6

应用程序设计

为了说明创建使用路由器和分解的自助服务程序中使用的技术，我们为此项目创建一个示例应用程序。本章将讨论用于创建此应用程序的 Java 设计技术。

此处介绍的信息旨在补充《使用 WebSphere Application Server 4.0 的自助服务模式》编号：SG24－6175。我们以那本书介绍的应用程序设计为基础，将设计延伸为包括消息发送功能。为此，我们非常重视 Sun 的 Java 消息服务（JMS）在企业消息发送应用程序中的作用。我们将探讨支持 API，如 Sun 的 Java 命名和目录接口。另外，还将介绍 WebSphere 的 JMS 侦听器，而 JMS 侦听器是 WebSphere 企业服务中扩展的消息发送支持的组成部分。

本章最后将从实例应用程序抽取一个应用案例，并详细讲述该应用程序设计。

6.1 JMS概述

Sun Microsystem 的 Java 消息服务（JMS）是定义 Java 应用程序与企业消息发送中间件（如 MQSeries）如何进行交互的 API。本节假定基本了解消息发送系统涉及的概念，并着重介绍如何把这些概念映射到 JMSAPI。

6.1.1 消息模型

首先，了解消息发送模型的不同类型非常重要。每种模型在为该模型定义专用操作的 JMS 中有一组接口。有两种体系结构可用于消息发送应用程序：点到点（PTP）和发布／预订（pub／sub）。

点到点应用程序是根据以下观点创建的：每个消息均以特定队列为其地址。接收端应用程序从该队列获取消息，并对其进行相应处理。在 JMS 中，PTP 类型带有“队列”前缀，如下表所示。

JMS 父级	**PTP 类型**
ConnectionFactory	QueueConnectionFactory
Connection	Que ueConnection
Destination	Queue
Session	QueueSession
MessageProducer	QueueSender
MessageConsumer	QueueReceiver

发布／预订应用程序则是按照以下思想创建的：根据消息内容路由消息。消息被发送到消息代理，该消息代理根据其内容把消息路由到适当订阅者。然后，订阅者就可对该消息进行适当处理。在 JMS 中，发布／预订类型带有“主题”（Topic）前缀。

在本书中，我们重点考察 PTP 应用程序。

6.1.2 消息组件

由于 JMS 是为创建、发送和接收消息而存在，因而您会认为消息是 JMS 的核心组件。JMS 消息由以下几部分构成：

- 标题：包含识别和路由消息的信息。
- 属性：可以有选择地添加到消息的定制值。属性可以是：
  - 应用程序特定的：添加到消息的属性由 JMS 应用程序使用。
  - 标准：JMS 属性。
  - 提供者特定的：某个消息提供者特有的属性。
- 正文：数据。

6.1.3 消息类型

JMS 提供五种不同专业的消息，以提供访问每个消息内容的不同操作。图 6－1 显示这些类型。

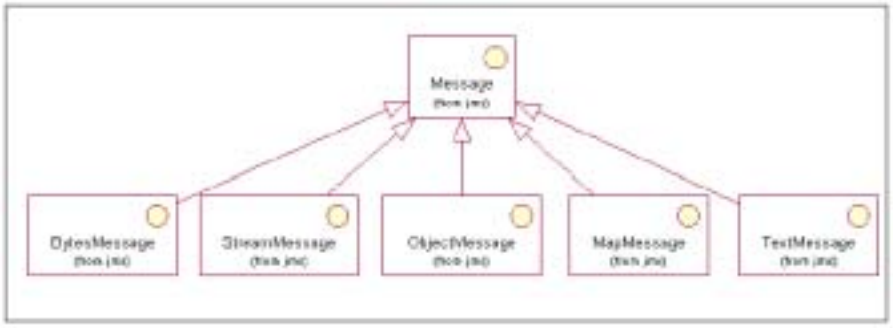

图6－1 JMS 消息类型层次

不同消息类型包括：

- BytesMessage：包含访问字节流的操作。此消息类型将与任何现有消息格式相匹配。
- StreamMessage：包含访问 Java 原始值流的操作。
- ObjectMessage： 包含访问序列化 Java 对象的操作。如果应用程序设计需要序列化一个以上的对象，那么就请使用“集合”（Collection）对象。

- MapMessage：包含从消息正文访问一组键值对的操作。关键字必需为字符串，值必须为原始类型。
- TextMessage：包含访问作为字符串的消息正文的操作。在实例应用程序中，我们把 TextMessage 用作传输消息（用 XM 进行编码）的手段。

6.1.4 目标类型和子类型

下一个要掌握的概念是“ 目标 ”类型和子类型。基本上，目标是消息的容器。消息从一个应用程序发送到目标，然后由另一个应用程序从目标中将其删除。

熟悉 MQSeries 的人也应该熟悉术语“ 队列 ”，而队列就是消息的目标。JMS 把队列称为 PTP 应用程序中目标的一种特殊类型。按照消息发送模型，目标既可称为队列（用于 PTP），也可称为主题（用于发布／预订）。

另外，每个消息发送模式有一个更加专业化的目标类型：TemporaryQueue 和 TemporaryTopic。这些类型为在“ 连接 ” （Connection）期间存在的目标。

图 6－2 显示目标层次结构。

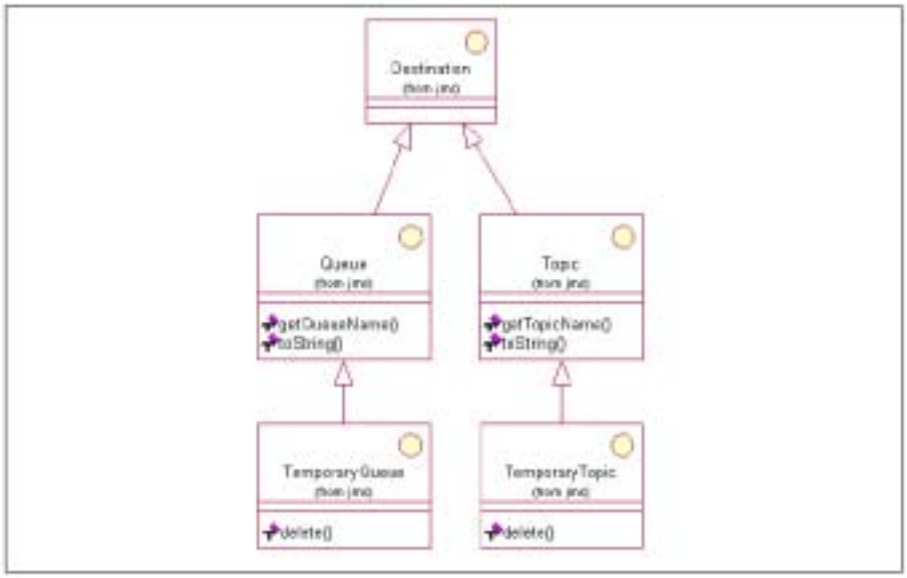

图6－2 JMS 目标类型层次

下面是 JMS API 中其他关键字类型的摘要：

- 分别使用 MessageProducer 和 MessageConsumer 将消息发送到目标或从目标接收消息。在 PTP 模型中，子类型 QueueSender 将消息发送到目标，而 QueueReceiver 从目标接收消息。
- MessageProducer 和 MessageConsumer 是使用‘会话”（PTP 模型中的 QueueSession）创建的。会话还提供各种消息类型的库方法和临时目标（例如，TemporaryQueue）。
- 会话实例是使用连接创建的。连接代表着与 JMS 提供者的连接。在 PTP 模型中，子类型为 QueueConnection。
- 最后，连接是使用 ConnectionFactory（PTP 模型中的 QueueConnectionFactory）创建的。ConnectionFactory 将是建立消息发送应用程序的起点。

这只是对 JMS 的简单概述。有关 JMS 的更多信息，请参阅“ Java 消息服务 ”规范，可从以下地址下载：

http：／／java.sun.com／products／jms／index.html

6.2 JMS和JNDI

为了提高消息发送应用程序的可移植性，JMS 依靠 Sun 的 Java 命名和目录接口（JNDI），可向 Java 应用程序提供命名服务。

JMS 具有所谓的“ 管理对象 ”，管理对象具有两种类型 ：ConnectionFactory 或 Destination。管理员可以把具有这两种类型的对象放在 JNDI 命名空间中，供消息发送应用程序进行访问。

