Memento

- Observer
- State
- Strategy
- Template Method
- Visitor

除了前面提到的四位专家提出的类别之外，还有一些重要的Java相关类别，其中最有影响的是《Patterns in Java, Volume 1》一书，John Wiley & Sons出版（ISBN 0-471-25839-3），定义了下列重要模式：

- Delegation
- Interface
- Immutable
- Marker Interface
- Object Pool
- Layered Initialization
- Dynamic Linkage
- Cache Management
- Little Language
- Snapshot
- Single Threaded Execution
- Two-Phase Termination

由此可见，目前已经有许多重要模式。必须了解和领会这些重要模式之后才能正确选择和使用。本书介绍如何采用J2EE中的重要模式。

下节介绍J2EE模式的问题域及本书其余部分的组织。

J2EE模式的问题域

我们看到，不同模式类别由不同方式构造模式。本书不是介绍模式类别的，也不是正式介绍J2EE模式的，而是介绍如何在J2EE应用程序中采用现有设计模式。

为了简化模式选择，我们根据所用的问题域组织模式，对应于采用设计模式的方法。

本书涉及的模式及其应用组织成下面几部分：

- Patterns Applied to the Web Tier（Web层设计模式）
- Patterns Applied to a Persistence Framework （持久性框架设计模式）
- Patterns Applied to Improve Performance and Scalability（改进性能与伸缩性的设计模式）
- Patterns Applied to Manage Security（管理安全性的设计模式）
- Patterns Applied to Enable Enterprise Integration（企业集成设计模式）
- Patterns Applied to Enable Reusability, Maintainability, and Extendibility（复用性、维护性与扩展性设计模式）

“Web层设计模式”一章包括下列模式：

- Intercepting Filter（来自J2EE类别）
- Front Controller（来自J2EE类别）
- View Helper（来自J2EE类别）
- Composite View（来自J2EE类别）
- Service-to-Worker（来自J2EE类别）
- Dispatcher View（来自J2EE类别）

“持久性框架设计模式” 一章包括下列模式：

- Data Access Object(来自J2EE类别）
- Value Object(来自J2EE类别）
- Service Locator(来自J2EE类别）

“改进性能与伸缩性的设计模式” 一章包括下列模式：

- Value Object（来自J2EE类别）
- Value Object Assembler（来自J2EE类别）
- Business Delegate（来自J2EE类别）
- Session Façade（来自J2EE类别）
- Message Façade（来自The ServiceSide.com类别）
- Service Locator（来自J2EE类别）

“管理安全性的设计模式” 一章包括下列模式：

- Risk Assessment and Management
- Single Access Point
- Check Point
- Role

“企业集成设计模式” 一章包括下列模式：

- Integration Broker
- Wrapper
- Integration Mediator
- Virtual Component（Integration Façade）
- Data Access Object（来自J2EE类别）
- Data Mapping
- Process Automator

“复用性、可维护性与扩展性设计模式” 一章包括下列模式：

- Façade（来自四人专家组类别）
- Abstract Factory（来自四人专家组类别）
- Decorator（来自四人专家组类别）
- Template Method（来自四人专家组类别）
- Builder（来自四人专家组类别）

模式之间的关系

下一章要介绍来自四个源的模式，除了J2EE、TheServerSide.com和四人专家组模式之外，我们还要介绍集成模式。我们用图形显示它们之间的重要关系，不同模型组采用下列阴影，见图1.6所示。

图1.6

图1.7显示了模式之间的重要关系。

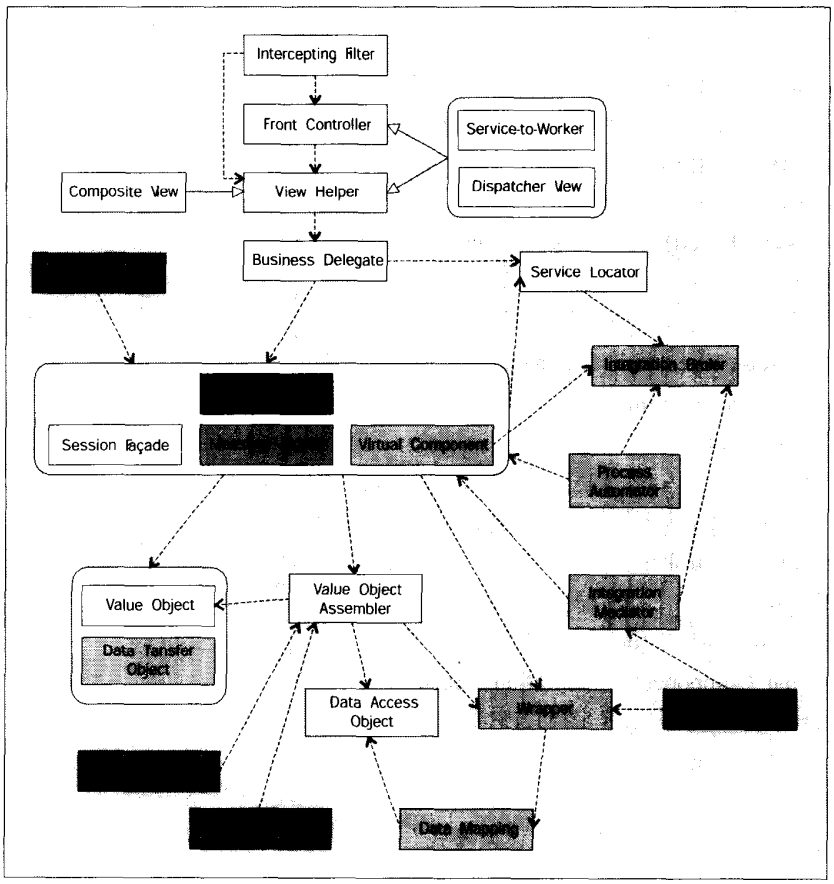

图1.7

小结

本章介绍了模式对达到高级复用很重要，可以复用思想。模式是一个情境中常见问题的最佳解决方案。尽管模式根源于建筑学和民用工程，但已经成为软件开发中的重要概念。模式可以得到常见问题的最佳解决方案。

模式可以用于软件开发中几乎所有阶段。也许最主要的是设计模式，是针对设计问题的。模式是找出来的，而不是凭空发明的。要找出模式，就需要大量成功项目和丰富的经验。找出模式是个困难而责任重大的工作，应能提供设计问题的最佳方案。因此，模式应尽量多多验证。

好在大部分人不需要找出模式，只要使用模式。要成功地使用模式，就应能选择最合适的模式。尽管如何选择最合适的模式有很多建议，但必须了解模式解决什么问题，采用何种解决方案，才能选择最合适的模式。了解模式之间的关系也很重要。

使用模式是设计软件的好办法，但必须明智地使用模式，即不能过份滥用，而应评估其利弊，否则可能使应用程序过量工程。另一方面，如果根本不用模式，则可能使应用程序不足量工程。这两种情况下可以通过因素改变改进设计。

读到模式时，就不能不介绍反模式。它们与模式相对，显示软件工程中的不良做法和不良解。了解反模式有助于避免这些常见错误。

设计模式可以是通用的，可以独立于特定编程语言、平台和域，也可以针对特定编程语言、平台和域。我们主要介绍J2EE设计模式。这些设计模式显示如何解决J2EE应用程序的常见设计问题。本章概述了最主要的J2EE模式和模式类别。

下面几章详细介绍如何在特定域中采用J2EE模式，介绍采用最主要的J2EE模式的具体例子和最佳做法。

第2章 Web层设计模式

设计Web应用程序的一个难点是创建结构合理的整洁的瘦客户机层，难点在于组合正确的外观和行为，同时包装表示逻辑，不把任何业务逻辑放到这一层中，并创建易于阅读和易于维护的代码。此外，表示层通常还有其他要求，如用户验证、数据加密、会话管理、用户个性化和请求处理。

本章介绍设计与管理Web应用程序顶层的问题，介绍客户层的技术需求和如何用J2EE表示设计模式解决。

我们开发一个宾馆订房管理系统的应用程序，讨论这些模式。这个应用程序主要考虑J2EE表示模式，但本身是个完整的可部署企业应用程序。

表示模式

模型视图控制器（MVC，Model View Controller）模式是第一个分开表示逻辑与业务逻辑的设计模式。MVC引入视图（表示层）、模型（数据）和协调两者的控制器。在出现MVC模式之前，用户界面设计通常把这些层合并在一起，而MVC模式则把它们分离开来，提高灵活性与复用性。

MVC定义“预订”或“通知”协议，分离视图与模型。视图要保证反映模型状态。模型数据发生改变时，所有视图得到通知，每个视图可以更新自己。这个方法可以对一个模型连接不同表示，可以创建新视图而不必改变模型或控制器。本章的“视图管理”一节将详细介绍这个问题。

随着Sun Java Center J2EE设计模式的引入，企业应用程序设计人员可以选择各种方案，解决多个问题，可以达到巧妙的解。

我们要介绍下列J2EE表示层模式：

- Intercepting Filter
- Front Controller
- View Helper
- Composite View
- Service-to-Worker
- Dispatcher View

还有下列支持模式：

- Model View Controller
- Observer
- Command
- Abstract Factory

案例：宾馆订房管理系统

这个应用程序主要考虑客户机管理方面，演示核心J2EE模式的用法。我们用这个例子描述方案中引入的模式。

首先要看看应用程序需求及其用例模型。

应用程序管理宾馆细节，基于某些条件搜索宾馆的功能，并提供一个订房系统，还包括其他宾馆信息，如名称、地址、类型、所订房间、宾馆细节变化情况，等等。它使宾馆集团或连锁店可以用一个远程系统管理各种订房需求。这个Web应用程序可以扩展成提供通用宾馆订房系统，使用户可以找到满足要求的宾馆，检查费用与是否有房以及订房。

宾馆系统应灵活，提供各种功能，这也取决于用户的角色。下面是要与宾馆系统交互的用户角色清单：

- Customers（客户）搜索宾馆以匹配指定条件，浏览所选宾馆的细节，检查费用与是否有房以及订房。
- 宾馆员工帮客户处理通过传真、E-mail或电话收到的订房请求。
- Administrators（管理员）可以作为Web客户或宾馆员工，还可以创建新宾馆项目、删除宾馆项目和管理信息。

这个应用程序的范围很大，但本例中只限于设计域和与管理员用例有关的特定区域的实现域。

用例模型

框图2.1显示了宾馆应用程序的用例模型，分为下列用例段。

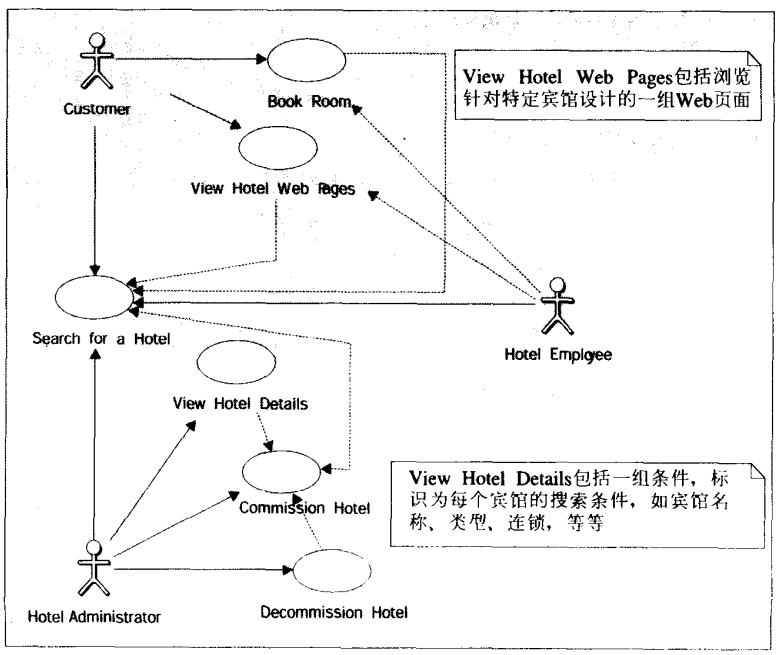

图2.1

宾馆管理员用例

本节介绍管理员角色交互的用例，包括：

- Commission Hotel（增加宾馆项目）
- Decommission Hotel（删除宾馆项目）
- View Hotel Details（浏览宾馆细节）

由于这些用例的格式非常相似，因此可以找出一个公用解。简单介绍每个用例之后，我们将介绍其模式标识与实现。

Commission Hotel用例

管理员可以创建新宾馆项目，包括选择条件。此外，还要在系统中部署一组描述新宾馆的Web页面（如图2.2所示）。

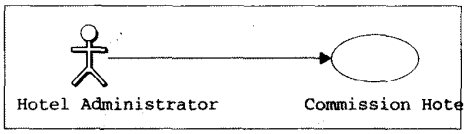

图2.2

求值用例的步骤如下：

- **Use Case Preconditions**
  提供了访问权限
- **On Successful Outcome of the Use Case**
  已经创建的新宾馆项目和增加新宾馆的Web页面
- **On Failed Outcome of the Use Case**
  未能创建新宾馆项目

Decommission Hotel用例

管理员可以从系统中删除宾馆项目，包括删除实体相关Web页面（如图2.3所示）。

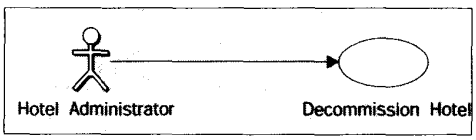

图2.3

求值用例的步骤如下：

- **Use Case Preconditions**
  系统中存在要删除的宾馆项目，包括所有相关Web页面
- **On Successful Outcome of the Use Case**
  顺利地从系统中删除宾馆项目，包括相关信息

- **On Failed Outcome of the Use Case**
  未能从系统中删除宾馆项目及相关信息

View Hotel Details用例

浏览宾馆细节包括浏览作为每个宾馆搜索条件的一组条件，如宾馆名称、类型、连锁，等等（如图2.4所示）。

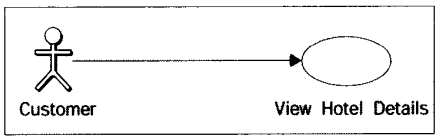

图2.4

- **Use Case Preconditions**
  系统中存在所选宾馆项目，包括所有相关Web页面
- **On Successful Outcome of the Use Case**
  顺利显示宾馆项目及所有相关信息
- **On Failed Outcome of the Use Case**
  未能显示宾馆项目及所有相关信息

标识模式

要确定完成上述用例所需的模式，就要解决与这种Web应用程序相关的域区域中的主要问题。我们标识了五大问题域，需要加以考虑，包括：

- 请求处理
- 会话管理
- 视图管理
- 验证
- 客户端安全

下面几节介绍前四个问题域，客户端安全域第5章介绍。在这些问题域情境中，我们要开发系统的Hotel Administrator部分。

请求处理

大多数软件应用程序都需要某种形式的请求处理，这是个非常有趣的应用程序设计问题，特别是在Web应用程序中。在简单情形中，请求处理可以是面向页面的，每一页管理自己的请求和响应处理。这时很可能产生大量重复代码并造成应用程序行为的不一致，使维护与扩展代码非常困难。

要解决请求处理问题，最好引入控制器。

Front Controller设计模式

Front Controller设计模式对常见请求处理工作采用集中控制，并委托给下一个视图。集中请求处理保证这个逻辑不会交织在不同视图中，在不同视图之间是相同的，使维护与扩展大为便利。Front Controller还促进表示逻辑与导航逻辑分开，这是非常有用的，因为可以改变导航逻辑而不影响表示逻辑。

控制器模式可以实现为小服务或JSP页面中嵌入的JavaBean。图2.5显示了Front Controller模式的结构。

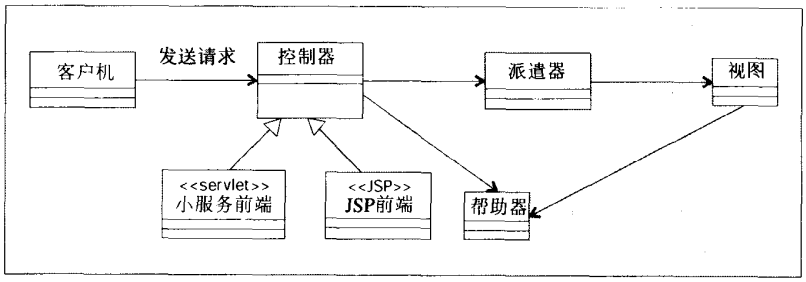

图2.5

Front Controller模式的参与者如下：

- **Controller（控制器）**
  处理请求和委托到下一视图的初始点
- **Dispatcher（派遣器）**
  负责视图管理与导航
- **View（视图）**
  向客户机表示和显示信息
- **Helper（帮助器）**
  帮助视图或控制器完成处理工作

在宾馆应用程序中，我们用小服务控制器实现Front Controller模式，使用命令与控制器策略。我们用小服务控制器而不用JSP实现方法的原因如下：

- JSP页面通常与显示格式有关，而不是与请求处理相关。用JSP实现控制器会混淆这两个不同目的。
- 在小服务控制器实现中，请求处理代码更容易维护，因为它与HTML分开。

这个策略可以将控制器委托任务分配到不同命令对象，进一步分离委托过程。Command设计模式将命令或操作表示为对象，将其行为分离，允许命令日志与撤销操作之类的其他可插入功能。

Front Controller实现

这个模式用AdminHotelController类实现。AdminHotelController扩展HttpServlet类，集中处理宾馆请求。这个类的源代码如下：

```
package hotel.presentationtier;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import javax.servlet.ServletConfig;
import javax.servlet.RequestDispatcher;
import java.io.IOException;

import hotel.util.HotelDetails;
import hotel.ejb.HotelHome;
import hotel.ejb.Hotel;
import java.math.BigDecimal;
import java.util.Properties;
import hotel.util.Command;
import hotel.util.RequestHelper;

public class AdminHotelController extends HttpServlet {

    private void processRequest(HttpServletRequest request,
                                HttpServletResponse response)
                throws ServletException, IOException {

      String next = "";
      try {
        RequestHelper helper = new HotelRequestHelper( request, response );
        Command command = helper.getCommand();
        next = command.execute( helper );
      } catch( Exception e ) {
        e.printStackTrace();
      }
      dispatch( request, response, next );
    }
```

这个方法处理HTTP GET请求：

```
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response )
              throws ServletException, IOException {
      processRequest( request, response );
    }
```

这个方法处理HTTP POST请求：

```
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response )
              throws ServletException, IOException {
      processRequest( request, response );
    }
```

```
    public String getServletInfo() {
      return getSignature();
    }
```

最后，这个方法将请求派遣到下一页：

```
    private void dispatch(HttpServletRequest request,
                          HttpServletResponse response,
                          String page )
                  throws ServletException, IOException {
      RequestDispatcher dispatcher=
         getServletContext().getRequestDispatcher( page );
      dispatcher.forward( request, response );
    }
```

注意页面是绝对路径，否则dispatcher为null：

```
    private String getSignature() {
      return "ServiceToWorker-Controller";
    }
}
```

Command设计模式

Command设计模式表示请求为一个对象，使客户机可以将请求参数化。由于每个命令表示为一个对象，因此很容易用新对象增加新命令的行为，而不会影响其他任何命令。另外，现有命令的行为可以修改，也可以添加新行为，而不会影响其他命令。必要时可以保存或登记命令状态。

类框图2.6显示了Command设计模式结构。

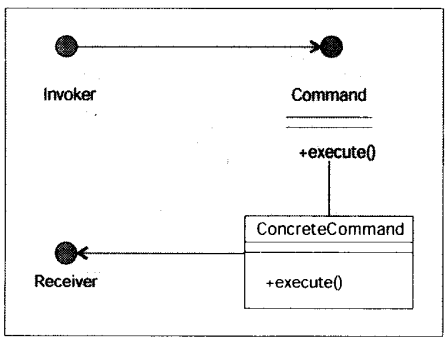

图2.6

Command设计模式的参与者如下：

- **Command（命令）**

  声明执行操作的接口

- **Concrete Command（具体命令）**
  定义接收对象与操作之间的关联
- **Invoker（调用者）**
  请求执行请求的命令
- **Receiver（接收者）**
  知道如何进行与执行请求相关的操作

框图2.7显示了Command模式的调用序列。

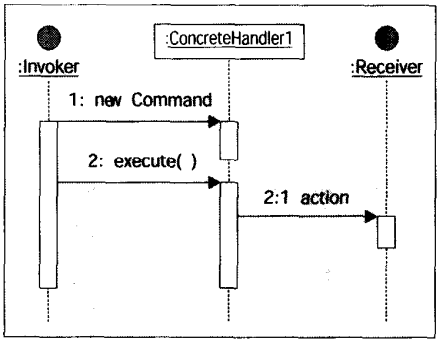

图2.7

Command实现

宾馆管理视图的命令类包装在框图2.8中。实际上有三种命令对象，包括CreateCommand、FindCommand与RemoveCommand。

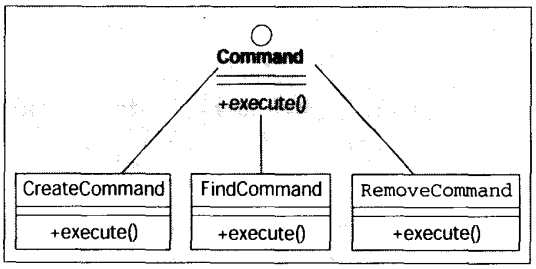

图2.8

每个对象有一个execute方法，映射该命令的操作。Command接口的源代码如下：

```
package hotel.util;

import javax.servlet.ServletException;
import java.io.IOException;
```

表示一个Command对象。根据Command设计模式，它将请求包装成对象，用不同请求将客户机参数化：

```
public interface Command {

   public String execute( RequestHelper helper )
                          throws ServletException, IOException;
}
```

这个接口提供了三个实现，分别是CreateHotelCommand、RemoveHotelCommand与FindHotelCommand。

以下类实现Create命令的Command接口：

```
package hotel.presentationtier;

import hotel.util.Command;
import hotel.util.HotelDetails;
import hotel.util.RequestHelper;

import hotel.businesstier.HotelBusinessDelegate;
import hotel.businesstier.HotelBusinessDelegateImpl;

import java.util.Properties;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;
import java.math.BigDecimal;

class CreateHotelCommand implements Command {
   private HotelDetails hoteldetails = null;
   private HotelBusinessDelegate delegate =
                          new HotelBusinessDelegateImpl();
   public CreateHotelCommand( ) {
   }

   private String getDisplayMessage( Properties properties ) {
     return properties.getProperty("id") + " is successfully created";
   }

   public synchronized String execute( RequestHelper helper )
     throws ServletException, IOException {
       try {
         Properties properties = helper.getProperties();
         hoteldetails = new HotelDetails(
                          properties.getProperty("name"),
                          properties.getProperty("purpose"),
                          properties.getProperty("regionOrTown"),
                          properties.getProperty("type"),
                          properties.getProperty("chain"),
         (BigDecimal)properties.get("swimmingPool"),
         (BigDecimal)properties.get("gym"),
         (BigDecimal)properties.get("conferenceRooms"));
```

```
            delegate.createHotel(properties.getProperty("id"), hoteldetails);
            helper.getRequest().setAttribute( "result",
                                    this.getDisplayMessage( properties ) );
            helper.getRequest().setAttribute( "hotel", hoteldetails );
          } catch( Exception e ) {
            e.printStackTrace();
            helper.getRequest().setAttribute( "result",
                                "Error processing Create Hotel Command"  +
                                e.toString() );
            return "/errorPage.jsp";
          }
          return "/adminHotelResult.jsp";
        }
      }
```

以下类实现remove命令的Command接口：

```
package hotel.presentationtier;

import hotel.util.Command;
import hotel.util.HotelDetails;
import hotel.util.RequestHelper;
import businesstier.HotelBusinessDelegate;
import businesstier.HotelBusinessDelegateImpl;
import java.util.Properties;
import java.io.IOException;
import javax.servlet.ServletException;

class RemoveHotelCommand implements Command {
   private HotelDetails hoteldetails;
   private HotelBusinessDelegate delegate =
                                    new HotelBusinessDelegateImpl();

   public RemoveHotelCommand( ) {
   }

   private String getDisplayMessage( String id ) {
     return "Hotel Id " + id + " is successfully removed";
   }

   public synchronized String execute( RequestHelper helper )
      throws ServletException, IOException {
      try {
         String id = helper.getProperties().getProperty("id");
         hoteldetails = delegate.getHotel( id );
         delegate.removeHotel( id );
         helper.getRequest().setAttribute( "result",
                                        this.getDisplayMessage( id ) );
```

```
            helper.getRequest().setAttribute( "hotel", hoteldetails );
        } catch( Exception e ) {
            e.printStackTrace();
            helper.getRequest().setAttribute("result",
                                "Error processing Remove Hotel Command" +
                                 e.toString() );
            return "/errorPage.jsp";
          }
        return "/adminHotelResult.jsp";
    }
}
```

以下类实现**find**命令的**Command**接口：

```
package hotel.presentationtier;

import hotel.util.Command;
import hotel.util.HotelDetails;
import hotel.util.RequestHelper;
import hotel.businesstier.HotelBusinessDelegate;
import hotel.businesstier.HotelBusinessDelegateImpl;
import java.util.Properties;
import java.io.IOException;

import javax.servlet.ServletException;
   class FindHotelCommand implements Command {
      private HotelDetails hoteldetails;
      private HotelBusinessDelegate delegate =
                        new HotelBusinessDelegateImpl();
      public FindHotelCommand( ) {
      }

      private String getDisplayMessage( Properties properties ) {
        return properties.getProperty("id") + " is found";
      }

      public synchronized String execute( RequestHelper helper )
                                throws ServletException, IOException {
        try {
          Properties properties = helper.getProperties();
          HotelDetails hoteldetails =delegate.getHotel(
                                      properties.getProperty("id") );
                                      helper.getRequest().
            setAttribute("result", this.getDisplayMessage( properties ) );
            helper.getRequest().setAttribute( "hotel", hoteldetails );
        } catch( Exception e ) {
          e.printStackTrace();
```

```
            helper.getRequest().setAttribute(
                        "result",
                        "Error processing Find Hotel Command"  +
                         e.toString() );
            return "/errorPage.jsp";
          }
          return "/adminHotelResult.jsp";
        }
      }
```

注意这三个命令对象都能访问包，因为Command对象的客户机不需要知道每个对象的特定具体类。所有命令类型可以用Command接口统一处理。

下节介绍如何用抽象工厂类CommandFactory创建和定义这些Command对象。

抽象工厂设计模式

由于有多种命令类型，因此要用整洁的方式建立这些命令，将其交给控制器，使控制器能够委托。为此，我们使用Abstract Factory设计模式。Abstract Factory可以创建相关对象系列而不暴露其具体类。

图2.9显示了Abstract Factory设计模式的结构。

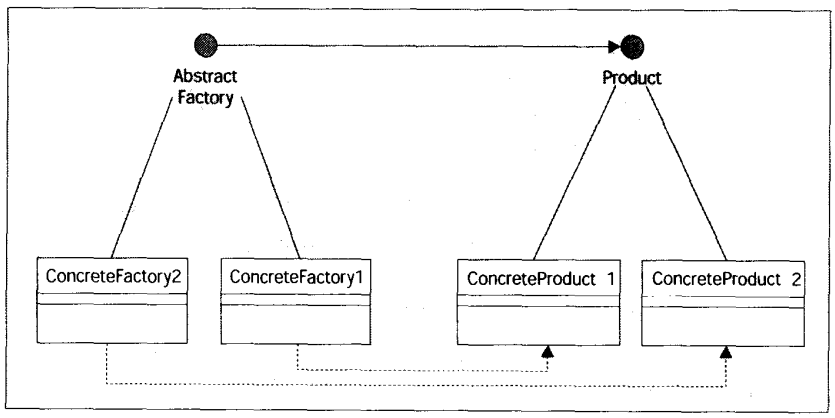

图2.9

Abstract Factory设计模式的参与者如下：

- **Abstract Factory（抽象工厂）**
  提供创建产品的接口抽象
- **Product（产品）**
  提供产品抽象的接口
- **Concrete Factory（具体工厂）**
  创建具体产品的抽象工厂实现方法

- **Concrete Product（具体产品）**
  具体产品实现方法

Abstract Factory实现

Abstract Factory设计模式实现为CommandFactory接口和CommandFactoryImpl类，实现这个接口。CommandFactory接口的createCommand方法可以创建命令。下面是Command-Factory接口的源代码：

```
package hotel.util;
```

表示create命令的抽象工厂接口。Abstract Factory提供的接口可以创建相关或依赖对象系列而不必指定具体类：

```
public interface CommandFactory {

    public Command createCommand( String action );
}
```

这是Command Factory设计模式的实现方法：

```
package hotel.presentationtier;

import hotel.util.Command;
import hotel.util.CommandFactory;

import java.util.HashMap;
import java.util.Enumeration;
import java.util.Properties;

class CommandFactoryImpl implements CommandFactory {

    private HashMap commands = new HashMap();
    private final static String
         ACTION_MAPPING_PROPERTIES = "actionMapping.properties";

    public CommandFactoryImpl() {
      try {
        Properties properties = new Properties();
        properties.load( getClass().getResourceAsStream(
        ACTION_MAPPING_PROPERTIES ));
        for( Enumeration e = properties.keys(); e.hasMoreElements(); ) {
           String action = ( String )e.nextElement();
           commands.put( action, ObjectCreator.createObject(
                        properties.getProperty( action ) ) );
        }
      } catch( Exception e ) {
        System.out.println( "Error: "  + e.toString() );
        e.printStackTrace();
      }
    }
```

```
    public Command createCommand( String action ) {
      return ( Command )commands.get( action );
    }
}
```

这个类装入属性文件actionMapping.properties中的操作和类名映射。映射如下：

```
create= hotel.presentationtier.CreateHotelCommand
remove= hotel.presentationtier.RemoveHotelCommand
find= hotel.presentationtier.FindHotelCommand
```

FactoryFacadeImpl用ObjectCreator类动态实例化与每个映射相关联的命令对象。本书不准备介绍这个类，源代码可以从http://www.wrox.com下载。

Command设计模式和Abstract Factory设计模式还不足以帮助控制器完成初始请求处理工作，还需要View Helper设计模式。

View Helper设计模式

View Helper设计模式帮助视图完成不与视图表示直接相关联的任务。使用View Helper的主要目的是把与业务层的交互和表示层分开，从而尽量分离这两个层，使一个层中的任何改变不会对另一层造成太大影响。

视图结构很简单，View Helper设计模式如框图2.10所示。

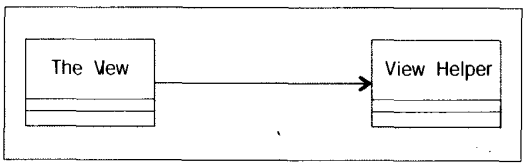

图2.10

Composite View Helper设计模式的参与者如下：

- **The View（视图）**

  是Web应用程序情境中的表示视图，可以是HTML、JSP或小服务。

- **The View Helper（视图帮助器）**

  是个视图帮助器，保存与表示格式没有直接关联的信息。视图帮助器可以实现以下内容：

  - Java bean实现
  - 定制标志实现

数值Bean帮助器比JavaBean帮助器更具体，通常用作JSP中的数据保持者。下面是数值Bean帮助器的JSP标志：

```
<jsp:useBean id="hotel" scope="request"
             class="hotel.util.HotelDetails" />
```

下面是个简单的定制标志帮助器：

```
<logic:iterate collection="<%=cdDB.getCds()%>"
               id="cd" type="database.CdDetails">
```

View Helper实现

宾馆管理视图的帮助器类包装在三个接口与类中，分别是RequestHelper接口、AbstractRequestHelper类和HotelRequestHelper类。RequestHelper对象收集特定参数信息。

图2.11显示了这些接口与类的相互联系。

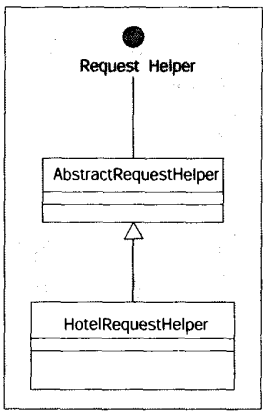

图2.11

RequestHelper接口的源代码如下：

```
package hotel.util;

import java.util.Properties;
import java.math.BigDecimal;
import java.util.Enumeration;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public interface RequestHelper {
    public HttpServletRequest getRequest();
    public HttpServletResponse getResponse();
    public Command getCommand();
    public Properties getProperties();
}
```

AbstractRequestHelper类的源代码如下：

```
package hotel.presentationtier;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import javax.servlet.ServletException;
import java.io.IOException;

import java.util.Properties;
import java.math.BigDecimal;
import java.util.Enumeration;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import hotel.util.RequestHelper;
import hotel.util.Command;

abstract class AbstractRequestHelper implements RequestHelper {

    private HttpServletRequest request = null;
    private HttpServletResponse response = null;
    private Properties properties = null;

    AbstractRequestHelper( HttpServletRequest request,
                           HttpServletResponse response){
      this.request = request;
      this.response = response;
      this.properties = getProperties( request );
    }

    public HttpServletRequest getRequest() {
      return request;
    }

    public HttpServletResponse getResponse() {
      return response;
    }

    public abstract Command getCommand();

    public Properties getProperties( ) {
      return properties;
    }
```

除了直接转换之外，这个方法将用户输入文本分析并过滤成实际Java对象和数值。例如，字符串“1”分析为数字：

```
private Properties getProperties(HttpServletRequest httpservletrequest) {
  Properties properties = new Properties();
  for( Enumeration enumeration =
              httpservletrequest.getParameterNames();
              enumeration.hasMoreElements();) {
    Object obj = enumeration.nextElement();
    String s = httpservletrequest.getParameterValues((String)obj)[0];

    System.out.println( "Parameter name ="+ obj.toString() + ", Parameter
```

```
                                value =" + s );
            if( ! isAny(s)) {
              try {
                properties.put(obj, new BigDecimal(s));
              } catch(NumberFormatException _ex) {
                properties.put(obj, s);
              }
            }
          }
          return properties;
        }

        private static boolean isAny(String s) {
            return s.equals("Any");
        }
    }
```

最后，下面是**HotelRequestHelper**类代码。这个类是最后一个类，只能访问包，因为表示层包域之外不能扩展和访问：

```
package hotel.presentationtier;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import hotel.util.CommandFactory;
import hotel.util.Command;
import hotel.util.RequestHelper;

import javax.servlet.ServletException;
import java.io.IOException;

final class HotelRequestHelper extends AbstractRequestHelper {

    private CommandFactory commandFactory = new CommandFactoryImpl();

    HotelRequestHelper( HttpServletRequest request,
                        HttpServletResponse response) {
      super( request, response );
    }

   public Command getCommand() {
      return commandFactory.createCommand(
        getProperties().getProperty("action") );
    }
```

会话管理

客户机会话可以定义成维护几个或一组客户机请求与一个服务器之间的会话状态。大多数情况下，需要管理客户机会话。会话管理分为客户机层和服务器层。

客户机会话可以保持在客户端代码中，使用HTML、cookies、URL或服务器代码中的隐藏字段。有些应用程序还使用数据库持久性或系列化。

有时还要先验证用户会话之后再处理请求。通常，用户登录时，服务器方实用程序产生惟一会话号，每个请求的惟一会话号在用户顺利登录之后转发，用于处理。如果用户退出或经过一定时间，则会话到期。

如果不加注意，则这种设计可能使客户机会话分段并且很难维护。可以用清晰的结构或设计模式解决这些问题，一种方法是使用上节介绍的Front Controller设计模式验证用户会话，减少分段。

图2.12显示了这个Front Controller设计模式，引入会话帮助器，可以创建新会话和验证现有会话。

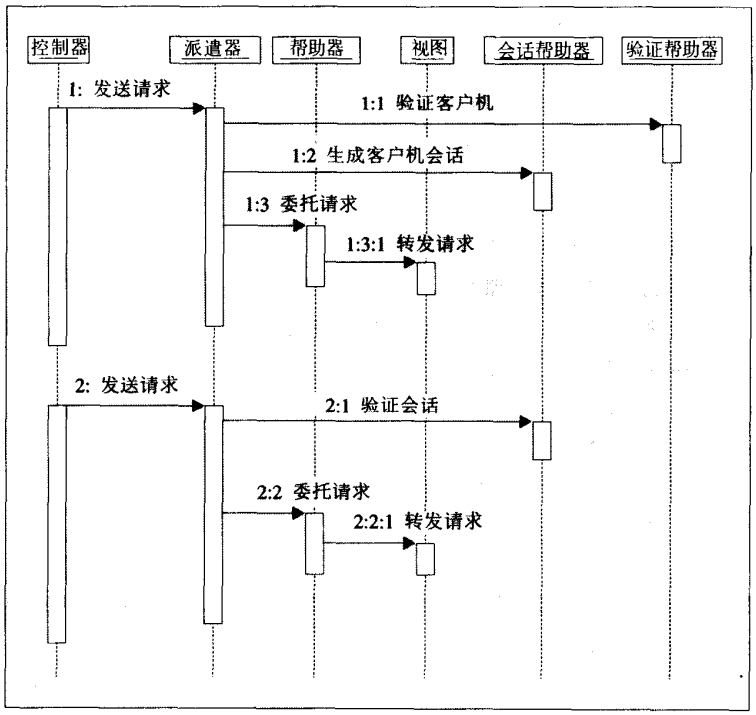

图2.12

另一种保证会话创建和验证的方法是使用Intercepting Filter设计模式。这是宾馆应用程序采用的方法，目的是在进入控制器代码之前先创建好会话和验证逻辑。如果请求会话无效，则退回请求。

Intercepting Filter设计模式

Intercepting Filter设计模式可以截获请求和对其采用一组过滤，然后退回请求或让其传递到所要目标。

Intercepting Filter设计模式的结构如图2.13所示。

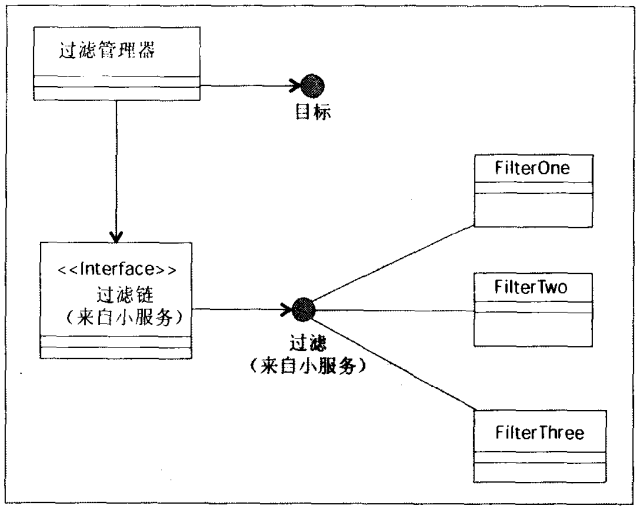

图2.13

Intercepting Filter设计模式的参与者如下：

- **FilterManager（过滤管理器）**
  过滤管理器管理过滤、排序和启动过滤链中的处理
- **FilterChain（过滤链）**
  过滤链是由各个过滤构成的结构
- **FilterOne、FilterTwo、FilterThree**
  各个过滤，可以和特定目标相关联
- **Target（目标）**
  客户机请求的目标

图2.14显示了宾馆应用程序采用的不同过滤类型。下节要介绍SessionFilter实现，本章验证部分要介绍ValidationFilter实现。

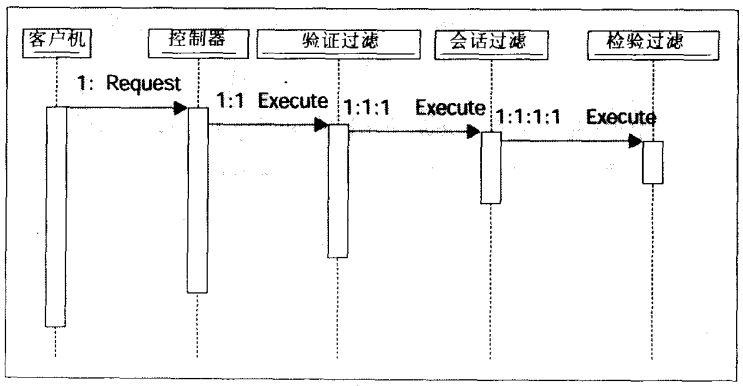

图2.14

Intercepting Filter实现

Servlet 2.3 API提供了现成的Intercepting Filter设计模式实现，我们实现javax.servlet.Filter以创建SessionFilter：

```
package hotel.presentationtier;

import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.FilterConfig;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.Filter;

import java.io.IOException;

public class SessionFilter implements Filter {

  private FilterConfig filterConfig = null;

  public void init( FilterConfig filterConfig ) throws ServletException {
    this.filterConfig = filterConfig;
  }

  public FilterConfig getFilterConfig() {
    return filterConfig;
  }

  public void setFilterConfig( FilterConfig filterConfig ) {
    this.filterConfig = filterConfig;
  }

  public void doFilter( ServletRequest request,
                        ServletResponse response,
                        FilterChain chain ) throws IOException,
                        ServletException {
    HttpServletRequest httpRequest = ( HttpServletRequest )request;
    HttpServletResponse httpResponse = ( HttpServletResponse )response;
    String session = ( String )httpRequest.getAttribute( "session" );
    if( session == null ) {
      httpRequest.setAttribute( "result",
                                "session is not valid" );
      RequestDispatcher dispatcher = filterConfig.getServletContext()
                                .getRequestDispatcher("/errorPage.jsp");
      dispatcher.forward( request, response );
    } else {
      chain.doFilter( request, response );
```

```
        }
    }

    public void destroy() {
    }
}
```

本书中这个过滤进行会话验证，而不创建新会话。过滤要在应用程序的web.xml描述项中声明。这是声明会话过滤的XML：

```
<filter>
   <filter-name>SessionFilter</filter-name>
   <filter-class>hotel.presentationtier.SessionFilter
   </filter-class>
</filter>

<filter-mapping>
  <filter-name>SessionFilter</filter-name>
  <url-pattern>/AdminHotelServlet/*</url-pattern>
</filter-mapping>
```

视图管理

宾馆应用程序采用的视图有两种：客户与员工共享的视图和管理视图。产品寿命周期的后续阶段可能需要更多视图，因此可以在设计时建立灵活性。这就要求提供方便和整洁的视图管理方法。

要解决这个问题，最好采用MVC设计模式，如图2.15所示。MVC将视图与模型分开。一个元素只能在一个模型中出现一次，但可以在不同视图中多次出现。可以根据用户要求调整视图，例如可以建立模型子集。

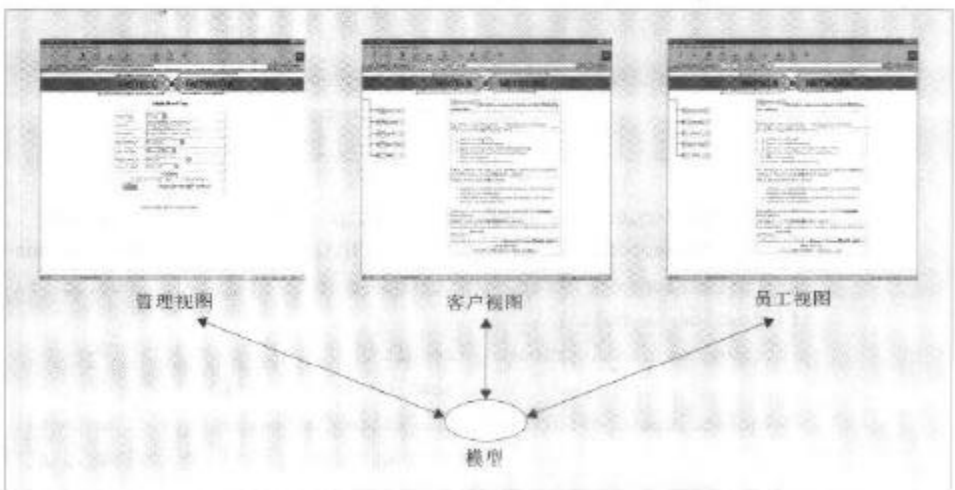

图2.15

通过采用前面介绍的Front Controller设计模式，我们对宾馆应用程序使用Model View Controller结构。Front Controller设计模式引入视图和控制器元素，模型在表示层之外。

除此之外，有些用户可能有权修改模型，这就需要更新机制，可以用Observer设计模式实现。

Observer定义对象之间的一对多依赖性，一个对象改变状态时，自动通知和更新依赖者。Web应用程序中更新视图并不容易，需要进行多次刷新、轮询或保持HTTP连接打开。

在宾馆例子中，不需要更新机制，因为这不是业务要求内容。宾馆可以在24小时内签约或撤约，订房时可能更频繁，但本例不准备介绍。

管理视图需要某种结构，因为需要框视图和子视图。例如，在宾馆用户视图中，我们要在页面开头显示一个标题，输入一组条件，在另一区中显示用户操作结果。为此可以使用Composite View设计模式。

Composite View设计模式

Composite View设计模式可以用父视图累计子视图，使总体视图成为各个小图的合成视图。这样可以减少不同视图重复部分重复代码，保证不同视图之间的一致性，使视图维护与管理更简单。

图2.16显示Composite View设计模式的基本结构。Composite View设计模式所基于的Composite模式最初在《Design Patterns, Elements of reusable object-oriented software》一书（ISBN 02-0163-361-2）中引入。

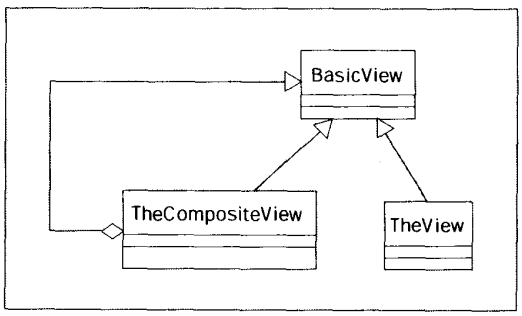

图2.16

Composite View设计模式的参与者如下：

- **BasicView（基本视图）**
  视图的基本抽象
- **TheCompositeView（复合视图）**
  由多个子视图构成的视图
- **TheView（视图）**
  没有子视图的简单视图

Composite View实现

在宾馆例子中，有四个子视图，可以在一个复合视图中使用，这是管理宾馆的视图：

- 管理宾馆帧视图
- 管理宾馆视图
- 管理宾馆结果视图
- 错误页面视图
- 标题视图

图2.17显示了视图复合。

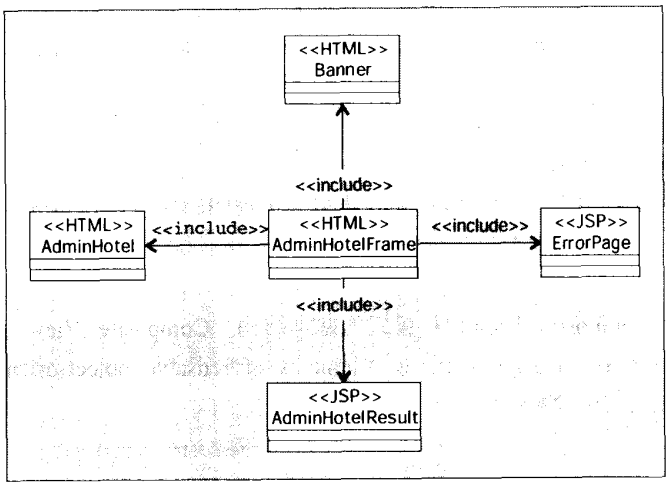

图2.17

对于管理视图页面，我们使用的复合视图策略混合简单HTML帧集策略和JavaBean视图策略。对于JavaBean视图策略，是在JSP页面情境中使用的。JavaBean很容易在JSP页面结构中创建与集成。

AdminHotel.html页面的源代码如下：

```
<html>
  <head>
    <title>Admin Hotel Page</title>
  </head>

<body>
  <h3><center>Admin Hotel Page</center></h3>
  <form action="AdminHotelServlet" method="GET" target="result">
   <p> </p>
    <div align="center"><center><table border="1" cellspacing="1"
        width="425" bgcolor="#FFFFFF" bordercolor="#FFFFFF"
        bordercolorlight="#FFFFFF" bordercolordark="#FFFFFF">
  <tr>
```

```
  <td width="125"><font size="2" face="Arial">Action Type</font></td>
  <td width="300"><select name="action" size="1">
  <option value="create">Create</option>
     <option value="find">Find</option>
     <option value="remove">Remove</option>
     </select></td>
</tr>
<tr>
   <td width="125"><font size="2" face="Arial">Hotel ID</font></td>
   <td width="300"><div align="left"><p>
      <input type="text" name="id" value="type hotel ID here" size="20">
   </td>
</tr>

<tr>
  <td width="125"><font size="2" face="Arial">Hotel Name</font></td>
  <td width="300"><div align="left"><p>
  <input type="text" name="name" value="type hotel name here" size="20">
  </td>
</tr>

<tr>
  <td width="125"><font size="2" face="Arial">Hotel Purpose</font></td>
  <td width="300"><select name="purpose" size="1">
     <option value="Business">Business</option>
     <option value="Cultural excursion">Cultural excursion</option>
     <option value="Skiing Holiday">Skiing Holiday</option>
     <option value="Beach Holiday">Beach Holiday</option>
     <option value="Business & Culture">Business & Culture</option>
     <option value="Beach & Culture">Beach & Culture</option>
     <option value="Business & Beach">Business & Beach</option>
   </select>
  </td>
</tr>

<tr>
  <td width="125"><font size="2" face="Arial">Type of Hotel</font></td>
  <td width="300"><div align="left"><p><select name="type" size="1">
     <option value="5 Star">Delux / 5 Star</option>
     <option value="4 Star">4 Star</option>
     <option value="3 Star">3 Star</option>
     </select>
  </td>
</tr>

<tr>
  <td width="125"><font size="2" face="Arial">Region or Town</font></td>
```

```
    <td width="300"><select name="regionOrTown" size="1">
       <option value="Ankara">Ankara</option>
       <option value="Istanbul">Istanbul</option>
       <option value="Cappadocia">Cappadocia</option>
       <option value="Bodrum">Bodrum</option>
       <option value="Erciyes">Erciyes (ski centre)</option>
       <option value="Side">Side</option>
       <option value="Palandoken">Palandoken (ski centre)</option>
       </select>
    </td>
  </tr>

  <tr>
    <td width="125"><font size="2" face="Arial">Chain or Agent</font></td>
    <td width="300"><select name="chain" size="1">
       <option value="Gold Trail">Gold Trail</option>
       <option value="Thomson">Thomson</option>
       <option value="LunnPoly">LunnPoly</option>
       <option value="First Choice">First Choice</option>
       <option value="Jmc">Jmc</option>
       <option value="Ottoman Hotels">Ottoman Hotels</option>
       </select>
    </td>
  </tr>

  <tr>
    <td width="425" align="center" colspan="2"><small><font
        face="Arial"><strong>Facilities:</strong><br>
      <input type="checkbox" name="swimmingPool" Value="1">
         Swimming Pool</font>,
      <input type="checkbox" name="gym" Value="1"><font
         face="Arial">Gym</font>,
      <input type="checkbox"
            name="conferenceRooms" Value="1"><font face="Arial">Conference
            Rooms.</font></small></td>
  </tr>

    <tr>
      <td width="125" align="center"><input type="submit"
          value="Submit"></td>
      <td width="300" align="center"<small><font
          face="Arial">
        <strong>Hotel servlet GET method:</strong></font></td>
    </tr>
  </table>
</center></div>
```

```
    </form>

    </body>
  </html>
```

注意高亮显示的操作标志指向AdminHotelServlet，这个名称映射hotel.war文件在weblogic web.xml描述项中的hotel.presentationtier.AdminServletController类。

最后，我们重新看看AdminHotelResult JSP页面。这个JSP页面利用数值Bean帮助器，是我们对Admin Hotel View开发的第二个帮助器：

```
<html>
  <head>
    <title>Admin Hotel Result Page</title>
  </head>

  <body link="#FFFFFF" vlink="#FFFFFF" alink="#FFFFFF" bgcolor="#FFFFFF"
       text="#000000">
    <H3><center> "<%=request.getAttribute( "result" )%>"</center></H3>
    <p>

   <jsp:useBean id="hotel" scope="request"
                class="hotel.util.HotelDetails" />
    <center>
    <h2>
        Hotel Details for
      <jsp:getProperty name="hotel" property="name"/>
    </h2>

     <table border=3>
        <tr>
          <td>
            Hotel Purpose :
        </td>

     <td>
        <jsp:getProperty name="hotel" property="purpose" />
     </td>
        </tr>

        <tr>
          <td>
            Hotel Region or Town :
          </td>
          <td>
            <jsp:getProperty name="hotel" property="regionOrTown" />
          </td>
        </tr>

        <tr>
```

```
            <td>
              Hotel Type :
            </td>
            <td>
              <jsp:getProperty name="hotel" property="type" />
            </td>
          </tr>

          <tr>
            <td>
              Hotel Chain :
            </td>
            <td>
              <jsp:getProperty name="hotel" property="chain" />
            </td>
          </tr>
        </table>
      </center>
    </body>
  </html>
```

View Helper实现

本章前面开发RequestHelper类时引入了View Helper设计模式。现在又要用到这个设计模式了。我们开发JavaBean帮助器HotelDetails，作为第二个帮助器，存储所读取宾馆的宾馆信息，以便向用户显示其细节。

HotelDetails类的代码如下：

```
package hotel.util;

import java.io.Serializable;
import java.math.BigDecimal;

public class HotelDetails implements Serializable {

    private String name = "";
    private String purpose = "";
    private String regionOrTown = "";
    private String type = "";
    private String chain = "";
    private BigDecimal swimmingPool = new BigDecimal(0.0D);
    private BigDecimal gym = new BigDecimal(0.0D);
    private BigDecimal conferenceRooms = new BigDecimal(0.0D);

    public HotelDetails( String name,
                         String purpose,
                         String regionOrTown,
                         String type,
```

```
                    String chain,
                    BigDecimal swimmingPool,
                    BigDecimal gym, BigDecimal conferenceRooms ) {
    if( name != null ) {
      this.name = name;
    }
    if ( purpose != null ) {
      this.purpose = purpose;
    }
    if ( regionOrTown != null ) {
      this.regionOrTown = regionOrTown;
    }
    if ( type != null ) {
      this.type = type;
    }
    if ( chain != null ) {
      this.chain = chain;
    }
    if ( swimmingPool != null ) {
      this.swimmingPool = swimmingPool;
    }
    if ( gym != null ) {
      this.gym = gym;
    }
    if ( conferenceRooms != null ) {
      this.conferenceRooms = conferenceRooms;
    }
  }

  public String getName() {
    return name;
  }

  public String getPurpose() {
    return purpose;
  }

  public String getRegionOrTown() {
    return regionOrTown;
  }

  public String getType() {
    return type;
  }

  public String getChain() {
    return chain;
  }
```

```
    public BigDecimal getSwimmingPool() {
      return swimmingPool;
    }

    public BigDecimal getGym() {
      return gym;
    }

    public BigDecimal getConferenceRooms() {
      return conferenceRooms;
    }

    public String toString() {
      StringBuffer stringbuffer = new StringBuffer(getClass().getName() +
                                                                    "::\n");
      stringbuffer.append("name=" + name + ", purpose=" + purpose + ",
                          region or town=" + regionOrTown + ", ");
      stringbuffer.append("type=" + type + ", chain=" + chain + ", ");
      stringbuffer.append("swimmingPool=" + swimmingPool + ", gym=" + gym +
                          ", ");
      stringbuffer.append("conferenceRooms=" + conferenceRooms);
      return stringbuffer.toString();
    }
}
```

图2.18显示了“视图管理”一节介绍的三个设计模式组合到一个框架中，这是前面介绍的三个设计模式。

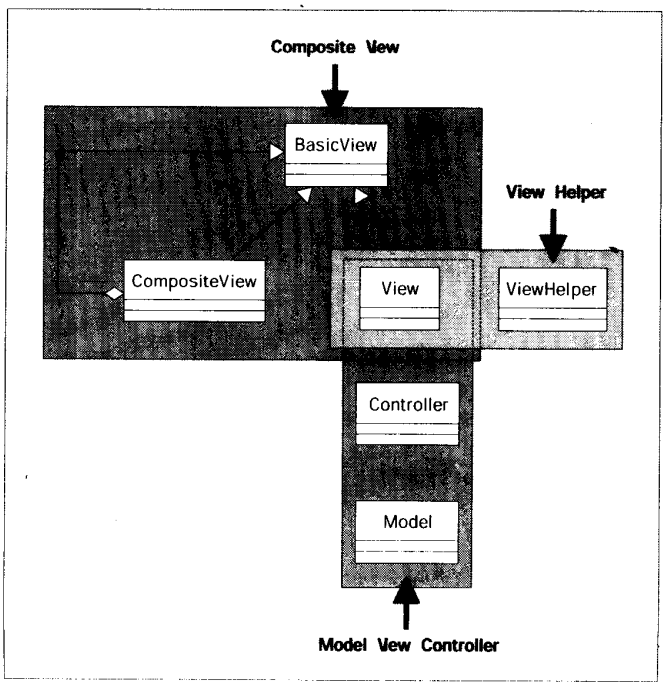

图2.18

- **Model View Controller（模型视图控制器）**
  将表示逻辑与业务逻辑分解到不同层，引入模型（业务数据）、视图（数据表示）和协调这两者的控制器。
- **Composite View（复合视图）**
  Composite View设计模式可以用父视图累计子视图，使总体视图成为各个小图的合成视图。
- **View Helper（视图帮助器）**
  View Helper设计模式帮助视图完成与视图表示不直接相关联的任务。

图2.18中，“Composite View”、“View Helper”与“Model View Controller”框表示各个设计模式，而“BasicView”、“CompositeView”、“View”、“ViewHelpe”、“Controller”与“Model”框表示模式参与者。

检验

表示层有两种检验，即客户端检验和服务器检验。客户端检验位于前端HTML层，可以用JavaScript之类脚本语言实现。一定要在进行类型检验时过滤空字段，例如区别字符串、数字与日期。服务器检验位于JSP页面与小服务代码之间的层，可以用视图帮助器类和过滤实现。

客户端检验和服务器检验都是有效检验输入所必须的，其他策略可能是使用数值Bean或把检验移到业务逻辑层。

Intercepting Filter实现

前面已经介绍过Intercepting Filter设计模式，可以看到这个设计模式也可以检验用户输入。这里我们用ValidationFilter实现javax.servlet.Filter接口，用另一个Validator类和工厂创建Validators。

图2.19显示在Intercepting Filter设计模式中用检验过滤，对宾馆管理页面的小服务请求进行检验。

ValidationFilter类实现doFilter方法，进行检验：

```
package hotel.presentationtier;

import hotel.util.Validator;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.FilterConfig;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.Filter;
```

```
import java.io.IOException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ValidationFilter implements Filter {

  private FilterConfig filterConfig = null;
  private Validator validator = new
                                ValidatorFactoryImpl().createValidator();
  public void init( FilterConfig filterConfig ) throws ServletException {
    this.filterConfig = filterConfig;
  }

  public FilterConfig getFilterConfig() {
    return filterConfig;
  }

  public void setFilterConfig( FilterConfig filterConfig ) {
    this.filterConfig = filterConfig;
  }

  public void doFilter( ServletRequest request,
                        ServletResponse response,
                        FilterChain chain )
          throws IOException, ServletException {
    HttpServletRequest httpRequest = ( HttpServletRequest )request;
    HttpServletResponse httpResponse = ( HttpServletResponse )response;
    if( ! validator.validate( request ) ) {
      httpRequest.setAttribute( "result",
             "Error data entered is not valid. Please check your input" );
             RequestDispatcher dispatcher =
             filterConfig.getServletContext()
            .getRequestDispatcher("/errorPage.jsp");
      dispatcher.forward( request, response );
    } else {
      chain.doFilter( request, response );
    }
  }

  public void destroy() {
  }
}
```

这是Validator对象的接口。Validator接口有个validate方法，返回true或false和取一个参数。这个参数抽象为Object类型，可以在任何情境中复用：

```
package hotel.util;

public interface Validator {
```

```
    public boolean validate( Object object );
}
```