图 6－3 显示与 Java 应用程序相关的 JMS 和 JNDI 的角色。这两种 API 高于任何特定服务提供者，并且封装任何特定供应商的信息。

因此，在支持消息发送的应用程序中使用这些技术的开发人员只需熟悉 API，而无需熟悉具体的消息发送系统。

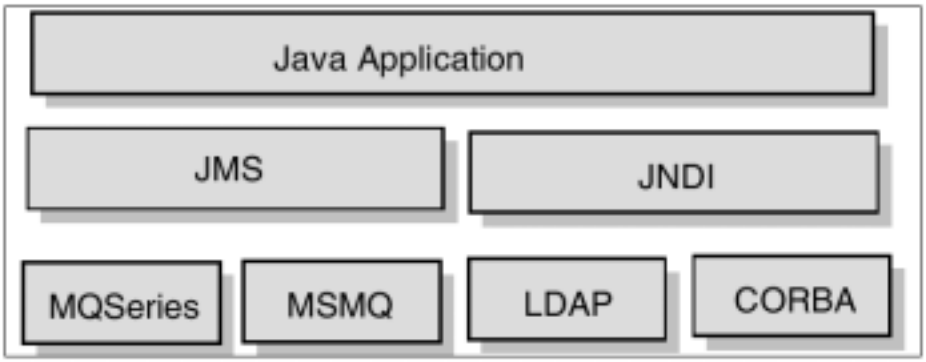

图6－3 与应用程序相关的 JMS 和 JNDI 的角色

6.2.1 什么是JNDI？

Java 命名和目录接口（JNDI）提供一种从 Java 应用程序中访问目录和命名功能的手段，并且不受任何特定目录提供者工具的限制。

通俗地说，JNDI 好似 Java 应用程序的电话簿。名称与信息关联，而信息采取对象的形式。在 JNDI 中，与对象关联的名称称为绑定。绑定存储在一个上下文内。

上下文像某个城市的电话簿。曼哈顿的电话簿存储居住在曼哈顿的人们的电话号码。要查找芝加哥某个人的电话号码，我们需要命名系统中不同的上下文。JNDI 没有绝对命名方法的概念。所有名称都与某个上下文相关。

命名系统是一组具有相同类型的上下文。这样，纽约州的所有电话薄就是一个命名系统。而所有电话簿中的所有名称的集合就是命名空间。即命名空间是命名系统中所有名称的集合。

命名服务的其他实例包括电子邮件、URL 和文件系统。

当然，上面提供的关于 JNDI 的信息已足以能够帮助您理解 JNDI 利用 JMS 能有多大帮助。有关更多信息，请参阅《Java 命名和目录（JNDI API）》，并可从以下链接下载：http：／／java.sun.com／products／jndi／index.html

6.2.2 为何使用JNDI？

JNDI 不是 JMS 的要求，那么，使用 JNDI 有什么好处？

Sun 设计 JMS 规范的目标之一就是可移植性。使用 JNDI，开发人员不必考虑如何连接到消息发送系统这样的细节，而且，无需把这些细节嵌入应用程序。

开发人员只需要知道应用程序所依赖的 JMS 对象的 JNDI 名称即可。系统管理员负责创建适当绑定。使用此策略，JMS 应用程序将不受 JMS 提供者的限制。

请看实例 6－1，该实例连接一个队列，并发送简单的文本消息。在此实例中，我们将说明不使用 JNDI 的弊端。

实例 6－1：不使用 JNDI 的编码实例

```
String QMGR ="";
String QUEUE ="SYSTEM.DEFAULT.LOCAL.QUEUE";

QueueConnectionFactory factory =null;
factory =new MQQueueConnectionFactory () ;
 ( (MQQueueConnectionFactory) factory) .setQueueManager (QMGR) ;
QueueConnection qCon =factory.createQueueConnection () ;
QueueSession qSession =qCon.createQueueSession (false,
Session.AUTO__ACKNOWLEDGE) ;
```

```
Queue q =qSession.createQueue (QUEUE ) ;
QueueSender qSender =qSession.createSender (q ) ;

TextMessage message =qSession.createTextMessage ("To the Bat－Queue!") ;
qSender.send (message ) ;

qCon.close () ;
```

第二个代码块需要加以注意。

JMS 应用程序中的第一个步骤是获取 QueueConnectionFactory 实例。注意 :在不使用 JNDI 的实例中，我们需要创建新的 MQQueueConnectionFactory 实例，该实例实施 QueueConnectionFactory 接口。这会存在许多方面的不便。

首先，看一下对高度依赖于特定供应商信息（如上例所示）的消息发送应用程序进行开发和维护所需要的角色：

- 角色 1：需要创建适当队列和队列管理器的管理员。
- 角色 2：开发、测试和应用消息发送应用程序的开发人员。

在上面的实例中，开发人员必须知道关于队列的特定管理信息，例如，队列管理器的名称以及队列的名称。可以用属性文件的信息使此信息具体化，但是，如果其中任何信息发生变化，开发人员（或负责维护应用程序的任何人）就需要在适当地方更新此信息。

使用 JNDI，开发人员就不再依赖于特定管理信息，而只需要知道如何从 JNDI 命名空间获取对象。管理员可以更改队列管理器和队列的名称，而且，只要 JNDI 名称保持不变，就不会影响消息发送应用程序。

其次，在讨论 JMS 应用程序中使用特定供应商的类的弊端时，要考虑 Sun 的可移植性目标 — 这一目标在上述实例中并未实现。假定要在具有许多消息发送平台的企业中部署应用程序。如果使用上述实例中的约定部署此应用程序，就会需要开发、测试、应用和维护多个应用程序版本。

最后，就像我们在实例应用程序中看到的一样，使用 JMS 的 WebSphere 应用程序需要在 JNDI 命名空间中注册适当对象。换句话说，我们必须具有一些在命名空间中注册的对象，否则，消息发送应用程序就无法工作。

6.2.3 JMS如何使用JNDI

JMS 通过管理对象利用 JNDI。管理员按照每个供应商特有的方式把对象放在 JNDI 命名空间中。

那么，JNDI 命名空间中可以放置什么呢？管理对象可以是以下两种类型之一：ConnectionFactory（图 6－4）或 Destination（第 65 页的图 6－2）。管理员可以把属于这两种类型的任何 JMS 对象放在该命名空间之中，供应用程序进行访问。

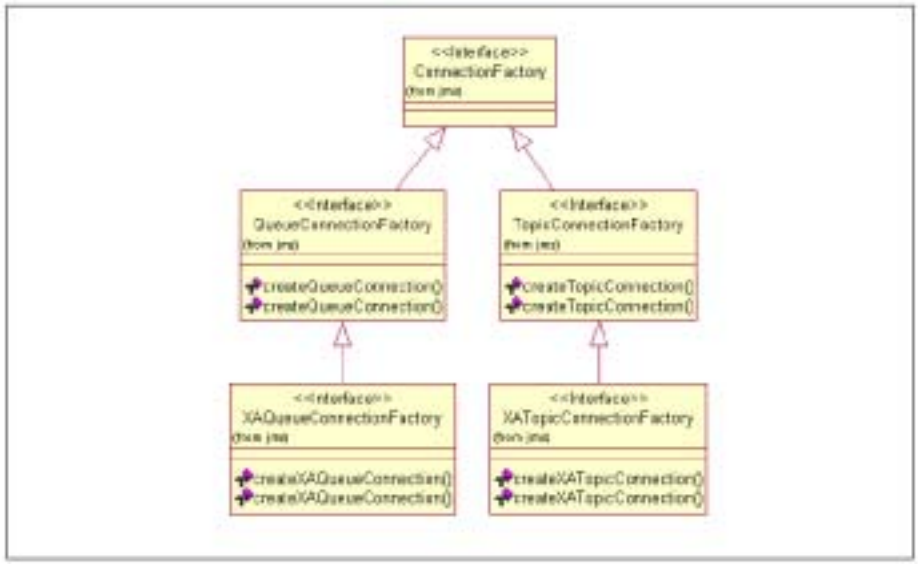

图6－4 JMS ConnectionFactory 层次结构

实例 6－2 显示如何重新设计第 67 页的实例 6－1 以支持 JNDI。此实例假定管理员已在 JNDI 命名空间中放入一个绑定到名称“ ivtQCF ”的 QueueConnectionFactory 以及一个绑定到名称“ ivtQ ”的 Destination 队列。

实例6－2：支持 JNDI 的 JMS 实例

```
java.util.Hashtable environment =new java.util.Hashtable () ;
environment.put (Context.PROVIDER__URL, "iiop：／／localhost") ;
```

```
environment.put (Context.INITIAL__CONTEXT__FACTORY,
"com.ibm.ejs.ns.jndi.CNInitialContextFactory") ;

Context ctx =new InitialContext (environment ) ;

QueueConnectionFactory factory = (QueueConnectionFactory) ctx.lookup ("ivtQCF") ;

QueueConnection qCon =factory.createQueueConnection () ;
QueueSession qSession =qCon.createQueueSession (false,
Session.AUTO__ACKNOWLEDGE) ;

Queue q = (Queue) ctx.lookup ("ivtQ") ;
QueueSender qSender =qSession.createSender (q ) ;

TextMessage message =qSession.createTextMessage ("To the Bat—Queue!") ;
qSender.send (message ) ;

qCon.close () ;
```