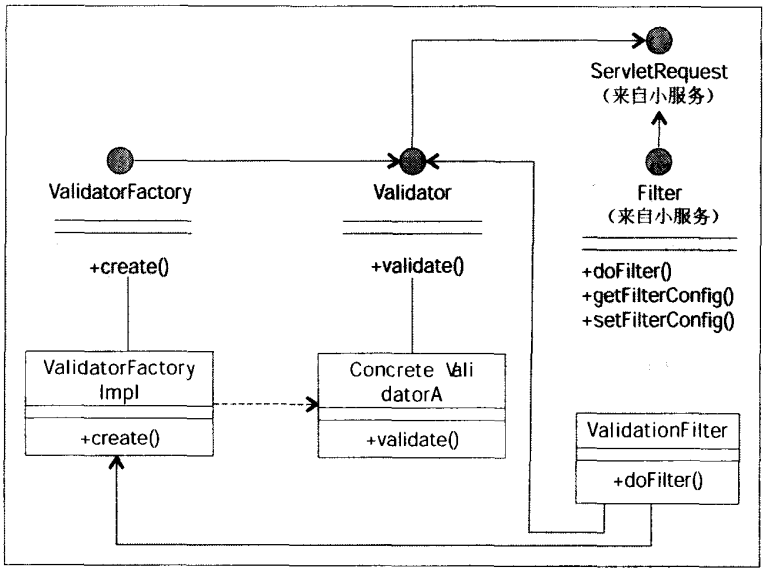

图2.19

要建立检验对象，就要再次采用抽象工厂，以便不指定具体类而创建Validator实现。ValidatorFactory接口的源代码如下：

```
package hotel.util;

public interface ValidatorFactory {

  /** Creates a Validator instance
  */
    public Validator createValidator( );

}
```

应用程序的web.xml描述项中要声明过滤，这是声明过滤的XML：

```
<filter>
<filter-name>ValidationFilter</filter-name>
<filter-class>hotel.presentationtier.ValidationFilter </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>ValidationFilter</filter-name>
    <url-pattern>/AdminHotelServlet/*</url-pattern>
  </filter-mapping>
```

注意本书的源代码下载提供了这个应用程序的完整源代码，包括XML描述项。

将模式汇编成框架

前面介绍了宾馆管理用例中使用的下列模式及其实现方法：

- **Intercepting Filter（截获过滤器）**

  Intercepting Filter设计模式可以截获请求和对其采用一组过滤，然后退回请求或让其传递到所要目标。

- **Front Controller（前端控制器）**

  Front Controller设计模式对常见请求处理工作采用集中控制，并委托给下一个视图。

- **View Helper（视图帮助器）**

  View Helper设计模式帮助视图分离与视图表示不直接相关联的任务为不同的帮助器类。

- **Composite View（复合视图）**

  Composite View设计模式可以把父视图分离为子视图，页面布局由独立的内容来管理。

- **Command（命令）**

  Command设计模式表示把命令或动作包装为对象。

- **Abstract Factory（抽象工厂）**

  Abstract Factory可以创建相关对象系列而不暴露其具体类。

最后要组合与浏览应用程序的整个框架，可以用两个J2EE表示模式作为这个框架的基础。包括：

- Service-to-Worker
- Dispatcher View

每个表示模式本身是个宏模式或微框架，我们将在下面两节介绍这些表示模式，用表格比较两者的微小差别。

Service-to-Worker设计模式

Service-to-Worker设计模式将两个子模式汇编成一个微框架，即Front Controller与View Helper，其参与者是控制器、派遣器、视图和帮助器的组合。

将这些模式汇编成可行框架对充分利用每个模式的长处至关重要。

和Front Controller设计模式中一样，Service-to-Worker设计模式集中控制并在中央位置放上多个请求的系统服务与业务逻辑，从而提高模块化和复用性，并把视图中的非表示逻辑分离到帮助器类中，分开模型与视图。图2.20显示了Service-to-Worker设计模式的参与者。

Service-to-Worker设计模式的参与者包括：

- **Controller（控制器）**

  这是处理请求的初始点和集中请求处理单元，用派遣器委托下一视图，用帮助器将功能块分解到不同模块中，负责验证、授权和委托。

- **Dispatcher（派遣器）**

  负责视图管理与导航。派遣器委托下一视图。在Service-to-Worker设计模式中，派遣器用帮助器将数据推到视图中。

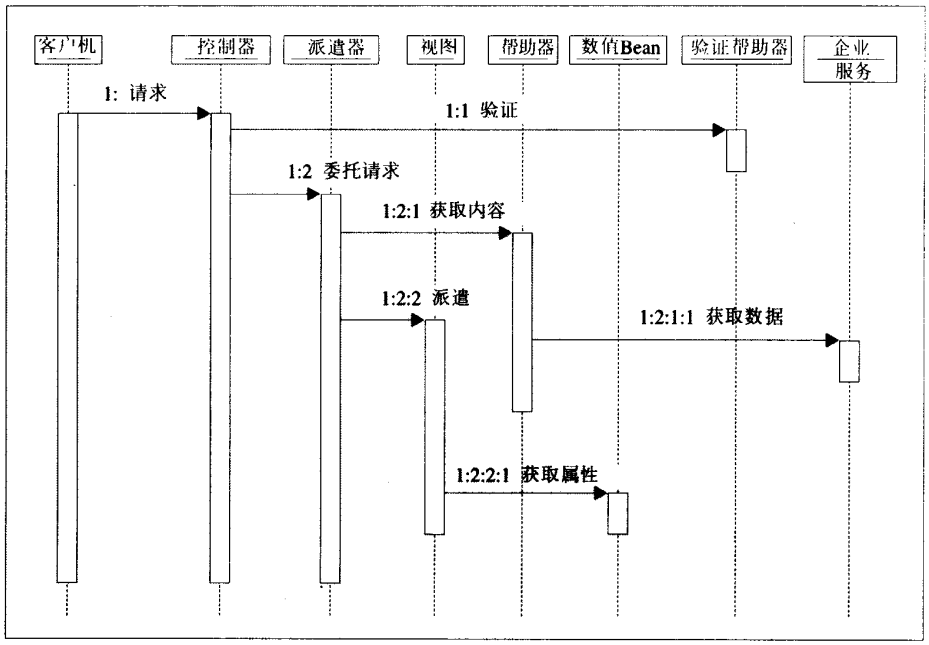

图2.20

- **View（视图）**
  向客户机表示和显示信息。
- **Helper（帮助器）**
  帮助视图或控制器完成处理工作。在Service-to-Worker设计模式中，可以用帮助器将数据推到视图中。
- **Value Bean（数值Bean）**
  这是帮助器的别名。
- **Business Service（业务服务）**
  客户机要访问的服务，是业务层的第一访问点，通常表示为业务委托。

Dispatcher View设计模式

Dispatcher View设计模式将两个子模式汇编成一个微框架，即Front Controller与View Helper。Service-to-Worker模式、Dispatcher View参与者是控制器、派遣器、视图和帮助器的组合。

Dispatcher View设计模式与Service-to-Worker设计模式描述相似的结构，但其参与者的作用不同。Dispatcher View将内容读取延迟到动态生成视图时，而Service-to-Worker则在控制器中先进行内容读取。

图2.21显示了Dispatcher View设计模式的参与者，注意Authentication Helper（验证帮助器）的作用。

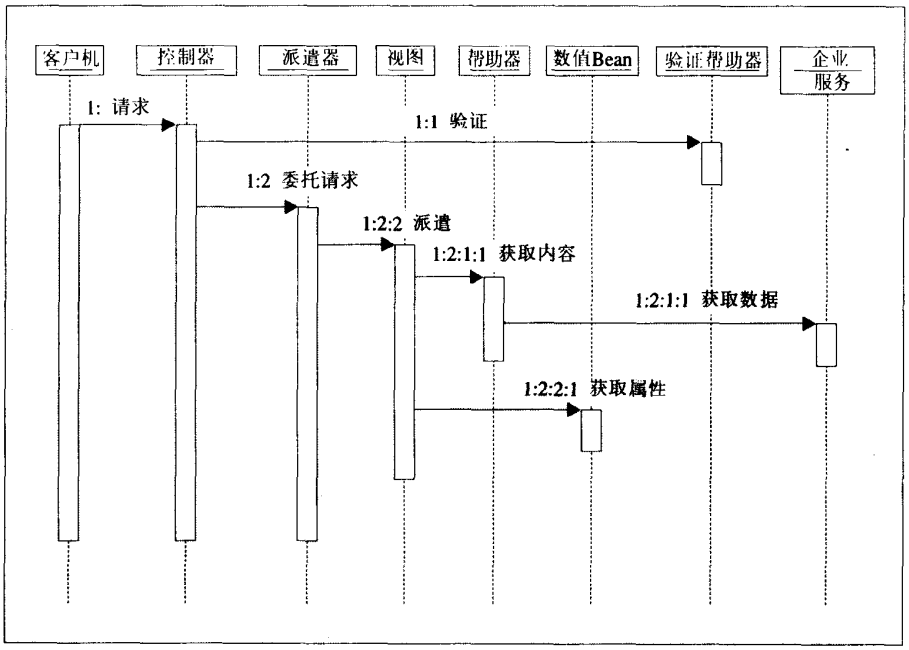

图2.21

Dispatcher View设计模式的参与者包括：

- **Controller（控制器）**
  这是处理请求的初始点和集中请求处理单元，还负责验证、授权和委托。
- **Dispatcher（派遣器）**
  负责视图管理与导航。派遣器委托下一视图。
- **View（视图）**
  向客户机表示和显示信息。在Dispatcher View中，视图用视图帮助器从数据源取得数据。
- **Helper（帮助器）**
  帮助视图或控制器完成处理工作。
- **Value Bean（数值Bean）**
  这是帮助器的别名。
- **Business Service（业务服务）**
  客户机要访问的服务，是业务层的第一访问点，通常表示为业务委托。

比较Dispatcher View与Service-to-Worker

尽管Dispatcher View设计模式与Service-to-Worker设计模式共享相似的结构和参与者，但两者有微妙的差别。

表2.1显示Dispatcher View与Service-to-Worker的相似与相异之处。

表2.1

Service-to-Worker模式	Dispatcher View模式
是表示层模式组合而成的宏模式或微框架	是表示层模式组合而成的宏模式或微框架
由两个子模式Front Controller与View Helper组成	由两个子模式Front Controller与View Helper组成
其参与者是控制器、派遣器、视图和视图帮助器的组合	其参与者是控制器、派遣器、视图和视图帮助器的组合
控制器与派遣器的作用很大，除了集中请求处理、视图管理和导航功能外，派遣器还从业务层读取内容	控制器与派遣器的作用较小
视图负责表示派遣器读取的数据	视图负责从业务层读取内容（在动态生成时）和显示数据
将更多逻辑与行为移到前端的控制器、派遣器和帮助器中，简化视图	将更多逻辑与行为移到视图及其帮助器中，使视图更复杂，通常要用脚本代码或定制标志完成控制器没有完成的任务
采用MVC推	采用拉MVC

有时要用Dispatcher View模式而不是Service-to-Worker模式，但这里选择Service-to-Worker，因为对控制器指定更多责任似乎更好，更容易管理视图和了解视图结构，从而提高维护性。

Service-to-Worker实现

我们的宾馆例子实现Service-to-Worker模式，使用控制器小服务和命令与控制器策略。本章前面曾介绍过，需要把上述表示模式汇编成可行的框架，从而得到每个模式的好处。此外，Service-to-Worker模式提供了很好的结构，使这些组件更容易复用和维护。较好的结构也使将来修改时具有更大的灵活性。

Service-to-Worker实现的细节请参考下列各节：

- Front Controller设计模式实现
- Command设计模式实现
- View Helper设计模式实现
- Intercepting Filter设计模式实现
- Composite View设计模式实现

最后，我们要显示前面介绍并采用的所有模式，汇编到一个框架中（如图2.22所示）。图2.23显示了实际类关系和上述模式如何映射类、接口、HTML与JSP页面。

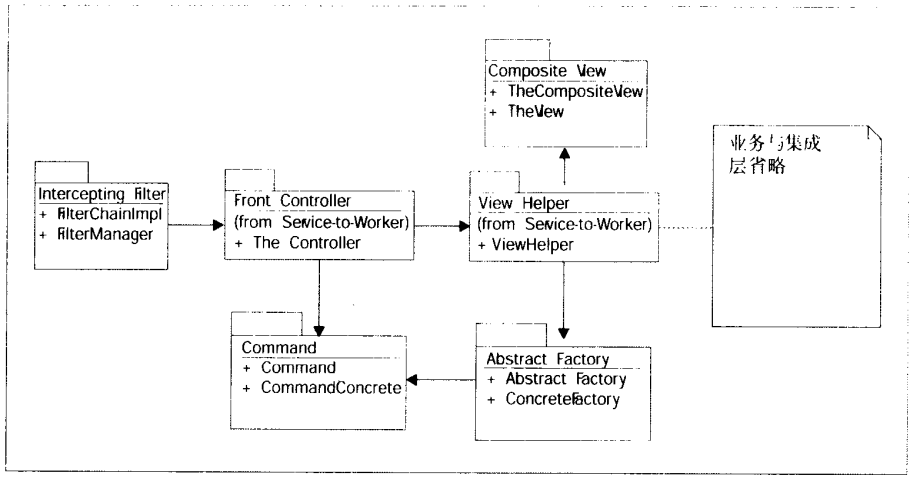

图2.22

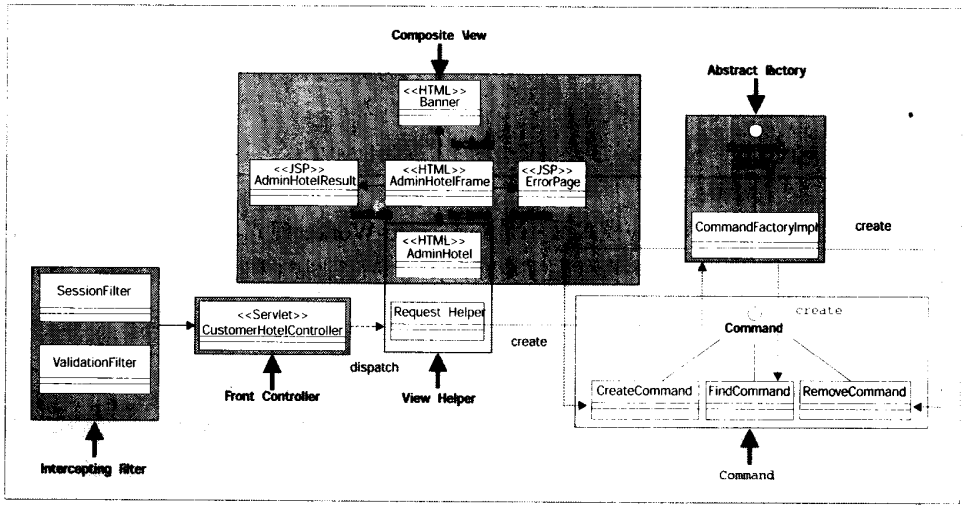

图2.23

Hotel Application Admin站点

部署和运行这个应用程序时，主管理页面如图2.24所示。

这个屏幕具有简单复合视图框架和两个子视图，是选择视图与结果视图。要创建新的宾馆项目，选择下拉清单中的create选项，输入宾馆号和宾馆名，然后选择条件。单击Submit按钮，结果在结果视图中显示，如图2.25所示。

要寻找一个宾馆，选择下拉清单中的find选项，输入宾馆号和宾馆名。单击Submit按钮，结果在结果视图中显示，如图2.26所示。

要删除宾馆项目，选择下拉清单中的remove选项，输入宾馆号和宾馆名。单击Submit按钮，结果在结果视图中显示，如图2.27所示。

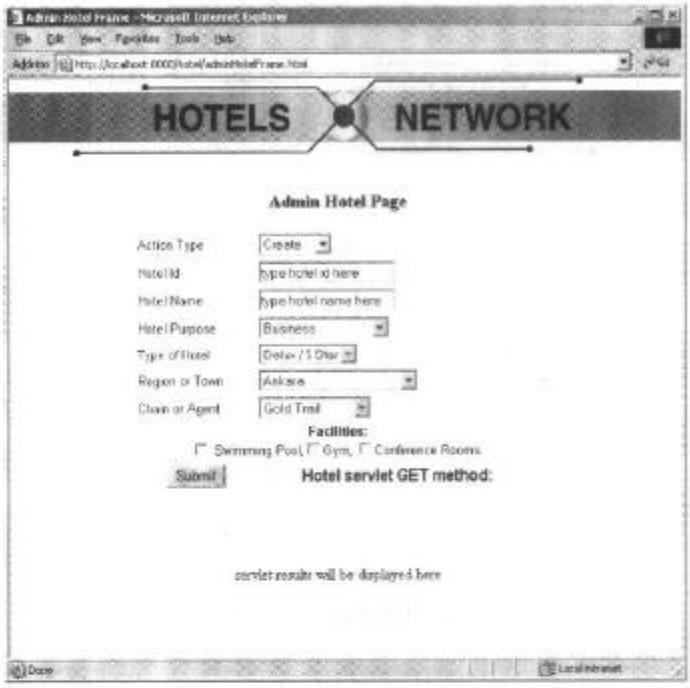
HOTELS NETWORK
Admin Hotel Page
Action Type
Create
Hotel Id
type hotel id here
Hotel Name
type hotel name here
Hotel Purpose
Business
Type of Hotel
Deluxe / 5 Star
Region or Town
Ankara
Chain or Agent
Gold Trail
Facilities:
Swimming Pool, Gym, Conference Rooms
Submit
Hotel servlet GET method:
servlet results will be displayed here

图2.24


HOTELS NETWORK
Admin Hotel Page
Action Type
Create
Hotel Id
tk_0006
Hotel Name
My Hotel
Hotel Purpose
Business
Type of Hotel
Deluxe / 5 Star
Region or Town
Ankara
Chain or Agent
Gold Trail
Facilities:
Swimming Pool, Gym, Conference Rooms
Submit
Hotel servlet GET method:
"tk_0006 is successfully created"
Hotel Details for My Hotel

图2.25


HOTELS NETWORK
Admin Hotel Page
Action Type
Find
Hotel Id
tk_0001
Hotel Name
Hotel Belle
Hotel Purpose
Business
Type of Hotel
Deluxe / 5 Star
Region or Town
Ankara
Chain or Agent
Gold Trail
Facilities:
Swimming Pool, Gym, Conference Rooms
Submit
Hotel servlet GET method:
"tk_0001 is found"
Hotel Details for Hotel Belle

图2.26


HOTELS NETWORK
Admin Hotel Page
Action Type
Remove
Hotel Id
tk_0006
Hotel Name
My Hotel
Hotel Purpose
Business
Type of Hotel
Deluxe / 5 Star
Region or Town
Ankara
Chain or Agent
Gold Trail
Facilities:
Swimming Pool, Gym, Conference Rooms
Submit
Hotel servlet GET method:
"Hotel Id tk_0006 is successfully removed"
Hotel Details for My Hotel

图2.27

小结

本章介绍了Web层管理的设计问题和企业应用程序设计，强调用Sun Java Center的J2EE表示层模式提供方案，包括Intercepting Filter模式、Front Controller模式、Composite View模式、View Helper模式、Service-to-Worker模式与Dispatcher View模式。

我们介绍的模式还包括《Elements of Reusable Object-Oriented Software》一书，由Addison-Wesley出版（ISBN 0-201-63361-2）中引入的模式，包括Command模式和Abstract Factory模式。

要点如下：

- 减轻客户层设计，只强调表示逻辑，把业务逻辑放在不同层中。
- 考虑构造客户层时可用的技巧。例如，将HTML与JSP代码和Java代码分开，这可以用Intercepting Filter与Front Controller之类的J2EE表示层模式实现。
- 尽量从高层检验用户输入，减少低层的检验和错误检查。
- 避免混合表示逻辑与导航逻辑。
- 更有效的方法是同时进行客户端和服务器方检验用户输入。
- 尽量减少客户端和服务器的交互。

第3章 持久性框架设计模式

数据是现代企业的血液，过去五到十年中，许多公司已经深切体会到，没有IT系统中的数据，企业就无法运转。数据不再被认为是日常业务操作的副产品，而是战略性资产。甚至可以说：

机构的数据在许多方面就是机构。没有一致和合理定义的服务集来访问机构中的由不同种类组成的数据源，就难以建立和集成应用程序。

但是，尽管人们越来越认识到机构数据的重要性，但IT部门通常没有深思熟虑的策略使应用程序访问与操纵数据。

IT部门通常把数据访问看成应用程序层问题，而不是从全面的、企业的角度看待数据使用，因此，可能失去良机，无法建立锁定公司数据源的数据访问与操纵的公共框架。

访问大多数软件开发人员，看看他们的应用程序是否基于基础开发框架，能够在应用程序之间方便地修改与复用，你可能遇到两种情形之一：

- 有的开发人员没有见过任何正式的软件体系结构或设计，因此不知道如何回答你的询问。
- 有的IT专家试图实现框架，但觉得用起来很麻烦，很难维护。他们相信框架能大大减少应用程序开发与维护工作的复杂性，但通常没有用足够的时间和经费正确建立持久性框架，最终设计出不好用的方案。

令人伤心的是，大多数应用程序都没有构造，是动态“编码”的，战术上提供某些业务功能，而不是着眼于战略目标，建立多个应用程序之间可以复用的软件服务集。

从短期看，开发人员能提供所要的功能，但对公司并不利，因为编写代码时没有考虑复用性与扩展性，使支持代码的长期成本很高。

许多IT部门认识到，要满足新应用程序的不断增长的需求，同时保持可控制成本，惟一的方法是用框架建立应用程序，提供一组可以复用的软件服务集。

要让IT机构正确设计、提供和维护任何类型的应用程序框架，就要求：

- **时间与经费**

  大多数机构不愿意花更多时间与经费建立可复用框架。系统体系结构是开发小组面对机构中不同业务单元的交货期压力时首先牺牲的项目。

- **递增实现**

  建立系统体系结构要求确定能提供量化结果的小模块。但大多数公司用大段方法实现公用系统体系结构。但是，结果常常令人失望，因为他们在编写初始框架后的几个月内通常无法看到任何直接业务好处。

- **愿意改变——体系结构不是设计出来的，而是演变出来的**

  这样，机构必须愿意重新考虑各种因素，重新设计代码，即使代码能够满足眼前的要求。有时，要改变整个体系结构以达到机构所需的复用性与扩展性。

许多IT开发小组把多层应用程序模型作为开发这种服务框架的方法。多层应用程序模型把应用程序分成三个逻辑层，每一层可以放在企业中不同硬件上。多层应用程序模型的三个逻辑层如下：

- 表示层——负责的最终用户处理信息表示的所有逻辑。
- 业务层——捕获机构的业务规则，作为多个应用程序间可复用的组件。
- 数据层——也称为集成层，负责抽象出数据源，使数据位于不同地方和用不同方法读取时，能够用业务规则访问与操纵数据。

人们通过大量研究寻找建立坚实表示层与业务层的方法。只要到书店看看，六、七年前就有关于建立表示层与业务层的书籍。

此外，还有许多开放源代码表示层与业务层框架，可以在应用程序开发项目中使用。其中两个例子见Apache Software Foundation公司的Jakarta web site（http://jakarta.apache.org）：

- Struts——Struts是个Model-View-Controller框架，基于Sun的JSP Model-2实现。提供几个应用程序插入点，明确分开应用程序表示层、逻辑层和数据层，变成定义合理的可维护组件。
- JetSpeed——JetSpeed是企业端开发框架。JetSpeed提供公用表示框架，可以迅速将Web应用程序集成为企业终端用户的聚合入口点。

这些框架显示了人们致力于分离表示与业务层逻辑。

但是，到最近以前，很少公司考虑过用任何正规的数据持久性框架建立数据层。大多数应用程序开发人员只是考虑中间层的业务逻辑复用。对许多公司而言，数据层是关系型数据库或公司大型机系统，不能明确映射要实现的许多面向对象机制。

多层应用程序的数据访问逻辑与中间层业务逻辑混合或在存储过程中模糊定义，通过存储过程向开发人员隐藏数据访问细节。

但是，越来越多的人们开始认识到，不能抽象出公司数据源的物理与厂家实现细节会直接导致应用程序长期维护的巨大成本。下节“何谓持久性框架”将介绍其中一些成本。

本章专门介绍用常见数据持久性设计模式与J2EE技术结合起来开发数据层。本章特别介绍下列内容：

- 何谓持久性框架？
- 实现持久性框架时可用的设计模式，包括：
  - Data Access Object模式
  - Value Object模式
  - Service Locator模式
- 建立持久性框架时要考虑的核心策略：
  - 主键生成
  - 并发性管理与锁
  - 事务管理
  - 性能问题

记住，实现框架通常是个观念问题。本章介绍的设计模式只是个准则。换句话说，它们是处理重复出现的编程问题的例子方案。本章显示的表示不是实现这些设计模式的惟一方

式。对所有数据持久性模式，要记住下列关键：

“数据持久性模式主要考虑建立逻辑门户，隐藏数据读取和操纵中的所有数据访问代码细节，完全抽象出开发小组建立应用程序时使用的数据源物理细节。”

假设你在一家假想杂志出版公司IT World Now的IT部工作，小组建立和部署了几个Java Web应用程序，功能是可行的。但IT部管理人员对维护现有应用程序的成本很不满意，而且编写新应用程序时，现有代码基础很难复用。IT部管理人员希望建立公用应用程序开发框架，在应用程序的三层中提供复用。

你要检查IT World Now预订数据库的不同方面，看看如何在这些数据库之上建立持久性框架，抽象出基础细节。你决定从预订数据库中取出几个数据项目，建立一个原型，由此建立预订持久性框架的其余部分。

原型用三个表建立，如图3.1所示。

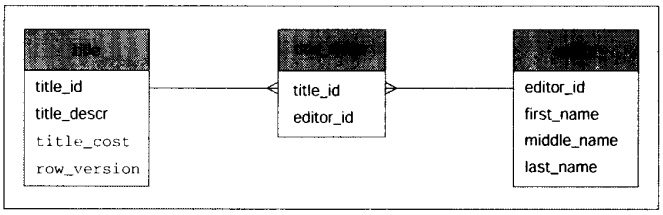

图3.1

- title——这个表保存预订数据库中现有的所有书名。
- editor——这个表保存IT World Now公司工作的所有编辑。
- title_editor——这是书名与编辑之间的多对多关系，一个编辑可以参与多本书，一本杂志可以有多个编辑。title_editor表建立书名和编辑之间的联系。

上表的SQL DDL可以从http://www.wrox.com/下载。

开始模型

下列JSP页面高亮显示了开发人员编写应用程序时常犯的错误。这个JSP页面要查询数据库预订的所有书名，然后在屏幕上显示查询结果。代码如下：

```
<%@page import="java.sql.*"%>
<%@page import="java.util.*"%>

<!--
   The following JSP page demonstrates a few of the more common mistakes
   made by application developers.
-->

<table border="1" align="center">
  <tr>
```

```
    <td align="center" nowrap>
      Id
    </td>
    <td align="center" wrap>
      Description
    </td>
    <td align="center" nowrap>
      Cost
    </td>
  </tr>
<%
  Connection conn = null;
  ResultSet  rs   = null;
  Statement statement = null;
  try{
    /* Problem # 1 */
    Class.forName("org.gjt.mm.mysql.Driver");

    /* Problem #2 */
    conn =  DriverManager.getConnection
      ("jdbc:mysql://192.168.1.98:3306/test/?username=gnu@192.168.1.98");

    /*Do something with the connection*/
    StringBuffer sql = new StringBuffer(64);
    sql.append("SELECT                    ");
    sql.append("  title_id    ,           ");
    sql.append("  title_descr ,           ");
    sql.append("  title_cost              ");
    sql.append("FROM                      ");
    sql.append("  title                   ");

    statement = conn.createStatement();

    rs   = statement.executeQuery(sql.toString());

    while (rs.next()) {
      out.println("<tr>");
      out.println("<td>");
      out.println( new Long( rs.getLong("title_id") ).toString() );
      out.println("</td>");

      out.println("<td>");
      out.println(   rs.getString("title_descr")  );
      out.println("</td>");

      out.println("<td>");
      out.println(   new Float (rs.getFloat("title_cost") ).toString() );
      out.println("</td>");
```

```
            out.println("</tr>");
          }
        } catch(SQLException e) {
           out.println("SQL Exception Error--> " + e.getMessage());
        }
        finally {
          if (statement!=null) try{ statement.close();} catch(SQLException e){}
          if (rs!=null) try{ rs.close(); } catch(SQLException e){}
          if (conn!=null) try{ conn.close();} catch(SQLException e){}
        }
    %>
    </table>
```

这个例子显示了数据库开发人员编写应用程序时典型的错误，包括：

- 没有把表示、业务、数据逻辑明确分解到不同层
- 应用程序要涉及数据所在的数据库平台
- 应用程序代码直接关联到数据库表格的物理结构（即title表）
- 强制应用程序开发人员学习和了解获取与操纵数据时所用数据访问API的细节

分解应用程序层

第一个问题是没有把表示、业务、数据逻辑明确分解到不同层，至今仍然是个常见的问题。尽管有大量多层体系结构书籍和现成的开放源代码框架，许多开发人员仍然没有把应用程序中的表示、业务、数据逻辑明确分解到不同层。

```
while (rs.next()) {
       out.println("<tr>");
       out.println("<td>");
       out.println(  new Long( rs.getLong("title_id") ).toString() );
       out.println("</td>");

       out.println("<td>");
       out.println(   rs.getString("title_descr")  );
       out.println("</td>");

       out.println("<td>");
       out.println(   new Float (rs.getFloat("title_cost") ).toString() );
       out.println("</td>");
       out.println("</tr>");
     }
```

JSP页面例子没有任何业务逻辑，但它没有分开表示层与数据层。没有把表示、业务、数据逻辑明确分解到不同层的结果是数据库代码会分散在应用程序中。对于小应用程序，这个问题不大，但对于大型应用程序，则这样可能很难维护。即使数据库的微小改变也会迫使开发人员查找应用程序代码中的业务代码。

隐藏数据库平台

这个JSP例子以极端形式显示了第二点。在上述JSP页面中，开发人员直接把连接信息嵌入JSP页面中：

```
/* Problem # 1 */
Class.forName("org.gjt.mm.mysql.Driver");

/* Problem #2 */
conn =  DriverManager.getConnection
      ("jdbc:mysql://192.168.1.98:3306/test/?username=gnu@192.168.1.98");
```

JSP页面直接知道要通过Java包与MySQL数据库交流，MySQL JDBC驱动器类名传入Class.forName()方法调用中。此外，代码中还嵌入了数据库连接信息和用户ID与口令。

也许你以为上述两个问题是小问题，但这些常见错误通常会在维护代码时造成大量开销。假设按上述风格编写大型应用程序，应用程序有40个到50个屏幕，多种情形需要重访每个屏幕，修改数据库连接信息。重访每个屏幕需要大量“简单”的编程工作和重新测试整个应用程序，保证访问与改变了所有数据库接触点。触发这种改变的例子包括：

- 数据库要移到不同服务器中，具有新的IP地址和机器名。
- 要改变应用程序所在的数据库平台。应用程序可能用MySQL之类开放源代码数据库，你准备改用Oracle DBMS平台。
- 开发人员破坏数据库安全性。任何能访问应用程序源代码的人也能够访问应用程序所访问的数据库。JSP页面更是如此，因为许多开发人员并不预编译代码，让应用程序服务器直接编译JSP源代码。
- 你发现数据库连接使用的池管理器有严重缺陷，准备改用另一厂家的池管理器。

最后，大多数开发人员发现不宜把数据库连接信息直接嵌入应用程序代码中，而通常把数据库连接过程包装在JDBC Connection工厂类的某个变体中，实例化JDBC Connection。但是，这个代码段演示了一个常见问题。开发人员通常把应用程序代码及其读取数据的数据库平台耦合起来，不隐藏数据库连接信息或直接在应用程序代码中使用厂家特定的数据库特性。

例如，Oracle数据库向开发人员提供了生成顺序编号的数据库对象，这个顺序及其在SQL语法中的用法是Oracle特定的。如果开发小组在应用程序中使用这个对象，则把应用程序移植到另一数据库平台时会遇到大量无效时间，因为他们要搜索整个应用程序，将代码改成使用其他一些顺序生成机制。

隐藏数据实体的物理结构

在上述JSP例子中，应用程序代码直接关联到数据库表格的物理结构（即title表）。上例中JSP页面直接在页面中发出SQL语句，从title表读取数据：

```
sql.append("SELECT                    ");
sql.append("  title_id    ,           ");
sql.append("  title_descr ,           ");
sql.append("  title_cost              ");
sql.append("FROM                      ");
```

```
sql.append("  title                    ");
```

直接在JSP页面中发出SQL语句就等于连接到title表的物理结构。如果title表中改变或删除用到的任何列，则开发人员要找到接触title表的每个应用程序，保证其不会破坏代码。

在应用程序中暴露数据实体的物理结构还可能在subscription数据库中title表与其他表之间的关系发生改变时破坏代码。上述SQL语句只是从title表读取数据，但本章稍后会介绍title与editor表之间具有多对多关系。如果数据库修改成使title与editor表之间具有一对多关系，则要修改对subscription数据库发出任何SQL命令的所有应用程序。

包装细节

这个代码例子的最后一个问题是每当应用程序开发人员想要从Subscription数据库中读取与操纵数据时，他们必须了解数据访问API的细节，如JDBC或JDO（Java Data Objects）。开发人员要了解如何连接数据库和查询语言语法，了解事务管理与隔离之类的概念。

大多数IT机构把应用程序开发人员分成层，初、中级开发人员建立业务应用程序，而高级开发人员建立基础结构和体系结构代码。这些机构要达到这个目标，就要保证向业务应用程序开发人员隐藏复杂和重复的工作，如从公司数据库访问数据。这些开发人员不要求知道如何取得数据，到哪里取得数据。业务应用程序开发人员要考虑编写业务代码，而不是基础结构代码。

何谓持久性框架

持久性框架是一组软件服务，将应用程序与其使用和操纵的数据源分离。持久性框架位于机构的数据源之上，隐藏访问这些数据源的数据访问API（如JDBC、JDO或实体EJB）。提供的服务应完全抽象和从这个数据源使用和操纵数据的物理细节。图3.2演示了持久性框架在系统体系结构中的地位。

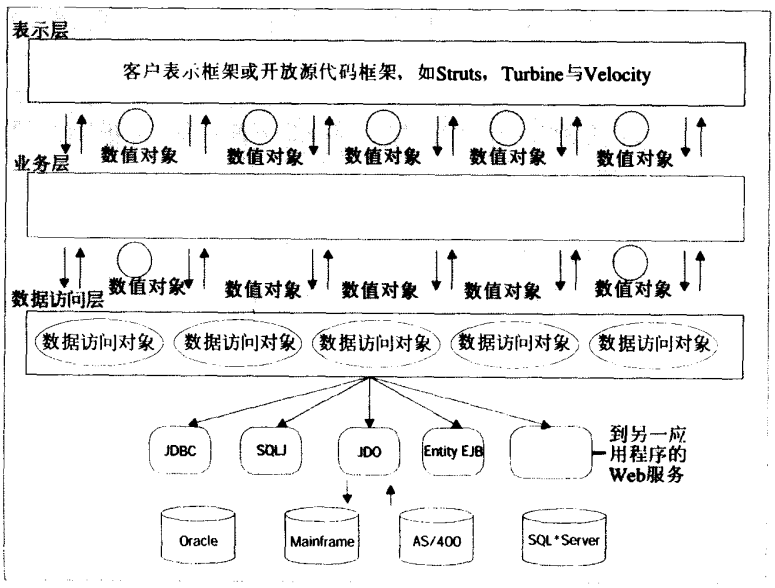

图3.2

图中，数据持久性框架完全抽象了访问数据方法的细节，严格分开表示、业务与数据层，同时使用两个设计模式：

- Data Access Object设计模式
- Value Object设计模式

Data Access Object设计模式完全包装数据读取与操纵，并包装与数据库交互的数据访问API，如JDBC、JDO、SQLJ，等等。Data Access Object模式用数值对象包装从数据库读取和发送到数据库的数据，与业务层通信。

图中显示的Value Object设计模式包装从数据库读取和发送到数据库的数据。Value Object模式向业务应用程序开发人员隐藏物理表格结构，抽象数据库中的数据关系，用Java Collection机制从数据库中取得父子关系、一对多关系和多对多关系。数值对象也可以向上传递到表示层，向最终用户显示信息。

在大机构中，几十个系统使用不同的数据访问机制。此外，机构中的数据可能由机构中不同系统访问，这些系统运行不同的数据库平台。为什么要向编写业务应用程序的开发人员显示这些细节呢？持久性框架可以向编写业务应用程序的开发人员隐藏这些细节。

持久性框架中提供下列服务，让开发人员在持久性框架之上建立应用程序：

- **提供分开数据持久逻辑与表示和业务逻辑的整洁机制**

  在上面的体系结构框图中，只有业务层可以访问公司数据库。业务层用Data Access Object模式与公司数据库交互，而不用JDBC之类的数据访问API访问数据库。

- **不让框架上建立的应用程序了解数据所在的数据库平台**

  将所有数据连接逻辑包装到数据访问对象中后，业务应用程序的开发人员不需要知道连接的数据库平台类型，连接数据库时所需的任何安全信息（用户ID与口令）或数据库网址。

- **抽象数据库中存储数据的物理细节和数据库中数据实体之间存在的关系**

  建立在上述体系结构布局之上的应用程序不必直接对数据库发出SQL查询，不必知道数据的物理结构，而用数值对象访问数据库。

- **简化开发过程，隐藏打开数据库连接、发出数据读取与操纵命令和事务管理的细节**

  Data Access Object与Value Object模式完全分离业务应用程序的开发人员与其在应用程序中使用的数据源，不需要知道数据访问API，而用简单接口读取与操纵数据。

下面详细介绍上述模式中的第一个模式Data Access Object，介绍其实现方法。

Data Access Object模式

Data Access Object（DAO）与Value Object（VO）模式是Sun Java Center中最先提出的。DAO模式抽象从数据源获取与操纵数据的方法。DAO模式有两个作用：

- 第一，DAO模式完全抽象用户请求的数据所在数据源。DAO模式的用户不知道数据来自Oracle数据库、Microsoft SQL Server、LDAP服务器或Web服务，数据源是完全透明的。
- DAO模式提供的第二个功能是抽象通常与访问数据源相关联的CRUD（Create、Replace、Update与Delete）逻辑。由于用户不知道如何查询和读取数据，因此改变数据

访问代码时（如DAO中发出的SQL语句）不受影响。DAO模式甚至允许应用程序建筑师从多个数据源读取数据，用一个逻辑对象向用户显示（换句话说，数值对象可以包含来自多个数据源的数据）。

DAO模式提供的这种灵活性是由于应用程序并不直接访问数据源，而是创建DAO对象，用其访问数据源。读取数据时，可以用数值对象保存取得的数据。Value Object模式抽象数据库中数据存储的物理结构。Value Object模式见本章稍后介绍。

图3.3显示了客户机代码如何用DAO对象取得数据。

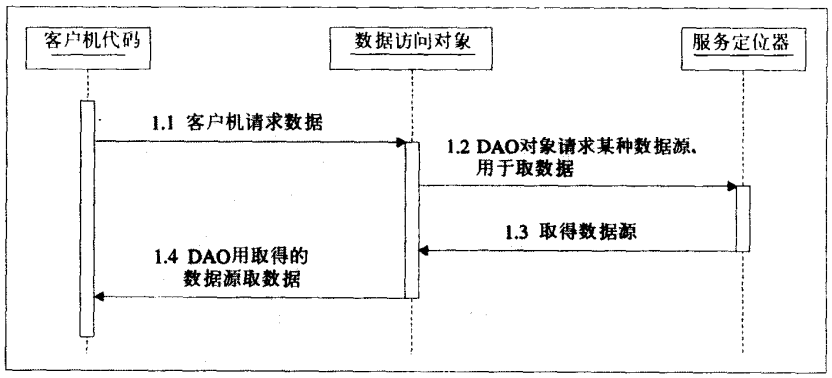

图3.3

DAO模式很强大，因为：

- 它抽象数据源读取数据的方法。DAO模式可以用任何Java数据访问API进行处理，包括：
  - JDBC API——JDBC是第一个标准数据库访问API，位于J2EE中，是J2SE的扩展。
  - SQLJ——SQLJ是较新的API，是几个一流数据库和Java应用程序服务器厂家提出的。SQLJ使开发人员可以把SQL代码直接嵌入Java代码中，然后用代码预处理器将其转换成Java代码。
  - Entity Enterprise Java Beans——实体EJB规范是数据持久性规范，开发人员可以抽象组件体系结构中的数据库代码。
  - Java Data Objects（JDO）——JDO规范是最新的数据库访问API，可以用标准Java对象取得和保存数据库中的数据。JDO对象比实体EJB简单，同时仍然提供与实体EJB相似的功能。
- 整洁分离业务与数据逻辑，减少应用程序开发人员混合业务与数据逻辑的情形。把所有数据访问代码集中到一个逻辑结构中之后，开发人员可以用逻辑接口读取与操纵数据。
- 提供一组标准编程接口，可以解决常见数据库开发问题，包括：
  - 主键生成
  - 并发性管理与锁
  - 事务管理

- 性能问题

持久性框架异常

持久性框架中捕获和发出的任何异常用DataAccessException重新抛出。这个重新抛出有两个目的：

- 捕获数据持久性框架中抛出的所有异常之后，就可以按一致方式处理和捕获所有异常。使用DataSourceFactory之类数据持久性框架类的开发人员只要捕获一个异常。
- 将DataSourceFactory中或数据持久性框架任何部分发出的异常重新抛出之后，就可以抽象持久性框架访问的数据源。持久性框架开发人员提供了一致异常，总是可以在使用框架的应用程序代码中捕获。

DataAccessException的代码如下：

```
package dataaccess;

public class DataAccessException extends Exception {
  Throwable exceptionCause = null;

  public DataAccessException(String pExceptionMsg) {
    super(pExceptionMsg);
  }

  public DataAccessException(String pExceptionMsg, Throwable pException) {
    super(pExceptionMsg);
    this.exceptionCause = pException;
  }

  public void printStackTrace() {
    if (exceptionCause != null) {
      System.err.println("An exception has been caused by: ");
      exceptionCause.printStackTrace();
    }
  }
}
```

DataAccessException类是一般化异常，可以重新抛出发出的任何非持久性框架异常和持久性框架处理过程中可能产生的任何小问题。框架中其他更特定的异常反映的特定事件可以在使用DAO的应用程序中跟踪和捕获。这些异常扩展DataAccessException，并且包括：

- NoDataFoundException
- OptimisticLockException
- ServiceLocatorException

后面介绍持久性框架时会介绍这些异常。

实现Data Access Object模式

DAO模式抽象了数据读取和操纵相关的任务，定义的Java接口有五个基本方法：

- findByPrimaryKey（Object pPrimaryKey）
- insert（ValueObject pValueObject）
- createValueObject()
- update（ValueObject pValueObject）
- delete（ValueObject pValueObject）

findByPrimaryKey()方法从数据源取得一个记录，取一个输入参数pPrimaryKey。这个参数是数据源中记录的惟一ID。在本章的例子中，这个参数表示关系型数据库中记录的主键字段。findByPrimaryKey()方法返回ValueObject类型的对象。ValueObject是个包装类，包装从数据源读取的数据，DAO表示为Java类。这样，使用DAO的代码并不直接与DAO包装的基础数据结构相联系。

insert()、update()与delete()方法分别进行相应操作，取一个ValueObject类型的参数，保存进行insert、update或delete操作时使用的所有数据。

为什么对delete()方法传入整个ValueObject，而不是只传入主键ID呢？主要是为了效率。如果DAO表示的数据实体要满足多个条件才能删除记录，则可以用ValueObject将数据传入DAO中，否则要检查每个条件之后才能删除记录，DAO要对每个条件进行数据库查找，确定是否符合条件。

在第三方厂家编写的数据库之上建立DAO层时，通常会遇到这个条件。作为持久性框架开发人员，你不能控制这些厂家在数据删除时如何支持数据实体的引用完整性。许多厂家把引用完整性代码直接嵌入应用程序中。这样，就要将数据库行为逆向转出工程代码，将这个行为放进DAO中，保证在数据库中删除记录时发生正确的插入、更新与删除。

createValueObject()方法创建空的ValueObject，可以填入数据，然后传递到DAO的insert()方法，插入DAO表示的数据源。

由于DAO模式抽象数据读取和操纵机制，因此可以用两种不同方式实现：

- 作为标准Java类
- 作为J2EE会话EJB

本章建立的所有DAO都作为使用JDBC的无状态会话EJB，也可以用简单Java类建立DAO。

简单Java类很容易实现，调试时间也更短，因为编写类的设置工作少得多。但是，本章选择J2EE无状态会话EJB方法，因为运行EJB的应用程序服务器提供了几个核心基础结构服务。这些基础结构服务包括：

- **基于容器的事务管理**

  使数据库开发人员不必负责在应用程序中编码复杂的事务管理代码。

- **声明式访问控制**

  声明式访问控制使开发人员可以控制用户与应用程序用什么通过会话EJB进行工作。

- **资源池**

  资源池能改进应用程序性能，将会话EJB池用于处理用户请求。

要对基于EJB的DAO建立框架，我们准备扩展EJB远程接口（EJBObject）和基础类（SessionBean），包括前面定义的五个DAO方法签名。

DAO远程接口

DAOObject接口是数据访问对象的远程接口：

```
package dataaccess;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import valueobject.ValueObject;

public interface DAOObject extends EJBObject {

  public ValueObject findByPrimaryKey(Object pPrimaryKey)
     throws DataAccessException, NoDataFoundException,
            ServiceLocatorException, RemoteException;

  public void insert(ValueObject pValueObject)
     throws DataAccessException, ServiceLocatorException,
            RemoteException;

  public void update(ValueObject pValueObject)
     throws DataAccessException, OptimisticLockException,
            ServiceLocatorException, RemoteException;

  public void delete(ValueObject pValueObject)
     throws DataAccessException, ServiceLocatorException,
            RemoteException;

  public ValueObject createValueObject()
     throws DataAccessException, ServiceLocatorException,
            RemoteException;
}
```

DAOBean类

DAOBean类扩展标准会话EJB实现中使用的SessionBean接口：

```
package dataaccess;

import javax.ejb.SessionBean;
import valueobject.ValueObject;

public interface DAOBean extends SessionBean {
  public ValueObject findByPrimaryKey(Object pPrimaryKey)
     throws DataAccessException, NoDataFoundException,
            ServiceLocatorException;

  public void insert(ValueObject pValueObject)
     throws DataAccessException, ServiceLocatorException;

  public void update(ValueObject pValueObject)
     throws DataAccessException, OptimisticLockException,
            ServiceLocatorException;
```

```
    public void delete(ValueObject pValueObject)
        throws DataAccessException, ServiceLocatorException;

    public ValueObject createValueObject()
        throws DataAccessException, ServiceLocatorException;

}
```

TitleDAO会话Bean

我们要建立一个DAO对象，表示预订数据库中的title表。由于这个DAO用会话EJB实现，因此包括三个Java文件：

- TitleDAO
- TitleDAOHome
- TitleDAOBean

最后，TitleDAO接口是EJB的远程接口，扩展DAOObject接口如下：

```
package session.title;

import dataaccess.DAOObject;

public interface TitleDAO extends DAOObject {}
```

然后，TitleDAOHome接口扩展EJBHome接口，负责返回TitleDAOBean类：

```
package session.title;

import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import javax.ejb.CreateException;

public interface TitleDAOHome extends EJBHome {

  public TitleDAO create() throws RemoteException, CreateException;

}
```

TitleDAOBean类

TitleDAOBean类持有TitleDAO代码的实际实现，这个类包装表示与业务逻辑中通常提供和交织的所有数据库代码。

从TitleDAOBean实现中查找，一定要注意下列两点：

- TitleDAO实现向开发人员隐藏数据读取与操纵方法，很容易从JDBC转换成其他某种数据访问API。
- title表的物理结构很容易修改，很容易规范化和反规范化。

这两个任务都很容易完成，因为TitleDAO提供的公开接口完全隐藏基础数据库代码。如果对数据结构作过大改变，则很难防止这种改变波及机构中的所有应用程序，但DAO提供了一个缓冲区，可以允许应用程序体系结构隔离应用程序与对数据库进行的大多数改变。

如果没有DAO对象提供的抽象层，则数据库的微小改变也可能要大量改写代码，在企业的多个应用程序中进行测试。

下面是TitleDAOBean类的框架（每个方法将在稍后详细介绍）：

```
package session.title;

import dataaccess.*;
import valueobject.*;
import session.counter.*;
import session.editor.*;
import javax.ejb.*;
import java.sql.*;
import java.rmi.RemoteException;
import java.util.HashMap;
import java.util.Collection;
import java.util.Iterator;

public class TitleDAOBean implements DAOBean {
  private SessionContext ctx;
  /*
   * The findByPrimaryKey() method will retrieve a single title
   * record from the subscription database. It returns a TitleVO
   * that contains all of the title data.
   */
  public ValueObject findByPrimaryKey(Object pPrimaryKey)
      throws DataAccessException, ServiceLocatorException,
             NoDataFoundException {
  }
  /*
   * The update() method will update a single record in the title
   * table with the valueobject passed in as a parameter.
   */
  public void update(ValueObject pValueObject)
      throws DataAccessException, OptimisticLockException,
             ServiceLocatorException {
  }

  /*
   * The delete() record will delete a single record from the title
   * table in the subscription database. It does this based on the
   * id of the value object passed into the table.
   */
  public void delete(ValueObject pValueObject)
      throws DataAccessException, ServiceLocatorException {
  }

  /*
```

```
     * The insert() method will insert a single record into the title
     * table. The data it inserts will be pulled from the value
     * object passed into the method.
     */
    public void insert(ValueObject pValueObject)
        throws DataAccessException, ServiceLocatorException {
    }

    /*
     * The createValueObject() method will create an empty value
     * object prepopulated with any mandatory data.  n example of
     * this mandatory data might be a primary key that for a particular
     * record.
     */
    public ValueObject createValueObject()
        throws DataAccessException, ServiceLocatorException {
    }

    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sessionCtx) {
      this.ctx = sessionCtx;
    }

}
```

findByPrimaryKey()方法

findByPrimaryKey()方法从title表取一个记录，返回ValueObject对象形式的数据。本章稍后将显示其实现方法。

TitleDAO.findByPrimaryKey()方法的代码用JDBC调用从subscription数据库取得请求的记录，创建一个ValueObject对象，填入数据，并返回调用find()方法的应用程序：

```
    public ValueObject findByPrimaryKey(Object pPrimaryKey)
        throws DataAccessException, ServiceLocatorException,
               NoDataFoundException {

      String pkPrimaryKey = (String) pPrimaryKey
      ServiceLocator serviceLocator = ServiceLocator.getInstance();
      Connection conn =
        serviceLocator.getDBConn(ServiceLocator.SUBSCRIPTIONDB);
      ResultSet rs   = null;

      //Getting my SQL Code for retrieving all of the titles
      SQLCode sqlCode = SQLCode.getInstance();
      String sql =
```

```
    sqlCode.getSQLStatement("titledao.findbyprimarykey().select");

  TitleVO titleVO = new TitleVO();

  try {
    PreparedStatement preparedStatement = conn.prepareStatement(sql);
    preparedStatement.setLong(1, Long.parseLong(pkPrimaryKey));
    rs = preparedStatement.executeQuery();

    // Populate a the value object with the resultset data
    if (rs.next()){
      titleVO.setTitleId(rs.getLong("title_id"));
      titleVO.setTitleDescr(rs.getString("title_descr"));
      titleVO.setTitleCost(rs.getFloat("title_cost"));
      titleVO.setRowVersion(rs.getLong("row_version"));

      EditorDAOHome editorDAOHome =
        (EditorDAOHome)serviceLocator.getEJBHome(
                                        ServiceLocator.EDITORDAO);
      EditorDAO editorDAO = editorDAOHome.create();
      titleVO.setEditors(
         editorDAO.findEditorByTitle(String.valueOf(
                                    titleVO.getTitleId())));
    } else {
      throw new NoDataFoundException(
                "Record id: " + pkPrimaryKey +
                "is not found in the title table.");
    }

    //Reset the flags so that we know the object is in pristine state.
    titleVO.resetFlags();
  } catch(SQLException e) {
    //Aborting the transaction
    ctx.setRollbackOnly();
    throw new DataAccessException(
              "Error in TitleDAO.findByPrimaryKey()", e);
  } catch(RemoteException e) {
    //Aborting the transaction
    ctx.setRollbackOnly();
    throw new DataAccessException(
              "Error in TitleDAO.findByPrimaryKey()",e);
  } catch(CreateException e) {
    //Aborting the transaction
    ctx.setRollbackOnly();
    throw new DataAccessException(
              "Error in TitleDAO.findByPrimaryKey()",e);
  } finally {
```

```
        try {
          if (rs != null) rs.close();
          if (conn != null) conn.close();
        } catch(SQLException e) {
          System.out.println("I am unable to close a resultset, " +
                             "statement, or connection " +
                             "TitleDAO.findByPrimaryKey().");
        }
        return titleVO;
      }
    }
```

findByPrimaryKey()方法前几行代码好像不太一般，findByPrimaryKey()方法首先把主键变元转换成String类型，因为应用程序中要使用这个数据类型，但也可以把主键变元转换成任何其他类型。

然后这个方法用ServiceLocator对象取得subscription数据库的JDBC Connection和持久性框架中使用的所有EJB的EJBHome接口：

```
ServiceLocator serviceLocator = ServiceLocator.getInstance();

Connection conn =
  serviceLocator.getDBConn(ServiceLocator.SUBSCRIPTIONDB);
```

ServiceLocator.getDBConn()方法取得subscription数据库的JDBC数据库Connection，本章稍后将介绍ServiceLocator代码。

取得数据库连接之后，我们用SQLCode对象取得SELECT语句，从title表中取得数据：

```
SQLCode sqlCode = SQLCode.getInstance();
String sql =
  sqlCode.getSQLStatement("titledao.findbyprimarykey().select");
```

SQLCode类在一个地方保存持久性框架中使用的所有SQL语句。需要SQL语句的DAO可以取得SQLCode类的实例，然后调用SQLCode.getSQLStatement()方法。调用这个方法时，传入的键值映射所要的SQL语句。SQL语句在TitleDAO.findByPrimaryKey()方法中使用如下：

```
SELECT title_id, title_descr, title_cost, row_version FROM title WHERE
       title_id=?
```

SQLCode类基于Singleton设计模式，有两个作用：

- 在属性文件sql.properties中集中应用程序使用的所有SQL语句。利用SQLCode类时，持久性框架中所有SQL代码的改变都发生在这个地方。
- 不需要在DAO的每个find()、insert()、update()和delete()方法中直接建立SQL语句。SQLCode类只有特定SQL语句的一个静态拷贝。这样，如果特定DAO方法调用十几次，则虚拟机中只存储一个表示该SQL语句的String对象。

SQLCode类如下，但这是非常简单的Java代码，本章不准备详细介绍。

```
package dataaccess;

import java.util.Properties;
import java.io.*;

/*
 *  The SQLCode class is a helper class that loads all of the SQL
 *  code used within the persistence framework into a private properties
 *  object stored within the SQLCode class.
 */
public class SQLCode {
  private static SQLCode sqlCode = null;
  private static Properties sqlCache = new Properties();

  /*Calling the SQLCode's constructor*/
  static {
    sqlCode = new SQLCode();
  }

  /*Retrieves the Singleton for the SQLCode class*/
  public static SQLCode getInstance() {
    return sqlCode;
  }

  /*
   * The SQLCode constructor loads all of the SQL code used in the
   * persistence framework from the sql.properties class into the sqlCache
   * properties object
   */
  private SQLCode() {
    try {
      String sqlFileName = "sql.properties";

      sqlCache.load(new FileInputStream(sqlFileName));
    } catch(IOException e) {
      System.out.println(e.toString());
    }
  }

  /*
   * The getSQLStatement() method will try to retrieve a SQL statement
   * from the SQLCache properties object based on the key passed into
   * the method.  If it can not find an the SQL statement, a DataAccess
   * Exception will be raised.
   */
  public String getSQLStatement(String pSQLKeyName)
      throws DataAccessException {

    //Checking to see if the requested SQL statement is in the sqlCache
```

```
        if (sqlCache.containsKey(pSQLKeyName)) {
          return (String) sqlCache.get(pSQLKeyName);
        } else{
          throw new DataAccessException("Unable to locate the SQL " +
              "statement requested in SQLCode.getSQLCode() " + pSQLKeyName);
        }
      }
    }
```

用ServiceLocator与SQLCode类进行抽象之后，findByPrimaryKey()方法其余部分执行SQL语句，并用title表中取得的数据建立TitleVO对象：

```
PreparedStatement preparedStatement = conn.prepareStatement(sql);
preparedStatement.setLong(1, Long.parseLong(pPrimaryKey));
rs = preparedStatement.executeQuery();

if (rs.next()){
   titleVO.setTitleId(rs.getLong("title_id"));
   ...
}
```

findByPrimaryKey()方法有几点需要记住。

第一，基础DAO接口定义一个findByPrimaryKey()方法，按主键取得记录。但是，DAO开发人员可以实现多个find()方法。DAO中实现的每个find()方法可以通过不同查询取得数据。如果从查询取得多个记录，则find()方法应返回包含ValueObjects的HashMap。HashMap中的每个ValueObject表示从find()方法所包装的查询取得的一个记录。

在TitleDAO.findByPrimaryKey()方法中，用EditorDAO.findEditorByTitle()方法取得对特定书名指定的所有编辑。

为了简单起见，本章不显示EditorDAO代码。但前面曾介绍过，可以从Wrox web site下载这个代码。

这个findEditorByTitle()方法返回的HashMap包含对这个书名指定的所有编辑。为什么用HashMap保存多个记录的结果呢？为什么不用ArrayList？之所以选择HashMap，是因为它可以用键值读取记录。在TitleDAO.findByPrimaryKey()方法中，调用EditorDAO.findByEditorTitle()方法返回的HashMap用每个编辑的主键作为键。

如果知道编辑ID，则可以直接从HashMap取得记录，而不必对映射中的每个项目进行迭代。如果用ArrayList或Vector之类的Collection对象，则在从集合中寻找记录时，没有这种灵活性。然后EditorDAO.findByPrimaryKey()方法返回的HashMap传入titleVO对象的setEditors()方法：

```
EditorDAOHome editorDAOHome =
  (EditorDAOHome) new InitialContext().lookup("editor/EditorDAO");
EditorDAO editorDAO = editorDAOHome.create();
```

```
titleVO.setEditors(editorDAO.findEditorByTitle
                    (new Long(titleVO.getTitleId()).toString()));
```

这个TitleDAO.findByPrimaryKey()方法演示了持久性框架如何处理一对多和多对多数据库关系。title表与editor表之间存在多对多关系。用户要取得TitleVO对象时，TitleDAO.findByPrimaryKey()方法还取得与这个书名相关联的所有编辑记录，并在返回最终用户的TitleVO对象中捕获这个关系。

如果findByPrimaryKey()方法没有找到数据，则发出NoDataFoundException。这个异常是从DataAccessException继承的，不一定表示DAO中有错，只是无法取得记录而已。

用EditorDAO.findEditorByTitle()方法取得与书名相关联的所有编辑记录时，要记住一点，这个方法调用可能成为性能瓶颈。例如，如果要读取1000个书名记录，则要1000次调用EditorDAO.findEditorByTitle()方法，这是持久性框架固有的基本弱点。

持久性框架设计成处理读取少量对象的小事务，可以进行修改与更新，而不适合读取大量数据集和进行批量插入、更新与删除。在findEditorByTitle()例子中，如果可能返回大量数据集，则要考虑是否对这个数据工作使用本章介绍的设计模式。

insert()方法

insert()方法的代码如下。下面看看insert()方法中发生的情况。insert()方法的第一步取得TitleDAO.insert()方法中要使用的SQL代码：

```
public void insert(ValueObject pValueObject)
    throws DataAccessException, ServiceLocatorException {

  //Getting all of my SQL code for inserting a title
  SQLCode sqlCode = SQLCode.getInstance();

  String sqlTitle = sqlCode.getSQLStatement("titledao.insert().insert");
  String sqlTitleEditor =
    sqlCode.getSQLStatement("titledao.insert().inserttitleeditor");
```

执行insert()方法时要使用两条SQL语句，第一条SQL语句将传入insert()方法的TitleVO中的书名数据插入title表，这个SQL语句如下，用键titledao.insert().insert表示：

```
INSERT INTO title (title_id, title_descr, title_cost, row_version )
VALUES ( ?,?,?,? )
```

取得SQL代码之后，从ServiceLocator取得数据库连接。然后用这个数据库连接取得PreparedStatement：

```
  //Building my SQL Code for inserting a record into the title table.
  TitleVO titleVO = (TitleVO) pValueObject;
  ServiceLocator serviceLocator = ServiceLocator.getInstance();
  Connection conn =
      serviceLocator.getDBConn(ServiceLocator.SUBSCRIPTIONDB);

  PreparedStatement preparedStatement = null;
```

```
try {
  //Populate the prepared statement with data from the value object
  preparedStatement = conn.prepareStatement(sqlTitle);
```

上述代码中还要注意，传入insert()方法中的ValueObject转换成TitleVO类型。TitleVO实例titleVO用来建立preparedStatement对象。建立preparedStatement对象之后，其执行INSERT语句：

```
preparedStatement.setLong(1, titleVO.getTitleId());
preparedStatement.setString(2, titleVO.getTitleDescr());
preparedStatement.setFloat(3, titleVO.getTitleCost());
preparedStatement.setLong(4,1);
preparedStatement.execute();
```

将记录插入title表之后，需要与editor表建立关系。insert()方法取得一个HashMap，包含要与这个书名记录相链接的所有编辑记录：

```
HashMap editorHash = titleVO.getEditors();
Collection editorCol = editorHash.values();
IteratoreditorIterator = editorCol.iterator();
```

然后它遍历editorIterator对象中的每个EditorVO对象，在title_editor表中增加项目：

```
while (editorIterator.hasNext()){
  EditorVO editorVO = (EditorVO) editorIterator.next();
  preparedStatement = conn.prepareStatement(sqlTitleEditor);
  preparedStatement.setLong(1,titleVO.getTitleId());
  preparedStatement.setLong(2,editorVO.getEditorId());
  preparedStatement.execute();
}
} catch(SQLException e) {
  //Aborting the transaction
  ctx.setRollbackOnly();
  throw new DataAccessException(
      "Error in TitleDAO.findByPrimaryKey()", e);
  } catch(RemoteException e) {
    //Aborting the transaction
    ctx.setRollbackOnly();
    throw new DataAccessException(
            "Error in TitleDAO.findByPrimaryKey()", e);
  } catch(CreateException e) {
    //Aborting the transaction
    ctx.setRollbackOnly();
    throw new DataAccessException(
            "Error in TitleDAO.findByPrimaryKey()", e);
  } finally {
    try {
```

```
            if (rs != null) rs.close();
            if (conn != null) conn.close();
          } catch(SQLException e) {
            System.out.println("I am unable to close a resultset, " +
                               "statement, or connection " +
                               "TitleDAO.findByPrimaryKey().");
          }
          return titleVO;
        }
      }
```

如果一切顺利，则执行代码的finally{}块，关闭preparedStatement和得到的数据库连接。如果插入期间遇到任何异常（如SQLException），则捕获异常并重新包装成DataAccessException。

看看insert()方法的代码，你可能提出两个问题：

- 记录主键如何产生，在哪里产生？
- 事务如何提交到数据库中？

title表中记录主键为title_id。对这个代码插入的值来自titleVO对象的getTitleId()方法。实际主键数值生成方法见本章稍后“主键生成策略”一节。

下列代码中没有发生数据库提交，这是因为TitleDAO会话EJB使用容器事务管理。应用程序服务器处理所有数据实现，代码惟一考虑的是在发生异常和撤销事务时通知应用程序。这是用ctx对象的setRollbackMethod()方法完成的。ctx对象是TitleDAO EJB的SessionContext。

```
ctx.setRollbackOnly();
```

事务管理见本章稍后“事务管理策略”一节。

createValueObject()方法

DAO实现中的createValueObject()方法创建ValueObject，可以填入数据，然后传入创建这个对象的DAO的insert()方法。createValueObject()方法与构造函数有许多相似之处，使DAO开发人员可以先正确初始化ValueObject之后再将其插入数据源中。

记住，在上面“insert()方法”一节中，要考虑记录的主键在哪里生成。答案是在createValueObject()方法中。TitleDAO的createValueObject()实现如下：

```
public ValueObject createValueObject()
    throws DataAccessException, ServiceLocatorException {

  TitleVO titleVO = new TitleVO();

  try {
     //Creating a counter instance. We will use this counter instance
     //to retrieve a primary key for the new record being used.
    ServiceLocator serviceLocator = ServiceLocator.getInstance();

    CounterHome counterHome = (CounterHome)
```

```
            serviceLocator.getEJBHome(ServiceLocator.COUNTER);
        Counter counter = counterHome.create();
        titleVO.setTitleId(counter.getNextVal("title_pk"));

        //Populating the rest of the value object with data
        titleVO.setTitleDescr("");
        titleVO.setTitleCost(0);
        titleVO.setRowVersion(1);
        titleVO.resetFlags();
      } catch(CreateException e){
        //Aborting the transaction
        ctx.setRollbackOnly();
        throw new DataAccessException(
                  "Error in TitleDAO.createValueObject()", e);
      } catch(RemoteException e){
        //Aborting the transaction
        ctx.setRollbackOnly();
        throw new DataAccessException(
                  "Error in TitleDAO.createValueObject()",e);
      } finally {
        return titleVO;
      }
    }
```

上述代码创建新的TitleDAO对象并预先填入数据。代码的有趣部分是显示新TitleDAO数值对象的主键如何生成。

要生成ValueObject的主键，就要创建调用Counter的会话EJB并调用其getNextVal()方法：

```
ServiceLocator serviceLocator = ServiceLocator.getInstance();

CounterHome counterHome = (CounterHome)
  serviceLocator.getEJBHome(ServiceLocator.COUNTER);
Counter counter = counterHome.create();
titleVO.setTitleId(counter.getNextVal("title_pk"));
```

Counter EJB将在“主键生成策略”一节详细介绍，对预订数据库生成所有主键。createValueObject()方法是一个完全独立于数据库的生成主键的机制。这很重要，因为几乎每个数据库平台都有自己的主键生成机制。Counter EJB分离主键生成方法与应用程序和持久性框架代码的其他部分。

将应用程序移到另一数据库平台时，只要在一个地方改变主键生成机制，而不必在多个地方改变。这个简单抽象可以在把DAO从一个数据库平台移到另一数据库平台时节省大量工作。

createValueObject()方法具有完全独立于数据库的主键生成机制，生成这些主键的Counter EJB仍然可能是与数据库相关的。

update()方法

TitleDAO的update()方法如下。这个方法用传入的ValueObject中包含的数据更新title表。

```
public void update(ValueObject pValueObject)
    throws DataAccessException, OptimisticLockException,
           ServiceLocatorException {

  //Getting my SQL Code for updating all of the titles
  SQLCode sqlCode = SQLCode.getInstance();
  String sql = sqlCode.getSQLStatement("titledao.update().update");

  TitleVO titleVO = (TitleVO) pValueObject;
  ServiceLocator serviceLocator = ServiceLocator.getInstance();
  Connection conn =
    serviceLocator.getDBConn(ServiceLocator.SUBSCRIPTIONDB);

  PreparedStatement preparedStatement = null;

  try {
    //Retrieving a row version number via the Counter session bean
    CounterHome counterHome =
      (CounterHome) serviceLocator.getEJBHome(ServiceLocator.COUNTER);
    Counter counter = counterHome.create();
    long nextRowVersion = counter.getNextVal("title_rv");

    //Populating the prepared statement's parameters
    preparedStatement = conn.prepareStatement(sql);
    preparedStatement.setString(1, titleVO.getTitleDescr());
    preparedStatement.setFloat( 2, titleVO.getTitleCost());
    preparedStatement.setLong( 3, nextRowVersion);
    preparedStatement.setLong( 4, titleVO.getTitleId());
    preparedStatement.setLong( 5, titleVO.getRowVersion());

    //Check to see if we were successful in updating the record.
    // If the queryResult does not equal 1, then we have run
    // into an optimistic lock situation.
    int queryResults = preparedStatement.executeUpdate();
    if (queryResults != 1) {
      ctx.setRollbackOnly();
      throw new OptimisticLockException("Stale data for title record:
                                        + titleVO.getTitleId());
    }
    //Cyclr through all of the editors and seeing if an editor
    //needs to be updated.
    Collection col = titleVO.getEditors().values();
    Iterator iterator = col.iterator();

    EditorDAOHome editorDAOHome = (EditorDAOHome)
```

```
            serviceLocator.getEJBHome(ServiceLocator.EDITORDAO);
        EditorDAO editorDAO = editorDAOHome.create();

        while (iterator.hasNext()) {
          EditorVO editorVO = (EditorVO) iterator.next();
          //If the updateFlag has been set to true then update the editor
          // vo record by invoking update on the editorDAO.
          if (editorVO.getUpdateFlag()) {
             editorDAO.update(editorVO);
          }
        }
      } catch(SQLException e) {
        //Aborting the transaction
        ctx.setRollbackOnly();
        throw new DataAccessException("Error in TitleDAO.update()", e);
      } catch(CreateException e) {
        //Aborting the transaction
        ctx.setRollbackOnly();
        throw new DataAccessException("Error in TitleDAO.update()", e);
      } catch(RemoteException e) {
        //Aborting the transaction
        ctx.setRollbackOnly();
        throw new DataAccessException("Error in TitleDAO.update()", e);
      } finally {
        try {
          if (preparedStatement!=null) preparedStatement.close();
          if (conn!=null) conn.close();
        } catch(SQLException e) {
          System.out.println("I am unable to close a resultset, " +
                             "statement, or connection TitleDAO.update().");
        }
      }
    }
```

update()方法的代码首先设置更新title表的SQL代码，取得预订数据库的JDBC连接：

```
SQLCode sqlCode = SQLCode.getInstance();
String sql = sqlCode.getSQLStatement("titledao.update().update");

TitleVO titleVO = (TitleVO) pValueObject;

ServiceLocator serviceLocator = ServiceLocator.getInstance();
Connection conn =
    serviceLocator.getDBConn(ServiceLocator.SUBSCRIPTIONDB);
```

然后update()方法用Counter EJB取得更新目标title表记录时要用的下一行版本号：

```
CounterHome counterHome =
```

```
        (CounterHome) serviceLocator.getEJBHome(ServiceLocator.COUNTER);
    Counter counter = counterHome.create();
    long nextRowVersion = counter.getNextVal("title_rv");
```

update()方法用开放锁策略处理多个用户同时更新同一数据库记录的问题。开放锁策略不在用户从数据库中取记录时对记录采用显式数据库锁，而是让每个用户取得同一记录的拷贝。记录被标上版本，用户要更新数据库中的记录时，检查这个版本号，保证用户要更新的数据没有被另一用户更新。

可以用几种不同方法实现开放锁策略。本章的开放锁策略要求数据库中每个表有一个row_version列。

row_version列包含一个数字，表示行的版本号。在本章建立的持久性框架中，从数据库中取记录时，将其读取到ValueObject中。部分数据放进ValueObject中时，读取row_version。

用户要实现对记录所作的任何改变时，将包含数据的ValueObject传入update()方法。然后update()方法用row_version构造UPDATE语句的WHERE从句。TitleDAO EJB中使用的UPDATE语句如下：

```
UPDATE title SET title_descr=? , title_cost=?, row_version=? WHERE title_id=? AND
row_version=?
```

在上述UPDATE语句中，row_version在UPDATE SQL语句中被更新。更新行版本号表示这个行已经修改，使用这个记录的其他用户用的是过时数据。Counter EJB对title表中的每一行生成一个row_version号。Subscription数据库中的每个表有一个项目放在counter表中，跟踪这个表的当前行版本号。

产生行版本号之后，用传入的TitleVO对象中包含的数据和新生成的行版本号建立准备语句，然后执行这个preparedStatement：

```
    preparedStatement = conn.prepareStatement(sql);
    preparedStatement.setString(1, titleVO.getTitleDescr());
    preparedStatement.setFloat( 2, titleVO.getTitleCost());
    preparedStatement.setLong( 3, nextRowVersion);
    preparedStatement.setLong( 4, titleVO.getTitleId());
    preparedStatement.setLong( 5, titleVO.getRowVersion());
```

如果两个用户要同时更新title表中的同一记录，会发生什么情形呢？两个用户通过从TitleDAO的find()方法取得的ValueObject收到相同的记录版本。用户一到TitleDAO EJB调用UPDATE方法时，发出一个更新。用户一的记录顺利更新数据库，因为UPDATE语句会找到数据库匹配记录。在更新记录过程中，用户所更新记录的row_version号会改变。

用户二的数据已经过时。这个用户要用TitleDAO.update()方法更新数据库时，会失败，因为UPDATE语句无法找到title_id与row_version匹配update()方法所得到ValueObject的记录。update()方法通过检查UPDATE语句更新的行号判断更新是否顺利完成。

```
    int queryResults = preparedStatement.executeUpdate();
    if (queryResults != 1) {
      throw new OptimisticLockException("Stale data for title record: "
```

```
                                        + titleVO.getTitleId());
    }
```

如果UPDATE SQL调用返回的行数不等于1，则更新title表的用户知道自己持有过时数据。

如果发生开放锁情形，则抛出OptimisticLockException。OptimisticLockException是从DataAccessException扩展的用户定义异常。OptimisticLockException向调用更新方法的应用程序抛出，使其可以确定如何处理数据。如果使用DAO的应用程序是个用户界面，则最终用户可能看到要处理的数据已经发生某种改变，需要再次提交改变。

本章“并发性策略”一节将介绍开放锁策略的不同实现方法。

更新title表中的记录之后，update()方法创建EditorDAO EJB：

```
EditorDAOHome editorDAOHome = (EditorDAOHome)
    serviceLocator.getEJBHome(ServiceLocator.EDITORDAO);
EditorDAO editorDAO = editorDAOHome.create();
```

创建EditorDAO EJB之后，update()方法遍历与TitleVO相关联的每个EditorVO对象。如果EditorVO的update标志设置为true（由EditorVO.getUpdateFlag()方法确定），则更新EditorVO的数据，将这个EditorVO传入EditorDAO.update()方法。

```
while (iterator.hasNext()) {
  EditorVO editorVO = (EditorVO) iterator.next();
  if (editorVO.getUpdateFlag()){
     editorDAO.update(editorVO);
  }
}
```

DAO要让用户更新、插入或删除传入DAO的ValueObjects中包含的ValueObject时，使用insert, update与delete标志。对于TitleDAO，我们使应用程序不仅能够更新TitleVO，而且可以更新这个TitleVO中包含的任何EditorVO。用户用TitleVO包含的HashMap读取这个EditorVO，改变任何要改变的值，然后将其update标志设置为true。然后对TitleDAO调用update()方法时，在Subscription数据库中也更新EditorVO。

delete()方法

delete()方法不仅删除title表中的记录，而且删除title_editor表中与目标书名记录相关联的任何记录。这样可以保证title与title_editor表之间的引用完整性。

```
public void delete(ValueObject pValueObject)
    throws DataAccessException, ServiceLocatorException {

  //Build the SQL Code for deleting a record from the title table.
  SQLCode sqlCode = SQLCode.getInstance();
  String sqlTitle =
    sqlCode.getSQLStatement("titledao.delete().deletetitle");
```

```
String sqlTitleEditor =
  sqlCode.getSQLStatement("titledao.delete().deletetitleeditor");

TitleVO titleVO = (TitleVO) pValueObject;

ServiceLocator serviceLocator = ServiceLocator.getInstance();
Connection conn =
  serviceLocator.getDBConn(ServiceLocator.SUBSCRIPTIONDB);

PreparedStatement preparedStatement = null;

try {
  //Delete the row from the title table
  preparedStatement = conn.prepareStatement(sqlTitle);
  preparedStatement.setLong(1, titleVO.getTitleId());
  preparedStatement.executeQuery();

  //Delete all rows from the title_editor table that match the
  //title_id
  preparedStatement = conn.prepareStatement(sqlTitleEditor);
  preparedStatement.setLong(1, titleVO.getTitleId());
  preparedStatement.executeQuery();
} catch(SQLException e) {
  //Aborting the transaction
  ctx.setRollbackOnly();
  throw new DataAccessException("Error in TitleDAO.delete()", e);
} finally{
  try {
    if (preparedStatement!=null) preparedStatement.close();
    if (conn!=null)       conn.close();
  } catch(SQLException e) {
    System.out.println("I am unable to close a resultset, " +
                       "statement, or connection. TitleDAO.delete()");
  }
}
```

DAO与关系

捕获与建模持久性框架中实体之间的关系是比较难的工作。在TitleDAO代码中，TitleVO与EditorVO之间有多对多关系。持久性框架开发人员要确定应用程序修改框架中的数据实体和数据关系时需要有多大控制权。

在本章的实现中，TitleDAO只管理title表与editor表之间的关系，不让开发人员增加editor表中没有的新编辑，也不能通过标为删除而删除editor表中的编辑（即在EditorVO对象中把deleteFlag设置为true）。TitleDAO的惟一工作是：

- 在插入新的书名记录时将项目加进title_editor表，遍历TitleVO的editors HashMap并将每个EditorVO的项目加进HashMap。但是，代码假设editor表中已经有编辑记录。

- 在title表中删除现有书名时从title_editor表中删除项目。
- 允许更新传入TitleDAO.update()方法的TitleVO中包含的EditorVO对象。

持久性框架中的核心设计决策之一是DAO控制与管理属于数据库中另一数据实体的记录的程度。

管理数据关系不仅是个编码问题，而是个数据管理问题。必须认真考虑DAO对直接范围之外的记录增加、更新与删除提供多大控制，如允许TitleDAO更新editor表中的记录。在代码中管理数据关系是相当困难的，如果代码写得不好，则可能搞乱数据库，破坏数据库中数据实体之间的引用完整性。

Value Object模式

Value Object模式首先作为业务层J2EE设计模式，减少使用实体EJB时需要的远程调用次数。EJB 2.0规范推出之前，客户机调用JEB方法时，发生远程调用。这个远程调用会产生大量网络开销，即使调用EJB的客户机与EJB在同一应用程序服务器中。

Value Object模式可以减少网络开销，让实体EJB在普通Java类中填入用户请求的所有数据。然后把数值对象发送回调用实体Bean的应用程序。调用应用程序处理数值对象，然后将其返回原先的实体。让调用应用程序而不让EJB取得和设置数值对象的数据可以大大减少使用实体EJB造成的潜在网络通信流量。下一章将介绍Value Object模式的这个用法。

但是，可以扩展Value Object模式，在持久性框架更好地利用。甚至可以把Value Object模式看成核心数据层模式，原因有两个：

- 首先，数值对象可以表示数据库中所包含数据的逻辑视图。可以用Value Object模式抽象数据关系（如实体间为一对多关系或多对多关系）。数值对象甚至可以用一个界面表示从多个数据源读取的数据。就使用数值对象的客户机应用程序而言，并不知道数据来自哪里，如何管理。
- 第二，数值对象把几层体系结构的所有层集成起来。可以用数值对象在表示层、业务逻辑层和数据层之间来回传递数据。用户在用户界面中键入数据时，可以把这个数据放在数值对象中，传入业务逻辑，然后转发到数据层。相反，用户请求信息时，数据层可以返回数值对象（一个或一组）到业务逻辑层，然后转发到表示层。

本章持久性框架中的所有数值对象都是从抽象类ValueObject派生的，其代码如下：

```
package valueobject;

import java.io.Serializable;

/*
 * The ValueObject is the abstract class that all ValueObjects in our
 * data persistence framework descend off of. It provides a number of
 * methods, including get/set methods for the VO status flags and the
 * rowVersion method.
 */
public abstract class ValueObject implements Serializable {
```

```
/*Below are the VO status flags*/
private boolean insertFlag = false;
private boolean updateFlag = false;
private boolean deleteFlag = false;
private long    rowVersion = 0;

/*
 * Retrieves the insertFlag. The insertFlag is used to tell the DAO that
 * data in the ValueObject should be inserted into the database.
 */
public boolean getInsertFlag() {
  return insertFlag;
}

/*
 * Retrieves the deleteFlag. The deleteFlag is used to tell the DAO that
 * the data in the ValueObject should be deleted from the database.
 */
public boolean getDeleteFlag() {
  return deleteFlag;
}

/*
 * Retrieves the updateFlag. The updateFlag is to tell the DAO that the
 * data in the ValueObject should be updated in the database.
 *
public boolean getUpdateFlag() {
  return updateFlag;
}

/*
 * Retrieves the rowVersion. The rowVersion holds the current rowVersion
 * of the record and is used by the DAO.update() method to enforce
 * optimistic locking exceptions.
 */
public long getRowVersion() {
  return rowVersion;
}
/*Sets the insertFlag and sets all other status flags to false*/
public void setInsertFlag(boolean pFlag) {
  insertFlag = pFlag;
  deleteFlag = false;
  updateFlag = false;
}

/*Sets the updateFlag and sets all other status flags to false*/
public void setUpdateFlag(boolean pFlag) {
```

```
    insertFlag = false;
    deleteFlag = false;
    updateFlag = pFlag;
  }

  /*Sets the deleteFlag and sets all other status flags to false*/
  public void setDeleteFlag(boolean pFlag) {
    insertFlag = false;
    deleteFlag = pFlag;
    updateFlag = false;
  }

  /*Sets the rowVersion property*/
  public void setRowVersion(long pRowVersion) {
    rowVersion = pRowVersion;
  }

  /*Resets all VO status flags to false*/
  public void resetFlags() {
    insertFlag = false;
    deleteFlag = false;
    updateFlag = false;
  }
}
```

ValueObject类的三个状态属性有自己的get()或set()方法：

- insertFlag
- updateFlag
- deleteFlag

这三个标志显示DAO要如何处理数值对象。本章前面曾介绍过TitleDAO bean如何在调用TitleDAO.update()方法时更新editor表。

记住，在TitleDAO.update()方法中，更新标志向TitleDAO显示如何处理TitleVO中包含的EditorVO。TitleDAO只在设置EditorVO的更新标志时才更新编辑记录。

任何时候只能设置其中一个标志。对一个状态属性调用set方法时，自动将其他状态属性设置为false。例如，调用setInsertFlag()方法时，记录的insertFlag设置为调用应用程序传入的值，updateFlag与deleteFlag自动设置为false：

```
public void setInsertFlag(boolean pFlag) {
  insertFlag = pFlag;
  deleteFlag = false;
  updateFlag = false;
}
```

ValueObject类还提供一个reset()方法。这个方法自动将插入、更新与删除状态标志值重新设置为false。

稍后将会介绍，每次调用任何set方法时，TitleVO自动将updateFlag设置为true。DAO首次创建ValueObject和填入数据时，这种行为是有问题的。在ValueObject返回最终用户之前，更新状态已经设置为true。可以用reset()方法取这个建好的ValueObject，保证将其所有状态标志值复位为false：

```
public void resetFlags(){
    insertFlag = false;
    deleteFlag = false;
    updateFlag = false;
  }
```

最后，ValueObject类对rowVersion属性提供get()与set()方法。rowVersion属性放在ValueObject类中，保证持久性框架中的每个ValueObject参与框架的开放锁策略。

我们已经从抽象层介绍ValueObject类，下面看看TitleVO数值对象的具体例子。

TitleVO数值对象

TitleVO类返回与title表相关联的所有信息，并通过getEditors()或setEditors()方法捕获title表与editor表之间的关系。但是，TitleVO类不向应用程序显示title与editor表之间的关系，这是个重要抽象，因为title与editor表之间的关系是可以改变的。这个改变要在TitleDAO中更新，但TitleVO不需要改变。这样就使利用TitleVO中数据的应用程序随基础数据库改变而改变。

应用程序代码只要知道，调用TitleVO.getEditors()方法即可找到与书名记录相关联的所有编辑记录。这是个重要抽象，特别是在title表与editor表之间的关系需要重建时。

```
package valueobject;

import java.util.HashMap;

/*
 * The TitleVO class represents data retrieved from the ValueObject.
 * It is an implementation of the ValueObject pattern.
*/
public class TitleVO extends ValueObject {
  private long   titleId;
  private String titleDescr;
  private float  titleCost;
  private HashMap editors;

  /*Constructor: Initializes the properties of the class to be empty*/
  public TitleVO() {
    titleId     = 0;
    titleDescr  = "";
    titleCost   = 0;
    editors     = new HashMap();
  }

  /*Retrieves the title id*/
```

```
    public long getTitleId() {
      return titleId;
    }

    /*Retrieves the title description*/
    public String getTitleDescr() {
      return titleDescr;
    }

    /*Retrieves the title cost*/
    public float getTitleCost() {
      return titleCost;
    }

    /*Returns all the editors associated with a title*/
    public HashMap getEditors() {
      return editors;
    }

    /*Sets the title id*/
    public void setTitleId(long pTitleId) {
      setUpdateFlag(true);
      titleId = pTitleId;
    }

    /*Sets the title descr*/
    public void setTitleDescr(String pTitleDescr) {
      setUpdateFlag(true);
      titleDescr = pTitleDescr;
    }

    /*Sets the title cost*/
    public void setTitleCost(float pTitleCost) {
      setUpdateFlag(true);
      titleCost = pTitleCost;
    }
    /*Sets a hashmap with all of the editors*/
    public void setEditors(HashMap pEditors) {
      setUpdateFlag(true);
      editors = pEditors;
    }
  }
```

设计数值对象层时，注意下面几点：

- **数值对象应是轻量级的**

  数值对象包装进入和离开数据库的数据。因此数值对象实现中应当很少有代码或没有代码。如果数值对象中放了业务逻辑，则应考虑修改数值对象，除去这些逻辑。这些逻辑最好放到业务层或DAO的数据库中。

- **数值对象是数据库中数据的视图**

  用DAO的find()方法返回数值对象的不同变形或视图是没问题的，特别是在DAO为多种应用程序服务时。例如，一个find()方法返回的TitleVO完全用编辑信息建立，而另一个find()方法返回的TitleVO的HashMap不用编辑信息建立。甚至可以用一个DAO返回完全不同类型的数值对象。

- **使数值对象的对象层次保持简单**

  如果要向应用程序返回复杂对象层次，则要在业务逻辑层进行。让业务对象取得几个不同DAO中的数值对象，并将其组织成层次，这样可以大大减少DAO中所开发的数据库代码的复杂性。

- **注意持久性框架中的数值对象个数**

  开发人员的一个常见毛病是数值对象太细，结果出现太多数值对象，很难使用与维护。

Service Locator模式

在本章的例子中，我们介绍了几种情形，取得EJB与JDBC数据库连接，在持久性框架中使用。这两个项目用Java命名目录接口（JNDI，Java Naming Directory Interfaces）API取得。JNDI的查找方法随不同厂家而不同。此外，重复JNDI查找的成本很高。

每次用户需要从JNDI取得服务时，要填入适当的JNDI环境信息，建立与JNDI服务器的连接，然后查找服务。需要单点创建和取得各种JNDI服务，这时可以使用Service Locator模式。

Service Locator模式抽象简单界面后面的所有JNDI查找，本章使用的Service Locator实现可以读取EJBHome接口和JDBC数据库连接。此外，这个Service Locator通过缓存EJBHome接口和DataSource对象减少所需的JNDI查找次数，在用户首次请求特定EJB与JDBC数据库连接时将其缓存到创建的JDBC连接。

Service Locator模式的代码如下：

```
package dataaccess;

import java.sql.*;
import javax.sql.DataSource;
import java.util.Hashtable;
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.PortableRemoteObject;
import session.counter.CounterHome;
import session.title.TitleDAOHome;
import session.editor.EditorDAOHome;

/*
 *  The Service Locator pattern is abstracts away the JNDI
 *  logic necessary for retrieving a JDBC Connection or EJBHome
```

```
 *  interface
 */
public class ServiceLocator{
  private static ServiceLocator serviceLocatorRef = null;
  private static Hashtable      ejbHomeCache      = null;
  private static Hashtable      dataSourceCache   = null;

  /*Enumerating the different services available from the
   * Service Locator
   */
  public static final int COUNTER        = 0;
  public static final int TITLEDAO       = 1;
  public static final int EDITORDAO      = 2;
  public static final int SUBSCRIPTIONDB = 3;

  /*The JNDI Names used to lookup a service*/
  private static final String COUNTER_JNDINAME="counter/Counter";
  private static final String TITLEDAO_JNDINAME="title/TitleDAO";
  private static final String EDITORDAO_JNDINAME="editor/EditorDAO";
  private static final String
     SUBSCRIPTIONDB_JNDINAME="java:/subscriptionDS";

  /*References to each of the different EJB Home Interfaces*/
  private static final Class COUNTERCLASSREF = CounterHome.class;
  private static final Class TITLECLASSREF   = TitleDAOHome.class;
  private static final Class EDITORCLASSREF  = EditorDAOHome.class;

  static {
    serviceLocatorRef = new ServiceLocator();
  }

  /*Private Constructor for the ServiceLocator*/
  private ServiceLocator(){
    ejbHomeCache    = new Hashtable();
    dataSourceCache = new Hashtable();
  }
  /*
   * The ServiceLocator is implemented as a Singleton. The getInstance()
   * method will return the static reference to the ServiceLocator stored
   * inside of the ServiceLocator Class.
   */
  public static ServiceLocator getInstance(){
    return serviceLocatorRef;
  }

  /*
   * The getServiceName will retrieve the JNDI name for a requested
   * service.  The service is indicated by the ServiceId passed into
```

```
 * the method.
 */
static private String getServiceName(int pServiceId)
    throws ServiceLocatorException {

  String serviceName = null;

  switch (pServiceId) {
    case COUNTER:
        serviceName = COUNTER_JNDINAME;
        break;
    case TITLEDAO:
        serviceName = TITLEDAO_JNDINAME;
        break;
    case EDITORDAO:
        serviceName = EDITORDAO_JNDINAME;
        break;
    case SUBSCRIPTIONDB:
        serviceName = SUBSCRIPTIONDB_JNDINAME;
        break;
    default:
        throw new ServiceLocatorException(
                "Unable to locate the service requested in " +
                 "ServiceLocator.getServiceName() method.  ");
  }
  return serviceName;
}

/*
 * Returns the EJBHome Class reference for a requested service.
 * If the method can not make a match, it will throw a
 * ServiceLocatorException.
 */
static private Class getEJBHomeRef(int pServiceId)
    throws ServiceLocatorException {
  Class homeRef = null;

  switch (pServiceId){
    case COUNTER:
        homeRef = COUNTERCLASSREF;
        break;
    case TITLEDAO:
        homeRef = TITLECLASSREF;
        break;
    case EDITORDAO:
        homeRef = EDITORCLASSREF;
        break;
```

```
        default:
            throw new ServiceLocatorException(
                    "Unable to locate the service requested in " +
                    "ServiceLocator.getEJBHomeRef() method.  ");
      }
      return homeRef;
    }

    /*
     * The getEJBHome method will return an EJBHome interface for a
     * requested service.  If it can not find the requested EJB, it will
     * throw a servicelocator exception.
     *
     * The getEJBHome interface caches a requested EJBHome so that the first
     * time an EJB is requested, a home interface will be retrieved but then
     * be placed into a cache.
     */
    public EJBHome getEJBHome(int pServiceId)
        throws ServiceLocatorException {
      //Trying to find the JNDI Name for the requested service
      String serviceName = getServiceName(pServiceId);
      EJBHome ejbHome    = null;

      try {
        //Checking to see if I can find the EJBHome interface in cache
        if (ejbHomeCache.containsKey(serviceName)) {
          ejbHome = (EJBHome) ejbHomeCache.get(serviceName);
          return ejbHome;
        } else {
          //If I could not find the EJBHome interface in the cache, look it
          // up and then cache it.
          Context ctx = new InitialContext();
          Object jndiRef = ctx.lookup(serviceName);

          Object portableObj =
            PortableRemoteObject.narrow(jndiRef, getEJBHomeRef(pServiceId));

          ejbHome = (EJBHome) portableObj;

          ejbHomeCache.put(serviceName, ejbHome);
          return ejbHome;
        }
      } catch(NamingException e) {
        throw new ServiceLocatorException(
              "Naming exception error in ServiceLocator.getEJBHome()" ,e);
      } catch(Exception e) {
        throw new ServiceLocatorException(
```

```
                "General exception in ServiceLocator.getEJBHome",e);
        }
    }

    /*
     * The getDBConn() method will create a JDBC connection for the
     * requested database.  It too uses a cachin algorithm to minimize
     * the number of JNDI hits that it must perform.
     */
    public Connection getDBConn(int pServiceId)
        throws ServiceLocatorException {
      //Getting the JNDI Service Name
      String     serviceName = getServiceName(pServiceId);
      Connection conn        = null;

      try {
        //Checking to see if the requested DataSource is in the Cache
        if (dataSourceCache.containsKey(serviceName)) {
           DataSource ds = (DataSource) dataSourceCache.get(serviceName);
           conn = ((DataSource)ds).getConnection();
           return conn;
        } else {
          // The DataSource was not in the cache.  Retrieve it from JNDI
          // and put it in the cache.
          Context ctx = new InitialContext();
          DataSource newDataSource = (DataSource) ctx.lookup(serviceName);
          dataSourceCache.put(serviceName, newDataSource);
          conn = newDataSource.getConnection();
          return conn;
        }
      } catch(SQLException e) {
        throw new ServiceLocatorException(
              "A SQL error has occurred in ServiceLocator.getDBConn()", e);
      } catch(NamingException e) {
        throw new ServiceLocatorException(
              "A JNDI Naming exception has occurred "+
              " in ServiceLocator.getDBConn()", e);
      } catch(Exception e) {
        throw new ServiceLocatorException(
              "An exception has occurred in ServiceLocator.getDBConn()", e);
      }
    }
}
```

下面看看**getEJBHome()**方法，这个方法从**ServiceLocator**取得**EJBHome**接口：

```
public EJBHome getEJBHome(int pServiceId)
    throws ServiceLocatorException {
```

用户需要特定EJB时，调用这个方法，传入服务ID。这些服务ID定义为ServiceLocator类中的static final int变量：

```
public static final int COUNTER        = 0;
public static final int TITLEDAO       = 1;
public static final int EDITORDAO      = 2;
public static final int SUBSCRIPTIONDB = 3;
```

其中一个值传入getEJBHome()方法时，转换成JNDI名称。JNDI名称是传入JNDI情境的lookup()方法，从命名目录中取得一个项目。将服务ID转换成JNDI名称是用getServiceName()方法完成的。

```
String serviceName = getServiceName(pServiceId);
```

getServiceName()方法就是一个switch/case语句，匹配JNDI名称与传入的服务ID。如果找不到匹配，则getServiceName()方法抛出ServiceLocatorException。这个例外在ServiceLocator中发生异常条件时发出。

取得JNDI名称之后，检查缓冲区，看看是否已经有从JNDI取得的EJBHome接口。这个ServiceLocator实现的缓存区是个Hashtable，用EJB的JNDI名称作为键：

```
if (ejbHomeCache.containsKey(serviceName)){
   ejbHome = (EJBHome) ejbHomeCache.get(serviceName);
   return ejbHome;
}
```

如果缓冲区找到匹配，则从缓冲区中取得EJBHome接口并返回调用应用程序。如果缓冲区中找不到EJBHome，则进行JNDI查找：

```
} else {
  Context ctx = new InitialContext();
  Object jndiRef = ctx.lookup(serviceName);
```

上述ctx.lookup()方法返回通过JNDI所查到项目的引用。然后代码将JNDI查找返回的抽象接口转换成适当类型。这是用PortableRemoteObject.narrow()调用完成的，传入要检查的项目，转换成Class类型：

```
Object portableObj =
  PortableRemoteObject.narrow(jndiRef, getEJBHomeRef(pServiceId));
```

传入narrow()方法的实际Class类型用getEJBHomeRef()调用取得。这个方法取得请求的特定EJB的类引用。getEJBHomeRef()方法和getServiceName()方法一样用switch/case语句映射服务名与Class类型。

取得EJBHome之后，将其放进缓冲区，然后返回请求应用程序：

```
ejbHomeCache.put(serviceName, ejbHome);
```

```
return ejbHome;
```

getDBConn()方法与getEJBHome()方法相似，用JNDI取得一个DataSource对象，表示用户请求的数据库连接。然后它用这个DataSource创建数据库连接。和getEJBHome()方法相似，getDBConn()方法首次从JNDI取得DataSource对象时，将其缓存。缓存DataSource之后，任何请求从缓冲区取得DataSource，而不是进行新的JNDI查找。

最后要注意一点，这个ServiceLocator实现采用Singleton模式。从本章的所有例子可以看出，用户并不用构造函数创建ServiceLocator，而是用ServiceLocator的getInstance()方法取得ServiceLocator类中存放的静态ServiceLocator的引用。

```
public static ServiceLocator getInstance() {
  return serviceLocatorRef;
}
```

ServiceLocator实现采用Singleton模式是相当有效的，因为这个类大量使用，不需要维护实例信息。

在ServiceLocator中使用非final静态方法实际上不符合EJB规范。但我们不管这一点，因为我们的缓冲区不要求在不同Java虚拟机上一致，而这是上述限制的原因。还要验证工厂类可以不加同步而同时在多个线程中使用，这个问题在规范中没有解决。这两个问题使我们的方案虽然能顺利工作，但不容易移植。为了提高移植性，可以牺牲一些性能，在Bean实例中进行缓存。

使用持久性框架

下面举两个例子说明如何利用已经建立的数据持久性框架读取和插入title数据。这些例子包括两个JSP页面：

- selectExample.jsp
  selectExample.jsp显示如何从TitleDAO.find()方法取一个书名记录，并显示如何用从TitleDAO取得的TitleVO对象的getEditors()方法取得TitleVO的编辑信息。
- insertExample.jsp
  insertExample.jsp演示如何用TitleDAO在title表中插入一个记录，并用EditorDAO取得EditorVO，然后可以和要插入的title记录相关联。

不能让JSP页面直接访问DAO层，而要用Session Façade代表JSP页面访问DAO。但是，为了演示DAO，建立这一层会大大增加演示用DAO插入与读取数据所需的代码量。此外，Session Façade与下一章关系更密切，因此放在下一章介绍。

关于如何正确地分出表示层、业务层和数据层，见本章末尾“移植策略”一节。

selectExample.jsp页面

```
<%@page import="java.util.*"%>
```

```
<%@page import="javax.naming.*"%>
<%@page import="dataaccess.*"%>
<%@page import="valueobject.*"%>
<%@page import="session.title.*"%>

<!-
    Remember this JSP is for example purposes only.
    In a real-world application you should never allow a JSP to directly
    access the DAO.
-->
<H1>Example 1 - Using a DAO to retrieve a title</H1>
<BR/>
<BR/>
<P>
   This JSP page will retrieve a single record from the Title table using
   a TitleDAO <BR/>
   and TitleVO object. It will also display all of the editors associated
   with this <BR/>
   Title by calling the getEditors() method on the TitleVO.
</P>
<BR/>
<%
   try {
```

selectExample页面从title表取一个记录，为此要取一个TitleDAO EJB：

```
ServiceLocator serviceLocator = ServiceLocator.getInstance();
TitleDAOHome titleDAOHome = (TitleDAOHome)
            serviceLocator.getEJBHome(ServiceLocator.TITLEDAO);

TitleDAO titleDAO = titleDAOHome.create();
```

要取得TitleDAO EJB又称titleDAO，用ServiceLocator寻找TitleDAO的EJBHome接口，然后用EJBHome接口取得TitleDAO的远程接口。

取得TitleDAO EJB之后，调用DAO的find()方法，传入所取记录的主键：

```
/*Calling the find method and retrieving a Title by title ID.*/
TitleVO titleVO  = (TitleVO) titleDAO.findByPrimaryKey("1");
```

通过TitleDAO.find()方法取得TitleVO之后，用它打印请求的书名记录信息：

```
out.println("<P>Title id    : " + titleVO.getTitleId()    + "<br/>");
out.println("<P>Title descr : " + titleVO.getTitleDescr() + "<br/>");
out.println("<P>Title cost  : " + titleVO.getTitleCost()  + "<br/>");

out.println("<P>Editor(s) Working on the Title  : <br/>");
```

记住，书名与编辑之间具有多对多关系。可以用titleVO的getEditors()方法调用与书名记录相关联的编辑：

```
        /*
         * Retrieving the editors that the user currently has.  We will
         * then grab an iterator off of the HashMap and then display the
         * names of each editor.
        */
        HashMap hashMap = titleVO.getEditors();
```

从getEditors()方法取得HashMap之后，很容易对HashMap进行循环，取得每个编辑记录的EditorVO信息。每个取得的EditorVO对象可以用来查询编辑姓名：

```
        Collection col = hashMap.values();

        Iterator iterator = col.iterator();

        while (iterator.hasNext()) {
          EditorVO editorVO = (EditorVO) iterator.next();

          out.println(editorVO.getFirstName() + " " +
                      editorVO.getMiddleName() + " " +
                      editorVO.getLastName() + "<br/>");
        }
      } catch(Exception e) {
        out.println("error--> " + e.getMessage());
      }
%>
```

本书的JSP页面例子很简单，但能说明问题。通过恰当使用持久性设计模式，可以完全分离JSP页面中的预订数据库信息。selectExample.jsp页面不知道书名数据来自哪里，如何读取。

顺利运行selectExample.jsp页面时，可以看到如图3.4所示的结果。

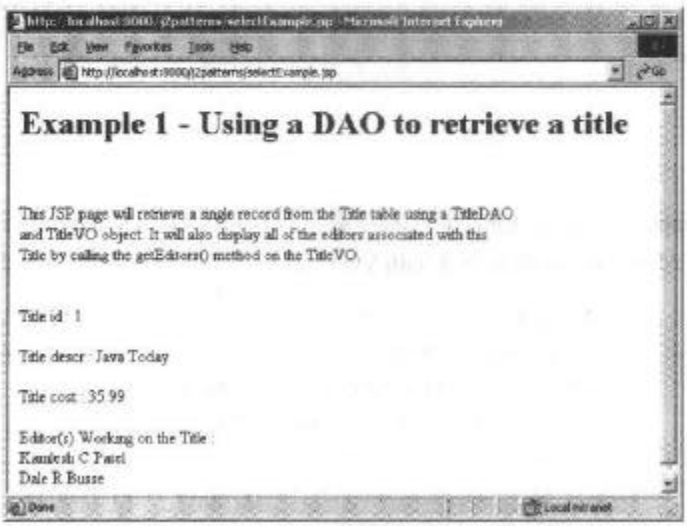

图3.4

insertExample.jsp页面

insertExample.jsp页面建立在**selectExample.jsp**页面的基础上，用**TitleDAO**在**title**表中插入新记录。在插入新记录的过程中，**insertExample.jsp**页面还用**EditorDAO**取得编辑，然后与插入的书名记录相关联。**insertExample.jsp**页面的代码如下：

```
<%@page import="java.util.*"%>
<%@page import="javax.naming.*"%>
<%@page import="javax.transaction.*"%>
<%@page import="dataaccess.ServiceLocator"%>
<%@page import="valueobject.*"%>
<%@page import="session.title.*"%>
<%@page import="session.editor.*"%>

<!--
    Remember this JSP is for example purposes only.  In a real-world
    Application you should never allow a JSP to directly access the DAO.
    You would instead access the DAO from a session EJB.
-->

<H1>Example 2 - Using a DAO to insert a title</H1>
<BR/>
<BR/>
<P>
   This JSP page will insert a single record from the Title table using a
   TitleDAO <BR/>
   and TitleVO object.
   It will also establish relationships between the title <BR/>
   and the editor by retrieving 3 editorVO's and then adding them to the
   HashMap returned <BR/>
   by the getEditors() method on the TitleVO.<BR/>
</P>
<BR/>

<%
   try {
```

insertExample.jsp页面首先取得应用程序服务器中的**TitleDAO**与**EditorDAO**，这两个DAO用来建立要插入**subscription**数据库的**TitleVO**对象。

```
/*Getting a TitleDAO for manipulating title information*/
ServiceLocator serviceLocator = ServiceLocator.getInstance();
TitleDAOHome titleDAOHome = (TitleDAOHome)
            serviceLocator.getEJBHome(ServiceLocator.TITLEDAO);
TitleDAO titleDAO = titleDAOHome.create();

/*Getting an EditorDAO to retrieve Editor information*/
EditorDAOHome editorDAOHome = (EditorDAOHome)
```

```
                serviceLocator.getEJBHome(ServiceLocator.EDITORDAO);
        EditorDAO editorDAO = editorDAOHome.create();
```

创建这两个DAO之后，从subscription数据库取得三个编辑记录（后面在insertExample.jsp页面代码中要将这些记录与创建的书名记录相关联）。

```
/*
 * Looking up three editors.  I am going to associate these editors
 * to the new title record
 */
EditorVO editorVO1 = (EditorVO) editorDAO.findByPrimaryKey("1");
EditorVO editorVO2 = (EditorVO) editorDAO.findByPrimaryKey("2");
EditorVO editorVO3 = (EditorVO) editorDAO.findByPrimaryKey("3");
```

现在要开始实质性工作了。titleDAO EJB的createValueObject()方法创建新的TitleVO对象newTitleVO。

```
/*Creating a new title vo by calling the TitleVO's
 * createValueObjectMethod
 */
TitleVO newTitleVO = (TitleVO) titleDAO.createValueObject();
```

NewTitleVO对象是个空白TitleVO对象，这个对象设置的惟一属性是titleId，这个属性保存createValueObject()方法中生成的主键值。如果你还记得，TitleDAO.createValueObject()方法用Counter EJB生成主键。

创建newTitleVO之后，JSP页面在对象中填入数据。

```
/*Populate the new TitleVO with data*/
newTitleVO.setTitleDescr("Geek World");
newTitleVO.setTitleCost(100);
```

JSP页面还与前面用EditorDAO取得的三个EditorVO相关。取得的每个EditorVO加进newTitleVO对象中存储的HashMap内。

```
/*Add the editors I retrieved from the EditorDAO to the TitleVO*/
newTitleVO.getEditors().put(
  new Long(editorVO1.getEditorId()), editorVO1);
newTitleVO.getEditors().put(
  new Long(editorVO2.getEditorId()), editorVO2);
newTitleVO.getEditors().put(
  new Long(editorVO3.getEditorId()), editorVO3);
```

加进newTitleVO对象的每个EditorDAO创建title_editor表中的一个项目。

```
/*Insert the record into the DAO*/
titleDAO.insert(newTitleVO);

/*Retrieving the record I just inserted into the database*/
TitleVO  titleVO  = (TitleVO) titleDAO.findByPrimaryKey(
```

```
                                    new Long(newTitleVO.getTitleId()).toString());

        /*Printing out the TitleVO data just inserted*/
        out.println("<B><P>Record just inserted:</P></B><BR/>");

        out.println("<P>Title id    : " + titleVO.getTitleId()    + "<br/>");
        out.println("<P>Title descr : " + titleVO.getTitleDescr() + "<br/>");
        out.println("<P>Title cost  : " + titleVO.getTitleCost()  + "<br/>");

        out.println("<P>Editor(s) Working on the Title  : <br/>");
        HashMap hashMap = titleVO.getEditors();

        Collection col = hashMap.values();
        Iterator iterator = col.iterator();

        while (iterator.hasNext()) {
          EditorVO editorVO = (EditorVO) iterator.next();

          out.println(editorVO.getFirstName() + " " +
                      editorVO.getMiddleName() + " " +
                      editorVO.getLastName() + "<br/>");
        }
      } catch(Exception e) {
        out.println("error--> " + e.getMessage());
      }
    %>
```

设置newTitleVO对象的insertFlag之后，用titleDAO EJB的insert()方法实际插入记录。insertExample.jsp页面中其余代码从subscription数据库中读取刚刚插入的记录并显示这个记录。

顺利运行insertExample.jsp页面时，可以看到如图3.5所示的结果。

图3.5

持久性框架策略

对机构中使用数据库的应用程序提供公共接口是个强大的结构，但要建立强大的持久性框架，就要解决几个问题，包括：

- 对要进入数据库的记录生成主键
- 处理多个用户并发更新同一数据库记录的问题
- 标识持久性框架中事务管理的最佳方法
- 优化持久性框架性能
- 将当前Java数据访问代码移植到持久性框架

下面几节介绍解决这些问题的不同策略。

主键生成策略

主键惟一标识数据库表格中的记录，通常是个数字值。几乎所有数据库都提供某种主键生成机制。从应用程序开发人员角度看，可以用不同方法对数据库插入生成主键，包括：

- 在记录插入数据库中时用数据库触发器自动产生主键。触发器用顺序对象（如果数据库支持）取得主键。
- 在记录插入数据库中时用自动递增列结构自动插入一个值。MySQL与Microsoft SQL*Server之类的一流厂家提供了产生主键的自动递增列。
- 直接用数据库顺序产生器生成主键，然后用于插入数据库中。例如Oracle的顺序对象可以查询下一个顺序值，然后在插入记录时使用这个值。
- 编写状态会话EJB或实体EJB，维护预先产生顺序的缓冲，然后可以从中取得顺序号。实体EJB从数据库表格中取得顺序号，然后更新表中存储的计数器值。会话EJB取得这些顺序号块并缓存。然后会话EJB将这些顺序提供给请求EJB者。需要多个顺序号时，实体EJB取得数据库中的另一组顺序号。这样，请求顺序号时就不必连续访问实体EJB。

这四种不同方法各有利弊。利用触发器、自动递增列和顺序对象之类的数据库结构很容易实现。使用这些方法时，用户不必考虑产生主键，但这些方法也有一些问题。

第一，用数据库触发器运行主键时，无法向进行数据库插入的代码返回生成的主键。如果操纵的数据中要建立父子关系，则这是个大问题。例如，如果代码插入一个订单，然后在表中插入订单细节行项目，则需要有所插入订单的主键以便插入订单细节行项目。

自动递增列与数据库触发器有同样的问题。插入之后怎么查找主键值呢？在JDK 1.3之前，这是有问题的，但JDK 1.4中的JDBC Statement对象可以表示是否从插入中返回自动产生的值。但由于这个特性在JDK 1.4中才发布，因此要认真检查JDBC驱动程序文档，看看驱动程序是否支持这个特性。

数据库顺序对象相当强大，可以在进行数据库插入之前查询和返回数值。Oracle提供了顺序对象，可以定义和查询主键值。MySQL提供了LAST_INSERT_ID()函数。数据库顺序对象相当强大但是数据库特定的。如果顺序对象分布在数据库代码中，而不是集中在一个服务

中，则应用程序要移植到新的数据库平台时，将会非常困难。

最后一个方法是用状态会话Bean缓存主键值，是很普及的实现方法，因为它很快，而且独立于数据库。缓存主键值需要的数据库调用更少，从而在事务量大的环境中大大提高性能。但是，这个方法也有几点需要注意：

- 在群集环境中很难用状态会话EJB缓存主键值，群集中的每个EJB要保证缓存的键不在任何其他EJB中重复。
- 如果运行的应用程序服务器关闭，则其中的状态会话EJB会丢失缓存的所有主键，从而造成数据库表格中所存储主键的“空白”。这通常不是大问题，但对有些数据库管理员，这种情形是不能接受的。
- 如果其他向数据库插入数据的应用程序不使用状态会话Bean，则会把数据搞乱。而使用触发器、自动递增列和顺序对象之类的数据库结构则不会遇到这类问题。

选择什么方法好呢？每个方法都已经顺利实现。使用数据库顺序对象通常是最简单的主键生成机制。由于数据库顺序对象是数据库特定的，因此一定要编写一个包装代码。

- 对应用程序的主键生成提供单个服务点
- 分离主键生成机制与实际使用主键的代码
- 将数据库特定特性隐藏在门户后面

在本章介绍的持久性框架中，主键用Counter EJB生成。Counter EJB负责根据传入getNextVal()方法的计数器ID生成惟一数字。Counter EJB对TitleDAO应用程序生成主键和行版本号。

Counter EJB用数据库表格组合跟踪顺序，用MySQL数据库的LAST_INSERT_ID()函数在预订数据库中生成主键。下面介绍如何实现Counter EJB，以便理解如何在持久性框架中采用这个EJB中的概念。

Counter EJB数据库结构

由于预订数据库中有多个计数器，因此创建counter表。Counter表有两个列：

- seq_num——表示计数器的当前值
- counter_name——是counter表的主键，表示数据库中的惟一计数器

Counter EJB是个标准会话EJB，有主接口、远程接口和Bean类。Counter EJB的主接口和远程接口如下：

```
package session.counter;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import dataaccess.DataAccessException;
import dataaccess.ServiceLocatorException;

public interface Counter extends EJBObject {
  public long getNextVal(String pCounterName)
    throws DataAccessException, ServiceLocatorException, RemoteException;
}

package session.counter;
```

```
import java.io.Serializable;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import javax.ejb.EJBHome;

public interface CounterHome extends EJBHome {
  Counter create() throws CreateException, RemoteException;
}
```

下面看看**Bean**实现类：

```
package session.counter;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.sql.*;
import dataaccess.*;

public class CounterBean implements SessionBean{
  SessionContext ctx = null;

  /*
   *  The getNextVal method will retrieve the next sequence number
   *  of the sequence the user requests.  The sequence values are stored
   *  in a database table called counter.
   */
  public long getNextVal(String pCounterName)
    throws DataAccessException, ServiceLocatorException {

    long              valueReturned   = 0;
    Connection        conn            = null;
    PreparedStatement updateStatement = null;
    ResultSet         rsCounter       = null;

    try {
      //Getting a connection to the subscription database
      ServiceLocator serviceLocator = ServiceLocator.getInstance();
      conn = serviceLocator.getDBConn(ServiceLocator.SUBSCRIPTIONDB);
```

用户调用**Counter EJB**的**getNextVal()**方法并传入一个**String**值，表示应用程序要递增的计数器时，即开始读取主键的过程。

下列代码更新所请求计数器的顺序号：

```
      //Building a SQL statement that will update the counter
      SQLCode sqlCode = SQLCode.getInstance();
      String sql = sqlCode.getSQLStatement("counter.nextval.update");

      //Executing the SQL statement
      updateStatement = conn.prepareStatement(sql.toString());
```

```
updateStatement.setString(1, pCounterName);
updateStatement.execute();
```

SQL代码类取得SQL语句如下：

```
UPDATE counter SET seq_num = LAST_INSERT_ID(seq_num+1)
WHERE counter_name = ?
```

注意UPDATE语句调用MySQL LAST_INSERT_ID()函数，传入当前seq_num列值并加1。LAST_INSERT_ID()函数象个顺序生成器，返回传入函数的表达式。LAST_INSERT_ID()函数返回的值对发出上述SQL语句的每个客户机惟一。由于我们用MySQL特定函数实现Counter EJB，因此如果要移植到不同的数据库平台，则要改变计数器类。

完成更新之后，代码发出一个SELECT语句，调用LAST_INSERT_ID()函数，立即取得递增seq_num值。

```
sql = sqlCode.getSQLStatement("counter.nextval.select");
//Executing the sql statement
updateStatement = conn.prepareStatement(sql.toString());
updateStatement.setString(1, pCounterName);
rsCounter = updateStatement.executeQuery();
```

与counter.nextval.select键相关联的SQL代码如下：

```
SELECT LAST_INSERT_ID() seq_num FROM counter WHERE counter_name = ?
```

在上述SELECT语句中调用LAST_INSERT_ID()函数后，返回前面调用UPDATE时更新的seq_num值。从SELECT语句取得的值返回调用getNextVal()方法的应用程序。

```
//Pulling the new sequence id
if (rsCounter.next()){
  valueReturned = rsCounter.getLong("seq_num");
}
} catch(SQLException e) {
  //Marking the session bean for rollback
  ctx.setRollbackOnly();
  throw new DataAccessException(
        "Error has occurred in CounterBean.getNextVal()", e);
} finally {
  try {
     //Close my database connections
     if (updateStatement != null) updateStatement.close();
     if (rsCounter!=null) rsCounter.close();
     if (conn!=null) conn.close();
  } catch(SQLException e) {
    System.out.println("Unable to close resultset, database " +
                "connection or statement in Counter.getNextVal");
  }
```

```
        }
        return valueReturned;
    }

    public void ejbCreate(){}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext pCtx){
        ctx = pCtx;
    }
}
```

记住，这只是实现主键生成策略的一种方法。这个方法的强大之处在于可以在应用程序中完全改写主键生成方法，而使用这个主键生成机制的DAO不需要改写。

并发性策略

假设下列情形：两个用户要更新同一记录，都取得记录的拷贝。用户1提交一些改变并实现改变，然后用户2对同一记录提交一些改变并实现改变，则用户2的改变会覆盖用户1保存的数据。

这里的问题是两个用户的事务是并发进行的。两个用户不知道对方也取得了同一记录的拷贝和处理这个拷贝。此外，用户1不知道用户2改变了自己刚刚实现的记录。这时需要一个锁策略，防止两个用户并发处理同一记录时一个用户覆盖另一用户的记录。可以采用两种不同策略：

- 第一个读取记录的用户可以对记录采用显式数据库锁，阻止所有其他用户读取与操纵这个记录，直到用户显式释放这个锁。这个策略称为保守式锁策略（pessimistic locking strategy），因为第一个用户使用记录时，别人都不能访问这个记录，即使只是读取记录。保守式锁策略是隔离不同用户事务的最安全方法，但在事务量较大的应用程序中，会使性能大大下降，因为一个用户锁住记录很容易使其他用户需要等待。由于每个用户需要等待，因此可能锁住其他用户需要的其他记录，从而造成死锁。
- 第二种策略是允许两个用户读取记录拷贝，记录拷贝中包含某种版本信息。所读取记录中的版本信息与数据库中存储的版本号进行比较。如果版本信息显示用户要更新的记录没有发生更新，则进行更新。这个策略称为开放锁，每个用户可以访问数据和修改数据，而没有放上显式数据库锁的开销。

开放锁特别适合事务量较大的应用程序中读取和迅速更新少量数据。

前面曾介绍过，本章建立的持久性框架使用开放锁策略。开放锁策略根据开放锁是只用于新应用程序开发还是用于现有应用程序，难易度不同。

可以用不同方法对新应用程序实现开放锁策略，两个最常见的机制是：

- 表中每行放一个行版本列。更新记录之前，比较要更新的记录的行版本与数据库中记录的行版本。

- 用日期/时间标志列保存上次更新记录的日期/时间标志。更新记录之前，比较要更新的记录的日期/时间标志与数据库中记录的日期/时间标志。如果要更新的记录的日期/时间标志比数据库中记录的日期/时间标志旧，则表明用于更新的记录是过时的。

Subscription数据库中实现基于row_version号的开放锁策略。row_version开放锁策略要求在数据库的每个表中增加row_version列。这个row_version列在每次更新记录时对特定记录更新。可以用两种方法实际检查要更新的数据是否过时：

- 在DAO层中放上代码，来检查要更新的数据是否过时。本章前面TitleDAO EJB中的update()方法就是这么干的。update()方法在UPDATE语句的WHERE从句中用主键和行版本更新title表中的记录。
- 用数据库触发器实现开放锁策略。数据库中每个表有个数据库触发器，在发生记录更新时触发。触发器比较要更新的记录的行版本与数据库中记录的行版本，如果相同，则更新记录，如果不同，则数据库触发器发出一个异常，不更新记录。

用数据库触发器管理开放锁策略实际上是比较容易实现的，DAO中不会有开放锁代码，一切都是在触发器层完成的。此外，你不必考虑应用程序中更新数据的所有应用程序是否也实现开放锁代码。

使用触发器方法的缺点是数据库触发器是数据库特定的，很难在多个数据库上支持应用程序。但是，ER-WIN之类工具可以帮助这种移植，可以通过工具对多个数据库产生触发器，从而使移植大大简化。

本章把开放锁代码放在DAO层，因为MySQL数据库不支持触发器。

对现有应用程序，实现开发锁策略比较困难。如果你拥有应用程序源代码，则实现开发锁策略时通常要遍历整个代码，将所有数据库选择与更新改成实现开发锁策略。此外，应用程序代码还要修改成处理事务过程中发生的任何开发锁异常，这是很不容易的。

如果你不拥有应用程序源代码，则要看看厂家能否实现开发锁策略。如果厂家不愿意实现，则毫无办法。但如果选择包时知道将要把机构中的其他应用程序进行集成，则要询问应用程序如何实现锁策略。

如果厂家使用保守锁，则通过对数据库的行或表格发出锁而锁住数据库资源，这样会在事务量大的环境中造成性能问题。如果不用任何锁策略（许多应用程序厂家不用任何锁策略），则可能在集成这个包与机构中的其他应用程序时把数据搞乱。在包选择阶段了解这些情况能够在后面进行应用程序集成时省去大量麻烦。

事务管理策略

J2EE提供了两种管理数据库事务的不同方法的Bean管理和容器管理。对于Bean管理事务，应用程序开发人员要把事务代码放在DAO中（假设JDBC代码直接放在DAO中）或实体Bean中（假设Bean管理持久性实体Bean）。开发人员负责开始事务和实现与撤销事务。

开发人员用Java Transaction API（JTA）控制事务。如果开发人员离开Bean管理事务EJB的方法调用之前没有实现与撤销事务，则EJB容器抛出异常。抛出的异常针对运行EJB的容器。

在容器管理事务中，也称为声明式事务管理，运行EJB的容器管理事务。应用程序开发人员在EJB部署描述项中声明容器如何管理事务。EJB部署描述项中可以声明六种不同事务类型：

- NotSupported

  对于NotSupported事务类型，EJB不能参与任何事务。如果EJB开始事务，然后调用另一个事务类型为NotSupported的EJB，则调用EJB的事务暂停。由于用DAO模式处理数据库事务，因此不能用这个事务类型声明DAO EJB。