提示：在上面的实例中，没有特定供应商的类。而且，代码中没有像队列管理器名称和队列名称这样的信息。所有信息都封装在管理员提供的管理对象中。

那么，管理员是如何把这些对象放在 JNDI 命名空间呢？如前所述，此步骤是供应商特定的。在 MQSeries 中，MA88 SupportPac 具有管理称为 JMSAdmin 的管理对象的工具。

6.3 WebSphere到MQSeries连接选项

从应用程序服务器放在 MQSeries 队列上的消息可能直接产生于某个服务件，也可能是从某个命令 Bean 或 EJB 发送来的。无论哪种方法，都会使用可用的 MQSeries Java API 把该消息发送到队列管理器。每个 API 均具有使其适合于某种情形的某些特征，具体取决于您所拥有的优先权。 但是，选择 API 可能会对用于分发应用程序组件的选项产生影响。

下面我们讨论两种 API：

- 用于 Java（MQ base Java）包的 MQSeries 类， com.ibm.mq.jar。MQ 基本 Java 使 Java 小程序、应用程序、服务件和 EJB 能够向 MQSeries 发出呼叫和查询。

- 用于 Java 消息服务（MQ JMS）包的 MQSeries 类， com.ibm.mqjms.jar。MQ JMS 实施 Sun 的 Java 消息服务 （JMS），从而使 JMS 程序能够访问 MQSeries。

如果应用程序可移植性、供应商独立性和位置透明性都很重要，那么 JMS 就是技术选择明显的优胜者。JMS 使用消息发送概念摘要提供独立于供应商的消息发送 API，而 MQSeries 在其下面实施了 JMS 接口。使用目录命名服务（MQSeries 消息服务），把作为 MQSeries 队列管理器和队列的真正实体 — 对象 — 定义到 JMS。MQ JMS 支持 JMS 的点到点和发布／预订模型。这是我们在实例中所使用的 API。

MQ base Java 和 MQ JMS 提供到达 MQSeries 的两个连接选项：

- 直接连接到队列管理器的绑定模式
- 使用 TCP／IP 连接到队列管理器的客户机模式

两个连接选项都支持连接池。

6.3.1 Java绑定模式

从 Java 到 MQSeries 的最快链接是使用 Java 绑定模式。它提供到达 MQSeries 队列管理器的直接连接，而 MQSeries 队列管理器与应用程序驻留在同一主机上。此案例中的关键连接参数是队列管理器名称。

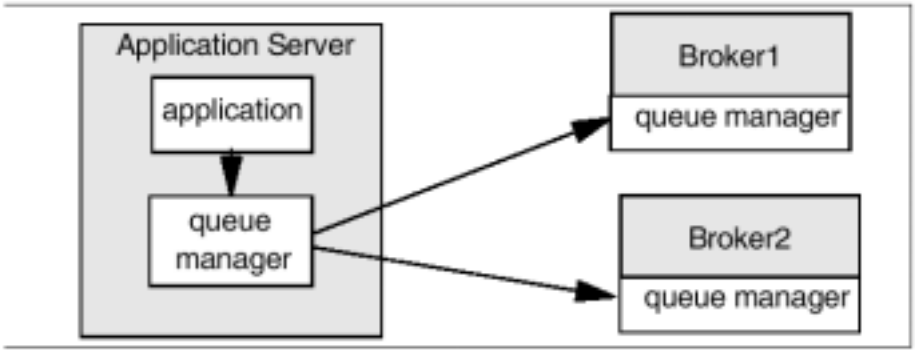

图6－5 Java 绑定模式

连接到本地队列管理器有多个重要优势。首先，与远程队列管理器的连接相反， 在您自己的主机上建立到达队列管理器的连接的可能性会非常高。其次，避免在建立到达队列管理器的网络连接时所花费的时间。第三，本地队列管理器可以在多个代理之间分配工作。在您的网络中，如果连接性能优先，那么使用绑定模式是必然选择。

使用绑定模式，针对支持 XOpen／XA 标准的数据库和驱动程序，您还可以把 MQSeries 用作需要 MQSeries 更新和数据库更新的工作单位的 XA 资源协调者。

Java 客户机模式

用于 Java 的 MQSeries 类以及用于 JMS 的 MQSeries 类以客户机模式提供与 MQSeries 的连接。这类似于绑定模式，与队列管理器的连接是通过服务器连接通道进行的，即应用程序可以连接到其他主机上的队列管理器。关键连接参数是主机名、TCP／IP 端口、以及服务器连接通道名称。

如果不想使 MQSeries 与应用程序服务器驻留在同一设备上，则最好使用客户机模式。客户机模式使您可以直接连接到远程 MQSeries 队列管理器。

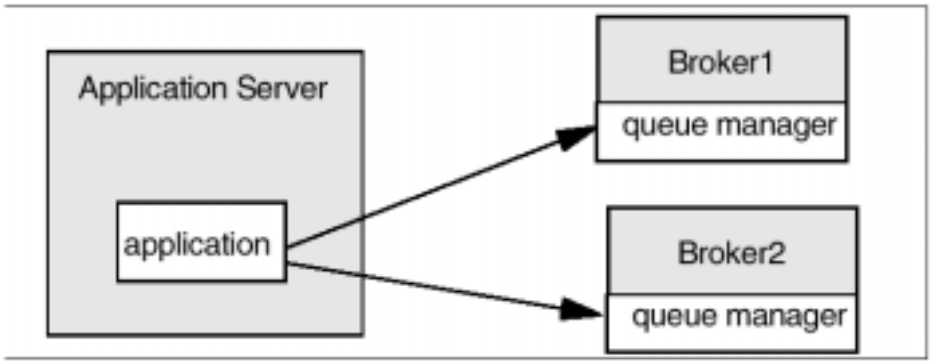

图6－6 到达远程代理的客户机模式

当直接连接到某个代理上的队列管理器（如图 6－6 所示）时，您将放弃队列管理器提供的任何工作负载分配。应用程序必须决定把工作发送到哪个代理，并且一切工作负载分配必须在应用程序本身中进行，而这种方法是不提倡的。即使把队列管理器放在群集中也无济于事，因为队列管理器总是把工作发送到代理的本地实例。另一个缺陷是，XOpen／XA 为对 MQSeries／JMS 协调的承诺提供便利，而且无法通过将 MQSeries 用

作事务处理协调者来使用数据库。

解决该问题的一种方法是连接到没有代理实例的远程队列管理器，但也只是用于工作负载分配，如图 6－7 所示。

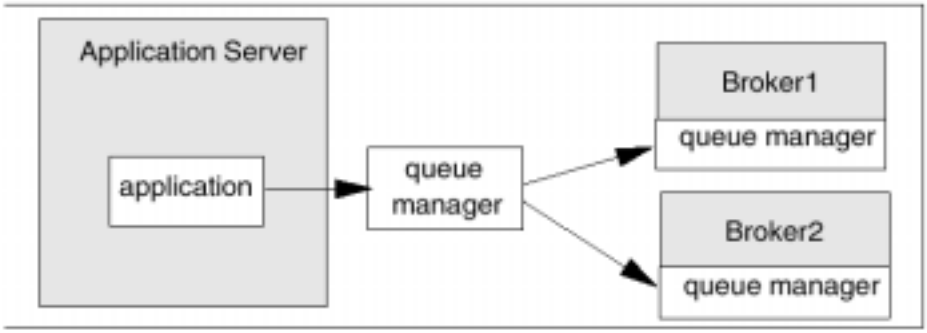

图6－7 到达远程队列管理器的客户机模式

仍然存在网络连接时间，事实上，由于引入了中间系统使得事情更加麻烦。但是，您确实拥有队列管理器工作负载分配的优势，以及连接到远程队列管理器的能力。而另一种方法是使用 TCP／IP 负载平衡，但这并不是 MQSeries 的功能，而且不在本书探讨范围之内。

凭借经过内部 TCP／IP 栈，客户机模式还可用来连接到本地队列管理器。很明显，这不如使用绑定模式方便，但它确实使您可以在一般环境中使用程序，而您不必知道队列管理器是否是本地的。还可以通过参数来确定不同连接选项，这样，不管连接类型为何，应用程序都处于就绪状态。无需确保传递正确的参数，就可以获得所需的连接类型。如果您正在与某个数据库进行交互，数据库表就是一个很好的选择。