- Supports

  Supports事务类型告诉容器在已经开始事务时在事务中运行EJB。如果还没有开始事务，则事务中不运行EJB。由于DAO可以插入、更新和删除数据库记录，因此一定要在事务中运行。由于Supports事务类型不能保证事务中运行EJB，因此建议不要用它建立本章介绍的DAO EJB。

- Required

  Required事务类型强制在事务中运行EJB。如果事务已经存在，则EJB在这个事务情境中运行；如果事务不存在，则开始一个事务EJB在这个新事务情境中运行。

- RequiresNew

  RequiresNew事务使EJB在独立于任何当前运行事务的事务中运行。这个事务类型用于独立运行数据库事务。

- Mandatory

  Mandatory事务类型要求EJB在已经运行的事务中运行。如果没有运行的事务，则向调用应用程序抛出javax.ejb.TransactionRequiredException。

- Never

  Never事务类型告诉EJB容器，EJB不能在事务中运行。如果事务中当前运行的代码调用它，则抛出异常。这个异常取决于EJB作为本地EJB还是远程EJB，分别为javax.ejb.EJBException或javax.ejb.RemoteException。

对应用程序编写数据持久性框架时，建议使用容器管理事务，原因如下：

- 用容器管理事务编写持久性框架更容易，所有事务管理都是声明式进行的，不必在事务代码中加入DAO代码。如果事务跨多个组件，则事务代码很难编写和调试。
- 容器管理事务更灵活。如果要改变EJB的事务行为，只要改变EJB的部署描述项和重新部署Bean。例如，假设要保证Counter EJB进行的实现总是在新事务中进行，如果Counter EJB当前设置事务类型为Required，只要在Counter的部署描述项中将事务类型变成RequiresNew，然后重新部署EJB，而不必改变Counter代码。
- 不好的事务代码可能影响应用程序性能。记住，事务进行时，应用程序服务器和数据库资源要跟踪事务进程。开发人员经常编写Bean管理事务代码，使事务需要太长时间打开。在高容量事务处理应用程序中（如电子商务Web站点），不好的事务管理代码会影响应用程序性能。

记住，Bean管理事务能提供更大的灵活性，使持久性框架开发人员能更好地进行事务控制与隔离。但是，这种控制在大多数应用程序中不需要。本节只是简要介绍EJB事务管理。

性能策略

抽象和包装是要付出性能代价的。建立数据持久性框架通常比直接使用数据的性能要

差。但是，实现持久性框架能提高复用性和降低维护成本，因此付出性能代价是值得的。

但是，实现持久性框架时一些常见错误会严重限制应用程序的性能与伸缩性。一些常见性能问题包括设计错误和误解EJB工作方式。

- 持久性框架中的DAO粒度不合适
- 从DAO find()方法调用返回大量数值对象
- 常见EJB实现问题

持久性框架开发人员常常把数据模型直接映射到DAO类，结果使持久性框架中太多DAO类，并且取得大量细粒数值对象。

持久性框架应利用粗粒对象，持久性框架中的DAO不应返回大量数值对象。因此，设计持久性框架时应记住下面几点：

- **观察框架中DAO返回的对象广度**

  find()方法返回太多数值对象会使应用程序很快造成Java虚拟机（JVM）内存问题。持久性框架如果没有正确的粒度，则要经常回收内存或使JVM内存超界。

- **观察框架中DAO返回的对象深度**

  采用持久性模式的一个常见错误是数值对象的对象层次太深，这也是个粒度问题。返回的每个数值对象中不应包含超过三层其他对象，否则即使读取和更新几个数值对象也会造成巨大开销。

本章介绍的持久性模式适合编写OLTP（联机事务处理）应用程序，其中应用程序处理少量由最终用户操纵的数据。但在报表数据库和数据仓库之类的OLAP（联机分析处理）应用程序中实现这些持久性模式时则会造成严重的伸缩性和性能问题。

在这些情形中，最好直接用JDBC访问数据，直接读取ResultSet。要取得ResultSet，应用程序就要取得JDBC连接，然后发出SQL查询和取得ResultSet。为此，前面建立的每个持久性框架抽象原则似乎都不合适。但是，体系结构不是绝对的，难免遇到例外情形。

但是，Optional Package（和JDK 1.4）中提供了更多选项。

JDBC Optional Package（见J2EE）引入了RowSet的概念。RowSet（行集）扩展JDBC ResultSet，有一个重要差别。RowSet可以和数据库连接切断，因此即使关闭JDBC连接，RowSet仍然可以读取和操纵数据。后面RowSet可以重新建立数据库连接和用其中包含的任何数据改变进行更新。

如果只要偶尔读取大量数据，则可以集成和增加find()方法到返回RowSet的DAO中。记住，DAO模式抽象数据实际读取方法，即可以让一个find()方法用JDBC返回RowSet，另一个find()方法返回从实体EJB读取的数值对象。对于批量更新、插入与删除，甚至可以将RowSet传回DAO，让DAO代替使用RowSet的应用程序更新、插入与删除。

实现J2EE持久性框架时另一个常见的性能问题是框架开发人员会遇到编写EJB应用程序时的错误。由于持久性框架通常要对数据库执行许多小事务，因此要记住下面两点：

- 注意每次进行JNDI查找时，都有相关联的开销。如果一个经常使用的DAO用多个其他DAO完成事务，则可以使用具有缓存策略的Service Locator减少要进行的JNDI查找次数。在事务量大的环境中，这样可以大大减少进行的JNDI查找次数。
- 适当使用本地和远程EJB接口。如果DAO层只让应用程序服务器中运行的应用程序代

码使用，则可以对DAO实现使用本地接口。本地接口是EJB 2.0规范内容，可以大大减少原先影响EJB应用程序开发的网络通信流量。

设计持久性框架时，一定要从一开始就考虑性能问题。在开发框架和开始编码时实现上述性能策略要比框架中已经建立几个应用程序之后容易得多。

好的体系结构从开始就考虑，而不是在部署项目之后再考虑。

移植策略

大多数IT开发人员都喜欢花时间编写新应用程序，但实际上开发人员的大部分时间都用在支持现有代码，集成应用程序。

新应用程序的持久性框架总是比较容易实现，但已经写好的J2EE应用程序如何实现持久性框架呢？从小处着手，慢慢实现。大多数J2EE应用程序利用会话EJB建立应用程序的业务逻辑，但整个会话EJB层交织了数据访问代码。

关键要从小处着手，标识持久性层要表示的几个关键数据实体。通常，最好的实体是机构的所有应用程序中都要使用的数据实体。对大多数公司，客户数据是数据库中最关键的信息。用这个数据实体作为建立持久性框架的基础。

开始实现持久性框架之前，无论对新应用程序开发还是移植现有应用程序，都要先有个建立整个框架的概念性地图。这个地图要布置框架中的所有层及各个组件如何交互。

图3.6所示的是个框架地图的例子。

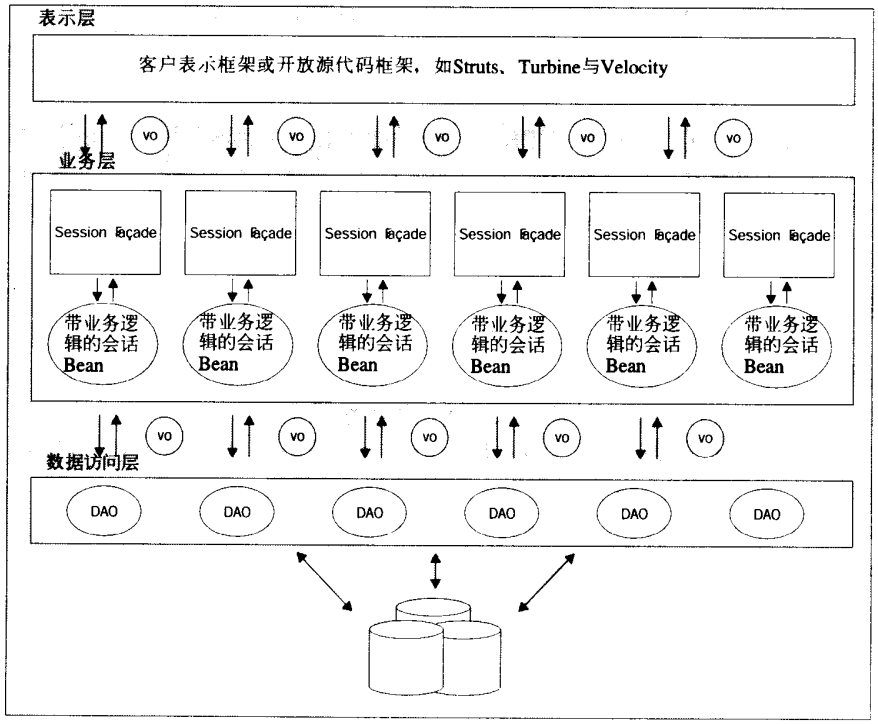

图3.6

上述框架是使用闭合层方法的多层体系结构。之所以是个闭合层方法，是因为每层只能与直接相邻的层交流。表示层只能与业务层交流，业务层只能与表示层和数据层交流，只有数据层才能访问所有公司数据源。

上述地图中的业务层分为两个部分：Session Façades和会话Bean。Session Façades是表示层与会话EJB之间的Java类，包装会话EJB并使表示层不关联到处理业务逻辑的EJB。数据访问层的DAO用会话Bean取得与操纵数据。这个体系结构中只有数值对象可以在不同层之间来回移动数据。

一旦建立整个框架的概念性地图和了解应用程序如何与数据层通信之后，就可以开始把现有应用程序群移植到持久性框架中。

这个移植过程中涉及的步骤包括：

- 检查当前使用客户数据的所有应用程序。确定应用程序中重复使用的核心客户数据元素，并确定不同业务过程如何使用这些数据元素。
- 建立持久性框架的基础类，标识需要的数据源并将其包装成DataSourceFactory。
- 标识如何产生框架中的主键。检查要访问的基础数据库，开发产生主键的集中机制，要与当前存储客户记录的方式一致。
- 标识是否需要实现开放锁机制。记住，如果从小处开始，则不必对数据库中每个表实现开放锁机制。
- 建立要用的DAO与VO类，如果大多数代码写成JDBC代码，则可以看看在会话EJB中直接使用JDBC。
- 遍历使用客户数据的每个应用程序，将其转换到新的持久性框架。

移植策略不能跨大步，而要从少量数据实体开始，不要在数据库访问技术上作太大变化。

这样，如果应用程序代码使用JDBC，则要在持久性框架实现期间坚持使用JDBC。不要跳到容器管理持久性或其他大不相同的方法。后面可以在持久性框架的其他迭代中这么干。否则不仅要调试实现的模式，还要支持小组可能不熟悉的新数据库访问技术。

小结

J2EE平台提供了建立持久性框架的灵活平台，其组件体系结构和多数据库API提供了建立持久性框架的现成工具，可以满足大部分应用程序开发需求。

数据是机构的IT基础结构中惟一不变的东西。应用程序不断改变，但应用程序使用的数据不会消失。数据可以移植到另一数据库平台或转换成不同格式与结构，但基本上还是相同的。毕竟，公司实现新的客户关系管理（CRM）应用程序时，并不抛弃原先的客户数据，只是把数据移植到新系统中，用新的数据结构处理。

数据持久性框架的强大之处正在于此。所有应用程序开发框架都提供一定程度的代码复用。但是，持久性框架不仅让开发人员复用代码，而是定义了读取与操纵数据实体的标准接口，这对机构非常重要。

持久性框架的基础技术可能改变，但可以得到复用性，因为移植或移动数据实体时，使用框架的应用程序不必全部改变，只有框架实现改变，而应用程序读取与操纵数据实体的标准接口并不改变。

持久性框架的强大之处在于可以抽象基础数据源实现。

本章提供的三个基本模式奠定了数据持久性框架的基础，包括：

- Data Access Object模式隐藏特定业务数据实体的读取、插入、更新与删除方法
- Value Object模式抽象公司数据源中数据实体之间的物理关系
- Service Locator模式抽象从数据库中取得EJB与JDBC连接的方法

除了这些核心模式之外，本章还介绍了处理下列问题的策略：

- 主键生成
- 并发性管理
- 事务管理
- 性能策略
- 将现有应用程序移到持久性框架的策略

第4章 改进性能与伸缩性的设计模式

使用J2EE平台时要考虑许多设计问题，其中一些（如性能与伸缩性）不是J2EE特有的，而是任何分布式系统都有的。这些设计元素可以通过使用适当风格和模式而大大改进。相反，不良设计可能使系统性能很差，再好的硬件也无济于事。

没有一类模式是专门提高性能与伸缩性的，但有些模式能解决性能与伸缩性问题。选择模式不是专门为了性能，但最好知道有些模式可以提高性能。

本章介绍几个改进性能与伸缩性的设计模式，大部分已经在前面几章介绍。

性能问题的原因

独立应用程序与使用J2EE框架的应用程序之间一个重要差别（从性能角度看）是涉及的虚拟机（VM）个数。独立应用程序在单VM中运行，而使用J2EE框架的应用程序可能有多个VM（Web服务器、小服务容器、应用程序服务器，等等）。此外，所有这些VM可以放在完全分开的机器上。即使放在同一台机器上，J2EE应用程序也需要在这些不同VM之间通信，从而可能造成巨大的性能问题。

远程方法调用

Java语言本身没有提供VM之间进行方法调用的机制，但有几个语言扩展提供了从一个VM中的一个对象对另一个VM中的一个对象进行方法调用的管道。

提供这个功能的基本扩展是远程方法调用（RMI，Remote Method Invocation）。RMI提供一组类和接口，可以透明地调用远程对象（与客户机在不同JVM中执行的对象）。RMI还要求低级传输，使对象可以调动到请求，在网络上传输。在J2EE中，这个协议称为Internet ORB间协议（IIOP，Internet Inter ORB Protocol）。

RMI向调用客户机隐藏对象地址。远程对象可以通过一个接口访问，其用残根和代理实现。代理将任何方法调用转发到远程对象，等待答复，然后将结果返回调用对象。客户机应用程序可以把远程对象当作本地对象一样，用与远程对象通信的本地代理的本地引用。

当然，情况没有这么简单。每个远程调用要经过三层抽象：

- **残根/框架层**

  残根/框架层截获客户机对接口引用进行的方法调用，将这些调用改向到远程对象。记住，残根针对客户端，而框架在服务器方。

- **远程引用层**

  这层截获与管理客户机对远程对象的引用，将客户机连接服务器上运行和导出的远程对象，使用一对一连接链路。在Java 2 SDK中，这层改进成支持激活框架（见稍后介绍）。

· 传输层

这层基于网络上机器之间的TCP/IP连接，提供基本连接和一些防火墙穿透策略。

图4.1显示了这三层和传输层的细节，右边是OSI层名（见http://www.webopedia.com/TERM/O/OSI.html/）。

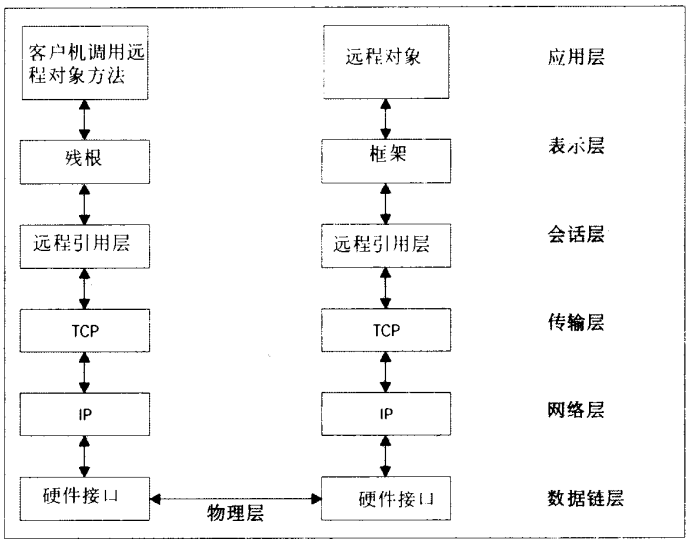

图4.1

从开发角度看，使用远程调用好像很简单，但实际上，象本地调用一样使用远程调用时，幕后发生了许多操作。

所有EJB远程主接口（扩展EJBHome）和远程组件接口（扩展EJBObject）用这个机制调用远程对象的方法。它们受到RMI实现要求的限制。远程接口要扩展java.rmi.Remote接口，这是通过扩展EJBHome和EJBObject实现的，它们也扩展java.rmi.Remote接口。此外，远程接口的所有方法调用还要抛出java.rmi.RemoteException。远程方法的变元和返回值应为Java基本类型、可序列化或远程引用。RMI通常用对象序列化按数值传递变元，而VM方法调用则传递对象引用。

远程客户机使用的代理对象不是轻量级的，而是复杂的资源占用对象，创建时需要大量资源，通常是一些网络调用的结果。如果客户机真是远程的（即与目标对象不在同一VM中），则要创建连接客户机与目标VM的网络套接字。

同样，服务器方目标对象也是复杂的资源占用对象，比客户端代理对象更加复杂。在传统RMI服务器中，远程对象有自己对应的客户端代理对象。服务器方“代理”从客户端代理对象取得请求，开包，并调用目标对象的相应方法。随着应用程序服务器的出现，情况有所改变，因为优化服务器实现可以用派遣机制接受请求，将其转发到目标对象。服务器也提供了缓存机制，可以用对象实例池处理大量客户机请求。

尽管各种缓存机制与优化支持服务器对象，但客户机通常无法提供这种复杂的管理机制。如果应用程序要求客户机保持成百上千个远程引用，则会对客户机资源带来巨大的负担。RMI最大的性能瓶颈之一是按数值传递对象涉及的工作。

网络调用锁存

VM内的方法调用通常只要几个微秒，而远程调用则要几毫秒、几秒才能完成，如果遇到网络问题，则可能无法完成。远程调用比VM内的方法调用慢好几个数量级。

使用远程接口时，一定要认真设计，减少网络调用次数，远程调用要完成最大量的有用工作。

常见设计问题

下面看看产生太多网络通信流的几种实现问题。

专用成员访问/更换方法

考虑典型Java对象，它有专用数据成员，用访问/设置方法访问和修改这些数据成员。将这些对象的方法直接映射更新远程接口会造成读取对象状态时需要许多方法调用。远程客户机首先要取得对象的远程引用，然后用多个getXXX()访问方法将状态数据传输到本地VM。这个模型在本地VM内的方法调用中是可行的，但进行远程调用时，多个getXXX()方法调用的锁存很大。换句话说，每个getXXX()调用都需要网络往返，会造成网络锁存问题。

聊天对象

一个常见的开发风格是用控制器实现用例。控制器对象包含的业务逻辑保持多个其他对象的引用，调节这些对象之间的交互，通常实现某种形式的工作流程。如果把这个控制器实现为客户端对象，而客户机是远程的，则客户机需要许多远程引用。客户端控制器还会对服务器方对象进行许多远程调用。这种聊天客户机是需要避免的。

每次进行远程调用时，都会影响性能，因为会出现网络锁存问题。因此，为了提高性能，就要减少远程调用次数。

伸缩性问题的原因

可伸缩系统有什么特征呢？对一定可测量范围的系统可接受性能（例如对请求的响应时间），测量系统工作量（如并发用户数时），如果工作量增加时性能仍保持在可接受性能范围内，则系统是可伸缩的。显然，这个增加不可能是无限的。

到一定时候，系统会超载，性能测试不再保持在可接受性能范围内。任何系统的容量都有一个上限，通常表示为面向Web系统的并发连接用户数或系统每秒钟可以进行的事务数。

紧耦合系统

假设用户与使用Web浏览器的系统交互，基础协议为HTTP或HTTPS，是基于连接的。客户机浏览器打开与Web服务器的连接，发送请求，然后等待答复。收到答复时，浏览器向用户显示页面。

浏览器活动通常采用请求/答复交互模型。Internet中可能遇到低速Web服务器。此外，由于人们分析信息的速度比计算机系统速度慢好几个数量级，因此这是可接受的交互模型。但是，对系统与系统的交互，则不能接受几秒钟的延迟。

阻止请求/答复交互模型有两个基本问题：

- 第一是阻止客户机。如果响应太慢，则客户机可能放弃，转到其他地方。
- 第二是阻止客户机对服务器的限制。服务器要立即响应，尽快答复。

连接客户机个数确定了服务器负荷。服务器无法选择如何或何时安排请求。每个请求都要尽快响应。这类面向连接系统称为紧耦合系统。为了提高伸缩性，需要减少系统中不同交互组件之间的耦合量。

可以通过几种方法减少耦合量，从而提高系统性能和伸缩性。

同步与异步行为

在使用J2EE平台的系统中，大多数分布式通信都是同步阻止/调用型的，客户端交互（如浏览器访问服务器）和组件间交互（如小服务调用会话Bean方法）都是这样。这种行为是VM内方法调用的自然扩展。CORBA与RMI之类技术提供远程引用，使远程对象像本地对象一样。这样也会得到VM内调用的阻止行为，但由于我们知道有网络锁存，因此知道会阻止这么久。但也可以不这样。

异步调用

在异步交互模型中，客户机发送请求，然后忘记它，进行下一个操作。过一定时间后，请求送到服务器中处理。如果客户机需要确认这个请求得到处理，或需要收到处理结果，则可以发生客户机与服务器之间的第二次交互。

显然，这种交互模型不适合所有业务过程，但异步进行的调用越多，对服务器资源的限制越少。

间接

要提高伸缩性，一种最明显的方法是投入更多硬件。这可以水平进行（在一层中群集资源），也可以垂直进行（增加内存、CPU等资源）。但是，如果组件紧耦合，显式链接相互的地址，则增加硬件资源要困难得多，因为要对新硬件资源更新客户机，增加网络配置。

因此，我们要通过间接减少耦合，使客户机不知道或不关心使用的服务器资源，然后即可增加更多服务器资源而不影响客户机。

城市休假订票应用程序

为了演示本章模式的应用，我们用城市休假订票系统协调特定城市度假时宾馆与机票的预订，主要用例包括（如图4.2所示）：

- **View available vacations（浏览度假）**

  列出数据库中所有度假的城市

- **Get a City Break vacation quote（取得城市休假配额）**
  - 选择城市
  - 选择宾馆
  - 选择航班（去程与回程）
- **Book a City Break vacation（订购城市休假）**
  - 注册为新客户或登记为现有客户
  - 用前面构造的配额订假期
- **Register as a new customer（注册为新客户）**
  - 发送新客户细节
- **Login as an existing customer（登记为现有客户）**
  - 发送登录系统的E-mail和口令细节

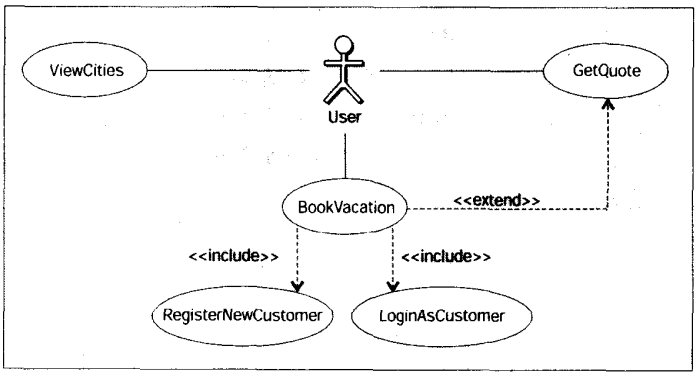

图4.2

我们的实体域对象包括：

- **City（城市）**
  城市信息，包括描述与机场
- **Hotel（宾馆）**
  每个城市的宾馆信息，包括名称、描述与可用房间数
- **Flight（航班）**
  所有航班信息，包括起飞时间、目的地、剩余票数
- **Customer（客户）**
  基本客户信息，用于登录和跟踪订票
- **Booking（预订）**
  航班、宾馆与客户信息的组合

我们用容器管理持久性2.0实体Bean实现这些实体如图4.3所示：

实现这些实体

所有实体Bean都很简单，都使用容器管理持久性和本地接口，但要注意下列几点：

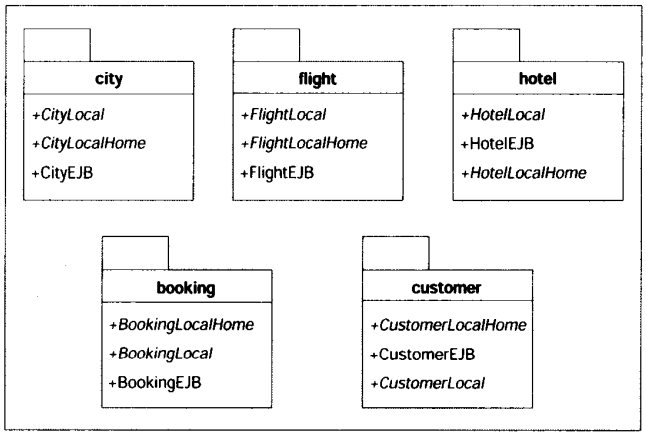

图4.3

- 我们用Counter会话Bean创建惟一ID，这是每个实体的主键。这个Counter bean与第3章使用的Counter bean相同。
- Counter bean本身用Service Locator模式定位，修改第3章的ServiceLocator类而实现。
- 我们对这个应用程序使用Value Object模式。有些实体Bean用数值对象创建和读取数据。本章稍后将介绍使用数值对象的好处。

这里只对每个实体提供Bean实现类，主接口和组件接口见本章的下载源代码部分（见http://www.wrox.com/）。

这些实体Bean是所要实现的简化版本，便于简化演示所用模式。

City实体Bean

City实体Bean包含三个属性：城市名、城市描述和该城市的机场名。它和稍后的Flight实体Bean一起用于选择航班（去程与回程）。

```
package entity.city;

import javax.ejb.*;
import common.*;
import common.counter.*;

public abstract class CityEJB implements EntityBean {

  public abstract Long getId();
   public abstract void setId(Long id);

   public abstract String getName();
   public abstract void setName(String name);

   public abstract String getDescription();
   public abstract void setDescription(String desc);
```

```
    public abstract String getAirport();
    public abstract void setAirport(String airport);

    public Long ejbCreate(String name, String desc, String airport)
        throws CreateException {

      try {
        ServiceLocator locator = ServiceLocator.getInstance();
        CounterHome home = (CounterHome)
          locator.getEJBHome(ServiceLocator.COUNTER);
        Counter counter = home.create();
        setId(counter.getNextVal("CITY"));
        setName(name);
        setDescription(desc);
        setAirport(airport);
      } catch (ServiceLocatorException sle) {
        throw new EJBException(sle);
      } catch (java.rmi.RemoteException re) {
        throw new EJBException(re);
      }

      return null;
    }

     public void ejbPostCreate(String name, String desc, String airport)
       throws CreateException {}

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbRemove() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}

}
```

Flight实体Bean

Flight实体Bean包含八个属性：起点站、终点站、起飞时间、到达时间、航线、票价、票数和航班日期。

```
package entity.flight;

import javax.ejb.*;
import vo.FlightVO;
import common.*;
import common.counter.*;

public abstract class FlightEJB implements EntityBean {
```

```
public abstract Long getId();
public abstract void setId(Long id);

public abstract String getFrom();
public abstract void setFrom(String from);

public abstract String getTo();
public abstract void setTo(String to);

public abstract String getDepartureTime();
public abstract void setDepartureTime(String time);
public abstract String getArrivalTime();
public abstract void setArrivalTime(String time);

public abstract String getAirline();
public abstract void setAirline(String airline);

public abstract int getAvailableSeats();
public abstract void setAvailableSeats(int seats);

public abstract String getDate();
public abstract void setDate(String date);

public abstract double getPrice();
public abstract void setPrice(double price);

public Long ejbCreate(String from, String to, String dept,
                      String arrival, String airline, int seats,
                      String date, double price)
    throws CreateException {

  try {
    ServiceLocator locator = ServiceLocator.getInstance();
    CounterHome home = (CounterHome)
        locator.getEJBHome(ServiceLocator.COUNTER);
    Counter counter = home.create();

    setId(counter.getNextVal("FLIGHT"));
    setFrom(from);
    setTo(to);
    setDepartureTime(dept);
    setArrivalTime(arrival);
    setAirline(airline);
    setAvailableSeats(seats);
    setDate(date);
    setPrice(price);
  } catch (ServiceLocatorException sle) {
    throw new CreateException();
  } catch (java.rmi.RemoteException re) {
    throw new CreateException();
  }
```

```
        return null;
    }

    public void ejbPostCreate(String from, String to, String dept,
                              String arrival, String airline, int seats,
                              String date, double price)
        throws CreateException {}

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbRemove() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
```

这个实体Bean用FlightVO数值对象返回数据，还有一个业务方法，可以在订票时减少余下票数：

```
    public FlightVO getFlightInfo() {
        FlightVO vo = new FlightVO();

        vo.flightId = getId();
        vo.from = getFrom();
        vo.to = getTo();
        vo.departure = getDepartureTime();
        vo.arrival = getArrivalTime();
        vo.airline = getAirline();
        vo.availableSeats = getAvailableSeats();
        vo.date = getDate();
        vo.price = getPrice();

        return vo;
    }

    public void bookSeats(int seats) {
        setAvailableSeats(getAvailableSeats() - seats);
    }
}
```

这个实体Bean还要一个寻找方法，可以按日期、终点站和起点站搜索航班（findByFlightDateToFrom()）。所要的EJB QL如下：

```
SELECT OBJECT(f) FROM flight f WHERE f.date=?1 AND f.from=?2 AND f.to=?3
```

Hotel实体Bean

Hotel实体Bean有五个属性：宾馆名、宾馆描述、地图、房间数和每晚房价。

```
package entity.hotel;

import javax.ejb.*;
import common.*;
import common.counter.*;
import vo.HotelVO;

public abstract class HotelEJB implements EntityBean {

   public abstract Long getId();
   public abstract void setId(Long id);

   public abstract String getName();
   public abstract void setName(String name);
   public abstract String getLocation();
   public abstract void setLocation(String location);

   public abstract int getAvailableRooms();
   public abstract void setAvailableRooms(int rooms);

   public abstract String getDescription();
   public abstract void setDescription(String desc);

   public abstract double getPricePerNight();
   public abstract void setPricePerNight(double price);

   public Long ejbCreate(String name, String location, int noRooms,
                         String description, double price)
      throws CreateException {

      try {
         ServiceLocator locator = ServiceLocator.getInstance();
         CounterHome home = (CounterHome)
            locator.getEJBHome(ServiceLocator.COUNTER);
         Counter counter = home.create();

         setId(counter.getNextVal("HOTEL"));
         setName(name);
         setLocation(location);
         setAvailableRooms(noRooms);
         setDescription(description);
         setPricePerNight(price);
      } catch (ServiceLocatorException sle) {
         throw new CreateException();
      } catch (java.rmi.RemoteException re) {
         throw new CreateException();
      }

      return null;
   }
```

```
    public void ejbPostCreate(String name, String location, int noRooms,
                              String description, double price)
        throws CreateException {}

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbRemove() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
```

和Flight实体Bean一样，Hotel实体Bean用数值对象返回数据，还有一个业务方法，可以在订房时减少余下房数。

```
    public HotelVO getHotelInfo() {

        HotelVO vo = new HotelVO();

        vo.id = getId();
        vo.name = getName();
        vo.description = getDescription();
        vo.location = getLocation();
        vo.noAvailableRooms = getAvailableRooms();
        vo.pricePerNight = getPricePerNight();

        return vo;
    }

    public void bookRooms(int rooms) {
        setAvailableRooms(getAvailableRooms() - rooms);
    }

}
```

还有一个寻找方法，可以按地址（**findByLocation()**）搜索特定城市的所有宾馆。所要的EJB QL如下：

```
SELECT OBJECT(h) FROM hotel h WHERE h.location=?1
```

Booking实体Bean

Booking实体Bean有八个属性，对应于城市度假的预订数据：客户ID、宾馆ID、该宾馆所要的房间数、航班ID（去程与回程）、这些航班中所需的票数、度假价格和起止日期。

```
package entity.booking;

import javax.ejb.*;
import vo.BookingVO;
import common.*;
import common.counter.*;
```

```
public abstract class BookingEJB implements EntityBean {

   public abstract Long getId();
   public abstract void setId(Long id);

   public abstract Long getCustomerId();
   public abstract void setCustomerId(Long id);

   public abstract Long getFlightIdOut();
   public abstract void setFlightIdOut(Long id);

   public abstract Long getFlightIdIn();
   public abstract void setFlightIdIn(Long id);
   public abstract Long getHotelId();
   public abstract void setHotelId(Long id);

   public abstract double getPrice();
   public abstract void setPrice(double price);

   public abstract int getNoRooms();
   public abstract void setNoRooms(int rooms);

   public abstract int getNoSeats();
   public abstract void setNoSeats(int seats);

   public abstract String getFromDate();
   public abstract void setFromDate(String from);

   public abstract String getToDate();
   public abstract void setToDate(String to);
```

预订只能用**BookingVO**数值对象进行。

```
   public Long ejbCreate(BookingVO vo) throws CreateException {

     try {
        ServiceLocator locator = ServiceLocator.getInstance();
        CounterHome home = (CounterHome)
           locator.getEJBHome(ServiceLocator.COUNTER);
        Counter counter = home.create();

        setId(counter.getNextVal("BOOKING"));
        setCustomerId(vo.customerId);
        setFlightIdOut(vo.flightIdOut);
        setFlightIdIn(vo.flightIdIn);
        setHotelId(vo.hotelId);
        setPrice(vo.price);
        setNoRooms(vo.noRooms);
        setNoSeats(vo.noSeats);
        setFromDate(vo.fromDate);
        setToDate(vo.toDate);
```

```
        } catch (ServiceLocatorException e) {
            throw new EJBException(e);
        } catch (java.rmi.RemoteException re) {
            throw new EJBException(re);
        }

        return null;
    }

    public void ejbPostCreate(BookingVO vo) throws CreateException {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbRemove() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
}
```

Customer实体Bean

Customer实体Bean有八个客户字段（但也可以增加）：客户姓名、地址、州、邮区号、E-mail地址、将来登录的口令、过去订票记录。

```
package entity.customer;

import javax.ejb.*;
import vo.CustomerVO;
import common.*;
import common.counter.*;

public abstract class CustomerEJB implements EntityBean {

    public abstract Long getId();
    public abstract void setId(Long id);

    public abstract String getName();
    public abstract void setName(String name);

    public abstract String getAddress1();
    public abstract void setAddress1(String addr1);

    public abstract String getAddress2();
    public abstract void setAddress2(String addr2);

    public abstract String getState();
    public abstract void setState(String state);

    public abstract String getZip();
    public abstract void setZip(String zip);
```

```
public abstract String getEmail();
public abstract void setEmail(String email);

public abstract String getPassword();
public abstract void setPassword(String pwd);

public abstract String getBookings();
public abstract void setBookings(String id);
```

和Booking实体一样，只能用CustomerVO数值对象创建新客户。

```
public Long ejbCreate(CustomerVO vo) throws CreateException {
   try {
      ServiceLocator locator = ServiceLocator.getInstance();
      CounterHome home = (CounterHome)
         locator.getEJBHome(ServiceLocator.COUNTER);
      Counter counter = home.create();

      setId(counter.getNextVal("CUSTOMER"));
      setName(vo.name);
      setAddress1(vo.addr1);
      setAddress2(vo.addr2);
      setState(vo.state);
      setZip(vo.zip);
      setEmail(vo.email);
      setPassword(vo.pwd);
   } catch (ServiceLocatorException e) {
      throw new CreateException();
   } catch (java.rmi.RemoteException re) {
      throw new CreateException();
   }
   return null;
}

public void ejbPostCreate(CustomerVO vo) throws CreateException {}

public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbLoad() {}
public void ejbRemove() {}
public void ejbStore() {}
public void setEntityContext(EntityContext ctx) {}
public void unsetEntityContext() {}
```

这个Bean还可以用数值对象返回客户信息（在登录成功后使用）。

```
public CustomerVO getCustomer() {

   CustomerVO vo = new CustomerVO();
```

```
        vo.customerId = getId();
        vo.name = getName();
        vo.addr1 = getAddress1();
        vo.addr2 = getAddress2();
        vo.state = getState();
        vo.zip = getZip();
        vo.email = getEmail();
        vo.pwd = getPassword();

        return vo;
    }
```

最后，还有另一业务方法，可以记录这个客户下过的订单，用管道分隔字符串记录订票ID。

```
    public void addBooking(Long id) {
        setBookings(getBookings() + "|" +id.toString());
    }
}
```

还有一个寻找方法，取E-mail地址和口令，检查现有客户能否登录系统（findByEmail-AndPassword()）。需要的EJB QL如下：

```
SELECT OBJECT(c) FROM customer c WHERE c.email=?1 AND c.password=?2
```

有了基本实体Bean之后，下面看看用什么设计模式实现用例。

标识提高性能的模式

在J2EE平台情境中，VM和潜在网络调用个数取决于系统的物理部署。包括Web服务器、小服务容器、EJB容器和数据库的完全分布式应用程序可能由四个不同VM构成。

但是，许多应用程序服务器厂家提供优化安装，把小服务容器与EJB容器放在同一VM中执行。还可以进一步优化，绕过远程接口按数值传递的语法，避免复制方法调用的参数，但这是不符合规范的。

但是，这些优化可以得到很大的性能好处，克服上述问题。但这并不符合规范，不能保证所有应用程序服务器厂家都能提供，而且可能导致移植问题。此外，可能无法把小服务容器与EJB容器实际地放置在一起，特别是使用复杂负载平衡和防火墙机制时。

典型部署利用分开的小服务容器与EJB容器，在专用机器上运行数据库。这样，基于Web的客户机可以通过小服务容器访问系统，非基于Web的客户机（其他系统）可以通过EJB容器访问系统。两种情况下，远程客户机和EJB容器之间进行远程调用，因为EJB通过远程接口（EJBHome与EJBObject）访问，因此这个领域是性能模式最有效的。

Service Locator模式

客户机要取得资源引用时，可以从JNDI命名服务来获取，也可以从另一方法调用的返

间值来获取。在任何比较复杂的J2EE应用程序中，都要访问大量资源。但是，查找与取得资源引用是相当费力的过程，特别是在层与层之间。

此外，对于EJB引用，客户机首先要访问EJBHome（远程调用），然后对主对象进行一些有用的工作，然后又要进行远程调用。

Service Locator模式与Singleton模式一起使用可以实现缓存机制，存储工厂对象的初始情境对象和引用（EJBHome、JMS连接工厂、JDBC数据源，等等），从而减少大量JNDI查找和需要的远程调用次数，如图4.4所示。

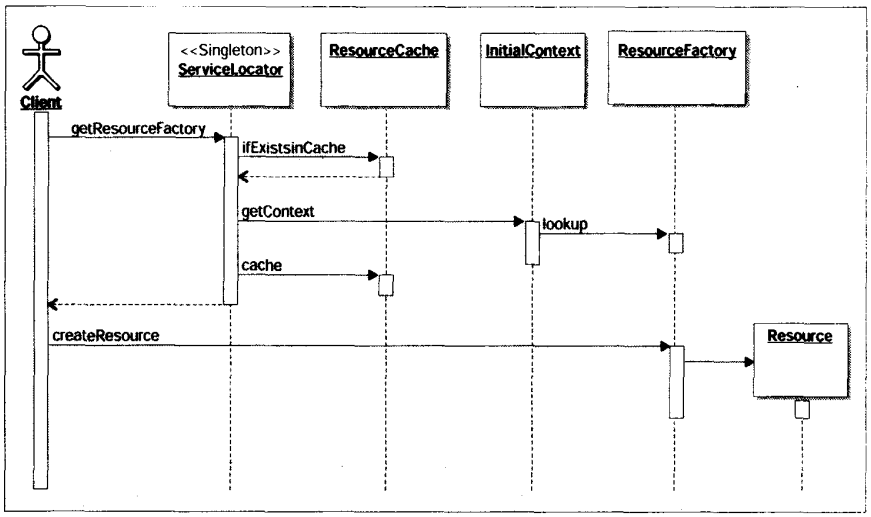

图4.4

在城市休假应用程序中实现Service Locator

上一章已经介绍Service Locator实现。这里只是进行扩展，以便缓存EJBLocalHomes和JMS资源（稍后将介绍其原因），并根据这个应用程序的Bean相应修改。

完整源代码见本章的下载源代码部分（见http://www.wrox.com/）。

要通过Service Locator提高性能，就要缓存任何资源工厂。

```
private static Hashtable ejbHomeCache                    = null;
private static Hashtable dataSourceCache                 = null;
private static Hashtable queueConnectionFactoryCache     = null;
private static Hashtable queueCache                      = null;
private static Hashtable ejbLocalHomeCache               = null;
```

要用Service Locator取得资源工厂时，可以先检查其是否已经缓存。例如：

```
if (ejbLocalHomeCache.containsKey(serviceName)){
   ejbLocalHome = (EJBLocalHome) ejbLocalHomeCache.get(serviceName);
   return ejbLocalHome;
} else {
```

如果是首次请求资源，则要进行查找，然后将引用放在缓冲区。

```
            Context ctx = new InitialContext();
            Object jndiRef = ctx.lookup(serviceName);
            ejbLocalHome = (EJBLocalHome) jndiRef;
            ejbLocalHomeCache.put(serviceName, ejbLocalHome);
            return ejbLocalHome;
          }
```

下面快速介绍新的方法。首先，getLocalHome()方法与getEJBHome()大致相同，只是用EJBLocalHome接口而不用远程EJBHome。

```
...
...
  public EJBLocalHome getEJBLocalHome(int pServiceId)
    throws ServiceLocatorException{

    //Trying to find the JNDI Name for the requested service
    String serviceName = getServiceName(pServiceId);
    EJBLocalHome ejbLocalHome    = null;

    try {
      //Checking to see if I can find the EJBLocalHome interface in cache
      if (ejbLocalHomeCache.containsKey(serviceName)){
         ejbLocalHome = (EJBLocalHome) ejbLocalHomeCache.get(serviceName);
         return ejbLocalHome;
      } else {
      //If I could not find the EJBLocalHome interface in the cache,
      //look it up and then cache it
        Context ctx = new InitialContext();
        Object jndiRef = ctx.lookup(serviceName);
        ejbLocalHome = (EJBLocalHome) jndiRef;
        ejbLocalHomeCache.put(serviceName, ejbLocalHome);
        return ejbLocalHome;
      }
    } catch(NamingException e){
        throw new ServiceLocatorException(
          "Naming exception error in ServiceLocator.getEJBLocalHome()" + e);
    } catch(Exception e){
        throw new ServiceLocatorException(
         "General exception in ServiceLocator.getEJBLocalHome" + e);
    }
  }
...
...
```

然后是两个方法，分别返回与缓存JMS QueueConnection与Queue。

```
...
...
  public QueueConnection getJMSQueueConn(int pServiceId)
      throws ServiceLocatorException {

   //Getting JNDI Service Name
   String serviceName = getServiceName(pServiceId);
   QueueConnection qc = null;

   //Check to see if requested service is in cache
   try   {
      if (queueConnectionFactoryCache.containsKey(serviceName)) {
         QueueConnectionFactory qcf = (QueueConnectionFactory)
             queueConnectionFactoryCache.get(serviceName);
         qc = qcf.createQueueConnection();
      } else {
         Context ctx = new InitialContext();
         QueueConnectionFactory qcf = (QueueConnectionFactory)
             ctx.lookup(serviceName);
         queueConnectionFactoryCache.put(serviceName, qcf);
         qc = qcf.createQueueConnection();
      }

   } catch(JMSException e){
      throw new ServiceLocatorException(
          "A JMS exception has occurred in ServiceLocator.getJMSQueueConn()"
          + e);
    } catch(NamingException e){
      throw new ServiceLocatorException(
         "A JNDI Naming exception has occurred in " +
         "ServiceLocator.getJMSQueueConn()" + e);
    } catch(Exception e){
       throw new ServiceLocatorException(
          "An exception has occurred in ServiceLocator.getJMSQueueConn()"
          + e);
    }
   return qc;
  }

  public Queue getJMSQueue(int pServiceId) throws ServiceLocatorException {

   String serviceName = getServiceName(pServiceId);
   Queue newQ = null;

   try   {
      if (queueCache.containsKey(serviceName)) {
```

```
            newQ = (Queue) queueCache.get(serviceName);
          } else {
            Context ctx = new InitialContext();
            newQ = (Queue) ctx.lookup(serviceName);
          }
        } catch(NamingException e){
          throw new ServiceLocatorException(
            "A JNDI Naming exception has occurred in " +
            "ServiceLocator.getJMSQueue()" + e);
        } catch(Exception e){
          throw new ServiceLocatorException(
            "An exception has occurred in ServiceLocator.getJMSQueue()" + e);
        }
       return newQ;
      }
...
...
```

这个应用程序其余部分会大量使用这个ServiceLocator类。

Value Object模式

实体Bean混合持久性与远程性和组件接口中提供的许多get/set方法（用于持久性）。这就使开发人员要让客户机进行许多细粒调用，取得与更新实体的状态数据。前面曾介绍过，这种get/set方法并不理想，每次都需要远程调用。远程调用次数应尽量减少。

Value Object模式可以将远程调用减少到一次。数值对象（VO）将所有状态数据包装成一个可系列化对象，然后在客户机与实体之间传递。数值对象应当可系列化，因为要用作RMI参数或返回值。数值对象向远程客户机提供代表服务器上内容的本地对象图（见图4.5所示）。

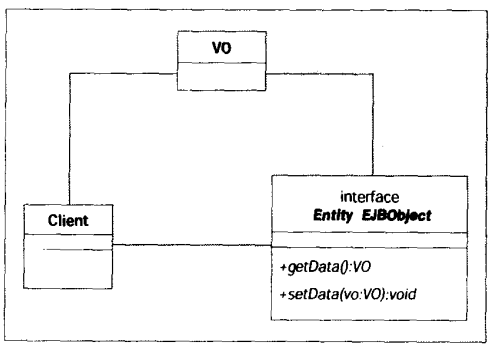

图4.5

本地接口

这时也许你会想："本地接口如何呢？"。EJB 2.0规范对本地接口形式作了大量补充。本地接口可以不用RMI而实现主接口与组件接口。换句话说，它们不实现java.rmi.Remote、

因此可以只在本地接口上指定get/set方法，从而避免远程开销。此外，本地接口还避免了按数值传递对象的性能瓶颈。利用本地接口就可以按引用传递对象。

但是，本地接口与标准VM内方法调用不大相同。本地接口避免RMI调用的开销，但EJB容器仍然要放在客户机调用与EJB对象之间，提供EJB的各种服务。因此，每个本地接口调用有自己的开销。此外，本地接口只能从EJB容器中调用。因此，在设计中，某个时候会需要进行远程调用，因为客户机是远程的。两种情况下，使用Value Object模式都可以简化设计和提高性能。

复合数值对象与数值对象汇编器

上述方法的问题在于客户机需要的信息通常不直接映射实体Bean结构。客户机通常需要来自几个实体Bean的信息。这时可以使用数值对象，包括客户机完成目标所需的信息。这种工作流特定数值对象可能把来自不同源的数据包装起来，传输到客户机。要集成这些域特定数值对象，就要使用Value Object Assembler设计模式。

Value Object Assembler设计模式可以进一步减少远程调用，将多个实体的访问合并到一个访问点。通常，汇编器用会话Bean实现，通过本地接口访问实体Bean，见Session Façade模式，如图4.6所示

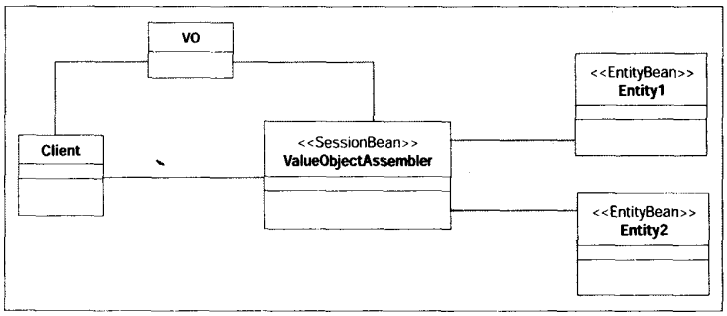

图4.6

与其在网络上来回重复，不如用一个远程调用返回需要的一切。

在城市休假应用程序中实现数值对象

城市休假应用程序有五个数值对象：

- CustomerVO
- BookingVO
- FlightVO
- HotelVO
- CityBreakVO

这些数值对象大部分与相关实体直接对应，但CityBreakVO实际上是两个实体（City与Hotel）的合并，因此要实现Value Object Assembler模式，把它们集成起来。

这些数值对象很容易实现，注意它们要实现Serializable，以便在EJB中传入和传出。

Hotel Value Object

Hotel Value Object的接口如下：

```
package vo;

public class HotelVO implements java.io.Serializable {
  public Long id;
  public String name;
  public String location;
  public int noAvailableRooms;
  public String description;
  public double pricePerNight;
}
```

Flight Value Object

Flight Value Object的接口如下：

```
package vo;

public class FlightVO implements java.io.Serializable {
  public Long flightId;
  public String from;
  public String to;
  public String departure;
  public String arrival;
  public String airline;
  public int availableSeats;
  public String date;
  public double price;
}
```

Booking Value Object

Booking Value Object实现如下：

```
package vo;

public class BookingVO implements java.io.Serializable {
  public Long hotelId;
  public Long flightIdOut;
  public Long flightIdIn;
  public Long customerId;
  public String fromDate;
  public String toDate;
  public int noSeats;
  public int noRooms;
  public double price;
}
```

Customer Value Object

Customer Value Object的接口如下：

```
package vo;

public class CustomerVO implements java.io.Serializable {
  public Long customerId;
  public String name;
  public String addr1;
  public String addr2;
  public String state;
  public String zip;
  public String email;
  public String pwd;
}
```

City Break Value Object

CityBreakVO比较复杂一些，包含HotelVO对象集合。

```
package vo;

import java.util.Collection;
import java.util.LinkedList;

public class CityBreakVO implements java.io.Serializable {

  public String cityName;
  public String cityDesc;
  public String airport;

  private LinkedList hotels_list = new LinkedList();

  public Collection getHotels() {
    return hotels_list;
  }

  public void addHotel(HotelVO vo) {
    hotels_list.add(vo);
  }
}
```

Value Object Assembler

由于CityBreakVO实际上是两个实体（City与Hotel）的集合，因此要用一个汇编程序处理City与Hotel实体Bean的连接和构造HotelVO数值对象。

```
private CityBreakVO assembleCityBreakValueObject(Long cityId) {

   CityBreakVO cbvo = new CityBreakVO();
   try {
```

```
        //First get info on the city
        ServiceLocator locator = ServiceLocator.getInstance();
        CityLocalHome chome = (CityLocalHome)
                        locator.getEJBLocalHome(ServiceLocator.CITY);
        CityLocal city = chome.findByPrimaryKey(cityId);
        cbvo.cityName = city.getName();
        cbvo.cityDesc = city.getDescription();
        cbvo.airport = city.getAirport();

        //Then get hotels for this city
        HotelLocalHome hhome = (HotelLocalHome)
                         locator.getEJBLocalHome(ServiceLocator.HOTEL);
        Collection hotels = hhome.findByLocation(cbvo.cityName);

        //For each hotel create a Hotel value object
        Iterator iter = hotels.iterator();
        while (iter.hasNext()) {
           Object ref = iter.next();
           HotelLocal hotel = (HotelLocal) ref;
           HotelVO hvo = hotel.getHotelInfo();
           cbvo.addHotel(hvo);
        }
      } catch (Exception e) {
         throw new EJBException(e);
      }
   }
```

注意我们用Service Locator模式查找EJBLocalHome对象。下列代码创建Service Locator实例并取得CityLocalHome对象：

```
ServiceLocator locator = ServiceLocator.getInstance();
CityLocalHome chome = (CityLocalHome)
                locator.getEJBLocalHome(ServiceLocator.CITY);
```

稍后将介绍这个代码在整个应用程序中的位置。

Session Façade模式

控制器或实现得到的接口要求客户机对业务层EJB多次调用，收集情境和存储调用之间的工作流与会话信息。如果把这个交互移到服务器上，则可以减少多个网络调用。Session Façade模式引入的粗粒接口可以解决这个问题。

客户机具有Session Façade的一个远程引用，而不是每个基础EJB（通常是实体Bean）多个引用。客户机处理一个对象，Session Façade提供粗粒方法（通常映射用例），只对façade进行远程调用，如图4.7所示。

远程客户机对Session Façade的调用是远程调用，但façade向支持EJB的调用可以利用VM内调用的容器提供优化，在EJB 2.0中可以通过本地接口实现。这可以用数值对象与façade的组合进一步改进。

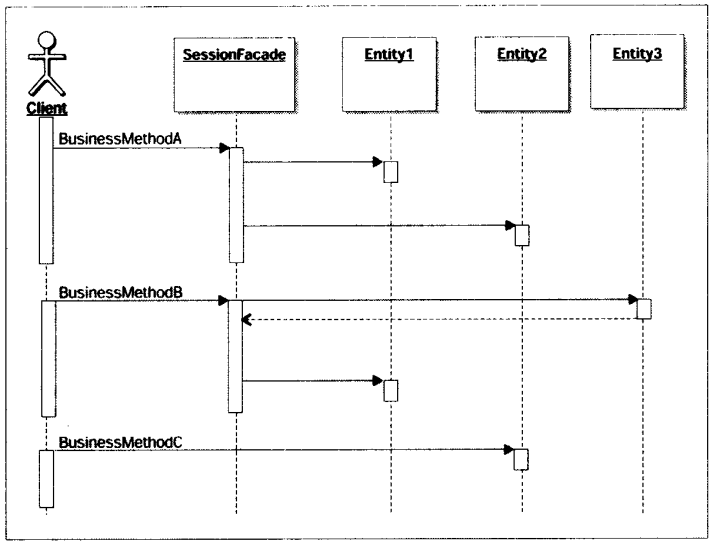

图4.7

在City Break应用程序中实现Session Façade

Session Façade提供业务接口的粗粒表示，供客户机使用。因此，一般策略是用不同Session Façade实现每个用例或相关用例。

在City Break应用程序中，客户机使用五个用例：

- View available vacations（浏览度假）
- Get a City Break vacation quote（取得城市休假配额）
- Book a City Break vacation（订购城市休假）
- Register as a new customer（注册为新客户）
- Log in as an existing customer（登记为现有客户）

可以实现五个会话Bean，对应每个用例，但通过分析可以看到，有些用例是相关的，因此我们只要构造两个Session Façade：

- CityBreakFaçade
  - View available vacations（浏览度假）
  - Get a City Break vacation quote（取得城市休假配额）
  - Book a City Break vacation（订购城市休假）
- CustomerFaçade
  - Register as a new customer（注册为新客户）
  - Log in as an existing customer（登记为现有客户）

每个Session Façade是个无状态会话Bean，用Service Locator模式查找相关实体Bean，用数值对象在会话层与实体层之间来回传递数据，如图4.8所示。

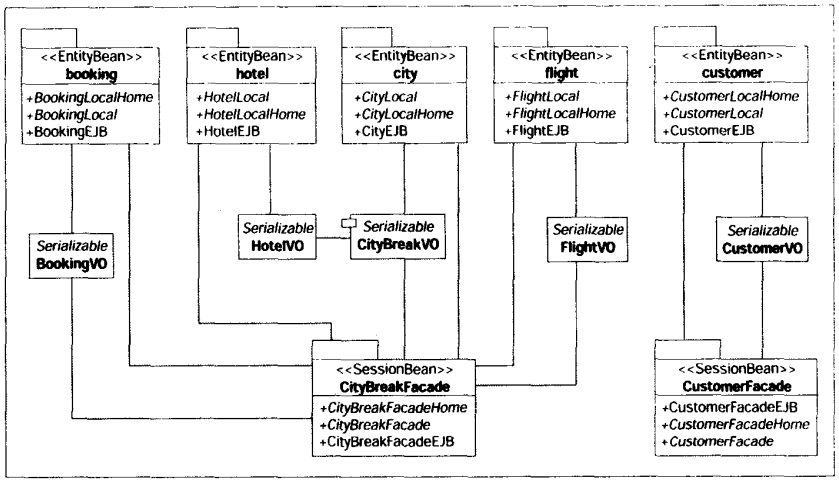

图4.8

CustomerFaçade会话Bean

CustomerFaçade会话Bean实现如下，我们要实现两个用例，因此远程接口包含这些用例的业务方法。

```
package facade.customer;

import javax.ejb.*;
import java.rmi.RemoteException;
import vo.CustomerVO;

public interface CustomerFacade extends EJBObject {

   public void registerNewCustomer(CustomerVO vo) throws RemoteException;

   public CustomerVO loginCustomer(String email, String pwd)
      throws RemoteException;
}
```

每个业务方法实现如下：

```
package facade.customer;

import javax.ejb.*;
import common.*;
import vo.CustomerVO;
import entity.customer.*;

public class CustomerFacadeEJB implements SessionBean {

   public void ejbCreate() throws CreateException {}
   public void ejbRemove() {}
   public void ejbActivate() {}
```

```
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext ctx) {}
    public void unsetSessionContext() {}

    //Create a new customer from Customer value object
    public void registerNewCustomer(CustomerVO vo) {

        try {
            ServiceLocator locator = ServiceLocator.getInstance();
            CustomerLocalHome home = (CustomerLocalHome)
                locator.getEJBLocalHome(ServiceLocator.CUSTOMER);
            home.create(vo);
        } catch (Exception e) {
            throw new EJBException (e);
        }
    }

    //Return customer details from a customer's email and password
    public CustomerVO loginCustomer(String email, String pwd) {

        CustomerVO vo = null;

        try {
            ServiceLocator locator = ServiceLocator.getInstance();
            CustomerLocalHome home = (CustomerLocalHome)
                locator.getEJBLocalHome(ServiceLocator.CUSTOMER);
            CustomerLocal cust = home.findByEmailandPassword(email, pwd);
            vo = cust.getCustomer();
        } catch (Exception e) {
            throw new EJBException(e);
        }

        return vo;
    }
}
```

CityBreakFaçade会话Bean

CityBreakFaçade会话Bean更复杂，要实现三个用例。

```
package facade.citybreak;

import javax.ejb.*;
import java.util.Hashtable;
import java.util.ArrayList;
import vo.CityBreakVO;
import java.rmi.RemoteException;

public interface CityBreakFacade extends EJBObject {
```

```
    public Hashtable getCityList() throws RemoteException;

    public CityBreakVO getInfoOnCity(Long cityId) throws RemoteException;

    public ArrayList getFlightInfoForFlight(String date, String from,
                                            String to)
        throws RemoteException;
}
```

此外，在Bean实现类中，还要使用前面介绍的Value Object Assembler模式。

```
package facade.citybreak;

import javax.ejb.*;
import common.*;
import entity.city.*;
import entity.hotel.*;
import entity.flight.*;
import vo.*;
import java.sql.*;
import java.util.Collection;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.ArrayList;
import javax.rmi.PortableRemoteObject;

public class CityBreakFacadeEJB implements SessionBean {

   public void ejbCreate() throws CreateException {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext ctx) {}
   public void unsetSessionContext() {}

   //Return a CityBreak Value object for a specific city
   public CityBreakVO getInfoOnCity(Long cityId) {

      CityBreakVO cbvo = assembleCityBreakValueObject(cityId);

      return cbvo;
   }

   private CityBreakVO assembleCityBreakValueObject(Long cityId) {
      ...
   }

   //Get flights that match a specific date between one airport and another
   public ArrayList getFlightInfoForFlight(String date, String from,
                                           String to) {
```

```
        ArrayList flights = new ArrayList();

        try   {
           ServiceLocator locator = ServiceLocator.getInstance();
           FlightLocalHome fhome = (FlightLocalHome)
               locator.getEJBLocalHome(ServiceLocator.FLIGHT);
           Collection flightmatchs =
               fhome.findByFlightDateToFrom(date, from, to);

           Iterator iter = flightmatchs.iterator();
           while (iter.hasNext()) {
              Object ref = iter.next();
              FlightLocal flight = (FlightLocal) ref;
              flights.add(flight.getFlightInfo());
           }

        } catch (Exception e) {
           throw new EJBException(e);
        }

        return flights;
     }
  }
```

这里还使用JDBC for Reading模式实现getCityList()业务方法（这个模式见ServerSide.com模式或《EJB设计模式》一书，Wiley出版（ISBN: 0-471-20831-0））。这是因为，我们只要列出数据库中的所有城市名单，用于读访问。如果使用findAll()方法，则会浪费服务器资源，因为这个一次性只读访问没有必要创建实体Bean的负载。因此，我们用会话Bean中的简单JDBC查询数据库。我们用Service Locator进行数据库查找。

```
//Return a list of cities in database
public Hashtable getCityList() {

   Hashtable tbl = new Hashtable();

   try {
      //Get connection to db
      ServiceLocator locator = ServiceLocator.getInstance();
      Connection conn = locator.getDBConn(ServiceLocator.CITYDB);
      String sql = "SELECT * FROM \"CityEJBTable\"";

      /*Executing the sql statement*/
      Statement statement = conn.createStatement();
       ResultSet rsCities = statement.executeQuery(sql);

      while (rsCities.next()) {
         Long id = new Long(rsCities.getLong("id"));
         tbl.put((Object)id, (Object)rsCities.getString("name"));
       }
```

```
        } catch (Exception e) {
            throw new EJBException(e);
        }

        return tbl;
    }
```

注意实现的CityBreakFaçade只是一个用例（BookVacation），稍后将介绍另一种实现方法。

标识提高伸缩性的模式

J2EE平台通过Java消息服务（JMS，Java Message Service）支持异步处理。尽管其在J2EE 1.2服务器中是可选的，但1.3规范中则是强制的，因为它引入了新的EJB类型：消息驱动Bean。这些会话Bean只能通过队列（点对点消息）或主题（发表/预订消息）访问。

Message Façade模式

Message Façade模式是Session Façade模式的一个版本，与Session Façade达到相同目的（将系统细节隐藏在粗粒接口后面），但还有一个好处，就是实现消息驱动Bean，从而可以异步执行，如图4.9所示。

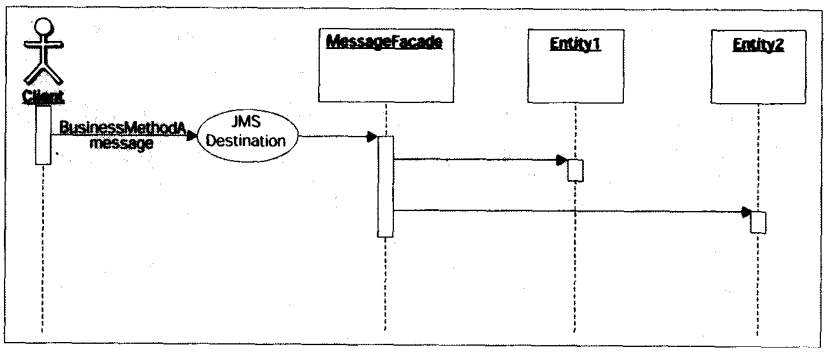

图4.9

Message Façade模式进一步提高伸缩性，还提供了组件之间的间接层。由于客户机只与一个façade对象交互，因此不知道门户后面对象与资源的位置和实现。

在City Break应用程序中实现Message Façade

订票的过程很复杂，需要创建新的Booking实体和更新Hotel、Customer与Flight实体中的值。可以将这些交互放在façade后面，因为这些更新都可以异步进行。

要创建新的Booking实体，就要用相关数据构造BookingVO数值对象，然后将对象发送到JMS队列（newBookingQ）。

```
BookingVO vo = new BookingVO();
vo.hotelId = 1;
vo.flightIdOut = 2;
vo.flightIdIn = 3;
vo.customerId = 4;
vo.fromDate = "01-01-02";
vo.toDate = "03-01-02";
vo.noSeats = 2;
vo.noRooms = 1;
vo.price = 200;

try {
   InitialContext ctx = new InitialContext();
   QueueConnectionFactory qf = (QueueConnectionFactory)
      ctx.lookup("jms/QueueConnectionFactory");
   QueueConnection qc = qf.createQueueConnection();
   QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
   Queue q = (Queue) ctx.lookup("jms/newBookingQ");
   QueueSender qSender = qs.createSender(q);
   ObjectMessage msg = qs.createObjectMessage(vo);
   qSender.send(msg);
} catch (Exception e) {
   System.out.println(e);
}
```

听取这个队列用消息驱动**Bean**，是我们实现的**Message Façade**模式。

```
package facade.bookingmsg;

import javax.ejb.*;
import javax.jms.*;
import vo.BookingVO;
import common.*;
import entity.booking.*;
import entity.flight.*;
import entity.hotel.*;
import entity.customer.*;
public class BookingMsgFacadeEJB implements MessageDrivenBean,
     MessageListener {

   public void ejbCreate() {}
   public void ejbRemove() {}
   public void setMessageDrivenContext(MessageDrivenContext ctx) {}
```

这个**Bean**的所有操作发生在**onMessage()**方法中，在新消息到达队列时由服务器调用。这个方法首先更新所有相关实体。

```
   public void onMessage(Message msg) {
```

```
QueueConnection qc = null;

try   {
   ObjectMessage omsg = (ObjectMessage) msg;
   BookingVO booking = (BookingVO) omsg.getObject();
   ServiceLocator locator = ServiceLocator.getInstance();

   //Make booking
   BookingLocalHome bhome = (BookingLocalHome)
      locator.getEJBLocalHome(ServiceLocator.BOOKING);
   BookingLocal bookingejb = bhome.create(booking);

   //Set available seats on flight Out
   FlightLocalHome fhome = (FlightLocalHome)
      locator.getEJBLocalHome(ServiceLocator.FLIGHT);
   FlightLocal flightOut =
      fhome.findByPrimaryKey(booking.flightIdOut);
   if ((flightOut.getAvailableSeats() - booking.noSeats) > 0) {
      flightOut.bookSeats(booking.noSeats);
   } else {
      throw new  Exception("Not enough seats for flight: " +
                  booking.flightIdOut.toString());
   }

   //Set available seats on flight in
   FlightLocal flightIn = fhome.findByPrimaryKey(booking.flightIdIn);
   if ((flightIn.getAvailableSeats() - booking.noSeats) > 0) {
      flightIn.bookSeats(booking.noSeats);
   } else {
      throw new
        Exception("Not enough seats for flight: " +
                                booking.flightIdIn.toString());
   }

   //Set available rooms in hotel
   HotelLocalHome hhome = (HotelLocalHome)
      locator.getEJBLocalHome(ServiceLocator.HOTEL);
   HotelLocal hotel = hhome.findByPrimaryKey(booking.hotelId);
   if ((hotel.getAvailableRooms() - booking.noRooms) > 0) {
      hotel.bookRooms(booking.noRooms);
   } else {
      throw new Exception("Not enough rooms for: " +
                               booking.hotelId.toString());
   }

   //Add booking id to customer
   CustomerLocalHome chome = (CustomerLocalHome)
      locator.getEJBLocalHome(ServiceLocator.CUSTOMER);
```

```
            CustomerLocal customer =
               chome.findByPrimaryKey(booking.customerId);
            customer.addBooking(bookingejb.getId());
```

然后为了确认预订，Bean发送订票号（新Booking实体的主键）到客户机（通过另一队列，bookingCompleteQ）。

```
            //Send booking id
            qc = locator.getJMSQueueConn(ServiceLocator.QUEUECONNFACTORY);
            Queue q = locator.getJMSQueue(ServiceLocator.BOOKING_COMPLETE);
            QueueSession qs = qc.createQueueSession(false,
                                                    Session.AUTO_ACKNOWLEDGE);
            QueueSender qSender = qs.createSender(q);
            TextMessage txtMsg = qs.createTextMessage(
                "Booking complete: id = " + bookingejb.getId());
            qSender.send(txtMsg);

         } catch (Exception e) {
            throw new EJBException(e);
         } finally {
            if (qc != null)   {
               try   {
                  qc.close();
               } catch (JMSException jmse)   {}
            }
         }

      }
   }
```

Business Delegate模式

J2EE组件最大的耦合区之一是通过资源定位机制。尽管使用JNDI方法就不需要知道特定网络地址，但客户机仍然要知道要使用和查找什么资源。结果，我们的客户机（在表示层）要自己处理使用远程对象的细节。在表示层与业务对象之间插入委托对象之后，客户机就可以调用业务特定调用而不需要知道操作和业务对象本身。Business Delegate模式如图4.10所示。

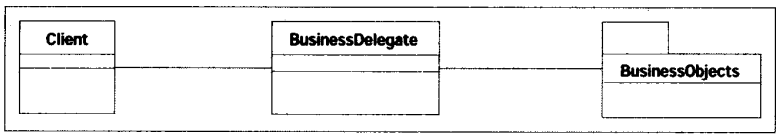

图4.10

Business Delegate模式的性能与去耦合可以进一步改善，可以组合Service Locator模式，处理从代理对业务对象的查找，如图4.11所示。

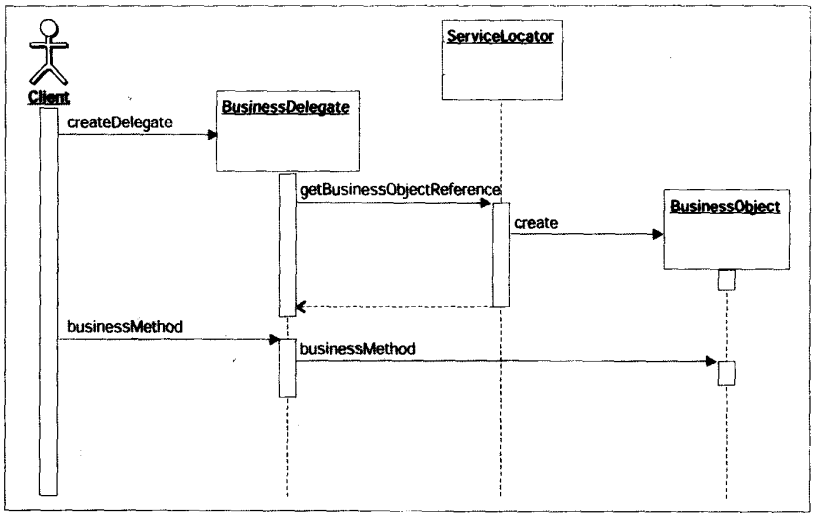

图4.11

在City Break应用程序中实现Business Delegates

我们已经通过Session Façade与Message Façade模式增加间接层。但是，要让客户机层与Session Façade交互，就要查找会话Bean（CityBreakFaçade与CustomerFaçade）。

为了用Business Delegate模式进一步分离客户机层，我们创建两个简单Java类，作为客户机层与Session Façade之间的代理。因此我们需要两个代理：CustomerDelegate代理CustomerFaçade与CityBreakDelegate代理CityBreakFaçade。

实现这些代理类很简单，因为它们只要提供业务接口，然后转发到façade。首先是CustomerDelegate。

```
package delegate;

import common.*;
import facade.customer.*;
import vo.CustomerVO;

public class CustomerDelegate {

   private static CustomerFacade cFacade = null;

   //Get the remote interface to the CustomerFacade session bean
   private static void setCustomerFacade() {

      try {
         ServiceLocator locator = ServiceLocator.getInstance();
         CustomerFacadeHome cfhome = (CustomerFacadeHome)
            locator.getEJBHome(ServiceLocator.CUSTOMERFACADE);
         cFacade = cfhome.create();
      } catch (ServiceLocatorException sle) {
```

```
            System.out.println("ServiceLocator exception thrown: " + sle);
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    //Create a new customer
    public void registerNewCustomer(String name, String addr1,
                                    String addr2, String state,
                                    String zip, String email,
                                    String password) {

        //Construct Customer Value Object
        CustomerVO vo = new CustomerVO();
        vo.name = name;
        vo.addr1 = addr1;
        vo.addr2 = addr2;
        vo.state = state;
        vo.zip = zip;
        vo.email = email;
        vo.pwd = password;

        //Connect to facade and pass on value object
        try {
            if (cFacade == null) {
                setCustomerFacade();
            }
            cFacade.registerNewCustomer(vo);

        } catch (Exception e) {
            throw new Exception(e);
        }
    }

    //Send email and password info to get customer id
    public Long loginCustomer(String email, String pwd) {

        CustomerVO vo = null;

        try {
            if (cFacade == null) {
                setCustomerFacade();
            }

            vo = cFacade.loginCustomer(email, pwd);
        } catch (Exception e) {
            System.out.println(e);
        }
```

```
        return vo.customerId;
    }
}
```

可以看出，这个代理还可以从客户端隐藏使用数值对象的细节，但可以将数值对象体系结构扩展到Web层，帮助层间进行连接。

最后是CityBreakDelegate：

```
package delegate;

import common.*;
import facade.citybreak.*;
import javax.jms.*;
import vo.*;
import java.util.Hashtable;
import java.util.ArrayList;

public class CityBreakDelegate {

   private static CityBreakFacade cbFacade = null;

   //Get the remote interface to the CityBreakFacade session bean
   private static void setCityBreakFacade() {

      try {
         ServiceLocator locator = ServiceLocator.getInstance();
         CityBreakFacadeHome cbfhome = (CityBreakFacadeHome)
            locator.getEJBHome(ServiceLocator.CITYBREAKFACADE);
         cbFacade = cbfhome.create();
      } catch (ServiceLocatorException sle) {
         System.out.println("ServiceLocator exception thrown: " + sle);
      } catch (Exception e) {
         System.out.println(e);
      }
   }

   //Return the cities available in the database
   public Hashtable getListOfCities() {

      Hashtable htble = null;

      try {
         if (cbFacade == null) {
            setCityBreakFacade();
         }
         htble = cbFacade.getCityList();

      } catch (Exception e) {
          throw new Exception(e);
      }
```

```
    return htble;
}

//Get a CityBreak Value Object for a specfic city
public CityBreakVO getInfoOnCity(Long cityId) {

    CityBreakVO vo = null;

    try {
        if (cbFacade == null) {
            setCityBreakFacade();
        }

        vo = cbFacade.getInfoOnCity(cityId);

    } catch (Exception e) {
         throw new Exception(e);
    }

    return vo;
}

//Get all flights for a specific date from one airport to another
public ArrayList getFlightInfoForFlight(String date, String from,
                                        String to) {
    ArrayList flights = null;
    try   {
        if (cbFacade == null) {
            setCityBreakFacade();
        }
        flights = cbFacade.getFlightInfoForFlight(date, from, to);
    } catch (Exception e) {
         throw new Exception(e);
    }
    return flights;
}

//Send a Booking Value Object to book a city break
public void bookCityBreak(Long hotelId, Long flightIdOut,
                          Long flightIdIn, Long customerId,
                          String fromDate, String toDate,
                          int noSeats, int noRooms, double price) {

    //Create value object and load with parameters
    BookingVO vo = new BookingVO();
    vo.hotelId = hotelId;
    vo.flightIdOut = flightIdOut;
    vo.flightIdIn = flightIdIn;
    vo.customerId = customerId;
```

```
            vo.fromDate = fromDate;
            vo.toDate = toDate;
            vo.noSeats = noSeats;
            vo.noRooms = noRooms;
            vo.price = price;

            QueueConnection qc = null;

            //Create a connection to the booking queue and send the value object
            try {
               ServiceLocator locator = ServiceLocator.getInstance();
               qc = locator.getJMSQueueConn(ServiceLocator.QUEUECONNFACTORY);
               QueueSession qs = qc.createQueueSession(false,
                                                   Session.AUTO_ACKNOWLEDGE);
               Queue q = locator.getJMSQueue(ServiceLocator.NEW_BOOKINGS);
               QueueSender qSender = qs.createSender(q);
               ObjectMessage msg = qs.createObjectMessage(vo);
               qSender.send(msg);

            } catch (ServiceLocatorException sle) {
                throw new Exception("ServiceLocator exception thrown: " + sle);
            } catch (JMSException jmse) {
                throw new Exception("JMS Exception thrown: " + jmse);
            } catch (Exception e) {
                throw new Exception(e);
            } finally {
               if (qc != null)   {
                  try   {
                     qc.close();
                  } catch (JMSException jmse)   {}
               }
            }
         }
      }
```

完整City Break体系结构

下面看看完整City Break应用程序体系结构，我们把基本过程分解为框图4.12。

客户过程

创建新客户和登录现有客户使用下列模式：

- Business Delegate模式（CustomerDelegate）
- Session Façade模式（CustomerFaçade）
- Value Object模式（CustomerVO）

- Service Locator模式（ServiceLocator）

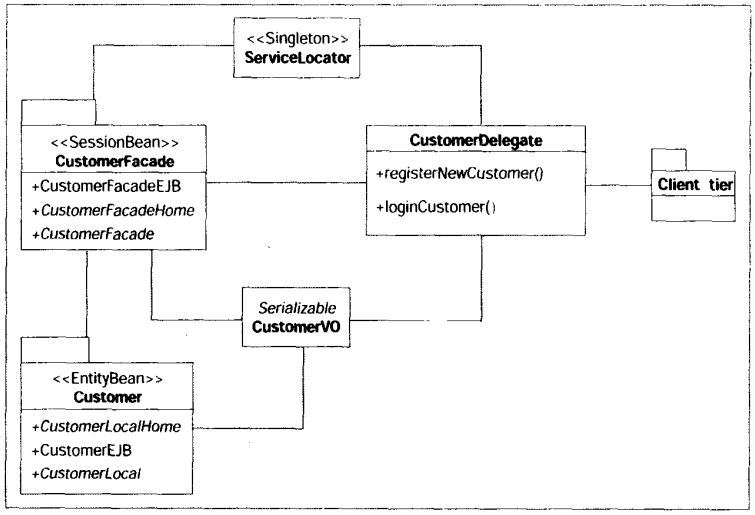

图4.12

配额过程

为了集成City Break假期，我们使用与客户过程相似的体系结构，但更加复杂，因为涉及的实体更多，如图4.13所示。我们使用下列模式：

- Business Delegate模式（CityBreakDelegate）
- Session Façade模式（CityBreakFaçade）
- Value Object模式（HotelVO、FlightVO和CityBreakVO）
- Value Object Assembler模式（CityBreakFaçade.assembleCityBreakValueObject()）
- JDBC for Reading模式（CityBreakFaçade）
- Service Locator模式（ServiceLocator）

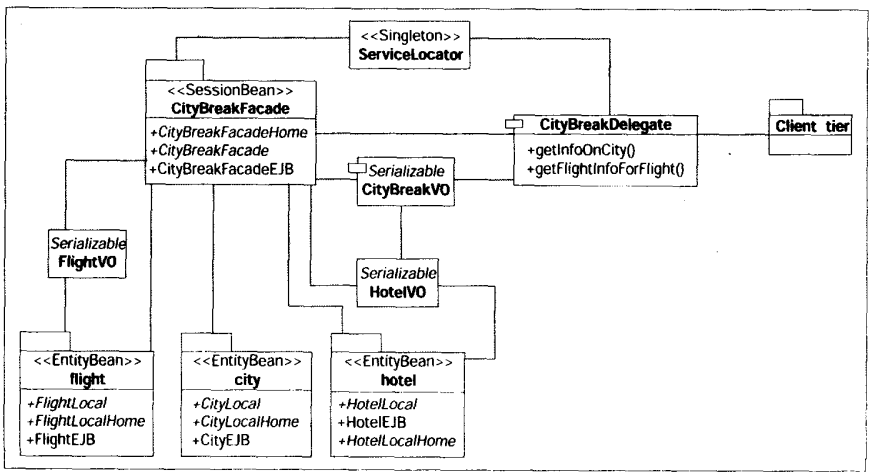

图4.13

订票过程

订票过程使用下列模式，如图4.14所示：

- Business Delegate模式（CityBreakDelegate）
- Message Façade模式（BookingMsgFacade）
- Value Object模式（BookingVO）
- Service Locator模式（ServiceLocator）

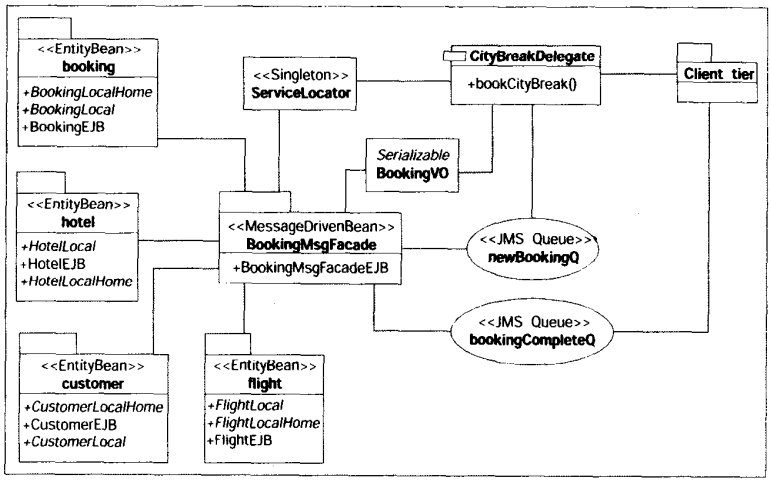

图4.14

小结

性能与伸缩性是两个重要的设计考虑，不仅设计J2EE应用程序时如此，任何分布式系统开发也都如此。设计不好的系统可运行性能很差，再多的硬件也无济于事。尽管J2EE模式类别中没有针对性能与伸缩性的模式，但有些模式可以解决性能与伸缩性问题。

本章介绍了如何在应用程序中用模式解决性能与伸缩性问题。我们首先分析各种性能与伸缩性问题，然后介绍这些设计模式在City Break Booking应用程序中的应用。

我们介绍的性能模式包括：

- **Service Locator模式**

  这个模式用缓存机制存储工厂对象的初始情境对象，避免太多JNDI查找

- **Value Object模式**

  这个模式将所有状态数据包装成一个可序列化对象，从而减少远程调用次数

- **Session Façade模式**

  这个模式引入粗粒接口，减少多个网络调用

我们介绍的伸缩性模式包括：

- **Message Façade模式**

  这个模式类似于Session Façade模式，但实现消息Bean，因此还能异步执行

- **Business Delegate模式**

  这个模式处理使用远程对象的细节。在表示层与业务对象之间插入委托对象，从而减少客户机了解业务对象的开销

下一章介绍的设计模式解决安全问题。

第5章 管理安全性的设计模式

本章通过J2EE Web Banking应用程序案例介绍解决安全问题的设计模式及其益处，定义这个应用程序的范围与要求，标识相关安全模式，并用于应用程序和操作环境的设计。我们要开发用例，最后要显示几个主要Java类的代码。

何谓安全模式

安全模式可以解决安全问题，J2EE和其他面向对象模式提供了解决已知编程问题的可靠技术。

安全模式组成一组最佳做法，最终目的是支持机构的安全政策，不仅解决应用程序安全性，而且解决主机与网络安全。这样，安全模式应用到应用程序设计与开发，主机配置与管理和这些应用程序所在的网络。但是，安全模式没有定义特定技术、编码风格和编程语言，没有标识行业厂家、应用程序版本号或补包层。

使用安全模式的好处

和标准面向对象模式相似，安全模式提供下列好处：

- 可以随时修改与实现，改进系统设计
- 使经验较少的人可以向经验较多的人学习
- 提供讨论、测试与开发的共同语言
- 很容易分类、搜索和改变
- 提供可复用、可重复、建档的安全做法

何时使用安全模式

安全模式可以在处理下列问题时提供准则：

- 接收和发送数据到外部系统、应用程序或对象时：
  - 是否根据长度、数值或类型验证信息？
  - 通信信道安全否？需要安全否？
  - 数据来自哪里，是信任源还是非信任源？
- 应用程序由信任或非信任用户访问时：
  - 谁访问应用程序？
  - 请求合法否？有没有关系？
  - 它知道如何处理请求否？如果不知道，该怎么办？
  - 它知道每个试图访问否？怎么确定？
- 数据保密或敏感时：

- 如何保护数据？
- 这些方法充分否？有无保证？
- 数据是否在其他地方存储或备份。这样足够否？

安全编程

前面曾介绍过，安全模式实际上是最佳做法，可以帮助设计安全应用程序。但是，安全模式无法代替安全编程技术。在所有语言中采用正确的编程标准是开发容错软件的关键。下面是几个正确编码的例子：

- 数据检验
- 代码、设计检查
- 确定范围
- 同步操作
- 安全（动态）类负载
- 正确的异常处理、错误报告和日志

Web银行案例分析

这个案例分析的目的是对Web银行应用程序设计采用安全模式。这个应用程序基于J2EE Web应用程序，作为现有银行系统的前端。我们标识应用程序的特性，然后定义关键业务和技术要求。

高级概述

Wrox银行是个国家银行，具有分支办事处和自动柜员机（ATM），分布在全国各地，利用现有计算（大型机）基础结构。客户要求银行提供联机银行服务。最近统计表明，下面三个服务对客户最重要：

- 浏览账号结余
- 浏览账号活动
- 账户间转钱

假设

可以对这个案例分析作下列假设：

- 现有计算（大型机）基础结构由信任大型机系统构成，能支持这个Web应用程序产生的所有活动。
- 通过专用高速网络连接后端大型机。
- Web账号（包括用户名和口令）在分支机构地址创建，不在本联机应用程序范围之内。

业务要求

业务要求定义应用程序的特性与服务。

Wrox Web Banking应用程序是基于Web的，可以用标准Web浏览器在Internet上访问

（这时还不支持无线设备）。它可以是现有银行基础结构的前端，即不重复大型机的核心账号信息。

这个应用程序有三类用户：

- 匿名用户只访问Web站点的公开页面，无法登录，不能进行任何银行活动。
- 普通客户进行下列活动：
  - 登录应用程序：通过窗体验证之后，应用程序创建用户会话，让客户访问其他服务。
  - 退出应用程序：终止用户会话。
  - 浏览账号结余，登录之后，应用程序显示客户活动账户清单及其结余。
- 优先客户可以进行普通客户的所有活动，还可以进行下列活动：
  - 浏览账号活动：从最近事务开始，显示最近（30天内）的事务。这时，客户不能浏览在此之前的活动。表5.1指定了每个事务显示的字段。

表5.1

字段	描述
Date	指定后端系统处理事务的日期和时间
Description	提供事务描述
Type	这个字段包含下列值之一：ATM、Cheque、Web、Service Charge或Other。ATM 指ATM上处理的事务。Cheque指用支票进行借贷。Service Charge指银行进行的事务，Other指所有其他类型事务
Reference Number	对Cheque类型事务指定支票号，否则为参考号（由后端系统产生）
Amount	指定取出的金额（负数）或存入的金额（正数）

- 活动账户之间转钱：这个服务使优先客户可以在活动账号之间转钱。所转金额没有限制，只要取钱账号有足够资金。转账只能在两个活动账号之间发生。

默认情况下，只有主支票/存款账号是活动账号。其他账号（如信用卡和信用线账号）只在分支地址的客户请求下才成为活动账号。客户只能对授权账号浏览账号信息，不能访问其他客户的账户信息。保密信息不会不加保护在非信任网络上传输。如果24小时内连续三次输入错误的口令，则用户账户锁定，直到下一个工作日。用户会话在10分钟不活动之后到期。

技术要求

技术要求也称为非功能要求，定义性能、伸缩性、可用性等问题。应用程序包括下列层，每一层放在不同网段，用防火墙保护，只允许重要网络活动通过。

- 表示（Web）：包括负载平衡器和Web服务器。
- 应用程序：支持应用程序服务器群集。

- 数据层：用户（数据）库包含至少下列信息：
  - 客户ID：后端大型机系统指定的惟一客户标识符
  - 用户名：应用程序向客户提供的名称
  - 全名：大型机系统中列出的姓与名
  - 口令：应用程序向客户提供的口令
  - 角色指定：由后端系统指定
  - 活动账户清单：Web银行应用程序客户能访问的账户清单

后端银行系统是所有其余客户信息的权威数据源，包括：

- 个人客户信息（全名、地址，等等）。
- 客户账号。
- 客户账户结余和历史。

通过HTTP发送给客户的所有保密信息都用128位SSL加密传输。Web基础结构支持高可用性，利用负载平衡并群集Web、应用程序与数据库服务器。每个网络设备和应用程序用最新补包巩固和配置。测试与预备环境配置成尽量接近的产品环境。只有授权人员才能访问产品和预备服务器。

安全模式

下面标识并介绍这个案例使用的安全模式。根据这个应用程序的范围，业务要求与技术要求，我们找出使应用程序更安全的安全模式。

Single Access Point模式

Single Access Point模式描述的系统只有一个入口，所有输入请求的入口是相同的，这是进入系统的惟一通道。所有用户（和其他应用程序）请求访问时都要先经过这个入口。采用Single Access Point模式后，应用程序可以保证任何用户都得到基本验证，不能回避任何身份检查。例如，对保护资源的非法请求可以自动改向到这个访问点进行正确检验。

这个模式通常向机构的网络与个人服务器提供单个登录提示。另一个例子是应用程序提供单个登录页，而不是每个服务提供不同登录页。

大型复杂应用程序更能利用这种模式。例如，J2EE应用程序具有扩展性，可能加进许多不同应用程序，如消息、规则引擎、和报表应用程序，每个应用程序可能都需要某种形式的用户验证。与其对每个辅助应用程序开发登录页面，不如用Single Access Point模式一次性收集用户信息，然后转发这个信息。

J2EE安全模型提供了实现Single Access Point模式的简单机制。在J2EE Web层采用声明式安全性之后，设计人员可以指定要保护的Web资源（URL、URL模式和HTTP方法）。匿名用户请求保护资源时，应用程序（Web容器）通过几种机制验证用户。J2EE平台自然支持基本验证、窗体验证和客户端证书。Wrox Web Banking应用程序采用窗体验证。

Check Point模式

Check Point模式集中并执行验证与授权，这个机制负责确定用户是否有足够权限访问请求的资源。通过集中这个逻辑，Check Point模式还可以更方便地管理应用程序政策和业

务规则。设计人员可以修改和扩展这些规则而不改变其余应用程序。Check Point模式适用于下列情形：

- 多层安全性的系统。为了最初进行访问，用户输入基本用户名和口令。Check Point模式让其访问匹配这级安全性的所有资源。要达到更高安全性，访问更敏感的信息，用户要提供更强的安全证明，如数字证书。另一个强验证机制是生理标识。Check Point模式管理资源的安全要求，确定安全级，并根据预定安全政策向用户提供访问。
- 多个用户库的系统，有些用户的验证信息存放在LDAP目录中，有些存放在RDBMS中，而有些存放在大型机系统中。Check Point模式负责所有这些用户库的通信和检验用户。
- 应用程序当前只通过HTTP浏览器访问，但要将服务扩展到IVR、移动电话、PDA等最终用户设备。可以创建网关，将这些设备的请求转发到应用程序，通过Check Point模式清理设备请求，保证安全性。

注意，简单应用程序可能只在单个校验点采用其中一种或两种情形，而更复杂的系统则可能有多个校验点，可能多次使用上述每种情形。例如，跨国公司可能在每个国家采用校验点，保证执行这个国家的安全政策，还可能加入地区校验点，支持本地最终用户设备。

Check Point模式的大部分功能可以在J2EE安全体系结构中用Java验证与授权服务（JAAS，Java Authentication and Authorization Service）自然支持。JAAS提供可插入和可堆叠验证。可插入验证模块（PAM，pluggable authentication module）框架从基础验证代码抽象应用程序，从而删除依赖性，提供更大的灵活性和验证机制选择。

PAM还提供可堆叠验证，定义任何特定机制何时为可选、必要和充分（‘optional’，‘required’与‘sufficient’）。在上述讨论中，系统具有多个安全级，需要用户顺利验证才能达到更高安全级。要访问低安全级资源，就不需要进一步验证，因为它们已经拥有足够权限。

Role模式

Role模式将用户与权限进行分离。用户指定为一个或几个角色，每个角色提供一个或多个权限。用户可以在责任改变时指定角色或从角色中删除，出现新资源和新对象时，角色可以提供新权限。这个方法提供了两个非常重要的好处：

- 更新角色定义和用户权限指定时，不需要修改基础访问控制对象。
- 管理权限更直观，因为角色对应于公司的组织结构和应用程序用户类型，如开发人员、管理员、经理、银牌、金牌与白金客户，等等。

Role模式的最复杂实现称为基于角色访问控制（RBAC，Role-Based Access Control），扩展Role模式，采用下列概念：

- 继承（或角色继承）：角色总权限等于该角色权限加上相关角色的权限。
- 分开责任：静态、动态与机构性分开责任，表示实际中各个人不能同时属于两个角色，此外也不能属于某些角色。例如，银行经理不能是审计员，银行客户不能是收银员。

Wrox Web Banking应用程序的业务要求确定了三种角色：Anonymous User、Regular Customer和Preferred Customer，各有相关的权限。具体地说，Anonymous user可以浏览所有公开

页面，Regular Customer可以浏览所有公开页面并能浏览账号结余，Preferred Customer具有Regular Customer的所有权限，还可以在账户间转钱和浏览账户历史。

J2EE平台将角色加进体系结构中，采用声明式安全模型。简单的角色访问控制可以在ejb-jar.xml与web.xml部署描述项中定义角色（用声明式语法）。请求保护资源时，验证用户，Web容器或企业bean引用角色声明，允许或拒绝访问Web资源。

风险评估与管理

安全就是风险管理。风险评估与管理确定保护应用程序及其资源所需的适当和合理工作，是安全分析的第一步，目的是以安全方式完成工作而不占用太多资源、过量工程代码或不必要地加密公开信息。可以说，风险与下面三个因素成正比：

- 威胁：成功的频率
- 脆弱性：成功的可能性
- 成本：破坏的成本或资源的价值

这些因素越大，总体风险就越大。正确的风险评估保证应用程序不仅得到正确保护，而且其直接或间接影响的每个系统也得到保护。例如，Wrox Web Banking应用程序的Web服务器是个攻击目标，不是因为其中包含敏感信息，而是因为可以通过其对后端银行系统等目标进行更深入更致命的攻击（当然，可以取得Web服务器上的信息和用于进一步攻击）。

显然，这些前端应用程序的风险很高。其弱点取决于应用程序加密的程度，可能很低。破坏应用程序的成本包括代码与客户数据被窃和发现与修复所花的时间。

权威数据源

应用程序如果盲目接受任何来源的数据，则可能处理过时或虚假的数据。因此，应用程序要从许多来源中识别数据的权威性。了解权威数据源就是识别数据来源，知道这个信息的可信程度。简而言之，不能够未经检验而轻信数据的有效性。

大多数情况下，确定权威数据源是业务过程拥有者的工作。拥有者比应用程序设计人员和开发人员更了解在环境中的信息用途。但是，信息安全小组和应用程序设计人员仍然要向业主说明所考虑数据源的有效性和完整性。

尽管Wrox Web Banking应用程序的界面不同（ATM或柜员），但账户信息都是从后端大型机系统取得的。本地重复客户的个人信息和账户信息会在现阶段带来不必要的技术问题和过程问题。

因此，银行应用程序从大型机系统访问所有客户账户信息。本地数据库中只存储与Web银行系统有关的信息，如Web银行用户名、口令和角色指定。客户账户信息的权威数据源是后端大型机系统，而Web银行用户名与口令权威数据源是本地数据库。

权威数据源还要求检验从用户或系统收到的信息。本例中，用户从窗体输入，其中包含用户名、口令、金额，每个数据字段至少要检验字符类型和长度。至少要在服务器方用小服务程序进行这个检验。此外，客户端检验（例如用JavaScript）可以先检查信息再将其提交到服务器中。

Wrox Web Banking用例

确定功能与技术要求和介绍几个安全模式之后，下面要介绍这个应用程序的用例。

角色

根据功能要求，角色为Anonymous users, Regular Customers与 Preferred Customers。

用例

这个应用程序的用例包括：

- View public pages（浏览公开页面）
- Log in to the application（登录应用程序）
- Log out of the application（退出应用程序）
- View account balances（浏览账户结余）
- View account activity（浏览账户活动）
- Transfer funds between Wrox accounts（在账户间转钱）

根据上述用例，可以看看下列用例框图5.1所示。

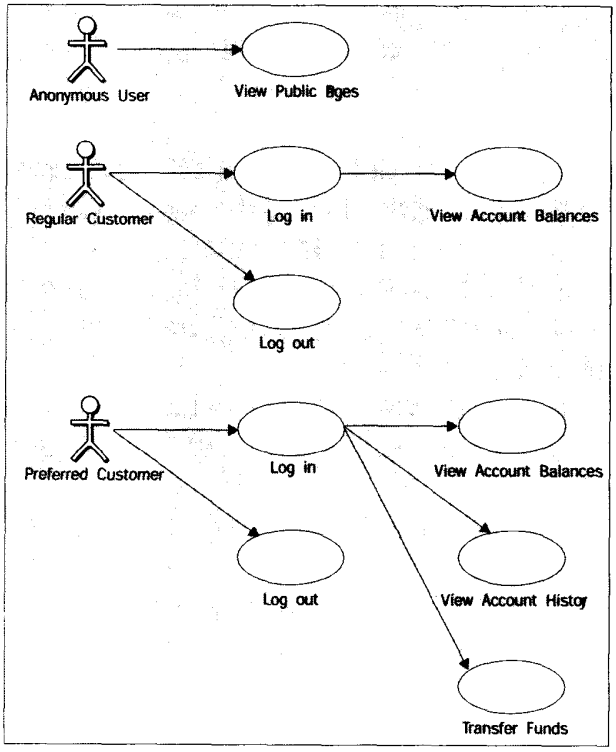

图5.1

用例框图显示Preferred Customer能访问Regular Customer的所有服务，还能访问View account activity与Transfer funds服务。

View Public Pages

这是个普通用例，表示任何非授权用户都可以浏览Web站点的公开页面。

主事件流

- 公开页面表示Web站点中不需要验证的部分
- 匿名用户可以浏览Web站点的公开页面

Log用例

Regular Customer和Preferred Customer在访问任何Wrox Web Banking服务时使用这个用例。

主事件流

提示客户输入用户名和口令，成功登录后，创建会话。

- 提示客户输入用户名和口令。
- 客户输入用户名和口令并提交信息。
- 应用程序首先用本地数据库检验用户名。
- 如果找到用户名，则应用程序用本地数据库检验口令。如果口令有效，则应用程序创建会话。
- 应用程序从本地数据库取得下列信息：用户名、姓和名、角色指定、活动账户清单，并将这个信息存放在客户的会话对象中。

其他事件流

- 如果找不到用户名，则显示下列错误消息："Invalid username or password, please try again"。
- 如果口令不匹配，则显示下列错误消息："Invalid username or password, please try again"。系统记录这个失败登录请求。如果是第三次失败登录请求，则账户锁定，直到下一个工作日（注意用户名与口令无效时显示相同的错误消息，防止攻击者根据返回的错误消息猜测用户名）。
- 如果应用程序无法创建用户会话，则显示下列错误消息："We are unable to process your request"。系统日志文件中登记更详细的错误消息，以便检查。

Logout用例

Preferred Customer与Regular Customer完成应用程序工作之后使用这个用例。

主事件流

客户顺利验证应用程序之后开始这个用例。客户选择Logout按钮并退出应用程序。

- 客户选择Logout按钮。
- 应用程序终止用户会话。
- 客户改向到表示退出成功的确认页面。

其他事件流

十分钟非活动时间过后，应用程序终止用户会话。

View Account Balances用例

Preferred Customer与Regular Customer用这个用例浏览活动账户的账户结余。

主事件流

客户顺利验证应用程序之后开始这个用例。应用程序取得并显示所有活动账户的账户结余：

- 应用程序检验客户的有效会话，如果没有，则改向到登录页面。
- 应用程序从用户会话对象取得客户的活动账户清单，并检验其角色指定。
- 应用程序从后端系统请求账户结余信息。
- 如果是Preferred Customer，则应用程序提供浏览账户历史的超链接。
- 应用程序向客户显示信息。

其他事件流

- 如果客户没有任何活动账户，则显示下列消息："You do not currently have any active accounts"。
- 如果应用程序无法取得客户的活动账户，则显示下列错误消息："The application is currently unable to process your request, please try again shortly"系统日志文件中登记更详细的错误消息，以便检查。

View Account Activity用例

Preferred Customer用这个用例浏览账户活动。

主事件流

Preferred Customer从账户结余清单中选择账户时开始这个用例。应用程序取得账户活动信息并显示：

- 客户从账户结余清单中选择账户。
- 应用程序检验客户的有效会话，如果没有，则改向到登录页面。
- 应用程序验证客户为Preferred角色，否则显示下列错误："You are currently not allowed to perform this function"，并登记更详细的消息。应用程序从后端系统取得账户活动信息并显示。

其他事件流

- 如果客户没有任何账户历史，则显示下列消息："You do not currently have any active accounts"。
- 如果应用程序无法取得客户的账户历史，则显示下列错误消息："The application is currently unable to process your request, please try again shortly" 系统日志文件中登记更详细的错误消息，以便检查。

Transfer Funds用例

Preferred Customer用这个用例在账户之间转钱。

主事件流

Preferred Customer看到账户结余清单时开始这个用例。Preferred Customer选择在两个账户之间转钱。应用程序检验源账户有足够的钱，从源账户取钱并存入目标账户，后端系统记录事务并产生一个事务号：

- 客户选择在两个活动账户之间转钱。
- 应用程序检验客户的有效会话，如果没有，则改向到登录页面。
- 应用程序验证客户为Preferred角色。
- 客户选择两个账户及要转的金额。
- 应用程序检验源账户有足够的钱。
- 应用程序向后端系统发出转钱请求。
- 后端系统转钱并返回状态码，还可以返回事务记录。
- 应用程序向客户显示确认消息和事务记录。

其他事件流

- 如果用户不是Preferred Customer，否则显示下列错误消息："You are currently not allowed to perform this function"，并登记更详细的消息。
- 如果用户Preferred Customer的源账户没有足够的钱，则显示下列错误消息："Sorry, you do not have sufficient funds to perform this transfer"。
- 如果后端系统遇到错误并返回错误消息，则显示这个消息。并登记更详细的消息。

实现案例

下面几节要用标准J2EE组件实现案例。

- Web层用控制器小服务、一组JSP页面和帮助类实现。
- 业务层用Enterprise JavaBean组件实现。
- Web层与EJB层用业务代理组件接口。

设计数据库

系统中的域对象包括：

- User

  授权使用系统的用户。

- User Role

  用户指定的不同角色，这里为Regular 与 Preferred Customers角色。

- Account

  用户持有的账户细节。

- Account Detail

  每个账户的事务细节。

图5.2显示了系统中的域对象模型。

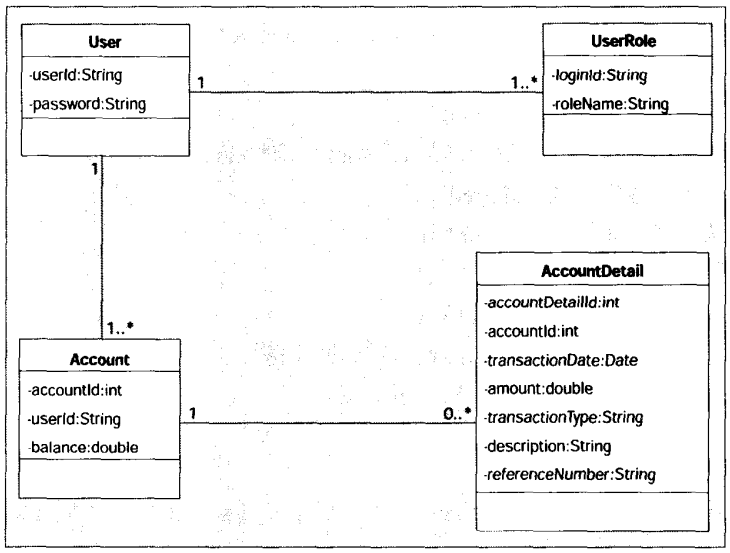

图5.2

图5.2显示了下列细节：

- 用户有用户ID和口令，可能有一个或几个角色，一个或几个账户。
- 账户可以显示用户的期末结余，可能有0个或多个相关事务。
- 账户细节具有事务日期、涉及金额、事务类型、描述和参考号。

域对象保存在关系型数据库表格中，下列脚本显示了创建表格的SQL命令，并增加一些样本数据：

```
DROP TABLE WR_ACCOUNT;
CREATE TABLE WR_ACCOUNT (
  id NUMBER PRIMARY KEY,
  user_id NUMBER,
  balance NUMBER);
```

```
DROP TABLE WR_ACCOUNT_DETAIL;
CREATE TABLE WR_ACCOUNT_DETAIL (
  id NUMBER PRIMARY KEY,
  account_id NUMBER,
  transaction_date DATE,
  amount NUMBER,
  transaction_type VARCHAR(30),
  description VARCHAR(60),
  ref_num VARCHAR(30));

DROP TABLE WR_USER;
CREATE TABLE WR_USER (
  id NUMBER PRIMARY KEY,
  login_id VARCHAR2(30),
  pwd  VARCHAR2(30));

DROP TABLE WR_USER_ROLE;
CREATE TABLE WR_USER_ROLE (
  id NUMBER PRIMARY KEY,
  login_id VARCHAR2(30),
  role_name VARCHAR2(30),
  role_group VARCHAR2(30));

INSERT INTO WR_USER VALUES (1, 'zola', 'striker');
INSERT INTO WR_USER VALUES (2, 'gallas', 'defender');

INSERT INTO WR_USER_ROLE VALUES (1, 'zola', 'standard', 'Roles');
INSERT INTO WR_USER_ROLE VALUES (2, 'gallas', 'preferred', 'Roles');

INSERT INTO WR_ACCOUNT VALUES (1, 1, 1000.00);
INSERT INTO WR_ACCOUNT VALUES (2, 2, 20000.00);

INSERT INTO WR_ACCOUNT_DETAIL VALUES (1, 1, '12-Dec-2000', 1000.00, 'Paid in by
cheque', 'No Description', '012345');
INSERT INTO WR_ACCOUNT_DETAIL VALUES (2, 2, '12-Dec-2000', 20000.00, 'Paid in by
cheque', 'No Description', '123456');
INSERT INTO WR_ACCOUNT_DETAIL VALUES (3, 2, '12-Dec-2000', 1000.00, 'Paid in by
cash', 'No Description', '234567');
INSERT INTO WR_ACCOUNT_DETAIL VALUES (4, 2, '12-Dec-2000', -1000.00, 'Paid out by
cheque', 'No Description', '345678');

COMMIT;
```

应用程序体系结构

用户通过浏览器访问系统时，系统验证用户的安全信息。完成验证之后，系统显示优先客户的本身细节和标准客户的账户结余，优先客户也可以浏览账户结余。

图5.3显示组件框图描述了高层应用程序体系结构。

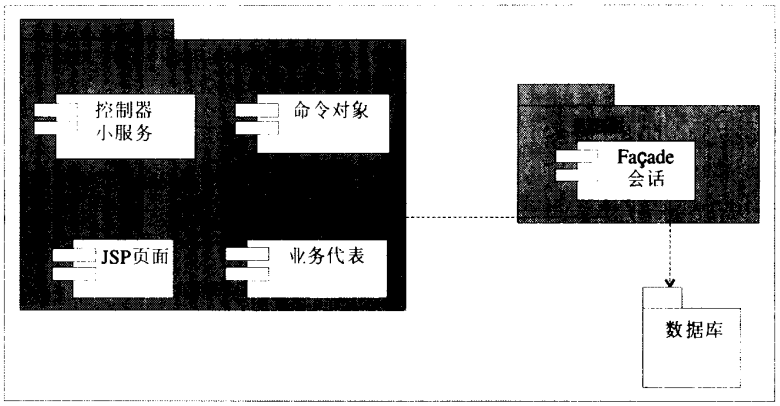

图5.3

- 客户机浏览器的所有HTTP请求由控制器小服务截获。
- 这个小服务基于请求中的一些信息选择处理请求的命令对象。
- 如果处理请求涉及利用EJB层提供的服务，则命令对象访问Session Façade，提供使用Business Delegates的业务服务。
- Session Façade用JDBC从关系型数据库返回数据。
- 命令对象将这个数据存放成请求范围Bean，将绘制下一个视图的JSP页面告诉控制器小服务。
- JSP页面从请求范围Bean取得数据并显示。

应用程序安全性

用J2EE开发应用程序时最重要的方面之一是这些应用程序可以利用所在容器提供系统级服务。这些由J2EE、EJB与小服务规范定义的系统级服务中包括安全服务。J2EE使应用程序可以用部署描述项声明式定义安全政策。小服务和EJB API还可以访问与安全相关的各个方面。因此可以用标准J2EE API和部署描述项特性实现前面介绍的大多数模式。

Single Access Point与Check Point模式

在应用程序中，所有公开请求映射控制器小服务，只有验证的用户才能访问控制器小服务。

应用程序用基于窗体登录验证用户。

- **API特性：**

  javax.servlet.http.HttpServletRequest中的getRemoteUser()方法返回当前验证用户。

  javax.servlet.http.HttpServletRequest中的getAuthType()方法标识使用的验证机制为BASIC, FORM, DIGEST或CLIENT_CERT。

  javax.ejb.EJBContext中的getCallerPrincipal()方法返回与当前执行线程的安全证明相关的主体。

javax.servlet.http.HttpServletRequest中的getRemoteUser()方法返回当前验证用户。

- **部署描述项特性：**

  Web部署描述项中的<login-config>元素指定登录机制。

  EJB部署描述项中的<method-permission>元素只让验证线程访问EJB方法。

Role模式

处理初始登录的命令对象根据验证用户角色确定哪个JSP向用户显示下一视图。

所有JSP页面存放在WEB-INF目录中，不能从浏览器中直接访问。

- **API特性：**

  javax.servlet.http.HttpServletRequest中的isUserInRole()方法确定当前验证用户是否属于指定角色。

  javax.ejb.EJBContext中的isCallerInRole()方法提供相同功能。

- **部署描述项特性：**

  Web部署描述项中的<security-constraint>、<auth-constraint>与<security-role>元素指定角色对URI模式的访问。

  EJB部署描述项中的<method-permission>与<security-role>元素指定角色访问EJB方法的权限。

  Web与EJB部署描述项中的<security-role-ref>与<role-link>元素将容器环境中定义的角色映射到EJB与Web组件中的编码角色。

会话

我们用服务器提供的特性管理应用程序会话。

- **API特性：**

  Web组件可以用javax.servlet.http.HttpSession完成会话功能。

  EJB组件可以用状态会话Bean完成会话功能。

- **部署描述项特性：**

  Web部署描述项中的<session-config>元素指定会话的最大非活动间隔。

除了上述特性之外，J2EE还提供JAAS（Java Authentication and Authorisation Service），实现可插入安全模块。本章稍后将介绍如何用容器提供者提供的基于JAAS模块的RDBMS实现部署描述项中定义的安全政策。

系统使用的角色

系统使用的角色包括：

- Standard：这个角色的用户只能浏览账户当前结余。
- Preferred：这个角色的用户可以浏览账户和事务报告。

请求URI

表5.2列出了系统提供的请求URI和访问这些请求URI所需的角色。

表5.2

URI	角色	描述
index.jsp	Standard, Preferred	访问的第一页
home.do	Standard, Preferred	主页URI
balance.do	Standard, Preferred	当前结余URI
statement.do	Preferred	事务报告URI

Web部署描述项

下面介绍Web层如何用部署描述项定义安全政策：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

    <display-name>WROX Internet Banking Application</display-name>

    <!-- Define the controller servlet -->
    <servlet>
        <servlet-name>Controller</servlet-name>
        <servlet-class>web.ControllerServlet</servlet-class>
        <!-- Map roles to coded values -->
        <security-role-ref>
            <role-name>PRF</role-name>
            <role-link>preferred</role-link>
        </security-role-ref>
        <security-role-ref>
            <role-name>STD</role-name>
            <role-link>standard</role-link>
        </security-role-ref>
    </servlet>

    <!-- Map all the *.do requests to the controller servlet -->
    <servlet-mapping>
<servlet-name>Controller</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <!-- Set the maximum inactive interval to thirty minutes -->
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>

    <!-- Define the home page -->
```

```
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
```

这个安全限制表示资源可以由**Standard**与**Preferred**角色的用户访问：

```
<security-constraint>
    <display-name>WROX Bank Security</display-name>
    <web-resource-collection>
        <web-resource-name>Standard\Preferred Customers</web-resource-name>
        <description>Resource to get bakance and login</description>
        <url-pattern>/balance.do</url-pattern>
        <url-pattern>/home.do</url-pattern>
        <url-pattern>/index.jsp</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>preferred</role-name>
        <role-name>standard</role-name>
    </auth-constraint>
</security-constraint>
```

这个安全限制表示资源只能由**Preferred**角色的用户访问：

```
    <security-constraint>
        <display-name>WROX Bank Security</display-name>
        <web-resource-collection>
            <web-resource-name>Preferred Customers</web-resource-name>
            <description>Resource to get statements</description>
            <url-pattern>/statement.do</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>preferred</role-name>
        </auth-constraint>
    </security-constraint>

    <!-- Define form login configuration -->
<login-config>
        <auth-method>FORM</auth-method>
        <realm-name>WROX Bank Authentication</realm-name>
        <form-login-config>
            <form-login-page>/Login.jsp</form-login-page>
            <form-error-page>/LoginError.jsp</form-error-page>
        </form-login-config>
```

```
    </login-config>

    <!-- Define the security roles -->
    <security-role>
        <description>Preferred Customers</description>
        <role-name>preferred</role-name>
    </security-role>
    <security-role>
        <description>Standard Customers</description>
        <role-name>standard</role-name>
    </security-role>

</web-app>
```

EJB部署描述项

取得结余和报告的用例用无状态Session Façade的方法实现。下面介绍如何根据EJB部署描述项中定义的角色控制这些方法的访问：

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar>

    <enterprise-beans>

        <session>

            <ejb-name>AccountBean</ejb-name>

            <home>ejb.AccountHome</home>
            <remote>ejb.Account</remote>
            <ejb-class>ejb.AccountEJB</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>

        </session>

    </enterprise-beans>

    <assembly-descriptor>

        <!-- Define the security roles -->
        <security-role>
            <description>Preferred Customer</description>
            <role-name>preferred</role-name>
        </security-role>
        <security-role>
            <description>Standard Customer</description>
        <role-name>standard</role-name>
        </security-role>

        <method-permission>
```

```
            <description>
                Permissions for preferred and standard customers
            </description>
            <role-name>preferred</role-name>
            <role-name>standard</role-name>
            <method>
                <ejb-name>AccountBean</ejb-name>
                <method-name>create</method-name>
            </method>
            <method>
                <ejb-name>AccountBean</ejb-name>
                <method-name>getBalance</method-name>
            </method>
        </method-permission>

        <method-permission>
            <description>Permissions for preferred customers</description>
            <role-name>preferred</role-name>
            <method>
                <ejb-name>AccountBean</ejb-name>
                <method-name>getStatement</method-name>
            </method>
        </method-permission>

    </assembly-descriptor>

</ejb-jar>
```

应用程序类

本节介绍应用程序中的各个类与接口如图5.4所示。

ControllerServlet小服务

这个小服务是个前端控制器：

```
package web;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;

import java.io.IOException;
public class ControllerServlet extends HttpServlet {

    public void process(HttpServletRequest req, HttpServletResponse res)
```

```
        throws ServletException, IOException {
    try {
```

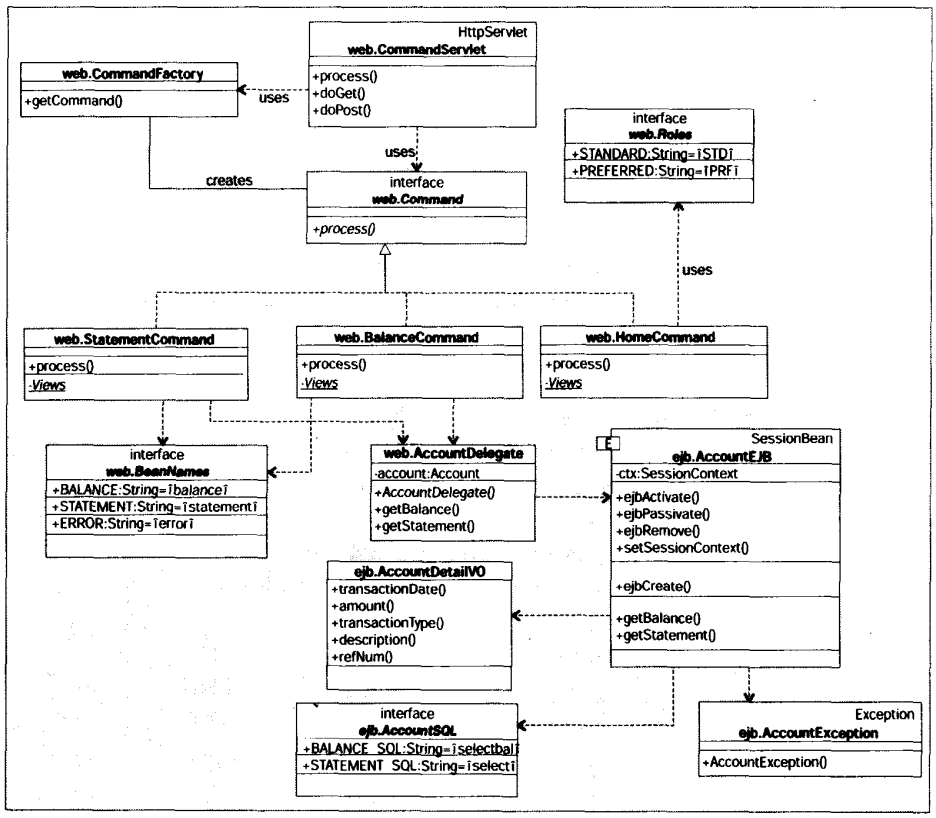

图5.4

根据小服务路径从工厂取得命令：

```
        Command com = CommandFactory.getCommand(req.getServletPath());
```

取得处理请求和读取下一视图JSP页面的命令：

```
        String view = com.process(req, res);
```

用请求派遣器将请求转发到JSP页面：

```
        RequestDispatcher rd = getServletContext().getRequestDispatcher(view);
        rd.forward(req, res);
    } catch(Throwable th) {
        throw new ServletException(th);
    }
}
```

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
    process(req, res);
}

public void doPost(HttpServletRequest req, HttpServletResponse res)
     throws ServletException, IOException {
    process(req, res);
}
```

Command接口

这个接口定义系统中使用的所有Command对象的合约：

```
package web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public interface Command {
```

所有Command对象都要实现这个方法：

```
    public String process(HttpServletRequest req, HttpServletResponse res);

}
```

Roles接口

Roles接口枚举Web层使用的编码角色值：

```
package web;

public interface Roles {
```

Standard用户在Web层使用的编码角色值如下：

```
    public static final String STANDARD = "STD";
```

Preferred用户在Web层使用的编码角色值：

```
    public static final String PREFERRED = "PRF";
}
```

BeanNames接口

BeanNames接口枚举JSP页面中使用的Bean名：

```
package web;

public interface BeanNames {
```

存储结余的Bean名如下：

```
    public static final String BALANCE = "balance";
```

存储报表的Bean名如下：

```
    public static final String STATEMENT = "statement";
```

存储错误消息的Bean名如下：

```
    public static final String ERROR = "error";
}
```

StatementCommand Command对象

StatementCommand Command对象处理报表请求：

```
package web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import ejb.AccountException;

public class StatementCommand implements Command {
```

定义命令视图的内接口如下：

```
    private static interface Views {

        public static final String STATEMENT = "/WEB-INF/Statement.jsp";
        public static final String ERROR = "/WEB-INF/Error.jsp";

    }

    public String process(HttpServletRequest req, HttpServletResponse res) {

        try {
```

用账户代理取得结余并将其存放成请求范围Bean，返回显示结余的JSP页面名：

```
            AccountDelegate delegate = new AccountDelegate();
            req.setAttribute(BeanNames.STATEMENT, delegate.getStatement());
            return Views.STATEMENT;

        } catch(AccountException ex) {
```

如果发生错误，则把错误消息存放成请求范围Bean，返回错误JSP页面名：

```
            req.setAttribute(BeanNames.ERROR, ex.getMessage());
            return Views.ERROR;
        }
    }
}
```

BalanceCommand Command对象

BalanceCommand Command对象处理结余请求：

```
package web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import ejb.AccountException;

public class BalanceCommand implements Command {
```

定义命令视图的内接口如下：

```
    private static interface Views {

        public static final String BALANCE = "/WEB-INF/Balance.jsp";
        public static final String ERROR = "/WEB-INF/Error.jsp";

    }
    public String process(HttpServletRequest req, HttpServletResponse res) {

        try {
```

用账户代理取得报表并将其存放成请求范围Bean，返回显示报表的JSP页面名：

```
                AccountDelegate delegate = new AccountDelegate();
                req.setAttribute(BeanNames.BALANCE, delegate.getBalance());
                return Views.BALANCE;

            } catch(AccountException ex) {
```

如果发生错误，则把错误消息存放成请求范围Bean，返回错误JSP页面名：

```
                req.setAttribute(BeanNames.ERROR, ex.getMessage());
                return Views.ERROR;
            }
        }
    }
```

HomeCommand Command对象

HomeCommand Command对象在初始登录后根据验证的用户角色决定显示哪个页面：

```
package web;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HomeCommand implements Command {
```

定义命令视图的内接口如下：