MQ JMS 和 MQ 基本 Java 都允许您把消息从 WebSphere 中的 Java 应用程序直接放到远程代理的队列中。如果您考虑这样做，您应同时考虑性能问题。创建网络连接的成本将添加到每个请求的总成本中。对于每个请求，都创建 MQ 到客户机的会话。不存在长久的网络连接，这会影响同时运行数千个会话的能力。如果您为每个请求创建本地 MQ 会话，开销将就会低得多。网络连接现在是通过发送端－接收端通道对维持的，并且持续相当长时间。最好是持续时间长的 MQ 会话。在 MQSeries 的下一个版本中，创建本地会话的开销可望得到显著降低，因而会进一步拉大客户机模式和绑定模式之间的性能差

距。

6.3.2 把WebSphere用作事务处理协调者

用于 Java 消息服务的 MQSeries 类包括 JMS XA 接口。这些接口允许 MQ JMS 参与事务处理管理器协调的两段式确认，该事务处理管理器符合 Java Transaction API （JTA）。使用这些类，WebSphere Application Server 高级版可以在全局事务处理中协调 JMS 发送、接收操作，以及数据库更新。

WebSphere 提供一对附加的管理对象，这样 MQ JMS 就可以与 WebSphere 集成：

- JMSWrapXAQueueConnectionFactory
- JMSWrapXATopicConnectionFactory

您可以与使用 MQQueueConnectionFactory 和 MQTopicConnectionFactory 完全相同的方式使用这对对象。 但是在幕后，它们使用 JMS 类的 XA 版本，并使 MQ XAResource 参与 WebSphere 事务处理。

如果某个事务处理中使用一个以上 XAResource，WebSphere 协调者仅调用真正的两段式确认。调用单项资源的事务处理是使用一相优化承诺的。这在很大程度上消除了在分布式和非分布式事务处理中使用不同 ConnectionFactory 的必要。

有关详细信息，请参阅《使用 Java 的 MQSeries》编号：SC34－5456－06。此手册包含在 MQSeries 产品中。

6.4 处理异步消息

JMS 侦听器是随 WebSphere Enterprise Services 封装在一起的定制服务。它可侦听用于消息的 JMS 目标。当某个消息发送到监视目标时，侦听器通知称为消息 Bean 的专用 Enterprise JavaBean，消息 Bean 指派为处理对该特定目标的请求。JMS 侦听器角色的图形表示如图 6－8。

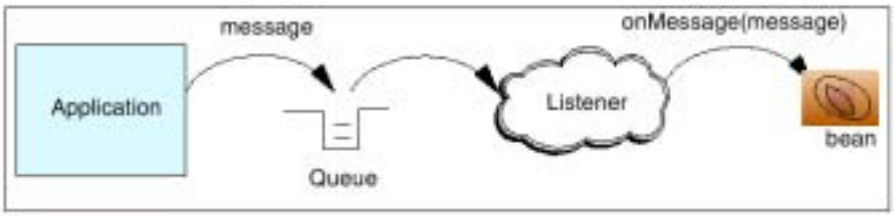

图6－8 使用 JMS 侦听器的应用程序间的消息流

请将上面的消息流与图 6－9 中的消息流进行比较。图 6－9 描述两个不使用侦听器的应用程序之间的消息流。

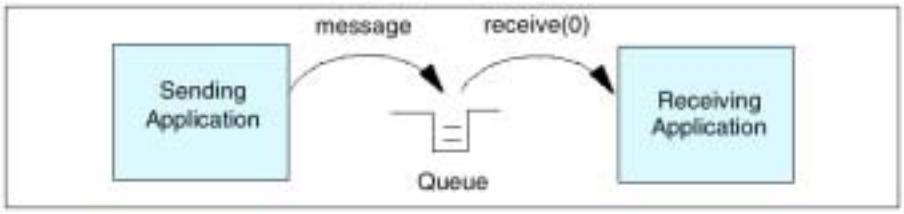

图6－9 不使用侦听器的应用程序之间的消息流

在图 6－9 中显示的情形下，接收端应用程序知道它将从给定目标接收消息。但是，并不知道何时接收以及接收的频繁程度。因此，接收端应用程序有两个从目标获取消息的选项：

1. 它可以进行呼叫以接收消息，并无限期等待。当消息最终到达时，呼叫应用程序重新获得对进程的控制。
2. 它可以使用呼叫接收消息并指定超时。一旦达到超时或从目标收到消息，呼叫进程重新获得对进程的控制。

这一问题的解决方案是使用 JMS 侦听器。这消除了开发人员对处理从目标到达的异步消息的逻辑进行编码的必要。此逻辑封装在侦听器内，这样接收到消息后，开发人员就可以专注于必须开发的逻辑。

侦听器的每个实例都映射到目标和消息 Bean。目标是受到监视的队列。消息 Bean 是接收消息时调用其 onMessage（）方法的类。

6.4.1 消息Bean

当从目标接收消息时，JMS 侦听器调用映射到目标的消息 Bean 中的 onMessage（）方法，从而传递接收到的消息。

与 WebSphere 的 WebSphere Enterprise Services 中 JMS 侦听器一起使用的消息 Bean，是对 Enterprise JavaBean （EJB） 2.0 规范中定义的消息驱动型 Bean 的早期改编。

WebSphere V4 中的 EJB 符合 EJB 1.1 规范，因此，使用 1.1 规范开发的任何消息 Bean 都应把最终迁移到 2.0 规范的必要考虑在内。在下面几段中，我们首先讨论使用 1.1 规范开发消息 Bean 的现行方法。接下来，我们将讨论消息驱动型 Bean 合同以及如何设计要迁移到要迁移到 2.0 规范的 1.1 Bean。

在 1.1 规范中，消息 Bean 只不过是包含商务方法 － onMessage（） － 的无国籍会话 Bean。此方法为何如此特别？

onMessage（）方法可在类型 javax.jms.MessageListener 上找到（请参见图 6－10）。在 JMS API 中，MessageListener 注册到 Session 或 MessageConsumers 以便异步通知它们传入的消息。

图6－10 javax.jms.MessageListener 接口

尽管它们不实施 MessageListener，我们创建的消息 Bean 必须具有同样的签名。

消息 Bean 必须具有以下组件。

远程接口

接口必须定义具有与 javax.jms.MessageListener 相同签名的 onMessage（）方法（Throws 子句除外）。要进行的逻辑事务似乎会扩展 MessageListener 接口；但是，远程接口中的所有方法必须丢弃 RemoteException。因此，该接口必须扩展 javax.ejb.EJBObject。

定义的商务方法必须遵循 RMI 的规则；因此，商务方法必须丢弃 java.rmi.RemoteException。

远程接口应看起来像实例 6－3 的样子：

实例 6－3：远程接口实例

```
public interface Example Message extends javax.ejb.EJBObject{
public void onMessage (Message message) throws RemoteException;
}
```

主接口

该接口必须扩展 javax.ejb.EJBHome。方法必须遵循 RMI 规则；因此，方法应丢弃 java.rmi.RemoteException。另外，Throws 子句还应包含 javax.ejb.CreateException。

该接口还必须定义一个不包含任何参数的 create（） 方法，并具有远程接口的一个返回类型。尽管会话 Bean 可以定义一个以上的 create（）方法（这些方法必须在 Bean 类中具有相应的 ejbCreate（）方法），2.0 规范中的消息驱动型 Bean 只能有一个不包含任何参数的 create（）方法。

实例 6－4 显示主接口看起来应是什么样子。

实例 6－4：主接口

```
public interface ExampleMessageHome extends javax.ejb.EJBHome{
public ExampleMessage create () throws CreateException, RemoteException;
}
```

Bean 类

该类必须实施 javax.ejb.SessionBean。该类必须是公用的，而且不能是最终的或抽象的。最后，该类还必须实施商务方法和 ejbCreate（）方法。

6.5 应用程序结构

本书的讨论和实例主要以名为 Guild of Weather Masters 的实例应用程序为基础。此应用程序最初是在《使用 WebSphere Application Server V4.0 的自助服务模式》编号：SG24－6175 中介绍的。从位于以下地址的电子商务模式网站下载 PDK－Lite 3.0 版，即可获得该应用程序：

http：／／www.ibm.com／framework／patterns

我们将把该应用程序扩展为包括消息发送功能。本节将逐个介绍各应用案例以阐释应用程序设计。然后，详细讲述一个案例来说明所使用的技术。有关与使用 JMS 无直接关

系的许多组件的详细信息，请参阅《使用 WebSphere Application Server V4.0 的自助服务模式》编号：SG24－6175。

6.5.1 模型－视图－控制器

自助式服务 Web 应用程序可视为浏览器和 Web 应用程序服务器之间的一组交互。这些交互始于浏览器对应用程序欢迎页（welcome page）的初始请求。这通常是由用户在浏览器上键入欢迎页 URL 完成的。用户通过单击按钮或链接，启动随后的所有交互，这样就把请求发送到 Web 应用程序服务器。Web 应用程序服务器处理该请求，动态生成结果页，并将其发送回客户机，同时带有一组按钮和链接以进行下一个请求。