```
    private static interface Views {

        public static final String STATEMENT = "/statement.do";
```

```
        public static final String BALANCE = "/balance.do";
    }
    public String process(HttpServletRequest req, HttpServletResponse res) {
```

返回的URI显示优先客户的报表：

```
        if(req.isUserInRole(Roles.PREFERRED)) return Views.STATEMENT;
```

返回的URI显示标准客户的结余：

```
        else if(req.isUserInRole(Roles.STANDARD)) return Views.BALANCE;
        else throw new IllegalArgumentException("Unexpected Role.");
    }
}
```

CommandFactory工厂

这个类实现创建Command对象的工厂方法：

```
package web;

public class CommandFactory {

    public static Command getCommand(String path) {
```

返回的Command对象基于指定的小服务路径：

```
        if(path.equals("/home.do")) return new HomeCommand();
        else if(path.equals("/balance.do")) return new BalanceCommand();
        else if(path.equals("/statement.do")) return new StatementCommand();
        else throw new IllegalArgumentException("Invalid path:" + path);
    }

}
```

Account远程接口

这是实现用例的Session Façade的组件接口：

```
package ejb;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

import java.util.ArrayList;

public interface Account extends EJBObject {
```

取得结余的业务方法：

```
    public Double getBalance() throws RemoteException, AccountException;
```

取得报表的业务方法：

```
    public ArrayList getStatement() throws RemoteException, AccountException;
}
```

AccountHome主接口

```
package ejb;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;

public interface AccountHome extends EJBHome {

    public Account create() throws RemoteException, CreateException;

}
```

AccountEJB Bean类

这个Bean类用于实现用例的Session Façade：

```
package ejb;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.ejb.SessionBean;
import javax.ejb.EJBException;
import javax.ejb.SessionContext;

import java.util.ArrayList;

public class AccountEJB implements SessionBean {

    private SessionContext ctx;

    public void ejbCreate() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }

    public Double getBalance() throws AccountException {
```

```
        Connection con = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        InitialContext iCtx = null;
        DataSource ds = null;

        try {
```

取得当前验证用户的登录ID，安全情境的Web容器传递到EJB容器：

```
            String loginId = ctx.getCallerPrincipal().getName();
```

查找数据源：

```
            iCtx = new InitialContext();
            ds = (DataSource)iCtx.lookup("java:/AccountDS");
```

发出SQL，取得验证用户的结余：

```
            con = ds.getConnection();
            stmt = con.prepareCall(AccountSQL.BALANCE_SQL);
            stmt.setString(1, loginId);

            rs = stmt.executeQuery();
```

返回结余：

```
            if(rs.next())
                return new Double(rs.getDouble(1));

            throw new AccountException("Account not found.");

        } catch(NamingException ex) {
            throw new EJBException(ex);
        } catch(SQLException ex) {
            throw new EJBException(ex);
        } finally {

            try {
                if(iCtx != null) iCtx.close();
                if(rs != null) rs.close();
                if(stmt != null) stmt.close();
                if(con != null) con.close();
            } catch(Throwable th) {
                th.printStackTrace();
            }
        }
    }
    public ArrayList getStatement() throws AccountException {

        Connection con = null;
```

```
PreparedStatement stmt = null;
ResultSet rs = null;
InitialContext iCtx = null;
DataSource ds = null;

try {
    String loginId = ctx.getCallerPrincipal().getName();

    iCtx = new InitialContext();
    ds = (DataSource)iCtx.lookup("java:/AccountDS");
```

发出SQL，取得验证用户的报表：

```
    con = ds.getConnection();
    stmt = con.prepareCall(AccountSQL.STATEMENT_SQL);
    stmt.setString(1, loginId);

    rs = stmt.executeQuery();

    ArrayList statement = new ArrayList();
    AccountDetailVO vo;
    while(rs.next()) {
        vo = new AccountDetailVO();
        vo.transactionDate = rs.getTimestamp(1);
        vo.amount = rs.getDouble(2);
        vo.transactionType = rs.getString(3);
        vo.description = rs.getString(4);
        vo.refNum = rs.getString(5);

        statement.add(vo);
    }
```

返回报表：

```
    if(statement.size() > 0)
        return statement;

    throw new AccountException("Account not found.");

} catch(NamingException ex) {
    throw new EJBException(ex);
} catch(SQLException ex) {
    throw new EJBException(ex);
} finally {
    try {
        if(iCtx != null) iCtx.close();
        if(rs != null) rs.close();
        if(stmt != null) stmt.close();
        if(con != null) con.close();
    } catch(Throwable th) {
```

```
                    th.printStackTrace();
                }
            }
        }
    }
```

AccountSQL接口

这个接口枚举系统中使用的SQL命令：

```
package ejb;

public interface AccountSQL {
```

取得结余的SQL：

```
    public static final String BALANCE_SQL =
      "SELECT balance " +
      "FROM WR_ACCOUNT, WR_USER " +
      "WHERE WR_ACCOUNT.user_id = WR_USER.id " +
      "AND WR_USER.login_id = ?";
```

取得报表的SQL：

```
  public static final String STATEMENT_SQL =
    "SELECT transaction_date, amount, transaction_type, description, ref_num " +
    "FROM WR_ACCOUNT, WR_USER, WR_ACCOUNT_DETAIL " +
    "WHERE WR_ACCOUNT.user_id = WR_USER.id " +
    "AND WR_ACCOUNT.id = WR_ACCOUNT_DETAIL.account_id " +
    "AND WR_USER.login_id = ?";

}
```

AccountDetailVO数值对象

这个数值对象从EJB层向表示层传输账户细节数据：

```
package ejb;

import java.sql.Timestamp;

public class AccountDetailVO {

    public Timestamp transactionDate;
    public double amount;
    public String transactionType;
    public String description;
    public String refNum;

}
```

AccountException异常

这是Session Façade使用的业务异常：

```
package ejb;

public class AccountException extends Exception {

    /* Creates a new instance of AccountException */
    public AccountException() {
    }

    public AccountException(String message) { super(message); }

}
```

AccountDelegate代理类

```
package web;

import ejb.Account;
import ejb.AccountHome;
import ejb.AccountException;

import javax.rmi.PortableRemoteObject;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import java.rmi.RemoteException;

import javax.ejb.CreateException;

import java.util.ArrayList;

public class AccountDelegate {
```

引用Session Façade：

```
    private Account account;

    /* Creates a new instance of AccountDelegate */
    public AccountDelegate() {

        try {
```

查找Session Façade的主引用，并用其创建远程引用：

```
            InitialContext ctx = new InitialContext();
            Object obj = ctx.lookup("AccountBean");
            AccountHome home =
                (AccountHome)PortableRemoteObject.narrow(obj, AccountHome.class);
            account = home.create();

        } catch(NamingException ex) {
```

```
            ex.printStackTrace();
            throw new RuntimeException(ex.getMessage());
        } catch(RemoteException ex) {
            ex.printStackTrace();
                throw new RuntimeException(ex.getMessage());
             } catch(CreateException ex) {
                ex.printStackTrace();
                throw new RuntimeException(ex.getMessage());
             }

        }
        public Double getBalance() throws AccountException {
            try {
```

将方法调用委托给Session FaCade：

```
                return account.getBalance();
            } catch(AccountException ex) {
                throw ex;
            } catch(Throwable ex) {
                ex.printStackTrace();
                throw new RuntimeException(ex.getMessage());
            }
        }
        public ArrayList getStatement() throws AccountException {
            try {
```

将方法调用委托给Session FaCade：

```
                return account.getStatement();
            } catch(AccountException ex) {
                throw ex;
             }catch(Throwable ex) {
                ex.printStackTrace();
                throw new RuntimeException(ex.getMessage());
            }
        }
    }
```

应用程序使用的JSP页面

下面要介绍应用程序使用的各种JSP页面。

index.jsp页面

这是欢迎JSP页面，将请求转发到URI home.do：

```
<%@page contentType="text/html"%>
 <jsp:forward page="/home.do"/>
```

Login.jsp页面

这个JSP页面在非验证用户访问系统时让容器进行验证：

```
<html>
    <head>
        <title>Login Page for Examples</title>
    </head>
    <body bgcolor="white">
        <form method="POST" action='<%= response.encodeURL("j_security_check") %>'
        >
            <table border="0" cellspacing="5">
                <tr>
                    <th align="right">Username:</th>
                    <td align="left"><input type="text" name="j_username"></td>
                </tr>
                <tr>
                    <th align="right">Password:</th>
                    <td align="left"><input type="password"
                    name="j_password"></td>
                </tr>
                <tr>
                    <td align="right"><input type="submit" value="Log In"></td>
                    <td align="left"><input type="reset"></td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

LoginError.jsp页面

这个JSP页面在验证失败时让容器使用：

```
<html>
    <head>
        <title>Error Page For Examples</title>
    </head>
    <body bgcolor="white">
        Invalid username and/or password,
    </body>
</html>
```

Statement.jsp页面

这个JSP页面显示报表：

```
<%@page contentType="text/html" import="java.util.*,ejb.AccountDetailVO"%>
<html>
    <head>
        <title>Account Statement</title>
    </head>
    <body>

        <jsp:useBean id="statement" scope="request" type="java.util.ArrayList" />
        <table>
            <tr>
                <th>Date</th>
                <th>Amount</th>
                <th>Type</th>
                <th>Description</th>
                <th>Reference</th>
            </tr>
            <%
                Iterator it = statement.iterator();
                while(it.hasNext()) {
                    AccountDetailVO vo = (AccountDetailVO)it.next();
            %>
            <tr>
                <td><%= vo.transactionDate %></td>
                <td><%= vo.amount %></td>
                <td><%= vo.transactionType %></td>
                <td><%= vo.description %></td>
                <td><%= vo.refNum %></td>
            </tr>
            <%
                }
            %>
        </table>

        <a href="balance.do">View Balance</a>
    </body>
</html>
```

Error.jsp页面

这个JSP页面显示错误消息：

```
    <%@page contentType="text/html" isErrorPage="true" import="web.BeanNames" %>
    <html>
```

```
<head><title>JSP Page</title></head>
<body>
    <%= request.getAttribute(BeanNames.ERROR) %>
</body>
</html>
```

小结

本章介绍了下列与安全相关的模式：

- Single Access Point模式
- Check Point模式
- Role模式

我们在Wrox Web Bank应用程序中用标准J2EE API与部署描述项特性实现这些模式。这些模式可以用于应用程序设计与开发、主机配置与管理，等等，但这些模式没有定义特定技术和编码方式。这些模式的最终目标是不仅支持应用程序安全，而且支持主机与网络安全。

第6章 企业集成设计模式

企业集成是信息系统成功的关键因素。随着信息访问与电子商务要求的不断提高，集成应用程序的需求不断提高。但是，大多数公司用各种不同技术、语言和体系结构开发了现有的独立应用程序。这些公司不能在一夜之间抛开这些应用程序，因为它们是任务关键应用程序。另一方面，公司需要引入新应用程序。

如今，公司需要支持电子商务，包括B2B（企业对企业）、B2C（企业对客户）和CRM（客户关系管理）、PRM（伙伴关系管理）与SCM（供应链管理），等等。

没有集成良好的后端企业信息系统（EIS），公司就无法成功地引入这些新应用程序。企业应用程序集成（EAI）已成为近几年来企业应用程序开发中面临的最大难题之一。首先，公司中要集成应用程序，称为Intra-EAI；其次，还要实现Inter-EAI或企业对企业集成。

初一看，具体集成问题和相应方案是五花八门的，但处理几个集成项目之后，可以看到这些问题和方案可被分类成共同类别，我们把其称为集成模式。集成模式能帮助理解不同的集成方案与方法，可以对手头问题选择最佳方法，可以从一定抽象层考察集成问题，同时，这些集成模式为Intra-EAI和Inter-EAI提供了良好的方案。

本章介绍企业集成设计模式及其在J2EE标准中的运用。我们要介绍下列内容：

- 何谓企业应用程序集成（EAI）
- J2EE集成模式：
  - Integration Broker 模式
  - Integration Wrapper 模式
  - Integration Mediator 模式
  - Virtual Component 模式
  - Data Mapping 模式
  - Process Automator 模式
  - 集成模式间的关系
- 简单集成情形
- 用集成模式设计
- 实现集成模式
- 对B2B使用集成模式

何谓企业应用程序集成（EAI）

企业应用程序集成（EAI）对业务过程具有战略意义，因为新的创造性业务方案要求集成不同业务单元、企业数据、应用程序和业务系统。集成信息系统通过统一和有效的信息访问提高竞争优势。集成应用程序更容易从不同来源访问相关的协调的信息。事实上，总和大

于各个部分的简单和。

从业务角度看，所有应用程序集成到统一信息系统后，EAI可以得到竞争优势，可以共享信息和支持业务工作流程。信息通常从几个域收集，集成到业务过程中。尽管所要的信息可能是现成的，可能在应用程序中以某种形式存在，但没有EAI，用户通常无法访问这些信息。

从技术角度看，EAI指集成不同应用程序与数据，以便在所有参与的应用程序之间共享数据和集成业务过程，而不必修改现有应用程序。EAI使用的方法和活动能提高成本与时间效率。

尽管EAI的定义很简单，但许多人认为这是最乱的领域之一。他们认为标识问题和选择方案是费时的，需要大量知识，通常要靠第六感官才能集成成功。尽管事实上集成是复杂的，需要大量知识，但利用科学和系统方法，采用集成模式，可以按有条不紊的方式进行集成，而且有很大的把握取得成功。

集成体系结构

EAI的主要难点是公司依赖于异构的现有系统混合体，而这些不同体系结构的混合体不是按统一方式设计的，而是不断增长的。这种异构会消耗开发资源。具体表现如下：

- 组合单体应用程序、客户机－服务器应用程序与多层应用程序。
- 混合过程性方案与面向对象方案。
- 混合不同编程语言。
- 不同类型的数据库管理系统（关系型、层次式、对象）。
- 用不同中间件方案进行通信（面向消息中间件、对象请求代理、远程过程调用）。
- 多种信息传输模型，包括发表与预订、请求与答复和会话式。
- 不同事务与安全管理中间体
- 不同共享数据的方式。
- EDI、XML和其他数据交换的可能用法。

EAI要求建立壮实和可伸缩的集成体系结构，最重要的集成级有五级：

- 数据层集成
- 应用程序接口集成
- 业务方法集成
- 表示集成
- Inter-EAI集成

应用程序接口集成与业务方法集成的组合也称为业务层集成，因为它们具有一些相同概念。下面一一介绍这些集成体系结构的层。

数据层集成

数据层集成通常是应用程序集成的起点，可以访问其他应用程序共享的数据，可以在不同数据库之间移动数据。这好像很简单，但如果涉及几百个数据库，则很难管理。

典型的困难包括了解结构、标识数据、在不同系统之间映射数据、保证数据一致性、

分布式数据库问题、更新锁存，等等。此外，还要统一数据模型，解决多年开发在信息系统中引入的冗余度和其他词法异常问题，这是相当困难的。这种集成也称为结构集成。与其他集成层相比，数据层集成比较容易实现，通常不需要改变现有应用程序源代码。

应用程序接口集成

应用程序接口集成是高级集成，应用程序可以使用其他应用程序的某些功能。这是用应用程序提供的应用程序接口（API）实现的。通常，传输请求与结果时要使用某种形式的中间件，如面向消息中间件（MOM）、远程方法调用（RMI）或远程过程调用（RPC）。

一定要注意，应用程序接口集成只解决了集成的技术方面。换句话说，应用程序用面向技术的低级接口达到相互操作性。但是，应用程序接口集成解决了数据层集成的几个问题，特别是不需重复而可以复用功能。

业务方法集成

业务方法集成提供的高级方法抽象业务方法，现有方案通过接口参与业务过程的不同步骤。业务方法集成按我们的要求显示企业级信息，明确要求要从集成系统取得什么，知道并支持现代技术，即信息系统接口基于新设计的体系结构。

但是，功能不是重新实现，而是利用现有应用程序。利用现有应用程序重新建模，使其提供业务过程层功能，集成到现代应用程序体系结构中。

要实现业务方法集成，通常要对业务过程进行改造，而不是单纯的技术问题。它要求实现几个技术层基础，在高层抽象中集成应用程序。

表示集成

表示集成要开发统一的表示层，隐藏后台的不同应用程序，有些是遗留应用程序，有些是新建应用程序。这样，我们可以提高最终用户效率，今后替换部分遗留系统时不会影响系统其他部分。图6.1显示了表示层用户接口组件访问业务逻辑组件。其中一些通过复用现有系统而实现操作，而有些则是新开发的。在表示层组件看来，两者是相同的。

Inter-EAI

如今，即使在公司内集成应用程序也还不够，还要实现企业间集成，通常称为B2B集成或电子商务。电子商务给信息系统提出了新的难题。如今的要求非常高，公司不能单单向Web页面发布联机产品目录，还要有联机的最新信息，提供质量、效率与可靠性。即使传统的知名企业也要不断努力才能在电子商务环境中保持自己的地位。

当然，有效B2B集成或电子商务的前提是集成企业信息系统（可能在业务过程层集成）。只有这一层集成才能满足请求处理要求。今天的客户要求立即响应，而不是通过批处理，几天后再收到确认函。但是，电子商务系统后端常常没有高效集成的企业信息系统。通过后端（企业信息系统）与前端（表示系统）的高度耦合集成达到立即响应性是成功的关键因素。

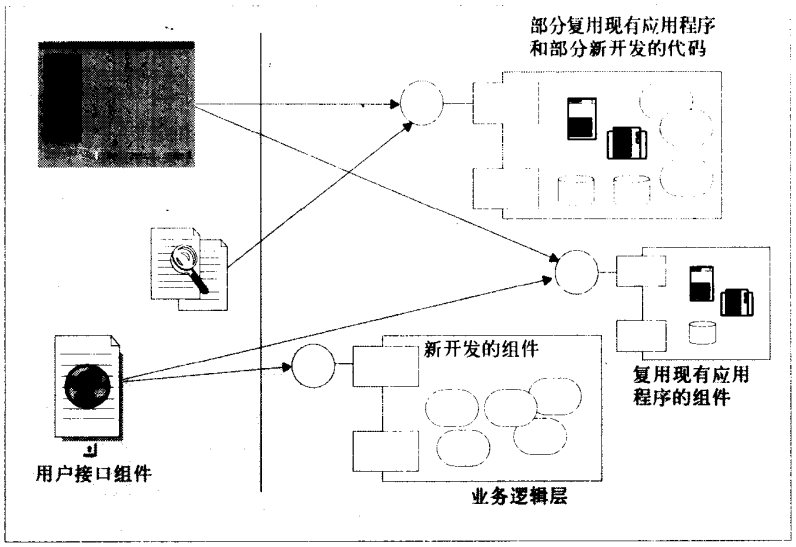

图6.1

Gartner Group之类一流顾问公司研究表明，当前很少前端系统能与后端高效集成的。大多数非集成应用程序无法满足业务要求，主要原因是缺少企业集成，这是成功电子商务和高效公司的最重要前提条件。

另一个重要事实是大多数前端应用程序可以用现有和遗留系统作为后端方案。提高这种系统集成的效率是成功的关键因素。特别地，必须立即响应和立即将数据传播到所有相关应用程序是主要问题。后端系统无法高效地支持前端应用程序显然无法满足所有要求。尽管这个预期是合理的，但即使到2005年，行业研究与顾问组织（如Gartner Group（http://www.gartner .com/））的预测表明，超过一半的前端系统仍然不能充分集成。

为了达到无缝B2B集成，公司的EAI间连接应基于EAI内连接。图6.2显示了在J2EE EAI体系结构上让B2B集成层用Web服务实现。

介绍最主要的集成层之后，下面介绍基于EAI体系结构的J2EE平台。

EAI与J2EE

众所周知，J2EE建立在多层应用程序模型之上。每层的建筑块为组件。

- 客户层的组件是应用程序客户机和小应用程序。
- Web组件层组件是小服务和JSP页面，可以为瘦客户机服务，响应HTTP请求和用HTML、XML等技术产生瘦客户机用户接口。

业务逻辑层的组件为Enterprise JavaBeans（EJB），通常支持事务、安全、持久性等服务。

惟一不由组件构成的层是EIS层，包含现有企业基础结构，如ERP系统、IP监视器、数据库管理系统和其他现有系统。

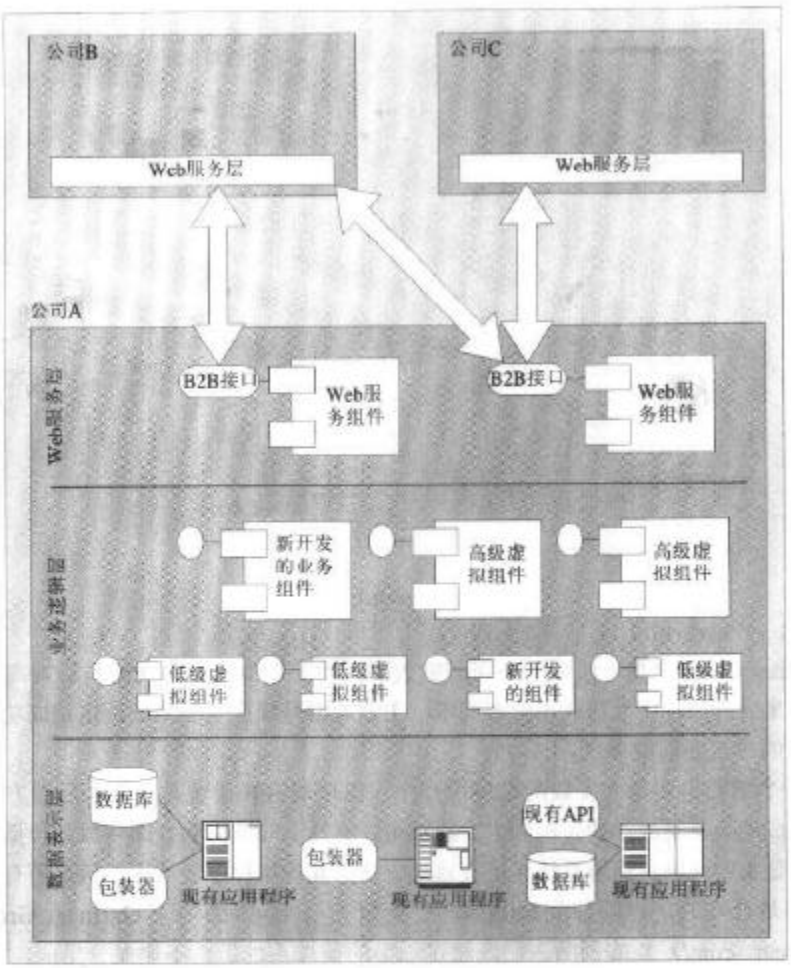

图6.2

J2EE支持几个层间通信开放标准，包括HTTP（S）与IIOP协议和中间件API。最主要的集成Java API包括：

- 数据访问中间件：JDBC
- 面向消息中间件：JMS
- 对象请求代理：RMI-IIOP与Java IDL
- 事务处理：JTA与JTS
- 安全：JAAS、J2SE的安全API
- Java连接体系结构（JCA，Java Connector Architecture），按通用方式访问现有EIS系统。
- 通过JAXP支持XML
- JNDI，集成不同命名与目录服务

• Web服务支持，不在J2EE版本1.3中，而是以另一个包提供。

J2EE支持层间通信开放标准，使我们可以在任何层中包括不是用Java开发的组件，从而集成现有应用程序。

在业务逻辑层，我们不仅可以部署EJB组件，而且可以通过JMS支持MOM部署CORBA（与RMI-IIOP）分布式对象和组件。这是最重要的集成事实之一。可以看到，在业务逻辑层开发虚拟组件时，不限于EJB，而可以用CORBA、RMI-IIOP与MOM组件。

注意CORBA与MOM组件不一定用Java开发，也可以用各种编程语言。CORBA与MOM产品支持各种常用编程语言，包括C++、C、Smalltalk、Ada、COBOL、Delphi与even Visual Basic。

这些非Java组件不在EJB容器中，因此不能利用容器提供的管理环境，但可以参与事务和安全机制。为此，它们要使用相应API，见稍后介绍。

在Web组件层，还可以部署其他非Java组件，包括任何能响应HTTP请求的组件。它们也无法在J2EE应用程序服务器提供的Web容器中执行，但我们不是直接从Web组件层访问现有应用程序，而是通过业务逻辑层访问。

同样，对于客户层，应用程序客户机不一定是Java应用程序，也可以用其他语言开发，用指定协议与其他层通信。例如Web浏览器、CORBA与MOM客户机，等等。

这样就得到扩展J2EE集成体系结构，可以达到J2EE EAI，如图6.3所示。

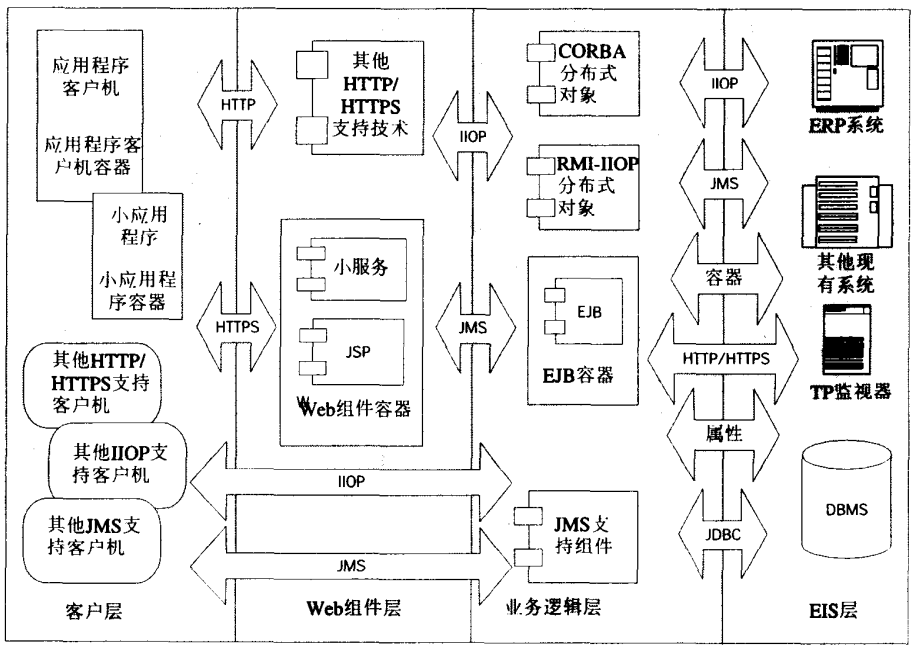

图6.3

可以看出扩展J2EE集成体系结构明确分开不同层，可以用有限协议进行层间通信。业务逻辑层分开Web组件层和客户层与EIS层。客户机和Web组件层不能访问EIS层。

因此，要访问EIS层，只能通过业务逻辑层。这里EJB、CORBA、RMI-IIOP与JMS支持组件用各种协议和技术访问EIS层。只有EIS层出现现有应用程序的异构性。

要在应用程序之间访问和交换数据，就要定义数据交换格式。过去几年，XML已经成为事实上的标准数据交换格式。XML的强大之处在于可以自己定义标志，这样就可以描述任何结构中的任何数据（但XML也许更适合层次式数据结构）。但是，随意定义标志可能在交换数据中造成词法问题，计算机无法从标志推测出数据的含义。

交换数据的双方要确定词汇协议，支持相同标志集，可以明确定义DTD或结构。公司内交换数据的双方要确定词汇协议比公司间容易，因此人们投入大量精力定义不同行业的标准化词汇。

J2EE集成模式

集成模式描述企业内外集成时使用的经过证明的方法与过程，是从常见集成方案中总结出来的。每个集成模式定义一个常见集成问题及其解决方案。本章介绍的集成模式针对J2EE，因此这些集成模式的例子和应用程序是针对J2EE的，但模式本身也适用于其他体系结构。

本章介绍下列结构性集成模式：

- Integration Broker模式（也称为Integration Messenger）
- Wrapper模式（Integration Adapter, Integration Connector）
- Integration Mediator模式
  - 单步应用程序集成
  - 多步应用程序集成
- Virtual Component（Integration Façade）
- Data Access Object（DAO）（Data Exchange模式）模式
- Data Mapping模式
  - Direct Data Mapping（直接数据映射）
  - Multi-Step Data Mapping（多步数据映射）
- Process Automator模式

其中Data Access Object（DAO）模式已经在J2EE设计模式中介绍，这里不再重复。其他模式是针对J2EE集成引入的，没有在任何模式类别中介绍，因此我们简要介绍这些模式，便于后面的集成案例中选择。关于EAI与J2EE的详细信息，见《Professional J2EE EAI》一书，Wrox Press出版（ISBN-1-861005-44-X）。

Integration Broker模式

首先要详细介绍Integration Broker模式。

情境

公司内或公司间集成应用程序时，通常要求达到几个不同应用程序之间的集成。将每个应用程序相互连接并不好，因为这样会增加依赖性和复杂性，从而使维护变得非常困难。

问题

在点对点集成中，应用程序交互逻辑与两个集成应用程序耦合。这样就使应用程序高度相互依赖，使维护变得复杂而费时。一个应用程序的小变动就可能要求修改所有其他连接的应用程序。

事实上，维护集成系统比维护应用程序本身更费时费钱，使集成的好处大打折扣。在典型集成情形中，我们要面对许多这样的应用程序（通常超过五十个），因此点对点方法是行不通的，会使复杂性呈指数级增加。

要求

要求如下：

- 分开不同操作的责任是集成的应用程序之间所必需的。
- 应用程序应提供公共集成接口，解决复杂性而不要对每个集成情景建立相互操作性接口。
- 集成逻辑应分开，简化维护工作。
- 客户机不能看到集成细节。
- 客户机不能看到集成应用程序的内部结构。
- 不能限制通信模型，每个应用程序应使用最佳方案。

方案

Integration Broker模式描述了集成许多不同应用程序的结构性方法，克服了点对点集成的缺点。这个模式减少集成应用程序间的依赖性，提供应用程序间的一个或多个通信机制。Integration Broker抽象中间件技术，用J2EE平台提供的中间件技术组合实现，提供必要的服务，如事务、安全、命名、寿命周期、伸缩性、管理、规则、路由、代理，等等。

Integration Broker用于应用程序集成达到不同集成级。应用程序通过编程或声明方式用接口透明访问集成代理。编程方式就是应用程序实现使用基础结构服务的代码，而声明方式则可以标记特定应用程序，声明它们使用什么服务，由基础结构负责服务调用细节。

应用程序所提供服务的透明性依赖于所选的技术。例如，通信、代理和路由服务可以用对象请求代理透明地实现，掩盖远程方法调用，在开发人员看来就像本地方法调用一样。而面向消息中间件则要求应用程序创建消息，分析输入消息和相应作出响应。声明式事务与安全服务可以向应用程序提供服务而不在应用程序中增加任何代码，等等。

Integration Broker模式可以建立集成层和复用前面的结果，它不是基于应用程序之间的点对点通信，从而可以把多对多的多样性减少到多对一，从而减少复杂性，简化维护工作。图6.4显示了这个结构。

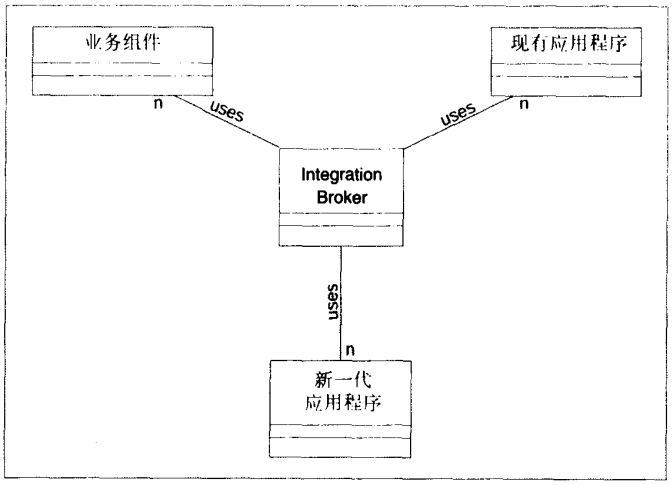

图6.4

Integration Broker定义某个集成交互中每个应用程序的作用。对每个交互，一个或几个应用程序从某个应用程序中要求某种服务。要求服务的应用程序称为客户机应用程序，而提供服务的应用程序称为服务器应用程序。注意客户机和服务器角色只对某个交互定义，在执行期间可以改变。通常，所有应用程序都具有双重角色，即是某些交互的服务器应用程序，又是某些交互的客户机应用程序。

Integration Broker不仅连接应用程序，而且根据应用程序之间的合约进行集成。这些合约通常表示为相互操作性接口。相互操作性接口定义客户机应用程序可以向服务器应用程序请求的服务。

相互操作性接口定义所依赖应用程序之间的关系。长期合约接口定义集成应用程序之间的耦合，提供一个façade，让客户机应用程序访问相互操作性服务，包装应用程序。只要接口保持不变，就可以将服务器应用程序的部分或全部进行替换而不影响任何客户机应用程序。因此，人们投入很大精力定义相互操作性接口。

图6.5显示了这种情形，集成代理用相互操作性接口连接五个应用程序，将在后面的案例中使用。

Integration Broker应支持下列通信模型中的一个或几个：

- 同步一对一通信，客户机从服务器请求立即响应并等待响应。
- 延迟同步一对一通信，客户机请求服务器稍后响应或通知（回调或事件）。
- 异步一对一通信，客户机不要求服务器答复，也不要求提交请求时服务器联机。
- 异步一对多通信，一个客户机可以和多个服务器通信。这种通信模型也称为发表/预订。

这个模式的典型代表是公用中间件技术，特别是远程方法调用（对象请求代理）和面向消息中间件。

通常，Integration Broker模式在集成中隐式使用。这个模式要求一个或几个中间件产

品，实现所要功能。如今，随着高层抽象的获取，这个模式不在设计时建模，但体系结构应支持这个模式，在组件之间采用通信请求。

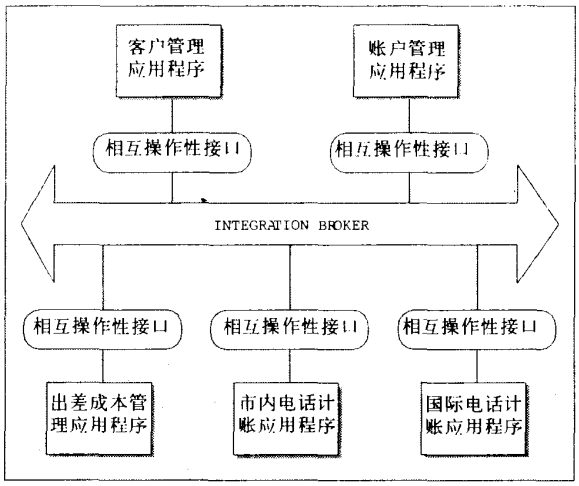

图6.5

后果

- 减少应用程序之间的连接数，从而减少集成应用程序之间的耦合水平。
- 应用程序之间的依赖性定义为合约，以相互操作性接口形式出现。
- 如果认真定义接口，则可以将应用程序的部分或全部进行替换而不影响任何其他参与的应用程序。
- 地址透明性被获取。
- 将维护工作与成本减少到可接受水平。

相关模式

这个模式与本章介绍的所有其他集成模式相关，也与GoF Mediator模式相关。

采用模式

在J2EE中，可以用下列技术及其组合实现Integration Broker模式：

- **RMI-IIOP**

  RMI（远程方法调用）是基于相互操作性接口的同步进程间通信的主要J2EE技术。由于使用IIOP，可以和CORBA相互操作。

- **CORBA**

  CORBA在组件与基于相互操作性接口之间提供同步、延迟同步和异步通信。CORBA不局限于Java语言，因此可以集成不同编程语言的应用程序。此外，由于RMI-IIOP客户机与CORBA兼容，因此EJB与RMI-IIOP组件也像是CORBA组件。这样就可以在业务逻辑层放上不是EJB的组件。CORBA在J2EE（与J2SE）中的实现称为Java IDL。

- **JMS（MOM）**

  Java Message Service（JMS）可以根据发送与接收消息进行异步通信，支持点对点和发表/预订模型。通过JMS可以用任何JMS支持MOM产品（假设JMS安装MOM客户机产品）。JMS提供抽象层，使应用程序很容易在不同MOM产品之间移植。这很重要，因为MOM中间件传统上一直用于公司实现应用程序间的部分集成。通过JMS很容易复用这个集成，将其连接EIS层现有应用程序的通信路径。

- **JDBC**

  要从DBMS访问数据，可以用JDBC API提供的统一接口通过公用接口访问不同的关系型DBMS产品。JDBC提供了四种不同驱动程序（JDBC-ODBC桥、JDBC自然驱动器桥、JDBC网络桥和纯粹Java驱动器）。

- **JCA**

  Java Connector Architecture（JCA）解决了当今访问不同EIS系统需要定制方案的问题。JCA定义了应用程序服务器与连接器之间连接管理、安全性和事务的系统级合约。每个EIS系统的连接器以EIS特定方式实现这些合约。J2EE组件通过标准API访问EIS，称为公用客户机接口（CCI，Common Client Interface）。CCI隐藏了EIS系统的差别，换句话说，JCA在EIS系统中就像JDBC在数据库中的作用一样。

Integration Wrapper模式

下面详细介绍Integration Wrapper模式。

情境

编程访问现有应用程序对复用现有应用程序功能和开发新方案非常重要。现有应用程序可能没有提供API或提供的API不够用。

问题

如果要复用现有应用程序功能，则要用某种方法访问。最明显的方法是通过现有应用程序提供的API访问。

但是，有时现有应用程序可能没有提供任何API访问功能，只能通过API访问一组功能，而无法访问另一组功能。

要求

要求如下：

- 要集成和复用现有应用程序功能，就要通过程序访问。
- 通过API访问，可以复用现有应用程序功能和数据。
- 通过API访问应用程序比直接访问数据库好，因为这样就不用避开业务逻辑。

方案

Integration Wrapper模式是个分层方法，在现有应用程序中增加API，即把现有应用程

序的服务与功能提供给其他应用程序，实现集成和相互操作性。这是从Adapter（GoF）模式演变而成的，但Integration Wrapper模式的目标是对多个客户机同时访问提供可复用接口。

要建立集成包装器，我们在现有应用程序上增加一层，加入必要的接口。我们把增加的应用程序称为接口包装器，把修改后的现有应用程序称为包装的现有应用程序。

Integration Wrapper有两个主要目标：

- 提供开放和可复用的相互操作性接口。
- 将调用变成复用现有应用程序的服务与功能。

对于后者，Integration Wrapper可以复用现有API或增加API。图6.6显示了Integration Wrapper结构。

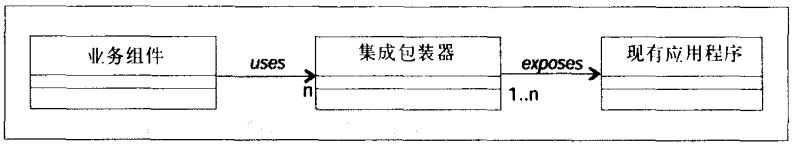

图6.6

一般来说，包装器可以用侵入式方式或非侵入式方式增加API。可以用两种方法在现有应用程序中增加包装器：

- 可以修改现有应用程序，提供新的访问功能方法。
- 可以用屏幕刮取或终端仿真来通过用户接口访问功能。

选择修改现有应用程序或者用屏幕刮取或终端仿真的条件之一是能否访问源代码和所需工具。我们要保证源代码版本完整和反映当前执行代码。如果没有源代码和所需工具与库来建立现有应用程序的工作执行文件，则只能利用用户接口包装（见本章稍后介绍）。

即使有了源代码和所需工具，也可能不选择修改现有应用程序而是用屏幕刮取或终端仿真。因为我们不想冒在现有应用程序中引入缺陷的风险、我们不熟悉现有应用程序的技术，等等。

选择这些方法之前，要先确定访问现有应用程序功能的操作，这是通过分析现有应用程序而进行的。我们要分析应用程序，要集中考虑实际需要。图6.7纲要式地显示了包装器。

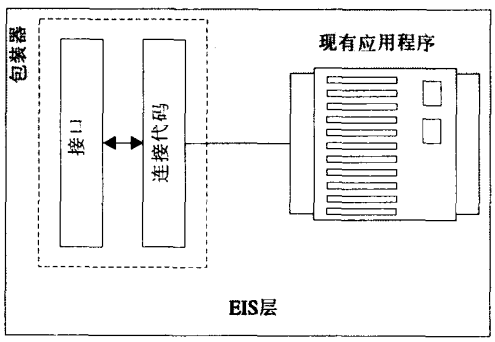

图6.7

Integration Wrapper模式与Integration Broker模式相关。根据通信类型，包装器结构会有所不同。如果使用同步通信方式，如远程方法调用，则基础结构要求类型通信，检查操作签名。使用异步通信时，包装器要负责分解消息。

结果

结果如下：

- Integration Wrapper可以编程与现有应用程序通信。
- 可以复用现有应用程序的服务与功能。
- 隐藏现有应用程序内部结构细节、技术、编程模型和通信细节。
- 减少复杂性和分离客户机。

相关模式

相关模式有Integration Broker、Virtual Component和GoF Adapter 。

采用模式

可以用各种技术开发包装器。一般来说，可以选择下列技术：

- ORB式通信，使用CORBA或COM+
- MOM通信，使用IBM MQ之类产品（支持JMS映射）
- 在协议层实现通信（SOAP、HTTP、TCP/IP等等）
- RPC式通信，使用分布式计算环境（DCE）
- 通过Java Native Interface（JNI）直接通信

事实上，Integration Wrappers很少用Java实现，因为现有应用程序通常不是Java应用程序。因此，我们主要介绍与J2EE相互操作的技术和对其他编程语言的支持，最重要的是CORBA、JMS (MOM)与SOAP。后面会介绍如何用CORBA开发包装器。

Integration Mediator模式

下面要详细介绍Integration Mediator模式。

情境

集成应用程序时，集成逻辑通常要与所有涉及的应用程序分开，并通过包装减少依赖性、增加复用性和简化维护。

问题

对需要集成的现有应用程序或公用功能分布在多个应用程序中或在多个现有应用程序中重复时，通常需要集成逻辑，用集成逻辑解决这些问题和以共同方式向客户机表示服务与功能。客户机不知道隐藏在调节器后面的细节。

要求

要求如下：

- 现有应用程序通常需要集成
- 一些功能分布在多个现有应用程序中
- 一些功能在多个现有应用程序中重复
- 应用程序间的交互很复杂
- 客户机不能知道这个细节

方案

Integration Mediator是现有应用程序的控制器，向客户机提供一个接口。它知道现有应用程序并包括集成逻辑，可以完成某些高级操作，需要与现有和新应用程序进行复杂交互。Integration Mediator中包含的集成逻辑可以在不同集成层使用，如数据层集成或功能与方法层集成。Integration Mediator不直接访问现有应用程序，而是使用集成包装器。

Integration Mediator结构显示了客户机、集成包装器和调节器，如图6.8所示。

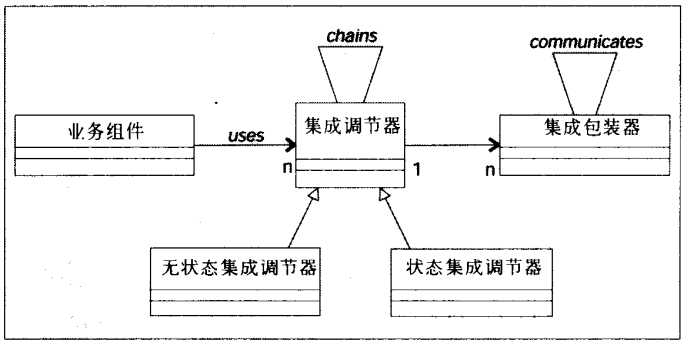

图6.8

一般来说，集成调节器有两种形式：

- 单步或无状态集成调节器
- 多步或状态集成调节器

无状态集成调节器用于与现有应用程序交互期间不需要维护状态的情形，即调节器的工作只取决于现有应用程序的响应。例如路由与代理和词法变换。无状态集成调节器的典型例子是XSLT引擎。

状态集成调节器用于跟踪应用程序的前次交互。然后调节器累计前次交互的数据，并用这个信息进一步交互。状态集成调节器也基于事件，记住事件，并在触发所有需要的事件之后执行某个操作。

状态集成调节器要求管理状态，有时还要保存状态。与现有应用程序交互时可能需要状态持久性，要更长时间维护系统。这种长时间交互可能很复杂，持续几分钟、几小时甚至几天。

后果

结果如下：

- 减少应用程序依赖性
- 包装集成逻辑并与参与的现有应用程序和新客户机分离
- 简化维护，集成逻辑集中起来，而不是分布在现有应用程序中
- 可以在调节器提供的功能之上建立服务，从而不必知道现有应用程序中的细节

相关模式

相关模式包括Integration Wrapper、Virtual Component、GoF Mediator和GoF Façade。

采用模式

在J2EE中开发集成调节器时可以选择不同技术，包括EJB、CORBA组件、RMI-IIOP分布式对象、JMS与JCA。但是，最常见的是EJB，直接映射不同调节器类型。无状态调节器可以用无状态会话Bean或消息驱动Bean；状态调节器可以用状态会话Bean或实体Bean，取决于所要的持久性类型。

Virtual Component模式

下面详细介绍Virtual Component模式。

情境

集成信息系统服务的访问方法是变化的，特别是集成信息系统使用不同技术实现不同的现有应用程序时。要直接访问这些服务，就要求客户机知道信息系统的内部结构。Virtual Component提供统一的共同服务访问点，从而作为现有应用程序的façade。

问题

客户机访问集成信息系统时不应知道现有应用程序的细节，不应直接访问。集成信息系统不是简单地连接现有应用程序，而是要提供新的服务与功能，应当以统一的共同方式向客户机提供。如果客户机直接访问现有应用程序，则会高度耦合，使维护与更换非常困难。

要求

要求如下：

- 现有应用程序不提供客户机所要的高级服务与功能。
- 客户机不应知道信息系统细节。
- 客户机应使用高级服务

方案

Virtual Component模式提供了集成现有应用程序与客户机的方案。这个模式起源于façade模式，但目标是集成和提供复用性，能够在虚拟组件背后用实际的新方法改造和更换

应用程序。基于Virtual Component模式的集成信息系统像新开发的信息系统，但实际上是复用现有应用程序功能。

从客户机角度看，虚拟组件和新开发的组件没什么不同。两者都通过接口提供功能。因此虚拟组件可以使现有应用程序像新开发的组件一样，以各种方式混合现有应用程序和新开发的应用程序。

Virtual Component模式包装现有应用程序服务于请求的细节及其服务方法。一方面，它通过抽象相互操作性接口显示现有应用程序。另一方面，Virtual Component与Integration Wrapper和Integration Mediator进行通信，如图6.9所示。

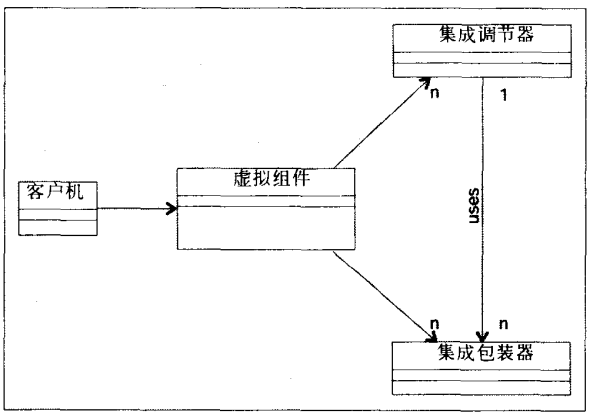

图6.9

例如，Virtual Component可以是个订单组件，提供下订单的简单接口，内部访问付款系统、库存系统等现有系统。

Virtual Component和抽象相互操作性接口一起提供独立于实现的应用程序视图。如果抽象接口保持不变，则其他应用程序不知道原应用程序所作的任何改变。这样，原应用程序发生改变时，就不需要修改依赖于这个应用程序的其他应用程序。换句话说，客户机虚拟组件无法确定服务器虚拟组件如何实现。

Integration Broker模式影响虚拟组件的实现方法。Virtual Component隐藏现有应用程序的细节，现有应用程序在后台实现高级业务功能。

Virtual Component在业务逻辑层部署。在集成期间分几个阶段建立。不同虚拟组件提供的接口对功能访问提供不同的抽象级。因此，我们把虚拟组件组织成业务逻辑层的几个子层。

我们建立下列虚拟层组件：

- 访问现有数据库的低级虚拟组件。这些虚拟组件在数据级集成中开发。
- 访问现有应用程序功能的低级虚拟组件。这些虚拟组件在应用程序接口级集成中开发。
- 提供高级业务方法的高级虚拟组件，这些虚拟组件在业务方法级集成中开发。

低级虚拟组件负责技术采用，建立在数据级和应用程序接口集成之上。高级虚拟组件将高级业务请求变成现有应用程序不同低级虚拟组件的一系列低级调用。高级虚拟组件还进行一些数据转换和适配，最后达到通过复用现有应用程序实现功能的目标。

图6.10显示了在业务逻辑层把虚拟组件组织成多个子层，高层虚拟组件访问三个低层虚拟组件，低层虚拟组件访问现有系统。箭头显示了图中的通信。

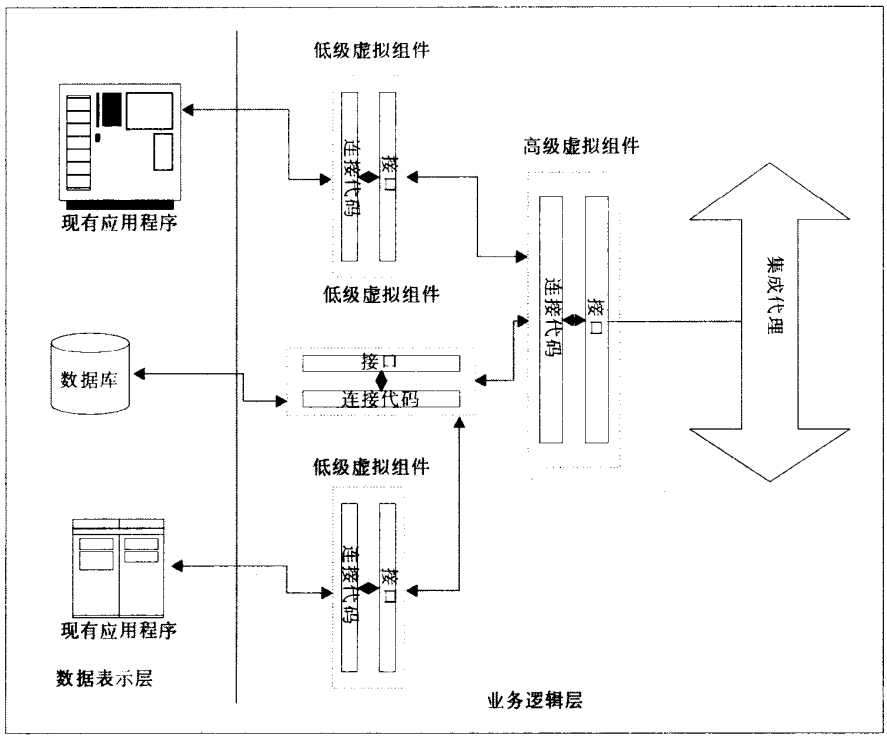

图6.10

我们通常用虚拟组件完成下列功能：

- 虚拟组件包装现有应用程序功能并以相同方式或修改的方式显示。例如，虚拟组件可以显示与现有应用程序API中相同的功能，这时只是掩盖技术差别，称为低级虚拟组件。
- 低级虚拟组件实现的代码将方法调用转变成现有应用程序和数据库访问的一个或几个API调用。这个代码通常要进行参数转换、数据类型对齐和其他转换，在J2EE技术与现有应用程序使用的技术之间掩盖技术差别。简单地说，低级虚拟组件是新应用程序和现有应用程序之间的适配器。
- 虚拟组件可以提供不同的高级接口，从而掩盖现有应用程序实现API的方式。这种虚拟组件称为高级虚拟组件，其接口基于全局设计模型定义，见本章稍后介绍。
- 虚拟组件可以包装几个现有应用程序和帮助维护事务完整性与安全性。
- 虚拟组件还可以包装和抽象持久数据。为此，虚拟组件可以直接访问EIS数据库或通过提供的协议访问。这时，虚拟组件通常还实现检验逻辑。这是另一层安全性（服务的应用程序可能没有处理），从而保持数据库处于一致状态。

- 虚拟组件可以对几个EIS数据库提供统一访问，可以在后台处理不同数据库组合。也可以用API访问数据，只要现有应用程序提供这些API。
- 虚拟组件通常是分层的，高级虚拟组件累计低级虚拟组件的行为并对EIS应用程序功能的多层抽象提供所需的高层抽象。
- 虚拟组件还可以映射技术差别。特别有用的情形是用其适应同步与异步通信模型。通过J2EE技术（特别是JMS），可以适应不同MOM产品。大多数情况下，适应ORB产品并不难，因为它们都基于IIOP协议（因此是相互可操作的）。

后果

结果如下：

- 虚拟组件提供集成信息系统服务与功能的统一视图。
- 虚拟组件抽象信息系统细节，从而提供某种façade。
- 虚拟组件提供相互操作性的高级面向业务过程接口。
- 虚拟组件很容易替换组件，只要接口保持不变。

相关模式

与Virtual Component模式相关模式有Façade、Integration Wrapper、Integration Mediator与Integration Broker。

采用模式

开发虚拟组件时，要遵循下列准则：

- 我们要设计虚拟组件，使其适用于不同情形。高级虚拟组件应支持低级虚拟组件的不同组合。低级虚拟组件应很容易根据不同EIS系统调整。
- 高级虚拟组件不能依赖于执行环境，不能对环境作任何假设，从而使更换新组件很容易。
- 虚拟组件应能在J2EE集成体系结构使用的事务和安全模型中协作。此外，应能将这些事务和安全模型映射到现有应用程序（如果现有应用程序支持事务与安全，否则虚拟组件应根据项目要求提供这个功能）。

我们还确定把虚拟组件部署到业务逻辑层。在J2EE集成体系结构中，可以使用下列技术：EJB、RMI-IIOP、CORBA、JMS、JCA、JAX/RPC（包括SOAP）与JAXM。后两个技术还在开发，因此目前不太普及，但今后可能普及。

选择相应技术并不太难。前三个技术是EJB、RMI-IIOP与CORBA，基于IIOP协议，很容易选择。它们提供同步、ORB式通信接口。建立在这三个技术之上的虚拟组件很相似，向Java和其他IIOP支持客户机提供远程方法调用功能。由于相似，可以交换这些技术的实现而不必改变客户机（客户机不知道这些差别）。

CORBA与RMI-IIOP

CORBA与RMI-IIOP特别适合低级虚拟组件，通常要与其他编程语言写成的现有应用程

序通信。如果要用专属机制访问无法从Java调用的现有应用程序API，则可以用CORBA。为此，我们考虑把CORBA和现有应用程序的编程语言一起使用。要访问这些CORBA虚拟组件，通常使用RMI-IIOP。

低级虚拟组件通常直接与EIS层的现有应用程序通信，用任何协议与技术组合访问EIS层，包括JMS、Connectors、IIOP、HTTP、JDBC和访问现有应用程序API或数据库的专属机制。

JMS

JMS用于低级虚拟组件，可以达到异步面向消息通信，通过一种或几种MOM产品实现。通常，我们在一端通过JMS使用MOM，另一端自然使用MOM产品。

EJB

EJB专门用于开发高级虚拟组件。这些虚拟组件通常与其他技术开发的低级虚拟组件通信。

如果要在RMI-IIOP上对API或包装器提供同步访问，则可以用会话Bean实现虚拟组件。它们通常要调用几个方法，还要变换和适配结果。如果虚拟组件不需要在客户机与Web组件层方法之间调用存储任何客户机相关数据，则我们选择无状态会话Bean，否则使用状态会话Bean。

如果要对API或包装器提供异步访问，则可以用消息驱动Bean向客户机提供功能。我们像会话Bean一样实现与现有系统的连接。消息Bean是无状态的，但并不会对异步请求造成问题，因为Bean通常不必记住客户机相关信息。

为了表示持久数据，可以用实体Bean。如果用JDBC直接访问数据库，则可以用容器管理持久性。如果通过专属协议或API或包装器访问数据，则要使用Bean管理持久性。

J2EE连接器体系结构

J2EE连接器体系结构（JCA，J2EE Connector Architecture）解决了当今访问不同EIS系统需要定制方案的问题。JCA定义了应用程序服务器与连接器之间连接管理、安全性和事务的系统级合约。每个EIS系统的连接器以EIS特定方式实现这些合约。J2EE组件通过标准API访问EIS，称为公共客户机接口（CCI，Common Client Interface）。CCI隐藏了EIS系统的差别。

JAX/RPC与JAXM

有时，特别是集成新系统或其他企业体系结构开发的新一代系统时，可以用SOAP之类的现代XML协议。为此，我们用包装不同现有系统时使用的相同体系结构。惟一的差别是我们用JAX/RPC与JAXM接口通过XML接口访问这些系统。使用SOAP之类的XML协议可以把集成推向B2B集成。

后面会介绍开发虚拟组件的例子。

Data Mapping模式

下面详细介绍Data Mapping模式。

情境

集成现有应用程序时，数据通常要从一个应用程序移到另一应用程序。现有应用程序数据库的访问随数据库类型（关系型、对象、层次式、平台文件等）和数据访问方法（直接访问数据库，通过应用程序访问，等等）而不同。

问题

数据映射逻辑可能很复杂，因此最好能对现有应用程序进行包装和分离。这样就可以使数据映射逻辑与现有应用程序更少耦合，从而提高维护性。

要求

要求如下：

- 现有应用程序间要移动数据。
- 数据在现有应用程序中存放成不同格式。
- 数据访问随存储类型和访问技术而不同。
- 应分离数据映射逻辑而不涉及技术细节。

方案

Data Mapping模式可以包装与抽象数据映射逻辑，便于在相关集成的现有应用程序之间传输、复制与更新数据。Data Mapping模式可以直接访问数据库、通过DAO或使用包装器。

Data Mapping模式有两种：

- Direct Data Mapping（直接数据映射）
- Multi-Step Data Mapping（多步数据映射）

直接数据映射模式在数据要从一个数据库移到另一数据库而不进行变换时处理映射。多步数据映射还变换数据。图6.11显示了Data Mapping模式的结构。

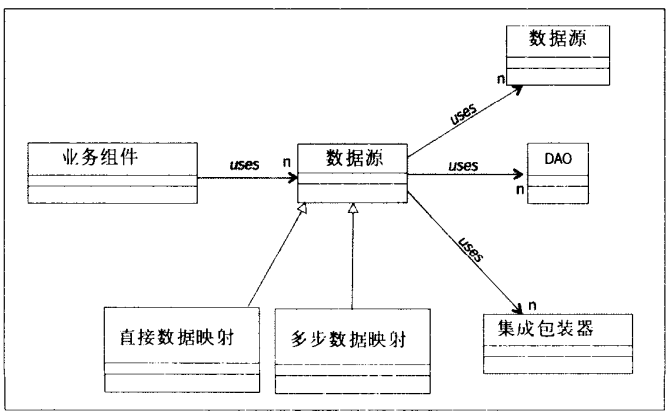

图6.11

后果

结果如下：

- 包装数据映射逻辑，与参与的现有应用程序及其数据库分离。
- 减少数据依赖性。
- 简化维护，数据映射逻辑集中起来，而不是分布在现有应用程序中。
- 业务组件可以使用Data Mapping模式提供的功能，从而不必知道现有应用程序的数据细节。

相关模式

Data Mapping模式的相关模式有Integration Wrapper、Integration Mediator与Data Access Object。

应用模式

Data Mapping模式可以用实体EJB实现。对单步应用程序集成，可以选择Bean和容器管理持久性（BMP, CMP）。对多步Data Mapping模式，只能选择BMP，因为CMP目前不提供所要的功能。可以选择用第三方关系型映射工具和XSLT引擎。

熟悉结构性集成模式之后，下面介绍一下Process Automator模式。

Process Automator模式

下面详细介绍Process Automator模式。

情境

系统交互通常要用过程控制器隐藏和抽象。业务过程控制器与信息系统的系统逻辑之间的依赖性应减到最少。

问题

集成信息系统的服务应通过反映业务过程的高级方法提供给客户机。典型业务过程方法要求与不同虚拟组件和集成调节器交互。这个交互不能委托给客户机，因为这样会增加复杂性、增加维护难度，不能使用声明式事务管理。

要求

要求如下：

- 集成信息系统的服务应通过反映业务过程的高级方法提供给客户机。
- 业务过程交互逻辑应在中间层抽象与包装。
- 客户机不能负责所要的操作调用。
- 业务过程逻辑通常要在事务中进行。
- 过程自动控制与信息系统技术之间的依赖性应减到最少。