仔细考察这些交互，就会发现一组需要在服务器端考虑的通用处理请求。这些交互可以很容易地映射到标准的模型－视图－控制器 （MVC）设计模式。

- ***模型：***我们称之为商业逻辑。它代表实施应用程序数据和商业逻辑的应用程序对象。我们建议用 JavaBeans 或 Enterprise JavaBeans 包装商业逻辑。这样就可以把商业逻辑与 Web 特定的交互控制器和显示页逻辑分离开来，从而把商业逻辑与 Web 编程的细节隔离开来，进而增强商业逻辑在 Web 应用程序和非 Web 应用程序中的复用性。
- ***视图：***我们称之为页面构造者。视图负责编排应用程序结果的格式，并负责生成要返回客户端的 HTML 页。尽管视图可以用 JSP 和／或服务件实施，JSP 允许直接用 HTML 开发模板页，并为页面的动态元素插入脚本逻辑以及为多部分页面插入 jsp：include 操作。因此，JSP 是实施构造页面组件的最佳选择。
- ***控制器：***我们称之为交互控制器。交互控制器是将独立于协议的商业逻辑与 Web 应用程序联接在一起的代码段。这就是说，交互控制器的主要责任是把 HTTP 协议特定的输入映射到独立于协议的商业逻辑（此商业逻辑可能由若干不同类型的应用程序使用）要求的输入，将商业逻辑的各个元素用脚本编写在一起，然后委派给页面构造组件，该页面构造组件将创建返回到客户机的响应页。

建议一般使用服务件来实施交互控制器。但是，对于不涉及条件逻辑或事务处理逻辑的简单应用程序，可以把交互控制器和页面构造逻辑组合成一个组件。在这种情况下，JSP 是最佳选择。

在《使用 WebSphere Application Server V4.0 的自助服务模式》编号：SG24－6175 中，详细讲述了 MVC。这里只是说明我们讨论的技术如何适合该体系架构。

6.5.2 实例应用程序应用案例

原始应用程序以如下假设为基础：存在一个称为 Guild of Weather Masters 的机构。该 Guild 有一笔总基金，可以把其中的资金（外太空） 转到各个站（行星）。只有 Guild 管理员可以转移资金，而所有用户均可显示 Guild 的资金余额或每个站的资金余额。

我们将扩展此应用程序，通过增加新的功能，使所有用户均可请求日志，该日志包含已执行的转帐和一个或多个站余额。

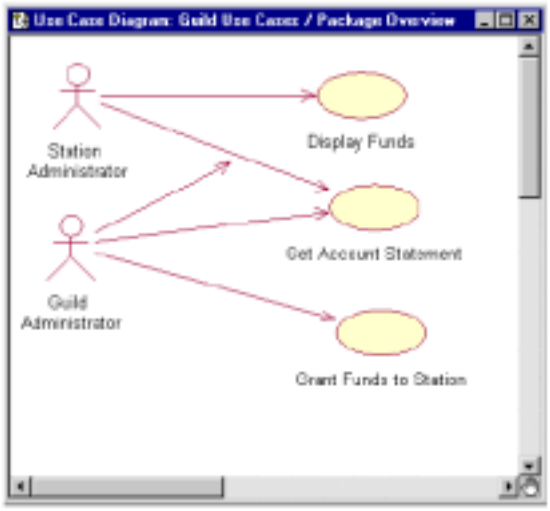

图 6－11 应用案例

我们的应用案例有两个角色，站管理员和 Guild 管理员。

站管理员可以显示 Guild 资金余额和各站资金余额，并请求显示各站转账记录的报表。

Guild 管理员可以显示 Guild 资金余额和各站资金余额，并可把资金从 Guild 转移到站，还可以请求显示各站转账记录的报表。

6.5.3 显示资金

该应用程序首先从后端数据库检索 Guild 资金余额以及各站资金余额，并把它们提交给用户。

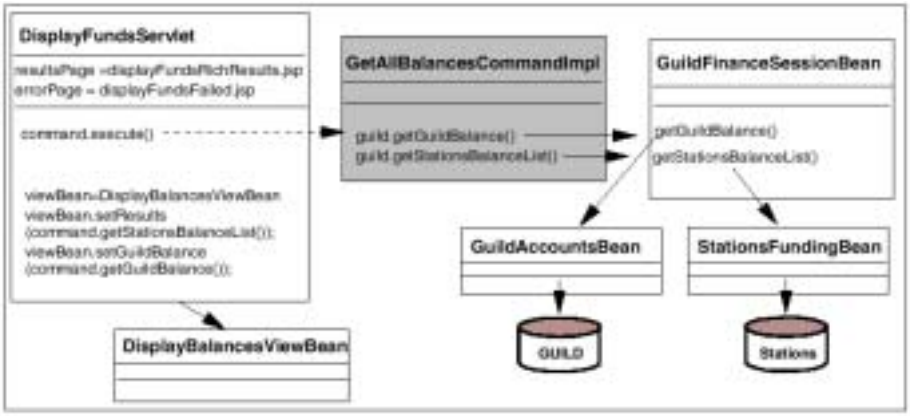

图6－12 显示资金流程

- ***视图：*index.jsp** 是进入应用程序的入口。此页上的链接“单击此处可获得最新余额”会把您引导到 DisplayFundsServlet （／PDK）。
- ***控制器：*DisplayFundsServlet** 代表交互控制器。服务件负责调用适当的命令 － 在此案例中，命令为 GetAllBalancesCommand。

执行该命令之后，服务件用具体实例说明视图 Bean （DisplayBalancesViewBean），并把从该命令检索的余额值存储在视图 Bean 中。

一旦“水合（hydrated）”视图 Bean，或用适当值填充视图 Bean，就将视图 Bean 添加到请求对象。最后，服务件把请求转发到适当 JSP。

- *模型：*GetAllBalancesCommand 代表从控制器到服务件的桥路。

  实施 GetAllBalancesCommandImpl 命令将引用 GuildFinanceSessionBean，这是该应用程序数据访问的正面。

  会话 Bean 使用代理模式把请求转发到 GuildAccounts 实体 Bean 和 StationsFunding 实体 Bean，以获取 Guild 余额和站余额。
- *模型：*GuildAccountsBean 和 StationsFundingBean：代表商业逻辑的简单 Java Beans。它们都实施 getBalance（）公用方法，该方法返回账户的当前值。
- *视图：*DisplayBalancesViewBean 由 DisplayFundsServlet 创建并保存 JSP 所需的结果。
- *视图：*DisplayFundsRichResults.jsp：代表显示页。它是构造响应页的 JSP，该响应页显示 Guild 账户余额以及每个气象站的余额。

6.5.4 转移资金

用户可以从 DisplayFundsRichResults JSP 中查看 Guild 余额以及各站余额。可以使用字段将资金从 Guild 转移到您所选择的站。

用户选择适当的站并输入要转移的金额。单击“ 转帐 ”按钮后，交易即可执行。

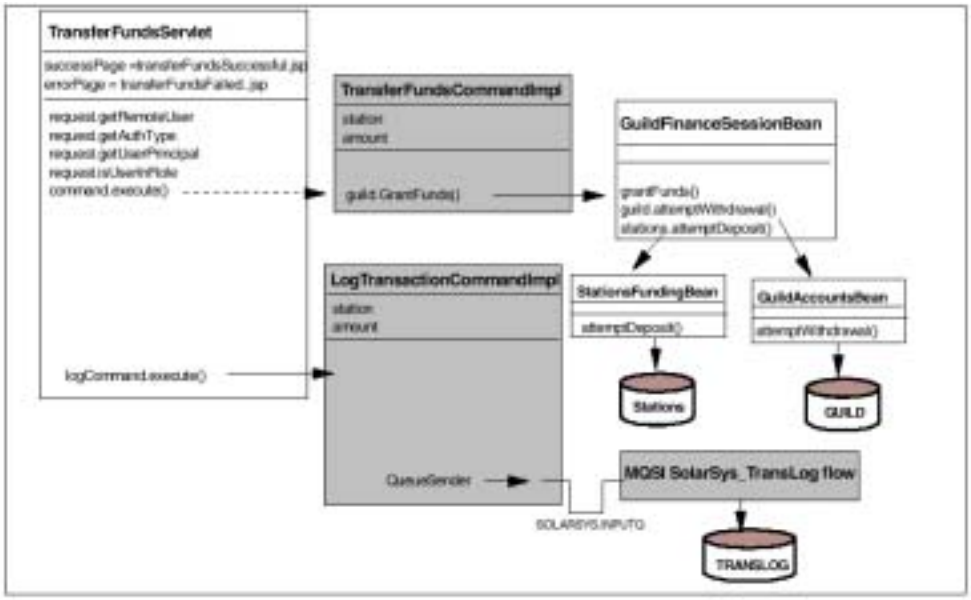

图6－13 转移资金流程

一旦用户请求要转移的资金，就会发生以下情况：

- *控制器：*TransferFundsServlet 代表交互控制器。

  首先，服务件从用户那里检索表格信息。然后，服务件从 CommandFactory 获取 TransferFundsCommand 实例，执行从 Guild 到所选站的资金转移。

  一旦执行该命令，服务件就从 CommandFactory 获得 LogTransactionCommand 实例，该实例记录这项交易。

  一旦记录交易，服务件就把请求转发到 transferFundsSuccessful JSP。

- *模型：*TransferFundsCommand 调用 GuildFinanceSessionBean 中的 guildGrantFunds（）方法。
  - *模型：*GuildFinanceSessionBean 调用两个方法。首先，它调用 GuildAccountsBean 实体 Bean 中的 attemptWithdrawal（）方法。然后，调用 StationsFundingBean 实体 Bean 中的 attemptDeposit（）方法。每个实体 Bean 针对其关联数据库执行请求的函数。

- 控制返回到 TransferFundsServlet。

- *控制器：*TransferFundsServlet 调用 LogTransactionCommand 把交易记录发送到交易日志数据库。以 XML 格式构造交易信息，并使用 JMS 把包含此信息的消息放在 MQSeries 队列 （SOLARSYS.INPUTQ）上。
  - MQSI 从该队列检索消息，并执行 SOLARSYS__TRANSLOG 消息流。此消息流在 TRANSLOG 数据库的 LOGENTRIES 表中为交易创建条目。SOLARSYS__TRANSLOG 消息流将在第 239 页的 14.2：“ 创建 SOLARSYS__TRANSLOG 消息流 ” 中讲述。
  - 控制返回到 TransferFundsServlet。
- *视图：*transferFundsSuccessful JSP 用于表示转帐成功。transferFundsFailed JSP 用于表示转帐失败。从这些显示中，用户可以单击链接，以便检索新的余额。单击此链接，使控制返回到 DisplayFundsServlet，DisplayFundsServlet 检索新的余额并显示它们。

6.6 应用程序技术

视图报表应用案例是对此应用程序中使用的所有技术很好的说明。我们将深入考察用于实施此应用案例的应用程序设计和编码技术。

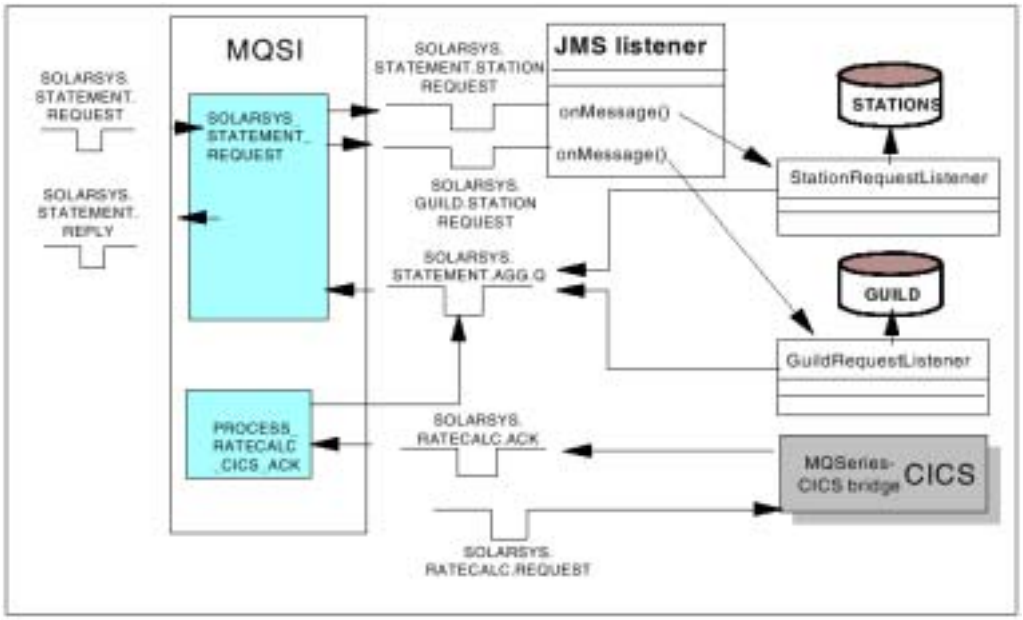

图6－14 发送和接受消息

6.6.1 视图报表应用案例概述

首先，让我们看一下结构，因为它与 MVC 模型相关。在我们的实例应用程序中，我们已经确定需要检索显示站余额和 Guild 余额的报表，以及检索执行的交易。我们通过向 MQSI 发送请求（把消息放在 MQSeries 队列上），可以使用命令来检索此数据。此命令将在本地执行（迄今为止，尚未使用命令传送）。

- *视图：*displayFundsRichResults JSP 包含一个链接，该链接使您可以查看包含各站余额以及交易记录的报表。当用户单击“查看报表”时，就会发生以下情况。
- *控制器：*ViewStatementServlet 调用 ViewStatementCommand 命令 Bean 处理请求。
- *模型：*ViewStatementCommand 创建一条包含请求信息的 XML 消息，并使用 JMS 把该消息放在 SOLARSYS.STATEMENT.REQUEST MQSeries 队列中。ViewStatementCommand 在 SOLARSYS.STATEMENT.REPLY 队列上等待回复。

- MQSI 获得该请求，把它分成三个单独的请求，并将这些请求发送到三个不同目标。
    - 对站余额的请求是在 SOLARSYS.STATEMENT.STATION.REQUEST 队列上发送的。此目标是 WebSphere Application Server，在这里，一个 JMS 侦听器检索此消息，并调用 StationRequestListener 消息 Bean 的 onMessage（）方法。该 StationRequestListener 是一个没有国籍的会话 Bean。它使用 JDBC 从 STATIONS 数据库检索请求的数据。响应在 SOLARSYS.STATEMENT.AGG.Q 上返回。

提示：消息 Bean 的角色应只是一种从队列获取消息的手段。 一旦删除，就可能有多种处理方法，但任何处理都不应由消息 Bean 进行。一旦消息 Bean 收到消息，就会把消息路由到知道如何处理消息的类。

我们的应用程序就是一个简单案例，其中 Bean 接收到的所有消息都以同样的方法进行处理。这很简单，因为该 Bean 知道每次需要传递到同样的类。该类与任何其他相关类共同对消息进行相应的处理。

如果 Bean 可以收到需要以不同方法处理的各种消息，事情就比较复杂。因为，Bean 必须知道如何把消息路由到不同的类。建议的解决方案是使 Bean 把消息交给某种 Factory，该 Factory 知道如何根据消息中的信息把消息路由到适当对象。

- 将对 Guild 余额的请求发送到 SOLARSYS.STATEMENT.GUILD.REQUEST 队列。此目标是同一 WebSphere Application Server。JMS 侦听器检索此消息，并调用 GuildRequestListener 消息 Bean 的 onMessage（）方法。JMS 侦听器使用 JDBC 从 GUILD 数据库检索请求数据。其响应在 SOLARSYS.STATEMENT.AGG.Q 上返回。
- 第三个请求在 SOLARSYS.RATECALC.REQUEST 队列上发送。目标是后端历史系统，在这里，CICS－MQ 桥把请求传递到 CICS。其响应在 SOLARSYS.RATECALC.ACK 队列上返回。当处理 CICS 时，在将消息发送回 SOLARSYS.STATEMENT.AGG.Q 之前，需要进行一些额外的处理工作。这将在第 269 页的 14.3.4 “ PROCESS__RATECALC__CICS__ACK ”中讲述。