方案

Process Automator模式可以收集和包装业务过程逻辑，从而减少业务过程自动逻辑与信息系统技术逻辑之间的依赖性。Process Automator控制器隐藏所有交互。这个模式可以提高业务过程质量、减少过程执行成本和减少开发工作。因此，Process Automator模式特别适合公司内和公司间定义集成过程。

Process Automator模式将业务过程活动排出顺序并将步骤委托给信息系统的相应部件。这是用虚拟组件和集成调节器完成的，自动化组件通过其访问现有应用程序功能。但是，Process Automator模式也可以访问新开发的组件。图6.12显示了Process Automator模式的结构。

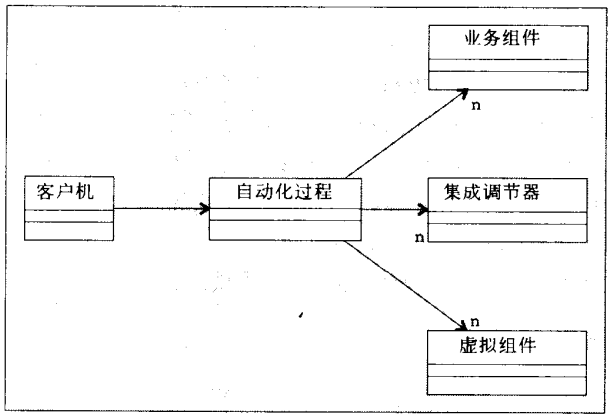

图6.12

Process Automator模式的常见用法是定义业务活动、定时器和过程情境信息，分为两种：

- Closed Process Automator（封闭的Process Automator）
- Open Process Automator（开放的Process Automator）

两者理解两种过程的词法不同。Closed Process Automator实现的过程是内部管理的，只通过数据交换扩展关键过程活动。客户机可以监视过程内的活动，但不能主动参与执行。

Open Process Automator可以在客户机之间共享业务过程，这种过程由多个客户机管理，对多个公司共享一个进程的EAI间或B2B集成特别有用。

后果

结果如下：

- 这个体系结构很容易分析业务过程、瓶颈、使用情况、无效时间，等等。
- 得到重定义业务过程的灵活性。
- Process Automator组件与业务经理的视图对齐，减少IT部门与管理部门的词法差别。
- 利用虚拟组件与集成调节器连接，可以定义高度灵活的集成体系结构。

相关模式

Process Automator的相关模式包括Virtual Component与Integration Mediator。

采用模式

对J2EE采用Process Automator模式要求选择相应技术。在J2EE中，最合适的组件为：

- 会话Bean
- 消息驱动Bean
- Web服务

组件选择取决于需要的交互模型。我们需要同步通信和立即响应请求时，使用会话Bean，根据过程性质选择有状态或无状态会话Bean。如果过程不需要存储任何中间的客户机相关状态，则可以使用无状态会话Bean；如果过程要维护状态，则用状态会话Bean。

如果要长时间的建模，则更新型过程而不需要立即答复，则应选择消息驱动Bean。其可以使建模过程自动化，不需要存储客户机相关状态，不需要立即通知客户机。

J2EE中的Web服务建立在EJB之上，特别适合实现inter-EAI（B2B）过程自动化器。本章稍后会演示如何用两种EJB实现过程自动化器组件。

介绍模式的应用程序之前，先要看看上述集成模式之间的关系。

集成模式之间的关系

要了解集成模式，就一定要了解集成模式之间的关系。下图显示了集成模式之间最主要的关系。可以看出，Process Automator通常依赖于一个或几个Virtual Component，其用Integration Mediator支持与Integration Wrapper和Data Mapping的通信。通常，Data Mapping使用Data Access Object。而Integration Wrapper、Integration Mediator与Virtual Component依赖于Integration Broker模式，如图6.13所示。

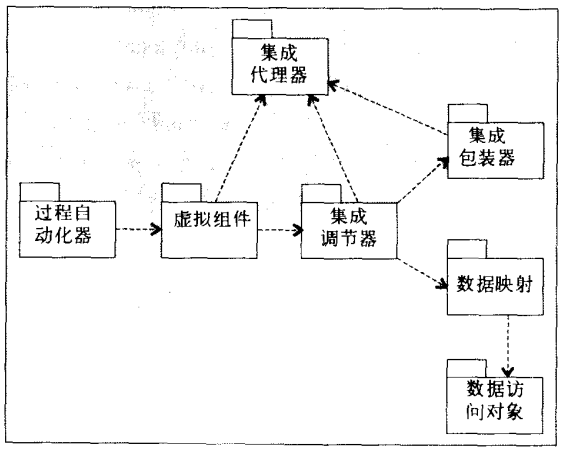

图6.13

要了解如何采用这些模式，我们引入一个案例，用实例实现这些集成模式。

简单集成情形

熟悉集成模式之后，下面看一个简单集成情形，演示如何采用这些模式。这个集成情形经过简化，基于一个假想移动电话经营公司。移动电话经营公司提供数字移动通信（GMS）服务，公司几年前开业时只有模拟服务，几年来不断扩展，现在只提供数字电话。主要服务是两大类移动账户：

- **预订账户**让客户每月收到发票，后面再付款。
- **预付账户**要让客户事先缴费，用完即停机。

信息系统的体系结构没有随公司演变而演变。移动电话经营公司没有集成信息系统，用一组互不连接或部分连接的应用程序（即现有应用程序）。公司使用的主要应用程序如下：

- **Mobile phone customer information application（移动电话客户信息应用程序）**
  可以管理电话客户，包括输入新客户、更新现有客户、增加客户的电话账户数据。
- **Application that manages the account balance for mobile phone users（对移动电话客户管理账户结余的应用程序）**
  管理两种账户，可以输入使用金额和存款的账户（根据账户类型，可以是发票付款或事先存款）。还可以搜索结余为负的账户（对预订账户）。对预付账户，可以搜索一定时间结余为零的账户，这些账户自动切断。最后，还可以搜索不同年龄的账户存款金额。注意这个应用程序和上一应用程序互不连接，需要手工输入。
- **Application that prints the invoices each month（每月打印发票的应用程序）**
  这个应用程序对预订账户打印指定月份的经费发票。
- **Application that evaluates the domestic mobile phone calls（求值市话电话的应用程序）**
  这个应用程序利用移动电话交换传输的数据计算每个账户的费用（仅对市话）。
- **Application that calculates the international mobile phone calls（求值长话电话的应用程序）**
  这个应用程序与市话应用程序相似，但与移动电话交换没有直接连接，而是从漫游伙伴接收输入数据。
- **Application that manages the mobile phone calls tariff information（管理移动电话话费信息的应用程序）**
  包括输入与显示话费信息，包括话费政策、话费类别和声音与数据通信的每分钟费用。

一些主要应用程序根本没有集成，因此用户要在不同应用程序中多次手工输入数据。有些主要应用程序提供基于共享数据文件的简单数据交换，用专属格式格式化。图6.14显示了应用程序及其依赖性，注意Manual表示要求手工输入数据，而Automatic表示自动数据传输。

这组应用程序无法满足几个要求，不提供集成信息系统所达到的效率。但我们可以看到，本章无法介绍集成系统的所有要求，而是主要考虑两个重要功能，这是现有系统不提供的，要用集成模式描述。

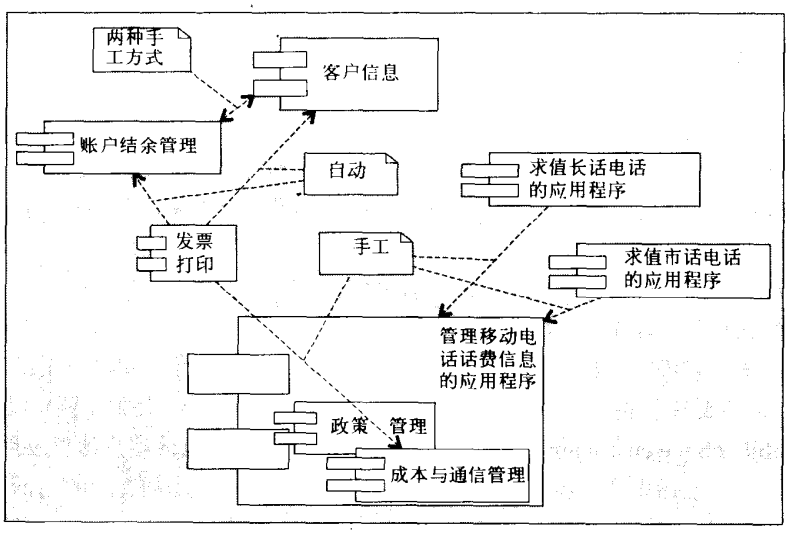

图6.14

- 移动电话用户要随时联机用移动电话（WAP或SMS）或PC（Internet浏览器）检查账户结余。提供的信息必须准确，包括所有市话与长话，以及任何折扣（例如根据每月电话用量或最近六个月的平均电话用量）。
- 移动电话用户要用移动电话（WAP或SMS）或PC（Internet浏览器）在移动电话账户上存款，从用户输入的信用卡中取钱，只有信用卡授权成功时才进行存钱。

还可以方便地看到两个用例都要进行授权和选择账号。然后用例框图6.15所示。

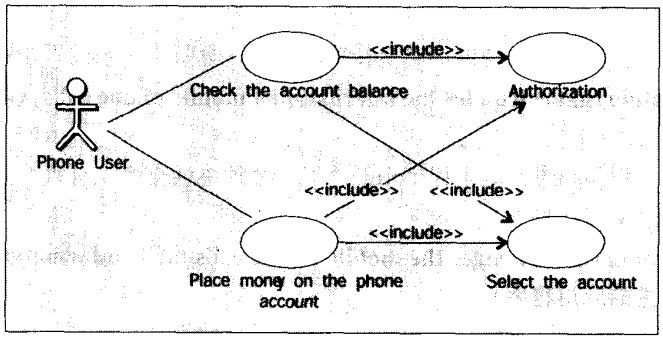

图6.15

模式标识与设计

对每个用例，要标识模式并根据这些模式设计集成体系结构。设计活动的主要任务用下列步骤完成：

- 标识模式
- 标识组件

- 标识现有应用程序依赖性
- 确定体系结构

下面先看看例子中的下面两个用例：

- Authorization（accessing the customer information）（授权，访问客户信息）
- Check Account Balance（检查账户结余）

授权（访问客户信息）

从集成角度看，授权与客户信息访问相连接，因此，我们看看现有应用程序中存储的客户信息如何显示。

我们要使用下列集成模式：

- Virtual Component模式
- Data Mapping模式
- Data Access Object（DAO）模式
- Integration Wrapper模式

注意这里只考虑集成模式。由于这些集成模式用J2EE技术实现，因此我们可以采用前面几章介绍的一般J2EE设计模式。

然后要标识组件。高级组件可以从分析模型标识。问题是，尽管这个任务好像很简单，但其实不然。选择正确的组件对信息系统的整体具有长期影响。这个选择还确定集成体系结构是否适合改造现有应用程序和换成新开发的方案。

我们的移动电话公司例子比较简单，因此标识虚拟组件也比较简单。我们用CustomerVC虚拟组件表示J2EE客户机的客户。

现在要标识现有应用程序和数据库的依赖性。可以看到所要的数据存放在账户结余管理数据库和客户信息应用程序中，如图6.16所示。

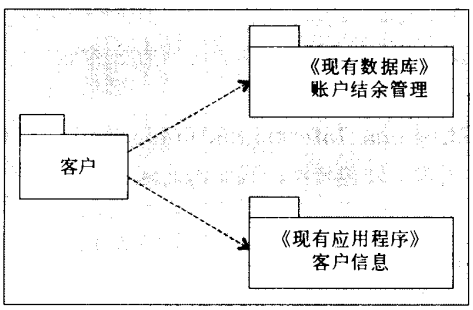

图6.16

可以看到，我们需要访问账户结余管理应用程序。为此，我们用Integration Wrapper模式。由于我们只访问数据，因此我们用DAO抽象这个交互。我们还用DAO访问现有客户信息数据库。为了从两个应用程序中取得、比较和连接数据，我们用Data Mapping模式。这样就得到如图6.17的设计。

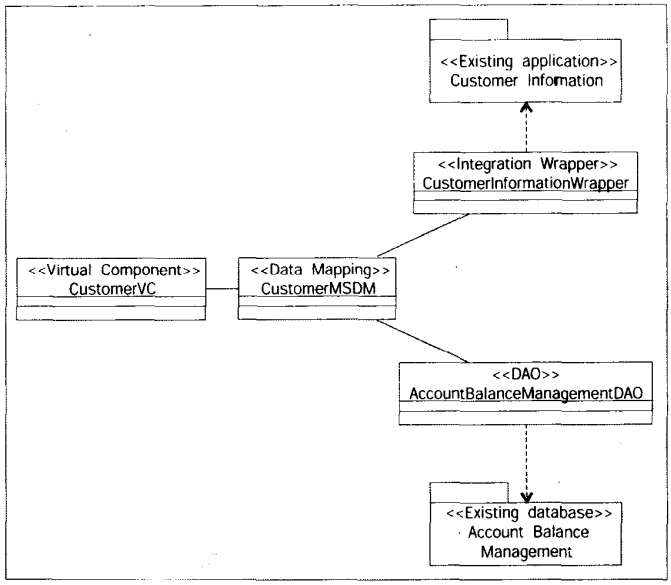

图6.17

下面要选择技术。CustomerVC虚拟组件用实体Bean和Bean管理持久性（BMP）。实体Bean适合表示实体组件，存储事务状态，在客户机之间共享。我们用BMP而不用CMP，因为我们要访问现有应用程序，映射两个不同应用程序中的数据。

我们访问数据并用Multi-Step Data Mapping模式进行必要的数据映射，实现为依赖对象。我们对DAO对象也使用依赖对象。这时实体Bean采用粗粒方式，减少远程方法调用。这个方法适合EJB 1.1与EJB 2.0。

另一种方法是使用本地接口和容器管理关系。但这个方法只限于EJB 2.0，性能稍差一些，因为容器仍然要涉及本地方法调用。关于这个方法的细节见《Professional EJB》一书Wrox Press出版（ISBN 1-861005-08-3）。

最后，我们要实现CustomerInformationWrapper，作为CORBA集成包装器。利用CORBA可以用不同编程语言开发包装器，我们需要同步访问现有应用程序。CORBA适合这个任务。

下面介绍Check Account Balance用例。

Check Account Balance用例

上面介绍了检查账户结余需要与几个现有应用程序交互。因此我们要以正确方式访问这些应用程序，协调整个交互逻辑。我们还要把这个功能表示为其他J2EE客户机能够访问的高级服务。

我们使用下列模式：

- Process Automator（过程自动化器）

- Virtual Component（虚拟组件）
- Integration Mediator（集成调节器）
- Integration Wrapper（集成包装器）

然后我们要标识高级组件，完成分析类框图（这里没有显示）。可以找到下列组件：

- CheckBalance组件实现Process Automator模式。
- AccountVC（账户实体组件）实现Virtual Component模式。

要标识其他组件，我们首先要标识相关的现有应用程序。我们主要考虑检查某个移动电话账户账户结余的操作。现有应用程序支持这个功能，但是这个功能没有放在一个现有应用程序中，因此这个组件依赖于多个现有应用程序。

要计算移动电话账户目前账户结余，组件要与现有应用程序交互，图6.18显示了组件的依赖性。

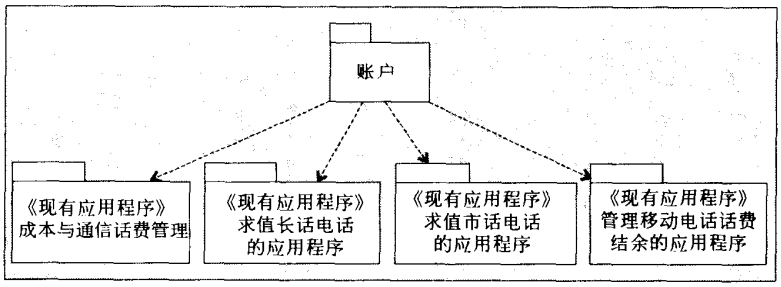

图6.18

这些依赖性可以标识其他需要的组件。要与现有应用程序交互，我们用集成包装器，可以编程访问。对市话与长话话费计算，我们用Integration Mediator模式，与两个现有应用程序协调交互逻辑。

这样就得到如图6.19所示的设计。

然后要对每个组件选择J2EE技术。本章前面介绍模式时曾介绍过，每个模式几乎都可以选择不同技术。因此本例选择如下：

- CheckBalance Process Automator模式用会话Bean。会话Bean适合状态和无状态Process Automator组件。这里要求同步请求/响应通信，因此最好选择会话Bean。也可以在需要异步通信时使用消息驱动Bean。
- AccountVC Virtual Component模式用BMP实体Bean，因为AccountVC要表示共享事务状态，实体Bean最适合这个工作。AccountVC要从不同的现有应用程序收集数据，因此不能用CMP。
- MobilePhoneEvaluationMediator Integration Mediator模式用RMI-IIOP分布式对象。我们选择RMI-IIOP是因为不希望会话Bean的容器开销，在更复杂的情形中，可能需要线程控制和同步。也可以用会话Bean或CORBA，但我们相信这也可以演示相互操作性。
- 所有集成包装器用CORBA包装器。我们用CORBA，因为需要同步通信。CORBA提供与RMI-IIOP的相互操作性，不限于Java语言，这对可开发包装器非常重要。

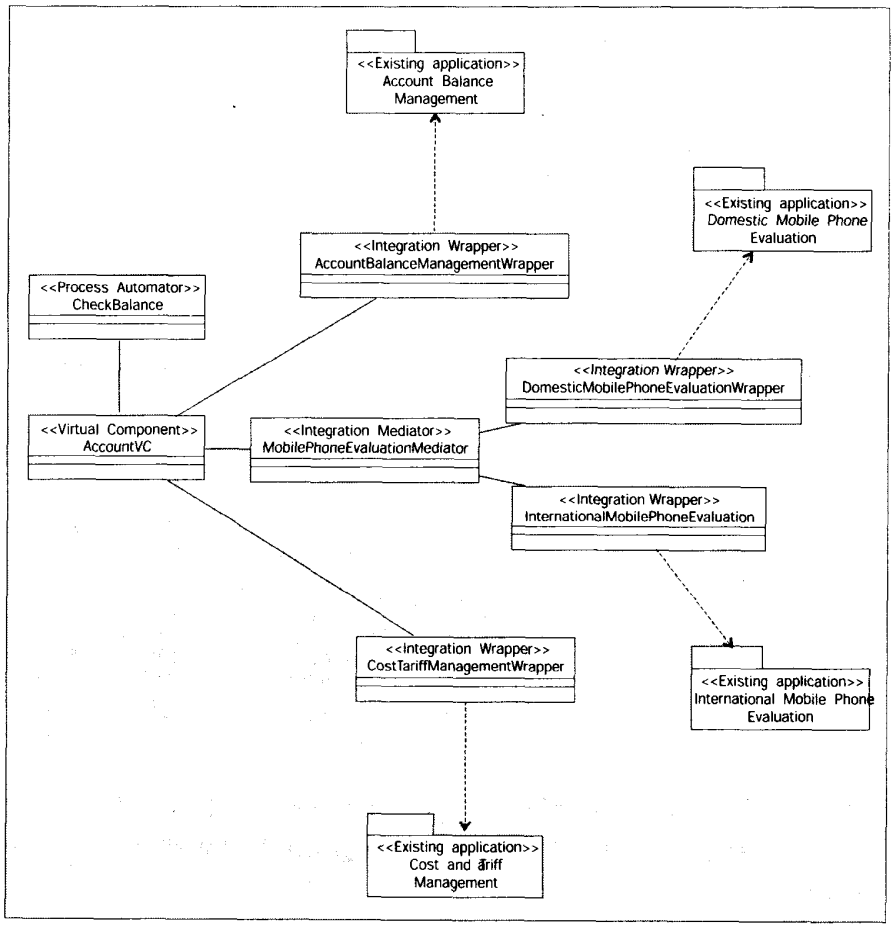

图6.19

特别地，集成包装器技术的选择取决于现有应用程序的体系结构，但我们选择CORBA，因为它在这方面广泛使用，支持各种编程语言的操作系统和平台。下面介绍如何实现这些模式。

实现集成模式

实现集成模式时有一定的限制，因为我们无法访问现有应用程序。因此，我们演示原型，显示开发特定组件类型并在J2EE中采用上述集成模式的概念。但我们不实现与现有应用程序的交互，而是在代码中加上说明。

实现Integration Wrapper模式

本节介绍如何用CORBA开发客户集成包装器。建立CORBA集成包装器（或组件包装

器）类似于建立通用CORBA分布式对象。我们用Java SDK 1.3所带的Java IDL建立简单的组件包装器。

定义接口

要建立集成包装器，我们使用现有应用程序提供的相同API。CORBA IDL如下：

```
module CustomerManagement {

  interface Customer {

    enum statusType {normal, silver, gold};

    enum typeOfAccounts {subscription, prepaid};

    struct dateType {
       short date;
       short month;
       short year;
    };

    exception customerNotFound {
       string reason;
       short errorCode;
    };

    exception customerNotUpdated {
       string reason;
       short errorCode;
    };

    exception accountNotAdded {
       string reason;
       short errorCode;
    };

    exception accountNotDeleted  {
       string reason;
       short errorCode;
    };

    void getCustomer ( in long customerId,
                       out wstring firstName,
                       out wstring lastName,
                       out wstring address,
                       out short numberOfPhoneAccounts,
                       out statusType status)
       raises (customerNotFound);

    long setCustomer ( in wstring firstName,
                       in wstring lastName,
```

```
                        in wstring address,
                        in short numberOfPhoneAccounts,
                        in statusType status)
        raises (customerNotUpdated);

    void newAccount ( in wstring phoneNumber,
                      in wstring startDate,
                      in typeOfAccounts typeOfAccount)
        raises (accountNotAdded);

    void deleteAccount ( in wstring phoneNumber)
        raises (accountNotDeleted);

  };
};
```

IDL编译器将IDL映射为编程语言，这是每个CORBA实现都有的。要编译定义的IDL接口，我们用JDK1.3以上版本所带的idlj编译器。要生成两个映射（客户端和服务器方），我们用-fall选项生成所有绑定。

连接现有应用程序

现在要将现有应用程序的操作与接口中已经定义的操作连接起来。这个例子也可以通过专属API或通过源代码修改。屏幕刮取和终端仿真访问现有应用程序。要实现组件包装器，我们要定义一个类，实现IDL接口的功能。我们把这个类称为CustomerImpl，继承_CustomerImplBase类，是IDL编译器提供的连接代码的例子如下：

```
package CustomerManagement;

public class CustomerImpl extends _CustomerImplBase {

  public void getCustomer(int customerId, org.omg.CORBA
          .StringHolder firstName, org.omg.CORBA.StringHolder lastName, org
            .omg.CORBA.StringHolder address, org.omg.CORBA
              .ShortHolder numberOfPhoneAccounts, CustomerManagement
                .CustomerPackage.statusTypeHolder status)
                  throws CustomerManagement.CustomerPackage
                    .customerNotFound {

    // Connect to the existing applicaiton
    // Locate the customer and retrieve the data
    Boolean successful = true;   // Determined from existing application

    if (successful) {

      // Fill in the data
      firstName.value = "Jack";
      lastName.value = "B. Good";
      address.value = "Warwick Road";
      numberOfPhoneAccounts.value = 2;
```

```
        status.value = CustomerManagement.CustomerPackage.statusType.silver;
      } else {
        throw new CustomerManagement.CustomerPackage
          .customerNotFound("Bad customerID", (short) 1);
      }
    }

    public int setCustomer(String firstName, String lastName, String address,
        short numberOfPhoneAccounts, CustomerManagement
            .CustomerPackage.statusType status)
              throws CustomerManagement.CustomerPackage.customerNotUpdated {

      // Connect to the existing application
      // Generate the customerID
      // Add the customer data
      int customerId = 1;
      Boolean successful = true;

      if (successful) {
        return customerId;
      } else {
        throw new CustomerManagement.CustomerPackage
          .customerNotUpdated("DB error", (short) 1);
      }
    }

    public void newAccount(String phoneNumber, String startDate,
        CustomerManagement.CustomerPackage.typeOfAccounts typeOfAccount)
          throws CustomerManagement.CustomerPackage.accountNotAdded {

      // Similar to above example
    }

    public void deleteAccount(String phoneNumber)
            throws CustomerManagement.CustomerPackage.accountNotDeleted {

      // Similar to above example
    }
}
```

实现服务器过程

这个阶段几乎已经实现组件包装器，但还要实现执行CORBA分布式对象的过程，要创建对象的至少一个实例。我们定义名称为Server的类，要取得ORB的引用，创建Customer包装器的新实例，并将这个实例连接ORB。这很简单，代码如下：

```
package CustomerManagement;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
```

```
import org.omg.CORBA.*;

public class Server {

  public static void main(String[] args) {

   try {
      ORB orb = ORB.init(args,null);

      Customer c = new CustomerImpl();

      orb.connect(c);
```

但这还不够，具体地说，我们要让客户机取得初始引用。可以用命名服务，代码如下：

```
      org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      NameComponent nc;
      NameComponent path[] = {
        Null
      };

      nc = new NameComponent("CustomerWrapper", "");
      path[0] = nc;
      ncRef.rebind(path, c);

      System.out.println("Server is ready...");
```

记住不能让服务器过程结束，因此要让过程保持活动，一种方法如下：

```
      java.lang.Object sync = new java.lang.Object();
      synchronized (sync) {
        sync.wait();
      }
```

最后还要捕获任何异常：

```
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

实现测试组件包装器的客户机

开发组件包装器之后，还要测试。为此，可以开发一个小CORBA客户机，调用几个方法。本例中，我们捕获CORBA用户异常、CORBA系统异常和所有其他异常：

```
package CustomerManagement;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;
```

```
public class Client {
  public static void main(String args[]) {
    try {

      // Connect to the wrapper
      ORB orb = ORB.init(args, null);

      org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      NameComponent nc = new NameComponent("CustomerWrapper", "");
      NameComponent path[] = {
        nc
      };
      Customer c = CustomerHelper.narrow(ncRef.resolve(path));

      // test the functionality
      StringHolder firstName = new StringHolder();
      StringHolder lastName = new StringHolder();
      StringHolder address = new StringHolder();
      ShortHolder numberOfPhoneAccounts = new ShortHolder();
      CustomerManagement.CustomerPackage.statusTypeHolder status =
        new CustomerManagement.CustomerPackage.statusTypeHolder();

      c.getCustomer((int) 1, firstName, lastName, address,
                    numberOfPhoneAccounts, status);
      System.out.println("First name: " + firstName.value);
      System.out.println("Last name:  " + lastName.value);
      System.out.println("Address:    " + address.value);
      System.out.println("No. of phone accounts: "
                         + numberOfPhoneAccounts.value);

      int customerId =
        c.setCustomer("Jack", "B. Good", "Birmingham", (short) 1,
                      CustomerManagement.CustomerPackage.statusType.gold);
      System.out.println("Setting new customer... id: " + customerId);

    } catch (UserException e) {
      System.out.println("User exception: " + e);

    } catch (SystemException e) {
      System.out.println("System exception: " + e);

    } catch (Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

要运行这个例子，首先要启动命名服务。Java SDK 1.3提供了一个临时命名服务tnameserv。然后在设置相应类路径之后启动Server.java与Client.java，分别在不同进程中，如图6.20所示。

图6.20

如果要在不同机器上启动客户机和服务器，则要用-ORBIntialHost开关提供命名服务的地址。也可以用-ORBInitialPort开关手工选择端口。

下节介绍如何实现Integration Mediator模式。

实现Integration Mediator模式

本节要介绍如何实现本章前面标识的移动电话求值调节器模式。我们用RMI-IIOP实现，也可以用会话Bean。之所以用RMI-IIOP，是因为我们不想出现容器开销。我们开发MobilePhoneEvaluation调节器，其中的单个操作是evaluateConsumption()。调节器连接现有市话和长话计算应用程序，并通过必要步骤计算和求值电话话费。

定义接口和连接现有应用程序

首先要定义相应RMI-IIOP接口。为了从远程客户机访问Java接口，接口要从java.rmi.Remote派生：

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface MobilePhoneEvaluation extends Remote {

  double evaluateConsumption (Date fromDate, Date toDate)
                  throws RemoteException;

}
```

下一步要实现接口的功能，即将接口方法连接现有应用程序。为此我们使用前面介绍的包装器。为此，我们声明一个类，实现上述MobilePhoneEvaluation接口。为了使这个类成为RMI-IIOP类（连接RMI-IIOP对象请求代理），实现类要从javax.rmi.PortableRemoteObject继承，并实现接口。实现类如下：

```
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;
import java.util.Date;

public class MobilePhoneEvaluationImpl extends PortableRemoteObject
    implements MobilePhoneEvaluation  {

  public MobilePhoneEvaluationImpl() throws RemoteException {
  }

  public double evaluateConsumption (Date fromDate, Date toDate)
  throws RemoteException {

     double cost = 0;
     // Connect to the domestic mobile phone evaluation existing application
     // Connect to the international mobile phone evaluation existing
     // application

     cost = 55;

     return cost;
  }
}
```

在实际中使用这个调节器组件时，要增加代码，将其实际连接现有应用程序。这对不同现有应用程序是不同的。然后我们要建立这个组件的实例，进行注册，使客户机能够找到。

RMI-IIOP组件需要一个执行过程。因此我们要提供一个服务器应用程序，创建一个服务器进程，用于对**RMI-IIOP**组件实现类实例化一个或几个实例。代码如下：

```
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;
import javax.naming.*;

public class Server {

  public static void main(String[] args) {

    try {

      Context iNamingContext = new InitialContext();

      MobilePhoneEvaluation mpe = new MobilePhoneEvaluationImpl();

      iNamingContext.rebind("MobilePhoneEvaluation", mpe);

      System.out.println("Server ready...");

    } catch (Exception e) {
      System.out.println(e);
      e.printStackTrace(System.out);
    }
  }
}
```

这样就基本完成了，但还要开发一个简单客户机，测试RMI-IIOP虚拟组件的功能。

开发客户机

要测试虚拟组件的功能，就要开发一个简单的RMI-IIOP客户机。这个客户机连接命名服务，解析名称，并调用远程操作。代码如下：

```
import java.util.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class Client {

  public static void main(String[] args) {

    MobilePhoneEvaluation mpe = null;

    try {
      Context inc = new InitialContext();
    mpe = (MobilePhoneEvaluation)
              PortableRemoteObject.narrow(inc.lookup(
                 "MobilePhoneEvaluation"),MobilePhoneEvaluation.class);
    } catch (Exception e) {
    System.out.println(e);
    }

    try {

      double r = mpe.evaluateConsumption(
                      (new GregorianCalendar(2,4,1)).getTime(),
                      (new GregorianCalendar(2,4,1)).getTime());
      System.out.println(r);

    } catch (Exception e) {
    System.out.println(e);
    }
  }
}
```

编译源代码之后，要用Java RMI Compiler或rmic产生残根与框架。关于rmic的更详细信息，见Java SDK文档。例中，我们使用-iiop开关。

启动服务器对象和客户机之前，要先激活命名服务提供者。为此可以用临时CORBA Naming Service, tnameserv。最后，我们可以启动服务器类，实例化RMI-IIOP组件。我们要指定初始命名工厂和命名服务提供者URL。要启动客户机，也还要指定初始命名工厂和命名服务提供者URL（注意，如果要在不同计算机上运行，就要相应修改URL），得到图6.21和图6.22奇妙的结果。

下面要演示如何开发Data Mapping模式。

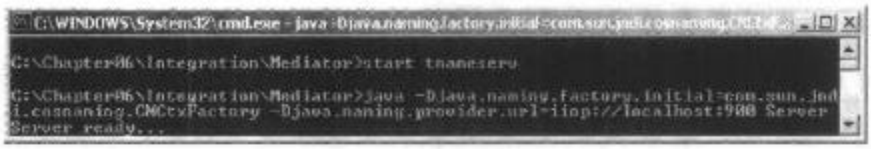

图6.21

图6.22

实现Data Mapping模式

我们用实体EJB实现Data Mapping模式，显示单步与多步情形。对单步数据映射，我们用CMP；对多步数据映射，我们用BMP。

定义接口

例子的远程组件接口如下：

```
package cmp1;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Customer extends EJBObject {

  public String getCustomerPrimaryKey() throws RemoteException;
  public String getCustomerName() throws RemoteException;
  public void setCustomerName(String name) throws RemoteException;
  public String getCustomerAddress() throws RemoteException;
  public void setCustomerAddress(String address) throws RemoteException;
  public int getCustomerNoOfPhoneAccounts() throws RemoteException;
  public void setCustomerNoOfPhoneAccounts(int noOfPhAcc)
                                          throws RemoteException;
  public CustomerData getCustomerData() throws RemoteException;
  public void setCustomerData(CustomerData custData) throws RemoteException;
}
```

还要声明数值对象CustomerData，声明如下：

```
package cmp1;

public class CustomerData implements java.io.Serializable {

   private String customerId = null;
   private String name = null;
```

```
    private String address = null;
    private int noOfPhoneAccounts = 0;

    public CustomerData () {
    }

    public CustomerData (String id, String nm, String ad, int na) {
       customerId = id;
       name = nm;
       address = ad;
       noOfPhoneAccounts = na;
    }

    public String getCustomerId() {
      return customerId;
    }
    public void setCustomerId(String id) {
      if (customerId==null)
        customerId = id;
    }

    public String getName() {
      return name;
    }
    public void setName(String nm) {
      name = nm;
    }

    public String getAddress() {
      return address;
    }
    public void setAddress(String ad) {
      address = ad;
    }

    public int getNoOfPhoneAccounts() {
      return noOfPhoneAccounts;
    }
    public void setNoOfPhoneAccounts(int pa) {
      noOfPhoneAccounts = pa;
    }

}
```

然后要定义主接口，提供创建与寻找实体Bean的方法。本例中，我们定义两个创建方法：一种方法取一个又一个参数变元，另一种方法取数值对象CustomerData的四个组成部分作为变元。我们还定义两个寻找方法：一个按主键找到实体Bean，一个按名称找到实体Bean。后者返回Collection，因为几个客户可能同名。远程主接口如下：

```
package cmp1;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;
import java.util.Collection;
import java.rmi.RemoteException;
public interface CustomerHome extends EJBHome {

  public Customer create(String customerId, String name, String address,
                         int noOfPhAcc)
      throws RemoteException, CreateException;

  public Customer create(CustomerData custData)
      throws RemoteException, CreateException;

  public Customer findByPrimaryKey(String primaryKey)
      throws RemoteException, FinderException;

  public Collection findByName(String name)
      throws RemoteException, FinderException;

}
```

本例中，我们用社会安全号（SSN）作为客户的主键（CustomerId）。

开发CMP的实现类

实现类中要提供组件接口中所有方法的实现。在实现类中，我们还实现主接口的方法和必要的容器回调方法，包括ejbLoad()、ejbStore()、ejbActivate()、ejbPassivate()、setEntityContext()、unsetEntityContext()、ejbCreate()、ejbRemove()与ejbFindXXX()方法。

如果使用CMP，则不必实现上述所有方法；容器会提供ejbLoad(), ejbStore()与ejbFindXXX()方法的实现，但我们要在部署描述项中指定其他信息。此外，还要定义属性。如果在DJB2.0中使用CMP，则要定义所有持久属性的抽象获取与设置方法，见稍后介绍。

要实现Customer bean的类，我们定义CustomerBean抽象类，实现EntityBean接口。这个类是抽象的，因为容器要实现几个方法，见稍后介绍。我们声明一个专用属性，存储情境，还有一个空构造函数：

```
package cmp1;

import javax.ejb.*;

abstract public class CustomerBean implements EntityBean {

  private EntityContext ctx;

  public CustomerBean() {
  }
```

要继续实现实体Bean实现类，我们要定义容器管理属性。在EJB2.0中，使用CMP时，

我们要对每个属性定义获取与设置方法时。在Bean部署时，容器会提供这些方法的实现。这样就要把这些方法声明为抽象方法，从而要把整个类声明为抽象类：

```
// Container-managed fields
abstract public String getCustomerId();
abstract public void setCustomerId(String customerID);
abstract public String getName();
abstract public void setName(String name);
abstract public String getAddress();
abstract public void setAddress(String address);
abstract public int getNoOfPhoneAccounts();
abstract public void setNoOfPhoneAccounts(int noOfPhAcc);
```

下一步要提供组件接口中方法的实现。由于实体Bean比较简单，因此实现这些方法也比较简单。获取与设置属性的细粒方法用前面定义的抽象获取与设置方法的简单方法调用实现：

```
// Component interface
public String getCustomerPrimaryKey() {
  return getCustomerId();
}
public String getCustomerName() {
  return getName();
}
public void setCustomerName(String name) {
  setName(name);
}
public String getCustomerAddress() {
  return getAddress();
}
public void setCustomerAddress(String address) {
  setAddress(address);
}
public int getCustomerNoOfPhoneAccounts() {
  return getNoOfPhoneAccounts();
}
public void setCustomerNoOfPhoneAccounts(int noOfPhAcc) {
  setNoOfPhoneAccounts(noOfPhAcc);
}
```

此外，我们还定义两个粗粒方法，使用数值对象CustomerData。其实现如下：

```
public CustomerData getCustomerData() {
  CustomerData cd = new CustomerData();
  cd.setCustomerId(getCustomerId());
  cd.setName(getName());
```

```
    cd.setAddress(getAddress());
    cd.setNoOfPhoneAccounts(getNoOfPhoneAccounts());
    return cd;
  }

  public void setCustomerData(CustomerData cd) {
    setCustomerId(cd.getCustomerId());
    setName(cd.getName());
    setAddress(cd.getAddress());
    setNoOfPhoneAccounts(cd.getNoOfPhoneAccounts());
  }
```

然后要实现主接口的方法。我们不实现**ejbFindXXX()**方法，但要实现**ejbCreate()**方法。前面曾介绍过，例子中有两个不同方法：一种方法取一个又一个变元，另一种方法取数值对象**CustomerData**的四个组成部分作为变元。实现如下：

```
  // home methods
  public String ejbCreate(String customerId, String name, String address,
                          int noOfPhAcc) throws CreateException {
    setCustomerId(customerId);
    setName(name);
    setAddress(address);
    setNoOfPhoneAccounts(noOfPhAcc);
    return null;
  }

  public void ejbPostCreate(String customerId, String name,
                            String address, int noOfPhAcc) {
  }

  public String ejbCreate(CustomerData cd) throws CreateException {
    setCustomerId(cd.getCustomerId());
    setName(cd.getName());
    setAddress(cd.getAddress());
    setNoOfPhoneAccounts(cd.getNoOfPhoneAccounts());
    return null;
  }

  public void ejbPostCreate(CustomerData cd) {
  }
```

还要声明前面介绍的其他**EJB**相关方法。好在这些方法可以保持空白，因为我们使用**CMP**，容器会实现这些方法：

```
  public void setEntityContext(EntityContext ctx) {
    this.ctx = ctx;
  }

  public void unsetEntityContext() {
```

```
    this.ctx = null;
  }
  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void ejbLoad() {
  }

  public void ejbStore() {
  }

  public void ejbRemove() throws RemoveException {
  }
}
```

这样就完成了Bean实现。下一步要定义合适的部署描述项。在部署描述项中，要提供在容器中部署企业Bean所需的其他信息。部署描述项放在本章的源代码文件中。

部署Bean之前，要先编译创建的类，将EJB包装到JAR文件中。前面曾介绍过，部署中的其余步骤是应用程序服务器特定的。EJB开发人员要按照相应应用程序服务器文档中的说明部署Bean。通常，这个过程包括调用EJB编译器或使用GUI接口。

实体Bean的客户机应用程序

顺利部署之后，要测试实体Bean。为此要编写一个简单客户机。下面的客户机应用程序调用实体Bean远程组件接口中的一些方法。要测试本地接口，就要开发另一个企业Bean，在同一容器中部署：

```
package cmp1;

import java.rmi.*;
import javax.rmi.*;
import java.util.*;

import javax.ejb.*;
import javax.naming.*;

public class Client {

  public Client() {
  }

  public static void main(String[] args) {
    System.out.println("EAI CMP1 client");
    System.out.println();

    try {
      // Creates a new instance
```

```
        Client client = new Client();

        // Invokes the example() method
        client.example();

      } catch (Exception e) {
        System.out.println(e);
      }
    }

    public void example() {
      try {
        // Creates initial naming context
        Context ctx = new InitialContext();

        // Resolves to the home object and narrows
        CustomerHome home = (CustomerHome)PortableRemoteObject.narrow(
          (CustomerHome)ctx.lookup("eail-CustomerHome"), CustomerHome.class);

        // Creates a new bean
        Customer c = home.create("1","Matjaz","Wrox",5);

        // Invokes a method and writes the result to console
        System.out.println("Customer name is: "+c.getCustomerName());

        // Removes the bean
        c.remove();

      } catch(Exception e) {
        System.out.println(e);
      }
    }
  }
```

得到的结果如图6.23所示。

图6.23

用BMP实现多步数据映射

容器管理持久性无法满足集成的所有需求，CMP很快会遇到限制，特别是使用更复杂的关系型结构和存放成非关系型结构的数据时。这时只好使用Bean管理持久性。利用BMP，我们不用容器持久性服务，而是自己实现所要的方法，特别是ejbLoad()与ejbStore()方法，还要手工实现寻找方法。

但是，组件与主接口定义是相同的，因此主要改变发生在实现类和部署描述项中。下

面看看如何编写这个例子的BMP实现类CustomerBean。和上例一样，我们用数值对象CustomerData。BMP实现类如下：

```
package bmp;

import javax.ejb.*;
import java.util.*;

public class CustomerBean implements EntityBean {

  private EntityContext ctx;

  public CustomerBean() {
  }

  private CustomerData cd;
```

然后定义组件接口方法：

```
  // Component interface
  public String getCustomerPrimaryKey() {
    return cd.getCustomerId();
  }
  public String getCustomerName() {
    return cd.getName();
  }
  public void setCustomerName(String name) {
    cd.setName(name);
  }
  public String getCustomerAddress() {
    return cd.getAddress();
  }
  public void setCustomerAddress(String address) {
    cd.setAddress(address);
  }
  public int getCustomerNoOfPhoneAccounts() {
    return cd.getNoOfPhoneAccounts();
  }
  public void setCustomerNoOfPhoneAccounts(int noOfPhAcc) {
    cd.setNoOfPhoneAccounts(noOfPhAcc);
  }

  public CustomerData getCustomerData() {
    return cd;
  }
  public void setCustomerData(CustomerData cd) {
    this.cd = cd;
  }
```

现在要定义主方法。ejbCreate()方法与CMP例子中非常相似：

```
// home methods
public String ejbCreate(String customerId, String name,
                String address, int noOfPhAcc) throws CreateException {
  // Connect to the existing database
  // and create a new record
  // Store this record in the value object
  cd = new CustomerData();
  cd.setCustomerId(customerId);
  cd.setName(name);
  cd.setAddress(address);
  cd.setNoOfPhoneAccounts(noOfPhAcc);
  return customerId;
}

public void ejbPostCreate(String customerId, String name, String address,
                          int noOfPhAcc) {
}

public String ejbCreate(CustomerData cd) throws CreateException {
  // Connect to the existing database
  // and create a new record
  // Store this record in the value object
  this.cd = new CustomerData();
  this.cd = cd;
  return cd.getCustomerId();
}

public void ejbPostCreate(CustomerData cd) {
}
```

但我们还要实现ejbFindXXX()方法，因此要声明和实现两个方法ejbFindByPrimaryKey()与ejbFindByName()。这里要用任何适合的机制访问数据库，最常见的方法是使用JDBC。另一方面，要访问现有数据，有时要使用其他方法，包括访问其他类型的数据库或调用数据访问的包装器或DAO。

我们不列出用JDBC访问数据库的代码，因为这很简单。但值得一提的是，从性能角度看最好对JDBC使用准备语句。所要方法的框架如下（完整代码见下载包）：

```
public String ejbFindByPrimaryKey(String pk)
     throws ObjectNotFoundException {
  // Access the existing database
  // Locate the record by primary key
  // Return the primary key
  return "1";
}

public Collection ejbFindByName(String name) {
  Vector v = new Vector();
```

```
        // Access the existing database
        // Locate the records
        // Add primary keys to the vector
        v.addElement("1");
        // Return the vector
        return v;
    }
```

然后要实现ejbLoad()与ejbStore()方法，分别装入与更新数据库状态，使用与寻找方法相同的机制：

```
    public void ejbLoad() {
        cd = new CustomerData();

        // Access the existing databases and applications and load the data

        // Store the data to the value object
        cd.setCustomerId("1");
        cd.setName("Matjaz");
        cd.setAddress("Wrox");
        cd.setNoOfPhoneAccounts(5);
    }

    public void ejbStore() {
        // Access the existing databases and applications
        // Update the data from the bean to the database
    }
```

最后要提供其他EJB相关方法：

```
    public void setEntityContext(EntityContext ctx) {
        this.ctx = ctx;
    }

    public void unsetEntityContext() {
        this.ctx = null;
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() throws RemoveException {
    }
}
```

对BMP实体Bean，我们还要定义不同于CMP Bean部署描述项的部署描述项。我们不必指定容器管理属性的所有细节和CMP Bean所要的寻找方法，因此部署描述项实际上很简

单，这里没有列出，可以从本章源代码中找。

下面介绍如何实现Virtual Component模式。

实现Virtual Component模式

本节演示如何用EJB开发虚拟组件。我们开发Customer虚拟组件并显示如何访问前面用CORBA开发的包装器。由于Customer虚拟组件的性质，我们用BMP实体Bean。我们利用上节建立的例子，因此主接口与远程接口保持相同。

要开发Customer虚拟组件实体Bean（CustomerBean类），就要导入org.omg.CosNaming与org.omg.CORBA包和声明CustomerManagement.Custome类型的专用属性cust。属性cust存储CORBA组件包装器Customer的引用。代码开始如下：

```
package bmp2;

import javax.ejb.*;
import java.util.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class CustomerBean implements EntityBean {

  private EntityContext ctx;
  private CustomerData cd;
  private CustomerManagement.Customer cust;
```

然后要实现组件接口，和上例中的做法一样：

```
  // Component interface
  public String getCustomerPrimaryKey() {
    return cd.getCustomerId();
  }
  public String getCustomerName() {
    return cd.getName();
  }
  public void setCustomerName(String name) {
    cd.setName(name);
  }
  public String getCustomerAddress() {
    return cd.getAddress();
  }
  public void setCustomerAddress(String address) {
    cd.setAddress(address);
  }
  public int getCustomerNoOfPhoneAccounts() {
    return cd.getNoOfPhoneAccounts();
  }
  public void setCustomerNoOfPhoneAccounts(int noOfPhAcc) {
```

```
    cd.setNoOfPhoneAccounts(noOfPhAcc);
  }

  public CustomerData getCustomerData() {
    return cd;
  }
  public void setCustomerData(CustomerData cd) {
    this.cd = cd;
  }
```

然后提供EJB相关方法。在setEntityContext()中，我们取得与ORB的连接并连接组件包装器，这里是个无状态包装器。如果是状态包装器，则要用ejbActivate()方法。这里ejbActivate()是空的。我们还分别在unsetEntityContext()与ejbPassivate()中释放资源（这里ejbPassivate()是空的）。

```
public void setEntityContext(EntityContext ctx) {
  this.ctx = ctx;
  try {
     // Connect with the ORB
     String args[] = new String[1];
     args[0] = "";
     ORB orb = ORB.init(args, null);

     // Connect with the component wrapper if stateless
     org.omg.CORBA.Object objRef =
                          orb.resolve_initial_references("NameService");
     NamingContext ncRef = NamingContextHelper.narrow(objRef);

     NameComponent nc = new NameComponent("CustomerWrapper", "");
     NameComponent path[] = {nc};
     cust = CustomerManagement.CustomerHelper.narrow(ncRef.resolve(path));
  } catch (Exception e) {
    System.out.println(e);
  }
}

public void unsetEntityContext() {
  // Disconnect with the component wrapper if stateless
  cust = null;

  this.ctx = null;
}

public void ejbActivate() {
  // Connect with the component wrapper if stateful
}

public void ejbPassivate() {
  // Disconnect with the component wrapper if stateful
}
```

与包装器通信时，我们要相应处理CORBA相关异常和响应，可以将其转发到客户机，但只能转发相关异常。客户机不知道我们与CORBA包装器通信，因此不能向客户机转发任何CORBA相关异常，而要将其变成EJB相关异常，我们要像ejbCreate()方法中看到的一样抛出相应异常。

另一种方法是先处理异常，然后再将其转发到客户机。例如，如果不能调用CORBA包装器的方法，则可以先试试重新连接，然后再将错误通知客户机。

下面继续实现实体Bean，实现ejbLoad()与ejbStore()方法。我们用包装器导入和保存持久状态：

```
public void ejbLoad() {
  cd = new CustomerData();

  try {
     // Acquire the data from component wrapper
     StringHolder firstName = new StringHolder();
     StringHolder lastName = new StringHolder();
     StringHolder address = new StringHolder();
     ShortHolder numberOfPhoneAccounts = new ShortHolder();
     CustomerManagement.CustomerPackage.statusTypeHolder status =
               new CustomerManagement.CustomerPackage.statusTypeHolder();

     String cId = (String) ctx.getPrimaryKey();
     int ciId = (new Integer(cId)).intValue();

     cust.getCustomer (ciId, firstName, lastName,
                       address, numberOfPhoneAccounts, status);

     cd.setCustomerId(cId);
     cd.setName(lastName.value);
     cd.setAddress(address.value);
     cd.setNoOfPhoneAccounts(numberOfPhoneAccounts.value);
  } catch (Exception e) {
    System.out.println(e);
  }
}

public void ejbStore() {
  // Store the data in the component wrapper
  try {
     int r = cust.setCustomer(cd.getName(), "",
                    cd.getAddress(),
                    (short)cd.getNoOfPhoneAccounts(),
                    CustomerManagement.CustomerPackage.statusType.gold);
  } catch (Exception e) {
    System.out.println(e);
  }
}
```

然后实现主方法，首先是ejbCreate()方法：

```
// home methods
public String ejbCreate(String customerId, String name, String address,
                        int noOfPhAcc) throws CreateException {
  cd = new CustomerData();
  try {
     // Store to the bean
     cd.setCustomerId(customerId);
     cd.setName(name);
     cd.setAddress(address);
     cd.setNoOfPhoneAccounts(noOfPhAcc);

     // Create the corresponding record
     // in the existing system though wrapper
     int r = cust.setCustomer(cd.getName(), "",
                 cd.getAddress(),
                 (short)cd.getNoOfPhoneAccounts(),
                 CustomerManagement.CustomerPackage.statusType.gold);

     // Return the primary key
     return customerId;
  } catch (Exception e) {
    System.out.println(e);
    throw new CreateException();
  }
}

public void ejbPostCreate(String customerId, String name,
                                          String address, int noOfPhAcc) {
}

public String ejbCreate(CustomerData cd) throws CreateException {
  try {
     this.cd = new CustomerData();
     this.cd = cd;

     // Create the corresponding record
     // in the existing system though wrapper
     int r = cust.setCustomer(cd.getName(), "",
             cd.getAddress(),
             (short)cd.getNoOfPhoneAccounts(),
             CustomerManagement.CustomerPackage.statusType.gold);

     return cd.getCustomerId();
  } catch (Exception e) {
    System.out.println(e);
    throw new CreateException();
```

```
    }
  }

  public void ejbPostCreate(CustomerData cd) {
  }

  public void ejbRemove() throws RemoveException {
  }
```

最后要实现寻找方法。实现寻找方法**ejbFindByPrimaryKey()**很简单：

```
  public String ejbFindByPrimaryKey(String pk)
                                  throws ObjectNotFoundException {
    cd = new CustomerData();

    try {
      StringHolder firstName = new StringHolder();
      StringHolder lastName = new StringHolder();
      StringHolder address = new StringHolder();
      ShortHolder numberOfPhoneAccounts = new ShortHolder();
      CustomerManagement.CustomerPackage.statusTypeHolder status =
                new CustomerManagement.CustomerPackage.statusTypeHolder();

      int ciId = (new Integer(pk)).intValue();

      cust.getCustomer (ciId, firstName, lastName,
                              address, numberOfPhoneAccounts, status);

      cd.setCustomerId(pk);
      cd.setName(lastName.value);
      cd.setAddress(address.value);
      cd.setNoOfPhoneAccounts(numberOfPhoneAccounts.value);

      return pk;
    } catch (Exception e) {
      System.out.println(e);
      throw new ObjectNotFoundException();
    }
  }
```

实现**ejbFindByName()**则需要深思，因为当前组件没有提供合适的方法。这个方案可能对所有客户迭代，比较每个名称，但这要求多次远程方法调用，会对网络通信流造成不利影响，给**EJB**服务器和组件包装器带来多余负载。

更好的方案是修改组件包装器（假设能够访问代码和有权修改），增加按名称定位客户的方法。然后这种方法返回主键清单，这是实体**Bean**所需要的。这里不准备详细介绍，因为这很简单。

```
  public Collection ejbFindByName(String name) {
    Vector v = new Vector();
```

```
        // Delegate the search to the wrapper
        return v;
    }
}
```

这个实体Bean的部署描述项与上例中开发的BMP实体Bean的部署描述项相同。要按正确顺序建立实体Bean，首先要用一个IDL编译器从CustomerInt.idl，生成客户端映射，如Sun公司的 idlj。然后要编译源代码文件，包括生成访问CORBA包装器的接口和类。最后要把这些类和普通内容（部署描述项）一起包装到JAR文件中，在容器中部署。

我们还要启动CORBA包装器。首先要运行CORBA Naming Service（tnameserv），然后运行CORBA包装器服务器方对象。要测试实体Bean，我们使用与上例基本相同的简单客户机，主要差别是实体Bean的JNDI名称（eai3-CustomerHome）和使用bmp2包。

编译客户机源代码之后，运行下列命令，输出结果如图6.24所示。

图6.24

最后，我们看看如何实现Process Automator模式。

实现Process Automator模式

本节演示如何开发Process Automator组件，介绍两个例子。第一，我们开发Check Balance组件，是前面设计阶段标识的组件。对于Check Balance组件，我们用无状态会话Bean。然后我们介绍如何使用消息驱动Bean。

首先看看Check Balance会话Bean。要实现这个会话Bean，首先要定义会话Bean（Check-Balance）的组件接口，可以是本地或远程接口。我们用远程接口，因为Process Automator通常要远程访问。我们定义一个calculateBalance()方法。会话Bean的远程组件接口如下：

```
package session1;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface CheckBalance extends EJBObject {

  public double calculateBalance(int accountNo) throws RemoteException;

}
```

定义远程接口之后，要定义主接口。主接口提供寿命周期会话Bean方法。会话Bean不向客户机提供标识，因此主接口只有create()方法，这里只有单个create()方法。和组件接口

样，主接口也可以是本地或远程接口。远程主接口声明如下：

```
package session1;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface CheckBalanceHome extends EJBHome {

  public CheckBalance create() throws CreateException, RemoteException;

}
```

定义接口之后，就可以继续开发实现类。在实现类中，我们要提供组件与主接口声明的所有方法的实现，还要实现一些容器回调方法：**ejbActivate()**、**ejbPassivate()**、**setSessionContext()**、**ejbCreate()**与**ejbRemove()**。

会话**Bean**比实体**Bean**简单，因此需要更少**EJB**相关方法。我们声明**CheckBalanceBean**类，要实现**SessionBean**接口。实现类包括所有必要的方法之后仍然很简单。首先声明如下：

```
package session1;

import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.util.*;

public class CheckBalanceBean implements SessionBean {
```

会话**Bean**调用**MobilePhoneEvaluation**调节器，是前面在**RMI-IIOP**组件上开发的。注意实际情形中**Process Automator**要调用多个组件，这里显示一个简化方案，因为我们的目的是演示技术方面。

如果调用**RMI-IIOP**组件，则最好在创建会话**Bean**实例时连接组件，并储存连接。进行连接要求查找命名服务和缩小引用，这些操作的成本很高，从性能角度看不宜在每次方法调用之前建立连接。

最好把资源分成客户机依赖资源和非客户机依赖资源。对于无状态会话**Bean**，不存在这个问题，因为**Bean**本身是无状态的，因此不能存储任何客户机相关状态。如果需要存储客户机相关状态，则要使用状态会话**Bean**，见稍后介绍。

我们继续会话**Bean**实现类，声明专用属性。一个存储会话情境，一个连接**MobilePhoneEvaluation**组件。注意无状态会话**Bean**可以存储客户机无关状态，因此可以在**Bean**中存储与虚拟组件的连接。我们还声明几个**EJB**相关方法，在这个简单例子中不需要实现。

```
  private SessionContext ctx;
  private MobilePhoneEvaluation mpe;

  public void ejbActivate() {
  }

  public void ejbRemove() {
```

```
}

public void ejbPassivate() {
}

public void setSessionContext(SessionContext ctx) {
  this.ctx = ctx;
}
```

然后要实现主接口的创建方法。ejbCreate()方法在容器调用setSessionContext()之后调用，即容器中创建Bean实例之后。因此，ejbCreate()方法中可以建立与MobilePhoneEvaluation RMI-IIOP组件的连接。

```
// home
public void ejbCreate() throws CreateException {
  System.out.println("create: Connecting to the mediator");
  try {
    Properties h = new Properties();
    h.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.cosnaming.CNCtxFactory");
    h.put(Context.PROVIDER_URL, "iiop://localhost:900");
    Context inc = new InitialContext(h);
    Object a = inc.lookup("MobilePhoneEvaluation");
 mpe = (MobilePhoneEvaluation)
          PortableRemoteObject.narrow(
          inc.lookup("MobilePhoneEvaluation"),
          MobilePhoneEvaluation.class);
  } catch (Exception e) {
    System.out.println(e);
    throw new CreateException();
  }
}
```

对这个设置，我们不必修改前面开发的RMI-IIOP服务器。

还要修改RMI-IIOP服务器，用相同命名服务注册，还要管理异常。上例中，我们把任何连接RMI-IIOP服务器期间发生的异常作为创建异常进行转发。在生产系统中，则要建立更复杂的异常管理机制。

最后，我们要实现业务方法calculateBalance()。前面曾介绍过，这个方法通常调用多个现有系统。这里，为了简单起见，我们只调用一个RMI-IIOP组件，以突出技术问题。

```
// business method
public double calculateBalance(int accountNo) {
  double result = 0;
  System.out.println("calculateBalance: invoking remote " +
                     "method on the mediator");
  try {
```

```
            result = mpe.evaluateConsumption(
                                (new GregorianCalendar(2,4,1)).getTime(),
                                (new GregorianCalendar(2,4,1)).getTime());

          } catch (Exception e) {
        System.out.println(e);
          }

          return result;
        }
      }
```

这样就完成了无状态会话Bean的实现。下面要定义部署描述项和把例子建成JAR文件，然后进行部署和测试。无状态会话Bean的部署描述项（ejb-jar.xml）比较简单，放在本章的代码中。

最后，可以编译源文件，把所有需要的文件集成到JAR文件中。除了通常文件之外，还要包括RMI-IIOP接口 MobilePhoneEvaluation（MobilePhoneEvaluation.class）和必要的残根（_MobilePhoneEvaluation_Stub.class）。可以用前面建立的残根，也可以用rmic再次建一个残根。我们要把会话Bean部署到应用程序服务器中，要根据所选应用程序服务器的步骤进行部署。

客户机

要测试会话Bean，我们编写一个简单客户机，通过远程接口测试其功能。下列客户机代码假设我们用名称eaiSS-CheckBalanceHome向JNDI注册会话Bean：

```
package session1;

import java.rmi.*;
import javax.rmi.*;
import java.util.*;

import javax.ejb.*;
import javax.naming.*;

public class Client {
  public Client() {
  }

  public static void main(String[] args) {
    System.out.println("EAI process automator client");
    System.out.println();

    // Create a new instance and invoke the example() method
    try {
      Client client = new Client();
      client.example();
    } catch (Exception e) {
```

```
            System.out.println(e);
        }
    }

    public void example() {

        try {
            // Obtain a reference to the initial naming context
            Context ctx = new InitialContext();

            // Resolve the JNDI home name and narrow
            CheckBalanceHome home = (CheckBalanceHome)PortableRemoteObject.narrow
                    ((CheckBalanceHome)ctx.lookup("eaiSS-CheckBalanceHome"),
                                                    CheckBalanceHome.class);

            // Create a new session bean instance
            CheckBalance cb = home.create();

            // Invoke the method
            System.out.println("The balance of account no. 1 is: "
                                            +cb.calculateBalance(1));