- 然后，MQSI 流把响应聚集成一个回复，并将其发送回 SOLARSYS.STATEMENT.REPLY 队列上的 ViewStatementCommand。

- *视图：*在此应用程序中，视图由从回复直接发送到浏览器的 XML 构成。如果此数据最终发送到用户，这就不是有用的显示。但是，由于结果是 XML 文档，因而，我们可以利用几种技术为该用户编排数据格式。
  - 为与应用程序中的其他视图保持一致，我们可以分析该 XML，使用数据填充视图 Bean，并把数据发送到将要显示的 JSP。这是一种非常完美的解决方案，但是，这种解决方案的代价是：
    - 我们需要开发具有足够智能的对象，以便根据 XML 数据构造视图 Bean。
    - 我们需要花费额外的处理时间和资源。
  - 把数据从 XML 转换为视图的另一个选项是可扩充样式表语言（XSL）。实际上，XSL 是创建 XSL 样式表的语言。而这些样式表又可以把 XML 转换为另一种格式（例如，HTML）。

    XSL 不在本书讲述的范围之内。有关详细信息，请参阅位于以下地址的 XSL W3C 建议：http：／／www.w3.org／TR／xsl／，以及位于以下地址的 Xalan（一种 XSL 转换（XSLT）处理器）：http：／／xml.apache.org／xalan－j／index.html。

6.6.2 视图：DisplayFundsRichResults

此应用案例的应用程序流假定已执行“显示资金”应用案例流，以及可能的“转移资金”应用案例流。此时，用户将查看显示各站及 Guild 的资金状态的 DisplayFundsRichResults JSP。

“视图”和“控制器”之间的接口是通过使用视图 Bean （DisplayBalancesViewBean）实现的。视图 Bean 是把数据发送到 JSP 的手段。结果 Bean 是从命令返回数据的手段。在此情况下，该命令扮演结果 Bean 的角色，并在命令执行时自动填充数据。然后，服务件从该命令中删除数据，并水合（填充）视图 Bean，该视图 Bean 发送到视图。

实例 6－5：DisplayFundsRichResults.jsp

```
<!DOCTYPE HTML PUBLIC "－／／W3C／／DTD HTML 4.0 Transitional／／EN">
```

```
<%@page session="false"info="Patterns"errorPage="WEB－INF／error.jsp"%>
<%@page import="com.ibm.pdk.*"%>
<!-- This bean encapsulates the data from the back end -->
<!—此Bean压缩后端数据 －－>
<jsp    :    useBean    id="guildViewBean"type="com.ibm.pdk.view.DisplayBalancesViewBean"
scope="request"／>
```

用户可以单击窗口底部的“ 视图报表 ”链接以检索资金状况报表。单击此链接即可调用 ViewStatementServlet。

实例6－6：调用控制器

```
<center><a href="／PDK／ViewStatementServlet">View Statement<／a><／center>
```

6.6.3 控制器：ViewStatementServlet

用于应用程序的交互控制器是通过 ViewStatementServlet 实施的。该交互控制器调用使用命令 Bean 的商业逻辑。服务件为：

- 实例化命令
- 设置命令属性
- 调用命令的执行方法
- 检索命令的结果

实例6－7：ViewStatementServlet

```
  ／／Instantiate command (实例化命令)
  ViewStatementCommand command=


 (ViewStatementCommand) CommandFactory.instance () .getCommand (CommandFactory.VIEW__
STATEMENT__COMMAND) ;
  writeToLog ("Obtained ViewStatementCommand") ;

  ／／Set command target (设置命令目标)
  command.setCommandTarget   (  CommandTargetFactory.instance   (  )   .getCommandTarget
   (CommandTargetFactory.LOCAL__COMMAND__TARGET) ) ;

  ／／Set command properties (设置命令属性)
    command.setUserRole (role ) ;
    command.setStation (station ) ;
    ／／set interest rate info  (this should be put into a db somewhere)
     (设置利率信息<此信息应放在某个数据库中>)
    InterestRate rate =new InterestRate () ;
    rate.setCreditRating (0 ) ;
    rate.setDuration (48 ) ;
    rate.setType ("NEW") ;
    rate.setYear ("2001") ;
```

```
    command.setInterestRate (rate ) ;

    //execute the command (执行命令)
    command.execute () ;

    //send the results to the user (把结果发送给用户)
    writeToLog ("Command executed.Putting results to UI") ;
     response.getOutputStream () .println (command.getMessage () ) ;
     response.getOutputStream () .flush () ;

}catch (Throwable theException) {
    // uncomment the following line when unexpected exceptions (当意外情况发生时，解出以下
    代码行的注释)
    // are occurring to aid in debugging the problem. (有助于调试问题。)
    theException.printStackTrace () ;
}
```

6.6.4 使用命令调用商业逻辑

应用程序中的下一步是命令执行。其中包含三个步骤：

- 创建命令接口
- 执行命令
- 执行目标

创建命令接口：ViewStatementCommand

命令接口通过 TargetableCommand 接口而扩展，如果需要，还可通过 CompensableCommandInterface 扩展。在我们的实例中，ViewStatementCommand 说明最常见的实施方法，即实施用于可确定命令的命令接口。

实例6－8：ViewStatementCommand

```
package com.ibm.pdk.command;
import com.ibm.websphere.command.*;
import com.ibm.pdk.*;
public interface ViewStatementCommand extends TargetableCommand {
  public StationsBalanceList getBalances () ;
  public InterestRate getInterestRate () ;
  String getMessage () ;
  public String getStation () ;
  public TransactionMap getTransactions () ;
  public String getUserRole () ;
  public void setBalances (StationsBalanceList balanceList) ;
  public void setInterestRate (InterestRate rate) ;
  public void setStation (String station) ;
```

```
  public void setUserRole (String role) ;
}
```

显而易见，ViewStatementCommand 定义多个命令特定的方法。

此外，TargetableCommand 接口说明如表 6－1 显示的方法。

表 6－1：TargetableCommand 方法

方法	目的
setCommandTarget（）	指定命令的目标对象。
getCommandTarget（）	返回命令的目标对象。
setCommandTargetName（）	指定命令的目标名称。
getCommandTargetName（）	返回命令的目标名称。
hasOutputProperties（）	表明命令是否具有必须复制回客户机的输出。实施类还提供 setHasOutputProperties（）方法，用于设置此方法的输出。在默认情况下，hasOutputProperties（）返回“真”值。
setOutputProperties（）	保存来自命令的输出值，以便返回到特定客户机。
performExecute（）	封装应用程序特定的工作，并由命令接口中声明的 execute（）方法调用。

应用程序开发人员必须实施的唯一方法即 performExecute（）方法。TargetableCommandImpl 类实施其余方法，以及命令接口中声明的 execute（）方法。

执行命令：ViewStatementCommandImpl

如果要传送某个命令，则必须通过扩展 TargetableCommandImpl 类来实现 TargetableCommand 类。该类实施 TargetableCommand 接口中的所有方法，但 performExecute（）方法除外。此方法必须由应用程序开发人员编写。它还实施来自命令接口的 execute（）方法。

继续我们的实例，先看命令执行 ViewStatementCommandImpl 的责任。

定义实例和类变量

ViewStatementCommandImpl 类声明方法在类中使用的变量。

实例6－9：定义实例和类变量

```
public class ViewStatementCommandImpl extends TargetableCommandImpl implements
ViewStatementCommand {
  private StationsBalanceList balances;
  private com.ibm.pdk.InterestRate interestRate;
  private java.lang.String station;
  private java.lang.String userRole;
  private com.ibm.pdk.TransactionMap transactions;
  private java.lang.String message;
/**
  *ViewStatementCommandImpl constructor comment.
  */
public ViewStatementCommandImpl () {
  super () ;
}
```

实施命令特定的方法

ViewStatementCommandImpl 实施命令特定的方法。

实例6－10：实施命令特定的方法

```
public StationsBalanceList getBalances () {
  return balances;}
public com.ibm.pdk.InterestRate getInterestRate () {
  return interestRate;}
public java.lang.String getMessage () {
  return message;}
public java.lang.String getStation () {
  return station;}
public com.ibm.pdk.TransactionMap getTransactions () {
    if (transactions ==null ) {
    setTransactions (new TransactionMap () ) ;}
    return transactions;}
public java.lang.String getUserRole () {
  return userRole;}
public void setBalances (StationsBalanceList newBalances) {
  balances =newBalances;}
public void setInterestRate (com.ibm.pdk.InterestRate newInterestRate) {
  interestRate =newInterestRate;}
public void setStation (java.lang.String newStation) {
  station =newStation;}
public void setUserRole (java.lang.String newUserRole) {
  userRole =newUserRole;}
```

实施命令接口方法

必须实施来自命令接口的两个方法：isReadyToCallExecute（）和 reset（）。

实例 6－11：实施来自命令接口的方法

```
public boolean isReadyToCallExecute () {
  return true;}


public void reset () {
  setBalances (null) ;
  setInterestRate (null ) ;
  setStation (null ) ;
  setTransactions (null ) ;
  setUserRole (null) ;}
```

实施 TargetableCommand 接口方法

需要实施一个 TargetableCommand 方法：performExecute（）。覆盖 setOutputProperties（）的默认实施也可能是适当的，因为 setOutputProperties（）不保存最终的、临时的或静态的字段。

实例 6－12：实施来自 TargetableCommand 接口的 performExecute ()

```
public void performExecute () throws Exception {

  Hashtable environment =new Hashtable () ;
  envienvronment.put (
      Context.PROVIDER__URL, PdkProperties.singleton () .getProviderURL () ) ;
   environment.put (
      Context.INITIAL__CONTEXT__FACTORY,
      PdkProperties.singleton () .getInitialContextFactoryClassName () ) ;

    Moore code here to send and receive message from MQSeries
    这里有更多需要从MQSeries发送和接收的消息

    writeToLog ("Message received from reply q![message="+replyMessage+"]") ;
     setMessage (replyMessage.getText () ) ;

    writeToLog ("Closing") ;
    qCon.close () ;
}
```

执行目标：LocalCommandTarget

作为 TargetableCommand 目标的对象必须实施 CommandTarget 接口。此对象可以是实际服务器端对象，如实体 Bean，也可以是用于某个服务器的客户端适配器。CommandTarget 接口的实施者负责确保在所需目标服务器环境中正确执行命令。

在实例中，我们实施 CommandTarget 接口作为本地目标，该接口在客户机的 JVM 中运行。要接受命令，该接口必须实施 CommandTarget 接口的单一方法：executeCommand（）方法。

实例 6－13：LocalCommandTarget

```
public class LocalCommandTarget implements CommandTarget, Serializable {
public TargetableCommand executeCommand (TargetableCommand command) throws CommandException
{

  try {
    command.performExecute () ;
    writeToLog ("Command executed") ;
    return command;
  }catch  (CommandException ce) {
    ce.printStackTrace () ;
    throw ce;
  }catch (Exception e) {
    e.printStackTrace () ;
    throw new CommandException (e) ;
  }
}
```

6.6.5 将消息放在队列上和检索消息

该应用程序需要从 MQSeries 队列发送和接收消息。我们的实施假定 MQSeries 以本地方式安装在 WebSphere 主机上。由于本地 MQSeries 队列管理器是群集的一部分，因此其承担了确定目标队列准确位置的任务。它还使我们可以利用 MQSeries 工作负载分配以及确定的交付功能。

如前所述，消息放在来自命令 Bean － ViewStatementCommandImpl－ 的 MQSeries 队列上。现在，我们看看实际把消息放在这个队列上的 performExecute（）方法中的代码。

performExecute（）方法负责创建 XML 消息，把它放在队列上，并等待响应，该响应包含要向用户显示的数据。此方法使用以下输入属性：

实例6－14：输入属性

```
provider.url=iiop：／／localhost
initial.context.factory=com.ibm.websphere.naming.
WsnInitialContextFactory
input.Queue.name=SOLARSYS.INPUTQ
qcf.jndi.name=gwmQCF
txn.record.xml.tag.name=transactionrecord
txn.date.xml.tag.name=transdate
station.xml.tag.name=station amount.xml.tag.name=amount
```

初始化

实例 6－15：建立与 MQSeries 的连接

```
  ／／Getting initial context（获取初始上下文）
  InitialDirContext context =null；
  context =new InitialDirContext (environment) ；

  String qcfJndiName =
    PdkProperties.singleton () .getQueueConnectionFactoryJNDIName () ；
   writeToLog ("Get QCF from JNDI[name="+qcfJndiName +"]") ；
   QueueConnectionFactory qConFactory =
       (QueueConnectionFactory) context.lookup (qcfJndiName) ；
```

建立连接

第一步是创建基本传输 MQSeries 的连接。该连接为临时队列提供范围和位置，在该位置可保存控制如何连接到 MQSeries 的参数（例如，队列管理器名称，以及使用 MQSeries Java 客户机连接性的远程主机的名称）。

对于点到点情形，连接是使用 QueueConnection 建立的。

实例6－16：建立与 MQSeries 的连接

```
  String qcfJndiName =
    PdkProperties.singleton () .getQueueConnectionFactoryJNDIName () ；
   writeToLog ("Get QCoF from JNDI[name="+qcfJndiName +"]") ；
   QueueConnectionFactory qConFactory =
     (QueueConnectionFactory) context.lookup (qcfJndiName) ；

  ／／Getting queue connection（建立队列连接）
  QueueConnection qCon =qConFactory.createQueueConnection () ；
```

建立会话

会话提供产生和使用消息的上下文，其中包括用于创建 MessageProducer 和 MessageConsumer 的方法。会话还包含 HCONN（队列管理器的句柄），因此可定义交易范围。

至于我们的点到点连接，会话是使用 QueueSession 建立的。

实例 6－17：建立会话

```
// Get queue session（获取队列会话）
QueueSession qSession=qCon.createQueueSession（false，Session.AUTO__ACKNOWLEDGE）；
```

发送消息

MessageProducer 用来发送消息。它包含对象句柄（HOBJ），其可定义用于写入和读取的特定队列。对于点到点连接，这是通过 QueueSender 实现的。

下面的代码创建要发送的 XML 消息，并把该消息放在队列上。

实例 6－18：创建消息并把消息放在队列上

```
String requestQName =PdkProperties.singleton（）.getRequestQueueJNDIName（）；
String replyQName =PdkProperties.singleton（）.getReplyQueueJNDIName（）；
writeToLog（"Getting queues from JNDI namespace[requestJNDIName="
+requestQName +"][replyJNDIName="+replyQName +"]"）；
Queue requestQueue =（Queue）context.lookup（requestQName ）；
Queue replyQueue =（Queue）context.lookup（replyQName）；

writeToLog（"Building queue sender."）；
QueueSender qSender =qSession.createSender（requestQueue）；
／／Creat message（创建消息）
TextMessage requestMessage =qSession.createTextMessage（）；
String message =getRequestString（）；
requestMessage.setText（message）；
requestMessage.setJMSReplyTo（replyQueue）；
requestMessage.setJMSCorrelationID（"BobLuvsGoats"）；
writeToLog（"Message built[xml="+message +"]"）；
／／Send message（发送消息）
qSender.send（requestMessage）；
//The reply＇s Correlation ID should match this MessageID - so capture it
（回复的"Correlation ID"应匹配该 MessageID，这样就可对其进行捕获）
String messageId =requestMessage.getJMSMessageID（）； writeToLog（"Correlation ID of sent
message ="+messageId）；
```

检索消息

MessageConsumer 用来接收消息。就像 MessageProducer 一样，MessageConsumer 包含一个对象句柄 （HOBJ），该对象句柄定义用于写入和读取的特定队列。对于点到点连接，这是通过 QueueReceiver 实现的。

实例6－19：检索回复

```
  //flip the connection to receive mode／／把连接转换为“接收”模式
  qCon.start () ;

  ／／用一个msg选择器设置q接收器，以便把请求连接到此msg
  String selector ="JMSCorrelationID =\'"+messageId +"\'";
  QueueReceiver qReceiver =qSession.createReceiver (replyQueue, selector) ;
  writeToLog ("QueueReceiver built －msg selector[selector="+selector+"]") ;

  long timeout =PdkProperties.singleton () .getReplyTimeout () ;
  writeToLog ("Waiting for reply[timeout="+timeout +"]") ;
  TextMessage replyMessage = (TextMessage) qReceiver.receive (timeout) ;

  if  (replyMessage ==null) {
    writeToLog ("ERROR：No reply message received!") ;
    setMessage ("No message received") ;
    return;
  }
  writeToLog ("Message received from reply q![message="+replyMessage +"]") ;
  setMessage (replyMessage.getText () ) ;

  writeToLog ("Closing") ;
  qCon.close () ;
}
```

提示：

- 在规定时间，每个 HCONN 只能进行一项操作，因此，不能同时调用与 Session 关联的 MessageProducer 或 essageConsumer。这与 JMS 限制相一致，即每个 Session 一个线程。
- PUT 可以使用远程队列，而 GET 则只适用于本地队列管理器上的队列。一般 JMS 接口细分为多个用于点到点和发布／预订行为的具体版本。
- Connection 是线程安全的（thread safe），但 Session、MessageProducer 和 MessageConsumers 则不是。建议策略是每个应用程序线程使用一个 Session。