            // Remove the session bean instance
            cb.remove();

        } catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

运行例子

要运行这个会话Bean，采用下列步骤：

- 向应用程序服务器部署Bean。
- 运行命名服务tnameserv。
- 运行RMI-IIOP服务器方对象——本章前面开发的MobilePhoneEvaluation调节器组件。
- 最后，可以用图6.25所示的命令运行EJB客户机。

图6.25

对Process Automators使用状态会话Bean

前面曾介绍过，如果需要在客户机间进行不同方法调用储存相关状态，则要使用状态会话Bean。用状态会话Bean作为Process Automators与使用无状态会话Bean相似，主要差别在于实现类中要声明公开属性，让状态会话Bean保存会话状态。

容器会保留这些公开属性的状态，并临时保存，必要时复用相同Bean实例。容器用序列化存储状态，因此前面曾经指出，维护会话状态的所有属性都应可序列化。

此外，状态会话Bean可以激活和钝化，即要像实体Bean本章划分Bean使用的资源。客户机依赖的资源应在ejbActivate()方法中取得，在ejbPassivate()方法中释放，而非客户机依赖的资源应在ejbCreate()方法中取得，在ejbRemove()方法中释放。

Process Automators使用消息驱动Bean

会话Bean适合Process Automators，因为Process Automators需要与远程客户机同步RMI-IIOP通信，即客户机调用会话Bean方法时要等待响应。这个同步性质有几个结果。由于客户机受阻，因此最好能较快地运行操作。对有些操作，这是不可能的，或要求EJB层太多资源。在许多信息系统中，都会遇到峰值使用时间，具有大量同时访问的客户机。

解决的办法是用消息驱动Bean（MDB）通过JMS异步通信。使用异步通信时，可以存储请求，在非峰值时间处理。但事实上我们无将结果返回客户机，因此消息驱动Bean适合更新操作并且不需要将结果返回客户机，或用其他机制报告结果。

这里演示如何用消息驱动Bean开发简单Process Automators。我们开发的DisableAccountsBean消息驱动Bean搜索三个月以上零结余的账户并将其关闭。搜索与关闭账户是用现有账户结余管理应用程序完成的，这个任务通常需要大量时间，不需要将结果返回立即开始操作的客户机，因此适合用消息驱动Bean。

消息驱动Bean要实现两个接口：MessageDrivenBean与MessageListener。第一个接口要求实现EJB容器回调方法（如setMessageDrivenContext()与ejbRemove()），实现容器实例池。第二个接口要求实现onMessage()方法。消息驱动Bean用这个方法取得请求。具体地说，消息驱动Bean通过onMessage()方法取得消息，然后分析和响应这个消息。

可以看出消息驱动Bean是松耦合的，可以用onMessage()方法响应不同消息，不需要手工进行编译时类型检查和消息分析。

因此，只要用实现类实现消息驱动Bean。本例中我们定义DisableAccountsBean实现类，实现上述接口。在Bean中，我们实现onMessage()方法，搜索的账户和将其关闭。

```
package msg;

import javax.ejb.*;
import javax.jms.*;

public class DisableAccountsBean implements MessageDrivenBean,
      MessageListener {

  private MessageDrivenContext mctx;

  public void ejbRemove() {
```

```
    }

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
      mctx = ctx;
    }

    public void ejbCreate () throws CreateException {
    }

    // Implementation of MessageListener

    public void onMessage(Message msg) {
      try {
        // Extract the message
        TextMessage tm = (TextMessage) msg;
        String text = tm.getText();
        System.out.println("Received request: " + text);

        // Process the request
        System.out.println("Processing the request: disabling accounts...");

        // Contact the existing application
        // Search for accounts and disable them
      }
      catch(JMSException ex) {
        ex.printStackTrace();
      }
    }
}
```

定义实现类之后，需要编写部署描述项，放在本章源代码中。然后要编译源文件和准备包括DisableAccountsBean.class和部署描述项的JAR文件。然后要把JAR文件部署到应用程序服务器上。但是，我们还要创建名称为disableAccounts的JMS Topic。

客户机

消息驱动Bean的客户机可以是任何客户机，将消息发送到某个主题（或队列，如果用Queue而不用Topic）。这可以是使用JMS的Java客户机，也可以是使用兼容MOM的现有应用程序。因此，消息驱动Bean适合处理现有应用程序的请求。

这里显示在Java中简单的JMS客户机，创建主题连接并启动，创建新的主题会话，创建主题发表者，创建文本消息并发表。

```
package msg;

import java.rmi.RemoteException;
import java.util.Properties;
import javax.jms.*;
import javax.naming.*;
import javax.ejb.CreateException;
```

```
import javax.ejb.RemoveException;
import javax.rmi.PortableRemoteObject;

public class Client {
  // The name of the topic the client and the message bean use
  static private String TOPIC_NAME = "disableAccounts";

  public Client() {
  }

  public static void main(String[] args) {
    System.out.println("EAI message client");
    System.out.println();

    try {
      Client client = new Client();
      client.example();
    } catch (Exception e) {
      System.out.println(e);
    }
  }

  public void example() {
    try {
      // Create a message context
      Context mctx = new InitialContext();

      // Create the topic connection and start it
      TopicConnectionFactory cf = (TopicConnectionFactory)
        mctx.lookup("javax.jms.TopicConnectionFactory");
      TopicConnection mTopicConn = cf.createTopicConnection();
      mTopicConn.start();

      Topic newTopic = null;
      TopicSession session = null;

      try {
        // Create a new topic session
        session = mTopicConn.createTopicSession(
                              false,   // non transacted
                              Session.AUTO_ACKNOWLEDGE);

        // Lookup the topic name (if exist, otherwise exception)
        newTopic = (Topic) mctx.lookup(TOPIC_NAME);

      } catch(NamingException ex) {

        // If the topic doesn't exist create it
        newTopic = session.createTopic(TOPIC_NAME);
        mctx.bind(TOPIC_NAME, newTopic);
      }
```

```
            // Create a topic publisher
            TopicPublisher sender = session.createPublisher(newTopic);

            // Create a text message
            TextMessage tm = session.createTextMessage();
            tm.setText('EAI-disableAccounts');

            // Publish the message
            sender.publish(tm);

          }catch(Exception ex) {
            ex.printStackTrace();
          }
        }
      }
```

要运行客户机，就要先编译。运行时要指定初始命名工厂和命名提供者URL（和上例一样）。注意客户机不输出任何结果，因为它请求的操作还没有完成。但客户机将控制返回操作系统（客户机在操作期间不受阻止）。我们要看看应用程序服务器窗口，看看消息驱动Bean如何实际工作，如图6.26所示

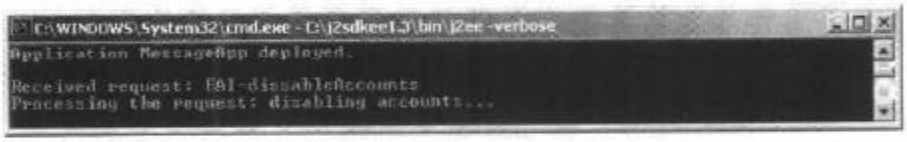

图6.26

这个演示完成了集成模式的实现，下节介绍在B2B集成中使用这些模式。

在B2B集成中使用集成模式

上述集成模式适用于EAI内和EAI间与B2B集成。我们的原则是：EAI间与B2B集成只能建立在定义合理的EAI内集成之上。因此，我们说，上述集成模式一步一步实现了EAI内集成，然后要在集成体系结构上建立B2B交互。在这个情境中，上述所有集成模式都很重要，但Process Automator是其中最值得注意的。通常，我们用这个模式实现业务过程，随交互类型不同而不同。

小结

本章介绍了企业集成的模式。首先简要介绍了企业应用程序集成（EAI）和最重要的阶段：数据级集成、应用程序接口集成、业务方法表示和EAI间与B2B集成，然后介绍了EAI与J2EE，可以看出J2EE支持几个开放标准技术与协议，可以集成现有应用程序和开发Web服务。

然后我们介绍了最重要的集成模式，包括：

- Integration Broker模式
- Integration Wrapper模式
- Integration Mediator模式
- Virtual Component模式
- Data Mapping模式
- Data Access Object（DAO）模式
- Process Automator模式

其中只有DAO模式是已知的，已经在J2EE设计模式中介绍，因此我们介绍了其他集成模式，介绍了选择哪个集成模式和如何将其运用到总体集成体系结构中。我们用移动电话公司的例子显示了几个用例的模式。

最后，我们介绍如何在J2EE中采用这些模式；介绍如何选择相应技术和实际实现；介绍如何用CORBA实现Integration Wrappers，用RMI-IIOP实现Integration Mediators；如何用实体Bean实现Data Mapping模式，用各种EJB实现Virtual Components与Process Automators。

第7章 复用性、可维护性与扩展性设计模式

本章介绍实现复用性、可维护性与扩展性的设计模式，然后介绍在一般性框架中演示这些方面的模式，包括J2EE框架情境中的一系列组件。这里要介绍复用性、可维护性与扩展性设计模式。本章介绍的模式包括：

- Façade模式
- Abstract Factory模式
- Decorator模式
- Template Method模式
- Builder模式

现代软件又大又复杂，因此要利用面向对象分析方法（OOA）和面向对象设计方法。采用面向对象方法的三个好处是复用性、可维护性和扩展性，这些思想可以在类、组件、服务、模式等不同层次采用。

- 可以设计一个或几个类来解决特定任务。典型例子是Debug类之类的实用程序类，或一组类配合使用。通常，这些类包装起来，一起部署。
- 组件是可复用软件模型，发表和注册接口。组件可以在自己的环境中重新部署（如Enterprise JavaBean之类的J2EE组件）。
- 服务是复用框架和合约接口。服务提供者可以发表服务，对该服务感兴趣的客户机注册表示兴趣；例如，在应用程序服务器环境中，可以向域中的客户机组件提供事务管理、安全提供、数据库池等优先服务。
- 模式描述通信对象与类，通过定制解决特定情境中的一般性问题。有些模式促进去耦合、闭合和复用与扩展代码。理论上，复用模式使类、组件和服务更容易实现复用性、可维护性和扩展性。

高质量的面向对象软件应具有下列特征：

- 包装好
- 模块化
- 松耦合
- 通用化，适用于不同情境

为何编写可复用软件

软件复用性在商业世界中没有达到潜在期望值，在GUI工具层产品中不如业务组件产品中那么成功。

软件复用在推动“软件思维”方面起着重要作用。创造性公司开始一个新方向，填补市场空白，例如应用程序服务器市场。成功的应用程序服务器产品可以复用与扩展，支持每

天不同公司为不同业务服务开发的几百个不同产品。

另一个例子是JDK本身。Java从早期开始就以其丰富的库与包著称。基本Java语言类在不同Java编程人员的应用程序中一遍遍复用。例如Hashtable类就是可复用类的范例。这个类接受Object类型的参数和返回值。Java中实现的任何类都可以复用这个类。这个类还与其他类整洁地分离，很容易了解如何使用它的方法，而不需要深入了解它的实现。

最近还有Log4J与Ant之类成功的小故事，因此，编写良好的面向对象可复用的软件是Java软件开发的重要部分，是它的业务目标和卖点。

软件复用性的好处包括：

- **Time-to-market（交货期）**
  复用现有代码比从头开始编写应用程序要快。
- **Quality（质量）**
  复用经过全面测试的代码可以减少缺陷，提高软件质量。
- **Ease of learning（易学习）**
  使用相同代码可以减少开发人员的学习难度，保证把时间用在最需要的地方（如处理新问题）。

为何编写可维护软件

在软件世界中，可维护性保证软件或系统在寿命周期中能继续正确工作。

软件需要维护，否则可能造成致命后果。编写软件的成本很高，但开发成本只占软件寿命周期中总成本的20%，而80%是维护成本。

软件产品随一个项目开发小组人员的增加而变得更加复杂。在项目过程中，工作人员来来去去，除非认真关注，否则维护软件的时间与难度会大大增加。为了便于维护，应采用两个简单原则：

- **Structure（结构）**
  每一层都要采用明确定义和易于阅读的结构。应用程序应能方便地标出处理不同方面的段，如GUI和数据处理。这样能更方便地在几千行代码中找出所要的代码。方法应尽量小，否则很容易让人迷失方向。
- **Legibility（合法性）**
  每一行代码应格式合理，便于阅读。说明语句应简明扼要。有时采用了一个小窍门，几个月后自己都不知道是怎么干的。

为何编写可扩展软件

由于开发软件产品要投入很大的成本，因此应尽量延长产品的使用寿命。但是，我们生活在一个不断改变的世界中，对软件产品的要求也在不断改变。

扩展软件功能和增加新功能可以延长产品的使用寿命，这比开发新产品的成本要低得多。模块化和松耦合能保证只改变需要的部分，将调整工作减到最少。

下节介绍一般性案例的设计和开发，演示上述三大目标。

基于组件的案例

假设公司要求开发一组可扩展和可复用组件，减少每个项目的开发工作。这些组件可以在项目之间内部共享。

系统与专属数据源交互，使用专属查询语言。要开发的组件支持在J2EE情境中构造和建立这些查询。

要开发的基本组件如下：

- 一组可扩展数值对象类，可以传输每个查询的内容参数。
- 一组基本查询类，可以扩展成大查询系列。
- 一组工厂与建立类，创建和构造数值对象与查询对象。

接口

要开发的组件的重要方面是接口，客户机类用接口访问我们建立的服务。这个接口应易于使用，易于理解，还要统一和单一，应减少学习难度。然后用组件进行开发时，就可以集中开发应用程序的业务逻辑。

可以用几个模式隐藏子系统细节和分离客户机类：

- **Façade**——对子系统中的一组类和接口提供高级抽象，使其更容易使用。
- **Session Façade**——是个企业Bean，隐藏表示层与业务逻辑层交互的细节，对J2EE框架中的业务对象提供粗粒访问。
- **Business Delegate**——分离客户机层与业务层，隐藏J2EE情境中EJB查找之类的基础业务服务。

这三个模式是相似的，但用不同方法达到稍有不同的目的。Session Façade与Business Delegate都是针对J2EE情境的Façade模式特例。Session Façade是个企业Bean，提供粗粒访问，而后者则隐藏业务层的基础业务服务。Façade设计模式更一般化，因此适用于不同情形，更适合我们的案例。

Façade设计模式

Façade设计模式对子系统中的一组接口提供更高水平的接口或高级抽象，使其更容易使用。

使用这个模式的主要动机是减少向客户机类提供的细节，减少客户机处理的类的个数。Façade模式有两大优点。第一，不让客户机了解子系统的内部细节，使其更容易使用。第二，促进客户机类与子系统类之间的低耦合，使其更容易改变或扩展子系统内部功能，而不会影响使用子系统的客户机代码。图7.1显示了Façade设计模式的结构。

Façade设计模式的参与者如下：

- **Façade（门户）**——子系统中一组接口与类的高级接口。
- **Subsystem classes（子系统类）**——子系统内部的类和接口。

- **Client classes（客户机类）**——子系统外部的类，需要子系统服务。

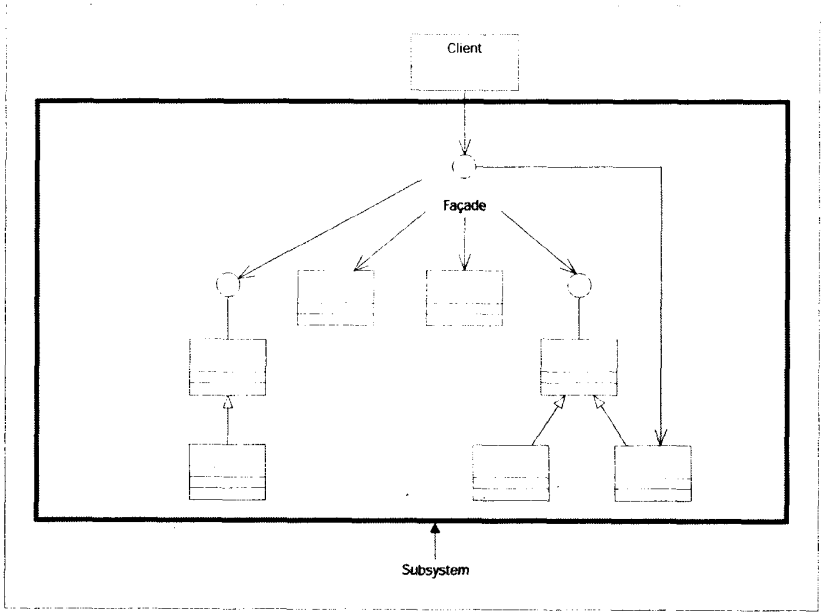

图7.1

在J2EE情境中实现Façade

在J2EE情境中，Façade可以作为其他J2EE模式的支持模式，隐藏子系统的一些内部细节。

与C++不同的是，Java定义公开类和包类。公开类提供给所在包以外的包，而包类只在所在包可见。这个特性可以实现Façade模式，只显示一些实现类包。执行这个规则可以改变类实现而不影响包范围之外的类。

J2EE系统中一个常见的问题是向客户机类提供数值对象。本章开发了样本Façade接口及其实现方法，并且在这个情境中显示Façade模式。此外，我们定义了ValueObject接口及其实现。本章的代码演示了介绍的思想及这些思想如何实现。

Façade接口的样本代码如下：

```
package reuse;

/* Represents a facade interface to this package */

public interface Facade {

  /* Creates and returns a value object specific to this package
   * @return ValueObject specific to this package
   */
  public ValueObject getValueObject();

  //other facade methods

}
```

这个阶段的实现很简单：

```
package reuse;

/* Represents a facade implementation */

public class FacadeImpl implements Facade {

  /* Creates and returns a value object specific to this package
   * @return ValueObject specific to this package
   */
  public ValueObject getValueObject() {
    return new ValueObjectImpl();
  }

  //other facade methods

}
```

对**ValueObject**接口，我们用动态属性设置策略和弱类型属性值，这比基于**bean getX()**与**setX()**方法的典型数值对象思想更灵活、更可复用，但是弱类型的，因此需要更多验证工作。

ValueObject接口的源代码如下：

```
package reuse;

import java.io.Serializable;
import java.util.Iterator;
/* A set of dynamic properties expressed as { name, value } pairs.
 * Typically, client code is expected to use the Facade interface
 * to retrieve the correct ValueObject implementation.
 */
public interface ValueObject extends Serializable {

   /* Returns the names of the properties stored in the name space.
    * @return java.util.Iterator
    */
   public Iterator getPropertyNames();

   /* Sets a property to a given value. Validation may be performed for
    * any given value, depending on the implementation being used.
    * @param name  Name of the property to set.
    * @param value New value for the property.
    * @throws ValidationException if the value is incorrect.
    * @throws com.woe.cmi.exception.ValidationException
    */
   public void setProperty(String name, Object value);

   /* Returns the value of a property.
    * @param name Name of the property to retrieve.
```

```
     * @return java.lang.Object representing the value, or null if no
     * property has been set with this name.
     */
    public Object getProperty(String name);
}
```

要增加新属性或扩展现有属性集，不需要改变这个接口，只要修改其实现，增加一些检验规则即可。

ValueObjectImpl类只有包显示性，因此很容易扩展或替换，这个实现用**Hashmap**保存属性集：

```
package reuse;

import java.util.Map;
import java.util.Iterator;
import java.util.HashMap;

/* Provides a ValueObject implementation. */

class ValueObjectImpl implements ValueObject {
   private Map nameMap = new HashMap (10);

   /* Returns the names of the properties stored in this value object
    * @return java.util.Iterator
    */
  public Iterator getPropertyNames() {
    return nameMap.keySet().iterator();
  }

   /* Sets a property to a given value.
    * @param name  Name of the property to set.
    * @param value New value for the property.
    */
  public void setProperty(java.lang.String name, java.lang.Object value) {
    nameMap.put(name, value);
  }

   /* Returns the value of a property.
    * @param name Name of the property to retrieve.
    * @return java.lang.Object representing the value, or null if no
    * property has been set with this name.
    */
  public Object getProperty(java.lang.String name) {
    return nameMap.get(name);
  }

  /* returns a string representation of myself */
  public String toString() {
    StringBuffer buf = new StringBuffer( this.getClass().getName() + "::\n"
```

```
        );
        for (Iterator it = getPropertyNames(); it.hasNext(); ) {
          String name = it.next().toString();
          Object value = getProperty(name);
          buf.append(name + "=" + value);
          if( it.hasNext() ) {
            buf.append( ", " );
          }
        }
        return buf.toString();
      }
    }
```

图7.2显示了前面开发的接口和类及其关系。

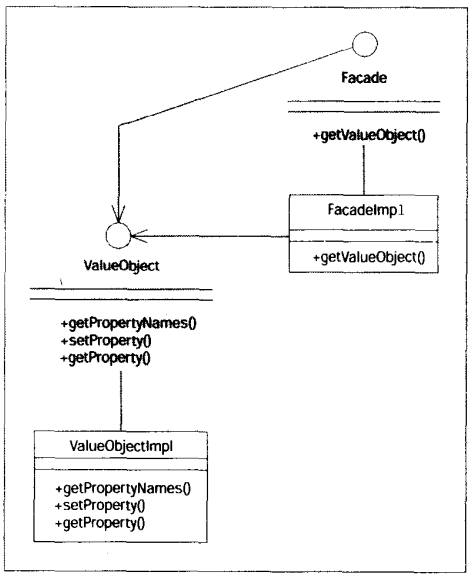

图7.2

也许你已经注意到，Façade还不太够，我们还要一个模式来创建不同的数值对象实现来引入工厂，可以保证能够扩展现有数值对象实现，还定义了简单明了的框架，可以根据今后改变进行调整。

今后的不同项目可能有几个不同数值对象实现。为了管理数值对象系列的创建，可以用工厂模式。

可以用两个工厂模式创建ValueObject类：

- **Factory Method（工厂方法）**——定义创建对象的接口，但将实例化延迟到子类中。
- **Abstract Factory（抽象工厂）**——定义创建相关对象系列的接口。

Factory Method模式可以连接两个相关对象系列的层次，用工厂方法创建。Abstract Factory模式通过引入工厂接口（见下节介绍）创建相关对象系列。Factory Method模式使用创建器，不一定是工厂接口，还可以包括其他函数。Abstract Factory总是用专门创建对象的工厂接口。为此，我们认为这个模式最适合达到我们的目的。

Abstract Factory设计模式

Abstract Factory设计模式可以创建相关对象系列而不声明具体类；可以控制应用程序中类的创建，让客户机通过抽象接口使用类。因此，客户机代码不必知道任何具体类。扩展创建的类系列也很容易管理，因为客户机代码不知道使用的实际具体类。图7.3显示了Abstract Factory设计模式的结构。

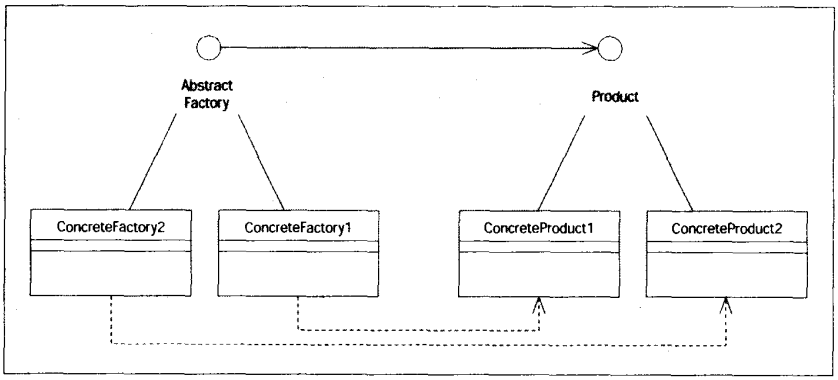

图7.3

Abstract Factory设计模式的参与者如下：

- **Abstract Factory（抽象工厂）**——提供创建产品的接口抽象
- **Product（产品）**——提供产品抽象的接口
- **Concrete Factory（具体工厂）**——创建具体产品的抽象工厂实现
- **Concrete Product（具体产品）**——产品具体实现

在J2EE情境中实现Abstract Factory

我们用两种方法在Java中实现Abstract Factory，一种是标准方法，基于工厂类定义中命名产品。

本例要建立两个产品，是ValueObjectA与ValueObjectB，这些类的代码见本章的代码下载。

ValueObjectFactory接口如下：

```
package reuse;

/* Value object factory interface */
interface ValueObjectFactory {

  /* Creates a product called ValueObjectA
```

```
     * @return the product created
     */
    public ValueObject createValueObjectA();

    /* Creates a product called ValueObjectB
     * @return the product created
     */
    public ValueObject createValueObjectB();

}
```

这个类实现如下：

```
/* Value object factory implementation */

class ValueObjectFactoryImpl implements ValueObjectFactory {

    /* Creates a product called ValueObjectA
     * @return the product created
     */
    public ValueObject createValueObjectA() {
      return new ValueObjectA();
    }

    /* Creates a product called ValueObjectB
     * @return the product created
     */
    public ValueObject createValueObjectB() {
      return new ValueObjectB();
    }

}
```

修改后的类框图如图7.4所示，包括开发的新工厂类。

第二种方法是较强类型的，基于命名一个参数来标识产品，从而允许创建这个产品。

这个方法更灵活，因为增加新产品并不要求改变代码。为此，要改变**ValueObjectFactory**接口及其实现。

```
package reuse;

/* Value object factory interface */

interface ValueObjectFactory {

    /* Creates a product called based on the name specified
     * @param name of product to create
     * @return the product created
     */
    public ValueObject createValueObject( String name )
                                throws CreateException;

}
```

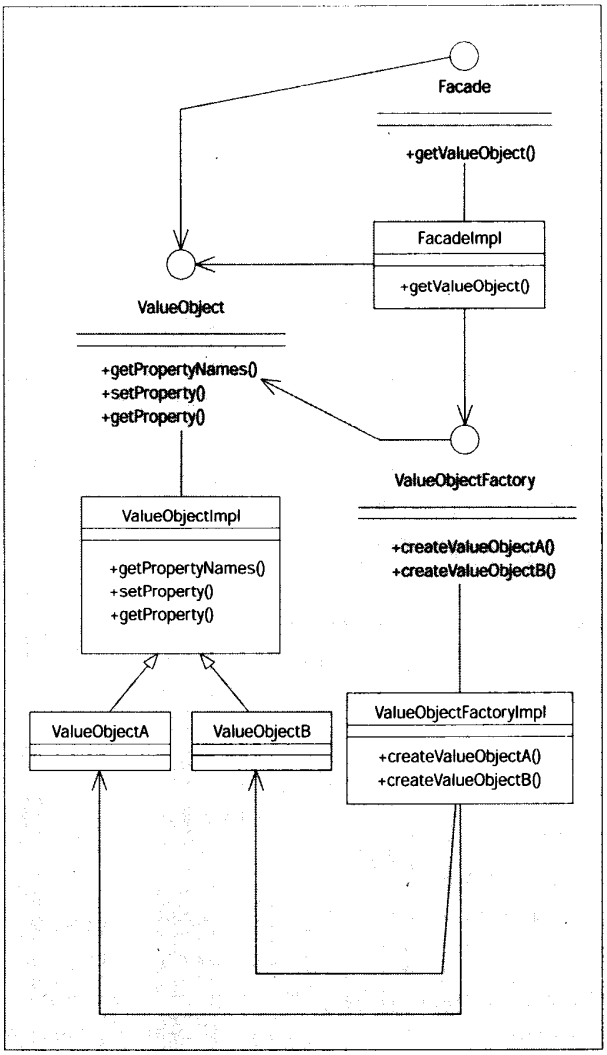

图7.4

上述接口的实现如下：

```
package reuse;

import java.util.*;
import java.lang.reflect.*;

/* Value object factory implemenation */

class ValueObjectFactoryImpl implements ValueObjectFactory {

  private Properties properties = new Properties();
  private final static String VALUE_OBJECT_MAPPING_PROPERTIES =
```

```
                                "valueObjectMapping.properties";

// Creates instance of me
ValueObjectFactoryImpl() {
  try {
    properties.load( getClass().getResourceAsStream(
    VALUE_OBJECT_MAPPING_PROPERTIES ));
  } catch( java.io.IOException e ) {
    // handle exception here
  }
}

/* Creates a product called based on the name specified
 * @param name of product to create
 * @return the product created
 */
public ValueObject createValueObject( String name )
    throws CreateException {

  try {
    return ( ValueObject )ObjectCreator.createObject(
    properties.getProperty( name ) );
  } catch( Exception e ) {
    throw new CreateException( "Error occured while creating " +
                               name, e);
  }
}

static class ObjectCreator {
 // Private constructor. This class cannot be instantiated
    private ObjectCreator() {}

    /* Instantiate an Object from a given class name
     * @param className full qualified name of the class
     * @return the instantaited Object
     * @exception java.lang.Exception if instantiation failed
     */
    public static Object createObject(String className)
        throws Exception {
      return createObject( Class.forName(className));
    }

    /* Instantiate an Object instance
     * @param classObject Class object representing the object type
     * to be instantiated
     * @return the instantaied Object
     * @exception java.lang.Exception if instantiation failed
     */
```

```
public static Object createObject(Class classObject)
    throws Exception {
  Object object = null;
  return  classObject.newInstance();
}

/* Instantiate an Object instance, requires a constructor with
 * parameters
 * @param className full qualified name of the class
 * @param params an array including the required parameters to
 * instantiate the object
 * @return the instantaited Object
 * @exception java.lang.Exception if instantiation failed
 */
public static Object createObject(String className, Object[]
    params) throws Exception {
  return createObject( Class.forName(className), params);
}

/* Instantiate an Object instance, requires a constructor with
 * parameters
 * @param classObject, Class object representing the object type
 * to be instantiated
 * @param params an array including the required parameters to
 * instantiate the object
 * @return the instantaied Object
 * @exception java.lang.Exception if instantiation failed
 */
public static Object createObject(Class classObject, Object[]
    params) throws Exception {
  Constructor[] constructors = classObject.getConstructors();
  Object object = null;
  for(int counter = 0; counter<constructors.length; counter++) {
    try {
      object = constructors[counter].newInstance(params);
    } catch(Exception e) {
      if(e instanceof InvocationTargetException)
          ((InvocationTargetException)e).getTargetException().
              printStackTrace();
        // Do nothing, try the next constructor
      }
  }
  if(object == null)
    throw new InstantiationException();
  return object;
```

```
        }
    }
}
```

这个类从属性文件读取数值对象的属性名与类名映射，动态实例化这些类。这个方法不如前面介绍的方法安全，但更加灵活。

属性文件内容如下：

```
ValueObjectA=reuse.ValueObjectA
ValueObjectB=reuse.ValueObjectB
```

最后，需要补充Facade接口及其实现，使其可以创建不同类型的数值对象的产品。

```
package reuse;

/** Represets a facade interface to this package
*/

public interface Facade {

  /** Creates and returns a value object specific to this package
* @return ValueObject specific to this package
*/
  public ValueObject getValueObject( String name ) throws CreateException;

  //other facade methods

}
```

总之，这个包中使用的工厂接口和类只能在这个包中访问，这个包中目前惟一公开类是Façade接口、其实现、ValueObject接口和包中使用的异常类。这样就使扩展与维护更加方便，因为减少了客户机与实现类之间的耦合。

修改后的类框图如图7.5所示，显示了我们开发的Abstract Factory的第二个策略。

前面曾介绍过，前面开发的数值对象的类是灵活和弱类型的。这样就要用检验函数来检验数值对象的内容。在维护灵活性的同时在类中增加检验功能的一个好办法是使用Decorator模式。

要引入检验，我们可以用两种方法：

- **Using sub-classing（使用子类）**
  用子类增加责任
- **Using Decorator（使用Decorator）**
  不用子类增加责任

在Decorator中可以透明增加新行为或责任而不影响其他对象。这在增加的行为需要放弃或无法采用子类时特别有用。由于Java是单重继承的，不能通过继承增加不同行为。

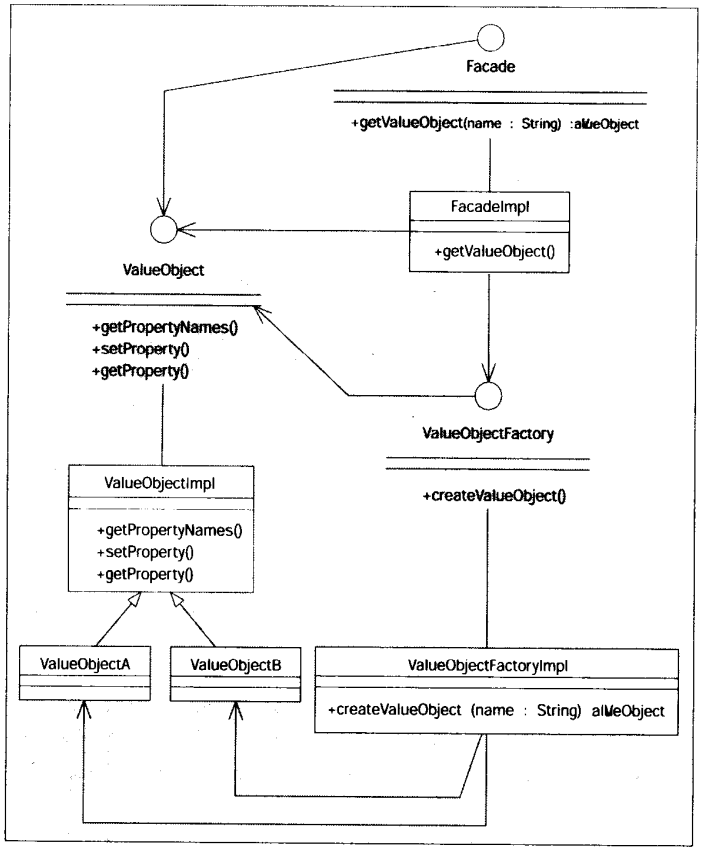

图7.5

Decorator设计模式

Decorator设计模式不用子类而在另一类中动态增加行为，提供了不用子类而扩展类的更灵活方法。这个模式可以代替子类。图7.6显示了Decorator设计模式的结构。

Decorator设计模式的参与者如下：

- **Component（组件）**——定义组件接口，可以装饰
- **Concrete Component（具体组件）**——要求增加行为或责任
- **Decorator（装饰）**——不用子类而在具体组件中动态增加行为

在J2EE情境中实现Decorator

为此而把ValueObject接口稍作修改，在setProperty()方法中抛出ValidateException。我们可以让ValueObject保持不变，让ValidateException扩展运行异常，但更好的方法是声明异常。如果这样引入太多代码改变，则最好让ValidateException成为运行异常，如图7.6所示。

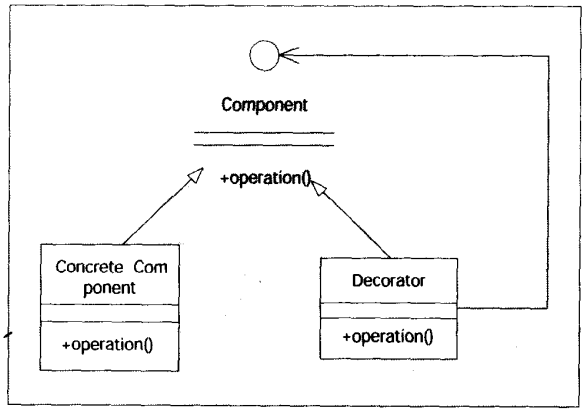

图7.6

改变的接口如下：

```
package reuse;

import java.io.Serializable;
import java.util.Iterator;

/* A set of dynamic properties expressed as { name, value } pairs.
 * Typically, client code is expected to use the Facade interface
 * to retrieve the correct ValueObject implementation.
 */
public interface ValueObject extends Serializable {

   /* Returns the names of the properties stored in the name space.
    * @return java.util.Iterator
    */
   public Iterator getPropertyNames();

   /* Sets a property to a given value. Validation may be performed for
    * any given value, depending on the implementation being used.
    * @param name  Name of the property to set.
    * @param value New value for the property.
    * @throws ValidationException if the value is incorrect.
    * @throws com.woe.cmi.exception.ValidationException
    */
   public void setProperty(String name, Object value)
       throws ValidateException;

   /* Returns the value of a property.
    * @param name Name of the property to retrieve.
    * @return java.lang.Object representing the value, or null if no
      property has
```

```
     * been set with this name.
     */
    public Object getProperty(String name);
}
```

数值对象装饰类用代理而不用继承进行装饰，这是更灵活的策略，因为可以用这个装饰类装饰Valueobject接口的多个实现。这是Decoratedvalueobject类的源代码：

```
package reuse;

import java.util.Map;
import java.util.Iterator;
import java.util.HashMap;

/* Provides a decorated ValueObject implementation with validation. */
class DecoratedValueObjectImpl implements ValueObject {

   private ValueObject valueObject = null;

   /* Creates instance of me */
   DecoratedValueObjectImpl( ValueObject valueObject ) {
     this.valueObject = valueObject;
   }

   /* Returns the names of the properties stored in this value object
    * @return java.util.Iterator
    */
   public Iterator getPropertyNames() {
     return valueObject.getPropertyNames();
   }

   /* Sets a property to a given value.
    * @param name  Name of the property to set.
    * @param value New value for the property.
    */
   public void setProperty(java.lang.String name, java.lang.Object value)
       throws ValidateException {
     if( name == null || value == null ) {
       throw new ValidateException("Name or value being set is null");
     } else if ( name != null && name.equals( "" ) ) {
       throw new ValidateException("Name being set is empty value" );
     } else {
       valueObject.setProperty(name, value);
     }
   }

   /* Returns the value of a property.
    * @param name Name of the property to retrieve.
    * @return java.lang.Object representing the value, or null if no
```

```
     * property has been set with this name.
     */
    public Object getProperty( java.lang.String name ) {
      return valueObject.getProperty( name );
    }

    /* Returns a string representation of myself */
    public String toString() {
      return valueObject.toString();
    }

}
```

要使用Decorator模式，我们需要改变ValueObjectFactoryImpl中的方法，这是createValueObject()方法，目前代码如下：

```
public ValueObject createValueObject(String name) throws CreateException {
  try {
    return ( ValueObject )ObjectCreator.createObject(
              properties.getProperty( name ) );
  } catch( Exception e ) {
    throw new CreateException(
                "Error occurred while creating " + name, e );
  }
}
```

修改之后，createValueObject()方法变成如下：

```
public ValueObject createValueObject(String name) throws CreateException {
  try {
    return new DecoratedValueObjectImpl(
                ( ValueObject )ObjectCreator.createObject(
                                properties.getProperty( name ) ) );
  } catch( Exception e ) {
    throw new CreateException(
                "Error occurred while creating " + name, e );
  }
}
```

注意由于类的所有数值对象系列和Value Object Factory实现类只能在包中访问，因此即使在产品成熟之后，增加这些修改也是完全安全的。

图7.7显示了DecoratedValueObjectImpl在其余Value Object类系列中的地位。

最后，为了使这些类集完整，应包括ValidateException class的源代码。ValueObject接口用这个类检验设置的属性：

```
package reuse;

import java.io.PrintStream;
```

```
import java.io.PrintWriter;

/* Indicates that a validation exceptional condition has occurred */

public class ValidateException extends BaseException {

  /*Creates an instance of me */
  public ValidateException() {
    super();
  }

  /*Creates an instance of me */
  public ValidateException(String msg) {
    super(msg);
  }

  /*Creates an instance of me */
  public ValidateException(String msg, Throwable cause) {
    super(msg, cause);
  }

  /*Creates an instance of me */
  public ValidateException(Throwable cause) {
    super(cause);
  }

}
```

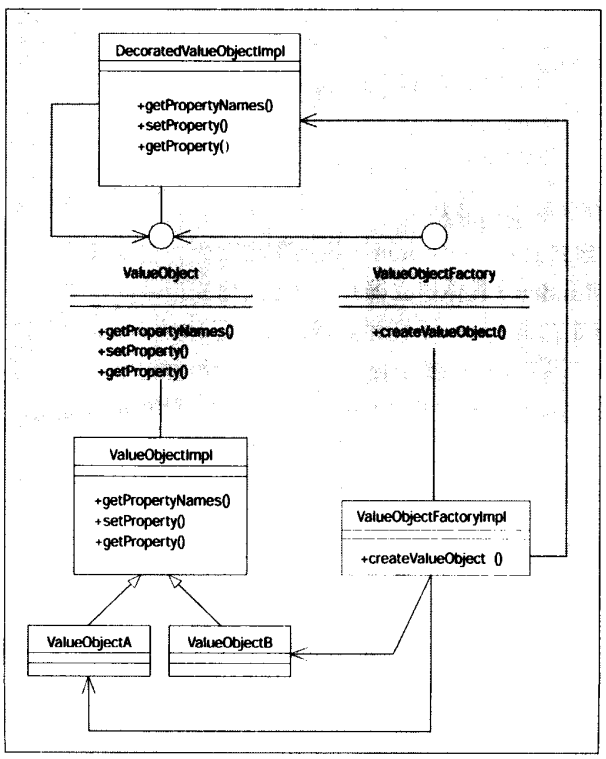

图7.7

这个类扩展BaseException类。注意这个异常引入了嵌套异常。增加的特性使异常类变得通用化，在不同情形中更易复用。Our Façade接口提供了更多功能，而不只是创建数值对象，用于其他J2EE包中。

下面要开发和设计的组件是查询类系列及其相关类。查询类可以通过Data Access Object对数据源方便地构造专属查询。使用这些查询对象的开发人员需要用方便而可靠的方式建立这些对象。这样就要开发一个查询建立器。Builder设计模式正好适合这个用途。

Builder设计模式

与Abstract Factory模式不同，Builder设计模式不是考虑创建相关产品系列，而是一步一步构造复杂对象。

这个模式可以更好地控制构造过程，定义建立产品所需的步骤，然后一次性提供最终产品。图7.8显示了Builder设计模式的结构。

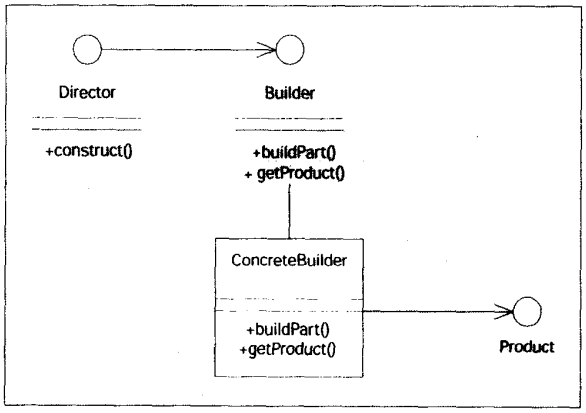

图7.8

Builder设计模式的参与者如下：

- **Builder（建立器）**——定义用一组步骤建立产品的接口
- **Concrete Builder（具体建立器）**——定义建立产品的步骤
- **Director（定向器）**——委托建立器建立产品部件
- **Product（产品）**——由多个部件构成的复杂组件

Builder使构造过程更加模块化，另外，可以对同一结构进行变化，重复产品建立过程，从而复用建筑块。

此外，还可以建立具有相似组件的不同产品，这是今后扩展的强大工具，见下节介绍。

在J2EE情境中实现Builder

有时在J2EE应用程序中需要用数据库以外的专属数据源。例如，数据源可以使用特定查询语言。通常，会话Bean或实体Bean用Data Access Object访问外部数据源。Data Access Object还用查询建立器向外部数据源建立和发表查询。

查询接口如下，是建立器的产品：

```
package reuse;

/* Specifies the interface of a query */
public interface Query {

  /* Represents the string·value of the query
   * @return a String representing me
   */
  public String stringValue();

  /* Returns a String representation of me
   * @return a String representing me
   */
  public String toString();

}
```

QueryBuilder还定义接口如下：

```
package reuse;

/* Interface for building queries uses Builder Pattern */

public interface QueryBuilder {

  /* Builds the header of the query */
  public void buildQueryHeader();

  /* Builds the query token */
  public void buildQueryToken();

  /* Builds the query body */
  public void buildQueryBody();

  /* Returns the built query */
  public Query getQuery();

}
```

我们有个包类，实现**QueryBuilder**接口。实现类如下：

```
package reuse;

import java.util.Iterator;

class QueryBuilderImpl implements QueryBuilder {

  private String queryHeader = "";
  private String queryToken = "";
  private String queryBody = "";
  private ValueObject bodyParams;

  /* Creates instance of me */
```

```
    public QueryBuilderImpl( ValueObject bodyParams ) {
      this.bodyParams = bodyParams;
    }

    /* Builds the header of the query */
    public void buildQueryHeader() {
      queryHeader = "External Data Source Query";
    }

    /* Builds the query token */
    public void buildQueryToken() {
      queryToken = "CR";
    }

    /* Builds the query body */
    public void buildQueryBody() {
      StringBuffer buf = new StringBuffer();
      Iterator propertyNames = bodyParams.getPropertyNames();
      while( propertyNames.hasNext() ) {
        String propertyName = ( String )propertyNames.next();
        buf.append( propertyName + "="
                    + bodyParams.getProperty( propertyName ) );
        if( propertyNames.hasNext() ) {
          buf.append(",");
        }
      }
      queryBody = buf.toString();
    }

    private String getQueryHeader() {
      return queryHeader;
    }

    private String getQueryToken() {
      return queryToken;
    }

    private String getQueryBody() {
      return queryBody;
    }

    /* Returns the built query */
    public Query getQuery() {
      return new QueryImpl( getQueryHeader(),
                            getQueryToken(),
                            getQueryBody() );
    }

  }
```

QueryImpl类如下，实现Query接口：

```
package reuse;

/* Defines a simple implementation of a query */
class QueryImpl implements Query {

  private String queryHeader = "";
  private String queryToken = "";
  private String queryBody = "";

  QueryImpl( String queryHeader,
             String queryToken,
             String queryBody ) {
    this.queryHeader = queryHeader;
    this.queryToken = queryToken;
    this.queryBody = queryBody;
  }

  public String toString() {
    return stringValue();
  }

  public String stringValue() {
     StringBuffer buf = new StringBuffer();
     buf.append( getQueryHeader() + "|" );
     buf.append( getQueryToken() + "|" );
     buf.append( getQueryBody() + "||" );
     return buf.toString();
  }

  protected String getQueryHeader() {
    return queryHeader;
  }

  protected String getQueryToken() {
    return queryToken;
  }

  protected String getQueryBody() {
    return queryBody;
  }

}
```

最后，Façade类的第二个方法根据建立的ValueObject对客户机创建查询：

```
package reuse;

/* Represents a facade interface to this package */

public interface Facade {
```

```
    /* Creates and returns a value object specific to this package
     * @return ValueObject specific to this package
     */
    public ValueObject getValueObject( String name ) throws CreateException;

    /* Builds and returns a query for specified value object
     * @param valueObject query parameters
     * @return Query the built query
     */
    public Query getQuery( ValueObject valueObject );

    //other facade methods

}
```

FacadeImpl类是个定向器，用Builder接口构造查询产品，如下列Java代码所示：

```
package reuse;

/* Represents a facade implementation */

public class FacadeImpl implements Facade {
  ValueObjectFactory factory = new
  ValueObjectFactoryImpl();

  /* Creates and returns a value object specific to this package
   * @return ValueObject specific to this package
  */
  public ValueObject getValueObject( String name ) throws CreateException {
    return factory.createValueObject( name );
  }

  /* Builds and returns a query for specified value object
   * @param valueObject query parameters
   * @return Query the built query
   */
  public Query getQuery( ValueObject valueObject ) {
    QueryBuilder builder = new QueryBuilderImpl( valueObject );
    builder.buildQueryHeader();
    builder.buildQueryToken();
    builder.buildQueryBody();
    return builder.getQuery();
  }

  //other facade methods

}
```

修改后的类框图7.9所示，显示了查询类系列的查询接口和建立器类。

由于门户只提供Query接口，因此很容易改变查询实现而不影响任何客户机代码。

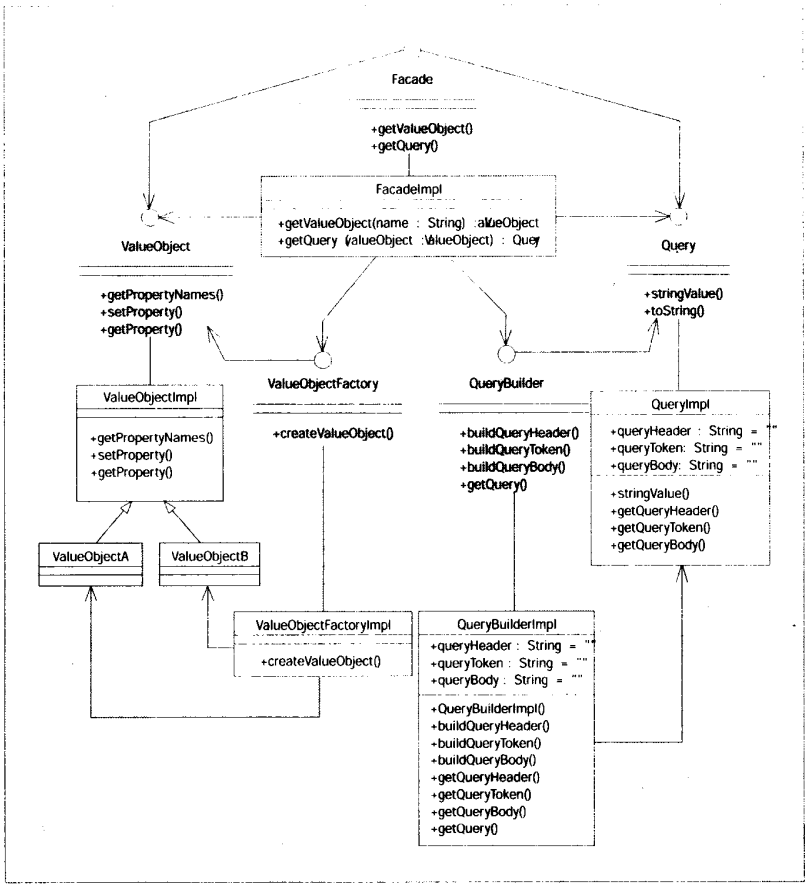

图7.9

Builder允许今后开发人员根据与当前查询相同结构定义新的查询，具有高度复用性。要引入新的或修改的实现，只要创建新的建立器实现和新的查询实现。

需要定义明确的扩展框架，同时在查询类系列之间共享共同行为。由于今后的项目可能使用这些查询，因此需要扩展查询类系列。帮助实现这个目标的设计模式是Template Method设计模式。

Template Method设计模式

这个设计模式可以让类定义操作的算法结构，而把操作中的一些步骤推迟到派生类中定义。

Template Method设计模式提供了强大的复用与扩展工具，可以让类与父类共享共同行为，在派生类中定义不同行为，同时保持相同结构的方法调用。Template Method还可以控制类扩展方式。

这个设计模式利用模板操作，在父抽象类中定义，并定义算法步骤。派生类不能覆盖

模板操作。模板操作调用不同的方法，如具体操作和抽象操作。这里的抽象操作通常指原始操作。图7.10显示了Template Method设计模式的结构。

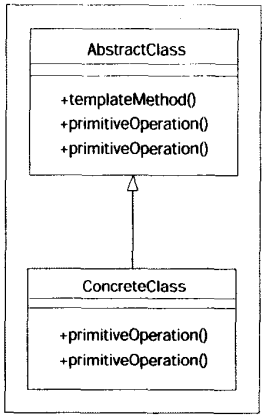

图7.10

Template Method类的参与者如下：

- **Abstract Class（抽象类）**——定义模板方法，调用原始操作实现算法步骤，由派生类实现。
- **Concrete Class（具体类）**——实现原始操作，是抽象类定义和扩展的。

在J2EE情境中实现Template Method

在J2EE情境中，和任何Java应用程序中一样，Template Method设计模式提供了管理扩展性和演变一组相关类的强大技术。

作为演示，我们决定扩展上节开发的QueryImpl类，提供查询实现的变形。

查询用Builder设计模式建立，可以用子类支持不同类型的查询。QueryImpl类是抽象的，定义模板操作stringValue()。这个操作是最终的，派生类不能覆盖。这个操作调用六个保护的具体与抽象操作。Template Method和其调用的其他操作在下列代码中高亮显示：

```
package reuse;

/* Represents a query implementation */

abstract class QueryImpl implements Query {

  private String queryHeader = "";
  private String queryToken = "";
  private String queryBody = "";

  /* Creates instance of me */
  QueryImpl( String queryHeader,
             String queryToken,
             String queryBody ) {
```

```
    this.queryHeader = queryHeader;
    this.queryToken = queryToken;
    this.queryBody = queryBody;
  }

  public String toString() {
    return stringValue();
  }

  public final String stringValue() {
     StringBuffer buf = new StringBuffer();
     buf.append( getQueryHeader() );
     buf.append( getExtraHeaderInfo() + "|" );
     buf.append( getQueryToken() + "," );
     buf.append( getExtraTokens() + "|" );
     buf.append( getQueryBody() + "," );
     buf.append( getExtraBody() + "||" );
     return buf.toString();
  }

  /* A Template method uses a primitive operation */
  protected String getQueryHeader() {
    return queryHeader;
  }

  /* A Template method uses a primitive operation */
  protected String getQueryToken() {
    return queryToken;
  }

  /* A Template method uses a primitive operation */
  protected String getQueryBody() {
    return queryBody;
  }

  /* A primitive operation defined by derived class */
  abstract protected String getExtraHeaderInfo();

  /* A primitive operation defined by derived class */
  abstract protected String getExtraTokens();

  /* A primitive operation defined by derived class */
  abstract protected String getExtraBody();

}
```

两个类QueryA与QueryB扩展QueryImpl类，定义上面高亮显示的原始操作。QueryA类如下：

```
package reuse;
```

```
/* A special query of type A */
final class QueryA extends QueryImpl {

  /* Creates instance of me */
  QueryA( String queryHeader,
          String queryToken,
          String queryBody ) {
    super( queryHeader, queryToken, queryBody );
  }

  /* A primitive operation called by super class */
  protected String getExtraHeaderInfo() {
    return "";
  }

  /* A primitive operation called by super class */
  protected String getExtraTokens() {
    return "A";
  }

  /* A primitive operation called by super class */
  protected String getExtraBody() {
    return "className=" + this.getClass().getName();
  }

}
```

QueryB类如下：

```
package reuse;

/* A special query of type B */

final class QueryB extends QueryImpl {

  /* Creates instance of me */
  QueryB( String queryHeader,
          String queryToken,
          String queryBody ) {
    super( queryHeader, queryToken, queryBody );
  }

  /* A primitive operation called by super class */
  protected String getExtraHeaderInfo() {
    return "";
  }

  /* A primitive operation called by super class */
  protected String getExtraTokens() {
    return "B";
  }
```

```
    /* A primitive operation called by super class */
    protected String getExtraBody() {
      return "className=" + this.getClass().getName();
    }
}
```

最后，类框架图7.11显示了Template Method设计模式的工作情况。

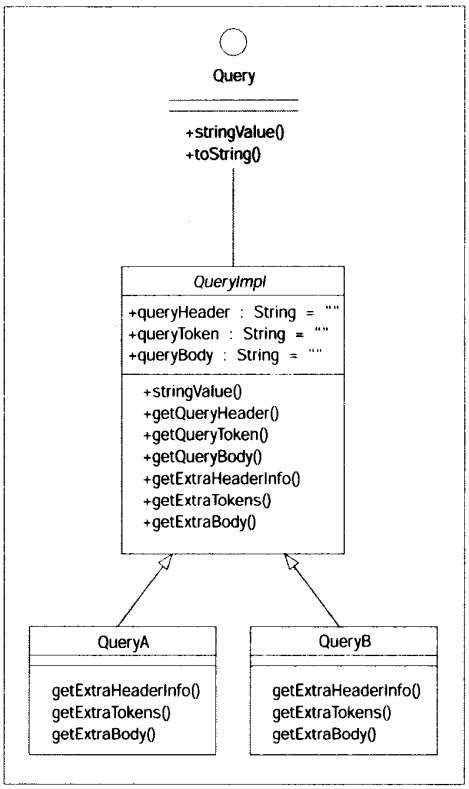

图7.11

将模式汇编成框架

前面对我们的案例采用了下列模式：

- **Façade模式**
  提供高水平接口对子系统进行内部抽象，使其更容易使用并减少耦合。
- **Abstract Factory模式**
  Abstract Factory设计模式可以创建相关对象系列而不声明具体类。
- **Decorator模式**
  Decorator设计模式不用子类而在类中增加行为。
- **Builder模式**

这个模式可以更好地控制构造过程，定义创建产品所需的步骤，然后一次性提供最终产品。

- **Template Method**模式

这个设计模式可以让类定义操作的算法结构，而把操作中的一些步骤推迟到派生类中定义。

最后，我们将前面介绍的所有模式汇编成框架图7.12。

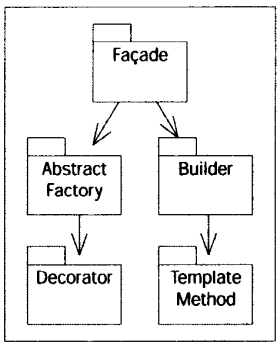

图7.12

下列框图7.13显示了实际类关系及模式如何映射类与接口。

小结

可以用许多软件设计模式增加软件的复用性、可维护性和扩展性。下列模式可以提高将来的扩展性：

- Decorator设计模式
- Template Method设计模式
- Abstract Factory设计模式
- Façade设计模式
- Builder设计模式

通过设计模式可以大大增加软件的复用性、维护性和扩展性。我们开发与设计的组件只演示了许多模式中的几个。

可以总结出下列经验：

- 为了得到更有效的可复用设计，设计时应定义尽量多个接口，还要提供Ximpl类，实现相关接口。
- 避免用基本类型的参数或返回值，而要使用对象，可以用toString()与equals()之类的方法，帮助维护更整洁的Java代码和促进包装，例如每个对象实现自己的toString()与equals()方法。
- 组件越抽象，越容易复用和扩展。

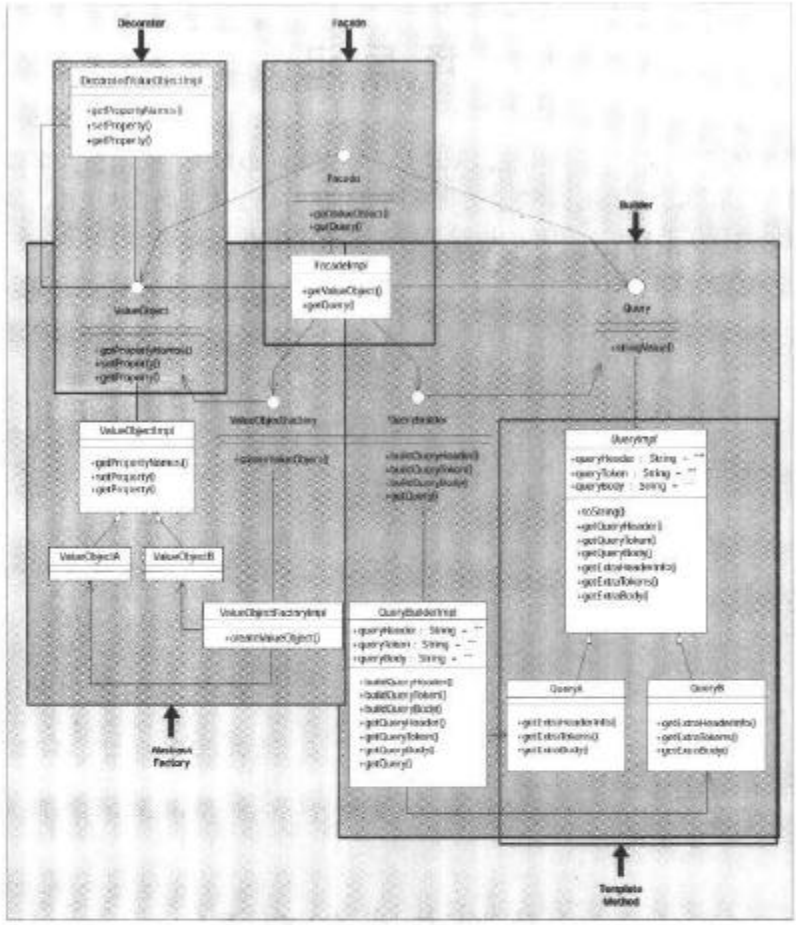

图7.13

- 采用促进扩展的设计模式（如Façade, Abstract Factory, Builder, Decorator与Template Method）。
- 采用促进包装与分离的设计模式，如Façade与Abstract Factory。
- 用代理尽量缩小类，例如用Decorator设计模式。长度并不重要，小而全的类更容易维护、复用和扩展。
- 设计继承时，一定要保证实际面向对象关系，如通用化和特殊化。许多情况下，使用委托与累计更好。在许多应用程序中，滥用继承，可能使应用程序紧耦合，很难维护和复用。